Multiprecision arithmetic

Peter Schwabe

October 20, 2014

SPACE 2014, Pune, India

Multiprecision arithmetic

(from primary school to Asiacrypt)

Peter Schwabe

October 20, 2014

SPACE 2014, Pune, India

PART I

Joint work with Michael Hutter

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

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$
$$4 + 8 = \quad ?$$

Still in the first year of primary school

Crossing the ten barrier

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$
$$4 + 8 = \quad ?$$

- 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

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$
$$4 + 8 = \quad ?$$

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

Still in the first year of primary school

Crossing the ten barrier

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$
$$4 + 8 = \quad ?$$

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

What happens with the carry?

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

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

- 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

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

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

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

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

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

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

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

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)

Why do we do all that again?

- Mainly asymmetric cryptography heavily relies on multiprecision arithmetic
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers

Why do we do all that again?

- Mainly asymmetric cryptography heavily relies on multiprecision arithmetic
- 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, 414 bits ...

Why do we do all that again?

- Mainly asymmetric cryptography heavily relies on multiprecision arithmetic
- 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, 414 bits ...
- Example 3:
  - Genus-2 hyperelliptic curves defined over finite fields
  - Typically use HEC over large-characteristic prime fields
  - Field size for 128-bit security: 128 bits

Why do we do all that again?

- Mainly asymmetric cryptography heavily relies on multiprecision arithmetic
- 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, 414 bits ...
- Example 3:
  - Genus-2 hyperelliptic curves defined over finite fields
  - Typically use HEC over large-characteristic prime fields
  - Field size for 128-bit security: 128 bits
- For now mainly interested in 160-bit and 256-bit arithmetic

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

- Add 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 2 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", 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
```

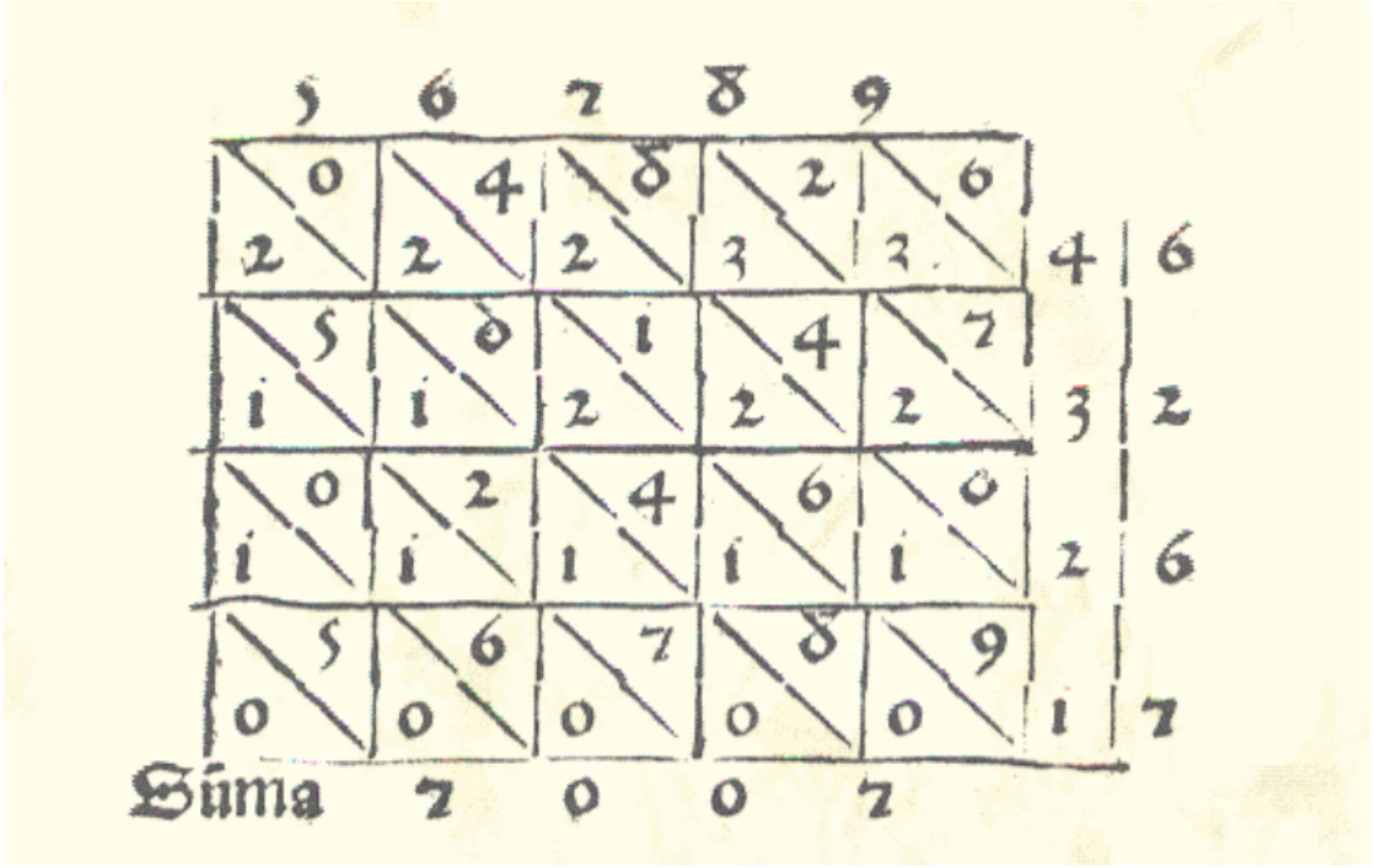
```
MUL R2, R9
ADD R14, R0
ADC R15, R1
ADC R16, R5
MUL R3, R8
ADD R14, R0
ADC R15, R1
ADC R16, R5
MUL R4, R7
ADD R14, R0
ADC R15, R1
ADC R16, R5
STD Z+2, R14
CLR R17
```

```
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

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

Multiplication complexity

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

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

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

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

- 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

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

- 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 $k = n/2$

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 $M = (A_\ell + A_h) \cdot (B_\ell + B_h) \hat{=} (m_0, \ldots, m_n)$

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 $L = A_\ell \cdot B_\ell \hat{=} (\ell_0, \ldots, \ell_{n-1})$
- Compute $H = A_h \cdot B_h \hat{=} (h_0, \ldots, h_{n-1})$
- Compute $M = |A_\ell - A_h| \cdot |B_\ell - B_h| \hat{=} (m_0, \ldots, m_{n-1})$
- Set $t = 0$, if $M = (A_\ell - A_h) \cdot (B_\ell - B_h)$; $t = 1$ otherwise
- Compute $\hat{M} = (-1)^t M = (A_\ell - A_h)(B_\ell - B_h)$ $\hat{=} (\hat{m}_0, \ldots, \hat{m}_{n-1})$
- 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 values 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 values 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 values with this condition byte
- Don't negate the condition byte
- Subtract the condition byte (`0xff` or `0x00` from all bytes)
- Saves two `NEG` instructions

