DESIGN AND DEVELOPMENT OF AN OPTIMIZED HARDWARE ENCRYPTION MODULE OF GIGABIT THROUGHPUT BASED ON GALOIS FIELD ARITHMETIC FOR SECURE COMMUNICATION

by

Abhishek Bajpai
Secure Applications Section, Computer Division,
Bhabha Atomic Research Center, Mumbai
Enrollment Number- ENGG 01200701050

A dissertation submitted to the Board of Studies in Engineering Sciences In partial fulfillment of requirements for the Degree of

MASTER OF TECHNOLOGY

of

HOMI BHABHA NATIONAL INSTITUTE

August, 2010

Homi Bhabha National Institute

Recommendations of the Thesis Examining Committee

As members of the Thesis examining Committee, we recommend that the dissertation prepared by **Mr. Abhishek Bajpai** entitled **"Design and Development of an Optimized Hardware Encryption Module of Gigabit Throughput based on Galois Field Arithmetic for Secure Data Communication "** be accepted as fulfilling the dissertation requirement for the Degree of Master of Technology.

Designation	**Name**	**Signature**
Member-1	Mr. K V Nagesh, SO/H APPD	
Member-2	Mr. D A Roy, SO/G, RcnD	
Member-3	Mr. N O Kawade, SO/E, L&PTD	
Technical adviser	Mr. S K Parulkar, SO/H, Section Head SAS	
Co-guide	–	
Guide/Convener	Mr. V M Joshi, OS, Head EISD	
Chairman	Mr. C K Pithawa, OS, Head ED	

Declaration

I, hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree / diploma at this or any other Institution / University.

Abhishek Bajpai

Certificate

This is to certify that the dissertation entitled **"Design and Development of an Optimized Hardware Encryption Module of Gigabit Throughput based on Galois Field Arithmetic for Secure Data Communication "** being submitted by **Mr. Abhishek Bajpai** towards the partial fulfilment of requirement for the award of degree of Master of Technology in Electronics Science Engineering, HBNI, BARC, is a record of student's own work under our guidance. The matter contained in this report has not been submitted for the award of any other degree in any university or institute.

Technical adviser
Mr. S K Parulkar,
SO/H, Section Head SAS.

Guide
Mr. V M Joshi,
OS, Head EISD.

Acknowledgement

I would like to express my sincere thanks to Shri A G Apte, Head Computer Division who gave me this opportunity to pursue Mtech and present project on **"Design and Development of an Optimized Hardware Encryption Module of Gigabit Throughput based on Galois Field Arithmetic for Secure Data Communication "**.
Sincere thanks to Shri V M Joshi, Head EISD and Shri S K Parulkar, Head SAS for their valuable guidance, inspiration, and constant encouragement throughout this project.
Special thanks to Shri Bhagwan Bathe,Scientific Officer E Computer Division who provided me right tools and guided me on the FPGA technology facilitating me to jump start on the project quickly after algorithm design.
Thanks to Renuka Ghate SO(E), Mahesh Date SO(F), Gurmeet Singh Dhillon SO(D), D K Dixit SO(E) and Chetna Rastogi SA(E) for there constant support and encouragement.
I am grateful to my family and friends for there continuous support.

Contents

List of Figures

List of Tables

List of Pseudo Code

Chapter 1

Introduction

There is a constant security threat to the information we communicate on public media. Most of the communication is done without or with moderate security. It is required to shift to the secured environment in which the information is conveyed in encrypted form. But this shift is not possible with today 's available 8/32 bit processors as encryption algorithm are computational intensive. It is observed [3] that on average a processor architecture takes 22 cycles/byte for AES encryption resulting throughput of 36.3 Mbps for 100 Mhz clock.

On the other hand hardware approach based on FPGA is bit promising. As algorithm runs parallel in the core and there is a lot of scope for pipe-lining. Modular Hardware based on this approach can give us good solution for our security problems in communication devices.

Implementation of an Encryption algorithm on FPGA is not a new concept. But there is a scope in its optimum implementation in hardware.

Encryption hardware also gives us functional abstraction for a communicating device as device doesn't have to bother about the security or encryption. And you can always change the algorithm by just reprogramming it.

In this project different approaches for implementing AES encryption have been studied and it has been observed that the most compact design implementations are based on Composite field inversion.

Different optimization approaches are also been studied and developed for generating most compact core. For example Sbox and Inverse Sbox are implemented

as a single module which share a common core of field inversion.

Similarly Mix column and Inverse mix column are also implemented in a single module with deep byte level resource sharing.

A new design of Memory mapped State array is discussed which works as a accumulator and can handle 32 bit row as well as 32 bit column transformations.

For high throughput these cores can be implemented in parallel topology. Thus giving Giga bit throughput.

128 bit Design is also developed and throughput analysis is done for both designs.

Chapter 2

Algorithm Overview

For the AES (Rijndael)algorithm, the length of the input block, the output block and the State is 128 bits. This is represented by Nb = 4, which reflects the number of 32-bit Words (number of columns) in the State.

The length of the Cipher Key, K, is 128, 192, or 256 bits. The key length is represented by Nk = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key.

The number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by Nr, where Nr = 10 when Nk = 4, Nr = 12 when Nk = 6, and Nr = 14 when Nk = 8.

The only Key-Block-Round combinations that conform to this standard are given in table[2.1]. For implementation issues relating to the key length, block size and number of rounds.

	Key Length (Nk words)	**Block Size (Nb words)**	**Number of Rounds (Nr)**
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Table 2.1: Key-Block-Round Combinations.

For both its Cipher and Inverse Cipher, the AES algorithm uses a round function

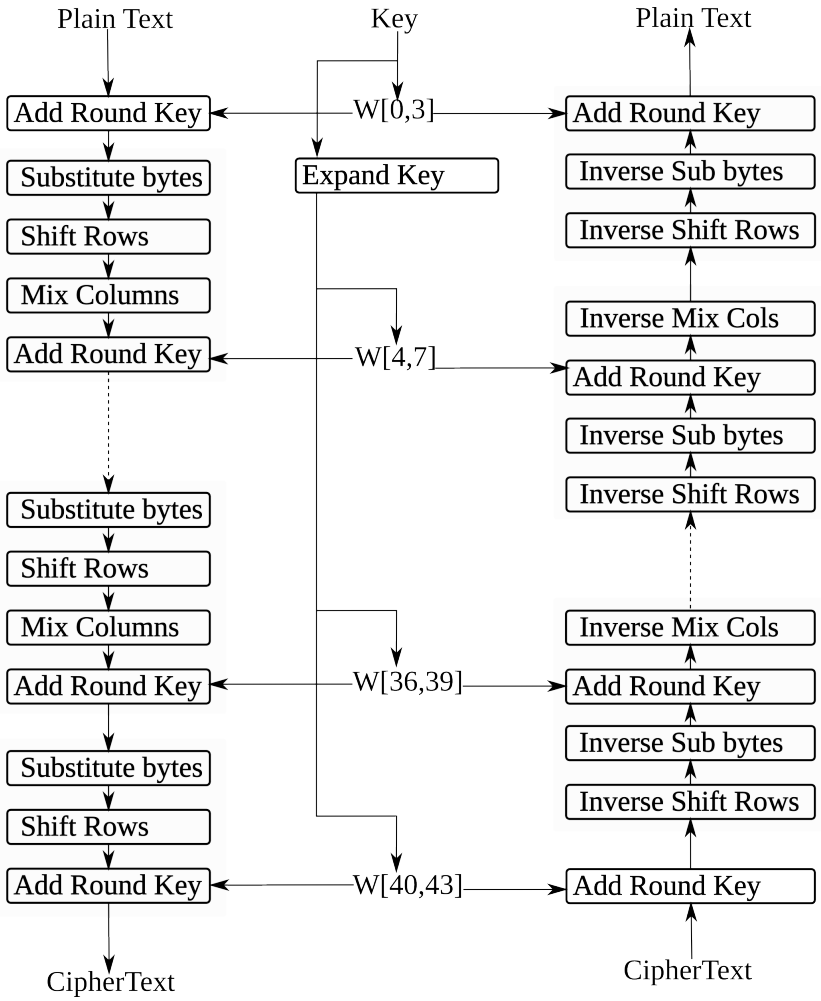


Figure 2.1: AES Algorithm Flow Diagram[10]

that is composed of four different byte-oriented transformations: 1) byte substitution using a substitution table (S-box), 2) shifting rows of the State array by different offsets, 3) mixing the data within each column of the State array, and 4) adding a Round Key to the State. These transformations (and their inverses) are described later in the report.

2.1 Cipher

At the start of the Cipher, the input is copied to the State array. 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. The round function is parametrized using a key schedule that consists of a one-dimensional array of four-Byte words derived using the Key Expansion routine.

The Cipher is described in pseudocode[1] (see figure[2.1]) . The individual transformations - SubByte(), ShiftRow(), MixColumn(), and AddRoundKey() - process the State and are described in the following subsections. In pseudocode[1], the array w[] contains the key schedule. As shown in pseudocode[1], all Nr rounds are identical with the exception of the final round, which does not include the MixColumn() transformation.

Pseudo Code 1 Pseudo Code for the Cipher

```
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. 2.5
    for round = 1 step 1 to Nr - 1
        SubBytes(state) // See Sec. 2.2
        ShiftRows(state) // See Sec. 2.3
        MixColumns(state) // See Sec. 2.4
        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
```

2.2 SubBytes()Transformation

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 (see figure[2.2]), which is invertible, is constructed by composing two transformations:

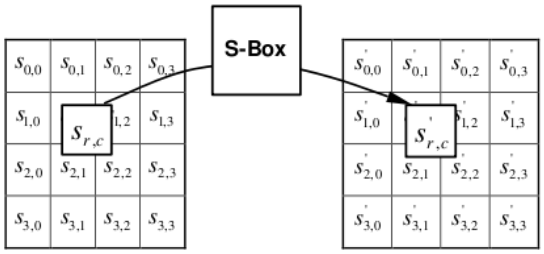


Figure 2.2: SubBytes() applies the S-box to each byte of the State

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

$$b'_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i \tag{2.1}$$

for $0 \leq i < 8$, where b_i is the i^{th} Bit of the byte, and c_i 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. In matrix form, the affine transformation element of the S-box can be expressed as:

$$\begin{bmatrix} bo \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{2.2}$$

SubBytes() transformation is typically used as a lookup table called as Sbox. It is presented in hexadecimal form (see figure[2.3]). For example, if s1,1 = {53}, then the substitution value would be determined by the intersection row with index '5 'and the column with index '3 'in Drawing2. This would result in s1,1 having a value of {ed}.

		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

Figure 2.3: S-box: substitution values for the byte xy (in hexadecimal format)

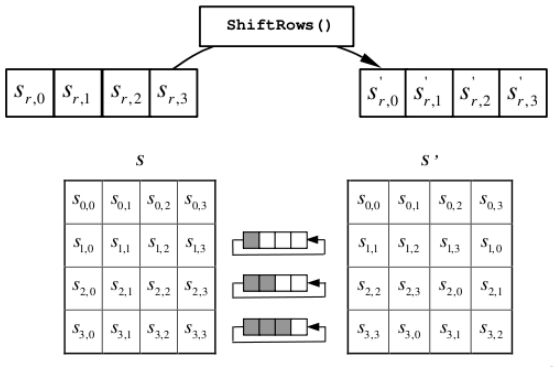


Figure 2.4: ShiftRow() cyclically shifts the last three rows in the State.

2.3 ShiftRows() Transformation

In the ShiftRow() transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). (see figure[2.4])The first row, r = 0, is not shifted. Specifically, the ShiftRow() transformation proceeds as follows:

$$s'_{r,c} = s_{r,(c+shift(r,Nb))modNb} \; for \; 0 < r < 4 \; and \; 0 \leq c < Nb \tag{2.3}$$

where the shift value shift(r,Nb) depends on the row number, r, as follows (recall that Nb = 4):

$$shift(1,4) = 1; shift(2,4) = 2; shift(3,4) = 3 \tag{2.4}$$

This has the effect of moving bytes to "lower"positions in the row (i.e., lower values of c in a given row), while the "lowest"bytes wrap around into the "top"of the row (i.e., higher values of c in a given row). Drawing3 illustrates the ShiftRow() transformation.

2.4 MixColumn() Transformation

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

$$a(x) = \{03\}\ x^3 \ + \ \{01\}\ x^2 \ + \ \{01\}\ x \ + \ \{02\} \tag{2.5}$$

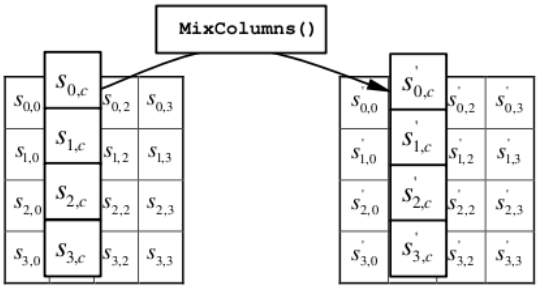


Figure 2.5: MixColumn() operates on the State column-by-column.

This can be written as a matrix multiplication. Let

s '(x) = a (x) $\oplus$ s (x) :

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \oplus \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{2.6}$$

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

$$s'_{0,c} = (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \tag{2.7}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} \tag{2.8}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) \tag{2.9}$$

$$s'_{0,c} = (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c}) \tag{2.10}$$

2.5 AddRoundKey() Transformation

In the AddRoundKey() transformation, a Round Key is added to the State by a simple bit-wise XOR operation. Each Round Key consists of Nb words from the key schedule. Those Nb Words are each added into the columns of the State, such that

$$\begin{bmatrix} s'_{0,c} & s'_{1,c} & s'_{2,c} & s'_{3,c} \end{bmatrix} = \begin{bmatrix} s_{0,c} & s_{1,c} & s_{2,c} & s_{3,c} \end{bmatrix} \oplus [w_{roundNb+c}] \tag{2.11}$$

where $[w_i]$ are the key schedule Words, and round is a value in the range $0 \leq$ round $\leq$ Nr. In the Cipher, the initial Round Key addition occurs when round

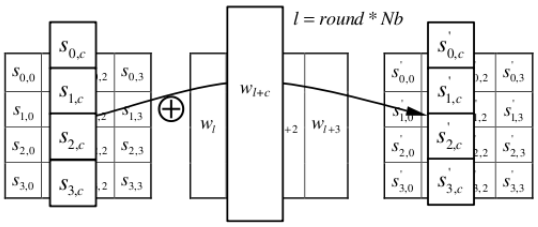


Figure 2.6: AddRoundKey() XORs each column of the State with a word from the key schedule

= 0, prior to the first application of the round function. The application of the AddRoundKey() transformation to the Nr rounds of the Cipher occurs when 1 âĽď round âĽď Nr.

The action of this transformation is illustrated figure[2.6], where l = round * Nb.

2.6 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 $[w_i]$, with i in the range 0 âĽď i $<$ Nb(Nr + 1). The expansion of the input key into the key schedule proceeds according to the pseudocode[2].

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 [a0,a1,a2,a3] as input, performs a cyclic permutation, and returns the word $[a_1, a_2, a_3, a_0]$. The round constant word array, Rconi, 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 $GF(28^{)}$ (note that i starts at 1, not 0).

From pseudocode[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, Rconi. 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()).

It is important to note that the Key Expansion routine for 256- Cipher Keys (Nk = 8) is slightly different than for 128- and 192-Bit Cipher Keys. If Nk = 8 and i-4 is a multiple of Nk, then SubWord() is applied to w[i-1] prior to the XOR.

Pseudo Code 2 Pseudo Code for Key Expansion

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

2.7 Inverse Cipher

The Cipher transformations in Sec.2.1 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 - InvShiftRow(), InvSubByte(),InvMixColumn(), and AddRoundKey() - process the State and are described in the following subsections.
The Inverse Cipher is described in the pseudocode[3]. In pseudocode[3], the array w[ ] contains the key schedule.

Pseudo Code 3 Pseudo Code for Inverse Cipher

```
InvCipher(byte in[4 X Nb], byte out[4 X Nb], word w[Nb X (Nr+1)]) begin
    byte state[4,Nb]
    state = in
    AddRoundKey(state, w[Nr X Nb, (Nr+1) X Nb-1]) // See Sec. 2.5
    for round = Nr-1 step -1 downto 1
        InvShiftRows(state) // See Sec. 2.8
        InvSubBytes(state) // See Sec. 2.9
        AddRoundKey(state, w[round X Nb, (round+1) X Nb-1])
        InvMixColumn(state) // See Sec. 2.10
    end for
    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[0, Nb-1])
    out = state
end
```

2.8 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 âĹŠ shift (r , Nb) bytes, where the shift value shift(r,Nb) depends on the row number, and is given in equation (2.4) (see Sec. 2.3).

Specifically, the InvShiftRows() transformation proceeds as follows:

$$s'_{r,(c+shift(r,Nb))modNb} = s_{r,c} \quad for\ 0 < r < 4\ and\ 0 \leq c < Nb \tag{2.12}$$

Drawing6 illustrates the InvShiftRows() transformation.

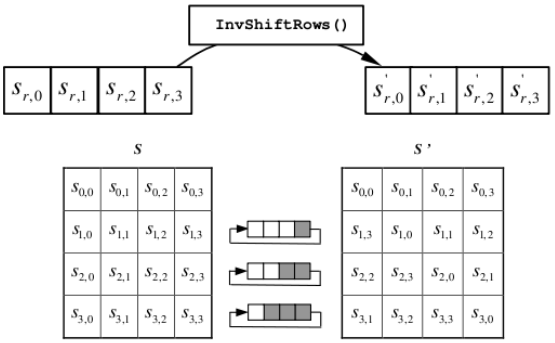


Figure 2.7: InvShiftRows()cyclically shifts the last three rows in the State.

		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

Figure 2.8: Inverse S-box: substitution values for the byte xy (in hexadecimal format).

2.9 InvSubByte() Transformation

InvSubByte() 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 see eq:(2.2) followed by taking the multiplicative inverse in $GF(2^8)$. The inverse S-box used in the InvSubByte() transformation is presented in figure 2.8

2.10 InvMixColumn() Transformation

InvMixColumn() is the inverse of the MixColumns() transformation. glsInvMixColumn() operates on the State column-by-column, treating each column as a four-term polynomial. 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

$$a^-1(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \tag{2.13}$$

this can be written as a matrix multiplication. Let

$s(x) = a1(x) \oplus s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \oplus \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{2.14}$$

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

$$s'_{0,c} = (\{0e\} \cdot s_{0,c}) \oplus (\{0b\} \cdot s_{1,c}) \oplus (\{0d\} \cdot s_{2,c}) \oplus (\{09\} \cdot s_{3,c}) \tag{2.15}$$

$$s'_{1,c} = (\{09\} \cdot s_{0,c}) \oplus (\{0e\} \cdot s_{1,c}) \oplus (\{0b\} \cdot s_{2,c}) \oplus (\{0d\} \cdot s_{3,c}) \tag{2.16}$$

$$s'_{2,c} = (\{0d\} \cdot s_{0,c}) \oplus (\{09\} \cdot s_{1,c}) \oplus (\{0e\} \cdot s_{2,c}) \oplus (\{0b\} \cdot s_{3,c}) \tag{2.17}$$

$$s'_{3,c} = (\{0b\} \cdot s_{0,c}) \oplus (\{0d\} \cdot s_{1,c}) \oplus (\{09\} \cdot s_{2,c}) \oplus (\{0e\} \cdot s_{3,c}) \tag{2.18}$$

2.11 Inverse AddRoundKey() Transformation

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

2.12 Equivalent Inverse Cipher

In the straightforward Inverse Cipher presented in Sec. 2.7 and pseudo-code[3], the sequence of the transformations differs from that of the Cipher, while the form of the key schedules for encryption and decryption remains the same. However, several properties of the AES algorithm allow for an Equivalent Inverse Cipher that has the same sequence of transformations as the Cipher (with the transformations replaced by their inverses). This is accomplished with a change in the key schedule.

