ARM64 상에서의 LSH 구현

https://youtu.be/E_TkFk38-4Y

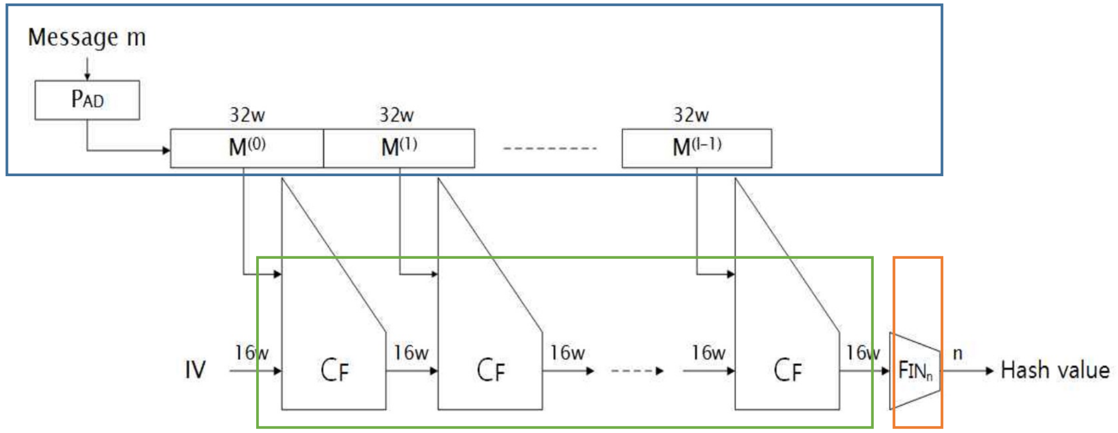
한성대학교
HANSUNG UNIVERSITY

CryptoCraft LAB

LSH(Lightweight Secure Hash)

- 2014년에 개발된 국산 해시 함수

- LSH-8w-n(w 비트 워드 단위로 동작, n 비트의 출력값)
  - w : 32, 64 / 8w : 256, 512
  - $1 \leq n \leq 8w$
    - 224, 256, 384, 512



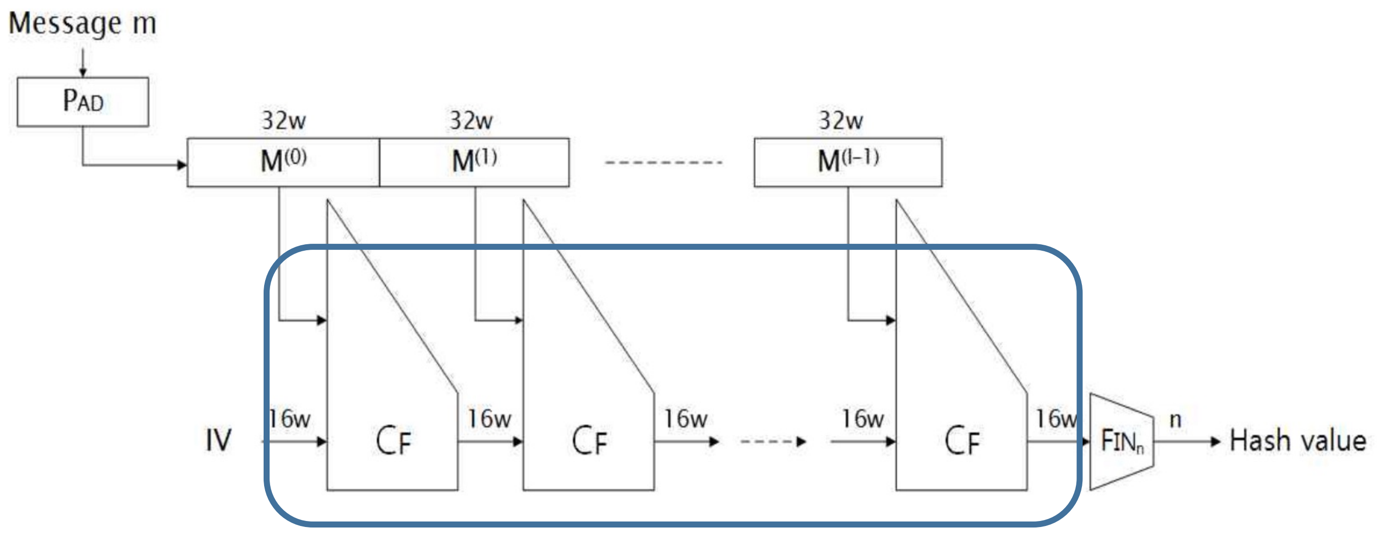
- 3단계를 통해 해시값 출력
  - Initialization(초기화)
    - 입력메시지(m)를 메시지 블록 비트 길이의 배수(1024, 2048)가 되도록 패딩 후, 메시지 블록 단위로 분할
    - 연결 변수를 IV로 초기화
  - Compression(압축)
    - 32w 배열 메시지 블록을 입력으로 사용하여 얻은 출력값을 연결 변수로 사용하여 마지막 블록까지 반복하여 압축 함수 실행
  - Finalization(완료)
    - 연결 변수에 최종 저장된 값으로부터 n 비트 길이의 출력값 생성

