Arithmetic and Functional Activity

Arnaud TISSERAND

CNRS

WRACH 2025 Roscoff

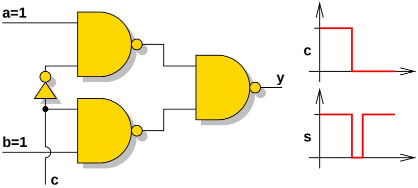
This work received funding from the France 2030 program, managed by the French National Research Agency under grant agreement No. ANR-22-PECY-0004 ARSENE

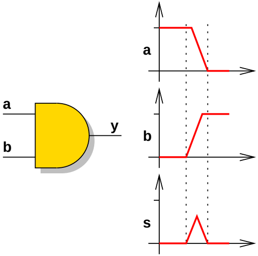
- Representations of numbers and arithmetic algorithms
- Implementation:
  - hardware: accelerators and dedicated processors (ASIC, FPGA)
  - software: embedded processors, high-end multicores
  - targets: high performances, low power, security
- Crypto applications: asymmetric, hash functions, symmetric ciphers, homomorphic encryption
- Protections against observation and perturbation attacks
- Design and validation tools

Electrical Activity in Digital Circuits

Power consumption:

- Static power due to leakages
- Dynamic power due to transitions
- Functional/logical/switching/useful transitions due to state changes at bit level ($0 \to 1$ and $1 \to 0$)
- Parasitic transitions due to timings imperfections (skew, glitches, . . .)





This work only deals with functional activity

General principle:

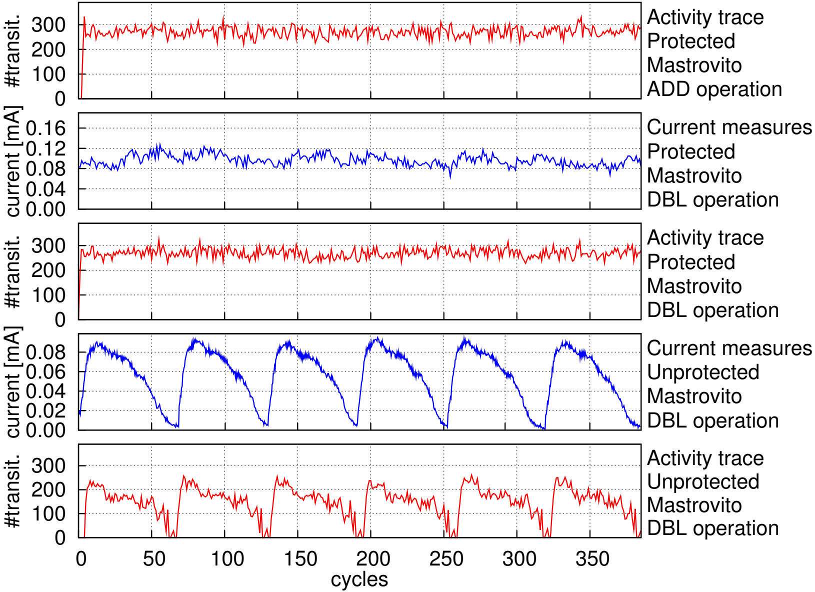
1. Measure/observe external physical parameter(s) on a running device
2. Deduce internal (secret) informations

Examples:

- Timings
- Power consumption
- Electromagnetic radiation
- Temperature
- Number of cache misses
- ...

Attacks are always improving (advanced models, strong statistics, deep learning, experiences...)

Example: Secure Hardware Accelerator for ECC



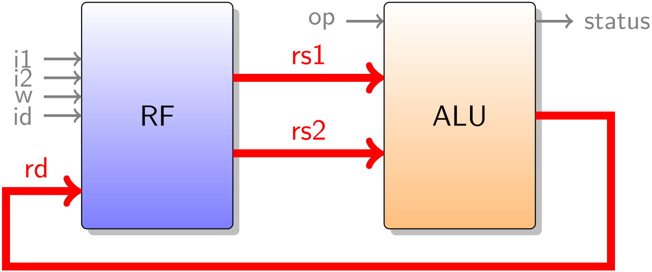
PhD Thesis Danuta Pamula 2012: https://theses.hal.science/tel-00767537 https://www.pepr-cyber-arsene.fr/

