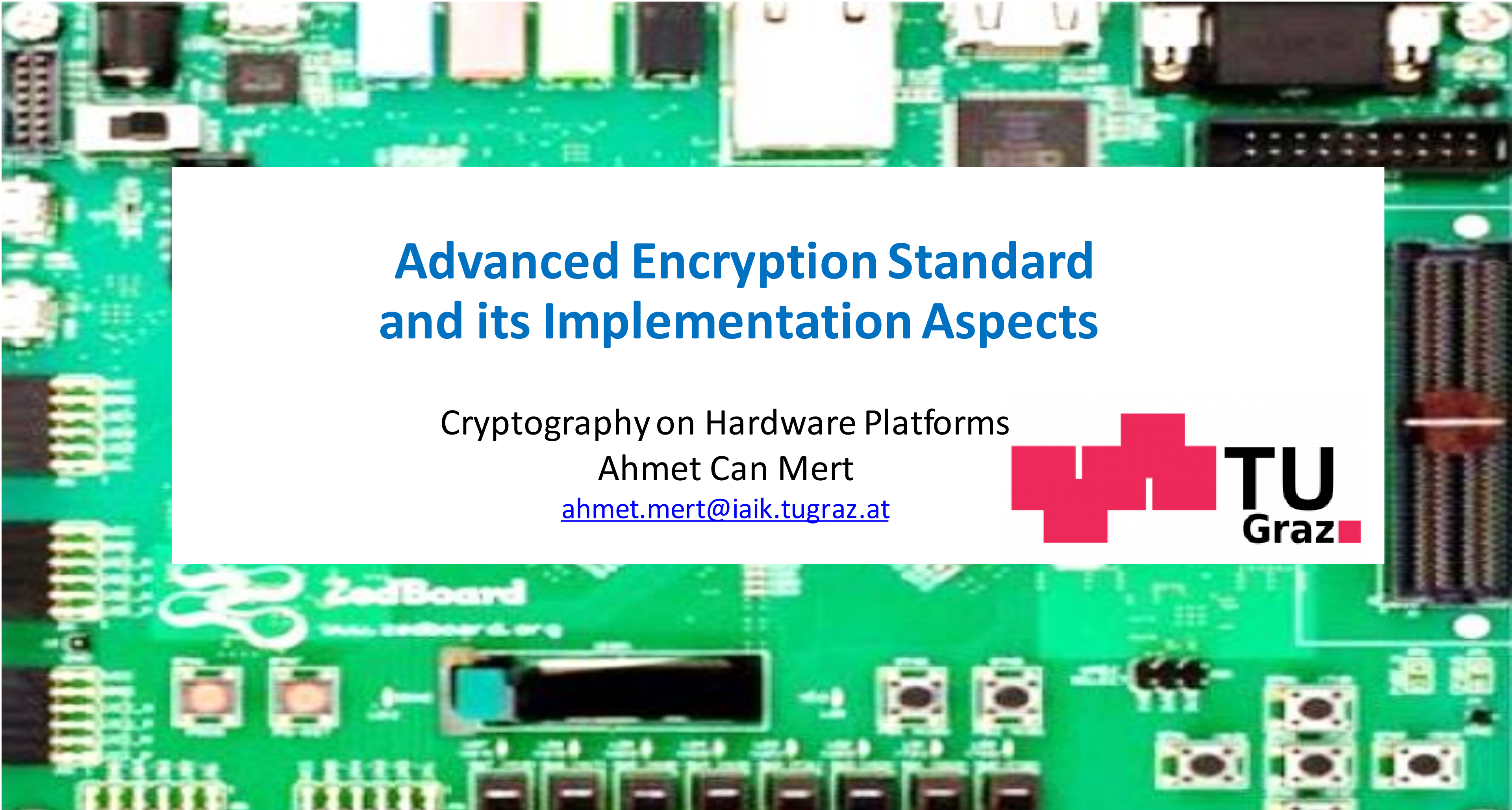


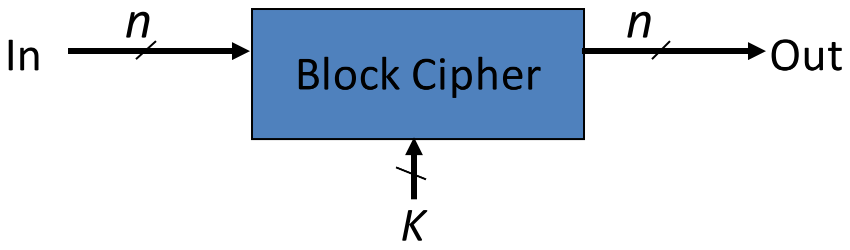
Advanced Encryption Standard and its Implementation Aspects

Cryptography on Hardware Platforms

Ahmet Can Mert
ahmet.mert@iaik.tugraz.at

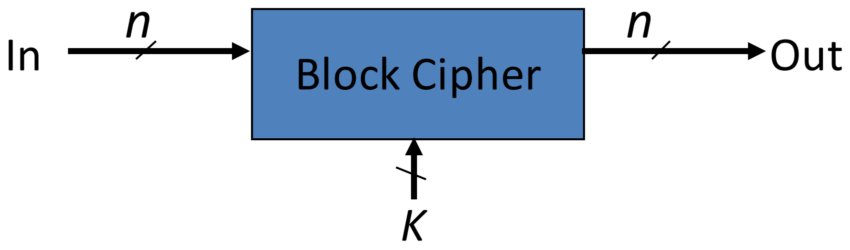
Block Ciphers

- A family of cryptographic functions that map an *n*-bit plaintext block into n-bit ciphertext block.
  - It is parameterized by its key bit length, *K*.

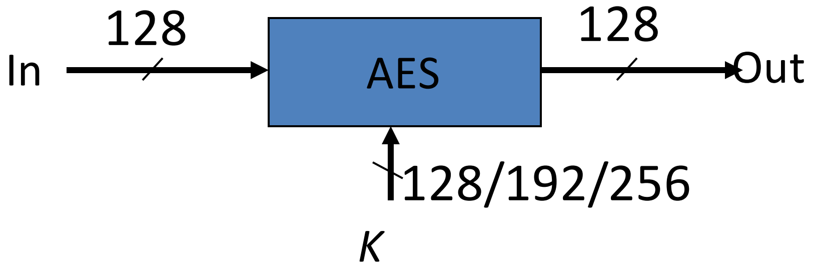


Block Ciphers

- A family of cryptographic functions that map an *n*-bit plaintext block into n-bit ciphertext block.
  - It is parameterized by its key bit length, *K*.



- Rijndael is selected as AES in 2000.



Key Length (*K*)	Nr
128	10
192	12
256	14

AES Overview

- 128-bit (16 bytes) input is arranged into a 4x4 matrix in column-major order.
  - Each matrix entry is an element of GF(2^8) with $x^8+x^4+x^3+x+1$.

$$\underbrace{a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0},}_{\text{Column\#1}} \underbrace{a_{0,1}, a_{1,1}, a_{2,1}, a_{3,1},}_{\text{Column\#2}} \underbrace{a_{0,2}, a_{1,2}, a_{2,2}, a_{3,2},}_{\text{Column\#3}} \underbrace{a_{0,3}, a_{1,3}, a_{2,3}, a_{3,3}}_{\text{Column\#4}}$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

AES Overview

- 128-bit (16 bytes) input is arranged into a 4x4 matrix in column-major order.
  - Each matrix entry is an element of GF(2^8) with $x^8+x^4+x^3+x+1$.

$$\underbrace{a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0},}_{\text{Column\#1}} \underbrace{a_{0,1}, a_{1,1}, a_{2,1}, a_{3,1},}_{\text{Column\#2}} \underbrace{a_{0,2}, a_{1,2}, a_{2,2}, a_{3,2},}_{\text{Column\#3}} \underbrace{a_{0,3}, a_{1,3}, a_{2,3}, a_{3,3}}_{\text{Column\#4}}$$

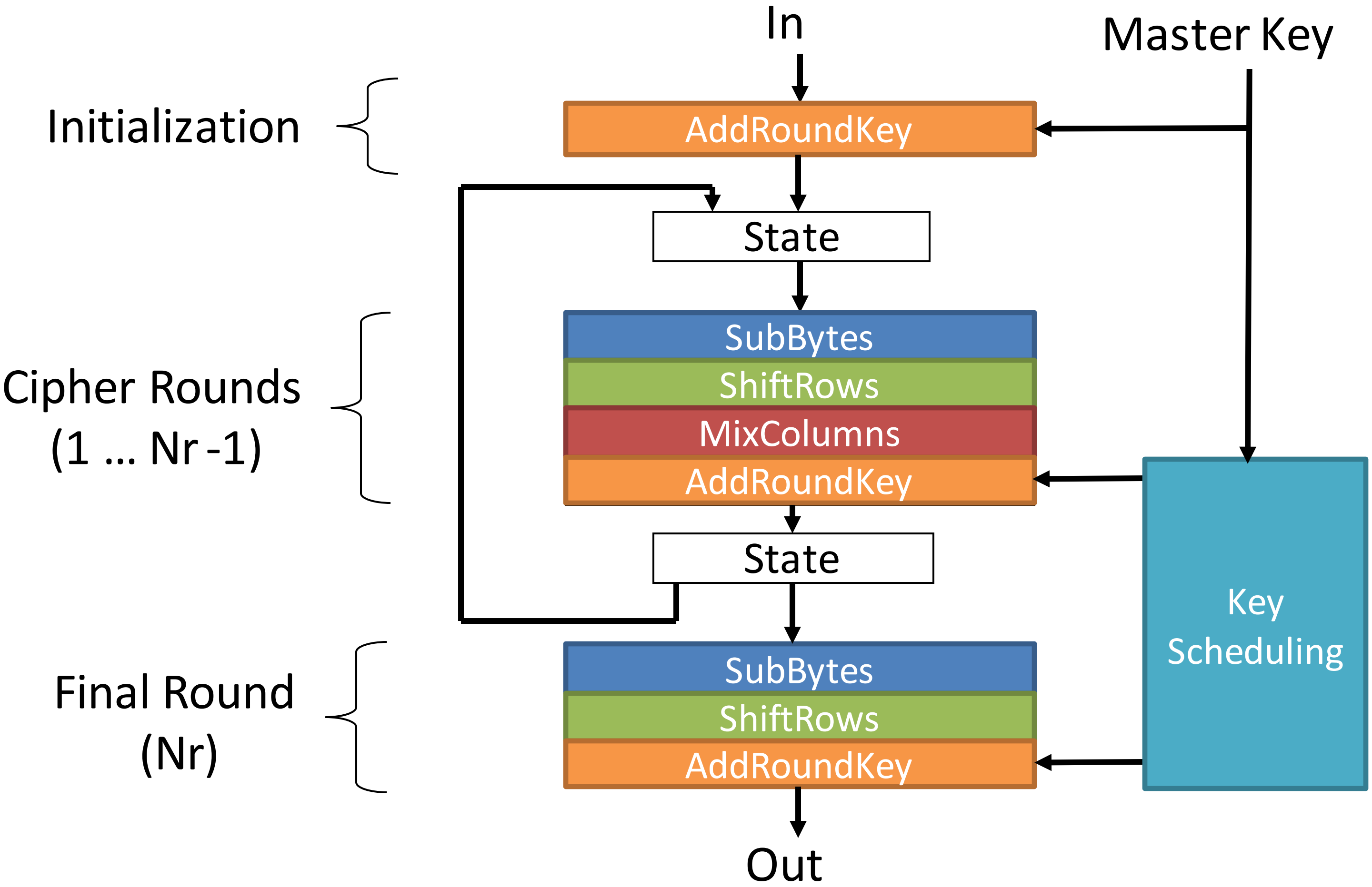
$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

AES Overview

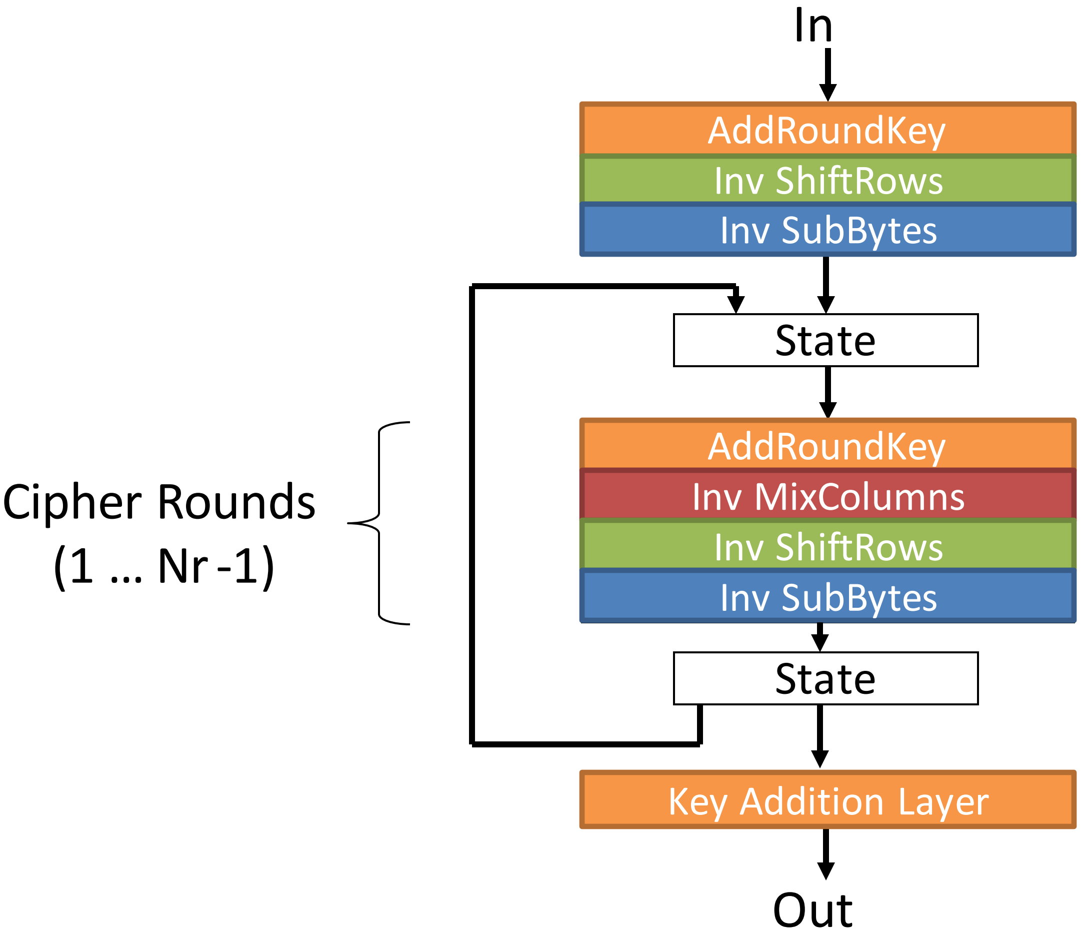
- Rijndael has four main operations:
  - AddRoundKey: XORing the block with the round key.
  - SubBytes**:** Substitute a byte with another byte.
  - ShiftRows**:** Each row of the block is rotated.
  - MixColumns: Each column of the block is multiplied with a polynomial.

