Finite field arithmetic

Peter Schwabe

Radboud University Nijmegen, The Netherlands

September 11, 2013

ECC 2013 Summer School

Elliptic-curve addition

- Computing $P + Q$ for two elliptic-curve points P and Q means performing a few operations in the underlying field
- Example: Add projective $(X_P : Y_P : Z_P)$ and $(X_Q : Y_Q : Z_Q)$ on curve $E : y^2 = x^3 + ax + b$.

  $t_1 \leftarrow Y_P \cdot Z_Q$
  $t_2 \leftarrow X_P \cdot Z_Q$
  $t_3 \leftarrow Z_P \cdot Z_Q$
  $u \leftarrow Y_Q \cdot Z_P - t_1$
  $uu \leftarrow u^2$
  $v \leftarrow X_Q \cdot Z_P - t_2$
  $vv \leftarrow v^2$
  $vvv \leftarrow v \cdot vv$
  $R \leftarrow vv \cdot t_2$
  $A \leftarrow uu \cdot t_3 - vvv - 2 \cdot R$
  $X_R \leftarrow v \cdot A$
  $Y_R \leftarrow u \cdot (R - A) - vvv \cdot t_1$
  $Z_R \leftarrow vvv \cdot t_3$
  return $(X_R : Y_R : Z_R)$

The EFD

- There are many formulas for different curve shapes and point representations
- Best overview: The **Explicit Formulas Database (EFD)**:

`http://www.hyperelliptic.org/EFD/`

- Compiled from many papers and talks by Dan Bernstein and Tanja Lange
- Contains verification scripts, 3-operand code, ...

The problem with large integers

- C has data types for 8-bit, 16-bit, 32-bit, and 64-bit integers
- Why are there no data types for 256-bit integers?
  - Magma does not have problems with large integers
  - Python has datatype `long` for arbitrary-size integers
  - Java has `BigInteger` class

The problem with large integers

- C has data types for 8-bit, 16-bit, 32-bit, and 64-bit integers
- Why are there no data types for 256-bit integers?
  - Magma does not have problems with large integers
  - Python has datatype `long` for arbitrary-size integers
  - Java has `BigInteger` class
- C is "portable assembly", very close to what computers really do
- Computers work on data in *registers* (very small, very fast storage units)

The problem with large integers

- C has data types for 8-bit, 16-bit, 32-bit, and 64-bit integers
- Why are there no data types for 256-bit integers?
  - Magma does not have problems with large integers
  - Python has datatype `long` for arbitrary-size integers
  - Java has `BigInteger` class
- C is "portable assembly", very close to what computers really do
- Computers work on data in *registers* (very small, very fast storage units)
- Typical register sizes: 8 bit, 16 bit, 32 bit, 64 bit $\dots$ but not 256 bit

The problem with large integers

- C has data types for 8-bit, 16-bit, 32-bit, and 64-bit integers
- Why are there no data types for 256-bit integers?
  - Magma does not have problems with large integers
  - Python has datatype `long` for arbitrary-size integers
  - Java has `BigInteger` class
- C is "portable assembly", very close to what computers really do
- Computers work on data in *registers* (very small, very fast storage units)
- Typical register sizes: 8 bit, 16 bit, 32 bit, 64 bit $\dots$ but not 256 bit
- **That's a lie!**

The problem with large integers

- C has data types for 8-bit, 16-bit, 32-bit, and 64-bit integers
- Why are there no data types for 256-bit integers?
  - Magma does not have problems with large integers
  - Python has datatype `long` for arbitrary-size integers
  - Java has `BigInteger` class
- C is "portable assembly", very close to what computers really do
- Computers work on data in *registers* (very small, very fast storage units)
- Typical register sizes: 8 bit, 16 bit, 32 bit, 64 bit $\dots$ but not 256 bit
- **That's a lie!**
- Yeah, you're right. We *do* have 256-bit registers (AVX on Intel and AMD processors)
- But those do not hold a single 256-bit integer (but vectors of integers or floats)

The problem with large integers

- C has data types for 8-bit, 16-bit, 32-bit, and 64-bit integers
- Why are there no data types for 256-bit integers?
  - Magma does not have problems with large integers
  - Python has datatype `long` for arbitrary-size integers
  - Java has `BigInteger` class
- C is "portable assembly", very close to what computers really do
- Computers work on data in *registers* (very small, very fast storage units)
- Typical register sizes: 8 bit, 16 bit, 32 bit, 64 bit $\dots$ but not 256 bit
- **That's a lie!**
- Yeah, you're right. We *do* have 256-bit registers (AVX on Intel and AMD processors)
- But those do not hold a single 256-bit integer (but vectors of integers or floats)
- Why can't they just hold a 256-bit integer?

The problem with large integers

- C has data types for 8-bit, 16-bit, 32-bit, and 64-bit integers
- Why are there no data types for 256-bit integers?
  - Magma does not have problems with large integers
  - Python has datatype `long` for arbitrary-size integers
  - Java has `BigInteger` class
- C is "portable assembly", very close to what computers really do
- Computers work on data in *registers* (very small, very fast storage units)
- Typical register sizes: 8 bit, 16 bit, 32 bit, 64 bit $\dots$ but not 256 bit
- **That's a lie!**
- Yeah, you're right. We *do* have 256-bit registers (AVX on Intel and AMD processors)
- But those do not hold a single 256-bit integer (but vectors of integers or floats)
- Why can't they just hold a 256-bit integer?
- Because arithmetic units cannot perform arithmetic on 256-bit integers (only on 8-bit, 16-bit, 32-bit, and 64-bit integers)

So, what do we have?

- Consider the processor in my laptop here (Intel Core i7, Ivy Bridge)

So, what do we have?

- Consider the processor in my laptop here (Intel Core i7, Ivy Bridge)
- Addition, subtraction and multiplication of 64-bit integers
- Multiplication produces a 128-bit result in 2 registers

So, what do we have?

- Consider the processor in my laptop here (Intel Core i7, Ivy Bridge)
- Addition, subtraction and multiplication of 64-bit integers
- Multiplication produces a 128-bit result in 2 registers
- Addition, subtraction and multiplication of smaller integers (less interesting)

So, what do we have?