- Secure processors (32 and 64-bit RISC-V)
- Secure SoC: RNG, secure memories, cryptographic accelerators
- Secure OS and software
- Validation methods and tools
- Prototyping

Personal topics:

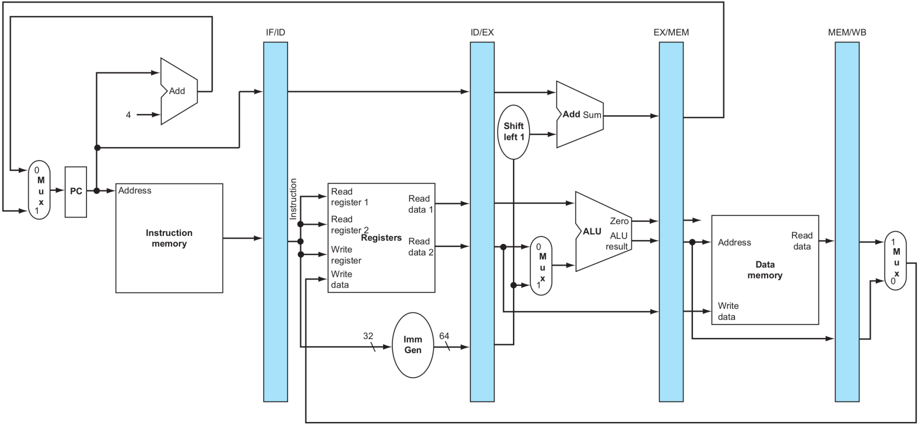
- Cryptographic accelerators
- Analysis and secure design of processors against SCA / FIA

- Very simplified 32-bit processor
- Register file (RF): 32 registers, 2 read ports (rs1, rs2) and 1 write port (rd) active at each instruction
- Basic instruction set and arithmetic and logic unit (ALU)
- Simulation: only logical transitions (no glitch), 1-cycle instructions
- Only (rs1, rs2, rd) are observable (not other signals!)
- Start with random data in registers (crypto context)

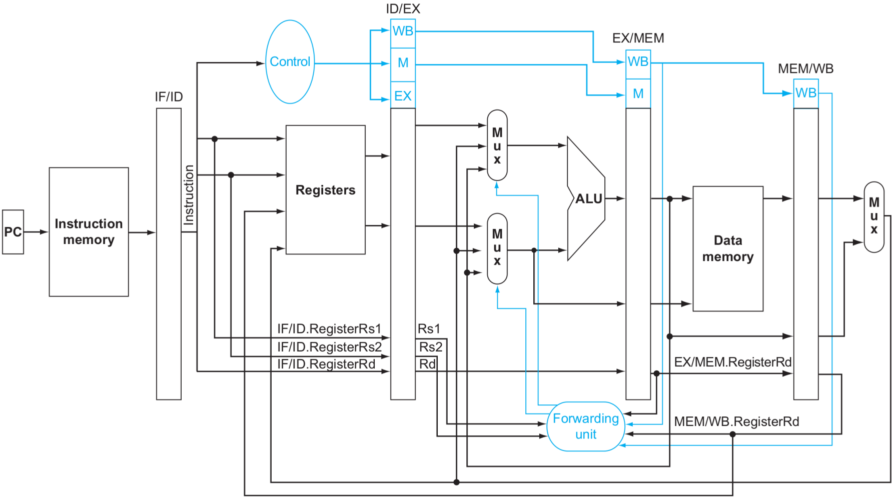


See presentation “Representations of Numbers and Electrical Activity” at NAC 2024 (Paris, Feb.)

