# MODULE 3: SYMMETRIC CIPHER

- The Advanced Encryption Standard (AES) was published by NIST (National Institute of Standards and Technology) in 2001.
- AES is a symmetric block cipher that is intended to replace DES as the approved standard for a wide range of applications.
- The AES cipher form the latest generation of block ciphers, and now we see a significant increase in the block size - from the old standard of 64-bits up to 128-bits; and keys from 128 to 256-bits.

AES characteristics:

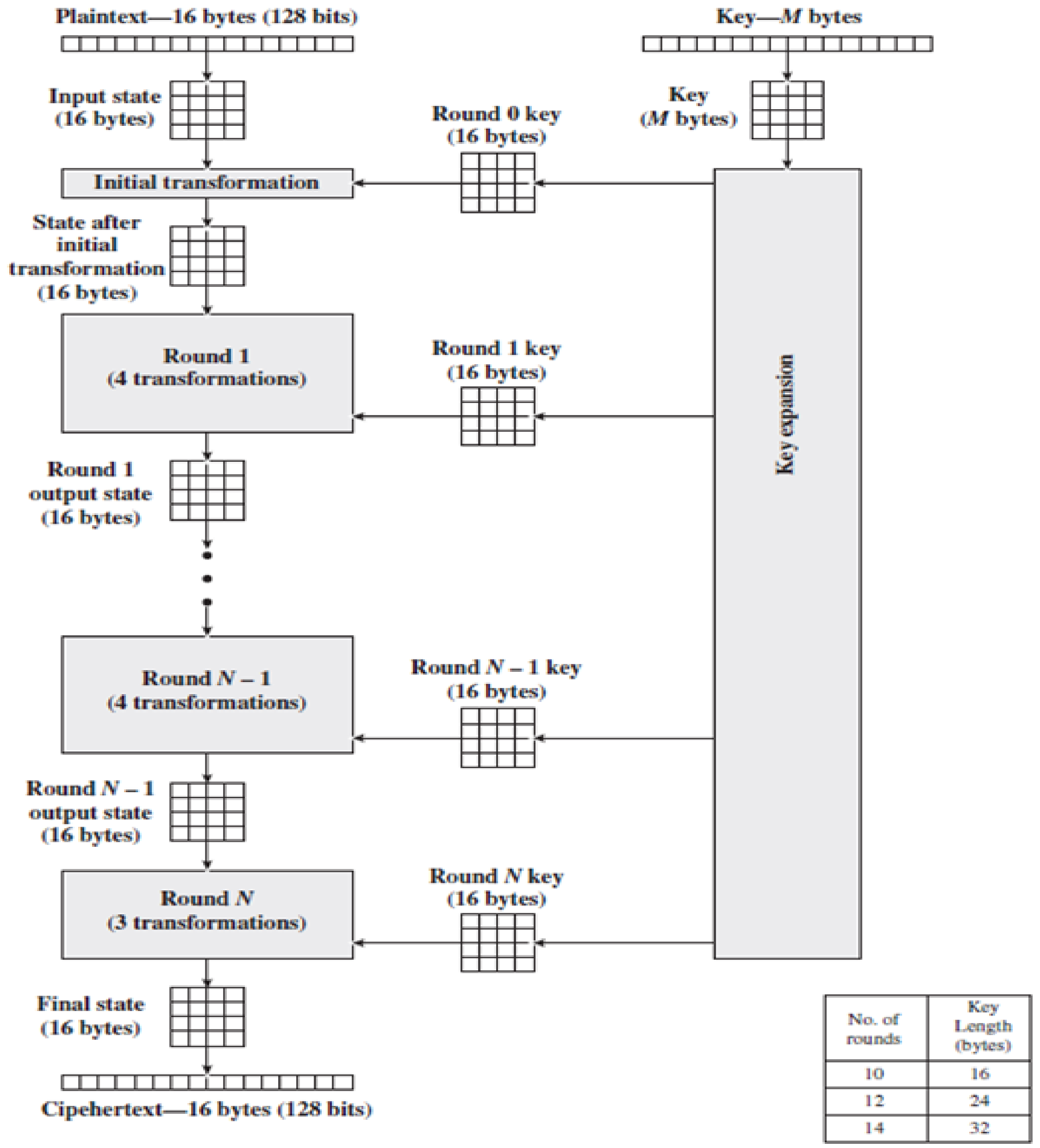
- Resistance against all known attacks,
- Speed and code compactness on a wide range of platforms.
- Design simplicity.

## AES Structure

### General Structure:

- The cipher takes a plaintext block size of 128 bits, or 16 bytes.
- The key length can be 16, 24, or32 bytes (128, 192, or 256 bits). The algorithm is referred to as AES-128, AES-192, orAES-256, depending on the key length.
- The input to the encryption and decryption algorithms is a single **128-bit block.** This block is depicted as a 4 * 4 square matrix of bytes.
- This block is copied into the **State** array, which is modified at each stage of encryption or decryption. After the final stage, **State** is copied to an output matrix. These operations are depicted in Figure 5.2a.
- Similarly, the key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words. Figure 5.2b shows the expansion for the 128-bit key.
- Each word is four bytes, and the total key schedule is 44 words for the **128-bit key**. Note that the ordering of bytes within a matrix is by column.
- So, for example, the first four bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the **in** matrix, the second four bytes occupy the second column, and so on.
- Similarly, the first four bytes of the expanded key, which form a word, occupy the first column of the **w** matrix.
- The cipher consists of *N* rounds, where the number of rounds depends on the key length: 10 rounds for a 16-byte key, 12 rounds for a 24-byte key, and 14 rounds for a 32-byte key (Table 5.1). The first *N* - 1 rounds consist of four distinct transformation functions**: SubBytes, ShiftRows, MixColumns, and AddRoundKey**.
- The final round contains only three transformations, and there is a initial single transformation (AddRoundKey) before the first round, which can be considered Round 0.

- Each transformation takes one or more 4 * 4 matrices as input and produces a 4 * 4 matrix as output.
- Figure 5.1 shows that the output of each round is a 4 * 4 matrix, with the output of the final round being the cipher text. Also, the key expansion function generates *N* + 1 round keys, each of which is a distinct4 * 4 matrix.
- Each round key serves as one of the inputs to the Add Round Key transformation in each round.


Plaintext—16 bytes (128 bits)
Key—M bytes
Input state (16 bytes)
Key (M bytes)
Round 0 key (16 bytes)
Initial transformation
State after initial transformation (16 bytes)
Round 1 (4 transformations)
Round 1 key (16 bytes)
Key expansion
Round 1 output state (16 bytes)
Round N – 1 (4 transformations)
Round N – 1 key (16 bytes)
Round N – 1 output state (16 bytes)
Round N (3 transformations)
Round N key (16 bytes)
Final state (16 bytes)
Cipehertext—16 bytes (128 bits)

| No. of rounds | Key Length (bytes) |
|---|---|
| 10 | 16 |
| 12 | 24 |
| 14 | 32 |

Figure 5.1 AES Encryption Process

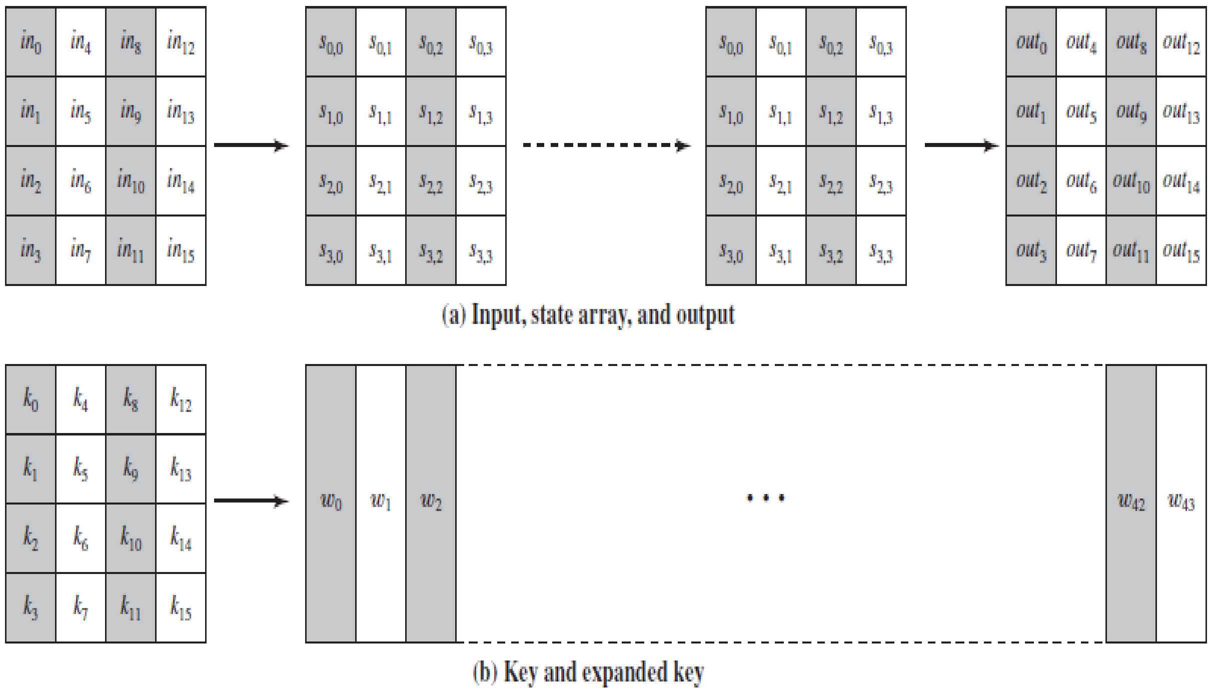


Figure 5.2 AES Data Structures

Table 5.1 AES Parameters

| | | | |
|---|---|---|---|
| **Key Size (words/bytes/bits)** | 4/16/128 | 6/24/192 | 8/32/256 |
| **Plaintext Block Size (words/bytes/bits)** | 4/16/128 | 4/16/128 | 4/16/128 |
| **Number of Rounds** | 10 | 12 | 14 |
| **Round Key Size (words/bytes/bits)** | 4/16/128 | 4/16/128 | 4/16/128 |
| **Expanded Key Size (words/bytes)** | 44/176 | 52/208 | 60/240 |

**Detailed Structure:**

Figure 5.3 shows the AES cipher in more detail, indicating the sequence of transformations in each round and showing the corresponding decryption function.
Discussion about the overall AES structure:

- This structure will not belongs to Feistel structure. In the classic Feistel structure, half of the data block is used to modify the other half of the data block and then the halves are swapped. AES instead processes the entire data block as a single matrix during each round using substitutions and permutation.

- The key that is provided as input is expanded into an array of forty-four 32-bit words, **w**[*i*]. Four distinct words (128 bits) serve as a round key for each round; these are indicated in Figure 5.3.

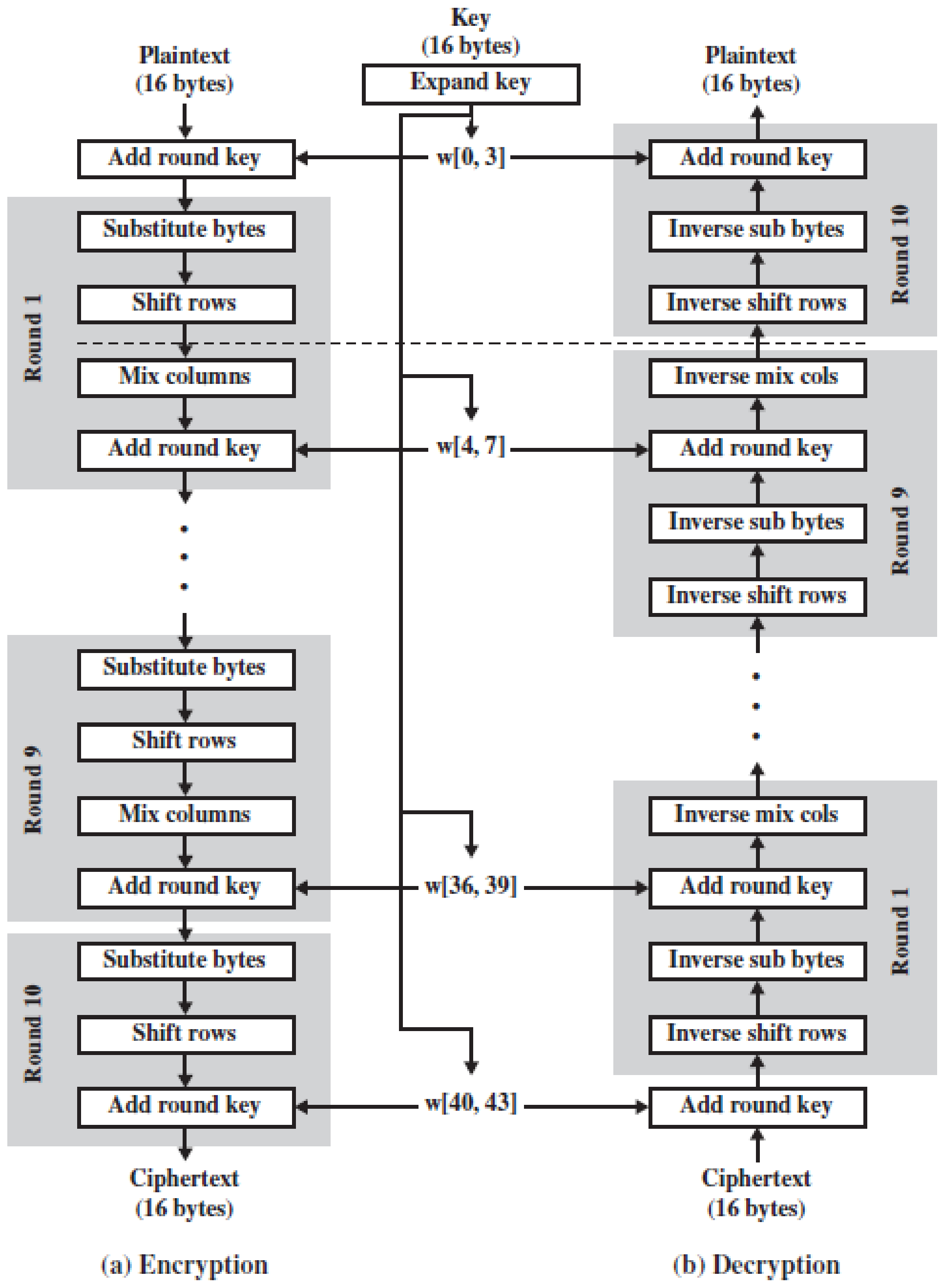


Figure 5.3 AES Encryption and Decryption

- Four different stages are used, one of permutation and three of substitution:
  • **Substitute bytes:** Uses an S-box to perform a byte-by-byte substitution of the block.
  • **ShiftRows:** A simple permutation.
  • **MixColumns:** A substitution that makes use of arithmetic over GF($2^8$).
  • **AddRoundKey:** A simple bitwise XOR of the current block with a portion of the expanded key.

