Internal Use Only

Formal Verification with Cadence Jasper

Hugh Lo
Sr. Principal Application Engineer

Dec. 2024

cadence®

Formal Verification Introduction

What is formal verification?

Formal Verification = Mathematical analysis of the space of possible behaviors of a design

Family of tools/apps: toolkit, not 1 tool

The best way to verify: **best use of engineering and math tools**

Can analyze from inputs, outputs, or states

No need to drive particular testcases

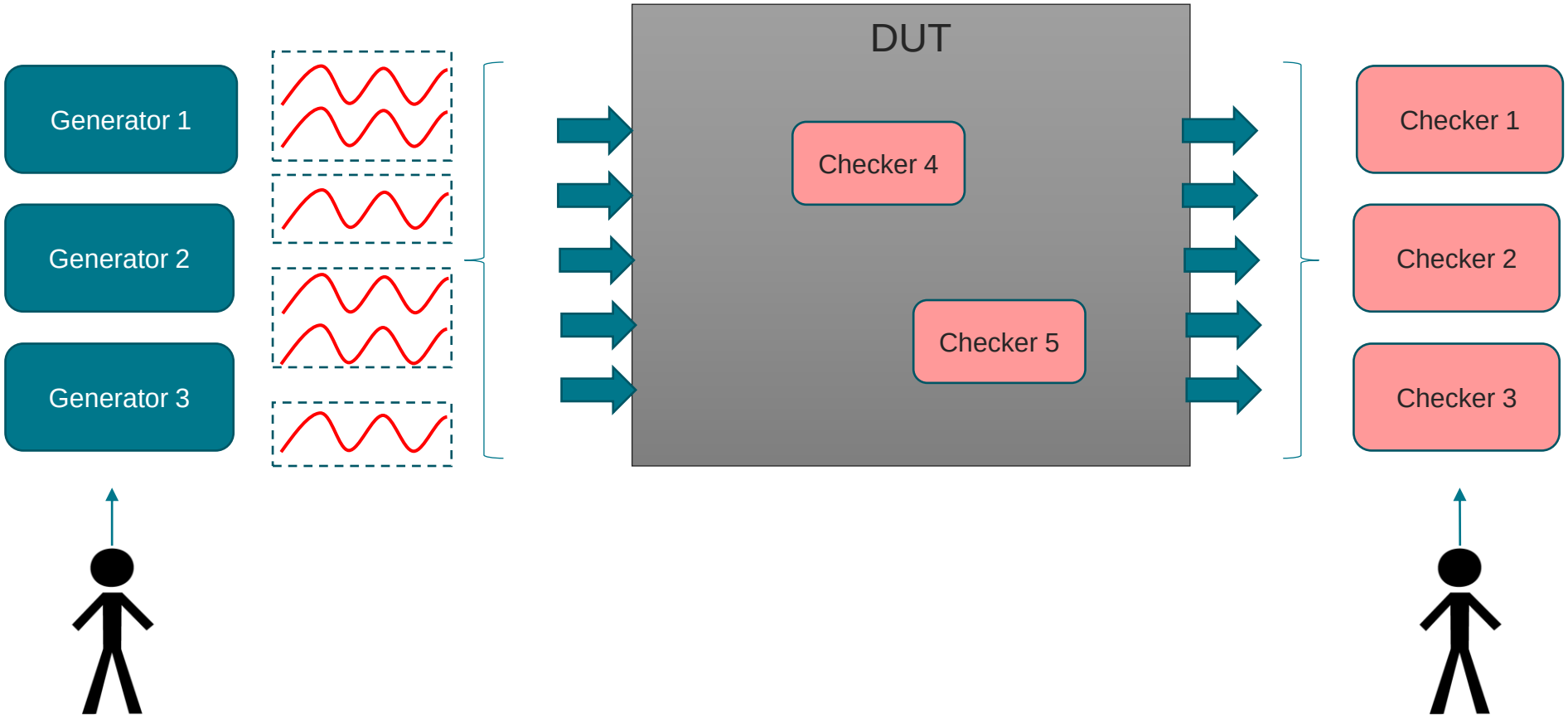
- Checks exhaustively if a model meets a given spec
  - Model ➔ synthesizable RTL
  - Spec ➔ properties
- Key differences between **simulation** and **formal**

	Simulation	Formal
Scope	Simulation can only detect bugs	Formal proves absence of bugs
Inputs	User creates given stimulus set	User specifies only illegal stimulus
Testbench	TB is a complicated wrapper around design	TB is a set of properties connected to design

Simulation : Input-Driven

DD/DVs create generators to drive stimulus and sensitize the design

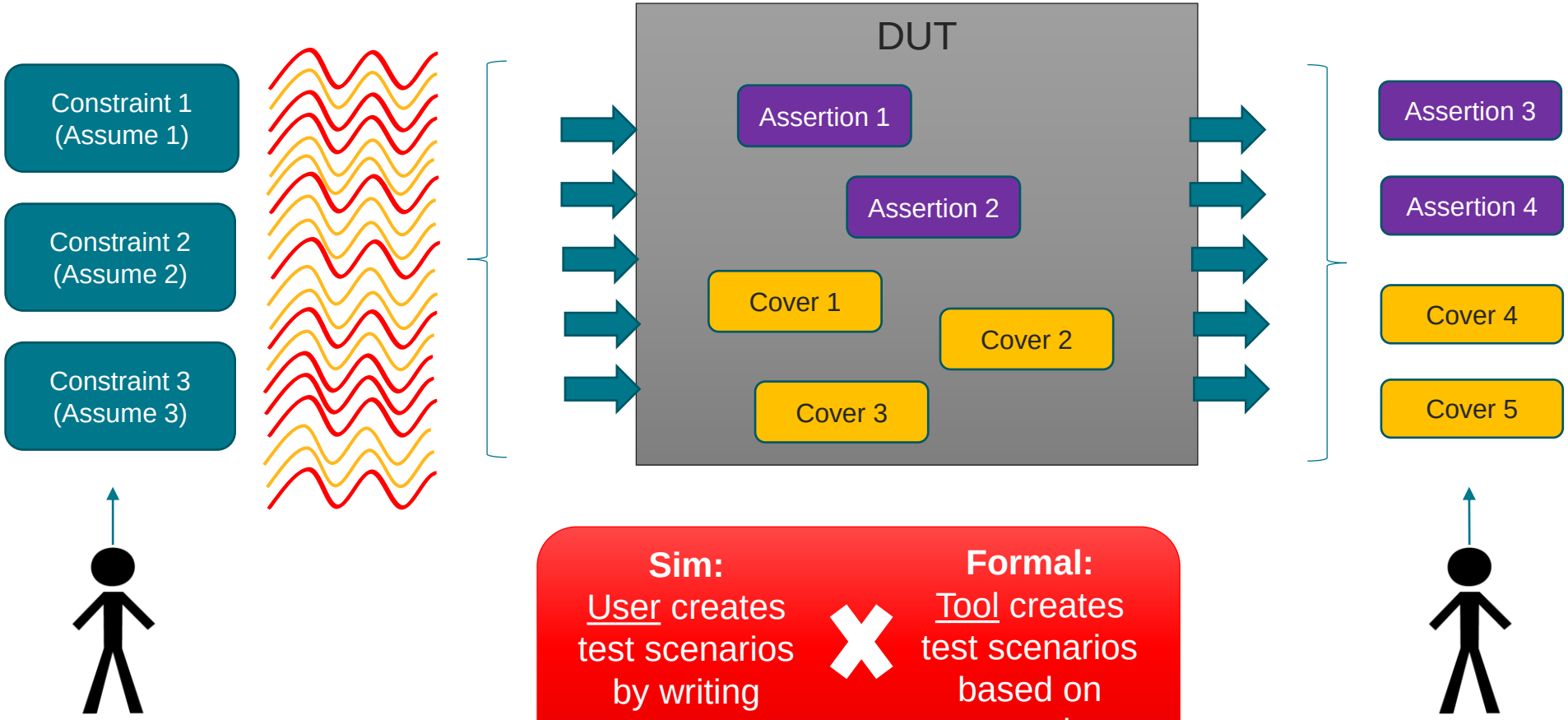
DD/DVs create checkers to observe design and flag for errors



Formal Analysis : Spec-Driven

Initially formal will drive **all possible stimulus** through the design (legal and illegal)

DD/DV create **assertion/covers** to list the behaviors/specs **which wants to be verified**



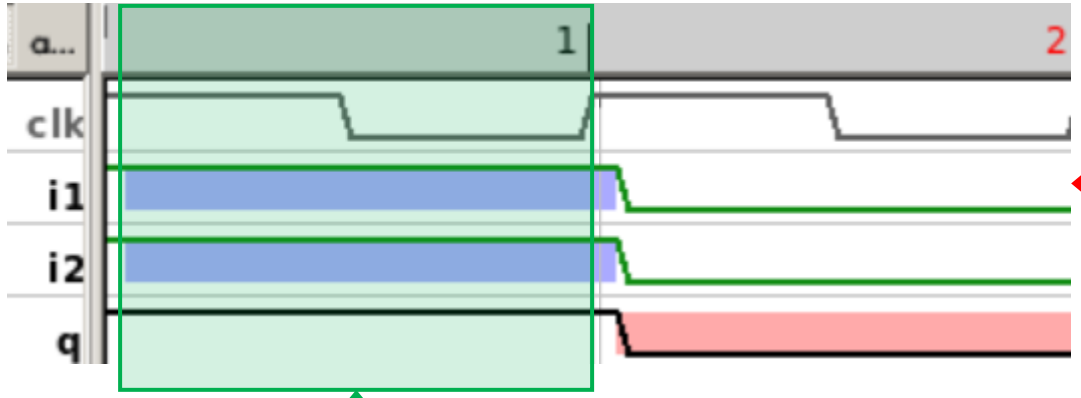
Sim:
User creates test scenarios by writing testbench

Formal:
Tool creates test scenarios based on properties

Reachable States in Formal

- Let's analyze which states of a DUT are reachable
  - Example: 2 inputs (i1, i2) and 1 internal element q

```
q = i1 & i2;
```

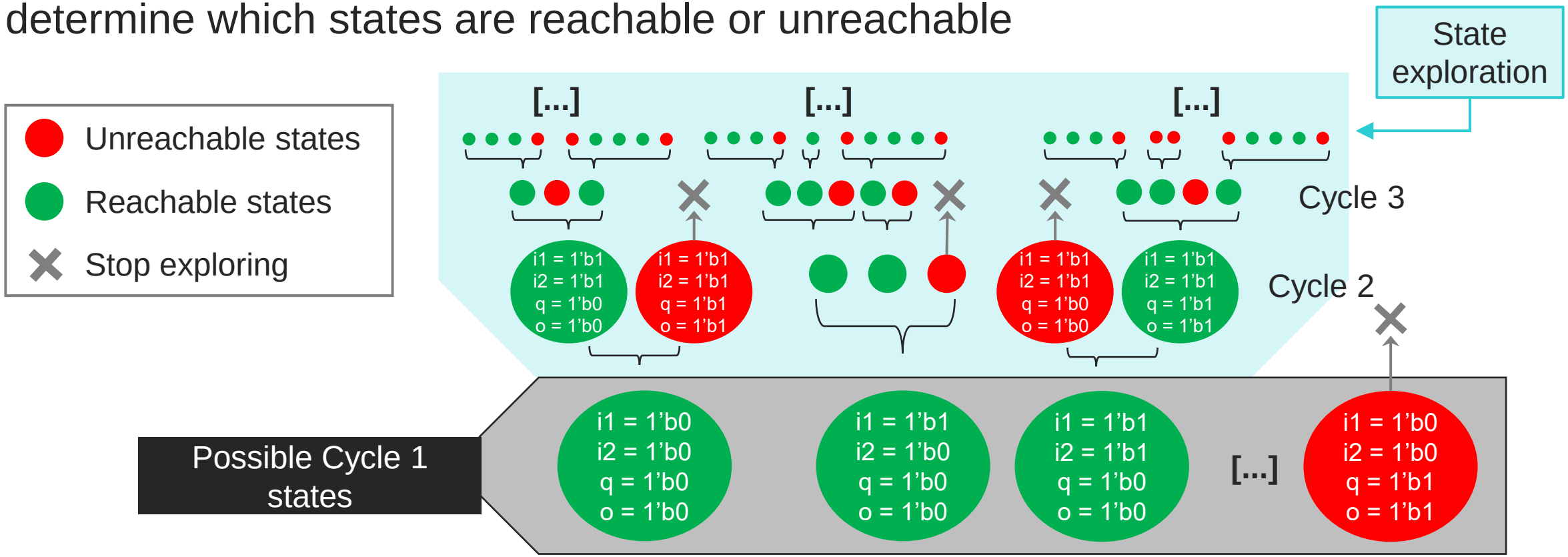


If "i1" and "i2" are 1'b1 in the **1st** cycle. "q" being 1'b1 in the **1st** cycle is a reachable state

