C-S²QED Gap-Free Verification of Processor Cores

Presentation · October 2020

CITATIONS
0

READS
128

2 authors:

Keerthikumara Devarajegowda
Siemens EDA
32 PUBLICATIONS **171** CITATIONS
SEE PROFILE

Mohammad Rahmani Fadiheh
RPTU - Rheinland-Pfälzische Technische Universität Kaiserslautern Landau
12 PUBLICATIONS **81** CITATIONS
SEE PROFILE

C-S²QED
Gap-Free Verification of Processor Cores

Keerthi Devarajegowda and Mohammad R. Fadiheh

accellera
SYSTEMS INITIATIVE

2020
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE

Collaborators

Infineon Technologies,TU Kaiserslautern, Stanford University

Outline

- Motivation
- Processor Bugs
- Processor Verification by C-S²QED
- Experimental Results
- Demo
- Conclusion

Pre-Silicon Verification

- Very few “real” innovations over past decade
  - Race against time & complexity
- Product features limited by verification
  - Existing techniques not sustainable

Pre-Silicon Verification

- Very few “real” innovations over past decade
  - Race against time & complexity
- Product features limited by verification
  - Existing techniques not sustainable

Processor Verification (is hard)

- Highly complex microarchitectural optimizations
- Consumes **≥70%** of design cycle
- Requires in-depth microarchitecture knowledge
- Requires high verification expertise
- Simulation inadequate

Processor Verification (anecdote)

Industrial microcontroller

- 60 instructions, 1.8k flipflops
- Highly optimized microarchitecture
- Safety critical applications

3D camera

Airbag

Pressure Monitor

Processor Verification (anecdote)

Industrial microcontroller

- 60 instructions, 1.8k flipflops
- Highly optimized microarchitecture
- Safety critical applications

3D camera

Airbag

Method	Effort	
	Initial RTL	Updated RTL
Directed simulation	5 person month	1-3 person month
Constrained random simulation	12 person month	3-6 person month
Formal verification	5 person month	1 person month

Pressure Monitor

Processor Verification (anecdote)

Industrial microcontroller

- 60 instructions, 1.8k flipflops
- Highly optimized microarchitecture
- Safety critical applications

3D camera

Airbag

We need new methods for Pre-Silicon Verification!

we propose, **Complete S²QED (C-S²QED)**

A gap-free processor verification method

- Highly automated formal verification technique
- Extends S²QED to satisfy completeness criterion [Bormann05,Nguyen08,Fadiheh18]
- Detects all functional bugs during pre-silicon verification
- Simplifies the property set
- Requires low manual effort and low formal expertise

[Bormann05] "Method for determining the quality of a set of properties", EP1764715, 09 2005

[Nguyen08] "Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants", TCAD, 2008

[Fadiheh18] "Symbolic Quick Error Detection using Symbolic Initial State for Pre-silicon Verification," DATE 2018

Terminology: Processor Bugs

Logic bugs in a processor can be categorized as:

- Single-instruction bugs
- Multiple-instruction bugs

Terminology: Processor Bugs

Logic bugs in a processor can be categorized as:

- Single-instruction bugs
- Multiple-instruction bugs

Example program:

…	*//assume [R1] = $0xFFFF$; [R2] = 1*
$BNE \quad R1, R2, \#20$	*// [R1] ≠ [R2]? PC=PC+20 : PC=PC+4*

Terminology: Processor Bugs

Logic bugs in a processor can be categorized as:

- Single-instruction bugs
- Multiple-instruction bugs

Example program:

$\ldots$ *//assume [R1] = $0xFFFF$; [R2] = 1*

$BNE \quad R1, R2, \#20$ *// [R1] ≠ [R2]? PC=PC+20 : PC=PC+4*

- Bug occurs always, due to wrong branch control for ***BNE***
- Program context is irrelevant for single-instruction bugs

Terminology: Processor Bugs

Logic bugs in a processor can be categorized as:

- Single-instruction bugs
- Multiple-instruction bugs

Example program:

```
LW R4, R3, #1      // bug activation – step 1
ADD R2, R3, R4     // bug occurs – wrong forwarding of R4
NOP                // bug not activated
NOP                // bug not activated
ADD R2, R3, R4     // ADD executed without error
```

Terminology: Processor Bugs

Logic bugs in a processor can be categorized as:

- Single-instruction bugs
- Multiple-instruction bugs

Example program:

```
LW R4, R3, #1      // bug activation – step 1
ADD R2, R3, R4     // bug occurs – wrong forwarding of R4
NOP                // bug not activated
NOP                // bug not activated
ADD R2, R3, R4     // ADD executed without error
```

- Bug occurs due to errors in the hazard detection unit
- Program context is relevant for multiple-instruction bugs

Processor Bugs: Observation

- A single-instruction bug consists of:
  - An instruction that activates the bug
  - The same instruction propagating the bug into program-visible registers

Example program:

```
…                          // assume [R1] = 0xFFFF; [R2] = 1
BNE  R1, R2, #20           // [R1] ≠ [R2]? PC=PC+20 : PC=PC+4
```

Processor Bugs: Observation

- A single-instruction bug consists of:
  - An instruction that activates the bug
  - The same instruction propagating the bug into program-visible registers

Example program:

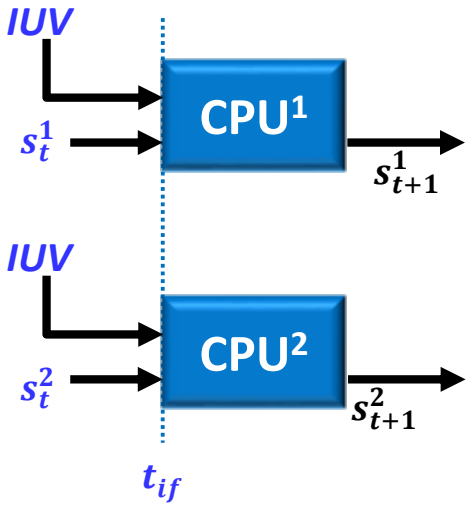
```
...                          // assume [R1] = 0xFFFF; [R2] = 1
BNE  R1, R2, #20             // [R1] ≠ [R2]? PC=PC+20 : PC=PC+4
```

- A multiple-instruction bug consists of:
  - A sequence of instructions activating the bug
  - One instruction propagating the bug into program-visible registers

Example program:

```
LW R4, R3, #1                // bug activation – step 1
ADD R2, R3, R4               // bug occurs – wrong forwarding of R4
NOP                          // bug not activated
NOP                          // bug not activated
ADD R2, R3, R4               // ADD executed without error
```