- The structure is quite simple. For both encryption and decryption, the cipher begins with an Add Round Key stage, followed by nine rounds that each includes all four stages, followed by a tenth round of three stages. Figure 5.4 depicts the structure of a full encryption round.
- Only the Add Round Key stage makes use of the key. For this reason, the cipher begins and ends with an Add Round Key stage.
- The Add Round Key stage is, in effect, a form of Vernam cipher and by itself would not be formidable. The other three stages together provide confusion, diffusion, and nonlinearity, but by themselves would provide no security because they do not use the key. The cipher as alternating operations of XOR encryption (Add Round Key) of a block, followed by scrambling of the block (the other three stages), followed by XOR encryption, and so on. This scheme is both efficient and highly secure.
- Each stage is easily reversible. For the Substitute Byte, Shift Rows, and Mix Columns stages, an inverse function is used in the decryption algorithm .For the Add Round Key stage, the inverse is achieved by XORing the same round key to the block, using the result that $A \oplus B \oplus B = A$.
- As with most block ciphers, the decryption algorithm makes use of the expanded key in reverse order. However, the decryption algorithm is not identical to the encryption algorithm. This is a consequence of the particular structure of AES.
- Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. Figure 5.3 lays out encryption and decryption going in opposite vertical directions. At each horizontal point (e.g., the dashed line in the figure), **State** is the same for both encryption and decryption.
- The final round of both encryption and decryption consists of only three stages. Again, this is a consequence of the particular structure of AES and is required to make the cipher reversible.

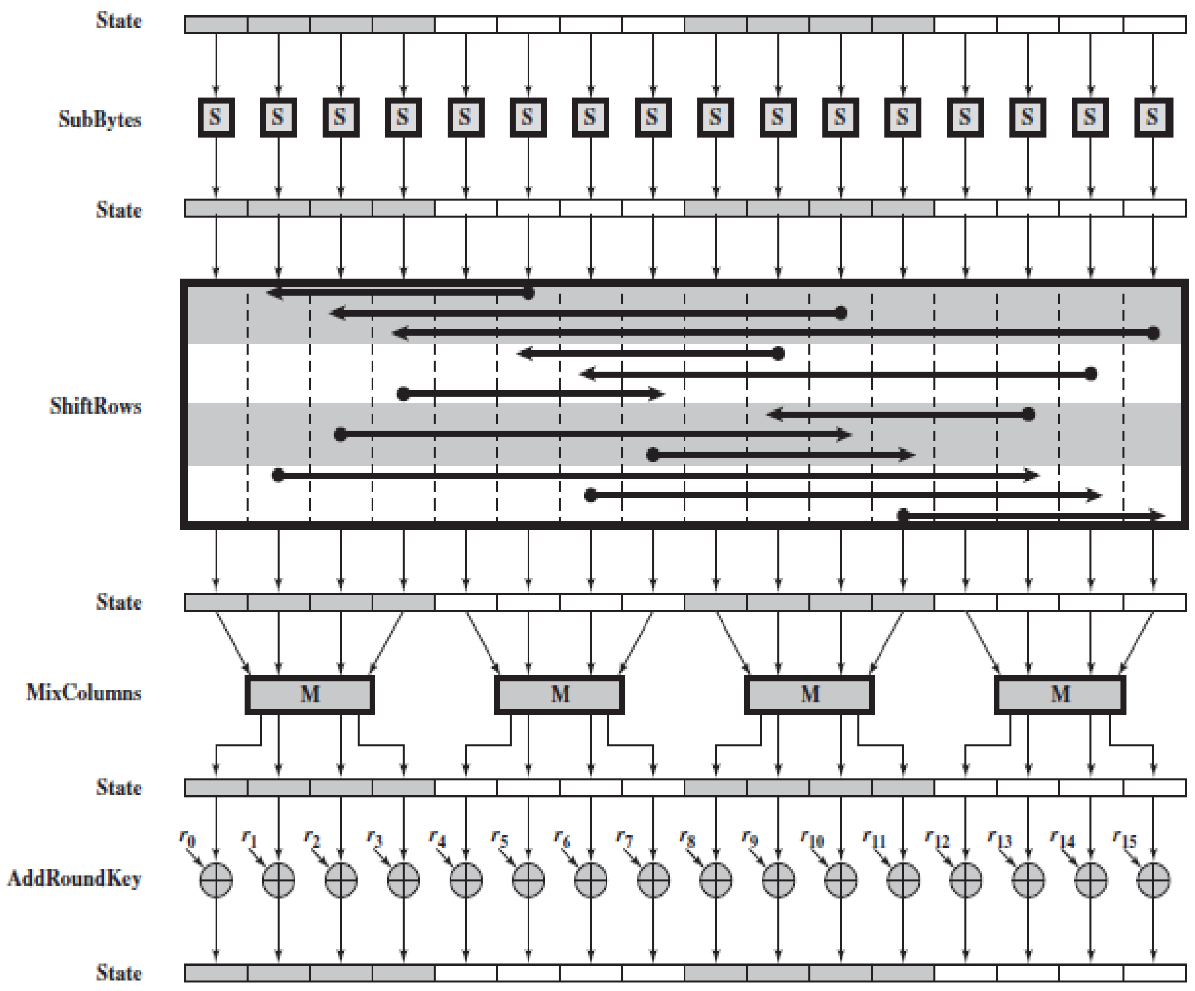
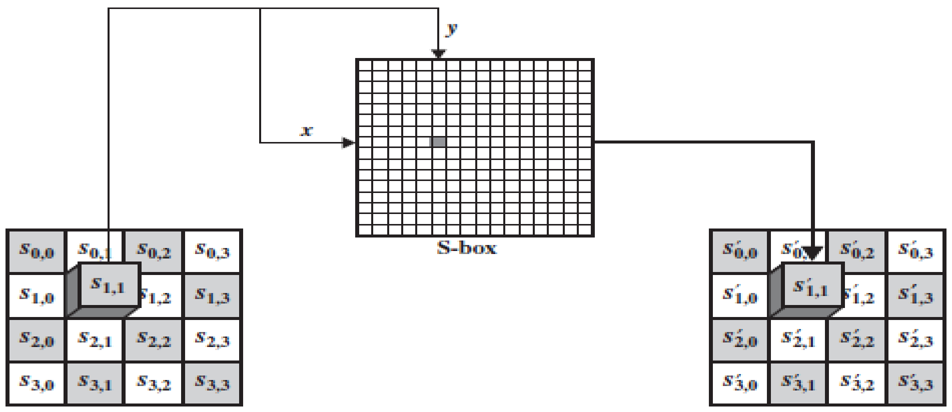

State
SubBytes
S
State
ShiftRows
State
MixColumns
M
State
AddRoundKey
$r_0$ $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$ $r_7$ $r_8$ $r_9$ $r_{10}$ $r_{11}$ $r_{12}$ $r_{13}$ $r_{14}$ $r_{15}$
State

Figure 5.4 AES Encryption Round

**AES Transformation Functions:**
For each stage, we describe the forward (encryption) algorithm, the inverse (decryption) algorithm, and the rationale for the stage.
**Substitute Bytes Transformation:-**
**Forward and Inverse Transformations:** The **forward substitute byte transformation**, called SubBytes, is a simple table lookup (Figure 5.5a).

- ✓ AES defines a 16 * 16 matrix of byte values, called an S-box (Table 5.2a), that contains a permutation of all possible 256 8-bit values.
- ✓ Each individual byte of **State** is mapped into a new byte in the following way: 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 unique8-bit output value.


y
x
S-box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

**(a) Substitute byte transformation**

- ✓ For example, the hexadecimal value {95} references row 9,column 5 of the S-box, which contains the value {2A}. Accordingly, the value {95} is mapped into the value {2A}.

| | | | |
|---|---|---|---|
| EA | 04 | 65 | 85 |
| 83 | 45 | 5D | 96 |
| 5C | 33 | 98 | B0 |
| F0 | 2D | AD | C5 |

$\rightarrow$

| | | | |
|---|---|---|---|
| 87 | F2 | 4D | 97 |
| EC | 6E | 4C | 90 |
| 4A | C3 | 46 | E7 |
| 8C | D8 | 95 | A6 |

Table 5.2 AES S-Boxes

| | | y | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 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 |
| x | 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 |

(a) S-box

| | | y | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| | 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 |
| x | 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 |

(b) Inverse S-box

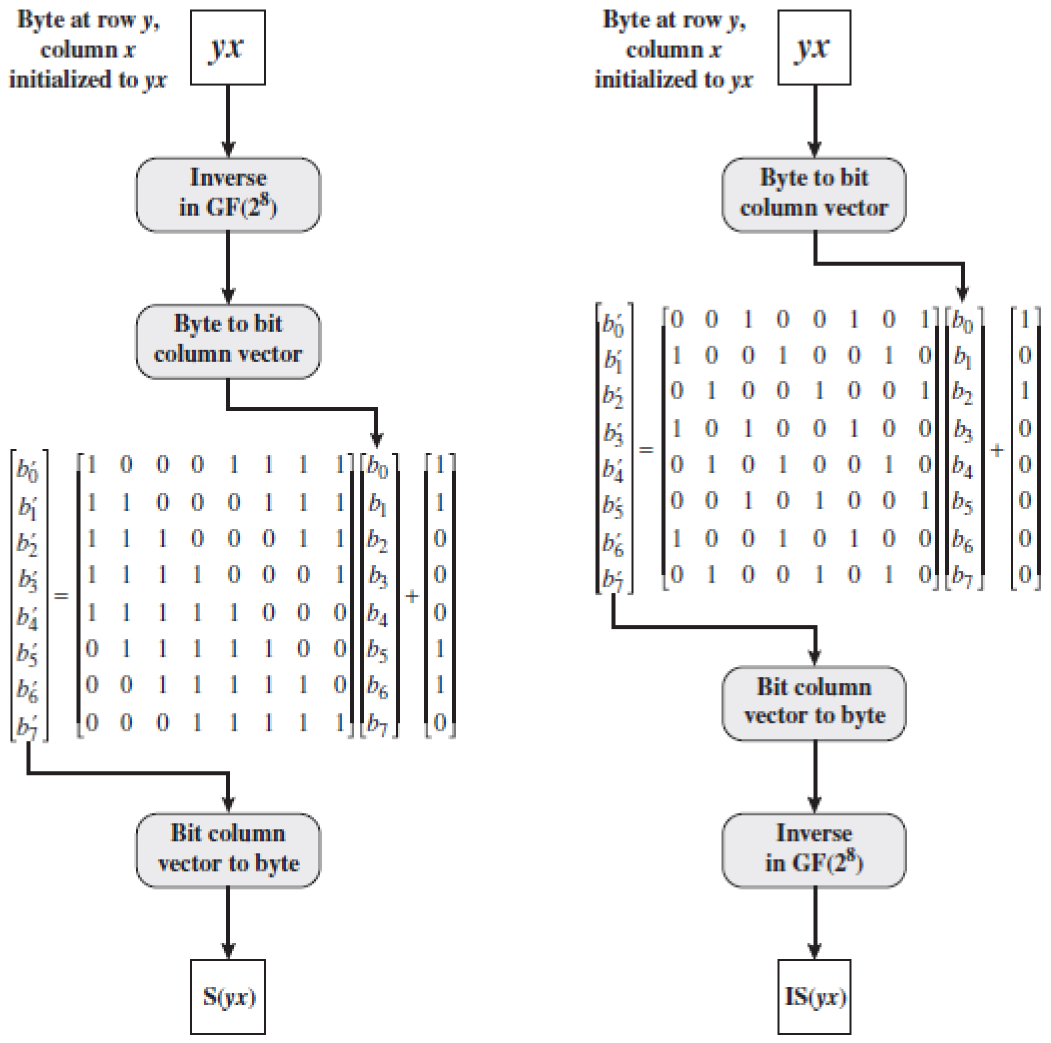
Here is an example of the SubBytes transformation:

| | | | |
|---|---|---|---|
| EA | 04 | 65 | 85 |
| 83 | 45 | 5D | 96 |
| 5C | 33 | 98 | B0 |
| F0 | 2D | AD | C5 |

→

| | | | |
|---|---|---|---|
| 87 | F2 | 4D | 97 |
| EC | 6E | 4C | 90 |
| 4A | C3 | 46 | E7 |
| 8C | D8 | 95 | A6 |

The S-box is constructed in the following fashion (Figure 5.6a).


Byte at row y, column x initialized to yx
yx
Inverse in GF(2^8)
Byte to bit column vector
Bit column vector to byte
S(yx)
Byte at row y, column x initialized to yx
yx
Byte to bit column vector
Bit column vector to byte
Inverse in GF(2^8)
IS(yx)

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

**(a) Calculation of byte at row *y*, column *x* of S-box**

**(a) Calculation of byte at row *y*, column *x* of IS-box**

Figure 5.6 Constuction of S-Box and IS-Box

**1.** Initialize the S-box with the byte values in ascending sequence row by row.The first row contains {00}, {01}, {02}, c, {0F}; the second row contains{10}, {11}, etc.; and so on. Thus, the value of the byte at row *y*, column *x* is {*yx*}.
**2.** Map each byte in the S-box to its multiplicative inverse in the finite fieldGF($2^8$); the value {00} is mapped to itself.
**3.** Consider that each byte in the S-box consists of 8 bits labeled (b7, b6, b5, b4, b3,b2, b1, b0). Apply the following transformation to each bit of each byte in the S-box:.