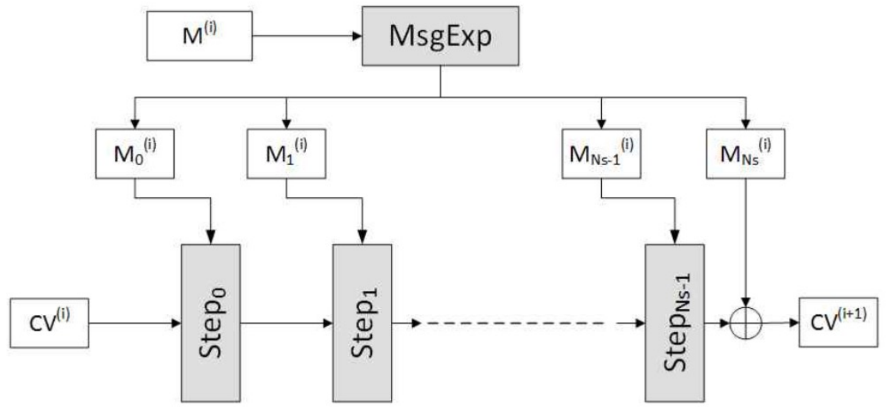
LSH

	n	w	N_s	연쇄 변수 비트 길이	메시지 블록 비트 길이
LSH-224	224	32	26	512	1024
LSH-256	256	32	26	512	1024
LSH-512-224	224	64	28	1024	2048
LSH-512-256	256	64	28	1024	2048
LSH-284	384	64	28	1024	2048
LSH-512	512	64	28	1024	2048

- N_s : Compress 함수 내 Step 함수의 반복 횟수
- $1 \leq n \leq 8w$ ($8w$: 256 or 512)

LSH





Algorithm 2 Compression function CF

```
Input: The i-th chaining variable $\mathsf{CV}^{(i)}$ and the i-th message block $\mathsf{M}^{(i)}$
Output: The $(i+1)$-th chaining variable $\mathsf{CV}^{(i+1)}$
1: $\{\mathsf{M}_j^{(i)}\}_{j=0}^{\mathrm{N}_s} \leftarrow \textsc{MsgExp}(\mathsf{M}^{(i)})$
2: $\mathsf{T} \leftarrow \mathsf{CV}^{(i)}$
3: for $j = 0$ to $\mathrm{N}_s - 1$ do
4:     $\mathsf{T} \leftarrow \textsc{Step}_j(\mathsf{T}, \mathsf{M}_j^{(i)}) \begin{cases} \mathsf{T} \leftarrow \textsc{MsgAdd}(\mathsf{T}, \mathsf{M}_j^{(i)}) \\ \mathsf{T} \leftarrow \textsc{Mix}_j(\mathsf{T}) \\ \mathsf{T} \leftarrow \textsc{WordPerm}(\mathsf{T}) \end{cases}$
5: end for
6: $\mathsf{CV}^{(i+1)} \leftarrow \textsc{MsgAdd}(\mathsf{T}, \mathsf{M}_{\mathrm{N}_s}^{(i)})$
7: return $\mathsf{CV}^{(i+1)}$
```

