A Hardware Design Methodology to Prevent Microarchitectural Transition Leakages

April 3rd 2025

Mathieu Escouteloup[1], **Vincent Migliore**[2]
1. Bordeaux University, IMS Lab, France 2. Toulouse University, LAAS-CNRS, France

Context
Overview

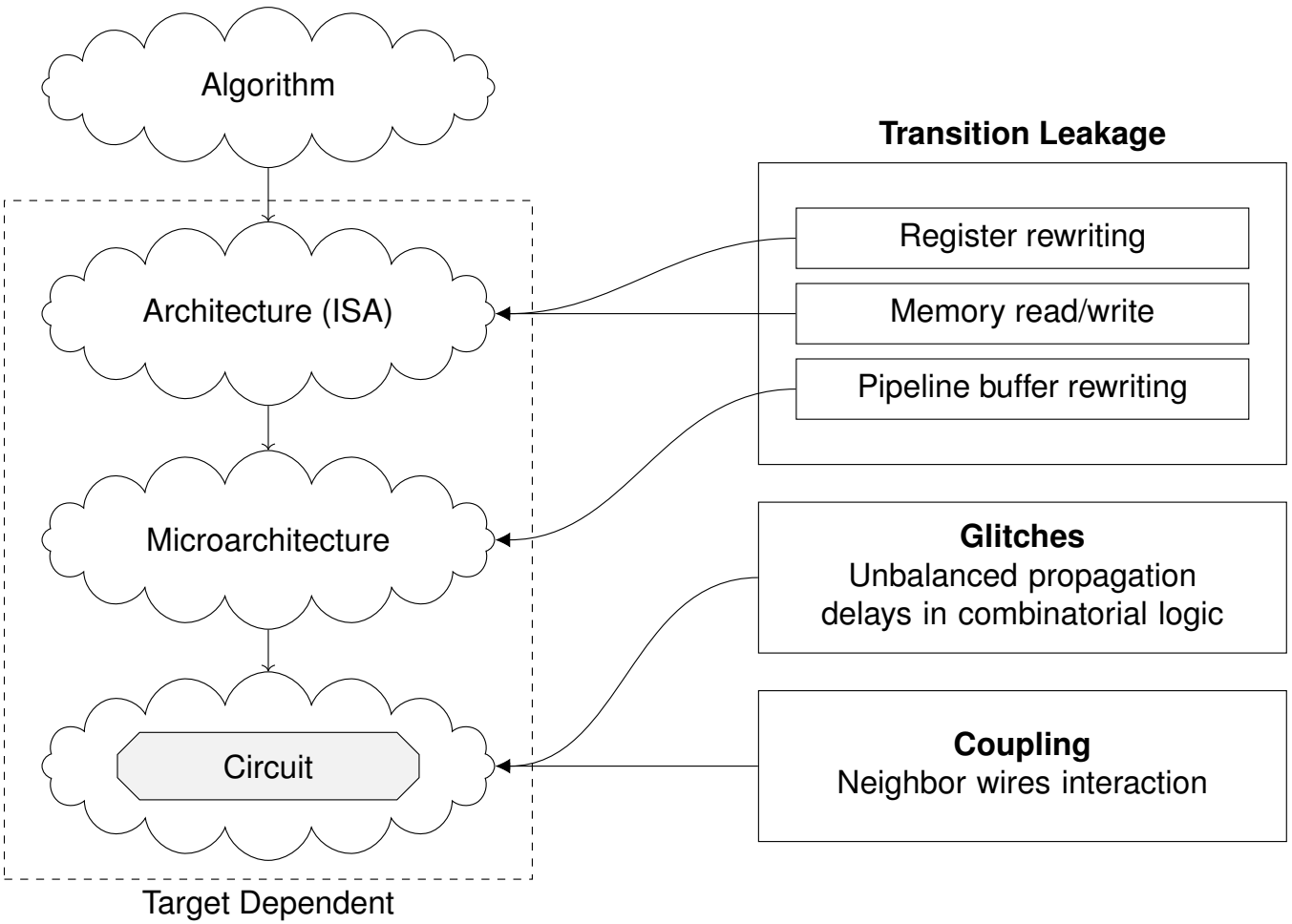
Cyberphysical and connected objets have been widely adopted to address some important societal challenges. They are the subject of two trends:

1. **Increasing complexity**: typically following System On Chip (SoC) architectures.
2. **Increasing attack surface**: with hardware-oriented attacks.

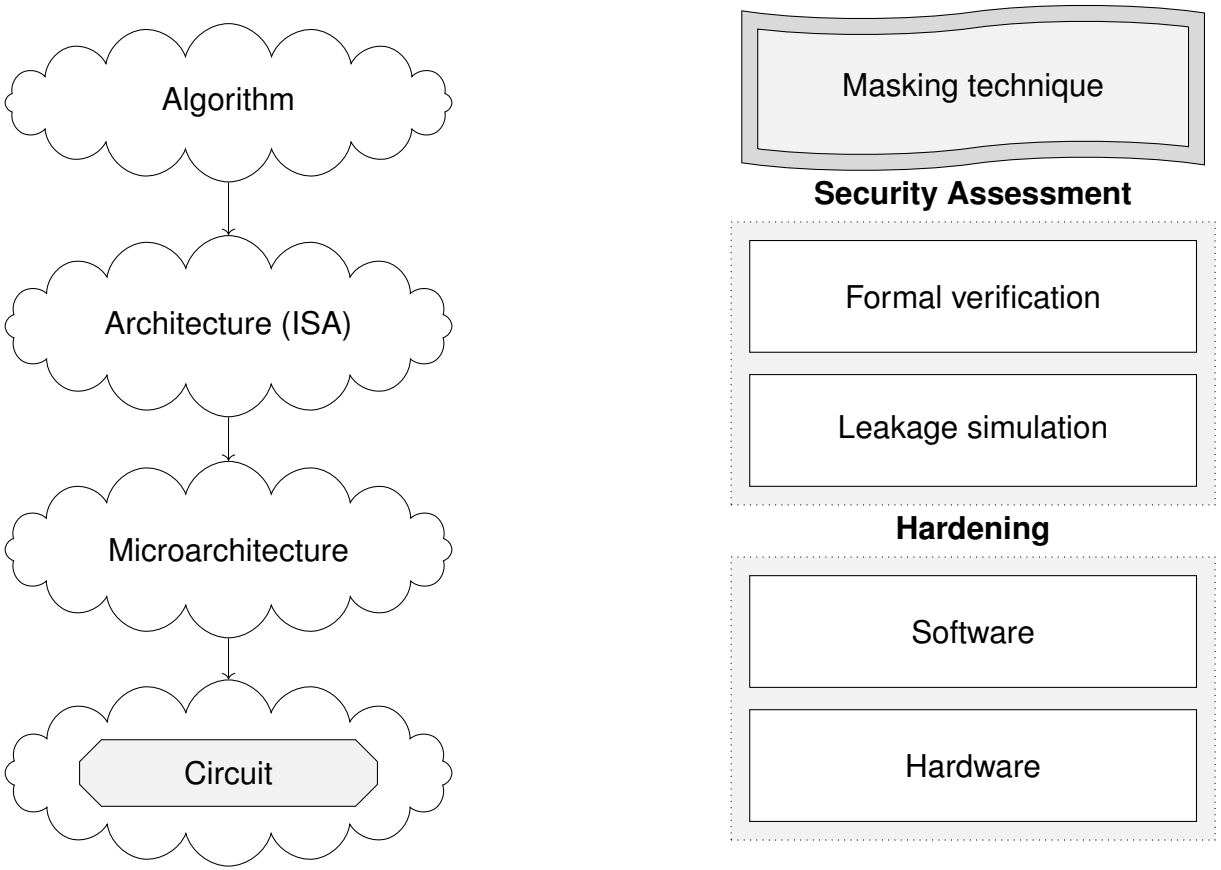
Side-Channel Attacks
Overview

- Side-channel attacks (SCA) are gaining momentum since they are an effective way to break cryptography (along with fault injection).
- This presentation focus on power-based side-channel attacks, i.e. extracting secrets from energy consumption or electromagnetic field.

Side-Channel Leakage Sources
Overview

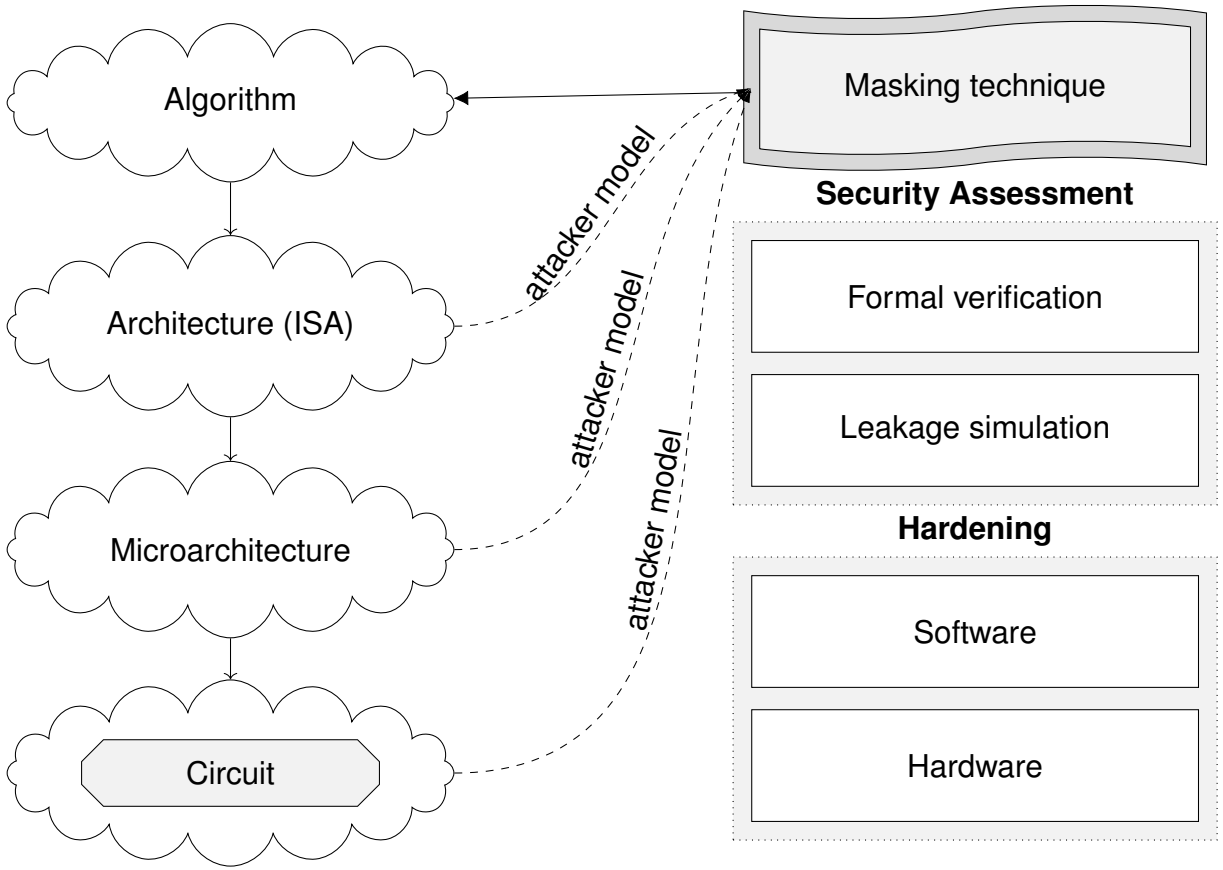


Side-Channel Analysis and Mitigation
State Of The Art



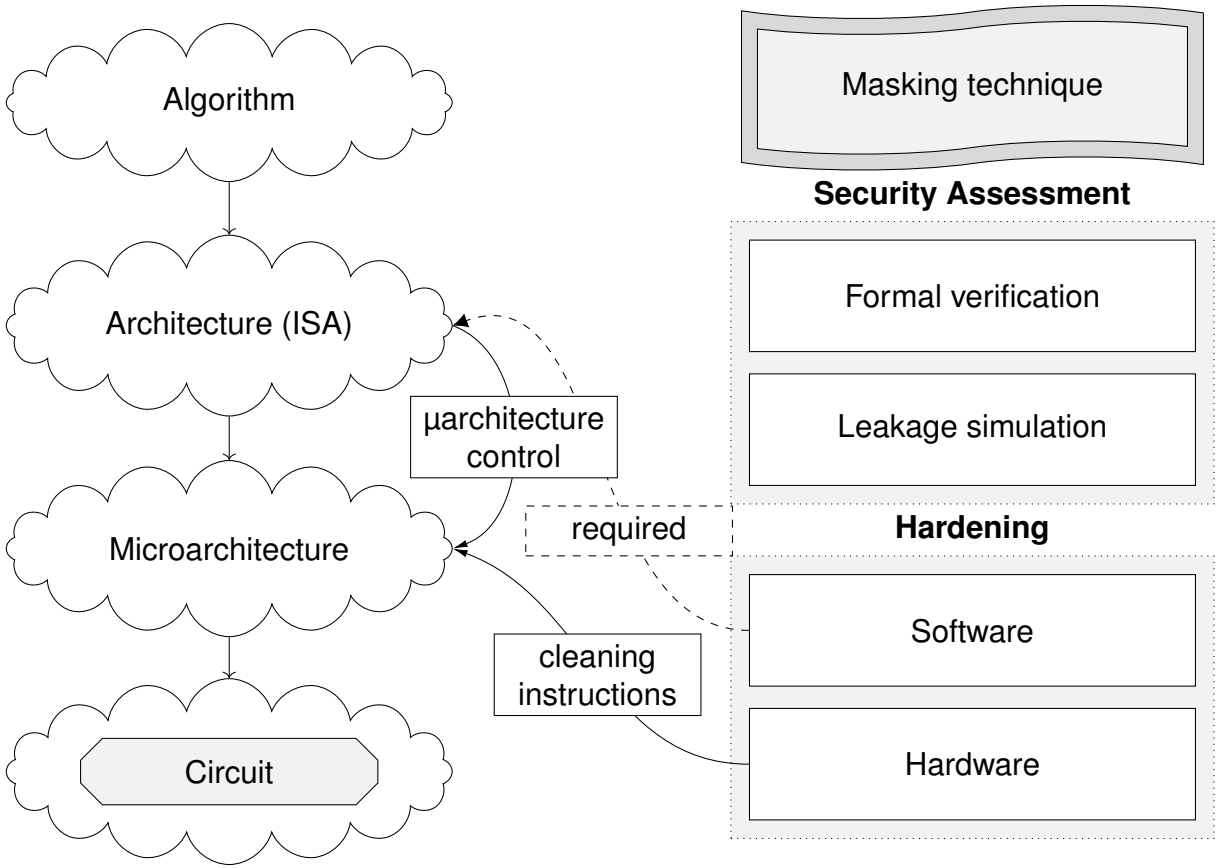
Several methods and tools have been proposed to verify security and harden software or hardware against SCA, with possible interactions, to increase the security level of a cryptographic implementation.

