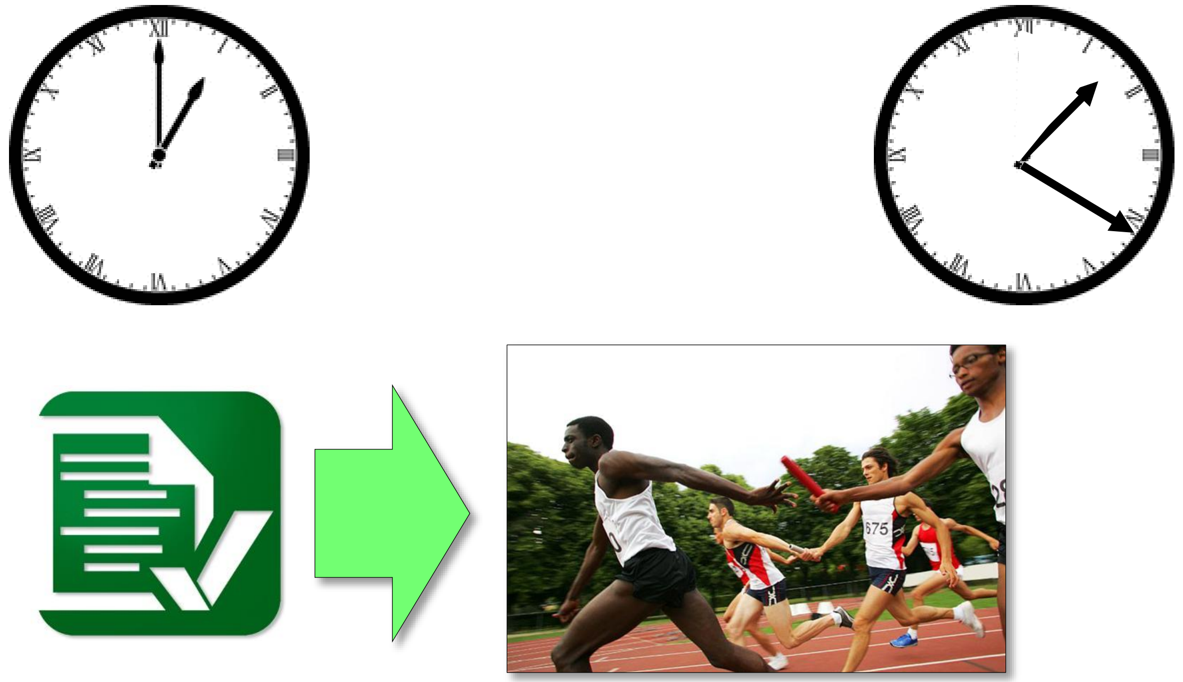
It’s Been 24 Hours – Should I Kill My Formal Run?

Mark Eslinger, Formal Verification Product Engineer
Jin Hou, Formal Verification Product Engineer
Joe Hupcey III, Verification Product Technologist
Jeremy Levitt, Formal R&D Principal Engineer

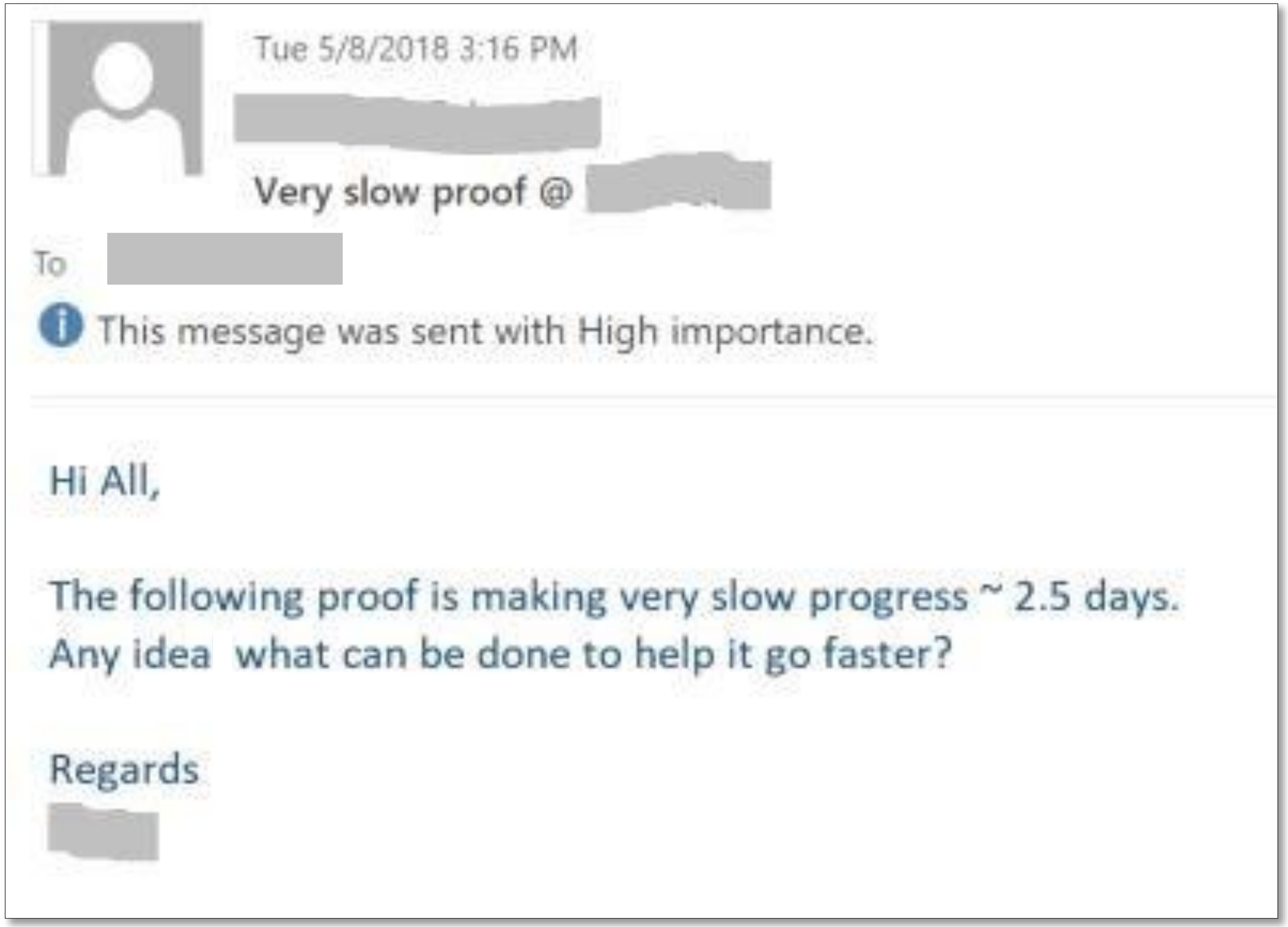
98.7% of Time Formal Runs Fast



But Sometimes …

Yesterday **Today**

Occasionally We Get Messages Like This



Tue 5/8/2018 3:16 PM

[redacted]

Very slow proof @ [redacted]

To [redacted]

This message was sent with High importance.

Hi All,

The following proof is making very slow progress ~ 2.5 days.
Any idea what can be done to help it go faster?

Regards
[redacted]

What Now?

Keep Running

???

Pros:

✓ Prior jobs also ran long

✓ Resources aren’t THAT expensive

Cons:

× Waste of compute resources

× Manual effort to monitor run

Kill & Start Over

Pros:

✓ Focus on most promising strategy

✓ Efficient use of compute resources

Cons:

× Waste of your valuable time

× Schedule impact

What You Will Learn Today

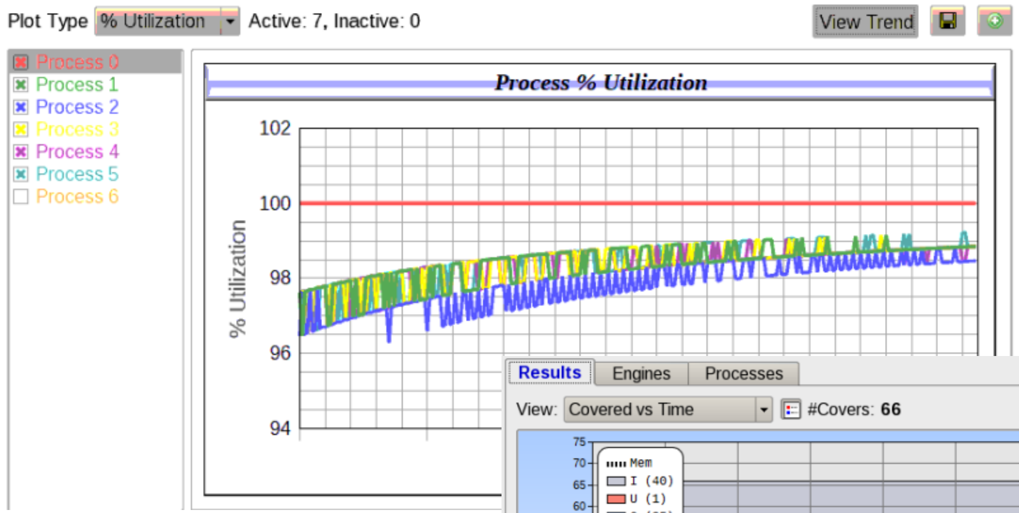
- What can you do right now
- What you can do before you run a new job

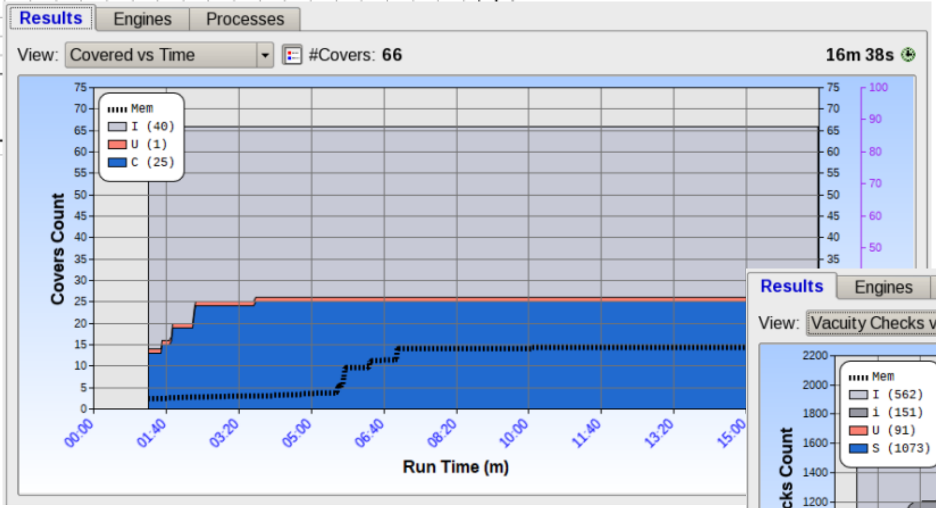
What CAN You Do Right NOW?

