



Jasper Formal Verification Overview

NCKU VSD Lecture

Nelson Zheng,
Oct. 2022

cādence®

Outline

- Introduction to Jasper[®] Formal Analysis
- Formal Friendly SVA Coding
- Jasper Superlint



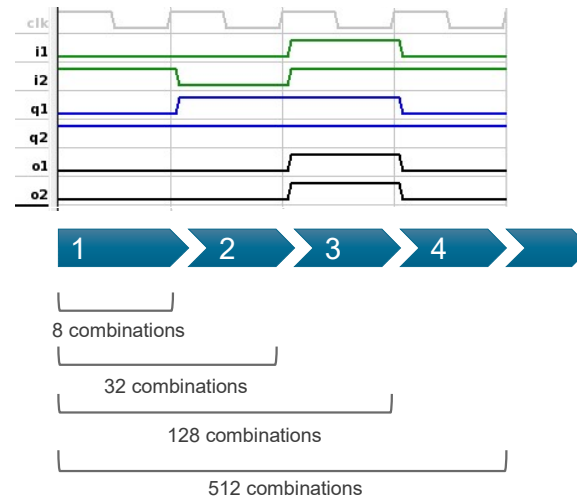
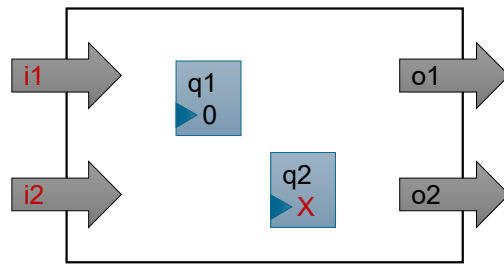
Introduction to Jasper[®] Formal Analysis

What is formal verification?

- Checks exhaustively/completely if a model meets a given spec
 - Model → synthesizable RTL
 - Spec → properties (assertions and covers)
- Key differences between **simulation** and **formal**
 - Simulation → User creates given stimulus set using a complicated TB wrapper
 - Formal → User specifies illegal stimulus using a set of properties
- What are the main benefits of formal verification?
 - Shortening the time to market by starting verification with designers months earlier
 - Improving quality by exposing corner case bugs
- Trick is to know where/when to apply Jasper

Formal Analysis

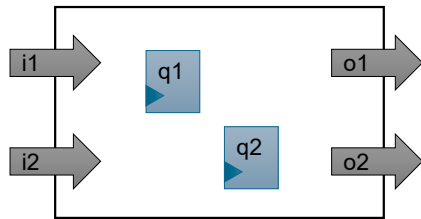
- Formal tests all possible stimulus, one cycle at a time
 - All value combinations on uninitialized registers at the first cycle
 - All value combinations on inputs and undriven wires at every cycle



- 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 cover properties are hit
 - Report when assert properties are violated



✓ cover (q2 ##2 o1)

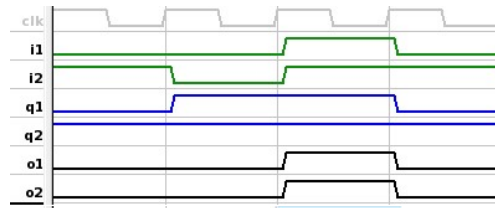
Waveform saved
in database

✗ assert (o1 && o2 ==> q1)

Waveform saved
in database

Found scenario
that hits cover!

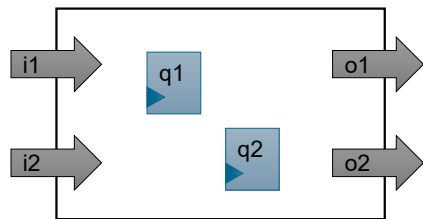
Found scenario
that violates assertion!



- 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

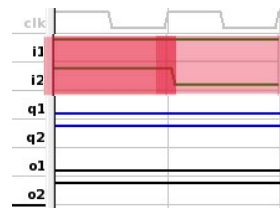


cover (q2 ##2 o1)

assert (o1 && o2 ==> q1)

assume (i1 ==> !i2)

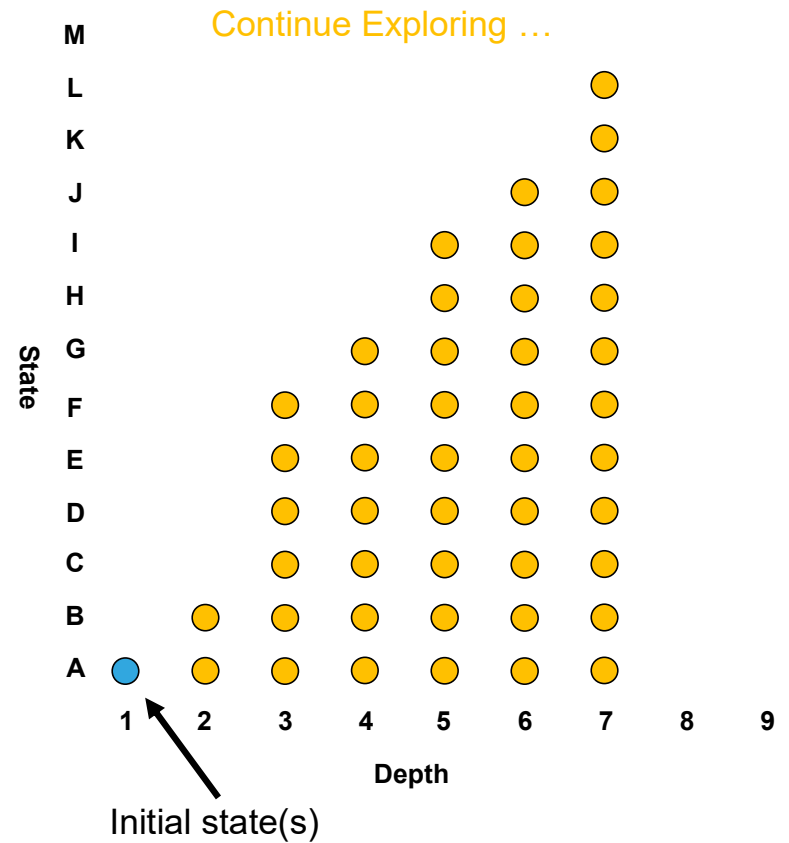
Found scenario
that violates assumption!



Discard this scenario
and continue

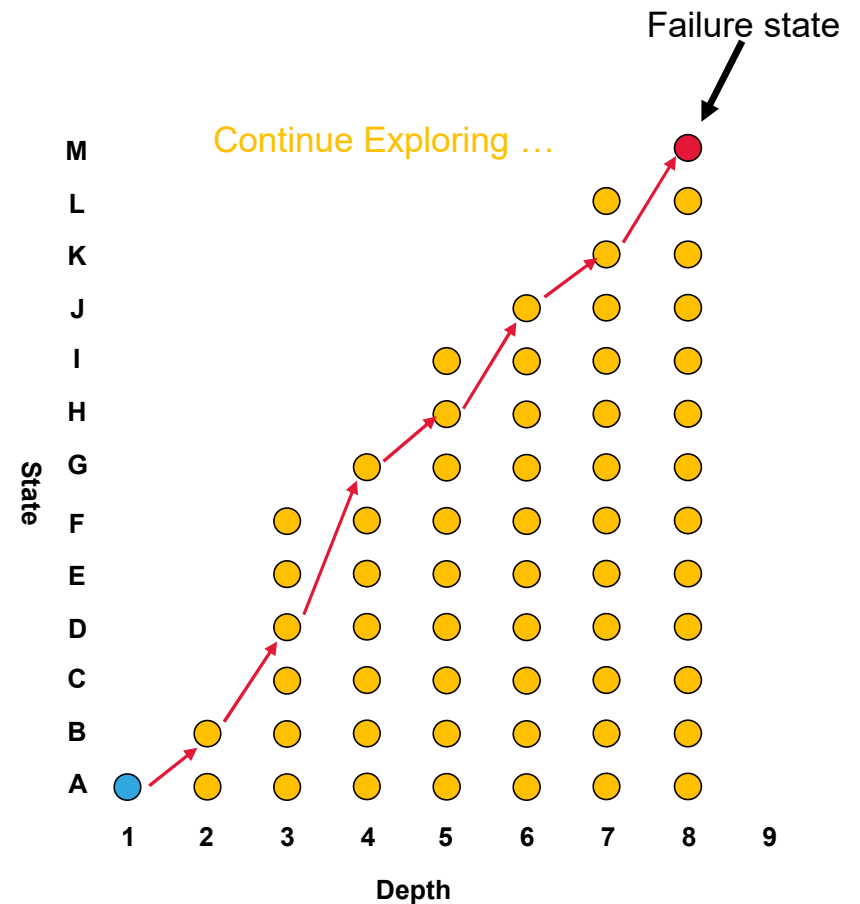
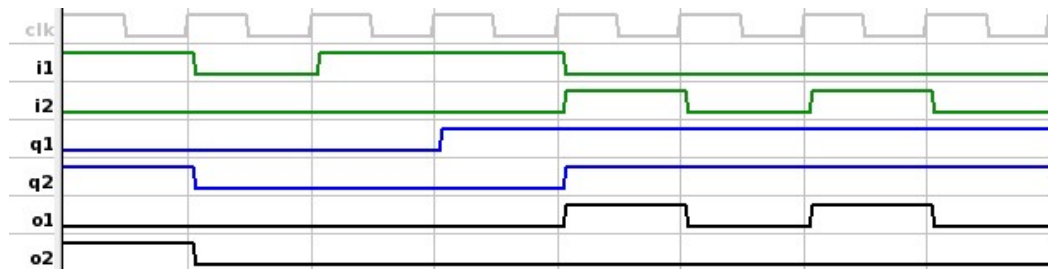
Formal State Space Exploration