The two properties that allow for this Equivalent Inverse Cipher are as follows:

1. The SubBytes() and ShiftRow() transformations commute; that is, a SubBytes() transformation immediately followed by a ShiftRow() transformation is equivalent to a ShiftRow() transformation immediately followed buy a SubBytes() transformation. The same is true for their inverses, InvSubBytes() and InvShiftRows.

2. The column mixing operations - MixColumn() and InvMixColumn() - are linear with respect to the column input, which means

$$InvMixColumn(StateXORRoundKey) = \\ InvMixColumn(State) \oplus InvMixColumn(RoundKey) \tag{2.19}$$

These properties allow the order of InvSubByte() and InvShiftRows() transformations to be reversed. The order of the AddRoundKey() and InvMixColumn() transformations can also be reversed, provided that the columns (words) of the decryption key schedule are modified using the InvMixColumn() transformation.

The equivalent inverse cipher is defined by reversing the order of the InvSubByte() and InvShiftRows() transformations shown in Table4, and by reversing the order of the AddRoundKey() and InvMixColumn() transformations used in the "round loop "after first modifying the decryption key schedule for round = 1 to Nr-1 using the InvMixColumn() transformation. The first and last Nb words of the decryption key schedule shall not be modified in this manner.

Given these changes, the resulting Equivalent Inverse Cipher offers a more efficient structure than the Inverse Cipher described in Sec. 2.7 and pseudo-code[3]. Pseudo code for the Equivalent Inverse Cipher appears in pseudo-code[4]. (The word array w[ ] contains the modified decryption key schedule. The modification to the Key Expansion routine is also provided in pseudo-code[4])

Pseudo Code 4 Pseudo Code for the Equivalent Inverse Cipher.

```
InvCipher(byte in[4 X Nb], byte out[4 X Nb], word dw[Nb X (Nr+1)])
begin
    byte state[4,Nb]
    state = in
    AddRoundKey(state, dw[Nr X Nb, (Nr+1) X Nb-1])
    for round = Nr-1 step -1 downto 1
        InvSubBytes(state)
        InvShiftRows(state)
        InvMixColumns(state)
        AddRoundKey(state, dw[round X Nb, (round+1) X Nb-1])
    end for
    InvSubBytes(state)
    InvShiftRows(state)
    AddRoundKey(state, dw[0, Nb-1])
    out = state
end
```

For the Equivalent Inverse Cipher, the following pseudo code is added at the end of the Key Expansion routine

```
    for i = 0 step 1 to (Nr+1) X Nb-1
        dw[i] = w[i]
    end for
    for round = 1 step 1 to Nr-1
        InvMixColumns(dw[round X Nb, (round+1) X Nb-1]) // note change of type
    end for
```

Note that, since InvMixColumn operates on a two-dimensional array of bytes while the Round Keys are held in an array of words, the call to InvMixColumn in this code sequence involves a change of type (i.e. the input to InvMixColumn() is normally the State *which is considered to be a two-dimensional array of bytes, whereas the input here is a Round array, Key computed as a one-dimensional array of words).*

Chapter 3

InvMixColumn Decomposition and Multilevel Resource Sharing[5]

The MixColumn transformation multiplies each column of the State by polynomial c(X) in the ring R. The c(X) is defined as follows as describe in section 2.4:

$$c(X) = \{03\}X^3 + X^2 + X + \{02\} \tag{3.1}$$

The InvMixColumn transformation is the inverse of the MixColumn operation. InvMixColumn multiplies each column of the State by as describe in section 2.10

$$d(X) = \{0B\}X^3 + \{0D\}X^2 + \{09\}X + \{0E\} \tag{3.2}$$

where d(X) = c^-1 (X)
From eq: 3.1 and eq: 3.2, we can see that coefficients of d(X) are more complex than coefficients of c(X).As a result, hardware implementing AES decryption is larger and slower than for encryption. In order to reduce hardware cost, the InvMixColumn can be decomposed to share logic resources with MixColumn.

3.1 Byte-Level Resource Sharing in MixColumn

The first byte of the MixColumn implementation based on byte-level resource sharing

$$b_0 = \{02\}a_0 + \{03\}a_1 + a_2 + a_3 \tag{3.3}$$

Multiplication in algebraic fields is distributive over addition. This property enables byte-level resource sharing. I can further reduce this equation to

$$=> (a_0 + a_1 + a_2 + a_3) + \{02\}(a_0 + a_1) + a_0 \tag{3.4}$$

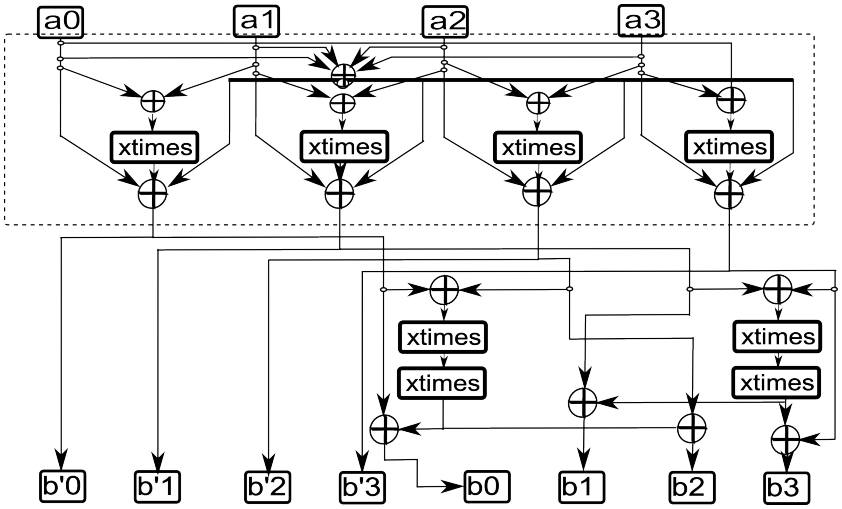


Figure 3.1: Multi Level Resource Sharing

the term a0+ a1 + a2 + a3 can be shared by all four bytes of the MixColumn function.

$$b0 = (a_0 + a_1 + a_2 + a_3) + \{02\}(a_0 + a_1) + a_0 \tag{3.5}$$

$$b1 = (a_0 + a_1 + a_2 + a_3) + \{02\}(a_1 + a_2) + a_1 \tag{3.6}$$

$$b2 = (a_0 + a_1 + a_2 + a_3) + \{02\}(a_2 + a_3) + a_2 \tag{3.7}$$

$$b3 = (a_0 + a_1 + a_2 + a_3) + \{02\}(a_3 + a_0) + a_3 \tag{3.8}$$

3.2 Byte-Level Resource Sharing in InvMixColumn

The first byte of inverse mix column is

$$b'_0 = \{0E\}a_0 + \{0B\}a_1 + \{0D\}a_2 + \{09\}a_3 \tag{3.9}$$

it can be further expanded as

$$b'_0 = \{02\}(a_0 + a_1) + a_1 + a_2 + a_3 + \{04\}(\{02\}(a_0 + a_1) + \{02\}(a_2 + a_3) + (a_0 + a_2)) \tag{3.10}$$

where

$$\begin{aligned} &(\{02\}(a_0 + a_1) + \{02\}(a_2 + a_3) + (a_0 + a_2)) \\ &=> \{03\}a_0 + \{02\}a_1 + \{03\}a_2 + \{03\}a_3 \\ &=> (\{02\}a_0 + \{03\}a_1 + a_2 + a_3) + (a_0 + a_1 + \{03\}a_2 + \{02\}a_3) \\ &=> b_0 + b_2 \end{aligned} \tag{3.11}$$

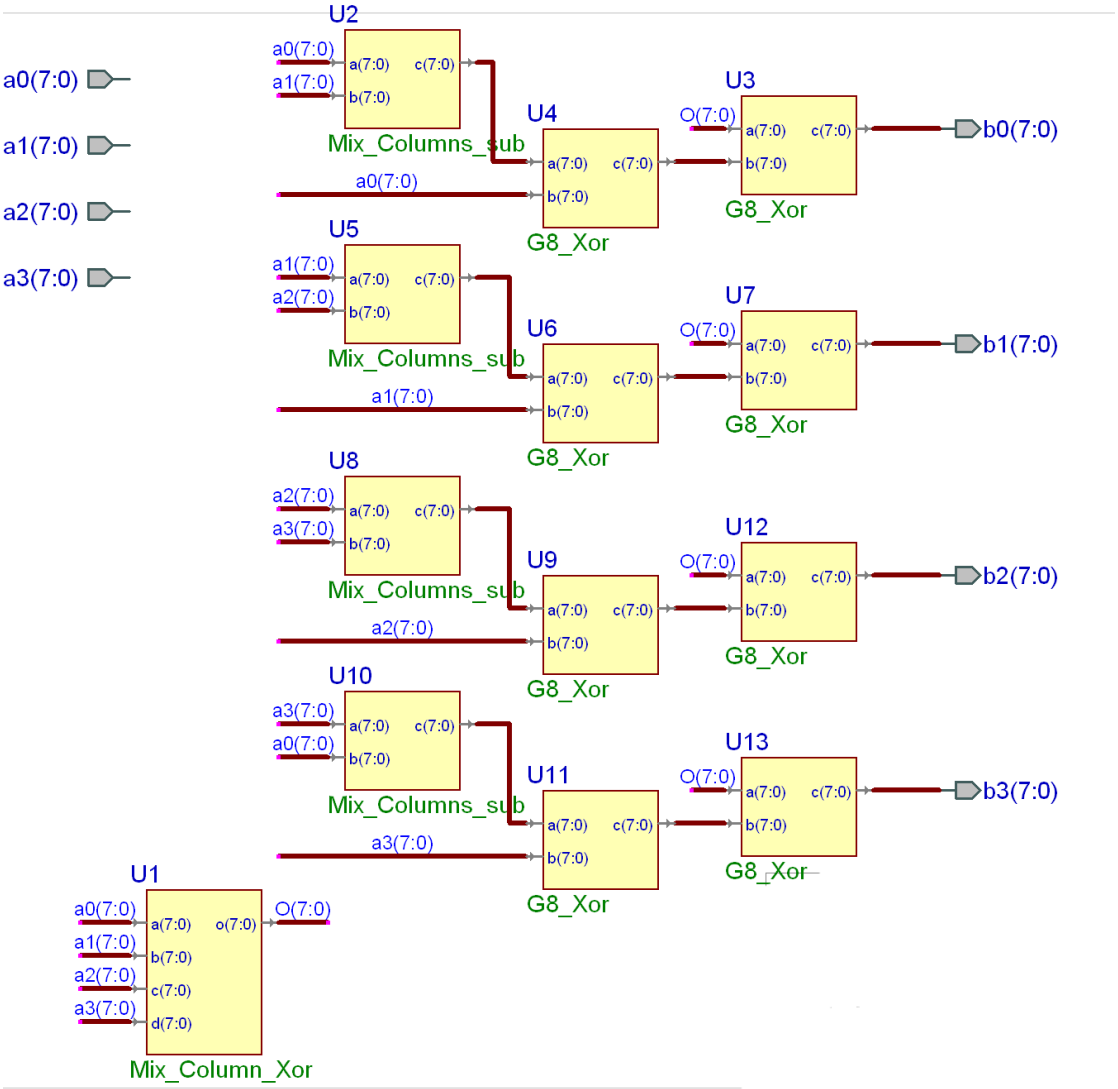


Figure 3.2: Mix Column Design

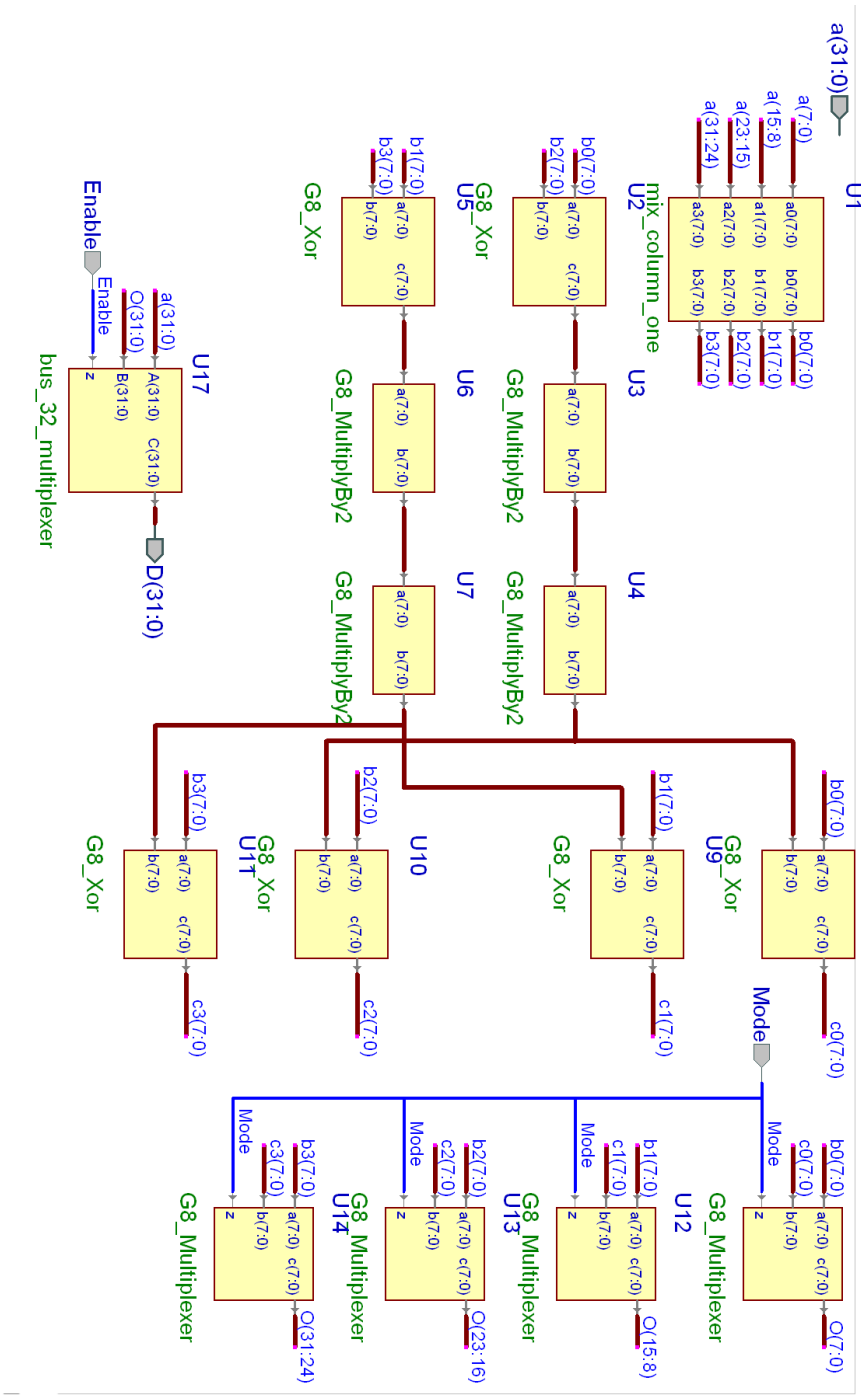


Figure 3.3: Inverse Mix Column with Resource sharing

and

$$\begin{aligned} & \{02\}(a_0 + a_1) + a_1 + a_2 + a_3 \\ => & \{02\}(a_0 + a_1) + a_1 + a_2 + a_3 + \mathbf{a_0} \ + \ \mathbf{a_0} \\ => & b_0 \end{aligned} \tag{3.12}$$

so

$$b'_0 = b_0 + \{04\}(b0 + b2) \tag{3.13}$$

similarly

$$\begin{aligned} b'_1 &= b_1 + \{04\}(b1 + b3) \\ b'_2 &= b_2 + \{04\}(b0 + b2) \\ b'_3 &= b_3 + \{04\}(b1 + b3) \end{aligned} \tag{3.14}$$

in this way it can be shown that Inv Mix Column is related to Mix column and further logic can be reduced order of times by resource sharing.

In the figure 3.1 the scheme is shown for Multilevel Resource Sharing for Mix-column and inverse mixcolumn.

Chapter 4

Byte Substitution[13]

The Byte Substitution transformation operates independently on each byte of the state. The operation comprises of 2 sub-steps: as described in section 2.2

1. Inversion: Multiplicative inverse of each byte is taken in $GF(2^8)$, and $\{00\}$ is mapped to itself.
2. Affine Transformation: This sub-step is performed in GF(2).

To implement the Byte Substitution transformation, many techniques have been reported. Those are, for instances;

1. The table lookup technique where step 2 is usually combined into a single table known as S-box. For the current technology and design, the table size of 256×8 bit is not considerably big. This technique has been adopted in many realizations.
2. Synthesis and optimized logic function of S-box using CAD tools, and
3. Compute the inversion of element in $GF(2^8)$ and optimize the logic functions. The efficiency of the third technique is much depended on the mathematical theory of field element inversion. This approach is highly considered when the table lookup is not applicable or when the compact design is a case. It also provides desirable features for the highly-paralleled computation.

In this project option (3) is chosen since the field inversion hardware can be easily shared by both the encryption process and decryption process. The Byte Substitution (and similarly, the inverse Byte Substitution) transform of a byte is defined mathematically as:

$$D(x) = \delta A^{-1}(x)_{mod(x^8+1)} \oplus C(x) \tag{4.1}$$

where $C(x) = x^6 + x^5 + x + 1 = \{63\}$ and $\delta = \{1F\} = x^4 + x^3 + x^2 + x + 1$. For the inverse Byte Substitution computation, $= \{4A\} = x^6 + x^3 + x$ and $C(x) = x^2 + 1 = \{05\}$ are used respectively. The constant C(x) has been added in order that the Sbox has no fixed point (a map to a) and no opposite fixed point (a map to $\bar{a}$). Besides the field inversion, the implementation of such a transformation is fairly simple as the circuit can be built up form an array of XOR gates.

In order to show the actual calculations of the field implementation let me introduce the finite field arithmetic and Galois field inversion.

4.1 The Finite Field

finite field or Galois field (so named in honor of Evariste Galois) is a field that contains only finitely many elements.

