Chapter 1

Introduction

1.1 INTRODUCTION

For a long time, the Data Encryption Standard (DES) was considered as a standard for the symmetric key encryption. DES has a key length of 56 bits. However, this key length is currently considered small and can easily be broken. For this reason, the National Institute of Standards and Technology (NIST) opened a formal call for algorithms in September 1997[2]. A group of fifteen AES candidate algorithms were announced in August 1998. Next, all algorithms were subject to assessment process performed by various groups of cryptographic researchers all over the world.On October 2, 2000, NIST announced that the Rijndael algorithm was the winner. Rijndael can be specified with key and block sizes in any multiple of 32 bits, with a minimum of 128 bits and a maximum of 256 bits. Therefore, the problem of breaking the key becomes more difficult[1] .In cryptography, the AES is also known as Rijndael . AES has a fixed block size of 128 bits and a key size of 128, 192 or 256 bits.

1.2 NEED FOR DATA ENCRYPTION

Today, most laptops and PCs have some sort of antivirus and personal firewall software to prevent data hijacking. But what happens when a computer is stolen or when an overtired road warrior leaves her PDA in a cab? Headlines from any newspaper or news Web site around the world put data security vulnerabilities due to physical loss of devices into perspective. In the United States (U.S.), the Transportation Security Administration (TSA) reported that a stolen computer exposed more than 100,000 personal records. In the United Kingdom, a laptop storing personal data on 11,000 children was stolen from a Nottinghamshire hospital. Finally, the 2006 asset audit of the New Zealand Inland Revenue Department (IRD) showed that the IRD has no clue as to the whereabouts of 106 of its computers or their contents. The list goes on and on. A 2006 global study by market research firm Gartner indicates that while 25 percent of information theft is linked to network intrusion, 60 percent of data breaches can be attributed to lost or stolen mobile devices. With this in mind, it is critical for organizations to bolster defenses by encrypting data across the board.

Chapter 2

AES Encryption Algorithm

2.1 THE STATE REPRESENTATION

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State. The State consists of four rows of bytes, each containing Nb bytes, where Nb is the block length divided by 32. In the State array denoted by the symbol s, each individual byte has two indices, with its row number r in the range 0 < r < 4 and its column number c in the range 0 <c < Nb. This allows an individual byte of the State to be referred to as either sr,c or s[r,c]. For this standard, Nb=4, i.e., 0 < c < 4. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes out 0, out1, … out 15.

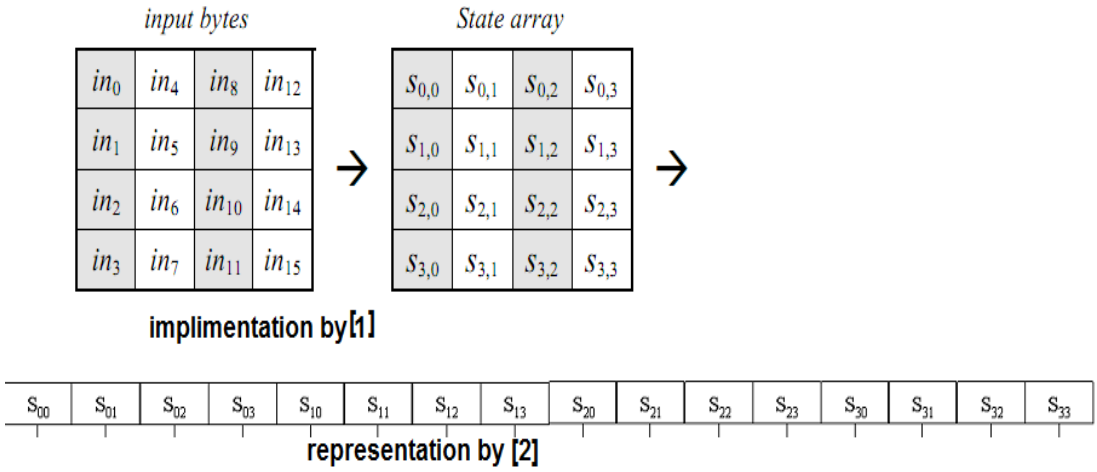


Fig 2.1: State array input and output

2.2 THE CIPHER

At the start of the Cipher, the input is copied to the State array using the conventions described in Sec.2.1. After an initial Round Key addition, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first ***Nr*** -1 rounds. The final State is then copied to the output.

Pseudo code:

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])

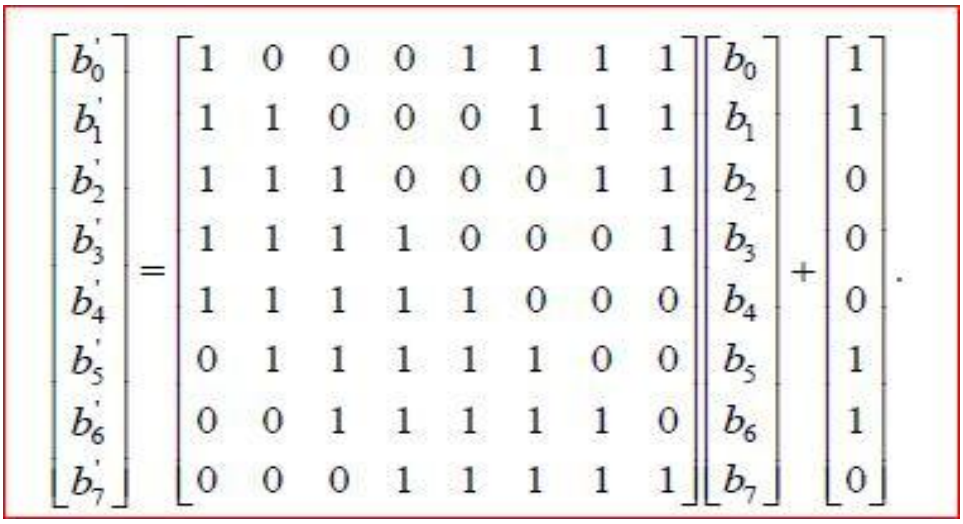
for round = 1 step 1 to Nr–1 SubBytes(state)
ShiftRows(state)
MixColumns(state)
AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
endfor
SubBytes(state) ShiftRows(state)
AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

out = state end

2.2.1 The SubByte() Transform

The **SubBytes()** transformation is a non-linear byte substitution that operates independently
on each byte of the State using a substitution table (S-box). This S-box (Fig. 2.3), which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field $GF(2^8)$, (described in Sec. 4.2[2]); the element {00} is mapped to itself.
2. Apply the following affine transformation (over GF(2)):



$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Fig 2.2: Matrix For Subbyte

for 0 £ *i* < 8, where *bi* is the i^{th} bit of the byte, and *ci* is the i^{th} bit of a byte *c* with the value {63} or {01100011}. Here and elsewhere, a prime on a variable (e.g., *b*¢) indicates that the variable is to be updated with the value on the right.

