Engineering Cryptographic Software

Multiprecision arithmetic

Radboud University, Nijmegen, The Netherlands

Winter 2021

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 $\dots$
- 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: AMD64 supports up to 64-bit integers, multiplication produces 128-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 ...
- Example 3: Poly1305 needs arithmetic on 130-bit integers
- An integer is "big" if it's not natively supported by the machine architecture
- Example: AMD64 supports up to 64-bit integers, multiplication produces 128-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
- Example architecture for today (most of the time): AVR ATmega

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: $0, 1, \ldots, 255$

Programming today

Available numbers: $0, 1, \ldots, 255$

Addition

```
uint8_t a = 42;
uint8_t b = 89;
uint8_t r = a + b;
```

Programming today

Available numbers: $0, 1, \ldots, 255$

Addition

```
uint8_t a = 42;
uint8_t b = 89;
uint8_t r = a + b;
```

Subtraction

```
uint8_t a = 157;
uint8_t b = 23;
uint8_t r = a - b;
```

Programming today

Available numbers: $0, 1, \ldots, 255$

Addition

```
uint8_t a = 42;
uint8_t b = 89;
uint8_t r = a + b;
```

Subtraction

```
uint8_t a = 157;
uint8_t b = 23;
uint8_t r = a - b;
```

- All results are in the set of available numbers
- Larger set of available numbers: `uint16_t`, `uint32_t`, `uint64_t`
- Basic principle is the same; for the moment stick with `uint8_t`

Still in the first year of primary school

Crossing the ten barrier

$$
\begin{aligned}
6 + 5 &= \quad ? \\
9 + 7 &= \quad ? \\
4 + 8 &= \quad ?
\end{aligned}
$$

Still in the first year of primary school

Crossing the ten barrier

$$
\begin{aligned}
6 + 5 &= \quad ? \\
9 + 7 &= \quad ? \\
4 + 8 &= \quad ?
\end{aligned}
$$

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

Still in the first year of primary school

Crossing the ten barrier

$$\begin{aligned} 6+5 &= \quad ? \\ 9+7 &= \quad ? \\ 4+8 &= \quad ? \end{aligned}$$

- 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

$$\begin{aligned} 6+5 &= \quad ? \\ 9+7 &= \quad ? \\ 4+8 &= \quad ? \end{aligned}$$

- 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_1a_0 with

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

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

. . . back to programming

```
uint8_t a = 184;
uint8_t b = 203;
uint8_t r = a + b;
```

. . . back to programming

```
uint8_t a = 184;
uint8_t b = 203;
uint8_t r = a + b;
```

- The result `r` now has the value of 131
- The carry is lost, what do we do?

. . . back to programming

```
uint8_t a = 184;
uint8_t b = 203;
uint8_t r = a + b;
```

- The result `r` now has the value of 131
- The carry is lost, what do we do?
- Could cast to `uint16_t`, `uint32_t` etc.,
  but that solves the problem only for this `uint8_t` example
- We really want to obtain the carry, and put it into another `uint8_t`

The AVR ATmega

- 8-bit RISC architecture
- 32 registers R0...R31, some of those are "special":
  - (R26,R27) aliased as X
  - (R28,R29) aliased as Y
  - (R30,R31) aliased as Z
  - X, Y, Z are used for addressing
  - 2-byte output of a multiplication always in R0, R1
- Most arithmetic instructions cost 1 cycle
- Multiplication and memory access takes 2 cycles

$184 + 203$

```
LDI R5, 184
LDI R6, 203
ADD R5, R6   ; result in R5, sets carry flag
CLR R6       ; set R6 to zero
ADC R6,R6    ; add with carry, R6 now holds the carry
```

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

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

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

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

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

Later in primary school

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)

AVR multiprecision addition...

- Add two n-byte numbers, returning an $n+1$ byte result:
- Input pointers X,Y, output pointer Z

```
LD R5,X+
LD R6,Y+
ADD R5,R6
ST Z+,R5

LD R5,X+
LD R6,Y+
ADC R5,R6
ST Z+,R5
```

```
LD R5,X+
LD R6,Y+
ADC R5,R6
ST Z+,R5

LD R5,X+
LD R6,Y+
ADC R5,R6
ST Z+,R5

...
```

```
CLR R5
ADC R5,R5
ST Z+,R5
```

. . . and subtraction

- Subtract two n-byte numbers, returning an $n+1$ byte result:
- Input pointers X,Y, output pointer Z
- Use highest byte $= -1$ to indicate negative result

```
LD R5,X+
LD R6,Y+
SUB R5,R6
ST Z+,R5

LD R5,X+
LD R6,Y+
SBC R5,R6
ST Z+,R5

LD R5,X+
LD R6,Y+
SBC R5,R6
ST Z+,R5

LD R5,X+
LD R6,Y+
SBC R5,R6
ST Z+,R5

...

CLR R5
SBC R5,R5
ST Z+,R5
```