The finite field GF(2) consists of the set {0, 1} where all operations work modulo 2. We use GF(2)[x] to denote polynomials with coefficients in GF(2). Define the field $GF(16) = GF(2)[x]/(x^4 + x + 1)$; the polynomials with coefficients in GF(2) modulo $x^4 + x + 1$. The field GF(16) is most easily thought of as consisting of the 16 polynomials of degree less than 4 where all operations work modulo $x^4 + x + 1$. That means we have $x^4 + x + 1 = 0$ or $x^4 = x + 1$ (note addition and subtraction are the same since coefficients work modulo 2 where -1 = 1, so adding two equal terms cancels them out). It is also useful to note that $x^5 = x^2 + x$ and $x^6 = x^3 + x^2$. So in GF(16), we have $(x^3 + x^2 + 1)(x^3) = x^6 + x^5 + x^3 = (x^3 + x^2) + (x^2 + x) + x^3 = x$. Note that the polynomial $x^4 + x + 1$ can not be factored (in a non-trivial way) into two polynomials in GF(2)[x], so we say that $x^4 + x + 1$ is irreducible over GF(2)[x]. This irreducibility makes $GF(16) = GF(2)[x]/(x^4 + x + 1)$ a field in a similar way to the fact that $GF(p) = Z/(p)$ is a field since prime numbers are irreducible over Z. Since GF(16) is a field, we can invert all non-zero elements. This is very similar to inverting elements in a finite field of the form GF(p) (the integers modulo p) where p is a prime number. That is because the Euclidean algorithm can be applied to polynomials as well. In the polynomial version, the remainder is always of lower degree than the divisor.

Let us review inversion in the more familiar setting of GF(229) (229 is prime, so this is just the integers modulo 229) and then see how it works in GF(16). Let us

invert 37 in GF(229). We first use the Euclidean algorithm to find the greatest common divisor of 37 and 229 (which is 1) and then work backwards to write 1 as an integer linear combination of 37 and 229 and reduce that equation modulo 229. We will then invert $x^3 + x^2$ in GF(16); the steps are essentially identical.

$$
\begin{aligned}
229 &= 6 \cdot 37 + 7 \\
37 &= 5 \cdot 7 + 2 \\
7 &= 3 \cdot 2 + 1 \\
1 &= 7 - 3 \cdot 2 \\
1 &= 7 - 3 \cdot (37 - 5 \cdot 7) \\
1 &= 16 \cdot 7 - 3 \cdot 37 \\
1 &= 16 \cdot (229 - 6 \cdot 37) - 3 \cdot 37 \\
1 &= 16 \cdot 229 - 99 \cdot 37 \\
1 &\equiv 16 \cdot 229 - 99 \cdot 37 \pmod{229} \\
1 &\equiv 16 \cdot 0 + 130 \cdot 37 \pmod{229} \\
37^{-1} &\equiv 130 \pmod{229}
\end{aligned}
$$

$$
\begin{aligned}
x^4 + x + 1 &= (x + 1)(x^3 + x^2) + (x^2 + x + 1) \\
x^3 + x^2 &= (x)(x^2 + x + 1) + x \\
x^2 + x + 1 &= (x + 1)(x) + 1 \\
1 &= (x^2 + x + 1) + (x + 1)(x) \\
1 &= (x^2 + x + 1) + (x + 1)((x^3 + x^2) + (x)(x^2 + x + 1)) \\
1 &= (x^2 + x + 1)(x^2 + x + 1) + (x + 1)(x^3 + x^2) \\
1 &= (x^2 + x + 1)((x^4 + x + 1) + (x + 1)(x^3 + x^2)) + (x + 1)(x^3 + x^2) \\
1 &= (x^2 + x + 1)(x^4 + x + 1) + (x^3 + x)(x^3 + x^2) \\
1 &\equiv (x^2 + x + 1)(x^4 + x + 1) + (x^3 + x)(x^3 + x^2) \pmod{x^4 + x + 1} \\
1 &\equiv (x^2 + x + 1)(0) + (x^3 + x)(x^3 + x^2) \pmod{x^4 + x + 1} \\
(x^3 + x^2)^{-1} &\equiv x^3 + x \pmod{x^4 + x + 1}
\end{aligned}
$$

The extended Euclidean algorithm can also be used to calculate the multiplicative inverse in a finite field.

Given the irreducible polynomial f(x) used to define the finite field, and the element a(x) whose inverse is desired, then a form of the algorithm suitable for determining the inverse is given by the following pseudo code.

NOTE: remainder() and quotient() are functions different from the arrays remainder[ ] and quotient[ ]. remainder() refers to the remainder when two numbers are divided, and quotient() refers to the integer quotient when two numbers are divided. For example, remainder(5/3) = 2 and quotient(5/3) = 1. Equivalent

operators in the C language are % and / respectively.

Pseudo Code 5 Pseudo Code for calculating inverse with Extended Euclidean algorithm

```
remainder[1] := f(x)
remainder[2] := a(x)
auxiliary[1] := 0
auxiliary[2] := 1
i := 2
while remainder[i] > 1
        i := i + 1
        remainder[i] := remainder(remainder[i-2] / remainder[i-1])
        quotient[i] := quotient(remainder[i-2] / remainder[i-1])
        auxiliary[i] := -quotient[i] * auxiliary[i-1] + auxiliary[i-2]
inverse := auxiliary[i]
```

Let us recalculate the $(x^3 + x^2)^{-1}$ in GF(16) based on extended Euclidean algorithm.

$$
\begin{array}{l}
x^3+x^2 \overline{)x^4+x+1}\,(x+1 \\
\quad \underline{x^4+x^3} \\
\quad\; x^3+x+1 \\
\quad\; \underline{x^3+x^2} \\
\quad\;\; x^2+x+1 \overline{)x^3+x^2}\,(x \\
\qquad\qquad\; \underline{x^3+x^2+x} \\
\qquad\qquad\qquad\; x \;\;)\overline{x^2+x+1}\,(x+1 \\
\qquad\qquad\qquad\quad\; \underline{x^2+x} \\
\qquad\qquad\qquad\qquad\; 1
\end{array}
$$

So the result is $(x^3 + x^2)^1 \equiv x^3 + x(mod x^4 + x + 1)$

example from $GF(2^8)$ where $GF(2^8)$ is defined by $GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$ lets calculate inverse of $x^6 + x^4 + x + 1 = \{53\}$.

i	remainder[i]	quotient[i]	auxiliary[i]
1	x^4+x+1		0
2	x^3+x^2		1
3	x^2+x+1	$x+1$	$x+1$
4	x	x	x^2+x+1
5	1	$x+1$	$x^3+\cancel{x^2}+\cancel{x}+\cancel{x^2}+\cancel{x}+\cancel{1}+x+\cancel{1}$

Table 4.1: Extended Euclidean auxiliary Table

$$
\begin{array}{l}
x^6+x^4+x+1\,\overline{)\,x^8+x^4+x^3+x+1}\;\big(x^2+1 \\
\quad\;\underline{x^8+x^6+x^3+x^2} \\
\quad\;\overline{x^6+x^4+x^2+x+1} \\
\quad\;\underline{x^6+x^4+x+1} \\
\qquad\qquad\quad\;\overline{x^2+1}\;\overline{)\,x^6+x^4+x+1}\;\big(x^4+x^2 \\
\qquad\qquad\qquad\;\underline{x^6} \\
\qquad\qquad\qquad\quad\;\overline{x^4+x+1} \\
\qquad\qquad\qquad\quad\;\underline{x^4} \\
\qquad\qquad\qquad\qquad\;\overline{x+1}\;\overline{)\,x^2+1}\;\big(x+1 \\
\qquad\qquad\qquad\qquad\quad\;\underline{x^2+x} \\
\qquad\qquad\qquad\qquad\qquad\;\overline{x+1} \\
\qquad\qquad\qquad\qquad\qquad\quad\;\underline{x+1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\;\overline{1}
\end{array}
$$

i	remainder[i]	quotient[i]	auxiliary[i]
1	$x^8+x^4+x^3+x+1$		0
2	x^6+x^4+x+1		1
3	x^2	x^2+1	x^2+1
4	$x+1$	x^4+x^2	$x^6+\cancel{x^4}+\cancel{x^4}+x^2+1$
5	1	$x+1$	$x^7+x^3+x+x^6+\cancel{x^2}+\cancel{1}+\cancel{x^2}+\cancel{1}$

Table 4.2: Extended Euclidean auxiliary Table

Thus, the inverse is $x^7+x^6+x^3+x=\{CA\}$,

Logic implementation of this method on FPGA is complicated and too recursive. Hardware design would also be very big.

4.2 Composite Galois Field

4.2.1 Mathematical background[12]

Field elements for a certain k in $GF(2^k)$ are distinct for a certain irreducible polynomial. Let $GF(2^k)$ denote the binary extension field defined over the prime field GF(2). In order to construct $GF(2^k)$ and represent its elements, we need a primitive element a, and an irreducible polynomial of degree k whose coefficients are in $GF(2^k)$. If a is a root of the irreducible polynomial, then the set $\{1, , ^2, ..., ^{k-1}\}$ forms a basis for the field $GF(2^k)$. An element A of $GF(2^k)$ can be expressed as $A = \sum_{i=0}^{k-1} a_i \alpha^i$; where $a_i \in GF(2) \quad for \quad i = 0, 1, ..., k-1, \quad and \quad a_0 \ a_1 \ ... \ a_{k-1}$ is its K-tuple representation. In case $GF(2^4)$ is considered:

$$A = \sum_{i=0}^{k-1} a_i \alpha^i = \sum_{i=0}^{3} a_i \alpha^i = a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3 \tag{4.2}$$

$$\Rightarrow A = a_0 \ a_1 \ a_2 \ a_3 \ \in \ GF(2^4) \ where \ a_i \ \in \ GF(2) \tag{4.3}$$

There are various ways to represent the elements of $GF(2^k)$. If k is the product of two integers as k = nm, then, it is possible to derive a different representation method by defining $GF(2^k)$ over the field $GF(2^n)$. The field $GF(2^n)$ over which the composite field is defined is called the ground field. In the quaternary case, the ground field becomes $GF(2^2) = GF(4)$. An extension field defined over a subfield of $GF(2^k)$ other than the prime field GF(2) is known as a composite field. We will use $GF((2^2)^m)$ to denote the quaternary composite field. Since there is only one field with 2k elements,

both the binary and the composite fields refer to the same field. However, their representation methods are different, and it is possible to obtain one representation from the other. Let $GF((2^n)^m)$ denote the extension field defined over the ground field $GF(2^n)$. In order to construct $GF((2^n)^m)$ and represent its elements, we need a primitive element a, and an irreducible polynomial of degree m whose coefficients are in $GF(2^n)$. If α is a root of the irreducible polynomial, then the set $\{1, \alpha, \alpha_2, ..., \alpha_m - 1\}$ forms a basis for the field $GF((2^n)^m)$. There are many such polynomials. An element A of $GF((2^2)^m)$ can be expressed as $A = \sum_{i=0}^{k-1} a_i \alpha^i$, where $a_i \ \in \ GF(2) = GF(4) \ for \ i = 0, 1, ..., m-1$. In case $GF((2^2)^2)$ is considered

$$A = \sum_{i=0}^{m-1} a_i \alpha^i = \sum_{i=0}^{1} a_i \alpha^i = a'_0 + a'_1\alpha \tag{4.4}$$

$$\Rightarrow A =< a_0' \;\; a_1' > \in \;\; GF((2^2)^2) \;\; where \;\; a_i \;\; \in \;\; GF(2^2) \;\; = \;\; GF(4) \qquad (4.5)$$

Let $GF(2^n)$ denote the binary extension field defined over the prime field GF(2). In order to construct $GF(2^n)$ and represent its elements, we need a primitive element β, and an irreducible polynomial of degree n whose coefficients are in GF(2). If β is a root of the irreducible polynomial, then the set $\{1, \beta, \beta^2, ..., \beta^{n-1}\}$ forms a basis for the field $GF(2^n)$. An element A of $GF(2^n)$ can be expressed as,$\sum_{j=0}^{n-1} a_j\beta^j$, $where \;\; a_j \in GF(2) \;\; for \;\; j = 0, 1, ..., n-1$. In case $GF(2^2)$ is considered, the only irreducible polynomial of degree 2 is $x^2 + x + 1$, and A can be represented as

$$A = \sum_{j=0}^{n-1} a_j\beta^j = \sum_{j=0}^{1} a_j\beta^j = a_0 + a_1\beta \qquad (4.6)$$

4.2.2 The Relation Between $GF(2^4)$ and $GF((2^2)^2)$[1]

From Eqs. 4.4 and 4.6, $a_i \in GF(2^2)$ hence

$$a_i' = \sum_{j=0}^{l} a_{ij}\beta^j \qquad (4.7)$$

$$\Rightarrow a_0' = a_{00} + a_{01}\beta \qquad (4.8)$$

$$\Rightarrow a_1' = a_{10} + a_{11}\beta \qquad (4.9)$$

$$\Rightarrow A =< a_{00} \;\; a_{01} \;\; a_{10} \;\; a_{11} > \in \;\; GF((2^2)^2) \qquad (4.10)$$

To establish the relation between Eqs. 4.3 and 4.9, the conversion parameter $\beta = \alpha^5$ can be found from 4.10

$$s = \frac{2^{nm} - 1}{2^m - 1} = \frac{2^4 - 1}{2^2 - 1} = 5 \Rightarrow \beta = \alpha^5, \qquad (4.11)$$

Substituting Eqs. 4.8, 4.9 and 4.11 into 4.4 will give

$$A = a_{00} + a_{01}\alpha^5 + a_{10} + a_{11}\alpha^6 \qquad (4.12)$$

Using $x^4 + x + 1$ as the irreducible polynomial for $GF(2^4)$,

$$\alpha^4 + \alpha + 1 = 0 \Rightarrow \alpha^4 = \alpha + 1 \qquad (4.13)$$

and

$$\Rightarrow \alpha^5 = \alpha^2 + \alpha \qquad (4.14)$$

$$\Rightarrow \alpha^6 = \alpha^3 + \alpha^2 \tag{4.15}$$

Substituting Eqs. 4.14 and 4.15 into 4.12 will give the representation of A in Galois field $GF(2^4)$:

$$A = a_{00} + (a_{01} + a_{10})\alpha + (a_{01} + a_{11})\alpha^2 + a_{11}\alpha^3 \tag{4.16}$$

$$\begin{aligned} \Rightarrow a_0 &= a_{00} \\ \Rightarrow a_1 &= a_{01} + a_{10} \\ \Rightarrow a_2 &= a_{01} + a_{11} \\ \Rightarrow a_3 &= a_{11} \end{aligned} \tag{4.17}$$

This will form the following matrix equation:

$$\begin{bmatrix} ao \\ a1 \\ a2 \\ a3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix} \tag{4.18}$$

Then, our conversion matrix T will be

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.19}$$

The Matrix T gives the representation of an element in the binary field $GF(2^4)$ given its representation in the composite field $GF((2^2)^2)$. The inverse transformation, i.e., the conversion from $GF(2^4)$ to $GF((2^2)^2)$, requires the computation of T^{-1},

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.20}$$

The irreducible polynomial F(x) for $GF((2^n)^m)$ can be found from

$$F(x) = (x + \alpha)(x + \alpha^{2^n})(x + \alpha^{2^{2n}})....(x + \alpha^{2^{(m-1)n}}) \tag{4.21}$$

For our $GF((2^2)^2)$, the irreducible polynomial becomes

$$\begin{aligned} F(x) &= (x+\alpha)(x+\alpha^4) \\ \Rightarrow &= x^2 + (\alpha+\alpha^4)x + \alpha^5 \end{aligned} \tag{4.22}$$

β **representation**	**Polynomial representation**
β^∞	0
β^0	1
β^1	x
β^2	x+1

Table 4.3: β Values

α **representation**	**Polynomial representation for** $GF(2^4) a0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3$
α^∞	0
α^0	1
α^1	x
α^2	x^2
α^3	x^3
α^4	$1+x$
α^5	$x+x^2$
α^6	x^2+x^3
α^7	$1+x+x^2$
α^8	$1+x^2$
α^9	$x+x^3$
α^10	$1+x+x^2$
α^11	$x+x^2+x^3$
α^12	$1+x+x^2+x^3$
α^13	$1+x^2+x^3$
α^14	$1+x^3$

Table 4.4: α Values

Using Table:4.3 and the irreducible polynomial x^4+x+1 for $GF(2^4)$, we can write:

$$\alpha + \alpha^4 = \alpha + \alpha + 1 \tag{4.23}$$

Substituting Eqs. 4.11 and 4.23 into Eq. 4.22, our irreducible polynomial for $GF((2^2)^2)$ becomes

$$\begin{aligned} F(x) &= x^2 + x + \beta \\ F(x) &= x^2 + x + 2 \end{aligned} \tag{4.24}$$

4.2.3 The Relation Between $GF(2^8)$ and $GF((2^4)^2)$ [12]

AES has adopted $m(x) = x^8 + x^4 + x^3 + x + 1$ as its field polynomial. Although such a polynomial is an irreducible but it is not a primitive one. Fortunately, with the field isomorphism property, we can map elements in $GF(2^8)$ as shown in [4] to the composite field $GF((2^4)^2)$ based on the polynomial $w(x) = x^2 + x + {}^14$, where ${}^14 = \{09\}$ denotes the element in $GF(2^4)$ of which $I(x) = x^4 + x + 1$ is the primitive irreducible polynomial. Let D be an element in $GF(2^8)$ and A be an element in $GF((2^4)^2)$, then $A = [T]D$ and $D = [T]^1A$

where

$$T = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{4.25}$$

and

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{4.26}$$

Here $[T]$ and $[T]^1$ are the field transformation matrices. The upper-left element in the above matrices denotes the least significant bit. In the composite field, let a byte-format data be expressed as

$$A = \{pq\} = px + q \tag{4.27}$$

4.2.4 Inversion

The inversion of A, say

$$B = A^{-1} = sx + t \tag{4.28}$$

The Extended Ecludian Algorithm as shown in the Pseudo Code is used to compute the inverse

Pseudo Code 6 Pseudo Code for calculating Field inversion in $GF(2^8)$

```
g8inverse( byte in[a], byte out[b] )
begin
    Nibble a1,a0,m4,m3,m2,m1,m0
    where irreducible polynomial is [0x01] x2 + [0x01] x + [0x09]
    Nibble d2=0,d1=0,d0=0,e1=0,e0=0,r1=0,r0=0
    Nibble tem=0x09
    (a1, a0) = Field Conversion from G8 to G4 ( a )
    if ( a1 != [0x00] ) {case where coefficient of x is not zero}
        d1 = g4divider( 0x01, a1) {field division implemented by further dividing
it into composite field.}
        r1 = g4multiplier (d1, a0) {field multiplication implemented by
further dividing it into composite field.}
        tem = r1 ⊕ [0x01]
        d0 = g4divider (tem, a0)
        r0 = g4multiplier (d0, a0)
        r0 = r0 ⊕ [0x01] {remainder}
        if ( r0 != 1 ) {case if remainder is non-zero}
            e1 = g4divider (a1, r0)
            a0 = a0 ⊕ [0x01]
            e0 = g4divider (a0, r0)
            m0 = g4multiplier (d0, e0)
```

```
                m1 = g4multiplier (d0, e1)
                m2 = g4multiplier (d1, e0)
                m3 = g4multiplier (d1, e1)
                m4 = g4multiplier (m3, [0x09])
                d1 = m1 ⊕ m2 ⊕ m3
                d0 = m0 ⊕ m4 ⊕ [0x01]
        else {case where coefficient of x is zero}
                d2 = g4divider ([0x01], a0)
                d1 = g4divider ([0x01], a0)
                tem = [0x09] ⊕ [0x01]
                d0 = g4divider (tem, a0)
                m4 = g4multiplier (d2, [0x09])
                d1 = d1 ⊕ d2
                d0 = d0 ⊕ m4
        b = Field Conversion from G8 to G4 (d1, d0)
end
```

