Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.24, NO.1, Feb. 2014 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2014.24.1.15

GEZEL을 이용한 SEED 및 ARIA 알고리즘 설계 방법*

권 태 웅,† 김 현 민, 홍 석 희‡
고려대학교 정보보호연구원

SEED and ARIA algorithm design methods using GEZEL*

TaeWoong Kwon,† Hyunmin Kim, Seokhie Hong‡
Center for Information Security Technologies, Korea University

요 약

스마트기기를 기반으로 한 사회적, 경제적 활동이 증가함에 따라 다양한 플랫폼에서의 사용자 프라이버시에 대한 안전성과 신뢰성 등의 문제가 대두되고 있다. 이에 따라 정보보호를 목적으로 한 국내 표준 암호 알고리즘들이 개발되었고 이를 다양한 환경에서 얼마나 효율적으로 구현하느냐 또한 중요한 이슈가 되었다. 또한 국내 암호 모듈의 사용이 권장됨에 따라 다양한 환경에서의 SEED와 ARIA의 설계와 구현방식이 논의되고 연구가 되고 있다. SEED와 ARIA는 국내 암호 표준으로써 128비트의 평문을 암호화하며 각각 Feistal, SPN 구조로 이루어져 있는 블록 암호 알고리즘이다. 본 논문은 국내 알고리즘인 SEED와 ARIA를 GEZEL 언어를 이용하여 구현한 첫 논문으로서 GEZEL을 이용한 설계방법과 특징을 구체적으로 설명하고, GEZEL을 이용한 개발의 효율성 및 유연성을 보였다. GEZEL로 구현한 SEED는 69043slice의 면적과 146.25Mhz의 최대 동작 주파수로 동작했고, ARIA는 7282 slice의 면적과 286.172Mhz의 최대 동작 주파수로 동작했다. 또한, SEED는 시그널플로우 방식으로 설계 시 296%가량 속도가 향상되었다.

ABSTRACT

Increasing the smart instrument based social and economical activity, problems of electronic business's safety, reliability and user's privacy are be on the rise. so variety standard cryptography algorithms for information security have been developed in korea and How to efficiently implement them in a variety of environments is issued. ARIA and SEED, developed in Korea, are standard block cipher algorithm to encrypt the 128-bit plaintext, are each configured Feistel, SPN structure. In this paper, SEED and ARIA were implemented using the GEZEL language that can be used easily in the software designer because grammar is simple compared to other hardware description language. In particular, in this paper, will be described in detail the characteristics and design method using GEZEL as the first paper that implements 128bits ARIA and SEED and it showed the flexibility and efficiency of development using GEZEL. SEED designed GEZEL is occupied 69043 slice, is operating Maximum frequency 146.25Mhz and ARIA is occupied 7282 slice, is operating Maximum frequency 286.172Mhz. Also, Speed of SEED designed and implemented signal flow method is improved 296%.

Keywords: ARIA, SEED, GEZEL

접수일(2013년 7월 23일), 수정일(2013년 9월 16일), 게재확정일(2013년 11월 26일)

* 본 연구는 미래창조과학부 및 정보통신산업진흥원의 IT융합 고급인력과정 지원사업의 연구결과로 수행되었음.

† 주저자, taewoong@korea.ac.kr
‡ 교신저자, shhong@korea.ac.kr (Corresponding author)

I. 서 론

스마트폰의 사용이 증가하고 스마트카드 같은 소형 전자 장비를 이용한 전자 거래가 보편화 되면서 이러한 장비들에 대한 안전성 문제가 크게 부각되고 있다. 하지만, 전자 장비들은 점점 소형화되는 반면에 소비자들이 요구하는 기능들은 점점 많아지면서 추가적으로 암호화 모듈을 장비 내에 탑재하기가 설계자들 입장에서는 많은 부담이 되고 있다. 따라서 기존에 전자 장비들을 개발할 때 얼마나 적은 면적으로 구현할지를 고려해야하고, 추가적으로 얼마나 안전하고 효율적인 암호모듈을 탑재해야하는지를 고려해야 한다. 이와 같이 안전하고 효율적이며 적은 면적으로 구현하기 위해서는 기존의 설계 알고리즘을 개선할 수도 있지만, 좀 더 효율적인 설계 언어와 설계 방법을 이용하여 손쉽게 이러한 기능들을 갖춘 소형 전자 장비를 설계할 수 있을 것이다.

안전한 전자 장비를 운용하기 위해서 상용 장비들은 안전성이 검증된 표준 암호 알고리즘을 탑재하고 있다. 이러한 암호 알고리즘들은 그 설계 방법에 따라 소프트웨어 설계, 하드웨어 설계, 그리고 소프트웨어와 하드웨어의 통합 설계 방식 등 크게 세 가지 방식으로 구분이 되고, 장비의 사용 용도와 구현 플랫폼에 따라 적합한 설계 방법을 선택하고 있다.

이 중 소프트웨어와 하드웨어를 모두 이용한 통합 설계 방법은 전체 코드의 유연성과 효율성이 가장 높지만 소프트웨어 프로그래머와 하드웨어 엔지니어서로 간의 분야에 대한 이해의 부족으로 설계에 어려움을 겪는다. 이러한 문제를 해결하고 System C와 Impulse C등의 많은 HDL들이 개발되었으며, 이를 이용하여 소프트웨어와 하드웨어의 통합설계 방법이 보편화 되었다.

Impulse C는 ANSI C 표준을 따르는 언어로, 모든 설계를 C언어만을 사용한다. 기존의 C언어에 대한 지식을 하드웨어 설계를 위해 재사용 할 수 있다는 점이 최대의 장점이나, 하드웨어를 표현, 생성하고, 병렬동작이나 구조적인 설계 등을 위해서는 '형정의' 및 '내장형함수정의'를 위한 라이브러리를 추가하여야 한다는 단점이 있다. 이는 기반이 되는 언어인 C언어가 병렬 동작 보다는 순차적인 실행을 목적으로 구현된 언어이기 때문에 가지는 제약이다. 이러한 제약들이 복잡한 하드웨어들을 설계함에 있어서 하드웨어의 구조적인 특징을 잘 나타내지 못 하기 때문에 System C에 비해 널리 사용되지 못하였다.

System C 역시 Impulse C와 마찬가지로 기존에 널리 사용되던 언어의 문법을 차용하여 기존의 소프트웨어 개발자들도 비교적 간편하게 하드웨어 단독 설계 또는 소프트웨어와의 통합설계가 가능 하도록 개발 되었다. 그러나 접근성이 좋다는 장점에도 불구하고 설계 시 구현 코드 자체가 많이 길어질 뿐 아니라 소프트웨어와 하드웨어의 분할에 있어서 그 용이성이 떨어지고, 하드웨어의 동작 특성을 완벽히 기술하지 못하는 단점을 가지고 있다. 이러한 단점을 극복하기 위하여 암호학자들에 의해서 암호 모듈의 소프트웨어와 하드웨어의 통합 설계에 특화된 GEZEL이란 설계 언어와 이를 이용한 설계 방법이 개발되었다.

GEZEL은 2003년 Patrick R. Schaumont등에 의해서 개발된 언어로 하드웨어 단독 설계뿐만 아니라 하드웨어와 소프트웨어의 통합 설계에도 적합하게 개발된 HDL이다. GEZEL은 암호 알고리즘을 개발하는데 있어서 최적의 형태로 만들어진 언어로 AES, DES, SHA-1, HECC, RIPEMD-160 등 다양한 암호 알고리즘들과 해쉬 암호를 개발하는데 사용되었다[7][9][15-16]. 다른 설계 언어들과 비교해 GEZEL의 가장 큰 장점들은 다음과 같다.

첫째, GEZEL은 합성 시 변수의 타입 선언이 명확하다. VHDL의 공유된 변수들(shared variable)이나 Verilog HDL의 레지스터 타입(reg)은 합성할 때 클럭 신호의 사용 여부 또는 레지스터의 위치에 따라서 다르게 합성이 된다. 하지만 GEZEL의 경우 시그널 타입은 wire로 레지스터 타입은 clocked flip-flop으로 변환되는 타입이 정해져있기 때문에 개발자가 의도하지 않은 래치(latch)가 생기는 등의 문제들을 손쉽게 해결 할 수 있다.

둘째, System C와 마찬가지로 소프트웨어와 하드웨어의 통합설계가 가능한 언어이기 때문에 암호 알고리즘을 개발하는데 있어서 유연성을 제공한다. 프로그램이 완성된 후에도 소프트웨어로 구현된 부분은 유지보수가 쉽고 수정이 간편하며 다른 언어와 호환성이 좋기 때문에 C, JAVA등의 고급언어들과 함께 개발이 가능하다[7][14].

셋째, System C, Verilog HDL, VHDL등의 기존의 HDL에 비해 직관적이고 간편하기 때문에 코드가 짧고 이해하기가 쉬워 코드의 생산성이 좋다.

마지막으로 GEZEL로 작성된 코드는 별도의 툴 없이도 자동으로 VHDL로 변환이 가능하기 때문에 다양한 플랫폼에서 개발이 가능하다.

