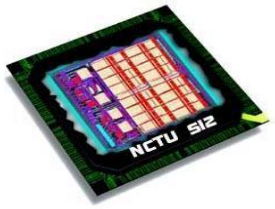
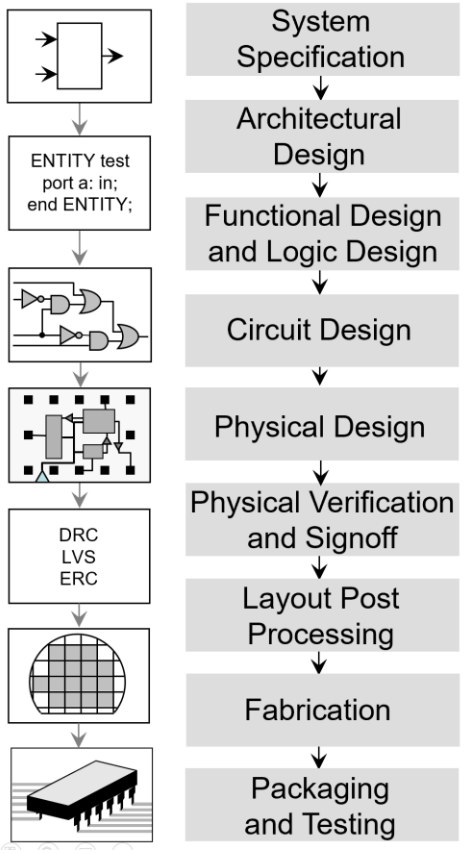
Functional Formal Verification with Cadence JasperGold



What is Verification? Recall Lab3

- **Verification == Bug Hunting**
  - A process in which a design is verified against a given design specification before tape out.
- **Verification include:**
  - Functionality (Main goal !!)
  - Performance
  - Power
  - Security
  - Safety
- **How to perform the verification?**
  - Simulation of RTL design model (Lab3)
  - Formal verification (This Chapter!!)
  - Power-aware simulations (Lab10)
  - Emulation/FPGA prototyping
  - Static and dynamic timing checks



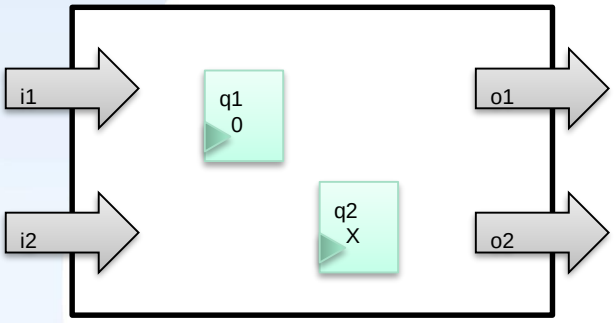
Outline

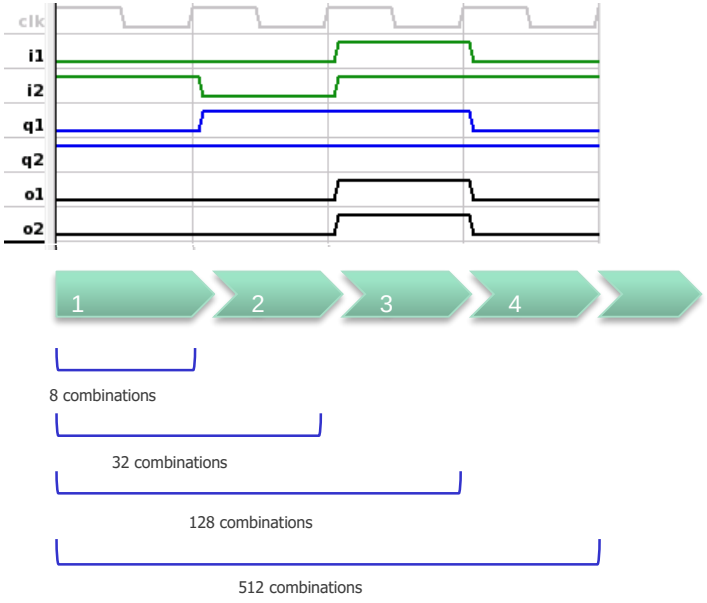
- **Section 1. Introduction to Formal Analysis**
- Section 2. Introduction to SVA (System Verilog Assertion)
- Section 3. JasperGold – Setup
- Section 4. JasperGold – Formal Coverage Analysis
- Section 5. JasperGold – Assertion Based Verification IP & Scoreboard

- Appendix A. JasperGold Setup
- Appendix B. Bound Analysis

Formal Analysis

- **Formal tests all possible stimulus, one cycle at a time**
  - All value combinations on inputs and undriven wires at every cycle
  - All value combinations on unitialized registers at first cycle

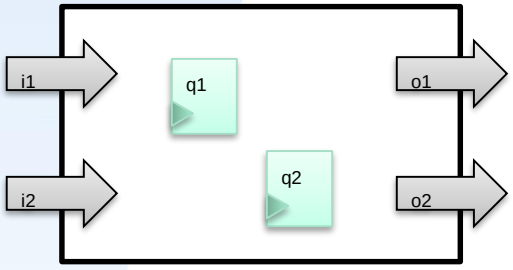




Similar to simulating `$random` at every input/register for all possible random values!

Properties

- **Formal checks properties as it is walking through stimuli**
  - Report when assert properties are violated
  - Report when cover properties are hit




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

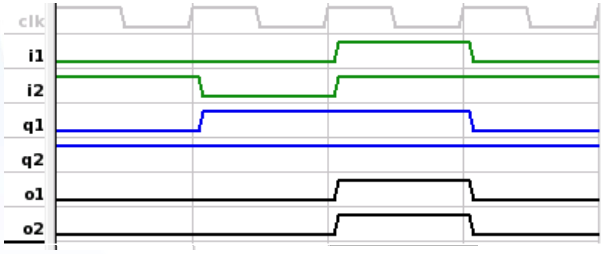
Waveform saved in database

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

Waveform saved in database

Found scenario that hits cover!

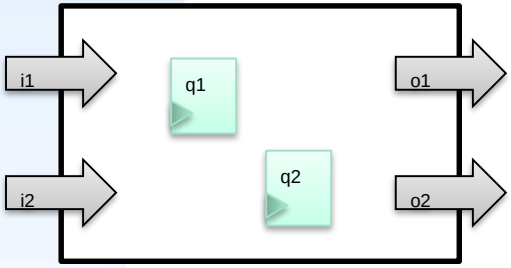
Found scenario that violates assertion!



- Assertions and covers are formal's checkers!

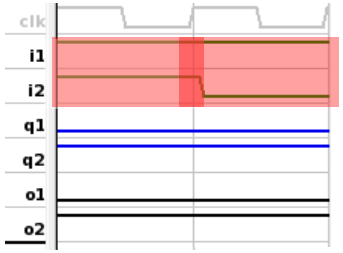
Constraints

- **Not all input stimulus combinations are legal**
- **Assume properties tell formal what is legal**
  - Only scenarios where assumptions are true will be considered
  - Formal throws away any stimulus that violates assumptions




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

Found scenario
that violates assumption!



Discard this scenario and
continue

Proof Results

Proven Unreachable
(Full Proof)

	Type	Bound
✓	Assert	Infinite
✗	Cover	Infinite

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

Undetermined

~= cycles

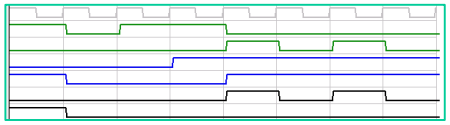
	Type	Bound
?	Assert	7 -
?	Cover	7 -

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

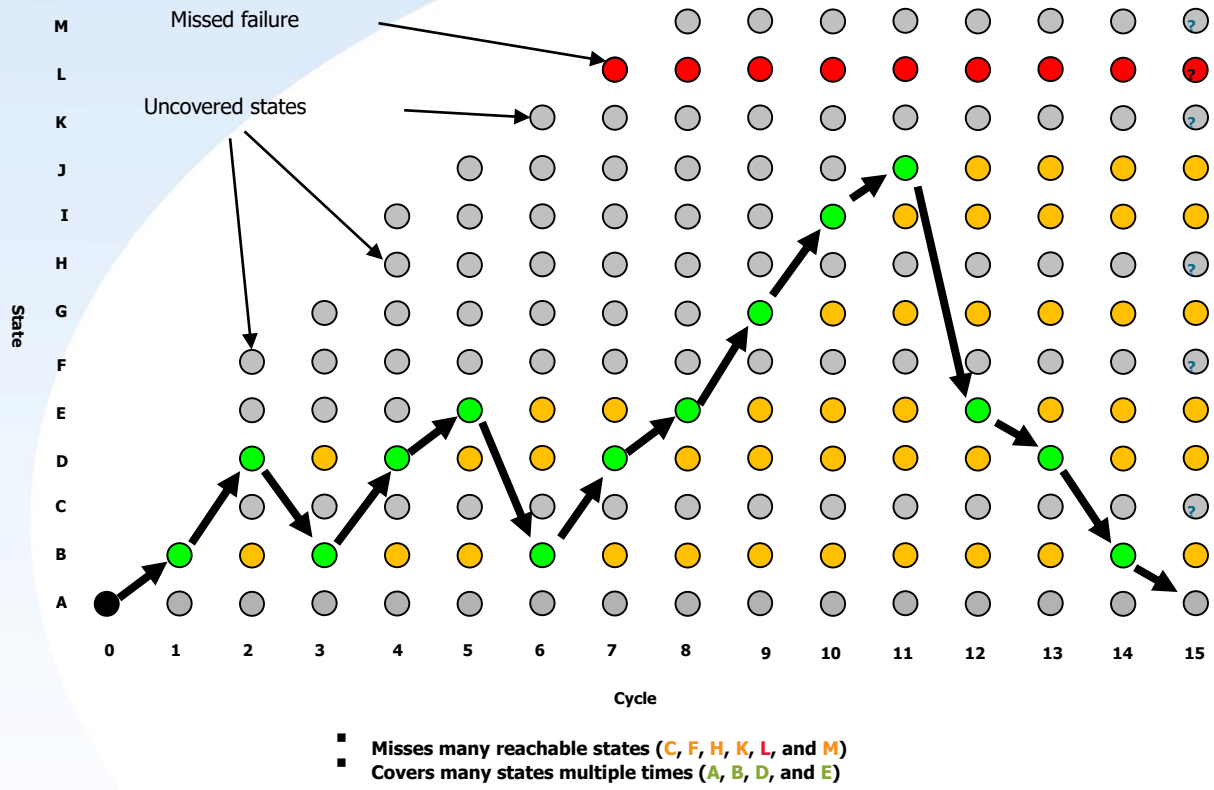
CEX Covered

	Type	Bound
✗	Assert	12
✓	Cover	49

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



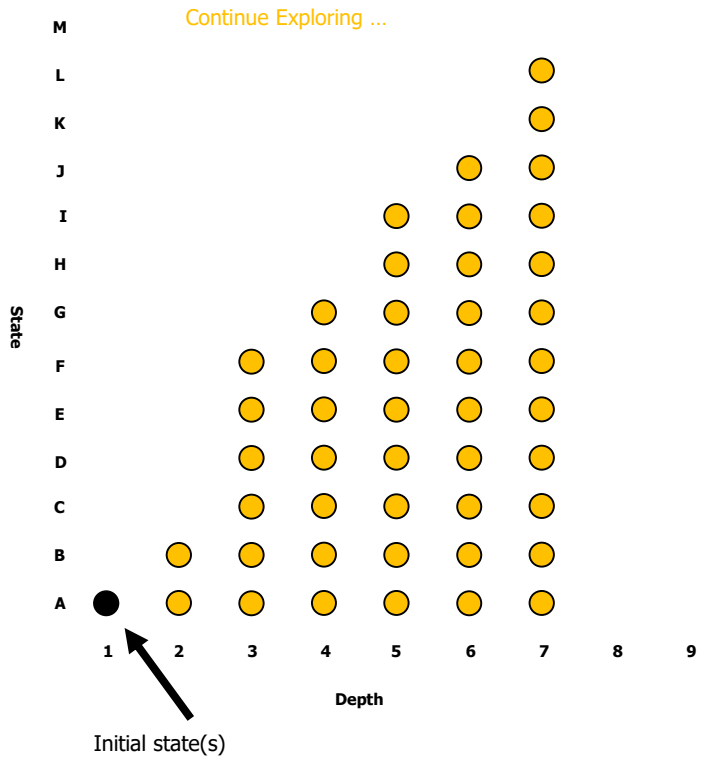
Compared with Simulation



State = value of all registers/inputs

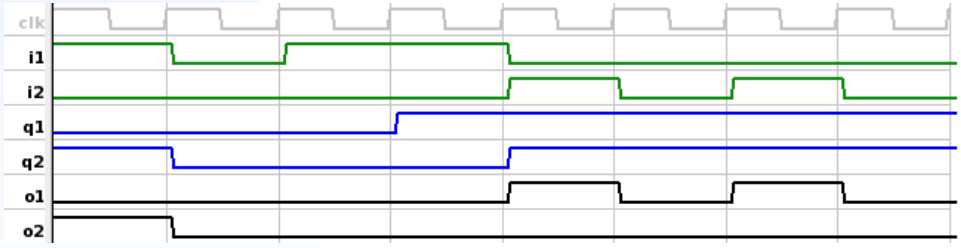
Formal State Space Exploration

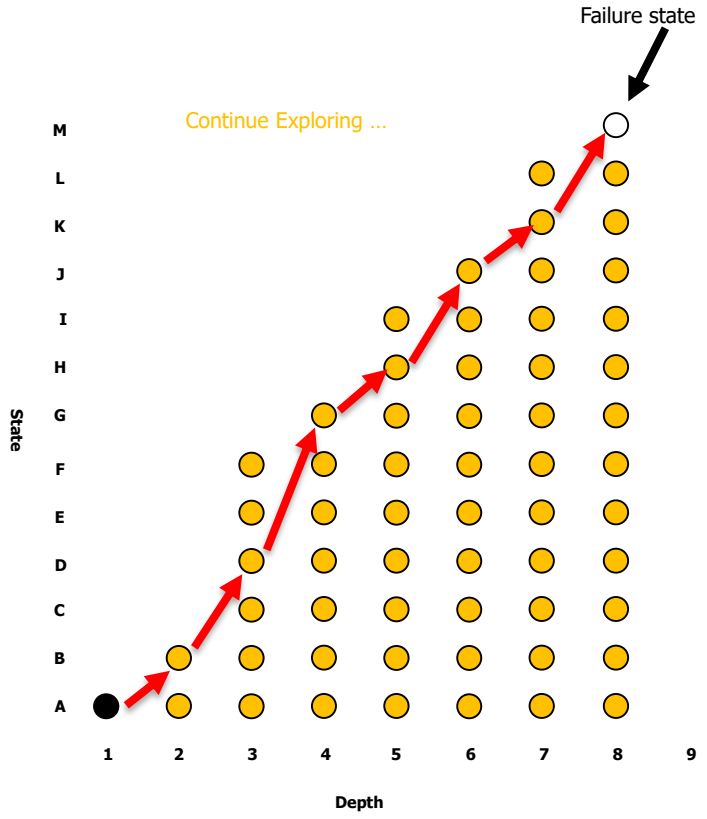
- **Formal visits DUT states as it walks through stimulus**
  - State = value of all registers/inputs



Formal State Space Exploration

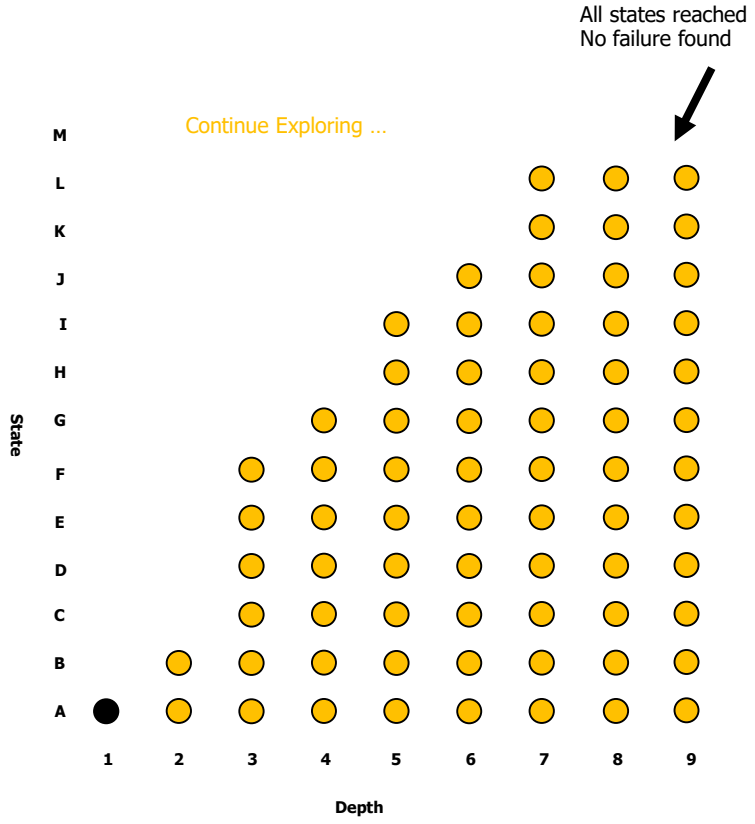
- **Formal visits DUT states as it walks through stimulus**
  - State = value of all registers/inputs
- **Formal may reach states that hit properties**
  - Assertion failures, cover hits
  - Converted to waveforms





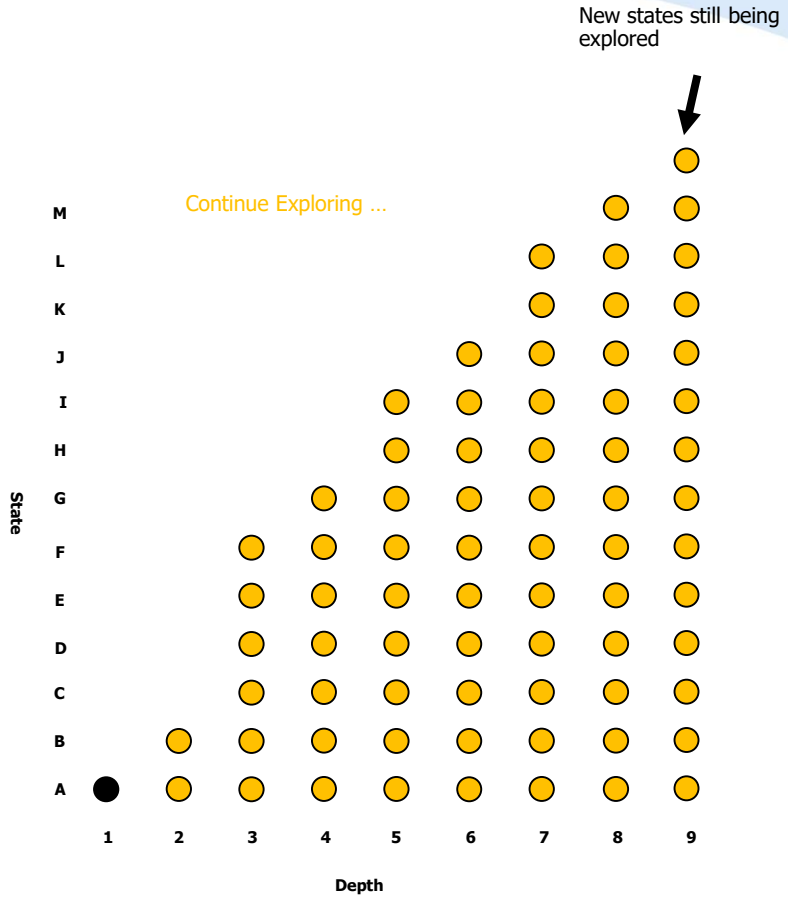
Formal State Space Exploration

- **Formal visits DUT states as it walks through stimulus**
  - State = value of all registers/inputs
- **Formal may reach states that hit properties**
  - Assertion failures, cover hits
  - Converted to waveforms
- **It may also explore all reachable states**
  - If no properties are hit, then they will never be hit!
  - "Full Proof"