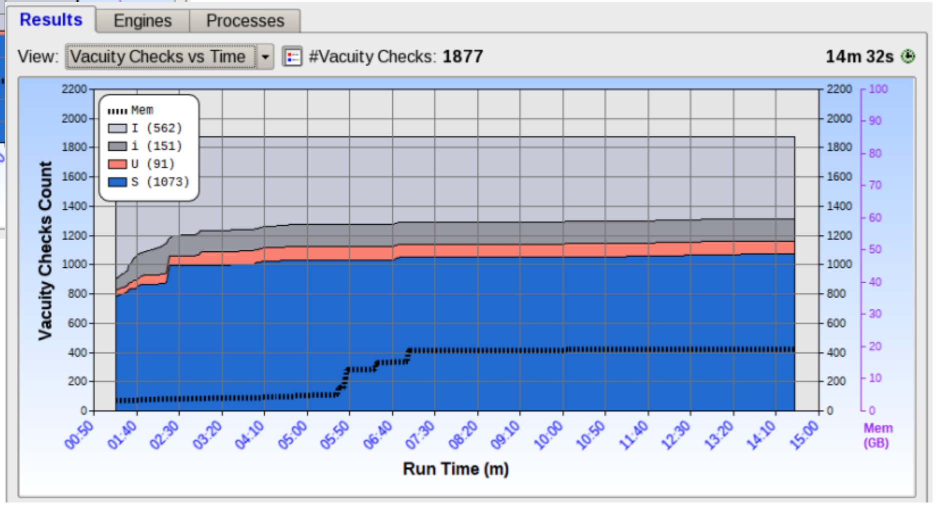
(0) Sanity Check Setup

- Does setup look correct?
  - Vacuously proved assertions?
  - Uncoverable cover properties?
  - Bogus firings?

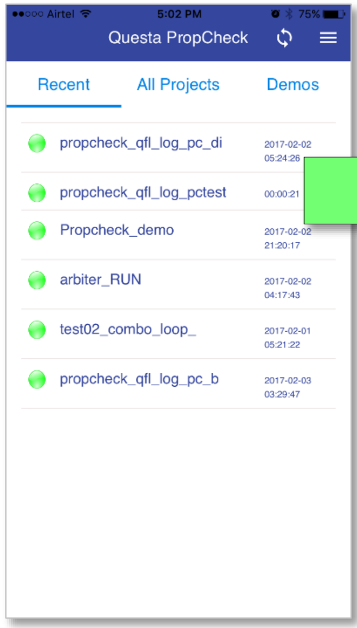
- Is hardware working efficiently?
  - Are CPUs all fully utilized?
  - Is memory consumption in line with that available?


Plot Type % Utilization Active: 7, Inactive: 0


View: Covered vs Time #Covers: 66 16m 38s


View: Vacuity Checks vs Time #Vacuity Checks: 1877 14m 32s

Check Progress Via Mobile App

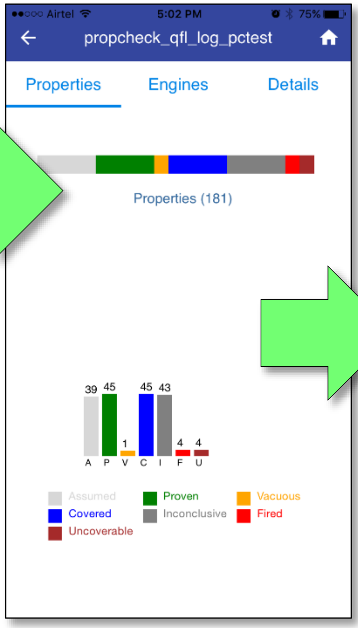


●●○○○ Airtel 5:02 PM 75%

Questa PropCheck

Recent | All Projects | Demos

propcheck_qfl_log_pc_di	2017-02-02 05:24:26
propcheck_qfl_log_pctest	00:00:21
Propcheck_demo	2017-02-02 21:20:17
arbiter_RUN	2017-02-02 04:17:43
test02_combo_loop_	2017-02-01 05:21:22
propcheck_qfl_log_pc_b	2017-02-03 03:29:47



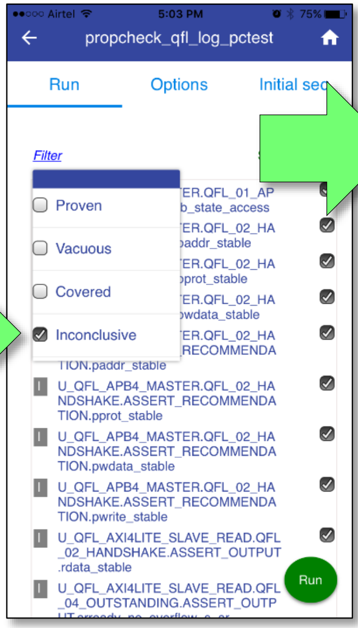
●●○○○ Airtel 5:02 PM 75%

propcheck_qfl_log_pctest

Properties | Engines | Details

Properties (181)

Assumed	Proven	Vacuous
Covered	Inconclusive	Fired
Uncoverable		



●●○○○ Airtel 5:03 PM 75%

propcheck_qfl_log_pctest

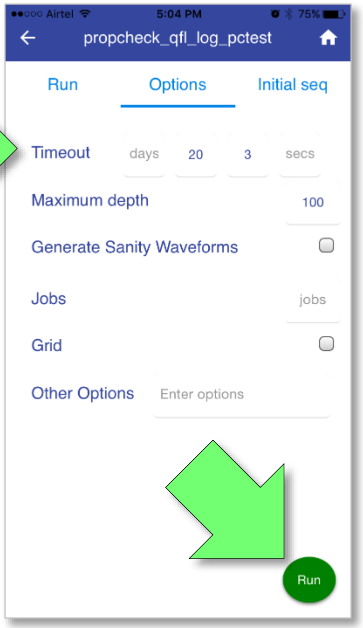
Run | Options | Initial seq

Filter

- [ ] Proven
- [ ] Vacuous
- [ ] Covered
- [x] Inconclusive

ER.QFL_01_AP b_state_access ✓
ER.QFL_02_HA paddr_stable ✓
ER.QFL_02_HA pprot_stable ✓
ER.QFL_02_HA pwdata_stable ✓
ER.QFL_02_HA RECOMMENDA TION.paddr_stable ✓
I U_QFL_APB4_MASTER.QFL_02_HANDSHAKE.ASSERT_RECOMMENDATION.pprot_stable ✓
I U_QFL_APB4_MASTER.QFL_02_HANDSHAKE.ASSERT_RECOMMENDATION.pwdata_stable ✓
I U_QFL_APB4_MASTER.QFL_02_HANDSHAKE.ASSERT_RECOMMENDATION.pwrite_stable ✓
I U_QFL_AXI4LITE_SLAVE_READ.QFL_02_HANDSHAKE.ASSERT_OUTPUT.rdata_stable ✓
I U_QFL_AXI4LITE_SLAVE_READ.QFL_04_OUTSTANDING.ASSERT_OUTP

