메모리가 제한된 장치를 위한 효율적인 유한체 연산 알고리즘

(Efficient Algorithms for Finite Field Operations on Memory-Constrained Devices)

한 태 윤 † 이 문 규 ††
(Tae Youn Han) (Mun-Kyu Lee)

요 약 본 논문에서는 초소형 장치 상에서 적은 메모리만으로 효율적으로 연산 가능한 $GF(2^m)$ 상의 연산방법을 제안한다. 기존 구현들은 속도의 향상을 위한 곱셈연산 방법만을 제시하였으나, 본 논문에서는 곱셈 연산시 덧셈의 순서를 바꿈으로써 연산시 사용하는 메모리의 양을 줄이는 방법을 제시한다. 실험에 따르면, 본 논문에서 제안한 방법은 $GF(2^{271})$의 곱셈연산에서 이전에 제안된 방법들과 비교해 비슷한 수행 시간을 사용하면서 약 20% 적은 메모리 사용량을 보였다.

키워드 : 유한체, 이진체, $GF(2^m)$, 초소형 장치

Abstract In this paper, we propose an efficient computation method over $GF(2^m)$ for memory-constrained devices. While previous methods concentrated only on fast multiplication, we propose to reduce the amount of required memory by cleverly changing the order of suboperations. According to our experiments, the new method reduces the memory consumption by about 20% compared to the previous methods, and it achieves a comparable speed with them.

Key words : Finite field, Binary field, $GF(2^m)$, memory-constrained device

· 이 논문은 인하대학교의 지원에 의하여 연구되었음
· 이 논문은 제35회 추계학술대회에서 '초소형 장치를 위한 효율적인 유한체 연산 알고리즘'의 제목으로 발표된 논문을 확장한 것임

† 정 회 원 : 인하대학교 컴퓨터정보공학부
dadoo1@gmail.com
†† 종신회원 : 인하대학교 컴퓨터정보공학부 교수
mklee@inha.ac.kr
(Corresponding author임)

논문접수 : 2008년 12월 19일
심사완료 : 2009년 2월 21일

Copyright©2009 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

1. 서 론

최근 제한된 CPU, 메모리, 전력, 통신 대역을 이용하는 초소형 장치들이 내장형시스템, 센서 네트워크 등의 다양한 분야에 사용되고 있다. 또한 무선 통신을 사용하게 됨으로 안정성 문제가 대두되었으며 대칭키뿐만 아니라 상대적으로 연산량과 메모리사용량이 더 필요한 공개키 기반 암호의 사용도 필요로 하게 되었다. 공개키 암호인 페어링[1]과 ECC[2,3]의 효율적인 구현은 이전부터 많이 연구 되어 왔으나, 이전 연구들은 대부분 속도 향상에 중점을 두었으며, 속도를 줄이기 위해서 유한체의 효율적인 구현을 제안하였다. 그 이유는 ECC나 페어링의 거의 모든 연산은 유한체를 기본으로 하고 있기 때문이다. 그러나 실제로 초소형 장치에서는 연산능력 이외에 메모리 사용이 극히 제한적이므로, 메모리를 효율적으로 사용하는 방안에 대한 연구 또한 중요하다.

본 논문에서는 초소형 장치에서 공개키 암호의 효율적인 구현을 위해 유한체 연산에서 메모리를 효율적으로 사용하여 연산하는 방법을 제시하고 제시한 방법과 이전 연구결과의 속도와 메모리사용량을 비교 분석하였다.

2. $GF(2^m)$ 상의 기본 연산

이번 장에서는 유한체에서의 연산들에 대해서 알아본다. 이후에 다항식 a(x)는 다음과 같이 표현되어 진다.

$a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \dots + a_1x + a_0,\ a_i \in \{0,1\}$

또한 8비트 프로세서 상에서의 구현이기 때문에 a(x)는 8비트 워드를 가지는 A의 변수에 저장한다. 즉, a(x) = (A[m/8 -1], A[m/8 -1], ..., A[1], A[0]), A[t] = ($a_{8t+7}, a_{8t+6}, \dots, a_{8t+1}, a_{8t}$). 그리고 a(x)에서 하위 j개의 워드를 잘라낸 배열을 A{j}로 나타낸다. 즉, A{j} = A[m/8 -1], A[m/8 -1], ..., A[j+1], A[j].

