Engineering Cryptographic Software

Multiprecision Arithmetic

Peter Schwabe

January 2026

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on “big integers”
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on “big integers”
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers
- Example 2:
  - Elliptic curves defined over finite fields
  - Typically use EC over large-characteristic prime fields
  - Typical field sizes: (160 bits, 192 bits), 256 bits, 448 bits $\ldots$

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on "big integers"
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers
- Example 2:
  - Elliptic curves defined over finite fields
  - Typically use EC over large-characteristic prime fields
  - Typical field sizes: $(160$ bits, 192 bits), 256 bits, 448 bits $\ldots$
- Example 3: Poly1305 needs arithmetic on 130-bit integers

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on “big integers”
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers
- Example 2:
  - Elliptic curves defined over finite fields
  - Typically use EC over large-characteristic prime fields
  - Typical field sizes: $(160$ bits, 192 bits$)$, 256 bits, 448 bits $\dots$
- Example 3: Poly1305 needs arithmetic on 130-bit integers
- An integer is “big” if it’s not natively supported by the machine architecture
- Example: ARMv7E-M supports up to 32-bit integers, multiplication produces 64-bit result, but not bigger than that.
- We call arithmetic on such “big integers” *multiprecision arithmetic*

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on "big integers"
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers
- Example 2:
  - Elliptic curves defined over finite fields
  - Typically use EC over large-characteristic prime fields
  - Typical field sizes: (160 bits, 192 bits), 256 bits, 448 bits $\dots$
- Example 3: Poly1305 needs arithmetic on 130-bit integers
- An integer is "big" if it's not natively supported by the machine architecture
- Example: ARMv7E-M supports up to 32-bit integers, multiplication produces 64-bit result, but not bigger than that.
- We call arithmetic on such "big integers" *multiprecision arithmetic*
- For now mainly interested in 160-bit and 256-bit arithmetic

The first year of primary school

Available numbers (digits): $(0), 1, 2, 3, 4, 5, 6, 7, 8, 9$

The first year of primary school

Available numbers (digits): $(0), 1, 2, 3, 4, 5, 6, 7, 8, 9$

Addition

$$3 + 5 = \quad ?$$
$$2 + 7 = \quad ?$$
$$4 + 3 = \quad ?$$

The first year of primary school

Available numbers (digits): $(0), 1, 2, 3, 4, 5, 6, 7, 8, 9$

Addition

$3 + 5 = \quad ?$
$2 + 7 = \quad ?$
$4 + 3 = \quad ?$

Subtraction

$7 - 5 = \quad ?$
$5 - 1 = \quad ?$
$9 - 3 = \quad ?$

The first year of primary school

Available numbers (digits): $(0), 1, 2, 3, 4, 5, 6, 7, 8, 9$

Addition

$$3 + 5 = \quad ?$$
$$2 + 7 = \quad ?$$
$$4 + 3 = \quad ?$$

Subtraction

$$7 - 5 = \quad ?$$
$$5 - 1 = \quad ?$$
$$9 - 3 = \quad ?$$

- All results are in the set of available numbers
- No confusion for first-year school kids

Programming today

Available numbers: $0, 1, \ldots, 2^{32} - 1$

Programming today

Available numbers: $0, 1, \ldots, 2^{32} - 1$

Addition

```
u32 a b r;
a = 23842;
b = 12390;
r = a + b;
```

Programming today

Available numbers: $0, 1, \ldots, 2^{32} - 1$

Addition

```
u32 a b r;
a = 23842;
b = 12390;
r = a + b;
```

Subtraction

```
u32 a b r;
a = 874157;
b = 622301;
r = a - b;
```

Programming today

Available numbers: $0, 1, \ldots, 2^{32} - 1$

Addition

```
u32 a b r;
a = 23842;
b = 12390;
r = a + b;
```

Subtraction

```
u32 a b r;
a = 874157;
b = 622301;
r = a - b;
```

- All results are in the set of available numbers
- On other architectures, may also have `u64` available, or maybe only `u16` or `u8`
- On Cortex-M4 (ARMv7E-M), working with register-size `u32` is natural

Still in the first year of primary school

Crossing the ten barrier

$$
\begin{aligned}
6 + 5 &= \quad ? \\
9 + 7 &= \quad ?
\end{aligned}
$$

Still in the first year of primary school

Crossing the ten barrier

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$

- Inputs to addition are still from the set of available numbers
- Results are allowed to be larger than 9

Crossing the ten barrier

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$

- Inputs to addition are still from the set of available numbers
- Results are allowed to be larger than 9
- Addition is allowed to produce a *carry*

Still in the first year of primary school

Crossing the ten barrier

$6 + 5 = \quad ?$
$9 + 7 = \quad ?$

- Inputs to addition are still from the set of available numbers
- Results are allowed to be larger than 9
- Addition is allowed to produce a *carry*

What happens with the carry?

