

# EDA385 Embedded Systems Design

## Advanced Course



**LUNDS**  
**UNIVERSITET**

Secure Embedded Systems

**Supervised by**  
Flavius Gruian

**Submitted by**  
Ahmed Mohammed Ibrahim (aso10ayo)  
Mohammed Shaaban Ibraheem Ali (aso10mib)

November 18<sup>th</sup>, 2011

## Abstract

In this project we are trying to solve the problem of the insecure data handling/communication by developing a simple security system that involves hardware and software components.

At the core of the system, there will be a hardware-based security engine. This engine is responsible for encrypting/decrypting the data using the Advanced Encryption Standard (AES) symmetric cipher, Keys generation, message handling (Padding, ...).

## Table of Contents

<b>1. INTRODUCTION.....</b>	<b>5</b>
<b>2. SYSTEM DESCRIPTION.....</b>	<b>6</b>
<b>3. THEORETICAL BACKGROUND .....</b>	<b>7</b>
<b>4. THE SOFTWARE .....</b>	<b>12</b>
4.1 Flowchart .....	12
4.1.1 Encryption.....	13
4.1.2 Decryption.....	13
4.2 Keys generation .....	13
4.3 Message padding.....	13
<b>5. THE HARDWARE.....</b>	<b>14</b>
5.1 Controller .....	14
5.2 AES Cipher .....	14
5.2.1 Round implementation .....	14
5.2.2 The subBytes implementation .....	16
5.2.3 The shift rows implementation .....	17
5.2.4 Mix Columns implementation .....	17
5.2.5 Add Round Key implementation.....	18
5.3 Decryption.....	18
5.4 Key Expansion Implementation.....	19
5.4.1 Key expansion control unit .....	20
5.4.2 The G module .....	21
5.4.3 Word XOR module .....	21

6. RESULTS .....	21
7. CONCLUSION .....	23
8. REFERENCES.....	23

# 1. Introduction

---

Nowadays many electronic devices (mobile phones, PDAs, Security Cameras... ) need to communicate with other devices via unsecured communication links, using such unsecured communication link is dangerous in case of communicating sensitive data, which may cause economic and/or security damages by illegally knowing it.

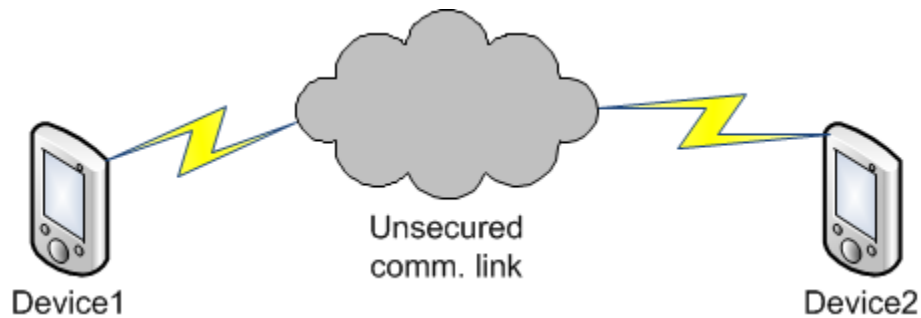


Figure 1: Two communicating devices via unsecure network

Even if the network itself is claimed to be secured, this cannot be taken for granted for communicating sensitive data, so the requirement of securing the data locally on the device level arises.

## 2. System Description

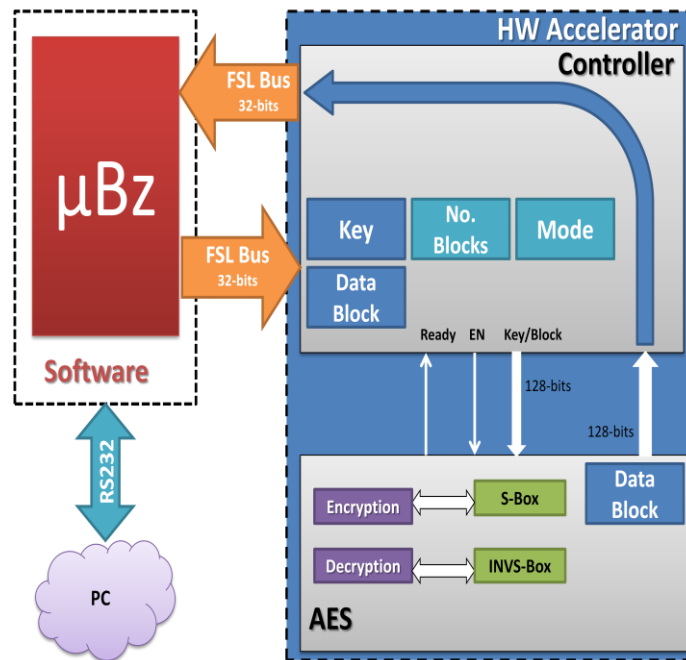


Figure 2: System block diagram

Figure 2 Shows the system's block diagram, two main components exist, the software part executing on the Microblaze softprocessor, and the hardware accelerated cipher, the central control of the system is handled by the software part.

