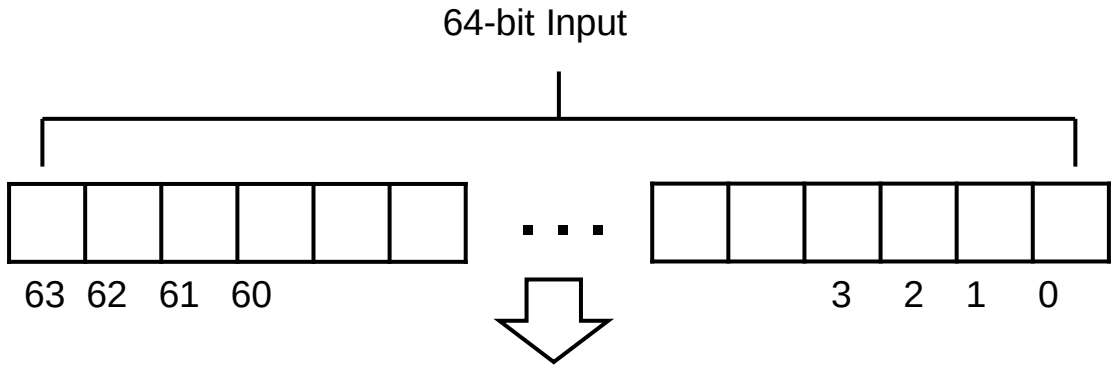
RISC-V PIPO 병렬 구현

https://youtu.be/mUiw8J32Ag4

1. 단일 평문 구현

- PIPO의 블록 길이는 64-bit
- 8-bit 단위로 연산을 함.
- 8개의 레지스터에 나누어 저장



7 ~ 0	7	6	5	4	3	2	1	0	X[0]
15 ~ 8	15	14	13	12	11	10	9	8	X[1]
23 ~ 16	23	22	21	20	19	18	17	16	X[2]
31 ~ 24	31	30	29	28	27	26	25	24	X[3]
39 ~ 32	39	38	37	36	35	34	33	32	X[4]
47 ~ 40	47	46	45	44	43	42	41	40	X[5]
55 ~ 48	55	54	53	52	51	50	49	48	X[6]
63 ~ 56	63	62	61	60	59	58	57	56	X[7]

1. 단일 평문 구현

- 레지스터에 입력문 저장 과정

```
//load pt
lw          a2, 0(a0)
srli        a3, a2, 8
srli        a4, a3, 8
srli        a5, a4, 8
lw          a6, 4(a0)
srli        a7, a6, 8
srli        t0, a7, 8
srli        t1, t0, 8
```

reg0	X[3]	X[2]	X[1]	X[0]
reg1	0	X[3]	X[2]	X[1]
reg2	0	0	X[3]	X[2]
reg3	0	0	0	X[3]
reg4	X[7]	X[6]	X[5]	X[4]
reg5	0	X[7]	X[6]	X[5]
reg6	0	0	X[7]	X[6]
reg7	0	0	0	X[7]

1. 단일 평문 구현

- Addroundkey 과정

```
.macro Addroundkey
    lw          t2, 0(a1)
    lw          t3, 4(a1)
    addi        a1, a1, 8

    xor         a2, a2, t2
    srli        t2, t2, 8
    xor         a3, a3, t2
    srli        t2, t2, 8
    xor         a4, a4, t2
    srli        t2, t2, 8
    xor         a5, a5, t2

    xor         a6, a6, t3
    srli        t3, t3, 8
    xor         a7, a7, t3
    srli        t3, t3, 8
    xor         t0, t0, t3
    srli        t3, t3, 8
    xor         t1, t1, t3
.endm
```

reg0	?	?	?	X[0]
reg1	?	?	?	X[1]
reg2	?	?	?	X[2]
reg3	?	?	?	X[3]
reg4	?	?	?	X[4]
reg5	?	?	?	X[5]
reg6	?	?	?	X[6]
reg7	?	?	?	X[7]

reg	RK[3]	RK[2]	RK[1]	RK[0]
reg	RK[7]	RK[6]	RK[5]	RK[4]

reg		RK[3]	RK[2]	RK[1]
reg		RK[7]	RK[6]	RK[5]

reg			RK[3]	RK[2]
reg			RK[7]	RK[6]

reg				RK[3]
reg				RK[7]

