Rijndael S-box의 세 가지 구현 방법에 따른 FPGA 설계

이윤경, 박영수, 전성익
한국 전자통신 연구원
전화 : 042-860-1389

FPGA Implementation of Rijndael Algorithm according to the Three S-box Implementation Methods

Yun-Kyung Lee, Yeong-Su Park, Sung-Ik Jun
IC Card Research Team/ ETRI
E-mail : neohappy@etri.re.kr

Abstract

Rijndael algorithm is known to a new private key block cipher which is substitute for DES. Rijndael algorithm is adequate to both hardware and software implementation, so hardware implementation of Rijndael algorithm is applied to high speed data encryption and decryption.

This paper describes three implementation methods of Rijndael S-box, which is important factor in performance of Rijndael coprocessor. It shows synthesis results of each S-box implementation in Xilinx FPGA. The three S-box implementation methods are implementation using lookup table only, implementation using both lookup table and combinational logic, and implementation using combinational logic only.

Ⅰ. 서 론

2001년 10월 DES의 후속인 AES 알고리듬으로 선택된 Rijndael 알고리듬은 안정성 및 하드웨어, 소프트웨어 구현성이 뛰어난 알고리듬으로 알려져 있다[1]. 블록 암호 알고리듬을 하드웨어로 구현함으로써 인증 및 보안을 요하는 데이터의 암/복호화를 빠른 시간에 처리할 수 있으며, 고속, 저전력, 소형화 칩으로 만들어 smart card에 응용할 수도 있다. 그리고 USIM(Universal Subscriber Identity Module)에 탑재하여 mobile 환경에서의 보안에도 응용할 수 있다.

본 논문에서는 Rijndael 알고리듬의 하드웨어 구현시 주요 관심사가 되는 S-box의 세 가지 설계 방법을 제시하고, Xilinx FPGA를 이용하여 구현한 결과를 기술하고자 한다. Rijndael 알고리듬에서 256 bits 크기의 S-box는 암호화 모듈 설계에 16개, 복호화 모듈 설계에 16개, 키생성 모듈 설계에 4개가 필요하다. 필요한 S-box의 개수는 설계 방법에 따라 차이가 있으나 128 bits 단위로 암/복호화를 처리한다고 가정할 때 앞서 기술한 것처럼 36개가 필요하다. 따라서 S-box의 설계 방법에 따라 Rijndael 암호 프로세서의 성능에 많은 차이를 가져올 것으로 보인다.

Ⅱ. Rijndael 알고리듬

Rijndael 알고리듬은 128 bits 블록 데이터를 각각 128 bits, 192 bits, 256 bits key를 사용하여 암/복호화를 수행하는 블록암호 이다. 사용하는 키의 크기에 따라 10 라운드, 12 라운드, 14 라운드의 오퍼레이션을 수행하여 암/복호화된 데이터를 생성한다. 암호화의 각 라운드는 S-box를 이용한 non-linear byte mapping 과정인 Substitution 과정, 행 단위의 쉬프트 연산(Rijndael 알고리듬에서는 4개의 행으로 이루어진 matrix를 기본 처리

단위로 하며 주로 열 단위의 오퍼레이션을 취한다.)을 하는 Shift Rows 과정, 특정 상수를 열 단위로 행렬 곱셈을 취하는 과정인 MixColumn 과정, 라운드 키와 MixColumn 결과 데이터들과 XORing 하는 과정인 Add Round Key 과정으로 이루어져 있다. 복호화 연산에서는 암호화의 역변환이 이루어지는데 Shift Row와 반대 방향으로 행단위 쉬프트 연산을 취하는 Inverse Shift Rows, Substitution 연산에서 사용한 S-box의 역변환인 SI-box를 이용한 non-linear byte mapping 과정인 Inverse Substitution, Add Round Key, MixColumn에서 각 칼럼에 곱하는 상수의 $GF(2^8)$에서의 역원을 각 칼럼에 곱하는 Inverse MixColumn의 순서로 복호화가 진행된다. 그림1을 참고하자.