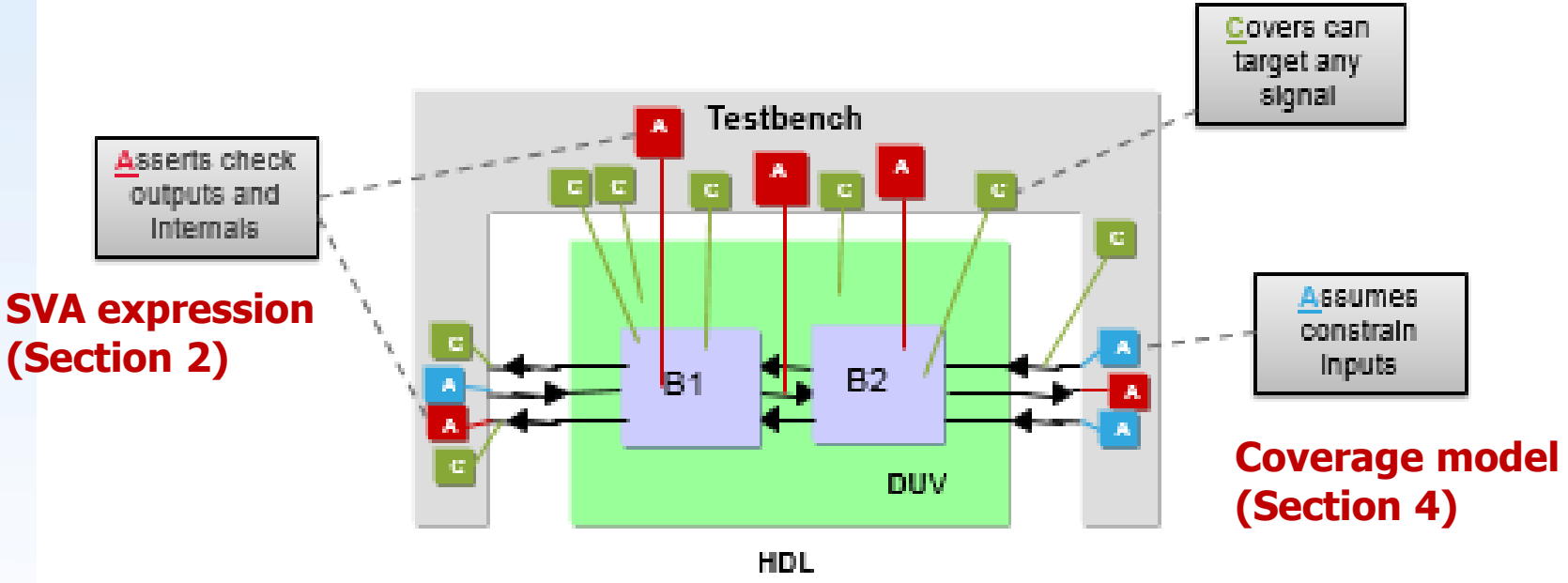
Formal State Space Exploration

- **Formal visits DUT states as it walks through stimulus**
  - State = value of all registers/inputs
- **Formal may reach states that hit properties**
  - Assertion failures, cover hits
  - Converted to waveforms
- **It may also explore all reachable states**
  - If no properties are hit, then they will never be hit
  - "Full Proof"
- **Or it may keep exploring new states until timeout**
  - Undetermined result



Formal Testbench

- Similar to sim testbench, in formal the testbench is also around the DUV



1. Input restrictions with "assume"
2. Checker property with "assert"
3. Stimulus coverage with "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

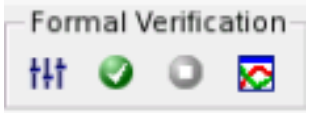
// c) FIFO cannot have full and empty asserted at the same time
```
ast_no_full_and_empty: assert property (!(full && empty));
```

// 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



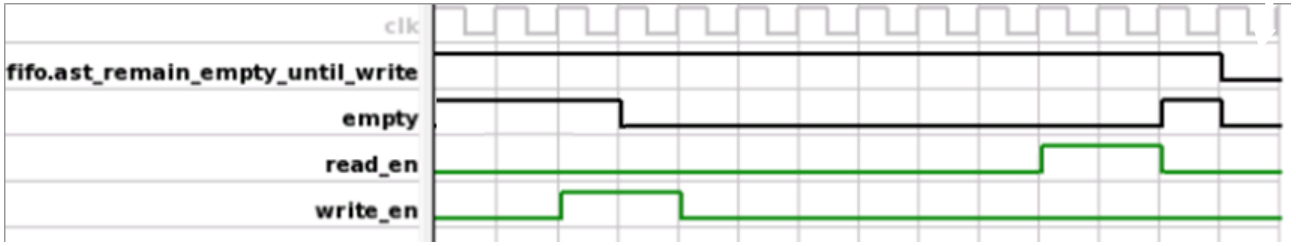
Full Proof: Impossible to violate this assertion

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



Benefits of Formal Analysis

- **Systematic method**
  - None or very little randomization
  - More deterministic
- **Less testbench effort required**
  - Formal testbench tends to be much simpler than sim testbench
- **Leads to higher quality**
  - Find bugs from a different angle: breadth-first search (formal) vs. depth-first search (sim)
  - Often reveals bugs that simulation would never catch
- **Improves productivity and schedule**
  - Can replace some block-level testbenches
  - Verification can begin prior to testbench creation and simulation

Outline

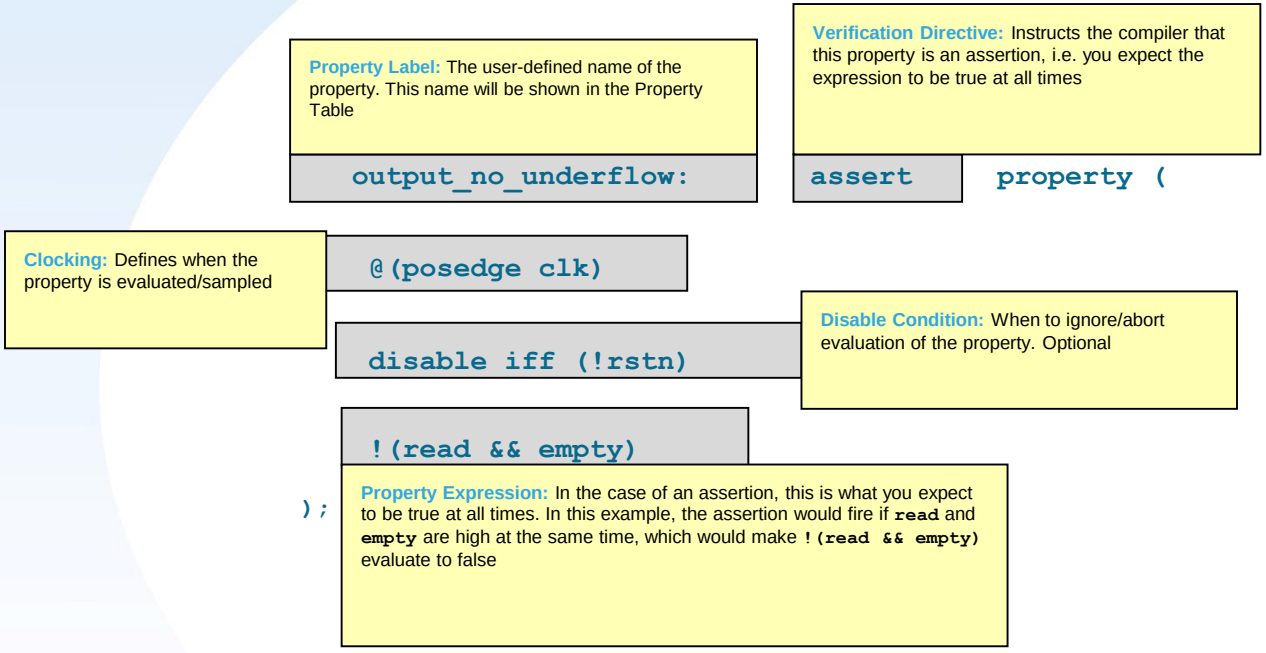
- Section 1. Introduction to Formal Analysis
- **Section 2. Introduction to SVA**
- Section 3. JasperGold – Setup
- Section 4. JasperGold – Formal Coverage Analysis
- Section 5. JasperGold – Assertion Based Verification IP & Scoreboard

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

SVA Syntax

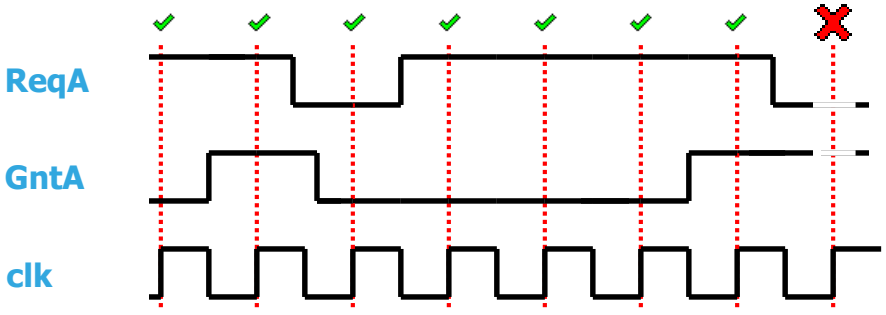
- **SVA properties are embedded in Verilog or SystemVerilog code**



SVA Example: Invariants

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

```
no_GntA_without_ReqA: assert property (
  @(posedge clk) disable iff (!rstn)
  !(GntA && !ReqA)
);
```

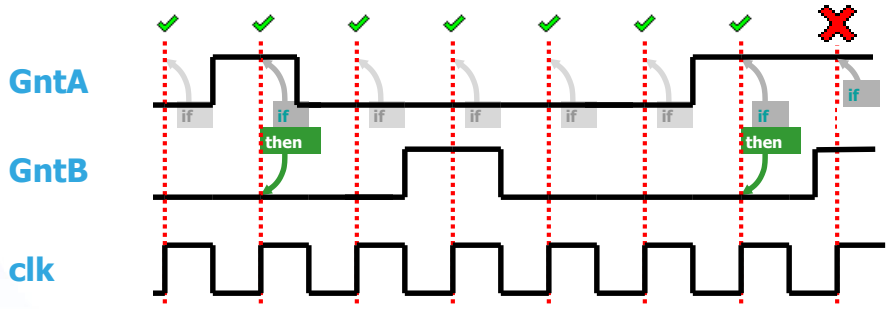


SVA Example: Implications

- **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"**
- **GntA |-> !GntB**

```
GntA_then_not_GntB: assert property (
  @(posedge clk) disable iff (!rstn)
  GntA |-> !GntB
);
```

Implication operator |-> expresses "if...then"



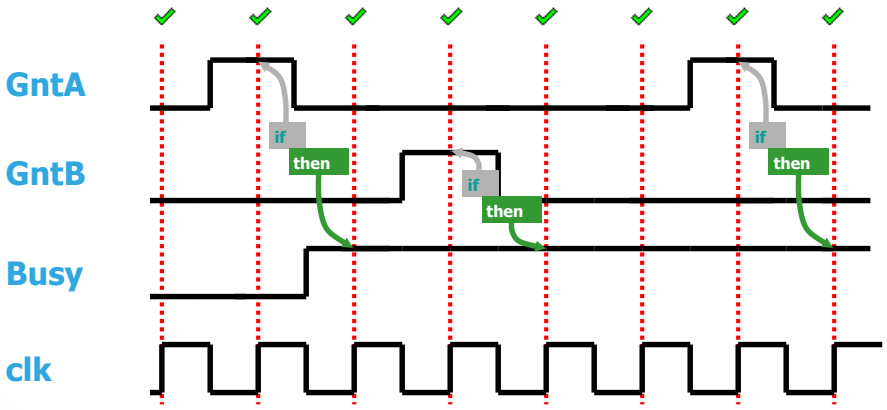
|-> and |=> ?

SVA Example: Multi-Cycle Implications

- Possible to delay checking consequence by one cycle
- e.g. "Grant is always followed by Busy"
- `(GntA || GntB) |=> Busy`

```
Gnt_followed_by_Busy: assert property (
  @(posedge clk) disable iff (!rstn)
  (GntA || GntB) |=> Busy
);
```

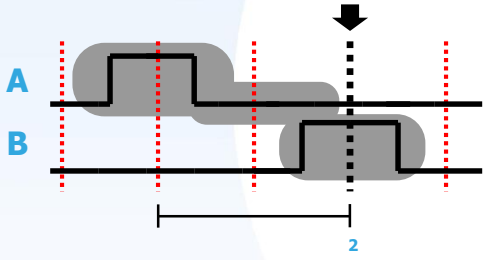
Next-cycle Implication operator |=> adds a one-cycle delay between "if" and "then"



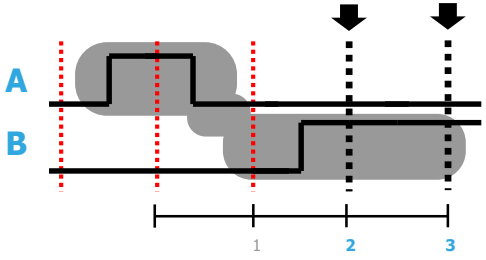
Sequences

- **Sometimes necessary to capture sequence of events in properties**
- **Most sequences are described using ## operator**

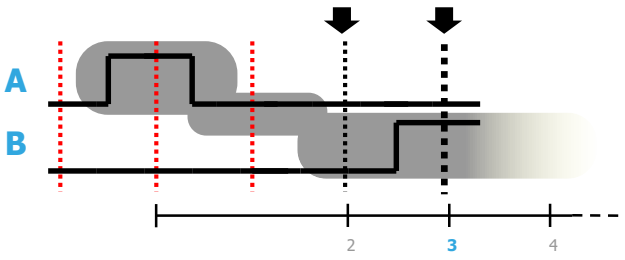
`A ##2 B`
"A happens then exactly 2 cycles later B happens"



`A ##[2:3] B`
"A happens then 2 to 3 cycles later B happens"



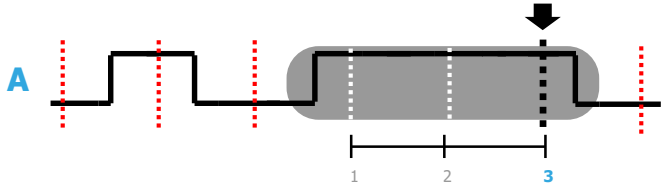
`A ##[2:$] B`
"A happens then 2 or more cycles later B happens"



Sequences

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

`A [*3]`
"A happens 3 times in a row

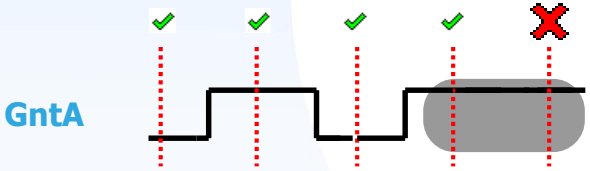


SVA Example: Sequences

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

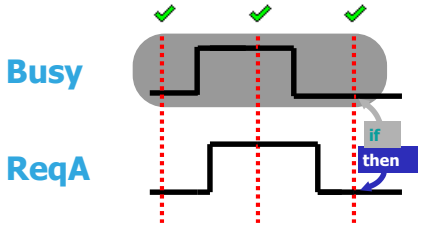
"Should never see two grants to A in successive cycles"

```
no_2_GntA: assert property (
  @(posedge clk)
  not (GntA [*2])
);
```



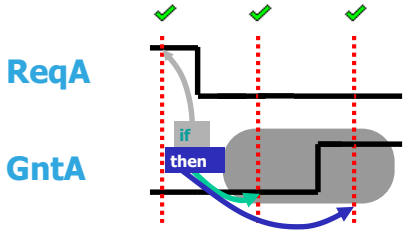
"Busy pulse should only happen if no request"

```
Busy_pulse: assert property (
  @(posedge clk)
  (!Busy ##1 Busy ##1 !Busy)
  |-> !ReqA
);
```



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

```
Req_to_Gnt: assert property (
  @(posedge clk)
  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 (@(posedge clk) $onehot0({GntA,GntB,GntC});`
`$countones` `$countzeros`	Returns the number of ones/zeros in the argument	*"Should never see more than 4 dirty lines"* `assert property (@(posedge clk) $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 (@(posedge clk) !Ready |=> $stable(Data));
$past	Return previous value of argument	"If active, then previous cycle command must not be IDLE" assert property (@(posedge clk) 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 (@(posedge clk) 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 (@(posedge clk) $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]

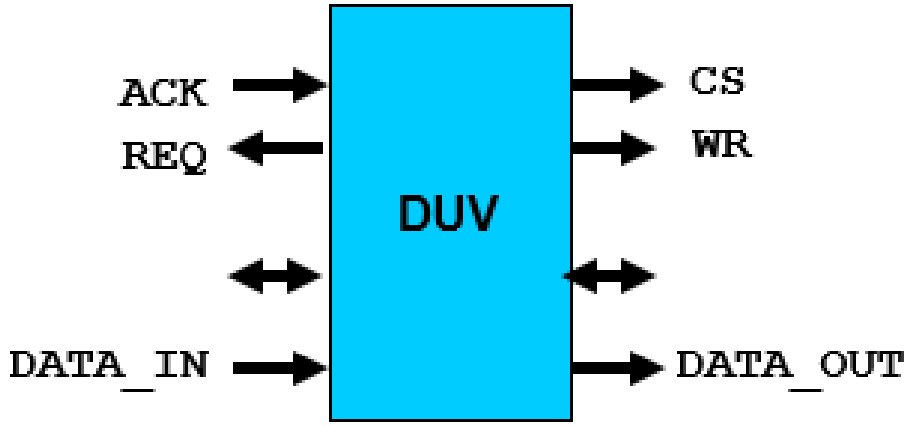
Combinatorial Functions
$onehot(a) $onehot0(b) $countones(c) $countzeros(d)

Temporal Functions
$stable(a) $past(b) $rose(c) $fell(d)

Functional Transaction Example

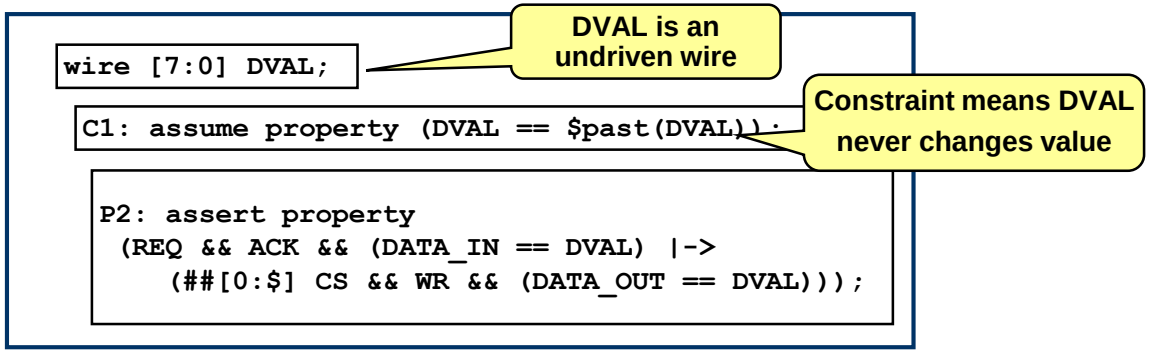
- **If a handshake transaction occurs, memory write transaction should eventually occur**
- **This is very powerful:**
  - Simple to express
  - Tests large part of design functionality

