Danuta PAMUŁA [1,2], Edward HRYNKIEWICZ [1], Arnaud TISSERAND [2]
[1] POLITECHNIKA ŚLĄSKA, ul.Akademicka 16, 44-100 Gliwice
[2] IRISA, CNRS, INRIA CENTRE RENNES – BRETAGNE, UNIVERSITY OF RENNES

Analiza algorytmów mnożenia w ciele GF(2^m)

Mgr inż. Danuta PAMUŁA

Ukończyła studia na Wydziale Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach w 2008 roku. Otrzymane w 2009 roku stypendium Rządu Francuskiego pozwoliło jej na prowadzenie badań związanych z doktoratem na Politechnice Śląskiej w Gliwicach oraz w laboratorium IRISA we Francji, pod kierunkiem dwóch promotorów. Jej zainteresowania naukowe dotyczą układów programowalnych, kryptografii i arytmetyki komputerów.

e-mail: dpamula@polsl.pl / danuta.pamula@irisa.fr

Dr hab. inż. Edward HRYNKIEWICZ

Ukończył studia na Wydziale Automatyki Politechniki Śląskiej w Gliwicach. Rozprawę doktorska obronił w roku 1978, a habilitował się w roku 1992 w dyscyplinie elektronika. Zajmuje się syntezą logiczną, układami programowalnymi, programowalnymi sterownikami logicznymi oraz cyfrowymi układami przetwarzania częstotliwości. Obecnie pełni funkcję dyrektora Instytutu Elektroniki na Wydziale Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach.

e-mail: edward.hrynkiewicz@polsl.pl

Ph.D. Arnaud TISSERAND

Rozprawę doktorską w dziedzinie Informatyki obronił w 1997 w École Normale Supérieure w Lyon, we Francji. Jest członkiem CNRS i obecnie prowadzi badania w laboratorium IRISA, we Francji. Jego zainteresowania dotyczą między innymi arytmetyki i architektury komputerów, projektowania układów cyfrowych, narzędzi CAD, cyfrowego przetwarzania sygnałów i kryptografii.

e-mail: arnaud.tisserand@irisa.fr

Streszczenie

Artykuł przedstawia analizę algorytmów mnożenia w ciele $GF(2^m)$. Algorytmy analizowane są pod kątem ich możliwości implementacji w sprzęcie. Badane są ich wady i zalety w celu ułatwienia projektantom kryptosystemów opartych na krzywych eliptycznych podjęcia decyzji co do tego jakiego algorytmu mnożenia w ciele skończonym użyć aby stworzone urządzenie było wydajne i nie zajmowało nadmiernej ilości zasobów.

Słowa kluczowe: kryptografia krzywych eliptycznych, GF(2^m), mnożenie.

Direct multiplication over GF(2^m) – analysis

Abstract

Cryptographic systems are based on mathematical theories, thus they strongly depend on the performance of arithmetic units comprising them. If an arithmetic operator does not take a considerable amount of resources or is time non efficient, it negatively impacts the performance of the whole cryptosystem. The purpose of this paper is to analyse the hardware possibilities of the algorithms performing multiplication in $GF(2^m)$ which are used for elliptic curve cryptography(ECC) applications. There are only two operations defined in this field: addition considered as a trivial one, it is a simple bitwise *xor* ,and multiplication - a very complex operation. To conform to the requirements of ECC systems, the multipliers should be fast, area efficient and, what is the most important, perform multiplication of big numbers (100 – 600 bit). The paper presents analysis of $GF(2^m)$ two-step modular multiplication algorithms. It considers classical (school) multiplication, matrix-vector approach and Karatsuba - Ofman algorithm, exploring thoroughly their advantages and disadvantages.

Keywords: ECC, finite fields, GF(2^m), multiplication, Karatsuba-Ofman.

1. Wstęp

W dzisiejszych czasach systemy kryptograficzne odgrywają coraz większa rolę, więc, aby dostosować się do ciągle zwiększających się prędkości działania i wymagań bezpieczeństwa systemów komputerowych, projektanci sprzętu muszą tworzyć coraz wydajniejsze urządzenia kryptograficzne. Z drugiej strony twórcy systemów kryptograficznych wykorzystują coraz bardziej złożone algorytmy algebraiczne w celu doskonalenia poziomu gwarantowanych zabezpieczeń. Z tego powodu bez efektywnych jednostek wykonujących operacje arytmetyczne żaden projektant nie jest w stanie stworzyć bezpiecznego i wydajnego kryptosystemu.