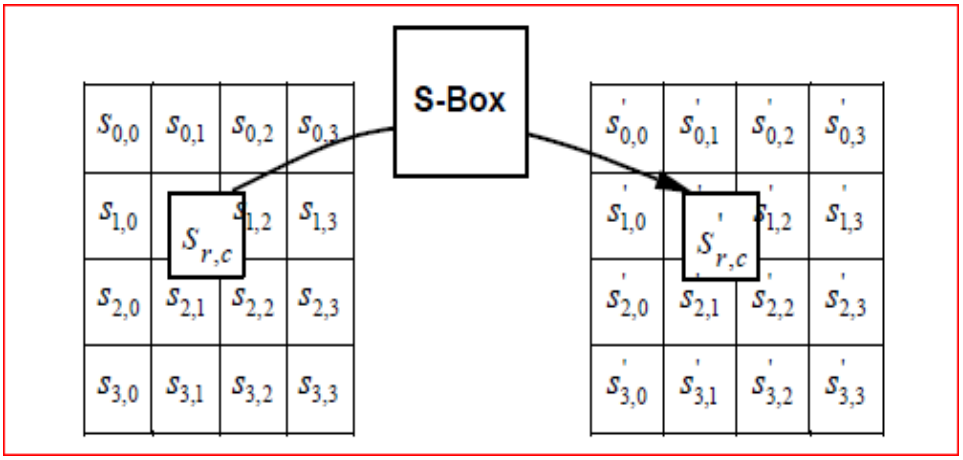


Fig2.3:S-BOX

The S-box used in the **SubBytes()** transformation is presented in hexadecimal form in Table. 2.1. .

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
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

Table 2.1: S-box: substitution values for the byte xy(in hexadecimal format)

2.2.2 The ShiftRow() Transform

In the **ShiftRows()** transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted. Specifically, the **ShiftRows()** transformation proceeds as follows:

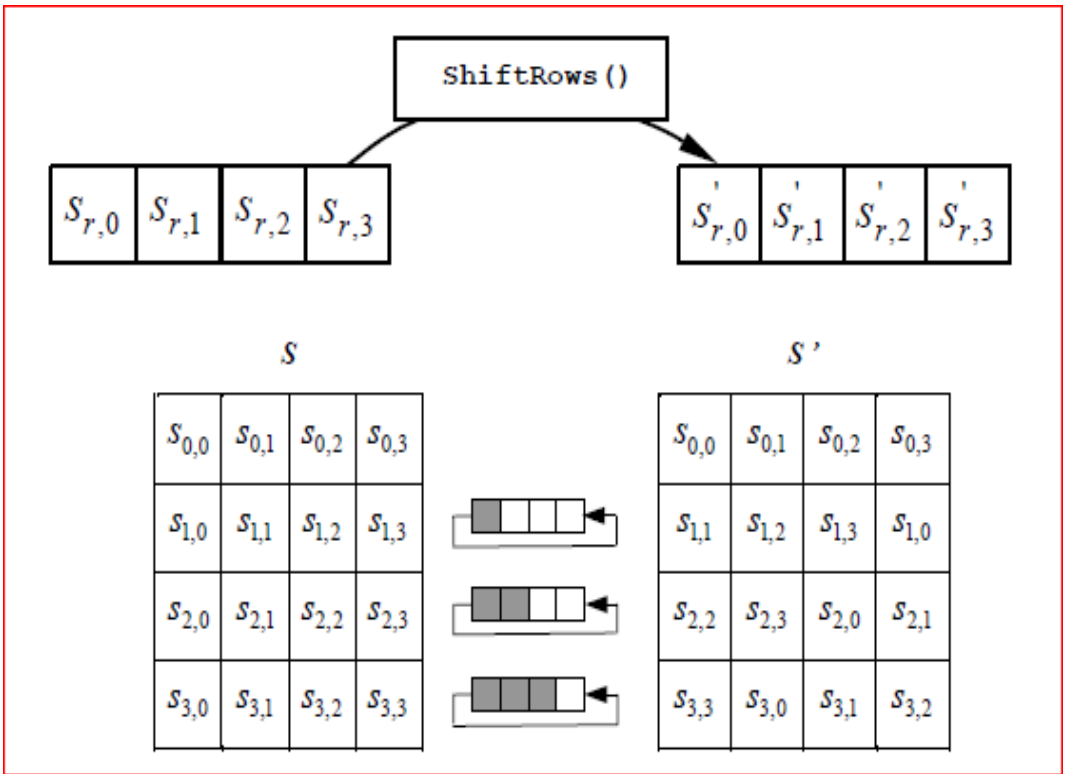


Figure 2.4 . ShiftRows()cyclically shifts the last three rows in the State

2.2.3 The mixcolumn() Transform

The **MixColumns()** transformation operates on the State column-by-column, treating each column as a four-term polynomial[2].The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

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

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

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

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

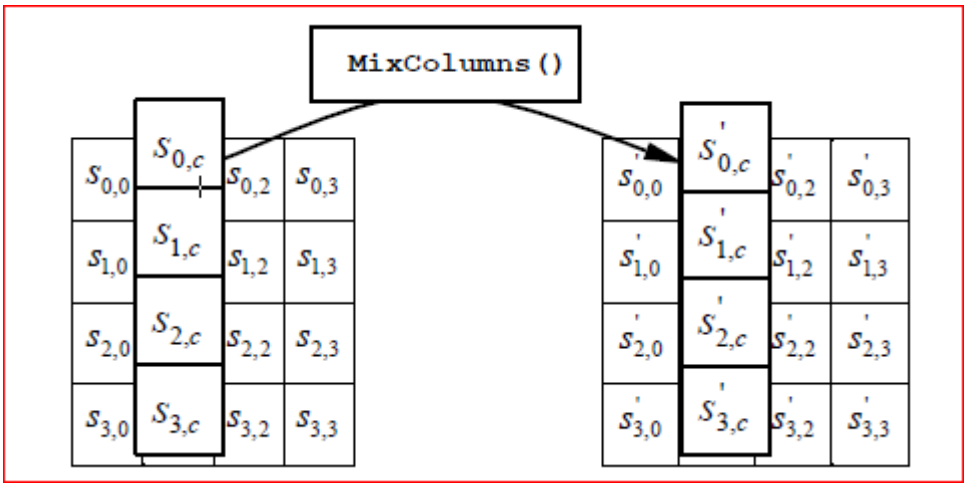


Figure 2.5. MixColumns()operates on the State column-by-column.