Pseudo Code 7 Pseudo Code for calculating g4divider and g4multiplier

```
g4divider ( Nibble in[a], Nibble in[b], Nibble out[c] )
begin
        Nibble d; {{A / B} = {A X B-1}}
        d = g4inverse (b) { implemented by table look-up}
        c = g4multiplier(a, d)
end
g4multiplier ( Nibble in[a], Nibble in[b], Nibble out[c] )
begin
        Nibble a1,a0,b1,b0,c1,c0,a1b1,a1b0,a0b1,a0b0
        (a1, a0) = Field Conversion from G4 to G2 ( a )
        (b1, b0) = Field Conversion from G4 to G2 ( b )
        a0b0 = g2multiplier (a0, b0) {implemented by table look-up}
        a0b1 = g2multiplier (a0, b1)
        a1b0 = g2multiplier (a1, b0)
        a1b1 = g2multiplier (a1, b1)
        c0 = g2multiplier (a1b1, [0x02])
        c0 = c0 ⊕ a0b0
        c1 = a1b1 ⊕ a1b0 ⊕ a0b1
        c = Field Conversion from G2 to G4 (c1, c0)
end
```

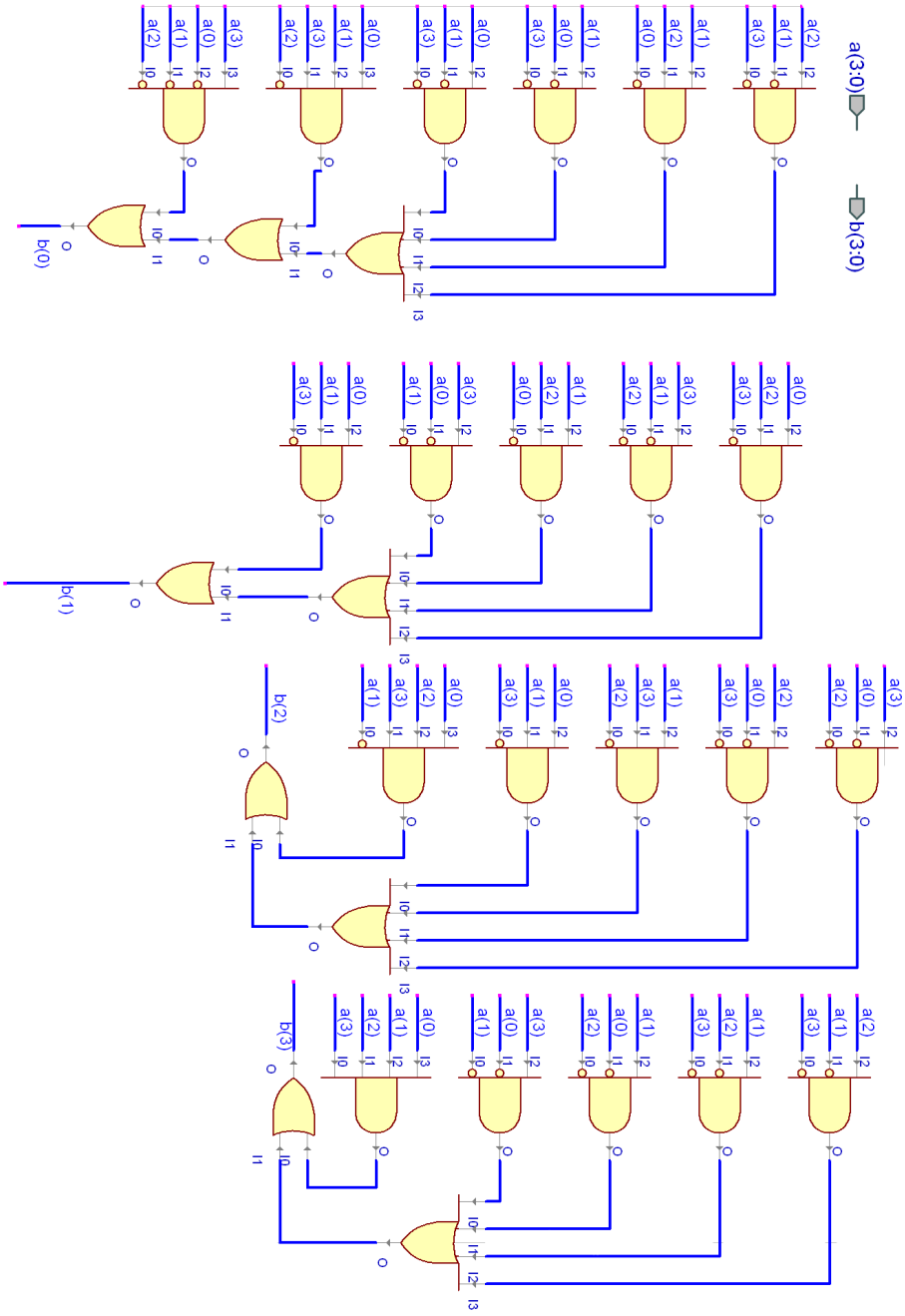


Figure 4.1: Logic implementation of G(4) inverse

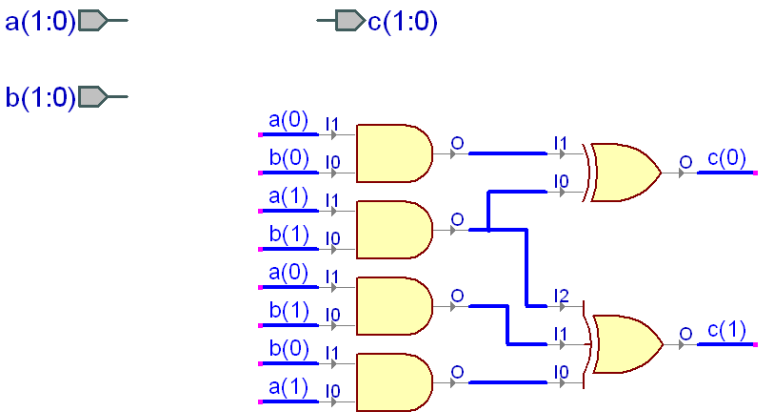


Figure 4.2: Logic implementation of G(2) multiplier

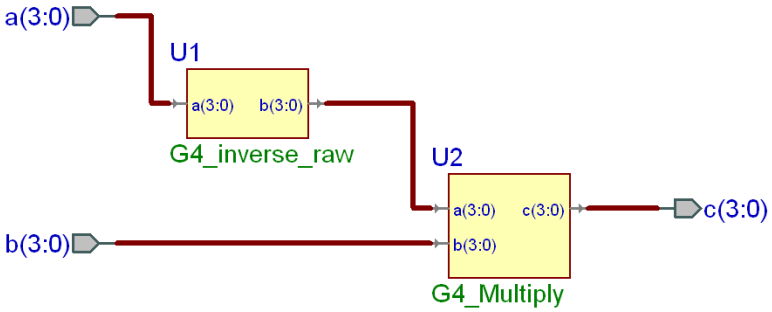


Figure 4.3: Logic implementation of G(4) division

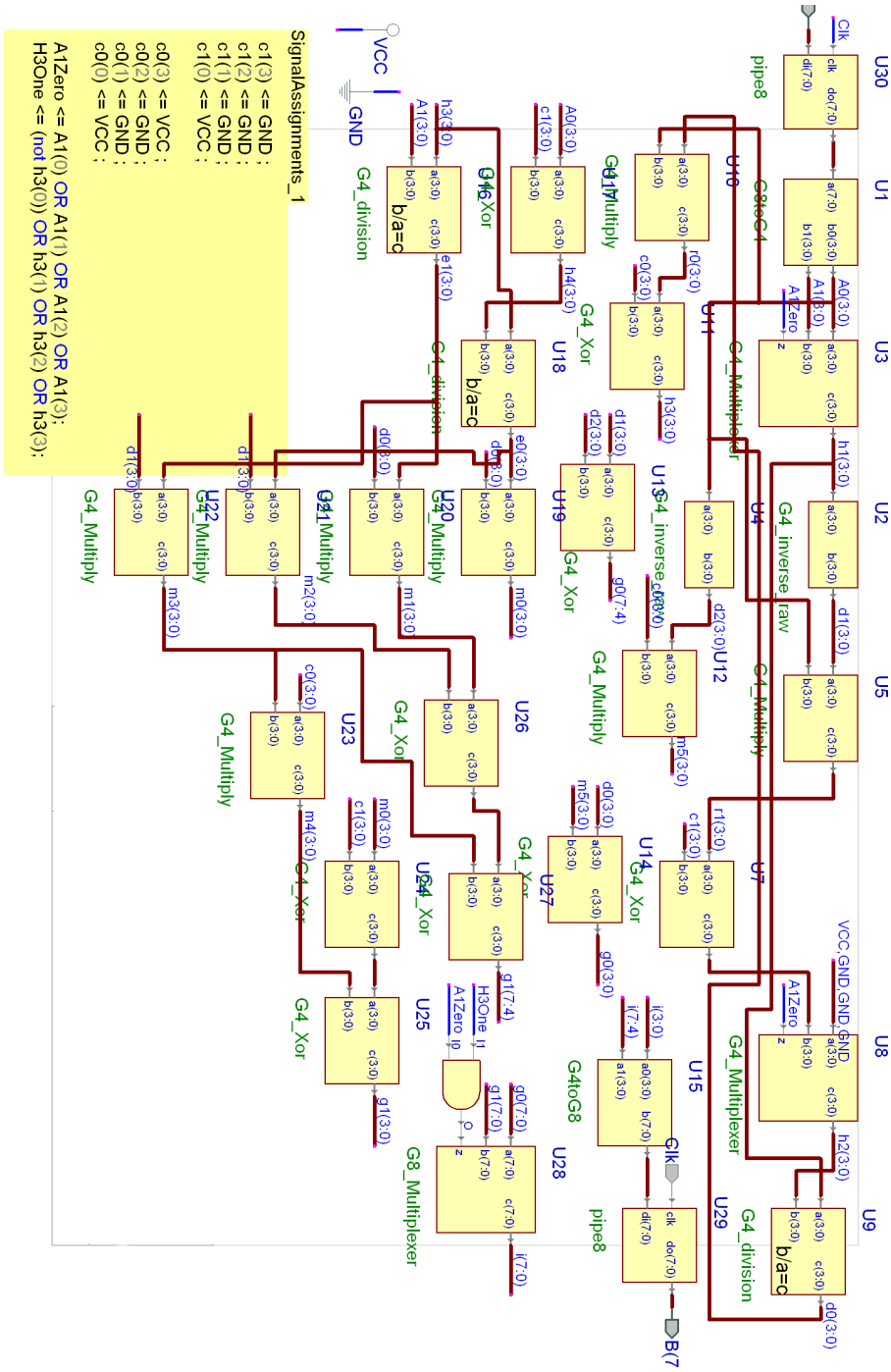


Figure 4.4: Logic implementation of G(8) inversion

x	x^{-1}**in GF(4)**
0	0
1	1
2	9
3	e
4	d
5	b
6	7
7	6
8	f
9	2
a	c
b	5
c	a
d	4
e	3
f	8

Table 4.5: Look up table for G(4) inverse

4.2.5 $G(2^4)^2$ Inversion

There is an another way to calculate G(24)2 inverse. Let suppose

$$B = A^{-1} \quad where \quad A, B \quad in \quad GF(2^4)^2 \tag{4.29}$$

so

$$A = a_1 X + a_0 \tag{4.30}$$

and

$$B = b_1 X + b_0 \tag{4.31}$$

as

$$B = A^{-1} \tag{4.32}$$

$$\begin{aligned} &\Rightarrow \quad AB = 1 \\ &\Rightarrow \quad (b_1 X + b_0)(a_1 X + a_0) = 1 \\ &\Rightarrow \quad b_1 a_1 X^2 + (b_1 a_0 + b_0 a_1) X + b_0 a_0 = 1 \end{aligned} \tag{4.33}$$

as

$$X^2 + X + 9 = 0 \quad is\ the\ irreducible\ polynomial$$

$$\begin{aligned} &\Rightarrow \quad [b_1 a_1 X^2 + (b_1 a_0 + b_0 a_1) X + b_0 a_0] mod_{X^2 + X + \beta^{14} = 1} \\ &\Rightarrow \quad [b_1 a_1 X^2 + (b_1 a_0 + b_0 a_1) X + b_0 a_0] mod_{X^2 + X \beta^{14} = 1} \\ &\Rightarrow \quad (b_1 a_0 + b_0 a_1 + b_1 a_1) X + (b_0 a_0 + b_1 a_1 \beta^{14}) = 1 \\ &\Rightarrow \quad (b_1 a_0 + b_0 a_1 + b_1 a_1) = 0 \\ &\Rightarrow \quad (b_0 a_0 + b_1 a_1 \beta^{14}) = 1 \\ &\Rightarrow \quad b_1 = a_1 (a_0^2 + a_1 a_0 + a_1^2 \beta^{14})^{-1} \qquad (4.34) \\ &\Rightarrow \quad b_0 = (a_1 + a_0)(a_0^2 + a_1 a_0 + a_1^2 \beta^{14})^{-1} \qquad (4.35) \end{aligned}$$

Circuit based on this approach is very promising and will reduce the code by orders. This Inversion only required two $GF(2^4)$ square three $GF(2^4)$ multiplication and one $GF(2^4)$ inversion. That can be implemented with very few no of gates.

$GF(2^4)$ square

$GF(2^4)$ square can be implemented as an combination logic with the help of only two xor.

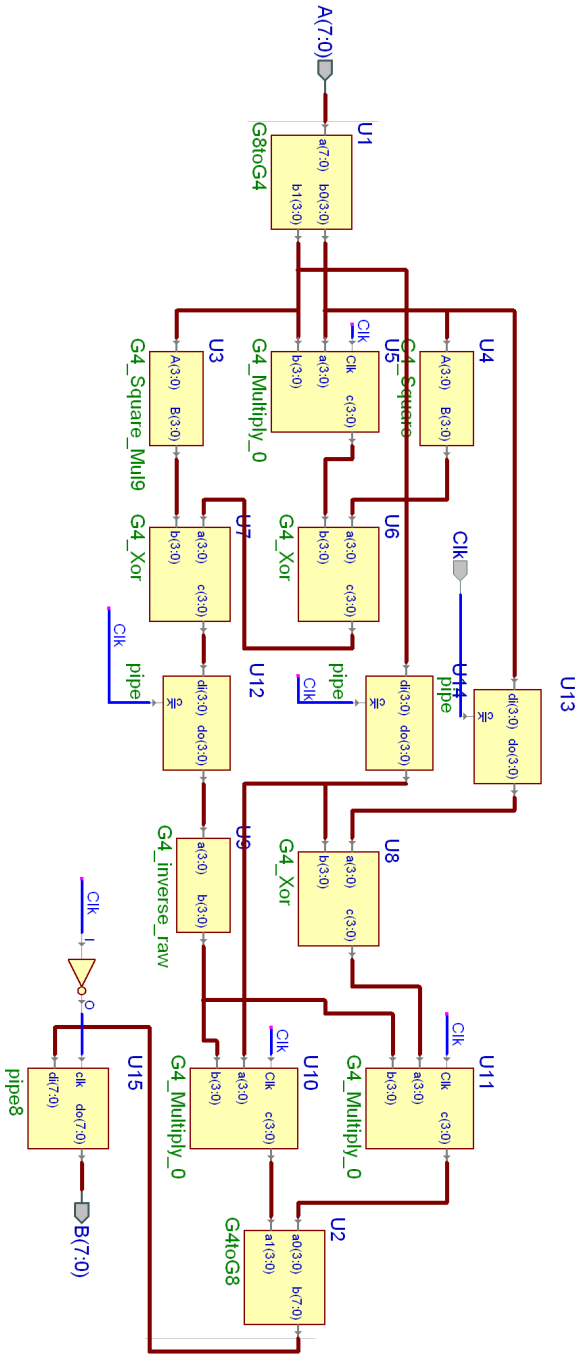


Figure 4.5: G(8)-1 pipe–lined design

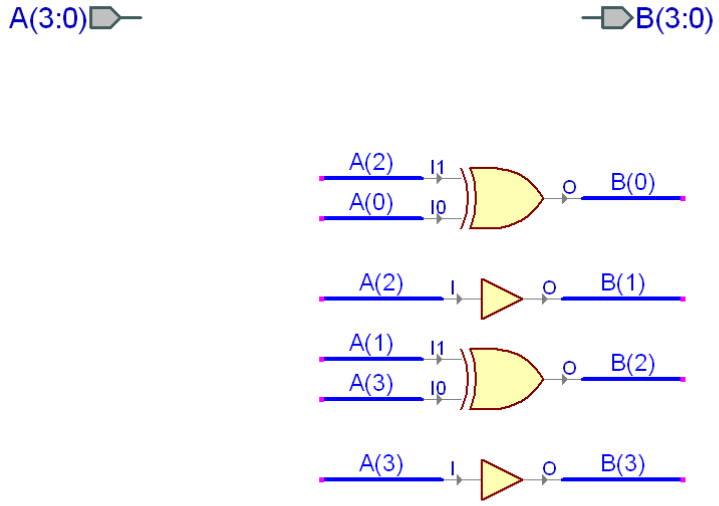


Figure 4.6: Gate Level Implementation of $GF(2^4)$

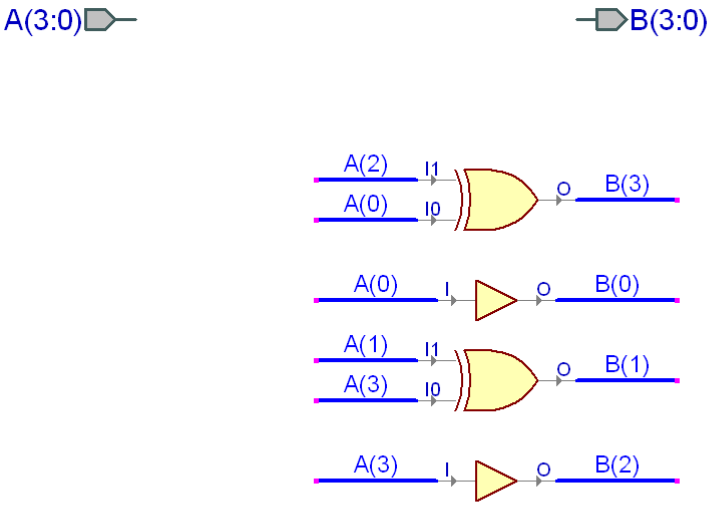


Figure 4.7: Gate Level Implementation of $(GF(2^4)^2)\beta^{14}$

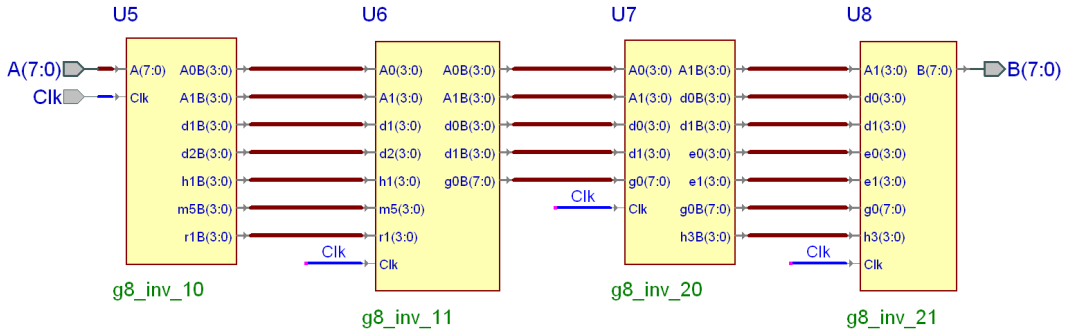


Figure 4.8: $G(2^8)$ inverse After Introducing Pipelining of Order 4

$GF(2^4)$ **square X** $\beta^1 4$

$A_1{}^2\beta^{14}$ can also be implemented as an combination logic with the help of only two xor.