아래에서는 유한체 $GF(2^m)$의 연산에 대해 살펴보겠다. 먼저, $GF(2^m)$ 상에서의 덧셈연산은 characteristic이 2이기 때문에 각 비트의 XOR연산으로 표현된다.

다음에, 곱하기 연산은 유한체상에서 가장 중요한 연산으로서, 가장 많이 사용되며 가장 많은 시간이 소요된다. 곱셈연산은 shift-and-add연산이 직관적이지만, 이는 소프트웨어 구현상에서 메모리 접근이 많아 좋지 않은 방법이다[3].

이를 빠르게 하기위해 나온 방법 중 하나는 Karatsuba-Ofman[4]이 있다. 이것은 분할정복 방법 중 하나로 연산량이 적은 덧셈연산의 추가로 곱하기의 복잡도를 줄이는 방법이다.

알고리즘 1 윈도우 4를 이용한 comb 곱셈방법

```
입력 : a(x) = (A[t-1], A[t-2], ... , A[0])
       b(x) = (B[t-1], B[t- 2], ... , B[0])
결과 : c(x) = a(x) · b(x)

 1: Compute T_u = u(x) · b(x) for all polynomials u(x) of degree at most 3
 2: C ← 0.
 3: for j ← 0 to t - 1 do
 4:     C{j} ← C{j} + T_{A[j]>>4}
 5: end for
 6: C ← C · x^4
 7: for j ← 0 to t - 1 do
 8:     C{j} ← C{j} + T_{A[j]&0x0F}
 9: end for
10: return c(x)
```

잘 알려진 다른 방법 중 하나는 윈도우를 사용하는 방법이다. 윈도우 방법은 속도를 개선하기 위해서 메모리를 사용하는 방법으로 미리 정의한 윈도우 크기에 대해서 가능한 다항식 u(x)를 가지고 u(x)·b(x)를 모두 미리 계산하여 메모리에 저장하여 연산중에 사용한다.

가장 대표적인 윈도우 방법 중 하나인 Comb 방법(알고리즘 1)은 $b(x)\cdot x^k$을 알고 있을 경우 $b(x)\cdot x^{Wj+k}$(단, W는 워드의 크기이다.)의 값은 배열의 위치만 바꿈으로서 쉽게 계산할 수 있음을 관찰하여 만든 방법이다.

한편 LD 방법 [1]은 알고리즘 1의 두 메인루프를 연관성을 찾아 메모리 접근의 수를 줄여 속도를 개선한 방법이다. 이는 한 워드의 상위 4비트와 하위 4비트에 관련된 연산을 한 번에 함으로서 결과에 대한 메모리 접근수를 줄였다.

또한 메인루프 안에서 워드와 다음 워드 사이의 연관성을 찾아 결과에 덧셈 연산을 함께하여 메모리 접근수를 줄여 속도를 개선한 SHKH방법이 있다[3].

한편, 제곱연산은 곱셈 연산의 특별한 형태로, 원소 a(x)의 연속된 비트 사이에 0을 넣음으로서 빠르게 연산을 할 수 있다. 또한 이 연산은 lookup table을 사용하면 더욱 빠르게 연산할 수 있다.

마지막으로, 모든 곱셈, 제곱의 결과는 주어진 기약다항식 f(x)로 모듈러 감산연산을 한다. 우리가 사용한 기약 다항식은 TinyPBC[1]에서 사용한 다항식으로 $f(x) = x^{271}+x^{207}+x^{175}+x^{111}+1$ 이며, 이는 제곱근 연산을 빠르게 하기위해 결정을 하였다[5]. 유한체 상에서의 모듈러 감산 연산을 빠르게 하기 위해서는 NIST에서 권고한 기약다항식이 있다. 그러나 우리가 사용한 다항식은 NIST에서 권고한 기약다항식은 아니지만 같은 방법을 이용하여 빠르게 연산할 수 있다[5].

