Internal Use Only

Formal Verification with Cadence Jasper

Hugh Lo

May. 2023

cadence®

Formal Verification Introduction

What is Formal Verification?

Formal Verification	=	Mathematical analysis of behaviors in the **design state space**

- Checks exhaustively if a model meets a given spec
  - Model ➔ synthesizable RTL
  - Spec ➔ properties (**assertions, covers, and assumes**)

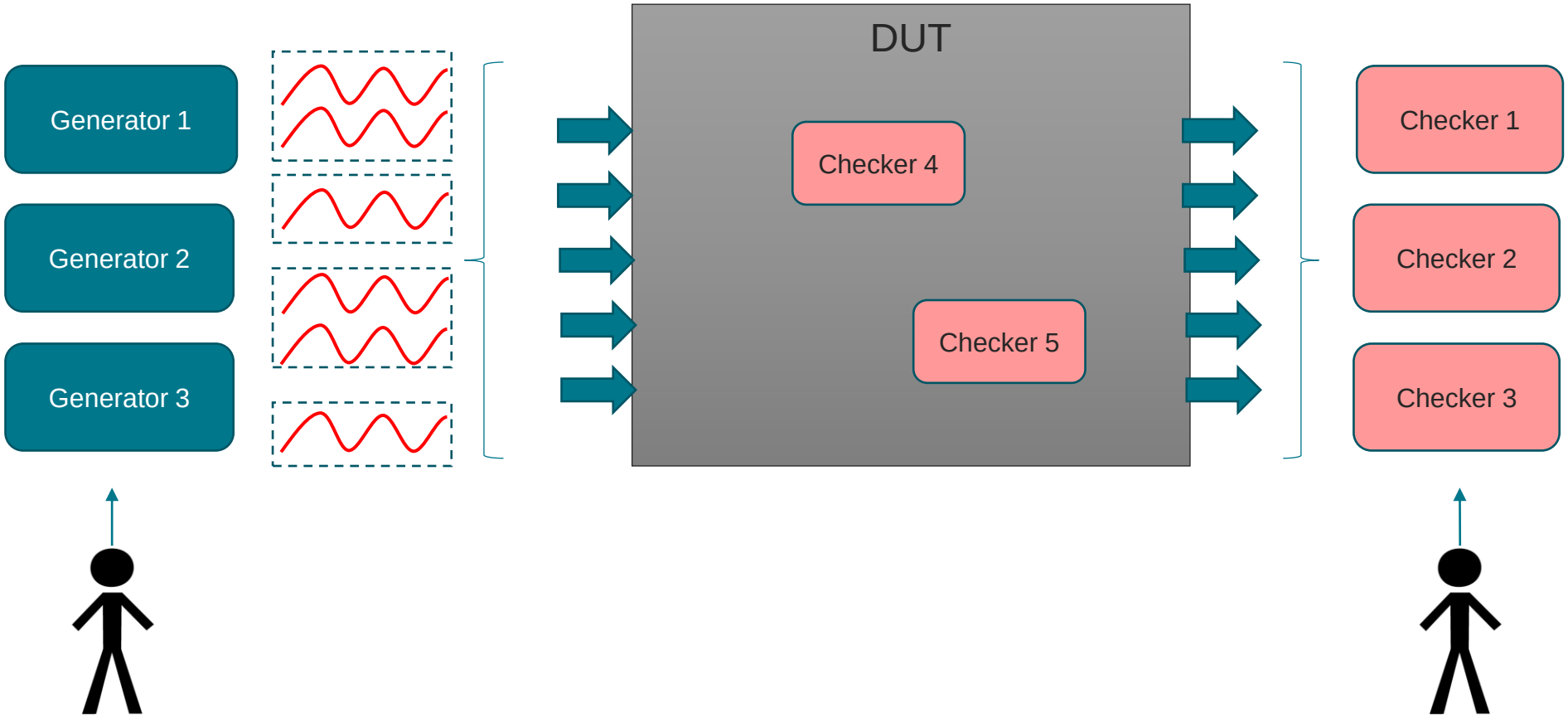
- Key differences between **simulation** and **formal**

	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

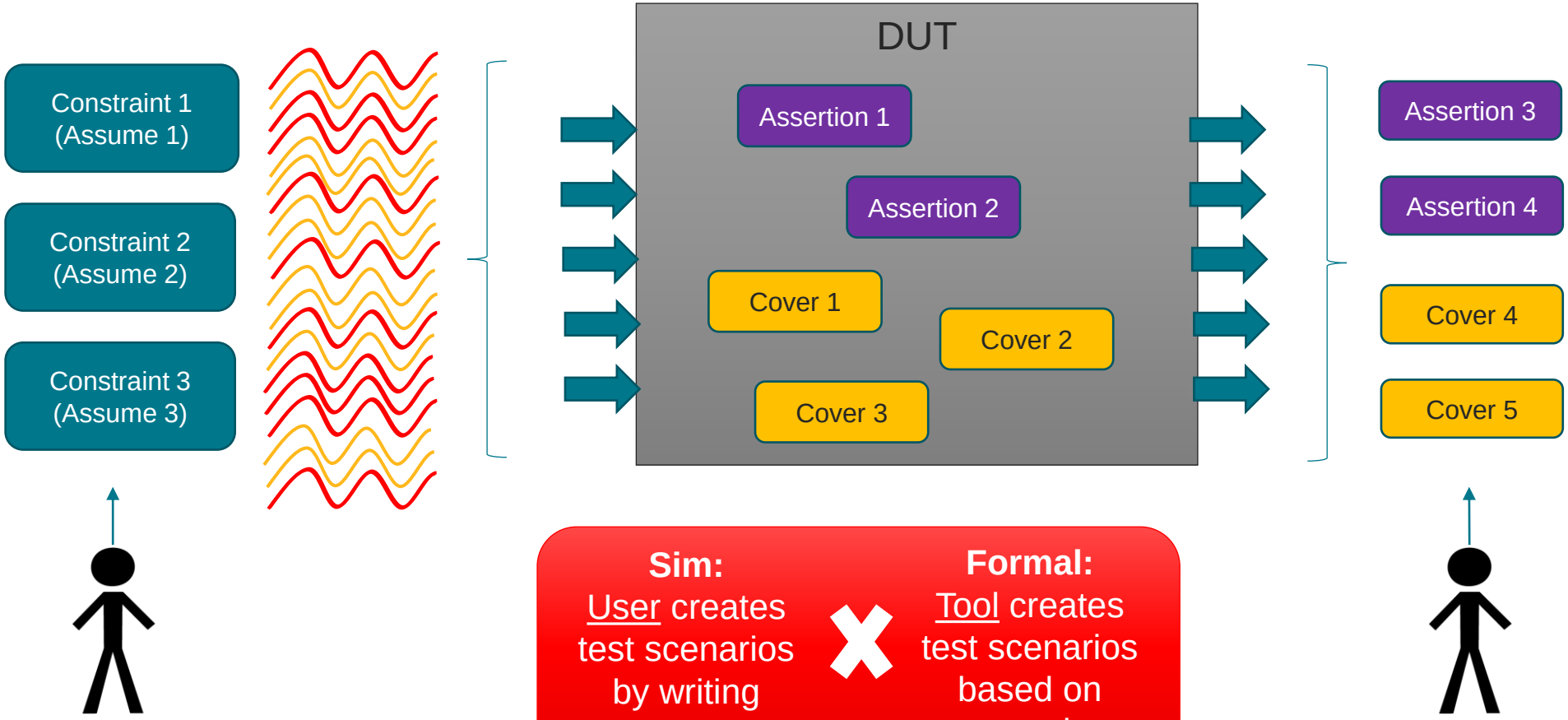
DD/DVs create checkers to observe design and flag for errors



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

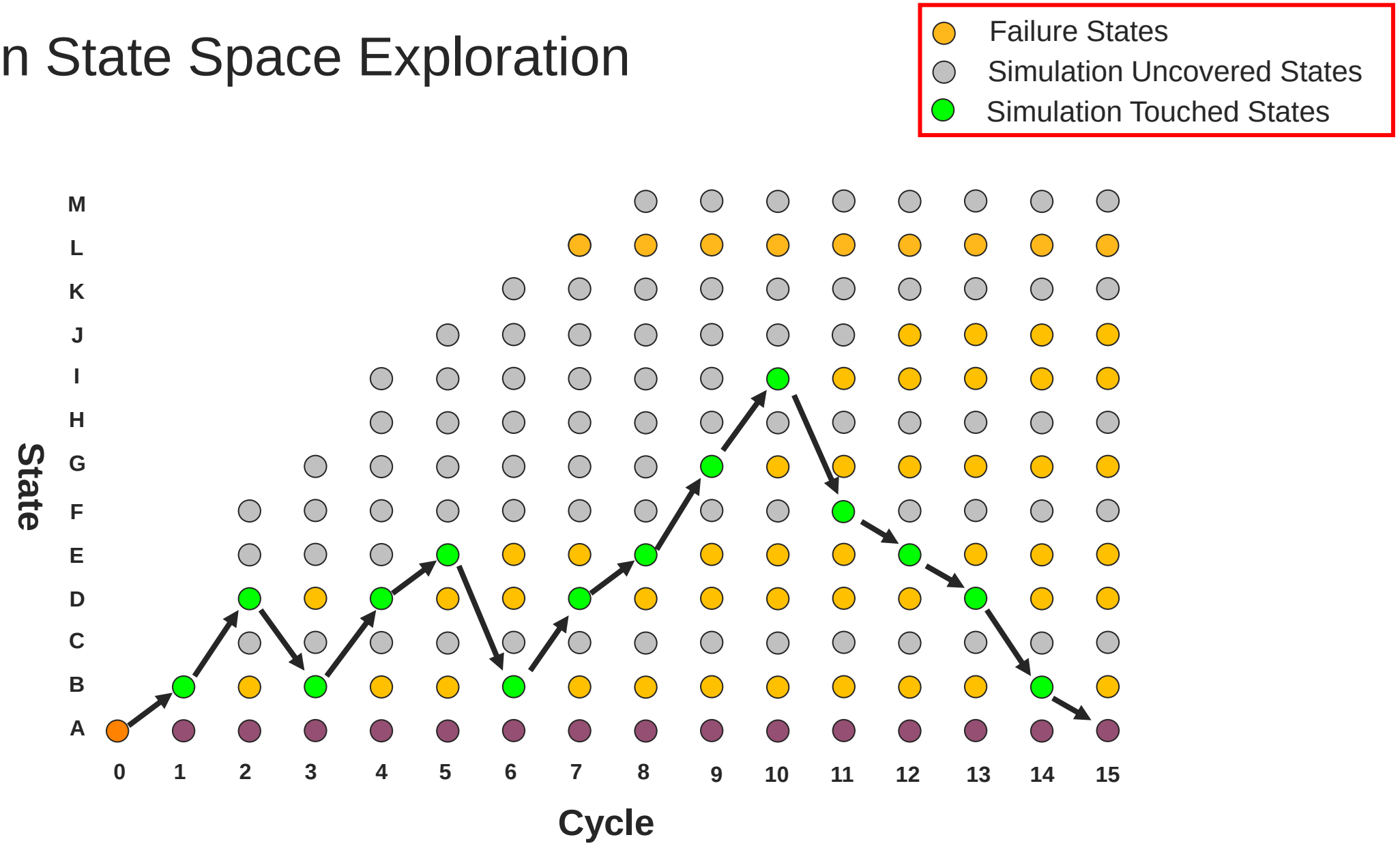
Formal:
Tool creates test scenarios based on properties

Formal vs. Simulation on Design State Space

- State = Values of all registers/inputs
- Design State Space = The set of possible states of DUT
  - As cycle increases, the space grows very fast

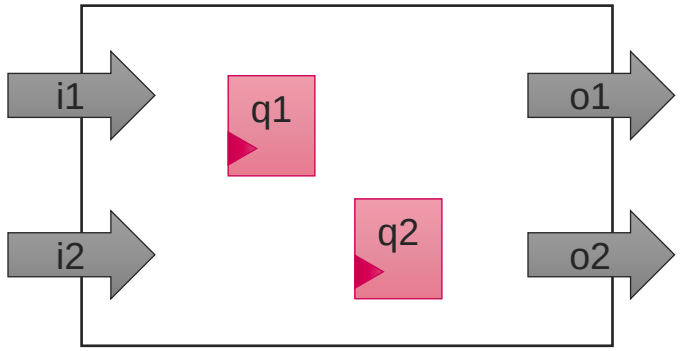
- Simulation cannot check whole space by limited input stimulus
  - The purpose of constrained random is to achieve good diversity while exploring the space
- Formal checks whole space by it's mathematical intrinsicality
  - It's what the word "Exhaustive" means

Design State Space Exploration



Assertion / Cover Property

- Formal checks properties in design state space
  - Report when assert properties are violated
  - Report when cover properties are hit
- Formal check == Mathematically Prove == Exhaustively Search




```
cover (q2 ##2 o1)
```

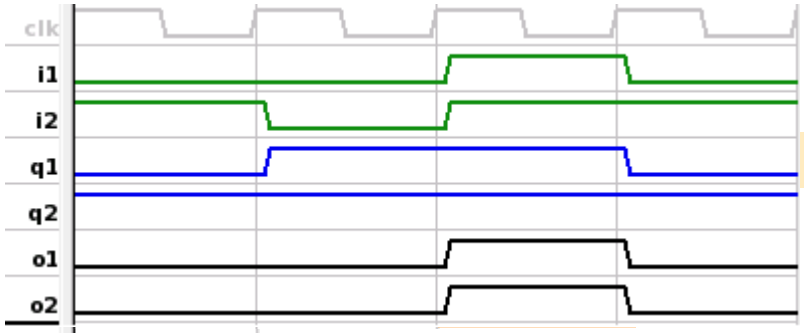
Waveform saved in database

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

Waveform saved in database

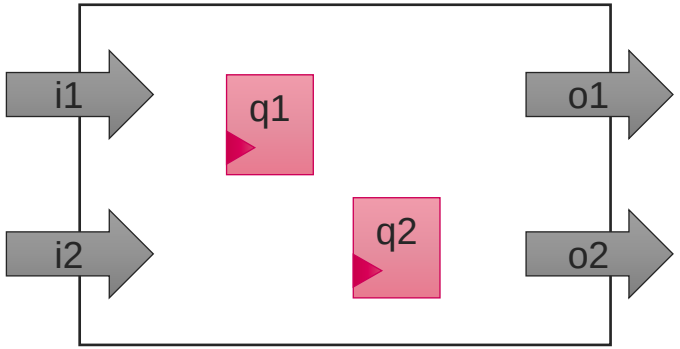
Found scenario that hits cover!

