2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION

UNITED STATES

SAN JOSE, CA, USA

FEBRUARY 27-MARCH 2, 2023

A Simulation Expert's Guide to Formally Proving SW Status and Interrupts

Neil Johnson

Ciena

accellera SYSTEMS INITIATIVE

Simulation vs Formal

Simulation		Formal Property Checking
Verify functional correctness	**What**	Verify functional correctness
Passing tests	**How**	Proven properties
<Simulator>	**Tool**	*<Formal Tool>*
• Test status • Code coverage • Functional coverage	**Closure Metrics**	• Proof status • Code coverage • Functional coverage

Simulation vs Formal

Simulation		Formal Property Checking
Verify functional correctness	**What**	Verify functional correctness
Passing tests	**How**	Proven properties
<Simulator>	**Tool**	*<Formal Tool>*
• Test status • Code coverage • Functional coverage	**Closure Metrics**	• Proof status • Code coverage • Functional coverage
• Default/known technology	**Upsides**	• Low infrastructure requirement • Exhaustive proofs • Implicit code coverage closure* • Low re-config cost
• High infrastructure requirement • Constrained random unknowns • Iterative coverage closure • High re-config cost	**Downsides**	• Applicability is TBD • Ramp-up/learning curve • Technology limitations • Depth/breadth of logic

SIMULATION VS FORMAL

Simulation		Formal Property Checking
Verify functional correctness	**What**	Verify functional correctness
Passing tests	**How**	Proven properties
<Simulator>	**Tool**	*<Formal Tool>*
• Test status rage l coverage	**Closure Metrics**	• Proof status • Code coverage • Functional coverage
own technology	**Upsides**	• Low infrastructure requirement • Exhaustive proofs • Implicit code coverage closure* • Low re-config cost
structure requirement ed random unknowns overage closure re-config cost	**Downsides**	• Applicability is TBD • Ramp-up/learning curve • Technology limitations • Depth/breadth of logic
• **Because**	**Motivation**	• **Eliminate sim cycles** • **Increase confidence**

