Software and Hardware Implementation of Elliptic Curve Cryptography

Jérémie Detrey
CARAMEL team, LORIA
INRIA Nancy – Grand Est, France
Jeremie.Detrey@loria.fr

CC caramel.c; echo f3 f2 f1 f0 p | ./a.out
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 $\text{char}(\mathbb{F}_q) \neq 2, 3$
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 $\text{char}(\mathbb{F}_q) \neq 2, 3$

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

$$E(\mathbb{F}_q) = \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q \mid (x, y) \text{ satisfy } E\} \cup \{O\}$$
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 $\text{char}(\mathbb{F}_q) \neq 2, 3$

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

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

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

[See D. Robert's lectures]
The group law
The group law

\[ P + Q \]
The group law

\[ E/\mathbb{F}_{17} : y^2 = x^3 + x + 7 \]

\[ P + Q \]

\[ O \]
The group law

\[ E/\mathbb{F}_{17} : y^2 = x^3 + x + 7 \]
Scalar multiplication and discrete logarithm

$E(\mathbb{F}_q)$ is a finite abelian group:

- let $\mathcal{G}$ be a cyclic subgroup of $E(\mathbb{F}_q)$
- let $\ell = \#\mathcal{G}$ the order of $\mathcal{G}$ and $P \in \mathcal{G}$ a generator of $\mathcal{G}$
Scalar multiplication and discrete logarithm

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

$$\mathcal{G} = \langle P \rangle = \{0, P, 2P, 3P, \ldots, (\ell - 1)P\}$$
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 $\mathbb{G}$

$$\mathbb{G} = \langle P \rangle = \{0, P, 2P, 3P, \ldots, (\ell - 1)P\}$$

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

$$\exp_P : \mathbb{Z}/\ell\mathbb{Z} \rightarrow \mathbb{G}$$

$$k \mapsto kP = P + P + \ldots + P$$

$k$ times
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 $\mathbb{G}$

$$\mathbb{G} = \langle P \rangle = \{0, P, 2P, 3P, \ldots, (\ell - 1)P\}$$

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

$$\exp_P : \mathbb{Z}/\ell\mathbb{Z} \rightarrow \mathbb{G}$$

$$k \mapsto kP = P + P + \ldots + P$$

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

$$\operatorname{dlog}_P = \exp_P^{-1} : \mathbb{G} \rightarrow \mathbb{Z}/\ell\mathbb{Z}$$

$$Q \mapsto k$$ such that $Q = kP$
Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

\[ P = k \cdot P \]

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

\[ Q = P^k \]

- That's a one-way function \( \Rightarrow \) Public-key cryptography!

- **private key**: an integer \( k \) in \( \mathbb{Z}/\mathbb{Z} \)

- **public key**: the point \( kP \) in \( G \subseteq E(\mathbb{F}_q) \)
Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

$$kP$$

- 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):

\[ Q = kP \]
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:

\[ k \rightarrow kP \]

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

\[ Q = kP \]

[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:

  \[ k \rightarrow kP \]

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

  \[ Q = kP \]

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

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

\[
\begin{align*}
\text{Alice} & \quad \text{Bob} \\
\text{aP} & \quad \text{bP}
\end{align*}
\]
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?

![Diagram showing the EC Diffie–Hellman key exchange process]

- Alice and Bob use a public channel to exchange messages:
  - Alice sends $aP$ to Bob.
  - Bob sends $bP$ to Alice.

- Each computes $abP$: Alice computes $(aP)bP = abP$, and Bob computes $(bP)aP = abP$.

- They both compute the same shared secret key $abP$. 

This process allows them to establish a secure communication channel without directly sharing the secret key.
Example 2: EC ElGamal encryption
Example 2: EC ElGamal encryption

Alice

PKI

Bob

$P$

$bP$

$P$

$bP$

$P$

$P$
Example 2: EC ElGamal encryption
Example 2: EC ElGamal encryption
Example 2: EC ElGamal encryption

- Alice
  - \( k \)
  - \( kP \)
  - \( bkP \)

- Bob
  - \( b \)

- PKI
  - \( P \)
  - \( bP \)
Example 2: EC ElGamal encryption

Alice

Bob

PKI
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]
Many possible meanings for efficiency:
Many possible meanings for efficiency:
- fast? → low latency or high throughput?
Efficient and secure implementation?

► Many possible meanings for efficiency:
  - fast? → low latency or high throughput?
  - small? → low memory / code / silicon usage?
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?**
- **power or electromagnetic analysis?**
- **etc.**

Possible attack scenarios depend on the application
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
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?
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]
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.)
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]
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]
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?
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? → 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?
  ● power or electromagnetic analysis?
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?
- power or electromagnetic analysis?
- etc.

⇒ Possible attack scenarios depend on the application
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.
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
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)

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
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)
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
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
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.)
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.)
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.)

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!

⇒ ECC is no more secure than its weakest link

In these lectures, we will mostly focus on the green layers.
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)

- 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!

⇒ ECC is no more secure than its weakest link

- In these lectures, we will mostly focus on the green 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.)
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**
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!

⇒ ECC is no more secure than its weakest link

In these lectures, we will mostly focus on the green layers.
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!

⇒ 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!
  ⇒ 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):
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.
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.
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]
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,

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

Mathematics of Public-Key Cryptography,
Steven D. Galbraith.
Some references

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

*Handbook of Elliptic and Hyperelliptic Curve Cryptography*,
Henri Cohen and Gerhard Frey (editors).

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 $G \subseteq E(\mathbb{F}_q)$, we want to compute

$$kP = P + P + \ldots + P$$

$k$ times
Scalar multiplication

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

$$kP = P + P + \ldots + P$$

$k$ times

Size of $\ell$ (and $k$) for crypto applications: between 250 and 500 bits
Scalar multiplication

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

  $$kP = P + P + \ldots + P$$

  $k$ times

- Size of $\ell$ (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 \(k\):

\[
\text{function scalar-mult}(k, P) :
\begin{align*}
T &\leftarrow 0 \\
\text{for } i &\leftarrow n - 1 \text{ downto } 0 : \\
&\quad T &\leftarrow 2T \\
&\quad \text{if } k_i = 1 : \\
&\quad &\quad T &\leftarrow T + P \\
\text{return } T
\end{align*}
\]
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 \(k\):

```plaintext
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: \(k = 431\)
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 \(k\):

\[
\textbf{function} \quad \text{scalar-mult}(k, P):
\]
\[
T \gets O
\]
\[
\text{for } i \gets n - 1 \text{ downto } 0:
\]
\[
T \gets 2T
\]
\[
\text{if } k_i = 1:
\]
\[
T \gets T + P
\]
\[
\text{return } T
\]

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):
  \(T \leftarrow O\)
  for \(i \leftarrow n - 1\) downto 0:
    \(T \leftarrow 2T\)
    if \(k_i = 1\):
      \(T \leftarrow T + P\)
  return \(T\)
  ```

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

  \[
  T = \quad \quad = O
  \]
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 \(k\):

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

Example: \(k = 431 = (110101111)_2\)

\[
T = P = P
\]
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 $k$:

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

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

  $T = P \cdot 2 = 2P$
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 \(k\):

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

Example: \(k = 431 = (110101111)_2\)

\[
T = P \cdot 2 + P = 3P
\]
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 \(k\):

\[
\text{function } \text{scalar-mult}(k, P):
\]

\[
T \leftarrow \emptyset
\]

\[
\text{for } i \leftarrow n - 1 \text{ downto } 0:
\]

\[
T \leftarrow 2T
\]

\[
\text{if } k_i = 1:
\]

\[
T \leftarrow T + P
\]

\[
\text{return } T
\]

Example: \(k = 431 = (110101111)\_2\)

\[
T = (P \cdot 2 + P) \cdot 2 = 6P
\]
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 \(k\):

\[
\text{function scalar-mult}(k, P):
T \leftarrow \emptyset
\]
\[
\text{for } i \leftarrow n - 1 \text{ downto } 0:
T \leftarrow 2T
\]
\[
\text{if } k_i = 1:
T \leftarrow T + P
\]

return \(T\)

Example: \(k = 431 = (110101111)_2\)

\[
T = (P \cdot 2 + P) \cdot 2^2 = 12P
\]
Denoting by \((k_{n-1} \ldots k_1 k_0)_2\), with \(n = \lceil \log_2 \ell \rceil\), the binary expansion of \(k\):

\[
\text{function scalar-mult}(k, P):
\]

\[
T \leftarrow O
\]

\[
\text{for } i \leftarrow n - 1 \text{ downto } 0:
\]

\[
T \leftarrow 2T
\]

\[
\text{if } k_i = 1:
\]

\[
T \leftarrow T + P
\]

return \(T\)

Example: \(k = 431 = (110101111)_2\)

\[
T = (P \cdot 2 + P) \cdot 2^2 + P
\]

\(= 13P\)
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: \(k = 431 = (110101111)_2\)

  \[
  T = \left( (P \cdot 2 + P) \cdot 2^2 + P \right) \cdot 2 = 26P
  \]
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: \(k = 431 = (110101111)_2\)

\[
T = ((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 = 52P
\]
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 \(k\):

```plaintext
function scalar-mult(k, P):
    T ← \(\emptyset\)
    for \(i \leftarrow n - 1\) downto 0:
        T ← 2T
        if \(k_i = 1\):
            T ← \(T + P\)
    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 \((k_{n-1} \ldots k_1 k_0)_2\), with \(n = \lceil \log_2 \ell \rceil\), the binary expansion of \(k\):

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

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

  \[
  T = (((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 = 106P
  \]
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 ← \emptyset
  for \(i \leftarrow n - 1\) downto 0:
    T ← 2\(T\)
    if \(k_i = 1\):
      \(T \leftarrow T + P\)
  return \(T\)
  ```

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

  \[
  T = (((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P = 107P
  \]
Double-and-add algorithm

Denoting by \((k_{n−1} \ldots k_1k_0)_2\), with \(n = \lceil \log_2 ℓ \rceil\), the binary expansion of \(k\):

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

Example: \(k = 431 = (11010111)_2\)

\[
T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 = 214P
\]
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 \(k\):

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

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

  \[
  T = (((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P = 215P
  \]
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 ← Ø
  for i ← n − 1 downto 0:
    T ← 2T
    if \(k_i = 1\):
      T ← T + P
  return T
  ```

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

  \[
  T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 = 430P
  \]
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: \(k = 431 = (1101011111)_2\)

  \[T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 + P = 431P\]
Double-and-add algorithm

Denoting by \((k_{n-1} \ldots k_1k_0)_2\), with \(n = \lceil \log_2 \ell \rceil\), the binary expansion of \(k\):

\[
\text{function scalar-mult}(k, P):
T \leftarrow \emptyset
\text{for } i \leftarrow n - 1 \text{ downto } 0:
T \leftarrow 2T
\text{if } k_i = 1:
T \leftarrow T + P
\text{return } T
\]

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

\[
\text{function} \quad \text{scalar-mult}(k, P): \\
T \leftarrow \emptyset \\
\text{for } i \leftarrow n - 1 \text{ downto } 0: \\
\quad T \leftarrow 2T \\
\quad \text{if } k_i = 1: \\
\quad \quad T \leftarrow T + P \\
\text{return } T
\]

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

Example with $w = 3$:

$k = 431 = (110101111)_2 = (657)_3$

$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P$
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 $w = 3$: $k = 431$
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 $w = 3$: $k = 431 = (110101111)_2$
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 $w = 3$: $k = 431 = (110 101 111)_2 = (657)_3$
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 $w = 3$: $k = 431 = (11010111)_2 = (657)_3$

\[ T = O \]
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 $w = 3$: $k = 431 = (11010111)_2 = (657)_3$

\[
T = 6P = 6P
\]
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 $w = 3$: $k = 431 = (110\ 101\ 111)_2 = (657)_3$

  $$T = 6P \cdot 2^3 = 48P$$
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 $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_3$
  
  $T = 6P \cdot 2^3 + 5P = 53P$

- Select $w$ carefully so that precomputation cost does not become predominant

- Sliding window variant: half as many precomputations
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 $w = 3$: $k = 431 = (110101111)_2 = (657)_{2^3}$
  
  \[ T = (6P \cdot 2^3 + 5P) \cdot 2^3 = 424P \]
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 $w = 3$: $k = 431 = (110\ 101\ 111)_2 = (657)_3$

  $$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$
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 $w = 3$: $k = 431 = (110101111)_2 = (657)_3$
  \[ T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P \]
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 $w = 3$: $k = 431 = (110\ 101\ 111)_2 = (657)_3$
  \[
  T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P
  \]

- 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 $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions

- Example with $w = 3$: $k = 431 = (110 101 111)_2 = (657)_3$
  \[
  T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P
  \]

- 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
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

Example with $w = 3$ (digits in $\{-3, -1, 0, 1, 3\}$):

$k = 431 = (3003000\bar{1})_2$

$T = 3P \cdot 2^3 + 3P \cdot 2^4 - P = \ldots$

Complexity:
- $n$ doublings, and
- $n / (w + 1)$ additions on average
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
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 $\{-3, 1, 0, 1, 3\}$): $k = 431$
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 $\{-3, -1, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$
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 $3P, 5P, \ldots, (2^{w-1} - 1)P$:

- 1 doubling, and
- $2^{w-2} - 1$ additions

Example with $w = 3$ (digits in $\{-3, -1, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

$$T = \ldots = \mathcal{O}$$
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 $\{\overline{3}, 1, 0, 1, 3\}$): $k = 431 = (\overline{30030001})_2$

$$T = 3P = 3P$$
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 $3P, 5P, \ldots, (2^{w-1} - 1)P$:

- 1 doubling, and
- $2^{w-2} - 1$ additions

Example with $w = 3$ (digits in $\{-3, 1, 0, 1, 3\}$): $k = 431 = (30030001)_2$

$$T = 3P \cdot 2 = 6P.$$
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 $3P, 5P, \ldots, (2^{w-1} - 1)P$:
- 1 doubling, and
- $2^{w-2} - 1$ additions

Example with $w = 3$ (digits in $\{-3, -1, 0, 1, 3\}$): $k = 431 = (300\underline{300001})_2$

$$T = 3P \cdot 2^2 = 12P$$
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 $3P, 5P, \ldots, (2^{w-1} - 1)P$:
- 1 doubling, and
- $2^{w-2} - 1$ additions

Example with $w = 3$ (digits in $\{-3, 1, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

\[ T = 3P \cdot 2^3 = 24P \]
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 $\{\overline{3}, \overline{1}, 0, 1, 3\}$): $k = 431 = (\overline{30030001})_2$

$$T = 3P \cdot 2^3 + 3P = 27P$$
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 $\{\overline{3}, \overline{1}, 0, 1, 3\}$): $k = 431 = (3003000\overline{1})_2$

  \[ T = (3P \cdot 2^3 + 3P) \cdot 2 = 54P \]
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 $3P, 5P, \ldots, (2^{w-1} - 1)P$:
  - 1 doubling, and
  - $2^{w-2} - 1$ additions

- Example with $w = 3$ (digits in $\{-3, -1, 0, 1, 3\}$): $k = 431 = (300300\overline{1})_2$
  
  $$T = (3P \cdot 2^3 + 3P) \cdot 2^2 = 108P$$
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 $\{-2^{w-1} + 1, \ldots, 2^{w-1} - 1\}$ 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 $\{3, 1, 0, 1, 3\}$): $k = 431 = (300300101)_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2^3 = 216P$$
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 $\{ar{3}, \bar{1}, 0, 1, 3\}$): $k = 431 = (300300\bar{1})_2$

$$T = (3P \cdot 2^3 + 3P) \cdot 2^4 = 432P$$
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 $3P$, $5P$, $\ldots$, $(2^{w-1} - 1)P$:
  - 1 doubling, and
  - $2^{w-2} - 1$ additions
- Example with $w = 3$ (digits in $\{-3, -1, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$
  $$T = (3P \cdot 2^3 + 3P) \cdot 2^4 - P = 431P$$
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 $3P$, $5P$, $\ldots$, $(2^{w-1} - 1)P$:
  - 1 doubling, and
  - $2^{w-2} - 1$ additions.

- Example with $w = 3$ (digits in $\{-3, -1, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$.

$$T = (3P \cdot 2^3 + 3P) \cdot 2^4 - P = 431P$$
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 $3P, 5P, \ldots, (2^{w-1} - 1)P$:
- 1 doubling, and
- $2^{w-2} - 1$ additions

Example with $w = 3$ (digits in $\{-3, -1, 0, 1, 3\}$): $k = 431 = (3003000\bar{1})_2$

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

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

- To compute the sum of several scalar multiplications

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

To compute the sum of several scalar multiplications
e.g., \( aP + bQ \), where \( a, b \in \mathbb{Z}/\ell \mathbb{Z} \) and \( P, Q \in E(\mathbb{F}_q) \)

Idea:

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