3. 개선된 곱셈연산

이전에 연구 되었던 Comb, TinyPBC[1], SHKH[3]는 윈도우 크기 4를 사용하여 15개의 유한체 원소를 저장하기 때문에 많은 메모리를 사용 하였다. 메모리를 줄이기 위해서 윈도우 크기 2를 사용하는 것도 고려 할 수 있으나 이렇게 하면 속도가 지나치게 느려지는 문제가 생기게 된다. 이 절에서는 같은 연산 시간을 필요로 하면서 메모리를 적게 사용하는 방법을 제안한다.

3.1 메모리 최적화

메모리를 줄이는 가장 좋은 방법은 미리 계산을 해 놓는 테이블 수를 줄이는 것이다. 제안하는 방법은 단순히 메모리의 크기만 줄여 윈도우의 크기만을 줄인 것이 아닌, 연산 순서를 바꾸어 메모리 접근의 수를 줄인 방법이다.

이전에 제안된 방법들은 a(x)의 값을 순차적으로 읽어 읽은 값에 해당하는 값을 미리 계산된 테이블에서 찾아 더하는 방식을 가졌었다. 그러나 제안된 방법은 Temp 변수를 사용하여 연산을 하는 중간에 테이블을 갱신하여 사용하는 방법을 사용한다. Temp 변수를 효율적으로 사용하기 위해서 덧셈 순서를 a(x)을 A[j]에서 j의 순서대로 읽는 것이 아니라, a(x)에 들어있는 값의 순서대로 연산을 하였다. 즉, 1·b(x)를 계산을 해놓고 a(x)를 읽으면서 A[j] 값이 1·b(x)인 부분을 찾아 사용을 한 후 Temp 변수를 x·b(x)로 계산해 놓고 a(x)의 값을 다시 읽으면서 x·b(x)가 필요한 부분에 사용하는 방식이며, 이를 $(x^3+x^2+x+1)\cdot b(x)$에 대한 부분까지 반복한다. 이렇게 하면 유한체 원소를 저장하는 임시 공간을 15개에서 1개(Temp)로 줄어든다.

3.2 속도 최적화

3.1에서 제안한 직관적인 방법은 메모리 사용량을 현저히 줄일 수 있으나 테이블을 필요에 따라 변경하는데 필요한 오버헤드로 인해 연산 시간이 늘어나는 문제가 있다. 따라서 이절에서는 테이블, 즉 메모리 사용을 최소로 하면서 속도를 이전 방법 수준으로 유지하는 방법을 제시한다.

이 절에서 제안하는 방법의 기본 아이디어는 연산의 순서를 변경하는 것이다. 연산을 $1\cdot b(x)$, $x\cdot b(x)$, ..., $(x^3+x^2+x+1)\cdot b(x)$와 같이 차례대로 한다면 테이블을 한 번에 연산을 해 놓는 것이 아니기 때문에 연산이 많아진다. 예를 들어 $(x+1)\cdot b(x)$ 이후에 $x^2\cdot b(x)$를 연산하기 위해서는 3번의 덧셈(XOR)연산을 하게 된다. 즉, $u_1\cdot b(x)$에서 $u_2\cdot b(x)$로 갱신을 할 경우 u_1과 u_2의 Hamming distance만큼의 덧셈연산을 하게 된다. 그래서 우리는 Temp 변수의 값을 갱신하는 순서를 u_1과 u_2 사이에 Hamming distance를 고려하여 정하였다.

그러나 위 방법만으로는 속도상의 손실을 완전히 만회 할 수는 없으며, a(x)를 읽는 횟수를 줄여야한다. 즉, 예전의 방법은 a(x)를 연산과정에서 1회 읽게 되어있는 반면에 위의 방법은 15회 읽게 되어있다. 이를 개선하기 위해서는 Temp의 크기를 늘려야 한다. 우리는 실험에서 Temp의 크기를 3개로 하였다.