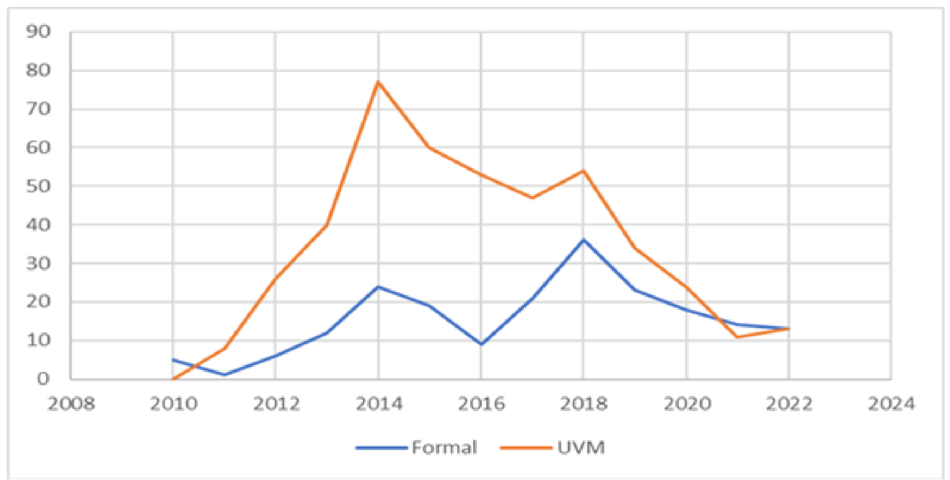


Figure 2 - DVCon Keyword Search Data 2010-2022: Number of Keyword Hits

Simulation vs Formal

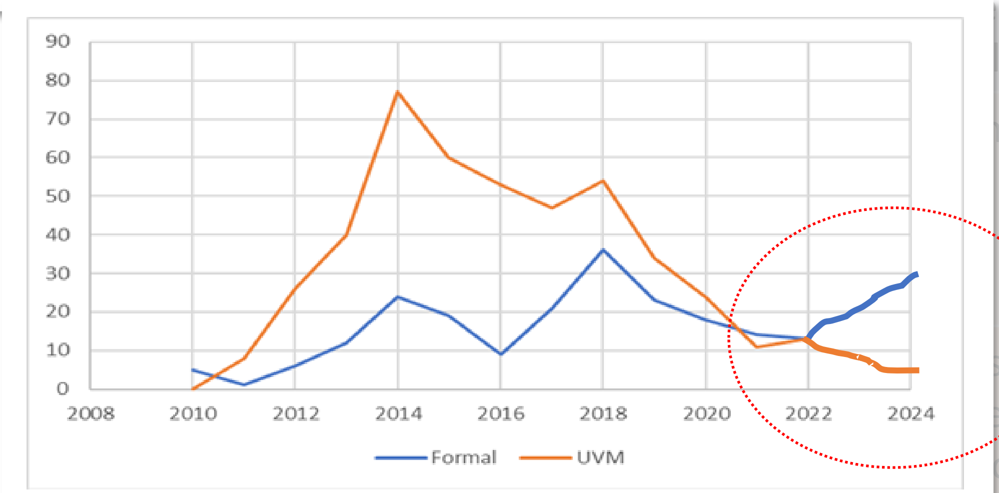


Figure 2 - DVCon Keyword Search Data 2010-2022: Number of Keyword Hits

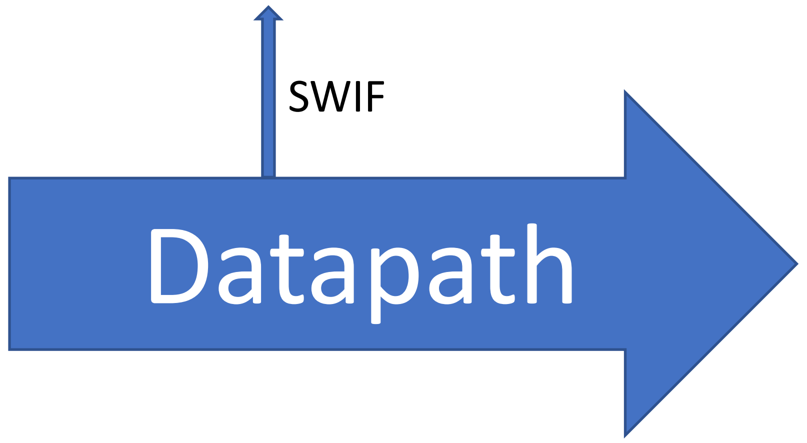
Simulation		Formal Property Checking
Verify functional cor		rrectness
Passing tests		
<Simulator>		
• Test status rage cove		age
own t		e requirement s • Implicit code coverage closure* • Low re-config cost
structure requirement d random unknowns overage closure re-config cost	Downsides	• Applicability is TBD ? • Ramp-up/learning curve • Technology limitations • Depth/breadth of logic
• ~~Because~~	Motivation	• **Eliminate sim cycles** ✓ • **Increase confidence**

Where to start…?

- Module level proofs
  - *aka: unit proofs… like simulated unit tests*
- SWIF Status/Interrupt Checking