The software is supposed to keep track of the operating mode (encryption or decryption of data), handle the data in terms of communicating it with the hardware accelerator via FSL links and the PC via a serial link, padding the data to fit into integer number of AES blocks, and finally the software is responsible for generating a completely random session key each session where we define a session here as a one message transfer.

The hardware accelerator consists of two parts, the hardware controller responsible for preparing the session key and the data as AES states (Blocks) received from the FSL links, keeping track of the operating mode to enable the correct cipher core (Encryption or Decryption), keeping track of the number of blocks per session to deliver the correct session key to the cipher core.

The AES core is only responsible for Encrypting/Decrypting one block at a time with the provided session key from the hardware controller.

Developing the encryption algorithm (AES) was the main challenge in this project, where we had to choose between different implementation methods, and implement the cipher's transformations on hardware.

## 3. Theoretical Background

---

As discussed in section 2, the project is centered around a famous cryptographic algorithm, the Advanced Encryption Standard (AES), we here very briefly introduce it, while section 5 of this report discusses how it was implemented.

### The Advanced Encryption Standard (AES)

Fig.3 shows the overall structure of AES. The input to the encryption and decryption algorithms is a single 128-bit data block which is mapped to a State array of 16 Bytes shown in figure 3, and a single 128-bit key, also mapped into a State array.

All the transformations in the AES algorithm occur on State typed data.

The input block (Plaintext in case of encryption), initially is subjected to an Add round key transformation, then 9 identical rounds operations are performed on the result, where each round contains 4 different transformations (Substitute Bytes, Shift Rows, Mix columns, Add round key), then an incomplete round – missing the Mix columns transformation - is further applied to generate the Ciphertext.

The round keys are generated from the original input key, where a key expansion algorithm is further applied to it, generating a 176 Byte expanded key –for a total of 11 Addroundkey transformation in the AES each operating on a 16 Byte Round key-

For the decryption, inverse transformations exists for each of the four transformations, and the order of execution is reversed too as shown in figure 3, however the key expansion is still the same algorithm.

Now we briefly describe the four transformations found in the AES.

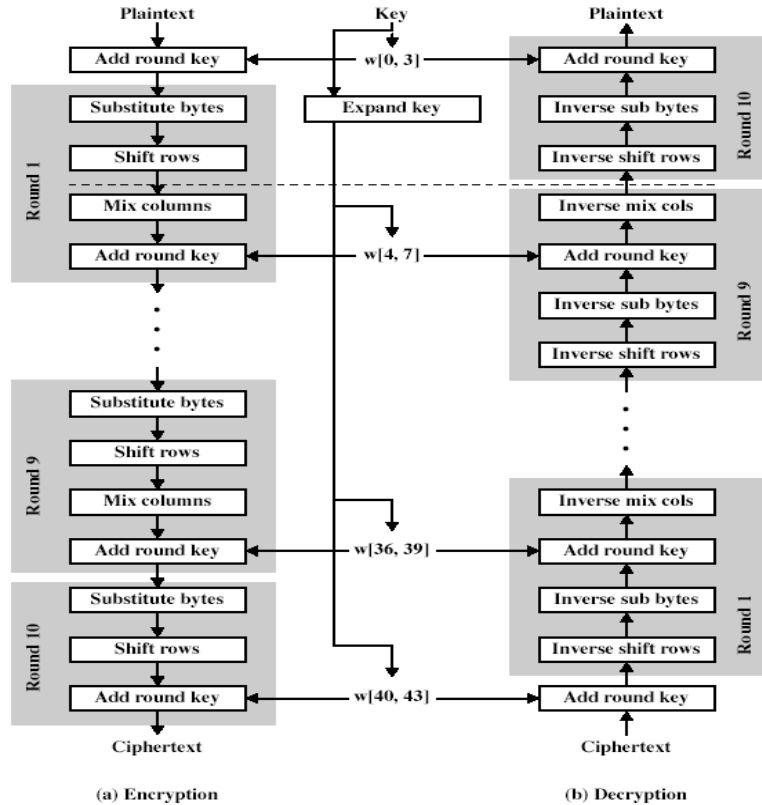


Figure 3 Overall AES Structure [1]

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33

Figure 4: AES State [1]

### SubBytes Transformation:

The substitute byte transformation is also known as SubBytes. SubBytes is a non-linear byte substitution of each byte in a State. This transformation is equivalent to a substitution using a look-up table

The look-up table is called an S-Box. Figure 5 illustrates the S-Box.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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 5: S-Box [1]

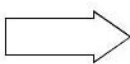
S-box, contains a permutation of all possible 256 8-bit values. Each byte of State is mapped into a new byte.

The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 8-bit output value. For example the value {95} is mapped into the value {2A}.

### ShiftRows Transformation

The Shiftrows transformation, is shown in Fig. 6 The first row of State is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed. The following is an example of ShiftRows:

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33



00	01	02	03
11	12	13	10
22	23	20	21
33	30	31	32

Figure 6: ShiftRows Transformation [1]

### MixColumns Transformation

The Mixcolumns transformation, operates on each column of a State individually. Each column is multiplied by a particular matrix to produce a new column which will be a column in the new State.