```
AST: assert property
 (REQ && ACK && (DATA_IN == 8'ha5) |->
    ##[0:$] CS && WR && (DATA_OUT == 8'ha5) );
```



Nondeterministic Constants

- **Conceptually formal tool chooses all possible values for DVAL**
- **The value chosen remains the same during the proof**
- **Nondeterministic constants have no meaning in simulation**
- **In simulation DVAL value will be deterministic**
  - Initial value determined by [System]Verilog LRM



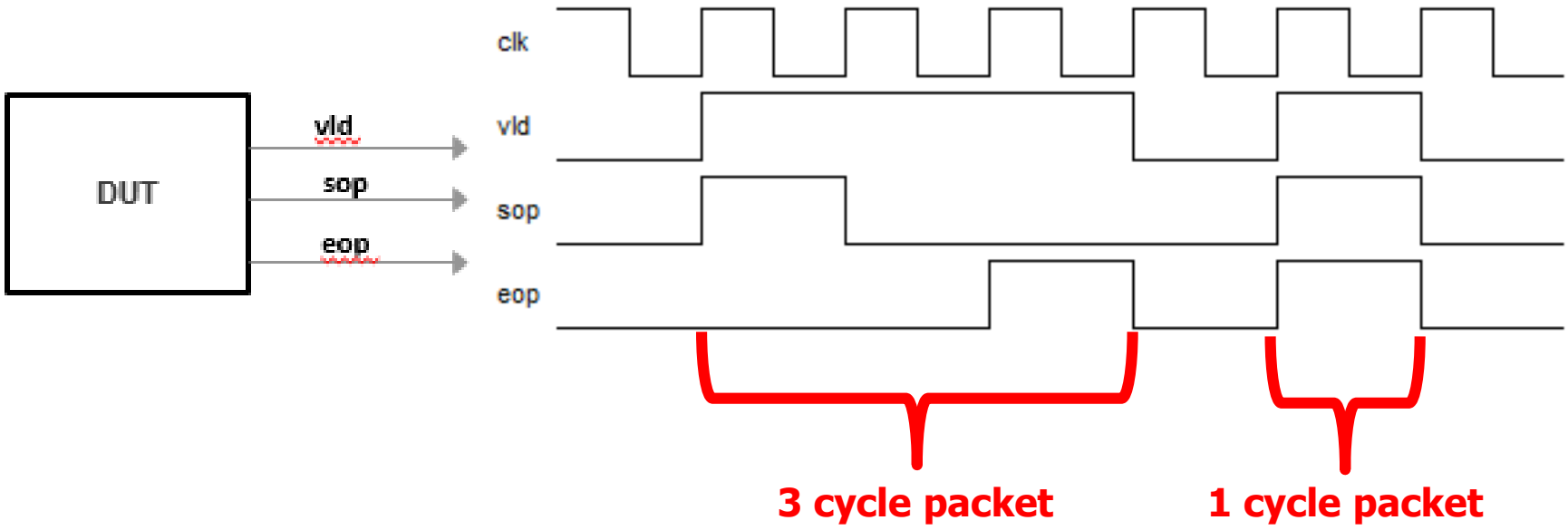
Glue Logic

- **Modeling complex behaviors, SVA may be complicated**
  - Using auxiliary logic to observe and track events
  - 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**
  - JasperGold does not care whether property is all SVA or SVA+glue logic
  - Recommendation is to choose based on clarity

Glue Logic Example: SOP/EOP Interface

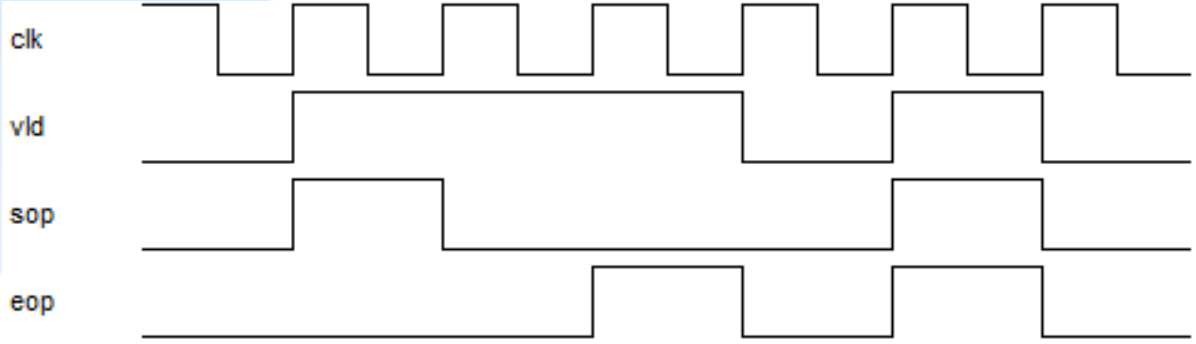
- **Notation:**
  - SOP: Start of Packet
  - EOP: End of Packet
- **Specification:**
  - No overlapping packets (SOP-EOP always in pairs)
  - Single-cycle packets allowed (SOP and EOP at the same cycle)
  - Continuous packet transfer (no vld holes between SOP-EOP)



Glue Logic 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 vld holes between SOP-EOP)



- Pure SVA version:

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

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

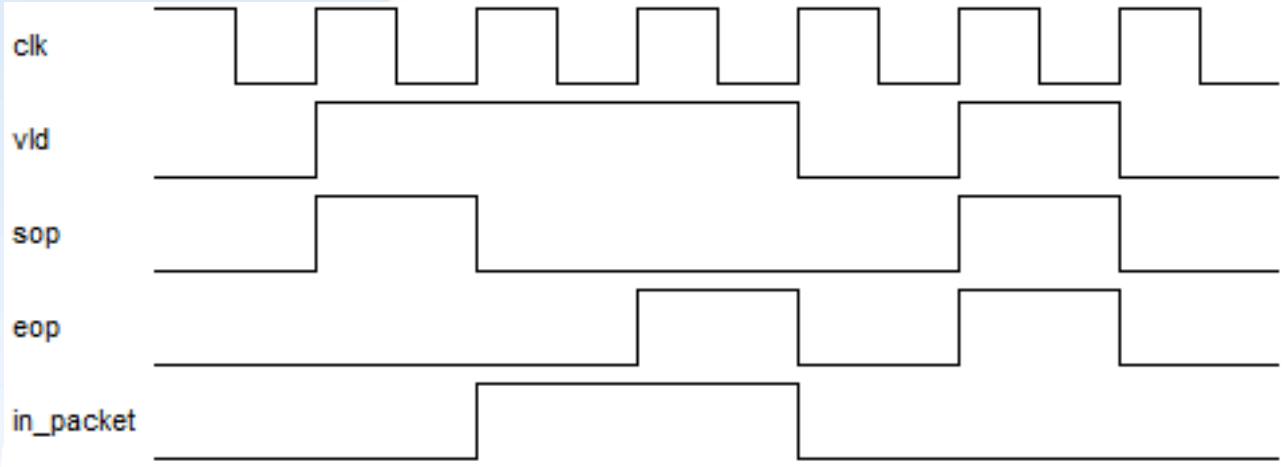
```
sop_first: assert property
            (vld && eop |-> sop_seen.ended)

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]))
)
```

Complex!

Glue Logic 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 vld holes between SOP-EOP)



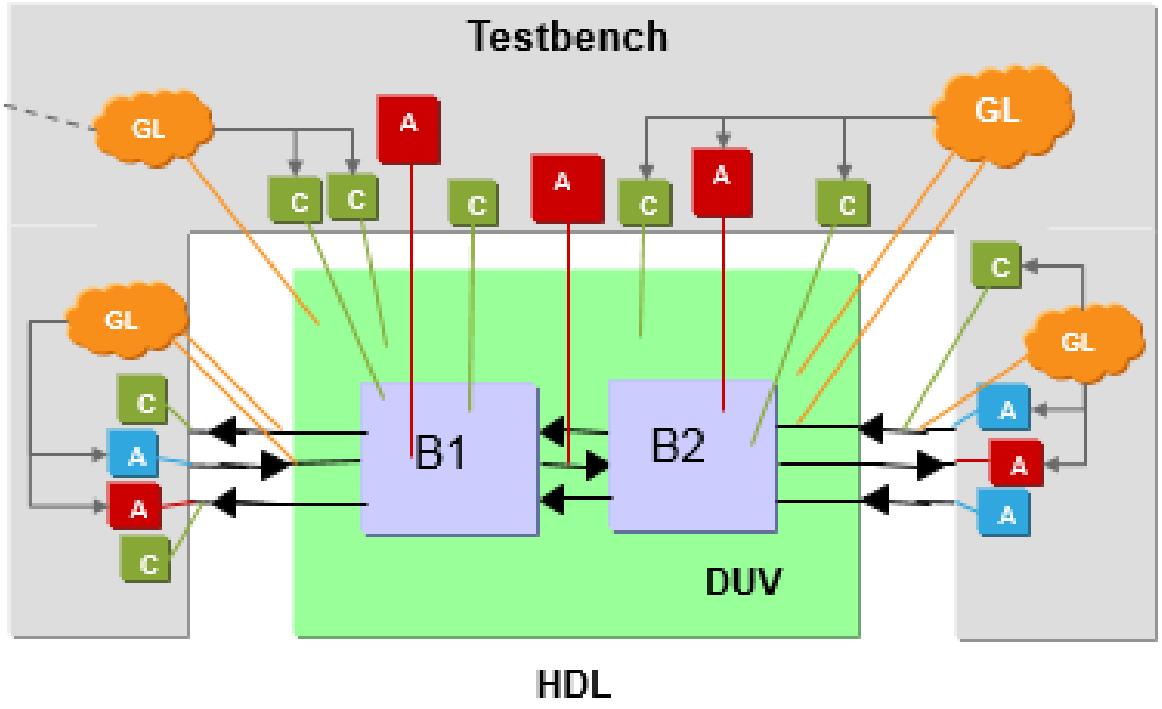
- Glue logic version:

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

no_holes_1: assert property( in_packet |-> vld) ;
no_holes_2: assert property( sop |-> vld) ;
```

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

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

Formal Testbench

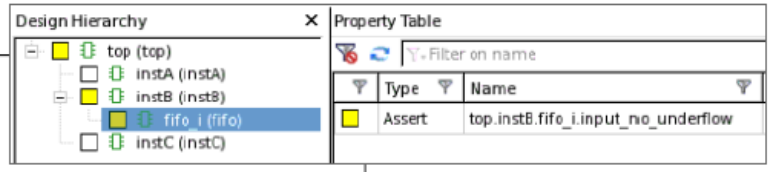
Glue Logic monitors design and feeds properties



SVA Embedding

Embed in RTL `fifo.v`

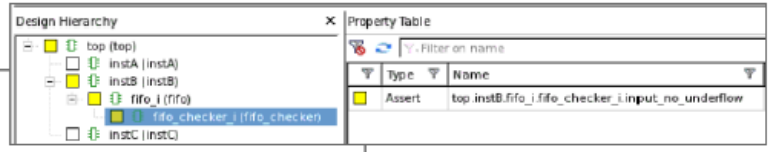
```
module fifo (input clk, rst_n, read, output empty, …)
  // Actual FIFO code:
  …
  // FIFO must not underflow
  input_no_underflow: assert property (@(posedge clk) !(read && empty));
endmodule
```



Best for designers

Use `bind` construct `fifo_bind.sv`

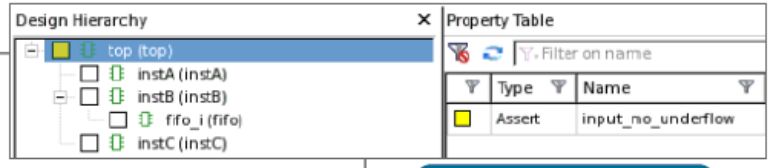
```
module fifo_checker (input clk, rst_n, read, empty);
  // FIFO must not underflow
  input_no_underflow: assert property (@(posedge clk) !(read && empty));
endmodule

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



Best for DV

Create in Tcl `jg_fifo.tcl`

```
analyze ...
elaborate ...
...
assert -name input_no_underflow {
  @(posedge clk) !(instB.fifo_i.read && instB.fifo_i.empty)
}
```



Best for quick experiments

Being Successful with SVA

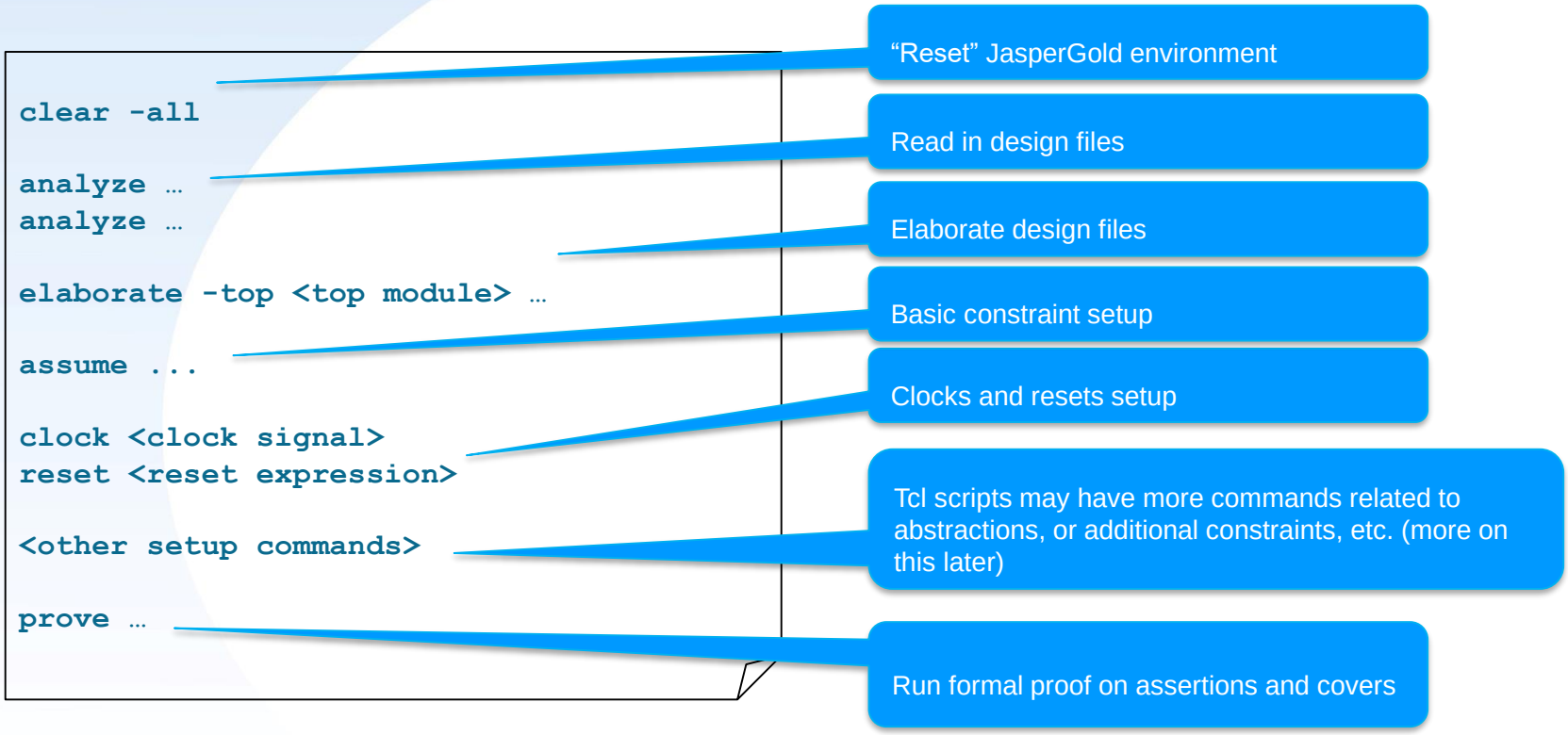
- **First, describe intent in your natural language, then code**
  - *"A and B should never be high at the same time"*
  - *"if X happens, then I should see Y within N cycles"*
  - *"either P or Q should be low if design is in state S"*
- **The key to learning SVA is to learn a small productive subset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

Outline

- Section 1. Introduction to Formal Analysis
- Section 2. Introduction to SVA
- **Section 3. JasperGold – Setup**
- Section 4. JasperGold – Formal Coverage Analysis
- Section 5. JasperGold – Assertion Based Verification IP & Scoreboard

JasperGold Flow Overview

- **A typical Tcl script for JasperGold looks like this:**



Outline

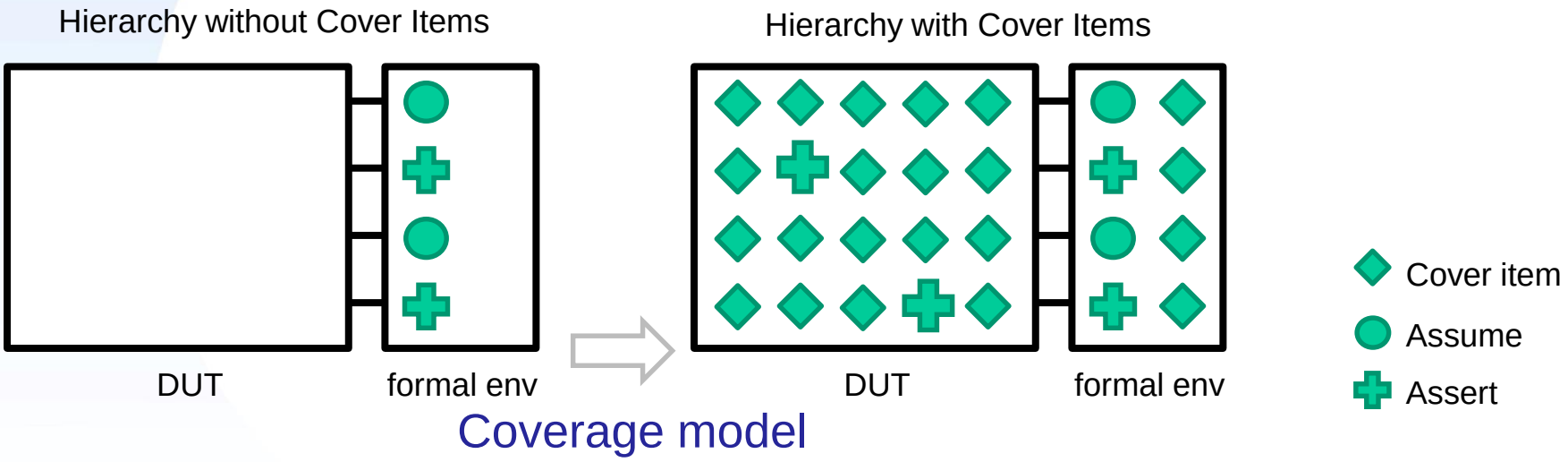
- Section 1. Introduction to Formal Analysis
- Section 2. Introduction to SVA
- Section 3. JasperGold – Setup
- **Section 4. JasperGold – Formal Coverage Analysis**
- Section 5. JasperGold – Assertion Based Verification IP & Scoreboard

Why coverage?

- ✓ **Coverage is about:**
  - Confidence of verification effort
  - Ability to track progress over time
  - Credibility of checker's robustness
- ✓ **Coverage is an accepted source of metrics for showing progress and signoff**
  - Often measured against a verification plan
  - What your boss is care about!!

Introduction to coverage models

- ✓ **Coverage models**
  - Indicates "targets" that can be "measured" during analysis
  - These individual targets are called "Cover Items"
  - How to create them?



Coverage model

Coverage Models

- **Function Coverage**
  - Property related covers (Lab09)
  - Covergroups (Lab09)

Pros:
Realizable!
Cons:
Corner case!
Time consuming!

- **Code Coverage**
  - Branch
  - Statement
  - Expression

Pros:
Automatically gen.!
Fully covered
Cons:
False alarm issue

Functional coverage – Property

- **Property's related covers**
  - Describe specific states, conditions, or sequences to be verified
  - SVA cover property – user defined