2.2.4 AddRoundkey() Transform

In the Add Round key transformation. A Round Key is added to the State resulted from the operation of the Mix Column. This function is a simple bit-wise XOR of the key and the generated state. So it has been directly implemented in the main AES module.

2.2.5 Key Expansion

The AES algorithm takes the Cipher Key, ***K***, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of ***Nb*** (***Nr*** + 1) words: the algorithm requires an initial set of ***Nb*** words, and each of the ***Nr*** rounds requires ***Nb*** words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted [*wi*], with *i* in the range 0 < *i* < ***Nb***(***Nr*** + 1).

SubWord() is a function that takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word. The function **RotWord()** takes a word [*a*0,*a*1,*a*2,*a*3] as input, performs a cyclic permutation, and returns the word [*a*1,*a*2,*a*3,*a*0]. The round constant word array, `Rcon[i]`, contains the values given by [x^{i-1},{00},{00},{00}], with x^{i-1} being powers of *x* (*x* is denoted as {02}) in the field GF(2^8) [2].

It can be seen that the first ***Nk*** words of the expanded key are filled with the Cipher Key. Every following word, **w**[[**i**]], is equal to the XOR of the previous word, **w**[[**i-1**]], and the word ***Nk*** positions earlier, **w**[[**i-*Nk***]]. For words in positions that are a multiple of ***Nk***, a transformation is applied to **w**[[**i-1**]] prior to the XOR, followed by an XOR with a round

constant, **Rcon[i]**. This transformation consists of a cyclic shift of the bytes in a word (**RotWord()**), followed by the application of a table lookup to all four bytes of the word (**SubWord()**).

```
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
```

2.3 INVERSE CIPHER

The Cipher transformations can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher -**InvShiftRows()**, **InvSubBytes()**,**InvMixColumns()**, and **AddRoundKey()** – process the State and are described in the following subsections [2].

```
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])

        for round = Nr-1 step -1 downto 1
         InvShiftRows(state)
         InvSubBytes(state)
         AddRoundKey(state,w[round*Nb,(round+1)*Nb-1])
         InvMixColumns(state)
        end for

        InvShiftRows(state)

         InvSubBytes(state)

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

out = state
end
```

2.3.1 InvShiftRows() Transformation

InvShiftRows() is the inverse of the **ShiftRows()** transformation. The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted. The bottom three rows are cyclically shifted by ***Nb*** *-shift*(*r, Nb*) bytes, where the shift value *shift(r,Nb)* depends on the row number.

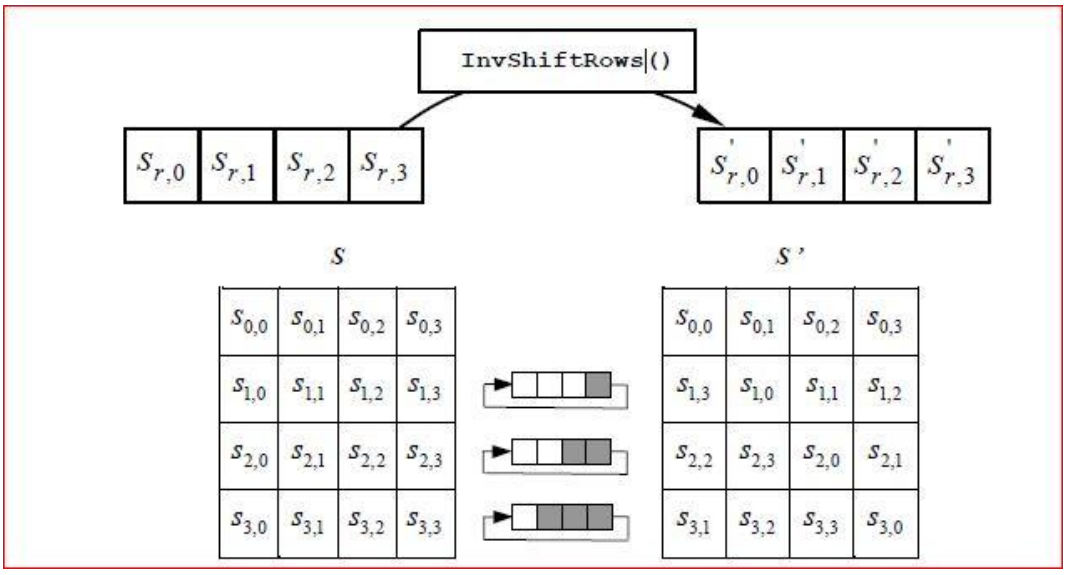


Figure 2.6.. InvShiftRows()cyclically shifts the last three rows in the State.

2.3.2 InvSubBytes() Transformation

InvSubBytes() is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation followed by taking the multiplicative inverse in $GF(2^8)$.

The inverse S-box used in the **InvSubBytes()**transformation is presented in Table 2.1.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
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

Table2.2: Inverse S-box: substitution values for the byte xy(in hexadecimal format).

2.3.3 InvMixColumns() Transformation

InvMixColumns() is the inverse of the **MixColumns()** transformation. **InvMixColumns()** operates on the State column-by-column, treating each column as a four-term polynomial[2] . The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by

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

As described in earlier, this can be written as a matrix multiplication. Let

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < Nb.$$

Fig 2.7: Matrix Multiplication

As a result of this multiplication, the four bytes in a column are replaced by the following:

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

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

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

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

Fig 2.8: Equation for InvMix Column

2.3.4 Inverse of the AddRoundKey() Transformation

AddRoundKey(), which was described in earlier, is its own inverse, since it only involves an application of the XOR operation.

Chapter 3

Implementation Issues

3.1 ARCHITECTURE OF BASIC COMPONENTS

The overall architecture of the AES hardware mirrors the structure of Algorithm discussed in 2.2 . It is a synchronous implementation of both the processes of cipher. It uses 2 128-registers.Every clock transition, these registers are loaded, except dataout1 and dataout2, which is loaded when an input state is completely ciphered. During the encryption process, Register0 is loaded with the input data or the partially encrypted text with the result of the **mixcolumn** and **AddRoundKey** component except first and last round in which mixcolumn operation is skipped , Register2 with the state after applying functions SubBytes and subsequently ShiftRows..The component that implements function AddRoundKey is simply a net of XOR gates that adds in GF(2^8) the key schedule to the current state. The component implementing function SubBytes uses 16 S-boxes stored in a Read-Only Memory (ROM). The obtained state is row-shifted before its storage in Register2. The component architecture is given in Fig. 3.1.