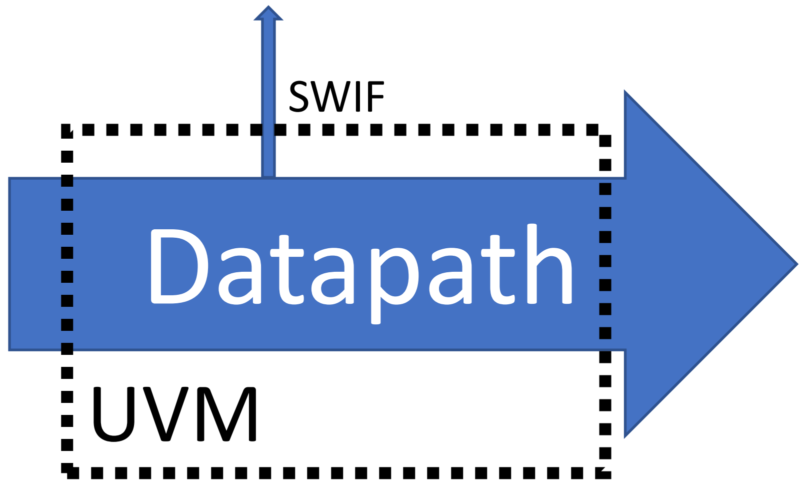
SWIF Interrupt/Status Checking

- Habitually difficult in simulation
  - UVM testbenches are architected around core functionality
  - Status/interrupt checking are an afterthought/overlay



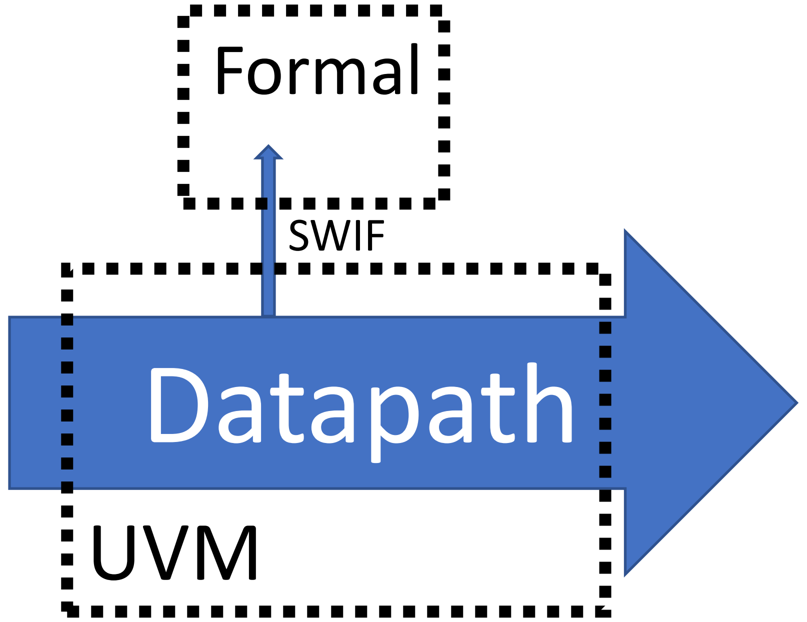
SWIF Interrupt/Status Checking

- Habitually difficult in simulation
  - UVM testbenches are architected around core functionality
  - Status/interrupt checking are an afterthought/overlay



SWIF Interrupt/Status Checking

- Habitually difficult in simulation
  - UVM testbenches are architected around core functionality
  - Status/interrupt checking are an afterthought/overlay
- Formal enables a more deliberate approach
  - Dedicated checking without the infrastructure/retrofits requirements



SWIF Interrupt/Status Checking

- Habitually difficult in simulation
  - UVM testbenches are architected around core functionality
  - Status/interrupt checking are an afterthought/overlay
- Formal enables a more deliberate approach
  - Dedicated checking without the infrastructure/retrofits requirements

Method

1. Map each status bit to CTRL/IO
2. Capture a checker strategy/feasibility
3. Build properties to verify each status output
4. Document/review the outcome

