# Introduction to Efficient and Secure Arithmetic Circuits

Arnaud TISSERAND

CNRS, Lab-STICC

*Conférence rentrée informatique ENS Paris-Scalay*. Sept. 2021

# Course Language

Slides have been prepared in *English*.

Some words/remarks are also given in FR *French* in case of not immediate translation or specific feature.

Questions, comments and help requests are welcome in both French and English.

# Licence

This document is licensed under a Creative Commons Attribution - NonCommercial 4.0 International License.

https://creativecommons.org/licenses/by-nc/4.0/

All figures and tables not from the author are presented with their source.

# Summary

Computer Arithmetic

Preliminaries on Digital Circuits

Addition & Multiplication

Introduction to Physical Attacks

Protections at the Arithmetic Level

References

*References to books, articles and links are given throughout and at the end of this document.*

# Computer Arithmetic

# What is Computer Arithmetic? (Personal Definition)

Branch of *computer engineering/science* that deals with:

- representations of numbers: formats, coding and behavior for (subsets of) $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{F}_q$, ..., fixed *vs* multiple precision;
- algorithms for operations: $\pm$, $\times$, $\div$, $\sqrt{\ }$, $\frac{1}{x}$, $\frac{1}{\sqrt{x}}$, $\frac{1}{\sqrt{x^2+y^2}}$, exp, log, sin, cos, mod, gcd, $(a+b) \bmod p$, conversions, ...;
- implementations in hardware or(/and) software;
- quality: error/accuracy, specific cases (div. by 0), reproducibility;
- speed: delay, latency, throughput;
- costs: silicon area, code/data memory, power/energy consumption;
- methods and tools: study, coding, validation, verification, porting, evaluation, ...;
- training of programmers and users;
- ?

# Computer Arithmetic Overview

representations of numbers

$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \simeq \mathbb{R}, \mathbb{F}_q$

algorithms

$\pm, \times, \div, \sqrt[q]{\ }, \mathrm{mod},$

$\lesseqqgtr, e^x, \simeq f(x), \ldots$

# Computer Arithmetic Overview

representations of numbers
$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \simeq \mathbb{R}, \mathbb{F}_q$

algorithms
$\pm, \times, \div, \sqrt[q]{}, \bmod,$
$\lesseqqgtr, e^x, \simeq f(x), \ldots$

implementation
soft GPP/SP,
ASIC, FPGA

# Computer Arithmetic Overview

representations of numbers
$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \simeq \mathbb{R}, \mathbb{F}_q$

algorithms
$\pm, \times, \div, \sqrt[q]{}, \bmod,$
$\lesseqqgtr, e^x, \simeq f(x), \ldots$

implementation
soft GPP/SP,
ASIC, FPGA

application

# Computer Arithmetic Overview

representations of numbers

$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \simeq \mathbb{R}, \mathbb{F}_q$

algorithms

$\pm, \times, \div, \sqrt[q]{}, \text{mod},$

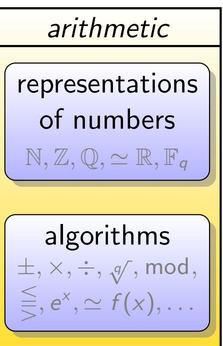
$\lesseqgtr, e^x, \simeq f(x), \ldots$

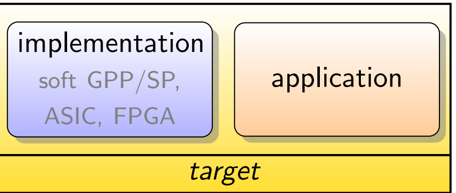
implementation

soft GPP/SP, ASIC, FPGA

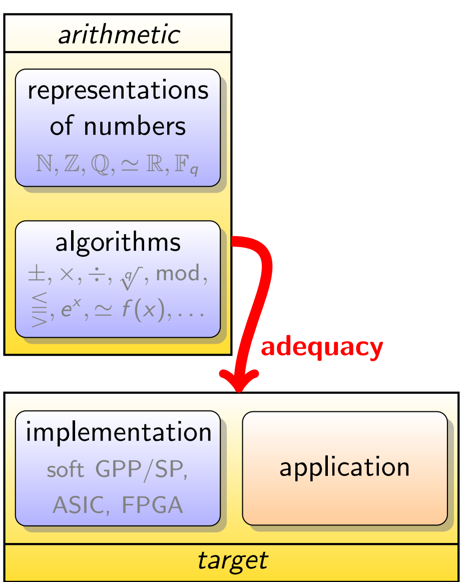
application

# Computer Arithmetic Overview

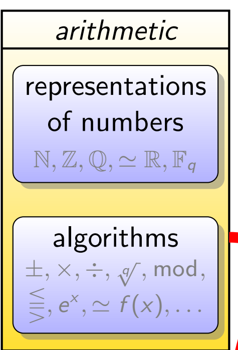


# Computer Arithmetic Overview



# Computer Arithmetic Overview




**adequacy**

implementation
soft GPP/SP, ASIC, FPGA

application

*target*

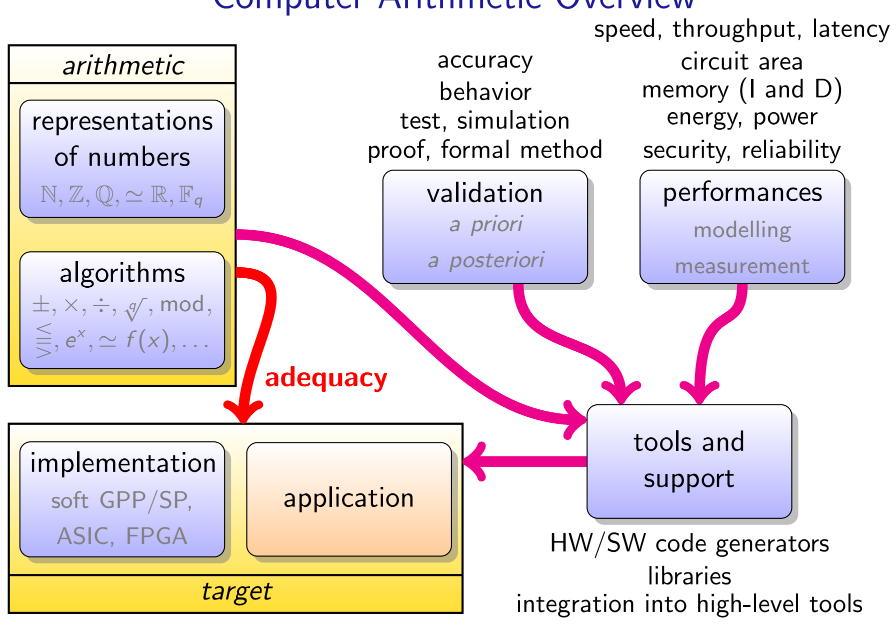
accuracy
behavior
test, simulation
proof, formal method

validation
*a priori*
*a posteriori*

speed, throughput, latency
circuit area
memory (I and D)
energy, power
security, reliability

performances
modelling
measurement

# Computer Arithmetic Overview



# Computer Arithmetic Close Domains

- microelectronics for digital circuits;
- computer architecture for processor design (units, instructions, registers, interruptions, counters . . . );
- programming languages and compilation;
- numerical computing and applied mathematics;
- formal proofs and verification methods;
- computer algebra ( FR *calcul formel* );
- specific application domains such as signal and image processing, AI
- and probably other domains. . .

# Computer Arithmetic in Software (Example SW1)

The following Python code:

```
a, b = 1, 9
c = a + b
print(c, type(c))

from math import *
x = pi + 1.0
print(x, type(x))

print([ sin(pi/n) for n in [4, 6, 12] ])
```

produces (using Python 3.7):

```
10 <class 'int'>
4.141592653589793 <class 'float'>
[0.7071067811865475, 0.49999999999999994, 0.25881904510252074]
```

Warning: do not perform *large* computations using "raw" Python, use NumPy standard library (see also Numba or PyPy)!

# Computer Arithmetic in Software (Example SW2)

The following C code:

```c
#include <stdio.h>
#include <math.h>

int main() {
  double n = 4.0;
  double x = M_PI;
  double y = sin(x/n);
  printf("y = %f\n", y);
  return 0;
}
```

compiled (gcc 8.3) using:

```
gcc -lm example_sin.c
```

produces:

```
y = 0.707107
```

# Computer Arithmetic in Software (Example SW3)

The following Python code:

```
a = 1.0
b = 12.345e50
c = 9.8765e-40
v = [a, -a, b, -b, c, -c]
print(sum(v))

from itertools import permutations
print(sorted(set( [ sum(e) for e in permutations(v) ] )))
```

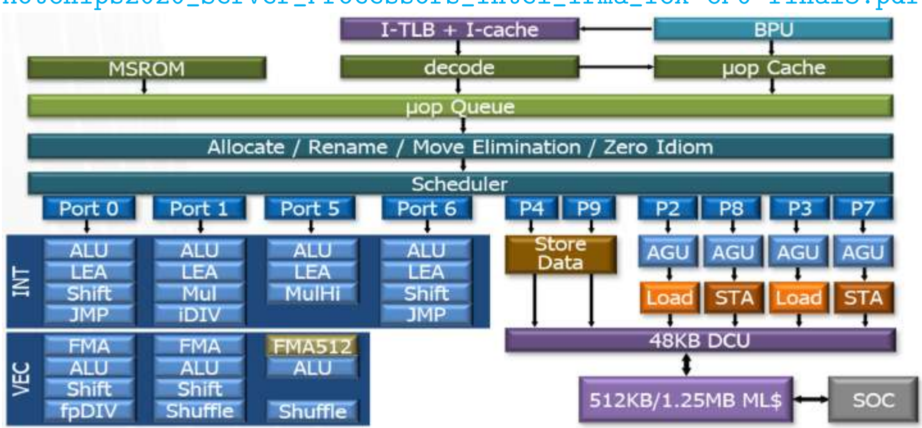
produces (using Python 3.7):

```
0.0
[-1.0, -9.8765e-40, 0.0, 9.8765e-40, 1.0]
```

Warning: associativity does not (necessarily) hold for floating-point arithmetic! See for instance: *David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23, March 1991, DOI:* https://doi.org/10.1145/103162.103163

# Computer Arithmetic in Hardware (Example HW1)

Overview of one core in an Intel Xeon processor, source: https://www.hc32.hotchips.org/assets/program/conference/day1/HotChips2020_Server_Processors_Intel_Irma_ICX-CPU-final3.pdf




Link to other examples: https://en.wikichip.org/wiki/WikiChip

# Computer Arithmetic in Hardware (Example HW2)

Source: *NVIDIA TURING GPU ARCHITECTURE* white paper (WP-09183-001_v01)

See also: `https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/`

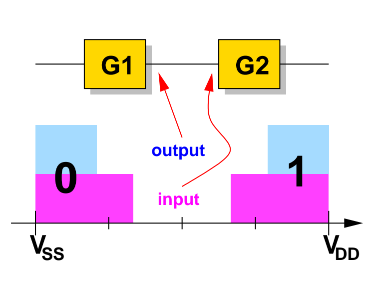
# Preliminaries on Digital Circuits

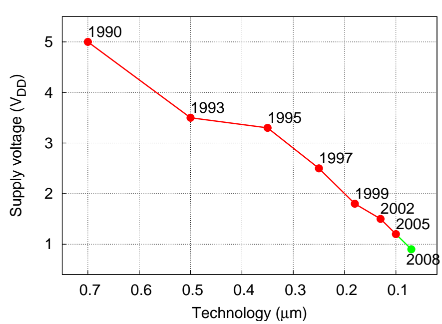
# Logic Values: Representation

The logic values $\{0, 1\}$ are represented using voltages:

- $0 \iff$ reference voltage or ground ($V_{SS}$, ⏚)
- $1 \iff$ supply voltage ($V_{DD} > 0$ or ↑)

Due to the noise in the circuit (from many sources), the logic values must be represented using voltage intervals (noise margins): digital vs. analog

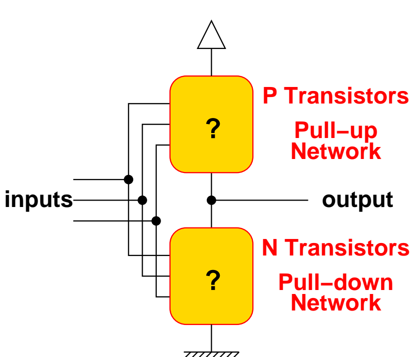


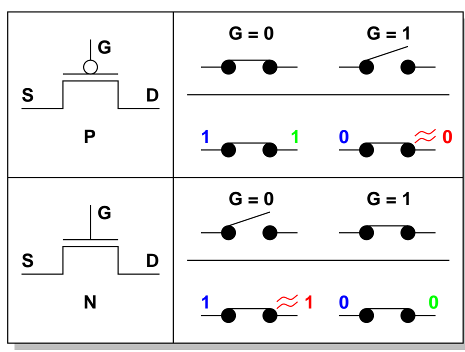



# CMOS Logic

CMOS = *complementary MOS*

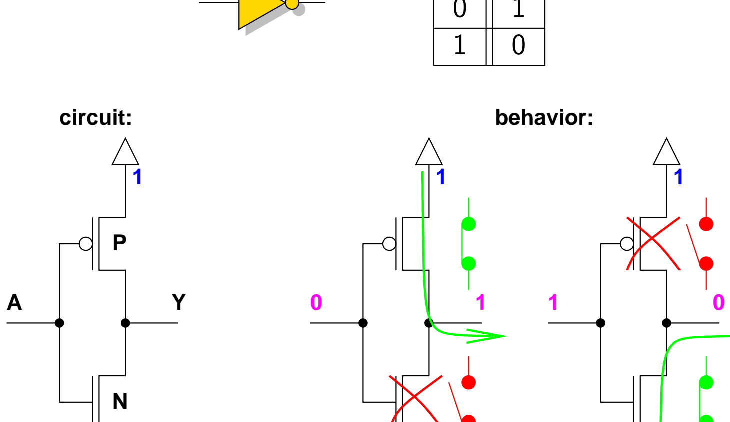
N and P transistors are only used for passing strong signals



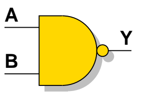
# Logic Gate: Inverter

The simplest gate: only 2 transistors (1 N and 1 P)

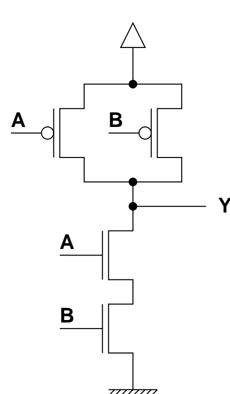



| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

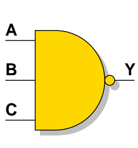
# Logic Gate: NAND2 (2-input not–and)



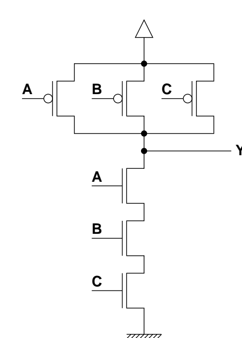

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



# Logic Gate: NAND3 (3-input NAND)



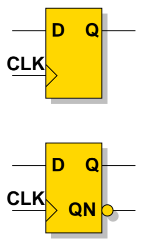

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |




The number of transistors in series is limited (3 to 5)

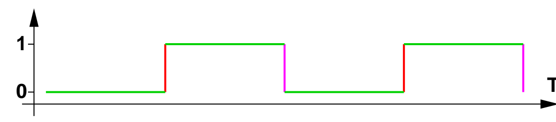
# Memory Elements

There are many types of memory elements. Here, we will only focus on standard flip-flops

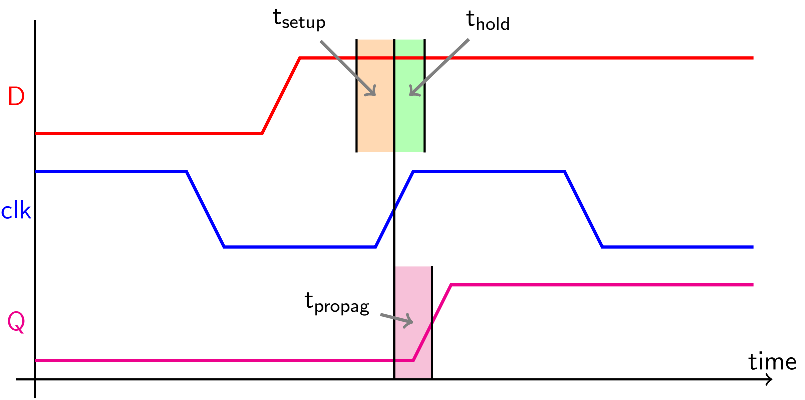



| CLK | D | Q(t+1) | QN(t+1) |
|---|---|---|---|
| 1 | X | Q(t) | QN(t) |
| 0 | X | Q(t) | QN(t) |
| ↑ | 0 | 0 | 1 |
| ↑ | 1 | 1 | 0 |

Remark:
↑ is the rising clock edge



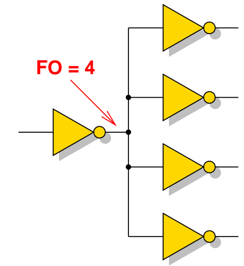
# Setup, Hold and Propagation Delays

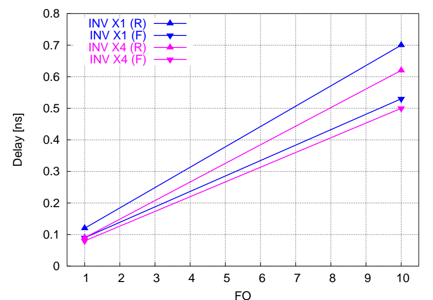


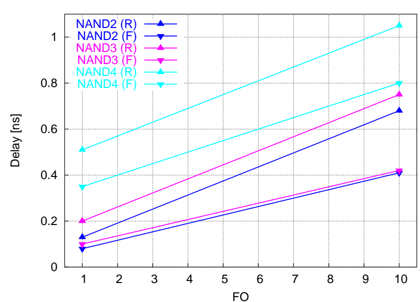

- **setup** delay ($t_{setup}$): data should be held steady *before* clock edge
- **hold** delay ($t_{hold}$): data should be held steady *after* clock edge
- **propagation** delay ($t_{propag}$): propagation time from D to Q

# Fanout ( [FR] *sortance* )

The gate delay (change output state) depends on the output load. Fanout measures this load as the number of inputs of gate connected to the output (normalized w.r.t. an inverter)



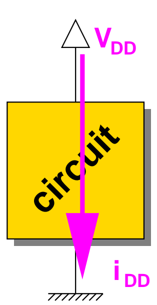



# Power Consumption: Basic Definitions

Instantaneous power:

$$P(t) = i_{DD}(t)\, V_{DD}$$




Energy over some time interval T:

$$E = \int_0^T i_{DD}(t)\, V_{DD}\, dt$$

Average power over interval T:

$$P_{avg} = \frac{E}{T} = \frac{1}{T} \int_0^T i_{DD}(t)\, V_{DD}\, dt$$

Units:

- current A
- voltage V
- power W
- energy J or Wh

# Power Consumption: Components

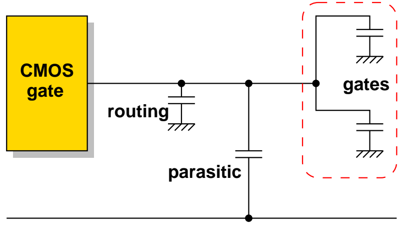
Power dissipation in CMOS circuits comes from 2 main components:

- static dissipation:
  - sub-threshold conduction through OFF transistors
  - leakage current through P-N junctions
  - tunneling current through gate oxide
  - ...
- dynamic dissipation:
  - charging and discharging of load capacitances (useful + parasitic)
  - short-circuit current

$$P_{\text{total}} = P_{\text{static}} + P_{\text{dynamic}}$$

# Charging and Discharging Load Capacitances

There are capacitances everywhere in the circuit: transistor gate, routing, parasitics. . .