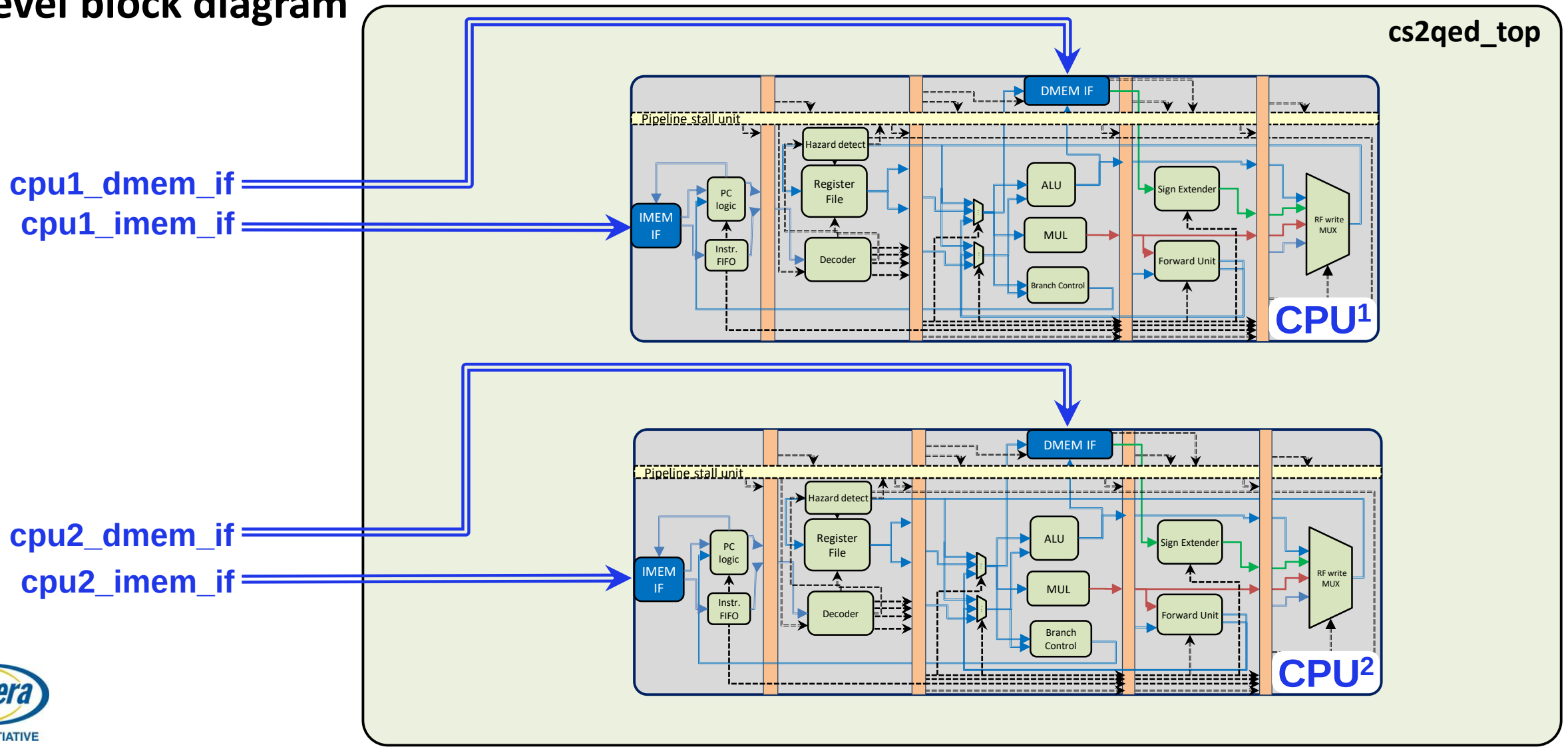
C-S²QED: Verification Setup



- Consider a pipelined processor core
- Two identical instances ***CPU¹*** and ***CPU²*** executing in parallel
- At t_{if}, both CPUs fetch the bug-exposing instruction ***(IUV)***
- They start in different starting states s_t^1 and s_t^2

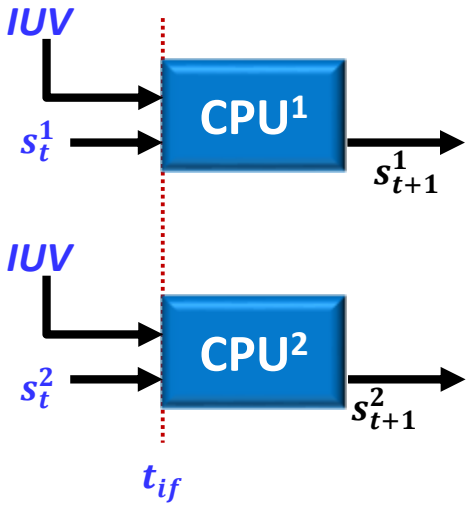
C-S²QED: Verification Setup

Top-level block diagram



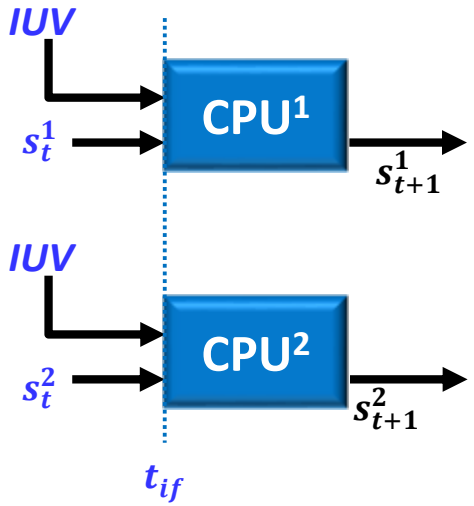
accellera SYSTEMS INITIATIVE

C-S²QED: Verification Setup



- s_t^1 is a flushed-pipeline state
- s_t^2 is symbolic (unconstrained)

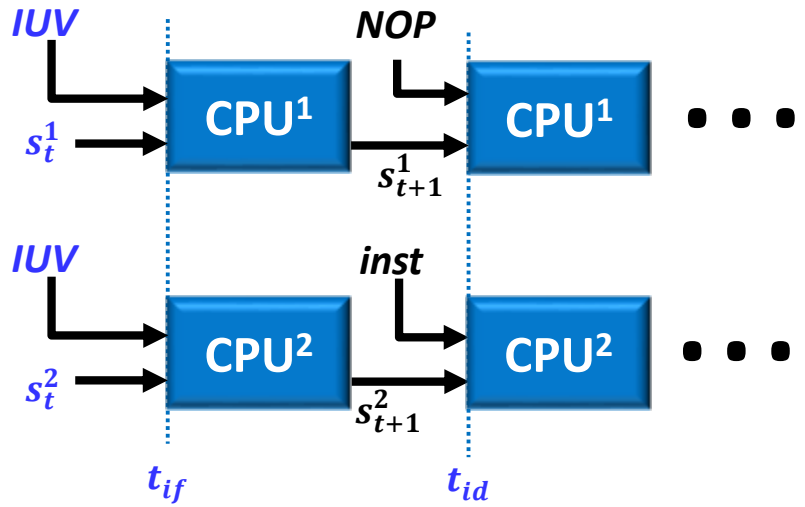
C-S²QED: Verification Setup



- s_t^1 is a flushed-pipeline state
- s_t^2 is symbolic (unconstrained)

```
assume:
        at t_if: cpu1_fetched_instr() = cpu2_fetched_instr();
        at t_if: cpu1_state = flushed_pipeline();
```