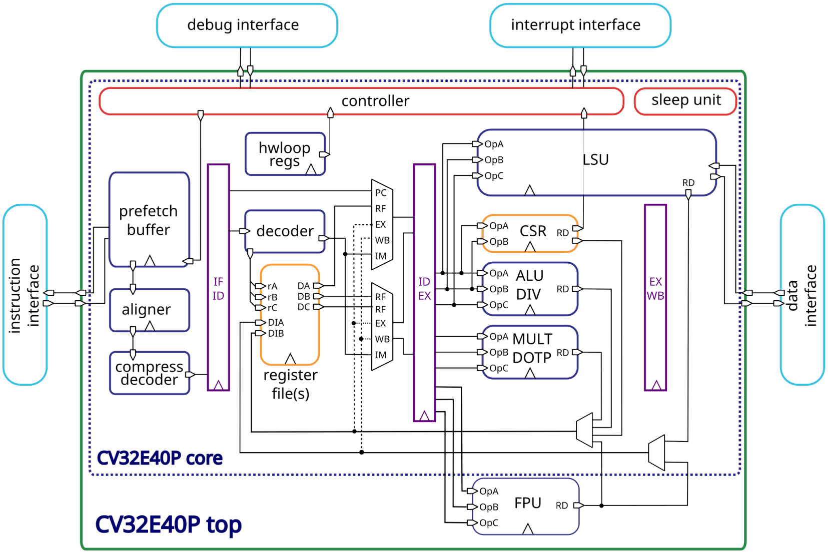
Source: page 279 of the reference book "Computer Organization and Design: The Hardware/Software Interface, RISC-V edition" by D.A. Patterson & J.L. Hennessy, Morgan Kaufmann, 2018



Source: page 314 of the reference book "Computer Organization and Design: The Hardware/Software Interface, RISC-V edition" by D.A. Patterson & J.L. Hennessy, Morgan Kaufmann, 2018



RISC-V CV32E40P core from OpenHW Group (documentation)



HDL code(s), complete software toolchain, numerous libraries and works

- CABA: cycle accurate, bit accurate (HDL-like description)
- Simulation of functional activity traces in all registers (no glitch)
- 32-bit words and functional unit (FU)
- Register file (RF): 32 registers, 2 read ports and 1 write port
- Start with random data in register file (crypto context)
- Basic instruction set:
  - `ADD Rd Ra Rb`
  - `SUB Rd Ra Rb`
  - `MUL Rd Ra Rb`
  - `SET Rd imm`
  - `CMP Ra Rb`
  - `NOP`
  - ...
- Pipeline with 4 stages: fetch | decode | exec | wb (write back)

Fetch:

```
if jump then
    instruction <- IMEM[@jump] | pc <- @jmp + 1
else
    instruction <- IMEM[pc]    | pc <- pc + 1
...
```

Decode:

```
id, ia, ib, operation, imm, jump, @jump, @dmem, ... <- decode(instru
reg_a, reg_b <- RF[ia], RF[ib]
...
```

Execution:

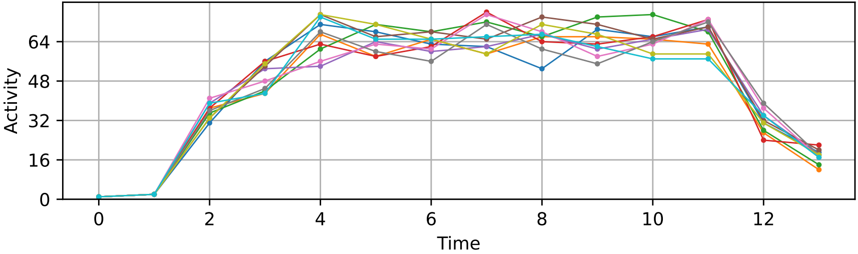
```
reg_r <- FU(operation, reg_a, reg_b)
...
```