Solutions:

- design small circuits (small transistor, short wires, technology shrinking)
- reduce the activity (algorithms, data coding, sleep mode)
- reduce $V_{DD}$ (without lowering speed)

# Simple Power Consumption Model

Average dynamic power dissipation (no leakage, no short circuit):

$$P = \alpha \times C \times f \times V_{DD}^2$$

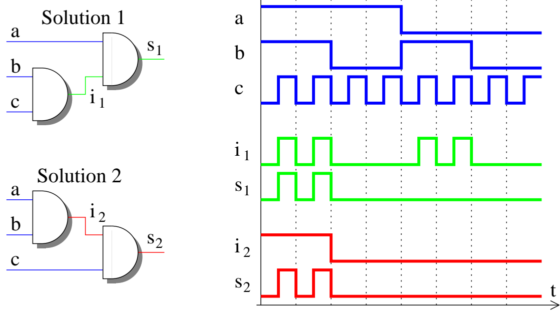
where

- $\alpha$ is the activity factor
- $C$ is the average switched capacitance (at each cycle)
- $f$ is the circuit frequency
- $V_{DD}$ is the supply voltage

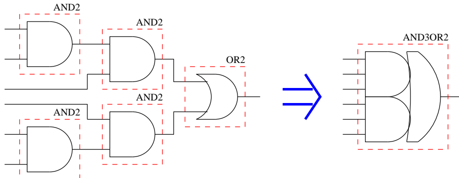
Remark: the gate delay is $d = \gamma \times \frac{C \times V_{DD}}{(V_{DD} - V_T)^2} \approx \frac{1}{V_{DD}}$

# Power Reduction at Gate Level

- gate and/or input reordering (reduce glitching power):




- use complex gates (reduce internal capacitances and area):



# Addition & Multiplication

# Positional Number System(s)

$$X = \sum_{i=-m}^{n-1} x_i \beta^i = (x_{n-1}x_{n-2}\cdots x_1x_0 \,.\, x_{-1}x_{-2}\cdots x_{-m})$$

- radix $\beta$ (usually a power of 2)
- digits $x_i$ ($\in \mathbb{N}$) in the digit set $\mathcal{D}$
- rank or position $i$, weight $\beta^i$
- $n$ integer digits, $m$ fractional digits

Examples:

- $\beta = 10, \mathcal{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\beta = 2, \mathcal{D} = \{0, 1\}$

# Positional Number System(s)

$$X = \sum_{i=-m}^{n-1} x_i \beta^i = (x_{n-1}x_{n-2}\cdots x_1x_0 \,.\, x_{-1}x_{-2}\cdots x_{-m})$$

- radix $\beta$ (usually a power of 2)
- digits $x_i$ ($\in \mathbb{N}$) in the digit set $\mathcal{D}$
- rank or position $i$, weight $\beta^i$
- $n$ integer digits, $m$ fractional digits

Examples:

- $\beta = 10, \mathcal{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\beta = 2, \mathcal{D} = \{0, 1\}$
- carry save: $\beta = 2, \mathcal{D}_{\mathrm{cs}} = \{0, 1, 2\}$
- borrow save: $\beta = 2, \mathcal{D}_{\mathrm{bs}} = \{-1, 0, 1\}$

# Positional Number System(s)

$$X = \sum_{i=-m}^{n-1} x_i \beta^i = (x_{n-1}x_{n-2}\cdots x_1x_0 \,.\, x_{-1}x_{-2}\cdots x_{-m})$$

- radix $\beta$ (usually a power of 2)
- digits $x_i$ ($\in \mathbb{N}$) in the digit set $\mathcal{D}$
- rank or position $i$, weight $\beta^i$
- $n$ integer digits, $m$ fractional digits

Examples:

- $\beta = 10, \mathcal{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\beta = 2, \mathcal{D} = \{0, 1\}$
- carry save: $\beta = 2, \mathcal{D}_{\mathrm{cs}} = \{0, 1, 2\}$
- borrow save: $\beta = 2, \mathcal{D}_{\mathrm{bs}} = \{-1, 0, 1\}$
- signed digits: $\beta > 2, \mathcal{D}_{\mathrm{sd},\alpha,\beta} = \{-\alpha, \ldots, \alpha\}$ with $2\alpha + 1 \geq \beta$

# Positional Number System(s)

$$X = \sum_{i=-m}^{n-1} x_i \beta^i = (x_{n-1}x_{n-2}\cdots x_1 x_0 \,.\, x_{-1}x_{-2}\cdots x_{-m})$$

- radix $\beta$ (usually a power of 2)
- digits $x_i$ ($\in \mathbb{N}$) in the digit set $\mathcal{D}$
- rank or position $i$, weight $\beta^i$
- $n$ integer digits, $m$ fractional digits

Examples:

- $\beta = 10, \mathcal{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\beta = 2, \mathcal{D} = \{0, 1\}$
- carry save: $\beta = 2, \mathcal{D}_{\mathrm{cs}} = \{0, 1, 2\}$
- borrow save: $\beta = 2, \mathcal{D}_{\mathrm{bs}} = \{-1, 0, 1\}$
- signed digits: $\beta > 2, \mathcal{D}_{\mathrm{sd},\alpha,\beta} = \{-\alpha, \ldots, \alpha\}$ with $2\alpha + 1 \geq \beta$
- theoretical systems: $\beta = \frac{1+\sqrt{5}}{2}$, $\beta = 1 + i \ldots$

# Radix-2 Signed Integers

- sign and magnitude (absolute value)

$$A = (s_a a_{n-2} \dots a_1 a_0) = (-1)^{s_a} \times \sum_{i=0}^{n-2} a_i 2^i$$

- 2's complement

$$A = (a_{n-1} a_{n-2} \dots a_1 a_0) = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

- biased (usually $B = 2^{n-1} - 1$)

$$A = A_{math} + B$$

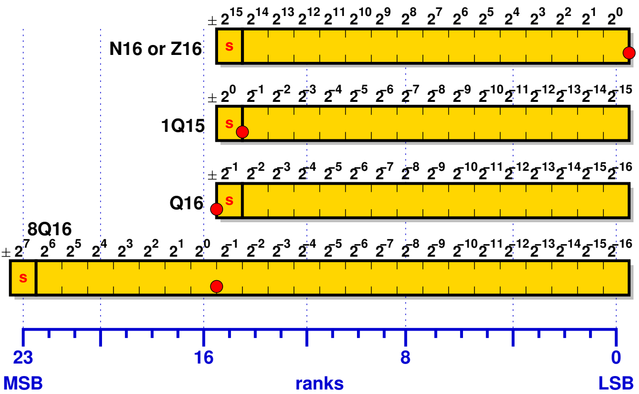
- ...

# Signed Integers

| integer | representations | | |
|---|---|---|---|
| | sign/magnitude | 2's complement | biased (B=7) |
| –8 | — | 1000 | — |
| –7 | 1111 | 1001 | 0000 |
| –6 | 1110 | 1010 | 0001 |
| –5 | 1101 | 1011 | 0010 |
| –4 | 1100 | 1100 | 0011 |
| –3 | 1011 | 1101 | 0100 |
| –2 | 1010 | 1110 | 0101 |
| –1 | 1001 | 1111 | 0110 |
| 0 | 0000 | 0000 | 0111 |
| 1 | 0001 | 0001 | 1000 |
| 2 | 0010 | 0010 | 1001 |
| 3 | 0011 | 0011 | 1010 |
| 4 | 0100 | 0100 | 1011 |
| 5 | 0101 | 0101 | 1100 |
| 6 | 0110 | 0110 | 1101 |
| 7 | 0111 | 0111 | 1110 |
| 8 | — | — | 1111 |

# Fixed-Point Representations

Widely used in DSPs and digital integrated circuits for higher speed, lower silicon area and power consumption compared to floating point




Typical fixed-point formats: 16, 24, 32 and 48 bits

# Floating-Point Representation(s)

Radix-$\beta$ floating-point representation of $x$:

- sign $s_x$, 1-bit encoding: $0 \Rightarrow x > 0$ and $1 \Rightarrow x < 0$
- exponent $e_x \in \mathbb{N}$ on $k$ digits and $e_{min} \leq e_x \leq e_{max}$
- mantissa $m_x$ on $n+1$ digits
- encoding:

$$x = (-1)^{s_x} \times m_x \times \beta^{e_x}$$

$$m_x = x_0 \,.\, x_1\, x_2\, x_3\, \cdots\, x_n$$

$$x_i \in \{0, 1, \ldots, \beta - 1\}$$

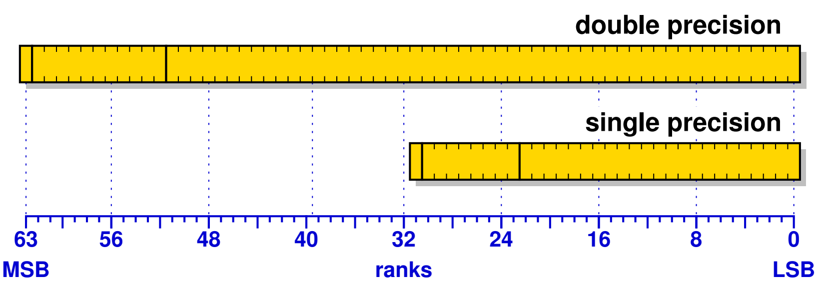
For accuracy purpose, the mantissa must be normalized $(x_0 \neq 0)$

Then $m_x \in [1, \beta[$ and a specific encoding is required for the number 0

# IEEE-754: basic formats

Radix $\beta = 2$, the first bit of the normalized mantissa is always a "1" (non-stored implicit bit)

| format | number of bits | | | |
|---|---|---|---|---|
| | total | sign | exponent | mantissa |
| double precision | 64 | 1 | 11 | $52 + 1$ |
| simple precision | 32 | 1 | 8 | $23 + 1$ |



# IEEE-754: Exponent and Special Values

| format | size | bias | unbiased | | biased | |
|---|---|---|---|---|---|---|
| | $k$ | $b$ | $e_{min}$ | $e_{max}$ | $e_{min}$ | $e_{max}$ |
| SP | 8 | 127 $(= 2^{8-1} - 1)$ | $-126$ | 127 | 1 | 254 |
| DP | 11 | 1023 $(= 2^{11-1} - 1)$ | $-1022$ | 1023 | 1 | 2046 |

| | |
|---|---|
| $-0$ | 1 00000000 00000000000000000000000 |
| $+0$ | 0 00000000 00000000000000000000000 |
| $-\infty$ | 1 11111111 00000000000000000000000 |
| $+\infty$ | 0 11111111 00000000000000000000000 |
| NaN | 0 11111111 00000000000000000000001 (for instance) |

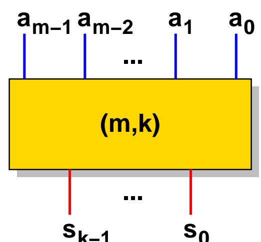
Not a Number (NaN) is the result of invalid operations such as $0/0$, $\sqrt{-1}$ or $0 \times \infty$

# Basic Cells for Addition

Useful circuit element in computer arithmetic: counter

A $(m, k)$-counter is a cell that counts the number of 1 on its $m$ inputs (result expressed as a $k$-bit integer)

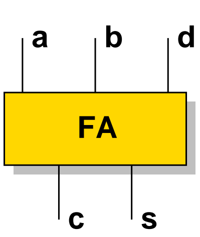
$$\sum_{i=0}^{m-1} a_i = \sum_{j=0}^{k-1} s_j 2^j$$




Standard counters:

- half-adder or HA is a (2,2)-counter
- full-adder or FA is a (3,2)-counter

# FA Cell



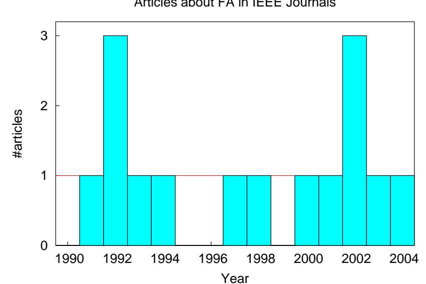

| $a$ | $b$ | $d$ | $c$ | $s$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Arithmetic equation:

$$2c + s = a + b + d$$

Logic equation:

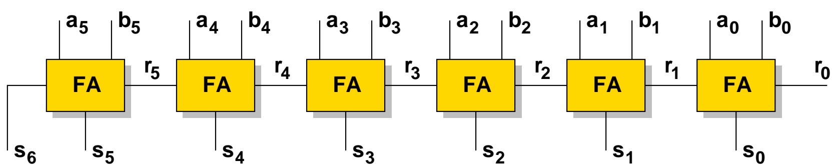
$$\begin{aligned} s &= a \oplus b \oplus d \\ c &= ab + ad + bd \end{aligned}$$




There many implementations of the FA cell

# Carry Ripple Adder (CRA)

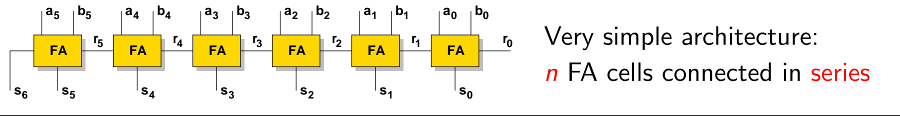
Very simple architecture: $n$ FA cells connected in series




| | complexity |
|---|---|
| delay | $O(n)$ |
| area | $O(n)$ |

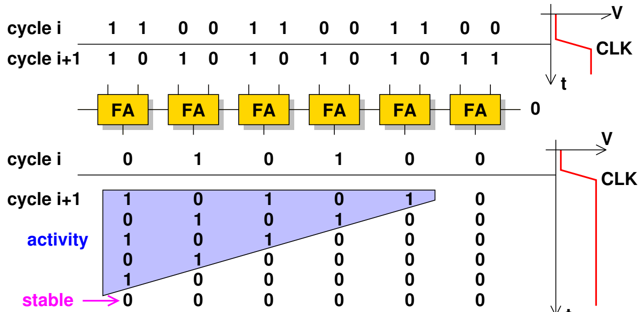
Warning: Sometimes a CRA is also called *Carry Propagate Adder* (CPA), but CPA also means a non-redundant adder (that propagates)

# Useless Activity in a Carry Ripple Adder




Very simple architecture:

*n* FA cells connected in series

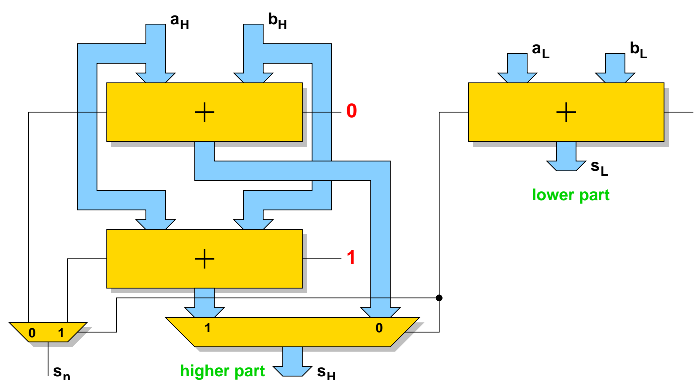



Theoretical models (equiprobable and uniform distribution of inputs):

- worst case $n^2/2$ transitions
- average $3n/2$ transitions and only $n/2$ useful

# Carry-Select Adder

Idea: computation of the higher half part for the 2 possible input carries (0 and 1) and selection when the output carry from lower half part is known




Recursive version $\longrightarrow$ $O(\log n)$ delay    but there is a fanout problem...

