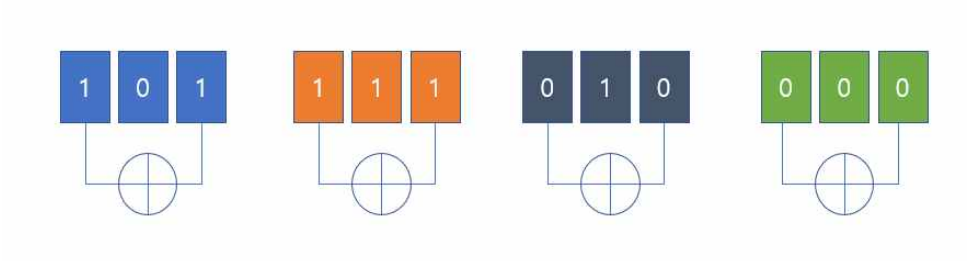
2021 암호분석경진대회			
6번 문제 풀이			
구분	일반부	팀명	HB

0. 개요

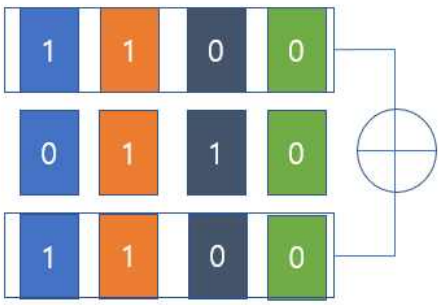
해당 암호는 Timing Attack을 방어하기 위한 비트 슬라이싱 기법이 적용되지 않은 대칭키 암호이다. 비트 슬라이싱 기법을 적용하여 8bit 저전력 프로세서에 맞게 최적화를 진행한다. 문제에는 8bit AVR 프로세서 아두이노 우노가 사용된다. 또한 S-Box 연산에 대해 비트 슬라이싱 기법을 활용하여야 한다. 기존에 주어진 코드를 실행 시 벤치마크 결과는 9705ms이다.

1. 비트 슬라이싱 (Bit-Slicing)

비트 슬라이싱이란, 작은 비트의 모듈을 모아 큰 비트의 프로세서를 제작하는 기술이다. 예를 들어, 4bit 프로세서에서 다음과 같은 연산을 한다고 하자.



위 연산은 3bit 값 4개에 대해 총 4번의 XOR 연산을 수행한다. 하지만 4bit 프로세서에서 는 기본 연산 단위가 4bit 이고, 해당 연산을 수행하기 위해서는 총 4번의 4bit XOR을 진행하는 것과 동일하므로, 최적의 방법은 해당 12bit를 4bit 3개로 바꾸어 연산(transpose)하는 것이다. 그러므로 다음과 같이 연산이 가능하다.



위 연산의 경우, 기존 12bit (4 x 3 bit)를 transpose 하여 3 x 4bit 로 변경한 것이다. 위와 같이 transpose 후 연산 시, 1번의 4bit XOR 연산을 통해 동일한 결과를 얻을 수 있다.

2. 카르노 맵 (Karnaugh Map)

S-Box 연산에 비트 슬라이싱을 적용하기 앞서, 카르노 맵에 대한 이해가 필요하다. 카르노 맵은 Bool 대수 위의 함수를 단순화 하는 방법으로, 확장된 논리 표현을 패턴인식에 의해 연관된 상호관계를 이용하여 줄이는 방법이다. 예를 들어, 문제에 주어진 S-Box의 각 원소에 대한 4bit 값의 MSB (7 = 0b0111 의 경우 MSB는 0)에 대해 카르노 맵을 그리면 다음과 같다.

(S-Box의 index에 해당하는 값에 대한 각 bit를 MSB부터 A,B,C,D라 하자. 7 = 0b0111의 경우, A=0, B=1, C=1, D=1 이다.)