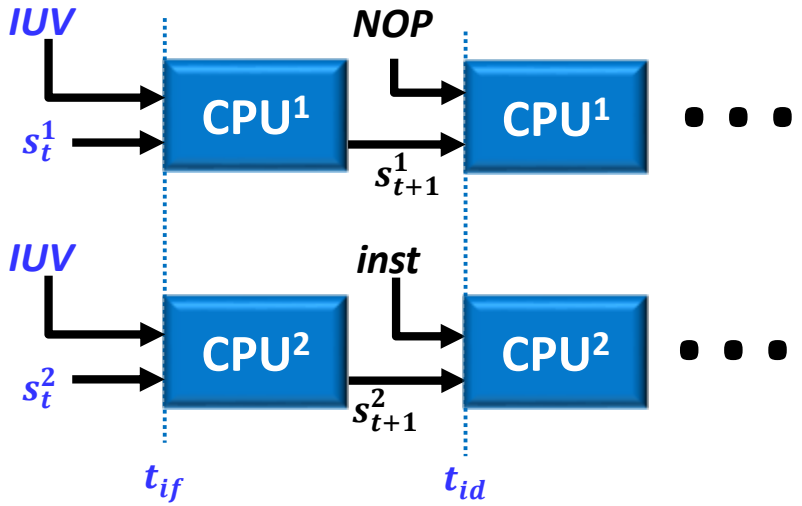
C-S²QED: Verification Setup



- s_t^1 is a flushed-pipeline state
- s_t^2 is symbolic (unconstrained)

```
assume:
        at t_if: cpu1_fetched_instr() = cpu2_fetched_instr();
        at t_if: cpu1_state = flushed_pipeline();
        at t_id: instr_register_type();
```

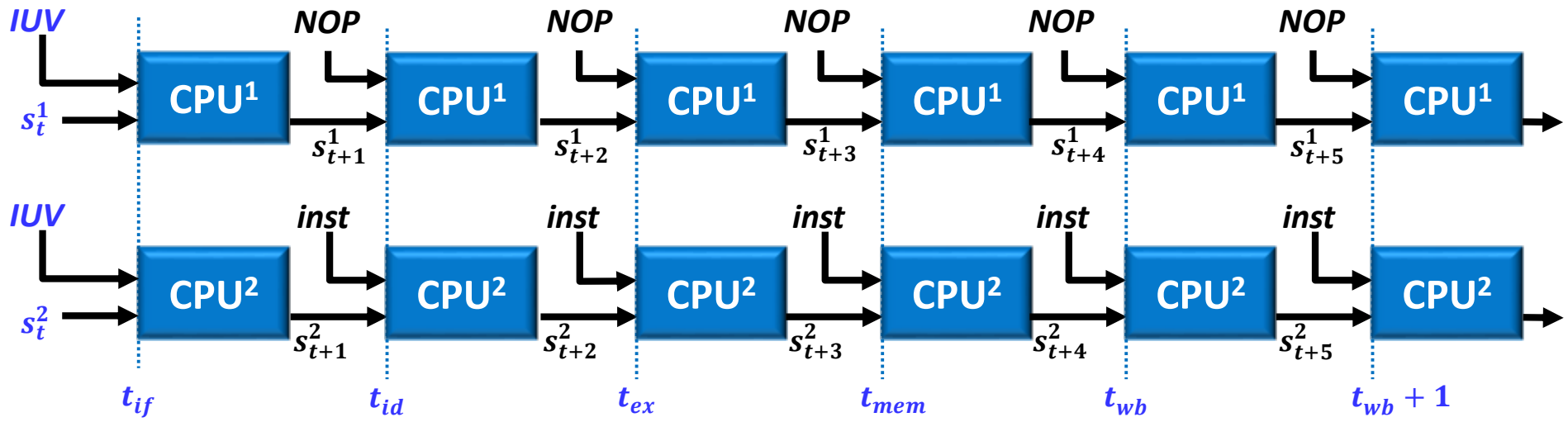
C-S²QED: Verification Setup



The two instances are unrolled for a time window as large as the execution time of an instruction.

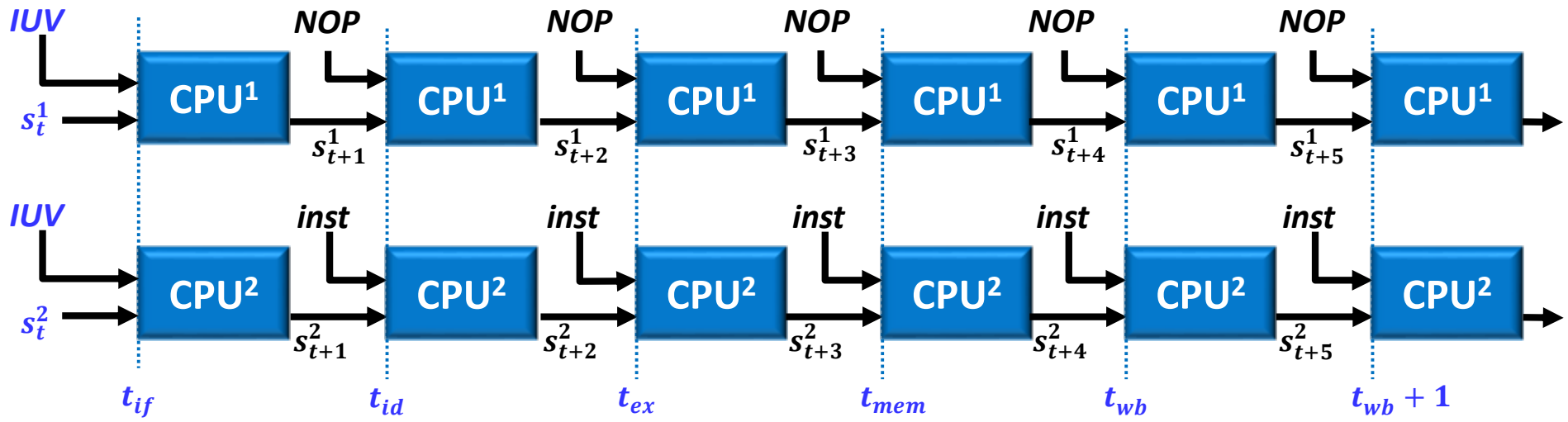
```
assume:
        at t_if: cpu1_fetched_instr() = cpu2_fetched_instr();
        at t_if: cpu1_state = flushed_pipeline();
        at t_id: instr_register_type();
```