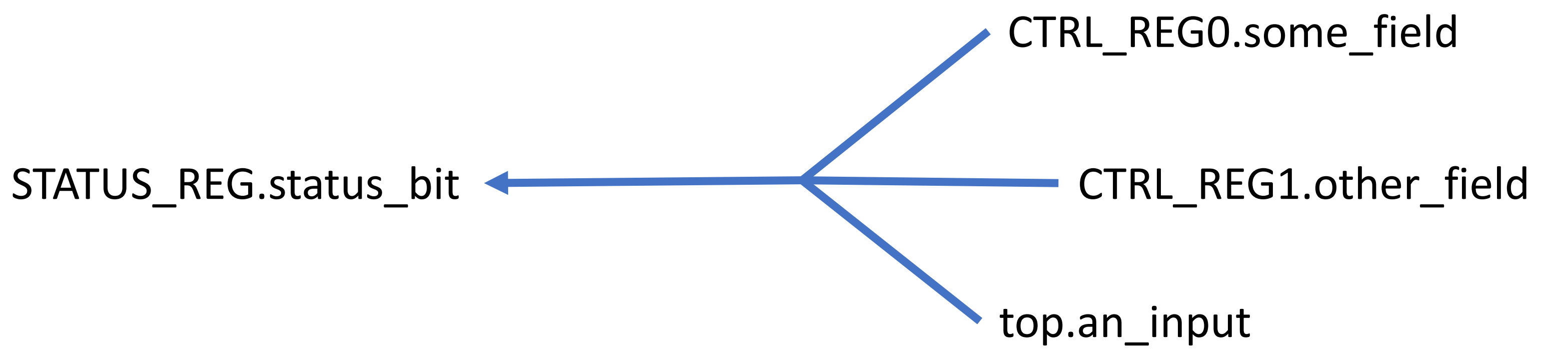
SWIF Interrupt/Status Checking

STATUS_REG.status_bit

Method

1. Map each status bit to CTRL/IO
2. Capture a checker strategy/feasibility
3. Build properties to verify each status output
4. Document/review the outcome

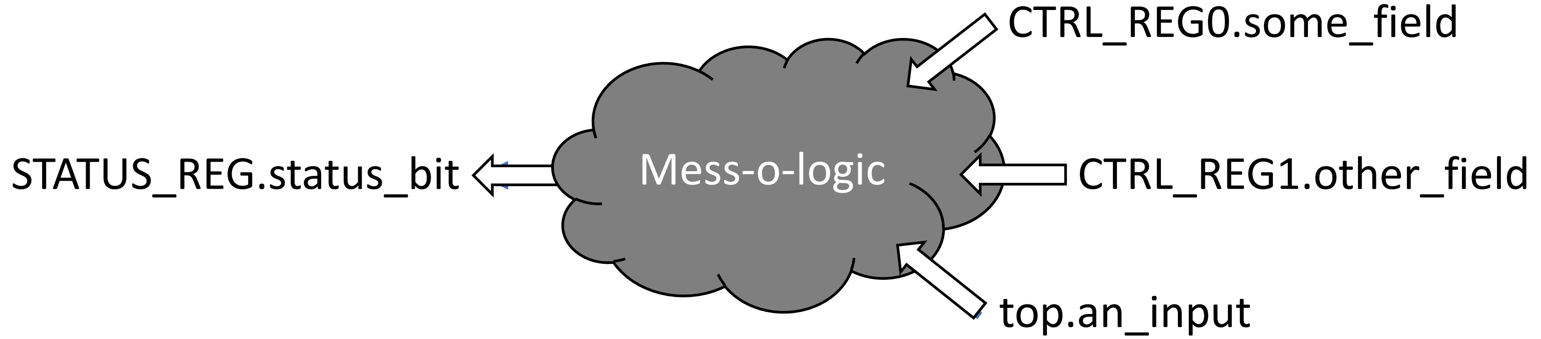
SWIF Interrupt/Status Checking



Method

➡ Map each status bit to CTRL/IO
2. Capture a checker strategy/feasibility
3. Build properties to verify each status output
4. Document/review the outcome

SWIF Interrupt/Status Checking



Method

1. Map each status bit to CTRL/IO
2. ➡ Capture a checker strategy/feasibility
3. Build properties to verify each status output
4. Document/review the outcome

Strategy	Output(s)	Limitations
General plan of attack	Status bits under test	Fear, uncertainty, doubt, etc.