4.2.6 Pipe-lining

Galois field inversion gives us a very compact design. But design have greater degree of Logic Levels. This design has delay of 13.9 nano seconds. By introducing Pipe-lining Delay is reduced to as low as 3.61 nano Seconds. Refer table:**??**. At the cost of some small increased gate size and complexities.

4.2.7 Affine/Inverse Affine Transformation

In matrix form, the affine transformation element of the S-box can be expressed as:(see section 2.2)

$$b(x) = \{1F\}d(x)_{mod(x^8+1)} \oplus c(x) \tag{4.36}$$

where

$$c(x) = \{63\} = x^6 + x^5 + x + 1$$

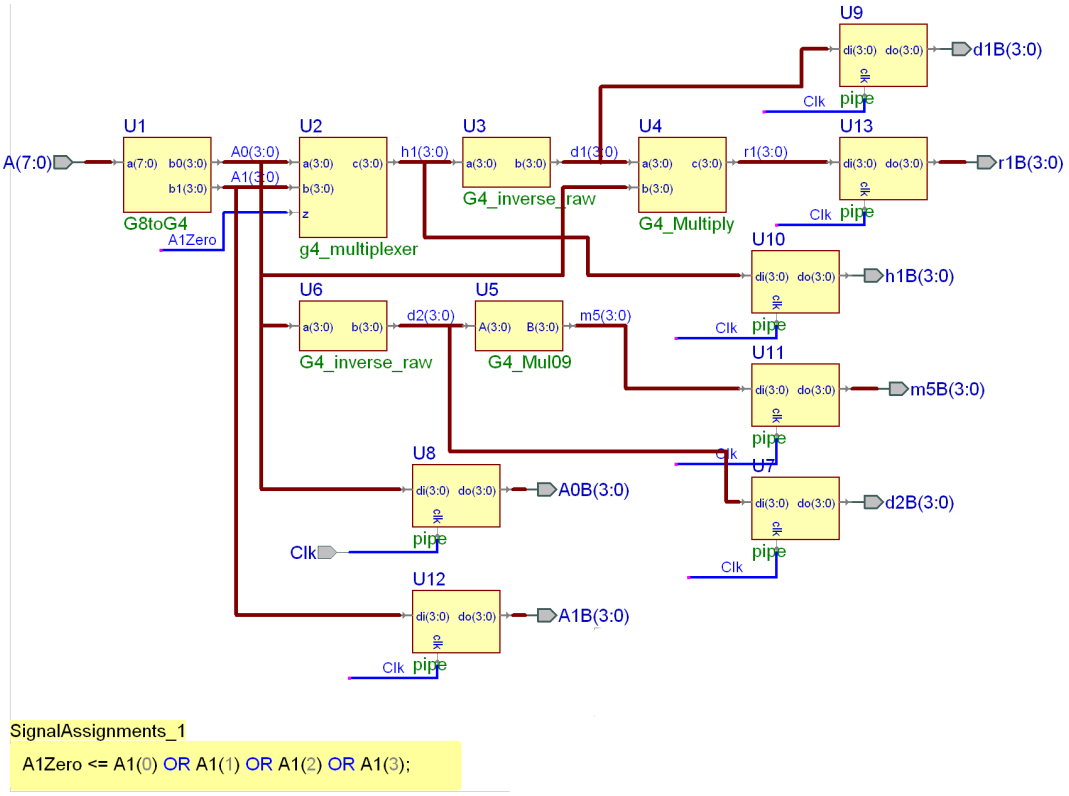


Figure 4.9: G8inverse10

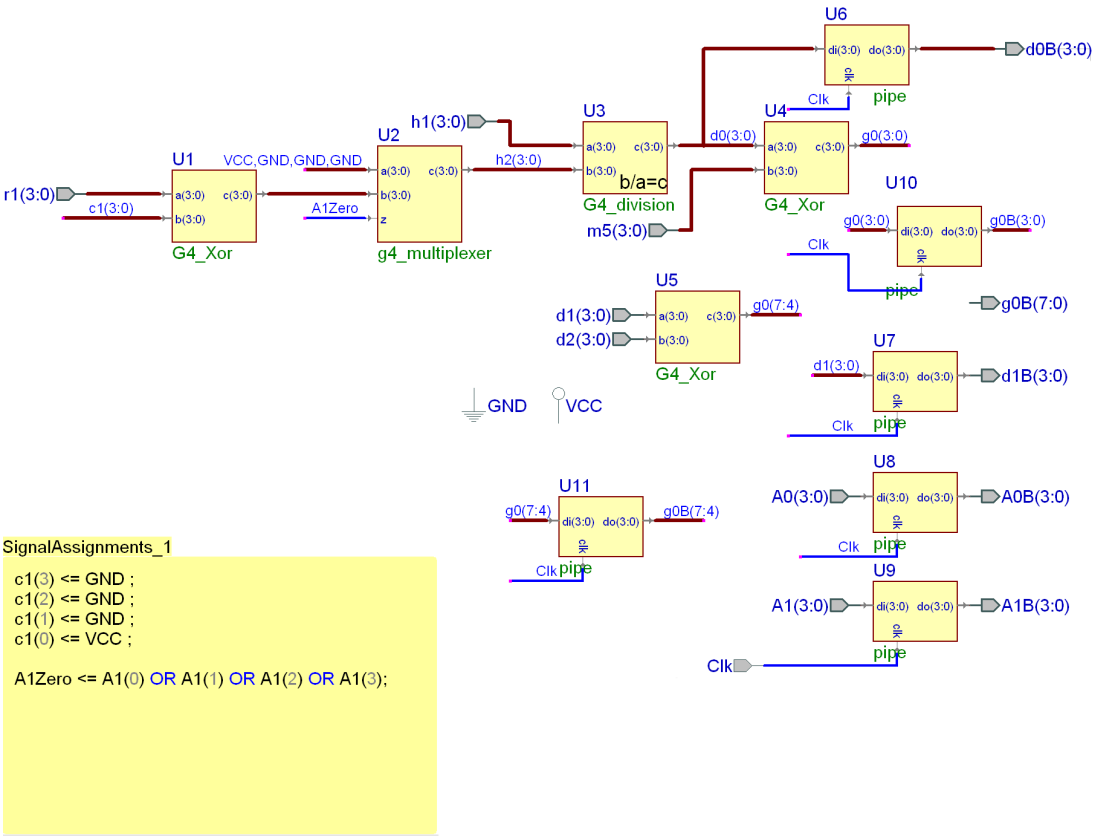


Figure 4.10: g8inverse11

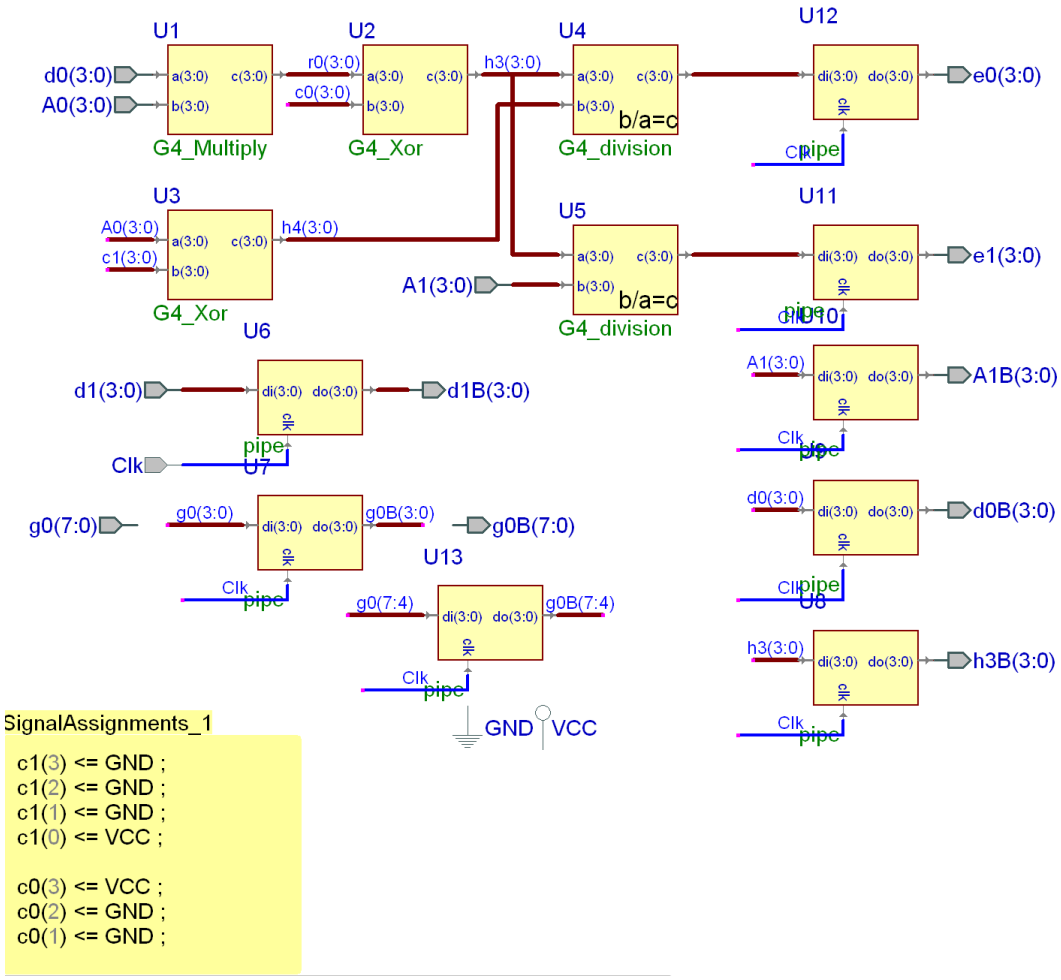
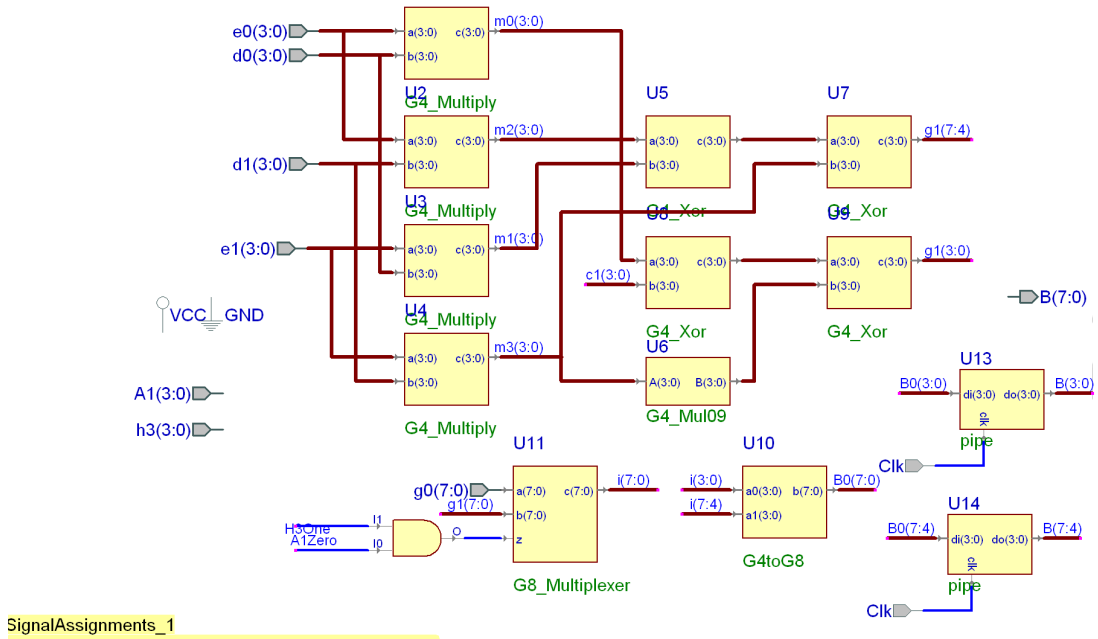


Figure 4.11: g8inverse20



SignalAssignments_1

```
c1(3) <= GND ;
c1(2) <= GND ;
c1(1) <= GND ;
c1(0) <= VCC ;
H3One <= (not h3(0)) OR h3(1) OR h3(2) OR h3(3);
A1Zero <=A1(0) OR A1(1) OR A1(2) OR A1(3);
```

Figure 4.12: g8inverse21

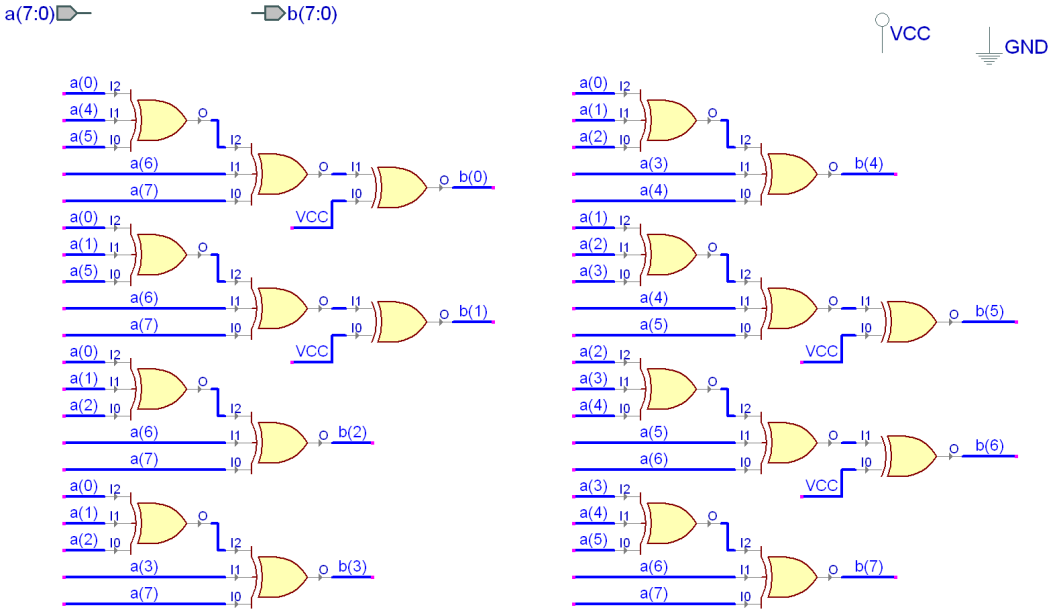


Figure 4.13: Logic implementation of Affine Transform

$$\begin{bmatrix} bo \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{4.37}$$

the inverse affine transformation element of the S-box can be expressed as:

$$b(x) = \{4A\}d(x)_{mod(x^8+1)} \oplus c(x) \tag{4.38}$$

where

$$c(x) = \{05\} = x^2 + 1$$

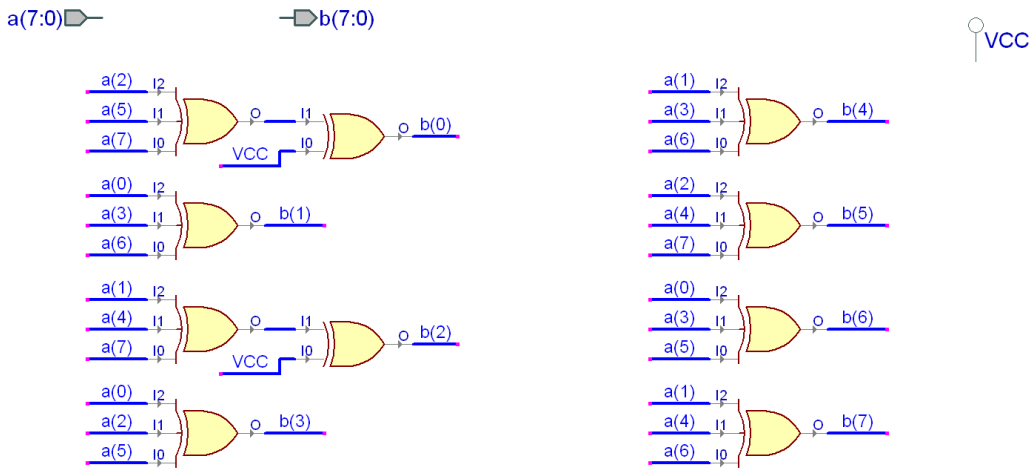


Figure 4.14: Logic implementation of Inverse Affine Transform

$$\begin{bmatrix} bo \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.39}$$

G4toG8 transform matrix and Affine Transform matrix can be mapped to a single transform matrix thus further reducing the size and delay of the Design.

$$AffineMapMatrix = G4toG8Transformation * AffineTransform \tag{4.40}$$

$$\begin{bmatrix} bo \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} \right) \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{4.41}$$

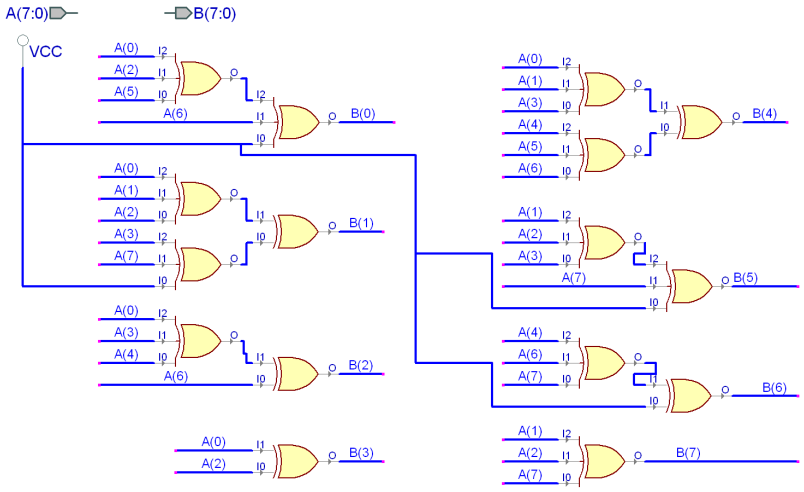


Figure 4.15: Logic implementation of Affine Map Matrix

$$\Rightarrow B = \begin{bmatrix} bo \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{4.42}$$

