ECE385
DIGITAL SYSTEMS LABORATORY
Advanced Encryption Standard (AES)

What is the Advanced Encryption Standard (AES):

- A cipher specification for electronic data encryption [1]
- Established by the U.S. National Institute of Standards and Technology in 2001 as Federal Information Processing Standards Publication 197 (FIPS PUB 197) to replace the older Data Encryption Standard (DES)
- A symmetric block cipher based on the Rijndael algorithm

A ***Cipher*** is an encryption process to transform a meaningful message into scrambled data for the purpose of data transfer security. A Cipher generally takes in a message, called a ***Plaintext***, and a secret key, called a ***Cipher Key*** as its inputs. Depending on the specification of the Cipher, the Plaintext and the Cipher Key will go through some kind of algorithm, and produces a scrambled data as the output, which is called a ***Ciphertext***. The ***Inverse Cipher***, the dual of the cipher, reverses the direction of the encryption process. It takes in a Ciphertext and a Cipher Key, goes through the reverse algorithm, and produces the original meaningful Plaintext message.

A block cipher is a deterministic cipher algorithm operating on a fixed number of bits, denoted a block, at a time. If the cipher uses the same set of cipher key for both the encoding and the decoding process, it is called a symmetric-key algorithm. The AES algorithm is a symmetric block cipher based on the Rijndael algorithm, which is originally designed to handle different block sizes and Cipher Key lengths. However the AES standard specifies the data block to be of 128 bits, using Cipher Keys with lengths of 128, 192, and 256 bits. In this tutorial we will introduce the 128-bit Cipher Key version of the AES algorithm in the following sections. This tutorial is adopted from FIPS PUB 197 [1], and it skips the mathematical theory behind the AES algorithm. For the detailed mathematical background, please refer to [1].

Interface and Data Structure:

- **Top-Level Inputs and Outputs**

Inputs

Plaintext	: logic [127:0] – 128-bit data input to the Cipher or output from the Inverse Cipher. Arranged in 16 Bytes and stored in a 4x4 Byte column-major matrix.
Cipher Key	: logic [127:0] – 128-bit secret, cryptographic key arranged in 16 Bytes and stored in a 4x4 Byte column-major matrix.

Outputs

Ciphertext	: logic [127:0] – 128-bit data output from the Cipher or input to the Inverse Cipher.

- **Terminalogy**

State	: logic [127:0] – 128-bit intermediate results during the AES algorithm. Also is arranged in 16 Bytes and stored in a 4x4 Byte column-major matrix, which can be denoted by four *Words*.
Word	: logic [31:0] – A group of 32 bits data within a State that are processed together as a single entity. Each Word is composed of the 4 Byte data from a single State column.
Round Key	: logic [127:0] – 128-bit keys derived from the Cipher Key using the Key Expansion routine. It is applied in different stages of the algorithm.
Key Schedule	: logic [1407:0] – $128\times(N_r+1)$-bit keys derived from the Cipher Key using the Key Expansion routine, where $N_r=10$ is the number of looping rounds for the 128-bit AES algorithm. Each of the individual 128-bit Keys is applied to different stages of the algorithm.

AES Encryption Algorithm:

- **AES Encryption Pseudo Code**

Figure 1 shows the pseudo code for the AES encryption algorithm, where $N_b=4$ is the number of columns in the State. Notice that there are only 9 full looping rounds for the 128-bit AES. The last round consists of the same modules as the looping rounds minus the MixColumns() module. Figure 2 shows the encryption algorithm flow.

```
AES(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
      byte state[4,Nb]
      state = in
      AddRoundKey(state, w[0, Nb-1])
      for round = 1 step 1 to Nr-1
          SubBytes(state)
          ShiftRows(state)
          MixColumns(state)
          AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      end for
      SubBytes(state)
      ShiftRows(state)
      AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
      out = state
end
```

Figure 1 Pseudo code for the AES encryption algorithm [1].