Run



●●○○○ Airtel 5:04 PM 75%

propcheck_qfl_log_pctest

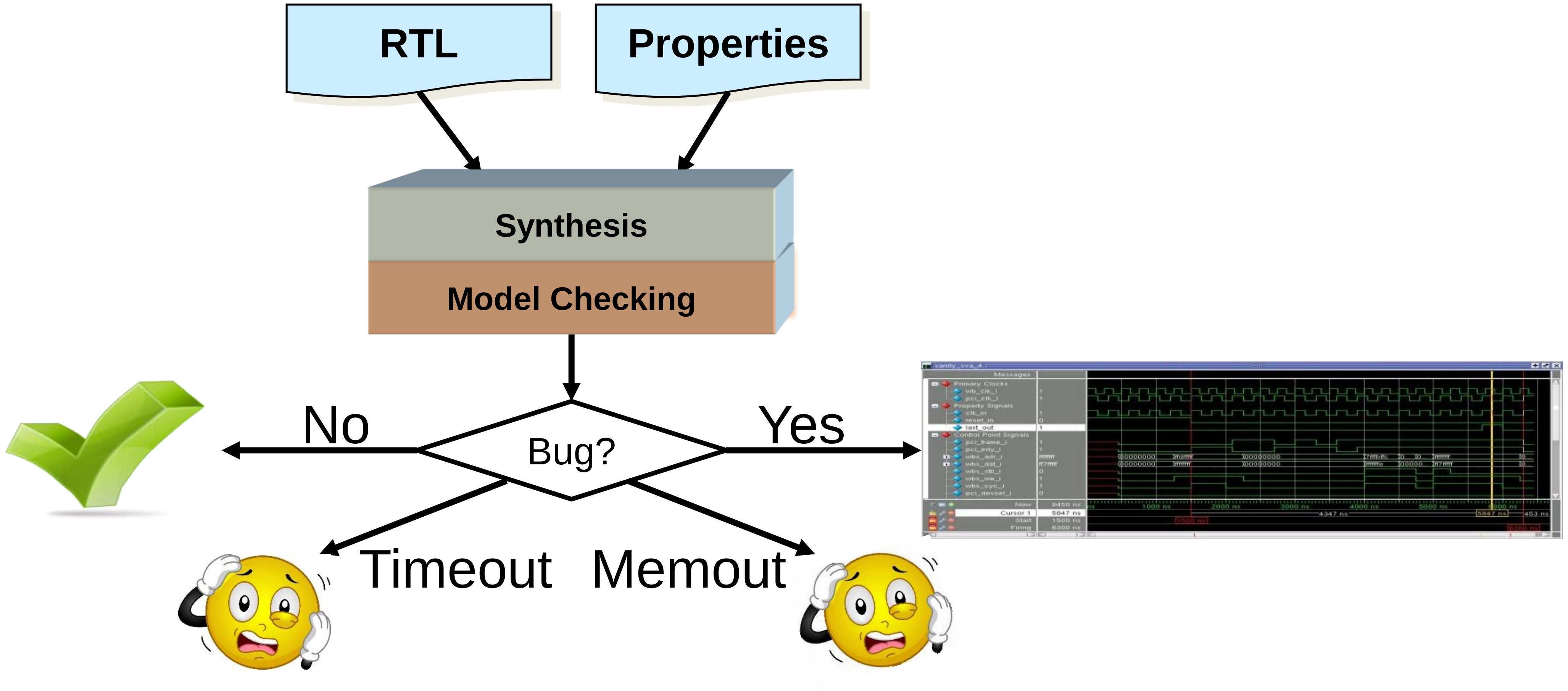
Run | Options | Initial seq

Timeout	days 20 3 secs
Maximum depth	100
Generate Sanity Waveforms	☐
Jobs	jobs
Grid	☐
Other Options	Enter options

Run

- Monitor and re-run formal jobs in real time while on-the-go
- Auto-reconnects with jobs in progress / results when re-gain signal
- Secure channel via mobile VPN
- Android and IOS phones and tablets supported

Formal is Awesome, Until It Isn’t



Why IS the Tool Still Running?

Obviously there are some “hard” properties:

- Temporal latency
- Formal unfriendly logic
- Lots of design states
- Ineffective heuristics → Bad luck?

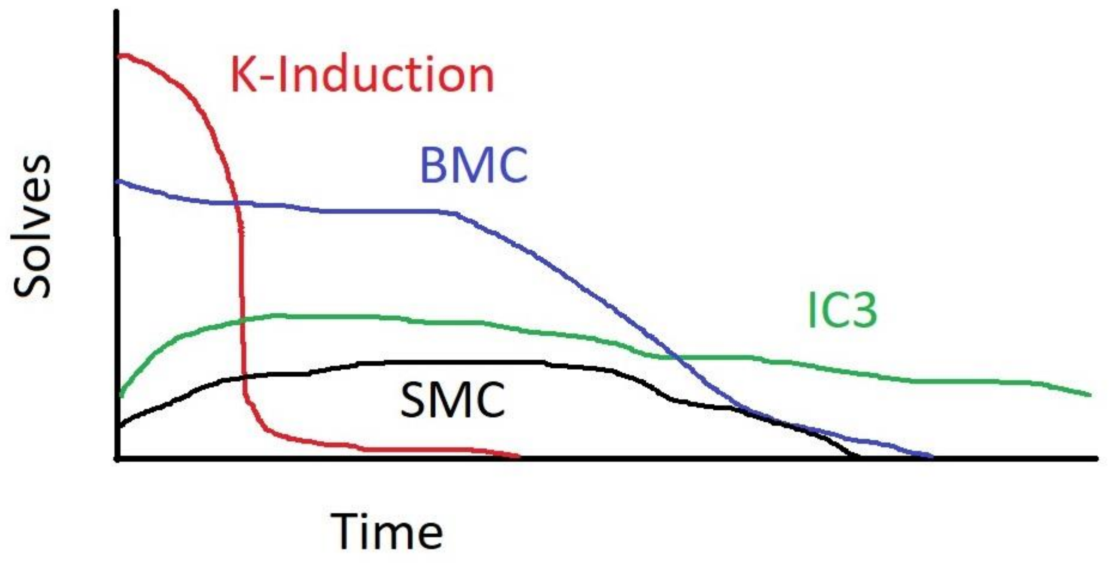
What Is The Tool Doing?

- Many different model checking algorithms exist, taking different approaches, e.g.
  - K-induction
  - SAT-based BMC
  - IC3
  - BDD-based SMC

- For a given design & property, one algorithm (or “engine”) is often much more effective than the others
  - Cannot tell which engine will solve first, until the solve happens
- Hence, tool runs each engine on each property
  - Either in parallel or iteratively until solution found

Each Engine Has A Different Profile

- K-induction: If proof not found early on, unlikely to find a proof
- BMC: If exponential slowdown w/increasing depth, unlikely to find CEX
- IC3: If rate of search space exploration slows, unlikely to find proof
- SMC: If state bits in model get too large, unlikely to find a proof



What You Can Do Now

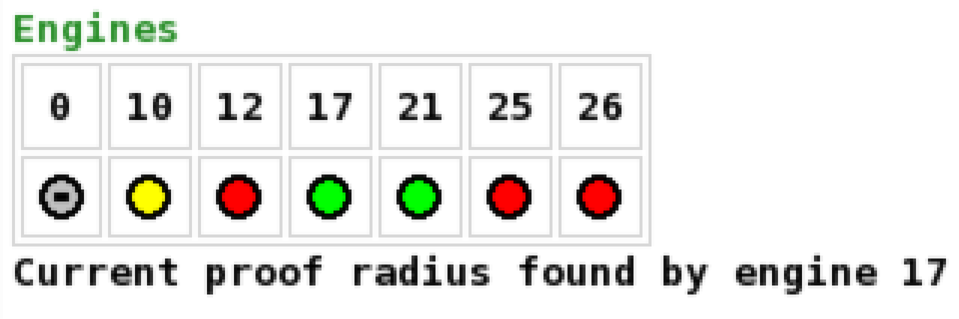
1. Monitor the formal engines' "health" in real time

2. Understand why a property is stuck

3. Keep running!

(1) Per Property Engine Health

- Engine developers can guess at likelihood of completion
  - Examine runtime parameters specific to each particular engine
    - Out-of-range parameters indicates rate of state-space exploration is poor
- R&D expertise codified & reported as “engine health”
  - Green/Yellow/Red, where Red indicates much less likely to complete
    - Red: If state-space exploration rate remains poor, engine won’t finish this month/year
    - While Engine Health can improve with time, this is not typical