- Introduce the decimal positional system
- Write an integer A in two digits $a_1 a_0$ with

$$A = 10 \cdot a_1 + a_0$$

- Note that at the moment $a_1 \in \{0, 1\}$

. . . back to programming

```
reg u32 a b r;
a = 3348129313;
b = 3810627668;
r = a + b;
```

. . . back to programming

```
reg u32 a b r;
a = 3348129313;
b = 3810627668;
r = a + b;
```

- Result of integer addition is $7\,158\,756\,981$
- The result `r` now has the value of $2\,863\,789\,685$

. . . back to programming

```
reg u32 a b r;
a = 3348129313;
b = 3810627668;
r = a + b;
```

- Result of integer addition is $7\,158\,756\,981$
- The result `r` now has the value of $2\,863\,789\,685$
- $2\,863\,789\,685 = 7\,158\,756\,981 - 2^{32}$
- Addition result produced a carry, which is lost. What do we do?

. . . back to programming


```
reg u32 a b r;
a = 3348129313;
b = 3810627668;
r = a + b;
```

- Result of integer addition is $7\,158\,756\,981$
- The result `r` now has the value of $2\,863\,789\,685$
- $2\,863\,789\,685 = 7\,158\,756\,981 - 2^{32}$
- Addition result produced a carry, which is lost. What do we do?
- Idea: obtain the carry, and put it into another `u32`

3348129313 + 3810627668


```
u32 a = 3348129313;
u32 b = 3810627668;

fn addab() -> reg u32[2] {
  reg u32[2] r;
  reg bool c;
  c, r[0] = a + b;
  r[1] = 0;
  _, r[1] += r[1] + c;
  return r;
}
```

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

Later in primary school

Addition

$42 + 78 = \quad ?$
$789 + 543 = \quad ?$
$7862 + 5275 = \quad ?$

$$\begin{array}{cr} & 7862 \\ + & 5275 \\ \hline + & 7 \end{array}$$

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

$$
\begin{array}{cr}
 & 7862 \\
+ & 5275 \\
\hline
+ & 37
\end{array}
$$

Later in primary school

Addition

$42 + 78 = \quad ?$
$789 + 543 = \quad ?$
$7862 + 5275 = \quad ?$

$$\begin{array}{rr} & 7862 \\ + & 5275 \\ \hline + & 137 \end{array}$$

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

$$
\begin{array}{cr}
 & 7862 \\
+ & 5275 \\
\hline
+ & 13137
\end{array}
$$

Later in primary school

Addition

$42 + 78 = \quad ?$
$789 + 543 = \quad ?$
$7862 + 5275 = \quad ?$

$$\begin{array}{rr} & 7862 \\ + & 5275 \\ \hline + & 13137 \end{array}$$

- Once school kids can add beyond 1000, they can add arbitrary numbers

Multiprecision addition is old

“Oh Līlāvatī, intelligent girl, if you understand addition and subtraction, tell me the sum of the amounts 2, 5, 32, 193, 18, 10, and 100, as well as [the remainder of] those when subtracted from 10000.”
—“Līlāvatī” by Bhāskara (1150)

Multiprecision addition in Jasmin


```
fn bigint_add(reg ptr u32[N+1] r, reg ptr u32[N] a b) -> reg ptr u32[N+1] {
  reg u32 t, u;
  reg bool c;
  inline int i;

  t = a[0];
  u = b[0];
  c, t += u;
  r[0] = t;
  for i = 1 to N {
    t = a[i];
    u = b[i];
    c, t += u + c;
    r[i] = t;
  }
  t = 0;
  _, t += t + c;
  r[N] = t;

  return r;
}
```

. . . and subtraction


```
fn bigint_sub(reg ptr u32[N+1] r, reg ptr u32[N] a b) -> reg ptr u32[N+1] {
  reg u32 t, u;
  reg bool c;
  inline int i;

  t = a[0];
  u = b[0];
  c, t -= u;
  r[0] = t;
  for i = 1 to N {
    t = a[i];
    u = b[i];
    c, t -= u - c;
    r[i] = t;
  }
  t = 0;
  _, t -= t - c;
  r[N] = t;

  return r;
}
```

How about multiplication?

- Consider multiplication of 1234 by 789

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 6 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 06 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 106 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 11106 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 11106 \\ 9872 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 11106 \\ 9872 \\ 8638 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$
\begin{array}{lr}
 & 1234 \cdot 789 \\
\hline
 & 11106 \\
+ & 9872 \\
+ & 8638 \\
\hline
 & 973626
\end{array}
$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 11106 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{lr} & 1234 \cdot 789 \\ \hline & 11106 \\ + & 9872 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 20978 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$
\begin{array}{r}
1234 \cdot 789 \\
\hline
20978 \\
+ \quad 8638
\end{array}
$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 973626 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\frac{1234 \cdot 789}{973626}$$

- This is also an old technique
- Earliest reference I could find is again the Līlāvatī (1150)

Let’s do that in Jasmin


```
export fn bigint_mul(reg mut ptr u32[6] rp, reg ptr u32[3] ap bp) -> reg ptr u32[6] {
```