# Carry Lookahead Adder

Idea: compute all carries as fast as possible (instead of propagating them)

At rank $i$, the input carry $c_i$ is 1 in the following cases:

- rank $i-1$ generates a carry
  $\hookrightarrow g_{i-1} = 1$
- rank $i-1$ propagates a carry generated at rank $i-2$
  $\hookrightarrow p_{i-1} = g_{i-2} = 1$
- ranks $i-1$ and $i-2$ propagate a carry generated at rank $i-3$
  $\hookrightarrow p_{i-1} = p_{i-2} = g_{i-3} = 1$

$\vdots$

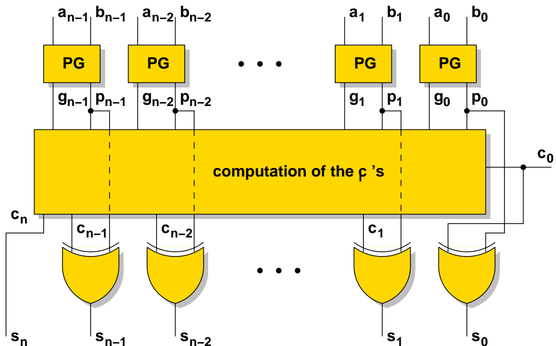
- ranks $i-1$ to $0$ propagate the adder input carry $c_0$ (set to 1)
  $\hookrightarrow p_{i-1} = p_{i-2} = \ldots = p_1 = p_0 = c_0 = 1$

All carries can be computed using the relation ($c_i = g_{i-1} + c_{i-1}p_{i-1}$):

$$c_i = g_{i-1} + p_{i-1}g_{i-2} + p_{i-1}p_{i-2}g_{i-3} + \ldots + p_{i-1}\cdots p_1 g_0 + p_{i-1}\cdots p_0 c_0$$

CLA architecture: parallel evaluation of

- $(g_i, p_i)$ for all $i$
- carries $c_i$ for all $i$ using the above equation
- sums using $s_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i$



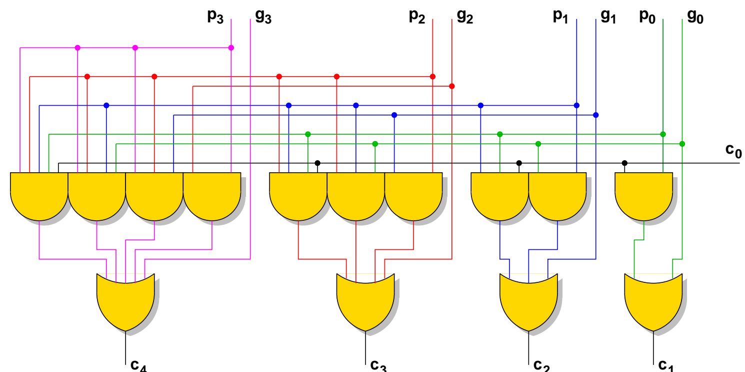
# Carry Lookahead Adder: 4-Bit Example

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

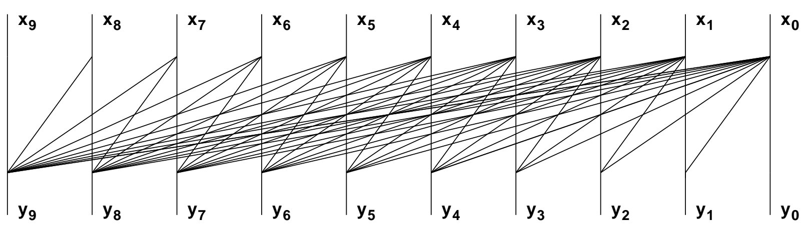
$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$



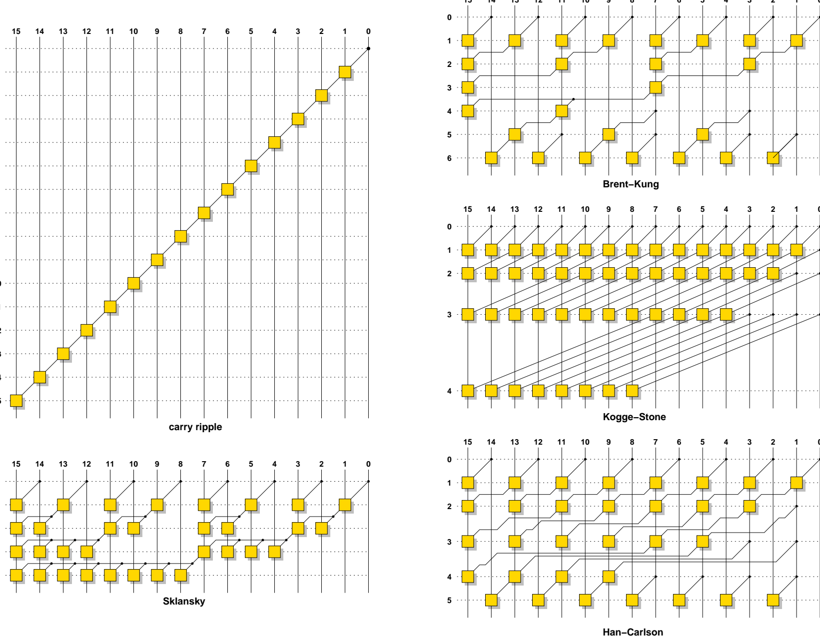
# Parallel-Prefix Problems

The $n$ outputs $(y_{n-1}, y_{n-2}, \cdots, y_0)$ are computed using the $n$ inputs $(x_{n-1}, x_{n-2}, \cdots, x_0)$ and the associative operator $\square$:

$$
\begin{aligned}
y_0 &= x_0 \\
y_1 &= x_1 \square x_0 \\
y_2 &= x_2 \square x_1 \square x_0 \\
&\vdots \\
y_{n-1} &= x_{n-1} \square x_{n-2} \square \cdots \square x_1 \square x_0
\end{aligned}
$$



# Parallel-Prefix Addition: Standard Architectures



# Redundant or Constant Time Adders

To speed-up the addition, one solution consists in "saving" the carries and using them (this makes sense only in case of multiple additions)

In 1961, Avizienis suggested to represent numbers in radix $\beta$ with digits in $\{-\alpha, -\alpha+1, \ldots, 0, \ldots, \alpha-1, \alpha\}$ instead of $\{0, 1, 2, \ldots, \beta-1\}$ with $\alpha \leq \beta - 1$

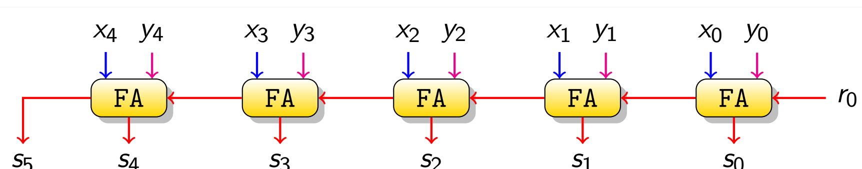
Using this representation, if $2\alpha + 1 > \beta$ some numbers have several possible representation at the bit level. For instance, the value 2345 (in the standard representation) can be represented in radix 10 with digits in $\{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$ by the values 2345, 235(-5) or 24(-5)(-5)

Such a representation is said redundant

In a redundant number system there is constant-time addition algorithm (without carry propagation) where all computations are done in parallel

# Addition

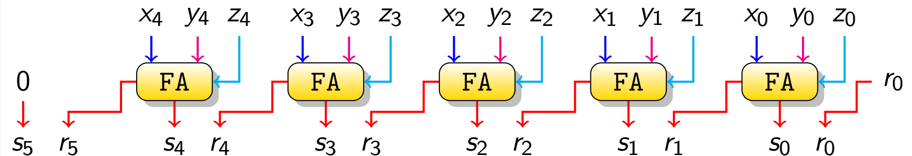
Q: How can we speed up addition?



# Addition

Q: How can we speed up addition?

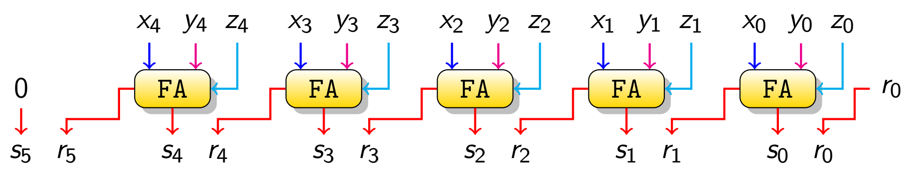
R: Save the carries!



# Addition

Q: How can we speed up addition?

R: Save the carries!




$$X + Y + Z = S + R = \sum_{i=0}^{n} (s_i + r_i)\, 2^i$$

The computation time does not depend on $n$ $T(n) = O(1)$

# Addition using the *carry-save* representation

Q: How can we speed up addition?

R: Save the carries!

$x_4$ $y_4$ $z_4$ $x_3$ $y_3$ $z_3$ $x_2$ $y_2$ $z_2$ $x_1$ $y_1$ $z_1$ $x_0$ $y_0$ $z_0$

0 FA FA FA FA FA $r_0$

$s_5$ $r_5$ $s_4$ $r_4$ $s_3$ $r_3$ $s_2$ $r_2$ $s_1$ $r_1$ $s_0$ $r_0$

$w_5$ $w_4$ $w_3$ $w_2$ $w_1$ $w_0$

$$X + Y + Z = S + R = \sum_{i=0}^{n} (s_i + r_i)\, 2^i$$

$$= \quad W \quad = \sum_{i=0}^{n} w_i\, 2^i \quad \text{avec} \quad w_i = s_i + r_i \in \{0, 1, 2\}$$

The computation time does **not depend** on $n$ $T(n) = O(1)$

# Addition using the *carry-save* representation

Q: How can we speed up addition?

R: Save the carries!



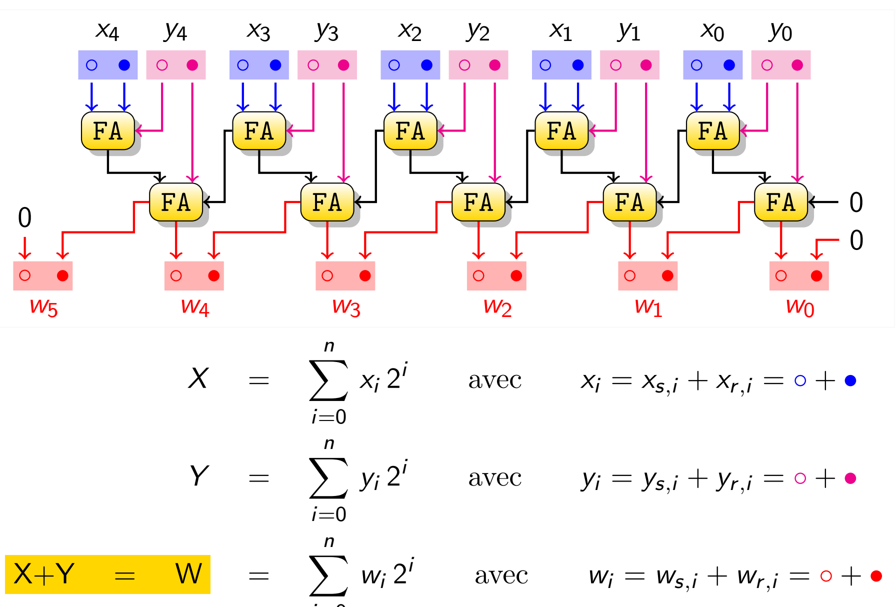
$$X + Y + Z = S + R = \sum_{i=0}^{n} (s_i + r_i)\, 2^i$$

$$= \quad W \quad = \sum_{i=0}^{n} w_i\, 2^i \quad \text{avec} \quad w_i = s_i + r_i \in \{0, 1, 2\}$$

$$= \left( w_n w_{n-1} \dots w_1 w_0 \right)_{\text{cs}} = \left( \begin{array}{|c|}\hline s_n \\ r_n \\ \hline \end{array} \begin{array}{|c|}\hline s_{n-1} \\ r_{n-1} \\ \hline \end{array} \cdots \begin{array}{|c|}\hline s_1 \\ r_1 \\ \hline \end{array} \begin{array}{|c|}\hline s_0 \\ r_0 \\ \hline \end{array} \right)_{\text{cs}}$$

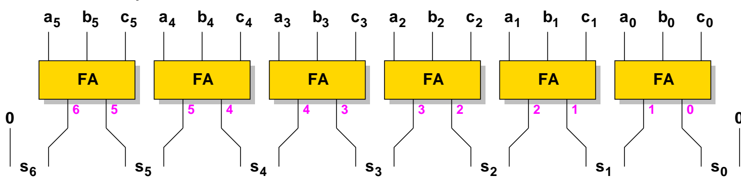
The computation time does not depend on $n$ → $T(n) = O(1)$

# Addition of 2 *Carry-Save* Numbers



# Carry-Save Trees

Example with 3 inputs: $A$, $B$ and $C$




Carry-save reduction tree: $n(h)$ non-redundant inputs can be reduced by a $h$-level carry-save tree where $n(h) = \lfloor 3n(h-1)/2 \rfloor$ and $n(0) = 2$

| $h$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $n(h)$ | 3 | 4 | 6 | 9 | 13 | 19 | 28 | 42 | 63 | 94 | 141 |

# Shift-And-Add Multiplication

The product $P = A \times B$ can be performed using additions and shifts with the following (parallel-serial) algorithm:

```
1  P ⟵ 0
2   for i from 0 to n−1 do
3       P ⟵ P + a_i B 2^i
```

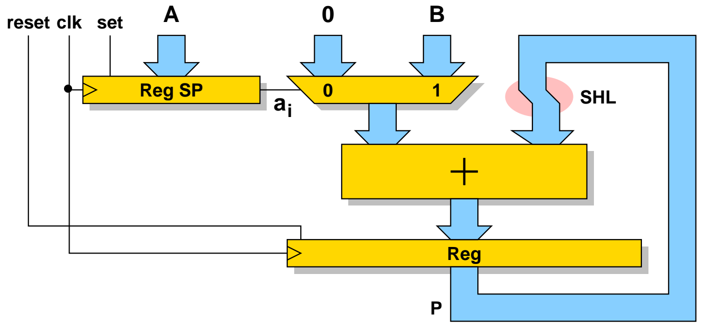
Remark: This algorithm requires a shifter operator (variable shift amount)

Simplification (constant shift):

```
1   P ⟵ 0
2   for i from 0 to n−1 do
3       P ⟵ (P + a_i B) × 2^−1
4   P ⟵ P 2^n
```

Operation on line 4 is virtual

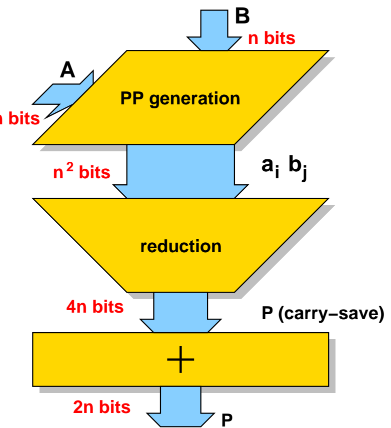
# Shift-And-Add Multiplication: Implementation




| | complexity |
|---|---|
| delay | $O(n)$ |
| area | $O(n)$ |

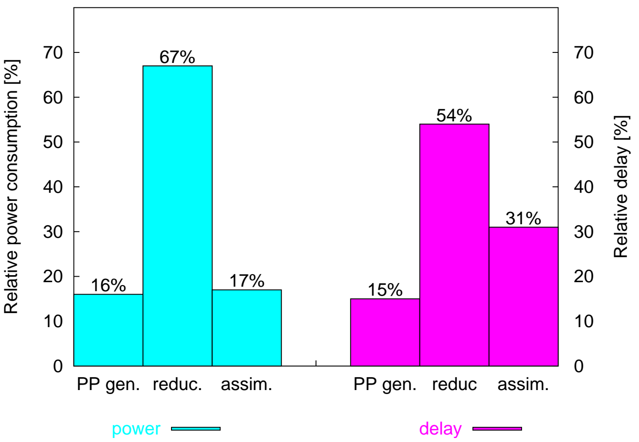
# Fast Multipliers

1. partial products generation $a_i b_j$ (with or without recoding)
   $\hookrightarrow$ delay in $O(1)$ (fanout $a_i$,$b_j$ $O(\log n)$)
2. sum of the partial products using a *carry-save* reduction tree
   $\hookrightarrow$ delay in $O(\log n)$
3. assimilation of the carries using a fast adder
   $\hookrightarrow$ delay in $O(\log n)$




Multiplication delay $O(\log n)$, area $O(n^2)$

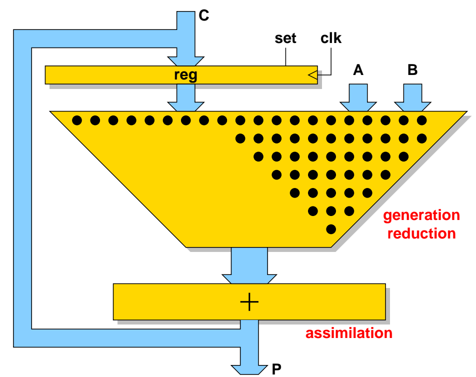
# Power Consumption in Fast Multipliers



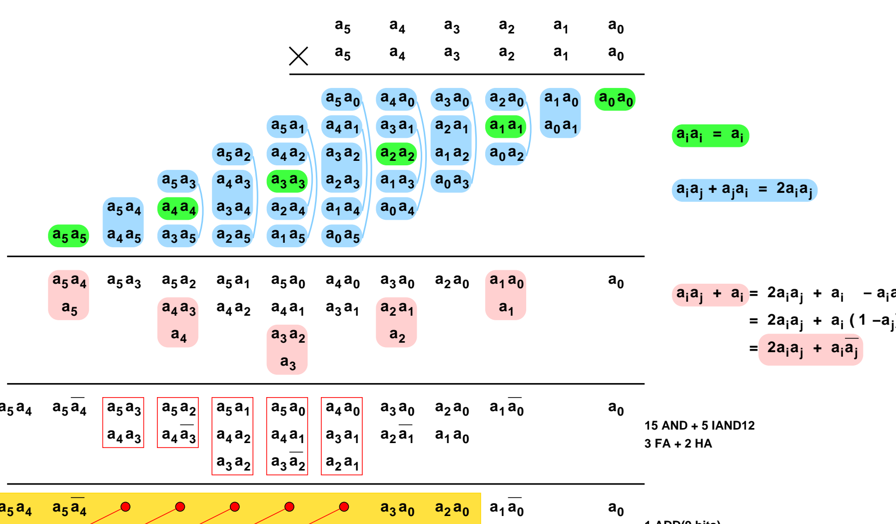

- 30% to 70% of redundant transitions (useless)
- place and route steps based on the internal arrival time
- add a pipeline stage