This can be illustrated by the following equation:

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

Figure 7: Mixcolumns equation [1]

Multiplication by 01 does not need any processing. Multiplication by 02 can be implemented by a simple shift left one bit. If the result is greater than 127, subtract 0x1B from the result (using bitwise XOR.) Multiplication by 03 is only the addition of the result of multiplication times 1 and multiplication times 2. This addition is also a bitwise XOR operation.

### AddRoundKey Transformation

The AddRound key transformation, simply XOR the bits of a State with those of the round key.

$$\begin{bmatrix} 04 & E0 & 48 & 28 \\ 66 & CB & F8 & 06 \\ 81 & 19 & D3 & 26 \\ E5 & 9A & 7A & 4C \end{bmatrix} \oplus \begin{bmatrix} A0 & 88 & 23 & 2A \\ FA & 54 & A3 & 6C \\ FE & 2C & 39 & 76 \\ 17 & B1 & 39 & 05 \end{bmatrix} = \begin{bmatrix} A4 & 68 & 6B & 02 \\ 9C & 9F & 5B & 6A \\ 7F & 35 & EA & 50 \\ F2 & 2B & 43 & 49 \end{bmatrix}$$

The first matrix is the State, and the second matrix is the round key.

The reverse transformations are not identical, however they do the exact reverse calculations on the Ciphertext to produce back the original Plaintext.

### AES Key Expansion

The AES key expansion algorithm takes as input a 4-word (16-byte) key and produces a linear array of 44 words (176 bytes). This is sufficient to provide a 4-word round key for the initial AddRoundKey stage and each of the 10 rounds of the cipher. The following pseudocode describes the expansion:

```
KeyExpansion (bytekey[16], word w[44])
{
  word temp;
  for i = 0 to 3 do
    w[i] = (key[4i], key[4i+1], key[4i+2], key[4i+3]);
  end loop;

  for i = 4 to 43 do

    temp = w[i-1];
    if i mod 4 = 0 then
```

```

temp = SubWord (RotWord (temp)) xorRcon[i/4];
w[i] = w[i-4] xor temp;
end loop;

```

The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word  $w[i]$  depends on the immediately preceding word,  $w[i-1]$ , and the word four positions back,  $w[i-4]$ . In three out of four cases, a simple XOR is used. For a word whose position in the  $w$  array is a multiple of 4, a more complex function is used. RotWord performs a one-byte circular left shift on a word. SubWord performs a byte substitution on each byte of its input word, using the S-box.

The result of steps 1 and 2 is XORed with a round constant,  $Rcon[j]$ .

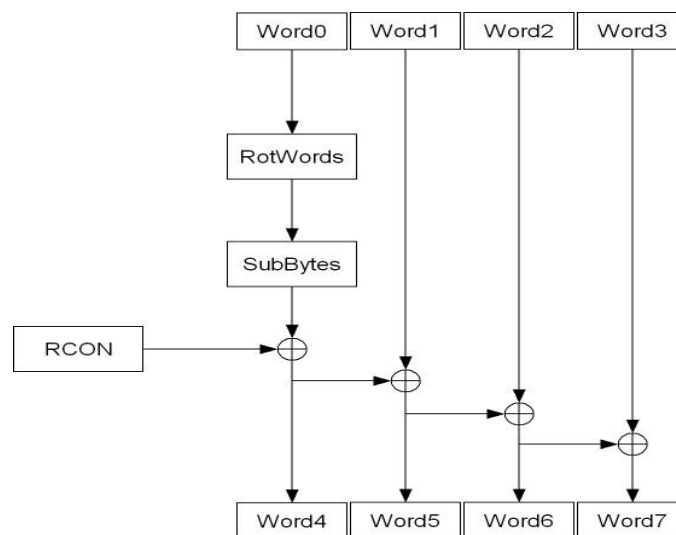


Figure 8: AES Key expansion for one round key [1]

Table 1 shows the values of the round constant (RCON) for each round.

Round Number	RCON
1	01000000
2	02000000
3	04000000
4	08000000
5	10000000
6	20000000
7	40000000
8	80000000
9	1B000000
10	36000000

Table 1: RCON values for the different rounds

## 4. The Software

As discussed in section 2, the software part is the main system controller, and it operates in two modes (encryption-decryption), determined initially by the user.

In this section we describe the software's functionality using its flowchart, then we discuss two of the main software's functions. The session keys generation, and the message padding.