In the **2nd** cycle, "i1" and "i2" are 1'b0 and "q" is 1'b0, which is also a reachable state

Design State Space in Formal

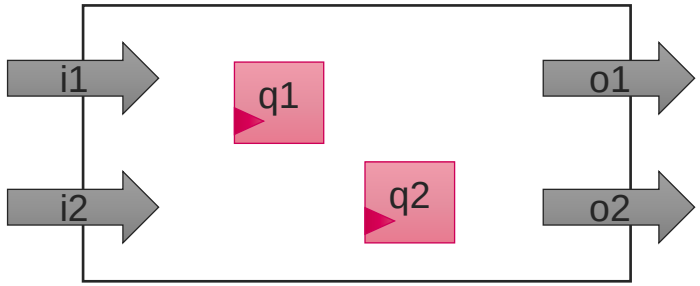
- Formal uses math algorithms to create the State Space for a given DUT and determine which states are reachable or unreachable



Unreachable states are not going to be analyzed during the property validation

Property Analysis

- After we have all the reachable states, formal solves each of your properties against them
  - Report a Counter-Example (CEX) when assert properties are violated by a reachable state
  - Report a cover trace when cover properties are hit by a reachable state
  - Report "unreachable" if none of the reachable states can hit cover properties
  - Report "proven" in none of the reachable states violate the assertions




```
✔ cover (q2 ##2 o1)
```

Waveform saved in database

```
✘ assert (o1 && o2 |=> q1)
```

Waveform saved in database

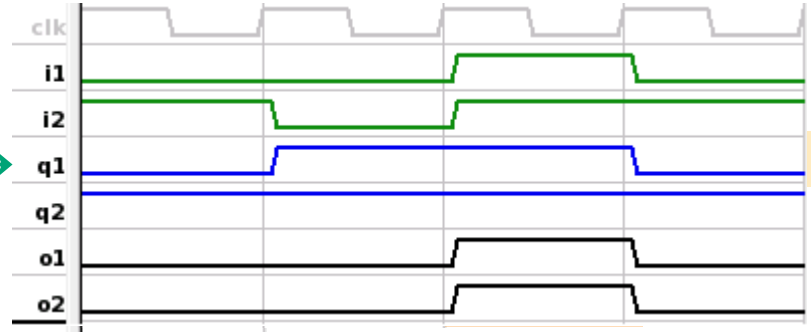
```
✔ assert (!o1 && !o2 |=> q2)
```

No reachable state violated this assertion

Only reachable states

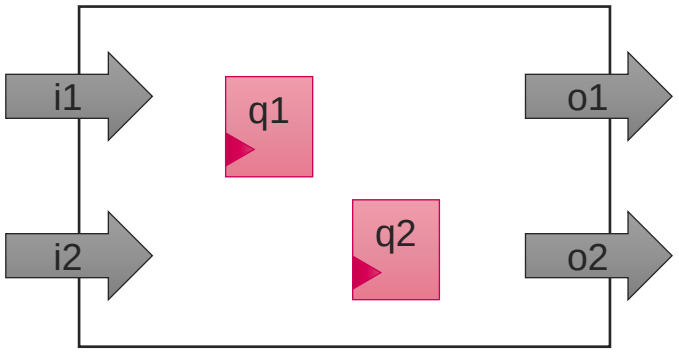
Found scenario that hits cover!

Found scenario that violates assertion!

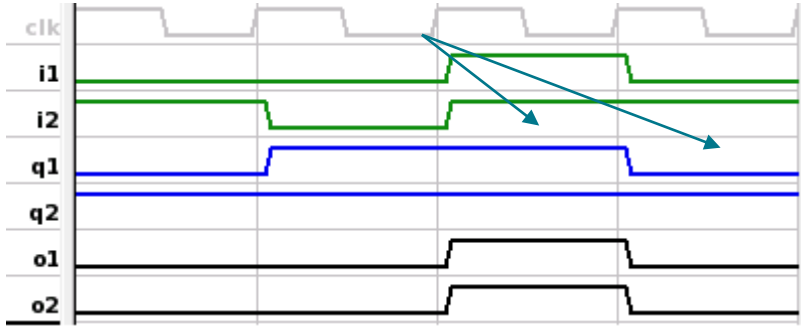


Constraints

- Not all reachable design states are legal in the context of the design functionality
- Assume properties tell formal what is legal
  - Any trace that is found for a cover property will honor all assumptions
  - Any failing assertion (counter-example) will honor all assumptions




```
assume (i1 |=> i2)
```




```
assert (o1 && o2 |=> q1)
```

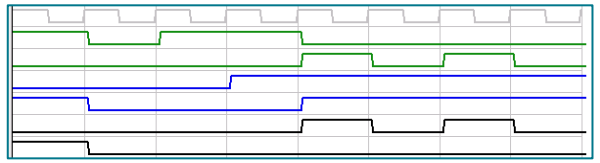
Proof Results

Not possible with simulation!

CEX or Covered

	Type	Bound
✗	Assert	12
✓	Cover	49

- Formal found a state that hits a property
- Assertion failure
- Cover hit
- Waveform available



Undetermined

	Type	Bound
?	Assert	7 -
?	Cover	7 -

- Formal analyzed a subset of all reachable states
- Assertion cannot fail in less than 7 cycles
- Cover is not reachable in less than 7 cycles

Proven or Unreachable

(Full Proof)

	Type	Bound
✓	Assert	Infinite
✗	Cover	Infinite

- Formal analyzed all reachable states
- Impossible to violate assertion
- Impossible to hit cover

A Simple Example

Constraints

```
// a) If FIFO is full, then there shouldn't be any further writes
asm_no_write_when_full: assume property ((full |-> !write_en));

// b) If FIFO empty, then there shouldn't be any further reads
asm_no_read_when_empty: assume property ((empty |-> !read_en));
```

Control inputs

Assertions

```
// c) FIFO cannot have full and empty asserted at the same time
ast_no_full_and_empty: assert property (!(full && empty));

// d) FIFO must keep full asserted until a read occurs
ast_remain_full_until_read:…((full & !read_en) |=> full);

// e) FIFO must keep empty asserted until a write occurs
ast_remain_empty_until_write:…((empty & !write_en) |=> empty);
```

Verify DUT

A Simple Example

Formal Verification

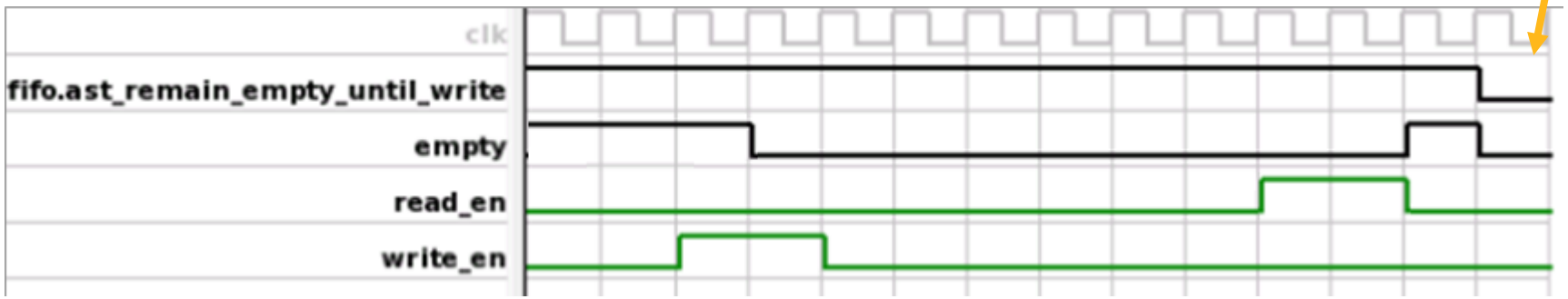
Full Proof: Impossible to violate this assertion

Undetermined: No failure found in 157 cycles

Counterexample: Found 14-cycle failure

	Type	Name	Engine	Bound
	Assume	fifo.asm_no_write_when_full	?	
	Assume	fifo.asm_no_read_when_empty	?	
	Assert	fifo.ast_no_full_and_empty	N (3)	Infinite
	Assert	fifo.ast_remain_full_until_read	B	157 -
	Assert	fifo.ast_remain_empty_until_write	N	14

Assertion Failure



Strengths and Challenges of Formal

Strengths

- More comprehensive than simulation
- Simpler debug than simulation
- Leads to higher quality
- Improves productivity and schedule

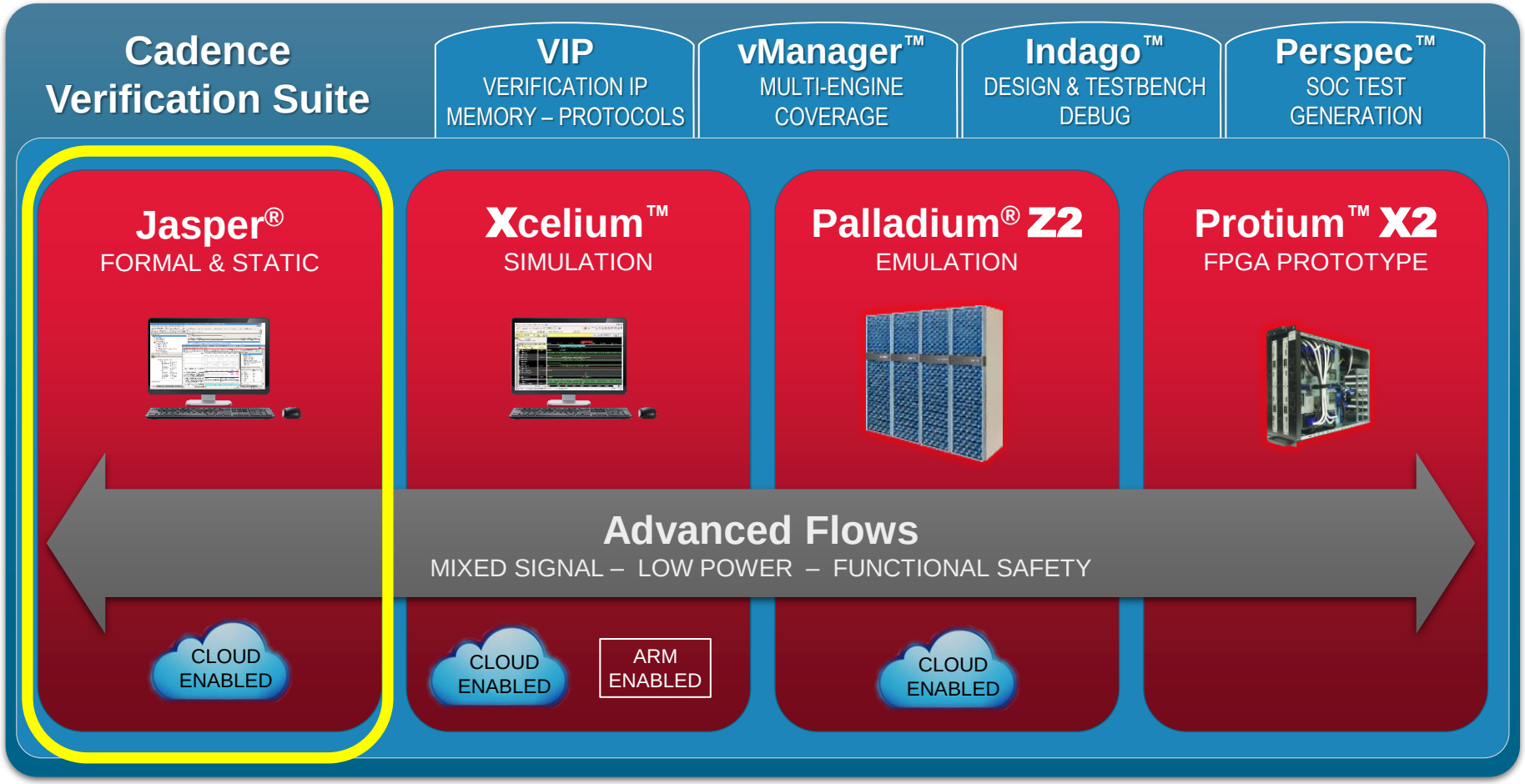
Challenges

- Requires a different mindset than simulation
- Consider the DUT complexity
- Metrics are just as important as simulation
- May require learning some formal techniques