표 1 메모리가 3개일 경우 연산 순서

순서	T 1	T 2	T 3
1	0010·b(x)	0100·b(x)	0110·b(x)
2	0011·b(x)	0101·b(x)	0111·b(x)
3	1101·b(x)	1011·b(x)	1110·b(x)
4	1100·b(x)	1010·b(x)	1111·b(x)
5	1001·b(x)		
6	1000·b(x)		
7	1000·b(x)		
8	1001·b(x)		
9	1100·b(x)	1010·b(x)	1111·b(x)
10	1101·b(x)	1011·b(x)	1110·b(x)
11	0011·b(x)	0101·b(x)	0111·b(x)
12	0010·b(x)	0100·b(x)	0110·b(x)

표 1은 Temp의 개수를 T1, T2, T3의 3개로 했을 경우 a(x)를 처리하는 값들을 차례대로 나타낸 것이다. 표시된 숫자는 다항식을 이진법으로 나타낸 것으로 예를 들어 0111·b(x)는 $(x^2+x+1)\cdot b(x)$를 나타낸다. 순서 1-6은 하위 4비트를 연산할 경우의 순서이고 7-12는 상위 4비트를 연산할 경우의 순서이다. 하위 비트를 연산 후에 상위 비트를 바로 연산하기 때문에 이전에 있던 메모리를 그대로 사용하기 위해서 순서를 역순으로 했으며 순서 5, 6과 순서 7, 8을 또한 합치지 않고 나누어 계산하였다.

3.3 알고리즘

그림 1은 제안한 방법을 이용하여 $c(x) = a(x)\cdot b(x)$를 계산하는 알고리즘의 순서도이다. 이 알고리즘은 T1, T2, T3의 값을 갱신하면서 결과 다항식 c(x)의 적절한 위치에 T1~T3중 필요한 값을 가져와 더하는 일을 반복하도록 구성되어 있다. i가 1-6일 경우에는 다항식 a(x)의 8비트 워드표현에서 j번째 워드인 A[j]의 상위 4비트를 보면서 연산을 하며 i가 7-12일 경우에는 A[j]의 하위 4비트를 보면서 연산을 하도록 되어 있다. 정확한 알고리즘은 부록에 제시되어 있다.

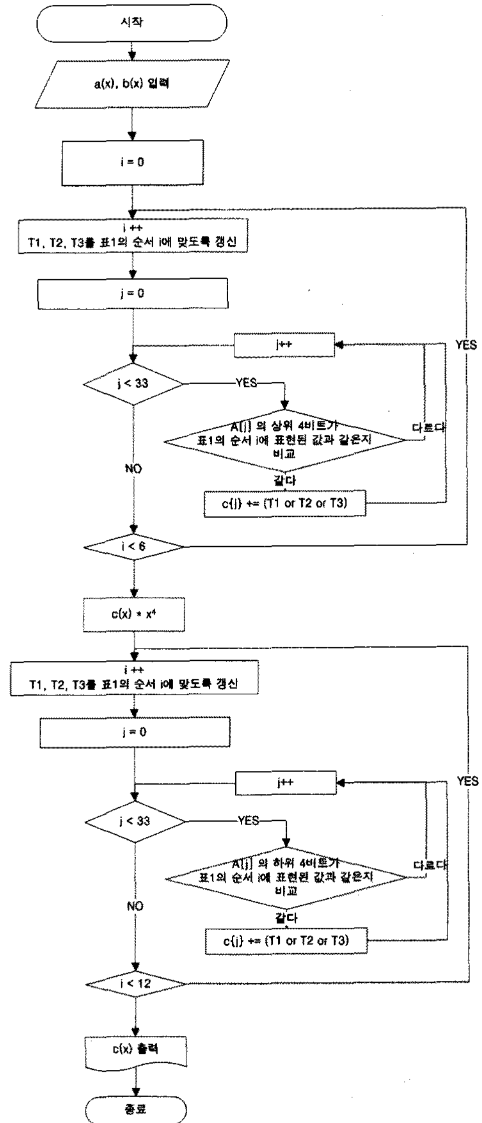

그림 1 제안된 알고리즘

4. 구현 결과 및 분석

본 논문에서 Comb, LD, SHKH 그리고 제안한 방법을 가지고 페어링 연산을 하였을 때의 시간과 곱셈에 사용된 메모리를 비교 분석 하였다.

사용된 유한체는 $GF(2^{271})$이며, embedding degree 4를 갖는 타원곡선 $y^2+y=x^3+x^2$를 사용하였다.

