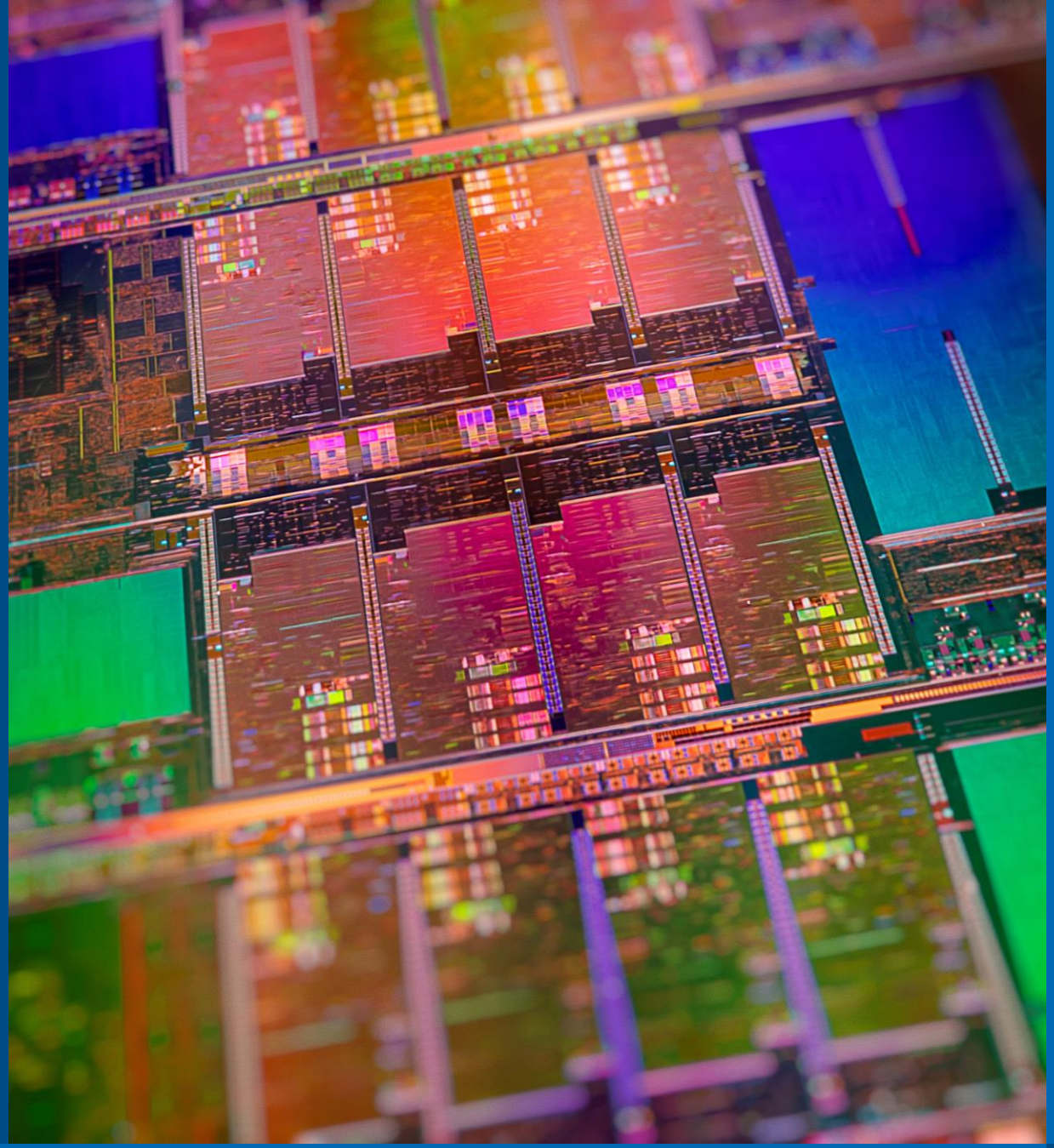

intel

Hardware Security Leak Detection by Symbolic Simulation

Neta Bar Kama
Roope Kaivola

FMCAD 2021

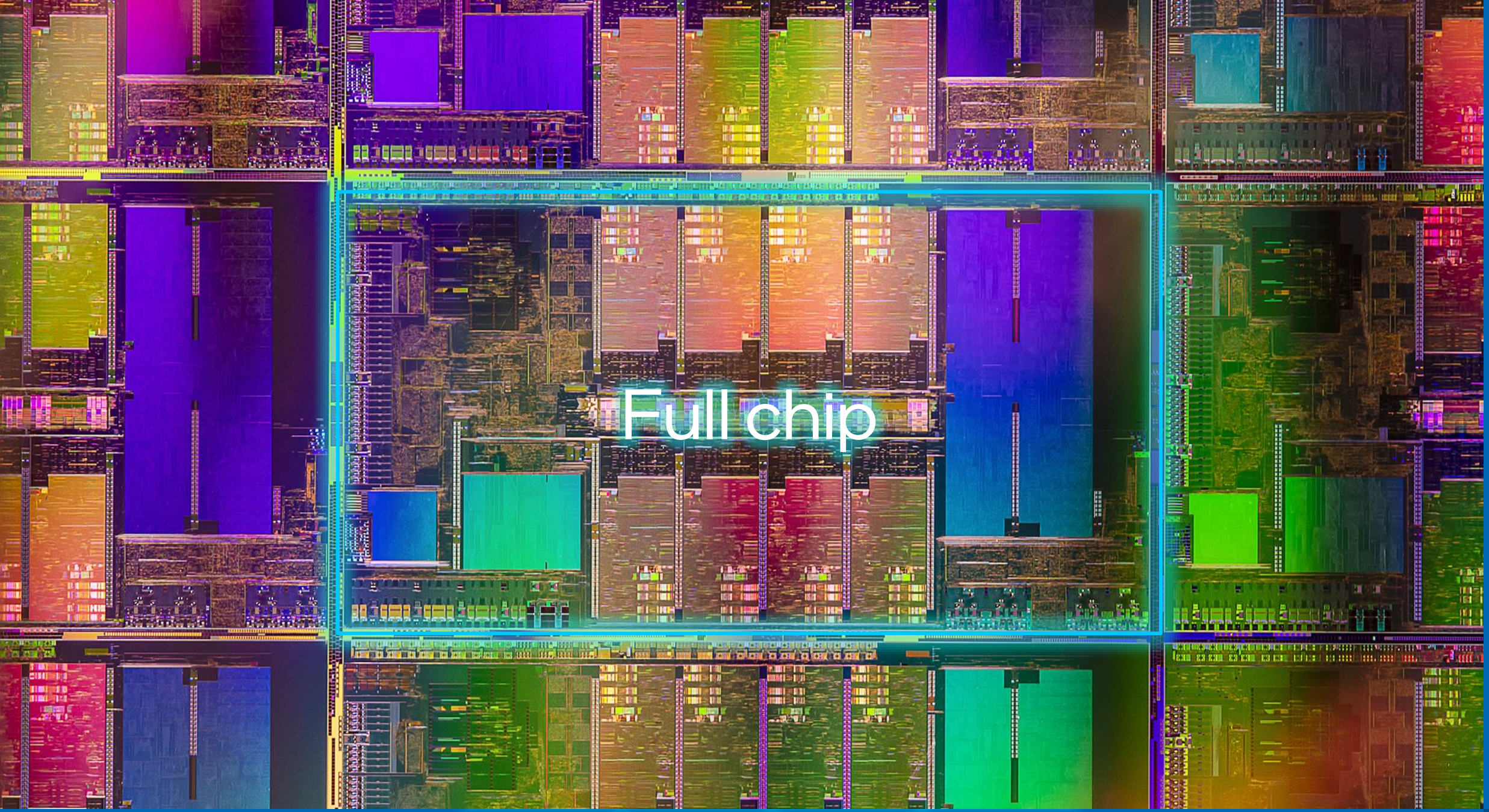


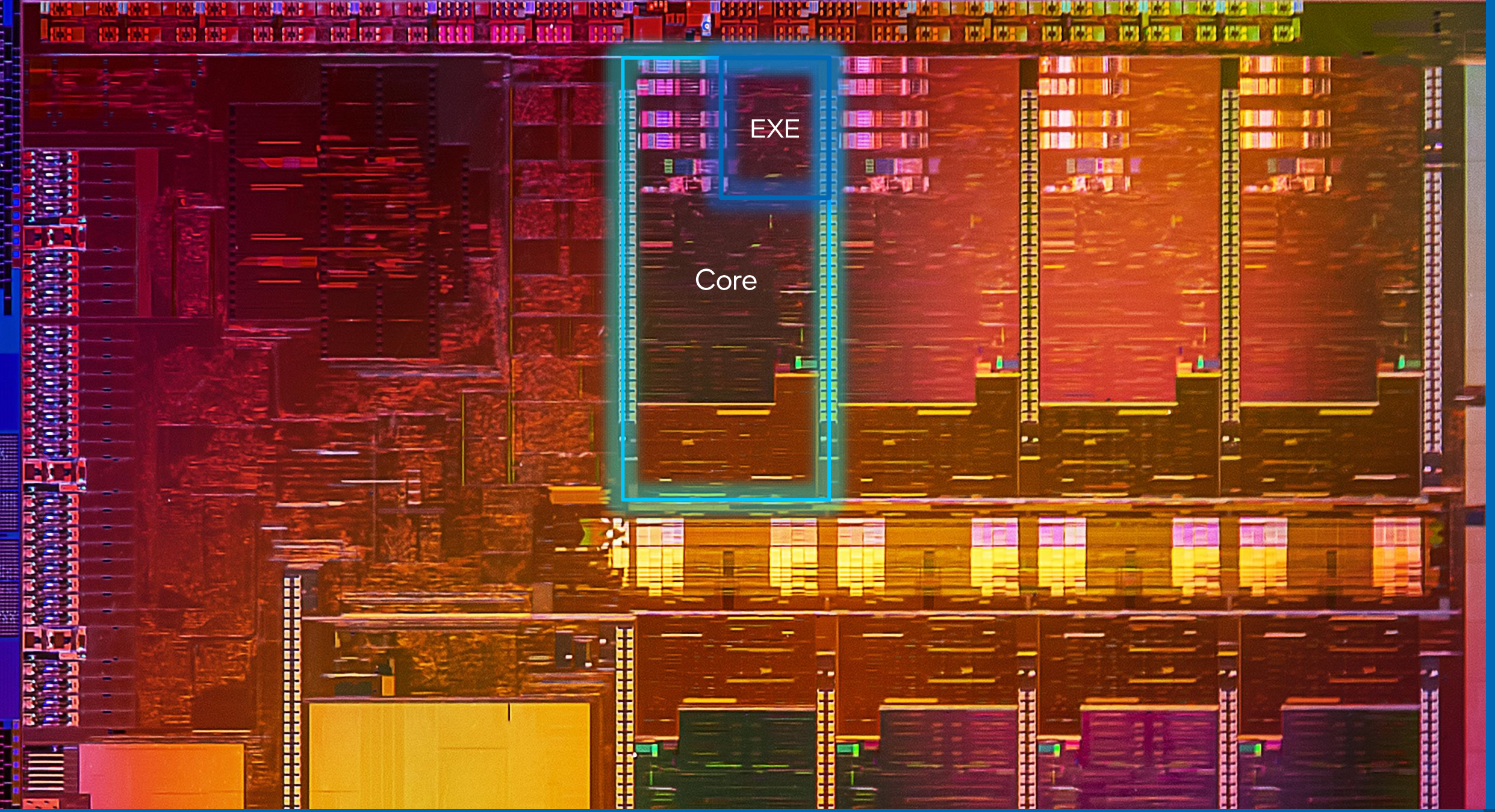
Introduction

- In the aftermath of the Spectre and Meltdown vulnerabilities, security has become a greater focus area for validation.
- Formal verification of arithmetic data-paths has been a focus area at Intel ever since the Pentium® FDIV bug in 1994.
- The primary vehicle for FV in the execution cluster (EXE) on an Intel Core processor is symbolic simulation.