Found scenario that violates assertion!



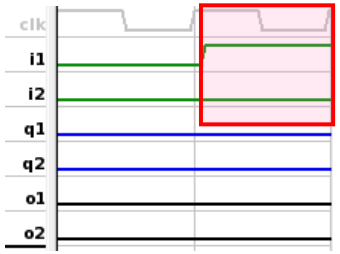
Assume Property (Constraint)

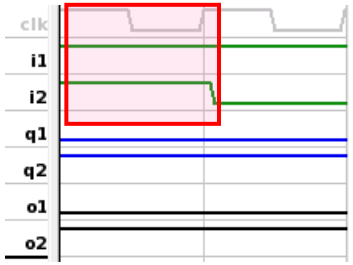
- Not all input stimulus combinations are meaningful to us
- Assumes (Constraints) tell formal what is **legal**
  - Only scenarios where all assumptions are true will be considered
  - Formal analysis only applies on the **reduced design state space**




```
cover (q2 ##2 o1)
assert (o1 && o2 |=> q1)
assume (i1 |-> !i2)
```

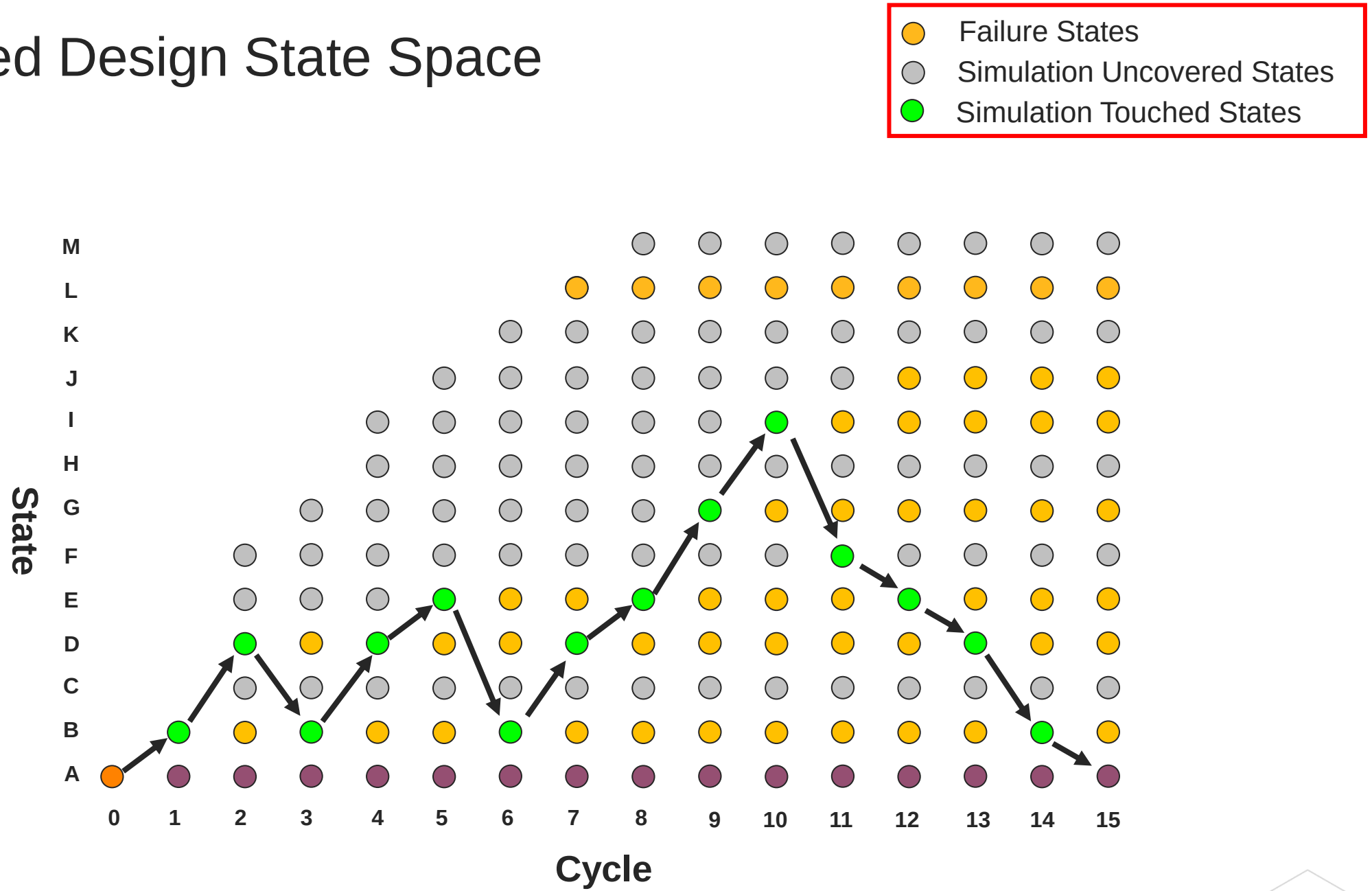
Found scenario that violates assumption!





Discard this scenario From search space

Reduced Design State Space



Proof Results

Assert : Proven
Cover : Unreachable

	Type	Bound
✓	Assert	Infinite
✗	Cover	Infinite

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

Undetermined

	Type	Bound
?	Assert	7 -
?	Cover	7 -

- Formal analyzed a subset of all reachable states
- Assertion does not fail in these states
- Cover is not hit in these states

Assert : CEX (Counter Example)
Cover : Covered (Hit)

	Type	Bound
✗	Assert	12
✓	Cover	49

- Formal found a state that hits a property
- Assertion failure
- Cover hit
- **Waveform (Trace) is available**

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

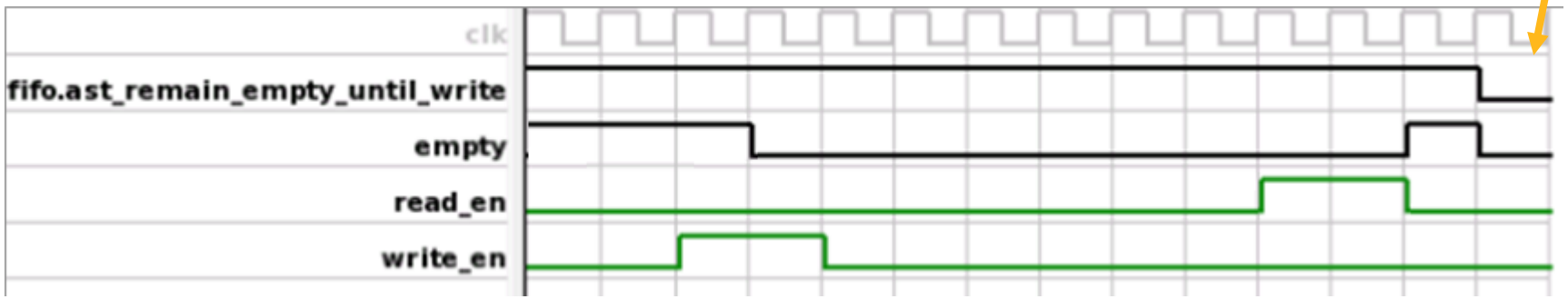
Full Proof: Impossible to violate this assertion

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 of Formal

- More comprehensive than simulation
  - Provides a definitive answer: “This assertion can never fail”
  - Exhaustively explore design states rather than randomly ‘stumbling’ into a bug
  - Breadth-first search can hit interesting states more efficiently
- Simpler debug than simulation
  - Shorter traces make debugging easier
  - Powerful root-cause analysis capabilities in Jasper (Visualize ‘Why’, Quiet-trace)
- Leads to higher quality
  - Find bugs from a different angle: breadth-first search (formal) vs. depth-first search (sim)
  - Often reveals corner case bugs that simulation would never catch
  - Ability to analyze coverage, including finding gaps in your Assertion checking
- Improves productivity and schedule
  - Jasper apps and methodologies can allow bugs to be exposed much sooner (even during design)
  - Time to find the ‘first real bug’ and ‘first corner case bug’ will be greatly reduced with formal
  - High level of re-use. Properties created can be used by other teams/methodologies

Challenges of Formal

- Requires a different mindset than simulation
  - Complexity issue
  - Overconstraint / Underconstraint
  - Bounded proof
- Metrics are just as important as simulation
  - Establish a set of coverage/sign-off goals that clearly map to your verification plan
- May require learning some formal techniques
  - Expressing SVA in a formal-friendly way
  - Choose suitable App for minimizing effort
  - Initialize DUT correctly
  - Reduction or Abstraction

Where can I learn more?

http://support.cadence.com/Jasper_apps

Your Local Jasper AE

FREE Online training courses:

Formal Fundamentals and **Jasper Formal Expert** is just one of the many **FREE** online courses available to Cadence customers.

- Recorded classes with instructors
- Self-paced learning
- LABs and exercise resolution

Go to: https://support.cadence.com/

For Blended/Virtual courses, reach out to us at www.cadence.com/training > Contact Training

Introduction to SVA

Agenda

- Goal: Learn basics of SystemVerilog Assertions (SVA), focusing on usage for formal

- Topics
  - Guidelines for being successful with SVA
  - Using “glue logic” to build complex SVA checkers

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

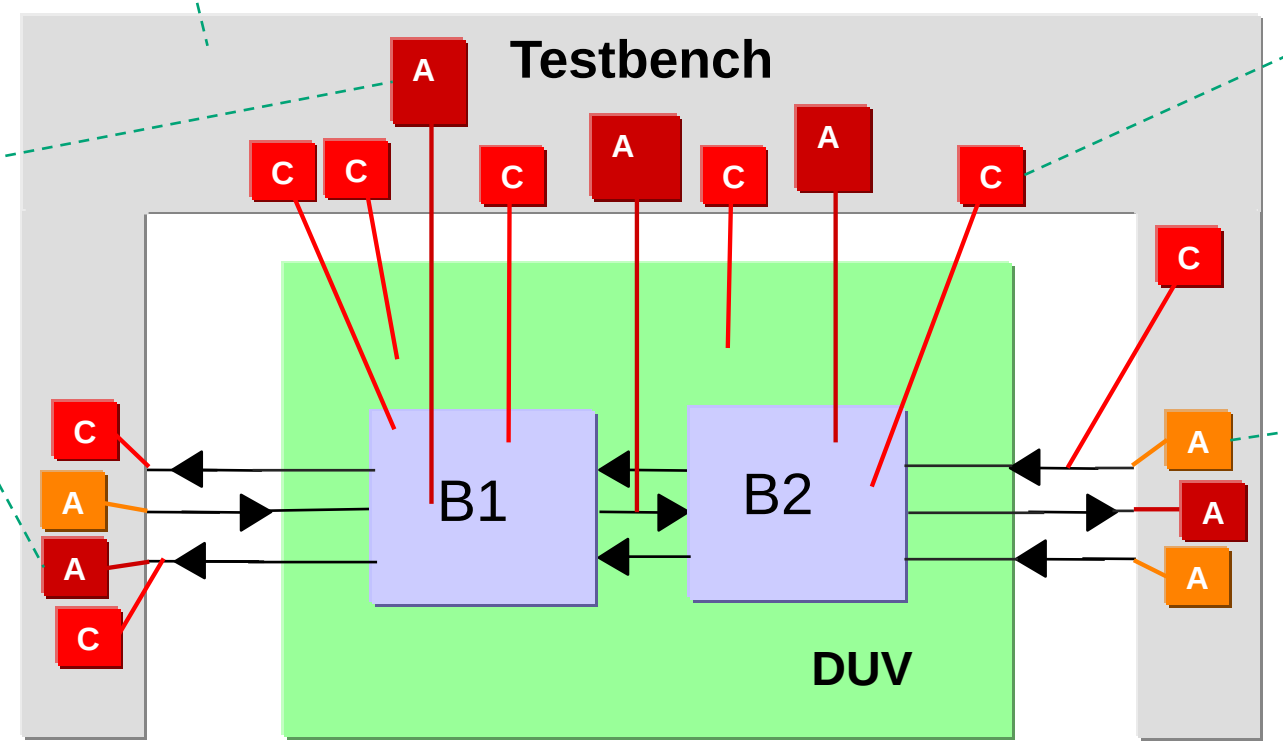
Properties described in **SVA syntax**

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

Covers can target any signal

Asserts check outputs and internals

Assumes constrain inputs



HDL

SVA Syntax

- SVA properties are embedded in Verilog or SystemVerilog code

Property Label: The user-defined name of the property. This name will be shown in the Property Table

Verification Directive: Instructs the compiler that this property is an assertion, i.e. you expect the expression to be true at all times

```
output_no_underflow: assert property (
```

Clocking: Defines when the property is evaluated or sampled. Optional

```
@(posedge clk)
```

```
disable iff (!rstn)
```

Disable Condition: When to ignore/abort evaluation of the property. Optional

```
!(read && empty)
```