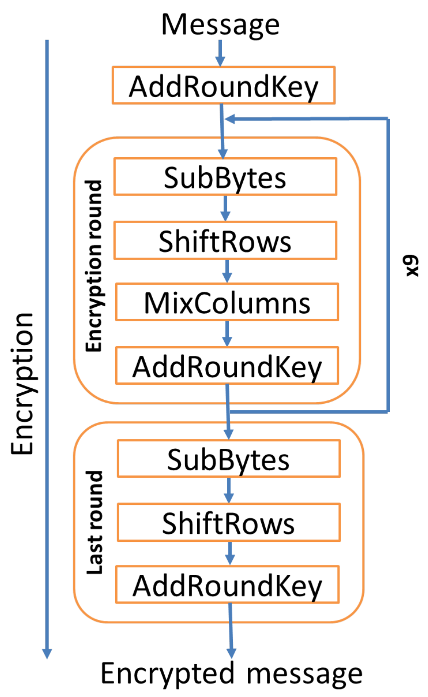

Figure 2 AES encryption algorithm flow.

- **AES modules overview**

  The Pseudo code for the AES algorithm in Figure 1 takes *in* (Plaintext) and *w* (expanded Cipher Key, using a separate module **KeyExpansion()**). It then goes through several rounds of modules and computes *out* (Ciphertext). Below describes the purpose of each module.

  KeyExpansion () – Takes the Cipher Key and performs a Key Expansion to generate a series of Round Keys (4-Word matrix) and store them into Key Schedule.

  AddRoundKey() – A Round Key of 4-Word matrix is applied to the updating State through a simple XOR operation in every round.

  SubBytes() – Each Byte of the updating State is non-linearly transformed by taking the multiplicative inverse in Rijndael's finite field (or the Galois field – $GF(2^8)$) then applying an affine transformation. The process is usually simplified into applying a lookup table called the Rijndael S-box (substitution box).

  ShiftRows() – Each row in the updating State is shifted by some offsets.

  MixColumns() – Each of the four Words in the updating State undergoes separate invertible linear transformations over $GF(2^8)$ such that the four Bytes of each Word are linearly combined to form a new Word.

- **KeyExpansion()**
  Key Expansion generates a Round Key (N_b Words) at a time based on the previous Round Key (use the original Cipher Key to generate the first Round Key), for a total of $N_r + 1$ Round Keys, called the Key Schedule. Each Round Key in the Key Schedule will be used in each looping rounds of the algorithm as the updated Cipher Key.

  The KeyExpansion function consists of three separate steps. Figure 3 shows the pseudo code for the KeyExpansion function. The process is as follows: the first four Words of the Key Schedule is filled with the original Cipher Key as the first Round Key. This first Round Key will be used later in the first AddRoundKey module in the AES algorithm before the 9 looping rounds.

