Electrical Activity in Arithmetic Operations vs Security

Arnaud TISSERAND

CNRS, Lab-STICC

arnaud.tisserand@cnrs.fr

https://www.arnaud-tisserand.fr

RAIM 2025 Lyon

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

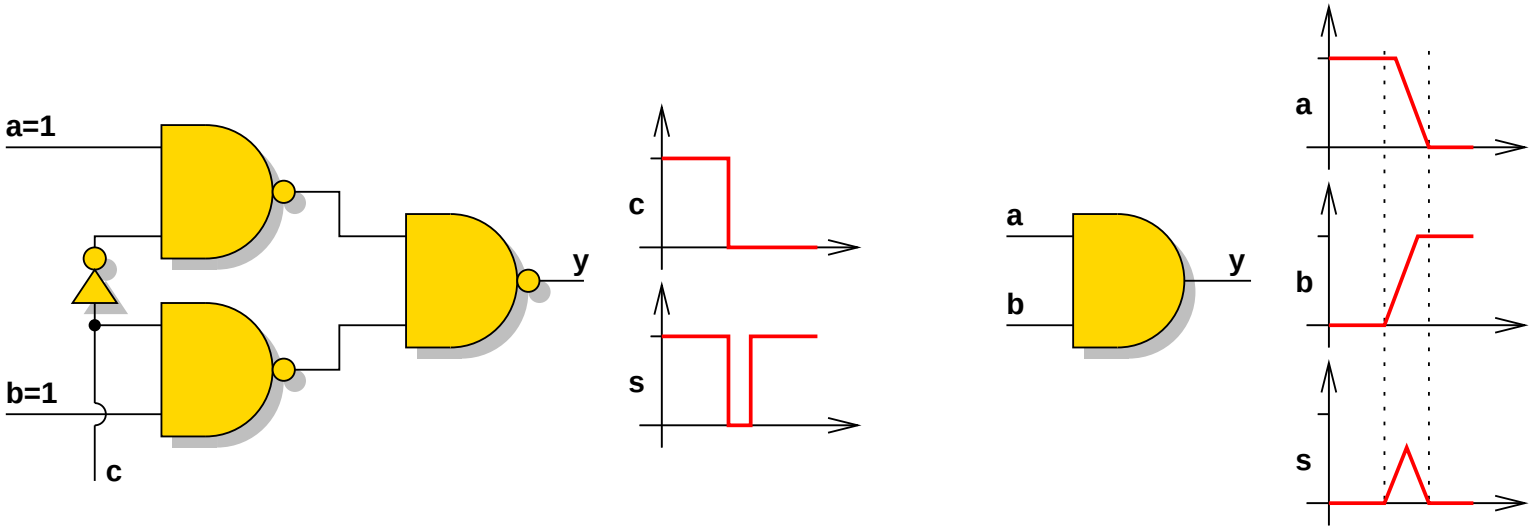
https://www.pepr-cyber-arsene.fr/

- Secure processors (32 and 64-bit RISC-V) with protections against software and hardware attacks
- Secure SoC: RNG, secure memories, cryptographic accelerators
- Links with secure OS and software
- Validation methods and tools
- Prototyping (ASIC and FPGA)

This talk: electrical activity (at transition level) variations when executing arithmetic operations in a simple 32b RISC processor micro-architecture

Power consumption:

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



This work only deals with functional activity (working on parasitics...)

General principle:

1. Measure/observe variations of external physical parameter(s) on a running device (processor, ASIC, FPGA, . . .)
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. . .)

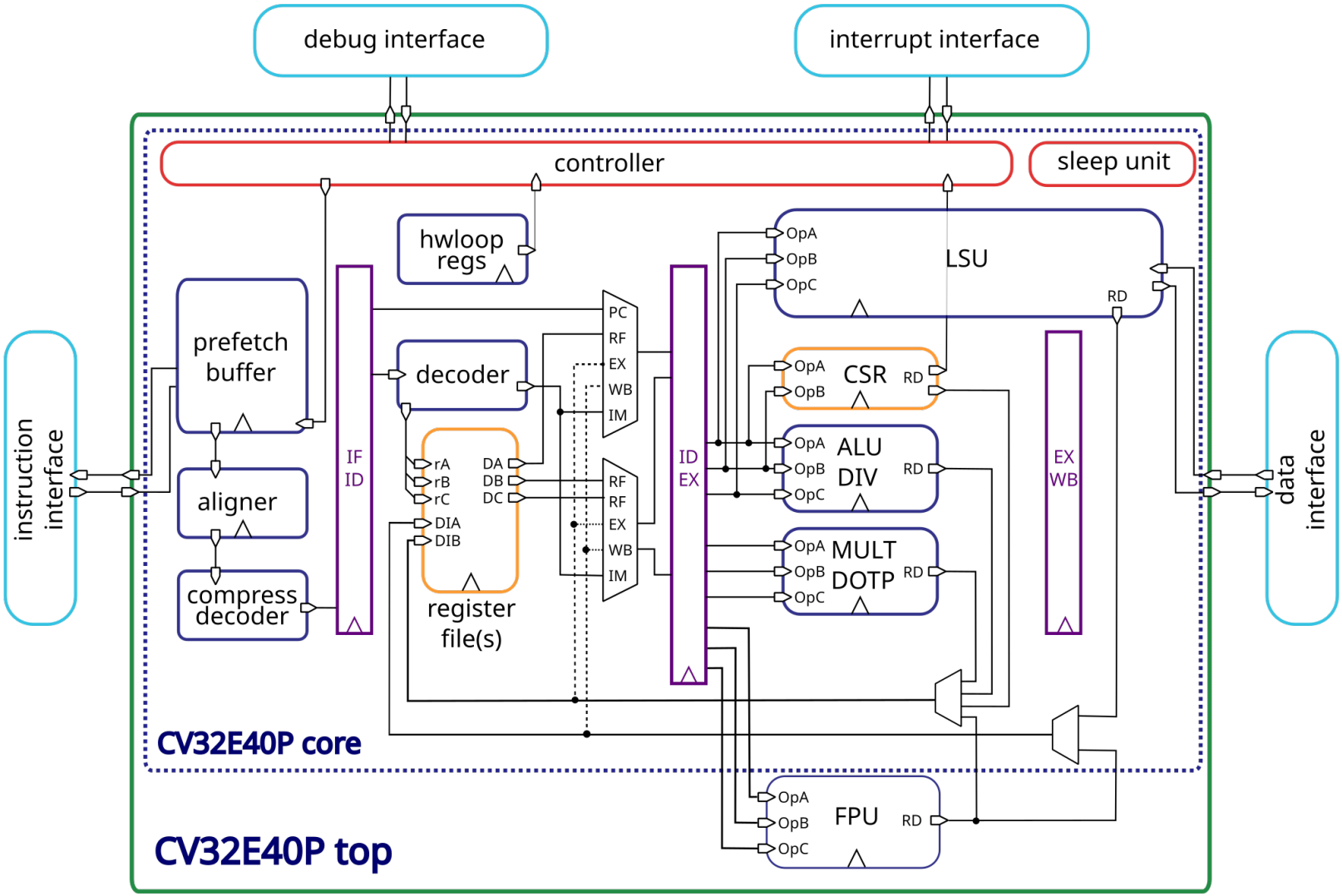
previous bit state	next bit state	activity
"0"	"0"	no transition
"0"	"1"	one transition
"1"	"0"	one transition
"1"	"1"	no transition

When a value with (50% “0”s, 50% “1”s) bits is replaced by a another value with the same distribution, there is a 50% activity rate in the register.

Replacing such a value by a another with a different distribution (or the opposite way), will lead to a different activity rate which can be observed through physical measurements.

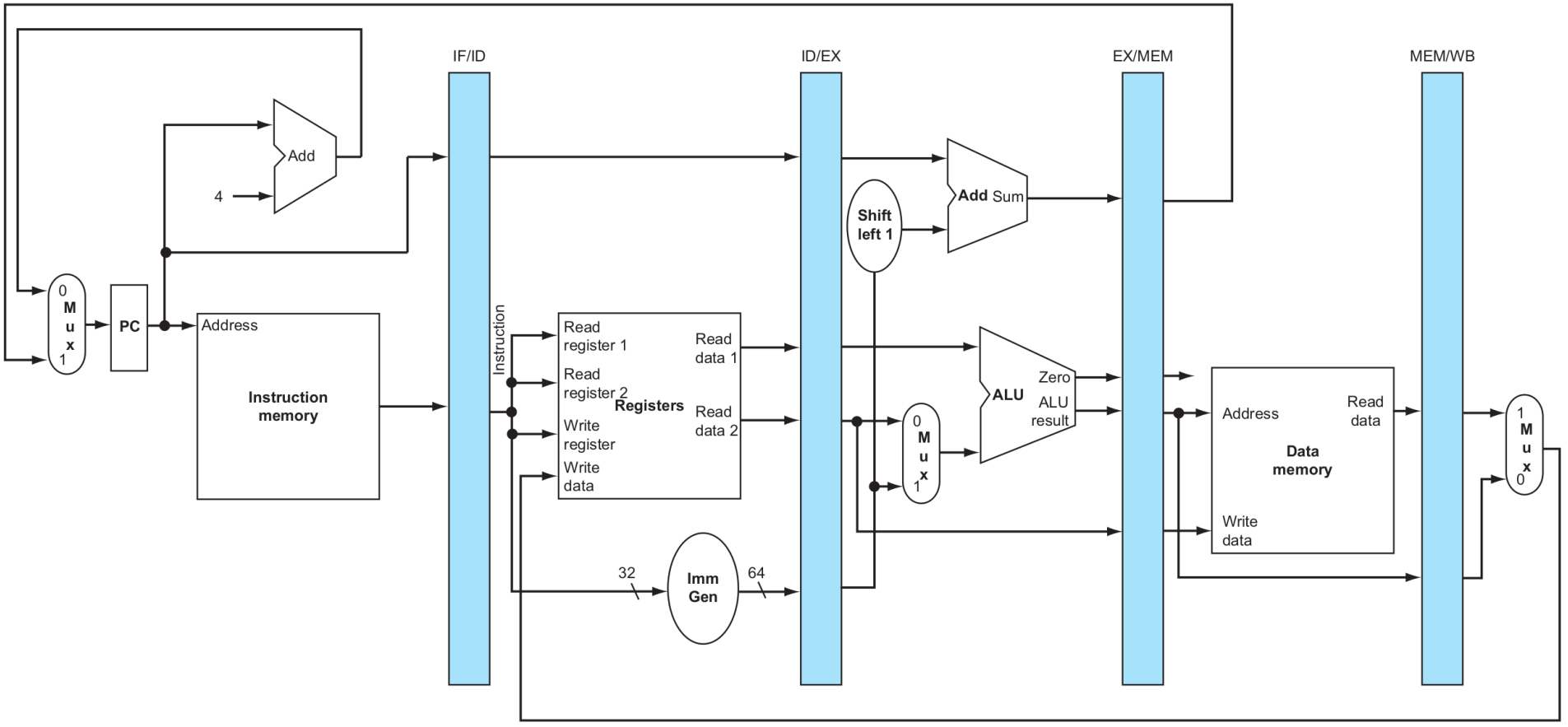
Here I assume “0 to 1” and “1 to 0” transitions use the same energy

RISC-V CV32E40P core from OpenHW Group (documentation)