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

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

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

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

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

$$\begin{array}{rr} & 1234 \cdot 789 \\ \hline & 20978 \\ + & 8638 \end{array}$$

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

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

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

Let's do that on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+

LD R7, Y+

MUL R2,R7
ST Z+,R0
MOV R8,R1

MUL R3,R7
ADD R8,R0
CLR R9
ADC R9,R1

MUL R4,R7
ADD R9,R0
CLR R10
ADC R10,R1
```

Let's do that on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+

LD R7, Y+

MUL R2,R7
ST Z+,R0
MOV R8,R1

MUL R3,R7
ADD R8,R0
CLR R9
ADC R9,R1

MUL R4,R7
ADD R9,R0
CLR R10
ADC R10,R1
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADD R8,R12
ST Z+,R8
ADC R9,R13
ADC R10,R14
CLR R11
ADC R11,R15
```

Let's do that on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+

LD R7, Y+

MUL R2,R7
ST Z+,R0
MOV R8,R1

MUL R3,R7
ADD R8,R0
CLR R9
ADC R9,R1

MUL R4,R7
ADD R9,R0
CLR R10
ADC R10,R1
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADD R8,R12
ST Z+,R8
ADC R9,R13
ADC R10,R14
CLR R11
ADC R11,R15
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADC R9,R12
ST Z+,R9
ADC R10,R13
ADC R11,R14
CLR R12
ADC R12,R15
```

Let's do that on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+

LD R7, Y+

MUL R2,R7
ST Z+,R0
MOV R8,R1

MUL R3,R7
ADD R8,R0
CLR R9
ADC R9,R1

MUL R4,R7
ADD R9,R0
CLR R10
ADC R10,R1
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADD R8,R12
ST Z+,R8
ADC R9,R13
ADC R10,R14
CLR R11
ADC R11,R15
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADC R9,R12
ST Z+,R9
ADC R10,R13
ADC R11,R14
CLR R12
ADC R12,R15
```

```
ST Z+,R10
ST Z+,R11
ST Z+,R12
```

Let's do that on the AVR

- Problem: Need $3n + c$ registers for $n \times n$-byte multiplication

Let's do that on the AVR

- Problem: Need $3n + c$ registers for $n \times n$-byte 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 on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+
LD R7, Y+
LD R8, Y+
LD R9, Y+


MUL R2, R7
MOV R13, R1
STD Z+0, R0
CLR R14
CLR R15

MUL R2, R8
ADD R13, R0
ADC R14, R1
MUL R3, R7
ADD R13, R0
ADC R14, R1
ADC R15, R5
STD Z+1, R13
CLR R16
```

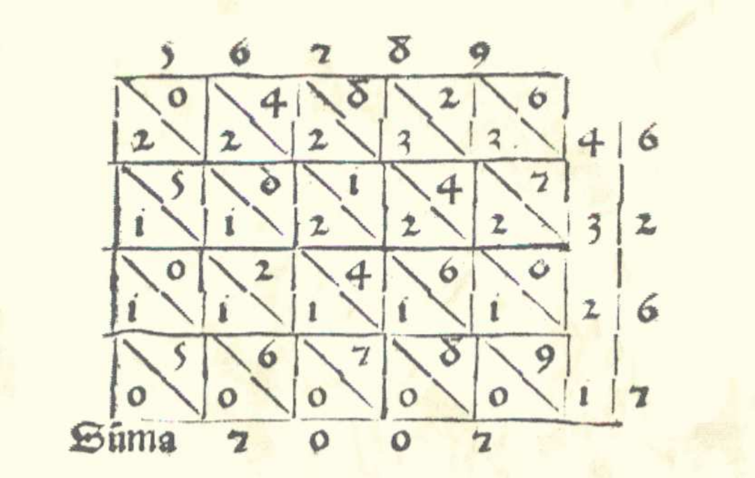
```
MUL R2, R9
ADD R14, R0
ADC R15, R1
ADC R16, R5
MUL R3, R8
ADD R14, R0
ADC R15, R1
ADC R16, R5
MUL R4, R7
ADD R14, R0
ADC R15, R1
ADC R16, R5
STD Z+2, R14
CLR R17
```