- Consider the processor in my laptop here (Intel Core i7, Ivy Bridge)
- Addition, subtraction and multiplication of 64-bit integers
- Multiplication produces a 128-bit result in 2 registers
- Addition, subtraction and multiplication of smaller integers (less interesting)
- Single-precision and double-precision floating-point arithmetic

So, what do we have?

- Consider the processor in my laptop here (Intel Core i7, Ivy Bridge)
- Addition, subtraction and multiplication of 64-bit integers
- Multiplication produces a 128-bit result in 2 registers
- Addition, subtraction and multiplication of smaller integers (less interesting)
- Single-precision and double-precision floating-point arithmetic
- Arithmetic on vectors of 2 64-bit integers
- Integer-vector multiplication only produces 2 64-bit results

So, what do we have?

- Consider the processor in my laptop here (Intel Core i7, Ivy Bridge)
- Addition, subtraction and multiplication of 64-bit integers
- Multiplication produces a 128-bit result in 2 registers
- Addition, subtraction and multiplication of smaller integers (less interesting)
- Single-precision and double-precision floating-point arithmetic
- Arithmetic on vectors of 2 64-bit integers
- Integer-vector multiplication only produces 2 64-bit results
- Arithmetic on vectors of 4 double-precision floats

What do we need?

- For this talk consider arithmetic in a field $\mathbb{F}_p$ of large prime order p (for example 256-bit long)

What do we need?

- For this talk consider arithmetic in a field $\mathbb{F}_p$ of large prime order p (for example 256-bit long)
- Addition of ≈ 256-bit integers
- Subtraction of ≈ 256-bit integers

What do we need?

- For this talk consider arithmetic in a field $\mathbb{F}_p$ of large prime order p (for example 256-bit long)
- Addition of ≈ 256-bit integers
- Subtraction of ≈ 256-bit integers
- Reduction modulo p after addition and subtraction

What do we need?

- For this talk consider arithmetic in a field $\mathbb{F}_p$ of large prime order p (for example 256-bit long)
- Addition of ≈ 256-bit integers
- Subtraction of ≈ 256-bit integers
- Reduction modulo p after addition and subtraction
- Multiplication of ≈ 256-bit integers
- Squaring of ≈ 256-bit integers

What do we need?

- For this talk consider arithmetic in a field $\mathbb{F}_p$ of large prime order p (for example 256-bit long)
- Addition of ≈ 256-bit integers
- Subtraction of ≈ 256-bit integers
- Reduction modulo p after addition and subtraction
- Multiplication of ≈ 256-bit integers
- Squaring of ≈ 256-bit integers
- Reduction of a ≈ 512-bit multiplication result modulo p

What do we need?

- For this talk consider arithmetic in a field $\mathbb{F}_p$ of large prime order p (for example 256-bit long)
- Addition of ≈ 256-bit integers
- Subtraction of ≈ 256-bit integers
- Reduction modulo p after addition and subtraction
- Multiplication of ≈ 256-bit integers
- Squaring of ≈ 256-bit integers
- Reduction of a ≈ 512-bit multiplication result modulo p
- Inversion modulo p

Representing 256-bit integers

- Let's start with 64-bit integers, that seems easiest
- Represent 256-bit integer A through 4 64-bit integers a_0, a_1, a_2, a_3 (a total of 256 bits)

Representing 256-bit integers

- Let's start with 64-bit integers, that seems easiest
- Represent 256-bit integer A through 4 64-bit integers a_0, a_1, a_2, a_3 (a total of 256 bits)
- Value of A is $\sum_{i=0}^{3} a_i 2^{64 \cdot i}$

Representing 256-bit integers

- Let's start with 64-bit integers, that seems easiest
- Represent 256-bit integer A through 4 64-bit integers a_0, a_1, a_2, a_3 (a total of 256 bits)
- Value of A is $\sum_{i=0}^{3} a_i 2^{64 \cdot i}$
- This is called *radix*-2^{64} representation

Representing 256-bit integers

- Let's start with 64-bit integers, that seems easiest
- Represent 256-bit integer A through 4 64-bit integers a_0, a_1, a_2, a_3 (a total of 256 bits)
- Value of A is $\sum_{i=0}^{3} a_i 2^{64 \cdot i}$
- This is called *radix*-2^{64} representation
- Let's write that in C code:

```
typedef struct{
  unsigned long long a[4];
}  bigint256;
```

Addition of two `bigint256`

```
void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- What's wrong about this?

Addition of two `bigint256`

```
void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- What's wrong about this?

Addition of two `bigint256`

```
void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- What's wrong about this?
- This performs arithmetic on a vector of 4 independent 64-bit integers (modulo 2^{64})

Addition of two `bigint256`

```
void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- What's wrong about this?
- This performs arithmetic on a vector of 4 independent 64-bit integers (modulo 2^{64})
- This is *not* the same as arithmetic on 256-bit integers

Addition of two `bigint256`

```
void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- What's wrong about this?
- This performs arithmetic on a vector of 4 independent 64-bit integers (modulo 2^{64})
- This is *not* the same as arithmetic on 256-bit integers
- `x->a[0] + y->a[0]` may have 65 bits
- Need to put low 64 bits into `r.a[0]` and add *carry bit* into `r.a[1]`
- Same for all subsequent additions

Addition of two bigint256

```
void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- What's wrong about this?
- This performs arithmetic on a vector of 4 independent 64-bit integers (modulo 2^{64})
- This is *not* the same as arithmetic on 256-bit integers
- `x->a[0] + y->a[0]` may have 65 bits
- Need to put low 64 bits into `r.a[0]` and add *carry bit* into `r.a[1]`
- Same for all subsequent additions
- Note: The result may not even fit into a `bigint256`!

How do we get the carry bits?

- in C something like:

```
unsigned long long carry = 0;
if(r.a[0] < x.a[0]) carry = 1;
```

How do we get the carry bits?

- in C something like:

```
unsigned long long carry = 0;
if(r.a[0] < x.a[0]) carry = 1;
```

- The computer actually remembers the carry in a *flag register*
- We can use this carry flag when using assembly
- No direct access from C level (so much for "portable assembly")

How do we get the carry bits?