그림 1에 나타나 있듯이 Rijndael 알고리듬의 첫 라운드는 Add Round Key 오퍼레이션만으로 이루어져 있고, 그 다음 라운드부터 4가지 오퍼레이션을 순서대로 수행한다. 또한 암/복호화 모듈의 재사용성을 높이기 위하여 복호화 과정에서 Inverse Substitution과 Inverse Shift Rows 의 순서를 바꾸어 라운드 오퍼레이션을 하도록 구현하였다. 두 오퍼레이션 모두 바이트 단위로 오퍼레이션을 수행하므로 이 둘의 순서를 바꾸어 라운드 오퍼레이션을 수행하여도 복호화된 데이터에는 영향을 주지 않는다.

Ⅲ. S-box의 구현

Rijndael 알고리듬에서 라운드 연산은 128 bits 단위가 기본이다. 그러나 레지스터를 사용하여 64 bits 또는 32 bits 단위로 데이터 흐름을 조절 할 수도 있다. 본 논문에서 제안하는 방법에서는 추가적인 레지스터의 사용을 배제하고 빠른 속도를 위하여 128 bits 단위로 라운드 연산을 수행한다.

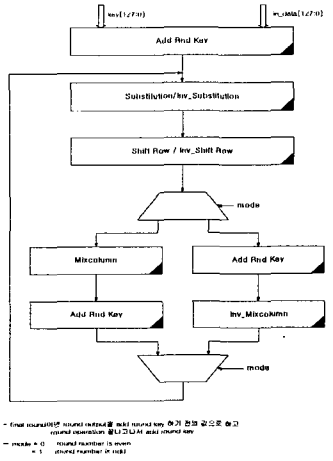


그림 1. Rijndael 알고리듬의 순서도

각 라운드 연산을 128 bits 단위로 처리한다면 Substitution 연산에 16개의 S-box가 필요하고, Inverse Substitution 연산에 16개의 SI-box가 필요하다. 또한 키 생성에 4개의 S-box가 필요하므로 총 36개의 S-box (or SI-box)가 필요하다. S-box(SI-box)는 256개의 바이트 엘리먼트로 이루어져 있는데, 1 바이트의 입력에 대해서 1 바이트의 비 선형 매핑 결과를 내보낸다.

암/복호화에 필요한 S-box, 즉 S-box와 SI-box는 동시에 사용되지 않지만 내용이 다르므로 각각 구현되어야 하고 이들은 Rijndael 암호 프로세서의 총 면적에서 많은 부분을 차지한다. 따라서 S-box/SI-box의 면적 및 속도 면에서의 효율적 구현은 Rijndael 암호 프로세서 구현에 있어 중요한 이슈가 된다.

S-box는 0부터 255까지의 수에 대해서 $GF(2^8)$에서의 곱셈에 대한 역원을 구한 후(단, '0'의 $GF(2^8)$에서의 곱셈에 대한 역원은 '0'라 둔다.) 이를 다시 Affine transform을 하여 총 256 개의 엘리먼트를 얻고, SI-box는 0부터 255 까지의 수에 대해서 Inverse Affine transform을 한 후 이 결과값에 대해서 $GF(2^8)$에서의 곱셈에 대한 역원을 구하여 얻는다. Affine transform은 1 바이트의 16진 입력값을 바이너리로 표현하여 이 값에 8*8 행렬을 곱하고 결과 값에 16진 값 "63"을 더해서 얻는다[1]. 아래 그림 2를 참고하자. 그림 2에서 bi (0 < i < 7)는 1바이트의 입력값을 바이너리로 나타낸 것으로 MSB는 b_7이고 LSB는b_0이다.

$$\begin{pmatrix} b0' \\ b1' \\ b2' \\ b3' \\ b4' \\ b5' \\ b6' \\ b7' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b0 \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

