KIPS Trans. Comp. and Comm. Sys.
Vol.11, No.6 pp.167~174
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2022.11.6.167

Optimized Implementation of PIPO Lightweight Block Cipher on 32-bit RISC-V Processor

Eum Si Woo† · Jang Kyung Bae†† · Song Gyeong Ju† · Lee Min Woo††† · Seo Hwa Jeong††††

ABSTRACT

PIPO lightweight block ciphers were announced in ICISC'20. In this paper, a single-block optimization implementation and parallel optimization implementation of PIPO lightweight block cipher ECB, CBC, and CTR operation modes are performed on a 32-bit RISC-V processor. A single block implementation proposes an efficient 8-bit unit of Rlayer function implementation on a 32-bit register. In a parallel implementation, internal alignment of registers for parallel implementation is performed, and a method for four different blocks to perform Rlayer function operations on one register is described. In addition, since it is difficult to apply the parallel implementation technique to the encryption process in the parallel implementation of the CBC operation mode, it is proposed to apply the parallel implementation technique in the decryption process. In parallel implementation of the CTR operation mode, an extended initialization vector is used to propose a register internal alignment omission technique. This paper shows that the parallel implementation technique is applicable to several block cipher operation modes. As a result, it is confirmed that the performance improvement is 1.7 times in a single-block implementation and 1.89 times in a parallel implementation compared to the performance of the existing research implementation that includes the key schedule process in the ECB operation mode.

Keywords : PIPO, RISC-V, Lightweight Block Cipher, Optimization Implementation

32-bit RISC-V상에서의 PIPO 경량 블록암호 최적화 구현

엄 시 우† · 장 경 배†† · 송 경 주† · 이 민 우††† · 서 화 정††††

요 약

PIPO 경량 블록암호는 ICISC'20에서 발표된 암호이다. 본 논문에서는 32-bit RISC-V 프로세서 상에서 PIPO 경량 블록암호 ECB, CBC, CTR 운용 모드의 단일 블록 최적화 구현과 병렬 최적화 구현을 진행한다. 단일 블록 구현에서는 32-bit 레지스터 상에서 효율적인 8-bit 단위의 Rlayer 함수 구현을 제안한다. 병렬 구현에서는 병렬 구현을 위한 레지스터 내부 정렬을 진행하며, 서로 다른 4개의 블록이 하나의 레지스터 상에서 Rlayer 함수 연산을 진행하기 위한 방법에 대해 설명한다. 또한 CBC 운용모드의 병렬 구현에서는 암호화 과정에 병렬 구현 기법 적용이 어렵기 때문에 복호화 과정에서의 병렬 구현 기법 적용을 제안하며, CTR 운용모드의 병렬 구현에서는 확장된 초기화 벡터를 사용하여 레지스터 내부 정렬 생략 기법을 제안한다. 본 논문에서는 병렬 구현 기법이 여러 블록암호 운용모드에 적용 가능함을 보여준다. 결과적으로 ECB 운용모드에서 키 스케줄 과정을 포함하고 있는 기존 연구 구현의 성능 대비 단일 블록 구현에서는 1.7배, 병렬 구현에서는 1.89배의 성능 향상을 확인하였다.

키워드 : PIPO, RISC-V, 경량 블록암호, 최적화 구현

1. 서 론

현재 암호화 알고리즘의 대부분은 데스크탑과 서버 환경을 위해 설계되었기 때문에 이러한 알고리즘 중 많은 부분이 제한된 환경으로 동작하는 장치에서 사용하기에는 알맞지 않다. NIST(National Institute of Standards and Technology)에서도 2015년에 경량 암호 공모전을 개최하였으며, 이를 통해 제한된 환경에서 사용하기 적합한 여러 경량 암호 알고리즘을 개발하고 있다[1].

PIPO는 ICISC'20에서 발표된 경량 블록암호이다. 11개의 비선형 연산과 23개의 선형 비트 연산을 사용하는 효율적인 Bitslice로 구현되어 있어 효율적인 마스킹 구현이 가능한 특징을 가지고 있다[2].

※ 이 논문은 부분적으로 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No.2018-0-00264, IoT 융합형 블록체인 플랫폼 보안 원천 기술 연구, 30%) 그리고 이 논문은 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2021-0-00540, GPU/ASIC 기반 암호알고리즘 고속화 설계 및 구현 기술개발, 30%) 그리고 이 논문은 2022년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF- 2020R1F1A1048478, 30%). 본 연구는 한성대학교 교내학술연구비 지원과제임.
※ 이 논문은 2021년 한국정보처리학회 ACK 2021의 우수논문으로 "32-bit RISC-V상에서의 경량 블록암호 PIPO 최적 병렬 구현"의 제목으로 발표된 논문을 확장한 것임.
† 준 회 원 : 한성대학교 IT융합공학부 석사과정
†† 준 회 원 : 한성대학교 정보컴퓨터공학과 박사과정
††† 준 회 원 : 한성대학교 IT융합공학부 학사과정
†††† 종신회원 : 한성대학교 IT융학공학부 조교수
Manuscript Received : December 28, 2021
First Revision : January 19, 2022
Accepted : February 17, 2022
* Corresponding Author : Seo Hwa Jeong(hwajeong84@gmail.com)

본 논문은 기존의 연구 내용을 포함한 확장 논문으로[3] 기존 연구의 병렬 구현 기법을 활용하여 해당 기법을 여러 블록암호 운용모드에 적용한 최적화 기법을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 PIPO 경량 블록암호, 32-bit RISC-V 그리고 블록암호 운용모드를 설명한다. 3장에서는 블록암호 운용모드별 구현을 설명한다. 4장에서는 제안 기법을 적용한 구현의 성능을 평가한다. 마지막으로 5장에서는 본 논문의 결론을 내린다.