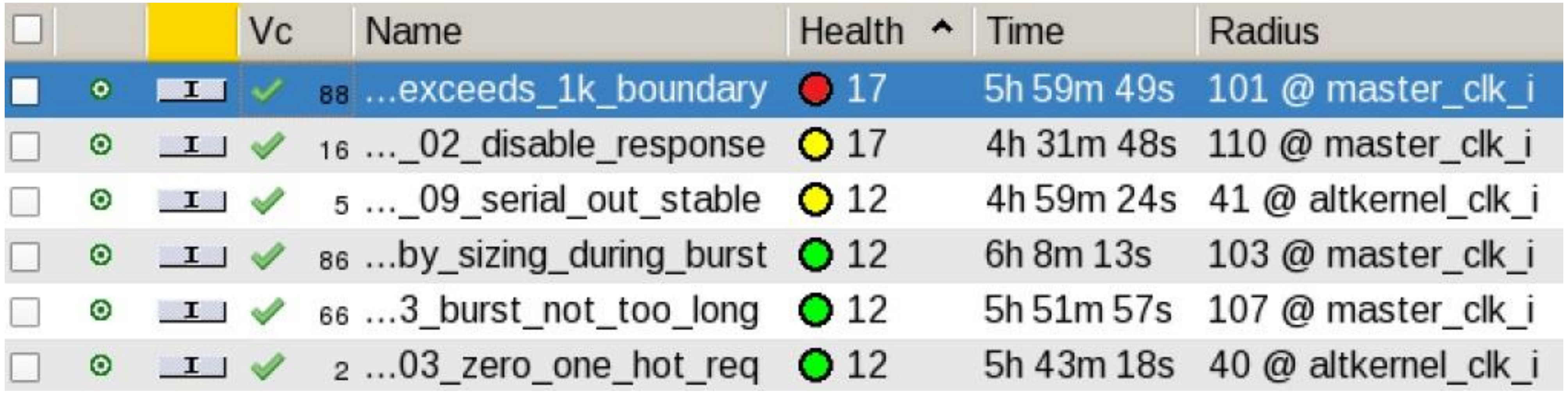
Engines

0	10	12	17	21	25	26
⊖	🟡	🔴	🟢	🟢	🔴	🔴

Current proof radius found by engine 17

(1) Monitoring Engine Health

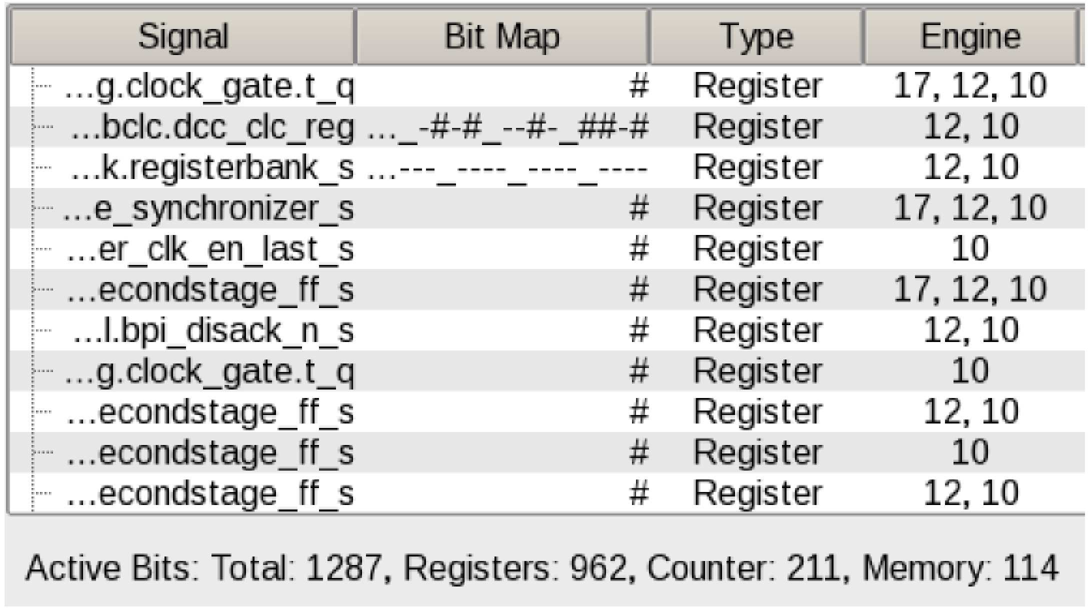
- For each property, health of healthiest engine is summarized
- Start looking at the “Red” properties first



			Vc	Name	Health	Time	Radius
			✓	88 ...exceeds_1k_boundary	🔴 17	5h 59m 49s	101 @ master_clk_i
			✓	16 ..._02_disable_response	🟡 17	4h 31m 48s	110 @ master_clk_i
			✓	5 ..._09_serial_out_stable	🟡 12	4h 59m 24s	41 @ altkernel_clk_i
			✓	86 ...by_sizing_during_burst	🟢 12	6h 8m 13s	103 @ master_clk_i
			✓	66 ...3_burst_not_too_long	🟢 12	5h 51m 57s	107 @ master_clk_i
			✓	2 ...03_zero_one_hot_req	🟢 12	5h 43m 18s	40 @ altkernel_clk_i

(2) Understand Why A Property is Stuck

- Are the engines stuck analyzing logic known to be difficult?
  - Temporal latency too great => counters?
  - Related logic is formal unfriendly => large multipliers, LSFR, ECC, etc.?
  - Too much design state involved => memories?
- Examine logic being analyzed by the engines



Signal	Bit Map	Type	Engine
…g.clock_gate.t_q	#	Register	17, 12, 10
…bclc.dcc_clc_reg	…_-#-#_--#-_##-#	Register	12, 10
…k.registerbank_s	…---_----_----_----	Register	12, 10
…e_synchronizer_s	#	Register	17, 12, 10
…er_clk_en_last_s	#	Register	10
…econdstage_ff_s	#	Register	17, 12, 10
…l.bpi_disack_n_s	#	Register	12, 10
…g.clock_gate.t_q	#	Register	10
…econdstage_ff_s	#	Register	12, 10
…econdstage_ff_s	#	Register	10
…econdstage_ff_s	#	Register	12, 10

Active Bits: Total: 1287, Registers: 962, Counter: 211, Memory: 114

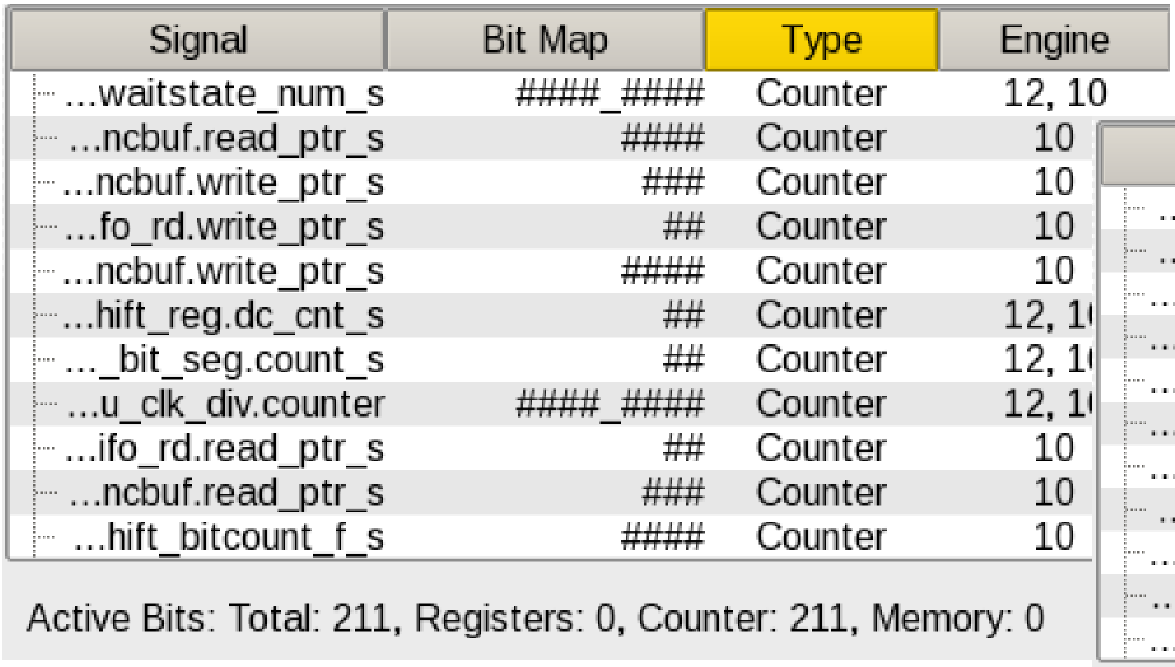
(2) Even Irrelevant Logic Can Hurt

```
assign C = counter < 32’h3fffffff || f(x,y,z);
assert property (A & B |-> C);
```

- Focusing on the counter logic quickly yields deep proof bounds
  - Formal tool may decide to expand the counter logic
- Actual proof depends on *f(x,y,z)* holding when *A* asserted
- Engines get stuck exploring counter logic to an impossible depth
  - Appears to be making progress, but strategy will not lead to a proof
  - A lot of time will be wasted

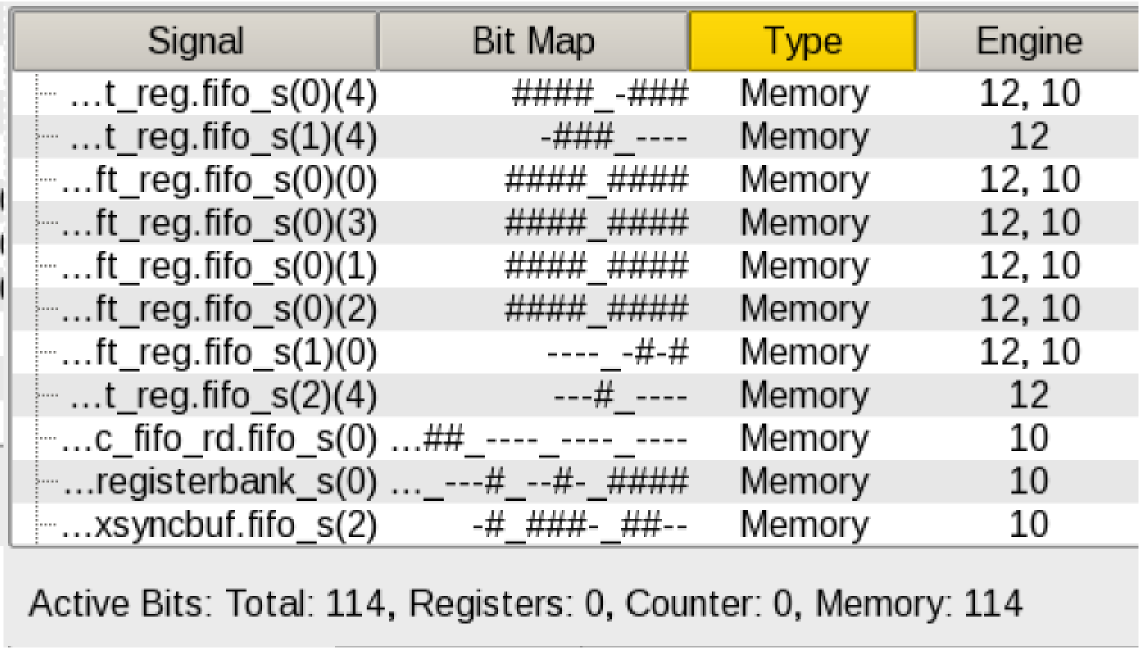
(2) Look for Problem Logic!