이와 같은 장점 때문에 [8]에서는 기존의 결과들과

제안하는 방법을 소프트웨어만으로 개발한 결과와 GEZEL을 이용한 소프트웨어와 하드웨어의 통합설계 결과를 비교하여 GEZEL을 이용한 설계 방법의 우수성을 나타냈으며, [9]에서도 8051 마이크로프로세서에서 GEZEL을 코프로세서로 이용하여 HECC를 설계한 결과를 시뮬레이션을 통해 GEZEL의 우수성을 검증하였다. 또한 [10]에서는 GEZEL을 이용하여 공개키 암호를 소프트웨어와 하드웨어의 통합설계를 통해 개발하여 그 성능을 높이는 방법을 소개하였다. [11]에서는 블록 암호 AES를 GEZEL을 통해 개발하는 방법에 대해서 소개하여 하드웨어 단독 설계에 있어서도 다른 언어에 비해 설계가 간단함을 보였다. 또한, [7][11-13]에서 GEZEL을 이용하여 하드웨어적으로 설계한 AES와 소프트웨어와 하드웨어의 통합 설계 방법으로 구현한 다른 IP들의 결과가 기존의 HDL을 사용하여 구현한 결과들과 비교하였을 때 GEZEL을 통한 설계방법이 얼마나 효율적인지를 증명하였다. 이처럼 많은 논문들에서 GEZEL의 우수성과 효율성이 검증되었고, 외국에서는 그 활용도가 높음에도 불구하고 국내에서는 아직 GEZEL을 이용한 설계가 이루어지고 있지 않다.

본 논문은 국내에서 GEZEL을 이용한 첫 하드웨어 구현 결과이고 System C등의 기존의 소프트웨어와 하드웨어 통합설계에 이용하는 언어들이 가지고 있는 단점을 극복하고 이를 통해 좀 더 효과적이고 효율적인 암호 알고리즘에 대한 개발 방법론을 제안한다. 제안하는 방법에 사용한 암호 알고리즘은 국내 표준인 SEED와 ARIA 알고리즘으로 국내에서 사용되는 대표적인 블록 암호이다. 서로 다른 구조를 갖는 두 알고리즘을 제안하는 방법으로 구현함으로써 GEZEL을 이용한 국산 암호 알고리즘 개발의 가능성과 그 효율성을 보이고자 한다.

GEZEL을 이용한 SEED와 ARIA 암호 알고리즘의 구현은 동작속도와 처리량에 초점을 맞춰서 구현하였다. 따라서 SEED와 ARIA의 각 연산별로 데이터패스를 만들고 각각 한 클럭에 동작할 수 있도록 하였으며 각 연산 사이에 레지스터를 두어 파이프라이닝 방식으로 동작하도록 구현하였다. ARIA의 경우 3가지 연산 중 라운드키 덧셈과 확산계층 두 가지 연산을 한 클럭에 처리가 가능하도록 구현하였으며 치환계층은 테이블 치환 방법을 사용하여 구현하였다. SEED의 경우에는 G함수를 데이터패스로 구현하고 F함수에 사용되는 Exclusive-or와 모듈러 덧셈의 결과를 레지스터에 저장을 함으로써 각 연산별로 파이프라이닝 방식으로 동작하도록 구현을 하였다. 또한 각 알고리즘에 사용되는 키스케쥴은 암호화 연산과 별도로 구현하고 on-the-fly방식으로 키를 생성하여 키 생성 시간이 전체 암호 알고리즘 동작시간에 영향을 미치지 않도록 하였다. 이렇게 구현한 결과 SEED는 146.25Mhz, ARIA는 286.17Mhz로 동작하였다.

본 논문의 구성은 다음과 같다. 2장에서는 국내 표준 암호 알고리즘인 SEED와 ARIA에 대해 소개한다. 3장에서는 본 논문에서 사용한 GEZEL 언어에 대한 소개와 특징에 대해서 설명하고, GEZEL을 이용한 SEED와 ARIA 알고리즘에 대한 효과적인 구현 방법에 대해서 구체적으로 보인다. 4장에서는 앞장에서 구현한 SEED와 ARIA를 8051프로세서와의 통합설계 방법을 언급하며 비교하고 5장에서 결론을 맺는다.

II. SEED 및 ARIA 알고리즘

본 장에서는 GEZEL을 이용하여 구현한 국내 표준 블록 암호 알고리즘인 SEED와 ARIA 알고리즘에 대해서 간단히 설명한다.

2.1 SEED 알고리즘

SEED는 국내기술로 개발한 Feistel구조의 128비트 블록 암호 알고리즘이다. 본 절에서는 Fig.1. 에 나타낸 SEED의 기본 구조에 대해서 설명하며 사용되는 표기들은 다음과 같다.

- $G(x)$: 입력 x에 대한 G함수의 출력
- $\oplus$: 배타적 논리합 연산(XOR)
- $\boxplus$: 법 2^{32}에서의 덧셈
- $\boxminus$: 법 2^{32}에서의 뺄셈
- $x \gg n$: x를 오른쪽으로 n비트씩 순환이동
- $x \ll n$: x를 왼쪽으로 n비트씩 순환이동
- $x \| y$: x와 y의 연접
- RK_i : i라운드 암·복호화키

2.1.1 F 함수

Feistel 구조를 갖는 블록 암호 알고리즘은 F함수의 특성에 따라 구분될 수 있다. SEED의 F함수는 Fig.2. 와 같이 64비트의 입출력을 갖는다. F함수는 각 32비트 블록 2개(C, D)를 입력으로 받아, 32비트

블록 2개(C′, D′)를 출력한다. 즉, 암호화 과정에서 64비트 블록(C, D)와 64비트 라운드키 $RK_i = (RK_{i,0} \| RK_{i,1})$를 F함수의 입력으로 처리하여 64비트 블록(C′,D′)를 출력한다.

- $C' = G[G[G\{(C \oplus RK_{i,0}) \oplus (D \oplus RK_{i,1})\} \boxplus (C \oplus RK_{i,0})] \boxplus G\{(C \oplus RK_{i,0}) \oplus (D \oplus RK_{i,1})\}] \boxplus G[G\{(C \oplus RK_{i,0}) \oplus (D \oplus RK_{i,1})\} \boxplus (C \oplus RK_{i,0})]$
- $D' = G[G[G\{(C \oplus RK_{i,0}) \boxplus (D \oplus RK_{i,1})\} \boxplus (C \oplus RK_{i,0})] \boxplus G\{(C \oplus RK_{i,0}) \boxplus (D \oplus RK_{i,1})\}]$

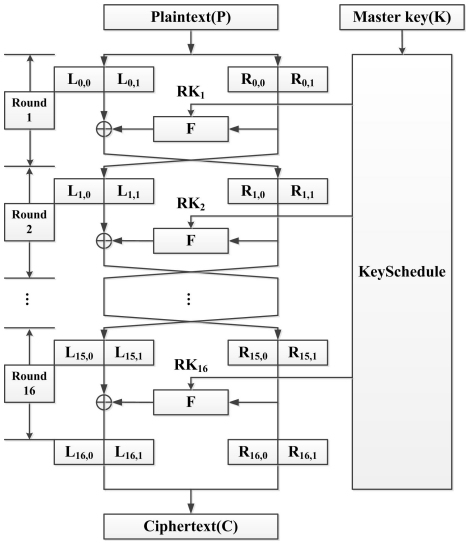


Fig.1. SEED's Encryption process

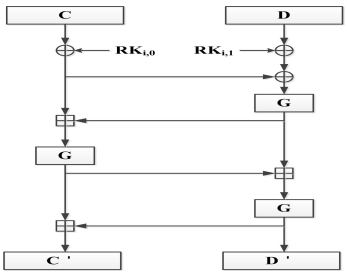


Fig.2. F function of SEED

2.1.2 G 함수

SEED의 G함수는 32비트의 입출력을 갖는 함수로 4개의 S박스와 확산계층으로 이루어져있다. G함수의 입력을 $A = (a_3 \| a_2 \| a_1 \| a_0)$이라 할 때 a_0부터 a_3은 각각 S박스의 입출력으로 사용된다. G함수에 사용되는 S박스는 2종류로 교대로 사용된다. 즉, a_0과 a_2는 s_1에 a_1와 a_3은 s_2의 입력으로 사용된다. 치환된 결과는 확산계층의 입력 값으로 사용되고, 확산계층은 32×32의 이진행렬이다. 전체 G함수의 연산은 다음과 같고, Fig.3. 은 G함수의 구조도를 나타내었다.