Write back WB:

```
RF[id] <- reg_r
...
```

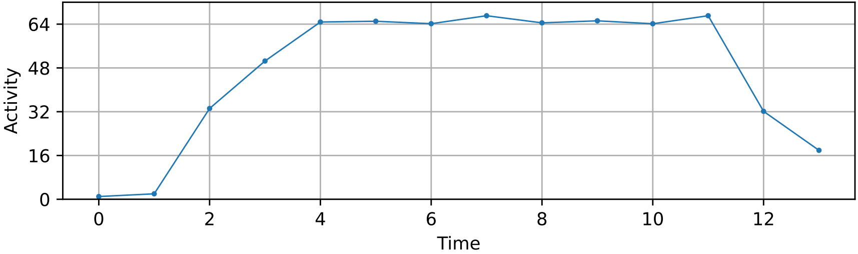
Functional Activity Traces


```
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15
 6   ADD  R16, R17, R18
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```

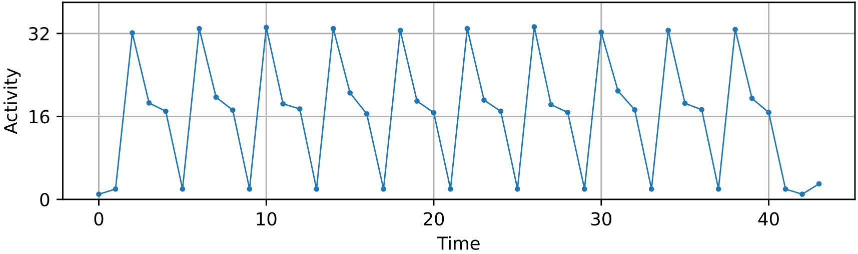


Functional Activity Average Trace

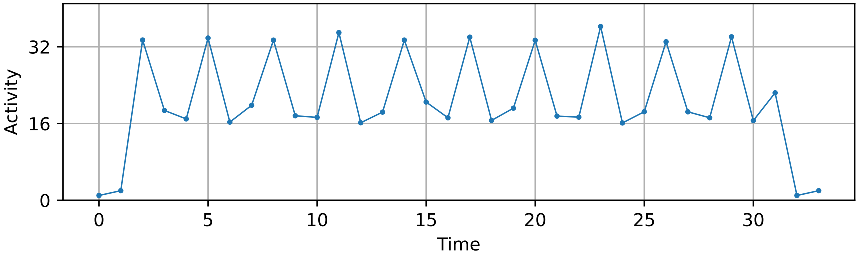
```
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15
 6   ADD  R16, R17, R18
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```



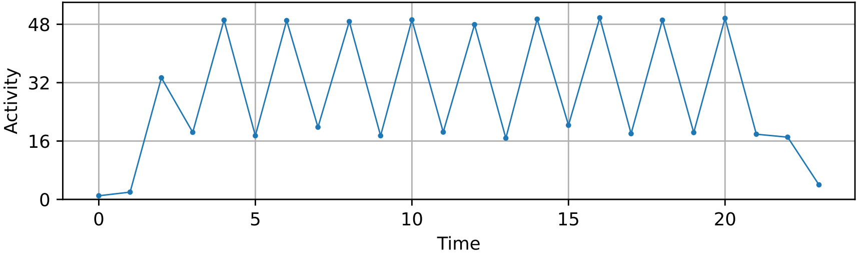
```
// pipeline mode: 3 cycles stall between instructions
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15
 6   ADD  R16, R17, R18
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```



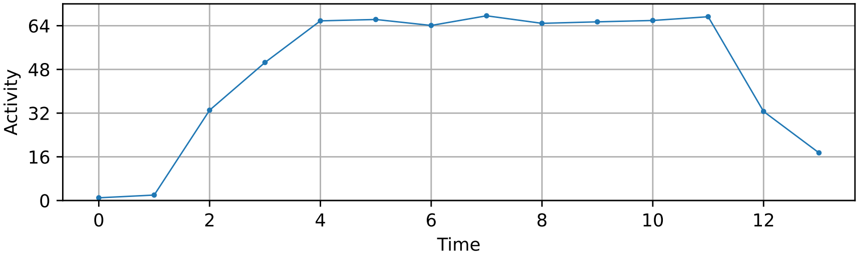
```
// pipeline mode: 2 cycles stall between instructions
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15
 6   ADD  R16, R17, R18
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```



```
// pipeline mode: 1 cycle stall between instructions
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15
 6   ADD  R16, R17, R18
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```

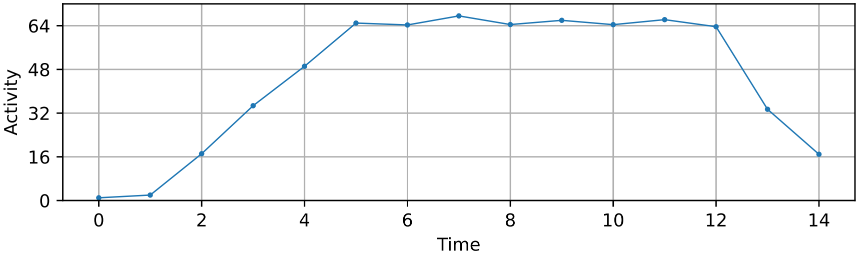


```
// pipeline "normal" mode: 0 cycle stall between instructions
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15
 6   ADD  R16, R17, R18
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```