Coraz większą popularność zdobywa ostatnio kryptografia oparta na krzywych eliptycznych zdefiniowanych nad ciałem skończonym. Dzieje się tak między innymi ze względu na fakt, że gdy używana jest w systemach kryptograficznych z kluczem publicznym wymaga użycia kluczy mniejszych rozmiarów niż kryptografia oparta na klasycznych teoriach matematycznych (np. algorytm RSA), zapewniając ten sam lub wyższy poziom bezpieczeństwa.

Budując kryptosystem zawsze powinniśmy zaczynać od budowania najbardziej podstawowych jednostek arytmetycznych. W przypadku kryptografii opartej na krzywych eliptycznych zdefiniowanych nad ciałem skończonym takimi jednostkami są te, które wykonują operacje w ciałach skończonych. Najbardziej popularnymi ciałami skończonymi [1] wykorzystywanymi w kryptografii krzywych eliptycznych są $GF(p)$, gdzie p jest liczbą pierwszą, i $GF(2^m)$, tak zwane rozszerzenia ciał binarnych. W tej pracy zajmować się będziemy algorytmami zdefiniowanymi dla ciał $GF(2^m)$. Intencją autorów jest analiza, która pomogłaby by projektantom sprzętu zdecydować, którego algorytmu użyć dla implementacji danej operacji w ciele $GF(2^m)$.

2. Teoria skończonych ciał binarnych

Ciało składa się ze zbioru $\boldsymbol{F}$ i dwóch operacji: dodawania i mnożenia. Jeżeli zbiór $\boldsymbol{F}$ jest skończony wtedy mówimy, że ciało jest również skończone, czyli zawiera skończoną ilość elementów. Ciała skończone nazywane są również ciałami Galois i oznaczane są przez $GF(q = p^m)$ lub $F_{q\text{-}p}{}^m$, gdzie p nazywane jest charakterystyką pola. Tutaj będziemy zajmować się rozszerzeniami ciał binarnych $GF(2^m)$ (gdzie p=2) [2]. Do stworzenia takiego ciała skończonego używa się nierozkładalnego wielomianu *f(x)* stopnia *m*-tego:

$$f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_2x^2 + f_1x + 1, \qquad (1)$$

gdzie $f_i \in GF(2)$.

Ciało możemy potraktować jako przestrzeń wektorową, której elementy reprezentowane są za pomocą specyficznej bazy – w tym przypadku bazy wielomianowej: {1, x, x^2,..., x^{m-2}, x^{m-1}}. Przy użyciu tej bazy element ciała oznaczamy jako:

$$A = \sum_{i=0}^{m-1} a_i x^i = a_0 + a_1x + a_2x^2 + \ldots + a_{m-2}x^{m-2} + a_{m-1}x^{m-1}, \qquad (2)$$

gdzie $a_i \in GF(2)$. W ciele zdefiniowane są dwie operacje: dodawanie i mnożenie. Operacja dodawania jest uznawana za trywialną, ponieważ można ją zrealizować jako operację XOR. Warto jednak zauważyć, że wykonując operację sumy modulo 2 na bardzo dużych wektorach, sprzętowa implementacja tej trywialnej operacji może pochłonąć wiele zasobów sprzętowych. Operacja mnożenia natomiast jest bardziej złożona i jest zdefiniowana następująco: niech $A,B \in GF(2^m)$, będą wielomianami stopnia *(m-1)* i niech *f(x)*, będzie nieredukowalnym wielomianem generującym ciało. Wtedy

$$C = A \cdot B \bmod F(x), \tag{3}$$

gdzie C jest także elementem tego ciała. Jest to iloczyn dwóch wielomianów zredukowany modulo generator ciała $f(x)$ [1].

Istnieje wiele algorytmów wykonujących mnożenie w rozszerzonych ciałach binarnych [3]. Można je podzielić na dwie grupy: dwustopniowe oraz naprzemienne. Dwustopniowe algorytmy wykonują najpierw mnożenie elementów, a następnie redukują otrzymany iloczyn. W algorytmach naprzemiennych natomiast jednocześnie dokonuje się mnożenia i redukcji elementów. Ten artykuł analizuje i porównuje tylko pierwszą grupę algorytmów, czyli algorytmy dwustopniowe.