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

Resources online

Paper:
Michael Hutter, Peter Schwabe. *"Multiprecision multiplication on AVR revisited"*.
http://cryptojedi.org/papers/#avrmul

Software: http://cryptojedi.org/crypto/#avrmul

Resources online

Paper:
Michael Hutter, Peter Schwabe. *"Multiprecision multiplication on AVR revisited"*.
http://cryptojedi.org/papers/#avrmul

Software: http://cryptojedi.org/crypto/#avrmul

Code generator for operand-scanning, product-scanning, hybrid, and operand-caching by Hutter and Wenger:
http://mhutter.org/research/avr/index.htm#mulopcache

PART II

Joint work with Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange

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
- Arithmetic on *vectors*

From 8-bit to 64-bit processors

Main differences (for us)

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

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 $[0, \ldots, 2^{63} - 1]$

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 $\mathbb{Z}[x]$: 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 $\mathbb{Z}[x]$: 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 $\mathbb{Z}[x]$: 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

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

Using floating-point limbs

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

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

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

Using floating-point limbs

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

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

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

Using floating-point limbs

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

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

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

Using floating-point limbs

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

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

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

Addition and subtraction

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

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

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

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)

Vector arithmetic

- Most modern processors have *vector units*
- Work on vectors of floats or ints instead of scalars
- Much larger computational power, e.g., Intel Nehalem:

Vector arithmetic

- Most modern processors have *vector units*
- Work on vectors of floats or ints instead of scalars
- Much larger computational power, e.g., Intel Nehalem:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle

Vector arithmetic

- Most modern processors have *vector units*
- Work on vectors of floats or ints instead of scalars
- Much larger computational power, e.g., Intel Nehalem:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - $4\times$ 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle

Vector arithmetic

- Most modern processors have *vector units*
- Work on vectors of floats or ints instead of scalars
- Much larger computational power, e.g., Intel Nehalem:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - $4\times$ 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle
- **Vector instructions are almost as fast as scalar instructions but do $4\times$ the work**

Vector arithmetic

- Most modern processors have *vector units*
- Work on vectors of floats or ints instead of scalars
- Much larger computational power, e.g., Intel Nehalem:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - 4× 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle
- **Vector instructions are almost as fast as scalar instructions but do 4× the work**
- Some things are cost extra:
  - Variably indexed loads (lookups) into vectors
  - Data-dependent branches are expensive in SIMD
  - Shuffeling entries in vectors

Vectorizing EC scalar multiplication

Computing multiple scalar multiplications

- Changes the rules of the game
- Increases size of active data set

Vectorizing EC scalar multiplication

Computing multiple scalar multiplications

- Changes the rules of the game
- Increases size of active data set

Parallelism inside multiprecision arithmetic

- Addition (in redundant representation) is trivially vectorized
- Vectorizing multiplication needs many shuffles
- Vectorization "eats up" instruction-level parallelism

Vectorizing EC scalar multiplication

Computing multiple scalar multiplications

- Changes the rules of the game
- Increases size of active data set

Parallelism inside multiprecision arithmetic

- Addition (in redundant representation) is trivially vectorized
- Vectorizing multiplication needs many shuffles
- Vectorization "eats up" instruction-level parallelism

Parallelism inside EC arithmetic

- Vectorize independent multiplications in EC addition
- May still need some shuffles (after each block of operations)
- Efficiency depends on EC formulas

Example: Montgomery ladder

function ladderstep($x_{Q-P}, X_P, Z_P, X_Q, Z_Q$)

$t_1 \leftarrow X_P + Z_P$

$t_6 \leftarrow t_1^2$

$t_2 \leftarrow X_P - Z_P$

$t_7 \leftarrow t_2^2$

$t_5 \leftarrow t_6 - t_7$

$t_3 \leftarrow X_Q + Z_Q$

$t_4 \leftarrow X_Q - Z_Q$

$t_8 \leftarrow t_4 \cdot t_1$

$t_9 \leftarrow t_3 \cdot t_2$

$X_{P+Q} \leftarrow (t_8 + t_9)^2$

$Z_{P+Q} \leftarrow x_{Q-P} \cdot (t_8 - t_9)^2$

$X_{[2]P} \leftarrow t_6 \cdot t_7$

$Z_{[2]P} \leftarrow t_5 \cdot (t_7 + ((A+2)/4) \cdot t_5)$

return $(X_{[2]P}, Z_{[2]P}, X_{P+Q}, Z_{P+Q})$

end function

Example: Montgomery ladder

function ladderstep($x_{Q-P}, X_P, Z_P, X_Q, Z_Q$)

$t_1 \leftarrow X_P + Z_P; t_2 \leftarrow X_P - Z_P; t_3 \leftarrow X_Q + Z_Q; t_4 \leftarrow X_Q - Z_Q$

$t_6 \leftarrow t_1 \cdot t_1; t_7 \leftarrow t_2 \cdot t_2; t_8 \leftarrow t_4 \cdot t_1; t_9 \leftarrow t_3 \cdot t_2$

$t_{10} \leftarrow ((A+2)/4) \cdot t_6$
$t_{11} \leftarrow ((A+2)/4 - 1) \cdot t_7$

$t_5 \leftarrow t_6 - t_7; t_4 \leftarrow t_{10} - t_{11}; t_1 \leftarrow t_8 - t_9; t_0 \leftarrow t_8 + t_9$

$Z_{[2]P} \leftarrow t_5 \cdot t_4; X_{P+Q} \leftarrow t_0^2; X_{[2]P} \leftarrow t_6 \cdot t_7; t_2 \leftarrow t_1 \cdot t_1$

$Z_{P+Q} \leftarrow x_{Q-P} \cdot t_2$

return $(X_{[2]P}, Z_{[2]P}, X_{P+Q}, Z_{P+Q})$
end function

A better candidate: Kummer surfaces

- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation

A better candidate: Kummer surfaces

- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation
- Easier way to think about it:
  - Group modulo negation
  - Map from group to Kummer surface by rational map X
  - Elements represented projectively as $(x : y : z : t)$
  - $(x : y : z : t) = (rx : ry : rz : rt)$ for any $r \neq 0$
  - Efficient doubling and efficient *differential addition*

