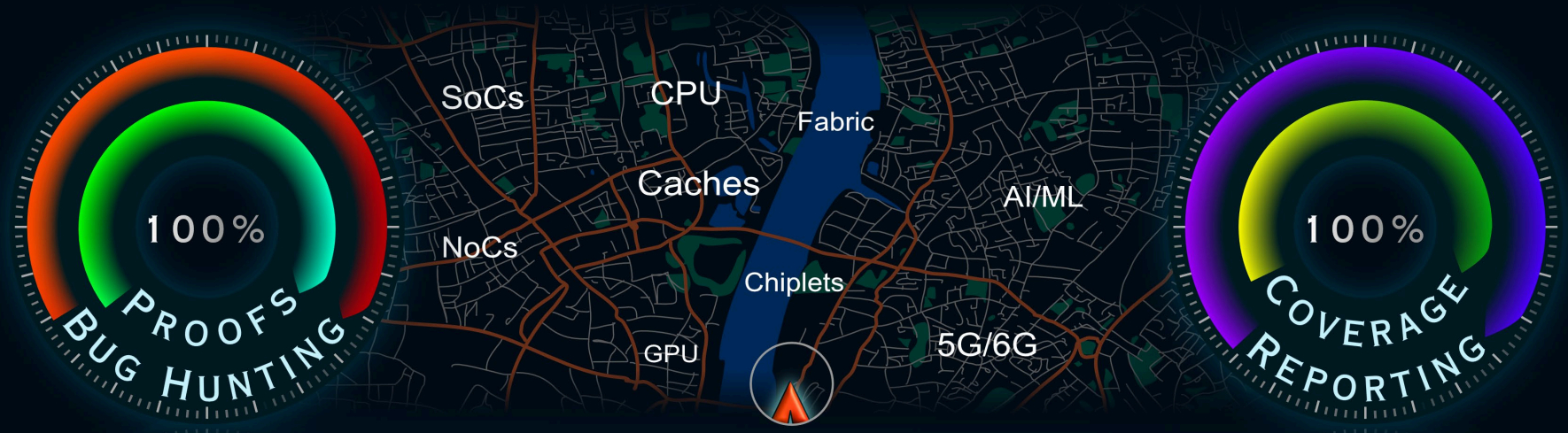
Power. Performance. Proofs
Scaling Formal for the AI-Driven Compute Revolution

Ashish Darbari, DVCon India 2025

axiomise
predictable formal verification

axiomise®

predictable formal verification



Where to use formal

SELECT MODE

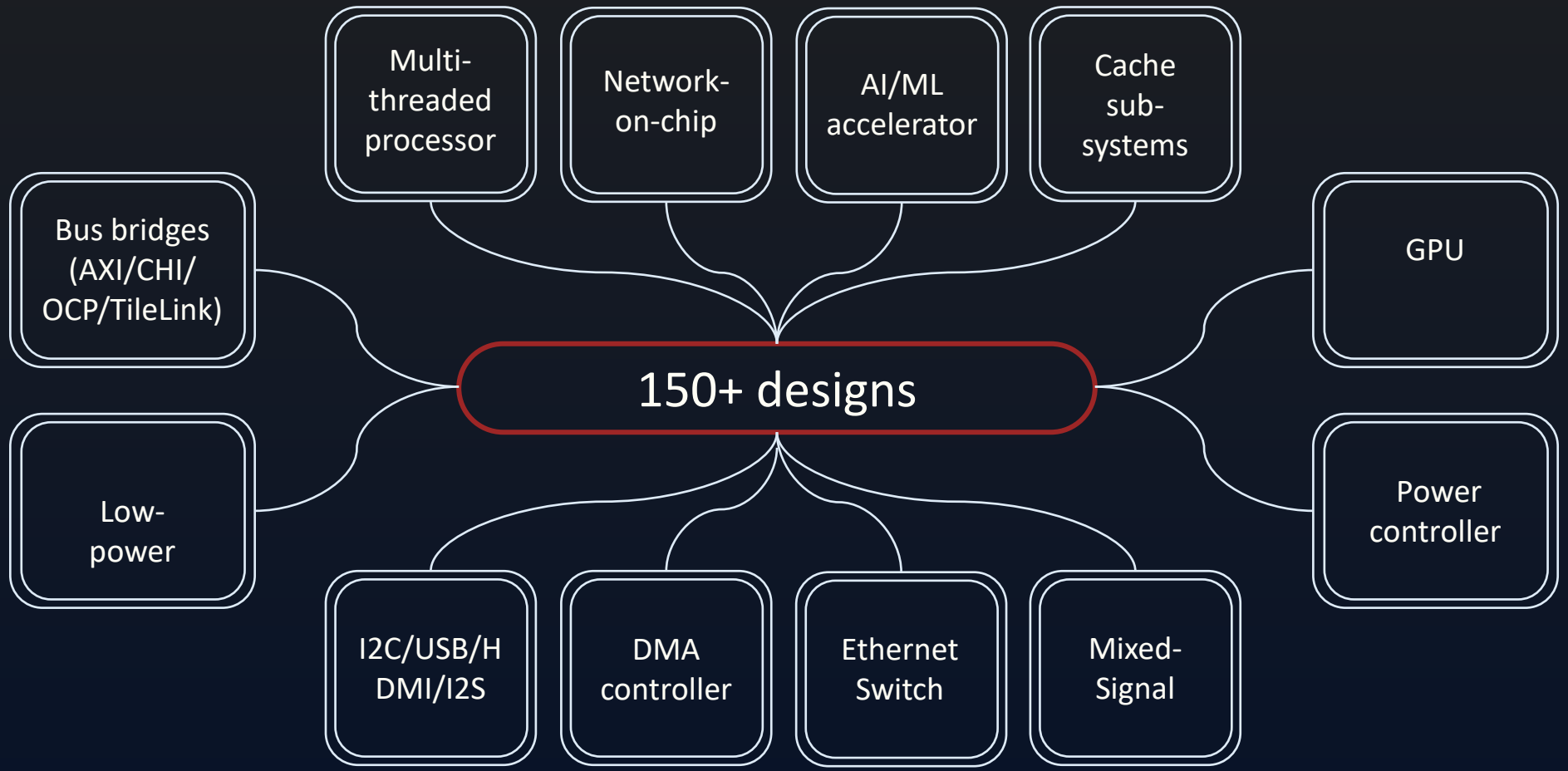
CONSULTING SERVICES TRAINING APPS

Accelerating sign-off using exhaustive formal methods

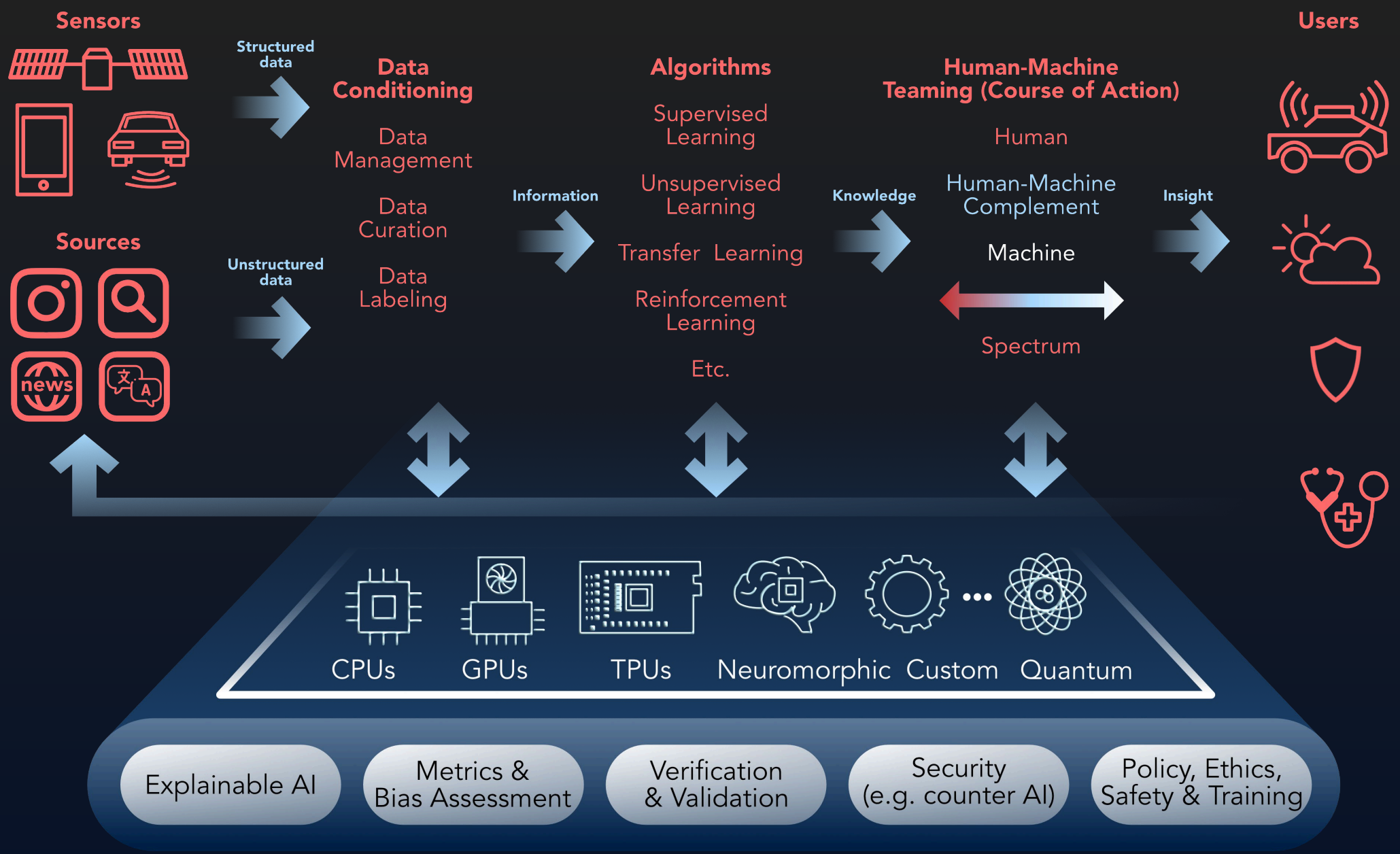
Making Formal Normal

Axiomise Experience Across Different Domains

Scaling formal for big designs – enabling end-to-end sign-off



axiomise
predictable formal verification



axiomise
predictable formal verification

Survey of Machine Learning Accelerators, Reuther et al., MIT Lincoln Labs, 2020

Accelerating Compute for AI/ML

Hetrogeneous compute

GPU/Parallel Processing

Caches & Cache Coherency

NoC Fabric

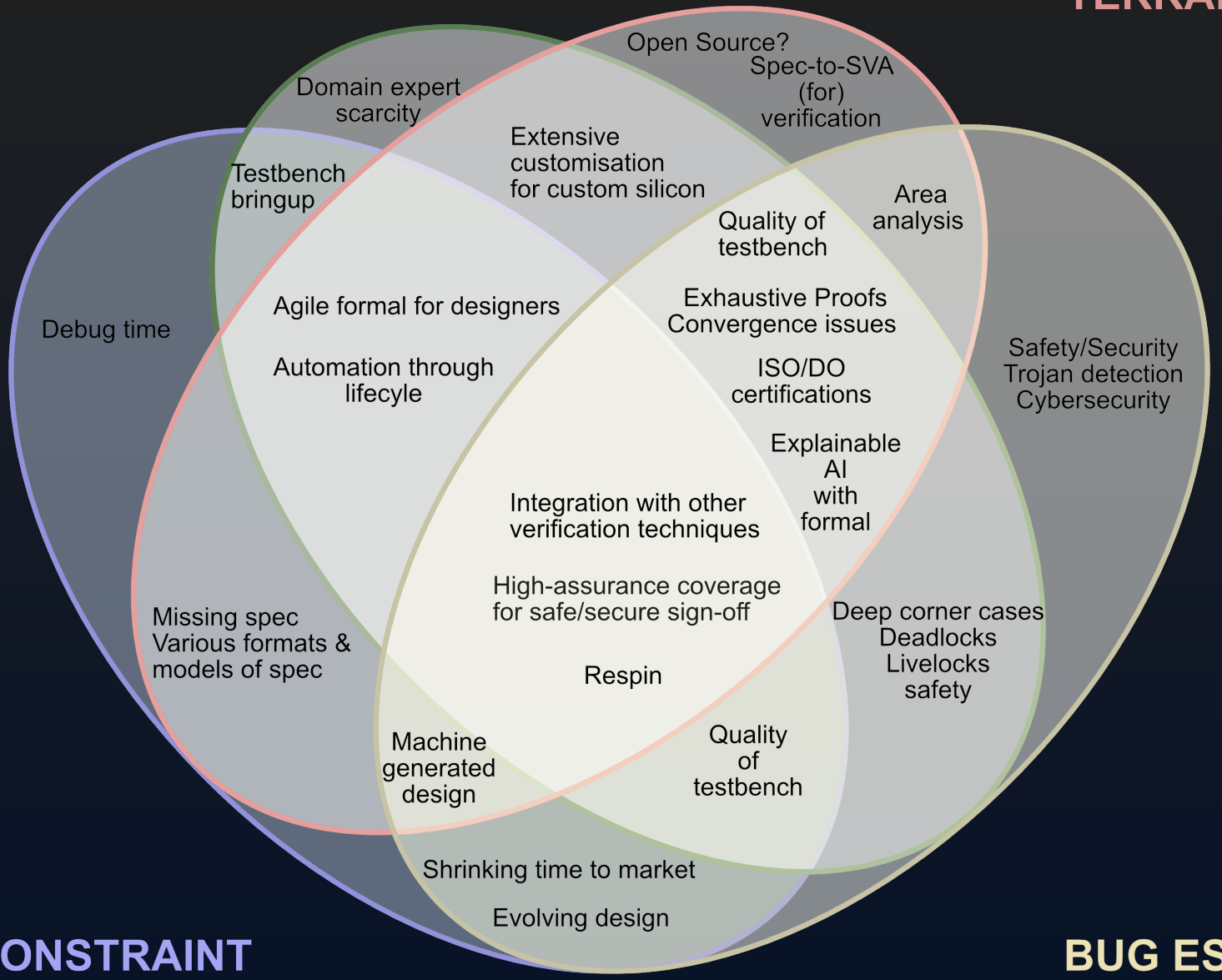
Datapath

Formal Verification

Scaling Up Challenges

LACK OF KNOWLEDGE

TERRAIN DIFFICULTY



TIME CONSTRAINT

BUG ESCAPE

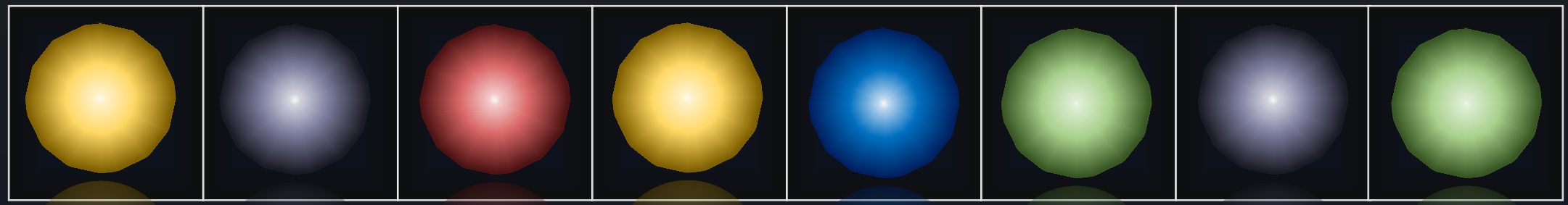
axiomise
predictable formal verification