3. Dwustopniowe algorytmy mnożenia

Niech A, B, C będą wielomianami stopnia $(m-1)$ należącymi do ciała $GF(2^m)$, $f(x)$ nieredukowalnym wielomianem generującym ciało, a D iloczynem A,B , czyli wielomianem o stopniu 2m-2. W dwustopniowym algorytmie mnożenia wykonujemy najpierw mnożenie: $D = A \cdot B$, a następnie redukcję: $C=D \bmod F(x)$. Najbardziej popularnymi metodami wykorzystywanymi przy dwustopniowy mnożeniu są: metoda klasyczna (szkolna), podejście macierzowo-wektorowe oraz podejście typu „dziel-i-rządź" zazwyczaj z wykorzystaniem sztuczki Karatsuby-Ofmana [4]. Operacja redukcji może zostać zaimplementowana za pomocą operacji mnożenia, dlatego nie jest tutaj rozważana.

Urządzenia przedstawione w artykule zostały opisane w języku VHDL oraz syntezowane bez dodatkowej optymalizacji i implementowane z użyciem środowiska Xilinx ISE 9.2 oraz 12.1. Przetestowano je w układzie FPGA typu Spartan3E 1200 firmy Xilinx. Układ ten został wybrany ze względu na swoje niewielkie rozmiary i z powodu tego, że nie zawiera żadnych dodatkowych podzespołów, co jest częste np. w układach rodziny Virtex.

3.1. Klasyczne algorytmy mnożenia

Jedną z najpopularniejszych metod mnożenia wielomianów jest metoda przesuń-i-dodaj [5]. Opiera się ona na następujących rozważaniach:

$$\begin{aligned} D = A \cdot B = & \left(\sum_{i=0}^{m-1} a_i x^i\right) \cdot \left(\sum_{i=0}^{m-1} b_i x^i\right) = \\ & b_0\left(\sum_{i=0}^{m-1} a_i x^i\right) + b_1 x\left(\sum_{i=0}^{m-1} a_i x^i\right) + b_2 x\left(\sum_{i=0}^{m-1} a_i x^i\right), \\ & + \ \ldots \ + \\ & b_{m-2} x^{m-2}\left(\sum_{i=0}^{m-1} a_i x^i\right) + b_{m-1} x^{m-1}\left(\sum_{i=0}^{m-1} a_i x^i\right) \end{aligned} \tag{4}$$

Czyli kolejno mnożymy każdy współczynnik wielomianu $b(x)$ razy wielomian $a(x)$ i dodajemy rezultaty do siebie. W tym wypadku mnożenie jest zwykłym przesunięciem, a dodawanie operacją XOR. Algorytm można przenieść w sprzęt bezpośrednio, jako układ kombinacyjny lub podzielić go na kolejne kroki i stworzyć urządzenie działające sekwencyjnie.

Na początek zbadano wariacje układu czysto kombinacyjnego, zaimplementowanego zgodnie z równaniem (4). Pierwszym badanym układem mnożącym był układ 4-bitowy, którego synteza dała obiecujące rezultaty. Następnie zbadano układy mnożące większe liczby, niestety okazało się, że wraz z podwojeniem rozmiaru wielomianów wejściowych liczba zajętych bloków LUT rośnie około 4-krotnie. Zgodnie z tą zależnością wyliczono, że dla 256-bitowych wektorów wejściowych moduł zajmie ponad 52000 bloków LUT. Układ Spartan3E 1200 posiada 17344 bloków LUT, czyli kombinacyjna implementacja równania (4) dla 256-bitowego wektora wejściowego znacznie przekracza jego rozmiary.

Podsumowując można stwierdzić, że aby moduł mnożący duże liczby, oprócz bycia szybkim zajmował również odpowiednio mało zasobów sprzętowych, nie należy projektować go jako układu czysto kombinacyjnego. Ponieważ tworzenie czysto kombinacyjnego układu dało niezadowalające rezultaty, przeanalizowane zostały możliwości zbudowania układu sekwencyjnego. Pierwszym pomysłem było stworzenie prawie całkowicie sekwencyjnego układu, czyli wykonanie każdego kroku z (4) w osobnym takcie zegarowym. Na wykonanie mnożenia należałoby wtedy poświęcić tyle taktów ile bitów zawiera mnożnik.