Let's do that in Jasmin


```
reg u32 r0 r1 r2 r3 r4 r5;
reg u32 a0 a1 a2;
reg u32 b0 b1 b2;
reg u32 t0 t1 t2 t3 hi lo z;
reg bool c;
z = 0;

a0 = ap[0];
a1 = ap[1];
a2 = ap[2];

b0  = bp[0];
t1, r0 = a0 * b0;
rp[0] = r0;

hi, r1 =  a1 * b0;
c, r1 += t1;
c, hi += z + c;

r3, r2 = a2 * b0;
c, r2 += hi + c;
_, r3 += z + c;
```

```
b1  = bp[1];
t1, t0 = a0 * b1;

hi, lo =  a1 * b1;
c, t1 += lo;

r4, t2 = a2 * b1;
c, t2 += hi + c;
c, r4 += z + c;

c, r1 += t0;
c, r2 += t1 + c;
c, r3 += t2 + c;
_, r4 += z + c;
rp[1] = r1;
```

```
  b2  = bp[2];
  t1, t0 = a0 * b2;

  hi, lo =  a1 * b2;
  c, t1 += lo;

  r5, t2 = a2 * b2;
  c, t2 += hi + c;
  _, r5 += z + c;

  c, r2 += t0;
  c, r3 += t1 + c;
  c, r4 += t2 + c;
  _, r5 += z + c;

  rp[2] = r2;
  rp[3] = r3;
  rp[4] = r4;
  rp[5] = r5;

  return rp;
}
```

Schoolbook multiplication analysis

- n^2 multiplication instructions to multiply two n-limb big integers
- About 2 additions per multiplication

Schoolbook multiplication analysis

- n^2 multiplication instructions to multiply two n-limb big integers
- About 2 additions per multiplication
- Problem: Need $3n + c$ registers for $n \times n$-word multiplication

Schoolbook multiplication analysis

- n^2 multiplication instructions to multiply two n-limb big integers
- About 2 additions per multiplication
- Problem: Need $3n + c$ registers for $n \times n$-word multiplication
- Can add on the fly, get down to $2n + c$, but more carry handling

Can we do better?

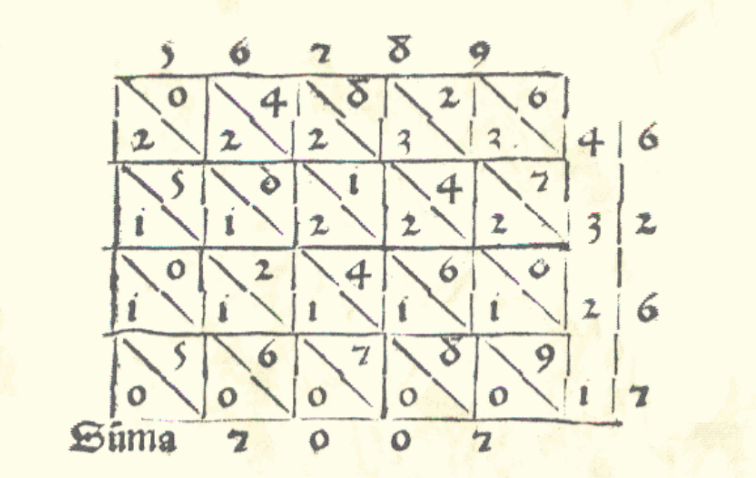
> *"Again as the information is understood, the multiplication of 2345 by 6789 is proposed; therefore the numbers are written down; the 5 is multiplied by the 9, there will be 45; the 5 is put, the 4 is kept; and the 5 is multiplied by the 8, and the 9 by the 4 and the products are added to the kept 4; there will be 80; the 0 is put and the 8 is kept; and the 5 is multiplied by the 7 and the 9 by the 3 and the 4 by the 8, and the products are added to the kept 8; there will be 102; the 2 is put and the 10 is kept in hand. . . "*

From "Fibonacci's Liber Abaci" (1202) Chapter 2
(English translation by Sigler)

Product scanning in Jasmin


```
z = 0;

a0 = ap[0];
a1 = ap[1];
a2 = ap[2];

b0 = bp[0];
b1 = bp[1];
b2 = bp[2];

r1, r0 = a0 * b0;
rp[0] = r0;


r2, lo = a0 * b1;
c, r1 += lo;
_, r2 += z + c;

hi, lo = a1 * b0;
c, r1 += lo;
c, r2 += hi + c;
_, r3 = z + z + c;
rp[1] = r1;

hi, lo = a0 * b2;
c, r2 += lo;
c, r3 += hi + c;
_, r4 = z + z + c;

hi, lo = a1 * b1;
c, r2 += lo;
c, r3 += hi + c;
_, r4 += z + c;

hi, lo = a2 * b0;
c, r2 += lo;
c, r3 += hi + c;
_, r4 += z + c;
rp[2] = r2;

hi, lo = a1 * b2;
c, r3 += lo;
c, r4 += hi + c;
_, r5 = z + z + c;

hi, lo = a2 * b1;
c, r3 += lo;
c, r4 += hi + c;
_, r5 += z + c;
rp[3] = r3;


hi, lo = a2 * b2;
c, r4 += lo;
_, r5 += hi + c;

rp[4] = r4;
rp[5] = r5;

return rp;
}
```

Even better. . . ?



From the Treviso Arithmetic, 1478

Carries, carries everywhere!