AB\CD	00	01	11	10
00	0	1	1	0
01	1	0	0	1
11	1	0	0	1
10	1	0	0	1

위 표에서와 같이 1로만 구성된 값에 대해 $2^n \times 2^m$ 꼴의 박스를 만들 수 있다. (표를 넘는 것 허용) 파란 박스의 경우, C bit에 상관없이, 항상 1의 값을 가지고, A와 B는 0, D는 1일 때 파란 박스의 값이 1임을 알 수 있다. 즉, S-Box[0b0001] = 0b1..., S-Box[0b0011] = 0b1... 과 같다. ('...' 은 해당 bit를 제외한 나머지 bit)

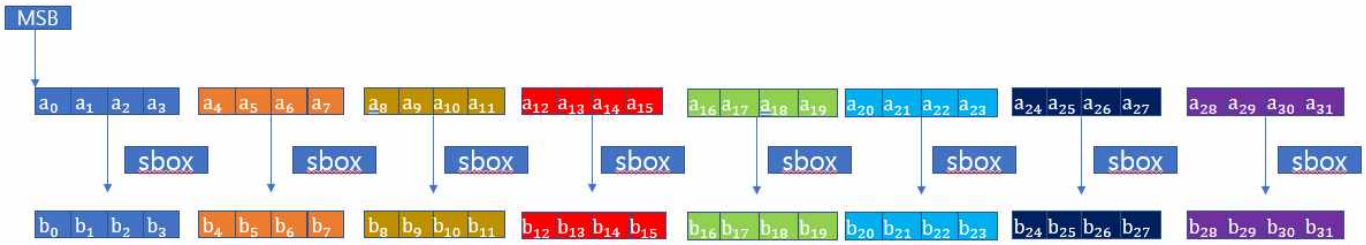
그러므로 다음과 같이 식을 적을 수 있다. $\overline{A} \wedge \overline{B} \wedge D = 1$

이와 같이 나머지 칸에 대해서도 적용할 수 있으며, 최종적으로 위 표를 만족하는 식은 다음과 같음을 알 수 있다. $f(A,B,C,D) = (\overline{A} \wedge \overline{B} \wedge D) \vee (\overline{A} \wedge B \wedge \overline{D}) \vee (A \wedge \overline{D})$

물론, 해당 결과가 유일한 해는 아니며, 논리 연산을 통해 더 적은 연산으로 최적화가 가능하다.

3. 비트 슬라이싱을 활용한 S-Box 연산

기존에 주어진 u32 s_box_gen(u32 text) 함수는 다음과 같이 동작한다.



기존 S-Box 연산은 4bit 단위로 진행되지만, 문제에 사용되는 아두이노 우노는 8bit AVR 프로세서이므로 8bit 단위의 비트 슬라이싱 연산이 필요하다. 즉, 현재 8 x 4 bit 의 u32 text를 4 x 8bit로 transpose한 결과를 이용하여 8bit AVR 프로세서에 맞게 최적화를 진행해야 한다.

비트 슬라이싱을 위해 우선 주어진 S-Box에 대한 카르노 맵을 구하자. 주어진 S-Box는 다음과 같다.

input : 4bit					output : sbox 4bit			
A (MSBof4bit)	B	C	D		A (MSBof4bit)	B	C	D
0	0	0	0		0	0	1	1
0	0	0	1		1	0	0	1
0	0	1	0		0	1	1	0
0	0	1	1		1	1	1	1
0	1	0	0		1	1	1	0
0	1	0	1		0	1	0	1
0	1	1	0	→	1	1	0	1
0	1	1	1		0	1	0	0
1	0	0	0		1	1	0	0
1	0	0	1		0	1	1	1
1	0	1	0		1	0	1	0
1	0	1	1		0	0	1	0
1	1	0	0		1	0	1	1
1	1	0	1		0	0	0	1
1	1	1	0		1	0	0	0
1	1	1	1		0	0	0	0

이에 대해 카르노 맵을 적용한 결과는 다음과 같다.