- A novel usage of symbolic simulation led to discoveries of previously unknown potential data leakages in the EXE cluster.

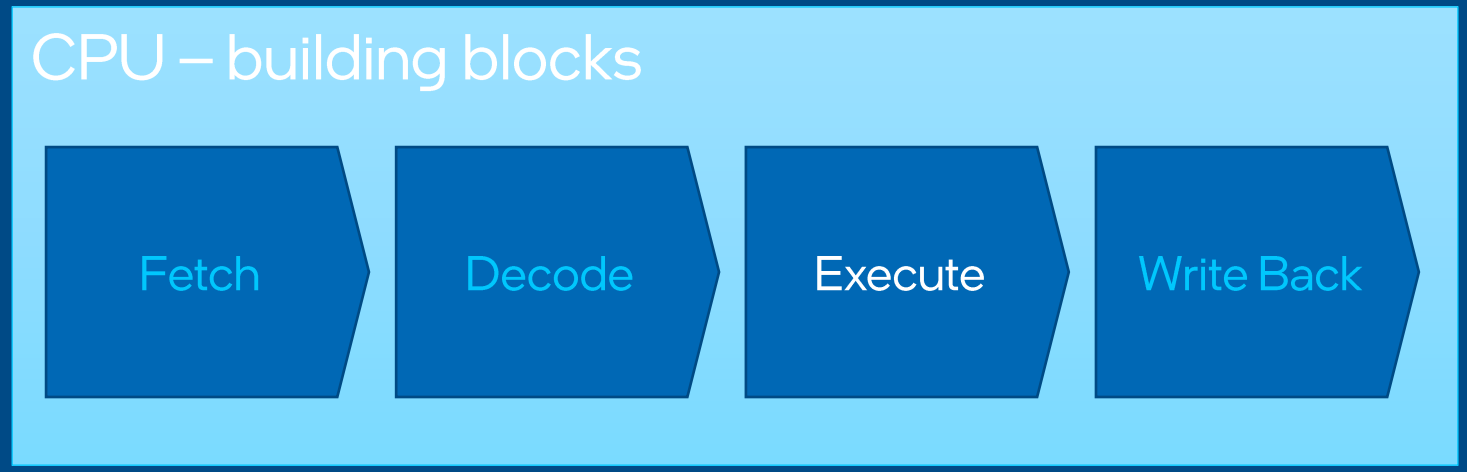
Execution Cluster

Verification and security challenges


Full chip

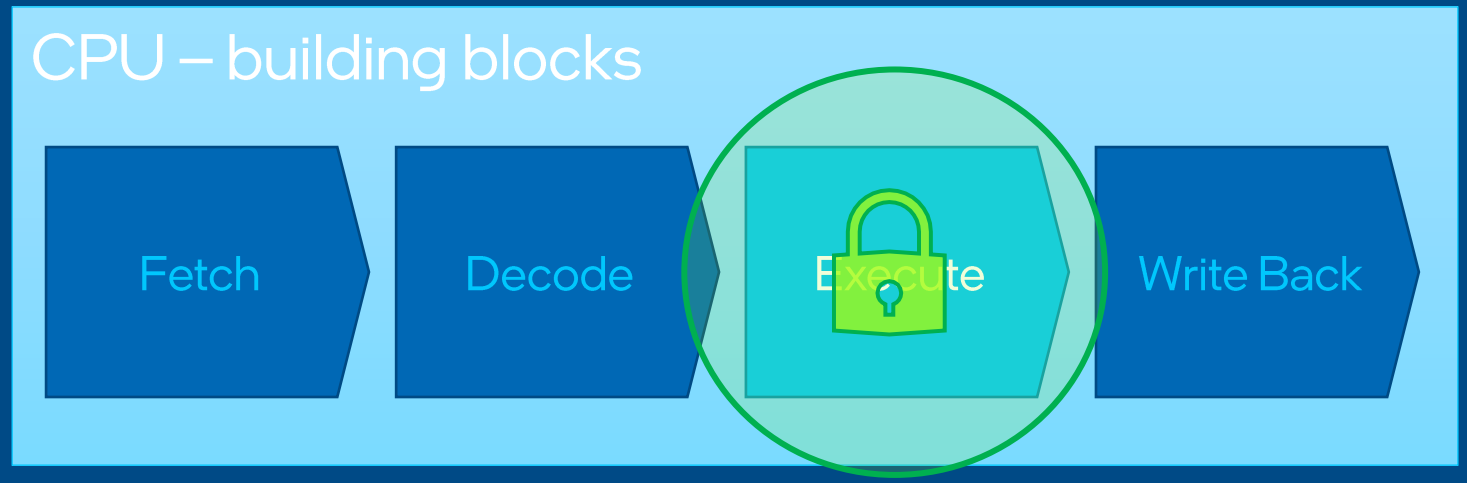

EXE
Core

Execution Cluster



- ~5000 micro-operations in Intel Core Processor EXE cluster.
  - Arithmetic, logic, branch operations, address calculations and more.
- Hundreds of thousands of lines of hardware description language code.
- No prior knowledge of where security vulnerabilities are hiding.