그림 2. Affine Transform

Inverse Affine Transform은 Affine Transform의 역변환을 뜻하는데, 그림 2에서 예측할 수 있듯이 입력값에 16진값 "63"을 더한 후 Affine Transform에서 사용한 행렬의 역행렬을 곱해서 얻는다. 그림 3은 Inverse Affine Tranform을 잘 보여준다.

$$\begin{pmatrix} b0 \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \left(\begin{pmatrix} b0' \\ b1' \\ b2' \\ b3' \\ b4' \\ b5' \\ b6' \\ b7' \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \right)$$

그림 3. Inverse Affine Transform

S-box(SI-box)를 구현하는 방법에는 ROM을 사용하여 Lookup Table 형태로 구현하는 방법, 곱셈에 대한 역원은 lookup table 형태로 구현하고, Affine transform과 Inverse Affine transform을 combinational logic으로 구현하여 S-box와 SI-box가 곱셈에 대한 역원의 lookup table을 공유하는 방법이 있다. 마지막으로 곱셈에 대한 역원과 Affine transform 모두 combinational logic으로 구현하는 방법이 있다. 그림4를 참고하자.

왼쪽 그림은 S-box와 SI-box(S-box의 역변환을 SI-box로 표현)를 각각 구현하여 암호화 모드에는 S-box를, 복호화 모드에는 SI-box를 이용하여 Substitution/Inverse Substitution 오퍼레이션을 하는 과정을 나타낸 것이고, 오른쪽 그림은 MI-box($GF(2^8)$에서의 곱셈에 대한 역원을 lookup table로 구현)와 Affine transform, Inverse Affine Transform을 이용한 Substitution/Inverse Substitution 과정을 나타낸 것이다.

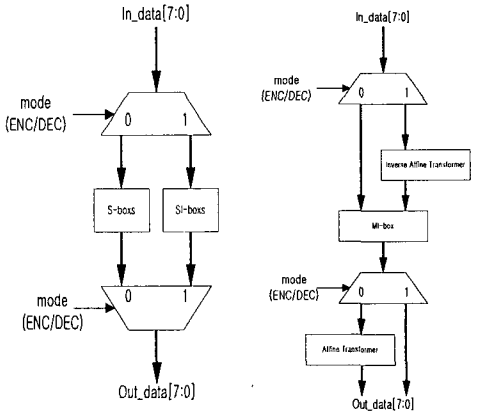


그림 4. S-box의 다양한 구현 방법

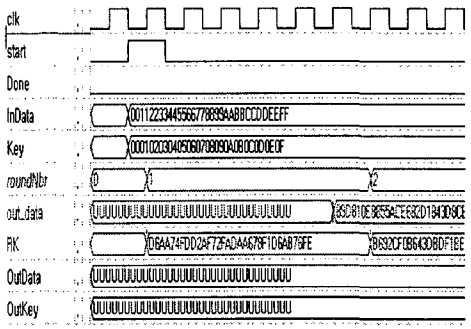
또한 Affine transform과 Inverse Affine transform은 행렬 곱셈부분을 combinational logic으로 구현한다. 이렇게 구현할 경우 MI-box 부분을 암호화와 복호화 모두에 적용할 수 있으므로 S-box와 SI-box를 각각 구현하는 것보다 약 40% 정도의 게이트수를 줄일 수 있다. 이는 암호화와 복호화는 동시에 발생하지 않는다는 것에 착안한 것이다.