  The next 10 Round Keys are generated based on the immediate previous Round Key. Every Word $w[i]$ is generated by XORing $w[i-1]$ and $w[i-N_k]$, where N_k is the number of 32-bit Words in the Cipher Key. In addition, if i is a multiple of N_k, i.e., for the first Word in every Round Key, the Word has to first go through a **`RotWord()`** function then a **`SubWord()`** function, and then XOR the result with the corresponding Word from the **Rcon** table (Round Constant Word array). The **`RotWord()`** function simply cyclically rotate the current Word from $[a_{0,i} \quad a_{1,i} \quad a_{2,i} \quad a_{3,i}]^T$ into $[a_{1,i} \quad a_{2,i} \quad a_{3,i} \quad a_{0,i}]^T$. The **`SubWord()`** function is identical to the **`SubBytes()`** function, where the individual Bytes are substituted using the S-box lookup table (more explanation is provided in the following `SubBytes()` module section). Table 1 shows the Rcon table, which consists of the values given by $[x^{i-1} \quad \{00\} \quad \{00\} \quad \{00\}]^T$, $i = 1 \sim 10$.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
word temp
   i = 0
   while (i < Nk)
       w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
       i = i+1
   end while
   i = Nk
   while (i < Nb * (Nr+1)]
       temp = w[i-1]
       if (i mod Nk = 0)
           temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
       end if
       w[i] = w[i-Nk] xor temp
       i = i + 1
   end while
end
```

Figure 3 Pseudo code for the KeyExpansion module [1].

01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Table 1 Rcon lookup table.

- **AddRoundKey()**
  An N_b-Word Round Key is fetched form the pre-computed Key Schedule where each Byte is bitwise XORed with the corresponding Byte from the updating State. Figure 4 shows the AddRoundKey module.

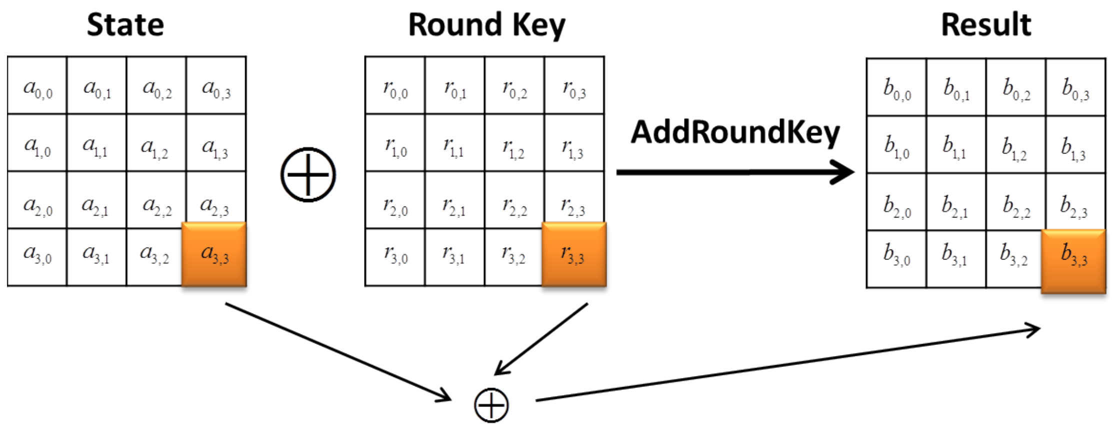
State
Round Key
Result
AddRoundKey
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$r_{0,0}$ $r_{0,1}$ $r_{0,2}$ $r_{0,3}$
$r_{1,0}$ $r_{1,1}$ $r_{1,2}$ $r_{1,3}$
$r_{2,0}$ $r_{2,1}$ $r_{2,2}$ $r_{2,3}$
$r_{3,0}$ $r_{3,1}$ $r_{3,2}$ $r_{3,3}$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$\oplus$

Figure 4 AddRoundKey module.

- **SubBytes()**
  The transformation process for each Byte in the State is pre-calculated and stored into S-box for the encryption process. Table 2 shows the lookup table for the S-box. To apply the transformation, simply loop through each Byte in the State and look up the table using the two hex values of the Byte as the row and column indices of the table to find the transformed value, such that

$$a(i,j) = SBox[a(i,j)],$$

where $a(i,j)$ is the Byte under transformation, and $SBox$ is stored as a 256 Byte consecutive array instead of a 16×16 matrix. Figure 5 shows the SubBytes module. Table 2 shows the lookup table for S-box.

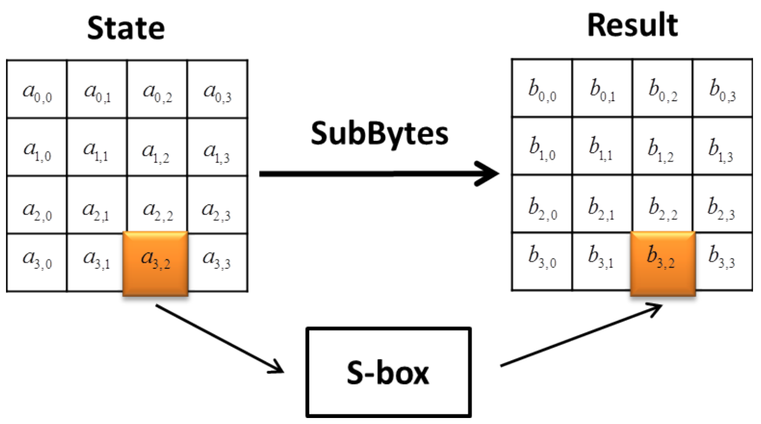
State
Result
SubBytes
S-box
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

