Efficient Finite Field and Elliptic Curve Arithmetic

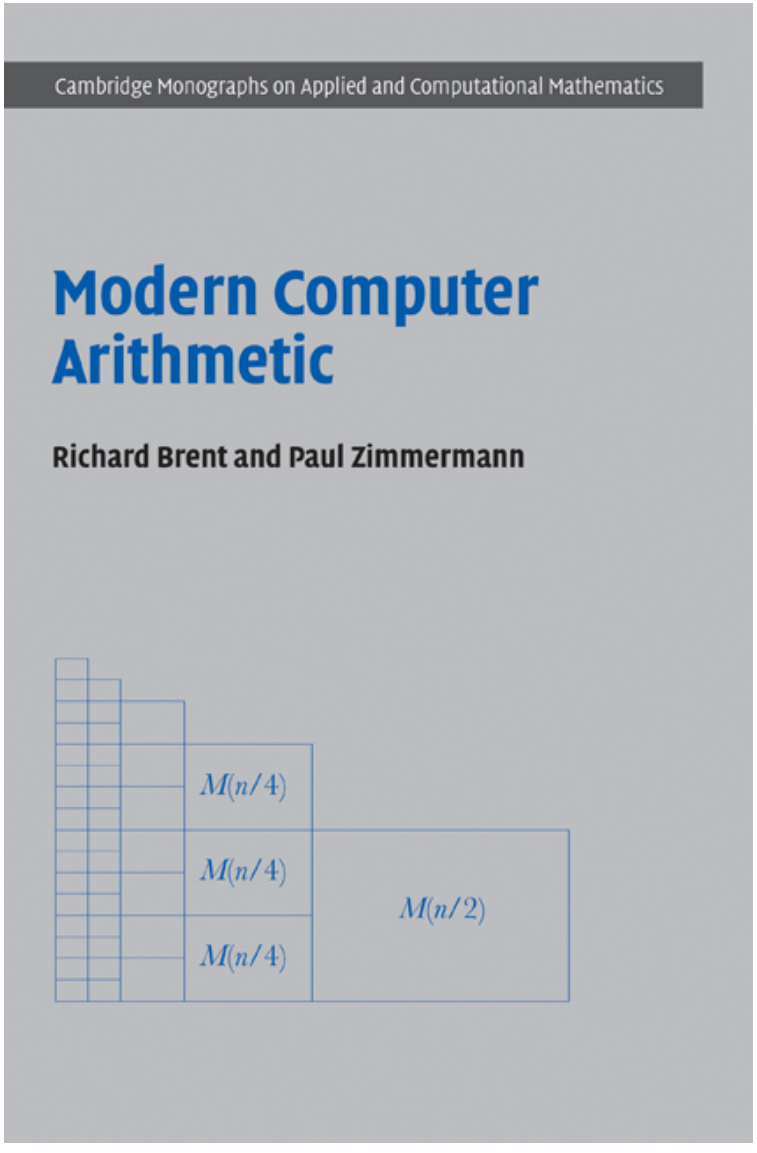
Laurent Imbert

CNRS, LIRMM, Université Montpellier 2

Summer School ECC 2011 – Nancy, September 12-16, 2011

Part 1

Modular and finite field arithmetic


Cambridge Monographs on Applied and Computational Mathematics
Modern Computer Arithmetic
Richard Brent and Paul Zimmermann
M(n/4)
M(n/4)
M(n/4)
M(n/2)

Finite fields

- The order of a finite field is always a prime or a prime power
- If $q = p^k$ is a prime power, there exists a unique finite field of order q, denoted $\mathbb{F}_{p^k}$ or $GF(p^k)$
- p is called the field characteristic and $\mathbb{F}_p \subset \mathbb{F}_{p^k}$
- If $k = 1$ the *prime field* $\mathbb{F}_p$ is the field of residue classes modulo p

$$\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$$

- If $k > 1$: degree-k extension of $\mathbb{F}_p$

$$\mathbb{F}_{p^k} = \mathbb{F}_p[X]/(f(X))$$

  where $f(X) \in \mathbb{F}_p[X]$ is an irreducible polynomial of degree k.
- Finite fields $GF(2^k)$ are often called *binary fields*

Efficient modular and finite field arithmetic

Outline

- How do we represent the elements and how do we compute the basic arithmetic operations $\pm, \times, \div$ efficiently in $\mathbb{Z}/p\mathbb{Z}$? (p prime or not)
- What are the best know algorithms for arbitrary primes p?
- How do we represent the elements and how do we compute efficiently in $\mathbb{F}_{p^k}$? (special attention to the case $p = 2$)
- Are there any special finite fields for which these operations can be made even faster?

Multiple precision arithmetic

- Single precision: 32 or 64 bits on current processors ; 8 or 16 bits on constrained devices or smart cards

- Large integers: base β expansion, array of word-size "integers"

$$A = a_{n-1}\beta^{n-1} + \cdots + a_1\beta + a_0, \qquad \text{with } 0 \leq a_i \leq \beta - 1$$

  size $n = O(\log A)$

- Polynomials: array of coefficients: $A(X) = \sum_{i=0}^{d-1} a_i X^i$

  size $n = O(d)$

Complexity of arithmetic operations

Basic arithmetic operations

- Addition, subtraction: $O(n)$
- Multiplication: $M(n)$
- Division: $O(M(n))$

Complexity of arithmetic operations

Basic arithmetic operations

- Addition, subtraction: $O(n)$
- Multiplication: $M(n)$
- Division: $O(M(n))$

Fast multiplication algorithms

- Scholar multiplication: $M(n) = O(n^2)$
- Karatsuba multiplication: $M(n) = O(n^{\log_2 3})$
- Toom-Cook r-way multiplication: $M(n) = O(n^{\log_r (2r-1)})$
- FFT-based multiplication: $M(n) = O(n \log n \log \log n)$

Scholar multiplication

Algorithm 1 BasecaseMultiply

Input: $A = (a_{m-1}, \ldots, a_0)_\beta$, $B = (b_{n-1}, \ldots, b_0)_\beta$
Output: $C = AB = (c_{m+n-1}, \ldots, c_0)_\beta$

1: $C \leftarrow A \times b_0$
2: For $i = 1, \ldots, n-1$ do
3: $\quad C \leftarrow C + (A \times b_i)\beta^i$
4: Return C