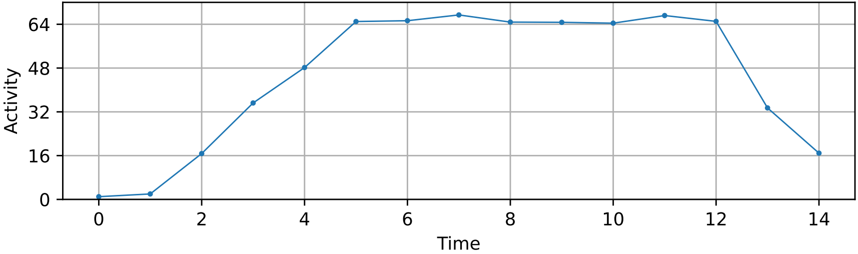


Add 0


```
NOP
SET  R21, 0
ADD  R1, R2, R3
ADD  R4, R5, R6
ADD  R7, R8, R9
ADD  R10, R11, R12
ADD  R13, R14, R15
ADD  R16, R17, R18
ADD  R19, R20, R21      // + 0
ADD  R22, R23, R24
ADD  R25, R26, R27
ADD  R28, R29, R30
```

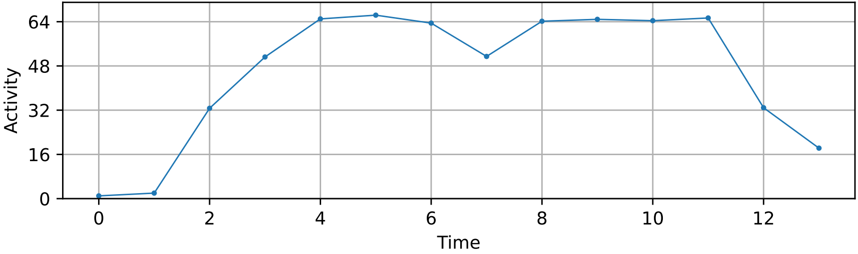


```
NOP
SET  R21, 0
ADD  R1, R2, R3
ADD  R4, R5, R6
ADD  R7, R8, R9
ADD  R10, R11, R12
ADD  R13, R14, R15
ADD  R16, R17, R18
MUL  R19, R20, R21      // * 0
ADD  R22, R23, R24
ADD  R25, R26, R27
ADD  R28, R29, R30
```

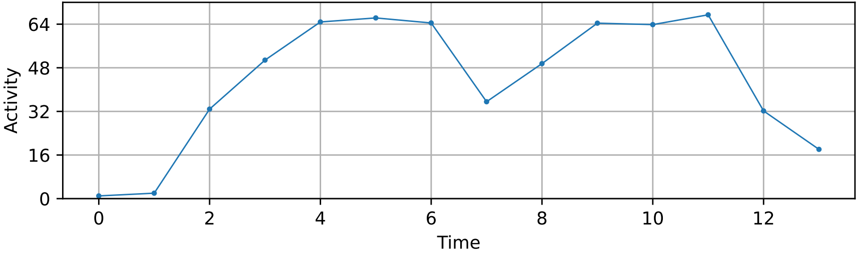


Register Reuse (1/2)


```
// reuse second operand at lines 5 and 6
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15      // R14 + R15
 6   ADD  R16, R17, R15      // R17 + R15
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```

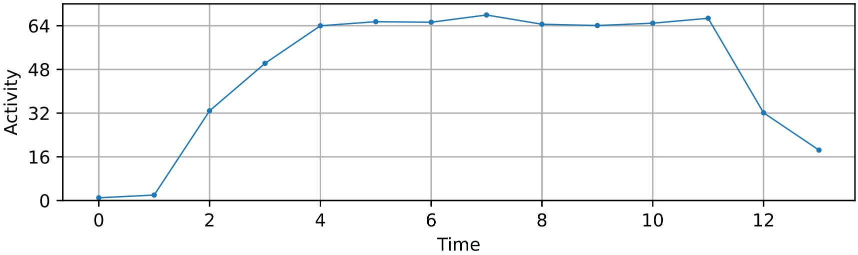


```
// reuse both operands at lines 5 and 6
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15      // R14 + R15
 6   ADD  R16, R14, R15      // R14 + R15
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```