- $m_0 = 0xfc, m_1 = 0xf3, m_2 = 0xcf, m_3 = 0x3f$
- $Y_3 = S_2(a_3),\ Y_2 = S_1(a_2),\ Y_1 = S_2(a_1),\ Y_0 = S_1(a_0)$
- $b_3 = (Y_0 \& m_3) \oplus (Y_1 \& m_0) \oplus (Y_2 \& m_1) \oplus (Y_3 \& m_2)$
- $b_2 = (Y_0 \& m_1) \oplus (Y_1 \& m_2) \oplus (Y_2 \& m_3) \oplus (Y_3 \& m_0)$
- $b_1 = (Y_0 \& m_2) \oplus (Y_1 \& m_3) \oplus (Y_2 \& m_0) \oplus (Y_3 \& m_1)$
- $b_0 = (Y_0 \& m_0) \oplus (Y_1 \& m_1) \oplus (Y_2 \& m_2) \oplus (Y_3 \& m_3)$

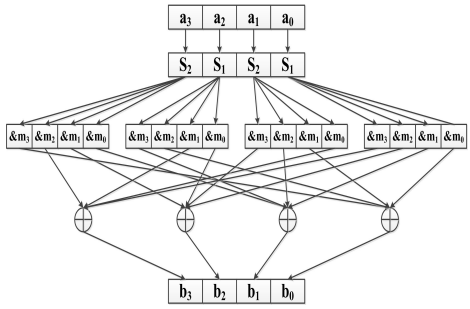


Fig.3. G function of SEED

2.1.3 키 스케줄링

SEED의 키스케줄은 Fig.4. 와 같이 128비트 비밀키 $K = (A, B, C, D)$를 64비트씩 나누어 이들을 교대로 8비트 로테이션 연산을 수행한 후, 결과의 4워드들에 대한 간단한 산술 연산과 G함수를 적용하여 라운드키를 생성한다. 먼저, A와 C를 법2^{32}에서의 덧셈 연산 후 다시 각 라운드 상수 KC와 다시 뺄셈 연산한다. 그 결과를 G함수의 입력으로 하여 처음 32비트 라운드키를 생성하고, B와 D를 법2^{32}에서의 뺄셈 연산 후 라운드 상수 KC와 덧셈 연산 한 후 G함수 입력으로 넣어 하위 32비트 라운드키를 생성한다. 또한 각 32비트 레지스터 A, B, C, D는 홀수 라운드에서는 앞의 A, B의 연접한 결과를 8비트 로테이션 연산을 수행하고, 짝수 라운드에서는 C, D의 연접을 8비트 로테이션 연산이 수행된다.

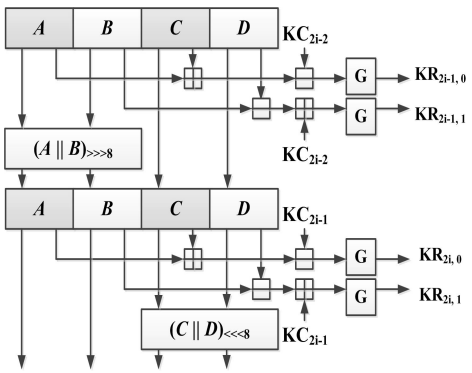


Fig.4. SEED's key scheduling

2.2 ARIA

ARIA는 Fig.5. 에서 나타낸 것처럼 ISPN 구조의 128비트 블록 암호 알고리즘이다. 128, 192, 256 비트 세 가지 길이의 마스터키를 선택적으로 사용할 수 있으며 그에 따라 라운드 수도 12, 14, 16라운드로 구성이 된다. ARIA는 유한체 안에서의 곱셈 연산을 사용한 치환과 확산 그리고 라운드키를 더하는 연산으로 이루어져 있다. 본 논문에서 사용되는 표기들은 다음과 같다.

- $S(x)$: 입력 x에 대한 S-박스의 출력
- $P(x)$: 입력 x에 대한 확산 함수의 출력
- $\oplus$: 배타적 논리합 연산(XOR)
- $x \gg n$: x를 오른쪽으로 n비트씩 순환이동
- $x \ll n$: x를 왼쪽으로 n비트씩 순환이동
- $x \parallel y$: x와 y의 연접
- F_o : 홀수 라운드 함수
- F_e : 짝수 라운드 함수
- F_l : 마지막 라운드 함수
- ek_i : i라운드 암호화키
- dk_i : i라운드 복호화키

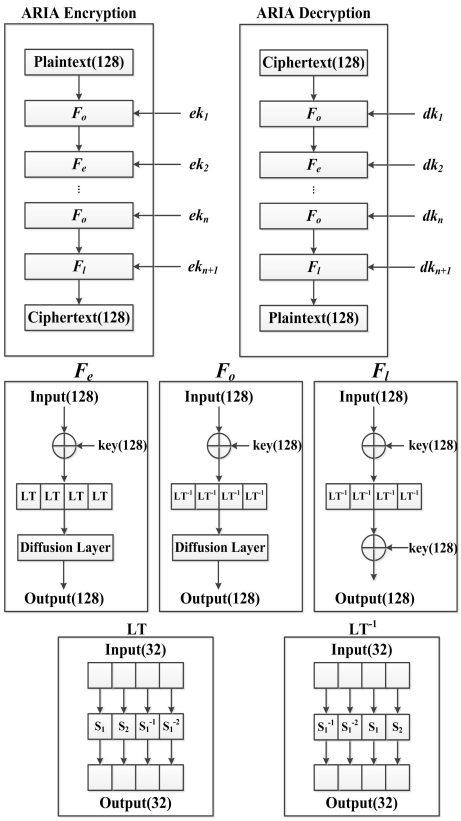


Fig.5. ARIA's Encryption and Decryption process

2.2.1 치환 계층(Substitution Layer)

치환 계층은 입력 값에 대응되는 S-box에 내부 상태 값을 출력해주는 비선형 대치 연산이다. ARIA의 경우 S_1, S_2, S_1^{-1}, S_2^{-1} 4개의 S-box가 존재한다. S_1^{-1}, S_2^{-1}는 각각 S_1, S_2의 역 치환 연산이다.

2.2.2 확산 계층(Diffusion Layer)

확산 계층은 16개의 state 값들을 행렬 곱 연산을 통하여 뒤섞는 변환이다. 확산계층에서는 involution 성질($A^{-1}=A$)을 만족하는 16×16 이진 행렬 A를 사용한다. 확산 함수의 입력 값을 $(a_0,a_1,\cdots,a_{15})$라 하고, 출력 값을 $(b_0,b_1,\cdots,b_{15})$라 하면,

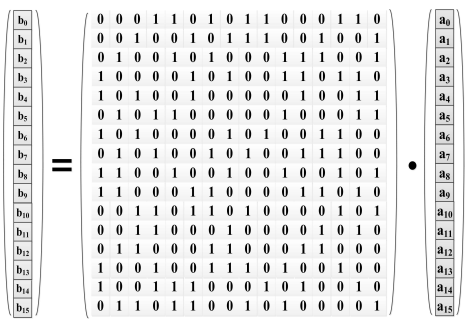


Fig.6. ARIA's Diffusion Layer

A가 고정이므로 $(a_0,a_1,\cdots,a_{15})$가 어떤 값이 들어오더라도 $(b_0,b_1,\cdots,b_{15})$을 구성하는 연산은 정해져있다. Fig.6. 은 확산 계층에 사용하는 행렬을 나타내었다.

2.2.3 키 스케줄링

ARIA의 마스터키 길이는 128, 192, 256비트가 될 수 있다. 입력된 마스터키를 128비트씩 나눠 KL, KR라하고 KR의 부족한 비트는 0으로 패딩한다. 즉, 128비트와 192비트의 마스터키가 입력된 경우 각각을 0으로 128, 64비트씩 패딩하여 KR을 만들어준다. KL, KR를 이용하여 128비트의 W_0, W_1, W_2, W_3 생성하며 그 과정은 다음과 같다.

- $W_0 = KL$
- $W_1 = F_o(W_0 \oplus CK_1) \oplus KR$
- $W_2 = F_e(W_1 \oplus CK_2) \oplus KL$
- $W_3 = F_o(W_2 \oplus CK_3) \oplus W_1$

Fig.7. 은 마스터키를 이용하여 W_0, W_1, W_2, W_3를 생성하는 과정을 보인 것이다. 생성된 W_0, W_1, W_2, W_3는 라운드키를 생성하는 키요소로 사용된다. 라운드에 따라 비트 순환이동과 XOR연산을 수행하여 라운드키를 생성한다. Fig.8. 은 라운드키를 생성하는 과정을 나타낸 것이다.

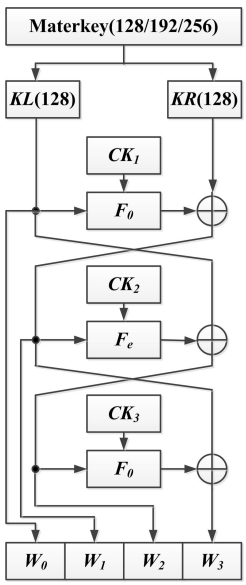


Fig.7. keyfactor generation block diagram

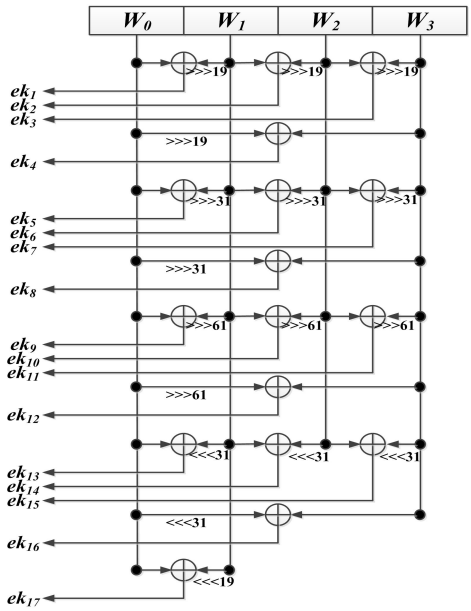


Fig.8. ARIA's key scheduling

III. GEZEL을 이용한 SEED 및 ARIA 설계

3.1 GEZEL

GEZEL의 주요 개발 목적 중 하나는 소프트웨어와 하드웨어의 통합설계에 있다. 따라서 소프트웨어 설계에 익숙한 설계자들이 쉽게 하드웨어 설계를 할 수 있도록 GEZEL은 문법 자체에서 C언어와 유사한 문법을 적용함에도 불구하고 하드웨어적으로 병렬처리가 가능하도록 되어있다. 이와 같이 GEZEL은 소프트웨어 설계에 익숙한 설계자들도 HDL 언어보다 간단하고 쉽게 하드웨어 설계에 접근할 수 있도록 한다. GEZEL과 비교해서 C/C++등의 소프트웨어 개발자가 하드웨어를 개발하는데 있어서 자신들의 경험을 살리지 못 하는 데에는 소프트웨어와 하드웨어 설계에 크게 다음과 같은 세 가지 차이점이 존재하기 때문이다.