Side-Channel Analysis and Mitigation
State Of The Art



Masking: Method to allow side-channel observation without leaking secret, based on a formal model of the attacker (t-probing for instance).
Challenge: Implement masking with minimal security loss.

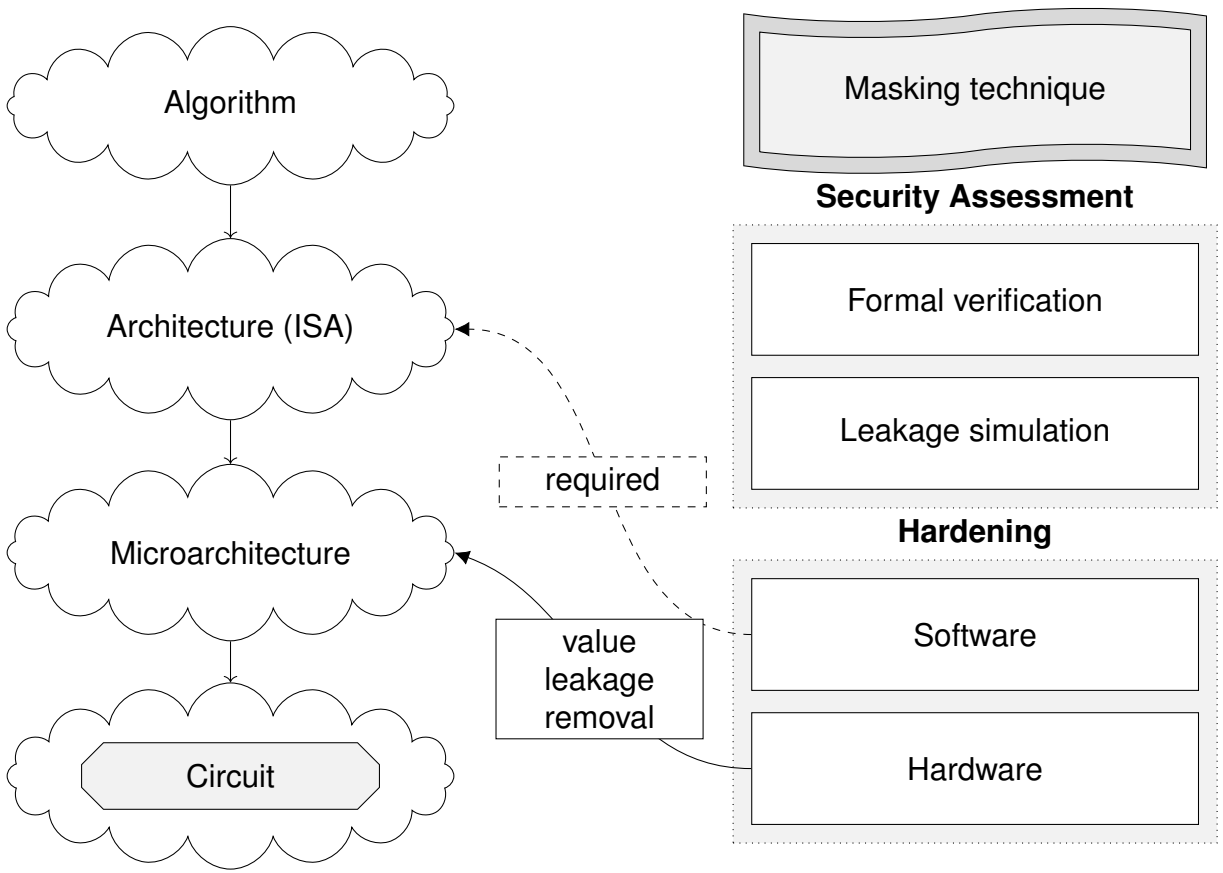
Side-Channel Analysis and Mitigation
State Of The Art



FENL:

1. ISA modification to add fencing instructions to overwrite hardware registers and prevent transitions between the previous/next instructions.
   Limitation: Recompilation required and security assessment on simulated or real hardware required.

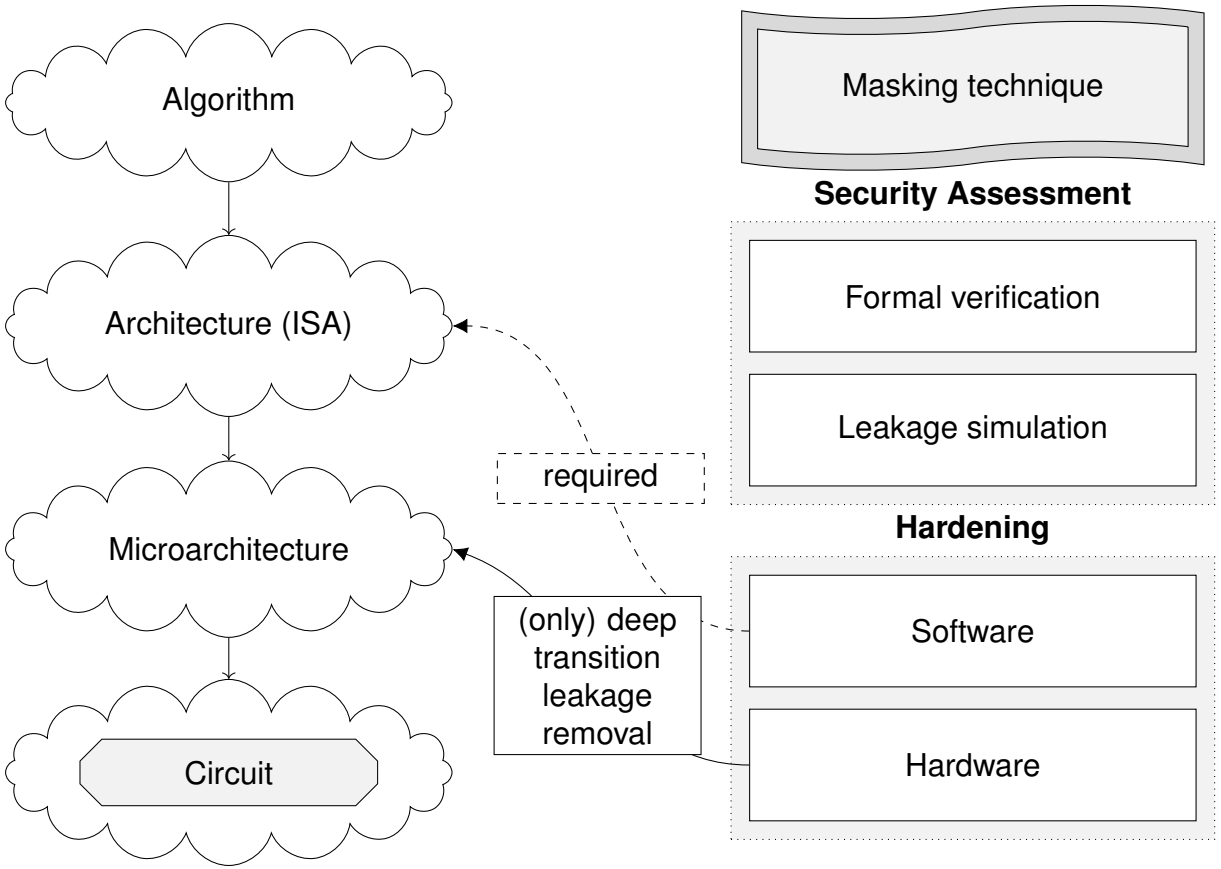
Side-Channel Analysis and Mitigation
State Of The Art



PARAM:
1. Direct data value manipulation obfuscation.
2. Can also increase masking order transparently.

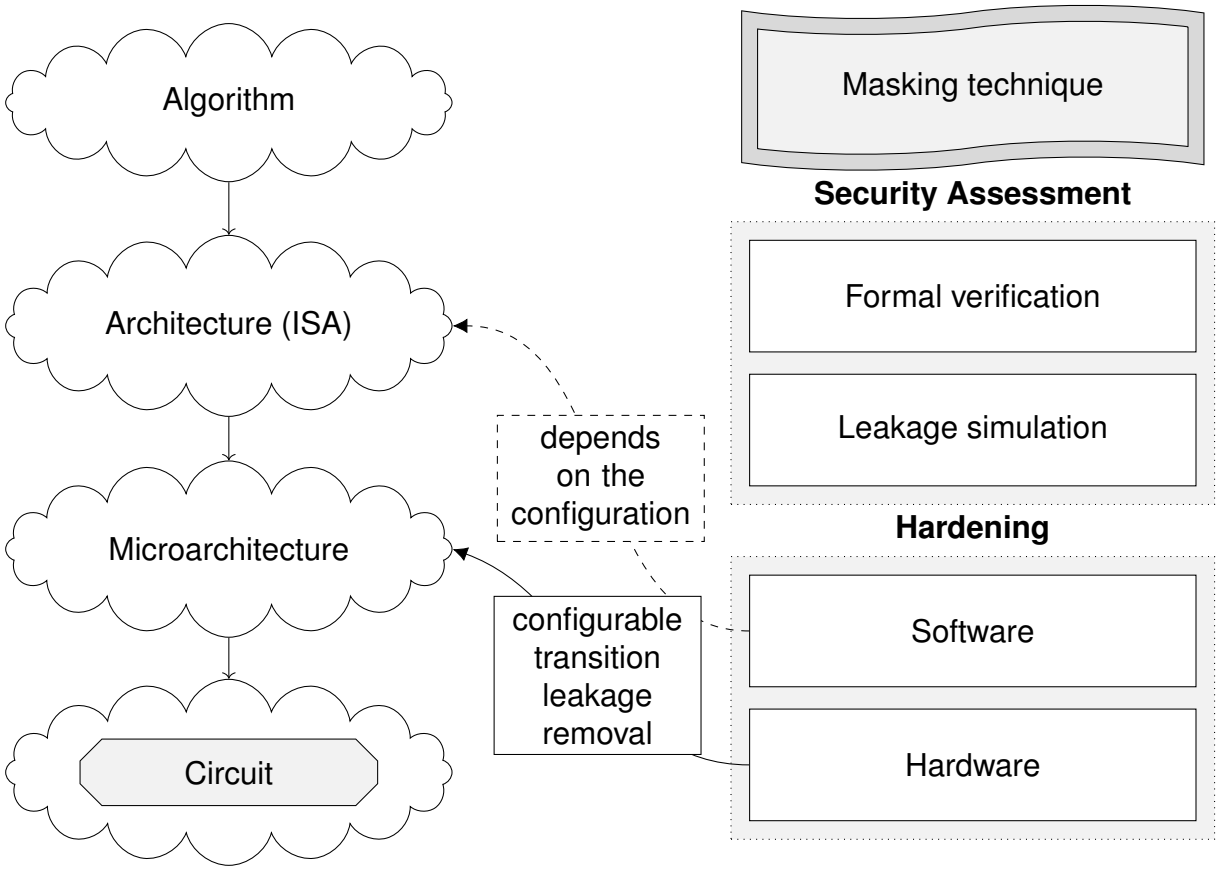
Limitation: Transition leakage not addressed.

Side-Channel Analysis and Mitigation
State Of The Art



COCO:
1. Hardening process split between hardware and software.
2. Deeply hidden transitions in hardware only.
   Limitation: Important part is still delegated to the software: successive instructions or architectural overwritting. Usefull for simple processors without pipeline.