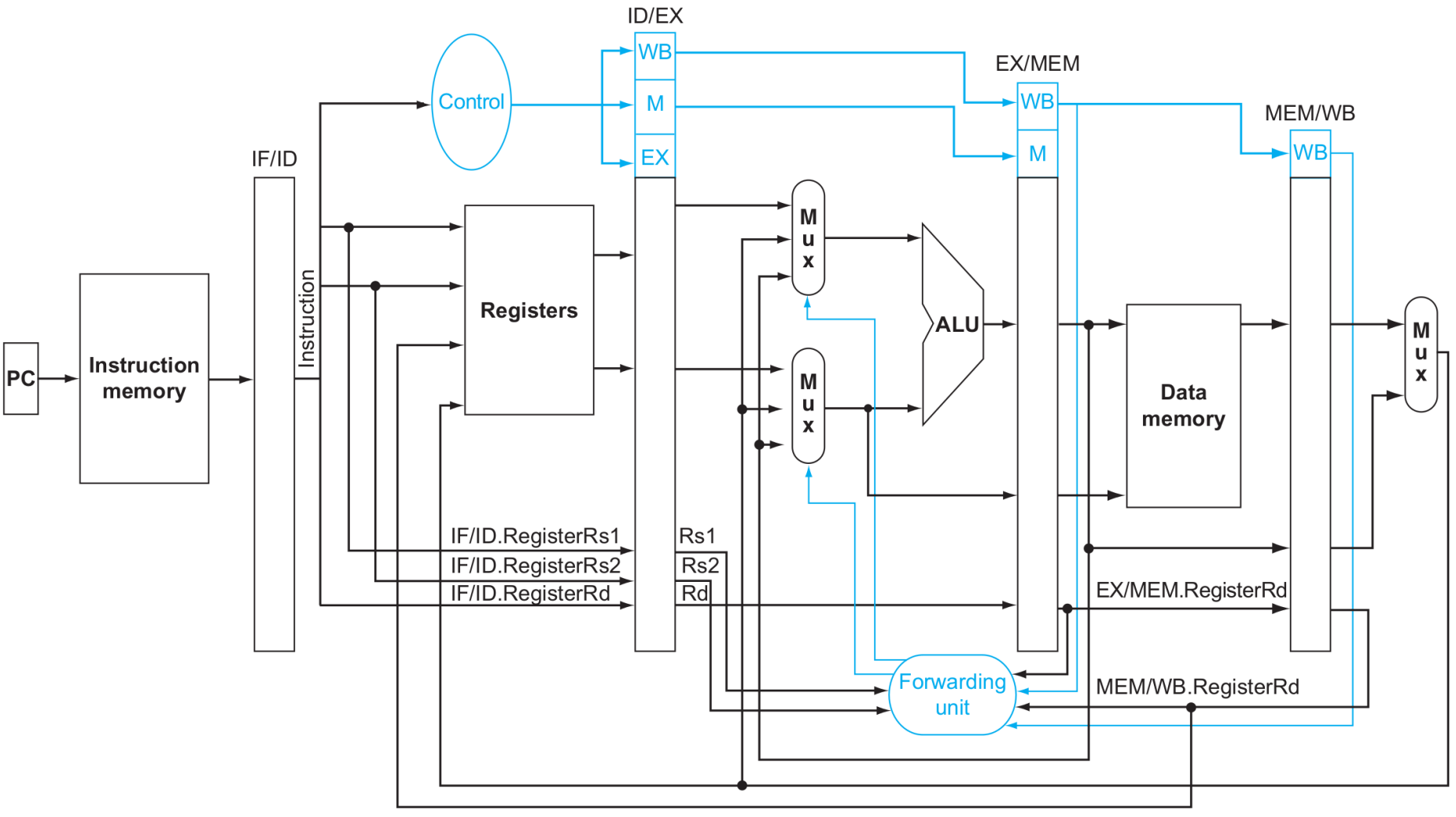
HDL code(s), complete software toolchain, numerous libraries and works

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



Simple RISC Processor Core Example (2/2)

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



- 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 | write back (wb)

Processor Pipeline (Simplified Overview)

Fetch:

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

Decode:

```
id, ia, ib, operat., imm, jmp, @jmp, @dmem, ... <- decode(instr.)
reg_a, reg_b <- RF[ia], RF[ib]
...
```

Execution:

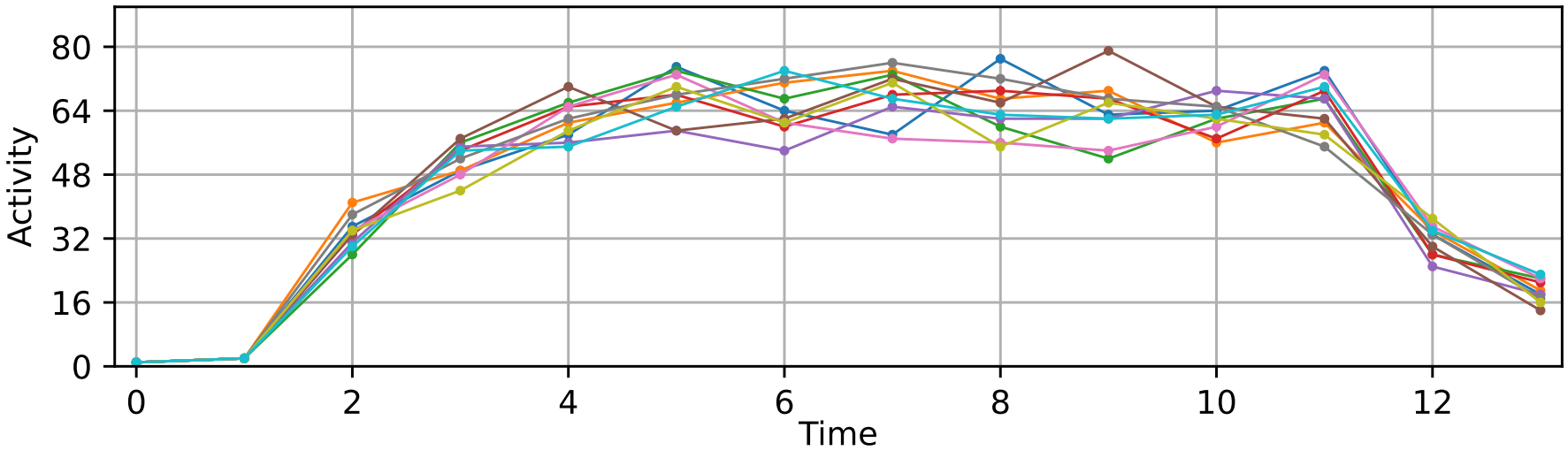
```
reg_r <- FU(operat., reg_a, reg_b)
...
```

Write back:

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

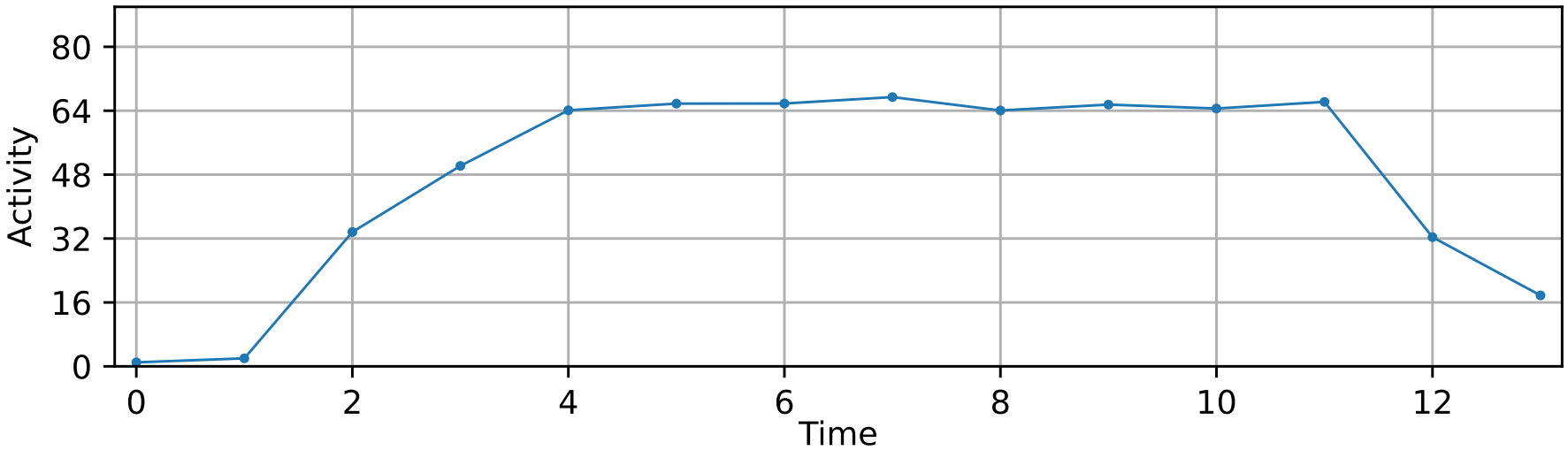
Functional Activity Traces


```
 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
```



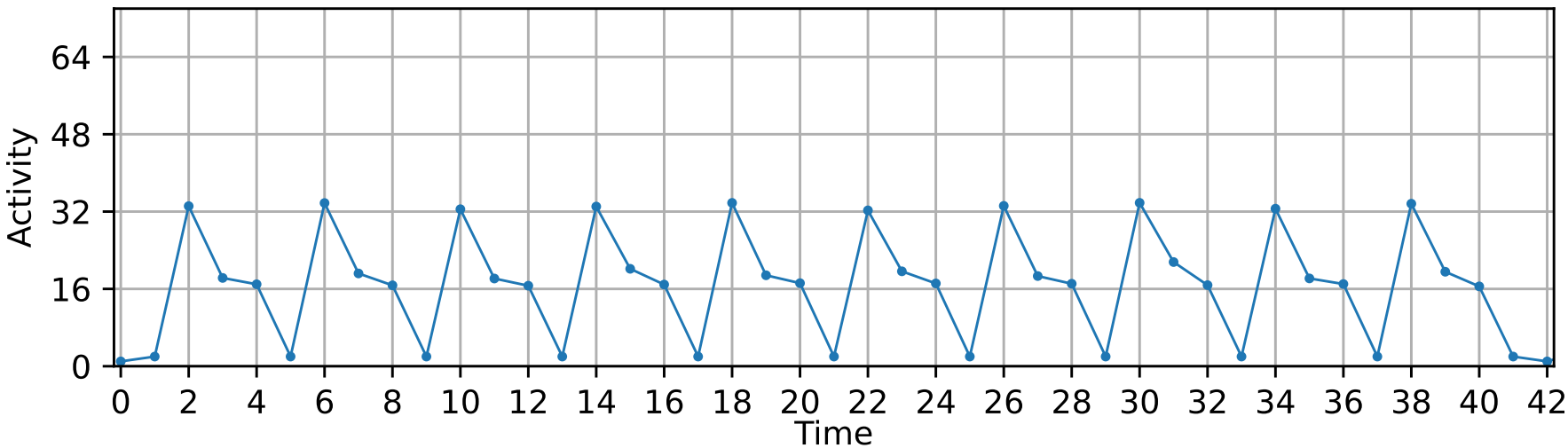
Functional Activity Average Trace


```
 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
```



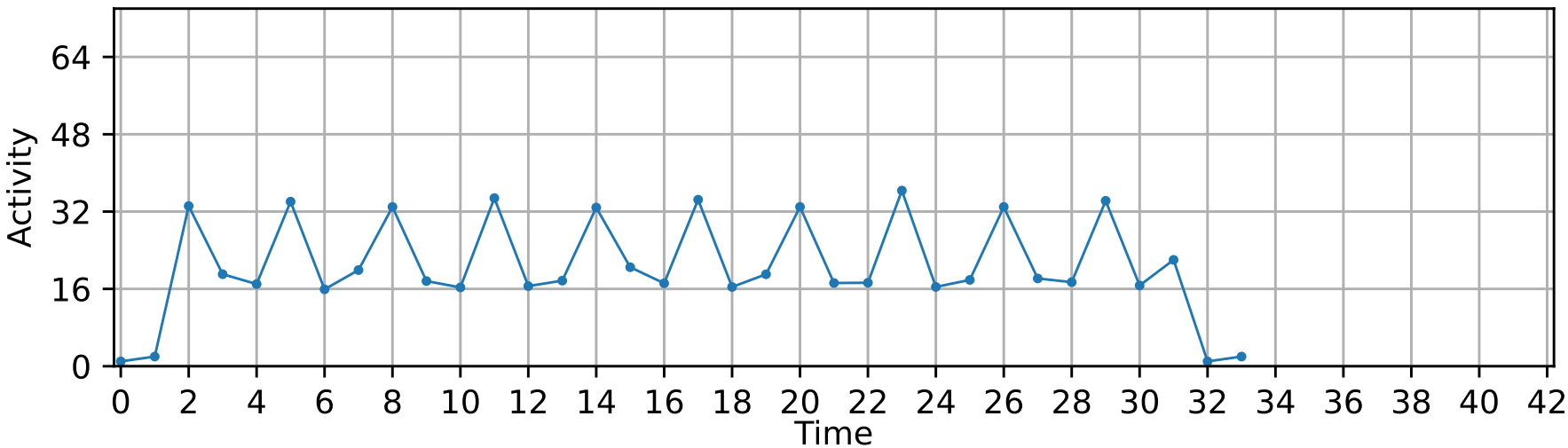
Processor Pipeline (1/4)


```
// 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
```



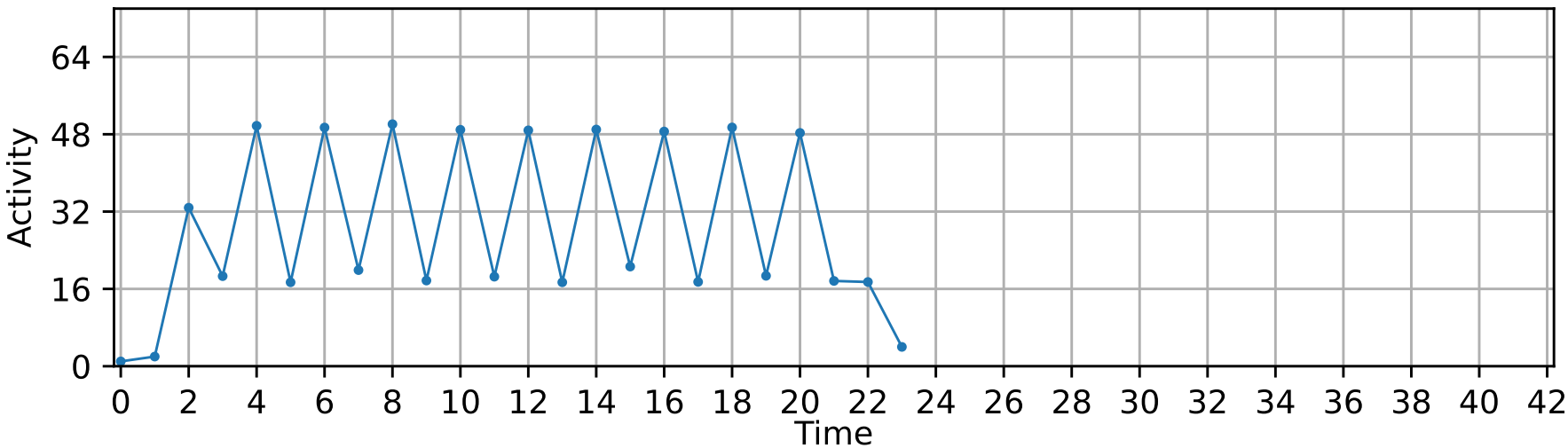
Processor Pipeline (2/4)

```
// 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
```