```
MUL R3, R9
ADD R15, R0
ADC R16, R1
ADC R17, R5
MUL R4, R8
ADD R15, R0
ADC R16, R1
ADC R17, R5
STD Z+3, R15

MUL R4, R9
ADD R16, R0
ADC R17, R1
STD Z+4, R16

STD Z+5, R17
```

Even better. . . ?



From the Treviso Arithmetic, 1478 (http://www.republicaveneta.com/doc/abaco.pdf)

Hybrid multiplication

- Idea: Chop whole multiplication into smaller blocks
- Compute each of the smaller multiplications by schoolbook
- Later add up to the full result
- See it as two nested loops:
  - Inner loop performs operand scanning
  - Outer loop performs product scanning

Hybrid multiplication

- Idea: Chop whole multiplication into smaller blocks
- Compute each of the smaller multiplications by schoolbook
- Later add up to the full result
- See it as two nested loops:
  - Inner loop performs operand scanning
  - Outer loop performs product scanning
- Originally proposed by Gura, Patel, Wander, Eberle, Chang Shantz, 2004

Hybrid multiplication

- Idea: Chop whole multiplication into smaller blocks
- Compute each of the smaller multiplications by schoolbook
- Later add up to the full result
- See it as two nested loops:
  - Inner loop performs operand scanning
  - Outer loop performs product scanning
- Originally proposed by Gura, Patel, Wander, Eberle, Chang Shantz, 2004
- Various improvements, consider 160-bit multiplication:
  - Originally: 3106 cycles
  - Uhsadel, Poschmann, Paar (2007): 2881 cycles
  - Scott, Szczechowiak (2007): 2651 cycles
  - Kargl, Pyka, Seuschek (2008): 2593 cycles

Operand-caching multiplication

- Hutter, Wenger, 2011: More efficient way to decompose multiplication
- Inside separate chunks use product-scanning
- Main idea: re-use values in registers for longer

Operand-caching multiplication

- Hutter, Wenger, 2011: More efficient way to decompose multiplication
- Inside separate chunks use product-scanning
- Main idea: re-use values in registers for longer
- Performance:
  - 2393 cycles for 160-bit multiplication
  - 6121 cycles for 256-bit multiplication

Operand-caching multiplication

- Hutter, Wenger, 2011: More efficient way to decompose multiplication
- Inside separate chunks use product-scanning
- Main idea: re-use values in registers for longer
- Performance:
  - 2393 cycles for 160-bit multiplication
  - 6121 cycles for 256-bit multiplication
- Followup-paper by Seo and Kim: "Consecutive operand caching":
  - 2341 cycles for 160-bit multiplication
  - 6115 cycles for 256-bit multiplication

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

$$A_0B_0 + \qquad 2^m(A_0B_1 + B_0A_1) \qquad + 2^{2m}A_1B_1$$

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

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

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `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

$$
\begin{aligned}
&A_0B_0 + \qquad 2^m(A_0B_1 + B_0A_1) \qquad + 2^{2m}A_1B_1 \\
=&A_0B_0 + 2^m((A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1) + 2^{2m}A_1B_1
\end{aligned}
$$

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

Does that help on the AVR?

The straight-forward approach

Consider multiplication of n-byte numbers

$$A \mathrel{\hat{=}} (a_0, \ldots, a_{n-1}) \text{ and}$$
$$B \mathrel{\hat{=}} (b_0, \ldots, b_{n-1})$$

The straight-forward approach

Consider multiplication of n-byte numbers

$$A \mathrel{\hat{=}} (a_0, \ldots, a_{n-1}) \text{ and}$$
$$B \mathrel{\hat{=}} (b_0, \ldots, b_{n-1})$$

- Write $A = A_\ell + 2^{8k} A_h$ and $B = B_\ell + 2^{8k} B_h$ for k-byte integers A_ℓ, A_h, B_ℓ, and B_h and $k = n/2$

The straight-forward approach

Consider multiplication of n-byte numbers

$$A \mathrel{\hat{=}} (a_0, \dots, a_{n-1}) \text{ and}$$
$$B \mathrel{\hat{=}} (b_0, \dots, b_{n-1})$$

- Write $A = A_\ell + 2^{8k} A_h$ and $B = B_\ell + 2^{8k} B_h$ for k-byte integers A_ℓ, A_h, B_ℓ, and B_h and $k = n/2$
- Compute $L = A_\ell \cdot B_\ell \mathrel{\hat{=}} (\ell_0, \dots, \ell_{n-1})$
- Compute $H = A_h \cdot B_h \mathrel{\hat{=}} (h_0, \dots, h_{n-1})$
- Compute $M = (A_\ell + A_h) \cdot (B_\ell + B_h) \mathrel{\hat{=}} (m_0, \dots, m_n)$

The straight-forward approach

Consider multiplication of n-byte numbers

$$A \hat{=} (a_0, \ldots, a_{n-1}) \text{ and}$$
$$B \hat{=} (b_0, \ldots, b_{n-1})$$

- Write $A = A_\ell + 2^{8k} A_h$ and $B = B_\ell + 2^{8k} B_h$ for k-byte integers A_ℓ, A_h, B_ℓ, and B_h and $k = n/2$
- Compute $L = A_\ell \cdot B_\ell \hat{=} (\ell_0, \ldots, \ell_{n-1})$
- Compute $H = A_h \cdot B_h \hat{=} (h_0, \ldots, h_{n-1})$
- Compute $M = (A_\ell + A_h) \cdot (B_\ell + B_h) \hat{=} (m_0, \ldots, m_n)$
- Obtain result as $A \cdot B = L + 2^{8k}(M - L - H) + 2^{8n} H$

Multiplication by the carry in M

- Can expand carry to `0xff` or `0x00`
- Use `AND` instruction for multiplication

Multiplication by the carry in M

- Can expand carry to `0xff` or `0x00`
- Use `AND` instruction for multiplication
- Does not help for recursive Karatsuba

Multiplication by the carry in M

- Can expand carry to `0xff` or `0x00`
- Use `AND` instruction for multiplication
- Does not help for recursive Karatsuba

Subtractive Karatsuba

- Compute $L = A_\ell \cdot B_\ell \mathrel{\hat{=}} (\ell_0, \dots, \ell_{n-1})$
- Compute $H = A_h \cdot B_h \mathrel{\hat{=}} (h_0, \dots, h_{n-1})$
- Compute $M = |A_\ell - A_h| \cdot |B_\ell - B_h| \mathrel{\hat{=}} (m_0, \dots, m_{n-1})$
- Set $t = 0$, if $M = (A_\ell - A_h) \cdot (B_\ell - B_h)$; $t = 1$ otherwise
- Compute $\hat{M} = (-1)^t M = (A_\ell - A_h)(B_\ell - B_h)$ $\mathrel{\hat{=}} (\hat{m}_0, \dots, \hat{m}_{n-1})$
- Obtain result as $A \cdot B = L + 2^{8k}(L + H - \hat{M}) + 2^{8n} H$

Conditional negation

The easy solution

