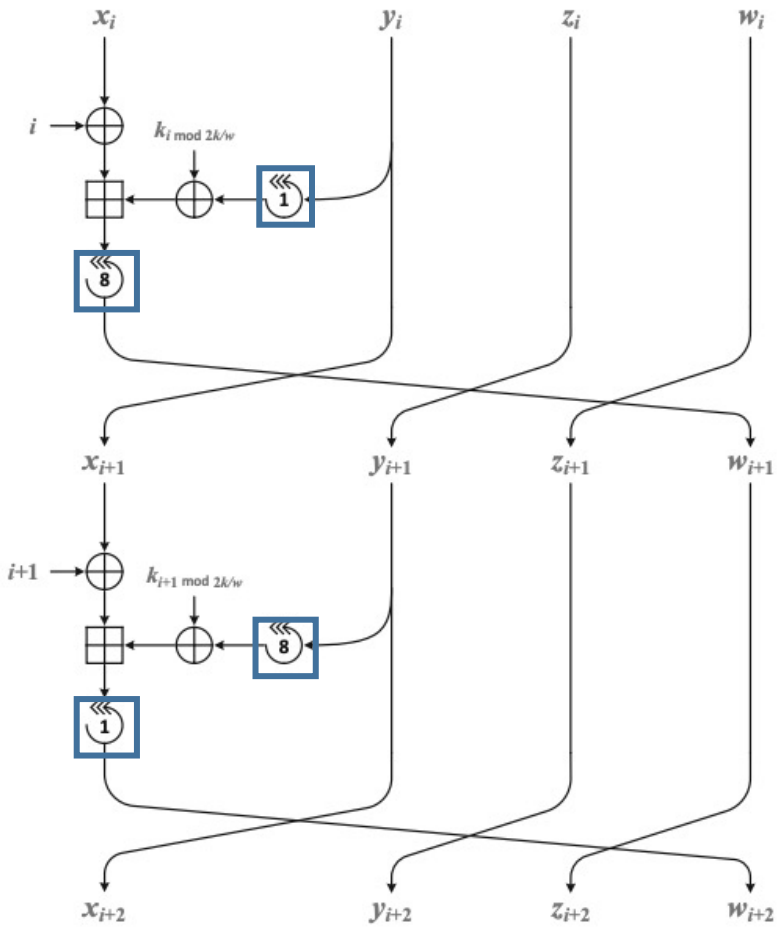
RISC-V 상에서의 CHAM-64/128 구현

https://youtu.be/l73T26Kj_Po

CHAM

- ICISC'17에서 발표된 국산 경량 블록암호
- ARX(Addition, Rotation, XOR) 연산
- Feistel 구조
  - 홀수 라운드에 ROL 연산(1, 8)
  - 짝수 라운드에 ROL 연산 (8, 1)



Cipher	n	k	r → r'(revised)
CHAM-64/128	64	128	80 → 88
CHAM-128/128	128	128	80 → 112
CHAM-128/256	128	256	96 → 120

RISC-V

- 오픈 소스로 제공되는 명령어 셋을 기반으로 한 프로세서

- 32-bit 프로세서인 RV32I에서는 32개의 32-bit 레지스터 제공

X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15
ZERO	RA	SP	GP	TP	T0	T1	T2	S0	S1	A0	A1	A2	A3	A4	A5
X16	**X17**	**X18**	**X19**	**X20**	**X21**	**X22**	**X23**	**X24**	**X25**	**X26**	**X27**	**X28**	**X29**	**X30**	**X31**
A6	A7	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	T3	T4	T5	T6

RA : Return Address Register
SP Stack Pointer Register
GP : Global Pointer Register
TP : Tread Pointer Register

T0~T6 : temporal registers
S0~S11 : saved registers(callee saved)
A0~A7 : function arguments and return value

simon-speck & riscv: https://www.youtube.com/watch?v=MbTFGbFLTbo&list=PLdOq9g7U6Pdt5ZlWeffEU-ViDUS6j-jFS&index=55