```
cover 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

Functional coverage - Covergroups

- **Covergroups are supported**
  - Convenient for creating comprehensive coverage involving variable values/transitions/crosses

To enable: `elaborate -extract_covergroups`

Property Table

Filter on name

Type	Name
Cover	cover_item_covergroup_cg_inst_cg1_coverpoint_packet_type_bin_auto[TYPE1]
Cover	cover_item_covergroup_cg_inst_cg1_coverpoint_packet_type_bin_auto[TYPE0]
Cover	cover_item_covergroup_cg_inst_cg1_coverpoint_has_crc_bin_auto[1]
Cover	cover_item_covergroup_cg_inst_cg1_coverpoint_has_crc_bin_auto[0]

```
enum logic {TYPE0, TYPE1} packet_type;
logic has_crc;

covergroup cg @(posedge clk);
  packet_type: coverpoint my_packet.packet_type;
  data_length: coverpoint my_packet.data_length;
  has_crc: coverpoint my_packet.has_crc;
endgroup
```

JG automatically creates one property (bin) for each possible value of the signal

Code coverage models – Branch

- **Branch Model**
  - Enumerates all branches in the code as cover items
  - Constructs: if/else, case, Verilog "?:"
  - Optionally creates cover items for empty else
  - Cover items are given a unique name and ID#:
    `_automatic_coveritem_branch_condition__if_stmt_then__47___B__173_0_1`

- **Example**

```
if (!rstN)
        trans_started <= 1'b0;      } Branch
else if (new_trans)
        trans_started <= 1'b1;      } Branch
else
        trans_started <= 1'b0;      } Branch
```

– Three branch cover items- 2 "then" and 1 "else"

Code coverage models – Statement

- **Statement Model**
  - Enumerates all statements in code as cover items
  - Includes assignments and conditional expressions
  - Cover items are given a unique name and ID#:

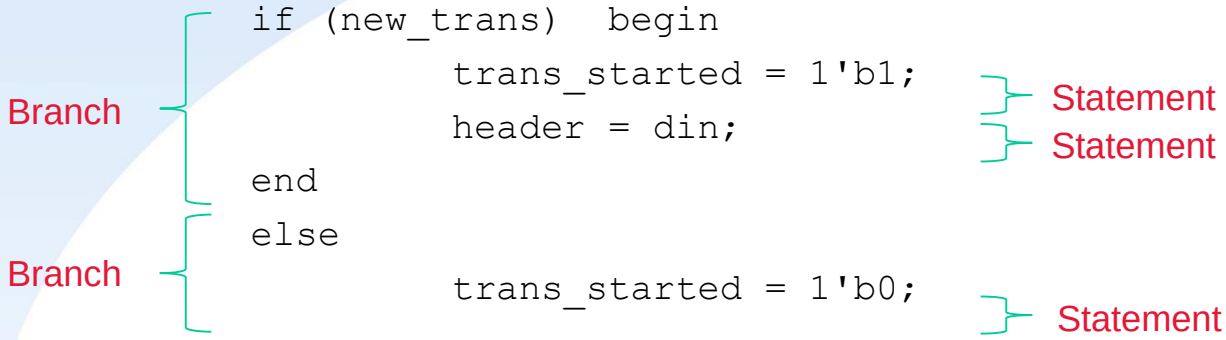
    `_automatic_coveritem_stmt_condition__blocking_assignment__107___S__217_0_82`

- **Example**

```
if (!rstN)
        trans_started <= 1'b0;      Statement
else if (new_trans)
        trans_started <= 1'b1;      Statement
else
        trans_started <= 1'b0;      Statement
```

Code coverage models – Branch/Statement

- **Example**



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	Expression
1	0*	0	Expression
1*	1*	1	Expression

* controlling operand

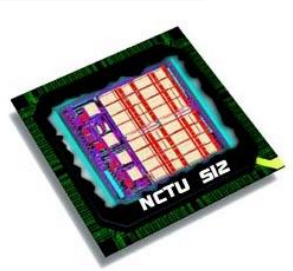
Control model considers A=0 and B=0 redundant with these.

Code vs. Functional Coverage Model

- **Functional coverage**
  - Possible to represent all meaningful design functionality
  - Implements the verification plan – what needs to be verified
  - Noise-free – nothing is don't-care
  - Requires planning, coding, and debug
  - May be incomplete due to human error
- **Code coverage**
  - Easy to generate automatically
  - 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 – "don't-care" or duplicate covers

Measuring Coverage

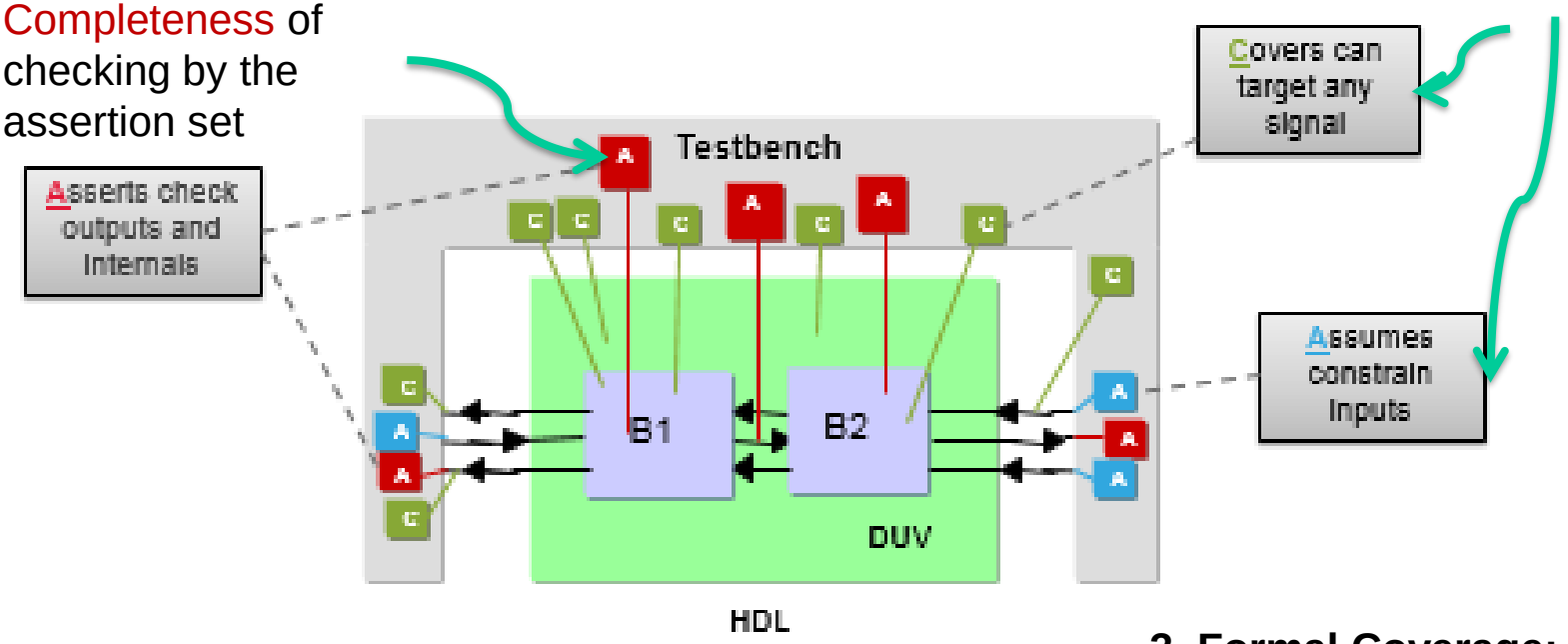
Formal Coverage Analysis



Measuring Coverage - Types

1. Stimuli Coverage:
Reachability analysis applies to all coverage models

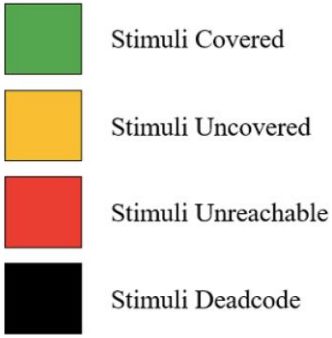
2. Checker Coverage:
Completeness of checking by the assertion set



3. Formal Coverage:
Consolidation of Stimuli and Checker Coverage results

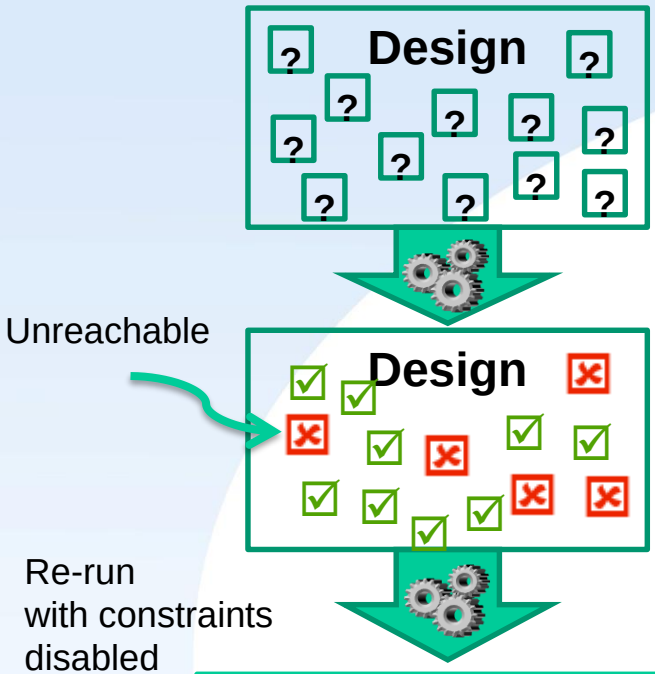
Stimuli Coverage

- **What code or functionality is reachable by the formal testbench(Stimuli)?**
- **Reachability analysis applies to all coverage models**
  - Formal engines attempt to find the stimuli necessary to "hit" a cover
  - Result is **Covered**, **Uncovered**, **Unreachable**, or **Deadcode**



- **Unreachable** status
  - Formal engines have determined cover is unreachable with constraints (assumes) applied => overconstrained
- **Deadcode** status
  - Formal engines have determined cover is still unreachable even if remove constraints (assumes) applied => design problem => dead code
    - Any covers initially found unreachable are automatically re-run with constraints disabled to determine if they are deadcode

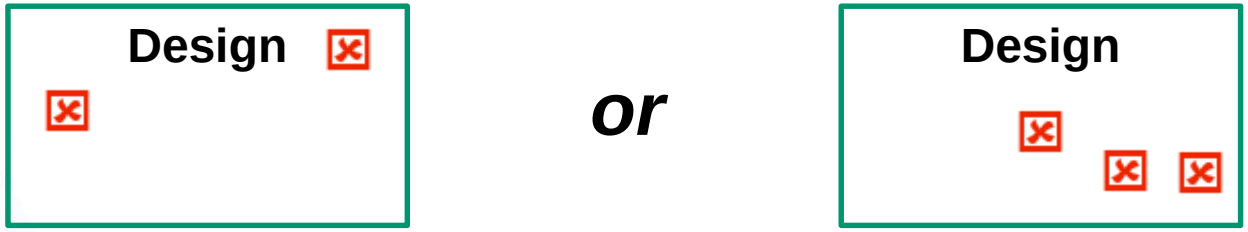
Stimuli Coverage



- Engines attempt to hit all cover items
- Formally proves items that are unreachable

Categorization of unreachable items:

1. Dead code: impossible for any stimulus to hit
2. Environment has overconstrained stimulus



Dead Code

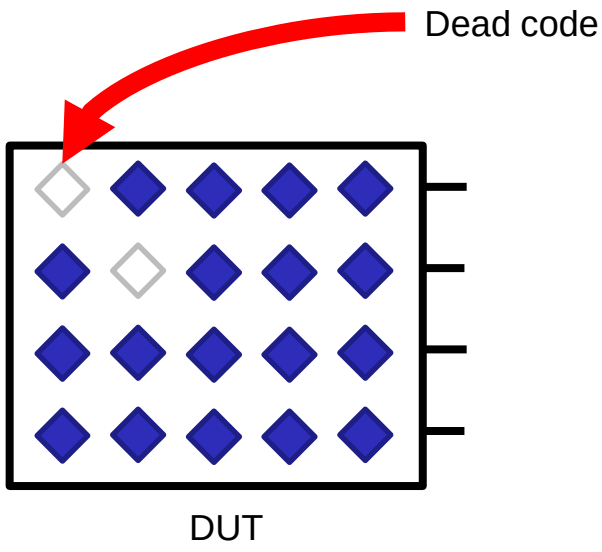
1) Designer Investigates
2) Fix, or waive from coverage

Overconstrained Stimulus

1) Overconstraint identified
2) Fix, or waive from coverage report

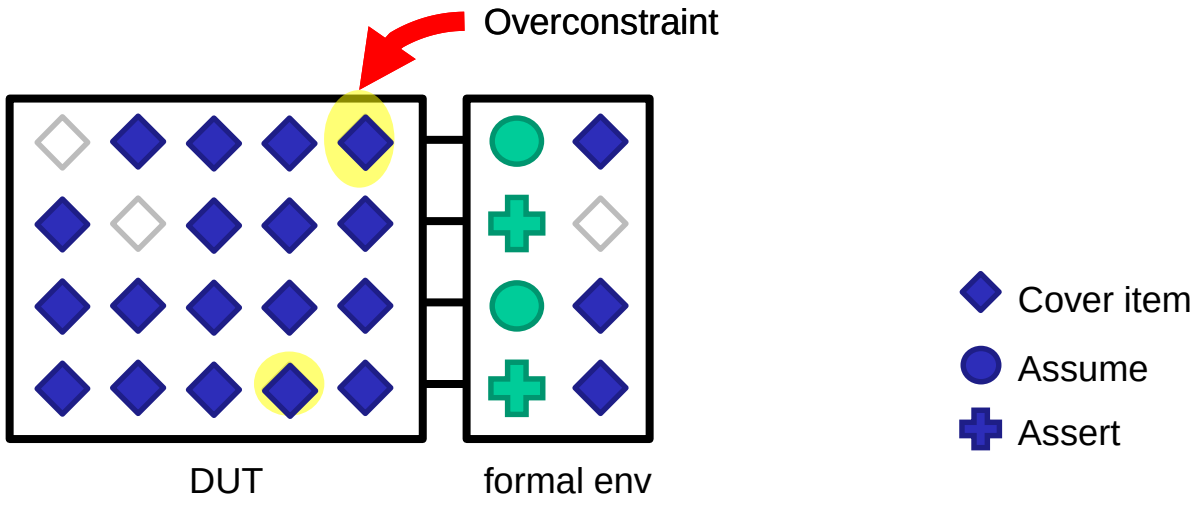
Stimuli Coverage

- **Cover Items that are unreachable without the formal environment's influence are dead code**
- **Actions:**
  - Designer Investigates
  - Fix, or waive from coverage



Stimuli Coverage

- Cover Items that become unreachable when formal environment is present are overconstraints
- Overconstraints reduce the scope of the verification environment and may mask bugs



Stimuli Coverage

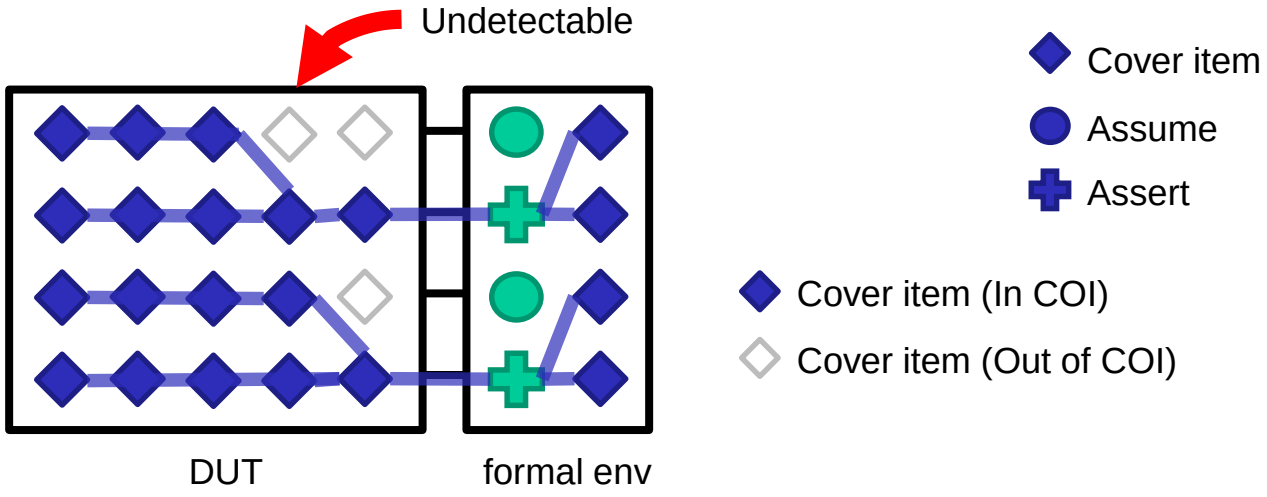
- **Classification of the environment regarding constraints**

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

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

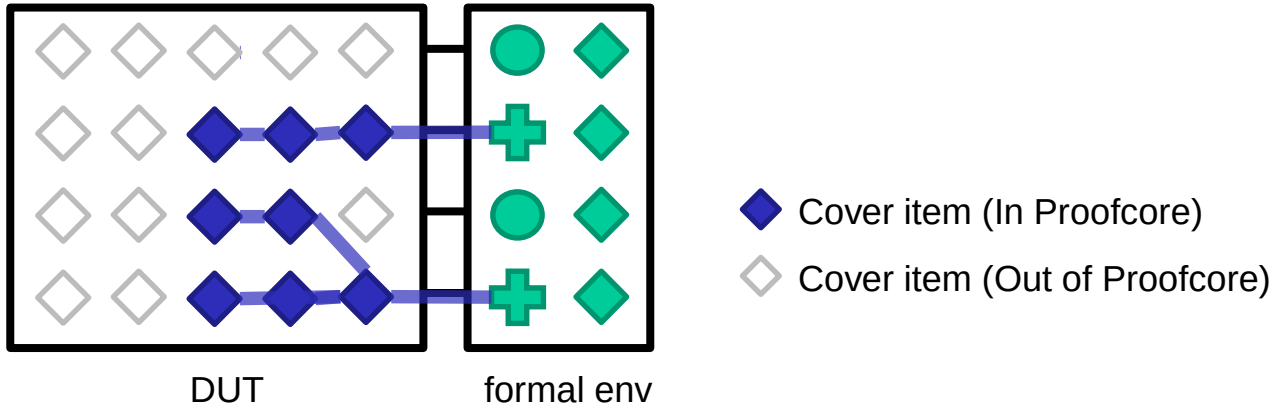
Cone-Of-Influence (COI) Coverage

- **Each assertion affected by some cover items**
  - The region will be called Cone-Of-Influence (COI)
- **Finds the union of the all assertion COIs**
  - The remaining Out of COI cover items indicate holes in the assertion set – code that is not checked by any asserts
  - Fast measurement – no formal engines are run