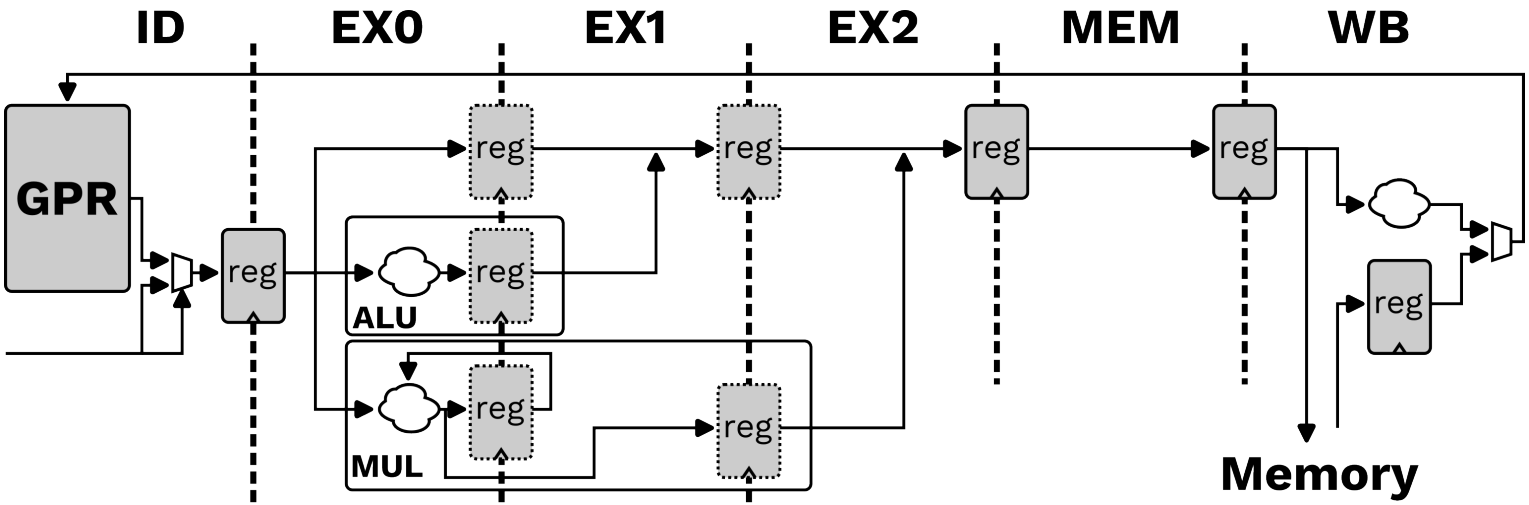
Side-Channel Analysis and Mitigation
State Of The Art



This Work:

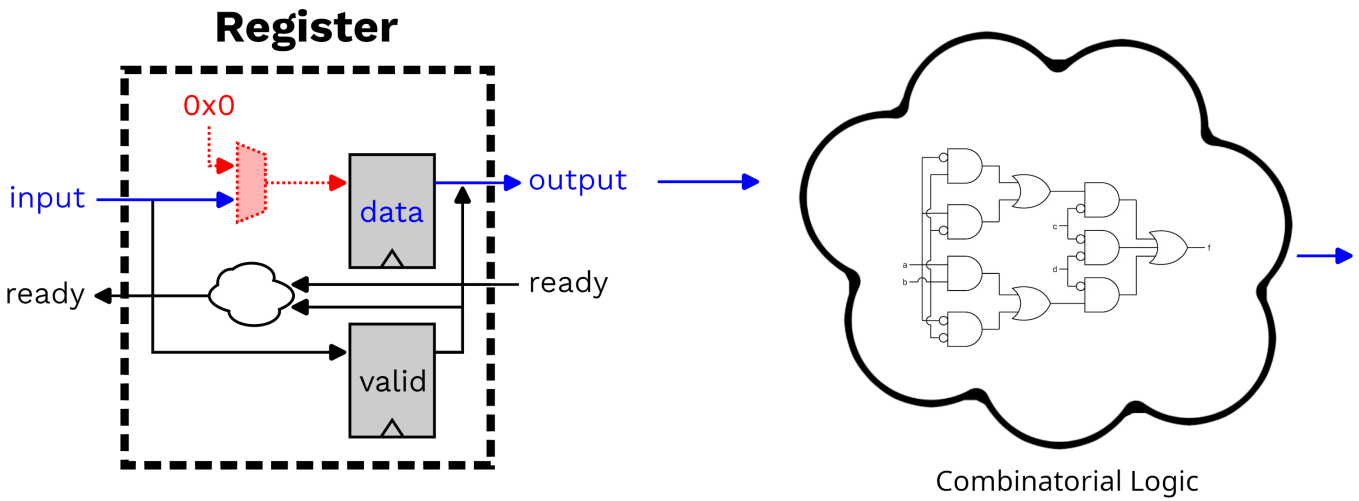
1. Methodology to address transition leakage when designing processors with large pipelines.
2. Configurable balance between software and hardware hardening.
3. Microbenchmarks to assess transition leakage security (in RTL simulation or real target).

Datapath Description and Hardening Strategy



1. **Microarchitectural Hardening (mutually exclusive)**
   - S_1 *Complete Stage Overwriting*
     Delayed Data Multiplexing
   - S_2 *Microarchitecture Register Duplication*
2. **Architectural Hardening**
   - S_3 *Architectural Register Pre-Writing*
     Decoupled Memory Operations

Strategy S1: *Complete Stage Overwriting*
Overview



Objective

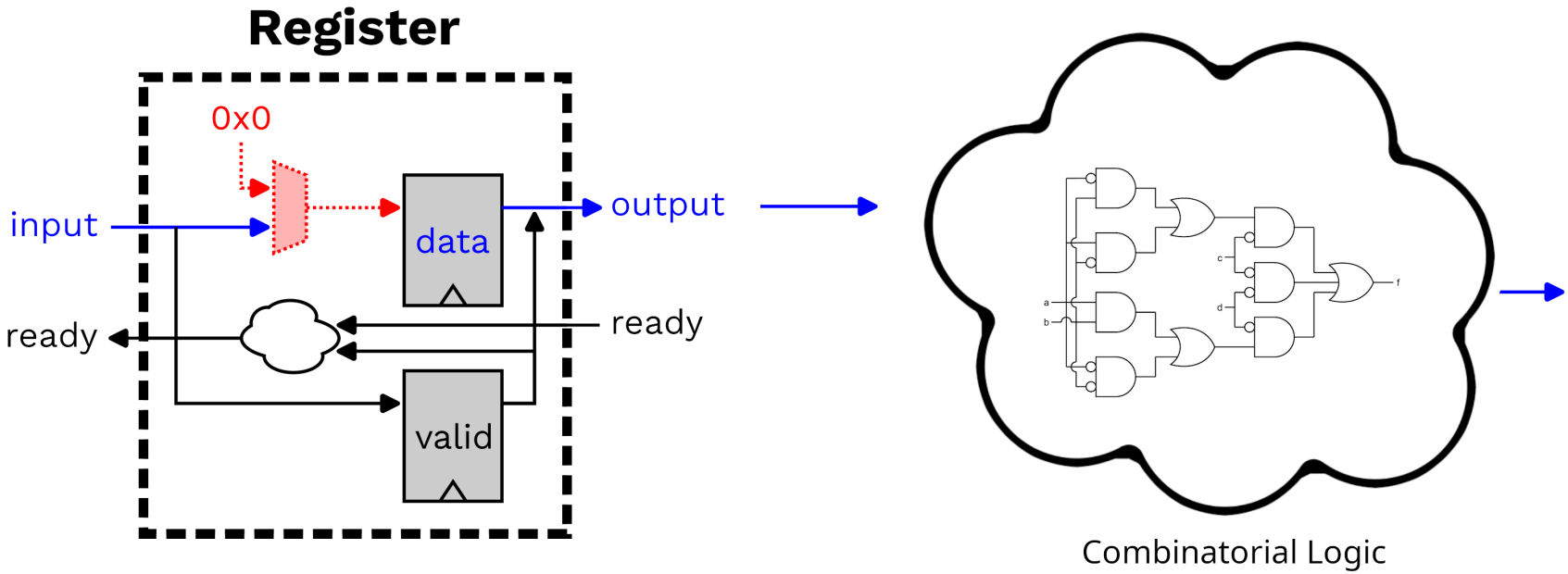
Full pipeline cleaning (including combinatorial logic):

Before: $S_0 \rightarrow S_1 \rightarrow S_2$

After: $S_0 \rightarrow 0\text{x}0 \rightarrow S_1 \rightarrow 0\text{x}0 \rightarrow S_2$

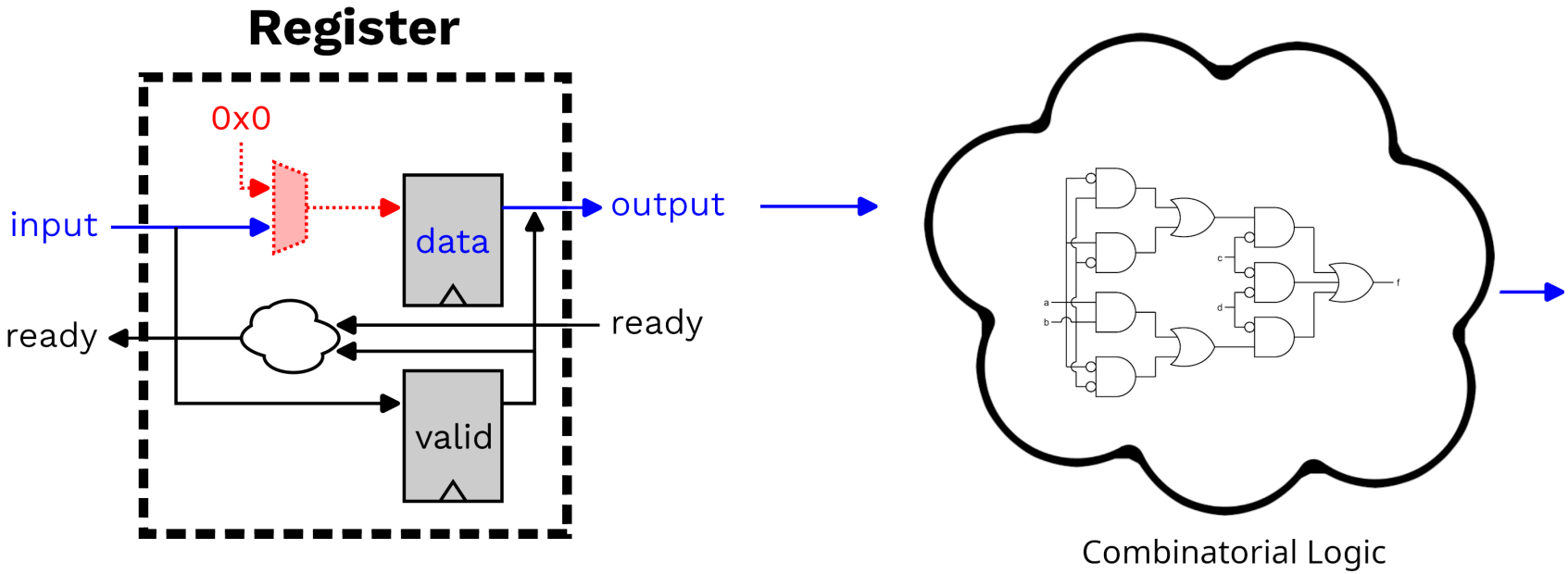
Remark: Only 1 buffer is represented, but all data dependent buffers in a stage are patched with 0x0 insertion synchronized.

Strategy S1: *Complete Stage Overwriting*
Stage Evolution Cycle By Cycle



cycle	
Register's value	

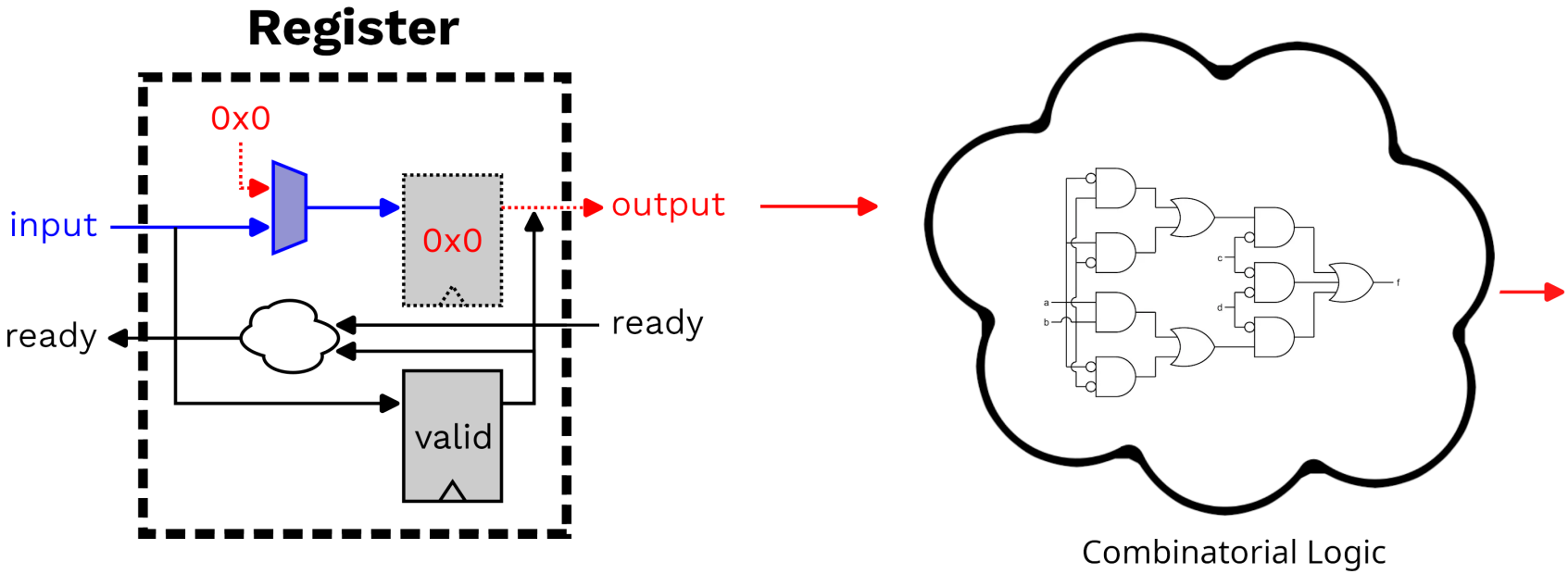
Strategy S1: *Complete Stage Overwriting*
Stage Evolution Cycle By Cycle