SWIF Interrupt/Status Checking

```
property status_bit_asserted;
  @(posedge i_clk)
  disable iff (!i_sresetn)
    some_field_seq and
    other_field_seq and
    an_input_seq |->
      status_bit
endproperty
assert property (status_bit_asserted);
```

Method

1. Map each status bit to CTRL/IO
2. Capture a checker strategy/feasibility
3. ➡ Build properties to verify each status output
4. Document/review the outcome

SWIF Interrupt/Status Checking

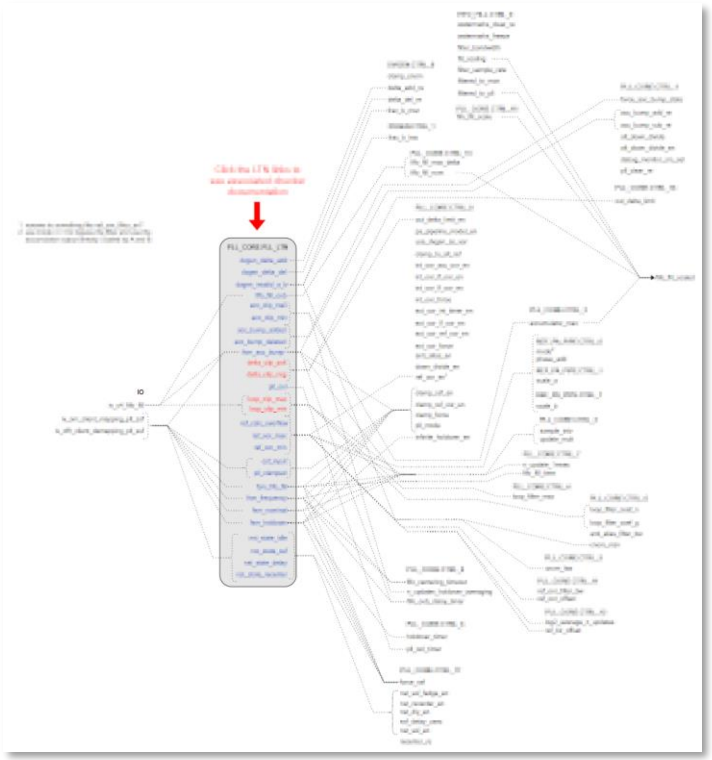
Property	When condition…	Check this…
status_bit_asserted	some_field and another_field and an_input <do something>	status_bit is asserted