Proof Coverage

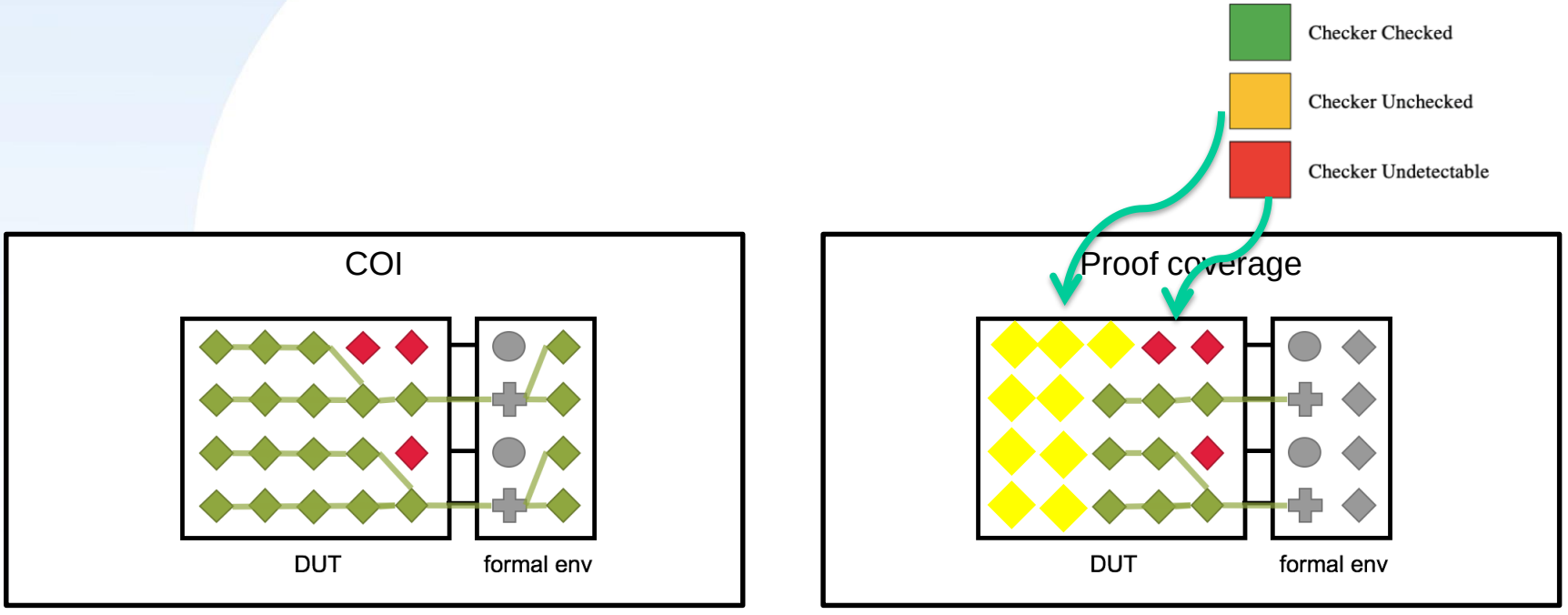
- **Find the region cannot truly influence assertion status**
  - Example:
  - assign a = b+c; } COI
  - assign m = a; } coverage } Proof coverage
  - **assert** property (@(posedge clk) a == m);
- **Represents the portion of the design verified by formal engines**
  - Subset of the COI – COI represents the maximum potential of proof coverage



Proof Coverage

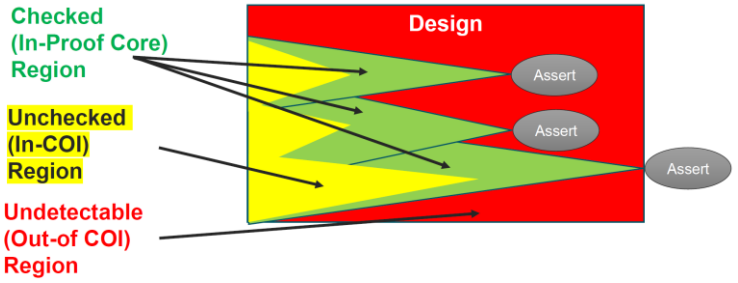
Proof vs. COI analysis

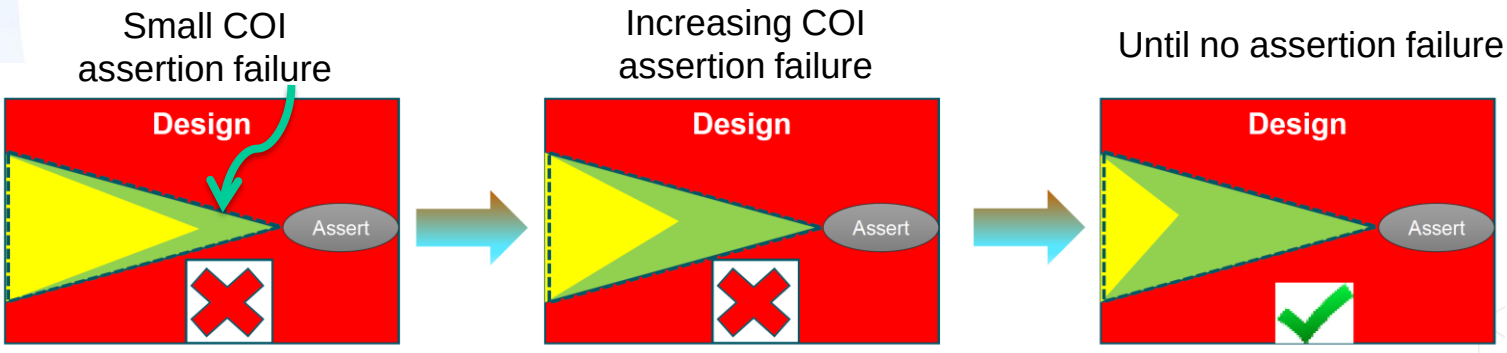
- **Proof coverage is a subset of the COI**
  - COI represents the maximum potential of proof coverage
- **COI doesn't require a proof to take place, while proof coverage does.**



Proof Coverage

- **Identify more unchecked code than COI measurement, with greater tool effort**
  - Slower measurement than COI – requires running formal engines
- **Iterative process of changing assertion-relevant design region**
  - Starts with small subset of COI
  - Engines check for assertion failure
  - Subset of COI is increased until no failure is found
  - This subset when a full or bounded proof is established is the "proof core"



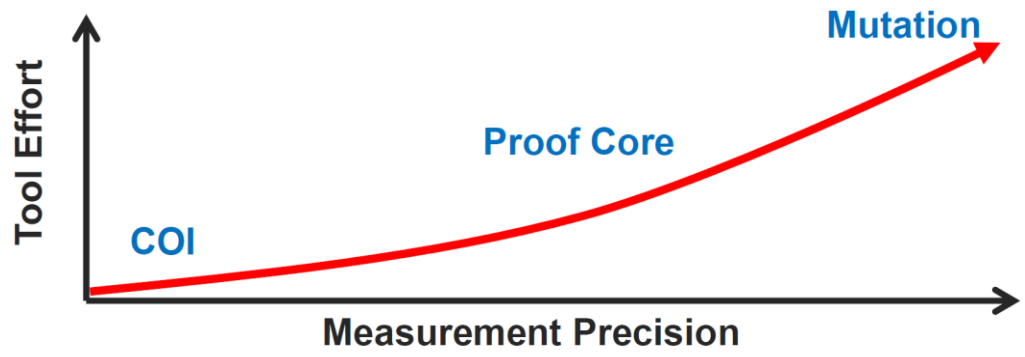


Formal Coverage Analysis

- **Formal testbench analysis**
  - Stimuli Coverage: Reachability by formal environment
    - Identifies overconstraints or dead code
  - COI Coverage: code within the cone-of-influence of assertions
    - Identifies holes in the set of assertions
  - Proof Coverage: design verified after formal proofs
    - Identifies areas of the design not verified by full or bounded proofs

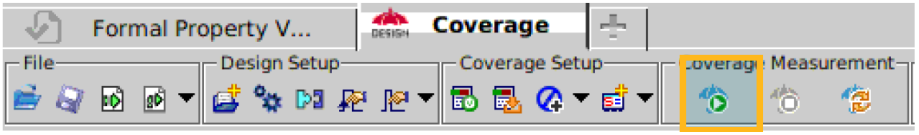
Stimuli Coverage

Checker Coverage

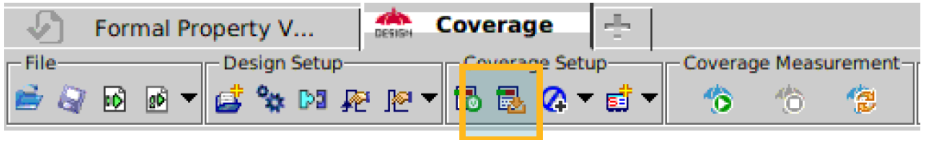


Measuring Coverage

- ✓ "check_cov –measure" automates running and collecting selected coverage types

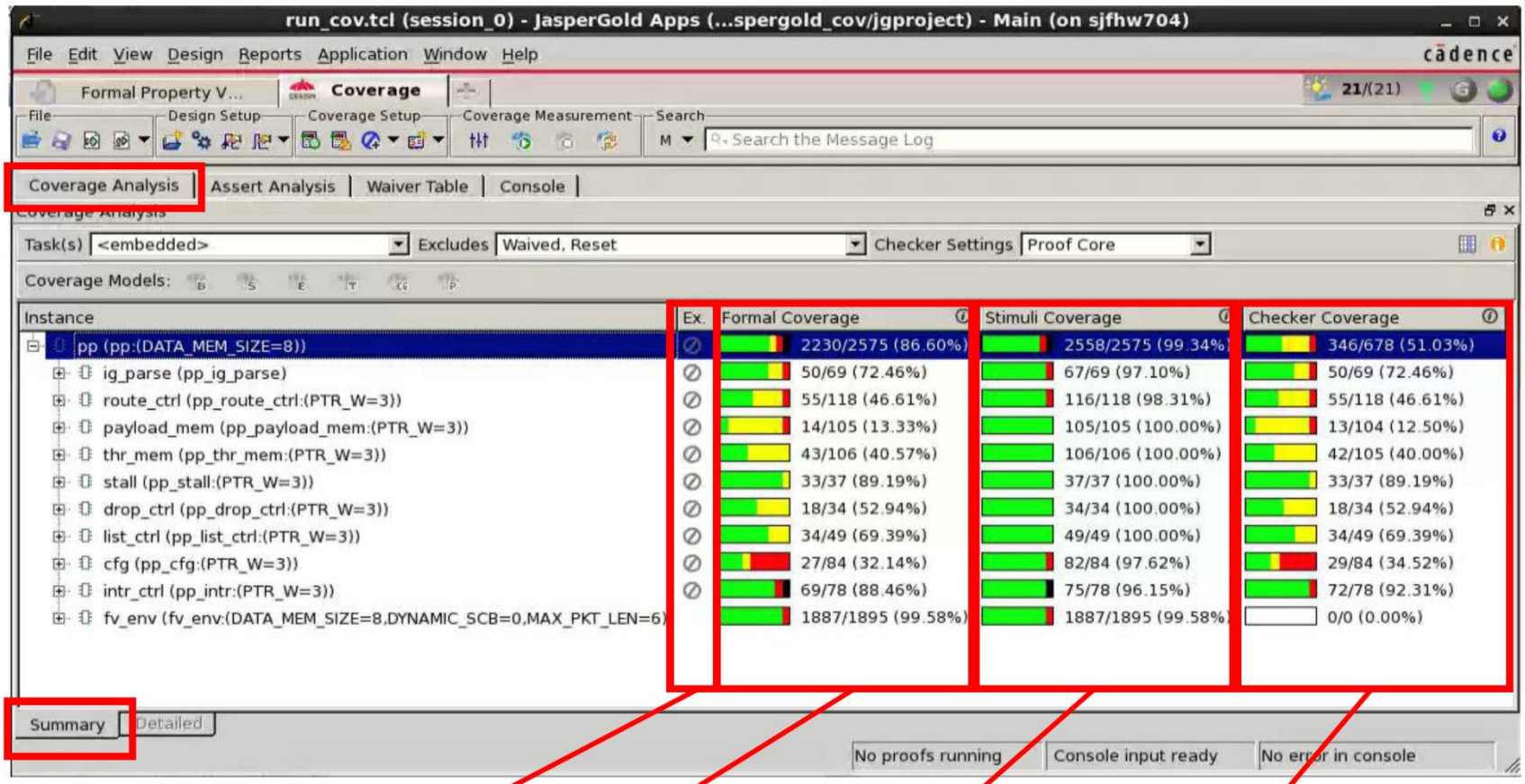


- ✓ GUI button or check_cov–report command can be used to generate these reports



- ✓ Coverage Summary view
  - Hierarchical Formal, Stimuli, and Checker Coverage data for entire design
  - Navigation to detailed coverage views

Coverage Analysis – Summary View



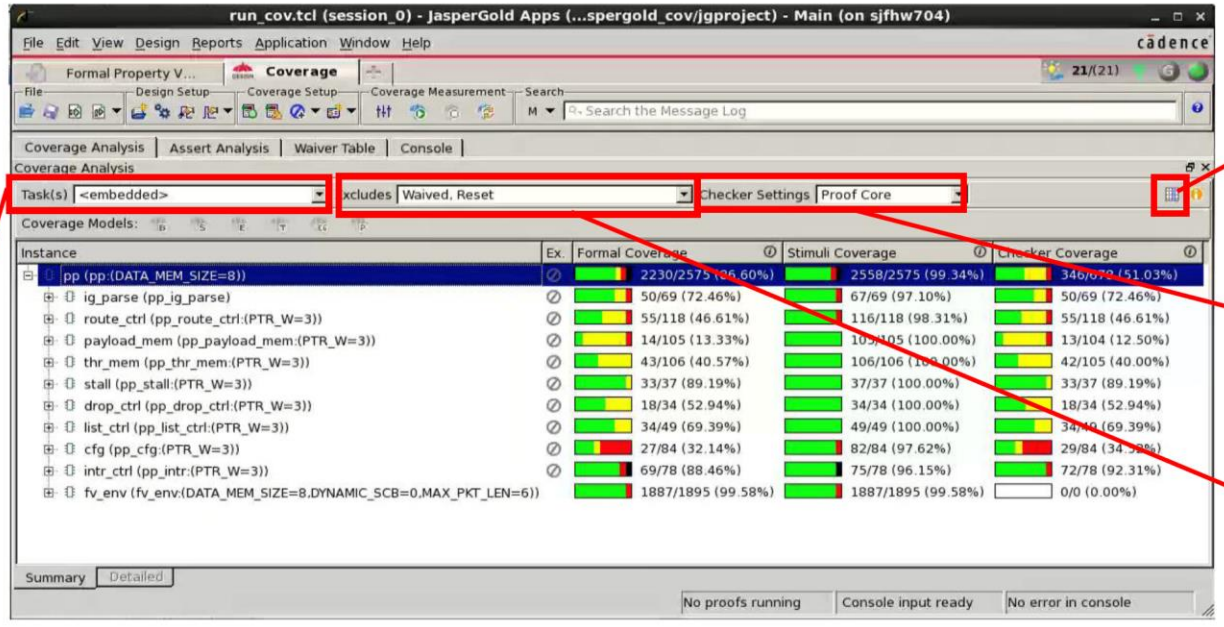
Exclusion Indicator

Formal coverage – combines stimuli and checker

Stimuli coverage- formal testbench correctness

Checker coverage – assertion completeness

Summary View Selections



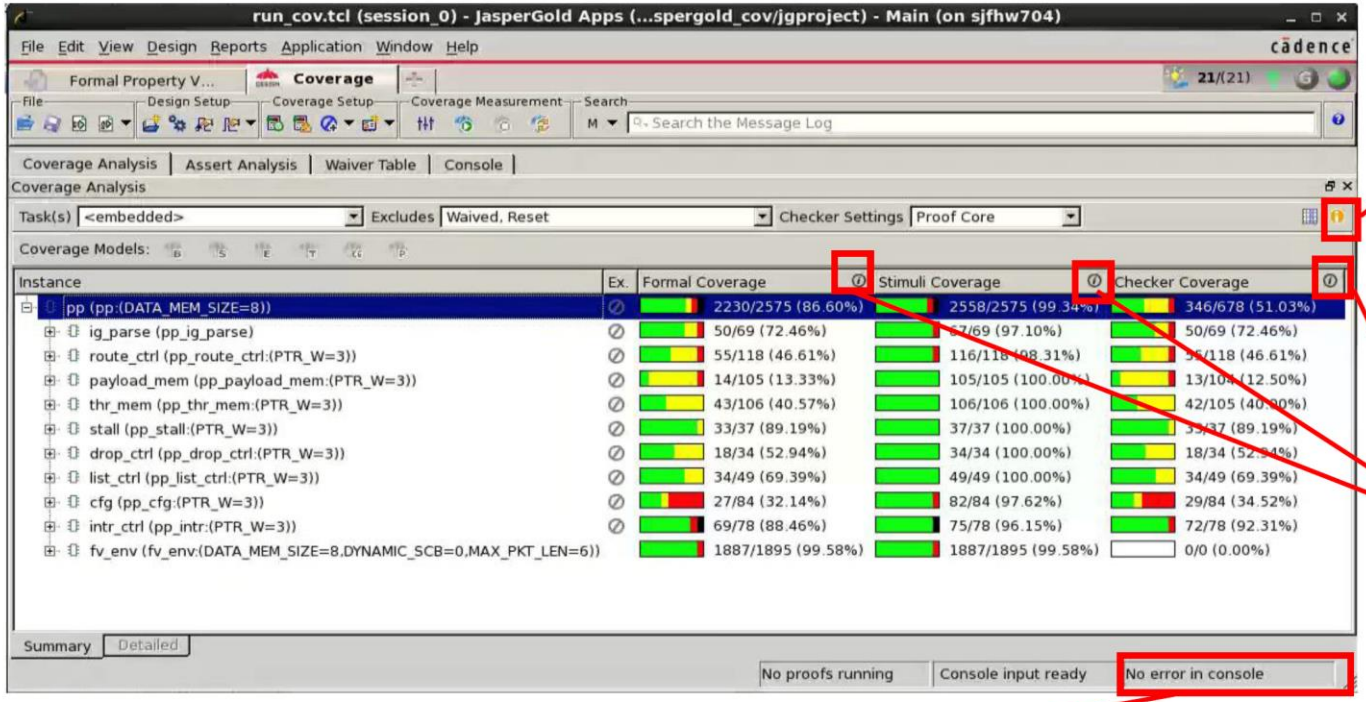
Task Selection – can select/merge multiple tasks