2. 관련 연구

2.1 PIPO 경량 블록암호

PIPO의 암호화 알고리즘은 SPN(Substitution Permutation Network) 구조를 사용한다. 블록 길이는 64-bit, 키 길이는 128-bit, 256-bit를 지원한다. 키 길이에 따라서 13, 17라운드를 진행하며 암호화가 진행되는 전체적인 알고리즘 구조는 Fig. 1(A)와 같다[2].

PIPO의 암호화 과정에서 내부 데이터의 상태는 Fig. 1(B)와 같은 상태로 암호화가 진행된다. 라운드 키는 입력받은 키를 64-bit 단위로 나누어 사용하게 된다. 예를 들어 키 길이가 128-bit인 경우 64-bit 두 개로 나뉘며 앞 64-bit(RK1), 뒤 64-bit(RK2)일 때, RK2부터 Addroudkey 함수 연산에서 라운드 상수와 XOR된 상태로 사용된다.

SLayer 함수는 Unbalanced-bridge 구조로 23개의 선형 연산과 11개의 비선형 연산으로 구현되어 있다. RLayer 함수는 Byte 단위의 비트 회전만 사용하는 비트 순열로 구현되어 있다. RLayer와 SLayer의 동작은 Fig. 2와 같다.

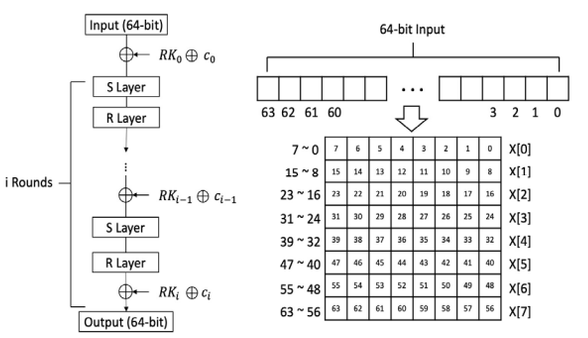

Fig. 1. Structure of PIPO Algorithm(A), Internal Data State(B)

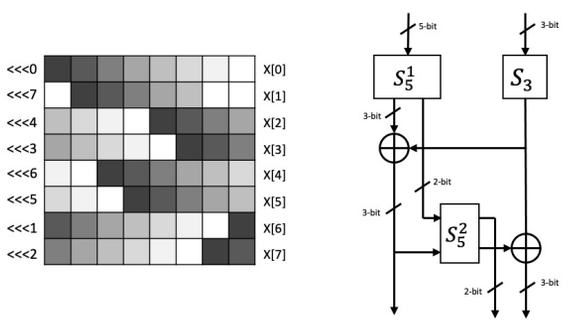

Fig. 2. R layer(Left) and S layer(Right) of Round Functions

2.2 RISC-V Processor

UC 버클리 대학에서 2010년부터 개발중인 RISC(Reduced Instruction Set Computer) 기반의 컴퓨터 아키텍쳐이다[4]. 무료로 공개된 하드웨어 구조와 명령어 집합으로 접근성과 확장성이 좋아 많은 기업과 연구 기관에서 주목하고 있다.

RISC-V는 RV32I, RV64I의 모델이 있으며, 각각 32-bit, 64-bit 레지스터를 사용한다. 본 논문에서는 32-bit를 지원하는 RV32I를 사용한다. RV32I를 사용하는 RISC-V 프로세서는 32-bit 레지스터 32개를 제공하는데 각 레지스터의 용도는 Table 1과 같다[5].

zero(x0) 레지스터는 항상 0의 값을 갖고 있는 레지스터이다. Callee saved 레지스터는 사용하기 이전의 값을 보존하고 사용 후 값을 복원해야 하는 레지스터로 sp(x2)와 s0~s11 레지스터가 이에 해당한다. a0~a7는 함수 인자와 return 값을 갖는 레지스터로 함수 인자는 a0부터 순서대로 값이 저장되어 사용된다.

Table 2는 RISC-V에서 사용되는 기본적인 명령어와 본 논문에서 활용한 명령어를 확인할 수 있다. 명령어 사용은 기본적으로 opcode destination, oprand1, oprand2 순으로 사용된다. 예를 들어 add a0, a1, a2 는 a1 레지스터와 a2 레지스터의 add(덧셈) 연산 결과가 a0 레지스터에 저장된다.