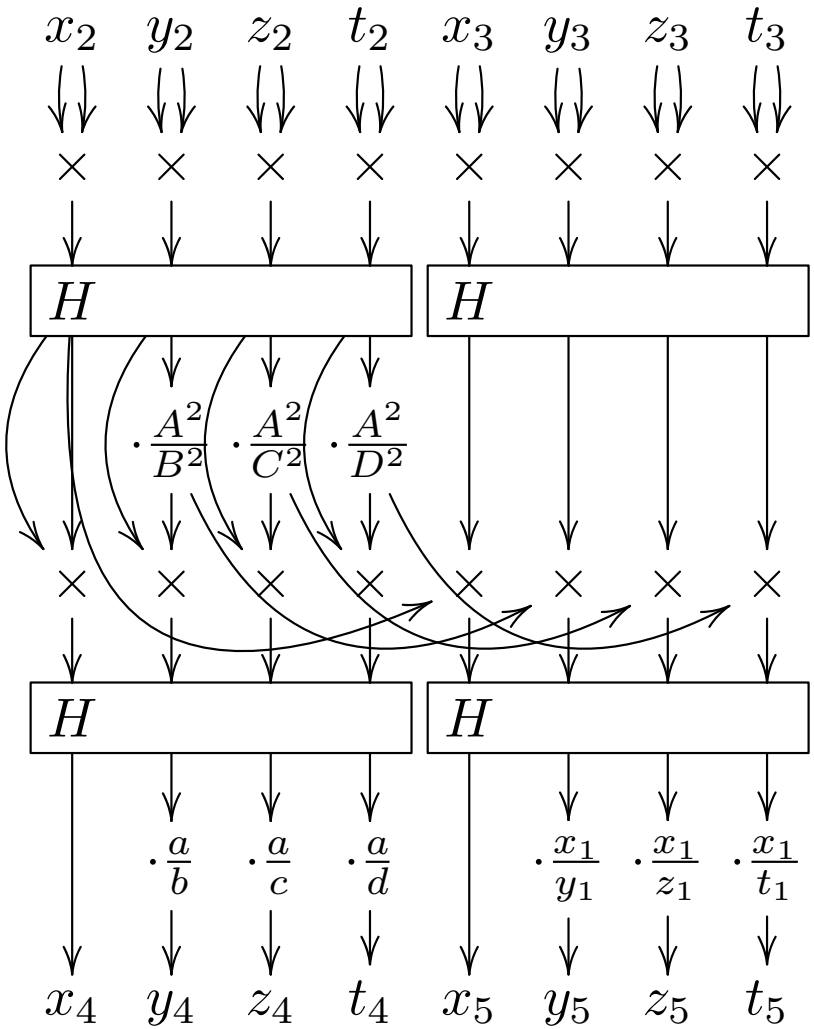
A better candidate: Kummer surfaces

- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation
- Easier way to think about it:
  - Group modulo negation
  - Map from group to Kummer surface by rational map X
  - Elements represented projectively as $(x : y : z : t)$
  - $(x : y : z : t) = (rx : ry : rz : rt)$ for any $r \neq 0$
  - Efficient doubling and efficient *differential addition*
- Ladderstep: gets as input $X(P) = (x_2 : y_2 : z_2 : t_2)$, $X(Q) = (x_3 : y_3 : z_3 : t_3)$, and $X(Q - P) = (x_1 : y_1 : z_1 : t_1)$
  - Computes $X(2P) = (x_4 : y_4 : z_4 : t_4)$
  - Computes $X(P + Q) = (x_5 : y_5 : z_5 : t_5)$

A better candidate: Kummer surfaces

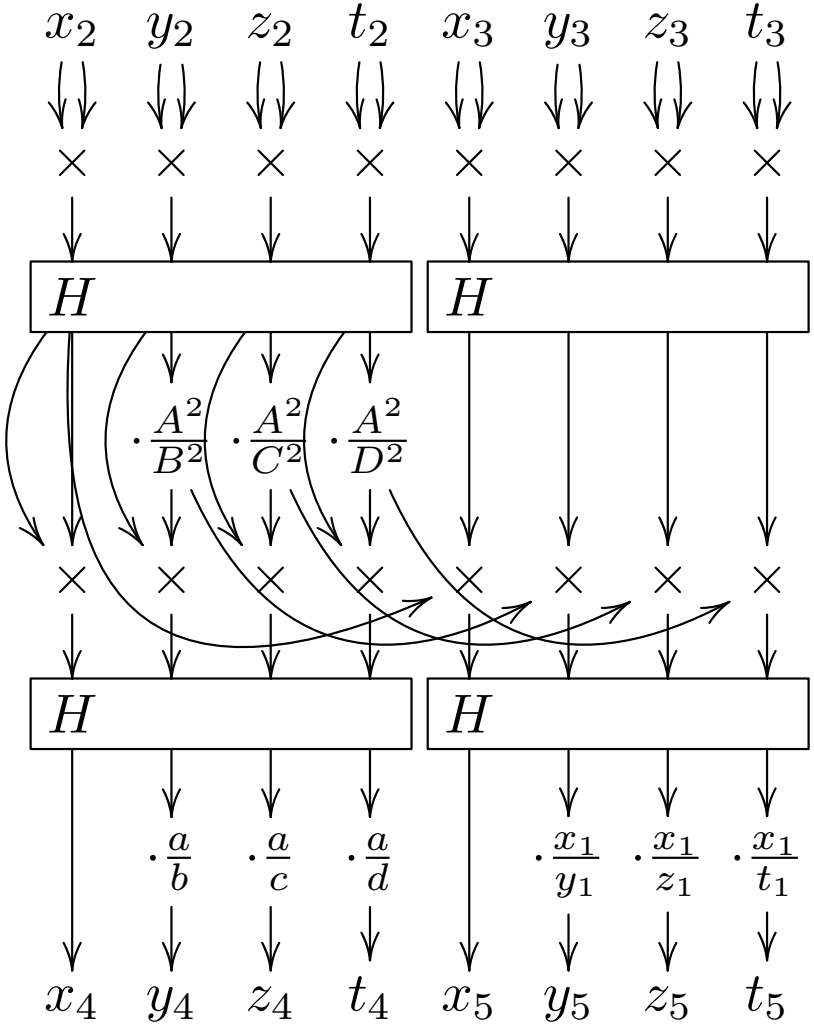
- Think of a Kummer surface as the Jacobian of a hyperelliptic curve modulo negation
- Easier way to think about it:
  - Group modulo negation
  - Map from group to Kummer surface by rational map X
  - Elements represented projectively as $(x : y : z : t)$
  - $(x : y : z : t) = (rx : ry : rz : rt)$ for any $r \neq 0$
  - Efficient doubling and efficient *differential addition*
- Ladderstep: gets as input $X(P) = (x_2 : y_2 : z_2 : t_2)$, $X(Q) = (x_3 : y_3 : z_3 : t_3)$, and $X(Q - P) = (x_1 : y_1 : z_1 : t_1)$
  - Computes $X(2P) = (x_4 : y_4 : z_4 : t_4)$
  - Computes $X(P + Q) = (x_5 : y_5 : z_5 : t_5)$
- Coordinates are elements of a (large) finite field
- Efficient Kummer surface arithmetic wants efficient multiprecision arithmetic