$$f(A,B,C,D) = (\overline{A} \wedge \overline{B} \wedge D) \vee (B \wedge \overline{D}) \vee (A \wedge \overline{D}) = A\ of\ output$$
$$g(A,B,C,D) = (A \wedge \overline{B} \wedge \overline{C}) \vee (\overline{A} \wedge C) \vee (\overline{A} \wedge B) = B\ of\ output$$
$$h(A,B,C,D) = (\overline{A} \wedge \overline{C} \wedge \overline{D}) \vee (B \wedge \overline{C} \wedge \overline{D}) \vee (A \wedge \overline{B} \wedge D) \vee (\overline{B} \wedge C) = C\ of\ output$$
$$k(A,B,C,D) = (\overline{A} \wedge B \wedge C \wedge \overline{D}) \vee (\overline{A} \wedge \overline{B} \wedge \overline{C}) \vee (\overline{A} \wedge \overline{B} \wedge D) \vee (A \wedge B \wedge \overline{C}) \vee (\overline{C} \wedge D)$$
$$= D\ of\ output$$

각 f, g, h, k에 대해 NOT, AND, OR 연산을 최소화하는 방향으로 식을 정리하면 다음과 같다.
(기존 f, g, h, k의 연산 횟수 : 10, 10, 17, 23회)

$$f(A,B,C,D)$$
$$= (\overline{A} \wedge \overline{B} \wedge D) \vee (B \wedge \overline{D}) \vee (A \wedge \overline{D})$$
$$= (\overline{A} \wedge \overline{B} \wedge D) \vee ((A \vee B) \wedge \overline{D})$$
$$= (\overline{(A \vee B)} \wedge D) \vee ((A \vee B) \wedge \overline{D})$$

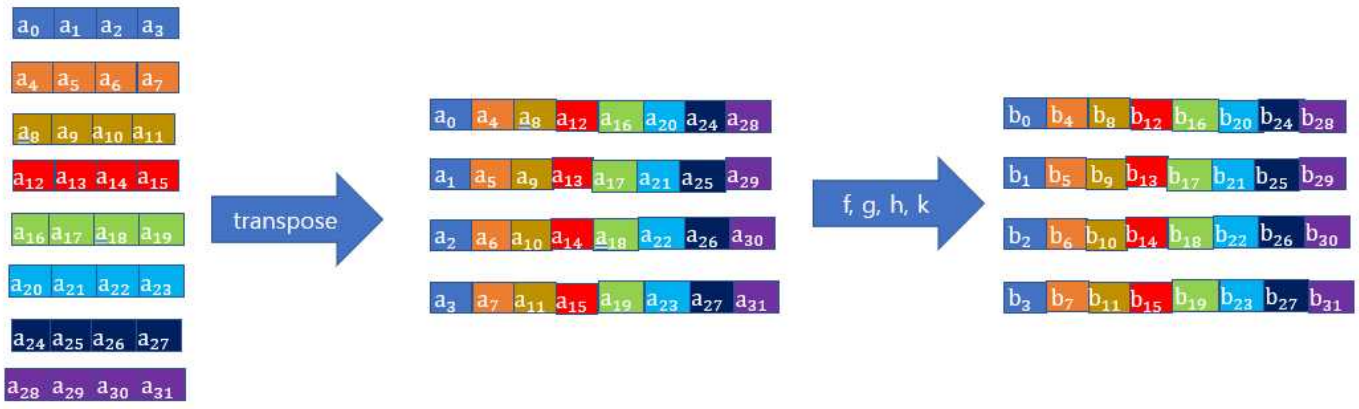
$$g(A,B,C,D)$$
$$= (A \wedge \overline{B} \wedge \overline{C}) \vee (\overline{A} \wedge C) \vee (\overline{A} \wedge B)$$
$$= (A \wedge \overline{B} \wedge \overline{C}) \vee (\overline{A} \wedge (B \vee C))$$
$$= (A \wedge \overline{(B \vee C)}) \vee (\overline{A} \wedge (B \vee C))$$