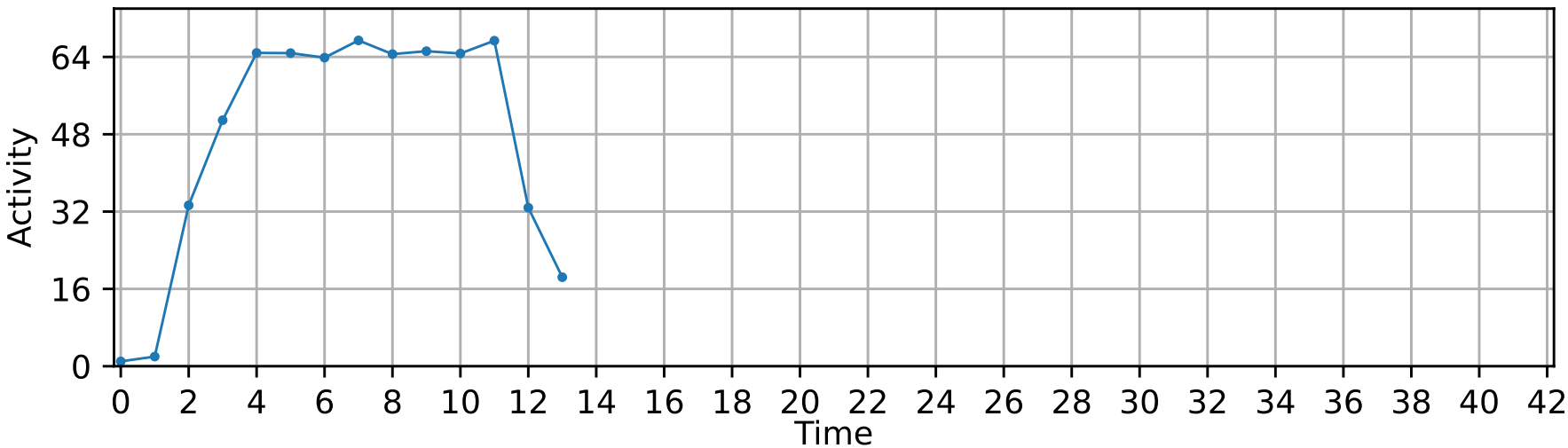
Processor Pipeline (3/4)

```
// 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
```



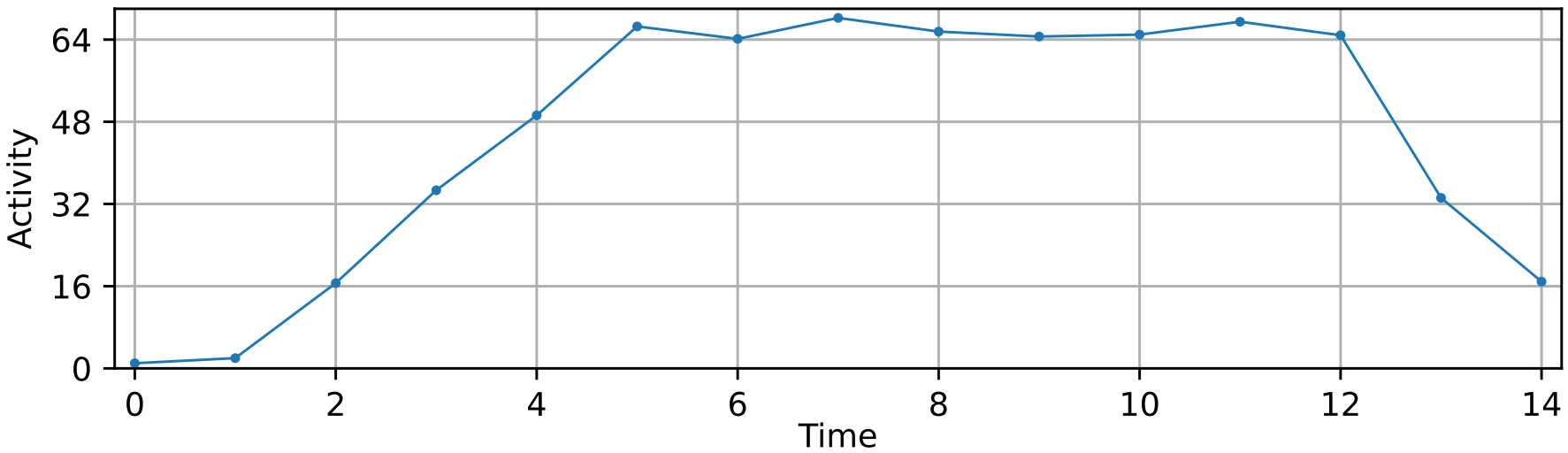
Processor Pipeline (4/4)

```
// 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
```



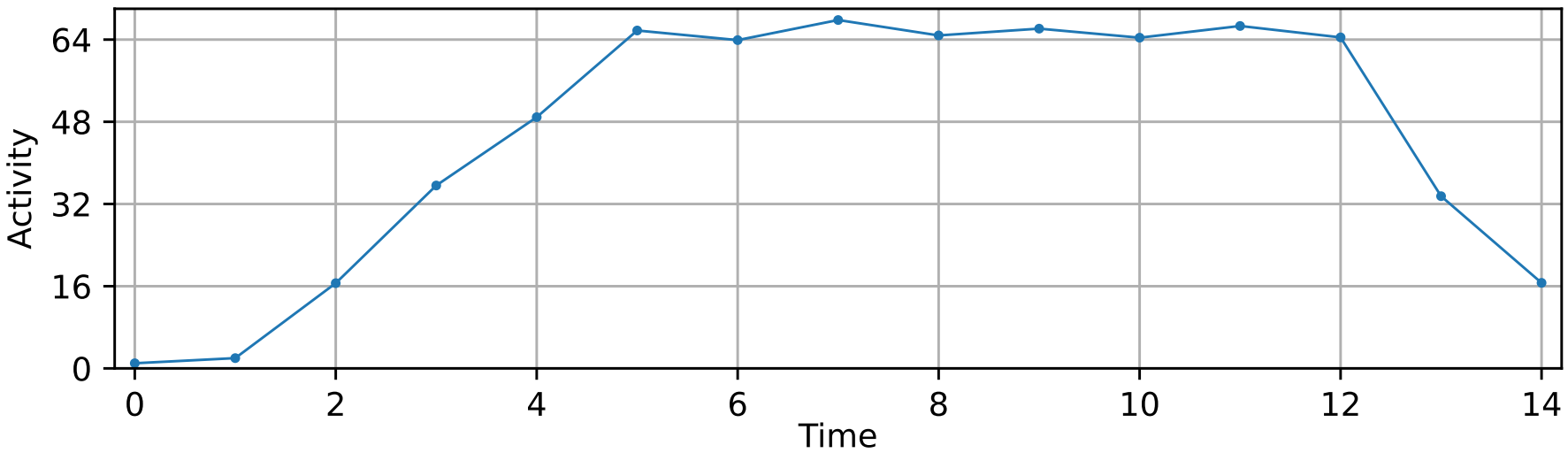
Add 0


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



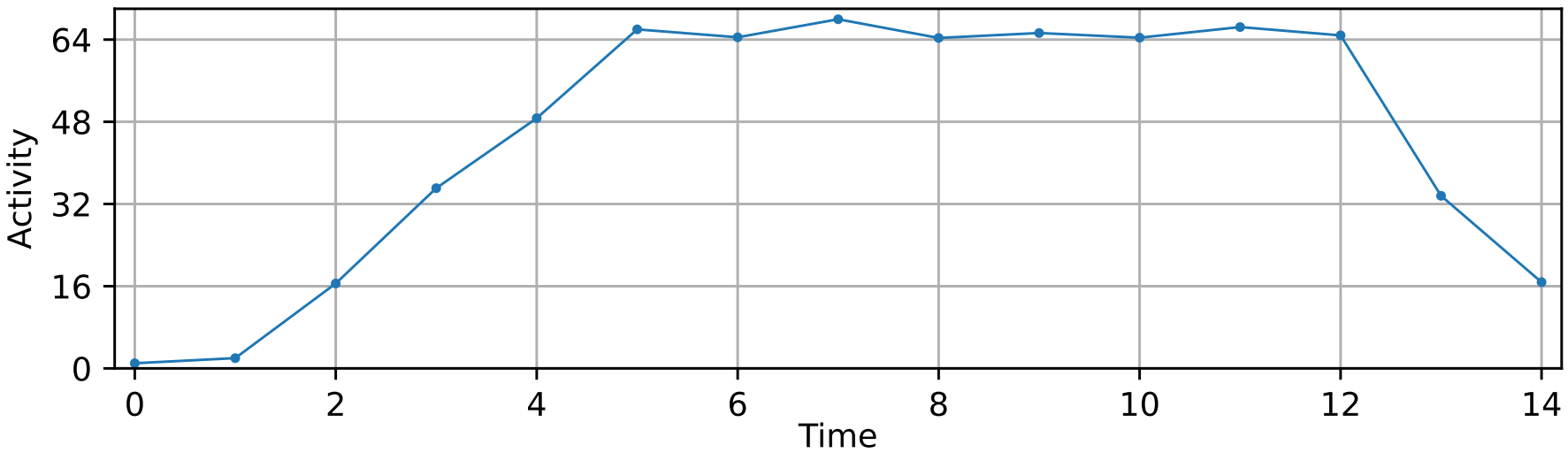
Add 1


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

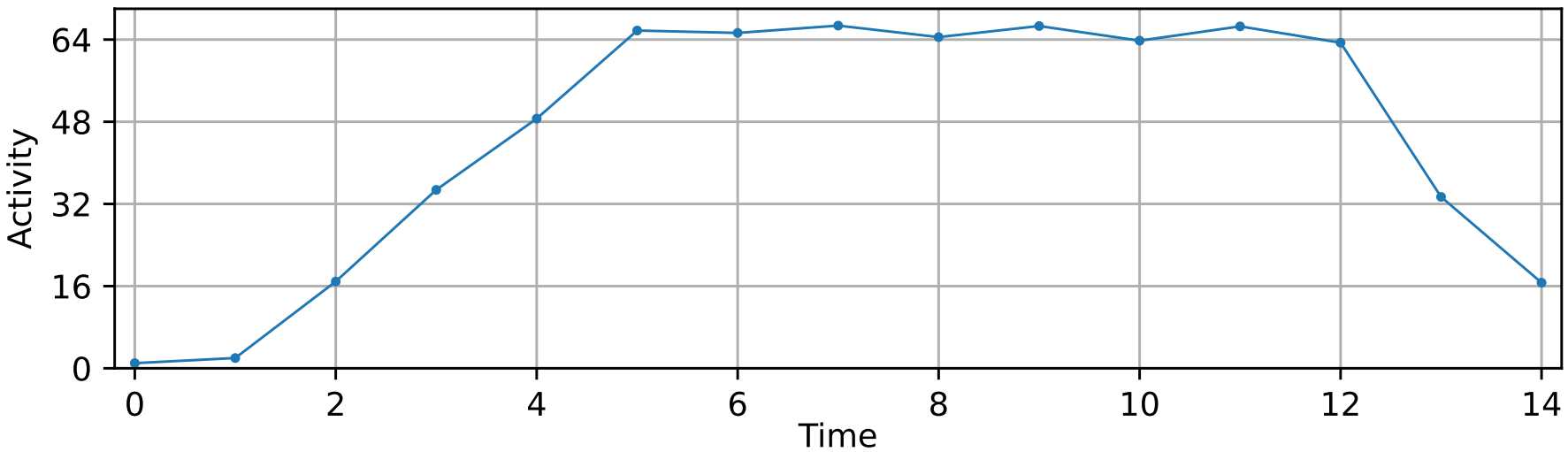


Add -1

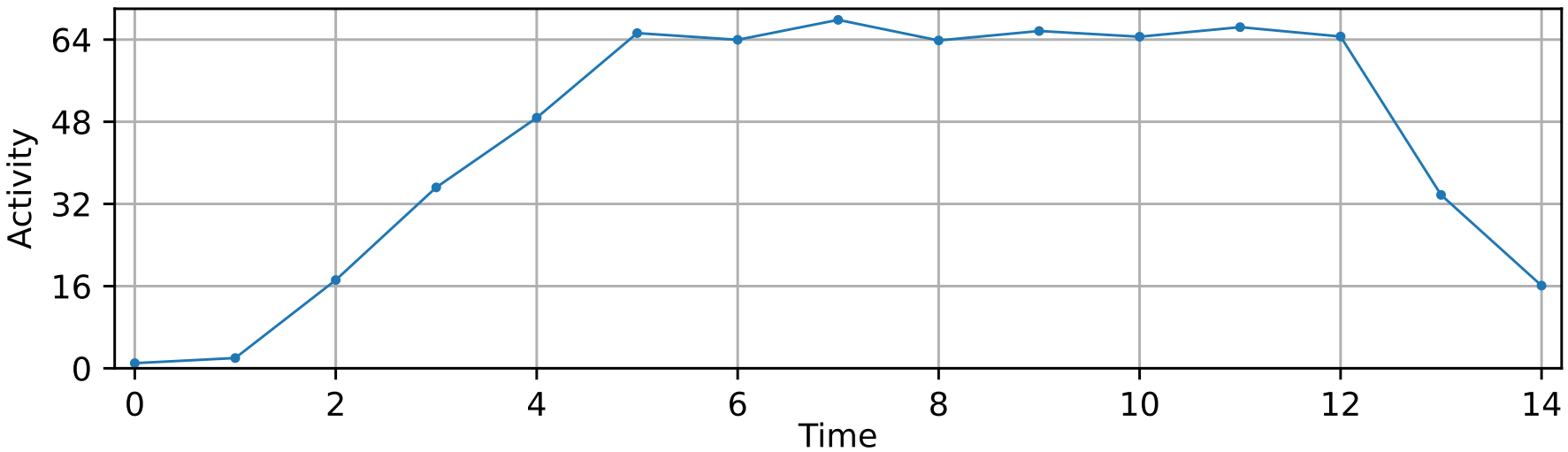

```
 0    NOP
 1    SET  R21, -1
 2    ADD  R1, R2, R3
 3    ADD  R4, R5, R6
 4    ADD  R7, R8, R9
 5    ADD  R10, R11, R12
 6    ADD  R13, R14, R15
 7    ADD  R16, R17, R18
 8    ADD  R19, R20, R21        // - 1
 9    ADD  R22, R23, R24
10    ADD  R25, R26, R27
11    ADD  R28, R29, R30
```