- Rijndael also has a key scheduling mechanism to generate round keys.

AES Encryption



AES Decryption



Arithmetic in GF(2^8)

- GF(2^k) is a Galois field of 2^k elements.
  - Also called binary fields.

- GF(2^k) elements in polynomial basis
  - Every element can be represented as a polynomial of power k-1.

$$E = (E_{k-1}E_{k-2} \dots E_1E_0) = E_{k-1}x^{k-1} + E_{k-2}x^{k-2} + \dots + E_1x + E_0$$

$$E_i: \{0, 1\}$$

Arithmetic in GF(2^8)

- GF(2^k) is a Galois field of 2^k elements.
  - Also called binary fields.

- GF(2^k) elements in polynomial basis
  - Every element can be represented as a polynomial of power k-1.

$$E = (E_{k-1}E_{k-2} \ldots E_1E_0) = E_{k-1}x^{k-1} + E_{k-2}x^{k-2} + \ldots + E_1x + E_0$$

$$E_i: \{0, 1\}$$

- AES is using GF(2^8) with irreducible polynomial $\mathbf{x^8 + x^4 + x^3 + x + 1}$.

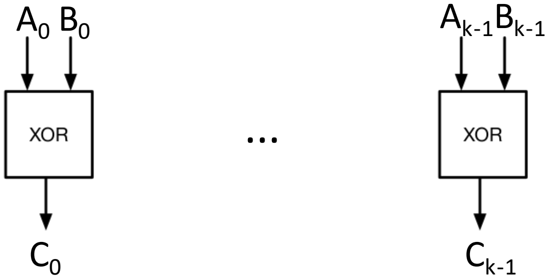
Arithmetic in GF(2^8): Addition

- Addition in GF(2^8) with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.
  - GF(2) addition of the individual bits.
  - GF(2) addition corresponds to the XOR operation in Boolean logic.

- A, B, C in GF(2^8):

$$C_i = A_i + B_i \text{ (mod 2), for } i = 0, \ldots, k\text{-}1$$

- Subtraction is the same as addition



Arithmetic in GF(2^8): Multiplication

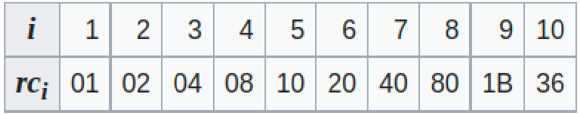
- Multiplication in GF(2^8) with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.
  - Polynomial multiplication (each coefficient is in GF(2)).
  - Reduction with irreducible polynomial.

- Example:

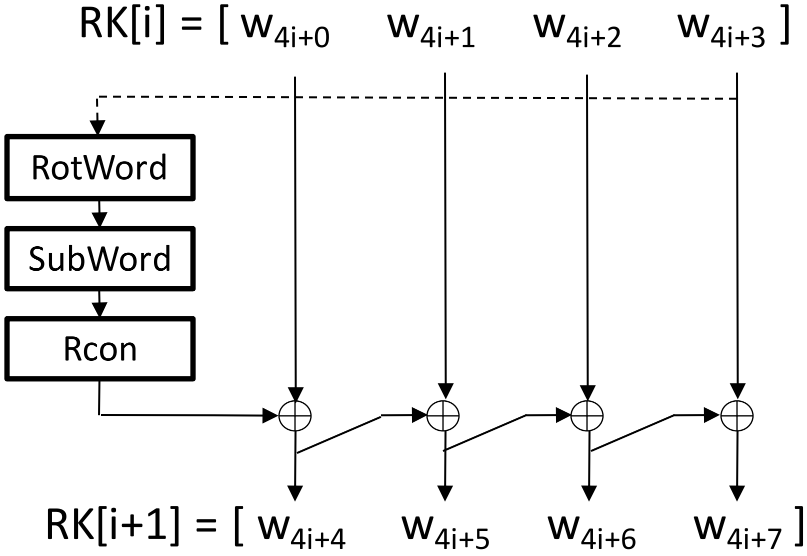
$$\begin{aligned}
201 \,.\, 2 &= (11001001)_2 \,.\, (00000010)_2 \\
&= (x^7 + x^6 + x^3 + 1) \,.\, (x) \\
&= x^8 + x^7 + x^4 + x \;(\text{mod } x^8 + x^4 + x^3 + x + 1) \\
&= x^7 + x^4 + x - x^4 - x^3 - x - 1 \\
&= x^7 + x^3 + 1 \\
&= (10001001)_2 = 129
\end{aligned}$$

AES Key Schedule

- AES takes a single key and generates round keys with the input key and its key scheduling (expansion) algorithm.
  - RotWord: Cyclic left shift
  - SubWord: AES S-box for each byte
  - Rcon: Add with [rc_i 00 00 00]
    - $rc_i = x^{i-1}$ is round constant (can be stored as a table)



i	1	2	3	4	5	6	7	8	9	10
rc_i	01	02	04	08	10	20	40	80	1B	36


RK[i] = [w_{4i+0} w_{4i+1} w_{4i+2} w_{4i+3}]
RotWord
SubWord
Rcon
RK[i+1] = [w_{4i+4} w_{4i+5} w_{4i+6} w_{4i+7}]

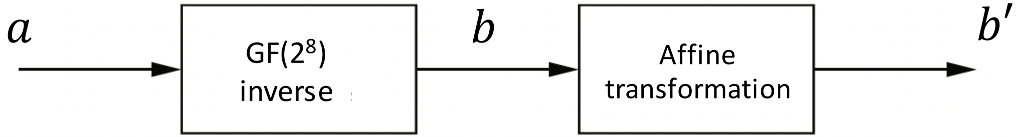
AddRoundKey

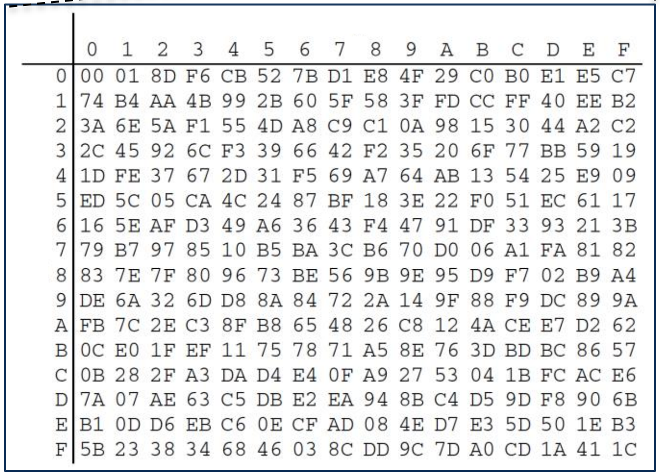
- Round Key Addition.
  - Addition of the current state with the round key in GF(2^8).
  - Simple bit-wise addition (XOR) of state bytes with round key bytes.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \oplus \begin{pmatrix} k_0^i & k_4^i & k_8^i & k_{12}^i \\ k_1^i & k_5^i & k_9^i & k_{13}^i \\ k_2^i & k_6^i & k_{10}^i & k_{14}^i \\ k_3^i & k_7^i & k_{11}^i & k_{15}^i \end{pmatrix}$$

SubBytes

- Byte Substitution (Forward S-box).
  - First, GF(2^8) multiplicative inverse of each byte in round state is computed. Then, an affine transformation is applied to each byte.





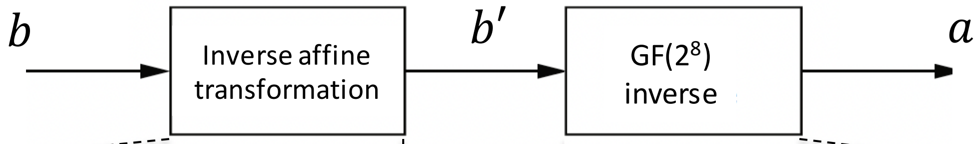
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09
5	ED	5C	05	CA	4C	24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

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

$b(x)(x^7+x^6+x^5+x^4+1) + (x^7+x^6+x^2+x) \bmod x^8+1$

Inv SubBytes

- Inverse Byte Substitution (Inverse S-box).
  - An inverse affine transform is followed by multiplicative inverse operation in GF(2^8) for each state byte.



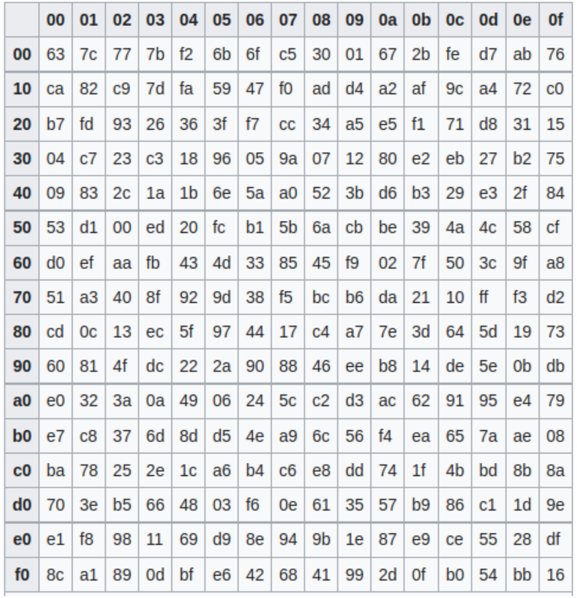
$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1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

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

SubBytes and Inv SubBytes

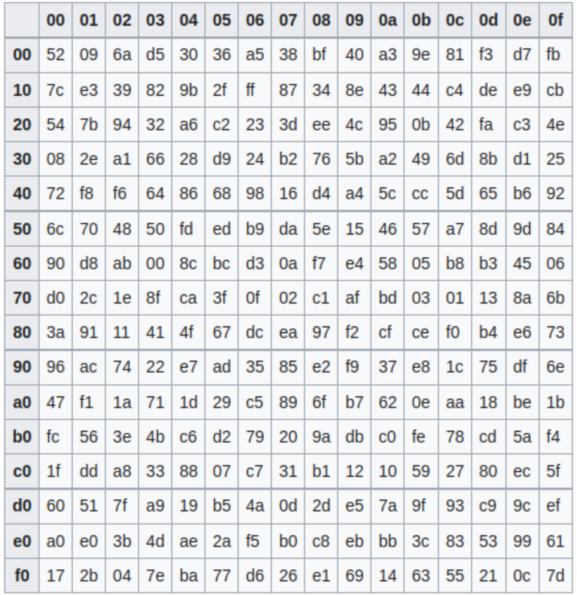
- You can use a table (S-box) to combine affine transformation and GF(2^8) inverse.

Forward S-Box



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

Inverse S-Box



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

ShiftRows

- Shift Row Layer
  - Four rows of the state matrix are shifted cyclically to the left by offsets of
    - 0

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{pmatrix} \leftarrow \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$

ShiftRows

- Shift Row Layer
  - Four rows of the state matrix are shifted cyclically to the left by offsets of
    - 0, 1

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{pmatrix} \leftarrow \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$

ShiftRows

- Shift Row Layer
  - Four rows of the state matrix are shifted cyclically to the left by offsets of
    - 0, 1, 2

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{pmatrix} \leftarrow \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$

ShiftRows

- Shift Row Layer
  - Four rows of the state matrix are shifted cyclically to the left by offsets of
    - 0, 1, 2, 3

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{pmatrix} \leftarrow \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$

ShiftRows

- Shift Row Layer
  - Four rows of the state matrix are shifted cyclically to the left by offsets of
    - 0, 1, 2, 3

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{pmatrix} \leftarrow \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$

- Inv ShiftRows performs right circular shift.

MixColumns

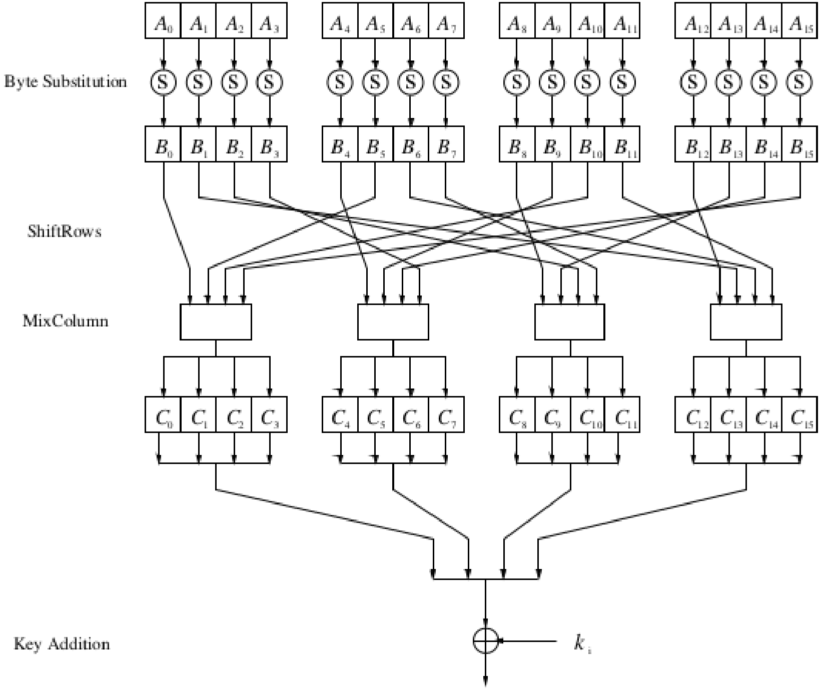
- Mix Column Layer
  - Each column of the state (4-bytes) is considered as a degree-3 polynomial in GF(2^8)$[x]/x^4+1$
  - Then, each polynomial is multiplied with a constant polynomial in the same ring
    - $03.x^3 + 01.x^2 + 01.x + 02$
    - This multiplication can be written as a matrix-vector multiplication

$$\begin{pmatrix} d_0 & d_4 & d_8 & d_{12} \\ d_1 & d_5 & d_9 & d_{13} \\ d_2 & d_6 & d_{10} & d_{14} \\ d_3 & d_7 & d_{11} & d_{15} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} c_0 & c_4 & c_8 & c_{12} \\ c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \end{pmatrix}$$

  - Inverse Mix Column layer uses the inverse of $03.x^3 + 01.x^2 + 01.x + 02$
    - $0B.x^3 + 0D.x^2 + 09.x + 0E$

AES Round Overview

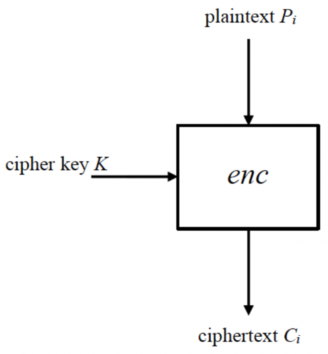
- AES Round.