Similarly G8toG4 transform matrix and Inverse affine Transform Matrix can also be mapped.

$$InverseMapMatrix = Inverseaffinetransform * G8toG4Transformation \tag{4.43}$$

$$\begin{bmatrix} bo \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right) \tag{4.44}$$

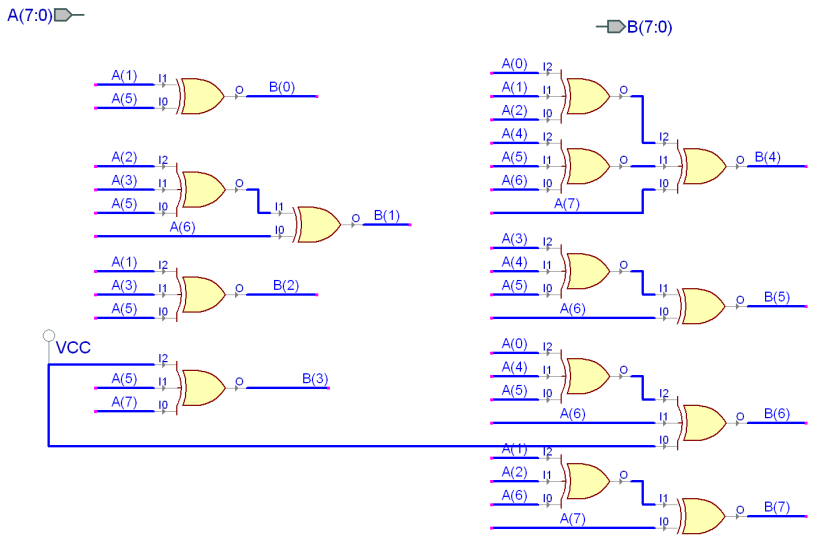


Figure 4.16: Logic implementation of Inverse Map Matrix

$$\Rightarrow B = \begin{bmatrix} bo \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{4.45}$$

4.2.8 Sbox Inverse Sbox

Sbox and Inverse Sbox has been implemented in a single block considering that at a given time either encryption or decryption process is taking place. In the design Sbox or Inverse Sbox functionality can be selected by "M"signal.(see figure:4.17) Further optimization can introduced by merging Affine Transformation and G4to8 Transformation as stated

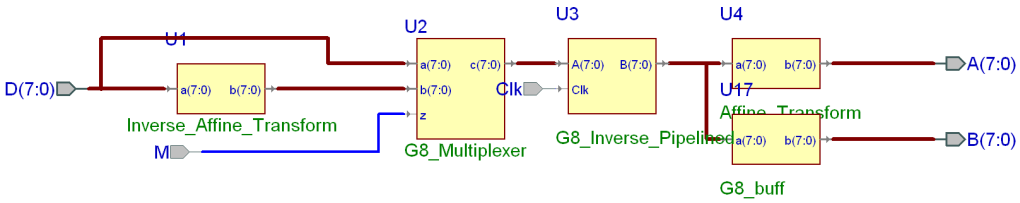


Figure 4.17: Logic implementation of Sbox

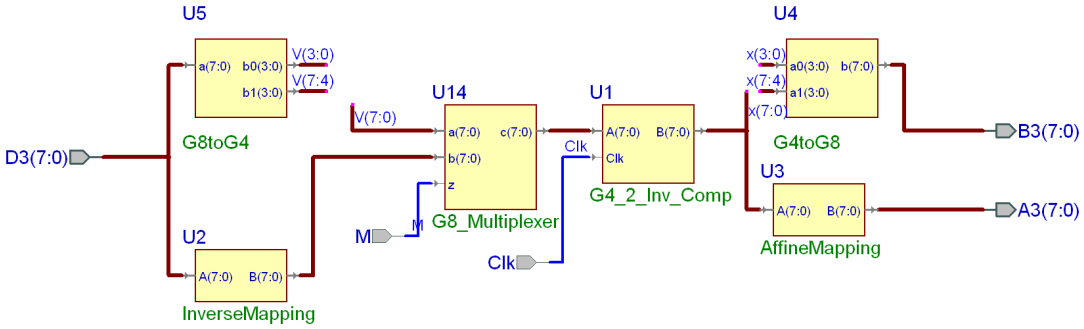


Figure 4.18: Derived design by merging Affine Transformation and G4to8 Transformation sbox _ invsbox _ 8

Chapter 5

Shift Row

Shift Row have been implemented with the help of bus multiplexers.

This process operates on an individual row with individual offset byte. The transform throughput is 32 bits per clock cycle and can be made pipelines for column order. For a wider data path (128-bit) or the higher throughput such as 128 bits per clock cycle, the switches SS0, SS1 and multiplexers are not necessary.

As discussed in the previous paragraph 128 bit design is also developed. In this design Shift Row is merged with the Substitution module. Refer to the figures. Thus simplifying the design and eliminating some multiplexers and reducing the size.

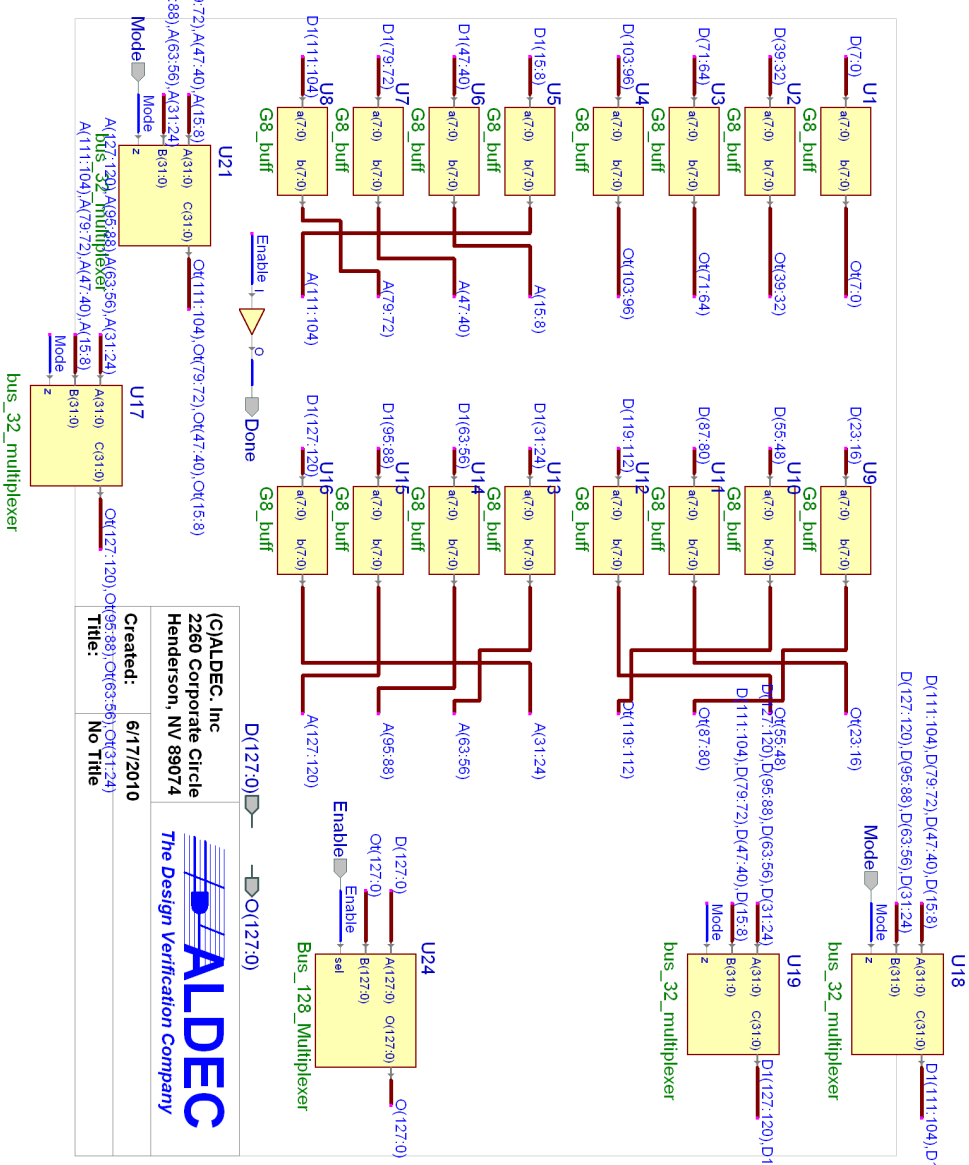


Figure 5.1: 128 bit Shift Row

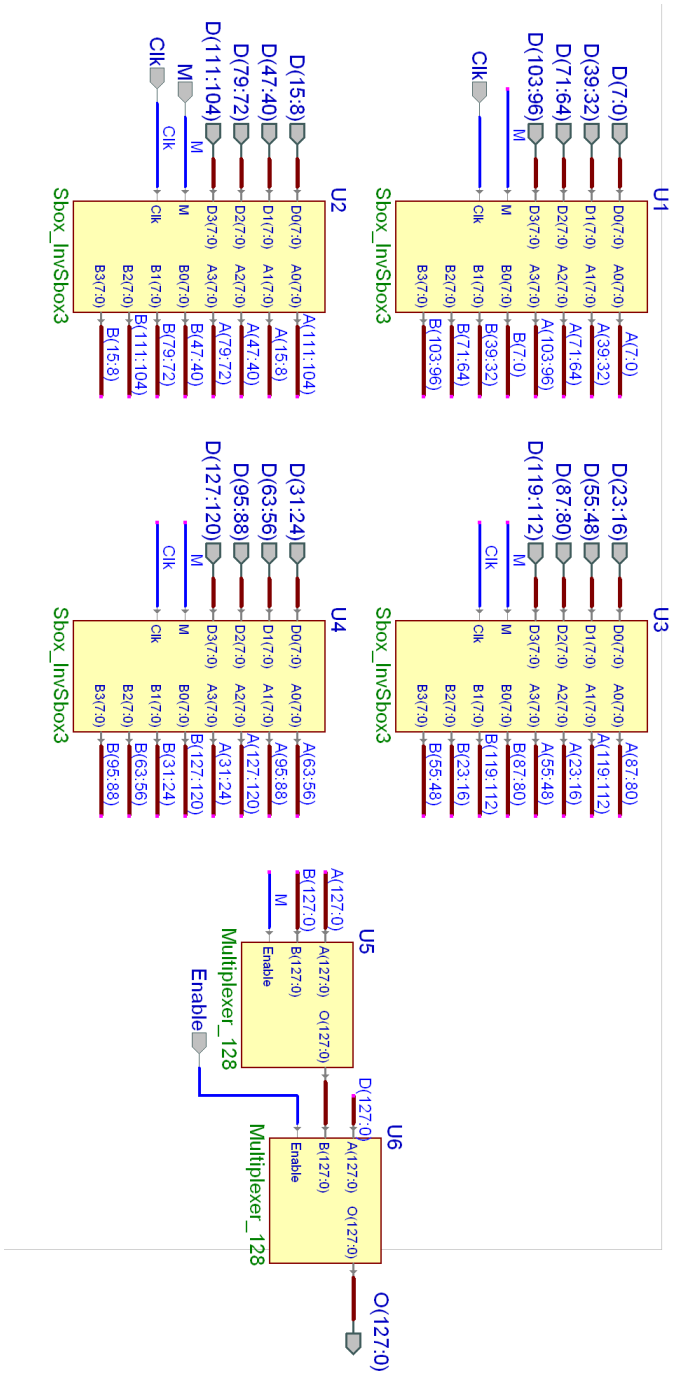


Figure 5.2: 128 bit Shift Row, Merged with Sbox (Eliminating Couple of multiplexers)

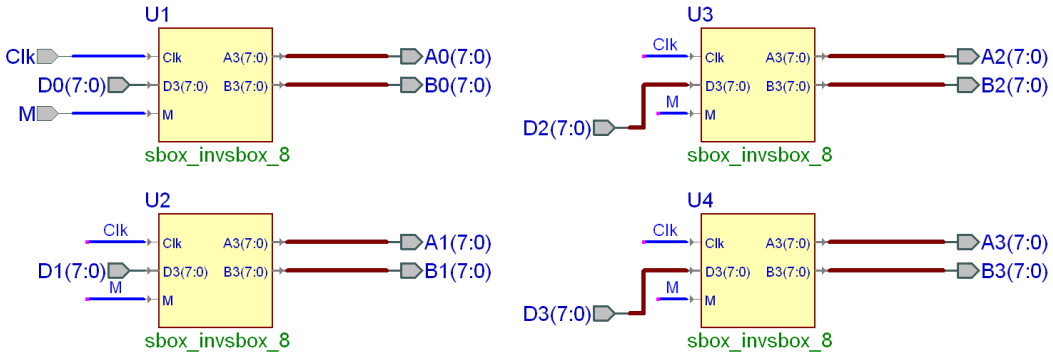


Figure 5.3: Sbox _ invSbox3

Chapter 6

Core Design

6.1 State

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 $s_{r,c}$ or s[r,c]. For this standard, Nb=4, i.e., $0 <$ c < 4

At the start of the Cipher and Inverse Cipher (described in Sec. 1.1 &1.7), the input array of bytes $in_0, in_1, ...in_{15}$ is copied into the State array. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output array of bytes out_0, out_1, out_{15}

Hence, at the beginning of the Cipher or Inverse Cipher, the input array, in, is copied to the State array according to the scheme:

$$s[r, c] = in[r + 4c] \quad for \quad 0 < r < 4 \quad and \quad 0 < c < Nb, \tag{6.1}$$

and at the end of the Cipher and Inverse Cipher, the State is copied to the output array out as follows:

$$out[r + 4c] = s[r, c] \quad for \quad 0 < r < 4 \quad and \quad 0 < c < Nb. \tag{6.2}$$

The four bytes in each column of the State array form 32-bit words, where the row number r provides an index for the four bytes within each word. The state

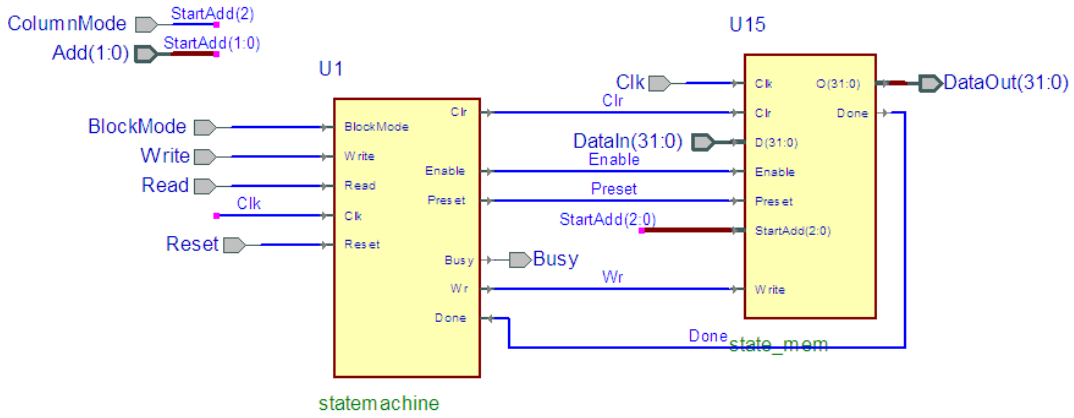


Figure 6.1: block diagram for State matrix

can hence be interpreted as a one-dimensional array of 32 bit words (columns), w0...w3, where the column number c provides an index into this array. Hence, the State can be considered as an array of four words, as follows:

$$w_0 = s_{0,0}s_{1,0}s_{2,0}s_{3,0} \tag{6.3}$$

$$w_2 = s_{0,2}s_{1,2}s_{2,2}s_{3,2} \tag{6.4}$$

$$w_1 = s_{0,1}s_{1,1}s_{2,1}s_{3,1} \tag{6.5}$$

$$w_3 = s_{0,3}s_{1,3}s_{2,3}s_{3,3} \tag{6.6}$$

Hence Data transactions from the State array is in two way.

32 bit Row Word

32 bit Column Word

Here State matrix is implemented in such a way that it behaves like a 32 bit read write memory buffer (device) consisting of 8 words. First four word represent 32 bit Row Word and other four word represent 32 bit Column Word.

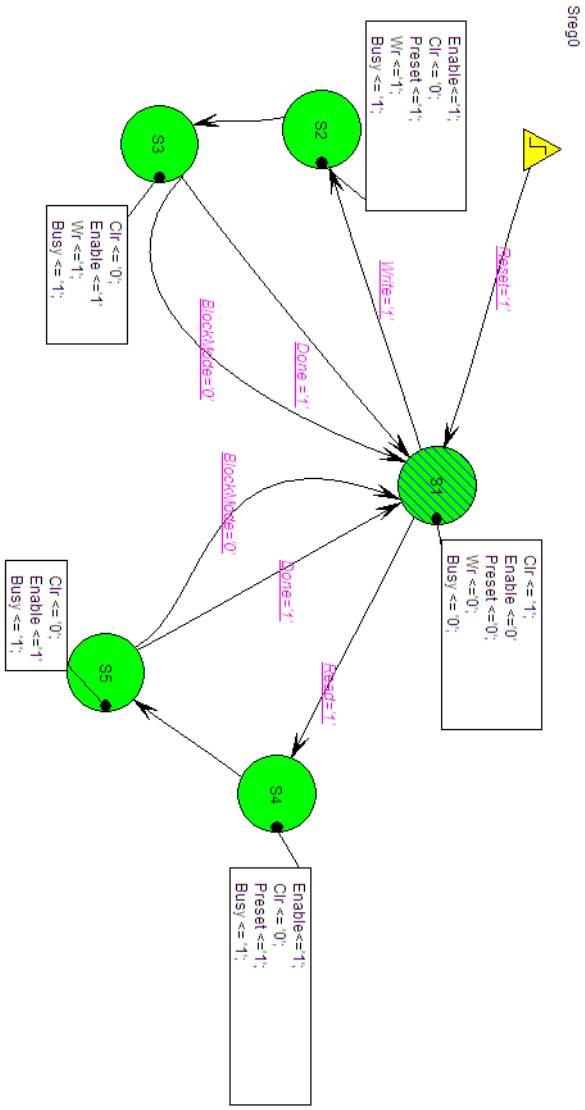


Figure 6.2: State machine for controlling the State Matrix for different operations

6.2 32 bit Core Design

Initially 32 bit core is designed. In this design each transformation takes 4 cycles. State buffer is designed in a special way so that it can work in three modes. (1) Column mode, (2) Row mode and (3) memory block mode. In memory block mode data is transferred in blocks to/from State from/to the main memory buffer.

Design has one accumulator named out_ register. Transformation can be perform in either way (1) State - core transformation - out_ register, (2) out_ register - core transformation - State, By the help of this shuttle mechanism lot of time is being saved from saving the result again in the State.

All the transformations (sub byte, Mix column .etc) are implemented in aes_ modules as an memory device. That means different address of aes_ modules corresponds to different transformation.