Function MixColumns is implemented by a massively parallel component that computes all the bytes of the new state in a single clock. It uses four components of the same architecture. This basic component produces one column os the new state. Its architecture is described in Fig. 3.1, wherein component mult yields the a special product of a given byte from the state times{01}, {02}or{03}.The architecture of component multi presented in Fig. 3.2.Component xtime computes the xtime operation as defined in[3] and shown in Fig. 3.3.

Equivalent invMixColumns can be implemented in the same way as MixColumns, , wherein component invmult yields the a special product of a given byte from the state times{0e}, {0b},{0d} or {09}.

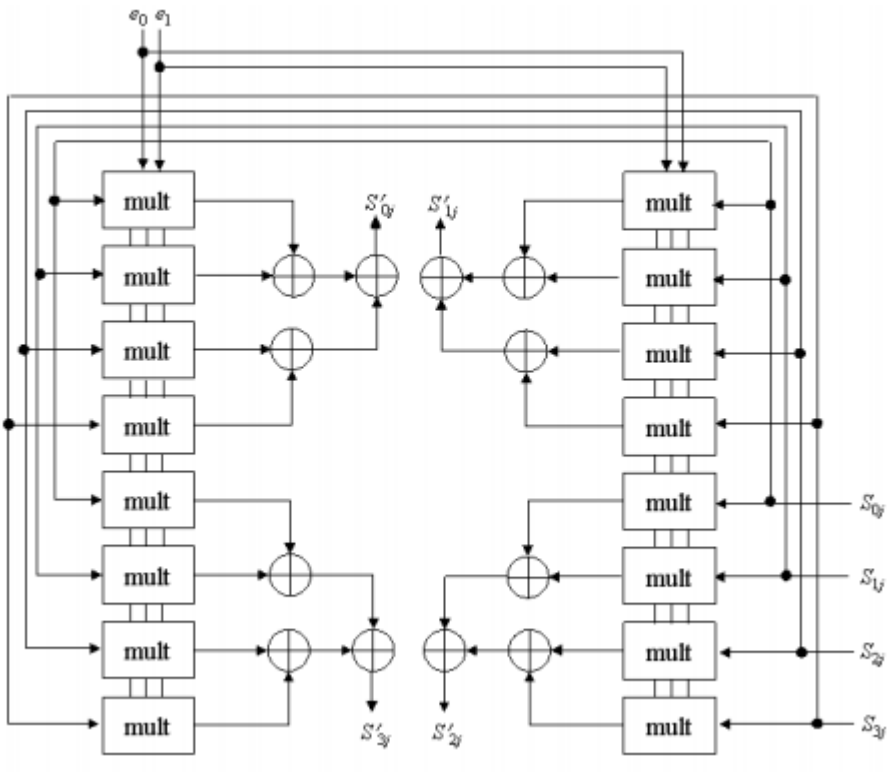


Fig. 3.1: Basic component inMixColumnscomponent

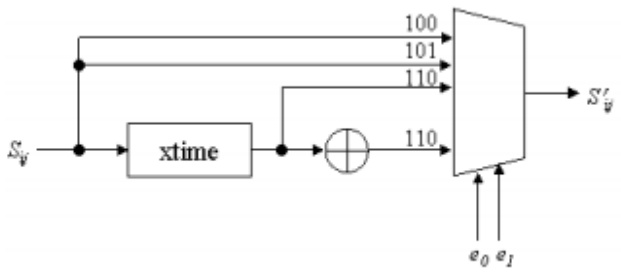


Fig. 3.2:. Architecture of the multcomponent

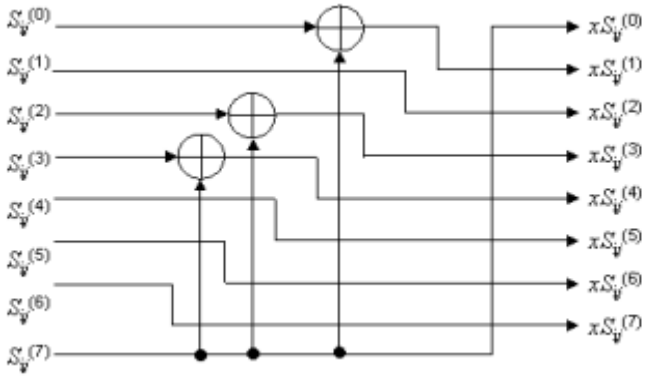


Fig.3.3:. Description of operation xtime

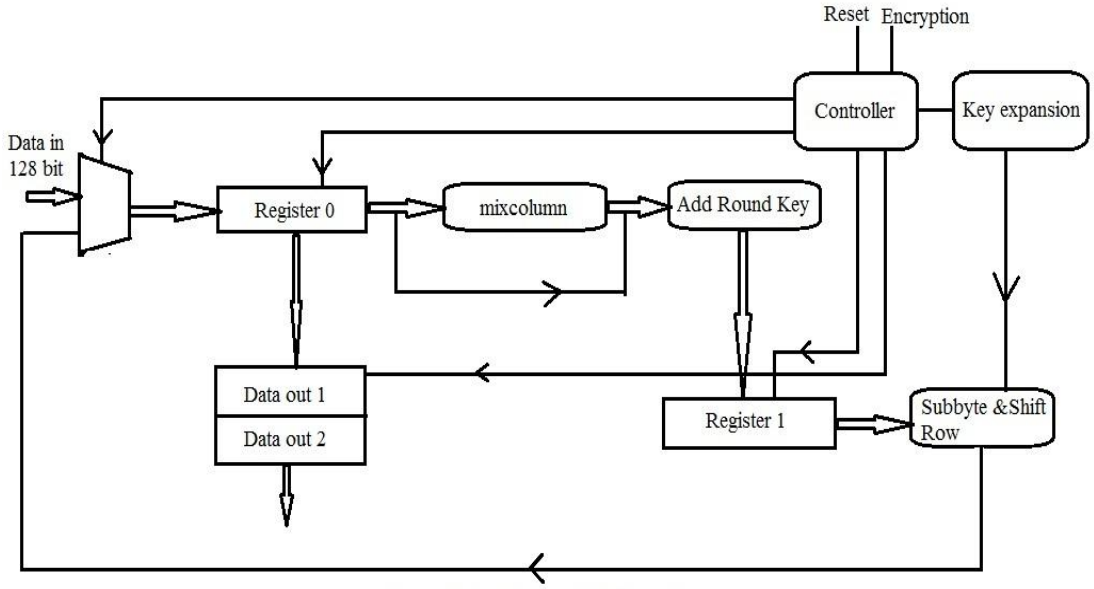
3.2 PIPELINING

2 stage pipelined architecture is shown in fig.3.4 it two 128 bit data block in first two clock cycles . There were two possible ways of pipelining we decided upon. In one case, there is a possibility of using one stage of pipelining for each of the 11 stages of the encryption process, thereby increasing the throughput. The other is the one described in [3]. A 11 stage pipeline would obviously have a 3 to 4 times increase in throughput over a 3-stage one: but it requires 11 128-bit registors, thereby taking up too much space and a possibility of overflow of space in the FPGA. So we stuck with the implementation of the 2 stage pipeling. The following are the design issues we faced:

