

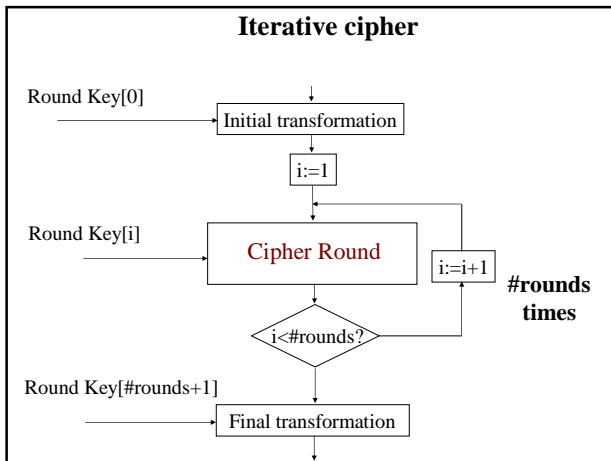
ECE 746: Lecture 4

**AES implementations
in software and hardware**

AES in Software

**Reference software
implementation**

Iterative cipher



Round function - encryption

```

KeyAddition(a,rk[0],BC);

/* ROUNDS-1 ordinary rounds */

for(r = 1; r < ROUNDS; r++) {
  Substitution(a,S,BC);
  ShiftRow(a,0,BC);
  MixColumn(a,BC);
  KeyAddition(a,rk[r],BC);
}

/* Last round is special: there is no MixColumn */

Substitution(a,S,BC);
ShiftRow(a,0,BC);
KeyAddition(a,rk[ROUNDS],BC);
  
```

Data structures and parameters

```

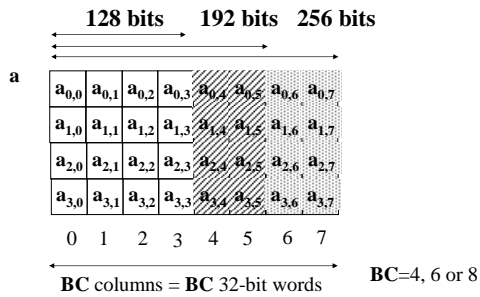
int rijndaelEncryptRound (word8 a[4][MAXBC],
                          int keyBits,
                          int blockBits,
                          word8 rk[MAXROUNDS+1][4][MAXBC], int rounds)

{
  int r, BC, ROUNDS;

  switch (blockBits) {
    case 128: BC = 4; break;
    case 192: BC = 6; break;
    case 256: BC = 8; break;
    default : return (-2);
  }
  switch (keyBits >= blockBits ? keyBits : blockBits) {
    case 128: ROUNDS = 10; break;
    case 192: ROUNDS = 12; break;
    case 256: ROUNDS = 14; break;
    default : return (-3); /* this cannot happen */
  }
}
  
```

Variable block size

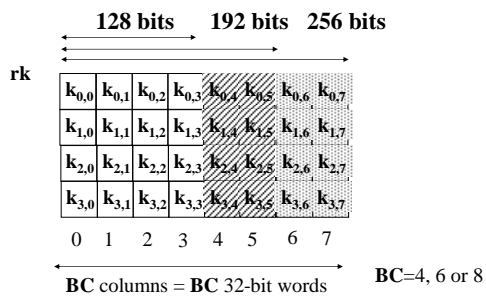
$a_{0,0} a_{1,0} a_{2,0} a_{3,0} a_{0,1} a_{1,1} a_{2,1} a_{3,1} a_{0,2} a_{1,2} a_{2,2} a_{3,2} a_{0,3} a_{1,3} a_{2,3} a_{3,3} \dots$



Key, Internal keys

Variable key size

$k_{0,0} k_{1,0} k_{2,0} k_{3,0} k_{0,1} k_{1,1} k_{2,1} k_{3,1} k_{0,2} k_{1,2} k_{2,2} k_{3,2} k_{0,3} k_{1,3} k_{2,3} k_{3,3} \dots$



Number of rounds

Key length

Block length	128 bits Nk=4	192 bits Nk=6	256 bits Nk=8
128 bits Nb=4	10	12	14
required by the standard			
192 bits Nb=6	12	12	14
256 bits Nb=8	14	14	14
non-standard extensions			

Round function - decryption

```

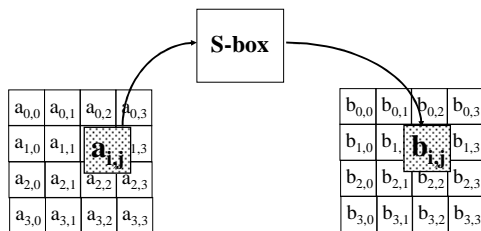
KeyAddition(a,rk[ROUNDS],BC);
Substitution(a,Si,BC);
ShiftRow(a,1,BC);

/* ROUNDS-1 ordinary rounds */
for(r = ROUNDS-1; r > 0; r--)
{
    KeyAddition(a,rk[r],BC);
    InvMixColumn(a,BC);
    Substitution(a,Si,BC);
    ShiftRow(a,1,BC);
}

/* End with the extra key addition */
KeyAddition(a,rk[0],BC);

```

SubBytes



Reference implementation - SubBytes (1)

BC = number of 32-bit words in the input block
MAXBC=8 - maximum number of 32-bit words in the input block

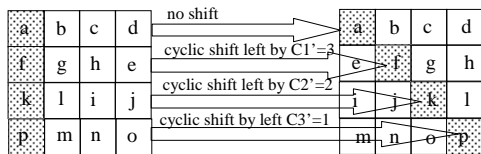
```

void Substitution(word8 a[4][MAXBC], word8 box[256], word8 BC)
{
    /* Replace every byte of the input by the byte at that place
    in the nonlinear S-box */
    int i, j;

    for(i = 0; i < 4; i++)
        for(j = 0; j < BC; j++) a[i][j] = box[a[i][j]] ;
}

```

InvShiftRows



	Block size		
	128 bits	192 bits	256 bits
C1'	3	5	7
C2'	2	4	5
C3'	1	3	4

Reference implementation ShiftRows, InvShiftRows(1)

```
static word8 shifts[3][4][2] = {
    0, 0,
    1, 3,
    2, 2,
    3, 1,
    for BC=4, block size=128 bits
    0, 0,
    1, 5,
    2, 4,
    3, 3,
    for BC=6, block size=192 bits
    0, 0,
    1, 7,
    3, 5,
    4, 4,
    for BC=8, block size=256 bits
};
```