Cadence Jasper Platform

Jasper Formal Apps

- Cadence Verification Suite

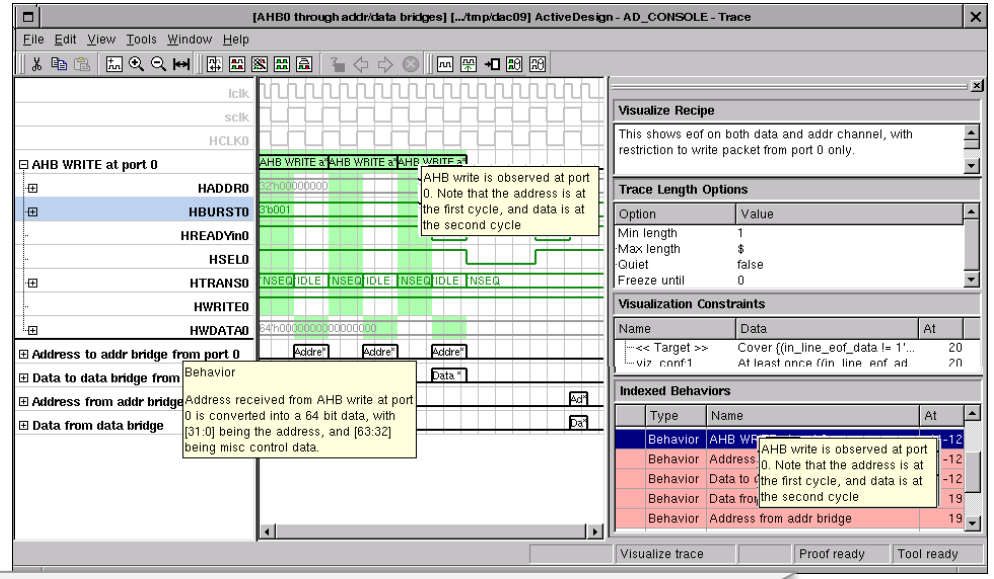


Jasper : Easiest Formal Verification Platform

Solve **specific verification problems** with targeted Jasper® Apps

- Formal Property Verification App
- SuperLint (AFL) App
- Design Coverage Verification App
- Sequential Equivalence Checking App
- X-Propagation Verification App
- Control/Status Register Verif. App
- Connectivity Verification App
- Coverage Unreachability App
- Clock Domain Crossing App
- Functional Safety Verification App
- Low Power Verification App
- Security Path Verification App

Highly interactive **formal debug** transforms to fit the App



Broad **formal engine** and infrastructure

Assertion Based Verification IPs for AMBA and other common protocols

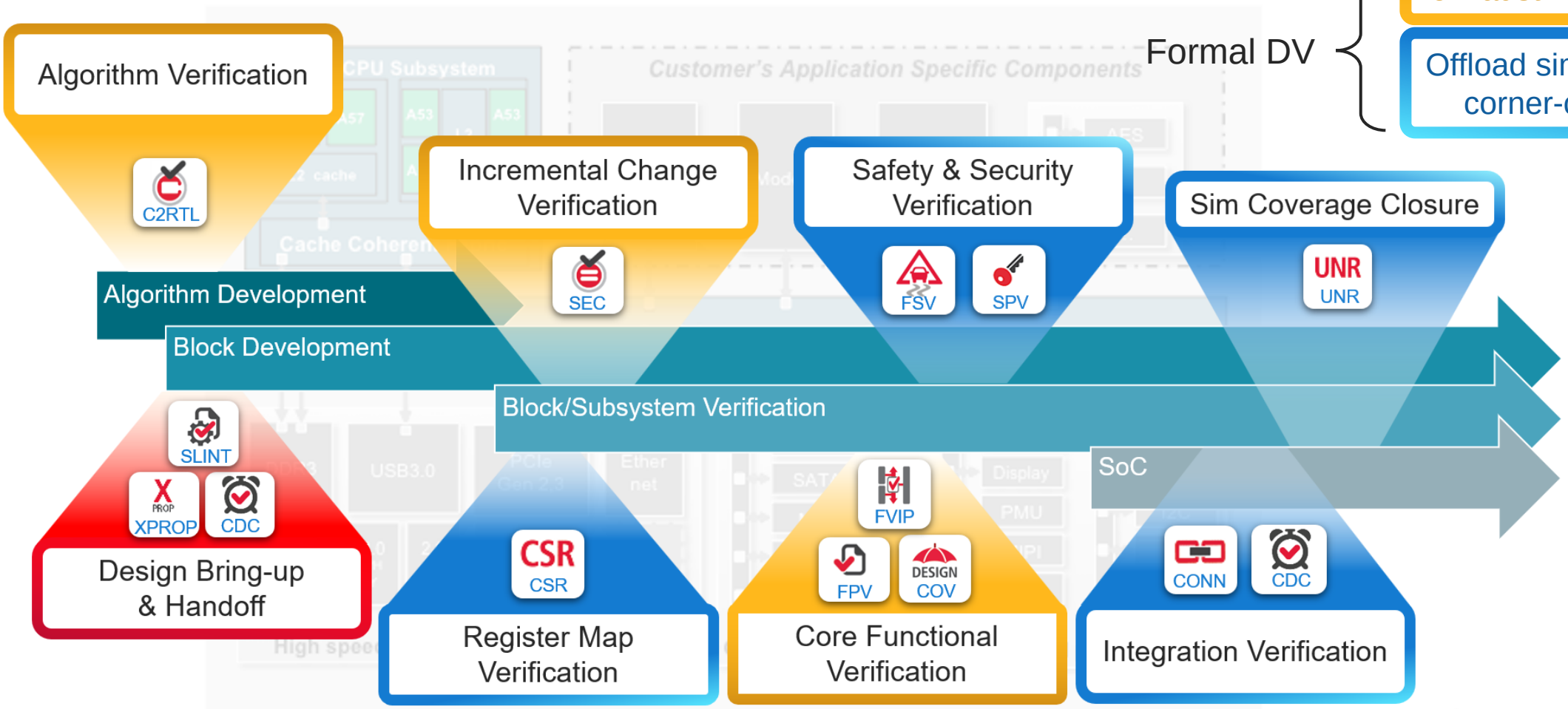
Programmable Interface via TCL

ProofGrid™ Manager assigns best engine for task

Breadth of Jasper Apps Across Flow

Designers
- Handoff higher quality code earlier (shift left)
- Improve quality through exhaustive verification

Formal DV
- Offload simulation, find corner-case bugs



Jasper Use Models

Expanding your horizons with Formal Verification

"FV wherever we can, Simulate where we must"

FPV
Formal Property Verification App

Core Formal

RTL Design Analysis

Superlint
SuperLint App

XPROP
X-Propagation Verification App

CDC
Clock Domain Crossing App

Fantastic for Formal

JasperC
JasperC and C2RTL

CONN
Connectivity Verification App

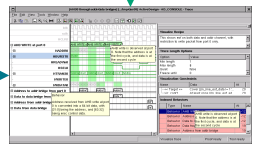
SEC
Sequential Equivalence Checking App

CSR
Control/Status Register Verif. App

Bug Hunting and Bug fix validation

DBH
Deep Bug Hunting

Formal to "Shift Left" and catch bugs sooner



Jasper Visualize and debugging tools

FormalVIP
VIPCAT FormalVIP catalog

Formal to Complement Simulation

UNR
Coverage Unreachability App

Formal Sign-off

COV
Design Coverage Verification App

Security and Low-Power Verification

SPV
Security Path Verification App

FSV
Functional Safety Verification App

LPV
Low Power Verification App

cadence

How are customers using Jasper?

Formal to "Shift Left" | RTL Design Optimization

- Leverage JG-Superlint for Linting and Auto-formal checks
- Enable FormalVIP for fast ramp-up on protocol verification

"FV wherever we can" | Formal Sign-off

- Select the best tool for each verification objective
- Jasper can enable innovative approaches to your verification problems
- Use Formal Techniques where it's high effort to generate stimuli testbenches for simulation or to catch complex corner case bugs

Security and Low-Power Verification

- Security Path Verification (SPV), Functional Safety Verification (FSV), and Low Power Verification (LPV) to prevent security holes in hardware

Hewlett Packard Enterprise — Verifying a Complex Crossbar with Multiple Transaction Types

Texas Instruments — FSM automatic formal check methodology for broad deployment

intel — Accelerate DDR-PHY Formal Verification using DFI5 FVIP

Google — Bootstrapping Formal Coverage Analysis

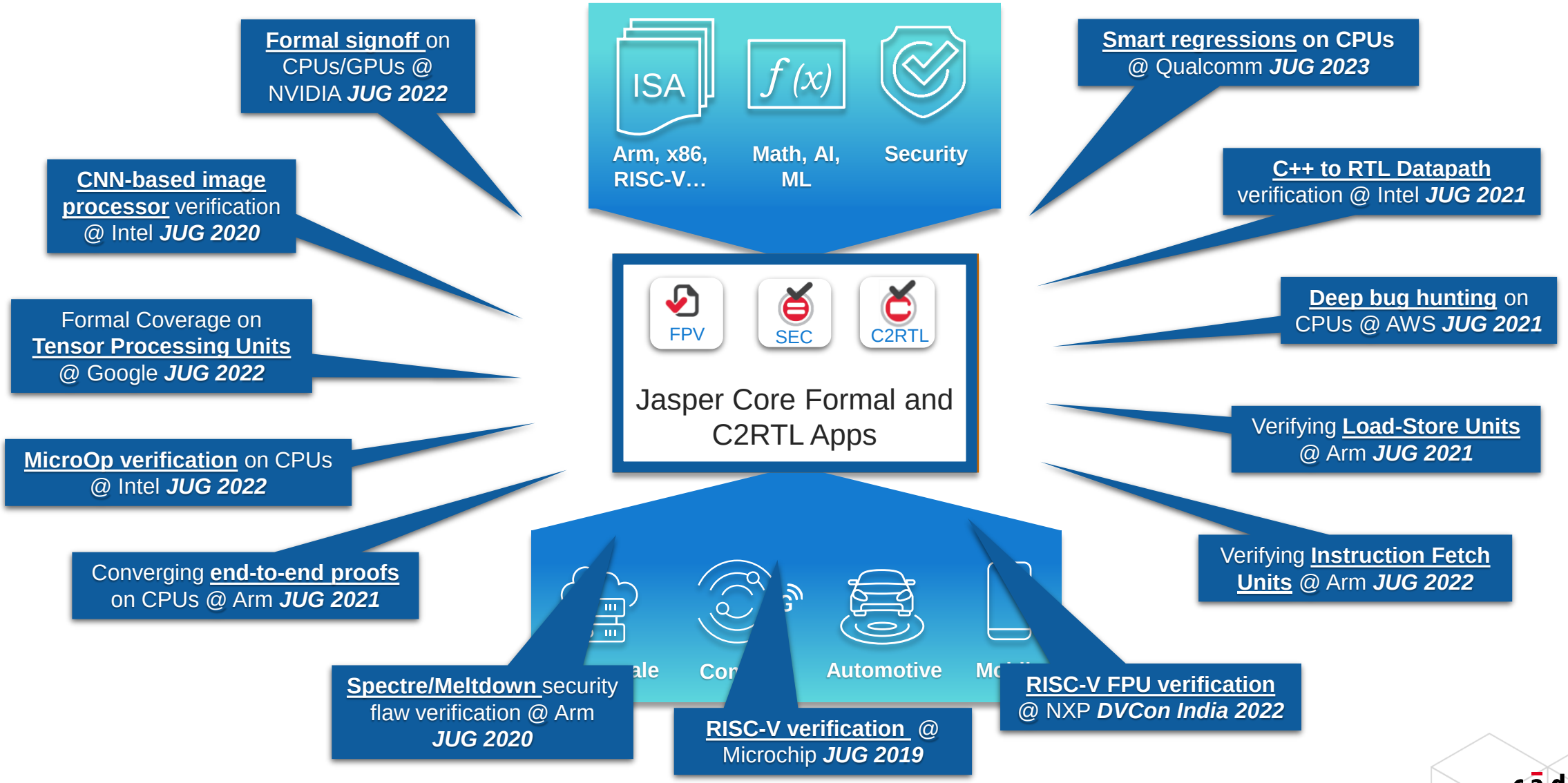
NVIDIA — End-to-End Formal Signoff Methodology

intel — FV Signoff in the Context of Mainstream Formal Verification

intel — HW Security Path Validation Using Formal Methods: Intel Case Studies

arm — Towards Enabling Security Formal Verification of the Load-Store Unit of A-class Arm CPUs using SPV App

Customer Successes



Best-in-Class Jasper Formal Verification Platform

- Jasper™ is the industry's leading formal verification platform
  - Adopted in 19 of the top 20 semiconductor companies.
- Fastest and most scalable formal verification solution
  - Proves properties and finds bugs faster, on wider range of bigger designs.
  - Largest R&D team by far ensures we stay ahead.
- Easiest formal verification solution to adopt
  - Comprehensive range of formal apps that automate property generation for specific tasks.
  - Powerful root-cause analysis and design exploration with the Visualize™ environment.

Formal Technology Leadership

- Higher verification throughput.
- On bigger designs.
- With optimal compute resource (in-house or cloud).

Introduction to SVA

What is SVA?

