RAIM 2017 — October 26, 2017

Factoring integers with ECM on the Kalray MPPA-256 processor

Jérémie Detrey

CARAMBA team, LORIA

INRIA Nancy – Grand Est

Jeremie.Detrey@loria.fr

Joint work with:

Masahiro Ishii (NAIST, Nara, Japan)

Pierrick Gaudry (CARAMBA team, LORIA)

Atsuo Inomata (NAIST, Nara, Japan)

Kazutoshi Fujikawa (NAIST, Nara, Japan)

Context: Integer factorization

- Central problem in public-key cryptography:
  - integer factorization is a (supposedly) difficult problem
  - e.g., basis for the security of the RSA public-key cryptosystem:
    - private key: large primes p and q
    - public key: $N = p \cdot q$

Context: Integer factorization

- Central problem in public-key cryptography:
  - integer factorization is a (supposedly) difficult problem
  - e.g., basis for the security of the RSA public-key cryptosystem:
    - private key: large primes p and q
    - public key: $N = p \cdot q$

- Efficient factorization is important for
  - establishing factorization records
  - recommending secure key lengths
  - breaking weak instances of RSA (short keys)

Context: Integer factorization

- Central problem in public-key cryptography:
  - integer factorization is a (supposedly) difficult problem
  - e.g., basis for the security of the RSA public-key cryptosystem:
    - private key: large primes p and q
    - public key: $N = p \cdot q$

- Efficient factorization is important for
  - establishing factorization records
  - recommending secure key lengths
  - breaking weak instances of RSA (short keys)
- Current (publicly known) record:
  - factorization of the RSA-768 challenge (768 bits, or 232 digits)
  - the computation took $\sim$ 2000 core-years [Kleinjung *et al.*, 2010]

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p)$

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p) = \tilde{O}(\exp(\log p))$

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p) = \tilde{O}\left(\exp\left(\log p\right)\right)$
  - ECM (Elliptic Curve Method) [Lenstra, 1987]:

$$\exp\left(\left(\sqrt{2} + o(1)\right)\sqrt{\log p \log\log p}\right)$$

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p) = \tilde{O}\left(\exp\left(\log p\right)\right)$
  - ECM (Elliptic Curve Method) [Lenstra, 1987]:

$$\exp\left(\left(\sqrt{2} + o(1)\right)\sqrt{\log p \log\log p}\right)$$

- Find all prime factors of an integer N:

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p) = \tilde{O}\left(\exp\left(\log p\right)\right)$
  - ECM (Elliptic Curve Method) [Lenstra, 1987]:

$$\exp\left(\left(\sqrt{2} + o(1)\right)\sqrt{\log p \log\log p}\right)$$

- Find all prime factors of an integer N:
  - (G)NFS (General Number Field Sieve) [Buhler *et al.*, 1993]:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right)(\log N)^{1/3}(\log\log N)^{2/3}\right)$$

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p) = \tilde{O}\left(\exp\left(\log p\right)\right)$
  - ECM (Elliptic Curve Method) [Lenstra, 1987]:

$$\exp\left(\left(\sqrt{2} + o(1)\right)\sqrt{\log p \log\log p}\right)$$

- Find all prime factors of an integer N:
  - (G)NFS (General Number Field Sieve) [Buhler *et al.*, 1993]:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right)(\log N)^{1/3}(\log\log N)^{2/3}\right)$$

- For factoring RSA moduli ($\sim$ 500 bits and above):

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p) = \tilde{O}\left(\exp\left(\log p\right)\right)$
  - ECM (Elliptic Curve Method) [Lenstra, 1987]:

$$\exp\left(\left(\sqrt{2} + o(1)\right)\sqrt{\log p \log\log p}\right)$$

- Find all prime factors of an integer N:
  - (G)NFS (General Number Field Sieve) [Buhler *et al.*, 1993]:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right)(\log N)^{1/3}(\log\log N)^{2/3}\right)$$

- For factoring RSA moduli ($\sim$ 500 bits and above):
  - use NFS (best asymptotic complexity)

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p) = \tilde{O}(\exp(\log p))$
  - ECM (Elliptic Curve Method) [Lenstra, 1987]:

$$\exp\left(\left(\sqrt{2} + o(1)\right)\sqrt{\log p \log \log p}\right)$$

- Find all prime factors of an integer N:
  - (G)NFS (General Number Field Sieve) [Buhler *et al.*, 1993]:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right)(\log N)^{1/3}(\log \log N)^{2/3}\right)$$

- For factoring RSA moduli ($\sim$ 500 bits and above):
  - use NFS (best asymptotic complexity)
  - requires factoring a huge quantity of smaller integers ($\sim$ 200 bits)

A few factorization algorithms

- Find small- to medium-size prime factors p of an integer N:
  - Trial division: $\tilde{O}(p) = \tilde{O}\left(\exp\left(\log p\right)\right)$
  - ECM (Elliptic Curve Method) [Lenstra, 1987]:

$$\exp\left(\left(\sqrt{2} + o(1)\right) \sqrt{\log p \log \log p}\right)$$

- Find all prime factors of an integer N:
  - (G)NFS (General Number Field Sieve) [Buhler *et al.*, 1993]:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right) (\log N)^{1/3} (\log \log N)^{2/3}\right)$$

- For factoring RSA moduli ($\sim$ 500 bits and above):
  - use NFS (best asymptotic complexity)
  - requires factoring a huge quantity of smaller integers ($\sim$ 200 bits)
    $\rightarrow$ use ECM for those

Outline of the talk

- ECM in a nutshell
- The Kalray MPPA-256 processor
- Multiprecision modular arithmetic
- Results and conclusion

Outline of the talk

- ECM in a nutshell
- The Kalray MPPA-256 processor
- Multiprecision modular arithmetic
- Results and conclusion

Elliptic curves

- Let K be a field

Elliptic curves

- Let K be a field
- An elliptic curve E over K is a projective plane curve given by an equation of the form

$$E : y^2 = x^3 + Ax + B,$$

with parameters $A, B \in K$ so that E is smooth

Elliptic curves

- Let K be a field
- An elliptic curve E over K is a projective plane curve given by an equation of the form

$$E : y^2 = x^3 + Ax + B,$$

with parameters A, $B \in K$ so that E is smooth

- Its set of points on K is given as

$$E(K) = \{(x, y) \in K \times K \mid (x, y) \text{ satisfies } E\}$$

Elliptic curves

- Let K be a field
- An elliptic curve E over K is a projective plane curve given by an equation of the form

$$E : y^2 = x^3 + Ax + B,$$

with parameters $A, B \in K$ so that E is smooth

- Its set of points on K is given as

$$E(K) = \{(x, y) \in K \times K \mid (x, y) \text{ satisfies } E\} \cup \{\mathcal{O}\},$$

where $\mathcal{O}$ is called the point at infinity

Elliptic curves

- Let K be a field
- An elliptic curve E over K is a projective plane curve given by an equation of the form

$$E : y^2 = x^3 + Ax + B,$$

  with parameters $A, B \in K$ so that E is smooth
- Its set of points on K is given as

$$E(K) = \{(x, y) \in K \times K \mid (x, y) \text{ satisfies } E\} \cup \{\mathcal{O}\},$$

  where $\mathcal{O}$ is called the point at infinity
- One can define an commutative addition law on $E(K)$:
  - $\mathcal{O}$ is the neutral element
  - $E(K)$ is therefore an abelian group

Elliptic curves

- Let K be a field
- An elliptic curve E over K is a projective plane curve given by an equation of the form

$$E : y^2 = x^3 + Ax + B,$$

with parameters $A, B \in K$ so that E is smooth

- Its set of points on K is given as

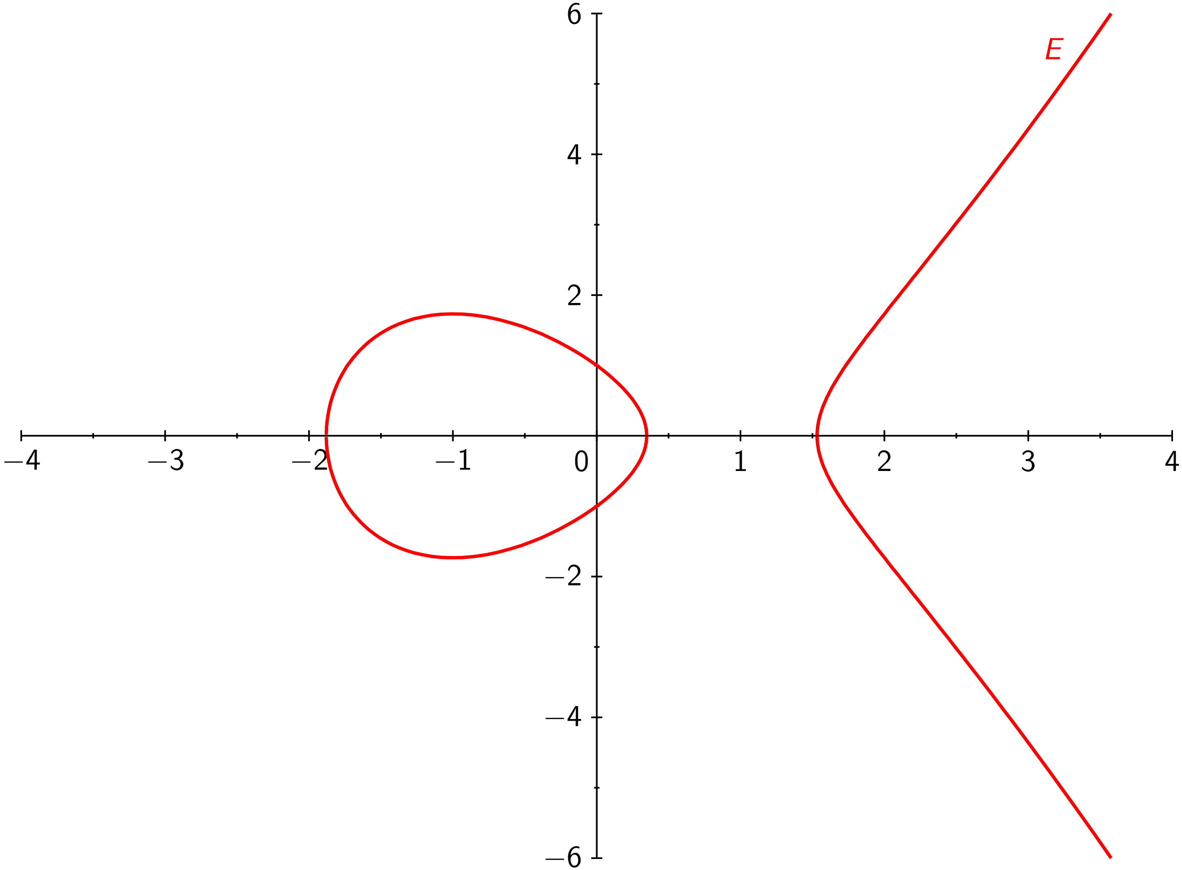
$$E(K) = \{(x, y) \in K \times K \mid (x, y) \text{ satisfies } E\} \cup \{\mathcal{O}\},$$

where $\mathcal{O}$ is called the point at infinity

- One can define an commutative addition law on $E(K)$:
  - $\mathcal{O}$ is the neutral element
  - $E(K)$ is therefore an abelian group
  - if K is finite, then so is $E(K)$

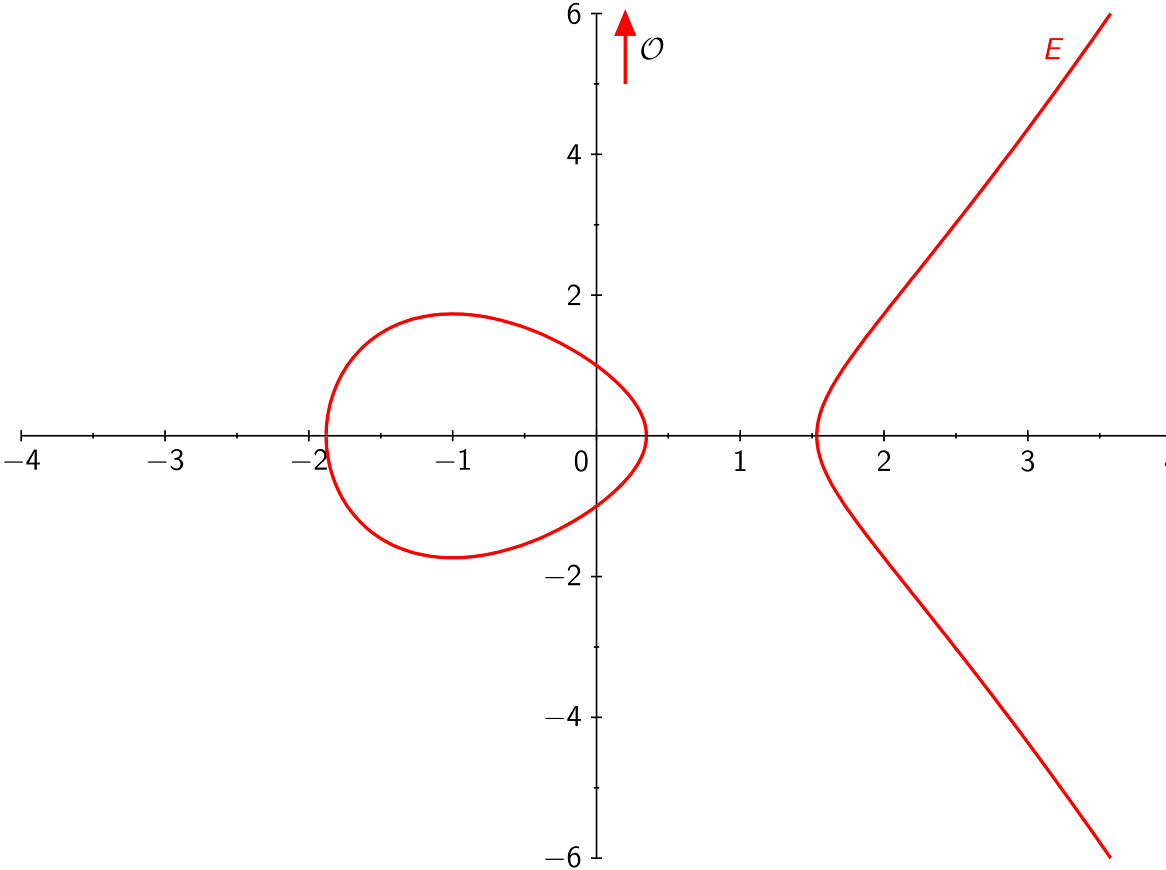
Example over the reals

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



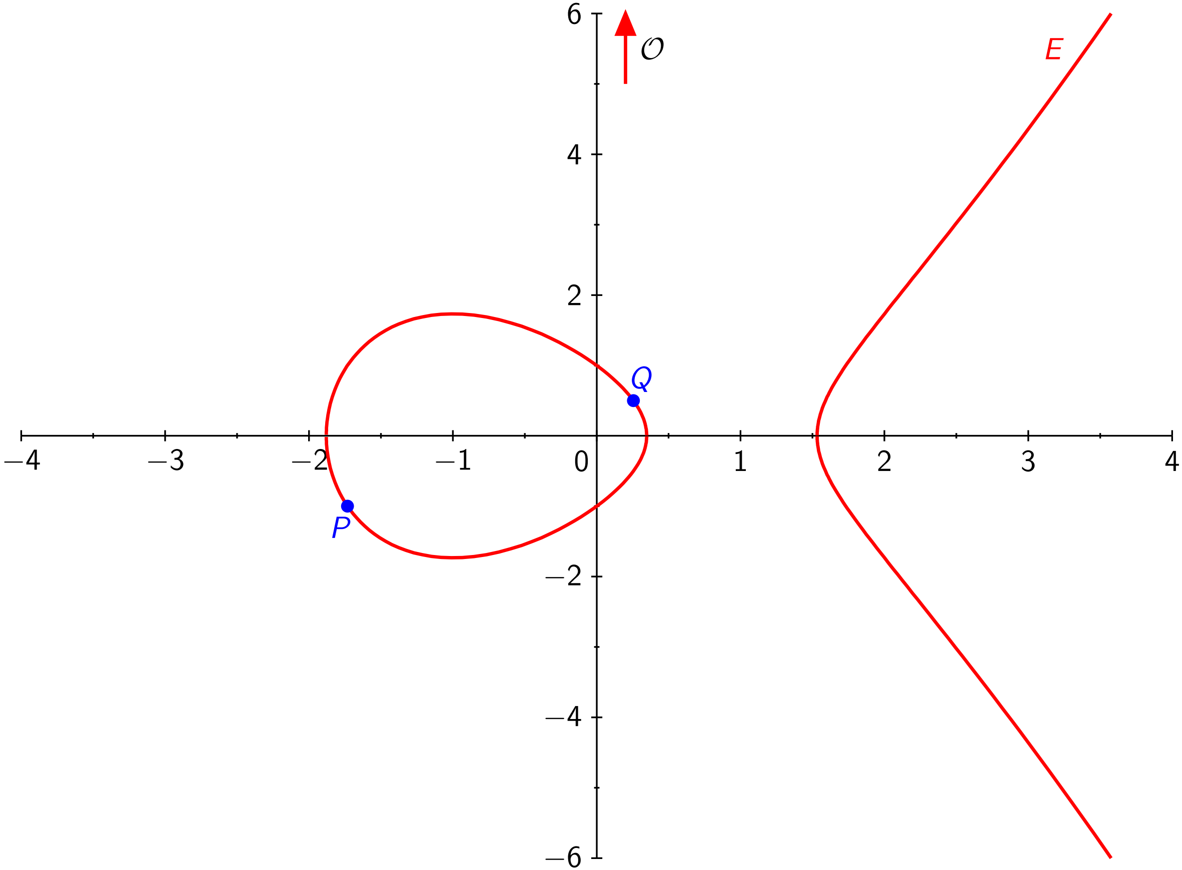
Example over the reals

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



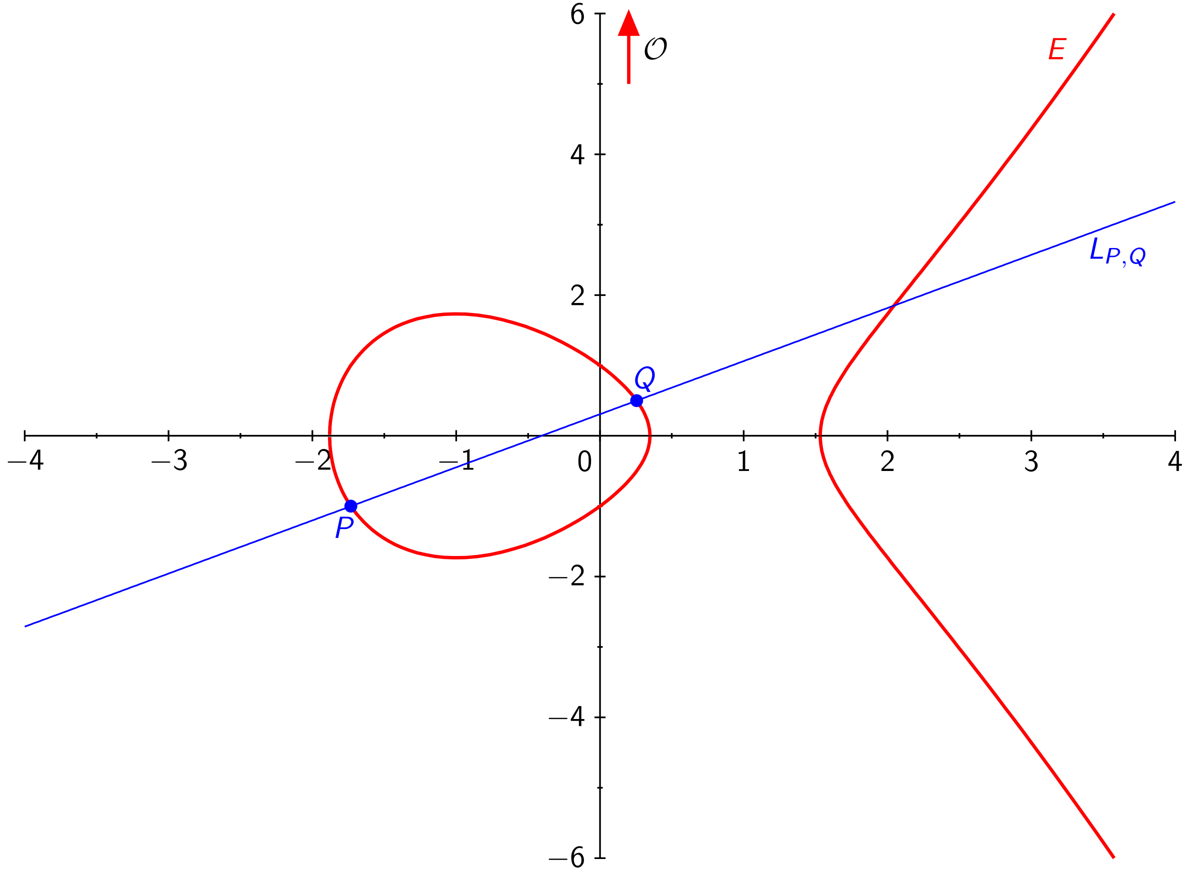
Example over the reals

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



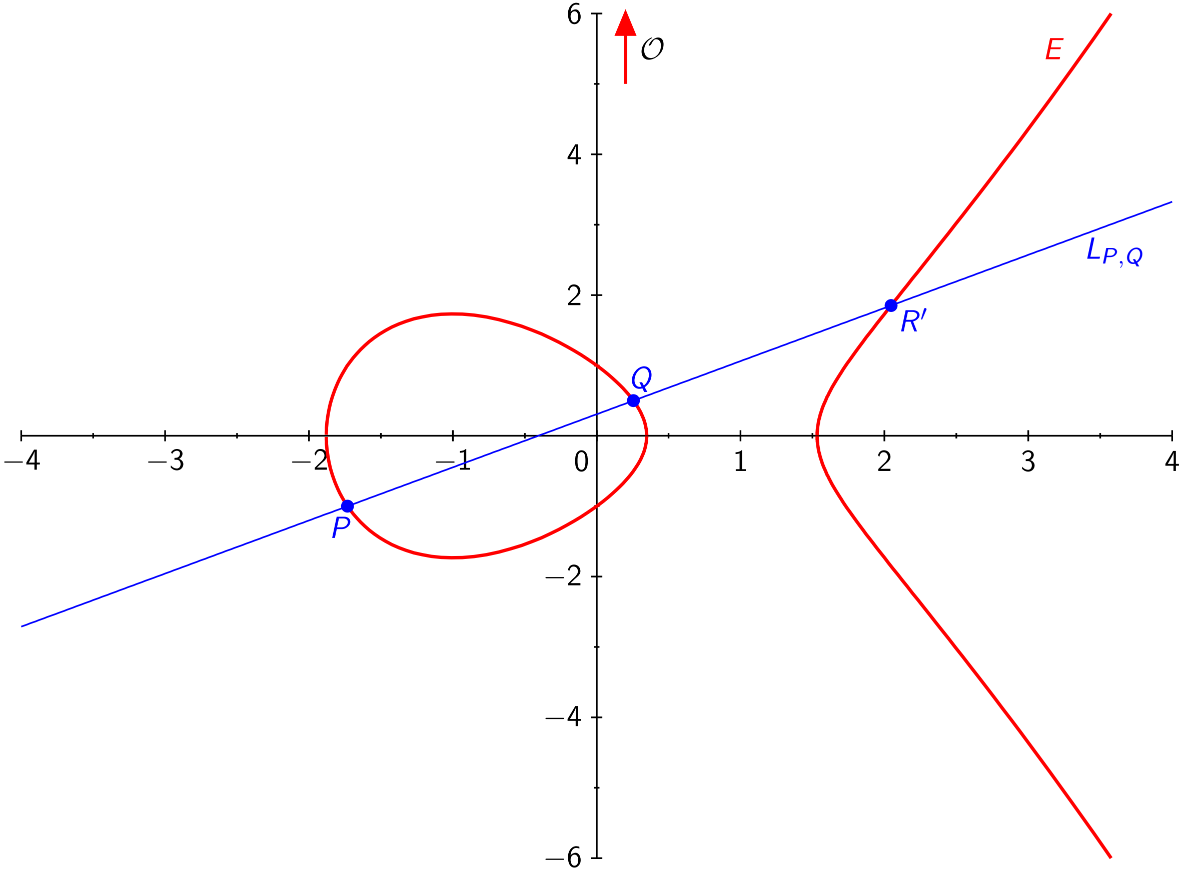
Example over the reals

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



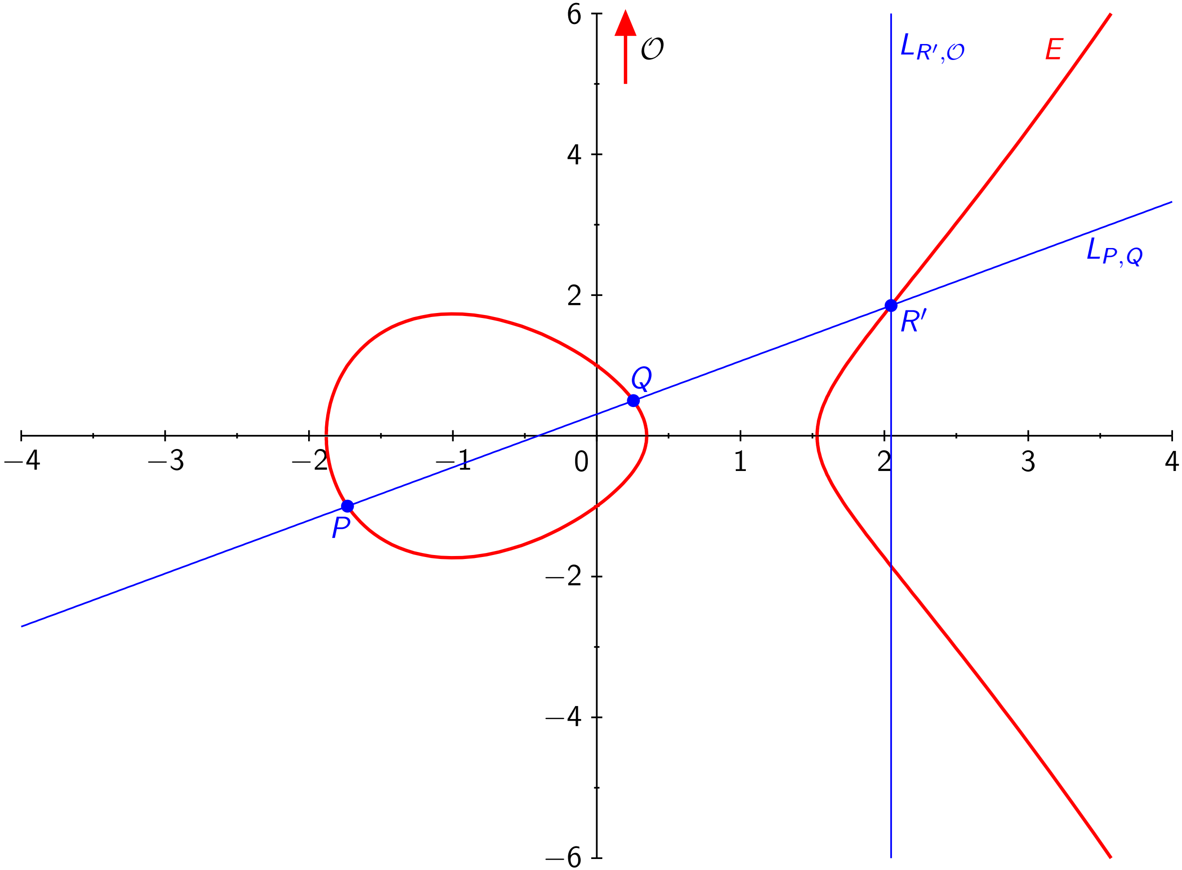
Example over the reals

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



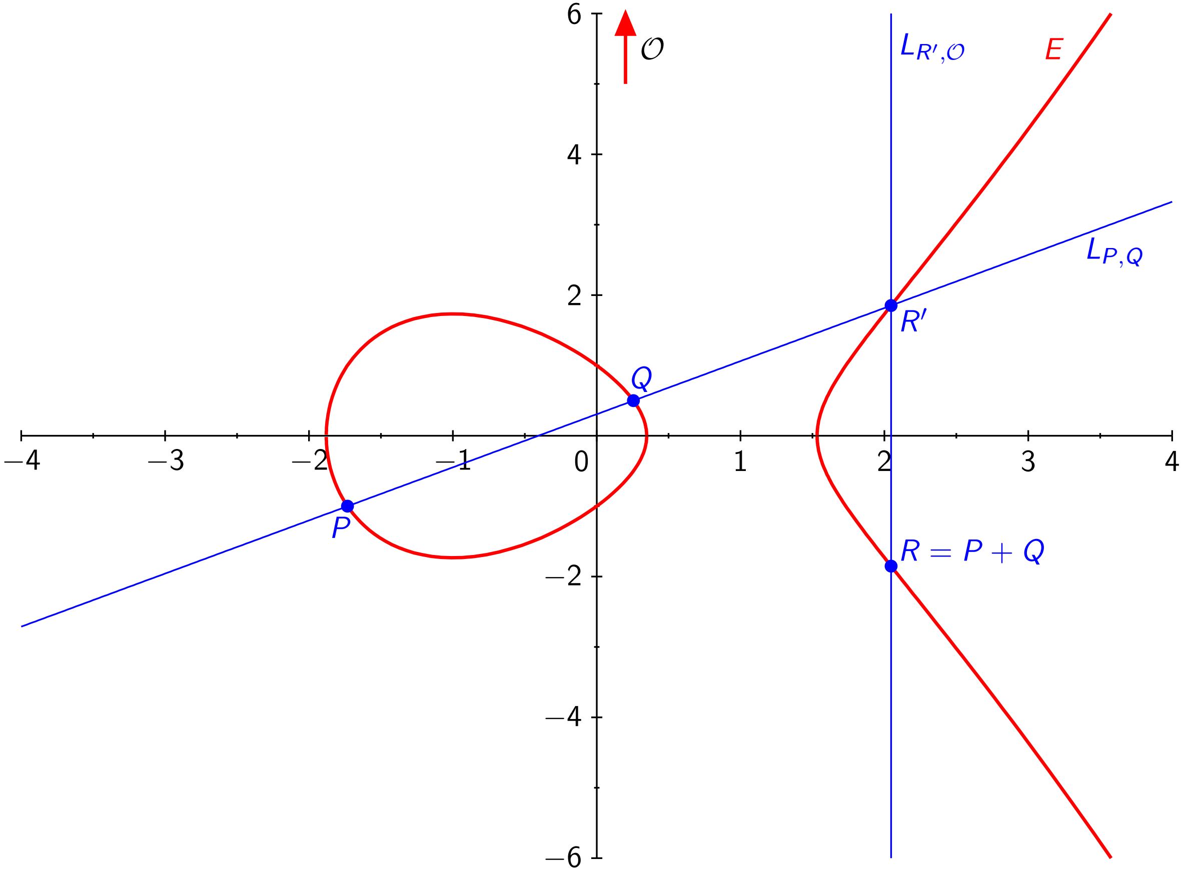
Example over the reals

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$

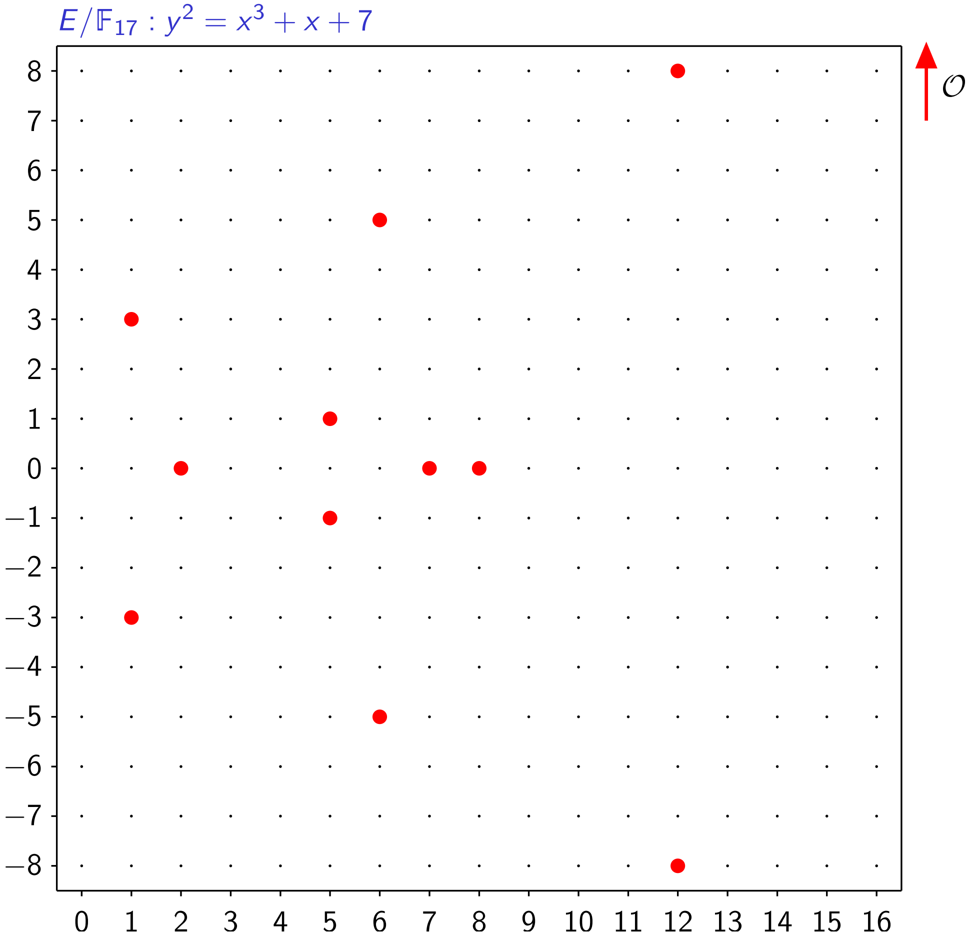


Example over the reals

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$

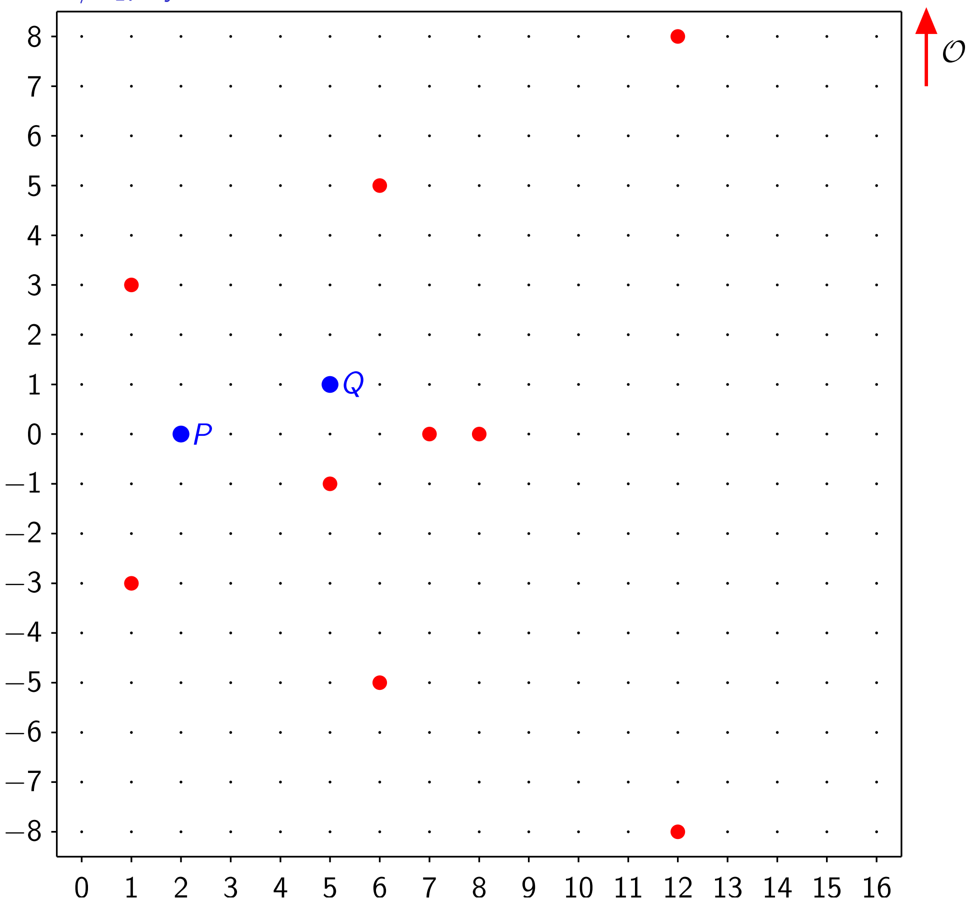


Example over a finite field



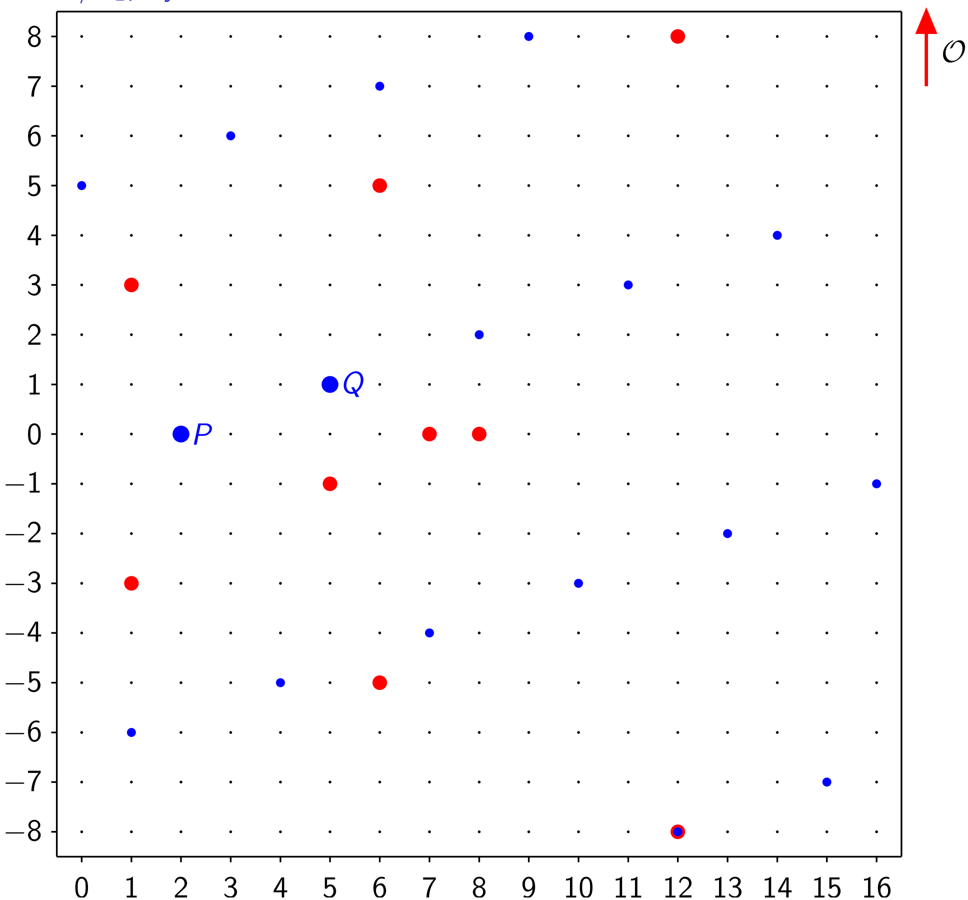
Example over a finite field

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



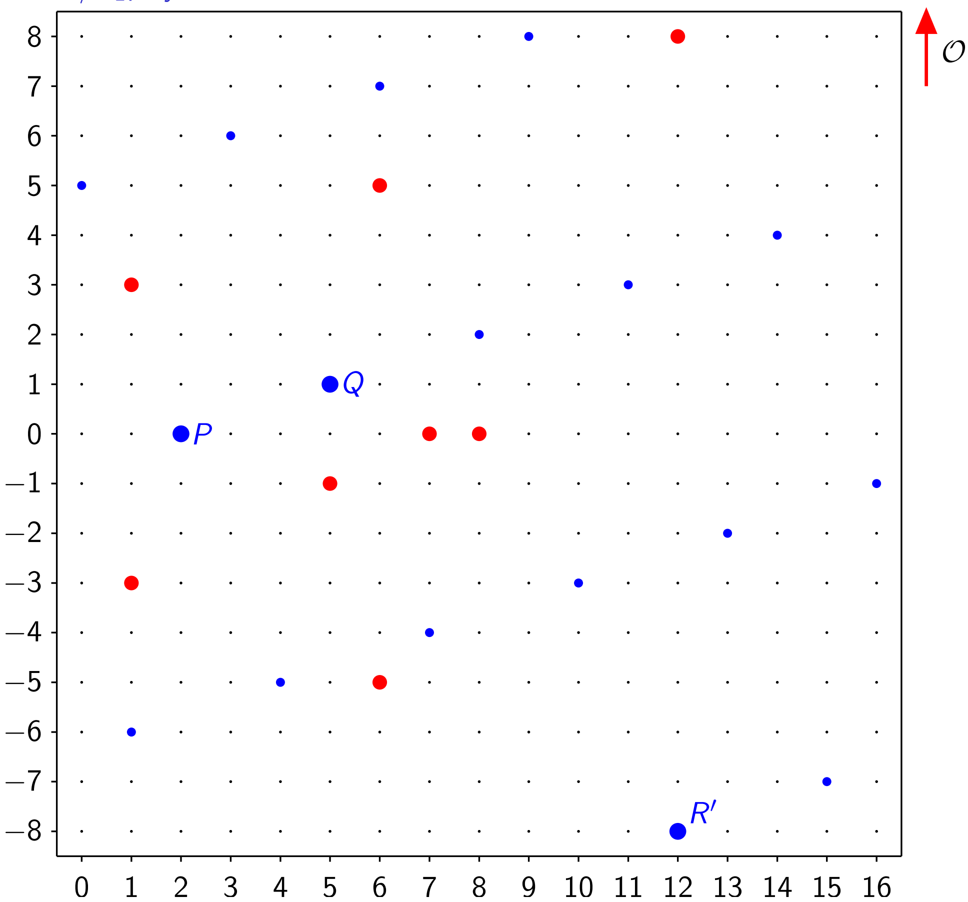
Example over a finite field

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



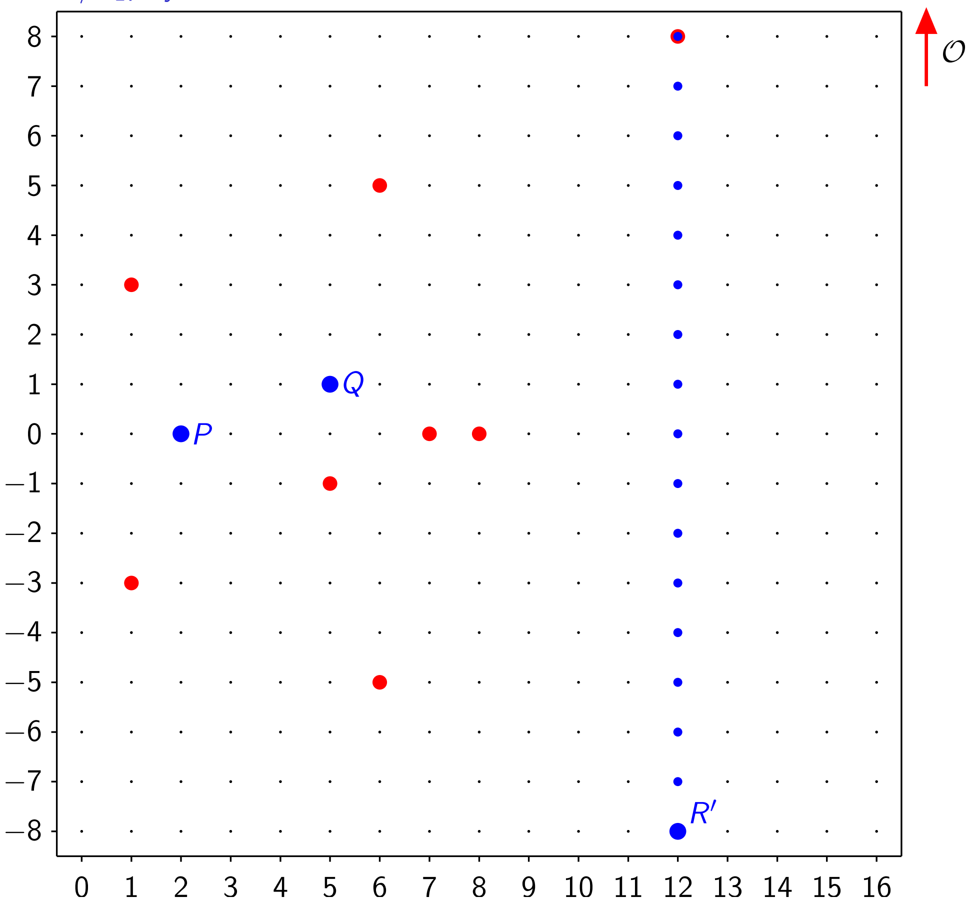
Example over a finite field

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

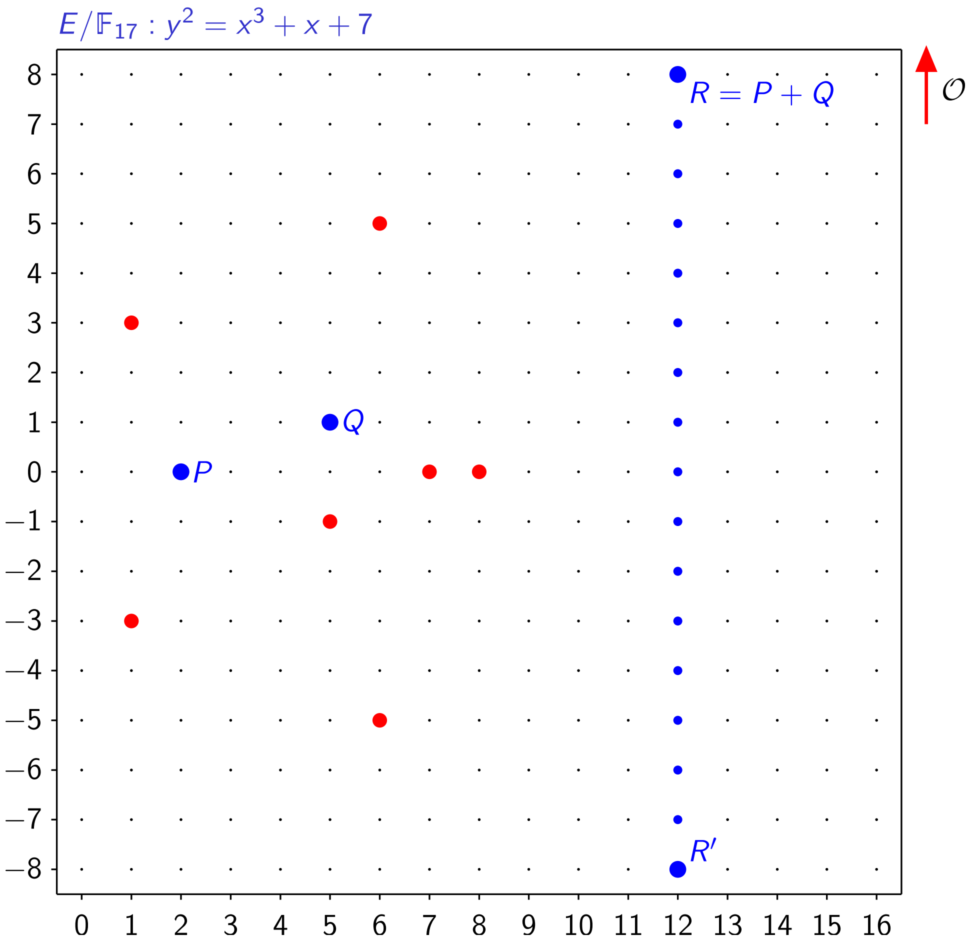


Example over a finite field

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



Example over a finite field



Back to integer factorization

$$E/K : y^2 = x^3 + Ax + B$$

Back to integer factorization

$$E/K : y^2 = x^3 + Ax + B$$

- What happens if we take $K = \mathbb{Z}/N\mathbb{Z}$, with N a composite integer?

Back to integer factorization

$$E/K : y^2 = x^3 + Ax + B$$

- What happens if we take $K = \mathbb{Z}/N\mathbb{Z}$, with N a composite integer?
  - K is not a field, and $E(\mathbb{Z}/N\mathbb{Z})$ is **not a group**!

Back to integer factorization

$$E/K : y^2 = x^3 + Ax + B$$

- What happens if we take $K = \mathbb{Z}/N\mathbb{Z}$, with N a composite integer?
  - K is not a field, and $E(\mathbb{Z}/N\mathbb{Z})$ is **not a group**!
  - however, by Chinese remaindering, $E(\mathbb{Z}/N\mathbb{Z})$ embeds a copy of $E(\mathbb{F}_{p_i})$ for each prime p_i dividing N:

$$E(\mathbb{Z}/N\mathbb{Z}) \cong E(\mathbb{F}_{p_1}) \times E(\mathbb{F}_{p_2}) \times \cdots \times E(\mathbb{F}_{p_r}) \times \text{some other stuff}$$

Back to integer factorization

$$E/K : y^2 = x^3 + Ax + B$$

- What happens if we take $K = \mathbb{Z}/N\mathbb{Z}$, with N a composite integer?
  - K is not a field, and $E(\mathbb{Z}/N\mathbb{Z})$ is **not a group**!
  - however, by Chinese remaindering, $E(\mathbb{Z}/N\mathbb{Z})$ embeds a copy of $E(\mathbb{F}_{p_i})$ for each prime p_i dividing N:

$$E(\mathbb{Z}/N\mathbb{Z}) \cong E(\mathbb{F}_{p_1}) \times E(\mathbb{F}_{p_2}) \times \cdots \times E(\mathbb{F}_{p_r}) \times \text{some other stuff}$$

- In fact, the addition law usually breaks when the result
  - is $\mathcal{O}$ modulo some (but not all) of the p_i's
  - is not $\mathcal{O}$ modulo the other ones

Back to integer factorization

$$E/K : y^2 = x^3 + Ax + B$$

- What happens if we take $K = \mathbb{Z}/N\mathbb{Z}$, with N a composite integer?
  - K is not a field, and $E(\mathbb{Z}/N\mathbb{Z})$ is **not a group**!
  - however, by Chinese remaindering, $E(\mathbb{Z}/N\mathbb{Z})$ embeds a copy of $E(\mathbb{F}_{p_i})$ for each prime p_i dividing N:

$$E(\mathbb{Z}/N\mathbb{Z}) \cong E(\mathbb{F}_{p_1}) \times E(\mathbb{F}_{p_2}) \times \cdots \times E(\mathbb{F}_{p_r}) \times \text{some other stuff}$$

- In fact, the addition law usually breaks when the result
  - is $\mathcal{O}$ modulo some (but not all) of the p_i's
  - is not $\mathcal{O}$ modulo the other ones
- In that case, a non-zero, non-invertible element $\xi \in \mathbb{Z}/N\mathbb{Z}$ pops up

Back to integer factorization

$$E/K : y^2 = x^3 + Ax + B$$

- What happens if we take $K = \mathbb{Z}/N\mathbb{Z}$, with N a composite integer?
  - K is not a field, and $E(\mathbb{Z}/N\mathbb{Z})$ is **not a group**!
  - however, by Chinese remaindering, $E(\mathbb{Z}/N\mathbb{Z})$ embeds a copy of $E(\mathbb{F}_{p_i})$ for each prime p_i dividing N:

$$E(\mathbb{Z}/N\mathbb{Z}) \cong E(\mathbb{F}_{p_1}) \times E(\mathbb{F}_{p_2}) \times \cdots \times E(\mathbb{F}_{p_r}) \times \text{some other stuff}$$

- In fact, the addition law usually breaks when the result
  - is $\mathcal{O}$ modulo some (but not all) of the p_i's
  - is not $\mathcal{O}$ modulo the other ones
- In that case, a non-zero, non-invertible element $\xi \in \mathbb{Z}/N\mathbb{Z}$ pops up
  $\rightarrow$ Compute $\gcd(\xi, N)$ and collect a non-trivial factor!

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$
  - pick a random point $P \in E(\mathbb{Z}/N\mathbb{Z}) \setminus \{\mathcal{O}\}$

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$
  - pick a random point $P \in E(\mathbb{Z}/N\mathbb{Z}) \setminus \{\mathcal{O}\}$
  - compute $Q \leftarrow kP$, with $k = \prod_{\pi^e \leq B_1} \pi^e$ (prime powers less than B_1)

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$
  - pick a random point $P \in E(\mathbb{Z}/N\mathbb{Z}) \setminus \{\mathcal{O}\}$
  - compute $Q \leftarrow kP$, with $k = \prod_{\pi^e \leq B_1} \pi^e$ (prime powers less than B_1)
    if the group law fails $\rightarrow$ get factor!

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$
  - pick a random point $P \in E(\mathbb{Z}/N\mathbb{Z}) \setminus \{\mathcal{O}\}$
  - compute $Q \leftarrow kP$, with $k = \prod_{\pi^e \leq B_1} \pi^e$ (prime powers less than B_1)

    if the group law fails $\rightarrow$ get factor!
  - compute πQ for every prime π with $B_1 < \pi \leq B_2$

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$
  - pick a random point $P \in E(\mathbb{Z}/N\mathbb{Z}) \setminus \{\mathcal{O}\}$
  - compute $Q \leftarrow kP$, with $k = \prod_{\pi^e \leq B_1} \pi^e$ (prime powers less than B_1)

    if the group law fails $\rightarrow$ get factor!
  - compute πQ for every prime π with $B_1 < \pi \leq B_2$

    if the group law fails $\rightarrow$ get factor!

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$
  - pick a random point $P \in E(\mathbb{Z}/N\mathbb{Z}) \setminus \{\mathcal{O}\}$
  - compute $Q \leftarrow kP$, with $k = \prod_{\pi^e \leq B_1} \pi^e$ (prime powers less than B_1)

    if the group law fails $\rightarrow$ get factor!
  - compute πQ for every prime π with $B_1 < \pi \leq B_2$

    if the group law fails $\rightarrow$ get factor!
  - divide N by all factors found, rinse and repeat

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$
  - pick a random point $P \in E(\mathbb{Z}/N\mathbb{Z}) \setminus \{\mathcal{O}\}$
  - compute $Q \leftarrow kP$, with $k = \prod_{\pi^e \leq B_1} \pi^e$ (prime powers less than B_1)

    if the group law fails $\rightarrow$ get factor!
  - compute πQ for every prime π with $B_1 < \pi \leq B_2$

    if the group law fails $\rightarrow$ get factor!
  - divide N by all factors found, rinse and repeat
- It is a probabilistic algorithm
  - the larger B_1 and B_2, the higher the probability of success

The ECM algorithm

- Parameters: bounds B_1 and B_2 with $0 < B_1 < B_2$
- For a given composite integer N:
  - pick a random elliptic curve E over $\mathbb{Z}/N\mathbb{Z}$
  - pick a random point $P \in E(\mathbb{Z}/N\mathbb{Z}) \setminus \{\mathcal{O}\}$
  - compute $Q \leftarrow kP$, with $k = \prod_{\pi^e \leq B_1} \pi^e$ (prime powers less than B_1)

    if the group law fails $\rightarrow$ get factor!
  - compute πQ for every prime π with $B_1 < \pi \leq B_2$

    if the group law fails $\rightarrow$ get factor!
  - divide N by all factors found, rinse and repeat
- It is a probabilistic algorithm
  - the larger B_1 and B_2, the higher the probability of success
  - ... but also the more expensive the computation

The ECM algorithm

- Easily parallelizable:
  - try many different curves on a single N
  - try to factor several N's in parallel

The ECM algorithm

- Easily parallelizable:
  - try many different curves on a single N
  - try to factor several N's in parallel
- State-of-the-art implementations:
  - software: EECM-MPFQ [Bernstein *et al.*, 2010]
  - on GPUs: [Bos & Kleinjung, 2012] and [Miele *et al.*, 2014]

The ECM algorithm

- Easily parallelizable:
  - try many different curves on a single N
  - try to factor several N's in parallel
- State-of-the-art implementations:
  - software: EECM-MPFQ [Bernstein *et al.*, 2010]
  - on GPUs: [Bos & Kleinjung, 2012] and [Miele *et al.*, 2014]
- Manycore processors are potentially good target architectures for ECM

Outline of the talk

- ECM in a nutshell
- The Kalray MPPA-256 processor
- Multiprecision modular arithmetic
- Results and conclusion

Kalray MPPA-256?

Kalray MPPA-256?

- Kalray: French start-up (CEA spin-off), launched in 2008

Kalray MPPA-256?

- Kalray: French start-up (CEA spin-off), launched in 2008
- MPPA (Multi-Purpose Processor Architecture): (co)processor

Kalray MPPA-256?

- Kalray: French start-up (CEA spin-off), launched in 2008
- MPPA (Multi-Purpose Processor Architecture): (co)processor
- 256: 256 compute cores

Kalray MPPA-256?

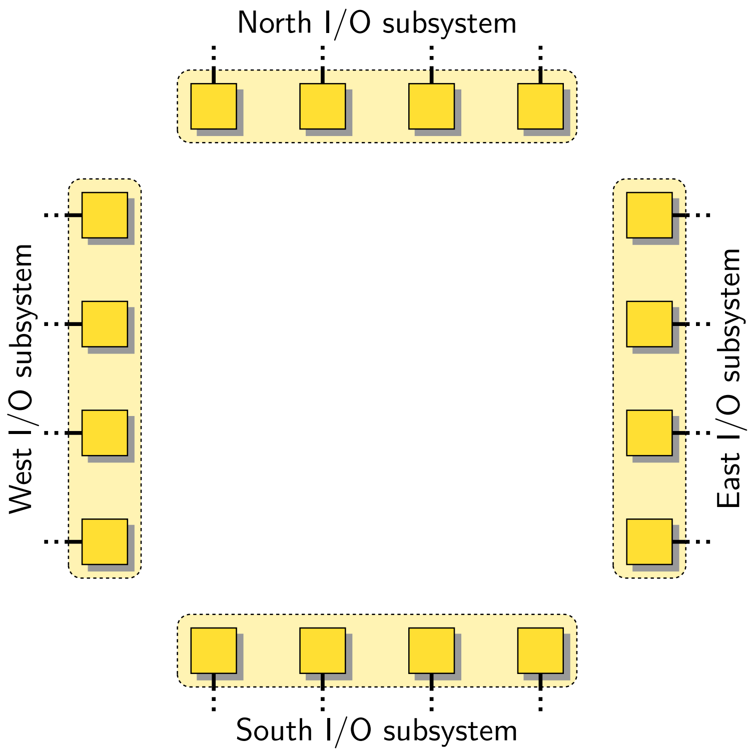
- Kalray: French start-up (CEA spin-off), launched in 2008
- MPPA (Multi-Purpose Processor Architecture): (co)processor
- 256: 256 compute cores
- For more info, visit `www.kalray.eu` (or ask Nicolas Brunie!)

Architecture

- Global view:

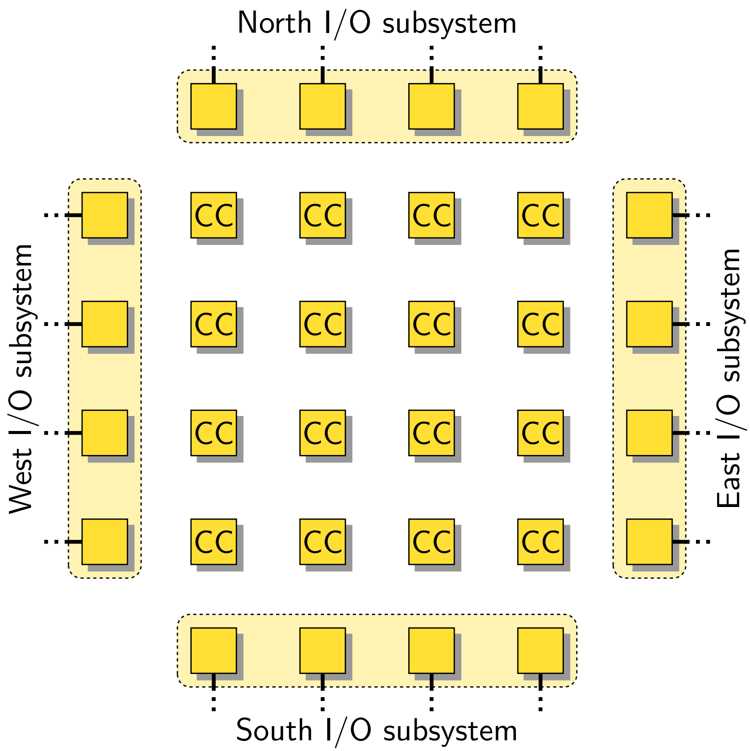
Architecture

- Global view:
  - 4 quad-core processors for I/Os (DDR RAM, PCI-e, etc.)



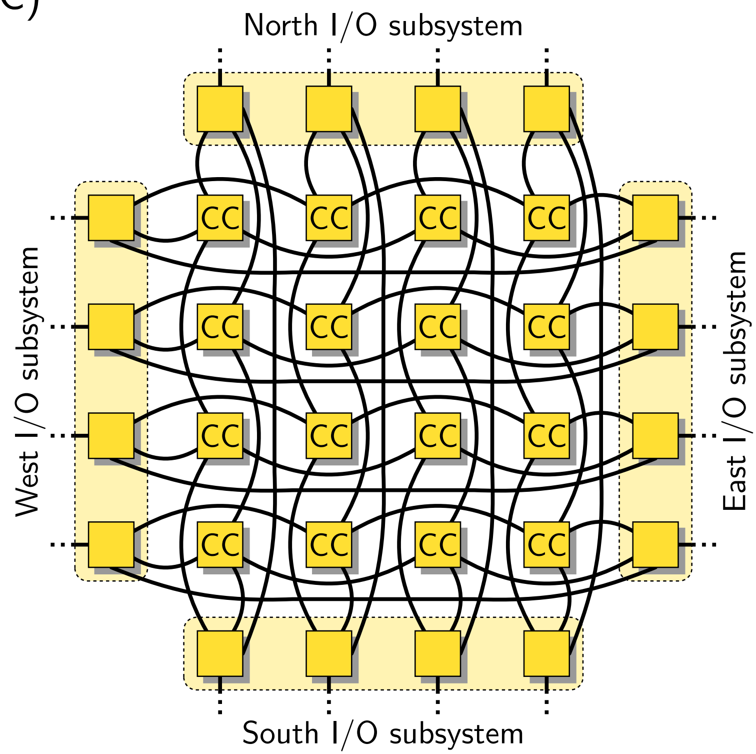
Architecture

- Global view:
  - 4 quad-core processors for I/Os (DDR RAM, PCI-e, etc.)
  - 4 × 4 compute clusters (CC)



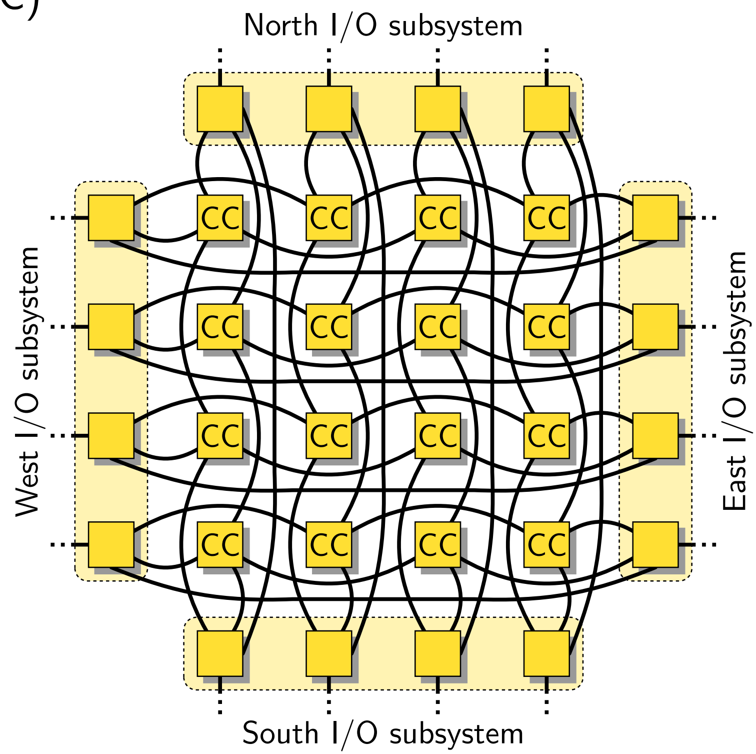
Architecture

- Global view:
  - 4 quad-core processors for I/Os (DDR RAM, PCI-e, etc.)
  - 4×4 compute clusters (CC)
  - toric network-on-chip (NoC)



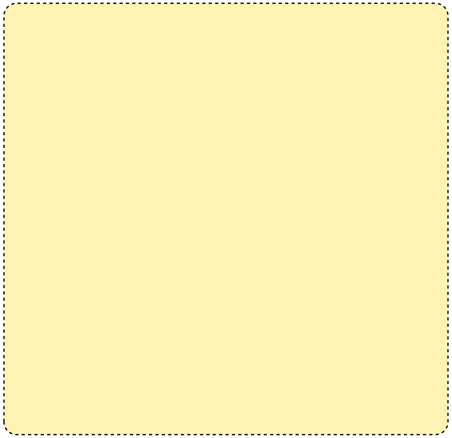
Architecture

- Global view:
  - 4 quad-core processors for I/Os (DDR RAM, PCI-e, etc.)
  - 4×4 compute clusters (CC)
  - toric network-on-chip (NoC)
  - frequency: 400 MHz
  - low power: $\leq$12-16 W



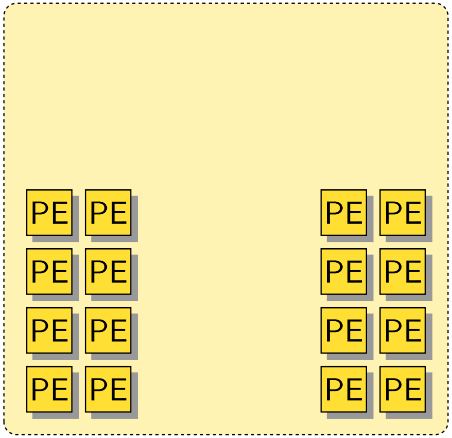
Compute clusters

- In each compute cluster:



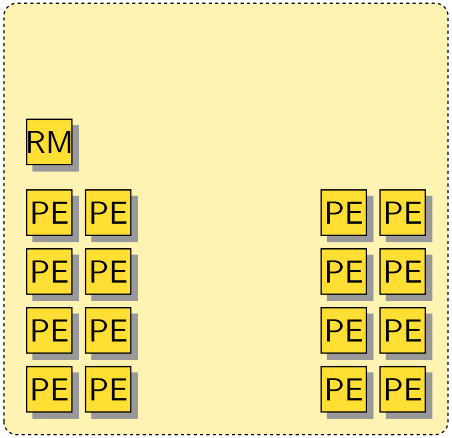
Compute clusters

- In each compute cluster:
  - 16 compute cores (PE)



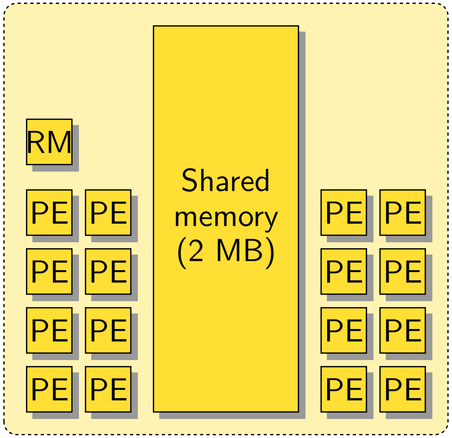
Compute clusters

- In each compute cluster:
  - 16 compute cores (PE)
  - $+$ 1 system core (RM)



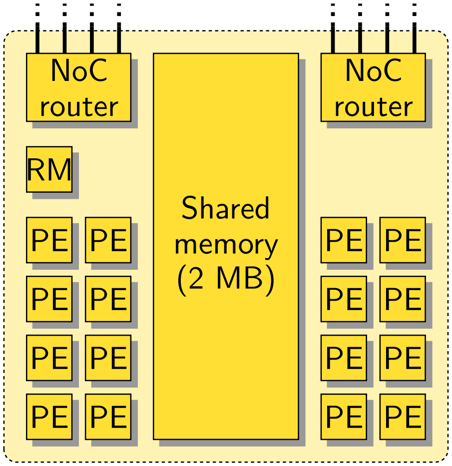
Compute clusters

- In each compute cluster:
  - 16 compute cores (PE)
  - + 1 system core (RM)
  - 2 MB shared memory



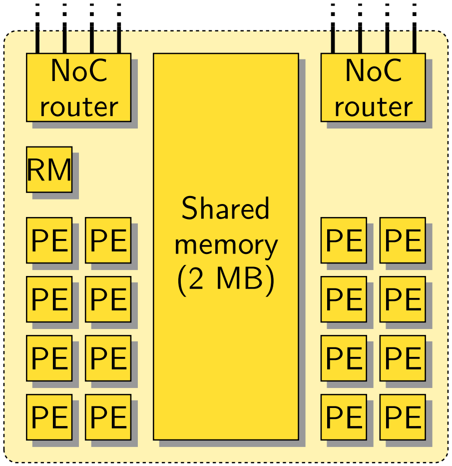
Compute clusters

- In each compute cluster:
  - 16 compute cores (PE)
  - + 1 system core (RM)
  - 2 MB shared memory
  - network interfaces



Compute clusters

- In each compute cluster:
  - 16 compute cores (PE)
  - + 1 system core (RM)
  - 2 MB shared memory
  - network interfaces
  - NodeOS (POSIX-like) + pthreads



Compute core microarchitecture

- Kalray-1 (K1) microarchitecture

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:
  - ALU_0 (32 bits)
  - ALU_1 (32 bits)

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:
  - ALU_0 (32 bits) } 64-bit ALU
  - ALU_1 (32 bits) }

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:
  - ALU_0 (32 bits) } 64-bit ALU
  - ALU_1 (32 bits) } 64-bit ALU
  - MAU (multiply & add)

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:
  - ALU_0 (32 bits) } 64-bit ALU
  - ALU_1 (32 bits) } 64-bit ALU
  - MAU (multiply & add) / FPU (floating point)

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:
  - ALU_0 (32 bits) } 64-bit ALU
  - ALU_1 (32 bits) }
  - MAU (multiply & add) / FPU (floating point) / ALU_{tiny}

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:
  - ALU_0 (32 bits) } 64-bit ALU
  - ALU_1 (32 bits) } 64-bit ALU
  - MAU (multiply & add) / FPU (floating point) / ALU_{tiny}
  - LSU (load & store)

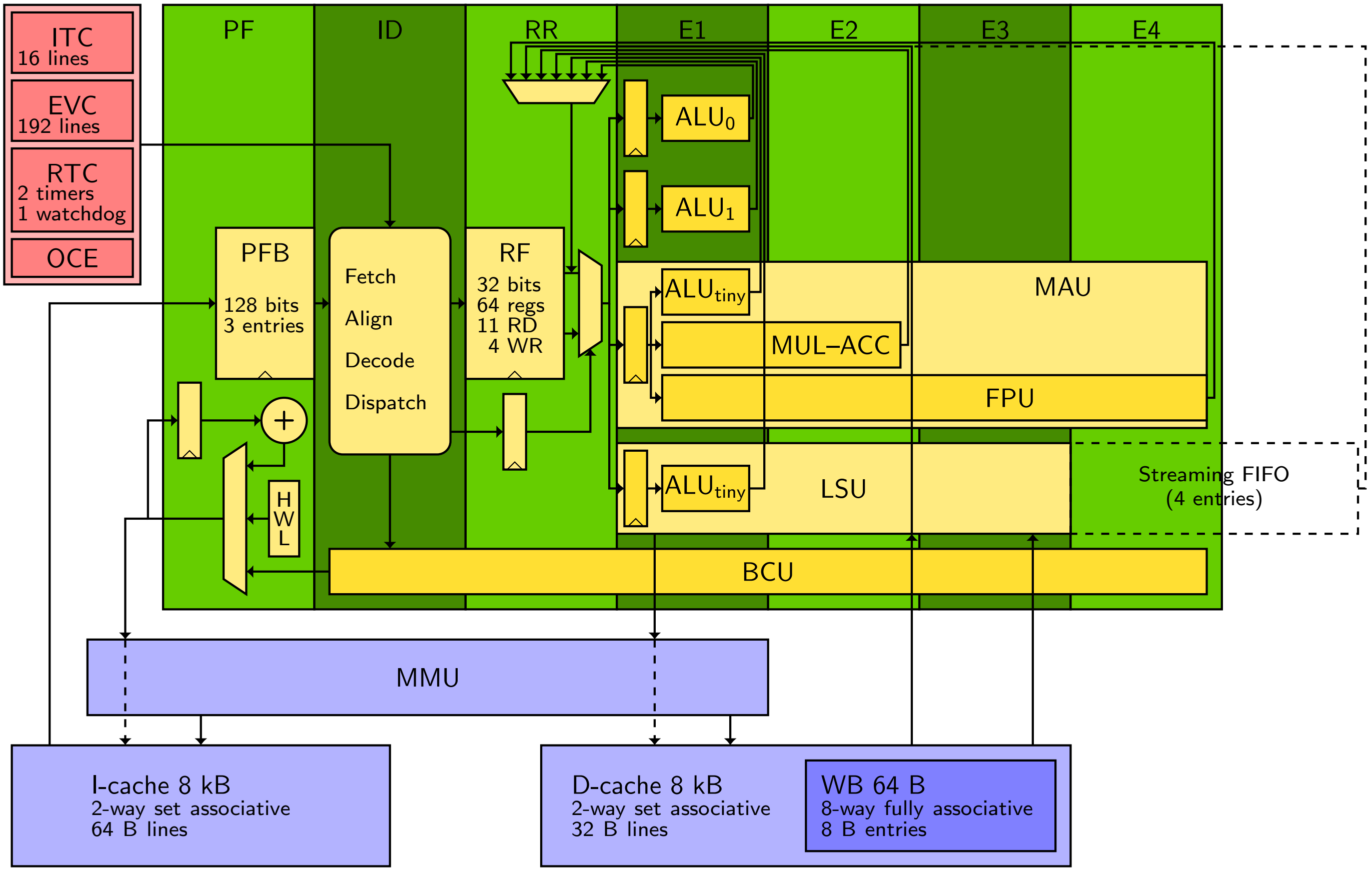
Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:
  - ALU_0 (32 bits) } 64-bit ALU
  - ALU_1 (32 bits) } 64-bit ALU
  - MAU (multiply & add) / FPU (floating point) / ALU_{tiny}
  - LSU (load & store) / ALU_{tiny}

Compute core microarchitecture

- Kalray-1 (K1) microarchitecture
- 32-bit processor:
  - 64 registers of 32 bits each
  - 32- and 64-bit loads / stores
- 5 computation units:
  - ALU_0 (32 bits) } 64-bit ALU
  - ALU_1 (32 bits) } 64-bit ALU
  - MAU (multiply & add) / FPU (floating point) / ALU_{tiny}
  - LSU (load & store) / ALU_{tiny}
  - BCU (branch & control)

Compute core microarchitecture



Compute core microarchitecture

- VLIW : Very Long Instruction Word
  - specify at each clock cycle what each computation unit will do
  - instruction word for a computation unit: 32 or 64 bits
  - one "instruction bundle" issued at each cycle: from 32 to 256 bits

Compute core microarchitecture

- VLIW : Very Long Instruction Word
  - specify at each clock cycle what each computation unit will do
  - instruction word for a computation unit: 32 or 64 bits
  - one "instruction bundle" issued at each cycle: from 32 to 256 bits
- 8-stage pipeline (actually, 5 stages for most instructions)

Compute core microarchitecture

- VLIW : Very Long Instruction Word
  - specify at each clock cycle what each computation unit will do
  - instruction word for a computation unit: 32 or 64 bits
  - one "instruction bundle" issued at each cycle: from 32 to 256 bits
- 8-stage pipeline (actually, 5 stages for most instructions)
- Low branching penalty:
  - 1 cycle for unconditional branches
  - 2 cycles for conditional branches

Compute core microarchitecture

- Low-latency instructions, along with write-back bypass:
  - 1 cycle for ALU instructions (e.g., 32 or 64-bit addition)
  - 2 cycles for MAU instructions (e.g., muladd $64 \leftarrow 32 \times 32 + 64$)

Compute core microarchitecture

- Low-latency instructions, along with write-back bypass:
  - 1 cycle for ALU instructions (e.g., 32 or 64-bit addition)
  - 2 cycles for MAU instructions (e.g., muladd $64 \leftarrow 32 \times 32 + 64$)
- Caches:
  - I-cache and D-cache of 8 kB each
  - fast: 32- or 64-bit cached load in 2 cycles

Compute core microarchitecture

- Low-latency instructions, along with write-back bypass:
  - 1 cycle for ALU instructions (e.g., 32 or 64-bit addition)
  - 2 cycles for MAU instructions (e.g., muladd $64 \leftarrow 32 \times 32 + 64$)
- Caches:
  - I-cache and D-cache of 8 kB each
  - fast: 32- or 64-bit cached load in 2 cycles
- Lots of useful less conventional instructions:
  - zero-penalty hardware loops
  - multiplication of 8×8 matrices over $\mathbb{F}_2$
  - arbitrary boolean functions $\{0,1\}^4 \to \{0,1\}^2$, vectorized on 32 bits
  - etc.

Development

- Toolchain: `k1-gcc`, `k1-ld`, etc.
  - GCC backend for the K1 microarchitecture
  - LIBC port on NodeOS
  - seamless integration in usual C/ASM development environments

Development

- Toolchain: `k1-gcc`, `k1-ld`, etc.
  - GCC backend for the K1 microarchitecture
  - LIBC port on NodeOS
  - seamless integration in usual C/ASM development environments
- An application = (at least) 3 binaries
  - 1 for the host PC (x86-64)
  - 1 for the PCI-e I/O processor (K1)
  - 1 or more for the compute clusters (K1)

Development

- Toolchain: `k1-gcc`, `k1-ld`, etc.
  - GCC backend for the K1 microarchitecture
  - LIBC port on NodeOS
  - seamless integration in usual C/ASM development environments
- An application = (at least) 3 binaries
  - 1 for the host PC (x86-64)
  - 1 for the PCI-e I/O processor (K1)
  - 1 or more for the compute clusters (K1)
- A bit of Makefile magic can take care of everything

Development

- Debugging:
  - simulator $\rightarrow$ execution traces
  - simulator + GDB
  - live debugging with GDB through JTAG port

Development

- Debugging:
  - simulator $\rightarrow$ execution traces
  - simulator + GDB
  - live debugging with GDB through JTAG port
- Optimizing critical code:
  - extensive use of assembly language
  - execution times are very stable: reproducible benchmarks
  - can predict execution times with 1-cycle accuracy

Optimizing at the assembly level

- A few rules should be followed in order to gain a few extra cycles

Optimizing at the assembly level

- A few rules should be followed in order to gain a few extra cycles
- The Pre-Fetch Buffer (PFB) can only fetch 128 bits per clock cycle from the I-cache

  $\Rightarrow$ no more than 128 bits per bundle, so as not to starve the PFB

Optimizing at the assembly level

- A few rules should be followed in order to gain a few extra cycles
- The Pre-Fetch Buffer (PFB) can only fetch 128 bits per clock cycle from the I-cache

  $\Rightarrow$ no more than 128 bits per bundle, so as not to starve the PFB
- 1-cycle penalty for a load accessing two different 64-byte cache lines

  $\Rightarrow$ keep data aligned on 8-byte (64-bit) boundaries in memory

  $\Rightarrow$ pack instruction bundles with `nops` to maintain code alignment

Optimizing at the assembly level

- A few rules should be followed in order to gain a few extra cycles
- The Pre-Fetch Buffer (PFB) can only fetch 128 bits per clock cycle from the I-cache

  ⇒ no more than 128 bits per bundle, so as not to starve the PFB
- 1-cycle penalty for a load accessing two different 64-byte cache lines

  ⇒ keep data aligned on 8-byte (64-bit) boundaries in memory

  ⇒ pack instruction bundles with `nops` to maintain code alignment
- MAU: accumulator and result have to be pairs of registers $\$\mathtt{r}_{2i}$:$\$\mathtt{r}_{2i+1}$

  ⇒ if need be, use an explicit 64-bit addition to avoid this constraint

Outline of the talk

- ECM in a nutshell
- The Kalray MPPA-256 processor
- Multiprecision modular arithmetic
- Results and conclusion

Arithmetic requirements

- For a given integer N to be factored, ECM requires:
  - additions / subtractions modulo N
  - multiplications / squarings modulo N
  - one inversion modulo N
  - two GCDs

Arithmetic requirements

- For a given integer N to be factored, ECM requires:
  - additions / subtractions modulo N
  - multiplications / squarings modulo N
  - one inversion modulo N
  - two GCDs
- Typical size of N: from 192 to 512 bits

Arithmetic requirements

- For a given integer N to be factored, ECM requires:
  - additions / subtractions modulo N
  - multiplications / squarings modulo N
  - one inversion modulo N
  - two GCDs
- Typical size of N: from 192 to 512 bits
- What we have at our disposal:
  - basic integer arithmetic (addition, multiplication, comparisons)
  - on 32- and 64-bit words

Arithmetic requirements

- For a given integer N to be factored, ECM requires:
  - additions / subtractions modulo N
  - multiplications / squarings modulo N
  - one inversion modulo N
  - two GCDs
- Typical size of N: from 192 to 512 bits
- What we have at our disposal:
  - basic integer arithmetic (addition, multiplication, comparisons)
  - on 32- and 64-bit words

$\Rightarrow$ Write an optimized library for multiprecision modular arithmetic

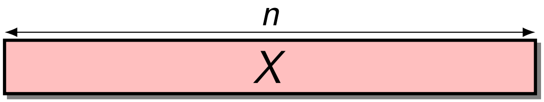
- all low-level functions (add, sub, mul, etc.) in pure ASM
- higher-level functions (GCD, modular inversion) in C
- no multi-threading: all computations on a single compute core

Multiprecision representation

- Consider $X \in \mathbb{Z}/N\mathbb{Z}$, with N an n-bit integer

Multiprecision representation

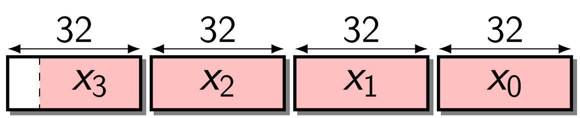
- Consider $X \in \mathbb{Z}/N\mathbb{Z}$, with N an n-bit integer
  - since $0 \leq X < N$, it also fits into n bits



Multiprecision representation

- Consider $X \in \mathbb{Z}/N\mathbb{Z}$, with N an n-bit integer
  - since $0 \leq X < N$, it also fits into n bits
  - split X into $n_W = \lceil n/32 \rceil$ 32-bit words (or limbs):

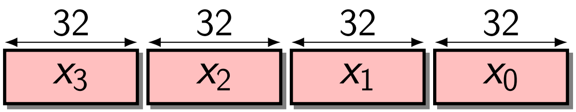
$$X = x_{n_W-1}2^{32(n_W-1)} + \cdots + x_1 2^{32} + x_0$$



Multiprecision representation

- Consider $X \in \mathbb{Z}/N\mathbb{Z}$, with N an n-bit integer
  - since $0 \leq X < N$, it also fits into n bits
  - split X into $n_W = \lceil n/32 \rceil$ 32-bit words (or limbs):

$$X = x_{n_W-1} 2^{32(n_W-1)} + \cdots + x_1 2^{32} + x_0$$

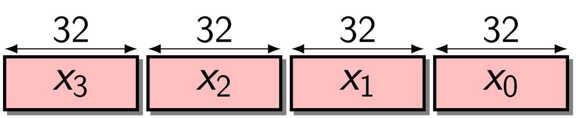


Multiprecision representation

- Consider $X \in \mathbb{Z}/N\mathbb{Z}$, with N an n-bit integer
  - since $0 \leq X < N$, it also fits into n bits
  - split X into $n_W = \lceil n/32 \rceil$ 32-bit words (or limbs):

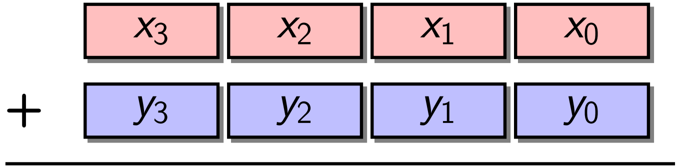
$$X = x_{n_W-1} 2^{32(n_W-1)} + \cdots + x_1 2^{32} + x_0$$

- In our library, n_W is fixed at compile-time:
  - `uint32_t` $X[n_W]$
  - supported values: $2 \leq n_W \leq 16$, i.e. from 64 to 512 bits
  - write (or generate) code optimized for each value of n_W



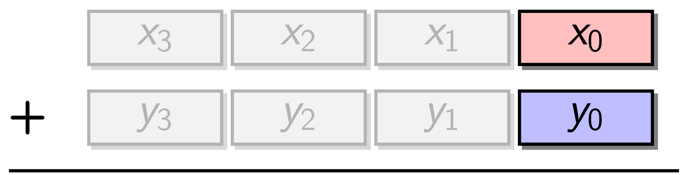
Multiprecision addition

- addn(R, X, Y): addition of two n_W-word integers X and Y



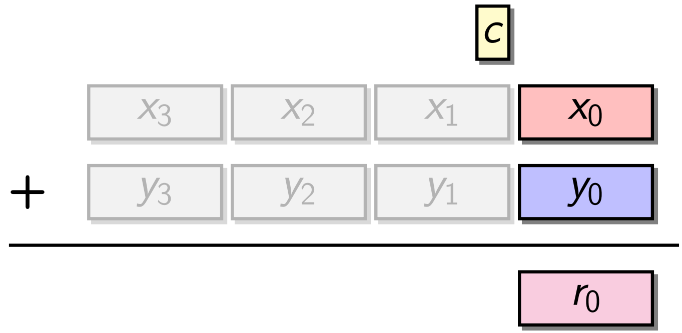
Multiprecision addition

- `addn(`R`, `X`, `Y`)`: addition of two n_W-word integers X and Y
  - right-to-left word-wise addition



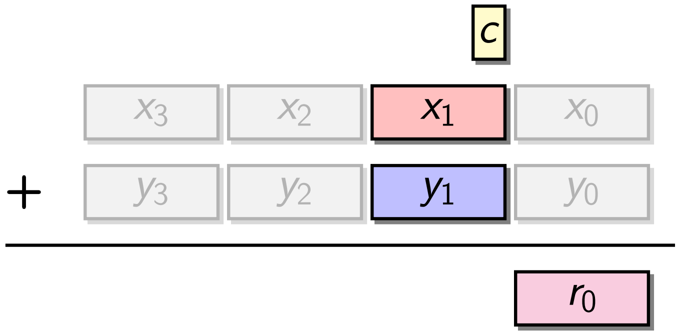
Multiprecision addition

- addn(R, X, Y): addition of two n_W-word integers X and Y
  - right-to-left word-wise addition
  - need to propagate carry



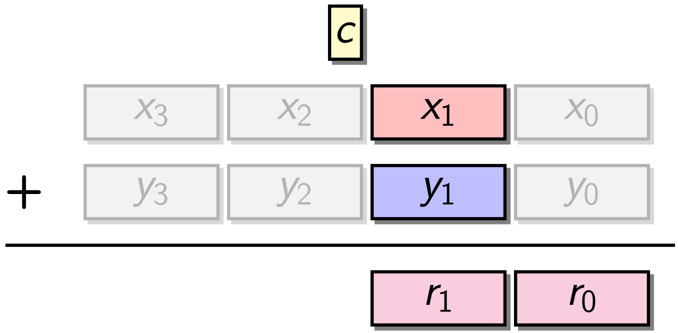
Multiprecision addition

- `addn(`R, X, Y`)`: addition of two n_W-word integers X and Y
  - right-to-left word-wise addition
  - need to propagate carry



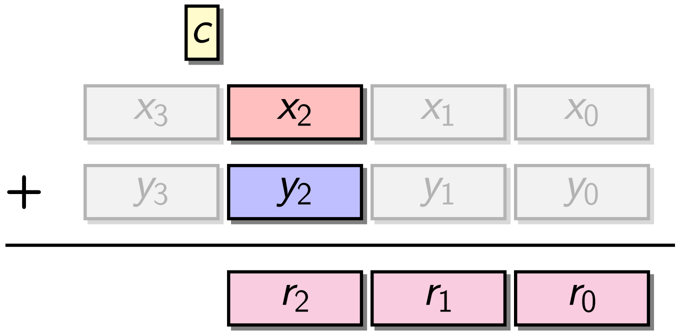
Multiprecision addition

- `addn`(R, X, Y): addition of two n_W-word integers X and Y
  - right-to-left word-wise addition
  - need to propagate carry



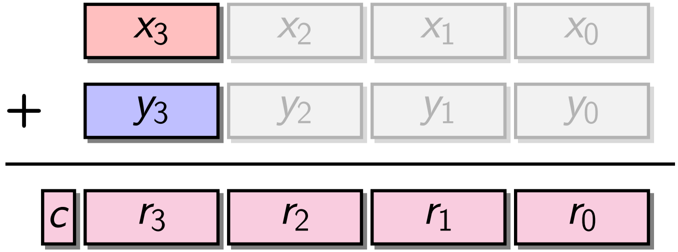
Multiprecision addition

- `addn(`R`,` X`,` Y`)`: addition of two n_W-word integers X and Y
  - right-to-left word-wise addition
  - need to propagate carry



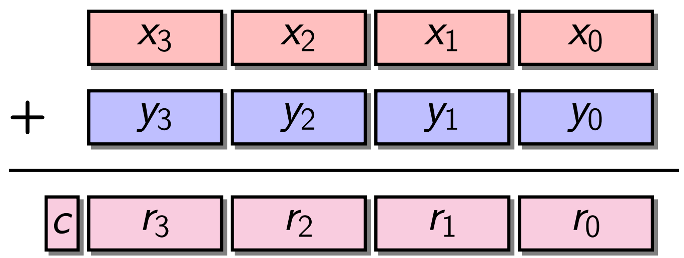
Multiprecision addition

- `addn`(R, X, Y): addition of two n_W-word integers X and Y
  - right-to-left word-wise addition
  - need to propagate carry



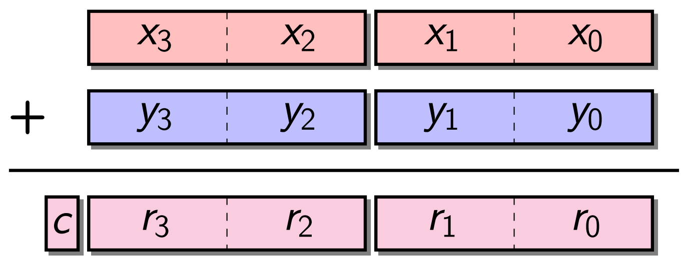
Multiprecision addition

- `addn(`R`,` X`,` Y`)`: addition of two n_W-word integers X and Y
  - right-to-left word-wise addition
  - need to propagate carry



Multiprecision addition

- `addn(`R`,` X`,` Y`)`: addition of two n_W-word integers X and Y
  - right-to-left word-wise addition
  - need to propagate carry
  - use 64-bit additions to halve the number of operations



Multiprecision addition

- `addn`(R, X, Y): addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)

Multiprecision addition

- `addn`(R, X, Y): addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)

cycle	BCU	LSU	MAU	ALU_1	ALU_0

Multiprecision addition

- `addn`(R, X, Y): addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow$ `ld` $8i[X]$			

Multiprecision addition

- `addn`(R, X, Y): addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow$ `ld` $8i[X]$			
$t+1$		$y \leftarrow$ `ld` $8i[Y]$			

Multiprecision addition

- $\texttt{addn}(R, X, Y)$: addition of two n_W-word integers X and Y
  - $\texttt{ld}$ / $\texttt{sd}$: 64-bit memory accesses
  - $\texttt{adddc}$: 64-bit addition with carry (on ALU_0 and ALU_1)

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow \texttt{ld}$ $8i[X]$			
$t+1$		$y \leftarrow \texttt{ld}$ $8i[Y]$			
$t+2$					

Multiprecision addition

- `addn`(R, X, Y): addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow$ `ld` $8i[X]$			
$t+1$		$y \leftarrow$ `ld` $8i[Y]$			
$t+2$					
$t+3$				$r \leftarrow$ `adddc` x, y	

Multiprecision addition

- `addn`(R, X, Y): addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)

cycle	BCU	LSU	MAU	ALU_1	ALU_0
…					
t		$x \leftarrow$ `ld` $8i[X]$			
$t+1$		$y \leftarrow$ `ld` $8i[Y]$			
$t+2$					
$t+3$				$r \leftarrow$ `adddc` x, y	
$t+4$		$8i[R] \leftarrow$ `sd` r			

Multiprecision addition

- `addn(`R`, `X`, `Y`)`: addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)
  - total latency: $5\lceil n_W/2 \rceil + O(1)$ cycles

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow$ `ld` $8i$`[`X`]`			
$t+1$		$y \leftarrow$ `ld` $8i$`[`Y`]`			
$t+2$					
$t+3$				$r \leftarrow$ `adddc` x`,` y	
$t+4$		$8i$`[`R`]` $\leftarrow$ `sd` r			
$t+5$		$x \leftarrow$ `ld` $8(i+1)$`[`X`]`			
$t+6$		$y \leftarrow$ `ld` $8(i+1)$`[`Y`]`			
$t+7$					
$t+8$				$r \leftarrow$ `adddc` x`,` y	
$t+9$		$8(i+1)$`[`R`]` $\leftarrow$ `sd` r			
...					

Multiprecision addition

- `addn(`R`,` X`,` Y`)`: addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)
  - total latency: $5\lceil n_W/2\rceil + O(1)$ cycles

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow$ `ld` $8i$`[`X`]`			
$t+1$		$y \leftarrow$ `ld` $8i$`[`Y`]`			
$t+2$		$x \leftarrow$ `ld` $8(i+1)$`[`X`]`			
$t+3$		$y \leftarrow$ `ld` $8(i+1)$`[`Y`]`		$r \leftarrow$ `adddc` x`,` y	
$t+4$		$8i$`[`R`]` $\leftarrow$ `sd` r			
$t+5$				$r \leftarrow$ `adddc` x`,` y	
$t+6$		$8(i+1)$`[`R`]` $\leftarrow$ `sd` r			
$t+7$					
$t+8$					
$t+9$					
...					

Multiprecision addition

- `addn(`R`,` X`,` Y`)`: addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)
  - total latency: $5\lceil n_W/2\rceil + O(1)$ cycles

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow$ `ld` $8i$`[`X`]`			
$t+1$		$y \leftarrow$ `ld` $8i$`[`Y`]`			
$t+2$					
$t+3$		$x \leftarrow$ `ld` $8(i+1)$`[`X`]`		$r \leftarrow$ `adddc` x`,` y	
$t+4$		$y \leftarrow$ `ld` $8(i+1)$`[`Y`]`			
$t+5$		$8i$`[`R`]` $\leftarrow$ `sd` r			
$t+6$				$r \leftarrow$ `adddc` x`,` y	
$t+7$					
$t+8$		$8(i+1)$`[`R`]` $\leftarrow$ `sd` r			
$t+9$					
...					

Multiprecision addition

- addn(R, X, Y): addition of two n_W-word integers X and Y
  - ld / sd: 64-bit memory accesses
  - adddc: 64-bit addition with carry (on ALU_0 and ALU_1)
  - total latency: $5\lceil n_W/2 \rceil + O(1)$ cycles

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow$ ld $8i$[X]		$r \leftarrow$ adddc x, y	
$t+1$		$y \leftarrow$ ld $8i$[Y]			
$t+2$		$8(i-1)$[R] $\leftarrow$ sd r			
$t+3$		$x \leftarrow$ ld $8(i+1)$[X]		$r \leftarrow$ adddc x, y	
$t+4$		$y \leftarrow$ ld $8(i+1)$[Y]			
$t+5$		$8i$[R] $\leftarrow$ sd r			
$t+6$				$r \leftarrow$ adddc x, y	
$t+7$					
$t+8$		$8(i+1)$[R] $\leftarrow$ sd r			
$t+9$					
...					

Multiprecision addition

- `addn(`R`,` X`,` Y`)`: addition of two n_W-word integers X and Y
  - `ld` / `sd`: 64-bit memory accesses
  - `adddc`: 64-bit addition with carry (on ALU_0 and ALU_1)
  - total latency: $5\lceil n_W/2\rceil + O(1)$ cycles

cycle	BCU	LSU	MAU	ALU_1	ALU_0
...					
t		$x \leftarrow$ `ld` $8i$`[`X`]`		$r \leftarrow$ `adddc` x, y	
$t+1$		$y \leftarrow$ `ld` $8i$`[`Y`]`			
$t+2$		$8(i-1)$`[`R`]` $\leftarrow$ `sd` r			
$t+3$		$x \leftarrow$ `ld` $8(i+1)$`[`X`]`		$r \leftarrow$ `adddc` x, y	
$t+4$		$y \leftarrow$ `ld` $8(i+1)$`[`Y`]`			
$t+5$		$8i$`[`R`]` $\leftarrow$ `sd` r			
$t+6$		$x \leftarrow$ `ld` $8(i+2)$`[`X`]`		$r \leftarrow$ `adddc` x, y	
$t+7$		$y \leftarrow$ `ld` $8(i+2)$`[`Y`]`			
$t+8$		$8(i+1)$`[`R`]` $\leftarrow$ `sd` r			
$t+9$				$r \leftarrow$ `adddc` x, y	
...					

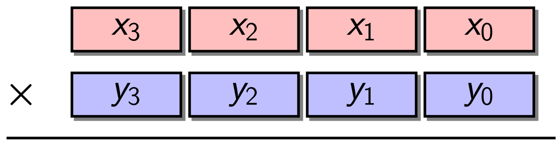
Multiprecision addition

- addn(R, X, Y): addition of two n_W-word integers X and Y
  - ld / sd: 64-bit memory accesses
  - adddc: 64-bit addition with carry (on ALU_0 and ALU_1)
  - total latency: $3\lceil n_W/2\rceil + O(1)$ cycles

cycle	BCU	LSU	MAU	ALU_1	ALU_0
…					
t		$x \leftarrow$ ld $8i[X]$		$r \leftarrow$ adddc x, y	
$t+1$		$y \leftarrow$ ld $8i[Y]$			
$t+2$		$8(i-1)[R] \leftarrow$ sd r			
$t+3$		$x \leftarrow$ ld $8(i+1)[X]$		$r \leftarrow$ adddc x, y	
$t+4$		$y \leftarrow$ ld $8(i+1)[Y]$			
$t+5$		$8i[R] \leftarrow$ sd r			
$t+6$		$x \leftarrow$ ld $8(i+2)[X]$		$r \leftarrow$ adddc x, y	
$t+7$		$y \leftarrow$ ld $8(i+2)[Y]$			
$t+8$		$8(i+1)[R] \leftarrow$ sd r			
$t+9$		$x \leftarrow$ ld $8(i+3)[X]$		$r \leftarrow$ adddc x, y	
…					

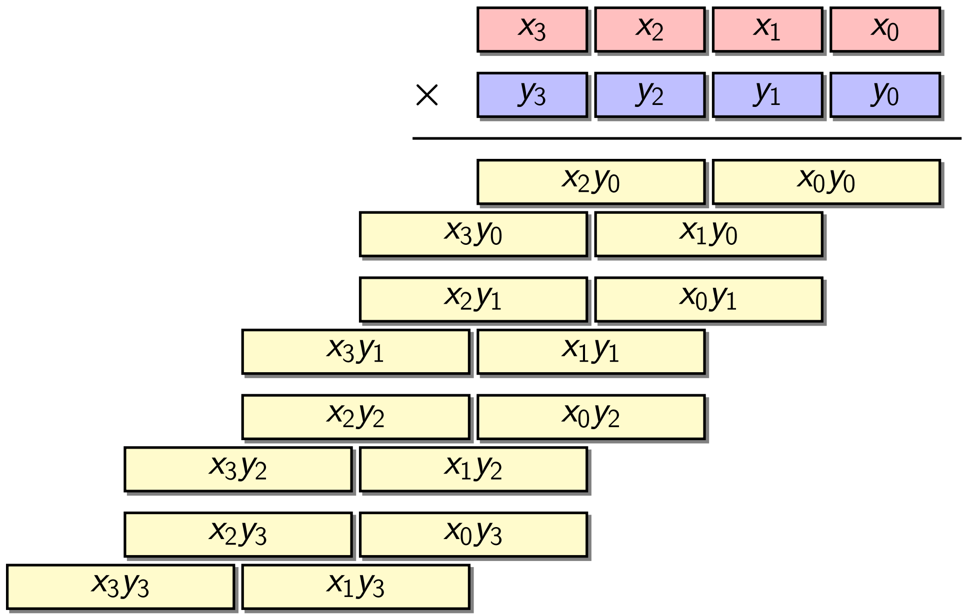
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y



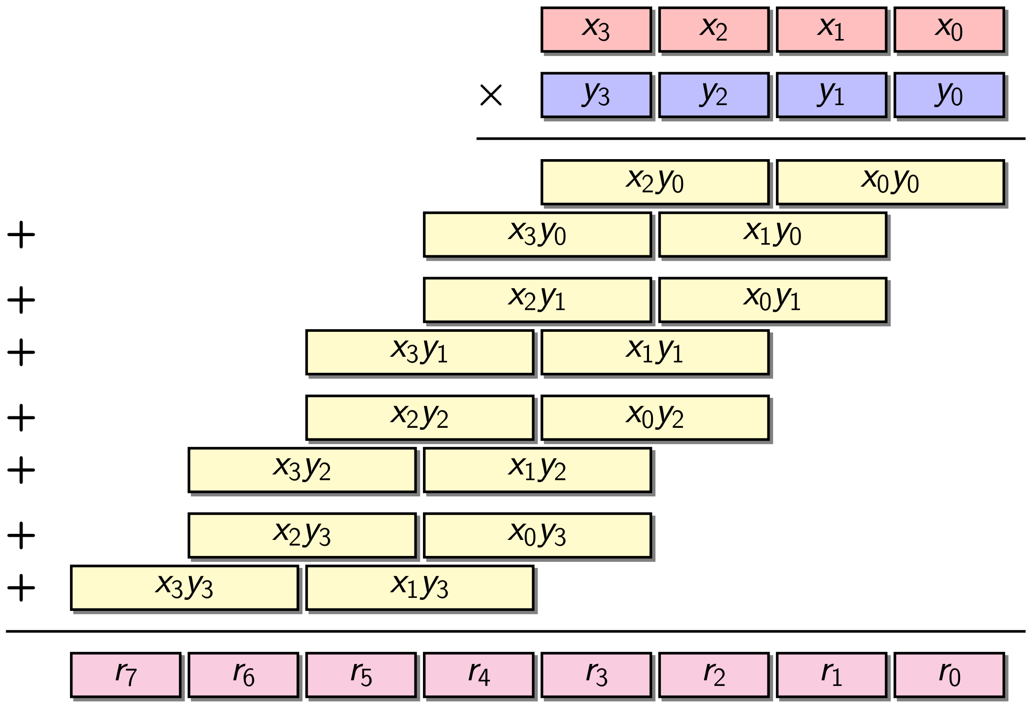
Multiprecision multiplication

- muln(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts



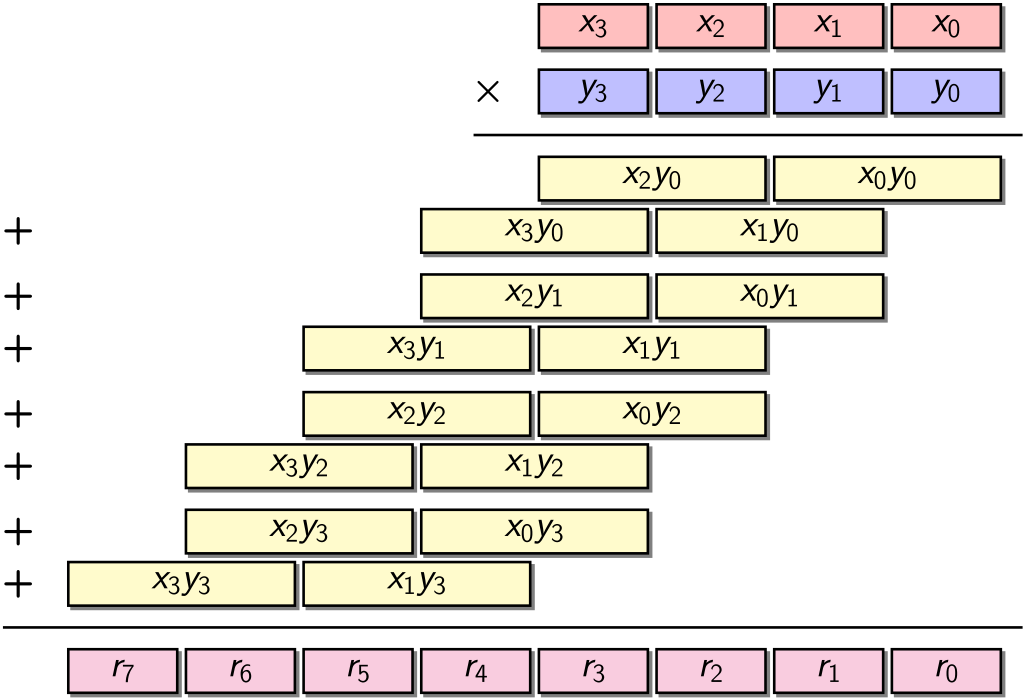
Multiprecision multiplication

- muln(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words



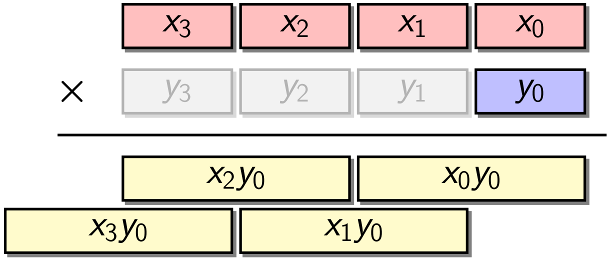
Multiprecision multiplication

- `muln(`R, X, Y`)`: multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



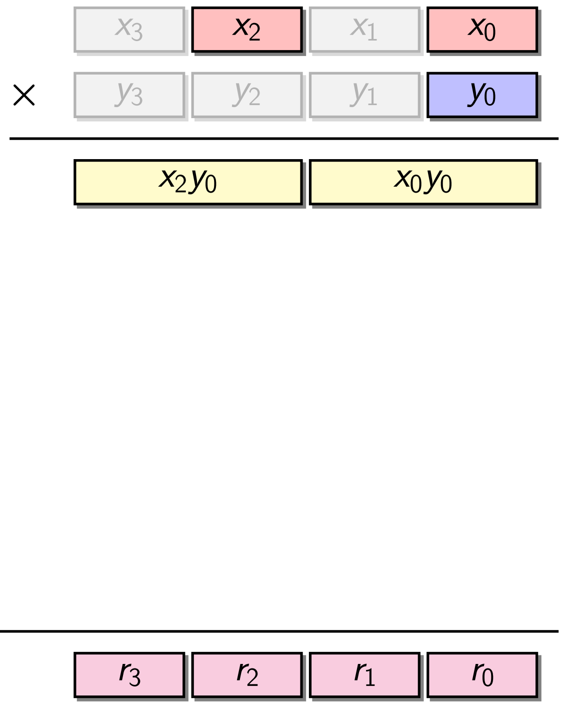
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



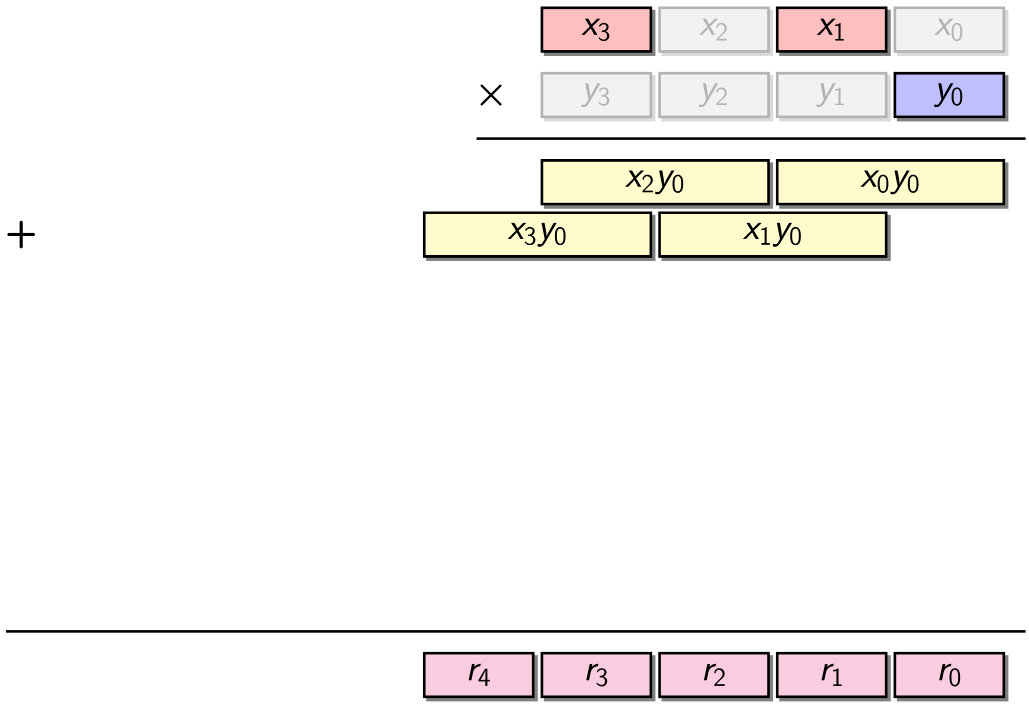
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



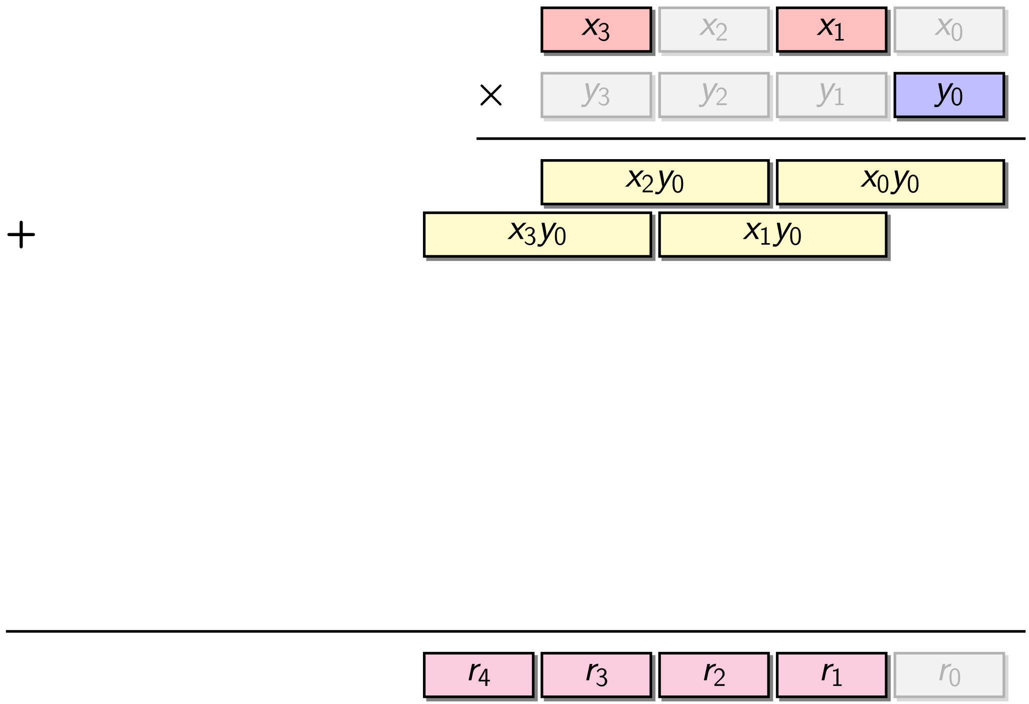
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



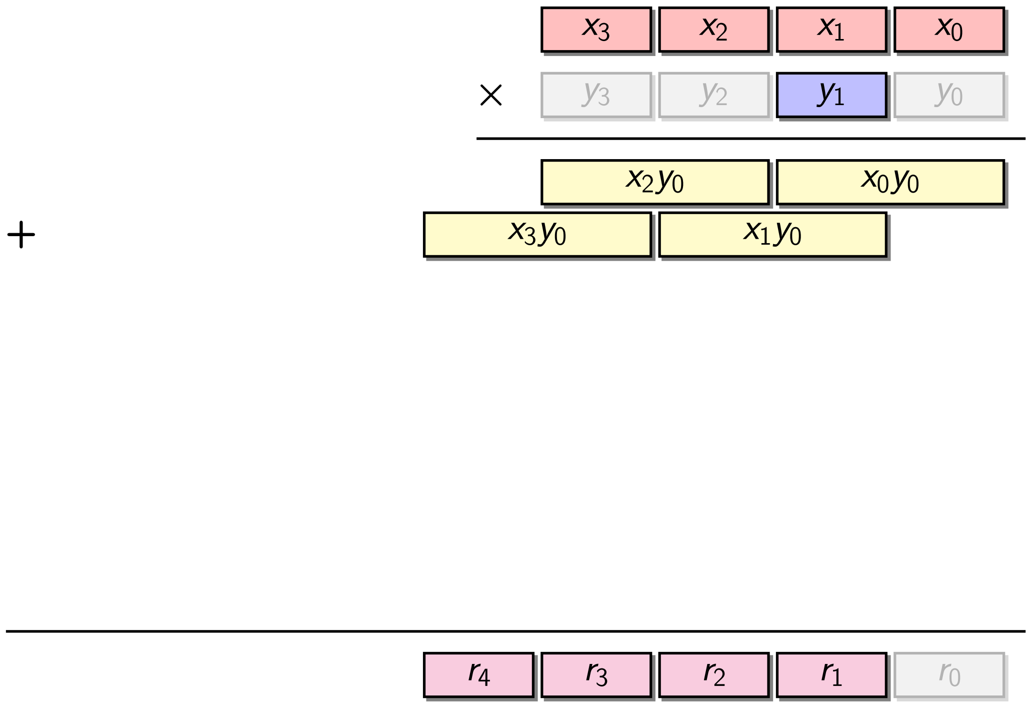
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



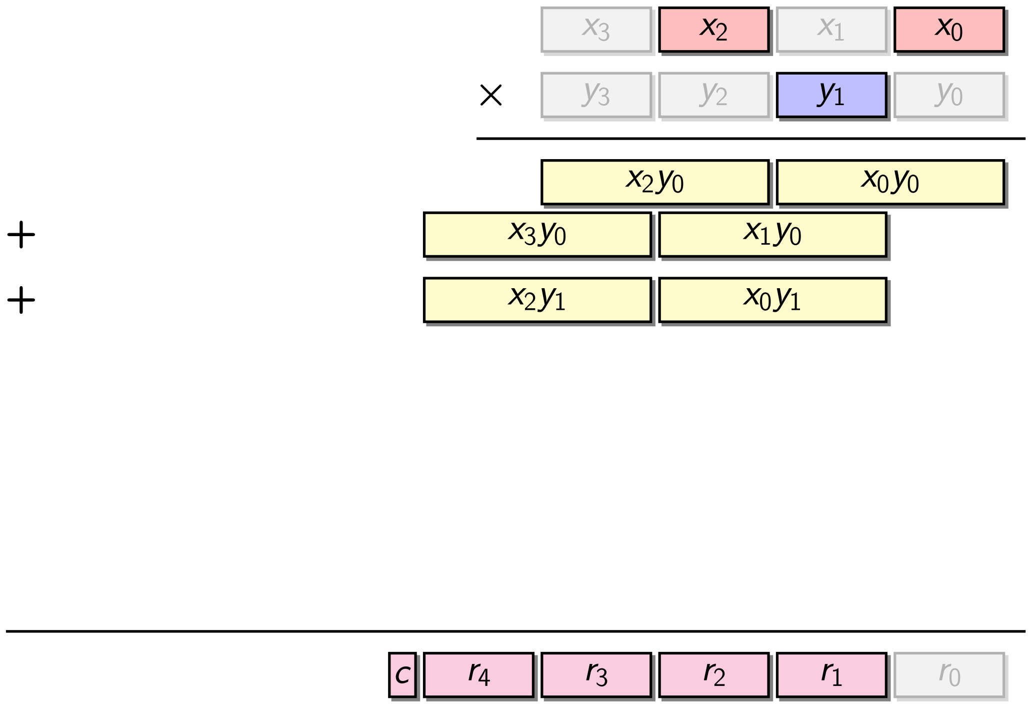
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



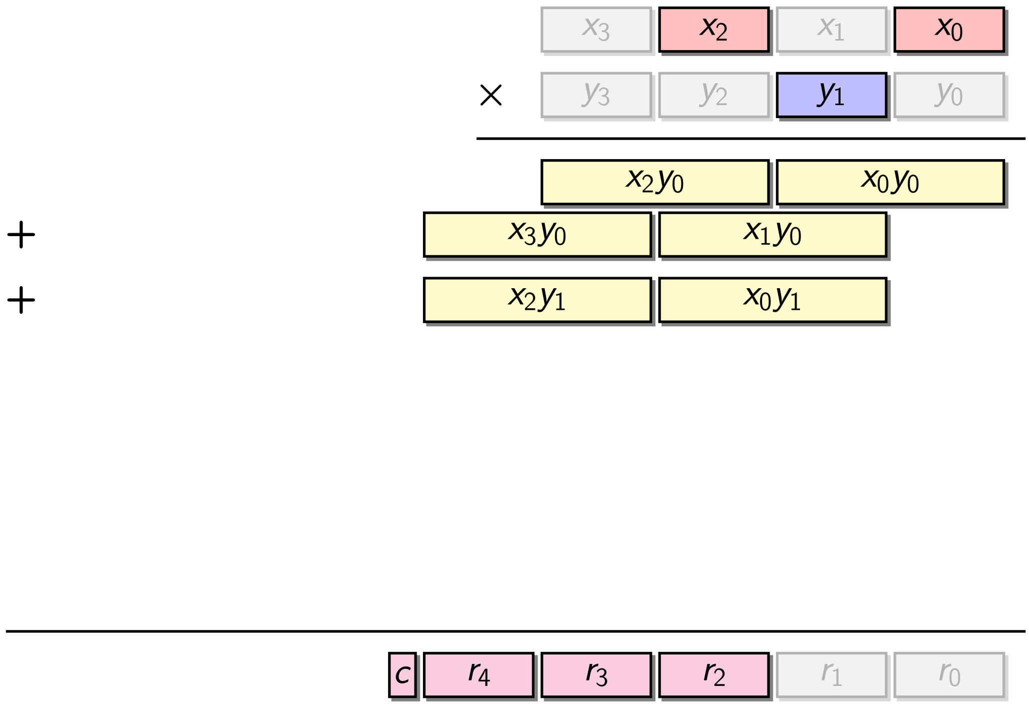
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



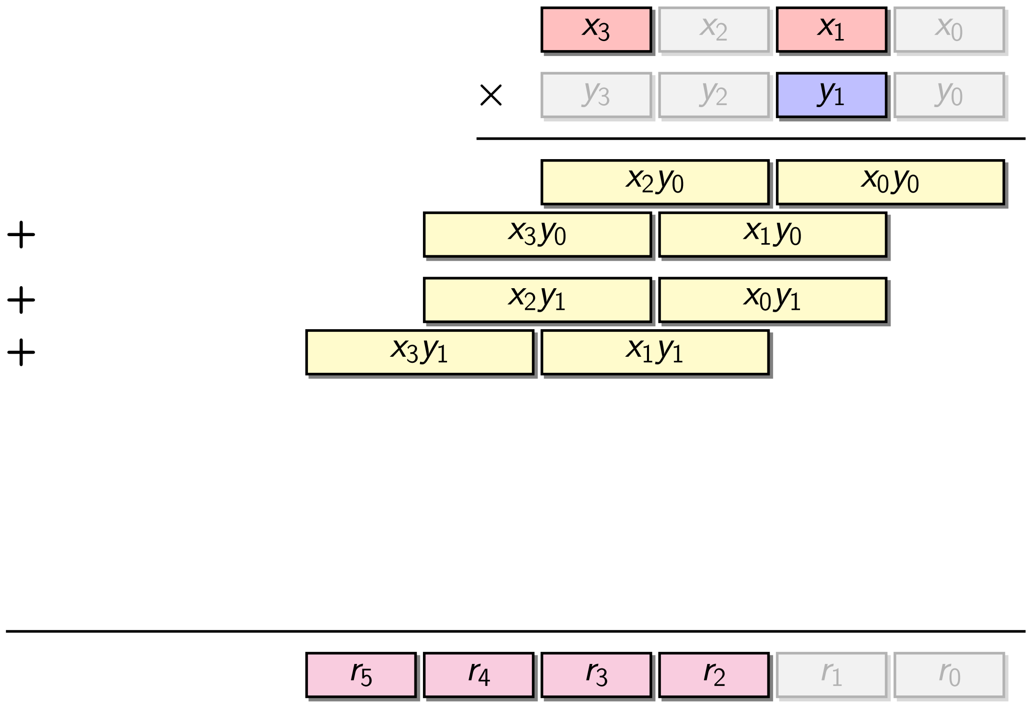
Multiprecision multiplication

- muln(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



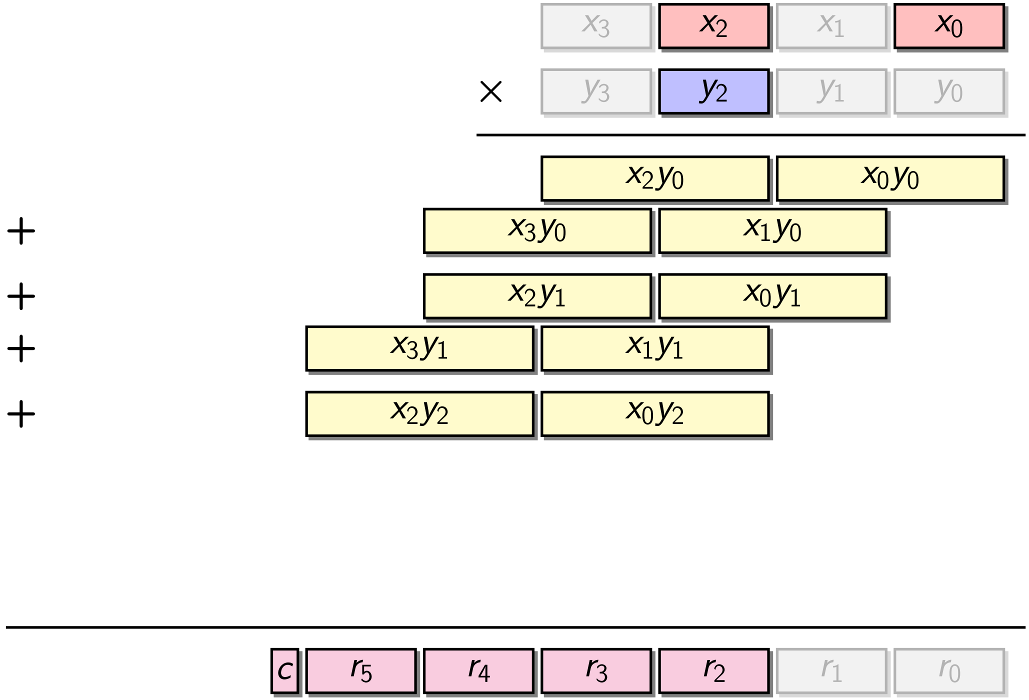
Multiprecision multiplication

- `muln(`R`, `X`, `Y`)`: multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



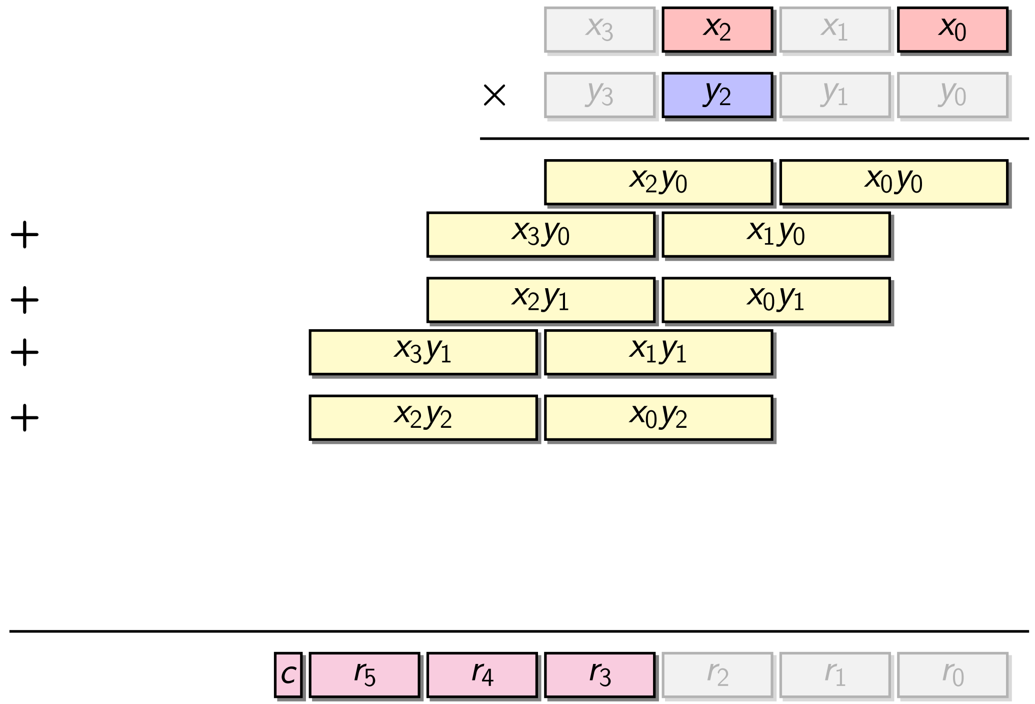
Multiprecision multiplication

- $\texttt{muln}(R, X, Y)$: multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



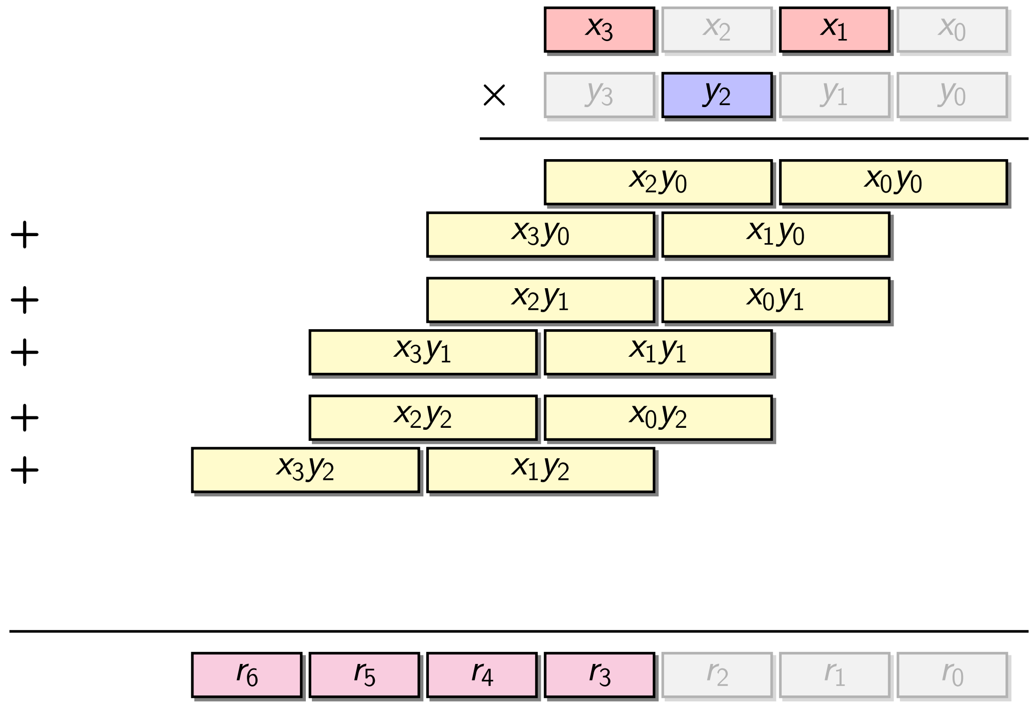
Multiprecision multiplication

- `muln(`R, X, Y`)`: multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



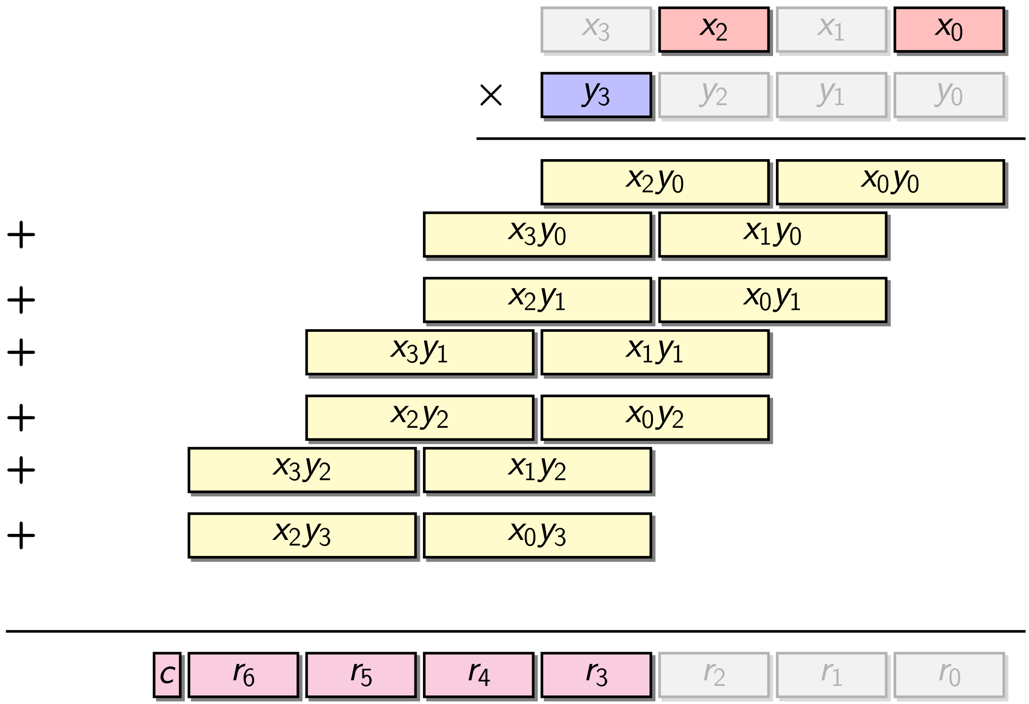
Multiprecision multiplication

- muln(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



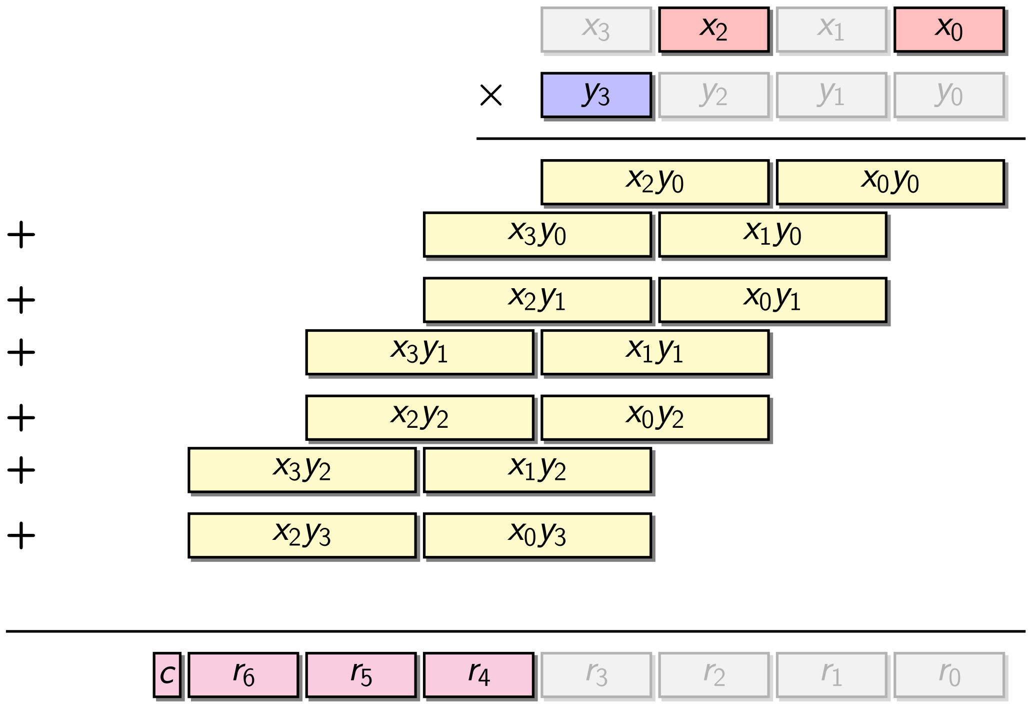
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



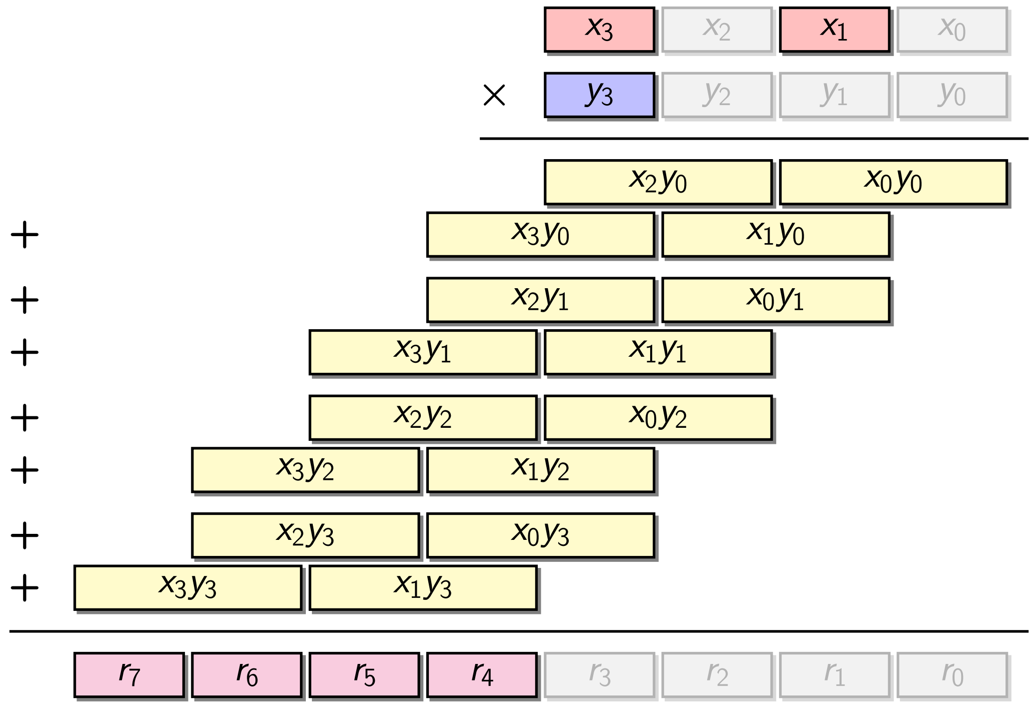
Multiprecision multiplication

- muln(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



Multiprecision multiplication

- muln(R, X, Y): multiplication of two n_W-word integers X and Y
  - schoolbook method: n_W^2 32-by-32-bit subproducts
  - final product fits into $2n_W$ words
  - order for subproducts: operand scanning (simpler loop control)



Multiprecision multiplication

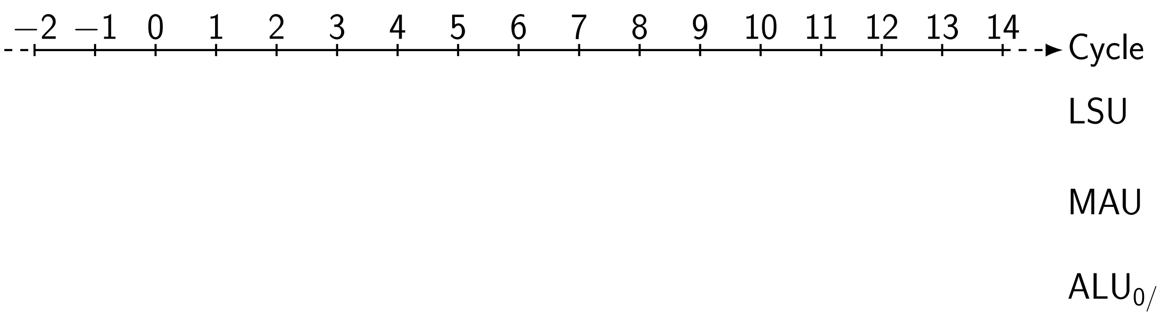
- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y

Multiprecision multiplication

- `muln(`R, X, Y`)`: multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products

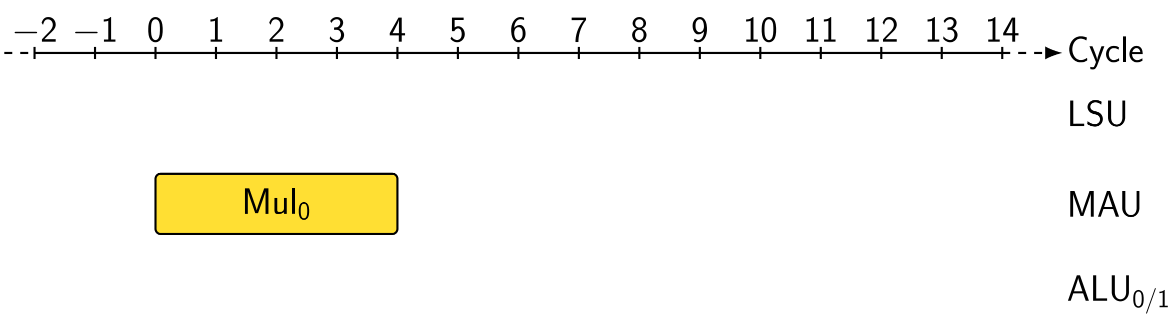
Multiprecision multiplication

- `muln(`R`,` X`,` Y`)`: multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products
- Basic iteration: $Acc \leftarrow Acc + X \times y_i$ (a.k.a. `addmul_1` in GMP)



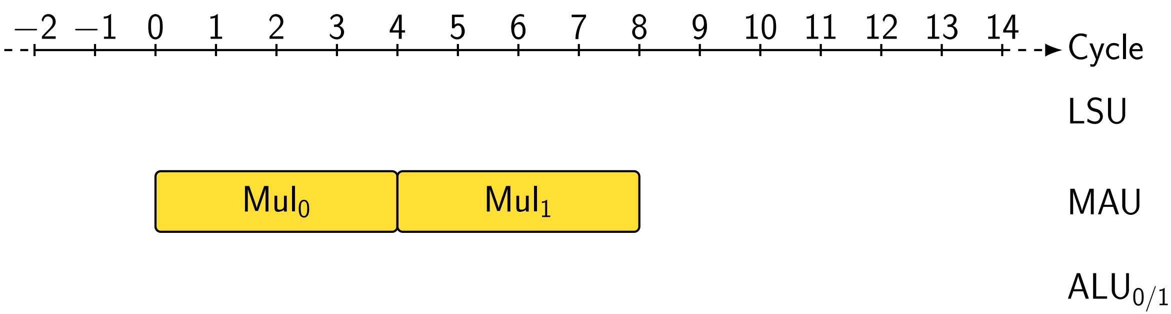
Multiprecision multiplication

- `muln(`R, X, Y`)`: multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products
- Basic iteration: $Acc \leftarrow Acc + X \times y_i$ (a.k.a. `addmul_1` in GMP)
  - first the even-rank partial products $x_{2j} \times y_i$



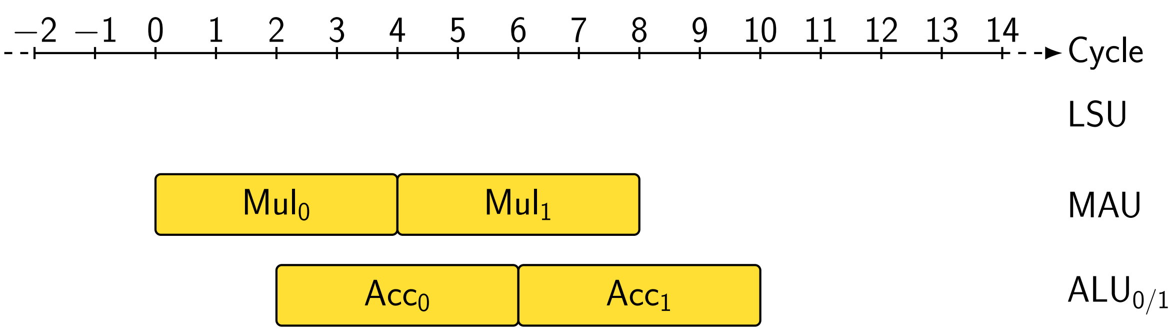
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products
- Basic iteration: $Acc \leftarrow Acc + X \times y_i$ (a.k.a. `addmul_1` in GMP)
  - first the even-rank partial products $x_{2j} \times y_i$
  - then the odd-rank partial products $x_{2j+1} \times y_i$



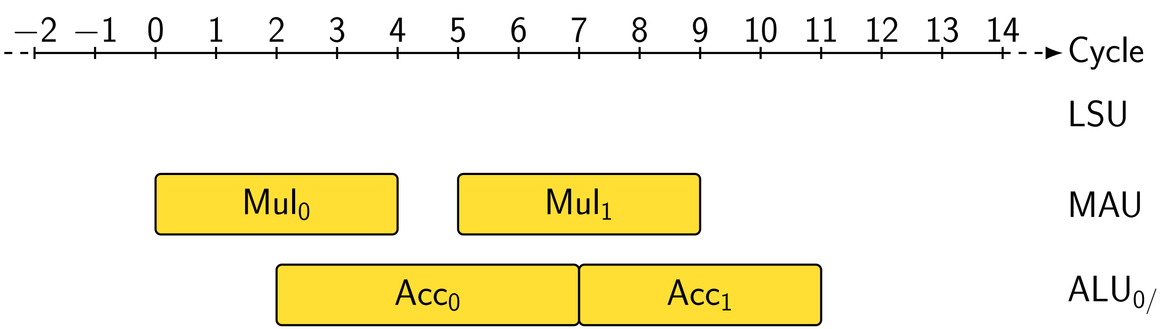
Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products
- Basic iteration: $Acc \leftarrow Acc + X \times y_i$ (a.k.a. `addmul_1` in GMP)
  - first the even-rank partial products $x_{2j} \times y_i$
  - then the odd-rank partial products $x_{2j+1} \times y_i$
  - constraints on register pairs: explicit accumulation using `adddc`



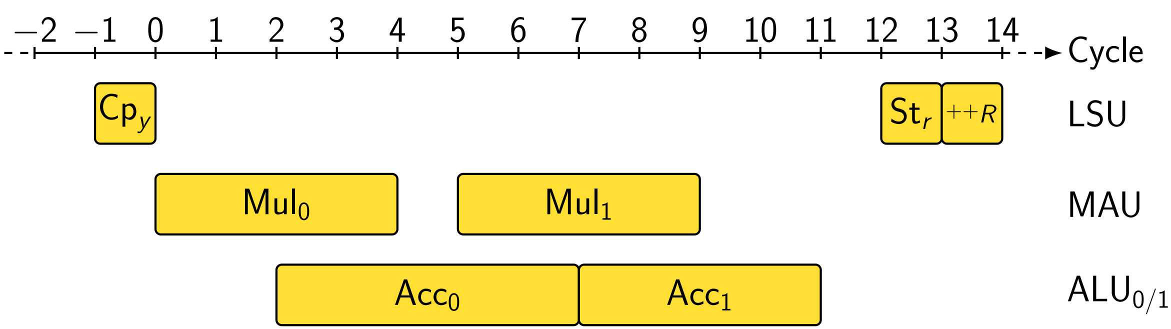
Multiprecision multiplication

- `muln(`R`,`X`,`Y`)`: multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products
- Basic iteration: $Acc \leftarrow Acc + X \times y_i$ (a.k.a. `addmul_1` in GMP)
  - first the even-rank partial products $x_{2j} \times y_i$
  - then the odd-rank partial products $x_{2j+1} \times y_i$
  - constraints on register pairs: explicit accumulation using `adddc`
  - Acc_0 requires an extra cycle to store the output carry



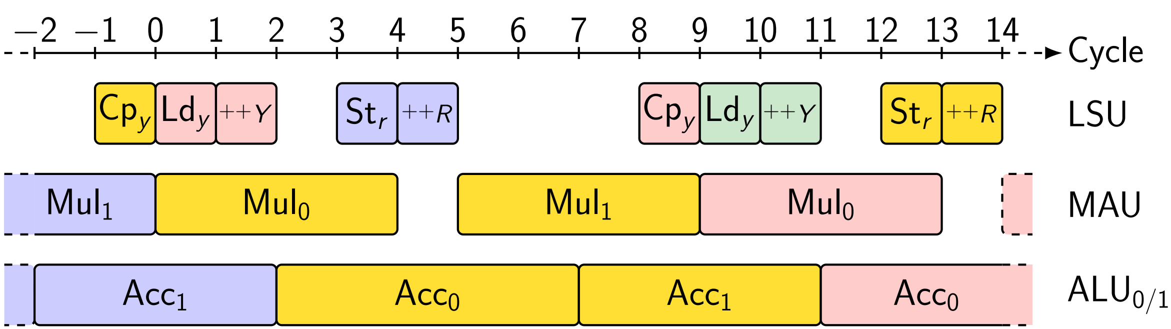
Multiprecision multiplication

- `muln(R, X, Y)`: multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products
- Basic iteration: $Acc \leftarrow Acc + X \times y_i$ (a.k.a. `addmul_1` in GMP)
  - first the even-rank partial products $x_{2j} \times y_i$
  - then the odd-rank partial products $x_{2j+1} \times y_i$
  - constraints on register pairs: explicit accumulation using `adddc`
  - Acc_0 requires an extra cycle to store the output carry



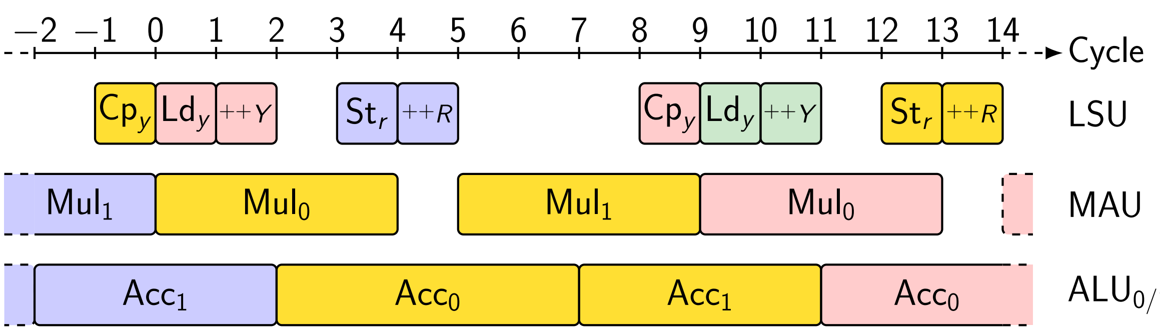
Multiprecision multiplication

- `muln(`R`,`X`,`Y`)`: multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products
- Basic iteration: $Acc \leftarrow Acc + X \times y_i$ (a.k.a. `addmul_1` in GMP)
  - first the even-rank partial products $x_{2j} \times y_i$
  - then the odd-rank partial products $x_{2j+1} \times y_i$
  - constraints on register pairs: explicit accumulation using `adddc`
  - Acc_0 requires an extra cycle to store the output carry
  - repeated n_W times



Multiprecision multiplication

- `muln`(R, X, Y): multiplication of two n_W-word integers X and Y
  - multiplicand X kept in the register file (whence $n_W \leq 16$)
  - multiplier Y processed sequentially, word by word
  - n_W-word accumulator for partial products
- Basic iteration: $Acc \leftarrow Acc + X \times y_i$ (a.k.a. `addmul_1` in GMP)
  - first the even-rank partial products $x_{2j} \times y_i$
  - then the odd-rank partial products $x_{2j+1} \times y_i$
  - constraints on register pairs: explicit accumulation using `adddc`
  - Acc_0 requires an extra cycle to store the output carry
  - repeated n_W times
- Total latency: $n_W(n_W + 1) + O(1)$ cycles, **≈ 1 cycle / subproduct**



Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)

Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32 n_W}) \bmod N$

Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32n_W}) \bmod N$
  - precomputation: $\mu \leftarrow (-N)^{-1} \bmod 2^{32}$

Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32n_W}) \bmod N$
  - precomputation: $\mu \leftarrow (-N)^{-1} \bmod 2^{32}$
  - repeat n_W times:

Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32n_W}) \bmod N$
  - precomputation: $\mu \leftarrow (-N)^{-1} \bmod 2^{32}$
  - repeat n_W times:
    - $q \leftarrow (x_0 \cdot \mu) \bmod 2^{32}$ // multiplication $32 \leftarrow 32 \times 32$

Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32n_W}) \bmod N$
  - precomputation: $\mu \leftarrow (-N)^{-1} \bmod 2^{32}$
  - repeat n_W times:
    - $q \leftarrow (x_0 \cdot \mu) \bmod 2^{32}$ // multiplication $32 \leftarrow 32 \times 32$
    - $Y \leftarrow X + q \cdot N$ // `addmul_1`; $Y \equiv 0 \pmod{2^{32}}$

Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32n_W}) \bmod N$
  - precomputation: $\mu \leftarrow (-N)^{-1} \bmod 2^{32}$
  - repeat n_W times:
    - $q \leftarrow (x_0 \cdot \mu) \bmod 2^{32}$ // multiplication $32 \leftarrow 32 \times 32$
    - $Y \leftarrow X + q \cdot N$ // `addmul_1`; $Y \equiv 0 \pmod{2^{32}}$
    - $X \leftarrow Y/2^{32}$ // exact division, for free

Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32n_W}) \bmod N$
  - precomputation: $\mu \leftarrow (-N)^{-1} \bmod 2^{32}$
  - repeat n_W times:
    - $q \leftarrow (x_0 \cdot \mu) \bmod 2^{32}$ // multiplication $32 \leftarrow 32 \times 32$
    - $Y \leftarrow X + q \cdot N$ // `addmul_1`; $Y \equiv 0 \pmod{2^{32}}$
    - $X \leftarrow Y/2^{32}$ // exact division, for free
  - if $X \geq N$ then $X \leftarrow X - N$

Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32n_W}) \bmod N$
  - precomputation: $\mu \leftarrow (-N)^{-1} \bmod 2^{32}$
  - repeat n_W times:
    - $q \leftarrow (x_0 \cdot \mu) \bmod 2^{32}$ // multiplication $32 \leftarrow 32 \times 32$
    - $Y \leftarrow X + q \cdot N$ // `addmul_1`; $Y \equiv 0 \pmod{2^{32}}$
    - $X \leftarrow Y/2^{32}$ // exact division, for free
  - if $X \geq N$ then $X \leftarrow X - N$
- Total latency: $n_W(n_W + 3) + O(1)$ cycles, still ≈ 1 cycle / subproduct

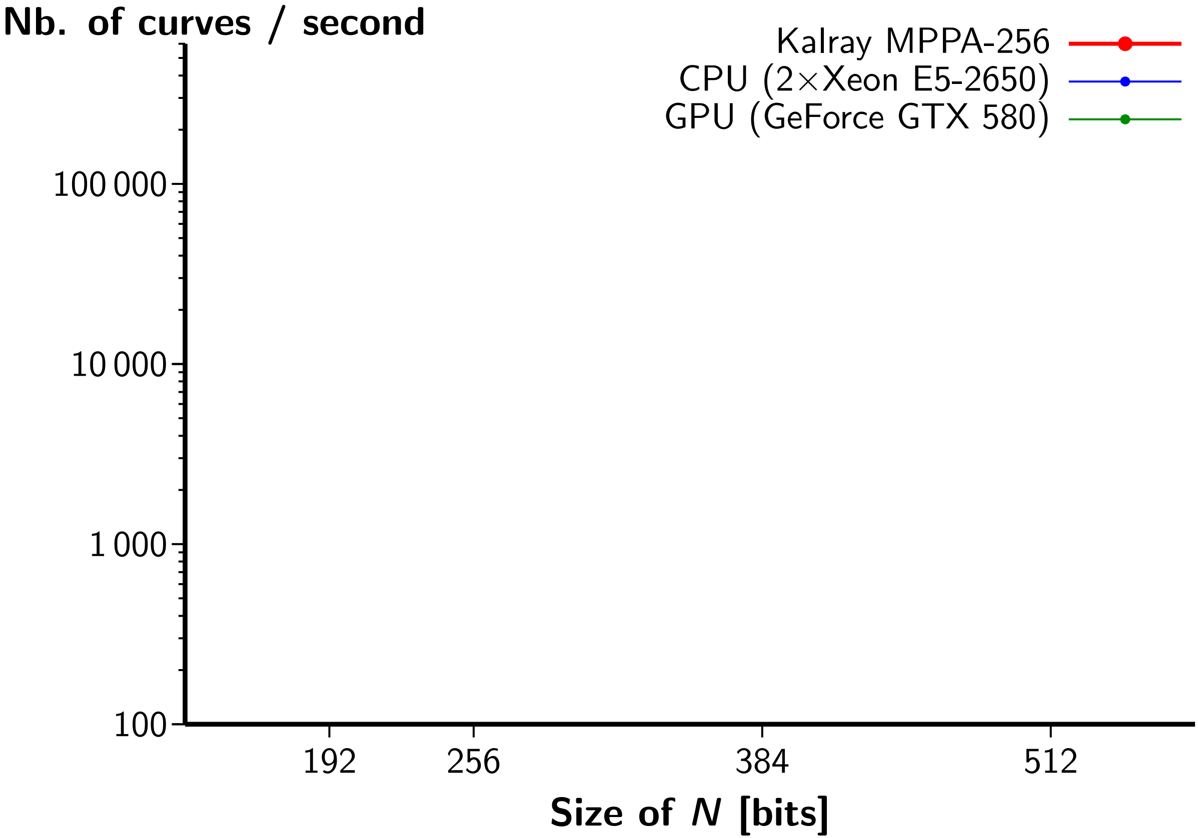
Modular arithmetic

- Perform computations modulo N (n_W words)
  - addition, subtraction: trivial
  - multiplication: use Montgomery reduction (REDC)
- REDC : $X \mapsto (X/2^{32n_W}) \bmod N$
  - precomputation: $\mu \leftarrow (-N)^{-1} \bmod 2^{32}$
  - repeat n_W times:
    - $q \leftarrow (x_0 \cdot \mu) \bmod 2^{32}$ // multiplication $32 \leftarrow 32 \times 32$
    - $Y \leftarrow X + q \cdot N$ // `addmul_1`; $Y \equiv 0 \pmod{2^{32}}$
    - $X \leftarrow Y/2^{32}$ // exact division, for free
  - if $X \geq N$ then $X \leftarrow X - N$
- Total latency: $n_W(n_W + 3) + O(1)$ cycles, still ≈ 1 cycle / subproduct
- Multiplication then REDC: $2n_W(n_W + 2) + O(1)$ cycles
  - for $n_W = 16$, in ASM: 614 cycles
  - same thing in C: 3221 cycles!

Outline of the talk

- ECM in a nutshell
- The Kalray MPPA-256 processor
- Multiprecision modular arithmetic
- Results and conclusion

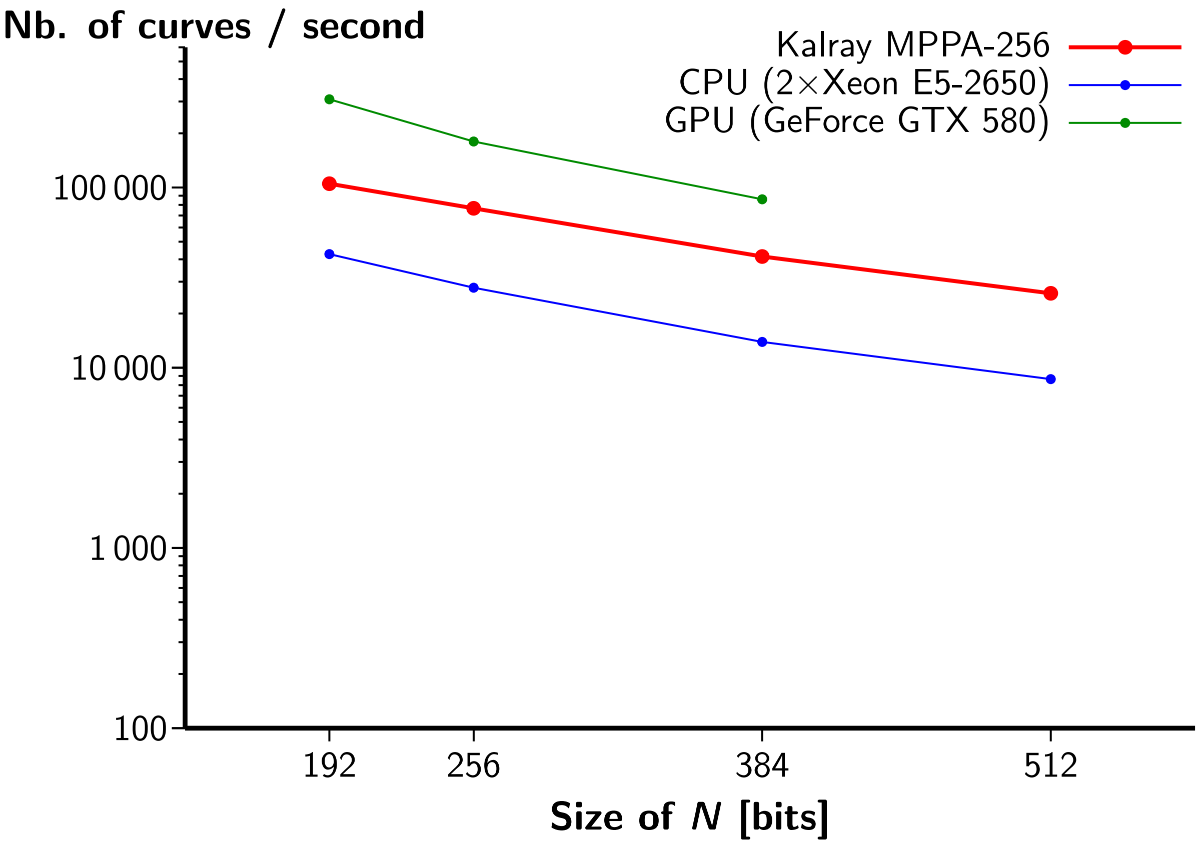
Throughput



Nb. of curves / second
Kalray MPPA-256
CPU (2×Xeon E5-2650)
GPU (GeForce GTX 580)
100 000
10 000
1 000
100
192
256
384
512
Size of N [bits]

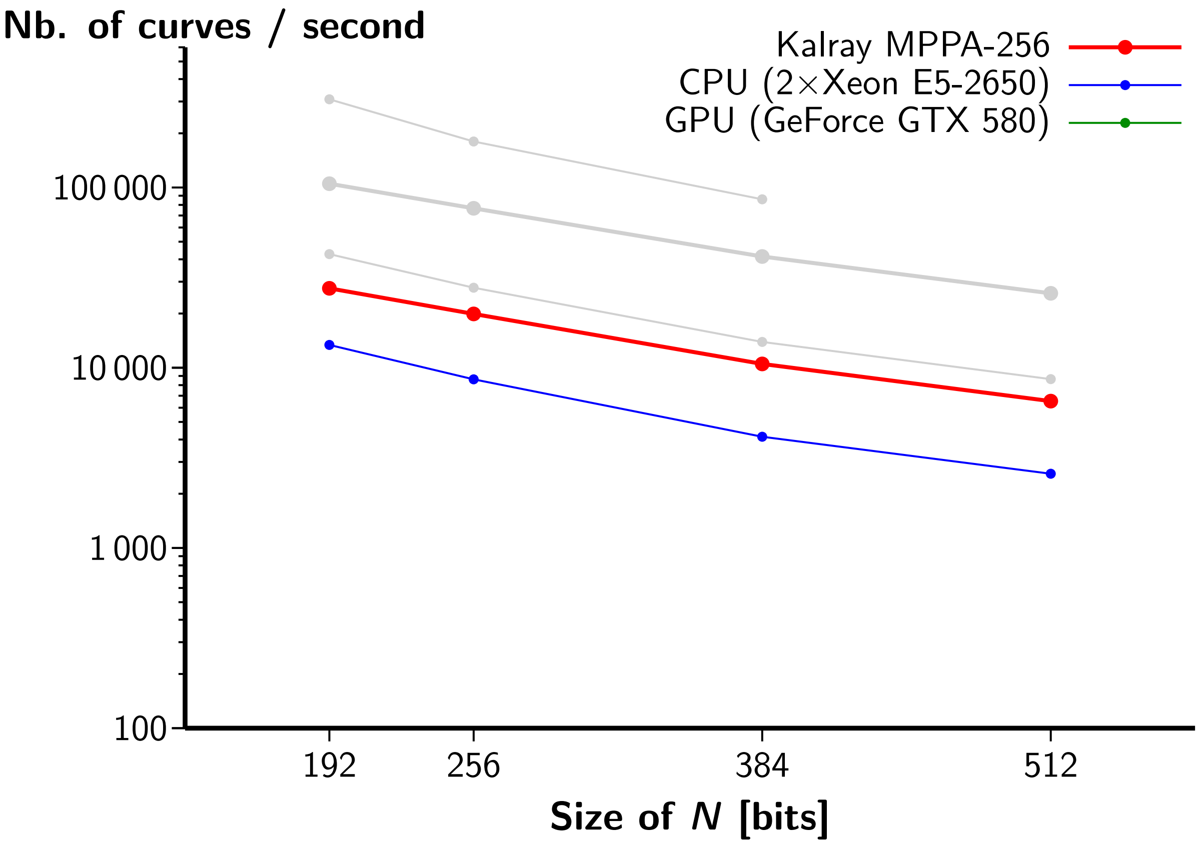
Throughput

- ECM with $B_1 = 256$ and $B_2 = 2^{14}$ (cost: 5 381 modular mults)



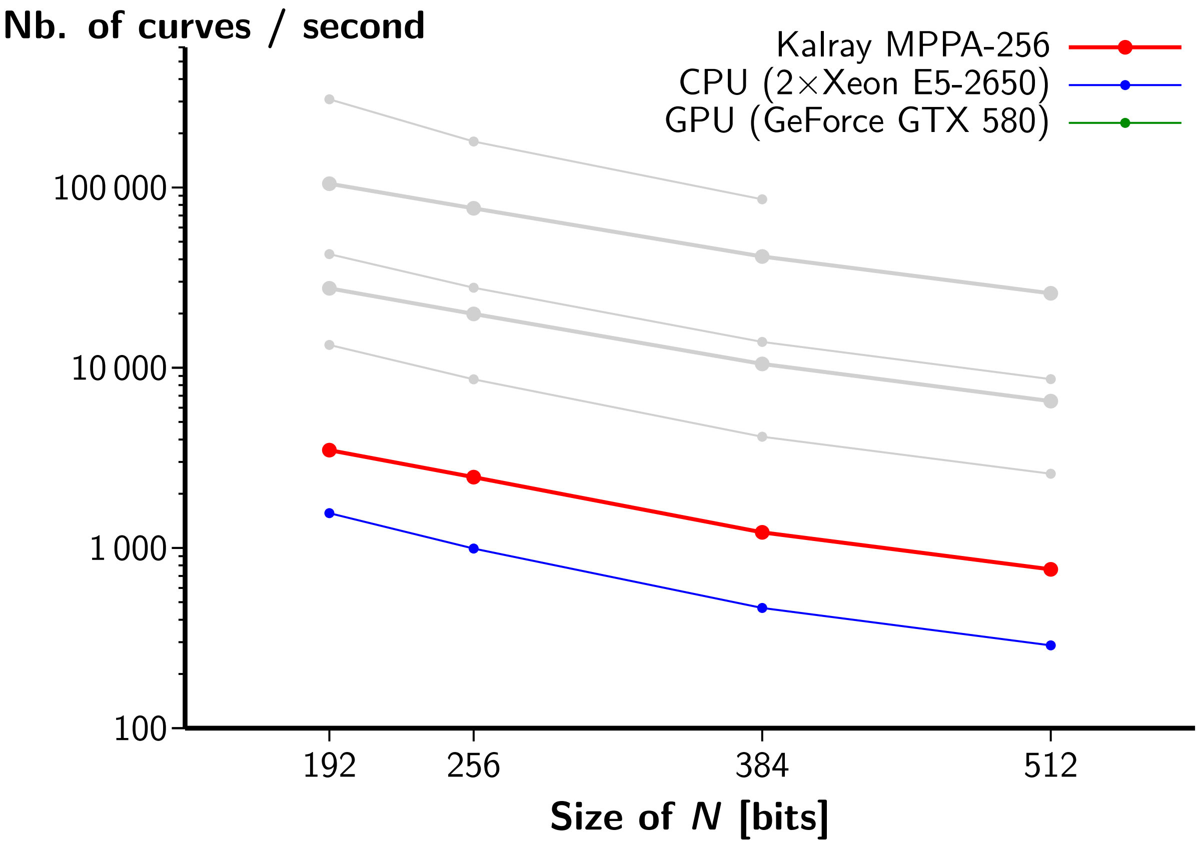
Throughput

- ECM with $B_1 = 1024$ and $B_2 = 7 \cdot 2^{14}$ (cost: 22 878 modular mults)



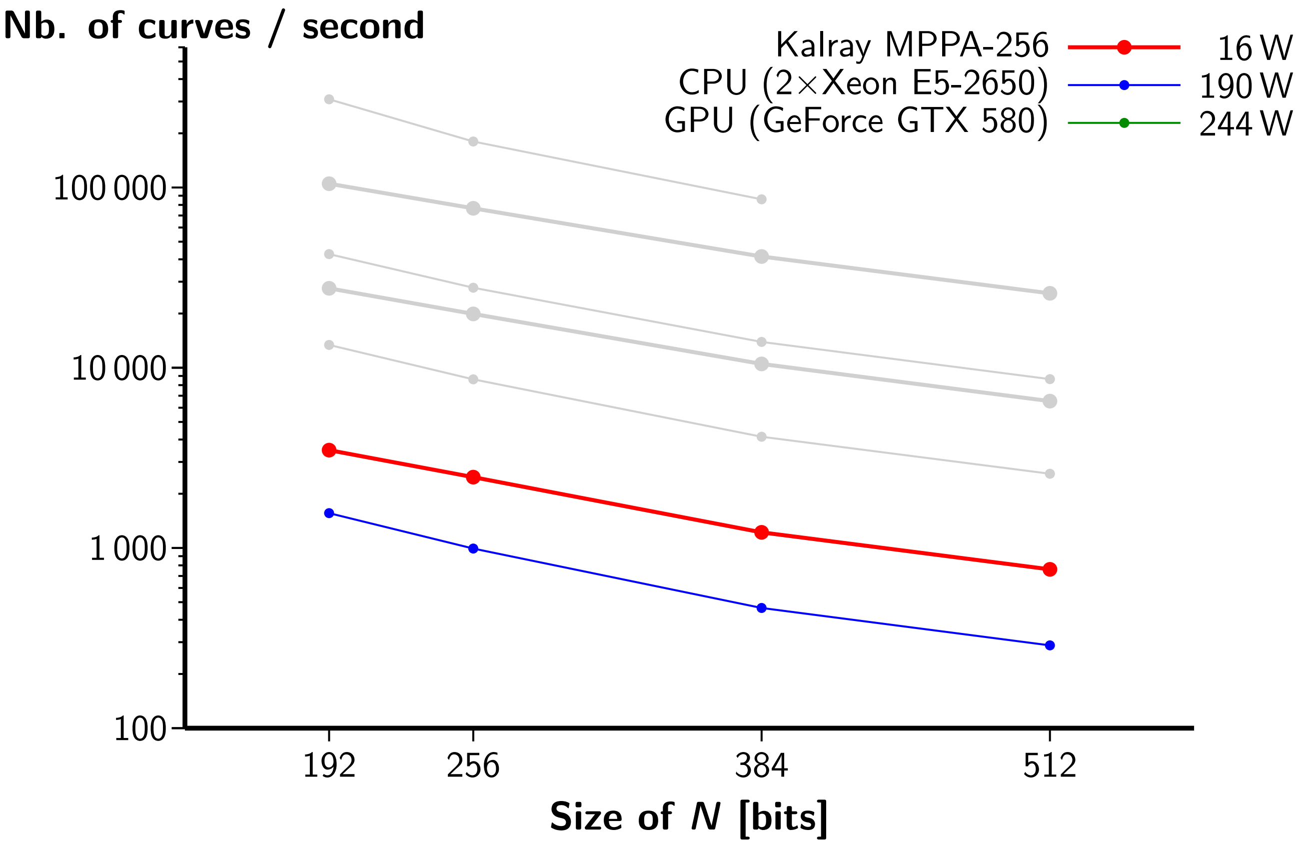
Throughput

- ECM with $B_1 = 8192$ and $B_2 = 80 \cdot 2^{14}$ (cost: 181 852 modular mults)

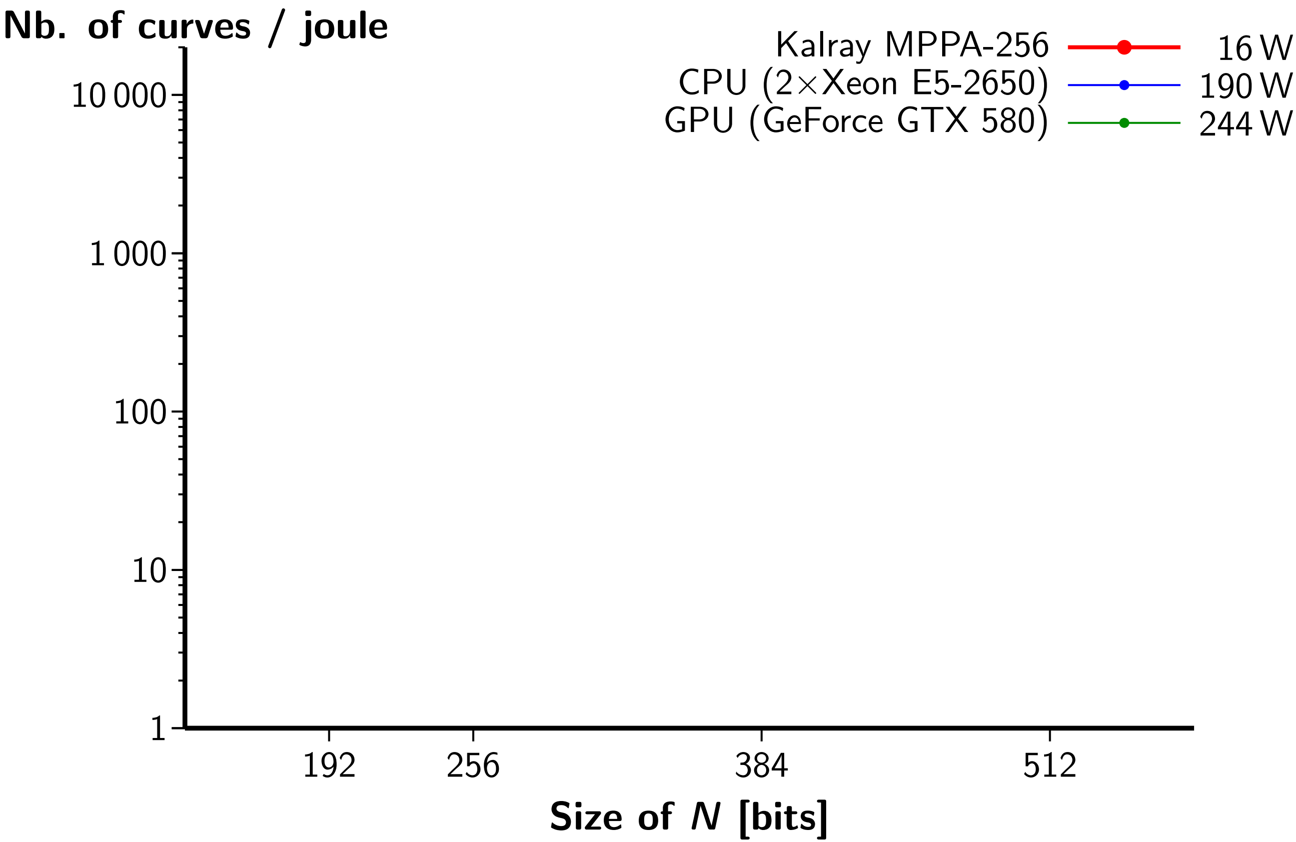


Throughput

- ECM with $B_1 = 8192$ and $B_2 = 80 \cdot 2^{14}$ (cost: 181 852 modular mults)

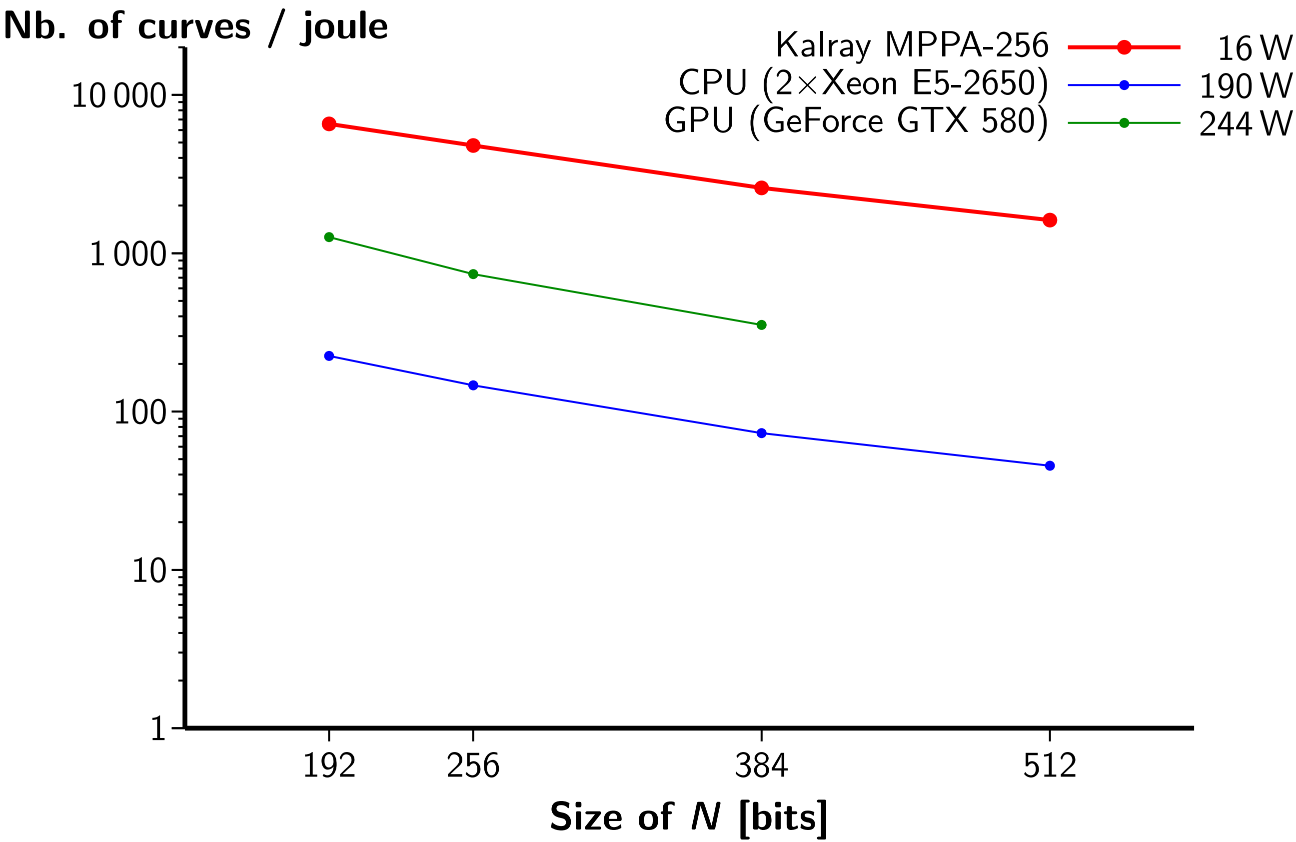


Energy efficiency



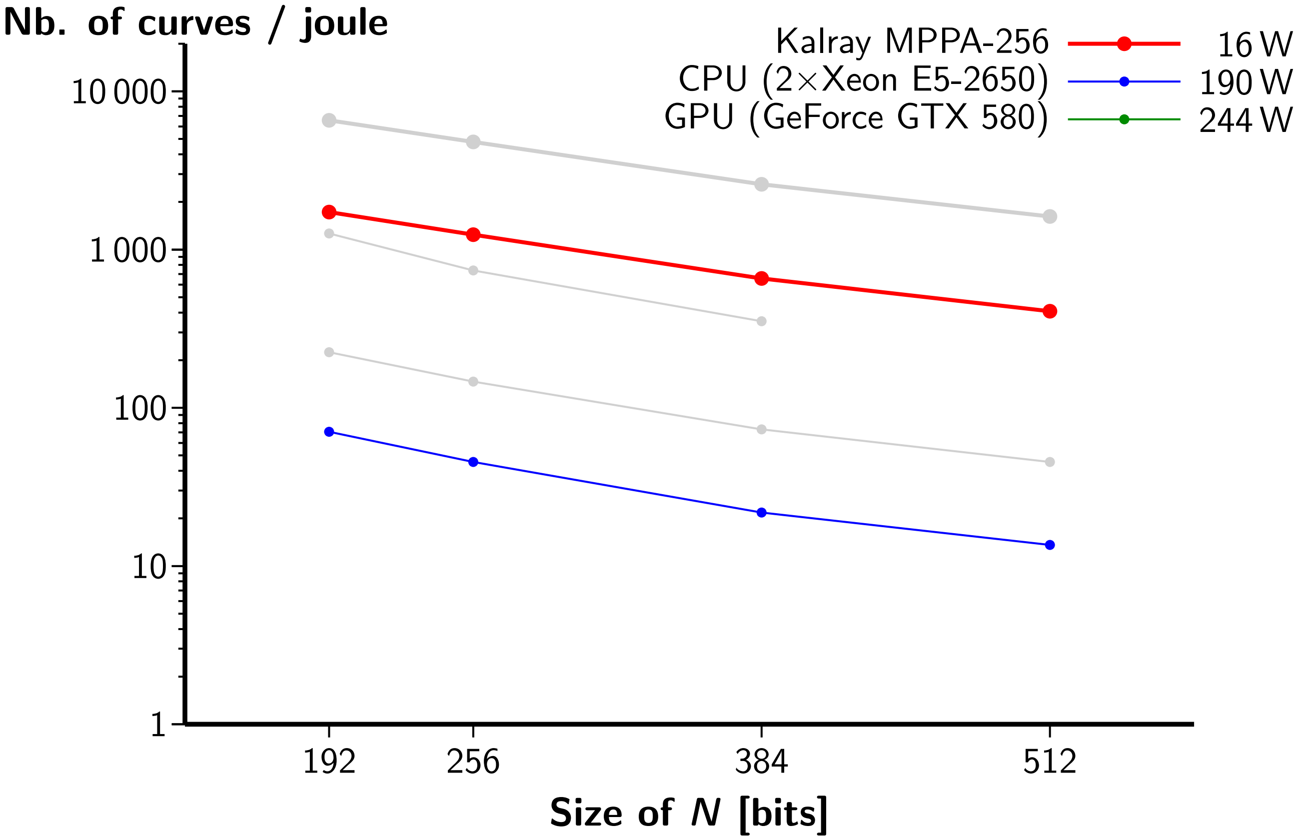
Energy efficiency

- ECM with $B_1 = 256$ and $B_2 = 2^{14}$ (cost: 5 381 modular mults)



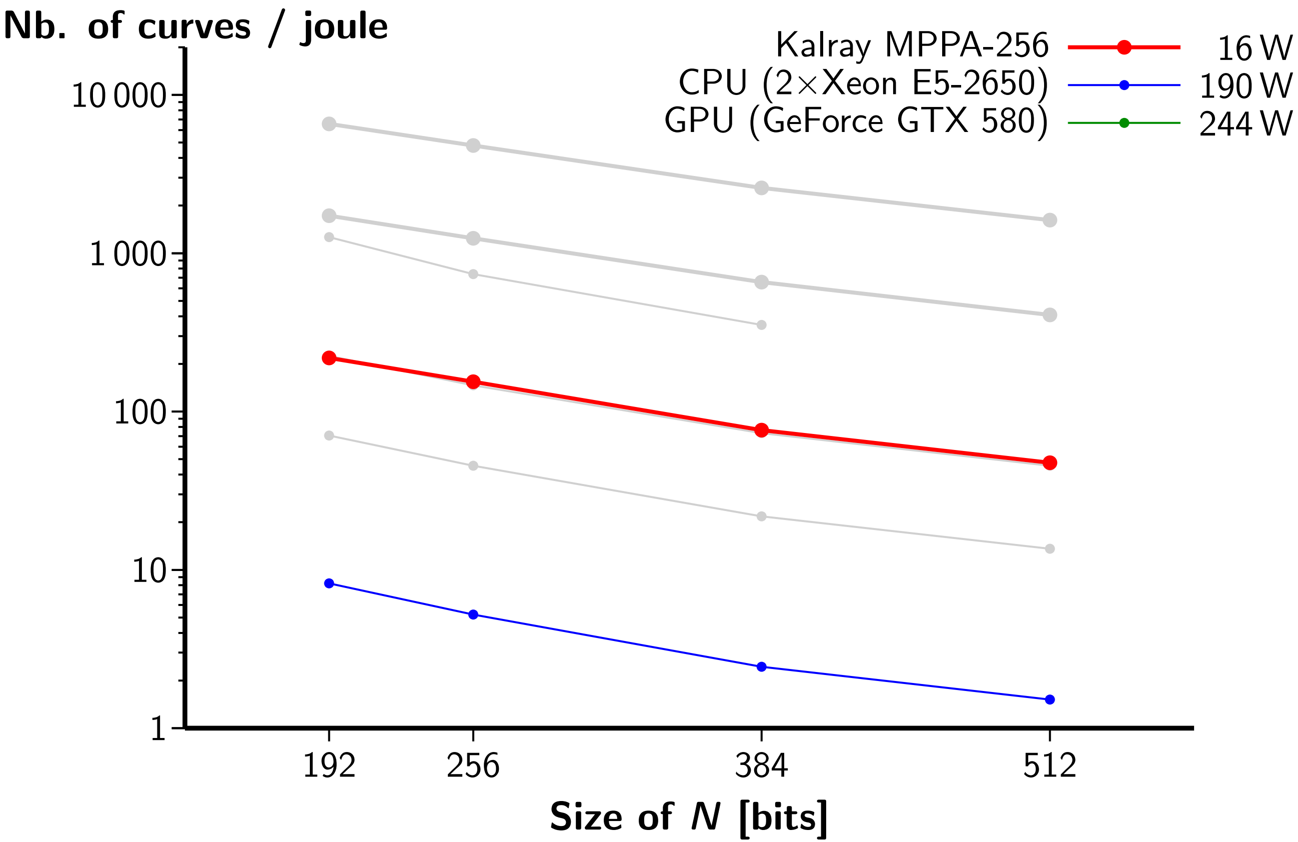
Energy efficiency

- ECM with $B_1 = 1024$ and $B_2 = 7 \cdot 2^{14}$ (cost: 22 878 modular mults)



Energy efficiency

- ECM with $B_1 = 8192$ and $B_2 = 80 \cdot 2^{14}$ (cost: 181 852 modular mults)



Concluding thoughts

- Library for efficient multiprecision modular arithmetic:
  - support for precision up to 512 bits
  - carefully optimized critical low-level functions

Concluding thoughts

- Library for efficient multiprecision modular arithmetic:
  - support for precision up to 512 bits
  - carefully optimized critical low-level functions
  - quasi-optimal cost for quadratic ops: $\approx$ **1 cycle / subproduct**

Concluding thoughts

- Library for efficient multiprecision modular arithmetic:
  - support for precision up to 512 bits
  - carefully optimized critical low-level functions
  - quasi-optimal cost for quadratic ops: $\approx$ **1 cycle / subproduct**
- State-of-the-art implementation of ECM:
  - quite fast: 2–3$\times$ speedup wrt. dual 8-core Intel CPU,
2–3$\times$ slowdown wrt. high-end GPU

Concluding thoughts

- Library for efficient multiprecision modular arithmetic:
  - support for precision up to 512 bits
  - carefully optimized critical low-level functions
  - quasi-optimal cost for quadratic ops: $\approx$ **1 cycle / subproduct**
- State-of-the-art implementation of ECM:
  - quite fast: 2–3$\times$ speedup wrt. dual 8-core Intel CPU,
    2–3$\times$ slowdown wrt. high-end GPU
  - energy efficient: $\sim$ 30$\times$ better than CPU, $\sim$ 5$\times$ better than GPU

Concluding thoughts

- Library for efficient multiprecision modular arithmetic:
  - support for precision up to 512 bits
  - carefully optimized critical low-level functions
  - quasi-optimal cost for quadratic ops: **≈ 1 cycle / subproduct**
- State-of-the-art implementation of ECM:
  - quite fast: 2–3× speedup wrt. dual 8-core Intel CPU,
    2–3× slowdown wrt. high-end GPU
  - energy efficient: ∼ 30× better than CPU, ∼ 5× better than GPU
  - more on-chip memory: can tackle larger sizes than GPUs

Thank you for your attention

Questions?

More info:

- Git repo: https://gforge.inria.fr/projects/kalray-ecm
- Paper: https://eprint.iacr.org/2016/365