- Encryption routine and key expansion routine run parallelly in our implementation. Each round key remain valid for two clock cycles to operate on two 128 bit data blocks.
- In first two clock cycles data is input into the cipher by asserting the control signal rw 10 .There are 10 rounds So after 22 clock cycles output data1 is ready . on the 22th clock cycle control signal rw is asserted 01 and data is read from the memory. It is maintained for two clock cycles.so in 23 clock cycles 2 outpus leading to a throughput of

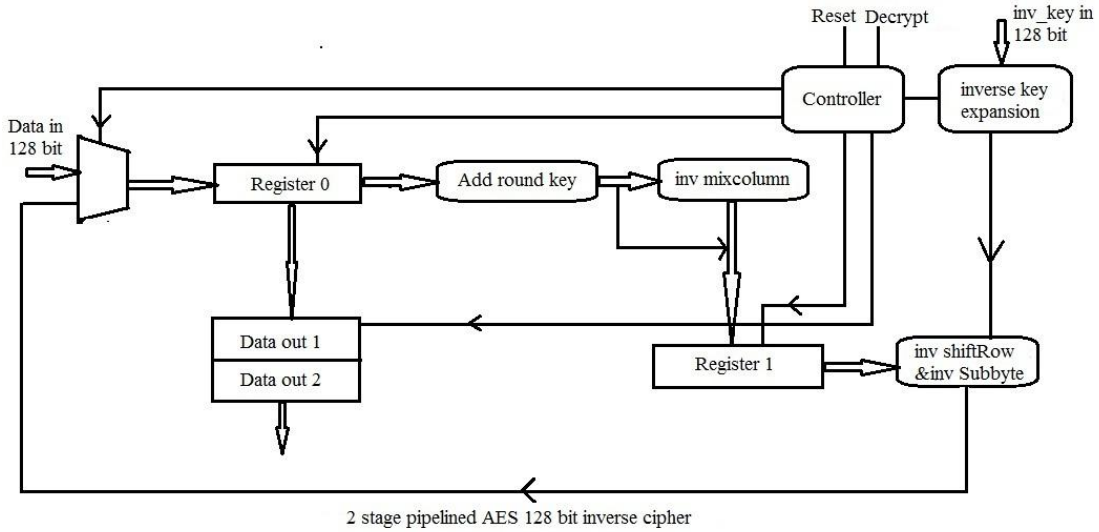
  T.p=(2*128)/(23*clock cycles)

The inverse cipher operate in the same way with operations are performed in reverse order.



2 stage pipelined AES 128 bit cipher block diagram

Fig 3.4



2 stage pipelined AES 128 bit inverse cipher

Fig3.5

3.3 KEY EXPANSION AND INVKEY EXPANSION ROUTINE

As only 128-bits are used at each stage in the encryption process, the key schedule for each stage can be generated dynamically. The implemetation is shown in Figure3.6. With each clock cycle, the previous value is loaded on the register. As is evident from our pipelined design, the key for stage i has to be held for two continuous clock cycles (as there are two stages in the pipeline–the same key acts on two different inputs in two clock cycles), and we employed a counter to achievethe same.

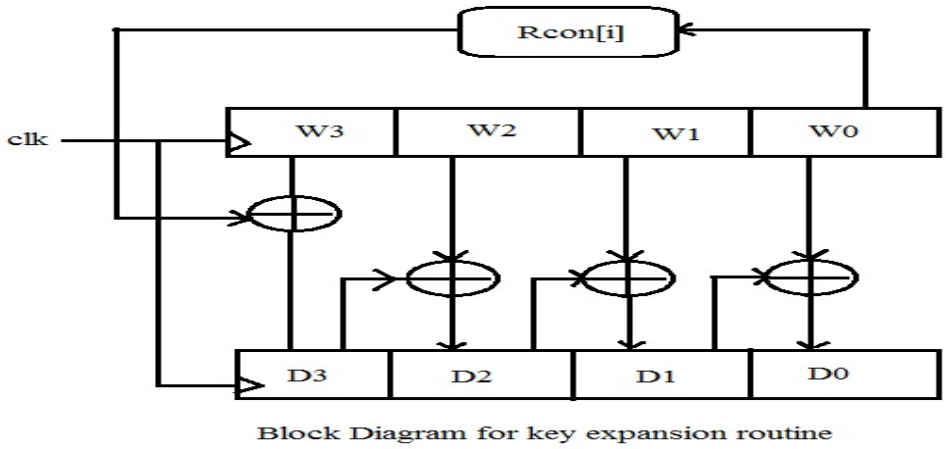


Fig 3.6

The key expansion routine for the inverse cipher can be generated by the method as shown in the following fig3.7

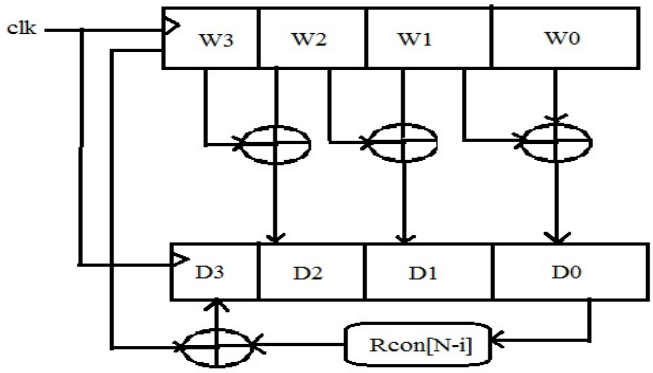


Fig3.7: Block diagram for inverse key expansion

3.4 MEMORY AND INPUT OUTPUT

The data is entered from the keyboard[4] and encrypted/decrypted data is displayed on the alphanumeric LCD on the STARTEN-3E FPGA board.Details of interfacing can be found in Appendix-B.