- in C something like:

```
unsigned long long carry = 0;
if(r.a[0] < x.a[0]) carry = 1;
```

- The computer actually remembers the carry in a *flag register*
- We can use this carry flag when using assembly
- No direct access from C level (so much for “portable assembly”)
- So, let’s do it in assembly (no worries, it’s not dark arts)

How do we get the carry bits?

- in C something like:

```
unsigned long long carry = 0;
if(r.a[0] < x.a[0]) carry = 1;
```

- The computer actually remembers the carry in a *flag register*
- We can use this carry flag when using assembly
- No direct access from C level (so much for "portable assembly")
- So, let's do it in assembly (no worries, it's not dark arts)
- Use somewhat simplified "C-like" `qhasm` syntax for assembly

bigint256 addition in qhasm

```
int64 x
int64 y

enter bigint256_add

x = mem64[input_1 + 0]
y = mem64[input_2 + 0]
carry? x += y
mem64[input_0 + 0] = x

x = mem64[input_1 + 8]
y = mem64[input_2 + 8]
carry? x += y + carry
mem64[input_0 + 8] = x

x = mem64[input_1 + 16]
y = mem64[input_2 + 16]
carry? x += y + carry
mem64[input_0 + 16] = x

x = mem64[input_1 + 24]
y = mem64[input_2 + 24]
carry? x += y + carry
mem64[input_0 + 24] = x

x = 0
x += x + carry

return x
```

bigint256 subtraction in qhasm

```
int64 x
int64 y

enter bigint256_sub

x = mem64[input_1 + 0]
y = mem64[input_2 + 0]
carry? x -= y
mem64[input_0 + 0] = x

x = mem64[input_1 + 8]
y = mem64[input_2 + 8]
carry? x -= y - carry
mem64[input_0 + 8] = x

x = mem64[input_1 + 16]
y = mem64[input_2 + 16]
carry? x -= y - carry
mem64[input_0 + 16] = x

x = mem64[input_1 + 24]
y = mem64[input_2 + 24]
carry? x -= y - carry
mem64[input_0 + 24] = x

x = 0
x += x + carry

return x
```

One step back. . .

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

One step back...

- Radix-2^{64} representation works and is sometimes a good choice
- 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!)

One step back. . .

- Radix-2^{64} representation works and is sometimes a good choice
- 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)

One step back...

- Radix-2^{64} representation works and is sometimes a good choice
- 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)
- Let's get rid of the carries, represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

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

- This is called radix-2^{51} representation

One step back. . .

- Radix-2^{64} representation works and is sometimes a good choice
- 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)
- Let's get rid of the carries, represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

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

- This is called radix-2^{51} representation
- Multiple ways to write the same integer A, for example $A = 2^{52}$:
  - $(2^{52}, 0, 0, 0, 0)$
  - $(0, 2, 0, 0, 0)$

One step back...

- Radix-2^{64} representation works and is sometimes a good choice
- 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)
- Let's get rid of the carries, represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

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

- 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 $a_i \in [0, \dots, 2^{52} - 1]$

Addition of two bigint256

```
typedef struct{
  unsigned long long a[5];
}  bigint256;

void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

Addition of two bigint256

```
typedef struct{
  unsigned long long a[5];
} bigint256;

void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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

Addition of two bigint256

```
typedef struct{
  unsigned long long a[5];
}  bigint256;

void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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
- This actually works as long as all coefficients are in $[0, \ldots, 2^{63} - 1]$

Addition of two `bigint256`

```
typedef struct{
  unsigned long long a[5];
}  bigint256;

void bigint256_add(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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
- This actually works as long as all coefficients are in $[0, \dots, 2^{63} - 1]$
- We can do quite a few additions before we have to carry (reduce)

Subtraction of two bigint256

```
typedef struct{
  unsigned long long a[5];
}  bigint256;

void bigint256_sub(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- Again: what's wrong here?

Subtraction of two bigint256

```
typedef struct{
  signed long long a[5];
}  bigint256;

void bigint256_sub(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- Again: what's wrong here?
- Slightly update our `bigint256` definition to work with *signed* 64-bit integers

Subtraction of two `bigint256`

```
typedef struct{
  signed long long a[5];
}  bigint256;

void bigint256_sub(bigint256 *r,
                   const bigint256 *x,
                   const bigint256 *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];
}
```

- Again: what's wrong here?
- Slightly update our `bigint256` definition to work with *signed* 64-bit integers
- Reduced if coefficients are in $[-2^{52}-1, 2^{52}-1]$

Back to reduced representation

- An addition/subtraction does not produce a reduced output for reduced inputs
- Can do quite a few additions, but at some point we need to reduce (i.e., *carry*)

Back to reduced representation

- An addition/subtraction does not produce a reduced output for reduced inputs
- Can do quite a few additions, but at some point we need to reduce (i.e., *carry*)
- Let's carry high bits of `r.a[0]` over to `r.a[1]`:

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

Back to reduced representation

- An addition/subtraction does not produce a reduced output for reduced inputs
- Can do quite a few additions, but at some point we need to reduce (i.e., *carry*)
- Let's carry high bits of `r.a[0]` over to `r.a[1]`:

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

- This requires that `>> 51` is an arithmetic shift (i.e., truncating division by 2^{51})
- Not defined in C standard (usually works, and no problem in assembly)

Back to reduced representation

- An addition/subtraction does not produce a reduced output for reduced inputs
- Can do quite a few additions, but at some point we need to reduce (i.e., *carry*)
- Let's carry high bits of `r.a[0]` over to `r.a[1]`:

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

- This requires that `>> 51` is an arithmetic shift (i.e., truncating division by 2^{51})
- Not defined in C standard (usually works, and no problem in assembly)
- Proceed:
  - Carry from `r.a[1]` to `r.a[2]`;

Back to reduced representation

- An addition/subtraction does not produce a reduced output for reduced inputs
- Can do quite a few additions, but at some point we need to reduce (i.e., *carry*)
- Let's carry high bits of `r.a[0]` over to `r.a[1]`:

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

- This requires that `>> 51` is an arithmetic shift (i.e., truncating division by 2^{51})
- Not defined in C standard (usually works, and no problem in assembly)
- Proceed:
  - Carry from `r.a[1]` to `r.a[2]`;
  - Carry from `r.a[2]` to `r.a[3]`;

Back to reduced representation

- An addition/subtraction does not produce a reduced output for reduced inputs
- Can do quite a few additions, but at some point we need to reduce (i.e., *carry*)
- Let's carry high bits of `r.a[0]` over to `r.a[1]`:

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

- This requires that `>> 51` is an arithmetic shift (i.e., truncating division by 2^{51})
- Not defined in C standard (usually works, and no problem in assembly)
- Proceed:
  - Carry from `r.a[1]` to `r.a[2]`;
  - Carry from `r.a[2]` to `r.a[3]`;
  - Carry from `r.a[3]` to `r.a[4]`;

Back to reduced representation

- An addition/subtraction does not produce a reduced output for reduced inputs
- Can do quite a few additions, but at some point we need to reduce (i.e., *carry*)
- Let's carry high bits of `r.a[0]` over to `r.a[1]`:

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

- This requires that `>> 51` is an arithmetic shift (i.e., truncating division by 2^{51})
- Not defined in C standard (usually works, and no problem in assembly)
- Proceed:
  - Carry from `r.a[1]` to `r.a[2]`;
  - Carry from `r.a[2]` to `r.a[3]`;
  - Carry from `r.a[3]` to `r.a[4]`;
  - Carry from `r.a[4]` to ...?

Reducing modulo p

- When adding integers, the result naturally grows
- For integers, we do not really have any place to carry from `r.a[4]`, except create a new limb `r.a[5]`, etc.

Reducing modulo p

- When adding integers, the result naturally grows
- For integers, we do not really have any place to carry from `r.a[4]`, except create a new limb `r.a[5]`, etc.
- We want to perform arithmetic in a field $\mathbb{F}_p$, we can reduce modulo p

Reducing modulo p

- When adding integers, the result naturally grows
- For integers, we do not really have any place to carry from `r.a[4]`, except create a new limb `r.a[5]`, etc.
- We want to perform arithmetic in a field $\mathbb{F}_p$, we can reduce modulo p
- Let's fix some p, say $p = 2^{255} - 19$

Reducing modulo p

- When adding integers, the result naturally grows
- For integers, we do not really have any place to carry from `r.a[4]`, except create a new limb `r.a[5]`, etc.
- We want to perform arithmetic in a field $\mathbb{F}_p$, we can reduce modulo p
- Let's fix some p, say $p = 2^{255} - 19$
- Imagine, that we *did* carry to `r.a[5]`. Then we get an integer

$$A = a_0 + 2^{51}a_1 + 2^{102}a_2 + 2^{153}a_3 + 2^{204}a_4 + 2^{255}a_5$$

Reducing modulo p

- When adding integers, the result naturally grows
- For integers, we do not really have any place to carry from `r.a[4]`, except create a new limb `r.a[5]`, etc.
- We want to perform arithmetic in a field $\mathbb{F}_p$, we can reduce modulo p
- Let's fix some p, say $p = 2^{255} - 19$
- Imagine, that we *did* carry to `r.a[5]`. Then we get an integer

$$A = a_0 + 2^{51}a_1 + 2^{102}a_2 + 2^{153}a_3 + 2^{204}a_4 + 2^{255}a_5$$

- Note that $2^{255} \equiv 19 \pmod{p}$
- Modulo p, the integer A is congruent to

$$A = (a_0 + 19a_5) + 2^{51}a_1 + 2^{102}a_2 + 2^{153}a_3 + 2^{204}a_4$$

Reducing modulo p

- When adding integers, the result naturally grows
- For integers, we do not really have any place to carry from `r.a[4]`, except create a new limb `r.a[5]`, etc.
- We want to perform arithmetic in a field $\mathbb{F}_p$, we can reduce modulo p
- Let's fix some p, say $p = 2^{255} - 19$
- Imagine, that we *did* carry to `r.a[5]`. Then we get an integer

$$A = a_0 + 2^{51}a_1 + 2^{102}a_2 + 2^{153}a_3 + 2^{204}a_4 + 2^{255}a_5$$

- Note that $2^{255} \equiv 19 \pmod{p}$
- Modulo p, the integer A is congruent to

$$A = (a_0 + 19a_5) + 2^{51}a_1 + 2^{102}a_2 + 2^{153}a_3 + 2^{204}a_4$$

- We can reduce `r.a[4]` as follows (modulo p):

```
signed long long carry = r.a[4] >> 51;
r.a[0] += 19*carry;
carry <<= 51;
r.a[4] -= carry;
```

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-P_{192}", FIPS186-2, 2000)
  - $2^{224} - 2^{96} + 1$ ("NIST-P_{224}", FIPS186-2, 2000)
  - $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ ("NIST-P_{256}", FIPS186-2, 2000)
  - $2^{255} - 19$ (Bernstein, 2006)
  - $2^{251} - 9$ (Bernstein, Hamburg, Krasnova, Lange, 2013)

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-P$_{192}$", FIPS186-2, 2000)
  - $2^{224} - 2^{96} + 1$ ("NIST-P$_{224}$", FIPS186-2, 2000)
  - $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ ("NIST-P$_{256}$", FIPS186-2, 2000)
  - $2^{255} - 19$ (Bernstein, 2006)
  - $2^{251} - 9$ (Bernstein, Hamburg, Krasnova, Lange, 2013)
- All these primes come with (more or less) fast reduction algorithms

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-P$_{192}$", FIPS186-2, 2000)
  - $2^{224} - 2^{96} + 1$ ("NIST-P$_{224}$", FIPS186-2, 2000)
  - $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ ("NIST-P$_{256}$", FIPS186-2, 2000)
  - $2^{255} - 19$ (Bernstein, 2006)
  - $2^{251} - 9$ (Bernstein, Hamburg, Krasnova, Lange, 2013)
- All these primes come with (more or less) fast reduction algorithms
- More about *general primes* later
- For the moment let's stick to $2^{255} - 19$