Figure 5 AddRoundKey module.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1x	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2x	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3x	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4x	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5x	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6x	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7x	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8x	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9x	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
ax	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
bx	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
cx	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
dx	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
ex	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
fx	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 2 Rijndael S-box.

- **ShiftRows()**
  The rows in the updating State is cyclically shifted by a certain offset. Specifically, row n is left-circularly shifted by $n-1$ Bytes. That is, the first row remains unchanged; the second row is left-circularly by 1 Byte; the third row is left-circularly by 2 Bytes; and the fourth row is left-circularly by 3 Bytes. Figure 6 shows the ShiftRows module.

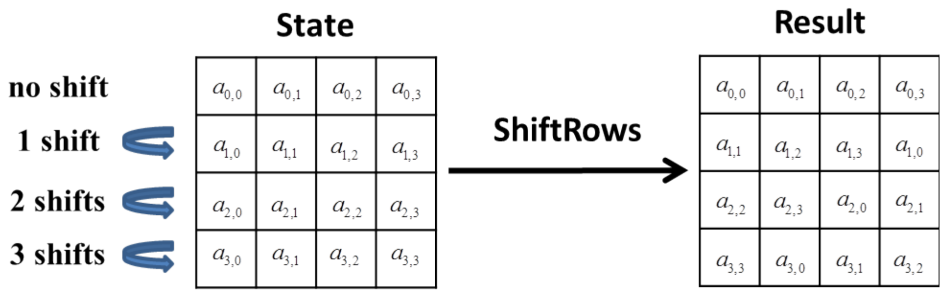
State
Result
no shift
1 shift
2 shifts
3 shifts
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

Figure 6 ShiftRows module.

- **MixColumns()**

Each Word of the updating State is considered as a polynomial over $GF(2^8)$, which is multiplied (denoted by $\bullet$) by a fixed polynomial matrix $c(x)$:

$$\begin{bmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{bmatrix} = \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix} \bullet \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix},$$

where $[a_{0,i} \quad a_{1,i} \quad a_{2,i} \quad a_{3,i}]^T$ is the ith Word in the updating State, such that the resulting column $[b_{0,i} \quad b_{1,i} \quad b_{2,i} \quad b_{3,i}]^T$ is calculated by:

$$\begin{aligned} b_{0,i} &= (\{02\} \bullet a_{0,i}) \oplus (\{03\} \bullet a_{1,i}) \oplus a_{2,i} \oplus a_{3,i} \\ b_{1,i} &= a_{0,i} \oplus (\{02\} \bullet a_{1,i}) \oplus (\{03\} \bullet a_{2,i}) \oplus a_{3,i} \\ b_{2,i} &= a_{0,i} \oplus a_{1,i} \oplus (\{02\} \bullet a_{2,i}) \oplus (\{03\} \bullet a_{3,i}) \\ b_{3,i} &= (\{03\} \bullet a_{0,i}) \oplus a_{1,i} \oplus a_{2,i} \oplus (\{02\} \bullet a_{3,i}). \end{aligned}$$

It is important to keep in mind that the multiplication over $GF(2^8)$ is different from our ordinary multiplication. It involves multiplication of polynomials then modulo an irreducible polynomial of degree 8. Please refer to Chapter 4 of [1] for the mathematical background and the precise calculation steps.