- SVA stands for System Verilog Assertion

- SVA is a language for expressing **properties**
  - Not only assertions, but covers and assumptions too!
  - Can be mixed with Verilog, SystemVerilog, and VHDL

- SVA was part of old SystemVerilog Accellera standard
- IEEE approved SystemVerilog as IEEE Std 1800-2005 on 11/09/2005
  - LRM can be downloaded from: http://ieeexplore.ieee.org

Formal Testbench

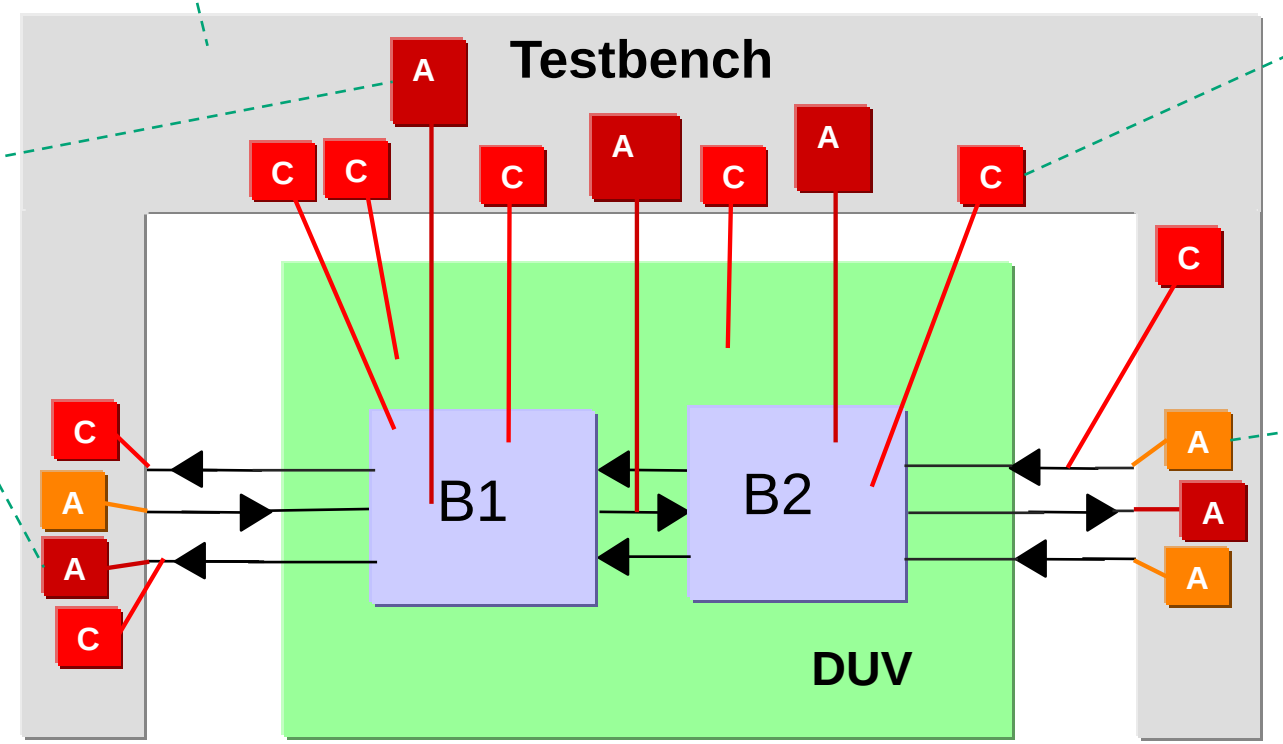
Properties described in **SVA syntax**

Most of testbench is contained in **SVA file**

Covers can target any signal

Asserts check outputs and internals

Assumes constrain inputs



HDL

Being Successful with SVA

- First, describe intent in your **natural language**, then code
  - *"A and B should never be high at the same time"*
  - *"if X happens, then I should see Y within N cycles"*
  - *"either P or Q should be low if design is in state S"*

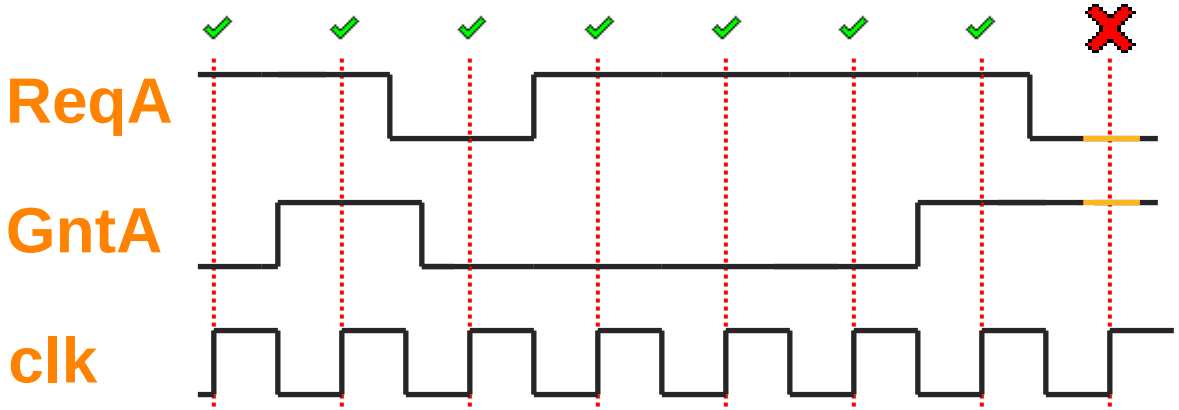
- The key to learning SVA is to learn a **small productive subset** of the language
  - Only 5-6 operators and 3-4 built-in functions is all you need!

- Write complex properties using **glue logic**, NOT complex SVA operators
  - Simple Verilog logic to keep track of events/state: state machines, counters, FIFOs, etc.
  - Refer to glue logic in SVA properties

SVA Example: Invariants

- Something that should always or never happen!
- e.g. “Should never see a Grant without a Request”

```
no_GntA_without_ReqA: assert property (not(GntA && !ReqA));
```

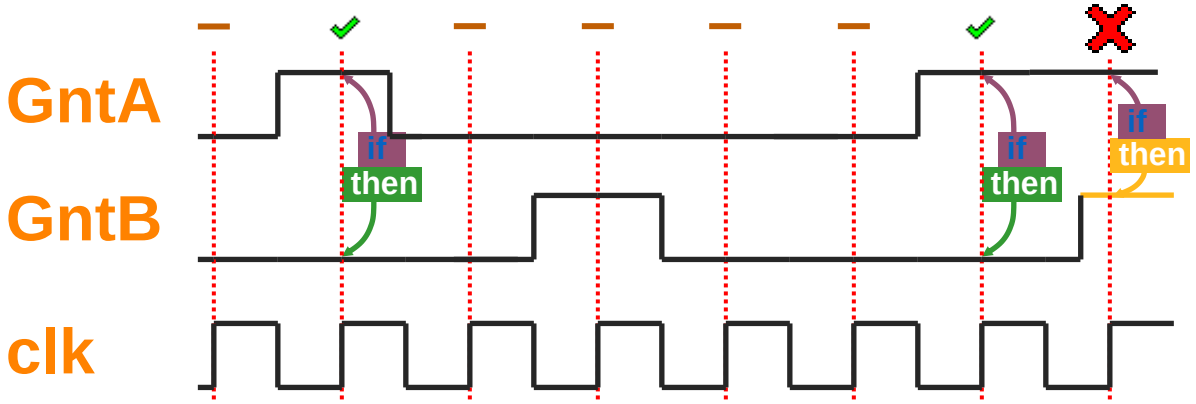


SVA Example: Same-Cycle Implications

- Something that should never happen IF a condition is met
- Assertion holds when:
  a) Condition is met and consequence is true
  b) Condition is not met
- e.g. “If A gets a grant, then B must not”

Implication operator |-> expresses “if...then”

```
GntA_then_not_GntB: assert property (GntA |-> !GntB);
```

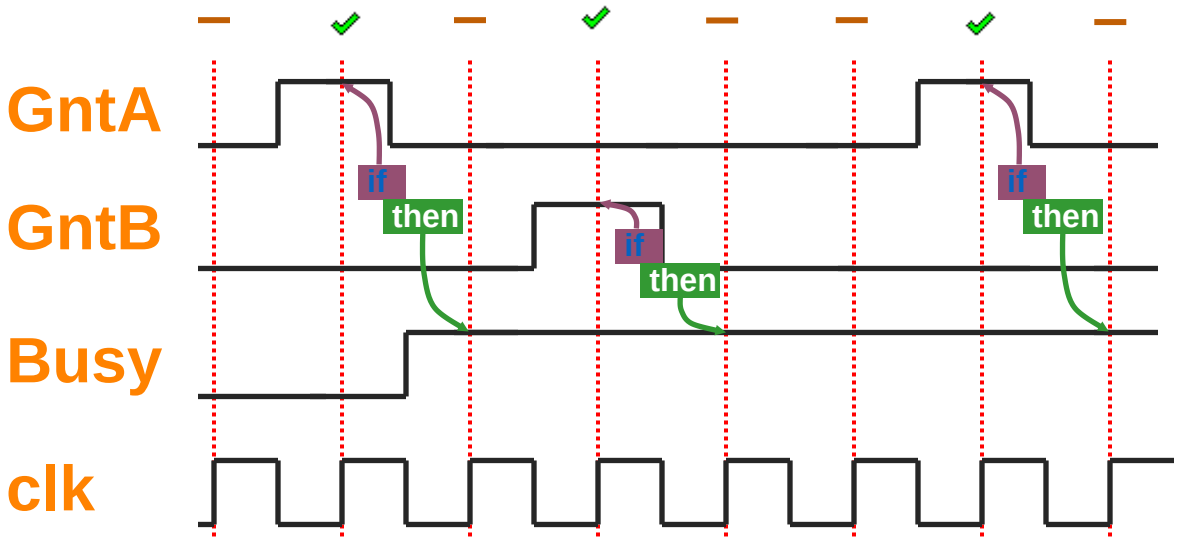


SVA Example: Next-Cycle Implications

- Possible to delay checking consequence by one cycle
- e.g. “any GntX is always followed by Busy”

Next-cycle Implication operator |=> adds a one-cycle delay between “if” and “then”

```
Gnt_followed_by_Busy: assert property ((GntA || GntB) |=> Busy);
```

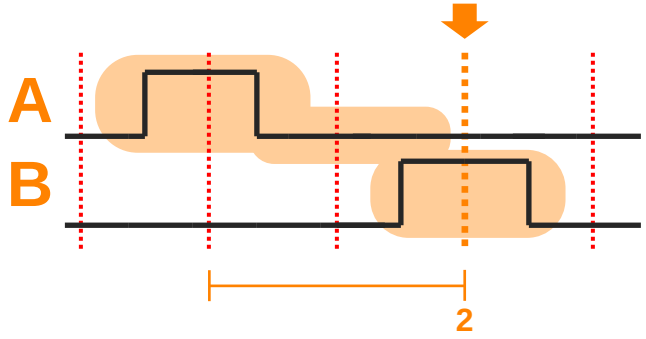


Sequences: Multi-Cycle events

- An intermediate cover point used to specify an order of events in a property
- Sequences are described using **##** operator

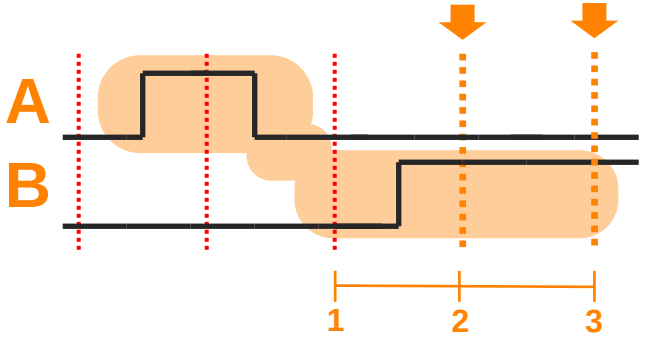
`A ##2 B`

"A happens then exactly 2 cycles later B happens"



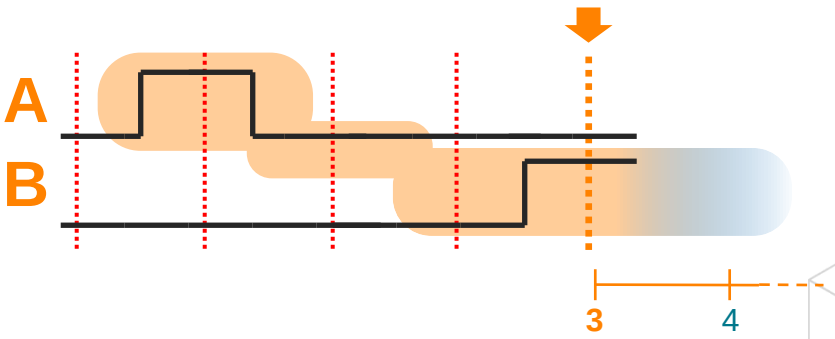
`A ##[1:3] B`

"A happens then 1 to 3 cycles later B happens"