Briefly back to carrying

- We first reduced `r.a[0]`, i.e., produced `r.a[0]` in interval $[-2^{51}, 2^{51}]$
- At the end we add `19*carry` to `r.a[0]`
- Carry has at most 12 bits (obtained by dividing a signed 64-bit integer by 2^{51})
- The absolute value of `19*carry` has at most 17 bits
- `r.a[0]+19*carry` is still within $[-2^{52}-1, 2^{52}-1]$, i.e., reduced

Multiplication

- We want to multiply two integers
  $A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$ and $B = \sum_{i=0}^{4} b_i 2^{51 \cdot i}$

Multiplication

- We want to multiply two integers $A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$ and $B = \sum_{i=0}^{4} b_i 2^{51 \cdot i}$
- Think about it like this:
  - Multiply polynomials $A = \sum_{i=0}^{4} a_i X^i$ and $B = \sum_{i=0}^{4} b_i X^i$

Multiplication

- We want to multiply two integers $A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$ and $B = \sum_{i=0}^{4} b_i 2^{51 \cdot i}$
- Think about it like this:
  - Multiply polynomials $A = \sum_{i=0}^{4} a_i X^i$ and $B = \sum_{i=0}^{4} b_i X^i$
  - Obtain result polynomial $R = \sum_{i=0}^{8} r_i X^i$

Multiplication

- We want to multiply two integers
  $A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$ and $B = \sum_{i=0}^{4} b_i 2^{51 \cdot i}$
- Think about it like this:
  - Multiply polynomials $A = \sum_{i=0}^{4} a_i X^i$ and $B = \sum_{i=0}^{4} b_i X^i$
  - Obtain result polynomial $R = \sum_{i=0}^{8} r_i X^i$
  - Evaluate R at 2^{51}

Multiplication

- We want to multiply two integers $A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$ and $B = \sum_{i=0}^{4} b_i 2^{51 \cdot i}$
- Think about it like this:
  - Multiply polynomials $A = \sum_{i=0}^{4} a_i X^i$ and $B = \sum_{i=0}^{4} b_i X^i$
  - Obtain result polynomial $R = \sum_{i=0}^{8} r_i X^i$
  - Evaluate R at 2^{51}
- The coefficients of R are:

$$
\begin{aligned}
r_0 &= a_0 b_0 \\
r_1 &= a_0 b_1 + a_1 b_0 \\
r_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
&\dots \\
r_8 &= a_4 b_4
\end{aligned}
$$

Multiplication

- We want to multiply two integers $A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$ and $B = \sum_{i=0}^{4} b_i 2^{51 \cdot i}$
- Think about it like this:
  - Multiply polynomials $A = \sum_{i=0}^{4} a_i X^i$ and $B = \sum_{i=0}^{4} b_i X^i$
  - Obtain result polynomial $R = \sum_{i=0}^{8} r_i X^i$
  - Evaluate R at 2^{51}
- The coefficients of R are:

$$
\begin{aligned}
r_0 &= a_0 b_0 \\
r_1 &= a_0 b_1 + a_1 b_0 \\
r_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
&\dots \\
r_8 &= a_4 b_4
\end{aligned}
$$

- If all a_i and b_i have 52 bits, the r_i will have up to 107 bits
- Doesn't fit into 64-bit registers, but remember that there is a multiplication instruction that produces 128-bit results in two registers.

Multiplication in C (idealized)

```
void mul(int128 r[9], const bigint256 *x, const bigint256 *y)
{
  const signed long long *a = x->a;
  const signed long long *b = y->a;
  r[0] = a[0]*b[0];
  r[1] = a[0]*b[1] + a[1]*b[0];
  r[2] = a[0]*b[2] + a[1]*b[1] + a[2]*b[0];
  r[3] = a[0]*b[3] + a[1]*b[2] + a[2]*b[1] + a[3]*b[0];
  r[4] = a[0]*b[4] + a[1]*b[3] + a[2]*b[2] + a[3]*b[1] + a[4]*b[0];
  r[5] = a[1]*b[4] + a[2]*b[3] + a[3]*b[2] + a[4]*b[1];
  r[6] = a[2]*b[4] + a[3]*b[3] + a[4]*b[2];
  r[7] = a[3]*b[4] + a[4]*b[3];
  r[8] = a[4]*b[4];
}
```

Multiplication in C (idealized)

```
void mul(int128 r[9], const bigint256 *x, const bigint256 *y)
{
  const signed long long *a = x->a;
  const signed long long *b = y->a;
  r[0] = a[0]*b[0];
  r[1] = a[0]*b[1] + a[1]*b[0];
  r[2] = a[0]*b[2] + a[1]*b[1] + a[2]*b[0];
  r[3] = a[0]*b[3] + a[1]*b[2] + a[2]*b[1] + a[3]*b[0];
  r[4] = a[0]*b[4] + a[1]*b[3] + a[2]*b[2] + a[3]*b[1] + a[4]*b[0];
  r[5] = a[1]*b[4] + a[2]*b[3] + a[3]*b[2] + a[4]*b[1];
  r[6] = a[2]*b[4] + a[3]*b[3] + a[4]*b[2];
  r[7] = a[3]*b[4] + a[4]*b[3];
  r[8] = a[4]*b[4];
}
```

- Can evaluate in arbitrary order:
  "operand scanning" vs. "product scanning"

Multiplication in C (idealized)

```
void mul(int128 r[9], const bigint256 *x, const bigint256 *y)
{
  const signed long long *a = x->a;
  const signed long long *b = y->a;
  r[0] = a[0]*b[0];
  r[1] = a[0]*b[1] + a[1]*b[0];
  r[2] = a[0]*b[2] + a[1]*b[1] + a[2]*b[0];
  r[3] = a[0]*b[3] + a[1]*b[2] + a[2]*b[1] + a[3]*b[0];
  r[4] = a[0]*b[4] + a[1]*b[3] + a[2]*b[2] + a[3]*b[1] + a[4]*b[0];
  r[5] = a[1]*b[4] + a[2]*b[3] + a[3]*b[2] + a[4]*b[1];
  r[6] = a[2]*b[4] + a[3]*b[3] + a[4]*b[2];
  r[7] = a[3]*b[4] + a[4]*b[3];
  r[8] = a[4]*b[4];
}
```

