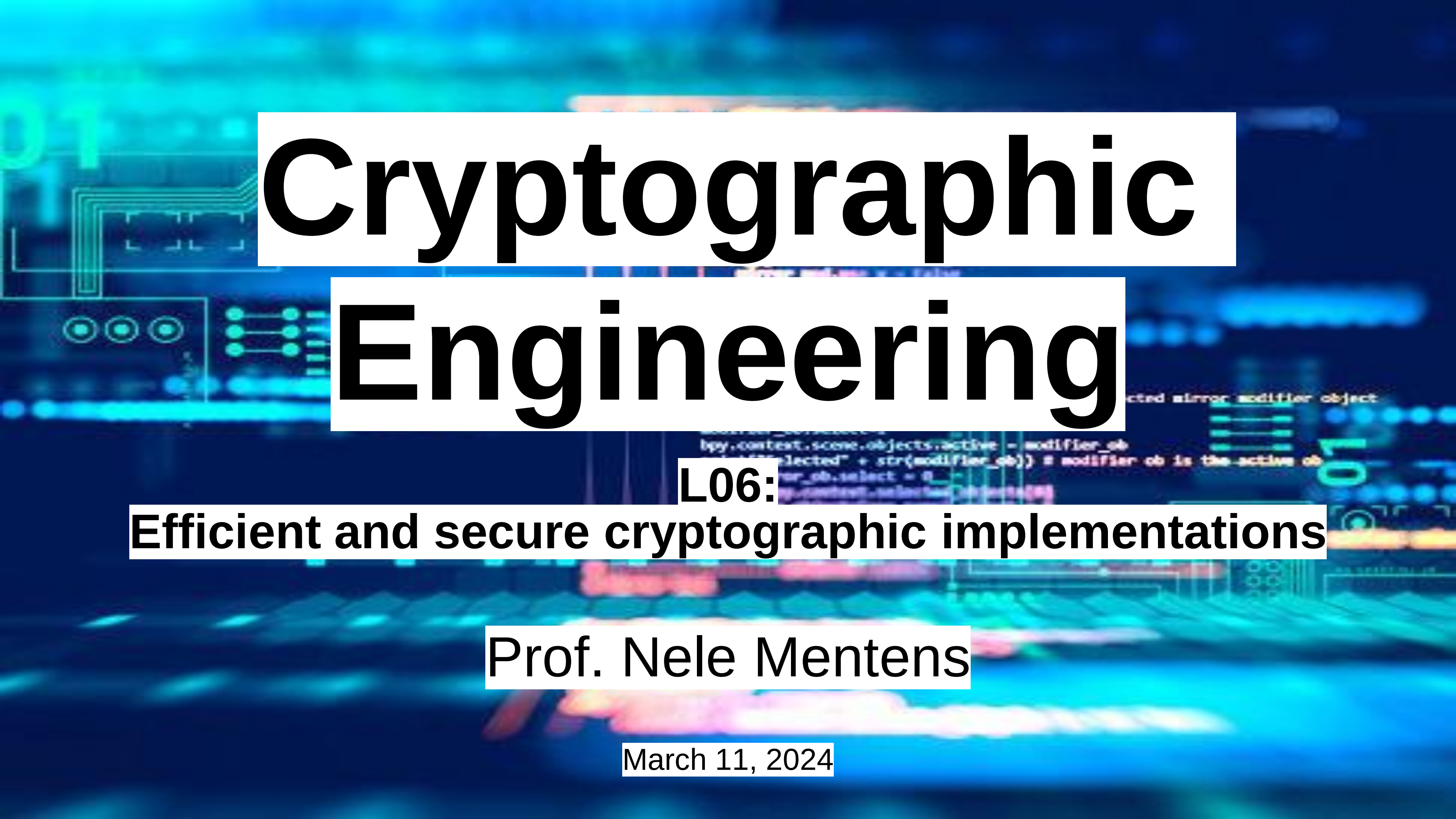


Cryptographic Engineering

L06:
Efficient and secure cryptographic implementations

Prof. Nele Mentens

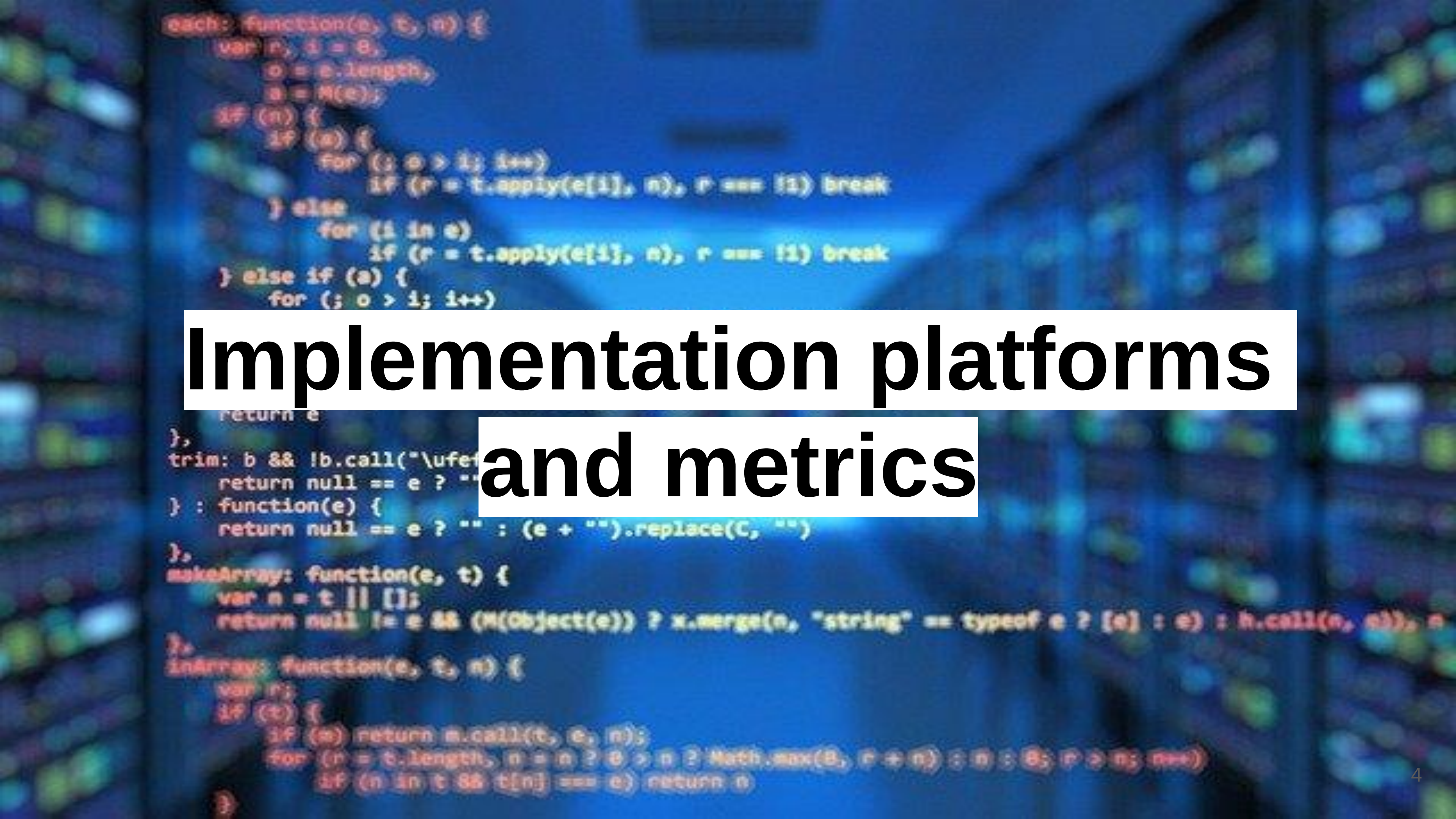
March 11, 2024

Outline

- Implementation platforms and metrics
- Traditional public-key crypto algorithms:
  - RSA
  - Elliptic Curve Cryptography (ECC)
- Post-quantum crypto algorithms

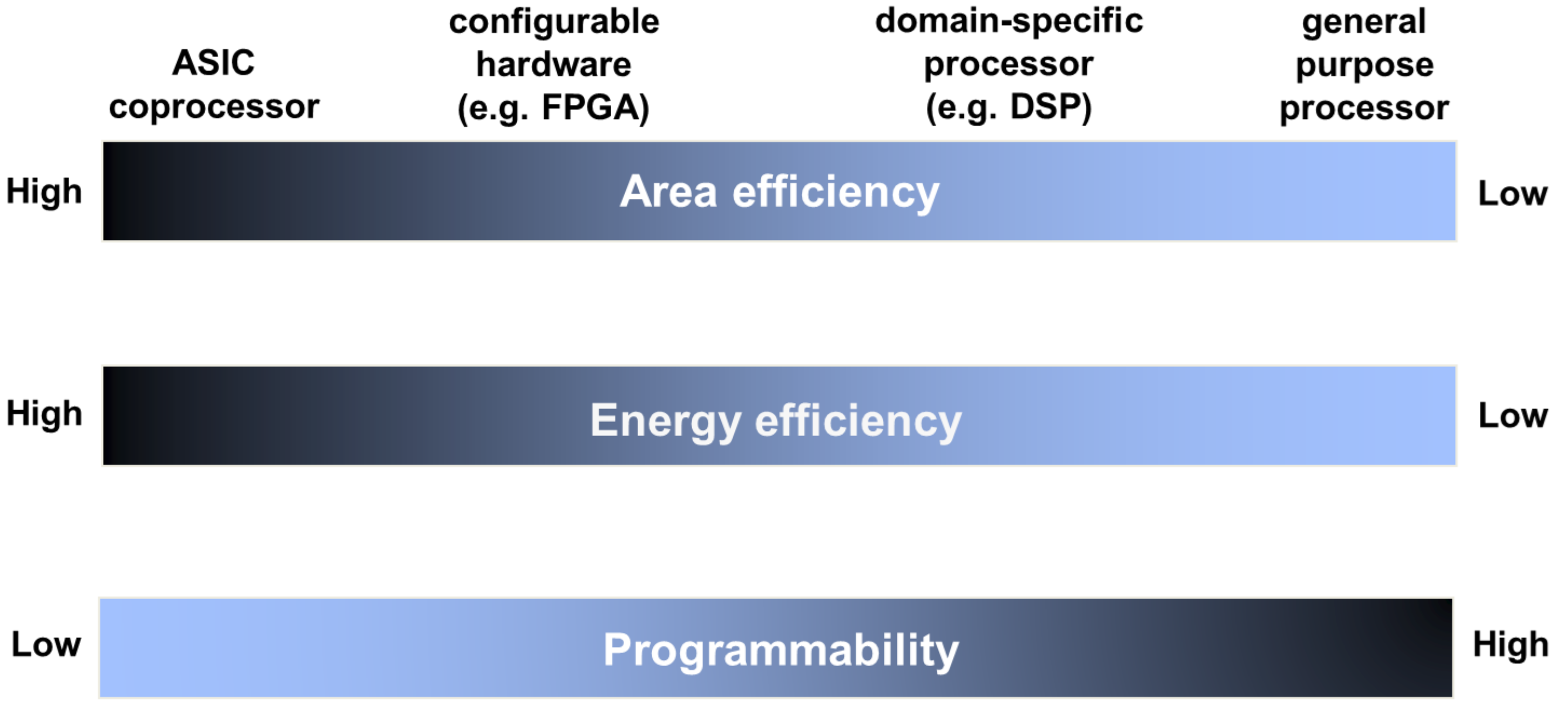
Outline

- **Implementation platforms and metrics**
- Traditional public-key crypto algorithms:
  - RSA
  - Elliptic Curve Cryptography (ECC)
- Post-quantum crypto algorithms



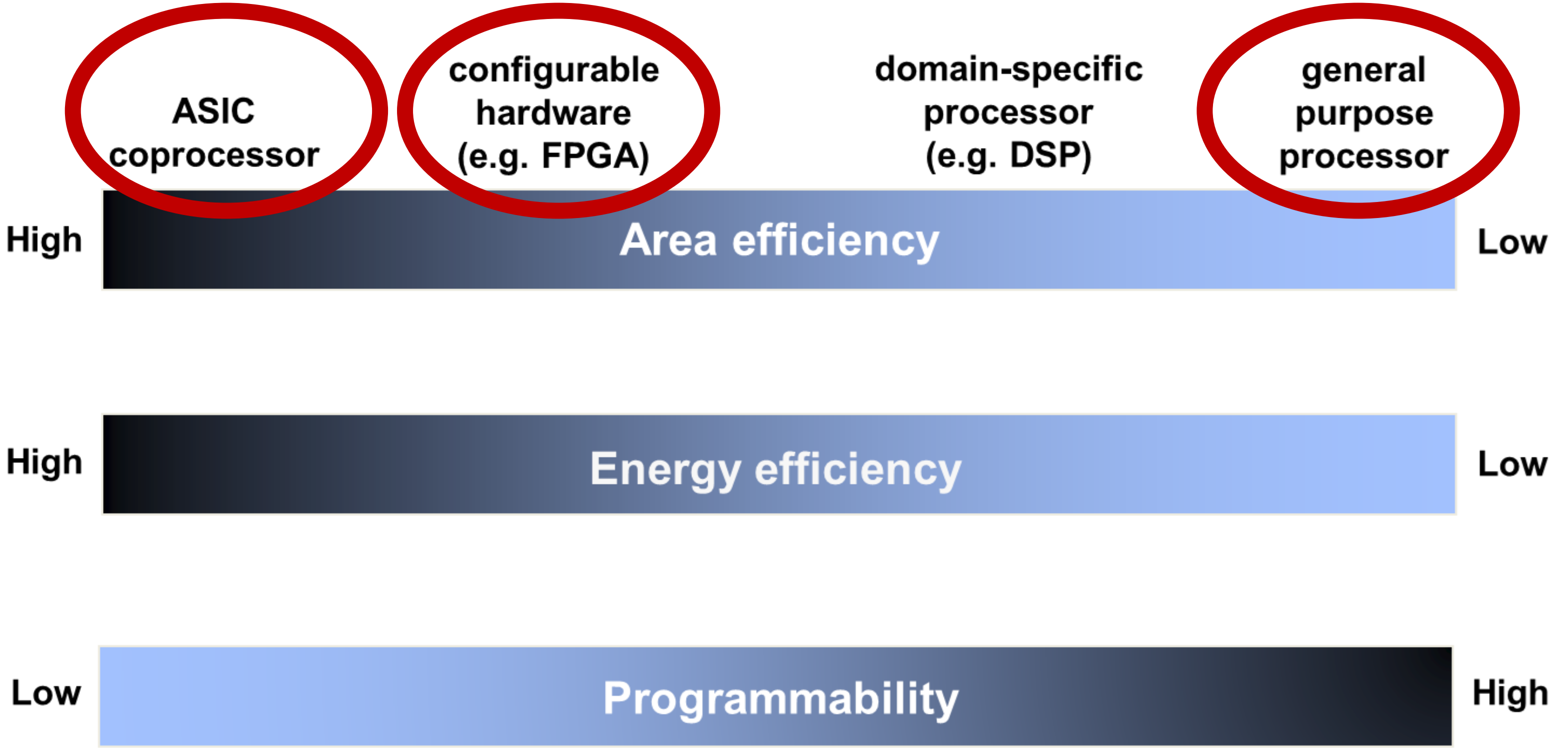
Implementation platforms and metrics

Implementation platforms

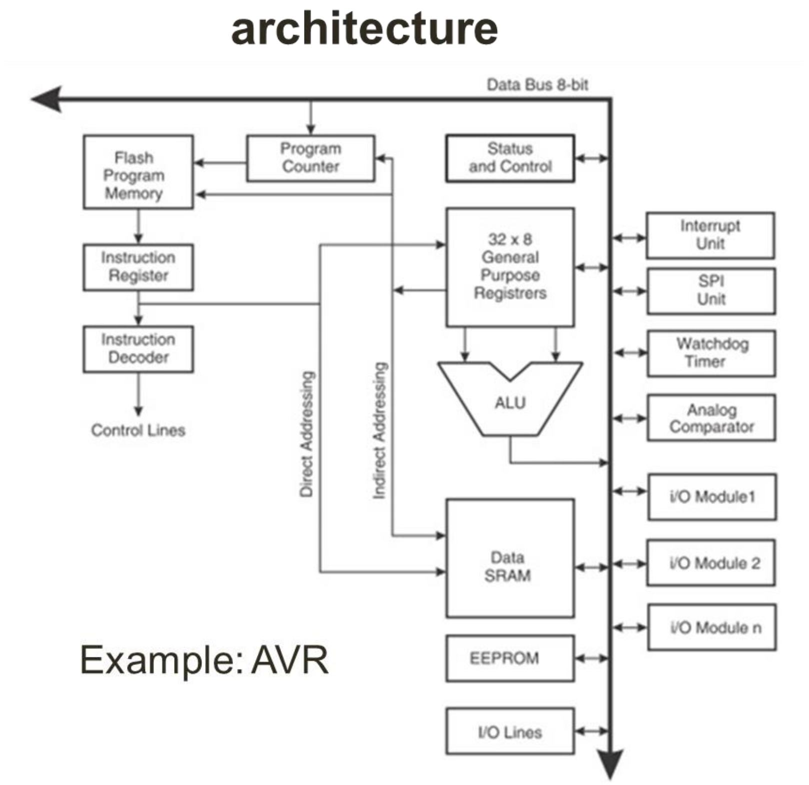

ASIC coprocessor
configurable hardware (e.g. FPGA)
domain-specific processor (e.g. DSP)
general purpose processor
High
Area efficiency
Low
High
Energy efficiency
Low
Low
Programmability
High

Platforms covered in this course

Implementation platforms

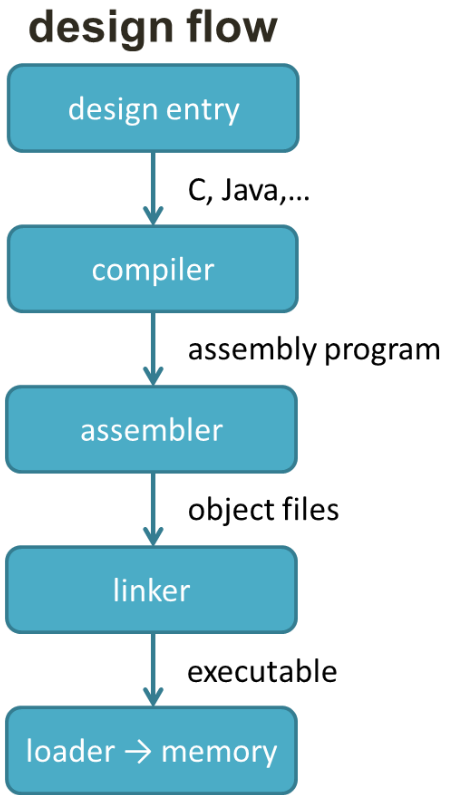

ASIC coprocessor
configurable hardware (e.g. FPGA)
domain-specific processor (e.g. DSP)
general purpose processor
High
Area efficiency
Low
High
Energy efficiency
Low
Low
Programmability
High

General-purpose microprocessor



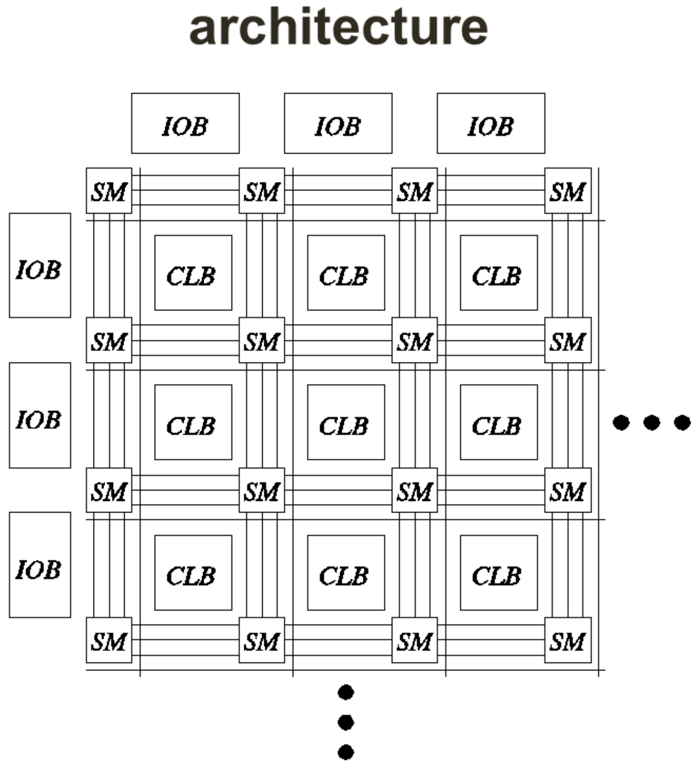
- The CPU is the heart of a microprocessor and contains a.o.:
  - ALU (Arithmetic Logic Unit)
  - Register file
  - Program memory (Flash)
  - Data memory (SRAM)

General-purpose microprocessor



- The hardware architecture of a microprocessor is fixed
- The code describes what should be executed on the fixed hardware
- The instructions end up in the program memory

FPGA = Field-Programmable Gate Array

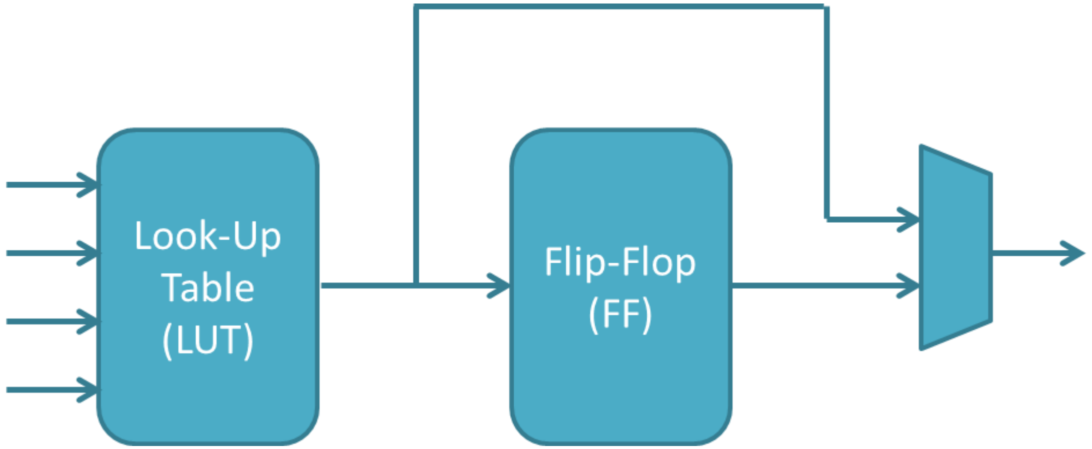


Basic components (Xilinx terminology):

- CLB = Configurable Logic Block
    - CLBs consist of slices
    - Slices consist of
        - Look-Up Tables (LUTs)
        - Multiplexers
        - Flip-Flops (FFs)
        - Carry logic
- SM = Switch Matrix
- IOB = Input/Output Block

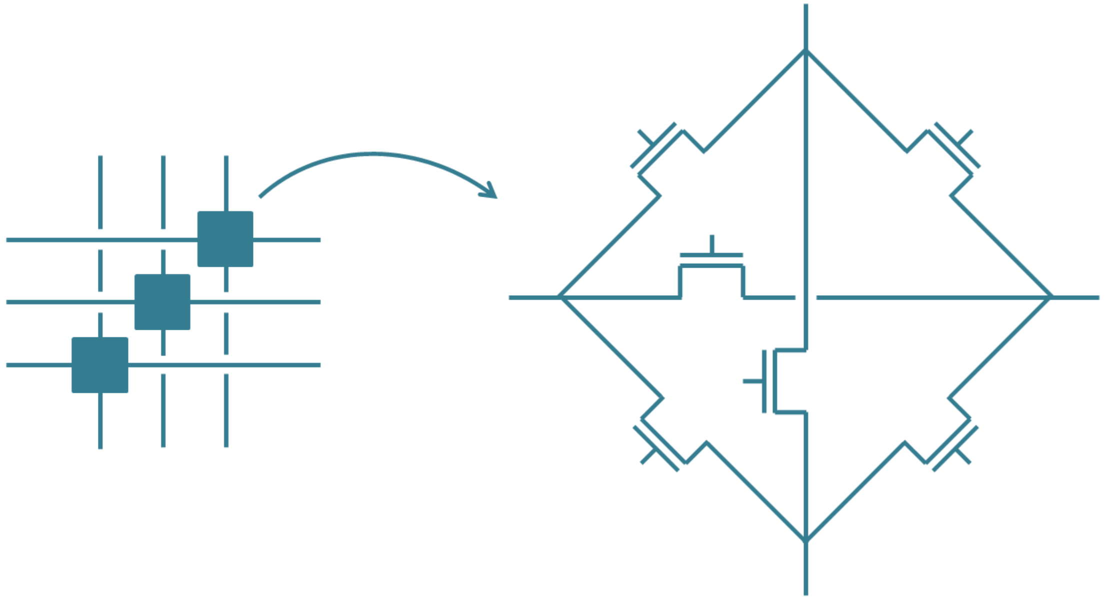
FPGA = Field-Programmable Gate Array

Basic content of a slice (excluding carry logic):

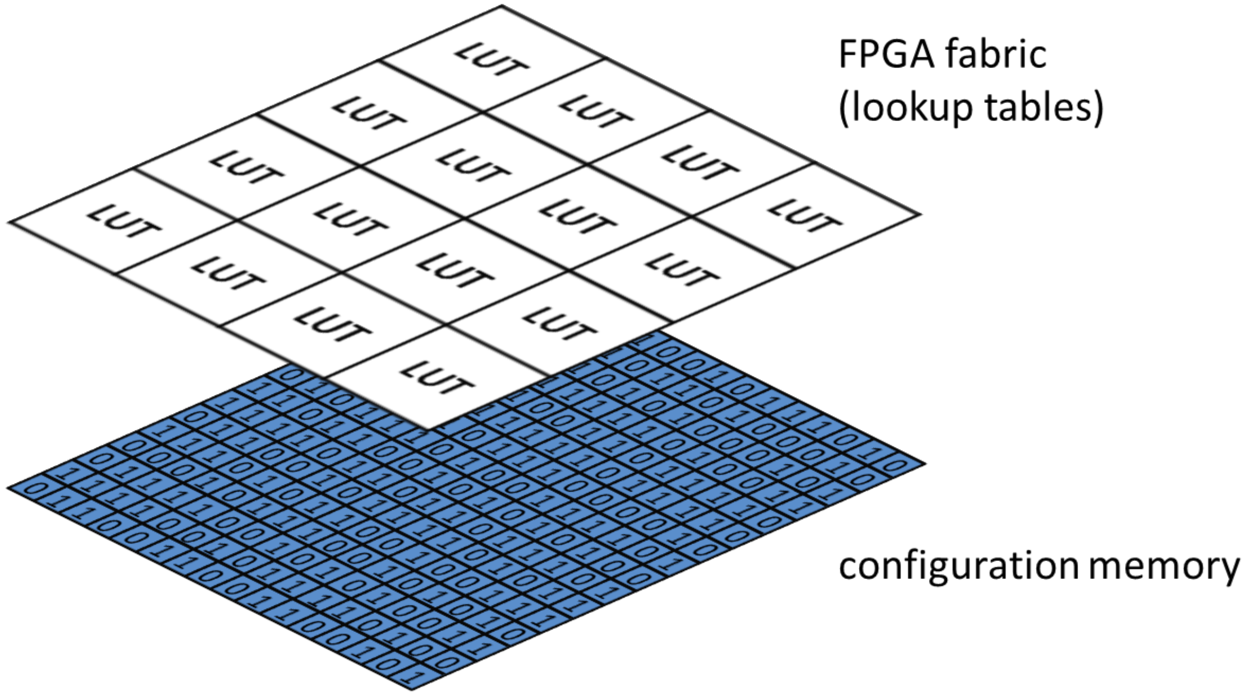


FPGA = Field-Programmable Gate Array

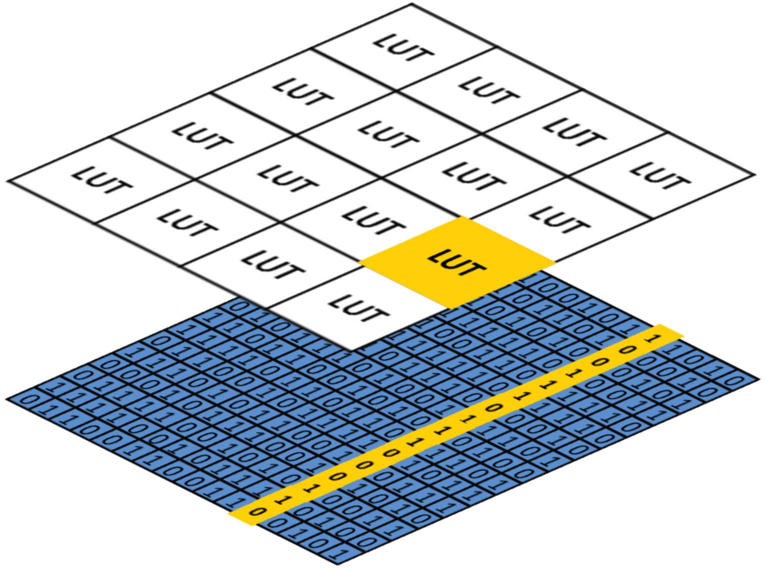
Basic principle of a switch matrix:

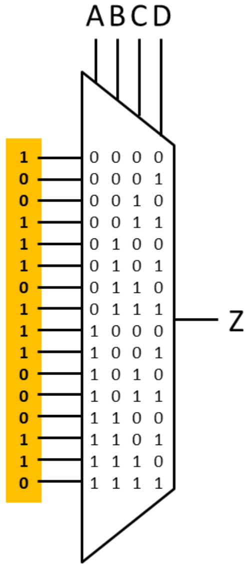


FPGA = Field-Programmable Gate Array



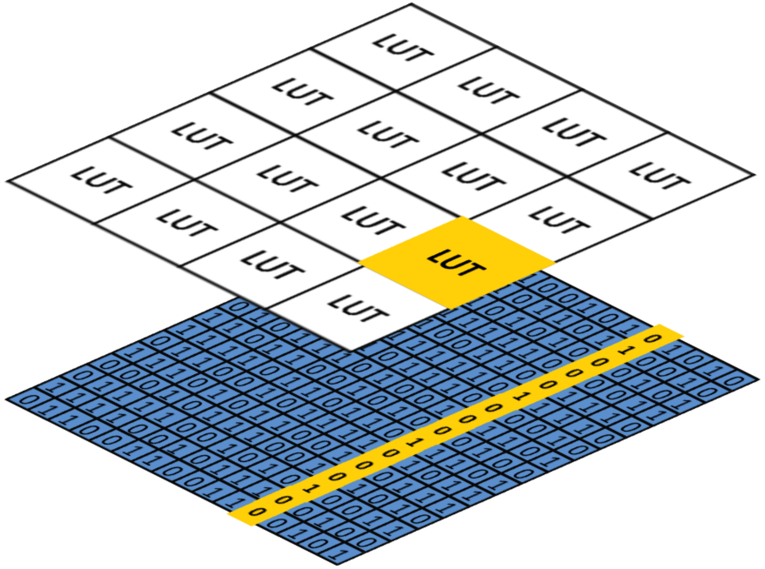
FPGA = Field-Programmable Gate Array

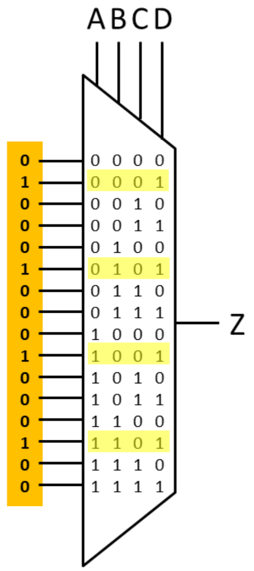


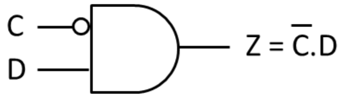


	A	B	C	D
1	0	0	0	0
0	0	0	0	1
0	0	0	1	0
1	0	0	1	1
1	0	1	0	0
1	0	1	0	1
0	0	1	1	0
1	0	1	1	1
1	1	0	0	0
1	1	0	0	1
0	1	0	1	0
0	1	0	1	1
0	1	1	0	0
1	1	1	0	1
1	1	1	1	0
0	1	1	1	1

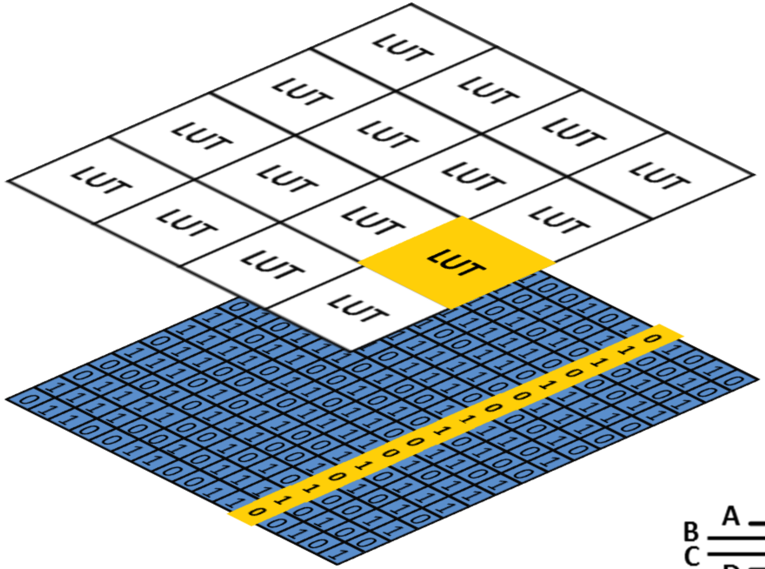
FPGA = Field-Programmable Gate Array

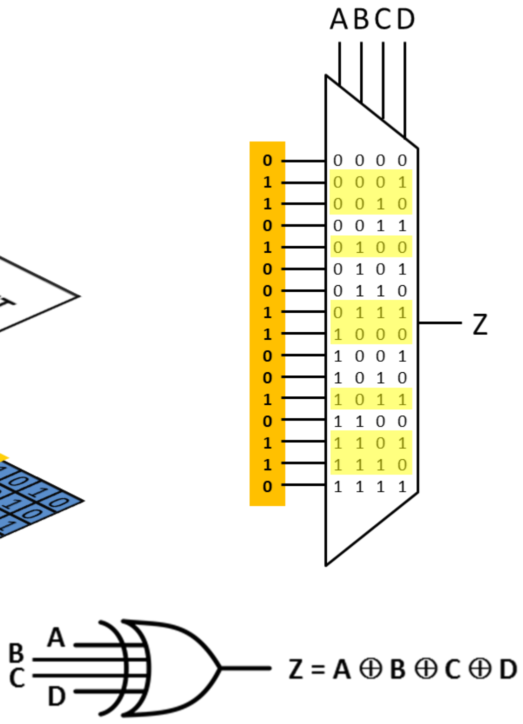






FPGA = Field-Programmable Gate Array

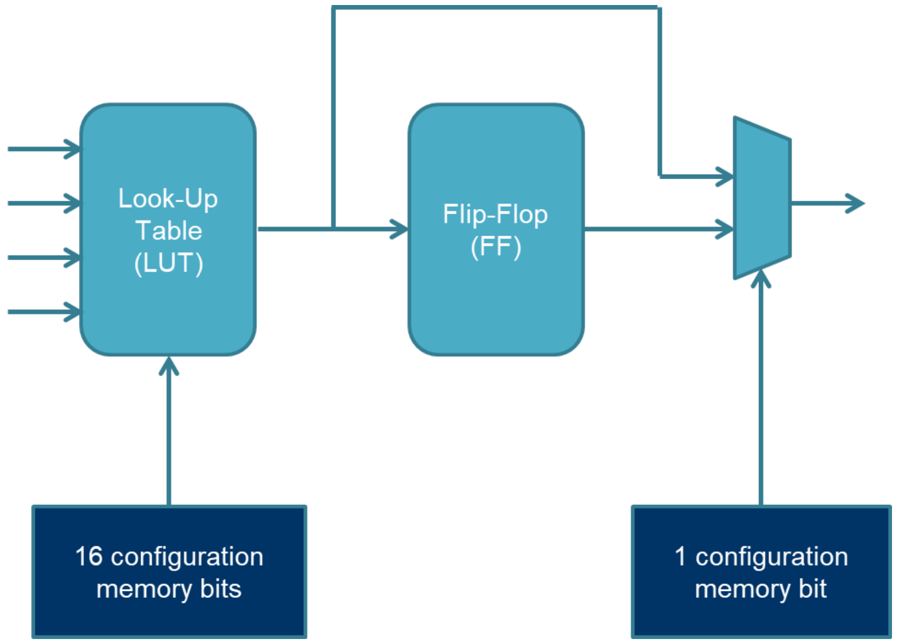




	A	B	C	D
0	0	0	0	0
1	0	0	0	1
1	0	0	1	0
0	0	0	1	1
1	0	1	0	0
0	0	1	0	1
0	0	1	1	0
1	0	1	1	1
1	1	0	0	0
0	1	0	0	1
0	1	0	1	0
1	1	0	1	1
0	1	1	0	0
1	1	1	0	1
1	1	1	1	0
0	1	1	1	1

FPGA = Field-Programmable Gate Array

Basic content of a slice (excluding carry logic) + configuration bits:



FPGA = Field-Programmable Gate Array

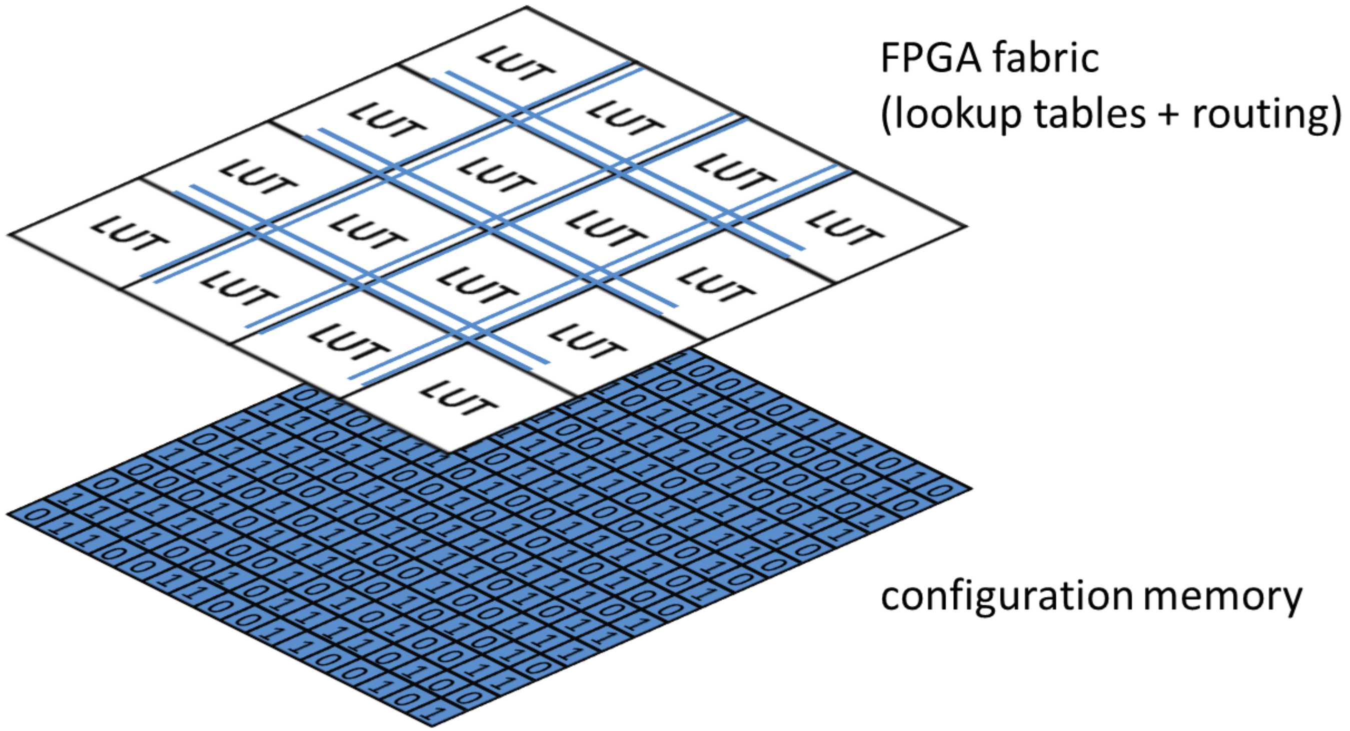
A	B	C	D	Z_0	Z_1	Z_2	Z_3	…	Z_{65280}	…	Z_{65535}
0	0	0	0	0	1	0	1		0		1
0	0	0	1	0	0	1	1		0		1
0	0	1	0	0	0	0	0		0		1
0	0	1	1	0	0	0	0		0		1
0	1	0	0	0	0	0	0		0		1
0	1	0	1	0	0	0	0		0		1
0	1	1	0	0	0	0	0		0		1
0	1	1	1	0	0	0	0		0		1
1	0	0	0	0	0	0	0		1		1
1	0	0	1	0	0	0	0		1		1
1	0	1	0	0	0	0	0		1		1
1	0	1	1	0	0	0	0		1		1
1	1	0	0	0	0	0	0		1		1
1	1	0	1	0	0	0	0		1		1
1	1	1	0	0	0	0	0		1		1
1	1	1	1	0	0	0	0		1		1

Why 16 configuration bits for a 4-to-1 LUT?

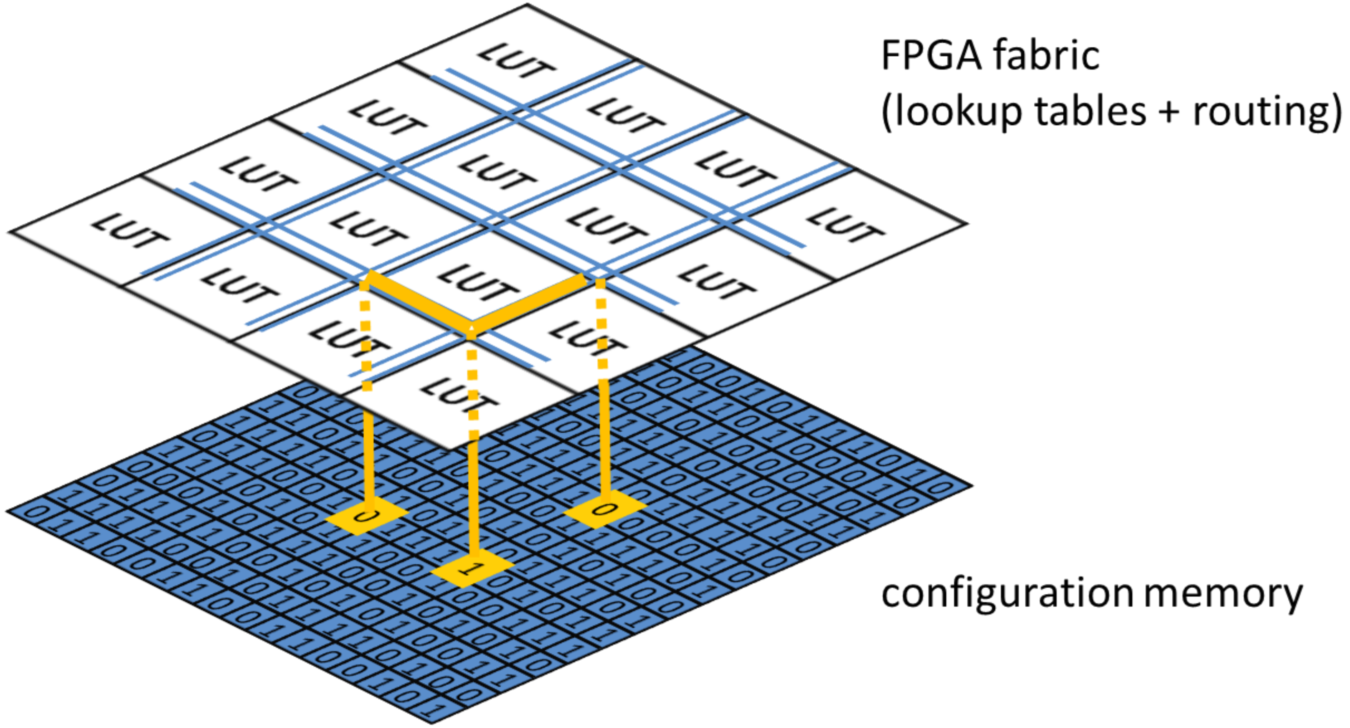
- 2^{16} possible output functions:
- $Z_0 = 0$
- $Z_1 = A'.B'.C'.D'$
- $Z_2 = A'.B'.C'.D$
- $Z_3 = A'.B'.C'$
- …
- $Z_{65280} = A$
- …
- $Z_{65535} = 1$

Note: modern FPGAs have 6-to-2 LUTs (which can be configured as one 6-to-1 LUT or two 5-to-1 LUTs with shared inputs)

FPGA = Field-Programmable Gate Array

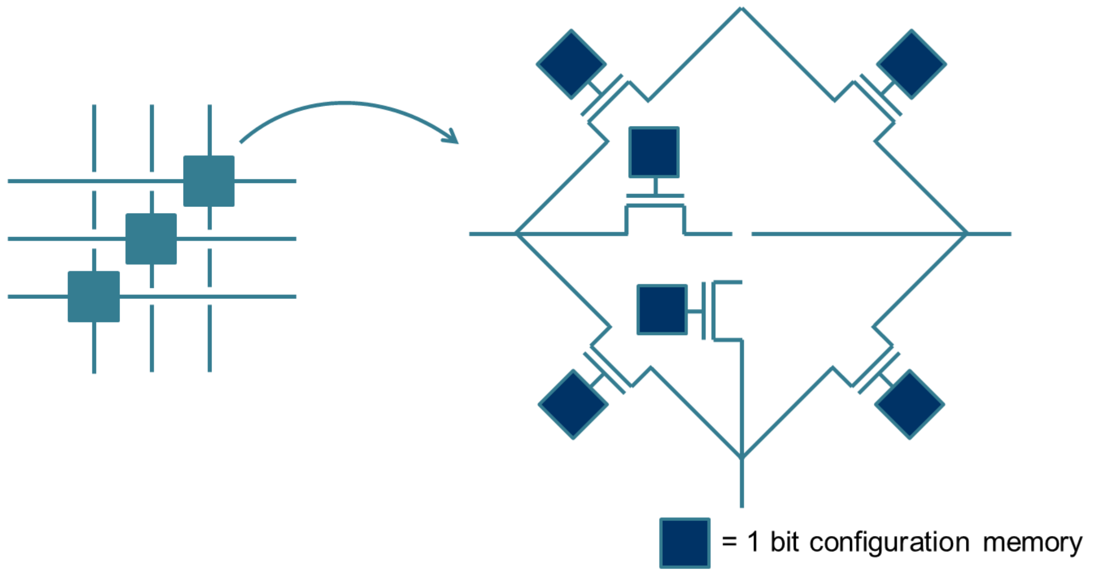


FPGA = Field-Programmable Gate Array

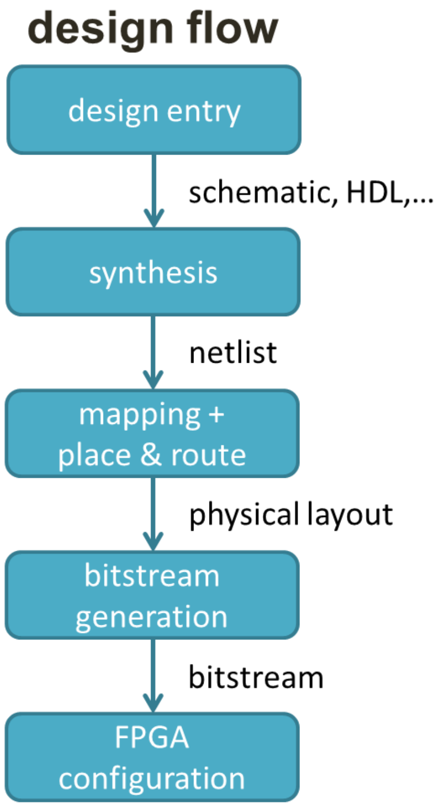


FPGA = Field-Programmable Gate Array

Basic principle of a switch matrix + configuration bits:



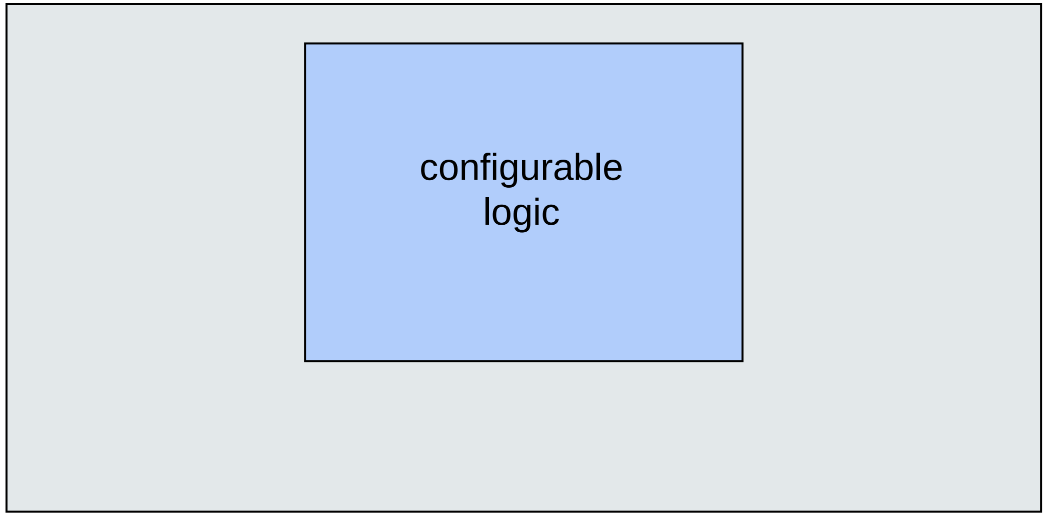
FPGA = Field-Programmable Gate Array



- The hardware architecture of an FPGA is configurable
- The code describes the hardware that we need
- The bitstream ends up in the configuration memory

FPGA = Field-Programmable Gate Array

1991: XC4000



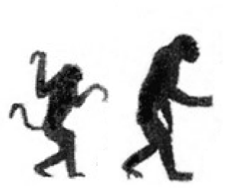
technology node:

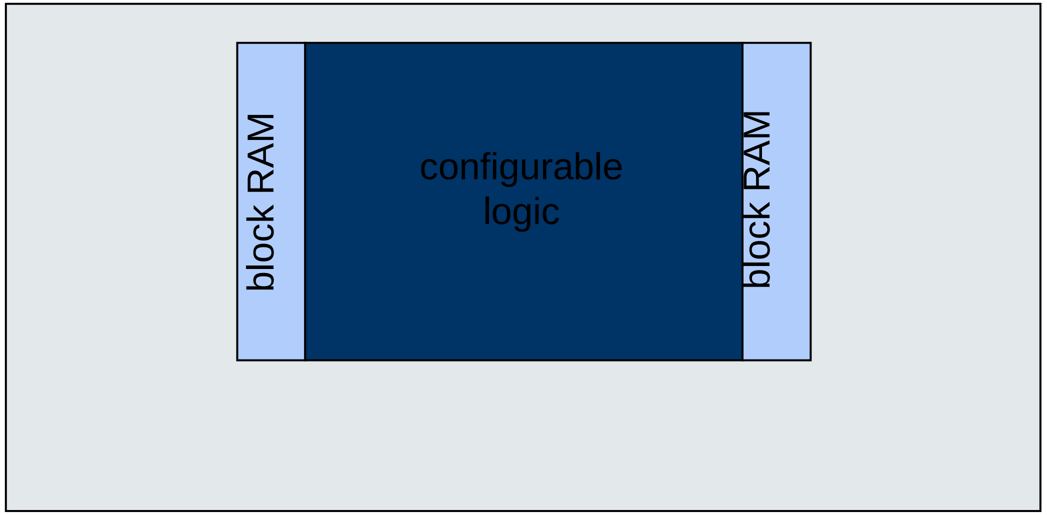
Evolution of Xilinx FPGAs
(selected devices + features)

FPGA = Field-Programmable Gate Array

1991: XC4000

1998: Virtex





technology node:

220 nm

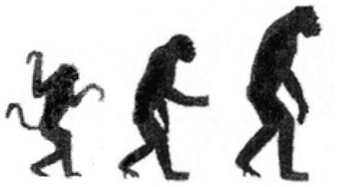
Evolution of Xilinx FPGAs
(selected devices + features)

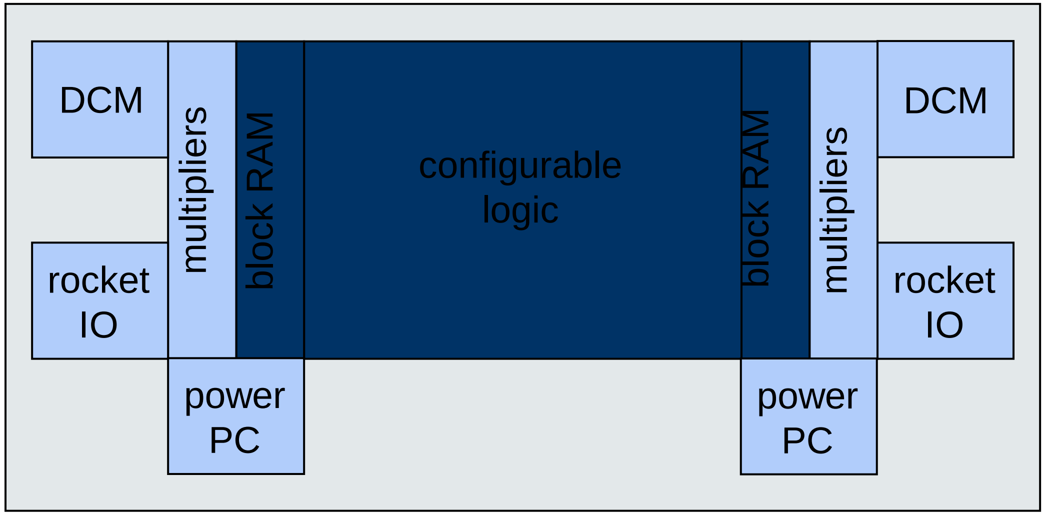
FPGA = Field-Programmable Gate Array

1991: XC4000

1998: Virtex

2002: Virtex-II Pro



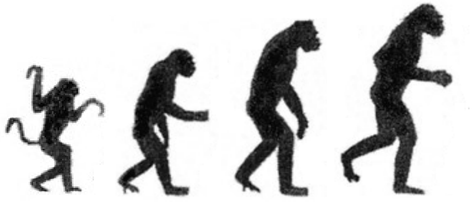


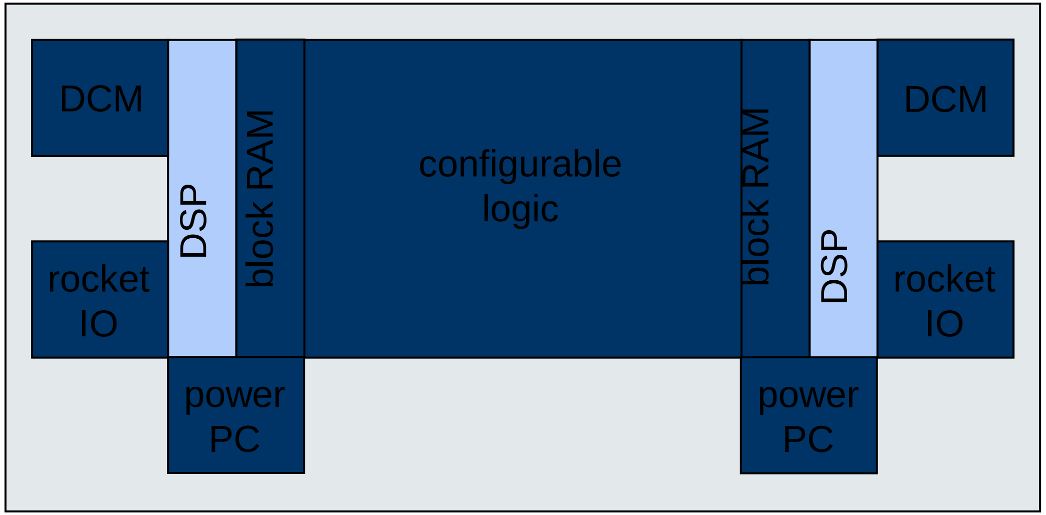
technology node:

Evolution of Xilinx FPGAs
(selected devices + features)

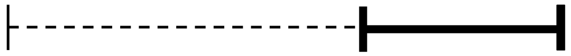
FPGA = Field-Programmable Gate Array

1991: XC4000
1998: Virtex
2002: Virtex-II Pro
2004: Virtex-4





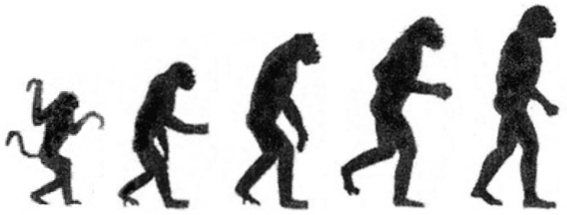
technology node:



90 nm

Evolution of Xilinx FPGAs
(selected devices + features)

FPGA = Field-Programmable Gate Array



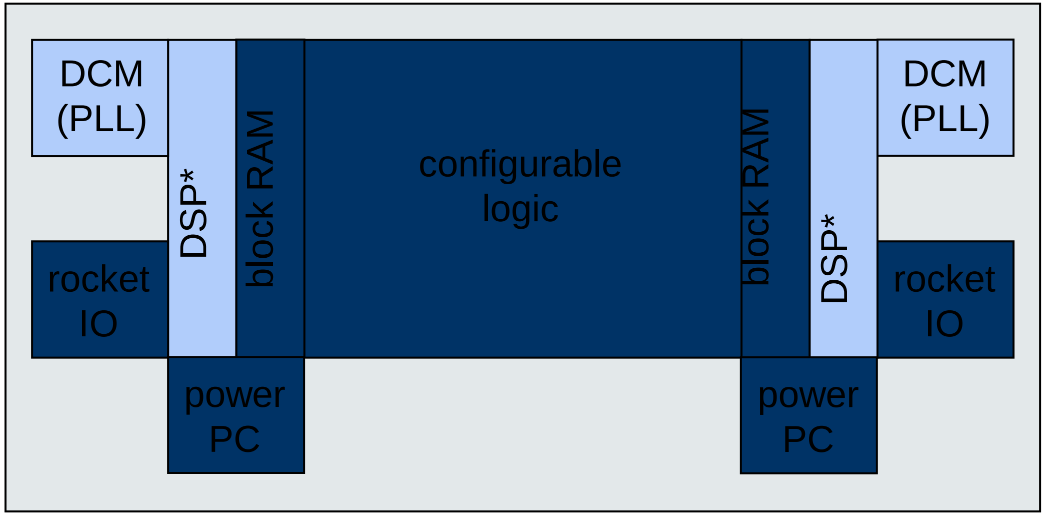
1991: XC4000

1998: Virtex

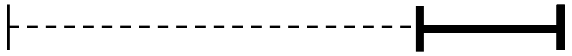
2002: Virtex-II Pro

2004: Virtex-4

2006: Virtex-5

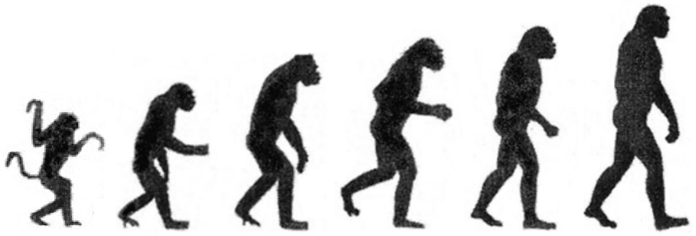


technology node:

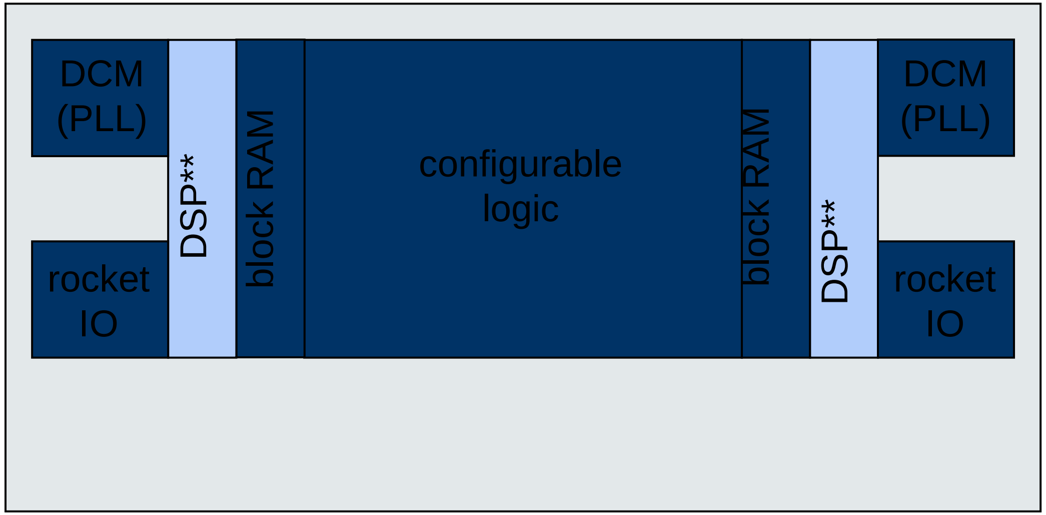


Evolution of Xilinx FPGAs
(selected devices + features)

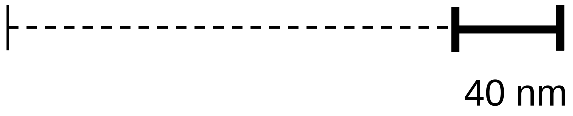
FPGA = Field-Programmable Gate Array



1991: XC4000
1998: Virtex
2002: Virtex-II Pro
2004: Virtex-4
2006: Virtex-5
2009: Virtex-6

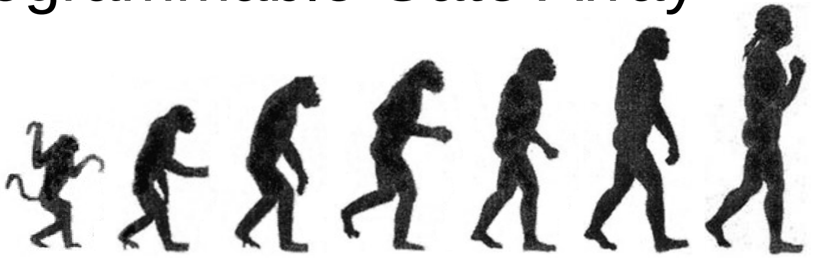


technology node:



Evolution of Xilinx FPGAs
(selected devices + features)

FPGA = Field-Programmable Gate Array



1991: XC4000

1998: Virtex

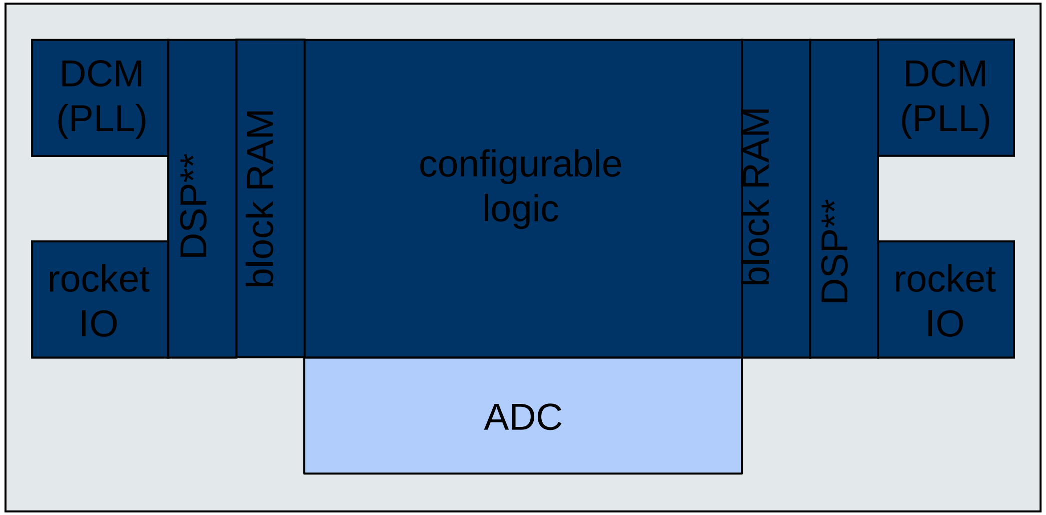
2002: Virtex-II Pro

2004: Virtex-4

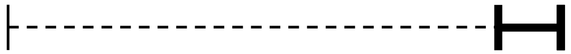
2006: Virtex-5

2009: Virtex-6

2010: 7 Series

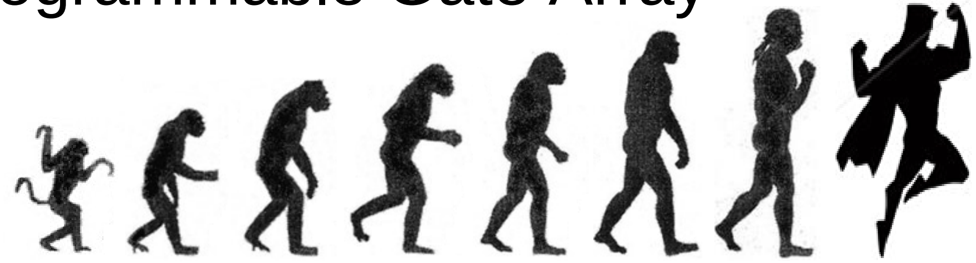


technology node:

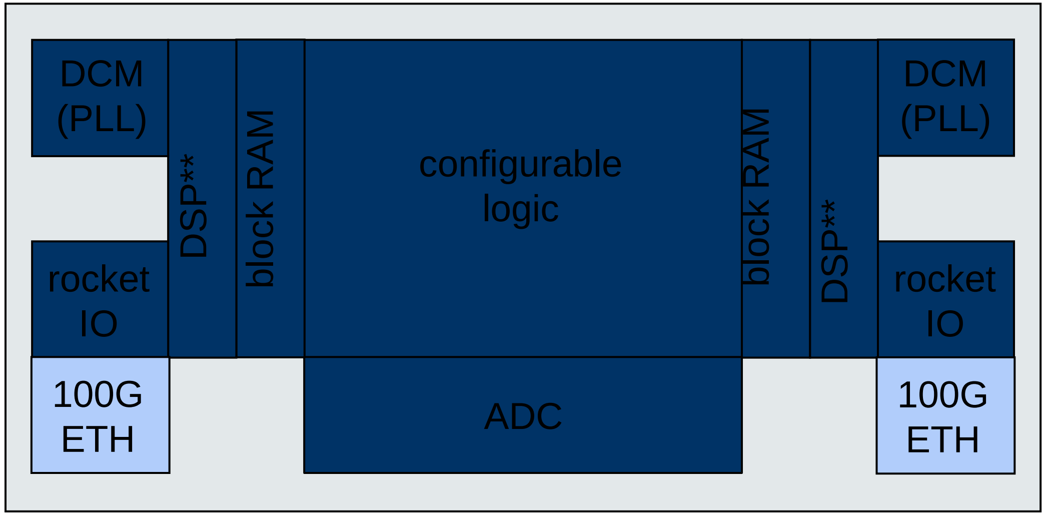


FPGA = Field-Programmable Gate Array

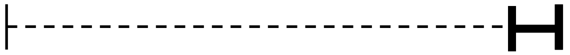
Evolution of Xilinx FPGAs
(selected devices + features)



1991: XC4000
1998: Virtex
2002: Virtex-II Pro
2004: Virtex-4
2006: Virtex-5
2009: Virtex-6
2010: 7 Series
2013: UltraScale

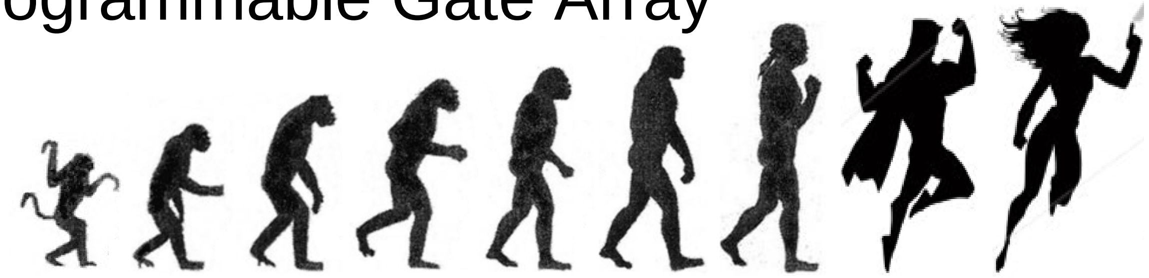


technology node:

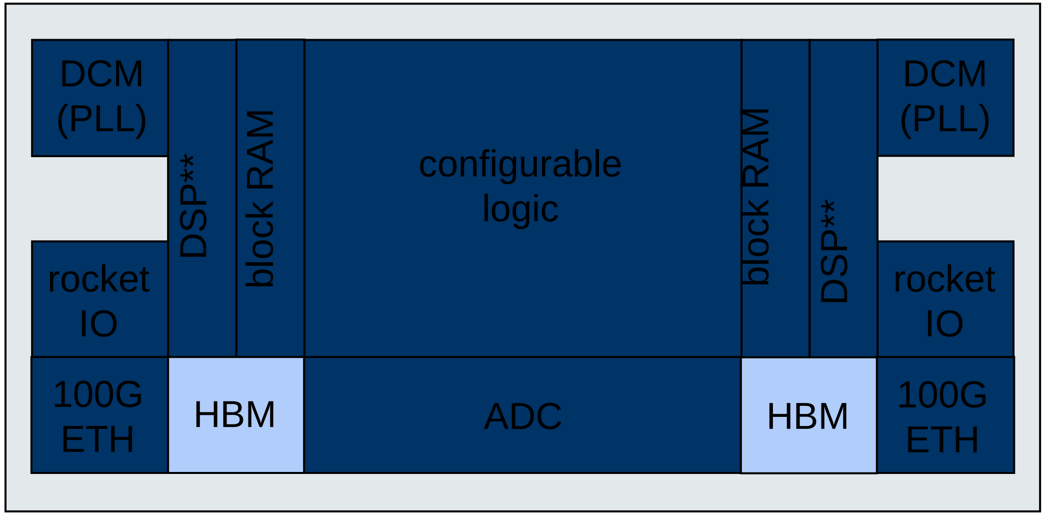


FPGA = Field-Programmable Gate Array

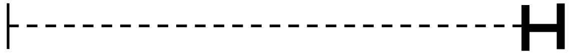
Evolution of Xilinx FPGAs
(selected devices + features)



1991: XC4000
1998: Virtex
2002: Virtex-II Pro
2004: Virtex-4
2006: Virtex-5
2009: Virtex-6
2010: 7 Series
2013: UltraScale
2015: UltraScale+



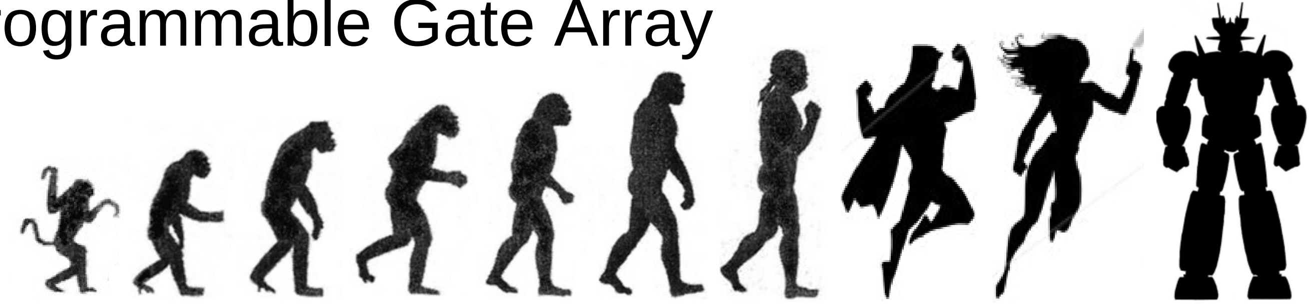
technology node:



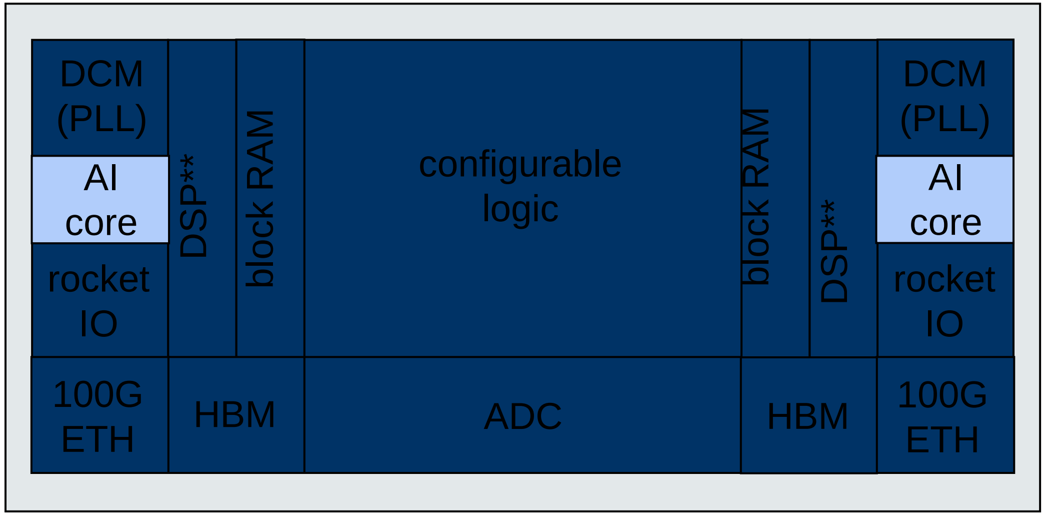
16 nm

Evolution of Xilinx FPGAs
(selected devices + features)

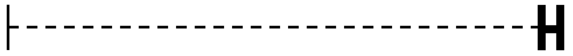
FPGA = Field-Programmable Gate Array



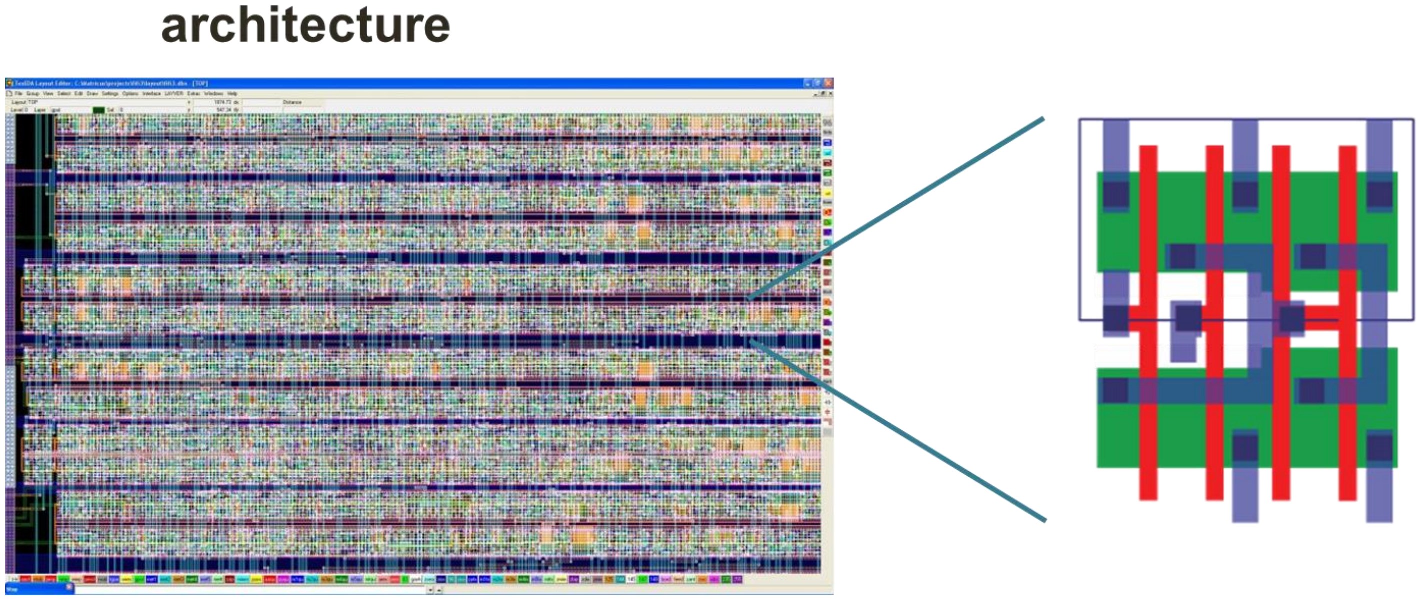
1991: XC4000
1998: Virtex
2002: Virtex-II Pro
2004: Virtex-4
2006: Virtex-5
2009: Virtex-6
2010: 7 Series
2013: UltraScale
2015: UltraScale+
2019: Versal



technology node:



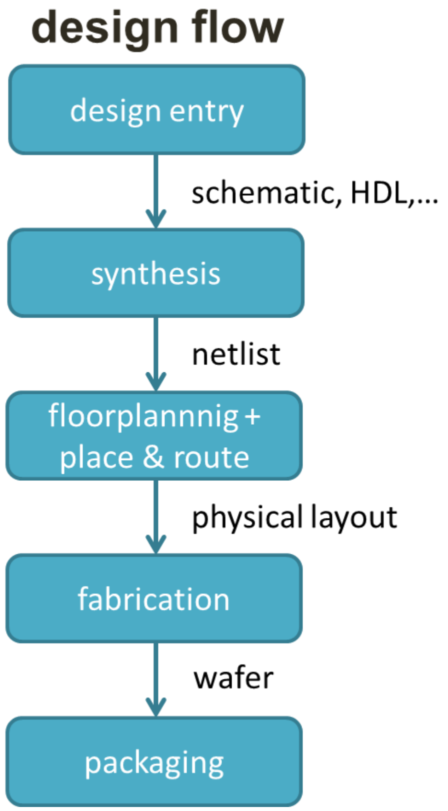
ASIC = Application-Specific Integrated Circuit



Basic components: standard cells from a standard cell library

- Logic cells and sequential cells

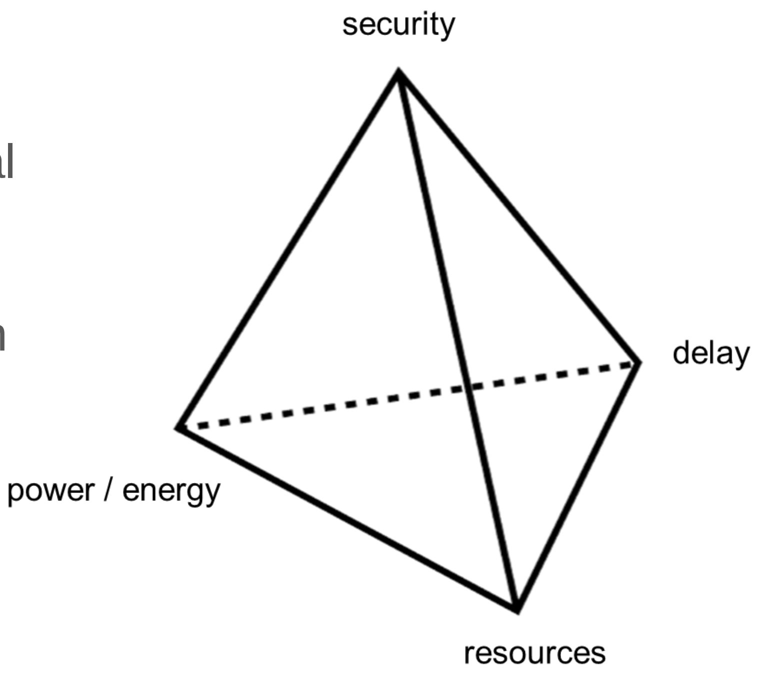
ASIC = Application-Specific Integrated Circuit



- The hardware architecture of an ASIC is fixed
- The code describes the hardware that we need
- The GDSII file contains the physical information that goes to the foundry

Metrics

- Metrics = what we measure (on the physical implementation platform or through simulation) to determine the efficiency of an implementation
- We use different units for these metrics on different implementation platforms



Metrics

Resources (typically related to cost):

- Microprocessor:
    - Size of the ROM (Flash) needed to store the program (in bytes)
    - Size of the SRAM needed to store the (intermediate) data (in bytes)

Metrics

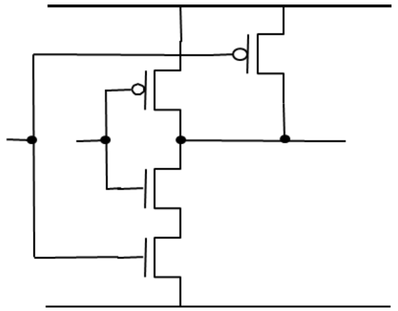
Resources (typically related to cost):

- FPGA:
  - Number of CLBs (or slices, or LUTs + flipflops)
  - Number of dedicated components (DSP slices, BRAM,…)

Metrics

Resources (typically related to cost):

- ASIC:
  - Number of equivalent NAND gates (Gate Equivalent = GE)
  - Area in μm^2



Metrics

Delay:

- Cycle count (in number of cycles)
- Latency = how long it takes to execute an algorithm from start to end = number of cycles divided by the clock frequency (in seconds)
- Throughput = how many bits are processed per second (in bits per second)

Metrics

Energy consumption:

- E = energy, P = power, t = execution time, I = current, V = supply voltage
- E = P * t (in Joule)
    - Amount of energy to execute an algorithm, capacity of the battery
- P = V * I (in Watt)
    - Instantaneous quantity, important for cooling and peak performance
    - Static + dynamic power consumption

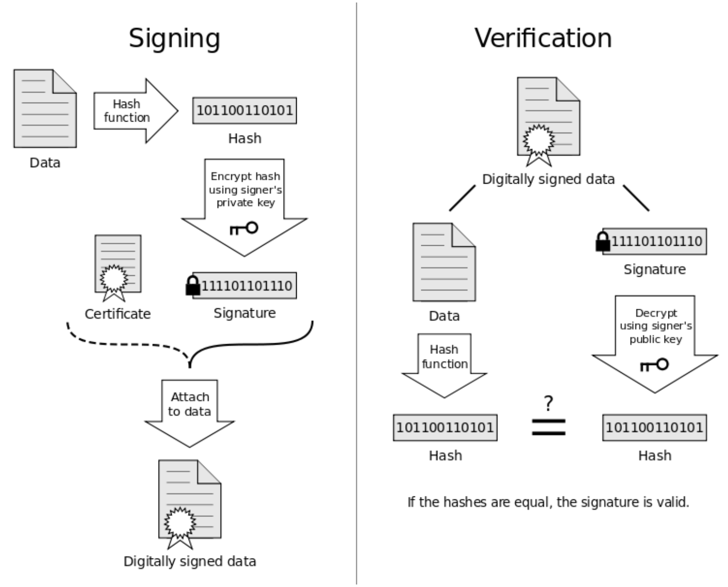
Outline

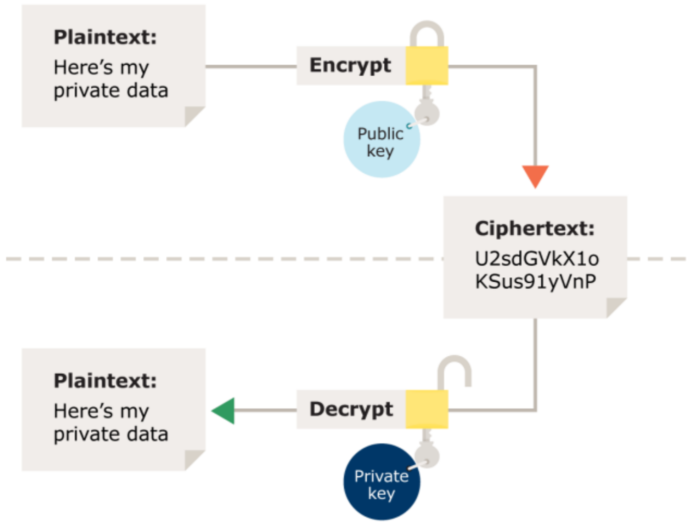
- Implementation platforms and metrics
- Traditional public-key crypto algorithms:
    - **RSA**
    - Elliptic Curve Cryptography (ECC)
- Post-quantum crypto algorithms

RSA

Reminder

- RSA is a public-key algorithm → public + private key pair
- RSA is used for digital signatures and public-key encryption/decryption





Reminder

Key generation:

- Choose two different large primes (p and q) and compute $n = p*q$
- Compute $\lambda(n)$
  $= \mathrm{lcm}(p-1,q-1)$
  $= (p-1)*(q-1)/\gcd(p-1,q-1)$
  (lcm = least common multiple)
  (gcd = greatest common divider)
- Choose $e < \lambda(n)$ and relatively prime to $\lambda(n)$
- Calculate $d = e^{-1} \bmod \lambda(n)$

Public key = (e,n)
Private key = (p,q) or d

Reminder

Key generation:

- Choose two different large primes (p and q) and compute $n = p*q$
- Compute $\lambda(n)$
  $= \mathrm{lcm}(p-1,q-1)$
  $= (p-1)*(q-1)/\gcd(p-1,q-1)$
  (lcm = least common multiple)
  (gcd = greatest common divider)
- Choose $e < \lambda(n)$ and relatively prime to $\lambda(n)$
- Calculate $d = e^{-1} \bmod \lambda(n)$

Public key = (e,n)
Private key = (p,q) or d

Public-key encryption/decryption:
Encryption: $c = m^e \bmod n$
Decryption: $m = c^d \bmod n$

Digital signature generation/verification:
Signature generation: $s = [H(m)]^d \bmod n$, then send m || s
Signature verification: $H(m) = s^e \bmod n$?

Efficient implementation of RSA

Key generation:

- Choose two different large primes (p and q) and compute $n = p*q$
- Compute $\lambda(n)$
  $= \mathrm{lcm}(p-1,q-1)$
  $= (p-1)*(q-1)/\gcd(p-1,q-1)$
  (lcm = least common multiple)
  (gcd = greatest common divider)
- Choose $e < \lambda(n)$ and relatively prime to $\lambda(n)$
- Calculate $d = e^{-1} \bmod \lambda(n)$

Public key = (e,n)
Private key = (p,q) or d

Key generation only needs to be done once, and can be done offline

Modular exponentiation is needed in each online step, and is a time- and energy-consuming operation

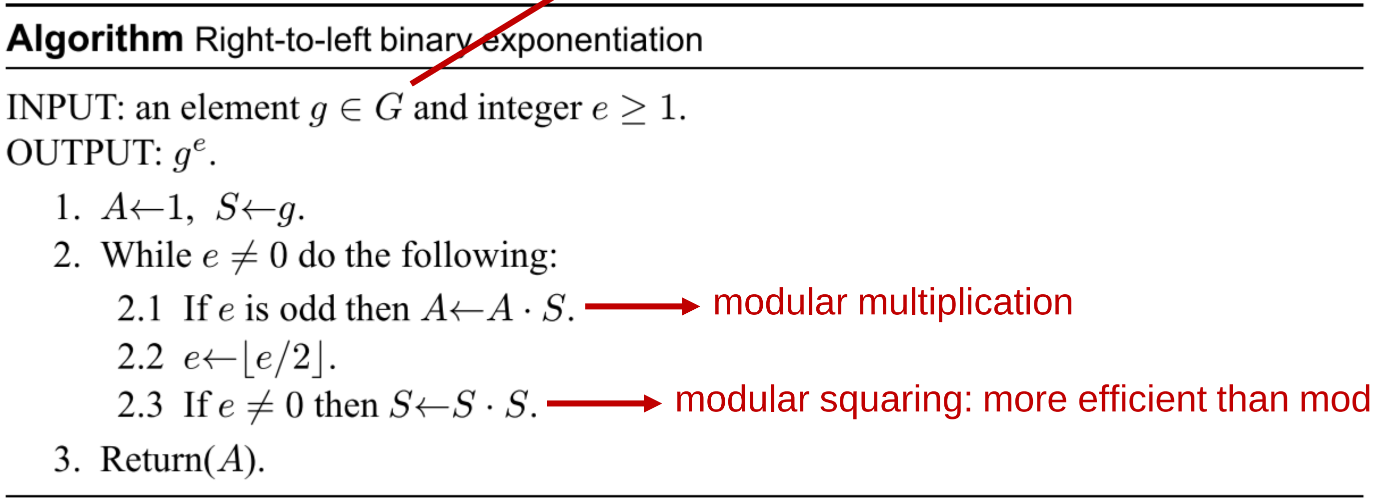
Public-key encryption/decryption:
Encryption: $c = m^e \bmod n$
Decryption: $m = c^d \bmod n$

Digital signature generation/verification:
Signature generation: $s = [H(m)]^d \bmod n$, then send m || s
Signature verification: $H(m) = s^e \bmod n$?

Efficient modular exponentiation

Naive algorithm:

G = finite group → all operations are modular (mod n)



Algorithm Right-to-left binary exponentiation

```
INPUT: an element $g \in G$ and integer $e \geq 1$.
OUTPUT: $g^e$.
  1. $A \leftarrow 1$, $S \leftarrow g$.
  2. While $e \neq 0$ do the following:
     2.1 If $e$ is odd then $A \leftarrow A \cdot S$.
     2.2 $e \leftarrow \lfloor e/2 \rfloor$.
     2.3 If $e \neq 0$ then $S \leftarrow S \cdot S$.
  3. Return($A$).
```

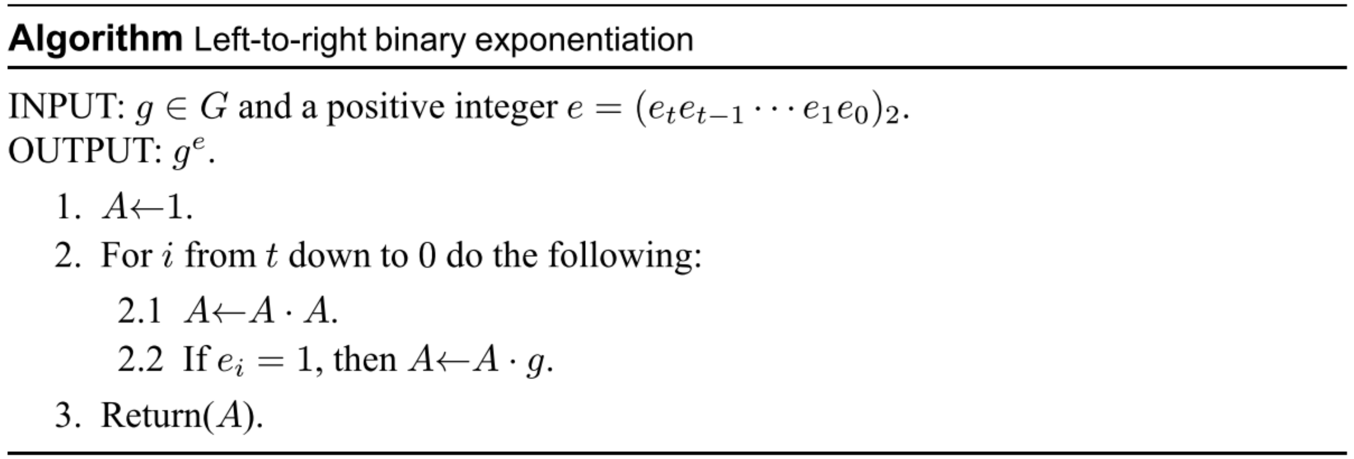
modular multiplication

modular squaring: more efficient than modular multiplication

Menezes et al., *Handbook of Applied Cryptography*, https://cacr.uwaterloo.ca/hac/

Efficient modular exponentiation

Another naive algorithm:



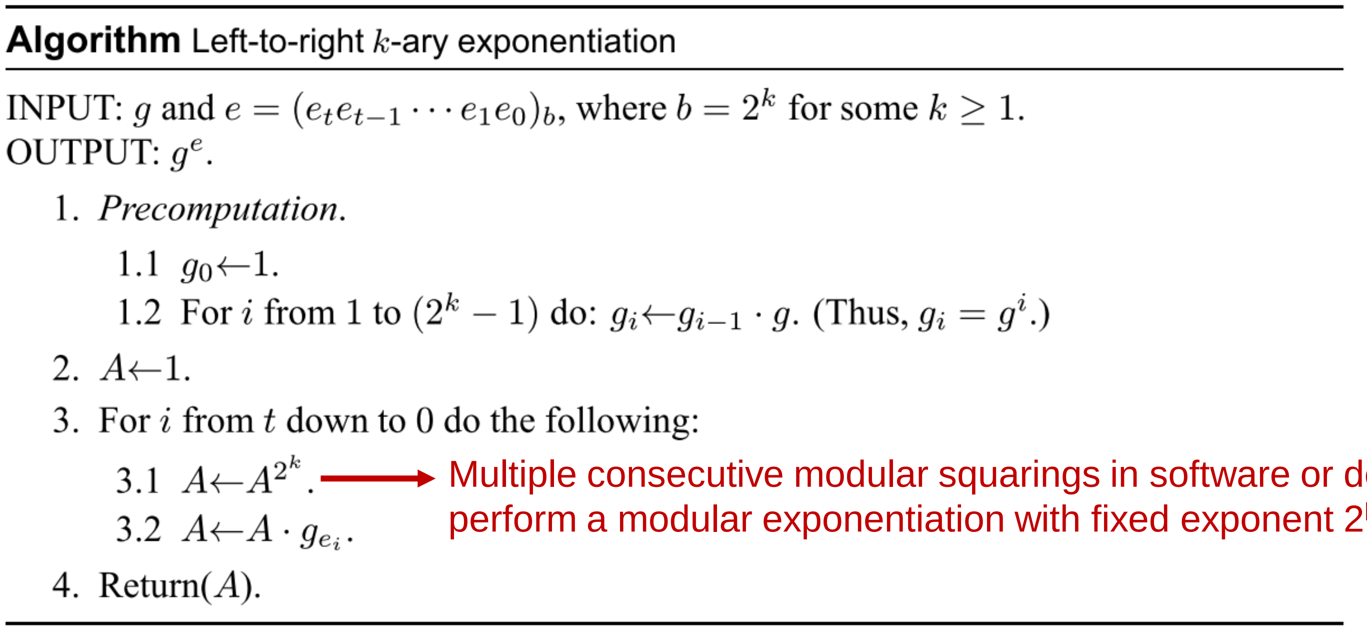
Algorithm Left-to-right binary exponentiation

INPUT: $g \in G$ and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$.
OUTPUT: g^e.

```
1. $A \leftarrow 1$.
2. For $i$ from $t$ down to 0 do the following:
    2.1 $A \leftarrow A \cdot A$.
    2.2 If $e_i = 1$, then $A \leftarrow A \cdot g$.
3. Return($A$).
```

Efficient modular exponentiation

First optimization: trading off memory for computational speed



Algorithm Left-to-right k-ary exponentiation

```
INPUT: $g$ and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$.
OUTPUT: $g^e$.
  1. *Precomputation.*
      1.1 $g_0 \leftarrow 1$.
      1.2 For $i$ from 1 to $(2^k - 1)$ do: $g_i \leftarrow g_{i-1} \cdot g$. (Thus, $g_i = g^i$.)
  2. $A \leftarrow 1$.
  3. For $i$ from $t$ down to 0 do the following:
      3.1 $A \leftarrow A^{2^k}$.
      3.2 $A \leftarrow A \cdot g_{e_i}$.
  4. Return($A$).
```

$\longrightarrow$ Multiple consecutive modular squarings in software or dedicated circuit to perform a modular exponentiation with fixed exponent 2^k in hardware

Exercises modular exponentiation

1) Assume: $g = 2$, $e = 303$, $n = 5$

   Compute $g^e \bmod n$ with the right-to-left binary method, the left-to-right binary method, and the left-to-right 2-ary method

2) Assume: $e = 2^{2048} – 3$, $n = 2^{2203} – 1$

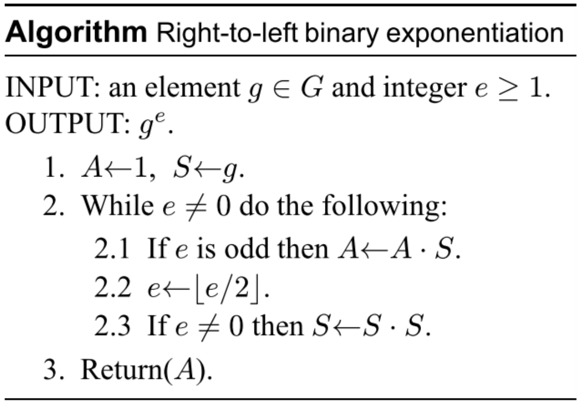
   How many modular multiplications and modular squarings are needed for each of the three methods?

Solution exercise 1: right-to-left with g = 2, e = 303, n = 5

A ← 1, S ← 2

- e = 303:
  - e is odd: A ← A.S = 1.2 = 2
  - e ← 151
  - e ≠ 0: S ← S.S = 2.2 = 4
- e = 151:
  - e is odd: A ← A.S = 2.4 = 3
  - e ← 75
  - e ≠ 0: S ← S.S = 4.4 = 1
- e = 75:
  - e is odd: A ← A.S = 3.1 = 3
  - e ← 37
  - e ≠ 0: S ← S.S = 1.1 = 1
- e = 37:
  - e is odd: A ← A.S = 3.1 = 3
  - e ← 18
  - e ≠ 0: S ← S.S = 1.1 = 1
- e = 18:
  - e ← 9
  - e ≠ 0: S ← S.S = 1.1 = 1
- e = 9:
  - e is odd: A ← A.S = 3.1 = 3
  - e ← 4
  - e ≠ 0: S ← S.S = 1.1 = 1
- e = 4:
  - e ← 2
  - e ≠ 0: S ← S.S = 1.1 = 1
- e = 2:
  - e ← 1
  - e ≠ 0: S ← S.S = 1.1 = 1
- e = 1:
  - e is odd: A ← A.S = 3.1 = 3
  - e ← 0

Return (3)



Algorithm Right-to-left binary exponentiation

INPUT: an element $g \in G$ and integer $e \geq 1$.
OUTPUT: g^e.

```
1. $A \leftarrow 1,\ S \leftarrow g$.
2. While $e \neq 0$ do the following:
    2.1 If $e$ is odd then $A \leftarrow A \cdot S$.
    2.2 $e \leftarrow \lfloor e/2 \rfloor$.
    2.3 If $e \neq 0$ then $S \leftarrow S \cdot S$.
3. Return($A$).
```

Solution exercise 1: left-to-right with g = 2, e = 303, n = 5

$e = (303)_{10} = (100101111)_2 = (e_8e_7e_6e_5e_4e_3e_2e_1e_0)_2$

A ← 1

i = 8:
- A ← A.A = 1.1 = 1
- e_8 = 1: A ← A.g = 1.2 = 2

i = 7:
- A ← A.A = 2.2 = 4

i = 6:
- A ← A.A = 4.4 = 1

i = 5:
- A ← A.A = 1.1 = 1
- e_5 = 1: A ← A.g = 1.2 = 2

i = 4:
- A ← A.A = 2.2 = 4

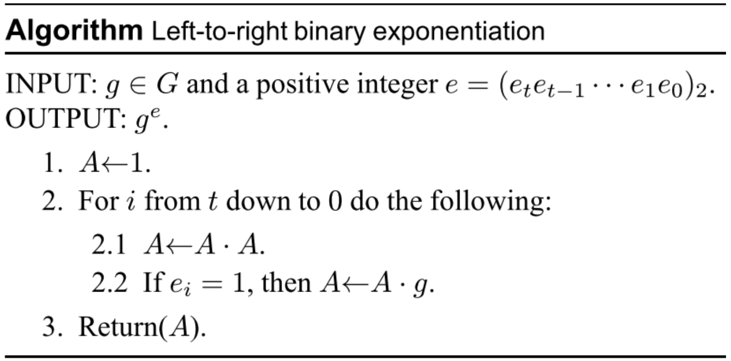
i = 3:
- A ← A.A = 4.4 = 1
- e_3 = 1: A ← A.g = 1.2 = 2

i = 2:
- A ← A.A = 2.2 = 4
- e_2 = 1: A ← A.g = 4.2 = 3

i = 1:
- A ← A.A = 3.3 = 4
- e_1 = 1: A ← A.g = 4.2 = 3

i = 0:
- A ← A.A = 3.3 = 4
- e_0 = 1: A ← A.g = 4.2 = 3

Return (3)



Algorithm Left-to-right binary exponentiation

INPUT: $g \in G$ and a positive integer $e = (e_te_{t-1} \cdots e_1e_0)_2$.
OUTPUT: g^e.

```
1. $A \leftarrow 1$.
2. For $i$ from $t$ down to 0 do the following:
    2.1 $A \leftarrow A \cdot A$.
    2.2 If $e_i = 1$, then $A \leftarrow A \cdot g$.
3. Return($A$).
```

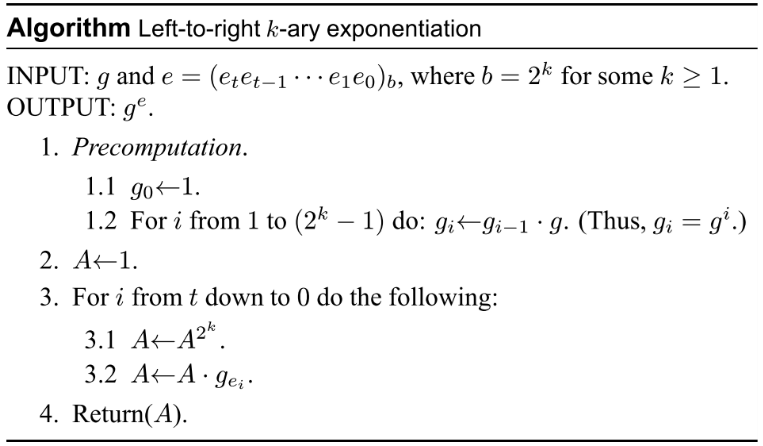
Solution exercise 1: k-ary with k = 2, g = 2, e = 303, n = 5

$e = (303)_{10} = (100101111)_2 = (10233)_4 = (e_4e_3e_2e_1e_0)_4$

Precomputation

```
        $g_0 \leftarrow 1$
        i = 1: $g_1 \leftarrow g_0.g = 1.2 = 2$
        i = 2: $g_2 \leftarrow g_1.g = 2.2 = 4$
        i = 3: $g_3 \leftarrow g_2.g = 4.2 = 3$

$A \leftarrow 1$
i = 4:
        $A \leftarrow A^4 = 1^4 = 1$
        $A \leftarrow A.g_1 = 1.2 = 2$
i = 3:
        $A \leftarrow A^4 = 2^4 = 1$
        $A \leftarrow A.g_0 = 1.1 = 1$
i = 2:
        $A \leftarrow A^4 = 1^4 = 1$
        $A \leftarrow A.g_2 = 1.4 = 4$
i = 1:
        $A \leftarrow A^4 = 4^4 = 1$
        $A \leftarrow A.g_3 = 1.3 = 3$
i = 0:
        $A \leftarrow A^4 = 3^4 = 1$
        $A \leftarrow A.g_3 = 1.3 = 3$
Return (3)
```



Algorithm Left-to-right k-ary exponentiation

INPUT: g and $e = (e_te_{t-1} \cdots e_1e_0)_b$, where $b = 2^k$ for some $k \geq 1$.
OUTPUT: g^e.

```
1. *Precomputation.*
    1.1 $g_0 \leftarrow 1$.
    1.2 For $i$ from 1 to $(2^k - 1)$ do: $g_i \leftarrow g_{i-1} \cdot g$. (Thus, $g_i = g^i$.)
2. $A \leftarrow 1$.
3. For $i$ from $t$ down to 0 do the following:
    3.1 $A \leftarrow A^{2^k}$.
    3.2 $A \leftarrow A \cdot g_{e_i}$.
4. Return($A$).
```

Solution exercise 2: $e = 2^{2048} - 3$, $n = 2^{2203} - 1$

Right-to-left binary exponentiation and right-to-left binary exponentiation

$e = 2^{2048} - 3 = (e_{2047}e_{2046} \ldots e_1e_0)_2$

2048 modular squarings

$2^{2048} - 1 = (1111 \ldots 1111)_2$ (all ones)
$2^{2048} - 3 = (1111 \ldots 1101)_2$ (all ones but e1) → 2047 ones → **2047 modular multiplications**

2-ary modular exponentiation

$e = 2^{2048} - 3 = (e_{1023}e_{1022} \ldots e_1e_0)_4$

Precomputation: 3 modular multiplications

Computation: 1024 times $A \leftarrow A^{2^k}$. = 1024 * 2 modular squarings

$A \leftarrow A \cdot g_{e_i}$. = 1024 modular multiplications

Total: **2048 modular squarings** and **1027 modular multiplications**

Modular exponentiation: further optimization

The Chinese remainder theorem (CRT) can be used for the modular exponentiation in RSA decryption and signature generation. It reduces the computation to modular exponentiations with exponents of half the size, taking into account that n = p*q.

Precomputation:

1. $d_p = d \mod (p-1)$;
2. $d_q = d \mod (q-1)$;
3. $q_{inv} = (1/q) \mod p$, with $p > q$.

```
Algorithm    RSA decryption using the Chinese Remainder Theorem
Require: $p, q, d_p, d_q, q_{inv}, c,$
Ensure: $c^d$
  1: $m1 \leftarrow c^{d_p} \mod p$
  2: $m2 \leftarrow c^{d_q} \mod q$
  3: $h \leftarrow (q_{inv} \cdot (m1 - m2)) \mod p$
  4: $m \leftarrow m2 + h \cdot q$
  5: Return $m$
```

Question: Why can’t we use CRT for RSA encryption and signature verification?

Efficient modular multiplication

Naive algorithm:

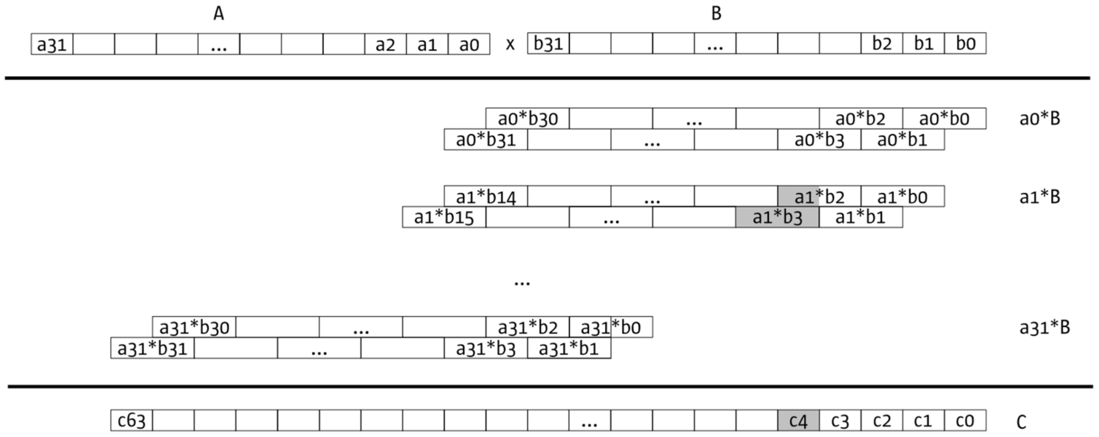
Algorithm Classical modular multiplication

```
INPUT: two positive integers $x$, $y$ and a modulus $m$, all in radix $b$ representation.
OUTPUT: $x \cdot y \bmod m$.
    1. Compute $x \cdot y$
    2. Compute the remainder $r$ when $x \cdot y$ is divided by $m$
    3. Return($r$).
```

integer multiplication

integer division (very compute-intensive)

Multi-precision integer multiplication:



Efficient modular multiplication

Montgomery multiplication:
(no integer division needed, but x*y/R is calculated instead of x*y, where R is a power of 2):

Algorithm Montgomery multiplication

```
INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $y = (y_{n-1} \cdots y_1 y_0)_b$
with $0 \le x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and $m' = -m^{-1} \bmod b$.
OUTPUT: $xyR^{-1} \bmod m$.
    1. $A \leftarrow 0$. (Notation: $A = (a_n a_{n-1} \cdots a_1 a_0)_b$.)
    2. For $i$ from 0 to $(n-1)$ do the following:
        2.1 $u_i \leftarrow (a_0 + x_i y_0) m' \bmod b$.
        2.2 $A \leftarrow (A + x_i y + u_i m)/b$.
    3. If $A \ge m$ then $A \leftarrow A - m$.
    4. Return($A$).
```

When b is a power of 2, mod b simply means cutting off the most-significant bits

When b is a power of 2, /b simply means shifting the bits to the right

Efficient modular multiplication

Montgomery multiplication:

- If we use Montgomery multiplication directly on the input values x and y, we calculate x*y/R mod n instead of x*y mod n
- Therefore, all inputs need to be transformed to “Montgomery representation” first: x_M = x*R mod n, y_M = x*R mod n
- Transforming the inputs to Montgomery representation can be done by multiplying each input with R^2 using Montgomery multiplication: x_M = MontMul(x, R^2) = x*R^2/R mod n = x*R mod n, y_M = MontMul(y, R^2) = y*R^2/R mod n = y*R mod n
- The product will also be in Montgomery representation: a_M = MontMul(x_M, y_M) = x_M*y_M/R = x*R*y*R/R = x*y*R
- We stick to Montgomery representation until the entire modular exponentiation is completed. Only at the very end, a transformation is done from Montgomery representation back to normal representation. This can be done by multiplying the result of the modular exponentiation with 1 using Montgomery multiplication: MontMul(c_M, 1) = c_M*1/R = c*R*1/R = c

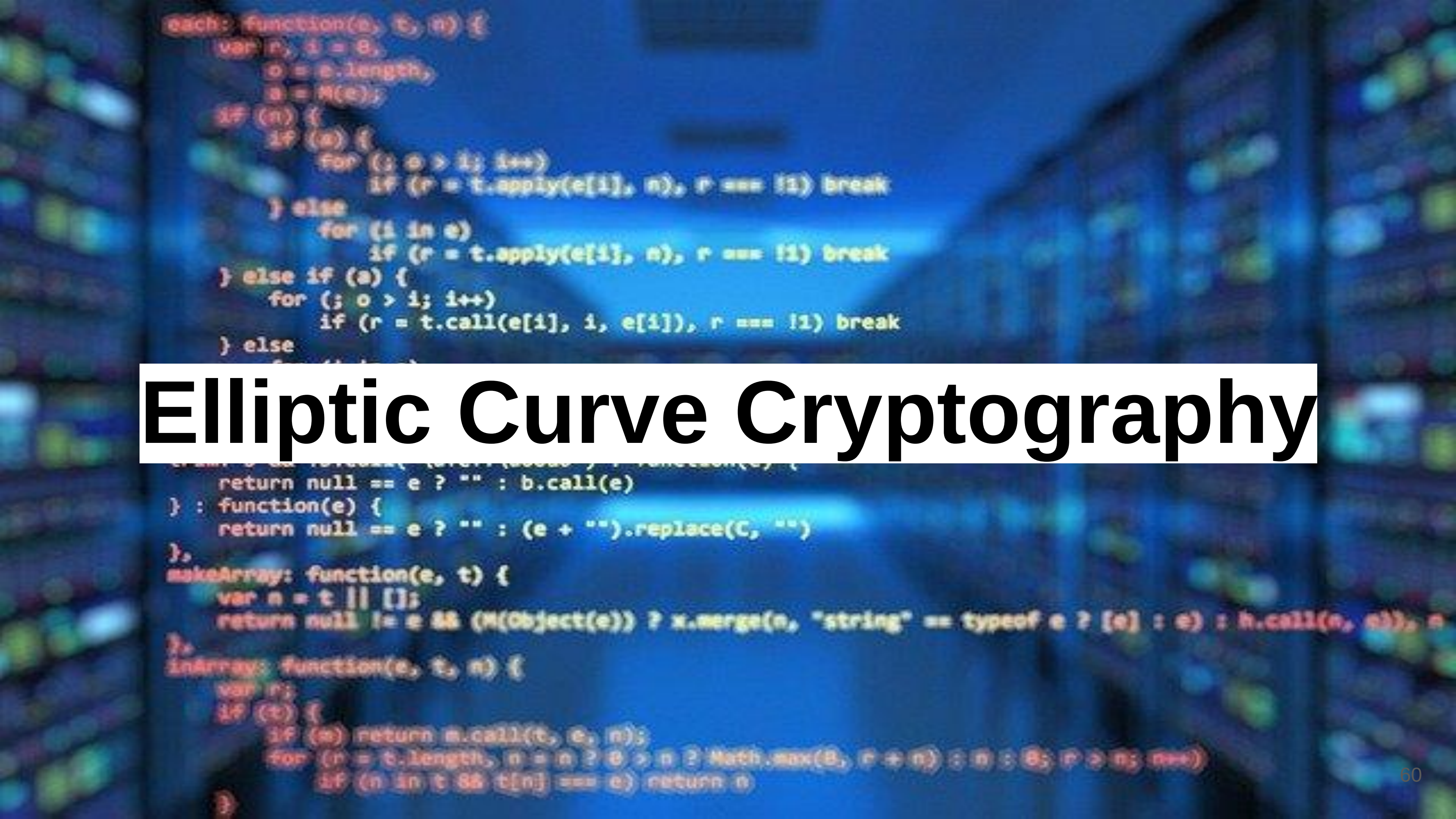
Efficient modular multiplication

- Montgomery multiplication in software:
  - Several methods to process the data in smaller chunks
  - Follow the original algorithm (2 slides back) or optimizations first introduced in

    Koc et al., Analyzing and comparing Montgomery multiplication algorithms, in IEEE Micro 16(3), pp. 26-33, 1996.
- Montgomery multiplication in hardware:
  - It is possible to process the data in parallel
  - FPGA: use DSP slices (and dedicated shift registers)
  - ASIC: use multipliers in the standard cell library

Outline

- Implementation platforms and metrics
- Traditional public-key crypto algorithms:
    - RSA
    - **Elliptic Curve Cryptography (ECC)**
- Post-quantum crypto algorithms

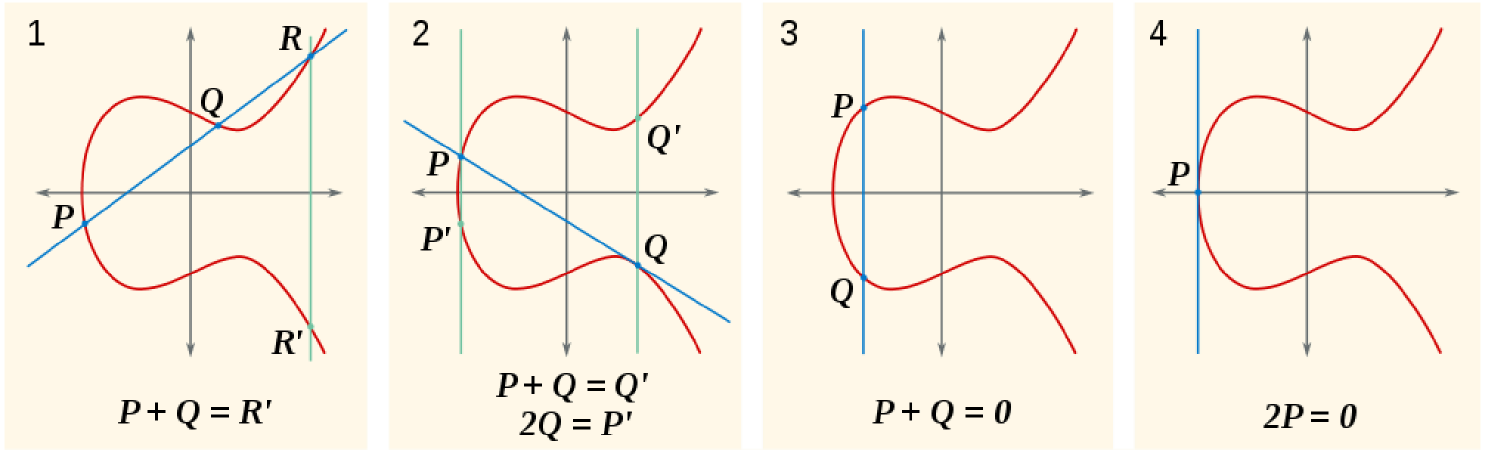


```
each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], i, e[i]), r === !1) break
    } else
        [illegible]
[illegible]
    return null == e ? "" : b.call(e)
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "")
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string" == typeof e ? [e] : e) : h.call(n, e)), n
},
inArray: function(e, t, n) {
    var r;
    if (t) {
        if (m) return m.call(t, e, n);
        for (r = t.length, n = n ? 0 > n ? Math.max(0, r + n) : n : 0; r > n; n++)
            if (n in t && t[n] === e) return n
    }
```

Elliptic Curve Cryptography

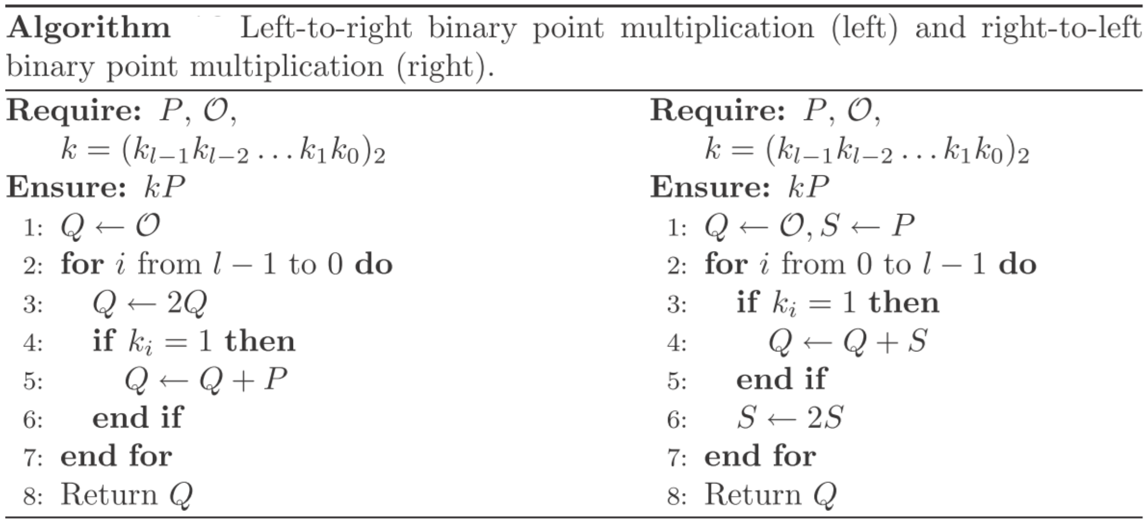
Elliptic Curve Cryptography

- An elliptic curve is the set of solutions to the equation $y^2 = x^3 + ax + b$
- The points on the elliptic curve together with the point at infinity form a group
- The group operation is point addition