ARM64 상에서의 LSH 구현

```
.macro lsh256_init  //init_for
    ld1.4s        {v18,v19},[x3], #32
    mix_even
    word_perm
    ld1.4s        {v18,v19},[x3], #32
    mix_odd
    word_perm
.endm
```

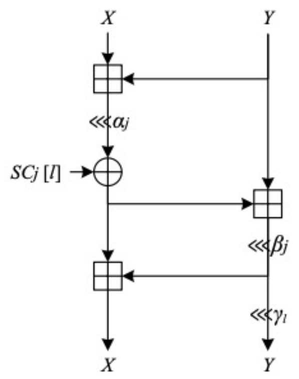
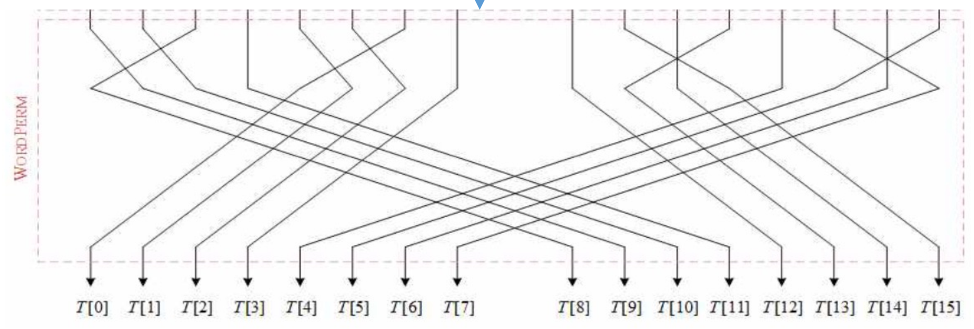


Fig. 1: Two-word mix function $\textsc{Mix}_{j,l}(X,Y)$

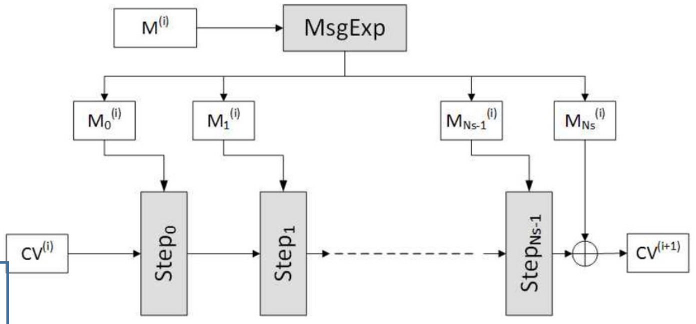
$$
\begin{aligned}
X &\leftarrow X \boxplus Y,\\
X &\leftarrow X^{\lll \alpha_j},\\
X &\leftarrow X \oplus SC_j[l],\\
Y &\leftarrow X \boxplus Y,\\
Y &\leftarrow Y^{\lll \beta_j},\\
X &\leftarrow X \boxplus Y,\\
Y &\leftarrow Y^{\lll \gamma_l}.
\end{aligned}
$$