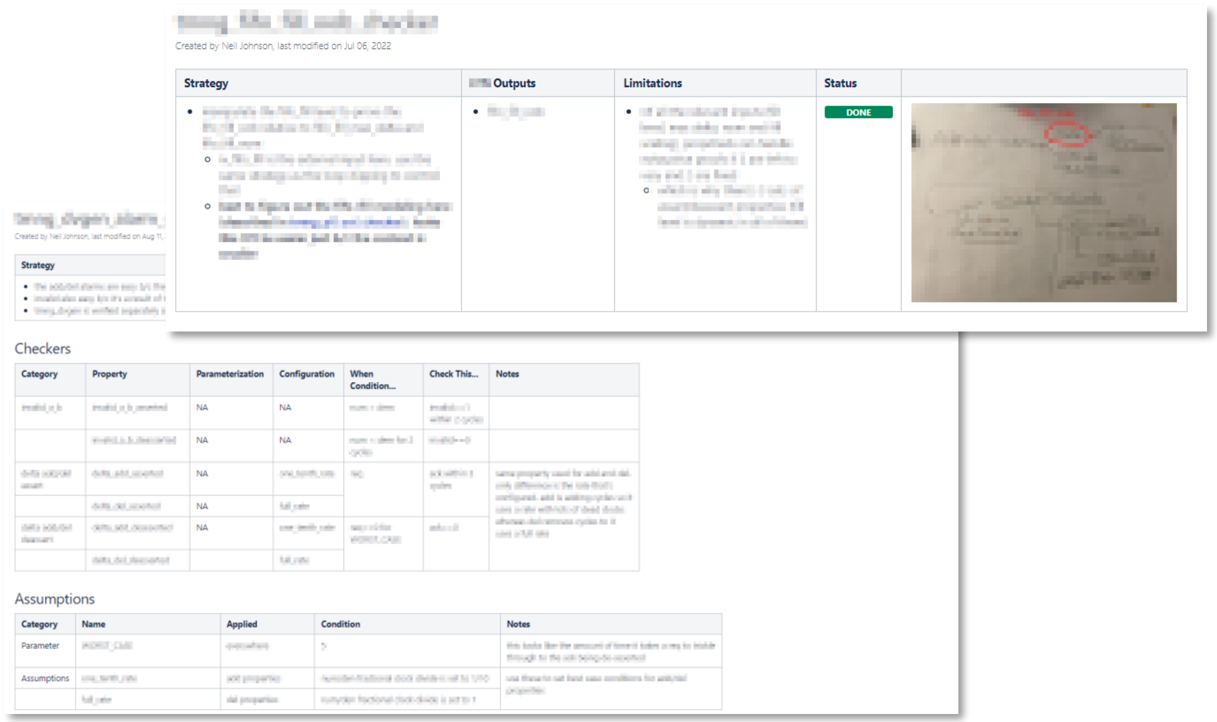
Method

1. Map each status bit to CTRL/IO
2. Capture a checker strategy/feasibility
3. Build properties to verify each status output

➡ Document/review the outcome

Example Artifacts





[illegible]

Created by Neil Johnson, last modified on Jul 06, 2022

Strategy	Outputs	Limitations	Status	
[illegible]	[illegible]	[illegible]	DONE	[illegible]

[illegible]

Created by Neil Johnson, last modified on Aug 11, [illegible]

Strategy
[illegible]

Checkers

Category	Property	Parameterization	Configuration	When Condition...	Check This...	Notes
[illegible]	[illegible]	NA	NA	[illegible]	[illegible]	
	[illegible]	NA	NA	[illegible]	[illegible]	
[illegible]	[illegible]	NA	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	NA	[illegible]			
[illegible]	[illegible]	NA	[illegible]	[illegible]	[illegible]	
	[illegible]		[illegible]			

Assumptions

Category	Name	Applied	Condition	Notes
Parameter	[illegible]	[illegible]	[illegible]	[illegible]
Assumptions	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	

Results/Observations

- 15 Status outputs verified
  - 6 bugs found w/69 properties
- No dependency on simulation infrastructure
  - Complementary but completely orthogonal
- Mapping was very useful
  - Low-level capture of low-level relationships
  - Documentation sparse but practical
  - Some outputs “didn’t fit” into formal
- Light on infrastructure
  - Use helper logic only when necessary
  - Keep the *entire* check simple as possible

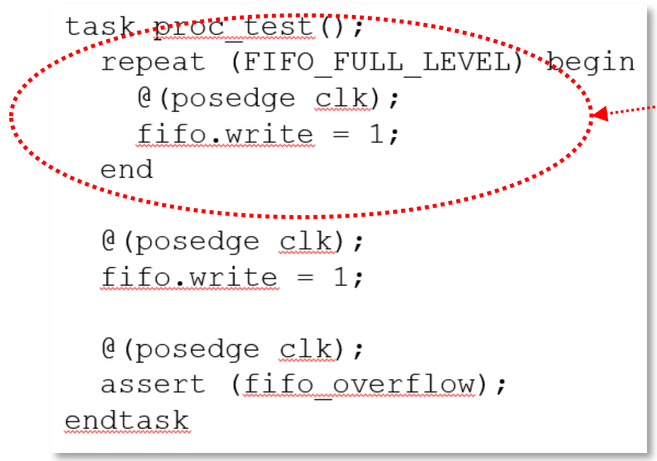
Results/Observations

- 15 Status outputs verified
  - 6 bugs found w/69 properties
- No dependency on simulation infrastructure
  - Complementary but completely orthogonal
- Mapping was very useful
  - Low-level capture of low-level relationships
  - Documentation sparse but practical
  - Some outputs “didn’t fit” into formal
- Light on infrastructure
  - Use helper logic only when necessary
  - Keep the *entire* check simple as possible

```
always @(posedge clk)
begin
  // helper logic
end

property p;
  // checker logic
endproperty
assert p;
```