1. 단일 평문 구현

- Slayer 구현

```
.macro S_layer
    //X[5] ^= (X[7] & X[6]);
    and         t5, t0, t1
    xor         a7, a7, t5

    //X[4] ^= (X[3] & X[5]);
    and         t5, a5, a7
    xor         a6, a6, t5

    //X[7] ^= X[4]; X[6] ^= X[3];
    xor         t1, t1, a6
    xor         t0, t0, a5

    //X[3] ^= (X[4] | X[5]);
    or          t5, a6, a7
    xor         a5, a5, t5
```

reg0	?	?	?	X[0]
reg1	?	?	?	X[1]
reg2	?	?	?	X[2]
reg3	?	?	?	X[3]
reg4	?	?	?	X[4]
reg5	?	?	?	X[5]
reg6	?	?	?	X[6]
reg7	?	?	?	X[7]

1. 단일 평문 구현

- Rlayer 구현
  - SLLI, SRLI, OR로 구현

```
//X[1] = ((X[1] << 7)) | ((X[1] >> 1));
andi        a3, a3, 0xff
slli        t6, a3, 7
srli        a3, a3, 1
xor         a3, a3, t6
```

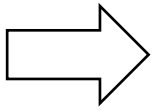
reg0	?	?	?	X[0]
reg1	?	?	?	X[1]
reg2	?	?	?	X[2]
reg3	?	?	?	X[3]
reg4	?	?	?	X[4]
reg5	?	?	?	X[5]
reg6	?	?	?	X[6]
reg7	?	?	?	X[7]

	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	7	6	5	4	3	2	1	0
SLLI 7	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	7	6	5	4	3	2	1	0	0	0	0	0	0	0	0
SRLI 1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	7	6	5	4	3	2	1

2. 4평문 병렬 구현

- 32-bit 를 채워서 사용하기 위해서 평문 4개를 받아서 암호화 진행
- 같은 연산을 진행하는 index를 하나의 레지스터에 저장하여 암호화를 진행한다.
- LBU 명령어, STACK 사용, 레지스터를 활용한 방법으로 내부 정렬

reg0	A[3]	A[2]	A[1]	A[0]
reg1	A[7]	A[6]	A[5]	A[4]
reg2	B[3]	B[2]	B[1]	B[0]
reg3	B[7]	B[6]	B[5]	B[4]
reg4	C[3]	C[2]	C[1]	C[0]
reg5	C[7]	C[6]	C[5]	C[4]
reg6	D[3]	D[2]	D[1]	D[0]
reg7	D[7]	D[6]	D[5]	D[4]



reg0	D[0]	C[0]	B[0]	A[0]
reg1	D[1]	C[1]	B[1]	A[1]
reg2	D[2]	C[2]	B[2]	A[2]
reg3	D[3]	C[3]	B[3]	A[3]
reg4	D[4]	C[4]	B[4]	A[4]
reg5	D[5]	C[5]	B[5]	A[5]
reg6	D[6]	C[6]	B[6]	A[6]
reg7	D[7]	C[7]	B[7]	A[7]