- Is there performance crippling logic?
  - E.g. large counters & RAMs, wide multipliers, etc.
  - State elements are ordered from most to least active



Signal	Bit Map	Type	Engine
...waitstate_num_s	####_####	Counter	12, 10
...ncbuf.read_ptr_s	####	Counter	10
...ncbuf.write_ptr_s	###	Counter	10
...fo_rd.write_ptr_s	##	Counter	10
...ncbuf.write_ptr_s	####	Counter	10
...hift_reg.dc_cnt_s	##	Counter	12, 1
..._bit_seg.count_s	##	Counter	12, 1
...u_clk_div.counter	####_####	Counter	12, 1
...ifo_rd.read_ptr_s	##	Counter	10
...ncbuf.read_ptr_s	###	Counter	10
...hift_bitcount_f_s	####	Counter	10

Active Bits: Total: 211, Registers: 0, Counter: 211, Memory: 0

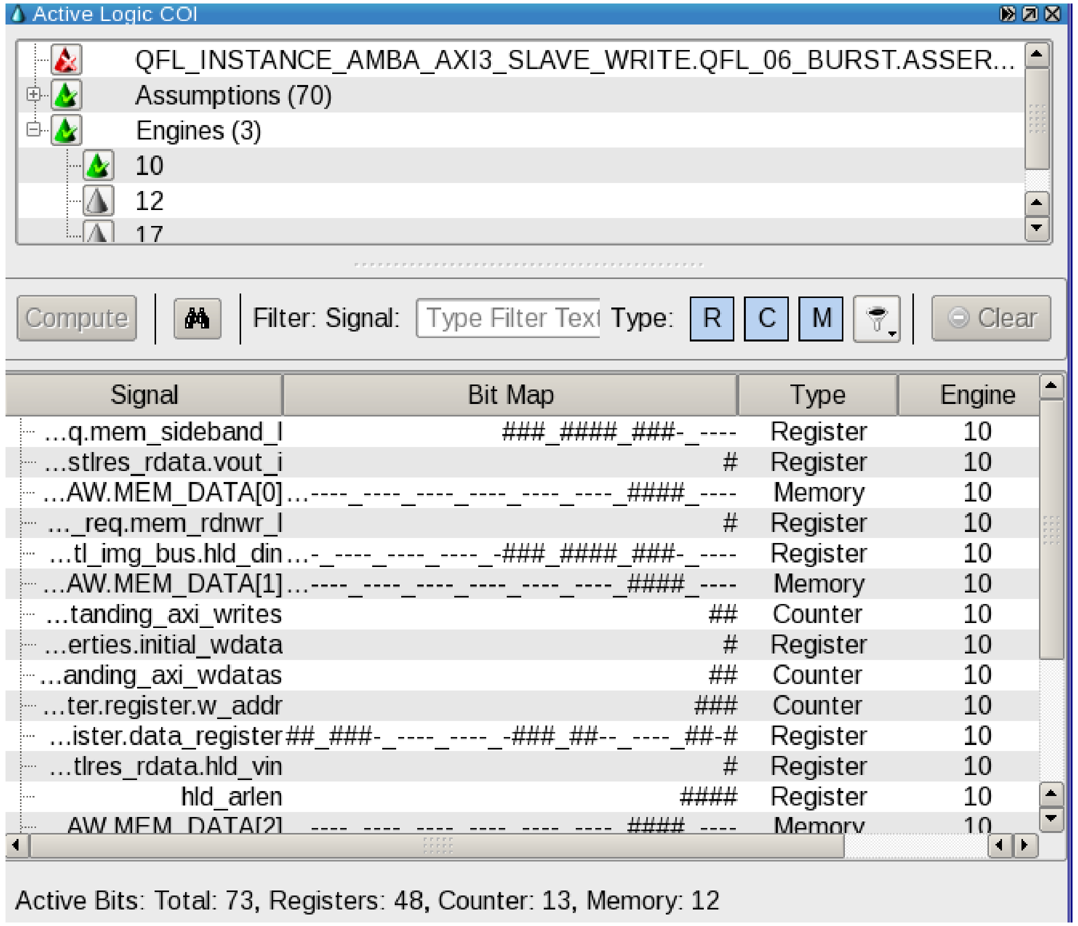


Signal	Bit Map	Type	Engine
...t_reg.fifo_s(0)(4)	####_-###	Memory	12, 10
...t_reg.fifo_s(1)(4)	-###_----	Memory	12
...ft_reg.fifo_s(0)(0)	####_####	Memory	12, 10
...ft_reg.fifo_s(0)(3)	####_####	Memory	12, 10
...ft_reg.fifo_s(0)(1)	####_####	Memory	12, 10
...ft_reg.fifo_s(0)(2)	####_####	Memory	12, 10
...ft_reg.fifo_s(1)(0)	----_-#-#	Memory	12, 10
...t_reg.fifo_s(2)(4)	---#_----	Memory	12
...c_fifo_rd.fifo_s(0)	...##_----_----_----	Memory	10
...registerbank_s(0)	..._---#_--#-_####	Memory	10
...xsyncbuf.fifo_s(2)	-#_###-_##--	Memory	10

Active Bits: Total: 114, Registers: 0, Counter: 0, Memory: 114

(2) Exploring Logic Pulled-in by "Assumes"

- *exclude* contributes from the assert
- *ignore* contributions from other engines
- Logic *being* analyzed by engine 10 that is *only* in the fan-in of assumes
- Check state bits: "less is more"



Signal	Bit Map	Type	Engine
...q.mem_sideband_l	###_####_###-_----	Register	10
...stlres_rdata.vout_i	#	Register	10
...AW.MEM_DATA[0]	...----_----_----_----_----_----_####_----	Memory	10
..._req.mem_rdnwr_l	#	Register	10
...tl_img_bus.hld_din	...-_----_----_----_-###_####_###-_----	Register	10
...AW.MEM_DATA[1]	...----_----_----_----_----_----_####_----	Memory	10
...tanding_axi_writes	##	Counter	10
...erties.initial_wdata	#	Register	10
...anding_axi_wdatas	##	Counter	10
...ter.register.w_addr	###	Counter	10
...ister.data_register	##_###-_----_----_-###_##--_----_##-#	Register	10
...tlres_rdata.hld_vin	#	Register	10
hld_arlen	####	Register	10
AW.MEM_DATA[2]	----_----_----_----_----_----_####_----	Memory	10

Active Bits: Total: 73, Registers: 48, Counter: 13, Memory: 12

(2) Triaging problem logic

- If engine performance is poor & suspicious logic present
  - Easy case: If logic not relevant to proof
    - If logic brought in via an unneeded assume, turn off assume
    - Otherwise, use cutpoint to remove it
  - Hard case: If logic relevant to proof, simplify problem
    - E.g. reduce parameter values, set constants, abstract the logic

- Either way, *current run is unlikely to complete* – **a re-run is needed**

(3) Keep Running!

- With enough time and memory, algorithms will find the answer

- (Do you have enough time and memory?)
  - *Caveat: Not possible to know in advance how much of either is required*

What You Can Do Before You Run a New Job

Setting Up For Success

1. Use “re-modeling”, “abstraction”, and black boxing
2. Limit your “assumptions” (a/k/a constraints)
3. Let the machines do the work
4. Sanity check your setup early on
5. Write properties more effectively
6. Leverage the “assume guarantee” principal
7. Simplify your formal testbench

Remodeling: "Modify" the DUT Without Touching The RTL

- Use tool commands to modify DUT

```
netlist cutpoint signal
netlist property –assume {<assertion constraining signal>}
```

- Use SV *bind* construct to non-invasively add modeling logic after cut of the design signal

```
netlist cutpoint signal –driver abs_signal
netlist property –assume {signal == abs_signal}
```

```
module abs_model (input clk, rstn, input [WIDTH-1:0] signal);
logic [WIDTH-1:0] abs_signal;
      <modelling code of abs_signal>
endmodule
bind dut abs_model …;
```

- Reduce design size: Use compile switch to reduce parameter values

```
formal compile –d dut –G WIDTH=8 –G DEPTH=16
```

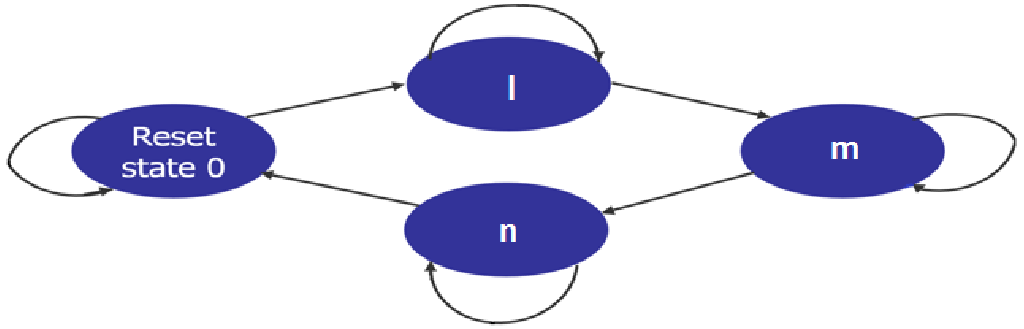
- Key applications: counter and memory abstraction

Counter Abstraction

- Reduce the sizes of counters or the values of counters to be used
- Set counters to an “X” value for its initial state
  - Let formal consider all potential values for counter initial state

```
netlist initial counter_signal –value x
```

- Replace counters with small state machines
  - Only critical values of counters that trigger actions are important
  - Example: Suppose value ‘l’, ‘m’ and ‘n’ of the counter are critical. Use the following state machine to replace the original counter.



https://blogs.mentor.com/verificationhorizons/blog/2018/09/28/how-to-reduce-the-complexity-of-formal-analysis-part-4-counter-abstraction/

Replacing A Counter with A State Machine

```
netlist cutpoint cnt
```

```
module abs_model  #(parameter WIDTH=16) (input clk, rst, input [WIDTH-1:0] cnt);
   reg [WIDTH-1:0] abs_cnt;
   parameter l='h60, m='hf0, n='hf1;
   always @(posedge clk or posedge rst)
     if (rst) abs_cnt <= 'h00;
     else begin
       if (abs_cnt == 'h00)         abs_cnt <= l;
       else if (abs_cnt == l)       abs_cnt <= m;
       else if (abs_cnt == m)       abs_cnt <= n;
       else                         abs_cnt <= 'h00;
     end
   assume_cnt:  assume property (@(posedge clk) cnt == abs_cnt);
endmodule // abs_model
bind test abs_model #(.WIDTH(8)) u_abs_model (.clk(clk), .rst(rst), .cnt(cnt));
```