$$h(A,B,C,D)$$
$$= (\overline{A} \wedge \overline{C} \wedge \overline{D}) \vee (B \wedge \overline{C} \wedge \overline{D}) \vee (A \wedge \overline{B} \wedge D) \vee (\overline{B} \wedge C)$$
$$= ((\overline{A} \vee B) \wedge \overline{(C \vee D)}) \vee (\overline{B} \wedge ((A \wedge D) \vee C))$$

$$k(A,B,C,D)$$
$$= (\overline{A} \wedge B \wedge C \wedge \overline{D}) \vee (\overline{A} \wedge \overline{B} \wedge \overline{C}) \vee (\overline{A} \wedge \overline{B} \wedge D) \vee (A \wedge B \wedge \overline{C}) \vee (\overline{C} \wedge D)$$
$$= (\overline{A} \wedge B \wedge C \wedge \overline{D}) \vee (A \wedge B \wedge \overline{C}) \vee (\overline{C} \wedge D) \vee ((\overline{A} \wedge \overline{B}) \wedge (\overline{C} \vee D))$$
$$= (B \wedge ((\overline{A} \wedge C \wedge \overline{D}) \vee (A \wedge \overline{C}))) \vee (\overline{C} \wedge D) \vee (\overline{(A \vee B)} \wedge (\overline{C} \vee D))$$

(수정한 f, g, h, k의 연산 횟수 : 7, 7, 10, 16회)

각 output (sbox 연산 결과)의 bit에 대한 카르노 맵을 알게 되었으므로, 기존 u32 text를 4 x 8bit 형태로 transpose 후, 새로 생성된 4 x 8bit에 대해 카르노 맵을 적용한 것이 비트 슬라이싱이다. 왜냐하면, 카르노 맵이 bit에 대한 것이고, 각 8bit 값을 f, g, h, k의 파라미터로 적용 시, 각 bit의 자리에 맞게 연산이 진행되기 때문이다.



즉, 다음과 같은 u32 new_s_box_gen(u32 text)를 만들 수 있다.

```
u32 new_s_box_gen(u32 text){
        u32 output = 0;
// transpose
        u8 num1 = (u8)(((text & 0x80000000) >> 24) | ((text & 0x8000000) >> 21) | ((text & 0x800000) >> 18) | ((text & 0x80000) >> 15) | ((text & 0x8000) >> 12) | ((text & 0x800) >> 9) | ((text & 0x80) >> 6) | ((text & 0x8) >> 3));
        u8 num2 = (u8)(((text & 0x40000000) >> 23) | ((text & 0x4000000) >> 20) | ((text & 0x400000) >> 17) | ((text & 0x40000) >> 14) | ((text & 0x4000) >> 11) | ((text & 0x400) >> 8) | ((text & 0x40) >> 5) | ((text & 0x4) >> 2));
        u8 num3 = (u8)(((text & 0x20000000) >> 22) | ((text & 0x2000000) >> 19) | ((text & 0x200000) >> 16) | ((text & 0x20000) >> 13) | ((text & 0x2000) >> 10) | ((text & 0x200) >> 7) | ((text & 0x20) >> 4) | ((text & 0x2) >> 1));
        u8 num4 = (u8)(((text & 0x10000000) >> 21) | ((text & 0x1000000) >> 18) | ((text & 0x100000) >> 15) | ((text & 0x10000) >> 12) | ((text & 0x1000) >>  9) | ((text & 0x100) >> 6) | ((text & 0x10) >> 3) | ((text & 0x1)       ));

// f, g, h, k with karnau map and bit slicing
        u8 f = (u8)((~(num1 | num2) & num4) | ((num1 | num2) & ~num4));
        u8 g = (u8)((num1 & ~(num2 | num3)) | (~num1 & (num2 | num3)));
        u8 h = (u8)(((~num1 | num2) & ~(num3 | num4)) | (~num2 & ((num1 & num4) | num3)));
        u8 k = (u8)((num2 & ((~num1 & num3 & ~num4) | (num1 & ~num3))) | (~num3 & num4) | (~(num1 | num2) & (~num3 | num4)));

        output = (((u32)f << 24) | ((u32)g << 16) | ((u32)h << 8) | ((u32)k));
        return output;
}
```