`A ##[3:$] B`

"A happens then 3 or more cycles later B happens"

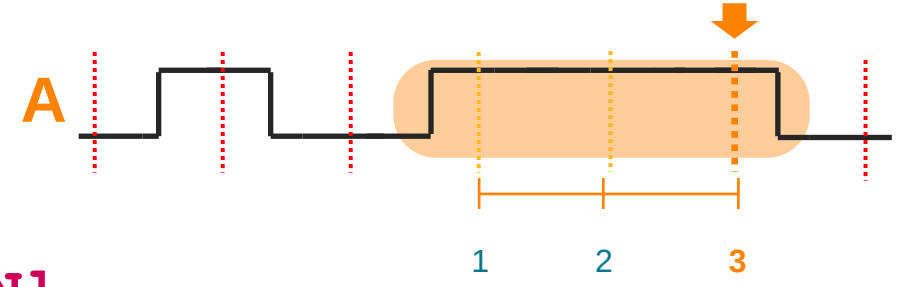


Sequences

- Repetition operator `[*N]` is also sometimes useful:

`A [*3]`

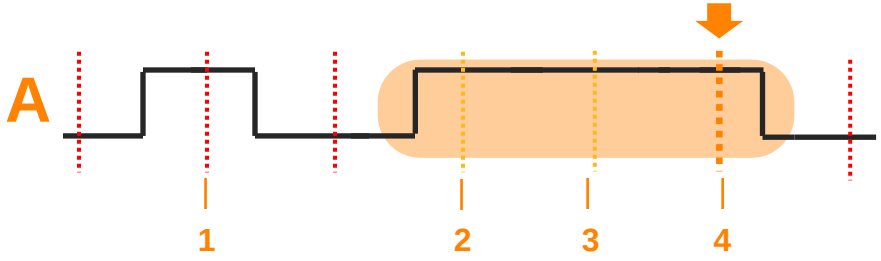
"A happens 3 times in a row



- Occurrence operator `[=N]`

`A [=4]`

"A has occurred 4 times



SVA Example: Sequences

- Sequences can be used in most places where you would write an expression

"Should never see two grants to A in successive cycles"

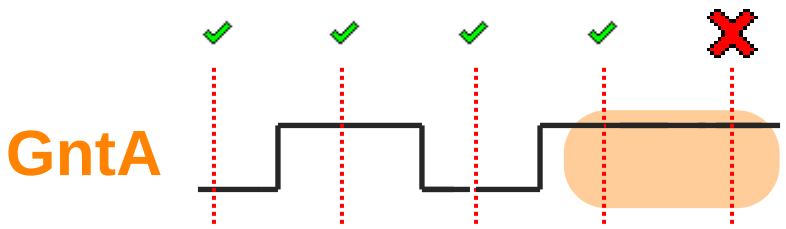
```
GntA_strobe: assert property (
  not (GntA [*2])
);
```

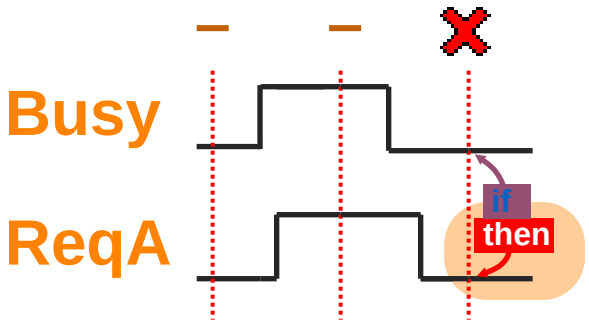
"Busy pulse should only happen if no request"

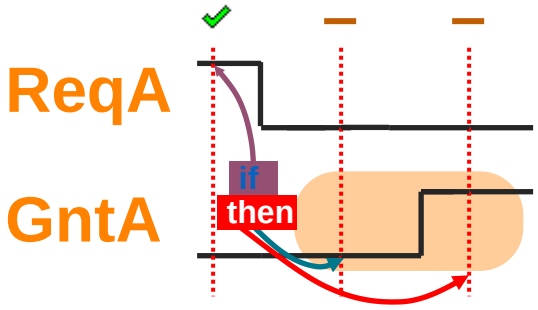
```
Busy_hold: assert property (
  (!Busy ##1 Busy ##1 !Busy)|-> !ReqA
);
```

"Request should be followed by Grant in 1 to 2 cycles"

```
Req_to_Gnt: assert property (
  ReqA |-> ##[1:2] GntA
);
```







Built-In Functions

- Combinatorial

Function	Description	Example

`$onehot`
`$onehot0`
	Returns true if argument has exactly one bit set (one-hot) or at most one bit set (one-hot-zero)	*"At most one grant should be given at a time"*
		`assert property (`
		`  $onehot0({GntA,GntB,GntC});`

`$countones`
`$countzeros`
	Returns the number of ones/zeros in the argument	*"Should never see more than 4 dirty lines"*
		`assert property (`
		`  $countones(Valid & Dirty) <= 4);`

Built-In Functions

- Temporal

Function	Description	Example

`$stable`	Returns true if argument is stable between clock ticks	*"Data must be stable if not ready"*
		`assert property (`
		`  !Ready |=> $stable(Data);`

`$past`	Return previous value of argument	*"If active, then command must not be IDLE"*
		`assert property (`
		`  active |-> $past(cmd) != IDLE);`

`$rose`	Returns true if argument is rising, that is, was low in previous clock cycle and is high on current clock cycle	*"Request must be followed by Valid rising"*
		`assert property (`
		`  Req |=> $rose(Valid);`

`$fell`	Returns true if argument is falling, that is, was high in previous clock cycle and is low on current clock cycle	*"If Done falls, then Ready must be high"*
		`assert property (`
		`  $fell(Done) |-> Ready);`

SVA in Formal

- All you need to know to be successful with SVA:

```
<label>: assert
         cover   property (@(posedge <clock>)
         assume            disable iff (<condition>)
                           <expression|sequence>);
```

Implication
`a |-> b` `a |=> b`

Sequences

`a ##1 b`
`a ##[2:3] b`
`a ##[4:$] b`
`a [*5]`
`a [=4]`

Combinatorial Functions

`$onehot(a)`
`$onehot0(b)`
`$countones(c)`
`$countzeros(d)`

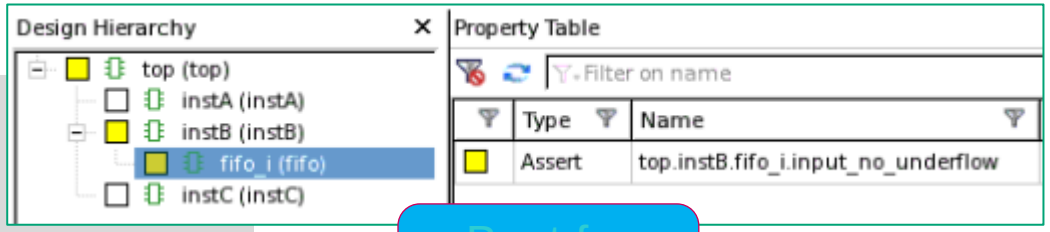
Temporal Functions

`$stable(a)`
`$past(b)`
`$rose(c)`
`$fell(d)`

SVA Mechanisms to Embed Properties

Inline RTL `fifo.v`

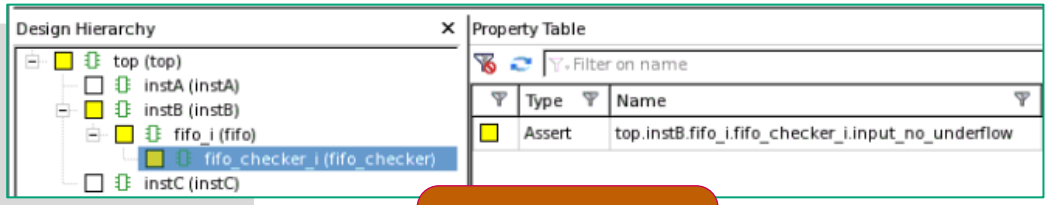
```
module fifo (input clk, rst_n, read, output empty, …)
  // Actual FIFO code:
  …
  `ifdef ASSERTS_ON
    logic ..
    ast_no_underflow: assert property (not(read && empty));
endmodule
```



Best for designers

Use `bind` construct `fifo_bind.sv`

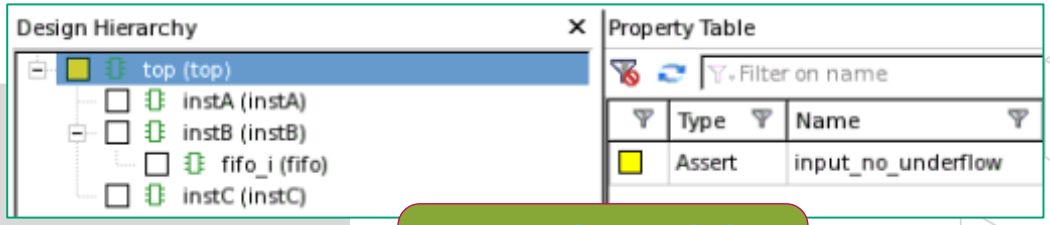
```
module fifo_checker (input clk, rst_n, read, empty);
  // FIFO must not underflow
  ast_no_underflow: assert property (not(read && empty));
endmodule

`ifdef ASSERTS_ON
  bind fifo fifo_checker fifo_checker_inst(.clk(clk), …);
```



Best for DV

Create in TCL `jg_fifo.tcl`

```
analyze ...
elaborate ...
...
assert -name ast_no_underflow {not(instB.fifo_i.read && instB.fifo_i.empty)}
```



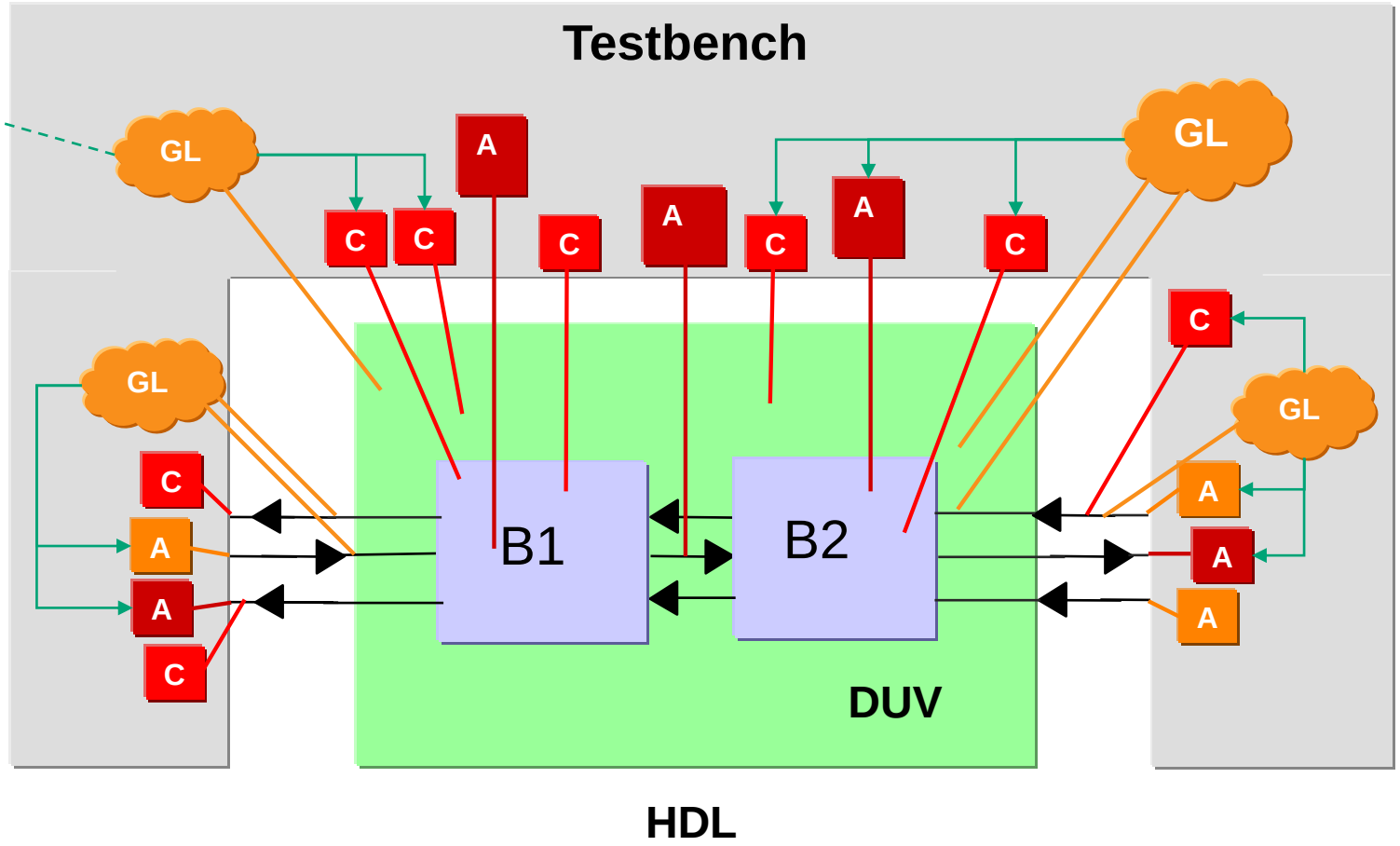
Best for quick experiments

Glue Logic

- When verifying or modeling complex behaviors, introducing auxiliary logic to observe and track events can greatly simplify coding
  - This logic is commonly referred to as “glue logic”
- Once glue logic is in place, expressing SVA properties may be trivial
- Glue logic comes at no extra price
  - Jasper does not care whether property is all SVA or SVA+glue logic
  - Recommendation is to choose based on clarity