Execution Cluster



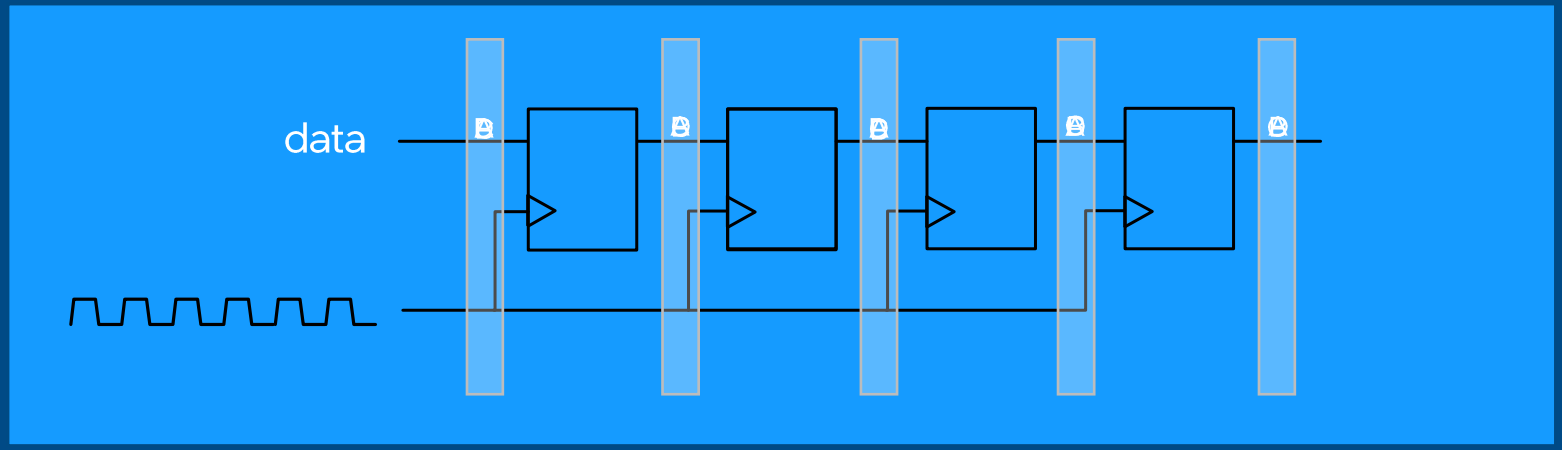
- The execution cluster (EXE) is a pipelined machine.
- Receives streams of micro-operations (μops).
- Data calculations are performed on input sources and result goes to the write-back output.

Execution Cluster

Security challenges:

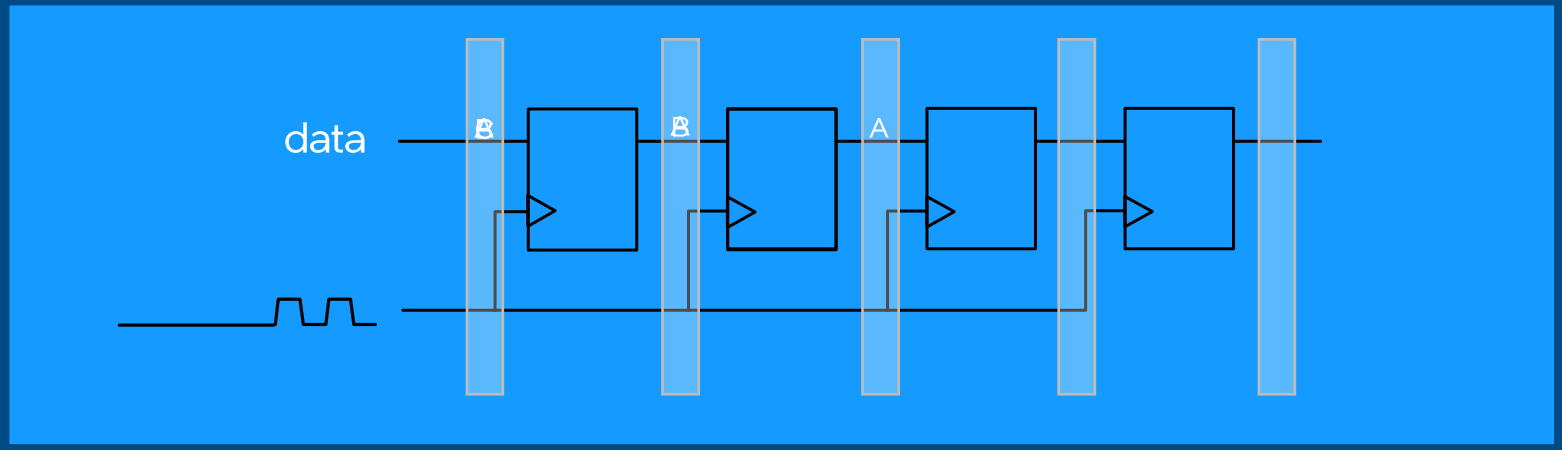
- Same data-path is used for secret and non-secret data.
- No awareness of what a secret is – it is context dependent.
- When clocks are powered down, data lingers in internal flops.
- This data may be exposed by a later operation, if its result is undefined.

Stale Data



All clocks toggling all the time → data flows freely

Stale Data



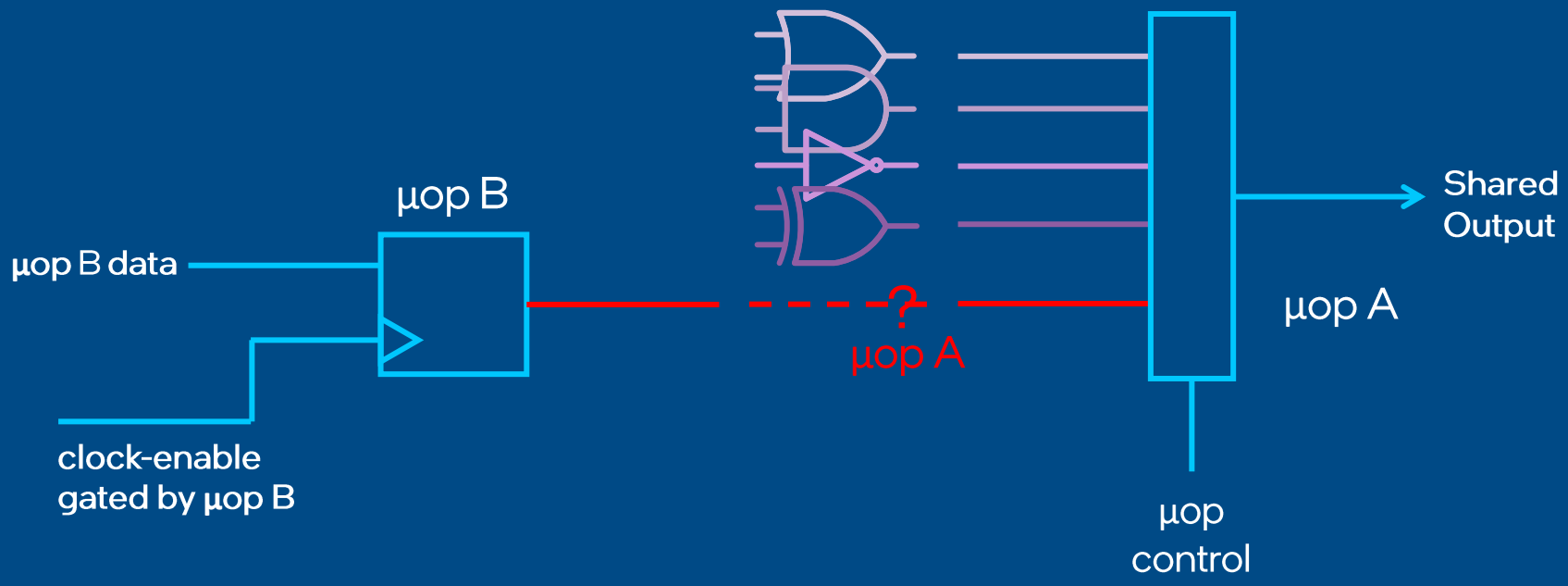
Clocks shut down → data can get stuck

The Undefined Space

- Many micro-operations do not have a fully defined expected result.
  - e.g., divide by zero, writing only flags
- The challenge: how do you define a verification goal and catch a problem without a specification?
- The risk: the cluster output is not checked in these cases. We do not know what kind of data is exposed on the cluster interface when these operations are issued.