Abstractions and Coverage

End-to-end sign-off with formal requires going beyond the bound

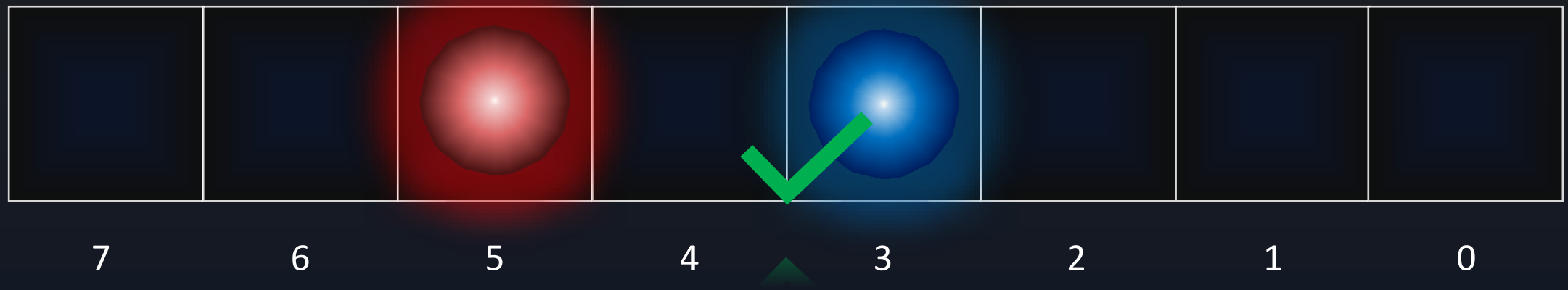
Two Transaction Abstraction

Ashish Darbari, CDNLive, 2014



Two Transaction Abstraction

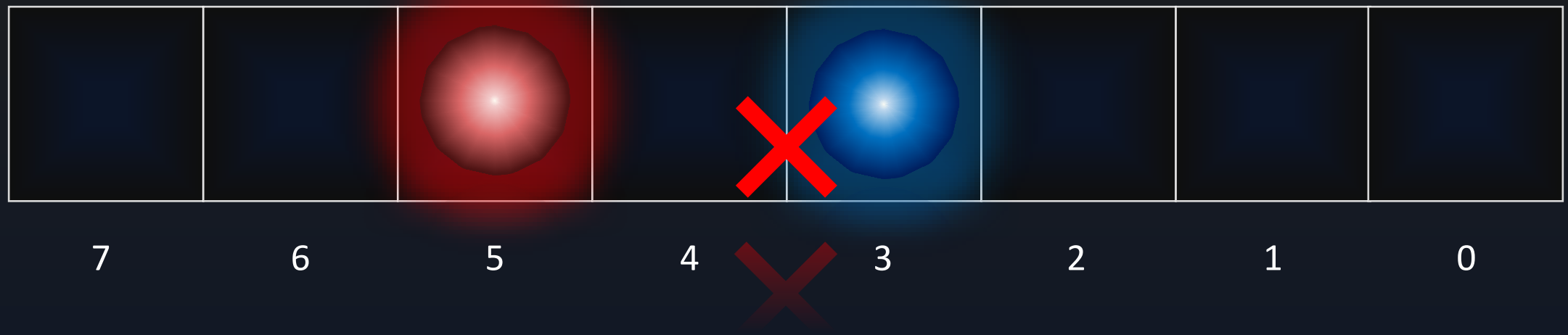
Ashish Darbari, CDNLive, 2014



axiomise
predictable formal verification

Two Transaction Abstraction

Ashish Darbari, CDNLive, 2014



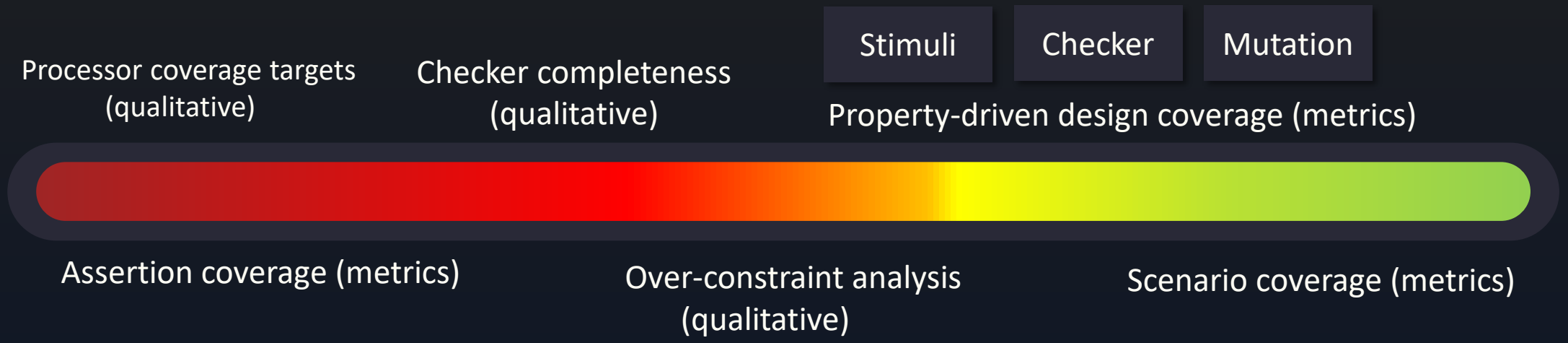
axiomise
predictable formal verification

Library of abstraction models invented at Axiomise

Efficient Complete Rigorous Scalable Reusable

Six-Dimensional Coverage for Formal

Coverage driven end-to-end vendor-neutral formal verification



We built the industry's first and only end-to-end vendor-neutral formal verification solution

Our solution works with multiple tools

You can verify beyond doubt

axiomise®
predictable formal verification

Coverage Goals

Are we done yet?

Do we have a list of all necessary assertions and covers?

Coverage targets

axiomise
predictable formal verification

Coverage Goals

Are we done yet?

Do we have a list of all necessary assertions and covers?

- Coverage targets

Do we have proofs that establish absence of bugs?

- Assertion coverage
- Design mutation coverage

axiomise
predictable formal verification

Coverage Goals

Are we done yet?

Do we have a list of all necessary assertions and covers?

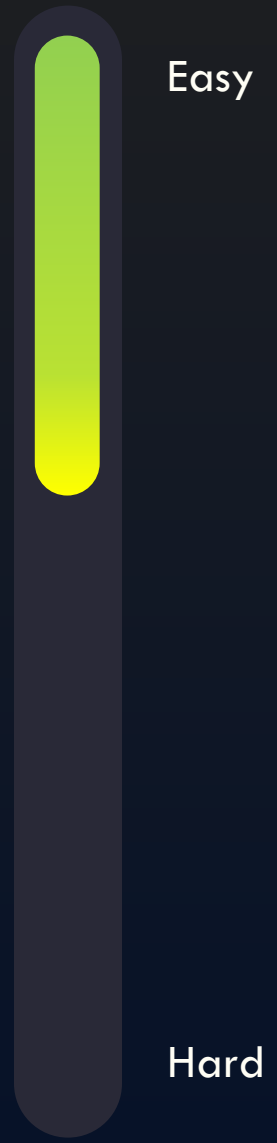
- Coverage targets

Do we have proofs that establish absence of bugs?

- Assertion coverage
- Design mutation coverage

What percentage of design code is covered by our properties?

- Property-driven design coverage



axiomise
predictable formal verification

Coverage Goals

Are we done yet?

Do we have a list of all necessary assertions and covers?

- Coverage targets

Do we have proofs that establish absence of bugs?

- Assertion coverage
- Design mutation coverage

What percentage of design code is covered by our properties?

- Property-driven design coverage

Do we have checker completeness?

- For any given check, is it checking what it needs to check in its entirety?
- Design mutation coverage

axiomise
predictable formal verification

Coverage Goals

Are we done yet?

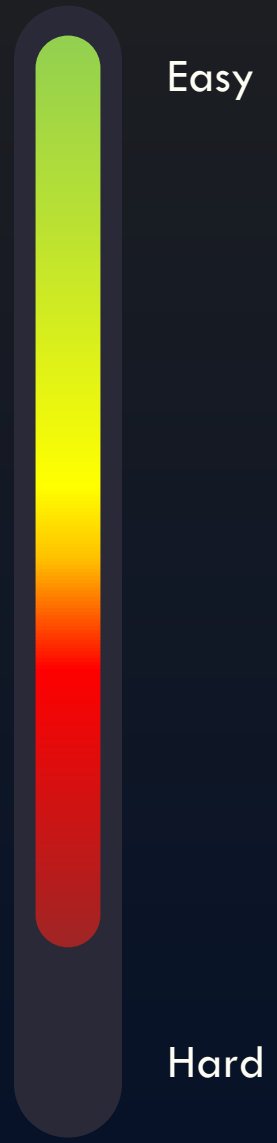
Do we have a list of all necessary assertions and covers?
- Coverage targets

Do we have proofs that establish absence of bugs?
- Assertion coverage
- Design mutation coverage

What percentage of design code is covered by our properties?
- Property-driven design coverage

Do we have checker completeness?
- For any given check, is it checking what it needs to check in its entirety?
- Design mutation coverage

Are constraints blocking relevant stimuli?
- Is my check blocked from doing the right thing?
- Design mutation + test-bench mutation coverage



axiomise
predictable formal verification

Coverage Goals

Are we done yet?

Do we have a list of all necessary assertions and covers?
- Coverage targets

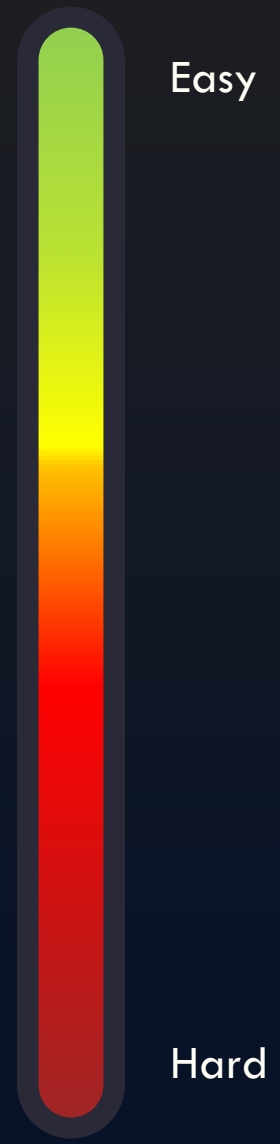
Do we have proofs that establish absence of bugs?
- Assertion coverage
- Design mutation coverage

What percentage of design code is covered by our properties?
- Property-driven design coverage

Do we have checker completeness?
- For any given check, is it checking what it needs to check in its entirety?
- Design mutation coverage

Are constraints blocking relevant stimuli?
- Is my check blocked from doing the right thing?
- Design mutation + test-bench mutation coverage

Are all interesting functional scenarios covered?
- Scenario coverage examines the interaction of legal and illegal behaviours



axiomise
predictable formal verification

What is Scenario Coverage?

Proving that specific instruction sequencing works and visualizing it

Conventional verification of designs relied heavily on architectural testing

Do interleaved instruction sequences always work as expected?

```
SUB   x7   x8   x9
ADD   x7   x10  x13
```

For even a small core, we are talking about 32 registers, say 60+ user instructions

How about verifying that certain optimizations do not break the functionality?

```
OR     x1   x25   x14
XOR   x11    x1    x0
AND   x13   x11   x15
```

DATA FORWARDING

Looks similar to simulation-based functional coverage
But goes well beyond simulation-based coverage

Specifying the Intent

Exploit and visualize concurrent and non-deterministic behaviours

WILL OR “ALWAYS” WORK CORRECTLY?

CAN OR WORK “AT ALL” CORRECTLY?

ISA COVERAGE ANALYZER®/ VISUALIZE ISA

	PREFIX			OBS		SUFFIX		
S.No.	PREFIX DELAY	INSTR NAME	FREQ	OBS DELAY	INSTR NAME	SUFFIX DELAY	INSTR NAME	FREQ
1	4	SUB	2	1	OR	1	AND	0

4 cycles after reset has been detected SUB instruction was issued, with two repetitions

Followed by one cycle later, an OR instruction was issued once

Followed by one cycle later, an AND instruction was issued 0 times

axiomise®
predictable formal verification

Specifying the Intent

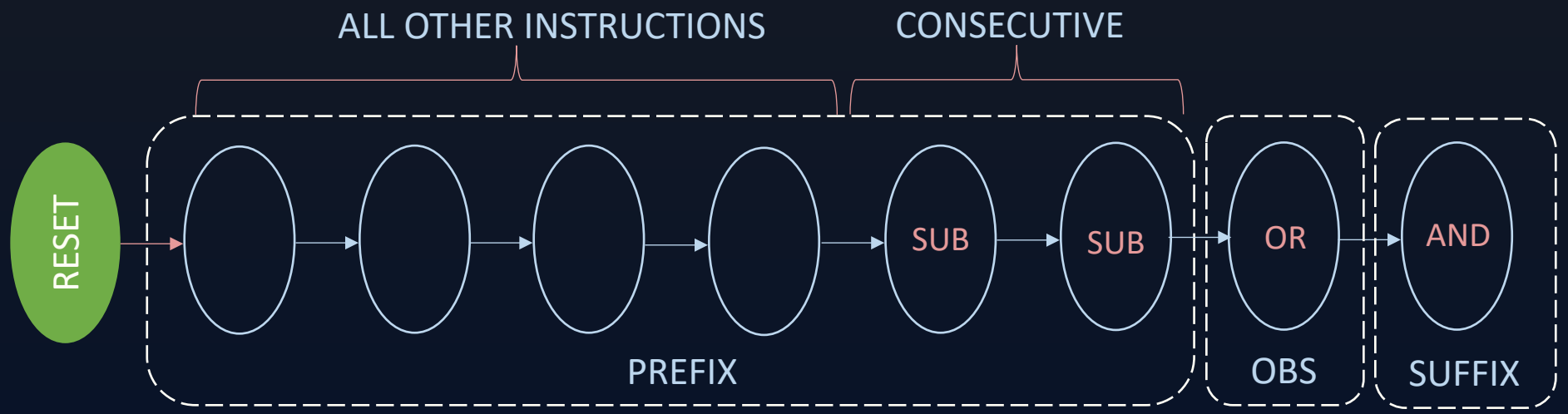
Exploit and visualize concurrent and non-deterministic behaviours

WILL OR “ALWAYS” WORK CORRECTLY?

CAN OR WORK “AT ALL” CORRECTLY?

