비트 슬라이스 구현에서의 블록암호 비트 순열 최적화*

김성겸*, 권동근*, 김선엽*, 김인성*, 홍득조**, 성재철***, 홍석희****

*, ****고려대학교 (대학원생, 교수), **전북대학교 (교수), ***서울시립대학교 (교수)

Removing Bit Permutation of Block Cipher on the Bitslice Implementation

Seonggyeom Kim*, Donggeun Kwon*, Sunyeop Kim,
Insung Kim, Deukjo Hong**, Jaechul Sung***, Seokhie Hong****

*, ****Korea University(Graduate student, Professor)
**Chonbuk National University(Professor) ***University of Seoul(Professor)

요 약

블록암호의 비트 슬라이스 구현에 있어서 S-box의 각 비트를 슬라이스로 구성하는 경우, 각 S-box의 입·출력 비트 순열(bit permutation)은 슬라이스 간 위치 변경을 요구한다. 하드웨어 구현상 비트 순열은 단순한 하드웨어 배선(wiring)으로 구현 비용이 거의 없는 반면, 소프트웨어 구현에서 이러한 위치 변경은 적어도 3개 이상의 명령어(instruction)와 1개의 추가 레지스터(register)가 요구되기 때문에 슬라이스 간 위치 변경을 없애면 소프트웨어 구현 최적화가 가능하다. 본 논문은 비트 슬라이스 구현에서 라운드 루프 풀기(loop unrolling)기반 구현을 가정하여 슬라이스 간 위치 변경을 없애는 방법을 제시한다. 제안 방법은 블록암호 PIPO에 적용되어, 아두이노 Uno 보드에서 암(복)호화를 12%(10%) 향상시켰다. 추가적으로, 전체 라운드의 루프 풀기가 아니라 부분적인 루프 풀기를 위해 S-box의 비트 순열에 요구되는 조건을 제시한다. 블록암호 PIPO의 경우, 128-(256-)비트 키 지원에 대해 각각 6(+1), 6(+5) 라운드 이상의 부분 루프 풀기가 요구된다.

I. 서론

비트 슬라이스 구현은 n-비트 프로세서 환경에서 n개의 비트 연산을 동시에 수행할 수 있음을 활용한 기법이다. 블록암호의 비트 슬라이스 구현은 Biham이 DES에 처음 적용하였으며[1], 이 방법은 64-비트 64개 블록들의 각 비트 위치 별로 64개의 슬라이스에 재정렬하는 방법을 사용하였다. 이러한 구현 방법은 블록암호의 S-box를 위해 필요한 메모리(즉, Look Up Table)가 요구되지 않기 때문에 경량 환경에 적합하고 캐시 타이밍 공격 저항성을 갖는다.

비트 슬라이스 구현 기법은 모든 블록암호에 적용 가능하나, 비트 슬라이스 구현을 위해서 입·출력 블록(들)을 슬라이스들로 변환하는 packing/unpacking 연산이 추가적으로 필요하다. 이에 NIST-LWC에 제출된 GIFT-COFB는 GIFT의 비트 슬라이스 구현을 고려하여 packing/unpacking을 제외하고 사용한다†. 이와 유사한 방식으로 Serpent, Keccak, Rectangle, PIPO는 사용하는 S-box의 입·출력 기준이 아닌 슬라이스 그대로 입·출력 블록을 표현한다.

다양한 packing 방법이 사용될 수 있으나, 단일 m-비트 S-box S n개로 구성된 S-Layer는 그림 1과 같이 packing이 가능하다. 이러한 비트 슬라이스 구성에서 S에 포함된 비트 순열은 m개의 슬라이스 간 위치 변환이 요구된다. 하드웨어 구현에 있어, 비트 순열은 단순한 하드웨어 배선으로 구현 비용이 거의 없는 반면, 소프트웨어 구현에서 적어도 3개 이상의 명령어가 요구된다‡. 따라서 S-Layer의 비트 슬라이스 표현 함수 $\mathbf{S}$에서 요구되는 슬라이스 간 위치 변환은 소프트웨어 구현 시 무시할 수 없는 비용이 될 수 있다.

