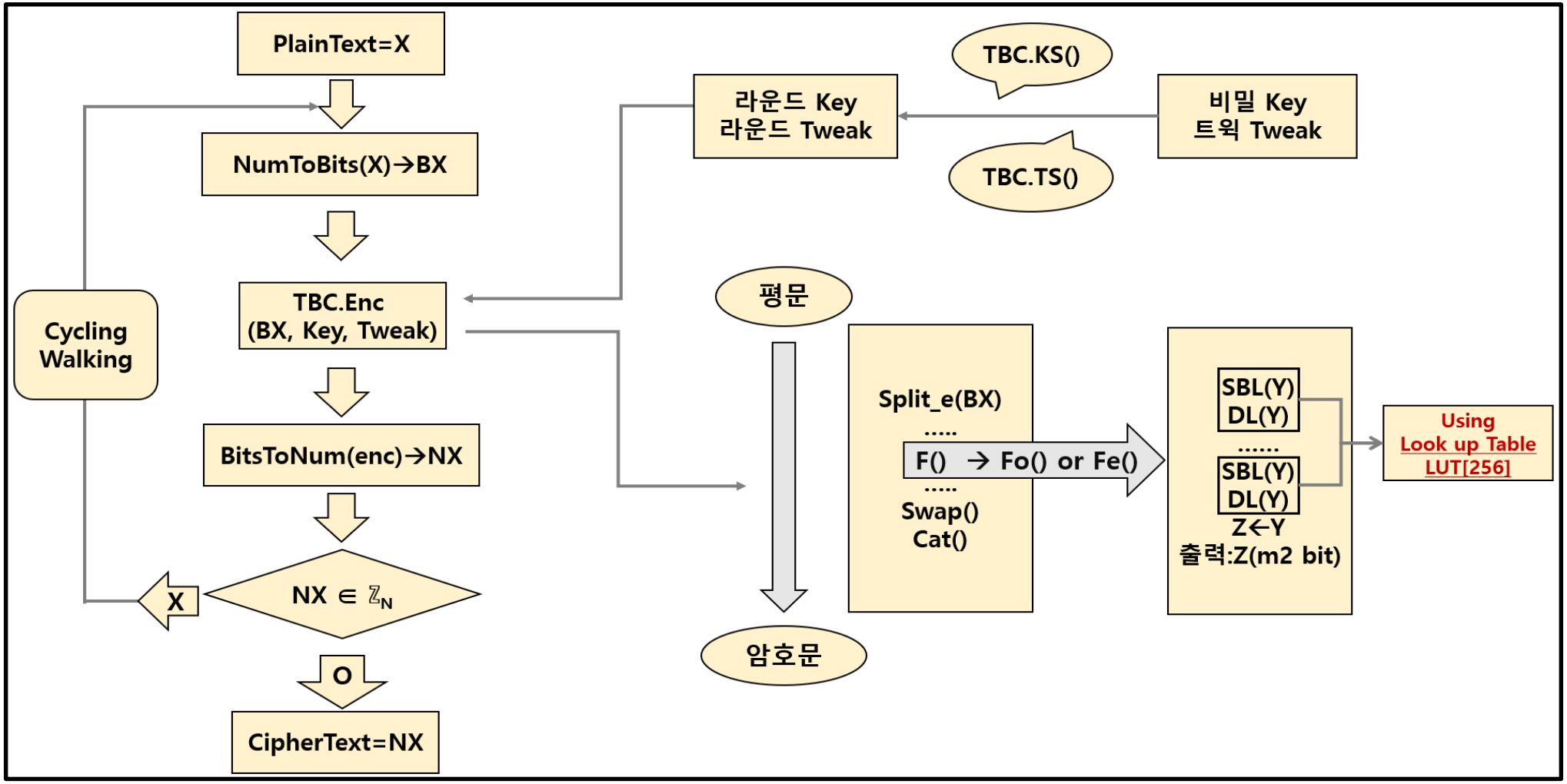
형태보존암호 최적화 구현 기법 제안

목차

- C언어로 구현한 형태보존암호
- 랭크 함수 최적화 기법
- SBL 및 DL 계층 최적화 기법

구현 알고리즘


PlainText=X
NumToBits(X)→BX
TBC.Enc
(BX, Key, Tweak)
BitsToNum(enc)→NX
NX ∈ ℤN
X
O
CipherText=NX
Cycling
Walking
라운드 Key
라운드 Tweak
TBC.KS()
TBC.TS()
비밀 Key
트윅 Tweak
평문
암호문
Split_e(BX)
.....
F() → Fo() or Fe()
.....
Swap()
Cat()
SBL(Y)
DL(Y)
.....
SBL(Y)
DL(Y)
Z←Y
출력:Z(m2 bit)
Using
Look up Table
LUT[256]

구현 결과 사진(10진수 평문 암복호화 결과) #1

1

```
Enter the number you want to encrypt
: 49241
{0xc0,0x59}
----------------printf Plain Text-----------------

Plain Text :

n = 64
c0  59
Number of digits : 5  Decimal digit

-----------------end Plain Text--------------------

--------------- print Encryption Text---------------

Encrypt Text :

n = 64
8c  5f
Number of digits : 5  Decimal digit

35935

Cycling walking 1 times

--------------- end Encryption Text---------------

--------------- print Decryption Text---------------

Decrption Text :

n = 64
c0  59
Number of digits : 5  Decimal digit

49241

--------------- end Decryption Text---------------
```

2

```
Enter the number you want to encrypt
: 495839645
{0x1d,0x8d,0xe9,0x9d}
----------------printf Plain Text-----------------

Plain Text :

n = 64
1d  8d  e9  9d
Number of digits : 9  Decimal digit

-----------------end Plain Text--------------------

--------------- print Encryption Text---------------

Encrypt Text :

n = 64
8  15  4c  b7
Number of digits : 9  Decimal digit

135613623

Cycling walking 1 times

--------------- end Encryption Text---------------

--------------- print Decryption Text---------------

Decrption Text :

n = 64
1d  8d  e9  9d
Number of digits : 9  Decimal digit

495839645


--------------- end Decryption Text---------------
```

구현 결과 사진(10진수 평문 암복호화 결과) #2

3

```
Enter the number you want to encrypt
: 94835284958432
{0x56,0x40,0x8f,0x78,0xdc,0xe0}
----------------printf Plain Text-----------------

Plain Text :

n = 64
56  40  8f  78  dc  e0
Number of digits : 14  Decimal digit

-----------------end Plain Text--------------------

--------------- print Encryption Text---------------

Encrypt Text :

n = 64
3c  28  53  6e  ef  77
Number of digits : 14  Decimal digit

66143896137591

Cycling walking 1 times

--------------- end Encryption Text---------------

--------------- print Decryption Text---------------

Decrption Text :

n = 64
56  40  8f  78  dc  e0
Number of digits : 14  Decimal digit

94835284958432


--------------- end Decryption Text---------------
```