Quadratic complexity: $O(mn)$ word operations

Squaring: $\approx n^2/2$ word operations

line 3 uses the processor's MAC (Multiply Accumulate) instruction
Best if A is the larger operand

Karatsuba Multiplication

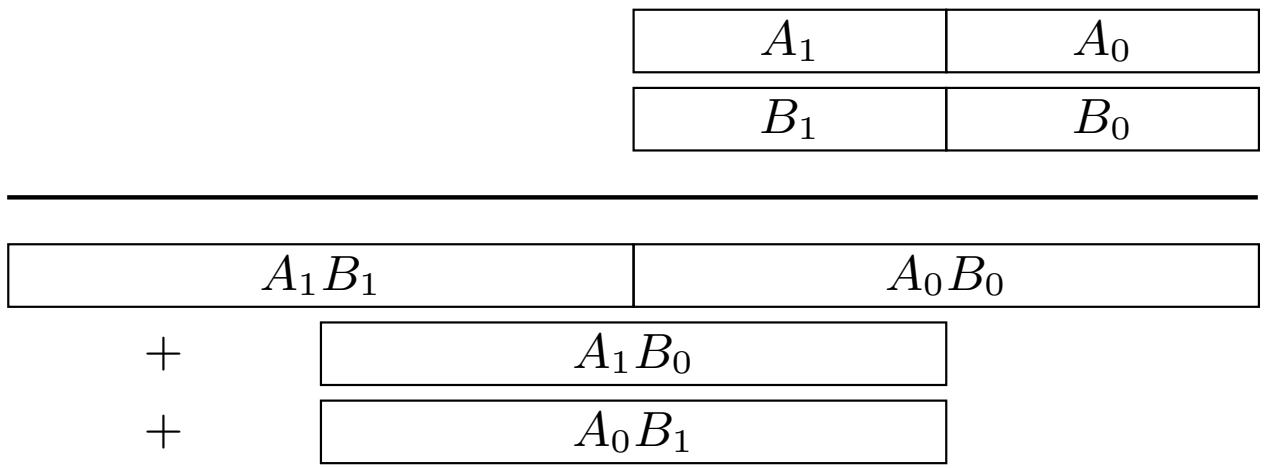
Let $A = A_1\beta^{n/2} + A_0, \qquad B = B_1\beta^{n/2} + B_0$

A_1	A_0
B_1	B_0

Karatsuba Multiplication

Let $A = A_1\beta^{n/2} + A_0, \qquad B = B_1\beta^{n/2} + B_0$

$$AB = A_1B_1\beta^n + \beta^{n/2}(A_1B_0 + A_0B_1) + A_0B_0$$

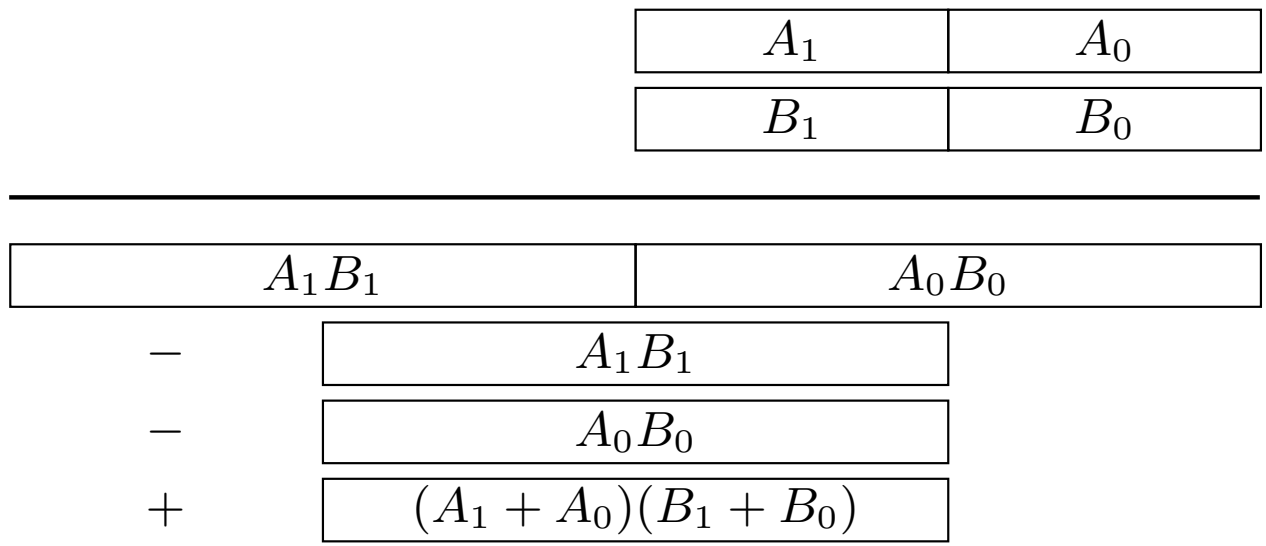


Karatsuba Multiplication

Let $A = A_1\beta^{n/2} + A_0, \qquad B = B_1\beta^{n/2} + B_0$

$$AB = A_1B_1\beta^n + \beta^{n/2}(A_1B_0 + A_0B_1) + A_0B_0$$

$$= A_1B_1\beta^n + \beta^{n/2}((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0) + A_0B_0$$



Complexity of Karatsuba multiplication

Multiplying two operands of size n requires 3 multiplications of size $n/2$ (at the cost of a few extra additions)

$$K(n) = 3K(n/2) + O(n)$$

Applying the above algorithm recursively leads to subquadratic complexity:

$$K(n) = O(n^{\alpha}), \quad \text{with } \alpha = \log_2 3 \approx 1.585$$

Stop recursion and use BaseCaseMultiply when the operands get small enough. How small? Depends on the architecture.

Exercise: implement a subtractive variant of Karatsuba.
Hint: replace $(A_0 + A_1)(B_0 + B_1)$ by $(|A_0 - A_1|)(|B_0 - B_1|)$.