# MAC and FMA

MAC: multiply and accumulate $P(t) = A \times B + P(t-1)$
$A$, $B$ are $n$-bit values and $P$ a $m$-bit with $m >> n$ (e.g., $16 \times 16 + 40 \longrightarrow 40$ in some DSPs)
FMA: fused multiply and add $P = A \times B + C$ where $A, B, C$ and $P$ can be stored in different registers



# Squarer



# Multiplication by Constants (1/2)

Problem: substitute a complete multiplier by an optimized sequence of shifts and additions and/or subtractions

Example: $p = 111463 \times x$

| algo. | $p = 111463 \times x =$ | #op. |
|---|---|---|
| direct | $(x \ll 16) + (x \ll 15) + (x \ll 13) + (x \ll 12) + (x \ll 9)$ $+ (x \ll 8) + (x \ll 6) + (x \ll 5) + (x \ll 2) + (x \ll 1) + x$ | $10 \pm$ |
| CSD | $(x \ll 17) - (x \ll 14) - (x \ll 12) + (x \ll 10)$ $- (x \ll 7) - (x \ll 5) + (x \ll 3) - x$ | $7 \pm$ |
| Bernstein | $(((t_2 \ll 2) + x) \ll 3) - x$ where $t_1 = (((x \ll 3) - x) \ll 2) - x$ $t_2 = t_1 \ll 7 + t_1$ | $5 \pm$ |
| Our | $(t_2 \ll 12) + (t_2 \ll 5) + t_1$ where $t_1 = (x \ll 3) - x$ $t_2 = (t_1 \ll 2) - x$ | $4 \pm$ |

CSD: canonical signed digit, $111463 = 11011001101100111_2 = 100\bar{1}0\bar{1}0100\bar{1}0\bar{1}0100\bar{1}_2$

# Multiplication by Constants (2/2)

Power savings: 30 up to 60%

| operator | init. | [1] | [2] | our |
|---|---|---|---|---|
| DCT 8b | 300 | 94 | 73 | 56 |
| DCT 12b | 368 | 100 | 84 | 70 |
| DCT 16b | 521 | 129 | 114 | 89 |
| DCT 24b | 789 | 212 | — | 119 |

Power savings: 10%

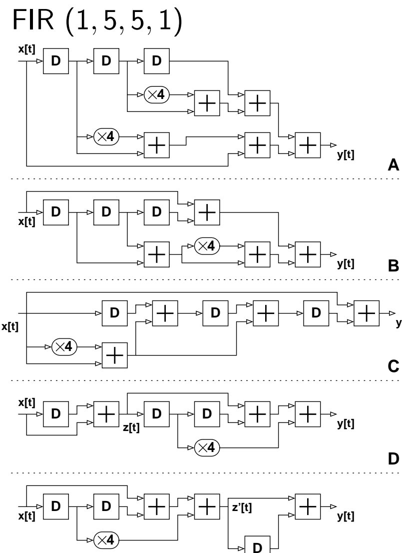
| operator | init. | [1] | [2] | our |
|---|---|---|---|---|
| $8 \times 8$ Had. | 56 | 24 | — | 24 |
| $(16, 11)$ R.-M. | 61 | 43 | 31 | 31 |
| $(15, 7)$ BCH | 72 | 48 | 47 | 44 |
| $(24, 12, 8)$ Golay | 76 | — | 47 | 45 |

Power savings: up to 40%

| operator | init. | [22] | our |
|---|---|---|---|
| 8 bits | 35 | 32 | 24 |
| 16 bits | 72 | 70 | 46 |

Parks-McClellan filter
`remez`$(25, [0\ 0.2\ 0.25\ 1], [1\ 1\ 0\ 0])$.

FIR $(1, 5, 5, 1)$



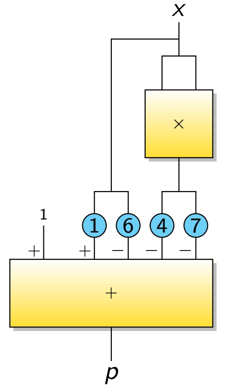
# Example: $\sqrt{x}$ over $[1,2]$ and $\mu \leq 8$ sb

Selection of coefficients leading to sparse recodings

$p^* = 1.00076383 + 0.48388463x - 0.071198745x^2$

$p = 1 + (0.100001\bar{1})_2 x - (0.0001001)_2 x^2$

replace $\times$ by a small number of $\pm$




| solution | area | period | #cycles | latency | power |
|---|---|---|---|---|---|
| wo. tools | 1.00 | 1.00 | 2 | 1.00 | 1.00 |
| w. tools | 0.59 | 0.97 | 1 | 0.48 | 0.45 |

# Modular Exponentiation for RSA

Computation of operations such as : $a^b \bmod n$

$$a^b = \underbrace{a \times a \times a \times a \times \ldots \times a \times a \times a}_{a \text{ appears } b \text{ times}}$$

Order of magnitude of exponents: $2^{\text{size of exponent}} \leadsto 2^{2048} \ldots 2^{4096}$

Fast exponentiation principle:

$$\begin{aligned} a^b = & \quad (a^2)^{\frac{b}{2}} && \text{when } b \text{ is even} \\ = & \quad a \times (a^2)^{\frac{b-1}{2}} && \text{when } b \text{ is odd} \end{aligned}$$

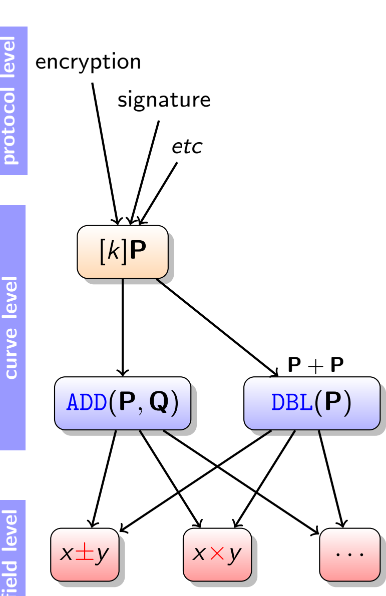
Least significant bit of the exponent: $\texttt{bit} = 0 \leadsto$ even and $\texttt{bit} = 1 \leadsto$ odd

# Square and Multiply Algorithm

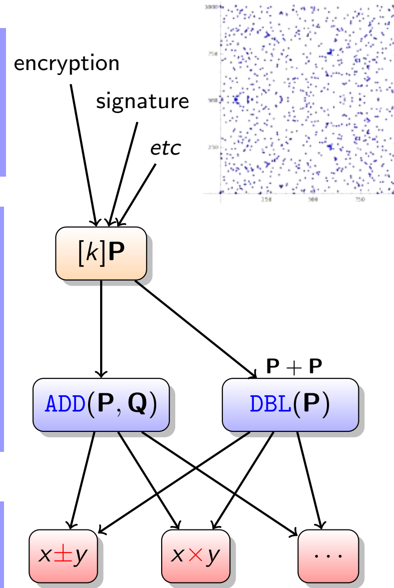
```
input:  a, b, n       where  b = (b_{t-1} b_{t-2} ... b_1 b_0)_2
output:  a^b mod n

r = 1
for i from 0 to t-1 do
    if b_i = 1 then
        r = r · a   mod n
    endif
    a = a^2   mod n
endfor
return r
```

This is the right to left version (there exists a left to right one)

# Elliptic Curve Cryptography in 1 Slide...



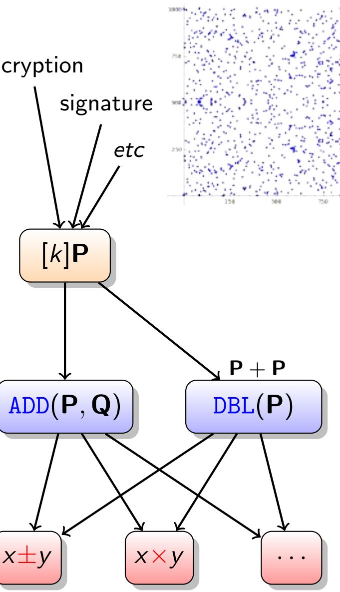
# Elliptic Curve Cryptography in 1 Slide...




$E : y^2 = x^3 + 4x + 20$ over $\mathbb{F}_{1009}$

points: $\mathbf{P}$, $\mathbf{Q} = (x, y)$ or $(x, y, z)$ or ...

# Elliptic Curve Cryptography in 1 Slide...




$E : y^2 = x^3 + 4x + 20$ over $\mathbb{F}_{1009}$

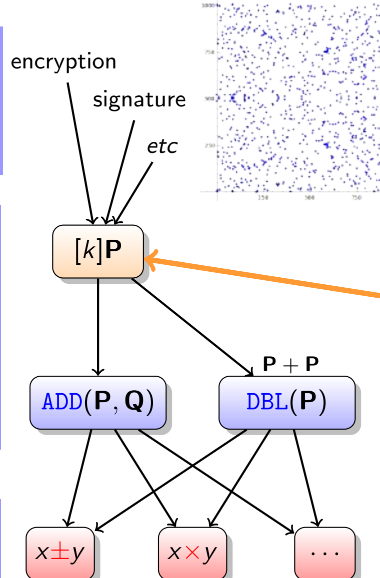
points: $\mathbf{P}$, $\mathbf{Q}$ $= (x, y)$ or $(x, y, z)$ or ...

coordinates: $x, y, z \in \mathbb{F}_q$

$\mathbb{F}_p$, $\mathbb{F}_{2^m}$, $t$ : 200–600 bits

$k = (k_{t-1} k_{t-2} \ldots k_1 k_0)_2 \in \mathbb{N}$

# Elliptic Curve Cryptography in 1 Slide...




$E : y^2 = x^3 + 4x + 20$ over $\mathbb{F}_{1009}$

points: $\mathbf{P}$, $\mathbf{Q} = (x, y)$ or $(x, y, z)$ or ...

coordinates: $x, y, z \in \mathbb{F}_q$

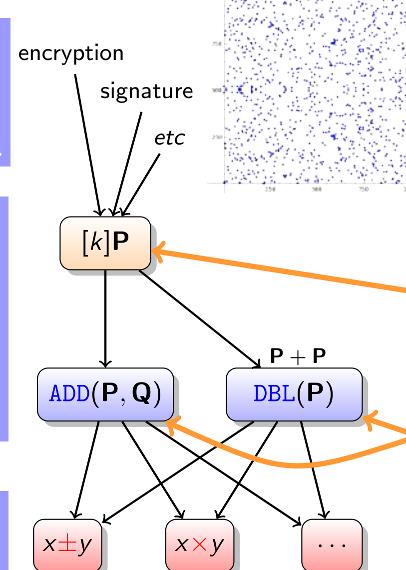
$\mathbb{F}_p$, $\mathbb{F}_{2^m}$, $t$ : 200–600 bits

$k = (k_{t-1}k_{t-2} \dots k_1 k_0)_2 \in \mathbb{N}$

**Scalar multiplication operation**

```
for i from 0 to t-1 do
    if k_i = 1 then Q = ADD(P, Q)
    P = DBL(P)
```

# Elliptic Curve Cryptography in 1 Slide...




$E : y^2 = x^3 + 4x + 20$ over $\mathbb{F}_{1009}$

points: $\mathbf{P}$, $\mathbf{Q} = (x, y)$ or $(x, y, z)$ or ...

coordinates: $x, y, z \in \mathbb{F}_q$

$\mathbb{F}_p$, $\mathbb{F}_{2^m}$, $t$ : 200–600 bits

$k = (k_{t-1}k_{t-2}\dots k_1k_0)_2 \in \mathbb{N}$

**Scalar multiplication operation**

```
for i from 0 to t-1 do
    if k_i = 1 then Q = ADD(P, Q)
    P = DBL(P)
```

**Point addition/doubling operations**

sequence of finite field operations

DBL: $v_1 = z_1^2, v_2 = x_1 - v_1, \dots$

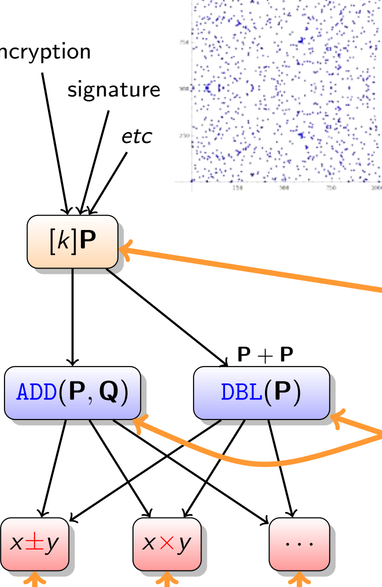
ADD: $w_1 = z_1^2, w_2 = z_1 \times w_1, \dots$

# Elliptic Curve Cryptography in 1 Slide...

protocol level

curve level

field level




$E : y^2 = x^3 + 4x + 20$ over $\mathbb{F}_{1009}$

points: $\mathbf{P}$, $\mathbf{Q} = (x, y)$ or $(x, y, z)$ or ...

coordinates: $x, y, z \in \mathbb{F}_q$

$\mathbb{F}_p$, $\mathbb{F}_{2^m}$, $t$ : 200–600 bits

$k = (k_{t-1}k_{t-2}\dots k_1k_0)_2 \in \mathbb{N}$

**Scalar multiplication operation**

```
for i from 0 to t-1 do
    if k_i = 1 then Q = ADD(P, Q)
    P = DBL(P)
```

**Point addition/doubling operations**

sequence of finite field operations

DBL: $v_1 = z_1^2, v_2 = x_1 - v_1, \dots$

ADD: $w_1 = z_1^2, w_2 = z_1 \times w_1, \dots$

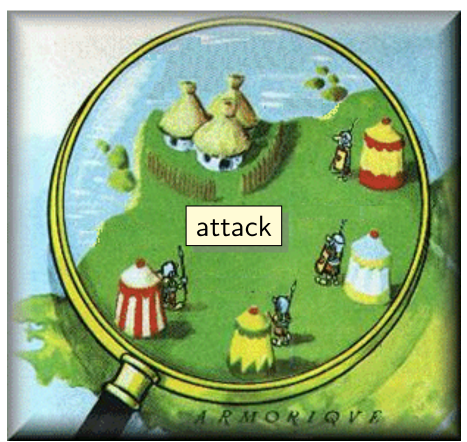
**$\mathbb{F}_p$ or $\mathbb{F}_{2^m}$ operations**

operation modulo large prime ($\mathbb{F}_p$)

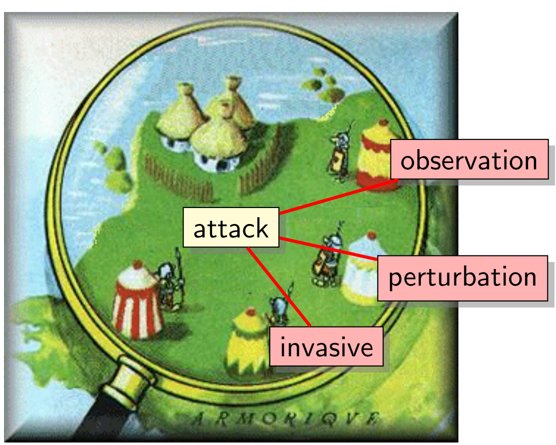
or irreducible polynomial ($\mathbb{F}_{2^m}$)

# Introduction to Physical Attacks

# Attacks



# Attacks



# Attacks

- attack
  - observation
    - timing analysis
    - power analysis
    - EMR analysis
  - perturbation
    - fault injection
  - invasive
    - probing
    - reverse engineering

EMR = Electromagnetic radiation

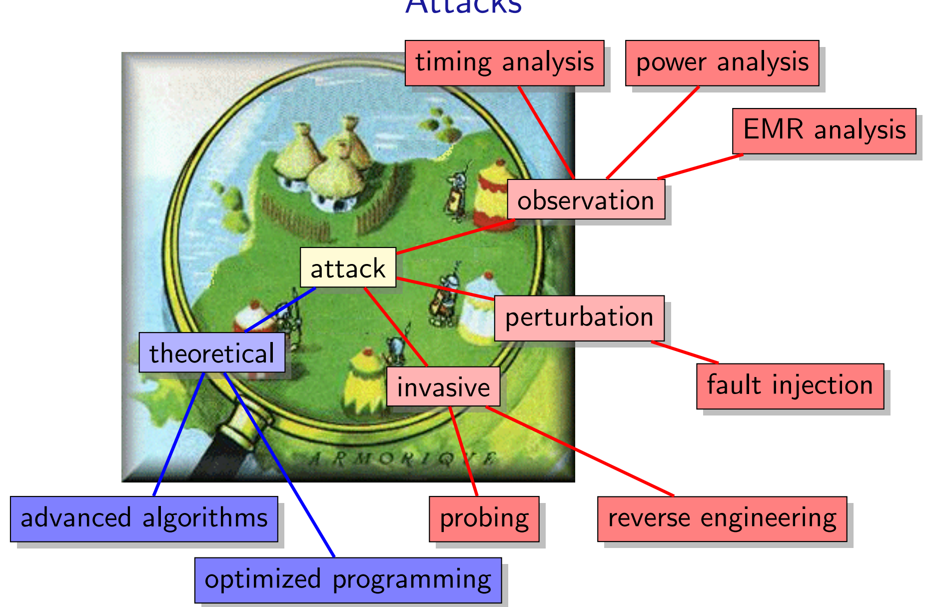
# Attacks

attack
- theoretical
- observation
  - timing analysis
  - power analysis
  - EMR analysis
- perturbation
  - fault injection
- invasive
  - probing
  - reverse engineering