Radix-2^{32} representation

- Currently, represent 256-bit integer A as $(a_0, \ldots, a_7)$ with

$$A = \sum_{i-0}^{7} a_i \cdot 2^{32i}$$

Carries, carries everywhere!

Radix-2^{32} representation

- Currently, represent 256-bit integer A as $(a_0, \ldots, a_7)$ with

$$A = \sum_{i-0}^{7} a_i \cdot 2^{32i}$$

- Very compact, also computationally efficient
- Unique representation for every 256-bit integer
- Every addition may generate carries
- Carry handling may get involved

Carries, carries everywhere!

Radix-2^8 representiaon

- Idea: use “unsaturated” representation $(a_0, \dots, a_{31})$ with

$$A = \sum_{i-0}^{31} a_i \cdot 2^{8i}$$

Carries, carries everywhere!

Radix-2^8 representiaon

- Idea: use "unsaturated" representation $(a_0, \ldots, a_{31})$ with

$$A = \sum_{i-0}^{31} a_i \cdot 2^{8i}$$

- More computations per big-integer operation
- Various ways to represent the same 256-bit integer, e.g., $512 = 2^9$
  - $(512, 0, 0, 0, 0, 0, 0, 0)$
  - $(0, 2, 0, 0, 0, 0, 0, 0)$
- Needs more space in memory
- Can ignore carries for quite a while

Some remarks about saturated vs. unsaturated representation

- On Cortex-M4, saturated representation is most efficient
  - Carries are annoying, but cheap
  - Minimize arithmetic and load/stores instructions
  - Setting flags is optional, carries aren't overwritten

Some remarks about saturated vs. unsaturated representation

- On Cortex-M4, saturated representation is most efficient
  - Carries are annoying, but cheap
  - Minimize arithmetic and load/stores instructions
  - Setting flags is optional, carries aren't overwritten
- This is different on other (micro-)architectures
  - RISC-V does not have a carry flag
  - On Intel Nehalem, **`adc`** is $6\times$ slower than **`add`**

Some remarks about saturated vs. unsaturated representation

- On Cortex-M4, saturated representation is most efficient
  - Carries are annoying, but cheap
  - Minimize arithmetic and load/stores instructions
  - Setting flags is optional, carries aren't overwritten
- This is different on other (micro-)architectures
  - RISC-V does not have a carry flag
  - On Intel Nehalem, `adc` is $6\times$ slower than `add`
- More efficient unsaturated code, e.g., radix-2^{26}

Some remarks about saturated vs. unsaturated representation

- On Cortex-M4, saturated representation is most efficient
  - Carries are annoying, but cheap
  - Minimize arithmetic and load/stores instructions
  - Setting flags is optional, carries aren't overwritten
- This is different on other (micro-)architectures
  - RISC-V does not have a carry flag
  - On Intel Nehalem, **`adc`** is $6\times$ slower than **`add`**
- More efficient unsaturated code, e.g., radix-2^{26}
- Unsaturated representation often used for first reference code
- Code in **`assignment2-ecdh25519`** uses radix-2^{8} representation

Addition in radix 2^8


```
fn bigint_add(reg ptr u32[N] r, reg ptr u32[N] a b) -> reg ptr u32[N] {
  reg u32 t, u;
  inline int i;

  for i = 0 to N {
    t = a[i];
    u = b[i];
    t += u;
    r[i] = t;
  }

  return r;
}
```

Addition in radix 2^8


```
fn bigint_add(reg ptr u32[N] r, reg ptr u32[N] a b) -> reg ptr u32[N] {
  reg u32 t, u;
  inline int i;

  for i = 0 to N {
    t = a[i];
    u = b[i];
    t += u;
    r[i] = t;
  }

  return r;
}
```

- This works as long as all coefficients are in $[0, \ldots, 2^{31} - 1]$

Addition in radix 2^8


```
fn bigint_add(reg ptr u32[N] r, reg ptr u32[N] a b) -> reg ptr u32[N] {
  reg u32 t, u;
  inline int i;

  for i = 0 to N {
    t = a[i];
    u = b[i];
    t += u;
    r[i] = t;
  }

  return r;
}
```

- This works as long as all coefficients are in $[0, \ldots, 2^{31} - 1]$
- We can do quite a few additions before we have to carry (reduce)

Subtraction in radix 2^8


```
fn bigint_sub(reg ptr u32[N] r, reg ptr u32[N] a b) -> reg ptr u32[N] {
  reg u32 t, u;
  inline int i;

  for i = 0 to N {
    t = a[i];
    u = b[i];
    t -= u;
    r[i] = t;
  }

  return r;
}
```

- Use signed coefficients to represent our big integers
- No need to worry about borrows

Carrying in radix-2^8

- With many additions, coefficients may grow larger than 31 bits
- They grow even faster with multiplication

Carrying in radix-2^8

- With many additions, coefficients may grow larger than 31 bits
- They grow even faster with multiplication
- Eventually we have to *carry* en bloc:

```
t = r[0];
u = r[1];
t = t >>s 8;
u += t;
t = t << 8;
t -= t;
r[0] = t;
r[1] = u;
```

Carrying in radix-2^8

- With many additions, coefficients may grow larger than 31 bits
- They grow even faster with multiplication
- Eventually we have to *carry* en bloc:

```
t = r[0];
u = r[1];
t = t >>s 8;
u += t;
t = t << 8;
t -= t;
r[0] = t;
r[1] = u;
```

- Continue by carrying from `r1` to `r2`, from `r2` to `r3`, etc.
- For the highest limb `r[N-1]`, need to create a new limb to carry to

- Note: Addition code would look *exactly* the same for 5-coefficient polynomial addition

Big integers and polynomials

- 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

Big integers and polynomials

- 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
- Nice thing about arithmetic in $\mathbb{Z}[x]$: no carries!