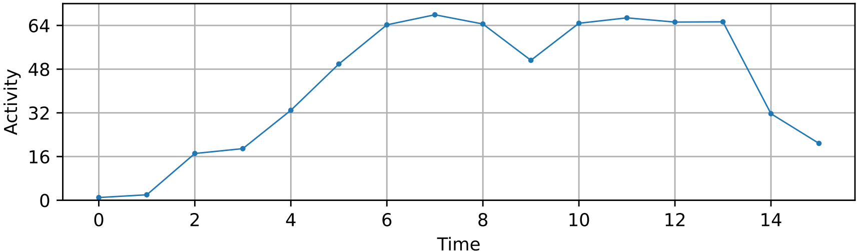


Avoid Register Reuse

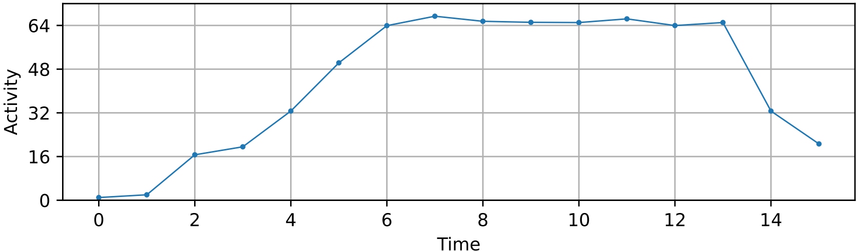
```
// reuse R15 at lines 5 and 6 with different source registers
 0   NOP
 1   ADD  R1, R2, R3
 2   ADD  R4, R5, R6
 3   ADD  R7, R8, R9
 4   ADD  R10, R11, R12
 5   ADD  R13, R14, R15      // R14 + R15
 6   ADD  R16, R15, R17      // R15 + R17
 7   ADD  R19, R20, R21
 8   ADD  R22, R23, R24
 9   ADD  R25, R26, R27
10   ADD  R28, R29, R30
```



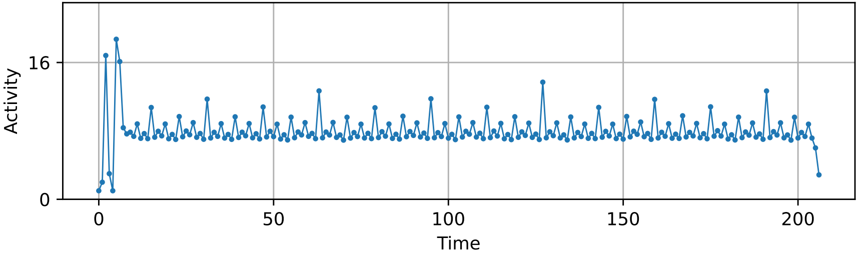
```
NOP
SET  R15, 0
SET  R18, 1
ADD  R1, R2, R3
ADD  R4, R5, R6
ADD  R7, R8, R9
ADD  R10, R11, R12
ADD  R13, R14, R15      // rnd + 0
ADD  R16, R17, R18      // rnd + 1
ADD  R19, R20, R21
ADD  R22, R23, R24
ADD  R25, R26, R27
ADD  R28, R29, R30
```



```
 0   NOP
 1   SET  R15, 0
 2   SET  R17, 1
 3   ADD  R1, R2, R3
 4   ADD  R4, R5, R6
 5   ADD  R7, R8, R9
 6   ADD  R10, R11, R12
 7   ADD  R13, R14, R15      // rnd + 0
 8   ADD  R16, R17, R18      // 1 + rnd
 9   ADD  R19, R20, R21
10   ADD  R22, R23, R24
11   ADD  R25, R26, R27
12   ADD  R28, R29, R30
```

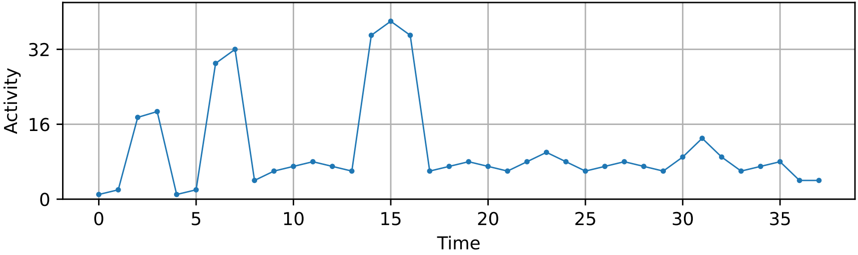


```
// unrolled counter incrementation
 0   NOP
 1   SET  R1, 1
 2   NOP
 3   NOP
 4   ADD  R4, R4, R1
 5   ADD  R4, R4, R1
 6   ADD  R4, R4, R1
 7   ADD  R4, R4, R1
 8   ADD  R4, R4, R1
 9   ADD  R4, R4, R1
 ...
```