개발환경은 Avr-gcc(4.12)을 이용하였으며, 타깃을 ATmega128로 하여 AVR-studio로 시뮬레이션 하였다. ATmega128은 8비트 프로세서로 7.3828MHz의 클럭을 가지며 128KB의 롬과 4KB의 램을 가지고 있다. 곱셈 이외의 대부분의 필드연산은 MIRACL[6]을 사용하였다.

다음의 표들은 페어링연산을 1회 수행 하였을 때 사용된 램, 롬, 스택의 크기와 시간을 나타낸다. 페어링 연산은 다양한 인자를 갖는 다수의 곱셈 연산을 포함하고 있으므로, 1회 수행에 대한 측정으로 성능 비교에 충분히 활용할 수 있다. 스택은 곱셈 이외에 페어링 연산을 하기위해 사용된 메모리이기 때문에 모든 방법에서 같은 크기를 갖는다.

표 2는 Karatsuba 방법을 1번 적용하였을 때의 성능을 보여주고, 표 3은 Karatsuba 방법을 사용하지 않았을 경우의 성능을 보여주고 있다. 표 2, 3에서 괄호안의 수는 사용된 윈도우의 크기를 나타낸다. 우리의 방법 중에서

표 2 Karatsuba를 사용한 페어링 연산

	시간 (s)	램 (Byte)	스택 (Byte)	롬 (Byte)
Comb.(2k)	9.67	221	3,428	43,764
LD(2k)	8.08	239	3,428	44,586
SHKH(2k)	8.02	239	3,428	46,962
Comb.(4k)	6.28	439	3,428	43,340
LD(4k)	5.44	439	3,428	44,318
SHKH(4k)	5.12	439	3,428	44,644
제안된방법	11.4 2	246	3,428	46,588

표 3 Karatuba를 사용하지 않은 페어링 연산

	시간 (s)	램 (Byte)	스택 (Byte)	롬 (Byte)
Comb.(2)	10.47	150	3,428	41,754
LD(2)	17.39	150	3,428	42,042
SHKH(2)	9.16	150	3,428	42,260
Comb.(4)	6.73	572	3,428	42,054
LD(4)	7.34	572	3,428	42,302
SHKH(4)	5.96	572	3,428	42,246
제안된방법	7.90	192	3,428	42,796

Karatsuba 방법을 사용하지 않는 것이 더 유리하였다.

표 3의 제안된 방법은 표 2의 LD(2k)와 SHKH(2k)에 비해 비슷한 연산시간을 가지고 있지만 약 20% 적은 메모리 사용량을 보여주고 있다. 한편 비슷한 메모리를 쓰는 알고리즘들을 대상으로 비교하면, comb(2k)에 비해서 18%의 속도 향상을 보였다.

한편, 내장형 시스템에서는 수행속도나 메모리 사용량 이외에도 에너지 소모나 발열 등의 문제도 중요하다. 발열은 결국 에너지 소모와 밀접하게 연관되므로 에너지 소모에 대해 살펴보기로 한다.

메모리의 에너지 사용량은 메모리 소모를 수십 바이트 줄인다 해도 메모리 디바이스 자체를 끌 수 있는 것은 아니기 때문에 크게 영향을 미치지 않을 것으로 보인다. 그러나 단위 시간당 에너지 소비량이 같다면 에너지는 시간에 비례하기 때문에 연산 시간이 적은 쪽에서 에너지가 적게 사용될 것으로 판단된다. 따라서, 같은 메모리를 사용한 경우 기존 방법에 비해 속도가 빠른 본 논문의 방법이 에너지 소모 측면에서 유리하다 할 수 있겠다.

또한 Dynamic Voltage Scaling(DVS) 기능이 있는 CPU를 사용할 경우, 본 논문의 결과에 따르면 같은 연산을 위해 더 적은 클럭을 사용할 수 있으므로 같은 데드라인이 주어질 때 더 낮은 전압으로 연산 가능하다. 에너지 소모는 전압의 제곱에 비례 하므로 본 논문에서 제안한 연산 방식을 사용하면 에너지 소모를 줄일 수 있을 것으로 기대된다[7].