MI-box를 combination logic으로 구현 하려고 할 때, Finite field에서 곱셈에 대한 역원을 구하는 잘 알려진 방법으로 extended Euclidean algorithm이 있는데 이 알고리듬은 하드웨어 구현에는 적합하지 않은 것으로 알려져 있다. 하드웨어 구현이 어렵고 구조가 복잡하며 하드웨어 구현의 장점인 빠른 속도를 얻는데도 문제가 있다. 따라서 MI-box를 combinational logic으로 구현할 때 $GF(2^8)$에서의 값을 two-term polynomial로 매핑하고, two-term polynomial의 곱셈에 대한 역원을 구한 후 결과값을 다시 $GF(2^8)$으로 매핑하는 방법을 이용하여 구현한다. Two-term polynomial의 곱셈에 대한 역원은 $GF(2^4)$에서의 곱셈과 몇 번의 XORing으로 구할 수 있으며, $GF(2^4)$에서의 곱셈 역시 XOR 게이트와 AND 게이트 만으로 구현 가능하다[2]. 따라서 combinational logic 만으로 S-box(SI-box)를 구현할 경우 세 가지 방법 중 가장 적은 게이트수로 구현이 가능하다. 그러나 combinational logic을 사용하면 lookup table을 사용하는 것보다 더 많은 게이트 딜레이를 갖게 되므로 속도는 많이 떨어지는 것으로 나타났다.

Ⅳ. 구현 결과

3절에서 설명한 세 가지 방법을 사용하여 S-box를 구현한 후 Xilinx FPGA를 사용하여 CLB 개수 및 암/복호화에 걸리는 시간을 측정 하였다. 그 결과 예상대로 lookup table만을 사용한 구조가 가장 많은 CLB로 구성되었고, combinational logic 만을 이용한 구조가 CLB 개수가 가장 적었다. 그러나 combinational logic을 이용할 경우 속도가 현저히 저하됨을 알 수 있었으며 따라서 Rijndael 암호 프로세서를 적용 하고자하는 어플리케이션에 따라 적절한 방법을 사용하여야 할 것으로 보인다. 예를들어 면적은 상관없이 빠른 속도를 요구하는 시스템에 적용된다면 lookup table 만을 이용하여 Rijndael 알고리듬을 구현하고, 무엇보다 면적이 작아야 한다면 combinational logic만을 사용하여 Rijndael 알고리듬을 구현하여야 한다.

그림 5는 구현한 Rijndael 알고리듬의 시뮬레이션 결과의 일부이다. 입력 키는 16진 값으로



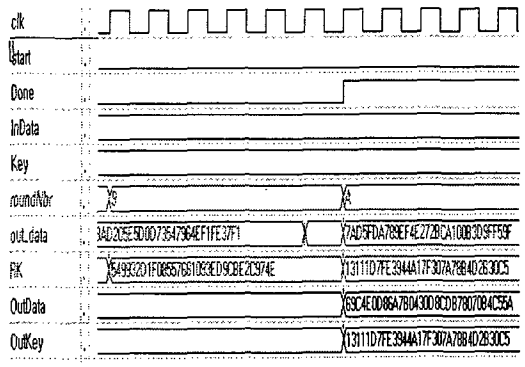


그림 5. 시뮬레이션 결과의 일부

"000102030405060708090a0b0c0d0e0f" 이고, 입력 이터는 16진 값으로 "00112233445566778899aa bbccddeeff"일 때 암호화된 데이터 "69c4e0d86 a7b0430d8cdb78070b4c55a" 의 결과가 나왔음을 보여준다.

Ⅳ. 결론

본 논문에서 제시한 세 가지 S-box 구현 방법을 사용하여 Rijndael 알고리듬을 VHDL로 구현한 후 Xilinx FPGA에서 테스트한 결과 lookup table 형태로 구현할 경우 Rijndael 암호 프로세서의 크기가 가장 컸고, combinational logic 형태로 구현하는 것이 게이트 수가 현저히 감소 하였다. 그러나 속도면을 고려하면 lookup table 형태로 구현한 것이 속도가 많이 빠름을 알 수 있었다. 따라서 Rijndael 암호 프로세서를 설계할 때 속도와 면적을 고려하여 적절히 배합하여 설계할 필요가 있으며, Rijndael 암호 프로세서를 적용하는 타겟에 따라 적절한 구조를 선택할 필요가 있다.

참고문헌

[1] Joan Daemen, Vincent Rijmen, "AES Proposal Rijndael"

[2] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger, "An ASIC Implementation of the AES Sboxes," CT-RSA 2002.