Loop Unrolling 방법을 적용한 해쉬 함수의 구현

정보선*, 정재훈*, 박동선**, 송광석***

*(주)퓨처시스템, 정보통신연구소,

**전북대학교, 정보통신공학과,

***한국전자통신연구원

Hardware Implementation of the Loop Unrolled Hash Function

Bo-Sun Jeong*, Jai-Hoon Chung*, Dong-Sun Park**, Kwang-Suk Song***

*Information & Communications Research Center, Future Systems, Inc.

**Department of Information & Communication Engineering, Chonbuk Univ.

***Electronics and Telecommunications Research Institute

요 약

본 논문에서는 loop unrolling 방법을 적용한 해쉬 함수의 하드웨어 구현에 관하여 기술한다. 해쉬 함수는 메시지의 무결성을 보장하기 위한 인증에 사용되는 알고리즘으로 메시지를 처리하는 전처리부, 데이터 압축을 수행하는 반복 프로세싱부, 그리고 처리된 결과를 출력하는 결과 출력부로 기능을 분리할 수 있다. 이때 데이터 연산 처리 속도를 개선하기 위하여 반복 프로세싱부에 loop unrolling 기법을 적용하였다. 본 논문에서는 loop unrolling 기법을 적용한 해쉬 함수의 구현에 관한 것과 이로 인한 성능 개선 효과에 대하여 기술한다.

I. 서론

인터넷 및 전산망을 이용한 정보통신에 대한 의존도가 심화되는 정보화 사회에서는 디지털 서명과 인증 기술은 그 핵심적인 기술이다. 특히 전자 상거래, 기업 또는 공공 기관 간 전자 거래, 재택 은행 거래, 재택 주식 거래, 전자 지불 시스템 등의 안전성을 요하는 서비스의 출현으로 인하여 그 중요성은 더 커지고 있다. 이러한 디지털 서명의 핵심적인 요소 기술이 바로 해쉬 함수이다. 이 해쉬 함수는 임의 길이의 데이터를 입력으로 받아 일정 길이의 출력을 내는 알고리즘으로 출력 결과를 가지고 입력 데이터를 알아낼 수 없는 안전성을 보장하는 알고리즘이어야 한다.[7]

본 논문은 고속 네트워크 장비 및 보안 응용 제품에 쓰이는 데이터의 암복호화 및 디지털 서명, 인증을 위한 코프로세서의 역할을 하기 위한 주문형 반도체 집적 회로에 쓰일 해쉬 함수 기능의 성능 향상에 대하여 논하고자 한다. 일반적으로 칩의 처리 속도를 높이기 위해서는 동작 클럭의 속도를 높이는 방법을 많이 사용한다. 그러나 암복호화용 코프로세서를 PCI와 같은 시스템 버스의 클럭으로 동작시키고자 한다거나 응용에 따라서 고속 클럭을 사용하는 것에 무리가 따르는 응용 제품이라면 칩 자체가 빠른 클럭을 지원하는 것이 무의미해진다. 이러한 경우에는 오히려 동작 클럭 시간을 효율적으로 사용하지 못하는 것이 된다. 또는 여러 알고리즘을 한 칩에 넣고자 하는 시스템온칩을 개발하고자 할 경우에, 여러 알고리즘의 임계 경로의 차이가 발생한다. 이 경우 임계 경로가 충분한 여유를 지니고 있는 알고리즘의 경우, 반복 수행해야 하는 연산의 수를 배로 늘리고 반복 횟수를 줄이도록 하는 loop unrolling 기법을 적용하여 보다 빠른 결과 출력을 기대할 수 있다. 본 논문은 이처럼 저속 클럭을 사용하여 하드웨어를 개발해야 하는 경우, 클럭 시간을 효율적으로 사용하여 연산 처리 속도를 높일 수 있는 해쉬 함수의 구현에 관한 것이며 이때 적절한 하드웨어의 재배치를 통하여 최소한의 하드웨어 증가를 이룰 수 있다.