Niestety okazało się, że struktura układu sekwencyjnego rośnie podobnie jak struktura układu kombinacyjnego, czyli 4-krotnie wraz z 2-krotnym wzrostem rozmiaru argumentów. Następnym pomysłem na stworzenie wydajnego urządzenia sekwencyjnego było wykorzystanie modułów kombinacyjnych, a nie jak poprzednio implementacja sekwencyjna równania (4). Przeanalizowano więc sekwencyjne 32-bitowe układy mnożące wykorzystujące moduły 4, 8 i 16-bitowe. Rezultaty zaprezentowane są w tabeli 1.

Tab. 1. Klasyczne układy mnożące – wyniki implementacji
Tab. 1. Schoolbook multipliers – results

Układ mnożący	Opóźnienie [ns] / Maksymalna częstotliwość działania [MHz]	Liczba bloków 4-1 LUTs
4 bitowy	7.675 ns	11
8 bitowy	8.9 ns	43
16 bitowy	9.4 ns	195
32 bitowy (kombinacyjny)	12 ns	818
32 bitowy (sekwencyjny, 1 blok 8-bitowy)	280 MHz	233
32 bitowy (sekwencyjny, 1 blok 16-bitowy)	235 MHz	303
32 bitowy (sekwencyjny, 4 bloki 8-bitowe)	250 MHz	279

3.2. Algorytmy wykorzystujące macierze i wektory

Niech $A,B \in GF(2^m)$, są wielomianami stopnia m-1, a D iloczynem A, B, czyli wielomianem stopnia 2m-2. Wielomiany D i B zapisane są w postaci wektorów o rozmiarach odpowiednio 2m-1 i m, a wielomian A w postaci macierzy. Iloczyn zdefiniowany jest następująco:

$$D = AB = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{m-2} \\ d_{m-1} \\ d_m \\ d_{m+1} \\ \\ d_{2m-3} \\ d_{2m-2} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & a_0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_0 & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 & a_0 \\ 0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-2} \\ b_{m-1} \end{bmatrix} \tag{5}$$

Ten algorytm został zaimplementowany w inny sposób niż poprzednie. Spowodowane było to tym, że wynikiem bezpośredniego przenoszenia w sprzęt (5) były dokładnie takie same rezultaty jak w przypadku czysto kombinacyjnego rozwiązania. Ponadto w jednej chwili musielibyśmy przechowywać całą macierz A, co w przypadku dużych liczb daje ogromną ilość danych.

Aby uniknąć zużywania nadmiernej ilości bloków układu FPGA założono, że jednocześnie przechowywane mogą być tylko dwie kolumny macierzy A, w pierwszej przechowywany jest pośredni rezultat mnożenia, a w drugiej wartości kolejnych kolumn. W ten sposób otrzymane rozwiązanie stało się układem sekwencyjnym. Powstała implementacja mnoży liczby o wielkości od 2 do 600 bitów. Niestety, aby wykonać mnożenie m-bitowych wejść potrzebne jest m taktów. Zaimplementowany moduł zajmował 126 bloków LUT i mógł działać z częstotliwością około 207 MHz. W porównaniu z poprzednim rozwiązaniem, wydaje się to być dużo, jednak dokładnie tyle samo bloków układ zajmie, gdy zechcielibyśmy pomnożyć dwie liczby 600-bitowe. W przypadku układu z poprzedniego rozdziału było by to już 1000 razy więcej bloków.

3.3. Algorytmy typu dziel-i-rządź i sztuczka Karatsuby-Ofmana

Algorytm Karatsuby-Ofmana [6] należy do grupy tak zwanych algorytmów dziel-i-rządź. Podstawą tego typu algorytmów jest sprowadzenie dużego problemu do serii mniejszych, łatwiejszych problemów. W przypadku mnożenia wielomianów dzieli się wektory wejściowe na wielomiany mniejszego stopnia. W wersji podstawowej zakłada się następujący sposób dzielenia wielomianów stopnia m dla m=2t (m parzyste):