4

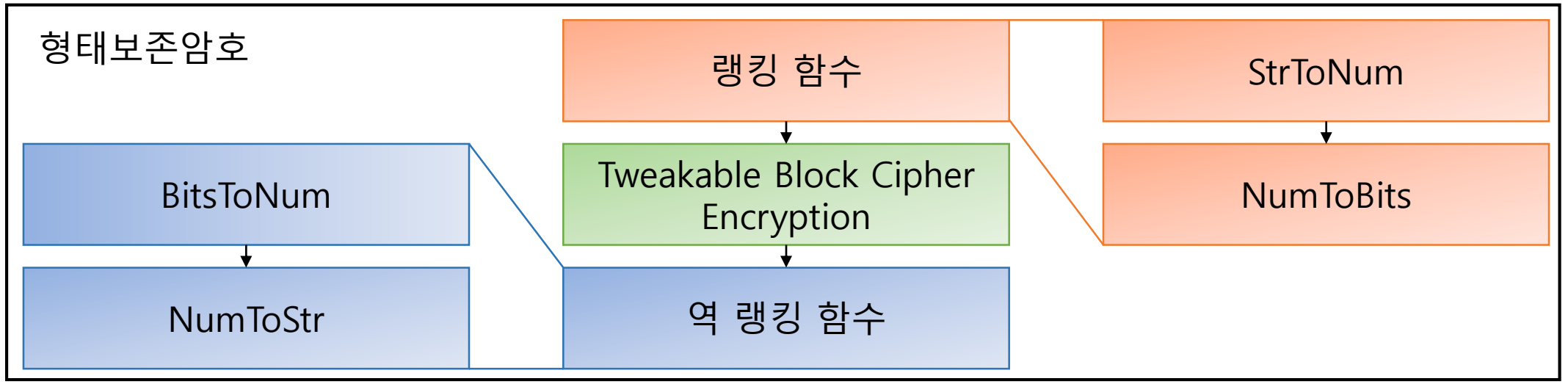
```
Enter the number you want to encrypt
: 12330594839428601230
{0xab,0x1f,0x12,0xad,0x19,0x68,0x15,0x8e}
----------------printf Plain Text-----------------

Plain Text :

n = 64
ab  1f  12  ad  19  68  15  8e
Number of digits : 20  Decimal digit

-----------------end Plain Text--------------------

--------------- print Encryption Text---------------

Encrypt Text :

n = 64
da  b5  22  b6  5  bd  c2  c5
Number of digits : 20  Decimal digit

157595406362286332 85

Cycling walking 5 times

--------------- end Encryption Text---------------

--------------- print Decryption Text---------------

Decrption Text :

n = 64
ab  1f  12  ad  19  68  15  8e
Number of digits : 20  Decimal digit

12330594839428601230


--------------- end Decryption Text---------------
```

랭크 함수 최적화 기법

- 랭크 함수 소개
- 랭크 함수 최적화

랭크 함수란?

- 임의의 집합에서 정의되는 일대일 함수
- 역 랭킹 함수는 랭킹 함수의 역 사상 (Inverse Function)으로 정의



StrToNum 그리고 NumToBits

- $StrToNum_b^m(a) = \sum_{i=0}^{m-1} a^i b^i$
  - 예시: 7 자리 26 진수 $a = [12\ 0\ 17\ 8\ 12\ 1\ 0]_{26}$
  - $StrToNum_{26}^7(a)$
    $= 12 \cdot 26^6 + 0 \cdot 26^5 + 17 \cdot 26^4 + 8 \cdot 26^3 + 12 \cdot 26^2 + 1 \cdot 26^1 + 0 \cdot 26^0$
    $= 3{,}714{,}906{,}650$

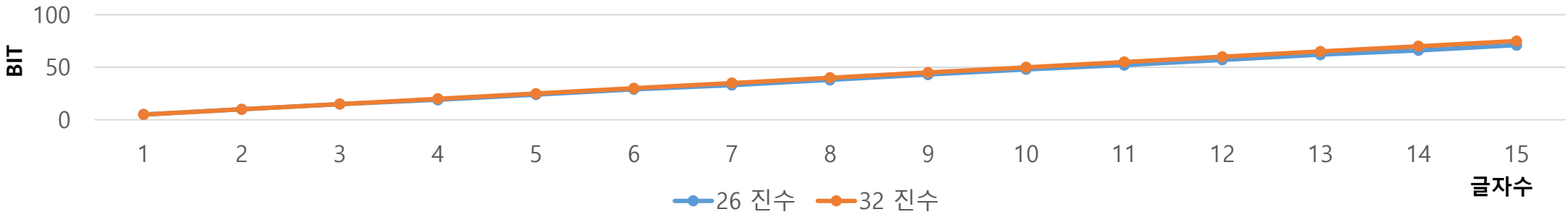
- $NumToBit^n()$
  - 자연수 N을 $\lceil \log_2 N \rceil$ 비트의 비트열로 변환

랭크 함수 최적화 기법 #1

- String to Num의 문제점
  - 26진수의 비효율성 → 32진수로 변환 (padding)하여 나타냄
  - 구현관점

	26진수	32진수
Modular 연산	% 연산자 이용 (느림)	&0x1F 연산자 이용 (빠름)
Division 연산	/ 연산자 이용 (느림)	>>5 연산자 이용 (빠름)

  - 영문 3 글자 (영문 이니셜) 인 경우 $\log_2 26^3 = 14.1013$ 이고 $\log_2 32^3 = 15$ 이므로 동일한 길이의 암호화 연산 수행



연산 효율성 비교 (modular 연산) #1

```
// modular 연산
temp = a % 26;
push        0
push        1Ah
mov         eax,dword ptr
push        eax
mov         ecx,dword ptr
push        ecx
call        _aullrem (0BE1480h)
mov         dword ptr, eax
mov         dword ptr, edx
```

Speed up

```
//  &0x1f 연산으로 대체

temp = a & 0x1f;
mov         edx,dword ptr
and         edx,1Fh
mov         eax,dword ptr
and         eax,0
mov         dword ptr, edx
mov         dword ptr, eax
```