A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11} A_{12} A_{13} A_{14} A_{15}
Byte Substitution
S
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14} B_{15}
ShiftRows
MixColumn
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9 C_{10} C_{11} C_{12} C_{13} C_{14} C_{15}
Key Addition
k_i

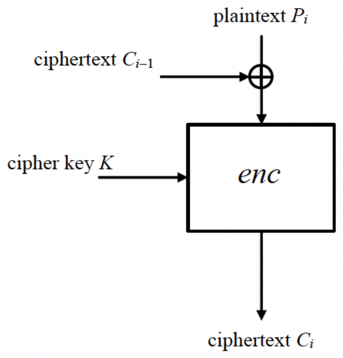
* Image source: https://tratliff.webspace.wheatoncollege.edu/2022_Fall/math202/index.html

Block Cipher/AES Modes

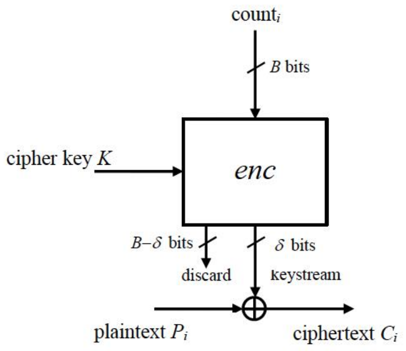
- In order to efficiently and securely use a block cipher, one must use the cipher in an appropriate mode of operation [H2020].
    - Electronic CodeBook Mode (ECB)
    - Cipher Block Chaining Mode (CBC)
    - Counter Mode (CTR)



(ECB)



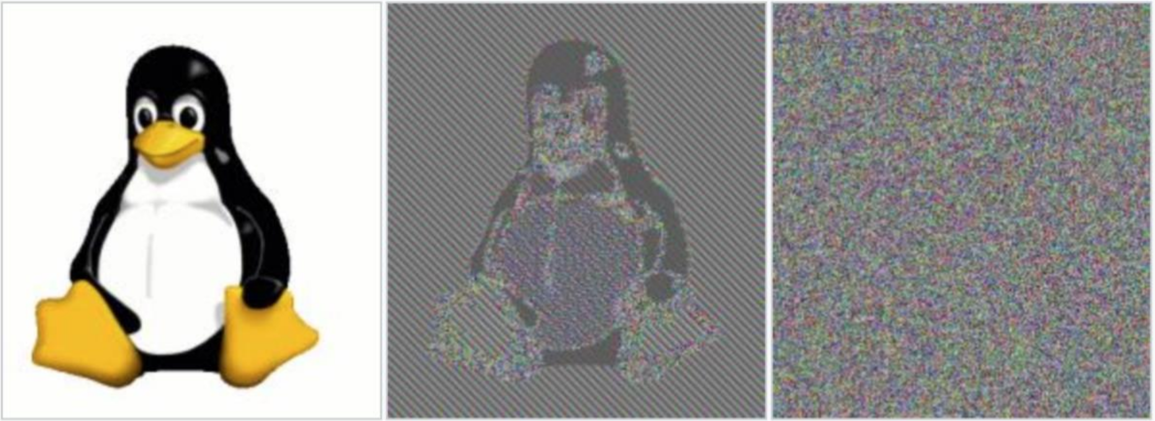
(CBC)



(CTR)

[H2020] H. M. Heys, *A Tutorial on the Implementation of Block Ciphers: Software and Hardware Applications*, 2020, IACR ePrint 2020/1545.

Block Cipher/AES Modes



Original image

Encrypted using ECB mode

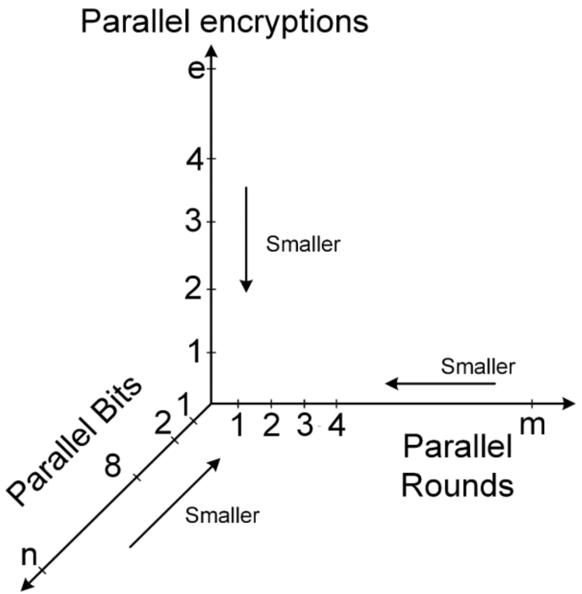
Modes other than ECB result in pseudo-randomness

Work Because Working From

* Image source: https://medium.com/@TalBeerySec/zooming-on-zoom-5-encryption-cc7e9b710b9f

AES Implementations

- Parallelism dimensions [AGS2014]

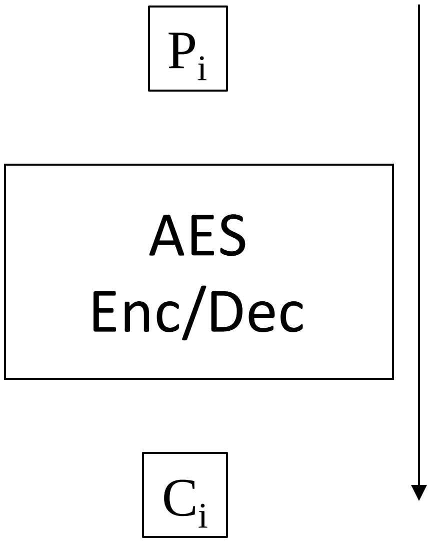


[AGS2014] A. Aysu *et al.*, *SIMON Says, Break the Area Records for Symmetric Key Block Ciphers on FPGAs*, ESL, 2014.

AES Implementations

- Efficiency parameters:

Latency



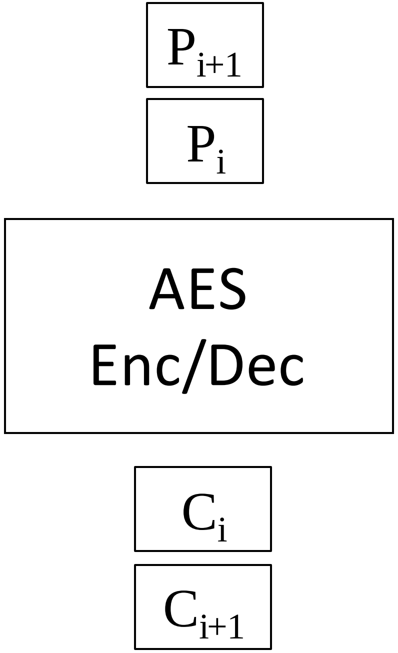
P_i

AES Enc/Dec

C_i

Time to encrypt/decrypt a single plaintext.

Throughput



P_{i+1}

P_i

AES Enc/Dec

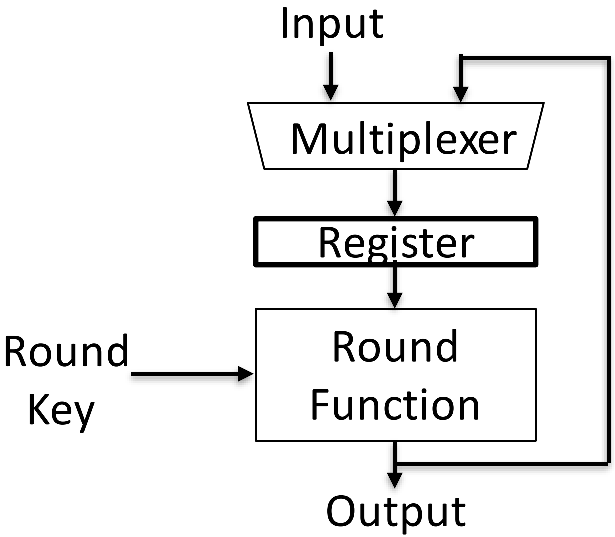
C_i

C_{i+1}

Number of plaintext encrypted/decrypted in a unit of time.

Block Cipher Implementations: Iterative Approach

- Implement the combinational logic required for one round (supplemented with register and multiplexers). Then, use it repeatedly.
  - Only one block of data is encrypted at a time.
  - The number of clock cycles necessary to encrypt a single block of data is equal to the number of cipher rounds.



Clock period (t_{clk}) = t

Latency ≈ t . (# of rounds)

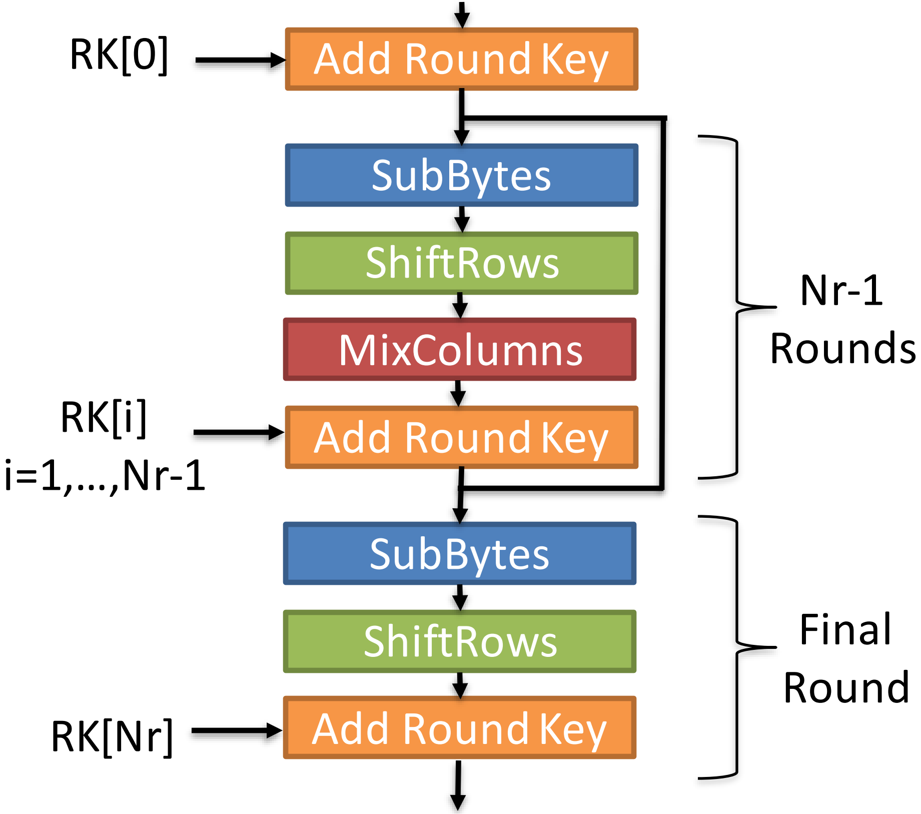
Throughput ≈ 1 / (t . (# of rounds))

AES Implementations: Iterative Approach

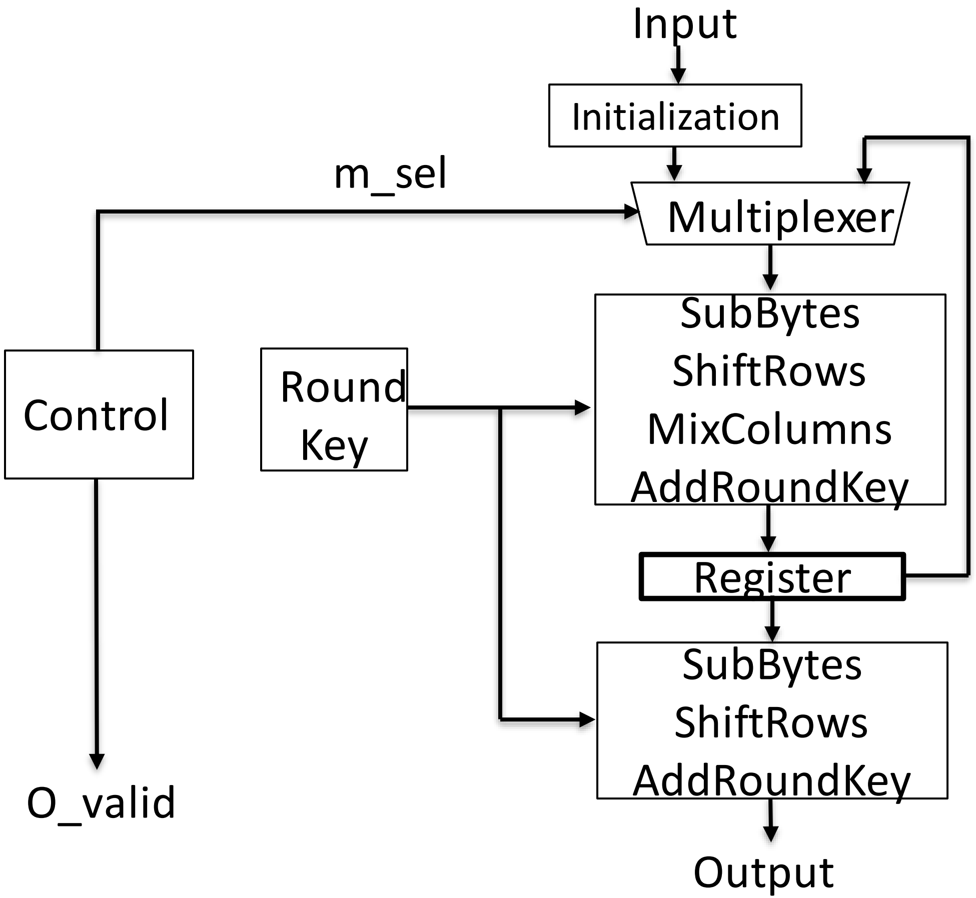
- Initialization

- Round (repeated Nr-1 times):
  - SubBytes
  - ShiftRows
  - MixColumns
  - AddRoundKey

- Final Round
  - SubBytes
  - ShiftRows
  - Add Round Key



AES Implementations: Iterative Approach



AES Implementations: Iterative Approach

- SubBytes is expensive and it is instantiated twice.
  - Can we do better?