- For a property that can only be fired when the counter reaches the value ‘n’, using the abstract model of the counter, the counter can reach ‘n’ in 3 cycles after reset
- Formal can quickly fire the property and generate much shorter error trace

Memory Abstraction

- Blackbox memories
- Reduce the sizes of memories
  – Reduce parameters for data width and address depth
- Abstract the memory entries not inferred by the property to free variables
- Replace a ROM with a look-up table
- Replace a memory with a cache of N entries
  – Remember the last N writes and abstract the rest as free variables

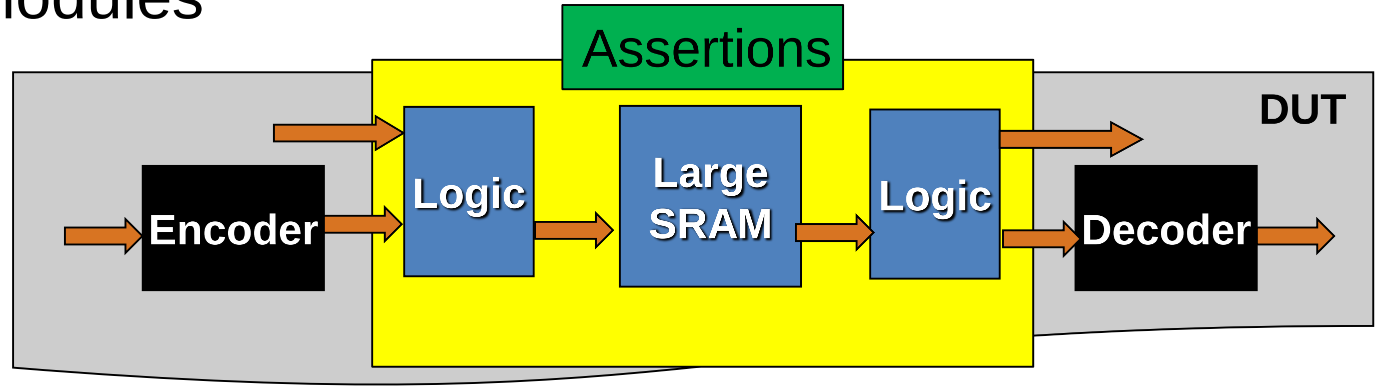
https://blogs.mentor.com/verificationhorizons/blog/2018/10/23/how-to-reduce-the-complexity-of-formal-analysis-part-5-memory-abstraction/

Black Boxing

- Blackboxing can improve compile and verify time

```
netlist blackbox <module_name>
netlist blackbox instance <instance_name>
```

  - All outputs of the blackboxed module or instance become free variables
  - Proofs are valid and firing need further investigation
  - Example: Verify SRAM and related logic by blackboxing Encoder and Decoder modules



Replacing ROM With A Look-up Table

- Reduce the number of state bits: Width x Depth -> Width

```
always @(addr)
case (addr)
`include "./zin_files/ext_lut_0008.dat"
`include "./zin_files/ext_lut_0010.dat"
`include "./zin_files/ext_lut_0018.dat"
`include "./zin_files/ext_lut_0020.dat"
default:    sram_data <= 32'h00000000;
endcase

always @(posedge HCLK)
if (!HRESETn)
        HRDATAM <= 32'h00000000;
else
        HRDATAM <= !HWRITEM ? sram_data : 32'h00000000;
```

```
24'h200000: sram_data <= 32'h284c_2f73 ;
24'h200002: sram_data <= 32'he55a_25fc ;
24'h200004: sram_data <= 32'hd75d_ba1c ;
24'h200006: sram_data <= 32'h64a0_ad14 ;
24'h200008: sram_data <= 32'h33e3_31c1 ;
24'h20000A: sram_data <= 32'hd5c6_435e ;
….
24'h202682: sram_data <= 32'h2a8c_a5aa ;
24'h202684: sram_data <= 32'h75f5_b99f ;
24'h202696: sram_data <= 32'hf0eb_f161 ;
24'h202698: sram_data <= 32'h7b58_0d0a ;
```

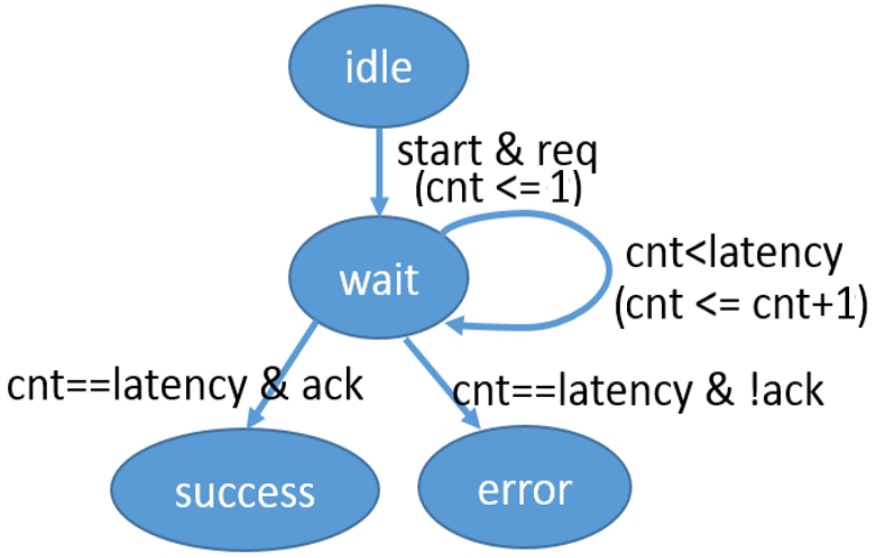
Pro Techniques:
Data Independence and Non-Determinism

- **Data Independence (DI):** your property/assertion does NOT depend on specific values of the data
  - Example: Verifying the data integrity of a fifo is data independent.

- **Non-Determinism (ND):** use "free variables" implemented as un-driven wires or extra inputs in a checker to tell the formal engines they are free to consider any cases involving all possible values of the variables at once
  - Example: req and ack can be overlapped.

```
Check_ack: assert property (@(posedge clk)
           req |-> ##latency ack);
```

  - Rewriting this assertion using a counter "cnt" (log2 latency) and a free variable "start".



- Details on the Verification Horizons blog: https://blogs.mentor.com/verificationhorizons/blog/2018/11/01/how-to-reduce-the-complexity-of-formal-analysis-part-6-leveraging-data-independence-and-non-determinism/

Limit Your "Assumptions"

- In constrained-random simulation, adding more assumptions is generally a good thing
  - More constraints can help the constraint solver converge faster
  - Irrelevant constraints typically don't have much performance impact

- However, in formal …
  - Initially all the logic touched by all your assertions is considered
  - Formal eventually figures out the relevant constraint logic, but a lot of clock cycles and memory are wasted

- Only use the assumptions necessary for the properties to be verified

"Less is More!"

Letting the Machines Do The Work

- Use the tool's multicore capabilities
  - More cores = better performance
  - The verify command switch –jobs <n>
  - GUI can define the number of cores or add more cores lively

- Submit jobs to grid system
  - Examples:

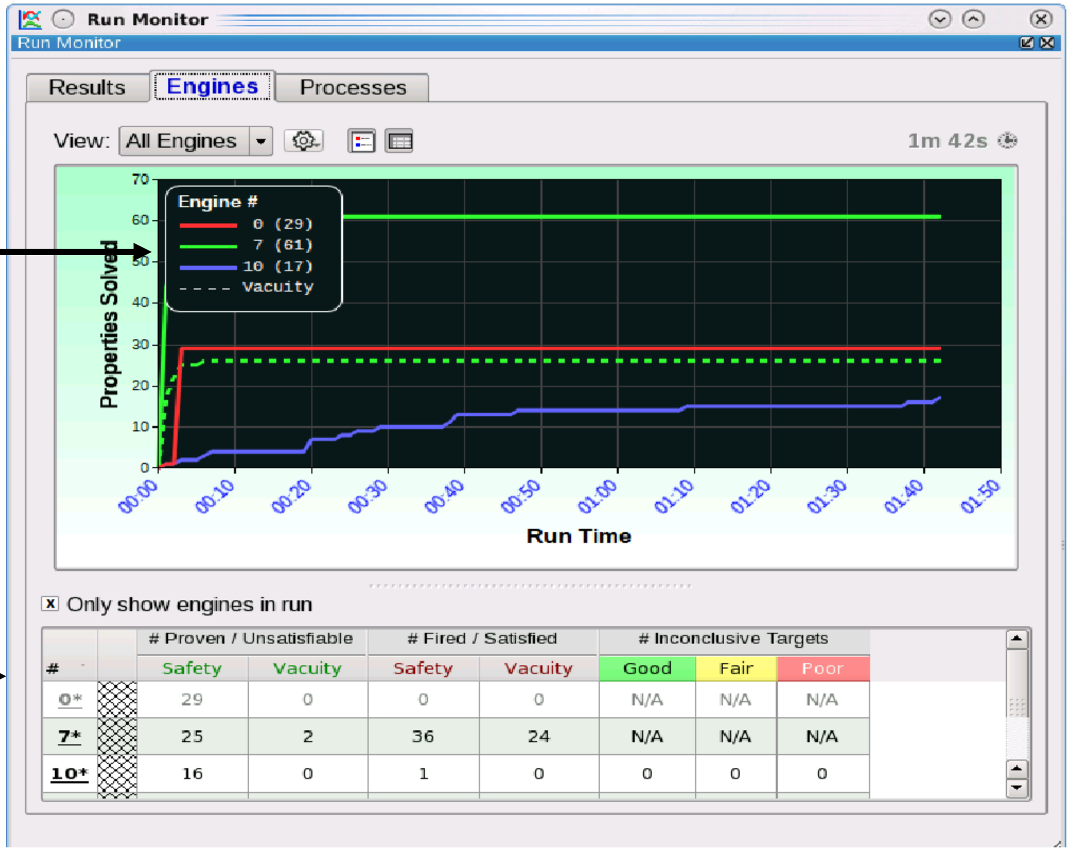
```
configure grid submit { qrsh -V -now n -q zin.q -l h_vmem=512M }
configure grid submit { qsub -l -j y -b y  -V -R n -w n -q mygid.p h_vmem=2G }
```