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

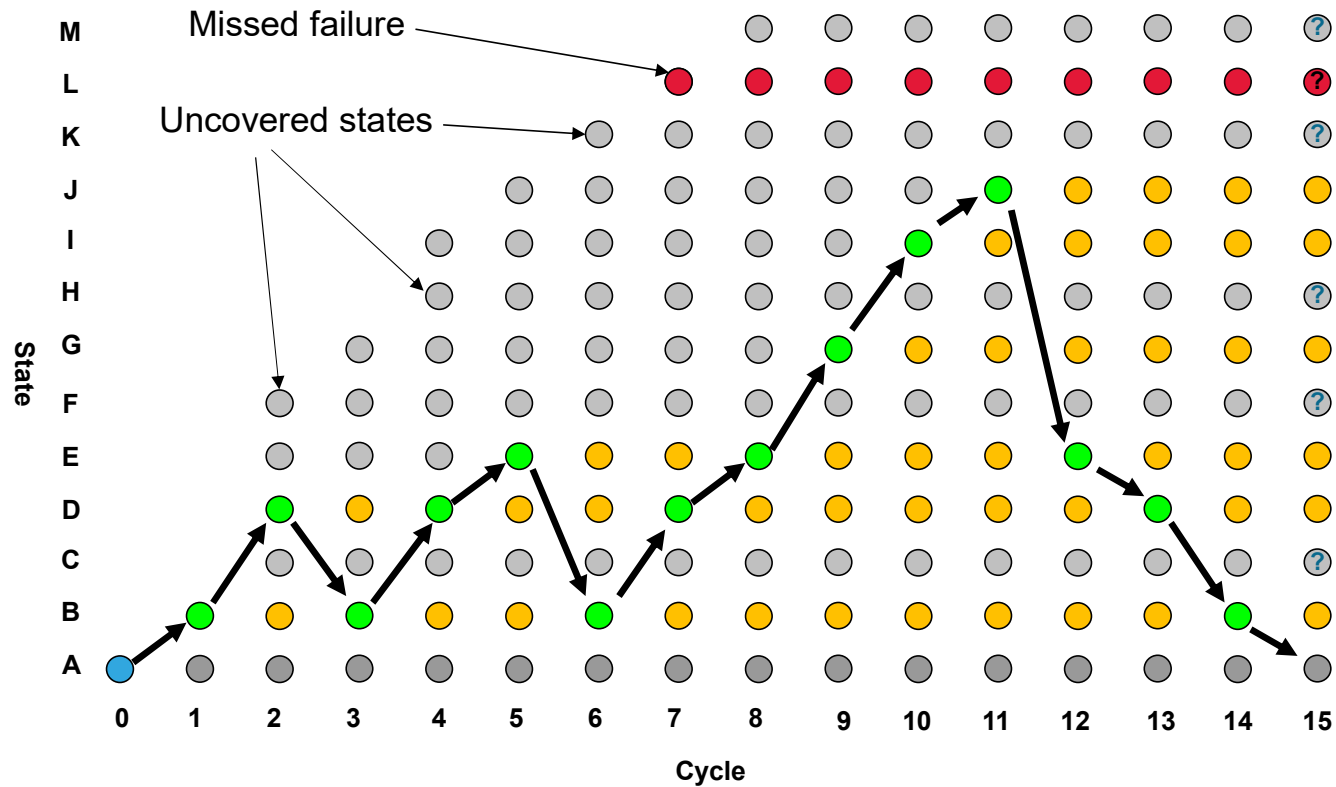


Formal State Space Exploration

- 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



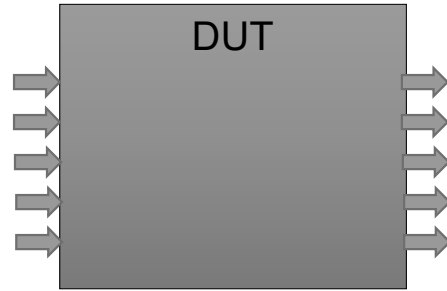
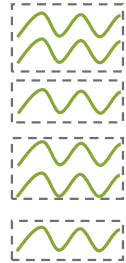
Compared with Simulation



- Misses many reachable states (C, F, H, K, L, and M)
- Covers many states multiple times (A, B, D, and E)

Constrained Random Simulation in a Nutshell

Verif. Engineer
creates generators to
drive stimulus and
sensitize the design



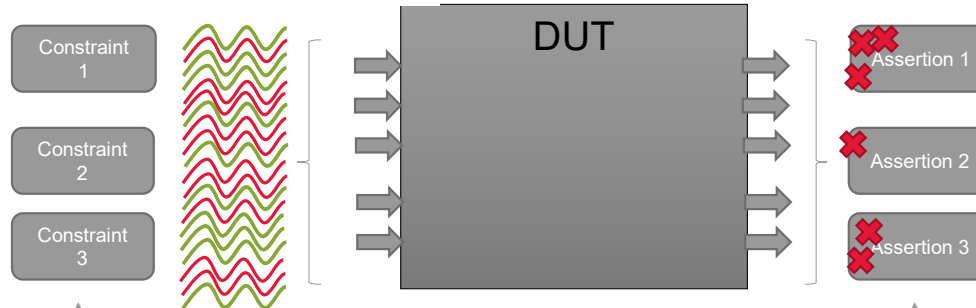
Verif. Engineer
creates checkers to
observe design and
flag for errors



Formal Verification in a Nutshell

Verif. Engineer creates constraints to prevent formal from driving illegal stimulus

Verif. Engineer creates assertions to observe design and flag for errors



Key difference compared to sim:

Sim:
User creates test scenarios by writing testbench



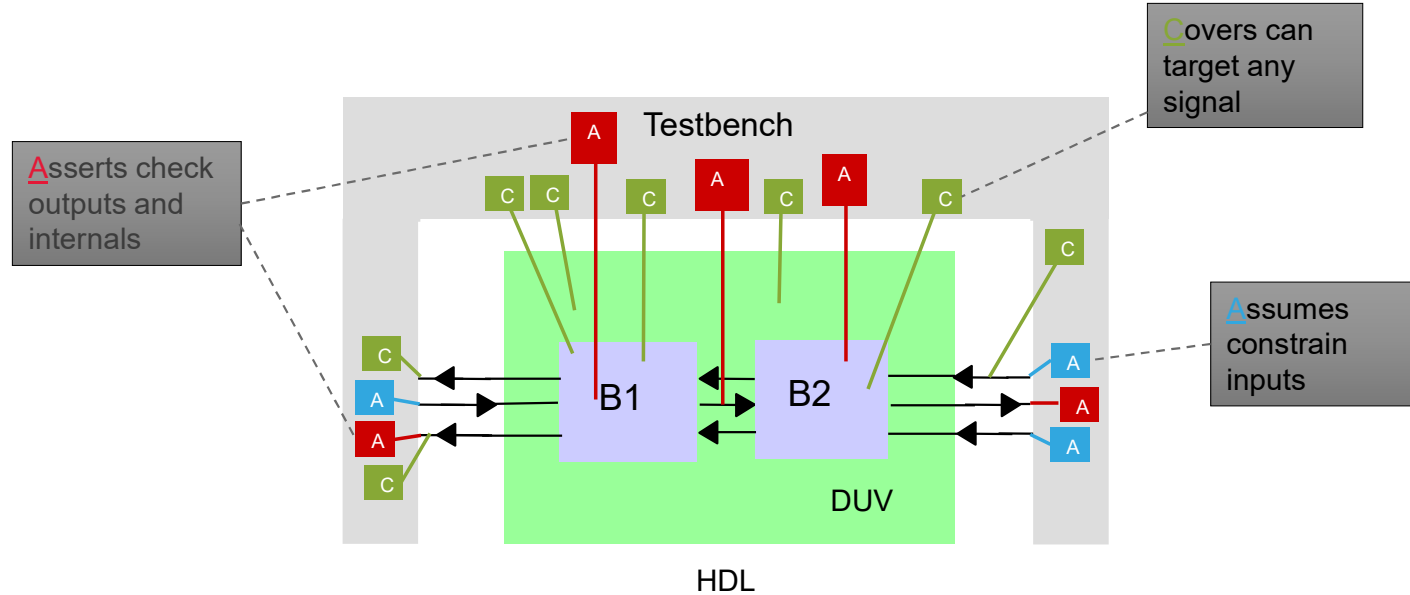
Formal:
Tool creates test scenarios based on properties



***Target-Driven Verification**

Formal Testbench

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



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



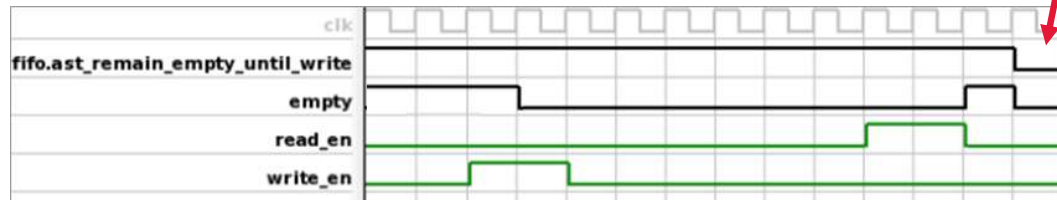
Full Proof: Impossible to violate this assertion

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







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

	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



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

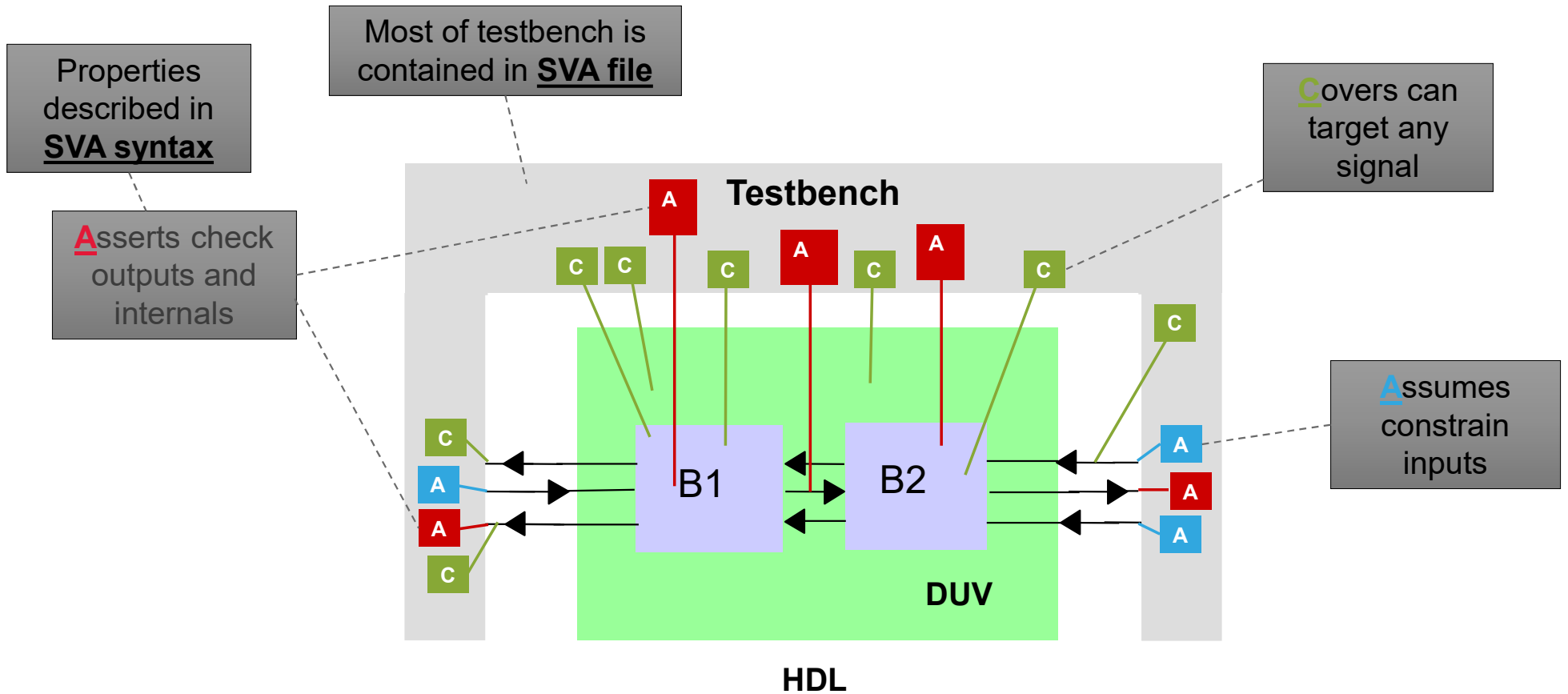


Formal Friendly SVA Coding

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 the 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