cycle	1
Register's value	S_0

Propagation of S_0 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

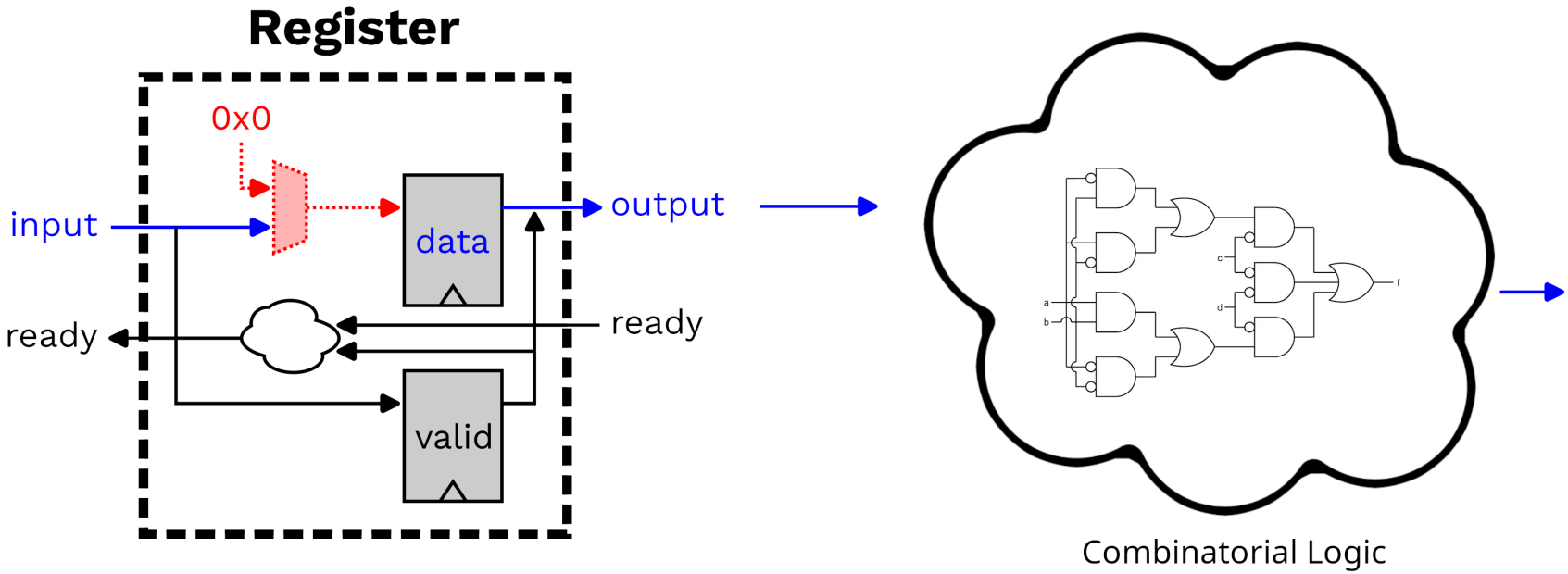
Strategy S1: *Complete Stage Overwriting*
Stage Evolution Cycle By Cycle



cycle	1	2
Register's value	S_0	0x0

Stage buffers and all combinatorial logic cleaning (i.e. set to 0).

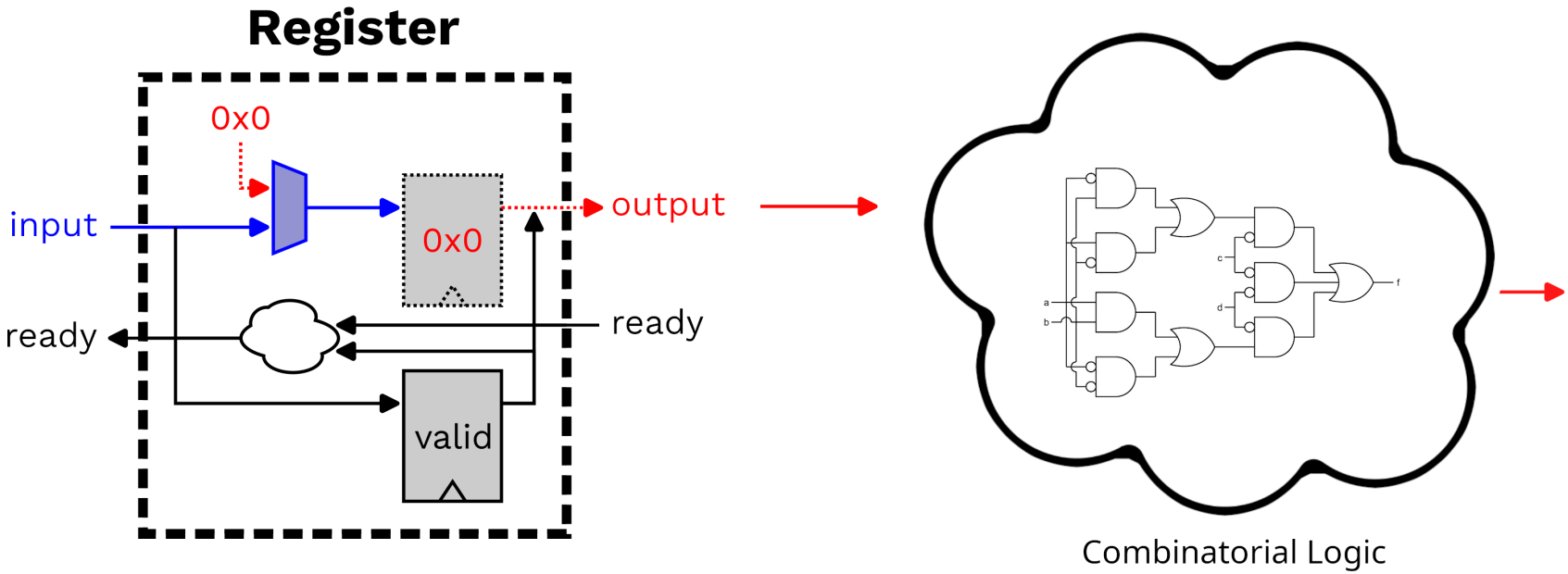
Strategy S1: *Complete Stage Overwriting*
Stage Evolution Cycle By Cycle



cycle	1	2	3
Register's value	S_0	0x0	S_1

Propagation of S_1 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

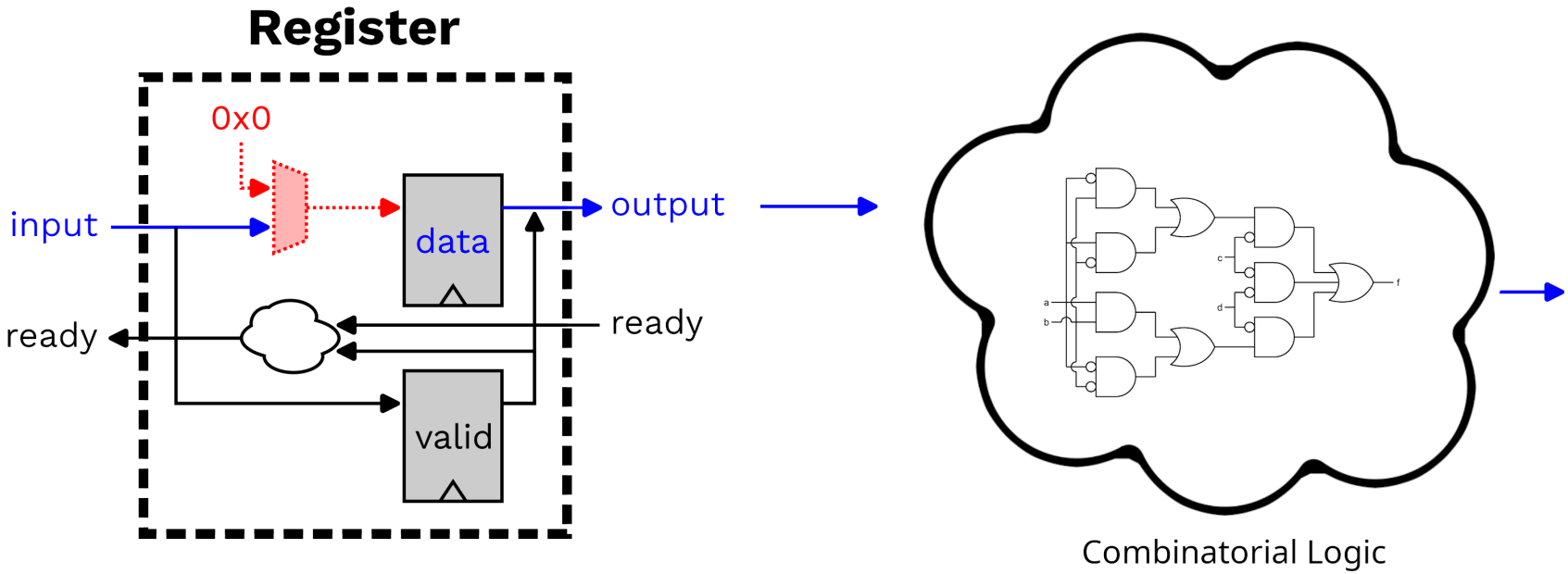
Strategy S1: *Complete Stage Overwriting*
Stage Evolution Cycle By Cycle



cycle	1	2	3	4
Register's value	S_0	0x0	S_1	0x0

Stage buffers and all combinatorial logic cleaning (i.e. set to 0).

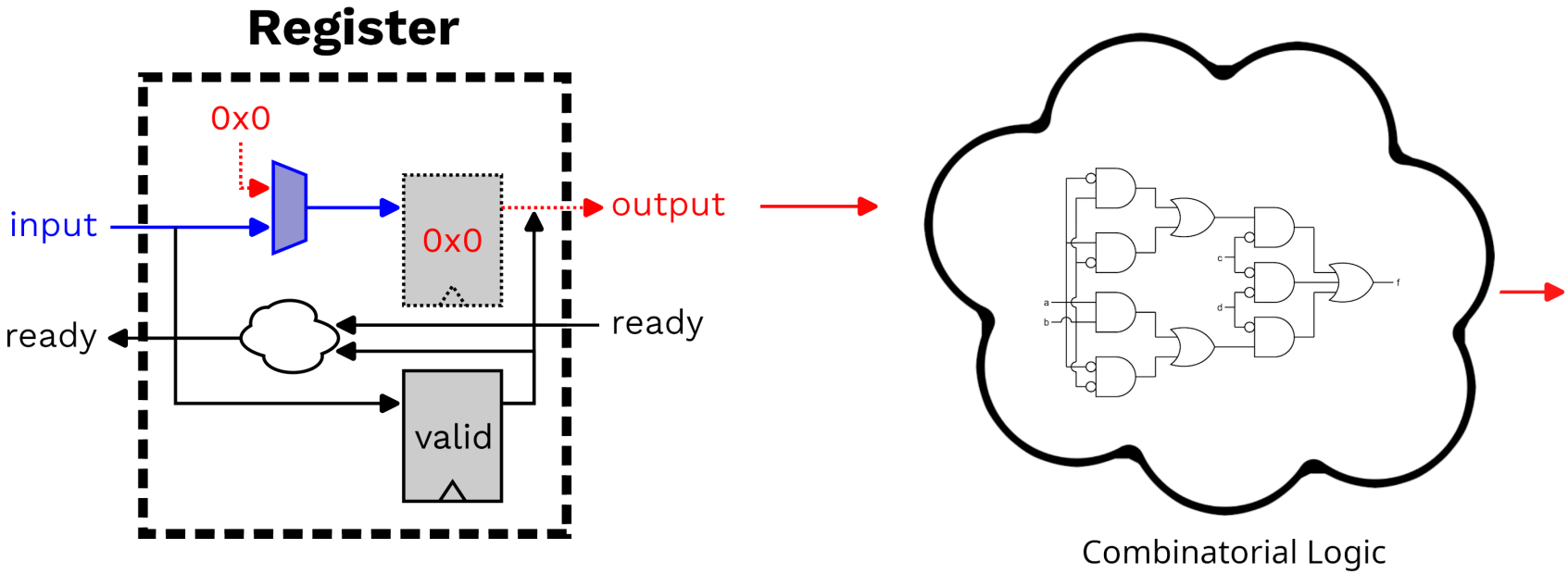
Strategy S1: *Complete Stage Overwriting*
Stage Evolution Cycle By Cycle



cycle	1	2	3	4	5
Register's value	S_0	0x0	S_1	0x0	S_2

Propagation of S_2 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

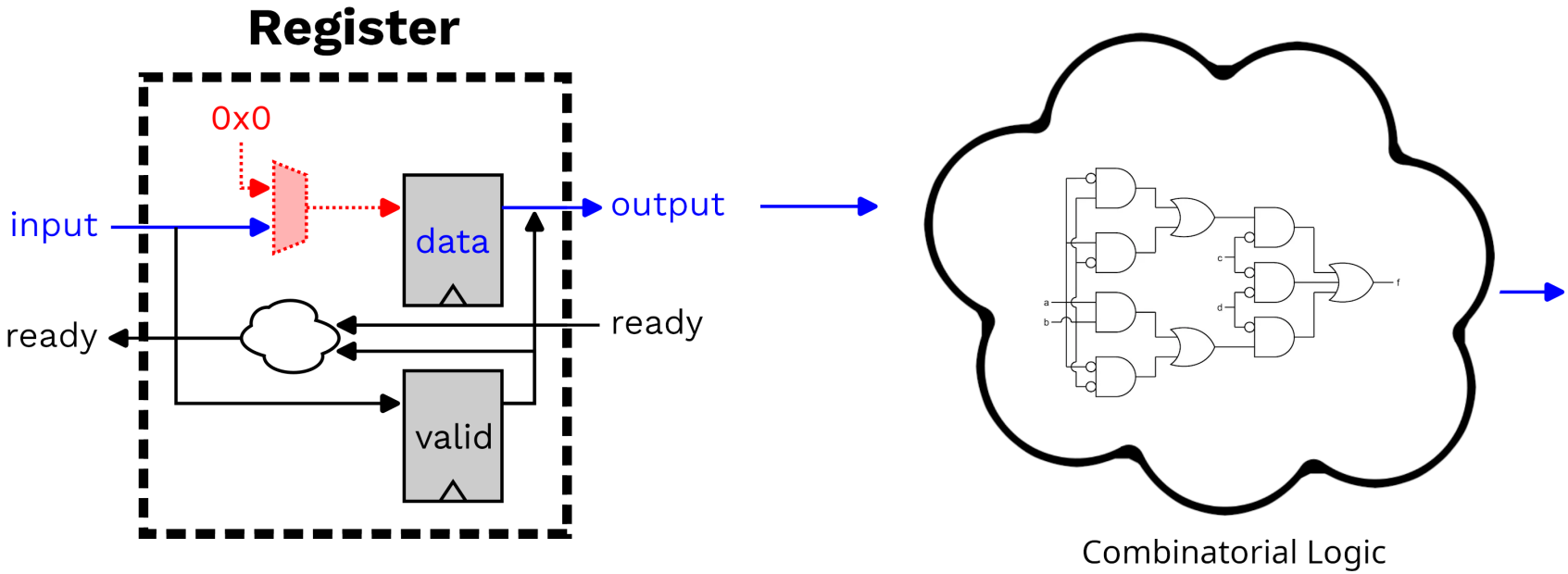
Strategy S1: *Complete Stage Overwriting*
Stage Evolution Cycle By Cycle



cycle	1	2	3	4	5	6
Register's value	S_0	0x0	S_1	0x0	S_2	0x0

Stage buffers and all combinatorial logic cleaning (i.e. set to 0).

