FourℚNEON: Faster Elliptic Curve Scalar Multiplications on ARM Processors

Selected Areas in Cryptography (SAC 2016)
St. Johns, Canada

Patrick Longa
Microsoft Research

Next-generation elliptic curves

- Recent effort to propose and deploy new elliptic curves for cryptography.
  E.g., Curve25519 (Bernstein) and Curve448 (Hamburg) recently adopted by IETF for use in the TLS protocol

Next-generation elliptic curves

- Recent effort to propose and deploy new elliptic curves for cryptography.
  E.g., Curve25519 (Bernstein) and Curve448 (Hamburg) recently adopted by IETF for use in the TLS protocol

Motivations:

1. Regain confidence and public acceptance after Snowden revelations

Next-generation elliptic curves

- Recent effort to propose and deploy new elliptic curves for cryptography.
  E.g., Curve25519 (Bernstein) and Curve448 (Hamburg) recently adopted by IETF for use in the TLS protocol

Motivations:

1. Regain confidence and public acceptance after Snowden revelations
2. Take advantage of state-of-the-art ECC algorithms with **improved implementation security** and **better performance**:
    - new curve models
    - faster scalar multiplication algorithms
    - faster finite fields
    - improved side-channel resistance

State-of-the-art ECC: Four$\mathbb{Q}$

- New elliptic curve recently proposed by Costello-Longa (ASIACRYPT 2015)

State-of-the-art ECC: Four$\mathbb{Q}$

- New elliptic curve recently proposed by Costello-Longa (ASIACRYPT 2015)
- Four$\mathbb{Q}$ uniquely combines state-of-the-art techniques in ECC:

- CM endomorphism [GLV01], Frobenius ($\mathbb{Q}$-curve) endomorphism [GLS09, Smi16, GI13]

- Edwards form [Edw07], efficient Edwards coordinates [BBJ+08, HCW+08]

- Arithmetic over the Mersenne prime $p = 2^{127} - 1$

State-of-the-art ECC: Four$\mathbb{Q}$

- New elliptic curve recently proposed by Costello-Longa (ASIACRYPT 2015)
- Four$\mathbb{Q}$ uniquely combines state-of-the-art techniques in ECC:

- CM endomorphism [GLV01], Frobenius ($\mathbb{Q}$-curve) endomorphism [GLS09, Smi16, GI13]
- Edwards form [Edw07], efficient Edwards coordinates [BBJ+08, HCW+08]
- Arithmetic over the Mersenne prime $p = 2^{127} - 1$

- Relevant features for next-generation ECC:

1. Uniqueness: only curve at the 128-bit security level with desired properties
2. Support for secure implementations and top performance

Implementation security

- The original Four$\mathbb{Q}$ paper describes a constant-time, exception-free implementation

Implementation security

- The original Four$\mathbb{Q}$ paper describes a constant-time, exception-free implementation
- The Four$\mathbb{Q}$lib library (http://research.microsoft.com/en-us/projects/fourqlib/) supports *three* core scalar multiplication operations:
  - $[k]P$, variable-base
  - $[k]P$, fixed-base
  - $[k]P + l[Q]$, double-scalar

Implementation security

- The original Four$\mathbb{Q}$ paper describes a constant-time, exception-free implementation
- The Four$\mathbb{Q}$lib library (http://research.microsoft.com/en-us/projects/fourqlib/) supports *three* core scalar multiplication operations:
  - $[k]P$, variable-base
  - $[k]P$, fixed-base
  - $[k]P + l[Q]$, double-scalar
- Operations are protected against timing attacks, cache attacks, exception attacks, invalid curve attacks and small subgroup attacks

Performance

Compared against other ECC alternatives:

$$\frac{\#cycles(\mathrm{Curve25519})}{\#cycles(\mathrm{Four}\mathbb{Q})} \approx 2.5$$

$$\frac{\#cycles(\mathrm{NIST\ P\!-256})}{\#cycles(\mathrm{Four}\mathbb{Q})} \approx 5.5$$

Performance

Compared against other ECC alternatives:

$$\frac{\#cycles(\mathrm{Curve25519})}{\#cycles(\mathrm{Four}\mathbb{Q})} \approx 2.5$$

$$\frac{\#cycles(\mathrm{NIST\ P}\!-256)}{\#cycles(\mathrm{Four}\mathbb{Q})} \approx 5.5$$

Compared against other high-performance alternatives:

$$\frac{\#cycles(\mathrm{Kummer})}{\#cycles(\mathrm{Four}\mathbb{Q})} \approx 1.2$$

Performance

- Results in previous slide were obtained on x64 CPUs
- In this work, we want to answer the question…

how does Four$\mathbb{Q}$ perform on another platforms, e.g., on ARM?

ARM processors

- ARM dominates the mobile and wearable market
  - Smartphones, smartwatches, tablets, etc.
- 32-bit Cortex-A and Cortex-M architectures have been key technologies for this success

ARM processors

- ARM dominates the mobile and wearable market
  - Smartphones, smartwatches, tablets, etc.
- 32-bit Cortex-A and Cortex-M architectures have been key technologies for this success
- In this work, we targeted several widely used **Cortex-A microarchitectures**: Cortex-A7, Cortex-A8, Cortex-A9 and Cortex-A15
- 32-bit ARM has a RISC-based architecture equipped with *sixteen* 32-bit registers and an instruction set supporting 32-bit operations, or a mix of 16-bit and 32-bit operations in the case of Thumb and Thumb2

ARM processors

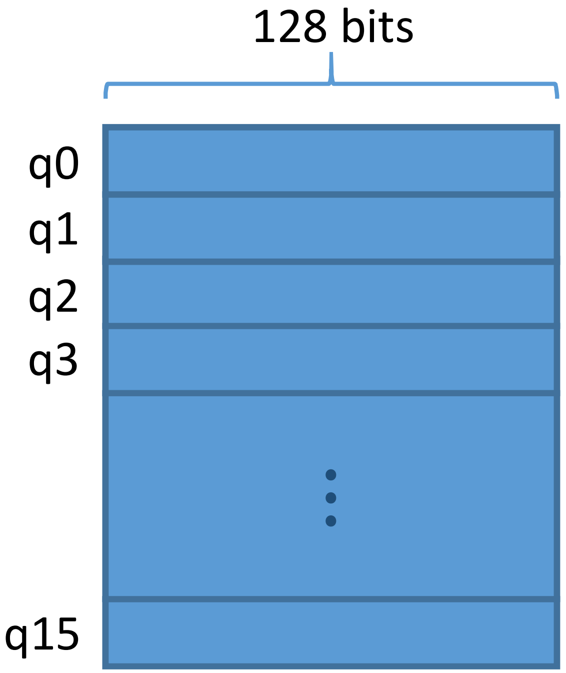
- ARM dominates the mobile and wearable market
  - Smartphones, smartwatches, tablets, etc.
- 32-bit Cortex-A and Cortex-M architectures have been key technologies for this success
- In this work, we targeted several widely used **Cortex-A microarchitectures**: Cortex-A7, Cortex-A8, Cortex-A9 and Cortex-A15
- 32-bit ARM has a RISC-based architecture equipped with *sixteen* 32-bit registers and an instruction set supporting 32-bit operations, or a mix of 16-bit and 32-bit operations in the case of Thumb and Thumb2
- Many ARM-based processors come equipped with NEON, a powerful 128-bit SIMD engine
- **In this talk, we exploit NEON to perform high-performance, constant-time Four$\mathbb{Q}$ scalar multiplications**

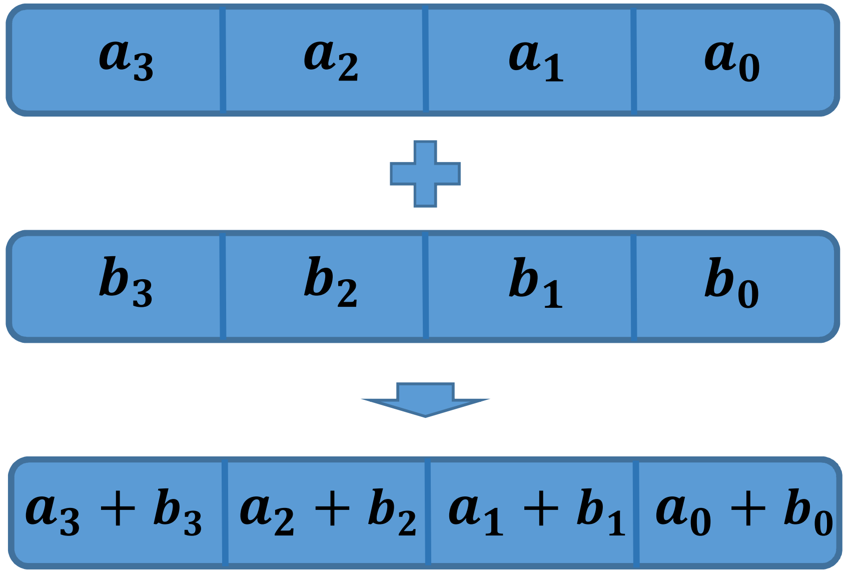
Vector ARM instructions: NEON

- NEON is supported by a wide range of ARM-based processors
- SIMD instructions can perform **128-bit wide operations** on 8-bit (**BYTE**), 16-bit (**WORD**), 32-bit (**DOUBLEWORD**) or 64-bit (**QUADWORD**) operands

Vector ARM instructions: NEON

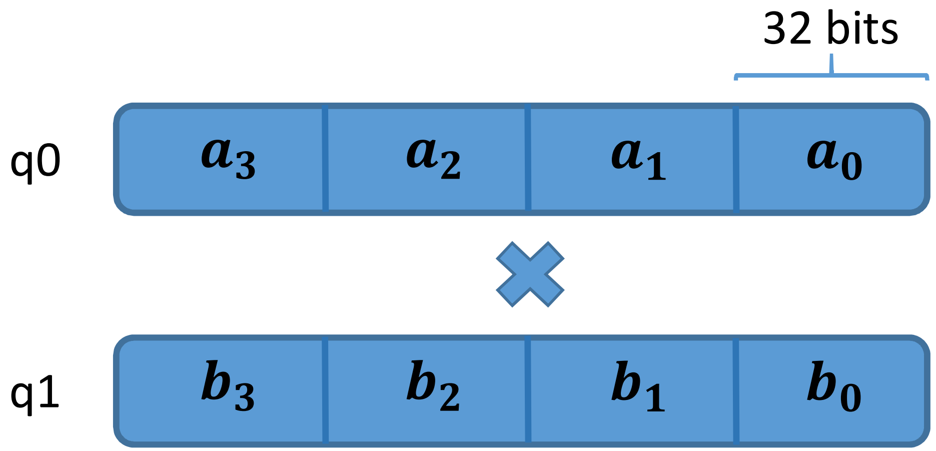
- NEON is supported by a wide range of ARM-based processors
- SIMD instructions can perform **128-bit wide operations** on 8-bit (**BYTE**), 16-bit (**WORD**), 32-bit (**DOUBLEWORD**) or 64-bit (**QUADWORD**) operands
- It comes with *sixteen* 128-bit registers (`q0–q15`)