Arithmetic on the Kummer surface

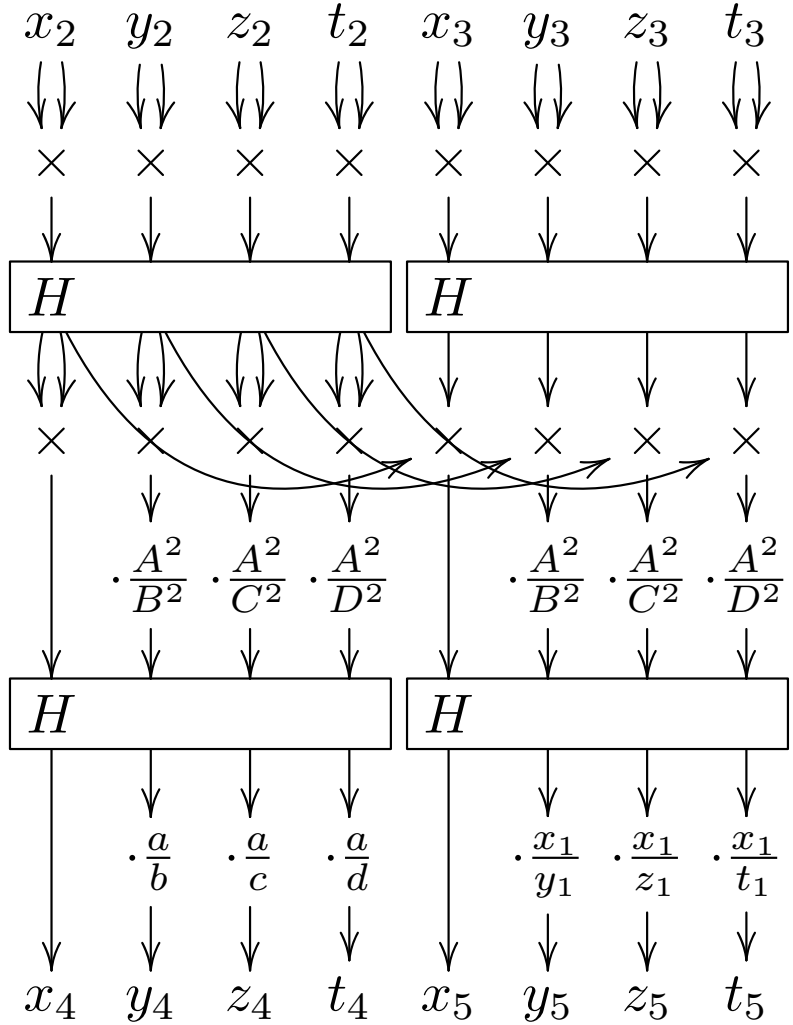


$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas

Arithmetic on the Kummer surface



$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas

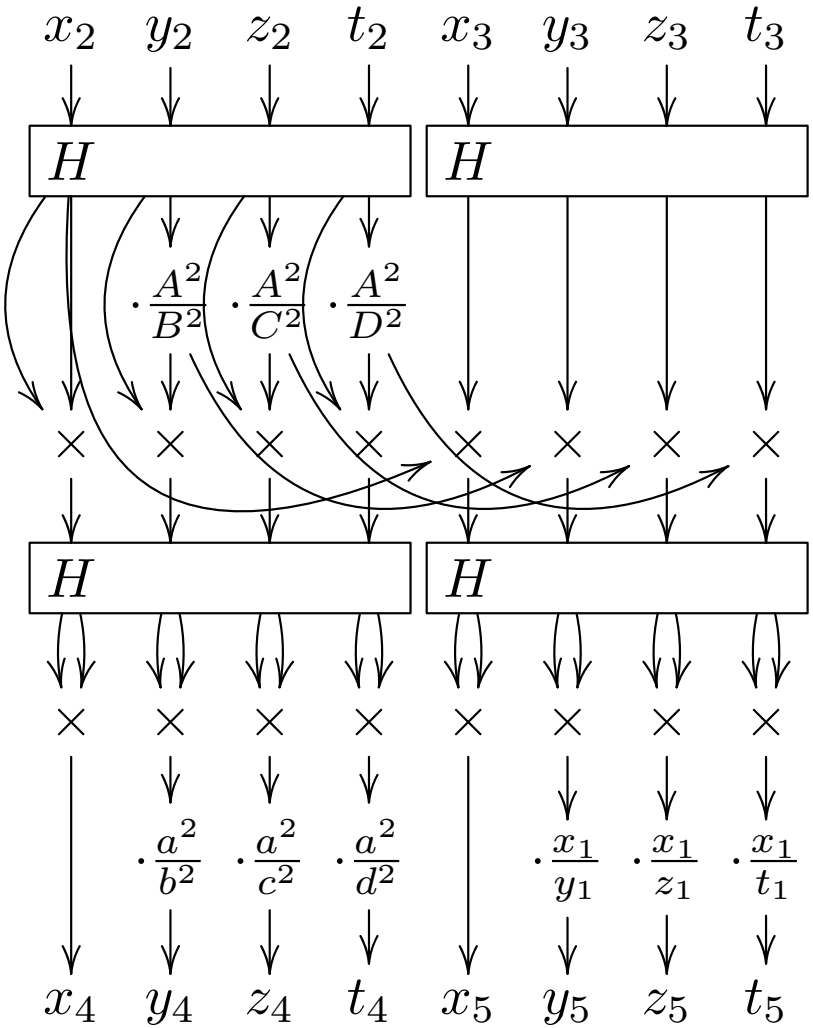


$7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ ladder formulas

The "squared Kummer surface"