- Can evaluate in arbitrary order:
  "operand scanning" vs. "product scanning"
- This doesn't work because we don't have `int128` data type
- Even in assembly, we don't have addition of 128-bit integers

A peek at multiplication in qhasm

```
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 0]
r0 = rax
r0h = rdx
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 8]
r1 = rax
r1h = rdx
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 16]
r2 = rax
r2h = rdx
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 24]
r3 = rax
r3h = rdx
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 32]
r4 = rax
r4h = rdx
```

A peek at multiplication in qhasm

```
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 0]
carry? r1 += rax
r1h += rdx + carry
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 8]
carry? r2 += rax
r2h += rdx + carry
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 16]
carry? r3 += rax
r3h += rdx + carry
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 24]
carry? r4 += rax
r4h += rdx + carry
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 32]
r5 = rax
r5h = rdx
```

A peek at multiplication in qhasm

```
...

mem64[input_0 +  0] = r0
mem64[input_0 +  8] = r0h
mem64[input_0 + 16] = r1
mem64[input_0 + 24] = r1h
mem64[input_0 + 32] = r2
mem64[input_0 + 40] = r2h

...

mem64[input_0 + 128] = r8
mem64[input_0 + 136] = r8h
```

Again: back to reduced representation

- We now have $r_0, \dots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \dots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51\cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51\cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51\cdot i}\right) \pmod{2^{255}-19}$$

Again: back to reduced representation

- We now have $r_0, \dots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \dots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51 \cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51 \cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51 \cdot i}\right) \pmod{2^{255} - 19}$$

- With the same reasoning as before, we can reduce modulo p as
  $r_0 \leftarrow r_0 + 19 r_5$

Again: back to reduced representation

- We now have $r_0, \ldots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \ldots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51\cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51\cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51\cdot i}\right) \pmod{2^{255}-19}$$

- With the same reasoning as before, we can reduce modulo p as

$$\begin{aligned} r_0 &\leftarrow r_0 + 19r_5 \\ r_1 &\leftarrow r_1 + 19r_6 \\ r_2 &\leftarrow r_2 + 19r_7 \\ r_3 &\leftarrow r_3 + 19r_8 \end{aligned}$$

Again: back to reduced representation

- We now have $r_0, \dots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \dots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51\cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51\cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51\cdot i}\right) \pmod{2^{255}-19}$$

- With the same reasoning as before, we can reduce modulo p as

$$\begin{aligned} r_0 &\leftarrow r_0 + 19r_5 \\ r_1 &\leftarrow r_1 + 19r_6 \\ r_2 &\leftarrow r_2 + 19r_7 \\ r_3 &\leftarrow r_3 + 19r_8 \end{aligned}$$

- Remaining problem: $r_0, \dots, r_4$ are too large

Again: back to reduced representation

- We now have $r_0, \dots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \dots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51\cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51\cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51\cdot i}\right) \pmod{2^{255}-19}$$

- With the same reasoning as before, we can reduce modulo p as

$$\begin{aligned}
r_0 &\leftarrow r_0 + 19r_5 \\
r_1 &\leftarrow r_1 + 19r_6 \\
r_2 &\leftarrow r_2 + 19r_7 \\
r_3 &\leftarrow r_3 + 19r_8
\end{aligned}$$

- Remaining problem: $r_0, \dots, r_4$ are too large
- Solution: carry!

A suitable carry chain

- Basically the same as before, but now with 128-bit values (tricky, but possible in assembly)

  ```
  signed int128 carry = r.a[0] >> 51;
  r.a[1] += carry;
  carry <<= 51;
  r.a[0] -= carry;
  ```

- Carry from r_0 to r_1; from r_1 to r_2, and so on
- Multiply carry from r_4 by 19 and add to r_0

A suitable carry chain

- Basically the same as before, but now with 128-bit values (tricky, but possible in assembly)

  ```
  signed int128 carry = r.a[0] >> 51;
  r.a[1] += carry;
  carry <<= 51;
  r.a[0] -= carry;
  ```

- Carry from r_0 to r_1; from r_1 to r_2, and so on
- Multiply carry from r_4 by 19 and add to r_0
- After one round of carries we have signed 64-bit integers
- Perform another round of carries to obtain reduced coefficients

Squaring

- Obviously working solution for squaring:
  `#define square(R,X) mul(R,X,X)`
- Question: Can we do better?

Squaring

- Obviously working solution for squaring:

  ```
  #define square(R,X) mul(R,X,X)
  ```

- Question: Can we do better?
- Using multiplication for squarings:

```
r[0] = a[0]*a[0];
r[1] = a[0]*a[1] + a[1]*a[0];
r[2] = a[0]*a[2] + a[1]*a[1] + a[2]*a[0];
r[3] = a[0]*a[3] + a[1]*a[2] + a[2]*a[1] + a[3]*a[0];
r[4] = a[0]*a[4] + a[1]*a[3] + a[2]*a[2] + a[3]*a[1] + a[4]*a[0];
r[5] = a[1]*a[4] + a[2]*a[3] + a[3]*a[2] + a[4]*a[1];
r[6] = a[2]*a[4] + a[3]*a[3] + a[4]*a[2];
r[7] = a[3]*a[4] + a[4]*a[3];
r[8] = a[4]*a[4];
```

Squaring

- Obviously working solution for squaring:

  ```
  #define square(R,X) mul(R,X,X)
  ```

- Question: Can we do better?
- Using multiplication for squarings:

```
r[0] = a[0]*a[0];
r[1] = a[0]*a[1] + a[1]*a[0];
r[2] = a[0]*a[2] + a[1]*a[1] + a[2]*a[0];
r[3] = a[0]*a[3] + a[1]*a[2] + a[2]*a[1] + a[3]*a[0];
r[4] = a[0]*a[4] + a[1]*a[3] + a[2]*a[2] + a[3]*a[1] + a[4]*a[0];
r[5] = a[1]*a[4] + a[2]*a[3] + a[3]*a[2] + a[4]*a[1];
r[6] = a[2]*a[4] + a[3]*a[3] + a[4]*a[2];
r[7] = a[3]*a[4] + a[4]*a[3];
r[8] = a[4]*a[4];
```

- Observation: We perform many multiplications twice!