Add additional metrics to summary

Checker coverage criteria – COI or Proof Core or Mutation

Exclude Cover Items

Summary View Help



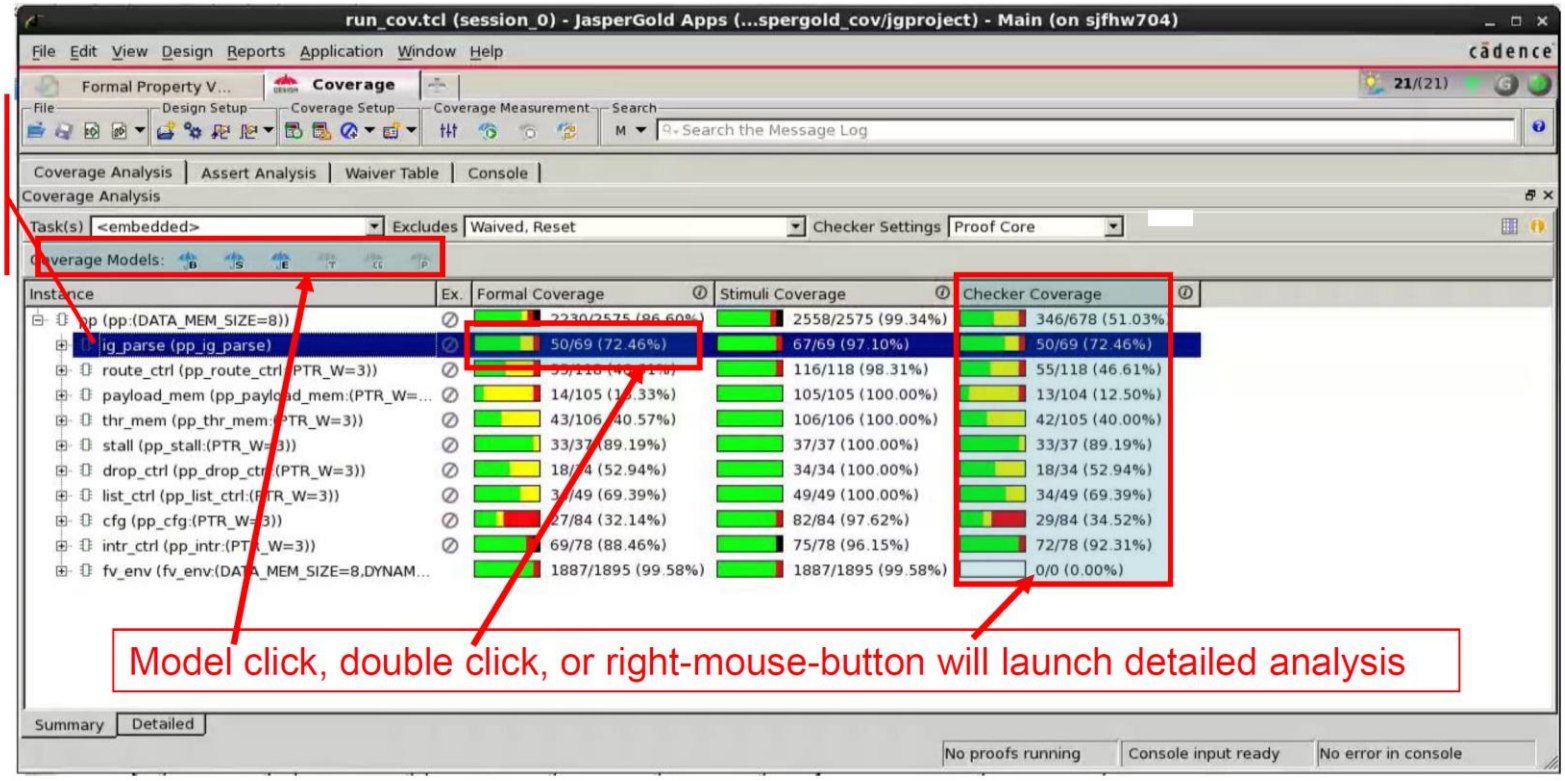
Click for step-by-step instructions

Info Buttons Provide Color Coding Key

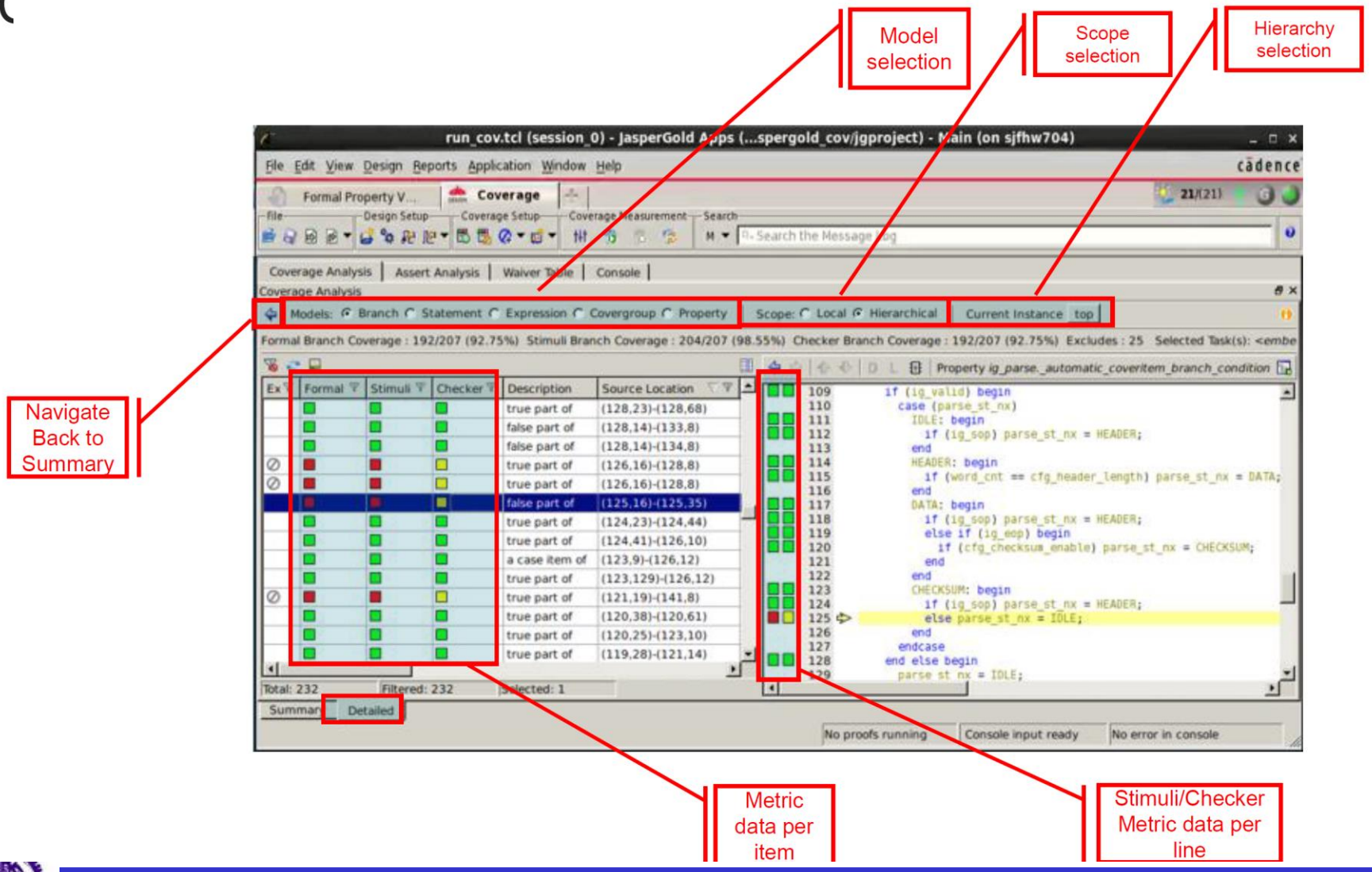
Indicates whether an error has occurred – click takes to console

Launching Detailed Analysis from Summary

Click to select an instance in the hierarchy



Coverage Detailed View



Outline

- Section 1. Introduction to Formal Analysis
- Section 2. Introduction to SVA
- Section 3. JasperGold – Setup
- Section 4. JasperGold – Formal Coverage Analysis
- **Section 5. JasperGold – Assertion Based Verification IP & Scoreboard**

Motivation

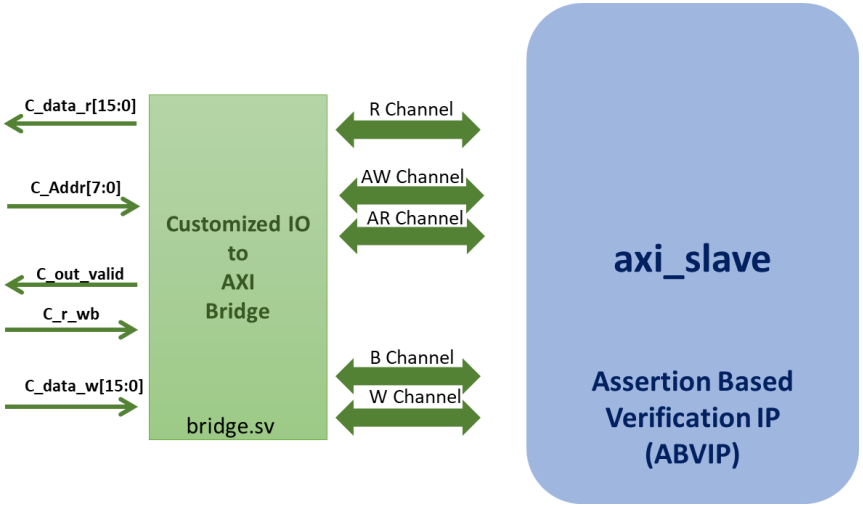
- Verifying a protocol and analyzing its completeness is a key challenge for engineers these days.
- The Assertion Based Verification Intellectual Properties(ABVIPs) are a comprehensive set of checkers and RTL that check for protocol compliance. (e.g. ARM AMBA AXI based systems and designs)

Setup Options

- **Instantiating ABVIP Components in a Top-Level Module**
  (In this lecture)
- **Connecting AXI ABVIP with DUV using bind Directive**

Example: Top-Level Module Instantiation

```
module top;

  parameter ADDR_WIDTH = 8;
  parameter DATA_WIDTH = 32;

  INF  inf(SystemClock);
  bridge    dut_b(.clk(SystemClock), .inf(inf.bridge_inf) );

    axi4_slave slave (
        .aclk           (SystemClock),
        .aresetn        (inf.rst_n),
        .awaddr         (inf.AW_ADDR),
        .awvalid        (inf.AW_VALID),
        .awready        (inf.AW_READY),

        .wdata          (inf.W_DATA),
        .wvalid         (inf.W_VALID),
        .wready         (inf.W_READY),

        .bresp          (inf.B_RESP),
        .bvalid         (inf.B_VALID),
        .bready         (inf.B_READY),

        .araddr         (inf.AR_ADDR),
        .arvalid        (inf.AR_VALID),
        .arready        (inf.AR_READY),

        .rdata          (inf.R_DATA),
        .rvalid         (inf.R_VALID),
        .rready         (inf.R_READY)

    );

    defparam slave.ADDR_WIDTH             = ADDR_WIDTH;
    defparam slave.DATA_WIDTH             = DATA_WIDTH;
```



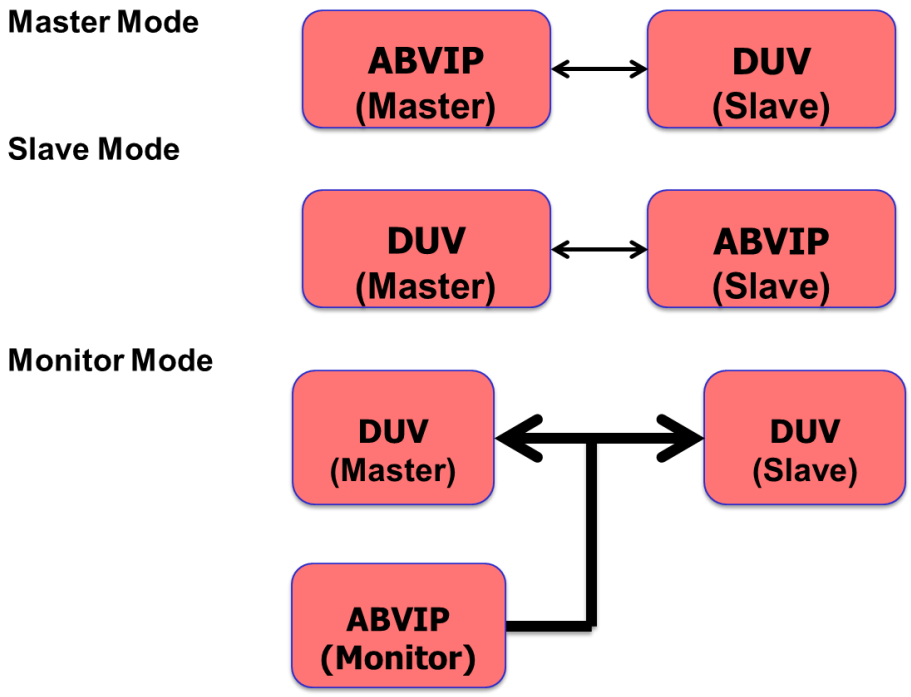
axi4_slave

- ✓ **A lot of assertions**

```
if (XCHECKS_ON) begin: genXChks

// ARM IHI 0022F: Section A3.1 on pg A3-41
// ARM IHI 0022F: Section A3.2 on pg A3-42
// Description     : No AXI 4 signal should have X or Z value when the corresponding valid signal is asserted
// Critical Signals: Assertion specific. Add signals specified in assertion definition

    if (!WRITEONLY_INTERFACE) begin: genXChksRDInf
        // Read Control Channel (Master driven signals)
        master_xcheck_arvalid  : assume property (aresetn |-> !$isunknown(arvalid))
          else $display ("%0t: arvalid should not have X or Z value when reset is de-asserted; ARM IHI 0022F: Section A3.1.2 on pg A3-40",$time);
        master_xcheck_araddr   : assume property (arvalid |-> !$isunknown(araddr))
          else $display ("%0t: araddr should not have X or Z value when arvalid is asserted; ARM IHI 0022F: Section A3.2.2 on pg A3-42,43",$time);
        master_xcheck_arprot   : assume property (arvalid |-> !$isunknown(arprot))
          else $display ("%0t: arprot should not have X or Z value when arvalid is asserted; ARM IHI 0022F: Section A3.2.2 on pg A3-42,43",$time);

        // Read Control Channel (Slave driven signals)
        slave_xcheck_arready   : assert property (arvalid |-> !$isunknown(arready))
          else $display ("%0t: arready should not have X or Z value when arvalid is asserted; ARM IHI 0022F: Section A3.2.2 on pg A3-42,43",$time);
```

Configuration ABVIP

- **Three Modes of ABVIP**
  - Master
  - Slave (Main Focus in this lecture)
  - Monitor



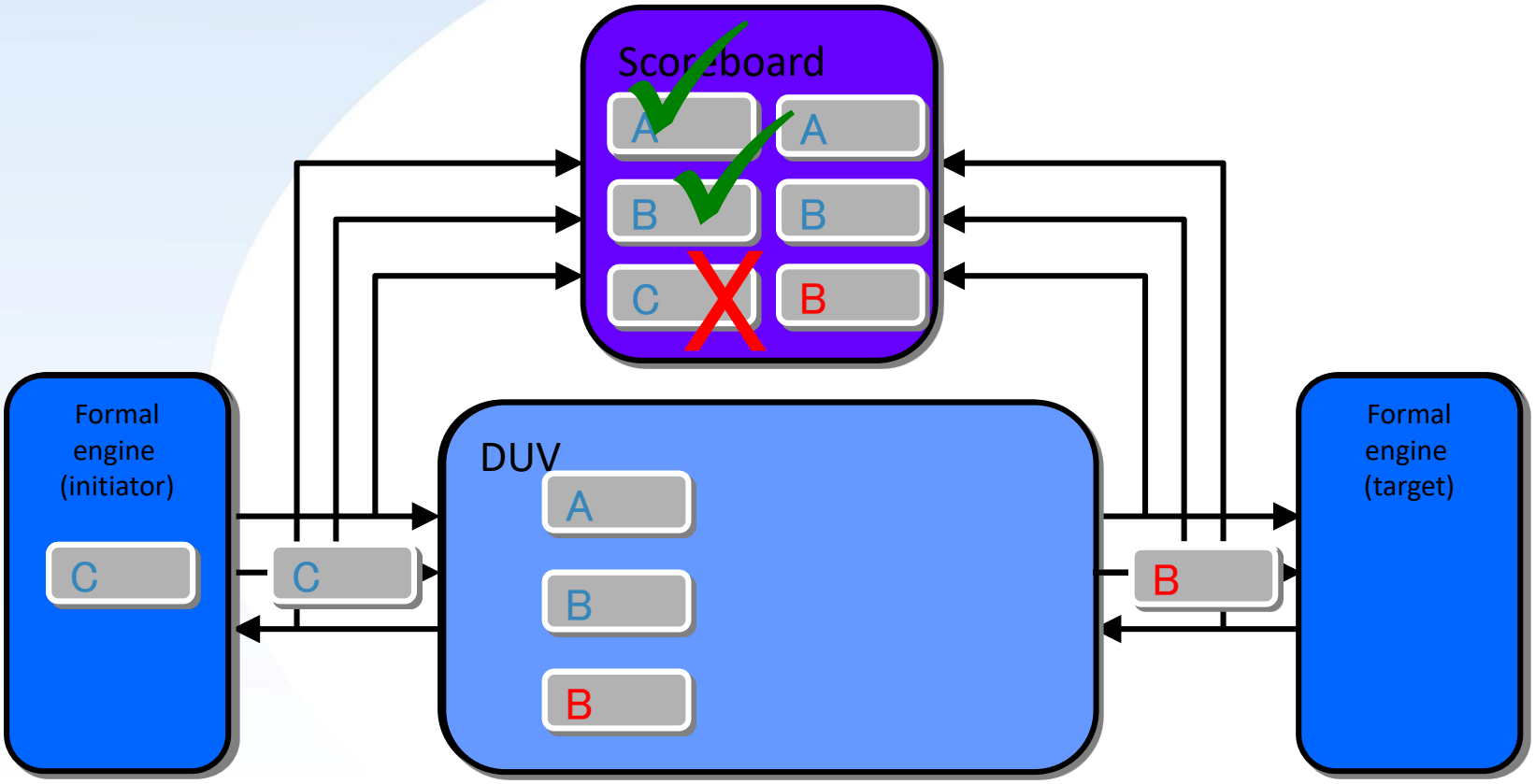
Environment Setting

- module top();
- wire declaration for wire connection;
- duv initialization();
- // you could leave some without connection to allow Formal Tool generate possible input in your design.
- abvip initialization();
- defparam abvip_parameter setting

endmodule

Scoreboard

- **Scoreboard behaves like a monitor**
  - Observe input data and output data of DUV



Jasper Scoreboard

- **Formal optimized to reduce state-space complexity**
  - Reduce barrier for adoption
- **What to check**
  - Data packet dropped
  - Duplicated Data Packets
  - Order of Data Packets
  - Corrupted Data Packets

Jasper_scoreboard_3

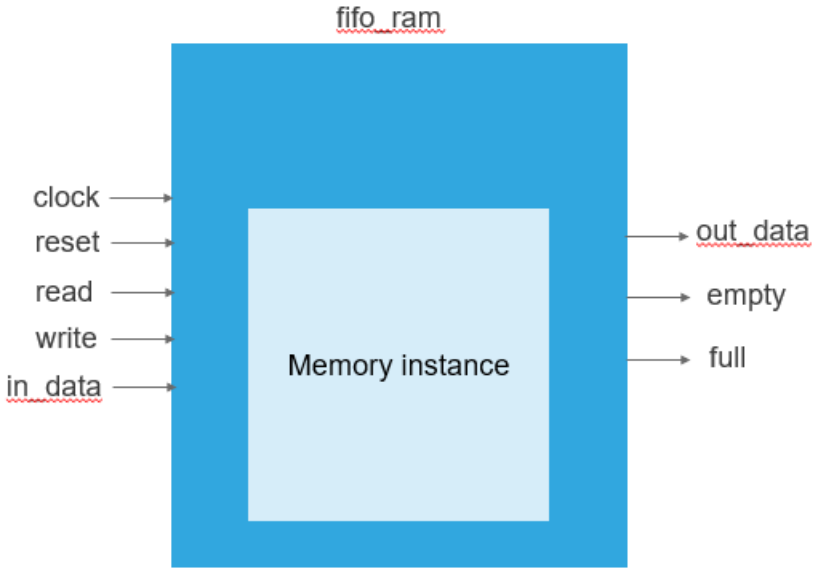
- **Mandatory parameters**

CHUNK_WIDTH	Smallest chunk of data that DUV can handle. (default value: 1; legal value ≥ 1)
IN_CHUNKS	num of chunks at ingress. (default value: 1; legal value ≥ 1)
OUT_CHUNKS	num of chunks at egress. (default value: 1; legal value ≥ 1)
SINGLE_CLOCK	1: in_data and out_data are driven on same clock 0: otherwise
ORDERING	Select in-order or out-of-order verification (default value: `JS3_IN_ORDER; legal value: `JS3_IN_ORDER or `JS3_OUT_OF_ORDER)
MAX_PENDING	Max. num of pending chunks that DUV can store.