Big integers and polynomials

- 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
- 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)
- Carrying means evaluating at the radix

Big integers and polynomials

- 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
- 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)
- Carrying means evaluating at the radix
- Thinking of multiprecision integers as polynomials is very powerful for efficient arithmetic

How about squaring?


```
inline fn bigint_square(reg ptr u32[N] r a) -> reg ptr u32[N] {
  r = bigint_mul(a, a);
}
```

How about squaring?

- What squaring will compute is the following:

$$
\begin{aligned}
r_0 &= a_0 a_0 \\
r_1 &= a_1 a_0 + a_0 a_1 \\
r_2 &= a_2 a_0 + a_1 a_1 + a_0 a_2 \\
&\dots \\
r_{61} &= a_{30} a_{31} + a_{31} a_{30} \\
r_{62} &= a_{31} a_{31}
\end{aligned}
$$

How about squaring?

- What squaring will compute is the following:

$$
\begin{aligned}
r_0 &= a_0 a_0 \\
r_1 &= a_1 a_0 + a_0 a_1 \\
r_2 &= a_2 a_0 + a_1 a_1 + a_0 a_2 \\
&\dots \\
r_{61} &= a_{30} a_{31} + a_{31} a_{30} \\
r_{62} &= a_{31} a_{31}
\end{aligned}
$$

- Many partial products are computed twice!

How about squaring?

- Idea: compute them only once!
- Precompute $2a_1, 2a_2, \ldots, 2a_{31}$, then

$$
\begin{aligned}
r_0 &= a_0 a_0 \\
r_1 &= 2a_1 a_0 \\
r_2 &= 2a_2 a_0 + a_1 a_1 \\
&\cdots \\
r_{61} &= 2a_{30} a_{31} \\
r_{62} &= a_{31} a_{31}
\end{aligned}
$$

How about squaring?

- Idea: compute them only once!
- Precompute $2a_1, 2a_2, \ldots, 2a_{31}$, then

$$
\begin{aligned}
r_0 &= a_0 a_0 \\
r_1 &= 2a_1 a_0 \\
r_2 &= 2a_2 a_0 + a_1 a_1 \\
&\cdots \\
r_{61} &= 2a_{30} a_{31} \\
r_{62} &= a_{31} a_{31}
\end{aligned}
$$

- Eliminate almost half of the multiplications (and additions)
- Precomputation can use addition, shift, or multiplication by 2

Squaring vs. multiplication

For 32 input limbs, multiplication needs

- $32^2 = 1024$ multiplications
- $31^2 = 961$ additions

Squaring needs

- 528 multiplications
- 465 additions
- 31 additions or shifts or multiplications by 2 for precomputation

Multiplication complexity

- So far, multiplication of $2\ n$-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1
- Compute

$$A_0 B_0 + \qquad 2^m (A_0 B_1 + B_0 A_1) \qquad + 2^{2m} A_1 B_1$$

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1
- Compute

$$\begin{aligned} & A_0 B_0 + 2^m (A_0 B_1 + B_0 A_1) + 2^{2m} A_1 B_1 \\ = & A_0 B_0 + 2^m ((A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1) + 2^{2m} A_1 B_1 \end{aligned}$$

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MULs`
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1
- Compute

$$
\begin{aligned}
& A_0 B_0 + 2^m (A_0 B_1 + B_0 A_1) + 2^{2m} A_1 B_1 \\
= & A_0 B_0 + 2^m ((A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1) + 2^{2m} A_1 B_1
\end{aligned}
$$

- Recursive application yields $\Theta(n^{\log_2 3})$ runtime

More on multiplication complexity

- For small multiplication numbers, Karatsuba is typically not faster
- Cutoff between quadratic-complexity and Karatsuba depends on
  - Size of registers and radix used to represent big integers
  - Relative cost of multiplications, additions, and load/stores
  - Cost of carry handling

More on multiplication complexity

- For small multiplication numbers, Karatsuba is typically not faster
- Cutoff between quadratic-complexity and Karatsuba depends on
  - Size of registers and radix used to represent big integers
  - Relative cost of multiplications, additions, and load/stores
  - Cost of carry handling
- Very rough rule of thumb: consider Karatsuba from $\approx$10 limbs