5. 결론 및 향후 연구

본 논문에서는 초소형 장치를 위한 유한체 연산 알고리즘을 제안 하였으며 이를 이전 방법들과 비교 하였다. 속도중심 곱셈 연산 방법에 비교해서 제안된 방법은 비슷한 속도에서 20%가량 적은 메모리 사용량을 보였으며, 비슷한 메모리 사용량을 보이는 알고리즘과 비교를 해보았을 경우에는 18%의 속도 향상을 보였다. 제안한 방법은 초소형 장치에서 암호 시스템에 구현함에 있어서 매우 효과적이라 할 수 있다.

향후의 연구 과제로는 페어링뿐만 아니라 ECC와 같은 $GF(2^m)$을 사용하는 다른 암호시스템에서의 성능을 분석하는 연구가 요구된다. 또한 유한체의 종류를 $GF(2^{271})$ 이외에 표준에 제안되고 지금 많이 사용되고 있는 $GF(2^{163})$에도 적용하여 성능을 분석하는 연구가 요구된다.

참 고 문 헌

[1] L.B. Oliveira, M.Scoot, J.Lopez, and R. Dahab, TinyPBC : Pairings for authenticated identity-based noninteractive key distribution in sensor networks, Cryptology ePrint Archive, Report 2007/482, 2007.

[2] A. Liu and P.Ning, TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks, Proceeding of the 2008 International Conference on Information Processing in Sensor Networks (IPSN 2008), Washington, DC, USA, pp. 245-256, IEEE Computer Society, 2008.

[3] S.C.Seo, D.G.Han, H.C.Kim, and S.Hong, TinyECCK : Efficient elliptic curve cryptography implementation over $GF(2^m)$ on 8-bit micaz mote, IEICE Transactions, Vol.42, No.3, pp. 239-271, 2007.

[4] A. Karatsuba And Y. Ofman, Multiplication of multidigit numbers on automata, Soviet Physics-Doklad, Vol.7, No.7, pp. 595-596, 1963.

[5] M.Scott, Optimal irreducible polynomials for $GF(2^m)$ arithmetic, Cryptology ePrint Archive, Report 2007/192, 2007.

[6] M.Scott, MIRACL - A Multiprecision Integer and Rational Arithmetic C++ Library, Shamus Software Ltd., Dublin, Ireland, 2003.

[7] BURD, T. D., AND BRODERSEN, R. W. Energy efficient CMOS microprocessor design. In Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture (Los Alamitos, CA, USA, Jan. 1995), T. N. Mudge and B. D. Shriver, Eds., IEEE Computer Society Press, pp. 288-297.

부록 : 제안하는 곱셈 알고리즘

알고리즘 2는 제안한 곱셈 알고리즘의 의사코드이다. 이는 8비트 프로세서를 기준으로 작성이 되어있다. 한

개의 for문은 표 1에서의 순서 하나에 해당된다. 즉, 줄 14-23이 순서 2에 해당된다. 줄 14는 사용할 Temp 변수를 갱신하는 단계이고 줄 16-17, 18-19, 20-21은 각각 a(x)의 값을 읽었을 때 0011·b(x), 0101·b(x), 0111·b(x)의 값이 있는 곳에서 덧셈을 해주는 단계이다.