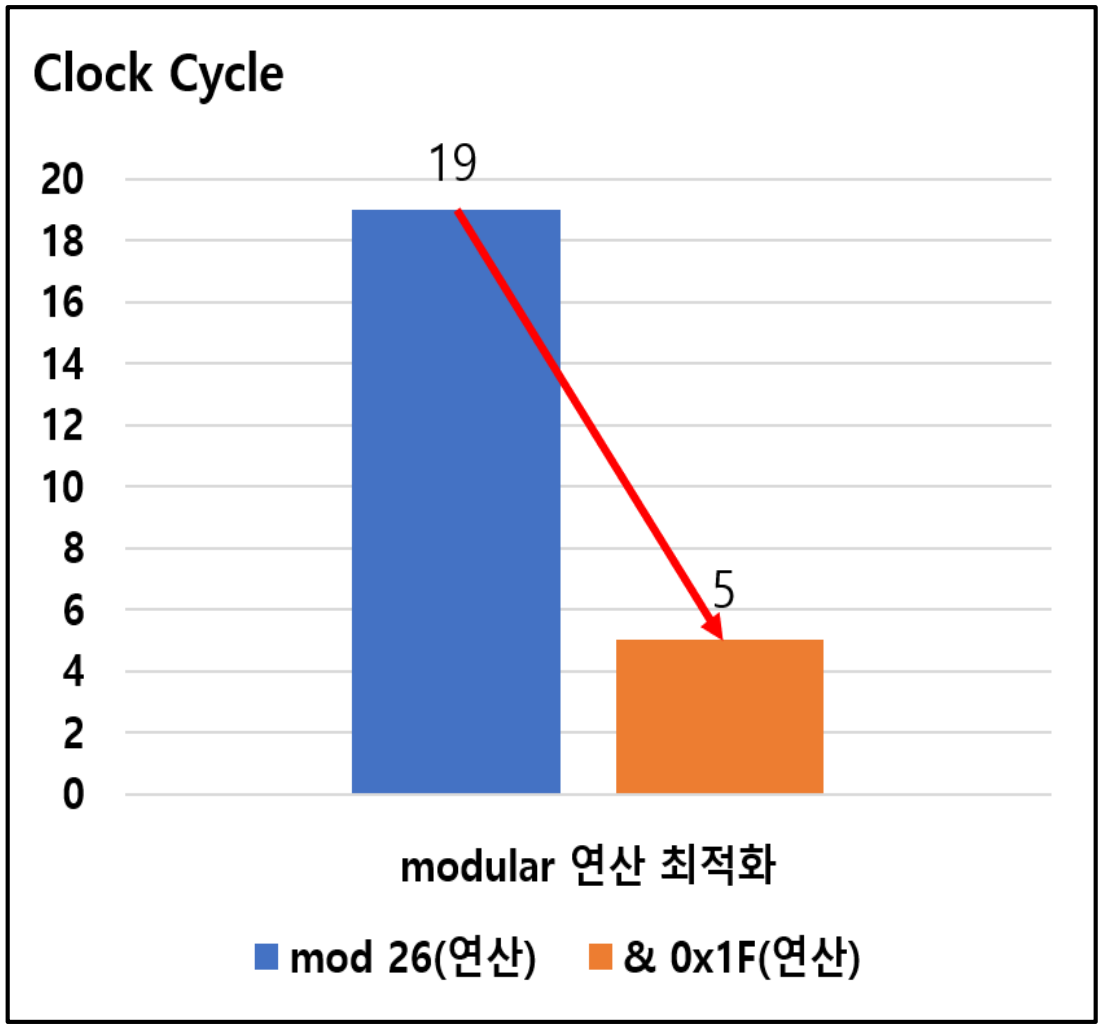
연산 효율성 비교 (division 연산) #2

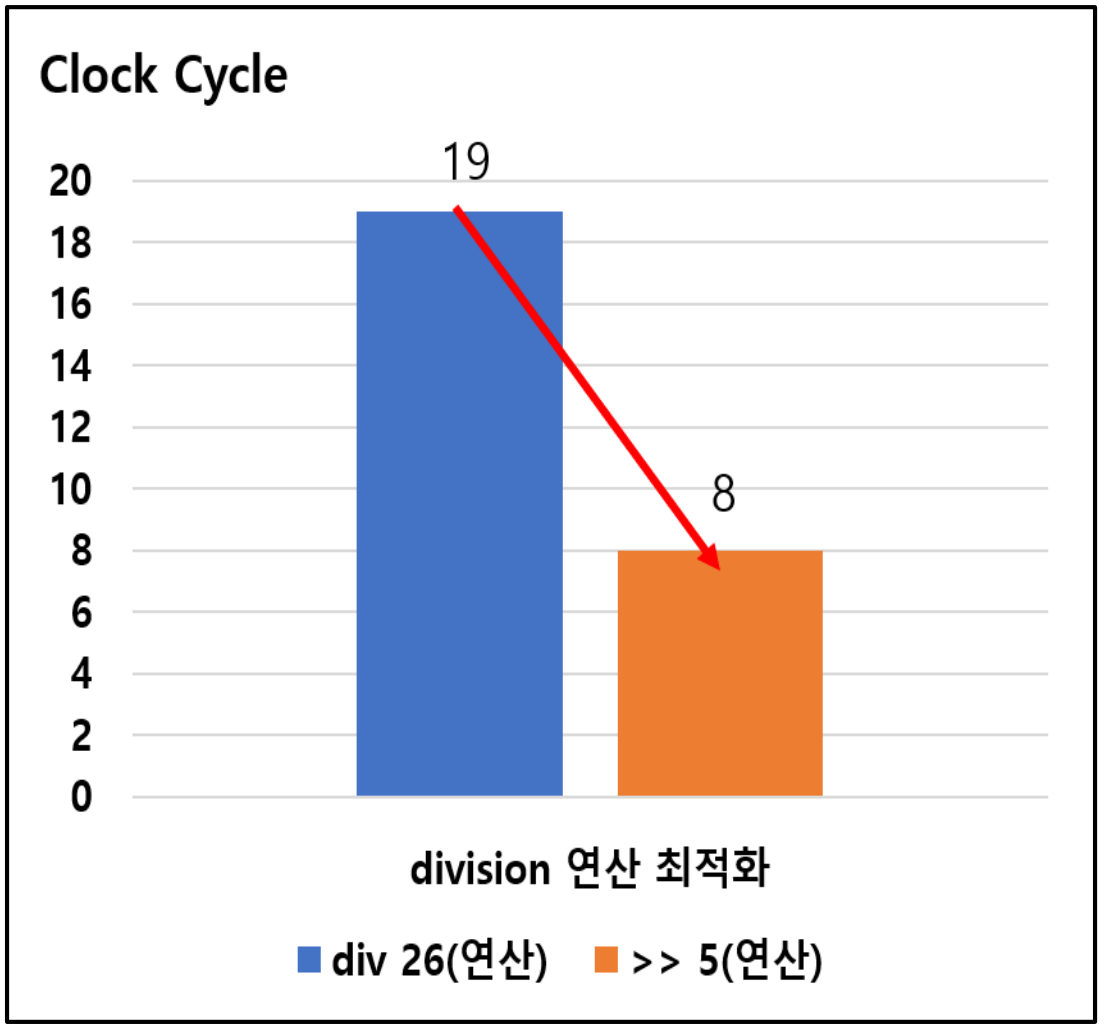
```
// division 연산
temp = a / 26;
push        0
push        1Ah
mov         eax,dword ptr
push        eax
mov         ecx,dword ptr
push        ecx
call        _aulldiv (0BE1250h)
mov         dword ptr, eax
mov         dword ptr, edx
```

Speed up

```
//  Shift 연산으로 대체

temp = a >>5;
mov         eax,dword ptr
mov         edx,dword ptr
mov         cl,5
call        _aullshr (0BE1230h)
mov         dword ptr, eax
mov         dword ptr, edx
```

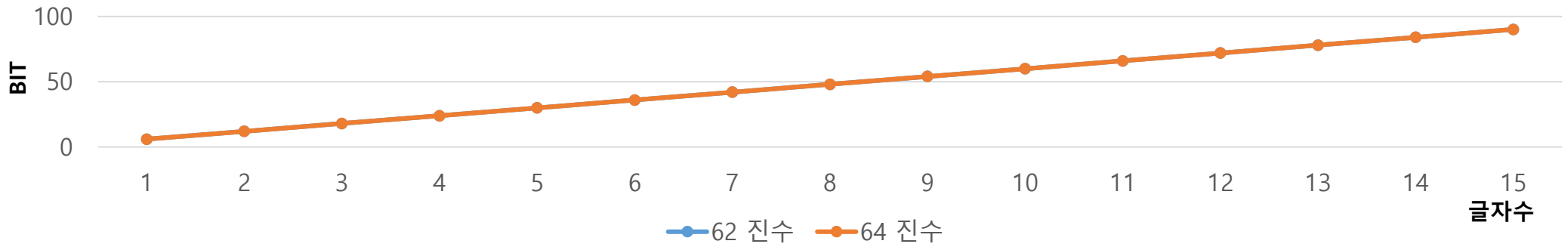
연산 최적화(64bit TestCase 1억개 mod, div 연산 최적화)





랭크 함수 최적화 기법 #2

- String to Num의 문제점 #2
  - 영문자의 소문자 대문자 구별과 숫자를 다 사용시
  - 26+26+10 → 62 진수 (div 62, mod 62는 비효율적임)
  - 따라서 **64진수로 패딩하여 사용(위와 동일한 효과)**
  - 시리얼 번호, 여권 번호에 사용 가능
  - **62진수와 64진수가 1~15자리인 경우 동일한 비트 필요**