Generalization of Karatsuba multiplication

View A, B as polynomials $A_1 x + A_0$, $B_1 x + B_0$ evaluated at $x = \beta^{n/2}$

- Evaluation at $0, 1, \infty$: ($0, -1, \infty$ for the subtractive version)

$$\begin{aligned} A_0 &= A(0) & B_0 &= B(0) \\ A_0 + A_1 &= A(1) & B_0 + B_1 &= B(1) \\ A_1 &= A(\infty) & B_1 &= B(\infty) \end{aligned}$$

- Multiplication:

$$C(0) = A(0)B(0) \quad C(1) = A(1)B(1) \quad C(\infty) = A(\infty)B(\infty)$$

- Interpolation:

$$C = C(0) + (C(1) - C(0) - C(\infty))\, x + C(\infty)x^2$$

Toom-Cook r-way multiplication

Follows the same evaluation/interpolation scheme

- View A, B as $A_0 + \cdots + A_{r-1}x^{r-1}$ and $B_0 + \cdots + B_{r-1}x^{r-1}$ evaluated at $x = \beta^{\lceil n/r \rceil}$. The product AB is of degree $2r-2$
- Evaluate $A(x)$ and $B(x)$ at $2r-1$ distinct points
- Interpolate and compute $C(\beta^{\lceil n/r \rceil})$

Complexity: $M(n) = O(n^{\log_r(2r-1)})$

The name "Toom-Cook algorithm" is often used for Toom-Cook 3-way.

The choice of interpolation points is important for fast multi-evaluation and interpolation

FFT Multiplication

The Fast Fourier Transform (FFT) can be used to speed-up the evaluation and interpolation steps

One needs to considers special interpolation points (roots of unity) and special values of r for those points to exist

Schönage-Strassen's algorithm: $M(n) = O(n \log n \log \log n)$

The FFT multiplication is faster than the other subquadratic algorithms for very large operands

GMP Multiplication thresholds

```
Parameters for ./mpn/x86_64/core2/gmp-mparam.h
Using: CPU cycle counter, supplemented by microsecond getrusage()
speed_precision 10000, speed_unittime 4.17e-10 secs, CPU freq 2400.00 MHz
DEFAULT_MAX_SIZE 1000, fft_max_size 50000

/* Generated by tuneup.c, 2011-08-31, gcc 4.2 */

[...]
#define MUL_TOOM22_THRESHOLD                  24
#define MUL_TOOM33_THRESHOLD                  65
#define MUL_TOOM44_THRESHOLD                 112

[...]
#define MUL_TOOM32_TO_TOOM43_THRESHOLD        69
#define MUL_TOOM32_TO_TOOM53_THRESHOLD       122

[...]
#define MUL_FFT_THRESHOLD                   5760
```

Modular arithmetic

Given $0 < P < \beta^n$, how do we compute efficiently modulo P?

Let $C \in \mathbb{Z}$. Then $C = PQ + R$, with $R = (C \bmod P) < P$ (Euclid)

Naive solution: compute the quotient Q by dividing C by P

$$R = C - \lfloor C/P \rfloor P$$

Goal: compute $R = C \bmod P$ without division

Barrett algorithm

Let $0 < P < \beta^n$ and $0 < C < P^2$

(C may be the result of a multiplication of $A < P$ and $B < P$)

1. Compute an approximation of the quotient $\lfloor C/P \rfloor$ as

$$Q = \left\lfloor \left\lfloor \frac{C}{\beta^n} \right\rfloor \nu/\beta^n \right\rfloor$$

   where $\nu = \left\lfloor \beta^{2n}/P \right\rfloor$ is precomputed

2. Compute $R = C - QP$

Complexity: $2M(n)$ (assuming divisions by β are free)

Exercise: R may not be fully reduced. How many subtractions may be needed to get $R < P$?

Montgomery algorithm

Let $0 < P < \beta^n$ and $0 < C < P^2$

(C may be the result of a multiplication of $A < P$ and $B < P$)

1. Compute the smallest integer Q s.t. $C + QP$ is a multiple of β^n

$$Q = \mu C \bmod \beta^n,$$

   where $\mu = -1/P \bmod \beta^n$ requirement: $(P, \beta) = 1$

2. Compute $R = (C + QP)/\beta^n$ exact division

Complexity: $2M(n)$

The result $R < 2P$ is congruent to $C\beta^{-n} \bmod P$

Montgomery representation

Let $0 < P < \beta^n$ and $A, B < P$

Suppose $MontgomeryMul(A, B, P)$ returns $AB\beta^{-n} \bmod P$

Change of representation:

$$A \longrightarrow A' = A\beta^n \bmod P$$
$$B \longrightarrow B' = B\beta^n \bmod P$$

$MontgomeryMul(A', B', P) = A'B'\beta^{-n} = AB\beta^n \bmod P$

Montgomery representation is stable for $MontgomeryMul$. Can be used for modular exponentiation

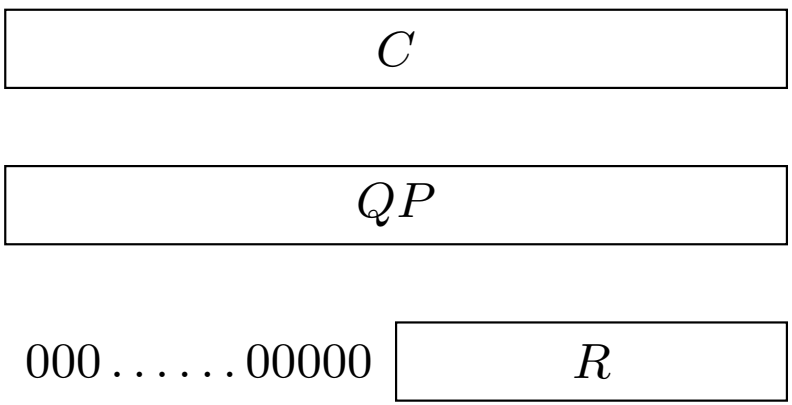
$MontgomeryMul(A^e\beta^n, 1, P) = A^e \bmod P$

Barrett vs Montgomery

Barrett (MSB algorithm)

Precomputation: $\lfloor \beta^{2n}/P \rfloor$

Complexity: $2M(n)$

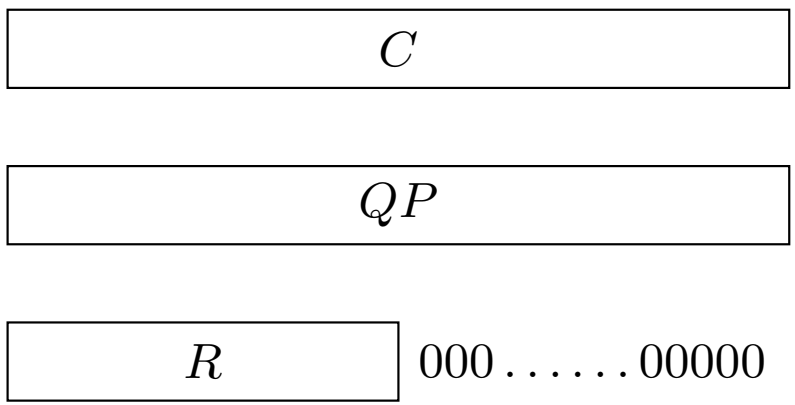


$$R = C - QP$$

Montgomery (LSB algorithm)