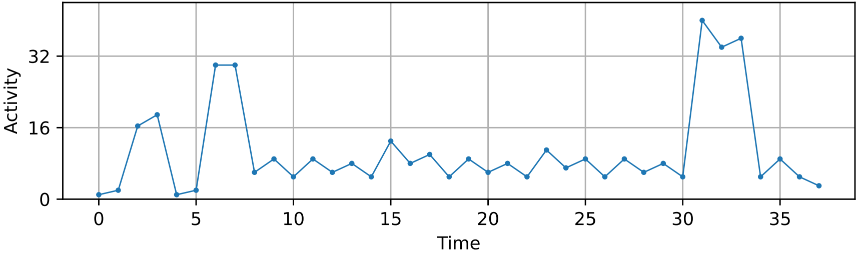


Counter (2/3)

```
0   NOP
1   SET  R1, 1
2   SET  R4, 1073741816      // 2^30 - 8
3   NOP
4   NOP
5   ADD  R4, R4, R1
6   ADD  R4, R4, R1
7   ADD  R4, R4, R1
8   ADD  R4, R4, R1
9   ADD  R4, R4, R1
...
```

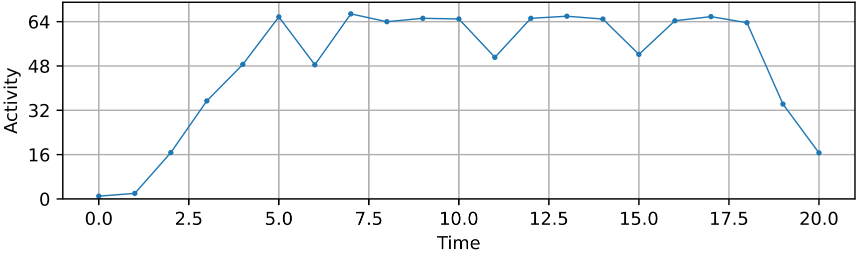


```
0   NOP
1   SET  R1, 1
2   SET  R4, 1073741799      // 2^30 - 25
3   NOP
4   NOP
5   ADD  R4, R4, R1
6   ADD  R4, R4, R1
7   ADD  R4, R4, R1
8   ADD  R4, R4, R1
9   ADD  R4, R4, R1
...
```



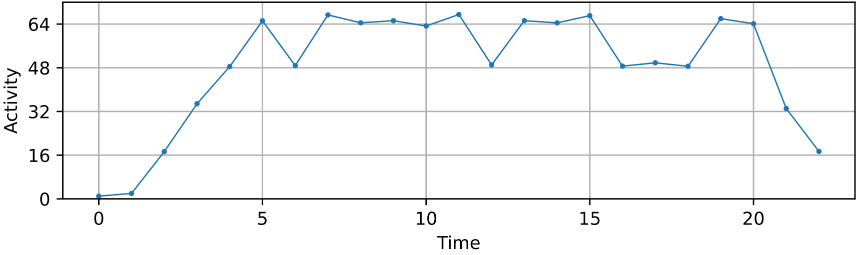
Comparison (1/2)


```
SET  R30, 0
ADD  R10, R26, R27      // R10
ADD  R11, R28, R29      // R11 != R10
ADD  R12, R11, R30      // R12 = R11
ADD  R1, R2, R3
ADD  R4, R5, R6
ADD  R7, R8, R9
CMP  R10, R11           // diff
ADD  R13, R14, R15
ADD  R16, R17, R18
ADD  R19, R20, R21
CMP  R11, R12           // eq
ADD  R22, R23, R24
```