II. 해쉬 함수의 연산 구조

그림 1은 일반적으로 알려진 해쉬 연산의 단계를 보여주고 있다. SHA-1, MD5, HAS-160 등의 해쉬 함수는 입력된 데이터에 대하여 패딩 비트 첨부와 메시지 길이 비트 첨부 등의 전처리 과정을 거친 후 초기 벡터를 설정하고 압축 알고리즘

의 반복 연산을 수행하는 반복 프로세싱부를 거쳐 생성되는 결과 데이터를 출력하는 과정을 거친다.[7]

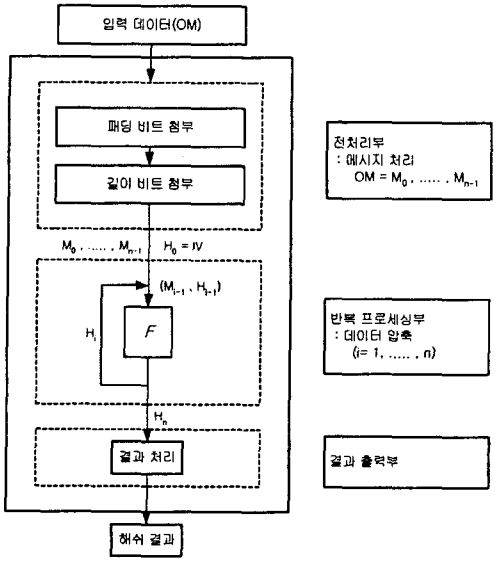

그림 1: 해쉬 알고리즘의 연산 구조

해쉬 함수의 입력되는 메시지는 512비트의 블록 단위로 처리된다. 단, 마지막 블록에서 입력 메시지의 길이를 표현하기 위해 64비트를 할당하기 때문에 한 블록에서 패딩 비트의 길이는 0~448이 되도록 부가되어야 한다. 패딩 방법은 메시지 다음에 1을 넣고 나머지 비트에 0을 채워 넣는다. 그림 1의 패딩 비트 첨부 단계에서 위와 같은 일을 한다. 해쉬 함수는 입력된 메시지의 크기를 항상 표시해 주도록 구성되어 있으며 패딩 비트 부가가 끝나면 64비트를 더 첨가해 메시지 길이를 표현한다. 메시지 길이 비트 첨부 단계에서 이 기능을 한다. 해쉬 함수에서는 해쉬 함수의 중간 결과와 최종 결과를 보관하기 위해 버퍼를 사용한다. 버퍼는 여러 개의 32비트 레지스터로 구성되었으며, 각 레지스터에 정해진 초기값을 지정하는 단계가 초기 벡터 설정 단계이다.[7]

반복 프로세싱부는 해쉬 함수의 가장 주요 단계로 데이터를 입력 받아 일정 길이의 출력을 만들어내는 단계이다. 역함수를 구하기 어려운 함수로, 한 개의 데이터 블록을 해쉬 기능 수행을 하는데 SHA-1과 HAS-160 함수의 경우는 80 번의 반복 수행이 MD5 함수의 경우는 64번의 반복 수행이 필요하다. 대다수의 해쉬 함수는 변환되어 출력된 값과 원래의 초기 벡터 값을 더하여 저장한다. 그리고 이 값을 출력하게 된다. 해쉬 출력 단계에서 이러한 기능을 하게 된다.

그림 1에서 볼 수 있듯이 해쉬 결과를 빨리 얻는 것은 반복 프로세싱부를 얼마나 빨리 처리하느냐에 달려있다. 주문형 반도체 집적 회로로 구현된 기존의 해쉬 함수들은 그림 1에서 보이고 있는 구조를 택하고 있다. 해쉬 구조는 정해져 있으므로 처리 속도를 높이는 방법은 주문형 반도체 집적 회로로 구현한 경우 동작 클럭 속도를 높이는 방법이 있다. 기존의 주문형 반도체 집적 회로로 구현된 해쉬 함수들은 임계 경로를 줄이고 클럭 속도를 높이는 방법으로 해쉬 연산 속도를 높여왔다. 대다수의 경우 이러한 문제 해결 방법이 적당하다. 하지만 몇몇 응용의 경우 동작 클럭이 정해져 있는 상태에서 제품을 개발해야 할 필요가 있다. 예를 들어 널리 알려진 PCI 시스템 클럭을 주문형 반도체 집적 회로의 동작 클럭으로 사용하는 경우는 클럭 속도가 33MHz 또는 66MHz로 일정하게 정해지고 이에 맞춰 제품을 설계해야 하는 경우이다. 또한 동작 클럭 속도를 높이게 되면 CMOS 커패시터 충방전 횟수가 많아지므로 전력 소모도 커지게 되고 반도체 칩 자체의 발열량도 증가하여 오동작이 발생할 수도 있으며 이를 해결하기 위한 또 다른 노력이 필요하게 된다. 본 논문은 이러한 문제를 해결하고 연산 성능을 높이기 위하여 loop unrolling 방법을 적용하여 동작 클럭을 낮추는 방법을 선택하였다.