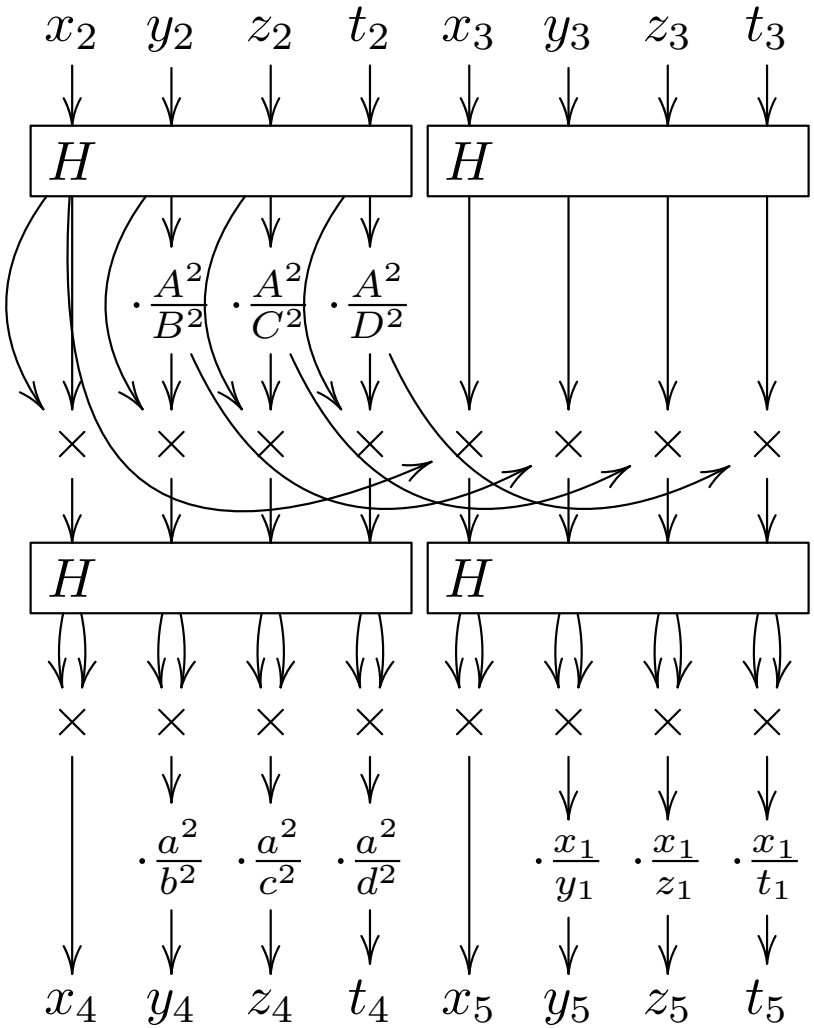
- In fact, we use arithmetic on a different, "squared" surface
- Each point $(x : y : z : t)$ on the original surface corresponds to $(x^2 : y^2 : z^2 : t^2)$ on the squared surface
- No operation-count advantages
- Easier to construct squared surface with small constants
- In the following rename $(x^2 : y^2 : z^2 : t^2)$ to $(x : y : z : t)$