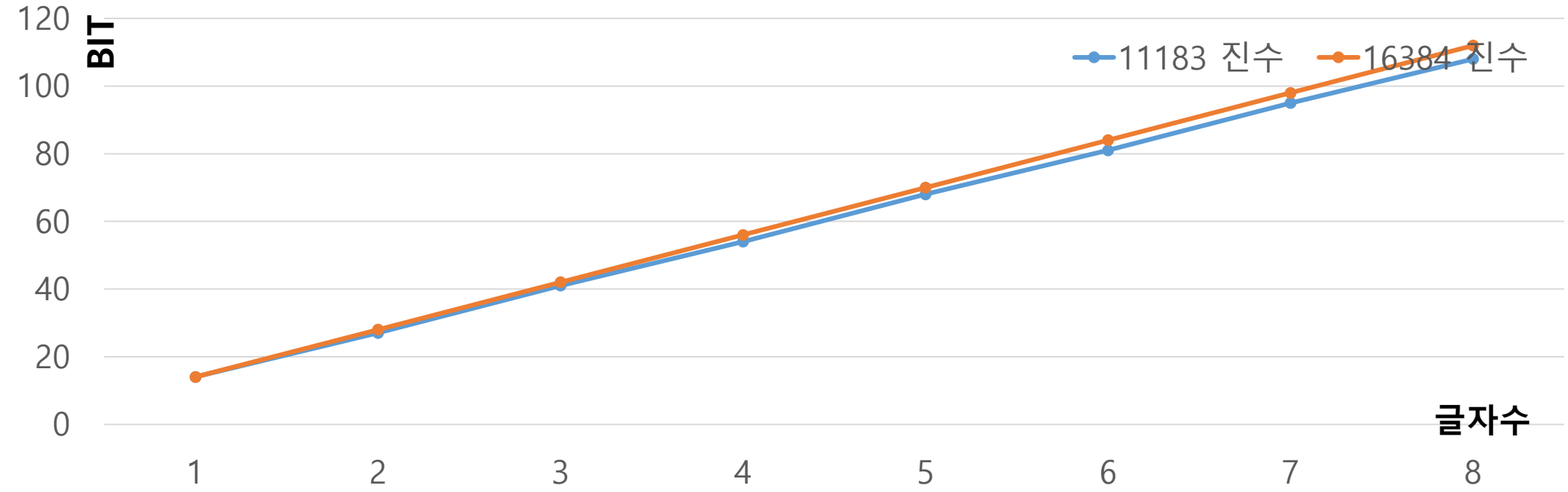


랭크 함수 최적화 기법 #3

- 설문지 답안 암호화 저장 (프라이버시 보장)
  - 4지 선다 문제의 경우 경우의 수가 총 4개임
  - 즉 2 진수를 이용하여 정보 저장 가능
  - 한정된 경우의 수만 활용하여 저장 공간 최소화

랭크 함수 최적화 기법 #4

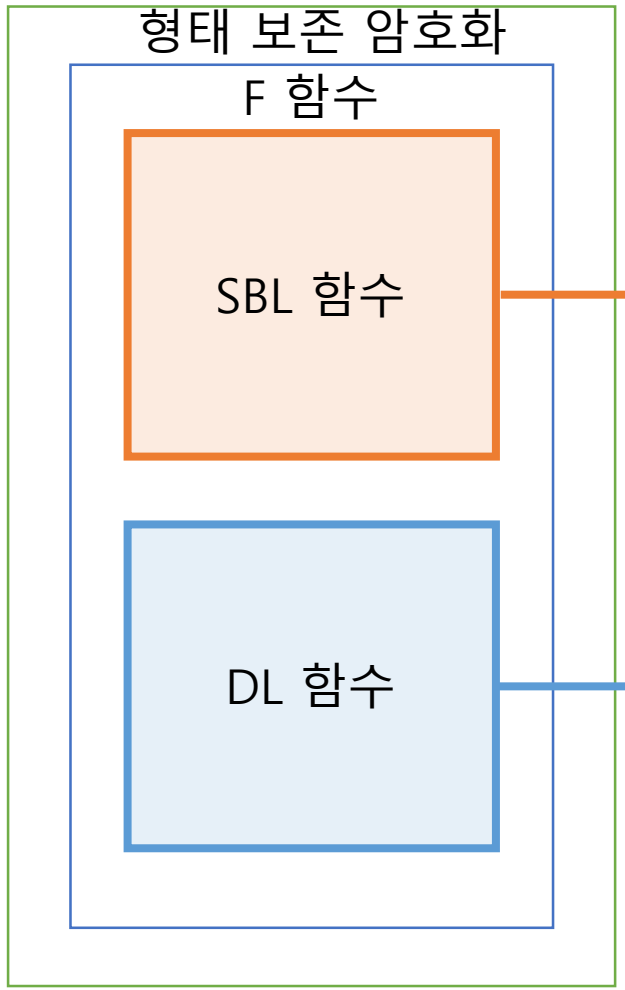
- UTF-8 한글의 경우 랭크 함수 적용 방안
  - UTF-8 상의 한글 코드 범위 {AC00-D7AF} → 11,183 진수 사용
  - **16,384 진수 사용(위와 동일한 효과)**



SBL 및 DL 계층 최적화 기법

- 형태 보존 암호 핵심 연산 소개
- SBL / DL 연산 소개
- SBL / DL 연산 최적화

형태 보존 암호 핵심 연산



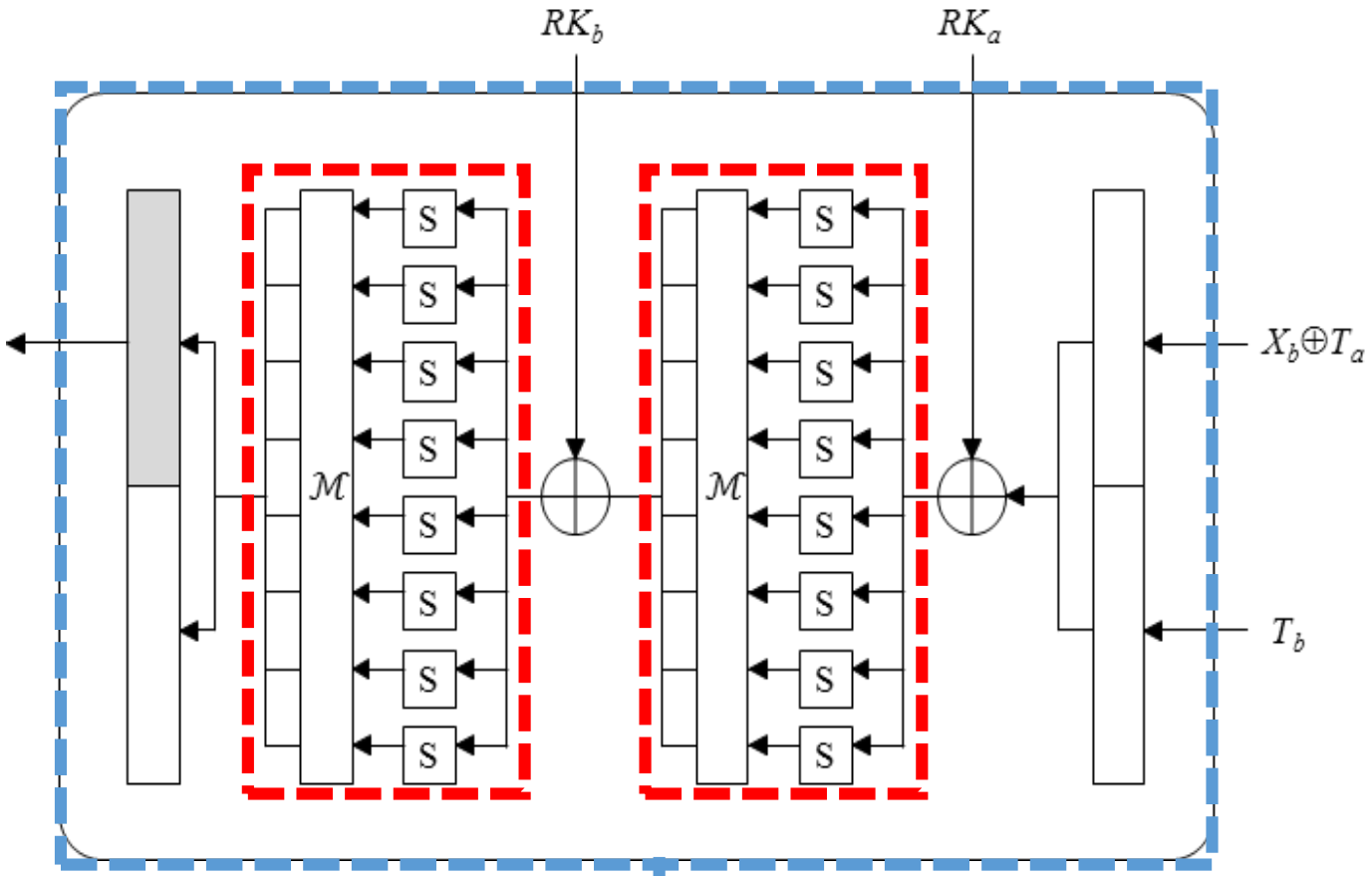
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	62	31	70	8e	bc	30	9c	78	e0	5c	ce	bb	42	ac	b8	df
1	29	e7	86	5f	ee	ba	3f	87	c0	36	c3	14	7c	ec	73	da
2	57	72	f6	77	98	3b	c5	c4	4c	52	81	20	15	97	26	fc
3	8b	3c	af	6e	c8	7e	f0	40	24	a1	b1	54	ff	ad	51	bd
4	c1	13	41	b5	6b	94	63	d6	de	6f	89	d2	a9	d4	17	38
5	a5	f2	e3	db	47	66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