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

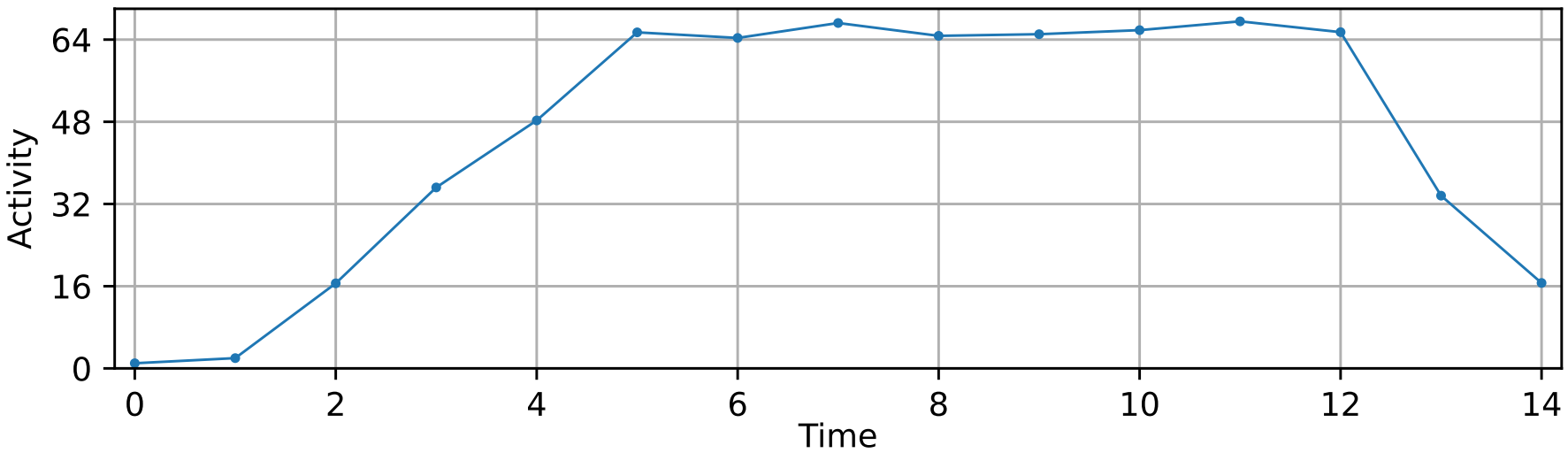


```
 0    NOP
 1    SET  R21, 1
 2    ADD  R1, R2, R3
 3    ADD  R4, R5, R6
 4    ADD  R7, R8, R9
 5    ADD  R10, R11, R12
 6    ADD  R13, R14, R15
 7    ADD  R16, R17, R18
 8    MUL  R19, R20, R21        // * 1
 9    ADD  R22, R23, R24
10    ADD  R25, R26, R27
11    ADD  R28, R29, R30
```

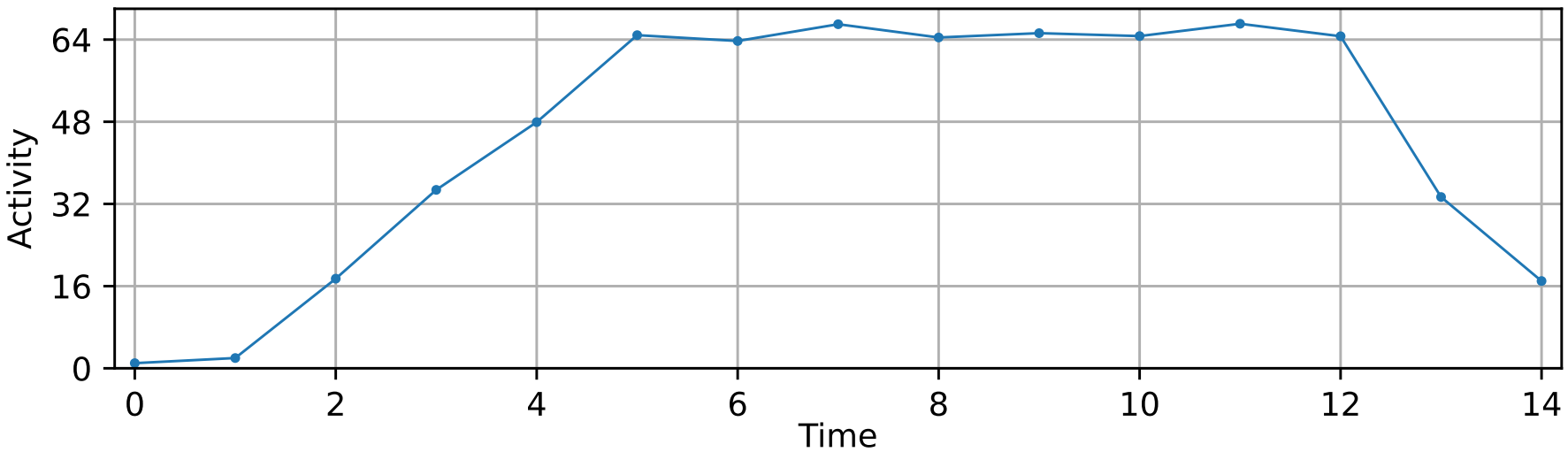


Mul -1


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

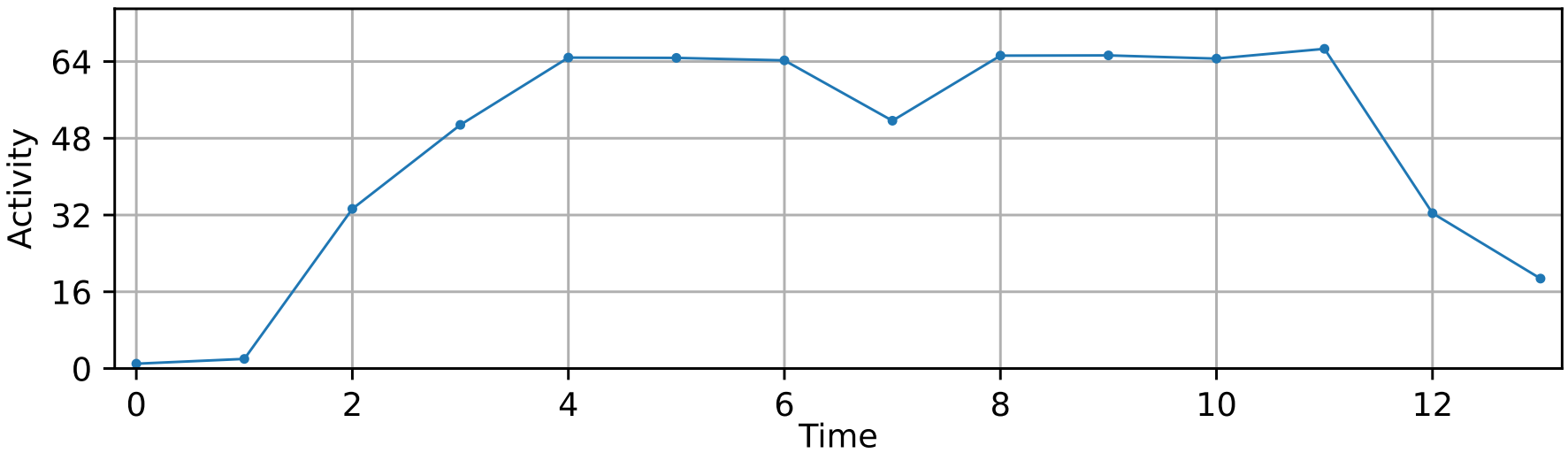


```
 0    NOP
 1    SET  R21, 17
 2    ADD  R1, R2, R3
 3    ADD  R4, R5, R6
 4    ADD  R7, R8, R9
 5    ADD  R10, R11, R12
 6    ADD  R13, R14, R15
 7    ADD  R16, R17, R18
 8    MUL  R19, R20, R21        // * 17
 9    ADD  R22, R23, R24
10    ADD  R25, R26, R27
11    ADD  R28, R29, R30
```



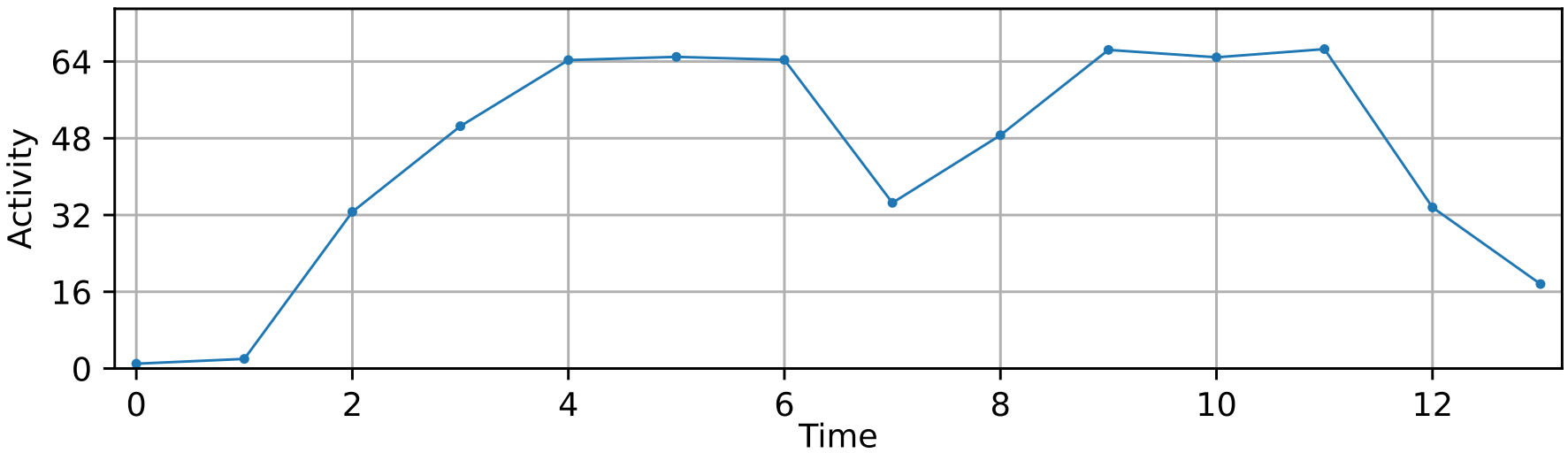
Register Reuse (1/3)

```
// 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
```