ISA COVERAGE ANALYZER®/ VISUALIZE ISA

	PREFIX			OBS		SUFFIX		
S.No.	PREFIX DELAY	INSTR NAME	FREQ	OBS DELAY	INSTR NAME	SUFFIX DELAY	INSTR NAME	FREQ
1	4	SUB	2	1	OR	1	AND	0



axiomise
predictable formal verification

Specifying the Intent

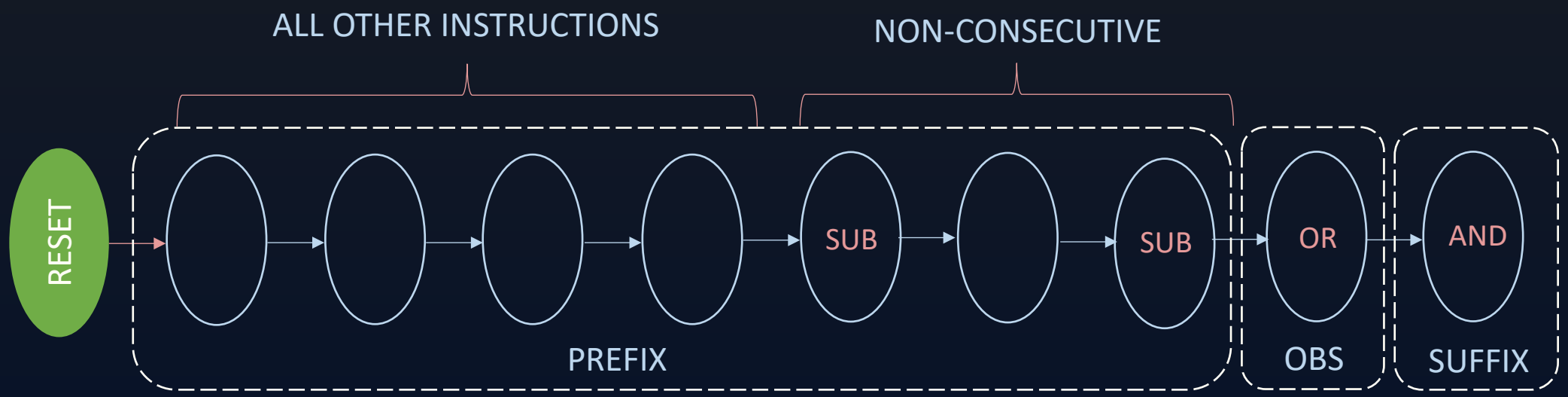
Exploit and visualize concurrent and non-deterministic behaviours

WILL OR “ALWAYS” WORK CORRECTLY?

CAN OR WORK “AT ALL” CORRECTLY?

ISA COVERAGE ANALYZER®/ VISUALIZE ISA

	PREFIX			OBS		SUFFIX		
S.No.	PREFIX DELAY	INSTR NAME	FREQ	OBS DELAY	INSTR NAME	SUFFIX DELAY	INSTR NAME	FREQ
1	4	SUB	2	1	OR	1	AND	0



axiomise
predictable formal verification

ISA Coverage Analyzer®

Data forwarding

ULTRA formalISA®
Making formal normal for RISC-V

We ask our coverage tool to show if data forwarding case is covered by our checks

```
SUB     V       A       B
 OR     W       V       C
AND     D       W       V
where {A, B, C, D, W, V} ∈{x0,...x31}
```

The question posed to the tool is to force a forwarding on registers shared between PREFIX and OBS and OBS and SUFFIX

No other constraints are imposed

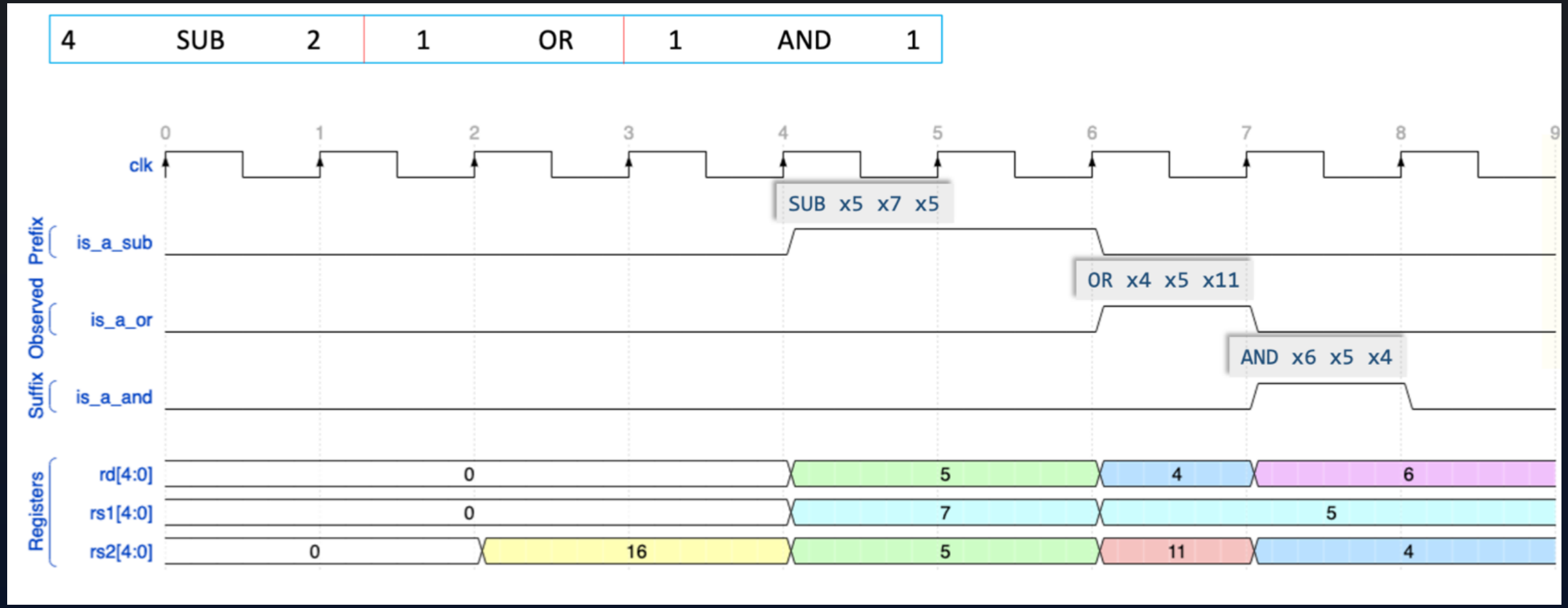
Therefore, all register combinations are checked and covered

axiomise®
predictable formal verification

Coverage Analyzer Example

Proving that specific instruction sequencing works and visualizing it

ULTRA formalISA®
Making formal normal for RISC-V



axiomise
predictable formal verification

How Many Scenarios to Analyze?

For a processor

ISA Coverage Analyzer

An example spec

ISA COVERAGE ANALYZER®/ VISUALIZE ISA																	
	PREFIX			OBS		SUFFIX											
S.No.	PREFIX DELAY	INSTR NAME	FREQ	OBS DELAY		SUFFIX DELAY	INSTR NAME	FREQ									
1	2	SB_2b01	1	1	OR	1	ADDI	1	45	2	ANDI	1	1	LW_2b01	2	AUIPC	1
2	2	SB_2b00	1	1	AND	1	LUI	1	46	2	ADD	1	1	LW_2b00	2	XORI	1
3	2	SB_2b10	1	1	ANDI	1	AUIPC	1	47	2	XORI	1	1	LW_2b10	2	SLL	1
4	2	SB_2b11	1	1	ADD	1	XORI	1	48	2	SUB	1	1	LW_2b11	2	SRA	1
5	3	SH_2b01	2	1	XORI	1	SLL	1	49	2	XOR	2	1	OR	1	AND	0
6	3	SH_2b00	2	1	SUB	1	SRA	1	50	3	ADD	1	1	ANDI	1	SUB	0
7	3	SH_2b10	2	1	OR	1	ADDI	1	51	4	SUB	2	1	OR	1	AND	0
8	3	SH_2b11	2	1	AND	1	LUI	1	52	1	AND	1	1	SUB	1	OR	0
9	3	SW_2b01	2	1	ANDI	1	AUIPC	1	53	5	XOR	2	1	ORI	1	AND	0
10	3	SW_2b00	2	1	ADD	1	XORI	1	54	6	OR	1	1	ANDI	1	SUB	0
11	3	SW_2b10	2	1	XORI	1	SLL	1	55	7	XOR	2	1	OR	1	SUB	0
12	3	SW_2b11	2	1	SUB	1	SRA	1	56	2	ADD	1	1	AND	1	SUB	0
13	2	LB_2b01	1	1	OR	1	ADDI	1	57	5	SUB	2	1	ORI	1	ANDI	0
14	2	LB_2b00	1	1	AND	1	LUI	1	58	8	ANDI	1	1	ADD	1	SUB	0
15	2	LB_2b10	1	1	ANDI	1	AUIPC	1	59	20	XOR	2	1	OR	1	AND	0
16	2	LB_2b11	1	1	ADD	1	XORI	1	60	4	XOR	2	1	OR	1	AND	0
17	2	LH_2b01	1	1	XORI	1	SLL	1	61	5	ADD	1	1	ANDI	1	XORI	0
18	2	LH_2b00	1	1	SUB	1	SRA	1	62	8	SUB	2	1	OR	1	AND	0
19	2	LH_2b10	1	1	OR	1	ADDI	1	63	9	AND	1	1	SUB	1	OR	0
20	2	LH_2b11	1	1	AND	1	LUI	1	64	2	XOR	2	1	ORI	1	AND	0
21	2	LW_2b01	1	1	ANDI	1	AUIPC	1	65	3	OR	1	1	ANDI	1	SUB	0
22	2	LW_2b00	1	1	ADD	1	XORI	1	66	2	XOR	2	1	OR	1	SUB	0
23	2	LW_2b10	1	1	XORI	1	SLL	1	67	9	ADD	1	1	AND	1	SUB	0
24	2	LW_2b11	1	1	SUB	1	SRA	1	68	3	SUB	2	1	ORI	1	ANDI	0
25	2	OR	1	1	SB_2b01	2	ADDI	1	69	8	ANDI	1	1	ADD	1	SLTS_0	0
26	2	AND	1	1	SB_2b00	2	LUI	1	70	2	ADD	2	1	XORI	1	SUB	0
27	2	ANDI	1	1	SB_2b10	2	AUIPC	1	71	3	AND	1	1	AUIPC	1	SUB	0
28	2	ADD	1	1	SB_2b11	2	XORI	1	72	4	ADD	2	1	SUB	1	AUIPC	0
29	2	XORI	1	1	SH_2b01	2	SLL	1	73	1	ADDI	1	1	LUI	1	BGES	0
30	2	SUB	1	1	SH_2b00	2	SRA	1	74	5	ORI	2	1	JAL	1	XORI	0
31	2	OR	1	1	SH_2b10	2	ADDI	1	75	6	XOR	1	1	JALR	1	ANDI	0
32	2	AND	1	1	SH_2b11	2	SUB	1	76	7	SUB	2	1	BEQ	1	AND	0
33	2	ANDI	1	1	SW_2b01	2	ADDI	1	77	2	OR	1	1	BNE	1	ORI	0
34	2	ADD	1	1	SW_2b00	2	XORI	1	78	5	ANDI	2	1	BLTS	1	XORI	0
35	2	XORI	1	1	SW_2b10	2	SLL	1	79	8	XOR	1	1	BLTU	1	SUB	0
36	2	SUB	1	1	SW_2b11	2	SLL	1	80	20	ADD	2	1	BGES	1	ADDI	0
37	2	OR	1	1	LB_2b01	2	ADDI	1	81	4	SUB	2	1	BGEU	1	XORI	0
38	2	AND	1	1	LB_2b00	2	LUI	1	82	3	AND	1	1	SLL	1	ORI	0
39	2	ANDI	1	1	LB_2b10	2	AUIPC	1	83	9	ADD	1	1	SRL	1	OR	0
40	2	ADD	1	1	LB_2b11	2	XORI	1	84	8	ADDI	1	1	SRA	1	ADD	0
41	2	XORI	1	1	LH_2b01	2	SLL	1									
42	2	SUB	1	1	LH_2b00	2	SRA	1									
43	2	OR	1	1	LH_2b10	2	ADDI	1									
44	2	AND	1	1	LH_2b11	2	LUI	1									

ISA Coverage Analyzer

Running with Cadence Jasper Platform

✔	Assert	cv32e40p_core.u_cov.Row_12__sw_2b11_sub_sra.NON_LOAD_STORE.as_PFXN_OBS	Tri (25)	Infinite	76.2	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_22__lw_2b00_add_xori.NON_LOAD_STORE.as_RST_PFXN_OBS	Tri (83)	Infinite	207.3	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_24__lw_2b11_sub_sra.NON_LOAD_STORE.as_RST_PFXN_OBS	Tri (83)	Infinite	172.4	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_24__lw_2b11_sub_sra.NON_LOAD_STORE.as_PFXN_OBS	Tri (61)	Infinite	66.3	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_58__andi_add_sub.NON_LOAD_STORE.as_RST_PFXN_OBS	Tri (89)	Infinite	313.0	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_58__andi_add_sub.NON_LOAD_STORE.as_PFXN_OBS	Tri (57)	Infinite	59.5	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_58__andi_add_sub.SUFFIX_PROPS.as_PFXN_OBS_SUFX_C	H (9)	Infinite	664.2	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_63__and_sub_or.NON_LOAD_STORE.as_PFXN_OBS	Tri (67)	Infinite	118.8	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_63__and_sub_or.SUFFIX_PROPS.as_PFXN_OBS_SUFX_C	Tri (89)	Infinite	157.3	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_69__andi_add_slts_0.NON_LOAD_STORE.as_RST_PFXN_OBS	Tri (89)	Infinite	302.2	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_69__andi_add_slts_0.NON_LOAD_STORE.as_PFXN_OBS	Tri (91)	Infinite	190.2	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_69__andi_add_slts_0.SUFFIX_PROPS.as_PFXN_OBS_SUFX_C	H (9)	Infinite	664.2	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_69__andi_add_slts_0.SUFFIX_PROPS.as_PFXN_OBS_SUFX_FWD_...	H (9)	Infinite	664.2	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_72__add_sub_auipc.SUFFIX_PROPS.as_PFXN_OBS_SUFX_FWD_...	Tri (61)	Infinite	62.8	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_73__addi_lui_bges.NON_LOAD_STORE.as_RST_PFXN_OBS	N (17)	Infinite	31.9	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_73__addi_lui_bges.NON_LOAD_STORE.as_PFXN_OBS	H (12)	Infinite	96.9	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_73__addi_lui_bges.SUFFIX_PROPS.as_PFXN_OBS_SUFX_C	N (19)	Infinite	36.3	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_73__addi_lui_bges.SUFFIX_PROPS.as_PFXN_OBS_SUFX_FWD_SY...	N (17)	Infinite	31.8	<embedded>		0	Analysis Ses...
✔	Assert	cv32e40p_core.u_cov.Row_1__sb_2b01_or_addi.NON_LOAD_STORE.as_RST_PFXN_OBS	M (20)	Infinite	77.0	<embedded>		0	Analysis Ses...