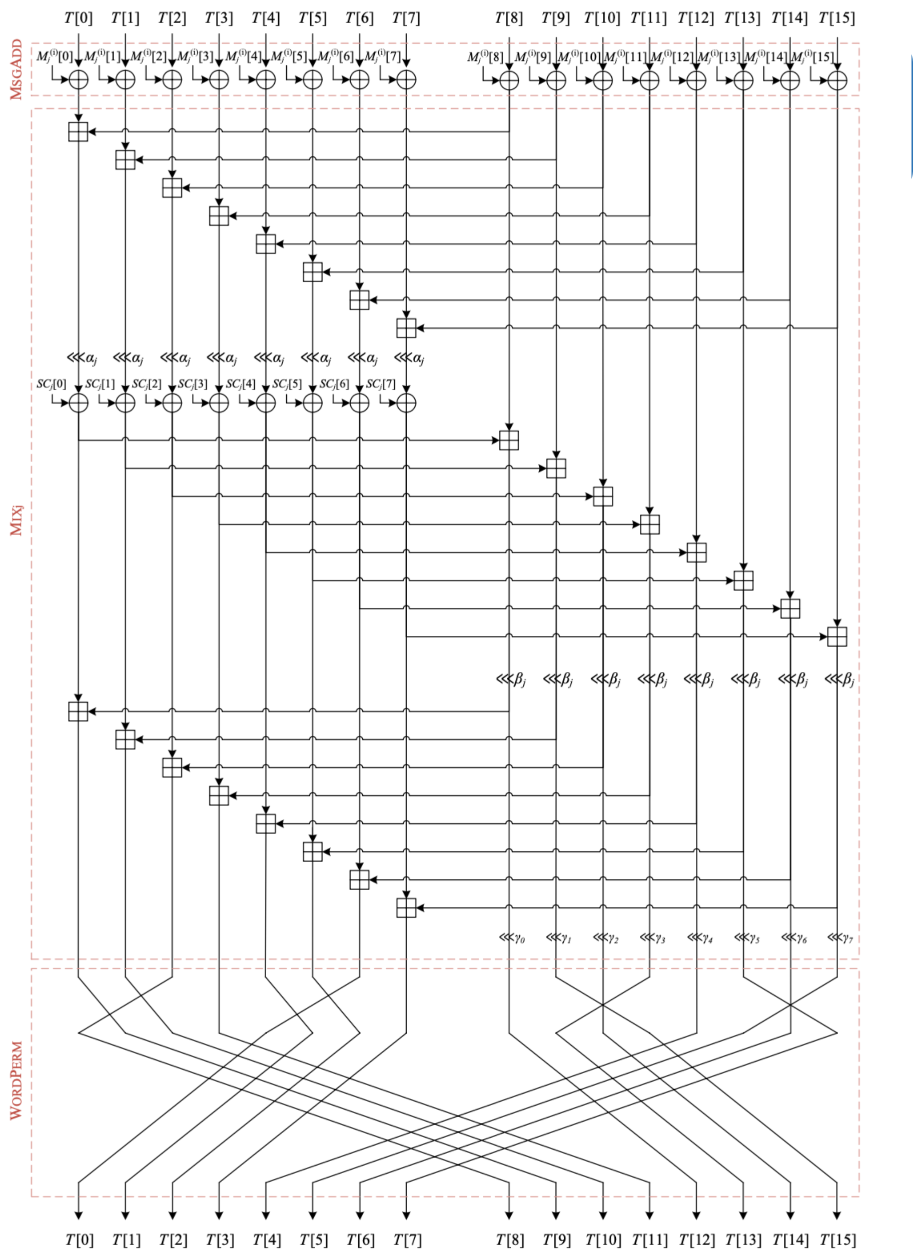


w	j	α_j	β_j	γ_0	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6	γ_7
32	짝수	29	1	0	8	16	24	24	16	8	0
	홀수	5	17								