Lessons Learned

- Assuming checker execution state
  - i.e. Overflow on FIFO full && write

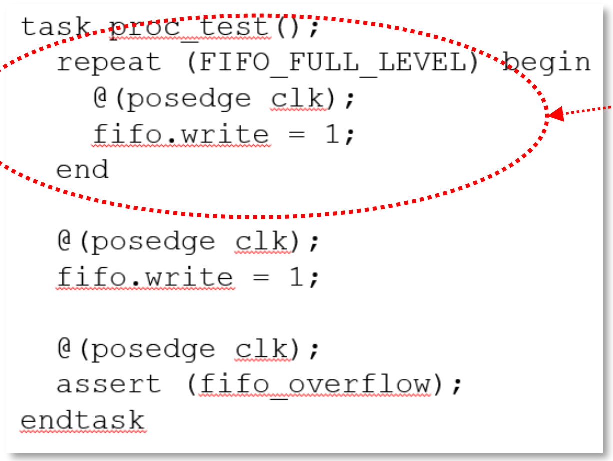


```
task proc_test();
  repeat (FIFO_FULL_LEVEL) begin
    @(posedge clk);
    fifo.write = 1;
  end

  @(posedge clk);
  fifo.write = 1;

  @(posedge clk);
  assert (fifo_overflow);
endtask
```

Reaching execution state procedurally

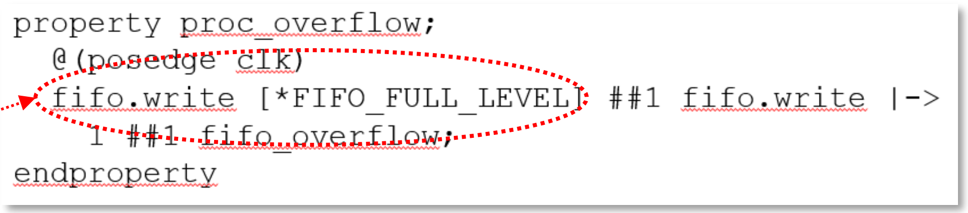
Lessons Learned

- Assuming checker execution state
  - i.e. Overflow on FIFO full && write



```
task proc_test();
  repeat (FIFO_FULL_LEVEL) begin
    @(posedge clk);
    fifo.write = 1;
  end

  @(posedge clk);
  fifo.write = 1;

  @(posedge clk);
  assert (fifo_overflow);
endtask
```

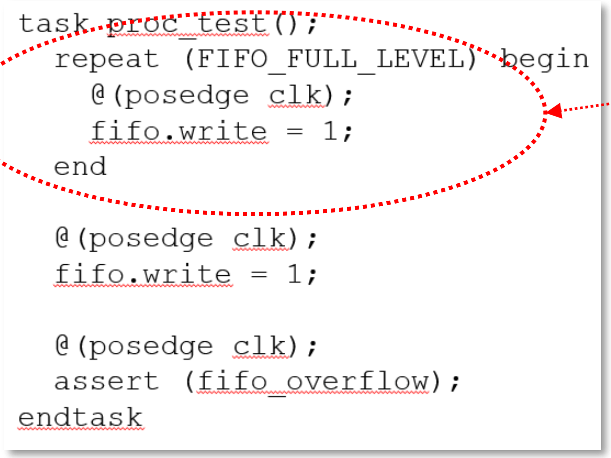
Reaching execution state procedurally



```
property proc_overflow;
  @(posedge clk)
  fifo.write [*FIFO_FULL_LEVEL] ##1 fifo.write |->
    1 ##1 fifo_overflow;
endproperty
```