The Undefined Space

Example



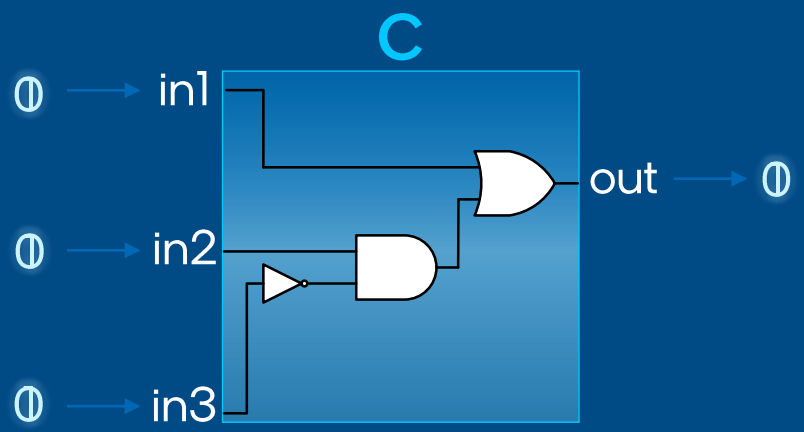
Security property:
Every µop's result depends only on its own inputs

Symbolic Simulation

STE: Symbolic Trajectory Evaluation in a nutshell

Symbolic Simulation

Problem: verify that circuit C satisfies the specification S



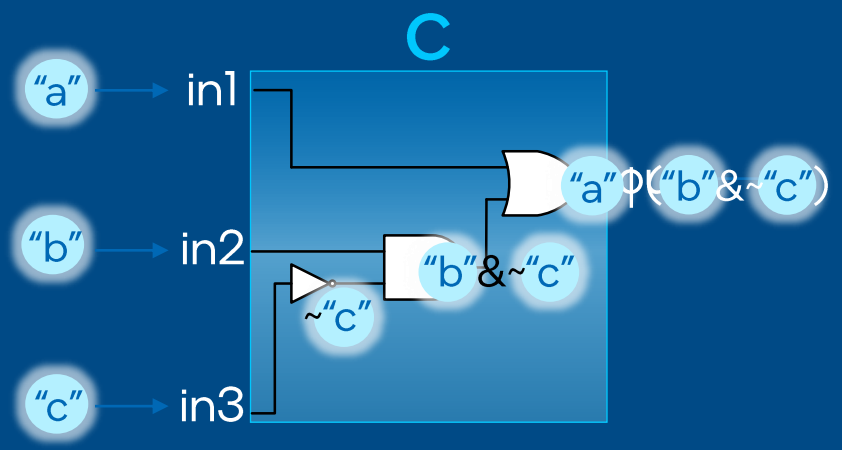
S

S (a,b,c) =
a OR (b AND NOT c)

- Traditional simulation:
  - Inject random values and compare result to reference model.
  - 2^n simulations to cover an n-bit wide logic.

Symbolic Simulation

Problem: verify that circuit C satisfies the specification S



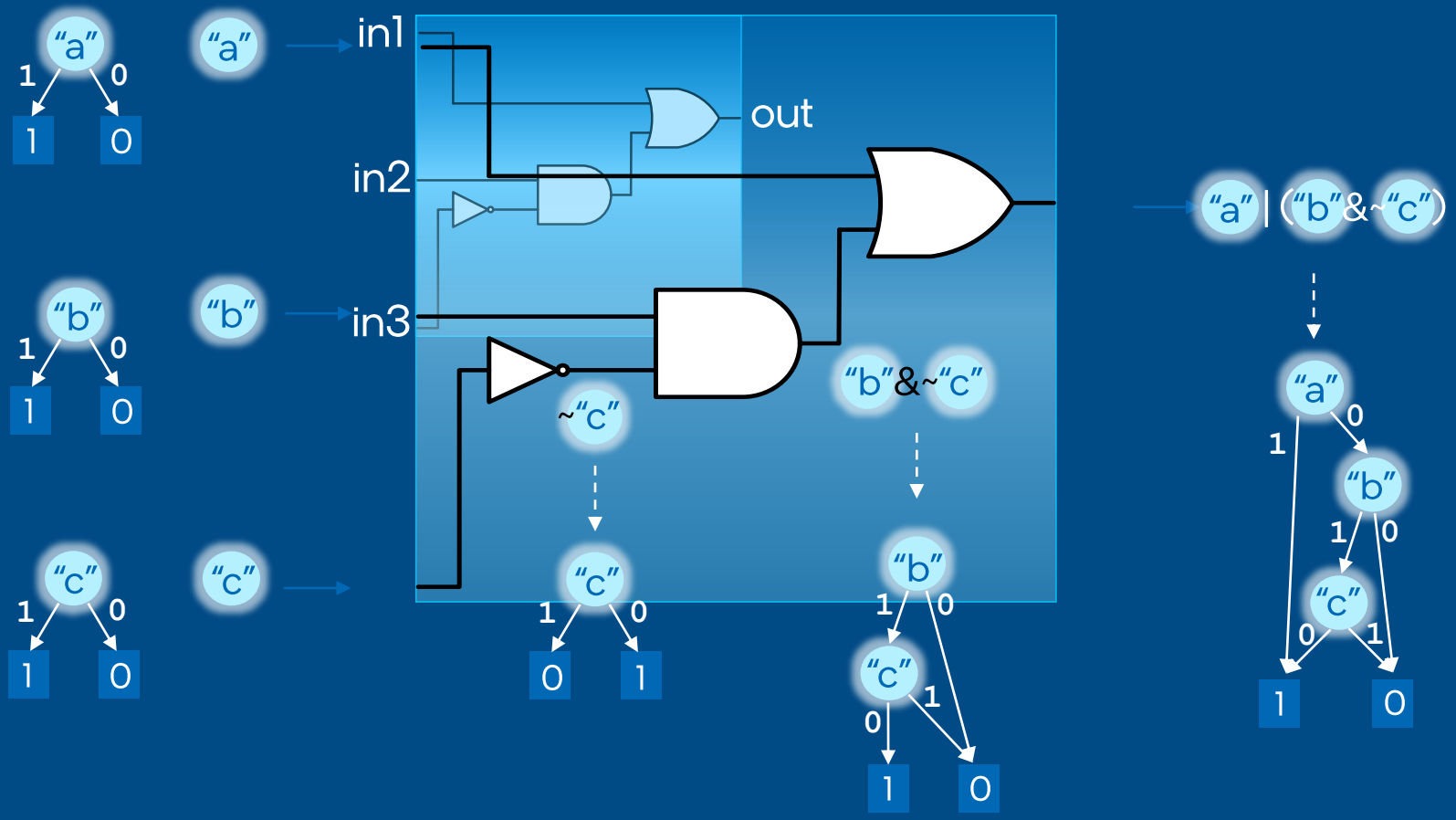
S

S (a,b,c) =
a OR (b AND NOT c)

- Symbolic simulation:
  - Inject symbols and compare result to reference model.
  - 1 simulation to cover all inputs.