```
if(b) a = -a
```

Conditional negation

The easy solution

```
if(b) a = -a
```

- `NEG` instruction does not help for multiprecision
- Can subtract from zero, but subtraction would overwrite zero

Conditional negation

The easy solution

```
if(b) a = -a
```

- `NEG` instruction does not help for multiprecision
- Can subtract from zero, but subtraction would overwrite zero
- Even worse, the `if` would create a timing side-channel!

Conditional negation

The easy solution

```
if(b) a = -a
```

- `NEG` instruction does not help for multiprecision
- Can subtract from zero, but subtraction would overwrite zero
- Even worse, the `if` would create a timing side-channel!

The constant-time solution

- Produce condition bit as byte `0xff` or `0x00`
- XOR all limbs with this condition byte

Conditional negation

The easy solution

```
if(b) a = -a
```

- `NEG` instruction does not help for multiprecision
- Can subtract from zero, but subtraction would overwrite zero
- Even worse, the `if` would create a timing side-channel!

The constant-time solution

- Produce condition bit as byte `0xff` or `0x00`
- XOR all limbs with this condition byte
- Negate the condition byte and obtain `0x01` or `0x00`
- Add this value to the lowest byte
- Ripple through the carry (`ADC` with zero)

Conditional negation

The easy solution

```
if(b) a = -a
```

- `NEG` instruction does not help for multiprecision
- Can subtract from zero, but subtraction would overwrite zero
- Even worse, the `if` would create a timing side-channel!

The constant-time solution

- Produce condition bit as byte `0xff` or `0x00`
- XOR all limbs with this condition byte
- Don't negate the condition byte
- Subtract the condition byte (`0xff` or `0x00` from all bytes)
- Saves two `NEG` instructions and the zero register

Refined Karatsuba

- Consider example of 4×4-byte Karatsuba multiplication:

	l_0	l_1	l_2	l_3	h_0	h_1	h_2	h_3
		-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
		+	l_0	l_1	l_2	l_3		
		+	h_0	h_1	h_2	h_3		

Refined Karatsuba

- Consider example of 4×4-byte Karatsuba multiplication:

l_0	l_1		l_2	l_3	h_0	h_1	h_2	h_3
		-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
		+	l_0	l_1	l_2	l_3		
		+	h_0	h_1	h_2	h_3		

- Karatsuba performs some additions twice
- Refined Karatsuba: do them only once

Refined Karatsuba

- Consider example of 4×4-byte Karatsuba multiplication:

l_0	l_1	l_2	l_3	h_0	h_1	h_2	h_3
	-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	+	l_0	l_1	l_2	l_3		
	+	h_0	h_1	h_2	h_3		

- Karatsuba performs some additions twice
- Refined Karatsuba: do them only once
- Merge additions into computation of H
- Compute $\mathbf{H} \hat{=} (\mathbf{h_0}, \mathbf{h_1}, \mathbf{h_2}, \mathbf{h_3}) = H + (l_2, l_3)$
- Note that $\mathbf{H}$ cannot "overflow"

Refined Karatsuba

- Consider example of 4×4-byte Karatsuba multiplication:

l_0	l_1	l_2	l_3	h_0	h_1	h_2	h_3
	-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	+	l_0	l_1	l_2	l_3		
	+	h_0	h_1	h_2	h_3		

- Karatsuba performs some additions twice
- Refined Karatsuba: do them only once
- Merge additions into computation of H
- Compute $\mathbf{H} \hat{=} (\mathbf{h_0}, \mathbf{h_1}, \mathbf{h_2}, \mathbf{h_3}) = H + (l_2, l_3)$
- Note that $\mathbf{H}$ cannot "overflow"

l_0	l_1			$\mathbf{h_0}$	$\mathbf{h_1}$	$\mathbf{h_2}$	$\mathbf{h_3}$
	-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	+	l_0	l_1				
	+	$\mathbf{h_0}$	$\mathbf{h_1}$	$\mathbf{h_2}$	$\mathbf{h_3}$		

Refined Karatsuba

- Consider example of 4×4-byte Karatsuba multiplication:

l_0	l_1	l_2	l_3	h_0	h_1	h_2	h_3
	-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	+	l_0	l_1	l_2	l_3		
	+	h_0	h_1	h_2	h_3		

- Karatsuba performs some additions twice
- Refined Karatsuba: do them only once
- Merge additions into computation of H
- Compute $\mathbf{H} \hat{=} (\mathbf{h_0}, \mathbf{h_1}, \mathbf{h_2}, \mathbf{h_3}) = H + (l_2, l_3)$
- Note that $\mathbf{H}$ cannot "overflow"

l_0	l_1	$\mathbf{h_0}$	$\mathbf{h_1}$	$\mathbf{h_0}$	$\mathbf{h_1}$	$\mathbf{h_2}$	$\mathbf{h_3}$
	-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	+	l_0	l_1	$\mathbf{h_2}$	$\mathbf{h_3}$		

Refined Karatsuba

- Consider example of 4×4-byte Karatsuba multiplication:

l_0	l_1	l_2	l_3	h_0	h_1	h_2	h_3
	-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	+	l_0	l_1	l_2	l_3		
	+	h_0	h_1	h_2	h_3		

- Karatsuba performs some additions twice
- Refined Karatsuba: do them only once
- Merge additions into computation of H
- Compute $\mathbf{H} \hat{=} (\mathbf{h_0}, \mathbf{h_1}, \mathbf{h_2}, \mathbf{h_3}) = H + (l_2, l_3)$
- Note that $\mathbf{H}$ cannot "overflow"

l_0	l_1	$\mathbf{h_0}$	$\mathbf{h_1}$	$\mathbf{h_0}$	$\mathbf{h_1}$	$\mathbf{h_2}$	$\mathbf{h_3}$
	-	$\hat{m}_0$	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$		
	+	l_0	l_1	$\mathbf{h_2}$	$\mathbf{h_3}$		

- Consequence: fewer additions, easier register allocation

Putting it together

Arithmetic cost of n-byte Karatsuba on AVR

- Cost of computing L, M, and $\mathbf{H}$

Putting it together

Arithmetic cost of n-byte Karatsuba on AVR

- Cost of computing L, M, and $\mathbf{H}$
- $4k + 2$ `SUB`/`SBC`, $2k$ `EOR` for absolute differences

Putting it together

Arithmetic cost of n-byte Karatsuba on AVR

- Cost of computing L, M, and $\mathbf{H}$
- $4k+2$ `SUB`/`SBC`, $2k$ `EOR` for absolute differences
- $n+1$ `ADD`/`ADC` to add $(l_0,\ldots,l_{k-1},\mathbf{h_k},\ldots,\mathbf{h_{n-1}})$

Putting it together

Arithmetic cost of n-byte Karatsuba on AVR

- Cost of computing L, M, and $\mathbf{H}$
- $4k+2$ `SUB`/`SBC`, $2k$ `EOR` for absolute differences
- $n+1$ `ADD`/`ADC` to add $(l_0, \ldots, l_{k-1}, \mathbf{h_k}, \ldots, \mathbf{h_{n-1}})$
- One `EOR` to compute t
- A `BRNE` instruction to branch, then either

Putting it together

Arithmetic cost of n-byte Karatsuba on AVR

- Cost of computing L, M, and $\mathbf{H}$
- $4k+2$ `SUB`/`SBC`, $2k$ `EOR` for absolute differences
- $n+1$ `ADD`/`ADC` to add $(l_0, \ldots, l_{k-1}, \mathbf{h_k}, \ldots, \mathbf{h_{n-1}})$
- One `EOR` to compute t
- A `BRNE` instruction to branch, then either
  - $n+2$ `SUB`/`SBC` instructions and one `RJMP`, or
  - $n+1$ `ADD`/`ADC`, one `CLR`, and one `NOP`

Putting it together

Arithmetic cost of n-byte Karatsuba on AVR

- Cost of computing L, M, and $\mathbf{H}$
- $4k + 2$ `SUB`/`SBC`, $2k$ `EOR` for absolute differences
- $n + 1$ `ADD`/`ADC` to add $(l_0, \ldots, l_{k-1}, \mathbf{h_k}, \ldots, \mathbf{h_{n-1}})$
- One `EOR` to compute t
- A `BRNE` instruction to branch, then either
  - $n + 2$ `SUB`/`SBC` instructions and one `RJMP`, or
  - $n + 1$ `ADD`/`ADC`, one `CLR`, and one `NOP`
- k `ADD`/`ADC` instructions to ripple carry to the end

48-bit Karatsuba on AVR

```
CLR R22
CLR R23
MOVW R12, R22
MOVW R20, R22

LD R2, X+
LD R3, X+
LD R4, X+
LDD R5, Y+0
LDD R6, Y+1
LDD R7, Y+2

MUL R2, R7
MOVW R10, R0
MUL R2, R5
MOVW R8, R0
MUL R2, R6
ADD R9, R0
ADC R10, R1
ADC R11, R23

MUL R3, R7
MOVW R14, R0
MUL R3, R5
ADD R9, R0
ADC R10, R1
ADC R11, R14
ADC R15, R23
MUL R3, R6
ADD R10, R0
ADC R11, R1
ADC R12, R15