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

output_no_underflow: **assert** **property** (

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

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
```

Design Hierarchy		Property Table	
			

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), ...);
end
```


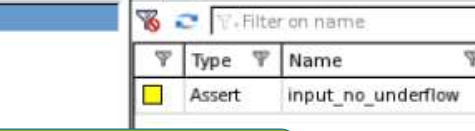
Design Hierarchy		Property Table	
			

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)}
```

Design Hierarchy		Property Table	
			

Best for quick
experiments

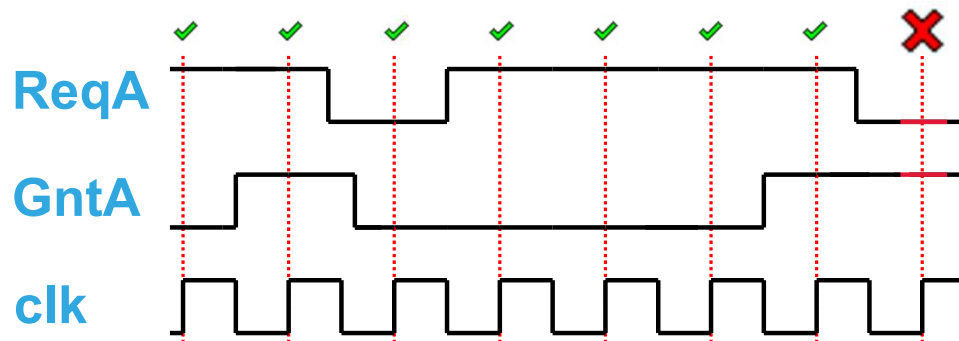
Being Successful with SVA

- 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 are 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)) ;
```

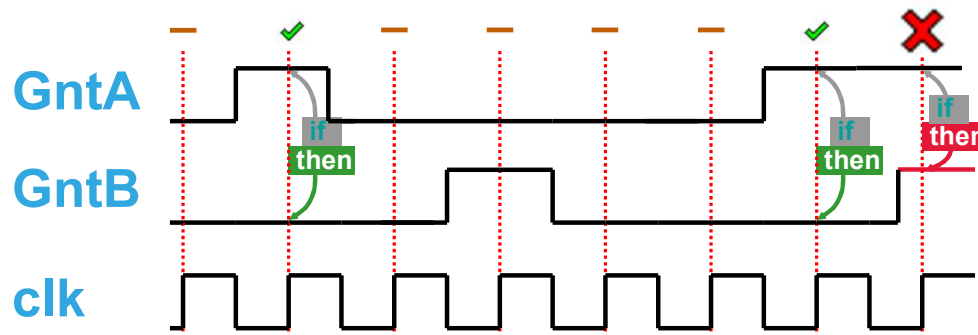


SVA Example: Same-Cycle Implications

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

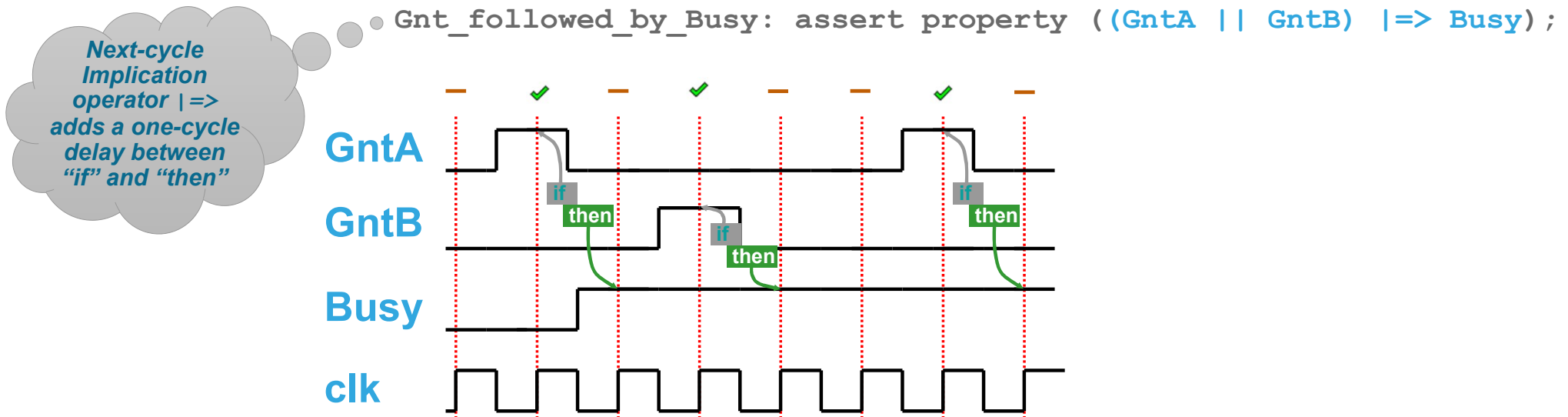
Implication
operator \rightarrow
expresses
“if...then”

• GntA_then_not_GntB: assert property (GntA \rightarrow !GntB);



SVA Example: Next-Cycle Implications

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

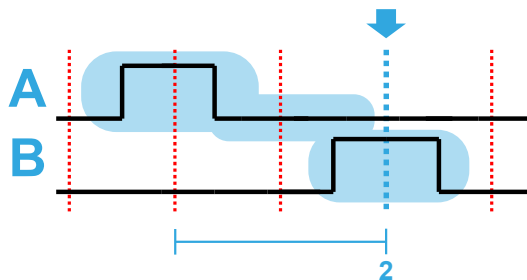


Sequences: Multi-Cycle events

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

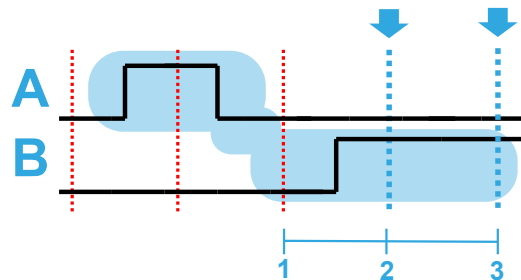
A ##2 B

"A happens then exactly 2 cycles later B happens"



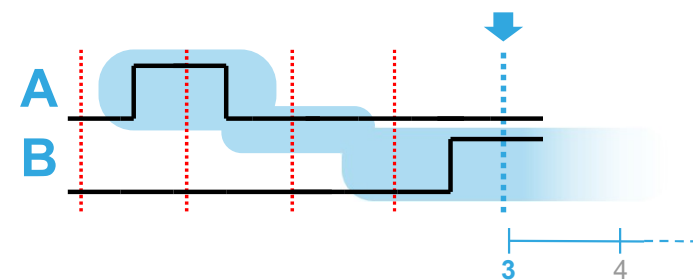
A ##[1:3] B

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



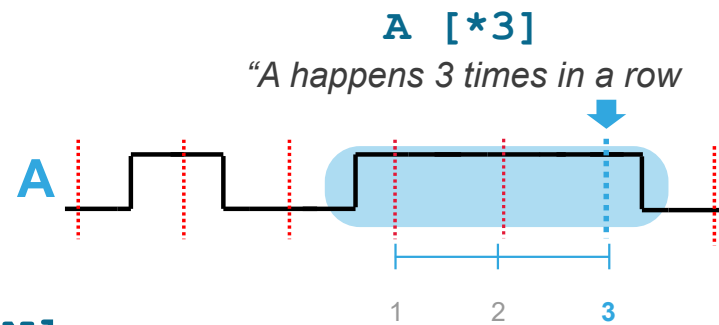
A ##[3:\$] B

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

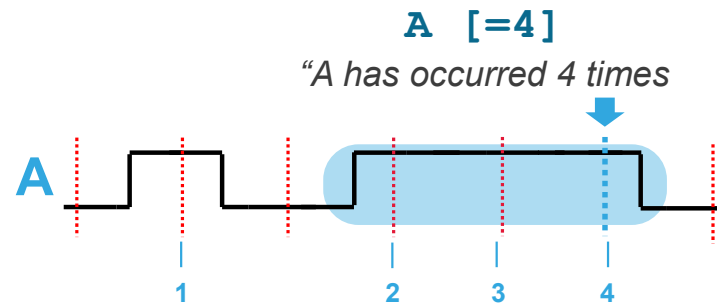


Sequences

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



- Occurrence operator **[=N]**

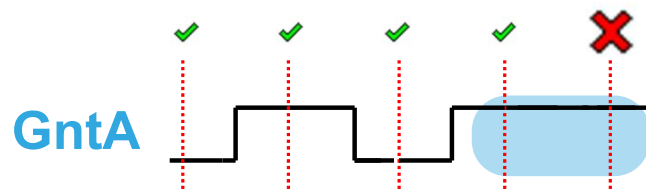


SVA Example: Sequences

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

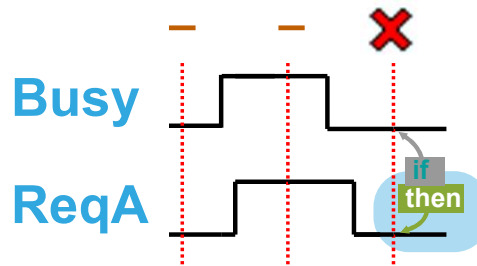
“Should never see two grants to A in successive cycles”