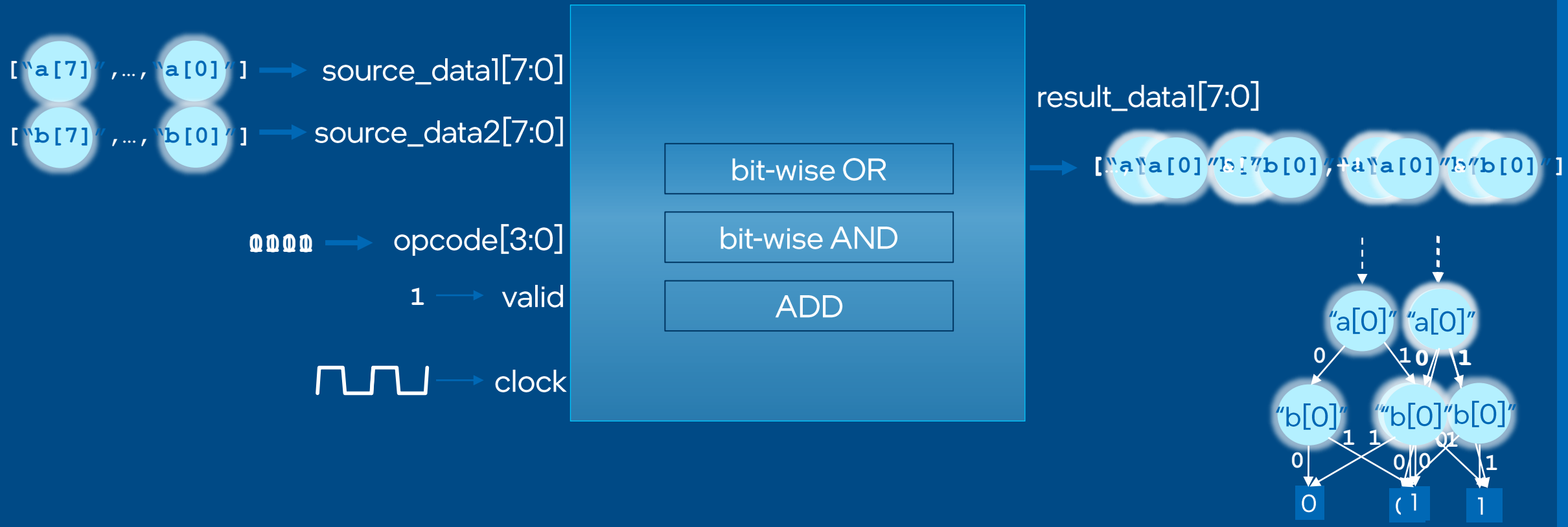
Symbolic Simulation

- What are symbolic expressions?



Symbolic Simulation

- In ‘real’ life

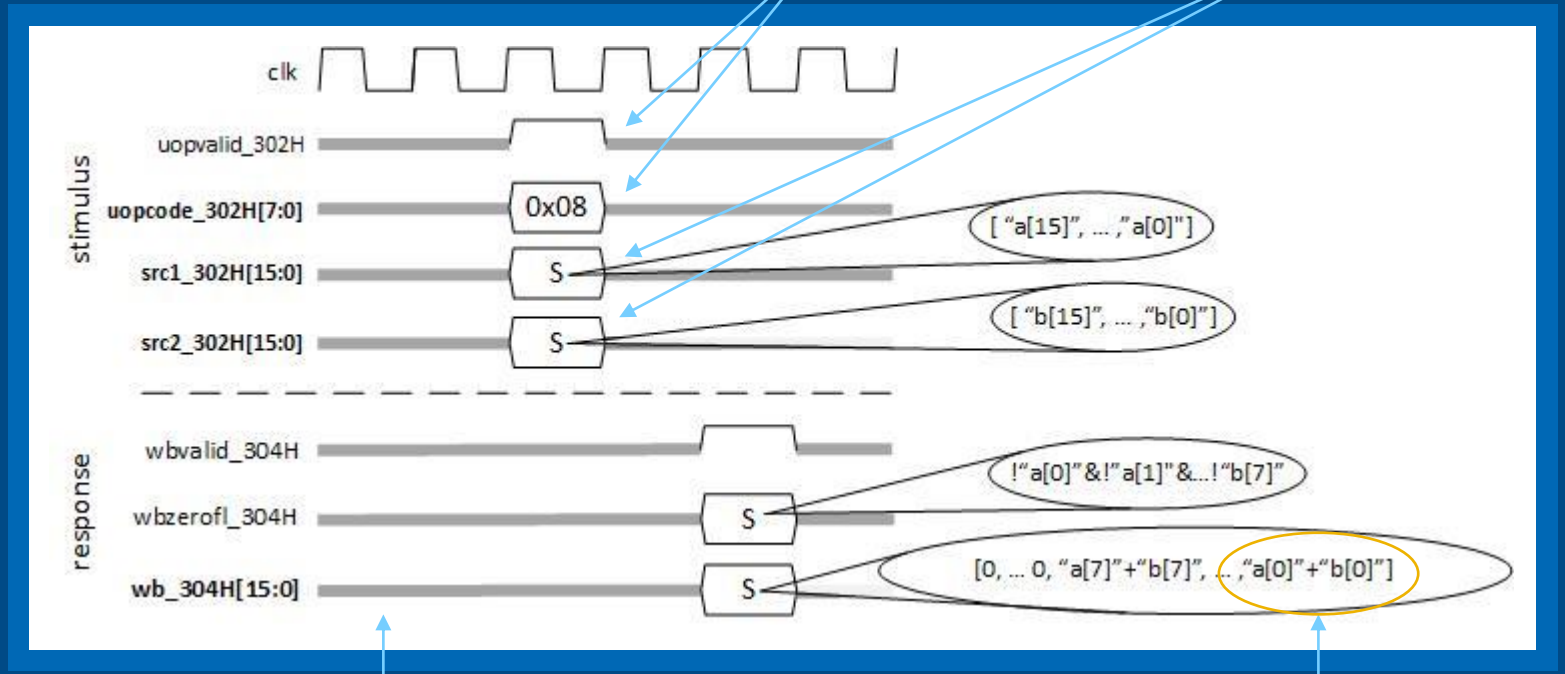


Symbolic Simulation

Waveform example

Constants driven to control signals

Symbols driven to data signals

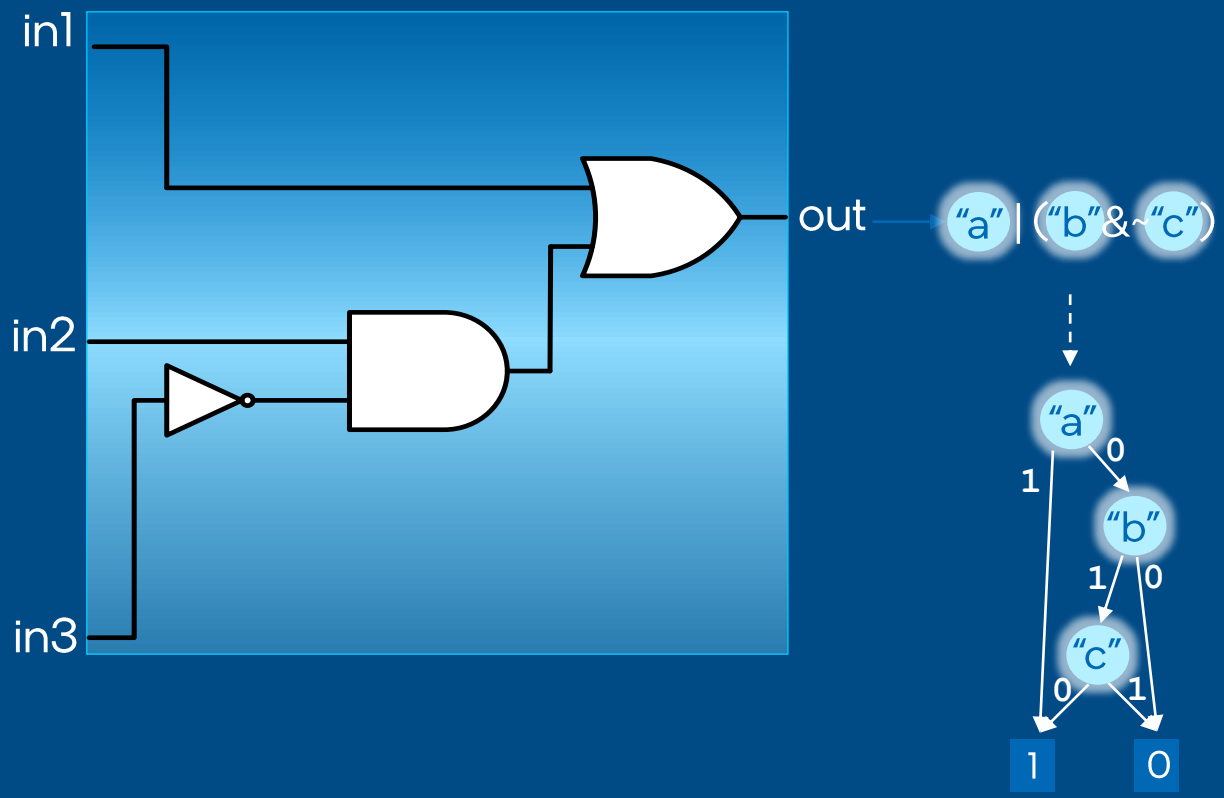


Every bit is a Boolean expression

Grey areas represent X – uninitialized values

Symbolic expressions sampled at the outputs

Symbolic Simulation



Symbolic Simulation

The special trait of symbolic simulation:

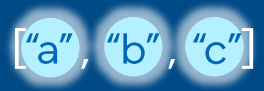
Every variable has a **name**

Circuit output: 

Dependency list: 

The Undefined Space

- **The challenge:** how do you define a verification goal and catch a failure without a specification?
- **Our solution:** the dependency list tells us what propagated to the output, without having to know the specification!