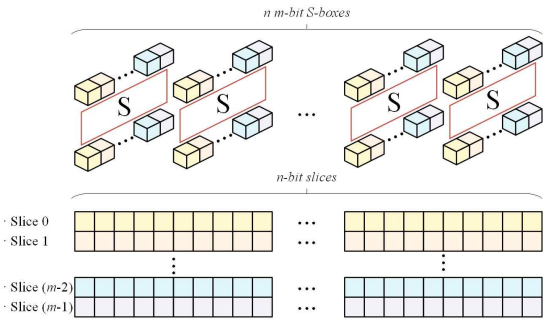

그림 1 단일 m-비트 S-box n개로 구성된 S-Layer의 비트 슬라이스 구현

S-box S의 LUT 표현으로부터 최적(혹은 최소)의 구

* 본 연구는 고려대 암호기술 특화연구센터(UD210027XD)를 통한 방위사업청과 국방과학연구소의 연구비 지원으로 수행되었습니다.

† 입력값이 이미 packing되어 있다고 가정

‡ 비트 순열이 단순히 2개 슬라이스간 SWAP을 수행한다고 했을 때, XOR 3번 또는 MOV 3번 + 임시 레지스터가 요구됨

현 비용을 도출 방법을 제시한 결과[3, 4]에서도 S에 속한 비트 순열은 소프트웨어 구현상 영향이 있음에도 소프트웨어 구현 비용이 없음을 주장하고 있다.

본 논문에서는 S-box에 포함된 비트 순열을 S-Layer의 비트 슬라이스 **S** 연산 중에 제거하는 방법에 대해 논하며, 암호화 구성에 미치는 영향을 제시한다. 특히, 제안 방법은 블록암호 PIPO에 적용되어, 아두이노 Uno 보드에서 암(복)호화를 12%(10%) 향상시켰다*. 여기서 비교 대상은 [5]에 제시한 8-비트 S-box S_8의 참조 코드를 그대로 사용하였다. 표 1는 구현 상세 결과 보여준다.

표 1 PIPO64/128 구현 결과 비교

언어	구분	코드 사이즈	속도 (Mbps)		향상
C	[5]	7518	Enc	769.6	
			Dec	774.5	
	Ours	7542	Enc	778.6	1.2%
			Dec	816.1	5.4%
ASM	[5]	7896	Enc	725.7	
			Dec	739.6	
	Ours	7454	Enc	816.6	12.5%
			Dec	816.7	10.4%

⊙ 코드 사이즈(Bytes) - 암·복호화 함수를 포함
⊙ Target Board - Arduino Uno
⊙ Arduino IDE Version - 1.8.19
⊙ Optimization Option - O3

논문의 구조는 다음과 같다. Ⅱ장에서는 PIPO를 간단히 소개한다. PIPO S-Layer 구현 최적화 방법은 Ⅲ장에서 서술하며, PIPO S-Layer S-box의 변경 및 그 효과를 분석한 결과는 Ⅳ장에서 다룬다. 마지막으로, Ⅴ장에서 본 논문의 결론을 맺는다.

II. 배경 지식

본 논문의 내용은 일반적인 블록암호의 비트슬라이스 구현에 사용될 수 있으나, 설명의 편의상 블록암호 PIPO에 한정하여 서술한다.

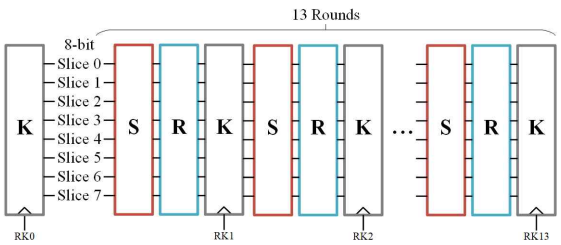

그림 2 PIPO-64/128

2.1 PIPO

ICISC 2020에서 제안된 블록암호 PIPO는 64-비트 블록과 128- 또는 256-비트 키를 지원하는 SPN구조의 경량 블록암호이다[5]. PIPO의 전체적인 구조는 그림 2와 같으며, 키의 크기에 따라 13, 17 라운드가 사용된다. PIPO는 비트 슬라이스 표현을 기본 데이터 구성 형태로 잡았기 때문에 암호화 수행에 있어서 입·출력에 대해 packing/unpacking을 수행하지 않아도 된다.