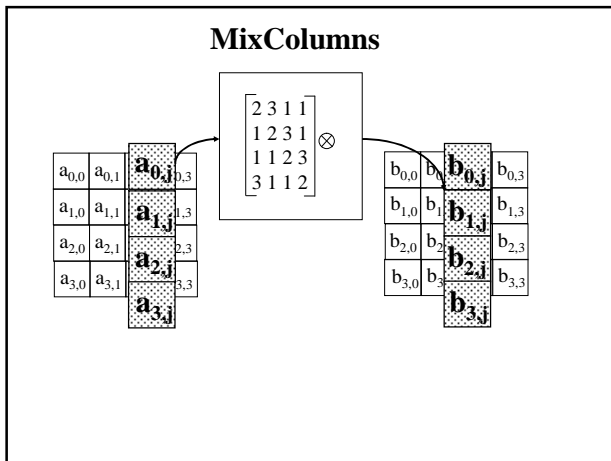
Number of shifts during encryption

Number of shifts during decryption

Reference implementation ShiftRows, InvShiftRows(2)

```
void ShiftRow(word8 a[4][MAXBC], word8 d, word8 BC) {
    /* Row 0 remains unchanged
    * The other three rows are shifted a variable amount
    */
    word8 tmp[MAXBC];
    int i, j;

    for(i = 1; i < 4; i++) {
        for(j = 0; j < BC; j++)
            tmp[j] = a[i][(j + shifts[SC][i][d]) % BC];
        for(j = 0; j < BC; j++) a[i][j] = tmp[j];
    }
}
```



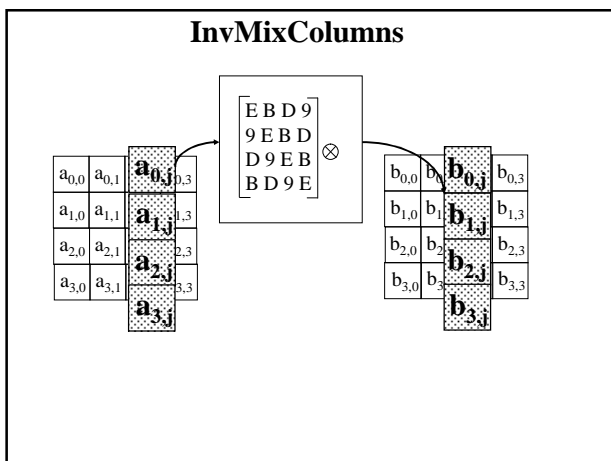
Reference implementation MixColumns (1)

```

void MixColumn(word8 a[4][MAXBC], word8 BC) {
    /* Mix the four bytes of every column in a linear way */
    word8 b[4][MAXBC];
    int i, j;

    for(j = 0; j < BC; j++)
        for(i = 0; i < 4; i++)
            b[i][j] = mul(2, a[i][j])
                ^ mul(3, a[(i + 1) % 4][j])
                ^ a[(i + 2) % 4][j]
                ^ a[(i + 3) % 4][j];
    for(i = 0; i < 4; i++)
        for(j = 0; j < BC; j++) a[i][j] = b[i][j];
}

```



Reference implementation InvMixColumns (1)

```

void InvMixColumn(word8 a[4][MAXBC], word8 BC) {
    /* Mix the four bytes of every column in a linear way
    * This is the opposite operation of Mixcolumn
    */
    word8 b[4][MAXBC];
    int i, j;

    for(j = 0; j < BC; j++)
    for(i = 0; i < 4; i++)
        b[i][j] = mul(0xe, a[i][j])
            ^ mul(0xb, a[(i + 1) % 4][j])
            ^ mul(0xd, a[(i + 2) % 4][j])
            ^ mul(0x9, a[(i + 3) % 4][j]);
    for(i = 0; i < 4; i++)
        for(j = 0; j < BC; j++) a[i][j] = b[i][j];
}

```

Multiplication in the Galois Field GF(2⁸)

```

word8 mul(word8 a, word8 b) {
    /* multiply two elements of GF(2^m)
    * needed for MixColumn and InvMixColumn
    */
    if (a && b)
        return Alogtable[(Logtable[a] + Logtable[b])%255];
    else return 0;
}

```

Multiplication in the Galois Field GF(2⁸) using logarithms and antilogarithms

Let $g \leftrightarrow g(x)$ be a generator of all non-zero elements of GF(2⁸), i.e.,

$$g, g^2, g^3, g^4, \dots, g^{254}, g^{255}=1$$

are all different .

Then for every pair of non-zero elements a, b there exist m, n , such that

$$\begin{aligned}
 a &= g^m & m &= \text{logarithm of } a \text{ to the base } g \\
 b &= g^n & n &= \text{logarithm of } b \text{ to the base } g
 \end{aligned}$$

Multiplication in the Galois Field GF(2⁸) using logarithms and antilogarithms

$$c = a \cdot b = g^m \cdot g^n = g^{m+n} = g^{(m+n) \bmod 255}$$

We can create tables

$$\text{Alogtable}[i] = g^i \quad i = 0..255$$

$$\text{Logtable}[e] = j, \quad \text{such that } g^j = e, \quad \text{where } e = \text{'01'..'FF'}$$

We choose

$$g = \{3\} \leftrightarrow g(x) = x+1$$

Table of values $a = g^i \leftrightarrow a(x) = (x+1)^i \bmod m(x)$ for $i=0..255$

```
word8 Alogtable[256] = {
  1, 3, 5, 15, 17, 51, 85, 255, 26, 46, 114, 150, 161, 248, 19, 53,
  95, 225, 56, 72, 216, 115, 149, 164, 247, 2, 6, 10, 30, 34, 102, 170,
  229, 52, 92, 228, 55, 89, 235, 38, 106, 190, 217, 112, 144, 171, 230, 49,
  83, 245, 4, 12, 20, 60, 68, 204, 79, 209, 104, 184, 211, 110, 178, 205,
  76, 212, 103, 169, 224, 59, 77, 215, 98, 166, 241, 8, 24, 40, 120, 136,
  131, 158, 185, 208, 107, 189, 220, 127, 129, 152, 179, 206, 73, 219, 118, 154,
  181, 196, 87, 249, 16, 48, 80, 240, 11, 29, 39, 105, 187, 214, 97, 163,
  254, 25, 43, 125, 135, 146, 173, 236, 47, 113, 147, 174, 233, 32, 96, 160,
  251, 22, 58, 78, 210, 109, 183, 194, 93, 231, 50, 86, 250, 21, 63, 65,
  195, 94, 226, 61, 71, 201, 64, 192, 91, 237, 44, 116, 156, 191, 218, 117,
  159, 186, 213, 100, 172, 239, 42, 126, 130, 157, 188, 223, 122, 142, 137, 128,
  155, 182, 193, 88, 232, 35, 101, 175, 234, 37, 111, 177, 200, 67, 197, 84,
  252, 31, 33, 99, 165, 244, 7, 9, 27, 45, 119, 153, 176, 203, 70, 202,
  69, 207, 74, 222, 121, 139, 134, 145, 168, 227, 62, 66, 198, 81, 243, 14,
  18, 54, 90, 238, 41, 123, 141, 140, 143, 138, 133, 148, 167, 242, 13, 23,
  57, 75, 221, 124, 132, 151, 162, 253, 28, 36, 108, 180, 199, 82, 246, 1,
};
```