C-S²QED: Verification Setup



Example, for a 5-stage pipeline processor

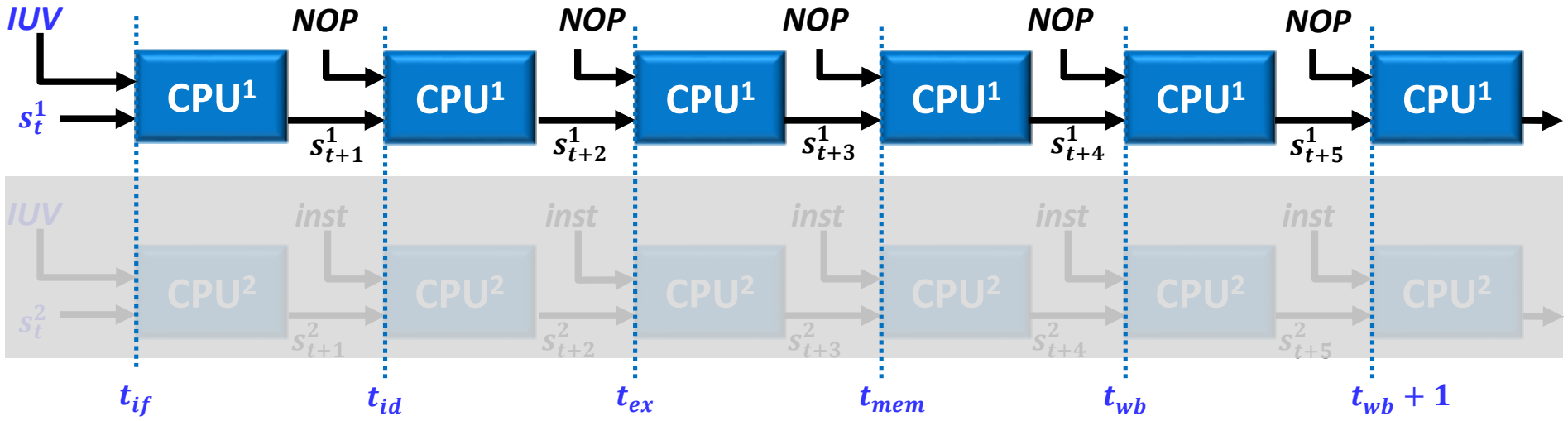
C-S²QED: Verification Setup



between$[t_{if}, t_{wb}+1]$:

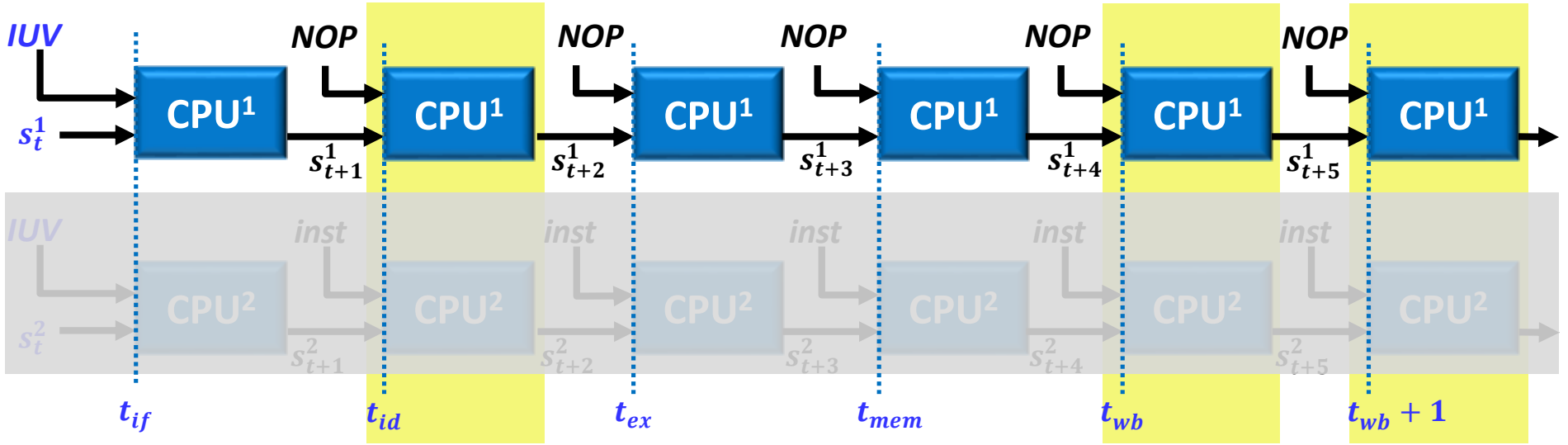
- ***CPU¹*** fetches only ***NOPs***
- ***CPU²*** fetches arbitrary instructions