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \qquad \textbf{(5.1)}$$

Where *ci* is the *i*th bit of byte *c* with the value {63}; that is, (*c*7*c*6*c*5*c*4*c*3*c*2*c*1*c*0) =(01100011). The prime ( `) indicates that the variable is to be updated by the value on the right. The AES standard depicts this transformation in matrix form as follows.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad \textbf{(5.2)}$$

As an example, consider the input value {95}. The multiplicative inverse in GF($2^8$) is $\{95\}^{-1} = \{8A\}$, which is 10001010 in binary.

$$\begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

The result is {2A}, which should appear in row {09} column {05} of the S-box.This is verified by checking Table 5.2a.
The **inverse substitute byte transformation**, called InvSubBytes, makes useof the inverse S-box shown in Table 5.2b. Note, for example, that the input {2A}produces the output {95}, and the input {95} to the S-box produces {2A}. The inverseS-box is constructed (Figure 5.6b) by applying the inverse of the transformation inEquation (5.1) followed by taking the multiplicative inverse in GF($2^8$). The inversetransformation is,

$$b'_i = b_{(i+2) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus d_i$$

where byte *d* = {05}, or 00000101. We can depict this transformation as follows.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

To see that InvSubBytes is the inverse of SubBytes, label the matrices in SubBytes and InvSubBytes as **X** and **Y**, respectively, and the vector versions of constants c and d as **C** and **D**, respectively. For some 8-bit vector **B**, Equation (5.2) becomes $\mathbf{B}' = \mathbf{XB} \oplus \mathbf{C}$. We need to show that $\mathbf{Y}(\mathbf{XB} \oplus \mathbf{C}) \oplus \mathbf{D} = \mathbf{B}$. To multiply out, we must show $\mathbf{YXB} \oplus \mathbf{YC} \oplus \mathbf{D} = \mathbf{B}$. This becomes

$$\begin{bmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{bmatrix} \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus$$

$$\begin{bmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} =$$

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

We have demonstrated that $\mathbf{YX}$ equals the identity matrix, and the $\mathbf{YC} = \mathbf{D}$, so that $\mathbf{YC} \oplus \mathbf{D}$ equals the null vector.

**Rationale** The S-box is designed to be resistant to known cryptanalytic attacks. Specifically, the Rijndael developers sought a design that has a low correlation between input bits and output bits and the property that the output is not a linear mathematical function of the input.The S-box must be invertible, that is, IS-box[S-box(*a*)] = *a*. However, the S-box does not self-inverse in the sense that it is not true that S-box(*a*) = IS-box(*a*). For example, S-box({95}) = {2A}, but IS-box({95}) = {AD}.

**ShiftRows Transformation:-**

**Forward and Inverse Transformations:** The **forward shift row transformation** called ShiftRows, is depicted in Figure 5.7a. The first row of **State** is not altered. Forthe second row, a 1-byte circular left shift is performed. For the third row, a 2-bytecircular left shift is performed. For the fourth row, a 3-byte circular left shift is performed. The following is an example of ShiftRows.

| | | | |
|---|---|---|---|
| 87 | F2 | 4D | 97 |
| EC | 6E | 4C | 90 |
| 4A | C3 | 46 | E7 |
| 8C | D8 | 95 | A6 |

→

| | | | |
|---|---|---|---|
| 87 | F2 | 4D | 97 |
| 6E | 4C | 90 | EC |
| 46 | E7 | 4A | C3 |
| A6 | 8C | D8 | 95 |

The **inverse shift row transformation**, called Inv Shift Rows, performs the circular shifts in the opposite direction for each of the last three rows, with a 1-byte circular right shift for the second row, and so on.

**Rationale** The shift row transformation is more substantial than it may first appear. This is because the **State**, as well as the cipher input and output, is treated as an array of four 4-byte columns. Thus, on encryption, the first 4 bytes of the plaintext are copied to the first column of **State**, and so on. Furthermore, as will be seen, the round key is applied to **State** column by column. Thus, a row shift moves an individual byte from one column to another, which is a linear distance of a multiple of 4 bytes. Also note that the transformation ensures that the 4 bytes of one column are spread out to four different columns. Figure 5.4 illustrates the effect.

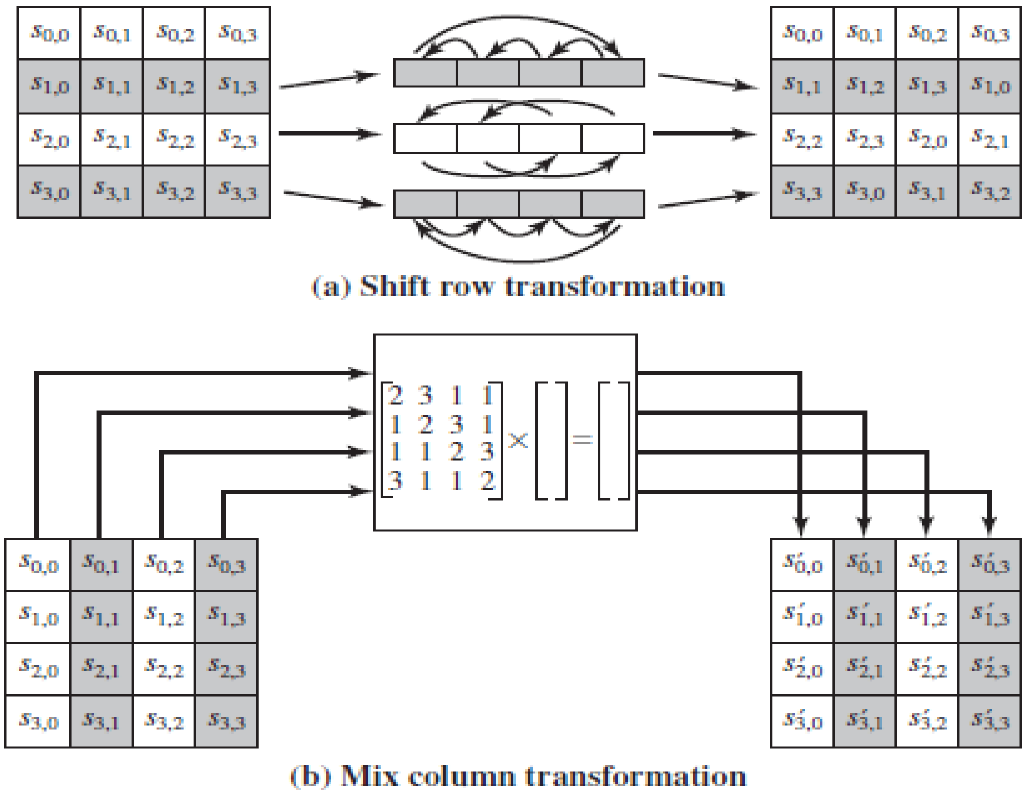


Figure 5.7 AES Row and Column Operations

**MixColumns Transformation:-**

**Forward and Inverse Transformations:** The **forward mix column transformation**, called Mix Columns, operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in that column. The transformation can be defined by the following matrix multiplication on **State** (Figure 5.7b):

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \quad \textbf{(5.3)}$$

The MixColumns transformation on a single column of **State** can be expressed as,

$$\begin{aligned}
s'_{0,j} &= (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\
s'_{1,j} &= s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j} \oplus s_{3,j} \\
s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j}) \\
s'_{3,j} &= (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j})
\end{aligned} \qquad \textbf{(5.4)}$$

The following is an example of MixColumns:

| | | | |
|---|---|---|---|
| 87 | F2 | 4D | 97 |
| 6E | 4C | 90 | EC |
| 46 | E7 | 4A | C3 |
| A6 | 8C | D8 | 95 |

$\rightarrow$

| | | | |
|---|---|---|---|
| 47 | 40 | A3 | 4C |
| 37 | D4 | 70 | 9F |
| 94 | E4 | 3A | 42 |
| ED | A5 | A6 | BC |

Let us verify the first column of this example. In GF($2^8$), addition is the bitwise XOR operation and that multiplication can be performed according to the rule In particular, multiplication of a value by *x* (i.e., by {02}) can be implemented as a 1-bit left shift followed by a conditional bitwise XOR with (0001 1011) if the leftmost bit of the original value (prior to the shift) is 1. Thus, to verify the MixColumns transformation on the first column, we need to show that,

$$\begin{array}{lllll}
(\{02\} \cdot \{87\}) & \oplus (\{03\} \cdot \{6E\}) & \oplus \{46\} & \oplus \{A6\} & = \{47\} \\
\{87\} & \oplus (\{02\} \cdot \{6E\}) & \oplus (\{03\} \cdot \{46\}) & \oplus \{A6\} & = \{37\} \\
\{87\} & \oplus \{6E\} & \oplus (\{02\} \cdot \{46\}) & \oplus (\{03\} \cdot \{A6\}) & = \{94\} \\
(\{03\} \cdot \{87\}) & \oplus \{6E\} & \oplus \{46\} & \oplus (\{02\} \cdot \{A6\}) & = \{ED\}
\end{array}$$

For the first equation, we have $\{02\} \cdot \{87\} = (0000\,1110) \oplus (0001\,1011) = (0001\,0101)$ and $\{03\} \cdot \{6E\} = \{6E\} \oplus (\{02\} \cdot \{6E\}) = (0110\,1110) \oplus (1101\,1100) = (1011\,0010)$. Then,

$$\begin{aligned}
\{02\} \cdot \{87\} &= 0001\,0101 \\
\{03\} \cdot \{6E\} &= 1011\,0010 \\
\{46\} &= 0100\,0110 \\
\{A6\} &= \underline{1010\,0110} \\
& \phantom{=} 0100\,0111 = \{47\}
\end{aligned}$$

The other equations can be similarly verified.

The **inverse mix column transformation**, called InvMixColumns, is defined by the following matrix multiplication:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \qquad \textbf{(5.5)}$$

It is not immediately clear that Equation (5.5) is the **inverse** of Equation (5.3). We need to show

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

which is equivalent to showing

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \textbf{(5.6)}$$

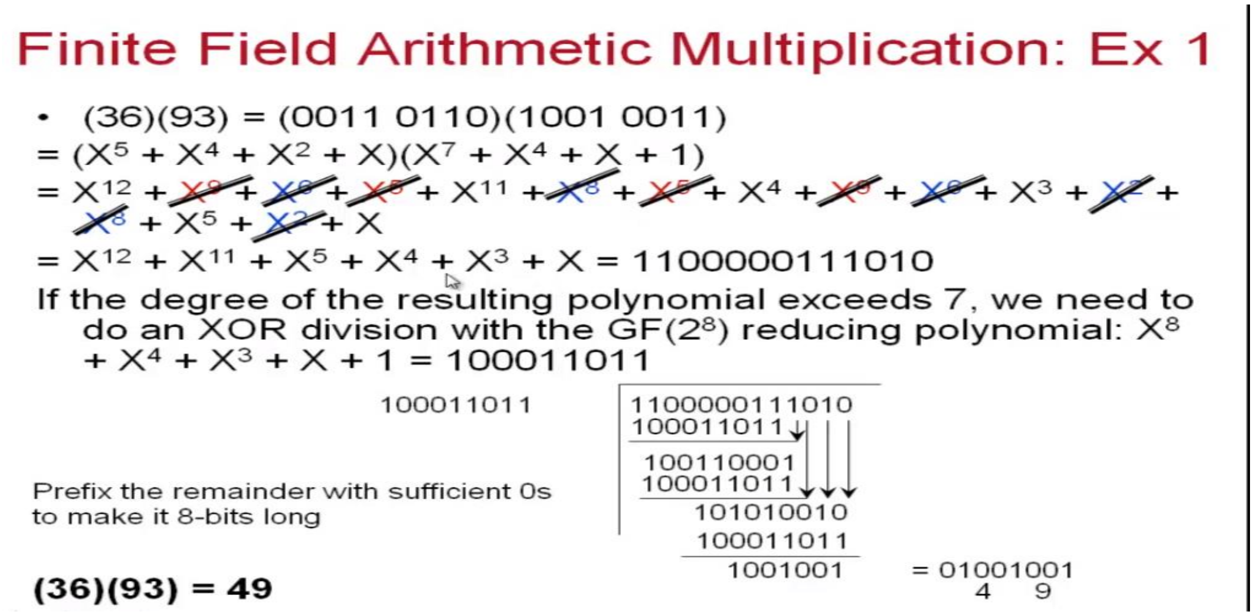
That is, the inverse transformation matrix times the forward transformation matrix equals the identity matrix. To verify the first column of Equation (5.6), we need to show

$$\begin{aligned}
(\{0E\} \cdot \{02\}) \oplus \{0B\} \oplus \{0D\} \oplus (\{09\} \cdot \{03\}) &= \{01\} \\
(\{09\} \cdot \{02\}) \oplus \{0E\} \oplus \{0B\} \oplus (\{0D\} \cdot \{03\}) &= \{00\} \\
(\{0D\} \cdot \{02\}) \oplus \{09\} \oplus \{0E\} \oplus (\{0B\} \cdot \{03\}) &= \{00\} \\
(\{0B\} \cdot \{02\}) \oplus \{0D\} \oplus \{09\} \oplus (\{0E\} \cdot \{03\}) &= \{00\}
\end{aligned}$$

For the first equation, we have $\{0E\} \cdot \{02\} = 00011100$ and $\{09\} \cdot \{03\} = \{09\} \oplus (\{09\} \cdot \{02\}) = 00001001 \oplus 00010010 = 00011011$. Then

$$\begin{aligned}
\{0E\} \cdot \{02\} &= 00011100 \\
\{0B\} &= 00001011 \\
\{0D\} &= 00001101 \\
\{09\} \cdot \{03\} &= \underline{00011011} \\
&\phantom{=} 00000001
\end{aligned}$$

For students reference



## Finite Field Arithmetic Multiplication: Ex 1

- (36)(93) = (0011 0110)(1001 0011)

$= (X^5 + X^4 + X^2 + X)(X^7 + X^4 + X + 1)$

$= X^{12} + \cancel{X^9} + \cancel{X^6} + \cancel{X^5} + X^{11} + \cancel{X^8} + \cancel{X^5} + X^4 + \cancel{X^9} + \cancel{X^6} + X^3 + \cancel{X^2} + \cancel{X^8} + X^5 + \cancel{X^2} + X$