More on multiplication complexity

- For small multiplication numbers, Karatsuba is typically not faster
- Cutoff between quadratic-complexity and Karatsuba depends on
  - Size of registers and radix used to represent big integers
  - Relative cost of multiplications, additions, and load/stores
  - Cost of carry handling
- Very rough rule of thumb: consider Karatsuba from $\approx$10 limbs
- Lower complexity is also possible (for even larger inputs):
  - $\Theta(n^{\log_3 5})$ for Toom-3 multiplication
  - $\Theta(n^{\log_4 7})$ for Toom-4 multiplication
  - $\Theta(n \log n \log\log n))$ for Schönhage-Strassen
  - $\Theta(n \log n))$ for Harvey and van-der-Hoeven (2019)
- For cryptography, we care about Karatsuba and Toom, but nothing beyond

Modular reduction

- We don’t just need arithmetic on big integers
- We need arithmetic in finite fields

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

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 2^8 */
  stack u32[32] a;
```

- Integer A is obtained as $\sum_{i=0}^{31} a_i 2^{8i}$
- Lot of space in top of limbs to accumulate carries

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 2^8 */
  stack u32[32] a;
```

- Integer A is obtained as $\sum_{i=0}^{31} a_i 2^{8i}$
- Lot of space in top of limbs to accumulate carries
- Multiplication produces `stack u32[63] r`
- For "carried" inputs, each limb in `r` has at most 21 bits

Modular reduction

- Let's fix some p, say $p = 2^{255} - 19$

Modular reduction

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

Modular reduction

- Let's fix some p, say $p = 2^{255} - 19$
- We know that $2^{255} \equiv 19 \pmod{p}$
- This means that $2^{256} \equiv 38 \pmod{p}$
- Reduce 31-word intermediate result `r` as follows:

```
for i = 0 to 31 {
  u = r[i];
  t = r[i+32];
  t = 38 * t;
  u += t;
  r[i] = u;
}
```

Modular reduction

- Let's fix some p, say $p = 2^{255} - 19$
- We know that $2^{255} \equiv 19 \pmod{p}$
- This means that $2^{256} \equiv 38 \pmod{p}$
- Reduce 31-word intermediate result `r` as follows:

```
for i = 0 to 31 {
  u = r[i];
  t = r[i+32];
  t = 38 * t;
  u += t;
  r[i] = u;
}
```

- Result is in `r[0]`, ..., `r[31]`

Primes are not rabbits

- "You cannot just simply pull some nice prime out of your hat!"

Primes are not rabbits

- “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

Primes are not rabbits

- "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)
  - $2^{448} - 2^{224} - 1$ (Hamburg, 2015)

Primes are not rabbits

- "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)
  - $2^{448} - 2^{224} - 1$ (Hamburg, 2015)
- All these primes come with (more or less) fast reduction algorithms

How about other prime fields?

- What if somebody just throws an ugly prime at you?

How about other prime fields?

- What if somebody just throws an ugly prime at you?
- Example: German BSI is pushing the "Brainpool curves", over fields $\mathbb{F}_p$ with

$$\begin{aligned} p_{224} =& 2272162293245435278755253799591092807334073\backslash \\ & 2145944992304435472941311 \\ =& 0xD7C134AA264366862A18302575D1D787B09F07579\backslash \\ & 7DA89F57EC8C0FF \end{aligned}$$

or

$$\begin{aligned} p_{256} =& 7688495639704534422080974662900164909303795\backslash \\ & 0200943055203735601445031516197751 \\ =& 0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D\backslash \\ & 52620282013481D1F6E5377 \end{aligned}$$

How about other prime fields?

- What if somebody just throws an ugly prime at you?
- Example: German BSI is pushing the “Brainpool curves”, over fields $\mathbb{F}_p$ with

$$\begin{aligned} p_{224} = & 2272162293245435278755253799591092807334073\backslash \\ & 2145944992304435472941311 \\ = & 0xD7C134AA264366862A18302575D1D787B09F07579\backslash \\ & 7DA89F57EC8C0FF \end{aligned}$$

or

$$\begin{aligned} p_{256} = & 7688495639704534422080974662900164909303795\backslash \\ & 0200943055203735601445031516197751 \\ = & 0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D\backslash \\ & 52620282013481D1F6E5377 \end{aligned}$$

- 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 $t \mod p$

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 $t \mod p$
- Idea: Perform big-integer division with remainder (expensive!)

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 $t \mod p$
- 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 $aR \mod p$
  - Multiplication of aR and bR yields $t = abR^2$ ($2n$ limbs)
  - Now compute *Montgomery reduction*: $tR^{-1} \mod p$

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 $t \mod p$
- 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 $aR \mod p$
  - Multiplication of aR and bR yields $t = abR^2$ ($2n$ limbs)
  - Now compute *Montgomery reduction*: $tR^{-1} \mod p$
  - 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 from 0 to $n-1$ **do**
  $u \leftarrow a_i p' \mod b$
  $A \leftarrow A + u \cdot p \cdot b^i$
end for
$A \leftarrow A/b^n$
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

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 takes $n^2 + n$ multiplication instructions
- n of those are “shortened” multiplications (modulo b)

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 takes $n^2 + n$ multiplication instructions
- n of those are “shortened” multiplications (modulo b)
- The cost is roughly the same as schoolbook multiplication

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 takes $n^2 + n$ 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!)