Table 1. Purpose of RISC-V Register

Register	Description	Saver
zero(x0)	Zero register	
ra(x1)	return address	
sp(x2)	stack pointer	callee
gp(x3)	global pointer	
tp(x4)	thread pointer	
a0~a7	function arguments and return value	
so~s11	saved registers	callee
t0~t6	temporal registers	

Table 2. Instruction of RISC-V

Instruction	Description
ADD	Addition
SLLI	Shift Left Logical Immediate
SRLI	Shfit Right Logical Immediate
JAL	Jump and Link
LI	Load Immediate
LW	Load Word
SW	Store Word

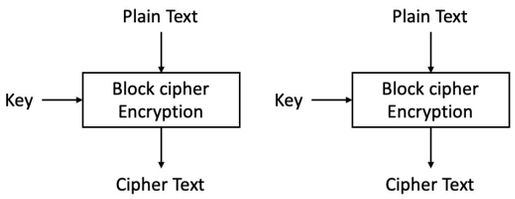

Fig. 3. ECB Block Cipher Modes of Operation

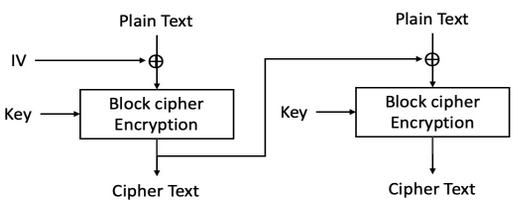

Fig. 4. CBC Block Cipher Modes of Operation

2.3 블록암호 운용모드(ECB, CBC, CTR)

본 장에서는 블록암호의 운용모드 중 ECB(Electronic CodeBlock), CBC(Cipher Block Chaining) 그리고 CTR (Counter)에 대해 설명한다.

1) ECB(Electronic) 운용모드

ECB 운용모드는 운용 방식 중 가장 간단한 구조를 가지며, 암호화 하려는 데이터를 여러 블록으로 나누어 각각 암호화하는 방식이다.

2) CBC(Cipher Block Chaining) 운용모드

CBC 운용모드[6]는 첫 번째 블록의 경우 초기화 벡터(IV) 값이 입력 값에 XOR되어 암호화가 진행되고, 이후의 블록은 초기화 벡터 대신 이전 블록의 암호화 값이 XOR된 값을 암호화한다. CBC 운용모드의 경우 암호화 과정에서 이전 블록의 암호화 값이 사용되기 때문에 병렬화가 어렵지만 복호화의 경우에는 병렬 구현이 가능하다.

3) CTR(Counter) 운용모드

CTR 운용모드[7]는 블록암호를 스트림 암호로 바꾸는 구조를 가진다. 각 블록의 순서를 의미하는 Counter 값이 존재하며, Counter 값과 Nonce 값을 합친 값(IV)을 암호화 입력값으로 사용한다. 암호화 과정을 통해 출력된 값을 평문과 XOR하여 최종적인 암호화 값이 연산된다. CTR 운용모드는 암호화 과정에서 이전 블록에 의존하지 않기 때문에 병렬 구현이 가능하다.

3. 구현 기법

본 장에서는 PIPO 경량 블록암호 단일 블록 구현과 병렬 구현의 각 운용 모드 구현에 관하여 설명한다.

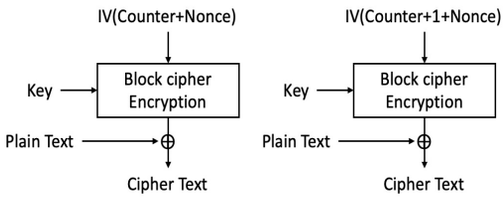

Fig. 5. CTR Block Cipher Modes of Operation

기존 연구인 Kwak et al.[8,9]은 같은 플랫폼인 32-bit RISC-V 프로세서 상에서의 단일 블록 구현과 병렬 구현에서 키 스케줄 과정을 암호화 과정에 포함시켜 구현하였다. 라운드 키를 메모리에 저장하지 않기 때문에 제한된 저사양 디바이스에서 메모리 효율을 높힌 구현이다. 본 논문에서는 키 스케줄 과정을 암호화 과정에 포함시키지 않고 속도 효율을 높힌 구현이다. 임베디드 환경에서 키 갱신은 자주 이루어지지 않기 때문에 키 스케줄을 포함하지 않는 것이 속도 면에서 더 유리하다.

3.1 단일 블록 구현

PIPO 블록암호는 8-bit 단위로 연산이 진행되기 때문에 한 개의 레지스터에 8-bit 값을 저장하여 연산을 진행해야 한다. 즉, 64-bit의 평문을 입력으로 받아 8-bit씩 나누어 8개의 레지스터에 저장한다. 메모리 접근을 최소화하기 위해서 LW 명령어를 사용하여 32-bit 씩 두 번 값을 불러오게 되고, SRLI 명령어를 사용하여 하위 8-bit에 원하는 값이 위치하도록 조정한다. 따라서 연산 과정에서는 32-bit 레지스터의 하위 8-bit만 사용하며, 상위 24-bit에 있는 값은 무시한다. 이는 Fig. 6과 같이 표현할 수 있다. X[i]는 8-bit의 값을 가지며 최종적으로 하위 8-bit에 X[0]~X[7]이 위치하는 것을 확인할 수 있다.

Addroundkey와 Slayer 함수의 연산은 레지스터의 상위 24-bit 값을 무시하고 하위 8-bit 값만 사용하여 연산을 진행할 수 있다. 하지만 Rlayer의 경우 상위 24-bit 값이 연산에 영향을 주게 된다.