### 4.1 Flowchart

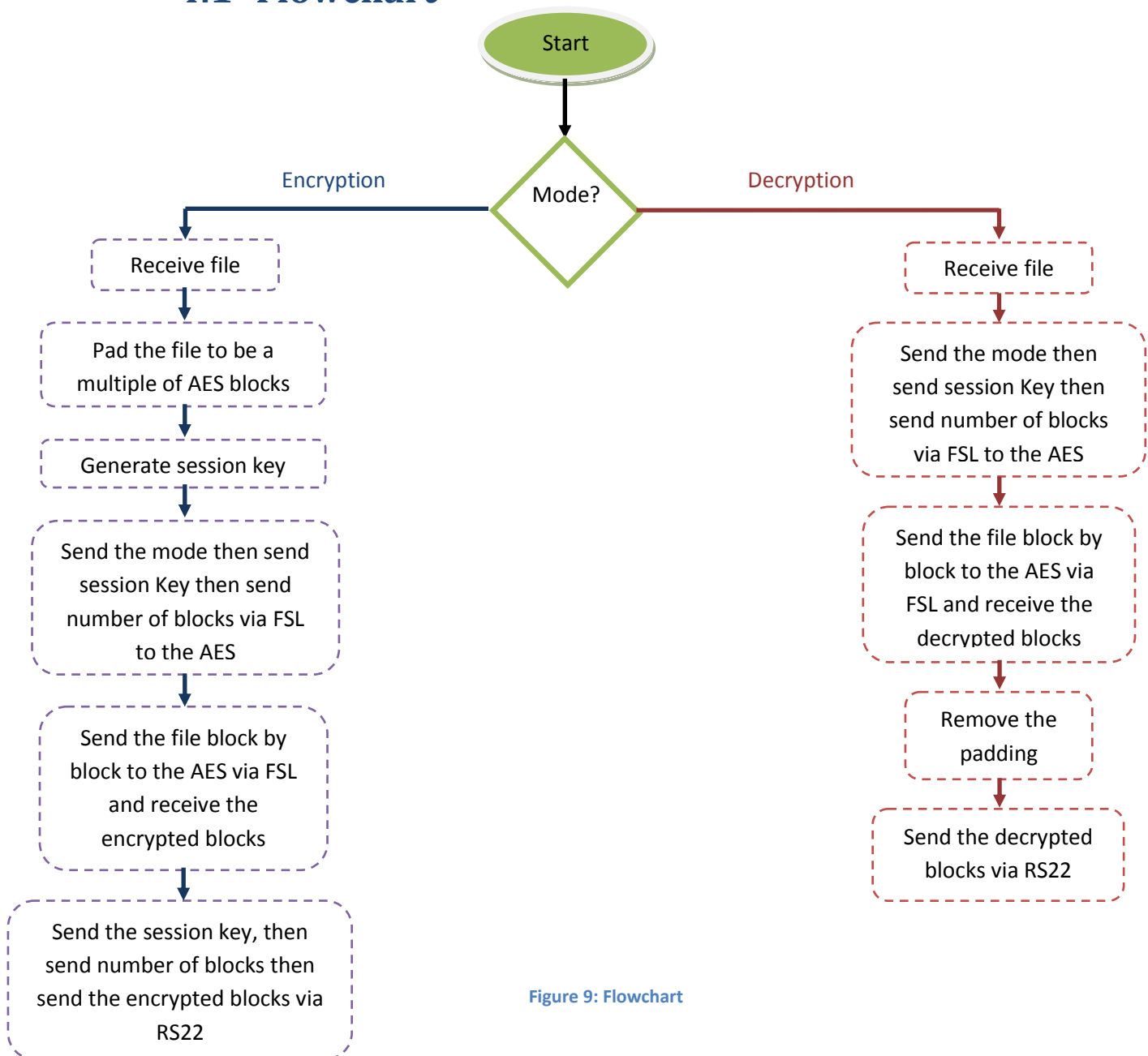


Figure 9: Flowchart

Figure 9 shows the software flowchart, the software first determines the operation mode (encryption/decryption), and then it starts executing consecutive functions as follows:

### 4.1.1 Encryption

First the software starts to receive the file to be encrypted from the RS232 port, then it calculates its size in bytes and perform padding to it as discussed in section 4.3.

A timer is initialized at the start of the program and stopped after the file is received, this timer is used to generate the session key.

The mode byte is sent to the hardware controller, then the number of blocks and the session key are sent, then we start sending the file block by block, and receive the corresponding encrypted blocks.

After we have received all the encrypted blocks, we start sending the whole encrypted file through the RS232 link.

### 4.1.2 Decryption

Unlike the encryption the received file contains the session key that will be used to decrypt the file, and the encrypted data, so there is no keys generation phase in decryption, session keys should be transferred securely between the communicating parties, however in this project we assume that the keys management is done outside of the system.

The mode and session key and the encrypted blocks are sent to the hardware to perform decryption, then we receive the encrypted blocks + the last padded block, then we start removing the padding by going through the last byte in the last block and remove the 0x00 bytes till we find a 0x01 byte then we know that this is the last padding byte, then the decrypted file is sent back through the RS232 link.

## 4.2 Keys generation

As discussed in 4.1.1 a completely random 128-bit value to serve as a session key was needed, so we used the built in Microblaze timers for that purpose by generating 4 different 32-bits random values.

## 4.3 Message padding

In order to make the plaintext a multiple number of AES blocks size -16 Bytes- we padded the input data by an initial “0x01” byte then with 0x00’s till the padding size.

The size of padding could be determined as follows:  $\text{PaddingSize} = 16 - (\text{txtLen} \% 16)$

Where txtLen is the length of the received data in Bytes, and the number of blocks (N) can be further determined after padding by:  $N = (\text{txtLen} + \text{PaddingSize}) / 16$

# 5. The Hardware

---

Figure 2 shows that the hardware part is composed of two main components, the Controller and the AES cipher accelerator, we here describe both.