첫째, C/C++등의 소프트웨어 개발에서는 순차적인(sequential) 설계방식을 이용하기 때문에 시간지연(delay)에 관한 개념이 존재하지 않는 반면 하드웨어에서는 각 모듈에 따라 시간지연이 존재하기 때문에 이를 고려하여 설계해야만 효율적으로 동작할 수 있다. 시간지연에 대한 개념이 부족한 소프트웨어 설계자들의 경우에 하드웨어 언어를 이용하여 설계 시 이러한 시간지연 문제에 대한 고려가 부족하여 자주 경

쟁효과(race condition)로 인한 하드웨어 동작상의 문제를 겪게 된다.

둘째, 동시성(concurrency)의 문제가 소프트웨어에는 존재하지 않는다. 소프트웨어는 기본적으로 프로그램이 기술된 순서대로 작업을 수행하는 반면 하드웨어는 모듈들이 동시에 동작을 수행하기도 한다. 다시 말해서 소프트웨어는 가장 먼저 수행되는 엔트리 포인트부터 순차적으로 소스코드를 수행해 나가는 반면, 하드웨어는 각 모듈별로 코드를 작성하고 특정한 모듈들이 프로그램 수행 또는 시뮬레이션 시 동시에 동작하는 방식으로 이루어진다. 예를 들면, 암호알고리즘에서 키 생성 부분과 라운드 수행 부분을 따로 구현할 시 소프트웨어에서는 키 생성을 먼저 수행하고 이후 암호화 알고리즘을 수행하거나 키 생성과 암호화 알고리즘을 교대로 수행시키는 반면 하드웨어는 키 생성과 암호화 알고리즘을 동시에 동작시켜 전체 수행시간을 줄일 수 있다. 이러한 동시성을 이해하지 못한 소프트웨어 설계자가 HDL을 사용하여 암호 알고리즘을 구현할 때 비효율적인 코드를 생산할 가능성이 높다.

마지막으로 소프트웨어와 하드웨어에서 사용되는 데이터 타입이 서로 완벽히 호환이 되지 않기 때문이다. 디지털 신호는 0과 1, high-impedance 값으로 표현된다. 이 중 high-impedance값은 소프트웨어에서는 표현할 방법이 없기 때문에 이러한 값을 갖는 변수의 타입에 대한 이해가 부족한 것이 소프트웨어 개발자들이 하드웨어 설계를 진행할 때 어려움을 겪게 한다. 하지만, GEZEL은 소프트웨어 설계자들이 하드웨어 설계 시 겪는 위와 같은 문제를 해결할 수 있게 만들어진 언어이다.

GEZEL은 사이클 기반의 하드웨어 기술 언어로 데이터패스를 이용한 유한 상태 머신(FSMD) 방식으로 구현한다. 데이터패스를 구현한 뒤 이를 동작 순서에 따라 입출력과 컨트롤러를 구현하는 방식을 FSMD이라고 한다. 이러한 데이터패스 또는 컨트롤러를 구현하는 방법이 기존의 HDL들보다 직관적이기 때문에 하드웨어 설계를 처음 접하는 소프트웨어 개발자들도 손쉽게 개발에 이용할 수 있다. 이외에도 GEZEL은 다음과 같은 큰 장점을 가진다.

- 사이클 기반의 언어로 클럭과 리셋 신호가 존재하지 않는다. VHDL이나 Verilog HDL에서 사용되는 변수들은 합성할 때 그 클럭과의 관계, 용도에 따라서 다르게 합성이 될 수 있지만 GEZEL의 경우 시그널 타입(wire)은 wire로 reg타입은 clocked flip-flop으로 변환되는 등 합성되는 데이터 타입이 정해져있다. 이는 개발자의 의도와 항상 정확하게 일치하여 합성이 되기 때문에 의도하지 않은 latch가 생기는 등의 합성 시 발생할 수 있는 예기치 못한 문제들을 해결 할 수 있다.

GEZEL

```
dp updown(out a : ns(4)){
        reg c : ns(4);
        sfg inc { c = c + 1;
                  a = c;  }
        sfg dec { c = c - 1;
                  a = c;  }
}

fsm ctl_updown(updown){
        initial s0;
        state   s1;
        @s0 if (c < 10) then  (inc) -> s0;
              else  (dec) -> s1;
        @s1 if (c >  0) then  (dec) -> s1;
              else  (inc) -> s0;
}
```

SystemC

```
SC_MODULE(fsm_counter) {
 sc_in <bool> clk;
 sc_in <sc_uint<2> > flags_counter;
 sc_out<sc_uint<3> > ins_counter;
 sc_signal<int> state, state_next;
 void eval_logic();
 void update_regs();
 SC_CTOR(fsm_counter) {
  SC_METHOD(eval_logic);
  sensitive << flags_counter << state;
  SC_METHOD(update_regs);
  sensitive_pos(clk);
  state = state_next = 0;
 }
};

void fsm_counter::eval_logic() {
 sc_uint<3> flags = flags_counter.read();
 switch(state) {
  case 0:
   if (flags[0]) {
    state_next = 1;
    ins_counter.write(c_do_dn | c_do_io);
   } else {
    state_next = 0;
    ins_counter.write(c_do_up | c_do_io);
   }
  break;
  case 1:
   if (flags[1]) {
    state_next = 0;
    ins_counter.write(c_do_up | c_do_io);
   } else {
    state_next = 1;
    ins_counter.write(c_do_dn | c_do_io);
   }
  break;
 }
}
void fsm_counter::update_regs() {
 state = state_next;
}
```

```
const int counter_do_io = 1;
const int counter_do_up = 2;
const int counter_do_dn = 4;
SC_MODULE(dp_counter) {
 sc_in <bool> clk;
 sc_in <sc_uint<3> > ins_counter;
 sc_in <sc_uint<2> > ud;
 sc_out<sc_uint<3> > a;
 sc_out<sc_uint<2> > flags_counter;
 sc_signal<sc_uint<3> > c, c_next;
 sc_signal<sc_uint<2> > u, u_next;
 sc_signal<sc_uint<3> > nc;
 void eval_logic();
 void update_regs();
 SC_CTOR(dp_counter) {
  SC_METHOD(eval_logic);
  sensitive << c << nc << ud;
  SC_METHOD(update_regs);
  sensitive_pos(clk);
  c = c_next = 0; u = u_next = 0;
 }
};

void dp_counter::eval_logic(){
 sc_uint<3> sfg = ins_counter.read();
 if(sfg & counter_do_io) {
  u_next = ud.read();
  a.write(nc);
  flags_counter.write(u);
 }
 if (sfg & counter_do_up) {
  nc = c.read() + 1;
  c_next = nc;
 }
 if (sfg & counter_do_dn) {
  nc = c.read() - 1;
  c_next = nc;
 }
}
void dp_counter::update_regs() {
 u = u_next;
 c = c_next;
}
```

Fig.9. Compare 'Counter' design using GEZEL and System C

- GEZEL은 결정적(deterministic) 특성을 갖는다. GEZEL은 소프트웨어와 하드웨어의 통합설계를 염두하고 만든 언어이기 때문에 소프트웨어와 하드웨어간의 데이터 타입의 추상화를 일치시키기 위한 노력을 했다. 즉, 소프트웨어와 하드웨어 간의 호환이 가능한 데이터 타입만을 허용하여 예측 불가능한 오작동을 미연에 방지하고 서로 간의 통신이 원활하게 했다. 또한, 모든 변수 값은 초기 값을 갖도록 하였으며 한 번에 두 개 이상의 값이 할당되는 것을 막아 결정적인 동작(deterministic behavior)을 하도록 하였다. 따라서 GEZEL로 구현한 프로그램은 동일 입력에 대한 출력 값이 항상 동일한 값을 갖는다.
- C로 작성된 코드를 GEZEL코드로 변경하는 것이 용이하다는 것이다. 모듈의 설계방식에는 크게 컨트롤 플로우 모델링과 데이터패스 모델링 방식이 있다. C는 컨트롤 플로우 모델링으로 설계되고 HDL의 경우 대부분 데이터패스 모델링 방식으로 설계가 된다. 이렇게 초기에 설계 방식이 다르기 때문에 소프트웨어와 하드웨어 코드의 상호간의 변환에 어려움이 있다. 변환을 한 경우에도 개발자의 의도에서 벗어나는 코드가 생기거나 초기 계획한 모델과는 다른 형태의 모델로 변환이 될 수도 있어 신중해야만 한다. 그렇기 때문에 오작동의 여지가 있으며 변환한 결과에 대한 검증이 필요하다. 하지만 GEZEL의 경우 두 가지 형태의 모델링이 모두 가능하기 때문에 코드 변환에서 생기는 문제에서 벗어날 수 있다.
- GEZEL은 FSMD모델을 사용하기 때문에 컨트롤모델과 데이터 프로세싱 모델이 분리되어 표현된다. Fig.9. 는 이러한 모델링의 분리가 주는 이점을 보여준 예이다. GEZEL의 경우 데이터의 증감을 계산하는 updown 데이터패스와 updown을 조건에 따라 호출할 FSM을 각각 구현하였기 때문에 코드가 간결하고 이해가 쉬운 반면, System C의 경우 두 모델의 분리가 완벽하지 않아 상대적으로 코드가 복잡하고 이해하기가 어렵다.
- 문법이 C와 매우 흡사하면서 직관적이기 때문에 기존의 SystemC, Verilog HDL, VHDL등의 언어로 구현된 결과보다 코드길이가 짧으며 별도의 툴이 없이도 VHDL로의 자동변환이 가능하기 때문에 기존 언어들과의 호환성이 좋다. 따라서 다른 언어로 작성된 코드와의 객관적인 성능 비교 평가가 가능하다.