```plaintext
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

\begin{function}
\text{double-scalar-mult}(a, P, b, Q):
\begin{align*}
S & \leftarrow P + Q \\
T & \leftarrow 0
\end{align*}
\begin{align*}
& \text{for } i \leftarrow n - 1 \text{ downto } 0:
& \quad T \leftarrow 2T \\
& \quad \text{if } a_i = 1 \text{ and } b_i = 1:\n& \quad \quad T \leftarrow T + S \\
& \quad \text{else if } a_i = 1:\n& \quad \quad T \leftarrow T + P \\
& \quad \text{else if } b_i = 1:\n& \quad \quad T \leftarrow T + Q
\end{align*}
\quad \text{return } T
\end{function}

\textbf{Example:}

\begin{align*}
a & = 21 = (10101)_2 \\
b & = 30 = (11110)_2
\end{align*}

\begin{align*}
T & = \left( (P + Q) \cdot 2 + Q \right) \cdot 2 + (P + Q) \cdot 2 + Q \\
& = (P + Q + P + Q) \cdot 2 + Q \\
& = \ldots
\end{align*}

\textbf{Complexity:}

\begin{itemize}
\item $n$ doublings,
\item $3n/4$ additions on average
\end{itemize}

\textbf{With signed digits:}

\begin{itemize}
\item joint sparse form (JSF): $n/2$ additions
\item interleaved w-NAF: $2n/(w + 1)$ additions
\end{itemize}
Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q):
    \( S \leftarrow P + Q \)
    \( T \leftarrow 0 \)
    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: \( a = 21 \) and \( b = 30 \)
Multi-exponentiation technique

```plaintext
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 ← P + Q
    T ← ∅
    for i ← n − 1 downto 0:
        T ← 2T
        if ai = 1 and bi = 1:
            T ← T + S
        else if ai = 1:
            T ← T + P
        else if bi = 1:
            T ← T + Q
    return T

Example: a = 21 = (10101)2
and b = 30 = (11110)2

T = O
Multi-exponentiation technique

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

\[
\begin{align*}
S & \leftarrow P + Q \\
T & \leftarrow 0 \\
\text{for } i \leftarrow n - 1 \text{ downto } 0: \\
& \quad T \leftarrow 2T \\
& \quad \text{if } a_i = 1 \text{ and } b_i = 1: \\
& \quad \quad T \leftarrow T + S \\
& \quad \quad \text{else if } a_i = 1: \\
& \quad \quad \quad T \leftarrow T + P \\
& \quad \quad \text{else if } b_i = 1: \\
& \quad \quad \quad T \leftarrow T + Q \\
\text{return } T
\end{align*}
\]

▶ Example: \( a = 21 = (10101)_2 \)
and \( b = 30 = (11110)_2 \)

\[
T = P + Q = P + Q
\]

▶ 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
Multi-exponentiation technique

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

\[
S \leftarrow P + Q \\
T \leftarrow 0
\]

for \(i \leftarrow n - 1\) downto 0:

\[
T \leftarrow 2T \\
\text{if } a_i = 1 \text{ and } b_i = 1:\ \\
T \leftarrow T + S \\
\text{else if } a_i = 1:\ \\
T \leftarrow T + P \\
\text{else if } b_i = 1:\ \\
T \leftarrow T + Q
\]

return \(T\)

▶ Example: \(a = 21 = (10101)_2\) and \(b = 30 = (11110)_2\)

\[
T = (P + Q) \cdot 2 = 2P + 2Q
\]
Multi-exponentiation technique

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

$S \leftarrow P + Q$

$T \leftarrow 0$

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 $b = 30 = (11110)_2$

$T = (P + Q) \cdot 2 + Q = 2P + 3Q$
Multi-exponentiation technique

function double-scalar-mult($a, P, b, Q$):
  $S \leftarrow P + Q$
  $T \leftarrow 0$
  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 $b = 30 = (11110)_2$

$T = ((P + Q) \cdot 2 + Q) \cdot 2 = 4P + 6Q$
Multi-exponentiation technique

```plaintext
function double-scalar-mult(\(a, P, b, Q\)):
    \(S \leftarrow P + Q\)
    \(T \leftarrow \emptyset\)
    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 \(b = 30 = (11110)_2\)

\[
T = ((P + Q) \cdot 2 + Q) \cdot 2 + P + Q = 5P + 7Q
\]
Multi-exponentiation technique

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

$S \leftarrow P + Q$
$T \leftarrow 0$

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 $b = 30 = (11110)_2$

$T = (((P + Q) \cdot 2 + Q) \cdot 2 + P + Q) \cdot 2 = 10P + 14Q$
Multi-exponentiation technique

\textbf{function} \textit{double-scalar-mult}(a, P, b, Q):
\begin{verbatim}
S \leftarrow P + Q
T \leftarrow 0
\textbf{for} i \leftarrow n - 1 \textbf{downto} 0:
\quad T \leftarrow 2T
\quad \textbf{if} a_i = 1 \textbf{ and} b_i = 1:
\quad \quad T \leftarrow T + S
\quad \textbf{else if} a_i = 1:
\quad \quad T \leftarrow T + P
\quad \textbf{else if} b_i = 1:
\quad \quad T \leftarrow T + Q
\textbf{return} T
\end{verbatim}

\begin{itemize}
\item \textbf{Example:} $a = 21 = (10101)_2$
\item and $b = 30 = (11110)_2$
\end{itemize}

\begin{equation}
T = (((P + Q) \cdot 2 + Q) \cdot 2 + P + Q) \cdot 2 + Q = 10P + 15Q
\end{equation}
Multi-exponentiation technique

```plaintext
function double-scalar-mult(\(a, P, b, Q\)):
    \(S \leftarrow P + Q\)
    \(T \leftarrow 0\)
    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 \(b = 30 = (11110)_2\)

\[T = (((((P + Q) \cdot 2 + Q) \cdot 2 + P + Q) \cdot 2 + Q) \cdot 2 = 20P + 30Q\]
Multi-exponentiation technique

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

\[
S \leftarrow P + Q \\
T \leftarrow 0 \\
\text{for } i \leftarrow n - 1 \text{ downto } 0: \\
\quad T \leftarrow 2T \\
\quad \text{if } a_i = 1 \text{ and } b_i = 1: \\
\quad \quad T \leftarrow T + S \\
\quad \text{else if } a_i = 1: \\
\quad \quad T \leftarrow T + P \\
\quad \text{else if } b_i = 1: \\
\quad \quad T \leftarrow T + Q \\
\text{return } T
\]

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\((a, P, b, Q)\):
  
  \[ S \leftarrow P + Q \]
  \[ T \leftarrow 0 \]
  
  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

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

▶ 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:
\begin{itemize}
\item \(n\) doublings, and
\item \(3n/4\) additions on average
\end{itemize}
Multi-exponentiation technique

function double-scalar-mult(a, P, b, Q):
    S ← P + Q
    T ← 0
    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:

- take an ordinary elliptic curve with a known efficiently computable endomorphism \( \psi \) of small norm
- the characteristic polynomial of \( \psi \) 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 \( \chi_\psi(T) \mod \ell \) such that \( \psi(P) = \lambda P \), for any \( P \in G \)

⇒ \( \lambda \)-adic decomposition of scalar \( k \) as \( k \equiv k_0 + \lambda k_1 \mod \ell \) so that \( kP = k_0 P + k_1 \psi(P) \)

⇒ 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 \( \lambda = \pm \sqrt{-1} \mod \ell \)
GLV curves

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

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 \( \psi^2(x, y) = (x, -y) = -1 \cdot (x, y) \), its characteristic polynomial is \( \chi_\psi(T) = T^2 + 1 \) and \( \lambda = \pm \sqrt{-1} \pmod{\ell} \).
GLV curves

Proposed by Gallant, Lambert, and Vanstone in 2000:

- take an ordinary elliptic curve with a known efficiently computable endomorphism $\psi$ of small norm
- the characteristic polynomial of $\psi$ 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 $\psi$ of small norm
- the characteristic polynomial of $\psi$ 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 $\chi_\psi(T) \mod \ell$ 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 $\psi$ of small norm
- the characteristic polynomial of $\psi$ 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 $\chi_\psi(T) \mod \ell$ such that

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

$\Rightarrow \lambda$-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 $\psi$ of small norm
• the characteristic polynomial of $\psi$ 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 $\chi_\psi(T) \mod \ell$ such that

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

$\Rightarrow$ $\lambda$-adic decomposition of scalar $k$ as $k \equiv k_0 + \lambda k_1 \mod \ell$ 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 $\psi$ of small norm
  - the characteristic polynomial of $\psi$ 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 $\chi_\psi(T) \mod \ell$ such that
    \[ \psi(P) = \lambda P, \text{ for any } P \in \mathbb{G} \]

  $\Rightarrow \lambda$-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 $\psi$ of small norm
  - the characteristic polynomial of $\psi$ 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 $\chi_\psi(T) \mod \ell$ such that
    \[
    \psi(P) = \lambda P, \text{ for any } P \in \mathbb{G}
    \]

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

\[
kP = k_0 P + k_1 \psi(P)
\]

⇒ 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 \( \psi \) of small norm
- the characteristic polynomial of \( \psi \) 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 \( \ell \) such that

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

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

\[
kP = k_0P + k_1\psi(P)
\]

\( \Rightarrow \) compute \( k_0P + k_1\psi(P) \) 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 \( \lambda = \pm \sqrt{-1} \pmod{\ell} \)
GLV curves

- Computation of $k_0$ and $k_1$:

- \( \{ (a, b) \in \mathbb{Z}^2 : a + b \lambda \equiv 0 \pmod{\ell} \} \) forms a 2-dimensional lattice \( \Lambda \)

- \( \Lambda \) is generated by \( (\ell, 0) \) and \( (-\lambda, 1) \)

- Precompute short basis (EEA)

- Given \( k \), find lattice point \((\tilde{k}_0, \tilde{k}_1) \) closest to \((k, 0)\)

- \( k \equiv k - (\tilde{k}_0 + \tilde{k}_1 \lambda) \pmod{\ell} \)

- \( k \equiv (k - \tilde{k}_0) + (-\tilde{k}_1 \lambda) \pmod{\ell} \)

- Take \( k_0 = (k - \tilde{k}_0) \mod \ell \) and \( k_1 = -\tilde{k}_1 \mod \ell \)

- Previous example with \( p = 953 \) and \( \mathbb{F}_p \):

- \( y^2 = x^3 + 5 \):

- 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 \( \psi(\lambda P) = \lambda P \) for all \( P \in G \), 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 \( \psi : (x, y) \mapsto (x^2, y^2) \)
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 $\Lambda$
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 $\Lambda$
- $\Lambda$ is generated by $(\ell, 0)$ and $(-\lambda, 1)$ → precompute short basis (EEA)

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 $\psi$: $(x, y) \mapsto (x^2, y^2)$
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 \(\Lambda\)
- \(\Lambda\) 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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$

\[
k \equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) \pmod{\ell}
\]

\[
\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda \pmod{\ell}
\]
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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$
  
  $k \equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) \pmod{\ell}$
  $\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda \pmod{\ell}$

- take $k_0 = (k - \tilde{k}_0) \mod \ell$ and $k_1 = -\tilde{k}_1 \mod \ell$
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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$

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

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

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

$\Rightarrow k_0$ 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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$

\[
\begin{align*}
    k &\equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) \pmod{\ell} \\
    &\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda \pmod{\ell}
\end{align*}
\]

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

\[\Rightarrow k_0 \text{ and } k_1 \text{ of size } \approx n/2 \text{ bits}\]

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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$
  \[
  k \equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) \pmod{\ell}
  \equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda \pmod{\ell}
  \]
- take $k_0 = (k - \tilde{k}_0) \mod{\ell}$ and $k_1 = -\tilde{k}_1 \mod{\ell}$
  $\Rightarrow$ $k_0$ 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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$

\[
\begin{align*}
k \equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) & \pmod{\ell} \\
& \equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda \pmod{\ell}
\end{align*}
\]

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

$\Rightarrow$ $k_0$ 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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$

\[
k \equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) \pmod{\ell}
\]

\[
\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda \pmod{\ell}
\]

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

$\Rightarrow k_0$ 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 $P \in G$, 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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$

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

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

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

$\Rightarrow$ $k_0$ 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 $\Lambda$
- $\Lambda$ 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 $(k, 0)$

\[
k \equiv k - (\tilde{k}_0 + \tilde{k}_1\lambda) \pmod{\ell}
\]

\[
\equiv (k - \tilde{k}_0) + (-\tilde{k}_1)\lambda \pmod{\ell}
\]

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

$\Rightarrow k_0$ 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

\[
kP = 2P + 7\psi(P)
\]

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:

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

At step $i$, point addition $T ← T + P$ is computed if and only if $k_i = 1$. Careful timing analysis will reveal the Hamming weight of the 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 $k_i = 1$. ⇒ Vulnerable to fault attacks.
Security issues

Back to the double-and-add algorithm:

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

  - At step \( i \), point addition \( T ← T + P \) 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:

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

At step i, point addition $T ← T + P$ 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:

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

- At step i, point addition T ← T + P is computed if and only if ki = 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:

```python
function scalar-mult(k, P):
    T ← 0
    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 $T ← T + P$ 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:

\[
\text{function scalar-mult}(k, P): \\
\quad T \leftarrow 0 \\
\quad \text{for } i \leftarrow n - 1 \text{ downto } 0: \\
\quad \quad T \leftarrow 2T \\
\quad \quad \text{if } k_i = 1: \\
\quad \quad \quad T \leftarrow T + P \\
\quad \quad \text{else:} \\
\quad \quad \quad Z \leftarrow T + P \\
\quad \text{return } T
\]

▶ 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 ← 0
      for i ← n − 1 downto 0:
          T ← 2T
          if ki = 1:
              T ← T + P
          else:
              Z ← T + P
      return T
  ```

  - At step i, point addition $T ← T + P$ 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 $k$

  ![Graph showing power consumption over time](image)

- Use **double-and-add-always** algorithm?
  - the result of the point addition is used if and only if $k_i = 1$
    ⇒ vulnerable to fault attacks
The Montgomery ladder

▶ Algorithm proposed by Montgomery in 1987:

\[
\text{function } \text{scalar-mult}(k, P):
\]

\[
T_0 \leftarrow \mathcal{O}
\]

\[
T_1 \leftarrow P
\]

\[
\text{for } i \leftarrow n - 1 \text{ downto } 0:
\]

\[
\text{if } k_i = 1:
\]

\[
T_0 \leftarrow T_0 + T_1
\]

\[
T_1 \leftarrow 2T_1
\]

\[
\text{else:}
\]

\[
T_1 \leftarrow T_0 + T_1
\]

\[
T_0 \leftarrow 2T_0
\]

\[
\text{return } T_0
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

function scalar-mult\((k, P)\):
\[
T_0 \leftarrow \mathcal{O} \\
T_1 \leftarrow P \\
\text{for } i \leftarrow n - 1 \text{ downto } 0:
\]
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:
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

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

Properties:
- perform one addition and one doubling at each step
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

function scalar-mult($k, P$):
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    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
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

```plaintext
function scalar-mult(k, P):
    T₀ ← Ø
    T₁ ← P
    for i ← n − 1 downto 0:
        if kᵢ = 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: \( T₁ = T₀ + P \)
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

```plaintext
function scalar-mult(k, P):
    \( T_0 \leftarrow O \)
    \( T_1 \leftarrow P \)
    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:

```plaintext
function scalar-mult(k, P):
    T₀ ← O
    T₁ ← P
    for i ← n − 1 downto 0:
        if kᵢ = 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:  \( T₁ = T₀ + P \)

Example:  \( k = 19 = (10011)₂ \)
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

\[
\text{function scalar-mult}(k, P):
\]

\[
T_0 \leftarrow \mathcal{O}
\]

\[
T_1 \leftarrow P
\]

for \( i \leftarrow n - 1 \) downto 0:

\[
\text{if } k_i = 1:
\]

\[
T_0 \leftarrow T_0 + T_1
\]

\[
T_1 \leftarrow 2T_1
\]

\[
\text{else:}
\]

\[
T_1 \leftarrow T_0 + T_1
\]

\[
T_0 \leftarrow 2T_0
\]