2. 4평문 병렬 구현

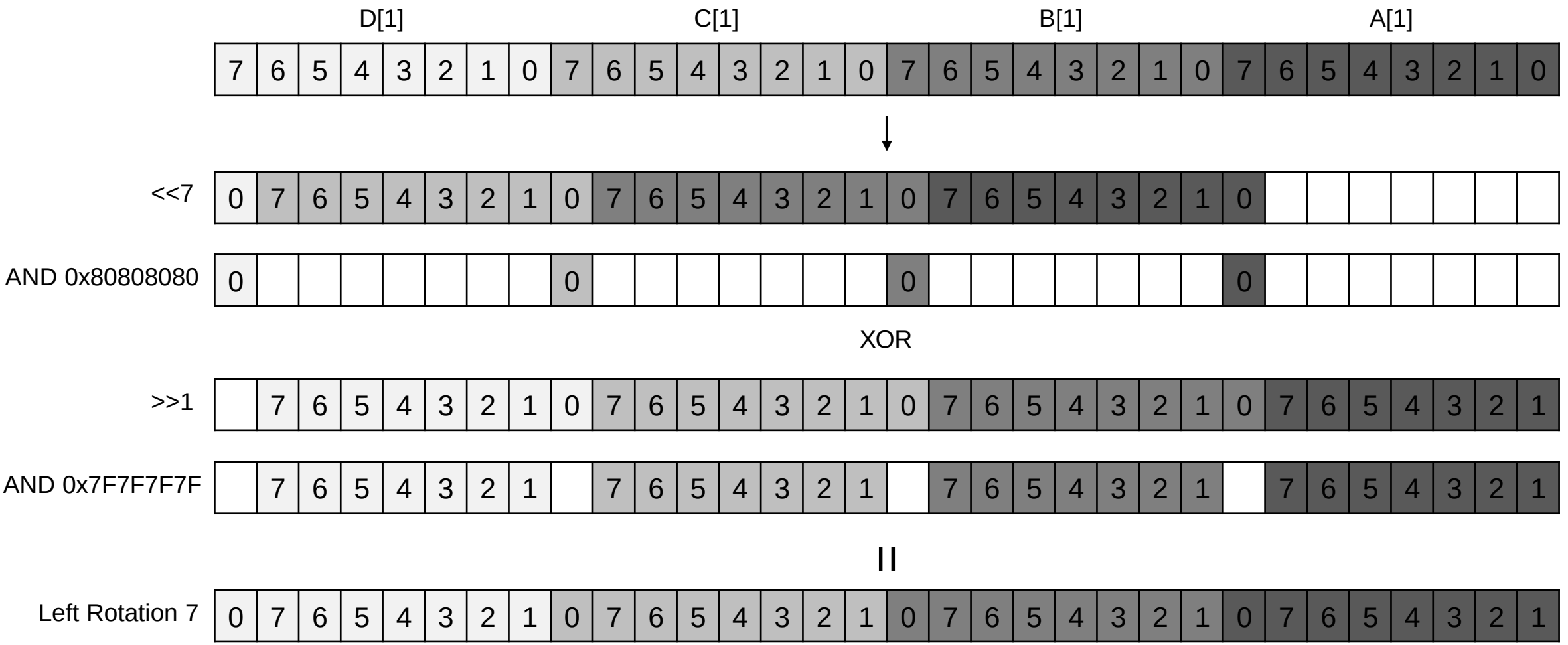
- Slayer 구현

```
.macro S_layer
    //X[5] ^= (X[7] & X[6]);
    and         t5, t0, t1
    xor         a7, a7, t5

    //X[4] ^= (X[3] & X[5]);
    and         t5, a5, a7
    xor         a6, a6, t5

    //X[7] ^= X[4]; X[6] ^= X[3];
    xor         t1, t1, a6
    xor         t0, t0, a5

    //X[3] ^= (X[4] | X[5]);
    or          t5, a6, a7
    xor         a5, a5, t5
```

reg0	D[0]	C[0]	B[0]	A[0]
reg1	D[1]	C[1]	B[1]	A[1]
reg2	D[2]	C[2]	B[2]	A[2]
reg3	D[3]	C[3]	B[3]	A[3]
reg4	D[4]	C[4]	B[4]	A[4]
reg5	D[5]	C[5]	B[5]	A[5]
reg6	D[6]	C[6]	B[6]	A[6]
reg7	D[7]	C[7]	B[7]	A[7]