AES Implementations: Iterative Approach

- SubBytes is expensive and it is instantiated twice.
  - Can we do better?

- See the order for a toy example: Nr = 3

AddRoundKey with ARK[0]
SubBytes
ShiftRows
MixColumns
AddRoundKey with ARK[1]
SubBytes
ShiftRows
MixColumns
AddRoundKey with ARK[2]
SubBytes
ShiftRows
AddRoundKey with ARK[3]

AES Implementations: Iterative Approach

- SubBytes is expensive and it is instantiated twice.
  - Can we do better?

- See the order for a toy example: Nr = 3

AddRoundKey with ARK[0]
SubBytes
ShiftRows
MixColumns
AddRoundKey with ARK[1]
SubBytes
ShiftRows
MixColumns
AddRoundKey with ARK[2]
SubBytes
ShiftRows
AddRoundKey with ARK[3]

AES Implementations: Iterative Approach

- SubBytes and AddRoundKey are instantiated twice.
  - Can we do better?

- See the order for a toy example: Nr = 3

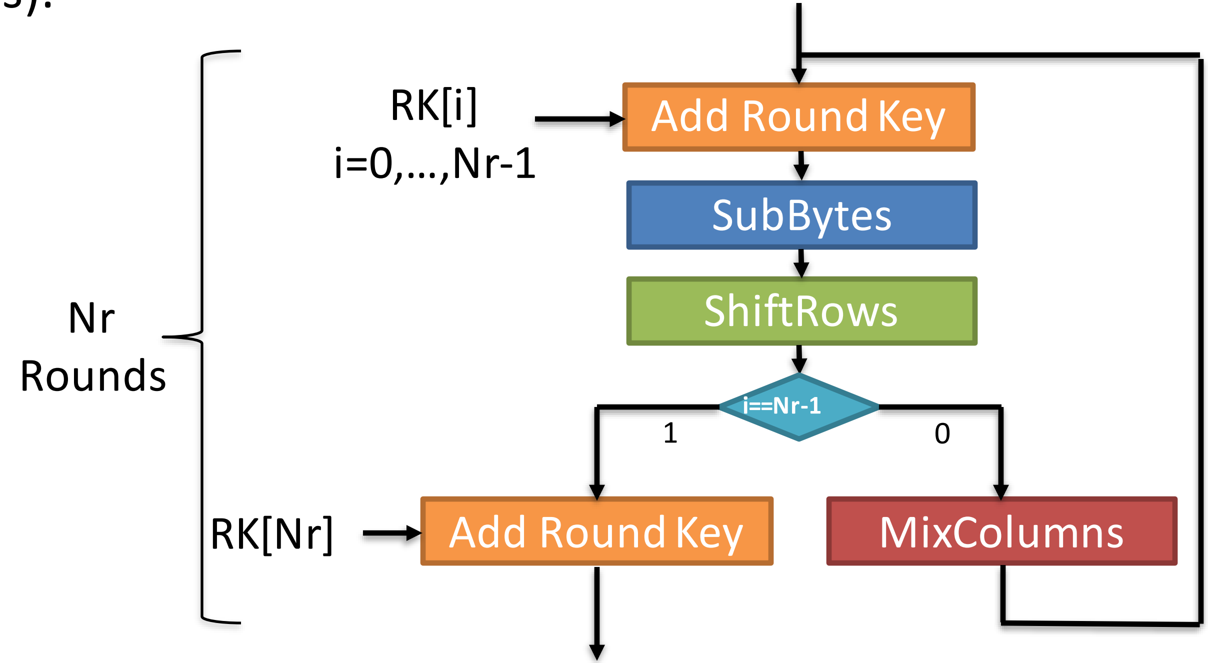
AddRoundKey with ARK[0]
SubBytes
ShiftRows
MixColumns
AddRoundKey with ARK[1]
SubBytes
ShiftRows
MixColumns
AddRoundKey with ARK[2]
SubBytes
ShiftRows
AddRoundKey with ARK[3]

AES Implementations: Iterative Approach

- Round (repeated Nr times):
  - AddRoundKey
  - SubBytes
  - ShiftRows
  - MixColumns

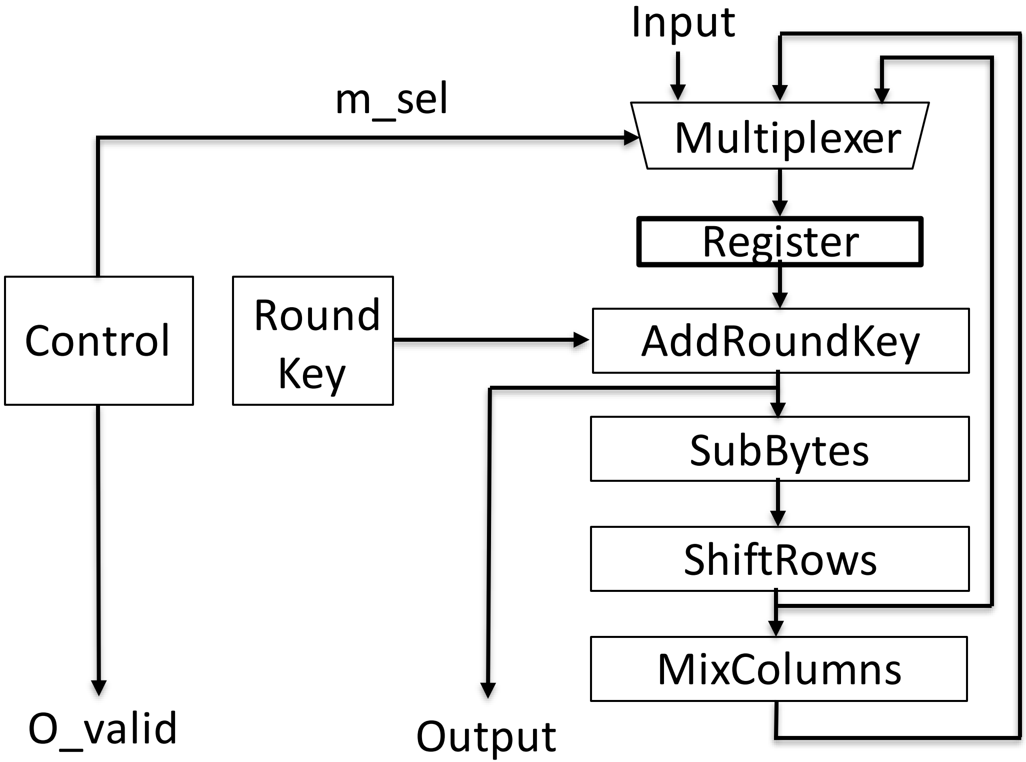
    or

    AddRoundKey



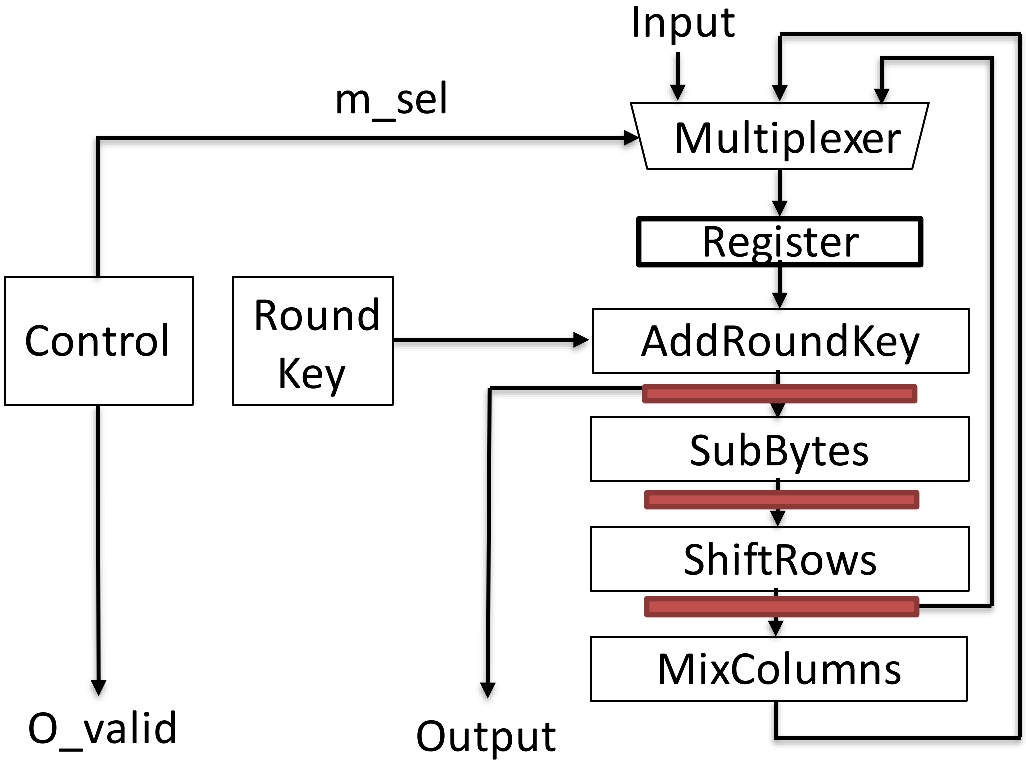
AES Implementations: Iterative Approach

- High-level diagram of the architecture



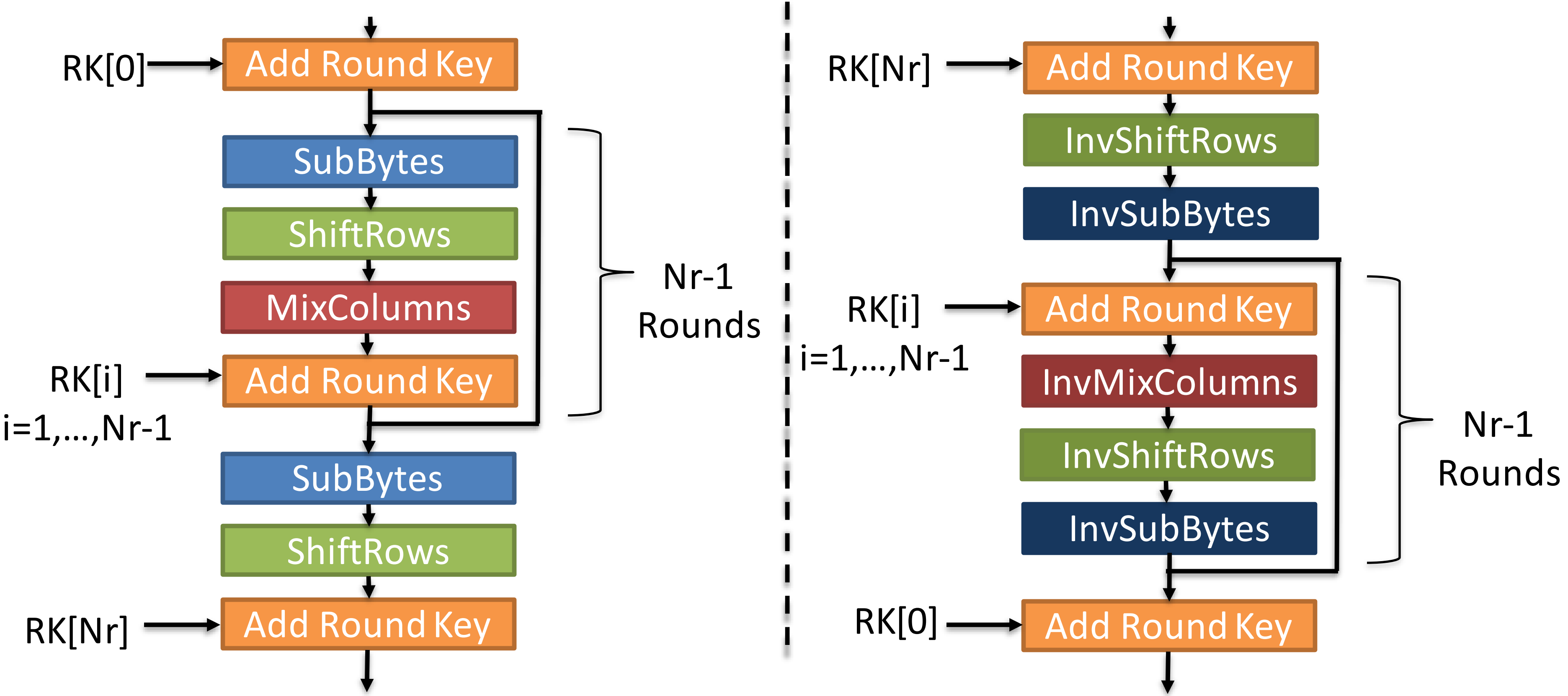
AES Implementations: Iterative Approach

- High-level diagram of the architecture
  - What happens if we divide a round into multiple stages?