Arithmetic on the squared Kummer surface

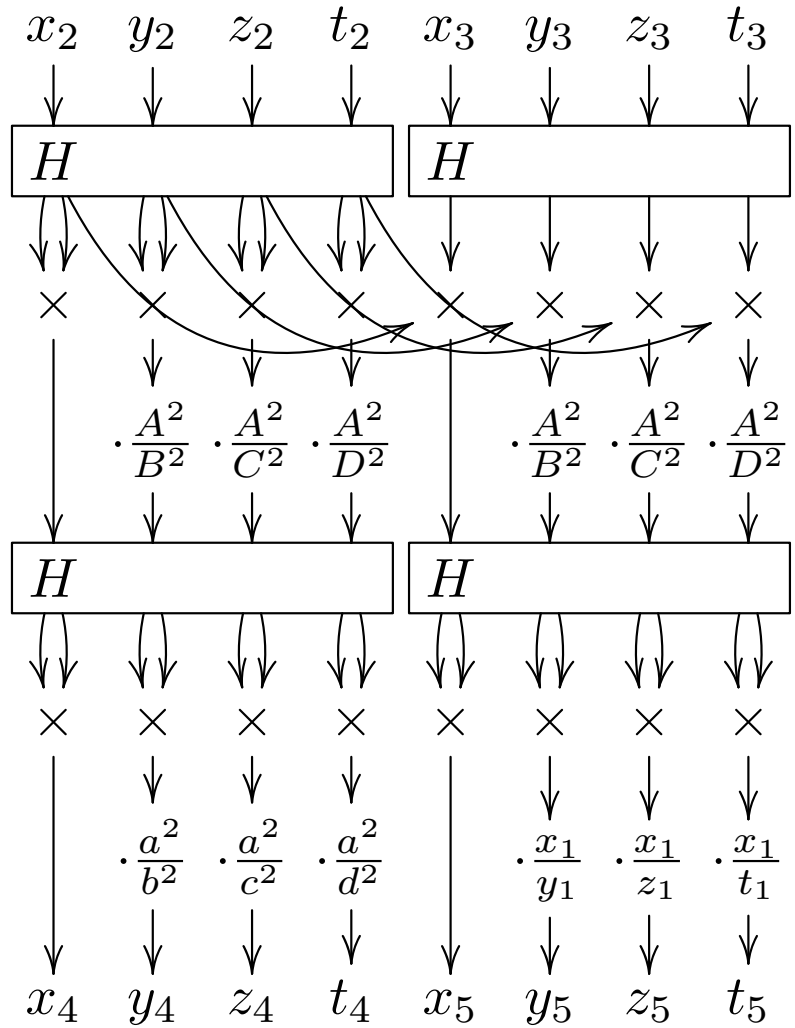


$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas

Arithmetic on the squared Kummer surface

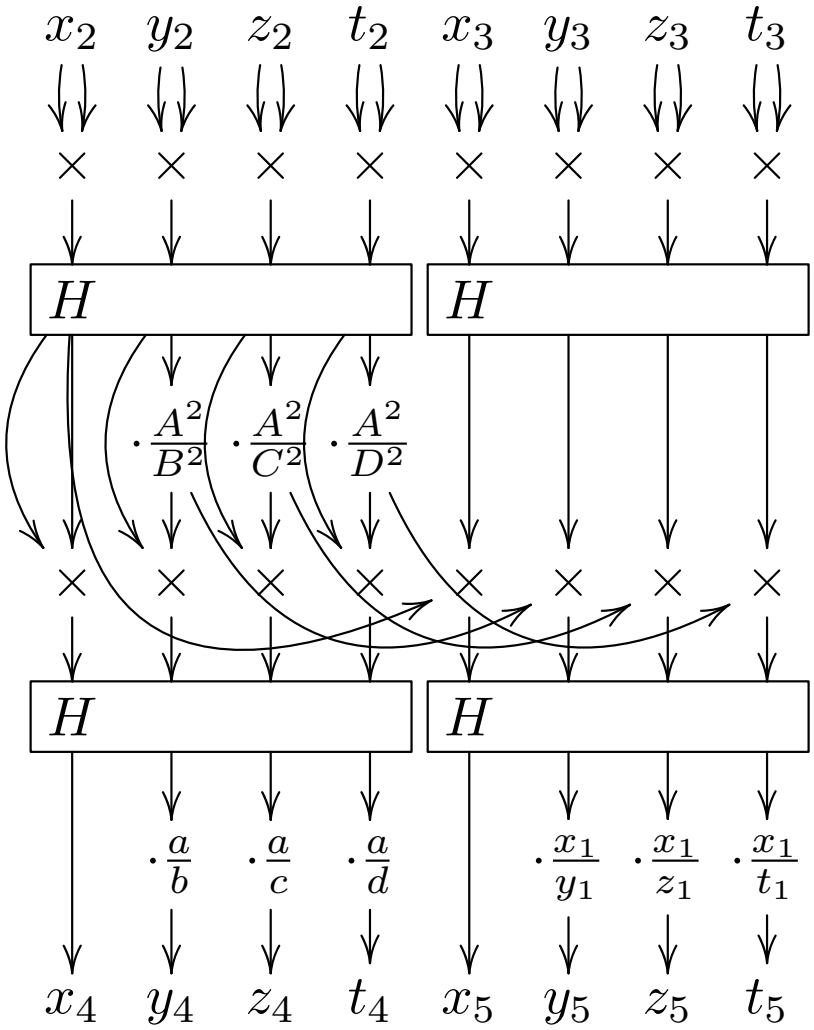


$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas

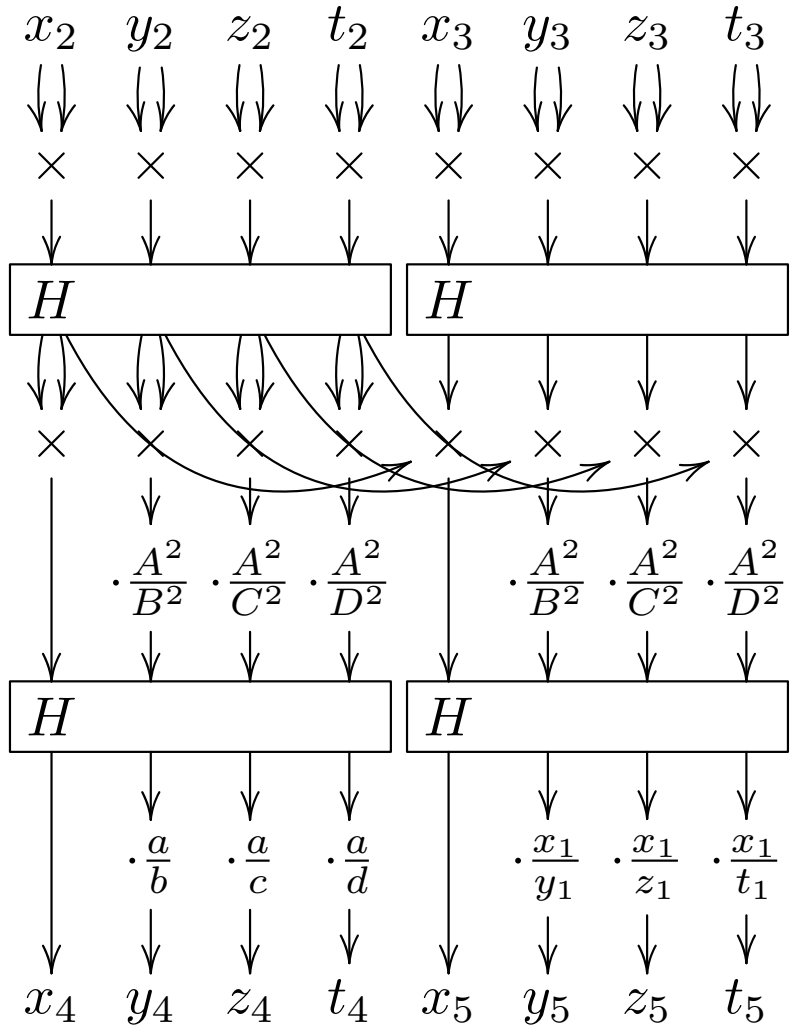


$7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ ladder formulas

Arithmetic on the (original) Kummer surface



$10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas



$7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ ladder formulas

A suitable Kummer surface

- Formulas for efficient Kummer surface arithmetic known for a while
  - Originally proposed by Chudnovsky, Chudnovsky, 1986
  - $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ formulas by Gaudry, 2006
  - $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ formulas by Bernstein, 2006

A suitable Kummer surface

- Formulas for efficient Kummer surface arithmetic known for a while
  - Originally proposed by Chudnovsky, Chudnovsky, 1986
  - $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ formulas by Gaudry, 2006
  - $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ formulas by Bernstein, 2006
- Problem: find cryptographically secure surface with small constants

A suitable Kummer surface

- Formulas for efficient Kummer surface arithmetic known for a while
  - Originally proposed by Chudnovsky, Chudnovsky, 1986
  - $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ formulas by Gaudry, 2006
  - $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ formulas by Bernstein, 2006
- Problem: find cryptographically secure surface with small constants
- Gaudry, Schost, 2012: suitable (squared) surface:
  - Defined over the field $\mathbb{F}_{2^{127}-1}$
  - $(1 : a^2/b^2 : a^2/c^2 : a^2/d^2) = (-114 : 57 : 66 : 418)$
  - $(1 : A^2/B^2 : A^2/C^2 : A^2/D^2) = (-833 : 2499 : 1617 : 561)$

A suitable Kummer surface

- Formulas for efficient Kummer surface arithmetic known for a while
  - Originally proposed by Chudnovsky, Chudnovsky, 1986
  - $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ formulas by Gaudry, 2006
  - $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ formulas by Bernstein, 2006
- Problem: find cryptographically secure surface with small constants
- Gaudry, Schost, 2012: suitable (squared) surface:
  - Defined over the field $\mathbb{F}_{2^{127}-1}$
  - $(1 : a^2/b^2 : a^2/c^2 : a^2/d^2) = (-114 : 57 : 66 : 418)$
  - $(1 : A^2/B^2 : A^2/C^2 : A^2/D^2) = (-833 : 2499 : 1617 : 561)$
- Finding this surface cost 1000000 CPU hours
- The same surface has been used by Bos, Costello, Hisil, and Lauter (Eurocrypt 2013)

Sandy Bridge/Ivy Bridge

- Pre-latest microarchitectures by Intel
- Very powerful vector instructions: AVX
- Operations on 256-bit vector registers

Sandy Bridge/Ivy Bridge

- Pre-latest microarchitectures by Intel
- Very powerful vector instructions: AVX
- Operations on 256-bit vector registers
- Can only do vectors of (single- or double-precision) floats
- Performance:
  - 1 vectorized double-precision multiplication per cycle, plus
  - 1 vectorized double-precision addition per cycle
  - A total of 8 double-precision operations per cycle!