Table of values $j = \log_g e \leftrightarrow j = \log_{x+1}(e(x)) \bmod m(x)$ for $e=1..255=\text{'01'..'FF'}$

```
word8 Logtable[256] = {
  0, 0, 25, 1, 50, 2, 26, 198, 75, 199, 27, 104, 51, 238, 223, 3,
  100, 4, 224, 14, 52, 141, 129, 239, 76, 113, 8, 200, 248, 105, 28, 193,
  125, 194, 29, 181, 249, 185, 39, 106, 77, 228, 166, 114, 154, 201, 9, 120,
  101, 47, 138, 5, 33, 15, 225, 36, 18, 240, 130, 69, 53, 147, 218, 142,
  150, 143, 219, 189, 54, 208, 206, 148, 19, 92, 210, 241, 64, 70, 131, 56,
  102, 221, 253, 48, 191, 6, 139, 98, 179, 37, 226, 152, 34, 136, 145, 16,
  126, 110, 72, 195, 163, 182, 30, 66, 58, 107, 40, 84, 250, 133, 61, 186,
  43, 121, 10, 21, 155, 159, 94, 202, 78, 212, 172, 229, 243, 115, 167, 87,
  175, 88, 168, 80, 244, 234, 214, 116, 79, 174, 233, 213, 231, 230, 173, 232,
  44, 215, 117, 122, 235, 22, 11, 245, 89, 203, 95, 176, 156, 169, 81, 160,
  127, 12, 246, 111, 23, 196, 73, 236, 216, 67, 31, 45, 164, 118, 123, 183,
  204, 187, 62, 90, 251, 96, 177, 134, 59, 82, 161, 108, 170, 85, 41, 157,
  151, 178, 135, 144, 97, 190, 220, 252, 188, 149, 207, 205, 55, 63, 91, 209,
  83, 57, 132, 60, 65, 162, 109, 71, 20, 42, 158, 93, 86, 242, 211, 171,
  68, 17, 146, 217, 35, 32, 46, 137, 180, 124, 184, 38, 119, 153, 227, 165,
  103, 74, 237, 222, 197, 49, 254, 24, 13, 99, 140, 128, 192, 247, 112, 7,
};
```

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}

+

k _{0,0}	k _{0,1}	k _{0,2}	k _{0,3}
k _{1,0}	k _{1,1}	k _{1,2}	k _{1,3}
k _{2,0}	k _{2,1}	k _{2,2}	k _{2,3}
k _{3,0}	k _{3,1}	k _{3,2}	k _{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}

- simple bitwise addition (xor) of round keys

Reference implementation AddRoundKey

```
void KeyAddition(word8 a[4][MAXBC], word8 rk[4][MAXBC], word8 BC)
{
    /* Exor corresponding text input and round key input bytes
    */
    int i, j;

    for(i = 0; i < 4; i++)
        for(j = 0; j < BC; j++) a[i][j] ^= rk[i][j];
}
```

Optimized software implementation

Mathematical description of the round transformations (1)

SubBytes

$$b_{i,j} = s[a_{i,j}]$$

ShiftRows

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j+c1} \\ b_{2,j+c2} \\ b_{3,j+c3} \end{bmatrix}$$

← sums mod BC=Nb

Mathematical description of the round transformations (2)

MixColumns

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}$$

AddRoundKey

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Mathematical description of the entire round

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s[a_{0,j}] \\ s[a_{1,j+c1}] \\ s[a_{2,j+c2}] \\ s[a_{3,j+c3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$



$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = s[a_{0,j}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus s[a_{1,j+c1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus s[a_{2,j+c2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus s[a_{3,j+c3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Fast implementation of the entire round (1)

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = s[a_{0,j}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus s[a_{1,j+c1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus s[a_{2,j+c2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus s[a_{3,j+c3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \\
 \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = T_0[a_{0,j}] \oplus T_1[a_{1,j+c1}] \oplus T_2[a_{2,j+c2}] \oplus T_3[a_{3,j+c3}] \\
 \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Definition of T-tables

$$T_0[a] = \begin{bmatrix} 02 \cdot S[a] \\ S[a] \\ S[a] \\ 03 \cdot S[a] \end{bmatrix} \quad T_1[a] = \begin{bmatrix} 03 \cdot S[a] \\ 02 \cdot S[a] \\ S[a] \\ S[a] \end{bmatrix} \\
 T_2[a] = \begin{bmatrix} S[a] \\ 03 \cdot S[a] \\ 02 \cdot S[a] \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ 03 \cdot S[a] \\ 02 \cdot S[a] \end{bmatrix}$$

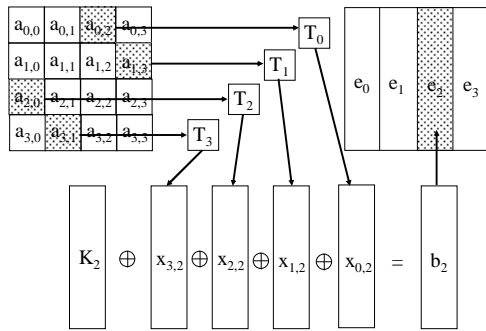
Look-up Tables T

```

static const u32 Te0[256] =
{
    0xc66363a5U, 0xf87c7c84U,
    0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU,
    0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U,
    0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U,
    0x4dababe6U, 0xec76769aU,
    0x8fcaca45U, 0x1f82829dU,
    0x89c9c940U, 0xfa7d7d87U,
    0xeffafa15U, 0xb25959ebU,
    0x8e4747c9U, 0xfbff0f00bU,
    . . . . .