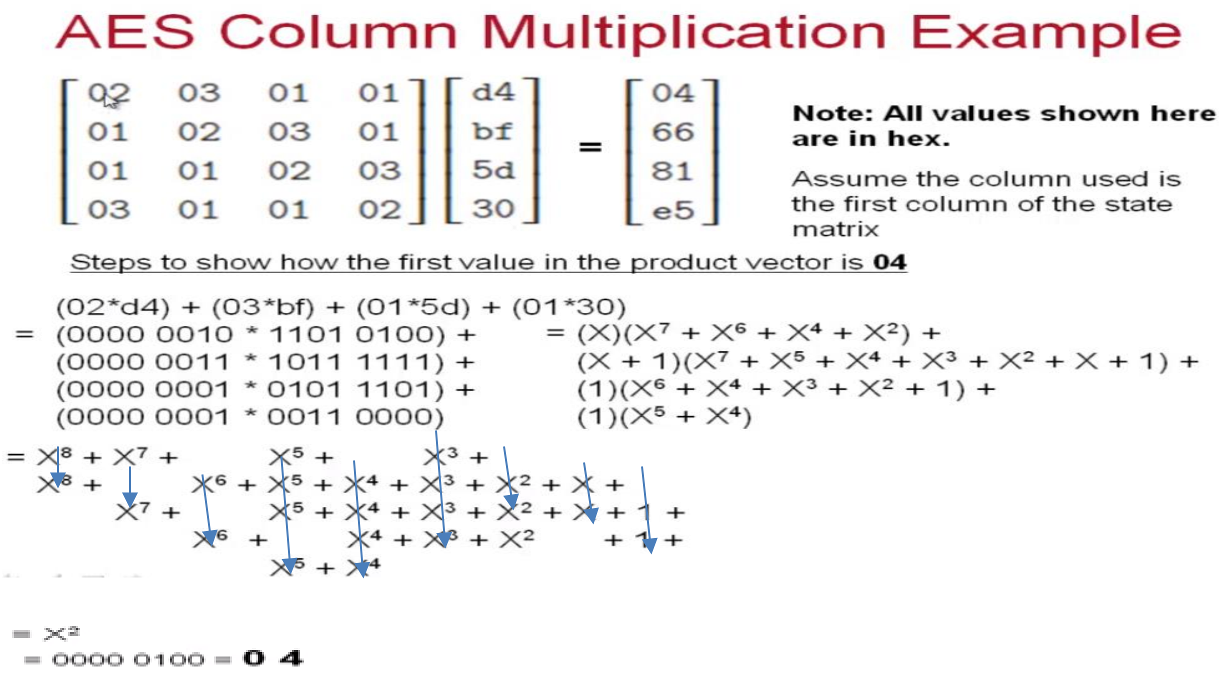
$= X^{12} + X^{11} + X^5 + X^4 + X^3 + X = 1100000111010$

If the degree of the resulting polynomial exceeds 7, we need to do an XOR division with the GF($2^8$) reducing polynomial: $X^8 + X^4 + X^3 + X + 1$ = 100011011

```
100011011   1100000111010
            100011011
             100110001
             100011011
                101010010
                100011011
                  1001001     = 01001001
                                 4   9
```

Prefix the remainder with sufficient 0s to make it 8-bits long

**(36)(93) = 49**



## AES Column Multiplication Example

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} d4 \\ bf \\ 5d \\ 30 \end{bmatrix} = \begin{bmatrix} 04 \\ 66 \\ 81 \\ e5 \end{bmatrix}$$

**Note: All values shown here are in hex.**

Assume the column used is the first column of the state matrix

Steps to show how the first value in the product vector is **04**

(02*d4) + (03*bf) + (01*5d) + (01*30)

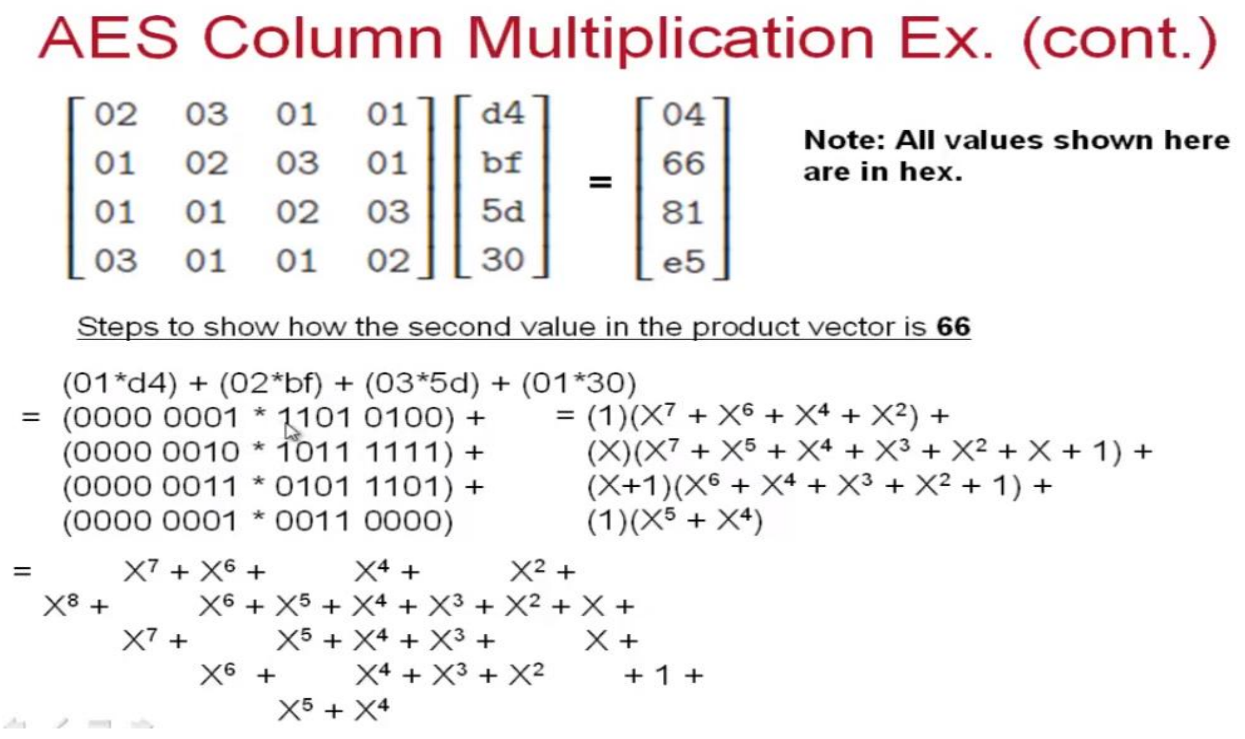
= (0000 0010 * 1101 0100) + (0000 0011 * 1011 1111) + (0000 0001 * 0101 1101) + (0000 0001 * 0011 0000)

$= (X)(X^7 + X^6 + X^4 + X^2) + (X + 1)(X^7 + X^5 + X^4 + X^3 + X^2 + X + 1) + (1)(X^6 + X^4 + X^3 + X^2 + 1) + (1)(X^5 + X^4)$

$= \cancel{X^8} + \cancel{X^7} + \cancel{X^5} + \cancel{X^3} +$
$\cancel{X^8} + \cancel{X^6} + \cancel{X^5} + \cancel{X^4} + \cancel{X^3} + \cancel{X^2} + \cancel{X} +$
$\cancel{X^7} + \cancel{X^5} + \cancel{X^4} + \cancel{X^3} + \cancel{X^2} + \cancel{X} + \cancel{1} +$
$\cancel{X^6} + \cancel{X^4} + \cancel{X^3} + X^2 + \cancel{1} +$
$\cancel{X^5} + \cancel{X^4}$

$= X^2$

= 0000 0100 = **0 4**



## AES Column Multiplication Ex. (cont.)

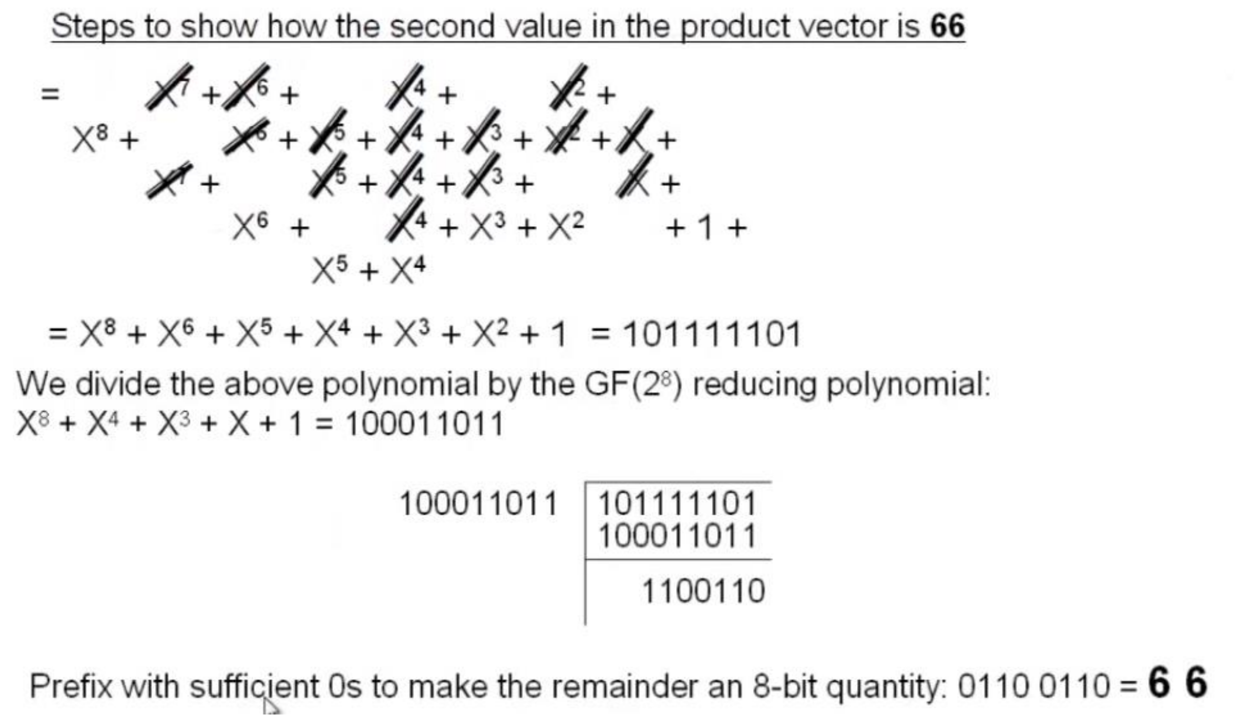
$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} d4 \\ bf \\ 5d \\ 30 \end{bmatrix} = \begin{bmatrix} 04 \\ 66 \\ 81 \\ e5 \end{bmatrix}$$

**Note: All values shown here are in hex.**

Steps to show how the second value in the product vector is **66**

$$\begin{aligned}
&(01*d4) + (02*bf) + (03*5d) + (01*30) \\
= &(0000\ 0001 * 1101\ 0100) + \quad = (1)(X^7 + X^6 + X^4 + X^2) + \\
&(0000\ 0010 * 1011\ 1111) + \quad\ \ (X)(X^7 + X^5 + X^4 + X^3 + X^2 + X + 1) + \\
&(0000\ 0011 * 0101\ 1101) + \quad\ \ (X+1)(X^6 + X^4 + X^3 + X^2 + 1) + \\
&(0000\ 0001 * 0011\ 0000) \qquad\ \ (1)(X^5 + X^4)
\end{aligned}$$

$$\begin{aligned}
= \quad & X^7 + X^6 + X^4 + X^2 + \\
& X^8 + X^6 + X^5 + X^4 + X^3 + X^2 + X + \\
& X^7 + X^5 + X^4 + X^3 + X + \\
& X^6 + X^4 + X^3 + X^2 + 1 + \\
& X^5 + X^4
\end{aligned}$$



Steps to show how the second value in the product vector is **66**

$$\begin{aligned}
= \quad & \cancel{X^7} + \cancel{X^6} + \cancel{X^4} + \cancel{X^2} + \\
& X^8 + \cancel{X^6} + \cancel{X^5} + \cancel{X^4} + \cancel{X^3} + \cancel{X^2} + \cancel{X} + \\
& \cancel{X^7} + \cancel{X^5} + \cancel{X^4} + \cancel{X^3} + \cancel{X} + \\
& X^6 + \cancel{X^4} + X^3 + X^2 + 1 + \\
& X^5 + X^4
\end{aligned}$$

$$= X^8 + X^6 + X^5 + X^4 + X^3 + X^2 + 1 = 101111101$$

We divide the above polynomial by the GF($2^8$) reducing polynomial:
$X^8 + X^4 + X^3 + X + 1 = 100011011$

$$\begin{array}{r|r} 100011011 & 101111101 \\ & 100011011 \\ \hline & 1100110 \end{array}$$

Prefix with sufficient 0s to make the remainder an 8-bit quantity: 0110 0110 = **6 6**

## Add Round Key Transformation:-

**Forward and Inverse Transformations** In the **forward add round key transformation** called Add Round Key, the 128 bits of **State** are bitwise XORed with the 128bits of the round key. As shown in Figure 5.5b, the operation is viewed as a column wise operation between the 4 bytes of a **State** column and one word of the roundkey; it can also be viewed as a byte-level operation. The following is an example of AddRoundKey:

$$\begin{array}{|c|c|c|c|}\hline 47 & 40 & A3 & 4C \\ \hline 37 & D4 & 70 & 9F \\ \hline 94 & E4 & 3A & 42 \\ \hline ED & A5 & A6 & BC \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|}\hline AC & 19 & 28 & 57 \\ \hline 77 & FA & D1 & 5C \\ \hline 66 & DC & 29 & 00 \\ \hline F3 & 21 & 41 & 6A \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline EB & 59 & 8B & 1B \\ \hline 40 & 2E & A1 & C3 \\ \hline F2 & 38 & 13 & 42 \\ \hline 1E & 84 & E7 & D6 \\ \hline \end{array}$$

The first matrix is **State**, and the second matrix is the round key. The **inverse add round key transformation** is identical to the forward add round key transformation, because the XOR operation is its own inverse.

**Rationale:** The add round key transformation is as simple as possible and affects every bit of **State**. The complexity of the round key expansion, plus the complexity of the other stages of AES, ensure security .Figure 5.8 is another view of a single round of AES, emphasizing the mechanisms and inputs of each transformation.