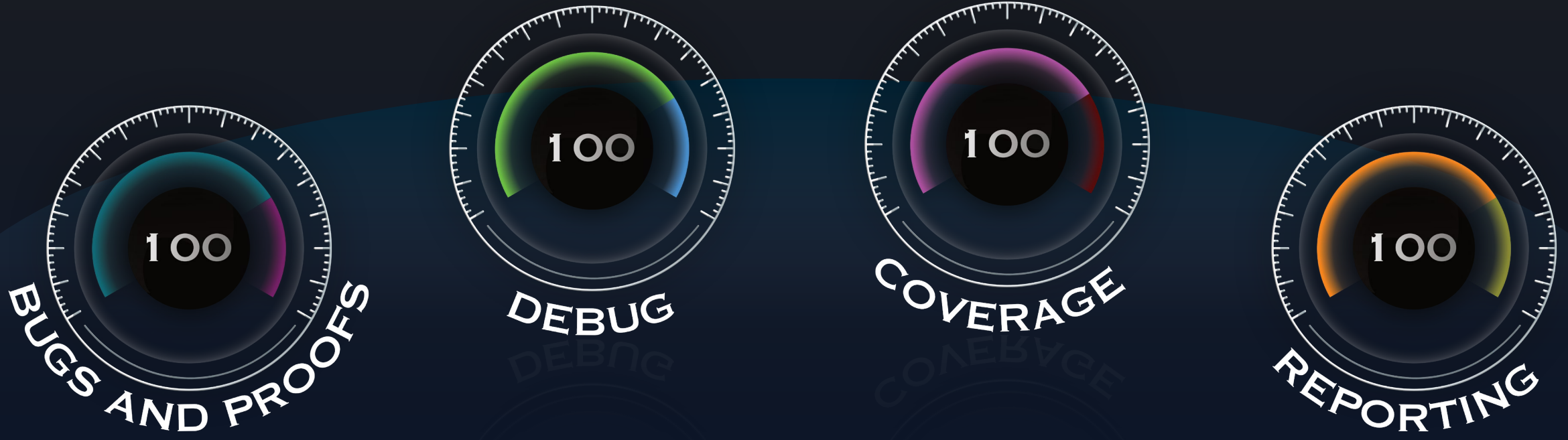
Total: 12964 | Filtered: 746 | Selected: 1 | Validity: 746:0:0:0 | Run: 0:0:0:746

Processors

Bounded proofs to exhaustive

formalISA®

Making formal normal for RISC-V



Accelerating debug and sign-off for custom designs using exhaustive formal

formalISA

Use any tool you like

✔		ibex_core.u_isa.axiomise__ISA_JALR
✔		ibex_core.u_isa.axiomise__ISA_LUI
✔		ibex_core.u_isa.axiomise__ISA_OR
✔		ibex_core.u_isa.axiomise__ISA_ORI
✔		ibex_core.u_isa.axiomise__ISA_SLL
✔		ibex_core.u_isa.axiomise__ISA_SLLI
✔		ibex_core.u_isa.axiomise__ISA_SLTSI_SET_TO_0
✔		ibex_core.u_isa.axiomise__ISA_SLTSI_SET_TO_1
✔		ibex_core.u_isa.axiomise__ISA_SLTS_SET_TO_0
✔		ibex_core.u_isa.axiomise__ISA_SLTS_SET_TO_1
✔		ibex_core.u_isa.axiomise__ISA_SLTUI_SET_TO_0
✔		ibex_core.u_isa.axiomise__ISA_SLTUI_SET_TO_1
✔		ibex_core.u_isa.axiomise__ISA_SLTU_SET_TO_0
✔		ibex_core.u_isa.axiomise__ISA_SLTU_SET_TO_1
✔		ibex_core.u_isa.axiomise__ISA_SRA
✔		ibex_core.u_isa.axiomise__ISA_SRAI
✔		ibex_core.u_isa.axiomise__ISA_SRL
✔		ibex_core.u_isa.axiomise__ISA_SRLI
✔		ibex_core.u_isa.axiomise__ISA_SUB
✔		ibex_core.u_isa.axiomise__ISA_XOR
✔		ibex_core.u_isa.axiomise__ISA_XORI

		u_isa.axiomise__ISA_SRLI
		u_isa.axiomise__ISA_ADDI
		u_isa.axiomise__ISA_SRL
		u_isa.axiomise__ISA_JALR
		u_isa.axiomise__ISA_XORI
		u_isa.axiomise__ISA_BEQ
		u_isa.axiomise__ISA_BNE
		u_isa.axiomise__ISA_SLTSI_SET_TO_1
		u_isa.axiomise__ISA_BGEU
		u_isa.axiomise__ISA_SLTSI_SET_TO_0
		u_isa.axiomise__ISA_XOR
		u_isa.axiomise__ISA_SLTUI_SET_TO_1
		u_isa.axiomise__ISA_OR
		u_isa.axiomise__ISA_ORI
		u_isa.axiomise__ISA_SLTUI_SET_TO_0
		u_isa.axiomise__ISA_ANDI
		u_isa.axiomise__ISA_SLLI
		u_isa.axiomise__ISA_SLTS_SET_TO_1
		u_isa.axiomise__ISA_SLTU_SET_TO_1
		u_isa.axiomise__ISA_BLTS
		u_isa.axiomise__ISA_SRAI
		u_isa.axiomise__ISA_SLTS_SET_TO_0
		u_isa.axiomise__ISA_SLTU_SET_TO_0
		u_isa.axiomise__ISA_JAL
		u_isa.axiomise__ISA_LUI
		u_isa.axiomise__ISA_BGES
		u_isa.axiomise__ISA_SRA
		u_isa.axiomise__ISA_BLTU
		u_isa.axiomise__ISA_ADD
		u_isa.axiomise__ISA_AND
		u_isa.axiomise__ISA_SUB
		u_isa.axiomise__ISA_SLL
		u_isa.axiomise__ISA_JAL_ret_address

✔	Assert	ibex_core.u_isa.inv_block_jal[4].axiomise_inv_isa_jal
✔	Assert	ibex_core.u_isa.inv_block_jalr[0].axiomise_inv_isa_jalr
✔	Assert	ibex_core.u_isa.inv_block_auipc[0].axiomise_inv_isa_auipc
✔	Assert	ibex_core.u_isa.inv_block_auipc[4].axiomise_inv_isa_auipc
✔	Assert	ibex_core.u_isa.inv_block_auipc[8].axiomise_inv_isa_auipc
✔	Assert	ibex_core.u_isa.inv_block_auipc[12].axiomise_inv_isa_auipc
✔	Assert	ibex_core.u_isa.inv_block_auipc[20].axiomise_inv_isa_auipc
✔	Assert	ibex_core.u_isa.inv_block_auipc[24].axiomise_inv_isa_auipc
✔	Assert	ibex_core.u_isa.inv_block_auipc[28].axiomise_inv_isa_auipc
✔	Assert	ibex_core.u_isa.axiomise_ISA_JAL
✔	Assert	ibex_core.u_isa.axiomise_ISA_JAL_ret_address
✔	Assert	ibex_core.u_isa.axiomise_ISA_JALR
✔	Assert	ibex_core.u_isa.axiomise_ISA_BEQ
✔	Assert	ibex_core.u_isa.axiomise_ISA_BNE
✔	Assert	ibex_core.u_isa.axiomise_ISA_BGEU
✔	Assert	ibex_core.u_isa.axiomise_ISA_BLTU
✔	Assert	ibex_core.u_isa.axiomise_ISA_LUI
✔	Assert	ibex_core.u_isa.axiomise_ISA_ADDI
✔	Assert	ibex_core.u_isa.axiomise_ISA_XORI
✔	Assert	ibex_core.u_isa.axiomise_ISA_ORI
✔	Assert	ibex_core.u_isa.axiomise_ISA_ANDI
✔	Assert	ibex_core.u_isa.axiomise_ISA_BLTS
✔	Assert	ibex_core.u_isa.axiomise_ISA_BGES
✔	Assert	ibex_core.u_isa.axiomise_ISA_SLTSI_SET_TO_1
✔	Assert	ibex_core.u_isa.axiomise_ISA_SLTSI_SET_TO_0
✔	Assert	ibex_core.u_isa.axiomise_ISA_SLTUI_SET_TO_1
✔	Assert	ibex_core.u_isa.axiomise_ISA_SLTUI_SET_TO_0
✔	Assert	ibex_core.u_isa.axiomise_ISA_SLLI
✔	Assert	ibex_core.u_isa.axiomise_ISA_SRLI
✔	Assert	ibex_core.u_isa.axiomise_ISA_SRAI
✔	Assert	ibex_core.u_isa.axiomise_ISA_SLL
✔	Assert	ibex_core.u_isa.axiomise_ISA_SRL

SYNOPSYS®

Mentor® A Siemens Business

cadence

axiomise®
predictable formal verification

formalISA®

Making formal normal for RISC-V

	ibex	zeroriscy	cv32e40p	WARP-V			Cheriot-ibex
Pipeline stages	2-stage	2-stage	4-stage	6-stage	4-stage	2-stage	2
No. of issues	65	77	5	30	30	30	6
Previously verified	Yes	Yes	No	Yes	Yes	Yes	Yes
How was it previously verified?	Simulation	Simulation	Simulation & Formal	Formal	Formal	Formal	Simulation & Formal
Time taken to find issues	< 30 seconds	< 30 seconds	< 30 seconds	< 30 seconds	< 30 seconds	< 30 seconds	<1 min
Nature of analysis and issues	Microarchitectural Deadlocks and Architectural	Microarchitectural Deadlocks and Architectural	Architectural	Architectural	Architectural	Architectural	Corner-case bugs
When was the issue found?	2019	2019	2020	2021	2021	2021	2024

axiomise®

predictable formal verification

Proof Convergence

When you encounter speed humps!

All the checks necessary to verify the processor are available in our app.

When properties do not converge, what do we do?

Scenario Coverage to the rescue.

Proof Convergence

How far can the bound go ?

? ⚡	Assert	mkCPU.u_isa.BASE_RTYPE_as_ISA_ADD_abstract	H	10331 -	0	1354.6
✔	Cover (related)	mkCPU.u_isa.BASE_RTYPE_as_ISA_ADD_abstract:precondition1	H	14	1	0.7
✔	Cover	mkCPU.u_isa.BASE_RTYPE_co_ISA_ADD_abstract	H	2	1	0.2
✔	Cover	mkCPU.u_isa.BASE_RTYPE_co_ISA_ADD_abstract_JG	H	16	1	6.9
? ⚡	Assert	mkCPU.u_isa.BASE_RTYPE_as_ISA_ADD	H	22 -	0	88.5
☐	Cover (related)	mkCPU.u_isa.BASE_RTYPE_as_ISA_ADD:precondition1	?	1 -	0	0.0
☐	Cover	mkCPU.u_isa.BASE_RTYPE_co_ISA_ADD	?	1 -	0	0.0
☐	Cover	mkCPU.u_isa.BASE_RTYPE_co_ISA_ADD_JG	?	1 -	0	0.0

axiomise
predictable formal verification

Going from Failures to Proofs

Encountered bumps pretty quickly trying to find stalls

```
co_directed_test_same_rd_but_interim_2_cycles_rd_eq_0_fail:
  `cp (
      $rose (rst_n) ##7     (add_instr_trigger && axiomise_rd  == `h1   &&
                             axiomise_rs1 == 'h2 && axiomise_rs2 == `h3 &&
                             axiomise_regfile[axiomise_rs1] == 'h6      &&
                             axiomise_regfile[axiomise_rs2] == 'h7
                             )
                    ##1 (axiomise_rd == 'h0) ##1 (axiomise_rd == 'h0)
                    //--send another add few cycles later
                    ##1 (add_instr_trigger && axiomise_rd == `h1        &&
                             axiomise_rs1 == 'h5 && axiomise_rs2 ==`h6  &&
                             axiomise_regfile[axiomise_rs1] == 'h8      &&
                             axiomise_regfile[axiomise_rs2] == 'h8
                             ) ##1 1'b1 ##1 1'b1);
```

Same rd register for the two ADDs and in the interim two cycles rd is 0

What's Going on with the Dead Cycles?

When rd register in the interim cycles was not 0 and not the same for the two ADDS

```
co_directed_test_different_rd_if_interim_rd_non_zero_fail:
  `cp (
      $rose (rst_n) ##7     (add_instr_trigger && axiomise_rd == 'h1     &&
                            axiomise_rs1 == 'h2 && axiomise_rs2 == 'h3 &&
                            axiomise_regfile[axiomise_rs1] == 'h6      &&
                            axiomise_regfile[axiomise_rs2] == 'h7
                            )
                    ##1 (axiomise_rd != 'h0) ##1 (axiomise_rd != 'h0
                    //--send another add few cycles later
                    ##1 (add_instr_trigger && axiomise_rd == 'h9       &&
                            axiomise_rs1 == 'h5 && axiomise_rs2 == 'h6 &&
                            axiomise_regfile[axiomise_rs1] == 'h8      &&
                            axiomise_regfile[axiomise_rs2] == 'h8
                            ) ##1 1'b1 ##1 1'b1);
```

Different rd register for the two ADDs and in the interim two cycles rd is NOT 0

What's Going on with the Dead Cycles?

Different rd registers for the two ADDs and interim two cycles rd is 0

```
co_directed_test_different_rd_interim_2_cycles_rd_eq_0_fail:
  `cp (
      $rose (rst_n) ##7     (add_instr_trigger && axiomise_rd  == 'h1    &&
                            axiomise_rs1 == 'h2 && axiomise_rs2 == 'h3   &&
                            axiomise_regfile[axiomise_rs1] == 'h6        &&
                            axiomise_regfile[axiomise_rs2] == 'h7
                            )
                    ##1 (axiomise_rd=='h0) ##1 (axiomise_rd=='h0)
                    //--send another add few cycles later
                    ##1 (add_instr_trigger && axiomise_rd == 'h9        &&
                            axiomise_rs1 == 'h5 && axiomise_rs2 == 'h6 &&
                            axiomise_regfile[axiomise_rs1] == 'h8      &&
                            axiomise_regfile[axiomise_rs2] == 'h8
                            ) ##1 1'b1 ##1 1'b1);
```

Different rd register for the two ADDs and in the interim two cycles rd is 0

Failures Becoming Passes

Spacing of 3 cycles, and we get a pass!

```
co_directed_test_same_rd_pass:
  `cp (
      $rose (rst_n) ##7 (add_instr_trigger && axiomise_rd  == 'h1      &&
                        axiomise_rs1 == 'h2 && axiomise_rs2 =='h3.    &&
                        axiomise_regfile[axiomise_rs1]=='h6           &&
                        axiomise_regfile[axiomise_rs2]=='h7
                        )
                    ##1 (axiomise_rd=='h0) ##1 (axiomise_rd=='h0) ##1 (axiomise_rd=='h0)
                    //--send another add few cycles later
                    ##1 (add_instr_trigger && axiomise_rd == 'h1      &&
                          axiomise_rs1 == 'h5 && axiomise_rs2 == 'h6 &&
                          axiomise_regfile[axiomise_rs1] == 'h8       &&
                          axiomise_regfile[axiomise_rs2] == 'h8
                          ) ##1 1'b1 ##1 1'b1);
```

Same rd register for the two ADDs and in the interim 3 cycles rd is 0

Failures Becoming Passes

Spacing of 3 cycles, and we get a pass!

```
co_directed_test_different_rd_pass:
  `cp (
      $rose (rst_n) ##7     (add_instr_trigger && axiomise_rd  == `h1    &&
                            axiomise_rs1 == 'h2 && axiomise_rs2 == 'h3   &&
                            axiomise_regfile[axiomise_rs1] == 'h6        &&
                            axiomise_regfile[axiomise_rs2] == 'h7
                            )
                            ##1 (axiomise_rd=='h0) ##1 (axiomise_rd=='h0) ##1 (axiomise_rd=='h0)
                  //--send another add few cycles later
                  ##1 (add_instr_trigger && axiomise_rd == `h9          &&
                            axiomise_rs1 == 'h5 && axiomise_rs2 == 'h6   &&
                            axiomise_regfile[axiomise_rs1] == 'h8        &&
                            axiomise_regfile[axiomise_rs2] == 'h8
                            ) ##1 1'b1 ##1 1'b1);
```

Different rd register for the two ADDs and in the interim 3 cycles rd is 0

What's Going On?