Faster squaring

```
signed long long _2a[4];
_2a[0] = a[0] << 1;
_2a[1] = a[1] << 1;
_2a[2] = a[2] << 1;
_2a[3] = a[3] << 1;

r[0] = a[0]*a[0];
r[1] = _2a[0]*a[1];
r[2] = _2a[0]*a[2] + a[1]*a[1];
r[3] = _2a[0]*a[3] + _2a[1]*a[2];
r[4] = _2a[0]*a[4] + _2a[1]*a[3] + a[2]*a[2];
r[5] = _2a[1]*a[4] + _2a[2]*a[3];
r[6] = _2a[2]*a[4] + a[3]*a[3];
r[7] = _2a[3]*a[4];
r[8] = a[4]*a[4];
```

- Multiplication needs 25 multiplications, 16 additions
- Squaring needs 15 multiplications, 6 additions (and 4 shifts)

Faster multiplication?

- Consider multiplication of two n-coefficient polynomials (degree $\leq n-1$)
- So far we needed n^2 multiplications and $(n-1)^2$ additions
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity

Faster multiplication?

- Consider multiplication of two n-coefficient polynomials (degree $\leq n-1$)
- So far we needed n^2 multiplications and $(n-1)^2$ additions
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Assume that $n = 2m$, then write an n-coefficient polynomial A as $A_0 + X^m A_1$
- Perform multiplication as

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

Faster multiplication?

- Consider multiplication of two n-coefficient polynomials (degree $\leq n-1$)
- So far we needed n^2 multiplications and $(n-1)^2$ additions
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Assume that $n = 2m$, then write an n-coefficient polynomial A as $A_0 + X^m A_1$
- Perform multiplication as

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

Faster multiplication?

- Consider multiplication of two n-coefficient polynomials (degree $\leq n-1$)
- So far we needed n^2 multiplications and $(n-1)^2$ additions
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Assume that $n = 2m$, then write an n-coefficient polynomial A as $A_0 + X^m A_1$
- Perform multiplication as

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

- We just turned one multiplication of size n into 3 multiplications of size $n/2$ (and about $8m$ additions)
- Recursive application yields asymptotic complexity $O(n^{\log_2 3})$

Even faster multiplication?

- Karatsuba equality:

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

Even faster multiplication?

- Karatsuba equality:

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

- *Refined* Karatsuba equality:

$$
\begin{aligned}
&(A_0 + X^m A_1)(B_0 + X^m B_1) \\
=&(1 - X^m)(A_0 B_0 - X^m A_1 B_1) + X^m (A_0 + A_1)(B_0 + B_1)
\end{aligned}
$$

Even faster multiplication?

- Karatsuba equality:

$$\begin{aligned}&(A_0 + X^m A_1) \cdot (B_0 + X^m B_1)\\ =&A_0B_0 + ((A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1)X^m + A_1B_1X^{2m}\end{aligned}$$

- *Refined* Karatsuba equality:

$$\begin{aligned}&(A_0 + X^m A_1)(B_0 + X^m B_1)\\ =&(1 - X^m)(A_0B_0 - X^m A_1B_1) + X^m(A_0 + A_1)(B_0 + B_1)\end{aligned}$$

- This reduces the $\approx 8m$ additions to $\approx 7m$ additions (see Bernstein "Batch binary Edwards", 2009)
- No reduction of asymptotic running time, but speedup in practice

Multiplication, can we go further?

- Toom-Cook multiplication has asymptotic complexity $O(n^{\log_3 5})$
- Schönhage-Strassen multiplication has asymptotic complexity $O(n \log n \log \log n)$
- Fürer's multiplication algorithm has running time $n \log n 2^{O(\log^* n)}$

Karatsuba for $\mathbb{F}_{2^{255}-19}$ (in idealized C)

```
signed int128 rm0,rm1,rm2,rm3,rm4;
signed long long am0,am1,am2,bm0,bm1,bm2;

am0 = a[0] + a[3];
am0 = a[1] + a[4];
am0 = a[2];
am0 = b[0] + b[3];
am0 = b[1] + b[4];
am0 = b[2];

r[0] = a[0]*b[0];
r[1] = a[0]*b[1] + a[1]*b[0];
r[2] = a[0]*b[2] + a[1]*b[1] + a[2]*b[0];
r[3] = a[1]*b[2] + a[2]*b[1];
r[4] = a[2]*b[2];

r[6] = a[3]*b[3];
r[7] = a[3]*b[4] + a[4]*b[3];
r[8] = a[4]*b[4];
```

Karatsuba for $\mathbb{F}_{2^{255}-19}$ (in idealized C) ctd.

```
rm[0] = am[0]*bm[0] - r[0] - r[6];
rm[1] = am[0]*bm[1] + am[1]*b[0] - r[1] - r[7];
rm[2] = am[0]*bm[2] + am[1]*b[1] + am[2]*b[0] - r[2] - r[8];
rm[3] = am[1]*bm[2] + am[2]*b[1] - r[3];
rm[4] = am[2]*bm[2] - r[4];

r[3] += rm[0];
r[4] += rm[1];
r[5]  = rm[2];
r[6] += rm[3];
r[6] += rm[4];
```

Karatsuba for $\mathbb{F}_{2^{255}-19}$ (in idealized C) ctd.

```
rm[0] = am[0]*bm[0] - r[0] - r[6];
rm[1] = am[0]*bm[1] + am[1]*b[0] - r[1] - r[7];
rm[2] = am[0]*bm[2] + am[1]*b[1] + am[2]*b[0] - r[2] - r[8];
rm[3] = am[1]*bm[2] + am[2]*b[1] - r[3];
rm[4] = am[2]*bm[2] - r[4];

r[3] += rm[0];
r[4] += rm[1];
r[5]  = rm[2];
r[6] += rm[3];
r[6] += rm[4];
```

- 22 multiplications, 4 small additions, 21 big additions
- Is this better? I doubt it.