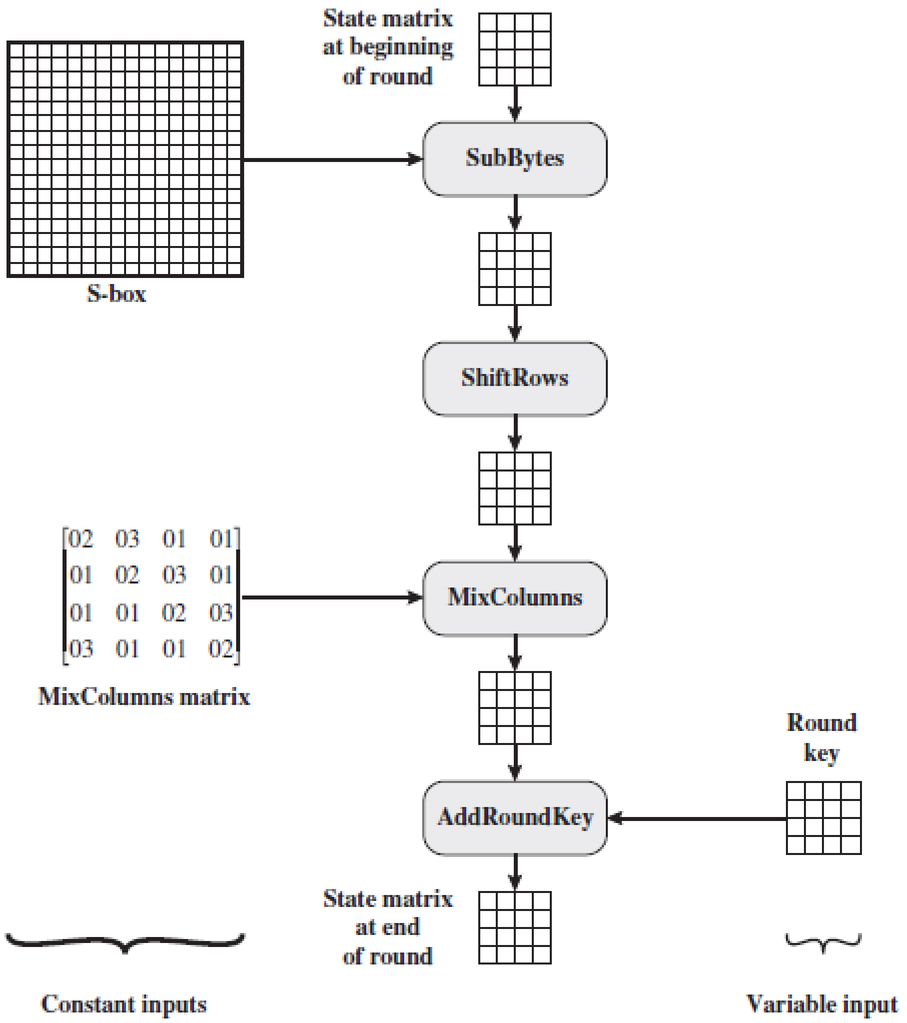


Figure 5.8 Inputs for Single AES Round

## AES Key Expansion:-

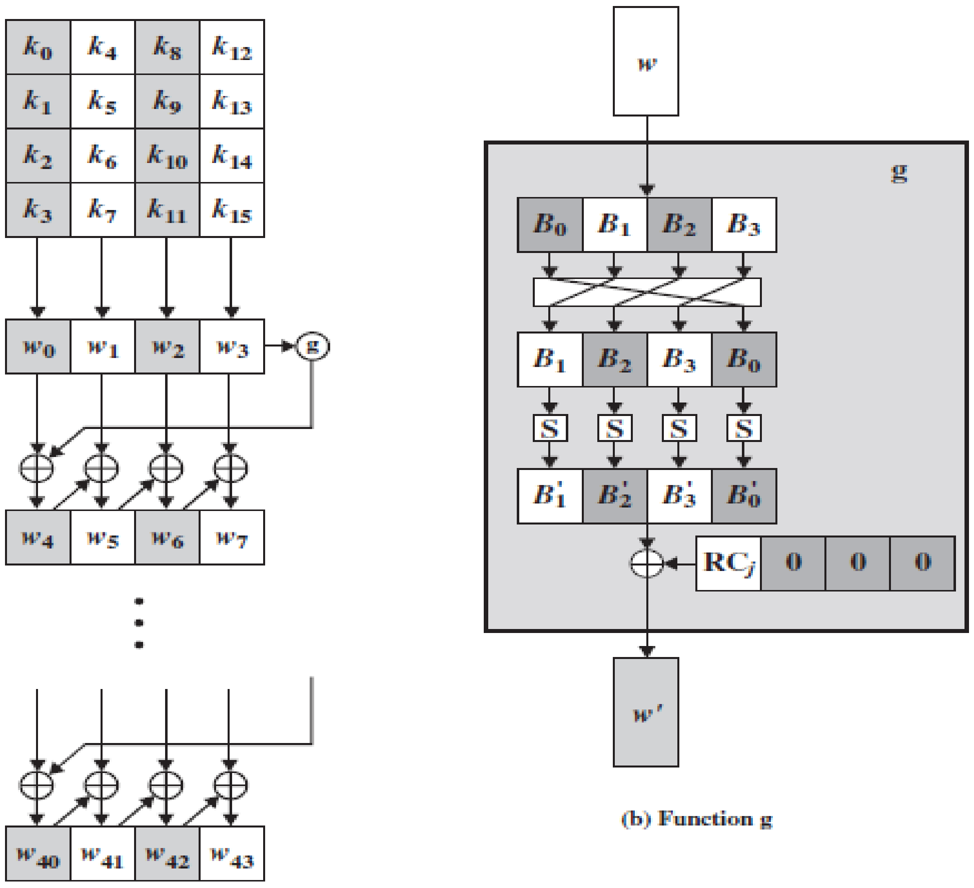
### Key Expansion Algorithm:

- ✓ The AES key expansion algorithm takes as input a four-word (16-byte) key and produces a linear array of 44 words (176 bytes).
- ✓ This is sufficient to provide a four word round key for the initial AddRound Key stage and each of the 10 rounds of the cipher.
- ✓ The pseudocode on the next page describes the expansion.
- ✓ 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 position 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.

- ✓ Figure 5.9illustrates the generation of the expanded key, using the symbol g to represent that complex function. The function g consists of the following subfunctions.

```
KeyExpansion (byte key[16], word w[44])
{
    word temp
    for (i = 0; i < 4; i++)   w[i] = (key[4*i], key[4*i+1],
                                      key[4*i+2],
                                      key[4*i+3]);

     for (i = 4; i < 44; i++)
     {
      temp = w[i - 1];
      if (i mod 4 = 0)   temp = SubWord (RotWord (temp))
                                ⊕ Rcon[i/4];
      w[i] = w[i-4]  ⊕  temp
     }
}
```



$k_0$ $k_4$ $k_8$ $k_{12}$
$k_1$ $k_5$ $k_9$ $k_{13}$
$k_2$ $k_6$ $k_{10}$ $k_{14}$
$k_3$ $k_7$ $k_{11}$ $k_{15}$
$w_0$ $w_1$ $w_2$ $w_3$
g
$w_4$ $w_5$ $w_6$ $w_7$
$w_{40}$ $w_{41}$ $w_{42}$ $w_{43}$
(a) Overall algorithm
w
g
$B_0$ $B_1$ $B_2$ $B_3$
$B_1$ $B_2$ $B_3$ $B_0$
S S S S
$B_1'$ $B_2'$ $B_3'$ $B_0'$
$RC_j$ 0 0 0
w'
(b) Function g

Figure 5.9 AES Key Expansion

1. RotWord performs a one-byte circular left shift on a word. This means that an input word [$B_0$, $B_1$, $B_2$, $B_3$] is transformed into [$B_1$, $B_2$, $B_3$, $B_0$].
2. SubWord performs a byte substitution on each byte of its input word, using the S-box (Table 5.2a).
3. The result of steps 1 and 2 is XORed with a round constant, Rcon[j].

The round constant is a word in which the three rightmost bytes are always 0. Thus, the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word. The round constant is different for each round and is defined as Rcon[j] = (RC[j], 0, 0, 0), with RC[1] = 1, RC[j] = 2 · RC[j−1] and with multiplication defined over the field GF($2^8$). The values of RC[j] in hexadecimal are

| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| RC[j] | 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1B | 36 |

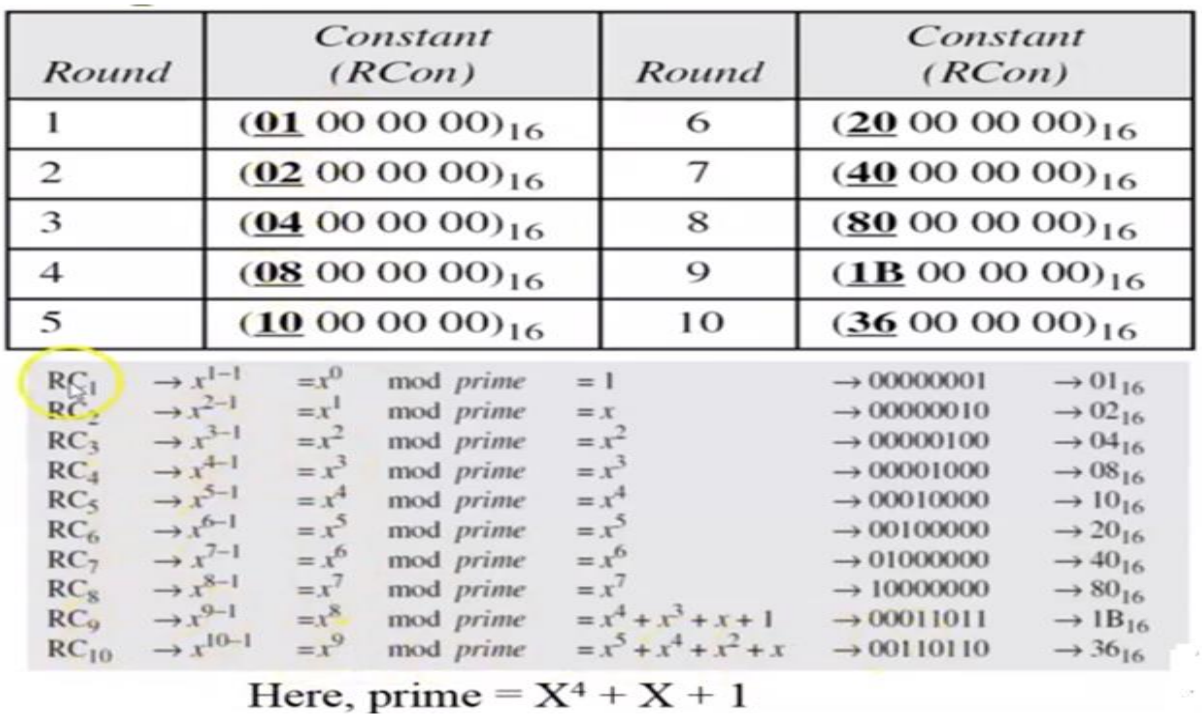
For example, suppose that the round key for round 8 is

EA D2 73 21 B5 8D BA D2 31 2B F5 60 7F 8D 29 2F

Then the first 4 bytes (first column) of the round key for round 9 are calculated as follows:

| i (decimal) | temp | After RotWord | After SubWord | Rcon (9) | After XOR with Rcon | w[i−4] | w[i] = temp ⊕ w[i−4] |
|---|---|---|---|---|---|---|---|
| 36 | 7F8D292F | 8D292F7F | 5DA515D2 | 1B000000 | 46A515D2 | EAD27321 | AC7766F3 |

**For students reference**



| Round | Constant (RCon) | Round | Constant (RCon) |
|---|---|---|---|
| 1 | $(\underline{\mathbf{01}}\ 00\ 00\ 00)_{16}$ | 6 | $(\underline{\mathbf{20}}\ 00\ 00\ 00)_{16}$ |
| 2 | $(\underline{\mathbf{02}}\ 00\ 00\ 00)_{16}$ | 7 | $(\underline{\mathbf{40}}\ 00\ 00\ 00)_{16}$ |
| 3 | $(\underline{\mathbf{04}}\ 00\ 00\ 00)_{16}$ | 8 | $(\underline{\mathbf{80}}\ 00\ 00\ 00)_{16}$ |
| 4 | $(\underline{\mathbf{08}}\ 00\ 00\ 00)_{16}$ | 9 | $(\underline{\mathbf{1B}}\ 00\ 00\ 00)_{16}$ |
| 5 | $(\underline{\mathbf{10}}\ 00\ 00\ 00)_{16}$ | 10 | $(\underline{\mathbf{36}}\ 00\ 00\ 00)_{16}$ |

| | | | | | | |
|---|---|---|---|---|---|---|
| $RC_1$ | $\rightarrow x^{1-1}$ | $= x^0$ | mod *prime* | $= 1$ | $\rightarrow 00000001$ | $\rightarrow 01_{16}$ |
| $RC_2$ | $\rightarrow x^{2-1}$ | $= x^1$ | mod *prime* | $= x$ | $\rightarrow 00000010$ | $\rightarrow 02_{16}$ |
| $RC_3$ | $\rightarrow x^{3-1}$ | $= x^2$ | mod *prime* | $= x^2$ | $\rightarrow 00000100$ | $\rightarrow 04_{16}$ |
| $RC_4$ | $\rightarrow x^{4-1}$ | $= x^3$ | mod *prime* | $= x^3$ | $\rightarrow 00001000$ | $\rightarrow 08_{16}$ |
| $RC_5$ | $\rightarrow x^{5-1}$ | $= x^4$ | mod *prime* | $= x^4$ | $\rightarrow 00010000$ | $\rightarrow 10_{16}$ |
| $RC_6$ | $\rightarrow x^{6-1}$ | $= x^5$ | mod *prime* | $= x^5$ | $\rightarrow 00100000$ | $\rightarrow 20_{16}$ |
| $RC_7$ | $\rightarrow x^{7-1}$ | $= x^6$ | mod *prime* | $= x^6$ | $\rightarrow 01000000$ | $\rightarrow 40_{16}$ |
| $RC_8$ | $\rightarrow x^{8-1}$ | $= x^7$ | mod *prime* | $= x^7$ | $\rightarrow 10000000$ | $\rightarrow 80_{16}$ |
| $RC_9$ | $\rightarrow x^{9-1}$ | $= x^8$ | mod *prime* | $= x^4 + x^3 + x + 1$ | $\rightarrow 00011011$ | $\rightarrow 1B_{16}$ |
| $RC_{10}$ | $\rightarrow x^{10-1}$ | $= x^9$ | mod *prime* | $= x^5 + x^4 + x^2 + x$ | $\rightarrow 00110110$ | $\rightarrow 36_{16}$ |

Here, prime = $X^4 + X + 1$

## Pseudo-Random-Sequence Generators and Stream Ciphers

Pseudo-Random-Sequence Generators and Stream Ciphers: Linear Congruential Generators, Linear Feedback Shift Registers, Design and analysis of stream ciphers, Stream ciphers using LFSRs (Text 2: Chapter 16: Section 1, 2, 3, 4)

## Linear Congruential Generators:

**Linear congruential generators** are pseudo-random-sequence generators of the form,

$$\mathbf{Xn = (aXn\text{-}1 + b) \bmod m}$$

in which Xn is the nth number of the sequence, and Xn-1 is the previous number of the sequence. The variables a, b, and m are constants: **a** is the **multiplier, b** is the **increment, and m is the modulus. The key, or seed, is the value of $X_0$.**This generator has a period no greater than *m*. If *a*, *b*, and *m* are properly chosen, then the generator will be a **maximal period generator** (sometimes called maximal length) and have period of *m*. (For example, *b* should be relatively prime to *m*.)