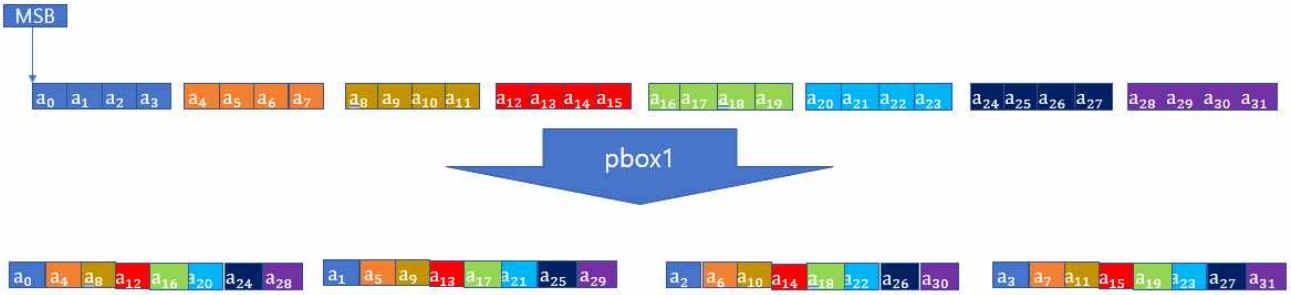
4. 최적화

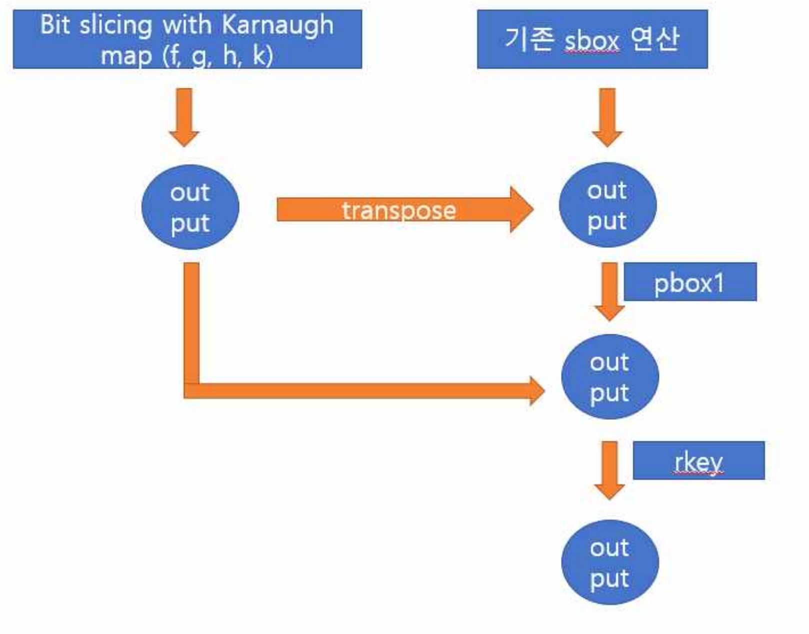
해당 문제는 new_bs_keygen, new_bs_enc 함수의 내부만을 수정하여 최적화를 진행한다. 최적화는 new_bs_enc, new_bs_keygen 순서로 진행하였다.