RISC-V에서는 로테이션 명령어를 지원하지 않기 때문에 Rlayer의 구현은 Shift 명령어와 OR 명령어를 활용하여 구현한다. 이때 로테이션 구현을 위하여 Logical Shift Right를 하게 되면 상위 24-bit의 값이 오른쪽으로 이동하면서 하

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]

Fig. 6. Plain Text Load Process

Table 3. A Part of Rlayer; (a3:X[1], t6:temp)

Input : X[1] Output : X[1]⟨⟨⟨7
//X[1] = ((X[1] ⟨⟨ 7)) ^ ((X[1] ⟩⟩ 1)); andi a3, a3, 0xff slli t6, a3, 7 srli a3, a3, 1 or a3, a3, t6

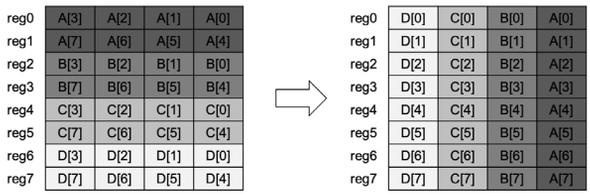

Fig. 7. Internal Alignment of Registers

위 8-bit에 위치하게 된다. 이로 인해 이전 라운드 함수(Addroundkey, Slayer)에서는 상위 24-bit를 무시하고 연산을 진행할 수 있었지만, Rlayer 함수에서는 Logical Shift Right를 하기 전에 상위 24-bit의 값을 0으로 바꾸는 작업을 추가하여야 한다.

Table 3은 Rlayer의 일부 코드이다. 1번 줄에서 a3 레지스터에 저장되어 있는 X[1]의 값과 0xFF의 값을 AND 연산하여 상위 24-bit의 값을 0으로 비운 후, SLLI, SRLI, OR 명령어를 사용하여 로테이션을 구현할 수 있다.

단일 블록 구현에서 ECB, CBC, CTR 운용 모드간 차이는 암호화 과정의 입력값과 출력값에 연산이 추가되는 정도의 작은 차이만 존재하기 때문에 각각의 운용 모드를 나누어 설명하지 않는다.

3.2 병렬 구현

병렬 구현에서는 운용 모드간 병렬 구현의 차이가 있기 때문에 공통적인 병렬 구현 방법과 CBC, CTR 병렬 구현을 나누어 설명한다.

1) 병렬 구현 기법

PIPO는 8-bit 단위로 연산이 진행된다. 이때 32-bit 레지스터에서 단일 구현처럼 8-bit만 사용하는 것은 비효율적이다. 32-bit 레지스터에는 4개의 8-bit 값을 저장할 수 있기 때문에, 4개의 블록을 입력으로 받아 한 번의 암호화 과정을 통해 4개의 블록을 동시에 암호화 하는 병렬 구현 기법을 제안한다. 먼저 병렬 구현의 레지스터 내부 정렬 방법과 각 라운드 함수 구현에 관하여 설명한다.

첫 번째로 병렬 구현을 위해 입력 받은 4개의 블록 중 같은 연산을 진행하는 State를 하나의 레지스터로 합쳐주는 레지스터 내부 정렬이 필요하다. 즉, A~D 평문이 있을 때 각각의 첫 번째 8-bit State에 해당하는 A[0], B[0], C[0], D[0]는 같은 연산을 진행한다. 따라서 각 평문의 State[0]을 하나의 레지스터에 저장한다. 4개 블록이 정렬된 상태는 Fig. 7과 같다.

두 번째로 라운드 함수 중 Rlayer 함수의 경우 단일 블록 구현과 동일한 문제가 발생한다. 로테이션 구현은 Logical Shift Left의 값과 Logical Shift Right의 값을 OR 연산하여 구현하게 되는데, 이때 Logical Shift 하게 될 경우 다른 블록 값에 영향을 주게 된다. 따라서 다른 블록에 영향을 주는 값들을 0으로 바꿔줘야 한다.

Fig. 8은 Rlayer 함수 중 X[1]에서의 로테이션 과정을 나타낸다. Fig. 8에서 볼 수 있듯이 Shift Left를 하게 되면 C[1]의 7~1번째 Bit가 D[1]의 위치로 이동하게 된 것을 볼 수 있다. 따라서 다른 블록에 영향을 주는 값들을 0으로 바꾸어 준다. 이를 위해 Shift Left 한 값을 0x80808080과 AND