MUL R4, R7
MOVW R14, R0
MUL R4, R5
ADD R10, R0
ADC R11, R1
ADC R12, R14
ADC R15, R23
MUL R4, R6
ADD R11, R0
ADC R12, R1
ADC R13, R15
STD Z+0, R8
STD Z+1, R9
STD Z+2, R10

LD R14, X+
LD R15, X+
LD R16, X+
LDD R17, Y+3
LDD R18, Y+4
LDD R19, Y+5

SUB R2, R14
SBC R3, R15
SBC R4, R16
SBC R26, R26

SUB R5, R17
SBC R6, R18
SBC R7, R19
SBC R27, R27

EOR R2, R26
EOR R3, R26
EOR R4, R26
EOR R5, R27
EOR R6, R27
EOR R7, R27

SUB R2, R26
SBC R3, R26
SBC R4, R26
SUB R5, R27
SBC R6, R27
SBC R7, R27
```

48-bit Karatsuba on AVR

```
MUL R14, R19
MOVW R24, R0
MUL R14, R17
ADD R11, R0
ADC R12, R1
ADC R13, R24
ADC R25, R23
MUL R14, R18
ADD R12, R0
ADC R13, R1
ADC R20, R25

MUL R15, R19
MOVW R24, R0
MUL R15, R17
ADD R12, R0
ADC R13, R1
ADC R20, R24
ADC R25, R23
MUL R15, R18
ADD R13, R0
ADC R20, R1
ADC R21, R25
```

```
MUL R16, R19
MOVW R24, R0
MUL R16, R17
ADD R13, R0
ADC R20, R1
ADC R21, R24
ADC R25, R23
MUL R16, R18
MOVW R18,R22
ADD R20, R0
ADC R21, R1
ADC R22, R25
```

```
MUL R2, R7
MOVW R16, R0
MUL R2, R5
MOVW R14, R0
MUL R2, R6
ADD R15, R0
ADC R16, R1
ADC R17, R23

MUL R3, R7
MOVW R24, R0
MUL R3, R5
ADD R15, R0
ADC R16, R1
ADC R17, R24
ADC R25, R23
MUL R3, R6
ADD R16, R0
ADC R17, R1
ADC R18, R25
```

```
MUL R4, R7
MOVW R24, R0
MUL R4, R5
ADD R16, R0
ADC R17, R1
ADC R18, R24
ADC R25, R23
MUL R4, R6
ADD R17, R0
ADC R18, R1
ADC R19, R25
```

48-bit Karatsuba on AVR

```
ADD R8, R11
ADC R9, R12
ADC R10, R13
ADC R11, R20
ADC R12, R21
ADC R13, R22
ADC R23, R23

EOR R26, R27
BRNE add_M

SUB R8, R14
SBC R9, R15
SBC R10, R16
SBC R11, R17
SBC R12, R18
SBC R13, R19
SBCI R23, 0
SBC R24, R24
RJMP final
```

```
add_M:
  ADD R8, R14
  ADC R9, R15
  ADC R10, R16
  ADC R11, R17
  ADC R12, R18
  ADC R13, R19
  CLR R24
  ADC R23, R24
  NOP