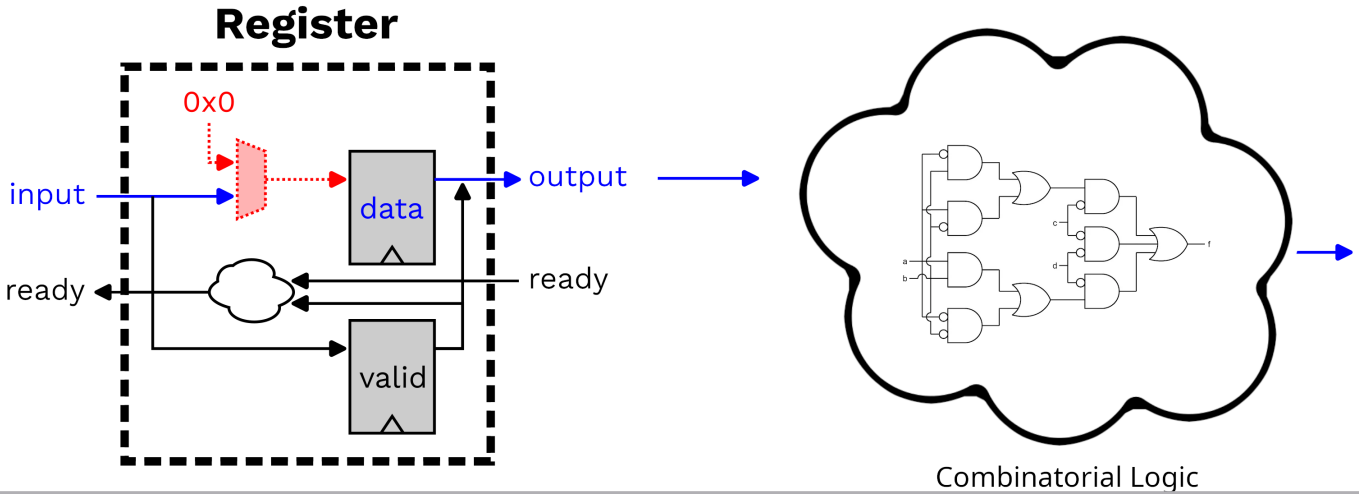
Strategy S1: *Complete Stage Overwriting*
Stage Evolution Cycle By Cycle



cycle	1	2	3	4	5	6	7
Register's value	S_0	0x0	S_1	0x0	S_2	0x0	S_3

Propagation of S_3 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

Strategy S1: *Complete Stage Overwriting*
Security exposed to ISA

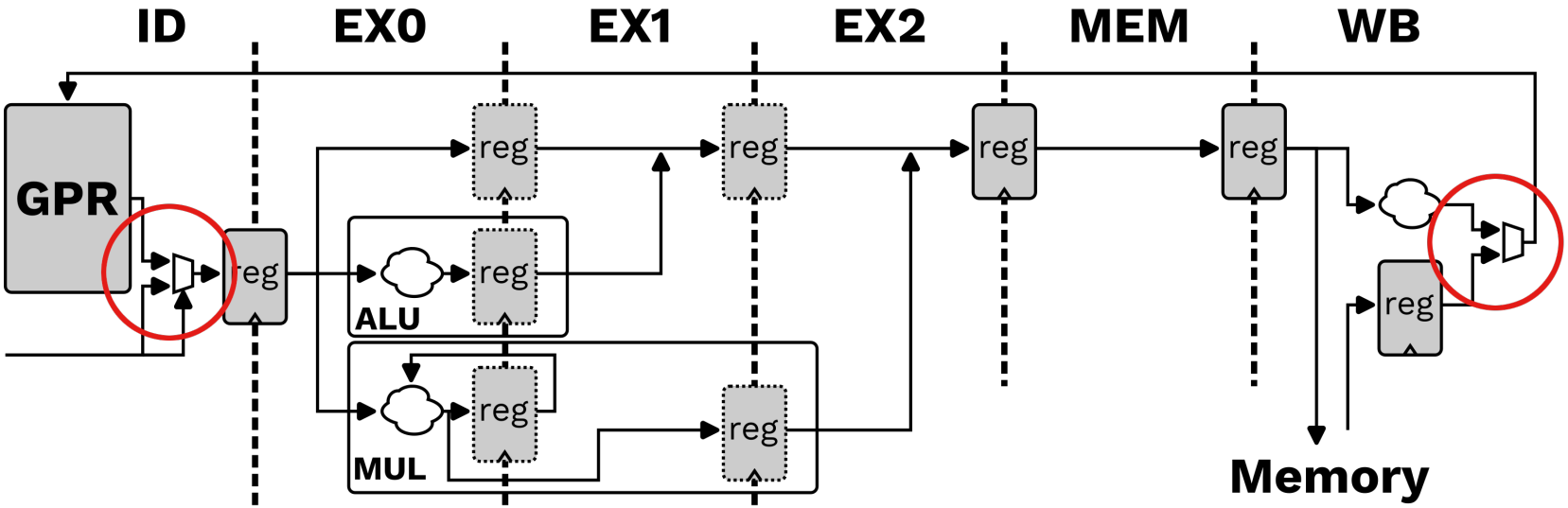


Security Property

For a given pipeline stage and all possible instruction's sequence manipulating data stored in the Register File, no transition leakage can occur in the **stage** (buffer + logic behind) *by design*.

Remark: Since ID/EX buffer must be patched, there is no reason to change strategy for following stages because the 2 cycles penalty will be propagated to the other stages.

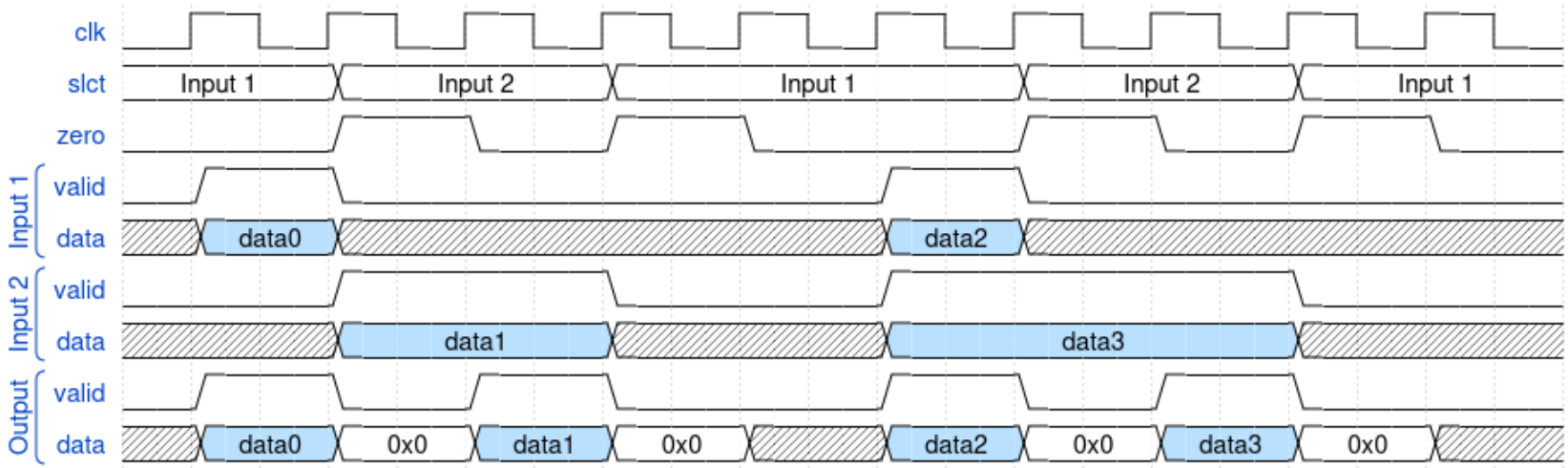
Strategy S1: *Delayed Data Multiplexing*
Standard Pipeline Issue



Data can come from different sources
$\rightarrow$ need multiplexing with 0x0 insertion guarantees.

Targeted mechanisms: register forwarding, load resize buffer, skid buffer (memory controller).

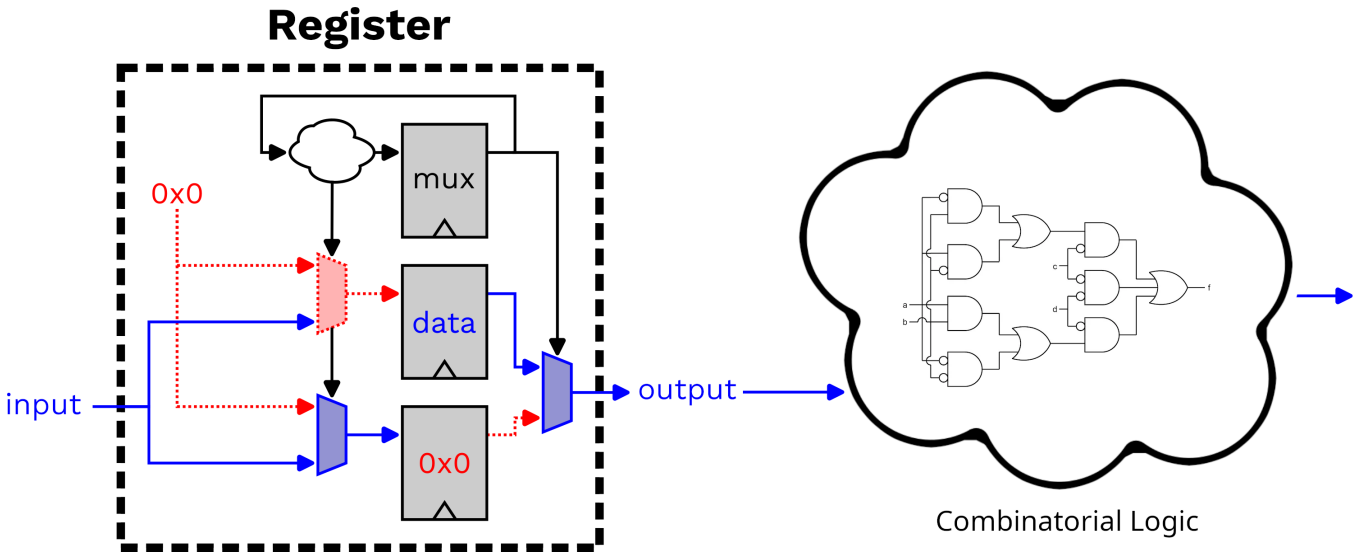
Strategy S1: *Delayed Data Multiplexing*
Standard Pipeline Issue



- Definition of a *Slot* Finite State Machine to alternate between inputs.
- Ready not displayed but required to delay inputs.

Hardware implementation: input selection using a `slct` $log_2(N_{inputs})$-bit register, force zero using a `zero` 1-bit register after each processed operation.