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 takes $n^2 + n$ 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

Still missing: inversion

- Inversion is typically *much* more expensive than multiplication
- Efficient ECC arithmetic avoids frequent inversions
- ECC can typically not avoid *all* inversions
- We need inversion, but we do (usually) not need it often

Still missing: inversion

- Inversion is typically *much* more expensive than multiplication
- Efficient ECC arithmetic avoids frequent inversions
- ECC can typically not avoid *all* inversions
- We need inversion, but we do (usually) not need it often
- Two approaches to inversion:
    1. Extended Euclidean algorithm
    2. Fermat's little theorem

Extended Euclidean algorithm

- Given two integers a, b, the Extended Euclidean algorithm finds
  - The greatest common divisor of a and b
  - Integers u and v, such that $a \cdot u + b \cdot v = \gcd(a, b)$

Extended Euclidean algorithm

- Given two integers a, b, the Extended Euclidean algorithm finds
  - The greatest common divisor of a and b
  - Integers u and v, such that $a \cdot u + b \cdot v = \gcd(a, b)$
- It is based on the observation that

$$\gcd(a, b) = \gcd(b, a - qb) \quad \forall q \in \mathbb{Z}$$

Extended Euclidean algorithm

- Given two integers a, b, the Extended Euclidean algorithm finds
  - The greatest common divisor of a and b
  - Integers u and v, such that $a \cdot u + b \cdot v = \gcd(a, b)$
- It is based on the observation that

$$\gcd(a, b) = \gcd(b, a - qb) \quad \forall q \in \mathbb{Z}$$

- To compute $a^{-1} \pmod p$, use the algorithm to compute

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

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

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 ← 1
v ← 0
d ← a
v_1 ← 0
v_3 ← b
while (v_3 ≠ 0) do
    q ← ⌊d / v_3⌋
    t_3 ← d mod v_3
    t_1 ← u − q v_1
    u ← v_1
    d ← v_3
    v_1 ← t_1
    v_3 ← t_3
end while
v ← (d − a u) / 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

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!)

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

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
- Other option: constant-time EEA, Bernstein-Yang, 2019:
  `https://eprint.iacr.org/2019/266.pdf`

Fermat's little theorem

Theorem

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

Fermat's little theorem

Theorem

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

- This implies that $a^{p-2} \equiv a^{-1} \pmod{p}$
- Obvious algorithm for inversion: Exponentiation with $p-2$

Fermat's little theorem

Theorem

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

- This implies that $a^{p-2} \equiv a^{-1} \pmod{p}$
- Obvious algorithm for inversion: Exponentiation with $p - 2$
- The exponent is quite large (e.g., 255 bits), is that efficient?

Fermat's little theorem

Theorem

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

- This implies that $a^{p-2} \equiv a^{-1} \pmod p$
- 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 week)
  - Inversion modulo $2^{255}-19$ needs 254 squarings and 11 multiplications in $\mathbb{F}_{2^{255}-19}$

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


```
fn invert(reg ptr u32[N] r x) -> reg ptr u32[N] {
  stack u32[N] z2 z9 z11 z2_5_0 z2_10_0 z2_20_0 z2_50_0 z2_100_0 t;
  inline int i;

  /* 2 */              z2       = gfe_square(z2,x);
  /* 4 */              t        = gfe_square(t,z2);
  /* 8 */              t        = gfe_square_inline(t);
  /* 9 */              z9       = gfe_mul(z9,t,x);
  /* 11 */             z11      = gfe_mul(z11,z9,z2);
  /* 22 */             t        = gfe_square(t,z11);
  /* 2^5 - 2^0 = 31 */ z2_5_0   = gfe_mul(z2_5_0,t,z9);
  /* 2^6 - 2^1 */      t        = gfe_square(t,z2_5_0);
  /* 2^10 - 2^5 */     for i = 1 to 5 { t = gfe_square_inline(t); }
  /* 2^10 - 2^0 */     z2_10_0  = gfe_mul(z2_10_0,t,z2_5_0);
  /* 2^11 - 2^1 */     t        = gfe_square(t,z2_10_0);
  /* 2^20 - 2^10 */    for i = 1 to 10 { t = gfe_square_inline(t); }
  /* 2^20 - 2^0 */     z2_20_0  = gfe_mul(z2_20_0,t,z2_10_0);
  /* 2^21 - 2^1 */     t        = gfe_square(t,z2_20_0);
  /* 2^40 - 2^20 */    for i = 1 to 20 { t = gfe_square_inline(t); }
  /* 2^40 - 2^0 */     t        = gfe_mul_inline(t,z2_20_0);
```