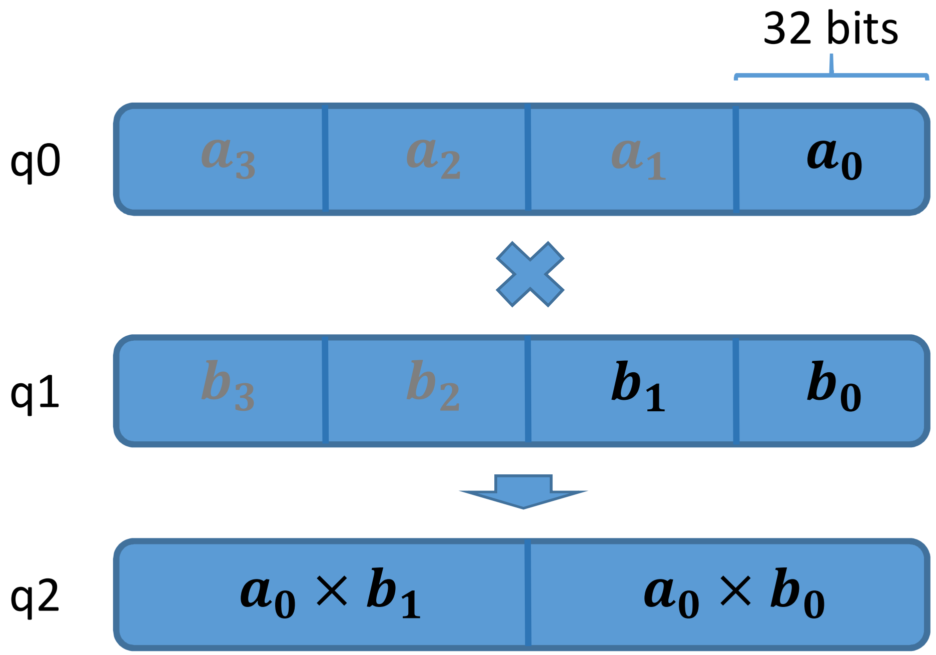
Vector ARM instructions: NEON

- **VMULL.S32:** signed 2-way 32 × 32-bit multiplies resulting in two 64-bit products



Vector ARM instructions: NEON

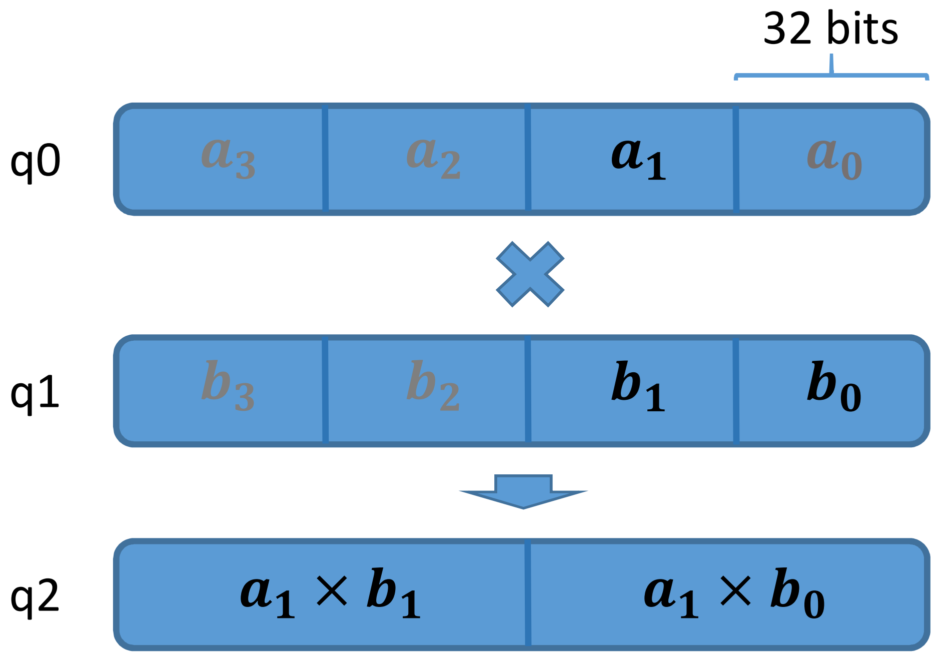
- **VMULL.S32:** signed 2-way 32 × 32-bit multiplies resulting in two 64-bit products



VMULL.S32 d4, d2, d0[0]

Vector ARM instructions: NEON

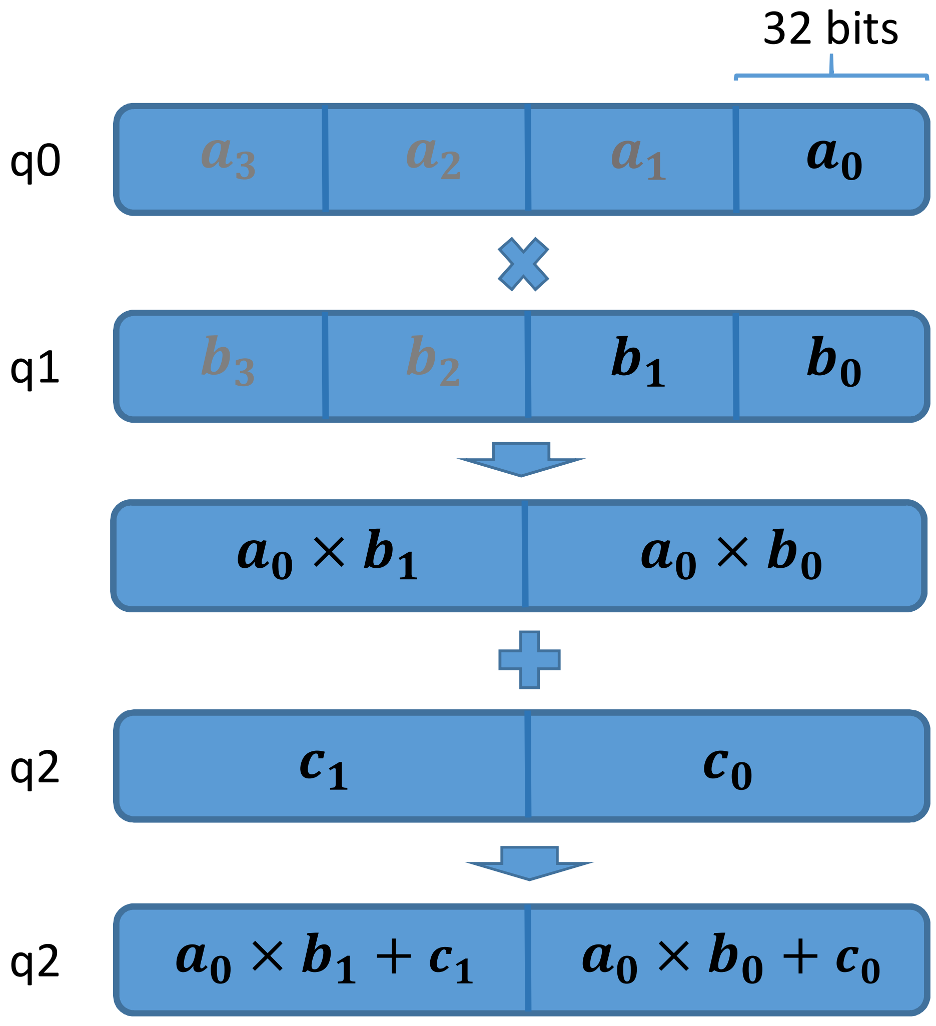
- **VMULL.S32:** signed 2-way 32 × 32-bit multiplies resulting in two 64-bit products



VMULL.S32 d4, d2, d0[1]

Vector ARM instructions: NEON

- **VMLAL.S32:** signed 2-way 32 × 32-bit multiplies resulting in two 64-bit products followed by 64-bit additions



VMLAL.S32 q2, d2, d0[0]

Vector ARM instructions: NEON

- When there are no pipeline stalls, most instructions take 1 cycle
- When there no pipeline stalls, `vmull.s32` and `vmlal.s32` take 2 cycles
  - Additions for accumulation are for free
- `vmull.s32` and `vmlal.s32` have latencies of 6 cycles

Vector ARM instructions: NEON

- When there are no pipeline stalls, most instructions take 1 cycle
- When there no pipeline stalls, `vmull.s32` and `vmlal.s32` take 2 cycles
  - Additions for accumulation are for free
- `vmull.s32` and `vmlal.s32` have latencies of 6 cycles

What we want to exploit:

- Special forwarding when a multiply or a multiply-and-add is followed by a multiply-and-add that depends on the result of the previous instruction: **instructions are executed back-to-back with maximal throughput of 2 cycles/instruction**

Vector ARM instructions: NEON

- When there are no pipeline stalls, most instructions take 1 cycle
- When there no pipeline stalls, `vmull.s32` and `vmlal.s32` take 2 cycles
  - Additions for accumulation are for free
- `vmull.s32` and `vmlal.s32` have latencies of 6 cycles

What we want to exploit:

- Special forwarding when a multiply or a multiply-and-add is followed by a multiply-and-add that depends on the result of the previous instruction: **instructions are executed back-to-back with maximal throughput of 2 cycles/instruction**

What we want to minimize:

- Shuffling data between vector registers introduces some overhead

FourQ

$$E/\mathbb{F}_{p^2}\colon -x^2 + y^2 = 1 + dx^2y^2$$

$d = 125317048443780598345676279555970305165i + 4205857648805777768770,$

$p = 2^{127} - 1$, $i^2 = -1$, $\#E = 392 \cdot N$, where N is a 246-bit prime.

Fourℚ

$$\boxed{E/\mathbb{F}_{p^2}: -x^2 + y^2 = 1 + dx^2y^2}$$

$$d = 125317048443780598345676279555970305165i + 4205857648805777768770,$$

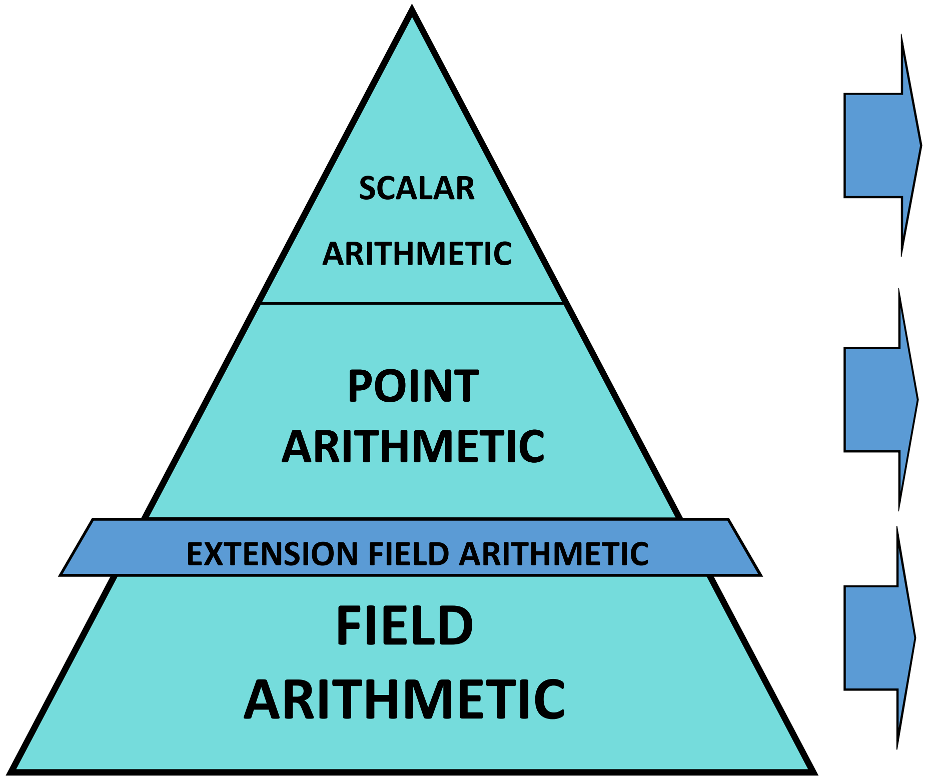
$p = 2^{127} - 1$, $i^2 = -1$, $\#E = 392 \cdot N$, where N is a 246-bit prime.