Precomputation: $-1/P \bmod \beta^n$

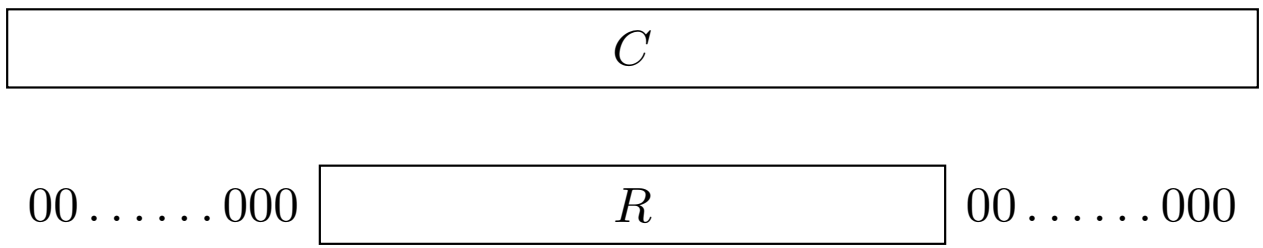
Complexity: $2M(n)$



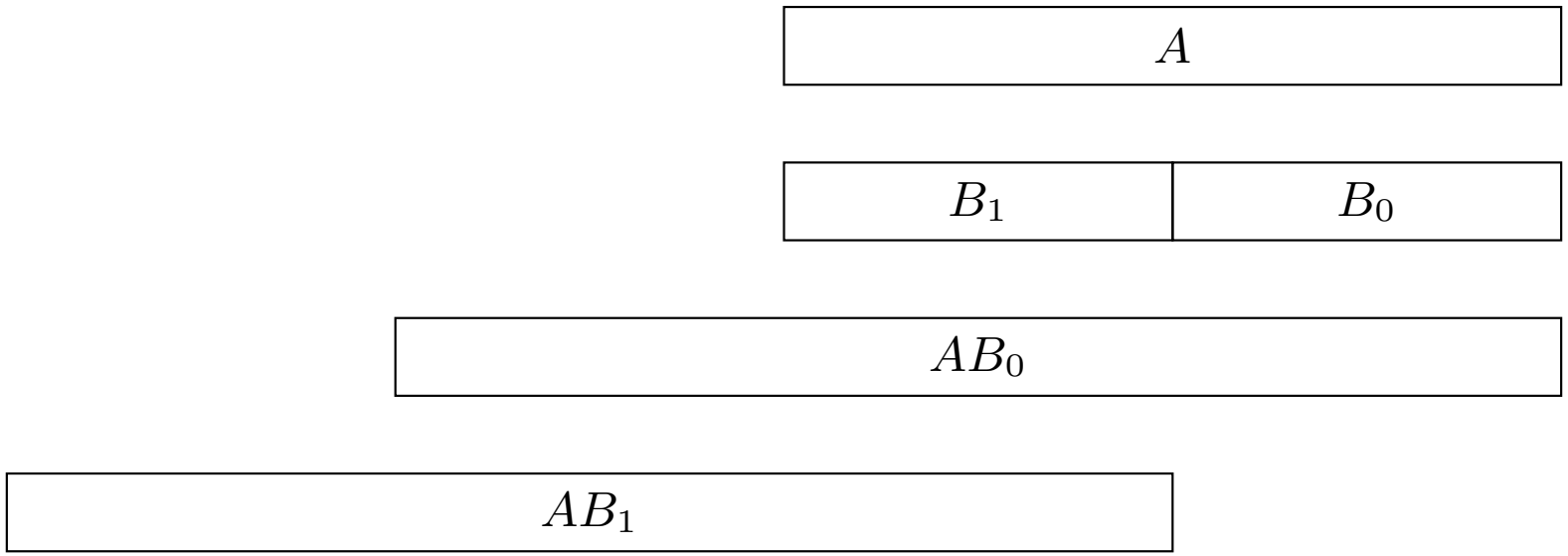
$$R = (C + QP)\beta^{-n}$$

Bipartite reduction [Kaihara, Takagi]

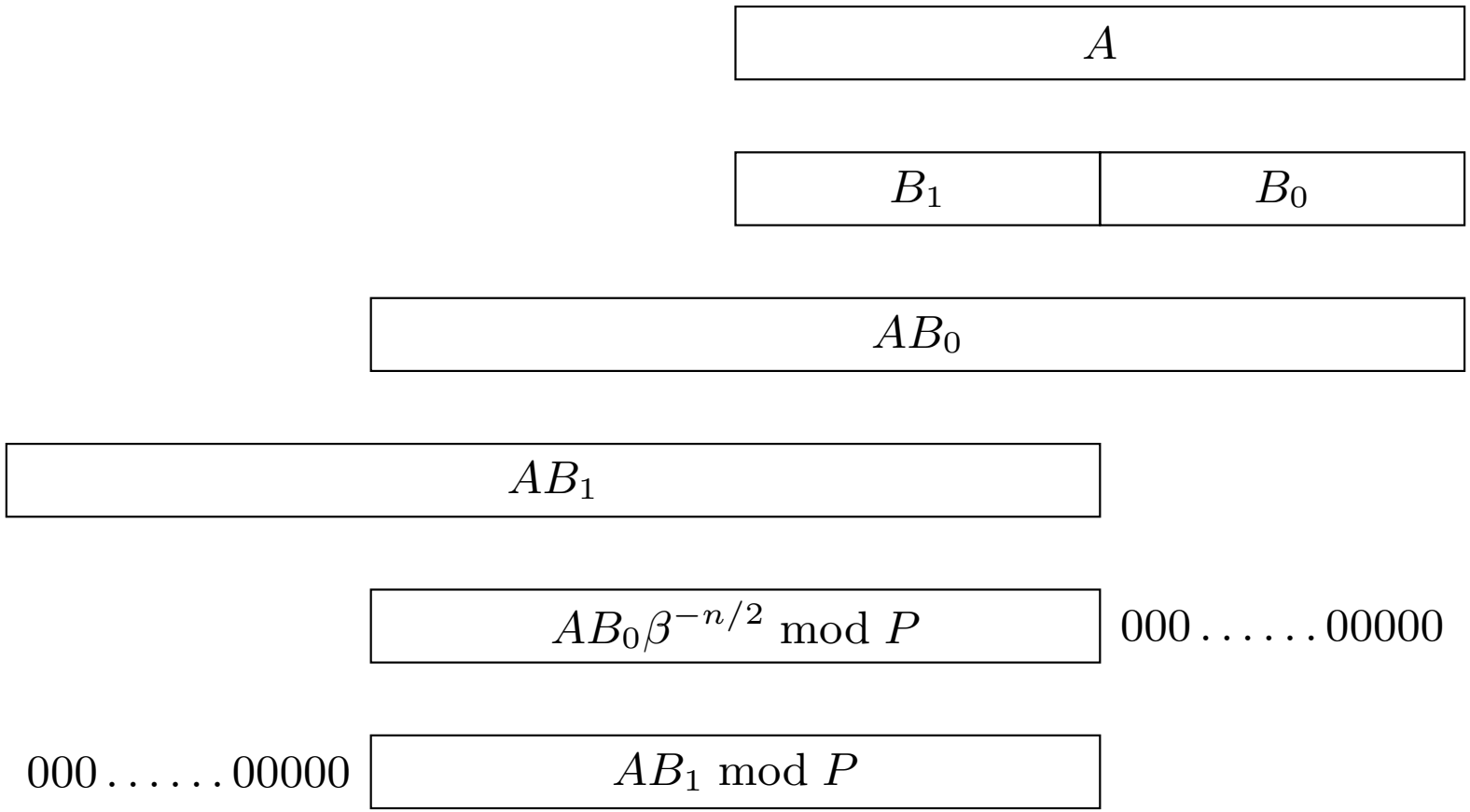
Idea: reduce the $n/2$ MSB using a classical division or a (partial) Barrett reduction and the $n/2$ LSB using a (partial) Montgomery reduction



Bipartite multiplication



Bipartite multiplication



$$AB\beta^{-n/2} \bmod P = (AB_1 \bmod P + AB_0\beta^{-n/2} \bmod P) \bmod P$$

Complexity of the bipartite multiplication

- Partial products AB_0 and AB_1

  $2M(n, n/2)$

- $AB_1 \bmod P$: partial Barrett reduction $(3n/2 \to n)$

  $M(n/2) + M(n, n/2)$

- $AB_0\beta^{-n/2} \bmod P$: partial Montgomery reduction $(3n/2 \to n)$

  $M(n/2) + M(n, n/2)$

Total cost: $2M(n/2) + 4M(n, n/2)$

Parallel cost: $M(n/2) + 2M(n, n/2) \approx 5M(n/2)$

Fast arithmetic modulo special primes

- Ideal choice: $P = \beta^n \pm 1$

  Let $C = C_1\beta^n + C_0$. Then $R = (C \bmod P) = C_0 \pm C_1 \bmod P$

- Pseudo Mersenne: $P = \beta^n \pm a$, with a "small"

  Let $C = C_1\beta^n + C_0$. Then $R = (C \bmod P) = C_0 \pm aC_1 \bmod P$

  Example: "old" speed record for ECDH using an elliptic curve defined over $\mathbb{F}_{2^{255}-19}$ [D. J. Bernstein]

- Generalized Mersenne [Solinas 99]: $P = f(2^n)$ where $f \in \mathbb{F}_2[X]$

  Example: NIST, SECG primes

NIST primes

The five NIST-recommended randomly chosen elliptic curves over prime fields $\mathbb{F}_p$ are defined modulo generalized Mersenne primes.

Primes	$f(t)$	t
$p_{192} = 2^{192} - 2^{64} - 1$	$t^3 - t - 1$	2^{64}
$p_{224} = 2^{224} - 2^{96} + 1$	$t^7 - t^3 + 1$	2^{32}
$p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	$t^8 - t^7 + t^6 + t^3 - 1$	2^{32}
$p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$	$t^{12} - t^4 - t^3 + t + 1$	2^{32}
$p_{521} = 2^{521} - 1$	$t - 1$	2^{521}

The SECG also recommends a set of special primes with similar properties.

Example: reduction modulo $P = p_{192}$

$C < P^2$ can be expressed as a polynomial of degree ≤ 5

$$C = C_5 t^5 + \cdots + C_1 t + C_0$$

Since $P = t^3 - t - 1$, we have

$$\begin{aligned} t^3 &\equiv t + 1 \pmod{P} \\ t^4 &\equiv t^2 + t \pmod{P} \\ t^5 &\equiv t^2 + t + 1 \pmod{P} \end{aligned}$$

The digits of $R = C \bmod P$ are then obtained by computing

$$\begin{aligned} R_0 &= C_0 + C_3 + C_5 \bmod P \\ R_1 &= C_1 + C_3 + C_4 + C_5 \bmod P \\ R_2 &= C_2 + C_4 + C_5 \bmod P \end{aligned}$$

Modular division and inversion

Division: $A/B \bmod N = A \times (1/B) \bmod N$

Inversion: $1/B \bmod N$ can be computed using the extended Euclidean algorithm (EEA)

$1/B$ exists iff $(B, N) = 1$. The EEA computes the *Bezout coefficients* (U, V) such that

$$1 = UB + VN$$

Hence

$$U = 1/B \bmod N$$

Greatest common divisor

Euclid: Let $R_0 = A$, $R_1 = B$. Compute the remainder sequence $R_{i+1} = R_{i-1} \bmod R_i$ until $R_{i+1} = 0$. Return R_i

Example: $A = 540738$, $B = 207717$, 125304, 82413, 42891, 39522, 3369, 2463, 906, 651, 255, 141, 114, 27, 6, 3, 0

The algorithm terminates: The sequence $(R_i)_{i \geq 0}$ is strictly decreasing

Complexity: $O(n^2)$ if $A, B < \beta^n$, or $O(\deg A \deg B)$ for polynomials

For $A, B \in \mathbb{Z}$, the worst case occurs when the quotients in $R_{i+1} = R_{i-1} - Q_i R_i$ are all ones, i.e. for consecutive Fibonacci numbers

Extended GCD

The extended GCD algorithm expresses the greatest common divisor as a linear combination of A and B

$$UA + VB = \gcd(A, B)$$

The idea consists of computing the sequences $(U_i)_{i\geq 0}$ and $(V_i)_{i\geq 0}$ during the computation of the remainder sequence $(R_i)_{i\geq 0}$.

$$R_0 = A = 1 \cdot A + 0 \cdot B$$

$$R_1 = B = 0 \cdot A + 1 \cdot B$$

$$\begin{aligned} R_{i+1} = R_{i-1} - Q_i R_i &= (U_{i-1}A + V_{i-1}B) - Q_i(U_i A + V_i B) \\ &= (U_{i-1} - Q_i U_i)A + (V_{i-1} - Q_i V_i)B \end{aligned}$$

Fast GCD computation for large integers

In practice, the quotients Q_i are small. The first terms of the quotient sequence $(Q_i)_{i\geq 0}$ can be correctly computed using the most significant digits of A and B only

Example: for $A = 540738$ and $B = 207717$, the Q_i's are given by the continued fraction expansion of A/B.

$$\begin{aligned} 540738/207717 &= [2, 1, 1, 1, 1, 11, 1, 2, 1, 2, 1, 1, 4, 4, 2] \\ 54/20 &= [2, 1, 2, 3] \\ 5407/2077 &= [2, 1, 1, 1, 1, 11, 1, 1, 1, 1, 1, 1, 2] \end{aligned}$$

Replace costly multiple precision division and multiplications by single precision operations

Lehmer's double-digit gcd: compute the remainder sequence with the most two significant words of the inputs until the smallest remainder fits in one word

Binary GCD

The binary gcd is based on the following observations:

- If A and B are both even, then $\gcd(A, B) = 2\gcd(A/2, B/2)$
- If A is even and B is not, then $\gcd(A, B) = \gcd(A/2, B)$
- If A and B are both odd, $\gcd(A, B) = \gcd(|A - B|, \min(A, B))$

The binary GCD has complexity $O(n^2)$ but is much faster than Euclid's algorithm for small operands (when divisions by 2 reduce to shifts on words)

Extended GCD using a Half GCD

In order to get a subquadratic algorithm, computing all the terms of the remainder sequence should be avoided

Given inputs of size n, the HalfGcd algorithm computes two consecutive terms in the remainder sequence of size $\approx n/2$.

$$\begin{pmatrix} R_j \\ R_{j+1} \end{pmatrix} = M_j \cdot \begin{pmatrix} R_0 \\ R_1 \end{pmatrix}, \quad \text{where} \quad M_j = \prod_{i=1}^{j} \begin{pmatrix} 0 & 1 \\ 1 & -Q_i \end{pmatrix}$$

HalfGcd reduces inputs from size n to size $\approx n/2$. Recursive calls further reduces to size $\approx n/4$, $\approx n/8$, ..., until the smallest remainder is 0. The extended gcd is then obtained by combining all the previously computed matrices

Complexity: $O(M(n) \log n)$

HalfGcd

$$a = a_1\beta^{n/2} + a_0, \quad b = b_1\beta^{n/2} + b_0$$

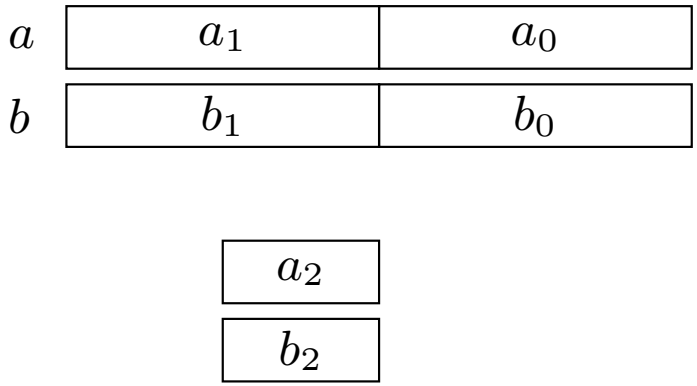
a	a_1	a_0
b	b_1	b_0

HalfGcd

$a = a_1\beta^{n/2} + a_0, \quad b = b_1\beta^{n/2} + b_0$

Compute consecutive remainders a_2, b_2
of size $\approx n/4$ *from* a_1, b_1

$M, a_2, b_2 = \mathsf{HalfGcd}(a_1, b_1)$



HalfGcd

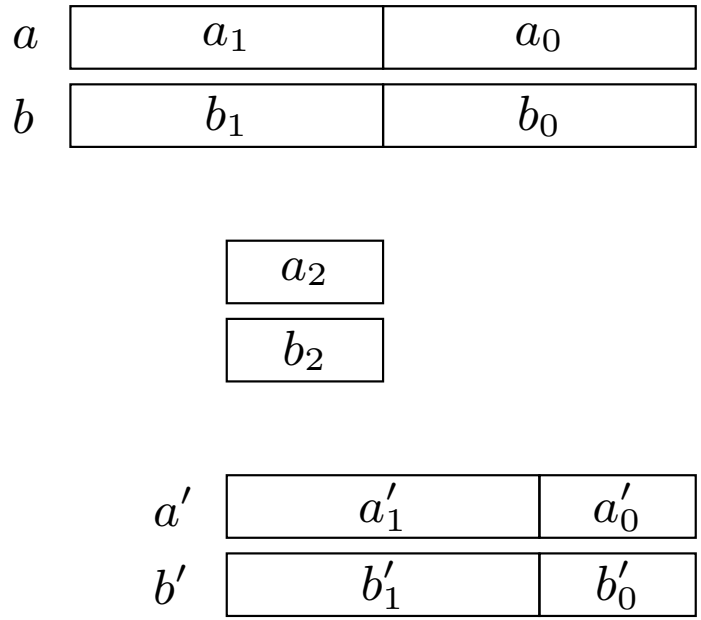
$a = a_1\beta^{n/2} + a_0, \quad b = b_1\beta^{n/2} + b_0$

Compute consecutive remainders a_2, b_2
of size $\approx n/4$ *from* a_1, b_1

$M, a_2, b_2 = \mathsf{HalfGcd}(a_1, b_1)$

Compute consecutive remainders a', b'
of size $\approx 3n/4$

$(a', b')^t = (a_2, b_2)^t \cdot 2^{n/2} + M \cdot (a_0, b_0)^t$



HalfGcd

$a = a_1\beta^{n/2} + a_0, \quad b = b_1\beta^{n/2} + b_0$

Compute consecutive remainders a_2, b_2
of size $\approx n/4$ *from* a_1, b_1

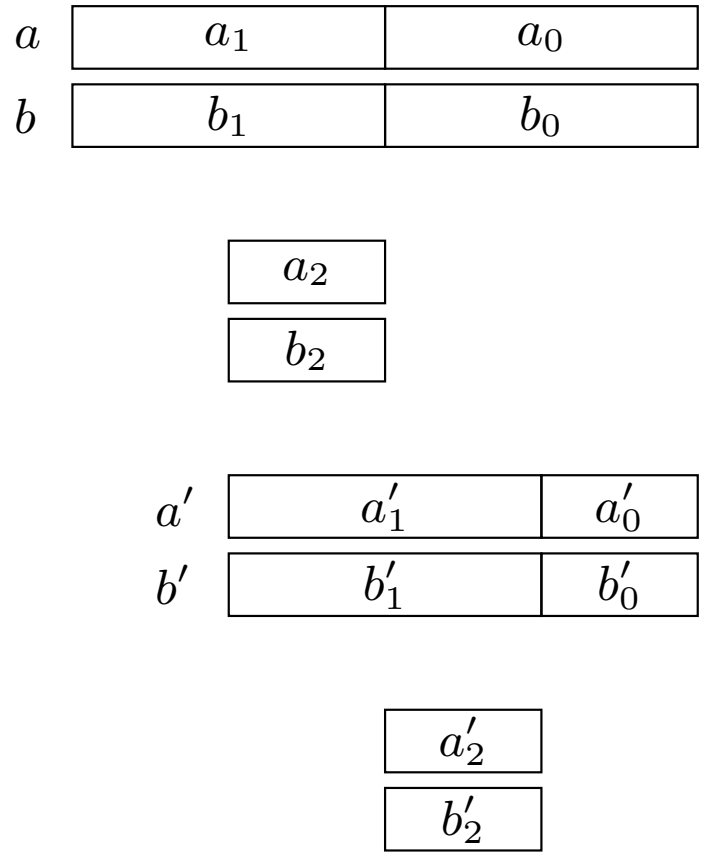
$M, a_2, b_2 = \mathsf{HalfGcd}(a_1, b_1)$

Compute consecutive remainders a', b'
of size $\approx 3n/4$

$(a', b')^t = (a_2, b_2)^t \cdot 2^{n/2} + M \cdot (a_0, b_0)^t$

Compute consecutive remainders a'_2, b'_2
of size $\approx n/4$ *from* a'_1, b'_1

$M', a'_2, b'_2 = \mathsf{HalfGcd}(a'_1, b'_1)$



HalfGcd

$a = a_1\beta^{n/2} + a_0, \quad b = b_1\beta^{n/2} + b_0$

Compute consecutive remainders a_2, b_2
of size $\approx n/4$ *from* a_1, b_1

$M, a_2, b_2 = \mathsf{HalfGcd}(a_1, b_1)$

Compute consecutive remainders a', b'
of size $\approx 3n/4$

$(a', b')^t = (a_2, b_2)^t \cdot 2^{n/2} + M \cdot (a_0, b_0)^t$

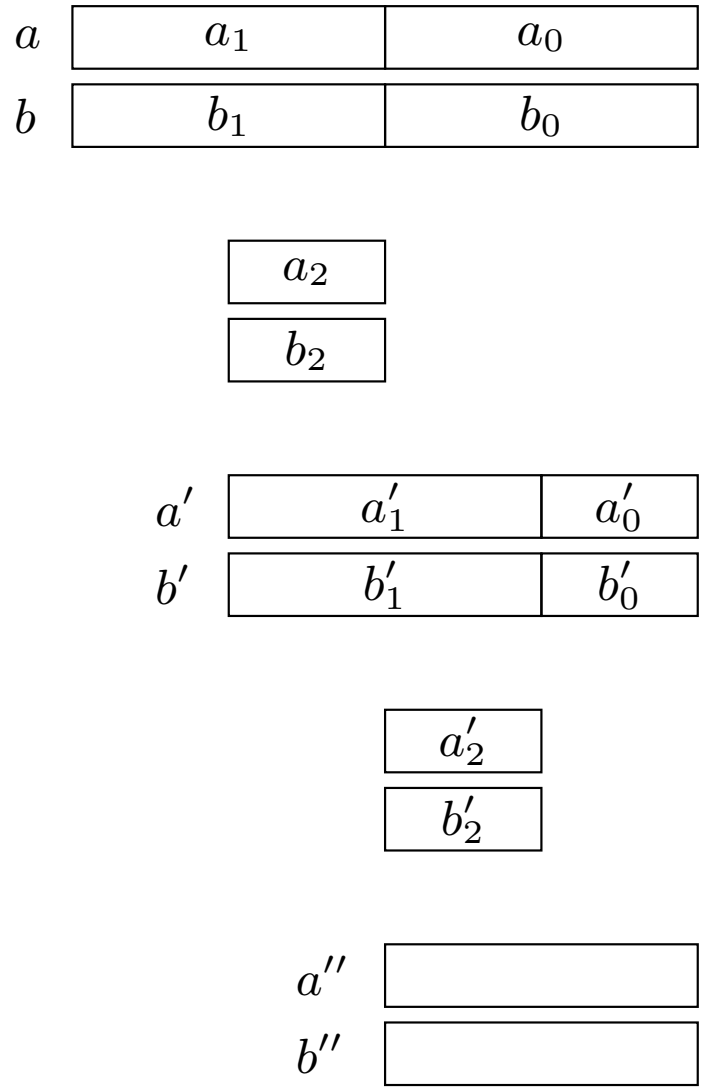
Compute consecutive remainders a'_2, b'_2
of size $\approx n/4$ *from* a'_1, b'_1

$M', a'_2, b'_2 = \mathsf{HalfGcd}(a'_1, b'_1)$

Compute a'', b'' *of size* $\approx n/2$

$(a'', b'')^t = (a'_2, b'_2)^t \cdot 2^{n/4} + M' \cdot (a'_0, b'_0)^t$

Return $M \cdot M', a'', b''$



Binary field arithmetic

Let $f(X)$ be an irreducible binary polynomial of degree m with coefficients in $\mathbb{F}_2$: $f(X) = X^m + g(X)$ with $\deg g < m$.