Example: fifo_ram

```
83 jasper_scoreboard_3 #(.CHUNK_WIDTH(16)
84   , .IN_CHUNKS(1)
85   , .OUT_CHUNKS(1)
86   , .SINGLE_CLOCK(1)
87   , .ORDERING(`JS3_IN_ORDER)
88   , .MAX_PENDING(32)
89
90 ) dp
91   ( .rstN(~reset)
92
93   , .clk(clock)
94   , .incoming_clk  (      )
95   , .outgoing_clk (      )
96
97   , .incoming_vld  (write)
98   , .incoming_data (in_data)
99
100  , .outgoing_vld (read)
101  , .outgoing_data (out_data)
102
103
104
105  );
```

- Depth = 32, Data bits = 16



TCL command when using jasper_scoreboard_3

- To use the macros `JS3_* to specify the ordering type of verification, need to run “jasper_scoreboard_3 –init” before analyzing the verification module that uses the macros.

```
 1 clear -all
 2
 3 jasper_scoreboard_3 -init
 4
 5 analyze -verilog fifo.v
 6 analyze -sv      fifo_check.sv
 7
 8 elaborate -top fifo_8x16
 9
10 clock clk
11 reset ~nreset
12
13 #prove -all -bg
```

Revision History

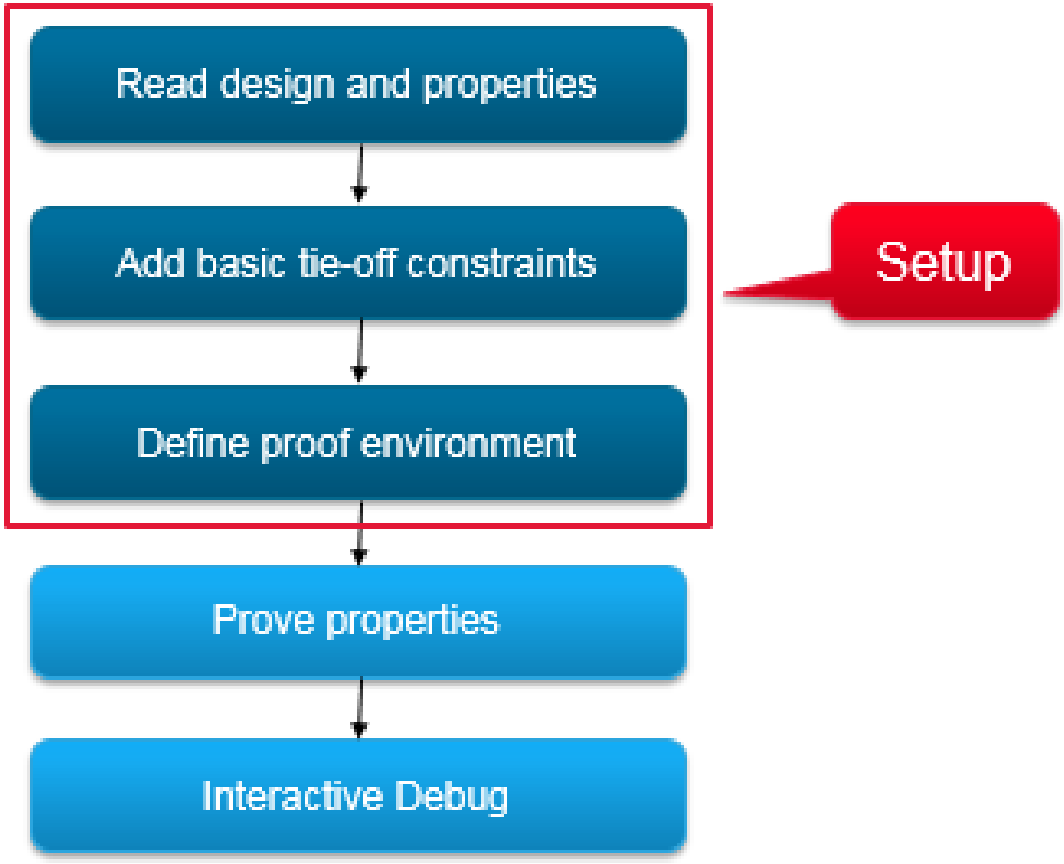
2018 Spring/Fall	:	Jiang Wei (main intro + scoreboard)
2019 Spring	:	Li-Wei Liu (add abvip)
2019 Fall	:	Li-Wei Liu
2020 Spring	:	Kai-Jyun Hung
2020 Fall	:	Kai-Jyun Hung (modify glue logic)
2021 Spring	:	Lin-Hung Lai (modify all pages & add coverage analysis)
2021 Fall	:	Wen-Yue Lin
2022 Spring	:	Wen-Yue Lin
2022 Fall	:	Po-Kang Chang
2023 Spring	:	Chan-Pin Hung

Outline

- Section 1. Introduction to Formal Analysis
- Section 2. Introduction to SVA
- Section 3. JasperGold – Setup
- Section 4. JasperGold – Formal Coverage Analysis
- Section 5. JasperGold – Assertion Based Verification IP & Scoreboard

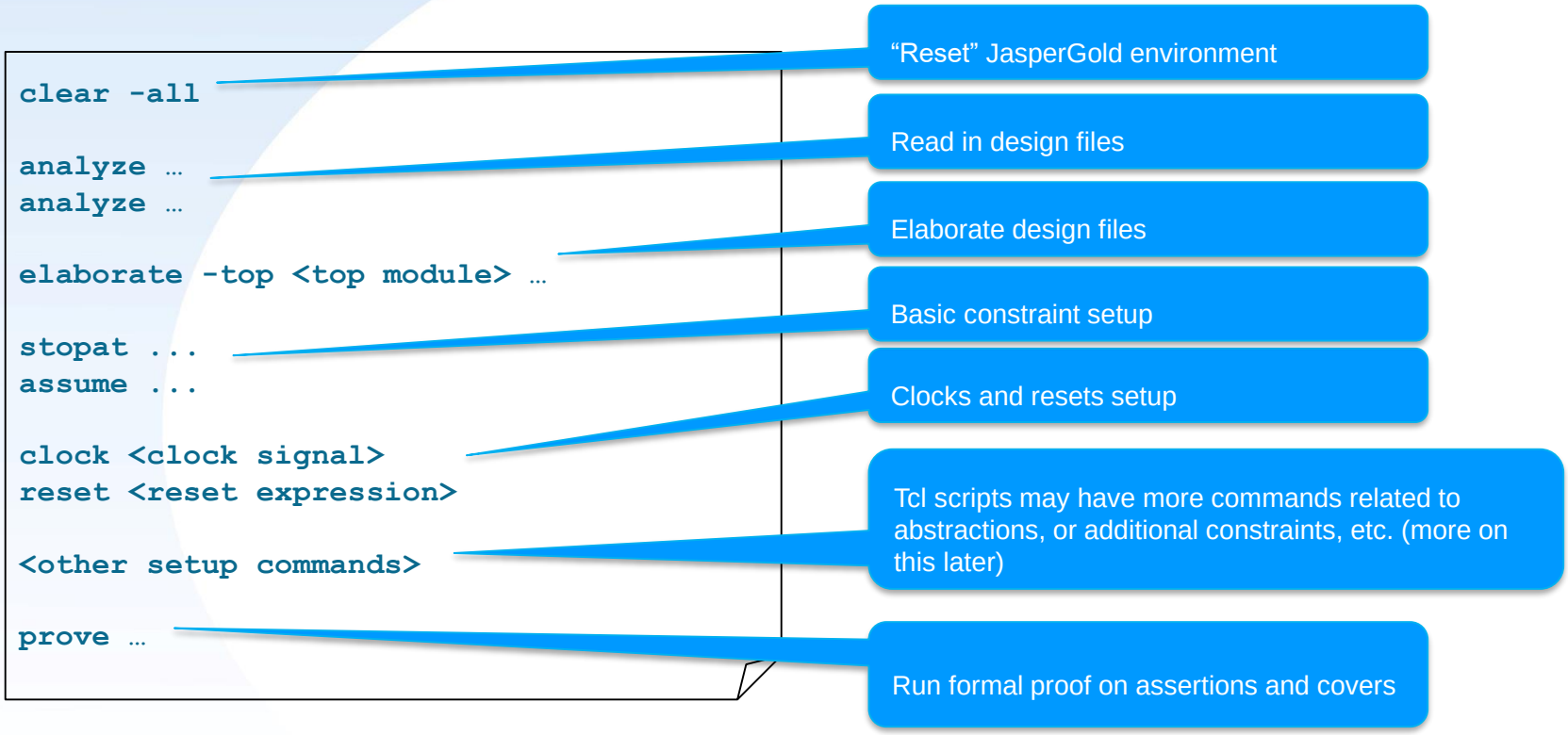
- **Appendix A. JasperGold Setup**
- Appendix B. Bound Analysis

JasperGold Flow Overview



JasperGold Flow Overview

- **A typical Tcl script for JasperGold looks like this:**



JasperGold Flow Overview

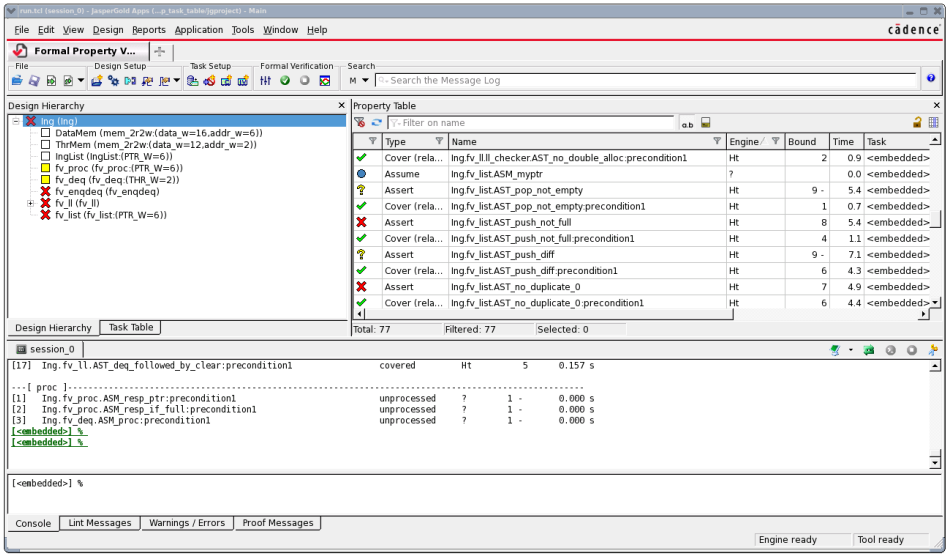
- **Start JasperGold Apps in either batch or GUI mode**
- **Call a Tcl script when you start or later through the command line or GUI**
- **Commands:**
  - Start in GUI mode

    ```
    % jg
    ```
  - Start with a Tcl script

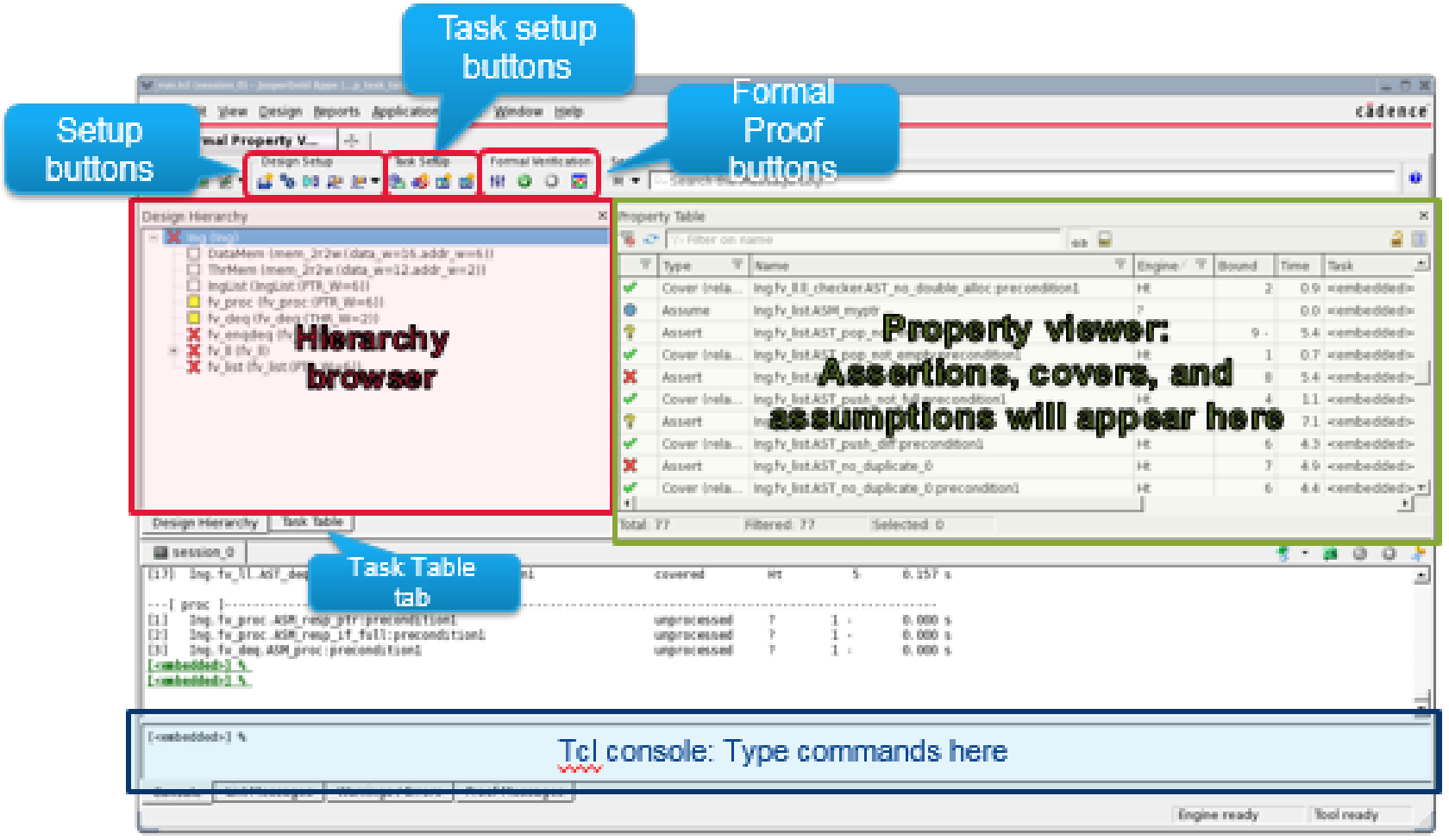
    ```
    % jg example.tcl
    ```
  - Start in text/interactive mode

    ```
    % jg -no_gui
    ```
  - Start with a Tcl script in batch mode
    - Exits tool at end of the Tcl file

    ```
    % jg -batch example.tcl
    ```



GUI Overview



Compilation

- **Read in design and property files and check for syntactical errors**
  - Read in files in Verilog, SystemVerilog, VHDL, PSL, etc.
  - Specify individual design files or specify file lists (-f)

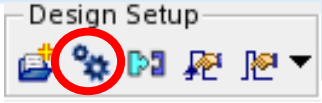
- **Command:**

```
% analyze ...
```

- **Examples:**

```
% analyze -verilog my_design.v          // Compile Verilog file
% analyze -sv my_monitor.sv             // Compile SystemVerilog
% analyze -verilog -psl my_vunit.v.psl  // Compile PSL vunit
```

Compilation



- **Elaborate to synthesize and build the design hierarchy**
- **Command:**

```
% elaborate
```

- **Examples:**

```
% elaborate   // Automatically infer top level module
% elaborate -top top_mod   // Specify top level module top_mod
% elaborate -vhdl -top top(rtl)// Specify top level entity top(rtl)
```

- **Command-line language**
  - The language of the top-level module is the default mode for entering expressions on the command line
  - Use `-mode` option to override the expression mode on the command line.

```
% elaborate -vhdl -top ALU
```

Basic Tie-off Constraints

- **Common to have to tie-off certain pins/registers as part of the setup**
  - Scan mode, test mode, etc.
- **Use `assume` command to apply constraints**
  - Use `-env` switch to apply constraint globally, to all tasks and reset analysis
  - Use `stopat` command to cut the driving logic of signal you need to constrain (e.g. config register)

- **Example:**
  - Global assumption (across all tasks and during reset analysis) – apply constant value

```
% assume -env {scan_mode == 1'b0}
```

  - Or task based one – apply constant value

```
% assume {func_mode == 2'b01}
```

Cutpoints

- **Background**
  - Sometimes you need to remove complex logic from proof
  - JG provides a way to "cut" the net and make it undriven
  - Cutpoint is called "**stopat**" in Jasper terminology

- **The `stopat` command can cut any net or bus in the design**
  - Add environmental stopat (will apply to all tasks and reset analysis)

```
% stopat -env <signal> <signal> …
```

  - Add task-specific stopat

```
% stopat -task <signal> <signal> …
```

  - Or to apply to the current task

```
% stopat <signal> <signal> …
```

- **Example: Place stopat on FIFO output bus**

```
% stopat top.fifo_dataout
```

Configuration Registers

- **In formal, configuration registers are programmed at the flip-flop/wire level**
  - Similar to a simulation force

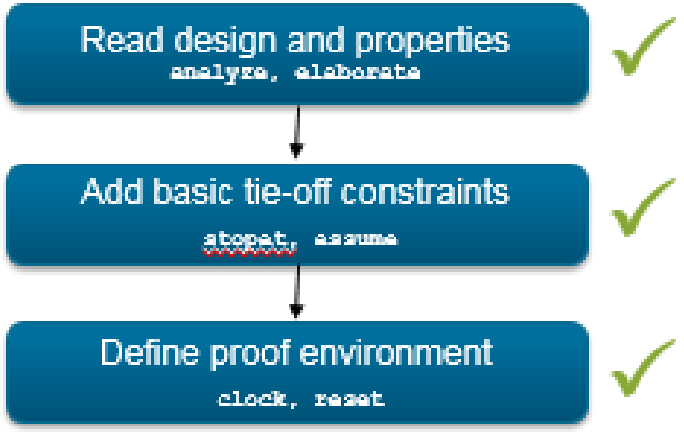
```
% stopat -env cfg.regs.prog_speed
% assume -env {cfg.regs.prog_speed == 8'h5a}
```