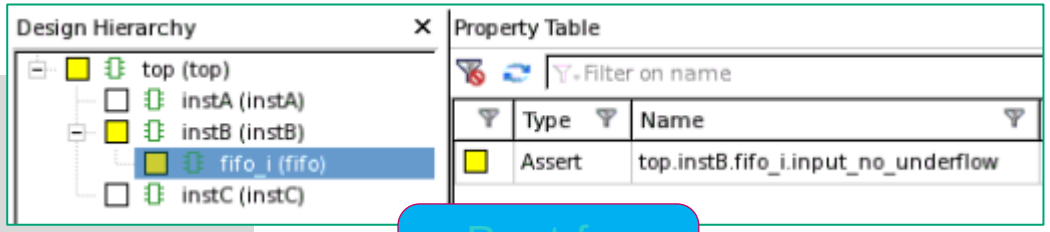
```
);
```

Property Expression: In the case of an assertion, this is what you expect to be true at all times. In this example, the assertion would fire if `read` and `empty` are high at the same time, which would make `!(read && empty)` evaluate to false

SVA Mechanisms to Embed Properties

Inline RTL `fifo.v`

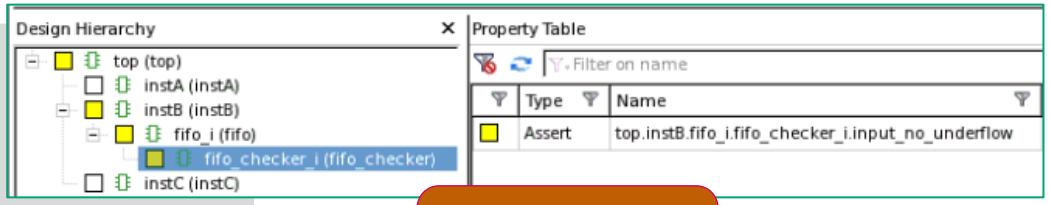
```
module fifo (input clk, rst_n, read, output empty, …)
  // Actual FIFO code:
  …
  `ifdef ASSERTS_ON
    logic ..
    ast_no_underflow: assert property (not(read && empty));
endmodule
```



Best for designers

Use `bind` construct `fifo_bind.sv`

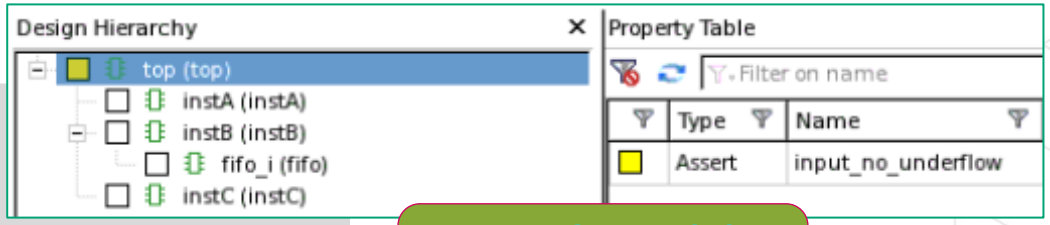
```
module fifo_checker (input clk, rst_n, read, empty);
  // FIFO must not underflow
  ast_no_underflow: assert property (not(read && empty));
endmodule

`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

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"*

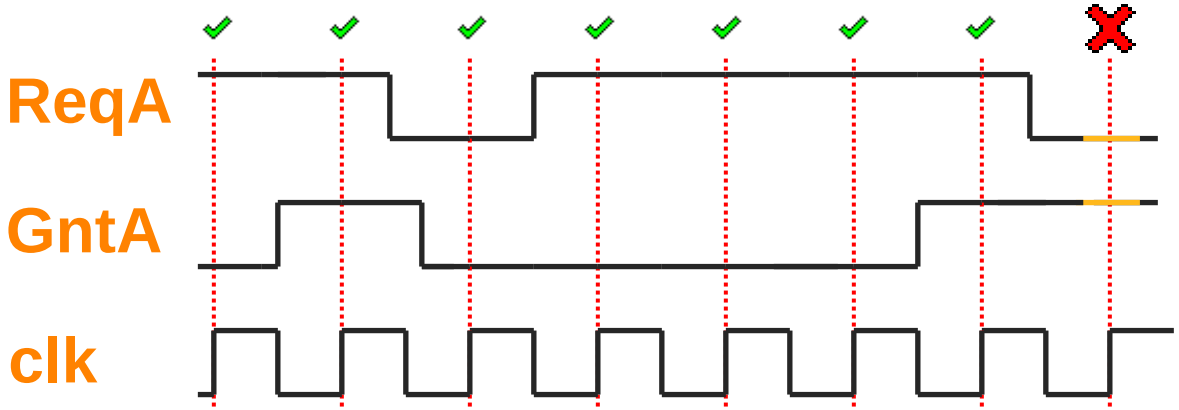
- The key to learning SVA is to learn a **small productive subset** of the language
  - Only 5-6 operators and 3-4 built-in functions is all you need!

- 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));
```

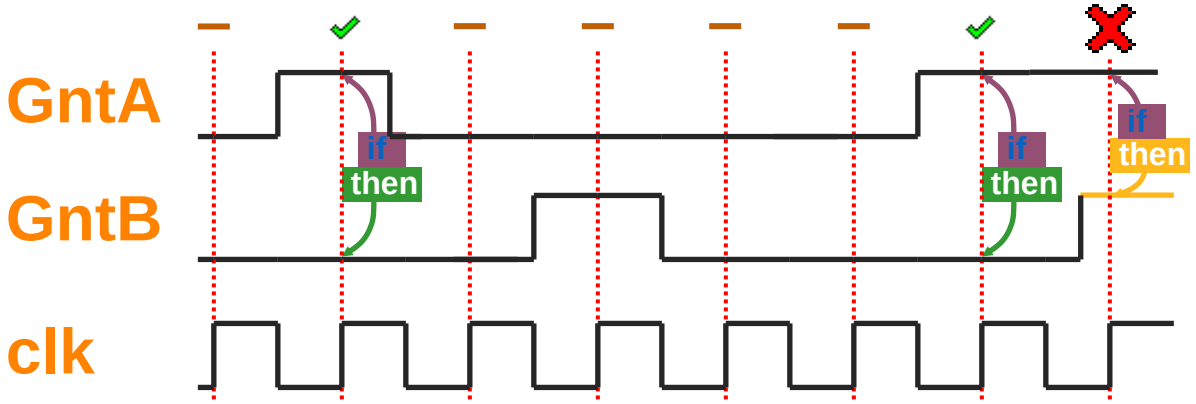


SVA Example: Same-Cycle Implications

- 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);
```

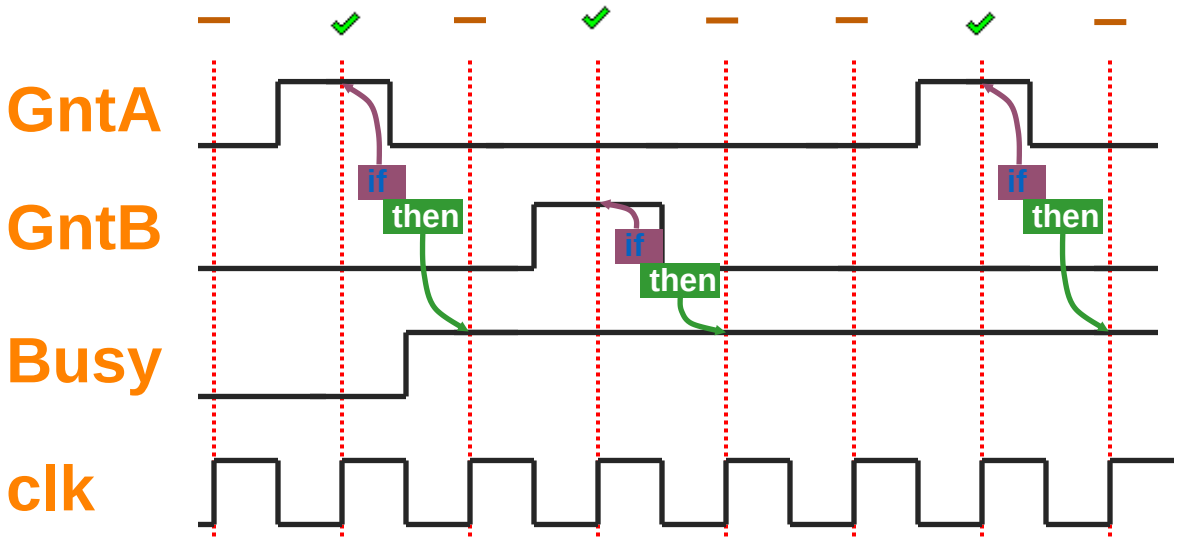


SVA Example: Next-Cycle Implications

- 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);
```

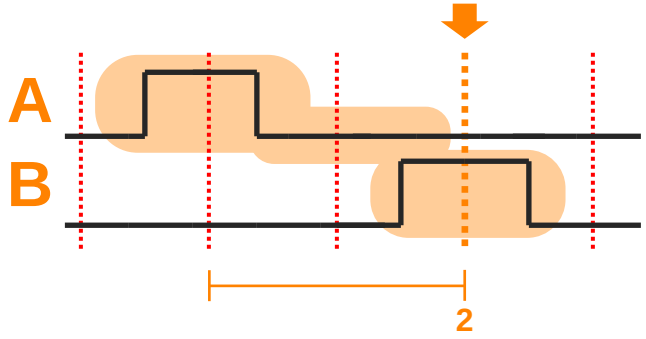


Sequences: Multi-Cycle events

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

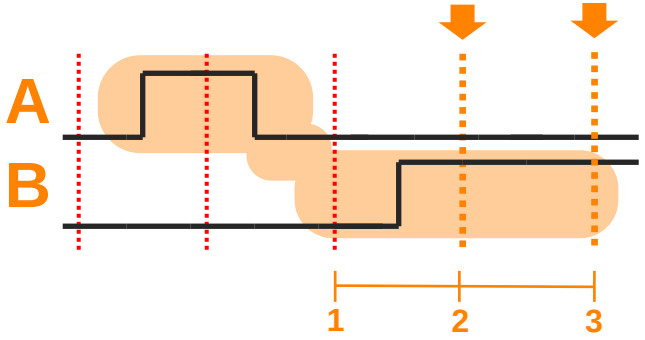
`A ##2 B`

"A happens then exactly 2 cycles later B happens"



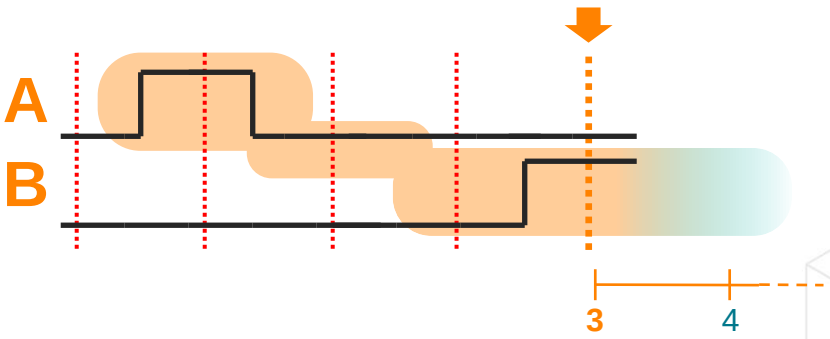
`A ##[1:3] B`

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



`A ##[3:$] B`

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

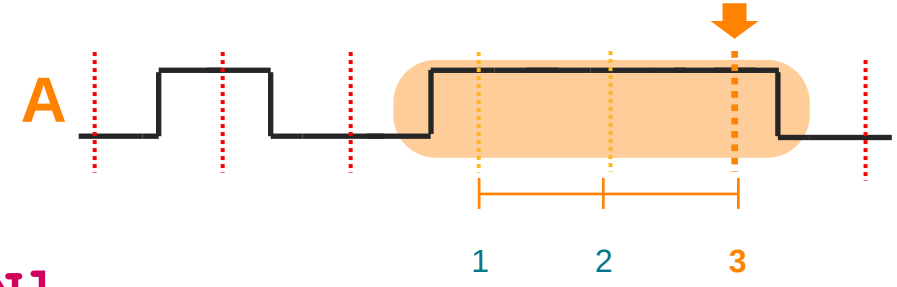


Sequences

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

`A [*3]`

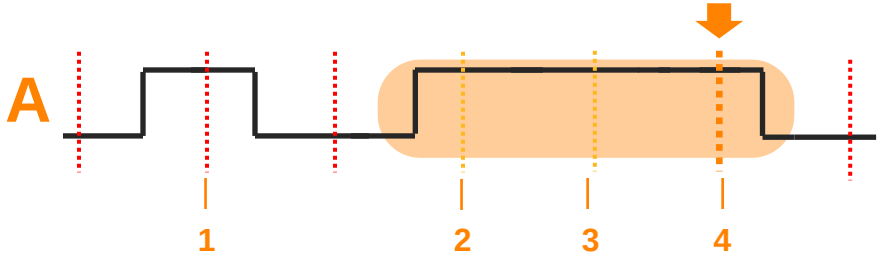
"A happens 3 times in a row



- 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"

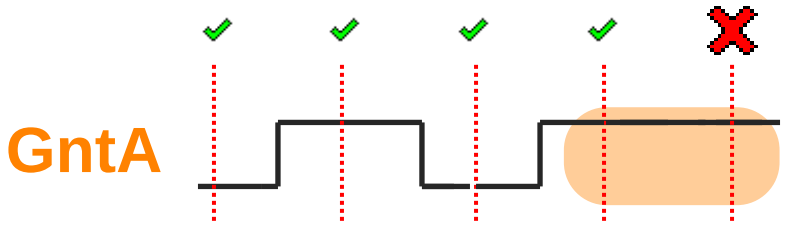
```
GntA_strobe: assert property (
  not (GntA [*2])
);
```