ARM64 상에서의 LSH 구현

```c
static INLINE void compress(struct LSH256_Context* ctx, const lsh_u32 pdMsgBlk[MSG_BLK_WORD_LEN])
{
    lsh_uint i;
    struct LSH256_internal i_state[1];

    const lsh_u32* const_v = NULL;
    lsh_u32* cv_l = ctx->cv_l;
    lsh_u32* cv_r = ctx->cv_r;

    load_msg_blk(i_state, pdMsgBlk);

    msg_add_even(cv_l, cv_r, i_state);
    load_sc(&const_v, 0);
    mix(cv_l, cv_r, const_v, ROT_EVEN_ALPHA, ROT_EVEN_BETA);
    word_perm(cv_l, cv_r);

    msg_add_odd(cv_l, cv_r, i_state);
    load_sc(&const_v, 8);
    mix(cv_l, cv_r, const_v, ROT_ODD_ALPHA, ROT_ODD_BETA);
    word_perm(cv_l, cv_r);

    for (i = 1; i < NUM_STEPS / 2; i++)
    {
        msg_exp_even(i_state);
        msg_add_even(cv_l, cv_r, i_state);
        load_sc(&const_v, 16 * i);
        mix(cv_l, cv_r, const_v, ROT_EVEN_ALPHA, ROT_EVEN_BETA);
        word_perm(cv_l, cv_r);

        msg_exp_odd(i_state);
        msg_add_odd(cv_l, cv_r, i_state);
        load_sc(&const_v, 16 * i + 8);
        mix(cv_l, cv_r, const_v, ROT_ODD_ALPHA, ROT_ODD_BETA);
        word_perm(cv_l, cv_r);
    }

    msg_exp_even(i_state);
    msg_add_even(cv_l, cv_r, i_state);
}
```





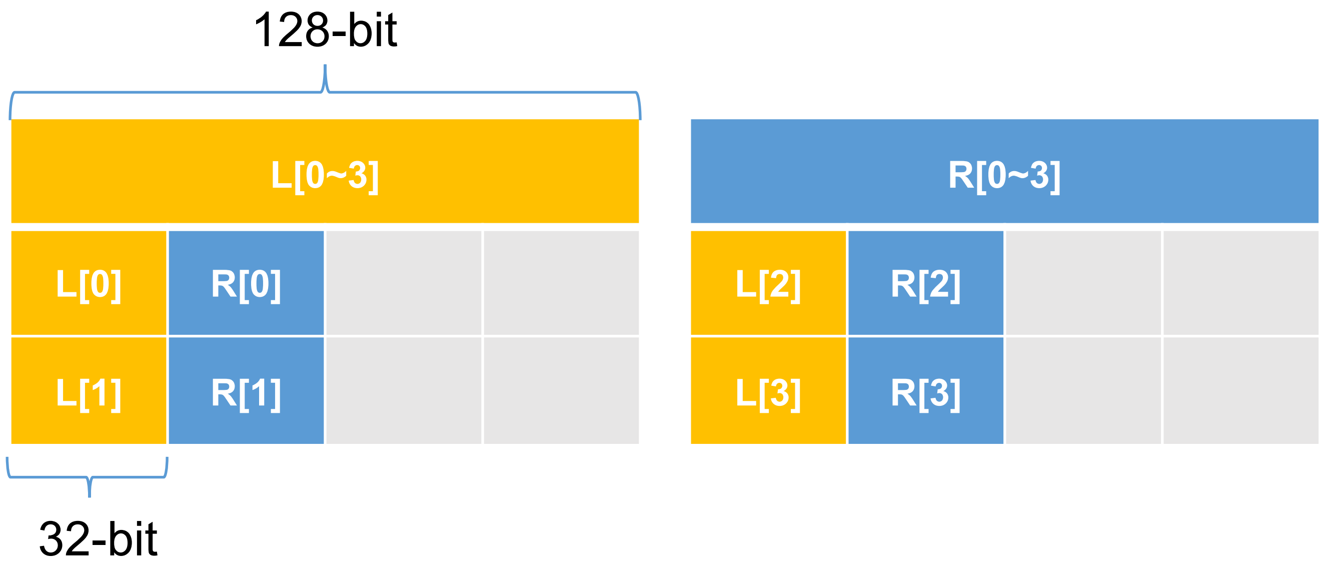
T[0] T[1] T[2] T[3] T[4] T[5] T[6] T[7] T[8] T[9] T[10] T[11] T[12] T[13] T[14] T[15]
MsgAdd
$M_j^{(i)}[0]$ … $M_j^{(i)}[15]$
Mix$_j$
$\lll \alpha_j$
$SC_j[0]$ … $SC_j[7]$
$\lll \beta_j$
$\lll \gamma_0$ … $\lll \gamma_7$
WordPerm