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


```
  /* 2^41 - 2^1 */      t         = gfe_square_inline(t);
  /* 2^50 - 2^10 */     for i = 1 to 10 { t = gfe_square_inline(t); }
  /* 2^50 - 2^0 */      z2_50_0   = gfe_mul(z2_50_0,t,z2_10_0);
  /* 2^51 - 2^1 */      t         = gfe_square(t,z2_50_0);
  /* 2^100 - 2^50 */    for i = 1 to 50 { t = gfe_square_inline(t); }
  /* 2^100 - 2^0 */     z2_100_0 = gfe_mul(z2_100_0,t,z2_50_0);
  /* 2^101 - 2^1 */     t         = gfe_square(t,z2_100_0);
  /* 2^200 - 2^100 */   for i = 1 to 100 { t = gfe_square_inline(t); }
  /* 2^200 - 2^0 */     t         = gfe_mul_inline(t,z2_100_0);
  /* 2^201 - 2^1 */     t         = gfe_square_inline(t);
  /* 2^250 - 2^50 */    for i = 1 to 50 { t = gfe_square_inline(t); }
  /* 2^250 - 2^0 */     t         = gfe_mul_inline(t,z2_50_0);
  /* 2^251 - 2^1 */     t         = gfe_square_inline(t);
  /* 2^252 - 2^2 */     t         = gfe_square_inline(t);
  /* 2^253 - 2^3 */     t         = gfe_square_inline(t);
  /* 2^254 - 2^4 */     t         = gfe_square_inline(t);
  /* 2^255 - 2^5 */     t         = gfe_square_inline(t);
  /* 2^255 - 21 */      r         = gfe_mul(r,t,z11);

  return r;
}
```

While we're at it: square roots

- We can *compress* a point (x, y) before sending
- Usually send only x and one bit of y
- When receiving such a compressed point we need to solve recompute y as $\sqrt{x^3 + ax + b}$ (Weierstrass curve)
- Similar for twisted Edwards curves (see `https://cryptojedi.org/papers/#ed25519`)

While we're at it: square roots

- We can *compress* a point (x, y) before sending
- Usually send only x and one bit of y
- When receiving such a compressed point we need to solve recompute y as $\sqrt{x^3 + ax + b}$ (Weierstrass curve)
- Similar for twisted Edwards curves (see `https://cryptojedi.org/papers/#ed25519`)
- If $p \equiv 3 \pmod 4$: compute square root of a as $a^{(p+1)/4}$

While we're at it: square roots

- We can *compress* a point (x, y) before sending
- Usually send only x and one bit of y
- When receiving such a compressed point we need to solve recompute y as $\sqrt{x^3 + ax + b}$ (Weierstrass curve)
- Similar for twisted Edwards curves (see https://cryptojedi.org/papers/#ed25519)
- If $p \equiv 3 \pmod 4$: compute square root of a as $a^{(p+1)/4}$
- If $p \equiv 5 \pmod 8$: compute β, such that $\beta^4 = a^2$ as $a^{(p+3)/8}$
- If $\beta^2 = -a$: multiply by $\sqrt{-1}$

While we're at it: square roots

- We can *compress* a point (x, y) before sending
- Usually send only x and one bit of y
- When receiving such a compressed point we need to solve recompute y as $\sqrt{x^3 + ax + b}$ (Weierstrass curve)
- Similar for twisted Edwards curves (see https://cryptojedi.org/papers/#ed25519)
- If $p \equiv 3 \pmod 4$: compute square root of a as $a^{(p+1)/4}$
- If $p \equiv 5 \pmod 8$: compute β, such that $\beta^4 = a^2$ as $a^{(p+3)/8}$
- If $\beta^2 = -a$: multiply by $\sqrt{-1}$
- Computing square roots is (typically) about as expensive as an inversion

Summary

- Multiprecision integers are represented as tuples of smaller integers
- Different representations possible
  - Saturated representation often most efficient
  - Unsaturated representation have easier carry handling

Summary

- Multiprecision integers are represented as tuples of smaller integers
- Different representations possible
  - Saturated representation often most efficient
  - Unsaturated representation have easier carry handling
- Multiprecision arithmetic is similar to polynomial arithmetic
- Difference is carries

Summary

- Multiprecision integers are represented as tuples of smaller integers
- Different representations possible
  - Saturated representation often most efficient
  - Unsaturated representation have easier carry handling
- Multiprecision arithmetic is similar to polynomial arithmetic
- Difference is carries
- For ECC, dominating cost is typically multiplications
  - Different approaches with quadratic complexity
  - Karatsuba (or Toom) may be worth considering

Summary

- Multiprecision integers are represented as tuples of smaller integers
- Different representations possible
  - Saturated representation often most efficient
  - Unsaturated representation have easier carry handling
- Multiprecision arithmetic is similar to polynomial arithmetic
- Difference is carries
- For ECC, dominating cost is typically multiplications
  - Different approaches with quadratic complexity
  - Karatsuba (or Toom) may be worth considering
- Modular reduction often for special primes
- For General primes typically use Montgomery reduction

Summary

- Multiprecision integers are represented as tuples of smaller integers
- Different representations possible
  - Saturated representation often most efficient
  - Unsaturated representation have easier carry handling
- Multiprecision arithmetic is similar to polynomial arithmetic
- Difference is carries
- For ECC, dominating cost is typically multiplications
  - Different approaches with quadratic complexity
  - Karatsuba (or Toom) may be worth considering
- Modular reduction often for special primes
- For General primes typically use Montgomery reduction
- Two main options for inversion:
  - Extended Euclidean algorithm (careful about timing attacks!)
  - Fermat's little theorem (less efficient, but trivially constant-time)