- E is equipped with *two* endomorphisms, ψ and ϕ
- $\psi(P) = [\lambda_{\psi}]P$ and $\phi(P) = [\lambda_{\phi}]P$ for all $P \in E[N]$ and $m \in [0, 2^{256})$

$$m \mapsto (a_1, a_2, a_3, a_4)$$

$$\boxed{[m]P = [a_1]P + [a_2]\phi(P) + [a_3]\psi(P) + [a_4]\psi(\phi(P))}$$

Four$\mathbb{Q}$'s arithmetic layers



Scalar multiplication operations via 4-way scalar decompositions

Doubling and addition of points: $2P$, $P+Q$

Field addition, subtraction, multiplication, squaring, inversion

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$

- Recall that FourQ works over $\mathbb{F}_{p^2} = \mathbb{F}_p[i]$ with $i^2 = -1$
- Let $a = (a_0 + a_1 i),\ b = (b_0 + b_1 i) \in \mathbb{F}_{p^2}$. Elements $a_0, a_1, b_0, b_1 \in \mathbb{F}_p$

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$

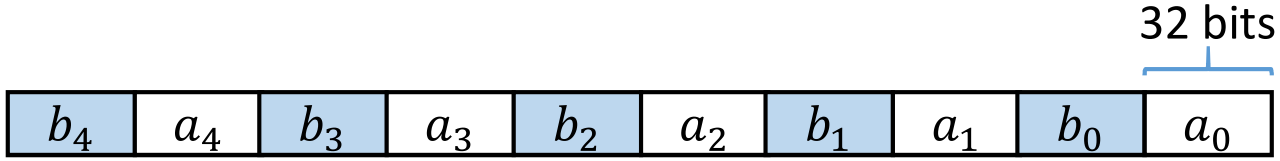
- Recall that Four$\mathbb{Q}$ works over $\mathbb{F}_{p^2} = \mathbb{F}_p[i]$ with $i^2 = -1$
- Let $a = (a_0 + a_1 i),\ b = (b_0 + b_1 i) \in \mathbb{F}_{p^2}$. Elements $a_0, a_1, b_0, b_1 \in \mathbb{F}_p$

$$
\begin{aligned}
&a + b = (a_0 + b_0, a_1 + b_1)\\
&a - b = (a_0 - b_0, a_1 - b_1)\\
&a \times b = (a_0 \cdot b_0 - a_1 \cdot b_1, a_0 \cdot b_1 + a_1 \cdot b_0)\\
&a^2 = \big((a_0 + a_1) \cdot (a_0 - a_1), 2a_0 \cdot a_1\big)\\
&a^{-1} = (a_0 \cdot (a_0^2 + a_1^2)^{-1}, -a_1 \cdot (a_0^2 + a_1^2)^{-1})
\end{aligned}
$$

- Computations only involve simple operations in $\mathbb{F}_{2^{127}-1}$

Four$\mathbb{Q}$ meets NEON: Four$\mathbb{Q}$NEON

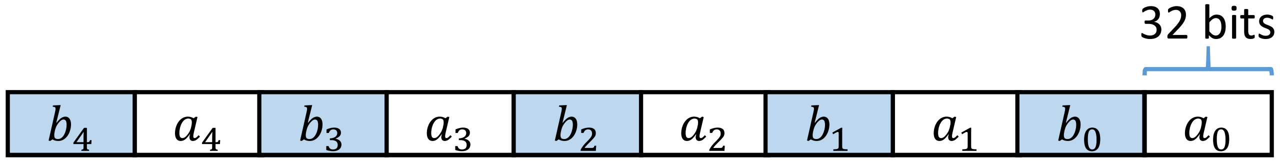
- An element $c = a + b \cdot i \in \mathbb{F}_{p^2}$ is represented as an *interleaved ten-coefficient vector*



where $a = a_0 + a_1 2^{26} + a_2 2^{52} + a_3 2^{78} + a_4 2^{104}$ and $b = b_0 + b_1 2^{26} + b_2 2^{52} + b_3 2^{78} + b_4 2^{104}$.

Four$\mathbb{Q}$ meets NEON: Four$\mathbb{Q}$NEON

- An element $c = a + b \cdot i \in \mathbb{F}_{p^2}$ is represented as an *interleaved ten-coefficient vector*



32 bits

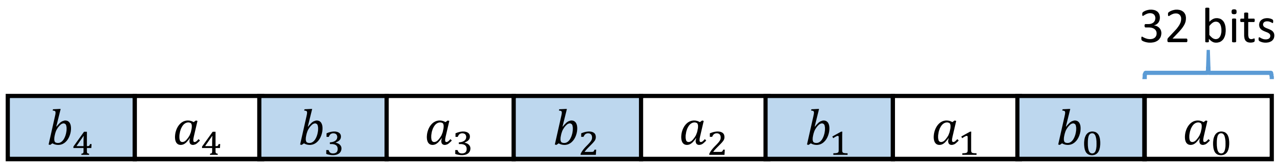
b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0

where $a = a_0 + a_1 2^{26} + a_2 2^{52} + a_3 2^{78} + a_4 2^{104}$ and $b = b_0 + b_1 2^{26} + b_2 2^{52} + b_3 2^{78} + b_4 2^{104}$.

- When fully reduced, $a_0, \ldots, a_3, b_0, \ldots, b_3$ have 26 bits and a_4, b_4 have 23 bits
- Coefficients are signed, i.e., $a_0, \ldots, a_3, b_0, \ldots, b_3 \in [-2^{26}, 2^{26}]$ and $a_4, b_4 \in [-2^{23}, 2^{23}]$ when fully reduced

Four$\mathbb{Q}$ meets NEON: Four$\mathbb{Q}$NEON

- An element $c = a + b \cdot i \in \mathbb{F}_{p^2}$ is represented as an *interleaved ten-coefficient vector*



32 bits

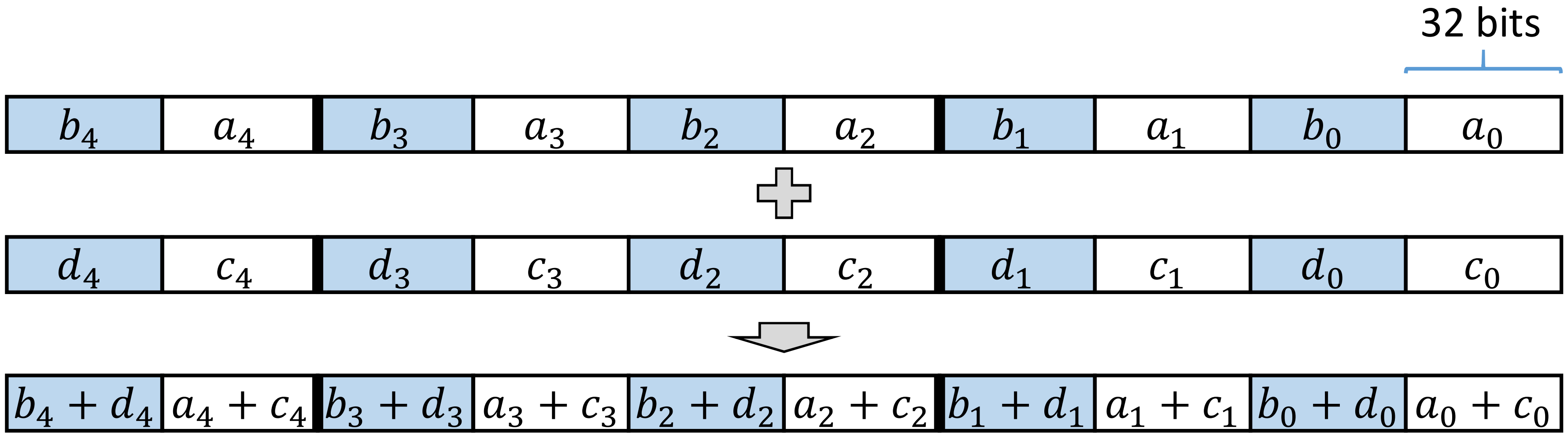
b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0

where $a = a_0 + a_1 2^{26} + a_2 2^{52} + a_3 2^{78} + a_4 2^{104}$ and $b = b_0 + b_1 2^{26} + b_2 2^{52} + b_3 2^{78} + b_4 2^{104}$.

- When fully reduced, $a_0, \ldots, a_3, b_0, \ldots, b_3$ have 26 bits and a_4, b_4 have 23 bits
- Coefficients are signed, i.e., $a_0, \ldots, a_3, b_0, \ldots, b_3 \in [-2^{26}, 2^{26}]$ and $a_4, b_4 \in [-2^{23}, 2^{23}]$ when fully reduced
- Functions to convert back and forth between vector and canonical representations are straightforward and are only required once at the beginning and once at the end of scalar multiplication

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$

- Addition and subtraction in $\mathbb{F}_{p^2}$ are simply coefficient-wise operations



32 bits

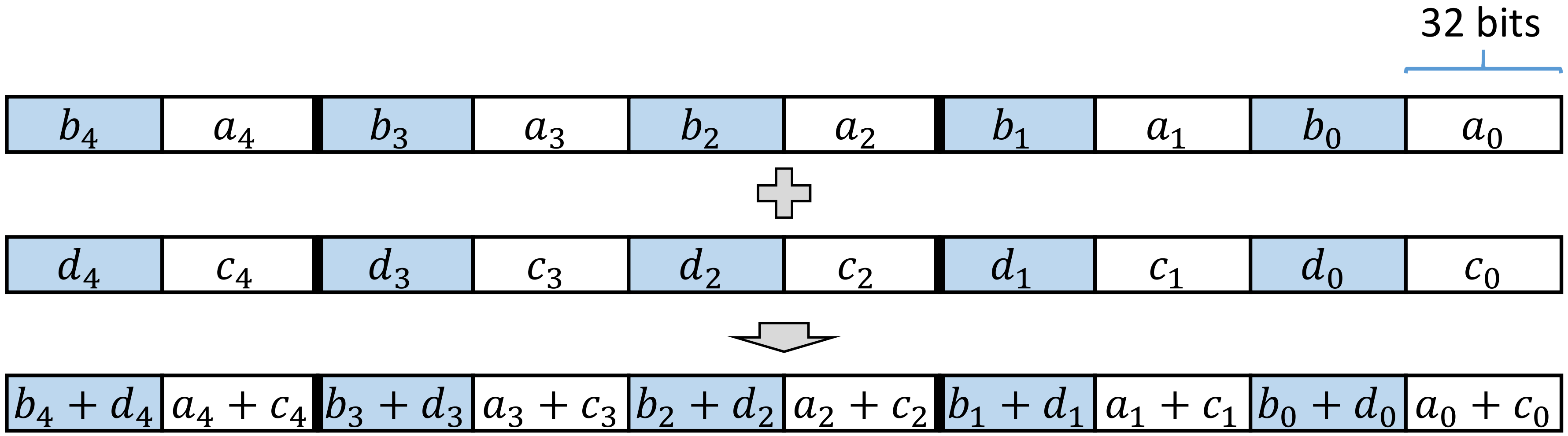
b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0

d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0

$b_4 + d_4$	$a_4 + c_4$	$b_3 + d_3$	$a_3 + c_3$	$b_2 + d_2$	$a_2 + c_2$	$b_1 + d_1$	$a_1 + c_1$	$b_0 + d_0$	$a_0 + c_0$

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$

- Addition and subtraction in $\mathbb{F}_{p^2}$ are simply coefficient-wise operations



32 bits

b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$b_4 + d_4$	$a_4 + c_4$	$b_3 + d_3$	$a_3 + c_3$	$b_2 + d_2$	$a_2 + c_2$	$b_1 + d_1$	$a_1 + c_1$	$b_0 + d_0$	$a_0 + c_0$