Fig. 2: The j-th step function STEP$_j$

T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]	T[11]	T[12]	T[13]	T[14]	T[15]
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]	L[7]	R[0]	R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]

ARM64 상에서의 LSH 구현

```c
static INLINE void msg_exp_even(struct LSH256_internal * i_state){
    lsh_u32 temp;
    temp = i_state->submsg_e_l[0];
    i_state->submsg_e_l[0] = i_state->submsg_o_l[0] + i_state->submsg_e_l[3];
    i_state->submsg_e_l[3] = i_state->submsg_o_l[3] + i_state->submsg_e_l[1];
    i_state->submsg_e_l[1] = i_state->submsg_o_l[1] + i_state->submsg_e_l[2];
    i_state->submsg_e_l[2] = i_state->submsg_o_l[2] + temp;
    temp = i_state->submsg_e_l[4];
    i_state->submsg_e_l[4] = i_state->submsg_o_l[4] + i_state->submsg_e_l[7];
    i_state->submsg_e_l[7] = i_state->submsg_o_l[7] + i_state->submsg_e_l[6];
    i_state->submsg_e_l[6] = i_state->submsg_o_l[6] + i_state->submsg_e_l[5];
    i_state->submsg_e_l[5] = i_state->submsg_o_l[5] + temp;
    temp = i_state->submsg_e_r[0];
    i_state->submsg_e_r[0] = i_state->submsg_o_r[0] + i_state->submsg_e_r[3];
    i_state->submsg_e_r[3] = i_state->submsg_o_r[3] + i_state->submsg_e_r[1];
    i_state->submsg_e_r[1] = i_state->submsg_o_r[1] + i_state->submsg_e_r[2];
    i_state->submsg_e_r[2] = i_state->submsg_o_r[2] + temp;
    temp = i_state->submsg_e_r[4];
    i_state->submsg_e_r[4] = i_state->submsg_o_r[4] + i_state->submsg_e_r[7];
    i_state->submsg_e_r[7] = i_state->submsg_o_r[7] + i_state->submsg_e_r[6];
    i_state->submsg_e_r[6] = i_state->submsg_o_r[6] + i_state->submsg_e_r[5];
    i_state->submsg_e_r[5] = i_state->submsg_o_r[5] + temp;
}
```



```asm
.macro  msg_exp_even
    mov     v3.s[0], v24.s[0]
    mov     v3.s[1], v26.s[0]
    mov     v9.s[0], v24.s[1]
    mov     v9.s[1], v26.s[1]
    mov     v10.s[0], v24.s[2]
    mov     v10.s[1], v26.s[2]
    mov     v14.s[0], v24.s[3]
    mov     v14.s[1], v26.s[3]

    mov     v15.s[0], v28.s[0]
    mov     v15.s[1], v30.s[0]
    mov     v16.s[0], v28.s[1]
    mov     v16.s[1], v30.s[1]
    mov     v21.s[0], v28.s[2]
    mov     v21.s[1], v30.s[2]
    mov     v23.s[0], v28.s[3]
    mov     v23.s[1], v30.s[3]

    mov.4s     v20, v3
    add.4s     v3, v15, v14
    add.4s     v14, v23, v9
    add.4s     v9, v16, v10
    add.4s     v10, v21, v20

    mov     v24.s[0], v3.s[0]
    mov     v26.s[0], v3.s[1]
    mov     v24.s[1], v9.s[0]
    mov     v26.s[1], v9.s[1]
    mov     v24.s[2], v10.s[0]
    mov     v26.s[2], v10.s[1]
    mov     v24.s[3], v14.s[0]
    mov     v26.s[3], v14.s[1]

    mov     v28.s[0], v15.s[0]
    mov     v30.s[0], v15.s[1]
    mov     v28.s[1], v16.s[0]
    mov     v30.s[1], v16.s[1]
    mov     v28.s[2], v21.s[0]
    mov     v30.s[2], v21.s[1]
    mov     v28.s[3], v23.s[0]
    mov     v30.s[3], v23.s[1]
```