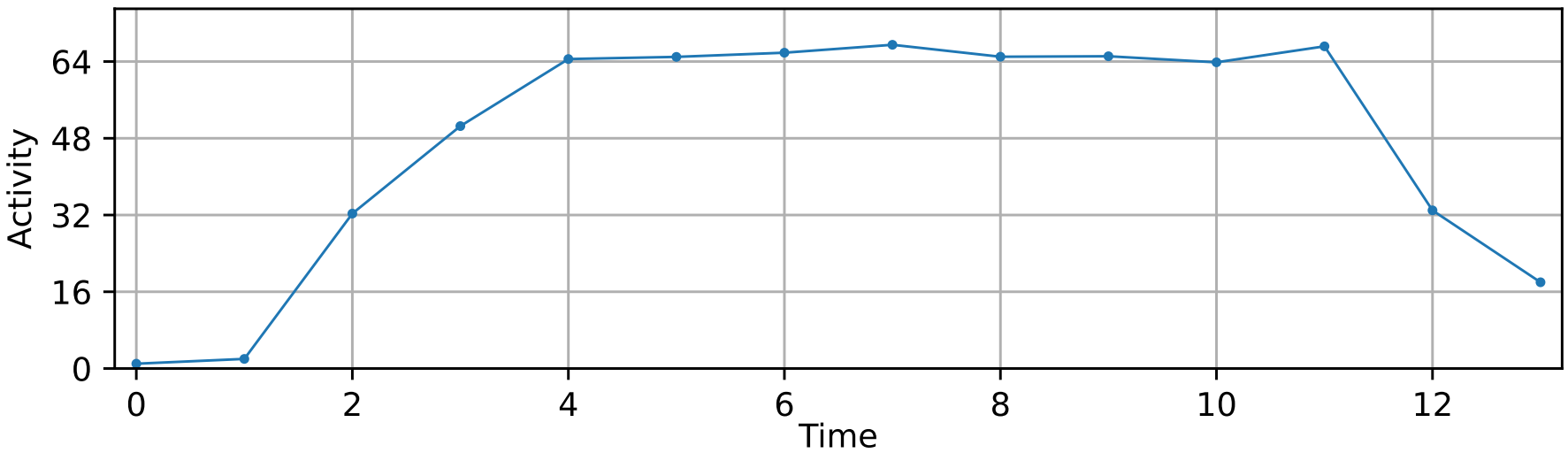
Register Reuse (2/3)

```
// 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
```



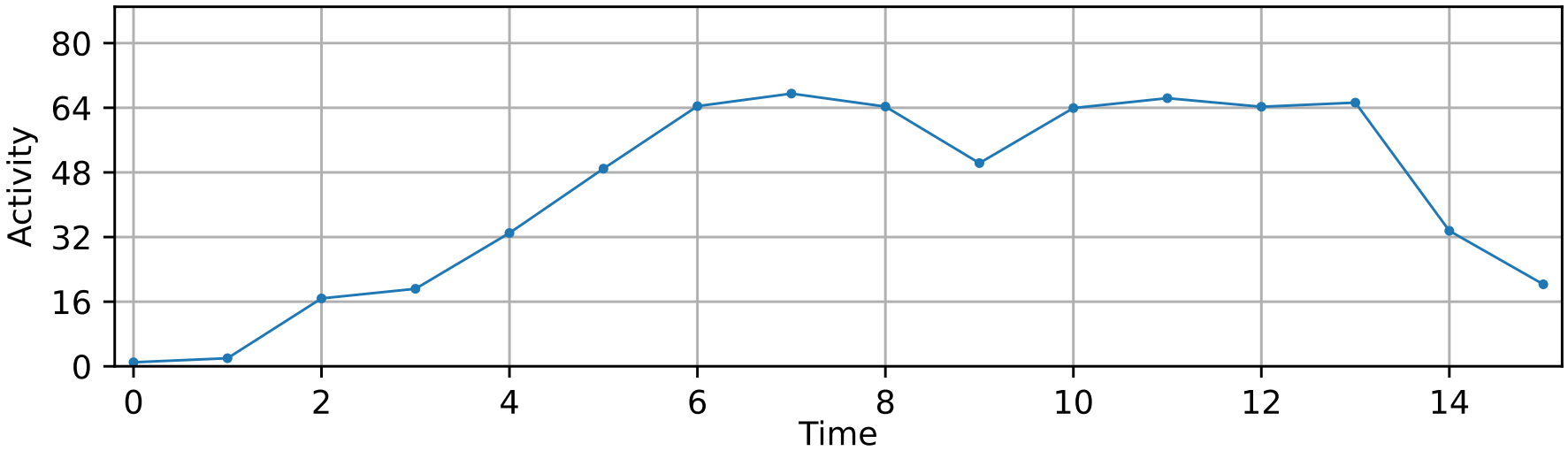
Register Reuse (3/3)

```
// 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
```



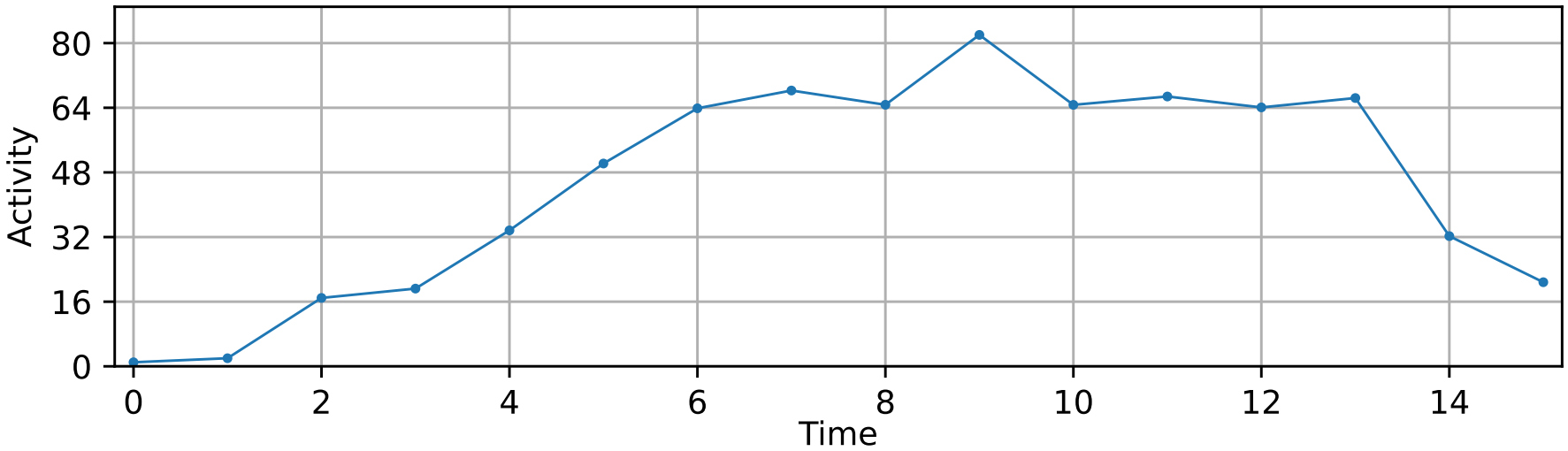
Close Operands (1/5)


```
 0   NOP
 1   SET  R15, 0
 2   SET  R18, 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      // rnd + 1
 9   ADD  R19, R20, R21
10   ADD  R22, R23, R24
11   ADD  R25, R26, R27
12   ADD  R28, R29, R30
```



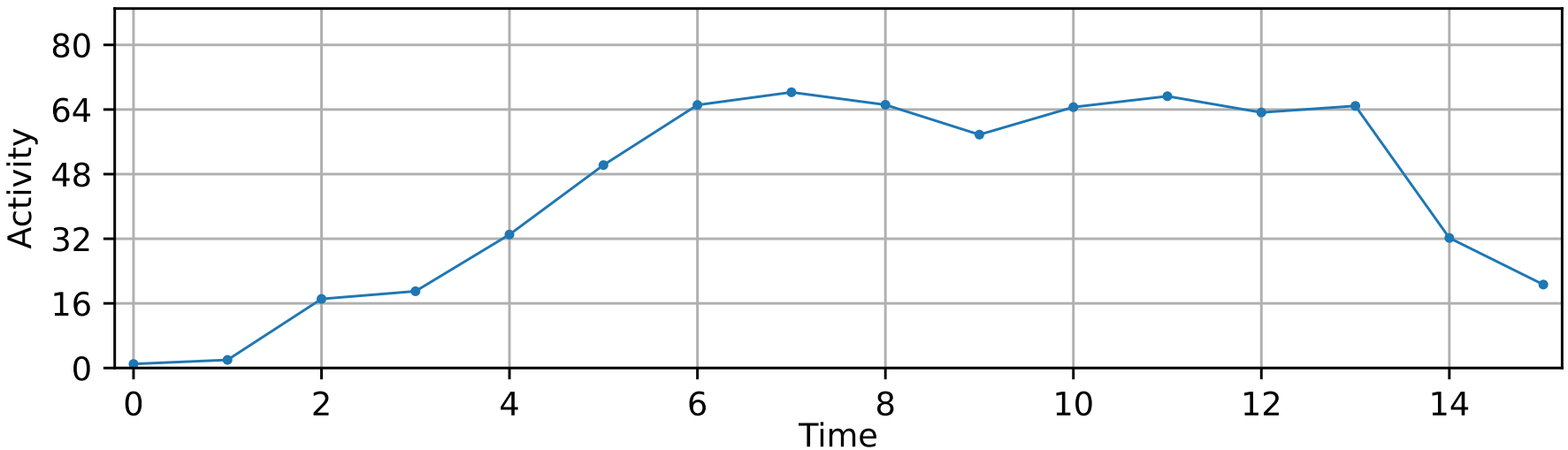
Close Operands (2/5)


```
 0   NOP
 1   SET  R15, 0
 2   SET  R18, -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      // rnd + -1
 9   ADD  R19, R20, R21
10   ADD  R22, R23, R24
11   ADD  R25, R26, R27
12   ADD  R28, R29, R30
```



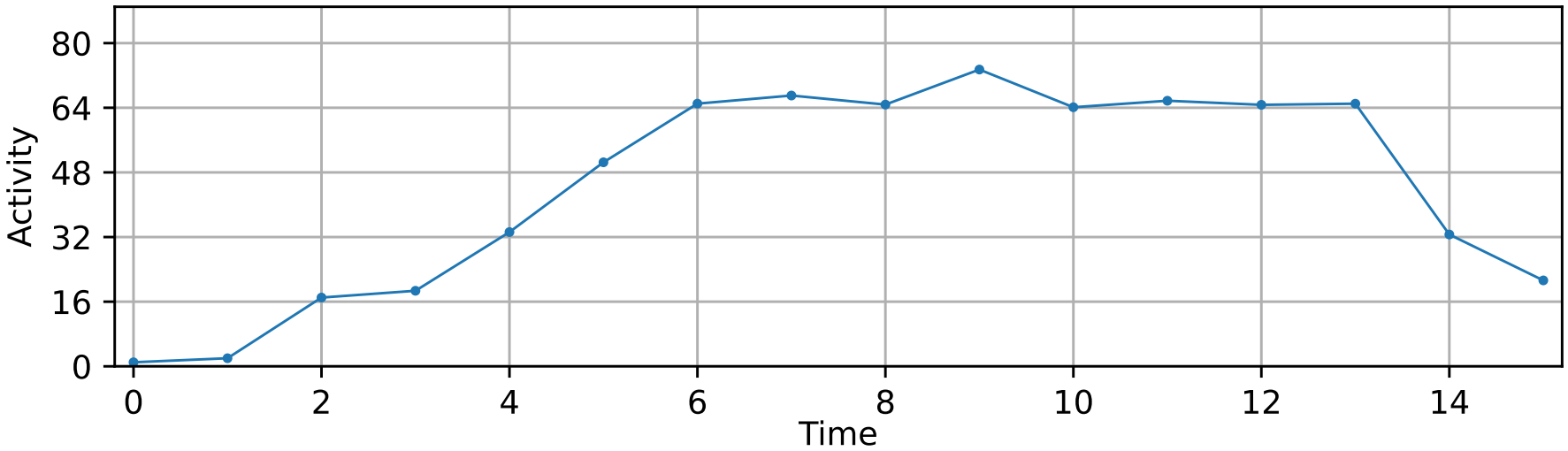
Close Operands (3/5)


```
 0   NOP
 1   SET  R15, 0
 2   SET  R18, 255
 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      // rnd + 255
 9   ADD  R19, R20, R21
10   ADD  R22, R23, R24
11   ADD  R25, R26, R27
12   ADD  R28, R29, R30
```



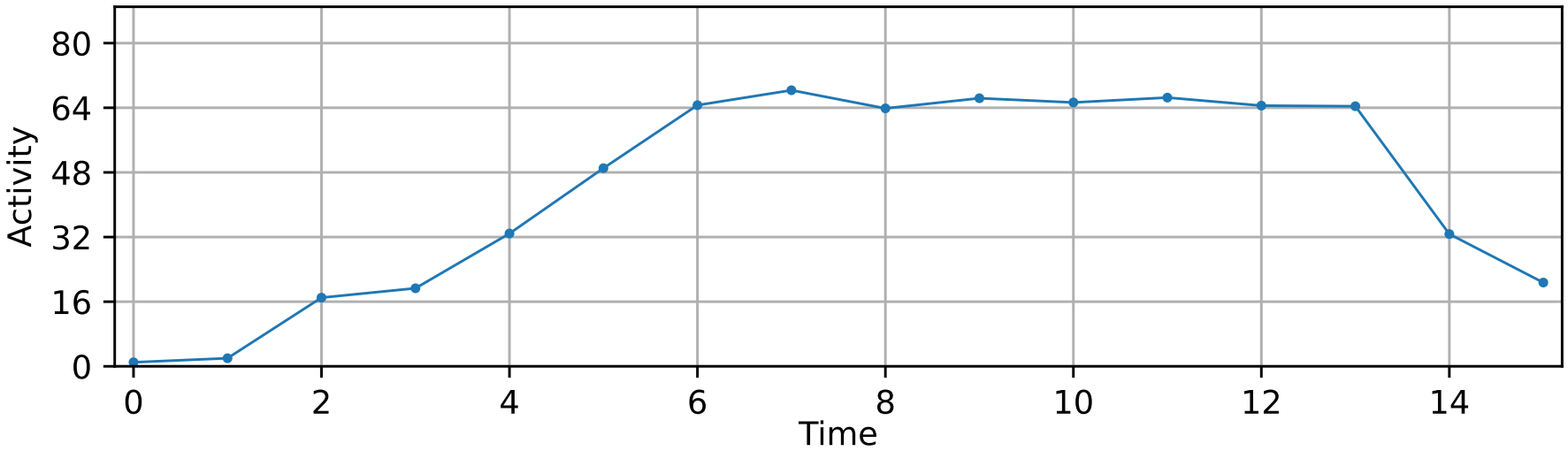
Close Operands (4/5)


```
 0   NOP
 1   SET  R15, 0
 2   SET  R18, 16777215      // 2^24 - 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      // rnd + 2^24-1
 9   ADD  R19, R20, R21
10   ADD  R22, R23, R24
11   ADD  R25, R26, R27
12   ADD  R28, R29, R30
```