Table 1. 은 소프트웨어 설계 엔지니어가 많이 사용하는 하드웨어 설계 언어중 하나인 System C와 GEZEL을 비교한 것이다[13].

Table 1. Compare GEZEL to System C

	GEZEL	System C
계산 모델	cycle true	event driven
모델링 단위	FSMD	HDL process
결정적 모델	○	×
언어	전용	범용
시뮬레이션	스크립트	컴파일
실행기반	○	×
어플리케이션	플랫폼 실행	시스템 모델링
코시뮬레이션 인터페이스	유저 정의 라이브러리	C++

```
입력: K01 - 1라운드 키
      P - (inA || inB || inC || inD)
출력: C - (outA || outB || outC || outD)

1.  use sbox00_r01(T2_r01[7:0], o_s00_r1);
2.  use sbox01_r01(T2_r01[15:8], o_s01_r1);
3.  use sbox02_r01(T2_r01[23:16], o_s02_r1);
4.  use sbox03_r01(T2_r01[31:24], o_s03_r1);
5.  use sbox10_r01(T4_r01[7:0], o_s10_r1);
6.  use sbox11_r01(T4_r01[15:8], o_s11_r1);
7.  use sbox12_r01(T4_r01[23:16], o_s12_r1);
8.  use sbox13_r01(T4_r01[31:24], o_s13_r1);
9.  use sbox20_r01(T6_r01[7:0], o_s20_r1);
10. use sbox21_r01(T6_r01[15:8], o_s21_r1);
11. use sbox22_r01(T6_r01[23:16], o_s22_r1);
12. use sbox23_r01(T6_r01[31:24], o_s23_r1);
13.
14. always {
15.    T0_r01 = inC ^ K01[63:32];
16.    T1_r01 = inD ^ K01[31:0];
17.    T2_r01 = T1_r01 ^ T0_r01;
18.    T3_r01 = o_s00_r01 ^ o_s01_r01 ^ o_s02_r01 ^ o_s03_r01;
19.    T4_r01 = T0_r01 + T3_r01;
20.    T5_r01 = o_s10_r01 ^ o_s11_r01 ^ o_s12_r01 ^ o_s13_r01;
21.    T6_r01 = T3_r01 + T5_r01;
22.    T7_r01 = o_s20_r01 ^ o_s21_r01 ^ o_s22_r01 ^ o_s23_r01;
23.    T8_r01 = T5_r01 + T7_r01;
24.
25.    outA = inA ^ T8_r01;
26.    outB = inB ^ T7_r01;}
```

Fig.10. F function's Pseudo Code

3.2 SEED 설계

GEZEL로 작성한 코드는 데이터패스부와 FSM부분으로 나눠진다. 이는 GEZEL만이 갖는 특성으로 데이터패스에 선언된 변수들을 조건에 따라 FSM에 한번만 명시하도록 되어있기 때문에 한 데이터패스내에서 값이 여러 번 할당되지 않는다. 따라서 변수들의

의존성이 적어 훨씬 수월하게 프로그래밍이 가능하다. 데이터패스부에는 사용되는 변수 외에도 하위 데이터 패스를 정의하는 것이 가능하다. Fig.10. 에서는 F함수에 사용되는 S박스들을 데이터패스부에 선언을 하고 각각에 사용되는 입출력 포트들을 지정하는 방법으로 F함수를 구현하였다.

Fig.10. 에서와 같이 S박스의 입력으로 사용되는 T0_r01 ~ T8_r01의 값을 만들어주기 위해서 always블록 안에서 덧셈과 XOR연산을 통해 연산식을 작성한다. 프로그램이 동작을 시작하면 always 블록 안의 모든 식들이 동시에 계산을 시작한다. 라운드의 입력으로 inC, inD가 K01와 함께 들어오기 때문에 T0_r01과 T1_r01이 가장 먼저 올바른 값으로 계산이 된다. T0_r01과 T1_r01는 T2_r01가 계산되는데 사용되고 T2_r01은 sbox00_r01 ~ sbox03_r01의 입력으로 사용된다. S박스인 sbox00_r01 ~ sbox03_r01에 입력 값이 들어가면 그에 따라 치환 값 o_s00_r01 ~ o_s03_r01이 출력되고 이를 이용하여 T3_r01가 올바른 값으로 출력이 된다. 이와 같은 방식으로 모든 변수가 계산이 되고 알고리즘이 수행되면 F함수의 최종 출력 값인 outA, outB이 나오게 된다. outC와 outD값은 출력 핀인 inC와 inD로 출력되게 된다. 이렇게 계산된 outA, outB, outC, outD를 inC, inD, inA, inB에 차례로 연결해 두면 2라운드의 입력으로 들어가 2라운드가 올바르게 수행된 결과를 출력한다. 완성된 1라운드의 데이터패스를 복사하여 그대로 나열하게 되면 loop unrolling을 기반으로 한 설계가 되며 그대로 동작하게 된다. 또한, 처리량의 증가를 위해 매 연산의 결과를 레지스터에 저장하는 방식으로 설계하여 파이프라이닝이 가능하도록 하였다.

라운드키 생성과정은 암호화 함수와 별도로 구현하고 이를 on-the-fly방식으로 키를 생성하였다. 그 결과 각 라운드에 사용된 S박스가 12개씩 16라운드이므로 192개와 키 생성에 8개씩 16라운드로 128개, 총의 320개의 S박스가 사용되었다. Fig.11. 은 SEED의 라운드키 생성을 수도 코드로 나타내었다.

GEZEL로 하드웨어 모듈을 구현 시 각 모듈들을 데이터패스 형태로 구현하였다면, FSM 방식으로 동작 방식을 결정하게 된다. Fig.12. 는 GEZEL로 구현 한 SEED의 FSM 코드를 나타내고 있다. 각 상태(state)를 정의한 후 FSM에 연결된 테스트 벤치의 입력 신호 값의 변화에 따라 상태의 변화가 일어나게 된다. 각 상태에서의 동작은 한 클럭 동안 발생한다.

Fig.13. 과 Fig.14. 는 GEZEL로 하드웨어 구현 시 일반적으로 쓰이는 테스트 벤치의 형태와 반드시 들어가야 하는 시스템 설정 모듈을 각각 나타내었고 Fig.15. 는 본 논문에서 구현한 SEED 알고리즘에 대한 전체적인 하드웨어 구조도를 나타내었다. 전체 SEED의 암호화 동작은 146cycle에 수행되었다.

```
입력: IK - (i_A || i_B || i_C || i_D)
출력: OK - (o_A || o_B || o_C || o_D)
1.    use seed_sbox00_kr01(T0[7:0],o_s00_kr01);
2.    use seed_sbox01_kr01(T0[15:8],o_s01_kr01);
3.    use seed_sbox02_kr02(T0[23:16],o_s02_kr01);
4.    use seed_sbox03_kr03(T0[31:24],o_s03_kr01);
5.    use seed_sbox10_kr01(T1[7:0],o_s10_kr01);
6.    use seed_sbox11_kr01(T1[15:8],o_s11_kr01);
7.    use seed_sbox12_kr01(T1[23:16],o_s12_kr01);
8.    use seed_sbox13_kr01(T1[31:24],o_s13_kr01);
9.
10. always {
11.       T0 = i_A + i_C + i_KC;
12.       T1 = i_B + i_KC - i_D;
13.       o_K = ((o_s00_kr01 ^ o_s01_kr01 ^ o_s02_kr01 ^ o_s03_kr01)
14.                  <<32) | (o_s10_kr01 ^ o_s11_kr01 ^ o_s12_kr01
15.             ^ o_s13_kr01));
16.       o_A = (i_A<<8) ^ (i_B>>24);
17.       o_B = (i_B<<8) ^ (i_A>>24);
18.       o_C = i_C;
19.       o_D = i_D;}
```

Fig.11. Roundkey generation

```
1. fsm ftestbench(testbench) {
2. initial s0;
3. state s1,s2;
4. @s0 (start_1) -> s1;
5. @s1 (start_0) -> s2;
6. @s2 if (done) then (start_0) -> s1;
          else (start_0) -> s2;}
```