## 5.1 Controller

The hardware controller is responsible for communicating the data over the FSL link, it first receives the mode of operation then enables the correct AES cipher (Encryption or decryption), then receives the number of blocks, that is necessary to ensure the validity of the session key with the received block.

Then the data to be encrypted/decrypted is received and sent to the corresponding AES cipher, and after processing it sends back the processed block to the software part.

FSL links are 32 bits, however our data blocks are 128-bits, so for every block of data 4 wait/acknowledge state pairs exist.

## 5.2 AES Cipher

### 5.2.1 Round implementation

As stated before the AES cipher is composed of 9 successive identical rounds and one incomplete round at the final stage. Each round from those 10 rounds typically have two inputs: Round plaintext & Round key And one output : Round cipher text.

If this is the first round so Round plaintext is the output from applying the AddRoundKey stage to both the original plaintext and session key.

Also the Round key input to this round is the first four words output from the key expansion module.

For the rest 9 rounds (including the last incomplete round ) the Round plaintext for Round (i) is the Round cipher text output from the previous round (Round (i-1)).

And the Round key input to this round is also the corresponding four words output from the key expansion module.

For the last incomplete round the round cipher text output from it would be the cipher text output of the AES module.

Recall that the structure for a complete round contains four successive functions or transformation:

- 1) Substitute bytes.
- 2) Shift rows.
- 3) Mix columns.
- 4) Add round key.

Where the last incomplete round doesn't have the mix columns transformation.

Two different methods for implementing the AES could be applied.

- 1) Space optimized implementation.
- 2) Speed optimized (pipelined) implementation.

### 5.2.1.1 Space optimized implementation

- For applications or systems concerned with the size of the hardware implemented more than the total time of execution, this implementation is suitable.
- As seen from the previous discussion there are 9 identical rounds and a last incomplete round (but still have some components as the previous rounds) in the structure of the AES module.

Here we implement only one round on the hardware, and operating it for 10 successive times with a certain Round plaintext input each time is taken from the Round ciphertext output of the previous round(except for the first round), and with a Round key input is taken from the corresponding four words from the key expansion module, and for the last round disabling the mix columns function and taking its Round ciphertext output as the ciphertext output of the AES module.

A typical round structure for the previous description is shown in figure10

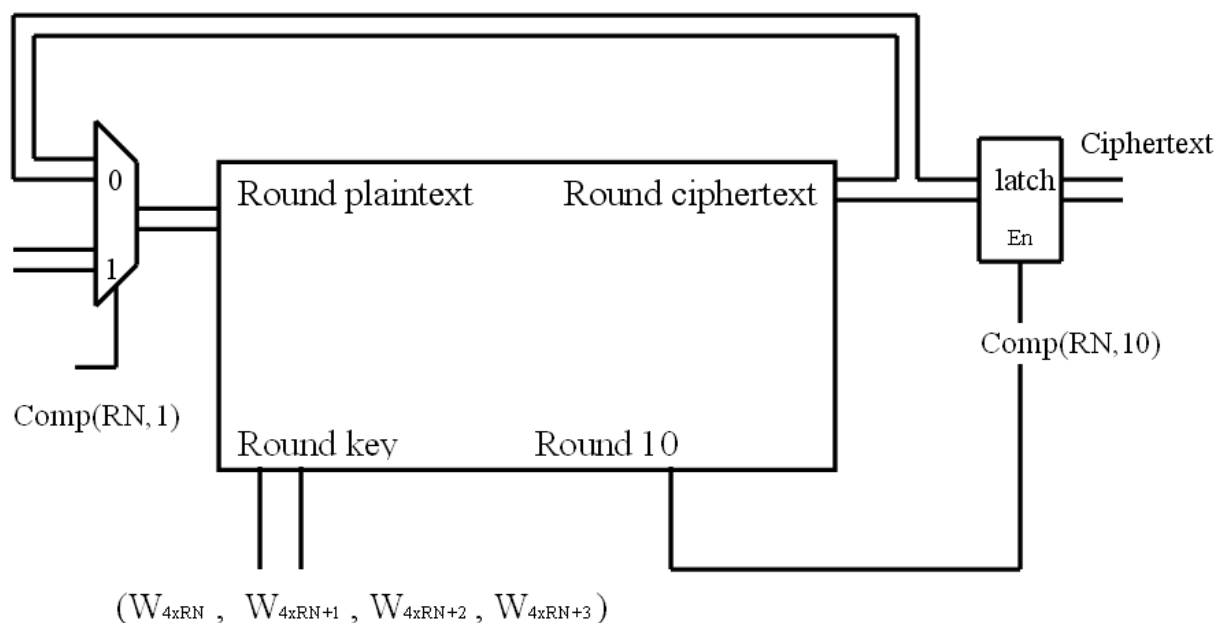


Figure 10: Typical Round Structure

Where:-

$R_N$  : Round number.

$\text{Comp}(R_N, x)$  : compare Round number with  $x$ .

$W_{4R_N+i}$  :  $Word_{4R_N+i}$  from the expanded key.