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: \( k = 19 = (10011)_2 \)

\[
T_0 = \mathcal{O}
\]

\[
T_1 = P
\]
The Montgomery ladder

➤ Algorithm proposed by Montgomery in 1987:

```plaintext
function scalar-mult(k, P):
    T₀ ← O
    T₁ ← P
    for i ← n − 1 downto 0:
        if kᵢ = 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: \( T₁ = T₀ + P \)

➤ Example: \( k = 19 = (10011)_2 \)

\[
\begin{align*}
    T₀ &= = O \\
    T₁ &= = P
\end{align*}
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

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

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: \( k = 19 = (10011)_2 \)

\[
T_0 = P \\
T_1 = P
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    \( T_0 \leftarrow \mathcal{O} \)
    \( T_1 \leftarrow P \)
    \( \text{for } i \leftarrow n - 1 \text{ downto } 0: \)
        \( \text{if } k_i = 1: \)
            \( T_0 \leftarrow T_0 + T_1 \)
            \( T_1 \leftarrow 2T_1 \)
        \( \text{else:} \)
            \( T_1 \leftarrow T_0 + T_1 \)
            \( T_0 \leftarrow 2T_0 \)
    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: \( k = 19 = (10011)_2 \)

\[
T_0 = P = P \\
T_1 = P \cdot 2 = 2P
\]
The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```plaintext
function scalar-mult(k, P):
    T₀ ← Ø
    T₁ ← P
    for i ← n − 1 downto 0:
        if kᵢ = 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: \( T₁ = T₀ + P \)

- Example: \( k = 19 = (10011)₂ \)

\[
\begin{align*}
T₀ &= P \\
T₁ &= P \cdot 2 = 2P
\end{align*}
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P):
  \( T_0 \leftarrow O \)
  \( T_1 \leftarrow P \)
  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: \( k = 19 = (10011)_2 \)

\[
\begin{align*}
T_0 &= P \\
T_1 &= P \cdot 2 + P = 3P
\end{align*}
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

```plaintext
function scalar-mult(k, P):
    T₀ ← O
    T₁ ← P
    for i ← n − 1 downto 0:
        if kᵢ = 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: \( T₁ = T₀ + P \)

Example: \( k = 19 = (10011)₂ \)

\[
T₀ = P \cdot 2 = 2P
\]
\[
T₁ = P \cdot 2 + P = 3P
\]
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: \( k = 19 = (10011)_2 \)

\[ T_0 = P \cdot 2 = 2P \]
\[ T_1 = P \cdot 2 + P = 3P \]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← 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: \( k = 19 = (10011)_2 \)

\[
T_0 = P \cdot 2 = 2P
\]
\[
T_1 = P \cdot 2 + P + 2P = 5P
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

\[
\text{function scalar-mult}(k, P): \\
\begin{align*}
T_0 & \leftarrow 0 \\
T_1 & \leftarrow P
\end{align*}
\]

for \( i \leftarrow n - 1 \) downto 0:

if \( k_i = 1 \):

\[
\begin{align*}
T_0 & \leftarrow T_0 + T_1 \\
T_1 & \leftarrow 2T_1
\end{align*}
\]

else:

\[
\begin{align*}
T_1 & \leftarrow T_0 + T_1 \\
T_0 & \leftarrow 2T_0
\end{align*}
\]

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: \( k = 19 = (10011)_2 \)

\[
\begin{align*}
T_0 & = P \cdot 2^2 = 4P \\
T_1 & = P \cdot 2 + P + 2P = 5P
\end{align*}
\]
The Montgomery ladder

▶ Algorithm proposed by Montgomery in 1987:

\[
\text{function scalar-mult}(k, P):
\]

\[
T_0 \leftarrow O
\]

\[
T_1 \leftarrow P
\]

\[
\text{for } i \leftarrow n - 1 \text{ downto } 0:
\]

\[
\text{if } k_i = 1:
\]

\[
T_0 \leftarrow T_0 + T_1
\]

\[
T_1 \leftarrow 2T_1
\]

\[
\text{else:}
\]

\[
T_1 \leftarrow T_0 + T_1
\]

\[
T_0 \leftarrow 2T_0
\]

\[
\text{return } T_0
\]

▶ 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 ← 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: $k = 19 = (10011)_2$

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

Algorithm proposed by Montgomery in 1987:

```plaintext
function scalar-mult(k, P):
    T₀ ← Ø
    T₁ ← P
    for i ← n − 1 downto 0:
        if kᵢ = 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: \( T₁ = T₀ + P \)

Example: \( k = 19 = (10011)_2 \)

\[
T₀ = P \cdot 2^2 + 5P = 9P \\
T₁ = (P \cdot 2 + P + 2P) \cdot 2 = 10P
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

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

\[
\begin{align*}
T_0 & \leftarrow \mathcal{O} \\
T_1 & \leftarrow P \\
\text{for } i & \leftarrow n - 1 \text{ downto 0:} \\
\text{if } k_i = 1: & \\
& \quad T_0 \leftarrow T_0 + T_1 \\
& \quad T_1 \leftarrow 2T_1 \\
\text{else:} & \\
& \quad T_1 \leftarrow T_0 + T_1 \\
& \quad T_0 \leftarrow 2T_0 \\
\text{return } T_0
\end{align*}
\]

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: \(k = 19 = (10011)_{2}\)

\[
\begin{align*}
T_0 & = P \cdot 2^2 + 5P = 9P \\
T_1 & = (P \cdot 2 + P + 2P) \cdot 2 = 10P
\end{align*}
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

```latex
\begin{align*}
    \textbf{function} & \text{ scalar-mult}(k, P): \\
    & T_0 \leftarrow O \\
    & T_1 \leftarrow P \\
    \text{for } i \leftarrow n - 1 \text{ downto 0:} \\
    & \quad \text{if } k_i = 1: \\
    & \qquad T_0 \leftarrow T_0 + T_1 \\
    & \qquad T_1 \leftarrow 2T_1 \\
    & \quad \text{else:} \\
    & \qquad T_1 \leftarrow T_0 + T_1 \\
    & \qquad T_0 \leftarrow 2T_0 \\
    \text{return } & T_0
\end{align*}
```

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: $k = 19 = (1001\underline{1})_2$

\[
\begin{align*}
    T_0 & = P \cdot 2^2 + 5P + 10P \quad = 19P \\
    T_1 & = (P \cdot 2 + P + 2P) \cdot 2 \quad = 10P
\end{align*}
\]
The Montgomery ladder

Algorithm proposed by Montgomery in 1987:

```plaintext
function scalar-mult(k, P):
  \( T_0 \leftarrow O \)
  \( T_1 \leftarrow P \)
  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: \( k = 19 = (10011)_2 \)

\[
T_0 = P \cdot 2^2 + 5P + 10P = 19P \\
T_1 = (P \cdot 2 + P + 2P) \cdot 2^2 = 20P
\]
The Montgomery ladder

▶ Algorithm proposed by Montgomery in 1987:

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

$T_0 \leftarrow \emptyset$
$T_1 \leftarrow P$

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: $k = 19 = (10011)_2$

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

$T_1 = (P \cdot 2 + P + 2P) \cdot 2^2 = 20P$
More security issues

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

\[
\begin{align*}
T_0 & \leftarrow \mathcal{O} \\
T_1 & \leftarrow P \\
\text{for } i & \leftarrow n - 1 \text{ downto 0:} \\
\quad \text{if } k_i = 1: \\
\quad & \quad \text{\(T_0 \leftarrow T_0 + T_1\)} \\
\quad & \quad \text{\(T_1 \leftarrow 2T_1\)} \\
\quad \text{else:} \\
\quad & \quad \text{\(T_1 \leftarrow T_0 + T_1\)} \\
\quad & \quad \text{\(T_0 \leftarrow 2T_0\)} \\
\end{align*}
\]

return \(T_0\)
More security issues

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

- The conditional branches depend on the value of secret bit $k_i$.
More security issues

function scalar-mult(k, P):
    \( T_0 \leftarrow \emptyset \)
    \( T_1 \leftarrow P \)
    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 \( k_i \)
⇒ might be vulnerable to branch prediction attacks
More security issues

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

\[
T_0 \leftarrow O \\
T_1 \leftarrow P \\
\text{for } i \leftarrow n - 1 \text{ downto } 0: \\
\quad T_{1-k_i} \leftarrow T_0 + T_1 \\
\quad T_{k_i} \leftarrow 2T_{k_i} \\
\text{return } T_0
\]

- 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\((k, P)\):
\[
\begin{align*}
T_0 & \leftarrow O \\
T_1 & \leftarrow P \\
\text{for } i & \leftarrow n - 1 \ \text{downto } 0: \\
T_{1-k_i} & \leftarrow T_0 + T_1 \\
T_{k_i} & \leftarrow 2T_{k_i} \\
\text{return } & T_0
\end{align*}
\]

- The conditional branches depend on the value of secret bit \(k_i\) ⇒ 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 $k_i$
⇒ 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 $k_i$
⇒ might be vulnerable to cache attacks
More security issues

function scalar-mult(k, P):
    \( T_0 \leftarrow \emptyset \)
    \( T_1 \leftarrow P \)
    for \( i \leftarrow n - 1 \) downto 0:
        \( M \leftarrow (k_i \ldots k_1)_2 \)
        \( R \leftarrow T_0 + T_1 \)
        \( S \leftarrow 2((M \& T_0) \mid (M \& T_1)) \)
        \( T_0 \leftarrow (M \& S) \mid (M \& R) \)
        \( T_1 \leftarrow (M \& R) \mid (M \& S) \)
    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

▶ Use bit masking to avoid secret-dependent memory access patterns
I. Scalar multiplication

II. Elliptic curve arithmetic

III. Finite field arithmetic

IV. Software considerations

V. Notions of hardware design
Addition and doubling


draw a graph with a red line and a red circle

R = P + Q

R' = 2P
Addition and doubling
Addition and doubling

$$R = P + Q$$

$$L_{P,Q}$$

$$E$$

$$O$$

$$P$$

$$Q$$
Addition and doubling

\[ R = P + Q \]

\[ L_{P,Q} \]

\[ R' = 2P \]

\[ O \]

\[ E \]
Addition and doubling

\[ R = P + Q \]

\[ L_{R', O} \]

\[ L_{P, Q} \]
Addition and doubling

$R = P + Q$

$L_{P,Q}$

$L_{R',O}$

$O$
Addition and doubling

Jérémie Detrey — Software and Hardware Implementation of Elliptic Curve Cryptography
Addition and doubling

\[
\begin{align*}
L_{P, P} &= P + P \\
R &= P + Q \\
\end{align*}
\]
Addition and doubling

\[ R = P + Q \]

\[ R = 2P \]
Addition and doubling

\[ R = P + Q \]

\[ R' = 2P \]
Addition and doubling

Addition in elliptic curve cryptography involves adding two points on the curve, while doubling involves adding a point to itself. The diagram illustrates these operations.

- **Addition**: The sum of two points $P$ and $Q$ is another point $R = P + Q$.
- **Doubling**: The sum of a point $P$ to itself is $R = 2P$.

The tangent line at $P$ intersects the curve at $R'$, and the line segment $PQ$ extended beyond $Q$ intersects the curve at $R = 2P$. These points are determined by solving the system of equations that define the elliptic curve.
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 \( Q = (x_Q, y_Q) \in E(\mathbb{F}_q) \backslash \{O\} \) (affine coordinates)
Addition and doubling formulae

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

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

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

  where

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

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

- Let \( P = (x_P, y_P) \) and \( Q = (x_Q, y_Q) \in E(\mathbb{F}_q) \setminus \{ O \} \) (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

\[
\begin{align*}
x_R &= \lambda^2 - x_P - x_Q \\
y_R &= \lambda(x_P - x_R) - y_P
\end{align*}
\]

where

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

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

\[ E/F_q : y^2 = x^3 + Ax + B \]

- Let \( P = (x_P, y_P) \) and \( Q = (x_Q, y_Q) \) \( \in E(F_q) \setminus \{O\} \) (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

\[
\begin{align*}
x_R &= \lambda^2 - x_P - x_Q \\
y_R &= \lambda(x_P - x_R) - y_P
\end{align*}
\]

where

\[
\lambda = \begin{cases} 
\frac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q, \\
\frac{-\left(\frac{\partial f}{\partial x}\right)(x_P, y_P)}{\left(\frac{\partial f}{\partial y}\right)(x_P, y_P)} &= \frac{3x_P^2 + a}{2y_P} & \text{if } P = Q
\end{cases}
\]

- Cost (number of inversions, multiplications and squares in \( F_q \)):
  - 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 \( Q = (x_Q, y_Q) \in E(\mathbb{F}_q) \setminus \{O\} \) (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

\[
x_R = \lambda^2 - x_P - x_Q \quad \text{and} \quad y_R = \lambda(x_P - x_R) - y_P
\]

where

\[
\lambda = \begin{cases} 
\frac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q, \text{ or} \\
-\frac{\partial f/\partial x}{\partial f/\partial y} (x_P, y_P) & \text{if } P = Q
\end{cases}
\]

- 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
  
  - Projective coordinates: points \((X:Y:Z)\) with \((x,y) = (X/Z,Y/Z)\)
  
  - Addition: 12M + 2S
  
  - Doubling: 7M + 5S
  
  - Jacobian coordinates: points \((X:Y:Z)\) with \((x,y) = (X/Z^2,Y/Z^3)\)
  
  - 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/]
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

- addition: \(12M + 2S\)
- doubling: \(7M + 5S\)

- 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/](http://hyperelliptic.org/EFD/)
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

- 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/](http://hyperelliptic.org/EFD/)
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: \(4M + 6S\)

- 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/
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 \((x, y) = (X/Z, Y/Z)\)

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

  \[ \text{http://hyperelliptic.org/EFD/} \]
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 \( \text{char}(\mathbb{F}_q) \neq 2 \).
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 \( \text{char}(\mathbb{F}_q) \neq 2 \)

- 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 \( \text{char}(\mathbb{F}_q) \neq 2 \)

  - 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
  \[ C/F_q : By^2 = x^3 + Ax^2 + x, \] with parameters \( A, B \in F_q \) and \( \text{char}(F_q) \neq 2 \)
  - 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 \( Q = (x_Q, y_Q) \in C(F_q)\backslash\{O\} \), 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 \( \text{char}(\mathbb{F}_q) \neq 2 \)

- 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 \( Q = (x_Q, y_Q) \in C(\mathbb{F}_q) \setminus \{O\} \), 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 \( \text{char}(\mathbb{F}_q) \neq 2 \)

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

\[ x_Rx_S(x_P - x_Q)^2 = (x_Px_Q - 1)^2 \]

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

- 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 \( Q = (x_Q, y_Q) \in C(\mathbb{F}_q) \setminus \{O\} \), 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
  \[ 4x_P x_R (x_P^2 + Ax_P + 1) = (x_P^2 - 1)^2 \]
Montgomery curves

- Proposed by Montgomery in 1987, Montgomery curves are of the form
  \[ C/F_q : By^2 = x^3 + Ax^2 + x, \] with parameters \( A, B \in F_q \) and \( \text{char}(F_q) \neq 2 \)

- 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 \( Q = (x_Q, y_Q) \in C(F_q)\setminus\{O\} \), with \( P \neq \pm Q \)
- then, writing \( R = P + Q = (x_R, y_R) \) and \( S = P - Q = (x_S, y_S) \), we have
  \[ x_Rx_S(x_P - x_Q)^2 = (x_Px_Q - 1)^2 \]

- 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
  \[ 4x_Px_R(x_P^2 + Ax_P + 1) = (x_P^2 - 1)^2 \]

  \( \Rightarrow \) \( x \)-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 \( \text{char}(\mathbb{F}_q) \neq 2 \)

- 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 \( Q = (x_Q, y_Q) \in C(\mathbb{F}_q) \setminus \{O\} \), 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
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 \( \text{char}(\mathbb{F}_q) \neq 2 \)

- 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 \( Q = (x_Q, y_Q) \in C(\mathbb{F}_q) \setminus \{O\} \), 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
  \[ 4x_P x_R (x_P^2 + A x_P + 1) = (x_P^2 - 1)^2 \]
  \( \Rightarrow \) \( x \)-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 \( \text{char}(\mathbb{F}_q) \neq 2 \)

- 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
    \[ 4x_P x_R (x_P^2 + A x_P + 1) = (x_P^2 - 1)^2 \]
  \( \Rightarrow \) \( x \)-only doubling

We can drop the \( y \)-coordinate altogether in the scalar multiplication

- use projective coordinates: points \((X : Z)\) with \( x = X/Z \)
- cheap differential addition \((4M + 2S)\) and doubling \((2M + 2S)\)
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 \( \text{char}(\mathbb{F}_q) \neq 2 \)

- 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 \( Q = (x_Q, y_Q) \in C(\mathbb{F}_q) \backslash \{O\} \), 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 \((4M + 2S)\) and doubling \((2M + 2S)\)
- compatible with the Montgomery ladder (since \( T_1 - T_0 = P \)
Proposed by Edwards in 2007, Edwards curves are of the form

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

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

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

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

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

▶ Addition and doubling formulae (assuming \( d \) is not a square in \( \mathbb{F}_q \))
Edwards curves

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

- 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

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

\[
P + Q = \left( \frac{x_Py_Q + x_Qy_P}{1 + dx_Px_Qy_Py_Q}, \frac{y_Py_Q - x_Px_Q}{1 - dx_Px_Qy_Py_Q} \right)
\]

- **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
    \[
    P + Q = \left( \frac{x_Py_Q + x_Qy_P}{1 + dx_Px_Qy_Py_Q}, \frac{y_Py_Q - x_Px_Q}{1 - dx_Px_Qy_Py_Q} \right)
    \]
  - **doubling**: same as addition

- **Strongly unified and complete** addition law:
  - works for both **addition** and **doubling**
  - no **exceptional case**
Edwards curves

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

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

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

- **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: \( 3M + 4S \)
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 \( \text{char}(\mathbb{F}_q) \neq 2 \)
  - 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 $\mathbb{F}_q$

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

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
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
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), $a_{k-1}, \ldots, a_1, a_0$:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0$$
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), $a_{k-1}$, ..., $a_1$, $a_0$:

\[
A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0
\]
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), \( a_{k-1}, \ldots, a_1, a_0 \):

\[
A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0
\]

▶ Addition of \( A \) and \( B \in \mathbb{F}_P \):

\[
\begin{array}{c}
a_3 & a_2 & a_1 & a_0 \\
+ & b_3 & b_2 & b_1 & b_0
\end{array}
\]
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), $a_{k-1}, \ldots, a_1, a_0$:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0$$

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), \( a_{k-1}, \ldots, a_1, a_0 \):

\[
A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0
\]

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), $a_{k-1}$, ..., $a_1$, $a_0$:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0$$

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

- right-to-left word-wise addition
- need to propagate carry

\[ \begin{array}{c}
\begin{array}{cccc}
 a_3 & a_2 & a_1 & a_0 \\
 b_3 & b_2 & b_1 & b_0 \\
\end{array}
+ \end{array} \Rightarrow \begin{array}{c}
\begin{array}{cc}
 c & r_0 \\
\end{array}\end{array} \]
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), \( a_{k-1}, \ldots, a_1, a_0 \):

\[
A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0
\]

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), $a_{k-1}, \ldots, a_1, a_0$:
    \[ A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0 \]

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

```
  +  

  \begin{array}{cccc}
    c  \\
    a_3 & a_2 & a_1 & a_0  \\
  \end{array}

  +  

  \begin{array}{cccc}
    b_3 & b_2 & b_1 & b_0  \\
  \end{array}

  \begin{array}{cccc}
    r_2 & r_1 & r_0  \\
  \end{array}
```
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), \( a_{k-1}, \ldots, a_1, a_0 \):
    \[
    A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0
    \]

- 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), $a_{k-1}$, ..., $a_1$, $a_0$:
    $$A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0$$

- 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), $a_{k-1}, \ldots, a_1, a_0$:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0$$

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), $a_{k-1}, \ldots, a_1, a_0$:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0$$

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), \( a_{k-1}, \ldots, a_1, a_0 \):

\[
A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0
\]

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!)

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  + \quad & b_3 & b_2 & b_1 & b_0 \\
  \hline
  c & r_3 & r_2 & r_1 & r_0 \\
  \hline
  - p_3 & p_2 & p_1 & p_0 \\
  \hline
  r'_3 & r'_2 & r'_1 & r'_0
\end{array}
\]
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), $a_{k-1}$, ..., $a_1$, $a_0$:
    $$A = a_{k-1}2^{(k-1)w} + \cdots + a_12^w + a_0$$

- 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

```
|   | a_3 | a_2 | a_1 | a_0 |
+---+-----+-----+-----+-----+
|   | b_3 | b_2 | b_1 | b_0 |
| c | r_3 | r_2 | r_1 | r_0 |
| p_3 | p_2 | p_1 | p_0 |
| r'_3 | r'_2 | r'_1 | r'_0 |
```
MP multiplication

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

$$\begin{array}{cccc}
a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
r_0 & r_1 & r_2 & r_3 & r_4 & r_5 & r_6 & r_7
\end{array}$$
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
Multiplication of $A$ and $B \in \mathbb{F}_p$:
- schoolbook method: $k^2$ $w$-by-$w$-bit multiplications

\[ a_3 \ a_2 \ a_1 \ a_0 \times \ b_3 \ b_2 \ b_1 \ b_0 \]
\[ \underline{a_2 b_0 \ a_0 b_0} \]
\[ \underline{a_3 b_0 \ a_1 b_0} \]
Multiplication of $A$ and $B \in \mathbb{F}_p$:
- schoolbook method: $k^2$ $w$-by-$w$-bit multiplications

\[
\begin{align*}
& a_3 \quad a_2 \quad a_1 \quad a_0 \\
\times & \quad b_3 \quad b_2 \quad b_1 \quad b_0 \\
& a_3 b_0 \quad a_1 b_0 \\
& a_2 b_1 \quad a_0 b_1 \\
& a_3 b_1 \quad a_1 b_1
\end{align*}
\]
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

\[
\begin{array}{c}
a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & a_2 b_0 & a_0 b_0 \\
 & a_3 b_0 & a_1 b_0 \\
 & a_2 b_1 & a_0 b_1 \\
 & a_3 b_1 & a_1 b_1 \\
 & a_2 b_2 & a_0 b_2 \\
 & a_3 b_2 & a_1 b_2 \\
 & a_2 b_3 & a_0 b_3 \\
 & a_3 b_3 & a_1 b_3 \\
\end{array}
\]
MP multiplication

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

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  b_3 & b_2 & b_1 & b_0 \\
\end{array}
\times
\begin{array}{cccc}
  a_2 b_0 & a_0 b_0 \\
  a_3 b_0 & a_1 b_0 \\
  a_2 b_1 & a_0 b_1 \\
  a_3 b_1 & a_1 b_1 \\
  a_2 b_2 & a_0 b_2 \\
  a_3 b_2 & a_1 b_2 \\
  a_2 b_3 & a_0 b_3 \\
  a_3 b_3 & a_1 b_3 \\
\end{array}
\]
MP multiplication

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

$$
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  \times & b_3 & b_2 & b_1 \\
  \hline
  a_2 b_0 & a_0 b_0 \\
  + & a_3 b_0 & a_1 b_0 \\
  + & a_2 b_1 & a_0 b_1 \\
  + & a_3 b_1 & a_1 b_1 \\
  + & a_2 b_2 & a_0 b_2 \\
  + & a_3 b_2 & a_1 b_2 \\
  + & a_2 b_3 & a_0 b_3 \\
  + & a_3 b_3 & a_1 b_3 \\
  \hline
  r_7 & r_6 & r_5 & r_4 & r_3 & r_2 & r_1 & r_0
\end{array}
$$
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 $P$)!

\[
\begin{array}{c}
\begin{array}{cccc}
a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\phantom{+}a_2 b_0 \\
\phantom{+}a_0 b_0 \\
\end{array} \\
\begin{array}{c}
\phantom{+}a_3 b_0 \\
\phantom{+}a_1 b_0 \\
\end{array} \\
\begin{array}{c}
\phantom{+}a_2 b_1 \\
\phantom{+}a_0 b_1 \\
\end{array} \\
\begin{array}{c}
\phantom{+}a_3 b_1 \\
\phantom{+}a_1 b_1 \\
\end{array} \\
\begin{array}{c}
\phantom{+}a_2 b_2 \\
\phantom{+}a_0 b_2 \\
\end{array} \\
\begin{array}{c}
\phantom{+}a_3 b_2 \\
\phantom{+}a_1 b_2 \\
\end{array} \\
\begin{array}{c}
\phantom{+}a_2 b_3 \\
\phantom{+}a_0 b_3 \\
\end{array} \\
\begin{array}{c}
\phantom{+}a_3 b_3 \\
\phantom{+}a_1 b_3 \\
\end{array} \\
\begin{array}{c}
r_7 \\
r_6 \\
r_5 \\
r_4 \\
r_3 \\
r_2 \\
r_1 \\
r_0 \\
\end{array}
\end{array}
\]
MP multiplication: operand vs. product scanning

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

```
   a3   a2   a1   a0
   b3   b2   b1   b0
```

MP multiplication: operand vs. product scanning

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

- operand scanning

$$\begin{array}{c}
a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0
\end{array}$$
MP multiplication: operand vs. product scanning

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

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  b_3 & b_2 & b_1 & b_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
  a_2 b_0 & a_0 b_0 \\
  a_3 b_0 & a_1 b_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
  r_4 & r_3 & r_2 & r_1 & r_0 \\
\end{array}
\]
MP multiplication: operand vs. product scanning

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

- operand scanning

$$
\begin{array}{c}
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  b_3 & b_2 & b_1 & b_0 \\
\end{array}
\end{array}
$$

$$
\begin{array}{c}
\begin{array}{c}
  a_2 b_0 & a_0 b_0 \\
  + & a_3 b_0 & a_1 b_0 \\
  + & a_2 b_1 & a_0 b_1 \\
  + & a_3 b_1 & a_1 b_1 \\
\end{array}
\end{array}
$$

$$
\begin{array}{c}
\begin{array}{cccc}
  r_5 & r_4 & r_3 & r_2 & r_1 & r_0 \\
\end{array}
\end{array}
$$
MP multiplication: operand vs. product scanning

In which order should we compute the subproducts \( a_i b_j \)?

- operand scanning

\[
\begin{align*}
\begin{array}{c}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\end{array}
\quad \times 
\begin{array}{c}
b_0 \\
b_1 \\
b_2 \\
b_3 \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
a_0 b_0 \\
a_1 b_0 \\
a_2 b_0 \\
a_3 b_0 + a_1 b_0 \\
a_2 b_1 + a_0 b_1 \\
a_3 b_1 + a_1 b_1 \\
a_2 b_2 + a_0 b_2 \\
a_3 b_2 + a_1 b_2 \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
r_0 \\
r_1 \\
r_2 \\
r_3 \\
r_4 \\
r_5 \\
r_6 \\
\end{array}
\end{align*}
\]
MP multiplication: operand vs. product scanning

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

- **Operand scanning**: straightforward, regular loop control

```
\begin{array}{cccc}
a_3 & a_2 & a_1 & a_0 \\
b_3 & b_2 & b_1 & b_0 \\
\end{array}
\times
\begin{array}{cccc}
\hline
 a_2 b_0 & a_0 b_0 \\
 a_3 b_0 & a_1 b_0 \\
 a_2 b_1 & a_0 b_1 \\
 a_3 b_1 & a_1 b_1 \\
 a_2 b_2 & a_0 b_2 \\
 a_3 b_2 & a_1 b_2 \\
 a_2 b_3 & a_0 b_3 \\
 a_3 b_3 & a_1 b_3 \\
\hline
 r_7 & r_6 & r_5 & r_4 \\
r_3 & r_2 & r_1 & r_0 \\
\end{array}
```
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 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
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 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

![Diagram of subproducts]

\[
\begin{align*}
\text{operand scanning:} & \quad (a_3 b_3) + (a_2 b_2) + (a_1 b_1) + (a_0 b_0) \\
\text{product scanning:} & \quad (a_3 b_0) + (a_2 b_1) + (a_1 b_2) + (a_0 b_3)
\end{align*}
\]
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 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

\[
\begin{array}{cccc}
\text{Operand scanning:} & a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
& a_2 b_0 & a_0 b_0 \\
+ & a_3 b_0 & a_1 b_0 \\
+ & a_2 b_1 & a_0 b_1 \\
+ & a_3 b_1 & a_1 b_1 \\
+ & a_2 b_2 & a_0 b_2 \\
+ & a_1 b_2 \\
+ & a_0 b_3 \\
+ & a_1 b_3 \\
\end{array}
\]
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.
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.
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

\[
\begin{array}{cccc}
\hline
a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
\end{array}
\]
MP multiplication: operand vs. product scanning

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

- **Operand scanning**: straightforward, regular loop control
- **Product scanning**: fewer memory accesses and carry propagations
- Many variants, such as left-to-right

```
a_3  a_2  a_1  a_0
\times b_3  b_2  b_1  b_0
\hline
a_2 b_0  a_0 b_0
+ a_3 b_0  a_1 b_0
+ a_2 b_1  a_0 b_1
+ a_3 b_1  a_1 b_1
+ a_2 b_2  a_0 b_2
+ a_3 b_2  a_1 b_2
+ a_2 b_3  a_0 b_3
+ a_3 b_3  a_1 b_3
\hline
r_7  r_6  r_5  r_4  r_3  r_2  r_1  r_0
```
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 $R = A \mod P$

![Diagram showing the division of $A$ and $P$]
MP modular reduction

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

\[ R' \leftarrow c \cdot A_H + A_L \quad \text{(one 1×w-word multiplication)} \]

\[ R'' \leftarrow A'_{H,H} + A'_{L,L} \quad \text{(one 1×1-word multiplication)} \]

Final subtraction might be necessary.

\[ A \equiv R' \mod P \]
MP modular reduction

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

- Easy case: $P$ is a pseudo-Mersenne prime $P = 2^n - c$ with $c$ “small” (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$

![Diagram of MP modular reduction](image)
MP modular reduction

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

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 $R = A \mod P$

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

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

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 \( 1 \times 1 \)-word multiplication)
Given an integer $A < P^2$ (on $2k$ words), compute $R = A \mod P$

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 $1 \times 1$-word multiplication)
Given an integer $A < P^2$ (on $2k$ words), compute $R = A \mod P$.

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 $1 \times 1$-word multiplication)
MP modular reduction

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

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 $1 \times 1$-word multiplication)
- final subtraction might be necessary
MP modular reduction

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

- 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 $1 \times 1$-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!

- Barrett reduction:
  - precompute $P' = \lfloor 2^{2k}w / P \rfloor$ ($k$ 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 $\tilde{A} \leftarrow (\tilde{Q} \cdot P) \mod 2^{(k+1)w}$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A - \tilde{A}$ at most two extra subtractions
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:**
  - precompute \( P' = \lfloor 2^{2\ell} / P \rfloor \) (\( \ell \) words)
  - given \( A < P^2 \), get the \( \ell + 1 \) most significant words
    \( A_H \leftarrow \lfloor A / 2^{(\ell-1)\ell} \rfloor \)
  - compute \( \tilde{Q} \leftarrow \lfloor A_H \cdot P' / 2^{(\ell+1)\ell} \rfloor \) (one \( (\ell+1) \times \ell \)-word multiplication)
  - compute \( \tilde{A} \leftarrow (\tilde{Q} \cdot P) \mod 2^{(\ell+1)\ell} \) (one \( \ell \times \ell \)-word multiplication)
  - compute remainder \( R \leftarrow A - \tilde{A} \) **at most two extra subtractions**
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 $P' = \lfloor 2^{kw}/P \rfloor$ ($k$ 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 \( P' = \lfloor 2^{kw}/P \rfloor \) (\( k \) 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 $P' = \lfloor 2^{2kN}/P \rfloor$ ($k$ words)
  - given $A < P^2$, get the $k + 1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)N} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)N} \rfloor$ (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 $A_H = \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} = \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k + 1) \times k$-word multiplication)
  - compute $\tilde{A} = \tilde{Q} \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R = A - \tilde{A}$
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 $\tilde{A} \leftarrow (\tilde{Q} \cdot P) \mod 2^{(k+1)w}$ (one $k \times k$-word short multiplication)
  - compute remainder $R \leftarrow A - \tilde{A}$
  - at most two extra subtractions
Montgomery reduction (REDC): like Barrett, but on the least significant words

- requires $P$ odd (on $k$ words) and $A < 2^{kw}$
- precompute $P' \leftarrow (P - 1) \mod 2^{kw}$ (on $k$ words)
- given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (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 $R = (A \cdot 2^{-kw}) \mod P$, not $A \mod P$!

- represent $X \in \mathbb{F}_p$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \mod P$
- if $Z = (X \cdot Y) \mod P$, then REDC($\hat{X} \cdot \hat{Y}$) = $(X \cdot Y \cdot 2^{kw}) \mod P = \hat{Z}$

- that's the so-called Montgomery multiplication

- conversions: $\hat{X} = \text{REDC}(X, 2^{2kw} \mod P)$ and $X = \text{REDC}(\hat{X}, 1)$

- Montgomery representation is compatible with addition / subtraction in $\mathbb{F}_P$ ⇒ do all computations in Montgomery repr. instead of converting back and forth

- REDC can be computed iteratively (one word at a time) and interleaved with the computation of $\hat{X} \cdot \hat{Y}$
MP modular reduction: general case

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

\[ p_3 \quad p_2 \quad p_1 \quad p_0 \]
Montgomery reduction (REDC): like Barrett, but on the least significant words

- requires $P$ odd (on $k$ words) and $A < 2^{kw}P$
- precompute $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)

\[ p'_3 \quad p'_2 \quad p'_1 \quad p'_0 \]
Montgomery reduction (REDC): like Barrett, but on the least significant words

- requires $P$ odd (on $k$ words) and $A < 2^{kw} P$
- precompute $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
- given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (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 $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
  - given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (one $k \times k$-word short multiplication)

\[
\begin{array}{cccc}
p_0' & p_1' & p_2' & p_3' \\
a_7 & a_6 & a_5 & a_4 \\
\end{array}
\begin{array}{cccc}
a_0 & a_1 & a_2 & a_3 \\
\end{array}
\]

\[\hat{X} = (X \cdot 2^{kw}) \mod P\]

- conversions: $\hat{X} = \text{REDC}(X, 2^{2kw} \mod P)$ and $X = \text{REDC}(\hat{X}, 1)$
- Montgomery representation is compatible with addition / subtraction in $\mathbb{F}_P$ ⇒ do all computations in Montgomery repr. instead of converting back and forth
Montgomery reduction (REDC): like Barrett, but on the least significant words
- requires $P$ odd (on $k$ words) and $A < 2^{kw} P$
- precompute $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
- given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (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 $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
  - given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)

\[
\begin{array}{cccc}
\times & p_3 & p_2 & p_1 & p_0 \\
& a_7 & a_6 & a_5 & a_4 \\
\times & k_3 & k_2 & k_1 & k_0 \\
& k_7 & k_6 & k_5 & k_4 \\
\times & \tilde{p}_3 & \tilde{p}_2 & \tilde{p}_1 & \tilde{p}_0 \\
& \tilde{a}_7 & \tilde{a}_6 & \tilde{a}_5 & \tilde{a}_4 \\
\end{array}
\]
Montgomery reduction (REDC): like Barrett, but on the least significant words

- requires $P$ odd (on $k$ words) and $A < 2^{kw}P$
- precompute $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
- given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (one $k \times k$-word short multiplication)
- compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
- compute remainder $R \leftarrow A + \tilde{A}$

\[ \begin{array}{c|cccc|cccc|cccc} \times & a_7 & a_6 & a_5 & a_4 & p_3' & p_2' & p_1' & p_0' \\ \hline & k_7 & k_6 & k_5 & k_4 & a_3 & a_2 & a_1 & a_0 \\ \times & p_3 & p_2 & p_1 & p_0 & k_3 & k_2 & k_1 & k_0 \\ \hline & \tilde{a}_7 & \tilde{a}_6 & \tilde{a}_5 & \tilde{a}_4 & \tilde{a}_3 & \tilde{a}_2 & \tilde{a}_1 & \tilde{a}_0 \\ a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \end{array} \]
Montgomery reduction (REDC): like Barrett, but on the least significant words

- requires $P$ odd (on $k$ words) and $A < 2^{kw} P$
- precompute $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
- given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (one $k \times k$-word short multiplication)
- compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
- compute remainder $R \leftarrow A + \tilde{A}$

\[\begin{array}{c}
\times \\
\hline
a_7 & a_6 & a_5 & a_4 \\
\hline
k_7 & k_6 & k_5 & k_4 \\
\hline
\times \\
\hline
\tilde{a}_7 & \tilde{a}_6 & \tilde{a}_5 & \tilde{a}_4 \\
\hline
a_7 & a_6 & a_5 & a_4 \\
\hline
+ \\
\hline
r_7 & r_6 & r_5 & r_4 \\
\end{array}\]
Montgomery reduction (REDC): like Barrett, but on the least significant words

- requires \( P \) odd (on \( k \) words) and \( A < 2^{kw} P \)
- precompute \( P' \leftarrow (-P^{-1}) \mod 2^{kw} \) (on \( k \) words)
- given \( A \), compute \( K \leftarrow (A \cdot P') \mod 2^{kw} \) (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 $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
  - given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A + \tilde{A}$

\[
\begin{array}{c}
\times \quad a_7 & a_6 & a_5 & a_4 \\
k_7 & k_6 & k_5 & k_4 \\
\times \quad p'_3 & p'_2 & p'_1 & p'_0 \\
\tilde{a}_7 & \tilde{a}_6 & \tilde{a}_5 & \tilde{a}_4 \\
a_0 \quad a_1 \quad a_2 \quad a_3 \\
\tilde{a}_0 \quad \tilde{a}_1 \quad \tilde{a}_2 \quad \tilde{a}_3 \\
\times \quad p_3 & p_2 & p_1 & p_0 \\
ar_7 & r_6 & r_5 & r_4 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{array}
\]
Montgomery reduction (REDC): like Barrett, but on the least significant words

- requires $P$ odd (on $k$ words) and $A < 2^{kw} P$
- precompute $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
- given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (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 \( P' \leftarrow (-P^{-1}) \mod 2^{kw} \) (on \( k \) words)
  - given \( A \), compute \( K \leftarrow (A \cdot P') \mod 2^{kw} \) (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

\[
\begin{array}{c}
\times \\
\begin{array}{c}
| \quad \quad \quad | \\
| a_7 | a_6 | a_5 | a_4 | \\
| \quad \quad \quad |
\end{array}
\begin{array}{c}
| \quad \quad \quad | \\
| a_3 | a_2 | a_1 | a_0 | \\
| \quad \quad \quad |
\end{array}
\begin{array}{c}
| \quad \quad \quad | \\
| k_7 | k_6 | k_5 | k_4 | \\
| \quad \quad \quad |
\end{array}
\begin{array}{c}
| \quad \quad \quad | \\
| k_3 | k_2 | k_1 | k_0 | \\
| \quad \quad \quad |
\end{array}
\begin{array}{c}
| \quad \quad \quad | \\
| p_3 | p_2 | p_1 | p_0 | \\
| \quad \quad \quad |
\end{array}
\begin{array}{c}
| \quad \quad \quad | \\
| r_7 | r_6 | r_5 | r_4 | \\
\end{array}
\end{array}
\]
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 $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
  - given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (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 $R = (A \cdot 2^{-kw}) \mod P$, not $A \mod P$!
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 $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
  - given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (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 $R = (A \cdot 2^{-kw}) \mod P$, not $A \mod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \mod P$
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 \( P' \leftarrow (-P^{-1}) \mod 2^{kw} \) (on \( k \) words)
  
  - given \( A \), compute \( K \leftarrow (A \cdot P') \mod 2^{kw} \) (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 \( R = (A \cdot 2^{-kw}) \mod P \), not \( A \mod P \)!
  
  - represent \( X \in \mathbb{F}_P \) in Montgomery representation: \( \hat{X} = (X \cdot 2^{kw}) \mod P \)
  
  - if \( Z = (X \cdot Y) \mod P \), then

  \[
  \text{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \mod P = \hat{Z}
  \]

  → that’s the so-called **Montgomery multiplication**
Montgomery reduction (REDC): like Barrett, but on the least significant words
- requires $P$ odd (on $k$ words) and $A < 2^{kw}P$
- precompute $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
- given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (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 $R = (A \cdot 2^{-kw}) \mod P$, not $A \mod P$!
- represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \mod P$
- if $Z = (X \cdot Y) \mod P$, then
  $$\text{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \mod P = \hat{Z}$$
  → that’s the so-called Montgomery multiplication
- conversions:
  $$\hat{X} = \text{REDC}(X, 2^{2kw} \mod P) \quad \text{and} \quad X = \text{REDC}(\hat{X}, 1)$$
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 $P' \leftarrow (-P^{-1}) \mod 2^{kw}$ (on $k$ words)
  - given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{kw}$ (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 $R = (A \cdot 2^{-kw}) \mod P$, not $A \mod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \mod P$
  - if $Z = (X \cdot Y) \mod P$, then
    \[
    \text{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \mod P = \hat{Z}
    \]
    \quad → that’s the so-called Montgomery multiplication
  - conversions:
    \[
    \hat{X} = \text{REDC}(X, 2^{2kw} \mod P) \quad \text{and} \quad X = \text{REDC}(\hat{X}, 1)
    \]
  - 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 \( P' \leftarrow (-P^{-1}) \mod 2^{kw} \) (on \( k \) words)
- given \( A \), compute \( K \leftarrow (A \cdot P') \mod 2^{kw} \) (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 \( R = (A \cdot 2^{-kw}) \mod P \), not \( A \mod P \)!

- represent \( X \in \mathbb{F}_P \) in Montgomery representation: \( \hat{X} = (X \cdot 2^{kw}) \mod P \)
- if \( Z = (X \cdot Y) \mod P \), then
  \[
  \text{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \mod P = \hat{Z}
  \]
  → that’s the so-called Montgomery multiplication
- conversions:
  \[
  \hat{X} = \text{REDC}(X, 2^{2kw} \mod P) \quad \text{and} \quad X = \text{REDC}(\hat{X}, 1)
  \]
- Montgomery representation is compatible with addition / subtraction in \( \mathbb{F}_P \)
  ⇒ do all computations in Montgomery repr. instead of converting back and forth
Montgomery reduction (REDC): like Barrett, but on the least significant words

- requires $P$ odd (on $k$ words) and $A < 2^{k_w}P$
- precompute $P' \leftarrow (P^{-1}) \mod 2^{k_w}$ (on $k$ words)
- given $A$, compute $K \leftarrow (A \cdot P') \mod 2^{k_w}$ (one $k \times k$-word short multiplication)
- compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
- compute remainder $R \leftarrow (A + \tilde{A})/2^{k_w}$
- at most one extra subtraction

REDC($A$) returns $R = (A \cdot 2^{-k_w}) \mod P$, not $A \mod P$!

- represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{k_w}) \mod P$
- if $Z = (X \cdot Y) \mod P$, then
  \[
  \text{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{k_w}) \mod P = \hat{Z}
  \]
  $\rightarrow$ that’s the so-called Montgomery multiplication

- conversions:
  \[
  \hat{X} = \text{REDC}(X, 2^{2k_w} \mod P) \quad \text{and} \quad X = \text{REDC}(\hat{X}, 1)
  \]

- 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 $A^{-1} \mod P$
MP field inversion

Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \mod P$

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 $A^{-1} = U \mod P$
MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \mod P$

- 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 $A^{-1} = U \mod P$
  - can be adapted to Montgomery representation
MP field inversion

Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \mod P$

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 $A^{-1} = U \mod P$
- can be adapted to Montgomery representation
- fast, but running time depends on $A$

Fermat’s little theorem:
- we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
- 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 $A^{-1} \mod P$

- 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 $A^{-1} = U \mod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on $A$
  - requires randomization of $A$ to protect against timing attacks
MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \mod P$

- 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 $A^{-1} = U \pmod{P}$
  - can be adapted to Montgomery representation
  - fast, but running time depends on $A$
  - 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 $A^{-1} \mod P$

Extended Euclidean algorithm:
- compute Bézout’s coefficients: $U$ and $V$ such that $UA + VP = \gcd(A, P) = 1$
- then $UA \equiv 1 \mod P$ and $A^{-1} = U \mod P$
- can be adapted to Montgomery representation
- fast, but running time depends on $A$
  \[\Rightarrow\text{ requires randomization of } A \text{ to protect against timing attacks}\]

Fermat’s little theorem:
- we know that $A^{P-1} = 1 \mod P$, whence $A^{P-2} = A^{-1} \mod P$
- precompute short sequence of squarings and multiplications for fast exponentiation of $A$
MP field inversion

Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \mod P$

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 $A^{-1} \equiv U \pmod{P}$
- 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 $A^{P-1} \equiv 1 \pmod{P}$, 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 \( A^{-1} \mod P \)

**Extended Euclidean algorithm:**
- compute Bézout’s coefficients: \( U \) and \( V \) such that \( UA + VP = \gcd(A, P) = 1 \)
- then \( UA \equiv 1 \mod P \) and \( A^{-1} = U \mod P \)
- 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 \( A^{P-1} = 1 \mod P \), whence \( A^{P-2} = A^{-1} \mod P \)
- 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 $A^{-1} \mod P$

Extended Euclidean algorithm:
- compute Bézout’s coefficients: $U$ and $V$ such that $UA + VP = \gcd(A, P) = 1$
- then $UA \equiv 1 \mod P$ and $A^{-1} = U \mod P$
- can be adapted to Montgomery representation
- fast, but running time depends on $A$
  \[ \Rightarrow \text{requires randomization of } A \text{ to protect against timing attacks} \]

Fermat’s little theorem:
- we know that $A^{P-1} = 1 \mod P$, whence $A^{P-2} = A^{-1} \mod P$
- 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 \( A^{-1} \mod P \)

- 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 \( A^{-1} = U \mod P \)
  - 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]

\[
A \xrightarrow{S} A^2 \xrightarrow{S^2} A^9
\]
MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \mod P$

- Extended Euclidean algorithm:
  - compute Bézout’s coefficients: $U$ and $V$ such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \mod P$ and $A^{-1} = U \mod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on $A$
  - requires randomization of $A$ to protect against timing attacks

- Fermat’s little theorem:
  - we know that $A^{P-1} = 1 \mod P$, whence $A^{P-2} = A^{-1} \mod P$
  - precompute short sequence of squarings and multiplications for fast
    exponentiation of $A$
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

```
A → A^2 → A^9 → A^{11}
```
MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \mod P$

- 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 $A^{-1} = U \pmod{P}$
  - can be adapted to Montgomery representation
  - fast, but running time depends on $A$
  - 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 \( A^{-1} \mod P \)

Extended Euclidean algorithm:
- compute Bézout’s coefficients: \( U \) and \( V \) such that \( UA + VP = \gcd(A, P) = 1 \)
- then \( UA \equiv 1 \mod P \) and \( A^{-1} = U \mod P \)
- 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 \( A^{P-1} = 1 \mod P \), whence \( A^{P-2} = A^{-1} \mod P \)
- precompute short sequence of squarings and multiplications for fast exponentiation of \( A \)
- example: \( P = 2^{255} - 19 \) in 11M and 254S [Bernstein, 2006]

\[ A \rightarrow A^2 \rightarrow A^9 \rightarrow A^{11} \rightarrow A^{25-1} \rightarrow A^{210-1} \]
MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \mod P$

- 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 $A^{-1} = U \pmod{P}$
  - can be adapted to Montgomery representation
  - fast, but running time depends on $A$
  - 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 $A^{-1} \mod P$

- 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 $A^{-1} = U \mod P$
  - 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 $A^{-1} \mod P$

- 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 $A^{-1} = U \mod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on $A$
  ⇒ 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 $A^{-1} \mod P$

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 $A^{-1} = U \mod P$
- 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 \( A^{-1} \mod P \)

- Extended Euclidean algorithm:
  - compute Bézout’s coefficients: \( U \) and \( V \) such that \( UA + VP = \gcd(A, P) = 1 \)
  - then \( UA \equiv 1 \mod P \) and \( A^{-1} = U \mod P \)
  - 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 \( A^{P-1} = 1 \mod P \), whence \( A^{P-2} = A^{-1} \mod P \)
  - 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 $A^{-1} \mod P$

- **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 $A^{-1} = U \mod P$
  - can be adapted to Montgomery representation
  - fast, but running time depends on $A$
  \[ \Rightarrow \text{requires randomization of } A \text{ to protect against timing attacks} \]

- **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 $A^{-1} \mod P$

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 $A^{-1} = U \mod P$
- 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 $m_i$:
  $$m_i = 2^w - c_i,$$
  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 \( i \), \( M_i = M/m_i \)

- If \( P \leq M \), we can represent elements of \( \mathbb{F}_P \) in RNS
Let $B = (m_1, \ldots, m_k)$ 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 $m_i$:
  $$m_i = 2^w - c_i,$$
  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 \( a_i = A \mod m_i = |A|_{m_i} \), for all \( i \)
  → 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 $a_i = A \mod m_i = |A|_{m_i}$, 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):
  \[
  A \equiv \sum_{i=1}^{k} a_i \cdot M_i^{-1} \mod M
  \]
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 $\vec{A} = (a_1, \ldots, a_k)$ with $a_i = A \mod m_i = |A|_{m_i}$, for all $i$
  → that is the RNS representation of $A$ in base $\mathcal{B}$
- given $\vec{A} = (a_1, \ldots, a_k)$, retrieve the unique corresponding integer $A \in \mathbb{Z}/M\mathbb{Z}$ using the Chinese remaindering theorem (CRT):
  \[
  A = \left\lfloor \sum_{i=1}^{k} a_i \cdot M_i^{-1} \frac{|A|_{m_i}}{m_i \cdot M_i} \right\rfloor_M
  \]

If $P \leq M$, we can represent elements of $\mathbb{F}_P$ in RNS
Let $\vec{A} = (a_1, \ldots, a_k)$ and $\vec{B} = (b_1, \ldots, b_k)$.
Let $\vec{A} = (a_1, \ldots, a_k)$ and $\vec{B} = (b_1, \ldots, b_k)$.

- add., sub. and mult. can be performed in parallel on all “channels”:
  
  \[
  \vec{A} \pm \vec{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})
  \]
  
  \[
  \vec{A} \times \vec{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})
  \]
RNS arithmetic

Let $\vec{A} = (a_1, \ldots, a_k)$ and $\vec{B} = (b_1, \ldots, b_k)$

- add., sub. and mult. can be performed in parallel on all “channels”:
  \[
  \vec{A} \pm \vec{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k}) \\
  \vec{A} \times \vec{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})
  \]
RNS arithmetic

Let $\vec{A} = (a_1, \ldots, a_k)$ and $\vec{B} = (b_1, \ldots, b_k)$

- add., sub. and mult. can be performed in parallel on all “channels”:
  $$\vec{A} \pm \vec{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
  $$\vec{A} \times \vec{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$

- 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 divisions, modular reductions or comparisons
Let $\vec{A} = (a_1, \ldots, a_k)$ and $\vec{B} = (b_1, \ldots, b_k)$

- add., sub. and mult. can be performed in parallel on all “channels”:
  
  $\vec{A} \pm \vec{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$
  
  $\vec{A} \times \vec{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$

Limitations:
- operations are computed in $\mathbb{Z}/M\mathbb{Z}$: beware of overflows!
- no simple way to compute divisions, modular reductions or comparisons
RNS arithmetic

Let \( \vec{A} = (a_1, \ldots, a_k) \) and \( \vec{B} = (b_1, \ldots, b_k) \)

- add., sub. and mult. can be performed in parallel on all “channels”:
  - \( \vec{A} \pm \vec{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k}) \)
  - \( \vec{A} \times \vec{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k}) \)

- native parallelism: suited to SIMD instructions and hardware implementation
RNS arithmetic

Let \( \vec{A} = (a_1, \ldots, a_k) \) and \( \vec{B} = (b_1, \ldots, b_k) \)

- add., sub. and mult. can be performed in parallel on all “channels”:

\[
\vec{A} \pm \vec{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k}) \\
\vec{A} \times \vec{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})
\]

- 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 divisions, modular reductions or comparisons
RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS

From the CRT:

\[
A = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M^{-1}_i| m_i \cdot M_i \right\rfloor
\]

\[
M = \left( \sum_{i=1}^{k} |a_i \cdot M^{-1}_i| m_i \cdot M_i \right) - qM
\]

with \(0 \leq q < k\), whose actual value depends on \(A\)

Compute \(\tilde{q}\), approximation of \(q\):

\[
q = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M^{-1}_i| m_i \cdot M_i \right\rfloor \approx \frac{\left\lfloor \left( \sum_{i=1}^{w-t} |a_i \cdot M^{-1}_i| \right)^2 \right\rfloor}{2^t + \varepsilon}
\]

\[
\approx \frac{\sum_{i=1}^{w-t} c_i}{2^w - c_i}
\]

If \(0 \leq A < (1 - \varepsilon) M\), then \(\tilde{q} = q\) and

\[
A \mod P = \left( \left( \sum_{i=1}^{k} |a_i \cdot M^{-1}_i| m_i \cdot M_i \right) - |\tilde{q}M| \right) \mod P
\]
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 $P$

\[ A = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M - 1^i| \cdot m_i \cdot M^i \right\rfloor \]

\[ M = \left( \sum_{i=1}^{k} |a_i \cdot M - 1^i| \cdot m_i \cdot M^i \right) - q M \]

with $0 \leq q < k$, whose actual value depends on $A$

Compute $\tilde{q}$, approximation of $q$:

\[ \tilde{q} = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M - 1^i| \cdot m_i \cdot M^i \right\rfloor \approx \frac{1}{2} \sum_{i=1}^{k} \left\lfloor \frac{|a_i \cdot M - 1^i| \cdot m_i \cdot M^i}{2^w - t} \right\rfloor + \frac{1}{2} \sum_{i=1}^{k} c_i \quad \text{approximate} \quad m_i = 2^w - c_i \quad \text{by} \quad 2^w \quad \text{use only the} \quad t \quad \text{most significant bits of} \quad |a_i \cdot M - 1^i| \]

\[ \text{add fixed corrective term} \quad \frac{\sum c_i + k (2^w - t - 1)}{2^w} < \epsilon < 1 \]

If $0 \leq A < (1 - \epsilon) M$, then $\tilde{q} = q$ and

\[ A \mod P = \left( \left( \sum_{i=1}^{k} |a_i \cdot M - 1^i| \cdot m_i \cdot M^i \right) - \tilde{q} M \right) \mod P \]
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 $P$

- From the CRT:

  $$A = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M^{-1}_i| \cdot m_i \cdot M \right\rfloor_M$$
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 $P$

- From the CRT:

$$A = \left\lfloor \sum_{i=1}^{k} \left| a_i \cdot M_i^{-1} \right|_{m_i} \cdot M_i \right\rfloor_{M} = \left( \sum_{i=1}^{k} \left| a_i \cdot M_i^{-1} \right|_{m_i} \cdot M_i \right) - qM$$

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
  ⇒ Need to approximate CRT reconstruction and reduce it modulo $P$

- From the CRT:
  \[
  A = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i \cdot M_i}}{M} \right\rfloor = \left( \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i \cdot M_i}}{M} \right) - qM
  \]
  with $0 \leq q < k$, whose actual value depends on $A$

- Compute $\tilde{q}$, approximation of $q$:
  \[
  q = \left\lfloor \sum_{i=1}^{k} \frac{a_i \cdot M_i^{-1}}{M} \right\rfloor
  \]
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 $P$

- From the CRT:

$$
A = \left\lfloor \sum_{i=1}^{k} a_i \cdot M_i^{-1} \right\rfloor_{m_i \cdot M_i} = \left( \sum_{i=1}^{k} a_i \cdot M_i^{-1} \right)_{m_i \cdot M_i} - qM
$$

with $0 \leq q < k$, whose actual value depends on $A$

- Compute $\tilde{q}$, approximation of $q$:

$$
q = \left\lfloor \sum_{i=1}^{k} \frac{a_i \cdot M_i^{-1}}{M} \right\rfloor_{m_i \cdot M_i} = \left\lfloor \sum_{i=1}^{k} \frac{a_i \cdot M_i^{-1}}{m_i} \right\rfloor_{m_i}
$$
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 $P$

- From the CRT:
  $$A = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor = \left( \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right) - qM$$
  
  with $0 \leq q < k$, whose actual value depends on $A$

- Compute $\tilde{q}$, approximation of $q$:  
  $$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor = \left( \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{M} \right)$$

  - 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
  ⇒ Need to approximate CRT reconstruction and reduce it modulo $P$

- From the CRT:
  \[
  A = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M_i^{-1}| m_i \cdot M_i \right\rfloor = \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}| m_i \cdot M_i \right) - qM
  \]
  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
  ⇒ 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
  ⇒ Need to approximate CRT reconstruction and reduce it modulo $P$

- From the CRT:

\[
A = \left\lfloor \sum_{i=1}^{k} a_i \cdot M_i^{-1}|_{m_i \cdot M_i} \right\rfloor_{M} = \left( \sum_{i=1}^{k} a_i \cdot M_i^{-1}|_{m_i \cdot M_i} \right) - qM
\]

with $0 \leq q < k$, whose actual value depends on $A$

- Compute $\tilde{q}$, approximation of $q$:

\[
q = \left\lfloor \sum_{i=1}^{k} \frac{a_i \cdot M_i^{-1}|_{m_i \cdot M_i}}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}2^t} \right\rfloor
\]

- 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
  ⇒ Need to approximate CRT reconstruction and reduce it modulo $P$

- From the CRT:

$$A = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right\rfloor = \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$

with $0 \leq q < k$, whose actual value depends on $A$

- Compute $\tilde{q}$, approximation of $q$:

$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor + \varepsilon$$

  - 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 ($\sum_i c_i + k(2^{w-t} - 1))/2^w < \varepsilon < 1$
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 $P$

- 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 $q$:
  $$q = \left| \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right| \approx \left| \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right| + \varepsilon = \tilde{q}$$
  - 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
  ⇒ Need to approximate CRT reconstruction and reduce it modulo \( P \)

- From the CRT:

\[
A = \left| \sum_{i=1}^{k} a_i \cdot M_i^{-1} \right|_{m_i} \cdot M_i \mid M = \left( \sum_{i=1}^{k} a_i \cdot M_i^{-1} \right)_{m_i} \cdot M_i - qM
\]

with \( 0 \leq q < k \), whose actual value depends on \( A \)

- Compute \( \tilde{q} \), approximation of \( q \):

\[
q = \left[ \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right] \approx \left[ \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right] = \tilde{q}
\]

- 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 \( (\sum_i c_i + k(2^{w-t} - 1))/2^w < \varepsilon < 1 \)

- 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
  ⇒ Need to approximate CRT reconstruction and reduce it modulo $P$

- From the CRT:

  $$A = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right\rfloor = \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - qM$$

  with $0 \leq q < k$, whose actual value depends on $A$

- Compute $\tilde{q}$, approximation of $q$:

  $$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \cdot \frac{2^t}{2^t} + \varepsilon \right\rfloor = \tilde{q}$$

  - 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 $(\sum_i c_i + k(2^{w-t} - 1)) / 2^w < \varepsilon < 1$

- If $0 \leq A < (1 - \varepsilon)M$, then $\tilde{q} = q$ and

  $$A \mod P = \left( \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right) - \tilde{q}M \right) \mod P$$
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 $P$

- From the CRT:

$$A = \left\lfloor \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i \cdot M_i} \right\rfloor = \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i \cdot M_i} \right) - qM$$

with $0 \leq q < k$, whose actual value depends on $A$

- Compute $\tilde{q}$, approximation of $q$:

$$q = \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i \cdot M_i}}{M} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right\rfloor + \varepsilon = \tilde{q}$$

  - 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 $(\sum_i c_i + k(2^{w-t} - 1))/2^w < \varepsilon < 1$

- If $0 \leq A < (1 - \varepsilon)M$, then $\tilde{q} = q$ and

$$A \mod P = \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i \cdot |M_i|_P} \right) - |\tilde{q}M|_P$$
RNS modular reduction

- Not a positional number system: no equivalent of pseudo-Mersenne primes in RNS

  ⇒ 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 $q$:

\[
q = \left| \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i} \cdot M_i}{M} \right| \approx \left| \sum_{i=1}^{k} \frac{|a_i \cdot M_i^{-1}|_{m_i}}{2^{w-t}} \right| + \varepsilon = \tilde{q}
\]

  - 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 $(\sum_i c_i + k(2^w-t - 1))/2^w < \varepsilon < 1$

- If $0 \leq A < (1 - \varepsilon)M$, then $\tilde{q} = q$ and

\[
A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P}
\]
RNS modular reduction

\[ A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P} \]

**function** `reduce-mod-P(\vec{A})`:

\[ (\forall i) \quad z_i \leftarrow |a_i \cdot |M_i^{-1}|_{m_i}|_{m_i} \]

\[ (\forall i) \quad \tilde{z}_i \leftarrow |z_i/2^{w-t}| \]

\[ \tilde{q} \leftarrow |\sum_i \tilde{z}_i/2^t + \varepsilon| \]

\[ (\forall i) \quad r_i \leftarrow 0 \]

**for** \( j \leftarrow 1 \) **to** \( k \):

\[ (\forall i) \quad r_i \leftarrow |r_i + z_j \cdot |M_j|_P|_{m_i}|_{m_i} \]

\[ (\forall i) \quad r_i \leftarrow |r_i - |\tilde{q}M|_P|_{m_i}|_{m_i} \]

\[ \text{Precomputations:} \]

- for all \( i \in \{1, \ldots, k\} \), \( |M_i^{-1}|_{m_i} \) (\( k \) words)
- for all \( j \in \{1, \ldots, k\} \), \( \rightarrow |M_j|_P \) (\( k^2 \) words)
- for all \( \tilde{q} \in \{1, \ldots, k-1\} \), \( \rightarrow |\tilde{q}M|_P \) (\( k^2 \) words)

**Cost:** \( k \) mults + \( k^2 \) mults → quadratic complexity
RNS modular reduction

\[ A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P} \]

**function** `reduce-mod-P( \vec{A} )`:

\[
(\forall i) \quad z_i \leftarrow |a_i \cdot |M_i^{-1}|_{m_i}|_{m_i} \\
(\forall i) \quad \tilde{z}_i \leftarrow |z_i/2^{w-t}| \\
\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i/2^t + \varepsilon \rfloor \\
(\forall i) \quad r_i \leftarrow 0 \\
\text{for } j \leftarrow 1 \text{ to } k:\n\quad (\forall i) \quad r_i \leftarrow |r_i + z_j \cdot |M_j|_P|_{m_i}|_{m_i} \\
\quad (\forall i) \quad r_i \leftarrow |r_i - |\tilde{q}M|_P|_{m_i}|_{m_i} \\
\]

**Precomputations:**
RNS modular reduction

\[ A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P} \]

function reduce-mod-P(\(\vec{A}\)):

\[
\begin{align*}
(\forall i) & \quad z_i \leftarrow |a_i \cdot |M_i^{-1}|_{m_i}|_{m_i} \\
(\forall i) & \quad \tilde{z}_i \leftarrow |z_i/2^{w-t}| \\
\tilde{q} & \leftarrow |\sum_i \tilde{z}_i/2^t + \varepsilon| \\
(\forall i) & \quad r_i \leftarrow 0 \\
& \text{for } j \leftarrow 1 \text{ to } k: \\
(\forall i) & \quad r_i \leftarrow |r_i + z_j \cdot |M_j|_P|_{m_i}|_{m_i} | \text{ (for } j = 1, \ldots, k) \\
(\forall i) & \quad r_i \leftarrow |r_i - ||\tilde{q}M|_P|_{m_i}|_{m_i} | \text{ (for } j = 1, \ldots, k) \\
\end{align*}
\]

- Precomputations:
  - for all \( i \in \{1, \ldots, k\} \), \( |M_i^{-1}|_{m_i} \) (\( k \) words)
**RNS modular reduction**

\[
A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\bar{q}M|_P \pmod{P}
\]

**function** `reduce-mod-P(A)`:  

\[
(\forall i) \ z_i \leftarrow |a_i \cdot |M_i^{-1}|_{m_i}|_{m_i} \\
(\forall i) \ \tilde{z}_i \leftarrow |z_i/2^{w-t}| \\
\bar{q} \leftarrow |\sum_i \tilde{z}_i/2^t + \varepsilon| \\
(\forall i) \ r_i \leftarrow 0 \\
\text{for } j \leftarrow 1 \text{ to } k: \quad \\
\quad (\forall i) \ r_i \leftarrow |r_i + z_j \cdot |M_j|_P|_{m_i}|_{m_i} \\
(\forall i) \ r_i \leftarrow |r_i - |\bar{q}M|_P|_{m_i}|_{m_i}
\]

▶ Precomputations:

- for all \( i \in \{1, \ldots, k\}, \ |M_i^{-1}|_{m_i} \ (k \text{ words})
- for all \( j \in \{1, \ldots, k\}, \ |M_j|_P \ (k^2 \text{ words})
RNS modular reduction

\[ A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\bar{q}M|_P \quad (\text{mod } P) \]

**function** reduce-mod-\(P(\vec{A})\):

\[
\begin{align*}
(\forall i) \quad & z_i \gets |a_i \cdot M_i^{-1}|_{m_i} \\
(\forall i) \quad & \tilde{z}_i \gets |z_i/2^{w-t}| \\
& \tilde{q} \gets \left\lfloor \sum_i \tilde{z}_i/2^t + \varepsilon \right\rfloor \\
(\forall i) \quad & r_i \gets 0 \\
\text{for } j \gets 1 \text{ to } k: \\
(\forall i) \quad & r_i \gets |r_i + z_j \cdot ||M_j|_P|_{m_i}| \quad m_i \\
(\forall i) \quad & r_i \gets |r_i - ||\bar{q}M|_P|_{m_i}| \quad m_i 
\end{align*}
\]

**Precomputations:**

- for all \( i \in \{1, \ldots, k\} \), \(|M_i^{-1}|_{m_i}\) (\( k \) words)
- for all \( j \in \{1, \ldots, k\} \), \(|M_j|_P\) (\( k^2 \) words)
- for all \( \bar{q} \in \{1, \ldots, k - 1\} \), \(|\bar{q}M|_P\) (\( k^2 \) words)

\[\text{Cost: } k \text{ mults} + k^2 \text{ mults} \rightarrow \text{quadratic complexity}\]
RNS modular reduction

\[ A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_{i}^{-1} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P} \]

**function** reduce-mod-\(P(\vec{A})\):

\( (\forall i) \ z_i \leftarrow |a_i \cdot |M_{i}^{-1} \cdot |M_i|_P \rangle \)

\( (\forall i) \ \tilde{z}_i \leftarrow \lfloor z_i / 2^{w-t} \rfloor \)

\( \tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \rfloor \)

\( (\forall i) \ r_i \leftarrow 0 \)

**for** \( j \leftarrow 1 \) **to** \( k \):

\( (\forall i) \ r_i \leftarrow |r_i + z_j \cdot |M_j|_P \cdot |M_i|_P \rangle \)

\( (\forall i) \ r_i \leftarrow |r_i - |\tilde{q}M|_P \cdot |M_i|_P \rangle \)

**Precomputations:**

- for all \( i \in \{1, \ldots, k\} \), \( |M_{i}^{-1}|_m \) (**k** words)
- for all \( j \in \{1, \ldots, k\} \), \( |M_j|_P \) (**k** \(^2\) words)
- for all \( \tilde{q} \in \{1, \ldots, k-1\} \), \( |\tilde{q}M|_P \) (**k** \(^2\) words)

**Cost:**

\( k \) mults + \( k^2 \) mults \( \rightarrow \) quadratic complexity
RNS modular reduction

\[ A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P} \]

**function** reduce-mod-P(\( \vec{A} \)):

\[(\forall i)\ z_i \leftarrow |a_i \cdot M_i^{-1}|_{m_i} \]
\[(\forall i)\ \tilde{z}_i \leftarrow |z_i/2^{w-t}| \]
\[\tilde{q} \leftarrow \left\lfloor \sum_i \tilde{z}_i/2^t + \varepsilon \right\rfloor \]
\[(\forall i)\ r_i \leftarrow 0 \]

**for** \( j \leftarrow 1 \) **to** \( k \):

\[(\forall i)\ r_i \leftarrow |r_i + z_j \cdot ||M_j|_P|_{m_i}|_{m_i} | \]
\[(\forall i)\ r_i \leftarrow |r_i - ||\tilde{q}M|_P|_{m_i}|_{m_i} | \]

**Precomputations:**

- for all \( i \in \{1, \ldots, k\} \), \( |M_i^{-1}|_{m_i} \) (\( k \) words)
- for all \( j \in \{1, \ldots, k\} \), \( |M_j|_P \) (\( k^2 \) words)
- for all \( \tilde{q} \in \{1, \ldots, k-1\} \), \( |\tilde{q}M|_P \) (\( k^2 \) words)

**Cost:** \( k \) mults
**RNS modular reduction**

\[
A \mod P \equiv \left( \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P}
\]

**function** `reduce-mod-P(\vec{A})`:

\((\forall i)\) \(z_i \leftarrow a_i \cdot |M_i^{-1}|_{m_i}\)

\((\forall i)\) \(\tilde{z}_i \leftarrow \lfloor z_i/2^{w-t} \rfloor\)

\(\tilde{q} \leftarrow \lfloor \sum_i \tilde{z}_i/2^t + \varepsilon \rfloor\)

\((\forall i)\) \(r_i \leftarrow 0\)

for \(j \leftarrow 1\) to \(k\):

\((\forall i)\) \(r_i \leftarrow r_i + z_j \cdot |M_j|_P |m_i|_{m_i}\)

\((\forall i)\) \(r_i \leftarrow r_i - |\tilde{q}M|_P |m_i|_{m_i}\)

**Precomputations:**

- for all \(i \in \{1, \ldots, k\}\), \(|M_i^{-1}|_{m_i}\) (\(k\) words)
- for all \(j \in \{1, \ldots, k\}\), \(|M_j|_P\) (\(k^2\) words)
- for all \(\tilde{q} \in \{1, \ldots, k - 1\}\), \(|\tilde{q}M|_P\) (\(k^2\) words)

**Cost:** \(k\) mults + \(k^2\) mults
RNS modular reduction

\[ A \mod P \equiv \left( \sum_{i=1}^{k} \left| a_i \cdot M_i^{-1} \right| m_i \cdot |M_i|_P \right) - |\tilde{q}M|_P \pmod{P} \]

**function** reduce-mod-\(P(\vec{A})\):

\(\forall i\) \( z_i \leftarrow \left| a_i \cdot M_i^{-1} \right| m_i \)

\(\forall i\) \( \tilde{z}_i \leftarrow \left\lfloor z_i / 2^{w-t} \right\rfloor \)

\(\tilde{q} \leftarrow \left\lfloor \sum_i \tilde{z}_i / 2^t + \varepsilon \right\rfloor \)

\(\forall i\) \( r_i \leftarrow 0 \)

**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 \( j \in \{1, \ldots, k\} \), \( |M_j|_P \) (\( k^2 \) words)
- for all \( \tilde{q} \in \{1, \ldots, k - 1\} \), \( |\tilde{q}M|_P \) (\( k^2 \) words)

**Cost:** \( k \) mults + \( k^2 \) mults \( \rightarrow \) 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 $B_\alpha = (m_{\alpha,1}, \ldots, m_{\alpha,k})$ and $B_\beta = (m_{\beta,1}, \ldots, m_{\beta,k})$ 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 $\vec{X}_\alpha$ in base $B_\alpha$, BE($\vec{X}_\alpha, B_\alpha, B_\beta$) computes $\vec{X}_\beta$, the repr. of $X$ in base $B_\beta$
  - similarly, BE($\vec{X}_\beta, B_\beta, B_\alpha$) computes $\vec{X}_\alpha$ in base $B_\alpha$
RNS Montgomery reduction

- Requires two RNS bases $B_\alpha = (m_{\alpha,1}, \ldots, m_{\alpha,k})$ and $B_\beta = (m_{\beta,1}, \ldots, m_{\beta,k})$ 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 $B_\alpha$, BE($\overrightarrow{X_\alpha}, B_\alpha, B_\beta$) computes $\overrightarrow{X_\beta}$, the repr. of $X$ in base $B_\beta$
  - similarly, BE($\overrightarrow{X_\beta}, B_\beta, B_\alpha$) computes $\overrightarrow{X_\alpha}$ in base $B_\alpha$
  - similar to RNS modular reduction $\rightarrow O(k^2)$ complexity
RNS Montgomery reduction

\[
\begin{align*}
B_{\alpha} & \quad \overrightarrow{A}_{\alpha} \\
B_{\beta} & \quad \overrightarrow{A}_{\beta}
\end{align*}
\]
RNS Montgomery reduction

\[ \overline{A}_\alpha \rightarrow B_\alpha \]

\[ B_\alpha \]

\[ \overline{A}_\beta \rightarrow B_\beta \]

\[ B_\beta \]

\[ \overline{A}_\beta \]
### RNS Montgomery Reduction

#### $A_{\alpha}$

<table>
<thead>
<tr>
<th></th>
<th>$a_{\alpha,1}$</th>
<th>$a_{\alpha,2}$</th>
<th>$a_{\alpha,3}$</th>
<th>$a_{\alpha,4}$</th>
</tr>
</thead>
</table>

#### $B_{\alpha}$

(Not shown)

#### $B_{\beta}$

<table>
<thead>
<tr>
<th></th>
<th>$a_{\beta,1}$</th>
<th>$a_{\beta,2}$</th>
<th>$a_{\beta,3}$</th>
<th>$a_{\beta,4}$</th>
</tr>
</thead>
</table>

#### $A_{\beta}$

(Not shown)

---

The result is $(\rightarrow R_{\alpha}, \rightarrow R_{\beta}) \equiv (A \cdot M^{-1}_{\alpha}) \pmod{P}$

---

See recent results on this topic by Bigou and Tisserand.
RNS Montgomery reduction

\[
\begin{array}{c|cccc}
\overrightarrow{A}_\alpha & a_{\alpha,1} & a_{\alpha,2} & a_{\alpha,3} & a_{\alpha,4} \\
\times & \times & \times & \times \\
\overrightarrow{K}_\alpha & k_{\alpha,1} & k_{\alpha,2} & k_{\alpha,3} & k_{\alpha,4} \\
\end{array}
\]

\[
\begin{array}{c|cccc}
\overrightarrow{B}_\alpha & p'_{\alpha,1} & p'_{\alpha,2} & p'_{\alpha,3} & p'_{\alpha,4} \\
\overrightarrow{B}_\beta & a_{\beta,1} & a_{\beta,2} & a_{\beta,3} & a_{\beta,4} \\
\overrightarrow{A}_\beta & \end{array}
\]

Result is \((\overrightarrow{R}_\alpha, \overrightarrow{R}_\beta) \equiv (A \cdot M^{-1}_\alpha) \pmod{P}\)
RNS Montgomery reduction

\[ \overrightarrow{A}_\alpha \rightarrow B_\alpha \]
\[ \times \times \times \times \]
\[ \overrightarrow{(-P^{-1})_\alpha} \rightarrow \]
\[ \overrightarrow{K}_\alpha \rightarrow BE \rightarrow \]
\[ \overrightarrow{K}_\beta \rightarrow \]

\[ \begin{array}{cccc}
  a_{\alpha,1} & a_{\alpha,2} & a_{\alpha,3} & a_{\alpha,4} \\
  p'_{\alpha,1} & p'_{\alpha,2} & p'_{\alpha,3} & p'_{\alpha,4} \\
  k_{\alpha,1} & k_{\alpha,2} & k_{\alpha,3} & k_{\alpha,4} \\
\end{array} \]

\[ \begin{array}{cccc}
  a_{\beta,1} & a_{\beta,2} & a_{\beta,3} & a_{\beta,4} \\
  p'_{\beta,1} & p'_{\beta,2} & p'_{\beta,3} & p'_{\beta,4} \\
  k_{\beta,1} & k_{\beta,2} & k_{\beta,3} & k_{\beta,4} \\
\end{array} \]

Result is \((\overrightarrow{R}_\alpha, \overrightarrow{R}_\beta) \equiv (A \cdot M^{-1}_\alpha) \pmod{P}\)
### RNS Montgomery Reduction

<table>
<thead>
<tr>
<th>$A_{\alpha}$</th>
<th>$B_{\alpha}$</th>
<th>$A_{\beta}$</th>
<th>$B_{\beta}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{\alpha,1}$</td>
<td>$a_{\alpha,2}$</td>
<td>$a_{\alpha,3}$</td>
<td>$a_{\alpha,4}$</td>
</tr>
<tr>
<td>$p'_{\alpha,1}$</td>
<td>$p'_{\alpha,2}$</td>
<td>$p'_{\alpha,3}$</td>
<td>$p'_{\alpha,4}$</td>
</tr>
<tr>
<td>$k_{\alpha,1}$</td>
<td>$k_{\alpha,2}$</td>
<td>$k_{\alpha,3}$</td>
<td>$k_{\alpha,4}$</td>
</tr>
<tr>
<td>$p_{\alpha,1}$</td>
<td>$p_{\alpha,2}$</td>
<td>$p_{\alpha,3}$</td>
<td>$p_{\alpha,4}$</td>
</tr>
<tr>
<td>$p_{\beta,1}$</td>
<td>$p_{\beta,2}$</td>
<td>$p_{\beta,3}$</td>
<td>$p_{\beta,4}$</td>
</tr>
</tbody>
</table>

### Result

The result is $(\overrightarrow{R_{\alpha}}, \overrightarrow{R_{\beta}}) \equiv (A \cdot M^{-1}_{\alpha}) \pmod{P}$

---

See recent results on this topic by Bigou and Tisserand.
RNS Montgomery reduction

\[
\begin{array}{c}
\begin{array}{c}
A_\alpha \\
(\rightarrow)
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
B_\alpha \\
\times \times \times \times
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
P_\alpha \\
+ + + +
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
A_\alpha \\
\rightarrow
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
a_{\alpha,1} a_{\alpha,2} a_{\alpha,3} a_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p'_{\alpha,1} p'_{\alpha,2} p'_{\alpha,3} p'_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
k_{\alpha,1} k_{\alpha,2} k_{\alpha,3} k_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p_{\alpha,1} p_{\alpha,2} p_{\alpha,3} p_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
a_{\alpha,1} a_{\alpha,2} a_{\alpha,3} a_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p'_{\alpha,1} p'_{\alpha,2} p'_{\alpha,3} p'_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
k_{\alpha,1} k_{\alpha,2} k_{\alpha,3} k_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p_{\alpha,1} p_{\alpha,2} p_{\alpha,3} p_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
a_{\alpha,1} a_{\alpha,2} a_{\alpha,3} a_{\alpha,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p'_{\alpha,1} p'_{\alpha,2} p'_{\alpha,3} p'_{\alpha,4}
\end{array}
\end{array}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
B_\beta \\
\times \times \times \times
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
P_\beta \\
+ + + +
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
A_\beta \\
\rightarrow
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
a_{\beta,1} a_{\beta,2} a_{\beta,3} a_{\beta,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p'_{\beta,1} p'_{\beta,2} p'_{\beta,3} p'_{\beta,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
k_{\beta,1} k_{\beta,2} k_{\beta,3} k_{\beta,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p_{\beta,1} p_{\beta,2} p_{\beta,3} p_{\beta,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
a_{\beta,1} a_{\beta,2} a_{\beta,3} a_{\beta,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p'_{\beta,1} p'_{\beta,2} p'_{\beta,3} p'_{\beta,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
k_{\beta,1} k_{\beta,2} k_{\beta,3} k_{\beta,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
p_{\beta,1} p_{\beta,2} p_{\beta,3} p_{\beta,4}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
a_{\beta,1} a_{\beta,2} a_{\beta,3} a_{\beta,4}
\end{array}
\end{array}
\end{array}
\end{array}
\]

Result is \((\rightarrow R_\alpha, \rightarrow R_\beta) \equiv (A \cdot M^{-1}_\alpha) \pmod{P}\)

See recent results on this topic by Bigou and Tisserand.
RNS Montgomery reduction

\[ \overrightarrow{A_\alpha} \rightarrow \begin{bmatrix} a_{\alpha,1} & a_{\alpha,2} & a_{\alpha,3} & a_{\alpha,4} \end{bmatrix} \times \begin{bmatrix} p'_{\alpha,1} & p'_{\alpha,2} & p'_{\alpha,3} & p'_{\alpha,4} \end{bmatrix} \]

\[ \overrightarrow{K_\alpha} \rightarrow \begin{bmatrix} k_{\alpha,1} & k_{\alpha,2} & k_{\alpha,3} & k_{\alpha,4} \end{bmatrix} \times \begin{bmatrix} p_{\alpha,1} & p_{\alpha,2} & p_{\alpha,3} & p_{\alpha,4} \end{bmatrix} \]

\[ \overrightarrow{P_\alpha} \rightarrow \begin{bmatrix} a_{\alpha,1} & a_{\alpha,2} & a_{\alpha,3} & a_{\alpha,4} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \]

\[ T_\alpha \equiv 0 \pmod{M_\alpha} \]

\[ \overrightarrow{A_\beta} \rightarrow \begin{bmatrix} a_{\beta,1} & a_{\beta,2} & a_{\beta,3} & a_{\beta,4} \end{bmatrix} \times \begin{bmatrix} p_{\beta,1} & p_{\beta,2} & p_{\beta,3} & p_{\beta,4} \end{bmatrix} \]

\[ \overrightarrow{K_\beta} \rightarrow \begin{bmatrix} k_{\beta,1} & k_{\beta,2} & k_{\beta,3} & k_{\beta,4} \end{bmatrix} \times \begin{bmatrix} p_{\beta,1} & p_{\beta,2} & p_{\beta,3} & p_{\beta,4} \end{bmatrix} \]

\[ \overrightarrow{P_\beta} \rightarrow \begin{bmatrix} a_{\beta,1} & a_{\beta,2} & a_{\beta,3} & a_{\beta,4} \end{bmatrix} + \begin{bmatrix} t_{\beta,1} & t_{\beta,2} & t_{\beta,3} & t_{\beta,4} \end{bmatrix} \]

Result is \((\overrightarrow{R_\alpha}, \overrightarrow{R_\beta}) \equiv (A \cdot M - 1 \pmod{P}) \mod{M_\alpha}) \]