Strategy S2: *µarch. Register Duplication*
Overview

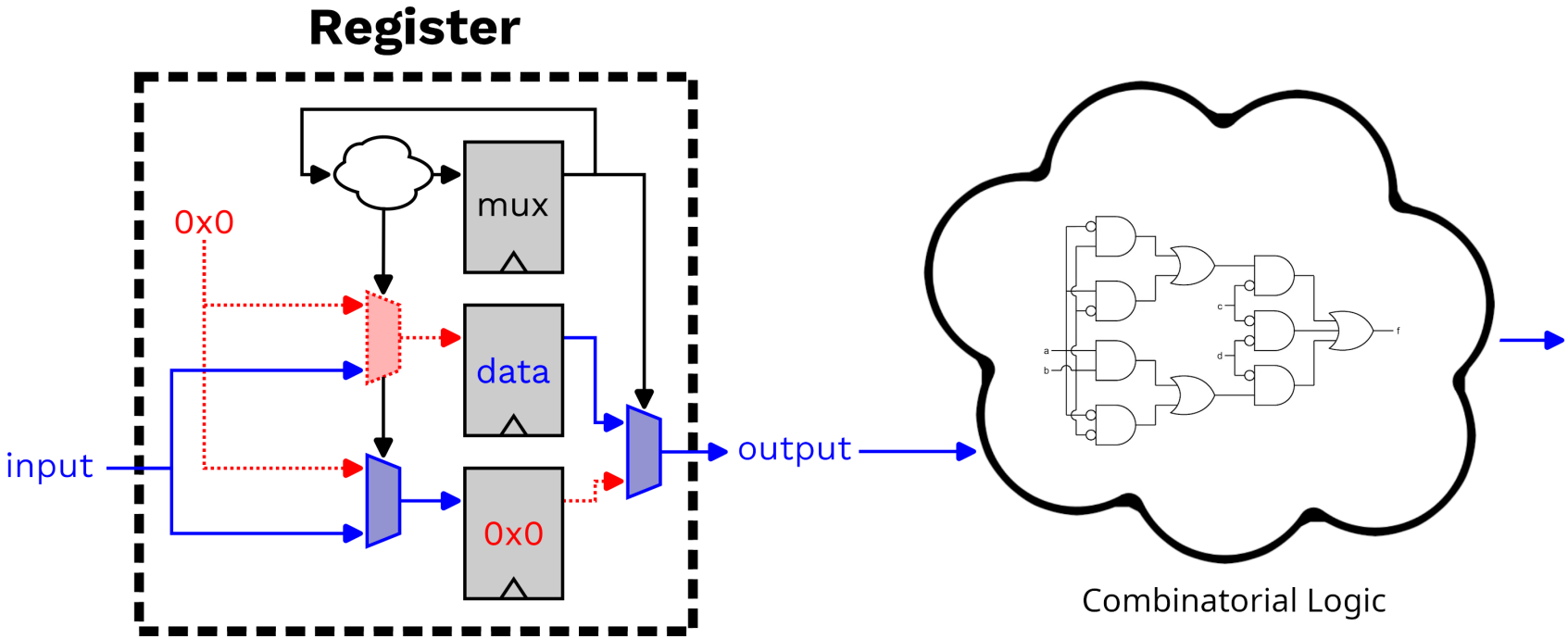


Objective

Stage buffer cleaning:

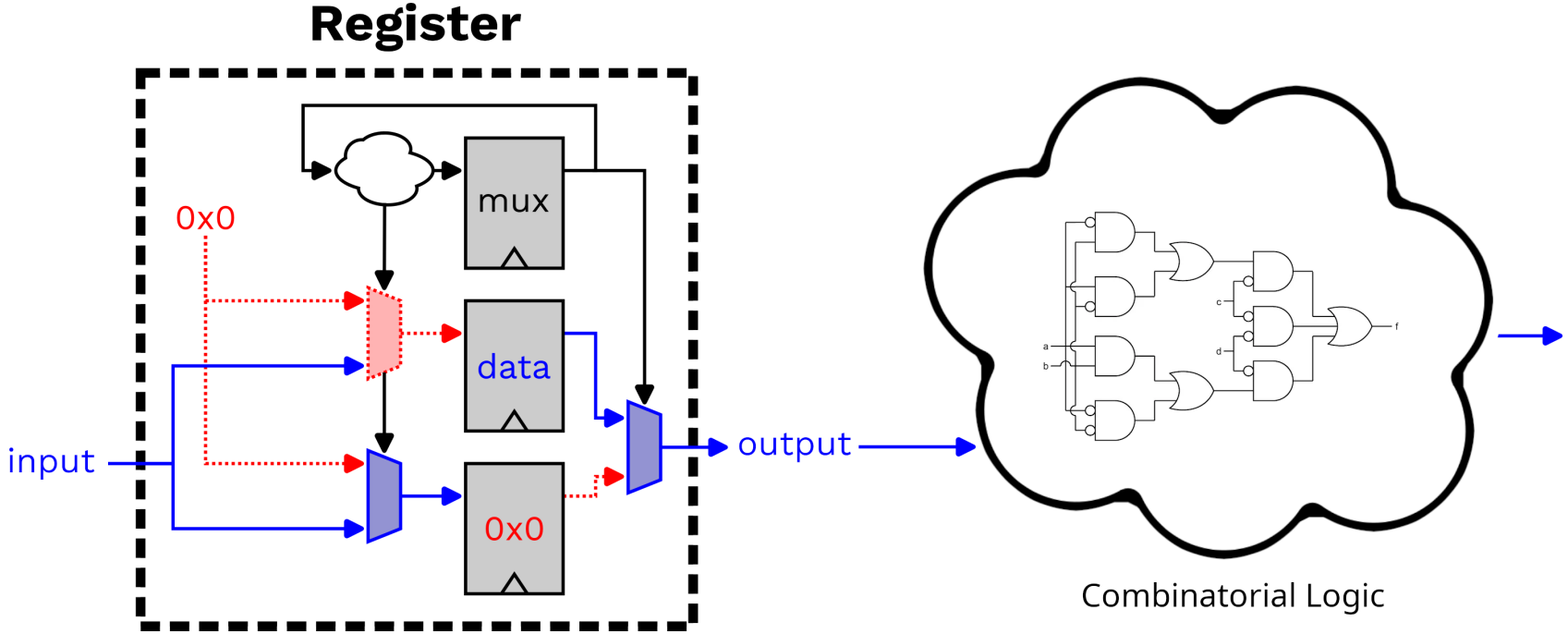
Before:		S_0	$\rightarrow$	S_1	$\rightarrow$	S_2	$\rightarrow$	S_3
After:	R_{above}	S_0	$\rightarrow$	0x0	$\rightarrow$	S_2	$\rightarrow$	0x0
	R_{below}	0x0	$\rightarrow$	S_1	$\rightarrow$	0x0	$\rightarrow$	S_3
	output	S_0	$\rightarrow$	S_1	$\rightarrow$	S_2	$\rightarrow$	S_3

Strategy S2: *µarch. Register Duplication*
Stage Evolution Cycle By Cycle



cycle	
R_{above}'s value	
R_{below}'s value	

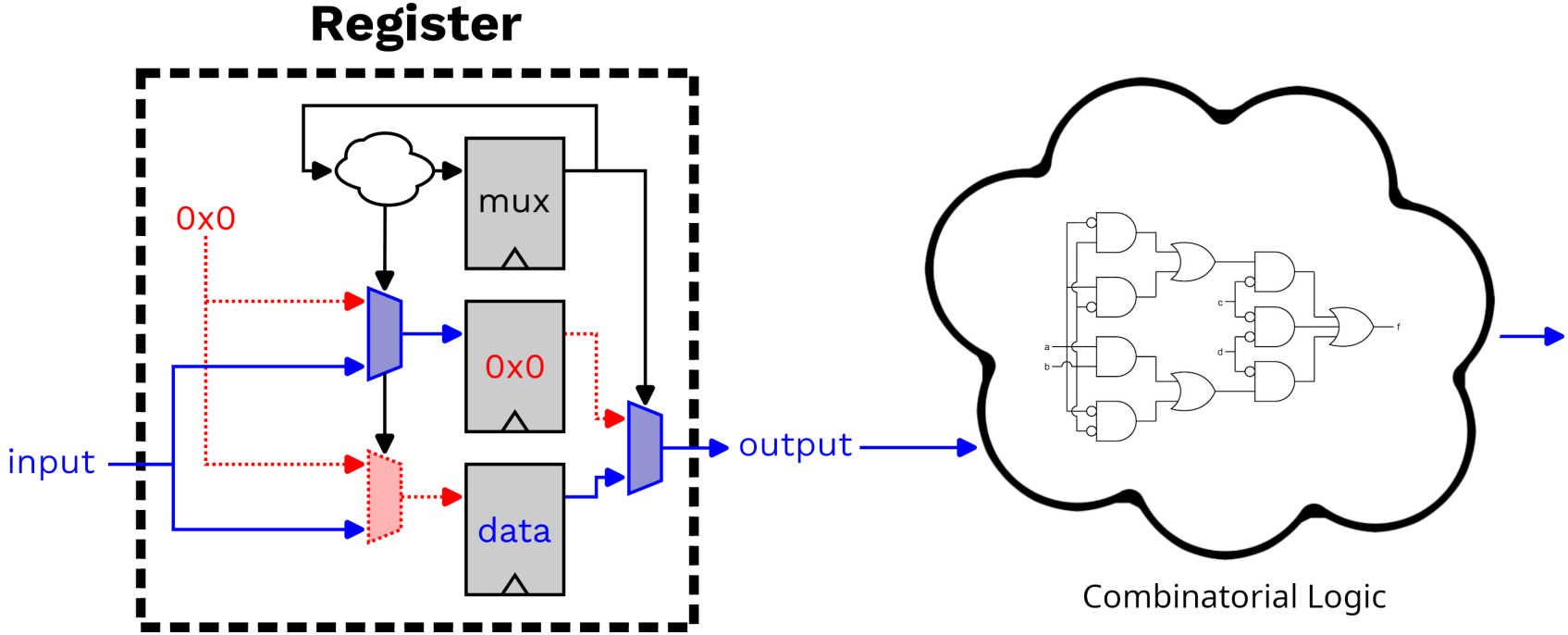
Strategy S2: *µarch. Register Duplication*
Stage Evolution Cycle By Cycle



cycle	1
R_{above}'s value	S_0
R_{below}'s value	0x0

Propagation of S_0 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

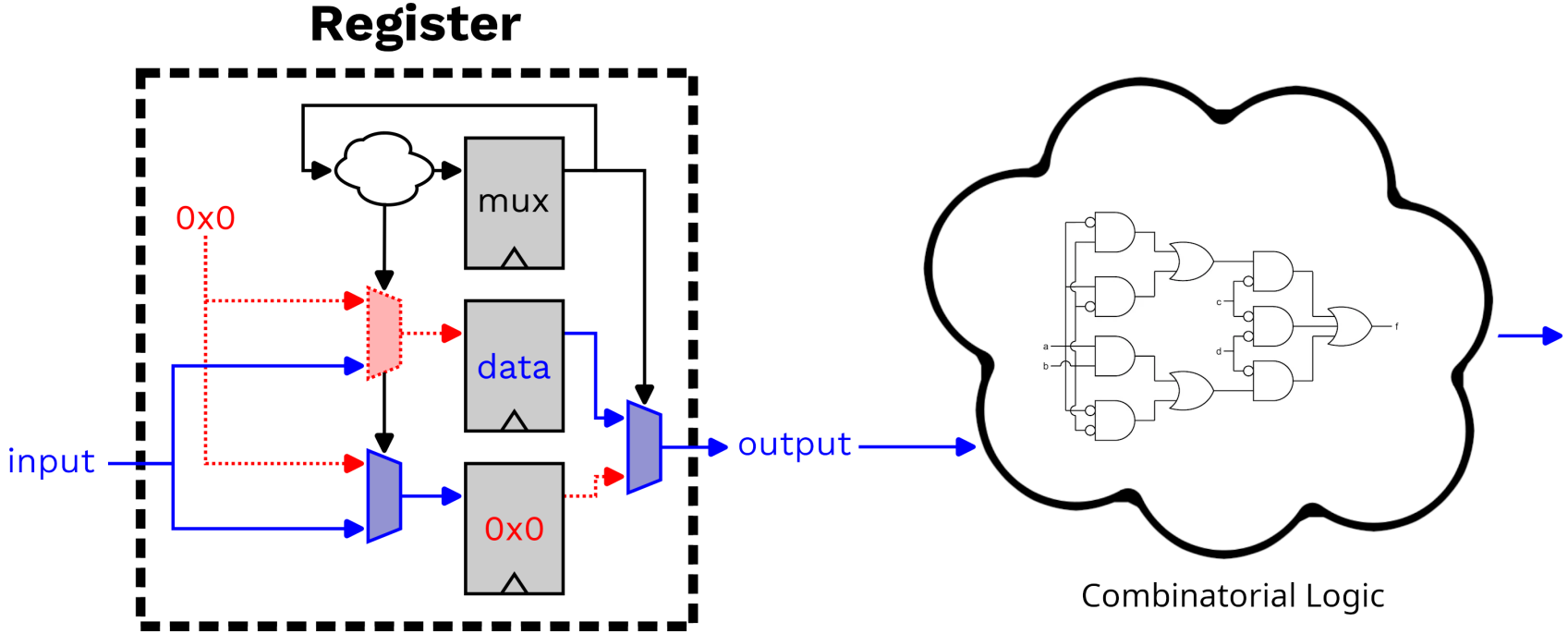
Strategy S2: *µarch. Register Duplication*
Stage Evolution Cycle By Cycle