Round 10 : If enabled so in the internal architecture of

the round disable the Mix Columns stage and

connect the output from the Shift Rows stage

to the input of the Add Round Key stage

With the previous architecture we've combined the first 9 rounds along with the last incomplete round in the same hardware, with the round number to choose between a complete and an incomplete round.

### 5.2.1.2 Speed optimized (pipelined) implementation

Another way for the AES cipher implementation is to construct its 10 successive round modules in hardware rather than the previous architecture of implementing only one round module and operating it for 10 successive times (with the round10 line to select a complete or an incomplete round).

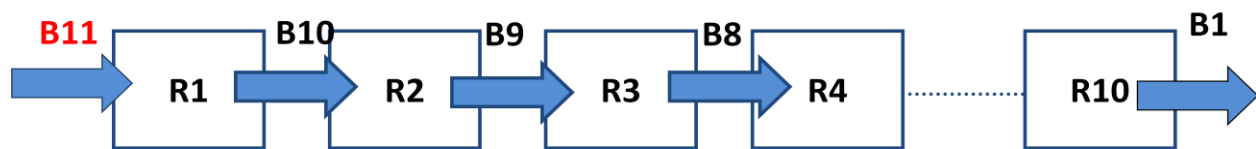


Figure 11: Pipelined Implementation

A Mix between the two approaches could be customized according to the Space/Speed limitations of the design.

In our project we adopted the Space optimized implementation of the AES.

### 5.2.2 The subBytes implementation

Recall from section 3 that the sub bytes transformation is a look up table (s-box) that have 256 different entries for different 256 inputs.

Each byte in the state to be processed has a corresponding substitution in the s-box.

The s-box was implemented as a block memory with width = 8 bits and of depth of 256 word so that its address range is from 0 to 255, which will be the byte to be substituted value.



The subbytes process will be accessing the block memory for 16 successive times, each for one byte substitution, which is considered a bottleneck in our design.

### 5.2.3 The shift rows implementation

The forward shift rows transformation is done by processing on the input states as follows :-

- The first row of state is not altered.
- The second row, a 1-byte circular left shift is performed.
- The third row, a 2-bytes circular left shift is performed.
- The fourth row, a 3-bytes circular left shift is performed.

### 5.2.4 Mix Columns implementation

The Mix Columns transformation is defined by a constant matrix multiplication with the input state in  $GF(2^8)$ .

The forward Mix Columns transformation defines the new bytes after applying the MixColumn function on the old states as follows:

$$S'_{0,J} = (2 \cdot S_{0,J}) \oplus (3 \cdot S_{1,J}) \oplus S_{2,J} \oplus S_{3,J}$$

$$S'_{1,J} = S_{0,J} \oplus (2 \cdot S_{1,J}) \oplus (3 \cdot S_{2,J}) \oplus S_{3,J}$$

$$S'_{2,J} = S_{0,J} \oplus S_{1,J} \oplus (2 \cdot S_{2,J}) \oplus (3 \cdot S_{3,J})$$

$$S'_{3,J} = (3 \cdot S_{0,J}) \oplus S_{1,J} \oplus S_{2,J} \oplus (2 \cdot S_{3,J})$$

Where

$S_{i,J}$  is the input byte

$S'_{i,J}$  is the output byte after transformation

J is the column number in the input \* output state

By inspecting the previous four equations we discovered that each input byte would be subjected to a multiplication by 2 & multiplication by 3 in  $GF(2^8)$ , so a straight forward method is to initially calculate all 16 values for the 16 bytes after multiplying them by 2 and 3 and storing them, then calculate the new byte value after transformation by applying equation x on it.

For example:

The array Mul\_2 holds the 16 values of the multiply by 2 results of the input bytes, and Mul\_3 array also holds the 16 values of the multiply by 3 results of the input bytes.

So when calculating  $S'(6)$  it will be as follows

$$S'(6) = S(4) \text{ xor } S(5) \text{ xor } \text{Mul\_2}(6) \text{ xor } \text{Mul\_3}(7)$$

Where  $S'(6) \equiv S'_{2,1}$

For the (multiplication by 2):-

First shift left the input byte by 1-bit and if MSB of the input's byte is initially equal '1' then XOR the shifted output with (00011011).

For the (multiplication by 3):-

Take the output of the previous process and XOR it with the input byte.

So, filling the  $\text{Mul\_2} * \text{Mul\_3}$  arrays would be a straight forward procedure.

### 5.2.5 Add Round Key implementation

Perhaps the easiest transformation between the four main transformations constructing an AES round, is the Add Round Key transformation, as it just XORs the input state with the corresponding four words of the expanded key to the current round.

## 5.3 Decryption

The decryption structure is identical to the reverse of the encryption one, with using the inverse transformation –Figure 2 (b)-, here we will only list the differences between the forward and reverse transformations.

In the subBytes transformation the same process as the forward transformation exists, but with an inverse S-Box used for substitution.

The shift rows is simply the inverse shifting of the forward one.