RISC-V 명령어

OPCODE	DEST, OP1, OP2	역할
LW	a1, 0(a0)	a0의 주소에 있는 값을 a1에 저장
SW	a1, 0(a0)	a1에 있는 값을 a0 주소에 저장
SLLI	a0, a0, 1	왼쪽으로 쉬프트 1번 한 값을 a0에 저장
SRLI	a0, a0, 1	오른쪽으로 쉬프트 1번 한 값을 a0에 저장
XOR	a0, a0, a1	a0 ^ a1을 a0에 저장
ADD	a0, a0, a1	a0+a1을 a0에 저장
ADDI	a0, a0, 1	a0+1을 a0에 저장
SUB	a0, a0, a1	a0-a1을 a0에 저장
LI	a0, 1	a0에 1을 load
SLTU	a0, a1, a2	a1 < a2 일 경우, a0 은 1 a1 > a2 일 경우, a1 은 0 (캐리 플래그 역할)

RISC-V상에서의 CHAM-64/128

```c
extern void cham(u16 *a, u16 *b);

int main() {

    u16 PT_64[4] = {0x1100, 0x3322, 0x5544, 0x7766};

    u16 RK_64[16] = {0x0301, 0x0705, 0x0b09, 0x0f0d,
                     0x1311, 0x1715, 0x1b19, 0x1f1d,
                     0x151e, 0x0308, 0x3932, 0x2f24,
                     0x4d46, 0x5b50, 0x616a, 0x777c};

    cham(PT_64, RK_64);


}
```

```
//a0 : pt
//a1 : rk

//a3 : x1
//a4 : x2
//a5 : x3
//a6 : x4
//a7 = x1 temp
//t2 , rk
//t3 : rc
//t4 = x2 tmp
//t5 : 0xffff
```

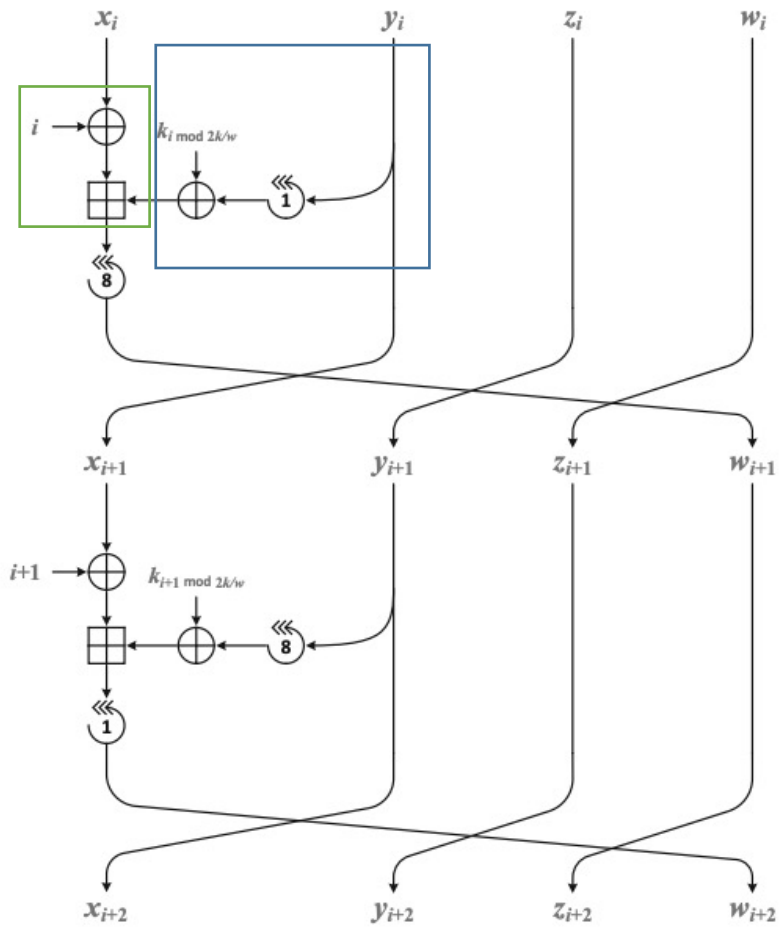
```
cham:
//a0 : pt, a1 : rk, a2 : ct

    lh      a3, 0(a0)   //x1
    lh      a4, 2(a0)   //x2
    lh      a5, 4(a0)   //x3
    lh      a6, 6(a0)   //x4

    li      t3, 0       //t3 = rc
    li      t5, 0xffff

    lh      t2, 0(a1)   //rk[0]
    li      t6, 32

    odd_round
    even_round

    odd_round
    even_round

    odd_round
    even_round

    odd_round
    even_round

////
    odd_round
    even_round

    odd_round
    even_round

    odd_round
    even_round
      ........
    odd_round
    even_round

    lh      t2, 0(a1)   //rk[0]
    sub     a1, a1, t6



    sh      a3, 0(a0)   //x1
    sh      a4, 2(a0)   //x2
    sh      a5, 4(a0)   //x3
    sh      a6, 6(a0)   //x4


    RET
```