**The advantage of linear congruential generators is that they are fast, requiring few operations per bit.**

- Unfortunately, linear congruential generators cannot be used for cryptography; they are predictable.
- Linear congruential generators were first broken by Jim Reeds and then by Joan Boyar. She also broke quadratic generators:

  $$X_n = (aX_{n-1}^2 + bX_{n-1} + c) \bmod m$$

  and cubic generators:

  $$X_n = (aX_{n-1}^3 + bX_{n-1}^2 + cX_{n-1} + d) \bmod m$$

  Linear congruential generators remain useful for noncryptographic applications, however, such as simulations. They are efficient and show good statistical behaviour with respect to most reasonable empirical tests.

**Example 1 LCG (5, 1, 16, 1)**

**Let us consider a simple example with a= 5, c=1, m=16, and X0 =1.**

**The sequence of pseudorandom integers generated by this algorithm is:**
1,6,15,12,13,2,11,8,9,14,7,4,5,10,3,0,1,6,15,12,13,2,11,8,9,14,

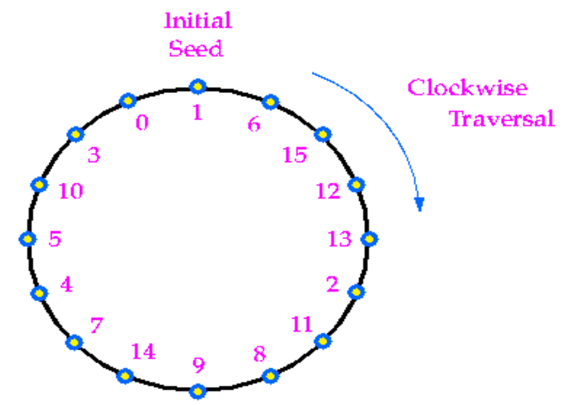


Figure 3: Random Number Cycle for Example 1 - LCG (5, 1, 16, 1).

**a= 5, b=1, m=16, and X0 =1**

*Xn = (aXn-1 + b) mod m*

- **x0=1**
- **x1=(5*1+1)mod16=6**
- **x2=(5*6+1)mod16=15**
- **x3=(5*15+1)mod16=12**

- **We observe :**
  - **The period (the number of integers before the sequence repeats) P is 16 - exactly equal to the modulus, m. Thus, for m=16 , this sequence is of long period (the longest possible), and uniform (it completely fills the space of integers from 0-15).**
  - **Sequence exhibits throughout its period the pattern of alternating odd and even integers.**
  - **It is readily apparent that the sequence is serially correlated. Due to this lack of randomness, the values should not be used as random digits.**
  - **The real numbers generated from the integer sequence are generally sufficiently random in the higher order (most significant) bits to be used in many application codes.**

## Linear Feedback Shift Registers:

- Shift register sequences are used in both cryptography and coding theory.
- Stream ciphers based on shift registers have been the workhorse of military cryptography since the beginnings of electronics.
- A **feedback shift register** is made up of two parts: a shift register and a **feedback function** (see Figure 16.1).
- The shift register is a sequence of bits. (The **length** of a shift register is figured in bits; if it is *n* bits long, it is called an *n*-bit shift register.)
- Each time a bit is needed, all of the bits in the shift register are shifted 1 bit to the right.
- The new left-most bit is computed as a function of the other bits in the register.
- The output of the shift register is 1 bit, often the least significant bit.
- The **period** of a shift register is the length of the output sequence before it starts repeating.
- Cryptographers have liked stream ciphers made up of shift registers: They are easily implemented in digital hardware.

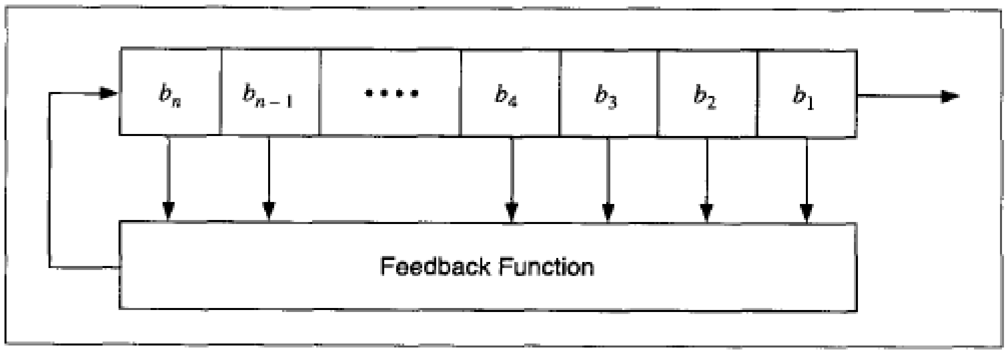


*Figure 16.1 Feedback shift register.*

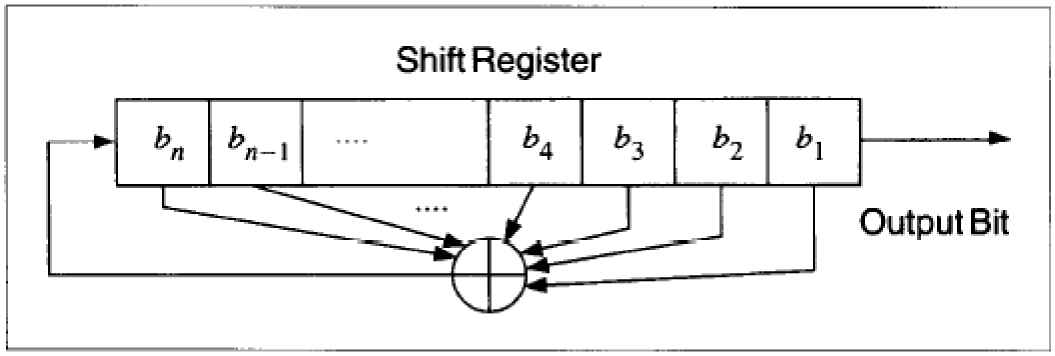


*Figure 16.2 Linear feedback shift register.*

- The simplest kind of feedback shift register is a **linear feedback shift register**, or LFSR (see Figure 16.2).
- The feedback function is simply the XOR of certain bits in the register; the list of these bits is called a **tap sequence**.
- Sometimes this is called a **Fibonacci configuration**. Because of the simple feedback sequence, a large body of mathematical theory can be applied to analyzing LFSRs.

- Cryptographers like to analyze sequences to convince themselves that they are random enough to be secure.
- LFSRs are the most common type of shift registers used in cryptography. Figure 16.3 is a 4-bit LFSR tapped at the first and fourth bit.
- If it is initialized with the value 1111, it produces the following sequence of internal states before repeating:

```
1 1 1 1
0 1 1 1
1 0 1 1
0 1 0 1
1 0 1 0
1 1 0 1
0 1 1 0
0 0 1 1
1 0 0 1
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0
1 1 0 0
1 1 1 0
```

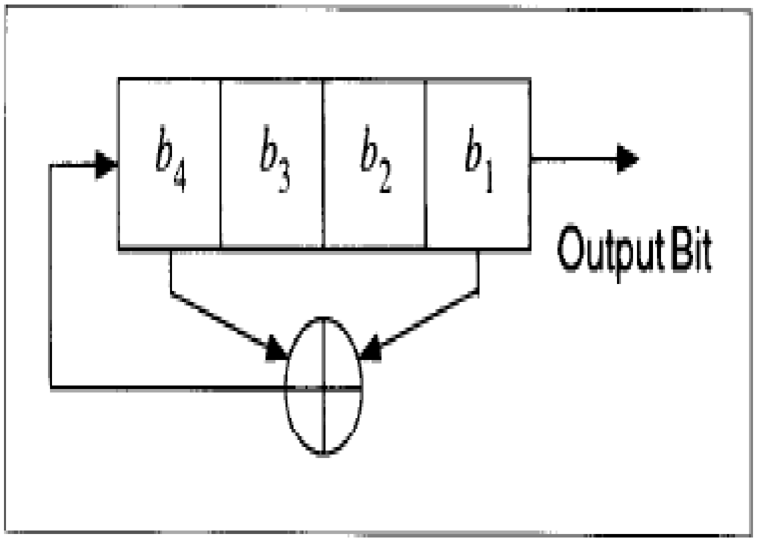


Figure 16.3 4-bit LFSR.

The output sequence is the string of least significant bits:

1 1 1 1 0 1 0 1 1 0 0 1 0 0 0....

- An $n$-bit LFSR can be in one of $2^n - 1$ internal states.
- This means that it can, in theory, generate a $2^n - 1$-bit-long pseudo-random sequence before repeating. (It's $2^n - 1$ and not 2n because a shift register filled with zeros will cause the LFSR to output a never ending stream of zeros—this is not particularly useful.)
- Only LFSRs with certain tap sequences will cycle through all $2^n - 1$ internal states; these are the maximal-period LFSRs.
- The resulting output sequence is called an **m-sequence**.
- In order for a particular LFSR to be a maximal-period LFSR, the polynomial formed from a tap sequence plus the constant 1 must be a primitive polynomial mod 2.
- The **degree** of the polynomial is the length of the shift register. A primitive polynomial of degree $n$ is an irreducible polynomial that divides $x^{2n-1} + 1$, but not $x^d + 1$ for any $d$ that divides $2^n – 1$.

The listing (32, 7, 5, 3, 2, 1, 0) means that the following polynomial is primitive modulo 2:

$$x^{32} + x^7 + x^5 + x^3 + x^2 + x + 1$$

- It's easy to turn this into a maximal-period LFSR.
- The first number is the length of the LFSR.
- The last number is always 0 and can be ignored.
- All the numbers, except the 0, specify the tap sequence, counting from the left of the shift register.
- That is, low degree terms in the polynomial correspond to taps near the left-hand side of the register.
- To continue the example, the listing (32, 7, 5, 3, 2, 1, 0) means that if you take a 32-bit shift register and generate the new bit by XORing the thirty-second, seventh, fifth, third, second, and first bits together (see Figure 16.4), the resultant LFSR will be maximal length; it will cycle through $2^{32}$ - 1 values before repeating.

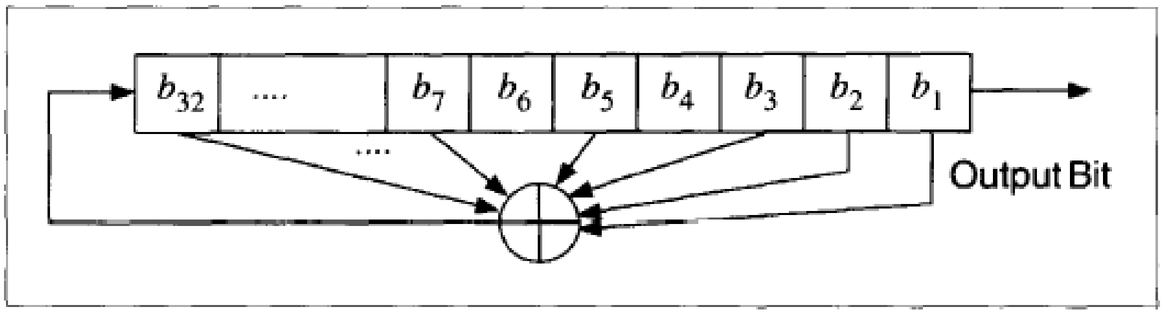


*Figure 16.4 32-bit long maximal-length LFSR.*

The C code for this LFSR looks like:

```
int LFSR () {
     static unsigned long ShiftRegister = 1;
     /* Anything but 0. */
     ShiftRegister = ((((ShiftRegister >> 31)
                 ^ (ShiftRegister >> 6)
                 ^ (ShiftRegister >> 4)
                 ^ (ShiftRegister >> 2)
                 ^ (ShiftRegister >> 1)
                 ^ ShiftRegister))
                 & 0x00000001)
                 << 31)
                 | (ShiftRegister >> 1) ;
     return ShiftRegister & 0x00000001;
}
```

- The code is a little more complicated when the shift register is longer than the computer's word size. Note that all of these listings have an odd number of coefficients.
- For example, if ($a$, $b$, 0) is primitive, then ($a$, $a$ - $b$, 0) is also primitive. If ($a$, $b$, $c$, $d$, 0) is primitive, then ($a$, $a$ - $d$, $a$ - $c$, $a$ - $b$, 0) is also primitive. Mathematically:

$$\text{if } x^a + x^b + 1 \text{ is primitive, so is } x^a + x^{a-b} + 1$$

$$\text{if } x^a + x^b + x^c + x^d + 1 \text{ is primitive, so is } x^a + x^{a-d} + x^{a-c} + x^{a-b} + 1$$

- Primitive trinomials are fastest in software, because only two bits of the shift register have to be XORed to generate each new bit.
- LFSRs are competent pseudo-random-sequence generators all by themselves, but they have some annoying nonrandom properties.
- Sequential bits are linear, which makes them useless for encryption.
- For an LFSR of length $n$, the internal state is the next $n$ output bits of the generator.
- Even if the feedback scheme is unknown, it can be determined from only 2$n$ output bits of the generator, by using the highly efficient Berlekamp-Massey algorithm.