3.5 KEY LENGTH REQUIREMENTS

An implementation of the AES algorithm shall support at least one of the three key of lengths 128, 192, or 256 bits (i.e., Nk = 4, 6, or 8, respectively). Implementations may optionally support two or three key lengths, which may promote the interoperability of algorithm implementations[2].

Chapter 4

Conclusion

4.1 CONCLUSION

We implemented the hardware described throughout this paper using reconfigurable hardware. The FPGA family used is SPARTEN-3E. The architecture allows one to perform the core computation of the algorithm is a pipelined manner. The throughput of the cryptographic hardware is more than 550Mbits per seconds. The pipelined execution of the AES algorithm allows an increase of the number of rounds without much loss of efficiency. Increasing the number of rounds applied, improves the resistance of the AES algorithm to cryptanalysis attacks. Recall that the resistance of AES-based encryption against cryptanalysis attacks depends entirely on the number of rounds used. The pipelined implementation we propose throughout this report can be easily adapted to a higher round number and this can be done without much loss in efficiency. To be able to increase the number of round, component *KeyExpansion* needs to generate more key schedules and therefore the delay introduced by it increases with the number of rounds.

4.2 SCOPE FOR FUTURE IMPROVEMENTS

Throughput can be increased with increasing the number of pipelined stages. This can be done at cost of hardware. Besides this increasing the number of rounds makes it more secure.

Appendix A

A.1 Expansion of a 128-bit Cipher Key

This section contains the key expansion of the following cipher key:

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

for ***Nk*** = 4, which results in $w0$ = **2b7e1516** $w1$ = **28aed2a6** $w2$ = **abf71588** $w3$ = **09cf4f3c.**

i (dec)	temp	After RotWord()	After SubWord()	Rcon[i/Nk]	After XOR with Rcon	w[i-Nk]	w[i]= temp XOR w[i-Nk]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc

24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

Table A.1:Expansion of 128 bit cipher key

Appendix A.2 – Cipher Example

The following diagram shows the values in the State array as the Cipher progresses for a block length and a Cipher Key length of 16 bytes each (i.e., *Nb* = 4 and *Nk* = 4).

Input = 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c.

The Round Key values are taken from the Key Expansion example in Appendix A.1.

Round Number	Start of Round	After SubBytes	After ShiftRows	After MixColumns		Round Key Value	
input	32 88 31 e0 43 5a 31 37 f6 30 98 07 a8 8d a2 34				⊕	2b 28 ab 09 7e ae f7 cf 15 d2 15 4f 16 a6 88 3c	=
1	19 a0 9a e9 3d f4 c6 f8 e3 e2 8d 48 be 2b 2a 08	d4 e0 b8 1e 27 bf b4 41 11 98 5d 52 ae f1 e5 30	d4 e0 b8 1e bf b4 41 27 5d 52 11 98 30 ae f1 e5	04 e0 48 28 66 cb f8 06 81 19 d3 26 e5 9a 7a 4c	⊕	a0 88 23 2a fa 54 a3 6c fe 2c 39 76 17 b1 39 05	=
2	a4 68 6b 02 9c 9f 5b 6a 7f 35 ea 50 f2 2b 43 49	49 45 7f 77 de db 39 02 d2 96 87 53 89 f1 1a 3b	49 45 7f 77 db 39 02 de 87 53 d2 96 3b 89 f1 1a	58 1b db 1b 4d 4b e7 6b ca 5a ca b0 f1 ac a8 e5	⊕	f2 7a 59 73 c2 96 35 59 95 b9 80 f6 f2 43 7a 7f	=
3	aa 61 82 68 8f dd d2 32 5f e3 4a 46 03 ef d2 9a	ac ef 13 45 73 c1 b5 23 cf 11 d6 5a 7b df b5 b8	ac ef 13 45 c1 b5 23 73 d6 5a cf 11 b8 7b df b5	75 20 53 bb ec 0b c0 25 09 63 cf d0 93 33 7c dc	⊕	3d 47 1e 6d 80 16 23 7a 47 fe 7e 88 7d 3e 44 3b	=
4	48 67 4d d6 6c 1d e3 5f 4e 9d b1 58 ee 0d 38 e7	52 85 e3 f6 50 a4 11 cf 2f 5e c8 6a 28 d7 07 94	52 85 e3 f6 a4 11 cf 50 c8 6a 2f 5e 94 28 d7 07	0f 60 6f 5e d6 31 c0 b3 da 38 10 13 a9 bf 6b 01	⊕	ef a8 b6 db 44 52 71 0b a5 5b 25 ad 41 7f 3b 00	=
5	e0 c8 d9 85 92 63 b1 b8 7f 63 35 be e8 c0 50 01	e1 e8 35 97 4f fb c8 6c d2 fb 96 ae 9b ba 53 7c	e1 e8 35 97 fb c8 6c 4f 96 ae d2 fb 7c 9b ba 53	25 bd b6 4c d1 11 3a 4c a9 d1 33 c0 ad 68 8e b0	⊕	d4 7c ca 11 d1 83 f2 f9 c6 9d b8 15 f8 87 bc bc	=