cycle	1	2
R_{above}'s value	S_0	0x0
R_{below}'s value	0x0	S_1

Propagation of S_1 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

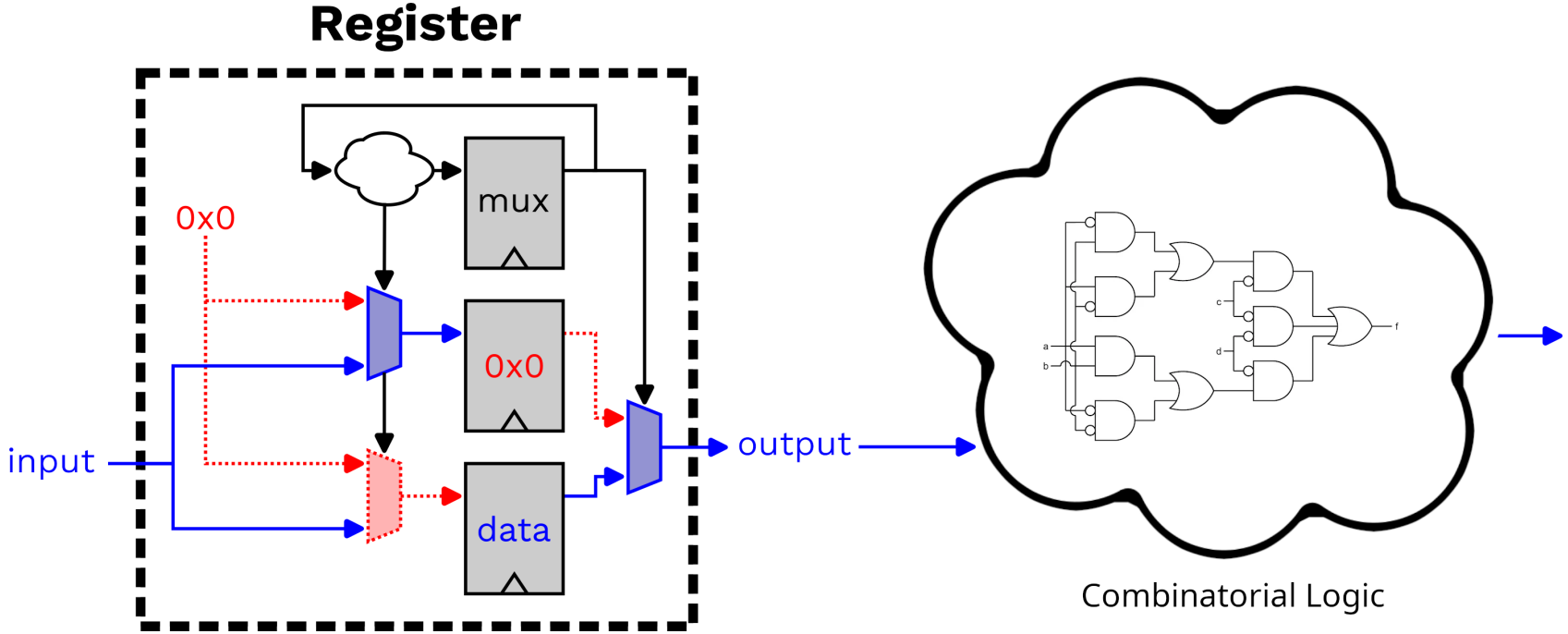
Strategy S2: *µarch. Register Duplication*
Stage Evolution Cycle By Cycle



cycle	1	2	3
R_{above}'s value	S_0	0x0	S_2
R_{below}'s value	0x0	S_1	0x0

Propagation of S_2 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

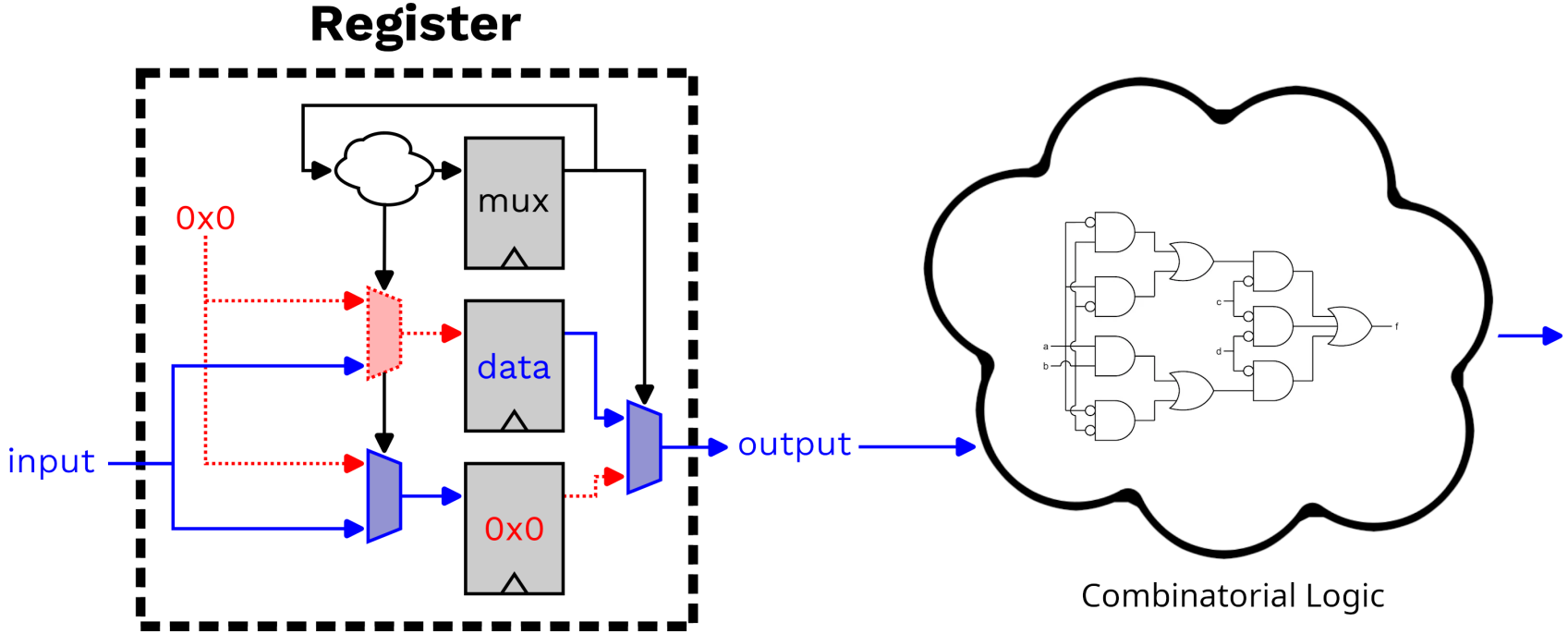
Strategy S2: *µarch. Register Duplication*
Stage Evolution Cycle By Cycle



cycle	1	2	3	4
R_{above}'s value	S_0	0x0	S_2	0x0
R_{below}'s value	0x0	S_1	0x0	S_3

Propagation of S_3 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

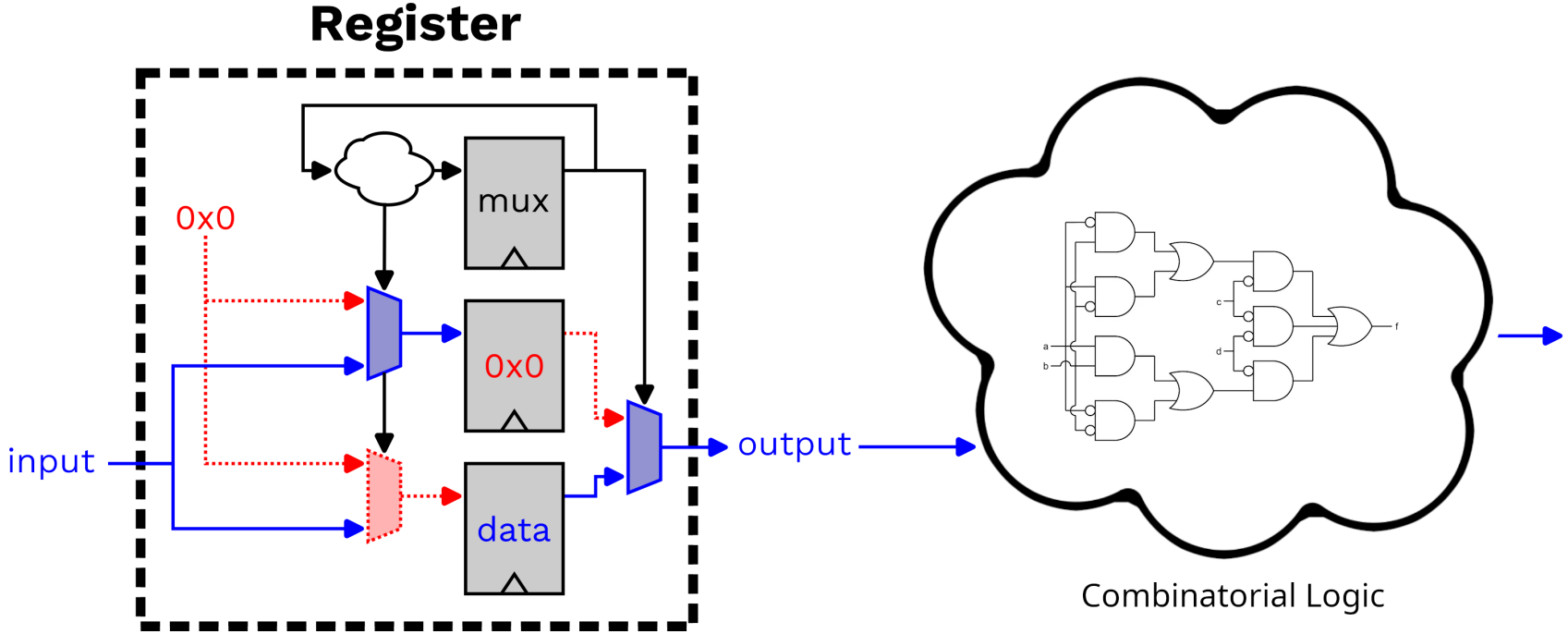
Strategy S2: *µarch. Register Duplication*
Stage Evolution Cycle By Cycle



cycle	1	2	3	4	5
R_{above}'s value	S_0	0x0	S_2	0x0	S_4
R_{below}'s value	0x0	S_1	0x0	S_3	0x0

Propagation of S_4 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

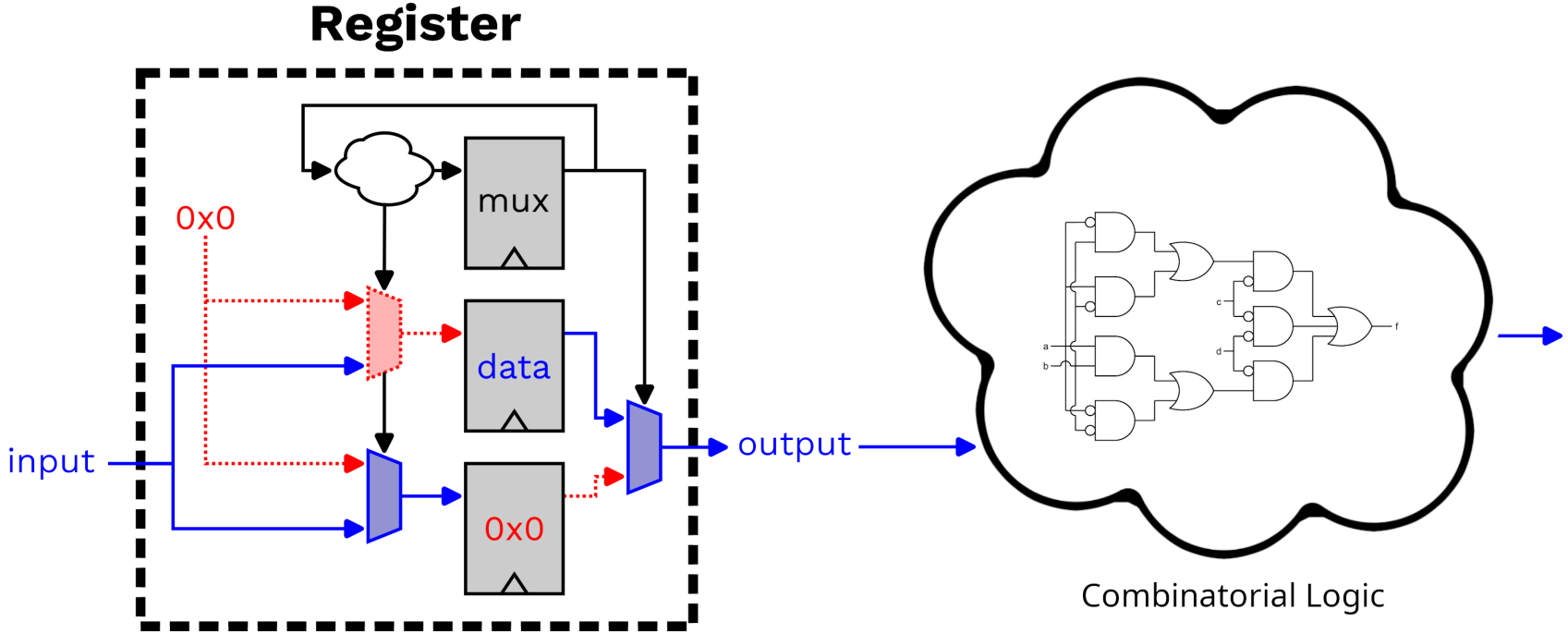
Strategy S2: *µarch. Register Duplication*
Stage Evolution Cycle By Cycle



cycle	1	2	3	4	5	6
R_{above}'s value	S_0	0x0	S_2	0x0	S_4	0x0
R_{below}'s value	0x0	S_1	0x0	S_3	0x0	S_5

Propagation of S_5 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

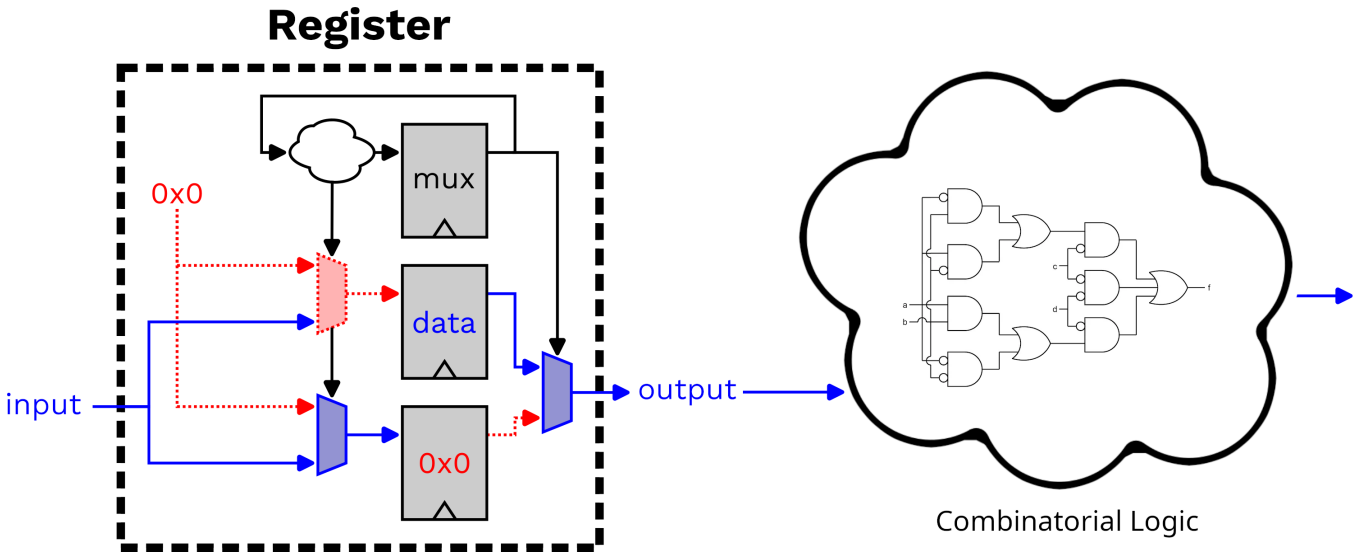
Strategy S2: *µarch. Register Duplication*
Stage Evolution Cycle By Cycle



cycle	1	2	3	4	5	6	7
R_{above}'s value	S_0	0x0	S_2	0x0	S_4	0x0	S_6
R_{below}'s value	0x0	S_1	0x0	S_3	0x0	S_5	0x0

Propagation of S_6 bits *(+ all outputs from other stage buffers)* through the combinatorial logic.