ARMORIQUE

EMR = Electromagnetic radiation

# Attacks




EMR = Electromagnetic radiation

# Side Channel Attacks (SCAs) (1/2)

**Attack**: attempt to find, without any knowledge about the secret:

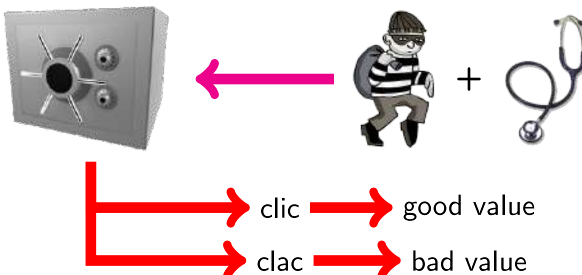
- the message (or parts of the message)
- informations on the message
- the secret (or parts of the secret)

# Side Channel Attacks (SCAs) (1/2)

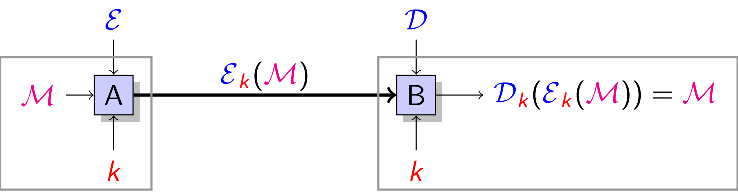
**Attack**: attempt to find, without any knowledge about the secret:

- the message (or parts of the message)
- informations on the message
- the secret (or parts of the secret)

**"Old style" side channel attacks**:

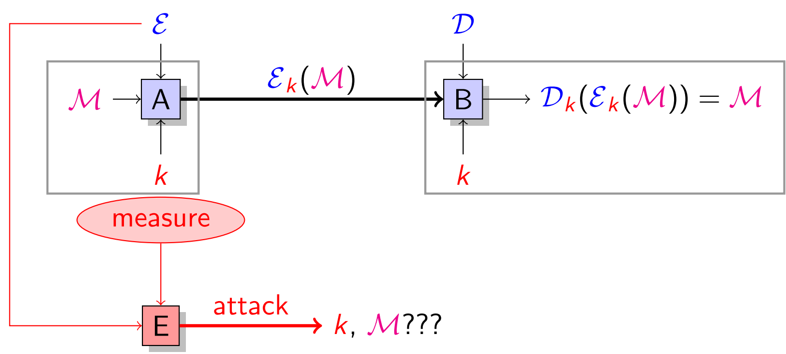


# Side Channel Attacks (2/2)




**General principle:** measure external parameter(s) on running device in order to deduce internal informations

# Side Channel Attacks (2/2)




**General principle:** measure external parameter(s) on running device in order to deduce internal informations

# What Should be Measured?

**Answer**: everything that can "enter" and/or "get out" in/from the device

- power consumption
- electromagnetic radiation
- temperature
- sound
- computation time
- number of cache misses
- number and type of error messages
- ...

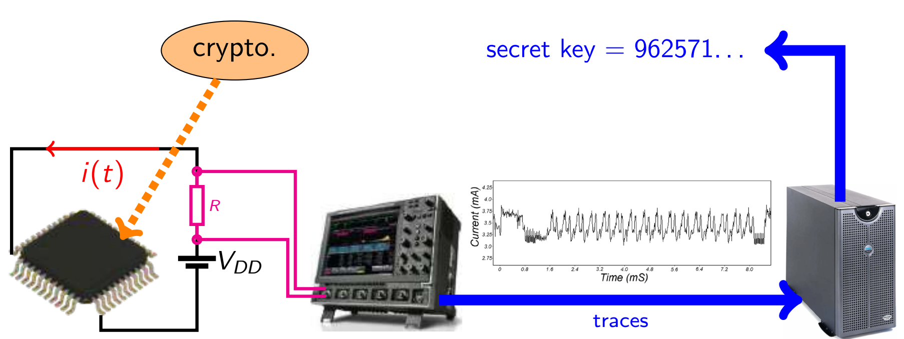
The measured parameters may provide informations on:

- global behavior (temperature, power, sound...)
- local behavior (EMR, # cache misses...)

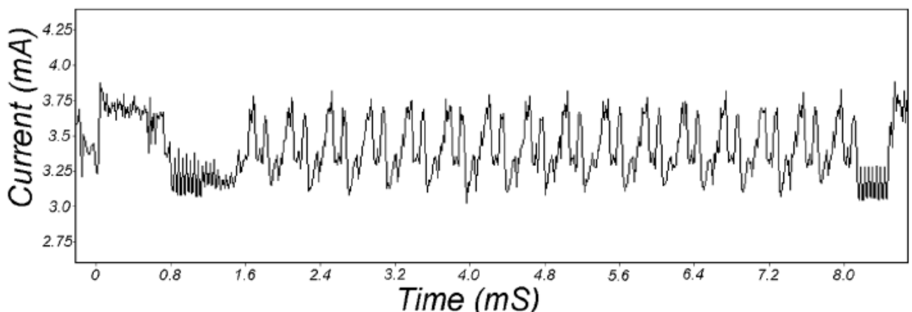
# Power Consumption Analysis

**General principle:**

1. measure the current $i(t)$ in the cryptosystem
2. use those measurements to “deduce” secret informations

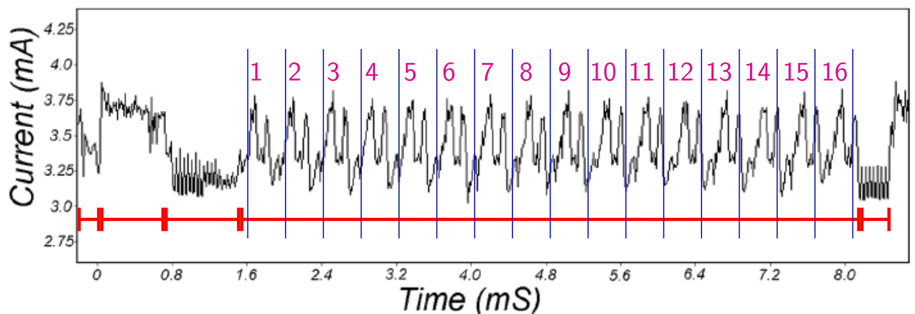


# “Read” the Traces




Source: [8] Kocher, Jaffe and Jun. **Differential Power Analysis**, Crypto99

# “Read” the Traces




- algorithm → decomposition into steps
- detect loops
  - constant time for the loop iterations
  - non-constant time for the loop iterations

Source: [8] Kocher, Jaffe and Jun. **Differential Power Analysis**, Crypto99

# Differences & External Signature

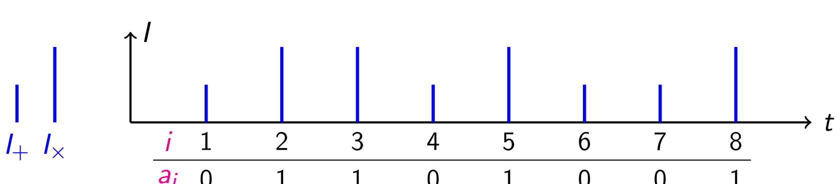
An algorithm :

$r = c_0$
for $i$ from 1 to $n$ do
  if $a_i = 0$ then
    $r = r + c_1$
  else
    $r = r \times c_2$

# Differences & External Signature

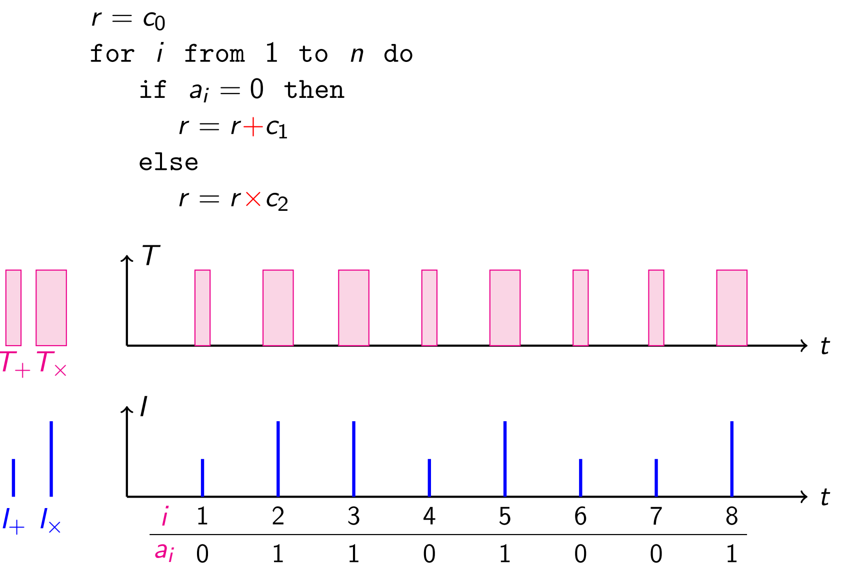
An algorithm has a current signature :

```
r = c_0
for i from 1 to n do
    if a_i = 0 then
        r = r + c_1
    else
        r = r × c_2
```

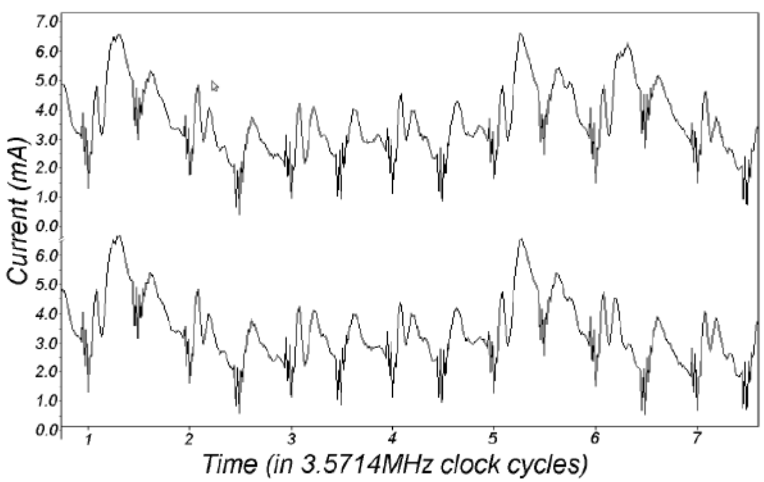


# Differences & External Signature

An algorithm has a current signature and a time signature:

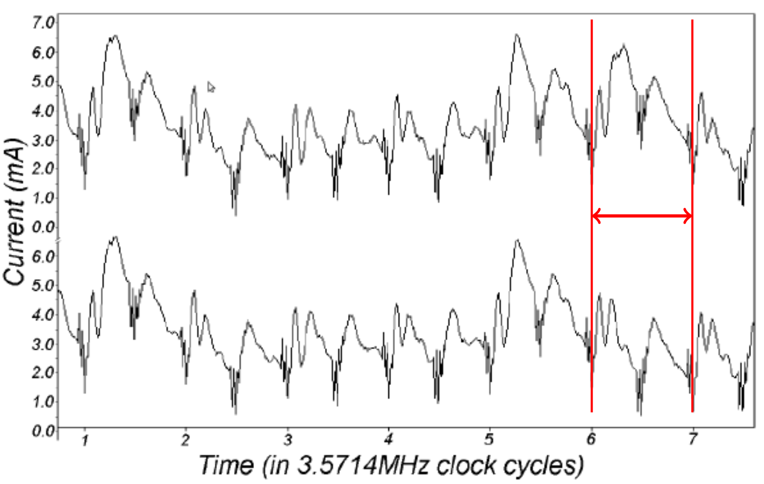


# Simple Power Analysis (SPA)




Source: [8]

# Simple Power Analysis (SPA)




Source: [8]

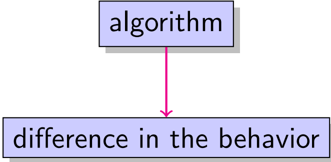
# SPA in Practice

**General principle**:

algorithm

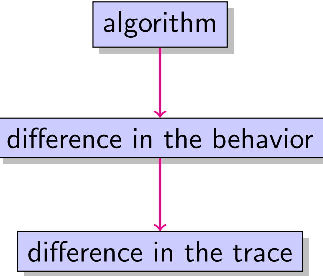
# SPA in Practice

**General principle**:



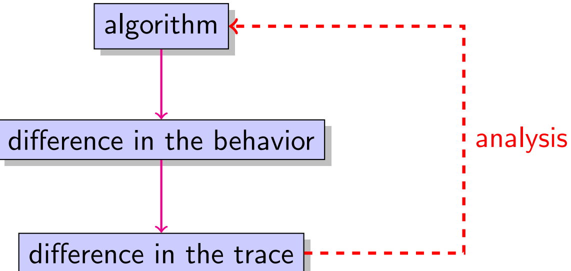
# SPA in Practice

**General principle**:



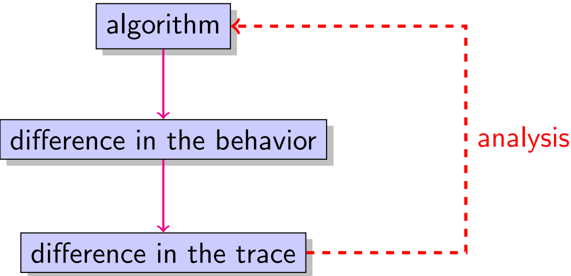
# SPA in Practice

**General principle**:



# SPA in Practice

**General principle**:

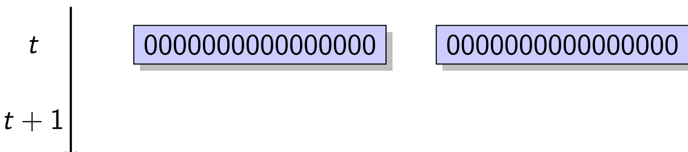



**Methods**: interpretation of the differences in

- control signals
- computation time
- operand values
- ...

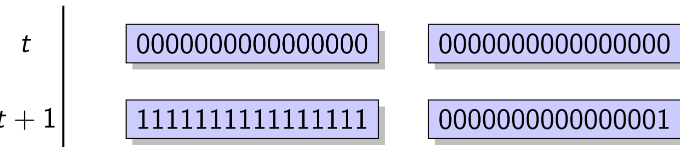
# Limits of the SPA

Example of behavior difference: (activity into a register)



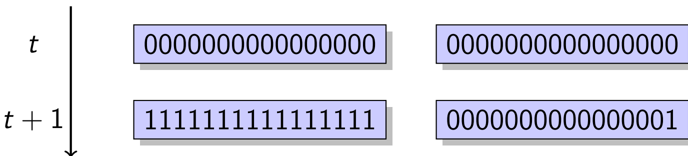
# Limits of the SPA

Example of behavior difference: (activity into a register)



# Limits of the SPA

Example of behavior difference: (activity into a register)




**Important**: a small difference may be evaluated has a noise during the measurement ➔ traces cannot be distinguished

**Question**: what can be done when differences are too small?

# Limits of the SPA

Example of behavior difference: (activity into a register)

| | | |
|---|---|---|
| $t$ | 0000000000000000 | 0000000000000000 |
| $t+1$ | 1111111111111111 | 0000000000000001 |

**Important**: a small difference may be evaluated has a noise during the measurement → traces cannot be distinguished

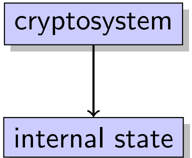
**Question**: what can be done when differences are too small?