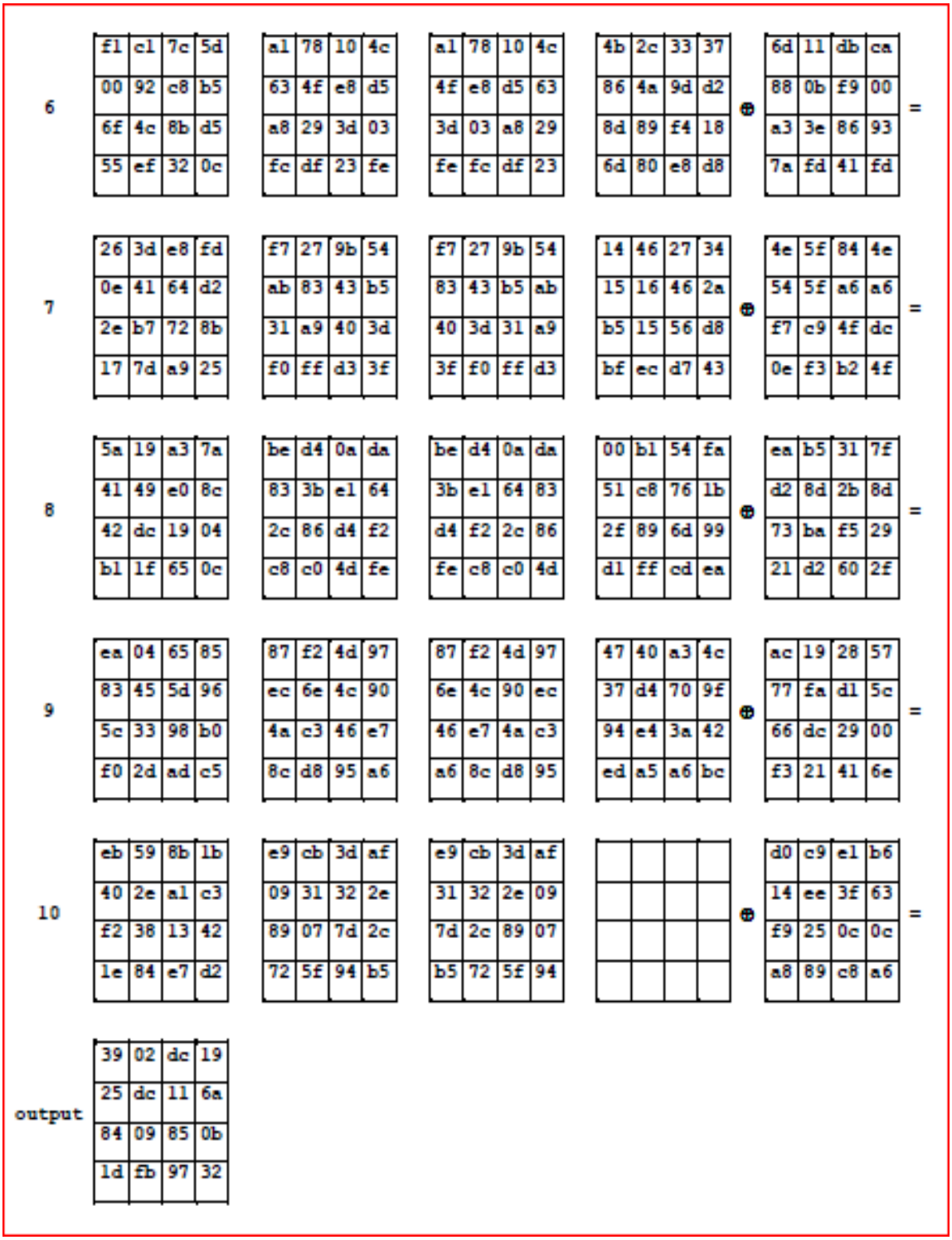


Fig A.2: Cipher Example

Appendix B

Appendix B.1 –Character LCD Interfacing

The Spartan®-3E FPGA Starter Kit board prominently features a 2-line by 16-character liquid crystal display (LCD). The FPGA controls the LCD via the 4-bit data interface shown . Although the LCD supports an 8-bit data interface, the Starter Kit board uses a 4-bit data interface to remain compatible with other Xilinx development boards and to minimize total pin count.

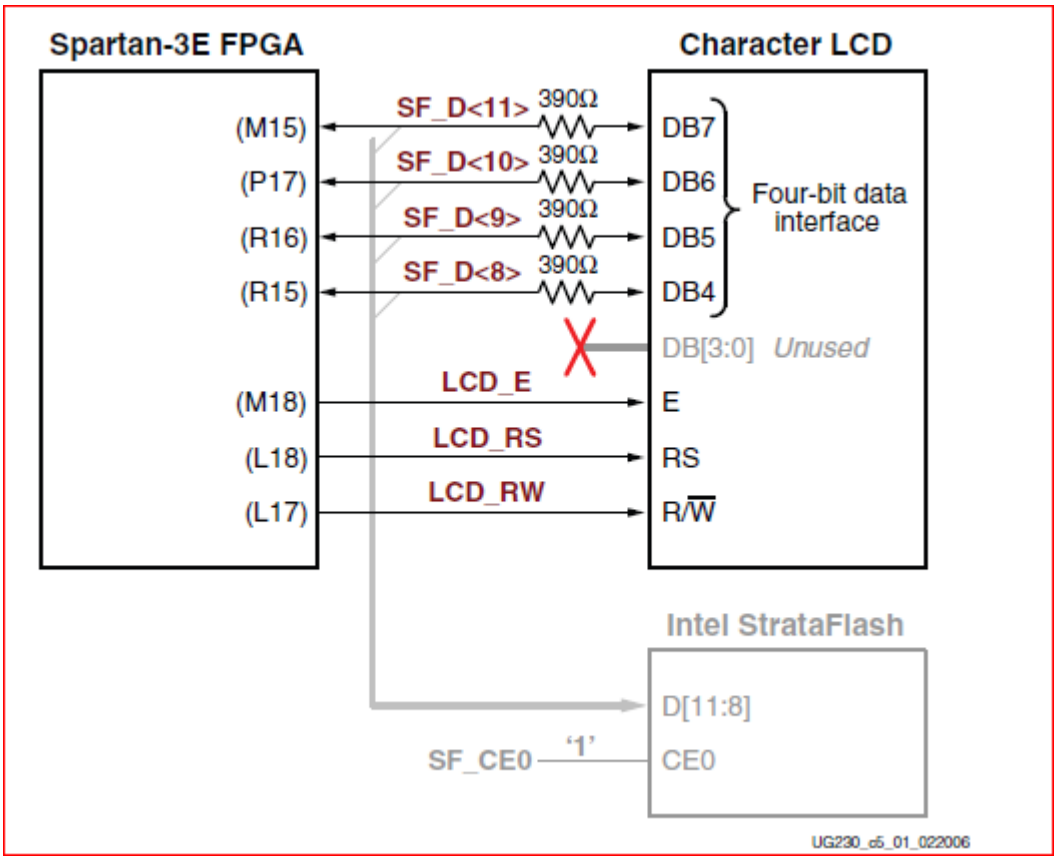


Fig B.1: Character LCD Interfacing

Operation

Four-Bit Data Interface

The board uses a 4-bit data interface to the character LCD. FigureB.2 illustrates a write operation to the LCD, showing the minimum times allowed for setup, hold, and enable pulse length relative to the 50 MHz clock (20 ns period) provided on the board.