2.2 PIPO의 라운드 함수

PIPO의 R-Layer(**R**)는 8개의 8-비트 슬라이스 $X[i]$에 독립적으로 비트 회전(Bit Rotation)을 수행한다. 각각의 슬라이스마다 (0, 7, 4, 3, 6, 5, 1, 2)의 크기로 왼쪽 비트 회전을 수행한다. 여기서 회전 크기는 주어진 S-Layer(**S**)에 대해서 차분/선형 공격 저항성 관점에서 최적인 값으로 설정되었다.

S-Layer(**S**)는 8개의 슬라이스 간 연산을 수행하는 것으로, 비선형 연산(AND, OR)이 포함되어 있다. 그림 1의 관점에서 볼 때, 8개의 동일한 S-box로 구성되어 있는 계층으로 볼 수 있다. 이러한 S-box는 그림 3과 같이 Unbalanced-Bridge 형태를 가지고 있으며 출력값에 비트 순열이 존재한다.

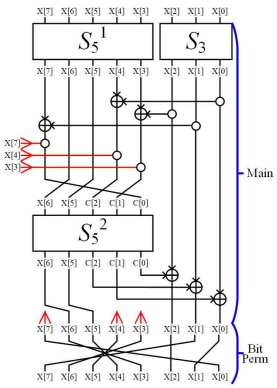

그림 3 PIPO의 S-box S

S-box를 구성하는 내부 구성요소 S_5^1, S_3, S_5^2의 정보 및 구현은 [5]를 참조하면 된다. 본 논문에서는 [5]에 제시한 S_5^1, S_3, S_5^2 구성 및 구현 방법은 그대로 따르면서 출력 비트 순열을 수행하지 않는 방법을 다룬다. 따라서 PIPO의 마스킹 구현 연구 결과[5, 6]에도 그대로 적용 가능하다. 구현 코드(ASM)는 표 3과 같다. 표 3에 제시되어 있는 바와 같이 비트 순열 제거 시 S-Layer(**S**)는 총 12개의 명령어를 사용하지 않아도 된다.

III. PIPO S-Layer 구현 최적화

본 장에서는 단일 S-box $P_{out} \circ S_{main} \circ P_{in} = S$로 구성된 S-Layer의 비트 슬라이스 구현에서 입·출력 비트 순열 (P_{in}, P_{out})을 수행하지 않는 방법을 제시하고 PIPO의 S-Layer 구현을 최적화한다.

그림 2에서 S의 비트 순열 (P_{in}, P_{out})를 추가적으로 표현하면 그림 4와 같다. 해당 표현으로 부터 슬라이스 구현상에서 비트 순열 (P_{in}, P_{out})는 슬라이스 간 위치 변환임을 파악할 수 있다.

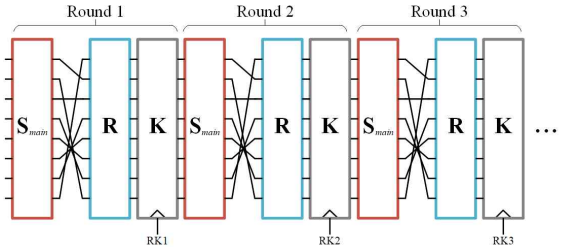

그림 4 PIPO S의 비트 순열 $(P_{in} = I, P_{out})$

* 구현 코드 https://github.com/jeffgyeom/CISC_PIPO_64_128_AVR

3.1 비트 슬라이스 구현의 슬라이스 위치 변환

비트 슬라이스 구현에 있어서 슬라이스 위치 변환은 수행은 재정렬된(re-ordered) **R**, **K**를 통해 다음 라운드로 넘길 수 있게 된다. 예를 들어, 그림 4 첫 번째 라운드의 각 슬라이스 S_i의 *위치*에 맞추어 구현된 R-Layer $\mathbf{R}(S_7,S_6,S_5,S_4,S_3,S_2,S_1,S_0)$ 에 대해,

$$\mathbf{R}(X[1],X[3],X[4],X[5],X[6],X[2],X[0],X[7])$$

를 수행하면 된다. $\mathbf{S}_{main}$, **K**도 동일하게 적용되며, 1회 변경한 함수를 $\mathbf{S}^{(1)}_{main}$, $\mathbf{R}^{(1)}$, $\mathbf{K}^{(1)}$라고 할 때, 결과적으로 그림 4는 그림 5와 같이 표현할 수 있다.