- Requires two 128-bit NEON additions (resp. subtractions) and one 64-bit NEON addition (resp. subtraction) using `vadd.32` (resp. `vsub.s32`)

- We can perform many additions and subtractions without overflowing the 32-bit coefficient storage capacity

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Multiplication and squaring in $\mathbb{F}_{p^2}$ use as basis a schoolbook-like multiplication
- Define field elements $a = a_0 + a_1 2^{26} + a_2 2^{52} + a_3 2^{78} + a_4 2^{104}$ and $b = b_0 + b_1 2^{26} + b_2 2^{52} + b_3 2^{78} + b_4 2^{104}$

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Multiplication and squaring in $\mathbb{F}_{p^2}$ use as basis a schoolbook-like multiplication
- Define field elements $a = a_0 + a_1 2^{26} + a_2 2^{52} + a_3 2^{78} + a_4 2^{104}$ and $b = b_0 + b_1 2^{26} + b_2 2^{52} + b_3 2^{78} + b_4 2^{104}$
- We compute $c = a \times b \bmod p$ as

$$
\begin{aligned}
c_0 &= a_0 b_0 + 8(a_1 b_4 + a_4 b_1 + a_2 b_3 + a_3 b_2)\\
c_1 &= a_0 b_1 + a_1 b_0 + 8(a_2 b_4 + a_4 b_2 + a_3 b_3)\\
c_2 &= a_0 b_2 + a_2 b_0 + a_1 b_1 + 8(a_3 b_4 + a_4 b_3)\\
c_3 &= a_0 b_3 + a_3 b_0 + a_1 b_2 + a_2 b_1 + 8(a_4 b_4)\\
c_4 &= a_0 b_4 + a_4 b_0 + a_1 b_3 + a_3 b_1 + a_2 b_2
\end{aligned}
$$

where $c = c_0 + c_1 2^{26} + c_2 2^{52} + c_3 2^{78} + c_4 2^{104}$.

(Note that $2^{130} \equiv 8$)

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- We describe the case of multiplication (the most time-critical operation)
- Let $A = (a + bi),\ B = (c + di) \in \mathbb{F}_{p^2}$

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- We describe the case of multiplication (the most time-critical operation)
- Let $A = (a + bi),\ B = (c + di) \in \mathbb{F}_{p^2}$
- Let $A = (b_4, a_4, b_3, a_3, b_2, a_2, b_1, a_1, b_0, a_0)$ and $B = (d_4, c_4, d_3, c_3, d_2, c_2, d_1, c_1, d_0, c_0)$ using the interleaved representation
- We compute $A \times B$ as $(a \times c - b \times d) + (a \times d + b \times c)i$

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

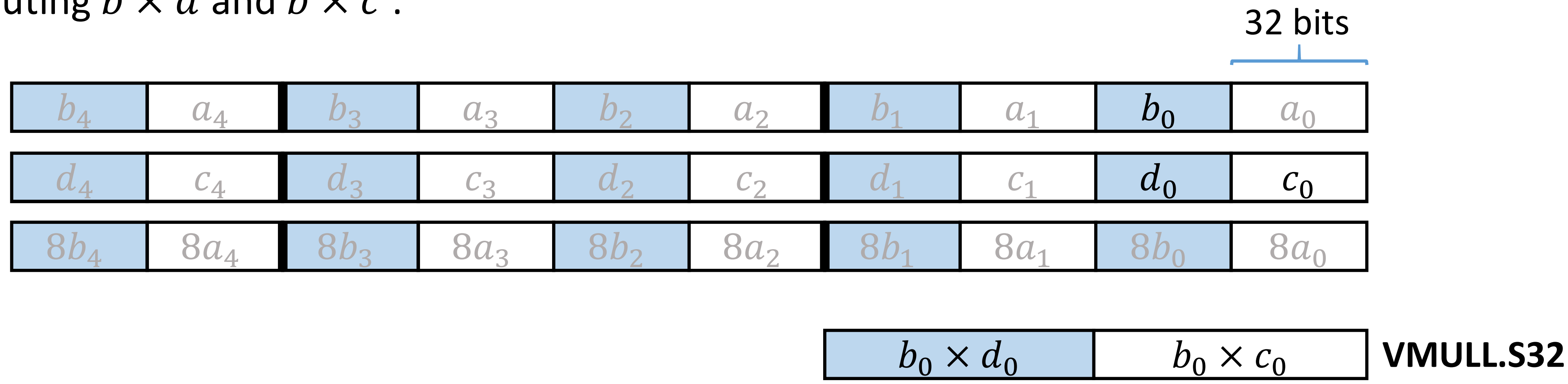
- Computing $b \times d$ and $b \times c$:

32 bits

b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$8b_4$	$8a_4$	$8b_3$	$8a_3$	$8b_2$	$8a_2$	$8b_1$	$8a_1$	$8b_0$	$8a_0$

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $b \times d$ and $b \times c$:



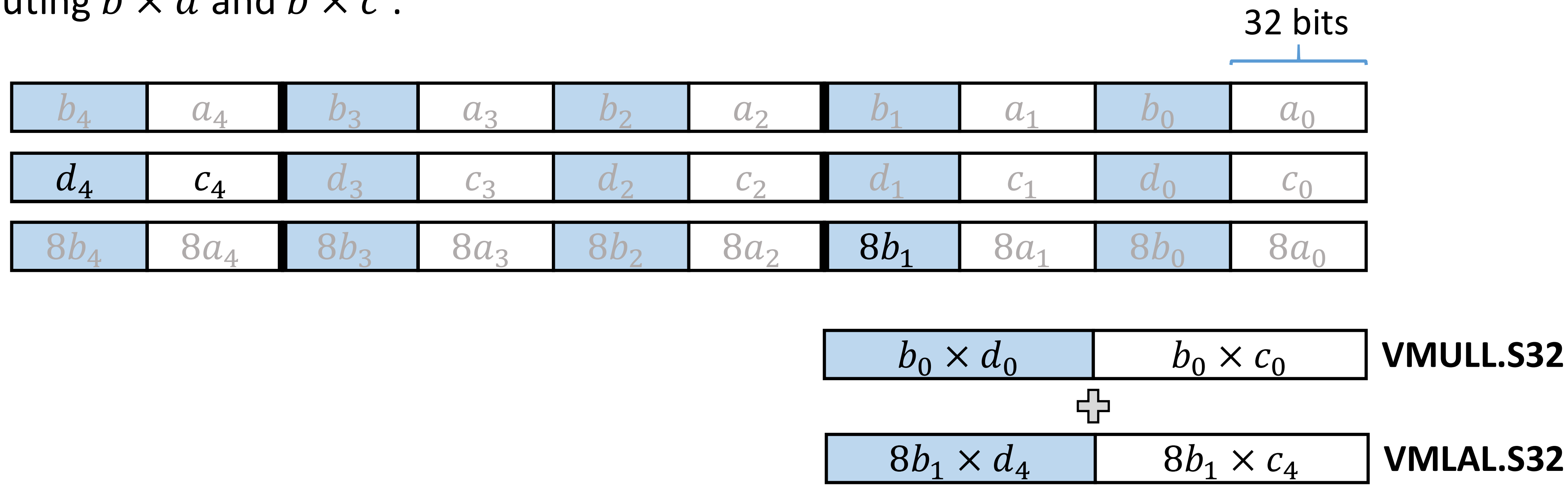
32 bits

b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$8b_4$	$8a_4$	$8b_3$	$8a_3$	$8b_2$	$8a_2$	$8b_1$	$8a_1$	$8b_0$	$8a_0$

$b_0 \times d_0$	$b_0 \times c_0$	**VMULL.S32**

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $b \times d$ and $b \times c$:



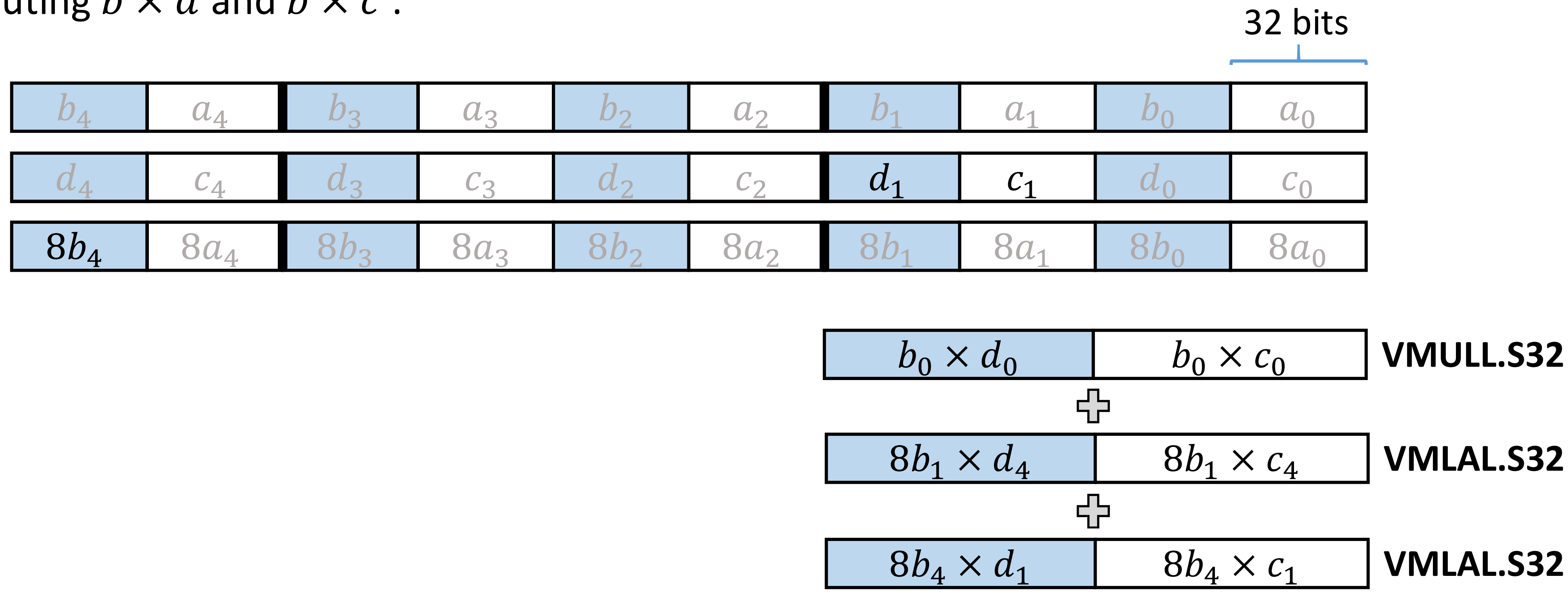
32 bits

b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$8b_4$	$8a_4$	$8b_3$	$8a_3$	$8b_2$	$8a_2$	$8b_1$	$8a_1$	$8b_0$	$8a_0$

$b_0 \times d_0$	$b_0 \times c_0$	**VMULL.S32**
$8b_1 \times d_4$	$8b_1 \times c_4$	**VMLAL.S32**

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $b \times d$ and $b \times c$:



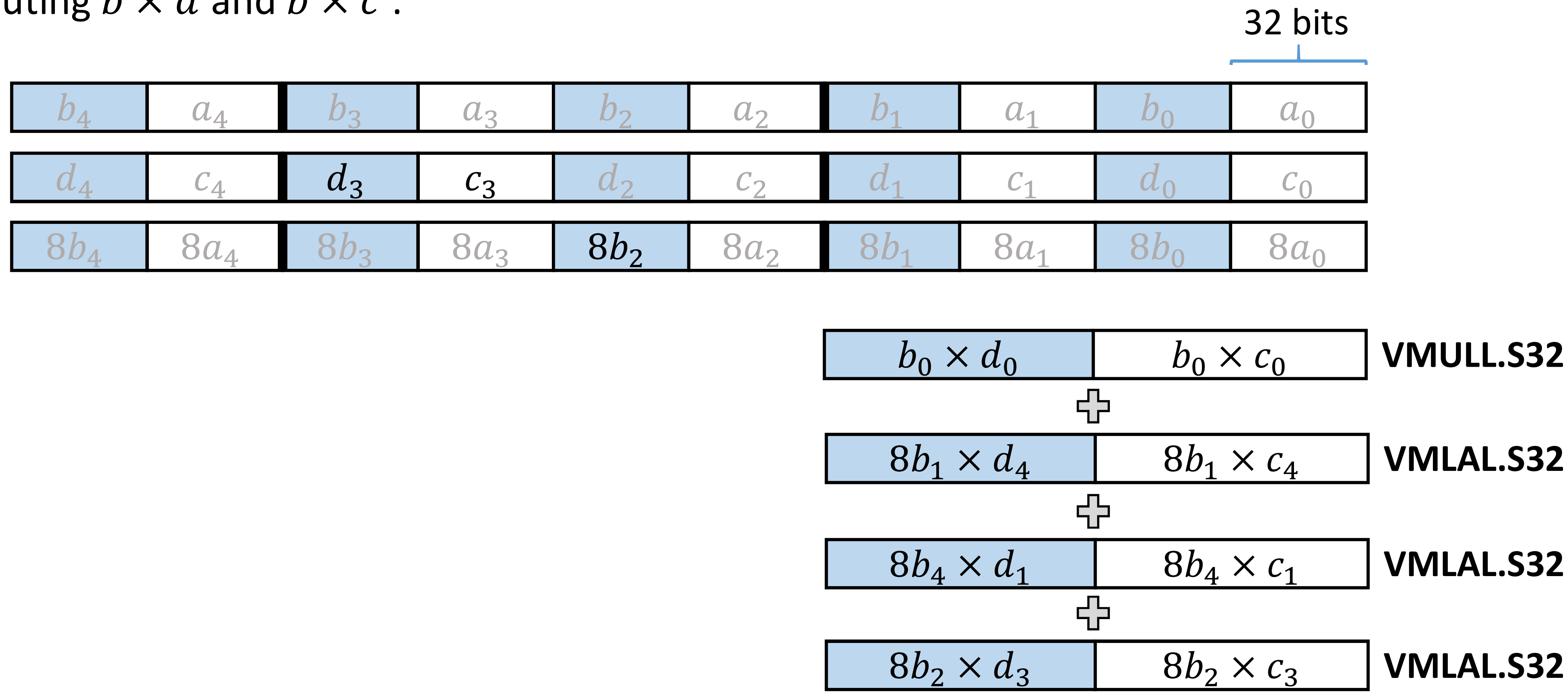
32 bits

b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$8b_4$	$8a_4$	$8b_3$	$8a_3$	$8b_2$	$8a_2$	$8b_1$	$8a_1$	$8b_0$	$8a_0$

$b_0 \times d_0$	$b_0 \times c_0$	**VMULL.S32**
+		
$8b_1 \times d_4$	$8b_1 \times c_4$	**VMLAL.S32**
+		
$8b_4 \times d_1$	$8b_4 \times c_1$	**VMLAL.S32**

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $b \times d$ and $b \times c$:



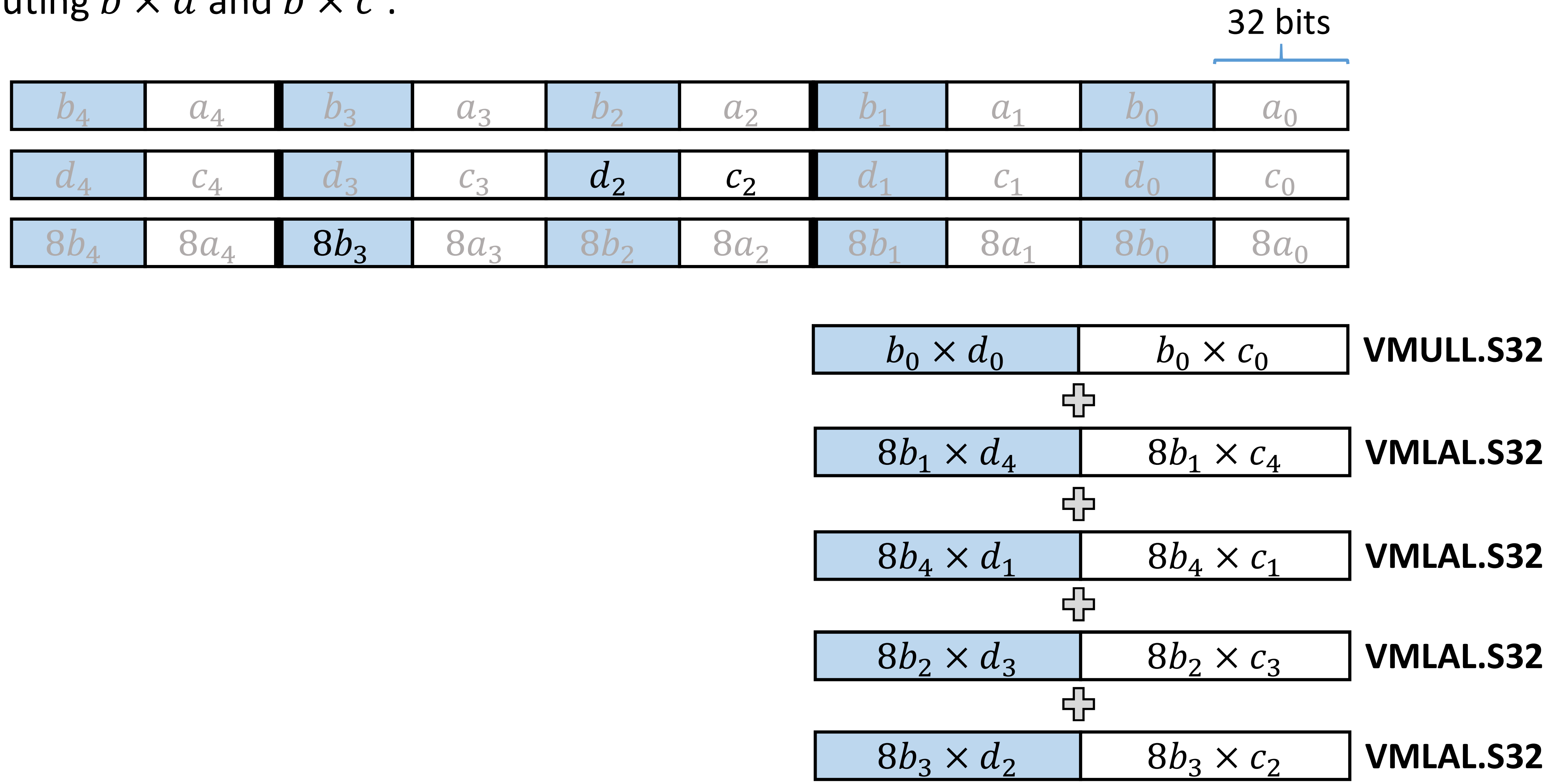
32 bits

b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$8b_4$	$8a_4$	$8b_3$	$8a_3$	$8b_2$	$8a_2$	$8b_1$	$8a_1$	$8b_0$	$8a_0$

$b_0 \times d_0$	$b_0 \times c_0$	**VMULL.S32**
$8b_1 \times d_4$	$8b_1 \times c_4$	**VMLAL.S32**
$8b_4 \times d_1$	$8b_4 \times c_1$	**VMLAL.S32**
$8b_2 \times d_3$	$8b_2 \times c_3$	**VMLAL.S32**

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $b \times d$ and $b \times c$:



32 bits

b_4 a_4 b_3 a_3 b_2 a_2 b_1 a_1 b_0 a_0

d_4 c_4 d_3 c_3 d_2 c_2 d_1 c_1 d_0 c_0

$8b_4$ $8a_4$ $8b_3$ $8a_3$ $8b_2$ $8a_2$ $8b_1$ $8a_1$ $8b_0$ $8a_0$

$b_0 \times d_0$ $b_0 \times c_0$ **VMULL.S32**

+

$8b_1 \times d_4$ $8b_1 \times c_4$ **VMLAL.S32**

+

$8b_4 \times d_1$ $8b_4 \times c_1$ **VMLAL.S32**

+

$8b_2 \times d_3$ $8b_2 \times c_3$ **VMLAL.S32**

+

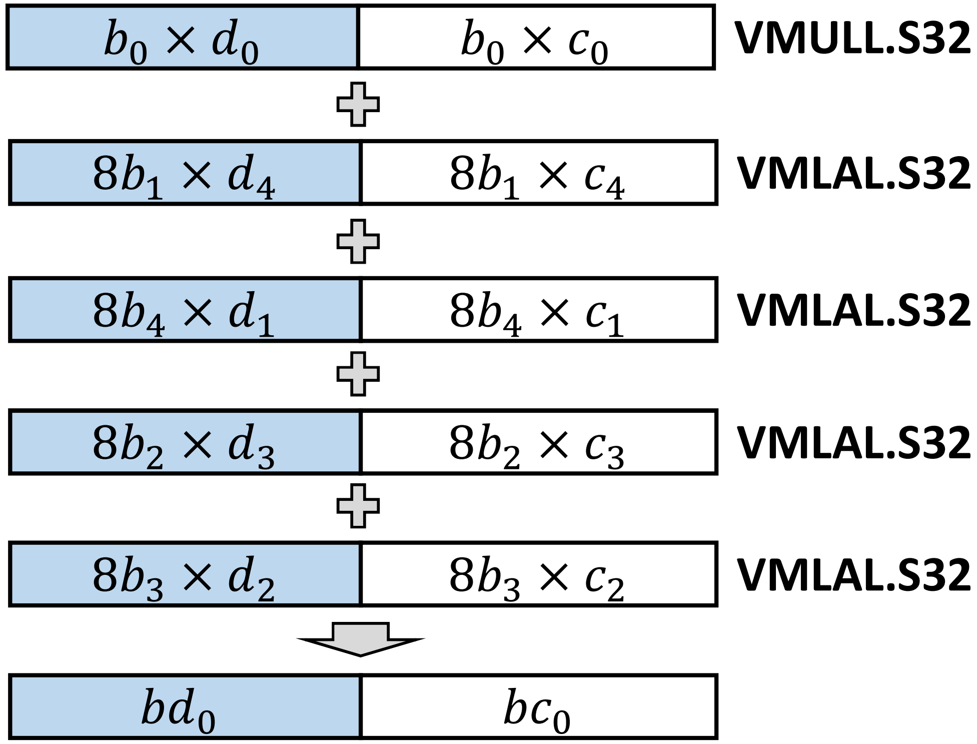
$8b_3 \times d_2$ $8b_3 \times c_2$ **VMLAL.S32**

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $b \times d$ and $b \times c$:

32 bits

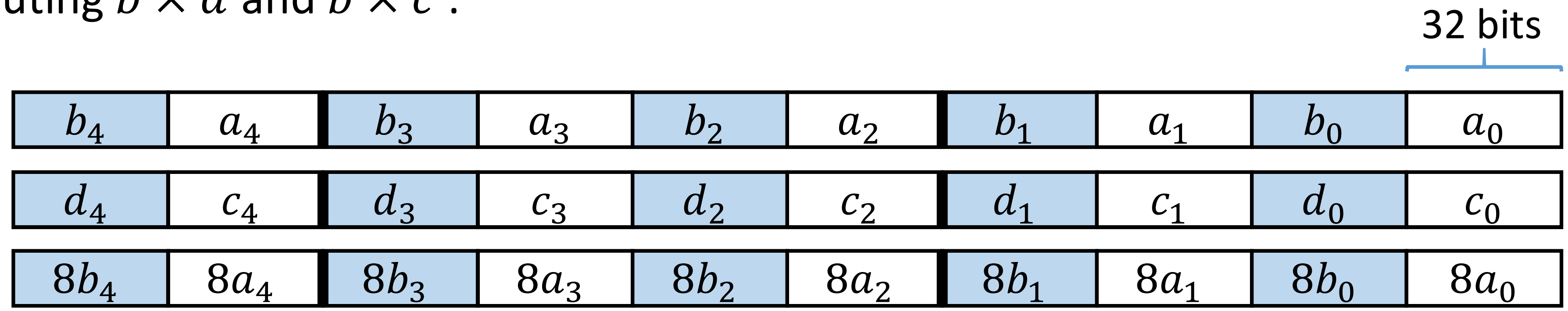
b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$8b_4$	$8a_4$	$8b_3$	$8a_3$	$8b_2$	$8a_2$	$8b_1$	$8a_1$	$8b_0$	$8a_0$



$b_0 \times d_0$	$b_0 \times c_0$	**VMULL.S32**
$8b_1 \times d_4$	$8b_1 \times c_4$	**VMLAL.S32**
$8b_4 \times d_1$	$8b_4 \times c_1$	**VMLAL.S32**
$8b_2 \times d_3$	$8b_2 \times c_3$	**VMLAL.S32**
$8b_3 \times d_2$	$8b_3 \times c_2$	**VMLAL.S32**
bd_0	bc_0	

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $b \times d$ and $b \times c$:

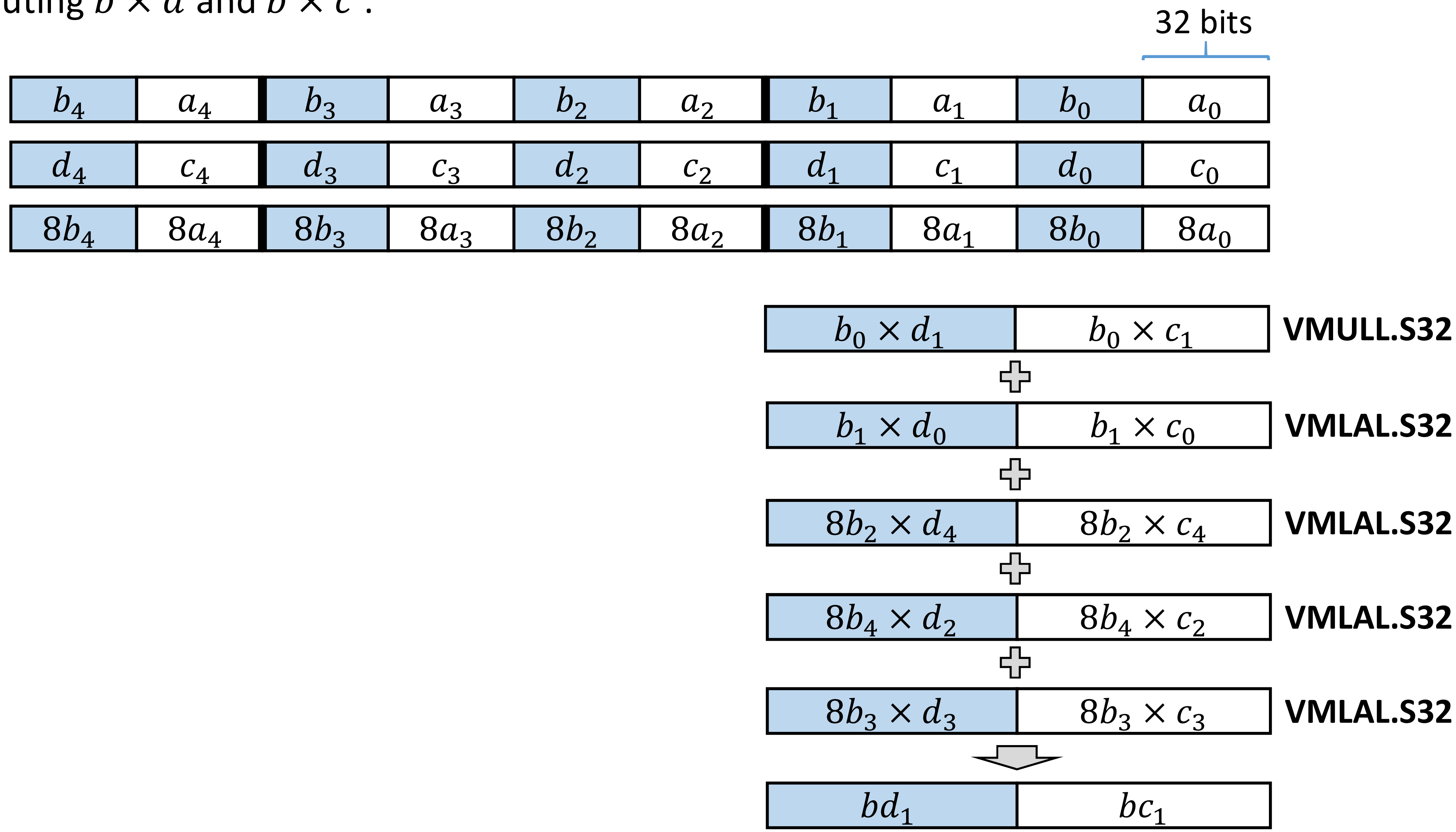


32 bits

b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$8b_4$	$8a_4$	$8b_3$	$8a_3$	$8b_2$	$8a_2$	$8b_1$	$8a_1$	$8b_0$	$8a_0$

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $b \times d$ and $b \times c$:



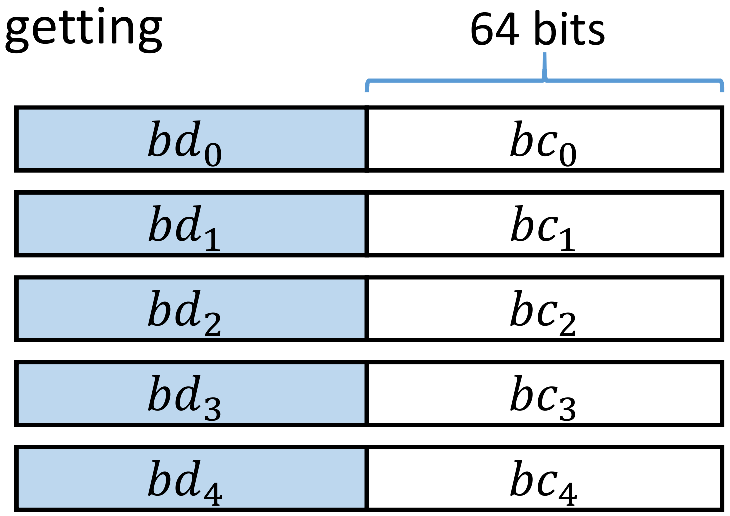
32 bits

b_4	a_4	b_3	a_3	b_2	a_2	b_1	a_1	b_0	a_0
d_4	c_4	d_3	c_3	d_2	c_2	d_1	c_1	d_0	c_0
$8b_4$	$8a_4$	$8b_3$	$8a_3$	$8b_2$	$8a_2$	$8b_1$	$8a_1$	$8b_0$	$8a_0$

$b_0 \times d_1$	$b_0 \times c_1$	**VMULL.S32**
$b_1 \times d_0$	$b_1 \times c_0$	**VMLAL.S32**
$8b_2 \times d_4$	$8b_2 \times c_4$	**VMLAL.S32**
$8b_4 \times d_2$	$8b_4 \times c_2$	**VMLAL.S32**
$8b_3 \times d_3$	$8b_3 \times c_3$	**VMLAL.S32**
bd_1	bc_1	

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

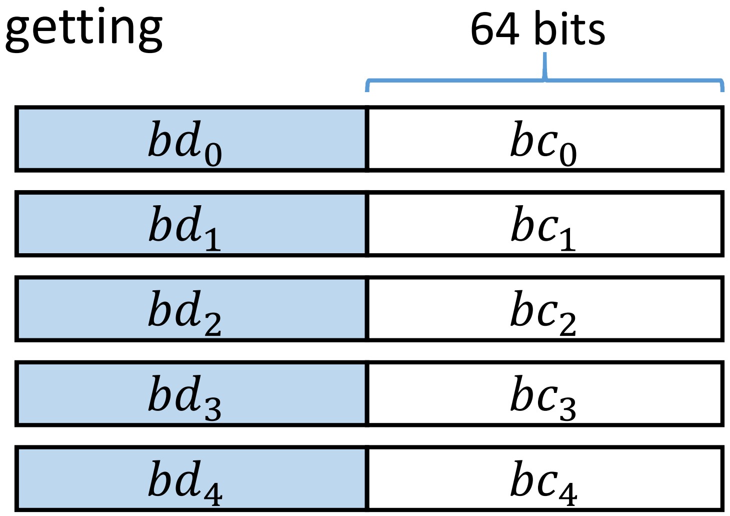
- Repeat until getting



	64 bits
bd_0	bc_0
bd_1	bc_1
bd_2	bc_2
bd_3	bc_3
bd_4	bc_4

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

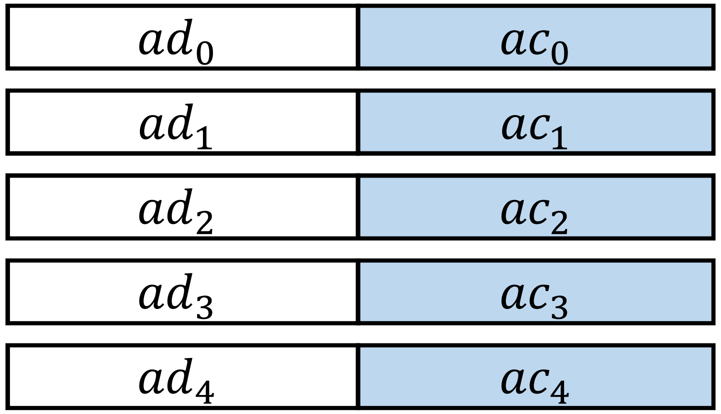
- Repeat until getting



64 bits

bd_0	bc_0
bd_1	bc_1
bd_2	bc_2
bd_3	bc_3
bd_4	bc_4

- Similar work done for computing $a \times c$ and $a \times d$



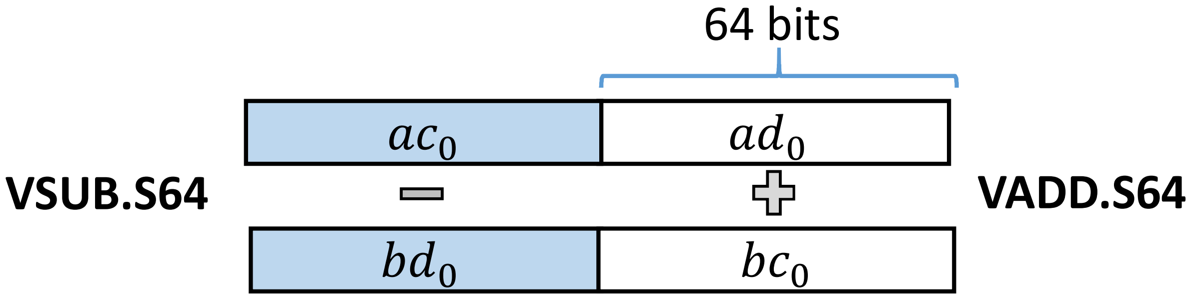
ad_0	ac_0
ad_1	ac_1
ad_2	ac_2
ad_3	ac_3
ad_4	ac_4