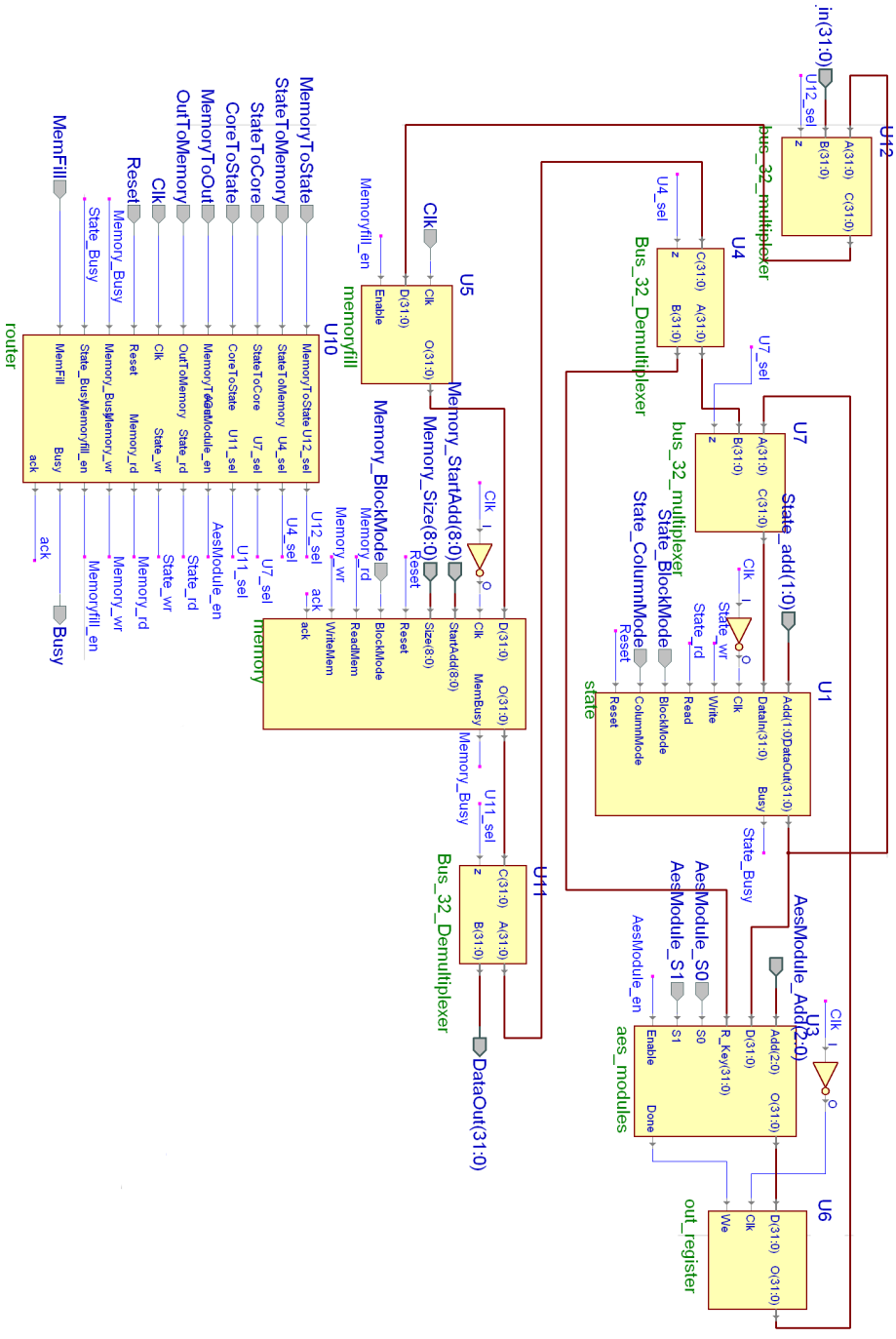


Figure 6.3: 32 bit AES Core Design

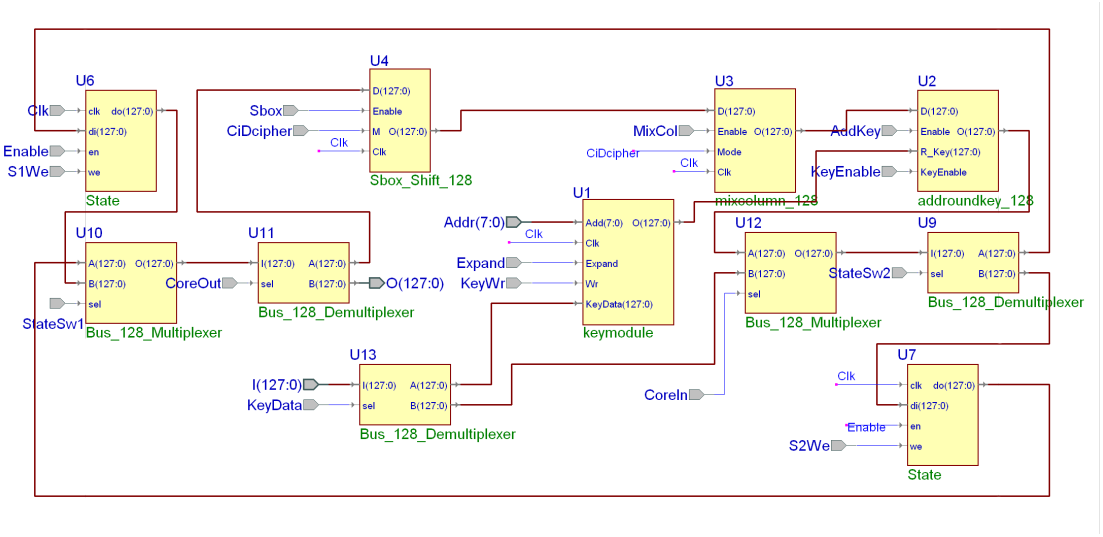


Figure 6.4: 128 bit AES Core Design

6.3 128 bit Core Design

Further for higher throughput a new design with 128 bit wide data bus is designed. In this each transformation is done in a state array in a single cycle. Thus increased throughput and reduced complexity.

6.4 Key Expansion Module

Based on Algorithm stated in pseudocode[2] section[2.6] Key Expansion core has bean generated. It has a 32 bit wide bus. As key expansion is required only once in a session key can be expended before starting encryption or decryption. Key can also be expended on the go (live mode) while doing encryption / decryption.

For 128 bit core design key expansion is done before encryption/ decryption.

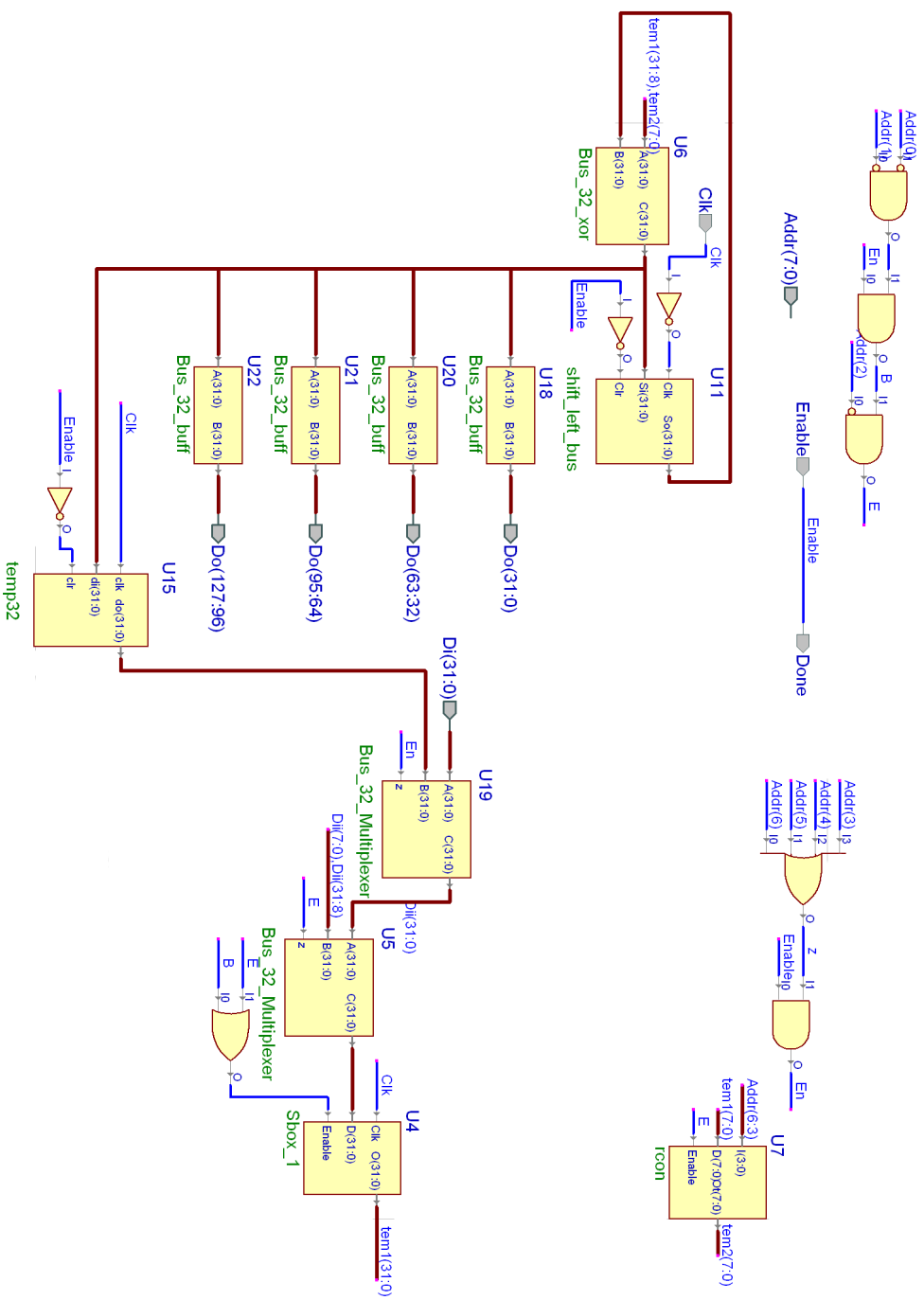


Figure 6.5: Key_Expansion Module

Chapter 7

Results

Various Design of S-boxes are studied and there size verses delay comparisons are done on the basis of [9].

further size delay comparison is done between different designs of S-boxes which are developed in this project in table 7.1 figure 7.1

Combined comparison is done between available and developed S-boxes in figure 7.1. Where figure shows S-box implementation for different architectures[9]. Green and Red dot represents designs developed during this projects and are based on Galois field inversion. Blue dots represent designs from a paper [9] where Satoh discussed different designs methodology. These design are basically ASIC Designs that's why delays are very less in contrast with the designs in this project as they are FPGA based. Implementing these designs in ASIC may further reduce the delays. Red dots represents the most optimum design with respect to size and delay.

Figure 7.1: Delay Vs CLB Comparison Between Different Architectures

SBox	**Total Slice**	**1/delay Mhtz**	**Throughput per Unit Size = Frequency/CLB**
Sbox_Euclidean	140	99.80	0.71
Sbox_Euclidean_1_Pipeline	149	88.50	0.59
Sbox_Euclidean_2_Pipeline	178	64.85	0.36
Sbox_Euclidean_4_Pipeline	211	215.05	1.02
Sbox_Optimized	48	71.94	1.50
Sbox_Optimized_2_Pipeline	79	150.83	1.91
Sbox_Optimized_4_Pipeline	81	277.01	3.42
S_Inverse_box_Optimized _4_Pipeline	47.5	277.01	5.83
Sbox_Inverse_Euclidean _4_Pipeline	110	215.05	1.96
Composite Field	354	456.62	1.29
PPRM	2148	990.09	0.46
SOP	1567	1298.7	0.83
LUT	1528	1428.57	0.93
BDD	1399	1449.28	1.04
TBDD	2818	2325.58	0.83

Table 7.1: Size, Max Frequency, And Throughput per Unit Size comparison of different implementation

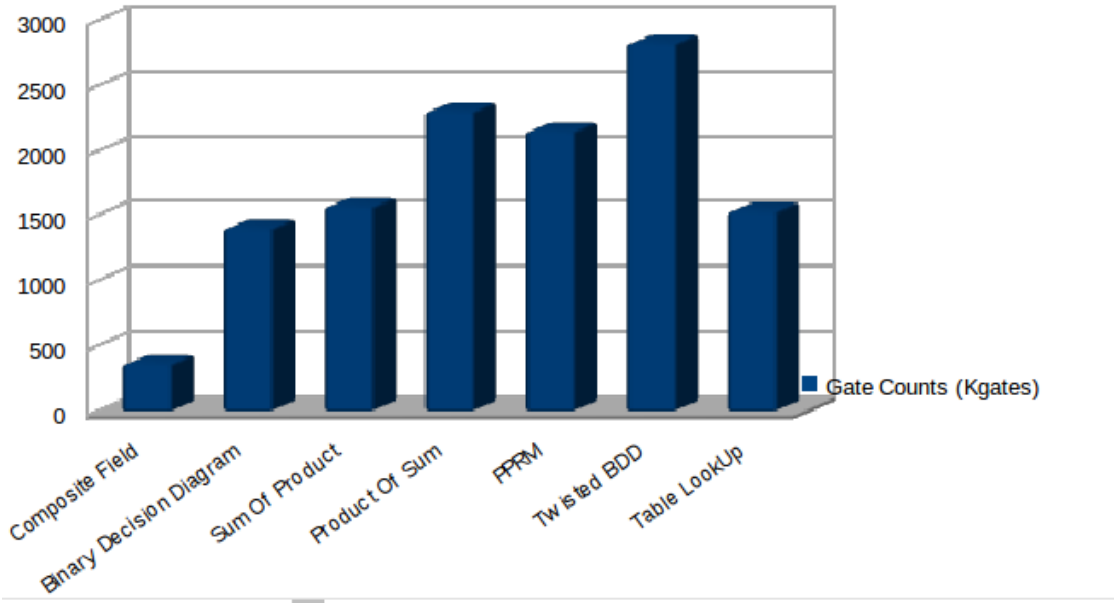


Figure 7.2: Size Comparison Between Different Architectures

In figure 7.3 comparison between throughput and size is done between all the S-box designs. In this comparison it is clearly illustrated that throughput of the two developed designs is maximum.

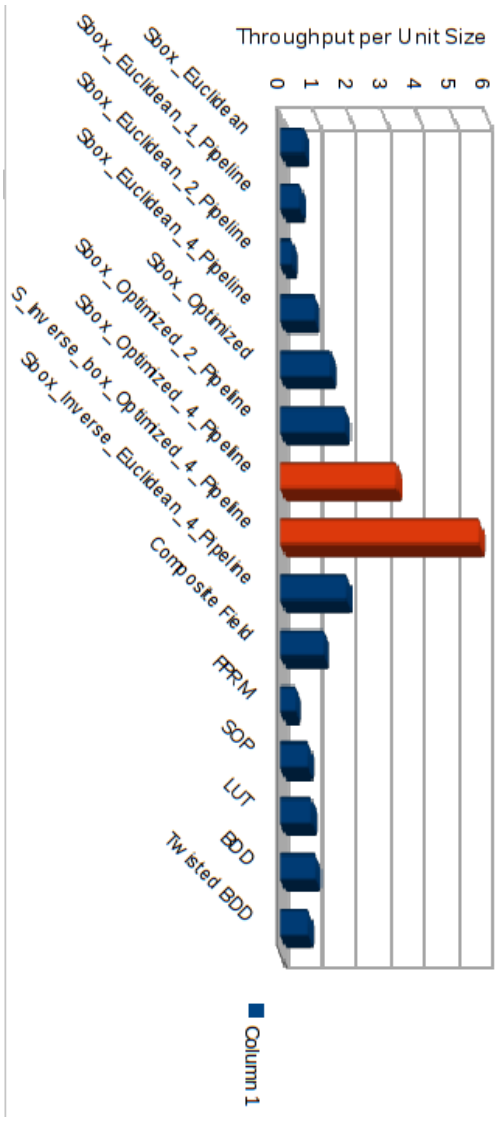


Figure 7.3: Throughput / unit Size Comparison between Different Architectures

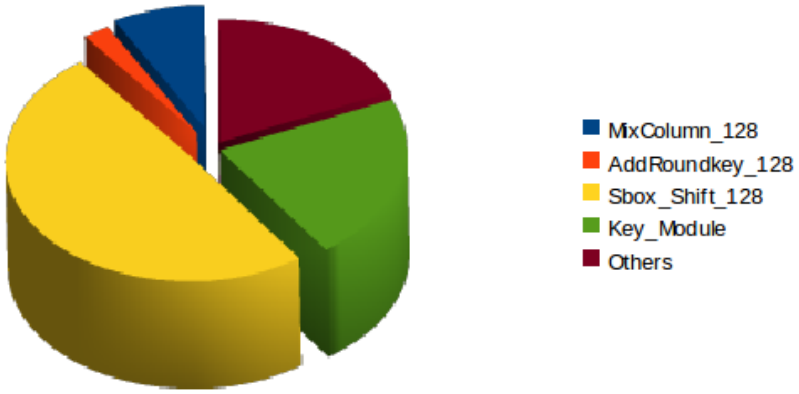


Figure 7.4: Module wise Size Distribution

Module	**CLB Slice**	**Slice Flip-flop**	**Delay**	**Max Levels of Logic**
MixColumn_128	254		2.44	7
AddRoundkey_128	74		2.44	3
Sbox_Shift_128	879	704	3.61	5
Key Module	529	171	3.61	5
Core	1930	1276	3.86	7

Table 7.2: Size and Delay comparison of different Core modules

In figure 7.4 Table 7.2 comparison between throughput and size is done between all the S-box designs. In this comparison it is clearly illustrated that throughput of the two developed designs is maximum.