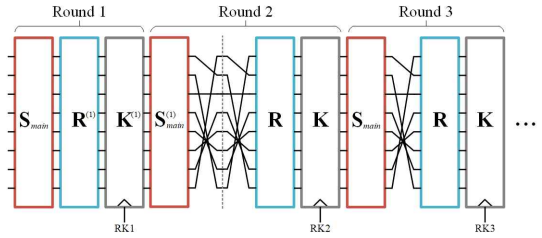

그림 5 PIPO S의 비트 순열 제거 (Round 1)

유사한 방법으로 i회 변경한 함수를 $\mathbf{S}^{(i)}_{main}$, $\mathbf{R}^{(i)}$, $\mathbf{K}^{(i)}$라고 하면, PIPO-64/128의 13 라운드를 슬라이스간 위치 변경 수행없이 구현할 수 있다.

표 1은 PIPO-64/128의 루프 풀기 구현에서 비트 순열을 포함한 방법과 완전히 제거한 방법을 비교한 결과이다. ASM 언어 사용에 있어서 암·복호화 연산 수행 시간을 10% 이상 향상시켰다.

3.2 슬라이스 위치 변환 제거의 제한사항

PIPO S의 비트 순열 제거를 위해 그림 5의 적용은 결과적으로 $\mathbf{S}_{main}$, **R**, **K** 뿐만 아니라 $\mathbf{S}^{(i)}_{main}$, $\mathbf{R}^{(i)}$, $\mathbf{K}^{(i)}$의 구현이 요구된다. PIPO S의 비트 순열 $P_{out} \circ P_{in}$의 위수(order)가 6이기 때문에,

$$\mathbf{S}_{main}=\mathbf{S}^{(6)}_{main},\ \mathbf{R}=\mathbf{R}^{(6)},\ \mathbf{K}=\mathbf{K}^{(6)}$$

를 만족한다. 따라서 6개의 부분 루프 풀기가 요구되며 13(=2×6+1) 라운드의 PIPO-64/128구현을 위해 6(+1)개의 $\mathbf{S}^{(i)}_{main}$, $\mathbf{R}^{(i)}$, $\mathbf{K}^{(i)}$ 구현이 요구된다.

결과적으로 슬라이스 위치 변환 제거는 $P_{out} \circ P_{in}$의 위수에 따라서 암·복호화 코드 사이즈 증가를 야기한다.

【관측】 r 라운드 블록암호의 S-box 비트 순열이 (P_{out},P_{in})이고 $P_{out} \circ P_{in}$의 위수가 $k \le r$이면, 블록암호의 비트 슬라이스 구현을 위해 적어도 $k+(r \bmod k)$개의 $\mathbf{S}^{(i)}_{main}$, $\mathbf{R}^{(i)}$, $\mathbf{K}^{(i)}$ 구현이 요구된다.

IV. PIPO S(R)-Layer 변경

본 장에서는 PIPO S의 비트 순열을 변경하여 기존 PIPO S의 암호학적 성질을 유지하면서 **【관측】** 제시된 부분 루프 풀기에 적합한 PIPO 변형을 제시한다. [5]에서 수행한 바와 같이 각각의 변형 S-Layer에 대해 최적의 R-Layer도 함께 고려되었다.

4.1 PIPO를 위한 변형 S-Layer

PIPO의 $\mathbf{S}_{main}$에 대한 입·출력 비트 순열의 변경은 Differential/Linear Branch Number 차분균일성, 선형성을 포함한 대부분의 암호학적 성질에 영향이 없다. 다만, 고정점($S(x)=x$)의 존재 유무는 다를 수 있다. 이에, 입·출력 비트 순열 변경 시에

조건 1. 고정점이 없음

를 고려하였다. 또한, 비선형 불변 공격과 불변 부분 공간 공격에 대한 저항성을 고려하여

조건 2. $P_{out} \circ \mathbf{S}_{main} \circ P_{in}$의 Cycle-분해에서 Subcycle 개수가 4개 이하*

도 추가로 고려하였다. 마지막으로, **【관측】**에 따라서

조건 3. $P_{out} \circ P_{in}$의 위수가 3 이하

를 고려하여 부분 루프 풀기에 최적이 되도록 하였다.

위의 조건을 검사해야 하는 고려대상 (P_{out},P_{in}) 조합은 $(8!)^2 \approx 2^{30.6}$개가 되나, **【제안】**에 의해 (P_{out},I)만 고려해도 충분하다. (즉, 8!)