Okay, we now get a pass

```
co_directed_test_different_rd_if_interim_rd_non_zero_for_3_pass:
  `cp (
      $rose (rst_n) ##7     (add_instr_trigger && axiomise_rd  == `h1    &&
                             axiomise_rs1 == 'h2 && axiomise_rs2 == 'h3  &&
                             axiomise_regfile[axiomise_rs1] == 'h6       &&
                             axiomise_regfile[axiomise_rs2] == 'h7
                             )
                    ##1 (axiomise_rd != 'h0) ##1 (axiomise_rd != 'h0) ##1 (axiomise_rd != 'h0)
                    //--send another add few cycles later
                    ##1 (add_instr_trigger && axiomise_rd == `h9         &&
                             axiomise_rs1 == 'h5 && axiomise_rs2 == 'h6  &&
                             axiomise_regfile[axiomise_rs1] == 'h8       &&
                             axiomise_regfile[axiomise_rs2] == 'h8
                             ) ##1 1'b1 ##1 1'b1);
```

So, it appears that if we sent two ADDs, too close to each other, the second one cannot complete in the expected timeframe.

Two ADDs: Different rd registers

Passing Covers

✔	Cover	mkCPU.u_isa.co_directed_test_same_rd_pass	Ht	28	1	0.8	<embedded>	
✔	Cover	mkCPU.u_isa.co_directed_test_different_rd_pass	Ht	28	1	0.8	<embedded>	
✔	Cover	mkCPU.u_isa.co_directed_test_different_rd_if_interim_rd_non_zero_for_3_pass	Ht	28	1	0.9	<embedded>	
✘	Cover	mkCPU.u_isa.co_directed_test_same_rd_but_interim_2_cycles_rd_eq_0_fail	Ht (1)	Infinite	0	0.4	<embedded>	
✘	Cover	mkCPU.u_isa.co_directed_test_different_rd_interim_2_cycles_rd_eq_0_fail	Ht (1)	Infinite	0	0.4	<embedded>	
✘	Cover	mkCPU.u_isa.co_directed_test_different_rd_if_interim_rd_non_zero_fail	Ht (1)	Infinite	0	0.4	<embedded>	

Going from Failures to Proofs

We thought we had a reasonable turf to move further but….

- Still, we can’t prove that two ADDs can prove under all conditions ☹
- What if we constrain all ADDs to be spaced out by 3 cycles?
- Does it only impact only ADDs or all RTYPE instructions or all for that matter?
- Spacer assumptions were inserted, and performance seemed to improve.
- Still, we can’t prove that two ADDs can prove under all conditions ☹

Towards Exhaustive Proofs

Digging deeper into stalls, and why all instructions cannot complete in a timely manner

- We suspect, that there surely is some dependency between operands.
- We play around a bit more and constrain
  - That no new instruction sent from decode to execute can have the rs1 and rs2 registers be the same as the rd register of previous instructions that were in flight.
- Force the pipeline to only have ADDs.
- With these two additional constraints, we see proofs!!!
  - For any two firings of the ADD
  - For any number of firings of the ADD for any operands
- We check this for other RTYPE instructions.
- However, when we allow any RTYPE instruction to flow in the pipe, proofs struggle.

DATA FORWARDING NOT IMPLEMENTED IN THE DESIGN YET

Towards Exhaustive Proofs: Why are the Proofs not Working?

Proofs not going through due to stalls arising out of a lack of data forwarding but is there anything else?

What else could be happening?

Shall we sign with bounded-proofs?

Is there even a known bound?

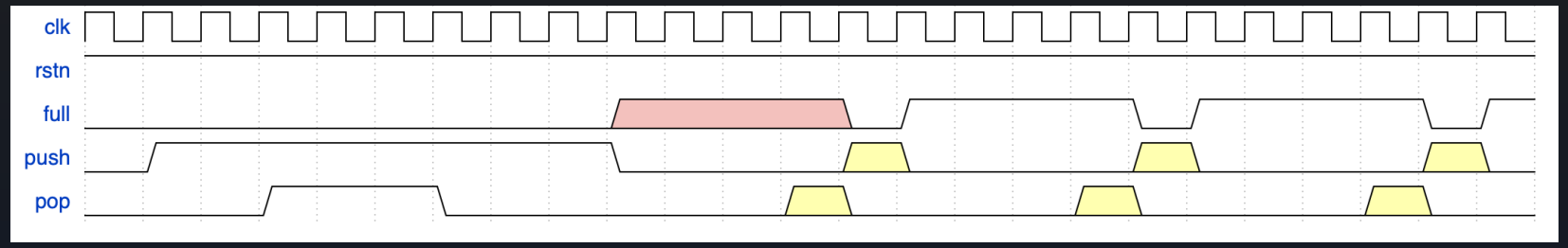
Proof Convergence

Why can’t new instructions be proven?

1. Why can't we see more instructions in the pipe?
2. Could this happen only when some FIFOs got full?
3. Why are we stalled?

Proof Convergence

What is going on?

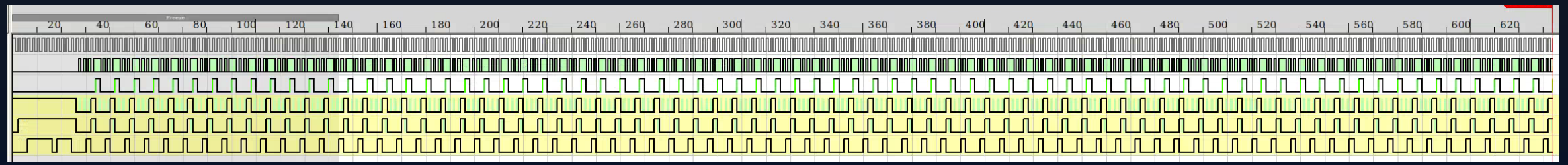


Push and pop can happen continuously until the FIFO becomes full for 3 cycles. After that we don't see continuous pops and pushes. The Decode pipe is stalled for 3 cycles every time the FIFO gets full.

We ask: If the FIFO remained full for 3 cycles, can we have a new push in the future?

```
fifo_stays_full_for_3_cycles: cover property (@(posedge CLK full[*3] ##[1:$] push)
```

Livelock for more than 500 cycles!



axiomise
predictable formal verification

What if the Processor Never Hit this Livelock?

What if we were lucky to never hit this livelock?

```
//-- Assume that the decode fifo can never remain full for 3 cycles

am_decode_fifo_never_full_for_3: `am (!full[*3]);
```

✔	Assert	mkCPU.u_isa.BASE_RTYPE_as_ISA_ADD_abstract	Ht (18)	Infinite	0	2.6
✔	Assert	mkCPU.u_isa.BASE_RTYPE_as_ISA_ADD	Hp (22)	Infinite	0	100.9

All instructions now prove exhaustively!

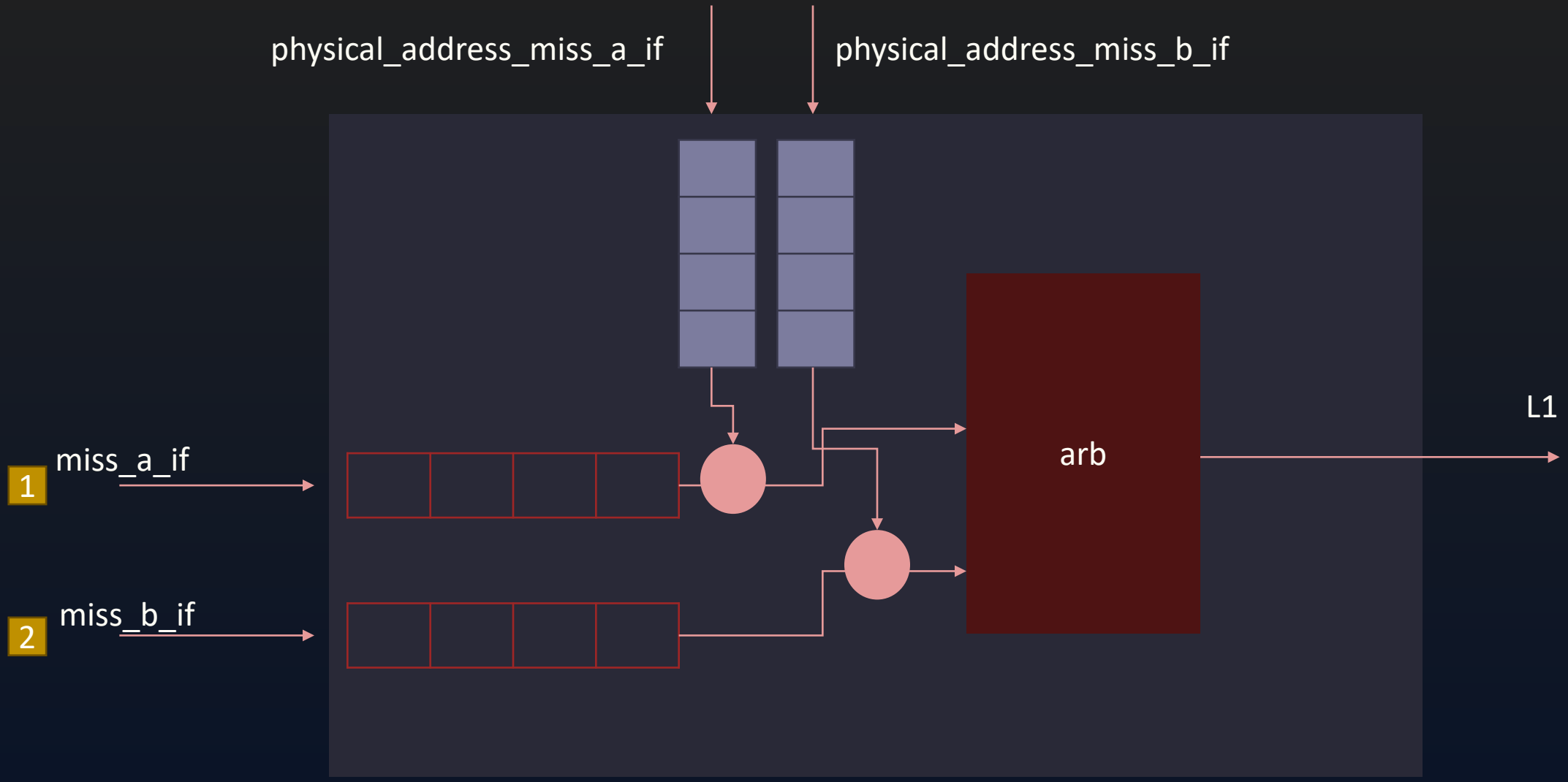
Machine Learning

Cache subsystems for multi-threaded processors

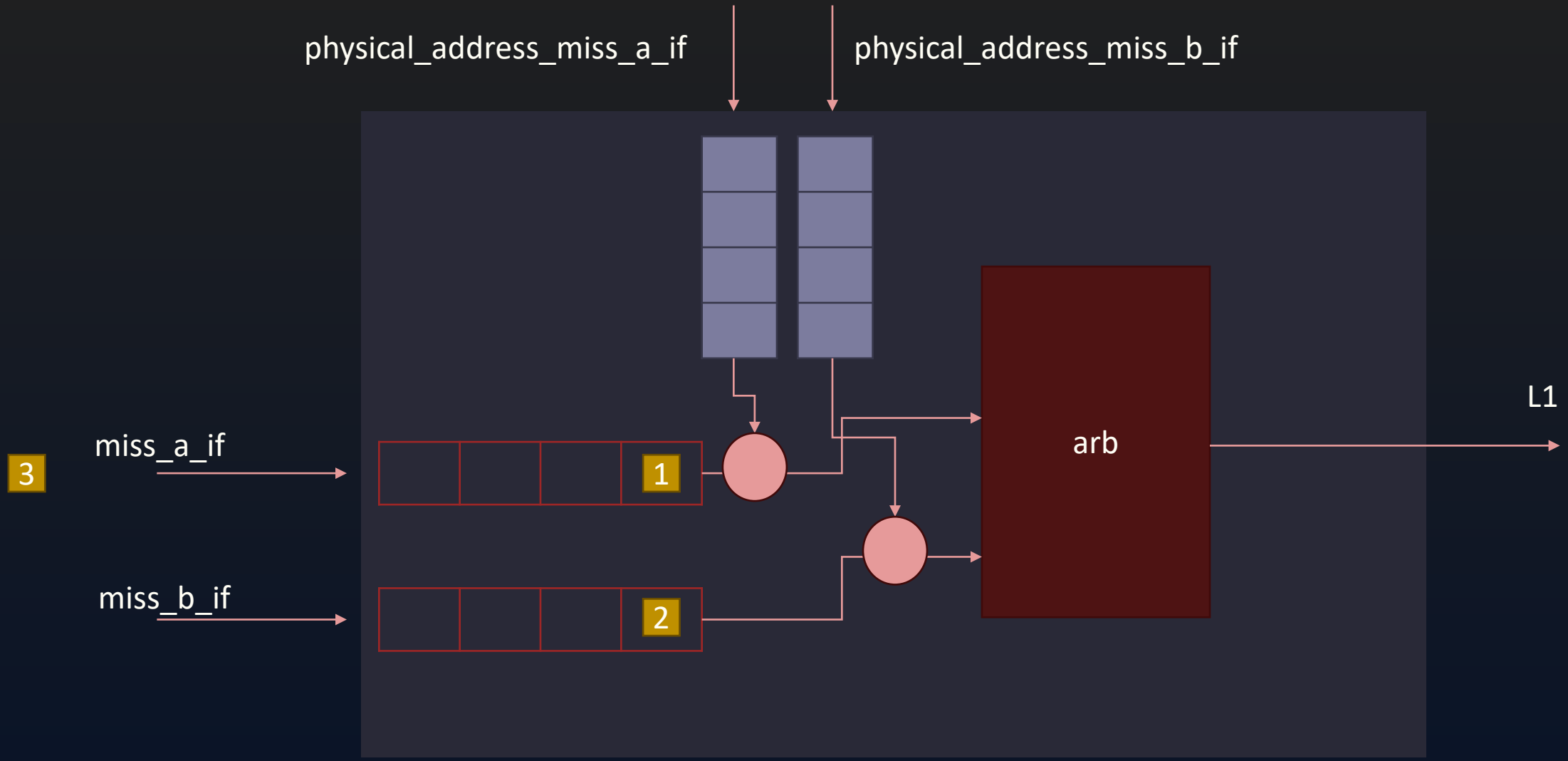
Multiple lookups, misses, virtual to physical translations, page-table faults