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



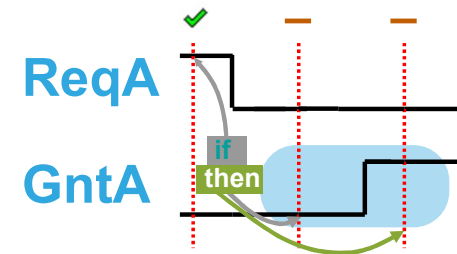
“Busy pulse should only happen if no request”

```
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
<code>\$onehot</code> <code>\$onehot0</code>	Returns true if argument has exactly one bit set (one-hot) or at most one bit set (one-hot-zero)	<i>“At least one grant should be given at a time”</i> <code>assert property (</code> <code> \$onehot ({GntA, GntB, GntC}) ;</code>
<code>\$countones</code> <code>\$countzeros</code>	Returns the number of ones/zeros in the argument	<i>“Should never see more than 4 dirty lines”</i> <code>assert property (</code> <code> \$countones (write_data[7:0]) <= 4) ;</code>

Built-In Functions

- Temporal

Function	Description	Example
<code>\$stable</code>	Returns true if argument is stable between clock ticks	<i>"Data must be stable if not ready"</i> <code>assert property (</code> <code>!Ready ==> \$stable(Data) ;</code>
<code>\$past</code>	Return previous value of argument	<i>"If active, then command must not be IDLE"</i> <code>assert property (</code> <code>active ==> \$past(cmd) != IDLE ;</code>
<code>\$rose</code>	Returns true if argument is rising, that is, was low in previous clock cycle and is high on current clock cycle	<i>"Request must be followed by Valid rising"</i> <code>assert property (</code> <code>Req ==> \$rose(Valid) ;</code>
<code>\$fell</code>	Returns true if argument is falling, that is, was high in previous clock cycle and is low on current clock cycle	<i>"If Done falls, then Ready must be high"</i> <code>assert property (</code> <code>\$fell(Done) ==> Ready ;</code>

Summary of Operators and Built-In Functions

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

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

Implication
<code>a -> b</code>
<code>a => b</code>

Sequences
<code>a ##1 b</code>
<code>a ##[2:3] b</code>
<code>a ##[4:\$] b</code>
<code>a [*5]</code>
<code>a [=4]</code>

Combinatorial Functions
<code>\$onehot(a)</code>
<code>\$onehot0(b)</code>
<code>\$countones(c)</code>
<code>\$countzeros(d)</code>

Temporal Functions
<code>\$stable(a)</code>
<code>\$past(b)</code>
<code>\$rose(c)</code>
<code>\$fell(d)</code>



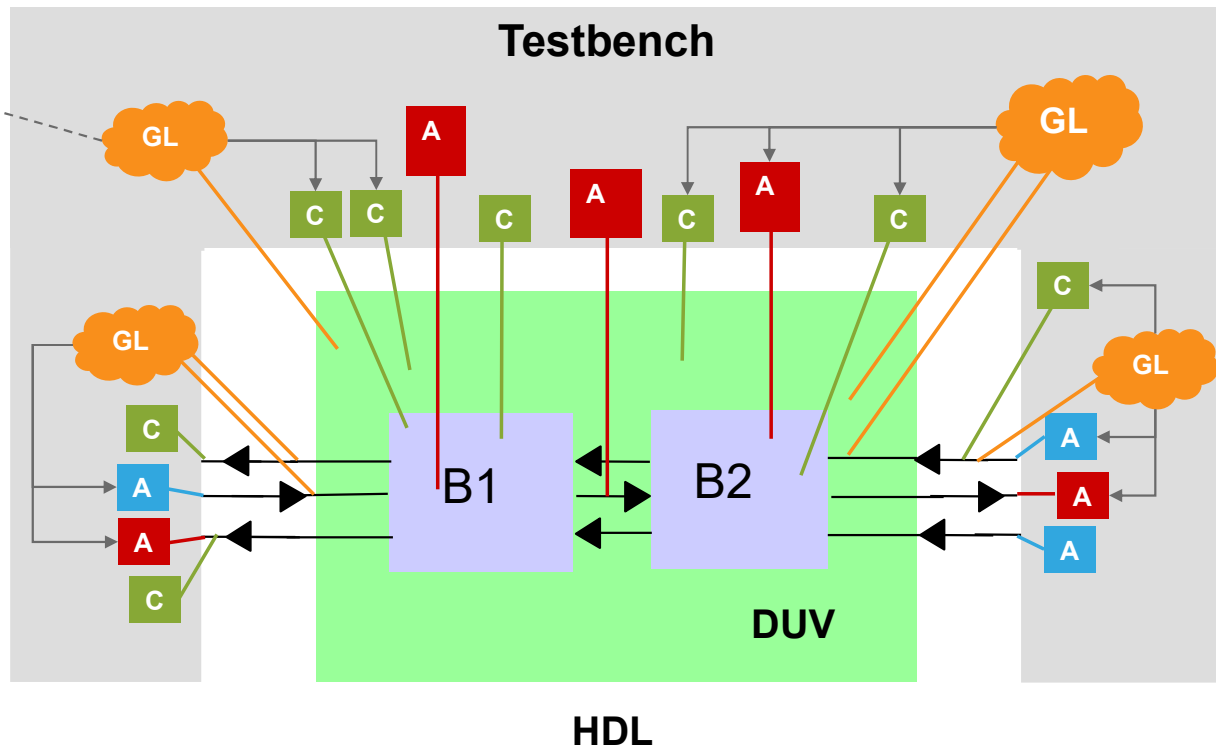
Glue Logic in Formal Environment

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 the property is all SVA or SVA+glue logic
 - Recommendation is to choose based on clarity

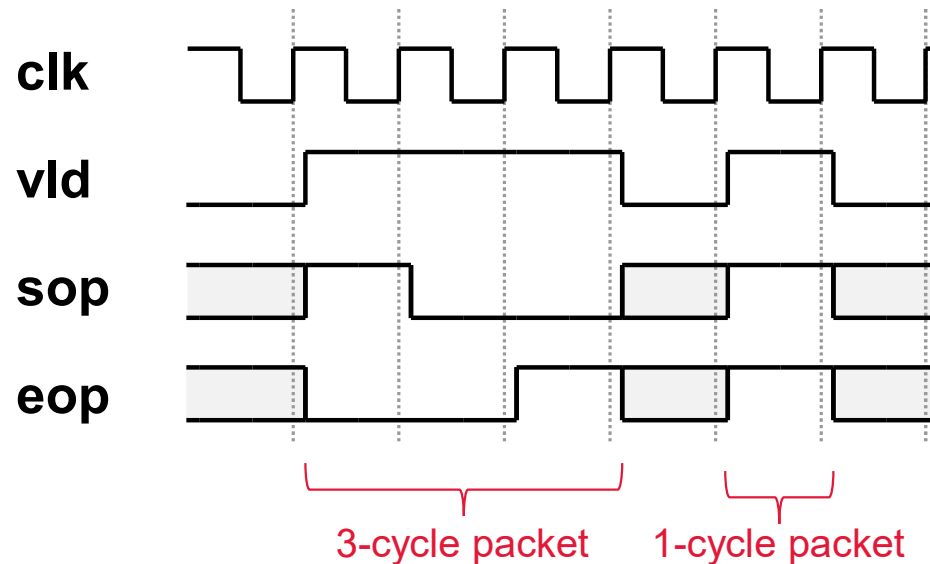
Formal Testbench

Glue Logic
monitors design
and feeds
properties

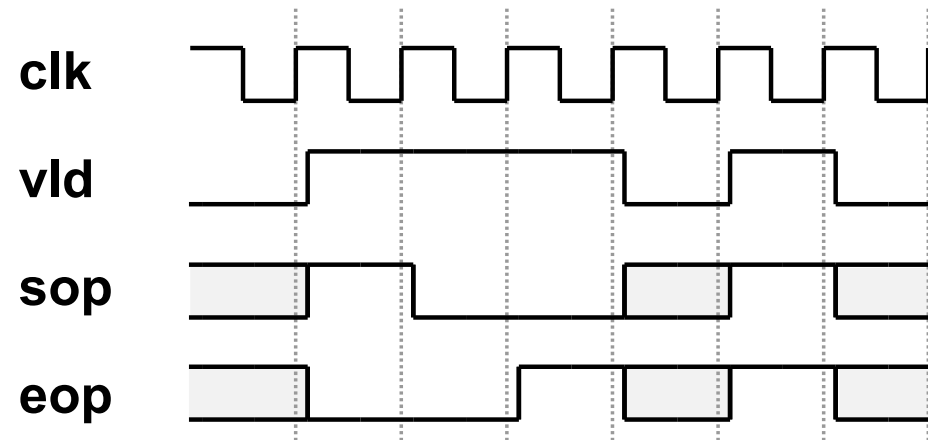


Example: SOP/EOP Interface

- 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!

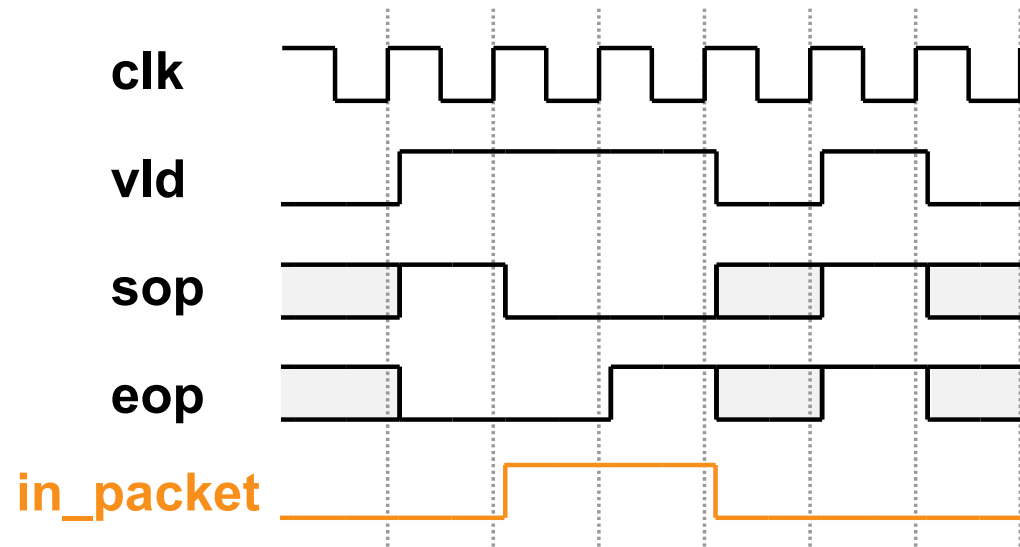
- Pure SVA version:

```
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 Superlint

Comprehensive RTL Signoff: Big Picture

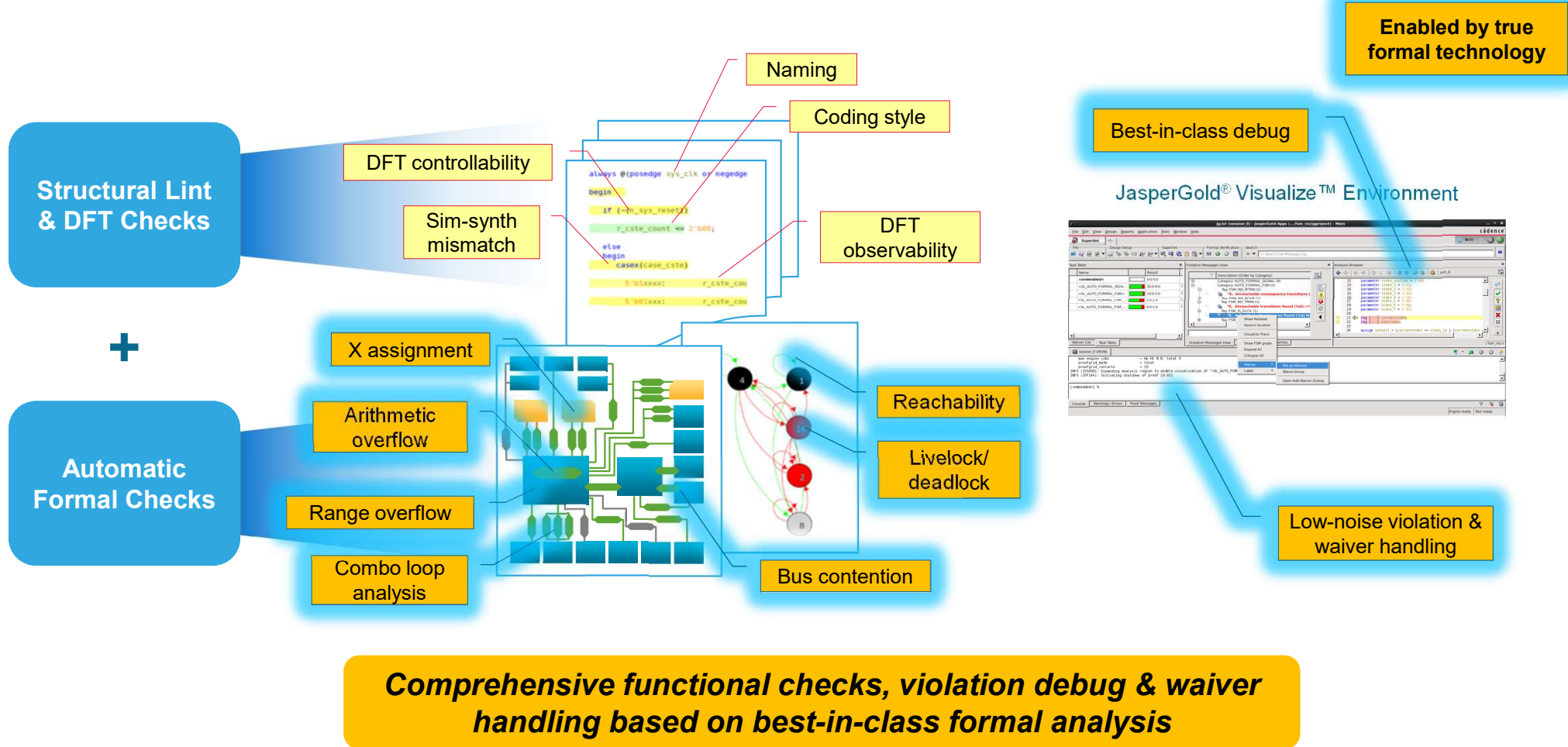
Regular RTL Signoff Includes

- Signoff the RTL against a comprehensive set of structure **Lint/DFT/CDC/RDC** checks
 - Jasper Superlint App Checks
 - Structural Lint Checks
 - Structural DFT Checks
 - Automatic Formal Checks

Structural Checks should be augmented with Formal Assisted Automatic Checks

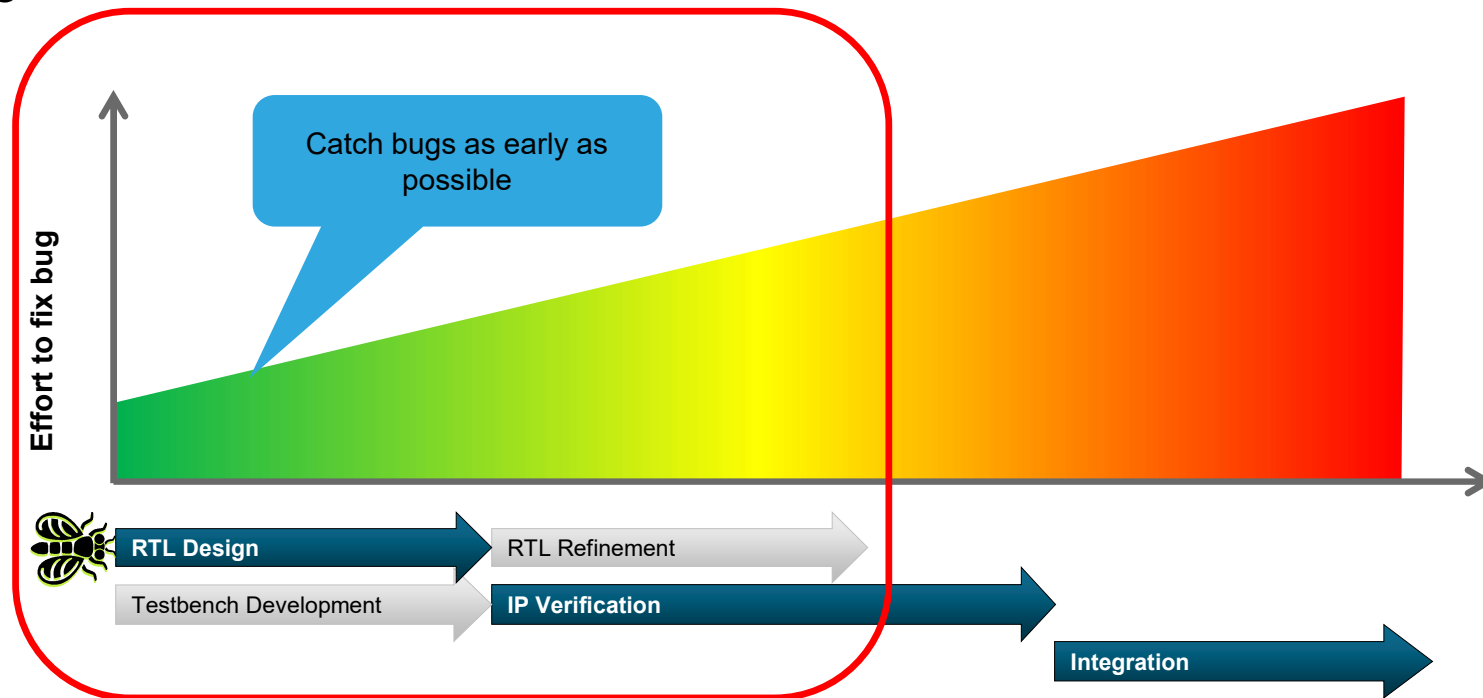
- High-value Auto-Formal Checks
 - Coverage checks: Dead code, FSM state reachability, Signal toggle and more...
 - Functional checks: FSM Livelock/Deadlock, bus contention, Auto-X check and more...

Jasper Superlint Overview



Need for Early Design Checking

- Effort to fix a bug increases significantly the further into the development cycle



Design Checking Using Superlint

- Superlint is an application which enables engineers to signoff RTL against a comprehensive set of checks
 - Ensures that issues are caught and fixed much before these travel with the RTL and become an expensive problem
- Designers
 - Early Validation
 - Use Superlint to eliminate common coding style, structural and functional design errors **before** verification starts
 - Incremental checking during design development/integration
 - **Use Superlint as a regression baseline for ongoing validation**
 - User fixes issues/creates waivers to reach a 'clean' status
- Verification Engineers
 - Complement the Formal Property Verification based bug hunting flow



Jasper Superlint Checks

Jasper Superlint checks

- Structural Lint Checks
- Structural DFT Checks
- Automatic Formal Checks

Structural Lint Checks Categories

NAMING

- Checks naming conventions on RTL elements e.g modules, instances functions atc', based on a rule file defined by user

FILEFORMAT

- Checks file formatting to make files portable and reusable e.g file name different from module name

CODINGSTYLE

- Checks to ensure there are no semantic and functional issues in the code e.g unconnected ports, unused and unassigned variables or undriven signals

SIM_SYNT

- Checks for scenarios which can cause mismatch between pre and post synthesis simulation results e.g incomplete sensitivity list

SYNTHESIS

- Checks for constructs which are not synthesizable e.g Initial statement to initialize the signal in Verilog code can result in unpredictable synthesis behavior

STRUCTURAL

- Checks for design structures which can cause functional or downstream tool issues e.g Flip-Flops missing resets

RACES

- Simulation race condition checks

Structural DFT Checks Categories

DFT_FUNCTIONAL

- Checks that your RTL complies with DFT rules and successfully inserts DFT logic in your design

DFT_SHIFT

- Checks for DFT readiness of the design in shift mode e.g Clock/Reset controllability in shift mode

DFT_CAPTURE

- Checks for DFT readiness of the design in capture mode e.g Clock drives the data pin and clock pin of one or more flip-flops in capture mode.

INTEGRATION

- Checks to ensure the consistency of constraints in case of hierarchical DFT checking e.g constraint value at full-chip is different from the value used at block level

Automatic Formal Checks Categories

DEAD_CODE

- RTL code block reachability

FSM

- FSM state reachability and Livelock / deadlock

SIGNALS

- Signal Stuck-at , Signal deadlock

BUS

- Bus contention and floating

OUT_OF_BOUND_INDEXING

- Index out of range

X_ASSIGNMENT

- A reachable x-assignment was found

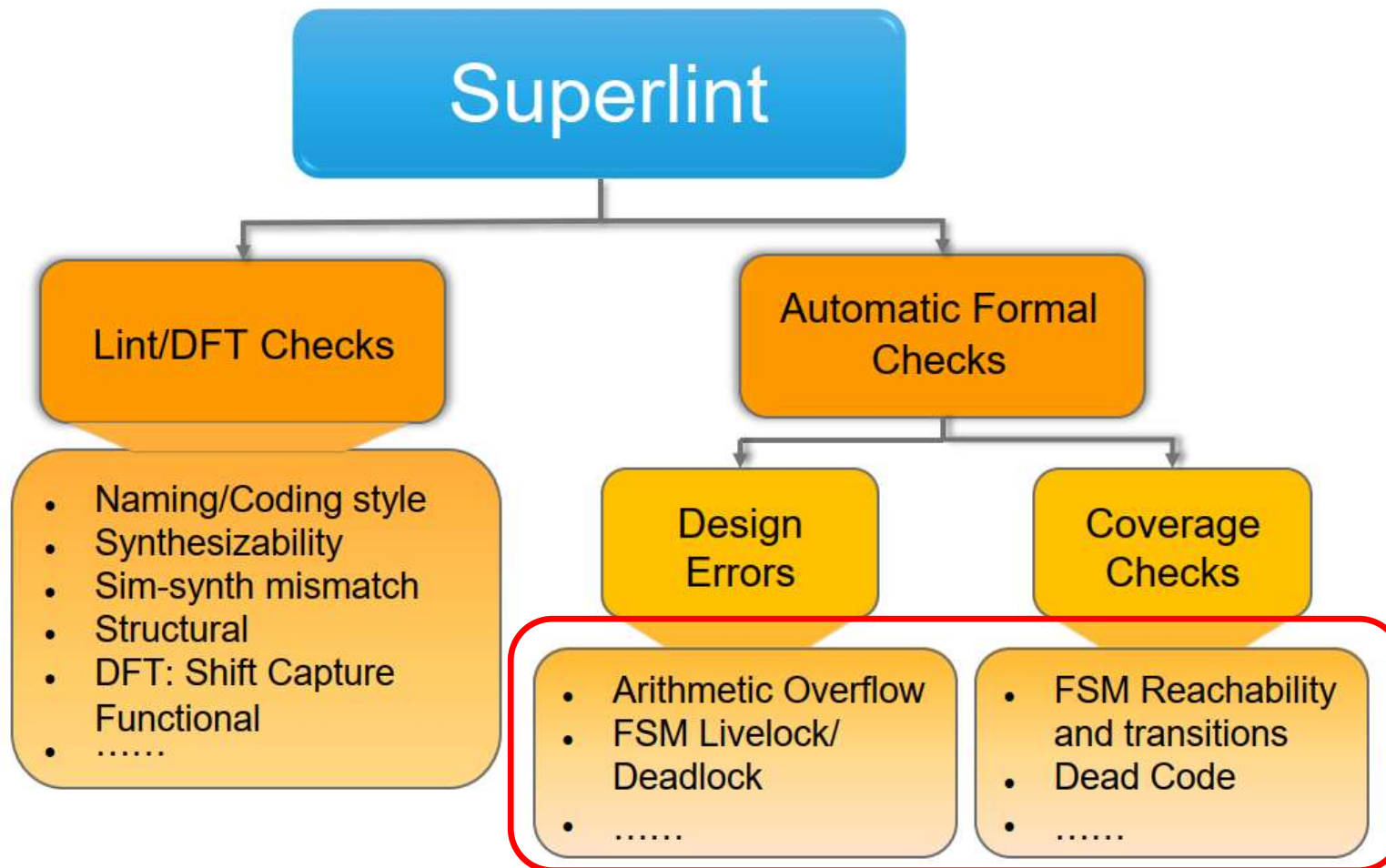
ARITHMETIC_OVERFLOW

- Overflow because of arithmetic operation



Jasper Superlint Selected AutoFormal Checks

Automatic Formal Checks Categories



AUTO_FORMAL_BUS

Check name : **BUS_IS_CONT** – more than one active driver for the bus

```
input a, b, port_a, port_b;  
output z;
```

```
assign z = a? port_a: 1'bz;  
assign z = b? port_b: 1'bz;
```



Violation is reported when 'a' and 'b' is 1.
'a' and 'b' must be mutually exclusive.