Dependency list: 

Every variable has a **name**

Data Leakage

Process of Detection by Symbolic Simulation

Data Leakage Analysis

How does it work?

- Identify micro-operations (uops) that do not have a fully-defined write-back result.
- Run symbolic simulation, sample the write-back, and extract the list of dependencies.
  - Remember: no need for specification to get the dependency list!
- Identify expected vs. suspicious variables in dependency list.
  - Each variable has a name – easy to filter automatically.
- Debug – where did the suspicious variable come from?

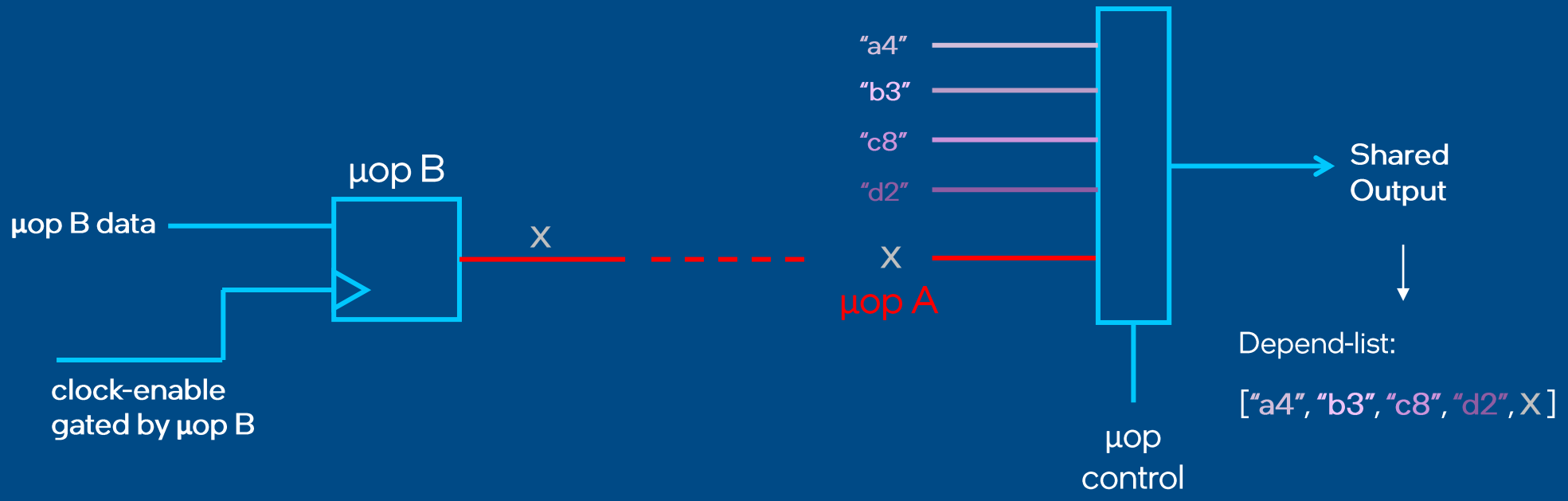
Results and Examples

Results

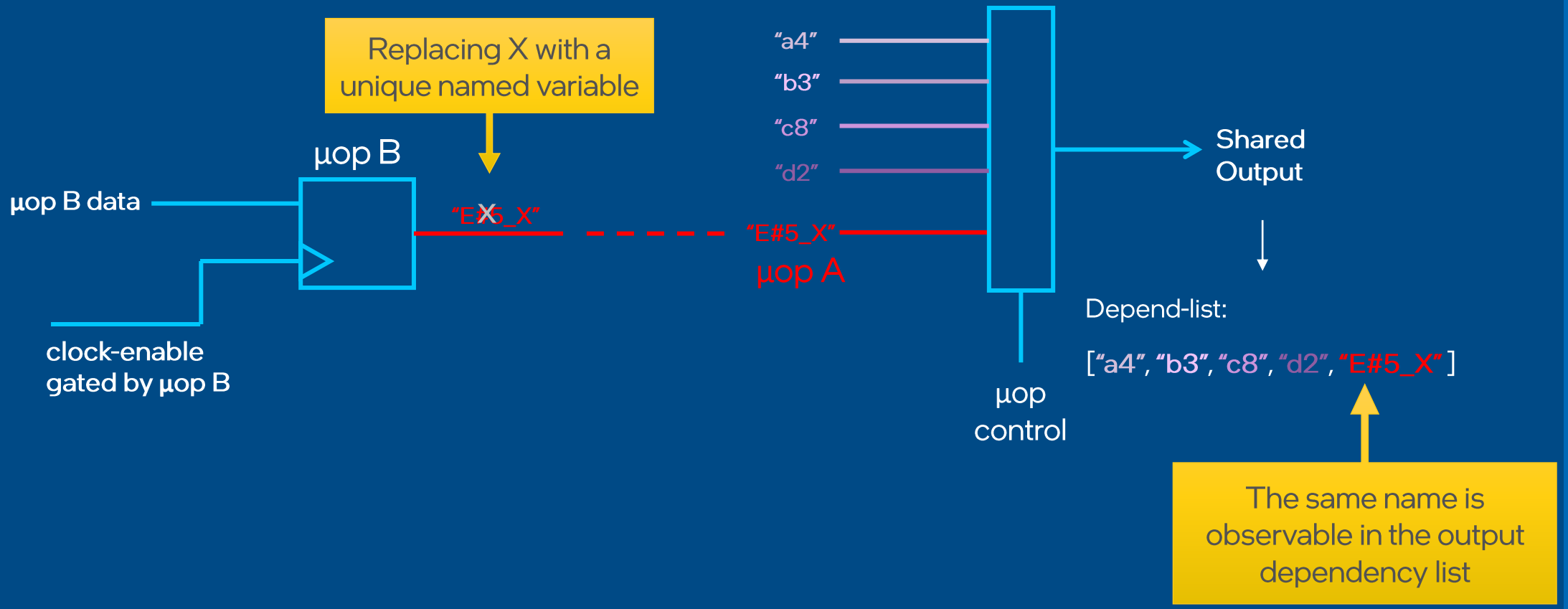
- Focus: write-back data interface buses of ~2000 micro-operations, for which these buses are relevant.
- 89.4% of these were completely specified and 10.6% of had a fully or partially undefined write-back result.
- Symbolic dependency analysis found that only 2.2% failed the symbolic dependency check.

  → 8.4% of micro-operations are undefined, but proven to have only expected data.
- In 6-8 weeks, several potential data leakage mechanisms were detected, all previously unknown.