Fortunately, there are shortcuts for performing the multiplication. Multiplication by $\{02\}$ can be implemented at the Byte level as "a left shift, followed by a conditional bitwise XOR with $\{1b\}$ if the 8th bit before the shift is 1." This module is denoted as `xtime()`. All other multiplier values can be recursively deduced from a combination of $(\{02\} \bullet a)$ and a (which is equivalent to $(\{01\} \bullet a)$), by treating the multiplication symbol '$\bullet$' as multiplication, and XOR operator '$\oplus$' as addition. Therefore, $(\{03\} \bullet a)$ is equivalent to XORing $(\{02\} \bullet a)$ with the multiplicand a itself, such that

$$\begin{aligned} \{02\} \bullet a &= \text{xtime()} \\ \{03\} \bullet a &= (\{02\} \bullet a) \oplus a = \text{xtime()} \oplus a. \end{aligned}$$

Figure 7 shows the MixColumns module.

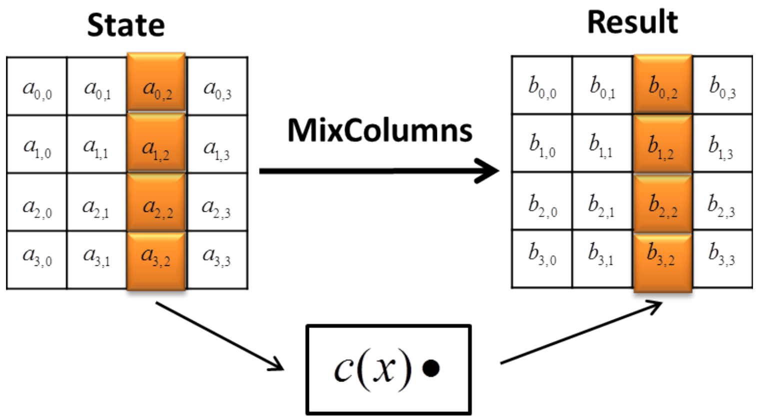
State
Result
MixColumns
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$c(x) \bullet$

Figure 7 MixColumns module.

AES Decryption Algorithm:

- **AES Decryption Pseudo Code**

  Figure 8 shows the pseudo code for the AES decryption algorithm. The general algorithm flow pretty much remains the same, except for the reversed looping rounds and the inverse of the original modules. Figure 9 shows the decryption algorithm flow.

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

Figure 8 Pseudo code for the AES decryption algorithm [1].

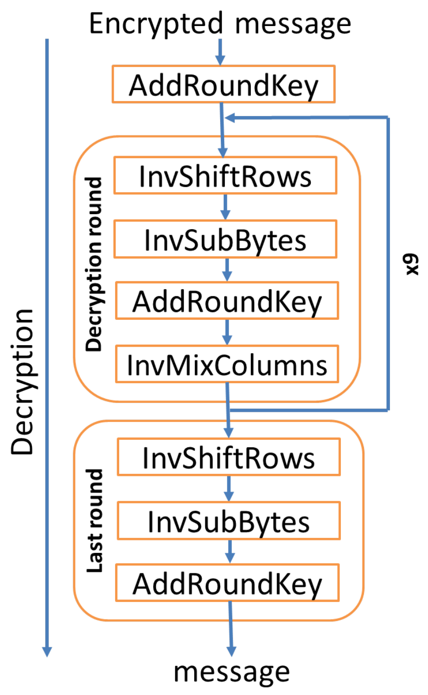

Figure 9 AES decryption algorithm flow.

- **InvSubBytes()**
  The Inverse SubBytes module is identical to the SubBytes module, except that it uses the Inverse S-box for the decryption process. Table 3 shows the lookup table for the Inverse S-box.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1x	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2x	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3x	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4x	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5x	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6x	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7x	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8x	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9x	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
ax	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
bx	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
cx	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
dx	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
ex	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
fx	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Table 3 Rijndael Inverse S-box.

- **InvShiftRows()**
  The Inverse ShiftRows module is identical to the ShiftRows module, except that it now shifts rightwards instead. Specifically, row n is right-circularly shifted by $n-1$ Bytes.

- **InvMixColumns()**
  Each Word of the updating State is considered as a polynomial over $GF(2^8)$, which is multiplied (denoted by $\bullet$) by a another fixed polynomial matrix $c(x)$:

$$\begin{bmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{bmatrix} = \begin{bmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{bmatrix} \bullet \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix},$$

where the resulting column is calculated by:

$$\begin{aligned} b_{0,i} &= (\{0e\} \bullet a_{0,i}) \oplus (\{0b\} \bullet a_{1,i}) \oplus (\{0d\} \bullet a_{2,i}) \oplus (\{09\} \bullet a_{3,i}) \\ b_{1,i} &= (\{09\} \bullet a_{0,i}) \oplus (\{0e\} \bullet a_{1,i}) \oplus (\{0b\} \bullet a_{2,i}) \oplus (\{0d\} \bullet a_{3,i}) \\ b_{2,i} &= (\{0d\} \bullet a_{0,i}) \oplus (\{09\} \bullet a_{1,i}) \oplus (\{0e\} \bullet a_{2,i}) \oplus (\{0b\} \bullet a_{3,i}) \\ b_{3,i} &= (\{0b\} \bullet a_{0,i}) \oplus (\{0d\} \bullet a_{1,i}) \oplus (\{09\} \bullet a_{2,i}) \oplus (\{0e\} \bullet a_{3,i}) \end{aligned}.$$

As described previously, multiplications with $\{09\}$, $\{0b\}$, $\{0d\}$ and $\{0e\}$ can be recursively deduced from a combination of $(\{02\} \bullet a)$ and a, where

$$\begin{aligned}
(\{04\} \bullet a_{0,i}) &= (\{02\} \bullet (\{02\} \bullet a_{0,i})) \\
(\{08\} \bullet a_{0,i}) &= (\{02\} \bullet (\{04\} \bullet a_{0,i})) \\
(\{09\} \bullet a_{0,i}) &= (\{08\} \bullet a_{0,i}) \oplus a_{0,i} \\
(\{0b\} \bullet a_{0,i}) &= (\{08\} \bullet a_{0,i}) \oplus (\{02\} \bullet a_{0,i}) \oplus a_{0,i} \\
(\{0d\} \bullet a_{0,i}) &= (\{08\} \bullet a_{0,i}) \oplus (\{04\} \bullet a_{0,i}) \oplus a_{0,i} \\
(\{0e\} \bullet a_{0,i}) &= (\{08\} \bullet a_{0,i}) \oplus (\{04\} \bullet a_{0,i}) \oplus (\{02\} \bullet a_{0,i}).
\end{aligned}$$

Oh the other hand, all possible multiplication values can be pre-calculated and placed into lookup tables, just like that of the S-box. Table 4 shows the lookup tables for the multiplication with $\{09\}$, $\{0b\}$, $\{0d\}$, and $\{0e\}$, respectively.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	09	12	1b	24	2d	36	3f	48	41	5a	53	6c	65	7e	77
1x	90	99	82	8b	b4	bd	a6	af	d8	d1	ca	c3	fc	f5	ee	e7
2x	3b	32	29	20	1f	16	0d	04	73	7a	61	68	57	5e	45	4c
3x	ab	a2	b9	b0	8f	86	9d	94	e3	ea	f1	f8	c7	ce	d5	dc
4x	76	7f	64	6d	52	5b	40	49	3e	37	2c	25	1a	13	08	01
5x	e6	ef	f4	fd	c2	cb	d0	d9	ae	a7	bc	b5	8a	83	98	91
6x	4d	44	5f	56	69	60	7b	72	05	0c	17	1e	21	28	33	3a
7x	dd	d4	cf	c6	f9	f0	eb	e2	95	9c	87	8e	b1	b8	a3	aa
8x	ec	e5	fe	f7	c8	c1	da	d3	a4	ad	b6	bf	80	89	92	9b
9x	7c	75	6e	67	58	51	4a	43	34	3d	26	2f	10	19	02	0b
ax	d7	de	c5	cc	f3	fa	e1	e8	9f	96	8d	84	bb	b2	a9	a0
bx	47	4e	55	5c	63	6a	71	78	0f	06	1d	14	2b	22	39	30
cx	9a	93	88	81	be	b7	ac	a5	d2	db	c0	c9	f6	ff	e4	ed
dx	0a	03	18	11	2e	27	3c	35	42	4b	50	59	66	6f	74	7d
ex	a1	a8	b3	ba	85	8c	97	9e	e9	e0	fb	f2	cd	c4	df	d6
fx	31	38	23	2a	15	1c	07	0e	79	70	6b	62	5d	54	4f	46

(a)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
	00	0b	16	1d	2c	27	3a	31	58	53	4e	45	74	7f	62	69
1x	b0	bb	a6	ad	9c	97	8a	81	e8	e3	fe	f5	c4	cf	d2	d9
2x	7b	70	6d	66	57	5c	41	4a	23	28	35	3e	0f	04	19	12