Check name : **BUS_IS_FLOT** – check if there is always at least one active driver for the bus

```
input en, port_c;  
output reg_c;
```

```
assign reg_c = en? port_c: 1'bz;
```



Violation is reported when 'en' is 0

AUTO_FORMAL_OUT_OF_BOUND_INDEXING

Check name : **ARY_IS_OOBI** – expressions and arrays are indexed within the defined range

```
input clk, en;
input [2:0] wptr, rptr;
input [3:0] din;
output [3:0] dout;

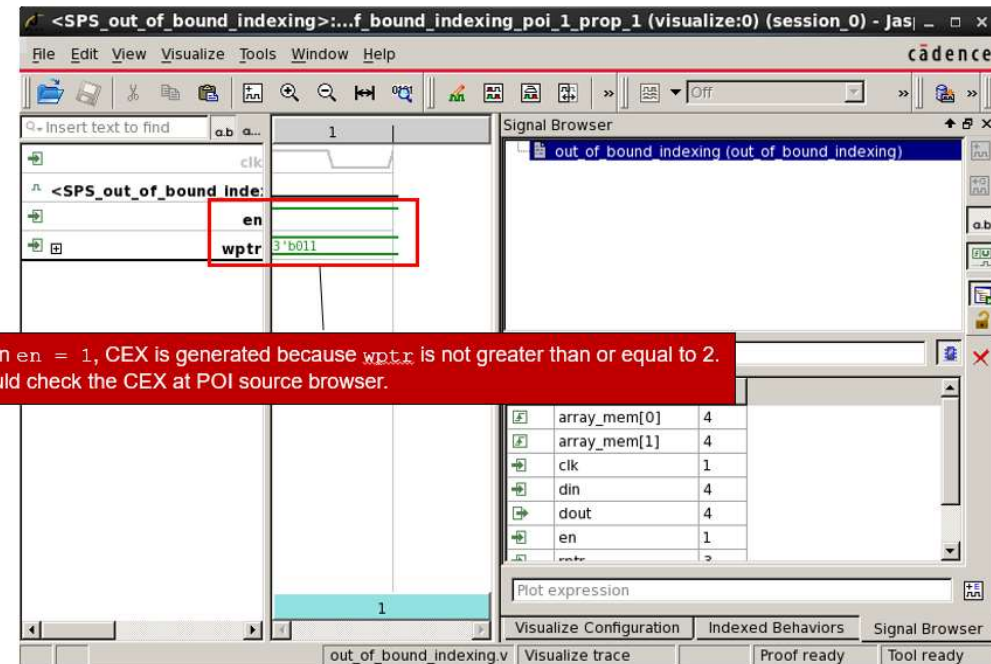
reg [3:0] arrary_mem [0:1];

always @(posedge clk)
    if(en) begin
        arrary_mem[wptr] <= din;
    end

end

assign dout = arrary_mem[rptr];
```

Violation is reported when 'wptr' or 'rptr' is greater than or equal to 2.



AUTO_FORMAL_FSM_REACHABILITY

Check name : **FSM_NO_RCHG** – a FSM state is reachable from the initial FSM state

Check name : **FSM_NO_TRRN** – a transition from one state is possible

```
input clk, resetn, start, finish;
reg [1:0] curr_state, next_state;
Parameter [1:0] s0=0, s1=1, s2=2, s3=3;

always @(posedge clk or negedge resetn)
    if(resetn) curr_state <= s0;
    else      curr_state <= next_state;
end

always @(*) begin
    next_state = s0;
    case (curr_state)
        s0 : begin
            if(start) next_state = s1;
            else      next_state = s0;
        end
        s1 : begin
            next_state = s2;
        end
        s2 : begin
            next_state = s0;
        end
        s3 : begin
            if(finish) next_state = s0;
            else      next_state = s3;
        end
    endcase
end
```



Violation is reported for state s3 as there is no next state assignment. (RCHG)



Violation is reported for state (s3→s0) and (s3→s3) because state s3 is not reachable. (TRRN)

AUTO_FORMAL_FSM_DEADLOCK_LIVELOCK_FSM

Check name : **FSM_IS_DDLK** – Deadlock condition found for FSM

- There exists no input sequence to move beyond the current state

```
input clk, resetn, start, finish;
reg [1:0] curr_state, next_state;
Parameter [1:0] s0=0, s1=1, s2=2, s3=3;

always @(posedge clk or negedge resetn)
    if(resetn) curr_state <= s0;
    else      curr_state <= next_state;
end

always @(*) begin
    next_state = s0;
    case (curr_state)
        s0 : begin
            if(start) next_state = s1;
            else      next_state = s0;
        end
        s1 : begin
            next_state = s2;
        end
        s2 : begin
            next_state = s2; // Deadlock
        end
        s3 : begin
            if(finish) next_state = s0;
            else      next_state = s3;
        end
    endcase
end
```



Violation is reported when 'curr_state' is 's2'.
After reaching 's2', the design remains in that state.

AUTO_AUTO_FORMAL_FSM_DEADLOCK_LIVELOCK_FSM

Check name : **FSM_IS_LVLK** – Livelock condition found for FSM

- There exists no input sequence to return to the idle/reset state

```
input clk, resetn, start, finish;
reg [1:0] curr_state, next_state;
Parameter [1:0] s0=0, s1=1, s2=2, s3=3;

always @(posedge clk or negedge resetn)
    if(resetn) curr_state <= s0;
    else      curr_state <= next_state;
end

always @(*) begin
    next_state = s0;
    case (curr_state)
        s0 : begin
            if(start) next_state = s1;
            else      next_state = s0;
        end
        s1 : begin
            next_state = s2;
        end
        s2 : begin
            next_state = s3;
        end
        s3 : begin
            next_state = s2;
        end
    endcase
end
```



After reaching 's2', there exists no input sequence can return to 's0'

AUTO_FORMAL_X_ASSIGNMENT

Check name : **ASG_IS_XRCH** – x variables cannot be reached

- If an x-assignment is reachable, it becomes an active source of x, and can lead to unexpected functionality
- x equals 1 or 0 in synthesis and is unknown in simulation

```
always @(*) begin
  case(sel)
    0 : out = a;
    1 : out = b;
    2 : out = c;
    3 : out = 1'b1;
  endcase
end
```



Violation is reported when 'sel' = 3.

The screenshot shows the Cadence JasperGold interface. The main window displays the source code for a module named `<SPS_x_assignment>::SPS_x_assignment_poi_1_prop_1`. The code is a Verilog-like case statement for a multiplexer. The signal `sel` is highlighted in red, and its value is shown as `2'b11`. A red box highlights the `sel` signal in the code. The Signal Browser on the right shows the signal `x_assignment(x_assignment)` and its value `2'b11`. A red banner at the bottom of the interface contains the following text:

- When `sel` = 3, CEX is generated
- Should check the CEX at POI source browser.

The Signal Browser table shows the following signals and their sizes:

Signal	Size
a	4
b	4
c	4
clk	1
out	4
sel	2

AUTO_FORMAL_SIGNALS

Signal type checkers default create at flop and design output, user can configure signal type or add customized signal list by user

Check name : **SIG_IS_STCK**– RTL logic elements are never stuck at a constant value

```
output [1:0] BID;  
assign BID = 2'b00;
```



JG default will filter out this kind of constant assignment

```
reg port_b;  
always @(posedge clk or negedge rst) begin  
    if (!rst)  
        port_b <= 1'b1;  
    else  
        port_b <= port_a | 1'b1;  
end
```



Violation is reported
because 'port_b' is stuck at a constant value(1'b1)

Check name : **SIG_NO_TGFL**– signals have toggled from 0 to 1

Check name : **SIG_NO_TGRS**– signals have toggled from 1 to 0

AUTO_FORMAL_DEAD_CODE

Check name : **BLK_NO_RCHB** – RTL provides access to all branches

```
assign sel = 1'b1;  
always @(*) begin  
    if (sel)  
        out = a;  
    else  
        out = b;  
end
```



A violation is reported for sel==1'b0 branch

AUTO_FORMAL_COMB_LOOP

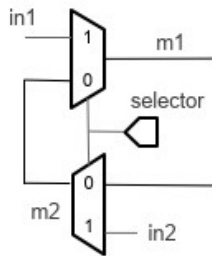
Check name : **MOD_IS_FCMB** – functionally true combinational loop

```
module MOD_IS_FCMB (out_a, in_a, in_b);  
  input in_a, in_b;  
  output out_a;  
  test_or a1(.in_a(in_a), .in_b(in_b), .out_a(out_a));  
  test_nand b1(.in_a(out_a), .in_b(in_b), .out_a(in_a));  
endmodule
```



Violation is reported
Because a combinational loop
detected is passing through a1.in_a

- Functionally true/false loops



selector == 1'b1 can remove circular
dependency

AUTO_FORMAL_OVERFLOW

Check name : **ASG_AR_OVFL** – RTL contains no errors that could cause the array to overflow.

```
module MOD_IS_OVFL (out_a, in_a, in_b);  
    input [3:0] in_a,  
    input [1:0] in_b;  
    output[1:0] out_c;  
    assign out_c = in_a + in_b;  
endmodule
```



Violations are reported because $in_a + in_b$ could lead to overflows.

EX:

- As $in_a = 4'b0001$ and $in_b = 2'b11$, it leads to ASG_AR_OVFL violation.