Fig.12. SEED's FSM Module

```
1. dp_testbech(out rstaes:ns(1);out key:ns(128);out plaintext:ns(128);
2.              in ciphertext : ns(128);in done : ns(1)) {
3.   reg dtext_out : ns(128);
4.   reg ddone_out : ns(1);
5.   sfg start_1 { rstaes  = 1;}
6.   sfg start_0 { rstaes = 0;}
7.
8.   always {
9.      plaintext = 0x000102030405060708090a0b0c0d0e0f;
10.    key     = 0x00000000000000000000000000000000;
11.    dtext_out = ciphertext;
12.    ddone_out = done;}
```

Fig.13. Testbench module

```
1. dp seed_sys {
2. sig i_start, done                : ns(1);
3. sig key, plaintext, ciphertext  : ns(128);
4. use testbench(i_start, key, plaintext, ciphertext, done);
5. use seed_top(i_start, key, plaintext, ciphertext, done);}
6.
7. system S {seed_sys;}
```

Fig.14. System module

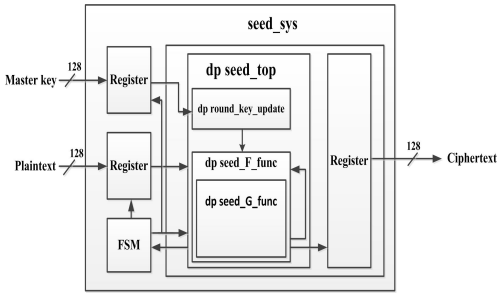


Fig.15. Block diagram of SEED

3.3 ARIA 설계

본 절에서는 GEZEL을 이용하여 ARIA를 설계하고 개발한 방법론에 대해서 설명한다. 암호 알고리즘의 설계에 있어 크게 고려되는 사항은 알고리즘의 동작속도와 모듈의 크기이다. 알고리즘의 동작속도를 높이기 위해서는 많은 양의 하드웨어 자원을 필요로 하게 되고 이 때문에 면적이 늘어나게 된다. 또한 면적을 최소화하기 위해서는 적은 양의 자원을 반복적으로 사용하여야하기 때문에 알고리즘의 동작 속도가 감소한다. 이 두 가지 방법은 상호보상적인(trade-off) 관계에 있기 때문에 설계자는 사용용도에 맞는 설계방법을 택하여 설계를 해야 한다. 본 논문에서는 ARIA의 동작 속도와 처리량을 높이는 것에 초점을 맞춰 병렬처리 방식을 이용하여 설계를 하였다. Fig.16. 은 ARIA를 구현하기에 앞서 작성한 FSM이다. Fig.16. 을 기반으로 ARIA구현에 필요한 데이터패스들을 GEZEL로 구현하도록 한다. 필요한 데이터패스는 6개로 전체 데이터패스를 동작시키는데 필요한 FSM, 암호화 알고리즘이 동작할 arai_top, 치환계층인 aria_sub, 확산계층인 aria_diff, 키요소를 만들어주는 aria_key_init, 라운드키 생성을 위한 round_key_gen이다. 데이터패스를 작성한 후에는

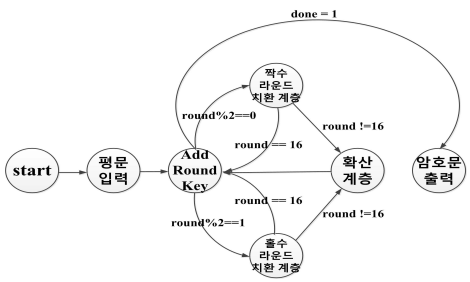


Fig.16. FSM of ARIA

```
입력: key - 마스터키 128비트
      plain - 평문 128비트
출력: cipher - 암호문 128비트
```

```
1.  use key_init(key_in, i_w0, i_w1, i_w2, i_w3, done);
2.  use sub_02(pos, ek1, link_0);
3.  use diff_02(link_0, out_temp_0);
4.  use sub_03(neg, ek2, link_1);
5.  use diff_03(link_1, out_temp_1);
6.  ……
7.  use sub_12(pos, ek11, link_10);
8.  use diff_12(link_10, out_temp_10);
9.  use sub_13(neg, ek12, out_temp11);
10.
11. always {
12.       pos=1;neg=0;
13.       ek1=plain ^ i_w0 ^ (i_w1 << 109 | i_w1 >> 19);
14.       ek2=out_temp0 ^ i_w1 ^ (i_w2 << 109 | i_w2 >> 19);
15. ……
16.       ek13=out_temp11 ^ i_w0 ^ (i_w1 << 31 | i_w1 >> 97);
17.       cipher = k1d ? 0 : ek13;}
```

Fig.17. ARIA's top module

각 데이터패스의 입출력을 정의한다. aria_key_init의 경우 입력으로는 사용자가 입력한 마스터키가 될 것이며, 출력으로는 라운드키를 생성할 키요소인 W_0, W_1, W_2, W_3가 출력될 것이다. Fig.17. 은 aria_top에 관한 수도 코드이다.

Fig.17. 에서는 하위 데이터패스로 사용할 aria_sub_α ($1 \leq \alpha \leq 13$)와 aria_diff_β ($1 \leq \alpha \leq 12$)를 데이터패스부에 선언하고 입출력을 지정했다. 치환계층인 aria_sub_α의 입력포트로는 치환 전 데이터인 ekα를, 출력 포트로는 link_α를 지정했다. 출력된 link_α는 확산계층의 입력으로 사용하기 위하여 확산계층의 입력포트에 연결하였다.

연결이 끝나면 앞서 언급한 바와 같이 데이터패스부에 선언된 변수들의 값을 결정하는 식을 always블록 안에 나열하여 변수들의 값이 올바르게 나타나도록 해야한다. 예를 들어 Fig.17. 의 ek1은 평문, i_w0와 i_w1을 오른쪽으로 19비트 순환 이동한 결과를 XOR연산한 결과가 ek1이다. 즉, ek1 = plain ^ i_w0 ^ (i_w1 〈〈109 | i_w1 〉〉 19) 와 같은 식을 always블록에 적어 ek1이 생성될 수 있도록 한다. 위와 같은 방식으로 always블록 안에 모든 변수에 대한 식을 작성하면 데이터패스의 구현이 마무리된다. Fig.18. 과 Fig.20. 은 Fig.17. 에서 사용된 확산계층과 치환계층에 대한 수도코드이다.

한편, 치환계층에도 하위 데이터패스인 S박스가 사용된다. GEZEL에는 lookup이라는 타입이 별도로 존재한다. 이를 이용하면 S박스의 구현이 매우 간편하다. Fig.19. 는 ARIA에 사용되는 S박스의 수도코드이다. Fig.19. 와 같이 입력데이터의 크기를 명

시하고 데이터를 나열한다. 그러면 입력 데이터인 din이 들어올 때마다 S변수의 din번째 변수를 출력해주기 때문에 S박스가 손쉽게 구현된다.

```
입력: sel - 홀수 라운드 : 1, 짝수 라운드 : 0
      din - 치환 전 128비트
출력: dout - 치환 후 128비트
```

```
1.   //LT1
2.  use aria_sbox00(s00in,s00out);
3.  use aria_sbox01(s01in,s01out);
4.  use aria_sbox02(s02in,s02out);
5.  use aria_sbox03(s03in,s03out);
6.  //LT2
7.  use aria_sbox10(s10in,s10out);
8.  use aria_sbox11(s11in,s11out);
9.  use aria_sbox12(s12in,s12out);
10. use aria_sbox13(s13in,s13out);
11. //LT3
12. use aria_sbox20(s20in,s20out);
13. use aria_sbox21(s21in,s21out);
14. use aria_sbox22(s22in,s22out);
15. use aria_sbox23(s23in,s23out);
16. //LT4
17. use aria_sbox30(s30in,s30out);
18. use aria_sbox31(s31in,s31out);
19. use aria_sbox32(s32in,s32out);
20. use aria_sbox33(s33in,s33out);
21.
22. always {
24. s00in = (sel) ? Din[31:24] : din[15:8];
25. s01in = (sel) ? Din[23:16] : din[7:0];
26. ……
27. s32in = (sel) ? Din[111:104] : din[127:120];
28. s33in = (sel) ? Din[103:96] : din[119:112];
29.
30. dout = (sel) ? ((s03out<<0) | (s02out<<8) |
31.          (s01out<<16) | (s00out<<24) |
32.          (s13out<<32) | (s12out<<40) |
33.          (s11out<<48) | (s10out<<56) |
34.          (s23out<<64) | (s22out<<72) |
35.          (s21out<<80) | (s20out<<88) |
36.          (s33out<<96) | (s32out<<104) |
37.          (s31out<<112) | (s30out<<120)) :
38.            ((s01out<<0) | (s00out<<8) |
39.          (s03out<<16) | (s02out<<24) |
40.          (s11out<<32) | (s10out<<40) |
41.          (s13out<<48) | (s12out<<56) |
42.          (s21out<<64) | (s20out<<72) |
43.          (s23out<<80) | (s22out<<88) |
44.          (s31out<<96) | (s30out<<104) |
45.          (s33out<<112) | (s32out<<120));}
```

Fig.18. ARIA's substitution layer

```
입력: din - 입력 데이터 8비트
출력: dout - 치환된 데이터 8비트
```

```
1. lookup S: ns(8) = {
2.           0x63, 0x7c, 0x77, …, 0x16}
3.
4. always {dout = S(din);}
```