III.효율적인 해쉬 함수 구현 방법

1. loop unrolling 방법을 이용

본 장에서는 해쉬 함수의 성능을 개선하기 위하여 loop unrolling 방법을 적용하는 방법에 대하여 설명한다. loop unrolling 방법이란 일정 연산을 반복하여 수행해야 하는 경우 한번에 수행할 연산의 수를 배로 배치하고 그 역으로 반복 횟수는 줄이도록 하는 방법이다. 본 논문에서는 여러 해쉬 함수 중 SHA-1 알고리즘을 예로 들어 설명하고자 한다. 앞서 설명한대로 해쉬 함수의 처리 속도를 얼마나 높일 수 있느냐는 반복 프로세싱부의 데이터 변환 동작 얼마나 줄일 수 있느냐에 있다. 수식 (1)은 SHA-1 해쉬 함수의 반복 프로세싱부에서의 한 라운드의 연산을 나타낸다.[2]

이때 R^{number}는 회전 연산을 나타내며 $F(X,Y,Z)$는 로직-게이트 연산을 나타낸다. 로직-게이트 연산식은 해쉬 함수마다 정해져 있으며 같은 해쉬 함수 중에도 반복 연산한 상태에 따라 다르다. SHA-1 해쉬 함수는 수식 (1)을 80번 반복 수행한 후 결과를 출력하게 된다. SHA-1 이외에 MD5와 HAS-160과 같은 다른 해쉬 함수들도 위 수식과 비슷한 식을 정해진 수만큼 반복하여 결과를 출력하는 구조이다.[1,2,3]

$$
\begin{aligned}
A_{N+1} &= \\
& R^5(A_N) + F(B_N, C_N, D_N) + E_N + W[N] + K \\
B_{N+1} &= A_N \\
C_{N+1} &= R^{30}(B) \\
D_{N+1} &= C_N \\
E_{N+1} &= D_N
\end{aligned}
\qquad (928)
$$

수식 (1)에 loop unrolling 방법을 적용하여 두 라운드를 처리해야 나올 수 있는 결과를 한 라운드 만에 나오도록 수식을 풀이보면 수식 (2)와 같다.

$$
\begin{aligned}
A_{N+2} &= R^5\left(\tfrac{1}{2}R^5(A_N)\right) + F(B_N, C_N, D_N) + E_N \\
&+ W[N] + K\tfrac{3}{4} + F\tfrac{1}{2}A_N, R^{30}(B_N), C_N\tfrac{3}{4} + D_N \\
&+ W[N+1] + K \\
B_{N+2} &= R^5(A_N) + F(B_N, C_N, D_N) + E_N \\
&+ W[N] + K \\
C_{N+2} &= R^{30}(A_N) \\
D_{N+2} &= R^{30}(B_N) \\
E_{N+2} &= C_N
\end{aligned}
\qquad (929)
$$

수식 (2)에서 볼 수 있듯이 연산이 증가하여 주문형 반도체 집적 회로 제작시 임계 경로는 길어지게 된다. 하지만 수식 (2)의 연산이 클럭 주기 안에 동작하도록 임계 경로를 구성하면 반복 프로세싱부의 반복 연산을 40번만 수행해도 수식 (1)에서 80번 반복 수행한 후의 결과와 같은 결과를 얻을 수 있으므로 그만큼의 해쉬 처리 속도 향상을 꾀할 수 있다. 또한 B_{N+2}의 경우는 A_{N+2}의 중간 결과를 뽑아내어 사용하면 되므로 하드웨어의 증가도 크게 늘어나지는 않게 된다. 수식 (2)의 결과는 loop unrolling 계수를 2로 하였을 경우로 계수를 높이면 높일수록 반복 연산의 횟수가 줄어들게 되므로 출력 결과를 더 빨리 얻을 수 있다.