Jasper Superlint Setup

Superlint User Interface

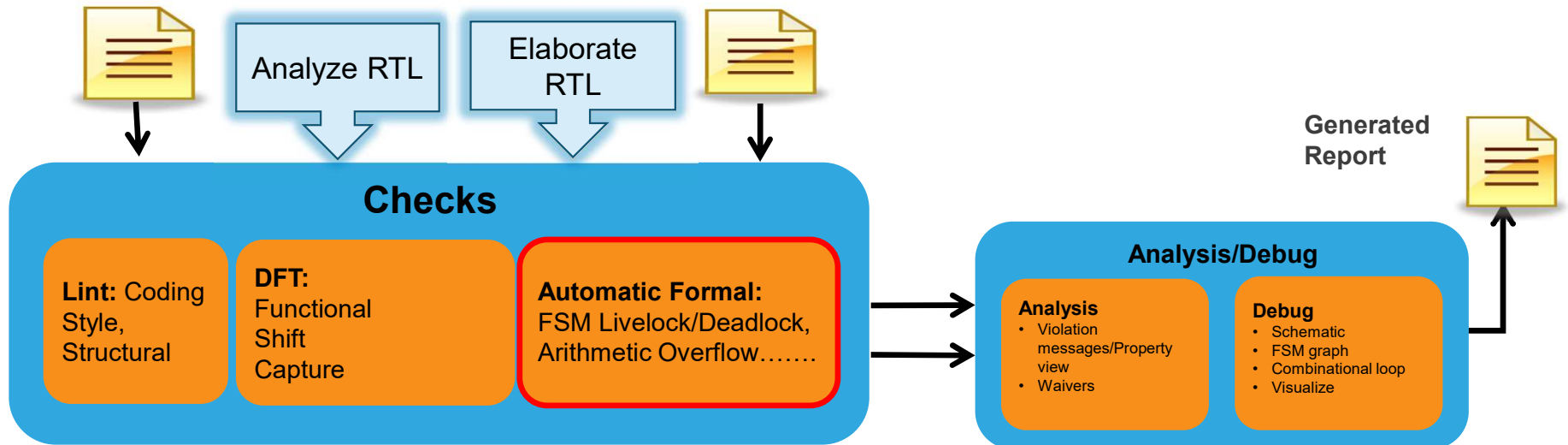
```
jg -superlint test.tcl
```

Custom Rule File/Configure Checks

- Custom categories
- Selected checks
- Custom parameters

Custom TCL File

- Clocks
- Reset
- Selected checks
- DFT Constraints



Creating a Custom Rule File

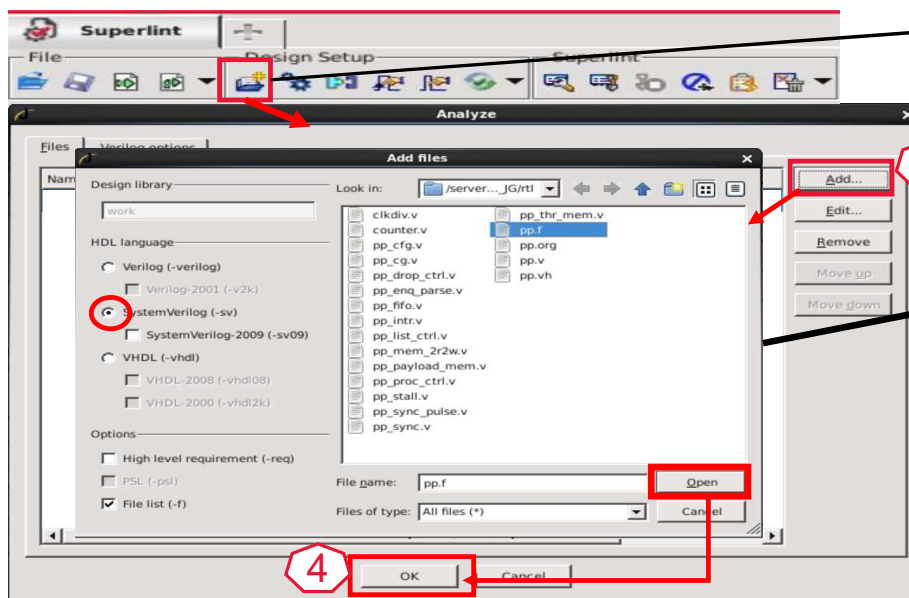
- Rules file contains the list of all LINT, DFT, and AutoFormal checks
 - The default rules file, Superlint.def, which contains the default Superlint categories and checks is located at: <jg_install> /etc/res/rtlds/rules/superlint.def
- User can create a custom rule file with a selected set of checks and customize the selected checks (**suggested flow**)
 - We will provide a list of suggested autoformal rule sets after the training
- Customizations that can be done by the user include
 - Category names
 - Checks included in each category
 - Enable or disable a category
 - Severity
 - Short message
 - Check name
 - Specific parameter for a check e.g pattern for naming convention checks

Analyze RTL

- RTL can be analyzed through tcl commands or GUI as shown:
- Tcl command examples:

```
analyze -v2k my_file.v
```

```
analyze -sv -f file_list.f +define+FPGA+
```
- Follow the bellow steps to analyze the RTL through GUI



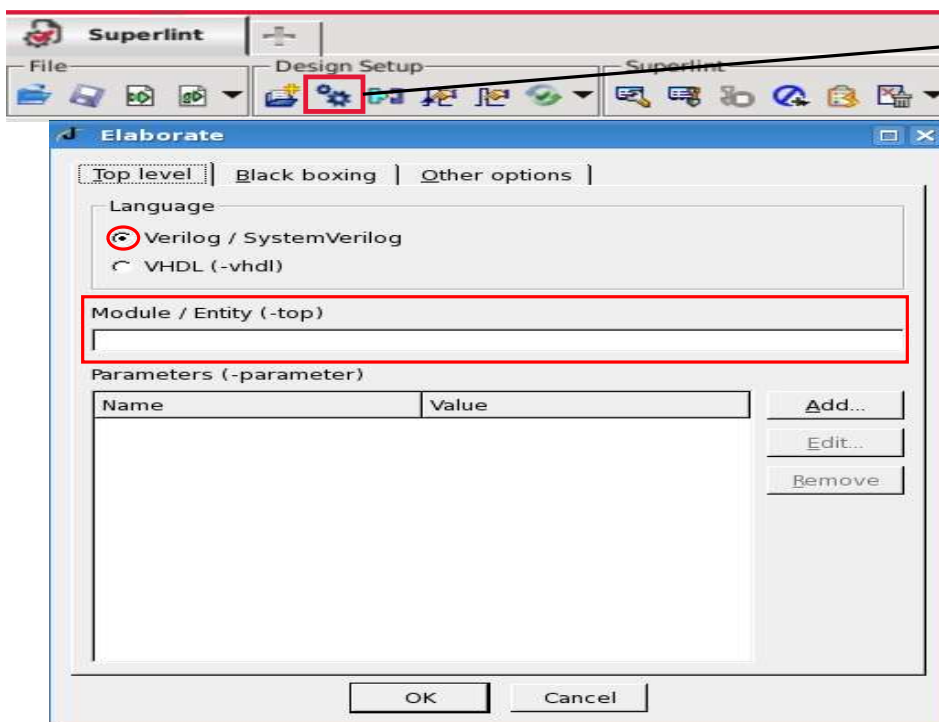
Click 'Analyze RTL' button

In 'Add files' dialogue:

- Choose the directory where the file is located from the Look in field
 - Select one or more filenames in the displayed list
- Choose from the following to specify an HDL language:
 - Verilog and Verilog 2001
 - SystemVerilog and SystemVerilog 2009
 - VHDL, VHDL 2000, VHDL 2008
- Click 'Open'
 - The Add files dialog closes and the Analyze dialog (Files view) returns
 - If necessary, repeat the previous steps for additional

Elaborate RTL

- Elaborate is to specify how you want to synthesize and read the netlist and extract verification tasks for files that have been analyzed
- Tcl command to Elaborate RTL is: `elaborate -top <top_module_name>`



Button to "Elaborate RTL"

- The Elaborate dialog appears, and you can begin specifying elaboration under the Top level tab
- Click a radio button to specify the Language of the top module of your design RTL
 - Verilog / SystemVerilog
 - VHDL
- In the Module / Entity field, type the top module's name

Clock Specification

- TCL clock command specifies the characteristics of how the clock is driven during a formal analysis run and clock related configuration
- Clock commands are cumulative
 - Multiple clock commands are required to specify multiple clocks
- Before you can run a proof or visualize a violation, Clock needs to be specified using one of the following:
 - “*clock <clk>*” to declare a signal as a clock
 - “*clock -none*” for free-running clock environment
 - “*clock -infer*” to specify that you want the tool to automatically infer primary clocks
 - “*clock -analyze*” returns a Tcl list containing inferred clocking signals
- Clock arguments must be a signal or negation of a signal, but cannot be an expression
 - Example:
 - *clock clk*
 - *clock ~clk*

Reset Specification

- This is used to specify the reset condition for the design under verification
- Reset conditions are always 'globally' applied and noncumulative
 - Multiple resets need to be specified in one command
 - Most recent reset condition overrides the condition which was previously defined
- Before you can run reset analysis, including the one that runs for "prove" and "visualize", reset needs to be specified using one of the following:
 - “reset <rst>” to declare a reset pin
 - “reset -sequence” to set a reset sequence file
 - “reset -init_state” to set a reset snapshot file
 - “reset -none” to specify a free-running reset environment
 - “reset -analyze” to get information about resets and set up the reset environment properly
 - Example :
 - *reset { ~rstn ~n_preset }*



Jasper Superlint Key Productivity Features

Superlint Key Productivity Features

Customized Failure Analysis – GUI Violation Layout

- Two layouts cater to two different types of users: violation only or violation + property

Persistent Waivers

- Waivers are persistent across RTL code changes and also across the hierarchy

Grouping

- Related violations are grouped to enable efficient analysis. Fixing primary violation might fix all the secondary violations in the group

Observability filter for Auto-Formal functional checks

- CEX is reported only if the failure gets propagated to an observable boundary signal e.g Flip-Flop or an output port (ARITHMETIC_OVERFLOW, X_ASSIGNMENT, OUT_OF_BOUND_INDEXING, BUS)

Leverage formal to reduce Structural Lint Noise

Violation Layout

For engineers primarily focusing on check violations without any consideration of the formal properties

The screenshot displays the Cadence Superlint interface with the following components:

- Violation Messages View:** A list of rule violations, including tags like MOD_NR_EBLK, OPR_NR_UCMP, and OTP_NR_ASYA.
- Violation tree:** A tree view showing the hierarchy of violations.
- Source code:** A window showing the Verilog code for the module `pp_sync_pulse`, with a callout pointing to the `output bit busy` line.
- Waiver List:** A table listing waived violations, including `arithmetic_overflow` and `EXP_IS_OVFL`.

The interface also includes a menu bar, a toolbar, and a status bar at the bottom.

Comment	Id	Source	Up	Tag	Category	Module	Inst
arithmetic_overflow	3			EXP_IS_OVFL	*	*	list
arithmetic_overflow	4			EXP_IS_OVFL	*	*	thr
arithmetic_overflow	5			EXP_IS_OVFL	*	*	*

Formal Property Layout

For engineers focusing on formal properties and proof results

The screenshot displays the Cadence JasperGold IDE interface for formal verification. The main window is titled "run_bridge.tcl (session_0) - JasperGold Apps (.../sharedRTL/jgproject) - Main". The interface is divided into several panes:

- Task Tree:** Located on the left, it shows a list of tasks. A callout labeled "Task table" points to this pane.
- Automatic Formal Properties:** A table in the center-left pane listing properties. A callout labeled "Property table" points to this pane. The table has columns for Tag, Type, and Name. It lists several properties, including "dead_code_prop_66", "dead_code_prop_69", "dead_code_prop_72", "arithmetic_overflow_prop_10", "arithmetic_overflow_prop_11", and "arithmetic_overflow_prop_12".
- Analysis Browser:** Located on the right, it shows a list of analysis results. A callout labeled "Source code" points to this pane.
- Source code editor:** The bottom right pane shows the source code for "bridge.v". It contains Verilog code for a FIFO and a read/write operation.