The Mix columns is quite difficult, the corresponding equations are:-

$$S'_{0,J} = (OE \cdot S_{0,J}) \oplus (OB \cdot S_{1,J}) \oplus (OD \cdot S_{2,J}) \oplus (O9 \cdot S_{3,J})$$

$$S'_{1,J} = (O9 \cdot S_{0,J}) \oplus (OE \cdot S_{1,J}) \oplus (OB \cdot S_{2,J}) \oplus (OD \cdot S_{3,J})$$

$$S'_{2,J} = (OD \cdot S_{0,J}) \oplus (O9 \cdot S_{1,J}) \oplus (OE \cdot S_{2,J}) \oplus (OB \cdot S_{3,J})$$

$$S'_{3,J} = (OB \cdot S_{0,J}) \oplus (OD \cdot S_{1,J}) \oplus (O9 \cdot S_{2,J}) \oplus (OE \cdot S_{3,J})$$

Each input byte would be subjected to a multiplication by O9, OB, OD, OE, so we can generate four arrays for  $\text{Mul\_9}$ ,  $\text{Mul\_B}$ ,  $\text{Mul\_D}$  and  $\text{Mul\_E}$  operations then fill them and substitute directly to get the new values for the Mix Columns transformation.

The procedure for finding the values  $Mul\_9(i)$ ,  $Mul\_B(i)$ ,  $Mul\_D(i)$  and  $Mul\_E(i)$  Where 'i' is the input byte number is :-

Find for each byte the  $Mul\_2$ ,  $Mul\_4$ ,  $Mul\_8$  values by successive 1-bit shifting and conditional XOR with (00011011), then substitute in the following equations:

$$Mul\_9(i) = S(i) \text{ XOR } Mul\_8(i)$$

$$Mul\_B(i) = S(i) \text{ XOR } Mul\_2(i) \text{ XOR } Mul\_8(i)$$

$$Mul\_D(i) = S(i) \text{ XOR } Mul\_4(i) \text{ XOR } Mul\_8(i)$$

$$Mul\_E(i) = Mul\_2(i) \text{ XOR } Mul\_4(i) \text{ XOR } Mul\_8(i).$$

## 5.4 Key Expansion Implementation

AES key expansion algorithm is identical in both encryption and decryption phases.

The only difference was that the round key corresponding to round 1 in the encryption is the round key corresponding to the last round in the decryption, so in case of decryption we need to generate the whole expanded key at the beginning before processing, as round 1 key would be the last 4 words in the expanded key.

However, implementing one iterative key expansion unit in case of encryption, then another one complete in case of decryption would be confusing task, so we decided to implement only the complete one, and use it in case of both encryption and decryption.

Recall the algorithm for key expansion, as for each round key (4 words) to be generated 3 out of them are generated by a simple XOR between the word preceding it ( $w-1$ ) with the word four positions back, this is a trivial implementation of XORing four bytes –Figure 8-.

The challenging part was the (g) function performed when calculating  $w(i)$ , where (i) is a multiple of 4, the XORed words would be in this case the word four positions back  $w(i-4)$  and the word results from applying the (g) function on the previous word i.e.( $g[w(i-1)]$ ), so we need to apply the (g) function on each word of index  $(4n-1)$  where (n) is an integer value.

Now we have three main modules:

- 1- G.
- 2- Word XOR.
- 3- Key expansion control unit.

As the name implies the G module is the implementation of the G function that process on the last word in each generated 4 words to get  $G[[w(i-1)]]$ .

In the beginning the original key is stored in the first four words in the expanded key buffer then we enable the G module that takes the last word and perform on it the G function, then pass it to the word

XOR module, which takes two inputs last round key and the output word from the (G) module, then generates the current round key according to the following equations:

$$W_{4R_N} = W_{4R_N-4} \text{XORG}(W_{4R_N-1})$$

$$W_{4R_N+1} = W_{4R_N-3} \text{XOR } W_{4R_N}$$

$$W_{4R_N+2} = W_{4R_N-2} \text{XOR } W_{4R_N+1}$$

$$W_{4R_N+3} = W_{4R_N-1} \text{XOR } W_{4R_N+2}$$

Where  $R_N$  = Round Number.

By iteratively continue on this procedure, We would generate the whole expanded key, Then storing it for the round function requests.

Now moving forward to the implementation of the three main modules in the key expansion unit.

### 5.4.1 Key expansion control unit

In the beginning when the AES module is enabled to start processing, The input key is expanded before processing on the plaintext.

So the main control unit of the AES module (the top module), asserts a generate expanded key signal to the key expansion control unit, when this signal is asserted, the key expansion control unit stores the input key in the first four words of the expanded key space then starts enabling the (G) module that processes on the last word only, then it enables the word XOR module that processes on both the G module's output and the previous round key.

This cycle continues for 10 successive times till the whole 44 words of the expanded key are filled. When finished, they key expansion control unit asserts an acknowledge signal to the main control unit to start processing one the input plaintext, the main control unit then starts fetching the corresponding round key from the key expansion control unit at the start of each round processing phase.

The main functions of the key expansion module are as follows:

- 1) Stores the input key in the first 4 words of the expanded key space.
- 2) Supplying the (G) module with the last word in the previous round's key and supplying the word XOR module with the whole previous round key.
- 3) Supplying the (G) module with the effective byte of the current Rcon value.
- 4) Enabling the G module to start processing and disabling it when finished generating the round key to store it then enables it again with a new input.
- 5) Supplying the main control unit with the requested round key ( By the round number input).