【제안】 S-box S의 고정점 존재 여부와 Subcycle 개수는 $P^{-1} \circ S \circ P$와 같다.

조건을 만족하는 변형 (P_{out},I)의 개수는 다음과 같다.

- **조건 1, 2**을 P_{out}의 위수가 2 : 154
- **조건 1, 2**을 P_{out}의 위수가 3 : 239

4.3 변형 S-Layer의 차분/선형 공격 저항성

[5]에서 수행한 바와 같이 각각의 변형 S-Layer에 대해 차분/선형 공격 저항성 측면에서 최적인 R-Layer를 각각 도출하였을 때 비교 결과는 표 2와 같다. P_{out}의 위수가 6에서 2, 3으로 감소하여도 7라운드에서 충분한 차분/선형 공격 저항성을 보일 수 있는 조합이 존재하였다.

표 2 변형 S-Layer의 차분/선형 공격 저항성 비교

구분	P_{out}	R-Layer	7-Round DC/LC	
PIPO	(7, 0, 2, 6, 5, 4, 3, 1)	(0, 7, 4, 3, 6, 5, 1, 2)	$2^{-65.0}$	$2^{-66.0}$
위수2	(2, 7, 0, 3, 4, 6, 5, 1)	(0, 7, 2, 1, 5, 6, 4, 3)	$2^{-65.4}$	$2^{-64.0}$
위수3	(0, 4, 2, 1, 3, 6, 7, 5)	(0, 7, 2, 4, 1, 3, 6, 5)	$2^{-64.4}$	$2^{-64.8}$
	(0, 1, 2, 7, 3, 5, 6, 4)	(0, 6, 7, 4, 1, 2, 3, 5)	$2^{-65.2}$	$2^{-64.0}$
	(6, 0, 7, 2, 4, 5, 1, 3)	(0, 7, 5, 2, 4, 3, 6, 1)		

V. 결론

본 논문은 비트 슬라이스 구현에서 슬라이스 간 위치 변경을 없애는 방법을 제시하였다. 제안 방법은 블록암호 PIPO에 적용되어, 아두이노 Uno 보드에서 암(복)호화 구현 속도를 12%(10%) 향상시켰다. 추가적으로 슬라이스 간 위치 변경을 없애는 구현에 적합한

* PIPO S-box의 Subcycle은 4개다.

변형 PIPO S-Layer를 도출하고 그것들의 차분/선형 공격 저항성을 평가하였다.

[참고문헌]

[1] Biham, Eli. "A fast new DES implementation in software." International Workshop on Fast Software Encryption. Springer, Berlin, Heidelberg, 1997.

[2] REIS, Tiago; ARANHA, Diego F.; LOPEZ, Julio. PRESENT runs fast. In: International Conference on Cryptographic Hardware and Embedded Systems. Springer, Cham, 2017. p. 644-664.

[3] Bao, Z., J. Guo, S. Ling, and Y. Sasaki. "PEIGEN – a Platform for Evaluation, Implementation, and Generation of S-Boxes". IACR Transactions on Symmetric Cryptology, vol. 2019, no. 1, Mar. 2019, pp.

[4] Jean, J., T. Peyrin, S. M. Sim, and J. Tourteaux. "Optimizing Implementations of Lightweight Building Blocks". IACR Transactions on Symmetric Cryptology, vol. 2017, no. 4, Dec. 2017, pp. 130-68.

[5] Kim, H. et al. (2021). PIPO: A Lightweight Block Cipher with Efficient Higher-Order Masking Software Implementations. In: Hong, D. (eds) Information Security and Cryptology – ICISC 2020. ICISC 2020. Lecture Notes in Computer Science(), vol 12593. Springer, Cham.

[6] Kim, Hyunjun, et al. "Masked Implementation of PIPO Block Cipher on 8-bit AVR Microcontrollers." International Conference on Information Security Applications. Springer, Cham, 2021.ger, Berlin, Heidelberg, 2006.