```

Table-lookup implementation



$$e_2 = T_0[a_{0,2}] \oplus T_1[a_{1,3}] \oplus T_2[a_{2,0}] \oplus T_3[a_{3,1}] \oplus K_2$$

Optimized implementation - Round function

```

t0 = Te0[(s0 >> 24)      ] ^
     Te1[(s1 >> 16) & 0xff] ^
     Te2[(s2 >>  8) & 0xff] ^
     Te3[(s3      ) & 0xff] ^
     rk[4];
t1 = Te0[(s1 >> 24)      ] ^
     Te1[(s2 >> 16) & 0xff] ^
     Te2[(s3 >>  8) & 0xff] ^
     Te3[(s0      ) & 0xff] ^
     rk[5];
. . . . .
    
```

Optimized encryption code

```

void rijndaelEncrypt
(const u32 rk[/*4*(Nr + 1)*/],
 int Nr,
 const u8 pt[16],
 u8 ct[16])
{
    u32 s0, s1, s2, s3,
        t0, t1, t2, t3, r;

    /* map byte array block to cipher state
    and add initial round key: */

    s0 = GETU32(pt      ) ^ rk[0];
    s1 = GETU32(pt +  4) ^ rk[1];
    s2 = GETU32(pt +  8) ^ rk[2];
    s3 = GETU32(pt + 12) ^ rk[3];
}
    
```

```

r = Nr >> 1;
for (;;)
{ t0 = Te0[(s0 >> 24)] ^
  Te1[(s1 >> 16) & 0xff] ^
  Te2[(s2 >> 8) & 0xff] ^
  Te3[(s3 ) & 0xff] ^
  rk[4];
t1 = Te0[(s1 >> 24)] ^
  Te1[(s2 >> 16) & 0xff] ^
  Te2[(s3 >> 8) & 0xff] ^
  Te3[(s0 ) & 0xff] ^
  rk[5];
t2 = Te0[(s2 >> 24)] ^
  Te1[(s3 >> 16) & 0xff] ^
  Te2[(s0 >> 8) & 0xff] ^
  Te3[(s1 ) & 0xff] ^
  rk[6];
t3 = Te0[(s3 >> 24)] ^
  Te1[(s0 >> 16) & 0xff] ^
  Te2[(s1 >> 8) & 0xff] ^
  Te3[(s2 ) & 0xff] ^
  rk[7];

rk += 8;
if (--r == 0)
{break;}

```

```

s0 = Te0[(t0 >> 24)] ^
  Te1[(t1 >> 16) & 0xff] ^
  Te2[(t2 >> 8) & 0xff] ^
  Te3[(t3 ) & 0xff] ^
  rk[0];
s1 = Te0[(t1 >> 24)] ^
  Te1[(t2 >> 16) & 0xff] ^
  Te2[(t3 >> 8) & 0xff] ^
  Te3[(t0 ) & 0xff] ^
  rk[1];
s2 = Te0[(t2 >> 24)] ^
  Te1[(t3 >> 16) & 0xff] ^
  Te2[(t0 >> 8) & 0xff] ^
  Te3[(t1 ) & 0xff] ^
  rk[2];
s3 = Te0[(t3 >> 24)] ^
  Te1[(t0 >> 16) & 0xff] ^
  Te2[(t1 >> 8) & 0xff] ^
  Te3[(t2 ) & 0xff] ^
  rk[3];
}

```

```

/* * apply last round
   * map cipher state to byte array block: */
s0 = (Te4[(t0 >> 24) ] & 0xff000000) ^
      (Te4[(t1 >> 16) & 0xff] & 0x00ff0000) ^
      (Te4[(t2 >> 8) & 0xff] & 0x0000ff00) ^
      (Te4[(t3 ) & 0xff] & 0x000000ff) ^
      rk[0];
PUTU32(ct , s0);
s1 = (Te4[(t1 >> 24) ] & 0xff000000) ^
      (Te4[(t2 >> 16) & 0xff] & 0x00ff0000) ^
      (Te4[(t3 >> 8) & 0xff] & 0x0000ff00) ^
      (Te4[(t0 ) & 0xff] & 0x000000ff) ^
      rk[1];
PUTU32(ct + 4, s1);
s2 = (Te4[(t2 >> 24) ] & 0xff000000) ^
      (Te4[(t3 >> 16) & 0xff] & 0x00ff0000) ^
      (Te4[(t0 >> 8) & 0xff] & 0x0000ff00) ^
      (Te4[(t1 ) & 0xff] & 0x000000ff) ^
      rk[2];
PUTU32(ct + 8, s2);
s3 = (Te4[(t3 >> 24) ] & 0xff000000) ^
      (Te4[(t0 >> 16) & 0xff] & 0x00ff0000) ^
      (Te4[(t1 >> 8) & 0xff] & 0x0000ff00) ^
      (Te4[(t2 ) & 0xff] & 0x000000ff) ^
      rk[3];
PUTU32(ct + 12, s3);}

```

Fast implementation of the entire round (1)

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = s[a_{0,j}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus s[a_{1,j+c_1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus s[a_{2,j+c_2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus s[a_{3,j+c_3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix}$$

$$\oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = T_0[a_{0,j}] \oplus T_1[a_{1,j+c_1}] \oplus T_2[a_{2,j+c_2}] \oplus T_3[a_{3,j+c_3}] \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Fast implementation of the entire round (2)

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = T_0[a_{0,j}] \oplus T_1[a_{1,j+c_1}] \oplus T_2[a_{2,j+c_2}] \oplus T_3[a_{3,j+c_3}]$$

$$\oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

Memory: 4 x 256 x 4 B = 4 kB
for tables T_0, T_1, T_2, T_3

Time: 20 LUTs + 16 XORs

$$e_j = T_0[a_{0,j}] \oplus \text{RotByte}(T_0[a_{1,j+c_1}] \oplus (\text{RotByte}(T_0[a_{2,j+c_2}] \oplus \text{RotByte}(T_0[a_{3,j+c_3}])))) \oplus K_j$$