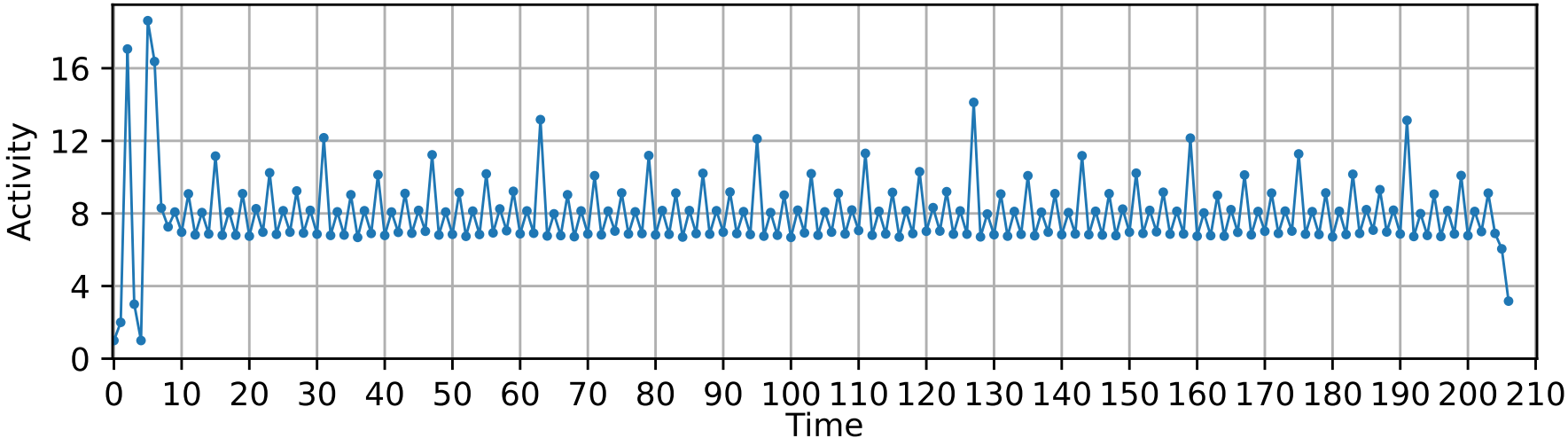
Close Operands (5/5)


```
 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
```



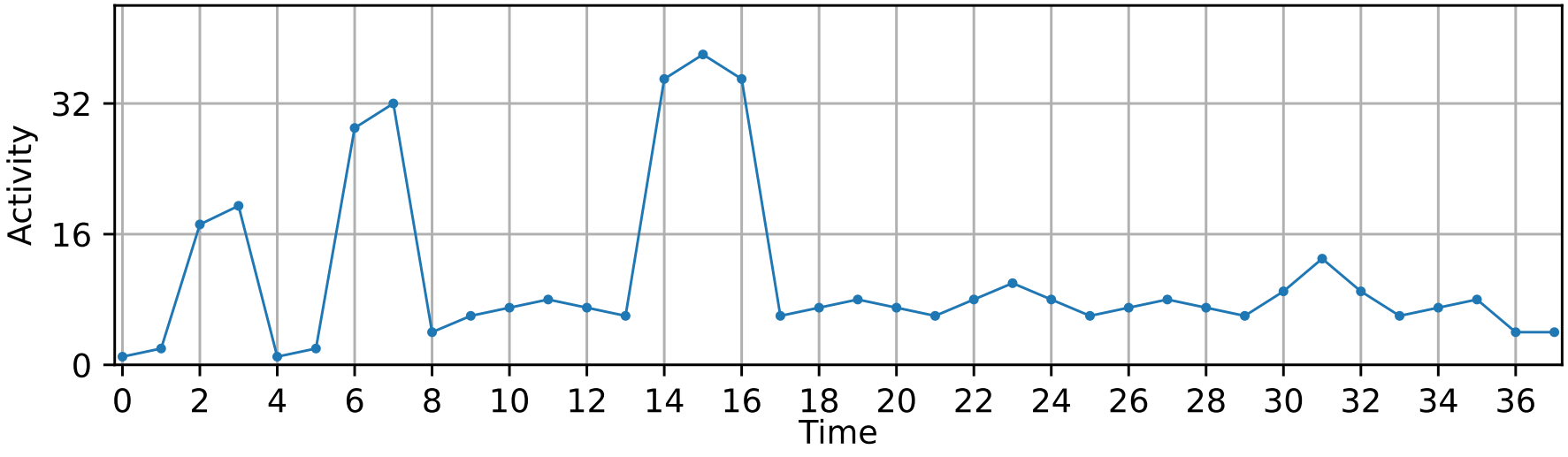
Counter (1/3)


```
// 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
 ...
```



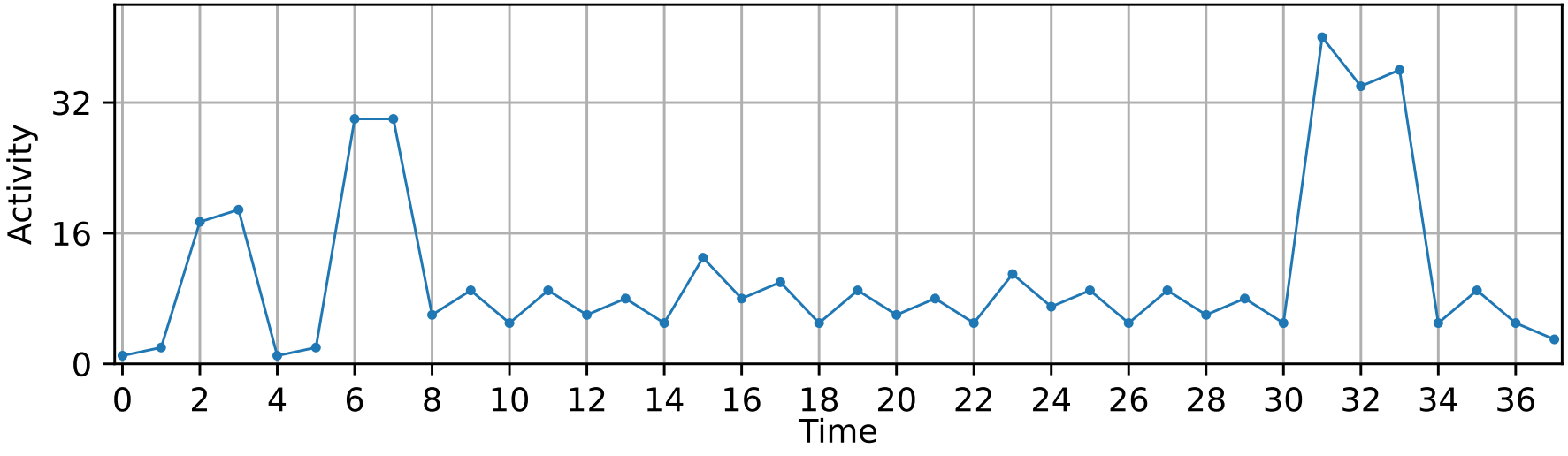
Counter (2/3)

```
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
...
```



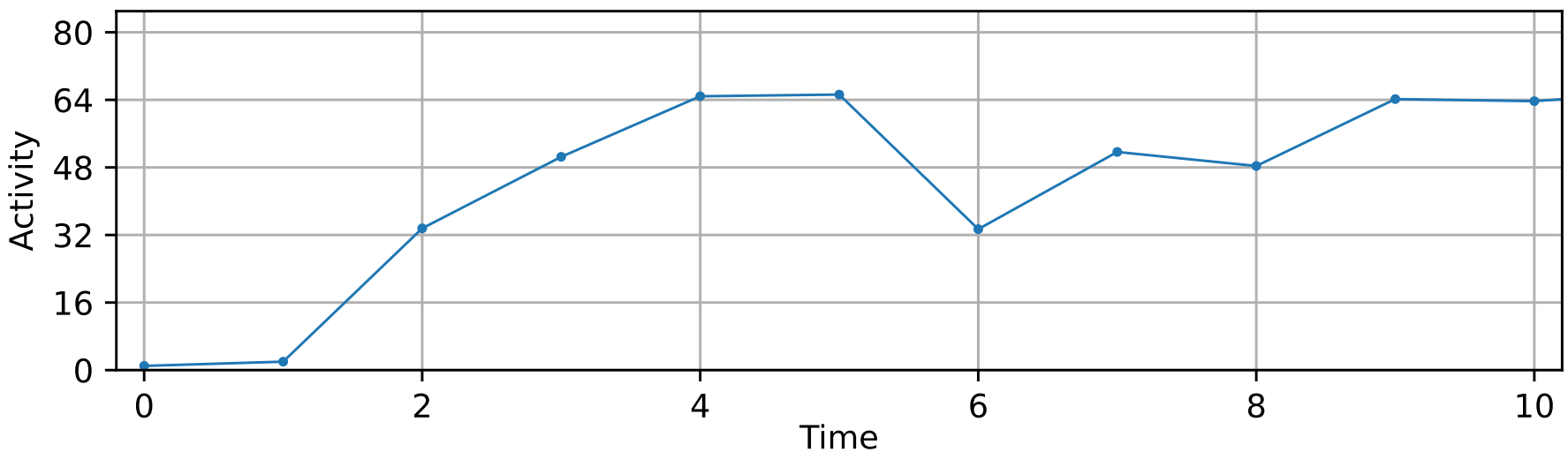
Counter (3/3)

```
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
...
```



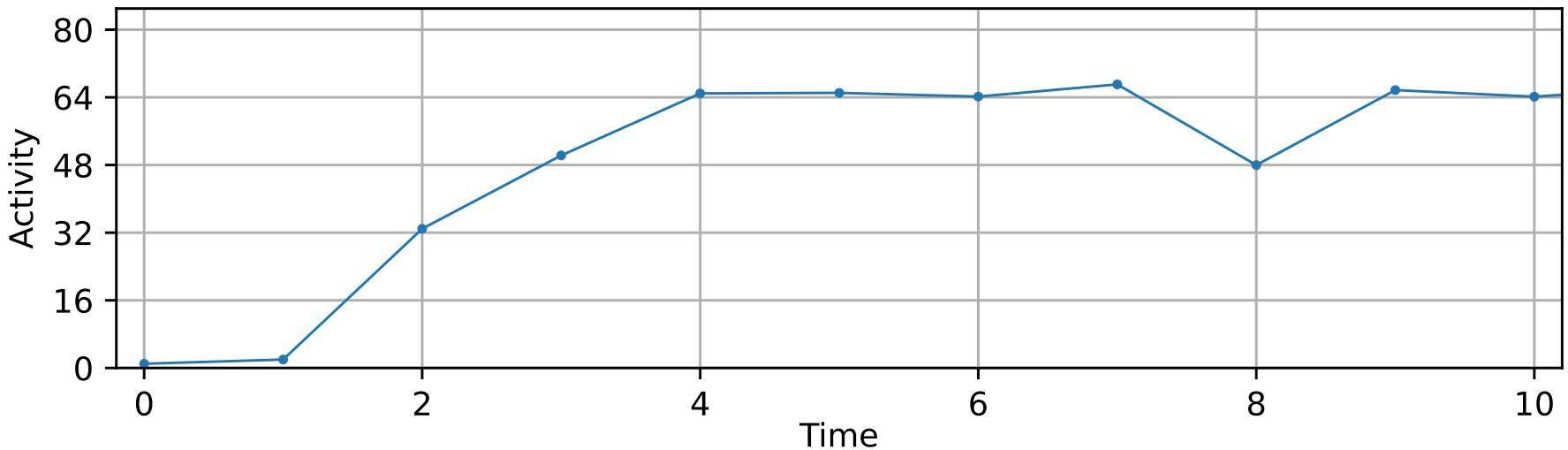
NOP Impact (1/5)