i. new_bs_enc

new_bs_enc 함수는 32bit x 128 배열인 r_key 와 32bit text를 파라미터로 받아 최종적으로 암호화된 text를 output으로 갖는다. 내부를 살펴보면, NUM_ROUND (128) 번의 loop 동안 s_box_gen, p_box1_gen, XOR r_key_in[i], p_box2_gen 연산을 수행한다. 먼저 "**3.비트 슬라이싱을 활용한 S-Box 연산**"에서 언급한 S-Box 연산에 대한 비트 슬라이싱 결과 (new_s_box_gen)와 p_box1_gen 함수를 살펴보면, 기존 s_box_gen 결과를 p_box1_gen의 input 파라미터로 넣는 것과 new_s_box_gen의 output 결과가 동일함을 알 수 있다.

p_box1_gen 함수는 input으로 들어오는 u32 text를 4bit x 8로 나누었을 때, 각 bit의 자리에 맞는 값끼리 재정렬하는 것이다. 즉, 8bit x 4로 transpose 한 것과 같은 결과를 가지며, 다음과 같이 동작한다.





그러므로, p_box1_gen과 s_box_gen 함수를 결합한 것이, new_s_box_gen 임을 알 수 있다. 이 과정까지 진행 시, 5609ms의 벤치마크 결과를 얻을 수 있다.

또한, new_s_box_gen 함수를 32bit output을 가지는 함수 호출이 아닌, new_bs_enc 내부에서 8bit 4개의 값으로 동작하도록 설계 후, XOR r_key_in[i]를 8bit 4개의 값으로 연산하고, p_box2_gen 함수 또한 함수 호출 대신, p_box2_gen과 동일한 기능을 하도록 bitwise 연산만으로 구성되게 설계할 수 있다. 이 과정을 수행하면 254ms의 벤치마크 결과를 얻을 수 있다.

이후, 기존의 128번 loop에 대해 128번 loop, 64번 loop, 32번 loop, 16번 loop, 8번 loop, 4번 loop, 2번 loop, loop 없음, 이렇게 총 8번의 실험 결과, 32번 loop가 253ms의 벤치마크 결과를 얻어 가장 빠르게 측정되었다.

ⅱ. new_bs_keygen

new_bs_keygen 함수에 대한 최적화는 ROL(constant_value)가 1bit rotation이므로 32번의 연산 시, 기존 값과 일치함을 이용하여 최적화를 진행하였다. 실제로 128번의 loop에 대한 constant_value 값은 다음과 같다.

```
0xAB00CD00, 0x56019A01, 0xAC033402, 0x58066805, 0xB00CD00A, 0x6019A015, 0xC033402A, 0x80668055,
0x00CD00AB, 0x019A0156, 0x033402AC, 0x06680558, 0x0CD00AB0, 0x19A01560, 0x33402AC0, 0x66805580,
0xCD00AB00, 0x9A015601, 0x3402AC03, 0x68055806, 0xD00AB00C, 0xA0156019, 0x402AC033, 0x80558066,
0x00AB00CD, 0x0156019A, 0x02AC0334, 0x05580668, 0x0AB00CD0, 0x156019A0, 0x2AC03340, 0x55806680,
0xAB00CD00, 0x56019A01, 0xAC033402, 0x58066805, 0xB00CD00A, 0x6019A015, 0xC033402A, 0x80668055,
0x00CD00AB, 0x019A0156, 0x033402AC, 0x06680558, 0x0CD00AB0, 0x19A01560, 0x33402AC0, 0x66805580,
0xCD00AB00, 0x9A015601, 0x3402AC03, 0x68055806, 0xD00AB00C, 0xA0156019, 0x402AC033, 0x80558066,
0x00AB00CD, 0x0156019A, 0x02AC0334, 0x05580668, 0x0AB00CD0, 0x156019A0, 0x2AC03340, 0x55806680,
0xAB00CD00, 0x56019A01, 0xAC033402, 0x58066805, 0xB00CD00A, 0x6019A015, 0xC033402A, 0x80668055,
0x00CD00AB, 0x019A0156, 0x033402AC, 0x06680558, 0x0CD00AB0, 0x19A01560, 0x33402AC0, 0x66805580,
0xCD00AB00, 0x9A015601, 0x3402AC03, 0x68055806, 0xD00AB00C, 0xA0156019, 0x402AC033, 0x80558066,
0x00AB00CD, 0x0156019A, 0x02AC0334, 0x05580668, 0x0AB00CD0, 0x156019A0, 0x2AC03340, 0x55806680,
0xAB00CD00, 0x56019A01, 0xAC033402, 0x58066805, 0xB00CD00A, 0x6019A015, 0xC033402A, 0x80668055,
0x00CD00AB, 0x019A0156, 0x033402AC, 0x06680558, 0x0CD00AB0, 0x19A01560, 0x33402AC0, 0x66805580,
0xCD00AB00, 0x9A015601, 0x3402AC03, 0x68055806, 0xD00AB00C, 0xA0156019, 0x402AC033, 0x80558066,
0x00AB00CD, 0x0156019A, 0x02AC0334, 0x05580668, 0x0AB00CD0, 0x156019A0, 0x2AC03340, 0x55806680
```