m-1	*m/2*	*m/2-1*	0
A_H		$AL.$	

$$a(x) = x^{\frac{m}{2}} A_H + A_L = x^{\frac{m}{2}} (x^{\frac{m}{2}-1} a_{m-1} + \cdots + a_{\frac{m}{2}}) + (x^{\frac{m}{2}-1} a_{\frac{m}{2}-1} + \cdots + a_0) \quad (6)$$

m-1	*m/2*	*m/2-1*	0
B_H		B_L	

$$b(x) = x^{\frac{m}{2}} B_H + B_L = x^{\frac{m}{2}} (x^{\frac{m}{2}-1} b_{m-1} + \cdots + b_{\frac{m}{2}}) + (x^{\frac{m}{2}-1} b_{\frac{m}{2}-1} + \cdots + b_0) \quad (7)$$

Rozwijając wzór (4) według powyższego schematu otrzymujemy:

$$\begin{aligned} d(x) &= a(x)b(x) = (x^{\frac{m}{2}} A_H + A_L)(x^{\frac{m}{2}} B_H + B_L), \quad (8) \\ &= x^m A_H B_H + x^{\frac{m}{2}} (A_H B_L + A_L B_H) + A_L B_L \end{aligned}$$

Analizując otrzymane równanie (8) możemy zauważyć, że mnożenie dwóch liczb 4-bitowych wymaga wykonania 4 mnożeń liczb m/2 -bitowych. Sztuczka Karatsuby-Ofmana [4]:

$$\begin{aligned} d(x) &= a(x)b(x) = (x^{\frac{m}{2}} A_H + A_L)(x^{\frac{m}{2}} B_H + B_L) \quad , (9) \\ &= x^m A_H B_H + A_L B_L + x^{\frac{m}{2}} ((A_H + A_L)(B_H + B_L) - A_H B_H - A_L B_L). \end{aligned}$$

w pewien sposób upraszcza równanie (8) i pokazuje, że zamiast przeprowadzać 4 mnożenia możemy przeprowadzić 3, kosztem dodatkowych operacji XOR. Podsumowując, w równaniu (8), aby uzyskać iloczyn *A, B* należy wykonać 4 mnożenia i 3 operacje XOR, natomiast w przypadku równania (9) wykonujemy 3 mnożenia i 6 operacji XOR. Przy założeniu, że operacja XOR jest mniej kosztowna ze względu na czas i zasoby niż mnożenie, można stwierdzić, że sztuczka Karatsuby-Ofmana pozwala na tworzenie szybszych i mniejszych urządzeń.

Zadecydowano jednocześnie testować zachowanie podstawowego algorytmu „dziel-i-rządź" oraz algorytmu z wykorzystaniem sztuczki Karatsuby-Ofmana. Pierwsze wyniki implementacji są nieco zaskakujące dla kogoś, kto spodziewał się zobaczyć przewagę sztuczki Karatsuby-Ofmana nad podstawową wersją algorytmu. Bezpośrednia implementacja równania (9) nie jest bardziej wydajniejsza od implementacji równania (8). Wręcz przeciwnie dla krótkich wektorów wejściowych (2, 4-bity) implementacja równania (9) zajmowała więcej zasobów sprzętowych i była znacznie wolniejsza. Korzyści z użycia sztuczki Karatsuby-Ofmana były zauważalne dopiero przy 16-bitowych i większych wartościach wejściowych. Analiza otrzymanych rozwiązań zaprezentowana jest poniżej. Jak w poprzednich przypadkach zaczęto od analizy rozwiązań dla krótkich wektorów wejściowych, a potem obserwowano jak korzystnie łączyć mniejsze jednostki w celu stworzenia dużych układów mnożących. Do mnożenia elementów użyto klasycznej metody opisanej w podrozdziale drugim.

Najpierw zbadano 8-bitowy układ mnożący zbudowany z 4-bitowych jednostek mnożących. Następnie przeanalizowano 16-bitowe i większe układy mnożące stworzone z jednostek 4 i 8-bitowych. Następnym przeprowadzonym testem było zbudowanie układu bazując na 8-bitowej jednostce mnożącej stworzonej z kombinacji 4-bitowych modułów kombinacyjnych. Początkowo użyto 4-bitowych modułów, które aby uzyskać iloczyn dwóch liczb 8-bitowych korzystają ze sztuczki Karatsuby-Ofmana. Następnie użyto tych samych modułów, ale wykorzystano rozwinięcie równania (8). Po uzyskaniu tych dwóch rodzajów jednostek 8-bitowych użyto ich do budowy 16-bitowego operatora mnożenia. Wyniki implementacji przedstawiono w tabeli 2.