Memory: 256 x 4 B = 1 kB
of memory table T_0

Time: 20 LUTs + 16 XORs + 12 ROTs

Optimized Decryption

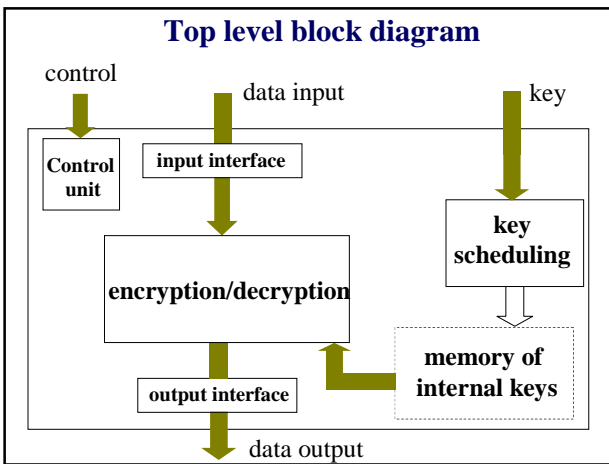
$$d_j = T_0^{-1}[a_{0,j}] \oplus T_1^{-1}[a_{1,j+3}] \oplus T_2^{-1}[a_{2,j+2}] \oplus T_3^{-1}[a_{3,j+1}] \oplus K_j \quad (50)$$

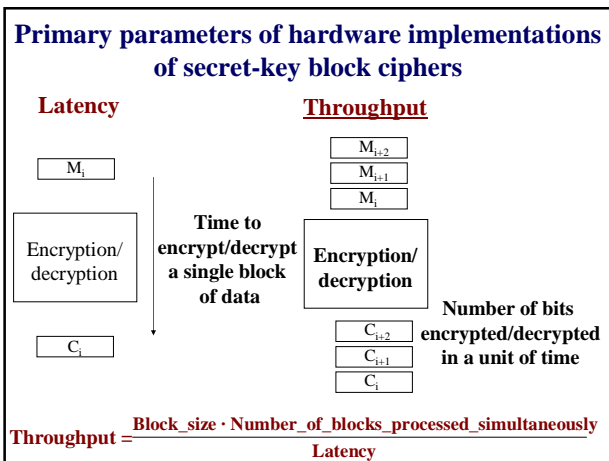
where $T_0^{-1}, T_1^{-1}, T_2^{-1}, T_3^{-1}$ are the precomputed 8x32-bit look-up tables, and K_j is a j -th word of a round key K . All indices $j+3, j+2, j+1$ are computed modulo 4.

$$T_0^{-1}[a] = \begin{bmatrix} 0E \cdot S[a] \\ 09 \cdot S[a] \\ 0D \cdot S[a] \\ 0B \cdot S[a] \end{bmatrix} \quad T_1^{-1}[a] = \begin{bmatrix} 0B \cdot S[a] \\ 0E \cdot S[a] \\ 09 \cdot S[a] \\ 0D \cdot S[a] \end{bmatrix}$$

$$T_2^{-1}[a] = \begin{bmatrix} 0D \cdot S[a] \\ 0B \cdot S[a] \\ 0E \cdot S[a] \\ 09 \cdot S[a] \end{bmatrix} \quad T_3^{-1}[a] = \begin{bmatrix} 09 \cdot S[a] \\ 0D \cdot S[a] \\ 0B \cdot S[a] \\ 0E \cdot S[a] \end{bmatrix}$$

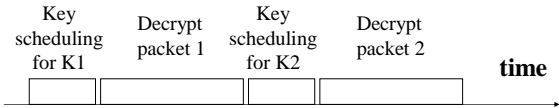
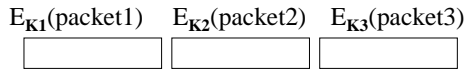
AES in Hardware





Key agility

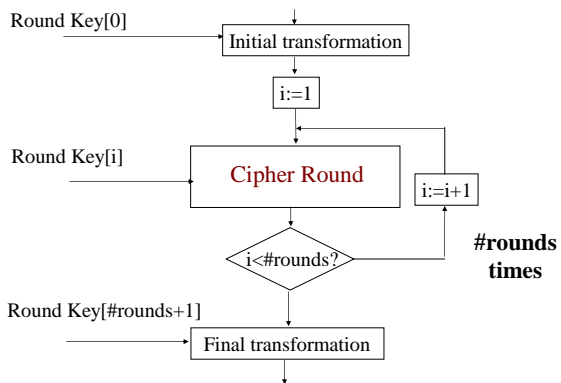
Packet-switched networks



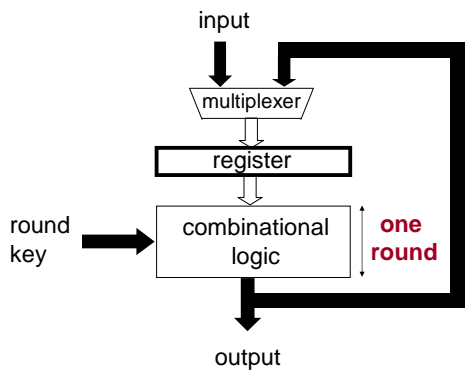
$$\text{Overhead \%} = \frac{\text{Time of key scheduling}}{\text{Time of single block decryption}} \cdot \frac{1}{\# \text{ blocks/packet}}$$

Hardware Architectures of Block Ciphers

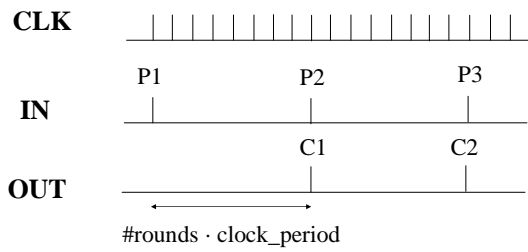
Typical Internal Structure of a Secret-Key Block Cipher



Basic iterative architecture



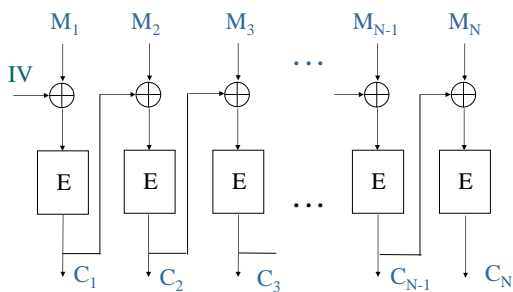
Basic architecture: Timing



$$Throughput_{iterative} = \frac{block_size}{\#rounds \cdot T_{CLK_{iterative}}}$$