C-S²QED finds all **Single-Instruction Bugs**



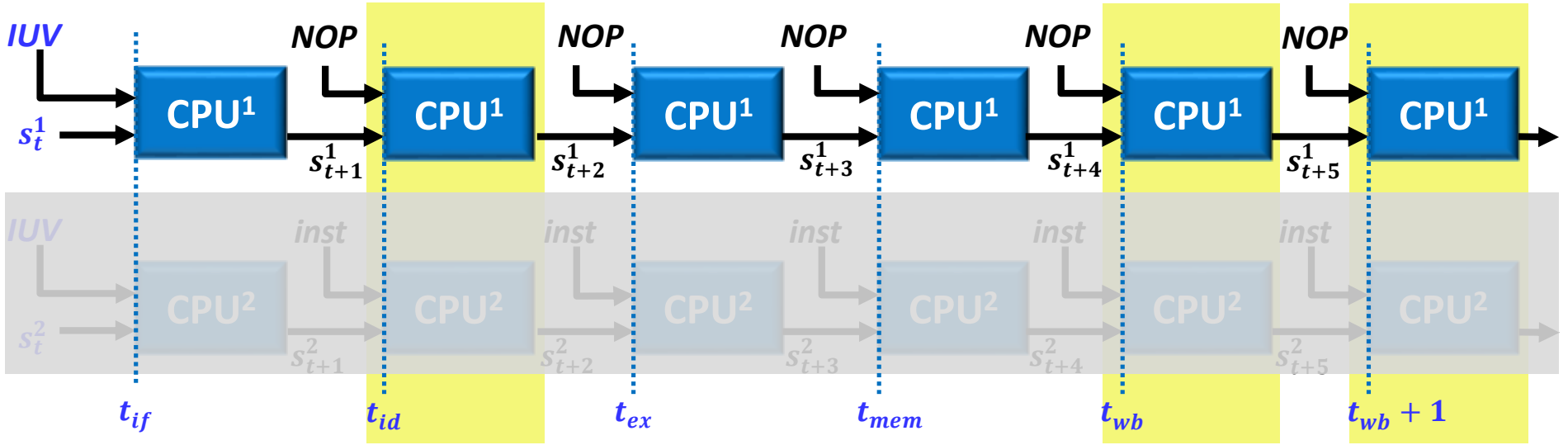
Let's consider the ***CPU¹*** instance

C-S²QED finds all **Single-Instruction Bugs**



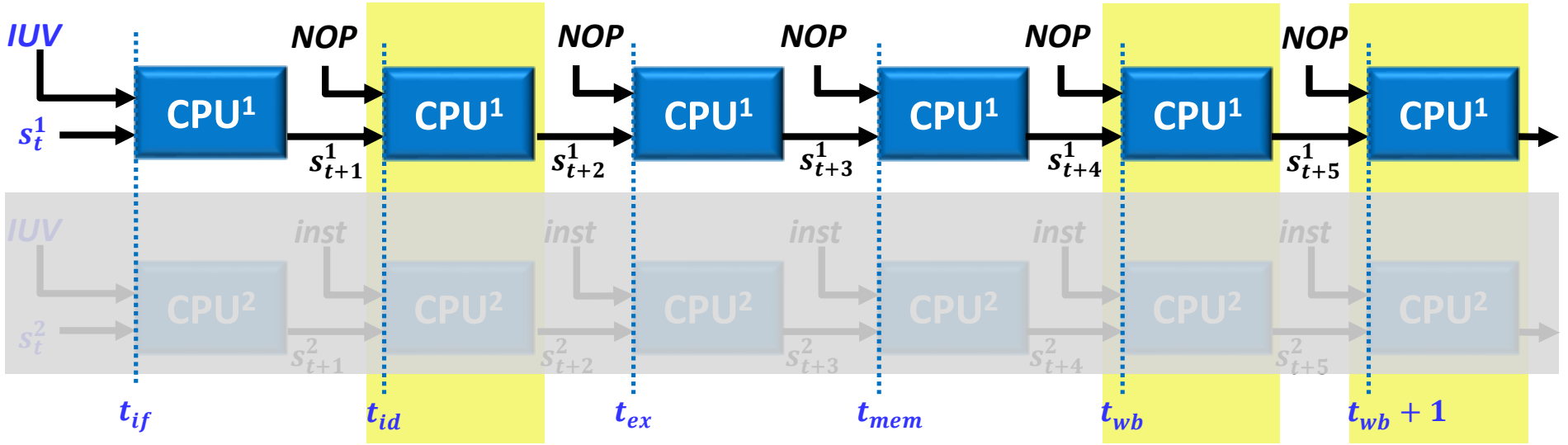
- @t_{id}: src/dest registers, immediate value, operations to be performed are decoded
- @t_{wb}: results of the ***IUV*** are written to the register file
- @$t_{wb}+1$: results are visible in the register file

C-S²QED finds all **Single-Instruction Bugs**



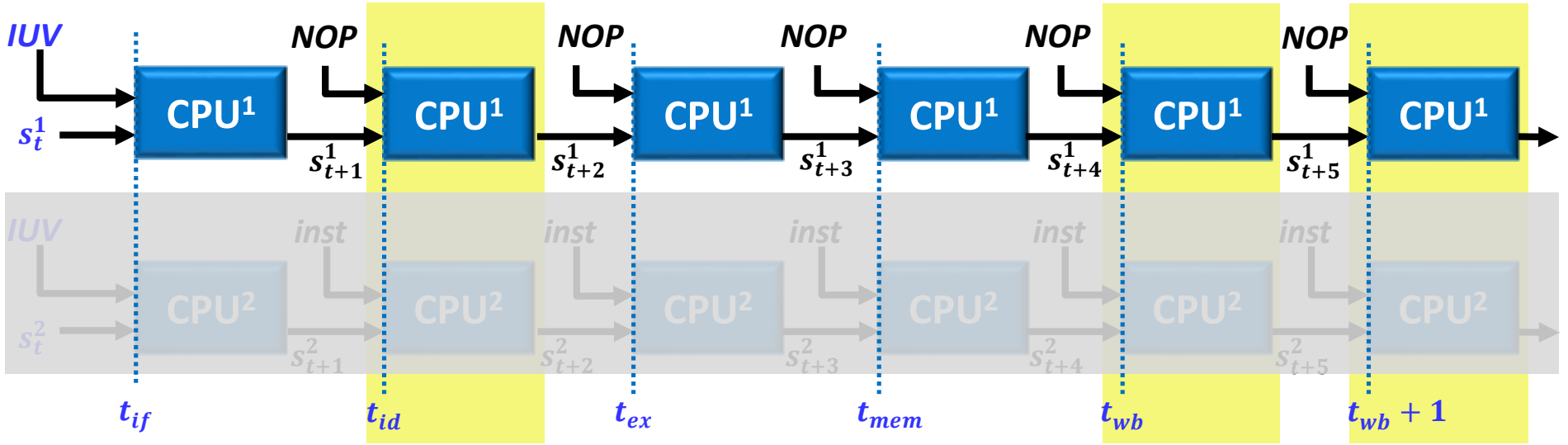
- at t_{if}, ***CPU¹*** is in a flushed pipeline state s_t^1
- i.e., ***IUV*** has no dependency on any other instruction

C-S²QED finds all **Single-Instruction Bugs**



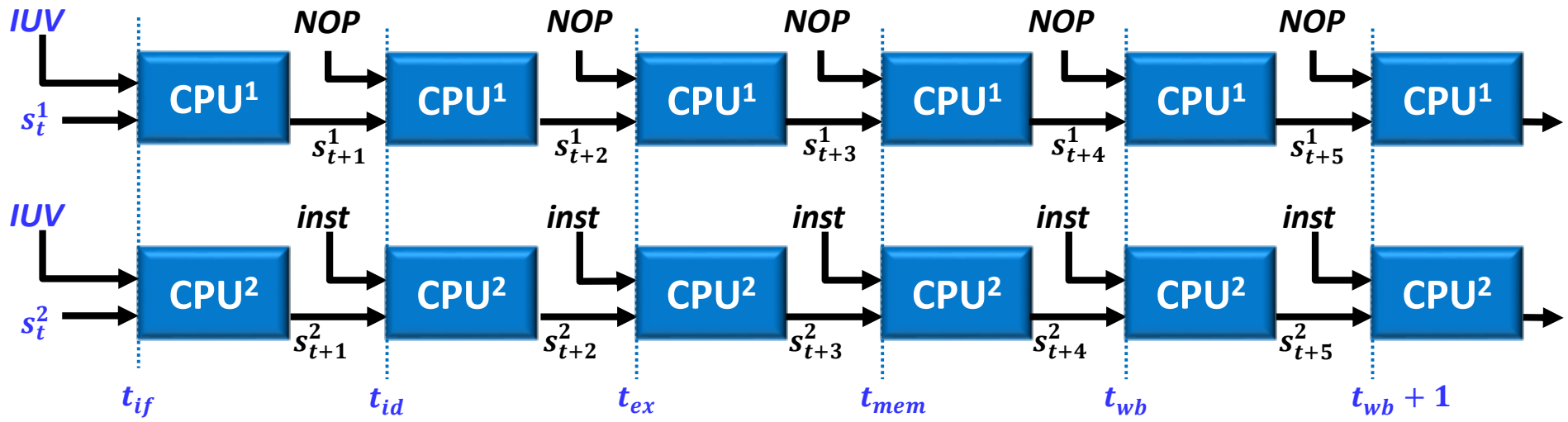

```
assume:
    at t_if: cpu1_fetched_instr() = cpu2_fetched_instr();
    at t_if: cpu1_state = flushed_pipeline();
    at t_id: instr_register_type();
prove:
    at t_wb + 1:cpu1_reg(wr_addr@t_id)=exp_value(funct@t_id,src1_addr@t_id,src2_addr@t_id);
```