final:
  STD Z+3, R8
  STD Z+4, R9
  STD Z+5, R10
  STD Z+6, R11
  STD Z+7, R12
  STD Z+8, R13

  ADD R20, R23
  ADC R21, R24
  ADC R22, R24

  STD Z+9, R20
  STD Z+10, R21
  STD Z+11, R22
```

Larger Karatsuba multiplication

- 48-bit Karatsuba is friendly; everything fits into registers
- Remember that previous speed records were achieved by eliminating loads/stores

Larger Karatsuba multiplication

- 48-bit Karatsuba is friendly; everything fits into registers
- Remember that previous speed records were achieved by eliminating loads/stores
- Karatsuba structure needs additional temporary storage
- Good performance needs careful scheduling and register allocation
- Very important is to compute $\mathbf{H} = H + (l_{k+1}, \ldots, l_{n-1})$ on the fly

Larger Karatsuba multiplication

- 48-bit Karatsuba is friendly; everything fits into registers
- Remember that previous speed records were achieved by eliminating loads/stores
- Karatsuba structure needs additional temporary storage
- Good performance needs careful scheduling and register allocation
- Very important is to compute $\mathbf{H} = H + (l_{k+1}, \ldots, l_{n-1})$ on the fly
- Use 1-level Karatsuba for 48-bit, 64-bit, 80-bit, 96-bit inputs
- Use 2-level Karatsuba for 128-bit, 160-bit, 192-bit inputs
- Use 3-level Karatsuba for 256-bit inputs

Results

Cycle counts for n-bit multiplication

	Input size n							
Approach	48	64	80	96	128	160	192	256
Product scanning:	235	395	595	836	—	—	—	—
Hutter, Wenger, 2011:	—	—	—	—	—	2393	3467	6121
Seo, Kim, 2012:	—	—	—	—	1532	2356	3464	6180
Seo, Kim, 2013:	—	—	—	—	1523	2341	3437	6115
Karatsuba:	217	360	522	780	1325	**1976**	2923	**4797**
— w/o branches:	222	368	533	800	1369	2030	2987	4961

- 160-bit multiplication now $> 18\%$ faster
- 256-bit multiplication now $> 23\%$ faster

From 8-bit to 64-bit processors

Main differences (for us)

- Arithmetic on larger (64-bit) integers

From 8-bit to 64-bit processors

Main differences (for us)

- Arithmetic on larger (64-bit) integers
- Arithmetic on floating-point numbers

From 8-bit to 64-bit processors

Main differences (for us)

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

From 8-bit to 64-bit processors

Main differences (for us)

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

Radix-2^{64} representation

- Let's consider representing 255-bit integers
- Obvious choice: use 4 64-bit integers a_0, a_1, a_2, a_3 with

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

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

Radix-2^{51} representation

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

Radix-2^{51} representation

- 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!)

Radix-2^{51} representation

- 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)

Radix-2^{51} representation

- 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

Radix-2^{51} representation

- 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 $A = 2^{52}$:
  - $(2^{52}, 0, 0, 0, 0)$
  - $(0, 2, 0, 0, 0)$

Radix-2^{51} representation

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

Addition of two bigint255

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

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

Addition of two bigint255

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

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

- This definitely works for reduced inputs

Addition of two bigint255

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

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

- This definitely works for reduced inputs
- This actually works as long as all coefficients are in $[0, \dots, 2^{63} - 1]$

Addition of two `bigint255`

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

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

- This definitely works for reduced inputs
- This actually works as long as all coefficients are in $[0, \ldots, 2^{63} - 1]$
- We can do quite a few additions before we have to carry (reduce)

Subtraction of two bigint255

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

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

- Slightly update our `bigint255` definition to work with *signed* 64-bit integers

Subtraction of two bigint255

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

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

- Slightly update our `bigint255` definition to work with *signed* 64-bit integers
- Reduced if coefficients are in $[-2^{52}+1, 2^{52}-1]$

Carrying in radix-2^{51}

- With many additions, coefficients may grow larger than 63 bits
- They grow even faster with multiplication

Carrying in radix-2^{51}

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

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

Big integers and polynomials

- 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

Using floating-point limbs

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

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

Using floating-point limbs

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

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

- For double-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 52$ "mantissa bits"
  - $e \in \{1,\dots,2046\}$ "exponent"
  - $t = 1023$

Using floating-point limbs

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

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

- For double-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 52$ "mantissa bits"
  - $e \in \{1,\dots,2046\}$ "exponent"
  - $t = 1023$
- For single-precision floats:
  - $s \in \{0,1\}$ "sign bit"
  - $m = 23$ "mantissa bits"
  - $e \in \{1,\dots,254\}$ "exponent"
  - $t = 127$

Using floating-point limbs

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

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

- For double-precision floats:
  - $s \in \{0,1\}$ “sign bit”
  - $m = 52$ “mantissa bits”
  - $e \in \{1,\dots,2046\}$ “exponent”
  - $t = 1023$
- For single-precision floats:
  - $s \in \{0,1\}$ “sign bit”
  - $m = 23$ “mantissa bits”
  - $e \in \{1,\dots,254\}$ “exponent”
  - $t = 127$
- Exponent $= 0$ used to represent 0

Using floating-point limbs

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

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

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

Addition and subtraction

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

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

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

Carrying

- For carrying integers we used a right shift (discard lowest bits)

Carrying

- For carrying integers we used a right shift (discard lowest bits)
- For floating-point numbers we can use multiplication by the inverse of the radix
- Example: Radix 2^{22}, multiply by 2^{-22}
- This does *not* cut off lowest bits, need to round

Carrying

- For carrying integers we used a right shift (discard lowest bits)
- For floating-point numbers we can use multiplication by the inverse of the radix
- Example: Radix 2^{22}, multiply by 2^{-22}
- This does *not* cut off lowest bits, need to round
- Some processors have efficient rounding instructions, e.g., `vroundpd`

Carrying

- For carrying integers we used a right shift (discard lowest bits)
- For floating-point numbers we can use multiplication by the inverse of the radix
- Example: Radix 2^{22}, multiply by 2^{-22}
- This does *not* cut off lowest bits, need to round
- Some processors have efficient rounding instructions, e.g., `vroundpd`
- Otherwise (for double-precision):
  - add constant $2^{52} + 2^{51}$
  - subtract constant $2^{52} + 2^{51}$
  - This will round the number to an integer according to the rounding mode (to nearest, towards zero, away from zero, or truncate)

Modular reduction

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

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 2^16 */
typedef signed long long bigint[16];
```

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

A quick look at product-scanning multiplication

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

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

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-bit intermediate result `r` as follows:

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

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-bit intermediate result `r` as follows:

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

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-bit intermediate result `r` as follows:

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

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

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)
  - $2^{251} - 9$ (Bernstein, Hamburg, Krasnova, Lange, 2013)
  - $2^{448} - 2^{224} - 1$ (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)
  - $2^{251} - 9$ (Bernstein, Hamburg, Krasnova, Lange, 2013)
  - $2^{448} - 2^{224} - 1$ (Hamburg, 2015)
- All these primes come with (more or less) fast reduction algorithms

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)
  - $2^{251} - 9$ (Bernstein, Hamburg, Krasnova, Lange, 2013)
  - $2^{448} - 2^{224} - 1$ (Hamburg, 2015)
- All these primes come with (more or less) fast reduction algorithms
- More about *general primes* later
- For the moment let's stick to $2^{255} - 19$

Carrying after multiplication

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

Carrying after multiplication

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

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

How about squaring?

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

How about squaring?

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

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

How about squaring?

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

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

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

Squaring vs. multiplication

Multiplication needs

- 256 multiplications
- 225 additions

Squaring needs

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

How about other prime fields?

- So far: reductions only modulo “nice” primes
- What if somebody just throws an ugly prime at you?

How about other prime fields?

- So far: reductions only modulo "nice" primes
- What if somebody just throws an ugly prime at you?
- Example: German BSI 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?

- So far: reductions only modulo "nice" primes
- 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 $a \cdot u + b \cdot v = d = \gcd(a, b)$

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

Some notes about the Extended Euclidean algorithm

- Core operation are divisions with remainder
- This lecture: no details about big-integer division
- Version without divisions: **binary extended gcd**:

  Handbook of applied cryptography, Alg. 14.61

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 lecture)
  - Inversion modulo $2^{255}-19$ needs 254 squarings and 11 multiplications in $\mathbb{F}_{2^{255}-19}$

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

```
void gfe_invert(gfe r, const gfe x)
{
gfe z2, z9, z11, z2_5_0, z2_10_0, z2_20_0, z2_50_0, z2_100_0, t;
int i;
/* 2 */                gfe_square(z2,x);
/* 4 */                gfe_square(t,z2);
/* 8 */                gfe_square(t,t);
/* 9 */                gfe_mul(z9,t,x);
/* 11 */               gfe_mul(z11,z9,z2);
/* 22 */               gfe_square(t,z11);
/* 2^5 - 2^0 = 31 */   gfe_mul(z2_5_0,t,z9);
/* 2^6 - 2^1 */        gfe_square(t,z2_5_0);
/* 2^10 - 2^5 */       for (i = 1;i < 5;i++) { gfe_square(t,t); }
/* 2^10 - 2^0 */       gfe_mul(z2_10_0,t,z2_5_0);
/* 2^11 - 2^1 */       gfe_square(t,z2_10_0);
/* 2^20 - 2^10 */      for (i = 1;i < 10;i++) { gfe_square(t,t); }
/* 2^20 - 2^0 */       gfe_mul(z2_20_0,t,z2_10_0);
/* 2^21 - 2^1 */       gfe_square(t,z2_20_0);
/* 2^40 - 2^20 */      for (i = 1;i < 20;i++) { gfe_square(t,t); }
/* 2^40 - 2^0 */       gfe_mul(t,t,z2_20_0);
```

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

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

Multiprecision libraries

- Why would you write low-level arithmetic yourself?
- Aren't there some good libraries for this?

Multiprecision libraries

- Why would you write low-level arithmetic yourself?
- Aren't there some good libraries for this?
- There are:
  - GMP (`http://gmplib.org`), high-performance arithmetic on multiprecision numbers

Multiprecision libraries

- Why would you write low-level arithmetic yourself?
- Aren't there some good libraries for this?
- There are:
  - GMP (http://gmplib.org), high-performance arithmetic on multiprecision numbers
  - NTL (http://shoup.net/ntl/), number-theory library, higher level than GMP, uses GMP

Multiprecision libraries

- Why would you write low-level arithmetic yourself?
- Aren't there some good libraries for this?
- There are:
  - GMP (http://gmplib.org), high-performance arithmetic on multiprecision numbers
  - NTL (http://shoup.net/ntl/), number-theory library, higher level than GMP, uses GMP
  - OpenSSL Bignum (http://openssl.org), low-level routines in OpenSSL

Multiprecision libraries

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

Limitations of libraries

- Libraries don't know the modulus (except for $\texttt{mp}\mathbb{F}_q$), cannot optimize for a fixed modulus

Limitations of libraries

- Libraries don't know the modulus (except for $\texttt{mp}\mathbb{F}_q$), cannot optimize for a fixed modulus
- Libraries don't know the sequence of field operations you're computing (e.g., point addition), cannot use lazy reduction

Limitations of libraries

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

Limitations of libraries

- Libraries don't know the modulus (except for $\texttt{mp}\mathbb{F}_q$), cannot optimize for a fixed modulus
- Libraries don't know the sequence of field operations you're computing (e.g., point addition), cannot use lazy reduction
- Libraries are not always timing-attack protected
- Consequence: ECC speed records are achieved with hand-optimized assembly implementations