Fig.19. pseudo code of ARIA's sbox

ARIA의 경우 라운드키 생성을 위해서는 4개의 키 요소가 필요하다. 입력받은 마스터키와 사전에 정의된 상수를 암호화 데이터패스의 입력으로 넣어 3라운드를 진행한다. 이때 각 라운드의 출력 값을 wγ_out

```
입력: din - 입력 데이터 128비트
출력: dout - 출력 데이터 128비트
```

```
1. always {
2.  T1 = din[31:24]^din[39:32]^din[79:72]^din[119:112];
3.  T2 = din[23:16]^din[47:40]^din[23:16]^din[127:120];
4.  T3 = din[15:8]^din[55:48]^din[95:88]^din[103:96];
5.  T4 = din[7:0]^din[63:56]^din[87:80]^din[111:104];
6.
7.  r01 = din[55:48]^din[71:64]^din[111:104]^T1;
8.  r06 = din[15:8]^din[87:80]^din[127:120]^T1;
9.  r12 = din[23:16]^din[63:56]^din[103:96]^T1;
10. r15 = din[7:0]^din[47:40]^din[95:88]^T1;
11. r02 = din[63:56]^din[79:72]^din[103:96]^T2;
12. r05 = din[7:0]^din[95:88]^din[119:112]^T2;
13. r11 = din[31:24]^din[55:48]^din[111:104]^T2;
14. r16 = din[15:8]^din[39:32]^din[87:80]^T2;
15. r03 = din[39:32]^din[87:80]^din[127:120]^T3;
16. r08 = din[31:24]^din[64:71]^din[104:111]^T3;
17. r10 = din[7:0]^din[47:40]^din[119:112]^T3;
18. r13 = din[23:16]^din[63:56]^din[79:72]^T3;
19. r04 = din[47:40]^din[95:88]^din[119:112]^T4;
20. r07 = din[23:16]^din[79:72]^din[103:96]^T4;
21. r09 = din[15:8]^din[39:32]^din[127:120]^T4;
22. r14 = din[31:24]^din[55:48]^din[71:64]^T4;
23.
24. dout = r04 | (r03<<8) | (r02<<16) | (r01<<24) |
25.            (r08<<32) | (r07<<40) | (r06<<48) |
26.            (r05<<56) | (r12<<64) | (r11<<72) |
27.            (r10<<80) | (r09<<88) | (r16<<96) |
28.            (r15<<104) | (r14<<112) | (r13<<120);}
```

Fig.20. ARIA's diffusion layer

($0 \leqq \gamma \leqq 3$)에 저장하고 이를 라운드키를 생성할 때 사용한다. 키요소 4개가 모두 생성이 되면 입력 받은 평문의 암호화를 수행한다. 암호화 수행과 더불어 라운드키 생성이 동시에 진행이 되고 on-the-fly방식으로 라운드키가 생성되어 전체 암호화 시간을 단축할 수 있도록 하였다. Fig.21. 은 키요소 생성을 위한 키 초기화에 대한 수도 코드이다.

전체 ARIA는 확산계층, 치환계층과 라운드키 덧셈을 두 개의 데이터패스로 만든 뒤, 이를 나열하여 loop unrolling방식으로 알고리즘이 동작하도록 하

```
입력: key_in - 마스터키 128비트
출력: w0_dout, w1_dout, w2_dout, w3_dout  - 키생성 요소 128비트
      dout - 암호문 출력 신호
```

```
1.  always {
2.    CK1 = 0x517cc1b727220a94fe13abe8fa9a6ee0
3.          ^ key_in;
4.    CK2 = 0x6db14acc9e21c820ff28b1d5ef5de2b0
5.          ^ w1_temp;
6.    CK3 = 0xdb92371d2126e9700324977504e8c90e
7.          ^ w2_temp ^ key_in;
8.    pos = 1; neg = 0;
9.
10.   w0_temp = key_in;
11.   w0_dout = w0_temp;
12.   w1_dout = w1_temp;
13.   w2_dout = w2_temp ^ key_in;
14.   w3_dout = w3_temp ^ w1_temp;
15.
16.   cnt = cnt + 1;
17.   done = (cnt >= 16) ? 1:0;}
```

Fig.21. ARIA's key initiailize

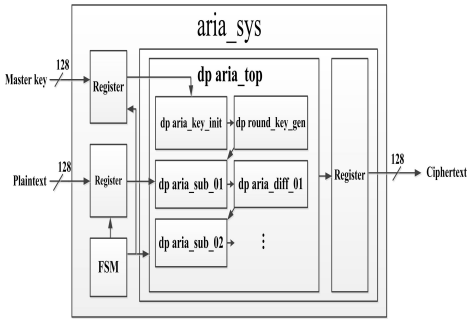


Fig.22. ARIA's Block diagram

였다. 라운드키 덧셈의 경우 클럭을 소요하지 않으므로 연속된 연산인 치환계층과 함께 구현하여 한 클럭에 동작할 수 있도록 하였다. 또한, 데이터 처리량을 늘리기 위하여 각 연산의 출력 변수를 레지스터에 저장하여 연산별로 파이프 라이닝이 되도록 구현하였다. 그 결과 총 240개의 8비트 S박스를 사용하여 구현하였으며 동작 속도는 첫 암호문의 출력까지 66cycle이 걸렸으며, 이 후 매 클럭 암호문이 출력됐다. 다음 Fig.22. 는 본 논문에서 구현한 ARIA 알고리즘에 대한 전체적인 하드웨어 구성도를 나타내었다.

IV. 8051프로세서와의 통합설계

본 장에서는 8051 마이크로 컨트롤러를 사용하여 앞장에서 소개한 ARIA와 SEED를 하드웨어와 소프트웨어의 통합설계 방식으로 구현한 결과를 보인다. GEZEL은 ARM, 8051 마이크로 컨트롤러와의 코디자인을 지원하기 때문에 아래와 같이 ipblock 코드를 삽입하면 별도의 컴파일 없이 8051과의 통신이 이루어지면서 결과 값을 출력한다. 앞서 GEZEL로 작성한 암호 알고리즘의 코어와 8051 프로세서에는 C 언어로 ECB, CBC, CFB, OFB, CTR등의 운영모드를 구현하였다. 운영모드는 소프트웨어로 구현하였기 때문에 추가적인 면적이 소요되지 않으며, 수정이 용이하다. Fig.24. 은 8051프로세서와의 통합설계를 나타낸 것이다. Table 2. 와 Table 3. 은 기존의 ARIA와 SEED의 연구결과와 제안하는 방법으로 구현된 코드를 VHDL로 자동 변환하여 비교한 것으로, ARIA의 경우 [3]에 비해 면적이 다소 증가하였지만, 최대 동작 주파수는 49% 증가하였다. 이는 VHDL로의 변환이 최적화되어 변환되지 않는 점을 고려할 때 기존의 결과들과 유사한 결과를 보인다. 또한 SEED는 69043 slice의 면적과 146.25Mhz을 보였으며,

```
1.  ipblock my8051 {
2.    iptype "i8051system";
3.    ipparm "exec=driver.ihx";
4.    ipparm "verbose=1";
5.  }
6.  ipblock my8051_ins(out data : ns(8)) {
7.    iptype "i8051systemsource";
8.    ipparm "core=my8051";
9.    ipparm "port=P0";
10. }
11. ipblock my8051_datain(out data : ns(8)) {
12.   iptype "i8051systemsource";
13.   ipparm "core=my8051";
14.   ipparm "port=P1";
15. }
16. dp sys {
17.   sig ins, din : ns(8);
18.   use my8051;
19.   use my8051_ins(ins);
20.   use my8051_datain(din);
21.   use hello_decoder(ins, din);
22. }
```

Fig.23. ipblock for communication with 8051 processor

시그널플로우 방식으로 구현할 경우 83cycle까지 조정이 가능하여 구현상의 유연성을 가짐을 확인하였다.

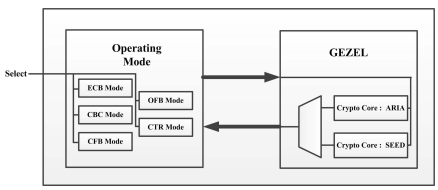


Fig.24. Codesign model using GEZEL and 8051 processor

Table 2. Comparision with previous ARIA research

논문명	면적	최대 동작 주파수 (MHz)
[1]	1,491(slice)	46.5
[2]	11,301(gates)	467
[3]	6,437(slice)	192.9
[4]	13,893(gates)	71
[5]	6,076(gates)	15
[6]	15,496(slice)	
제안하는 ARIA	7,282(slice)	286.172

Table 3. Comparision with previous SEED research

논문명	면적	최대 동작 주파수(Mhz)
[17]		5
[20]	36,678 (slice)	46.2
[21]	65,000(gates)	50.569
[22]	50,000(gates)	10
제안하는 SEED	69,043(slice)	146.25