C-S²QED finds all **Single-Instruction Bugs**



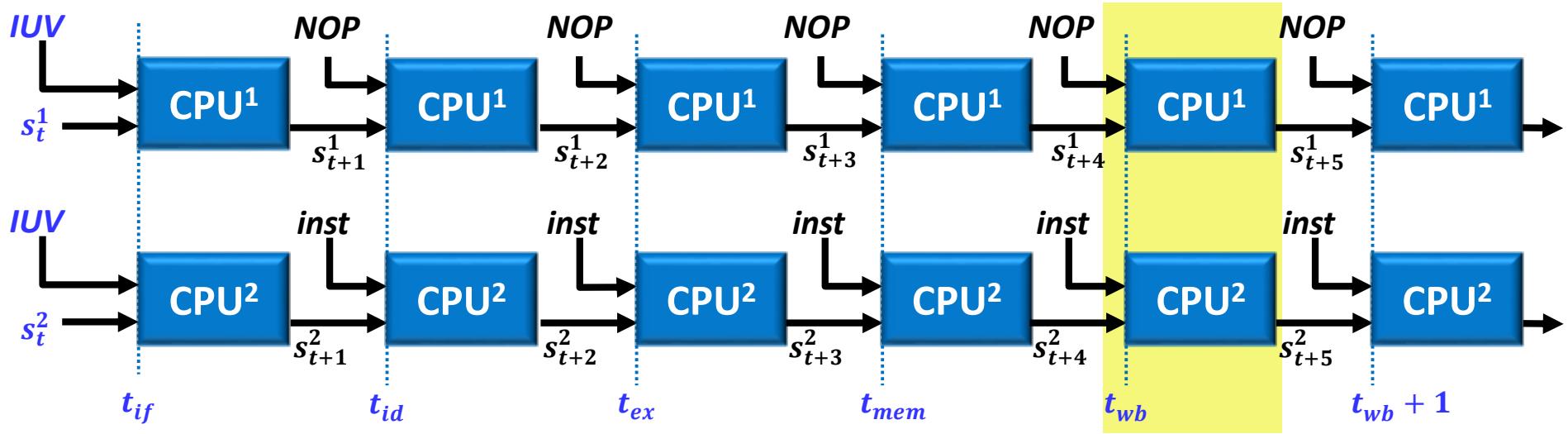
- **The macros can be automatically generated from an executable ISA model**

C-S²QED finds all **Multiple-Instruction Bugs**



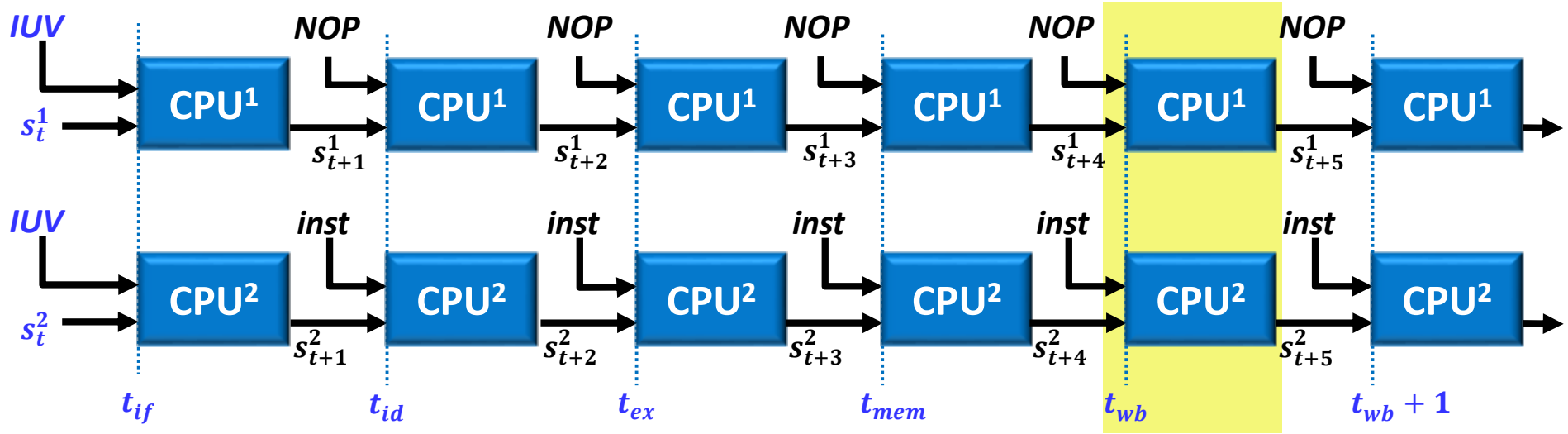
- For detecting "**multiple-instruction bugs**", create a scenario such that
  - ***CPU²*** instance gets the same operand values as ***CPU¹***
  - not necessarily from the register file (e.g., from forwarding unit)

C-S²QED finds all **Multiple-Instruction Bugs**



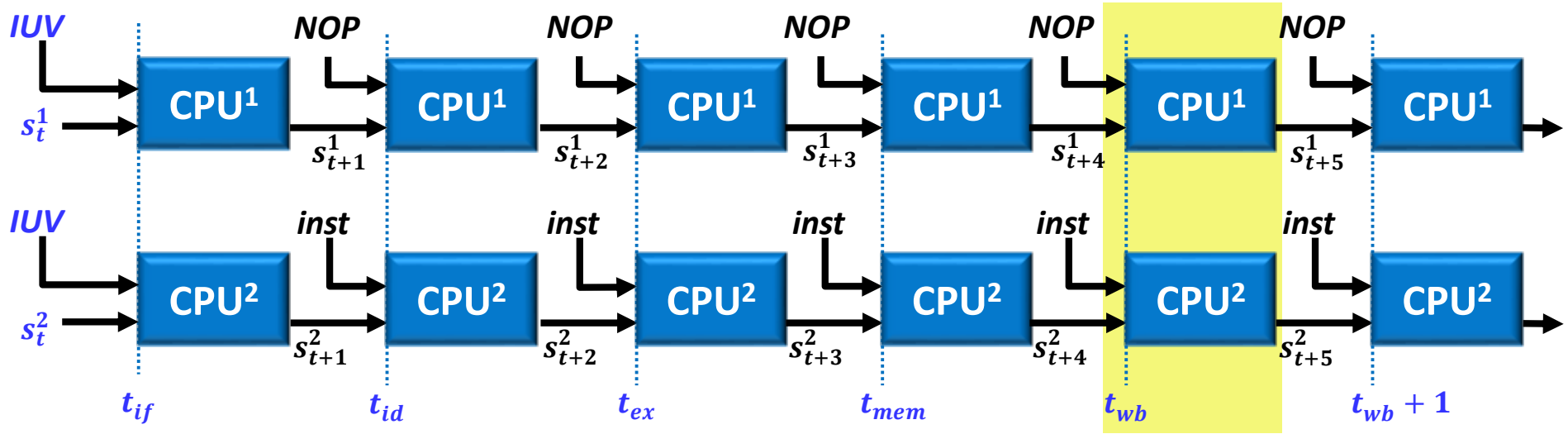
- The results of the ***IUV*** are visible in the register file at $t_{wb}+1$
- The results of the **instruction preceding the *IUV*** are visible at t_{wb}