2. 구성 모듈의 대체

다수의 해쉬 함수는 더하기 연산 작업과 회전 연산, 그리고 논리 게이트 연산으로 이루어져 있는데 이때 본 논문은 더하기 연산을 carry-save 더하기 연산과 carry-look-ahead 더하기 연산의 적절한 조합을 통하여 주문형 반도체 집적 회로 하드웨어의 임계 경로와 면적 사이의 협상을 고려할 수 있도록 하였다. 수식 (3)은 수식(2)의 A_{N+2}를 carry-save 더하기 연산과 일반 더하기 연산의 재배치를 통하여 최종 정리한 식이다.

이때 *CSA()*는 carry-save 더하기 연산을 나타내며 출력 값은 carry와 sum 두개이다. 그리고 *ADD()*는 일반 더하기 연산을 나타낸다. carry-save 더하기 연산의 경우 수식 (3)은 다소 복잡해 보이지만 하드웨이적으로 구현했을 경우는 carry-save 더하기 연산의 경우 일반 더하기 연산의 경우보다 1/4이상 면적 소모가 적기 때문에 총 하드웨어 면적이 줄어드는 결과를 볼 수 있으며, 임계 경로 또한 carry-save 더하기 연산의 경우는 굉장히 짧기 때문에 속도의 향상에도 충분한 효과를 볼 수 있다.

$$
\begin{aligned}
A_{N+2} &= ADD\tfrac{1}{2}CSA\tfrac{1}{2}R^5\tfrac{1}{2}ADD\tfrac{1}{2}CSA\tfrac{1}{2} \\
&R^5(A_N), CSA\tfrac{1}{2}F(B_N, C_N, D_N), CSA(\\
&E_N, W[N], K)\tfrac{3}{4}\tfrac{3}{4}\tfrac{3}{4}\tfrac{3}{4}, \\
&CSA\tfrac{1}{2}F\tfrac{1}{2}A_N, R^{30}(B_N), C_N\tfrac{3}{4}, CSA(\\
&D_N, W[N+2], K)\tfrac{3}{4}\tfrac{3}{4}\tfrac{3}{4}
\end{aligned}
\qquad (930)
$$

IV. 해쉬 함수의 구현

앞서 설명한 최종 결과인 수식 (3)을 반복 프로세싱부의 수식으로 입력하여 그림 1의 연산을 수행하게 되면 최종 원하는 해쉬 결과를 얻을 수 있게 된다.

그림 2는 SHA-1 해쉬 함수의 반복 프로세싱부의 구현 구성도를 보이고 있다. 굵은선으로 표시한 부분이 이 회로의 임계 경로가 된다. 즉, 입력 데이터가 게이트 연산을 거친 후 carry-save 더하기 연산 3개와 carry-look-ahead 더하기 연산 2개 그리고 회전 연산을 포함하는 경로를 말한다. 이러한 임계 경로가 시스템 클럭 주기 안에 동작하도록 구성하여 전체적인 동작에는 문제를 주지 않으면서 처리 속도는 높을 수 있는 장점이 있다.

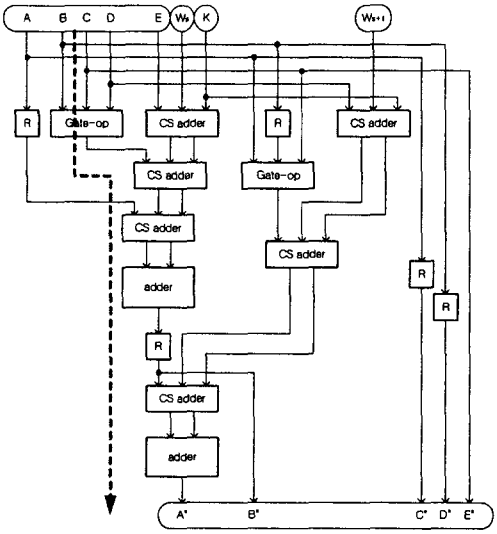

그림 2: SHA-1 해쉬 함수의 구현

구현된 해쉬 함수의 동작 파형을 그림 3에서 보이고 있다. 그림 3을 시간 간격으로 살펴보면 t1에서 t2까지는 해쉬 처리해야하는 데이터를 읽어오는 과정이며 t2에서 t3 구간에서는 패딩 작업이나 길이 비트 첨부와 초기 벡터 설정 등의 전처리 작업을 위한 것이다. t3부터 t4까지가 앞서 길게 설명한 반복 프로세싱부의 작업 처리 시간을 나타낸다. 그러므로 만약 loop unrolling 방법을 사용하지 않으면 SHA-1 해쉬 함수에서 80 클

럭 사이클이 필요하지만 본 논문의 방법을 적용하면 40 클럭 사이클만에 결과를 얻을 수 있다. t4에서 t5는 반복 프로세싱을 거친 후 최종 결과를 출력하는 시간을 나타낸다.