Formal Testbench

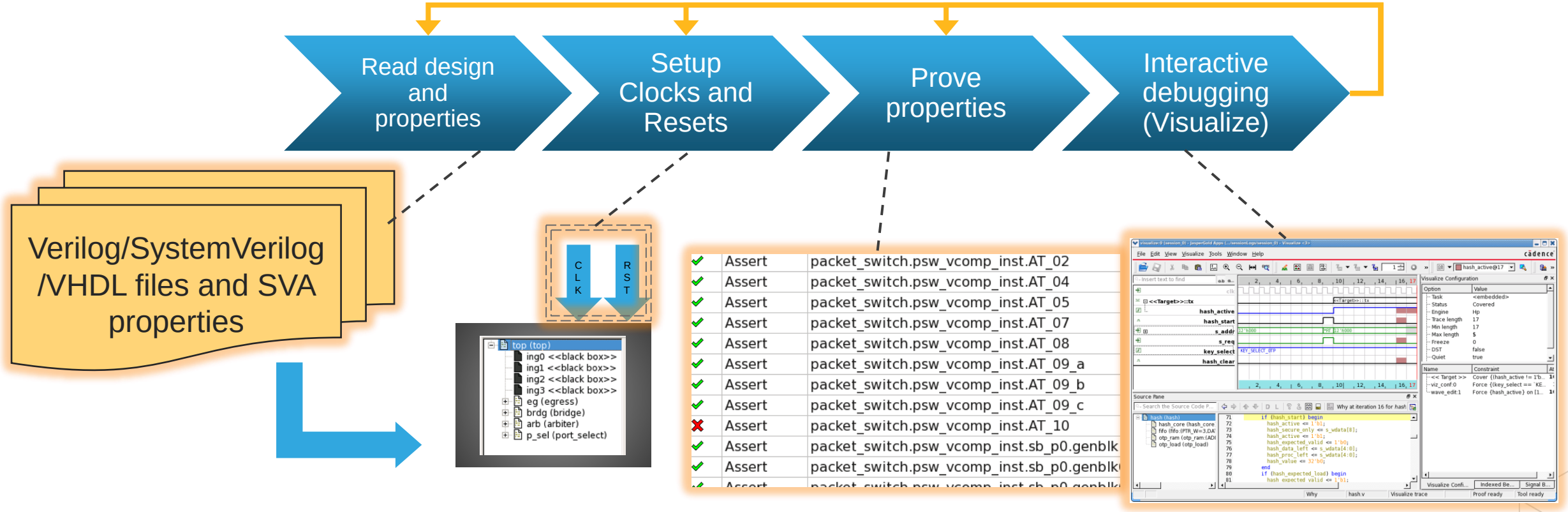
Glue Logic monitors design and feeds properties



Formal Property Verification (FPV) basic

Formal Property Verification (FPV) Flow

- The basic app with the highest flexibility to run formal



Read Design and Property (Compilation)

Design Hierarchy

Message log

Tcl command line interface

Properties loaded from design (SVA), but still not proven

Properties can be created interactively via Tcl

run.tcl (session_0) - JasperGold Apps (.../fpv101/jgproject) - Main

File Edit View Design Reports Application Window Help

Formal Prope...

Design Hierarchy

packet_switch (packet_switch)
psw_vcomp_inst (psw_vcomp)
sb_p0 (jasper_scoreboard_3:(CHUNK_WIDTH=32'b01010,MAX_.
sb_p1 (jasper_scoreboard_3:(CHUNK_WIDTH=32'b01010,MAX_.

Property Table

Type	Name
Cover	packet_switch.psw_vcomp_inst.sb_p0.genblk6.core.g...
Cover	packet_switch.psw_vcomp_inst.sb_p0.genblk6.core.g
Cover	packet_switch.psw_vcomp_inst.sb_p0.genblk6.core.g
Cover	packet_switch.psw_vcomp_inst.sb_p0.genblk6.core.g
Assert	packet_switch.psw_vcomp_inst.sb_p0.genblk6.core.g
Cover (re...	packet_switch.psw_vcomp_inst.sb_p0.genblk6.g
Assert	packet_switch.psw_vcomp_inst.sb_p0.genblk6.core.g
Cover (re...	packet_switch.psw_vcomp_inst.sb_p0.genblk6.core.g
Assert	packet_switch.psw_vcomp_inst.sb_p0.genblk6.cor
Cover (re...	packet_switch.psw_vcomp_inst.sb_p0.genblk6.cor
Assert	packet_switch.psw_vcomp_inst.sb_p0.genblk6.cor

Total: 114 Filtered: 114 Selected: 0 Validity: 0:0:87:27 Run: 1

session_0

```
[WARN (VDB-1013)] vcomp.sv(47): input port 'incoming_clk' is not connected on this instance
[WARN (VDB-1013)] vcomp.sv(72): input port 'incoming_clk' is not connected on this instance
WARNING (WOBS002): vcomp.sv(316): Weakly embedded unbounded sequence in a ASSERT directive.
WARNING (WOBS002): vcomp.sv(337): Weakly embedded unbounded sequence in a ASSERT directive.
INFO (INL003): Clearing all state information (assumes, stopats, and so forth).
packet_switch
```

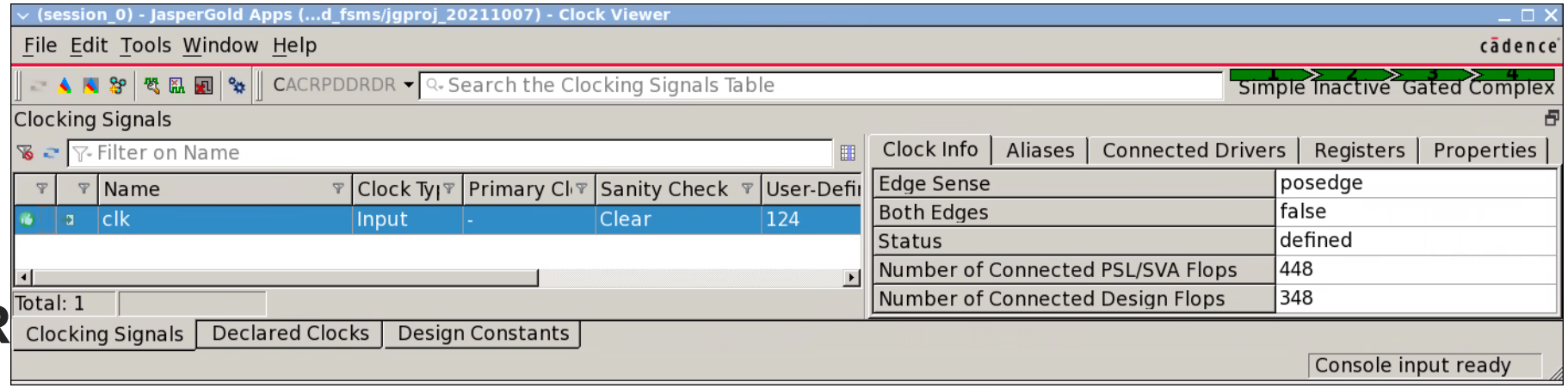
```
[<embedded>] %
```

Console | Lint Messages | Warnings / Errors | Proof Messages

No proofs running | Console input ready

Setup and Running the Proof

- **Clock/Reset:** Specify clocks and resets with the help from Clock Viewer and Reset Analysis



Clock Viewer

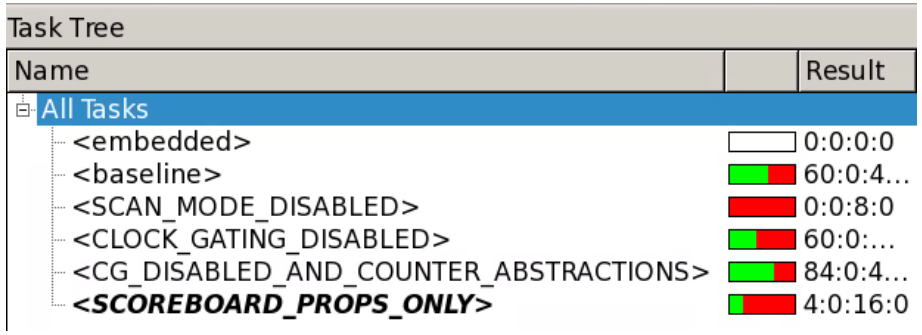
Reset Analysis

- **R**

ProofGrid Manager

Proof Settings

Run “Prove”



Proof Results

By double-clicking a failed assertion or a covered property, we can see the trace waveform.

DEBUG!

Assertion passed

Assertion failed

Cover unreacheable

Assumption

Property covered

Property Table

No filte… filter on name

	Type	Name	Engine	Bound	Time	Task	Traces
✓	Cover (related)	packet_switch.psw_vcomp_inst.AT_09_b:p…	N	2 - 4	0.0	<embedded>	1
✓	Assert	packet_switch.psw_vcomp_inst.AT_09_c	N (13)	Infinite	0.1	<embedded>	0
✓	Cover (related)	packet_switch.psw_vcomp_inst.AT_09_c:p…	N	2 - 4	0.0	<embedded>	1
✗	Assert	packet_switch.psw_vcomp_inst.AT_10	Ht	4	0.1	<embedded>	1
✓	Cover (related)	packet_switch.psw_vcomp_inst.AT_10:pre…	Ht	4	0.1	<embedded>	1
✗	Cover	packet_switch.psw_vcomp_inst.CV_07	N (15)	Infinite	0.0	<embedded>	0
✓	Cover	packet_switch.psw_vcomp_inst.CV_08	N	5	0.0	<embedded>	1
✗	Cover	packet_switch.psw_vcomp_inst.CV_09	N (15)	Infinite	0.0	<embedded>	0
✗	Cover	packet_switch.psw_vcomp_inst.CV_10	N (15)	Infinite	0.0	<embedded>	0
●	Assume (live)	packet_switch.psw_vcomp_inst._assume_1	?		0.0	<embedded>	0
✓	Cover (related)	packet_switch.psw_vcomp_inst._assume_…	N	1	0.1	<embedded>	1
✓	Assert	packet_switch.psw_vcomp_inst.sb_p0.gen…	PRE	Infinite	0.0	<embedded>	0
✓	Assert	packet_switch.psw_vcomp_inst.sb_p0.gen…	N (15)	Infinite	0.4	<embedded>	0
?	Assert	packet_switch.psw_vcomp_inst.sb_p0.gen…	Ht	9 -	0.5	<embedded>	0
✓	Assert (live)	packet_switch.psw_vcomp_inst.sb_p0.gen…	N (16)	Infinite	0.6	<embedded>	0
✓		ket_switch.psw_vcomp_inst.sb_p0.c		1	0.2	<embedded>	1

Total: 11 | Selected: 1 | Validity: 63:4:18:27 | Run: 28:0:0:8

Bounded Proof…

… up to cycle 9

cadence

Visualize Interactive UI Key Features

Source: www.deepchip.com

"Jasper Visualize is an incredible debug tool. We use it for debugging, finding root causes, and exploring."