axiomise®

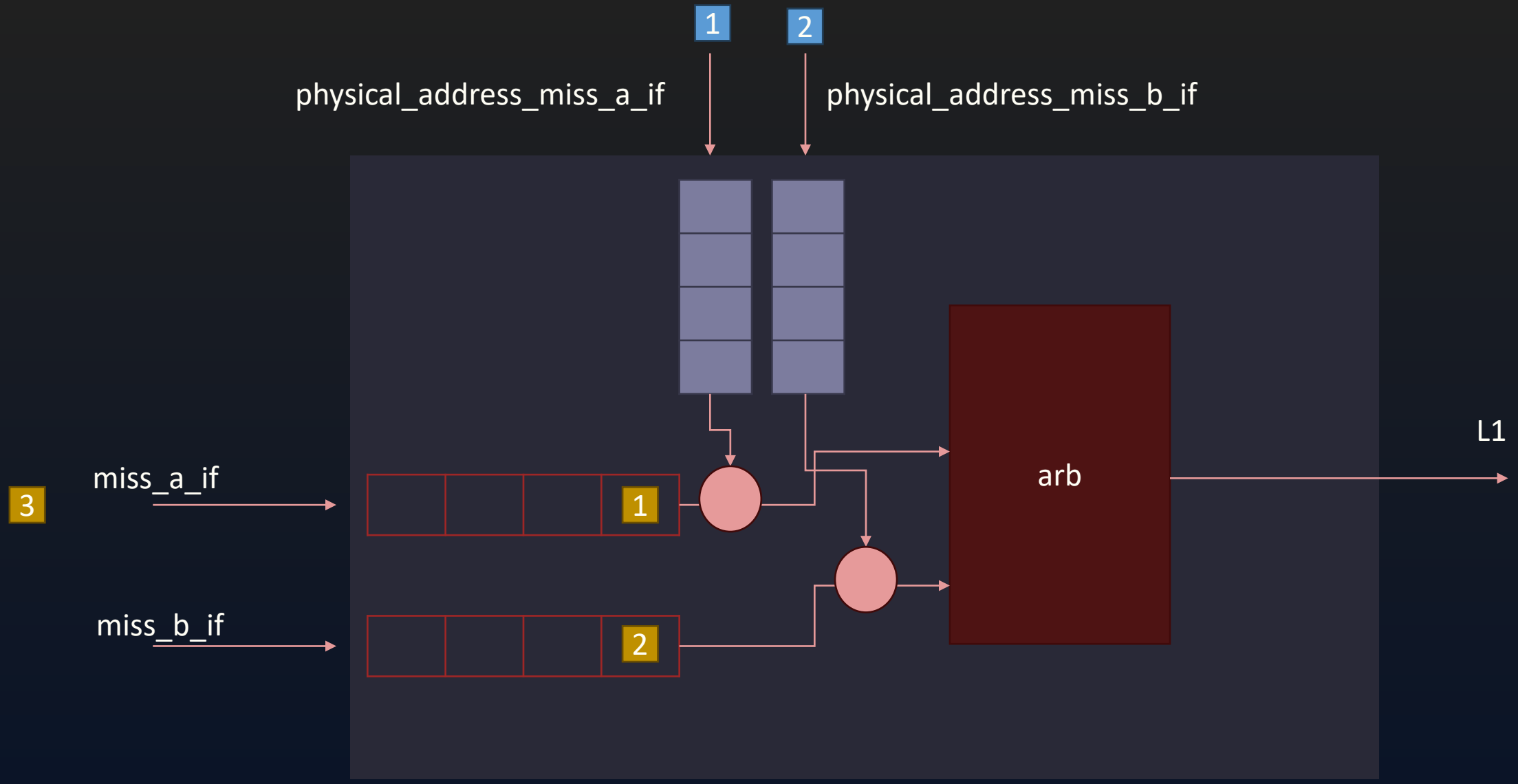
predictable formal verification



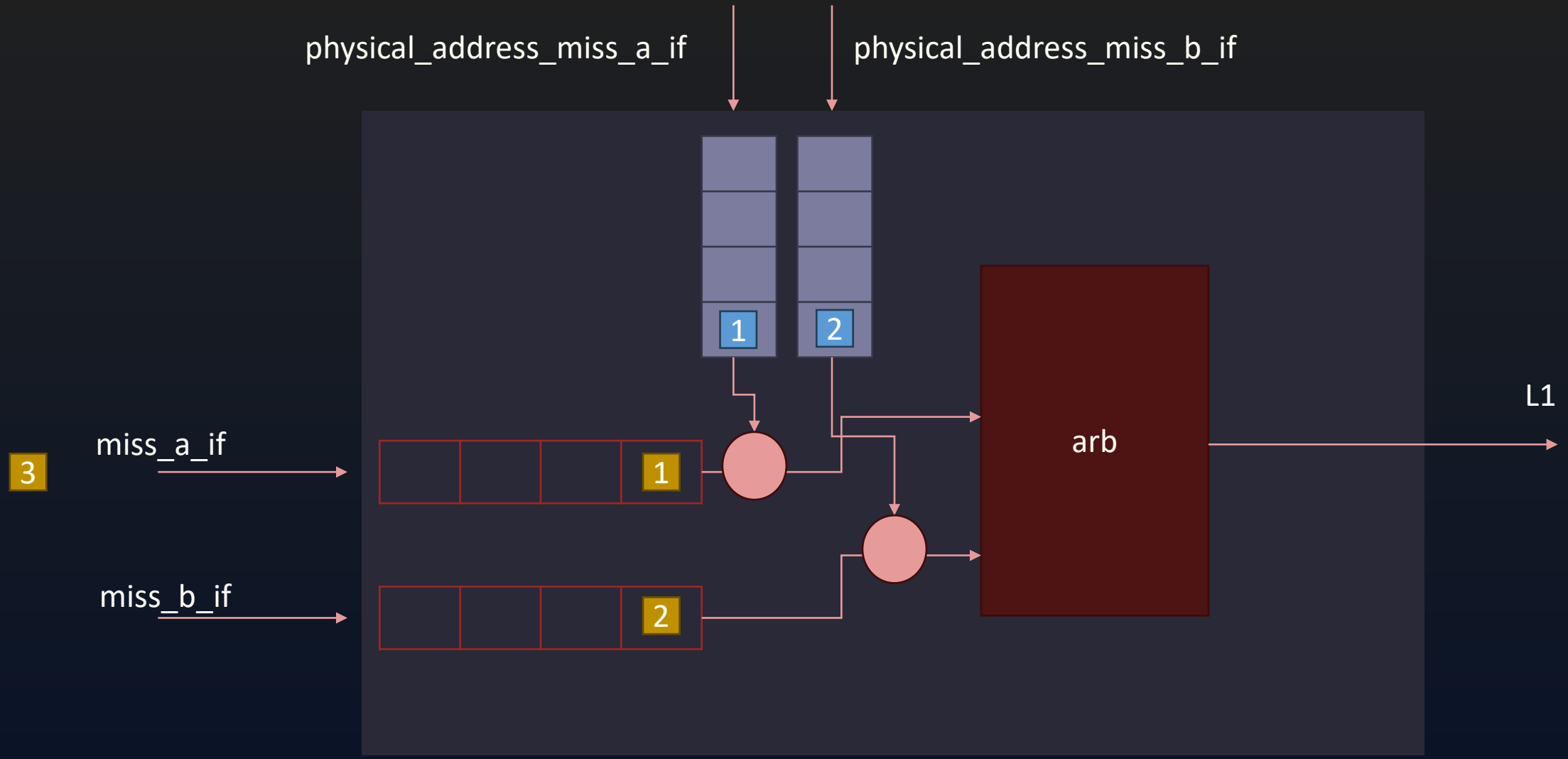
Two misses on port a and port b arrive at the same time.



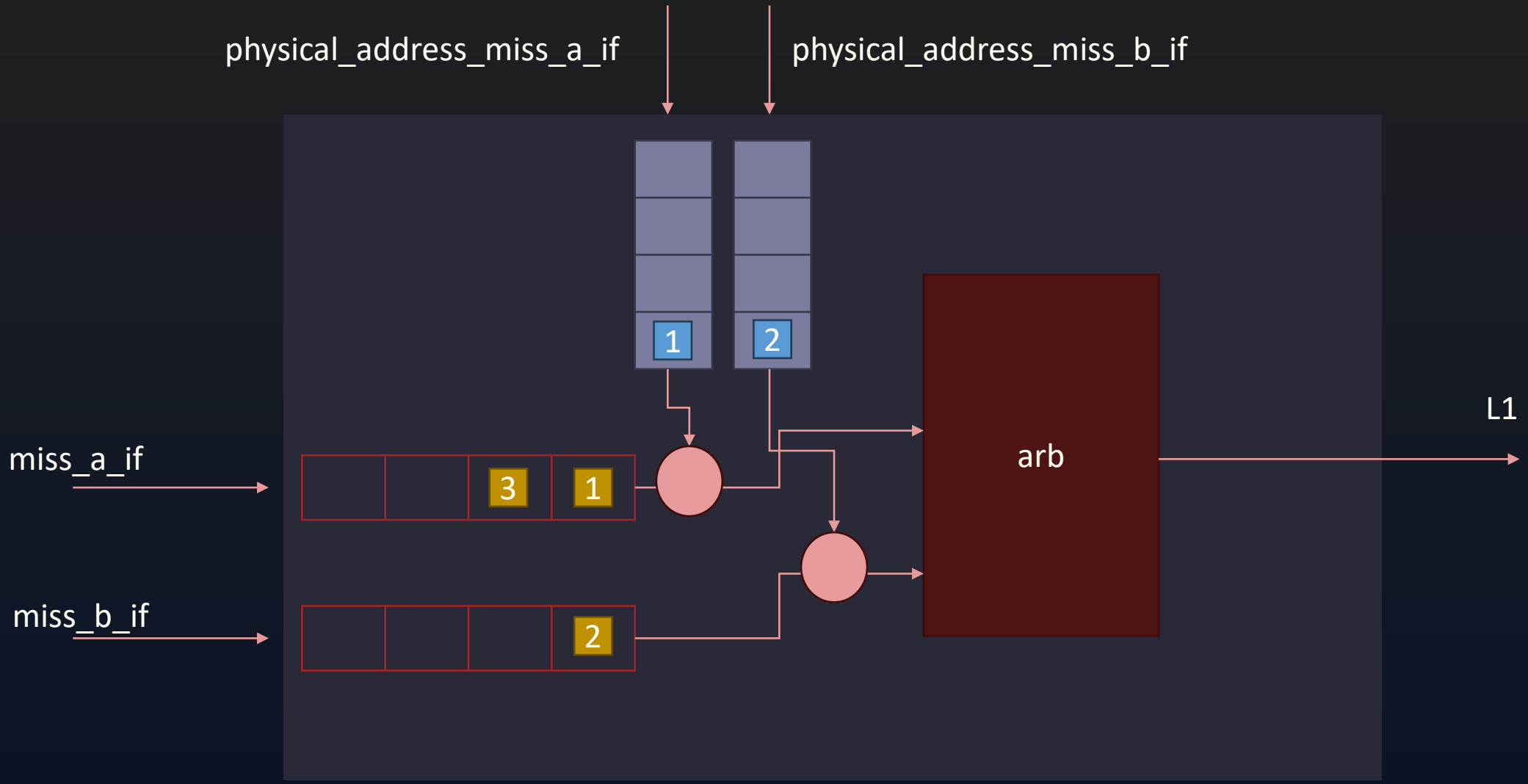
Two misses on port a and port b arrive at the same time.



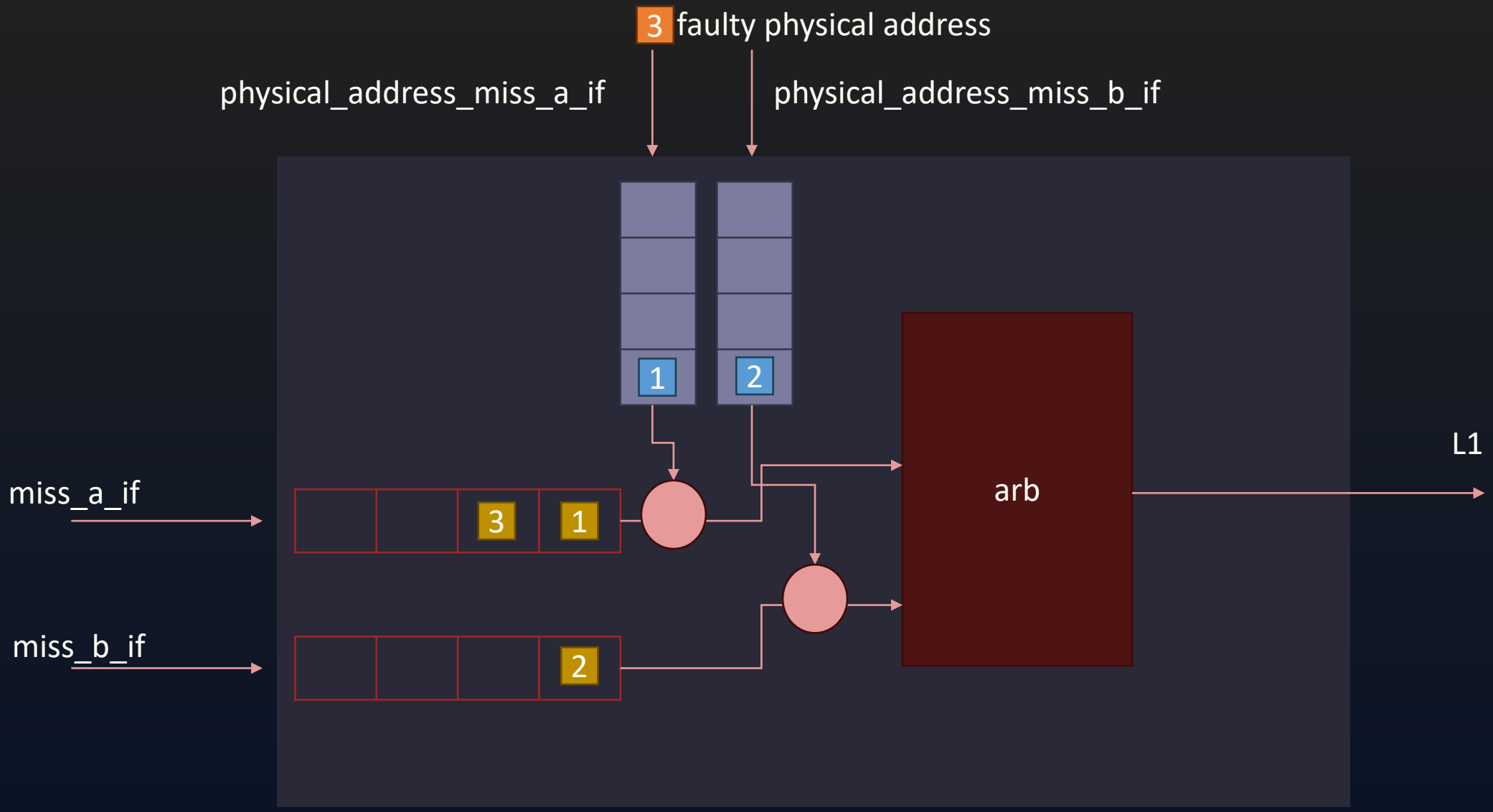
The physical address arrives afterwards for the two misses.