**Answer**: use statistics over several traces

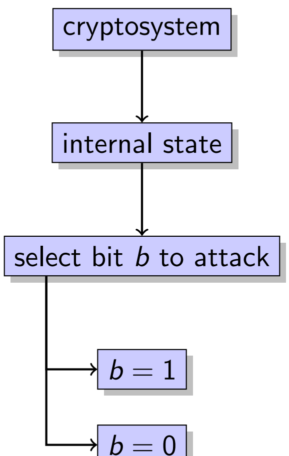
# Differential Power Analysis (DPA)

cryptosystem

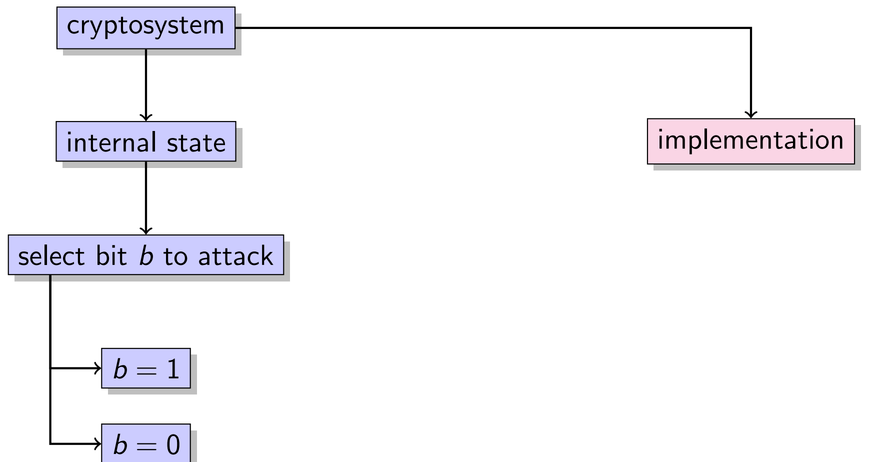
# Differential Power Analysis (DPA)



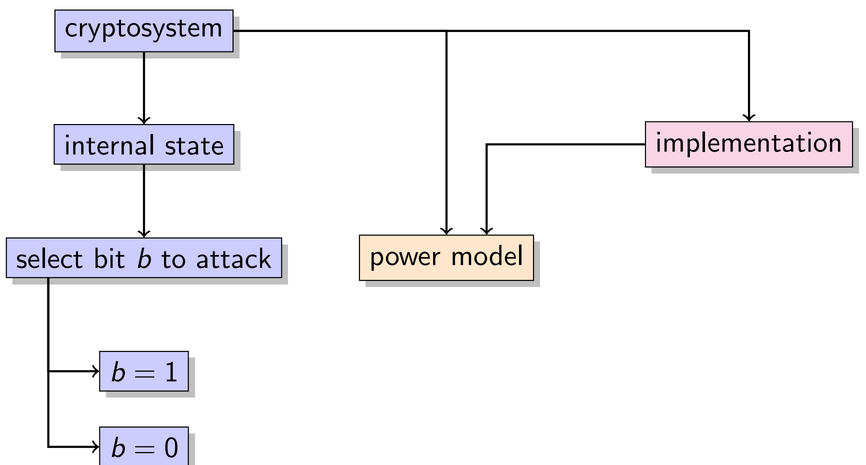
# Differential Power Analysis (DPA)



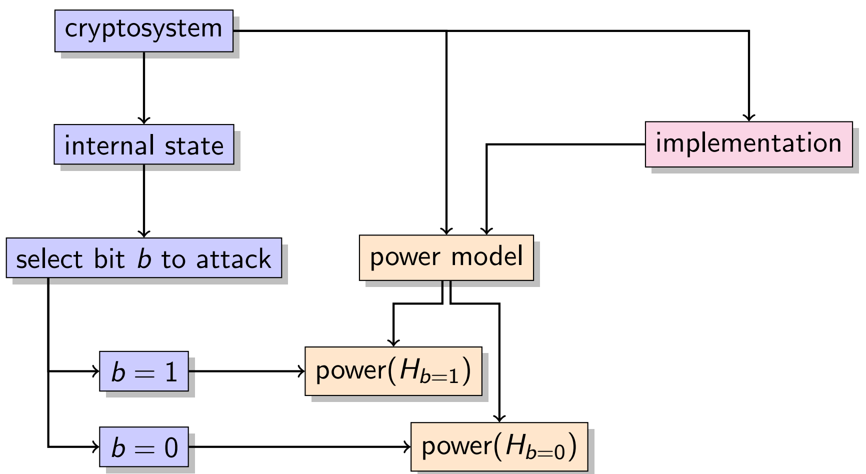
# Differential Power Analysis (DPA)



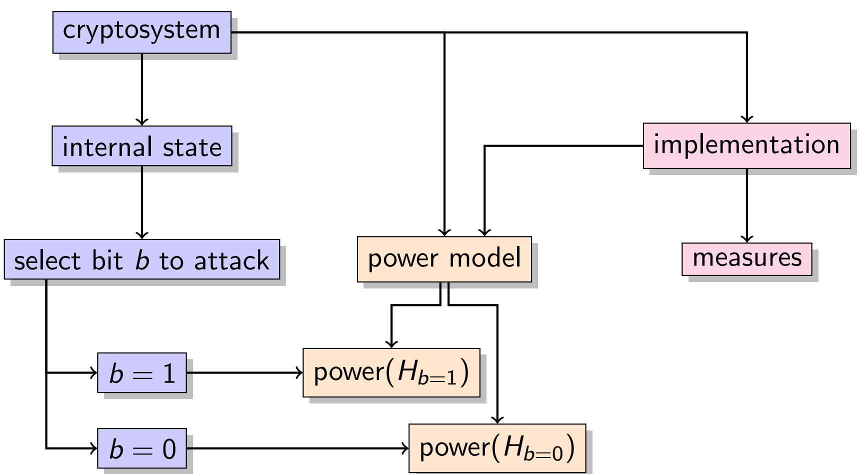
# Differential Power Analysis (DPA)



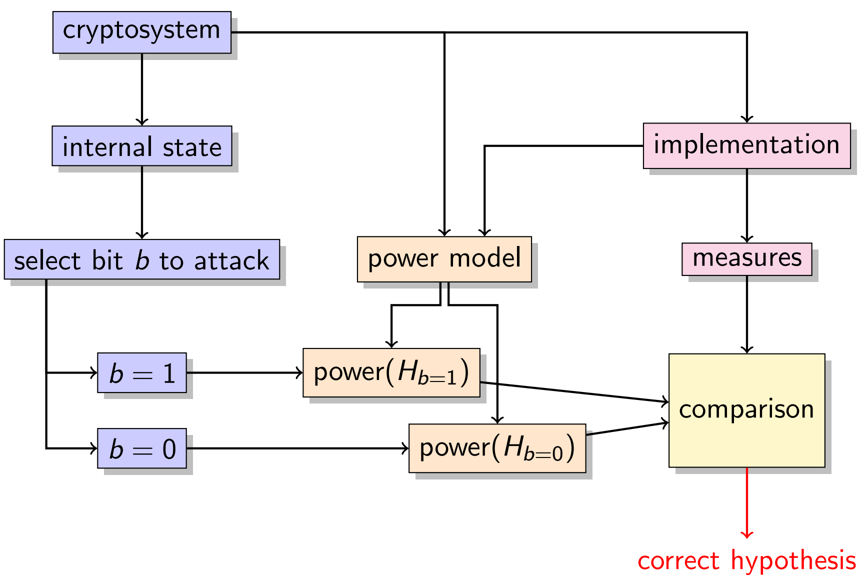
# Differential Power Analysis (DPA)



# Differential Power Analysis (DPA)



# Differential Power Analysis (DPA)



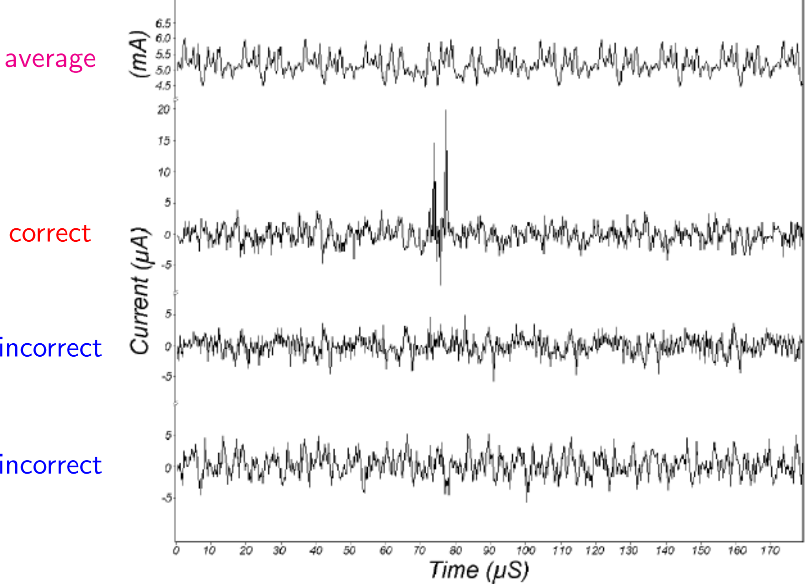
# Differential Power Analysis (DPA) Example

average

correct

incorrect

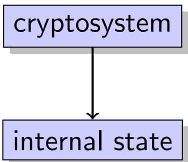
incorrect



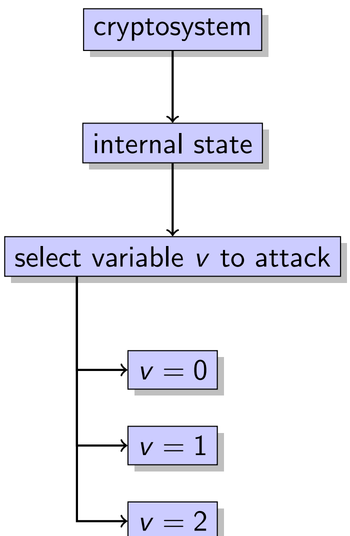
# Template Attack

cryptosystem

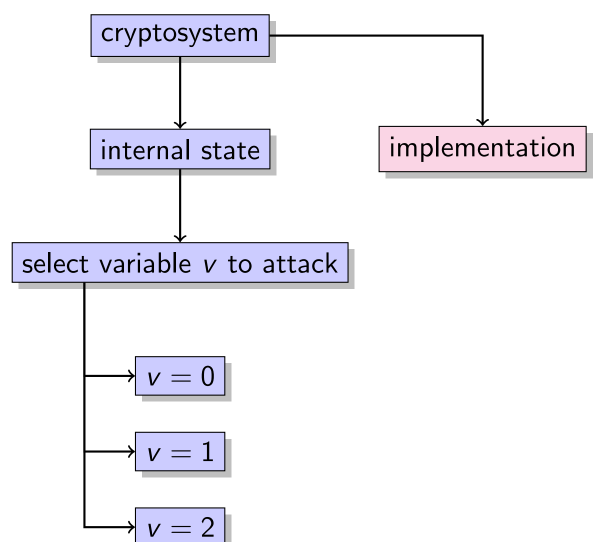
# Template Attack



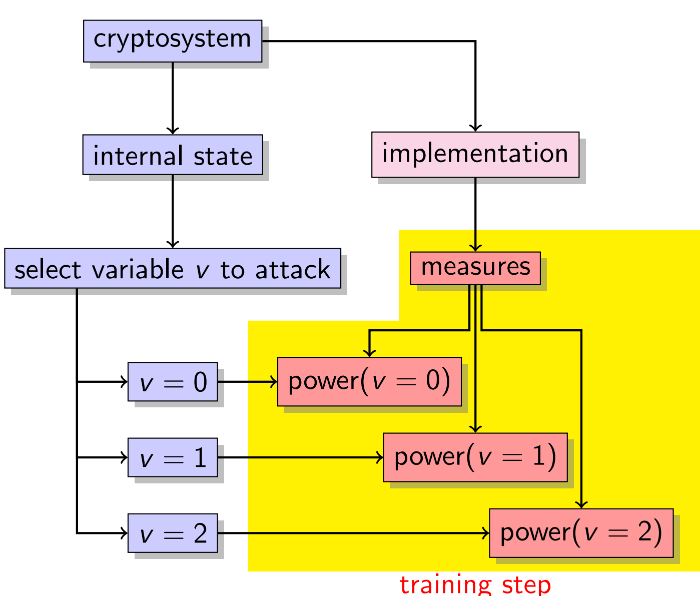
# Template Attack



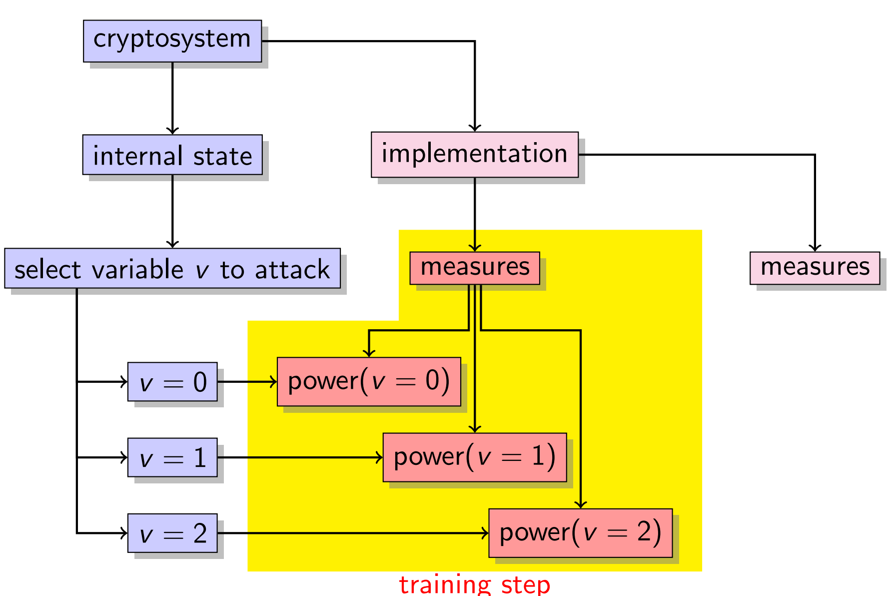
# Template Attack



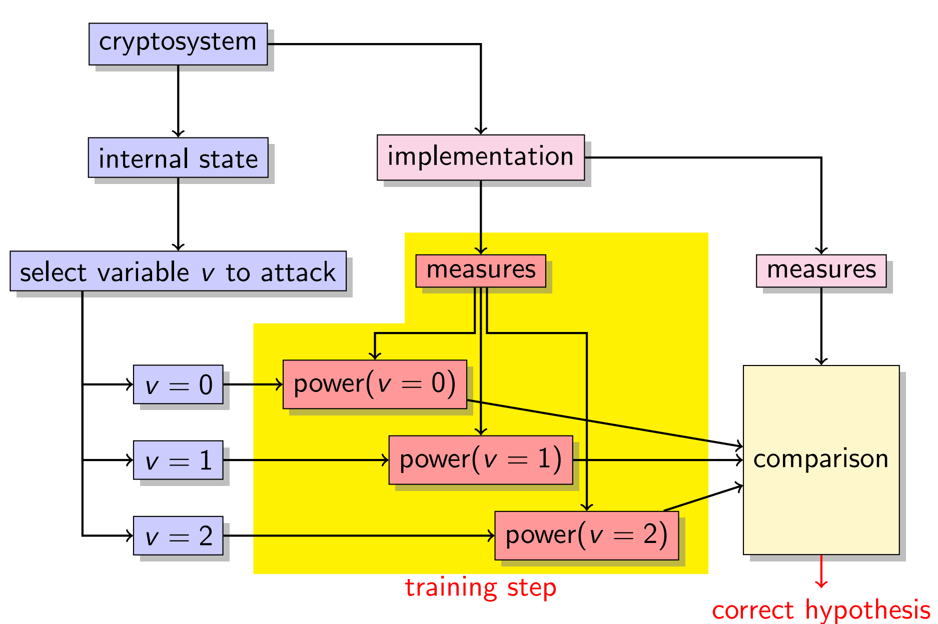
# Template Attack



# Template Attack

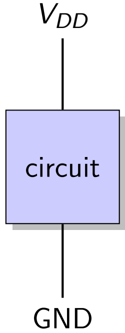


# Template Attack



# Electromagnetic Radiation Analysis (1/2)

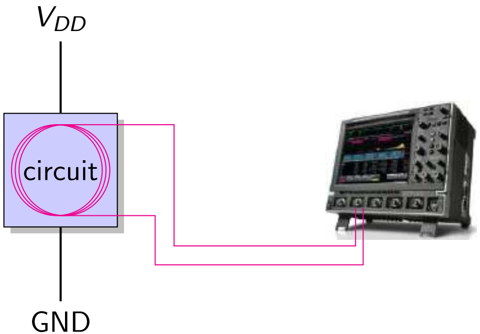
**General principle**: use a probe to measure the EMR




**EMR measurement**:

# Electromagnetic Radiation Analysis (1/2)

**General principle**: use a probe to measure the EMR

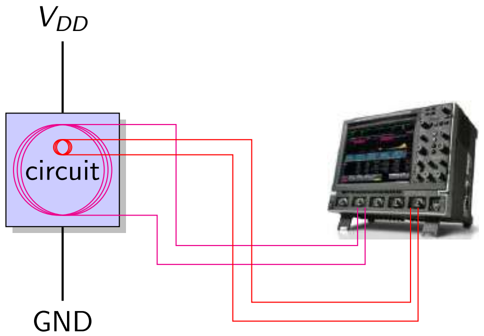



**EMR measurement**:

- global EMR with a large probe

# Electromagnetic Radiation Analysis (1/2)

**General principle**: use a probe to measure the EMR




**EMR measurement**:

- global EMR with a large probe
- local EMR with a micro-probe

# Electromagnetic Radiation Analysis (2/2)

EMR analysis methods:

- simple electromagnetic analysis: SEMA
- differential electromagnetic analysis: DEMA

Local EMR analysis may be used to determine internal architecture details, and then select weak parts of the circuit for the attack

X-Y table

# Side Channel Attack on ECC

protocol level

encryption

signature

*etc*

curve level

$[k]\mathbf{P}$

ADD($\mathbf{P}, \mathbf{Q}$)