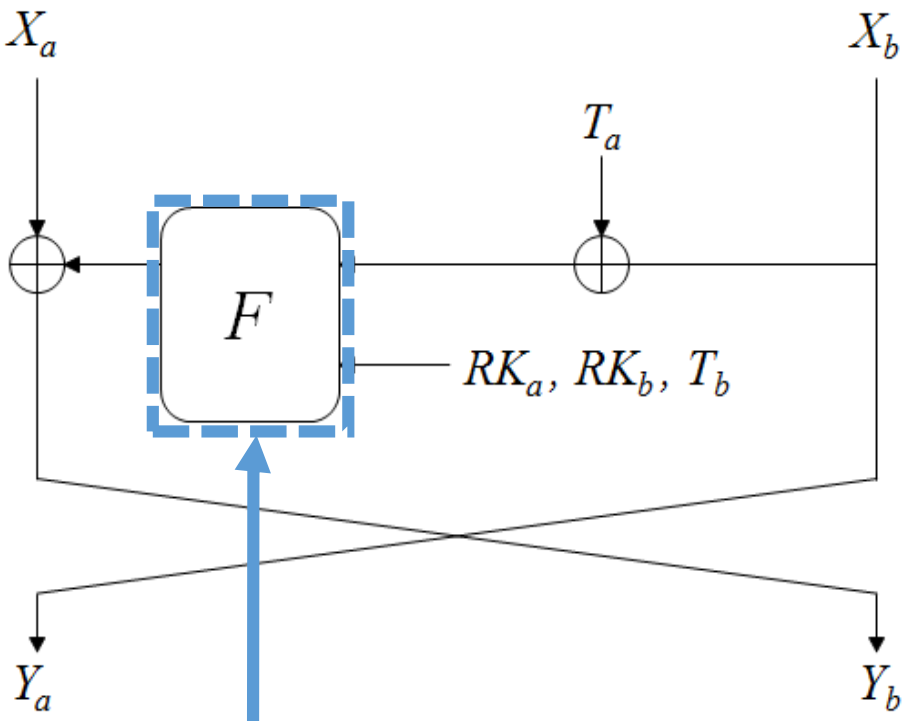
$$M = \begin{bmatrix} 28 & 1a & 7b & 78 & c3 & d0 & 42 & 40 \\ 1a & 7b & 78 & c3 & d0 & 42 & 40 & 28 \\ 7b & 78 & c3 & d0 & 42 & 40 & 28 & 1a \\ 78 & c3 & d0 & 42 & 40 & 28 & 1a & 7b \\ c3 & d0 & 42 & 40 & 28 & 1a & 7b & 78 \\ d0 & 42 & 40 & 28 & 1a & 7b & 78 & c3 \\ 42 & 40 & 28 & 1a & 7b & 78 & c3 & d0 \\ 40 & 28 & 1a & 7b & 78 & c3 & d0 & 42 \end{bmatrix}$$

치환 계층 (SBL):
8-비트 입력 값에 대한 치환된 8-비트 출력 도출

확산 계층 (DL):
8 × 8 행렬 M의 곱으로 표현

형태 보존 암호 핵심 연산





기존 SBL/DL 연산 과정

8개의 8-비트 인자의 입력 값이 $IN1 \sim IN8$ 인 경우

$$\begin{bmatrix} 28 & 1a & 7b & 78 & c3 & d0 & 42 & 40 \\ 1a & 7b & 78 & c3 & d0 & 42 & 40 & 28 \\ 7b & 78 & c3 & d0 & 42 & 40 & 28 & 1a \\ 78 & c3 & d0 & 42 & 40 & 28 & 1a & 7b \\ c3 & d0 & 42 & 40 & 28 & 1a & 7b & 78 \\ d0 & 42 & 40 & 28 & 1a & 7b & 78 & c3 \\ 42 & 40 & 28 & 1a & 7b & 78 & c3 & d0 \\ 40 & 28 & 1a & 7b & 78 & c3 & d0 & 42 \end{bmatrix} \circ \begin{pmatrix} S(IN1) \\ S(IN2) \\ S(IN3) \\ S(IN4) \\ S(IN5) \\ S(IN6) \\ S(IN7) \\ S(IN8) \end{pmatrix} = \begin{pmatrix} S(IN1) \cdot 0x28 + \cdots + S(IN8) \cdot 0x40 \\ S(IN1) \cdot 0x1a + \cdots + S(IN8) \cdot 0x28 \\ S(IN1) \cdot 0x7b + \cdots + S(IN8) \cdot 0x1a \\ S(IN1) \cdot 0x78 + \cdots + S(IN8) \cdot 0x7b \\ S(IN1) \cdot 0xc3 + \cdots + S(IN8) \cdot 0x78 \\ S(IN1) \cdot 0xd0 + \cdots + S(IN8) \cdot 0xc3 \\ S(IN1) \cdot 0x42 + \cdots + S(IN8) \cdot 0xd0 \\ S(IN1) \cdot 0x40 + \cdots + S(IN8) \cdot 0x42 \end{pmatrix}$$

기존 SBL/DL 연산 최적화

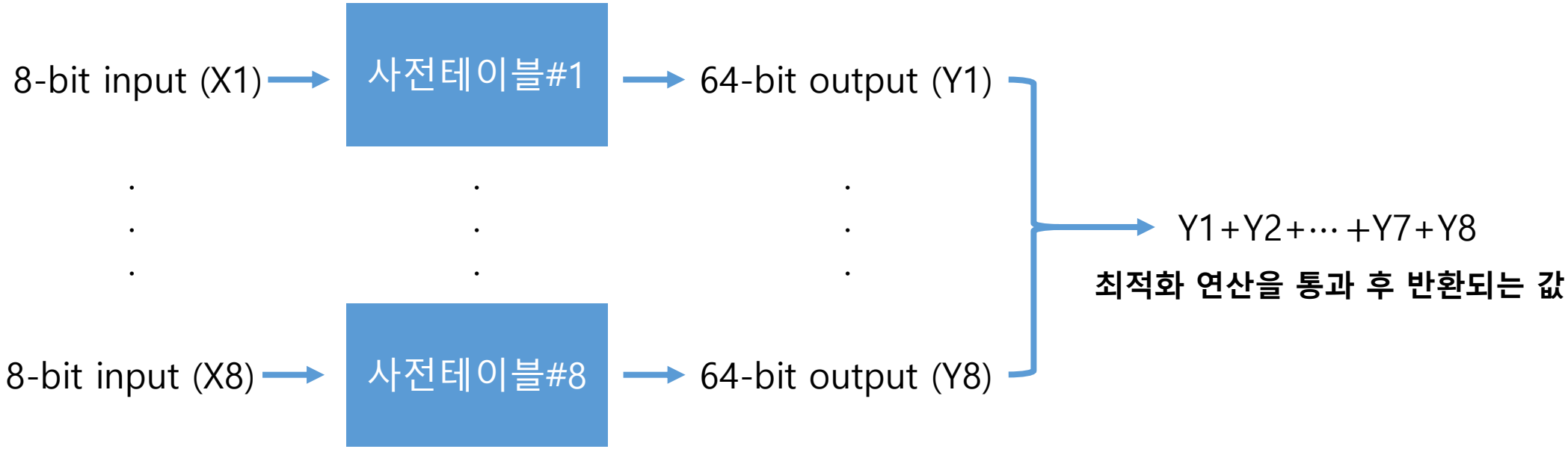
8개의 8-비트 인자의 입력 값이 /N1~/N8 인 경우 (DL의 M 박스의 반복 구조 표시)