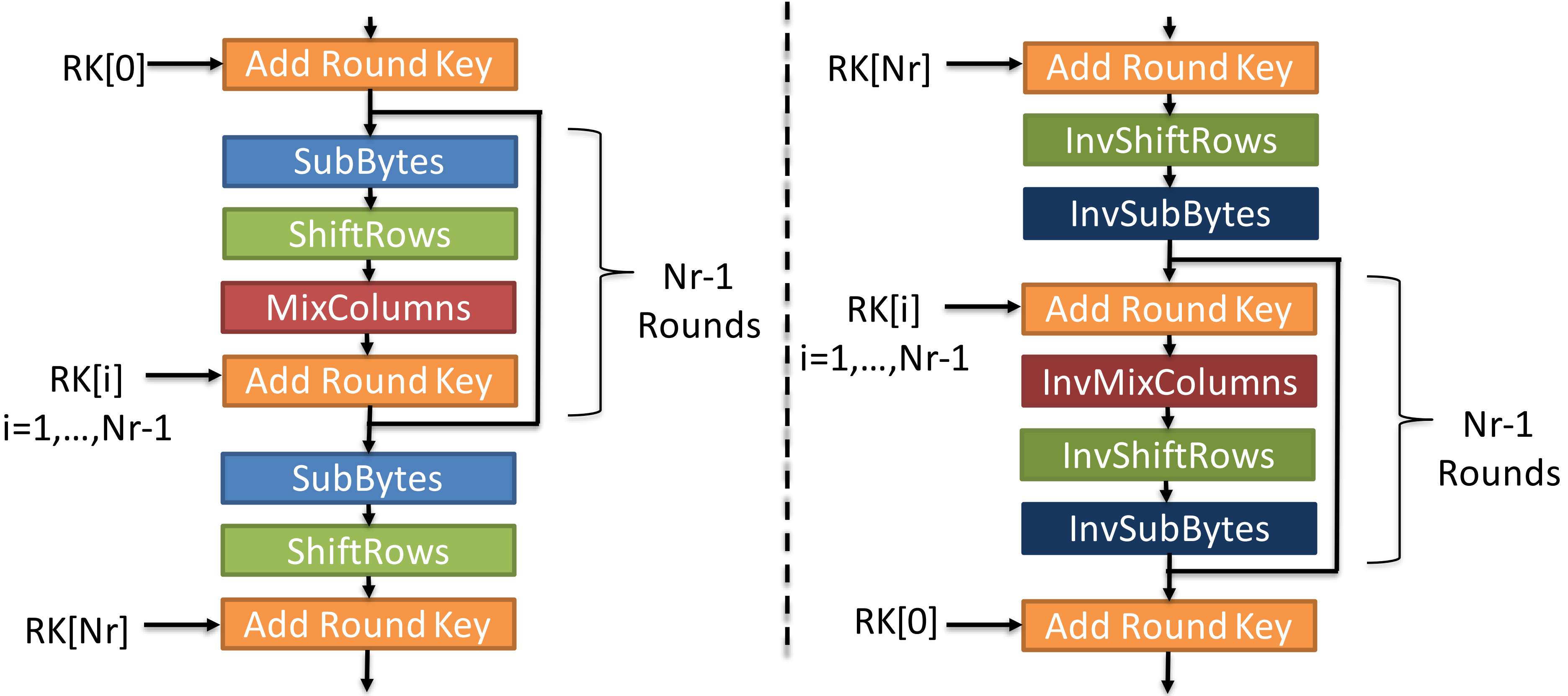
AES Implementations: Hardware

- What about decryption?



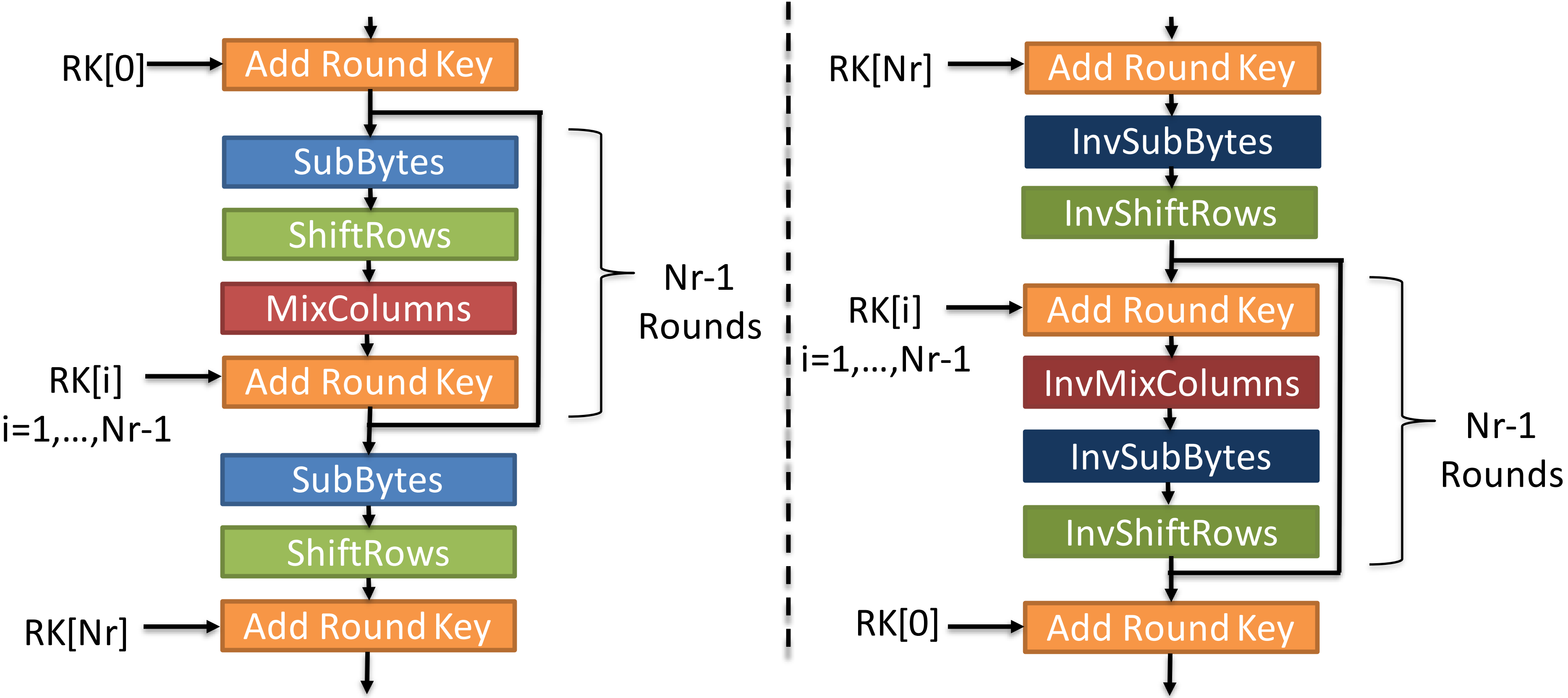
AES Implementations: Hardware

- Can we make Enc. and Dec. look similar?



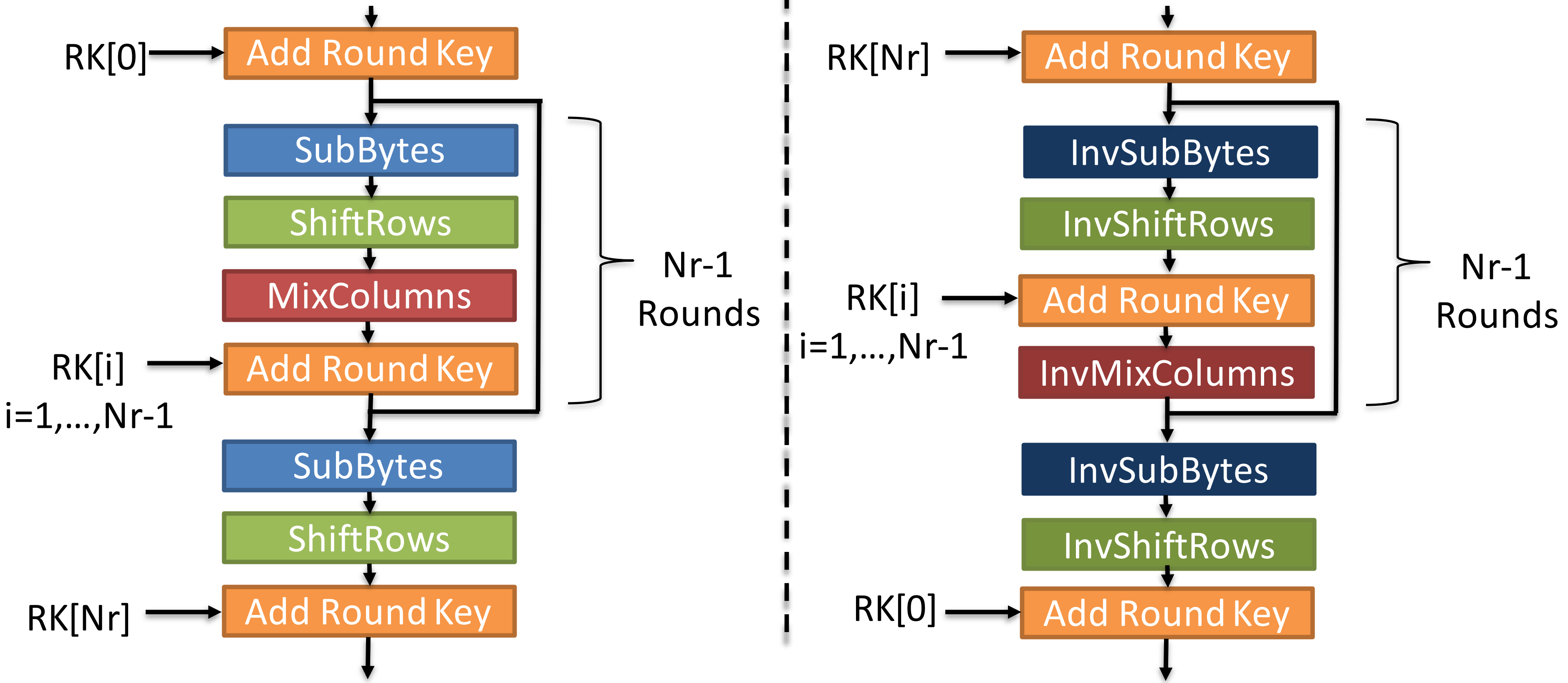
AES Implementations: Hardware

- Swap InvShiftRows and InvSubBytes



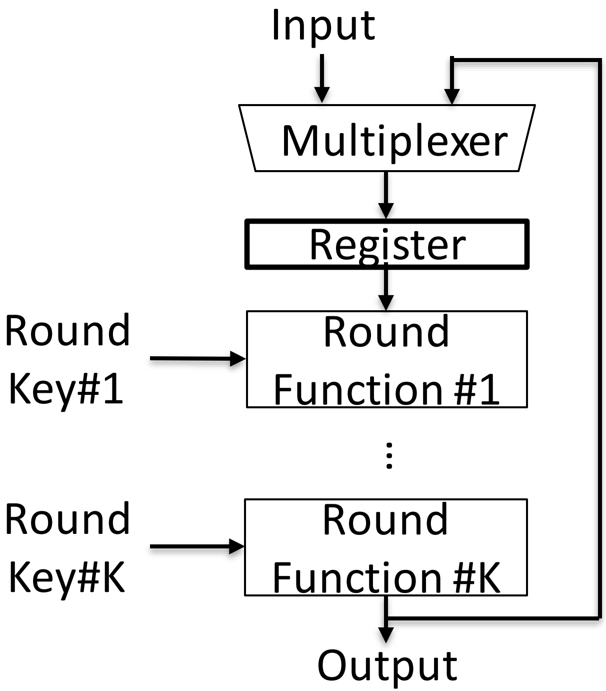
AES Implementations: Hardware

- Push InvShiftRows and InvSubBytes down



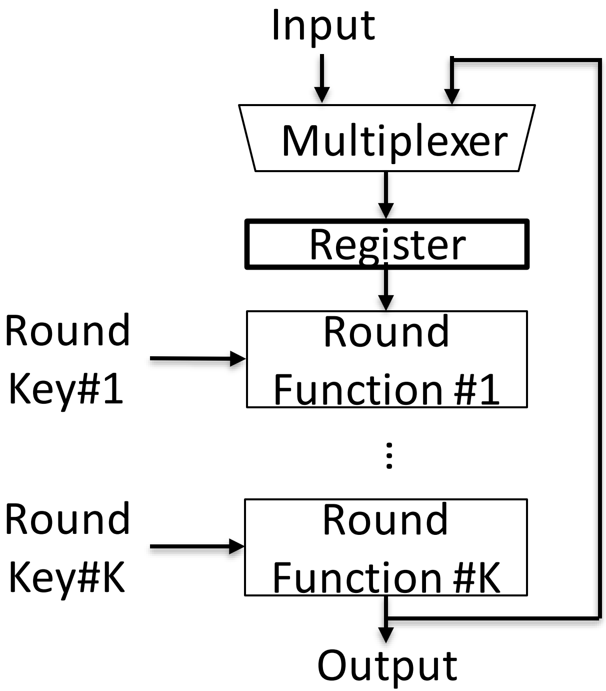
Block Cipher Implementations: Partial Loop Unrolling

- *K* round out of *Nr* round functions are implemented in combinational part.
  - Partial loop unrolling.