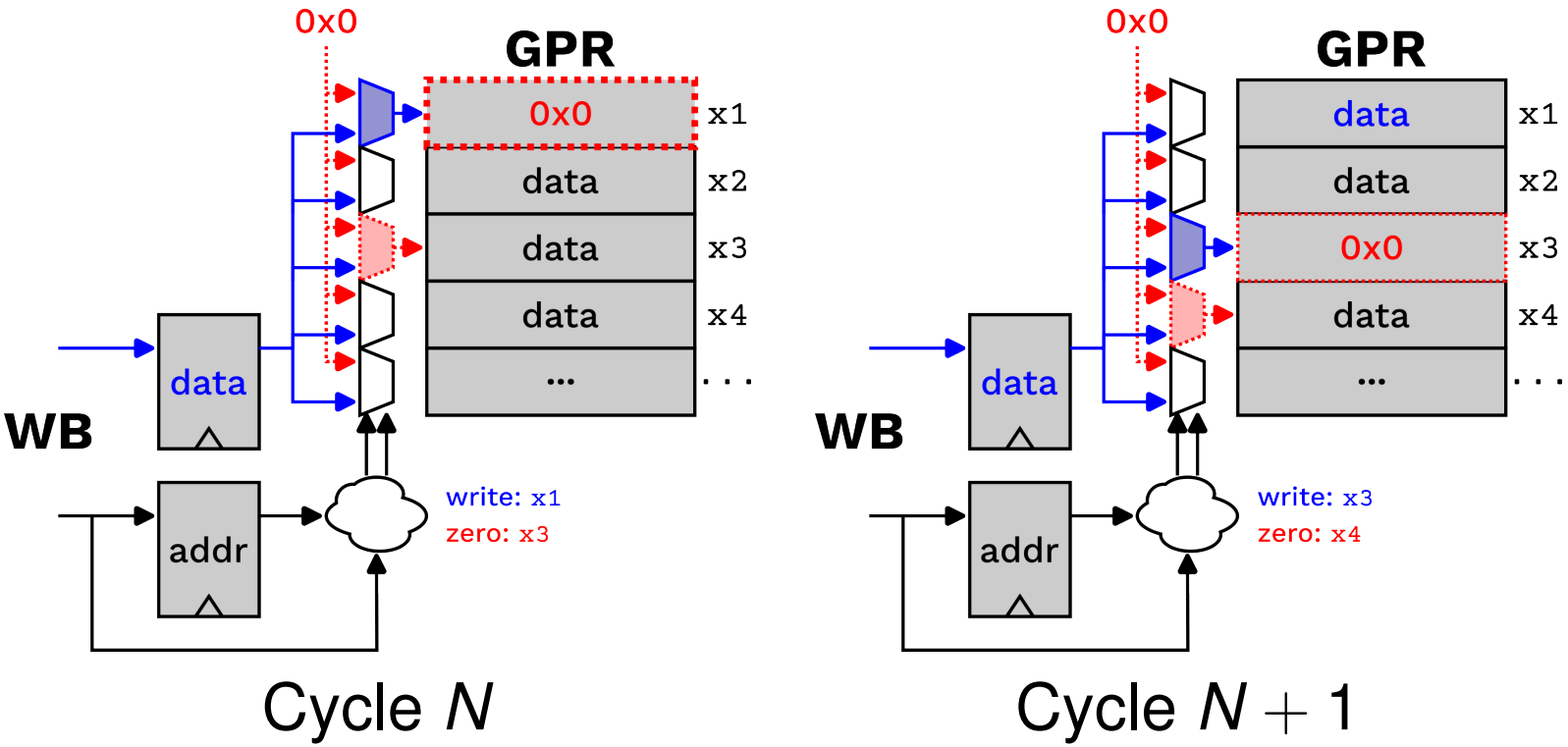
Strategy S2: *µarch. Register Duplication*
Security exposed to ISA



Security Property

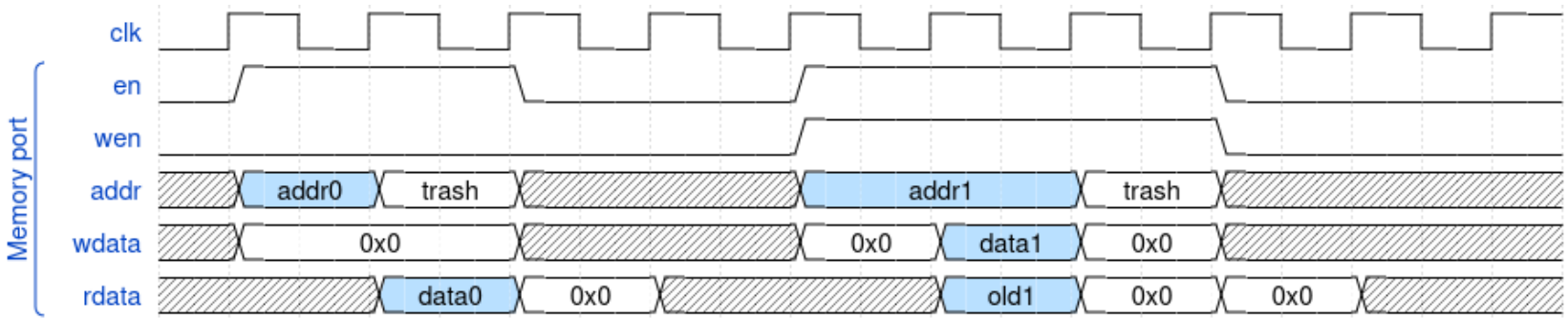
For a given pipeline stage and all possible instruction's sequence manipulating data stored in the Register File, no transition leakage can occur in the **buffer** *by design*.

Strategy S3.a: *Arch. Register Pre-Writing*
General Purpose Register File Hardening



Same strategy than S_1 *Complete Stage Overwriting*, but the Write Back allow anticipate which register must be cleaned.

Strategy S3.b: *Decoupled Memory Operations*
Memory Controller Hardening



Trash: a reserved memory location to perform hardware hardening operations.

Read: after each valid read, a second read is performed from the trash to clean the `rdata` signal.

Write: before each write, a previous operation overwrite **wdata** and the memory value / after, a write is performed to the trash to clean the `wdata` signal.

Implementation Results
Configurations Evaluated

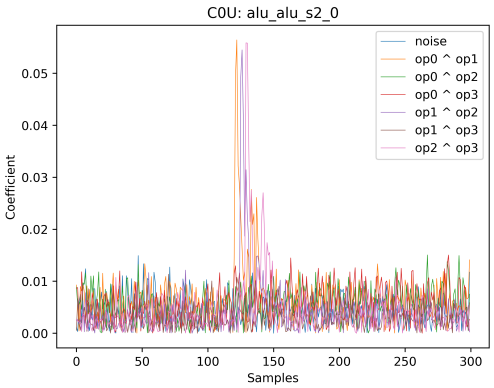
1. Target Architecture: RISCV32IM_Zicsr
2. Specifications in CHISEL
3. Validation with microbenchmarks:
   - In simulation with Verilator
   - In real target with chipwhisperer

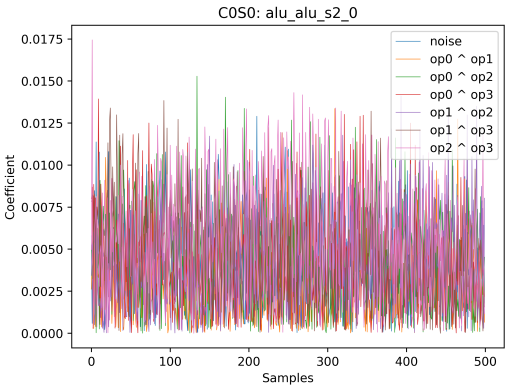
Conf.	Stages	S1.a	S1.b	S2	S3.a	S3.b
C5U	5					
C5S1	5	✓	✓		✓	✓
C5S2	5			✓	✓	⋆
C7U	7					
C7S1	7	✓	✓		✓	✓
C7S2	7			✓	✓	⋆

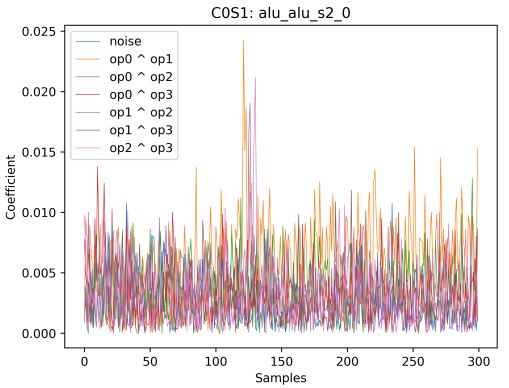
⋆ : Partial support, trash operations not needed to clean signals.

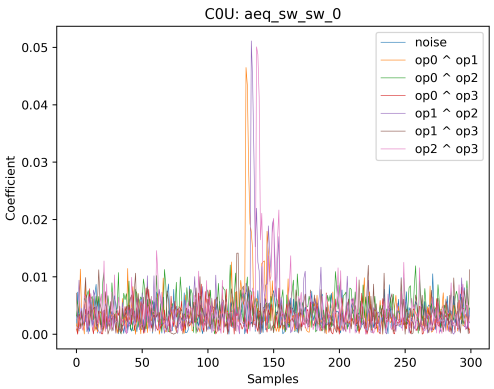
Implementation Results
Security with Correlation Power Analysis

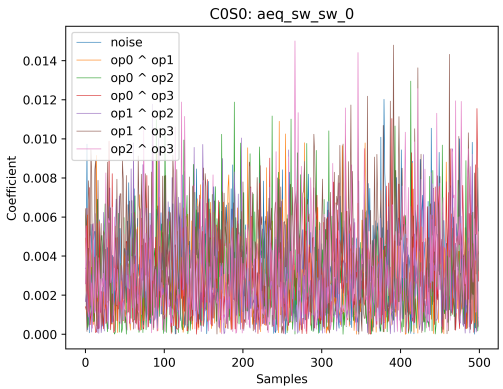
C5U | C5S1 | C5S2

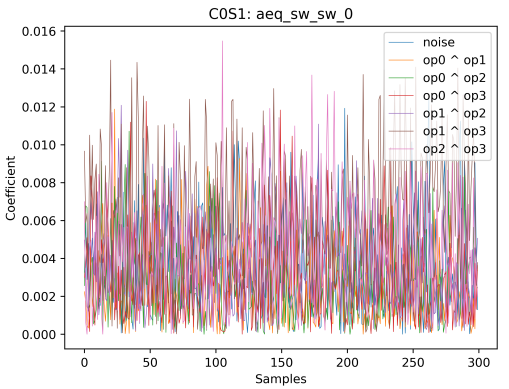












Above Arithmetical and logical operations sequence.

Below Store operations sequence.

Configuration: *50,000 traces*

Implementation Results
Performance

Conf.	Embench	LUT	FF
C5U	1	1	1
C5S1	1.8635	1.0533	1.0088
C5S2	1.0266	1.3746	1.2418
C7U	1	1	1
C7S1	1.4821	1.1116	1.0215
C7S2	1.0002	1.2751	1.3343

Ratio with unprotected cores (CU5 and CU7).

Conclusion

1. Addressing transition leakage from its root cause is possible (and implementable).
2. Design choices between security, area and cycles overhead are required (with possible software countermeasures).
3. Glitches may still occur here since it's not a µarchitectural issue (circuit layer), and are not clearly identified after removing other leakage.

Future Work

Other layers consideration:

Application	Impact of the different hardware strategies?
ISA	Possibility to enable/disable a strategy only when needed
Physical	Impact of synthesis / place-and-route steps?

Thanks for your attention. Don’t hesitate to ask your questions.