ARM64 상에서의 LSH 구현

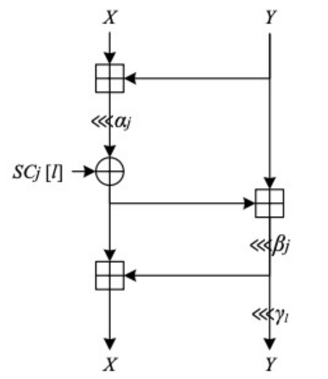


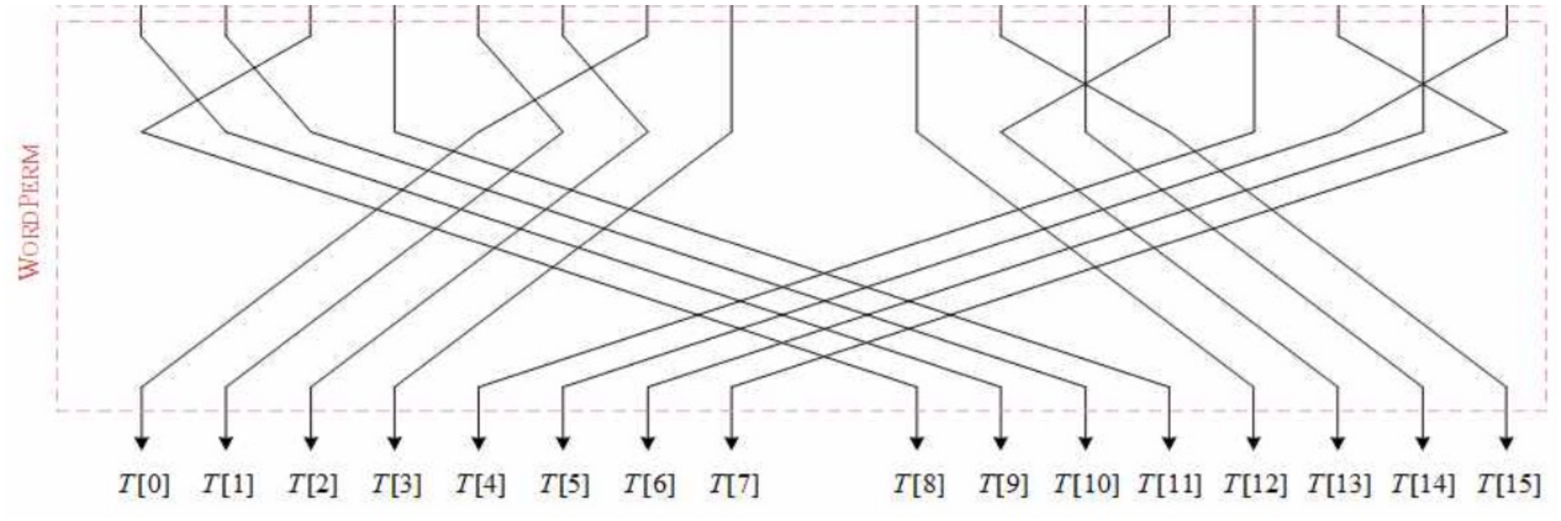
Fig. 1: Two-word mix function $\text{MIX}_{j,l}(X,Y)$

$$\begin{aligned}
X &\leftarrow X \boxplus Y,\\
X &\leftarrow X^{\lll \alpha_j},\\
X &\leftarrow X \oplus SC_j[l],\\
Y &\leftarrow X \boxplus Y,\\
Y &\leftarrow Y^{\lll \beta_j},\\
X &\leftarrow X \boxplus Y,\\
Y &\leftarrow Y^{\lll \gamma_l}.
\end{aligned}$$