Example 1

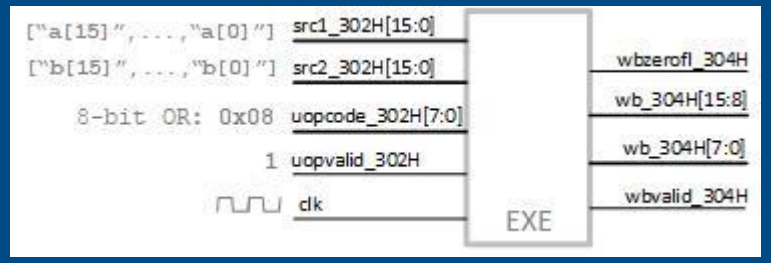

µop B
µop B data
clock-enable
gated by µop B
X
"a4"
"b3"
"c8"
"d2"
X
µop A
µop
control
Shared
Output
Depend-list:
["a4", "b3", "c8", "d2", X]

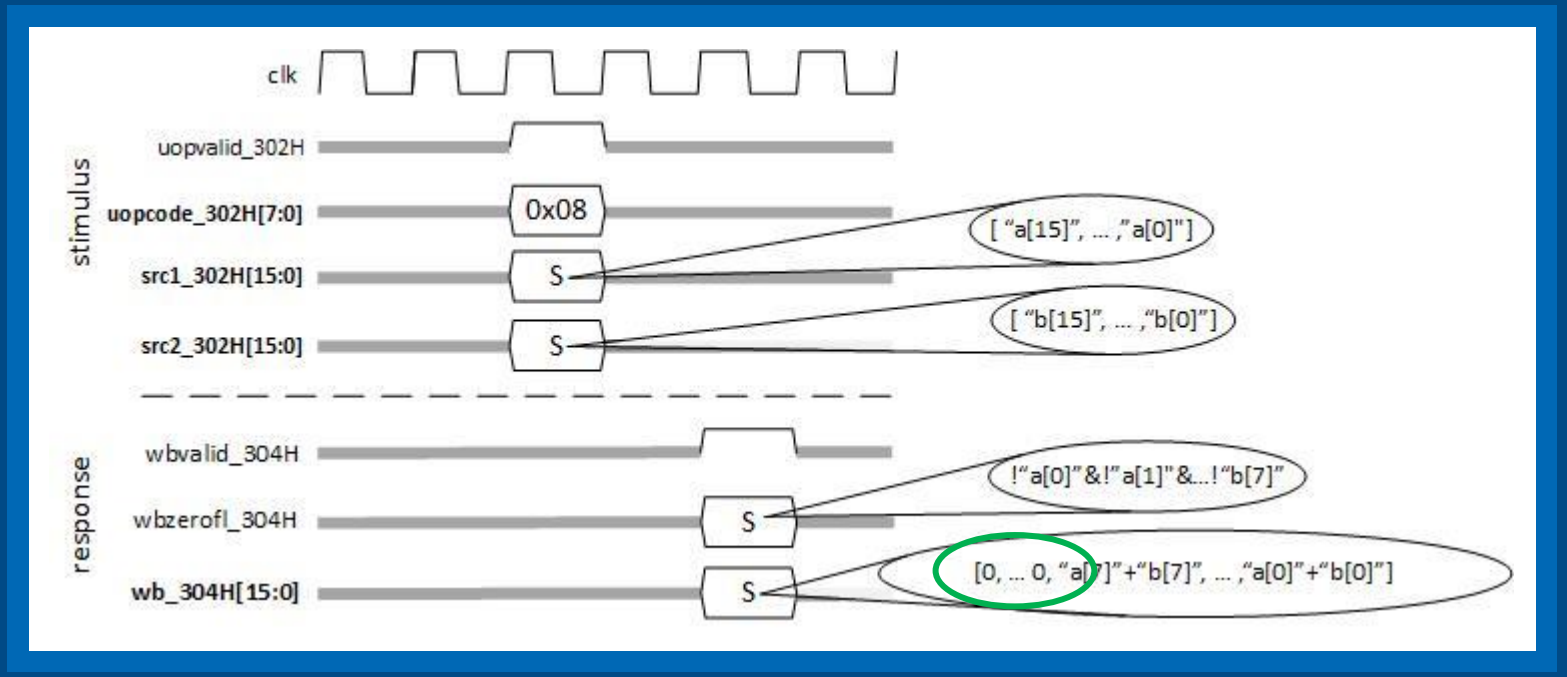
Example 1


Replacing X with a unique named variable
µop B
µop B data
clock-enable gated by µop B
"E#5_X"
"a4"
"b3"
"c8"
"d2"
"E#5_X"
µop A
Shared Output
Depend-list:
["a4", "b3", "c8", "d2", "E#5_X"]
µop control
The same name is observable in the output dependency list

Example 2

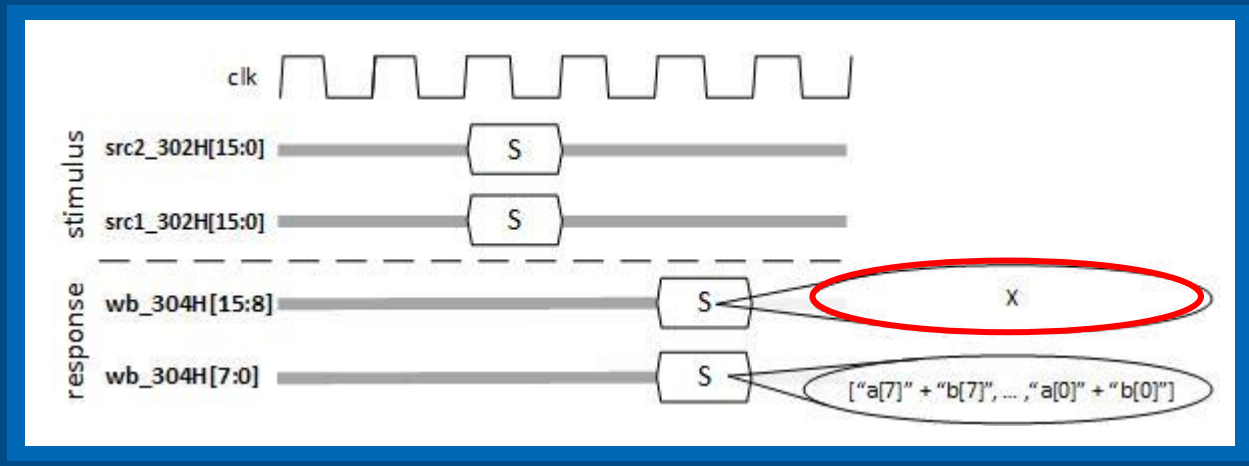
Safe: unused upper bits are zeroed.