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



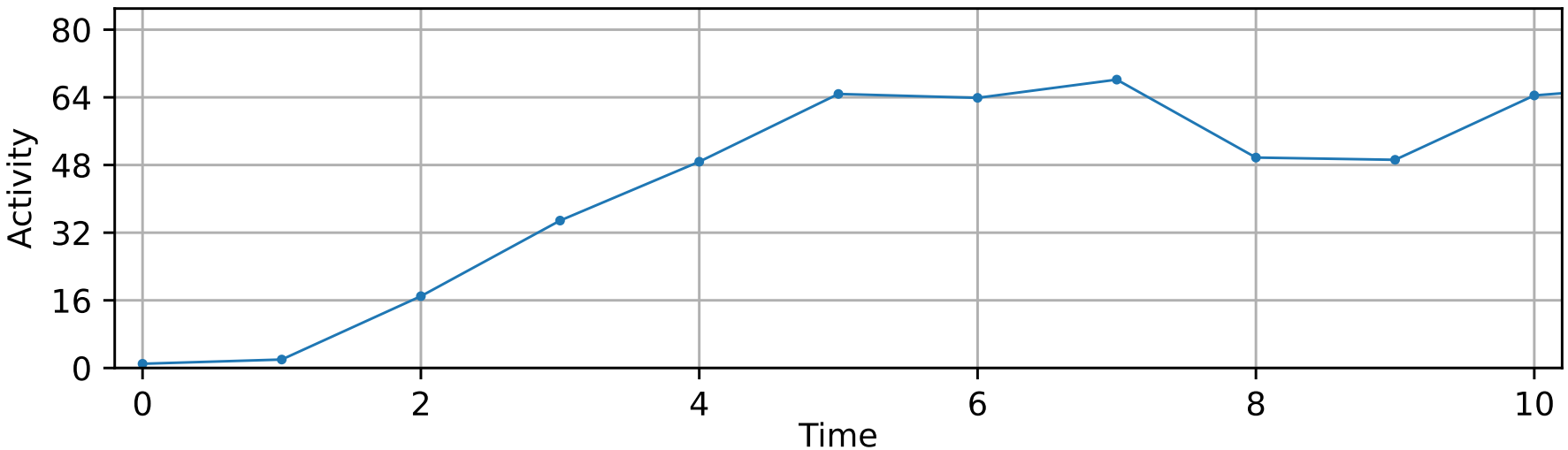
NOP Impact (2/5)


```
//
 0    NOP
 1    ADD  R1, R2, R3
 2    ADD  R4, R5, R6
 3    ADD  R7, R8, R9
 4    ADD  R10, R11, R12
 5    NOP                          // NOP = pseudo-inst  ADD R0, R0, 0
 6    ADD  R13, R14, R15
 7    ADD  R16, R17, R18
 8    ADD  R19, R20, R21
 9    ADD  R22, R23, R24
10    ADD  R25, R26, R27
11    ADD  R28, R29, R30
```



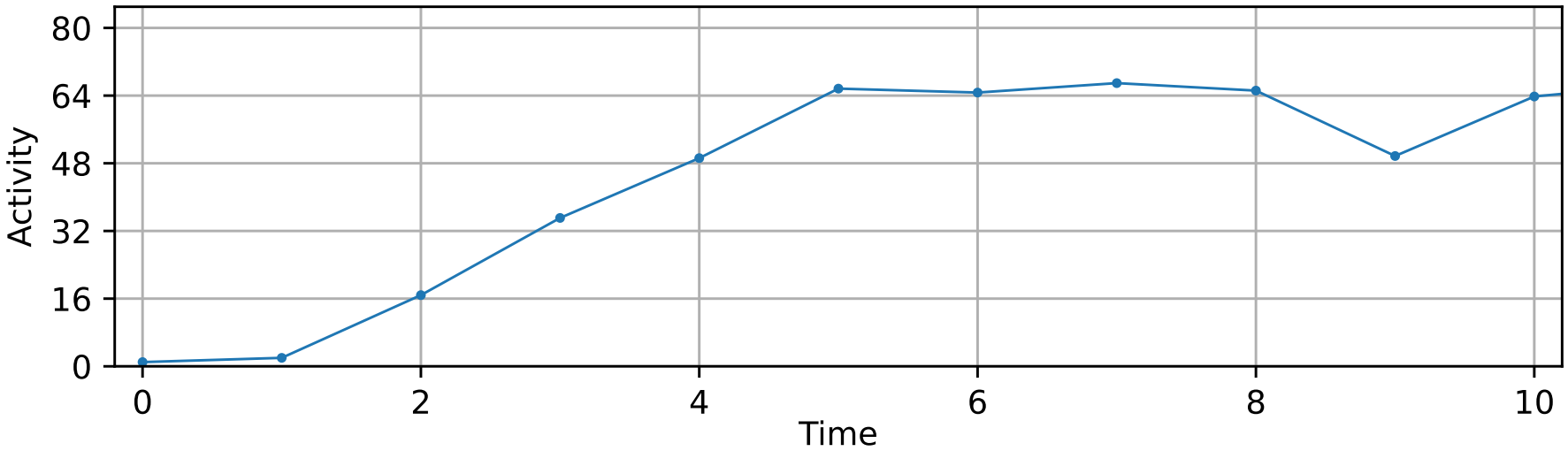
NOP Impact (3/5)


```
 0    NOP
 1    SET  R14, 1                  // 1
 2    ADD  R1, R2, R3
 3    ADD  R4, R5, R6
 4    ADD  R7, R8, R9
 5    ADD  R10, R11, R12
 6    NOP                          // NOP = pseudo-inst  ADD R0, R0, 0
 7    ADD  R13, R14, R15
 8    ADD  R16, R17, R18
 9    ADD  R19, R20, R21
10    ADD  R22, R23, R24
11    ADD  R25, R26, R27
12    ADD  R28, R29, R30
```



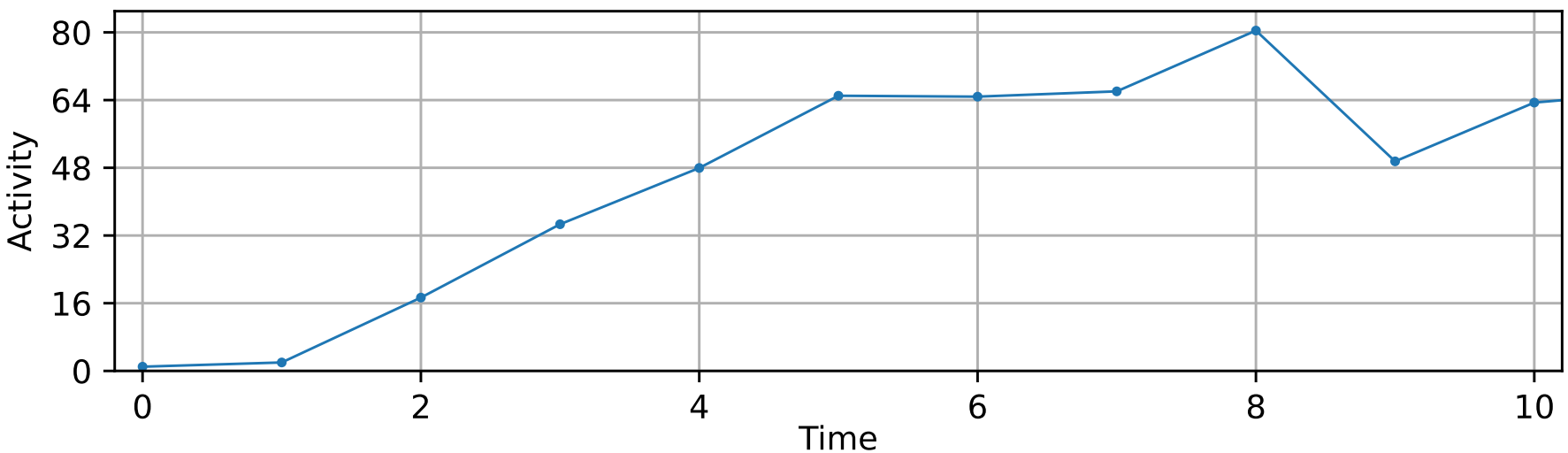
NOP Impact (4/5)


```
 0   NOP
 2   ADD  R1, R2, R3
 1   SET  R14, 65535          // 2^16 - 1
 3   ADD  R4, R5, R6
 4   ADD  R7, R8, R9
 5   ADD  R10, R11, R12
 6   NOP                      // NOP = pseudo-inst  ADD R0, R0, 0
 7   ADD  R13, R14, R15
 8   ADD  R16, R17, R18
 9   ADD  R19, R20, R21
10   ADD  R22, R23, R24
11   ADD  R25, R26, R27
12   ADD  R28, R29, R30
```



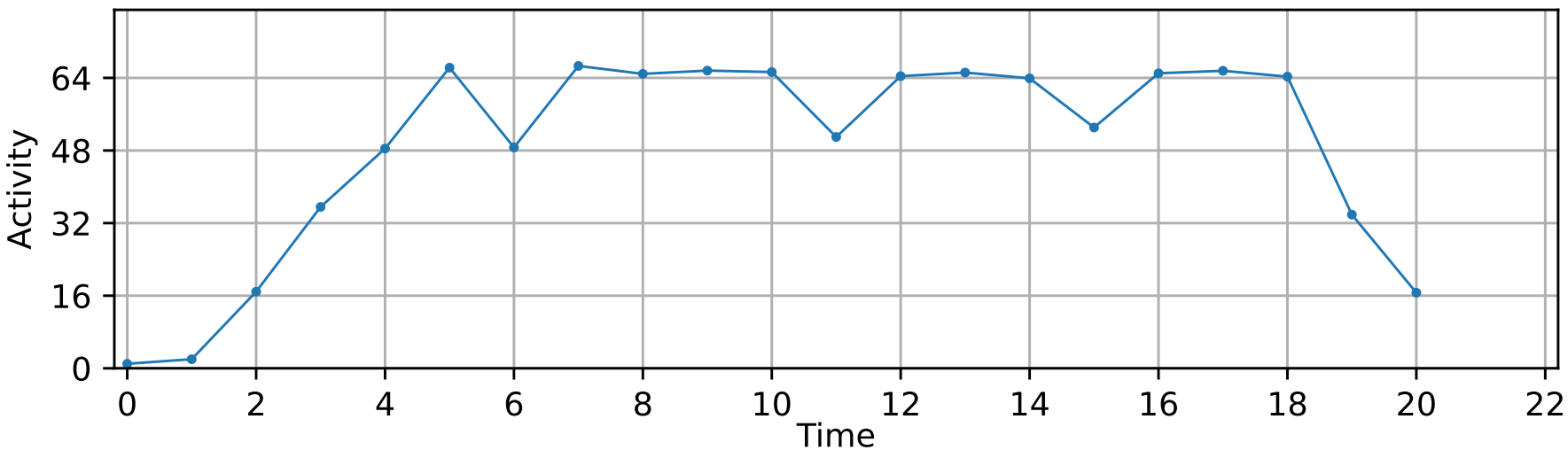
NOP Impact (5/5)


```
 0   NOP
 1   SET  R14, 4294967295     // 2^32 - 1
 2   ADD  R1, R2, R3
 3   ADD  R4, R5, R6
 4   ADD  R7, R8, R9
 5   ADD  R10, R11, R12
 6   NOP                      // NOP = pseudo-inst  ADD R0, R0, 0
 7   ADD  R13, R14, R15
 8   ADD  R16, R17, R18
 9   ADD  R19, R20, R21
10   ADD  R22, R23, R24
11   ADD  R25, R26, R27
12   ADD  R28, R29, R30
```



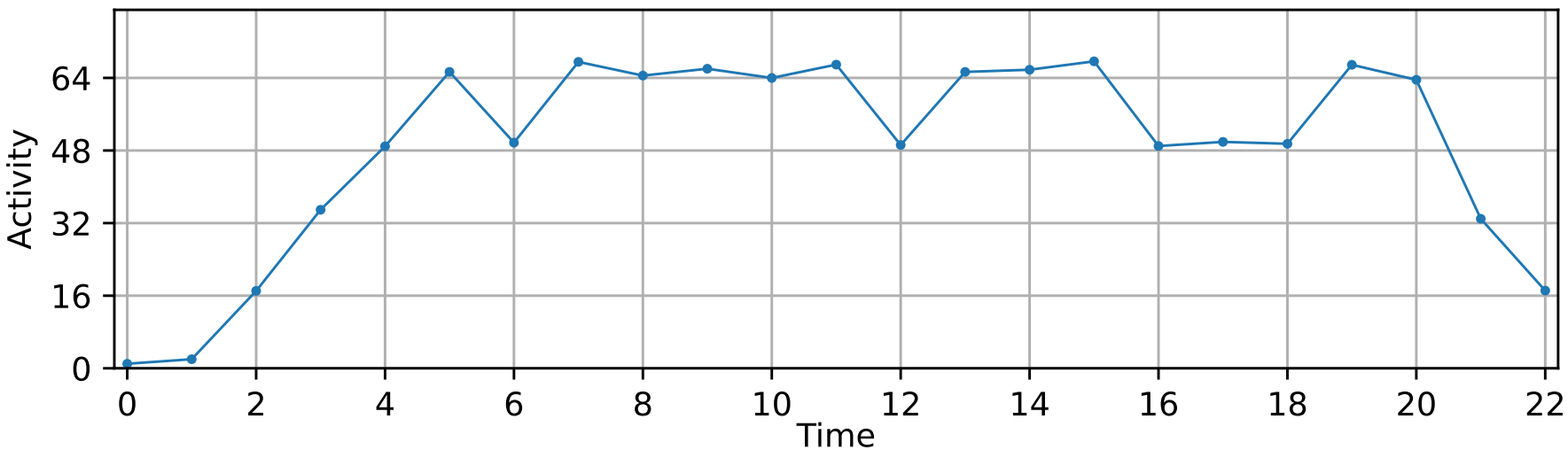
Comparison (1/2)


```
 1   SET  R30, 0
 2   ADD  R10, R26, R27       // R10
 3   ADD  R11, R28, R29       // R11 != R10
 4   ADD  R12, R11, R30       // R12 = R11
 5   ADD  R1, R2, R3
 6   ADD  R4, R5, R6
 7   ADD  R7, R8, R9
 8   CMP  R10, R11            // diff
 9   ADD  R13, R14, R15
10   ADD  R16, R17, R18
11   ADD  R19, R20, R21
12   CMP  R11, R12            // eq
13   ADD  R22, R23, R24
14
15
16
```