w	j	α_j	β_j	γ_0	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6	γ_7
32	짝수	29	1	0	8	16	24	24	16	8	0
	홀수	5	17								

```
.macro mix_even
    add.4s      v5, v5, v7                          //v5 : cvl0~3 / v6: cvl4~7
    add.4s      v6, v6, v8                          //add_blk(cv_l, cv_r);

    shl.4s      v20, v5, #29                        //even_rot_alpha(29, 3)
    sri.4s      v20, v5, #3
    mov.4s      v5, v20

    shl.4s      v21, v6, #29
    sri.4s      v21, v6, #3                         //    rotate_blk(cv_l, rot_alpha);
    mov.4s      v6, v21

    eor         v5.16b, v5.16b, v18.16b
    eor         v6.16b, v6.16b, v19.16b             //    xor_with_const(cv_l, const_v);

    add.4s      v7, v7, v5                          //v7 : cvr0~3 / v8: cvr4~7
    add.4s      v8, v8, v6                          //    add_blk(cv_r, cv_l);

    shl.4s      v20, v7, #1                         //even_rot_beta (1, 31)
    sri.4s      v20, v7, #31
    mov.4s      v7, v20

    shl.4s      v21, v8, #1
    sri.4s      v21, v8, #31                        //    rotate_blk(cv_r, rot_beta);
    mov.4s      v8, v21

    add.4s      v5, v5, v7                          //v5 : cvl0~3 / v6: cvl4~7
    add.4s      v6, v6, v8                          //add_blk(cv_l, cv_r);

    mov         v26.s[0], v7.s[1]
    mov         v26.s[1], v8.s[2]

    mov         v27.s[0], v7.s[3]
    mov         v27.s[1], v8.s[0]

    shl.4s      v20, v26, #8
    sri.4s      v20, v26, #24
    mov.4s      v26, v20

    shl.4s      v21, v27, #24
    sri.4s      v21, v27, #8
    mov.4s      v27, v21

    mov         v7.s[1], v26.s[0]
    mov         v8.s[2], v26.s[1]

    mov         v7.s[3], v27.s[0]
    mov         v8.s[0], v27.s[1]

    mov         v26.s[0], v7.s[2]
    mov         v26.s[1], v8.s[1]
    rev32       v26.8h, v26.8h

    mov         v7.s[2],v26.s[0]
    mov         v8.s[1],v26.s[1]                    //    rotate_msg_gamma(cv_r);

    movi.8h     v14, 0x00
    mov.8h      v26, v14
    mov.8h      v27, v14
.endm
```

ARM64 상에서의 LSH 구현



```
.macro word_perm
    mov         v20.s[0], v5.s[0]           //v20 : tmp
    mov         v5.s[0], v6.s[2]
    mov         v6.s[2], v8.s[2]
    mov         v8.s[2], v7.s[2]
    mov         v7.s[2], v5.s[1]

    mov         v5.s[1], v6.s[0]
    mov         v6.s[0], v8.s[0]
    mov         v8.s[0], v7.s[0]
    mov         v7.s[0], v5.s[2]

    mov         v5.s[2], v6.s[1]
    mov         v6.s[1], v8.s[3]
    mov         v8.s[3], v7.s[1]
    mov         v7.s[1], v20.s[0]
    mov         v20.s[0], v5.s[3]

    mov         v5.s[3], v6.s[3]
    mov         v6.s[3], v8.s[1]
    mov         v8.s[1], v7.s[3]
    mov         v7.s[3], v20.s[0]
.endm
```

전	T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]	T[11]	T[12]	T[13]	T[14]	T[15]
전	L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]	L[7]	R[0]	R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]
후	L[6]	L[4]	L[5]	L[7]	R[4]	R[7]	R[6]	R[5]	L[2]	L[0]	L[1]	L[3]	R[0]	R[3]	R[2]	R[1]
전’	L[0]	L[1]	L[2]	L[3]	L’[0]	L’[1]	L’[2]	L’[3]	R[0]	R[1]	R[2]	R[3]	R’[0]	R’[1]	R’[2]	R’[3]
후’	L’[2]	L’[0]	L’[1]	L’[3]	R’[0]	R’[3]	R’[2]	R’[1]	L[2]	L[0]	L[1]	L[3]	R[0]	R[3]	R[2]	R[1]

Q & A