**LFSRs in Software:**

- ➢ LFSRs are slow in software, but they're faster in assembly language than in C.
- ➢ One solution is to run 16 LFSRs (or 32, depending on your computer's word size) in parallel.
- ➢ This scheme uses an array of words that is the length of the LFSR, with each bit position in the words representing a different LFSR.
- ➢ Assuming all the feedback polynomials are the same, this can run quickly. In general, the best way to update shift registers is to multiply the current state by suitable binary matrices.
- ➢ It is also possible to modify the LFSR's feedback scheme.

- The resultant generator is no better cryptographically, but it still has a maximal period and is easy to implement in software.
- Instead of using the bits in the tap sequence to generate the new left-most bit, each bit in the tap sequence is XORed with the output of the generator and replaced; then the output of the generator becomes the new left-most bit (see Figure 16.5).
- This is sometimes called a **Galois configuration**.

In C, this looks like:

```
#define mask 0x80000057

static unsigned long ShiftRegister=1;
void seed_LFSR (unsigned long seed)
{
     if (seed == 0) /* avoid calamity */
          seed = 1;
     ShiftRegister = seed;
}

int modified_LFSR (void)
{
     if (ShiftRegister & 0x00000001) {
          ShiftRegister = ((ShiftRegister ^ mask >> 1) |
     0x8000000;
          return 1;
     } else {
          ShiftRegister >>= 1;
          return 0;
     }
}
```

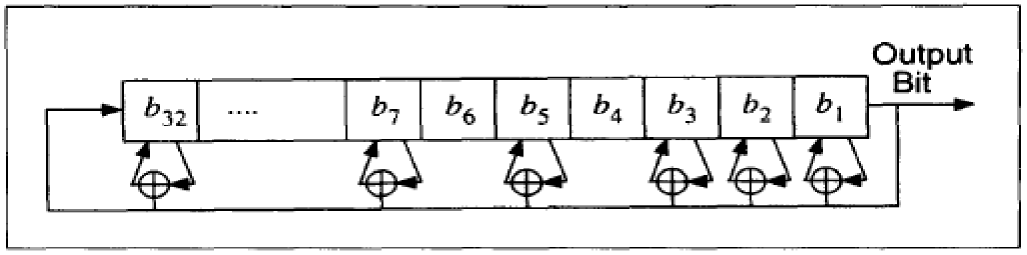


*Figure 16.5 Galois LFSR.*

- The savings here is that all the XORs can be done as a single operation.
- This can also be parallelized, and the different feedback polynomials can be different.
- The Galois configuration can also be faster in hardware, especially in custom VLSI implementations.

## Design and Analysis of Stream Ciphers:

- Most practical stream-cipher designs center around LFSRs.
- In the early days of electronics, they were very easy to build.
- A shift register is nothing more than an array of bit memories and the feedback sequence is just a series of XOR gates.
- Even in VLSI circuitry, a LFSR-based stream cipher can give you a lot of security with only a few logic gates.
- The problem with LFSRs is that they are very inefficient in software. Any stream cipher outputs a bit at a time; you have to iterate the algorithm 64 times to encrypt.
- A majority of military encryptions systems in use today are based on LFSRs.
- In fact, most Cray computers (Cray 1, Cray X-MP, Cray Y-MP) have a rather curious instruction generally known as "population count."
- It counts the 1 bits in a register and can be used both to efficiently calculate the Hamming distance between two binary words and to implement a vectorized version of a LFSR.

**[NOTE: (only for your reference)**
The Cray-1 was a supercomputer, these play an important role in the field of computational science, and are used for a wide range of computationally **intensive** tasks in various fields, including quantum mechanics, **weather forecasting**, climate research, oil and gas exploration,**molecular modeling** (computing the structures and properties).**]**

### Linear Complexity:

- Analyzing stream ciphers is often easier than analyzing block ciphers.
- For example, one important metric used to analyze LFSR-based generators is **linear complexity**, or linear span.
- This is defined as the length, *n*, of the shortest LFSR that can mimic the generator output. Any sequence generated by a finite-state machine over a finite field has a finite linear complexity.
- Linear complexity is important because a simple algorithm, called the **Berlekamp-Massey** algorithm, can generate this LFSR after examining only 2*n* bits of the key stream.
- A **linear complexity profile**, which measures the linear complexity of the sequence as it gets longer and longer.
- A high linear complexity does not necessarily indicate a secure generator, but a low linear complexity indicates an insecure one.

### Correlation Immunity:

- Cryptographers try to get a high linear complexity by combining the output of several output sequences in some nonlinear manner.
- The danger here is that one or more of the internal output sequences can be correlated with the combined key stream and attacked using linear algebra.
- Often this is called a **correlation attack** or a divide-and-conquer attack.
- The basic idea behind a correlation attack is to identify some correlation between the output of the generator and the output of one of its internal pieces.
- Then, by observing the output sequence, you can obtain information about that internal output.

- Using that information and other correlations, collect information about the other internal outputs until the entire generator is broken.

**[NOTE:**
**In cryptography, correlation attacks are a class of known plaintext attacks for breaking stream ciphers whose key stream is generated by combining the output of several linear feedback shift registers using a Boolean function.]**

**Other Attacks:**
There are other general attacks against keystream generators.

- The **linear consistency test** attempts to identify some subset of the encryption key using matrix techniques.
- **meet-in-the-middle consistency attack:** It targets block cipher cryptographic functions. Because the attacker tries to break the two-part encryption method from both sides simultaneously, a successful effort enables him to meet in the middle of the block cipher.
- **linear syndrome algorithm** relies on being able to write a fragment of the output sequence as a linear equation.
- **best affine approximation attack and derived sequence attack.**

## Stream Ciphers Using LFSRs:

- The basic approach to designing a key stream generator using LFSRs is simple.
- First take one or more LFSRs, generally of different lengths and with different feedback polynomials. (If the lengths are all relatively prime and the feedback polynomials are all primitive, the whole generator is maximal length.)
- The key is the initial state of the LFSRs.
- Every time you want a bit, shift the LFSRs once (this is sometimes called **clocking**).
- The output bit is a function, preferably a nonlinear function, of some of the bits of the LFSRs.
- This function is called the **combining function**, and the whole generator is called a **combination generator**. (If the output bit is a function of a single LFSR, the generator is called a **filter generator**.)

➢ Complications have been added. Some generators have LFSRs clocked at different rates; sometimes the clocking of one generator depends on the output of another.
➢ These are all electronic versions of pre-WWII (World-War II) cipher machine ideas, and are called **clock-controlled generators.**
➢ Clock control can be feedforward, where the output of one LFSR controls the clocking of another, or feedback, where the output of one LFSR controls its own clocking.
➢ These are susceptible to embedding and probabilistic correlation attacks.

Since LFSR-based ciphers are generally implemented in hardware, electronics logic symbols will be used in the figures. In the text, • is XOR, ^ is AND, ¦ is OR, and ¬ is NOT.

**Geffe Generator:**

- This key stream generator uses three LFSRs, combined in a nonlinear manner as shown in figure 16.6.
- Two of the LFSRs are inputs into a multiplexer, and the third LFSR controls the output of the multiplexer.
- If *a*1, *a*2, and *a*3 are the outputs of the three LFSRs, the output of the Geffe generator can be described by:

$$b = (a_1 \wedge a_2) \bullet ((\neg a_1) \wedge a_3)$$

If the LFSRs have lengths *n*1, *n*2, and *n*3, respectively, then the linear complexity of the generator is,

$$(n_1 + 1)n_2 + n_1 n_3$$

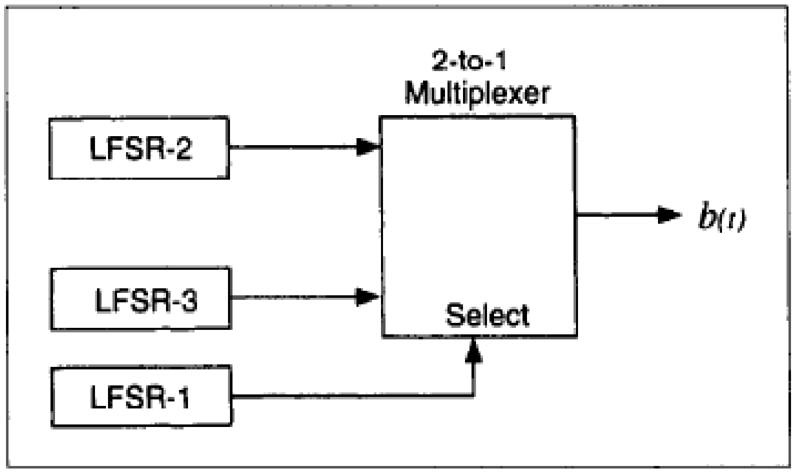


*Figure 16.6 Geffe generator.*

- The period of the generator is the least common multiple of the periods of the three generators.
- Assuming the degrees of the three primitive feedback polynomials are relatively prime, the period of this generator is the product of the periods of the three LFSRs.
- This generator is cryptographically weak and falls to a correlation attack**.**
- The output of the generator equals the output of LFSR-2 75 percent of the time.
- If the feedback taps are known, you can guess the initial value for LFSR-2 and generate the output sequence of that register.
- Then you can count the number of times the output of the LFSR-2 agrees with the output of the generator.
- If you guessed wrong, the two sequences will agree about 50 percent of the time; if you guessed right, the two sequences will agree about 75 percent of the time.
- Similarly, the output of the generator equals the output of LFSR-3 about 75percent of the time.
- With those correlations, the key stream generator can be easily cracked. For example, if the primitive polynomials only have three terms each, and the largest LFSR is of length *n*,

it only takes a segment of the output sequence 37*n*-bits long to reconstruct the internal states of all three LFSRs.

**Generalized Geffe Generator:**

This scheme chooses between k LFSRs, as long as k is a power of 2. There are k + 1 LFSRs total (see Figure 16.7). LFSR-1 must be clocked $\log_2 k$ times faster than the other k LFSRs. This generator will affect by possibility of Correlation attack.

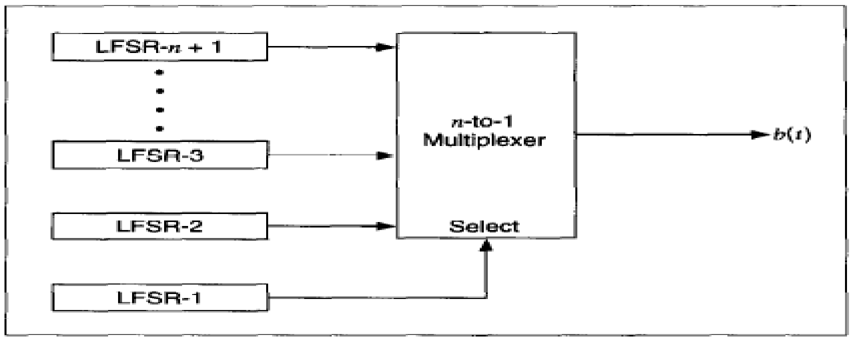


*Figure 16.7 Generalized Geffe generator.*

**Jennings Generator:**

- This scheme uses a multiplexer to combine two LFSRs.
- The multiplexer, controlled by LFSR-1, selects 1 bit of LFSR-2 for each output bit.
- There is also a function that maps the output of LFSR-2 to the input of the multiplexer (see Figure 16.8).
- The key is the initial state of the two LFSRs and the mapping function. Although this generator has great statistical properties, it fell to Ross Anderson’s meet-in-the-middle consistency attack and the linear consistency attack.

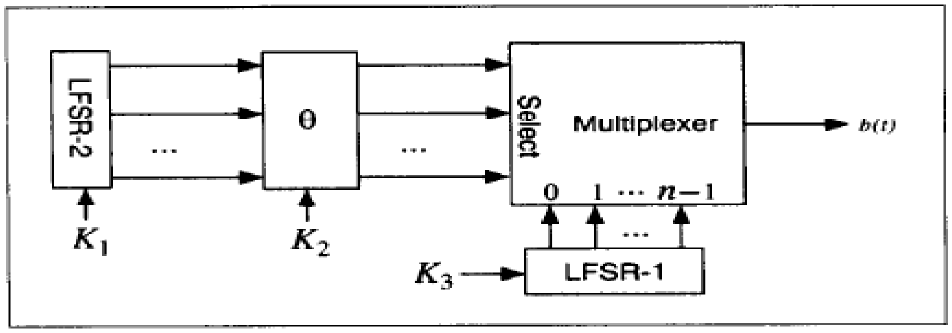


*Figure 16.8 Jennings generator.*

**Beth-Piper Stop-and-Go Generator:**

- This generator, shown in Figure 16.9, uses the output of one LFSR to control the clock of another LFSR.
- The clock input of LFSR-2 is controlled by the output of LFSR-1, so that LFSR-2 can change its state at time *t* only if the output of LFSR-1 was 1 at time *t* - 1.
- However, it falls to a correlation attack.

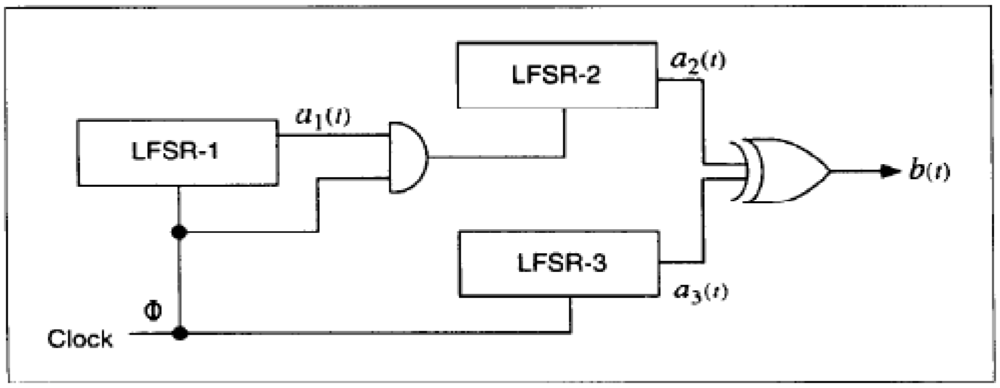


*Figure 16.9 Beth-Piper stop-and-go generator.*

**Alternating Stop-and-Go Generator:**

- This generator uses three LFSRs of different length. LFSR-2 is clocked when the output of LFSR-1 is 1; LFSR-3 is clocked when the output of LFSR-1 is 0.
- The output of the generator is the XOR of LFSR-2 and LFSR-3 (see Figure 16.10).
- This generator has a long period and large linear complexity.
- It is observed a correlation attack against LFSR-1, but it does not substantially weaken the generator.