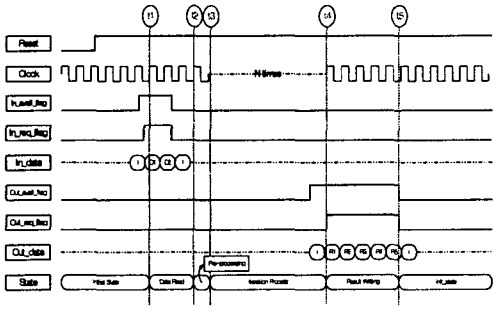

그림 3: 해쉬 함수 결과 파형

V.구현된 해쉬 함수의 성능

본 논문의 loop unrolling 방법을 적용하여 구현한 해쉬 함수를 반도체 집적 회로로 제작하였다. 0.35마이크론의 ATMEL 공정을 사용하였으며 동작 클럭은 33MHz이다. 표 1은 그 결과 처리 성능을 나타낸다.

표 1: 해쉬 함수의 처리 성능

Algorithm	Block size (bit)	Original Performance (Mbps)	Improved Performance (Mbps)
SHA-1	512	213	426
MD5	512	266	533
HAS-160	512	213	426

표 1에서 볼 수 있듯이 loop unrolling 기법을 적용하지 않은 해쉬 함수의 성능은 1/2 정도임을 알 수 있다.

면적적인 요소를 따져보면 다음과 같다. 우선 수식 (1)의 경우 carry-look-ahead 더하기 연산이 4개, 로직-게이트 연산이 1개가 사용되었다. 이때 회전 연산의 경우는 비트 위치만 바꿔주면 되므로 게이트 면적 소모는 없다. 수식 (3)의 경우는 carry-look-ahead 더하기 연산이 2개, carry-save 더하기 연산이 6개, 로직-게이트 연산이 2개가 소모되었다. 면적 측면에서 살펴보면 carry-save 더하기 연산은 carry-look-ahead 더하기 연산의 대략 1/4 정도이다.[8] 그리고 로직-게이트 연산과 carry-save 더하기 연산은 비슷한 면적을 차지한다.

이 경우 로직-게이트 연산의 면적을 1이라 가정하면 수식 (1)의 경우 17 (4x4+1), 수식 (3)의 경우 16 (2x4+6+2)으로 수식적으로 면적이 오히려 작은 결과를 얻을 수 있었다.

VI.결론

본 논문은 loop unrolling 방법을 적용하여 해쉬 함수를 반도체 집적 회로로 제작하였다. loop unrolling 방법을 적용함으로써 낮은 클럭 주파수에서 동작하더라도 최대 임계 경로값에 맞도록 로직을 설계하여 성능을 높일 수 있음을 보였다. 또한 내부 구성 모듈의 적절한 선택 활용 및 재배치를 통하여 성능 향상에 비하여 하드웨어의 증가를 억제할 수 있었다.

참고문헌

[1] IETF, The MD5 Message-Digest Algorithm, RFC 1321, April 1992.

[2] NIST, Secure Hash Standard (SHA-1), FIPS PUB 180-1, April 17, 1995.

[3] 한국정보통신기술협회, 해쉬함수표준-제2부: 해쉬함수알고리즘 (HAS-160), 1998년 11월.

[4]. IETF, The Use of HMAC-MD5-96 within ESP and AH, RFC 2403, November 1998.

[5] IETF, The Use of HMAC-SHA-1-96 within ESP and AH, RFC 2404, November 1998.

[6] IETF, HMAC: Keyed-Hashing for Message Authentication, RFC 2104, February 1997.

[7] 김철, 암호학의 이해, 영풍문고, pp165-204, 1996년 10월

[8] Keshab K. Parhi, VLSI Digital Signal Processing : Architectures and Algorithms, Kluwer Academic Publishers