3x	cb	c0	dd	d6	e7	ec	f1	fa	93	98	85	8e	bf	b4	a9	a2
4x	f6	fd	e0	eb	da	d1	cc	c7	ae	a5	b8	b3	82	89	94	9f
5x	46	4d	50	5b	6a	61	7c	77	1e	15	08	03	32	39	24	2f
6x	8d	86	9b	90	a1	aa	b7	bc	d5	de	c3	c8	f9	f2	ef	e4
7x	3d	36	2b	20	11	1a	07	0c	65	6e	73	78	49	42	5f	54
8x	f7	fc	e1	ea	db	d0	cd	c6	af	a4	b9	b2	83	88	95	9e
9x	47	4c	51	5a	6b	60	7d	76	1f	14	09	02	33	38	25	2e
ax	8c	87	9a	91	a0	ab	b6	bd	d4	df	c2	c9	f8	f3	ee	e5
bx	3c	37	2a	21	10	1b	06	0d	64	6f	72	79	48	43	5e	55
cx	01	0a	17	1c	2d	26	3b	30	59	52	4f	44	75	7e	63	68
dx	b1	ba	a7	ac	9d	96	8b	80	e9	e2	ff	f4	c5	ce	d3	d8
ex	7a	71	6c	67	56	5d	40	4b	22	29	34	3f	0e	05	18	13
fx	ca	c1	dc	d7	e6	ed	f0	fb	92	99	84	8f	be	b5	a8	a3

(b)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	0d	1a	17	34	39	2e	23	68	65	72	7f	5c	51	46	4b
1x	d0	dd	ca	c7	e4	e9	fe	f3	b8	b5	a2	af	8c	81	96	9b
2x	bb	b6	a1	ac	8f	82	95	98	d3	de	c9	c4	e7	ea	fd	f0
3x	6b	66	71	7c	5f	52	45	48	03	0e	19	14	37	3a	2d	20
4x	6d	60	77	7a	59	54	43	4e	05	08	1f	12	31	3c	2b	26
5x	bd	b0	a7	aa	89	84	93	9e	d5	d8	cf	c2	e1	ec	fb	f6
6x	d6	db	cc	c1	e2	ef	f8	f5	be	b3	a4	a9	8a	87	90	9d
7x	06	0b	1c	11	32	3f	28	25	6e	63	74	79	5a	57	40	4d
8x	da	d7	c0	cd	ee	e3	f4	f9	b2	bf	a8	a5	86	8b	9c	91
9x	0a	07	10	1d	3e	33	24	29	62	6f	78	75	56	5b	4c	41
ax	61	6c	7b	76	55	58	4f	42	09	04	13	1e	3d	30	27	2a
bx	b1	bc	ab	a6	85	88	9f	92	d9	d4	c3	ce	ed	e0	f7	fa
cx	b7	ba	ad	a0	83	8e	99	94	df	d2	c5	c8	eb	e6	f1	fc
dx	67	6a	7d	70	53	5e	49	44	0f	02	15	18	3b	36	21	2c
ex	0c	01	16	1b	38	35	22	2f	64	69	7e	73	50	5d	4a	47
fx	dc	d1	c6	cb	e8	e5	f2	ff	b4	b9	ae	a3	80	8d	9a	97

(c)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	00	0e	1c	12	38	36	24	2a	70	7e	6c	62	48	46	54	5a
1x	e0	ee	fc	f2	d8	d6	c4	ca	90	9e	8c	82	a8	a6	b4	ba
2x	db	d5	c7	c9	e3	ed	ff	f1	ab	a5	b7	b9	93	9d	8f	81
3x	3b	35	27	29	03	0d	1f	11	4b	45	57	59	73	7d	6f	61
4x	ad	a3	b1	bf	95	9b	89	87	dd	d3	c1	cf	e5	eb	f9	f7
5x	4d	43	51	5f	75	7b	69	67	3d	33	21	2f	05	0b	19	17
6x	76	78	6a	64	4e	40	52	5c	06	08	1a	14	3e	30	22	2c
7x	96	98	8a	84	ae	a0	b2	bc	e6	e8	fa	f4	de	d0	c2	cc
8x	41	4f	5d	53	79	77	65	6b	31	3f	2d	23	09	07	15	1b
9x	a1	af	bd	b3	99	97	85	8b	d1	df	cd	c3	e9	e7	f5	fb
ax	9a	94	86	88	a2	ac	be	b0	ea	e4	f6	f8	d2	dc	ce	c0
bx	7a	74	66	68	42	4c	5e	50	0a	04	16	18	32	3c	2e	20
cx	ec	e2	f0	fe	d4	da	c8	c6	9c	92	80	8e	a4	aa	b8	b6
dx	0c	02	10	1e	34	3a	28	26	7c	72	60	6e	44	4a	58	56
ex	37	39	2b	25	0f	01	13	1d	47	49	5b	55	7f	71	63	6d
fx	d7	d9	cb	c5	ef	e1	f3	fd	a7	a9	bb	b5	9f	91	83	8d

(d)

Table 4 Lookup tables for the multiplication with (a) $\{09\}$, (b) $\{0b\}$, (c) $\{0d\}$, and (d) $\{0e\}$.

- **InvAddRoundKey ()**
  Since the original AddRoundKey module only applies an XOR operation, which is its own inverse, the Inverse AddRoundKey module is completely identical to the original AddRoundKey module.

References:

[1] "Announcing the Advanced Encryption Standard (AES)", National Institute of Standards and Technology, 2001. Available at: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf