SDU
University of
Southern Denmark

Cryptographic Engineering

Multiprecision Arithmetic II

Ruben Niederhagen
(based on content by Peter Schwabe)

Department of Mathematics and Computer Science (IMADA)

Department of Mathematics
and Computer Science (IMADA)

Last time, in “Multiprecision Arithmetic I”:

- Addition, subtraction,
- schoolbook multiplication,
- operation orders,
- Karatsuba, and faster algorithms.

Examples on a small 8-bit AVR microcontroller.

From 8-bit to 64-bit processors:

Main differences (for us):

- Arithmetic on larger (64-bit) integers.
- Arithmetic on floating-point numbers.
- Pipelined and superscalar execution.
- (Arithmetic on vectors.)

Department of Mathematics
and Computer Science (IMADA)

Consider representing 255-bit integers:

- Obvious choice: use four 64-bit integers a_0, a_1, a_2, a_3 with

$$A = \sum_{i=0}^{3} a_i 2^{64i}$$

- Arithmetic works just as before (except with larger registers).

Radix-2^{51} Representation

- Radix-2^{64} representation works and is sometimes a good choice.
- It highly depends on the efficiency of handling carries:

  Example 1: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles. (Carries cost a factor of 6!)

  Example 2: When using vector arithmetic, carries are typically lost (*very* expensive to recompute).

Department of Mathematics
and Computer Science (IMADA)

- Let's get rid of the carries:
  Represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{51i}.$$

- This is called radix-2^{51} representation.
- Multiple ways to write the same integer A, for example $A = 2^{52}$:
  - (2^{52} , 0, 0, 0, 0)
  - (0, 2, 0, 0, 0)
- Let's call a representation $(a_0, a_1, a_2, a_3, a_4)$ reduced, if all $a_i \in [0, \ldots, 2^{51} - 1]$.

Addition of two `bigint255`'s:


```
typedef struct{
  unsigned long long a[5];
} bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] + y->a[0];
  r->a[1] = x->a[1] + y->a[1];
  r->a[2] = x->a[2] + y->a[2];
  r->a[3] = x->a[3] + y->a[3];
  r->a[4] = x->a[4] + y->a[4];
}
```

- This definitely works for reduced inputs.
- The result is not necessarily in reduced representation.
- This actually works as long as all coefficients are in $[0, \ldots, 2^{63-1}]$.
- We can do quite a few additions before we have to carry (reduce).

Subtraction of two `bigint255`'s:


```
typedef struct{
  signed long long a[5];
} bigint255;

void bigint255_sub(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  r->a[0] = x->a[0] - y->a[0];
  r->a[1] = x->a[1] - y->a[1];
  r->a[2] = x->a[2] - y->a[2];
  r->a[3] = x->a[3] - y->a[3];
  r->a[4] = x->a[4] - y->a[4];
}
```

- Slightly updated `bigint255` definition to work with *signed* 64-bit integers.
- Reduced if coefficients are in $[-2^{51}+1, 2^{51}-1]$.

Carrying in radix-2^{51}:

- After many additions, coefficients may grow larger than 63 bits.
- They grow even faster during multiplication.
- Eventually we have to *carry* en bloc:

```
signed long long carry = r.a[0] >> 51;
r.a[1] += carry;
carry <<= 51;
r.a[0] -= carry;
```

Department of Mathematics
and Computer Science (IMADA)

- Note: Addition code would look exactly the same for 5-coefficient polynomial addition.
- This is no coincidence: We actually perform arithmetic in $\mathbb{Z}[x]$.
- Inputs to addition are 5-coefficient polynomials:

$$f(X) = a_4X^4 + a_3X^3 + a_2X^2 + a_1X + a_0$$

- Nice thing about arithmetic in $\mathbb{Z}[x]$: no carries!
- To go from $\mathbb{Z}[x]$ to $\mathbb{Z}$, evaluate at the radix (this is a ring homomorphism):

$$f(2^{51}) = a_4(2^{51})^4 + a_3(2^{51})^3 + a_2(2^{51})^2 + a_1 2^{51} + a_0$$

- Carrying means evaluating at the radix.
- Thinking of multiprecision integers as polynomials is helpful for efficient arithmetic.

Department of Mathematics
and Computer Science (IMADA)

- On some microarchitectures floating-point arithmetic is faster than integer arithmetic.
- An IEEE-754 floating-point number has value

$$(-1)^s \cdot (1.b_{m-1}b_{m-2}\ldots b_0) \cdot 2^{e-t} \text{ with } b_i \in \{0,1\}.$$

- For double-precision floats:
  - $s \in \{0,1\}$ “sign bit”
  - $m = 52$ “mantissa bits”
  - $e \in \{1,\ldots,2046\}$ “exponent”
  - $t = 1023$
- For single-precision floats:
  - $s \in \{0,1\}$ “sign bit”
  - $m = 23$ “mantissa bits”
  - $e \in \{1,\ldots,254\}$ “exponent”
  - $t = 127$
- Exponent = 0 used to represent 0.
- Any number that can be represented like this, will be precise.
- Other numbers will be *rounded*, according to a rounding mode.

Addition and Subtraction


```
typedef struct{
  double a[12];
} bigint255;

void bigint255_add(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  int i;
  for (i=0; i<12; i++)
    r->a[i] = x->a[i] + y->a[i];
}
```

```
void bigint255_sub(bigint255 *r,
                   const bigint255 *x,
                   const bigint255 *y)
{
  int i;
  for (i=0; i<12; i++)
    r->a[i] = x->a[i] - y->a[i];
}
```

Department of Mathematics
and Computer Science (IMADA)

- For carrying integers we used a right shift (discard lowest bits).
- For floating-point numbers we can use multiplication by the inverse of the radix.

  Example: Radix 2^{22}, multiply by 2^{-22}.
- This does not cut off lowest bits, need to round.
- Some processors have efficient rounding instructions, e.g., `vroundpd`.
- Otherwise (for double-precision):
  - add constant $2^{52} + 2^{51}$,
  - subtract constant $2^{52} + 2^{51}$.

  This will round the number to an integer according to the rounding mode (to nearest, towards zero, away from zero, or truncate).

Modular Reduction

- We don't just need arithmetic on big integers:
  We need arithmetic in finite fields.
- In other words, we need reduction modulo a prime p.
- Let's fix some size and representation:

```
/* 256-bit integers in radix 2^16 */
typedef signed long long bigint[16];
```

- Integer A is obtained as $\sum_{i=0}^{15} a_i 2^{16i}$.
- Lot of space in top of limbs to accumulate carries.

Modular Reduction


```
void mul_prodscan(signed long long r[31],
                  const long long x[16],
                  const long long y[16])
{
  r[ 0]  = x[ 0] * y[ 0];
  r[ 1]  = x[ 1] * y[ 0];
  r[ 1] += x[ 0] * y[ 1];
  r[ 2]  = x[ 2] * y[ 0];
  r[ 2] += x[ 1] * y[ 1];
  r[ 2] += x[ 0] * y[ 2];
  ...
  r[29] += x[14] * y[15];
  r[30]  = x[15] * y[15];
}
```

$$R = r_{31}2^{496} + \cdots + r_{16}2^{256} + \cdots + r_1 2^{16} + r_0$$

- Let's fix some p, say $p = 2^{255} - 19$.
- We know that $2^{255} \equiv 19 \mod p$.
- This means that $2^{256} \equiv 38 \mod p$.

  $h \cdot 2^{256} \equiv h \cdot 38 \mod p$

  $h \cdot 2^{256} + l \equiv h \cdot 38 + l \mod p$
- Reduce 31-bit intermediate result `r` as:

```
for (i=0; i<15; i++)
  r[i] += 38 * r[i+16];
```

- Result is in `r[0]`, ..., `r[15]`.

Department of Mathematics
and Computer Science (IMADA)

- “You cannot just simply pull some nice prime out of your hat!”
- In fact, very often we can.
- For cryptography we construct curves over fields of “nice” order.

Examples:

- $2^{192} - 2^{64} - 1$ (“NIST-P192”, FIPS186-2, 2000)
- $2^{224} - 2^{96} + 1$ (“NIST-P224”, FIPS186-2, 2000)
- $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ (“NIST-P256”, FIPS186-2, 2000)
- $2^{255} - 19$ (Bernstein, 2006)
- $2^{251} - 9$ (Bernstein, Hamburg, Krasnova, Lange, 2013)
- $2^{448} - 2^{224} - 1$ (Hamburg, 2015)

- All these primes come with (more or less) fast reduction algorithms.
- More about *general* primes later.
- For the moment let’s stick to $2^{255} - 19$.

Carrying after Multiplication


```
long long c;
for (i=0; i<15; i++)
{
  c = r[i] >> 16;
  r[i+1] += c;
  c <<= 16;
  r[i] -= c;
}
c = r[15] >> 16;
r[0] += 38*c;
c <<= 16;
r[15] -= c;
```

Coefficient `r[0]` may still be too large:
carry again to `r[1]`.

How about squaring?


```
#define bigint_square(R,X) bigint_mul(R,X,X)
```

How about squaring?


```
void square_prodscan(signed long long r[31],
                     const long long x[16])
{
  signed long long _2x[16];
  for (int i=0; i<16; i++)
    _2x[i] = 2*x[i];

  r[0]  =   x[0] * x[0];
  r[1]  = _2x[1] * x[0];
  r[2]  = _2x[2] * x[0];
  r[2] +=   x[1] * x[1];
  ...
  r[29] = _2x[15] * x[14];
  r[30] =   x[15] * x[15];
}
```

$$
\begin{array}{ccccc}
\cdots & x_3x_0 & x_2x_0 & x_1x_0 & x_0x_0 \\
 & + & + & + & \\
\cdots & x_2x_1 & x_1x_1 & x_0x_1 & \\
 & + & + & & \\
\cdots & x_1x_2 & x_0x_2 & & \\
 & + & & & \\
\cdots & x_0x_3 & & & \\
 & & & & \\
 & \vdots & & &
\end{array}
$$

Department of Mathematics
and Computer Science (IMADA)

Multiplication needs:

- 256 multiplications and
- 225 additions.

Squaring needs:

- 136 multiplications,
- 105 additions, and
- 15 additions or shifts or multiplications by 2 for precomputation.

How about other prime fields?

- So far: Reductions only modulo “nice” primes.
- What if somebody just throws an ugly prime at you?
- Example: German BSI pushing the “Brainpool curves” over fields $\mathbb{F}_p$ with

$$p_{224} = 22721622932454352787552537995910928073340732145944992304435472941311$$
$$= \texttt{0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF}$$

or

$$p_{256} = 76884956397045344220809746629001649093037950200943055203735601445031516197751$$
$$= \texttt{0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623526202820134 81D1F6E5377}$$

- Another example: Pairing-friendly curves are typically defined over fields $\mathbb{F}_p$ where p has some structure, but hard to exploit for fast arithmetic.

Montgomery Representation

- We have the following problem:
  - We multiply two n-limb big integers and obtain a $2n$-limb result t.
  - We need to find $t \bmod p$.
- Idea: Perform big-integer division with remainder (expensive!).
- Better idea (Montgomery, 1985):
  - Let R be such that $\gcd(R, p) = 1$ and $t < p \cdot R$.
  - Represent an element a of $\mathbb{F}_p$ as $aR \bmod p$.
  - Multiplication of aR and bR yields $t = abR^2$ ($2n$ limbs).
  - Now compute *Montgomery reduction*: $tR^{-1} \bmod p$.
  - For *some* choices of R this is more efficient than division.
  - Typical choice for radix-b representation: $R = b^n$.

Montgomery Reduction (pseudocode)

Require: $p = (p_{n-1}, \ldots, p_0)_b$ with $\gcd(p, b) = 1, R = b^n$,
$p' = -p^{-1} \mod b$ and $t = (t_{2n-1}, \ldots, t_0)_b$.
Ensure: $tR^{-1} \mod p$.

$A \leftarrow t$
for $i = 0$ **to** $n - 1$ **do**
  $u \leftarrow a_i p' \mod b$
  $A \leftarrow A + u \cdot p \cdot b^i \equiv A \pmod p$ $a_i \cdot p' \cdot p \equiv -a_i \pmod b$ $\rightarrow$ Word a_i becomes 0.
end for
$A \leftarrow A/b^n$ Now, division is equivalent to A shifted by n to the right.
if $A \geq p$ **then**
  $A \leftarrow A - p$
end if
return A

Some notes about Montgomery reduction:

- Some cost for transforming to Montgomery representation and back.
- Only efficient if many operations are performed in Montgomery representation.
- The algorithms requires $n^2 + n$ multiplication instructions.
- n of those are “shortened” multiplications (modulo b).
- The cost is roughly the same as schoolbook multiplication.
- Careful about conditional subtraction (timing attacks!).
- One can merge schoolbook multiplication with Montgomery reduction: “Montgomery multiplication”.

Still missing: Inversion

- Inversion is typically much more expensive than multiplication.
- Efficient ECC arithmetic avoids frequent inversions.
- ECC can typically not avoid all inversions.

$\Rightarrow$ We need inversion, but we do (usually) not need it often.

- Two approaches to inversion:
  1. Extended Euclidean algorithm, and
  2. Fermat's little theorem.

Department of Mathematics
and Computer Science (IMADA)

- Given two integers a, b, the Extended Euclidean algorithm finds:
  - the greatest common divisor of a and b as well as
  - integers u and v, such that $a \cdot u + b \cdot v = \gcd(a, b)$.
- To compute $a^{-1} \mod p$, use the algorithm to compute

$$a \cdot u + p \cdot v = \gcd(a, p) = 1.$$

- Now it holds that $u \equiv a^{-1} \mod p$.

Extended Euclidean Algorithm (pseudocode)

Require: Integers a and b.
Ensure: An integer tuple (u, v, d) satisfying $a \cdot u + b \cdot v = d = \gcd(a, b)$.

$u \leftarrow 1$; $v \leftarrow 0$; $d \leftarrow a$
$v_1 \leftarrow 0$; $v_3 \leftarrow b$
while $(v_3 \neq 0)$ **do**
  $q \leftarrow \lfloor \frac{d}{v_3} \rfloor$
  $t_3 \leftarrow d \mod v_3$
  $t_1 \leftarrow u - qv_1$
  $u \leftarrow v_1$; $d \leftarrow v_3$
  $v_1 \leftarrow t_1$; $v_3 \leftarrow t_3$
end while
$v \leftarrow \frac{d-au}{b}$
return (u, v, d)

Some notes about the Extended Euclidean algorithm:

- Core operation are divisions with remainder.
- This lecture: No details about big-integer division.
- Version without divisions: *binary extended gcd*.
  (Handbook of applied cryptography, Alg. 14.61)
- The running time (number of loop iterations) depends on the inputs.
- We usually do not want this for cryptography (timing attacks!).
- Possible protection — *blinding*:
  - Multiply a by random integer r.
  - Invert, obtain $r^{-1}a^{-1}$.
  - Multiply again by r to obtain a^{-1}.
- Note that this requires a source of randomness.

Department of Mathematics
and Computer Science (IMADA)

Theorem

Let p be prime. Then for any integer a it holds that $a^{p-1} \equiv 1 \mod p$.

- This implies that $a^{p-2} \equiv a^{-1} \mod p$.
- Obvious algorithm for inversion: exponentiation with $p - 2$.
- The exponent is quite large (e.g., 255 bits), is that efficient?
- Yes, fairly:
  - Exponent is fixed and known at compile time.
  - Can spend quite some time on finding an efficient addition chain (next lecture).
  - Inversion modulo $2^{255} - 19$ needs 254 squarings and 11 multiplications in $\mathbb{F}_{2^{255}-19}$.

Inversion in $\mathbb{F}_{2^{255}-19}$


```
void gfe_invert(gfe r, const gfe x)
{
  gfe z2, z9, z11, z2_5_0, z2_10_0, z2_20_0, z2_50_0, z2_100_0, t;
  int i;
  /* 2 */               gfe_square(z2,x);
  /* 4 */               gfe_square(t,z2);
  /* 8 */               gfe_square(t,t);
  /* 9 */               gfe_mul(z9,t,x);
  /* 11 */              gfe_mul(z11,z9,z2);
  /* 22 */              gfe_square(t,z11);
  /* 2^5 - 2^0 = 31 */ gfe_mul(z2_5_0,t,z9);
  /* 2^6 - 2^1 */      gfe_square(t,z2_5_0);
  /* 2^10 - 2^5 */     for (i = 1;i < 5;i++) { gfe_square(t,t); }
  /* 2^10 - 2^0 */     gfe_mul(z2_10_0,t,z2_5_0);
  /* 2^11 - 2^1 */     gfe_square(t,z2_10_0);
  /* 2^20 - 2^10 */    for (i = 1;i < 10;i++) { gfe_square(t,t); }
  /* 2^20 - 2^0 */     gfe_mul(z2_20_0,t,z2_10_0);
  /* 2^21 - 2^1 */     gfe_square(t,z2_20_0);
  /* 2^40 - 2^20 */    for (i = 1;i < 20;i++) { gfe_square(t,t); }
  /* 2^40 - 2^0 */     gfe_mul(t,t,z2_20_0);
  /* 2^41 - 2^1 */     gfe_square(t,t);
  /* 2^50 - 2^10 */    for (i = 1;i < 10;i++) { gfe_square(t,t); }
  /* 2^50 - 2^0 */     gfe_mul(z2_50_0,t,z2_10_0);
  /* 2^51 - 2^1 */     gfe_square(t,z2_50_0);
  /* 2^100 - 2^50 */   for (i = 1;i < 50;i++) { gfe_square(t,t); }
  /* 2^100 - 2^0 */    gfe_mul(z2_100_0,t,z2_50_0);
  /* 2^101 - 2^1 */    gfe_square(t,z2_100_0);
  /* 2^200 - 2^100 */  for (i = 1;i < 100;i++) { gfe_square(t,t); }
  /* 2^200 - 2^0 */    gfe_mul(t,t,z2_100_0);
  /* 2^201 - 2^1 */    gfe_square(t,t);
  /* 2^250 - 2^50 */   for (i = 1;i < 50;i++) { gfe_square(t,t); }
  /* 2^250 - 2^0 */    gfe_mul(t,t,z2_50_0);
  /* 2^251 - 2^1 */    gfe_square(t,t);
  /* 2^252 - 2^2 */    gfe_square(t,t);
  /* 2^253 - 2^3 */    gfe_square(t,t);
  /* 2^254 - 2^4 */    gfe_square(t,t);
  /* 2^255 - 2^5 */    gfe_square(t,t);
  /* 2^255 - 21 */     gfe_mul(r,t,z11);
}
```

Department of Mathematics
and Computer Science (IMADA)

- Why would you write low-level arithmetic yourself?
- Aren't there some good libraries for this?
- There are:
  - GMP (`http://gmplib.org`), high-performance arithmetic on multiprecision numbers.
  - NTL (`http://shoup.net/ntl/`), number-theory library, higher level than GMP, uses GMP.
  - OpenSSL Bignum (`http://openssl.org`), low-level routines in OpenSSL.
  - `mp`$\mathbb{F}_q$ (`http://mpfq.gforge.inria.fr/`), a finite-field library (generator).

Limitations of Libraries

- Libraries don't know the modulus (except for $\texttt{mp}\mathbb{F}_q$) and therefore cannot optimize for a fixed modulus.
- Libraries don't know the sequence of field operations you're computing (e.g., point addition) and therefore cannot use lazy reduction.
- Libraries are not necessarily intended for cryptography and thus are not always timing-attack protected.

$\Rightarrow$ Consequence: ECC speed records are achieved with hand-optimized assembly implementations.