해당 constant_value 값 128개를 테이블을 미리 전역으로 전언 후, 128번 loop, 64번 loop, 32번 loop, 16번 loop, 8번 loop, 4번 loop, 2번 loop, loop 없음. 총 8가지 경우에 대해 실험해본 결과, 128번 loop의 경우에서 252ms의 벤치마크 결과를 얻었다.

하지만 해당 table은 32개의 constant_value 값으로 구성된 테이블 4개가 중복된 것이고, 전역으로 선언한 테이블의 값을 호출하는 것이 아닌, constant_value가 일치하는 index끼리 모아 하드코딩하여, 위와 같이 128번 loop, 64번 loop, 32번 loop, 16번 loop, 8번 loop, 4번 loop, 2번 loop, loop 없음. 총 8가지에 대한 실험 결과, loop가 없을 때 245ms의 벤치마크 결과를 얻었다.

최종적으로, **245ms**의 벤치마크 결과를 얻었다.

5. 결과

위 최적화 과정을 요약하면 다음과 같다.

시행	벤치마크 결과(ms)
① 기본 구조	9705
② sbox 연산시 비트슬라이싱, pbox1 연산이 불필요 함을 이용	5609
③ 8bit 프로세서에 최적화되도록 32bit input, output을 8bit 4개로 연산 및 pbox2 함수와 동일하게 동작하도록 bitwise 연산만으로 구성	254
④ new_bs_enc 함수 내의 loop 32번	253
⑤ new_bs_keygen 함수의 constant_value 128개를 전역으로 미리 선언 후 loop 128번	252
⑥ constant_value 하드코딩 및 loop 없음	245

최종적으로 약 97.5% 성능이 향상된 245ms의 벤치마크 결과를 얻었고, no6_HB.ino 파일을 통해 확인할 수 있다.

(new_bs_keygen, new_bs_enc 함수 내부만을 수정하였고, 이외의 기존의 함수인 s_box_gen, p_box1_gen, p_box2_gen은 사용하지 않아 no6_HB.ino 파일에서 주석 처리 후 제출합니다.)

①

```
----------------
 TEST VECTOR
----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>> 9705
----------------
```

②

```
----------------
 TEST VECTOR
----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>> 5609
----------------
```

③

```
----------------
 TEST VECTOR
----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>> 254
----------------
```

④

```
----------------
 TEST VECTOR
----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>> 253
----------------
```

⑤

```
----------------
 TEST VECTOR
----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>> 252
----------------
```

⑥

```
----------------
 TEST VECTOR
----------------
>> CORRECT
>> CORRECT
>> CORRECT
----------------
 BENCHMARK
----------------
>> 245
----------------
```


참고자료

1. A Fast New DES Implementation in Software - Eli Biham (1997)
2. https://en.wikipedia.org/wiki/Karnaugh_map