$$Latency_{iterative} = \#rounds \cdot T_{CLK_{iterative}}$$

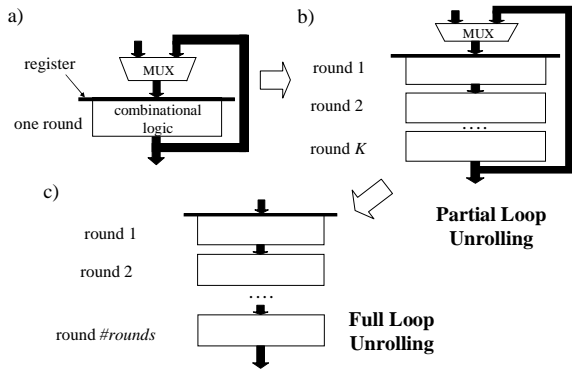
Feedback cipher modes - CBC



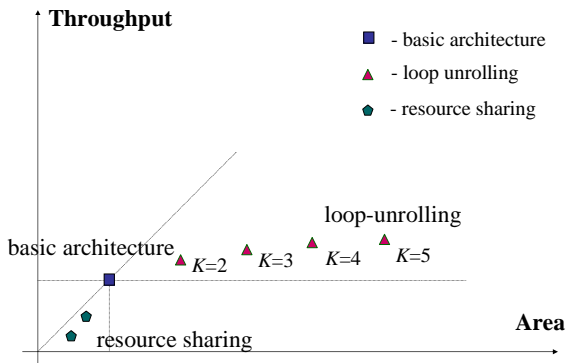
$$C_1 = AES(M_1 \oplus IV)$$

$$C_i = AES(M_i \oplus C_{i-1}) \quad \text{for } i=2..N$$

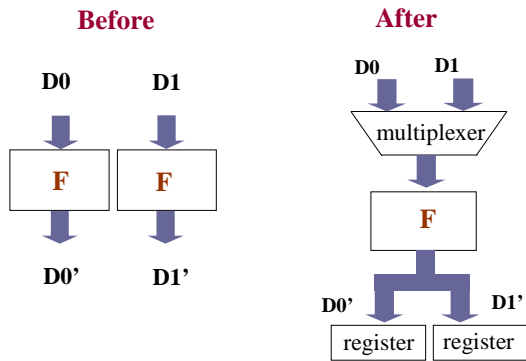
Architectures Suitable for Feedback Cipher Modes



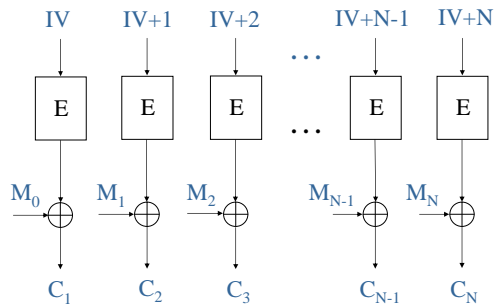
Throughput vs. Area for Architectures with Loop Unrolling



Decreasing area by resource sharing

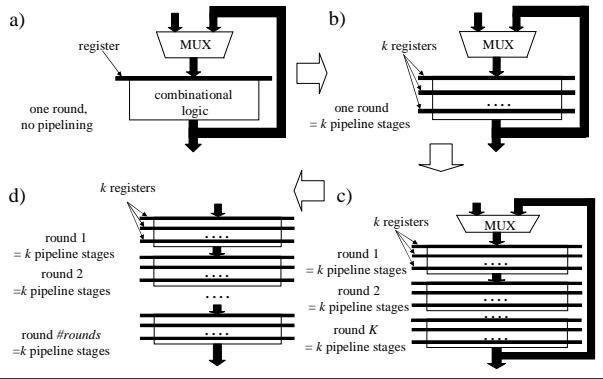


Non-feedback Counter Mode - CTR

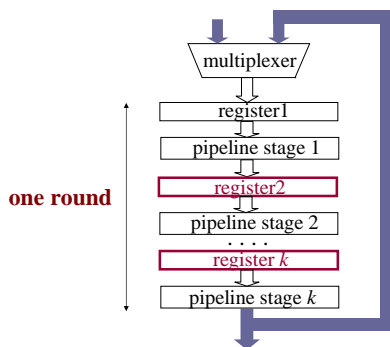


$$C_i = M_i \oplus \text{AES}(IV+i) \quad \text{for } i=0..N$$

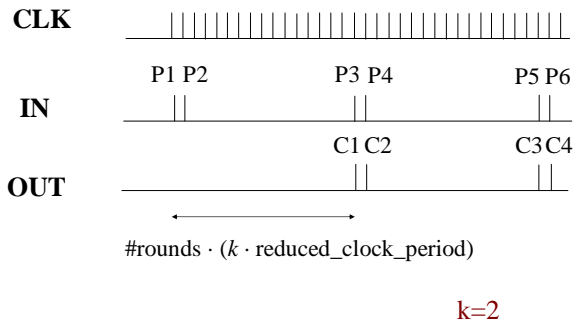
Architectures Suitable for Non-Feedback Cipher Modes