2. 4평문 병렬 구현

- Rlayer 구현



2. 4평문 병렬 구현

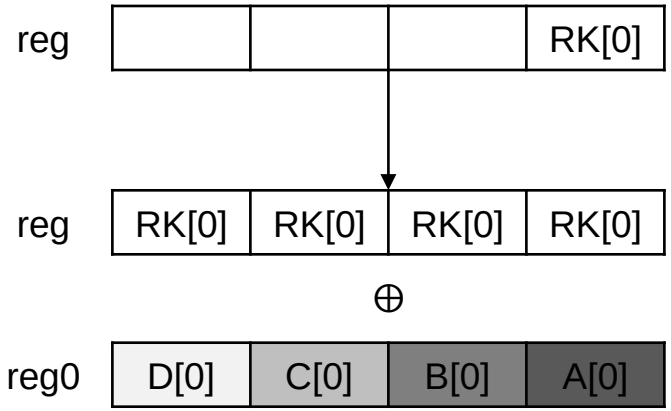
- Addroundkey 과정
  - 메모리 ↓ 속도 ↓ / 메모리 ↑ 속도 ↑

- 메모리 ↓ 속도 ↓
  - 단일 평문과 동일한 키스케줄로 라운드키를 생성
  - 4평문에 맞춰 8-bit 를 32-bit로 늘려주는 작업 필요

- 메모리 ↑ 속도 ↑
  - 8-bit 에서 32-bit로 늘리는 작업을 키스케줄에서 실행
  - 늘려주는 작업이 생략되어 암호화에서는 속도 향상

2. 4평문 병렬 구현

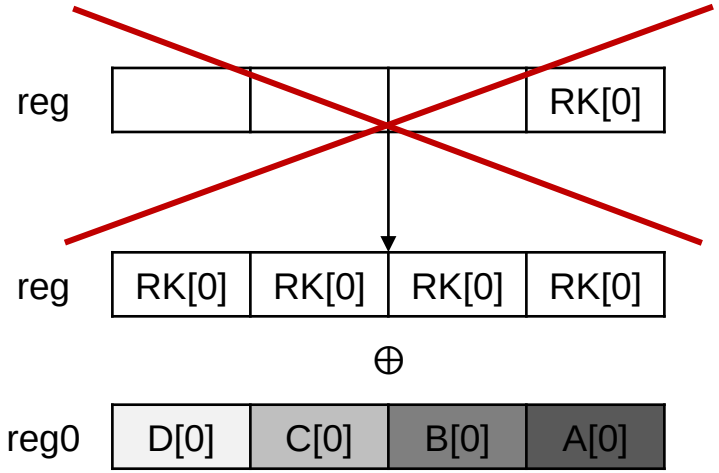
- 메모리↓ 속도↓
- LBU를 통해 1바이트씩 불러와서 32-bit로 늘리고 연산



reg0	D[0]	C[0]	B[0]	A[0]
reg1	D[1]	C[1]	B[1]	A[1]
reg2	D[2]	C[2]	B[2]	A[2]
reg3	D[3]	C[3]	B[3]	A[3]
reg4	D[4]	C[4]	B[4]	A[4]
reg5	D[5]	C[5]	B[5]	A[5]
reg6	D[6]	C[6]	B[6]	A[6]
reg7	D[7]	C[7]	B[7]	A[7]

2. 4평문 병렬 구현

- 메모리↑ 속도↑
- 주소에서 라운드키를 불러와서 바로 연산



reg0	D[0]	C[0]	B[0]	A[0]
reg1	D[1]	C[1]	B[1]	A[1]
reg2	D[2]	C[2]	B[2]	A[2]
reg3	D[3]	C[3]	B[3]	A[3]
reg4	D[4]	C[4]	B[4]	A[4]
reg5	D[5]	C[5]	B[5]	A[5]
reg6	D[6]	C[6]	B[6]	A[6]
reg7	D[7]	C[7]	B[7]	A[7]

3. 성능 결과

- 단일 평문 구현 : 152cpb
- 4평문 병렬 구현 : 85cpb (메모리 절약) , 59cpb (속도 우선)
- 단일 평문 대비 4평문 성능 향상
  - 80% / 157% 성능 향상 확인

Q & A