Tab. 2. Implementacja algorytmów typu dziel-i-rządź - wyniki
Tab. 2. Divide and conquer multiplication – implementation results

Układ mnożący (podstawowa jednostka)	Typ	Opóźnienie [ns]	Liczba bloków 4-1 LUT
8-bitowy			
4-bitowy kombinacyjny	dziel-rządź	9 ns	45
	Karatsuba-Ofman	9.3 ns	46
16-bitowy			
4-bitowy kombinacyjny	dziel-rządź	12.6 ns	158
8-bitowy kombinacyjny	dziel-rządź	10.3 ns	172
	Karatsuba-Ofman	11.7 ns	159
8-bitowy Karatsuba-Ofman	Karatsuba-Ofman	12.7 ns	170
32-bitowy	Karatsuba-Ofman		
16-bitowy, Karatsuba-Ofman, zbudowana z 8-bitowych bloków kombinacyjnych		13.3 ns	540
64 –bitowy	Karatsuba-Ofman		
32-bitowa, Karatsuba-Ofman zbudowana z 16-bitowych bloków Karatsuba-Ofman , zbudowanych z 8-bitowych bloków kombinacyjnych		16 ns	1753

Dalsza analiza miała na celu zbadanie jak rozmiar kombinacyjnego układu korzystającego ze sztuczki Karatsuby-Ofmana zależy od rozmiaru wektorów wejściowych. Poprzez analizę 32, 64 i 128 bitowych jednostek zauważono, że gdy podwajamy rozmiar wektora wejściowego rozmiar układu rośnie prawie 4-krotnie.

4. Podsumowanie i uwagi

Wszystkie prezentowane algorytmy zostały przeanalizowane pod kątem ich skalowalności, wydajności i elastyczności. Sprawdzono, które najlepiej nadają się do tworzenia dużych układów mnożących, o rozmiarze m na przykład równym 169, 233 lub 571 [7]. Jak można zauważyć najlepszym rozwiązaniem ze względu na zajętość zasobów jest układ mnożący oparty na algorytmach wykorzystujących macierze i wektory, jest to jednak najwolniejsze rozwiązanie. Wydaje się, że aby efektywnie wykorzystać sztuczkę Karatsuby-Ofmana należy używać jednostek podstawowych o regularnych strukturach, zajmujących niewiele zasobów. Im bardziej złożony jest moduł bazowy tym większą i bardziej skomplikowaną tworzy strukturę końcową. Rozsądnie należy również dobierać rozmiary układów bazowych. Z przedstawionej analizy można wywnioskować, że jednostki te nie powinny być ani zbyt małe ani zbyt duże, odpowiednim rozmiarem wydaje się być 8-bit.

Podsumowując, aby stworzyć wydajny układ mnożący duże liczby należy znaleźć kompromis pomiędzy szybkością działania, a rozmiarem urządzenia. Nie ma za wiele sensu tworzenie ani czysto kombinacyjnych ogromnych rozwiązań ani rozwiązań sekwencyjnych o niskiej przepustowości.

5. Literatura

[1] Lidl, R., Niederreiter, H.: Introduction to finite fields and their applications, Cambridge University Press. 1994.
[2] Hankerson, D., Menzes, A., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer-Verlag, NY, 2004.
[3] Erdem, S.S., Yanik, T., Koc, C.K.: Polynomial Basis Multiplication over GF(2m). Acta Applicandae Mathematicae. Vol. 93, 2006.
[4] Karatsuba, A., Ofman, Y.: Multiplication of Multidigit Numbers on Automata. Soviet Phys. Doklady, vol. 7, strony 595-596, 1963.
[5] Deschamps, J-P., Imana, J.L., Sutter G.D.: Hardware Implementation of Finite-Field Arithmetic. McGraw-Hill, 2009.
[6] Bernstein, D.J.: Multidigit Multiplication for Mathematicians, 2001.
[7] NIST. Digital Signature Standard. Draft – FIPS publication 186-3. National Institute of Standards and Technology , United States, 2008.

otrzymano / received: 15.10.2010
przyjęto do druku / accepted: 01.12.2010 *artykuł recenzowany*