표 3 PIPO의 S-Layer(**S**), R-Layer(**R**) AVR ASM 코드 비교

S(s7, s6, s5, s4, s3, s2, s1, s0), and **R**(s7, s6, s5, s4, s3, s2, s1, s0) AVR ASM Code

- t : Temp Slice
- c2, c1, c0 : Copied Slices

[5]

```
.macro S
 /* S5_1 */
 mov   t,      s6
 and   t,      s7
 eor   s5,     t
 mov   t,      s5
 and   t,      s3
 eor   s4,     t
 eor   s7,     s4
 eor   s6,     s3
 mov   t,      s5
 or    t,      s4
 eor   s3,     t
 eor   s5,      s7
 mov   t,      s6
 and   t,      s5
 eor   s4,     t

 /* S3 */
 mov   t,      s0
 and   t,      s1
 eor   s2,     t
 mov   t,      s1
 or    t,      s2
 eor   s0,     t
 mov   t,      s0
 or    t,      s2
 eor   s1,     t
 ser   t
 eor   s2,     t

 /* XOR */
 eor   s7,     s1
 eor   s3,     s2
 eor   s4,     s0

 /* COPY */
 mov   c0,     s7
 mov   c1,     s3
 mov   c2,     s4

 /* S5_2 */
 mov   t,      s5
 and   t,      c0
 eor   s6,     t
 eor   c0,     s6
 mov   t,      c1
 or    t,      c2
 eor   s6,     t
 eor   c1,     s5
 mov   t,      c2
 or    t,      s6
 eor   s5,     t
 mov   t,      c0
 and   t,      c1
 eor   c2,     t

 /* XOR */
 eor   s2,     c0
 mov   t,      c2
 eor   t,      s1
 mov   c0,     t
 mov   t,      c1
 eor   t,      s0

 /* Bit Permutation */
 mov   s1,     t
 mov   s0,     s7
 mov   s7,     c0
 mov   c1,     s3
 mov   s3,     s6
 mov   s6,     c1
 mov   c2,     s4
 mov   s4,     s5
 mov   s5,     c2
.endm
```

Ours

```
.macro S
 /* S5_1 */
 mov   t,      s6
 and   t,      s7
 eor   s5,     t
 mov   t,      s5
 and   t,      s3
 eor   s4,     t
 eor   s7,     s4
 eor   s6,     s3
 mov   t,      s5
 or    t,      s4
 eor   s3,     t
 eor   s5,      s7
 mov   t,      s6
 and   t,      s5
 eor   s4,     t

 /* S3 */
 mov   t,      s0
 and   t,      s1
 eor   s2,     t
 mov   t,      s1
 or    t,      s2
 eor   s0,     t
 mov   t,      s0
 or    t,      s2
 eor   s1,     t
 ser   t
 eor   s2,     t

 /* XOR */
 eor   s7,     s1
 eor   s3,     s2
 eor   s4,     s0

 /* COPY */
 mov   c0,     s7
 mov   c1,     s3
 mov   c2,     s4

 /* S5_2 */
 mov   t,      s5
 and   t,      c0
 eor   s6,     t
 eor   c0,     s6
 mov   t,      c1
 or    t,      c2
 eor   s6,     t
 eor   c1,     s5
 mov   t,      c2
 or    t,      s6
 eor   s5,     t
 mov   t,      c0
 and   t,      c1
 eor   c2,     t

 /* XOR */
 eor   s2,     c0
 eor   s1,     c2
 eor   s0,     c1

 12 Instructions are
      Removed

.endm
```

```
.macro R
  eor   t,      t
  //s1 = ((s1 << 7)) | ((s1 >> 1));
  bst   s1,     0
  lsr   s1
  bld   s1,     7
  //s2 = ((s2 << 4)) | ((s2 >> 4));
  swap  s2
  //s3 = ((s3 << 3)) | ((s3 >> 5));
  swap  s3
  bst   s3,     0
  lsr   s3
  bld   s3,     7
  //s4 = ((s4 << 6)) | ((s4 >> 2));
  swap  s4
  lsl   s4
  adc   s4,     t
  lsl   s4
  adc   s4,     t
  //s5 = ((s5 << 5)) | ((s5 >> 3));
  swap  s5
  lsl   s5
  adc   s5,     t
  //s6 = ((s6 << 1)) | ((s6 >> 7));
  lsl   s6
  adc   s6,     t
  //s7 = ((s7 << 2)) | ((s7 >> 6));
  lsl   s7
  adc   s7,     t
  lsl   s7
  adc   s7,     t
.endm
```