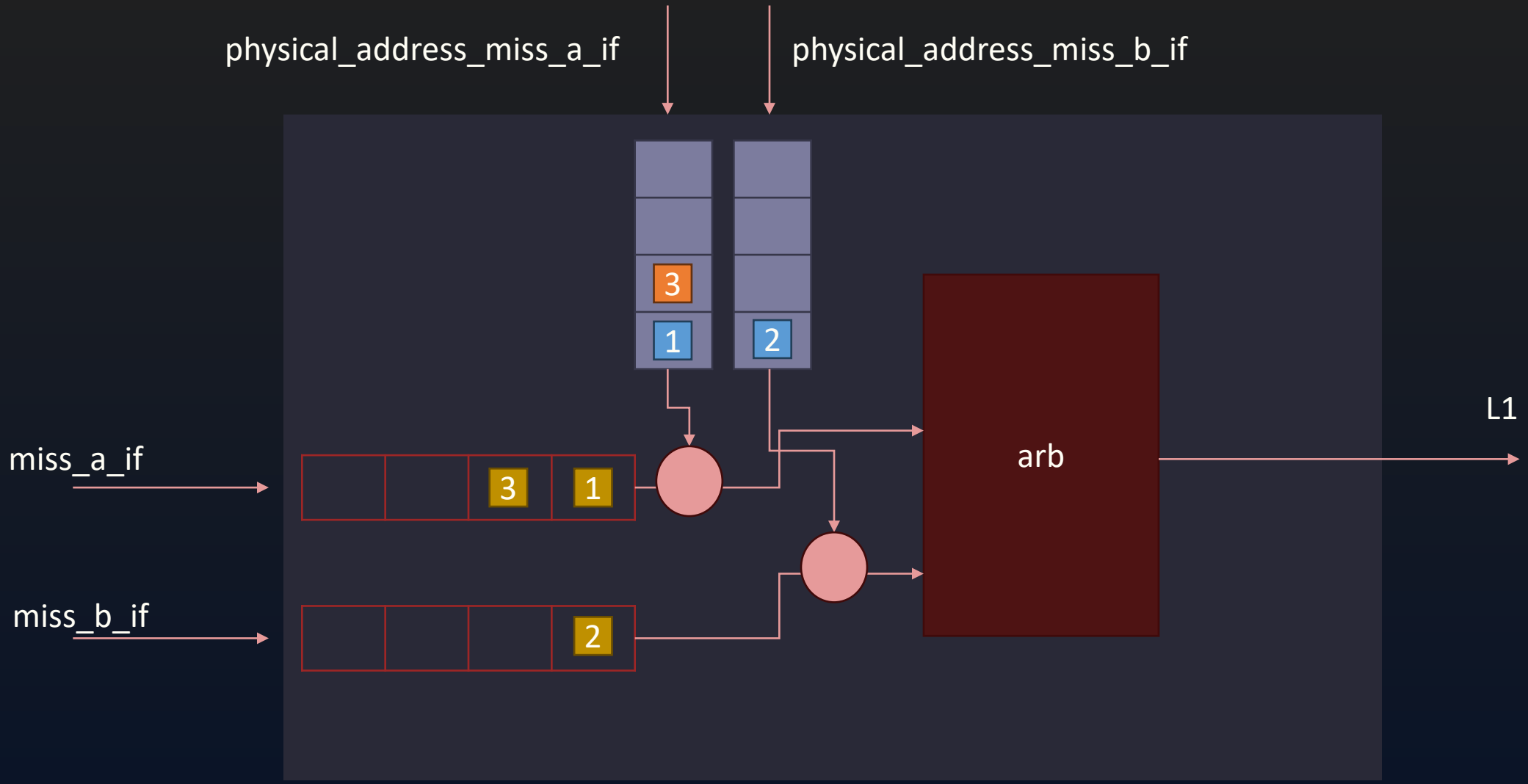
The physical address arrives afterwards for the two misses.



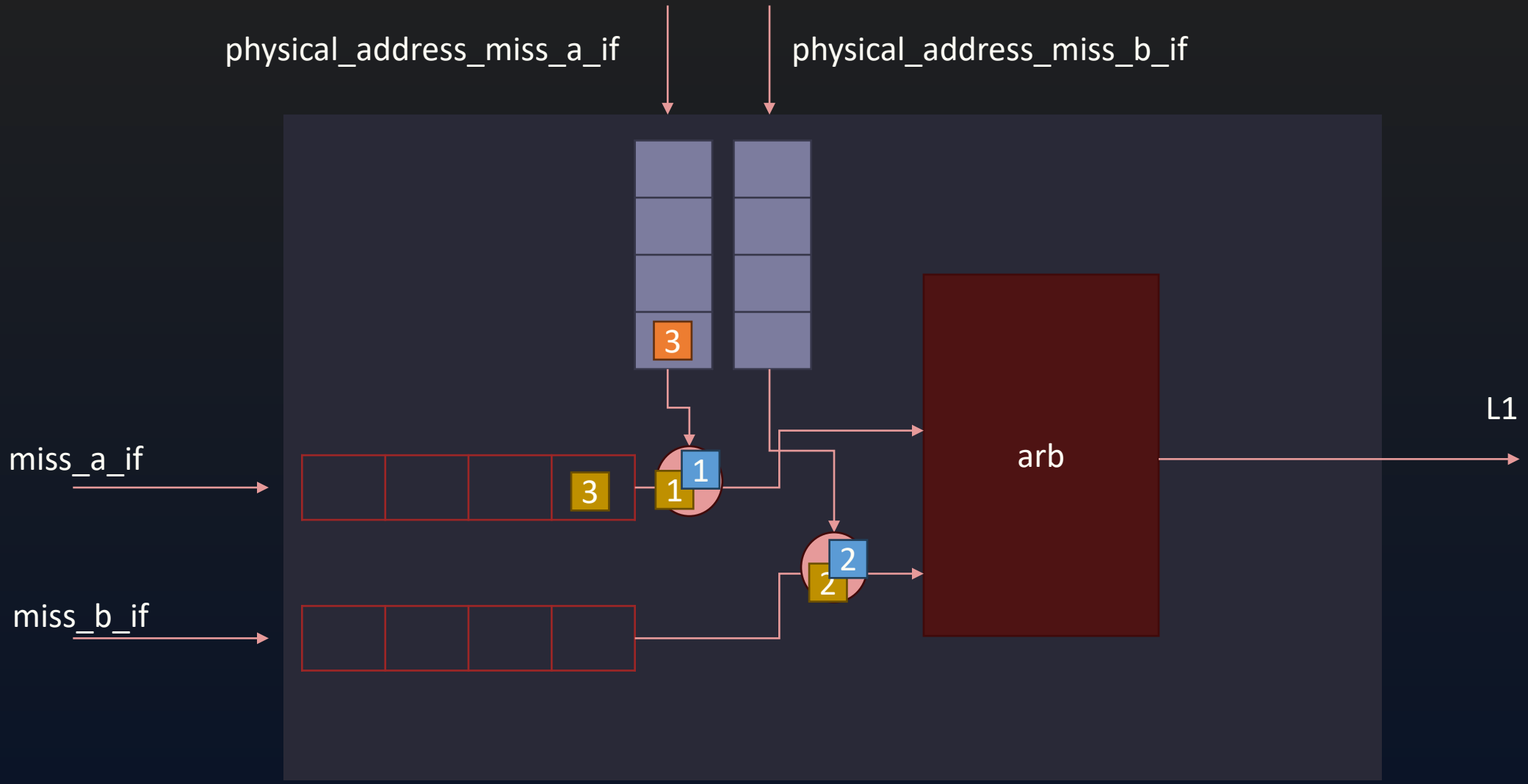
Third miss arrives on port a.



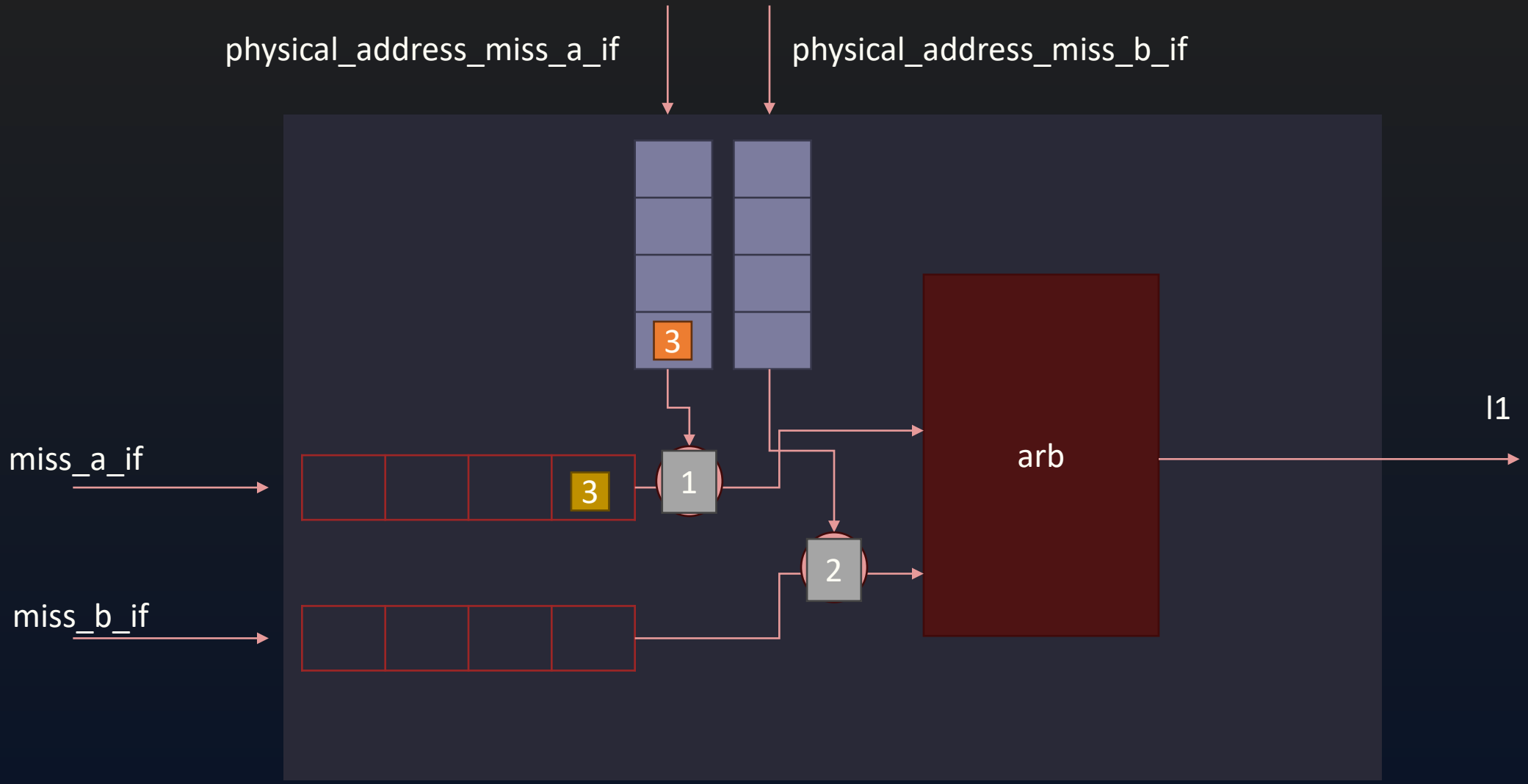
The physical address for the third miss arrives for port a but it is indicating a fault.



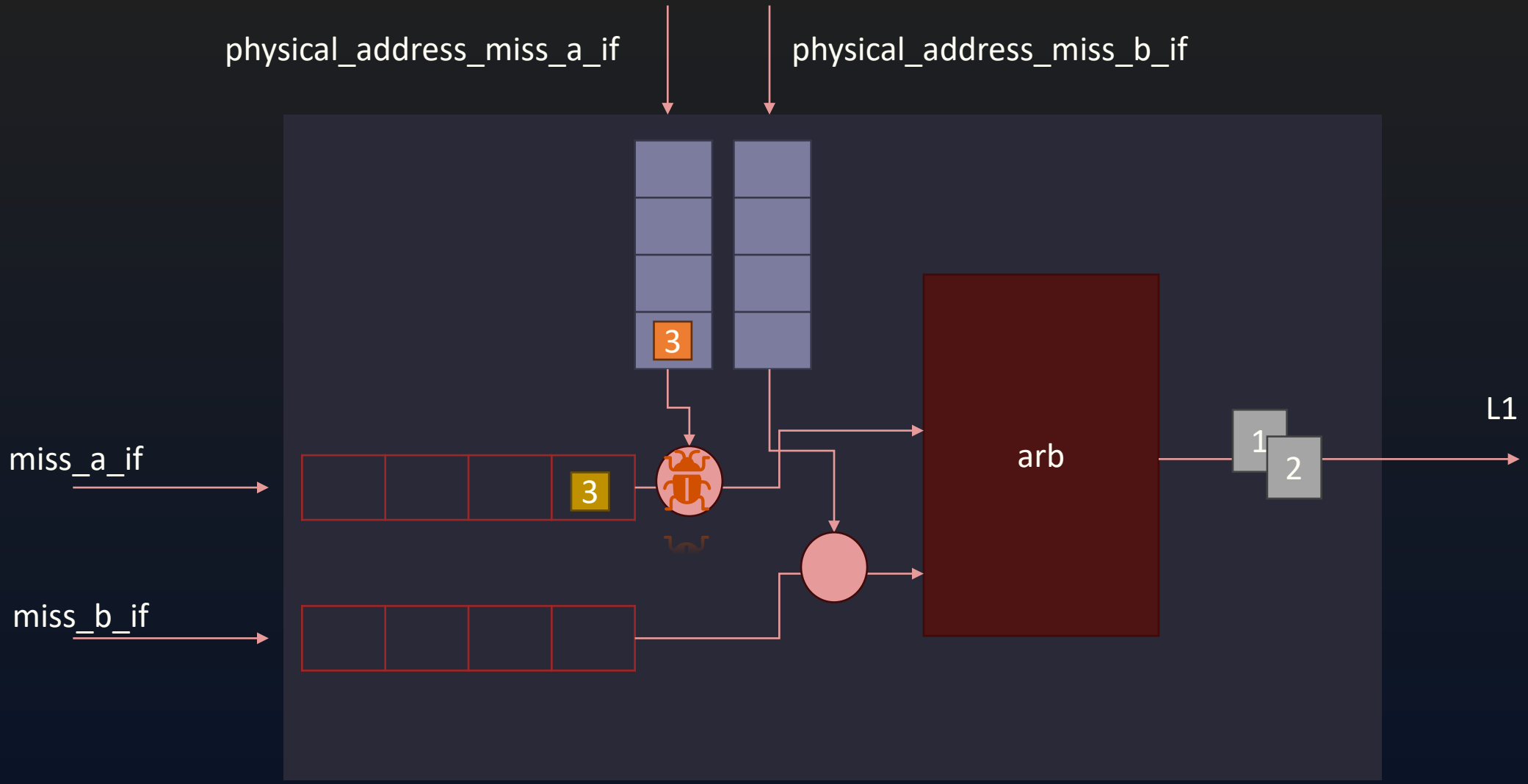
The physical address for the third miss arrives for port a but it is indicating a fault therefore the third miss should be dropped and not sent to L1.