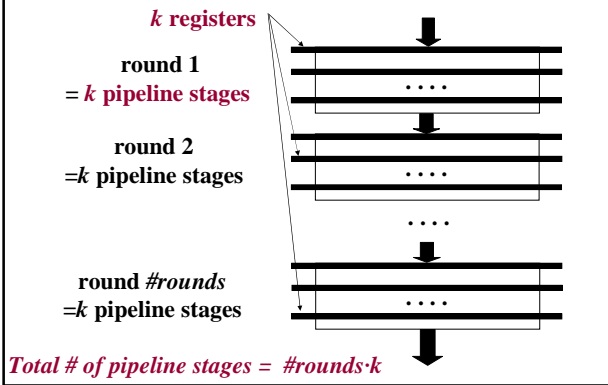
Inner-Round Pipelining



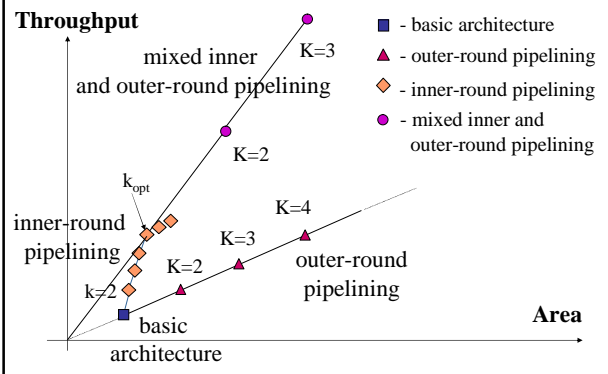
Inner-Round Pipelining: Timing



Full Mixed Inner- and Outer-round Pipelining



Throughput vs. Area for Architectures with Pipelining

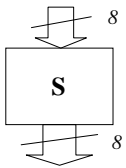


Implementation of Basic Operations of AES

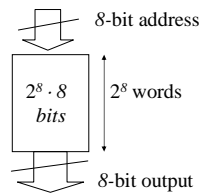
S-box and Inversion in $GF(2^8)$

Hardware

S-box 8×8



ROM

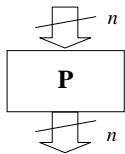


direct logic

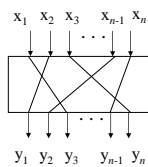


Permutation

Hardware

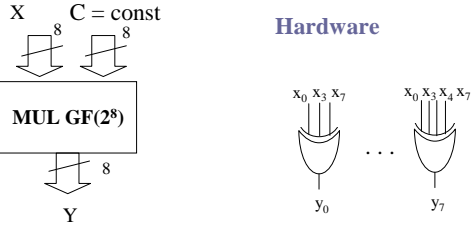


Permutation



order of wires

Multiplication by a constant in the Galois Field $GF(2^8)$



Hardware implementation - MixColumns

```
architecture mul_03 of mul_03 is
begin
    output(7) <= input(7) xor input(6);
    output(6) <= input(6) xor input(5);
    output(5) <= input(5) xor input(4);
    output(4) <= input(4) xor input(3) xor input(7);
    output(3) <= input(3) xor input(2) xor input(7);
    output(2) <= input(2) xor input(1);
    output(1) <= input(1) xor input(0) xor input(7);
    output(0) <= input(0) xor input(7);
end mul_03;

b0 <= a0_02 xor a1_03 xor a2 xor a3;
```

Hardware implementation - InvMixColumns

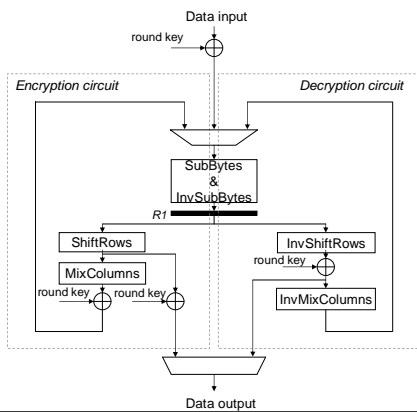
```
architecture mul_0E of mul_0E is
begin
    output(7) <= input(7) xor input(6) xor input(5) xor input(4);
    output(6) <= input(6) xor input(5) xor input(4) xor input(3) xor input(7);
    output(5) <= input(5) xor input(4) xor input(3) xor input(2) xor input(6);
    output(4) <= input(4) xor input(3) xor input(2) xor input(1) xor input(5);
    output(3) <= input(3) xor input(2) xor input(1) xor input(0) xor input(6) xor input(5);
    output(2) <= input(2) xor input(1) xor input(0) xor input(6);
    output(1) <= input(1) xor input(0) xor input(5);
    output(0) <= input(0) xor input(7) xor input(6) xor input(5);
end mul_0E;

b0 <= a0_0E xor a1_0B xor a2_0D xor a3_09;
```

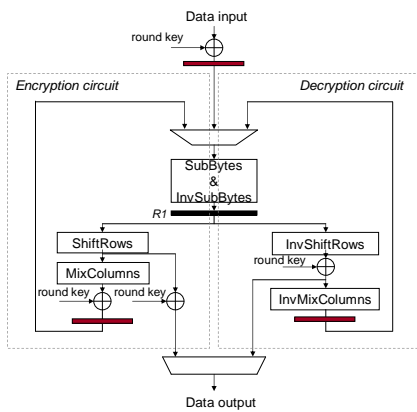
Conclusion: In hardware, decryption slower than encryption

S-box Based Architecture

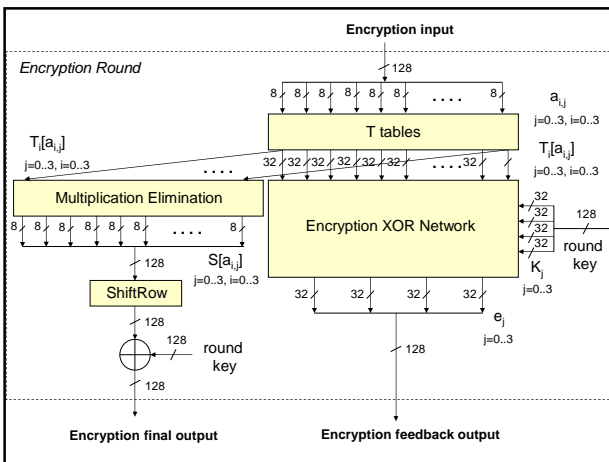
S-box Based Basic Iterative Architecture



S-box Based Architecture with Inner-Round Pipelining



T-box Based Architecture



Multiplication Elimination

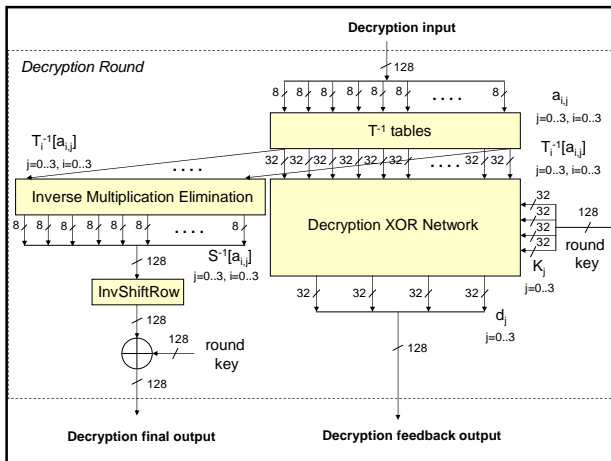
Since MixColumns operation is not performed in the last round of encryption, the last round needs to be treated in a special way.