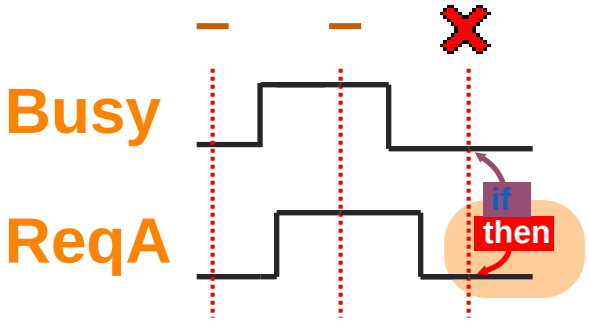
"Busy pulse should only happen if no request"

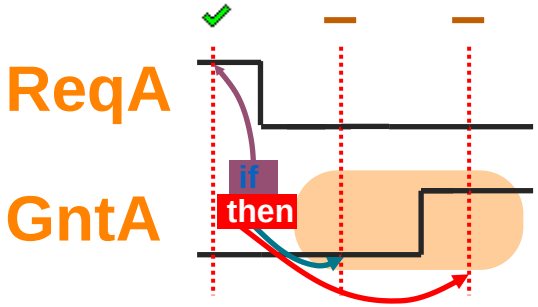
```
Busy_hold: assert property (
  (!Busy ##1 Busy ##1 !Busy)|-> !ReqA
);
```

"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);`

Summary of Operators and Built-In Functions

- 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

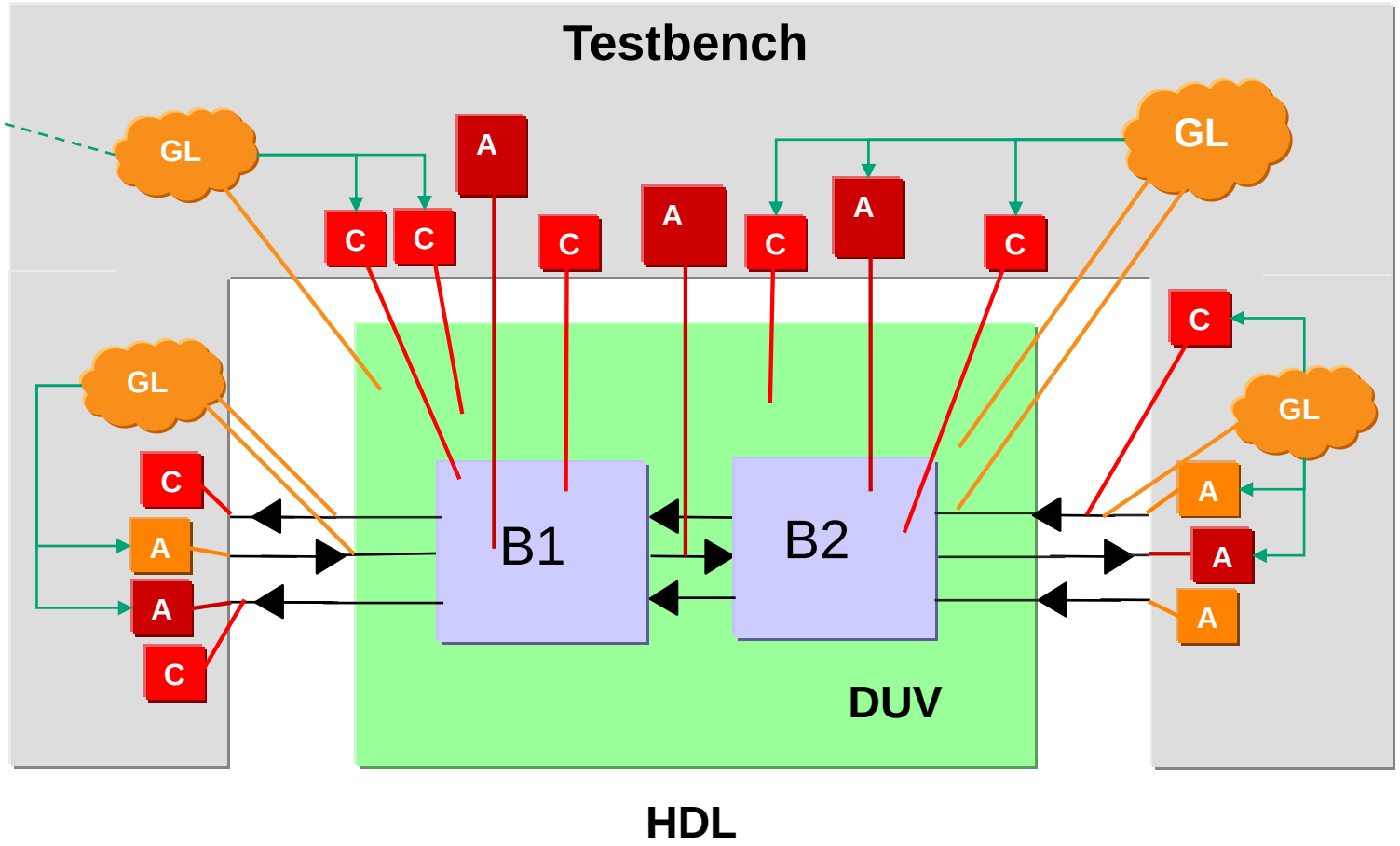
`$stable(a)`
`$past(b)`
`$rose(c)`
`$fell(d)`

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

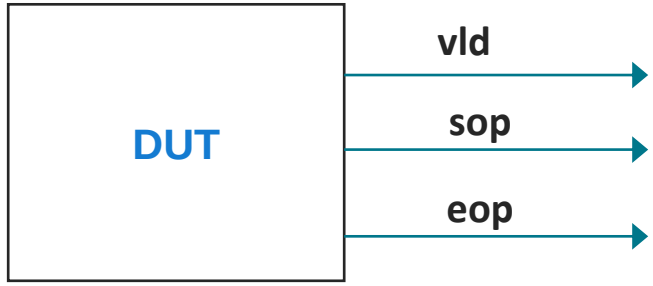
Formal Testbench

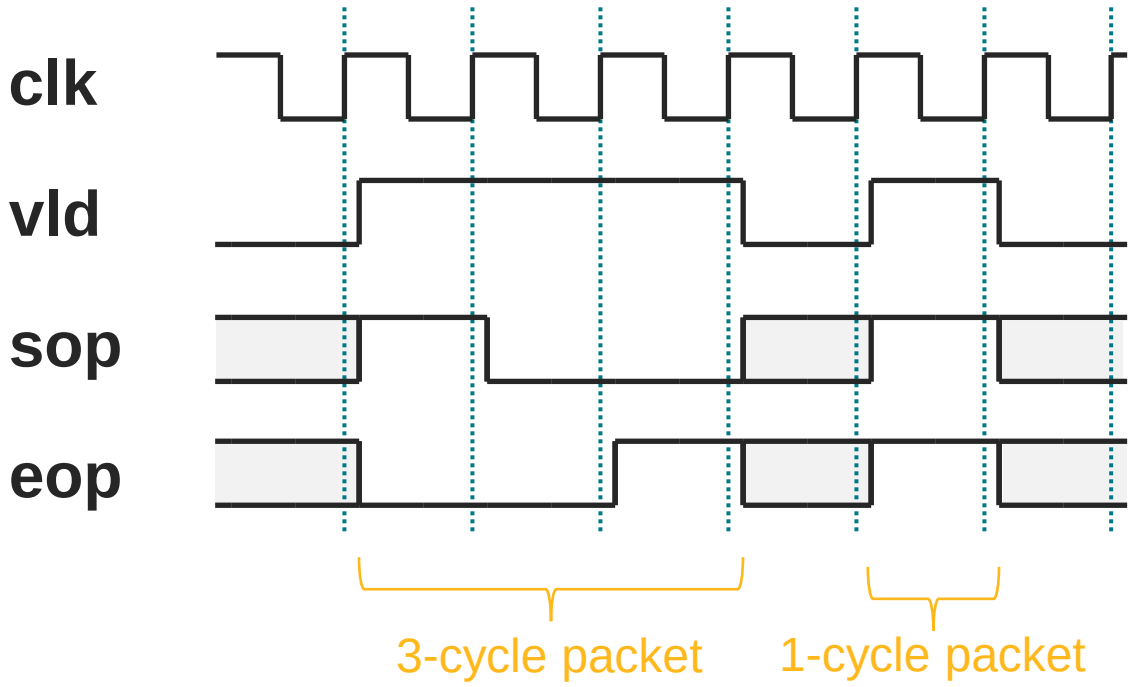
Glue Logic
monitors design
and feeds
properties



Example: SOP/EOP Interface

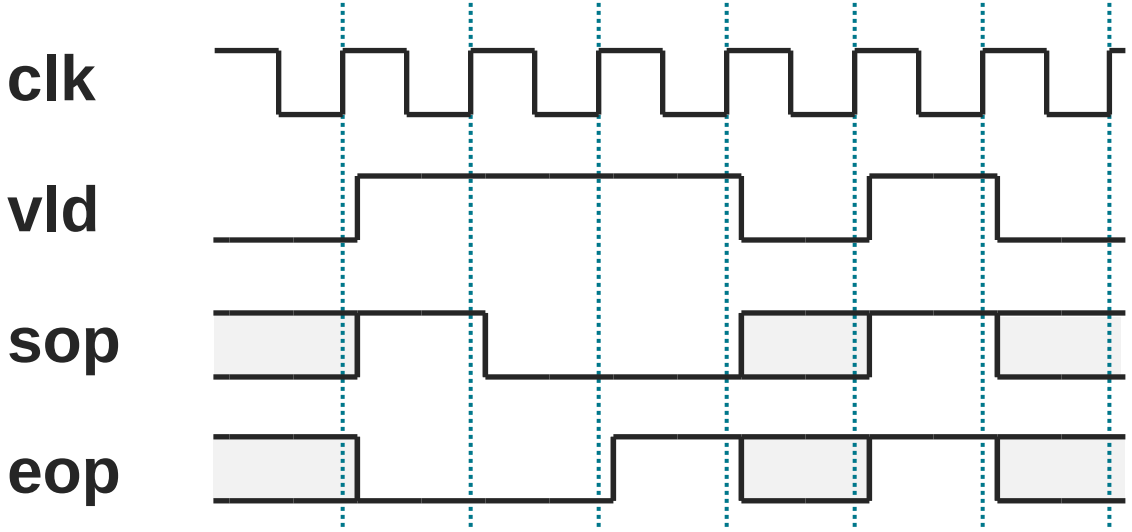
- Specification:
  - No overlapping packets (SOP-EOP always in pairs)
  - Single-cycle packets allowed (SOP and EOP at the same cycle)
  - Continuous packet transfer (no holes between SOP-EOP)





Native SVA Example: SOP/EOP Interface

Not recommended
Complex!



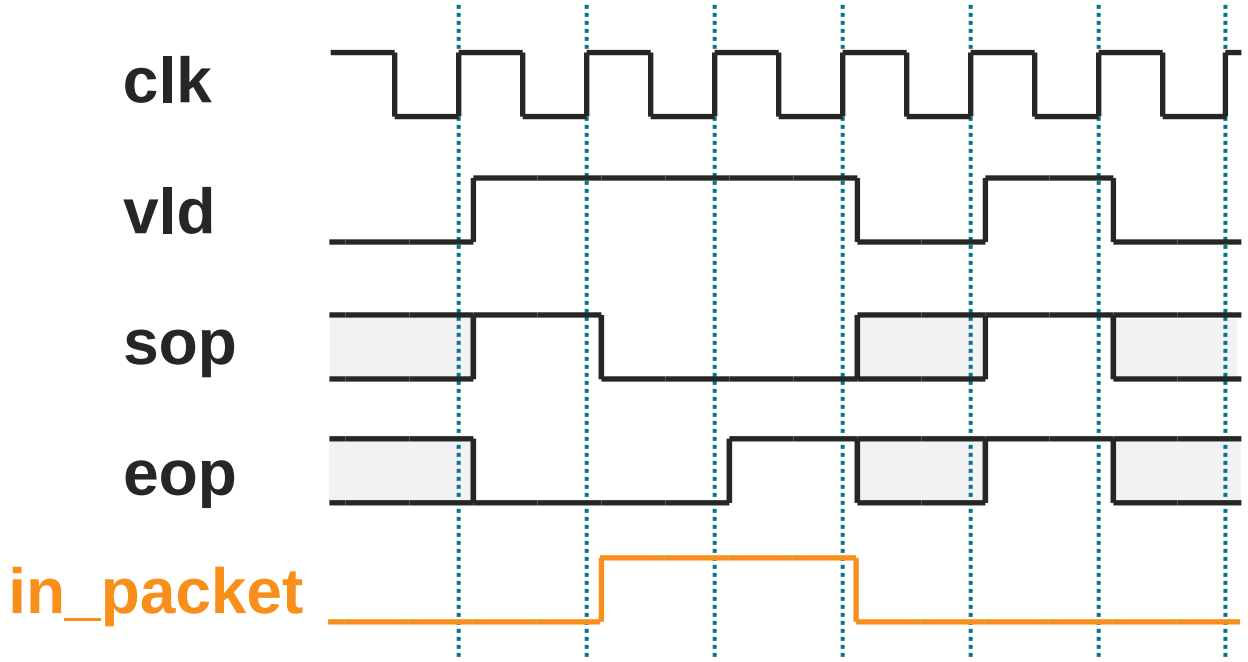
- Pure SVA version:

```
sequence sop_seen;
  sop ##1 1'b1[*0:$];
endsequence;

no_holes: assert property(
  sop |-> vld until_with eop
);
```

```
sop_first: assert property (vld && eop |-> sop_seen.ended);

eop_correct: assert property(
  not (!sop throughout ($past(vld && eop) ##0 vld && eop[->1]))
);

sop_correct: assert property(
  vld && sop && !eop |=>
  not(!$past(vld && eop) throughout (vld && sop[->1]))
);
```

If you're using **`throughout`**, **`intersect`**, **`until`**, **`.triggered`**, etc., then you're probably doing it wrong!