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $a \times c - b \times d$ and $a \times d + b \times c$:

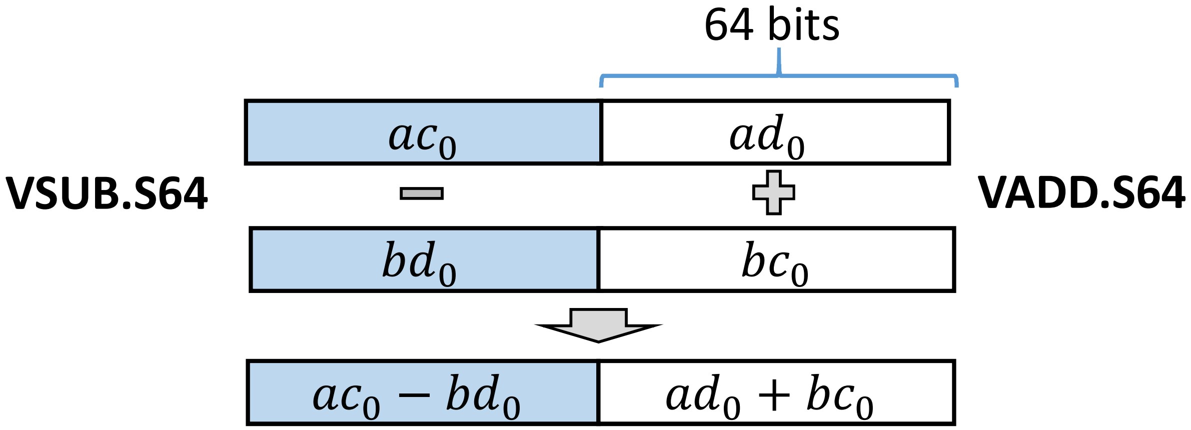
Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $a \times c - b \times d$ and $a \times d + b \times c$:



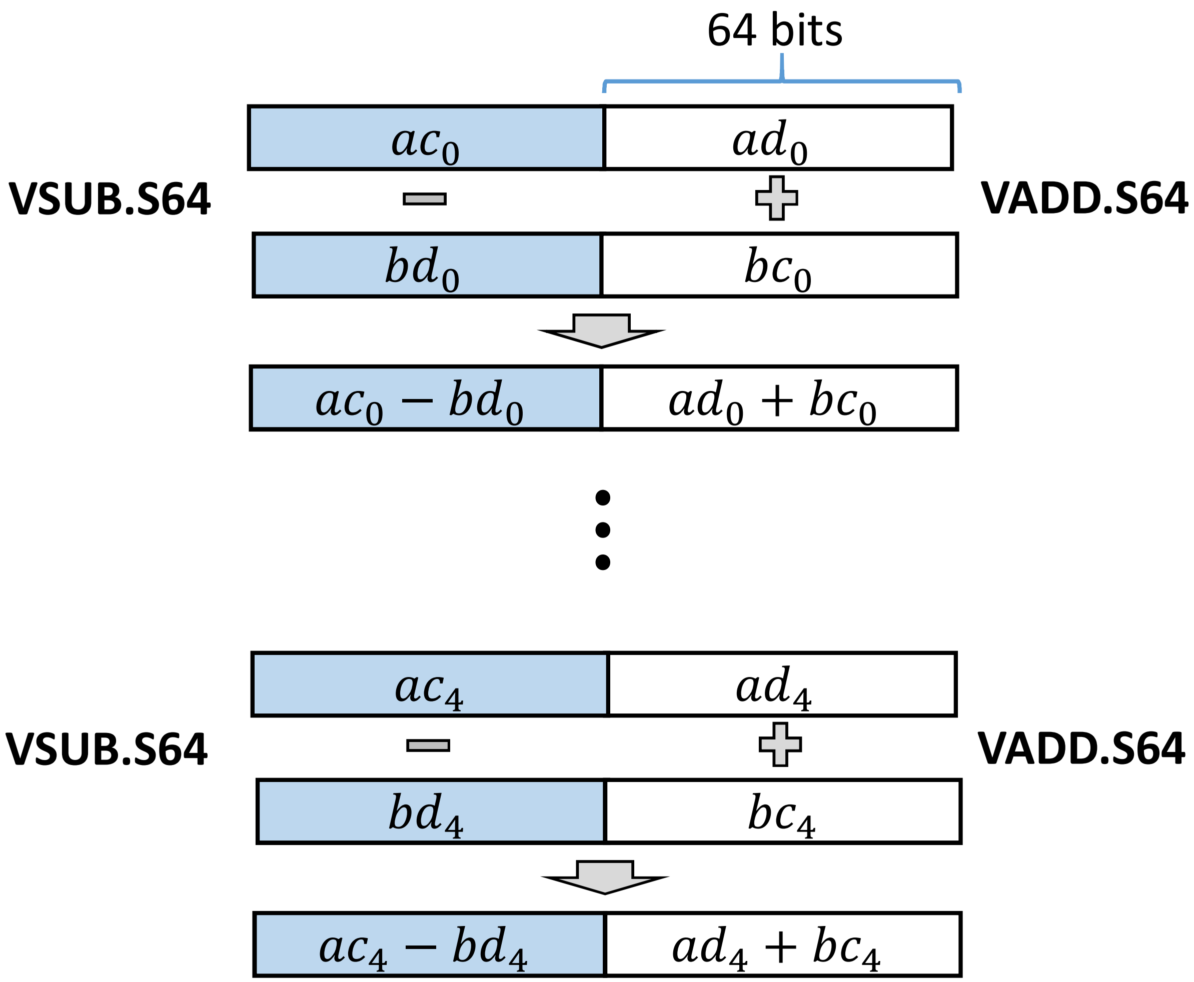
Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $a \times c - b \times d$ and $a \times d + b \times c$:



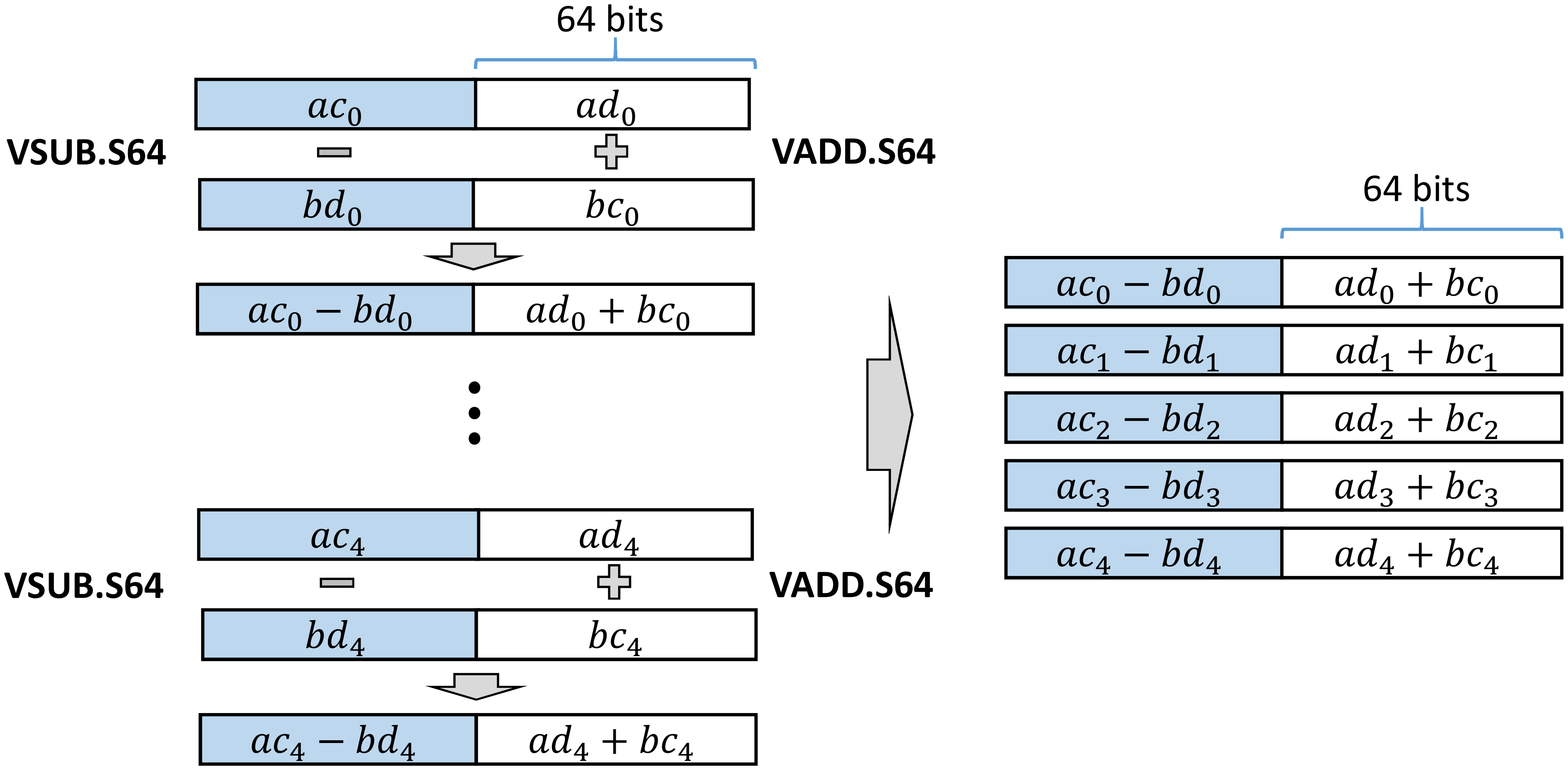
Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $a \times c - b \times d$ and $a \times d + b \times c$:



Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $a \times c - b \times d$ and $a \times d + b \times c$:



64 bits

ac_0	ad_0

VSUB.S64 − + VADD.S64

bd_0	bc_0

$ac_0 - bd_0$	$ad_0 + bc_0$

⋮

ac_4	ad_4

VSUB.S64 − + VADD.S64

bd_4	bc_4

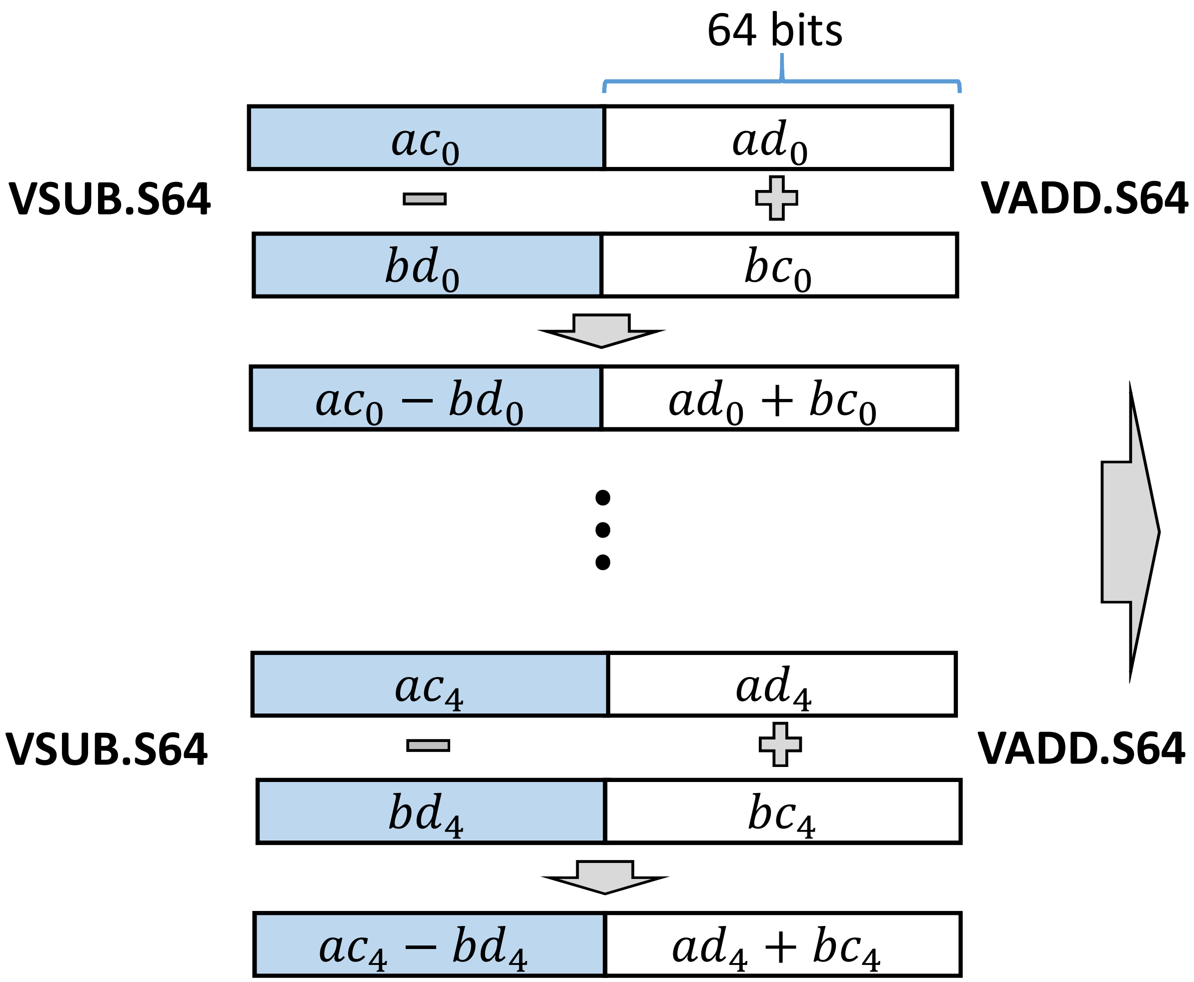
$ac_4 - bd_4$	$ad_4 + bc_4$

64 bits

$ac_0 - bd_0$	$ad_0 + bc_0$
$ac_1 - bd_1$	$ad_1 + bc_1$
$ac_2 - bd_2$	$ad_2 + bc_2$
$ac_3 - bd_3$	$ad_3 + bc_3$
$ac_4 - bd_4$	$ad_4 + bc_4$

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$: multiplication

- Computing $a \times c - b \times d$ and $a \times d + b \times c$:



64 bits

VSUB.S64

VADD.S64

ac_0 | ad_0

$-$ | $+$

bd_0 | bc_0

$ac_0 - bd_0$ | $ad_0 + bc_0$

⋮

VSUB.S64

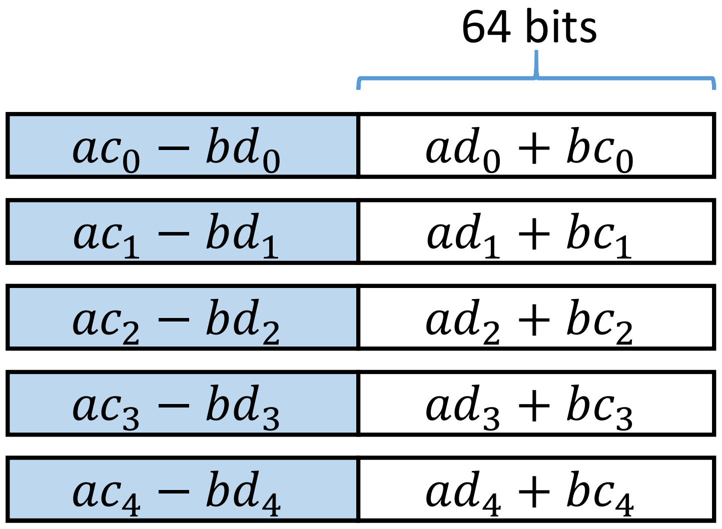
VADD.S64

ac_4 | ad_4

$-$ | $+$

bd_4 | bc_4

$ac_4 - bd_4$ | $ad_4 + bc_4$



64 bits

$ac_0 - bd_0$ | $ad_0 + bc_0$

$ac_1 - bd_1$ | $ad_1 + bc_1$

$ac_2 - bd_2$ | $ad_2 + bc_2$

$ac_3 - bd_3$ | $ad_3 + bc_3$

$ac_4 - bd_4$ | $ad_4 + bc_4$

- A final *carry correction* is required to reduce terms from 64 to 32 bits

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$

- In our implementation, we do not exactly execute operations in the order previously described
    - We reschedule operations to make them fit in the 16 available NEON registers

Arithmetic in $\mathbb{F}_{(2^{127}-1)^2}$

- In our implementation, we do not exactly execute operations in the order previously described
  - We reschedule operations to make them fit in the 16 available NEON registers

In the paper, we describe additional techniques to improve performance.

For example:

- Mixing ARM and NEON instructions in the $\mathbb{F}_{p^2}$ arithmetic in Cortex-A8 and A9.
- Interleaving memory and non-memory instructions in Cortex-A7, A8 and A9

Comparison with other 128-bit security curves

Cycles to compute variable-base scalar multiplication (in 10^4 cycles)

Curve	Field	Cortex-A7	Cortex-A8	Cortex-A9	Cortex-A15
Four$\mathbb{Q}$ (this work)	$\mathbb{F}_{p^2},\ p = 2^{127} - 1$	**378**	**242**	**257**	**133**
Kummer (Gaudry-Schost)	$\mathbb{F}_{p},\ p = 2^{127} - 1$	580 *	305	356	224 *
Curve25519 (Bernstein)	$\mathbb{F}_{p},\ p = 2^{255} - 19$	926 *	497	568	315
NIST K-283	binary, $\mathbb{F}_{2^{283}}$	-	934	1,148	736

Kummer: implementations by Bernstein et al [BCL+14]. Results from [eBACS].

Curve25519: implementations by Bernstein and Schwabe [BS12]. Results from [eBACS].

NIST K-283: implementation and results from Câmara et al. [CGL+13].

** Results obtained by running SUPERCOP on the targeted machine.*

Comparison with other 128-bit security curves

Cycles to compute variable-base scalar multiplication (in 10^4 cycles)

Curve	Field	Cortex-A7	Cortex-A8	Cortex-A9	Cortex-A15
Four$\mathbb{Q}$ (this work)	$\mathbb{F}_{p^2},\ p = 2^{127} - 1$	**378**	**242**	**257**	**133**
Kummer (Gaudry-Schost)	$\mathbb{F}_{p},\ p = 2^{127} - 1$	580 *	305	356	224 *
Curve25519 (Bernstein)	$\mathbb{F}_{p},\ p = 2^{255} - 19$	926 *	497	568	315
NIST K-283	binary, $\mathbb{F}_{2^{283}}$	-	934	1,148	736

Kummer: implementations by Bernstein et al [BCL+14]. Results from [eBACS].

Curve25519: implementations by Bernstein and Schwabe [BS12]. Results from [eBACS].

NIST K-283: implementation and results from Câmara et al. [CGL+13].

** Results obtained by running SUPERCOP on the targeted machine.*

Comparison with other 128-bit security curves

In summary, for variable-base scalar multiplication, Four$\mathbb{Q}$ is:

- Between **2.1—2.4 times faster** than Curve25519
- Between **1.3—1.7 times faster** than genus 2 Kummer

Comparison with other 128-bit security curves

- But… **results are even better in some practical scenarios!**

Comparison with other 128-bit security curves

- But… **results are even better in some practical scenarios!**
- For example, genus 2 Kummer does not support efficient fixed-base scalar multiplications

Comparison with other 128-bit security curves

- But… **results are even better in some practical scenarios!**
- For example, genus 2 Kummer does not support efficient fixed-base scalar multiplications
- Therefore, scenarios such as **ephemeral Diffie-Hellman key exchange** or **digital signatures** can be dramatically faster on Four$\mathbb{Q}$

Comparison with other 128-bit security curves

- But… **results are even better in some practical scenarios!**
- For example, genus 2 Kummer does not support efficient fixed-base scalar multiplications
- Therefore, scenarios such as **ephemeral Diffie-Hellman key exchange** or **digital signatures** can be dramatically faster on Four$\mathbb{Q}$

Cycles to compute scalar multiplication during signing (in 10^4 cycles)

Curve	Field	Cortex-A7	Cortex-A8	Cortex-A9	Cortex-A15
Four$\mathbb{Q}$ (this work)	$\mathbb{F}_{p^2},\ p = 2^{127} - 1$	**204**	**144**	**145**	**84**
Kummer (Gaudry-Schost)	$\mathbb{F}_{p},\ p = 2^{127} - 1$	580 *	305	356	224 *

Kummer: implementations by Bernstein et al [BCL+14]. Results from [eBACS]. Assuming that the cost is dominated by one ladder computation. Costs are slightly higher according to [CCS15].

** Results obtained by running SUPERCOP on the targeted machine.*

Comparison with other 128-bit security curves

In summary, for signing, it is estimated that Four$\mathbb{Q}$ is:

- At least between **2.1—2.8 times faster** than genus 2 Kummer

Relevant links

- The code is now part of Four$\mathbb{Q}$lib, version 2.0:

  Download it from: http://research.microsoft.com/en-us/projects/fourqlib/

References

[Ber06] D.J. Bernstein. Curve25519: New Diffie-Hellman speed records. PKC 2006.

[BBJ+08] D.J. Bernstein, P. Birkner, M. Joye, T. Lange and C. Peters. Twisted Edwards curves. AFRICACRYPT 2008.

[BCL+14] D. J. Bernstein, C. Chuengsatiansup, T. Lange and P. Schwabe. Kummer strikes back: New DH speed records. ASIACRYPT 2014.

[eBACS] D.J. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems, accessed on May 15, 2016. http://bench.cr.yp.to/results-dh.html

[CCS15] P. N. Chung, C. Costello and B. Smith: Fast, uniform, and compact scalar multiplication for elliptic curves and genus 2 Jacobians with applications to signature schemes, Cryptology ePrint Archive: Report 2015/983, 2015.

[Edw07] H. Edwards. A normal form for elliptic curves. Bulletin of the AMS, 2007.

[GLS09] S.D. Galbraith, X. Lin, M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. EUROCRYPT 2009.

[GLV01] R.P. Gallant, R.J. Lambert, S.A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. CRYPTO 2001.

[GS12] P. Gaudry and E. Schost. Genus 2 point counting over prime fields. J. Symbolic Computation, 2012.

[GI13] A. Guillevic and S. Ionica. Four-dimensional GLV via the Weil restriction. ASIACRYPT 2013.

[HCW+08] H. Hisil, G. Carter, K.K. Wong and E. Dawson. Twisted Edwards curves revisited. ASIACRYPT 2008.

[Smi16] B. Smith. The Q-curve construction for endomorphism-accelerated elliptic curves. J. Cryptology, 2016.

Four$\mathbb{Q}$NEON: Faster Elliptic Curve Scalar Multiplications on ARM Processors

Selected Areas in Cryptography (SAC 2016)
St. Johns, Canada

Patrick Longa
Website: *http://research.microsoft.com/en-us/people/plonga/*
Twitter: @PatrickLonga

Microsoft Research