The bottom of the interface includes a console area with tabs for "Console", "Warnings / Errors", and "Proof Messages".

Persistent Waivers

The screenshot displays the Cadence Superlint interface. The 'Violation Messages View' on the left lists several violations, with the selected one being: "Output port 'cfg.GEN_INTR_CLEAR_SYNC[12].sync_cfg_intr_clear_busy' is assigned asynchronously". A right-click context menu is open over this violation, with the 'Waive' option selected. This opens a 'Waiver Comment' dialog box where a comment is being entered: "t' is assigned asynchronously". Below the dialog, a 'Waiver List' window is visible, showing a table of existing waivers.

Waiver Comment

Please add a comment to create the waiver:

☐ Waive as a group

Waiver List

Comment	Id	Source	Up	Tag	Category	Module
arithmetic_overflow	3			EXP_IS_OVFL	*	*
arithmetic_overflow	4			EXP_IS_OVFL	*	*
arithmetic_overflow						
arithmetic_overflow						

Violations View:

- Tag: MOD_NR_EBLK (2)
- Tag: OPR_NR_UCMP (7)
- Tag: OPR_NR_UEOP (1)
- Tag: OPR_NR_UEAS (11)
- Tag: OTP_NR_ASYA (129)
- "Output port 'cfg.GEN_INTR_CLEAR_SYNC[12].sync_cfg_intr_clear_busy' is assigned asynchronously"

RTL Code:

```
1 include "pp.vh"
2
3 module pp_sync_pulse (
4     input bit clk1,
5     input bit clk2,
6     input bit rstn1,
7     input bit rstn2,
8     input bit in,
9     output bit out,
10    output bit busy
11);
12
13 bit a1, a2, a3;
14 bit b1, b2, b3;
15
```

Annotations:

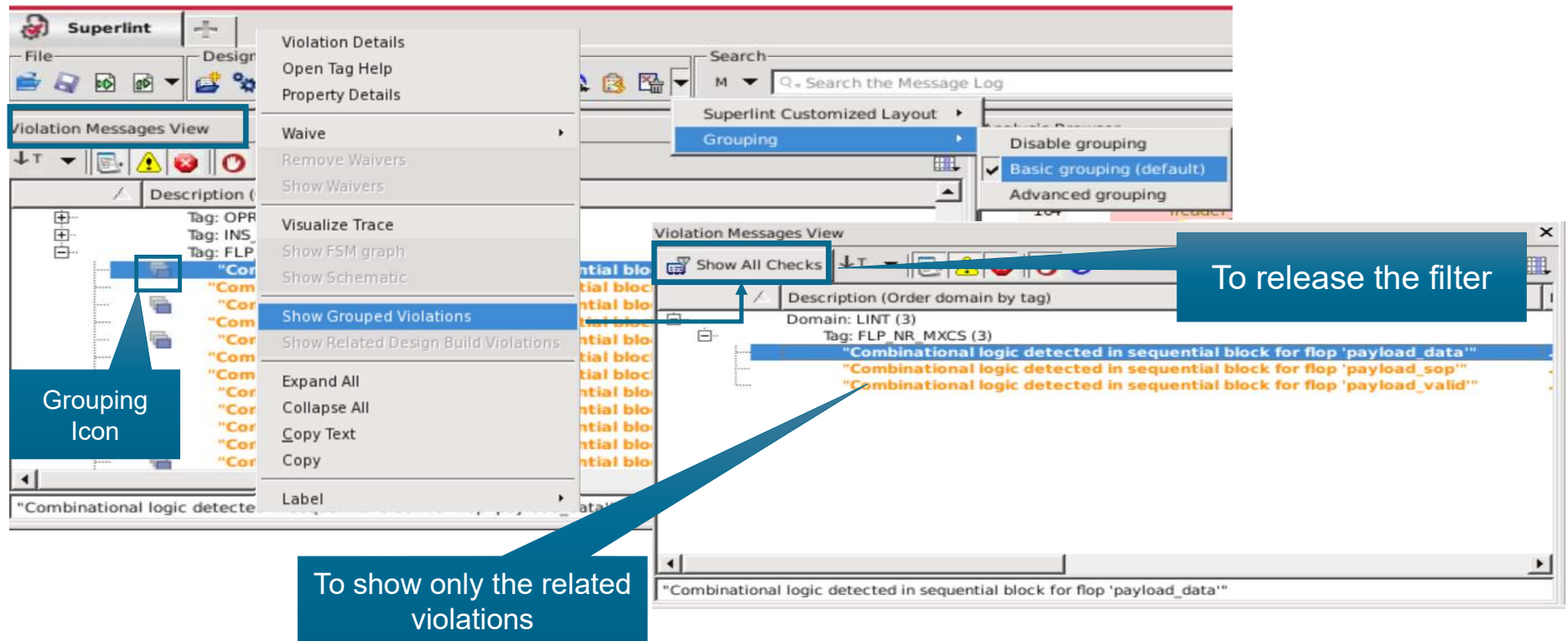
- Select 'Waive' on right mouse button click
- Waiver Comment

Waivers are persistent across RTL code changes and also across hierarchy. Waivers are highly portable as those are stored as tcl commands

Grouping for Efficient Analysis

The tool groups similar checks

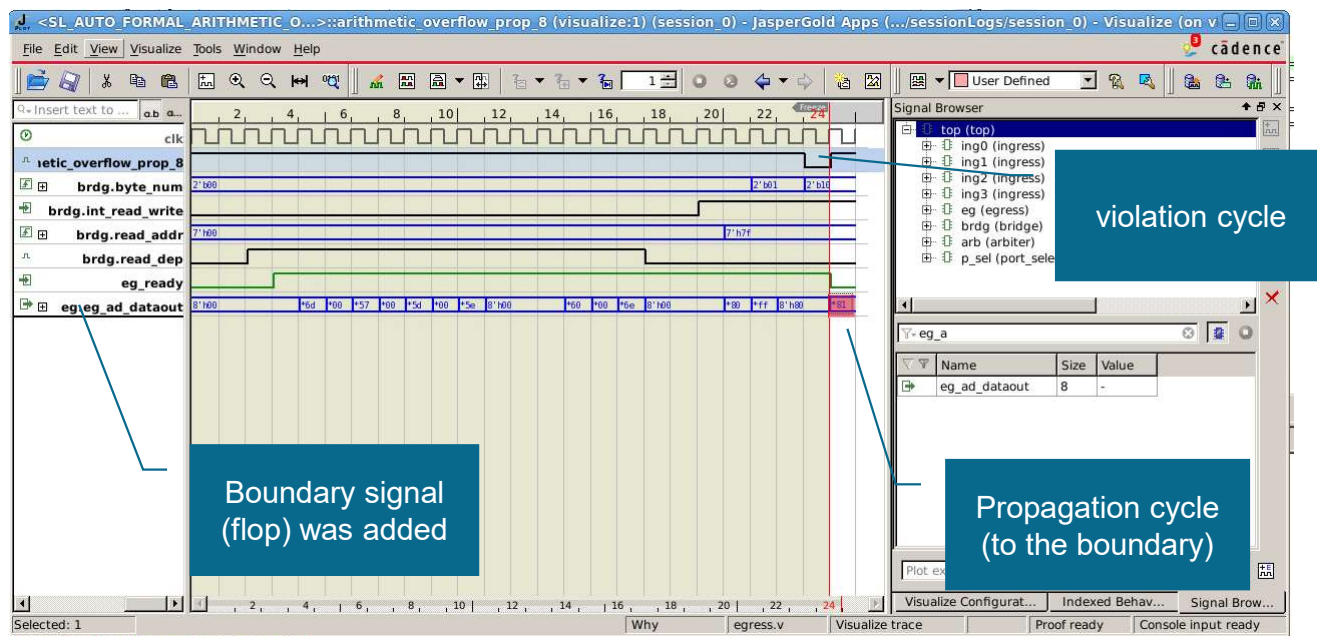
- Shows only the primary check-in property table or violation message view and adds a grouping icon to the left of this check



Visualize: Auto Formal - Observability Enabled Debug

Better debugging of observable violations with Visualize

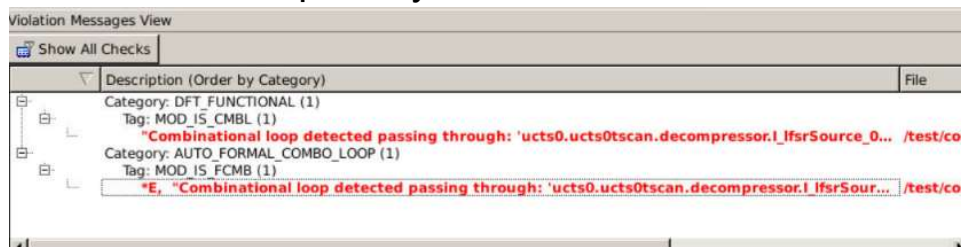
- Double click an observable violation
- Boundary signals are added to the window



Leverage Formal to Optimize Lint Results

For Lint checks which have a corresponding formal check

- If the formal property is **violated**, the corresponding lint violation is grouped with it and the formal violation is shown as the primary



2022.06FCS	
Lint/DFT Tag	Formal Tag
ASG_MS_RPAD	ASG_AR_OVFL
ASG_NR_POVF	ASG_AR_OVFL
MOD_NR_SYXZ	ASG_IS_XRCH
IDX_NR_ORNG	ARY_IS_OOBI
IDX_NR_LBOU	ARY_IS_OOBI
MOD_NR_SYXZ	BUS_IS_FLOT
MOD_IS_CMBL	MOD_IS_FCMB
SIG_IS_MDRV	BUS_IS_CONT
CST_NR_MSBZ	BUS_IS_FLOT
ASG_MS_RTRU	ASG_AR_OVFL

```
module IDX_NR_ORNG (count, d, q1);
  output q1;
  input [3:0] d;
  input [2:0] count;
  assign q1 = d[count];;
endmodule
```

- If the formal property is **proven**, the corresponding lint violation is automatically **waived** and shown with a distinct icon in the waiver table

Up	Id	Source	Tag	Category	Comment
	1		MOD...	*	Auto waiver generated by AUTO-FORMAL domain, tag: MOD_IS_FCMB (p...
	2		MOD...	*	Auto waiver generated by AUTO-FORMAL domain, tag: MOD_IS_FCMB (p...

Automatic
waivers are
generated

Auto-waivers can be accepted and turned
into TCL waivers – which will make them
exportable and allow reusing in next runs



Summary

Best-in-class 3rd-generation Smart Jasper

- Formal is now a mainstream verification technology
- Formal is growing rapidly in the verification mix: complementary to simulation
- Industry's leading formal technology is Jasper from Cadence

Formal Technology Leadership =

- more verification
- in less time
- on bigger designs
- with optimal compute resource (in-house or cloud)

cā dence[®]