In-line Glue Logic Example: SOP/EOP Interface



- Glue logic version:

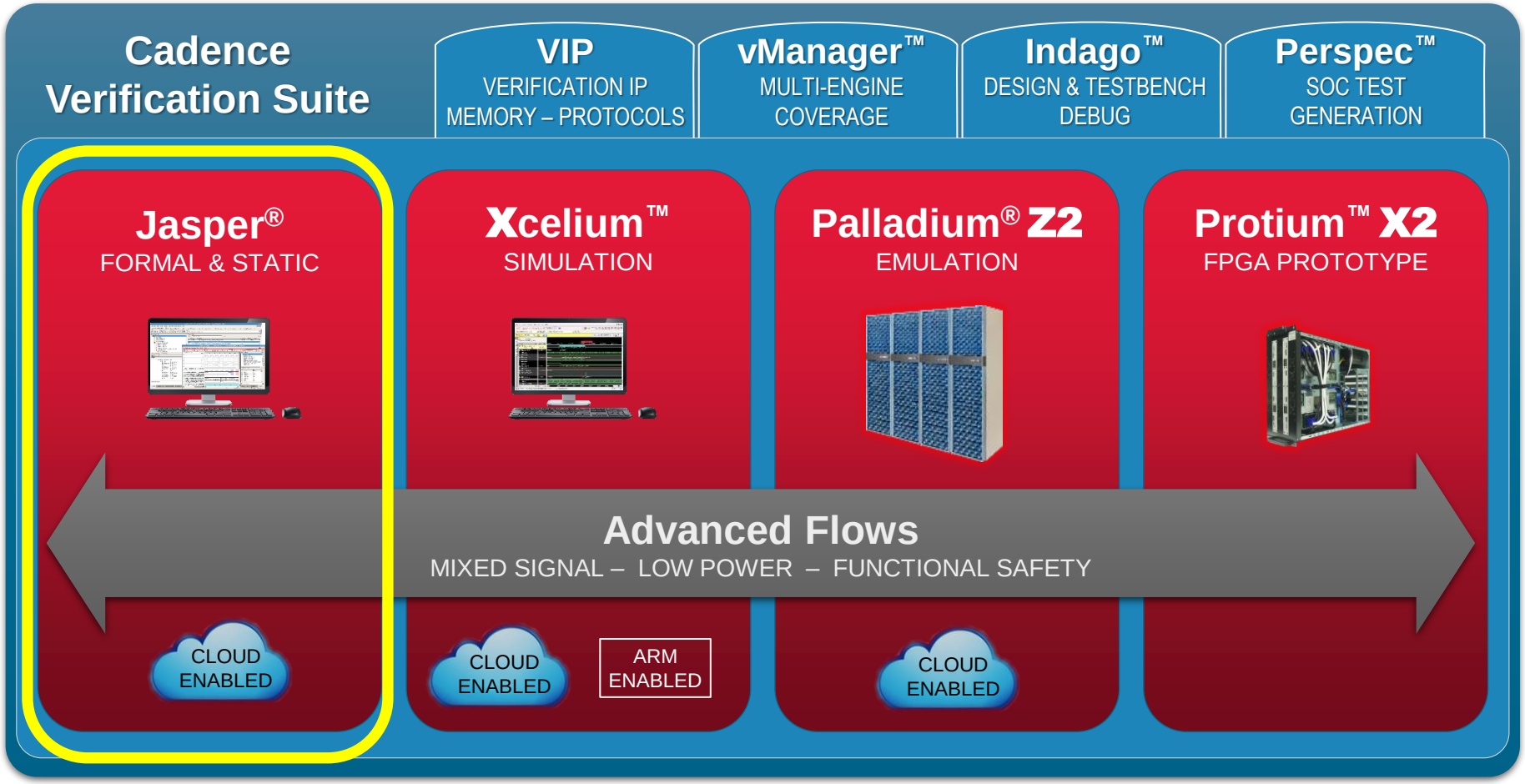
```
reg in_packet;
always @(posedge clk)
  if (!rstn || eop) in_packet <= 1'b0;
  else if (sop) in_packet <= 1'b1;

no_holes: assert property(
  in_packet |-> vld
);
```

```
eop_correct: assert property(
  vld && eop |-> in_packet || sop
);

sop_correct: assert property(
  vld && sop |-> !in_packet
);
```

Jasper Platform and FPV basic

Jasper Formal Apps

- Cadence Verification Suite

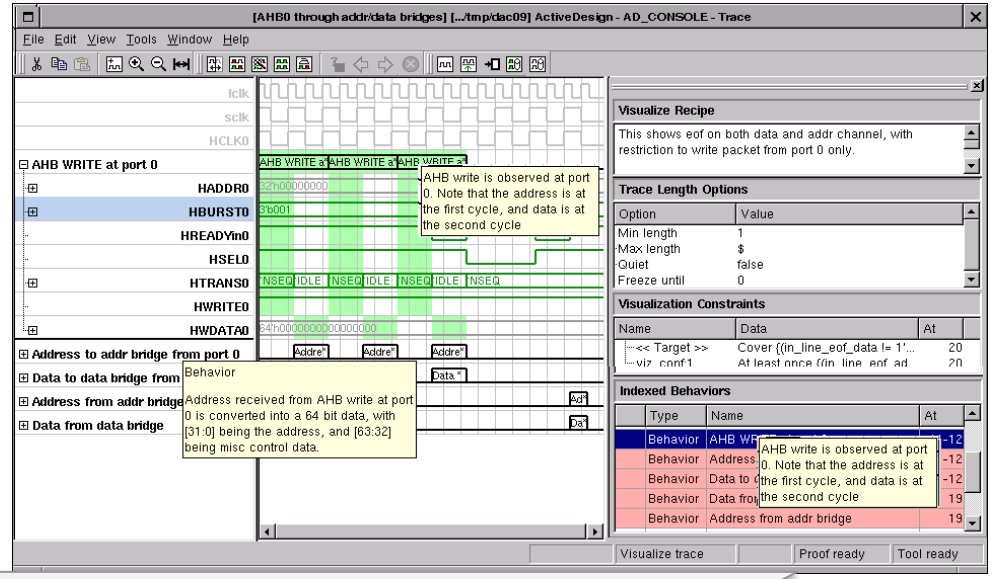


Jasper : Easiest Formal Verification to Adopt

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

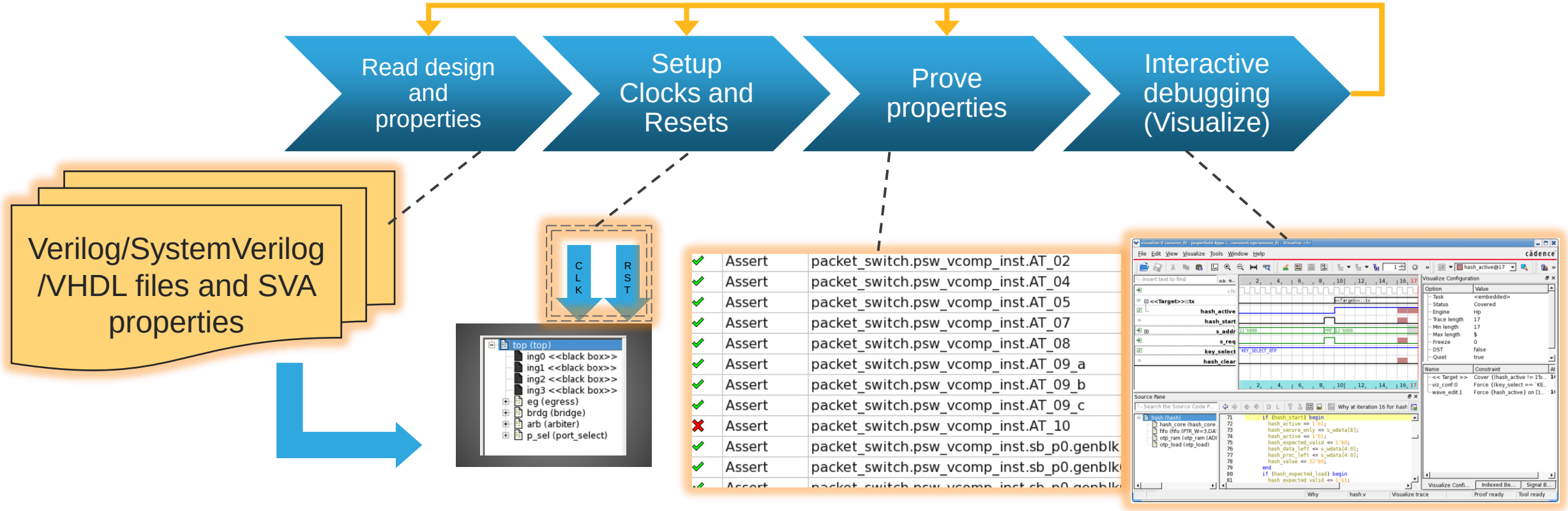
Highly interactive **formal debug** transforms to fit the App