알고리즘 2 제안된 곱셈 알고리즘

```
입력 : a(x), b(x)
결과 : c(x) = a(x)b(x)

  1: c(x) = 0
  2: T1 ← b(x)·x; T2 ← b(x)·x2; T3 ← T1+T2
  3: for j ← 0 to 33 do
  4:     if (A[j]&0xF0) = 0x20 then
  5:         C{j} ← C{j}+T1
  6:     else if (A[j]&0xF0) = 0x40 then
  7:         C{j} ← C{j}+T2
  8:     else if (A[j]&0xF0) = 0x60 then
  9:         C{j} ← C{j}+T3
 10:     else if (A[j]&0xF0) = 0x10 then
 11:         C{j} ← C{j}+b(x)
 12:     end if
 13: end for
 14: T1 ← T1+b(x); T2 ← T2+b(x);
     T3 ← T3+b(x)
 15: for j ← 0 to 33 do
 16:     if (A[j]&0xF0) = 0x30 then
 17:         C{j} ← C{j}+T1
 18:     else if (A[j]&0xF0) = 0x50 then
 19:         C{j} ← C{j}+T2
 20:     else if (A[j]&0xF0) = 0x70 then
 21:         C{j} ← C{j}+T3
 22:     end if
 23: end for
 24: T3 ← T3·x; T1 ← T1+T3; T2 ← T2+T3
 25: for j ← 0 to 33 do
 26:     if (A[j]&0xF0) = 0xD0 then
 27:         C{j} ← C{j}+T1
 28:     else if (A[j]&0xF0) = 0xB0 then
 29:         C{j} ← C{j}+T2
 30:     else if (A[j]&0xF0) = 0xE0 then
 31:         C{j} ← C{j}+T3
 32:     end if
 33: end for
 34: T1 ← T1+b(x); T2 ← T2+b(x);
     T3 ← T3+b(x)
 35: for j ← 0 to 33 do
 36:     if (A[j]&0xF0) = 0xC0 then
 37:         C{j} ← C{j}+T1
 38:     else if (A[j]&0xF0) = 0xA0 then
 39:         C{j} ← C{j}+T2
 40:     else if (A[j]&0xF0) = 0xF0 then
 41:         C{j} ← C{j}+T3
 42:     end if
 43: end for
 44: T1 ← T1+T2+T3
 45: for j ← 0 to 33 do
 46:     if (A[j]&0xF0) = 0x90 then
 47:         C{j} ← C{j}+T1
 48:     end if
 49: end for
 50: T1 ← T1+b(x)
 51: for j ← 0 to 33 do
 52:     if (A[j]&0xF0) = 0x80 then
 53:         C{j} ← C{j}+T1
 54:     end if
 55: end for
 56: c(x) ← c(x)·x4
 57: for j ← 0 to 33 do
 58:     if (a[j]&0x0F) = 0x08 then
 59:         C{j} ← C{j}+T1
 60:     end if
 61: end for
 62: T1 ← T1+b(x)
 63: for j ← 0 to 33 do
 64:     if (a[j]&0x0F) = 0x09 then
 65:         C{j} ← C{j}+T1
 66:     end if
 67: end for
 68: T1 ← T1 + T2 + T3
 69: for j ← 0 to 33 do
 70:     if (A[j]&0x0F) = 0x0C then
 71:         C{j} ← C{j} + T1
 72:     else if (A[j]&0x0F) = 0x0A then
 73:         C{j} ← C{j} + T2
 74:     else if (A[j]&0x0F) = 0x0F then
 75:         C{j} ← C{j} + T3
 76:     end if
 77: end for
 78: T1 ← T1+b(x); T2 ← T2+b(x);
     T3 ← T3+b(x)
 79: for j ← 0 to 33 do
 80:     if (A[j]&0x0F) = 0x0D then
 81:         C{j} ← C{j}+T1
 82:     else if (A[j]&0x0F) = 0x0B then
 83:         C{j} ← C{j} + T2
 84:     else if (A[j]&0x0F) = 0x0E then
 85:         C{j} ← C{j} + T3
 86:     end if
 87: end for
 88: T1 ← T1+T3; T2 ← T2+T3; T3 ← T3/x;
 89: for j ← 0 to 33 do
 90:     if (A[j]&0x0F) = 0x03 then
 91:         C{j} ← C{j} + T1
 92:     else if (A[j]&0x0F) = 0x05 then
 93:         C{j} ← C{j} + T2
 94:     else if (A[j]&0x0F) = 0x07 then
 95:         C{j} ← C{j} + T3
 96:     end if
 97: end for
 98: T1 ← T1+b(x); T2 ← T2+b(x);
     T3 ← T3+b(x)
 99: for j ← 0 to 33 do
100:     if (A[j]&0x0F) = 0x02 then
101:         C{j} ← C{j} + T1
102:     else if (A[j]&0x0F) = 0x04 then
103:         C{j} ← C{j} + T2
104:     else if (A[j]&0x0F) = 0x06 then
105:         C{j} ← C{j} + T3
106:     else if (A[j]&0x0F) = 0x01 then
107:         C{j} ← C{j} + b(x)
108:     end if
109: end for
110: return c(x)
```