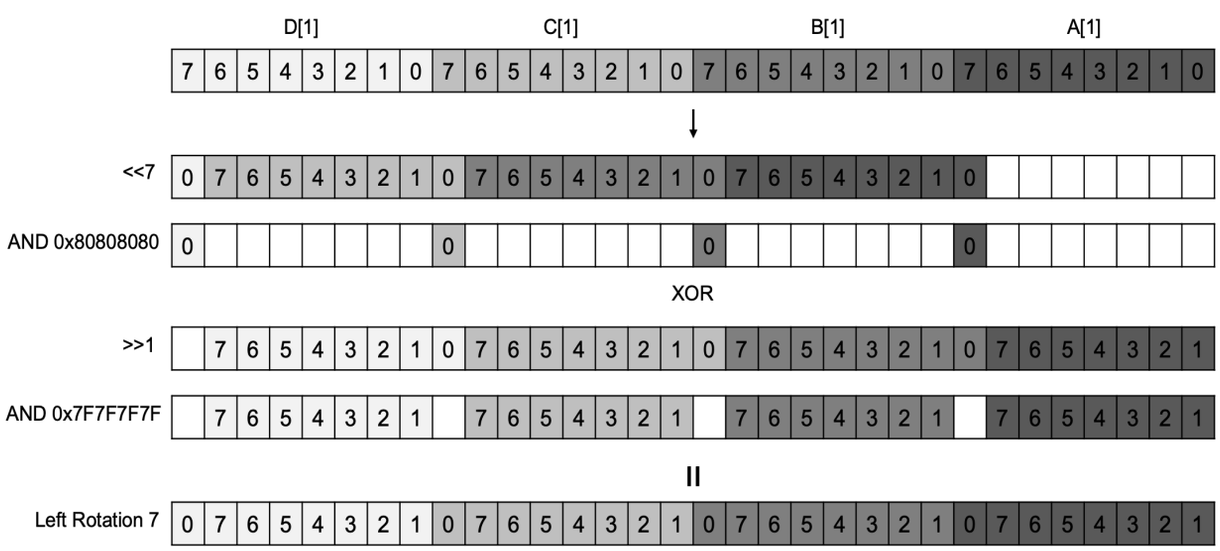

Fig. 8. The Process of Left Rotation 7

Table 4. A part of Rlayer; (a3:X[1], t2,t3,t4:temp)

Input : A[1], B[1], C[1], D[1] Output : (A[1], B[1], C[1], D[1])⟨⟨⟨7
//X[1] = ((X[1] ⟨⟨ 7)) ^ ((X[1] ⟩⟩ 1)); li t3, 0x80808080 li t4, 0x7F7F7F7F slli t2, a3, 7 and t2, t2, t3 srli a3, a3, 1 and a3, a3, t4 or a3, a3, t2

연산을 진행하게 되면 필요한 0번째 Bit만 남게 되고, Shift Right도 동일하게 진행한다. 이를 구현한 코드는 Table 4와 같다.

Table 4의 4,6번 줄은 다른 블록에 영향을 주는 Bit를 0으로 바꿔주는 연산을 진행한다. 이때 1,2번 줄의 t3, t4 레지스터에 저장되는 값은 각 State마다 로테이션 수가 다르기 때문에 로테이션 수에 따라서 저장되는 값이 다르다.

세 번째로 Addroundkey 구현은 메모리 최적화 구현과 속도 최적화 구현 두 가지를 제안한다.

메모리 최적화 구현은 단일 블록 구현과 라운드 키를 저장하는 메모리 크기가 동일하다. 이 경우에 라운드 키를 늘려주는 작업이 추가된다. 64-bit 길이의 라운드 키를 불러오게 되면 각 state에 맞게 연산하기 위하여 8-bit 단위로 나누어야 한다. 그리고 나뉜 8-bit의 RK[i](i=0...7)를 그대로 사용하게 되면 1개의 블록 state만 연산이 되기 때문에 8-bit의 값을 32-bit로 늘려주어야 한다.

속도 최적화 구현은 메모리 최적화 구현에서 라운드 키를 불러온 후에 32-bit로 늘려주는 작업을 키 스케줄에 포함시키는 방법이다. 라운드 키를 32-bit로 늘려주는 작업이 사전 연산되기 때문에 더 빠르게 암호화가 가능하다. 하지만 속도가 빨라지는 대신 라운드 키를 저장하는 메모리를 더 사용해야 하는 단점이 있다. 기존의 라운드 키를 저장하는 공간은 128-bit 키 길이 기준으로 보았을 때, 14개의 64-bit 라운드 키가 저장되어야 한다. 때문에 224-Byte의 저장 공간이 필요하다. 하지만 속도 최적화에서는 4배 늘어난 896-Byte가 필요하다.

AddroundKey 구현의 메모리 최적화와 속도 최적화 구현의 차이는 Fig. 9와 같이 라운드 키를 확장하는 부분의 유무이다.

마지막으로 Slayer 함수의 경우 단일 블록 구현을 그대로 병렬 구현에 사용 가능하다. 단일 블록에서는 상위 24-bit를 무시하고 연산을 했을 뿐, 하위 8-bit나 상위 24-bit나 같은 연산이 수행된다. 병렬 구현에서 상위 24-bit가 사용될 때에도 하위 8-bit와 동일한 연산을 진행하므로, 4개의 블록이 서로 영향 없이 같은 연산을 병렬로 처리할 수 있게 된다.

2) CBC 운용 모드 최적화

CBC 운용 모드의 병렬 구현에서는 최적화 관점을 두 개로 나누어 볼 수 있다.