7.1 Throughput Calculation

7.1.1 For 128 bit Bus Size

$$Max.\ Delay = 3.86\ nano\ Sec. = 3.86 \times 10^{-9}\ Sec$$
$$Max.\ Frequency = \frac{1}{3.86 \times 10^{-9}}\ Hz$$
$$. = 259 \times 10^{6}\ Hz$$
$$Throughput = \frac{[BusWidth] \times [Frequency]}{[No\ Of\ Rounds]}$$
$$. = \frac{128 \times 259 \times 10^{6}}{10}$$
$$. = 3315.2 \times 10^{6}\ bits/sec$$

7.1.2 For 32 bit Bus Size

$$Max.\ Delay = 3.86\ nano\ Sec. = 3.86 \times 10^{-9}\ Sec$$
$$Max.\ Frequency = \frac{1}{3.86 \times 10^{-9}}\ Hz$$
$$. = 259 \times 10^{6}\ Hz$$
$$Throughput = \frac{[BusWidth] \times [Frequency]}{[No\ Of\ Rounds]}$$
$$. = \frac{32 \times 259 \times 10^{6}}{10}$$
$$. = 828.8 \times 10^{6}\ bits/sec$$

	Device	**CLB Slices**	**Throughput (Mbits/s)**
Ichikawa [7]	VLSI		1950
Weeks [2]	VLSI		5163
Lutz [6]	VLSI		2263
Elbirt [4]	xcv1000	9004	1940
McLoone and McCanny [8]	XCV3200E	7576	3239
E Rodriguez-Henriquez [11]	XCV2000E	5677+80 Brams	4129
This Design	xc4vfx12	3206	3315.2

Table 7.3: Size and Delay comparison of different Core Implementations studied with this design

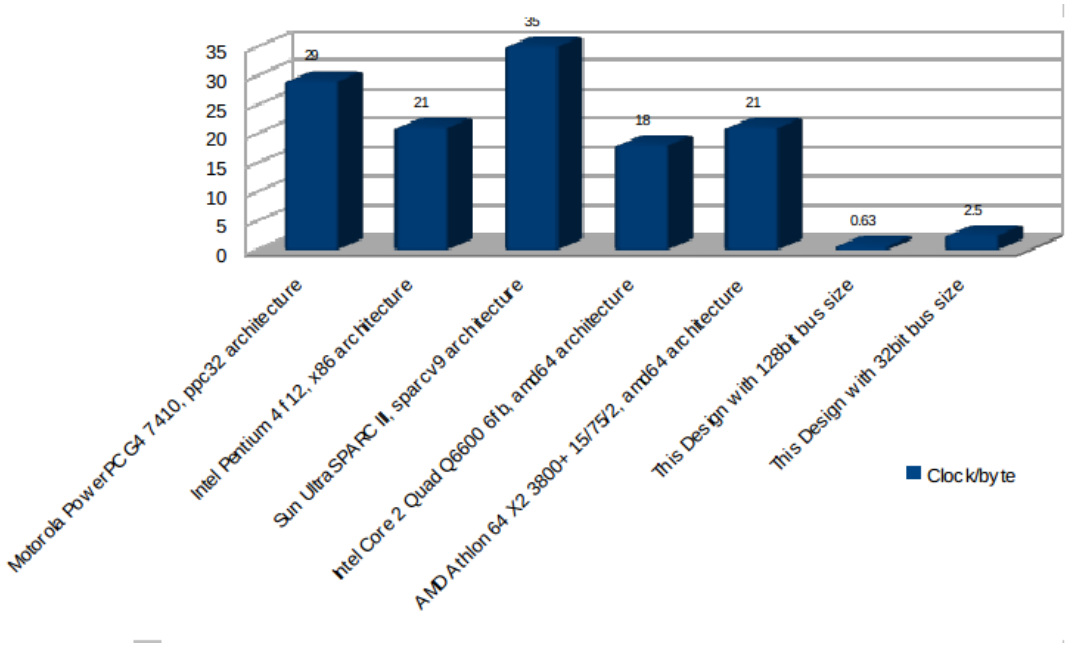


Figure 7.5: Comparison of Cycles / unit byte encryption between different Processors Architectures

7.2 Comparison with Micro-controllers

Micro-controllers and micro processors are general purpose devices generally used for general purpose computing. They are made to be a universal state-machines. Generally they have ALU , Multipliers Internal registers and memory catches hence lot of equivalent gates. A typical P4 Processor contains 55 Million gates. In contrast with FPGA Design those are generally application specific using much less resources.

As controllers are ASIC they works on Gigahertz Clocks whereas a typical FPGA works on 100 Mhz Clock. So in order to compare between the two we have to device a different bench mark.

Comparison done is based on number of clocks a core takes to encrypt a single byte (see Table 7.4 figure 7.5). Throughput are taken for different available architectures with a standard Openssl Speed Aes tool[3].

Core	**Clocks/Byte**	**Throughput Mbit/sec**
Motorola PowerPC G4 7410, ppc32 architecture	29	147.03
Intel Pentium 4 f12, x86 architecture	21	723.81
Sun UltraSPARC III, sparcv9 architecture	35	205.71
Intel Core 2 Quad Q6600 6fb, amd64 architecture	18	1066.67
AMD Athlon 64 X2 3800+ 15/75/2, amd64 architecture	21	761.9
This Design with 128bit bus size	0.63	3315.2
This Design with 32bit bus size	2.5	828.8

Table 7.4: Size and Delay comparison of different Core Implementations studied with this design

Chapter 8

Verification Simulation and Testing

All the modules are verified with there software counterparts through there functional simulation , Post-Route simulation and chip Scope hardware in loop simulation. As the maximum available clock is 100Mhz on ML403 Vertex 4 evaluation board hardware in loop simulations are based on 100Mhz clock. (see Figure 8.1 8.2)

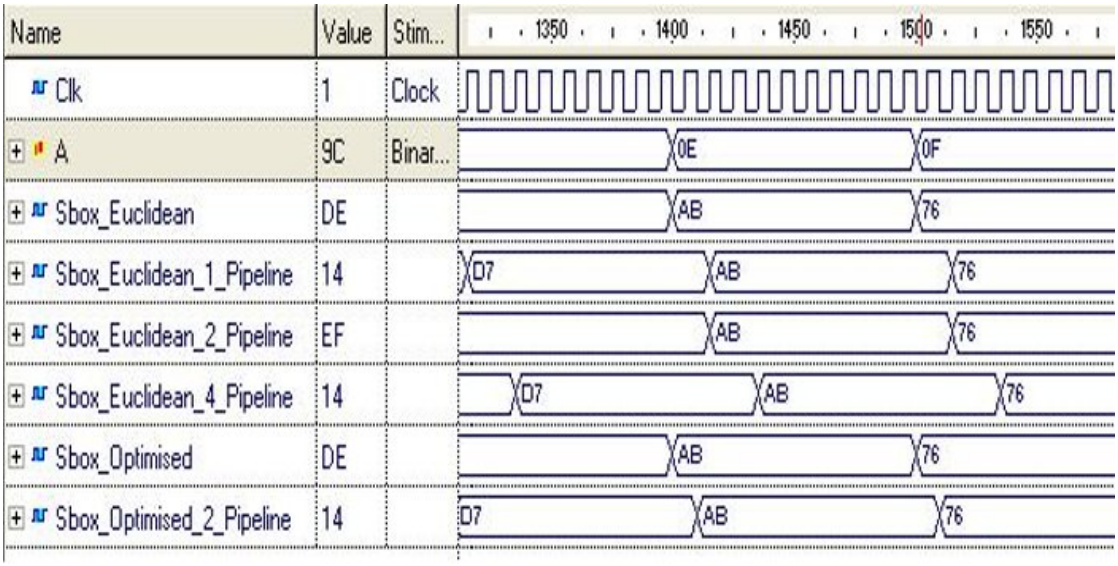


Figure 8.1: Functional Simulation of different Architecture of Sbox

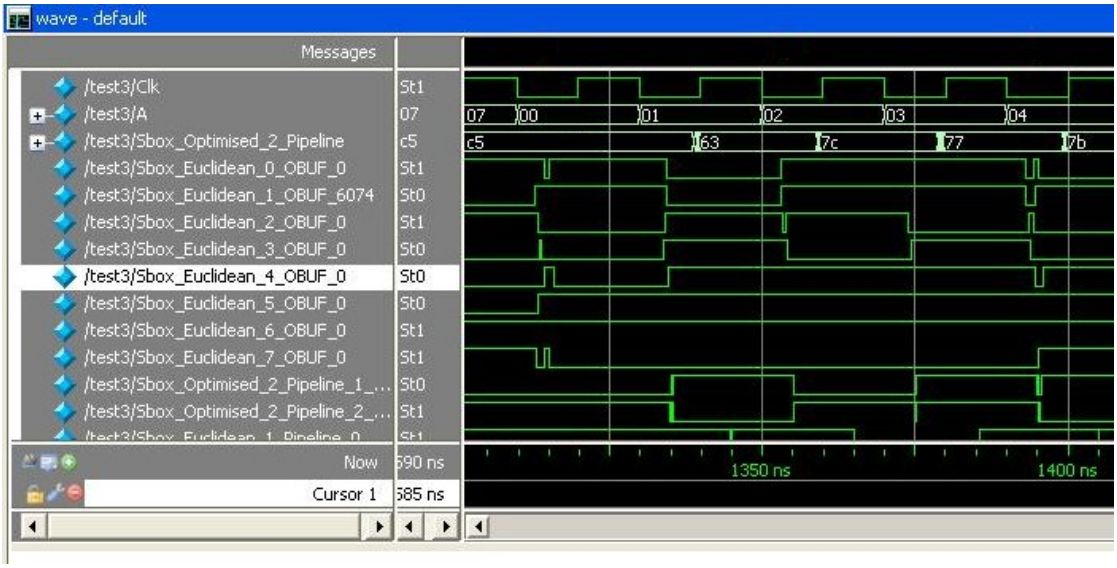


Figure 8.2: Post-Route Simulation of different Architecture of Sbox

Chapter 9

Conclusion

During the development of this project number of different FPGA based architectures are studied. This study is done on the basis of Design size and throughput. Among them, architecture based on composite field arithmetic is selected due to very compact design with moderate delays.

Hard-wire ShiftRow (and Inverse ShiftRow) operation in the sbox and byte level resource sharing between mix column and inverse mix column lead to both good speed and area saving.

All the transformations are optimized in the time domain by introducing pipelining in the transforms having larger gate depth.

Further Multiple parallel transform blocks are used in 32 bit and 128 bit designs in order to achieve parallelism. Alternatively, processing speed could be made higher by employing gate array or standard cell technology.

However, one should note that the pipeline structure is suitable for ECB (Electronic Code Book) mode of operation, but not very useful for other three modes (BCB, CFB, and OFB mode) where feedbacks are employed.

But in this design these feedback mode problem is taken care by dividing the pipelined channel into different independent parallel channels. Now these channels can work in (BCB, CFB, and OFB mode) feedback mods independently with lower throughputs. But total combined throughput of the design in these feedback modes will be equivalent to the throughput of design working as single channel in ECB mode.

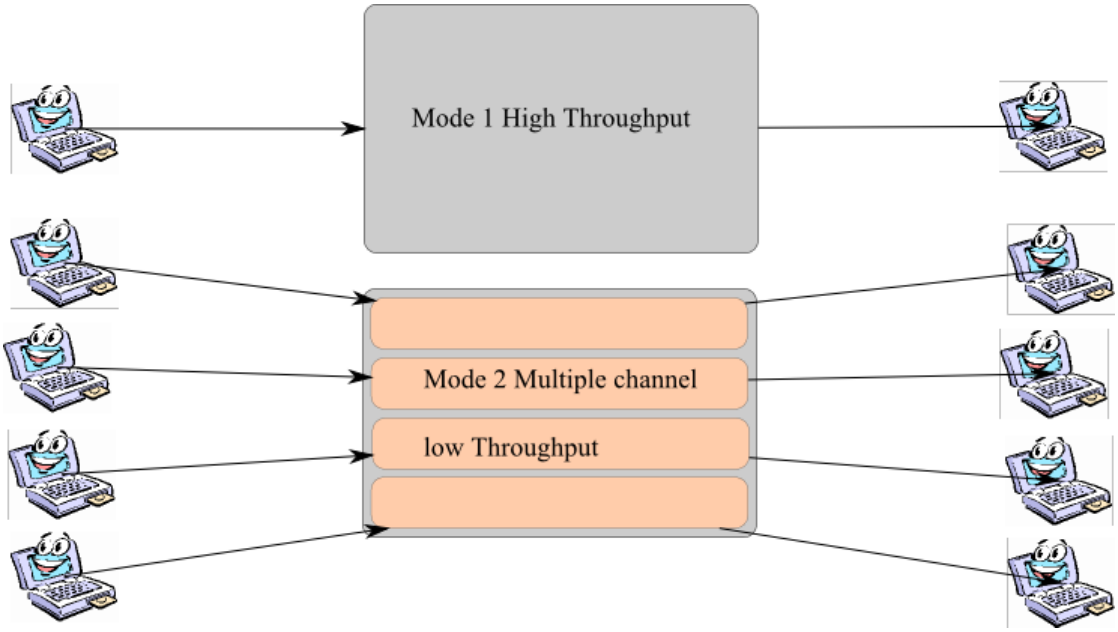


Figure 9.1: Different Modes

This technique is quite suitable for servers who have to make different multiple encrypted sessions with multiple clients. There high network throughput is required due to multiple clients. But in a single session high throughput is not required.

9.1 Specification

1. 128 bit bus size
2. Size 1300 CLB
3. Max Delay 3.86 nano seconds
4. Max Frequency 259 Mega Hz
5. Throughput 3312.2 Mbps
6. FPGA Design

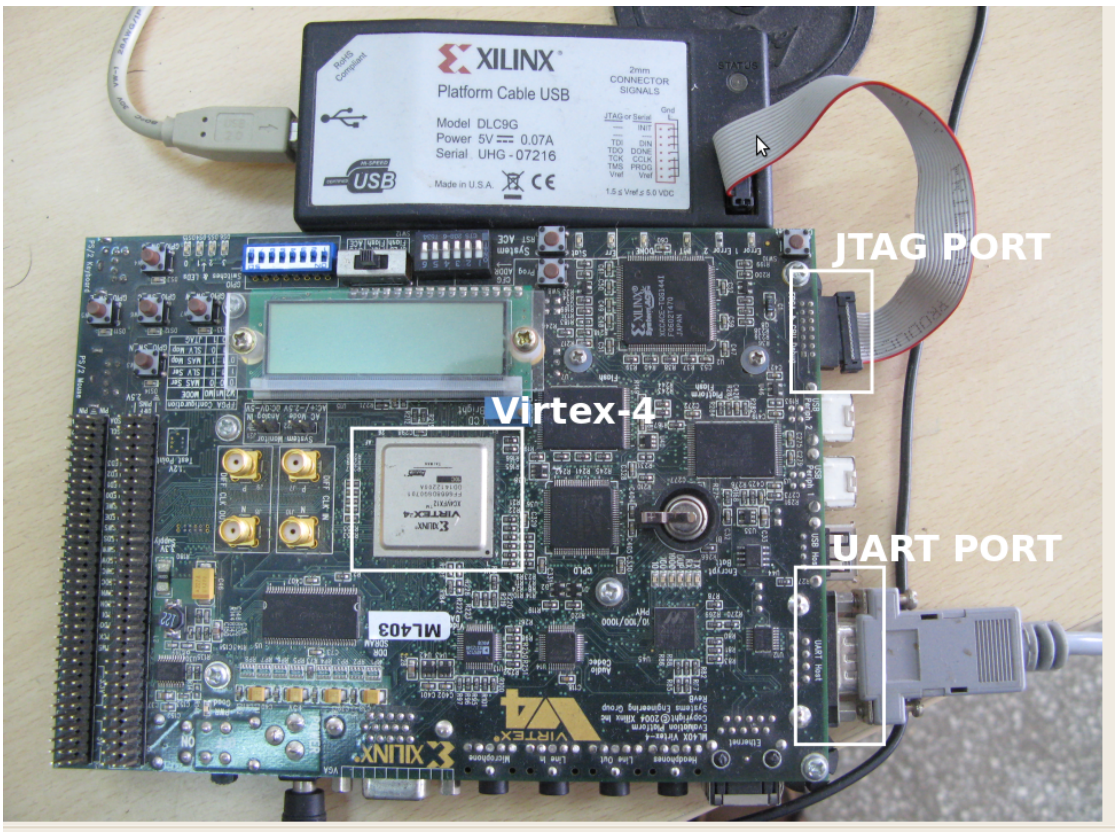


Figure 9.2: Setup

9.2 Future Work

Development of an encryption device based of a standard interface USB/SPI for mobile applications.

Development of an PCI based encryptor card that can be used in Servers and routers.

Bibliography

[1] Nabil Abu-Khader and Pepe Siy. Systolic galois field exponentiation in a multiple-valued logic technique. *Integr. VLSI J.*, 39(3):229–251, 2006.

[2] WEEKS B., BEAN M., ROZYLOWICZ T., and FICKE. Hardware performance simulations of round 2 advanced encryption standard algorithms. the third advanced encryption standard (aes3) candidate conference, new york, usa, 2000.

[3] Daniel J. Bernstein and Peter Schwabe. New aes software speed records.

[4] J. ELBIRT, YIP W., CHETWYND B., and PAAR C. A fpga implementation and performance evaluation of the aes block cipher candidate algorithm finalists. the third advanced encryption standard (aes3) candidate conference, new york, usa, 2000.

[5] V. Fischer, M. Drutarovsky, P. Chodowiec, and F. Gramain. Invmixcolumn decomposition and multilevel resource sharing in aes implementations. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(8):989 – 992, aug. 2005.

[6] LUTZ K. 2 gbit/s hardware realizations of rijndael and serpent, 2002.

[7] ICHIKAWA T.and KASUYA and MATSUI M. Hardware evaluation of the aes finalists. the third advanced encryption standard (aes3) candidate conference, new york, usa, 2000.

[8] MCLOONEs M. and MCCANNY J. High performance fpga rijndael algorithm implementations, 2000.

[9] Sumio Morioka and Akashi Satoh. A 10 gbps full-aes crypto design with a twisted-bdd s-box architecture. *Computer Design, International Conference on*, 0:98, 2002.

[10] Federal Information Processing and Announcing The. Announcing the advanced encryption standard (aes).

[11] F. Rodriguez-Henriquez, N. A. Saqib, and A. Diaz-Perez. 4.2 gbit/s single-chip fpga implementation of aes algorithm. *Electronics Letters*, 39(15):1115–1116, 2003.

[12] Atri Rudra, Pradeep Dubey, Charanjit Jutla, Vijay Kumar, Josyula Rao, and Pankaj Rohatgi. Efficient rijndael encryption implementation with composite field arithmetic. In etin Ko, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer Berlin / Heidelberg, 2001.

[13] Berk Sunar, Erkay Savas, and Cetin K. Ko? Constructing composite field representations for efficient conversion. *IEEE Transactions on Computers*, 52:1391–1398, 2003.

Glossary

$\oplus$ Exclusive-OR operation.. 10

$\times$ Finite field multiplication.. 13

AddRoundKey Transformation in the Cipher and Inverse Cipher in which a Round Key is added to the State using an XOR operation. The length of a Round Key equals the size of the State (i.e., for Nb = 4, the Round Key length equals 128 bits/16 bytes).. 5, 10–12, 17

AES Advance Encryption Standards. 3

array An enumerated collection of identical entities (e.g., an array of bytes).. 3, 5, 11, 12, 19

BDD Binary Decision Diagram. 72

Bit A binary digit having a value of 0 or 1.. 7, 12

Byte A group of eight bits that is treated either as a single entity or as an array of 8 individual bits.. 5

Cipher Series of transformations that converts plaintext to ciphertext using the Cipher Key.. 3, 5, 10, 11

Cipher Key Secret, cryptographic key that is used by the Key Expansion routine to generate a set of Round Keys; can be pictured as a rectangular array of bytes, having four rows and Nk columns.. 11, 12

Inverse Cipher Series of transformations that converts ciphertext to plaintext using the Cipher Key.. 12

InvMixColumn Transformation in the Inverse Cipher that is the inverse of MixColumns().. 12, 15, 17, 19

State Intermediate Cipher result that can be pictured as a rectangular array of bytes, having four rows and Nb columns.. 3, 5, 12, 17, 19, 61, 62, 64

SubByte Transformation in the Cipher that processes the State using a non-linear byte substitution table (S-box) that operates on each of the State bytes independently.. 5

SubWord Function used in the Key Expansion routine that takes a four-byte input word and applies an S-box to each of the four bytes to produce an output word.. 11, 12

TBDD Twisted Binary Decision Diagram. 72

Word A group of 32 bits that is treated either as a single entity or as an array of 4 bytes.. 3, 6, 10

XOR Exclusive-OR operation.. 11, 12