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**

Formal Property Verification (FPV) Flow



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
```

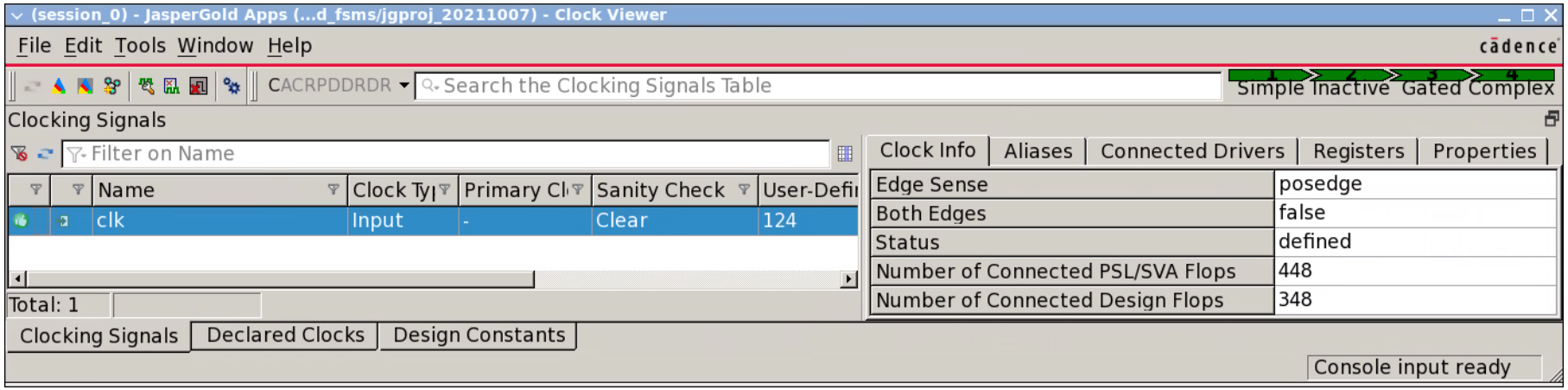
```
[<embedded>] %
```

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

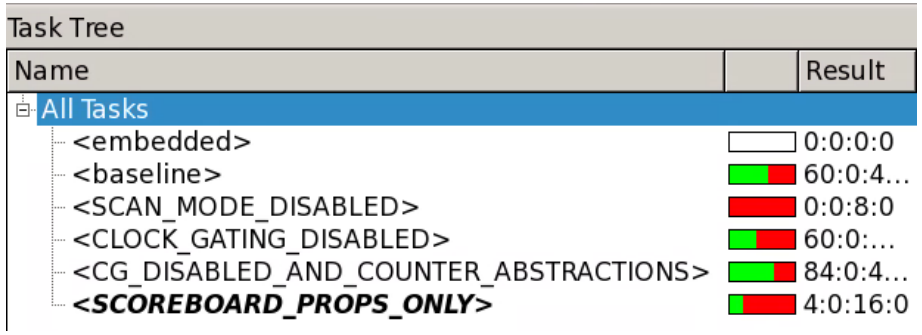
Reset Analysis

- **Run the proof!**

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

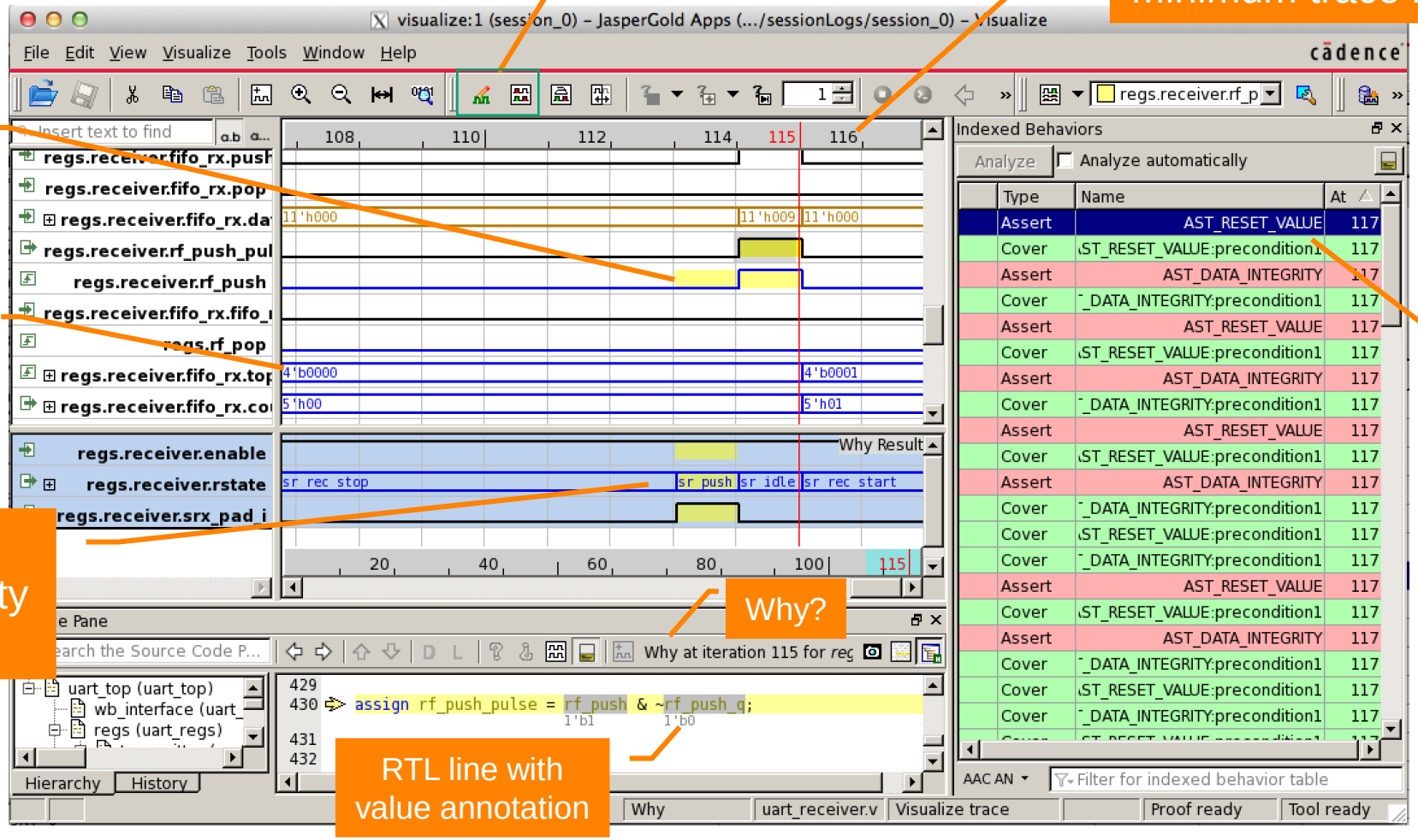
QuietTrace

Property that fails earliest

Preview with values for property or why results

Why?

RTL line with value annotation

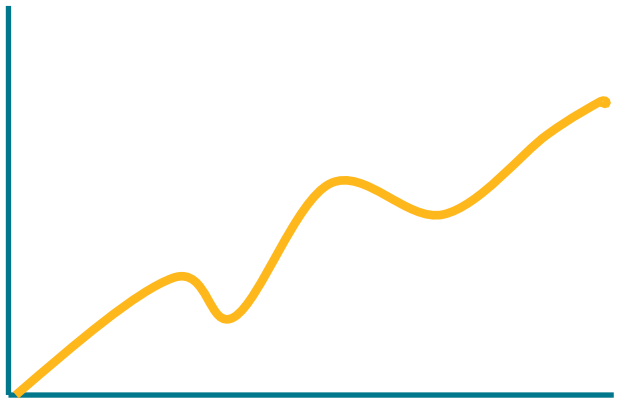


Mini FPV Demo Case

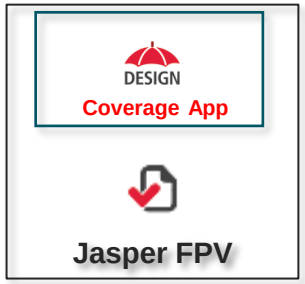
Jasper Formal Coverage Analysis

Why coverage?

- Coverage is an accepted source of metrics for showing progress and signoff
  - Often measured against a verification plan
- Provides confidence as to what is being verified
  - Will identify where verification is *incomplete*
- Gives feedback on verification efforts
  - Helps focus on risk areas
- Critical in formal, just like simulation



Formal Coverage



- **To answer formal-specific coverage related questions**
  - What code or functionality has been “exercised” by the formal testbench?
  - Is my formal checking set complete?

- Use Coverage app in conjunction with FPV (or other apps) !

Formal Coverage **Types**

- 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
  - Measurement Options
    - **Cone Of Influence (COI)** computation – structural fan-in analysis
    - **Proof Core** – formal engine-based abstraction (blackboxing)

Formal Coverage **Models**

- Code coverage models and functional coverage model
  - Representative subset (approximation) of design states to verify

- Supported code coverage models – automatically generated covers
  - Branch
  - Statement
  - Expression
  - Toggle

- Functional coverage model – user-defined
  - Property (SVA/PSL/TCL/Assertion-related)
  - Covergroup

Code vs. Functional Coverage Model

- Code coverage model
  - Easy to generate (automatic)
  - Guaranteed to be structurally complete – not prone to human error
  - Standard models capture much of the meaningful behavior of the design
  - May not capture all meaningful design *functionality*
  - Can be noisy – large number of covers, some may not be important

- Functional coverage model
  - Possible to represent all meaningful design functionality
  - Implements the verification plan – what needs to be verified
  - Noise-free – created deliberately
  - Requires planning, coding, and debug
  - May be incomplete due to human error

Code Coverage Models – Branch/Statement

- Branch model creates a cover for each continuously executed block of code
  - Contained (or could be contained) within begin/end
  - Branch and Statement scoring options create covers with finer granularity
  - Branch and Statement are default in JG, optional in Incisive

- Example

```
if (new_trans)  begin
        trans_started = 1'b1;
        header = din;
end
else
        trans_started = 1'b0;
```

Block: lines 1–5

Statement: `trans_started = 1'b1;`

Statement: `header = din;`

Block: lines 6–7

Statement: `trans_started = 1'b0;`

Code Coverage Models – Expression

- Expression model decomposes logical expressions into multiple covers
  - Control - checks if each input has controlled the output value of the expression

- Example 1:

```
assign C = A && B;
```

Table for && operator

A	B	Result
0*	1	0
1	0*	0
1*	1*	1

Expression

Expression

Expression

Control model considers A=0 and B=0 redundant with these.

* controlling operand

Code Coverage Models – Toggle

- Toggle model represent the ports and internal signals changing values

```
input enable;
output intr;