Example 2

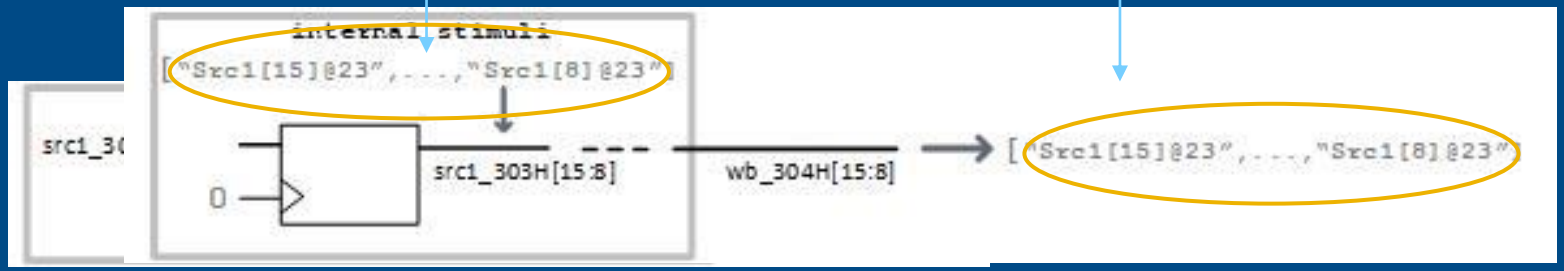
Unsafe: upper bits are X

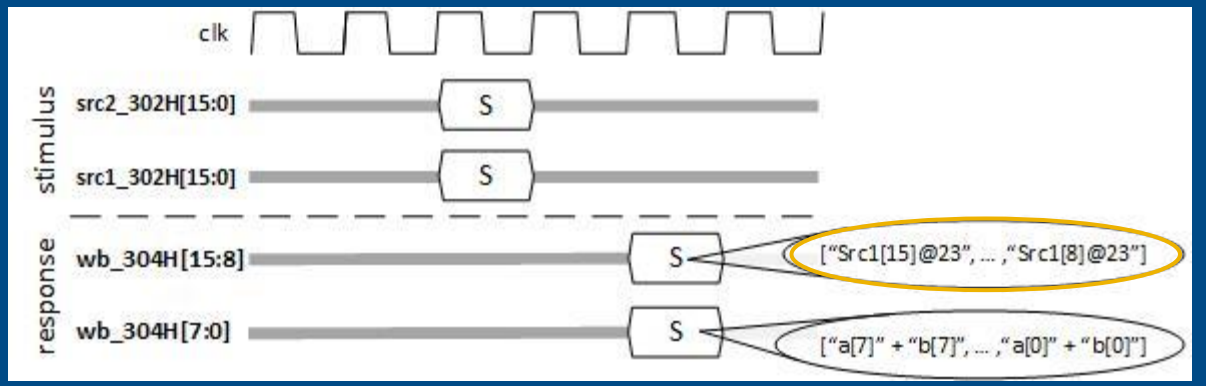


Example 2

Replaced X with a symbol: a unique named variable

Observed same name in output dependency list





Example 2

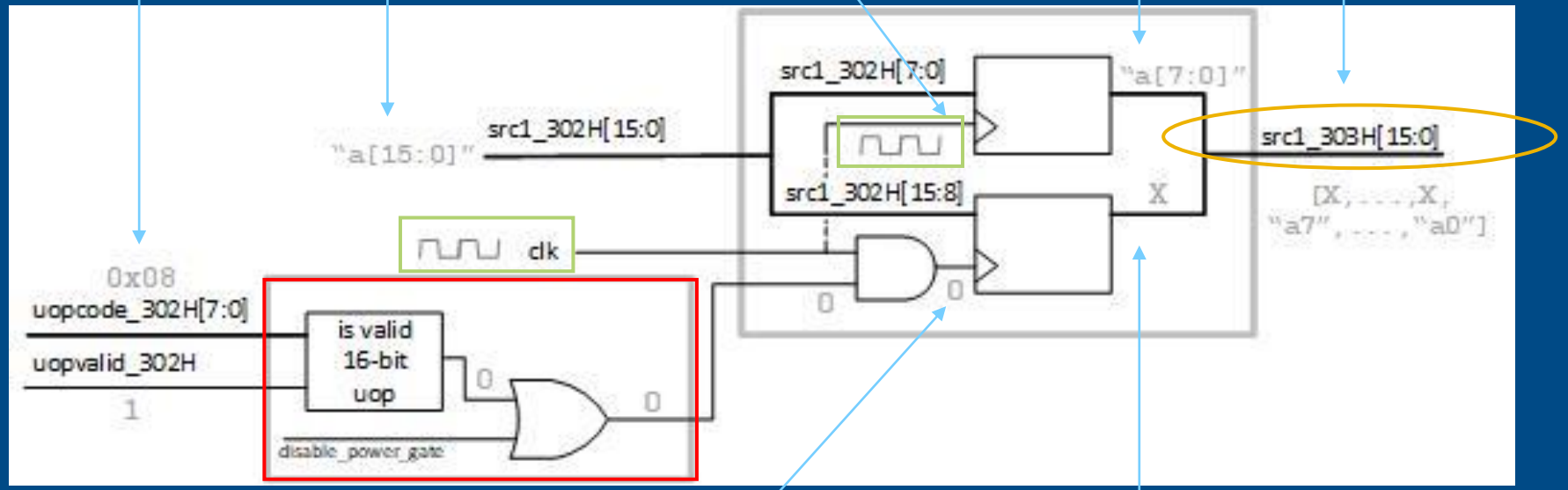
Shared by multiple operations

Symbolic values driven to external source bus

k always ggling

LSB's are always passed on

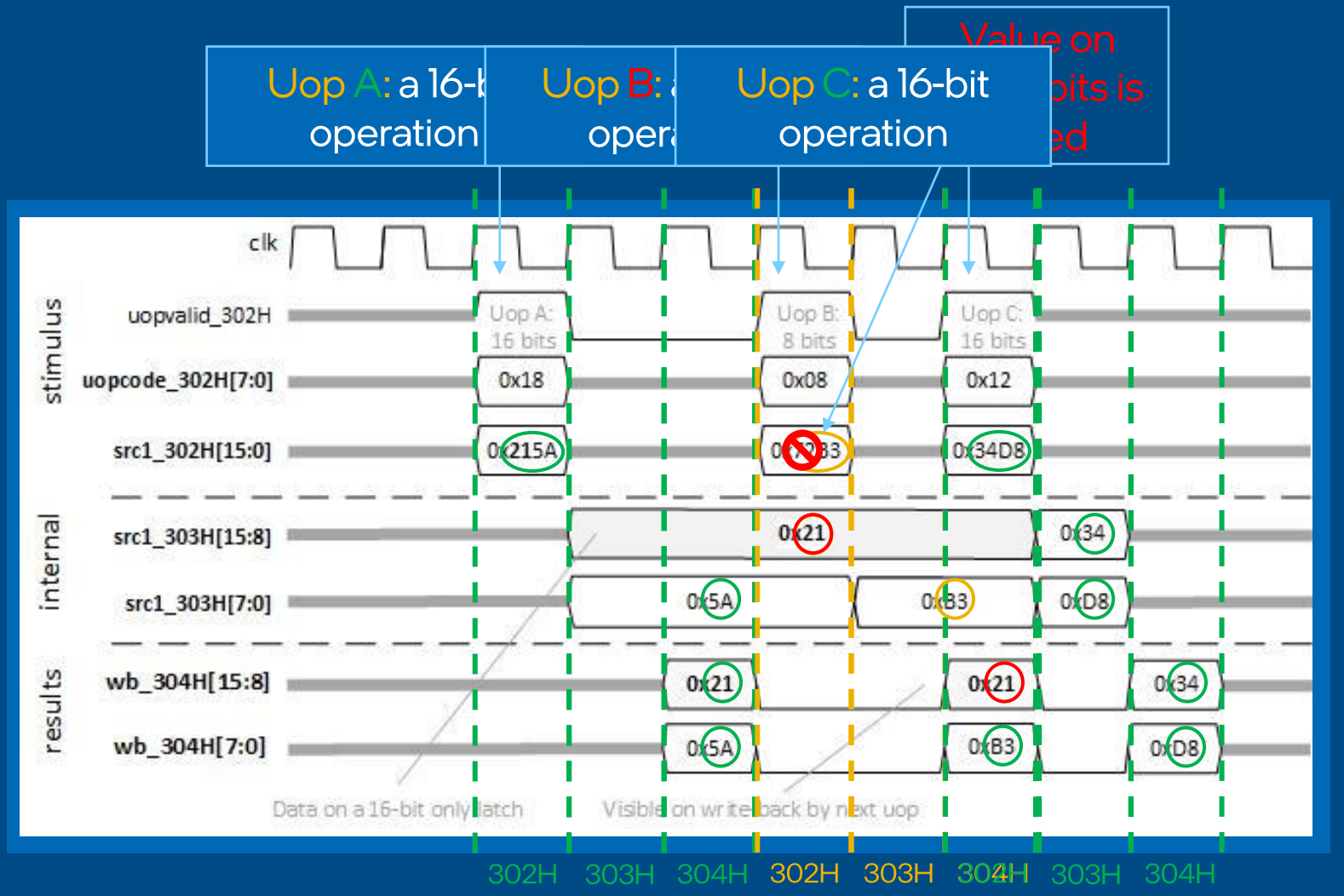
Internal source bus



Gated clock

MSB's are gated

Example 2


Uop A: a 16-bit operation
Uop B: a 8-bit operation
Uop C: a 16-bit operation
Value on bits is ed
clk
stimulus
uopvalid_302H
uopcode_302H[7:0]
src1_302H[15:0]
internal
src1_303H[15:8]
src1_303H[7:0]
results
wb_304H[15:8]
wb_304H[7:0]
Uop A: 16 bits
Uop B: 8 bits
Uop C: 16 bits
0x18
0x08
0x12
0x215A
0x34D8
0x21
0x34
0x5A
0xB3
0xD8
Data on a 16-bit only latch
Visible on write back by next uop
302H
303H
304H
302H
303H
304H
303H
304H

Summary

- Cluster level coverage, 6-8 weeks of work
- Found that 97.8% of micro-operation pose no risk, including 8.4% that were not fully defined.
- Failures were mapped to several potential data leakage mechanisms, all previously unknown.

Acknowledgments

We would like to thank:

- Arkady Neyshtadt for his security analysis
- Gilad Holzstein, Robert Jones, Alex Levin, Yoav Moratt and Nir Shildan for interesting discussions on security
- Annette Upton for detailed feedback on the paper
- David Turner, Yaniv Dana and Alon Flaisher for the opportunity to carry out this work
- Orly Cohen, Joe Leslie-Hurd, Chris Gaines, Michael Libby and Jesse Bingham for presentation feedback


intel