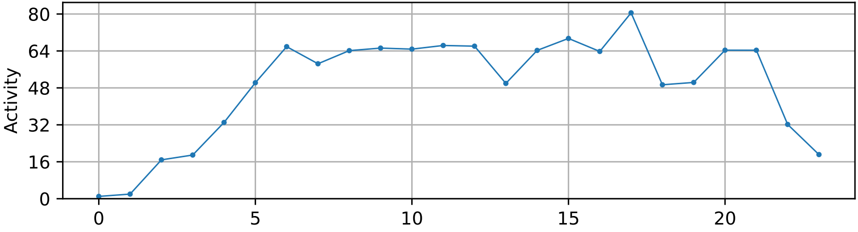
Comparison (2/2)

```
SET  R30, 0
ADD  R10, R26, R27      // R10
ADD  R11, R28, R29      // R11 != R10
ADD  R12, R11, R30      // R12 = R11
ADD  R1, R2, R3
ADD  R4, R5, R6
ADD  R7, R8, R9
MUL  R28, R2, R30      // * 0
CMP  R10, R11          // diff
ADD  R13, R14, R15
ADD  R16, R17, R18
ADD  R19, R20, R21
MUL  R27, R2, R30      // * 0
CMP  R11, R12          // eq
ADD  R22, R23, R24
ADD  R25, R26, R27
```



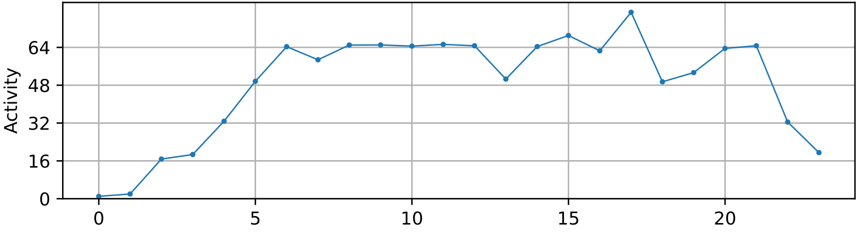
Comparison and Specific Value (1/2)

```
NOP
SET  R30, 8380417   // P = 2^23 - 2^13 +1
SET  R2, 16760834   // 2 * P
ADD  R10, R26, R27  // R10
ADD  R11, R28, R29  // R11 != R10
ADD  R12, R11, R30  // R12 = R11
ADD  R1, R2, R3
ADD  R4, R5, R6
ADD  R7, R8, R9
SUB  R28, R2, R30   // 2P - P
CMP  R10, R11       // diff
ADD  R13, R14, R15
ADD  R16, R17, R18
ADD  R19, R20, R21
SUB  R27, R2, R30   // 2P - P
CMP  R11, R12       // eq
ADD  R22, R23, R24
```



Comparison and Specific Value (2/2)


```
NOP
SET  R30, 8380417   // P=2^23 - 2^13 + 1 = 0b00000000011111111110000000000001
SET  R2, 25141251   // 3 * P             = 0b00000001011111111010000000000011
ADD  R10, R26, R27  // R10
ADD  R11, R28, R29  // R11 != R10
ADD  R12, R11, R30  // R12 = R11
ADD  R1, R2, R3
ADD  R4, R5, R6
ADD  R7, R8, R9
SUB  R28, R2, R30   // 3P - P
CMP  R10, R11       // diff
ADD  R13, R14, R15
ADD  R16, R17, R18
ADD  R19, R20, R21
SUB  R27, R2, R30   // 3P - P
CMP  R11, R12       // eq
ADD  R22, R23, R24
```



- Several instructions interact in the pipeline $\Longrightarrow$ unexpected effects
- Instructions and control flow are **very** important for SCA but *operands* also participate to side-channel leakage
- (Micro)-architecture details impact side-channel leakage
- What is a “good” description (/analysis) level???
  - algorithm
  - code
  - assembly
  - binary
  - micro-architecture
- Compilers (and CAD tools) optimizations can remove some protection “tricks”
- Do not overestimate “constant-time” benefit

- Arithmetic aspect**S** impact electrical activity
- Need more work:
  - modeling links between arithmetic (representations, algorithms) and electrical properties
  - selection/design of appropriate algorithms/implementations
  - take into account parasitic transitions (tricky)
  - "calibration" of library components from implementation**S** result**S**
- Challenge: design countermeasures against observation *and* perturbation attacks
- Teaching sessions

Thank you! Questions?

arnaud.tisserand@cnrs.fr / https://www.arnaud-tisserand.fr