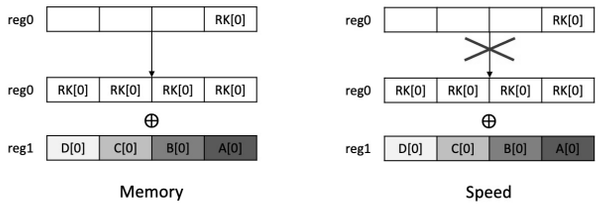

Fig. 9. Memory Optimization AddroundKey(Left) and Speed Optimization AddroundKey(Right)

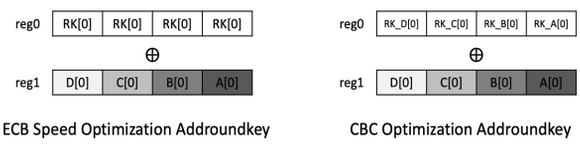

Fig. 10. CBC Block Cipher Modes of Operation

첫 번째는 위에서 설명한 병렬 구현은 ECB 운용모드이다. 이는 4개의 블록이 병렬로 처리될 때, 4개의 블록은 모두 하나의 데이터에서 나누어진 블록이다. 즉 A라는 데이터를 64-bit 단위로 나누어 4개씩 병렬 연산을 한 것이다. 하지만 CBC 운용모드에서는 이전 블록의 암호화 값이 다음 블록의 입력 값에 영향을 주기 때문에 첫 번째 블록이 암호화가 진행되어야 두 번째 블록을 암호화 할 수 있다. 따라서 A라는 데이터의 4개 블록이 아닌 서로 다른 데이터를 병렬 암호화 하는 방법이다. 즉 A, B, C, D 라는 데이터 4개에서 한 개의 블록씩 입력으로 받아서 병렬 암호화하는 방법으로 사용할 수 있다.

첫 번째 최적화 관점을 구현하기 위해서는 4개의 데이터가 같은 키를 사용할 경우, 병렬 구현(3.2.1)과의 차이는 라운드 함수 연산을 진행하기 전에 입력 값에 XOR 연산(첫 번째 블록의 경우 IV, 두 번째 블록 이후는 이전 블록의 암호값)이 추가되는 차이만 존재한다. 하지만 서로 다른 키를 사용할 경우 AddroundKey 함수 구현에서 속도 최적화 구현 기법(3.2.1)을 활용하여 구현한다. 이는 Fig. 9의 속도 최적화 방법과 같이 키 스케줄 과정에 RK[0]를 늘리는 과정을 추가하는 것이 아니라 Fig. 10과 같이 서로 다른 키를 합치는 과정을 추가한다. 이는 사전 연산이 가능하기 때문에 암호화 과정에서는 차이가 발생하지 않는다.

CBC 운용 모드의 두 번째 최적화 관점은 복호화 과정에서의 병렬 구현이다.

Fig. 11은 CBC 운용모드의 복호화 과정이다. CBC 운용모드의 복호화 과정에서는 암호문을 복호화한 값에 XOR하여 평문을 얻을 수 있다. 암호화와 다르게 이전 블록의 영향을 받지 않고, 이미 알고 있는 암호문과 IV값을 사용하기 때문에 4개의 블록을 병렬 처리가 가능하다. 이를 활용하여 암호화 과정에서 여러 블록을 한 번에 병렬 연산을 하는 것이 아니라 복호화 과정에서 여러 블록을 병렬 연산한다. 해당 복호화 과정을 도식화하면 Fig. 12와 같다.

암호문을 입력으로 받아 레지스터 내부 정렬 후 내부 정렬

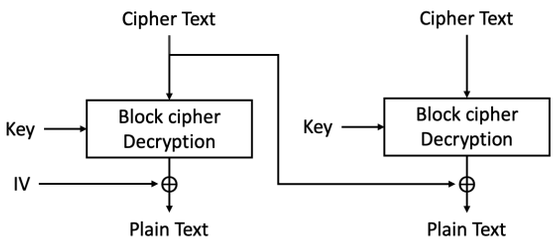

Fig. 11. Decryption Process of CBC Operation Mode

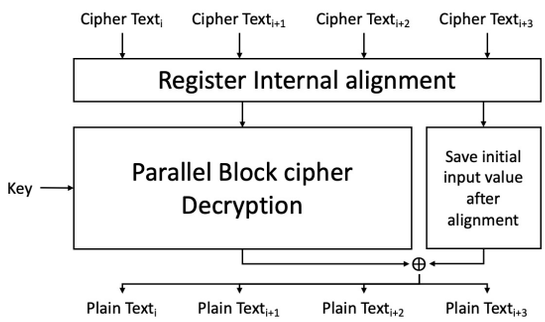

Fig. 12. Parallel Decryption Process of CBC Mode

된 값을 보존한다. 그 후 복호화를 진행하고 복호화된 값과 보존해 놓은 내부 정렬된 암호문을 XOR하는 과정으로 진행된다.

3) CTR 운용 모드 최적화

CTR 운용모드는 변수인 Counter값과 상수인 Nonce값을 사용하여 암호화를 진행한다. 즉 서로 다른 블록의 암호화에 사용되는 IV 값은 Counter 값이 다르다는 차이만 존재한다. 병렬 구현에서는 이러한 특징을 활용하여 최적화를 진행한다.

CTR 운용모드 병렬 구현에서 4개의 블록은 동일한 Nonce 값을 사용한다. 이때 ECB 병렬 구현에서는 입력으로 받은 4개의 블록을 레지스터 내부 정렬을 진행해야 한다. 하지만 CTR 운용모드에서는 Nonce 값이 이미 정해진 값이기 때문에 사전에 내부 정렬된 Nonce 값을 입력으로 받아 사용할 수 있다.