$$\begin{bmatrix} 28 & 1a & 7b & 78 & c3 & d0 & 42 & 40 \\ 1a & 7b & 78 & c3 & d0 & 42 & 40 & 28 \\ 7b & 78 & c3 & d0 & 42 & 40 & 28 & 1a \\ 78 & c3 & d0 & 42 & 40 & 28 & 1a & 7b \\ c3 & d0 & 42 & 40 & 28 & 1a & 7b & 78 \\ d0 & 42 & 40 & 28 & 1a & 7b & 78 & c3 \\ 42 & 40 & 28 & 1a & 7b & 78 & c3 & d0 \\ 40 & 28 & 1a & 7b & 78 & c3 & d0 & 42 \end{bmatrix} \circ \begin{pmatrix} S(IN1) \\ S(IN2) \\ S(IN3) \\ S(IN4) \\ S(IN5) \\ S(IN6) \\ S(IN7) \\ S(IN8) \end{pmatrix} = \begin{pmatrix} S(IN1) \cdot 0x28 + \cdots + S(IN8) \cdot 0x40 \\ S(IN1) \cdot 0x1a + \cdots + S(IN8) \cdot 0x28 \\ S(IN1) \cdot 0x7b + \cdots + S(IN8) \cdot 0x1a \\ S(IN1) \cdot 0x78 + \cdots + S(IN8) \cdot 0x7b \\ S(IN1) \cdot 0xc3 + \cdots + S(IN8) \cdot 0x78 \\ S(IN1) \cdot 0xd0 + \cdots + S(IN8) \cdot 0xc3 \\ S(IN1) \cdot 0x42 + \cdots + S(IN8) \cdot 0xd0 \\ S(IN1) \cdot 0x40 + \cdots + S(IN8) \cdot 0x42 \end{pmatrix}$$

기존 SBL/DL 연산 최적화 #1

SBL과 DL을 묶어서 사전테이블 구현: $S(IN1) \cdot 0x28 + \cdots + S(IN1) \cdot 0x40$ → $X \cdot 0x28 + \cdots + X \cdot 0x40$

여기서 X는 $0 \leq X \leq 255$ 이므로 총 $16KB$ $(8byte \times 256 \times 8)$의 **사전 계산 결과값 (사전 테이블)이 필요**



기존 SBL/DL 연산 최적화 #2

※ 0<=IN1~8<256 → 8bit 입력 값들

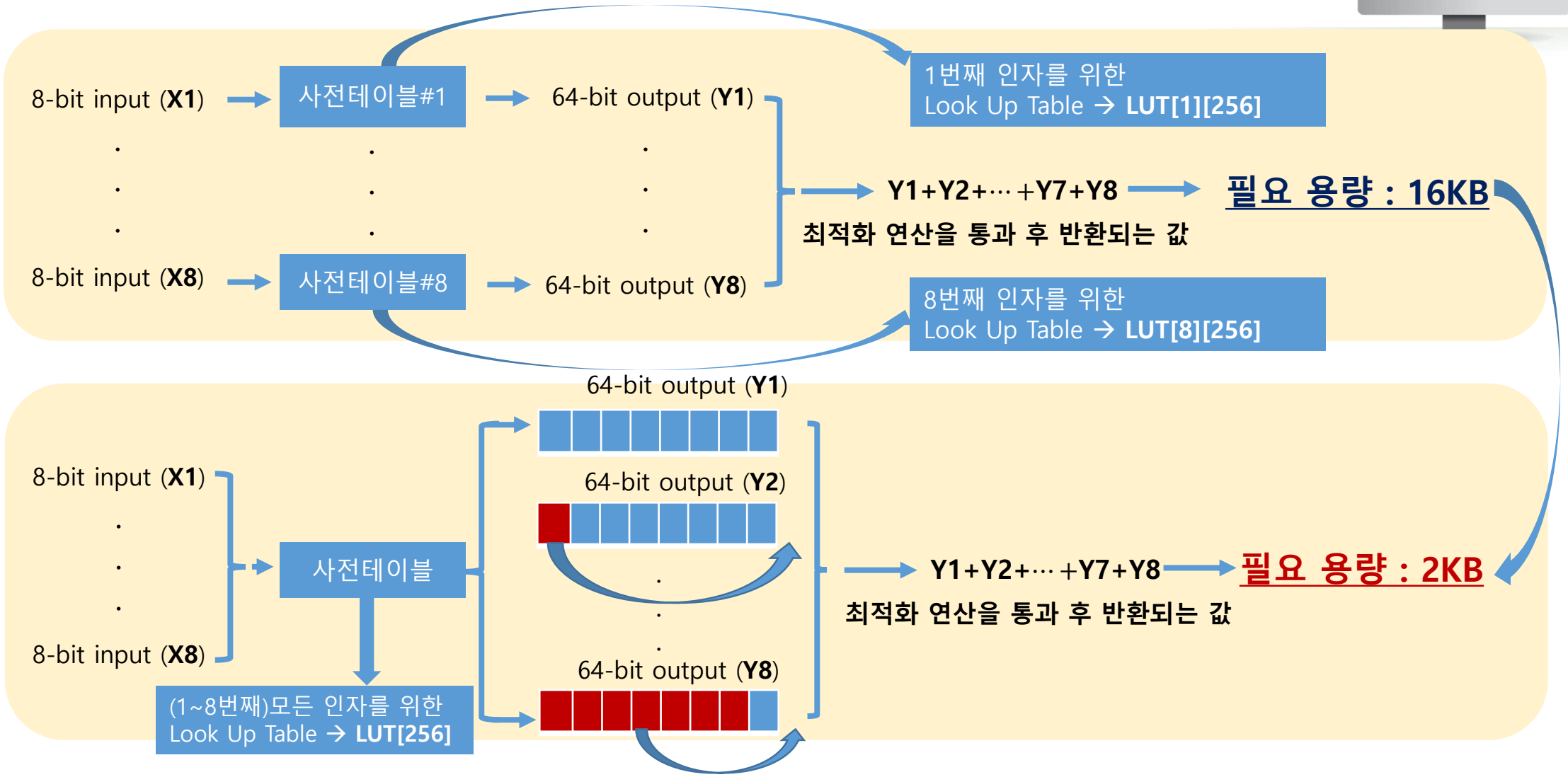
$$\begin{pmatrix} S(IN1)\cdot 0x28 + \cdots + S(IN8)\cdot 0x40 \\ S(IN1)\cdot 0x1a + \cdots + S(IN8)\cdot 0x28 \\ S(IN1)\cdot 0x7b + \cdots + S(IN8)\cdot 0x1a \\ S(IN1)\cdot 0x78 + \cdots + S(IN8)\cdot 0x7b \\ S(IN1)\cdot 0xc3 + \cdots + S(IN8)\cdot 0x78 \\ S(IN1)\cdot 0xd0 + \cdots + S(IN8)\cdot 0xc3 \\ S(IN1)\cdot 0x42 + \cdots + S(IN8)\cdot 0xd0 \\ S(IN1)\cdot 0x40 + \cdots + S(IN8)\cdot 0x42 \end{pmatrix}$$