What-If

Minimum trace length

Highlight Relevant Logic

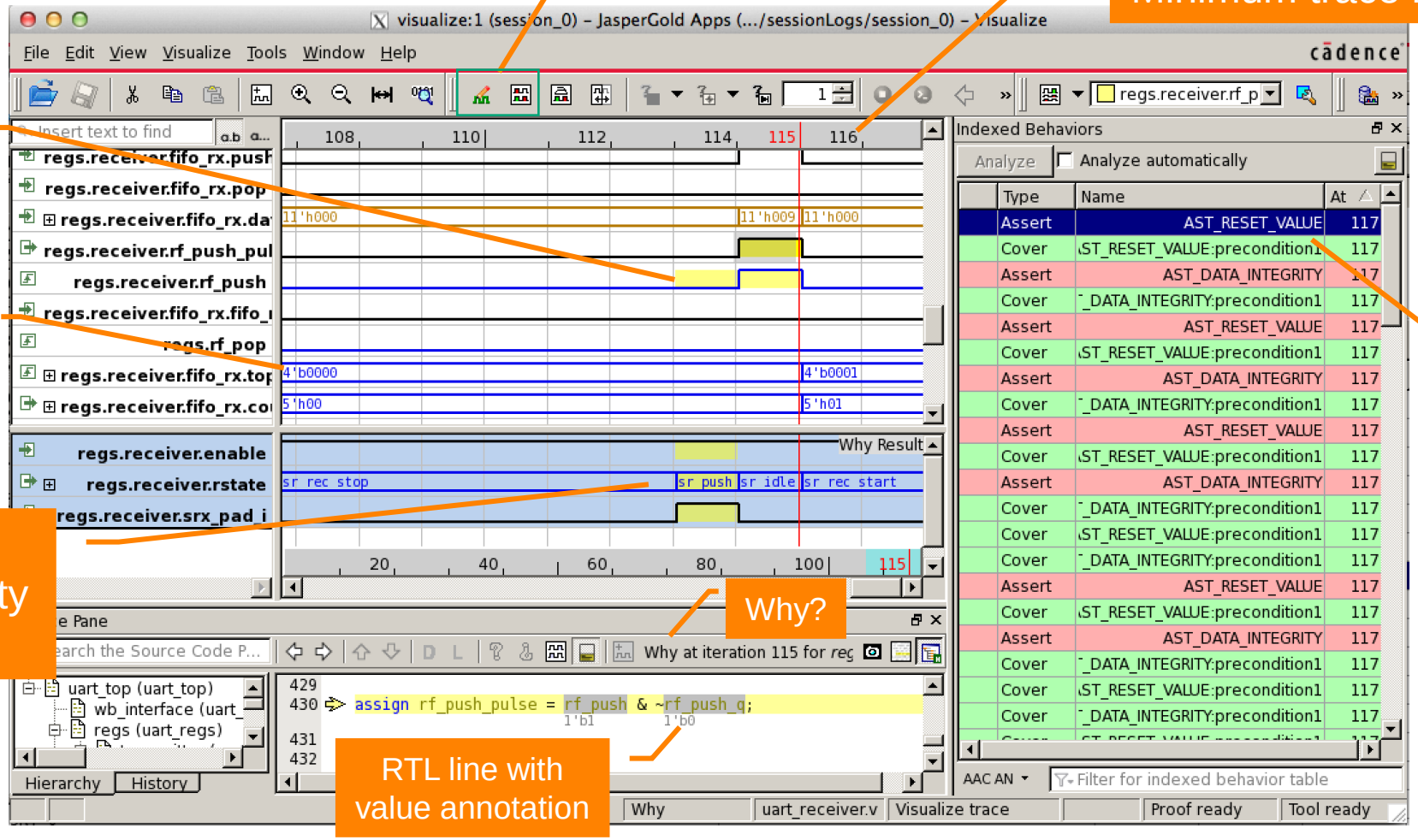
QuietTrace

Property that fails earliest

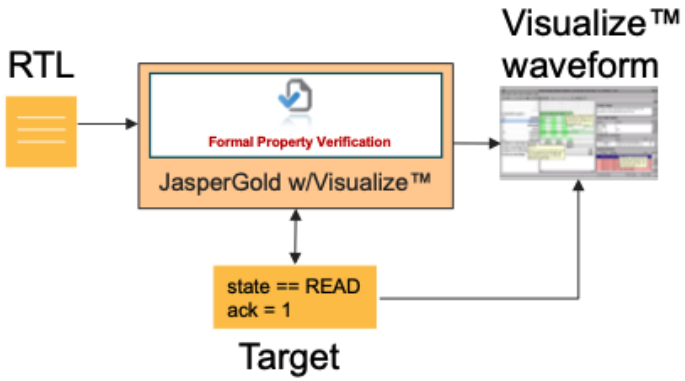
Preview with values for property or why results

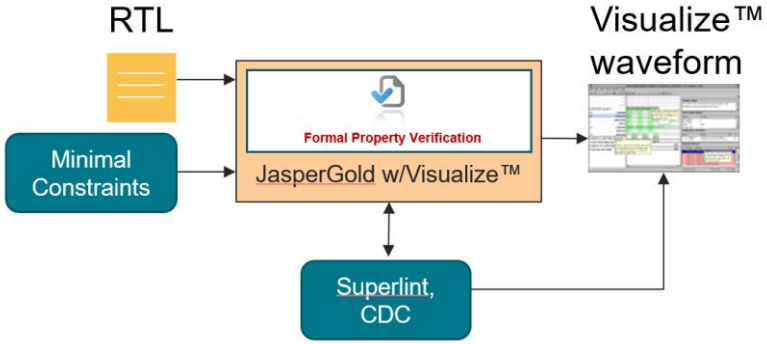
Why?

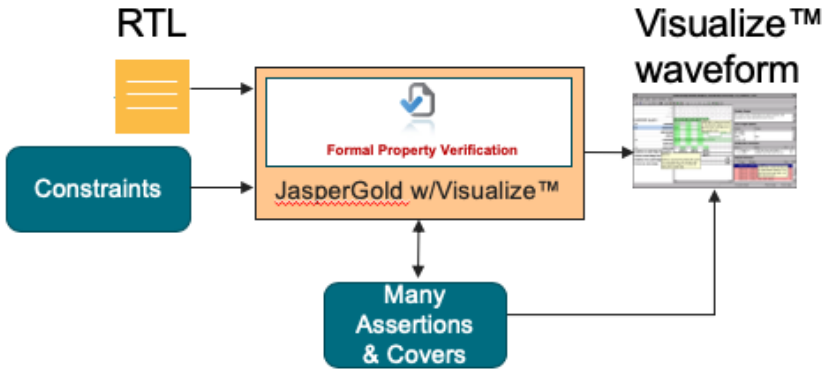
RTL line with value annotation



Jasper Usage for Designers vs. Verification Team







Design Exploration using Visualize™

- ✓ **Goal: Automatically generate waveforms for expected behaviors to catch bugs**
- ✓ Specify the target and let the formal engines generate the stimulus
- ✓ Interactively modify waveforms

Automatic Formal Checks

- ✓ **Goal: Remove bugs that can be automatically detected**
- ✓ Automatic checks from Superlint, CDC, XPROP
- ✓ (Optional) Leverage Formal VIP for interface constraints & checks

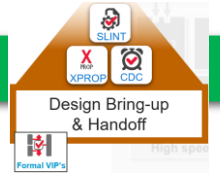
Designer Formal Analysis

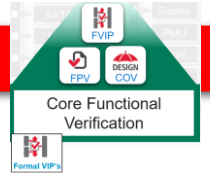
- ✓ **Goal: Robust unit-level analysis without any unit-level simulation!**
- ✓ ***Adds User SVA for functional constraints & checks***

Formal Verification

- ✓ **Goal: Full Proofs, Bounded Proofs and Deep Bug Hunting**
- ✓ May integrate multiple blocks
- ✓ Potentially complex Constraints required
- ✓ Exhaustively exercise design states
- ✓ Optionally measure coverage for sign-off
- ✓ Use Visualize for interactive debug

Design Bring-Up & Handoff





cadence

Mini FPV Demo Case

Jasper Formal Coverage Analysis

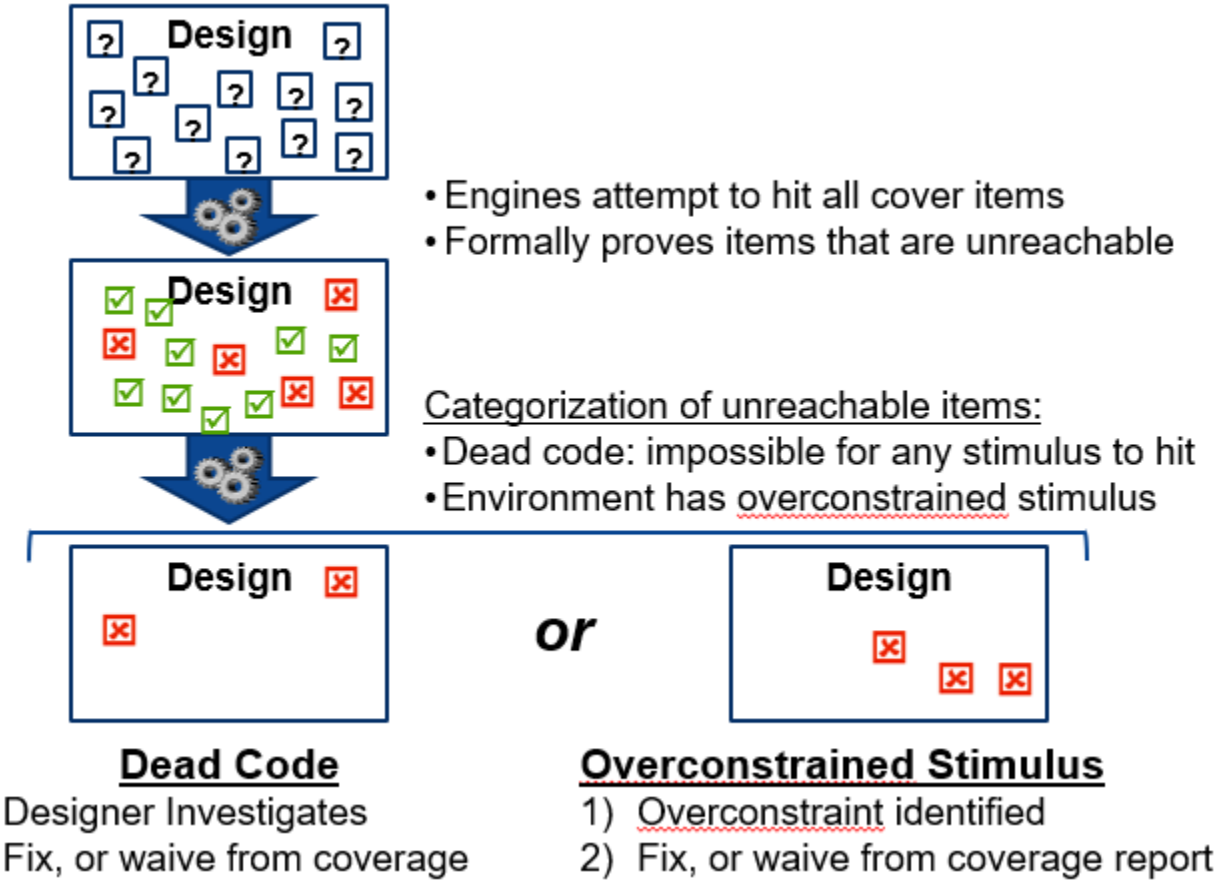
Measuring Coverage

- **Stimuli Coverage:** What code or functionality is reachable by the formal testbench?
  - Collected during classic formal or bug-hunting
  - Gives confidence that formal testbench is able to exercise all behavior that could yield a bug
- **Checker Coverage:** Is my formal checking complete?
  - Determines how much of the design is checked by assertions
  - Gives confidence that formal checking is complete enough to detect a bug
- **Formal Coverage:** Consolidation of Stimuli and Checker Coverage results
  - Cover item is marked “covered” if ***both*** its Stimuli and Checker results are “covered”
  - Provides a single-metric view of formal verification coverage

Stimuli Coverage

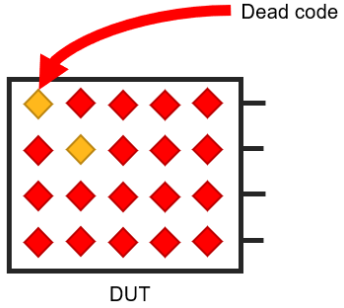
What code or functionality is reachable by the formal testbench?