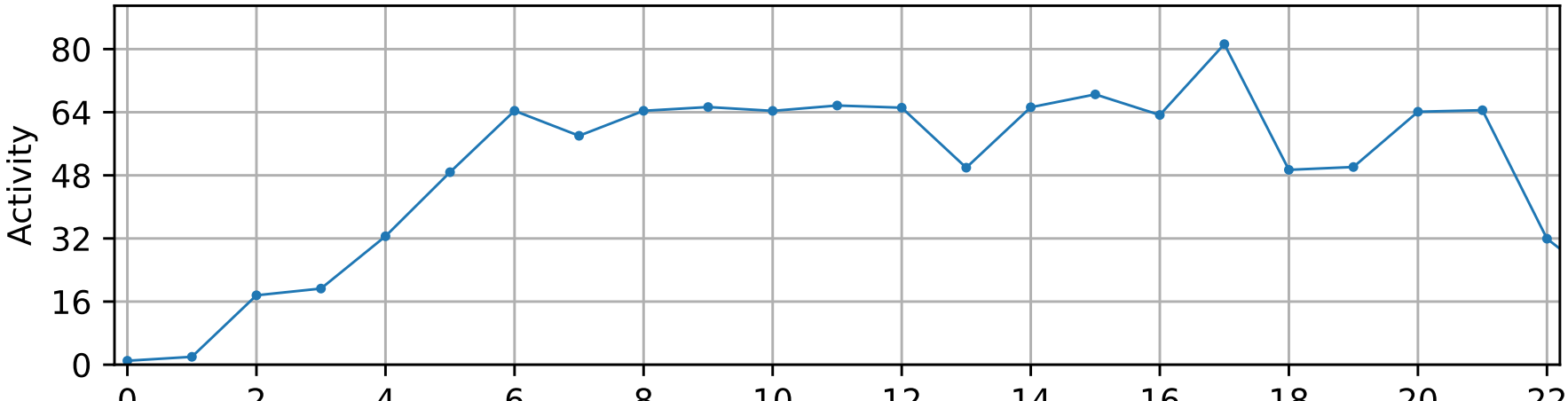
Comparison (2/2)


```
 1   SET  R30, 0
 2   ADD  R10, R26, R27       // R10
 3   ADD  R11, R28, R29       // R11 != R10
 4   ADD  R12, R11, R30       // R12 = R11
 5   ADD  R1, R2, R3
 6   ADD  R4, R5, R6
 7   ADD  R7, R8, R9
 8   MUL  R28, R2, R30        // * 0
 9   CMP  R10, R11            // diff
10   ADD  R13, R14, R15
11   ADD  R16, R17, R18
12   ADD  R19, R20, R21
13   MUL  R27, R2, R30        // * 0
14   CMP  R11, R12            // eq
15   ADD  R22, R23, R24
16
```



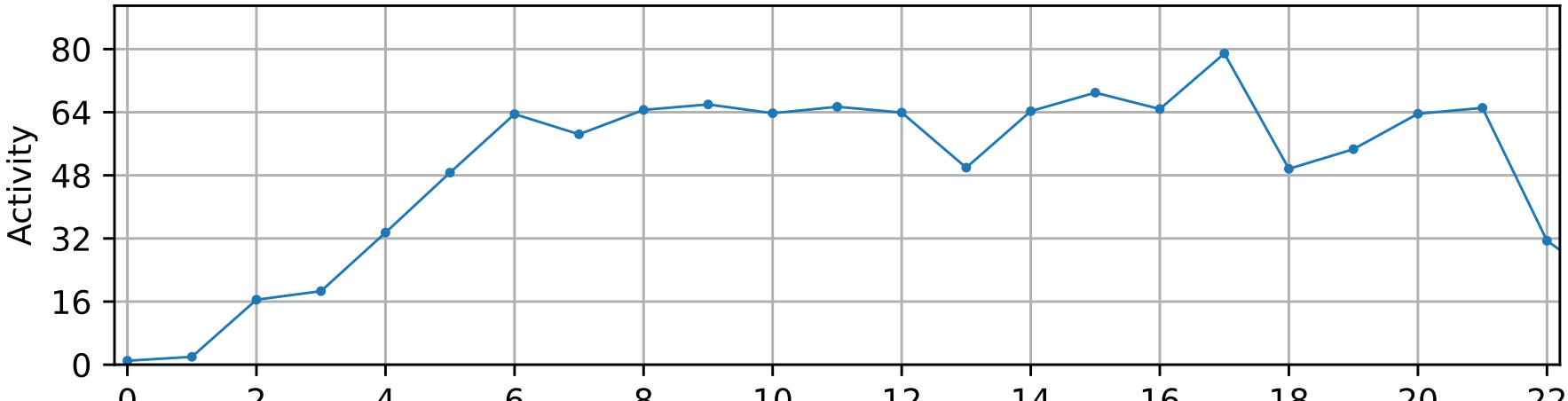
Comparison and Specific Values (1/2)


```
 0   NOP
 1   SET  R30, 8380417   // P = 2^23 - 2^13 +1
 2   SET  R2, 16760834   // 2 * P
 3   ADD  R10, R26, R27  // R10
 4   ADD  R11, R28, R29  // R11 != R10
 5   ADD  R12, R11, R30  // R12 = R11
 6   ADD  R1, R2, R3
 7   ADD  R4, R5, R6
 8   ADD  R7, R8, R9
 9   SUB  R28, R2, R30   // 2P - P
10   CMP  R10, R11       // diff
11   ADD  R13, R14, R15
12   ADD  R16, R17, R18
13   ADD  R19, R20, R21
14   SUB  R27, R2, R30   // 2P - P
15   CMP  R11, R12       // eq
16   ADD  R22, R23, R24
```



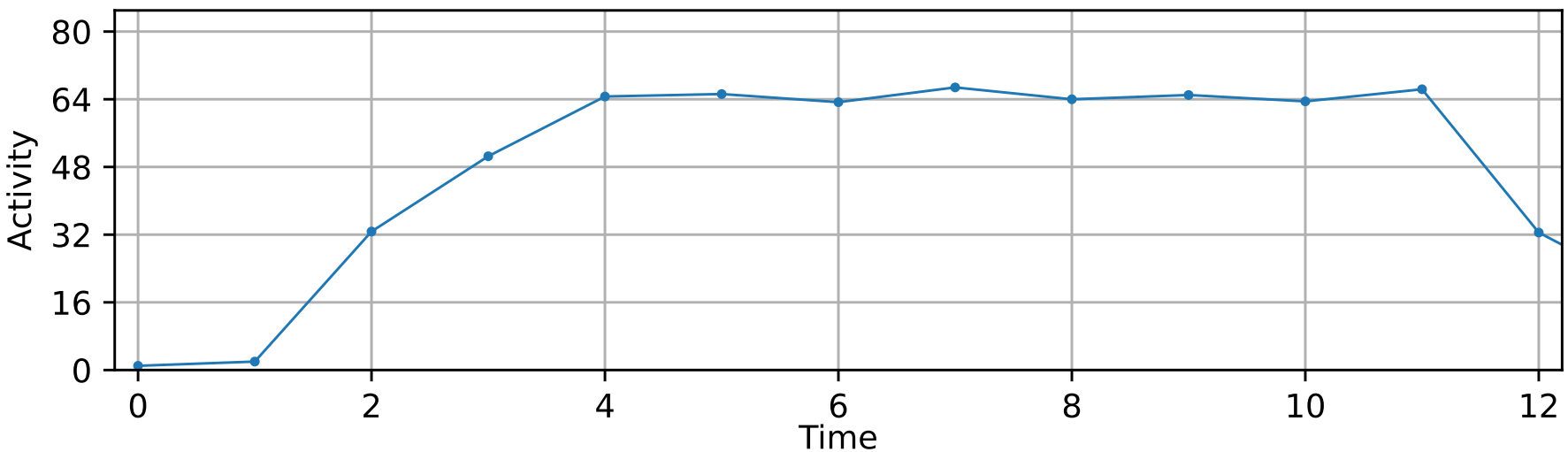
Comparison and Specific Values (2/2)


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



Square vs Multiply Instructions (1/2)

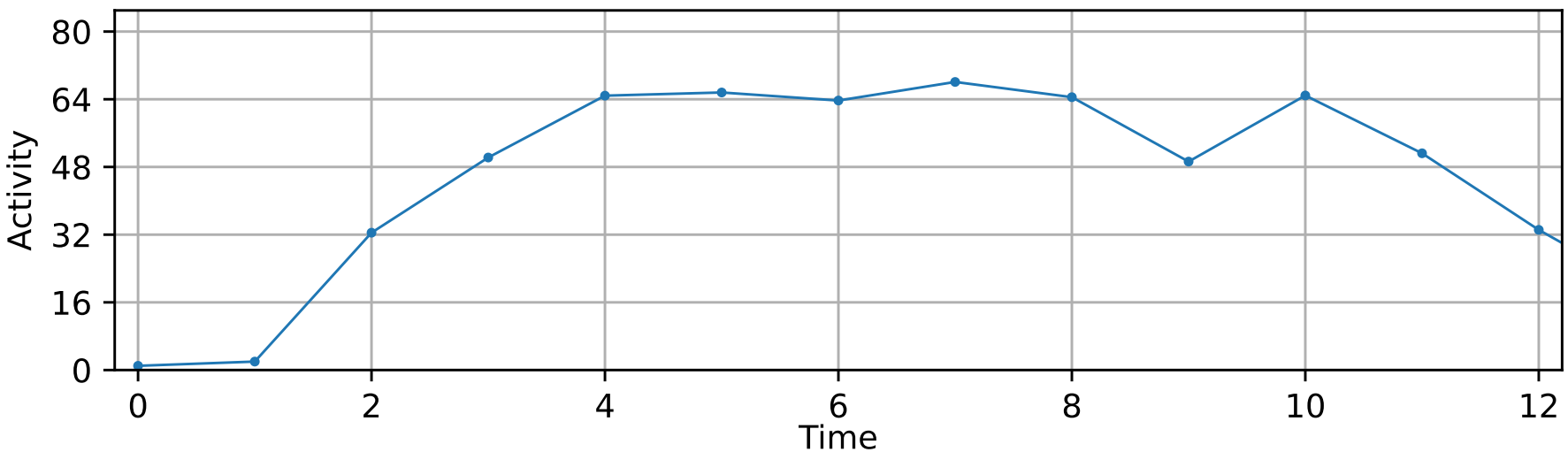

```
 0    NOP
 1    ADD   R1, R2, R3
 2    ADD   R4, R5, R6
 3    ADD   R7, R8, R9
 4    MUL   R10, R11, R12       // MUL
 5    ADD   R13, R14, R15
 6    ADD   R16, R17, R18
 7    ADD   R19, R20, R21
 8    MUL   R22, R23, R23       // SQR using MUL
 9    ADD   R25, R26, R27
10    ADD   R28, R29, R30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



Remark: I'm not talking about "square and multiply" like algorithms (which are weak w.r.t simple power/EM analysis), but support of instructions in the ISA

Square vs Multiply Instructions (2/2)


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



Remark: I'm not talking about "square and multiply" like algorithms (which are weak w.r.t simple power/EM analysis), but support of instructions in the ISA

The sequence of executed instructions and control flow must **not** depend on secret values

Yes, this is important even critical! (or your system is very weak)

BUT constant time avoids timing attacks (when the micro-architecture cooperates), but NOT all side channel attacks!

In some cases, it helps attackers for power or EM analysis. . .

- 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 (no way, too abstract!)
  - code (no way, still too abstract!)
  - assembly (not sufficient)
  - binary (not sufficient)
  - assembly + micro-architecture (maybe?)
- Compilers (and CAD tools) optimizations can remove some protection "tricks"
- Optimizations to reduce energy can be counter-productive

Arithmetic aspect**S** (also) impact electrical activity and then security, so we need more works:

- Modeling links between arithmetic (representations, algorithms) and electrical properties
- Selection/design of appropriate algorithms/implementations
- Secure arithmetic gadgets (and secure composition rules)
- What if speed is not the main target?
- Take into account parasitic activity (not simple)
- "Calibration" of models from implementation**S** result**S**
- Challenge: design arithmetic protections against observation *and* perturbation attacks
- Teaching sessions

Thank you! Questions?