### 5.4.2 The G module

The G function has three main sub-functions.

- 1) Rot word.
- 2) Sub word.
- 3) XOR with Rcon(J).

The Rot word is done initially by changing the input bytes position before substituting.

The Sub word is exactly as the Sub bytes transformation but here substitution is done on just four bytes.

Then XORing the last byte (generated by substitution) with the Rcon(J) input that is supplied by the key expansion control unit which keeps with the current round number in the key generation process.

### 5.4.3 Word XOR module

The word XOR module will have the round key & the output word from the (G) module, to process on, it's function is to just XOR words, then outputs the resultant four words to be stored in it's corresponding location in the expanded key buffer.

## 6. Results

Processing one block took 653 Clock Cycles, 360 of them are only for memory accesses in the different substitution stages, Which is a significant overhead we can avoid by implementing the S-Box transformation.

/clk	1																									
/reset	0																									
/input_key	{00}	{00}	{00}	{00}	{00}	{00}	{00}	{00}	{00}	{00}	{00}															
/plaintext	{AA}	{AA}	{AA}	{AA}	{AA}	{AA}	{AA}	{AA}	{AA}	{AA}	{AA}															
/cipher_enable	1																									
/cipher_text	{65}	{65}	{65}	{65}	{65}	{65}	{65}	{65}	{65}	{65}	{65}															
/round_key	{8E}	{16}	{63}	{63}	{AA}	{FB}	{99}	{AC}	{2B}	{EE}	{90}	{92}	{A7}	{9B}	{9B}	{F0}	{9F}	{FA}	{41}	{49}	{8E}	{18}	{8F}	{6F}	{CF}	{51}
/round_number_in	10	0	1	2	3	4	5	6	7	8	9	10														

Figure 12: AES encryption results

We were able to encrypt and decrypt a small text file, figure 13 shows a text file to be encrypted and the corresponding text after encryption.

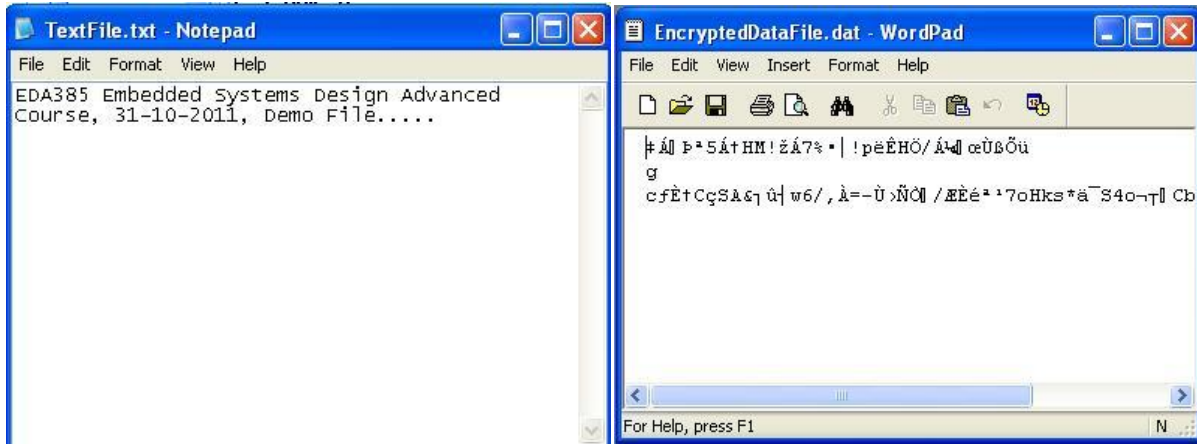


Figure 13: The original file and the result from AES encryption

Figure 14 shows the device utilization summary for both encryption and decryption and FS- to-AES controller.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	4652	8672	53%
Number of Slice Flip Flops	6374	17344	36%
Number of 4 input LUTs	5749	17344	33%
Number of bonded IOBs	70	250	28%
Number of BRAMs	6	28	21%
Number of GCLKs	2	24	8%

Figure 14: Hardware system device utilization

Figure 15 shows the device utilization for the whole system.

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Total Number Slice Registers	8,537	17,344	49%	
Number used as Flip Flops	8,505			
Number used as Latches	32			
Number of 4 input LUTs	8,492	17,344	48%	
Number of occupied Slices	7,560	8,672	87%	
Number of Slices containing only related logic	7,560	7,560	100%	
Number of Slices containing unrelated logic	0	7,560	0%	
Total Number of 4 input LUTs	8,575	17,344	49%	

Figure 15: System device utilization

## 7. Conclusion

---

Adding security to embedded systems is vital for guaranteeing secure storing/communicating sensitive data.

AES is a cryptography standard widely used.

In our project we encrypt/decrypt messages from a PC, however this can be easily modified to encrypt/decrypt any data in a embedded device.

AES can be implemented with regard to either space or speed optimization, we adopted the space optimized implementation.

SubBytes stage is a bottleneck as it depends on many memory accesses.

## 8. References

---

[1] Cryptography and Network Security Principles and Practices, Fourth Edition, By William Stallings