C-S²QED finds all **Multiple-Instruction Bugs**



- The results of the ***IUV*** are visible in the register file at $t_{wb}+1$
- The results of the **instruction preceding the *IUV*** are visible at t_{wb}
- Assume that ***CPU¹*** and ***CPU²*** instances are ***consistent*** *at* t_{wb}

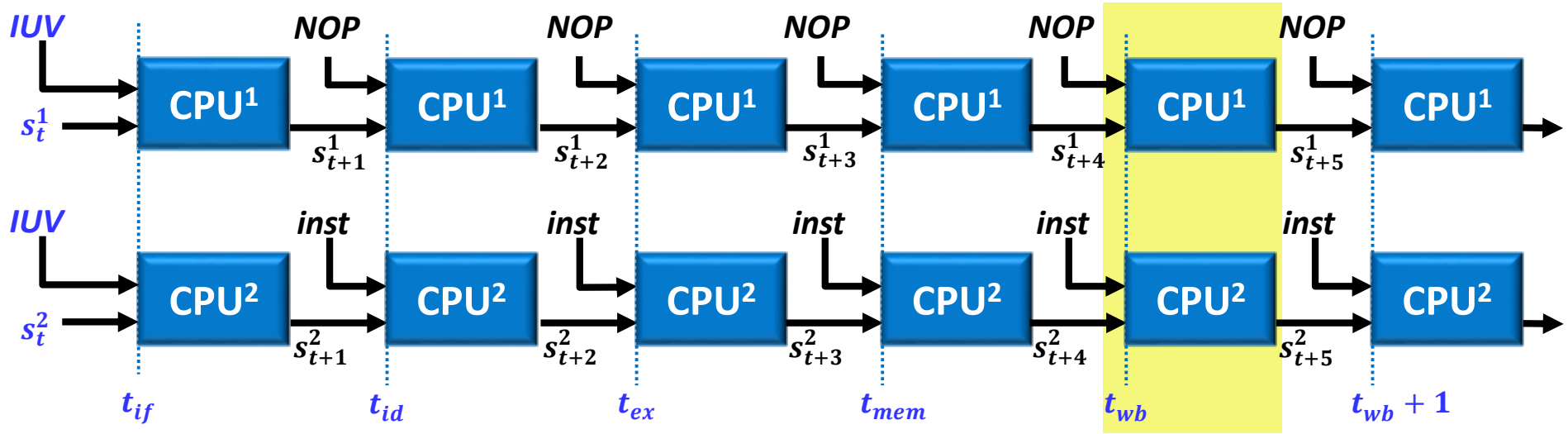
C-S²QED finds all **Multiple-Instruction Bugs**



- The register files R_{CPU^1} and R_{CPU^2} are consistent with each other, if

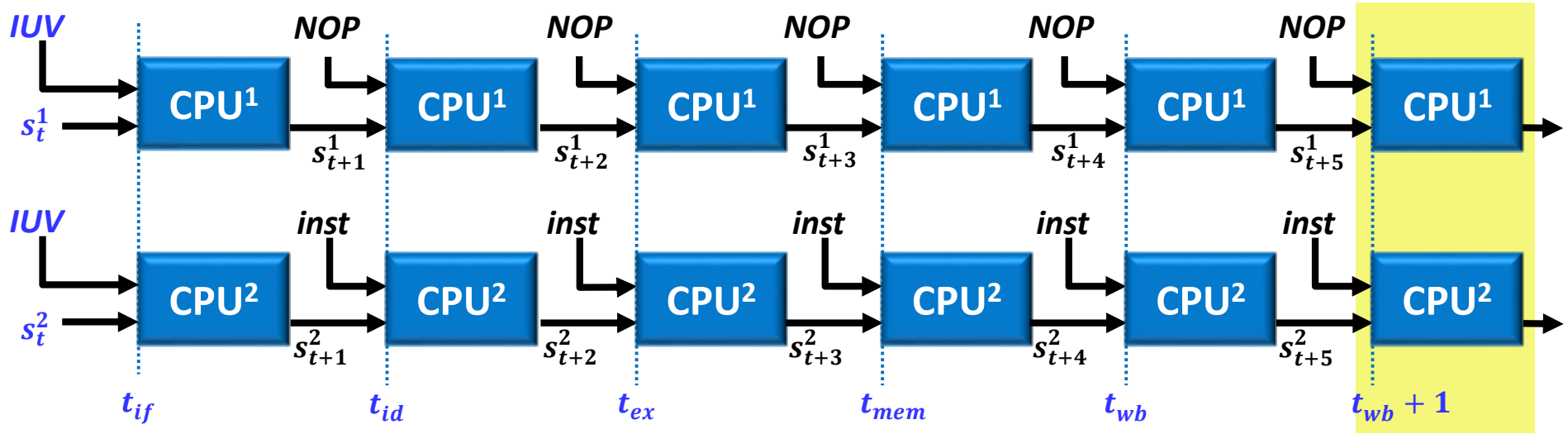
$$\textit{consistent\_registers()} := \bigwedge_{i=0}^{N-1} \left(R^i_{CPU^1} = R^i_{CPU^2}\right)$$