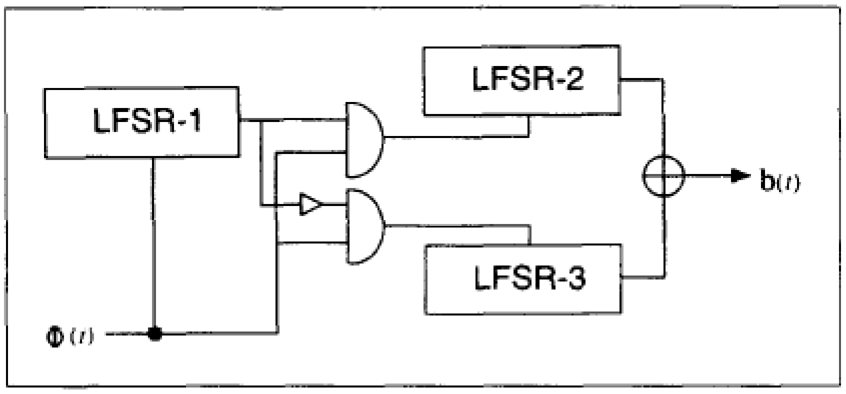


*Figure 16.10 Alternating stop-and-go generator.*

**Bilateral Stop-and-Go Generator:**

- This generator uses two LFSRs, both of length n (see Figure 16.11).
- The output of the generator is the XOR of the outputs of each LFSR.
- If the output of LFSR-2 at time t – 1 is 0 and the output at time t – 2 is 1, then LFSR-2 does not clock at time t. If the output of LFSR-1 at time t – 1 is 0 and the output at t – 2 is 1, and if LFSR-1 clocked at time t, then LFSR-2 does not clock at time t. The linear complexity of this system is roughly equal to the period.

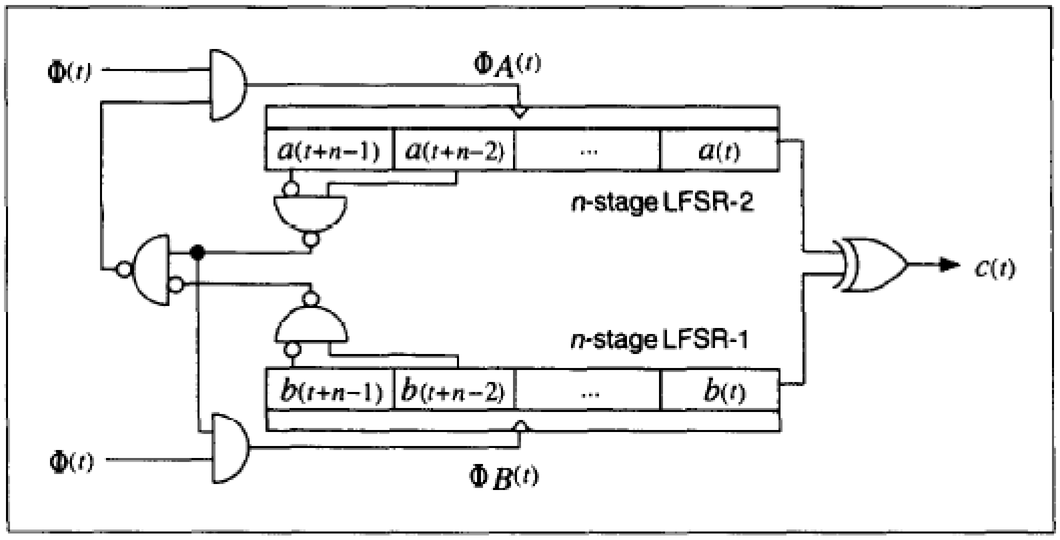


*Figure 16.11 Bilateral stop-and-go generator.*

**Threshold Generator:**

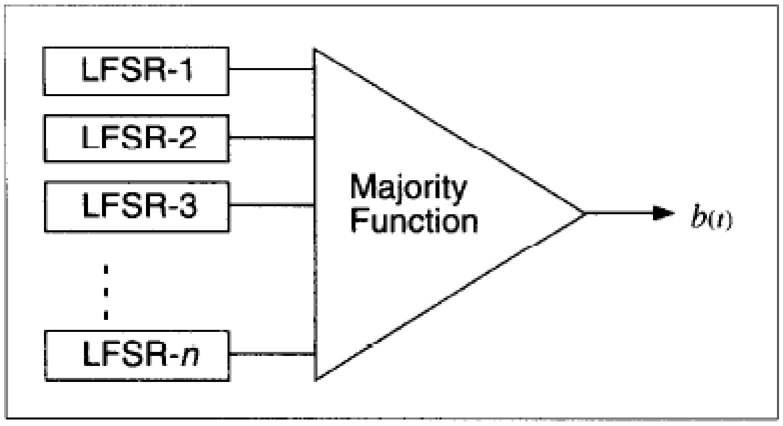


*Figure 16.12 Threshold generator.*

- This generator tries to get around the security problems of the previous generators by using a variable number of LFSRs. The theory is that if you use a lot of LFSRs, it's harder to break the cipher.
- This generator is illustrated in Figure 16.12. Take the output of a large number of LFSRs (use an odd number of them).
- Make sure the lengths of all the LFSRs are relatively prime and all the feedback polynomials are primitive: maximize the period.
- If more than half the output bits are 1, then the output of the generator is 1. If more than half the output bits are 0, then the output of the generator is 0.

With three LFSRs, the output generator can be written as:

$$b = (a_1 \wedge a_2) \bullet (a_1 \wedge a_3) \bullet (a_2 \wedge a_3)$$

This is very similar to the Geffe generator, except that it has a larger linear complexity of,

$$n_1n_2 + n_1n_3 + n_2n_3$$

where*n*1, *n*2, and *n*3 are the lengths of the first, second, and third LFSRs.
Each output bit of the generator yields some information about the state of the LFSRs—0.189 bit to be exact—and the whole thing falls to a correlation attack.

**Self-Decimated Generators:**
Self-decimated generators are generators that control their own clock. Two have been proposed, one by Rainer Rueppel (see Figure 16.13) and another by Bill Chambers and Dieter Gollmann (see Figure 16.14).

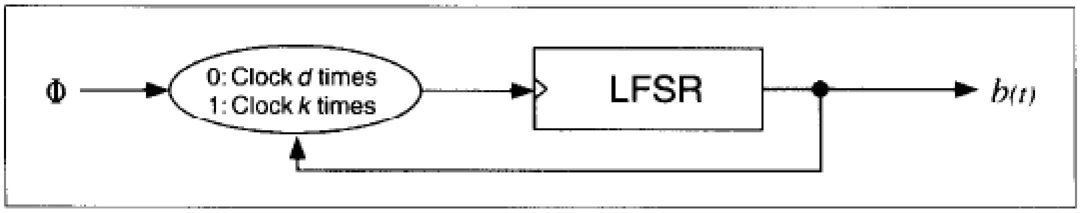


*Figure 16.13 Rueppel's self-decimated generator.*

In Rueppel's generator, when the output of the LFSR is 0, the LFSR is clocked d times. When the output of the LFSR is 1, the LFSR is clocked k times.

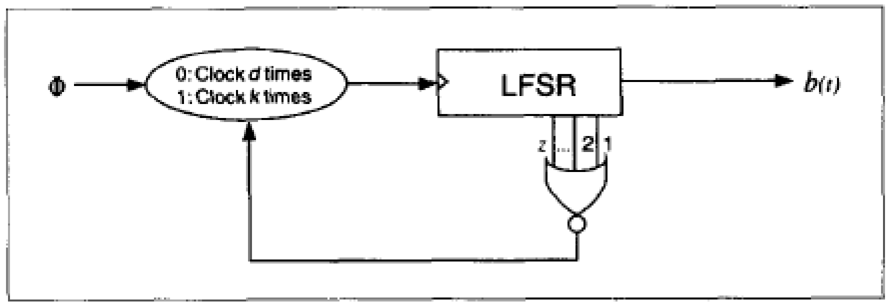


*Figure 16.14 Chambers's and Gollmann's self-decimated generator.*

Chambers's and Gollmann's generator is more complicated, but the idea is the same. Unfortunately, both generators are insecure, although some modifications have been proposed that may correct the problems.

**Multispeed Inner-Product Generator:**

- This generator, by Massey and Rueppel, uses two LFSRs clocked at two different speeds (see Figure 16.15).
- LFSR-2 is clocked *d* times as fast as LFSR-1.
- The individual bits of the two LFSRs are ANDed together and then XORed with each other to produce the final output bit of the generator.

- Although this generator has high linear complexity and it possesses excellent statistical properties, it still falls to a linear consistency attack.
- If $n1$ is the length of LFSR-1, $n2$ is the length of the LFSR-2, and $d$ is the speed multiple between the two, then the internal state of the generator can be recovered from an output sequence of length. $n1 + n2 + log_2 d$.

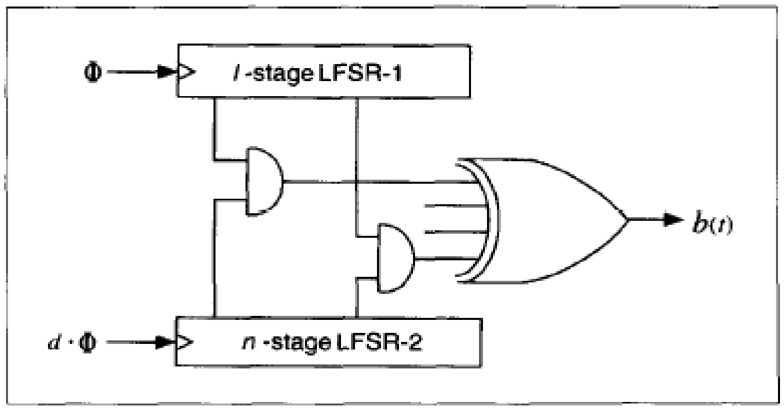


*Figure 16.15 Multispeed inner-product generator.*

**Summation Generator:**

This generator adds the output of two LFSRs (with carry) worked by Rainer Rueppel. This operation is highly nonlinear. Through the late 1980s, this generator was the security front-runner, but it fell to a correlation attack and it has been shown that this is an example of a feedback with carry shift register and can be broken.

**DNRSG:**

- This stands for "dynamic random-sequence generator".
- The idea is to have two different filter generators—threshold, summation, or anyone—fed by a single set of LFSRs and controlled by another LFSR.
- First clock all the LFSRs. If the output of LFSR-0 is 1, then compute the output of the first filter generator. If the output of LFSR-0 is 0, then compute the output of the second filter generator. The final output is the first output XOR the second.

**Gollmann Cascade:**

- The Gollmann cascade (see Figure 16.16), is a strengthened version of a stop-and-go generator. It consists of a series of LFSRs, with the clock of each controlled by the previous LFSR.
- If the output of LFSR-1 is 1 at time $t$ - 1, then LFSR-2 clocks.
- If the output of LFSR-2 is 1 at time $t$ - 1, then LFSR-3 clocks, and so on.
- The output of the final LFSR is the output of the generator.
- If all the LFSRs have the same length, $n$, the linear complexity of a system with $k$ LFSRs is, $n(2^n - 1)^{k-1}$

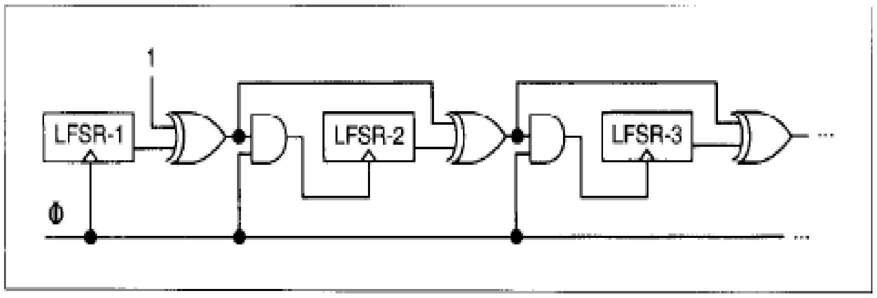


*Figure 16.16 Gollmann cascade.*

- Cascades are a cool idea: They are conceptually very simple and they can be used to generate sequences with huge periods, huge linear complexities, and good statistical properties.
- They are vulnerable to an attack called **lock-in.**
- This is a technique by which a cryptanalyst reconstructs the input to the last shift register in the cascade, then proceeds to break the cascade register by register.
- This is a serious problem in some situations and weakens the effective key length of the algorithm, but precautions can be taken to minimize the attack.

**Shrinking Generator:**

- The shrinking generator uses a different form of clock control than the previous generators.
- Take two LFSRs**: LFSR-1 and LFSR-2**. Clock both of them. If the output of LFSR-1 is 1, then the output of the generator is LFSR-2.
- If the output of LFSR-1 is 0, discard the two bits, clock both LFSRs, and try again.

This idea is simple, reasonably efficient, and looks secure. If the feedback polynomials are sparse (fewer coefficients), the generator is vulnerable, but no other problems have been found. One implementation problem is that the output rate is not regular; if LFSR-1 has a long string of zeros then the generator outputs nothing. Use buffering to solve this problem.

**Self-Shrinking Generator:**

- The self-shrinking generator is a variant of the shrinking generator. Instead of using two LFSRs, use pairs of bits from a single LFSR.
- Clock a LFSR twice. If the first bit in the pair is 1, the output of the generator is the second bit. If the first bit is 0, discard both bits and try again.
- While the self-shrinking generator requires about half the memory space as the shrinking generator, it is also half the speed.
- Self-shrinking generator is secure, it still has some unexplained behaviour and unknown properties.