Representing elements of $\mathbb{F}_{2^{127}-1}$

- Represent an element A in radix-$2^{127/6}$
- Write A as $a_0, a_1, a_2, a_3, a_4, a_5$, where
  - a_0 is a small multiple of 2^0
  - a_1 is a small multiple of 2^{22}
  - a_2 is a small multiple of 2^{43}
  - a_3 is a small multiple of 2^{64}
  - a_4 is a small multiple of 2^{85}
  - a_5 is a small multiple of 2^{106}

Multiplication

- Consider multiplication of A and B with reduction mod $2^{127} - 1$
- Make use of the fact that $2^{127} \equiv 1$
- With radix $2^{127/6}$ we obtain:

$$
\begin{aligned}
r_0 &= a_0b_0 + 2^{-127}a_1b_5 + 2^{-127}a_2b_4 + 2^{-127}a_3b_3 + 2^{-127}a_4b_2 + 2^{-127}a_5b_1 \\
r_1 &= a_0b_1 + a_1b_0 + 2^{-127}a_2b_5 + 2^{-127}a_3b_4 + 2^{-127}a_4b_3 + 2^{-127}a_5b_2 \\
r_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 2^{-127}a_3b_5 + 2^{-127}a_4b_4 + 2^{-127}a_5b_3 \\
r_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 2^{-127}a_4b_5 + 2^{-127}a_5b_4 \\
r_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 2^{-127}a_5b_5 \\
r_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0
\end{aligned}
$$

Multiplication

- Consider multiplication of A and B with reduction mod $2^{127} - 1$
- Make use of the fact that $2^{127} \equiv 1$
- With radix $2^{127/6}$ we obtain:

$$
\begin{aligned}
r_0 &= a_0b_0 + 2^{-127}a_1b_5 + 2^{-127}a_2b_4 + 2^{-127}a_3b_3 + 2^{-127}a_4b_2 + 2^{-127}a_5b_1 \\
r_1 &= a_0b_1 + a_1b_0 + 2^{-127}a_2b_5 + 2^{-127}a_3b_4 + 2^{-127}a_4b_3 + 2^{-127}a_5b_2 \\
r_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 2^{-127}a_3b_5 + 2^{-127}a_4b_4 + 2^{-127}a_5b_3 \\
r_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 2^{-127}a_4b_5 + 2^{-127}a_5b_4 \\
r_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 2^{-127}a_5b_5 \\
r_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0
\end{aligned}
$$

- Obviously, we always perform this whole thing $4\times$ in parallel

Multiplication

- Consider multiplication of A and B with reduction mod $2^{127} - 1$
- Make use of the fact that $2^{127} \equiv 1$
- With radix $2^{127/6}$ we obtain:

$$
\begin{aligned}
r_0 &= a_0b_0 + 2^{-127}a_1b_5 + 2^{-127}a_2b_4 + 2^{-127}a_3b_3 + 2^{-127}a_4b_2 + 2^{-127}a_5b_1 \\
r_1 &= a_0b_1 + a_1b_0 + 2^{-127}a_2b_5 + 2^{-127}a_3b_4 + 2^{-127}a_4b_3 + 2^{-127}a_5b_2 \\
r_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 2^{-127}a_3b_5 + 2^{-127}a_4b_4 + 2^{-127}a_5b_3 \\
r_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 2^{-127}a_4b_5 + 2^{-127}a_5b_4 \\
r_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 2^{-127}a_5b_5 \\
r_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0
\end{aligned}
$$

- Obviously, we always perform this whole thing $4\times$ in parallel
- Obviously, we specialize squaring

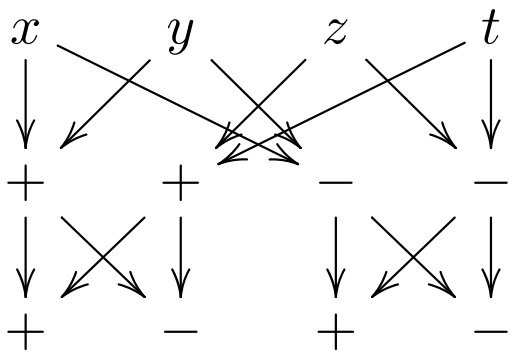
Multiplication

- Consider multiplication of A and B with reduction mod $2^{127} - 1$
- Make use of the fact that $2^{127} \equiv 1$
- With radix $2^{127/6}$ we obtain:

$$
\begin{aligned}
r_0 &= a_0b_0 + 2^{-127}a_1b_5 + 2^{-127}a_2b_4 + 2^{-127}a_3b_3 + 2^{-127}a_4b_2 + 2^{-127}a_5b_1 \\
r_1 &= a_0b_1 + a_1b_0 + 2^{-127}a_2b_5 + 2^{-127}a_3b_4 + 2^{-127}a_4b_3 + 2^{-127}a_5b_2 \\
r_2 &= a_0b_2 + a_1b_1 + a_2b_0 + 2^{-127}a_3b_5 + 2^{-127}a_4b_4 + 2^{-127}a_5b_3 \\
r_3 &= a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + 2^{-127}a_4b_5 + 2^{-127}a_5b_4 \\
r_4 &= a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 + 2^{-127}a_5b_5 \\
r_5 &= a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0
\end{aligned}
$$

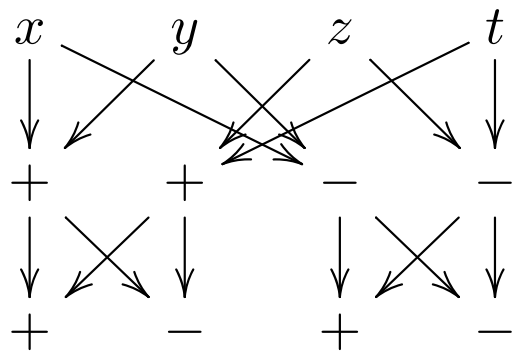
- Obviously, we always perform this whole thing $4\times$ in parallel
- Obviously, we specialize squaring
- Obviously, we specialize multiplications by small constants

The Hadamard transform



- Only shuffeling operation in Kummer arithmetic
- AVX has limited shuffeling across left and right half
- Plain Hadamard turns out to be expensive

The Hadamard transform

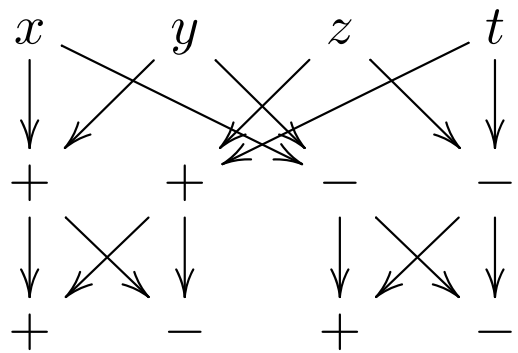


- Only shuffeling operation in Kummer arithmetic
- AVX has limited shuffeling across left and right half
- Plain Hadamard turns out to be expensive

Permuted and negated Hadamard

- Allow generalized Hadamard to output permuted vector
- Self-inverting permutation "cleans" after two generalized Hadamards