해당 과정을 총 11번 반복

RISC-V상에서의 CHAM-64/128

```
.macro odd_round
    mv      t4, a4      //t4 = a4

    slli    t0, t4, 1  //x2 <<1
    srli    t1, t4, 15
    xor     t4, t0, t1
    and     t4, t4, t5

    lh      t2, 0(a1)   //rk[0]
    xor     t4, t4, t2//roundkey xor <<1
    addi    a1, a1, 2

    xor     a3, a3, t3  //rc
    and     a3, a3, t5

    add     a3, a3, t4
    sltu    t0, a3, t4
    add     a3, a3, t0
    and     a3, a3, t5

    slli    t0, a3, 8   //x1<<8
    srli    t1, a3, 8
    xor     a3, t0, t1
    and     a3, a3, t5

    mv      a7, a3  //a7 =temp
    mv      a3, a4
    mv      a4, a5
    mv      a5, a6
    mv      a6, a7

    addi     t3, t3, 1   //t3 = rc

.endm
```

8bit avr의 ADC 역할



risc-v pipo 병렬구현 : https://www.youtube.com/watch?v=mUiw8J32Ag4&list=PLdOq9g7U6Pdt5ZlWeffEU-ViDUS6j-jFS&index=2

RISC-V상에서의 CHAM-64/128

```
.macro odd_round
    mv      t4, a4      //t4 = a4

    slli    t0, t4, 1  //x2 <<1
    srli    t1, t4, 15
    xor     t4, t0, t1
    and     t4, t4, t5

    lh      t2, 0(a1)   //rk[0]
    xor     t4, t4, t2//roundkey xor <<1
    addi    a1, a1, 2

    xor     a3, a3, t3  //rc
    and     a3, a3, t5

    add     a3, a3, t4
    sltu    t0, a3, t4
    add     a3, a3, t0
    and     a3, a3, t5

    slli    t0, a3, 8   //x1<<8
    srli    t1, a3, 8
    xor     a3, t0, t1
    and     a3, a3, t5

    mv      a7, a3  //a7 =temp
    mv      a3, a4
    mv      a4, a5
    mv      a5, a6
    mv      a6, a7

    addi     t3, t3, 1   //t3 = rc

.endm
```

```
.macro even_round
    mv      t4, a4      //t4 = a4

    slli    t0, t4, 8  //x2 <<8
    srli    t1, t4, 8
    xor     t4, t0, t1
    and     t4, t4, t5

    lh      t2, 0(a1)   //rk[0]
    xor     t4, t4, t2//roundkey xor <<8
    addi    a1, a1, 2

    xor     a3, a3, t3  //rc
    and     a3, a3, t5

    add     a3, a3, t4
    sltu    t0, a3, t4
    add     a3, a3, t0
    and     a3, a3, t5

    slli    t0, a3, 1   //x1<<1
    srli    t1, a3, 15
    xor     a3, t0, t1
    and     a3, a3, t5

    mv      a7, a3  //a7 =temp
    mv      a3, a4
    mv      a4, a5
    mv      a5, a6
    mv      a6, a7

    addi     t3, t3, 1   //t3 = rc

.endm
```

Q & A