logic flag;
```

Toggle cover items
Rise
Fall

Each signal is covered if both its rise and fall items are covered

Coverage Models – Property (Functional)

- SVA, PSL, TCL cover properties – user defined
  - Describe specific states, conditions, or sequences to be verified
  - Part of <embedded> or other tasks

- Assertion-related covers
  - `assert property (@(posedge clk) A |=> B);`

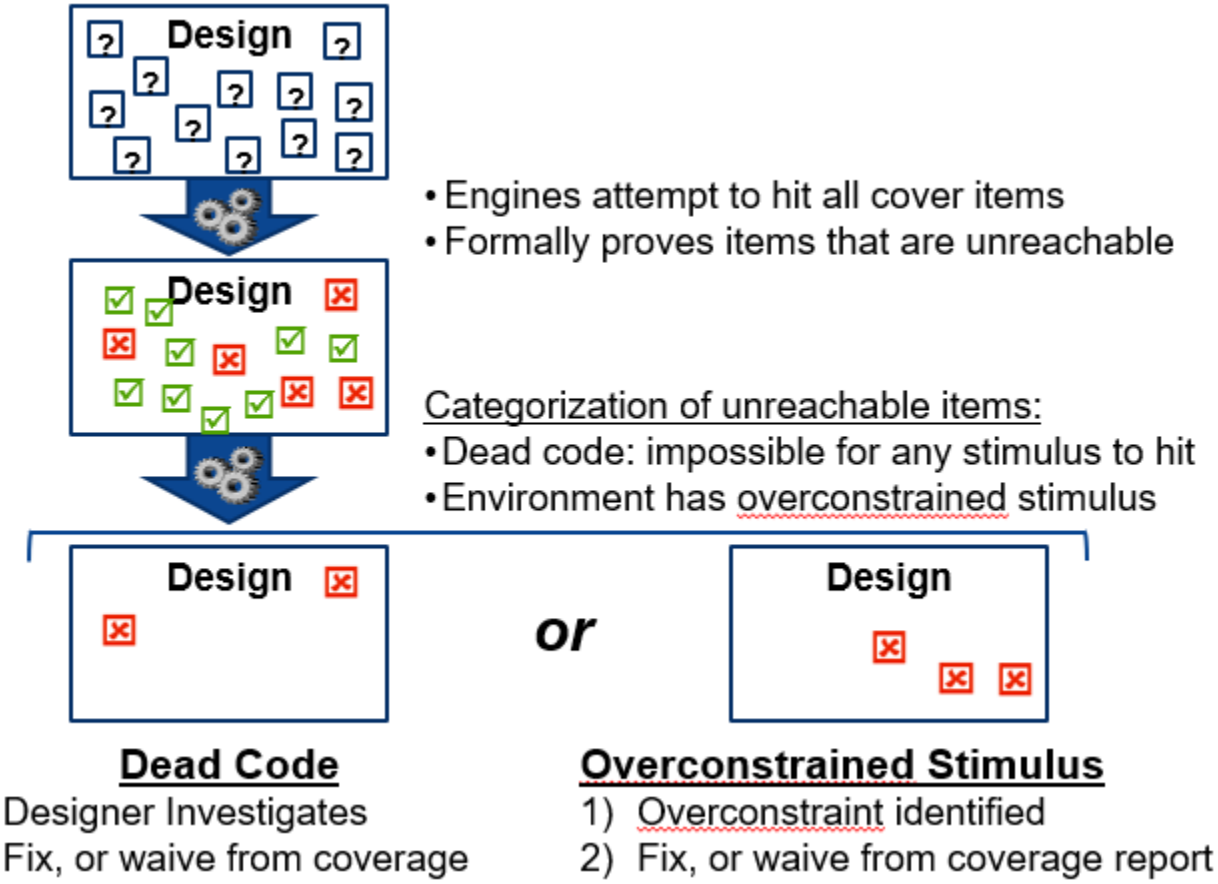
    - If related cover (precondition) is enabled: `cover property (A)`
      - Covered: The assertion is capable of being triggered
      - Unreachable: The assertion is never checked because it can never be triggered (assert status is Vacuous Pass)

    - If related cover (witness) is enabled: `cover property (A ##1 B)`
      - Covered: There is at least one scenario in which the assert is true
      - Unreachable: There is no scenario that satisfies the assert. All possible scenarios will result in CEX

Stimuli Coverage Type

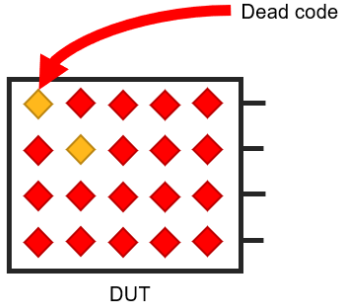
What code or functionality is reachable by the formal testbench?

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

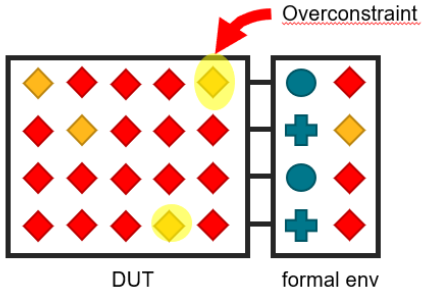


Stimuli Coverage – **Unreachable** vs. **Deadcode**

- **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



Stimuli Coverage

- Classification of the environment regarding constraints

Classification	Definition	Behavior	Remedy
Underconstrained	Environment allows more scenarios than specified for DUT	Asserts can produce false CEX	Refine the environment's assumptions
Well constrained	Environment can recreate all the scenarios in which DUT should operate	Asserts produce CEX only when there is a bug	(n/a)
Overconstrained	Environment restrict scenarios to a subset of what the DUT can operate	A bug may exist and not cause a CEX	Refine the environment's assumptions

Stimuli Coverage

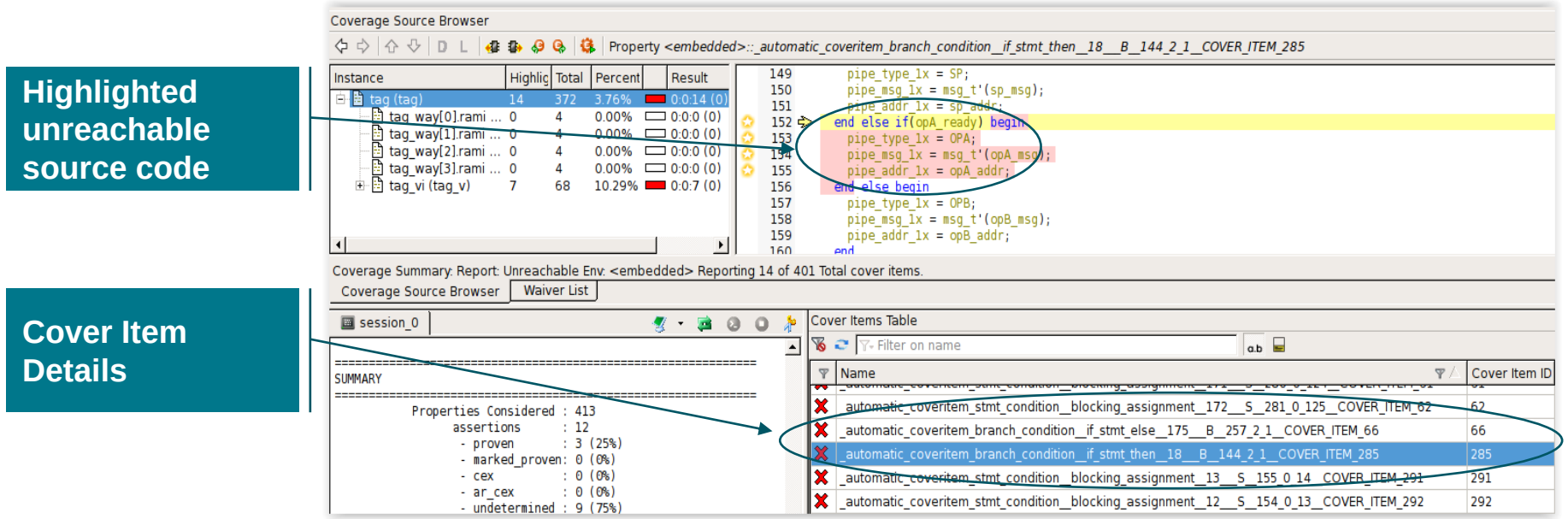
- Classification of the environment regarding constraints

Classification	Definition	Behavior	Remedy
Underconstrained	Environment allows more scenarios than specified for DUT	Asserts can produce false CEX	Refine the environment's assumptions
Well constrained	Environment can recreate all the scenarios in which DUT should operate	Asserts produce CEX only when there is a bug	(n/a)
Overconstrained	Environm scenarios what the DUT can operate		assumptions

Goal for the environment.