Block Cipher Implementations: Partial Loop Unrolling

- *K* round out of *Nr* round functions are implemented in combinational part.
    - Partial loop unrolling



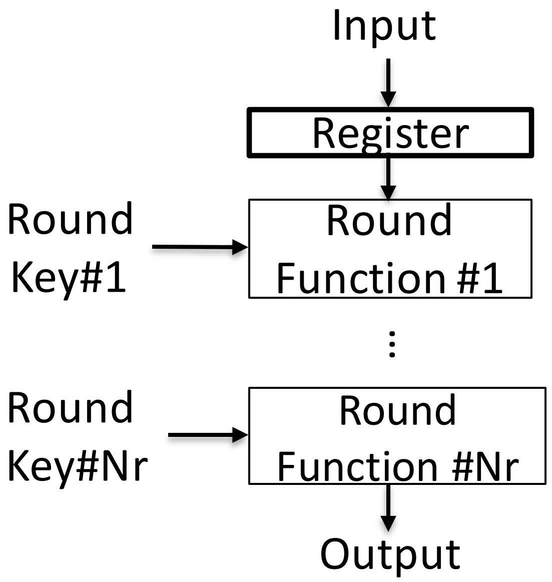
Clock period (t_{clk}) ≈ $K \, . \, t$

Latency ≈ $K \, . \, t$. [# of rounds/K]

Throughput ≈ 1 / ($K \, . \, t$. [# of rounds/K])

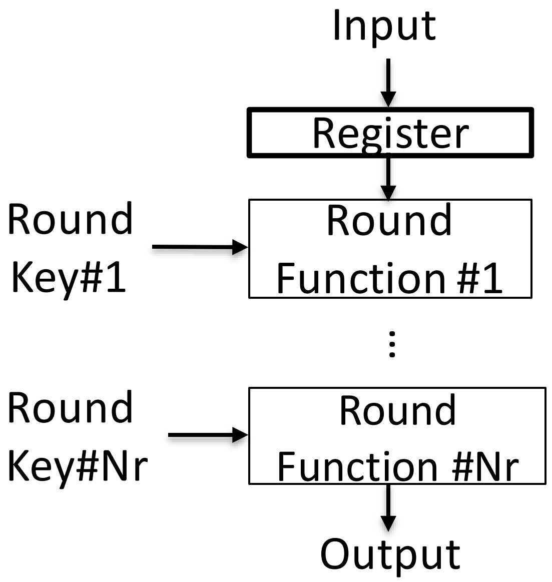
Block Cipher Implementations: Loop Unrolling

- All round functions are implemented in combinational part.
  - Full loop unrolling



Block Cipher Implementations: Loop Unrolling

- All round functions are implemented in combinational part
  - Full loop unrolling



Clock period (t_{clk}) ≈ (# of rounds) . *t*

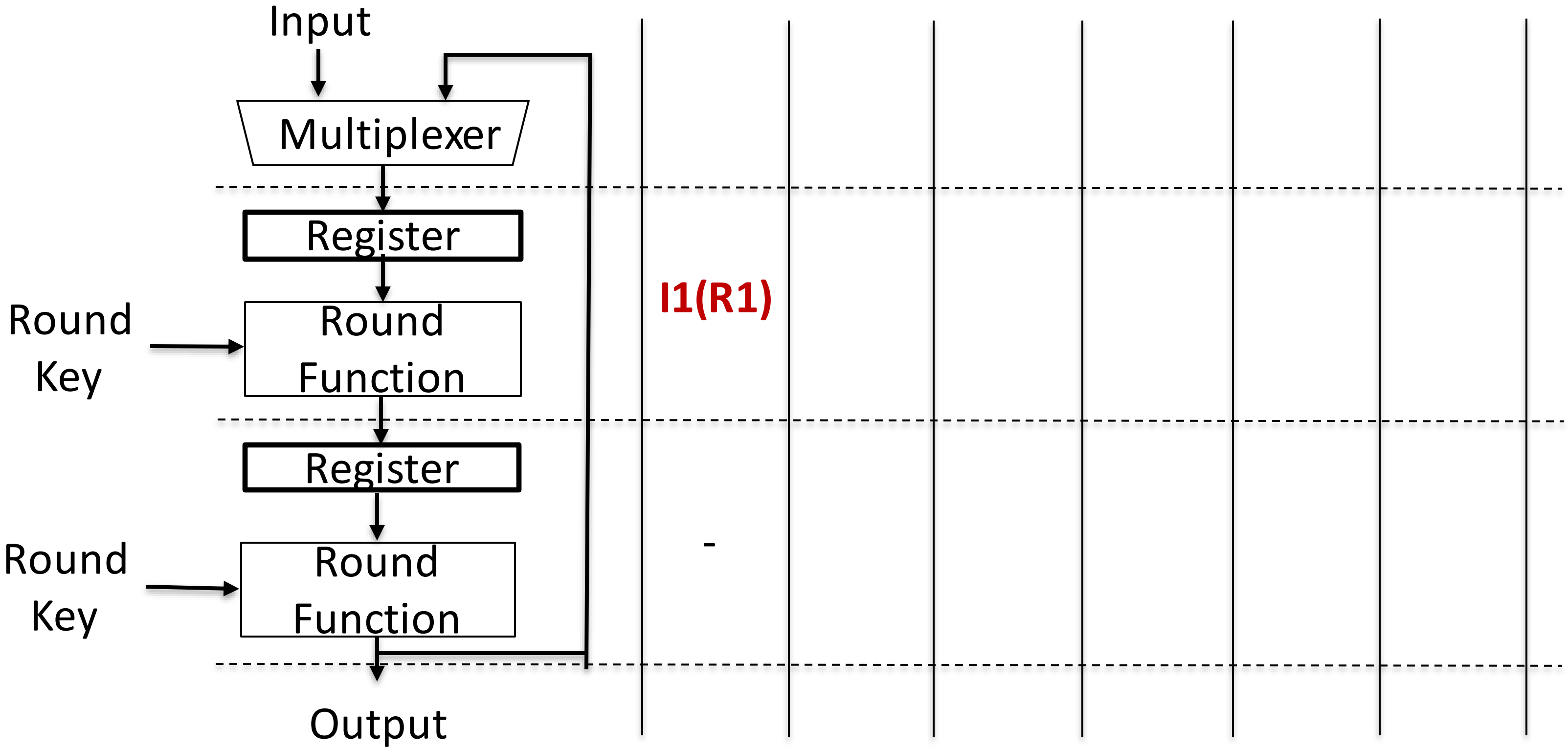
Latency ≈ *t* . (# of rounds)

Throughput ≈ 1 / (*t* . (# of rounds))

- Without pipelining, unrolling offers no throughput improvement.

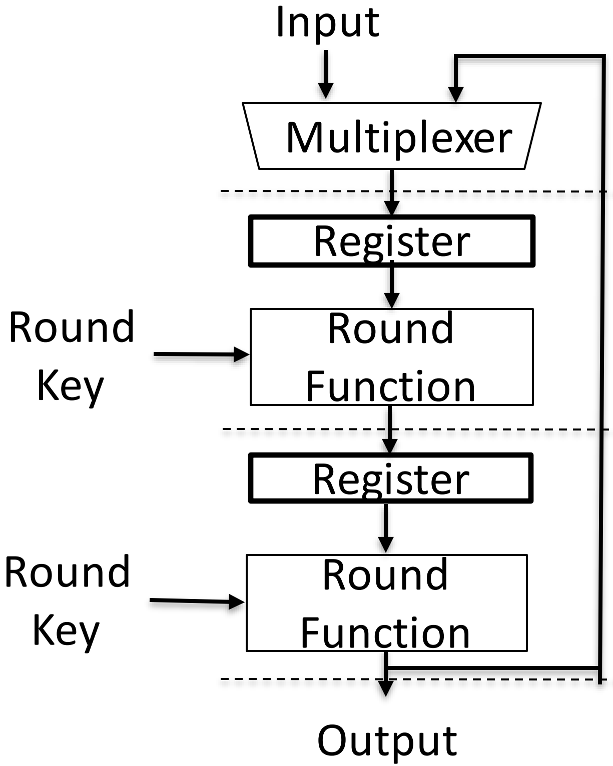
Block Cipher Implementations: Pipelining

- A traditional methodology for design of high-performance implementations.
  - Partial or full outer-loop pipelining (i.e., *K*=2 with *Nr*=4 rounds)



Block Cipher Implementations: Pipelining

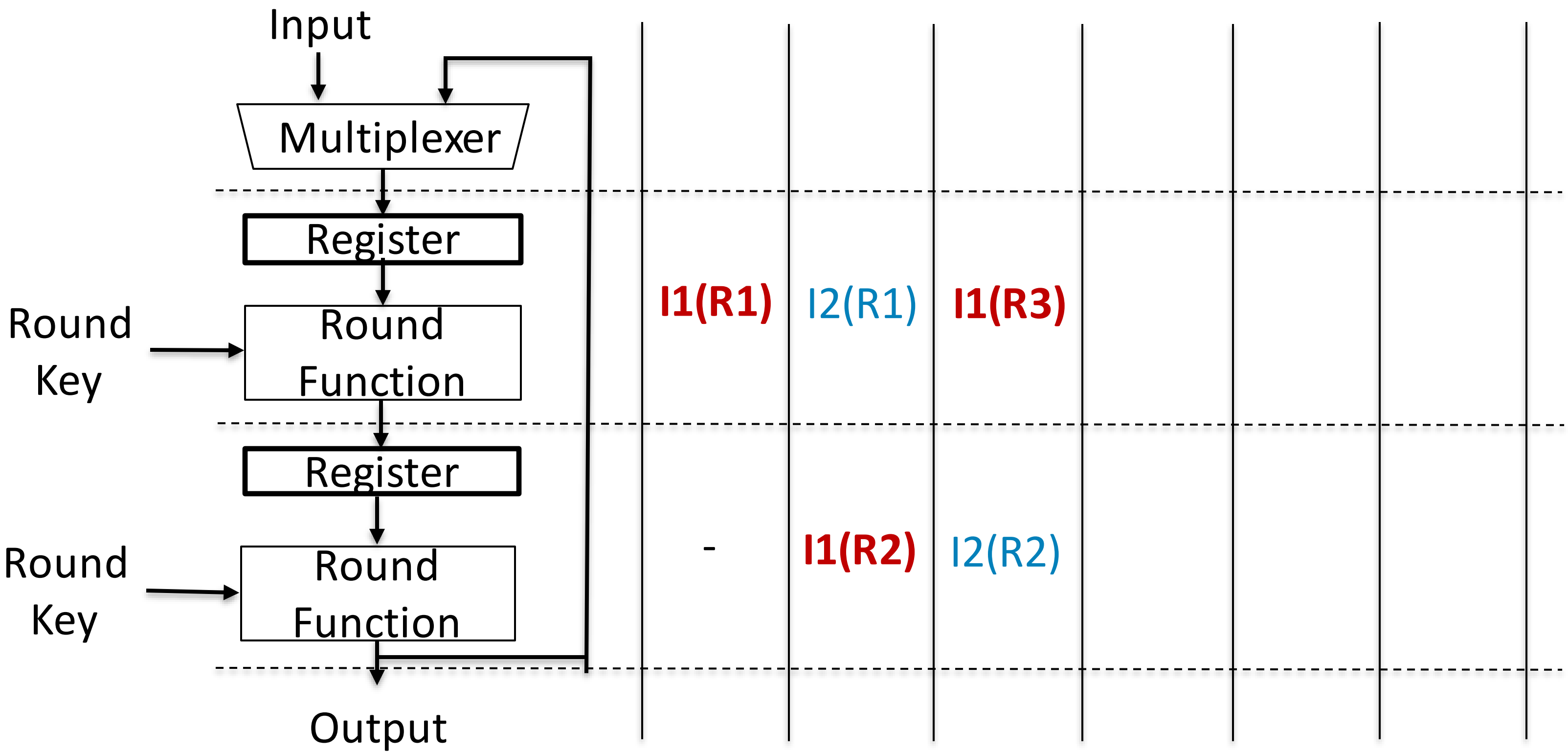
- A traditional methodology for design of high-performance implementations.
  - Partial or full outer-loop pipelining (i.e., *K*=2 with *Nr*=4 rounds)



I1(R1)	I2(R1)				
-	I1(R2)				

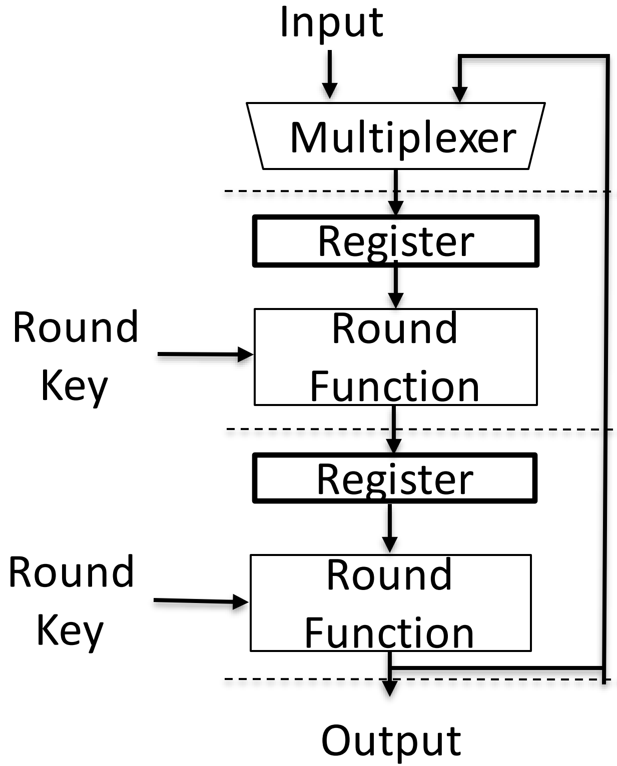
Block Cipher Implementations: Pipelining

- A traditional methodology for design of high-performance implementations.
  - Partial or full outer-loop pipelining (i.e., *K*=2 with *Nr*=4 rounds)



Block Cipher Implementations: Pipelining

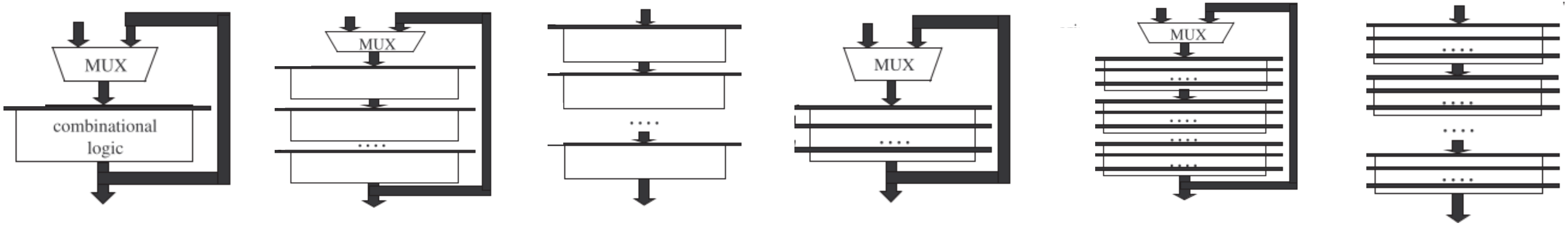
- A traditional methodology for design of high-performance implementations.
  - Partial or full outer-loop pipelining (i.e., *K*=2 with *Nr*=4 rounds)



I1(R1)	I2(R1)	**I1(R3)**	I2(R3)	I3(R1)	…
-	**I1(R2)**	I2(R2)	**I1(R4)** **(out)**	I2(R4) (out)	I3(R2)

Block Cipher Implementations: Pipelining

- A traditional methodology for design of high-performance implementations.
  - Partial or full outer-loop pipelining.
  - Inner-loop pipelining.
  - Partial or full outer-loop pipelining with inner loop pipelining.