예를 들어 PIPO에서는 64-bit 평문을 입력으로 받는다. 본 논문에서는 Counter 값은 16-bit, Nonce 값은 48-bit으로 정의한다. 기존의 레지스터 내부 정렬을 진행해야 하는 경우, 4개의 블록은 같은 Nonce를 사용하기 때문에 Nonce 48-bit를 8-bit 단위로 나누고 각각을 32-bit로 확장을 해야한다. Counter 값 16-bit도 마찬가지로 8-bit 단위로 나누고 32-bit로 확장을 하면서 +1을 해주면 암호화에 사용될 4개의 IV 값이 레지스터에 저장되게 된다. 본 논문에서는 최적화를 위하여 함수 내부에서 확장될 IV값을 확장된 상태로 사용하는 것을 제안한다. 즉 기존의 Nonce 값은 48-bit 크기를 갖고 있지만, 이를 확장된 상태로 사용하여 192-bit의 Nonce를 사용한다. Counter 값은 16-bit 값 그대로 사용하여 현재 Counter 값을 기준으로 확장하여 사용한다. 이 과정을 도식화하게 되면 Fig. 13과 같다.

4. 성능 평가

본 논문에서는 32-bit RISC-V 프로세서 환경에서 PIPO의 단일 블록 구현과 병렬 구현의 성능을 제시한다. SiFive사의 HiFive1 RevB 보드를 사용하며, SiFive에서 제공하는 FreedomStudio 프레임워크를 사용한다.

Table 5는 제안하는 기법을 적용한 PIPO 암호화 과정에서 발생하는 Cycle과 1-Byte를 암호화 하는데 발생하는 Cycle을 의미하는 cpb(Cycle Per Byte)이다.

단일 블록 구현에서의 성능 측정 결과, ECB 운용 모드에서 1217 Cycle의 성능을 보여준다. 기존 연구 구현[8]에서 키 스케줄 과정을 사전 연산을 통해 생략함으로써 1.7배 성능 향상을 확인하였다. 또한 CBC, CTR 운용 모드에서도 비슷한 성능 향상을 확인할 수 있다.

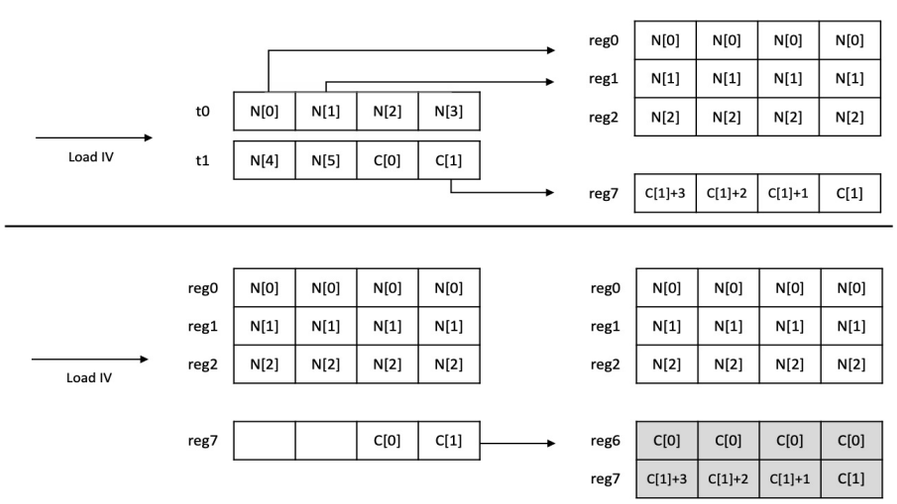

Fig. 13. Original IV used Process(Top), Expansion IV used Process(Bottom)

Table 5. Performance Results in RISC-V

PIPO-64/128	Cycle	Cpb
1-Block Implementation		
Kwak et al. [8]	2078	259.75
Our work(ECB)	1217	152.21
Our work(CBC)	1240	155
Our work(CTR)	1242	155.25
4-Block Implementation		
Kwak et al. [9]	-	113.7
Our work (ECB-Memory opt)	2715	84.85
Our work (ECB-Speed opt)	1918	59.93
Our work(CTR)	1907	59.59
Our work (CBC-Decryption)	1942	60.68

병렬 구현에서는 단일 구현에 비해 Cycle이 높게 측정되지만, 4개의 블록을 병렬 연산하기 때문에 비교를 위해 한번에 암호화되는 데이터의 Byte를 나누어준 cpb를 통해 비교한다. 본 논문에서 제안하는 기법을 적용한 ECB 운용 모드의 병렬 구현 성능 측정 결과 84.85cpb(메모리 최적화), 59.93cpb(속도 최적화)를 보여준다. 이는 단일 구현 대비 1.79배(메모리 최적화), 2.53배(속도 최적화)의 성능 향상을 보여준다. 또한 키 스케줄을 포함하고 있는 기존 연구 구현[9] 대비 1.34배(메모리 최적화), 1.89배(속도 최적화)의 성능 향상을 확인하였다. 단일 구현과 동일하게 CTR 운용 모드에서도 비슷한 성능 향상을 보이며, CBC 운용 모드의 복호화 과정에 해당 기법을 적용하여 비슷한 성능이 측정되는 것을 확인할 수 있다.