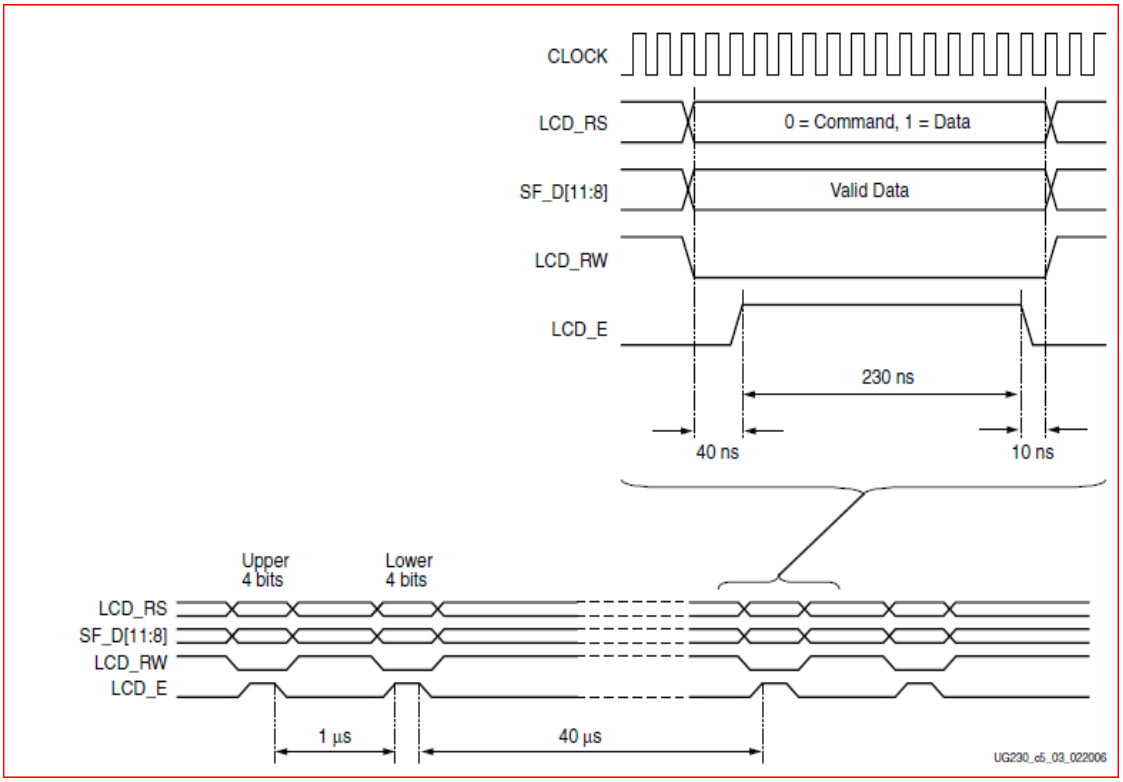


Fig B.2: Timing Diagram

The data values on SF_D<11:8>, and the register select (LCD_RS) and the read/write (LCD_RW) control signals must be set up and stable at least 40 ns before the enable LCD_E goes High. The enable signal must remain High for 230 ns or longer—the equivalent of 12or more clock cycles at 50 MHz . In many applications, the LCD_RW signal can be tied Low permanently because the FPGA generally has no reason to read information from the display.

Appendix B.2 –PS/2 Keyboard Interfacing

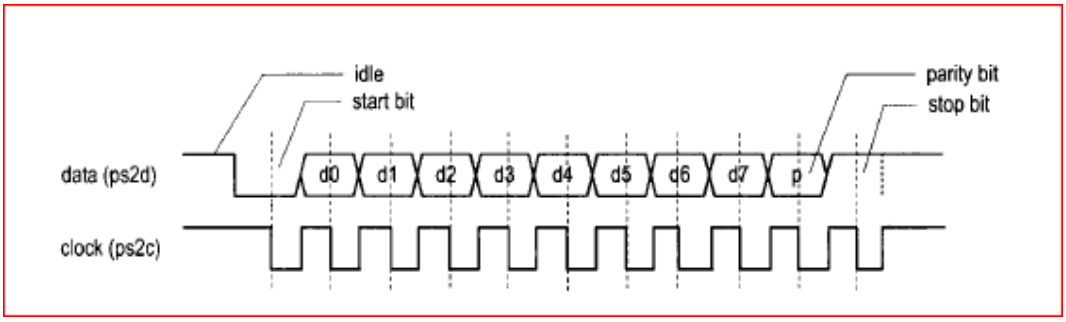
The Spartan®-3E FPGA Starter Kit board includes a PS/2 mouse/keyboard port and the standard 6-pin mini-DIN connector, labeled J14 on the board. Table shows the signals on the connector. Only pins 1 and 5 of the connector attach to the FPGA.

PS/2 DIN Pin	Signal	FPGA Pin
1	DATA (PS2_DATA)	G13
2	Reserved	G13
3	GND	GND
4	+5V	—
5	CLK (PS2_CLK)	G14
6	Reserved	G13

Table B.2: Pin Connection of PS/2 Keyboard and kit

Operation

The PS2 port was introduced in IBM's Personal System/2 personnel computers. It is a widely supported interface for a keyboard and mouse to communicate with the host. The PS2 port contains two wires for communication purposes. One wire is for data, which is transmitted in a serial stream. The other wire is for the clock information, which specifies when the data is valid and can be retrieved. The information is transmitted as an 11-bit "packet" that contains a start bit, 8 data bits, an odd parity bit, and a stop bit. Whereas the basic format of the packet is identical for a keyboard and a mouse, the interpretation for the data bits is different. The FPGA prototyping board has a PS2 port and acts as a host.



FigB.3: Timing Diagram of PS/2

REFERENCES

[1] J. Daemen and V. Rijmen,The Design of Rijndael: AES – The Advanced Encryption Standard, Springer-Verlag, 2002

[2] National Institute of Standard and Technology, Advanced Encryption Standard, Federal Information Processing Standards 197, November 2001.

[3]Nadia Nedjah, Luiza de Macedo Mourelle, Marco Paulo Cardoso, "A Compact Piplined Hardware Implementation of the AES-128 Cipher," itng, pp.216-221, Third International Conference on Information Tech-nology: New Generations (ITNG'06), 2006.

[4] FPGA Prototyping by verilog examples, Pong P. Chu Cleveland State University