Iterative

Partial unroll

Fully unroll

Iterative with inner pipeline

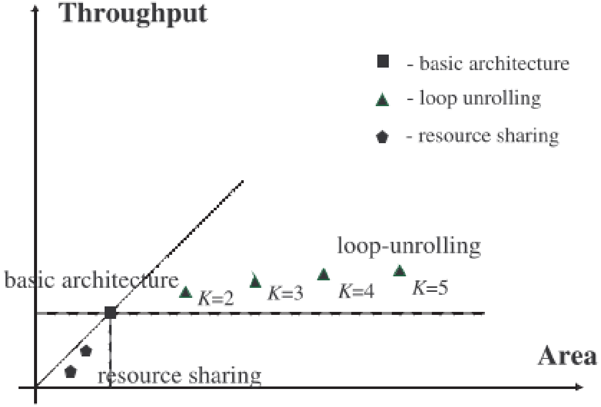
Partial unroll with inner-outer pipeline

Fully unroll with inner-outer pipeline

[GC2009] K. Gaj, *FPGA and ASIC Implementations of AES*, Cryptographic Engineering, 2009.

Block Cipher Implementations: Summary

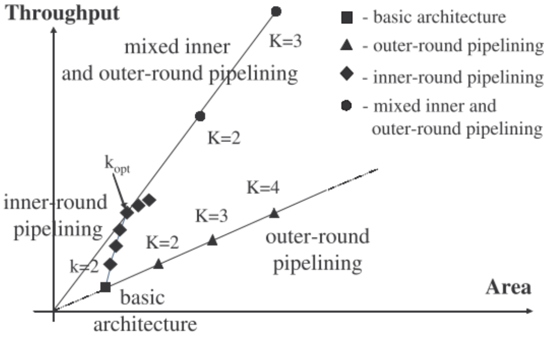
- Summary of implementation methods
  - Iterative
  - Partial unroll
  - Fully unroll

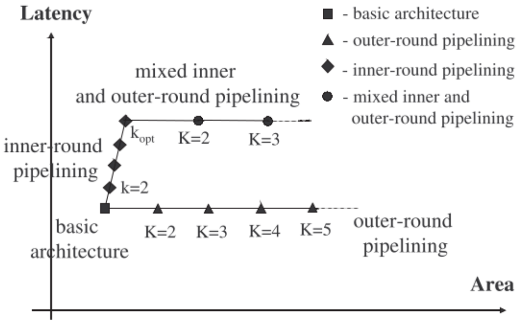


[GC2009] K. Gaj, *FPGA and ASIC Implementations of AES*, Cryptographic Engineering, 2009.

Block Cipher Implementations: Summary

- Summary of implementation methods
  - Iterative
  - Partial unroll
  - Fully unroll
  - Pipelining
    - Inner
    - Outer

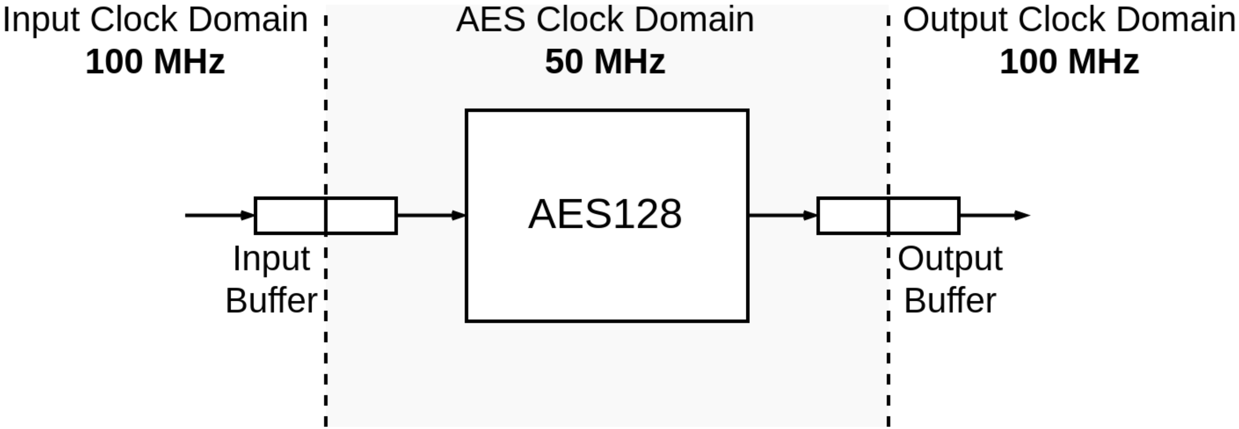




[GC2009] K. Gaj, *FPGA and ASIC Implementations of AES*, Cryptographic Engineering, 2009.

AES Implementations: I/O

- Assume that the input data rate is 100 Mb/sec (1Mb = 1,000,000 bits), the input and output buffers can store 128-bits each.
  - What would be your design strategy?



AES Implementations: SubBytes/S-box Implementation

- It takes one byte as input and produces one byte output. It has two components:
  - Multiplicative inverse in GF(2^8)
    - Complex operation
  - Affine transformation

- Three different approaches for implementation:
  - Look-up table
  - Look-up table + logic
  - Logic-only

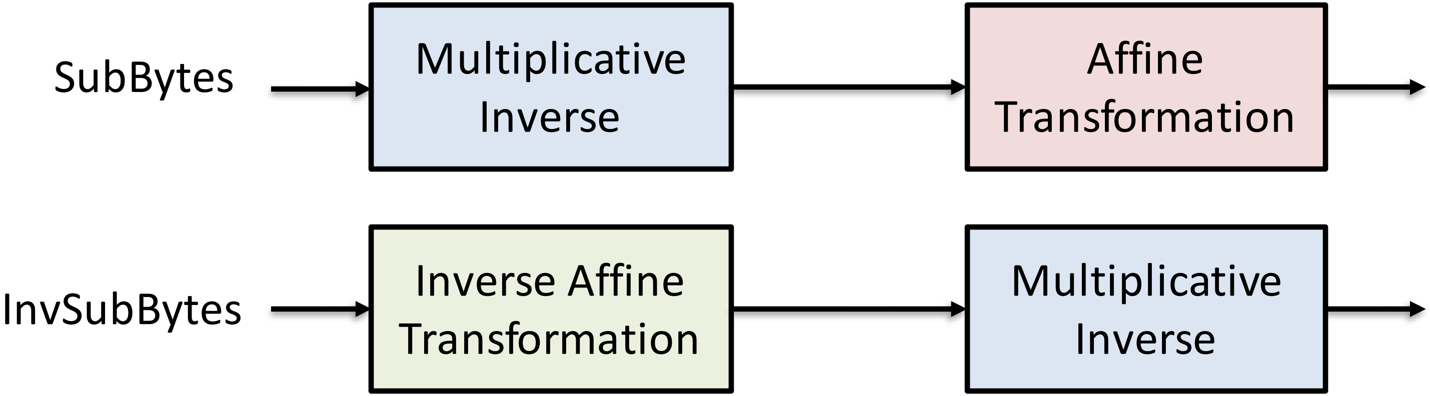
AES Implementations: SubBytes/S-box Implementation

- Look-up table:
  - Pre-compute and store SubBytes results for all possible inputs (0 to 255).
  - Each round state has 16 bytes, so 16 256x8 bits (2 Kbits) table is required.

- For a merged enc/dec design, table size is doubled.
  - i.e., use most significant bit of table address to distinguish forward and inverse conversions.

AES Implementations: SubBytes/S-box Implementation

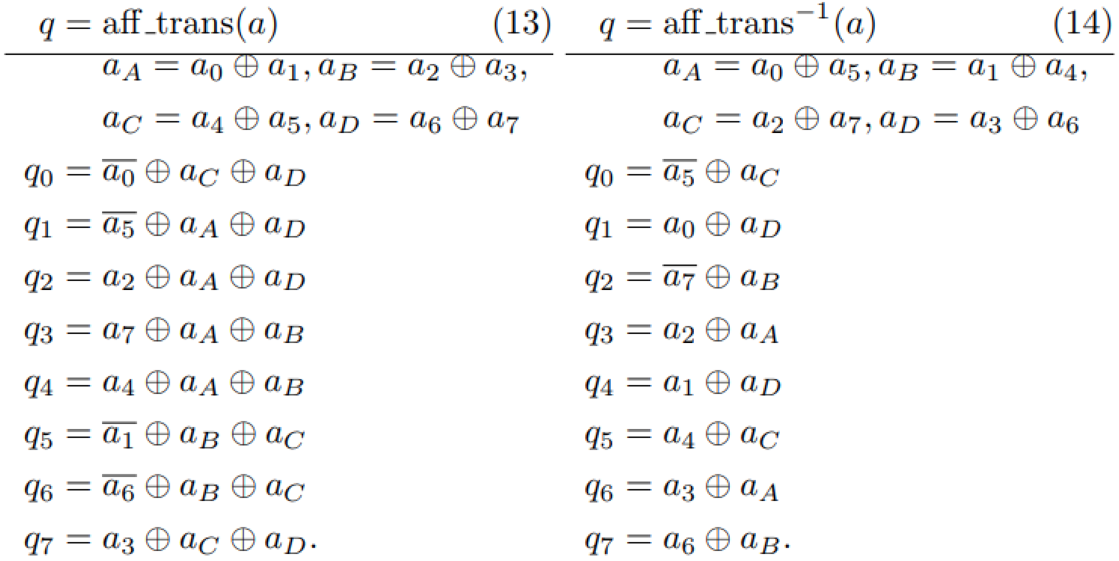
- Look-up table and logic:
  - InvSubBytes and SubBytes operations can share the same table.



  - Then, affine and inverse affine transformation operations can be implemented using XOR gates.

AES Implementations: SubBytes/S-box Implementation

- Look-up table and logic:
  - InvSubBytes and SubBytes operations can share the same table.



$$q = \mathrm{aff\_trans}(a) \tag{13}$$

$$\begin{aligned}
&a_A = a_0 \oplus a_1, a_B = a_2 \oplus a_3,\\
&a_C = a_4 \oplus a_5, a_D = a_6 \oplus a_7\\
q_0 &= \overline{a_0} \oplus a_C \oplus a_D\\
q_1 &= \overline{a_5} \oplus a_A \oplus a_D\\
q_2 &= a_2 \oplus a_A \oplus a_D\\
q_3 &= a_7 \oplus a_A \oplus a_B\\
q_4 &= a_4 \oplus a_A \oplus a_B\\
q_5 &= \overline{a_1} \oplus a_B \oplus a_C\\
q_6 &= \overline{a_6} \oplus a_B \oplus a_C\\
q_7 &= a_3 \oplus a_C \oplus a_D.
\end{aligned}$$

$$q = \mathrm{aff\_trans}^{-1}(a) \tag{14}$$

$$\begin{aligned}
&a_A = a_0 \oplus a_5, a_B = a_1 \oplus a_4,\\
&a_C = a_2 \oplus a_7, a_D = a_3 \oplus a_6\\
q_0 &= \overline{a_5} \oplus a_C\\
q_1 &= a_0 \oplus a_D\\
q_2 &= \overline{a_7} \oplus a_B\\
q_3 &= a_2 \oplus a_A\\
q_4 &= a_1 \oplus a_D\\
q_5 &= a_4 \oplus a_C\\
q_6 &= a_3 \oplus a_A\\
q_7 &= a_6 \oplus a_B.
\end{aligned}$$

[WOL2002] J. Wolkerstorfer *et al.*, *An ASIC Implementation of AES SBoxes*, CT-RSA, 2002.

AES Implementations: SubBytes/S-box Implementation

- Logic:
  - Table-based implementations can be costly for ASIC.
  - It also can limit maximum clock frequency in deeply-pipelined architectures.

- What are our options?
  - Construct truth-table and derive Boolean expression.
    - Very inefficient even with Boolean minimization techniques.

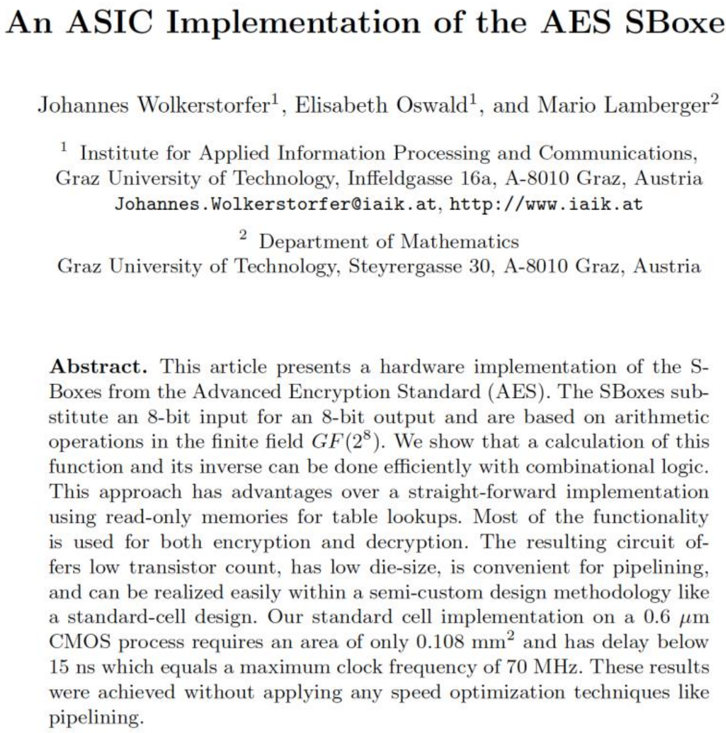
AES Implementations: SubBytes/S-box Implementation

- Logic:
  - Table-based implementations can be costly for ASIC.
  - It also can limit maximum clock frequency in deeply-pipelined architectures.