DBL($\mathbf{P}$)

field level

$x \pm y$

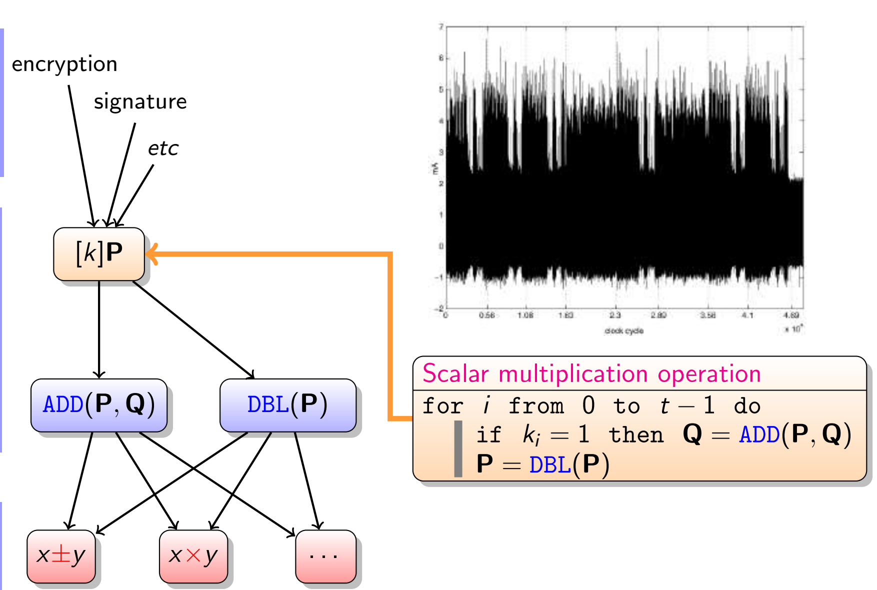
$x \times y$

$\cdots$

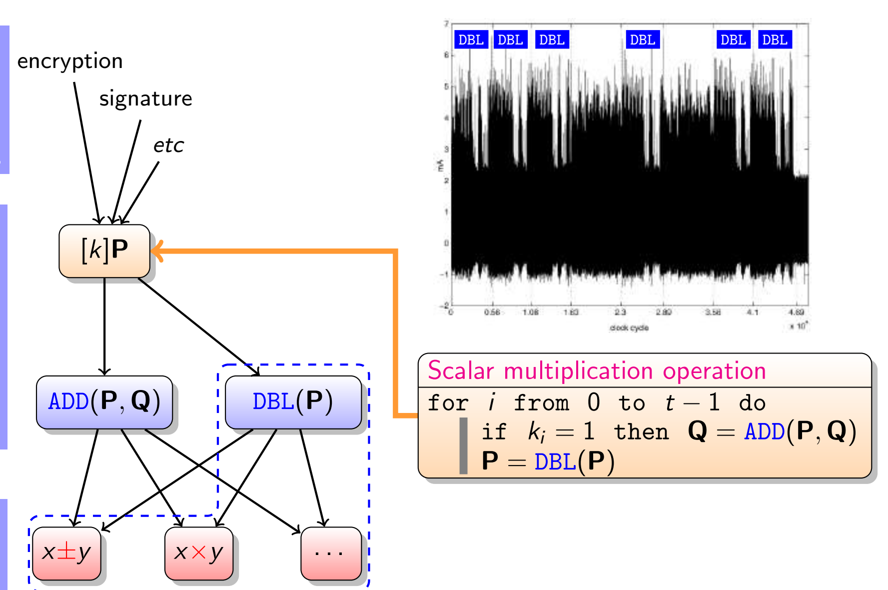
**Scalar multiplication operation**

```
for i from 0 to t-1 do
    if k_i = 1 then Q = ADD(P, Q)
    P = DBL(P)
```

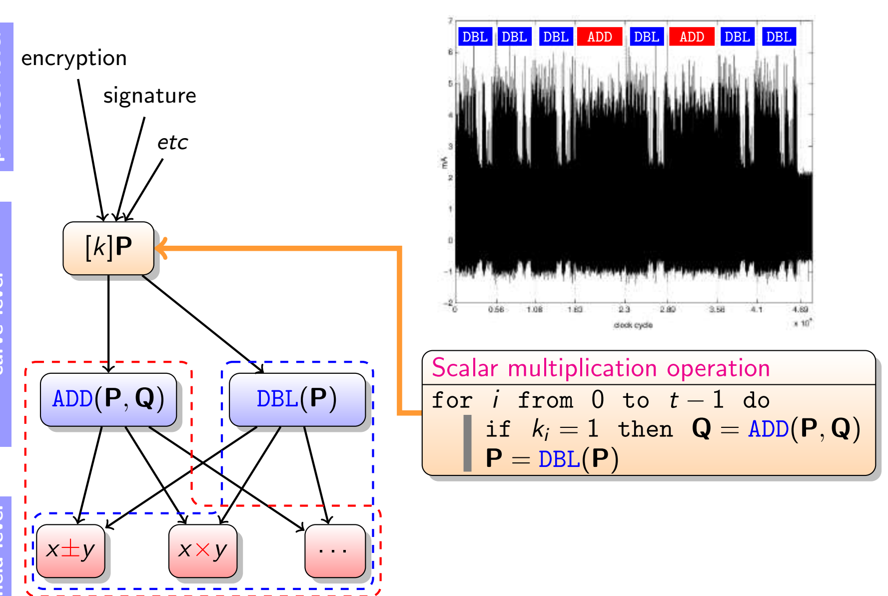
# Side Channel Attack on ECC



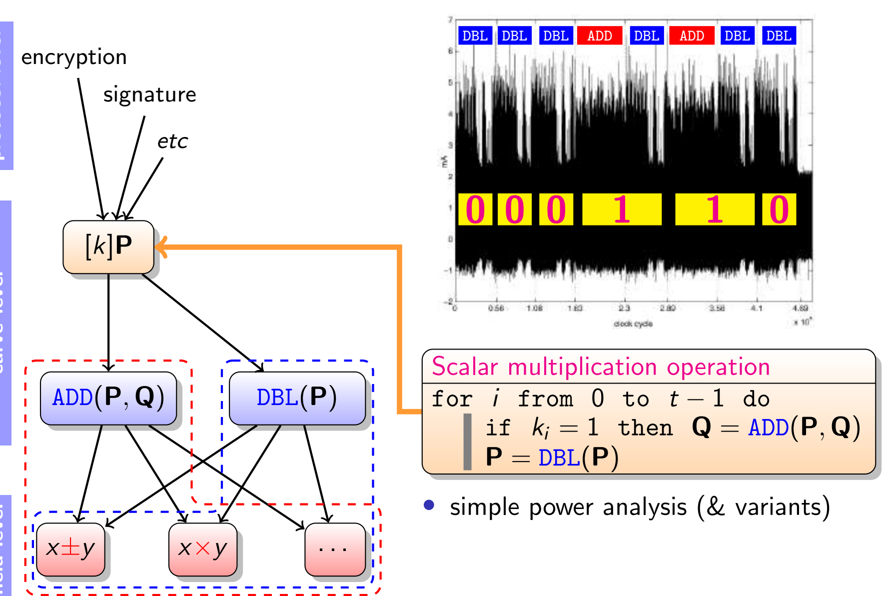
# Side Channel Attack on ECC



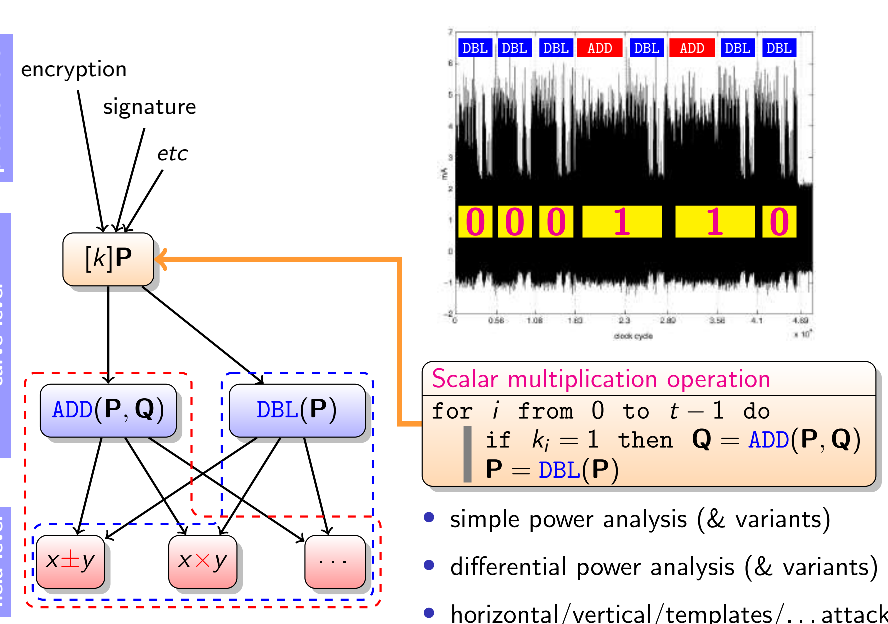
# Side Channel Attack on ECC



# Side Channel Attack on ECC



# Side Channel Attack on ECC



# Protections at the Arithmetic Level

# Countermeasure

**Principles for preventing attacks**:

- embed additional protection blocks
- modify the original circuit into a secured version
- application levels: circuit, architecture, algorithm, protocol. . .

# Countermeasure

**Principles for preventing attacks**:

- embed additional protection blocks
- modify the original circuit into a secured version
- application levels: circuit, architecture, algorithm, protocol...

**Countermeasures**:

- electrical shielding
- detectors, estimators, decoupling
- use uniform computation durations and power consumption
- use detection/correction codes (for fault injection attacks)
- provide a random behavior (algorithms, representation, operations...)
- add noise (e.g. masking, useless instructions/computations)
- circuit reconfiguration (algorithms, block location, representation of values...)

# Low-Level Coding and Circuit Activity

**Assumptions**:

- $b$ is a bit (i.e. $b \in \{0, 1\}$, logical or mathematical value)
- electrical states for a wire ▬▬ : $V_{DD}$ (logical 1) or GND (logical 0)

# Low-Level Coding and Circuit Activity

**Assumptions**:

- $b$ is a bit (i.e. $b \in \{0, 1\}$, logical or mathematical value)
- electrical states for a wire ▬▬ : $V_{DD}$ (logical 1) or GND (logical 0)

**Low-level codings of a bit**:

| | $b = 0$ | $b = 1$ |
|---|---|---|
| standard | ▬▬ GND | ▬▬ $V_{DD}$ |

# Low-Level Coding and Circuit Activity

**Assumptions**:

- $b$ is a bit (i.e. $b \in \{0, 1\}$, logical or mathematical value)
- electrical states for a wire ▬▬ : $V_{DD}$ (logical 1) or GND (logical 0)

**Low-level codings of a bit**:

| | $b = 0$ | $b = 1$ |
|---|---|---|
| standard | ▬▬ GND | ▬▬ $V_{DD}$ |
| dual rail | $r_0 = V_{DD}$, $r_1 = \text{GND}$ ] $(1, 0)_{\text{DR}}$ | $r_0 = \text{GND}$, $r_1 = V_{DD}$ ] $(0, 1)_{\text{DR}}$ |

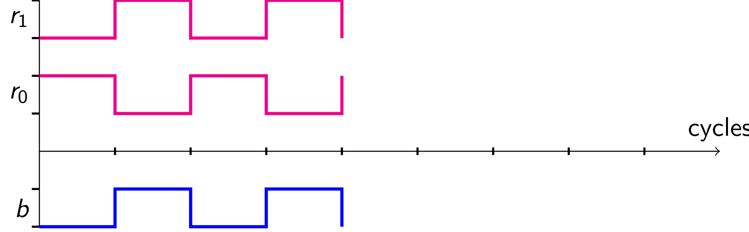
# Low-Level Coding and Circuit Activity

**Assumptions**:

- $b$ is a bit (i.e. $b \in \{0,1\}$, logical or mathematical value)
- electrical states for a wire ▬▬ : $V_{DD}$ (logical 1) or GND (logical 0)

**Low-level codings of a bit**:

| | $b = 0$ | $b = 1$ |
|---|---|---|
| standard | GND | $V_{DD}$ |
| dual rail | $r_0 = V_{DD}$, $r_1 = \mathrm{GND}$ $(1,0)_{\mathrm{DR}}$ | $r_0 = \mathrm{GND}$, $r_1 = V_{DD}$ $(0,1)_{\mathrm{DR}}$ |

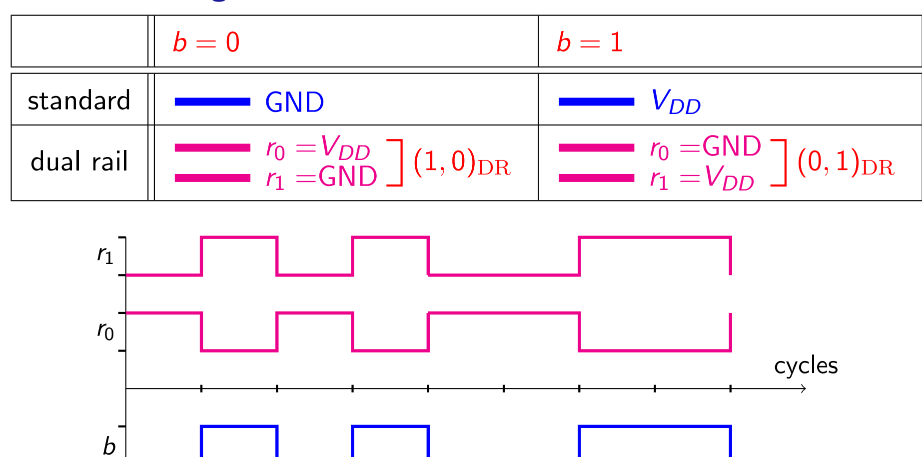


# Low-Level Coding and Circuit Activity

**Assumptions**:

- $b$ is a bit (i.e. $b \in \{0, 1\}$, logical or mathematical value)
- electrical states for a wire : $V_{DD}$ (logical 1) or GND (logical 0)

**Low-level codings of a bit**:



| | $b = 0$ | $b = 1$ |
|---|---|---|
| standard | GND | $V_{DD}$ |
| dual rail | $r_0 = V_{DD}$, $r_1 = \text{GND}$ ] $(1, 0)_{\text{DR}}$ | $r_0 = \text{GND}$, $r_1 = V_{DD}$ ] $(0, 1)_{\text{DR}}$ |

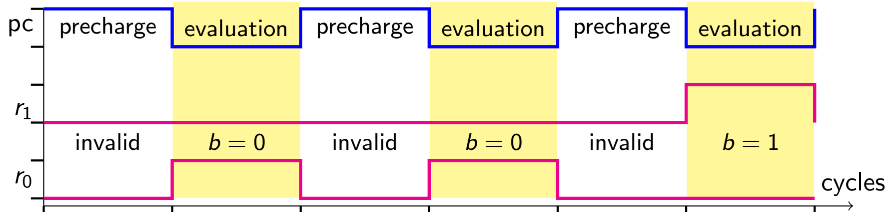
# Circuit Logic Style

**Countermeasure principles**: uniformize circuit activity and exclusive coding

# Circuit Logic Style

**Countermeasure principles**: uniformize circuit activity and exclusive coding

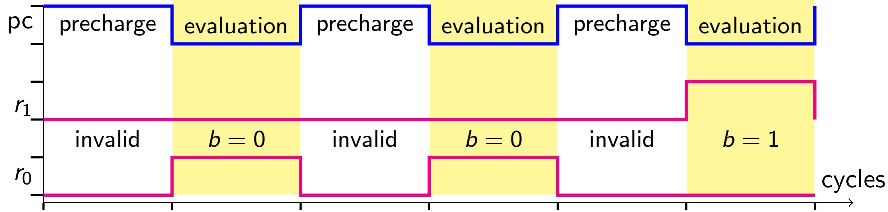
**Solution based on precharge logic and dual-rail coding**:



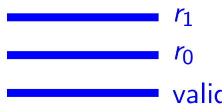
# Circuit Logic Style

**Countermeasure principles**: uniformize circuit activity and exclusive coding

**Solution based on precharge logic and dual-rail coding**:




**Solution based on validity line and dual-rail coding**:

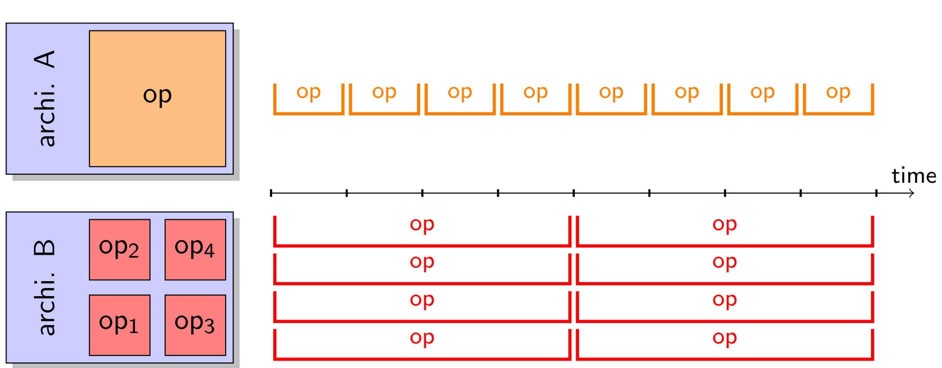



**Important overhead:** silicon area and local storage (registers)

# Countermeasure: Architecture

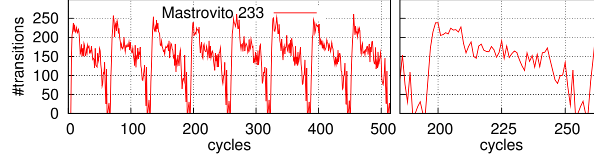
**Increase internal parallelism**:

- replace one fast but big operator
- by several instances of a small but slow one