No false failures are produced and all possible scenarios are tested.

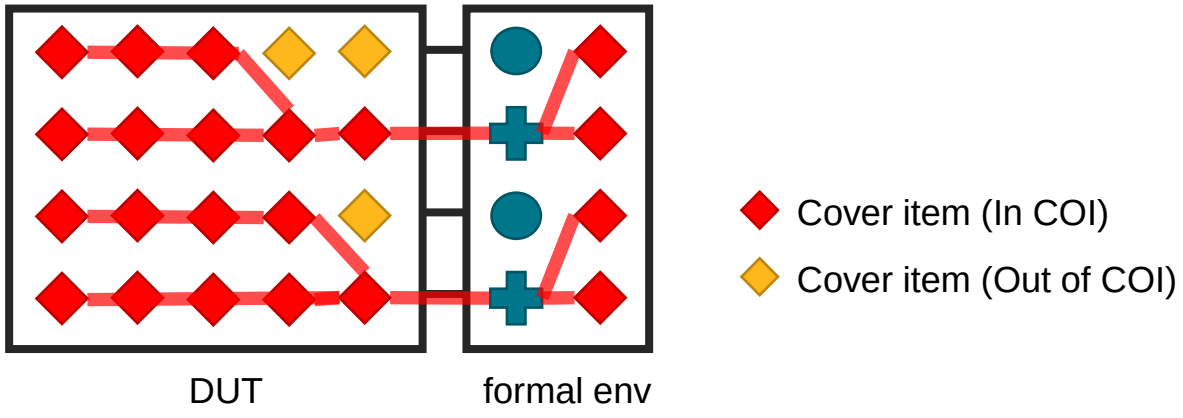
Stimuli Coverage Reporting

- Cover items with unreachable proof status are highlighted
- Name, ID, source location, cover item type are displayed in table
- Can automatically filter cover items that are unreachable due to reset or constant propagation



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

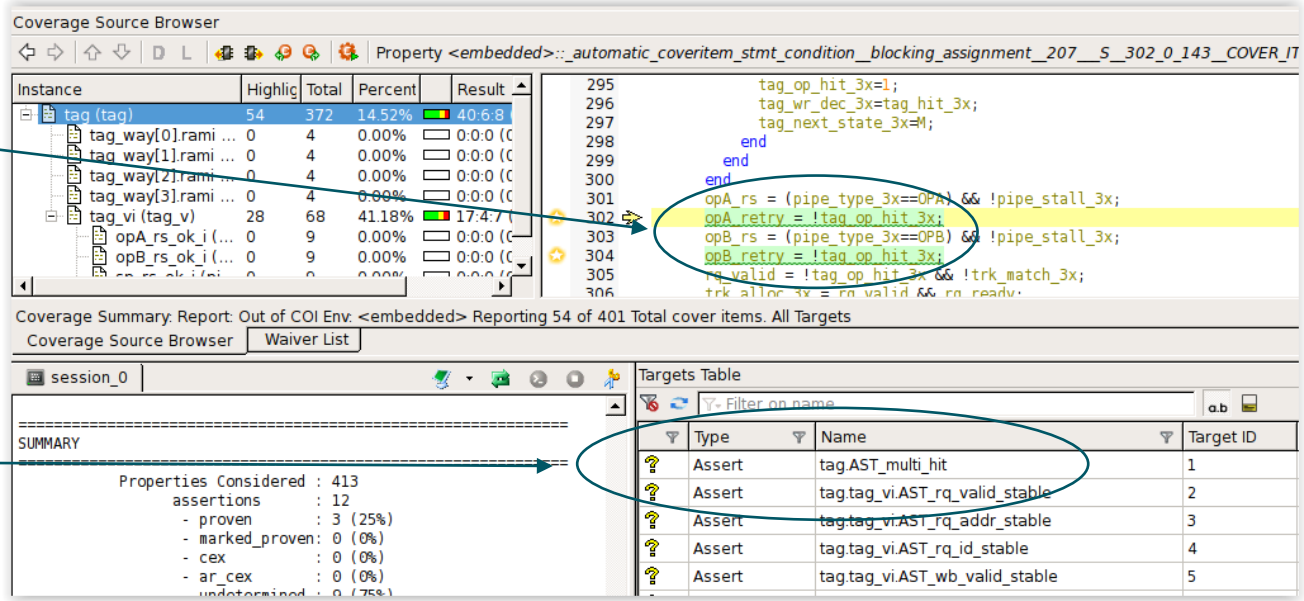


COI Coverage Reporting

- Reports the cover items not included in the COI of any assertions
- Reporting does not require running any proofs
- By default, COI reported is cumulative result of all targets (asserts)
- Can optionally select a subset of assertions for reporting

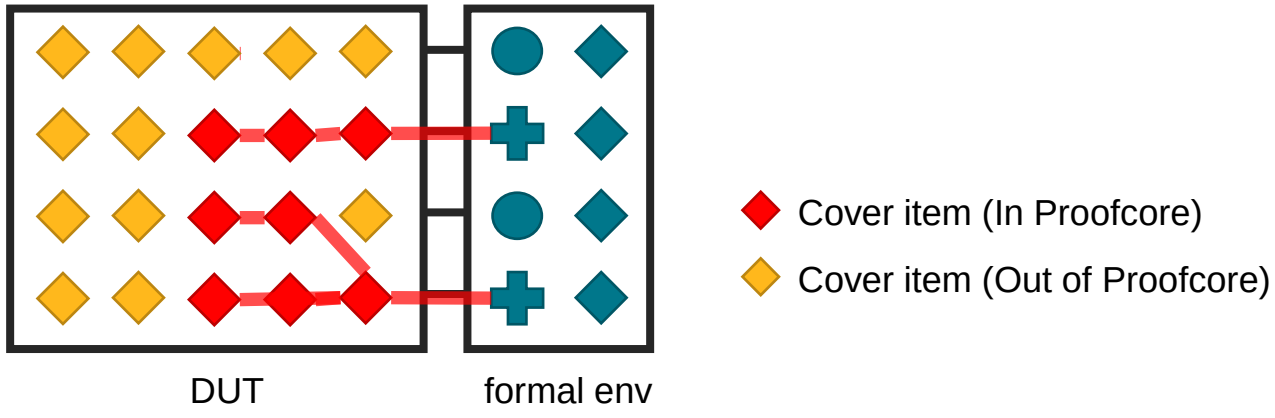
Cover items not within the COI of any assertions

Can select a subset of asserts for coverage reporting



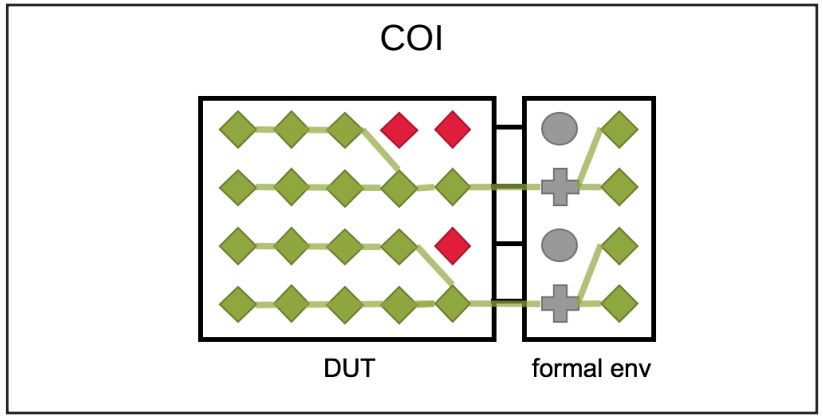
Checker Coverage Type – Proof Core Measurement

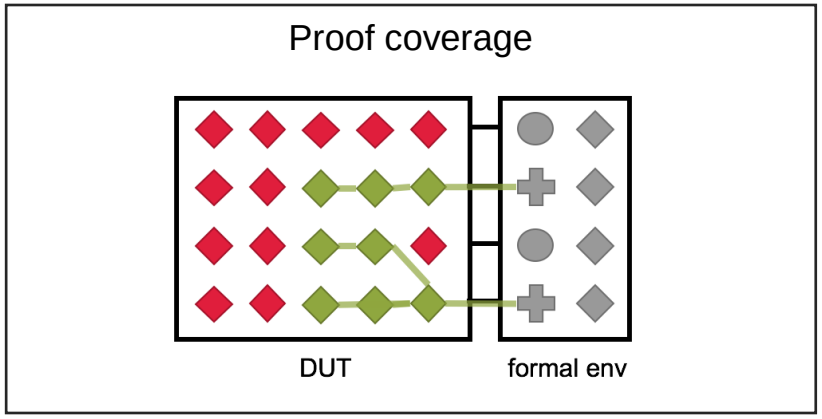
- Represents the portion of the design verified by formal engines
- Subset of the COI – COI represents the maximum potential of proof coverage
- Engines abstract a portion of the design during the proof process, iteratively consider a larger portion of the design until a proof, or bounded proof, is established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



Proof Coverage vs. COI analysis

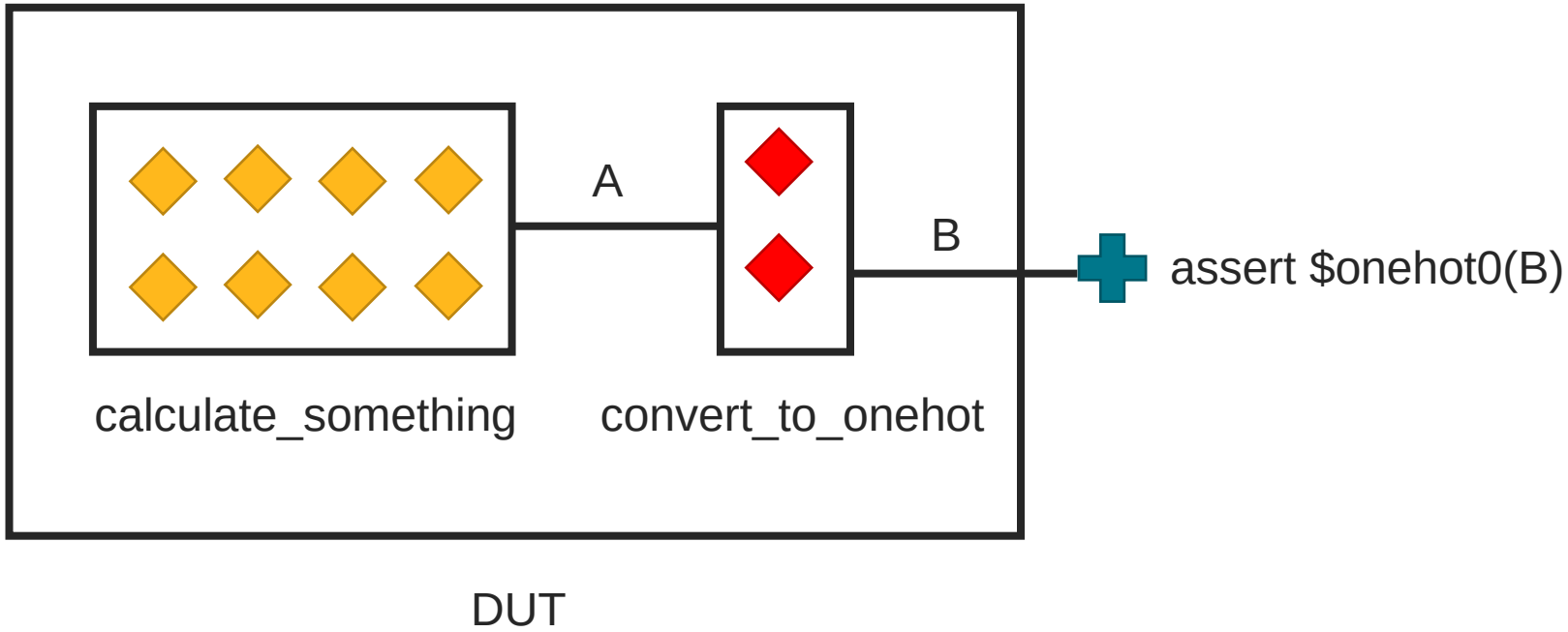
- Proof coverage is a subset of the COI – COI represents the maximum potential of proof coverage
- COI doesn't require a proof to take place, while proof coverage does.





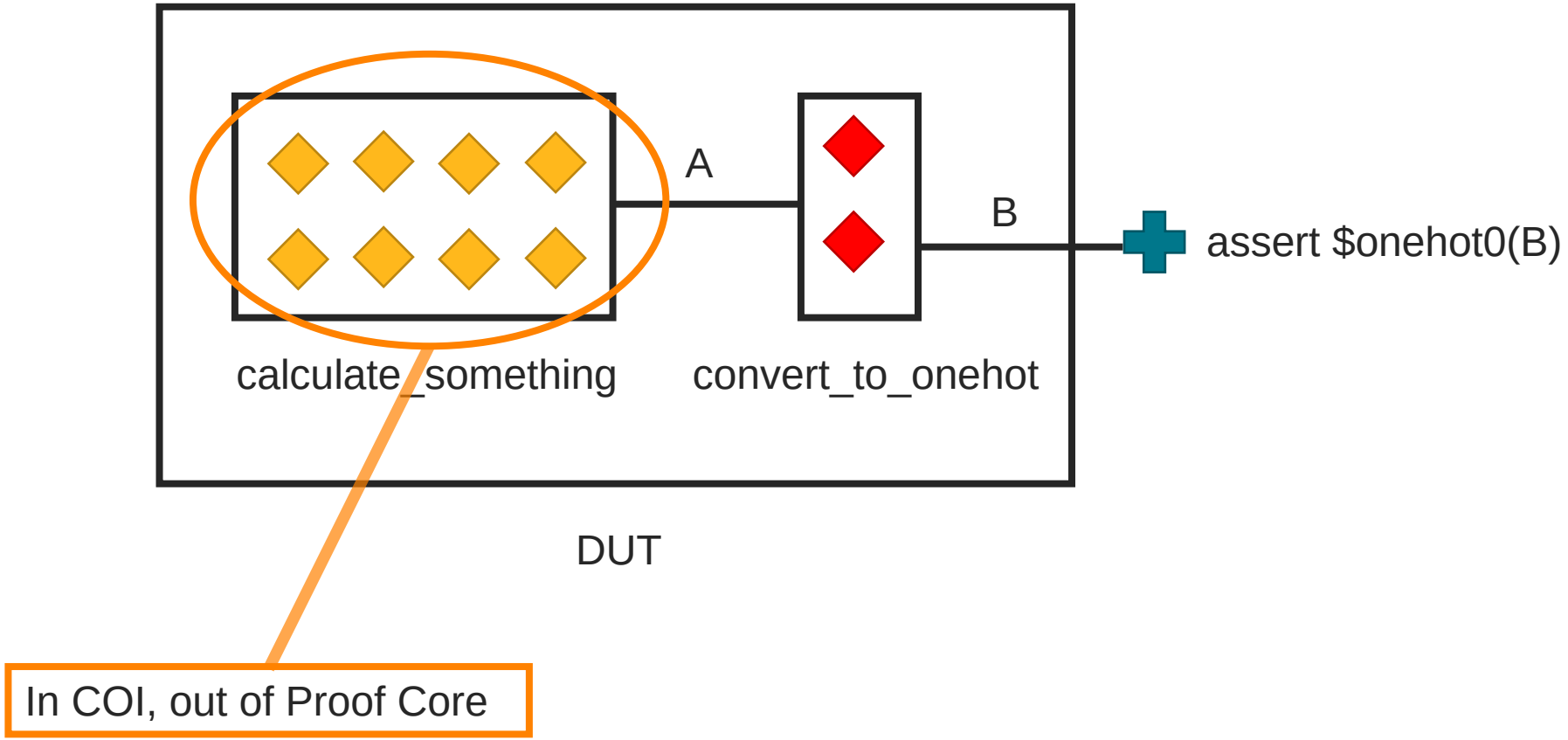
Proof Coverage

- How to interpret the result



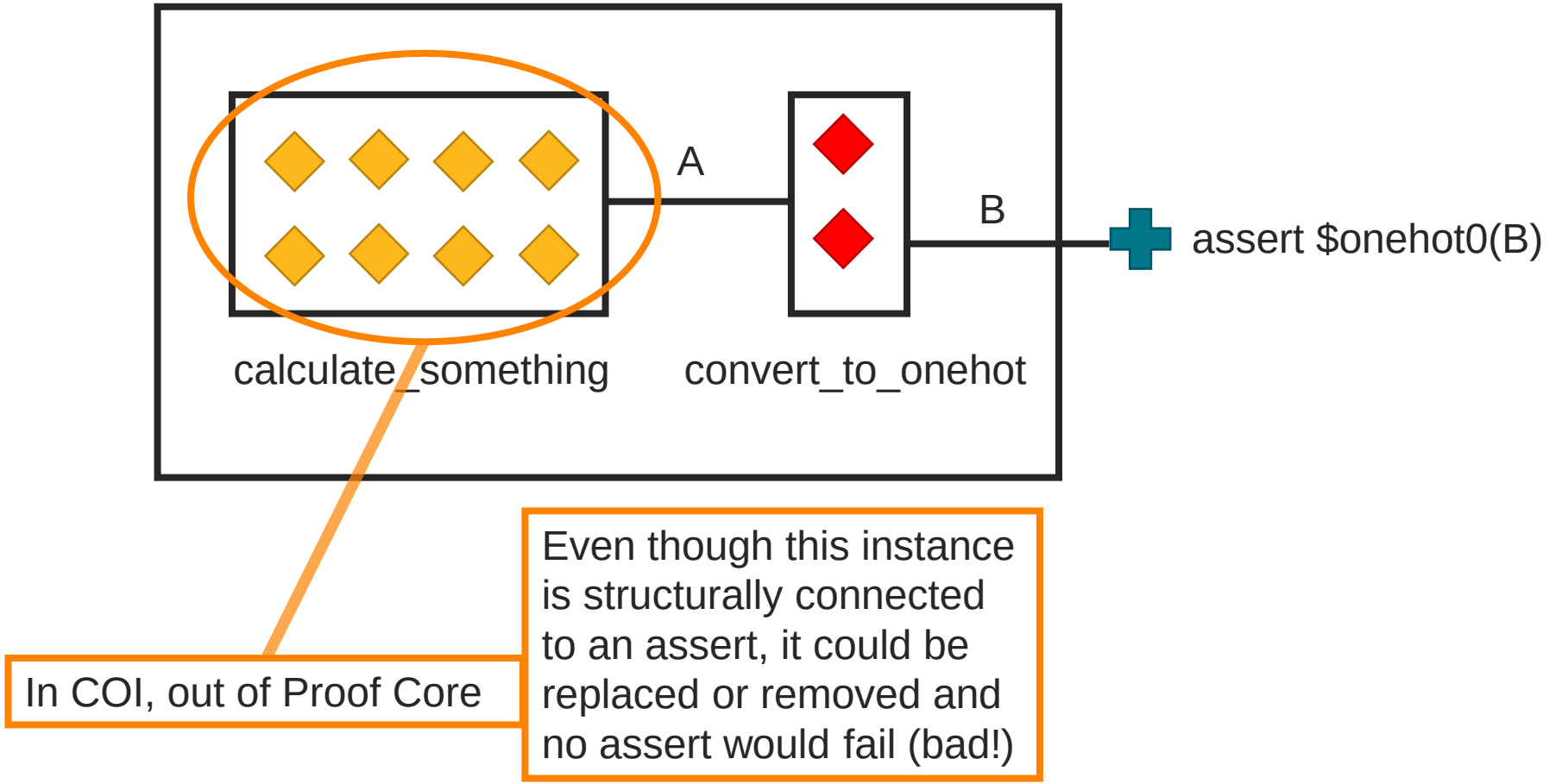
Proof Coverage

- How to interpret the result



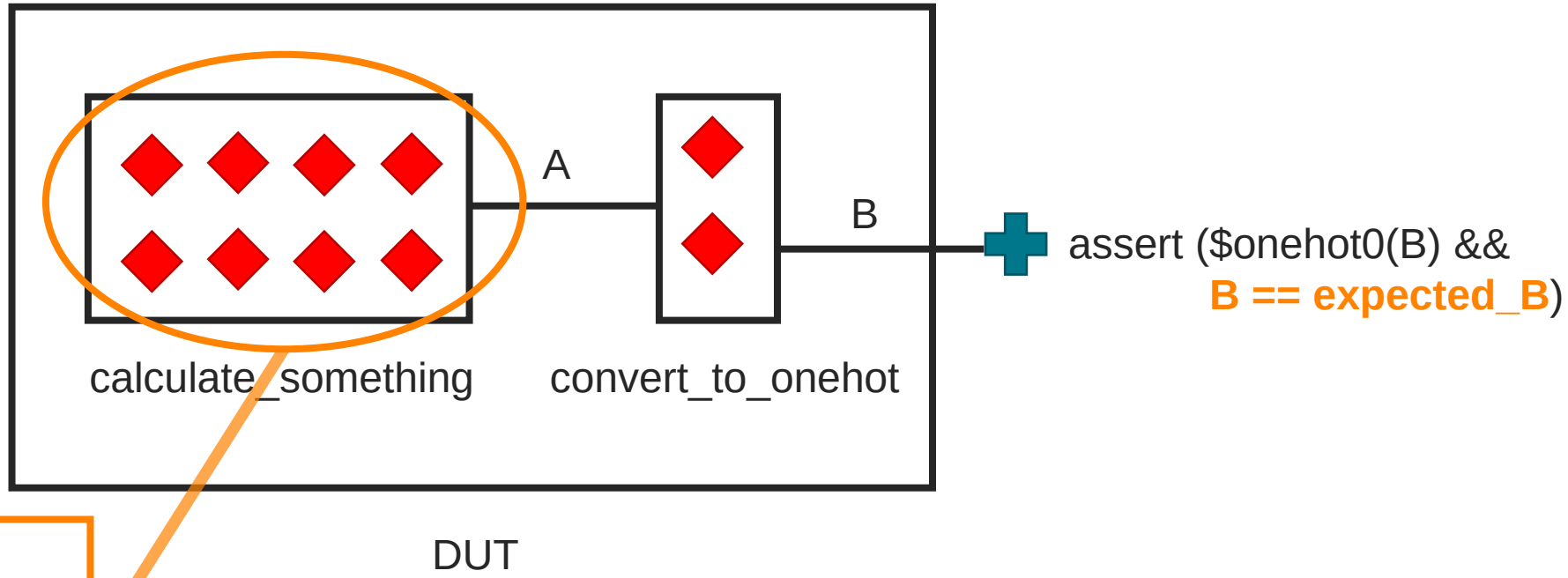
Proof Coverage

- How to interpret the result



Proof Coverage

- How to interpret the result



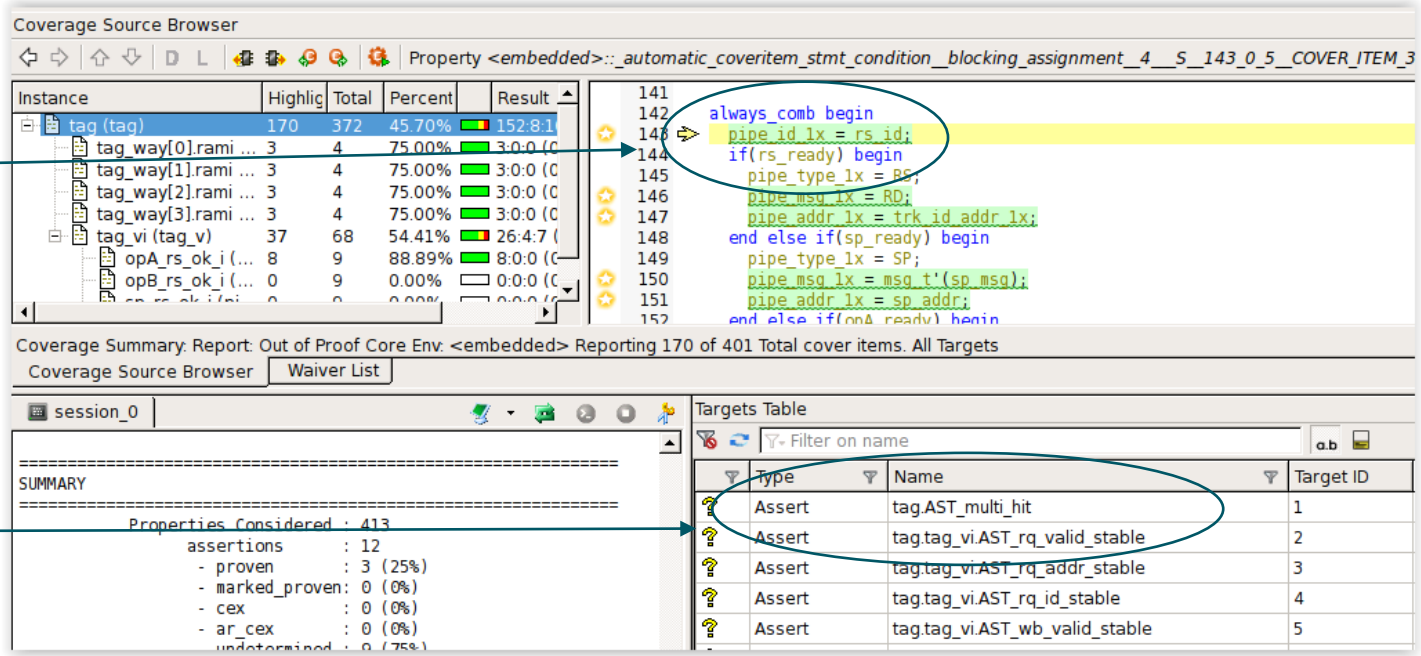
Adding checks will increase proof coverage, ensuring that the design's functionality is being verified (good!)

Proof Coverage Reporting

- Abstraction engines will contribute to proof coverage metric
- Reported as cumulative coverage result of all assertions by default

Cover items not verified by formal proofs

Can select a subset of asserts for coverage reporting



cādence®