- What are our options?
  - Construct truth-table and derive Boolean expression.
    - Very inefficient even with Boolean minimization techniques.
  - Implement multiplicative inverse operation for Rijndael's finite field.
    - Extended Euclidean Algorithm
    - Map operations to GF(2^4)

AES Implementations: SubBytes/S-box Implementation

- Map operations to GF(2^4). [WOL2002]



An ASIC Implementation of the AES SBoxes*

Johannes Wolkerstorfer[1], Elisabeth Oswald[1], and Mario Lamberger[2]

[1] Institute for Applied Information Processing and Communications,
Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria
Johannes.Wolkerstorfer@iaik.at, http://www.iaik.at

[2] Department of Mathematics
Graz University of Technology, Steyrergasse 30, A-8010 Graz, Austria

Abstract. This article presents a hardware implementation of the S-Boxes from the Advanced Encryption Standard (AES). The SBoxes substitute an 8-bit input for an 8-bit output and are based on arithmetic operations in the finite field $GF(2^8)$. We show that a calculation of this function and its inverse can be done efficiently with combinational logic. This approach has advantages over a straight-forward implementation using read-only memories for table lookups. Most of the functionality is used for both encryption and decryption. The resulting circuit offers low transistor count, has low die-size, is convenient for pipelining, and can be realized easily within a semi-custom design methodology like a standard-cell design. Our standard cell implementation on a 0.6 μm CMOS process requires an area of only 0.108 mm^2 and has delay below 15 ns which equals a maximum clock frequency of 70 MHz. These results were achieved without applying any speed optimization techniques like pipelining.

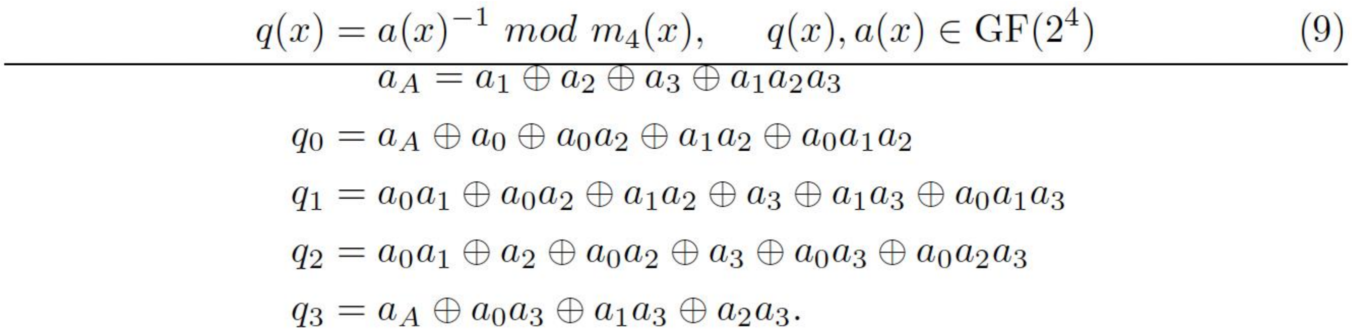
[WOL2002] J. Wolkerstorfer *et al.*, *An ASIC Implementation of AES SBoxes*, CT-RSA, 2002.

AES Implementations: SubBytes/S-box Implementation

- Map operations to GF(2^4). [WOL2002]
  - We can represent an element in GF(2^8) as a two-term polynomial with coefficients in GF(2^4).

$$a = a_h . x + al, \quad a \in \mathrm{GF}(2^8) \text{ and } a_h, al \ \in \mathrm{GF}(2^4)$$

- Example: Inversion in GF(2^4)



$$q(x) = a(x)^{-1}\ mod\ m_4(x), \qquad q(x), a(x) \in \mathrm{GF}(2^4) \qquad (9)$$

$$\begin{aligned}
a_A &= a_1 \oplus a_2 \oplus a_3 \oplus a_1 a_2 a_3 \\
q_0 &= a_A \oplus a_0 \oplus a_0 a_2 \oplus a_1 a_2 \oplus a_0 a_1 a_2 \\
q_1 &= a_0 a_1 \oplus a_0 a_2 \oplus a_1 a_2 \oplus a_3 \oplus a_1 a_3 \oplus a_0 a_1 a_3 \\
q_2 &= a_0 a_1 \oplus a_2 \oplus a_0 a_2 \oplus a_3 \oplus a_0 a_3 \oplus a_0 a_2 a_3 \\
q_3 &= a_A \oplus a_0 a_3 \oplus a_1 a_3 \oplus a_2 a_3 .
\end{aligned}$$

* Equations are retrieved from [WOL2002]

AES Implementations: SubBytes/S-box Implementation

- Inversion in GF(2^8) can be re-written in terms of addition, multiplication, squaring and inversion in GF(2^4):

$$(a_h.x \; + \; al) \otimes (a'_h.x \; + a'_l) = \; 0.x \; + \; 1, \quad a_h, a_l, a'_h, a'_l \in \mathrm{GF}(2^4)$$

- Then, inversion can be derived as:

$$(a_h.x \; + \; al)^{-1} = (a'_h.x \; + a'_l) = (a_h \otimes d).x + (a_h \oplus al) \otimes d$$

where

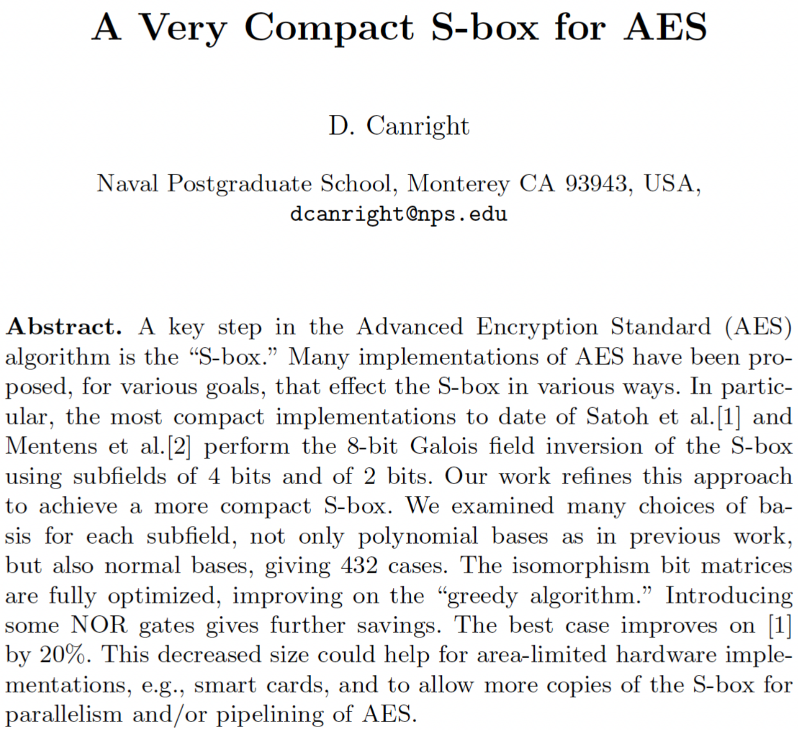
$$d = \left((a_h^2 \otimes E) \oplus (a_h \otimes a_l) \oplus a_l^2\right)^{-1}$$

- All operations for calculating d is in GF(2^4).

* Equations are retrieved from [WOL2002]

AES Implementations: S-box Implementation

- We can further map operations in GF(2^4) to GF(2). [C2005]



A Very Compact S-box for AES

D. Canright

Naval Postgraduate School, Monterey CA 93943, USA,
dcanright@nps.edu

Abstract. A key step in the Advanced Encryption Standard (AES) algorithm is the "S-box." Many implementations of AES have been proposed, for various goals, that effect the S-box in various ways. In particular, the most compact implementations to date of Satoh et al.[1] and Mentens et al.[2] perform the 8-bit Galois field inversion of the S-box using subfields of 4 bits and of 2 bits. Our work refines this approach to achieve a more compact S-box. We examined many choices of basis for each subfield, not only polynomial bases as in previous work, but also normal bases, giving 432 cases. The isomorphism bit matrices are fully optimized, improving on the "greedy algorithm." Introducing some NOR gates gives further savings. The best case improves on [1] by 20%. This decreased size could help for area-limited hardware implementations, e.g., smart cards, and to allow more copies of the S-box for parallelism and/or pipelining of AES.

[C2005] D. Canright, *A Very Compact S-Box for AES*, CHES, 2005.

AES Implementations: MixColumn

- The MixColumn Layer
  - It can be expressed as matrix multiplication
  - Each element of the matrix is a byte

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

- Each polynomial coefficient will be multiplied with a matrix element. Then, the resulting four bytes will be added (XORed)
  - 2 layers of XOR gates: 3-input XOR gates + 4-input XOR gates
  - i.e., $b_0 = 2 \otimes a_0 \oplus 3 \otimes a_1 \oplus 1 \otimes a_2 \oplus 1 \otimes a_3$

- Each coefficient multiplication can also be implemented using look-up tables

AES Implementations: MixColumn

- The Inverse MixColumn Layer
  - It can be expressed as matrix multiplication
  - Each element of the matrix is a byte

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

- Inverse MixColumn has larger coefficients.
  - 2 layers of XOR gates: **up to 6-input XOR gates** + 4-input XOR gates

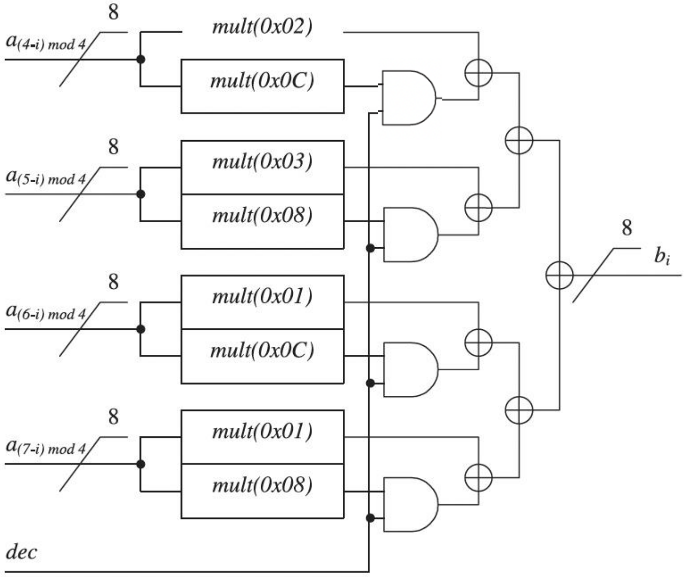
- Inverse MixColumn implementation will have larger area and longer critical path.

AES Implementations: MixColumn

- Since the hardware implementing inverse MixColumn layer is always larger, there are works targeting resource sharing between MixColumn and inverse MixColumn for reducing hardware cost. [W2001]

- Inverse matrix can be expressed as: [GC2009]

$$\begin{bmatrix} 0C & 08 & 0C & 08 \\ 08 & 0C & 08 & 0C \\ 0C & 08 & 0C & 08 \\ 08 & 0C & 08 & 0C \end{bmatrix} + \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$



[W2001] J. Wolkerstorfer. *An ASIC implementation of the AES MixColumn operation*. In Proc. Austrochip 2001

[GC2009] K. Gaj, *FPGA and ASIC Implementations of AES*, Cryptographic Engineering, 2009.

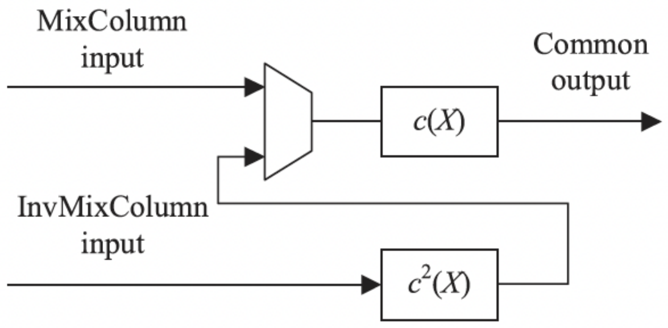
AES Implementations: MixColumn

- Since the hardware implementing inverse MixColumn layer is always larger, there are works targeting resource sharing between MixColumn and inverse MixColumn for reducing hardware cost. [W2001]

- Inverse matrix/polynomial $d(x)$ can be expressed as $c(x)^3$ where $c(x)$ is forward matrix/polynomial:

$$d(x) = c(x)\,.\,c(x)^2$$

- $c(x)^2$: $\begin{bmatrix} 05 & 00 & 04 & 00 \\ 00 & 05 & 00 & 04 \\ 04 & 00 & 05 & 00 \\ 00 & 04 & 00 & 05 \end{bmatrix}$



* Image resource: [GC2009]

References

[H2020] H. M. Heys, *A Tutorial on the Implementation of Block Ciphers: Software and Hardware Applications*, 2020, IACR ePrint 2020/1545.

[AGS2014] A. Aysu *et al.*, *SIMON Says, Break the Area Records for Symmetric Key Block Ciphers on FPGAs*, ESL, 2014.

[WOL2002] J. Wolkerstorfer *et al.*, *An ASIC Implementation of AES SBoxes*, CT-RSA, 2002.

[C2005] D. Canright, *A Very Compact S-Box for AES*, CHES, 2005.

[W2001] J. Wolkerstorfer. *An ASIC implementation of the AES MixColumn operation*. In Proc. Austrochip 2001

[GC2009] K. Gaj, *FPGA and ASIC Implementations of AES*, Cryptographic Engineering, 2009.