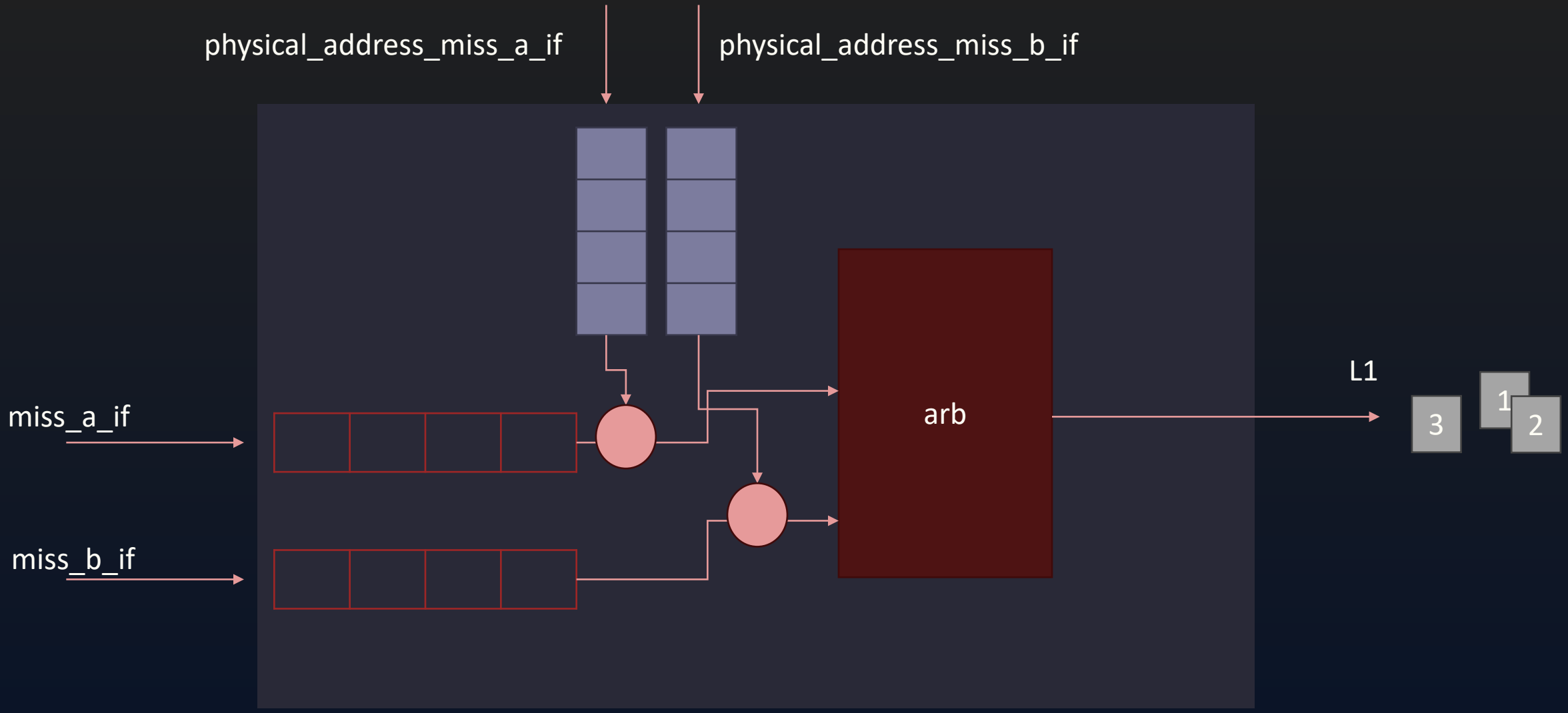
The miss on port a together with its physical address and the miss on port b and its physical address are processed.



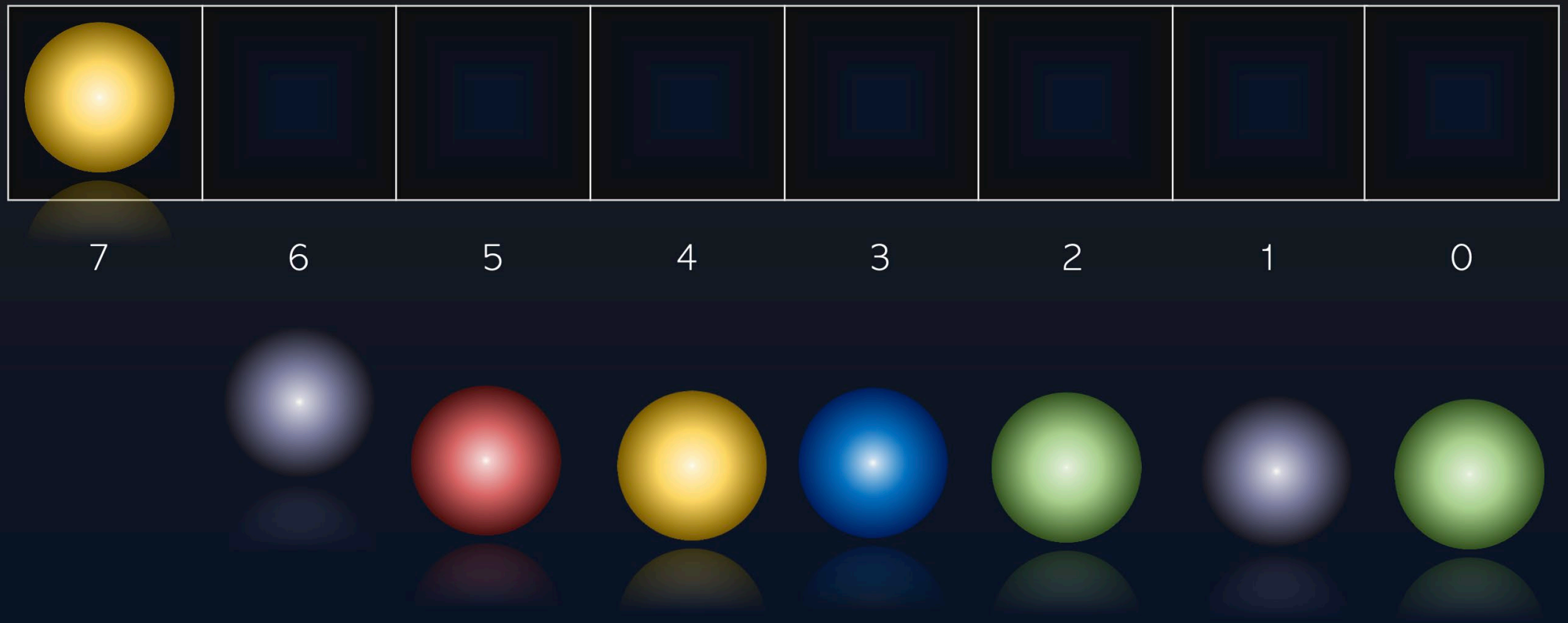
The miss on port a together with its physical address and the miss on port b with its physical address are processed.



Since they both have got the same physical address, they are combined and sent to L1.
A Bug in handling the faulty address…



Since the flag for faulty physical address is not cleared due to the RTL bug, the design proceeds to merge the faulty physical address with the non-faulty and sends the payload downstream to L1 instead of dropping the faulty physical address.



Two transaction tracking abstractions are used for verifying the cache subsystem, which helps find these kinds of bugs.

100% full proofs were obtained on the entire subsystem.

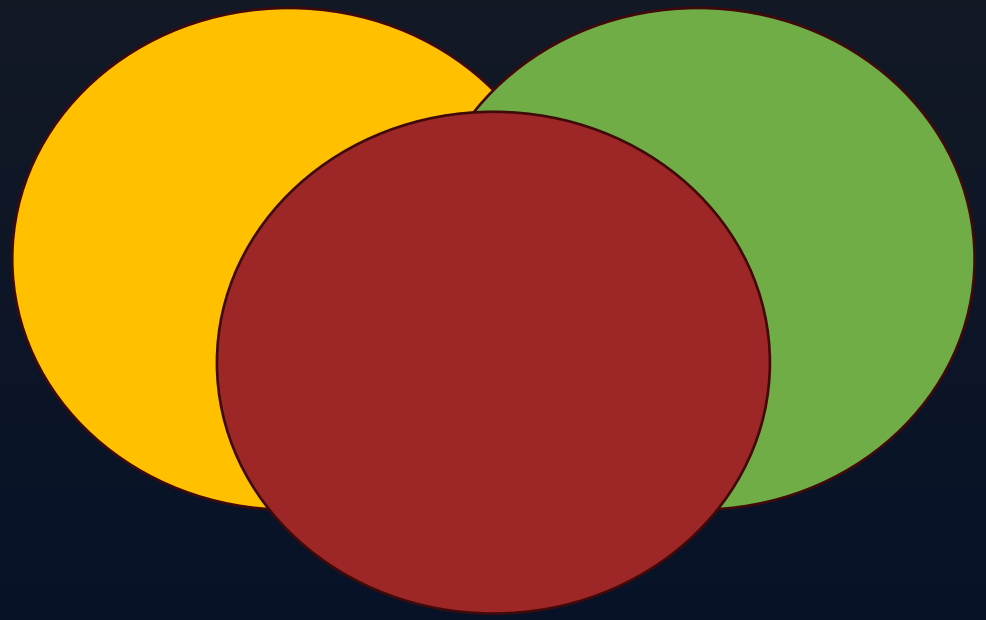
Formal Functional Verification is Cool

But what about PPA?

Existing Landscape of Tools

What they can and cannot find?

- **Linters** only perform structural analysis ; they cannot provide any redundancy analysis
- **Reachability** analysis only tells you what is “reachable”, not what is ”redundant”
- **Synthesis** will not guarantee safe removal of redundant components



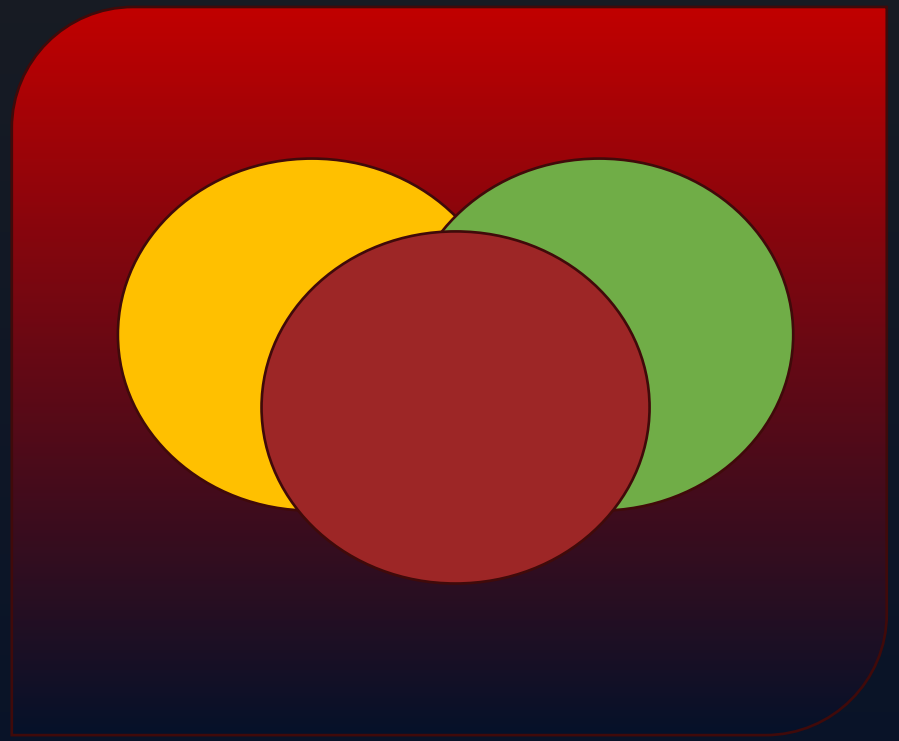
axiomise®
predictable formal verification

What is footprint®

PPA analysis at RTL level to increase design efficiency saving area and power

- Reachability
- Redundancy
- Complete Area Analysis
  - Full and Partial
  - Quantify redundancy on 100M+ gate designs
  - Component level
    - FIFO, Counter, MUX, FSM, Array, Register

footprint®
Area analyser for silicon



axiomise®
predictable formal verification

Footprint®

Area analyser for silicon

- ✓ Find out the redundant area in silicon
- ✓ Beyond reachability analysis and coverage
- ✓ Architects and designers can optimize PPA
- ✓ Exhaustive analysis powered by formal
- ✓ No need to learn formal
- ✓ Easy-to-use
- ✓ Zero noise
- ✓ Tested on 80+ designs

axiomise®
predictable formal verification

Example Results of footprint®

NoC

Gates saved per sub router: 8 * 4690 gates per instance = 37520 gates

Gates saved per router: 3 sub routers * 37520 gates = 112560 gates

Gates saved per Grid: (16 full routers * 112560) + (4 smaller routers * 37520)

=

1800960 + 150080 = 1951040 gates (1.95 Million)

For more testimonials, visit
www.axiomise.com/client-testimonials

FORMAL VERIFICATION

Typically, we focus on functional testing first and then performance; however, with the combination of tools that Axiomise used - *formalISA* for functional testing and *footprint* for performance and area - we were able to obtain valuable insights into both functionality and performance at the same time - something we hadn't fully appreciated formal verification could offer.

Charlie Hauck
CEO, Bluespec

axiomise
predictable formal verification

Functional and PPA is Cool

But what about Datapath?

floatrix™

Making formal normal for floating point

End-to-End sign-off for floating-point hardware

axiomise®
predictable formal verification

floatrix

End-to-end sign-off for floating-point hardware

floatrix™
Making formal normal for floating point

- ✓ Integrated with our Axiomiser® platform
  - Verify floating-point units in processors and GPUs.
- ✓ Support for all number formats relevant for AI/ML accelerators
  - FP64 (Double-precision), FP32 (Single-precision), FP16 (Half-precision), bfloat16, FP8, FP4
- ✓ Efficient bug hunting and proof convergence supported by Axiomise abstraction models
  - Catch corner-case bugs fast using Axiomise abstraction techniques and advanced proof engineering.
- ✓ Predictable run times
  - Prove the compliance of IEEE 754 and other floating-point micro-architecture implementations.
- ✓ Intelligent debug and reporting using SURF
  - Integrated floating point debug and report profiling.

axiomise®
predictable formal verification

Why Axiomise?

Covering the entire spectrum of verification requirements

High Proof Convergence

So that you have higher confidence

Bugs & Exhaustive Proofs

Making sure your design is bug free

Innovation in Abstractions

Allowing you to have the highest quality designs, without re-spin

Scalable Proof Engineering

Our solutions scale as your designs do

High Quality Sign-off

Functional Safety Security PPA

We find bugs that no-one else can; nobody gets proof convergence like us

axiomise®
predictable formal verification