Use The Most Effective Engines

- Run Monitor tab in GUI shows the engine usage report.
  - Know which engines worked best in the previous run
  - Run with the most effective engines
    - The verify command switch -engine

Summary of engines' performance

<u>Detail of engines' performance:</u>
- Engine 0 proved 29 safety properties
- Engine 7 proved 25 safety properties and 2 vacuity checks, and fired 36 safety properties and 24 vacuity checks
- Engine 10 proved 16 and fired 1 safety properties



[x] Only show engines in run

#		# Proven / Unsatisfiable		# Fired / Satisfied		# Inconclusive Targets		
		Safety	Vacuity	Safety	Vacuity	Good	Fair	Poor
0*	▨	29	0	0	0	N/A	N/A	N/A
7*	▨	25	2	36	24	N/A	N/A	N/A
10*	▨	16	0	1	0	0	0	0

Before You Begin: Follow the Law!

Obey the Two Great Laws of Formal Friendly Properties!

1. Keep properties as **SIMPLE** as possible
2. Keep properties as **SEQUENTIALLY SHORT** as possible

But Why?

**This gives formal engines more latitude
to optimize the state space it must analyze**

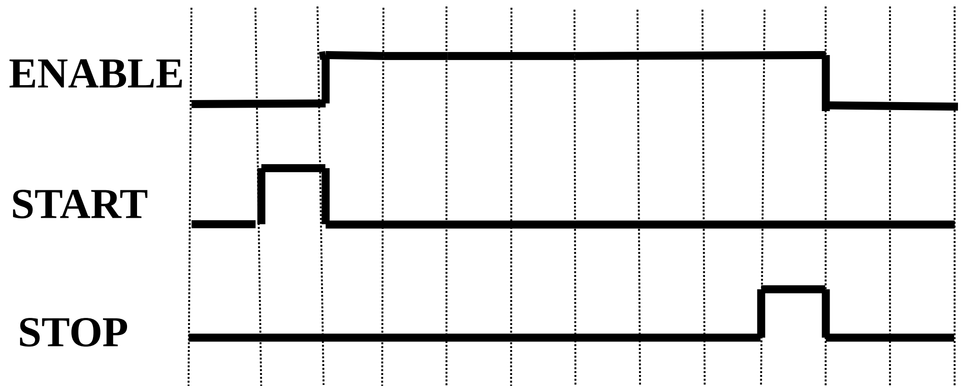
**Benefits: almost always yields better
wall clock run time, memory usage, and debug**

The Two Great Laws In Detail

1. Keep properties as **SIMPLE** as possible
   - The less state logic a property has, the better
   - Reference as little of the DUT as possible
   - Break complex properties into several simpler ones
   - Make use of modeling layer code to simplify the property

2. Keep properties as **SHORT** as possible
   - The shorter the sequential depth the better
   - Single-cycle assertions are best
   - Under 10 cycles is a rule of thumb
   - Function of design size and property depth determines results

1st Great Law: Simple Properties



```
$rose(START)  |=> (ENABLE && ~START && ~STOP)[*7] ##1 \
          (ENABLE && ~START && STOP) |=> (~ENABLE && ~START && ~STOP);

$rose(START)           |-> ~ENABLE ##1 ENABLE;
$rose(ENABLE)          |-> (~START && ~STOP)[*7];
$rose(STOP)            |-> ENABLE ##1 ~ENABLE;
$fell(START)           |=> ##5 $rose(STOP);
$rose(STOP)            |=> ~STOP;
```

If You Have Inconclusives the First 24hrs: “Decompose”

Original:

```
a_xyz: assert property (@(posedge clk) a && b |-> x && y && z );
```

Decomposed:

```
a_x: assert property (@(posedge clk) a && b |-> x );
a_y: assert property (@(posedge clk) a && b |-> y );
a_z: assert property (@(posedge clk) a && b |-> z );
```

Original:

```
a_tran12: assert property (@(posedge clk)
             condition_start |=> transaction1 or transaction2 );
```

Decomposed:

```
a_tran1: assert property (@(posedge clk)
             condition_start && type==TYPE1 |=> transaction1 );
a_tran2: assert property (@(posedge clk)
             condition_start && type==TYPE2 |=> transaction2 );
```

Leverage Modeling Code

- Verilog code which can help in writing an assertion
  - Simplify understanding the property or simplifying the property itself
- Example assertion file with modeling layer code:

```
module assert_top (input rstn, clk, A, B, C, wr, rd );
     // Requirement: Never > 5 outstanding wr's (without a rd)
     // Requirement: No rd before wr
           reg [2:0] my_cnt;
           always @(posedge clk or negedge rstn)
           if (!rstn)   my_cnt <= 3'b000;
           else
                 if       ( wr && !rd)   my_cnt <= my_cnt + 1;
                 else if (!wr &&  rd)   my_cnt <= my_cnt - 1;
                 else                    my_cnt <= my_cnt;

  a_wr_outstanding_le5: assert property (@(posedge clk)    my_cnt <= 3'd5 );
  a_no_rd_without_wr:   assert property (@(posedge clk)
                                                  !((my_cnt == 3'd0) && rd)    );
    endmodule
```

2nd Great Law: Sequentially Short Properties

```
a1: assert property (@(posedge clk) $onehot(state) );
```
1 cycle

```
a2: assert property (@(posedge clk) ~(A && B) );
```
1 cycle

```
a3: assert property (@(posedge clk) $rose(A) |=> ~A );
```
3 cycles

```
a4: assert property (@(posedge clk) disable iff (~rst_n)
        A ##1 B && C ##1 D |=> E );
```
4 cycles

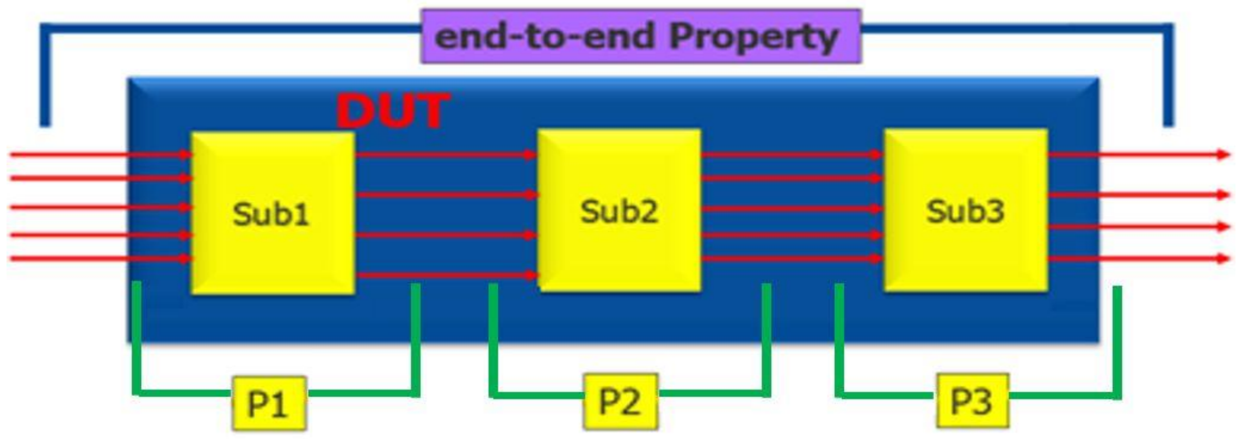
```
a5: assert property (@(posedge clk) disable iff (~rst_n)
        A ##1 B |=> C ##[1:5] D );
```
4-8 cycles

a6: May get CEX, No Proof

```
a6: assert property (@(posedge clk) disable iff (~rst_n)
     A ##1 B |=> C ##[1:100] D );
```
4-104 cycles

```
a7: assert property (@(posedge clk) disable iff (~rst_n)
     A ##1 B |=> ##1024 C );
```
1026 cycles

