ECC Summer School, Bordeaux, France — September 23–25, 2015

Software and Hardware Implementation of Elliptic Curve Cryptography

Jérémie Detrey

CARAMEL team, LORIA

INRIA Nancy – Grand Est, France

Jeremie.Detrey@loria.fr

Context: Elliptic curves

- Let us consider a finite field $\mathbb{F}_q$ and an elliptic curve $E/\mathbb{F}_q$

  e.g., $E : y^2 = x^3 + Ax + B$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2, 3$

Context: Elliptic curves

- Let us consider a finite field $\mathbb{F}_q$ and an elliptic curve $E/\mathbb{F}_q$

  e.g., $E : y^2 = x^3 + Ax + B$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2, 3$

- The set of $\mathbb{F}_q$-rational points of E is defined as

$$E(\mathbb{F}_q) = \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q \mid (x, y) \text{ satisfy } E\} \cup \{\mathcal{O}\}$$

Context: Elliptic curves

- Let us consider a finite field $\mathbb{F}_q$ and an elliptic curve $E/\mathbb{F}_q$

  e.g., $E : y^2 = x^3 + Ax + B$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2, 3$

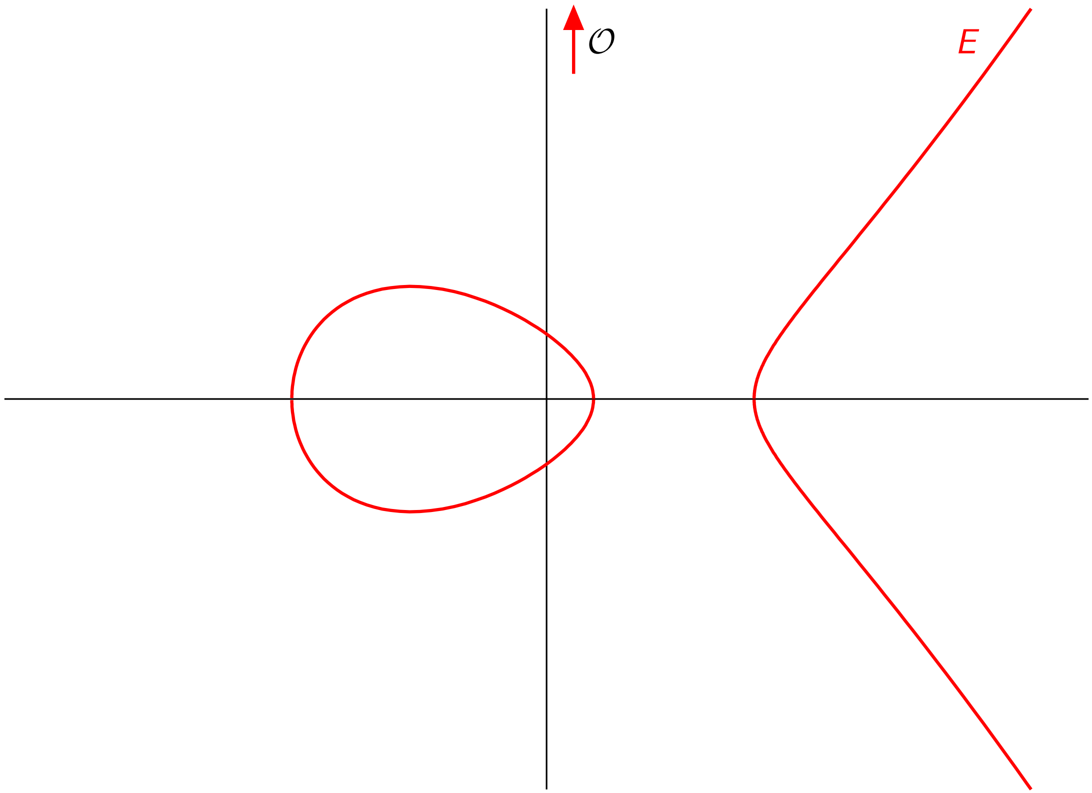
- The set of $\mathbb{F}_q$-rational points of E is defined as

$$E(\mathbb{F}_q) = \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q \mid (x, y) \text{ satisfy } E\} \cup \{\mathcal{O}\}$$

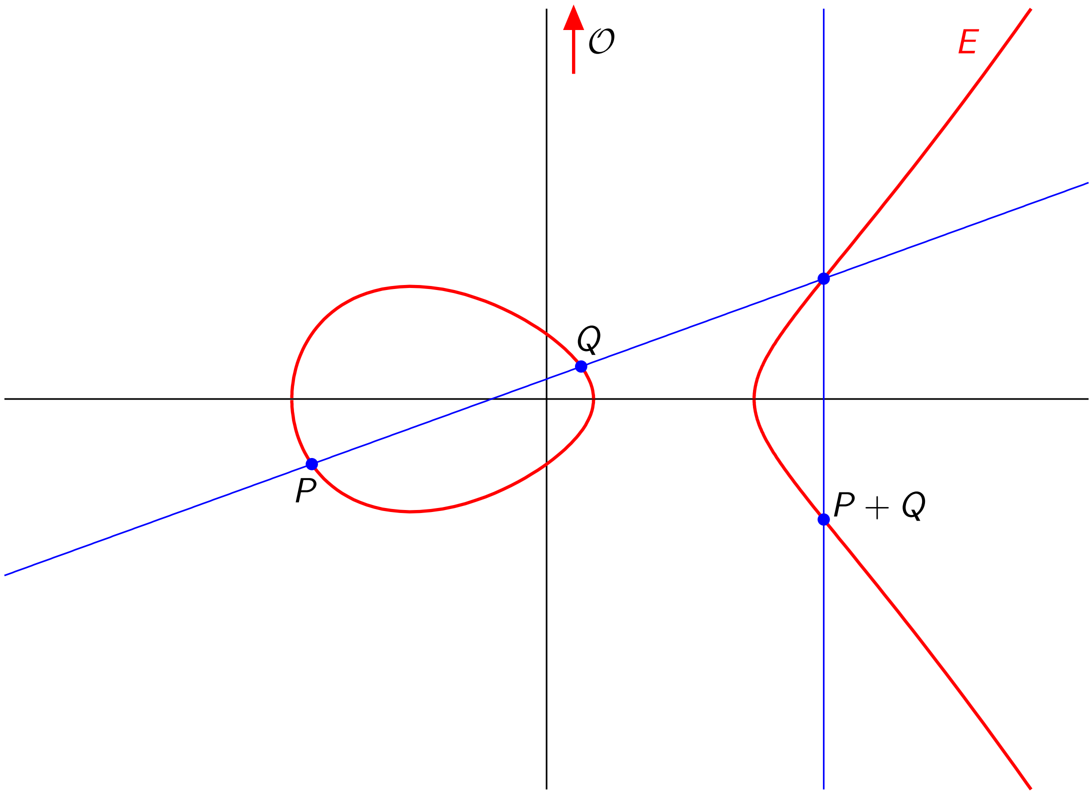
- Additive group law: $E(\mathbb{F}_q)$ is an abelian group
  - addition via the "chord and tangent" method
  - $\mathcal{O}$ is the neutral element

  [See D. Robert's lectures]

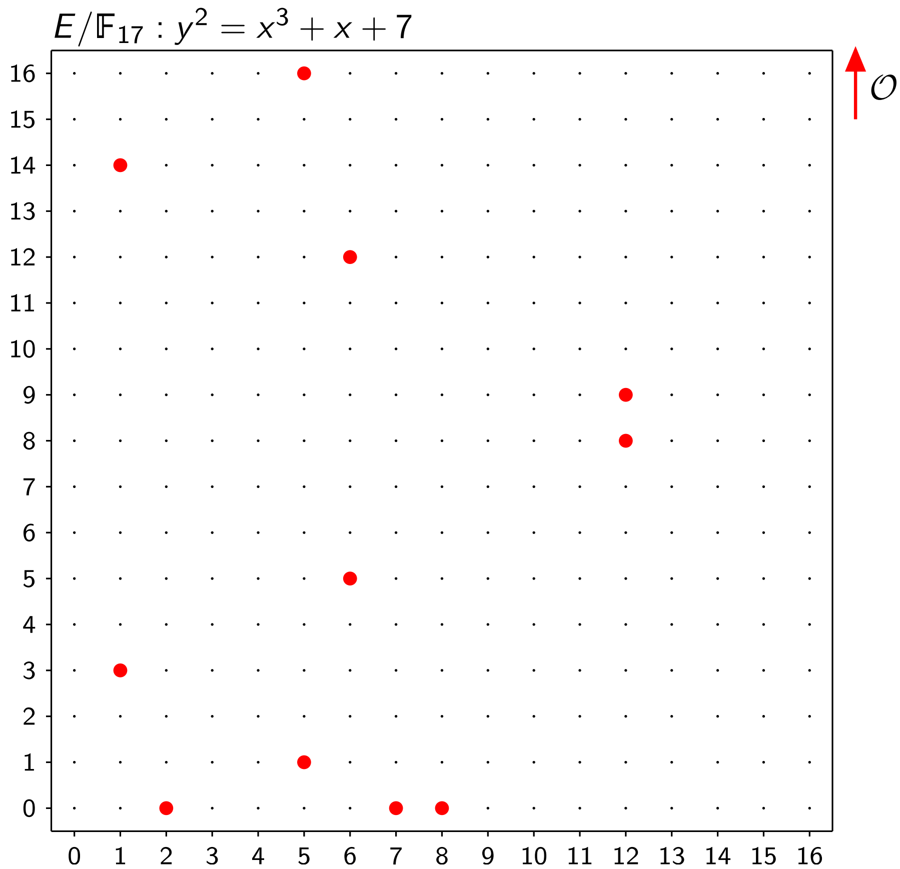
The group law



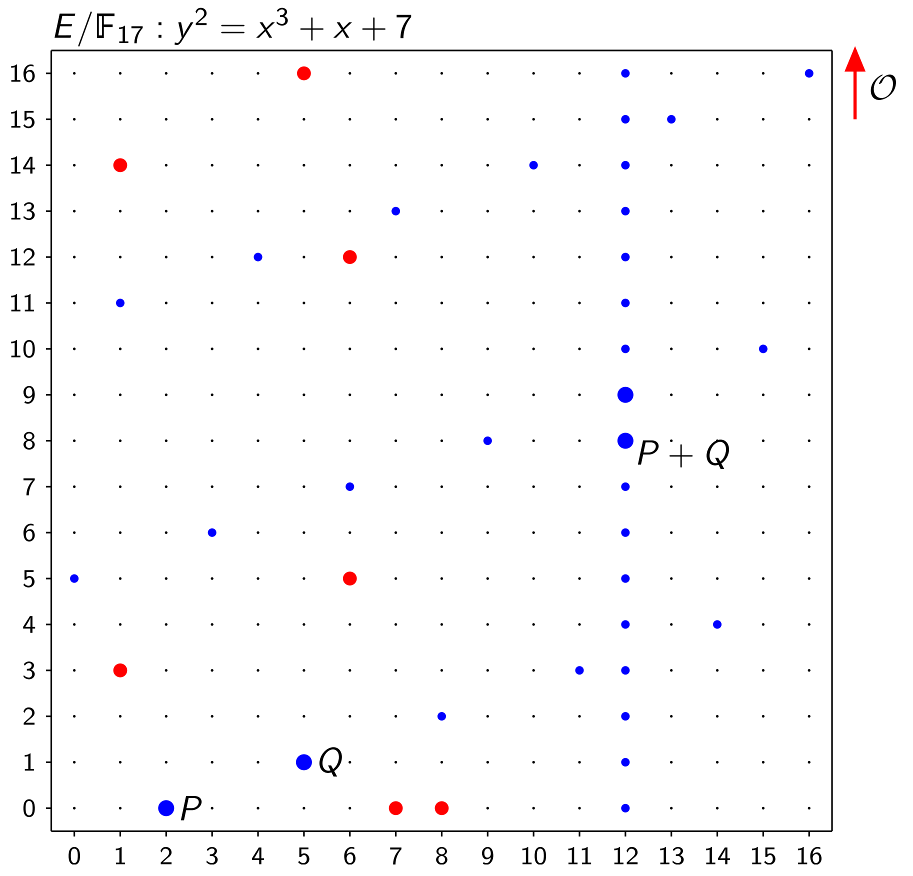
The group law



The group law



The group law



Scalar multiplication and discrete logarithm

- $E(\mathbb{F}_q)$ is a finite abelian group:
  - let $\mathbb{G}$ be a cyclic subgroup of $E(\mathbb{F}_q)$
  - let $\ell = \#\mathbb{G}$ the order of $\mathbb{G}$ and $P \in \mathbb{G}$ a generator of $\mathbb{G}$

Scalar multiplication and discrete logarithm

- $E(\mathbb{F}_q)$ is a finite abelian group:
  - let $\mathbb{G}$ be a cyclic subgroup of $E(\mathbb{F}_q)$
  - let $\ell = \#\mathbb{G}$ the order of $\mathbb{G}$ and $P \in \mathbb{G}$ a generator of $\mathbb{G}$

$$\mathbb{G} = \langle P \rangle = \{\mathcal{O}, P, 2P, 3P, \ldots, (\ell - 1)P\}$$

Scalar multiplication and discrete logarithm

- $E(\mathbb{F}_q)$ is a finite abelian group:
  - let $\mathbb{G}$ be a cyclic subgroup of $E(\mathbb{F}_q)$
  - let $\ell = \#\mathbb{G}$ the order of $\mathbb{G}$ and $P \in \mathbb{G}$ a generator of $\mathbb{G}$

$$\mathbb{G} = \langle P \rangle = \{\mathcal{O}, P, 2P, 3P, \ldots, (\ell - 1)P\}$$

- The scalar multiplication in base P gives an isomorphism between $\mathbb{Z}/\ell\mathbb{Z}$ and $\mathbb{G}$:

$$\begin{array}{rcccl} \exp_P & : & \mathbb{Z}/\ell\mathbb{Z} & \longrightarrow & \mathbb{G} \\ & & k & \longmapsto & kP = \underbrace{P + P + \ldots + P}_{k \text{ times}} \end{array}$$

Scalar multiplication and discrete logarithm

- $E(\mathbb{F}_q)$ is a finite abelian group:
  - let $\mathbb{G}$ be a cyclic subgroup of $E(\mathbb{F}_q)$
  - let $\ell = \#\mathbb{G}$ the order of $\mathbb{G}$ and $P \in \mathbb{G}$ a generator of $\mathbb{G}$

$$\mathbb{G} = \langle P \rangle = \{\mathcal{O}, P, 2P, 3P, \ldots, (\ell - 1)P\}$$

- The scalar multiplication in base P gives an isomorphism between $\mathbb{Z}/\ell\mathbb{Z}$ and $\mathbb{G}$:

$$\begin{array}{rccl} \exp_P : & \mathbb{Z}/\ell\mathbb{Z} & \longrightarrow & \mathbb{G} \\ & k & \longmapsto & kP = \underbrace{P + P + \ldots + P}_{k \text{ times}} \end{array}$$

- The inverse map is the so-called discrete logarithm (in base P):

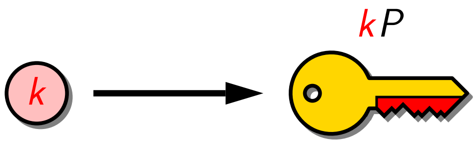
$$\begin{array}{rccl} \mathrm{dlog}_P = \exp_P^{-1} : & \mathbb{G} & \longrightarrow & \mathbb{Z}/\ell\mathbb{Z} \\ & Q & \longmapsto & k \qquad \text{such that } Q = kP \end{array}$$

Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

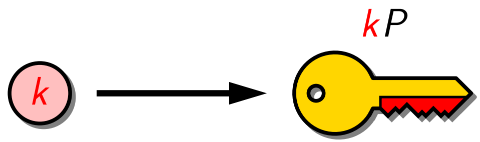
Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

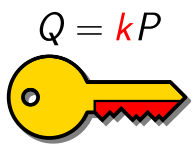


Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

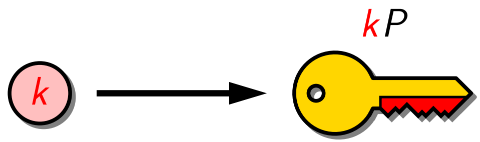


- Under a few conditions, discrete logarithm can only be computed in exponential time (as far as we know):

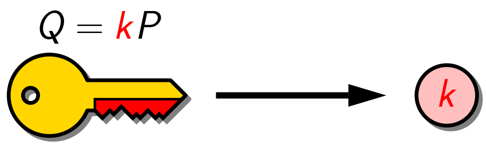


Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

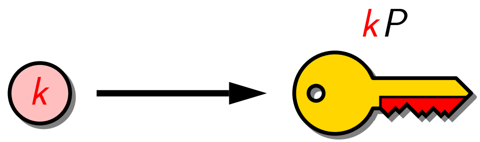


- Under a few conditions, discrete logarithm can only be computed in exponential time (as far as we know):

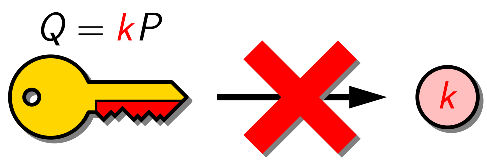


Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:



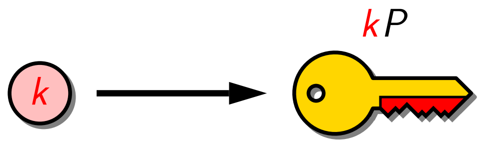
- Under a few conditions, discrete logarithm can only be computed in exponential time (as far as we know):



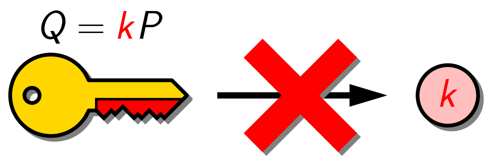
[See E. Thomé's lectures, and S. Galbraith's and M. Kosters' talks]

Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:



- Under a few conditions, discrete logarithm can only be computed in exponential time (as far as we know):

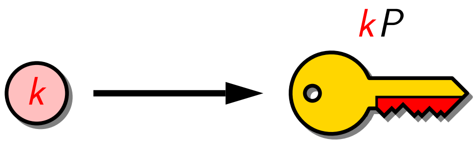


[See E. Thomé's lectures, and S. Galbraith's and M. Kosters' talks]

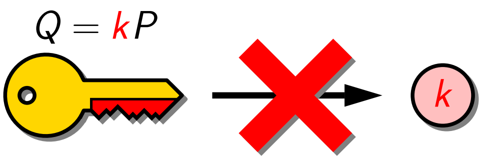
- That's a one-way function

Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:



- Under a few conditions, discrete logarithm can only be computed in exponential time (as far as we know):

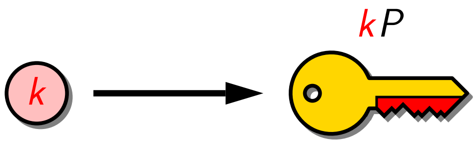


[See E. Thomé's lectures, and S. Galbraith's and M. Kosters' talks]

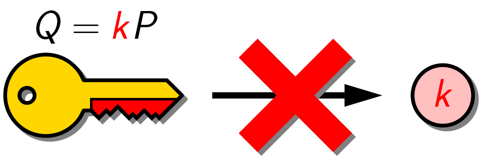
- That's a one-way function ⇒ Public-key cryptography!

Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:



- Under a few conditions, discrete logarithm can only be computed in exponential time (as far as we know):



[See E. Thomé's lectures, and S. Galbraith's and M. Kosters' talks]

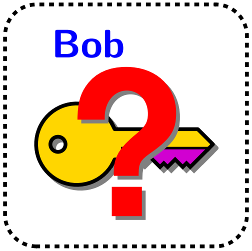
- That's a one-way function $\Rightarrow$ Public-key cryptography!
  - private key: an integer k in $\mathbb{Z}/\ell\mathbb{Z}$
  - public key: the point kP in $\mathbb{G} \subseteq E(\mathbb{F}_q)$

Example 1: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel

Example 1: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?



Example 1: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?

Example 1: EC Diffie–Hellman key exchange

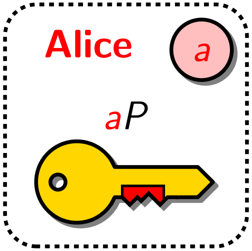
- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?

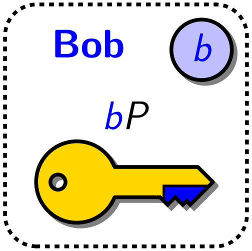
Example 1: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?

Example 1: EC Diffie–Hellman key exchange

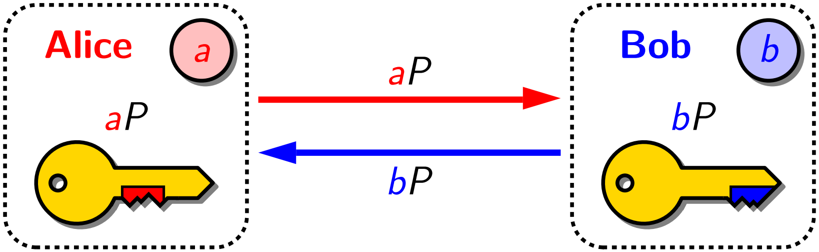
- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?





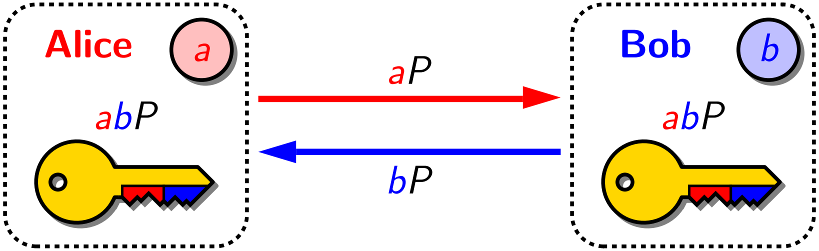
Example 1: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?

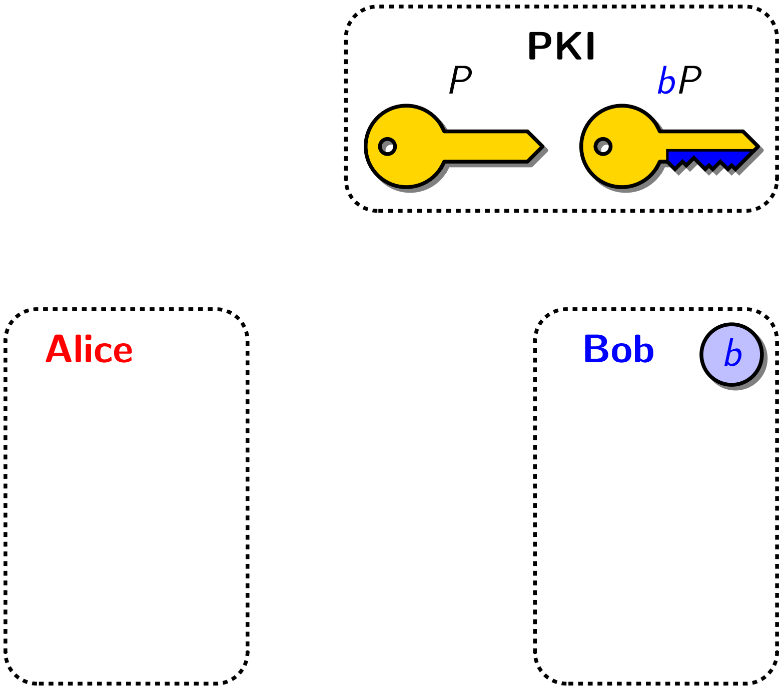


Example 1: EC Diffie–Hellman key exchange

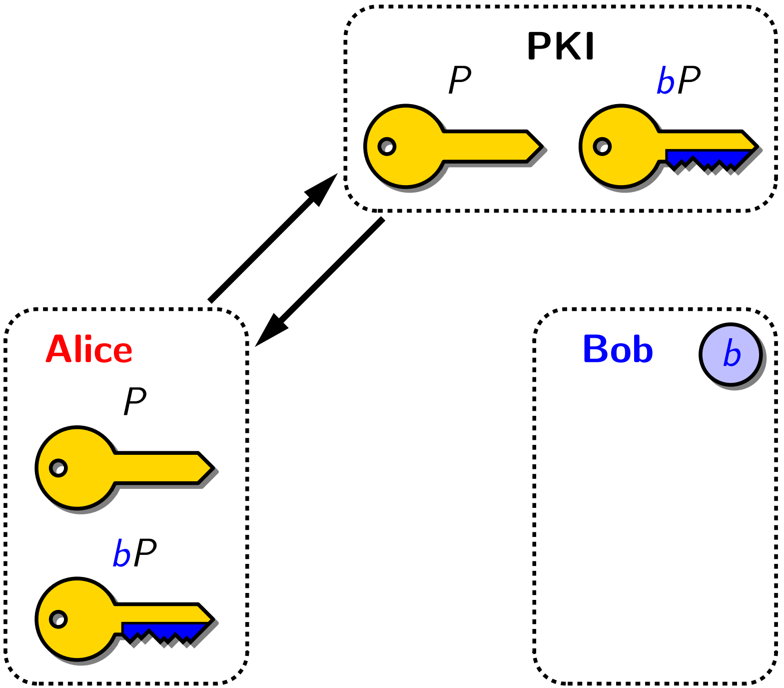
- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?



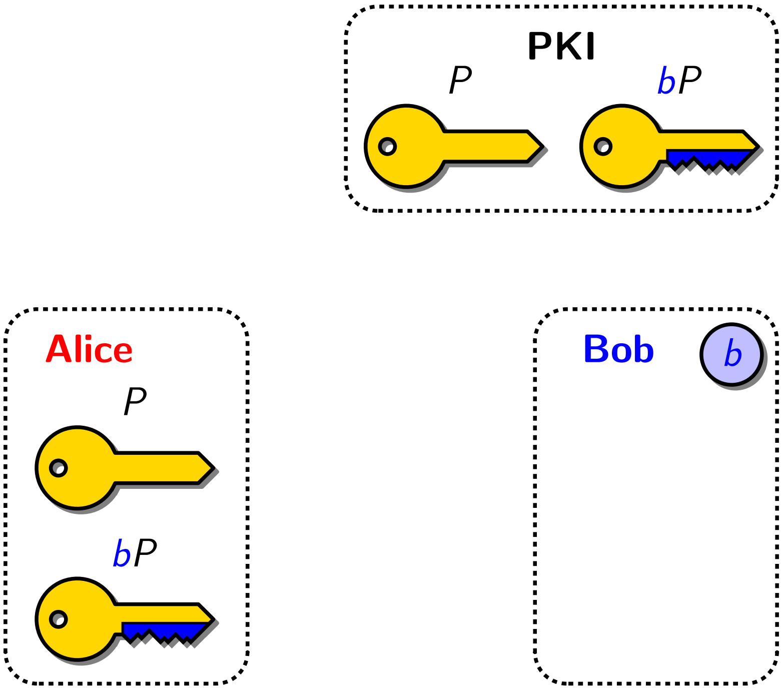
Example 2: EC ElGamal encryption



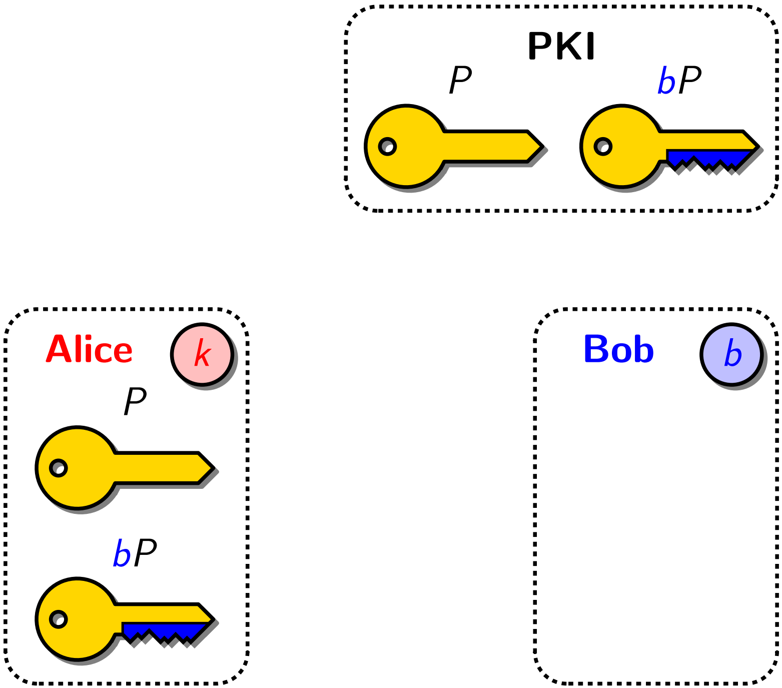
Example 2: EC ElGamal encryption



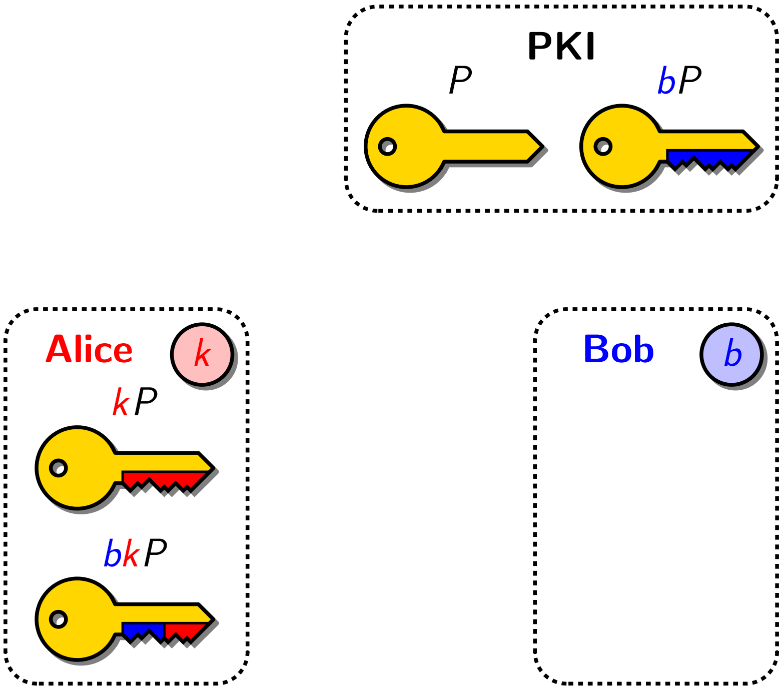
Example 2: EC ElGamal encryption



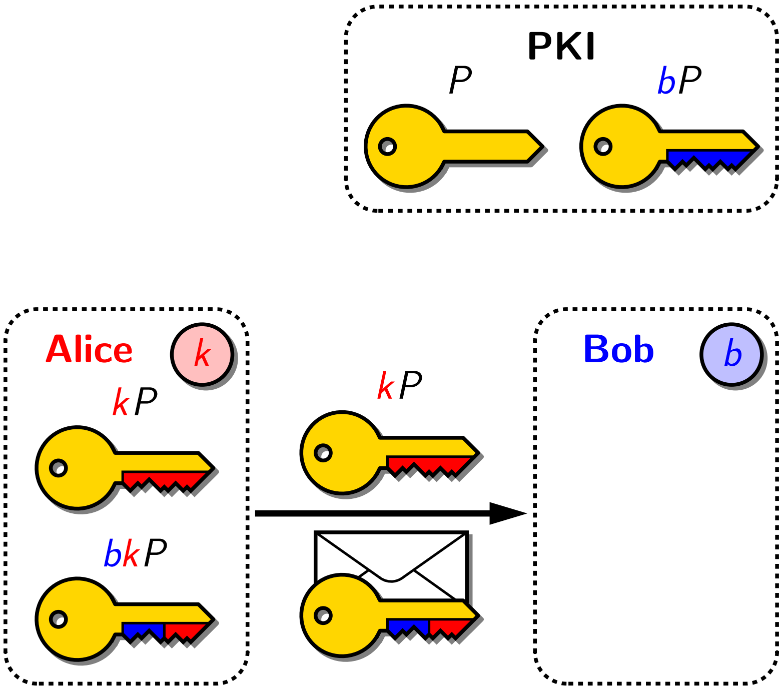
Example 2: EC ElGamal encryption



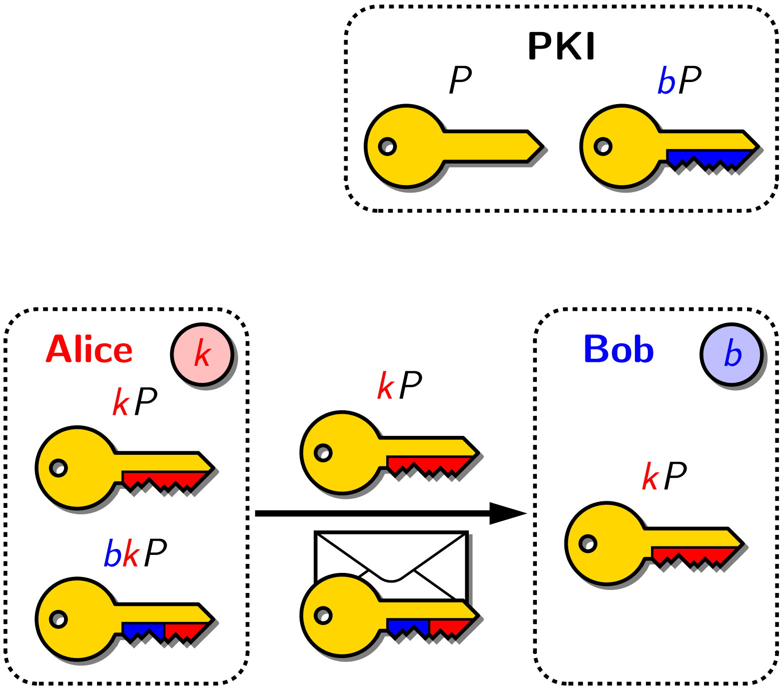
Example 2: EC ElGamal encryption



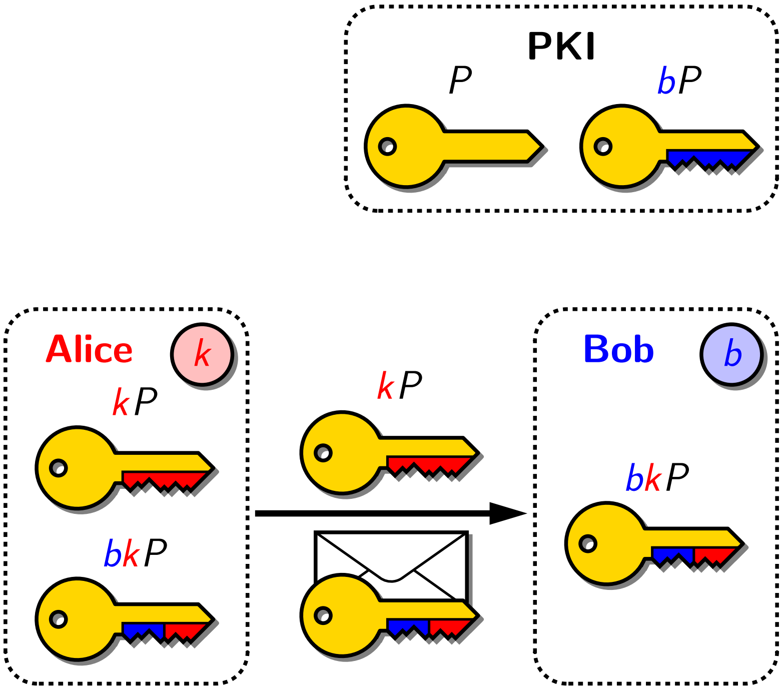
Example 2: EC ElGamal encryption



Example 2: EC ElGamal encryption



Example 2: EC ElGamal encryption



Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (1 double scalar mult)

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (1 double scalar mult)
- etc.

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (1 double scalar mult)
- etc.
- Other important operations might be required, such as pairings [See J. Krämer's talk]

Efficient and secure implementation?

- Many possible meanings for efficiency:

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (FREAK, LogJam, etc.) [See N. Heninger's talk]

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (FREAK, LogJam, etc.) [See N. Heninger's talk]
  - curve attacks? (weak curves, twist security, etc.)

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (FREAK, LogJam, etc.) [See N. Heninger's talk]
  - curve attacks? (weak curves, twist security, etc.)
  - timing attacks? [See P. Schwabe's talk]

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (FREAK, LogJam, etc.) [See N. Heninger's talk]
  - curve attacks? (weak curves, twist security, etc.)
  - timing attacks? [See P. Schwabe's talk]
  - fault attacks? [See J. Krämer's talk]

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? → low latency or high throughput?
  - small? → low memory / code / silicon usage?
  - low power?... or low energy?

  ⇒ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (FREAK, LogJam, etc.) [See N. Heninger's talk]
  - curve attacks? (weak curves, twist security, etc.)
  - timing attacks? [See P. Schwabe's talk]
  - fault attacks? [See J. Krämer's talk]
  - cache attacks?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? → low latency or high throughput?
  - small? → low memory / code / silicon usage?
  - low power?... or low energy?

  ⇒ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (FREAK, LogJam, etc.) [See N. Heninger's talk]
  - curve attacks? (weak curves, twist security, etc.)
  - timing attacks? [See P. Schwabe's talk]
  - fault attacks? [See J. Krämer's talk]
  - cache attacks?
  - branch-prediction attacks?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (FREAK, LogJam, etc.) [See N. Heninger's talk]
  - curve attacks? (weak curves, twist security, etc.)
  - timing attacks? [See P. Schwabe's talk]
  - fault attacks? [See J. Krämer's talk]
  - cache attacks?
  - branch-prediction attacks?
  - power or electromagnetic analysis?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (FREAK, LogJam, etc.) [See N. Heninger's talk]
  - curve attacks? (weak curves, twist security, etc.)
  - timing attacks? [See P. Schwabe's talk]
  - fault attacks? [See J. Krämer's talk]
  - cache attacks?
  - branch-prediction attacks?
  - power or electromagnetic analysis?
  - etc.

  $\Rightarrow$ Possible attack scenarios depend on the application

Which target platforms?

- Cryptography should be available everywhere:

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    $\rightarrow$ low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    $\rightarrow$ low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)
  - on wireless sensors
    $\rightarrow$ tiny 8-bit microcontroller (such as Atmel AVRs)

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    $\rightarrow$ low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)
  - on wireless sensors
    $\rightarrow$ tiny 8-bit microcontroller (such as Atmel AVRs)
  - on smart cards and RFID chips
    $\rightarrow$ custom cryptoprocessor (ASIC or ASIP) with dedicated hardware for cryptographic operations

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    → 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    → low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)
  - on wireless sensors
    → tiny 8-bit microcontroller (such as Atmel AVRs)
  - on smart cards and RFID chips
    → custom cryptoprocessor (ASIC or ASIP) with dedicated hardware for cryptographic operations
- Other possible target platforms, mostly for cryptanalytic computations:
  - clusters of CPUs
  - GPUs (graphics processors)
  - FPGAs (reconfigurable circuits)

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    → 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    → low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)
  - on wireless sensors
    → tiny 8-bit microcontroller (such as Atmel AVRs)
  - on smart cards and RFID chips
    → custom cryptoprocessor (ASIC or ASIP) with dedicated hardware for cryptographic operations
- Other possible target platforms, mostly for cryptanalytic computations:
  - clusters of CPUs
  - GPUs (graphics processors)
  - FPGAs (reconfigurable circuits)

  ⇒ In such cases, implementation security is usually less critical

Implementation layers

- A complete ECC implementation relies on many layers:

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)
  - finite field arithmetic (addition, multiplication, inversion, etc.)

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)
  - finite field arithmetic (addition, multiplication, inversion, etc.)
  - native integer arithmetic (CPU instruction set)

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)
  - finite field arithmetic (addition, multiplication, inversion, etc.)
  - native integer arithmetic (CPU instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)
  - finite field arithmetic (addition, multiplication, inversion, etc.)
  - native integer arithmetic (CPU instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)
  - finite field arithmetic (addition, multiplication, inversion, etc.)
  - native integer arithmetic (CPU instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires
  - transistors

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)
  - finite field arithmetic (addition, multiplication, inversion, etc.)
  - native integer arithmetic (CPU instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires
  - transistors
- When designing a cryptoprocessor, the hardware/software partitioning can be tailored to the application's requirements

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)
  - finite field arithmetic (addition, multiplication, inversion, etc.)
  - native integer arithmetic (CPU instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires
  - transistors
- When designing a cryptoprocessor, the hardware/software partitioning can be tailored to the application's requirements
- All top layers (esp. the blue and green ones) might lead to critical vulnerabilities if poorly implemented!
  $\Rightarrow$ ECC is no more secure than its weakest link

Implementation layers

- A complete ECC implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic primitives (ECDH, ECDSA, etc.)
  - **scalar multiplication**
  - **elliptic curve arithmetic** (point addition, point doubling, etc.)
  - **finite field arithmetic** (addition, multiplication, inversion, etc.)
  - native integer arithmetic (CPU instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires
  - transistors

- When designing a cryptoprocessor, the hardware/software partitioning can be tailored to the application's requirements

- All top layers (esp. the blue and green ones) might lead to critical vulnerabilities if poorly implemented!
  $\Rightarrow$ ECC is no more secure than its weakest link

- In these lectures, we will mostly focus on the **green layers**

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)
  - at the field arithmetic level: MPFQ, GF2X, NTL, GMP, etc.

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)
  - at the field arithmetic level: MPFQ, GF2X, NTL, GMP, etc.

- Available open-source hardware implementations of ECC:

Available implementations

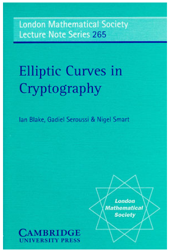
- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)
  - at the field arithmetic level: MPFQ, GF2X, NTL, GMP, etc.

- Available open-source hardware implementations of ECC:
  - implementation of NaCl's `crypto_box` [Ask P. Schwabe about it]

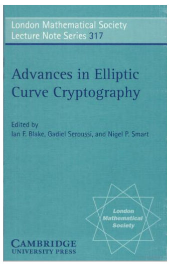
Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)
  - at the field arithmetic level: MPFQ, GF2X, NTL, GMP, etc.
- Available open-source hardware implementations of ECC:
  - implementation of NaCl's `crypto_box` [Ask P. Schwabe about it]
  - PAVOIS project (announced) [See A. Tisserand's talk]

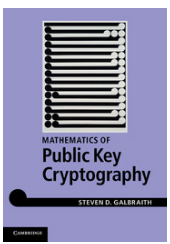
Some references



Elliptic Curves in Cryptography,
Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart.
London Mathematical Society 265,
Cambridge University Press, 1999.

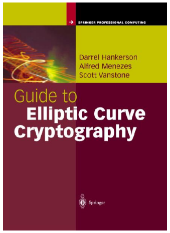


Advances in Elliptic Curves Cryptography,
Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart (editors).
London Mathematical Society 317,
Cambridge University Press, 2005.

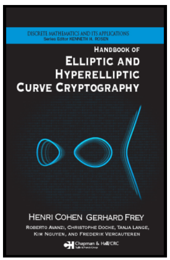


Mathematics of Public-Key Cryptography,
Steven D. Galbraith.
Cambridge University Press, 2012.

Some references



Guide to Elliptic Curve Cryptography,
Darrel Hankerson, Alfred Menezes, and Scott Vanstone.
Springer, 2004.



Handbook of Elliptic and Hyperelliptic Curve Cryptography,
Henri Cohen and Gerhard Frey (editors).
Chapman & Hall / CRC, 2005.

Proceedings of the CHES workshop and of other crypto conferences.

Outline

I. Scalar multiplication

II. Elliptic curve arithmetic

III. Finite field arithmetic

IV. Software considerations

V. Notions of hardware design

Outline

I. Scalar multiplication

II. Elliptic curve arithmetic

III. Finite field arithmetic

IV. Software considerations

V. Notions of hardware design

Scalar multiplication

- Given k in $\mathbb{Z}/\ell\mathbb{Z}$ and P in $\mathbb{G} \subseteq E(\mathbb{F}_q)$, we want to compute

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

Scalar multiplication

- Given k in $\mathbb{Z}/\ell\mathbb{Z}$ and P in $\mathbb{G} \subseteq E(\mathbb{F}_q)$, we want to compute

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

- Size of ℓ (and k) for crypto applications: between 250 and 500 bits

Scalar multiplication

- Given k in $\mathbb{Z}/\ell\mathbb{Z}$ and P in $\mathbb{G} \subseteq E(\mathbb{F}_q)$, we want to compute

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

- Size of ℓ (and k) for crypto applications: between 250 and 500 bits
- Repeated addition, in $O(k)$ complexity, is out of the question!

Double-and-add algorithm

- Available operations on $E(\mathbb{F}_q)$:
  - point addition: $(Q, R) \mapsto Q + R$
  - point doubling: $Q \mapsto 2Q = Q + Q$

Double-and-add algorithm

- Available operations on $E(\mathbb{F}_q)$:
  - point addition: $(Q, R) \mapsto Q + R$
  - point doubling: $Q \mapsto 2Q = Q + Q$
- Idea: iterative algorithm based on the binary expansion of k

Double-and-add algorithm

- Available operations on $E(\mathbb{F}_q)$:
  - point addition: $(Q, R) \mapsto Q + R$
  - point doubling: $Q \mapsto 2Q = Q + Q$
- Idea: iterative algorithm based on the binary expansion of k
  - start from the most significant bit of k
  - double current result at each step
  - add P if the corresponding bit of k is 1

Double-and-add algorithm

- Available operations on $E(\mathbb{F}_q)$:
  - point addition: $(Q, R) \mapsto Q + R$
  - point doubling: $Q \mapsto 2Q = Q + Q$
- Idea: iterative algorithm based on the binary expansion of k
  - start from the most significant bit of k
  - double current result at each step
  - add P if the corresponding bit of k is 1
  - same principle as binary exponentiation

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

```
function scalar-mult(k, P):
    T ← O
    for i ← n − 1 downto 0:
        T ← 2T
        if k_i = 1:
            T ← T + P
    return T
```

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
$T \leftarrow \mathcal{O}$
for $i \leftarrow n-1$ **downto** 0**:**
$T \leftarrow 2T$
if $k_i = 1$**:**
$T \leftarrow T + P$
return T

- Example: $k = 431$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101111)_2$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

```
function scalar-mult(k, P):
    T ← O
    for i ← n − 1 downto 0:
        T ← 2T
        if k_i = 1:
            T ← T + P
    return T
```

- Example: $k = 431 = (110101111)_2$

$$T = \qquad = \mathcal{O}$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
    $T \leftarrow \mathcal{O}$
    for $i \leftarrow n - 1$ **downto** 0**:**
        $T \leftarrow 2T$
        if $k_i = 1$**:**
            $T \leftarrow T + P$
    return T

- Example: $k = 431 = (\underline{1}10101111)_2$

$T = \quad P \quad\quad = \quad P$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

$$
\begin{array}{l}
\textbf{function } \text{scalar-mult}(k, P)\textbf{:} \\
\quad T \leftarrow \mathcal{O} \\
\quad \textbf{for } i \leftarrow n-1 \textbf{ downto } 0\textbf{:} \\
\quad\quad T \leftarrow 2T \\
\quad\quad \textbf{if } k_i = 1\textbf{:} \\
\quad\quad\quad T \leftarrow T + P \\
\quad \textbf{return } T
\end{array}
$$

- Example: $k = 431 = (1\underline{1}0101111)_2$

$$T = P \cdot 2 = 2P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n - 1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (1\underline{1}0101111)_2$

$$T = \quad P \cdot 2 + P \quad = \quad 3P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
    $T \leftarrow \mathcal{O}$
    for $i \leftarrow n - 1$ **downto** 0**:**
        $T \leftarrow 2T$
        if $k_i = 1$**:**
            $T \leftarrow T + P$
    return T

- Example: $k = 431 = (11\underline{0}101111)_2$

$$T = (P \cdot 2 + P) \cdot 2 = 6P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

$$
\begin{array}{l}
\textbf{function } \text{scalar-mult}(k, P)\textbf{:} \\
\quad T \leftarrow \mathcal{O} \\
\quad \textbf{for } i \leftarrow n-1 \textbf{ downto } 0\textbf{:} \\
\quad\quad T \leftarrow 2T \\
\quad\quad \textbf{if } k_i = 1\textbf{:} \\
\quad\quad\quad T \leftarrow T + P \\
\quad \textbf{return } T
\end{array}
$$

- Example: $k = 431 = (110\underline{1}01111)_2$

$$T = (P \cdot 2 + P) \cdot 2^2 = 12P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110\underline{1}01111)_2$

$$T = \quad (P \cdot 2 + P) \cdot 2^2 + P \quad = \quad 13P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (1101\underline{0}1111)_2$

$$T = ((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2 = 26P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101\underline{1}11)_2$

$$T = \quad ((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 \quad = \quad 52P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
    $T \leftarrow \mathcal{O}$
    for $i \leftarrow n-1$ **downto** 0**:**
        $T \leftarrow 2T$
        if $k_i = 1$**:**
            $T \leftarrow T + P$
    return T

- Example: $k = 431 = (110101111)_2$

$$T = ((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P = 53P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101\underline{1}11)_2$

$$T = (((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 \qquad = 106P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101\underline{1}11)_2$

$$T = \quad (((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P \qquad = 107P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101111)_2$

$$T = ((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 \qquad = 214P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n - 1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101111)_2$

$$T = ((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P \qquad = 215P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
    $T \leftarrow \mathcal{O}$
    for $i \leftarrow n-1$ **downto** 0**:**
        $T \leftarrow 2T$
        if $k_i = 1$**:**
            $T \leftarrow T + P$
    return T

- Example: $k = 431 = (11010111\underline{1})_2$

$$T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 \qquad = 430P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n - 1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (11010111\underline{1})_2$

$$T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 + P = 431P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
    $T \leftarrow \mathcal{O}$
    for $i \leftarrow n-1$ **downto** 0**:**
        $T \leftarrow 2T$
        if $k_i = 1$**:**
            $T \leftarrow T + P$
    return T

- Example: $k = 431 = (110101111)_2$

$$T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 + P = 431P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
$T \leftarrow \mathcal{O}$
for $i \leftarrow n-1$ **downto** 0**:**
$T \leftarrow 2T$
if $k_i = 1$**:**
$T \leftarrow T + P$
return T

- Example: $k = 431 = (110101111)_2$

$$T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 + P = 431P$$

- Complexity in $O(n) = O(\log_2 \ell)$ operations over $E(\mathbb{F}_q)$:
  - n doublings, and
  - $n/2$ additions on average

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, $\ldots$, $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = \qquad\qquad\qquad = \quad \mathcal{O}$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (\underline{110}\,101\,111)_2 = (\underline{6}57)_{2^3}$

$$T = 6P \qquad = \quad 6P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,\underline{101}\,111)_2 = (6\underline{5}7)_{2^3}$

$$T = 6P \cdot 2^3 \qquad = 48P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, $\ldots$, $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,\underline{101}\,111)_2 = (6\underline{5}7)_{2^3}$

$$T = 6P \cdot 2^3 + 5P \qquad = 53P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,\underline{111})_2 = (65\underline{7})_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 \qquad = 424P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, $\ldots$, $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,\underline{111})_2 = (65\underline{7})_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

- Complexity:
  - n doublings, and
  - $(1 - 2^{-w})n/w$ additions on average

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

- Complexity:
  - n doublings, and
  - $(1 - 2^{-w})n/w$ additions on average
- Select w carefully so that precomputation cost does not become predominant

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

- Complexity:
  - n doublings, and
  - $(1 - 2^{-w})n/w$ additions on average
- Select w carefully so that precomputation cost does not become predominant
- Sliding window variant: half as many precomputations

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w=3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

$$T = \qquad = \quad \mathcal{O}$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, $\ldots$, $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (\underline{3}003000\bar{1})_2$

$$T = 3P \qquad = \quad 3P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (300300 0\bar{1})_2$

$$T = 3P \cdot 2 \quad = \quad 6P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (300\underline{0}3000\bar{1})_2$

$$T = 3P \cdot 2^2 = 12P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

$$T = 3P \cdot 2^3 \qquad = 24P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (300\underline{3}000\bar{1})_2$

$$T = 3P \cdot 2^3 + 3P \qquad = 27P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, $\ldots$, $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003\underline{0}00\bar{1})_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2 \qquad = \quad 54P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, …, $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w=3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2^2 \qquad = 108P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \dots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003000\underline{0}\bar{1})_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2^3 \qquad = 216P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, $\ldots$, $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (300300\underline{0}\bar{1})_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2^4 \qquad = 432P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003000\underline{\bar{1}})_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2^4 - P = 431P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2^4 - P = 431P$$

Non-adjacent form

- Fact: computing the opposite of a point on $E(\mathbb{F}_q)$ has a negligible cost
- Idea: use signed digits to represent scalar k with minimal Hamming weight
- 2^w-ary non-adjacent form (w-NAF): use odd digits $\{-2^{w-1}+1, \ldots, 2^{w-1}-1\}$ and 0 to represent k so that at most every w-th digit is non-zero
- Precompute $3P$, $5P$, ..., $(2^{w-1}-1)P$:
  - 1 doubling, and
  - $2^{w-2}-1$ additions
- Example with $w = 3$ (digits in $\{\bar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2^4 - P = 431P$$

- Complexity:
  - n doublings, and
  - $n/(w+1)$ additions on average

Multi-exponentiation technique

- To compute the sum of several scalar multiplications

$$\text{e.g., } aP + bQ, \text{ where } a, b \in \mathbb{Z}/\ell\mathbb{Z} \text{ and } P, Q \in E(\mathbb{F}_q)$$

Multi-exponentiation technique

- To compute the sum of several scalar multiplications

$$\text{e.g., } aP + bQ, \text{ where } a, b \in \mathbb{Z}/\ell\mathbb{Z} \text{ and } P, Q \in E(\mathbb{F}_q)$$

- Idea:
  - compute and accumulate all scalar multiplications simultaneously
  - share doubling steps between multiplications

function double-scalar-mult(a, P, b, Q)**:**
- $S \leftarrow P + Q$
- $T \leftarrow \mathcal{O}$
- **for** $i \leftarrow n - 1$ **downto** 0**:**
  - $T \leftarrow 2T$
  - **if** $a_i = 1$ **and** $b_i = 1$**:**
    - $T \leftarrow T + S$
  - **else if** $a_i = 1$**:**
    - $T \leftarrow T + P$
  - **else if** $b_i = 1$**:**
    - $T \leftarrow T + Q$
- **return** T

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**

```
S ← P + Q
T ← O
for i ← n − 1 downto 0:
    T ← 2T
    if a_i = 1 and b_i = 1:
        T ← T + S
    else if a_i = 1:
        T ← T + P
    else if b_i = 1:
        T ← T + Q
return T
```

Multi-exponentiation technique

```
function double-scalar-mult(a, P, b, Q):
    S ← P + Q
    T ← O
    for i ← n − 1 downto 0:
        T ← 2T
        if a_i = 1 and b_i = 1:
            T ← T + S
        else if a_i = 1:
            T ← T + P
        else if b_i = 1:
            T ← T + Q
    return T
```

- Example: $a = 21$
  and $b = 30$

Multi-exponentiation technique

```
function double-scalar-mult(a, P, b, Q):
    S ← P + Q
    T ← O
    for i ← n − 1 downto 0:
        T ← 2T
        if a_i = 1 and b_i = 1:
            T ← T + S
        else if a_i = 1:
            T ← T + P
        else if b_i = 1:
            T ← T + Q
    return T
```

- Example: $a = 21 = (10101)_2$
  and $b = 30 = (11110)_2$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**
$S \leftarrow P + Q$
$T \leftarrow \mathcal{O}$
for $i \leftarrow n - 1$ **downto** 0**:**
$T \leftarrow 2T$
if $a_i = 1$ **and** $b_i = 1$**:**
$T \leftarrow T + S$
else if $a_i = 1$**:**
$T \leftarrow T + P$
else if $b_i = 1$**:**
$T \leftarrow T + Q$
return T

- Example: $a = 21 = (10101)_2$
  and $b = 30 = (11110)_2$

$T = \qquad = \qquad \mathcal{O}$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**
    $S \leftarrow P + Q$
    $T \leftarrow \mathcal{O}$
    for $i \leftarrow n - 1$ **downto** 0**:**
        $T \leftarrow 2T$
        if $a_i = 1$ **and** $b_i = 1$**:**
            $T \leftarrow T + S$
        else if $a_i = 1$**:**
            $T \leftarrow T + P$
        else if $b_i = 1$**:**
            $T \leftarrow T + Q$
    return T

- Example: $a = 21 = (\underline{1}0101)_2$
  and $b = 30 = (\underline{1}1110)_2$

$T = \quad P + Q \quad\quad = \quad P + \quad Q$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**

$S \leftarrow P + Q$
$T \leftarrow \mathcal{O}$
for $i \leftarrow n-1$ **downto** 0**:**
  $T \leftarrow 2T$
  if $a_i = 1$ **and** $b_i = 1$**:**
    $T \leftarrow T + S$
  else if $a_i = 1$**:**
    $T \leftarrow T + P$
  else if $b_i = 1$**:**
    $T \leftarrow T + Q$
return T

- Example: $a = 21 = (1\underline{0}101)_2$
  and $b = 30 = (1\underline{1}110)_2$

$$T = \quad (P + Q) \cdot 2 \qquad = \; 2P + \; 2Q$$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**

```
S ← P + Q
T ← O
for i ← n − 1 downto 0:
    T ← 2T
    if a_i = 1 and b_i = 1:
        T ← T + S
    else if a_i = 1:
        T ← T + P
    else if b_i = 1:
        T ← T + Q
return T
```

- Example: $a = 21 = (1\underline{0}101)_2$
  and $b = 30 = (1\underline{1}110)_2$

$$T = (P + Q) \cdot 2 + Q \qquad = 2P + 3Q$$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**

$S \leftarrow P + Q$
$T \leftarrow \mathcal{O}$
for $i \leftarrow n - 1$ **downto** 0**:**
  $T \leftarrow 2T$
  if $a_i = 1$ **and** $b_i = 1$**:**
    $T \leftarrow T + S$
  else if $a_i = 1$**:**
    $T \leftarrow T + P$
  else if $b_i = 1$**:**
    $T \leftarrow T + Q$
return T

- Example: $a = 21 = (10\underline{1}01)_2$
  and $b = 30 = (11\underline{1}10)_2$

$$T = ((P + Q) \cdot 2 + Q) \cdot 2 \qquad = 4P + 6Q$$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q):
- $S \leftarrow P + Q$
- $T \leftarrow \mathcal{O}$
- **for** $i \leftarrow n-1$ **downto** 0:
  - $T \leftarrow 2T$
  - **if** $a_i = 1$ **and** $b_i = 1$:
    - $T \leftarrow T + S$
  - **else if** $a_i = 1$:
    - $T \leftarrow T + P$
  - **else if** $b_i = 1$:
    - $T \leftarrow T + Q$
- **return** T

- Example: $a = 21 = (10\underline{1}01)_2$ and $b = 30 = (11\underline{1}10)_2$

$$T = ((P+Q)\cdot 2 + Q)\cdot 2 + P + Q \qquad = 5P + 7Q$$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**

$S \leftarrow P + Q$

$T \leftarrow \mathcal{O}$

for $i \leftarrow n-1$ **downto** 0**:**

$T \leftarrow 2T$

if $a_i = 1$ **and** $b_i = 1$**:**

$T \leftarrow T + S$

else if $a_i = 1$**:**

$T \leftarrow T + P$

else if $b_i = 1$**:**

$T \leftarrow T + Q$

return T

- Example: $a = 21 = (10101)_2$
  and $b = 30 = (11110)_2$

$$T = (((P + Q) \cdot 2 + Q) \cdot 2 + P + Q) \cdot 2 \qquad = 10P + 14Q$$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**
  $S \leftarrow P + Q$
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n - 1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $a_i = 1$ **and** $b_i = 1$**:**
      $T \leftarrow T + S$
    else if $a_i = 1$**:**
      $T \leftarrow T + P$
    else if $b_i = 1$**:**
      $T \leftarrow T + Q$
  return T

- Example: $a = 21 = (1010\underline{1}1)_2$ and $b = 30 = (111\underline{1}0)_2$

$$T = (((P + Q) \cdot 2 + Q) \cdot 2 + P + Q) \cdot 2 + Q \qquad = 10P + 15Q$$

Multi-exponentiation technique

```
function double-scalar-mult(a, P, b, Q):
    S ← P + Q
    T ← O
    for i ← n − 1 downto 0:
        T ← 2T
        if a_i = 1 and b_i = 1:
            T ← T + S
        else if a_i = 1:
            T ← T + P
        else if b_i = 1:
            T ← T + Q
    return T
```

- Example: $a = 21 = (1010\underline{1})_2$
  and $b = 30 = (1111\underline{0})_2$

$$T = ((((P + Q) \cdot 2 + Q) \cdot 2 + P + Q) \cdot 2 + Q) \cdot 2 \qquad = 20P + 30Q$$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**

$S \leftarrow P + Q$

$T \leftarrow \mathcal{O}$

for $i \leftarrow n-1$ **downto** 0**:**

$T \leftarrow 2T$

if $a_i = 1$ **and** $b_i = 1$**:**

$T \leftarrow T + S$

else if $a_i = 1$**:**

$T \leftarrow T + P$

else if $b_i = 1$**:**

$T \leftarrow T + Q$

return T

- Example: $a = 21 = (1010\underline{1})_2$
  and $b = 30 = (1111\underline{0})_2$

$$T = ((((P+Q)\cdot 2 + Q)\cdot 2 + P + Q)\cdot 2 + Q)\cdot 2 + P = 21P + 30Q$$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**

$S \leftarrow P + Q$
$T \leftarrow \mathcal{O}$
for $i \leftarrow n - 1$ **downto** 0**:**
  $T \leftarrow 2T$
  if $a_i = 1$ **and** $b_i = 1$**:**
    $T \leftarrow T + S$
  else if $a_i = 1$**:**
    $T \leftarrow T + P$
  else if $b_i = 1$**:**
    $T \leftarrow T + Q$
return T

- Example: $a = 21 = (10101)_2$
  and $b = 30 = (11110)_2$

$$T = ((((P + Q) \cdot 2 + Q) \cdot 2 + P + Q) \cdot 2 + Q) \cdot 2 + P = 21P + 30Q$$

Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q)**:**

$S \leftarrow P + Q$
$T \leftarrow \mathcal{O}$
for $i \leftarrow n - 1$ **downto** 0**:**
  $T \leftarrow 2T$
  if $a_i = 1$ **and** $b_i = 1$**:**
    $T \leftarrow T + S$
  else if $a_i = 1$**:**
    $T \leftarrow T + P$
  else if $b_i = 1$**:**
    $T \leftarrow T + Q$
return T

- Example: $a = 21 = (10101)_2$
  and $b = 30 = (11110)_2$

$$T = ((((P + Q) \cdot 2 + Q) \cdot 2 + P + Q) \cdot 2 + Q) \cdot 2 + P = 21P + 30Q$$

- Complexity:
  - n doublings, and
  - $3n/4$ additions on average

Multi-exponentiation technique

```
function double-scalar-mult(a, P, b, Q):
    S ← P + Q
    T ← O
    for i ← n − 1 downto 0:
        T ← 2T
        if a_i = 1 and b_i = 1:
            T ← T + S
        else if a_i = 1:
            T ← T + P
        else if b_i = 1:
            T ← T + Q
    return T
```

- Example: $a = 21 = (10101)_2$
  and $b = 30 = (11110)_2$

$$T = ((((P+Q)\cdot 2 + Q)\cdot 2 + P + Q)\cdot 2 + Q)\cdot 2 + P = 21P + 30Q$$

- Complexity:
  - n doublings, and
  - $3n/4$ additions on average
- With signed digits:
  - joint sparse form (JSF): $n/2$ additions
  - interleaved w-NAF: $2n/(w+1)$ additions

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:
  - take an ordinary elliptic curve with a known efficiently computable endomorphism ψ of small norm

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:
  - take an ordinary elliptic curve with a known efficiently computable endomorphism ψ of small norm
  - the characteristic polynomial of ψ is of the form $\chi_\psi(T) = T^2 - t_\psi T + n_\psi$

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:
  - take an ordinary elliptic curve with a known efficiently computable endomorphism ψ of small norm
  - the characteristic polynomial of ψ is of the form $\chi_\psi(T) = T^2 - t_\psi T + n_\psi$
  - there exists a root $\lambda \in \mathbb{Z}/\ell\mathbb{Z}$ of $\chi_\psi(T)$ mod ℓ such that

$$\psi(P) = \lambda P, \text{ for any } P \in \mathbb{G}$$

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:
  - take an ordinary elliptic curve with a known efficiently computable endomorphism ψ of small norm
  - the characteristic polynomial of ψ is of the form $\chi_\psi(T) = T^2 - t_\psi T + n_\psi$
  - there exists a root $\lambda \in \mathbb{Z}/\ell\mathbb{Z}$ of $\chi_\psi(T) \bmod \ell$ such that

$$\psi(P) = \lambda P, \text{ for any } P \in \mathbb{G}$$

$\Rightarrow$ λ-adic decomposition of scalar k as $k \equiv k_0 + \lambda k_1 \pmod{\ell}$ so that

$$kP = k_0 P + k_1 \psi(P)$$

$\Rightarrow$ compute $k_0 P + k_1 \psi(P)$ via multi-exponentiation

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:
  - take an ordinary elliptic curve with a known efficiently computable endomorphism ψ of small norm
  - the characteristic polynomial of ψ is of the form $\chi_\psi(T) = T^2 - t_\psi T + n_\psi$
  - there exists a root $\lambda \in \mathbb{Z}/\ell\mathbb{Z}$ of $\chi_\psi(T) \bmod \ell$ such that

$$\psi(P) = \lambda P, \text{ for any } P \in \mathbb{G}$$

  $\Rightarrow$ λ-adic decomposition of scalar k as $k \equiv k_0 + \lambda k_1 \pmod{\ell}$ so that

$$kP = k_0 P + k_1 \psi(P)$$

  $\Rightarrow$ compute $k_0 P + k_1 \psi(P)$ via multi-exponentiation

- Example:

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:
  - take an ordinary elliptic curve with a known efficiently computable endomorphism ψ of small norm
  - the characteristic polynomial of ψ is of the form $\chi_\psi(T) = T^2 - t_\psi T + n_\psi$
  - there exists a root $\lambda \in \mathbb{Z}/\ell\mathbb{Z}$ of $\chi_\psi(T)$ mod ℓ such that

$$\psi(P) = \lambda P, \text{ for any } P \in \mathbb{G}$$

  $\Rightarrow$ λ-adic decomposition of scalar k as $k \equiv k_0 + \lambda k_1 \pmod{\ell}$ so that

$$kP = k_0 P + k_1 \psi(P)$$

  $\Rightarrow$ compute $k_0 P + k_1 \psi(P)$ via multi-exponentiation

- Example:
  - let $p \equiv 1 \pmod 4$ and $E/\mathbb{F}_p : y^2 = x^3 + Ax$

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:
  - take an ordinary elliptic curve with a known efficiently computable endomorphism ψ of small norm
  - the characteristic polynomial of ψ is of the form $\chi_\psi(T) = T^2 - t_\psi T + n_\psi$
  - there exists a root $\lambda \in \mathbb{Z}/\ell\mathbb{Z}$ of $\chi_\psi(T) \bmod \ell$ such that

$$\psi(P) = \lambda P, \text{ for any } P \in \mathbb{G}$$

$\Rightarrow$ λ-adic decomposition of scalar k as $k \equiv k_0 + \lambda k_1 \pmod{\ell}$ so that

$$kP = k_0 P + k_1 \psi(P)$$

$\Rightarrow$ compute $k_0 P + k_1 \psi(P)$ via multi-exponentiation

- Example:
  - let $p \equiv 1 \pmod 4$ and $E/\mathbb{F}_p : y^2 = x^3 + Ax$
  - let $\xi \in \mathbb{F}_p$ a primitive 4-th root of unity (i.e., $\xi^2 = -1$ and $\xi^4 = 1$)

GLV curves

- Proposed by Gallant, Lambert, and Vanstone in 2000:
  - take an ordinary elliptic curve with a known efficiently computable endomorphism ψ of small norm
  - the characteristic polynomial of ψ is of the form $\chi_\psi(T) = T^2 - t_\psi T + n_\psi$
  - there exists a root $\lambda \in \mathbb{Z}/\ell\mathbb{Z}$ of $\chi_\psi(T) \bmod \ell$ such that

$$\psi(P) = \lambda P, \text{ for any } P \in \mathbb{G}$$

$\Rightarrow$ λ-adic decomposition of scalar k as $k \equiv k_0 + \lambda k_1 \pmod{\ell}$ so that

$$kP = k_0 P + k_1 \psi(P)$$

$\Rightarrow$ compute $k_0 P + k_1 \psi(P)$ via multi-exponentiation

- Example:
  - let $p \equiv 1 \pmod 4$ and $E/\mathbb{F}_p : y^2 = x^3 + Ax$
  - let $\xi \in \mathbb{F}_p$ a primitive 4-th root of unity (i.e., $\xi^2 = -1$ and $\xi^4 = 1$)
  - then $\psi : (x, y) \mapsto (-x, \xi y)$ is an endomorphism of E and, since

$$\psi^2(x, y) = (x, -y) = -(x, y),$$

its characteristic polynomial is $\chi_\psi(T) = T^2 + 1$ and $\lambda = \pm\sqrt{-1} \bmod \ell$

GLV curves

- Computation of k_0 and k_1:

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\rightarrow$ precompute short basis (EEA)

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\rightarrow$ precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ → precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) && \pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda && \pmod{\ell} \end{aligned}$$

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\rightarrow$ precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) && \pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda && \pmod{\ell} \end{aligned}$$

  - take $k_0 = (k - \tilde{k}_0) \bmod \ell$ and $k_1 = -\tilde{k}_1 \bmod \ell$

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\rightarrow$ precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) && \pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda && \pmod{\ell} \end{aligned}$$

  - take $k_0 = (k - \tilde{k}_0) \bmod \ell$ and $k_1 = -\tilde{k}_1 \bmod \ell$

  $\Rightarrow$ k_0 and k_1 of size $\approx n/2$ bits

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\rightarrow$ precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) &\pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda &\pmod{\ell} \end{aligned}$$

  - take $k_0 = (k - \tilde{k}_0) \bmod \ell$ and $k_1 = -\tilde{k}_1 \bmod \ell$

  $\Rightarrow$ k_0 and k_1 of size $\approx n/2$ bits

- Previous example with $p = 953$ and $E/\mathbb{F}_p : y^2 = x^3 + 5x$:

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\rightarrow$ precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) && \pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda && \pmod{\ell} \end{aligned}$$

  - take $k_0 = (k - \tilde{k}_0) \bmod \ell$ and $k_1 = -\tilde{k}_1 \bmod \ell$

  $\Rightarrow$ k_0 and k_1 of size $\approx n/2$ bits

- Previous example with $p = 953$ and $E/\mathbb{F}_p : y^2 = x^3 + 5x$:
  - as $\#E(\mathbb{F}_p) = 2 \cdot 449$, we take $\ell = 449$

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\rightarrow$ precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) && \pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda && \pmod{\ell} \end{aligned}$$

  - take $k_0 = (k - \tilde{k}_0) \bmod \ell$ and $k_1 = -\tilde{k}_1 \bmod \ell$

  $\Rightarrow$ k_0 and k_1 of size $\approx n/2$ bits

- Previous example with $p = 953$ and $E/\mathbb{F}_p : y^2 = x^3 + 5x$:
  - as $\#E(\mathbb{F}_p) = 2 \cdot 449$, we take $\ell = 449$
  - let $\xi = 442$ and check that $\xi^2 \equiv -1 \pmod{p}$

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\rightarrow$ precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) \pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda \pmod{\ell} \end{aligned}$$

  - take $k_0 = (k - \tilde{k}_0) \bmod \ell$ and $k_1 = -\tilde{k}_1 \bmod \ell$

  $\Rightarrow$ k_0 and k_1 of size $\approx n/2$ bits

- Previous example with $p = 953$ and $E/\mathbb{F}_p : y^2 = x^3 + 5x$:
  - as $\#E(\mathbb{F}_p) = 2 \cdot 449$, we take $\ell = 449$
  - let $\xi = 442$ and check that $\xi^2 \equiv -1 \pmod{p}$
  - $\psi : (x, y) \mapsto (-x, \xi y)$: we have $\psi(P) = \lambda P$ for all $P \in \mathbb{G}$, with $\lambda = 382$

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ $\to$ precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) && \pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda && \pmod{\ell} \end{aligned}$$

  - take $k_0 = (k - \tilde{k}_0) \bmod \ell$ and $k_1 = -\tilde{k}_1 \bmod \ell$

  $\Rightarrow$ k_0 and k_1 of size $\approx n/2$ bits

- Previous example with $p = 953$ and $E/\mathbb{F}_p : y^2 = x^3 + 5x$:
  - as $\#E(\mathbb{F}_p) = 2 \cdot 449$, we take $\ell = 449$
  - let $\xi = 442$ and check that $\xi^2 \equiv -1 \pmod{p}$
  - $\psi : (x, y) \mapsto (-x, \xi y)$: we have $\psi(P) = \lambda P$ for all $P \in \mathbb{G}$, with $\lambda = 382$
  - scalar $k = 431$ can be rewritten as $k \equiv 2 + 7\lambda \pmod{\ell}$, whence

$$kP = 2P + 7\psi(P)$$

GLV curves

- Computation of k_0 and k_1:
  - pairs $(a, b) \in \mathbb{Z}^2$ such that $a + b\lambda \equiv 0 \pmod{\ell}$ form a 2-dimensional lattice Λ
  - Λ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ → precompute short basis (EEA)
  - given k, find lattice point $(\tilde{k}_0, \tilde{k}_1) \in \Lambda$ closest to $(k, 0)$

$$\begin{aligned} k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) && \pmod{\ell} \\ &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda && \pmod{\ell} \end{aligned}$$

  - take $k_0 = (k - \tilde{k}_0) \bmod \ell$ and $k_1 = -\tilde{k}_1 \bmod \ell$

  ⇒ k_0 and k_1 of size $\approx n/2$ bits

- Previous example with $p = 953$ and $E/\mathbb{F}_p : y^2 = x^3 + 5x$:
  - as $\#E(\mathbb{F}_p) = 2 \cdot 449$, we take $\ell = 449$
  - let $\xi = 442$ and check that $\xi^2 \equiv -1 \pmod{p}$
  - $\psi : (x, y) \mapsto (-x, \xi y)$: we have $\psi(P) = \lambda P$ for all $P \in \mathbb{G}$, with $\lambda = 382$
  - scalar $k = 431$ can be rewritten as $k \equiv 2 + 7\lambda \pmod{\ell}$, whence

$$kP = 2P + 7\psi(P)$$

- Popular constructions exploiting endomorphism ring:
  - GLS curves (Galbraith, Lin, and Scott, 2008): large class of GLV-compatible curves
  - Koblitz curves: binary curves, with Frobenius map $\psi : (x, y) \mapsto (x^2, y^2)$

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

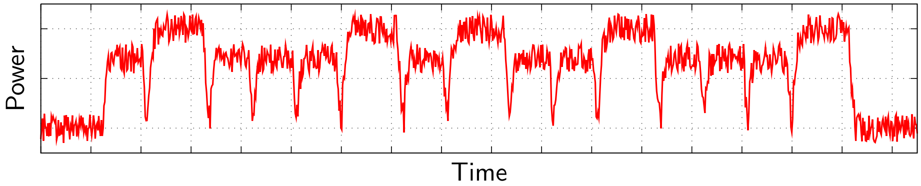
- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n - 1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - power analysis will leak bits of k

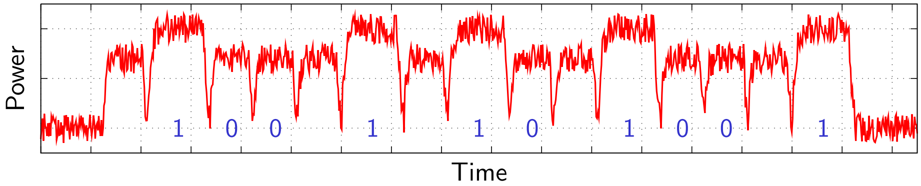


Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P):
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0:
    $T \leftarrow 2T$
    if $k_i = 1$:
      $T \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - power analysis will leak bits of k

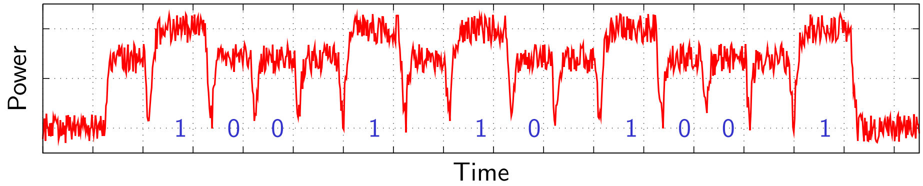


Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
    else:
      $Z \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - power analysis will leak bits of k



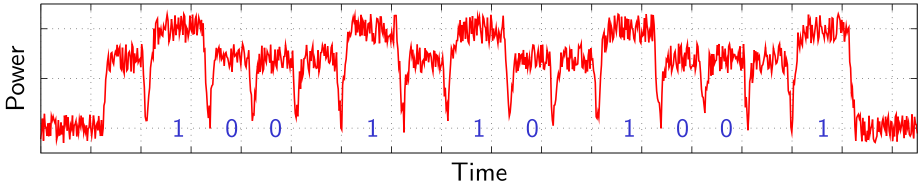
- Use double-and-add-always algorithm?

Security issues

- Back to the double-and-add algorithm:

```
function scalar-mult(k, P):
    T ← O
    for i ← n − 1 downto 0:
        T ← 2T
        if k_i = 1:
            T ← T + P
        else:
            Z ← T + P
    return T
```

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - power analysis will leak bits of k



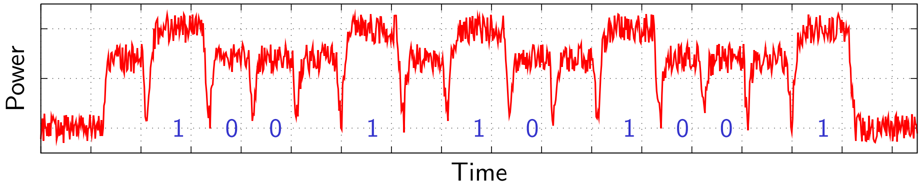
- Use double-and-add-always algorithm?
  - the result of the point addition is used if and only if $k_i = 1$

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
    else:
      $Z \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - power analysis will leak bits of k



- Use double-and-add-always algorithm?
  - the result of the point addition is used if and only if $k_i = 1$

  $\Rightarrow$ vulnerable to fault attacks

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n-1$ **downto** 0**:**
if $k_i = 1$**:**
$T_0 \leftarrow T_0 + T_1$
$T_1 \leftarrow 2T_1$
else:
$T_1 \leftarrow T_0 + T_1$
$T_0 \leftarrow 2T_0$
return T_0

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n - 1$ **downto** 0**:**
  if $k_i = 1$**:**
    $T_0 \leftarrow T_0 + T_1$
    $T_1 \leftarrow 2T_1$
  else:
    $T_1 \leftarrow T_0 + T_1$
    $T_0 \leftarrow 2T_0$
return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    for $i \leftarrow n - 1$ **downto** 0**:**
        if $k_i = 1$**:**
            $T_0 \leftarrow T_0 + T_1$
            $T_1 \leftarrow 2T_1$
        else:
            $T_1 \leftarrow T_0 + T_1$
            $T_0 \leftarrow 2T_0$
    return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10011)_2$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10011)_2$

$$T_0 = \qquad = \mathcal{O}$$
$$T_1 = P \qquad = P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$

- Example: $k = 19 = (\underline{1}0011)_2$

$$T_0 = \quad = \mathcal{O}$$
$$T_1 = P \quad = P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n - 1$ **downto** 0**:**
if $k_i = 1$**:**
$T_0 \leftarrow T_0 + T_1$
$T_1 \leftarrow 2T_1$
else:
$T_1 \leftarrow T_0 + T_1$
$T_0 \leftarrow 2T_0$
return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$

- Example: $k = 19 = (\underline{1}0011)_2$

$$T_0 = P \quad = P$$
$$T_1 = P \quad = P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    for $i \leftarrow n - 1$ **downto** 0**:**
        if $k_i = 1$**:**
            $T_0 \leftarrow T_0 + T_1$
            $T_1 \leftarrow 2T_1$
        else:
            $T_1 \leftarrow T_0 + T_1$
            $T_0 \leftarrow 2T_0$
    return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$

- Example: $k = 19 = (\underline{1}0011)_2$

$$T_0 = P \qquad = P$$
$$T_1 = P \cdot 2 \qquad = 2P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1\underline{0}011)_2$

$$T_0 = P \quad = P$$
$$T_1 = P \cdot 2 \quad = 2P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1\underline{0}011)_2$

$$T_0 = P \quad = P$$
$$T_1 = P \cdot 2 + P \quad = 3P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$

- Example: $k = 19 = (1\underline{0}011)_2$

$$T_0 = P \cdot 2 = 2P$$
$$T_1 = P \cdot 2 + P = 3P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10\underline{0}11)_2$

$$T_0 = P \cdot 2 = 2P$$
$$T_1 = P \cdot 2 + P = 3P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T0 ← O
    T1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T0 ← T0 + T1
            T1 ← 2T1
        else:
            T1 ← T0 + T1
            T0 ← 2T0
    return T0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10\underline{0}11)_2$

$$\begin{aligned} T_0 &= P \cdot 2 &&= 2P \\ T_1 &= P \cdot 2 + P + 2P &&= 5P \end{aligned}$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n-1$ **downto** 0**:**
if $k_i = 1$**:**
$T_0 \leftarrow T_0 + T_1$
$T_1 \leftarrow 2T_1$
else:
$T_1 \leftarrow T_0 + T_1$
$T_0 \leftarrow 2T_0$
return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10\underline{0}11)_2$

$$T_0 = P \cdot 2^2 = 4P$$
$$T_1 = P \cdot 2 + P + 2P = 5P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2 T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2 T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10011)_2$

$$T_0 = P \cdot 2^2 = 4P$$
$$T_1 = P \cdot 2 + P + 2P = 5P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (100\underline{1}1)_2$

$$T_0 = P \cdot 2^2 + 5P = 9P$$
$$T_1 = P \cdot 2 + P + 2P = 5P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    for $i \leftarrow n-1$ **downto** 0**:**
        if $k_i = 1$**:**
            $T_0 \leftarrow T_0 + T_1$
            $T_1 \leftarrow 2T_1$
        else:
            $T_1 \leftarrow T_0 + T_1$
            $T_0 \leftarrow 2T_0$
    return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10\underline{0}11)_2$

$$T_0 = P \cdot 2^2 + 5P = 9P$$
$$T_1 = (P \cdot 2 + P + 2P) \cdot 2 = 10P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n-1$ **downto** 0**:**
  if $k_i = 1$**:**
    $T_0 \leftarrow T_0 + T_1$
    $T_1 \leftarrow 2T_1$
  else:
    $T_1 \leftarrow T_0 + T_1$
    $T_0 \leftarrow 2T_0$
return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1001\underline{1})_2$

$$T_0 = P \cdot 2^2 + 5P = 9P$$

$$T_1 = (P \cdot 2 + P + 2P) \cdot 2 = 10P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    for $i \leftarrow n-1$ **downto** 0**:**
        if $k_i = 1$**:**
            $T_0 \leftarrow T_0 + T_1$
            $T_1 \leftarrow 2T_1$
        else:
            $T_1 \leftarrow T_0 + T_1$
            $T_0 \leftarrow 2T_0$
    return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1001\underline{1})_2$

$$T_0 = P \cdot 2^2 + 5P + 10P = 19P$$
$$T_1 = (P \cdot 2 + P + 2P) \cdot 2 = 10P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1001\underline{1})_2$

$$T_0 = P \cdot 2^2 + 5P + 10P = 19P$$
$$T_1 = (P \cdot 2 + P + 2P) \cdot 2^2 = 20P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10011)_2$

$$T_0 = P \cdot 2^2 + 5P + 10P = 19P$$
$$T_1 = (P \cdot 2 + P + 2P) \cdot 2^2 = 20P$$

More security issues

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n - 1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

More security issues

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- The conditional branches depend on the value of secret bit k_i

More security issues

function scalar-mult(k, P)**:**

$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n-1$ **downto** 0**:**
  if $k_i = 1$**:**
    $T_0 \leftarrow T_0 + T_1$
    $T_1 \leftarrow 2T_1$
  else:
    $T_1 \leftarrow T_0 + T_1$
    $T_0 \leftarrow 2T_0$
return T_0

- The conditional branches depend on the value of secret bit k_i
  $\Rightarrow$ might be vulnerable to branch prediction attacks

More security issues

function scalar-mult(k, P)**:**

$T_0 \leftarrow \mathcal{O}$

$T_1 \leftarrow P$

for $i \leftarrow n - 1$ **downto** 0**:**

$T_{1-k_i} \leftarrow T_0 + T_1$

$T_{k_i} \leftarrow 2T_{k_i}$

return T_0

- The conditional branches depend on the value of secret bit k_i
  $\Rightarrow$ might be vulnerable to branch prediction attacks

- Compute indices for T_0 and T_1 from k_i?

More security issues

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T_{1-k_i} \leftarrow T_0 + T_1$
    $T_{k_i} \leftarrow 2T_{k_i}$
  return T_0

- The conditional branches depend on the value of secret bit k_i
  $\Rightarrow$ might be vulnerable to branch prediction attacks
- Compute indices for T_0 and T_1 from k_i?
  - memory accesses to T_0 or T_1 depend on secret bit k_i

More security issues

function scalar-mult(k, P)**:**

$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n-1$ **downto** 0**:**
$T_{1-k_i} \leftarrow T_0 + T_1$
$T_{k_i} \leftarrow 2T_{k_i}$
return T_0

- The conditional branches depend on the value of secret bit k_i
  $\Rightarrow$ might be vulnerable to branch prediction attacks
- Compute indices for T_0 and T_1 from k_i?
  - memory accesses to T_0 or T_1 depend on secret bit k_i

  $\Rightarrow$ might be vulnerable to cache attacks

More security issues

function scalar-mult(k, P):

$$
\begin{aligned}
& T_0 \leftarrow \mathcal{O} \\
& T_1 \leftarrow P \\
& \textbf{for } i \leftarrow n-1 \textbf{ downto } 0\textbf{:} \\
& \quad M \leftarrow (k_i \dots k_i)_2 \\
& \quad R \leftarrow T_0 + T_1 \\
& \quad S \leftarrow 2((\overline{M} \& T_0) \mid (M \& T_1)) \\
& \quad T_0 \leftarrow (\overline{M} \& S) \mid (M \& R) \\
& \quad T_1 \leftarrow (\overline{M} \& R) \mid (M \& S) \\
& \textbf{return } T_0
\end{aligned}
$$

- The conditional branches depend on the value of secret bit k_i
  $\Rightarrow$ might be vulnerable to branch prediction attacks
- Compute indices for T_0 and T_1 from k_i?
  - memory accesses to T_0 or T_1 depend on secret bit k_i

  $\Rightarrow$ might be vulnerable to cache attacks
- Use bit masking to avoid secret-dependent memory access patterns

Outline

I. Scalar multiplication

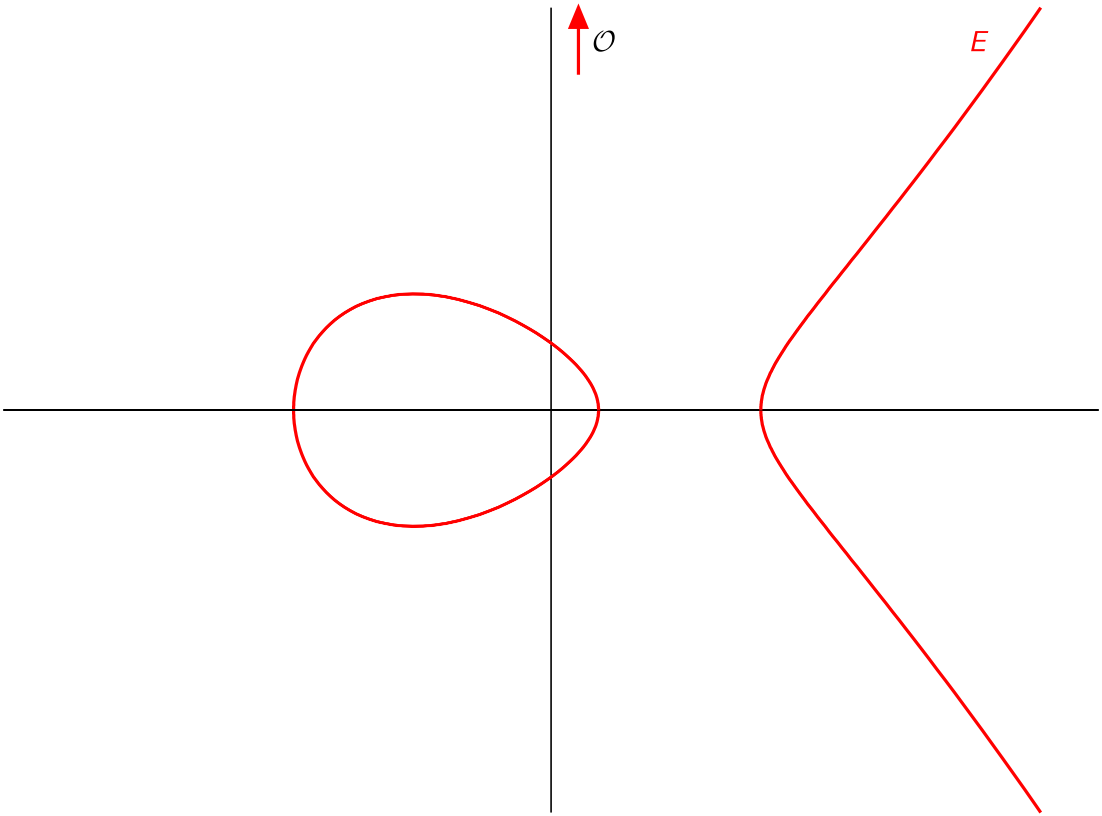
II. Elliptic curve arithmetic

III. Finite field arithmetic

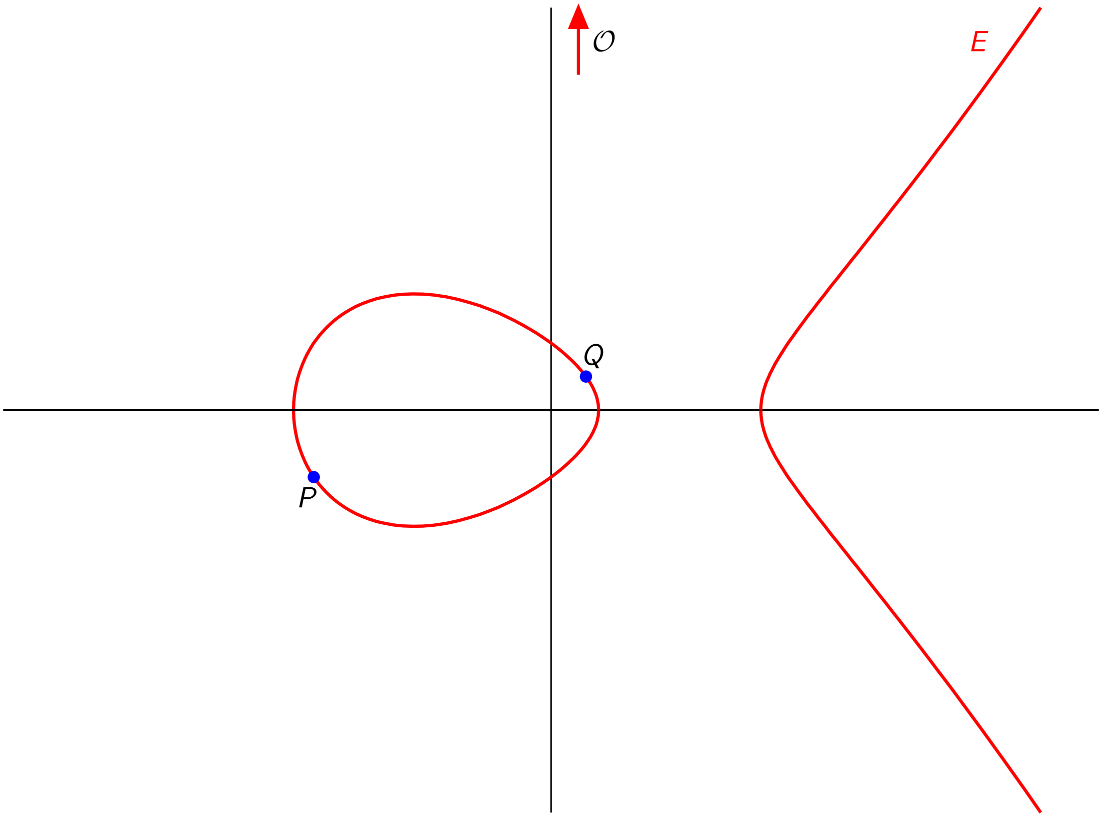
IV. Software considerations

V. Notions of hardware design

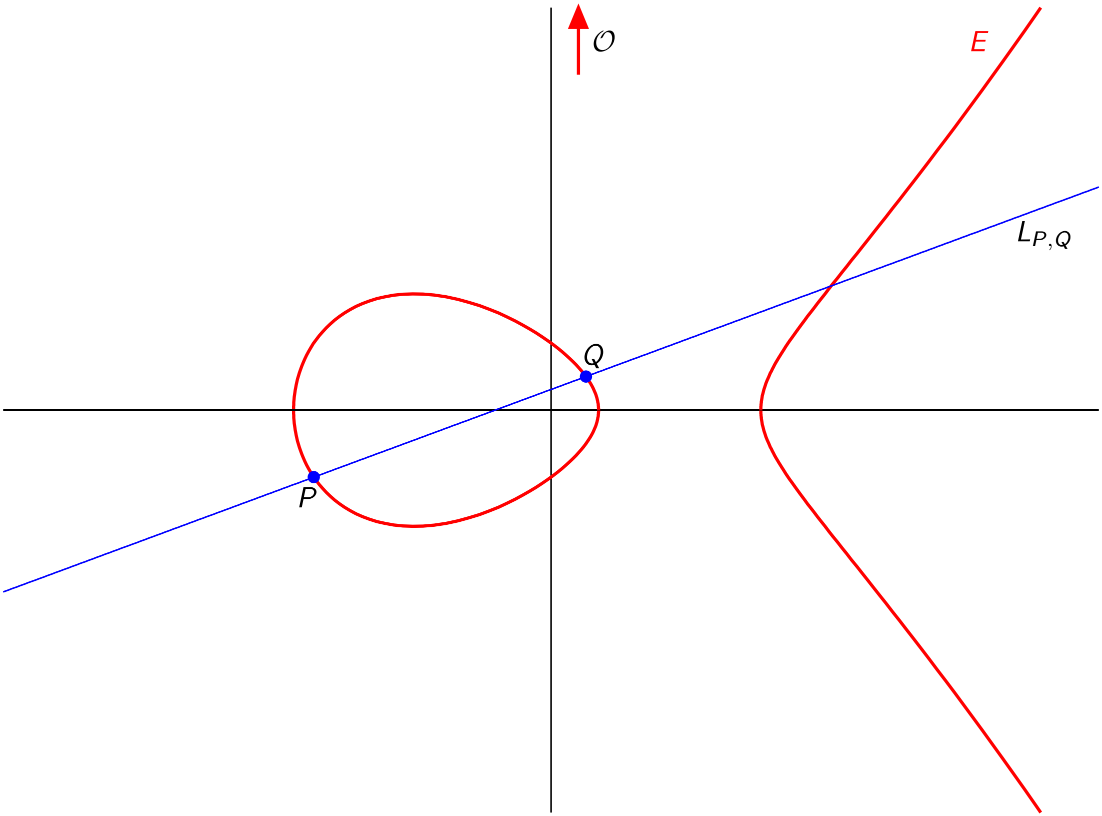
Addition and doubling



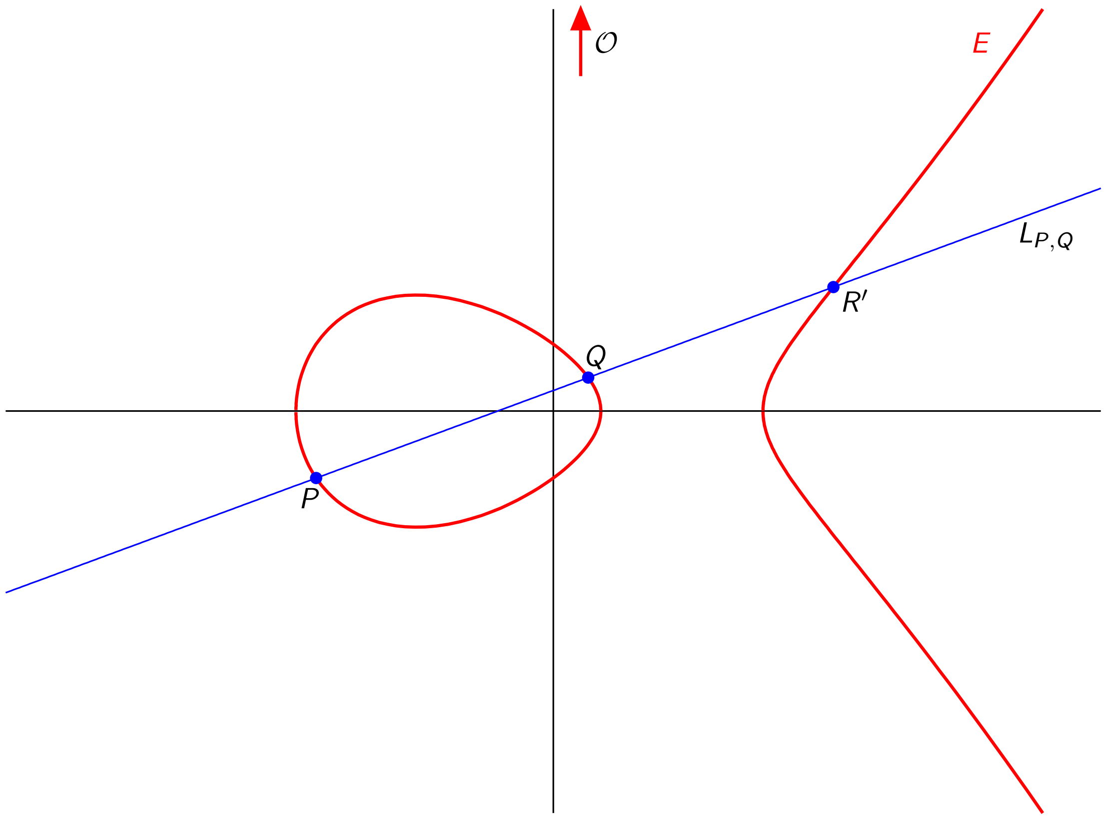
Addition and doubling



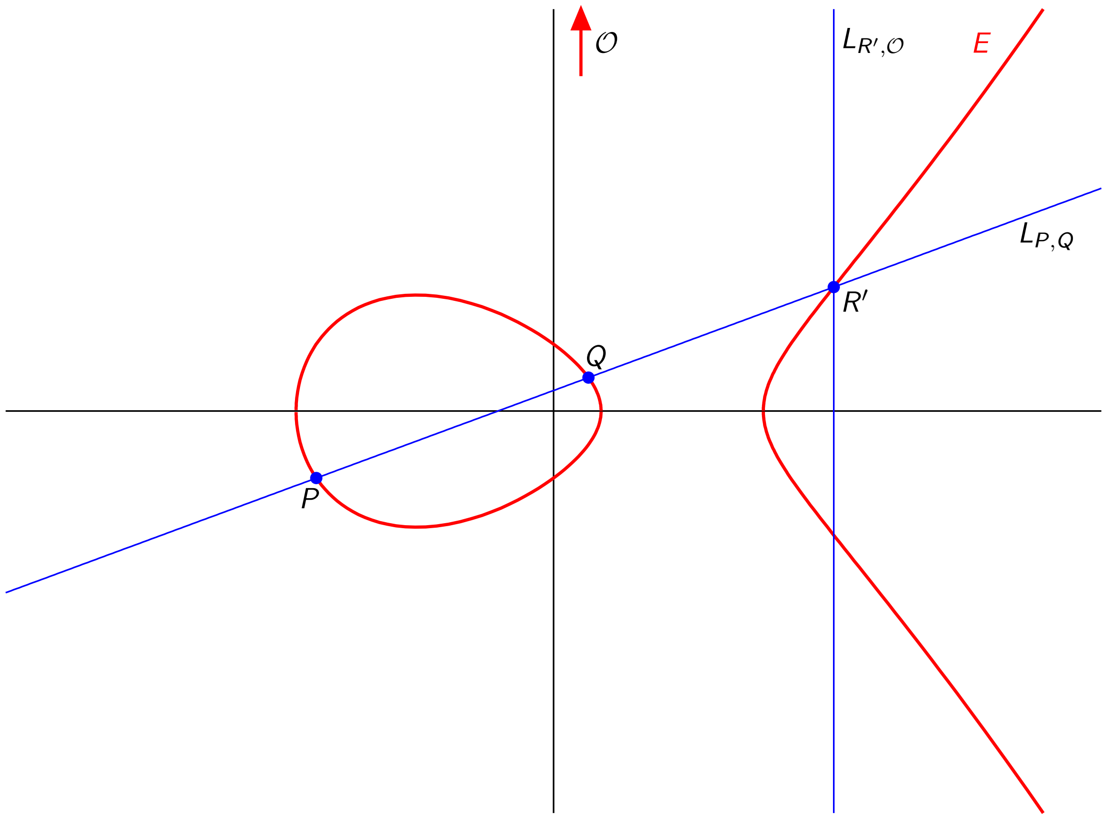
Addition and doubling



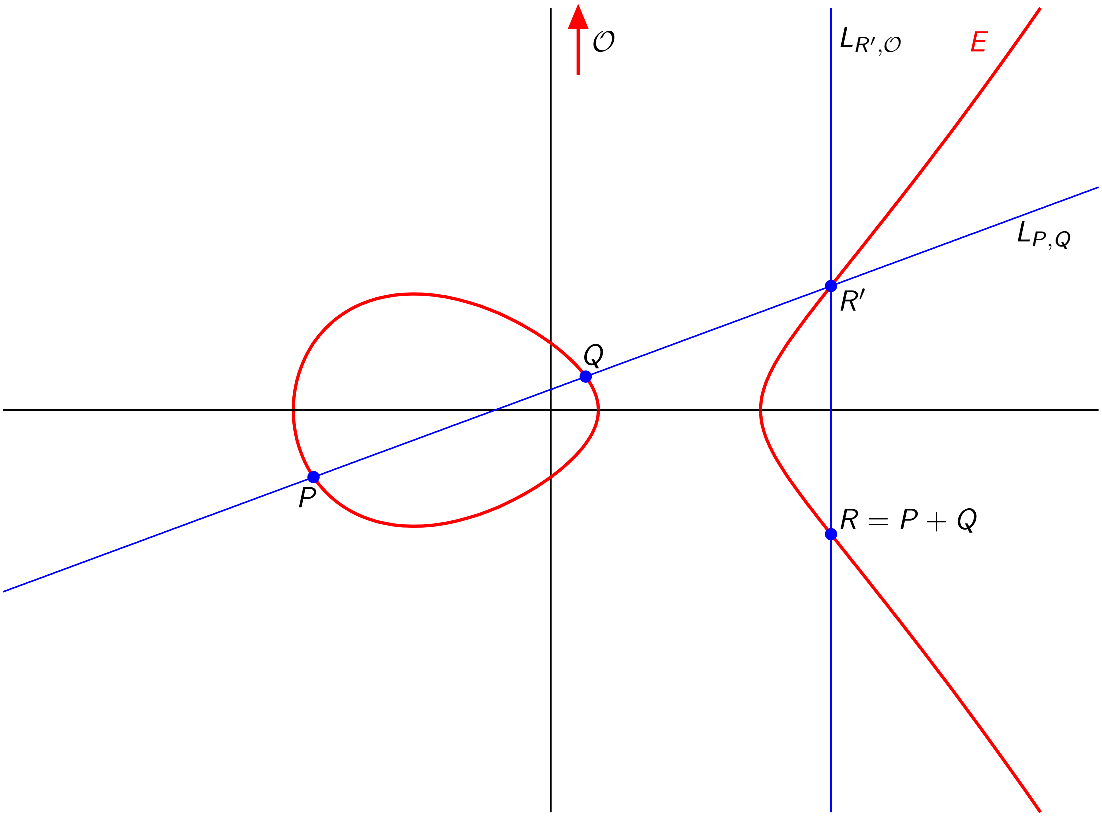
Addition and doubling



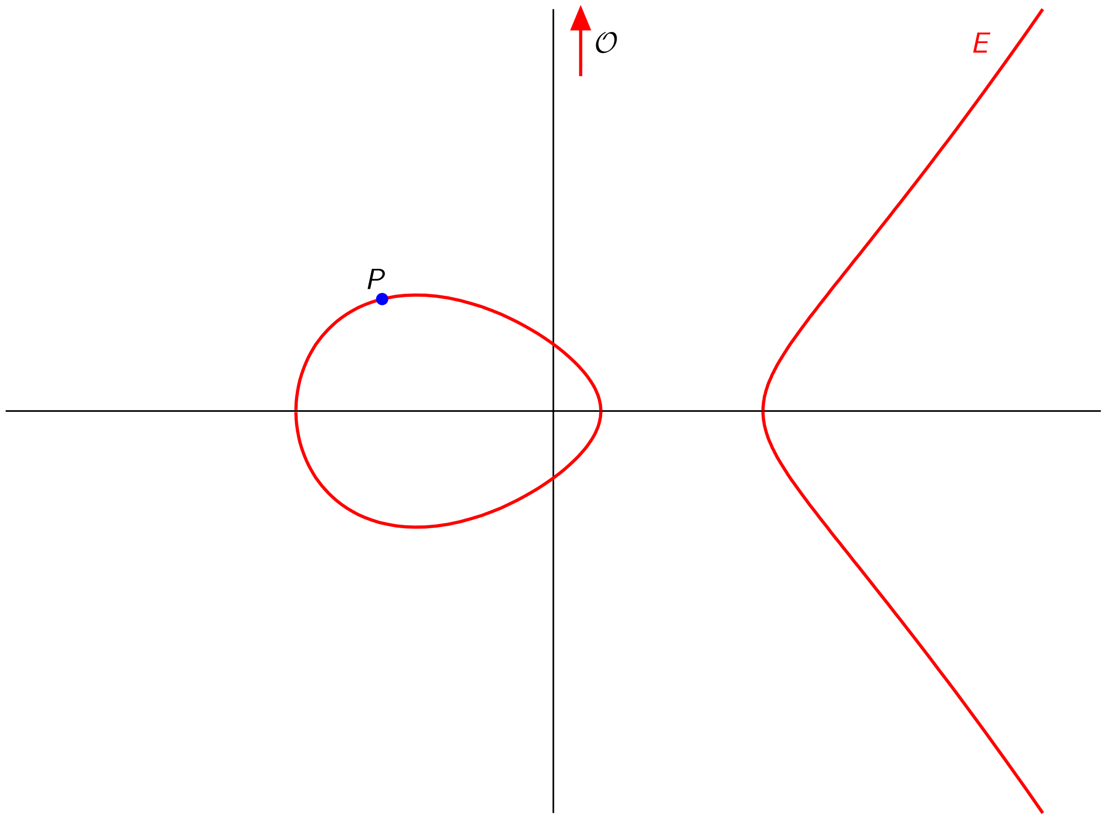
Addition and doubling



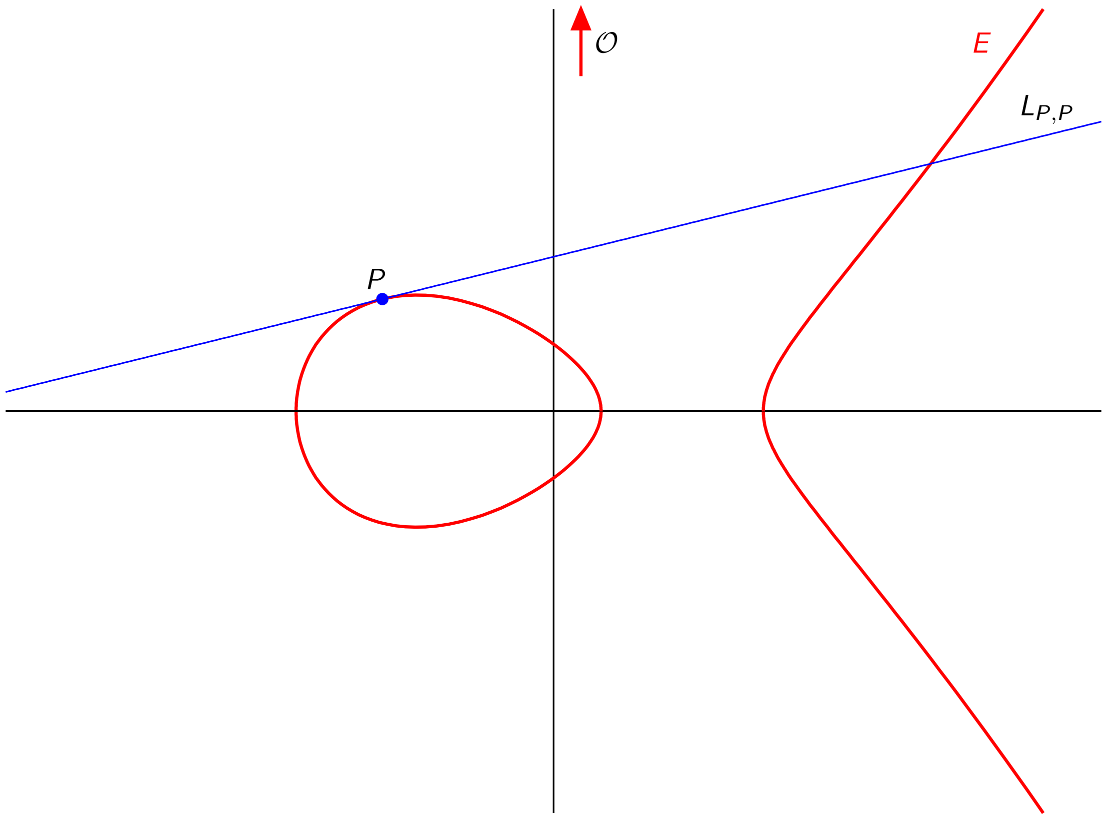
Addition and doubling



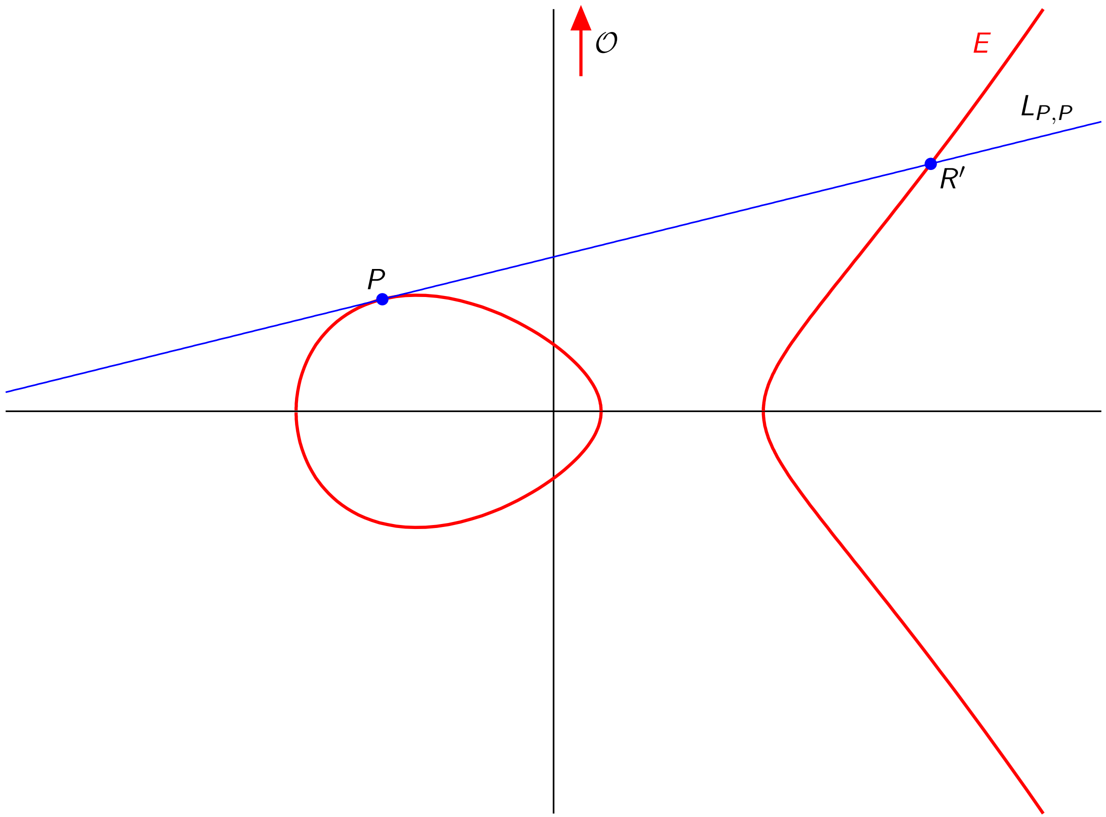
Addition and doubling



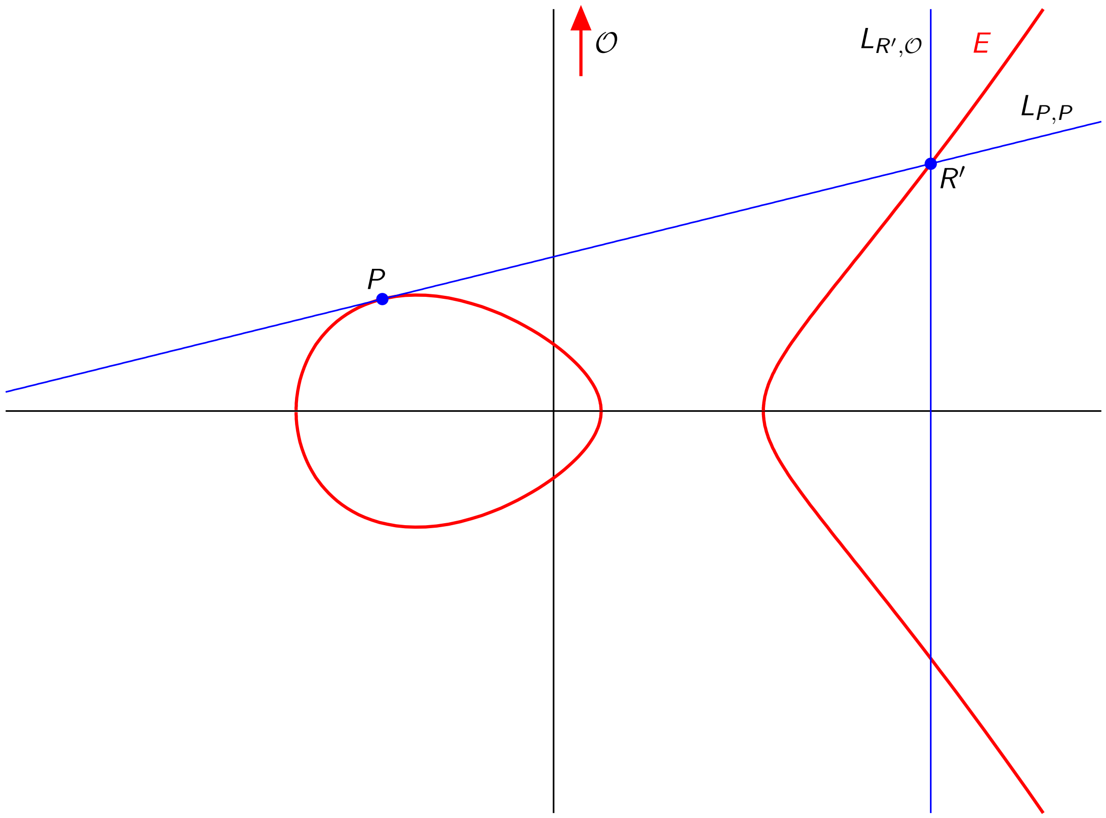
Addition and doubling



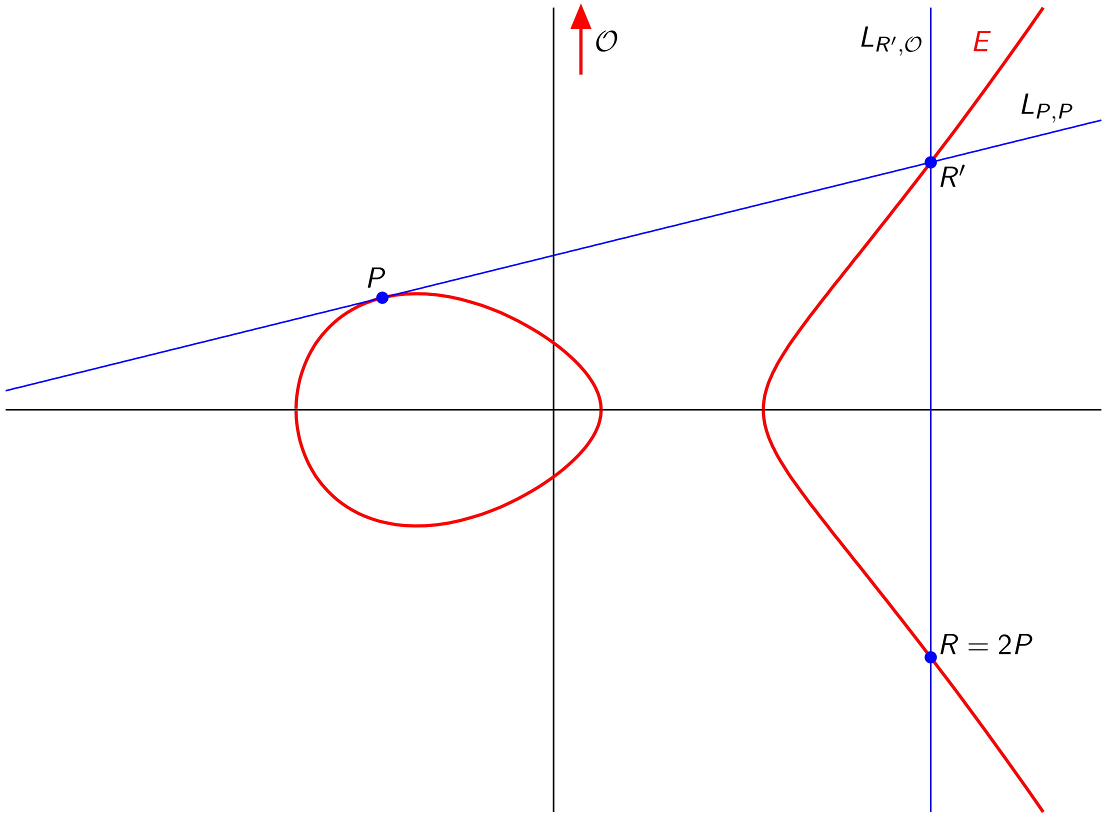
Addition and doubling



Addition and doubling



Addition and doubling



Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ (affine coordinates)

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q) \backslash \{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{, or} \\ -\dfrac{(\partial f/\partial x)(x_P, y_P)}{(\partial f/\partial y)(x_P, y_P)} = \dfrac{3x_P^2 + a}{2y_P} & \text{if } P = Q \end{cases}$$

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q) \backslash \{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q, \text{ or} \\ -\dfrac{(\partial f/\partial x)(x_P, y_P)}{(\partial f/\partial y)(x_P, y_P)} = \dfrac{3x_P^2 + a}{2y_P} & \text{if } P = Q \end{cases}$$

- Cost (number of inversions, multiplications and squares in $\mathbb{F}_q$):

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q) \backslash \{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{, or} \\ -\dfrac{(\partial f/\partial x)(x_P, y_P)}{(\partial f/\partial y)(x_P, y_P)} = \dfrac{3x_P^2 + a}{2y_P} & \text{if } P = Q \end{cases}$$

- Cost (number of inversions, multiplications and squares in $\mathbb{F}_q$):
  - addition: 1I + 2M + 1S

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q) \backslash \{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{, or} \\ -\dfrac{(\partial f/\partial x)(x_P, y_P)}{(\partial f/\partial y)(x_P, y_P)} = \dfrac{3x_P^2 + a}{2y_P} & \text{if } P = Q \end{cases}$$

- Cost (number of inversions, multiplications and squares in $\mathbb{F}_q$):
  - addition: 1I + 2M + 1S
  - doubling: 1I + 2M + 2S

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S
- Jacobian coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$

$$E/\mathbb{F}_q : Y^2 = X^3 + AXZ^4 + BZ^6$$

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S
- Jacobian coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$

$$E/\mathbb{F}_q : Y^2 = X^3 + AXZ^4 + BZ^6$$

  - addition: 12M + 4S
  - doubling: 4M + 6S

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S
- Jacobian coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$

$$E/\mathbb{F}_q : Y^2 = X^3 + AXZ^4 + BZ^6$$

  - addition: 12M + 4S
  - doubling: 4M + 6S
- And many others: modified jacobian coordinates, López–Dahab (over $\mathbb{F}_{2^n}$), etc.

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S
- Jacobian coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$

$$E/\mathbb{F}_q : Y^2 = X^3 + AXZ^4 + BZ^6$$

  - addition: 12M + 4S
  - doubling: 4M + 6S
- And many others: modified jacobian coordinates, López–Dahab (over $\mathbb{F}_{2^n}$), etc.
- Explicit-Formula Database (by Bernstein and Lange):

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

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form
  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q) \backslash \{\mathcal{O}\}$, with $P \neq \pm Q$

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)\backslash\{\mathcal{O}\}$, with $P \neq \pm Q$
  - then, writing $R = P + Q = (x_R, y_R)$ and $S = P - Q = (x_S, y_S)$, we have

$$x_R x_S (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q) \backslash \{\mathcal{O}\}$, with $P \neq \pm Q$
  - then, writing $R = P + Q = (x_R, y_R)$ and $S = P - Q = (x_S, y_S)$, we have

    $$x_R x_S (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

  - the x-coord. of $R = P + Q$ depends only on the x-coord. of P, Q, and $P - Q$
    $\Rightarrow$ x-only differential addition

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

$$C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x, \text{ with parameters } A, B \in \mathbb{F}_q \text{ and } \mathrm{char}(\mathbb{F}_q) \neq 2$$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)\backslash\{\mathcal{O}\}$, with $P \neq \pm Q$
  - then, writing $R = P + Q = (x_R, y_R)$ and $S = P - Q = (x_S, y_S)$, we have

$$x_R x_S (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

  - the x-coord. of $R = P + Q$ depends only on the x-coord. of P, Q, and $P - Q$ $\Rightarrow$ x-only differential addition
  - similarly, when $P = Q$ and $R = 2P = (x_R, y_R)$, we have

$$4 x_P x_R (x_P^2 + A x_P + 1) = (x_P^2 - 1)^2$$

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

$$C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x, \text{ with parameters } A, B \in \mathbb{F}_q \text{ and } \mathrm{char}(\mathbb{F}_q) \neq 2$$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)\backslash\{\mathcal{O}\}$, with $P \neq \pm Q$
  - then, writing $R = P + Q = (x_R, y_R)$ and $S = P - Q = (x_S, y_S)$, we have

$$x_R x_S (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

  - the x-coord. of $R = P + Q$ depends only on the x-coord. of P, Q, and $P - Q$
    $\Rightarrow$ x-only differential addition
  - similarly, when $P = Q$ and $R = 2P = (x_R, y_R)$, we have

$$4x_P x_R (x_P^2 + Ax_P + 1) = (x_P^2 - 1)^2$$

    $\Rightarrow$ x-only doubling

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q) \setminus \{\mathcal{O}\}$, with $P \neq \pm Q$
  - then, writing $R = P + Q = (x_R, y_R)$ and $S = P - Q = (x_S, y_S)$, we have

    $$x_R x_S (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

  - the x-coord. of $R = P + Q$ depends only on the x-coord. of P, Q, and $P - Q$
    $\Rightarrow$ x-only differential addition
  - similarly, when $P = Q$ and $R = 2P = (x_R, y_R)$, we have

    $$4 x_P x_R (x_P^2 + A x_P + 1) = (x_P^2 - 1)^2$$

    $\Rightarrow$ x-only doubling

- We can drop the y-coordinate altogether in the scalar multiplication

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)\backslash\{\mathcal{O}\}$, with $P \neq \pm Q$
  - then, writing $R = P + Q = (x_R, y_R)$ and $S = P - Q = (x_S, y_S)$, we have

    $$x_R x_S (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

  - the x-coord. of $R = P + Q$ depends only on the x-coord. of P, Q, and $P - Q$ $\Rightarrow$ x-only differential addition
  - similarly, when $P = Q$ and $R = 2P = (x_R, y_R)$, we have

    $$4x_P x_R (x_P^2 + Ax_P + 1) = (x_P^2 - 1)^2$$

    $\Rightarrow$ x-only doubling

- We can drop the y-coordinate altogether in the scalar multiplication
  - use projective coordinates: points $(X : Z)$ with $x = X/Z$

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)\backslash\{\mathcal{O}\}$, with $P \neq \pm Q$
  - then, writing $R = P + Q = (x_R, y_R)$ and $S = P - Q = (x_S, y_S)$, we have

    $$x_R x_S (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

  - the x-coord. of $R = P + Q$ depends only on the x-coord. of P, Q, and $P - Q$
    $\Rightarrow$ x-only differential addition
  - similarly, when $P = Q$ and $R = 2P = (x_R, y_R)$, we have

    $$4x_P x_R (x_P^2 + Ax_P + 1) = (x_P^2 - 1)^2$$

    $\Rightarrow$ x-only doubling

- We can drop the y-coordinate altogether in the scalar multiplication
  - use projective coordinates: points $(X : Z)$ with $x = X/Z$
  - cheap differential addition ($4\mathrm{M} + 2\mathrm{S}$) and doubling ($2\mathrm{M} + 2\mathrm{S}$)

Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form

  $$C/\mathbb{F}_q : By^2 = x^3 + Ax^2 + x, \text{ with parameters } A, B \in \mathbb{F}_q \text{ and } \mathrm{char}(\mathbb{F}_q) \neq 2$$

  - all Montgomery curves are elliptic curves
  - not all elliptic curves can be rewritten in Montgomery form

- Addition and doubling formulae
  - let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)\backslash\{\mathcal{O}\}$, with $P \neq \pm Q$
  - then, writing $R = P + Q = (x_R, y_R)$ and $S = P - Q = (x_S, y_S)$, we have

    $$x_R x_S (x_P - x_Q)^2 = (x_P x_Q - 1)^2$$

  - the x-coord. of $R = P + Q$ depends only on the x-coord. of P, Q, and $P - Q$ $\Rightarrow$ x-only differential addition
  - similarly, when $P = Q$ and $R = 2P = (x_R, y_R)$, we have

    $$4x_P x_R (x_P^2 + Ax_P + 1) = (x_P^2 - 1)^2$$

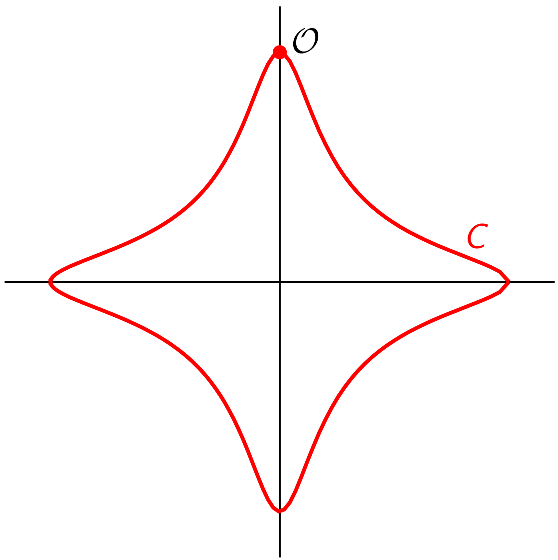
    $\Rightarrow$ x-only doubling

- We can drop the y-coordinate altogether in the scalar multiplication
  - use projective coordinates: points $(X : Z)$ with $x = X/Z$
  - cheap differential addition ($4\mathrm{M} + 2\mathrm{S}$) and doubling ($2\mathrm{M} + 2\mathrm{S}$)
  - compatible with the Montgomery ladder (since $T_1 - T_0 = P$)

Edwards curves

- Proposed by Edwards in 2007, Edwards curves are of the form

  $C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$, with parameter $d \in \mathbb{F}_q$ and $\text{char}(\mathbb{F}_q) \neq 2$

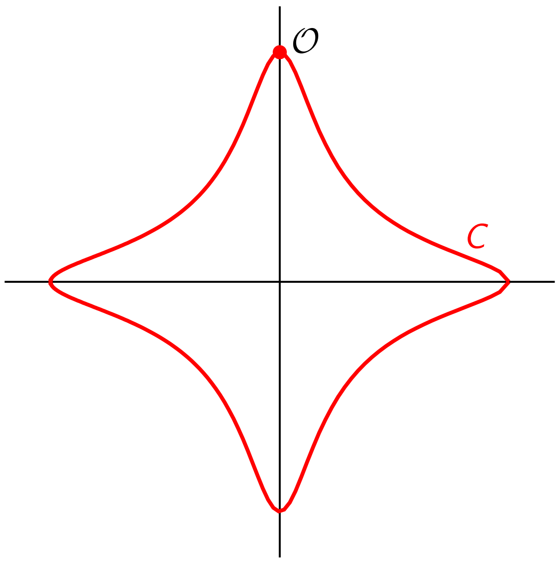


Edwards curves

- Proposed by Edwards in 2007, Edwards curves are of the form

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2, \text{ with parameter } d \in \mathbb{F}_q \text{ and } \mathrm{char}(\mathbb{F}_q) \neq 2$$

  - all Edwards curves are elliptic curves
  - not all elliptic curves can be rewritten in Edwards form



Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)

Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)
  - neutral element: $\mathcal{O} = (0, 1)$

Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)
  - neutral element: $\mathcal{O} = (0, 1)$
  - opposite: for all $P = (x_P, y_P) \in C(\mathbb{F}_q)$, $-P = (-x_P, y_P)$

Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)
  - neutral element: $\mathcal{O} = (0, 1)$
  - opposite: for all $P = (x_P, y_P) \in C(\mathbb{F}_q)$, $-P = (-x_P, y_P)$
  - addition: for all $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)$, then

$$P + Q = \left(\frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q}, \frac{y_P y_Q - x_P x_Q}{1 - d x_P x_Q y_P y_Q} \right)$$

Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)
  - neutral element: $\mathcal{O} = (0, 1)$
  - opposite: for all $P = (x_P, y_P) \in C(\mathbb{F}_q)$, $-P = (-x_P, y_P)$
  - addition: for all $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)$, then

$$P + Q = \left(\frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q}, \frac{y_P y_Q - x_P x_Q}{1 - d x_P x_Q y_P y_Q} \right)$$

  - doubling: same as addition

Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)
  - neutral element: $\mathcal{O} = (0, 1)$
  - opposite: for all $P = (x_P, y_P) \in C(\mathbb{F}_q)$, $-P = (-x_P, y_P)$
  - addition: for all $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)$, then

$$P + Q = \left(\frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q}, \frac{y_P y_Q - x_P x_Q}{1 - d x_P x_Q y_P y_Q} \right)$$

  - doubling: same as addition
- Strongly unified and complete addition law:
  - works for both addition and doubling
  - no exceptional case

Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)
  - neutral element: $\mathcal{O} = (0, 1)$
  - opposite: for all $P = (x_P, y_P) \in C(\mathbb{F}_q)$, $-P = (-x_P, y_P)$
  - addition: for all $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)$, then

$$P + Q = \left(\frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q}, \frac{y_P y_Q - x_P x_Q}{1 - d x_P x_Q y_P y_Q} \right)$$

  - doubling: same as addition
- Strongly unified and complete addition law:
  - works for both addition and doubling
  - no exceptional case

  $\Rightarrow$ resilient against timing or power analysis attacks

Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)
  - neutral element: $\mathcal{O} = (0, 1)$
  - opposite: for all $P = (x_P, y_P) \in C(\mathbb{F}_q)$, $-P = (-x_P, y_P)$
  - addition: for all $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)$, then

$$P + Q = \left(\frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q}, \frac{y_P y_Q - x_P x_Q}{1 - d x_P x_Q y_P y_Q} \right)$$

  - doubling: same as addition
- Strongly unified and complete addition law:
  - works for both addition and doubling
  - no exceptional case

  $\Rightarrow$ resilient against timing or power analysis attacks
- Inverted coordinates: points $(X : Y : Z)$ with $(x, y) = (Z/X, Z/Y)$
  - addition: 9M + 1S
  - doubling: 3M + 4S

Edwards curves

$$C/\mathbb{F}_q : x^2 + y^2 = 1 + dx^2y^2$$

- Addition and doubling formulae (assuming d is not a square in $\mathbb{F}_q$)
  - neutral element: $\mathcal{O} = (0, 1)$
  - opposite: for all $P = (x_P, y_P) \in C(\mathbb{F}_q)$, $-P = (-x_P, y_P)$
  - addition: for all $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in C(\mathbb{F}_q)$, then

$$P + Q = \left(\frac{x_P y_Q + x_Q y_P}{1 + dx_P x_Q y_P y_Q}, \frac{y_P y_Q - x_P x_Q}{1 - dx_P x_Q y_P y_Q} \right)$$

  - doubling: same as addition
- Strongly unified and complete addition law:
  - works for both addition and doubling
  - no exceptional case

  $\Rightarrow$ resilient against timing or power analysis attacks
- Inverted coordinates: points $(X : Y : Z)$ with $(x, y) = (Z/X, Z/Y)$
  - addition: 9M + 1S
  - doubling: 3M + 4S
- Generalization by Bernstein *et al.* (2008): twisted Edwards curves

  $C/\mathbb{F}_q : ax^2 + y^2 = 1 + dx^2y^2$, with parameter $a, d \in \mathbb{F}_q$ and $\mathrm{char}(\mathbb{F}_q) \neq 2$
  - birationally equivalent to Montgomery curves

Outline

I. Scalar multiplication

II. Elliptic curve arithmetic

III. Finite field arithmetic

IV. Software considerations

V. Notions of hardware design

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$
- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with $n = |p|$ between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with prime m between 250 and 500

    ... still secure? [See M. Kosters' talk]

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$
- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with $n = |p|$ between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with prime m between 250 and 500
    ... still secure? [See M. Kosters' talk]
- What we have at our disposal:
  - basic integer arithmetic (addition, multiplication)
  - left and right shifts
  - bitwise logic operations (bitwise NOT, AND, etc.)

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$

- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with $n = |p|$ between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with prime m between 250 and 500
    ... still secure? [See M. Kosters' talk]

- What we have at our disposal:
  - basic integer arithmetic (addition, multiplication)
  - left and right shifts
  - bitwise logic operations (bitwise NOT, AND, etc.)

- ... on w-bit words:
  - $w = 32$ or 64 on CPUs
  - $w = 8$ or 16 bits on microcontrollers
  - a bit more flexibility in hardware
    (but integer arithmetic with $w > 64$ bits is hard!)

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$

- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with $n = |p|$ between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with prime m between 250 and 500
    ... still secure? [See M. Kosters' talk]

- What we have at our disposal:
  - basic integer arithmetic (addition, multiplication)
  - left and right shifts
  - bitwise logic operations (bitwise NOT, AND, etc.)

- ... on w-bit words:
  - $w = 32$ or 64 on CPUs
  - $w = 8$ or 16 bits on microcontrollers
  - a bit more flexibility in hardware
    (but integer arithmetic with $w > 64$ bits is hard!)

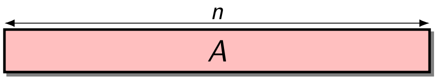
  $\Rightarrow$ elements of $\mathbb{F}_q$ represented using several words

Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime

Multiprecision representation

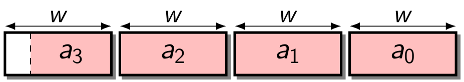
- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P



Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

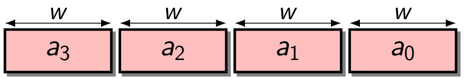
$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$



Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

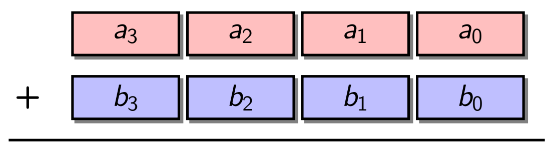


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:

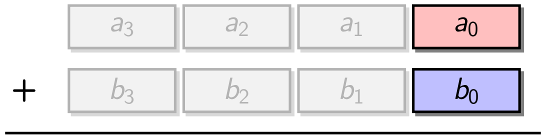


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition

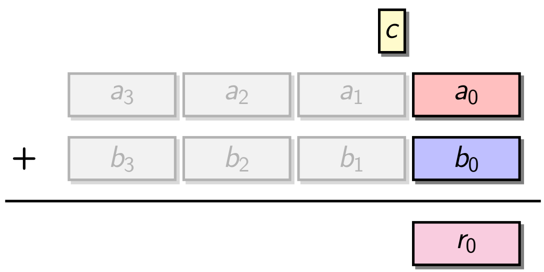


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

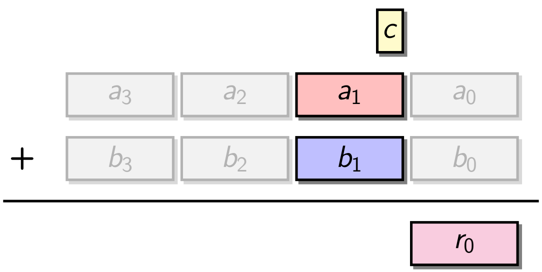


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

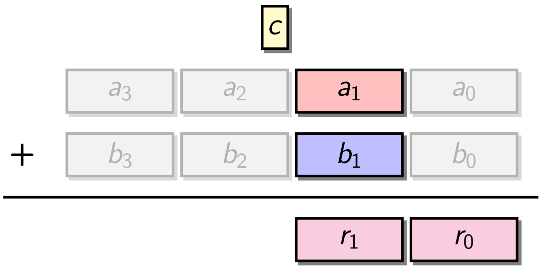


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

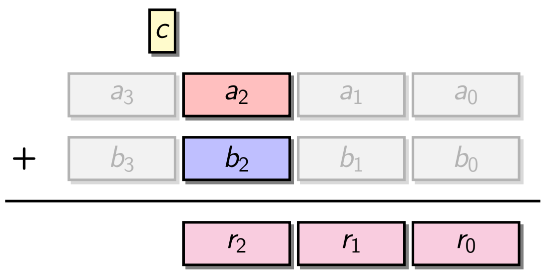


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

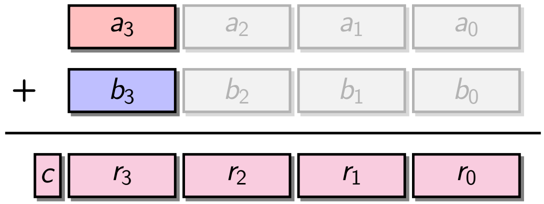


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

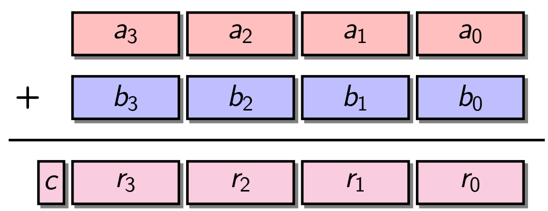


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

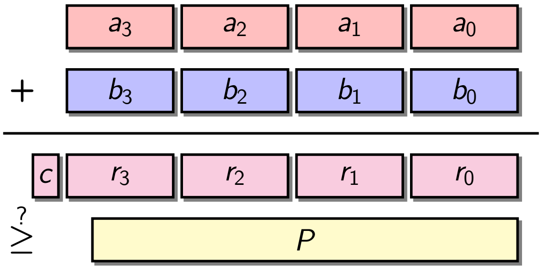


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry
  - might need reduction modulo P: compare then subtract (in constant time!)

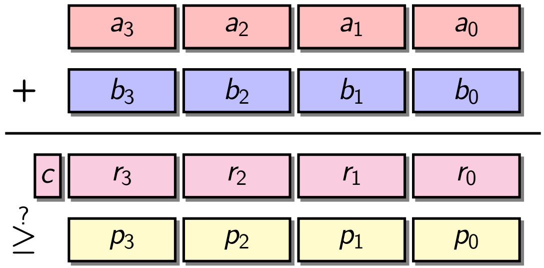


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry
  - might need reduction modulo P: compare then subtract (in constant time!)

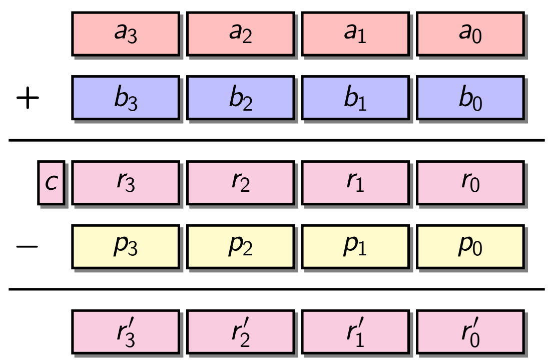


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry
  - might need reduction modulo P: compare then subtract (in constant time!)

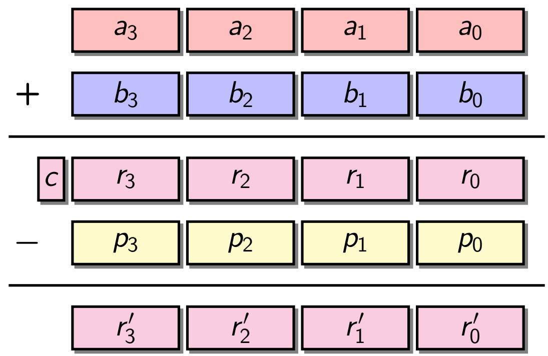


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

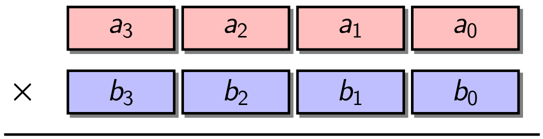
$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry
  - might need reduction modulo P: compare then subtract (in constant time!)
  - lazy reduction: if $kw > n$, do not reduce after each addition



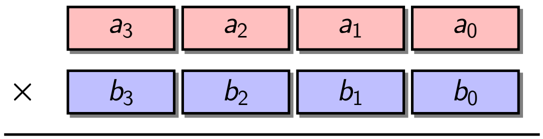
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:



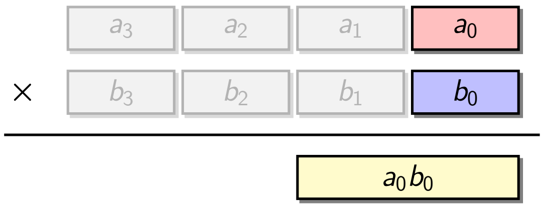
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



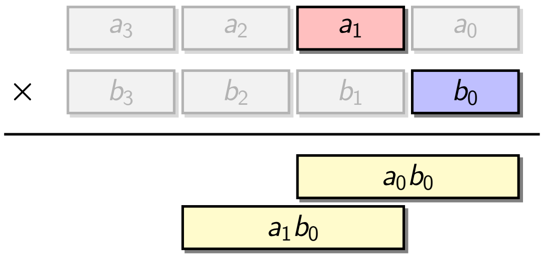
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



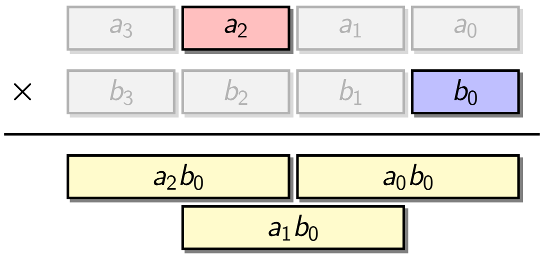
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



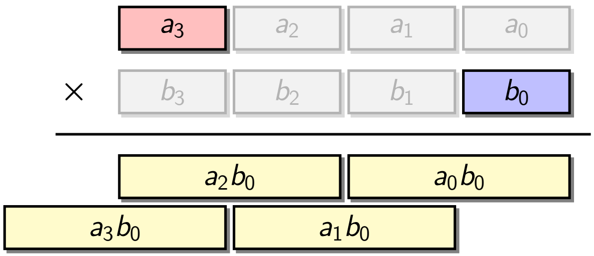
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



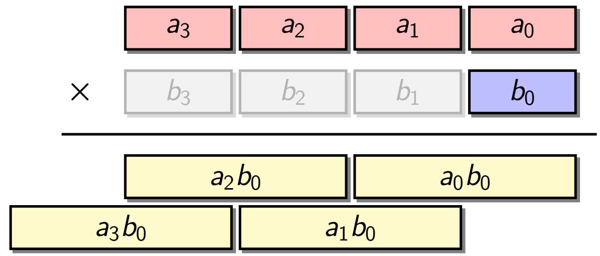
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



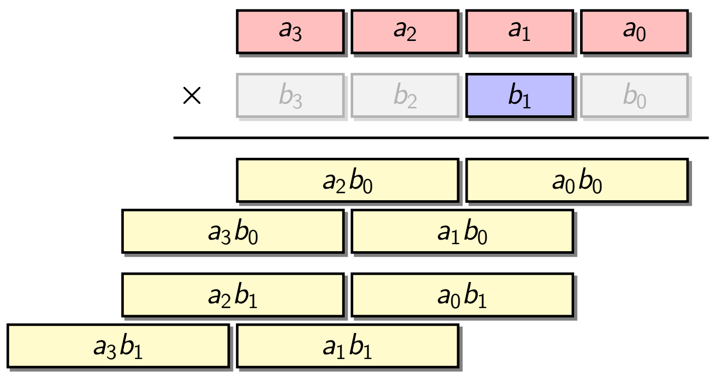
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



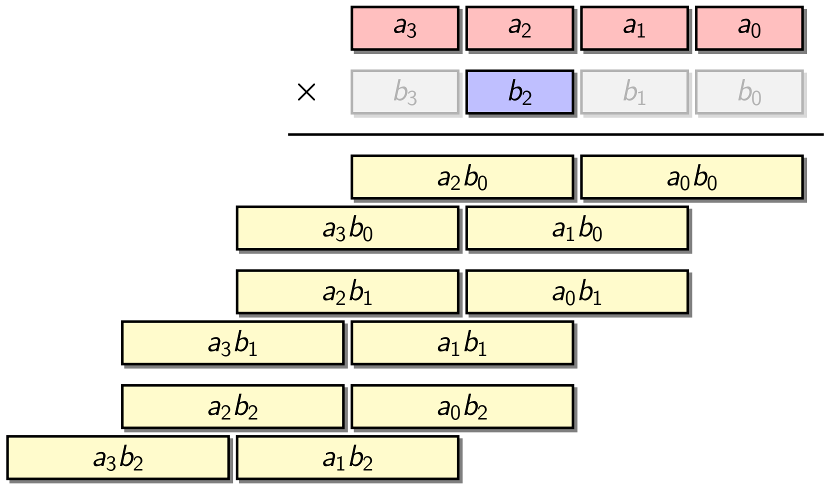
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



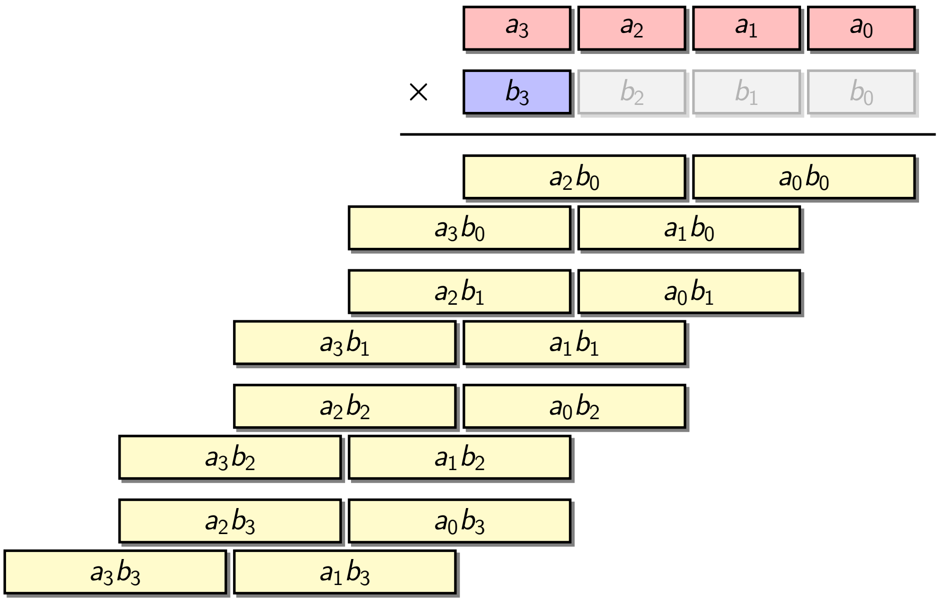
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



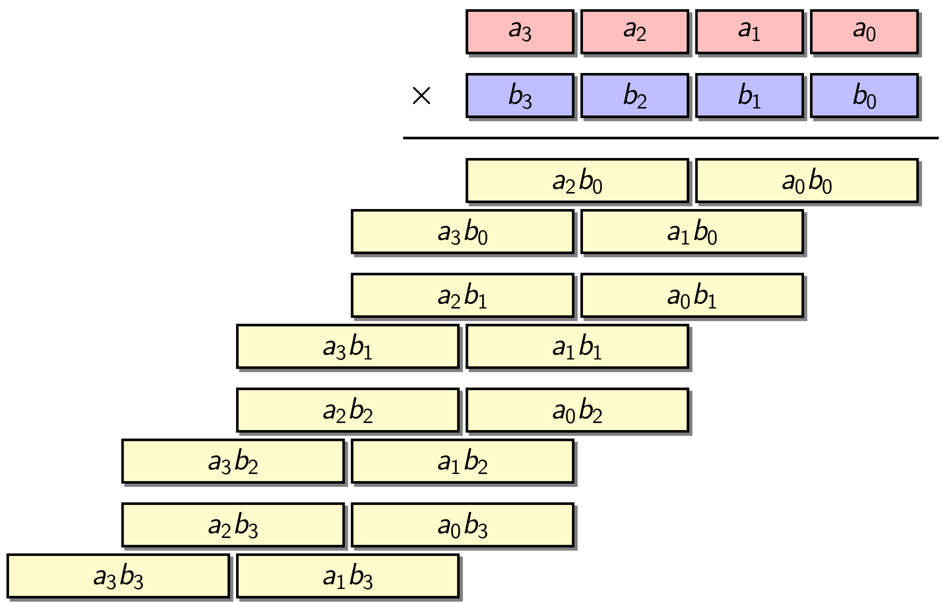
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



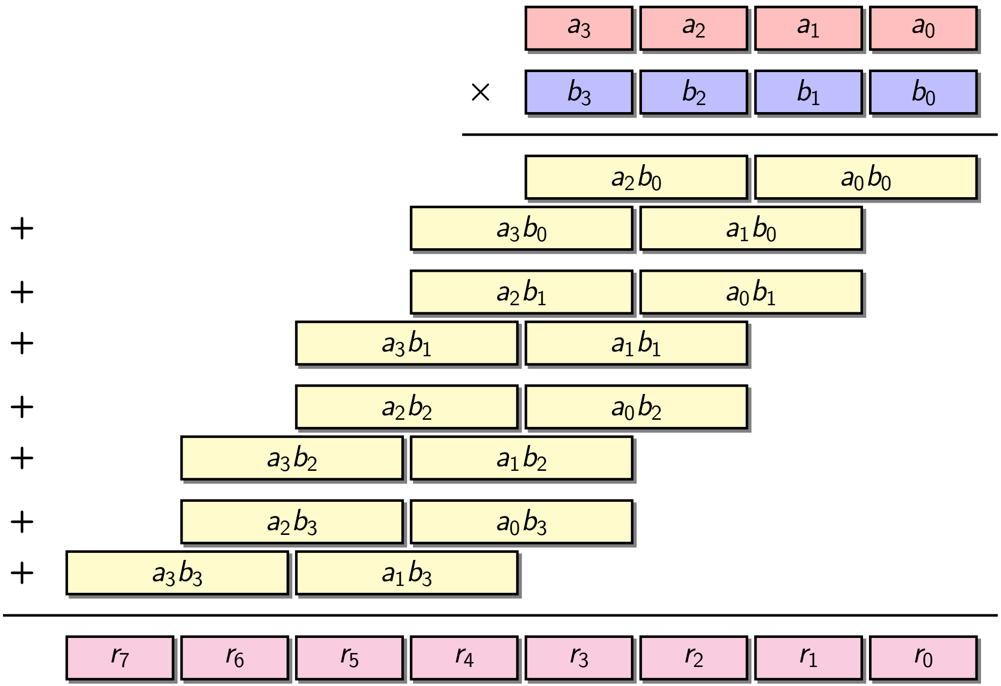
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications



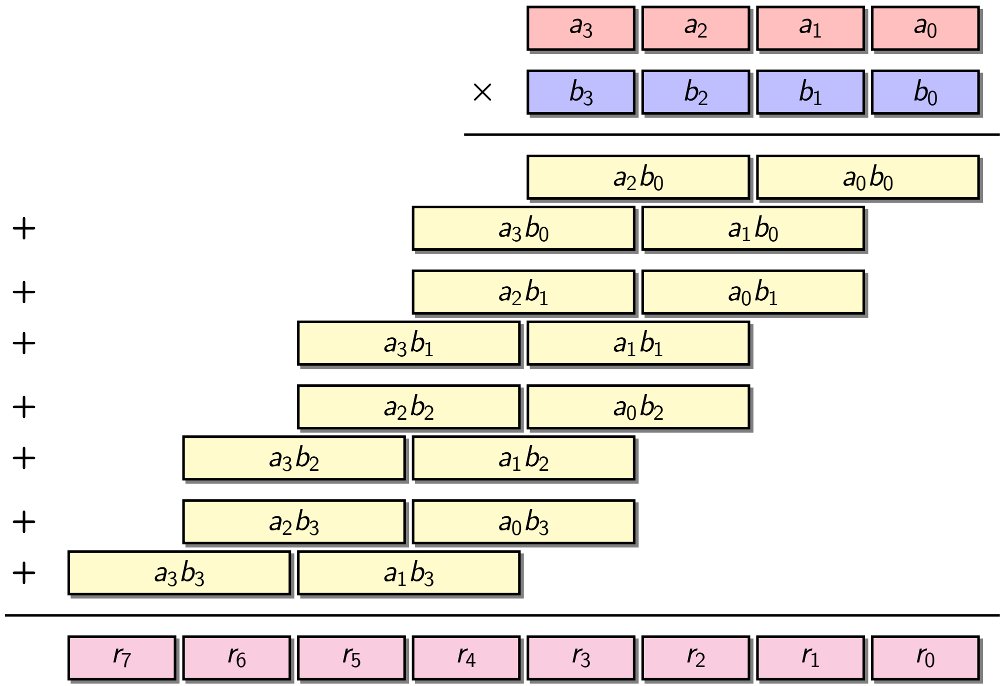
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications
  - final product fits into $2k$ words



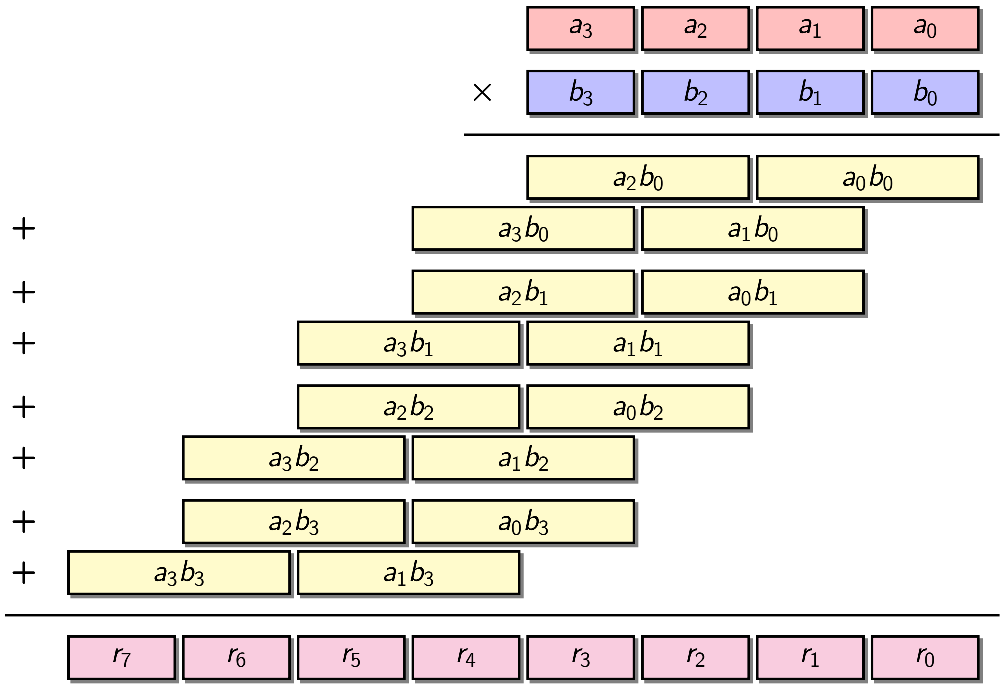
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications
  - final product fits into $2k$ words
  - need to reduce product modulo P (see later)



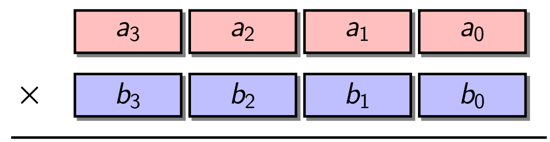
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_p$:
  - schoolbook method: k^2 w-by-w-bit multiplications
  - final product fits into $2k$ words
  - need to reduce product modulo P (see later)
  - should run in constant time (for fixed P)!



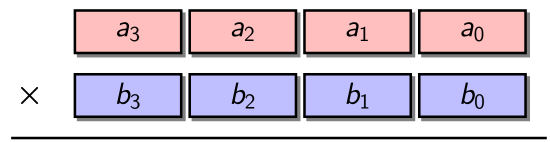
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?



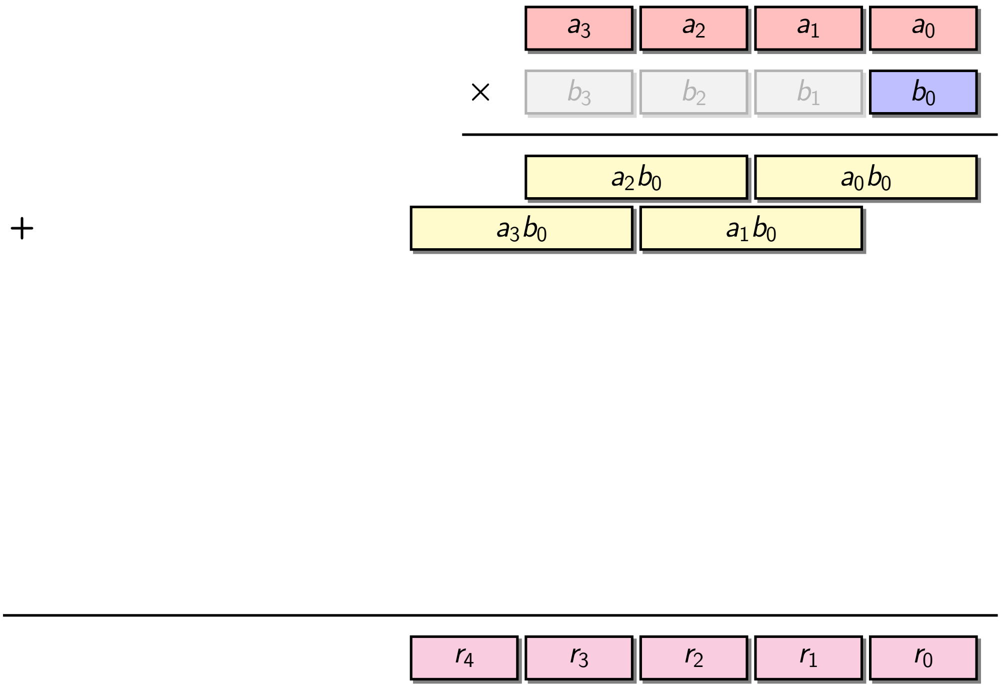
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning



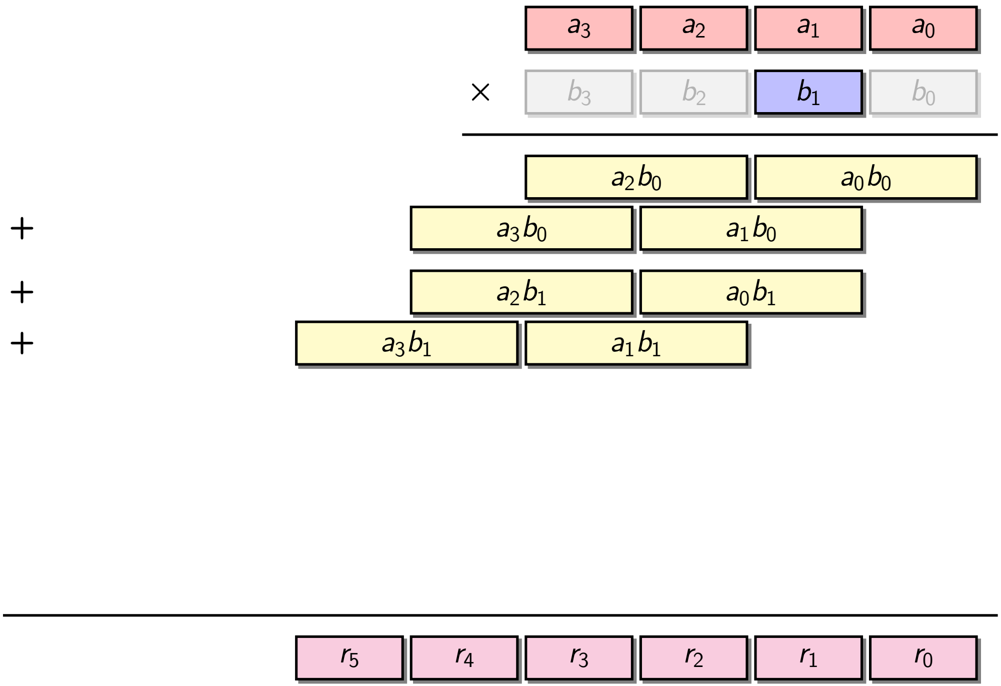
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning



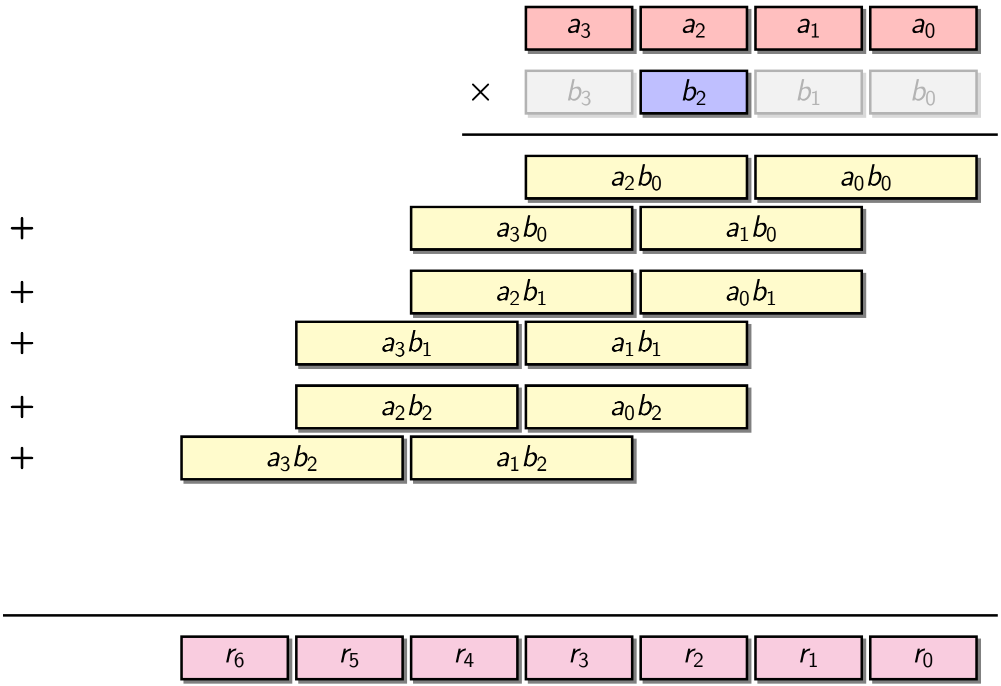
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning



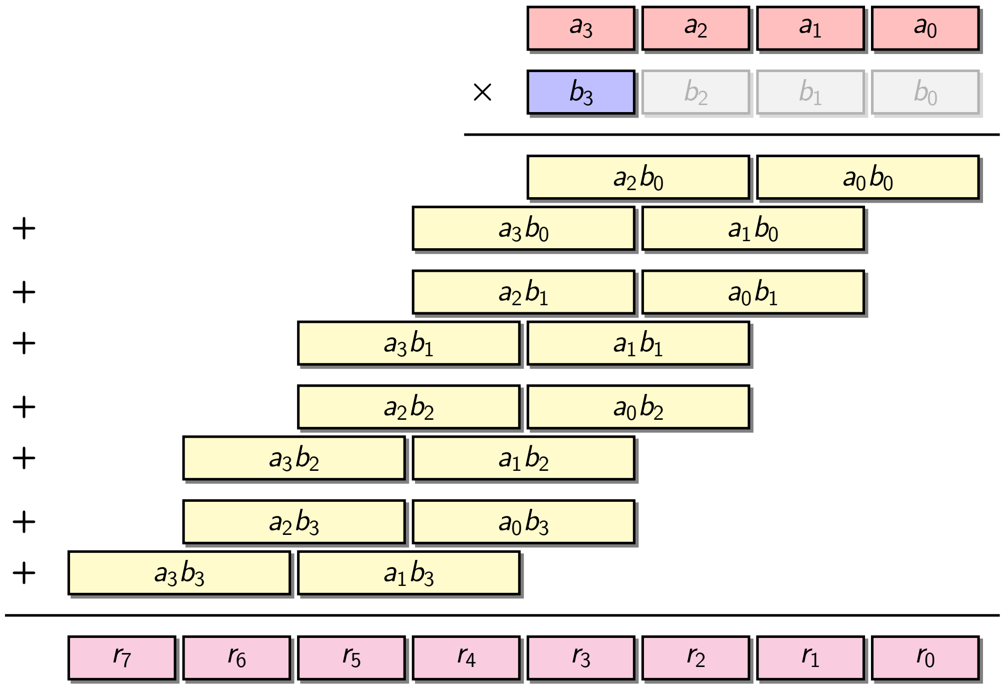
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning



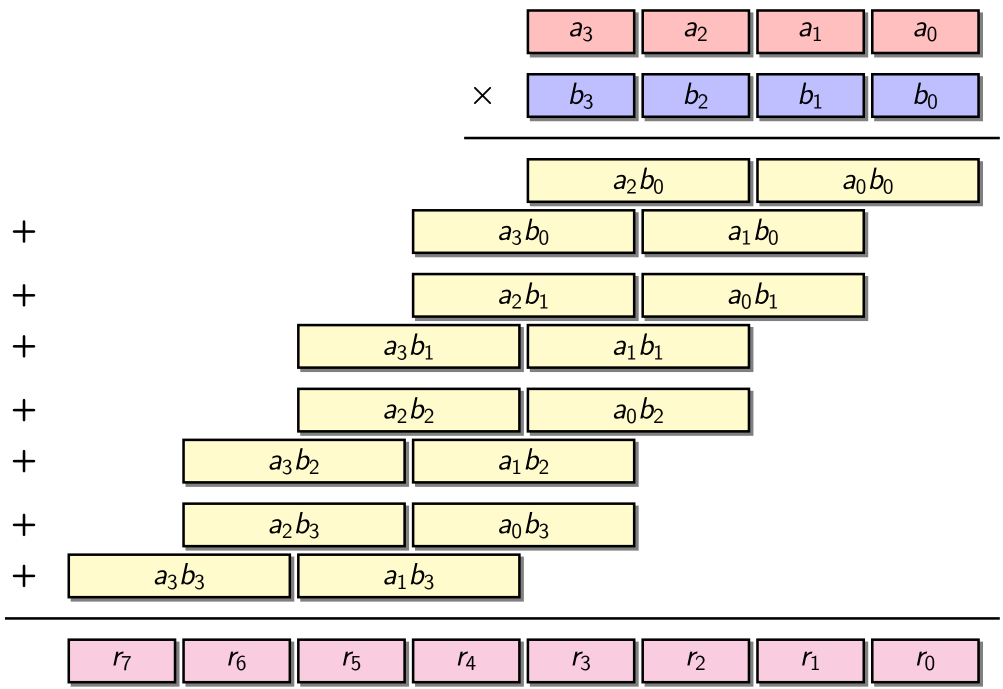
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning



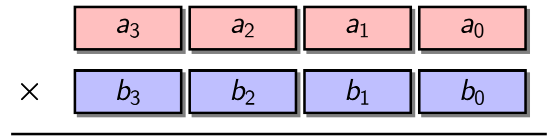
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control



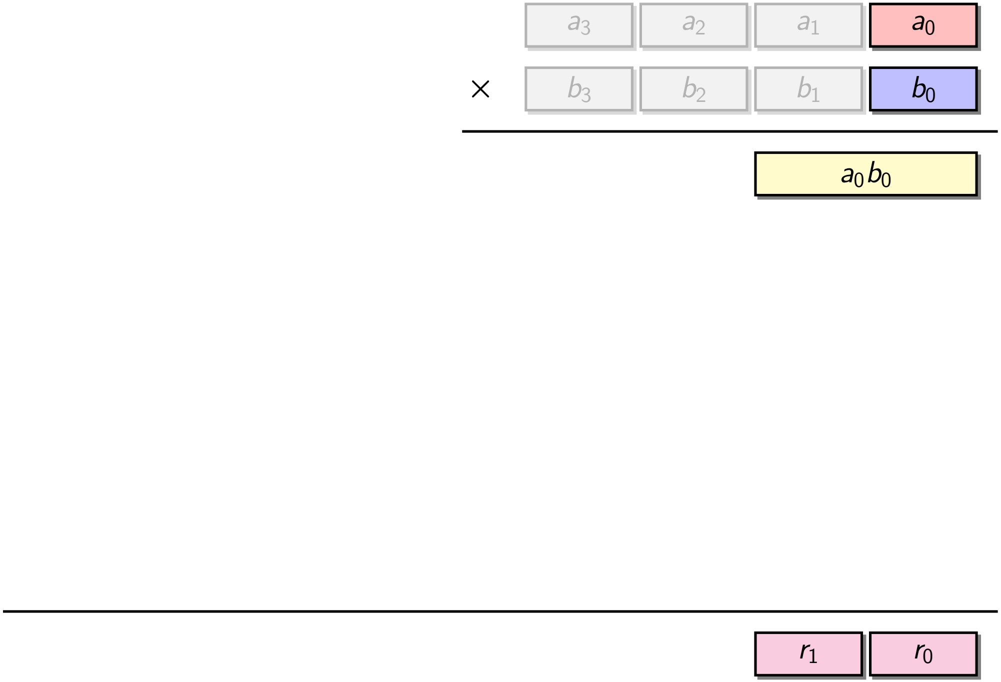
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning



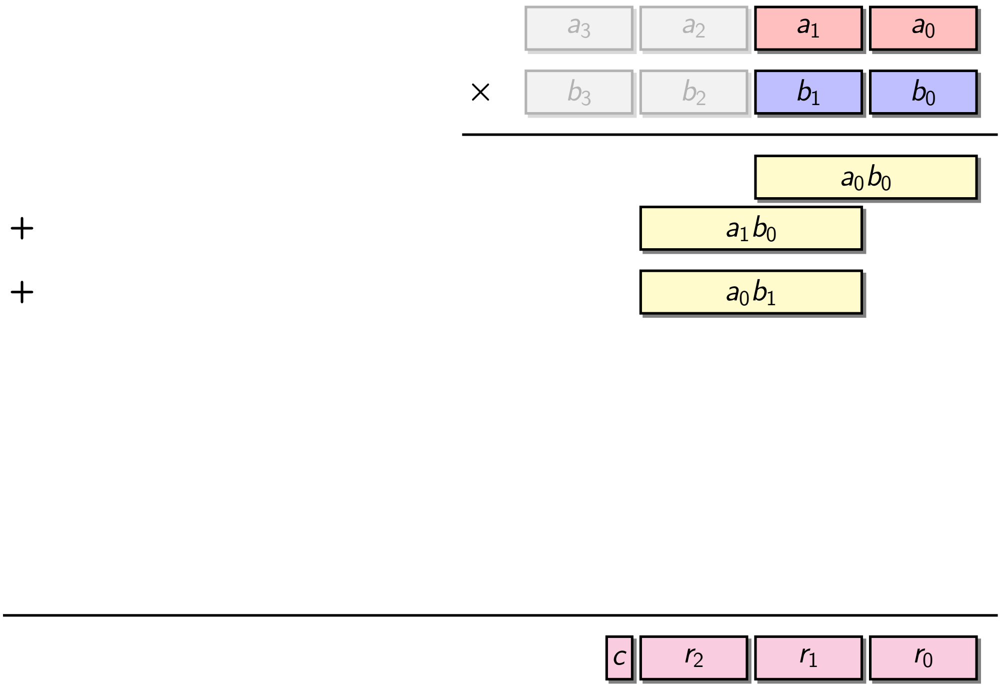
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning



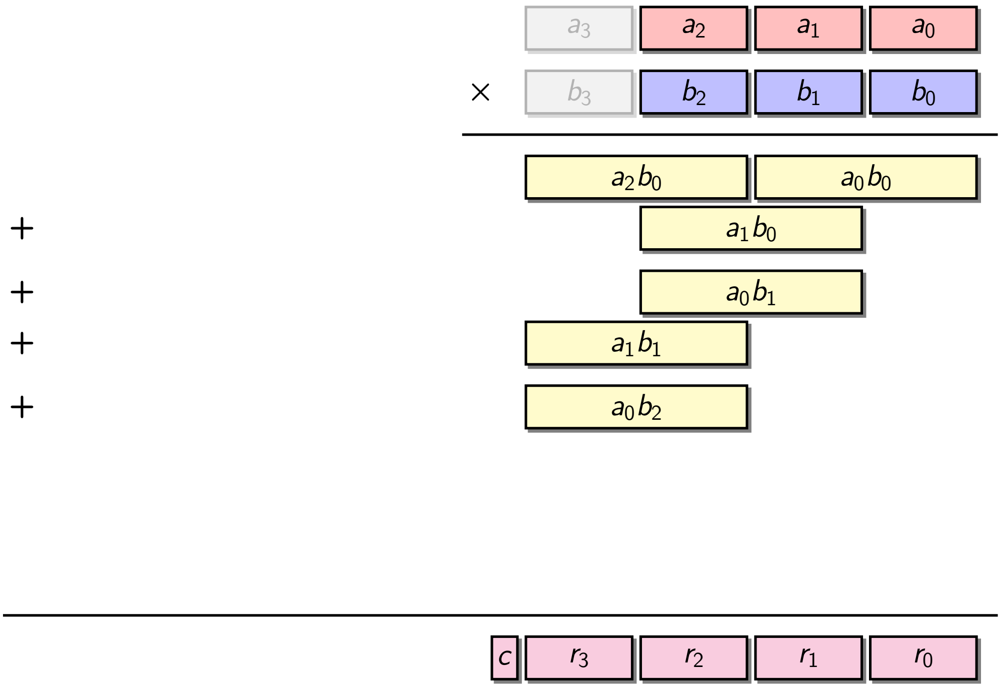
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning



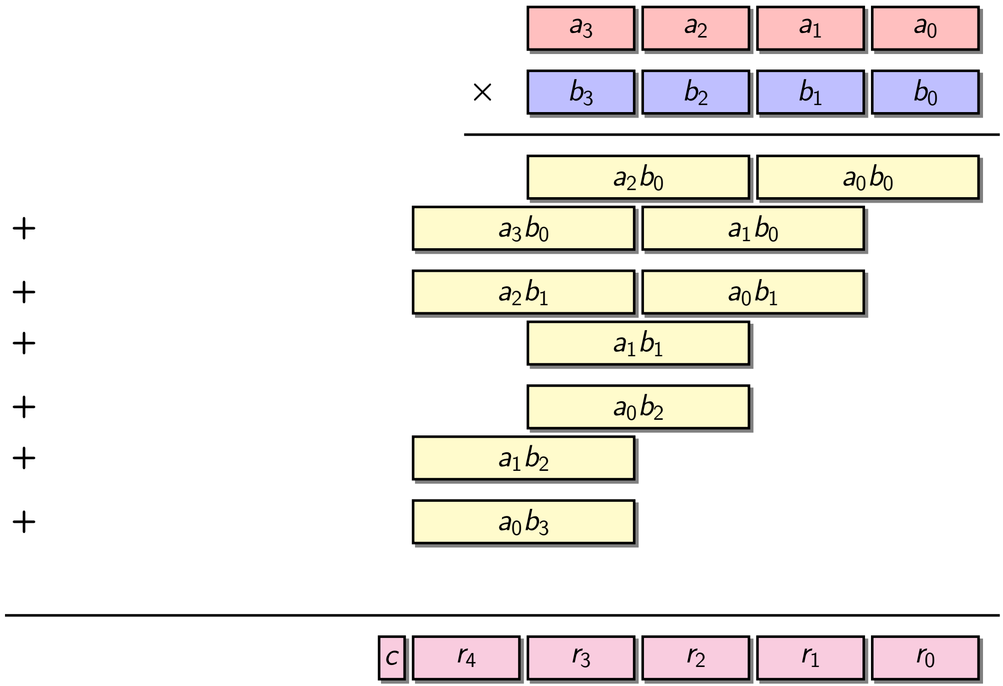
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning



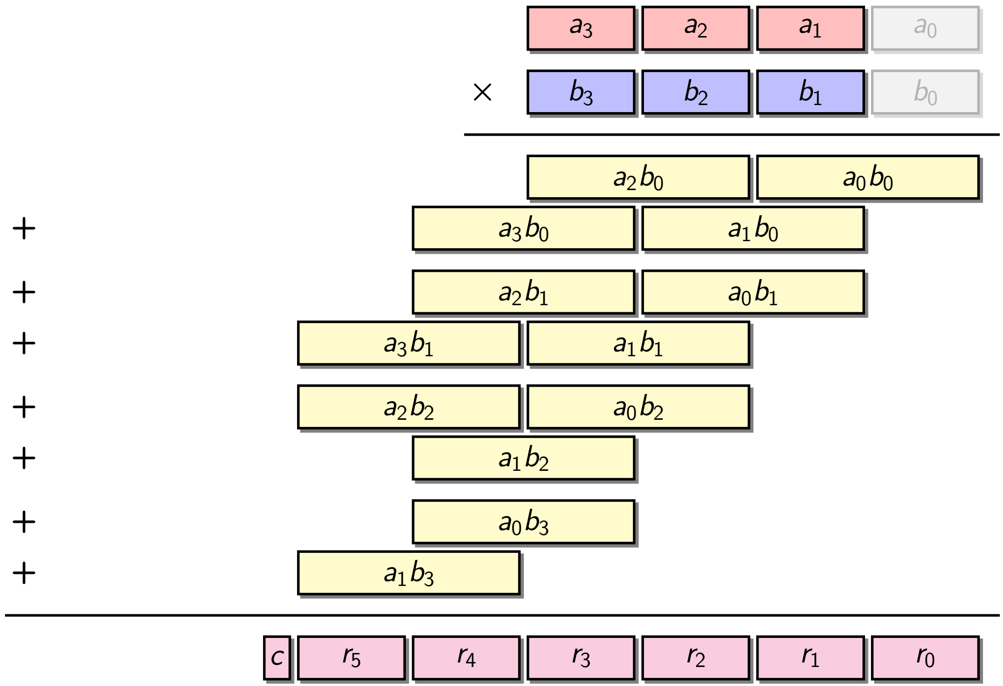
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning



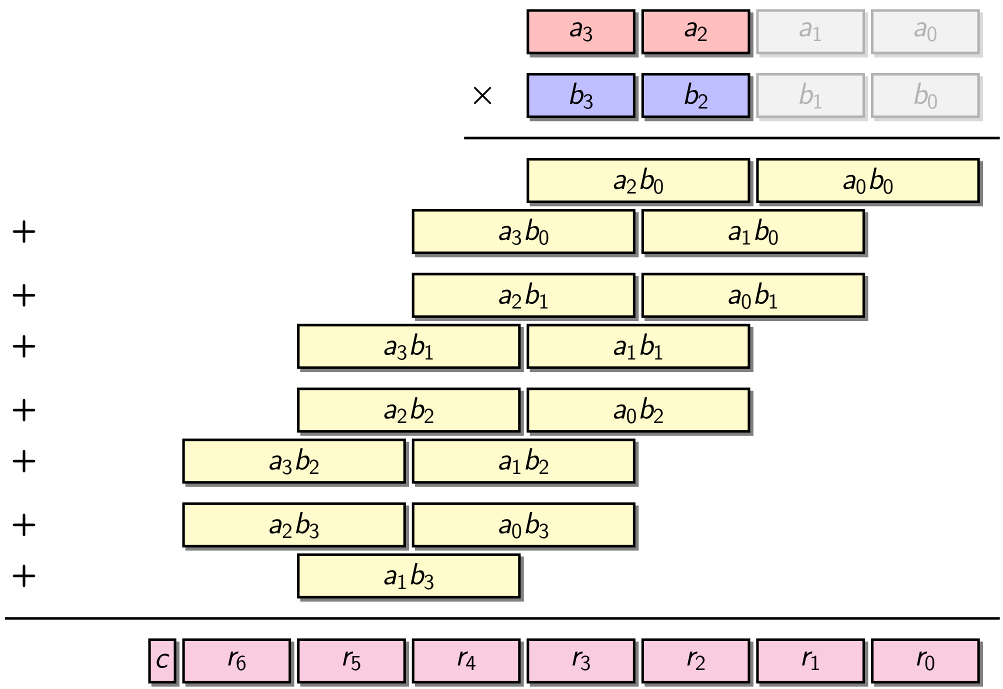
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning



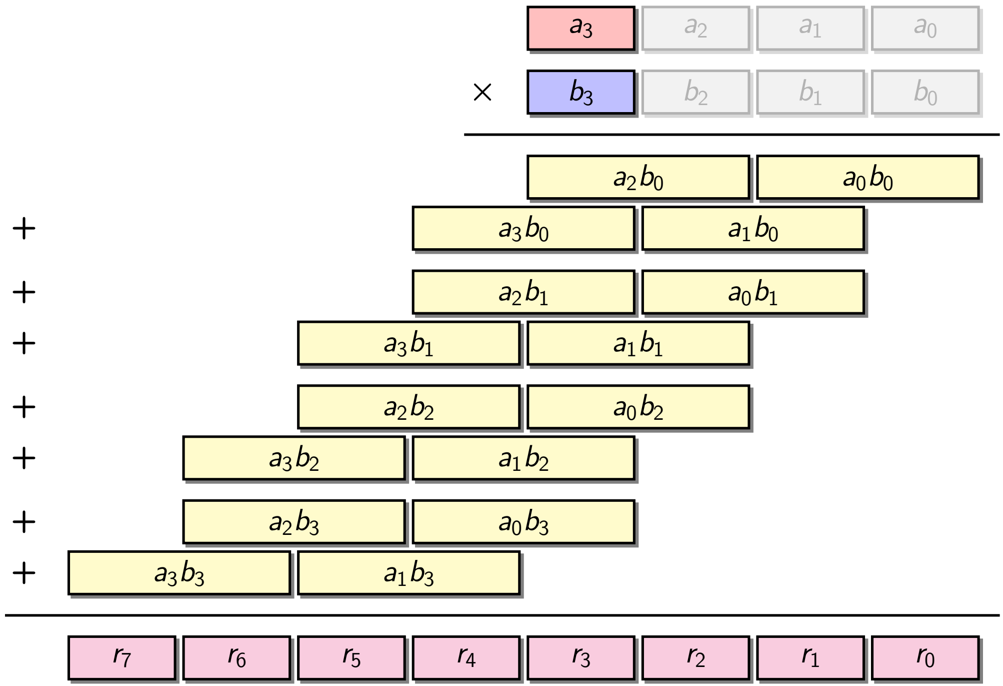
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning



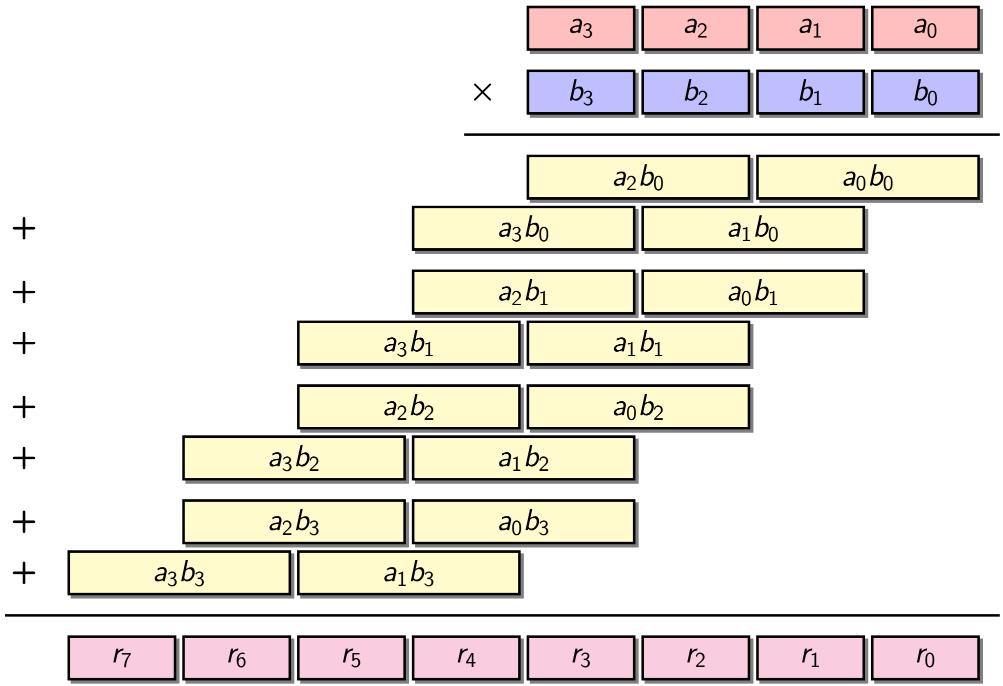
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning



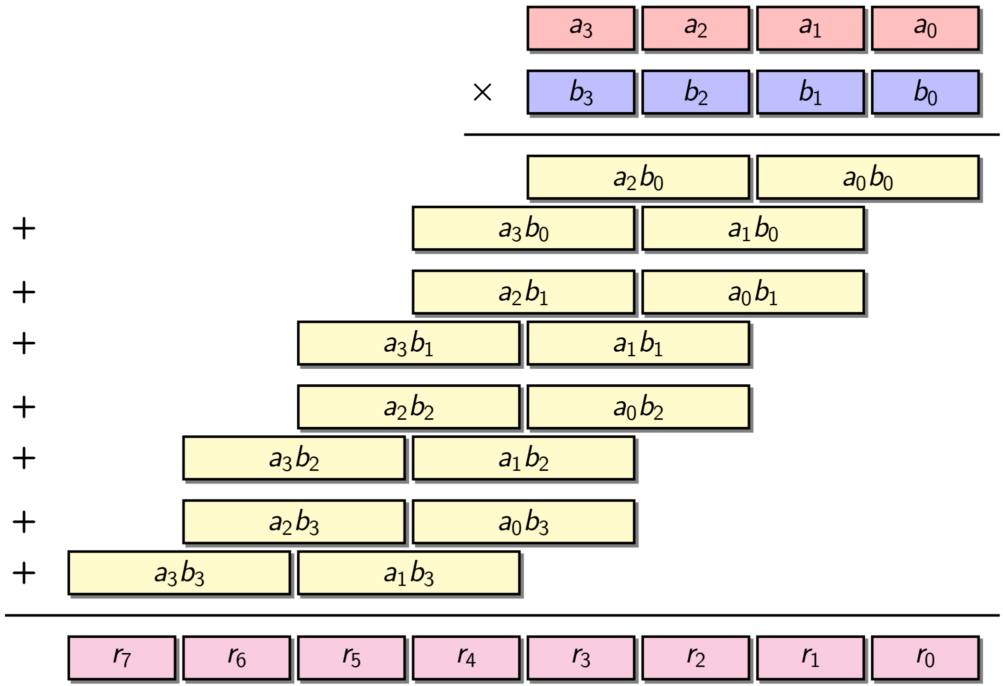
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning: fewer memory accesses and carry propagations



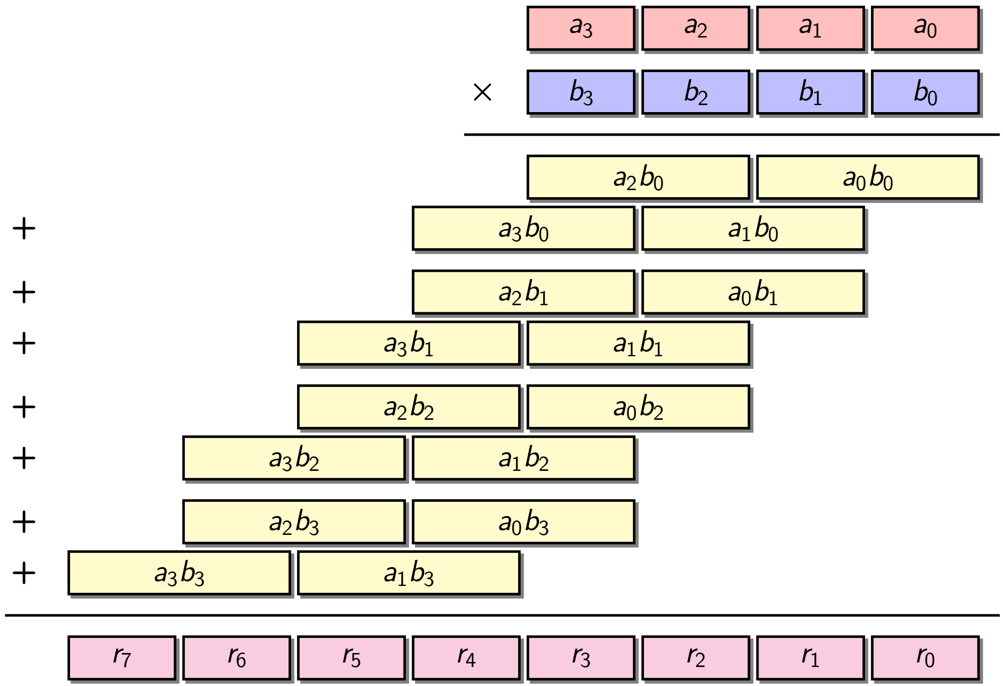
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning: fewer memory accesses and carry propagations
  - many variants, such as left-to-right



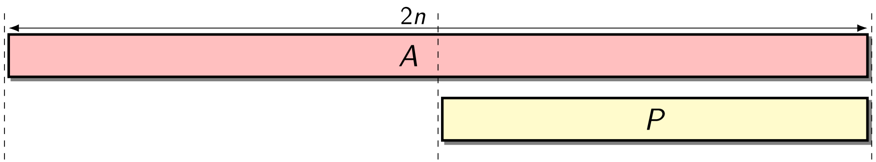
MP multiplication: operand vs. product scanning

- In which order should we compute the subproducts $a_i b_j$?
  - operand scanning: straightforward, regular loop control
  - product scanning: fewer memory accesses and carry propagations
  - many variants, such as left-to-right
  - subquadratic algorithms (e.g., Karatsuba) when k is large



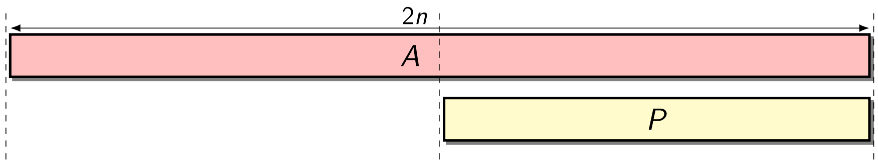
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$



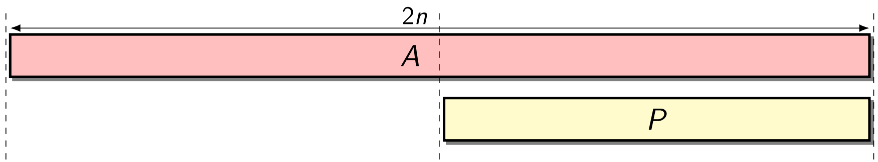
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)



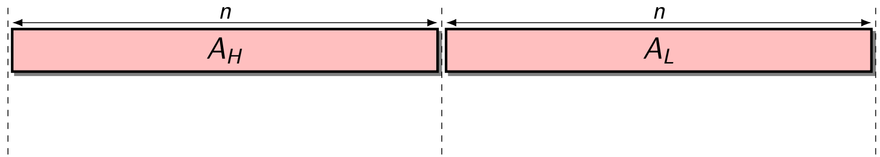
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$



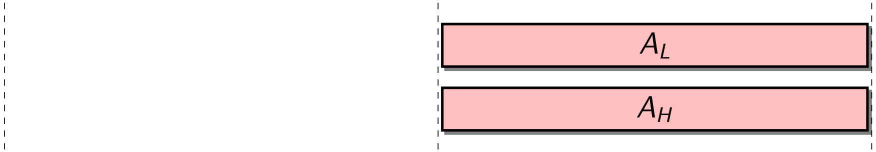
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$



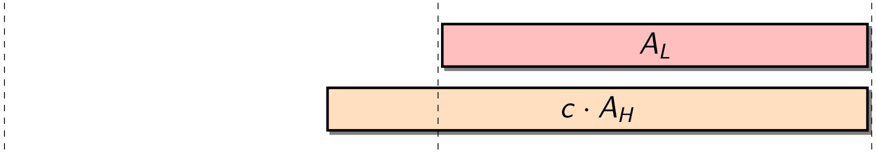
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times w$-word multiplication)



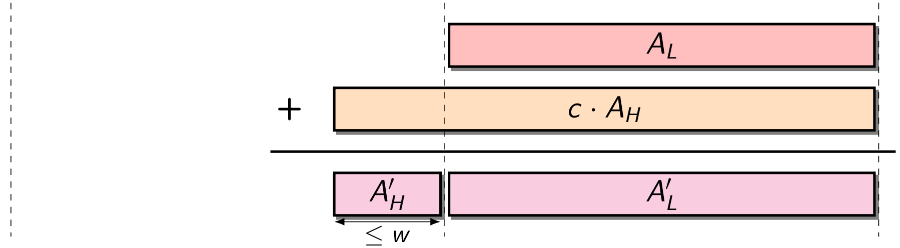
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times w$-word multiplication)



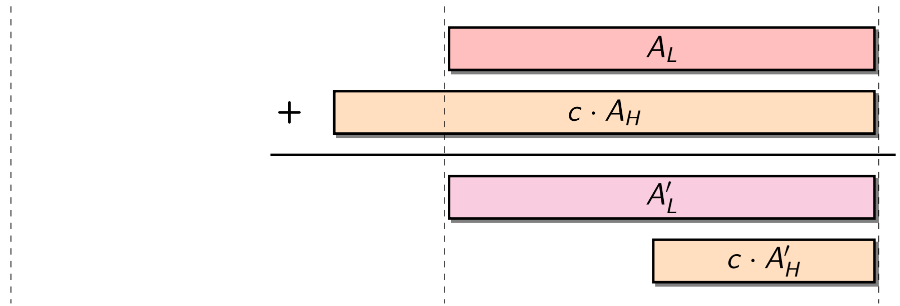
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times w$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)



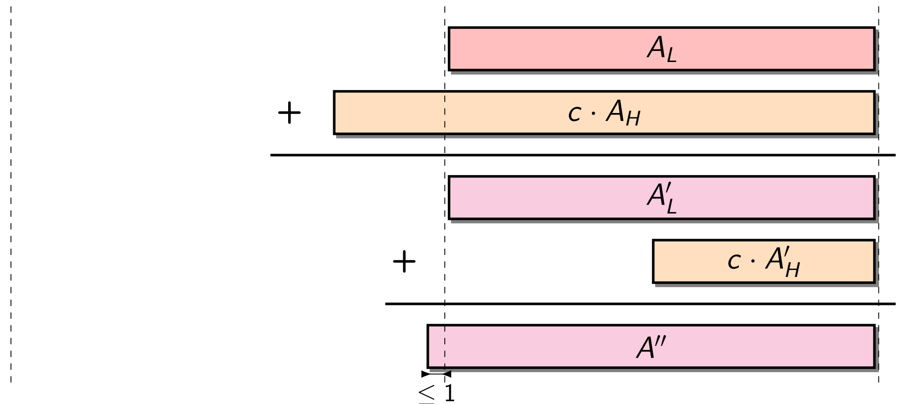
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times w$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)



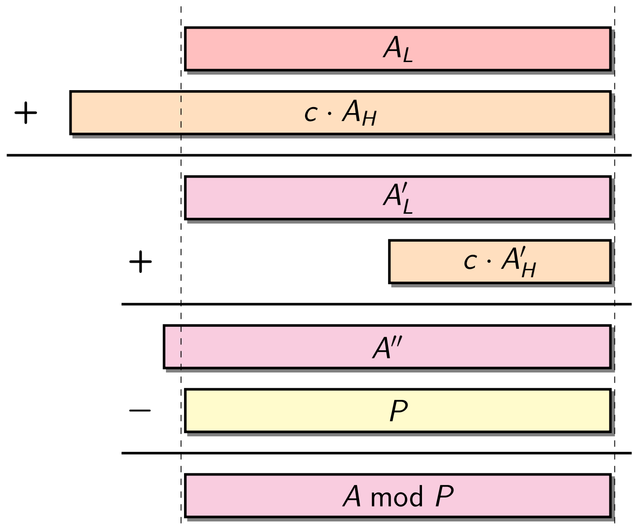
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times w$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)



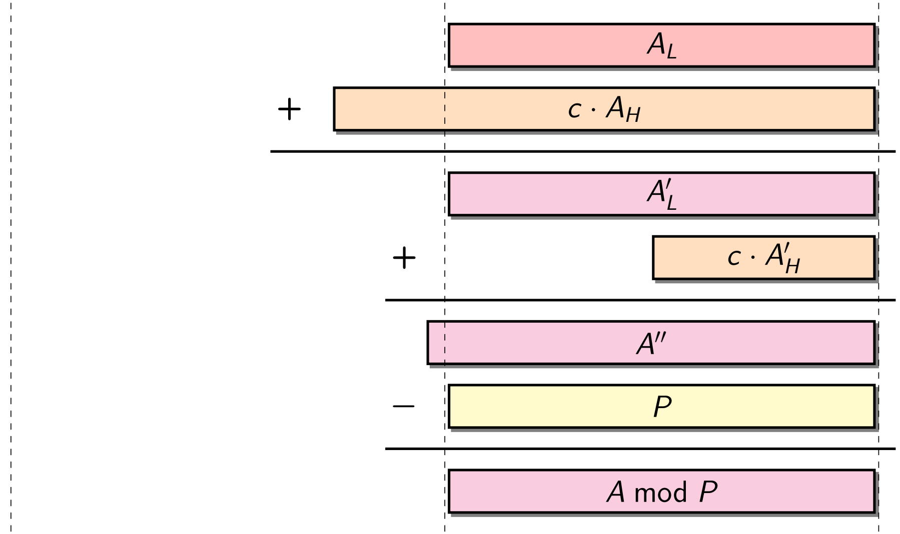
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c “small” (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times w$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)
  - final subtraction might be necessary



MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times w$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)
  - final subtraction might be necessary
- Examples: $P = 2^{255} - 19$ (Curve25519) or $P = 2^{448} - 2^{224} - 1$ (Ed448-Goldilocks)



MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$

MP modular reduction: general case

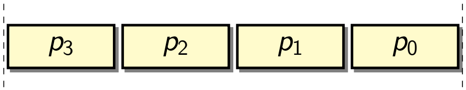
- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!

MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision

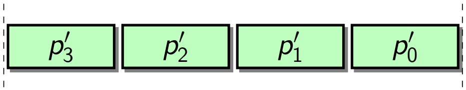
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:



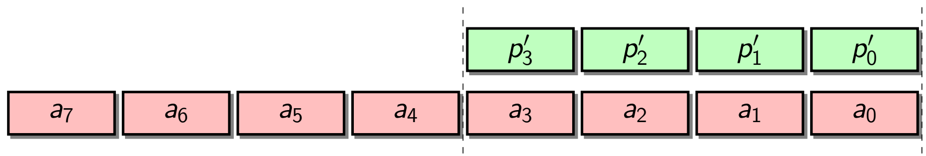
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)



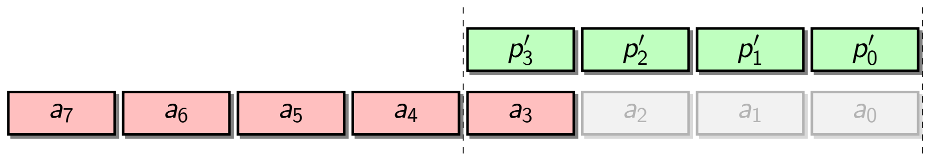
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$



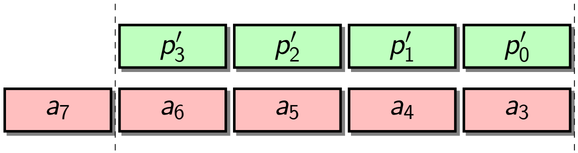
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$



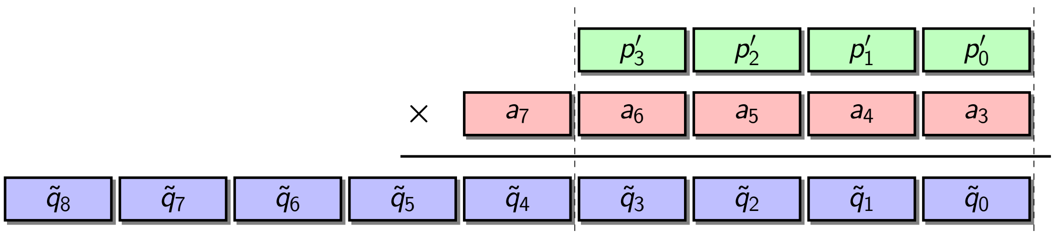
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$



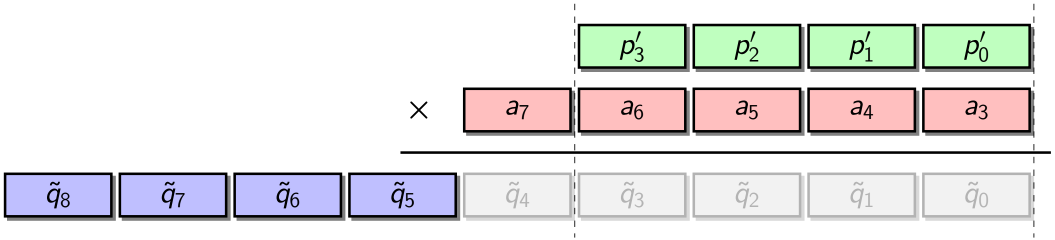
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times k$-word multiplication)



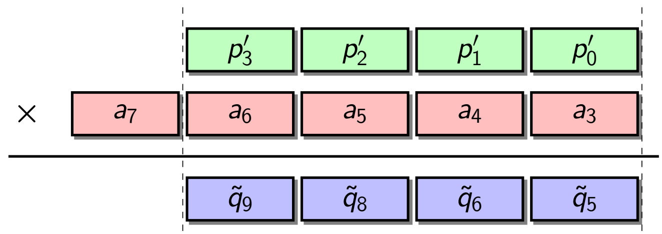
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times k$-word multiplication)



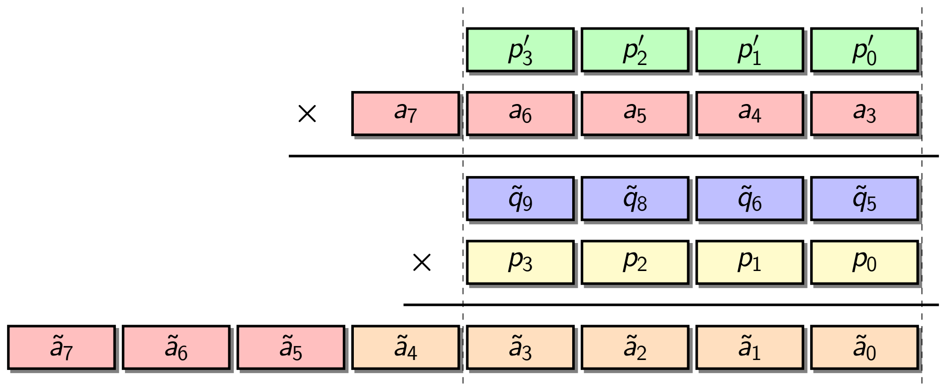
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times k$-word multiplication)



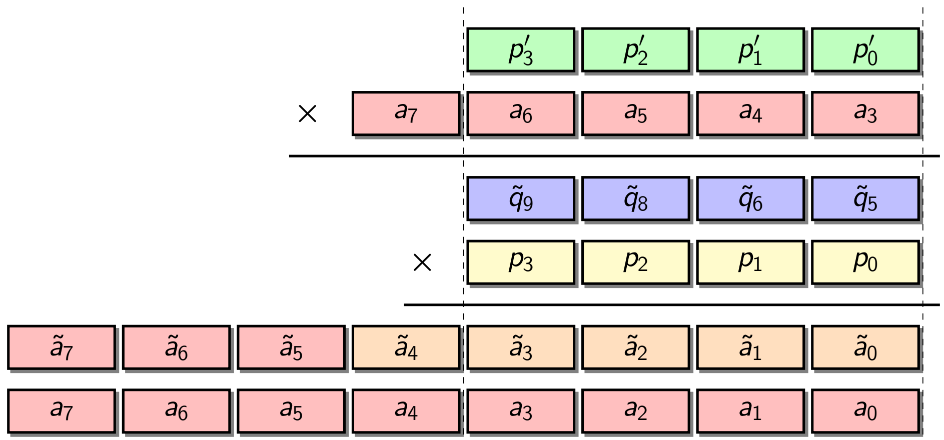
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times k$-word multiplication)
  - compute $\tilde{A} \leftarrow \tilde{Q} \cdot P$ (one $k \times k$-word multiplication)



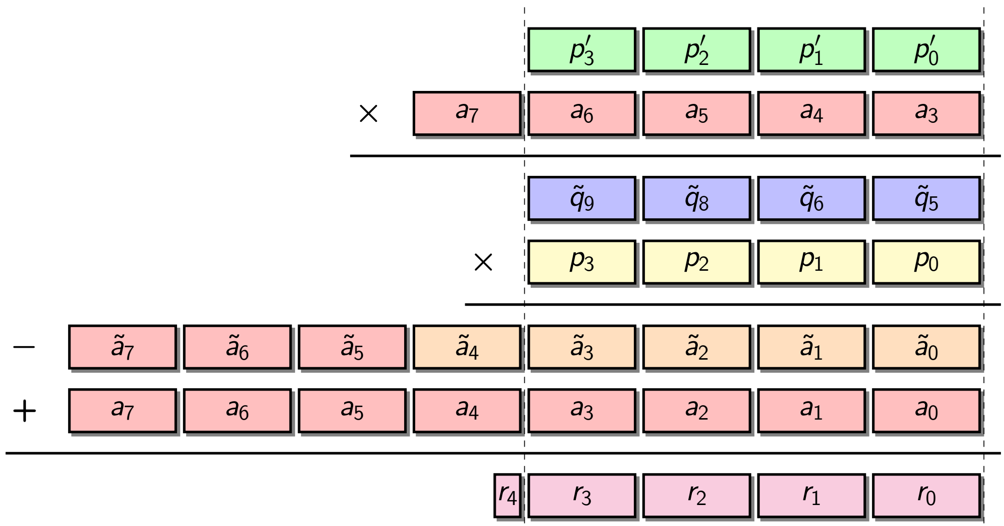
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times k$-word multiplication)
  - compute $\tilde{A} \leftarrow \tilde{Q} \cdot P$ (one $k \times k$-word multiplication)



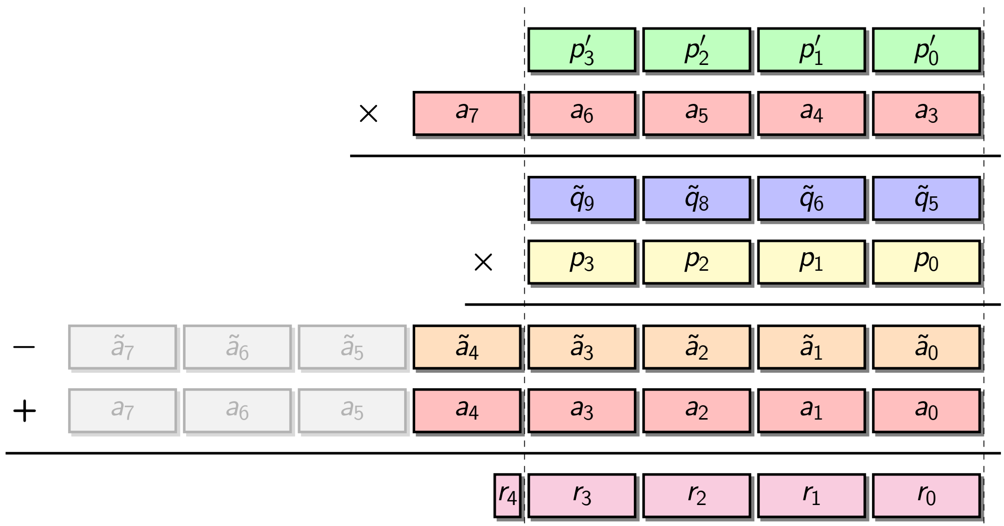
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times k$-word multiplication)
  - compute $\tilde{A} \leftarrow \tilde{Q} \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A - \tilde{A}$



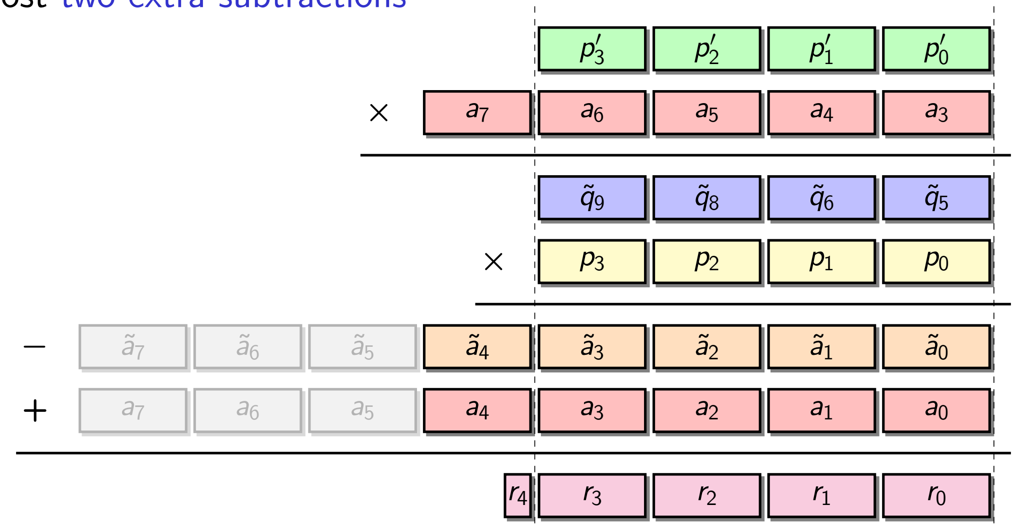
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times k$-word multiplication)
  - compute $\tilde{A} \leftarrow (\tilde{Q} \cdot P) \bmod 2^{(k+1)w}$ (one $k \times k$-word short multiplication)
  - compute remainder $R \leftarrow A - \tilde{A}$



MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ (k words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times k$-word multiplication)
  - compute $\tilde{A} \leftarrow (\tilde{Q} \cdot P) \bmod 2^{(k+1)w}$ (one $k \times k$-word short multiplication)
  - compute remainder $R \leftarrow A - \tilde{A}$
  - at most two extra subtractions

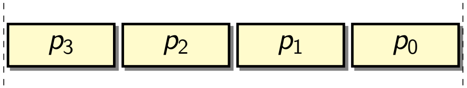


MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words

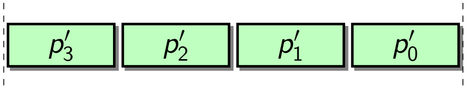
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$



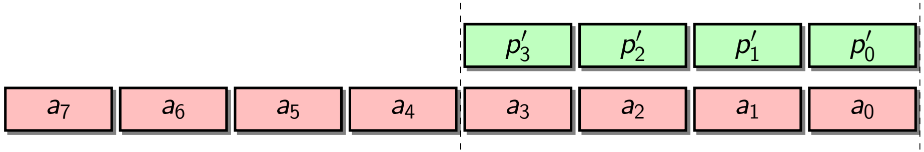
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)



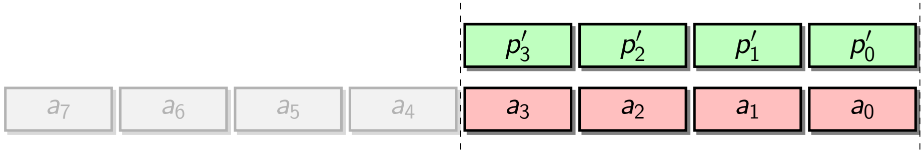
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)



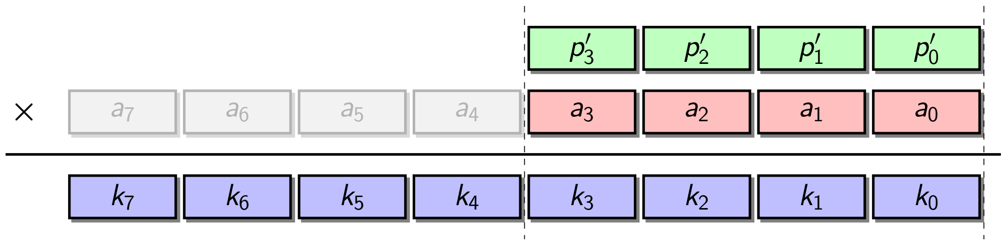
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)



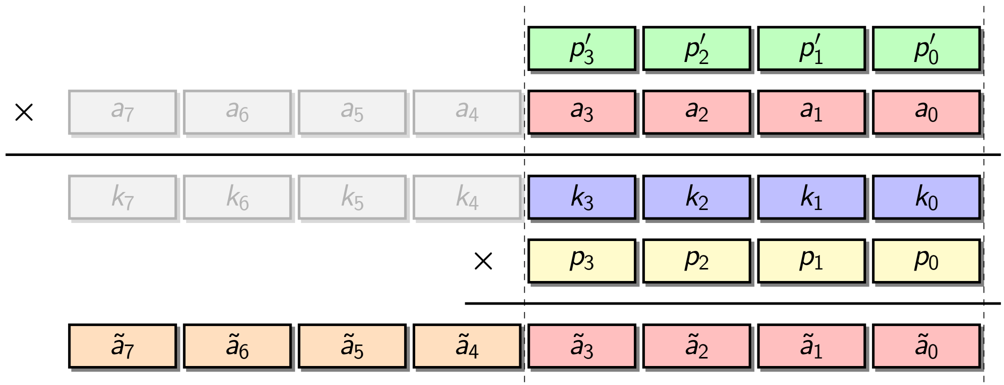
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)



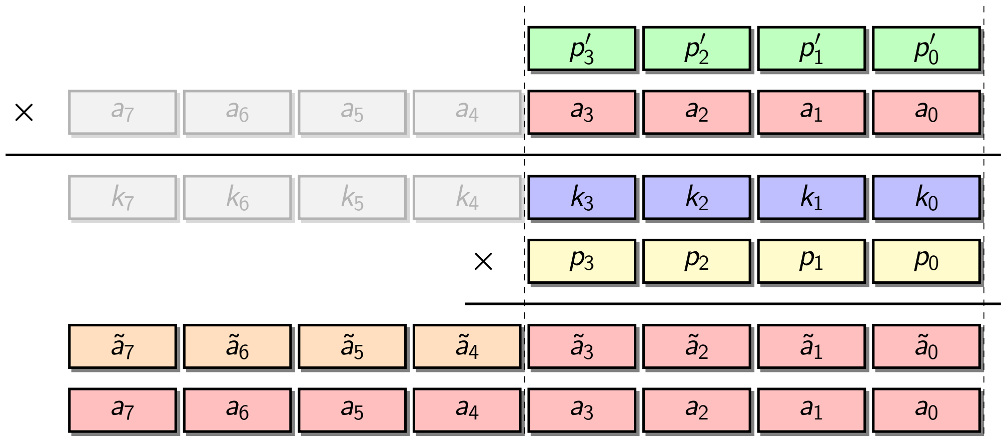
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)



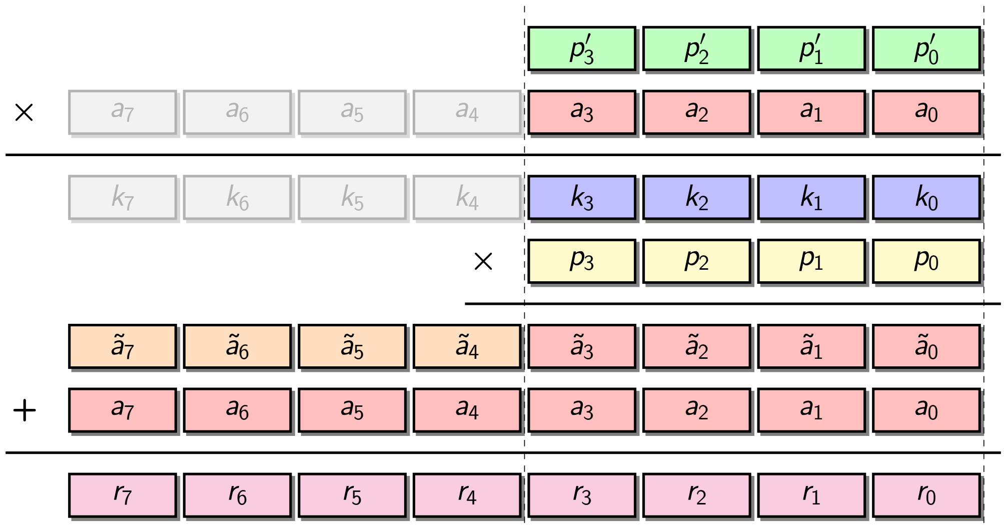
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow \ A + \tilde{A}$



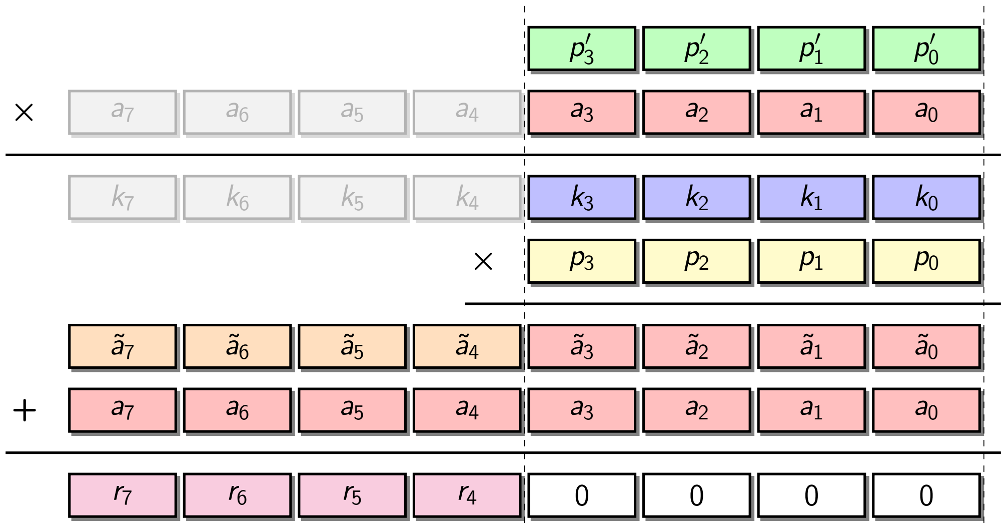
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A + \tilde{A}$



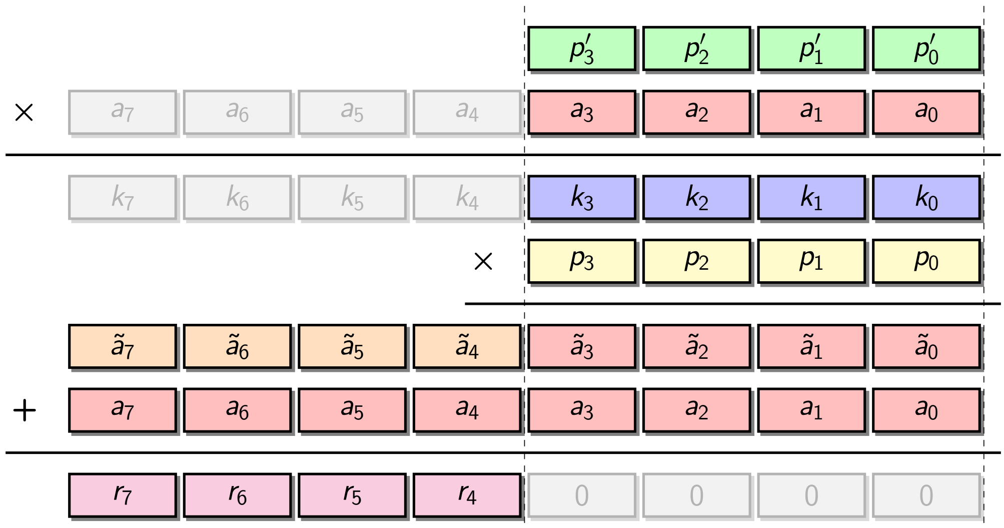
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow \; A + \tilde{A}$



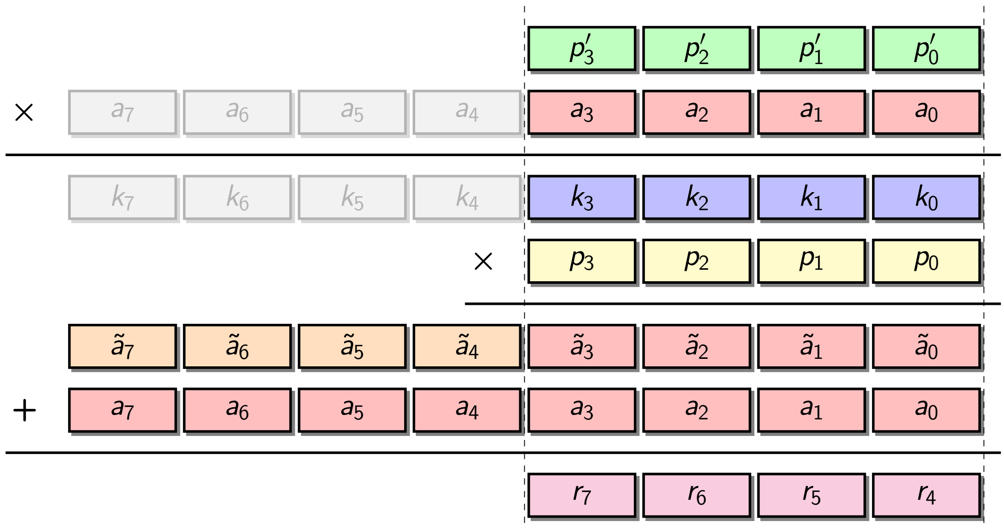
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A + \tilde{A}$



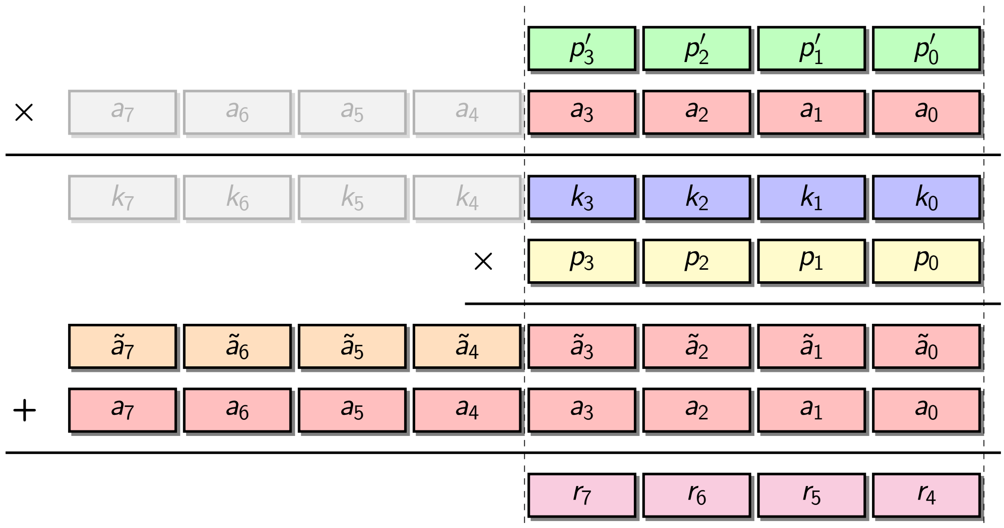
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$



MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction



MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- REDC(A) returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- REDC(A) returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- $\mathrm{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\mathrm{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

  $\rightarrow$ that's the so-called Montgomery multiplication

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- $\mathrm{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\mathrm{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

  $\rightarrow$ that's the so-called Montgomery multiplication
  - conversions:

$$\hat{X} = \mathrm{REDC}(X, 2^{2kw} \bmod P) \qquad \text{and} \qquad X = \mathrm{REDC}(\hat{X}, 1)$$

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction

- $\mathrm{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\mathrm{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

  $\rightarrow$ that's the so-called Montgomery multiplication
  - conversions:

$$\hat{X} = \mathrm{REDC}(X, 2^{2kw} \bmod P) \qquad \text{and} \qquad X = \mathrm{REDC}(\hat{X}, 1)$$

  - Montgomery representation is compatible with addition / subtraction in $\mathbb{F}_P$

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- $\mathrm{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\mathrm{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

    → that's the so-called Montgomery multiplication
  - conversions:

$$\hat{X} = \mathrm{REDC}(X, 2^{2kw} \bmod P) \qquad \text{and} \qquad X = \mathrm{REDC}(\hat{X}, 1)$$

  - Montgomery representation is compatible with addition / subtraction in $\mathbb{F}_P$

  ⇒ do all computations in Montgomery repr. instead of converting back and forth

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- $\mathrm{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\mathrm{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

    $\rightarrow$ that's the so-called Montgomery multiplication
  - conversions:

$$\hat{X} = \mathrm{REDC}(X, 2^{2kw} \bmod P) \qquad \text{and} \qquad X = \mathrm{REDC}(\hat{X}, 1)$$

  - Montgomery representation is compatible with addition / subtraction in $\mathbb{F}_P$

  $\Rightarrow$ do all computations in Montgomery repr. instead of converting back and forth
- REDC can be computed iteratively (one word at a time) and interleaved with the computation of $\hat{X} \cdot \hat{Y}$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

A

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

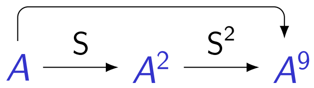
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

$$A \xrightarrow{S} A^2$$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

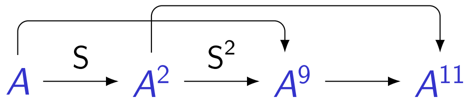
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

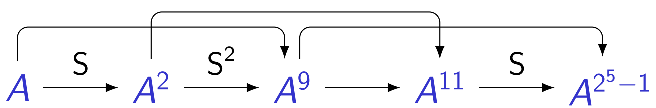
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

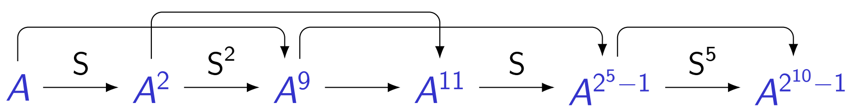
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

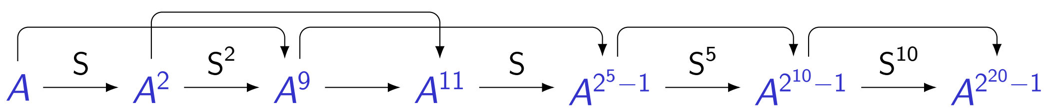
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

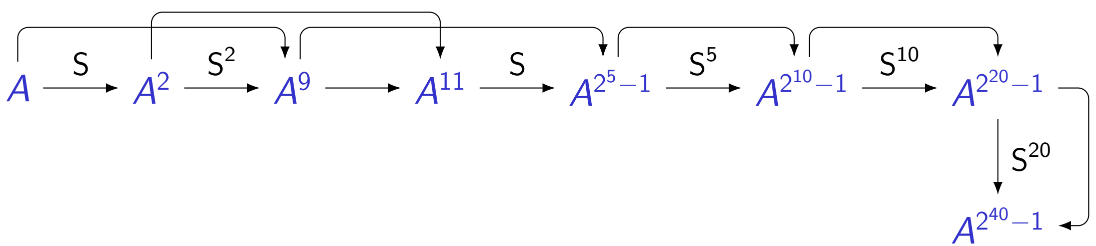
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

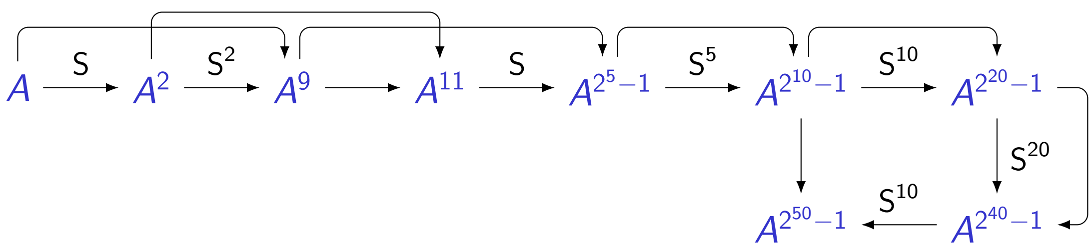
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

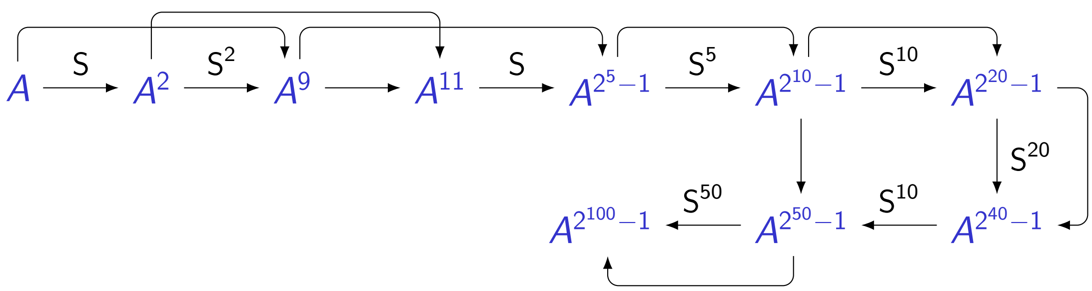
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

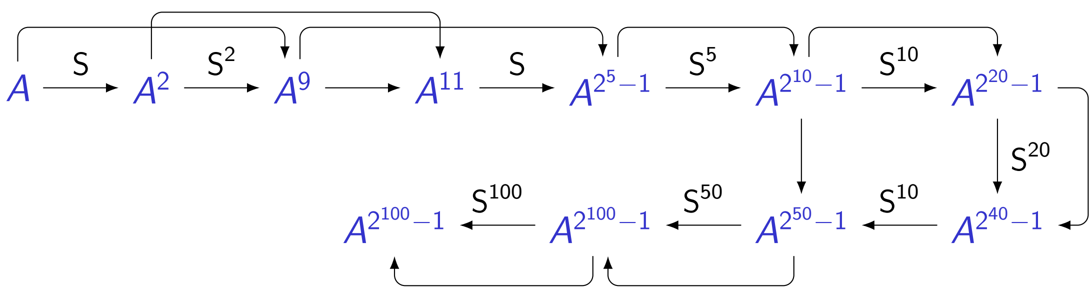
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

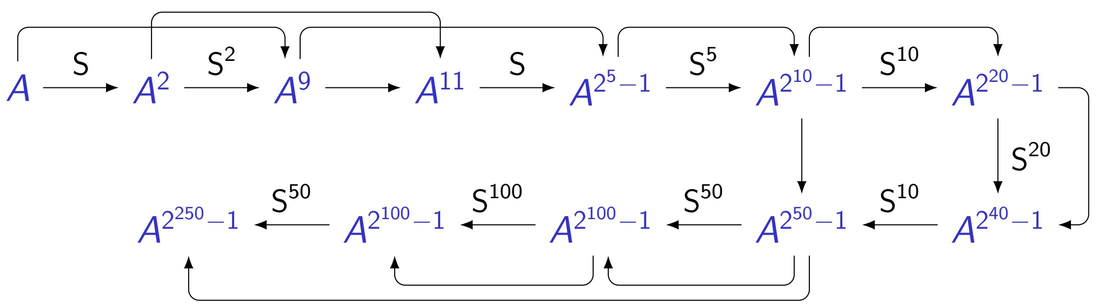
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

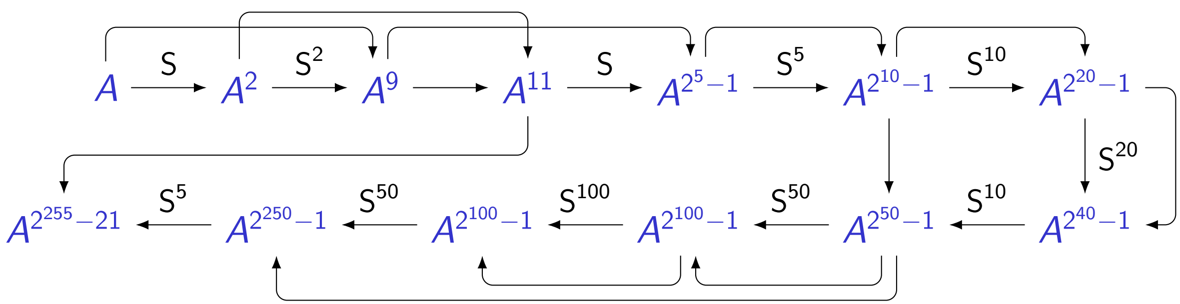
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} = U \bmod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} = 1 \pmod{P}$, whence $A^{P-2} = A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

- Let $A < M$ be an integer

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

- Let $A < M$ be an integer
  - represent A as the tuple $\overrightarrow{A} = (a_1, \ldots, a_k)$ with $a_i = A \bmod m_i = |A|_{m_i}$, for all i
    $\rightarrow$ that is the RNS representation of A in base $\mathcal{B}$

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

- Let $A < M$ be an integer
  - represent A as the tuple $\overrightarrow{A} = (a_1, \ldots, a_k)$ with $a_i = A \bmod m_i = |A|_{m_i}$, for all i
    $\rightarrow$ that is the RNS representation of A in base $\mathcal{B}$
  - given $\overrightarrow{A} = (a_1, \ldots, a_k)$, retrieve the unique corresponding integer $A \in \mathbb{Z}/M\mathbb{Z}$ using the Chinese remaindering theorem (CRT):

$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M$$

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

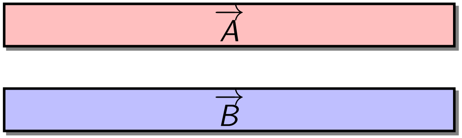
- Let $A < M$ be an integer
  - represent A as the tuple $\overrightarrow{A} = (a_1, \ldots, a_k)$ with $a_i = A \bmod m_i = |A|_{m_i}$, for all i
    → that is the RNS representation of A in base $\mathcal{B}$
  - given $\overrightarrow{A} = (a_1, \ldots, a_k)$, retrieve the unique corresponding integer $A \in \mathbb{Z}/M\mathbb{Z}$ using the Chinese remaindering theorem (CRT):

$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M$$

- If $P \leq M$, we can represent elements of $\mathbb{F}_P$ in RNS

RNS arithmetic

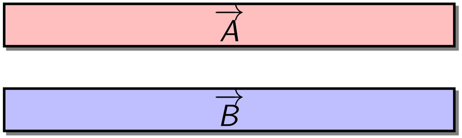
- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$



RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

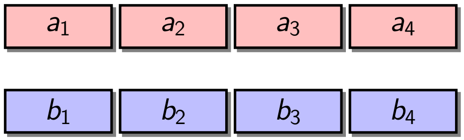
$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$



RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

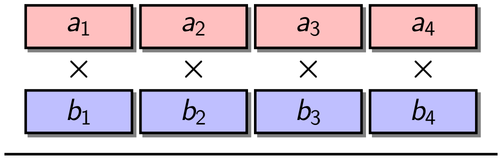
$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$



RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

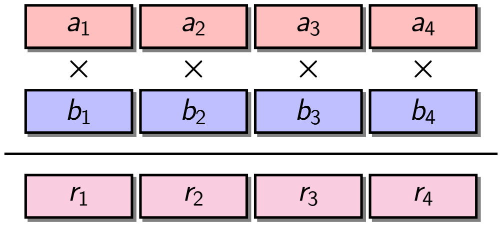
$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$



RNS arithmetic

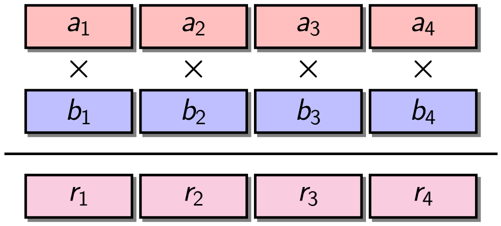
- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$



RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":
$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$
  - native parallelism: suited to SIMD instructions and hardware implementation



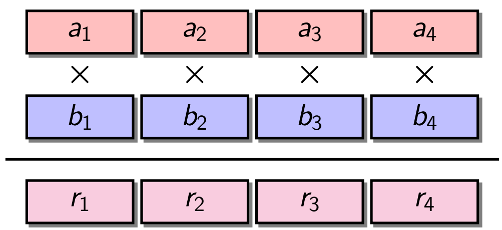
RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$

  - native parallelism: suited to SIMD instructions and hardware implementation

- Limitations:
  - operations are computed in $\mathbb{Z}/M\mathbb{Z}$: beware of overflows!
  - no simple way to compute divisons, modular reductions or comparisons



RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  ⇒ Need to approximate CRT reconstruction and reduce it modulo P

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:

$$A = \left| \sum_{i=1}^{k} \left| a_i \cdot M_i^{-1} \right|_{m_i} \cdot M_i \right|_M$$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:

$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$

with $0 \leq q < k$, whose actual value depends on A

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:
  $$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$
  with $0 \leq q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:
  $$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor$$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:
$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$
  with $0 \leq q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:
$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{m_i} \right\rfloor$$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:
  $$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$
  with $0 \le q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:
  $$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{m_i} \right\rfloor$$
  - approximate $m_i = 2^w - c_i$ by 2^w

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:
$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$
  with $0 \leq q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:
$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^w} \right\rfloor$$
  - approximate $m_i = 2^w - c_i$ by 2^w

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:
  $$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$
  with $0 \leq q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:
  $$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^w} \right\rfloor$$
  - approximate $m_i = 2^w - c_i$ by 2^w
  - use only the t most significant bits of $|a_i \cdot M_i^{-1}|_{m_i}$ to compute $\tilde{q}$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:

$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$

  with $0 \leq q < k$, whose actual value depends on A

- Compute $\tilde{q}$, approximation of q:

$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{\left\lfloor \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor}{2^t} \right\rfloor$$

  - approximate $m_i = 2^w - c_i$ by 2^w
  - use only the t most significant bits of $|a_i \cdot M_i^{-1}|_{m_i}$ to compute $\tilde{q}$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:
  $$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$
  with $0 \leq q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:
  $$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{\left\lfloor \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor}{2^t} + \varepsilon \right\rfloor$$
  - approximate $m_i = 2^w - c_i$ by 2^w
  - use only the t most significant bits of $|a_i \cdot M_i^{-1}|_{m_i}$ to compute $\tilde{q}$
  - add fixed corrective term $\left(\sum_i c_i + k(2^{w-t} - 1)\right)/2^w < \varepsilon < 1$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:
$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$
  with $0 \le q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:
$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{\left\lfloor \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor}{2^t} + \varepsilon \right\rfloor = \tilde{q}$$
  - approximate $m_i = 2^w - c_i$ by 2^w
  - use only the t most significant bits of $|a_i \cdot M_i^{-1}|_{m_i}$ to compute $\tilde{q}$
  - add fixed corrective term $\left(\sum_i c_i + k(2^{w-t} - 1)\right)/2^w < \varepsilon < 1$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:

$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$

  with $0 \leq q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:

$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{\left\lfloor \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor}{2^t} + \varepsilon \right\rfloor = \tilde{q}$$

  - approximate $m_i = 2^w - c_i$ by 2^w
  - use only the t most significant bits of $|a_i \cdot M_i^{-1}|_{m_i}$ to compute $\tilde{q}$
  - add fixed corrective term $\left(\sum_i c_i + k(2^{w-t} - 1)\right)/2^w < \varepsilon < 1$
- If $0 \leq A < (1 - \varepsilon)M$, then $\tilde{q} = q$ and

$$A = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - \tilde{q}M$$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:

$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$

  with $0 \leq q < k$, whose actual value depends on A

- Compute $\tilde{q}$, approximation of q:

$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{\left\lfloor \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor}{2^t} + \varepsilon \right\rfloor = \tilde{q}$$

  - approximate $m_i = 2^w - c_i$ by 2^w
  - use only the t most significant bits of $|a_i \cdot M_i^{-1}|_{m_i}$ to compute $\tilde{q}$
  - add fixed corrective term $\left(\sum_i c_i + k(2^{w-t} - 1)\right) / 2^w < \varepsilon < 1$

- If $0 \leq A < (1 - \varepsilon)M$, then $\tilde{q} = q$ and

$$A \bmod P = \left(\left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - \tilde{q}M \right) \bmod P$$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:

$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$

  with $0 \le q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:

$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{\left\lfloor \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor}{2^t} + \varepsilon \right\rfloor = \tilde{q}$$

  - approximate $m_i = 2^w - c_i$ by 2^w
  - use only the t most significant bits of $|a_i \cdot M_i^{-1}|_{m_i}$ to compute $\tilde{q}$
  - add fixed corrective term $\left(\sum_i c_i + k(2^{w-t} - 1)\right) / 2^w < \varepsilon < 1$
- If $0 \le A < (1 - \varepsilon)M$, then $\tilde{q} = q$ and

$$A \bmod P = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P$$

RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS
  $\Rightarrow$ Need to approximate CRT reconstruction and reduce it modulo P
- From the CRT:
  $$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M = \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$
  with $0 \leq q < k$, whose actual value depends on A
- Compute $\tilde{q}$, approximation of q:
  $$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{\left\lfloor \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor}{2^t} + \varepsilon \right\rfloor = \tilde{q}$$
  - approximate $m_i = 2^w - c_i$ by 2^w
  - use only the t most significant bits of $|a_i \cdot M_i^{-1}|_{m_i}$ to compute $\tilde{q}$
  - add fixed corrective term $\left(\sum_i c_i + k(2^{w-t} - 1)\right)/2^w < \varepsilon < 1$
- If $0 \leq A < (1 - \varepsilon)M$, then $\tilde{q} = q$ and
  $$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P}$$

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q} M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$**:**

($\forall i$**)** $z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$

($\forall i$**)** $\tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$

$\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$

($\forall i$**)** $r_i \leftarrow 0$

for $j \leftarrow 1$ **to** k**:**

($\forall i$**)** $r_i \leftarrow \left| r_i + z_j \cdot \left| |M_j|_P \right|_{m_i} \right|_{m_i}$

($\forall i$**)** $r_i \leftarrow \left| r_i - \left| |\tilde{q} M|_P \right|_{m_i} \right|_{m_i}$

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$**:**

$(\forall i)\ z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$

$(\forall i)\ \tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$

$\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$

$(\forall i)\ r_i \leftarrow 0$

for $j \leftarrow 1$ **to** k**:**

$(\forall i)\ r_i \leftarrow \left| r_i + z_j \cdot ||M_j|_P|_{m_i} \right|_{m_i}$

$(\forall i)\ r_i \leftarrow \left| r_i - ||\tilde{q}M|_P|_{m_i} \right|_{m_i}$

- Precomputations:

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q} M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$**:**

($\forall i$**)** $z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$

($\forall i$**)** $\tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$

$\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$

($\forall i$**)** $r_i \leftarrow 0$

for $j \leftarrow 1$ **to** k**:**

($\forall i$**)** $r_i \leftarrow \left| r_i + z_j \cdot ||M_j|_P|_{m_i} \right|_{m_i}$

($\forall i$**)** $r_i \leftarrow \left| r_i - ||\tilde{q} M|_P|_{m_i} \right|_{m_i}$

- Precomputations:
  - for all $i \in \{1, \ldots, k\}$, $|M_i^{-1}|_{m_i}$ (k words)

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$**:**

$(\forall i)$ $z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$

$(\forall i)$ $\tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$

$\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$

$(\forall i)$ $r_i \leftarrow 0$

for $j \leftarrow 1$ **to** k**:**

$(\forall i)$ $r_i \leftarrow \left| r_i + z_j \cdot ||M_j|_P|_{m_i} \right|_{m_i}$

$(\forall i)$ $r_i \leftarrow \left| r_i - ||\tilde{q}M|_P|_{m_i} \right|_{m_i}$

- Precomputations:
  - for all $i \in \{1, \ldots, k\}$, $\overrightarrow{|M_i^{-1}|_{m_i}}$ (k words)
  - for all $j \in \{1, \ldots, k\}$, $\overrightarrow{|M_j|_P}$ (k^2 words)

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$:

$(\forall i)$ $z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$

$(\forall i)$ $\tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$

$\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$

$(\forall i)$ $r_i \leftarrow 0$

for $j \leftarrow 1$ **to** k:

  $(\forall i)$ $r_i \leftarrow \left| r_i + z_j \cdot ||M_j|_P|_{m_i} \right|_{m_i}$

$(\forall i)$ $r_i \leftarrow \left| r_i - ||\tilde{q}M|_P|_{m_i} \right|_{m_i}$

- Precomputations:
  - for all $i \in \{1, \ldots, k\}$, $|M_i^{-1}|_{m_i}$ (k words)
  - for all $j \in \{1, \ldots, k\}$, $\overrightarrow{|M_j|_P}$ (k^2 words)
  - for all $\tilde{q} \in \{1, \ldots, k-1\}$, $\overrightarrow{|\tilde{q}M|_P}$ (k^2 words)

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$**:**
- $(\forall i)\ z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$
- $(\forall i)\ \tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$
- $\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$
- $(\forall i)\ r_i \leftarrow 0$
- **for** $j \leftarrow 1$ **to** k**:**
  - $(\forall i)\ r_i \leftarrow \left| r_i + z_j \cdot ||M_j|_P|_{m_i} \right|_{m_i}$
- $(\forall i)\ r_i \leftarrow \left| r_i - ||\tilde{q}M|_P|_{m_i} \right|_{m_i}$

- Precomputations:
  - for all $i \in \{1, \ldots, k\}$, $|M_i^{-1}|_{m_i}$ (k words)
  - for all $j \in \{1, \ldots, k\}$, $\overrightarrow{|M_j|_P}$ (k^2 words)
  - for all $\tilde{q} \in \{1, \ldots, k-1\}$, $\overrightarrow{|\tilde{q}M|_P}$ (k^2 words)
- Cost:

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q} M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$**:**

$(\forall i)$ $z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$

$(\forall i)$ $\tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$

$\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$

$(\forall i)$ $r_i \leftarrow 0$

for $j \leftarrow 1$ **to** k**:**

$(\forall i)$ $r_i \leftarrow \left| r_i + z_j \cdot ||M_j|_P|_{m_i} \right|_{m_i}$

$(\forall i)$ $r_i \leftarrow \left| r_i - ||\tilde{q} M|_P|_{m_i} \right|_{m_i}$

- Precomputations:
  - for all $i \in \{1, \ldots, k\}$, $|M_i^{-1}|_{m_i}$ (k words)
  - for all $j \in \{1, \ldots, k\}$, $\overrightarrow{|M_j|_P}$ (k^2 words)
  - for all $\tilde{q} \in \{1, \ldots, k-1\}$, $\overrightarrow{|\tilde{q} M|_P}$ (k^2 words)
- Cost: k mults

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$**:**
- $(\forall i)$ $z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$
- $(\forall i)$ $\tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$
- $\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$
- $(\forall i)$ $r_i \leftarrow 0$
- **for** $j \leftarrow 1$ **to** k**:**
  - $(\forall i)$ $r_i \leftarrow \left| r_i + z_j \cdot ||M_j|_P|_{m_i} \right|_{m_i}$
- $(\forall i)$ $r_i \leftarrow \left| r_i - ||\tilde{q}M|_P|_{m_i} \right|_{m_i}$

- Precomputations:
  - for all $i \in \{1, \ldots, k\}$, $|M_i^{-1}|_{m_i}$ (k words)
  - for all $j \in \{1, \ldots, k\}$, $\overrightarrow{|M_j|_P}$ (k^2 words)
  - for all $\tilde{q} \in \{1, \ldots, k-1\}$, $\overrightarrow{|\tilde{q}M|_P}$ (k^2 words)
- Cost: k mults + k^2 mults

RNS modular reduction

$$A \bmod P \equiv \left(\sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q} M|_P \pmod{P}$$

function reduce-mod-$P(\overrightarrow{A})$**:**

$(\forall i)$ $z_i \leftarrow \left| a_i \cdot |M_i^{-1}|_{m_i} \right|_{m_i}$

$(\forall i)$ $\tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor$

$\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor$

$(\forall i)$ $r_i \leftarrow 0$

for $j \leftarrow 1$ **to** k**:**

$(\forall i)$ $r_i \leftarrow \left| r_i + z_j \cdot ||M_j|_P|_{m_i} \right|_{m_i}$

$(\forall i)$ $r_i \leftarrow \left| r_i - ||\tilde{q} M|_P|_{m_i} \right|_{m_i}$

- Precomputations:
  - for all $i \in \{1, \ldots, k\}$, $|M_i^{-1}|_{m_i}$ (k words)
  - for all $j \in \{1, \ldots, k\}$, $\overrightarrow{|M_j|_P}$ (k^2 words)
  - for all $\tilde{q} \in \{1, \ldots, k-1\}$, $\overrightarrow{|\tilde{q} M|_P}$ (k^2 words)
- Cost: k mults + k^2 mults → quadratic complexity

RNS Montgomery reduction

- Requires two RNS bases $\mathcal{B}_\alpha = (m_{\alpha,1}, \ldots, m_{\alpha,k})$ and $\mathcal{B}_\beta = (m_{\beta,1}, \ldots, m_{\beta,k})$ such that $P < M_\alpha$, $P < M_\beta$, and $\gcd(M_\alpha, M_\beta) = 1$

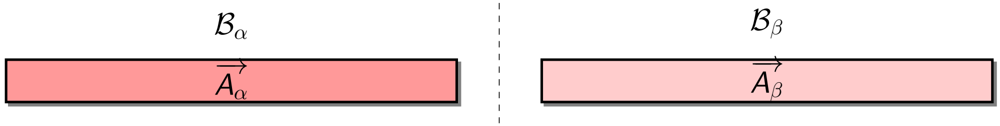
RNS Montgomery reduction

- Requires two RNS bases $\mathcal{B}_\alpha = (m_{\alpha,1}, \ldots, m_{\alpha,k})$ and $\mathcal{B}_\beta = (m_{\beta,1}, \ldots, m_{\beta,k})$ such that $P < M_\alpha$, $P < M_\beta$, and $\gcd(M_\alpha, M_\beta) = 1$
- RNS base extension algorithm (BE) [Kawamura *et al.*, 2000]
  - given $\overrightarrow{X_\alpha}$ in base $\mathcal{B}_\alpha$, $\mathrm{BE}(\overrightarrow{X_\alpha}, \mathcal{B}_\alpha, \mathcal{B}_\beta)$ computes $\overrightarrow{X_\beta}$, the repr. of X in base $\mathcal{B}_\beta$
  - similarly, $\mathrm{BE}(\overrightarrow{X_\beta}, \mathcal{B}_\beta, \mathcal{B}_\alpha)$ computes $\overrightarrow{X_\alpha}$ in base $\mathcal{B}_\alpha$

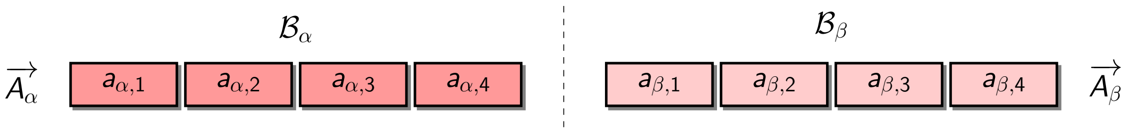
RNS Montgomery reduction

- Requires two RNS bases $\mathcal{B}_\alpha = (m_{\alpha,1}, \ldots, m_{\alpha,k})$ and $\mathcal{B}_\beta = (m_{\beta,1}, \ldots, m_{\beta,k})$ such that $P < M_\alpha$, $P < M_\beta$, and $\gcd(M_\alpha, M_\beta) = 1$
- RNS base extension algorithm (BE) [Kawamura *et al.*, 2000]
  - given $\overrightarrow{X_\alpha}$ in base $\mathcal{B}_\alpha$, $\mathrm{BE}(\overrightarrow{X_\alpha}, \mathcal{B}_\alpha, \mathcal{B}_\beta)$ computes $\overrightarrow{X_\beta}$, the repr. of X in base $\mathcal{B}_\beta$
  - similarly, $\mathrm{BE}(\overrightarrow{X_\beta}, \mathcal{B}_\beta, \mathcal{B}_\alpha)$ computes $\overrightarrow{X_\alpha}$ in base $\mathcal{B}_\alpha$
  - similar to RNS modular reduction $\rightarrow$ $O(k^2)$ complexity

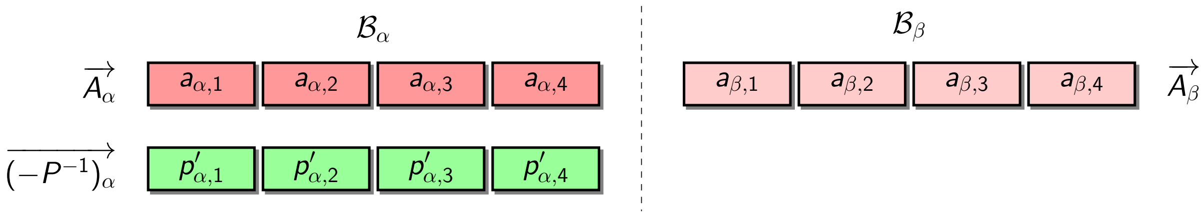
RNS Montgomery reduction



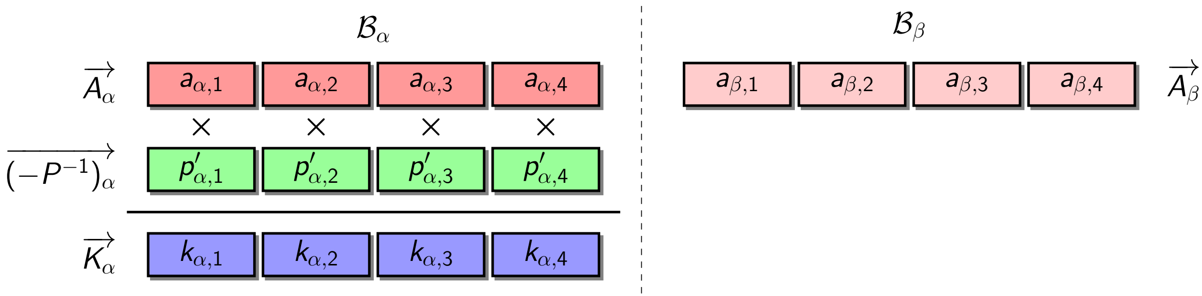
RNS Montgomery reduction



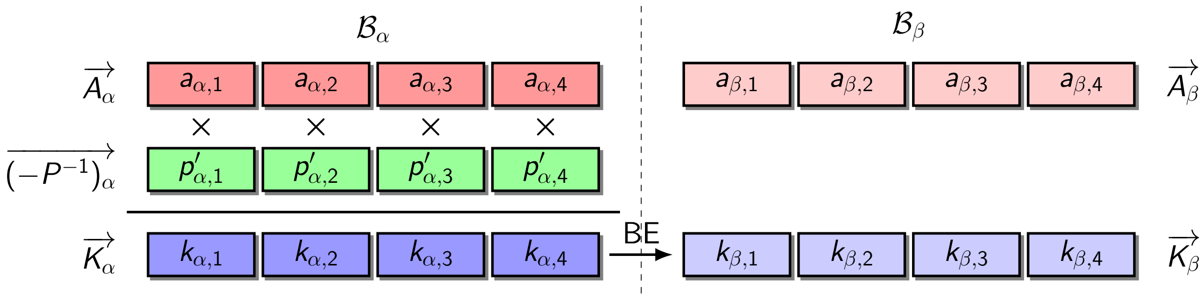
RNS Montgomery reduction



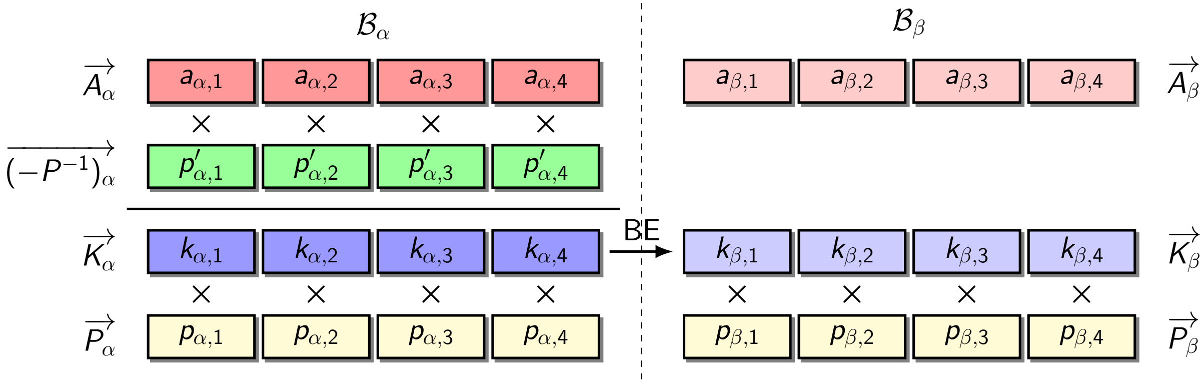
RNS Montgomery reduction



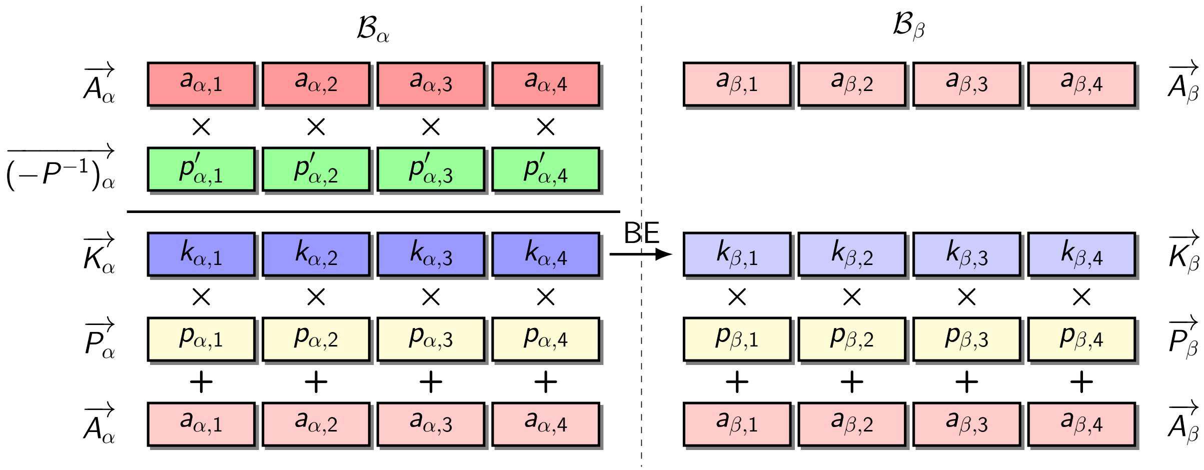
RNS Montgomery reduction



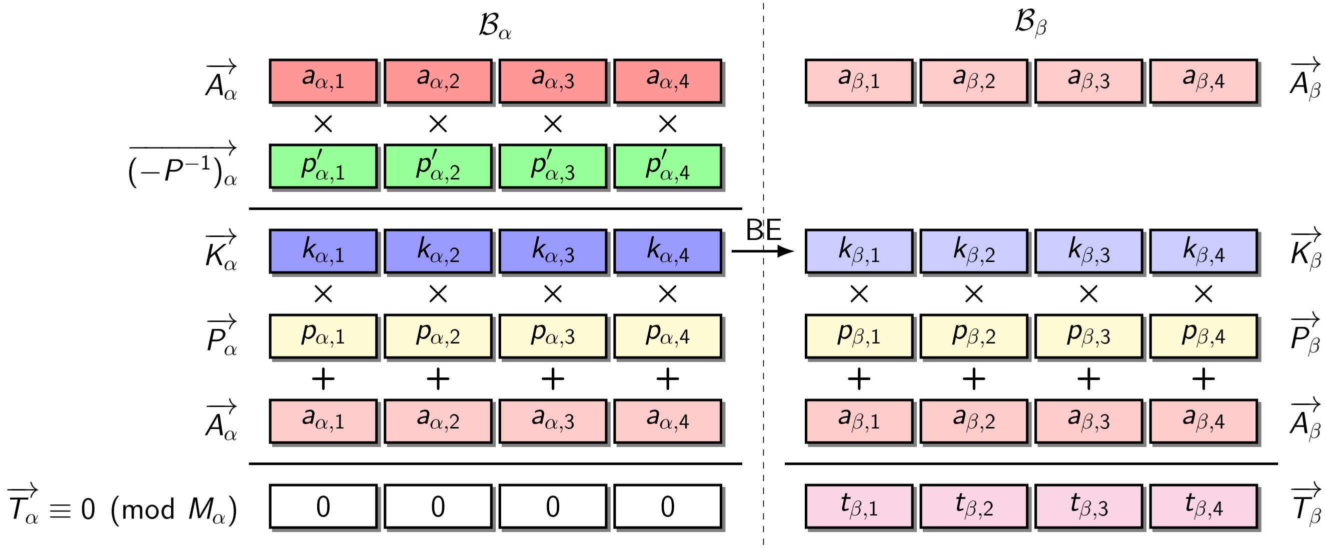
RNS Montgomery reduction



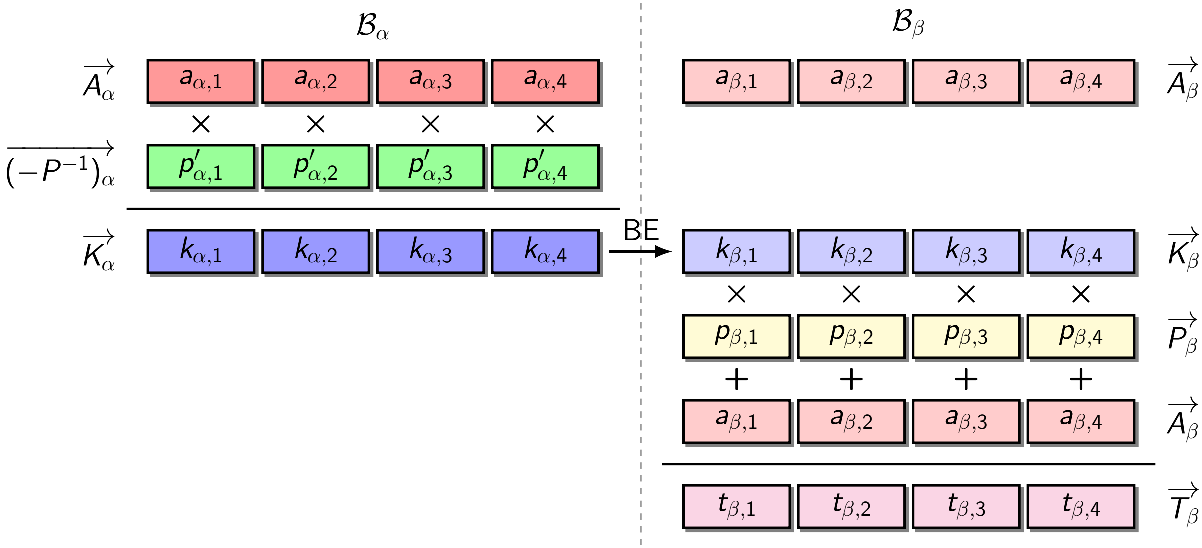
RNS Montgomery reduction



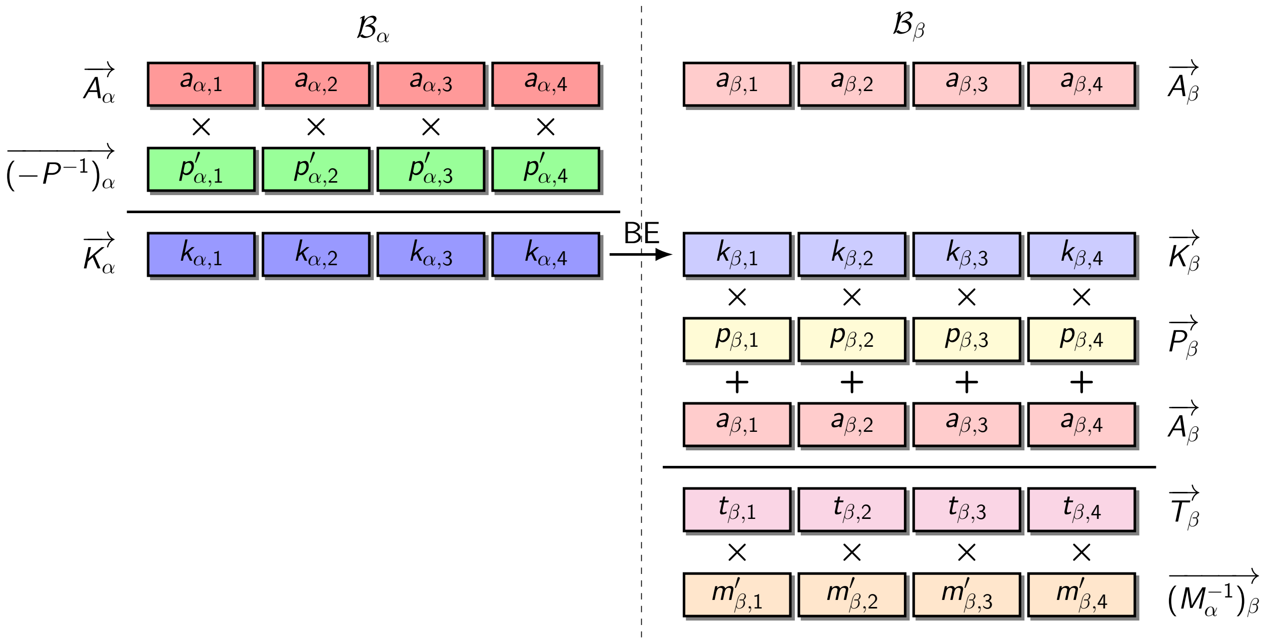
RNS Montgomery reduction



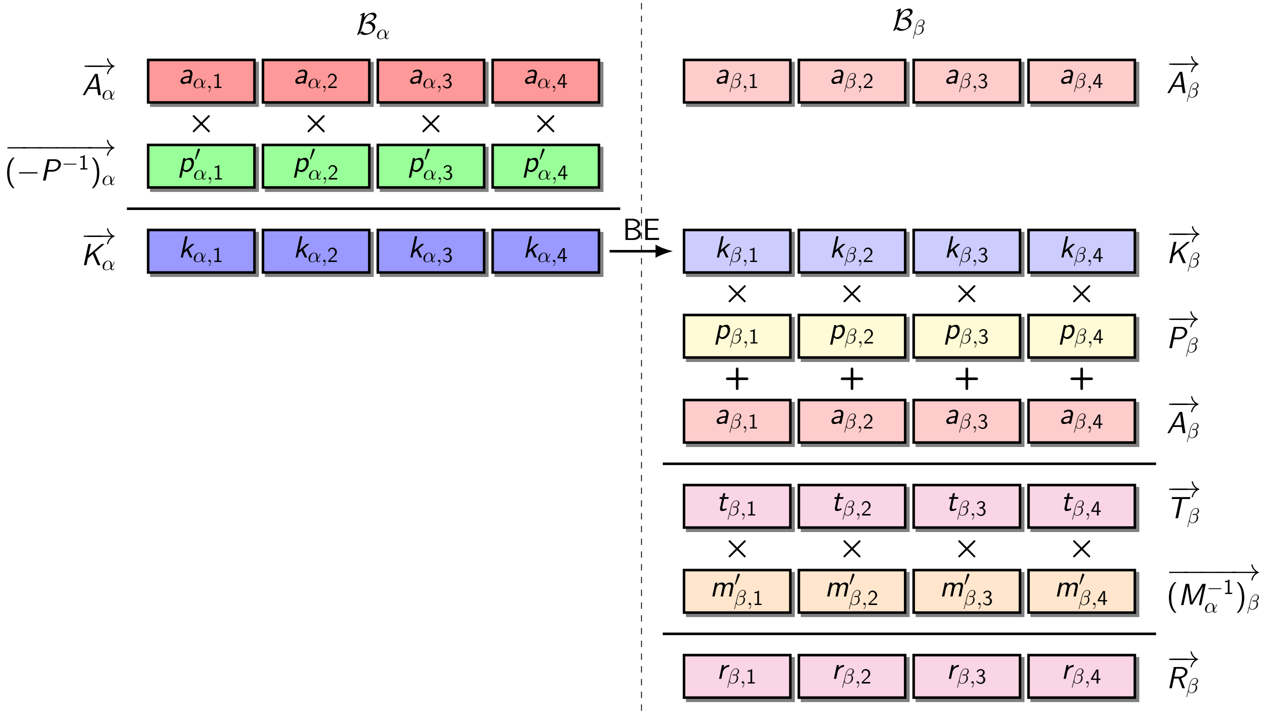
RNS Montgomery reduction



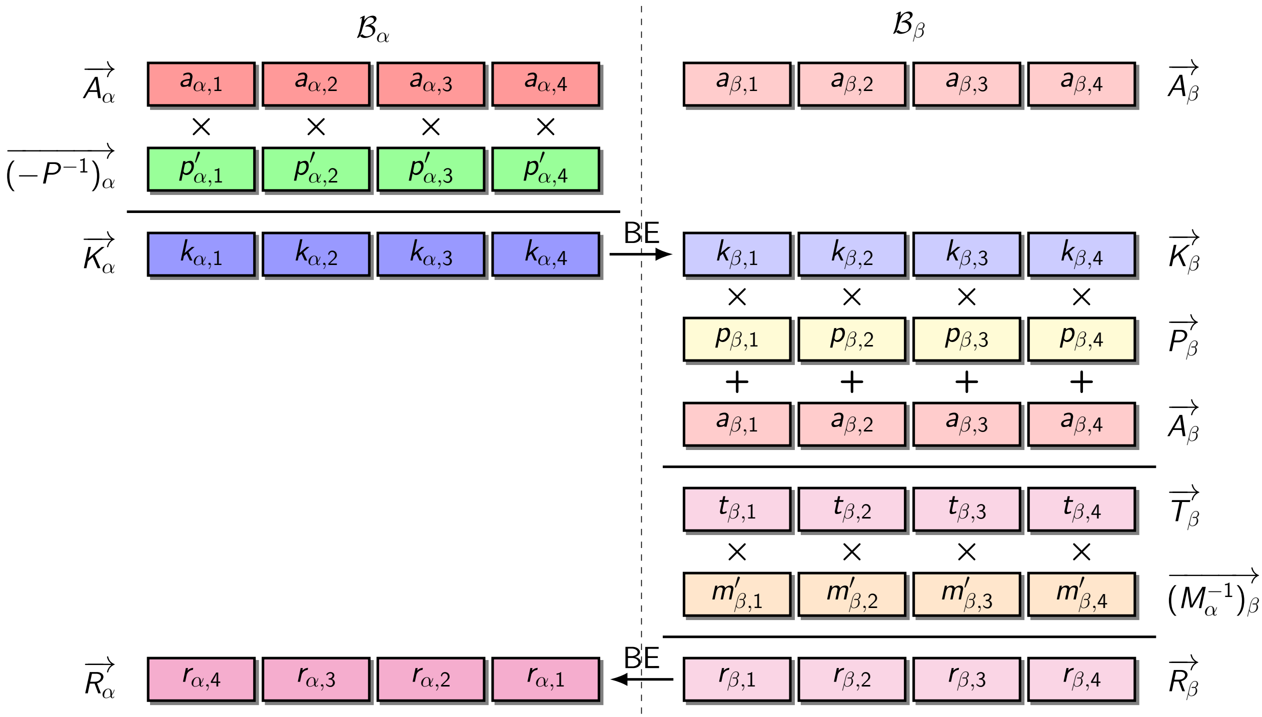
RNS Montgomery reduction



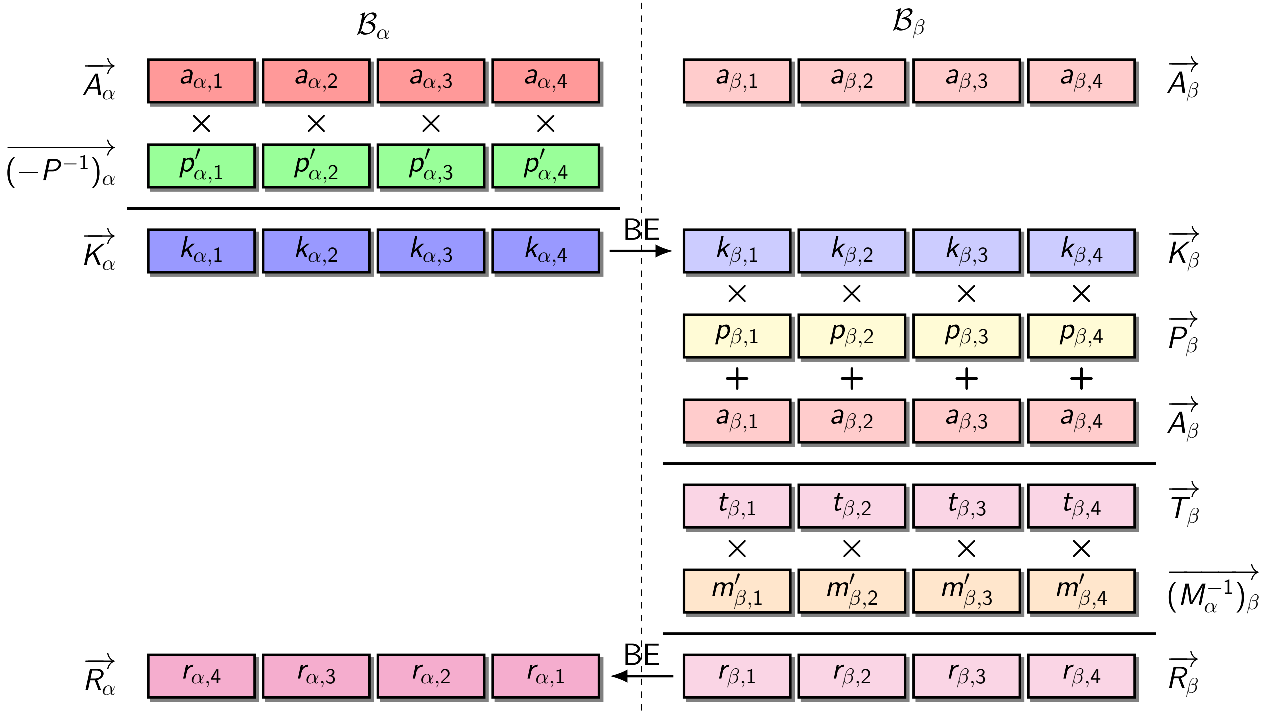
RNS Montgomery reduction



RNS Montgomery reduction

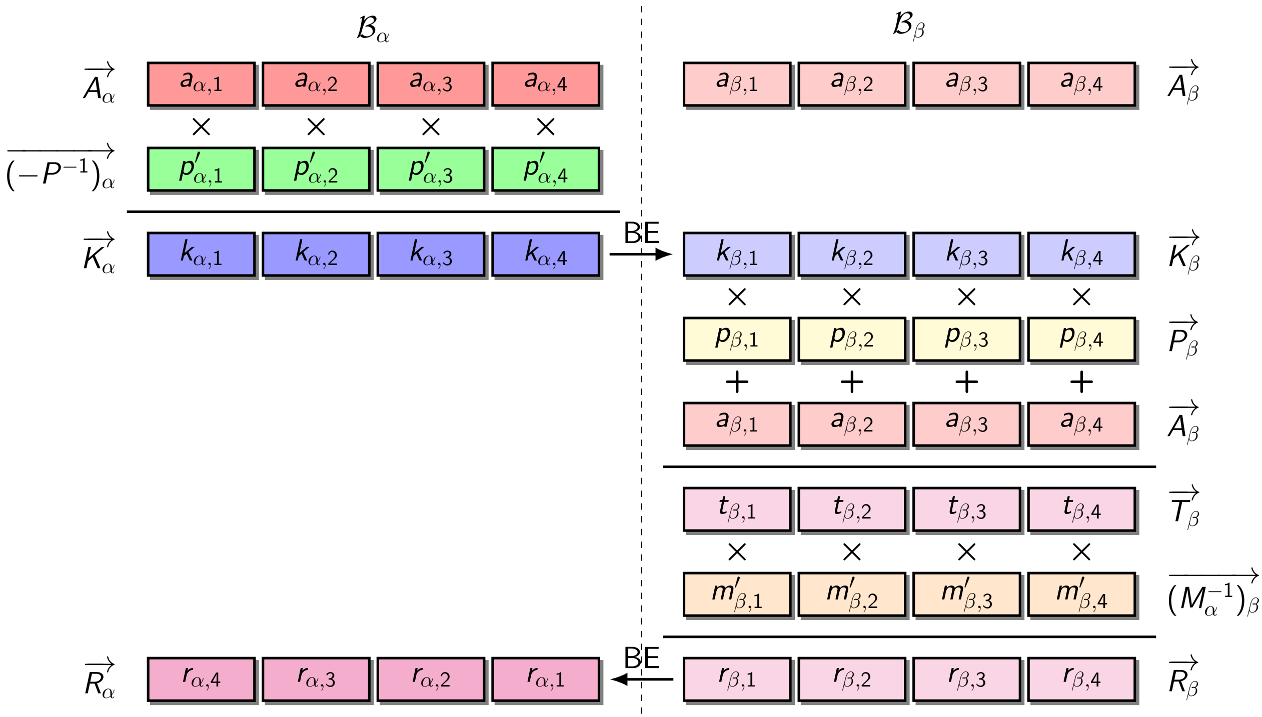


RNS Montgomery reduction



- Result is $(\overrightarrow{R_\alpha}, \overrightarrow{R_\beta}) \equiv (A \cdot M_\alpha^{-1}) \pmod{P}$

RNS Montgomery reduction



- Result is $(\overrightarrow{R_\alpha}, \overrightarrow{R_\beta}) \equiv (A \cdot M_\alpha^{-1}) \pmod{P}$
- See recent results on this topic by Bigou and Tisserand

Outline

Software considerations

- In fact, pretty much has already been said...

Software considerations

- In fact, pretty much has already been said...
- Know your favorite CPU's instruction set by heart!

Software considerations

- In fact, pretty much has already been said...
- Know your favorite CPU's instruction set by heart!
  - what's `PCLMULQDQ`? how many 32-bit words can fit in a NEON register?

Software considerations

- In fact, pretty much has already been said...
- Know your favorite CPU's instruction set by heart!
  - what's `PCLMULQDQ`? how many 32-bit words can fit in a NEON register?
  - sometimes, floating-point arithmetic is faster than integer arithmetic

Software considerations

- In fact, pretty much has already been said...
- Know your favorite CPU's instruction set by heart!
  - what's `PCLMULQDQ`? how many 32-bit words can fit in a NEON register?
  - sometimes, floating-point arithmetic is faster than integer arithmetic
  - download `http://www.agner.org/optimize/instruction_tables.pdf` to find all instruction latencies and thoughputs for Intel and AMD CPUs

Software considerations

- In fact, pretty much has already been said...
- Know your favorite CPU's instruction set by heart!
  - what's `PCLMULQDQ`? how many 32-bit words can fit in a NEON register?
  - sometimes, floating-point arithmetic is faster than integer arithmetic
  - download `http://www.agner.org/optimize/instruction_tables.pdf` to find all instruction latencies and thoughputs for Intel and AMD CPUs
- Beware of fancy CPU features!
  - avoid secret-dependent memory access patterns (cache attacks)
  - avoid secret-dependent conditional branches (timing, branch predictor attacks)

Software considerations

- In fact, pretty much has already been said...
- Know your favorite CPU's instruction set by heart!
  - what's `PCLMULQDQ`? how many 32-bit words can fit in a NEON register?
  - sometimes, floating-point arithmetic is faster than integer arithmetic
  - download `http://www.agner.org/optimize/instruction_tables.pdf` to find all instruction latencies and thoughputs for Intel and AMD CPUs
- Beware of fancy CPU features!
  - avoid secret-dependent memory access patterns (cache attacks)
  - avoid secret-dependent conditional branches (timing, branch predictor attacks)
- Have a look at existing libraries (from OpenSSL to MPFQ):
  - plenty of great ideas in there!
  - you might even find bugs and vulnerabilities

Software considerations

- In fact, pretty much has already been said...
- Know your favorite CPU's instruction set by heart!
  - what's `PCLMULQDQ`? how many 32-bit words can fit in a NEON register?
  - sometimes, floating-point arithmetic is faster than integer arithmetic
  - download `http://www.agner.org/optimize/instruction_tables.pdf` to find all instruction latencies and thoughputs for Intel and AMD CPUs
- Beware of fancy CPU features!
  - avoid secret-dependent memory access patterns (cache attacks)
  - avoid secret-dependent conditional branches (timing, branch predictor attacks)
- Have a look at existing libraries (from OpenSSL to MPFQ):
  - plenty of great ideas in there!
  - you might even find bugs and vulnerabilities
- Read, code, hack, experiment!

Outline

I. Scalar multiplication

II. Elliptic curve arithmetic

III. Finite field arithmetic

IV. Software considerations

V. Notions of hardware design

Describing hardware circuits

- We surely do **NOT** want to
  - program millions of logic cells / transistors by hand
  - connect their inputs and outputs by hand

Describing hardware circuits

- We surely do **NOT** want to
  - program millions of logic cells / transistors by hand
  - connect their inputs and outputs by hand
- Design circuits using a hardware description language (HDL)
  - VHDL, Verilog, etc.
  - usually independent from the target technology

Describing hardware circuits

- We surely do **NOT** want to
  - program millions of logic cells / transistors by hand
  - connect their inputs and outputs by hand
- Design circuits using a hardware description language (HDL)
  - VHDL, Verilog, etc.
  - usually independent from the target technology
- HDL paradigm completely different from software programming languages
  - used to describe concurrent systems: unable to express sequentiality
  - structural and hierarchical description of the circuit

A half-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin


end architecture;
```

$$x + y = s + 2co$$

A half-adder in VHDL

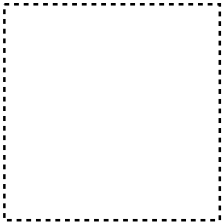
```
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin


end architecture;
```

$$x + y = s + 2co$$

A half-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin


end architecture;
```

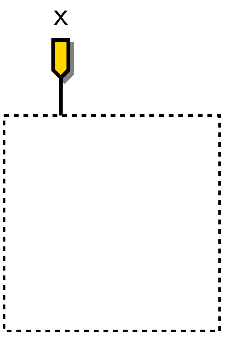
$$x + y = s + 2co$$



A half-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin


end architecture;
```

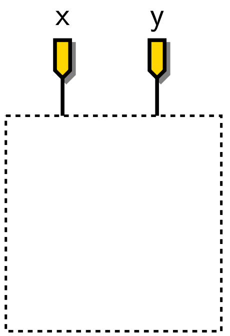
$x + y = s + 2co$



A half-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin


end architecture;
```

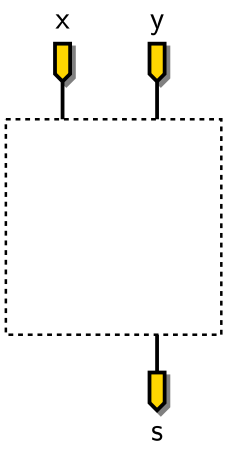
$$x + y = s + 2co$$



A half-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin


end architecture;
```

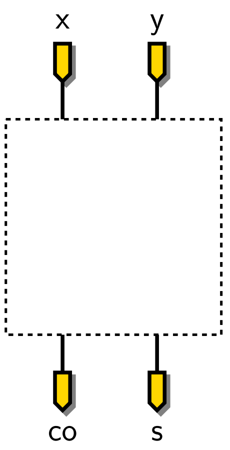
$x + y = s + 2co$



A half-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin


end architecture;
```

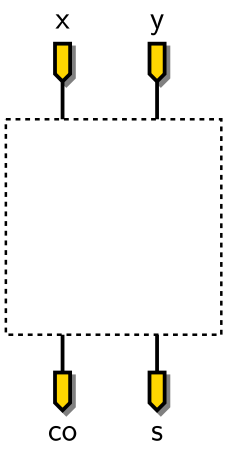
$$x + y = s + 2co$$



A half-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin


end architecture;
```

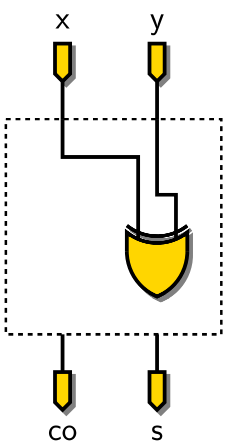
$$x + y = s + 2co$$



A half-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin
  s  <= x xor y;

end architecture;
```

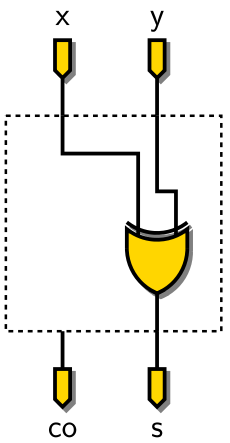
$$x + y = s + 2co$$



A half-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in   std_logic;
         y  : in   std_logic;
         s  : out  std_logic;
         co : out  std_logic );
end entity;

architecture arch of ha is
begin
  s  <= x xor y;

end architecture;
```

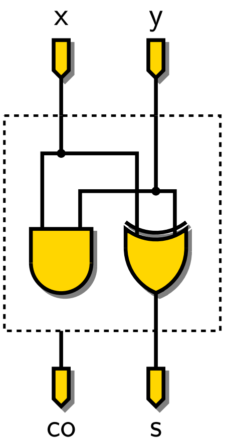
$$x + y = s + 2co$$



A half-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin
  s  <= x xor y;
  co <= x and y;
end architecture;
```

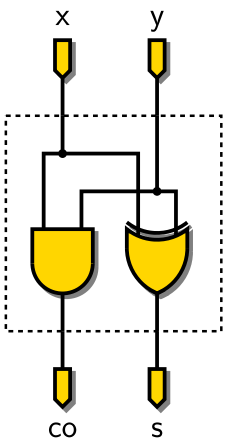
$$x + y = s + 2co$$



A half-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin
  s  <= x xor y;
  co <= x and y;
end architecture;
```

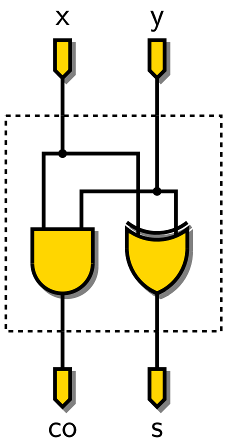
$$x + y = s + 2co$$



A half-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ha is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of ha is
begin
  s  <= x xor y;
  co <= x and y;
end architecture;
```

$$x + y = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is






begin





end architecture;
```

$$x + y + ci = s + 2co$$

A full-adder in VHDL

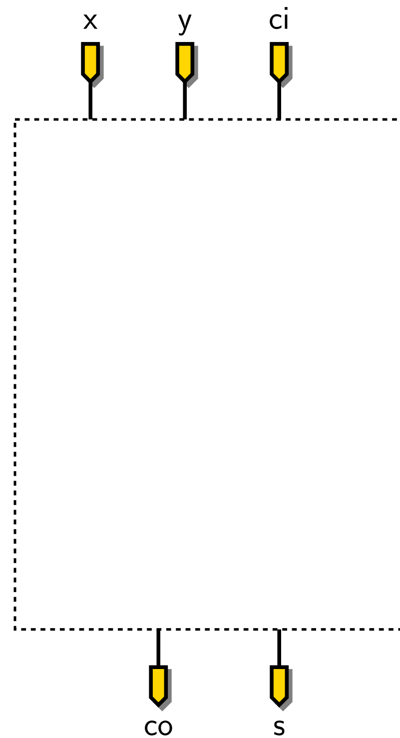
```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is







begin





end architecture;
```

$$x + y + ci = s + 2co$$

A full-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is







begin






end architecture;
```

$$x + y + ci = s + 2co$$



A full-adder in VHDL

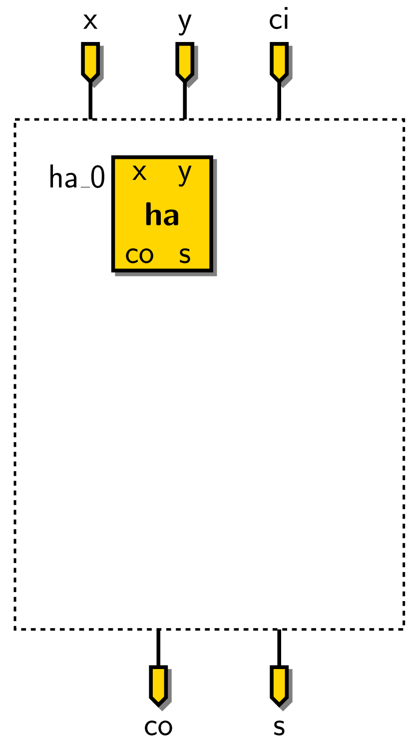
```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is







begin





end architecture;
```

$$x + y + ci = s + 2co$$

A full-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is







begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );



end architecture;
```

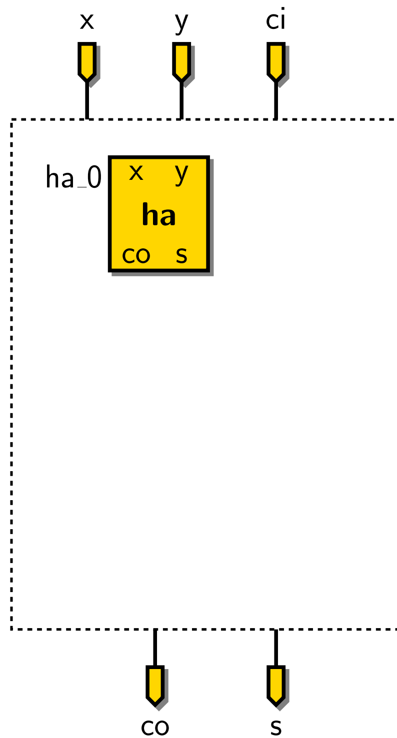
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;



begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );



end architecture;
```

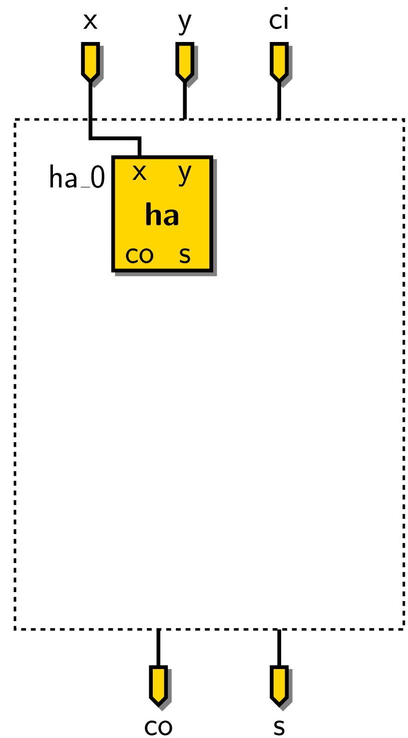
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x   : in  std_logic;
         y   : in  std_logic;
         ci  : in  std_logic;
         s   : out std_logic;
         co  : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;



begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );



end architecture;
```

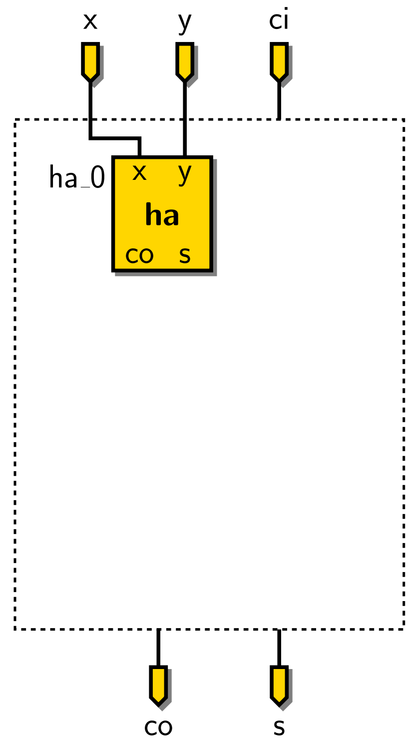
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x   : in   std_logic;
         y   : in   std_logic;
         ci  : in   std_logic;
         s   : out  std_logic;
         co  : out  std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in   std_logic; y   : in   std_logic;
           s : out  std_logic; co  : out  std_logic );
  end component;



begin
  ha_0 : ha port map ( x => x,    y  => y,
                       s => s_0,  co => co_0 );



end architecture;
```

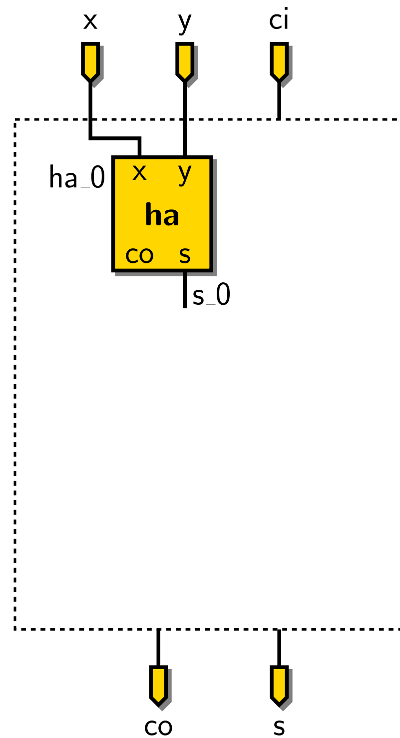
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x   : in   std_logic;
         y   : in   std_logic;
         ci  : in   std_logic;
         s   : out  std_logic;
         co  : out  std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in   std_logic; y   : in   std_logic;
           s : out  std_logic; co  : out  std_logic );
  end component;
  signal s_0   : std_logic;


begin
  ha_0 : ha port map ( x => x,    y  => y,
                       s => s_0,  co => co_0 );



end architecture;
```

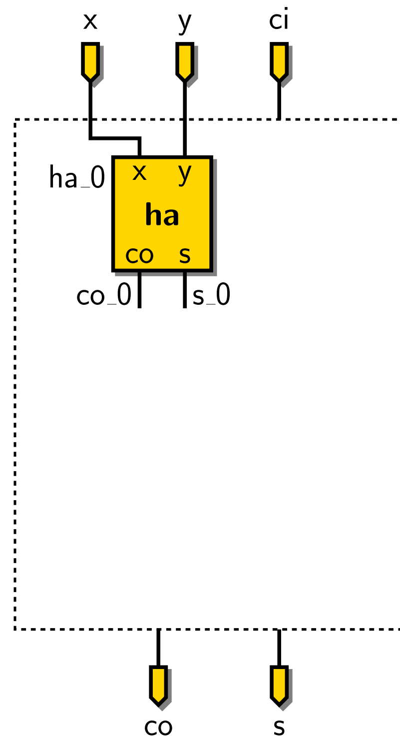
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;

begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );



end architecture;
```

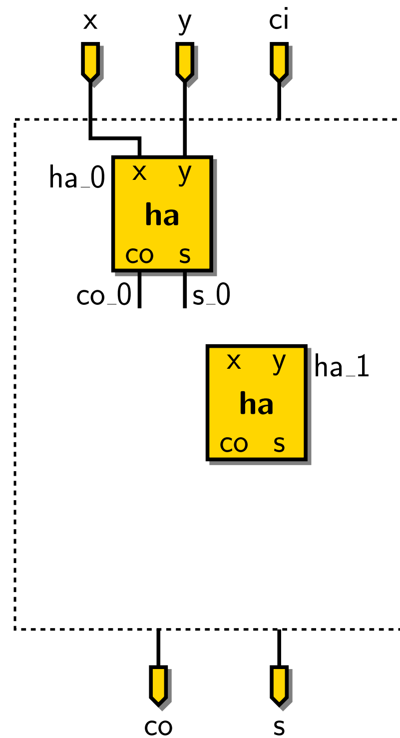
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;

begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );
  ha_1 : ha port map ( x => s_0, y  => ci,
                       s => s,   co => co_1 );

end architecture;
```

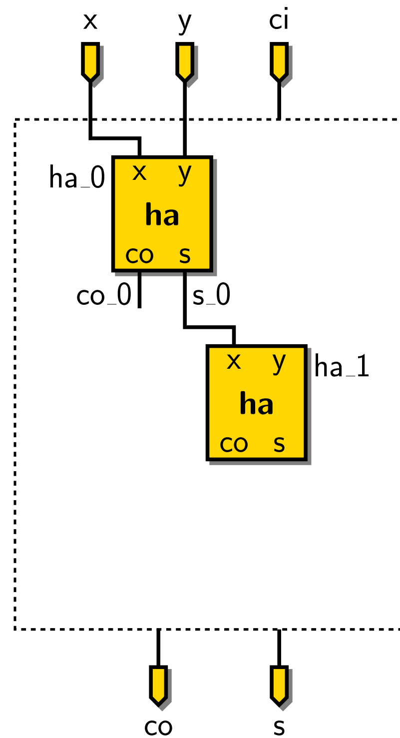
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x   : in   std_logic;
         y   : in   std_logic;
         ci  : in   std_logic;
         s   : out  std_logic;
         co  : out  std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in   std_logic; y  : in   std_logic;
           s : out  std_logic; co : out  std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;

begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );
  ha_1 : ha port map ( x => s_0, y  => ci,
                       s => s,   co => co_1 );

end architecture;
```

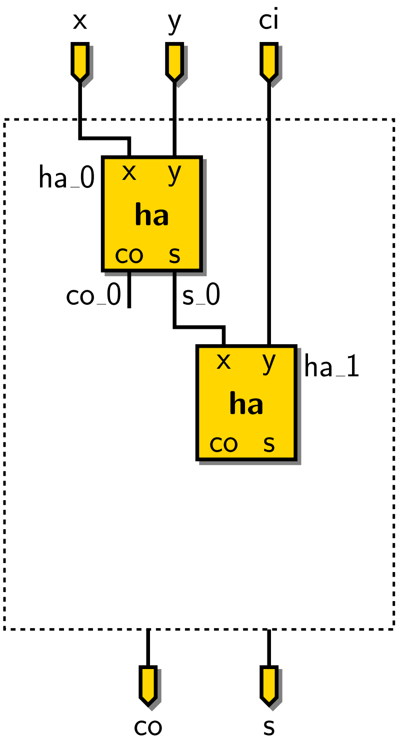
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;

begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );
  ha_1 : ha port map ( x => s_0, y  => ci,
                       s => s,   co => co_1 );

end architecture;
```

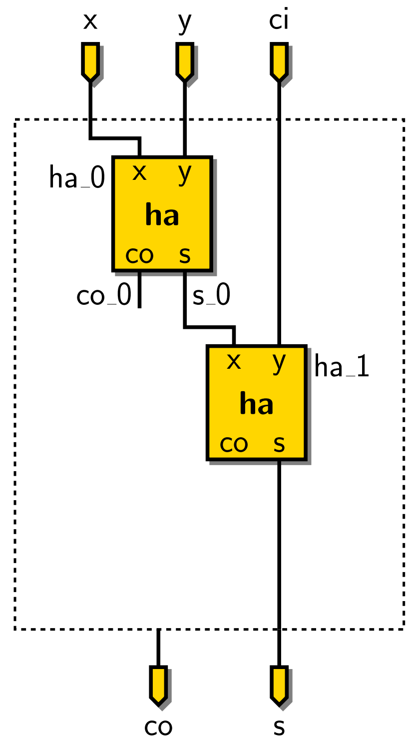
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;

begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );
  ha_1 : ha port map ( x => s_0, y  => ci,
                       s => s,   co => co_1 );

end architecture;
```

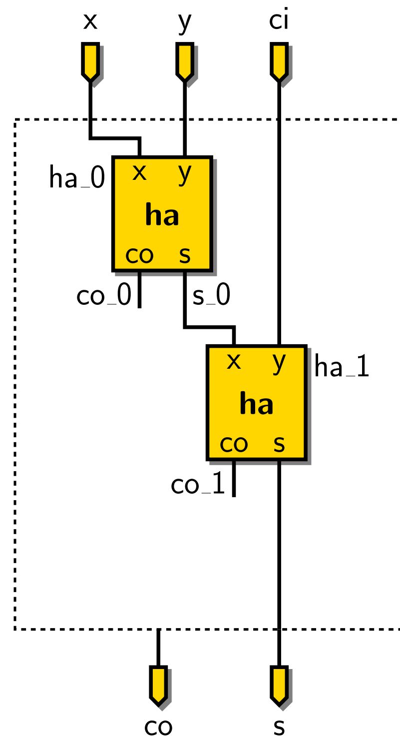
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x   : in  std_logic;
         y   : in  std_logic;
         ci  : in  std_logic;
         s   : out std_logic;
         co  : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;
  signal co_1 : std_logic;
begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );
  ha_1 : ha port map ( x => s_0, y  => ci,
                       s => s,   co => co_1 );

end architecture;
```

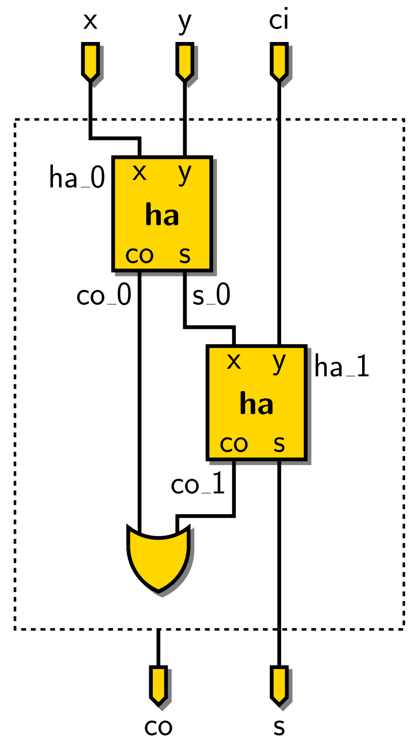
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x   : in  std_logic;
         y   : in  std_logic;
         ci  : in  std_logic;
         s   : out std_logic;
         co  : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;
  signal co_1 : std_logic;
begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );
  ha_1 : ha port map ( x => s_0, y  => ci,
                       s => s,   co => co_1 );
  co <= co_0 or co_1;
end architecture;
```

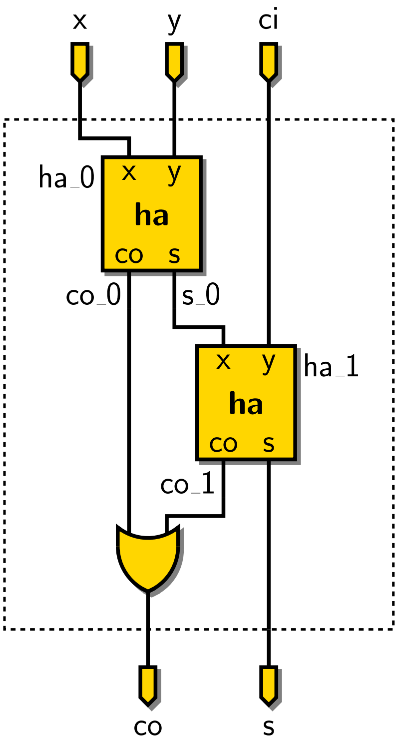
$x + y + ci = s + 2co$



A full-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         ci : in  std_logic;
         s  : out std_logic;
         co : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;
  signal co_1 : std_logic;
begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );
  ha_1 : ha port map ( x => s_0, y  => ci,
                       s => s,   co => co_1 );
  co <= co_0 or co_1;
end architecture;
```

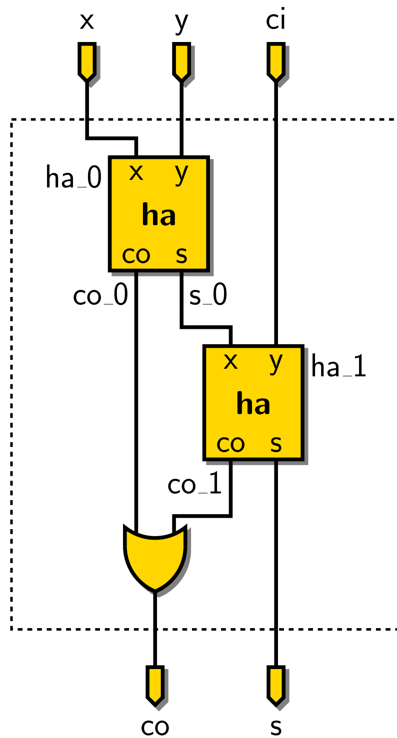
$$x + y + ci = s + 2co$$



A full-adder in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fa is
  port ( x   : in  std_logic;
         y   : in  std_logic;
         ci  : in  std_logic;
         s   : out std_logic;
         co  : out std_logic );
end entity;

architecture arch of fa is
  component ha is
    port ( x : in  std_logic; y  : in  std_logic;
           s : out std_logic; co : out std_logic );
  end component;
  signal s_0  : std_logic;
  signal co_0 : std_logic;
  signal co_1 : std_logic;
begin
  ha_0 : ha port map ( x => x,   y  => y,
                       s => s_0, co => co_0 );
  ha_1 : ha port map ( x => s_0, y  => ci,
                       s => s,   co => co_1 );
  co <= co_0 or co_1;
end architecture;
```

$$x + y + ci = s + 2co$$



Design process

- Verification and debugging
  - software simulator
  - feed the circuit with test vectors
  - extensive use of waveforms for debugging

Design process

- Verification and debugging
  - software simulator
  - feed the circuit with test vectors
  - extensive use of waveforms for debugging
- Synthesis
  - converts the circuit description (HDL) into a netlist
  - extraction of logic primitives (multiplexers, shifters, registers, adders, ...)
  - logic minimization effort
  - independent from the target technology

Design process

- Verification and debugging
  - software simulator
  - feed the circuit with test vectors
  - extensive use of waveforms for debugging
- Synthesis
  - converts the circuit description (HDL) into a netlist
  - extraction of logic primitives (multiplexers, shifters, registers, adders, ...)
  - logic minimization effort
  - independent from the target technology
- Implementation
  - mapping: builds a netlist of technology-dependent logic cells / transistors
  - place and route: place each logic cell on the chip and route wires between them

Arithmetic over $\mathbb{F}_{2^m}$

- Polynomial representation: $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/(F(x))$

Arithmetic over $\mathbb{F}_{2^m}$

- Polynomial representation: $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/(F(x))$
  - elements of $\mathbb{F}_{2^m}$ as polynomials modulo $F(x)$:

$$A = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, \qquad \text{with } a_i \in \mathbb{F}_2$$

  - 1 bit per coefficient

Arithmetic over $\mathbb{F}_{2^m}$

- Polynomial representation: $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/(F(x))$
  - elements of $\mathbb{F}_{2^m}$ as polynomials modulo $F(x)$:

$$A = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, \qquad \text{with } a_i \in \mathbb{F}_2$$

  - 1 bit per coefficient
- Addition: coefficient-wise addition over $\mathbb{F}_p$

Arithmetic over $\mathbb{F}_{2^m}$

- Polynomial representation: $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/(F(x))$
  - elements of $\mathbb{F}_{2^m}$ as polynomials modulo $F(x)$:

$$A = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, \qquad \text{with } a_i \in \mathbb{F}_2$$

  - 1 bit per coefficient
- Addition: coefficient-wise addition over $\mathbb{F}_p$
- Squaring: 2-nd power Frobenius

Arithmetic over $\mathbb{F}_{2^m}$

- Polynomial representation: $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/(F(x))$
  - elements of $\mathbb{F}_{2^m}$ as polynomials modulo $F(x)$:

$$A = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, \qquad \text{with } a_i \in \mathbb{F}_2$$

  - 1 bit per coefficient
- Addition: coefficient-wise addition over $\mathbb{F}_p$
- Squaring: 2-nd power Frobenius
  - linear operation: each coefficient of the result is a linear combination of the input coefficients
  - for instance, over $\mathbb{F}_{2^{409}} = \mathbb{F}_2[x]/(x^{409} + x^{87} + 1)$

$$A^2 = \ldots + (a_{86} + a_{247} + a_{408})x^{172} + \ldots + (a_{213} + a_{374})x^{17} + \ldots$$

Arithmetic over $\mathbb{F}_{2^m}$

- Polynomial representation: $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/(F(x))$
  - elements of $\mathbb{F}_{2^m}$ as polynomials modulo $F(x)$:

$$A = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, \qquad \text{with } a_i \in \mathbb{F}_2$$

  - 1 bit per coefficient
- Addition: coefficient-wise addition over $\mathbb{F}_p$
- Squaring: 2-nd power Frobenius
  - linear operation: each coefficient of the result is a linear combination of the input coefficients
  - for instance, over $\mathbb{F}_{2^{409}} = \mathbb{F}_2[x]/(x^{409} + x^{87} + 1)$

$$A^2 = \ldots + (a_{86} + a_{247} + a_{408})x^{172} + \ldots + (a_{213} + a_{374})x^{17} + \ldots$$

- Inversion: no need for a full blown extended Euclidean algorithm

Arithmetic over $\mathbb{F}_{2^m}$

- Polynomial representation: $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/(F(x))$
  - elements of $\mathbb{F}_{2^m}$ as polynomials modulo $F(x)$:

$$A = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, \qquad \text{with } a_i \in \mathbb{F}_2$$

  - 1 bit per coefficient

- Addition: coefficient-wise addition over $\mathbb{F}_p$

- Squaring: 2-nd power Frobenius
  - linear operation: each coefficient of the result is a linear combination of the input coefficients
  - for instance, over $\mathbb{F}_{2^{409}} = \mathbb{F}_2[x]/(x^{409} + x^{87} + 1)$

$$A^2 = \ldots + (a_{86} + a_{247} + a_{408})x^{172} + \ldots + (a_{213} + a_{374})x^{17} + \ldots$$

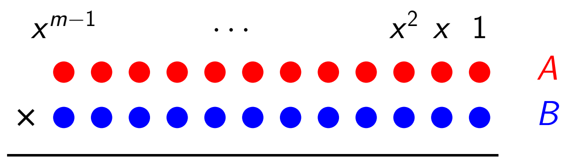
- Inversion: no need for a full blown extended Euclidean algorithm
  - use Fermat's little theorem: $A^{-1} = A^{2^m-2} = \left(A^{2^{m-1}-1}\right)^2$
  - computing $A^{2^{m-1}-1}$ only requires multiplications and Frobeniuses [Itoh and Tsujii, 1988]
  - no extra hardware for inversion

Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)

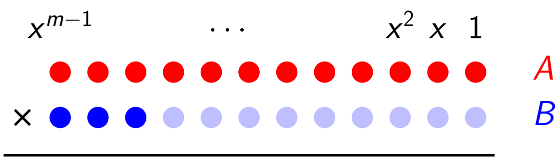
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



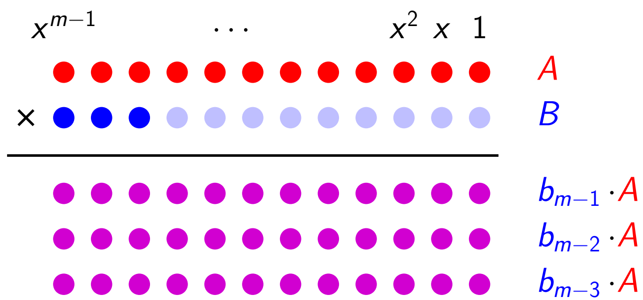
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



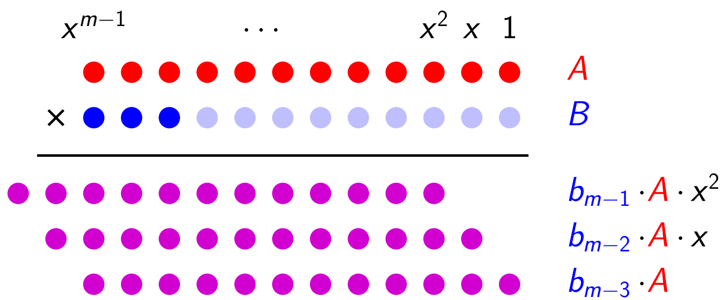
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



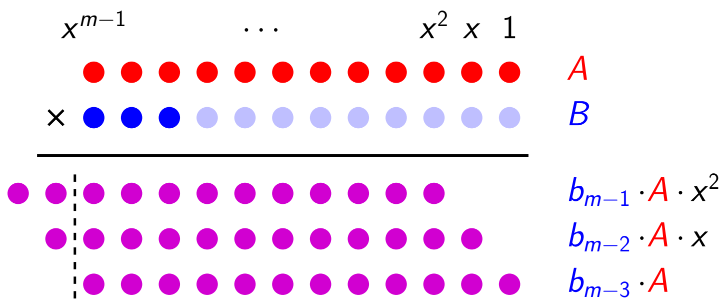
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



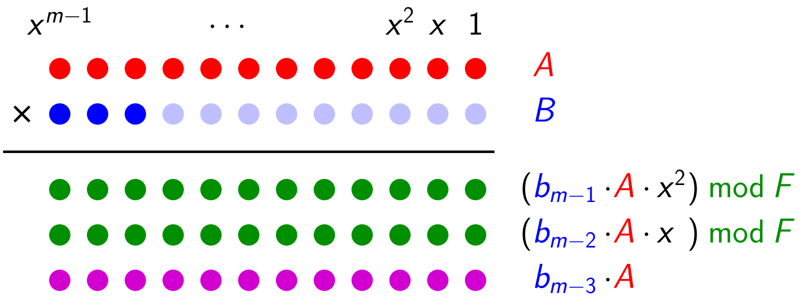
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - *d* coefficients of the second operand processed at each iteration (most-significant coefficients first)



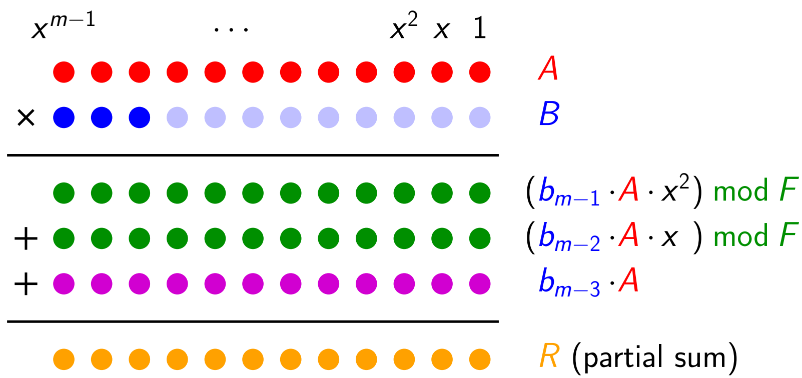
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



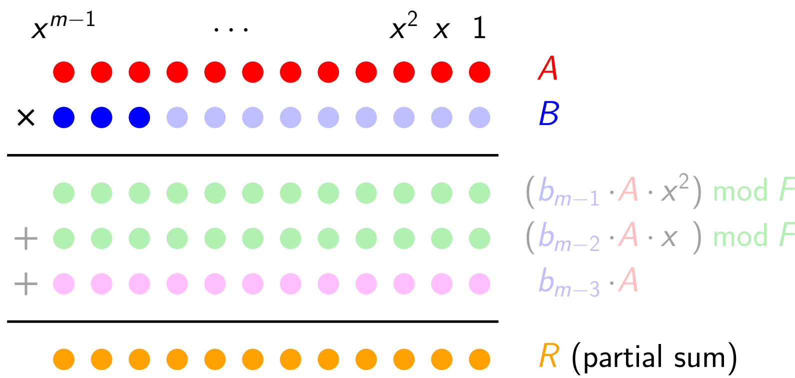
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



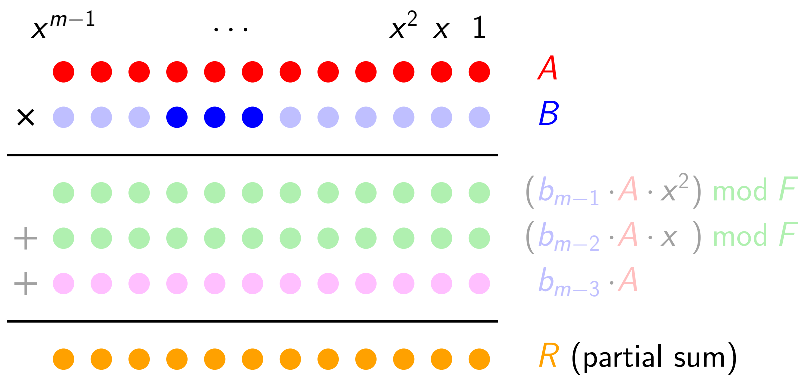
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



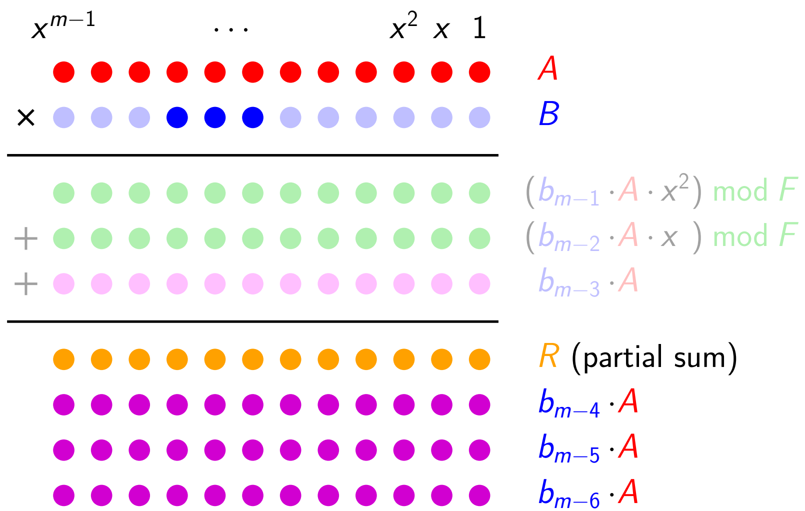
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



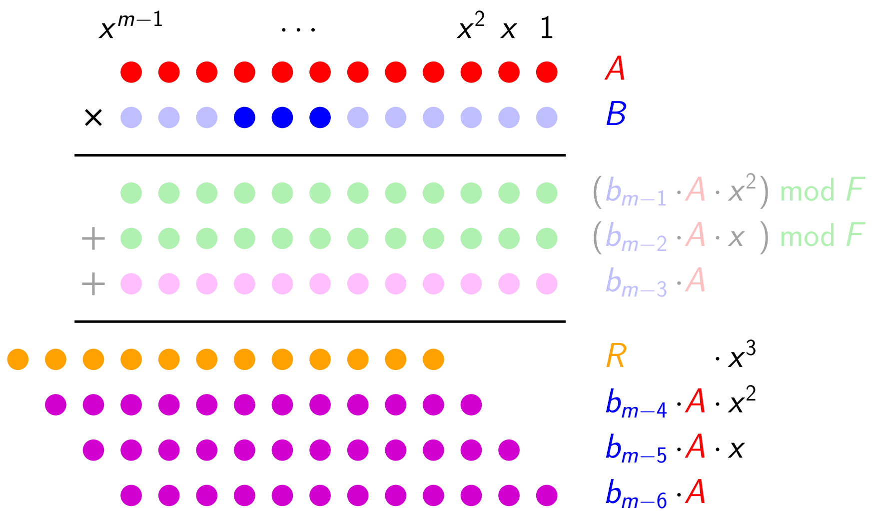
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



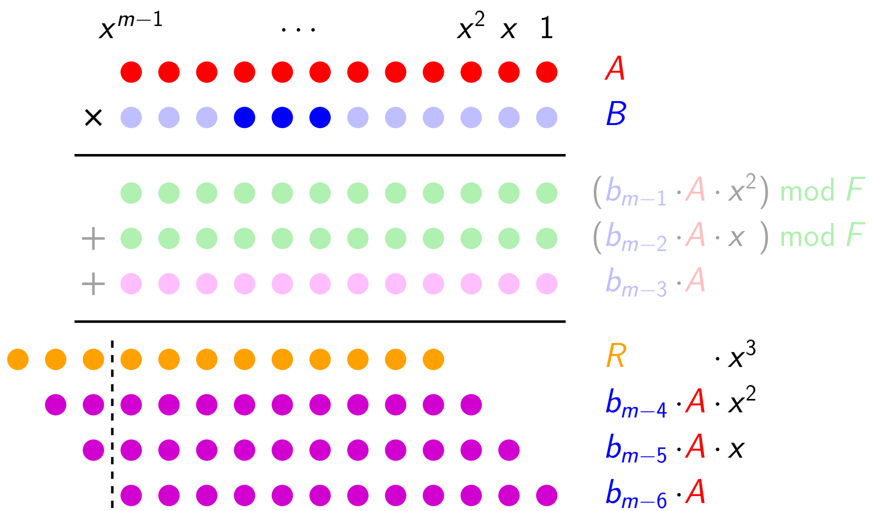
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



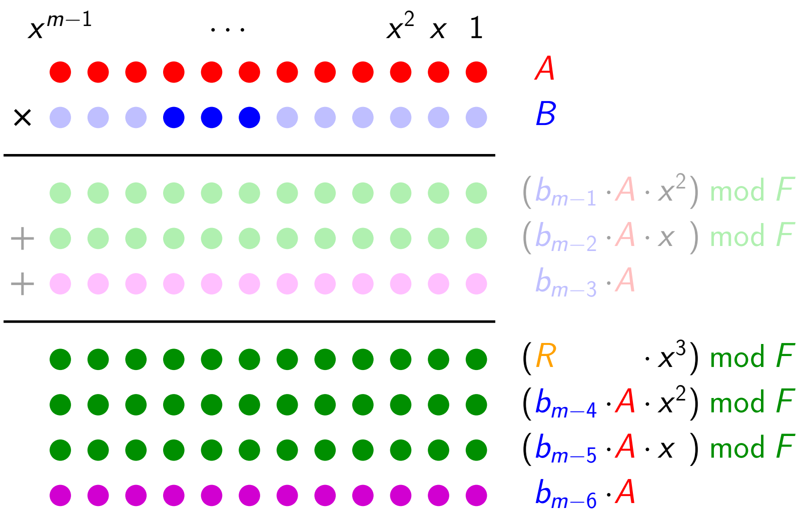
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



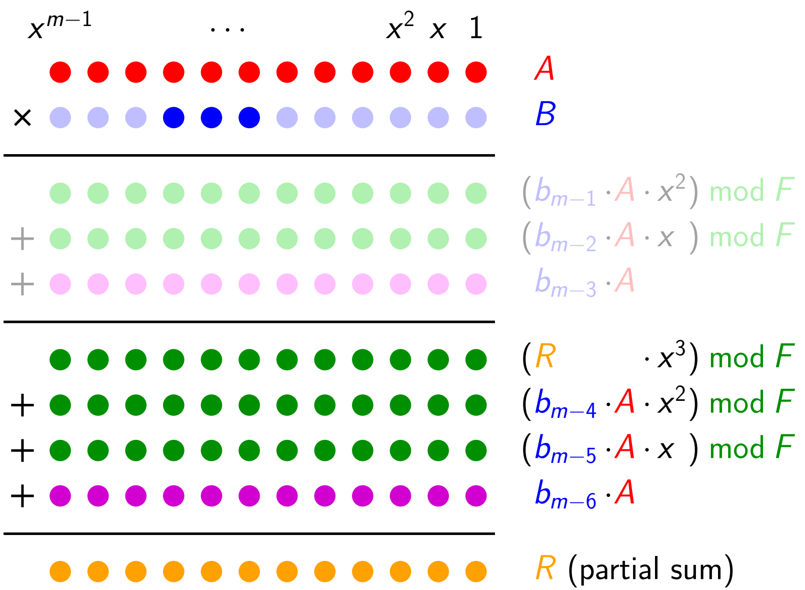
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



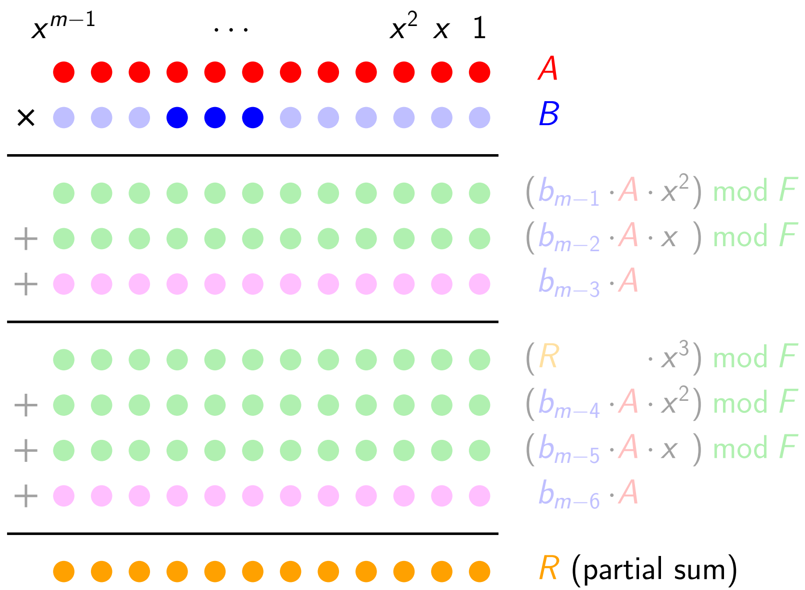
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



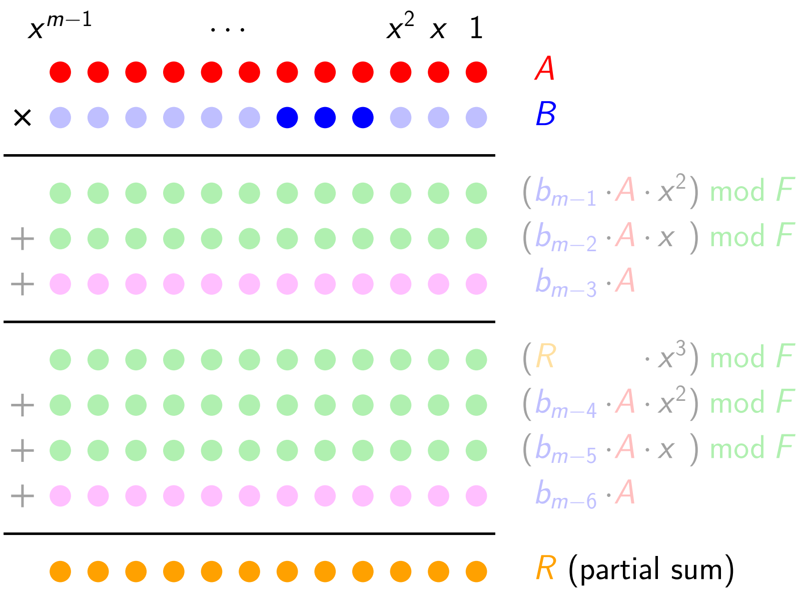
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)



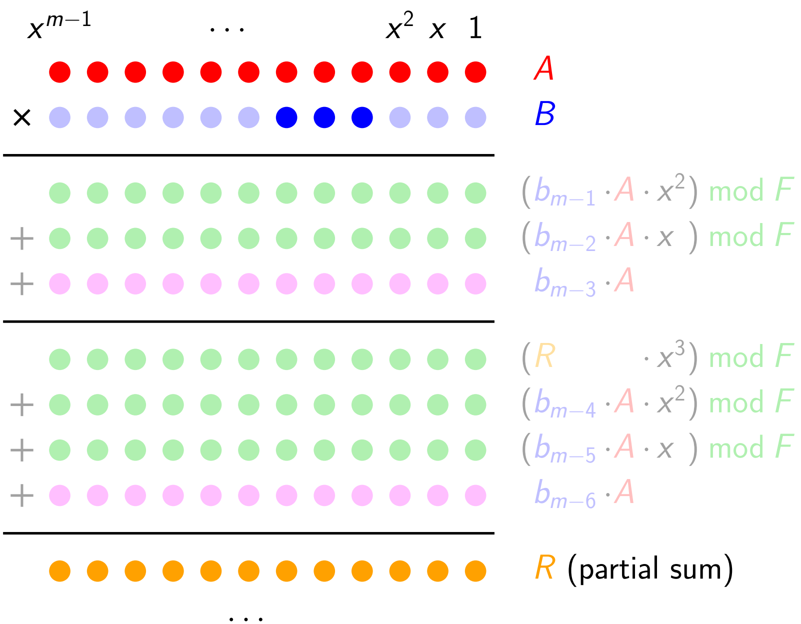
Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)

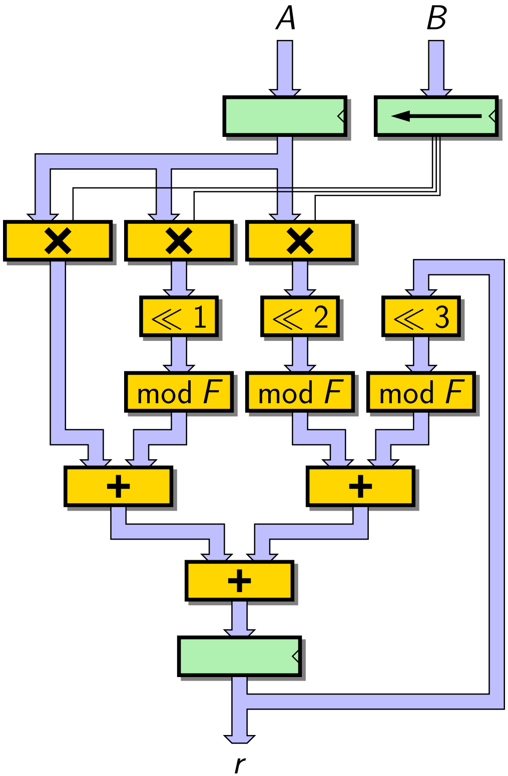


Multiplication over $\mathbb{F}_{2^m}$

- Low-area design: parallel–serial multiplier
  - iterative algorithm of quadratic complexity
  - d coefficients of the second operand processed at each iteration (most-significant coefficients first)
  - $\lceil m/d \rceil$ clock cycles for computing the product
  - area grows with d: area–time trade-off

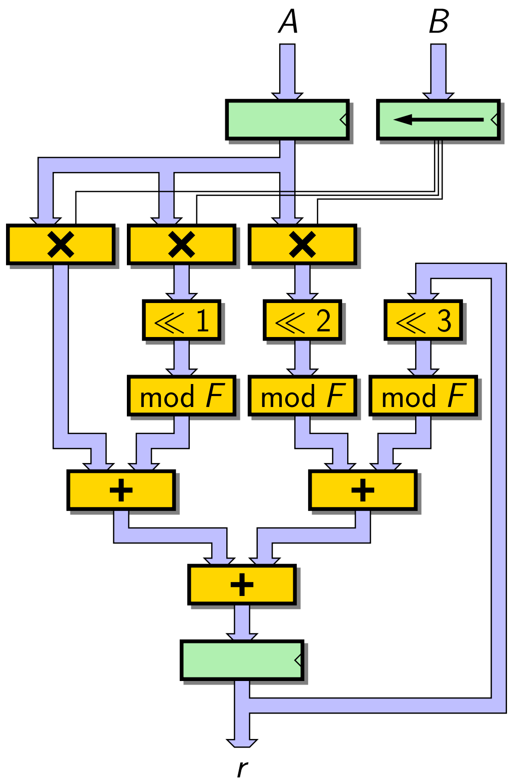


Multiplication over $\mathbb{F}_{2^m}$



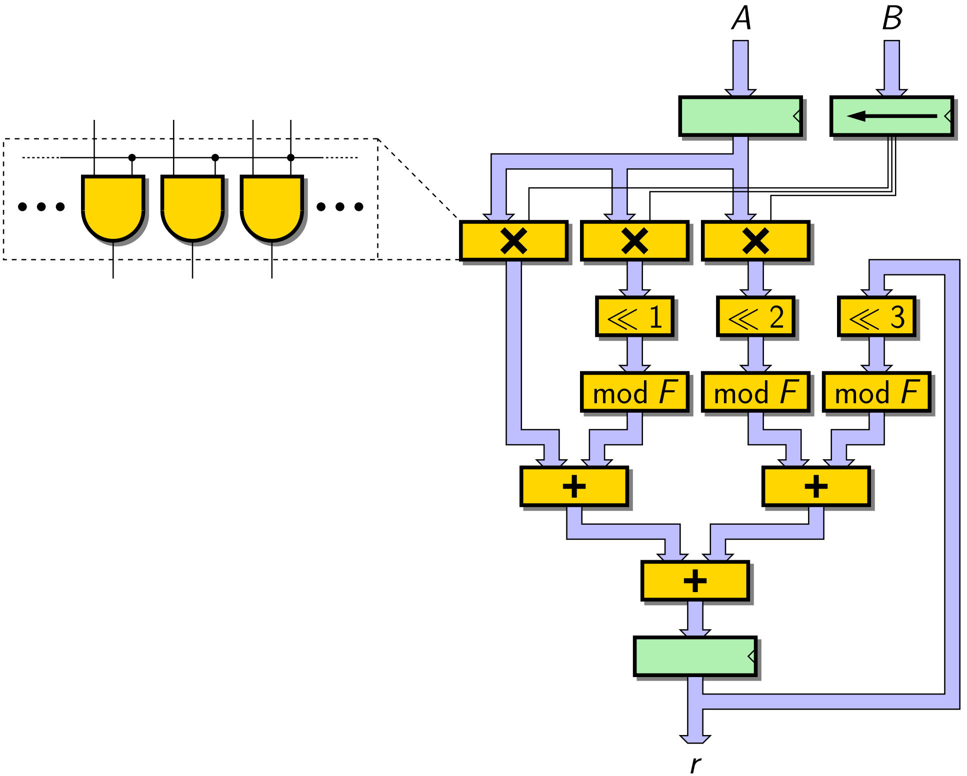
Multiplication over $\mathbb{F}_{2^m}$

- feedback loop for accumulation of the result



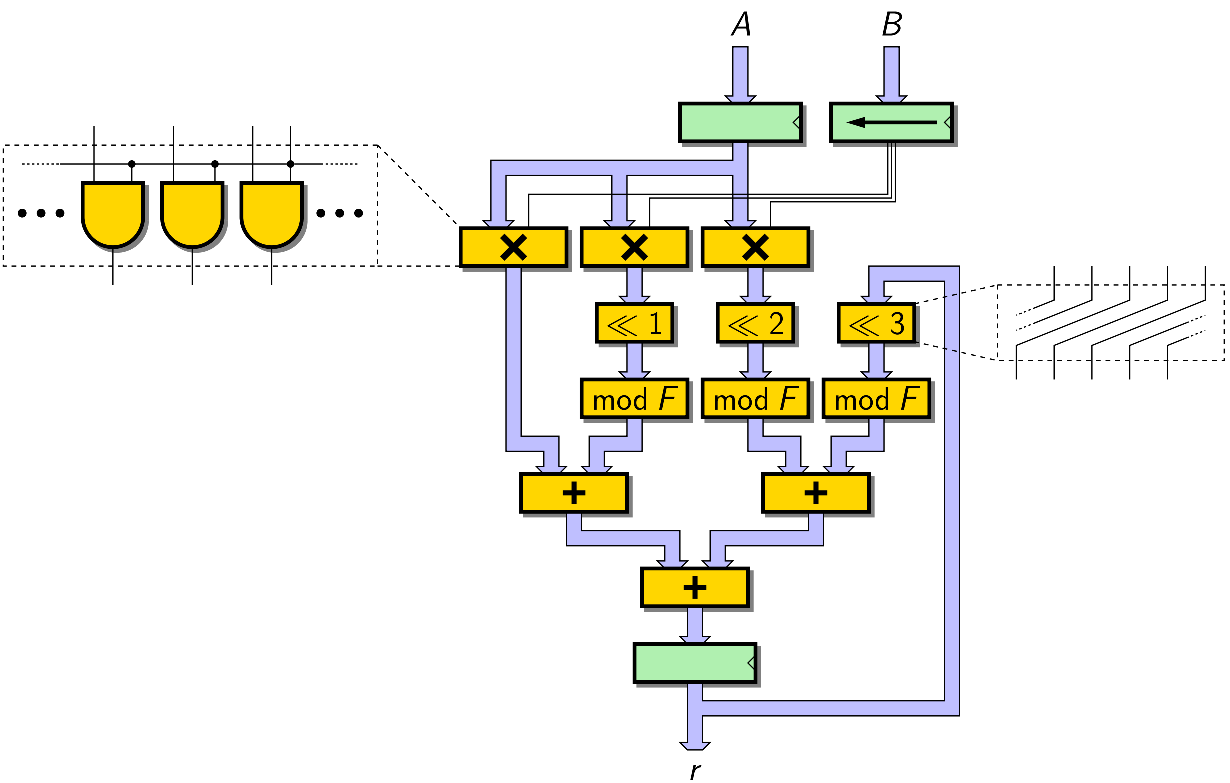
Multiplication over $\mathbb{F}_{2^m}$

- feedback loop for accumulation of the result
- coefficient-wise partial product with $\mathbb{F}_2$ multipliers (AND gates)



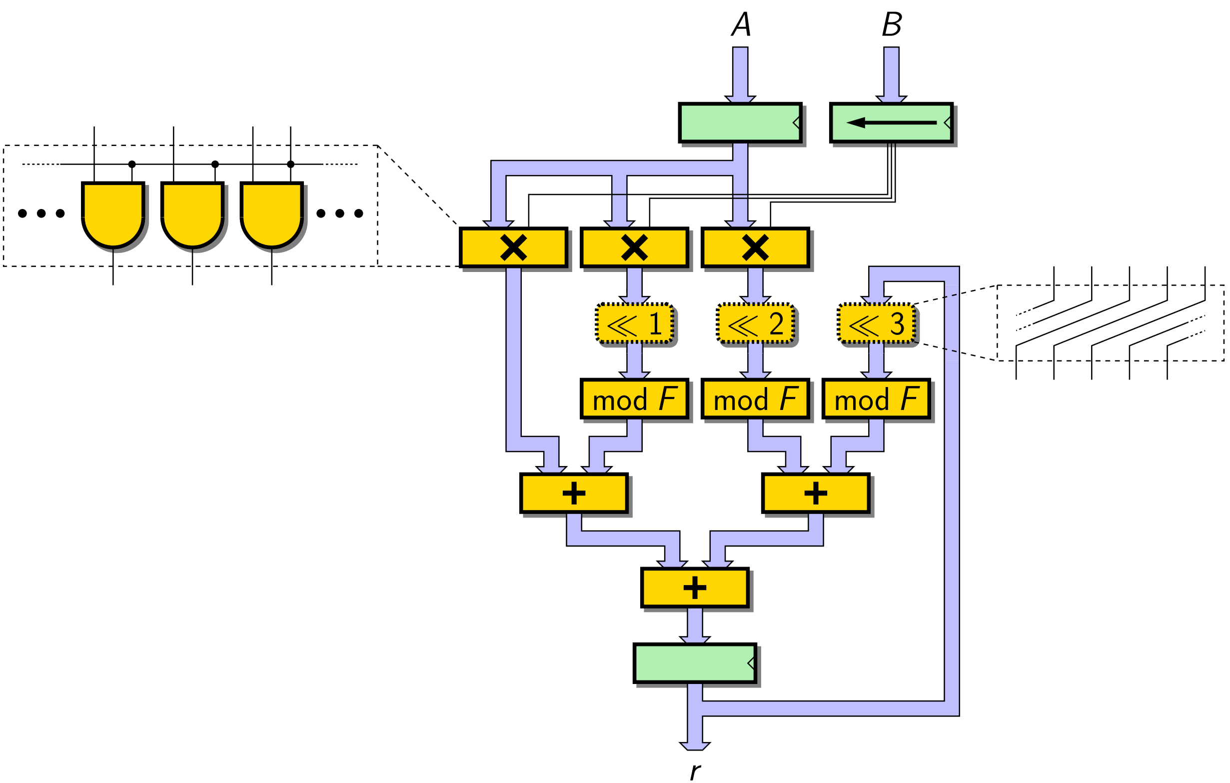
Multiplication over $\mathbb{F}_{2^m}$

- feedback loop for accumulation of the result
- coefficient-wise partial product with $\mathbb{F}_2$ multipliers (AND gates)
- free shifts!



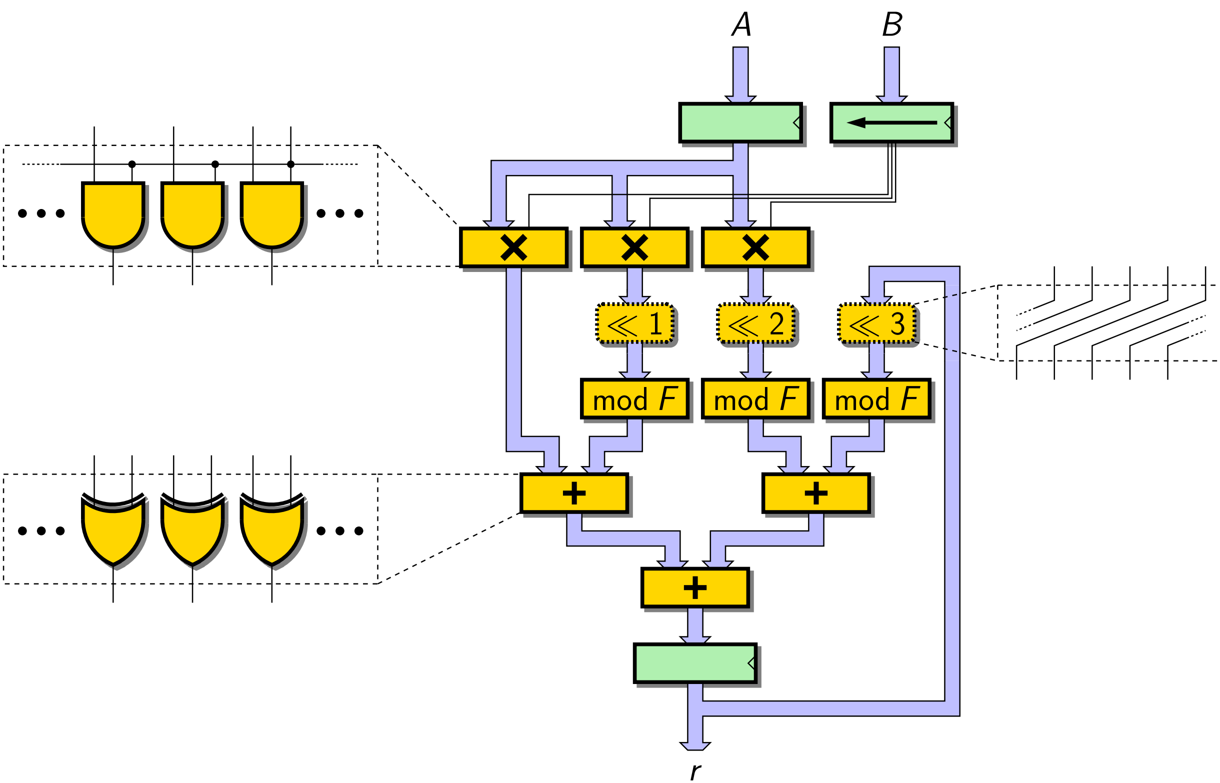
Multiplication over $\mathbb{F}_{2^m}$

- feedback loop for accumulation of the result
- coefficient-wise partial product with $\mathbb{F}_2$ multipliers (AND gates)
- free shifts!
- a few $\mathbb{F}_2$ adders for reduction modulo F



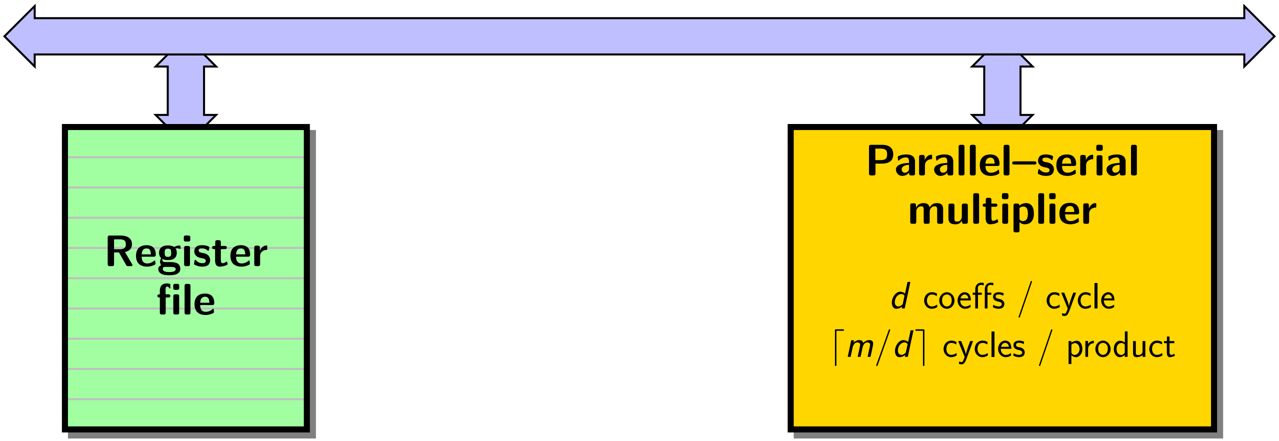
Multiplication over $\mathbb{F}_{2^m}$

- feedback loop for accumulation of the result
- coefficient-wise partial product with $\mathbb{F}_2$ multipliers (AND gates)
- free shifts!
- a few $\mathbb{F}_2$ adders for reduction modulo F
- coefficient-wise addition (XOR gates in $\mathbb{F}_2$)

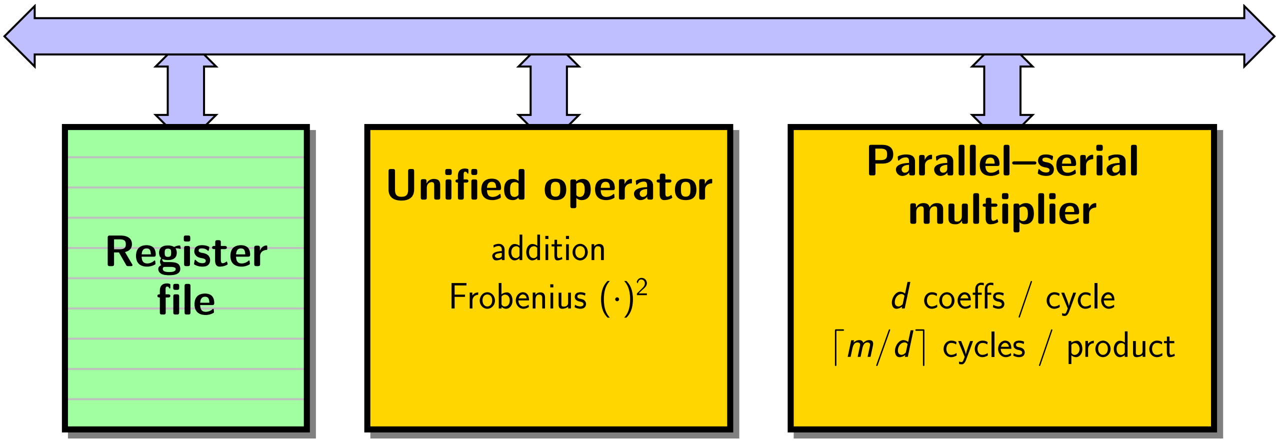


Arithmetic coprocessor for ECC over $\mathbb{F}_{2^m}$

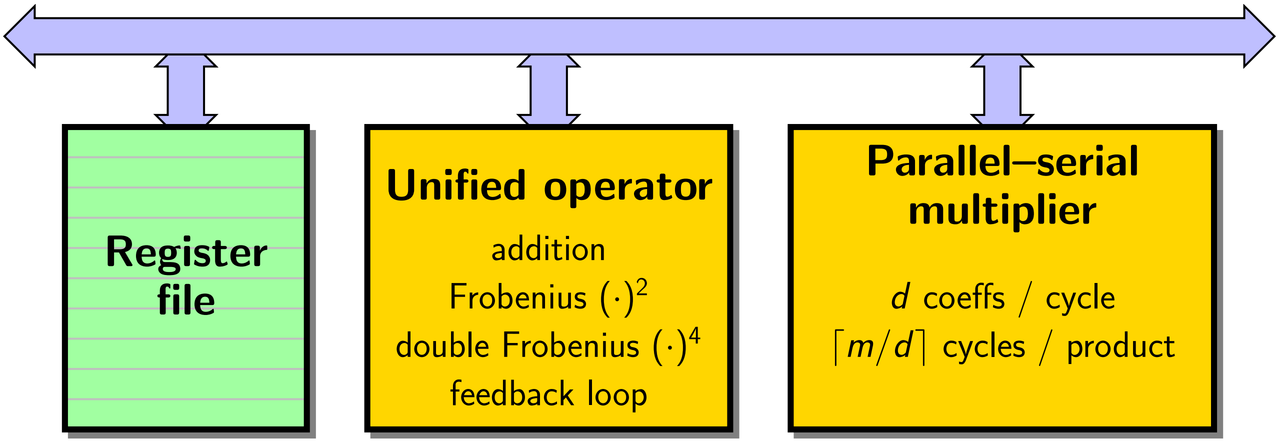
Arithmetic coprocessor for ECC over $\mathbb{F}_{2^m}$



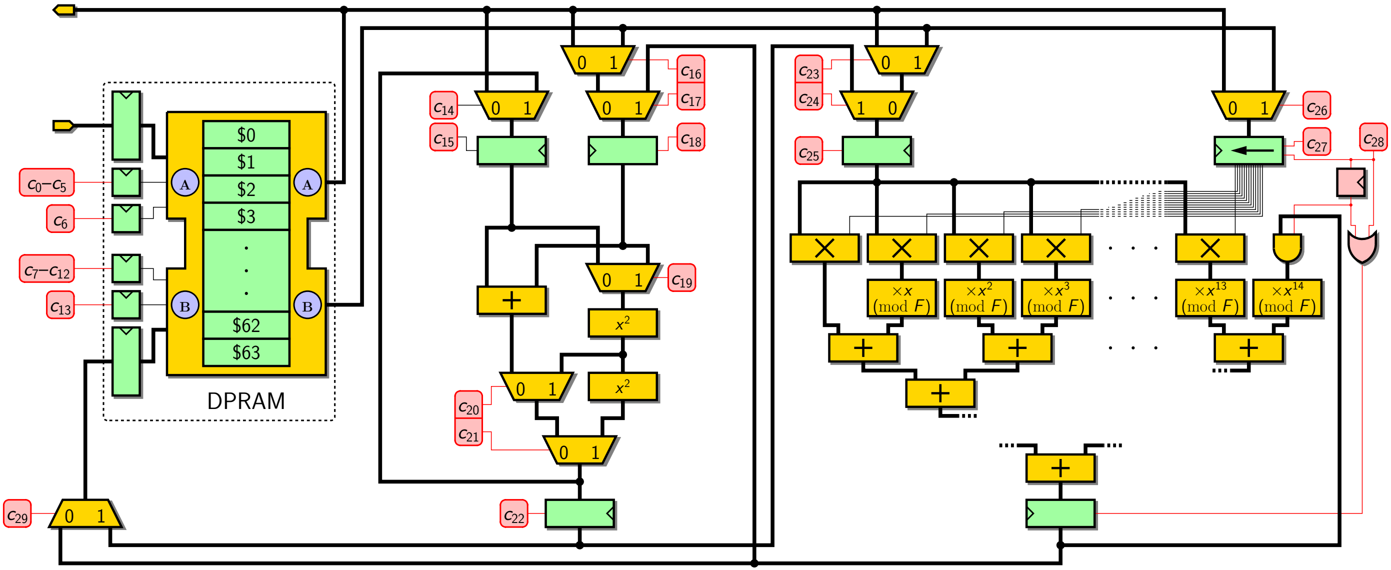
Arithmetic coprocessor for ECC over $\mathbb{F}_{2^m}$



Arithmetic coprocessor for ECC over $\mathbb{F}_{2^m}$



Arithmetic coprocessor for ECC over $\mathbb{F}_{2^m}$



Thank you for your attention

Questions?