Y_1 $Y_{2...7}$ Y_8

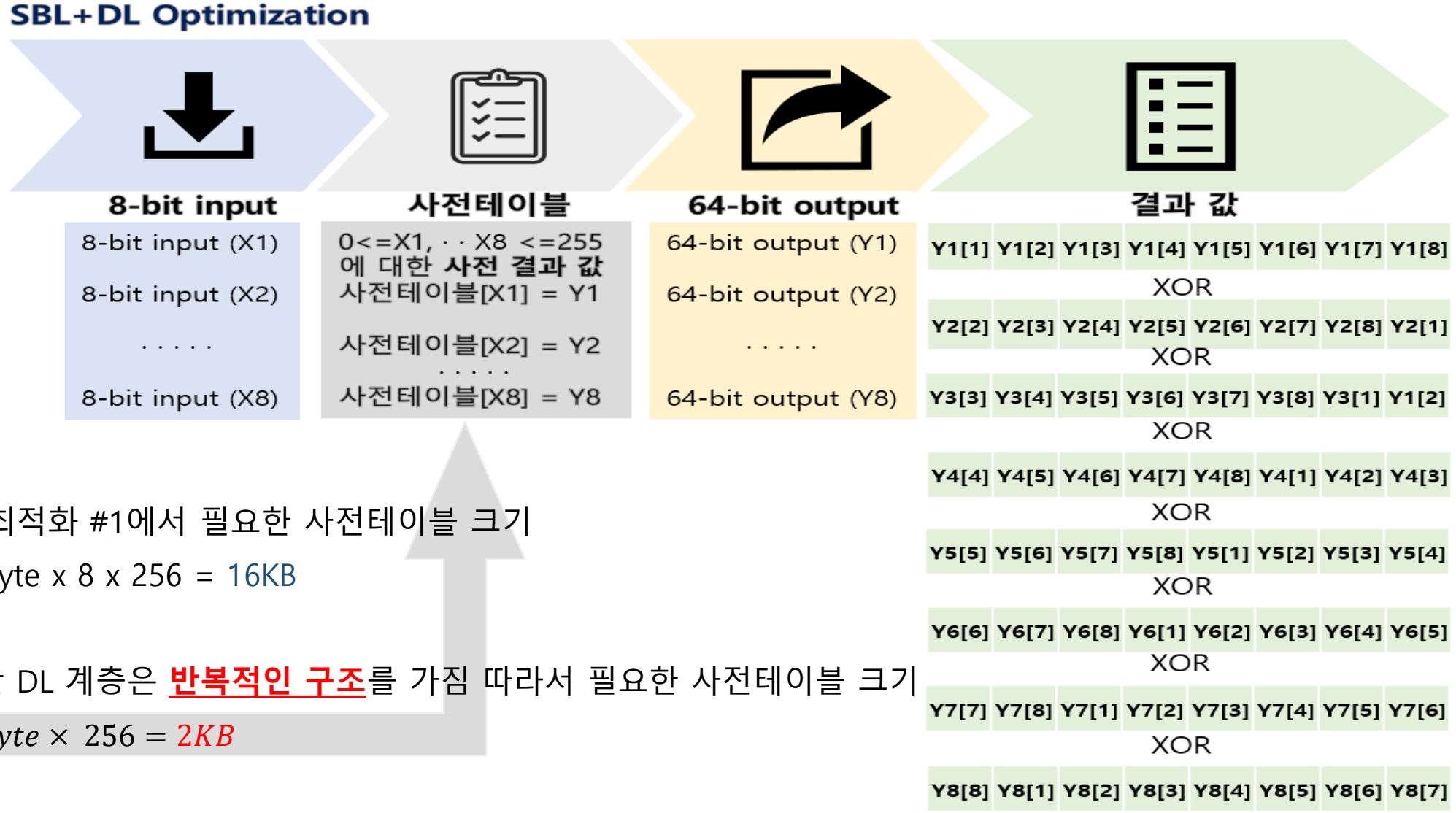
$X_1 = Y_1[0] + \cdots + Y_8[0]$
$X_2 = Y_1[1] + \cdots + Y_8[1]$
$X_3 = Y_1[2] + \cdots + Y_8[2]$
$X_4 = Y_1[3] + \cdots + Y_8[3]$
$X_5 = Y_1[4] + \cdots + Y_8[4]$
$X_6 = Y_1[5] + \cdots + Y_8[5]$
$X_7 = Y_1[6] + \cdots + Y_8[6]$
$X_8 = Y_1[7] + \cdots + Y_8[7]$

SBL과 DL을 통과 후 반환되는 값

기존 SBL/DL 연산 최적화 #3



기존 SBL/DL 연산 최적화 #4



- 연산 최적화 #1에서 필요한 사전테이블 크기
  - → 8byte x 8 x 256 = 16KB

- 하지만 DL 계층은 **반복적인 구조**를 가짐 따라서 필요한 사전테이블 크기
  - → $8byte \times 256 = 2KB$

기존 SBL/DL 연산 최적화 #5

- **사전 테이블 LUT[256]을 미리 정의 해주면 기존 SBL, DL를 위 과정으로 축약 할 수 있음**

※ LUT[256]에는 0<=in[0~7]<256에 대한 SBL, DL을 통과한 64Bits 값들이 저장되어 있음

```
Sum = LUT[in[0]] ^
(  (LUT[in[1]] << 8) ^ (LUT[in[1]] >> 56) ) ^ (  (LUT[in[2]] << 16) ^ (LUT[in[2]] >> 48) ) ^
(  (LUT[in[3]] << 24) ^ (LUT[in[3]] >> 40) ) ^ (  (LUT[in[4]] << 32) ^ (LUT[in[4]] >> 32) ) ^
(  (LUT[in[5]] << 40) ^ (LUT[in[5]] >> 24) ) ^ (  (LUT[in[6]] << 48) ^ (LUT[in[6]] >> 16) ) ^
(  (LUT[in[7]] << 56) ^ (LUT[in[7]] >> 8) );
```