- Formal engines attempt to find the stimuli necessary to “hit” a cover
- Result is **Covered**, Uncovered, **Unreachable**, or **Deadcode**

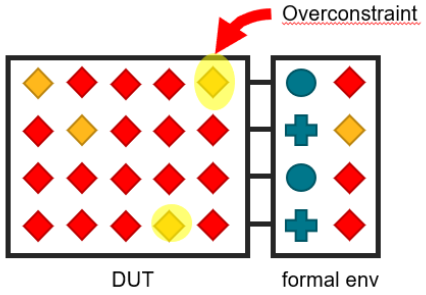


Stimuli Coverage – **Unreachable** vs. **Deadcode**

- **Deadcode** status
  - Formal engines have determined cover is unreachable **without** constraints (assumes) applied
  - Any covers initially found **unreachable** are automatically re-run with constraints disabled to determine if they are **deadcode**

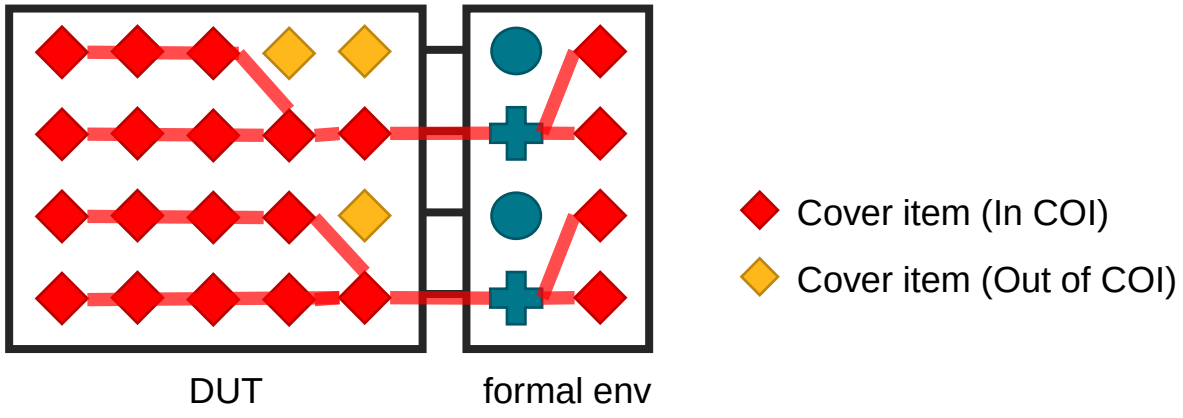


- **Overconstraint** status
  - Formal engines have determined cover is unreachable **with** constraints (assumes) applied



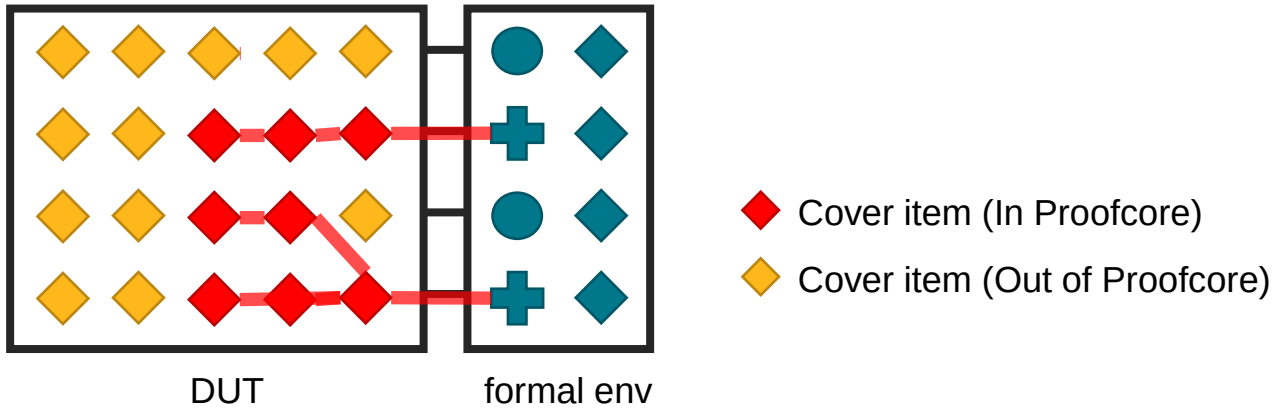
Checker Coverage Type – Cone-Of-Influence Measurement

- Determines the cover items in the COI of **each assertion**
- Finds the union of the assertion COIs
- The remaining Out of COI cover items indicate holes in the assertion set – code that is not checked by any asserts



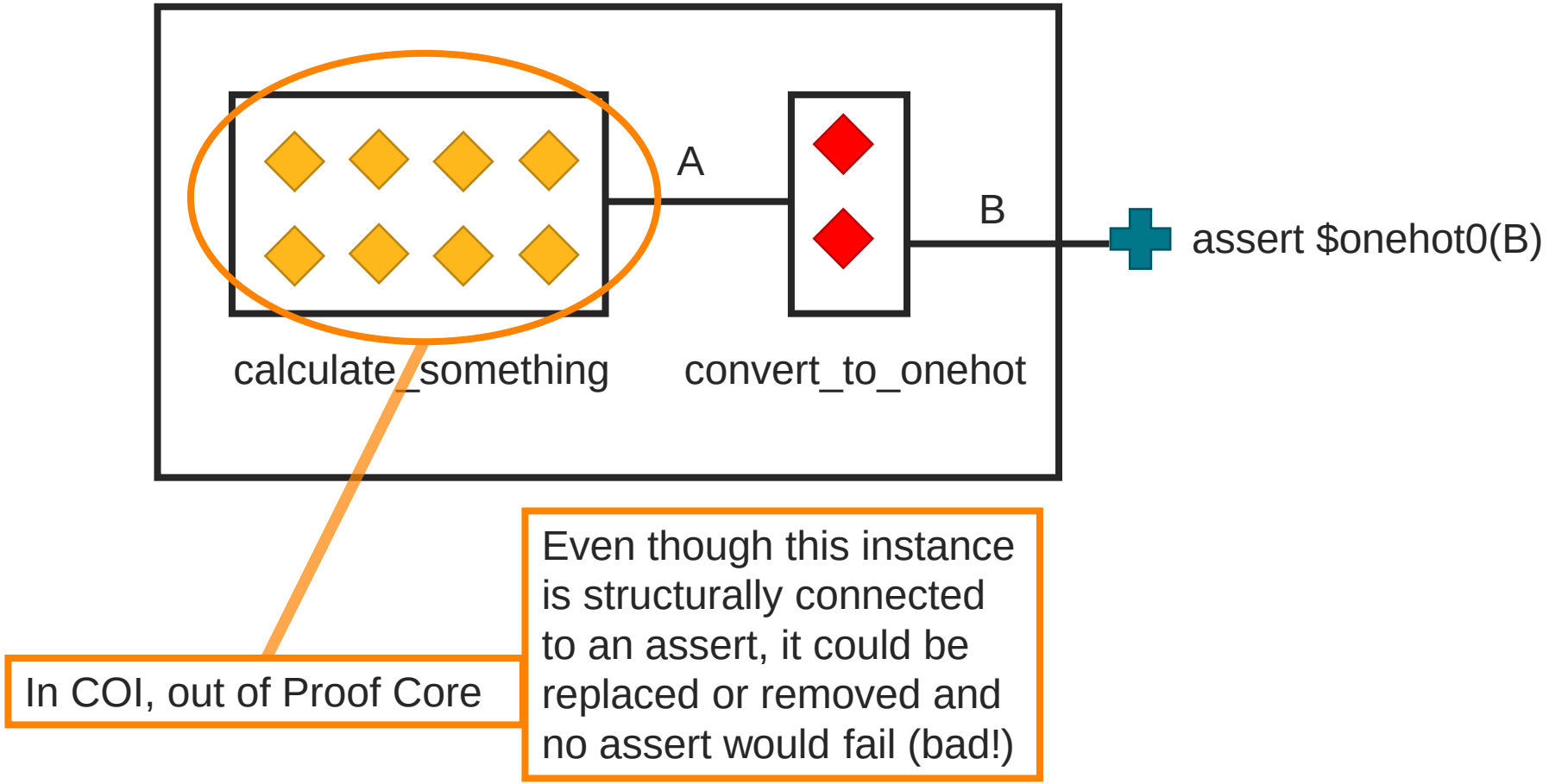
Checker Coverage Type – Proof Core Measurement

- Represents the portion of the design verified by formal engines
- Subset of the COI – COI represents the maximum potential of proof coverage
- Engines abstract a portion of the design during the proof process, iteratively consider a larger portion of the design until a proof, or bounded proof, is established
- Anything outside the “proof core” was unnecessary for proof, therefore not being checked
- **Key metric for showing formal verification progress**



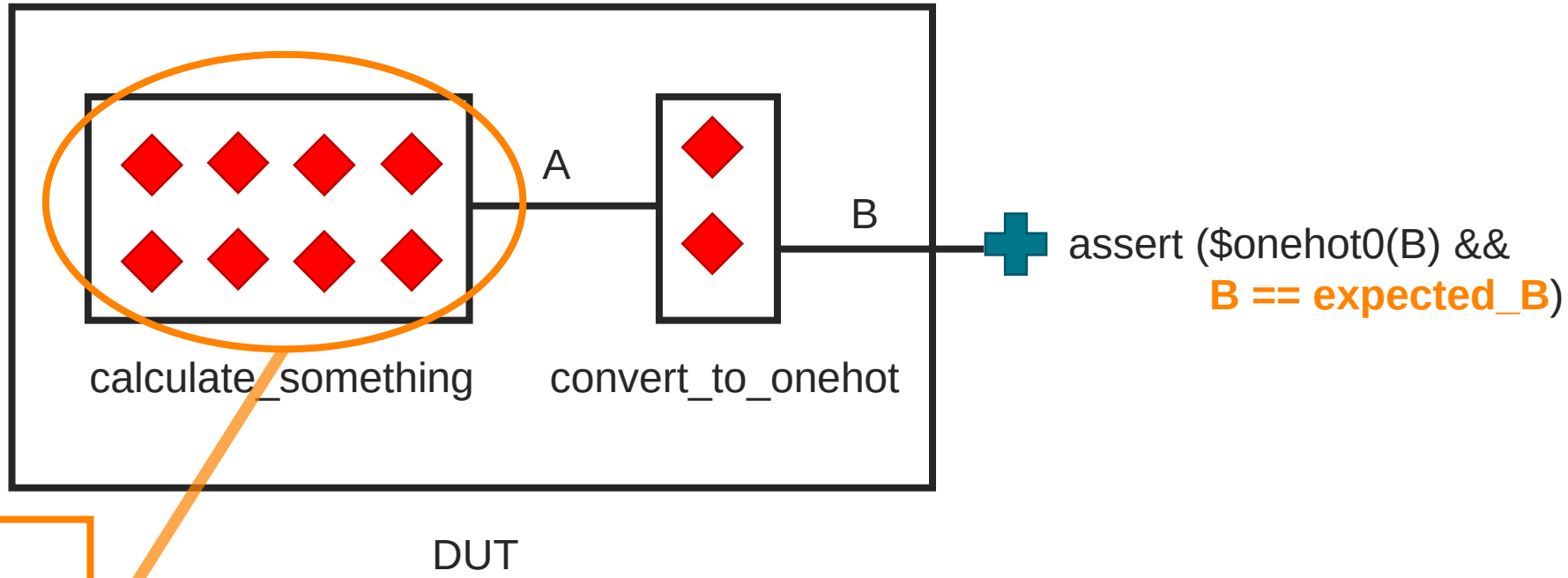
Proof Coverage

- How to interpret the result



Proof Coverage

- How to interpret the result



Adding checks will increase proof coverage, ensuring that the design's functionality is being verified (good!)

Coverage Analysis – Summary View

File Edit View Design Application Window Help

Coverage

Coverage Analysis | Assert Analysis | Waiver Table | Console

Coverage Analysis

Tasks | <embedded> | Excludes | Waived, Reset | Checker | Proof Core

Coverage Models:

Instance	Ex.	Formal Coverage	Stimuli Coverage	Checker Coverage
pp (pp:(DATA_MEM_SIZE=8))		276/624 (44.23%)	583/624 (93.43%)	274/622 (44.05%)
ig_parse (pp_ig_parse)		31/71 (43.66%)	64/71 (90.14%)	31/71 (43.66%)
route_ctrl (pp_route_ctrl:(PTR_W=3))		56/118 (47.46%)	113/118 (95.76%)	56/118 (47.46%)
payload_mem (pp_payload_mem:(PTR_W=3))		23/75 (30.67%)	70/75 (93.33%)	22/74 (29.73%)
thr_mem (pp_thr_mem:(PTR_W=3))		18/78 (23.08%)	71/78 (91.03%)	17/77 (22.08%)
stall (pp_stall:(PTR_W=3))		28/33 (84.85%)	30/33 (90.91%)	28/33 (84.85%)
drop_ctrl (pp_drop_ctrl:(PTR_W=3))		16/29 (55.17%)	26/29 (89.66%)	16/29 (55.17%)
list_ctrl (pp_list_ctrl:(PTR_W=3))		34/41 (82.93%)	37/41 (90.24%)	34/41 (82.93%)
cfg (pp_cfg:(PTR_W=3))		13/92 (14.13%)	89/92 (96.74%)	13/92 (14.13%)
intr_ctrl (pp_intr:(PTR_W=3))		57/87 (65.52%)	83/87 (95.40%)	57/87 (65.52%)
fv_env (fv_env:(DATA_MEM_SIZE=8,DYNAMIC_SCB=0,MAX_PKT_LEN=6)		0/0 (0.00%)	0/0 (0.00%)	0/0 (0.00%)
Packages				
fv_pkg (fv_pkg)		0/0 (0.00%)	0/0 (0.00%)	0/0 (0.00%)
Compilation Units		0/0 (0.00%)	0/0 (0.00%)	0/0 (0.00%)

Summary | Detailed

proofs running | Console

Exclusion Indicator

Formal coverage – combines stimuli and checker

Stimuli coverage- formal testbench correctness

Checker coverage – assertion completeness

Mini COV Demo Case

cādence®