Which multiplication algorithm to use

- Depends on the size of the field

Which multiplication algorithm to use

- Depends on the size of the field
- Depends on representation of field elements (signed vs. unsigned, radix, etc.)

Which multiplication algorithm to use

- Depends on the size of the field
- Depends on representation of field elements (signed vs. unsigned, radix, etc.)
- Depends on computer microarchitecture (speed of multiplication vs. speed of addition)

Which multiplication algorithm to use

- Depends on the size of the field
- Depends on representation of field elements (signed vs. unsigned, radix, etc.)
- Depends on computer microarchitecture (speed of multiplication vs. speed of addition)
- Rule of thumb:
  - For ≤ 10 limbs (coefficients) use schoolbook multiplication

Which multiplication algorithm to use

- Depends on the size of the field
- Depends on representation of field elements (signed vs. unsigned, radix, etc.)
- Depends on computer microarchitecture (speed of multiplication vs. speed of addition)
- Rule of thumb:
  - For ≤ 10 limbs (coefficients) use schoolbook multiplication
  - For > 10 start to think about (refined) Karatsuba

Which multiplication algorithm to use

- Depends on the size of the field
- Depends on representation of field elements (signed vs. unsigned, radix, etc.)
- Depends on computer microarchitecture (speed of multiplication vs. speed of addition)
- Rule of thumb:
  - For ≤ 10 limbs (coefficients) use schoolbook multiplication
  - For > 10 start to think about (refined) Karatsuba
  - For field sizes appearing in ECC, I never saw anybody using Toom-Cook or Schönhage-Strassen (however, Toom-Cook may become interesting in pairing computations)

Which multiplication algorithm to use

- Depends on the size of the field
- Depends on representation of field elements (signed vs. unsigned, radix, etc.)
- Depends on computer microarchitecture (speed of multiplication vs. speed of addition)
- Rule of thumb:
  - For ≤ 10 limbs (coefficients) use schoolbook multiplication
  - For > 10 start to think about (refined) Karatsuba
  - For field sizes appearing in ECC, I never saw anybody using Toom-Cook or Schönhage-Strassen (however, Toom-Cook may become interesting in pairing computations)
  - I don't know of any application using Fürer's algorithm

Still missing: inversion

- Inversion is typically *much* more expensive than multiplication
- This is why we like projective coordinates

Still missing: inversion

- Inversion is typically *much* more expensive than multiplication
- This is why we like projective coordinates
- Before sending an elliptic-curve point, we need to convert from projective coordinates to affine coordinates (for security reasons!)
- We need inversion, but we do (usually) not need it often

Still missing: inversion

- Inversion is typically *much* more expensive than multiplication
- This is why we like projective coordinates
- Before sending an elliptic-curve point, we need to convert from projective coordinates to affine coordinates (for security reasons!)
- 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)

Input: Integers a and b.
Output: 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
- Going into detail of multiprecision (big-integer) division would cost us lunch

Some notes about the Extended Euclidean algorithm

- Core operation are divisions with remainder
- Going into detail of multiprecision (big-integer) division would cost us lunch
- The running time (number of loop iterations) depends on the inputs
- We usually do not want this for cryptography (more this afternoon)

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?
- Answer: yes, fairly. Inversion modulo $2^{255}-19$ needs 254 squarings and 11 multiplications in $\mathbb{F}_{2^{255}-19}$

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?
- Answer: yes, fairly. Inversion modulo $2^{255}-19$ needs 254 squarings and 11 multiplications in $\mathbb{F}_{2^{255}-19}$
- Details in my talk this afternoon

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 recompute y as $\sqrt{x^3 + ax + b}$

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 recompute y as $\sqrt{x^3 + ax + b}$
- 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 recompute y as $\sqrt{x^3 + ax + b}$
- 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 recompute y as $\sqrt{x^3 + ax + b}$
- 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

Getting back to the rabbits

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

Getting back to the rabbits

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

$$\begin{aligned} p_{224} =& 22721622932454352787552537995910928073340 73\backslash \\ & 2145944992304435472941311 \\ =& 0xD7C134AA264366862A18302575D1D787B09F07579\backslash \\ & 7DA89F57EC8C0FF \end{aligned}$$

or

$$\begin{aligned} p_{256} =& 76884956397045344220809746629001649093037 95\backslash \\ & 0200943055203735601445031516197751 \\ =& 0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D\backslash \\ & 52620282013481D1F6E5377 \end{aligned}$$

Getting back to the rabbits

- 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 (but this would cost us lunch)

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 (but this would cost us lunch)
- 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 (but this would cost us lunch)
- 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 be more efficient than division
  - Typical choice for radix-b representation: b^n

Montgomery reduction (pseudocode)

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

$A \leftarrow t$
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 $A > p$ **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
- One can merge schoolbook multiplication with Montgomery reduction: "Montgomery multiplication"

Summary

- Efficiency of finite-field arithmetic highly depends on the representation of field elements
- The obvious representation is not always the best one

Summary

- Efficiency of finite-field arithmetic highly depends on the representation of field elements
- The obvious representation is not always the best one
- Carries are annoying (not only in C)

Summary

- Efficiency of finite-field arithmetic highly depends on the representation of field elements
- The obvious representation is not always the best one
- Carries are annoying (not only in C)
- Be careful with the complexity of multiplication
- In particular if somebody uses it to estimate real-world performance

Summary

- Efficiency of finite-field arithmetic highly depends on the representation of field elements
- The obvious representation is not always the best one
- Carries are annoying (not only in C)
- Be careful with the complexity of multiplication
- In particular if somebody uses it to estimate real-world performance
- Don't be afraid to use assembly, but consider `qhasm` (`http://cr.yp.to/qhasm.html`)

Summary

- Efficiency of finite-field arithmetic highly depends on the representation of field elements
- The obvious representation is not always the best one
- Carries are annoying (not only in C)
- Be careful with the complexity of multiplication
- In particular if somebody uses it to estimate real-world performance
- Don't be afraid to use assembly, but consider `qhasm` (`http://cr.yp.to/qhasm.html`)
- Remember the Explicit Formulas Database `http://www.hyperelliptic.org/EFD/`