$$\mathbb{F}_{2^m} = \mathbb{F}_2[X]/(f(X))$$

Polynomials with coefficients in $\mathbb{F}_2$ of degree at most $m-1$

$$a(X) = a_{m-1}X^{m-1} + \cdots + a_1 X + a_0$$

$$a = (a_{m-1}, \ldots, a_1, a_0)$$

Can be stored in an array of size $\lceil m/w \rceil$ on a w-bit architecture

000	$a_{m-1} \ldots a_{(t-1)w}$	$\ldots\ldots\ldots\ldots$	$a_{w-1} \ldots\ldots\ldots\ldots a_1 a_0$

Addition in $\mathbb{F}_{2^m}$

Polynomial addition: $a + b = c$

$$c = (a_{m-1} \oplus b_{m-1})X^{m-1} + \cdots + (a_1 \oplus b_1)X + (a_0 \oplus b_0)$$

$\oplus$: bitwise exclusive-or (XOR)

On a w-bit architecture, this requires $t = \lceil m/w \rceil$ independent XOR instructions

The result is a polynomial of degree at most $m - 1$. No reduction required

Multiplication in $\mathbb{F}_2[X]$

Polynomial multiplication is very similar to integer multiplication, but without carry propagation. It should then be faster!

This is not true for the vast majority of nowadays processors which do not embed an instruction to perform a multiplication over $\mathbb{F}_2[X]$ at the word level (*)

Let $a(X) = 1 + X^4 + X^7 + X^{10}$. On a 4-bit architecture, a can be stored in an array of size 3: $(0\,1\,0\,0 \quad 1\,0\,0\,1 \quad 0\,0\,0\,1)$

$$\begin{aligned} a(X) \cdot b(X) &= b(X) + X^4 b(X) + X^7 b(X) + X^{10} b(X) \\ &= b(X) + X^4(b(X) + X^3 b(X)) + X^8(X^2 b(X)) \end{aligned}$$

Multiplication in $\mathbb{F}_2[X]$

Polynomial multiplication is very similar to integer multiplication, but without carry propagation. It should then be faster!

This is not true for the vast majority of nowadays processors which do not embed an instruction to perform a multiplication over $\mathbb{F}_2[X]$ at the word level (*)

Let $a(X) = 1 + X^4 + X^7 + X^{10}$. On a 4-bit architecture, a can be stored in an array of size 3: $(0\,1\,0\,0 \quad 1\,0\,0\,1 \quad 0\,0\,0\,1)$

$$\begin{aligned} a(X) \cdot b(X) &= b(X) + X^4 b(X) + X^7 b(X) + X^{10} b(X) \\ &= b(X) + X^4(b(X) + X^3 b(X)) + X^8(X^2 b(X)) \end{aligned}$$

(*) PCLMULQDQ: new instruction available on the Intel Westmere family for multiplying two 64-bit operands without carry propagation, producing a 127-bit result

Squaring in $\mathbb{F}_2[X]$

Squaring is linear

$$a(X) = a_{m-1}X^{m-1} + \cdots + a_1 X + a_0$$

$$a(X)^2 = a_{m-1}X^{2m-2} + \cdots + a_2 X^4 + a_1 X^2 + a_0$$

$$(a_{m-1}, a_{m-2}, \ldots, a_2, a_1, a_0)$$

$$\downarrow$$

$$(a_{m-1}, 0, a_{m-2}, \ldots, a_2, 0, a_1, 0, a_0)$$

Reduction modulo $f(X)$

The product of $a(X)$ and $b(X)$ of degree at most $m-1$ is of degree at most $2m-2$ to be reduced modulo $f(X) = X^m + g(X)$

$$c(X) = c_{2m-2}X^{2m-2} + \cdots + c_m X^m + c_{m-1}X^{m-1} + \cdots + c_0$$

$$\equiv (c_{2m-2}X^{m-2} + \cdots + c_m)g(X) + c_{m-1}X^{m-1} + \cdots + c_0 \pmod{f(X)}$$

Reduction modulo $f(X)$ can be accelerated when $g(X)$ is of low-degree and/or $f(X)$ is a trinomial or a pentanomial

NIST reduction polynomials for curves defined over $\mathbb{F}_{2^m}$

The extension degrees m are prime and were selected so that there exists a Koblitz curve over $\mathbb{F}_{2^m}$ having almost-prime group order

$$f(X) = X^{163} + X^7 + X^6 + X^3 + 1$$

$$f(X) = X^{233} + X^{74} + 1$$

$$f(X) = X^{283} + X^{12} + X^7 + X^5 + 1$$

$$f(X) = X^{409} + X^{87} + 1$$

$$f(X) = X^{571} + X^{10} + X^5 + X^2 + 1$$

Optimal extension fields

Let p be prime and let $f(X)$ be an irreducible polynomial with coefficients in $\mathbb{F}_p$ of degree m

$$\mathbb{F}_{p^m} = \mathbb{F}_p[X]/(f(X))$$

An Optimal Extension Field (OEF) is a finite field $\mathbb{F}_{p^m}$ such that

- $p = 2^n - c$ with $|c| \leq n/2$
- there exists $a \in \mathbb{F}_p$ s.t. $f(X) = X^m - a$ is irreducible

Type I: $c \in \{\pm 1\}$

Type II: $w = 2$

Example: $\mathbb{F}_{2^{13}-1}[X]/(X^{13} - 2)$ is both a type I and II OEF


R. P. Brent and P. Zimmermann.
Modern Computer Arithmetic.
Cambridge University Press, 2010.


H. Cohen and G. Frey, editors.
Handbook of Elliptic and Hyperelliptic Curve Cryptography.
CRC Press, 2005.


R. Crandall and C. Pomerance.
Prime Numbers. A Computational Perspective.
Springer, 2001.


D. Hankerson, A. Menezes, and S. Vanstone.
Guide to Elliptic Curve Cryptography.
Springer, 2004.


D. E. Knuth.
The Art of Computer Programming, Vol. 2: Seminumerical Algorithms.
Addison-Wesley, Reading, MA, third edition, 1997.


A. Menezes, P. C. Van Oorschot, and S. A. Vanstone.
Handbook of applied cryptography.
CRC Press, 1997.


J. Von Zur Gathen and J. Gerhard.
Modern Computer Algebra.
Cambridge University Press, 1999.