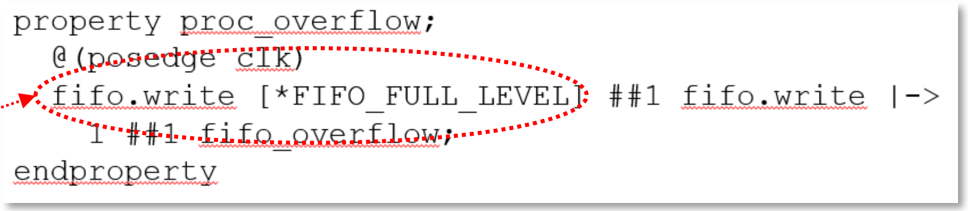
Lessons Learned

- Assuming checker execution state
  - i.e. Overflow on FIFO full && write
- State variables > procedures
  - Let the tool figure out how to get there



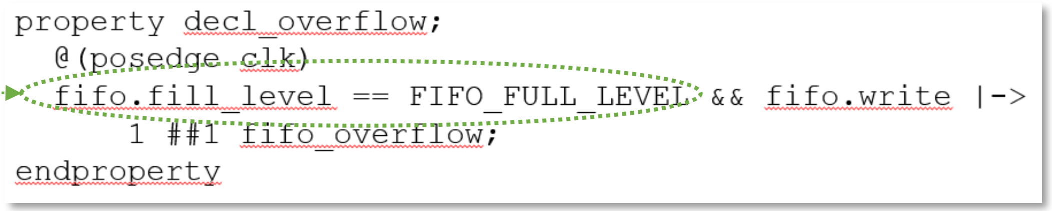
```
task proc_test();
  repeat (FIFO_FULL_LEVEL) begin
    @(posedge clk);
    fifo.write = 1;
  end

  @(posedge clk);
  fifo.write = 1;

  @(posedge clk);
  assert (fifo_overflow);
endtask
```

Reaching execution state procedurally



```
property proc_overflow;
  @(posedge clk)
  fifo.write [*FIFO_FULL_LEVEL] ##1 fifo.write |->
    1 ##1 fifo_overflow;
endproperty
```

Assuming execution state



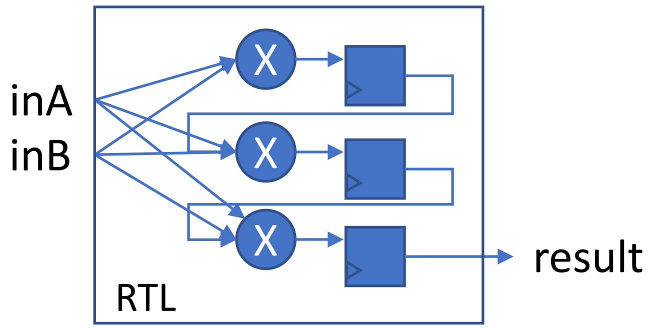
```
property decl_overflow;
  @(posedge clk)
  fifo.fill_level == FIFO_FULL_LEVEL && fifo.write |->
    1 ##1 fifo_overflow;
endproperty
```

Lessons Learned

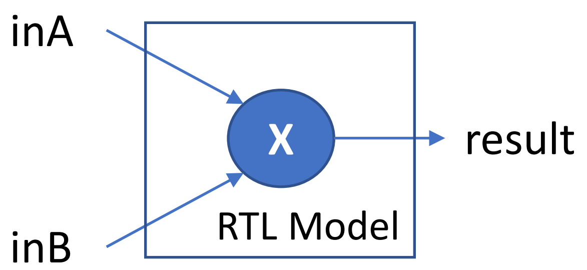
- RTL models for performance
  - i.e: multiply vs. pipelined multiply
    - Prove the RTL pipelined multiply in isolation
    - Use an RTL model everywhere else
  - Turned unusably slow into very fast

General recommendation…

- Deep pipelines
- Arithmetic functions
- Fast configurations



Slow



Fast

Summary

- ~~Because~~ why?
  - Simulation ruts run deep
  - Formal is undervalued
- Opportunities for collaborative sim + formal approaches
  - Software status/interrupts are a practical starting point
  - Anywhere low-level checking is feasible

What else is in the paper?

- Simulation vs. Formal in verification thought leadership
- Verilog configurations vs. binding for inserting models
- 4-phase checker template

Bonus Points!

- Merging code coverage with sim?