- **Also possible to allow a set of values for the register**

```
% stopat rf.if0.mode
% assume -constant rf.if0.mode
% assume {rf.if0.mode == 2'b01 || rf.if0.mode == 2'b10}
```

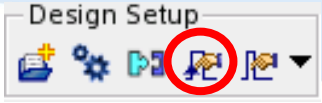
- **Do not try to program registers via interface writes!**
  - Suitable for simulation only
  - Major performance drag for formal

JasperGold Flow Overview




```
clear -all

analyze …
analyze …

elaborate -top <top module> …

stopat ...
assume ...

clock <clock signal>
reset <reset expression>

<other setup commands>

prove …
```

Clock Setup



- **Use the command `clock` to declare a signal as clock**
  - Declaring a signal as the clock is equivalent to connecting a clock generator to it
  - Use `-both_edges` switch if the design is sensitive to both edges of the clock

- **Example:**
  - Declare signal clk to be the fastest clock

```
% clock clk  # Positive edge active clock
% clock ~clk # Negative edge active clock
```

  - Design samples on both edges of clock

```
% clock clk -both_edges # Both edges active clock
```

- **Clock declaration is required**
  - If you do not wish to constrain the design clock signal or you do not have a system clock (combinational logic) execute the following command:

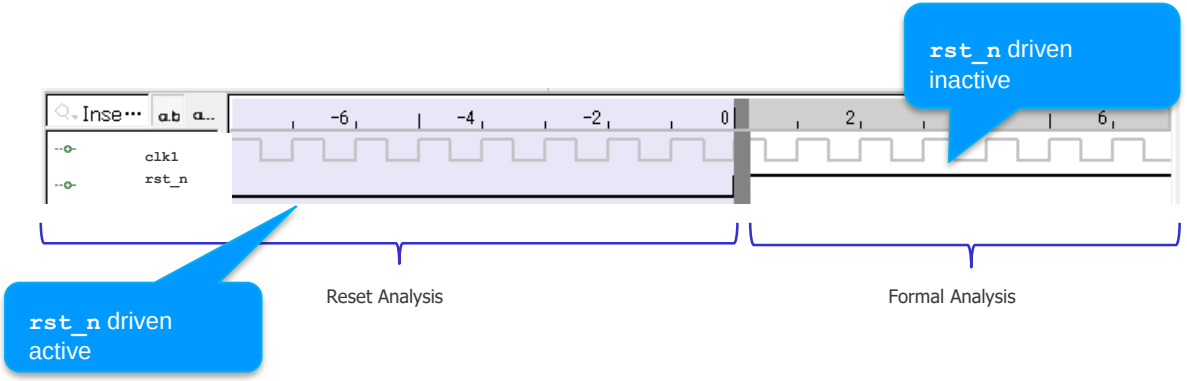
```
% clock -none # No clock on formal environment
```

Reset Setup

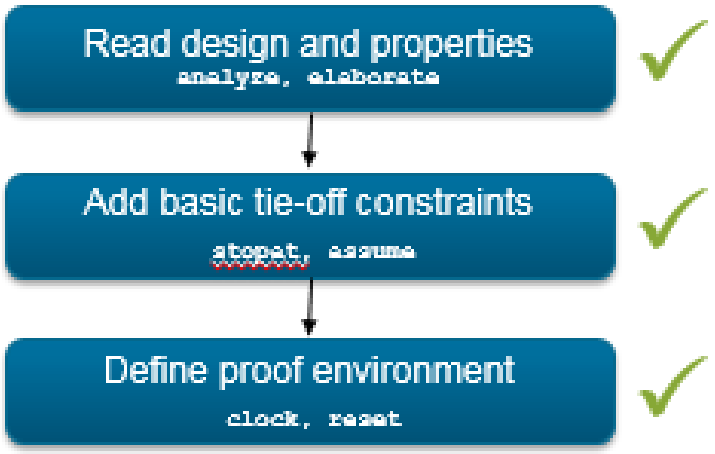
- **Use the command `reset` to configure design initialization**
  - Many different ways to initialize design: using reset signal, user-defined sequence, external waveform, etc.
  - Most common method is to specify reset signals that are used to reset design
    - Tool will automatically simulate the design with reset asserted
    - Reset signal will be constrained not to be asserted during formal analysis

- **Example:**
  - Specify signal `rst_n` as active low reset

```
% reset ~rst_n
```



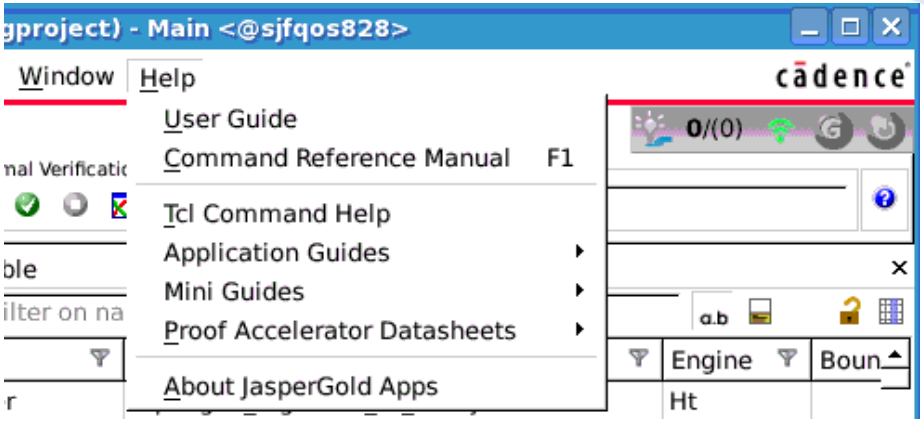
JasperGold Flow Overview




```
clear -all

analyze …
analyze …

elaborate -top <top module> …

stopat ...
assume ...

clock <clock signal>
reset <reset expression>

<other setup commands>

prove …
```

Getting Help in JasperGold

- ✓**Help Menu and Quick Help wizard button**



- ✓**Help Command**
  - Display help on a specific command
    ```
    % help <command_name>
    ```
  - Display help on specific info, warning, and error messages
    ```
    % help -message
    <msg_number>
    ```
- ✓**Examples:**
    ```
    % help
    % help -message
    % help view
    % help -message EAS007
    ```

Outline

- Section 1. Introduction to Formal Analysis
- Section 2. Introduction to SVA
- Section 3. JasperGold – Introduction & Setup
- Section 4. JasperGold – Formal Coverage Analysis

- Appendix A. JasperGold Setup
- **Appendix B. Bound Analysis**

Step 1 – DUT dead code (Stimuli coverage)

Objective

- Learn about DUT and env's dead code

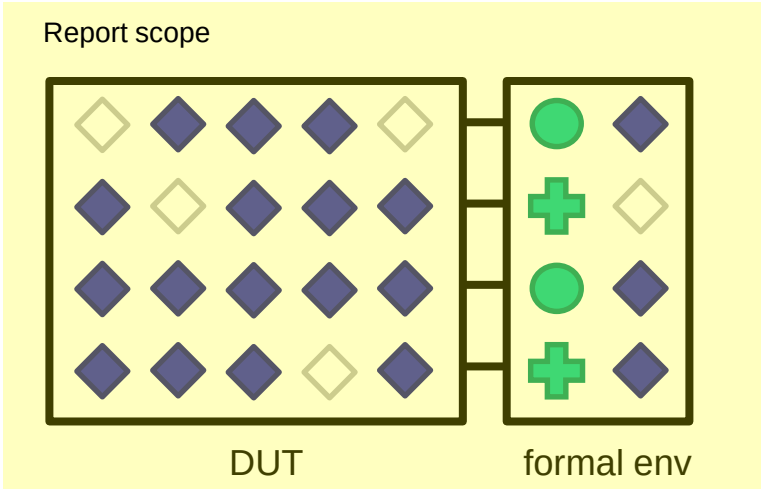
Report Scope and Model

- All – Run unreachable report with deadcode switch

Results

- If dead code is unintentional: BUG
- If dead code is intentional: waive from future analysis on DUT

Formal Hierarchy



Step 2 – Formal environment overconstraints

Objective

- Review unreachables caused by assumptions

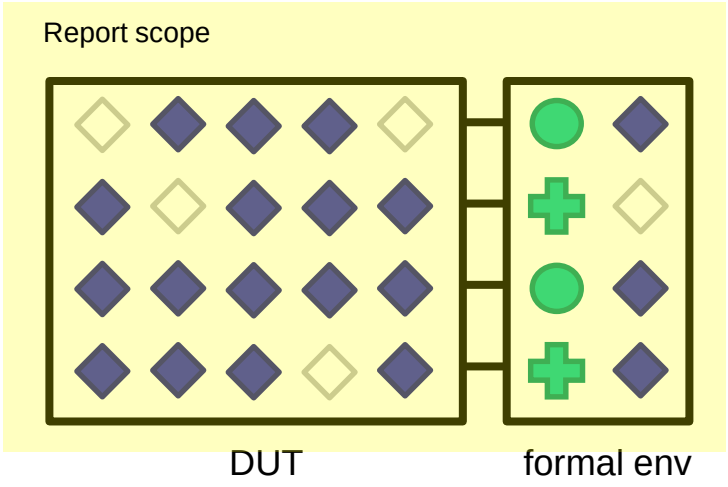
Report Scope and Model

- All – Run unreachable report with overconstrained switch

Results

- Unintentional overconstraint: Environment has to be refined
- Intentional overconstraint: Should be exposed with functional coverage, so that it won't be forgotten later

Formal Hierarchy



Step 3 – COI

Objective

- Assess checker completeness by finding logic of the DUT that is not structurally connected to any assertion

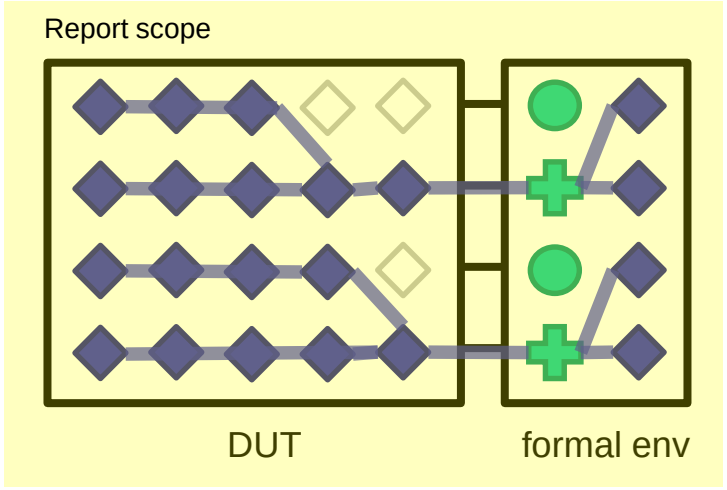
Report Scope and Model

- All – statement, expression, functional

Results

- More comprehensive assertion set (might require designer's assistance to understand a particular logic's functionality)
- Find irrelevant code (reachable but not propagated to outputs)

Formal Hierarchy



Step 4 – Proof Coverage

Objective

- Find logic of the DUT that doesn't influence the result of any assertion

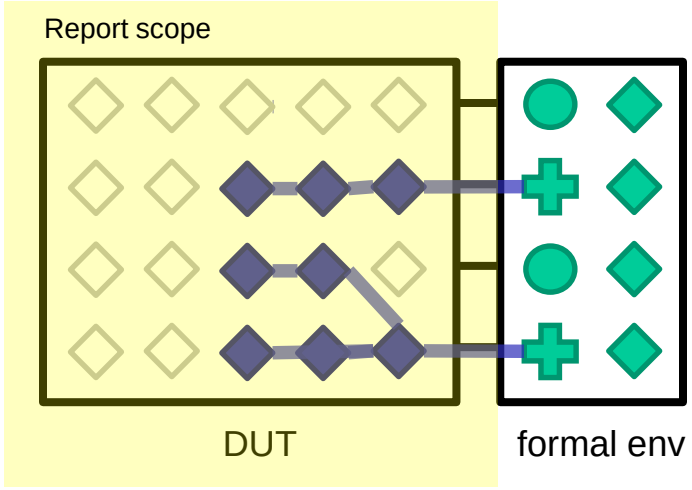
Report Scope and Model

- DUT – statement, expression

Results

- More comprehensive assertion set (might require designer's assistance to understand a particular logic's functionality)
- More prove runs to increase proof coverage by reaching deeper states

Formal Hierarchy



Step 5 – Bound analysis of undetermined assertions

Objective

- Discover if undetermined assertions were sufficiently explored

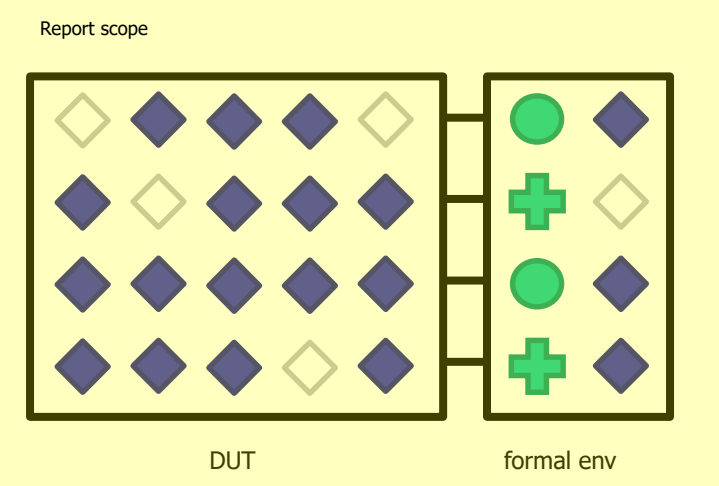
Report Scope and Model

- All – branch, expression, functional

Results

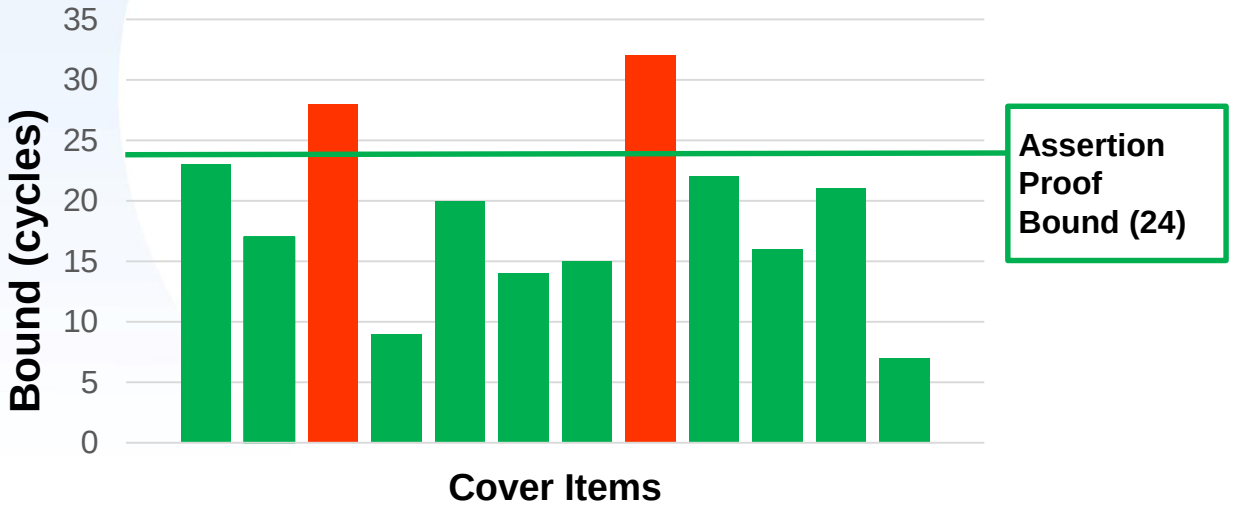
- If assert bound is insufficient:
  - Prove needs to
- If needed, apply techniques to improve bound coverage
  - Proof bound expansion
  - De

Formal Hierarchy



Bound Coverage

- **Provides analysis of the cycle bound achieved by a bounded proof**
- **Example – Bounded proof of 24 cycles**
  - Is 24 a sufficient bound?
  - What related functionality has NOT been exercised within 24 cycles?
- **Reports cover items in COI of the assertion not reachable within 24 cycles**
  - Items outside the COI are not relevant to the assertion
- **Covers beyond 24 cycles have not been exercised by the bounded proof**



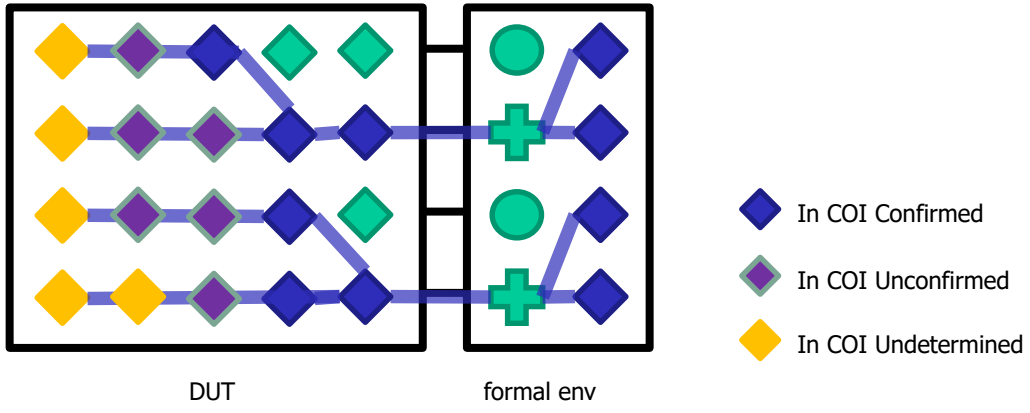
Bound Coverage

Why: For undetermined asserts, show if current assert bound is sufficient
How: Compares bound of code and functional Cover Items in COI of assert with assert bound

Results:

- **Items in COI:** Total number of connected Cover Items
- **Confirmed Covers (not part of default report):** Items in assert's COI with equal or lower bound than assert bound
- **Unconfirmed Covers:** Items in assert's COI that have a higher bound than assert bound
- **Undetermined Covers:** Items in assert's COI that don't have a bound to compare

Action: Use minimal traces, run longer proofs, use bound expansion and reduction techniques



Bound Coverage

Interpretation of Unconfirmed Items

- **Code: The current assert bound is lower than the minimum needed to exercise the relevant portion of design**
  - e.g.: An assert that checks a FIFO-full signal needs to reach at least the minimum bound in which the signal activates
- **Functional: The current assert bound is lower than the minimum needed to reproduce the described functionality**
  - e.g.: An assert that checks a 10-cycle packet needs to reach at least the minimum bound required for that packet to be processed

Interpretation of Undetermined Items

- **Undetermined items are items in COI with an "Undetermined" status in the reachability analysis, thus they don't have an associated bound information**

Coverage Methodology - Summary

Step	Model	Analysis	Application
Dead code	Branch, Expression	Stimuli	Expose DUT's dead code and waive from later steps
Env Overconstraints	Branch, Expression, Functional	Stimuli	Expose unreachable code caused by env configuration
COI	Statement, Expression, Functional	COI	Expose DUT's portions that are not being checked (using structural analysis)
Proof	Statement, Expression	Proof	Expose DUT's portions that are not being checked (using functional analysis returned by the engines)
Bound	Branch, Expression, Functional	Bound	Determine if depth of undetermined assertions in sufficient