In this round, S-boxes need to be used instead of T-boxes.

$$T_0[a] = \begin{bmatrix} 02 \cdot S[a] \\ S[a] \\ S[a] \\ 03 \cdot S[a] \end{bmatrix} \quad T_1[a] = \begin{bmatrix} 03 \cdot S[a] \\ 02 \cdot S[a] \\ S[a] \\ S[a] \end{bmatrix}$$

$$T_2[a] = \begin{bmatrix} S[a] \\ 03 \cdot S[a] \\ 02 \cdot S[a] \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ 03 \cdot S[a] \\ 02 \cdot S[a] \end{bmatrix}$$

$$S[a] = \text{byte}(1, T_0[a]) = \text{byte}(2, T_1[a]) = \text{byte}(3, T_2[a]) = \text{byte}(0, T_3[a])$$



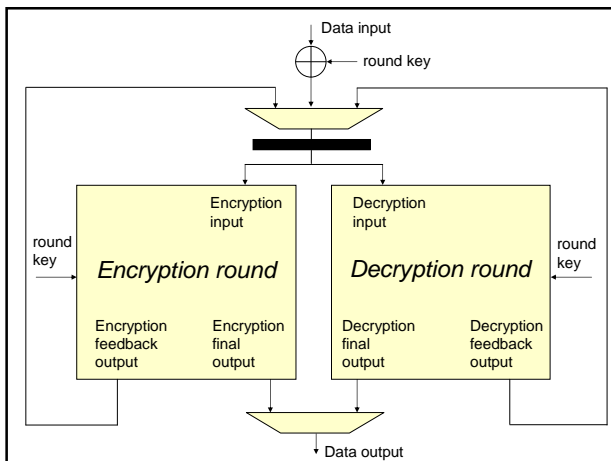
Inverse Multiplication Elimination

Since InvMixColumns operation is not performed in the last round of decryption, the last round needs to be treated in a special way.

$$T_0^{-1}[a] = \begin{bmatrix} 0E \cdot S[a] \\ 09 \cdot S[a] \\ 0D \cdot S[a] \\ 0B \cdot S[a] \end{bmatrix} \quad T_1^{-1}[a] = \begin{bmatrix} 0B \cdot S[a] \\ 0E \cdot S[a] \\ 09 \cdot S[a] \\ 0D \cdot S[a] \end{bmatrix}$$

$$T_2^{-1}[a] = \begin{bmatrix} 0D \cdot S[a] \\ 0B \cdot S[a] \\ 0E \cdot S[a] \\ 09 \cdot S[a] \end{bmatrix} \quad T_3^{-1}[a] = \begin{bmatrix} 09 \cdot S[a] \\ 0D \cdot S[a] \\ 0B \cdot S[a] \\ 0E \cdot S[a] \end{bmatrix}$$

$$\begin{aligned} S^{-1}[a] &= 0E^{-1} \cdot \text{byte}(0, T_0^{-1}[a]) = 09^{-1} \cdot \text{byte}(1, T_0^{-1}[a]) = \\ &= 0D^{-1} \cdot \text{byte}(2, T_0^{-1}[a]) = 0B^{-1} \cdot \text{byte}(3, T_0^{-1}[a]) = \\ &= E5 \cdot \text{byte}(0, T_0[a]) = 4F \cdot \text{byte}(1, T_0[a]) = \\ &= E1 \cdot \text{byte}(2, T_0[a]) = C0 \cdot \text{byte}(3, T_0[a]) \end{aligned}$$



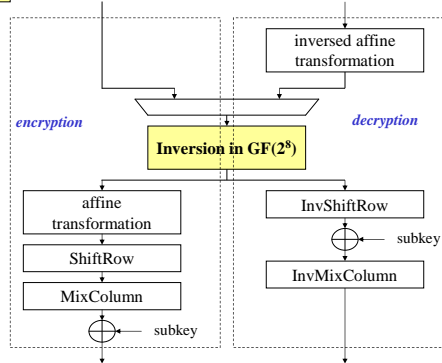
AES Specific Optimizations

Sharing Look-up Tables between SubBytes and InvSubBytes

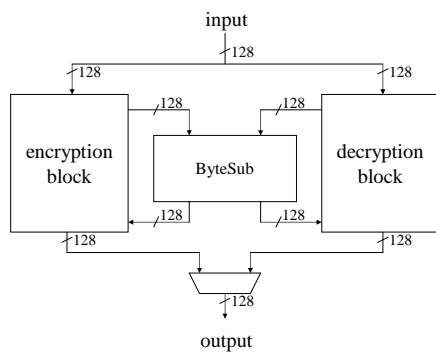
Substitution-Linear Transformation Network:

Rijndael in Hardware

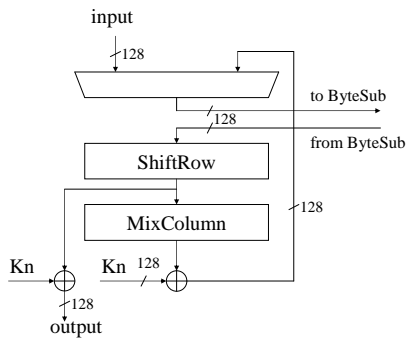
■ - units shared between encryption and decryption



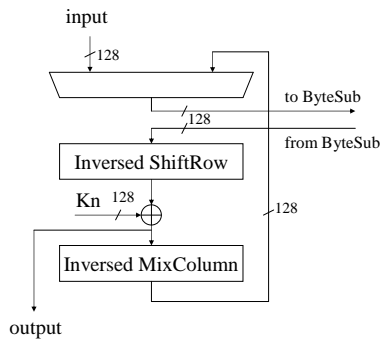
Hardware implementation - top level diagram



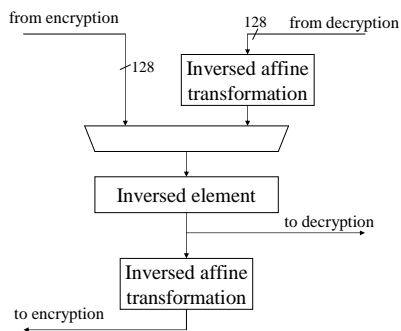
Hardware implementation - Encryption



Hardware implementation - Decryption



Hardware implementation - SubBytes



AES Specific Optimizations

Implementing SubBytes Using Logic Only

Implementing Inversion in GF(2⁸) using Logic Only (1)

- inversion in GF(2⁸) can be decomposed into a sequence of operations in GF(2⁴) (including addition, multiplication, and inversion)
- operations in GF(2⁴) can be expressed in terms of operations in GF(2²)
- operations in GF(2²) in terms of operations in GF(2)
- operations in GF(2) can be implemented using simple logic gates only (XOR gate for addition and AND gate for multiplication)

Implementing Inversion in GF(2⁸) using Logic Only (2)

- logic only implementation can be deeply pipelined leading to very-high-throughput implementations of AES in non-feedback cipher modes of operation (in excess of 20 Gbit/s for the current high-performance families of FPGAs)
- logic only implementation saves dedicated RAMs (Block RAMs) in FPGAs and overall area of the circuit in ASICs