C-S²QED finds all **Multiple-Instruction Bugs**



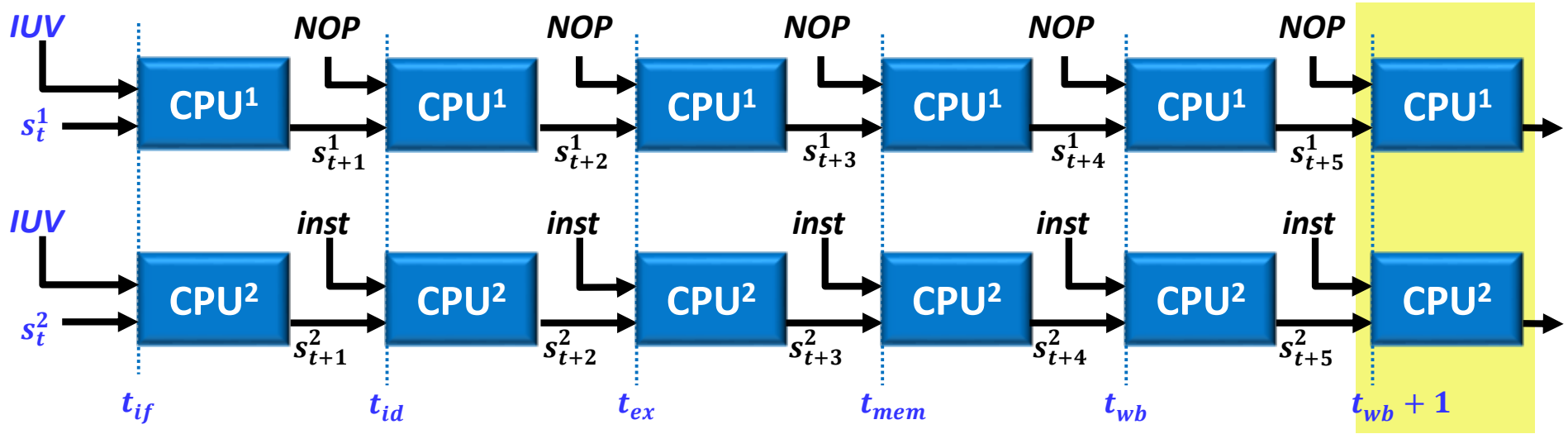

```
assume:
        at t_if: cpu1_fetched_instr() = cpu2_fetched_instr();
        at t_if: cpu1_state = flushed_pipeline();
        at t_id: instr_register_type();
        at t_wb: consistent_registers();
```

C-S²QED finds all **Multiple-Instruction Bugs**



- At $t_{wb}+1$, results of ***IUV*** are visible in register file
- In a bug-free processor, ***IUV*** execution must lead to
  - both instances having consistent registers

C-S²QED finds all **Multiple-Instruction Bugs**




```
assume:
        at t_if: cpu1_fetched_instr() = cpu2_fetched_instr();
        at t_if: cpu1_state = flushed_pipeline();
        at t_id: instr_register_type();
        at t_wb: consistent_registers();
prove:
        at t_wb + 1: cpu1_reg(wr_addr@t_id)=exp_value(funct@t_id,src1_addr@t_id,src2_addr@t_id);
        at t_wb + 1: consistent_registers();
```

Experimental Results

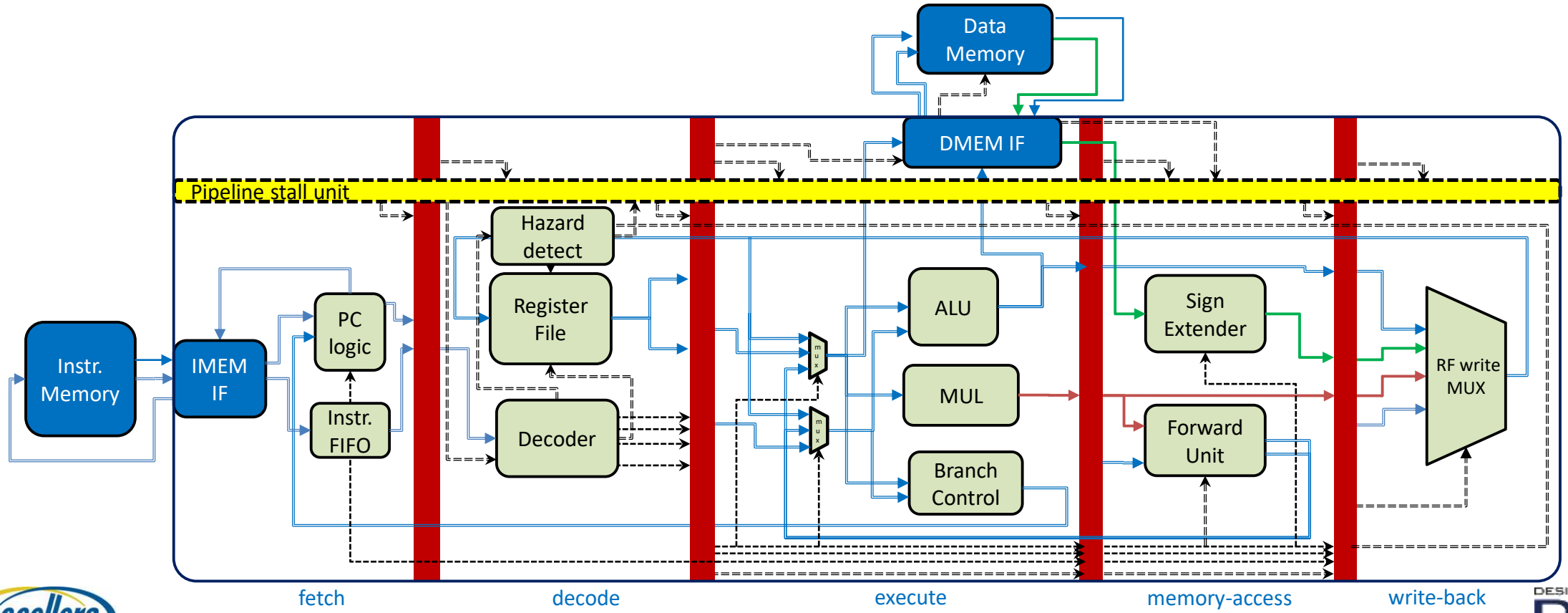
Effectiveness and feasibility of C-S²QED is shown by verifying:

- RISC-V processor, 5-stage pipeline, *RV32ICMXZicsr*

Experimental Results

Effectiveness and feasibility of C-S²QED is shown by verifying:

- RISC-V processor, 5-stage pipeline, *RV32ICMXZicsr*



Experimental Results

	C-IPC
Detect single-instruction bug	Yes
Detect multiple-instruction bug	Yes
Effort for base instruction set	16 person days
Effort for new extension	5 person days
Runtime with bugs	<30 s
Counterexample length [min, max] instructions	[1, 5]

- The core was initially verified with traditional property checking approach

Experimental Results

	C-IPC	C-S²QED
Detect single-instruction bug	Yes	Yes
Detect multiple-instruction bug	Yes	Yes
Effort for base instruction set	16 person days	5 person days
Effort for new extension	5 person days	1 person days
Runtime with bugs	<30 s	<30 s
Counterexample length [min, max] instructions	[1, 5]	[1, 5]

Experimental Results

	C-IPC	C-S²QED
Detect single-instruction bug	Yes	Yes
Detect multiple-instruction bug	Yes	Yes
Effort for base instruction set	16 person days	5 person days
Effort for new extension	5 person days	1 person days
Runtime with bugs	<30 s	<30 s
Counterexample length [min, max] instructions	[1, 5]	[1, 5]

- C-S²QED detected 3 new performance bugs that were not detected previously
- Performance bug is a type of multiple-instruction bug, which causes unnecessary stalling of the pipeline

accellera SYSTEMS INITIATIVE

2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

DEMO

C-S²QED

- Highly automated, gap-free processor verification technique
- Detects all functional processor bugs
- Verification Engg. is freed from developing complex properties
  - C-S²QED requires only little expertise in formal verification
  - C-S²QED incurs low manual effort

Thank You!
Questions?