Elliptic Curve Cryptography

- Elliptic curve point multiplication is used analogous to modular exponentiation
- Point mulplication multiplies a point on the curve with a scalar (an integer)
- Point multiplication can be done through consecutive point additions and point doublings (or k-ary point multiplication)



Algorithm Left-to-right binary point multiplication (left) and right-to-left binary point multiplication (right).

```
Require: $P$, $\mathcal{O}$,
    $k = (k_{l-1}k_{l-2}\dots k_1k_0)_2$
Ensure: $kP$
1: $Q \leftarrow \mathcal{O}$
2: for $i$ from $l-1$ to $0$ do
3:     $Q \leftarrow 2Q$
4:     if $k_i = 1$ then
5:         $Q \leftarrow Q + P$
6:     end if
7: end for
8: Return $Q$
```

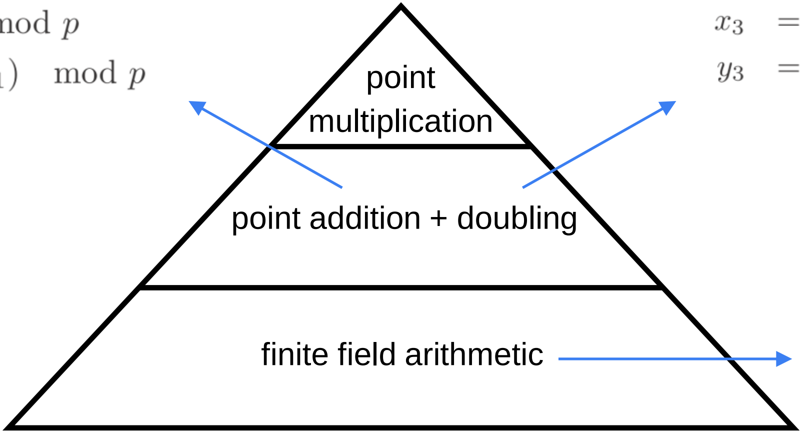
```
Require: $P$, $\mathcal{O}$,
    $k = (k_{l-1}k_{l-2}\dots k_1k_0)_2$
Ensure: $kP$
1: $Q \leftarrow \mathcal{O}, S \leftarrow P$
2: for $i$ from $0$ to $l-1$ do
3:     if $k_i = 1$ then
4:         $Q \leftarrow Q + S$
5:     end if
6:     $S \leftarrow 2S$
7: end for
8: Return $Q$
```

Elliptic Curve Cryptography

- Point addition and point doubling are computed through modular additions, subtractions, multiplications and divisions (divisions can be avoided with special coordinate systems)

$$\begin{aligned} P_3 &= P_1 + P_2 = (x_3, y_3) \\ \text{with} \quad \lambda &= \frac{y_2 - y_1}{x_2 - x1} \mod p \\ x_3 &= (\lambda^2 - x_1 - x_2) \mod p \\ y_3 &= ((x_1 - x_3) \cdot \lambda - y_1) \mod p \end{aligned}$$

$$\begin{aligned} P_3 &= 2P_1 = (x_3, y_3) \\ \text{with} \quad \lambda &= \frac{3 \cdot x_1^2 + a}{2 \cdot y_1} \mod p \\ x_3 &= (\lambda^2 - 2 \cdot x_1) \mod p \\ y_3 &= ((x_1 - x_3) \cdot \lambda - y_1) \mod p \end{aligned}$$

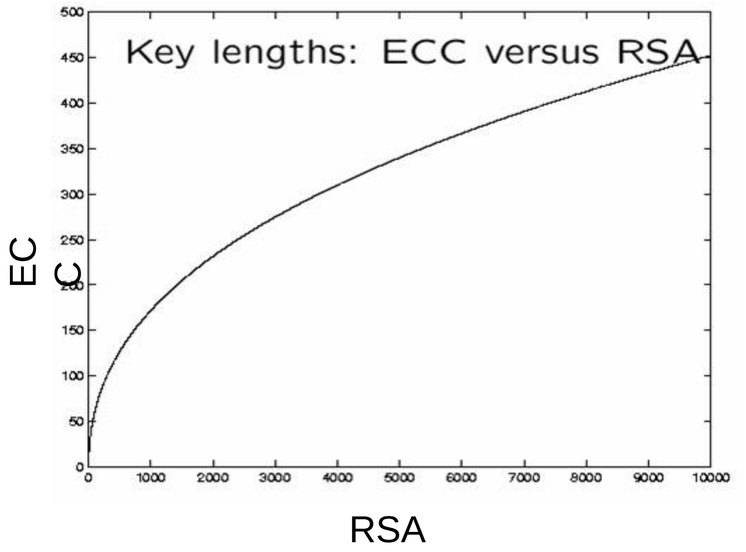


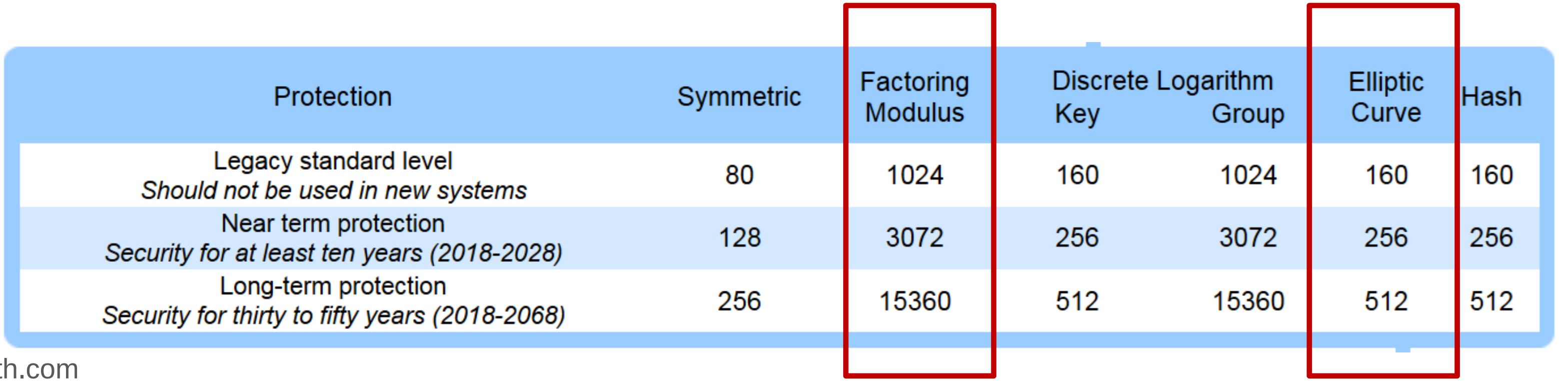
All computations in Montgomery representation

Elliptic Curve Cryptography

ECC (vs. RSA):

- Shorter parameters and signatures
- Security grows exponentially with length of parameters
- More efficient on most platforms



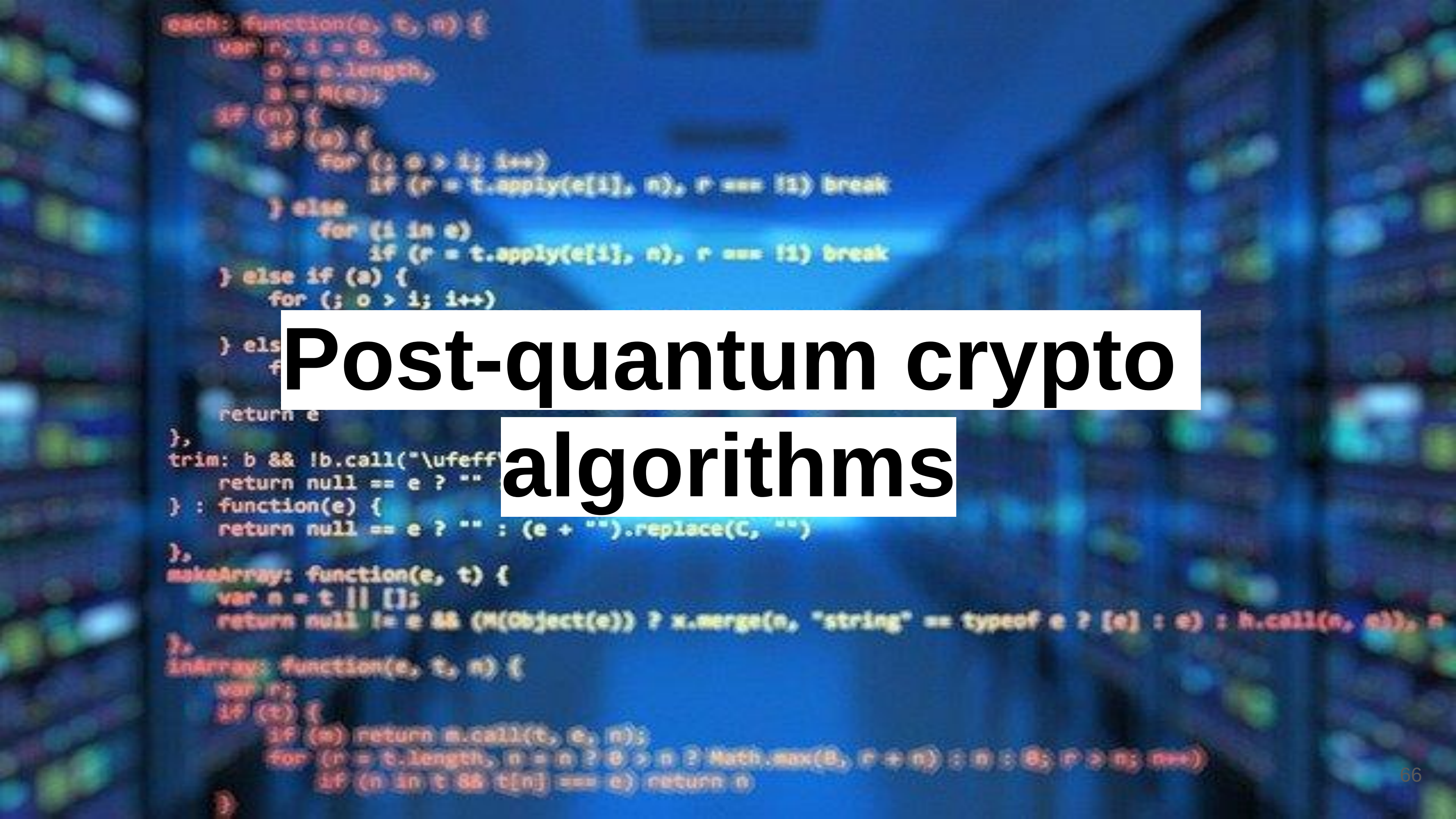


Protection	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash
Legacy standard level *Should not be used in new systems*	80	1024	160	1024	160	160
Near term protection *Security for at least ten years (2018-2028)*	128	3072	256	3072	256	256
Long-term protection *Security for thirty to fifty years (2018-2068)*	256	15360	512	15360	512	512

keylength.com

Outline

- Implementation platforms and metrics
- Traditional public-key crypto algorithms:
    - RSA
    - Elliptic Curve Cryptography (ECC)
- **Post-quantum crypto algorithms**



Post-quantum crypto algorithms

NIST Post-quantum competition

- When quantum computers become available, Shor's algorithm can be used to calculate discrete logarithms and to do integer factorization in polynomial time
- All algorithms that rely on the discrete logarithm problem or integer factorization (e.g., RSA) will become insecure
- That is why NIST issued a competition to search for new algorithms that do not rely on the discrete logarithm problem or integer factorization
- These algorithms are called post-quantum crypto algorithms
- Note: symmetric-key ciphers, like AES, will not become insecure when quantum computers become available, because they do not rely on the discrete logarithm problem or integer factorization

NIST Post-quantum competition

Two categories of algorithms:

- KEM/Encryption: Lattice, Code, Other

  (KEM = Key Encapsulation Mechanism: to enable the exchange of symmetric keys using public-key cryptography)

- Digital Signature: Lattice (Ring-LWE Signature, NTRU, BLISS), Hash/Symmetric, Multivariate (Unbalanced Oil and Vinegar), Supersingular Elliptic Curve Isogeny

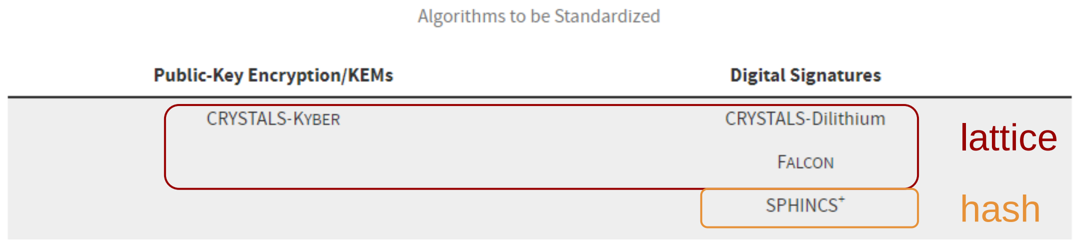
NIST Post-quantum competition

Implementation requirements:

- Lattice: based on finite field arithmetic
- Code: based on error correction codes
- Hash/Symmetric: based on symmetric-key crypto and hash functions
- Multivariate: based on finite field arithmetic
- Supersingular Elliptic Curve Isogeny: based on elliptic curve operations

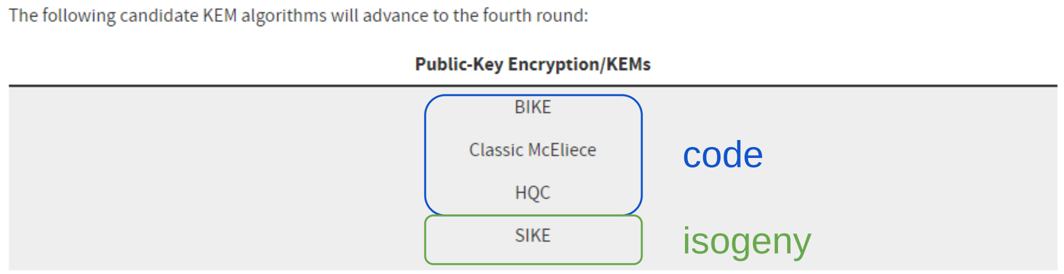
NIST Post-quantum competition

https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4


lattice
hash

Algorithms to be Standardized

Public-Key Encryption/KEMs	Digital Signatures
CRYSTALS-KYBER	CRYSTALS-Dilithium
	FALCON
	SPHINCS$^+$

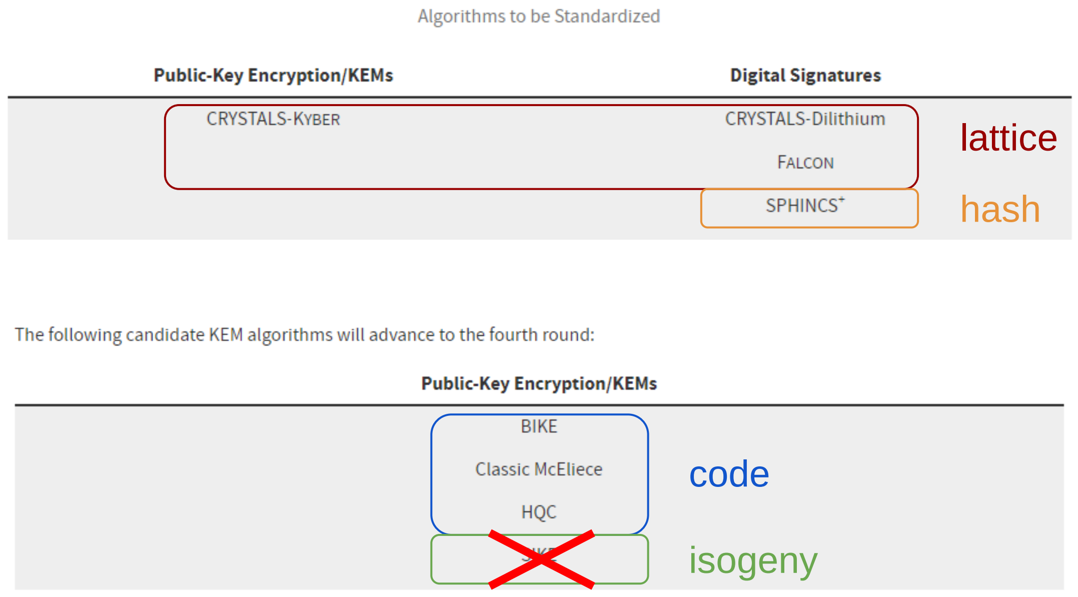

code
isogeny

The following candidate KEM algorithms will advance to the fourth round:

Public-Key Encryption/KEMs
BIKE
Classic McEliece
HQC
SIKE

NIST Post-quantum competition

https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4



Algorithms to be Standardized

Public-Key Encryption/KEMs	Digital Signatures
CRYSTALS-KYBER	CRYSTALS-Dilithium
	FALCON
	SPHINCS$^+$

lattice

hash

The following candidate KEM algorithms will advance to the fourth round:

Public-Key Encryption/KEMs
BIKE
Classic McEliece
HQC
~~SIKE~~

code

isogeny

BROKEN