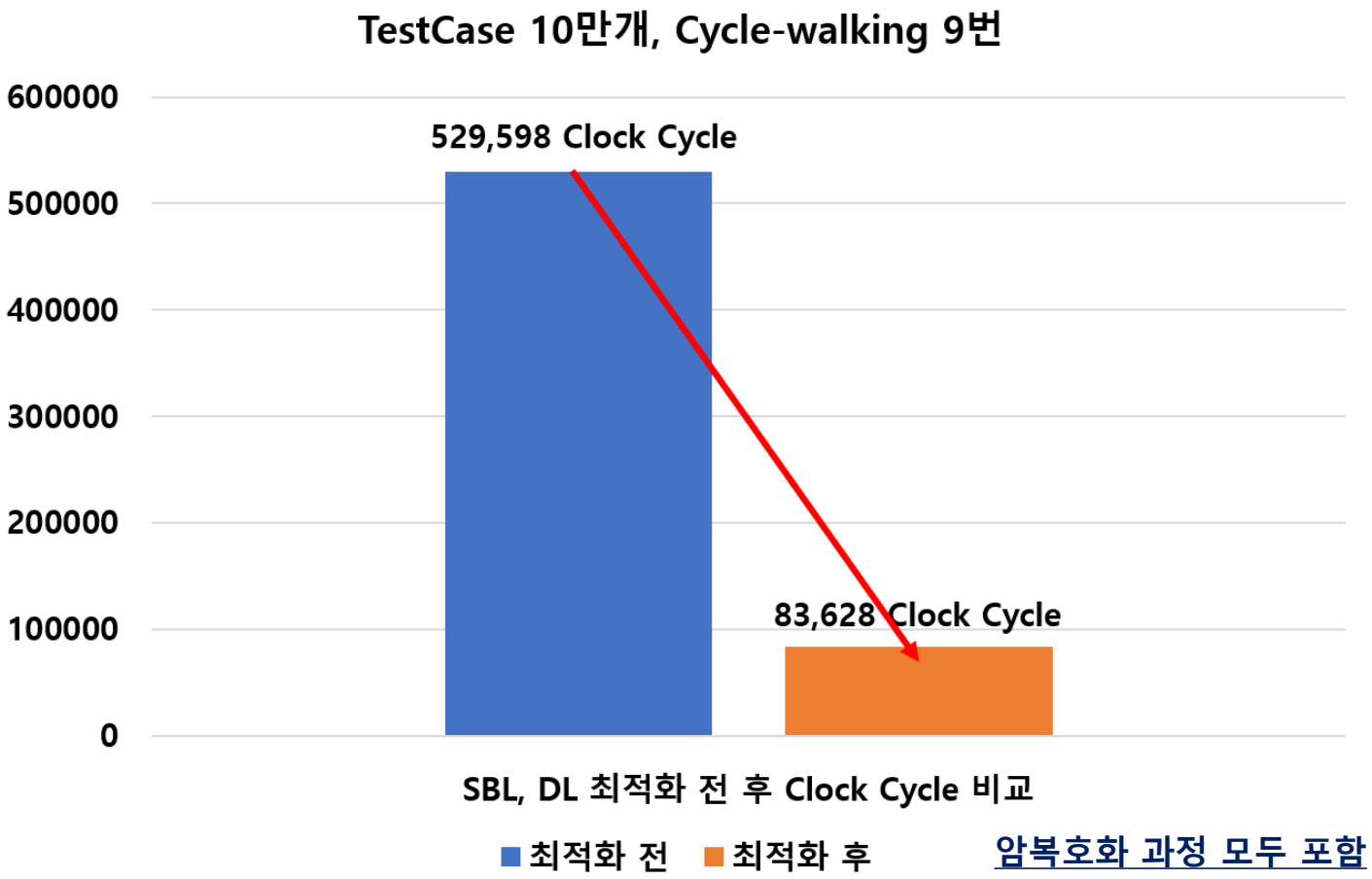
Setting Environment(PC) #1

- **Processor** : Intel® Core™ i5-6200U CPU @ 2.30GHz 2.40 GHz ACPI x64기반 PC
- **Memory(RAM)** : 8GB
- **Setting Environment** for using Microsoft visual Studio C++ 2010 Express x86 tools.
- **최적화 컴파일 옵션** : -O2 (PPT 12pg 만 -OD로 구현 → 연산 크기가 작기 때문)
- **구현언어** : C : 4~64-bit 평문 암복호화(내부 과정 전부 구현, 단계별 최적화에 따른 속도향상)
- **코드라인** : (2~8byte) 각 바이트의 평문 암복호화 코드 당 대략 1천 라인
- functions for TBC_KS operation **type 2**
- **구현 방식** : TTA-Standard 바탕으로 구현 후 SBL,DL & 랭킹함수 최적화

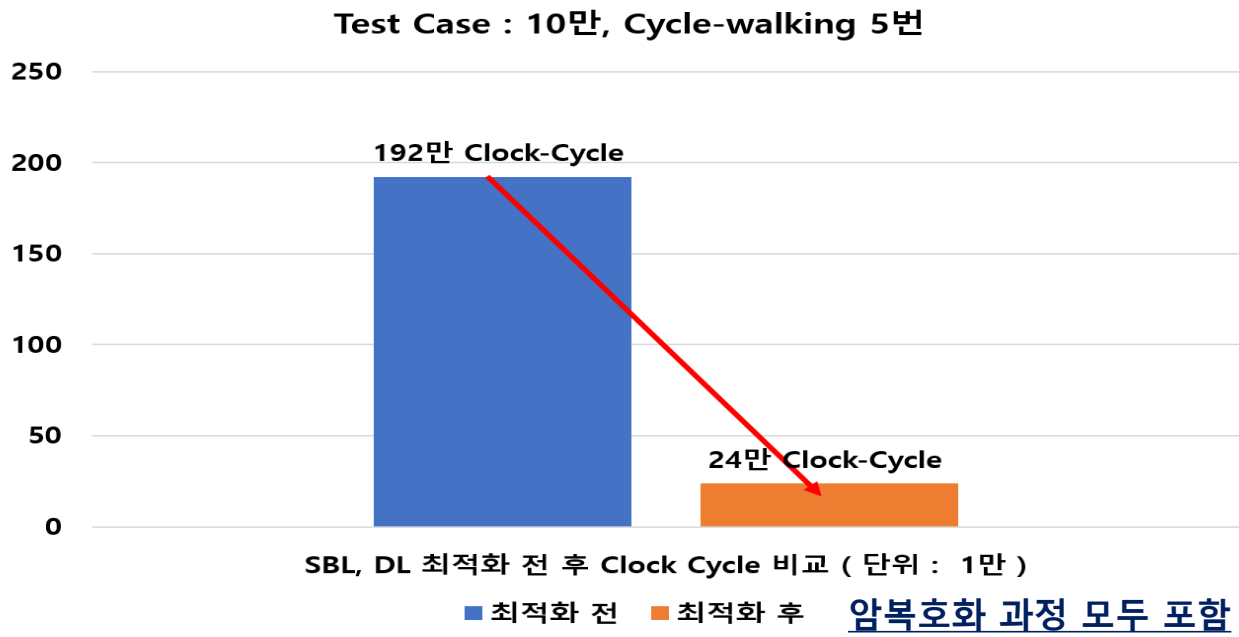
Setting Environment(Arduino 1.8.1) #2

- **제품** : Arduino UNO R3
- **성능** : 8-bit AVR Processor
- **Microcontroller** : ATmega328
- **Clock frequency** : 16MHz
- **EEPROM** : 1KB
- **SRAM** : 2KB
- **Flash memory** : 32KB
- **구현 환경** : Arduino IDE
- **구현 언어** : AVR-GCC : 4~32-bit 평문 암복호화(내부 과정 전부 구현)
- **구현 방식** : functions for TBC_KS operation type 2 (c 언어와 동일한 방식)

PC 에서 기존 SBL/DL 연산 최적화 결과



Aduino에서 FEA 성능 비교 및 분석



최적화에 따른 속도 향상

PC : 6배 아두이노 : 8배

→속도 향상 폭 : 아두이노 > PC

❖ 속도향상 폭 설명

1) PC 성능 > Aduino 성능 → PC는 따로 ALU를 사용 하지 않아도 아두이노 보다 연산 속도가 빠름
2) Aduino의 ALU 성능이 좋지 않기 때문에 Look Up Table를 사용하면 더 큰 효과를 볼 수 있음

결론

1) FEA 구현 및 랭킹함수 최적화, SBL,DL 최적화

2) 본 구현은 c언어에서 이뤄졌음, 따라서 보다 최적화된 결과를 위해 어셈블리 최적화 필요

3) 본 제안은 형태보존암호의 알고리즘 레벨에서 최적화함, 따라서 어셈블리 구현에서도 적용 가능

4) 추후 어셈블리 단계에서 얼마나 더 최적화 할 수 있을지 연구할 예정

참고 문헌

[1] *Format-Preserving Encryption Algorithm FEA-TTA Standard*

[2] *NIST Special Publication 800-38G*

[3] *https://github.com/robshep/JavaFPE*

[4] *https://github.com/kpdyer/libffx*

[5] https://en.wikipedia.org/wiki/Finite_field_arithmetic

[6] *https://spri.kr/posts/view/16676?code=inderstry_trend*

[7] *http://www.huffingtonpost.kr/pakghun/story_b_9745564.html*

감사합니다
Thank You