5. 결 론

본 논문에서는 32-bit RISC-V 프로세서 상에서 PIPO 경량 블록암호의 단일 블록 구현과 병렬 구현에 대한 연구를 진행하였다. 단일 블록 구현에서는 상위 24-bit 무시한 채로 연산을 진행하고, 상위 24-bit에 영향을 받는 Rlayer 함수에서 상위 24-bit를 0으로 바꾸는 작업을 추가하여 최적화 구현을 진행하였다. 병렬 구현에서는 블록암호 운용모드 별로 최적화 구현에 대한 연구를 진행하였다. 병렬 구현을 위한 레지스터 내부 정렬과 서로 다른 블록에 영향을 주는 Rlayer 함수의 구현 기법을 설명하고, CBC 운용 모드에서는 하나의 데이터를 병렬 구현하는 것이 아니라 서로 다른 데이터를 병렬 구현하는 방법과 복호화 과정에서의 병렬 구현을 제안하였다. CTR 운용모드에서는 레지스터 내부 정렬을 생략하기 위한 확장된 초기화 벡터(IV)를 제안하였다. 본 논문에서 제안하는 기법을 적용하여 성능을 측정한 결과, 키 스케줄을 암호화 과정에 포함시켜 구현한 기존의 연구 대비 단일 구현에서는 1.7배, 병렬 구현에서는 1.89배(속도 최적화) 성능 향상을 확인하였다. 또한 병렬 구현 기법을 ECB 운용모드외에 CBC, CTR 운용 모드에도 큰 성능 차이 없이 적용할 수 있음을 보여준다. 추후 연구로 해당 기법을 통한 다른 경량 블록암호의 최적화 연구를 제안한다.

References

[1] NIST, Lightweight Cryptography: Project Overview [Internet], https ://csrc.nist.gov/projects/lightweight-cryptography, 2021.

[2] H. G. Kim et al., "A new method for designing lightweight S-Boxes with high differential and linear branch numbers and its application," *International Conference on Information Security and Cryptology (ICISC 2020)*, Seoul, Korea, pp.62, 2020.

[3] S. W. Eum, K. B. Jang, G. J. Song, M. W. Lee, and H. J. Seo, "Optimized parallel implementation of Lightweight block-cipher PIPO on 32-bit RISC-V," *Proceedings of the Annual Conference of Korea Information Processing Society Conference (KIPS) 2021*, Vol.28, pp.201-204, 2021.

[4] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Baseuser-level isa," *EECS Department, UCBerkeley, Technical Report, UCB/EECS-2011-62 116*, 2011.

[5] SiFive, Inc. "The RISC-V instruction set manual volume I: User-level ISA document version 2.2," May 7, 2017. [Internet], https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[6] R. Pereira, and R. Adams, "The ESP CBC-mode cipher algorithms," RFC 2451, Nov. 1998.

[7] H. Lipmaa, P. Rogaway, and D. Wagner. "CTR-mode encryption," *First NIST Workshop on Modes of Operation*, Vol.39, 2000.

[8] Y. J. Kwak, Y. B. Kim, and S. C. Seo, "Benchmarking Korean block ciphers on 32-Bit RISC-V processor," *Journal of the Korea Institute of Information Security & Cryptology*, Vol.31, Iss.3, pp.331-340, 2021.

[9] Y. J. Kwak, Y. B. Kim, and S. C. Seo, "Parallel implementation of PIPO block cipher on 32-bit RISC-V processor," *International Conference on Information Security Applications*, pp.183-193, 2021.

엄 시 우

https://orcid.org/0000-0002-9583-5427
e-mail : shuraatum@gmail.com
2021년 한성대학교 IT융합공학부(학사)
2021년 ~ 현 재 한성대학교 IT융합공학부 석사과정
관심분야 : 정보보호, 암호 구현, 인공지능

장 경 배

https://orcid.org/0000-0001-5963-7127
e-mail : starj1023@gmail.com
2019년 한성대학교 IT응용시스템공학부 (학사)
2021년 한성대학교 IT융합공학부(석사)
2021년 ~ 현 재 한성대학교 정보컴퓨터공학과 박사과정
관심분야 : 정보보호, IoT, 양자컴퓨터

송 경 주

https://orcid.org/0000-0001-4337-1843
e-mail : thdrudwn98@gmail.com
2021년 한성대학교 IT융합공학부(학사)
2021년 ~ 현 재 한성대학교 IT융합공학부 석사과정
관심분야 : 정보보호, 암호, 인공지능

이 민 우

https://orcid.org/0000-0002-2356-3055
e-mail : minunejip@gmail.com
2017년 ~ 현 재 한성대학교 IT융합공학부 학사과정
관심분야 : 부채널분석, 자동차보안

서 화 정

https://orcid.org/0000-0003-0069-9061
e-mail : hwajeong84@gmail.com
2010년 부산대학교 컴퓨터공학과(학사)
2012년 부산대학교 컴퓨터공학과(석사)
2012년 ~ 2016년 부산대학교 컴퓨터공학과 (박사)
2016년~2017년 싱가포르 과학기술청 연구원
2019년 ~ 현 재 한성대학교 IT융학공학부 조교수
관심분야 : 정보보호, 암호화 구현, IoT