The Hadamard transform

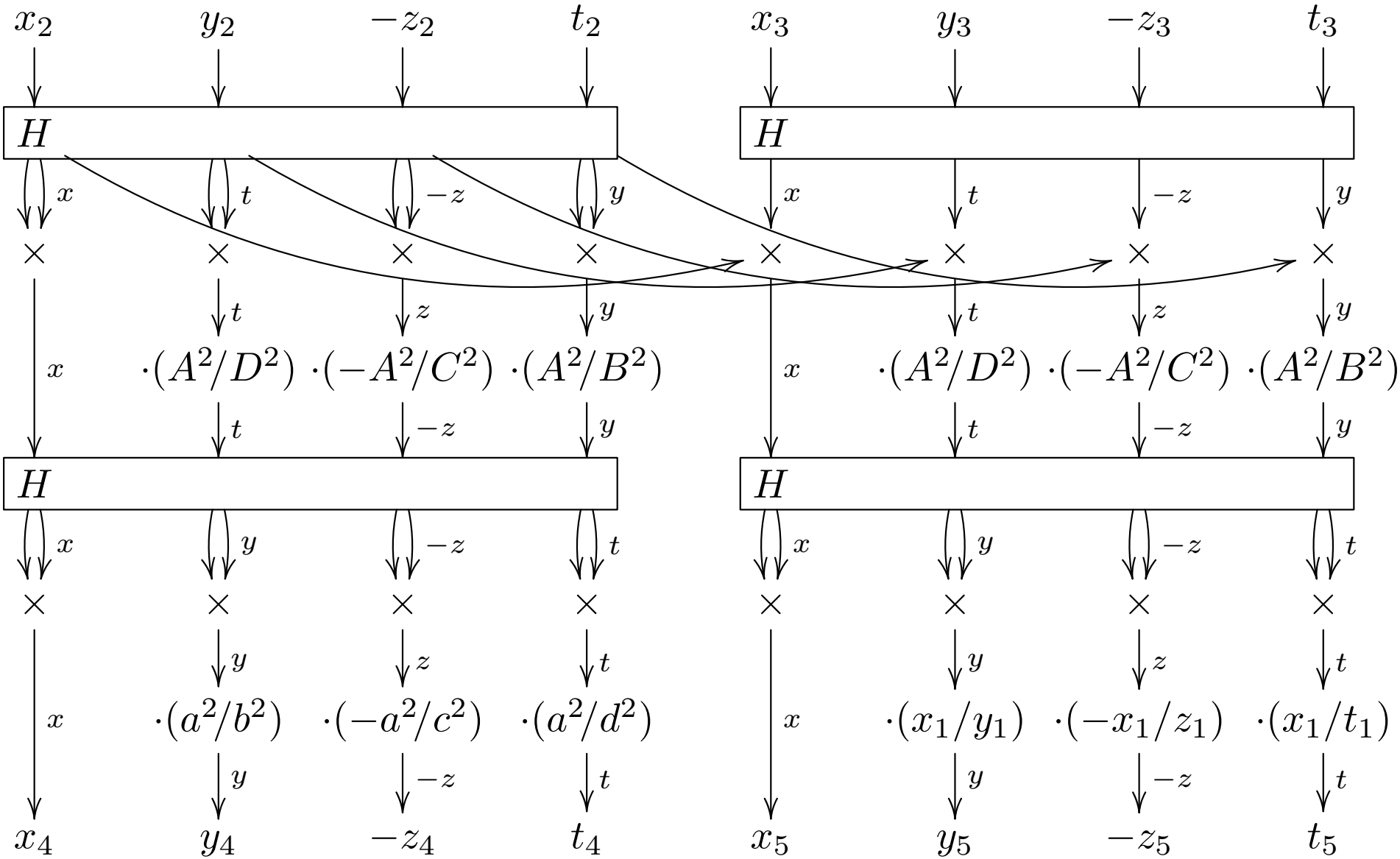


- Only shuffeling operation in Kummer arithmetic
- AVX has limited shuffeling across left and right half
- Plain Hadamard turns out to be expensive

Permuted and negated Hadamard

- Allow generalized Hadamard to output permuted vector
- Self-inverting permutation "cleans" after two generalized Hadamards
- Allow generalized Hadamard to negate vector entries
- "Clean" negations by multiplication by negated constants

Arithmetic on the squared Kummer surface



Results

128-bit secure, constant-time scalar multiplication

arch	cycles	open	g	source of software
Sandy	194036	yes	1	Bernstein–Duif–Lange–Schwabe–Yang CHES 2011
Sandy	153000?	no	1	Hamburg
Sandy	137000?	no	1	Longa–Sica Asiacrypt 2012
Sandy	122716	yes	2	Bos–Costello–Hisil–Lauter Eurocrypt 2013
Sandy	119904	yes	1	Oliveira–López–Aranha–Rodríguez-Henríquez CHES 2013
Sandy	96000?	no	1	Faz-Hernández–Longa–Sánchez CT-RSA 2014
Sandy	92000?	no	1	Faz-Hernández–Longa–Sánchez July 2014
Sandy	88916	yes	2	**new (our results)**

Results

128-bit secure, constant-time scalar multiplication

arch	cycles	open	g	source of software
Ivy	182708	yes	1	Bernstein–Duif–Lange–Schwabe–Yang CHES 2011
Ivy	145000?	yes	1	Costello–Hisil–Smith Eurocrypt 2014
Ivy	119032	yes	2	Bos–Costello–Hisil–Lauter Eurocrypt 2013
Ivy	114036	yes	1	Oliveira–López–Aranha–Rodríguez-Henríquez CHES 2013
Ivy	92000?	no	1	Faz-Hernández–Longa–Sánchez CT-RSA 2014
Ivy	89000?	no	1	Faz-Hernández–Longa–Sánchez July 2014
Ivy	88448	yes	2	**new (our results)**

More results

Also optimized for Intel Haswell

arch	cycles	open	g	source of software
Haswell	145907	yes	1	Bernstein–Duif–Lange–Schwabe–Yang CHES 2011
Haswell	100895	yes	2	Bos–Costello–Hisil–Lauter Eurocrypt 2013
Haswell	55595	no	1	Oliveira–López–Aranha–Rodríguez-Henríquez CHES 2013
Haswell	54389	yes	2	**new (our results)**

Even more results

Also optimized for ARM Cortex-A8

arch	cycles	open	g	source of software
A8-slow	497389	yes	1	Bernstein–Schwabe CHES 2012
A8-slow	305395	yes	2	**new (our result)**
A8-fast	460200	yes	1	Bernstein–Schwabe CHES 2012
A8-fast	273349	yes	2	**new (our result)**

Resources online

Paper:
Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, Peter Schwabe. *"Kummer strikes back: new DH speed records"*.
To be presented at Asiacrypt 2014
http://cryptojedi.org/papers/#kummer

Software:
Included in SUPERCOP, subdirectory `crypto_scalarmult/kummer/`
http://bench.cr.yp.to/supercop.html