V. 결 론

본 논문에서는 GEZEL을 이용한 국내 표준 블록 암호 알고리즘인 ARIA 및 SEED 알고리즘을 구현함으로써 기존의 설계 방법과는 다른 개발 방법론을 제안한다. GEZEL을 이용한 개발 방법이 갖는 장점은 문법이 간단하여 기존의 다른 구현 방법에 비해 개발 시간이 단축되고, 코드의 길이도 많이 줄어들게 된다. 또한, 소프트웨어 엔지니어들도 쉽게 하드웨어 구현을 할 수 있도록 문법 자체가 C언어와 비슷하고 유사한 특성을 지닌 SystemC와 비교해서도 구현이 간단하다. 그리고 GEZEL이 VHDL로 자동 변환이 가능하고, 8051, ARM등 여러 CPU와도 호환성이 좋아 소프트웨어와 하드웨어 통합설계가 간편하기 때문에 다양한 플랫폼에서 적용 가능한 효율적인 언어라고 하겠다. 본 논문은 국내에서 GEZEL을 활용하여 하드웨어를 설계한 첫 논문으로써 국내 표준 블록 암호 알고리즘인 ARIA와 SEED을 실제 구현해서 그 효율성을 검증하였다. 구현 결과 ARIA의 경우 7,282slice의 면적을 차지했으며 286.172Mhz로 동작하였다. 이는 기존의 결과와 면적 대비 동작속도를 비교 했을 시 매우 유사한 성능을 나타냈다. 또한 SEED는 69043slice와 146.25Mhz의 동작속도를 가지며 시그널플로우 방식으로 구현시 최대 동작 주파수가 약 296%까지 향상되었다. 외국에서는 GEZEL의 우수한 특성으로 인해서 많은 결과물들이 나오고 있음에도 불구하고 국내에서는 관련 연구가 전무하기 때문에 향후 국내에서도 GEZEL을 잘 활용한다면 ARIA와 SEED외에도 국산 서명 알고리즘인 KCDSA, EC-KCDSA등에서도 좋은 연구 결과들을 빠른 시간에 효율적으로 생산해 낼 수 있을 것으로 보이며 향후 연구로 이를 진행해 나갈 예정이다.

References

[1] Jinsub Park, Yeonsang Yun, Young-Dae Kim, Sangwoon Yang, Taejoo Chang, and Younggap You, "Design and Implementation of ARIA Cryptic Algorithm," the Institute of Electronics Engineers of Korea, 42(4), pp. 29-36, Apr. 2005

[2] Gwonho Ryu, Bonseok Koo, Sangwoon Yang, Taejoo Chang, "Area Efficient Implementation of 32-bit Architecture of ARIA Block Cipher Using Light Weight Diffusion Layer," Jonornal of The Korea Institute of information Security & Cryptology, 16(6), pp. 15-24, Dec. 2006

[3] Seong-Ju Ha and Chong-Ho Lee, "Design of High Speed Encryption/ Decryption Hardware for Block Cipher ARIA," the Institute of Electronics Engineers of Korea, IT-22(6), pp. 1652-1695, Sep. 2008

[4] Jinsub Park, Young-Dae Kim, Sangwoon Yang, and Younggap You, "Low power compact design of ARIA block cipher," Proceedings of the 2006 IEEE International Symposium on Circuits and Systems, pp. 313-316, May. 2006

[5] Sangwoon Yang, Jinsub Park, and Younggap You, "The smallest ARIA module with Proc. ICISC-LNCS 4296, pp. 107-117, 2006

[6] Bonseok Koo, Gwonho Ryu, Taejoo Chang, and Sangjin Lee, "Design and Implementation of Unified Hardware for 128-Bit Block Ciphers ARIA and AES," ETRI Journal, 29(6), pp. 820-822, Dec. 2007

[7] Patrick R. Schaumont, "A Practical Introduction to Hardware/Software Codesign," Springer, Dec. 2012.

[8] A. Hodjat, L. Batina, D. Hwang, and I. Verbauwhede, "A hyperelliptic curve crypto coprocessor for an 8051 microcontroller," Proc. SIPS'05, pp. 93 - 98, 2005.

[9] L. Batina, D. Hwang, A. Hodjat, B. Preneel, and I. Verbauwhede, "Hardware/Software Co-design for Hyperelliptic Curve Cryptography (HECC) on the 8051 μP," Proc. CHES-LNCS, no. 3659, pp. 106 - 118, Springer-Verlag, 2005.

[10] Kazuo Sakiyama, Lejla Batina, Bart Preneel, and I. Verbauwhede, "HW/SW Co-design for Accelerating Public-Key Cryptosystems over GF(p) on the 8051 μ-controller," WAC 2006, pp. 1 - 6, 2006

[11] Xu Guo, Zhimin Chen, and Patrick Schaumont, "Energy and Performance Evaluation of an FPGA-Based SoC Platform with AES and PRESENT Coprocessors", 8th international Workshop AMS, July. 2008.

[12] Patrick Schaumont and I. Verbauwhede, "Interactive Cosimulation with Partial Evaluation," Proc. DATE 2004, vol. 1, pp. 642-647, 2004.

[13] Patrick Schaumont, Doris Ching, and I. Verbauwhede, "An Interactive Codesign Environment for Domain-Specific Coprocessors," ACM Transactions on Design Automation of Electronic Systems Vol. 11, no. 1, pp. 70 - 87, 2006.

[14] Yusuke Matsuoka, Patrick Schaumont, Kris Tiri, and I. Verbauwhede, "Java Cryptography on KVM and its Performance and Security Optimization using HW/SW Co-design Techniques," CASES'04, pp. 22 - 25, Sept, 2004.

[15] Y.K. Lee, H. Chan, and I. Verbauwhede, "Throughput Optimized SHA-1 Architecture Using Unfolding Transformation," ASAP 2006, pp. 354-359, 2006.

[16] M. Knezevic, K. Sakiyama, Y.K. Lee, and I. Verbauwhede, "On the High-Throughput Implementation of RIPEMD-160 Hash Algorithm," Proc. ASAP 2008, pp. 6, 2008.

[17] Y.H. Seo, J.H. Kim, and D.W. Kim, "Hardware Implementation of 128-bit Symmetric Cipher SEED," APASIC 2000, Proc. the Second IEEE Asia Pacific Conference on ASIC, pp. 183-186, Aug, 2000.

[18] Choi Byeoung-Yoon and Suh Chung Wook, "Design of Cryptographic Coprocessor for SEED Algorithm," The Journal Of Korea Information and Communications Society, 25(9B), pp. 1609-1616, Sep. 2000

[19] Shin-woo Jeon and Young-Jin Jeong, "High Performance Hardware Implementation of the 128-bit SEED Cryptography Algorithm," Jonornal of The Korea Institute of information Security & Cryptology, 11(1), pp. 13-23, Feb. 2001

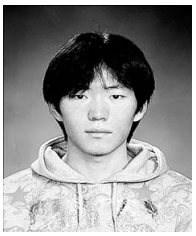
[20] Ye Chul Park and Kang Yi, "An Integrated Design and Implementation of 128 - block cipher SEED and UART with a low - cost FPGA," The Korean Institute of Information Scientists and Engineers, 30(2), pp. 205-207, Oct. 2003

[21] Kang Yi and Ye Chul Park, "Design and Implementation of a 128 - bit Block Cypher Algorithm SEED Using a Low - Cost FPGA for Embedded Systems," The Korean Institute of Information Scientists and Engineers, 31(7), pp. 402-413, Aug. 2004

[22] Shin Kwang Cheul and Lee Haeng Woo, "An Optimum Architecture for Implementing SEED Cipher Algorithm with Efficiency," Korean Society for Internet Information, 7(1), pp. 49-57, Feb. 2006

[23] R. Anderson and R. Needham, "Robustness principles for public key protocols," Advances in Cryptology, CRYPTO'95, LNCS 963, pp. 236-247, 1995.

〈저자소개〉



권 태 웅 (TaeWoong Kwon) 학생회원
2012년 2월: 숭실대학교 컴퓨터학부 학사 졸업
2012년 2월~현재: 고려대학교 정보보호대학원 정보보호학과 석사과정
〈관심분야〉 암호알고리즘 하드웨어 설계, 고속화 구현, 공개키 암호 알고리즘

김 현 민 (HyunMin Kim) 학생회원
2006년 2월: 동국대학교 전자공학과 학사 졸업
2005년 12월~2008년 12월: 삼성전자 반도체 총괄 연구원
2009년 3월~2011년 8월: 고려대학교 정보보호대학원 석사
2010년 9월~2011년 8월: COSIC, KULeuven, Belgium, International Scholer
2011년 9월~2012년 8월: COSIC, KULeuven, Belgium, Pre-doctoral Scholar
2013년 3월~현재: COSIC, KULeuven, Belgium, Ph.D candidate
2013년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 부채널 공격, 초경량 암호모듈 설계, Secure Logic Design, 보안칩 설계, 암호시스템 안전성 분석 및 고속화 구현

홍 석 희 (Seokhie Hong) 종신회원
1995년 2월: 고려대학교 수학과 학사 졸업
1997년 2월: 고려대학교 수학과 석사 졸업
2001년 8월: 고려대학교 수학과 박사 졸업
1999년 8월~2004년 2월: (주)시큐리티 테크놀로지스 선임연구원
2003년 8월~2004년 2월: 고려대학교 정보보호기술연구센터 선임연구원
2004년 4월~2005년 2월: K.U.Leuven. ESAT/SCD-COSIC 박사후연구원
2005년 3월~2013년 8월: 고려대학교 정보보호대학원 부교수
2013년 9월~현재: 고려대학교 정보보호대학원 정교수
〈관심분야〉 대칭키·공개키 암호 분석 및 설계, 컴퓨터 포렌식