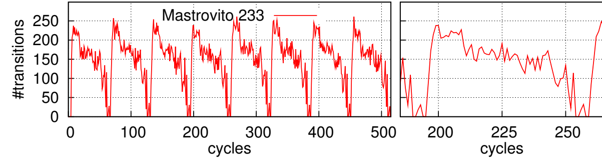
# Protected Multipliers

Unprotected



# Protected Multipliers

Unprotected



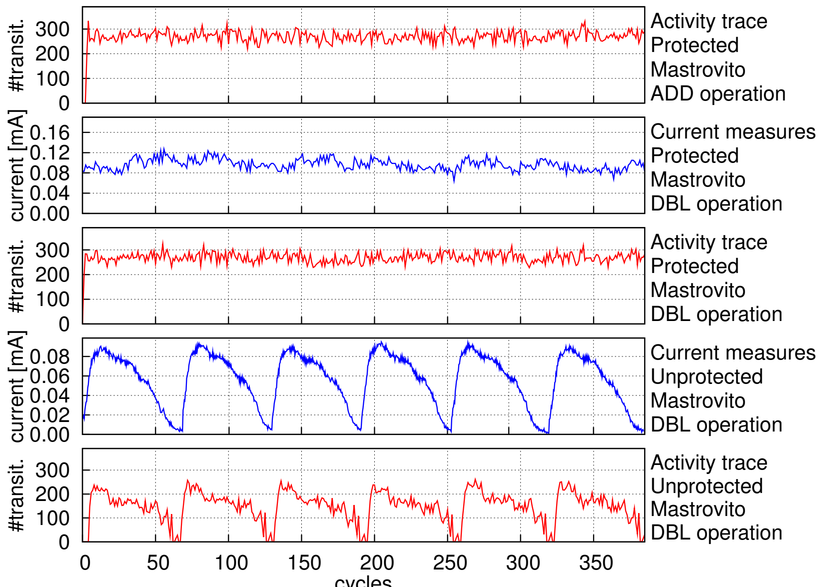

Protected



Overhead:
Area/time $< 10\,\%$

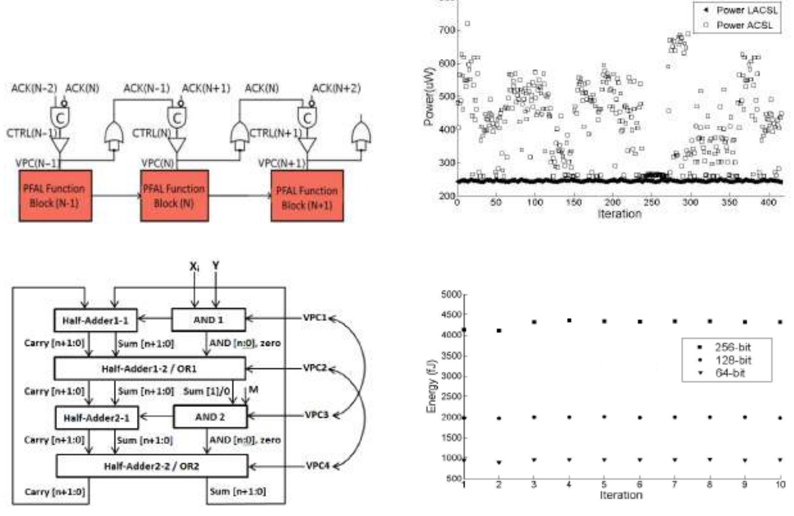
References:
PhD D. Pamula [9]
Articles: [12], [11], [10]

# Protected (Old) Accelerator




Warning: old dedicated accelerator (similar behavior is expected for our new one)

# Circuit-Level Protections for Arithmetic Operators

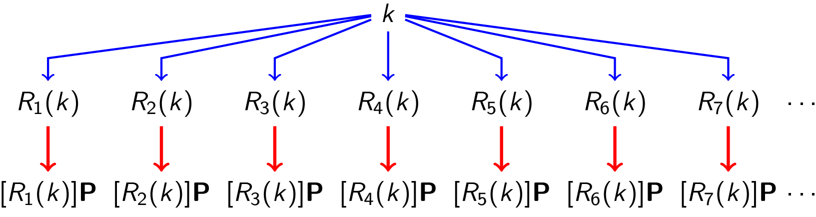



References: [5] and [6]

# Arithmetic Level Countermeasures

Redundant number system =

- a way to improve the performance of some operations
- a way to represent a value with different representations




**Important property:** $\forall i \quad [R_i(k)]\mathbf{P} = [k]\mathbf{P}$

**Proposed solution:** use random redundant representations of $k$

# Double-Base Number System

Standard radix-2 representation:

$$k = \sum_{i=0}^{t-1} k_i 2^i = \boxed{k_{t-1} \mid k_{t-2} \mid \cdots \mid k_2 \mid k_1 \mid k_0}$$

$t$ explicit digits

# Double-Base Number System

Standard radix-2 representation:

$$k = \sum_{i=0}^{t-1} k_i 2^i = $$

| $2^{t-1}$ | $2^{t-2}$ | $\cdots$ | $2^2$ | $2^1$ | $2^0$ | implicit weights |
|---|---|---|---|---|---|---|
| $k_{t-1}$ | $k_{t-2}$ | $\cdots$ | $k_2$ | $k_1$ | $k_0$ | $t$ explicit digits |

Digits: $k_i \in \{0, 1\}$, typical size: $t \in \{160, \ldots, 600\}$

# Double-Base Number System

Standard radix-2 representation:

$$k = \sum_{i=0}^{t-1} k_i 2^i =$$

| $2^{t-1}$ | $2^{t-2}$ | $\cdots$ | $2^2$ | $2^1$ | $2^0$ | implicit weights |
|---|---|---|---|---|---|---|
| $k_{t-1}$ | $k_{t-2}$ | $\cdots$ | $k_2$ | $k_1$ | $k_0$ | $t$ explicit digits |

Digits: $k_i \in \{0, 1\}$, typical size: $t \in \{160, \ldots, 600\}$

Double-Base Number System (DBNS):

$$k = \sum_{j=0}^{n-1} k_j 2^{a_j} 3^{b_j} =$$

# Double-Base Number System

Standard radix-2 representation:

$$k = \sum_{i=0}^{t-1} k_i 2^i = $$

| $2^{t-1}$ | $2^{t-2}$ | $\cdots$ | $2^2$ | $2^1$ | $2^0$ | implicit weights |
|---|---|---|---|---|---|---|
| $k_{t-1}$ | $k_{t-2}$ | $\cdots$ | $k_2$ | $k_1$ | $k_0$ | $t$ explicit digits |

Digits: $k_i \in \{0, 1\}$, typical size: $t \in \{160, \ldots, 600\}$

Double-Base Number System (DBNS):

$$k = \sum_{j=0}^{n-1} k_j 2^{a_j} 3^{b_j} = $$

| | | | | |
|---|---|---|---|---|
| $k_{n-1}$ | $\cdots$ | $k_1$ | $k_0$ | $n$ $(2,3)$−terms |
| $a_{n-1}$ | $\cdots$ | $a_1$ | $a_0$ | explicit "digits" |
| $b_{n-1}$ | $\cdots$ | $b_1$ | $b_0$ | explicit ranks |

$a_j, b_j \in \mathbb{N}$, $k_j \in \{1\}$ or $k_j \in \{-1, 1\}$, size $n \approx \log t$

# Double-Base Number System

Standard radix-2 representation:

$$k = \sum_{i=0}^{t-1} k_i 2^i =$$

| $2^{t-1}$ | $2^{t-2}$ | $\cdots$ | $2^2$ | $2^1$ | $2^0$ | implicit weights |
|---|---|---|---|---|---|---|
| $k_{t-1}$ | $k_{t-2}$ | $\cdots$ | $k_2$ | $k_1$ | $k_0$ | $t$ explicit digits |

Digits: $k_i \in \{0, 1\}$, typical size: $t \in \{160, \ldots, 600\}$

Double-Base Number System (DBNS):

$$k = \sum_{j=0}^{n-1} k_j 2^{a_j} 3^{b_j} =$$

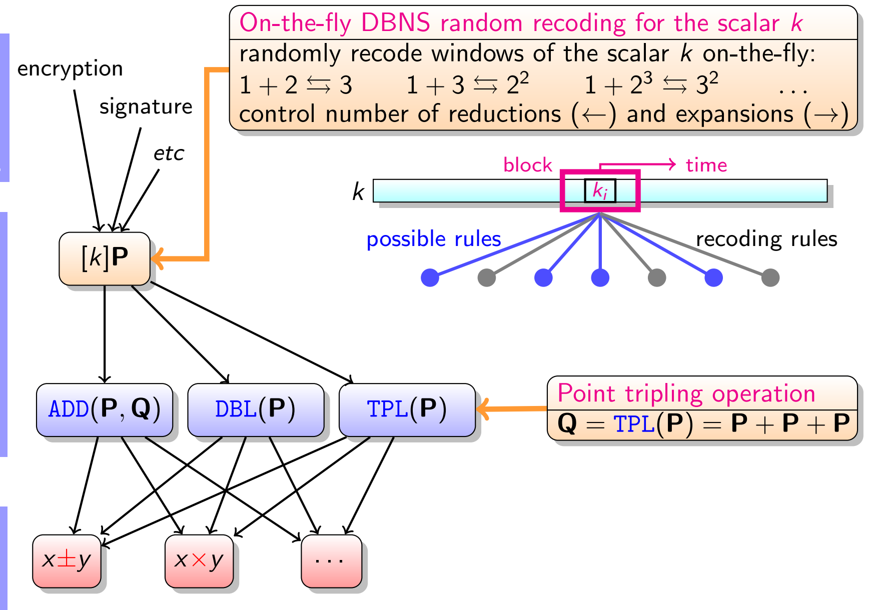
| | | | | |
|---|---|---|---|---|
| $k_{n-1}$ | $\cdots$ | $k_1$ | $k_0$ | $n$ $(2,3)-$terms |
| $a_{n-1}$ | $\cdots$ | $a_1$ | $a_0$ | explicit "digits" |
| $b_{n-1}$ | $\cdots$ | $b_1$ | $b_0$ | explicit ranks |

$a_j, b_j \in \mathbb{N}$, $k_j \in \{1\}$ or $k_j \in \{-1, 1\}$, size $n \approx \log t$

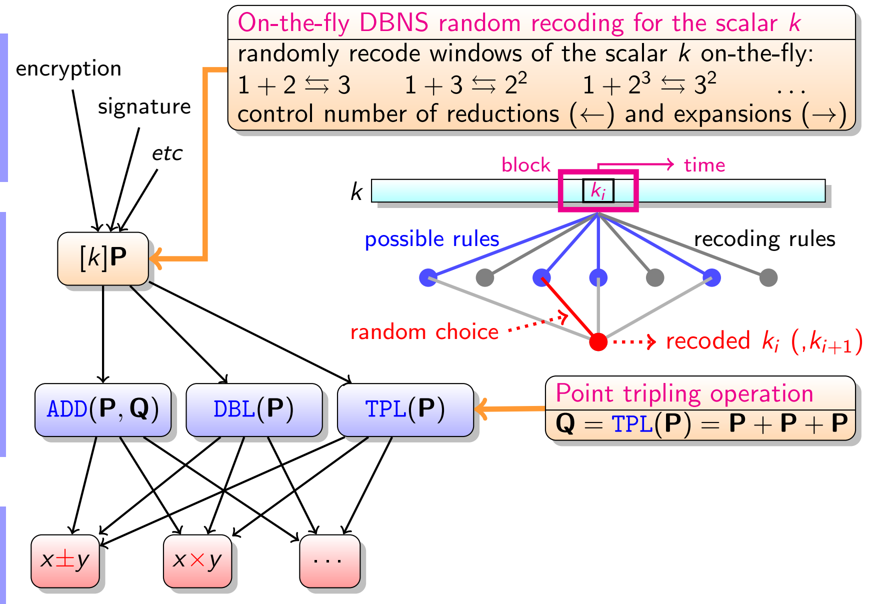
DBNS is a very redundant and sparse representation: $1701 = (11010100101)_2$

$$\begin{aligned}
1701 &= 243 + 1458 = 2^0 3^5 + 2^1 3^6 = (1, 0, 5), (1, 1, 6) \\
&= 1728 - 27 = 2^6 3^3 - 2^0 3^3 = (1, 6, 3), (-1, 0, 3) \\
&= 729 + 972 = 2^0 3^6 + 2^2 3^5 = (1, 0, 6), (1, 2, 5) \\
&\ldots
\end{aligned}$$

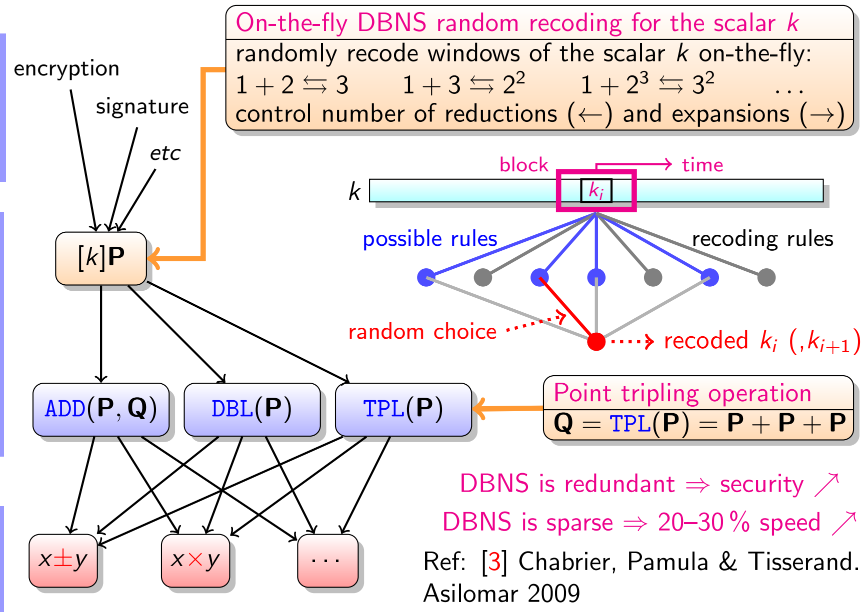
# Randomized DBNS Recoding of the Scalar $k$



# Randomized DBNS Recoding of the Scalar $k$



# Randomized DBNS Recoding of the Scalar $k$



# References

# References I

Surveys: Proc. IEEE 2006 [1], Proc. IEEE 2012 [2], IEEE TVLSI 2013 [7]


[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan.
The sorcerer's apprentice guide to fault attacks.
*Proceedings of the IEEE*, 94(2):370–382, February 2006.

[2] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache.
Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures.
*Proceedings of the IEEE*, 100(11):3056–3076, November 2012.

[3] T. Chabrier, D. Pamula, and A. Tisserand.
Hardware implementation of DBNS recoding for ECC processor.
In *Proc. 44rd Asilomar Conference on Signals, Systems and Computers*, pages 1129–1133, Pacific Grove, California, U.S.A., November 2010. IEEE.

[4] T. Chabrier and A. Tisserand.
On-the-fly multi-base recoding for ECC scalar multiplication without pre-computations.
In A. Nannarelli, P.-M. Seidel, and P. T. P. Tang, editors, *Proc. 21st Symposium on Computer Arithmetic (ARITH)*, pages 219–228, Austin, TX, U.S.A, April 2013. IEEE Computer Society.

[5] J. Chen, A. Tisserand, E. M. Popovici, and S. Cotofana.
Robust sub-powered asynchronous logic.
In J. Becker and M. R. Adrover, editors, *Proc. 24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–7, Palma de Mallorca, Spain, September 2014. IEEE.

[6] J. Chen, A. Tisserand, E. M. Popovici, and S. Cotofana.
Asynchronous charge sharing power consistent Montgomery multiplier.
In J. Sparso and E Yahya, editors, *Proc. 21st IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 132–138, Mountain View, California, USA, May 2015.

[7] D. Karaklajic, J.-M. Schmidt, and I. Verbauwhede.
Hardware designer's guide to fault attacks.
*IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, December 2013.

# References II

[8] P. C. Kocher, J. Jaffe, and B. Jun.
Differential power analysis.
In *Proc. Advances in Cryptology (CRYPTO)*, volume 1666 of *LNCS*, pages 388–397. Springer, August 1999.

[9] D. Pamula.
*Arithmetic Operators on GF($2^m$) for Cryptographic Applications: Performance - Power Consumption - Security Tradeoffs*.
Phd thesis, University of Rennes 1 and Silesian University of Technology, December 2012.

[10] D. Pamula, E. Hrynkiewicz, and A. Tisserand.
Analysis of GF($2^{233}$) multipliers regarding elliptic curve cryptosystem applications.
In *11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems (PDeS)*, pages 271–276, Brno, Czech Republic, May 2012.

[11] D. Pamula and A. Tisserand.
GF($2^m$) finite-field multipliers with reduced activity variations.
In *4th International Workshop on the Arithmetic of Finite Fields*, volume 7369 of *LNCS*, pages 152–167, Bochum, Germany, July 2012. Springer.

[12] D. Pamula and A. Tisserand.
Fast and secure finite field multipliers.
In *Proc. 18th Euromicro Conference on Digital System Design (DSD)*, pages 653–660, Madeira, Portugal, August 2015.

[13] J. Proy, N. Veyrat-Charvillon, A. Tisserand, and N. Meloni.
Full hardware implementation of short addition chains recoding for ECC scalar multiplication.
In *Actes Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS)*, Lille, France, June 2015.

# Good Books (in French)

**Micro et nano-électronique**

*Bases, Composants, Circuits*

Hervé Fanet

2006

Dunod

ISBN: 2–10–049141–5

---

**Arithmétique des ordinateurs**

Jean-Michel Muller

1989

Masson

ISBN: 2–225–81689–1

(web version)

# Good Books (in English)

**CMOS VLSI Design**

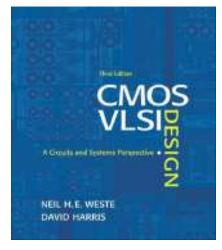
*A Circuits and Systems Perspective*

Neil Weste and David Harris

3rd edition, 2004

Addison Wesley

ISBN: 0–321–14901–7



**Power Analysis Attacks**

*Revealing the Secrets of Smart Cards*

Stefan Mangard, Elisabeth Oswald and Thomas Popp

2007

Springer

ISBN:978-0-387-30857-9

# Good Books (in English)

**Digital Arithmetic**

Milos Ercegovac and Tomas Lang

2003

Morgan Kaufmann

ISBN: 1–55860–798–6

# Thank you!

Contact (will change in a few months):


- mailto:arnaud.tisserand@univ-ubs.fr
- http://www-labsticc.univ-ubs.fr/~tisseran