See recent results on this topic by Bigou and Tisserand.
RNS Montgomery reduction

\[ \overrightarrow{A_\alpha} \rightarrow a_{\alpha,1} a_{\alpha,2} a_{\alpha,3} a_{\alpha,4} \times \times \times \times \overset{(-P^{-1})_\alpha}{\rightarrow} p'_{\alpha,1} p'_{\alpha,2} p'_{\alpha,3} p'_{\alpha,4} \]

\[ \overrightarrow{K_\alpha} \rightarrow k_{\alpha,1} k_{\alpha,2} k_{\alpha,3} k_{\alpha,4} \rightarrow \overrightarrow{B_\alpha} \]

\[ \overset{\text{BE}}{\rightarrow} \]

\[ \overrightarrow{K_\beta} \rightarrow k_{\beta,1} k_{\beta,2} k_{\beta,3} k_{\beta,4} \times \times \times \times \rightarrow \overrightarrow{P_\beta} \]

\[ \rightarrow \overrightarrow{A_\beta} \rightarrow a_{\beta,1} a_{\beta,2} a_{\beta,3} a_{\beta,4} \]

\[ \rightarrow \overrightarrow{T_\beta} \rightarrow t_{\beta,1} t_{\beta,2} t_{\beta,3} t_{\beta,4} \]
RNS Montgomery reduction

\[\begin{array}{c|c|c|c}
\hline
& A_{\alpha} & B_{\alpha} & A_{\alpha} \\
\hline
\overrightarrow{A_{\alpha}} & a_{\alpha,1} & a_{\alpha,2} & a_{\alpha,3} \\
\hline
(-P^{-1})_{\alpha} & p'_{\alpha,1} & p'_{\alpha,2} & p'_{\alpha,3} \\
\overrightarrow{K_{\alpha}} & k_{\alpha,1} & k_{\alpha,2} & k_{\alpha,3} \\
\hline
\overrightarrow{K_{\beta}} & k_{\beta,1} & k_{\beta,2} & k_{\beta,3} \\
\hline
P_{\beta} & p_{\beta,1} & p_{\beta,2} & p_{\beta,3} \\
\overrightarrow{A_{\beta}} & a_{\beta,1} & a_{\beta,2} & a_{\beta,3} \\
\hline
T_{\beta} & t_{\beta,1} & t_{\beta,2} & t_{\beta,3} \\
\hline
(M^{-1}_{\alpha})_{\beta} & m'_{\beta,1} & m'_{\beta,2} & m'_{\beta,3} \\
\hline
\end{array}\]
RNS Montgomery reduction

\[ \overrightarrow{A}_\alpha \rightarrow B_\alpha \]

\[ \overrightarrow{K}_\alpha \rightarrow B_\beta \rightarrow \overrightarrow{A}_\beta \]

\[ (-P^{-1})_\alpha \rightarrow \overrightarrow{P}_\alpha \rightarrow \overrightarrow{R}_\alpha \]

\[ \overrightarrow{T}_\beta \rightarrow (M_{\alpha}^{-1})_\beta \rightarrow \overrightarrow{R}_\beta \]

\[ \overrightarrow{K}_\alpha \rightarrow \overrightarrow{K}_\beta \]

\[ \overrightarrow{P}_\beta \rightarrow \overrightarrow{A}_\beta \]

\[ \overrightarrow{R}_\beta \]

Result is \((\overrightarrow{R}_\alpha, \overrightarrow{R}_\beta) \equiv (A \cdot M_{\alpha}^{-1}) \mod P\)
RNS Montgomery reduction

\[ \overrightarrow{A}_{\alpha} \rightarrow \overrightarrow{B}_{\alpha} \]

\[ (-P^{-1})_{\alpha} \rightarrow \overrightarrow{P}_{\alpha} \rightarrow \overrightarrow{R}_{\alpha} \]

\[ \overrightarrow{K}_{\alpha} \rightarrow \overrightarrow{B}_{\beta} \rightarrow \overrightarrow{A}_{\beta} \rightarrow \overrightarrow{R}_{\beta} \]

\[ \overrightarrow{K}_{\beta} \rightarrow \overrightarrow{P}_{\beta} \rightarrow \overrightarrow{R}_{\beta} \]

\[ \overrightarrow{t}_{\beta} \rightarrow \overrightarrow{M}_{\alpha}^{-1}_{\beta} \rightarrow \overrightarrow{R}_{\beta} \]

Result is \((\overrightarrow{R}_{\alpha}, \overrightarrow{R}_{\beta}) \equiv (A \cdot M_{\alpha}^{-1}, P_{\alpha})\mod T_{\alpha}\)

See recent results on this topic by Bigou and Tisserand
RNS Montgomery reduction

Result is $(\overrightarrow{R}_\alpha, \overrightarrow{R}_\beta) \equiv (A \cdot M_{\alpha}^{-1}) \pmod{P}$
RNS Montgomery reduction

Result is \((\overrightarrow{R_\alpha}, \overrightarrow{R_\beta}) \equiv (A \cdot M^{-1}_\alpha) \pmod{P})\)

See recent results on this topic by Bigou and Tisserand
I. Scalar multiplication

II. Elliptic curve arithmetic

III. Finite field arithmetic

IV. Software considerations

V. Notions of hardware design
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 throughputs 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!
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 throughputs 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!
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 throughputs 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 throughputs 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 throughputs 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
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
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
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
x + y = s + 2co
end architecture;
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 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

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity ha is
5   port ( x : in std_logic;
6         y : in std_logic;
7         s : out std_logic;
8         co : out std_logic );
9 end entity;
10
11 architecture arch of ha is
12 begin
13
14
15 end architecture;

\[ x + y = s + 2co \]
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;

\[ x + y = s + 2co \]
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;
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;
```
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;
```

\[ x + y = s + 2co \]
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;
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

```
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

```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
  ha0 : ha port map ( x => x,
                      y => y,
                      s => s0,
                      co => co0 );
  ha1 : ha port map ( x => s0,
                      y => ci,
                      s => s,
                      co => co1 );
  co <= co0 or co1;
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);

  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
begin
  ha 0: ha port map ( x => x,
                      y => y,
                      s => s0,
                      co => co0 );
  ha 1: ha port map ( x => s0,
                      y => ci,
                      s => s,
                      co => co1 );
  co <= co0 or co1;
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
  ha0: ha port map ( x => x,
                    y => y,
                    s => s0,
                    co => co0);
  ha1: ha port map ( x => s0,
                    y => ci,
                    s => s,
                    co => co1);
  co <= co0 or co1;
end architecture;
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

```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

```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

```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

```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;

  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;
```

\[ x + y + ci = s + 2co \]
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;

\[ x + y + ci = s + 2co \]
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;
A full-adder in VHDL

1 library ieee;
2 use ieee.std_logic_1164.all;

3 entity fa is
4   port ( x : in std_logic;
5          y : in std_logic;
6          ci : in std_logic;
7          s : out std_logic;
8          co : out std_logic );
9 end entity;
10
11 architecture arch of fa is
12   component ha is
13     port ( x : in std_logic; y : in std_logic;
14        s : out std_logic; co : out std_logic );
15   end component;
16
17 signal s_0 : std_logic;
18 signal co_0 : std_logic;
19 signal co_1 : std_logic;
20 begin
21   ha_0 : ha port map ( x => x, y => y,
22                         s => s_0, co => co_0 );
23   ha_1 : ha port map ( x => s_0, y => ci,
24                         s => s, co => co_1 );
25   co <= co_0 or co_1;
26 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 $F(x)$:
    $$A = a_{m-1}x^{m-1} + \cdots + a_1x + a_0, \quad \text{with } a_i \in \mathbb{F}_2$$
  - 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 $F(x)$:
    \[
    A = a_{m-1}x^{m-1} + \cdots + a_1x + a_0, \quad \text{with } a_i \in \mathbb{F}_2
    \]
  - 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 $F(x)$:
    \[ A = a_{m-1}x^{m-1} + \cdots + a_1x + a_0, \quad \text{with } a_i \in \mathbb{F}_2 \]
  - 1 bit per coefficient

- Addition: coefficient-wise addition over $\mathbb{F}_p$

- Squaring: 2-nd power Frobenius

• Inversion: no need for a full blown extended Euclidean algorithm
  - use Fermat's little theorem:
    \[ A^{-1} = A^{2^m-2} = (A^{2^m-1} - 1)^2 \]
    - computing $A^{2^m-1} - 1$ only requires multiplications and Frobenius
      \cite{Itoh and Tsujii, 1988}
    - no extra hardware for inversion
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 $F(x)$:
    \[ A = a_{m-1}x^{m-1} + \cdots + a_1x + a_0, \quad \text{with } a_i \in \mathbb{F}_2 \]
  - 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 $F(x)$:
    \[ A = a_{m-1}x^{m-1} + \cdots + a_1x + a_0, \quad \text{with} \ a_i \in \mathbb{F}_2 \]
  - 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 $F(x)$:
    \[
    A = a_{m-1}x^{m-1} + \cdots + a_1x + a_0, \quad \text{with } a_i \in \mathbb{F}_2
    \]
  - 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 \cite{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)

\[
x^{m-1} \ldots x^2 \times 1
\]

\[
\begin{array}{cccccccccc}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\times & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\hline
A & B
\end{array}
\]
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)

```
x^{m-1} \quad \ldots \quad x^2 \times 1
A
\times
B

\begin{align*}
&\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
&\times \quad \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
&\downarrow \quad \quad \downarrow \quad \quad \downarrow \quad \quad \downarrow \\
&\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \quad b_{m-1} \cdot A \\
&\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \quad b_{m-2} \cdot A \\
&\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \quad b_{m-3} \cdot A
\end{align*}
```
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)

```
x^{m-1} \cdots x^2 x 1
```

```
A
```

```
B
```

```
b_{m-1} \cdot A \cdot x^2
```

```
b_{m-2} \cdot A \cdot x
```

```
b_{m-3} \cdot A
```

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)

![Diagram of multiplication over $\mathbb{F}_{2^m}$]
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)

<table>
<thead>
<tr>
<th>$x^{m-1}$</th>
<th>$\cdots$</th>
<th>$x^2$</th>
<th>$x$</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
&x^{m-1} \quad \cdots \quad x^2 \quad x \quad 1 \\
&\times \\
&\underline{\text{Result}} \\
&b_{m-1} \cdot A \cdot x^2 \pmod F \\
&b_{m-2} \cdot A \cdot x \pmod F \\
&b_{m-3} \cdot A
\end{align*}
\]
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)

\[
\begin{array}{cccccc}
& x^{m-1} & \cdots & x^2 & x & 1 \\
\times & B & & & & \\
\hline
& A & & & & \\
+ & & & & & \\
+ & & & & & \\
\hline
& R \text{ (partial sum)} & & & & \\
\end{array}
\]

- $\lceil m/d \rceil$ clock cycles for computing the product
- Area grows with $d$: area–time trade-off
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)

\[
\begin{array}{cccc}
& x^{m-1} & \ldots & x^2 \times 1 \\
A & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
B & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
+ & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
+ & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\hline
& \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
R & \text{(partial sum)} \\
\end{array}
\]

\[(b_{m-1} \cdot A \cdot x^2) \mod F \]
\[(b_{m-2} \cdot A \cdot x) \mod F \]
\[b_{m-3} \cdot A \]
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)

\[
\begin{array}{cccccccc}
  x^{m-1} & \cdots & x^2 & x & 1 & A \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
  x & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
  \hline
  (b_{m-1} \cdot A \cdot x^2) \mod F & (b_{m-2} \cdot A \cdot x) \mod F & b_{m-3} \cdot A & R \text{ (partial sum)} \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]
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)

\[ R \] (partial sum)

\[ R \mod F \]

\[ (b_{m-1} \cdot A \cdot x^2) \mod F \]

\[ (b_{m-2} \cdot A \cdot x) \mod F \]

\[ b_{m-3} \cdot A \]

\[ b_{m-4} \cdot A \]

\[ b_{m-5} \cdot A \]

\[ b_{m-6} \cdot A \]
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)

\[
\begin{array}{ccccccc}
  x^{m-1} & \cdots & x^2 & x & 1 \\
  A \\
  \times \\
  B \\
\hline \\
  \text{(b}_{m-1}\cdot A \cdot x^2) \mod F \\
  \text{(b}_{m-2}\cdot A \cdot x) \mod F \\
  b_{m-3} \cdot A \\
\hline \\
  R \cdot x^3 \\
  b_{m-4} \cdot A \cdot x^2 \\
  b_{m-5} \cdot A \cdot x \\
  b_{m-6} \cdot A
\end{array}
\]
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)

\[
x^{m-1} \quad \ldots \quad x^2 \quad x \quad 1
\]

\[
\begin{array}{cccccccc}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\times & & & & & & & \\
\hline
\end{array}
\]

\[
\begin{array}{cccc}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
+ & & & & & & & \\
+ & & & & & & & \\
\hline
\end{array}
\]

\[
\begin{array}{cccc}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
(b_{m-1} \cdot A \cdot x^2) \mod F & & & & & & & \\
(b_{m-2} \cdot A \cdot x) \mod F & & & & & & & \\
b_{m-3} \cdot A & & & & & & & \\
\hline
\end{array}
\]

\[
\begin{array}{cccc}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
(R \cdot x^3) \mod F & & & & & & & \\
(b_{m-4} \cdot A \cdot x^2) \mod F & & & & & & & \\
(b_{m-5} \cdot A \cdot x) \mod F & & & & & & & \\
b_{m-6} \cdot A & & & & & & & \\
\end{array}
\]
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)

---

<table>
<thead>
<tr>
<th>$x^{m-1}$</th>
<th>$\ldots$</th>
<th>$x^2$</th>
<th>$x$</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$+$

$(b_{m-1} \cdot A \cdot x^2) \mod F$

$(b_{m-2} \cdot A \cdot x) \mod F$

$b_{m-3} \cdot A$

$(R \cdot x^3) \mod F$

$(b_{m-4} \cdot A \cdot x^2) \mod F$

$(b_{m-5} \cdot A \cdot x) \mod F$

$b_{m-6} \cdot A$

$R$ (partial sum)
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)

\[
\begin{align*}
    x^{m-1} & \quad \ldots \quad x^2 \quad x \quad 1 \\
    \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet & \quad A \\
    \times & \quad \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
    \hline \\
    + & \quad \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
    + & \quad \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
    + & \quad \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
    + & \quad \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
    + & \quad \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
    \hline \\
    \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet & \quad R \text{ (partial sum)}
\end{align*}
\]

\[\left(b_{m-1} \cdot A \cdot x^2\right) \mod F\]
\[\left(b_{m-2} \cdot A \cdot x\right) \mod F\]
\[b_{m-3} \cdot A\]
\[\left(R \cdot x^3\right) \mod F\]
\[\left(b_{m-4} \cdot A \cdot x^2\right) \mod F\]
\[\left(b_{m-5} \cdot A \cdot x\right) \mod F\]
\[b_{m-6} \cdot A\]
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

$$
x^{m-1} \quad \ldots \quad x^2 \quad x \quad 1
\begin{array}{cccccccccccc}
\text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} \\
\times & & & & & & & & & & & \\
\hline
\text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} & \text{●} \end{array}
$$

\begin{align*}
\text{\textcolor{red}{A}} & \quad \text{\textcolor{red}{B}} \\
\text{(} b_{m-1} \cdot A \cdot x^2 \text{)} \mod F \\
\text{(} b_{m-2} \cdot A \cdot x \text{)} \mod F \\
\text{b}_{m-3} \cdot A \\
\text{\textcolor{red}{R (partial sum)}} & \\
\text{(} R \cdot x^3 \text{) \mod F} \\
\text{(} b_{m-4} \cdot A \cdot x^2 \text{) \mod F} \\
\text{(} b_{m-5} \cdot A \cdot x \text{) \mod F} \\
\text{b}_{m-6} \cdot A
\end{align*}
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 $\mathbb{F}_2$
- Coefficient-wise addition (XOR gates in $\mathbb{F}_2$)

$$\begin{align*}
A \mod \mathbb{F} \quad &\ll 1 \quad \ll 2 \quad \ll 3 \\
B \mod \mathbb{F} \quad &\ll 1 \quad \ll 2 \quad \ll 3 \\
\end{align*}$$
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 $\mathbb{F}_{2^m}$

Parallel–serial multiplier
d coeffs / cycle
$\lceil \frac{m}{d} \rceil$ cycles / product

Unified operator
Frobenius ($\cdot)^2$
addition

feedback loop
double Frobenius ($\cdot)^4$
Arithmetic coprocessor for ECC over $\mathbb{F}_{2^m}$

Register file

Parallel–serial multiplier

\[ \frac{d}{\lceil \frac{m}{d} \rceil} \text{ cycles / product} \]
Arithmetic coprocessor for ECC over $\mathbb{F}_{2^m}$

- **Register file**
- **Unified operator**
  - addition
  - Frobenius $(\cdot)^2$
- **Parallel–serial multiplier**
  - $d$ coeffs / cycle
  - $\lceil m/d \rceil$ cycles / product
Arithmetic coprocessor for ECC over $\mathbb{F}_{2^m}$

- **Register file**
- **Unified operator**
  - Addition
  - Frobenius $(\cdot)^2$
  - Double Frobenius $(\cdot)^4$
  - Feedback loop
- **Parallel–serial multiplier**
  - $d$ coeffs / cycle
  - $\lceil m/d \rceil$ cycles / product
Arithmetic coprocessor for ECC over $F_{2^m}$
Thank you for your attention

Questions?