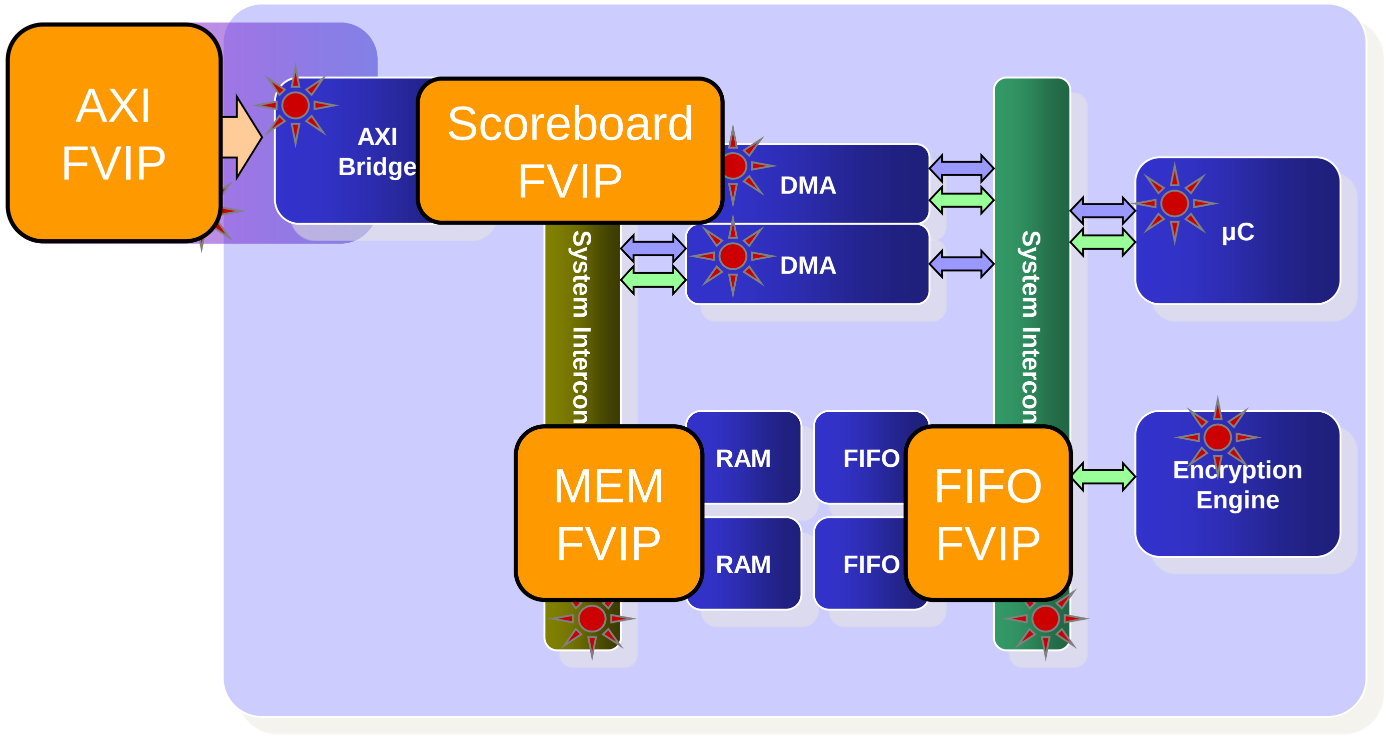
Leveraging “Assume Guarantee”



- Break apart “end-to-end Property” into “P1”, “P2”, and “P3”
- When P1 is proven for Sub1, use it as an assumption/constraint to run a proof of P2 on Sub2. Repeat …
- Results will be the same as if we ran on the big end-to-end property thanks to the “assume-guarantee” principle
- COIs for verifying the individual P1, P2, and P3 assertions are reduced dramatically → faster run time and memory performance!

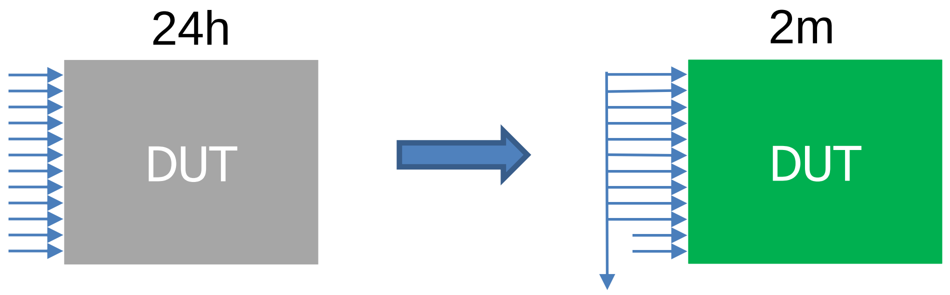
Leverage Formal VIP

- Formal verification IP is powerful and easy to use
  - Will already use many techniques to reduce state space

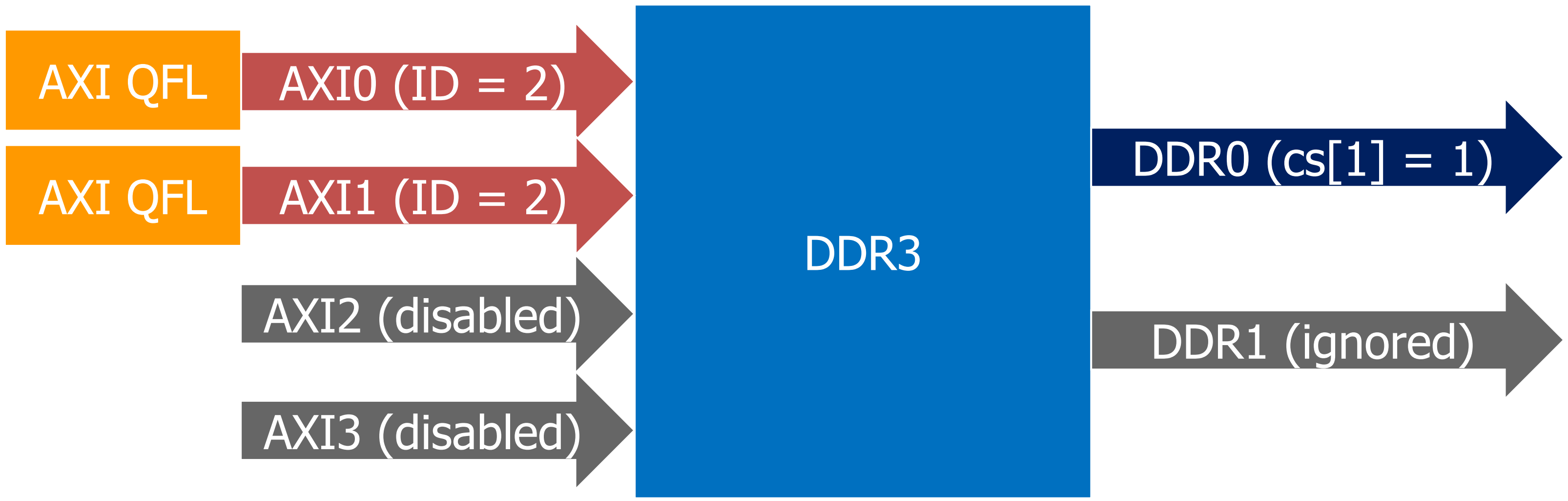


Over Constrain to Get Results

- Over constrain to turn inconclusive results into conclusive results
  - Useful bugs can be found, proofs generally not valid though provide info
- Constrain input state space

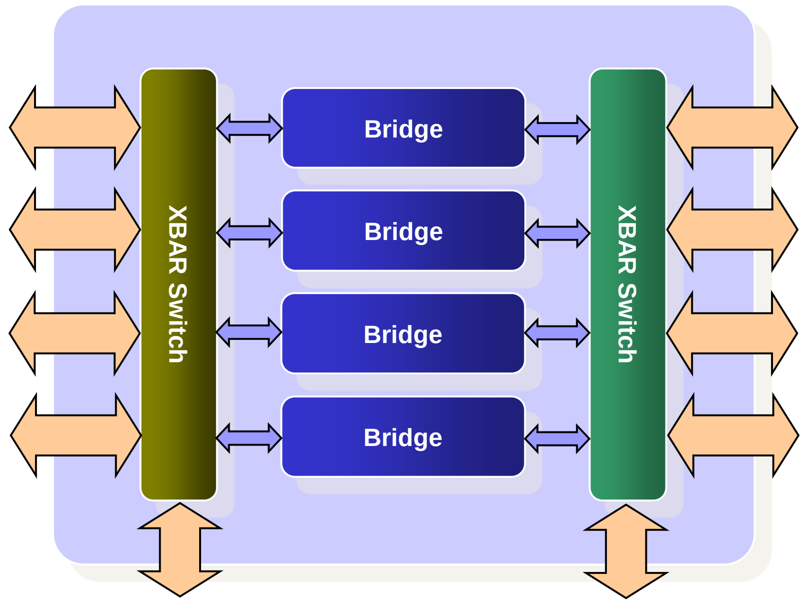


- Make use of Symmetry



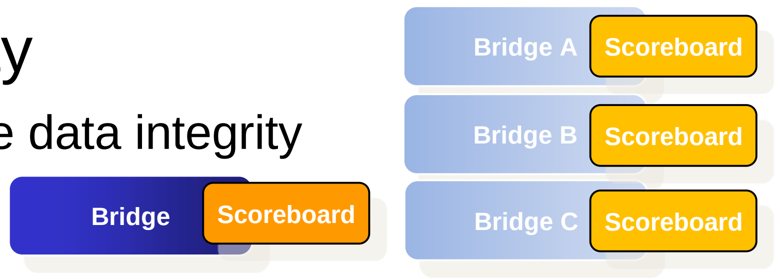
Simplify Formal Testbench

- Divide and Conquer approach is often used
  - Data integrity, functionality, connectivity

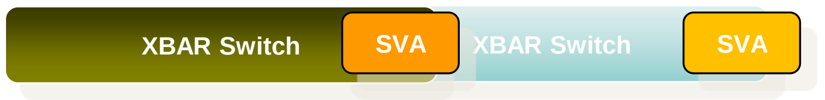


256 combinations
Selects stable during transmission

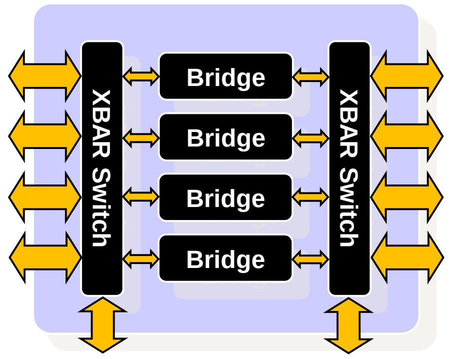
- Bridge data integrity



- XBAR functionality

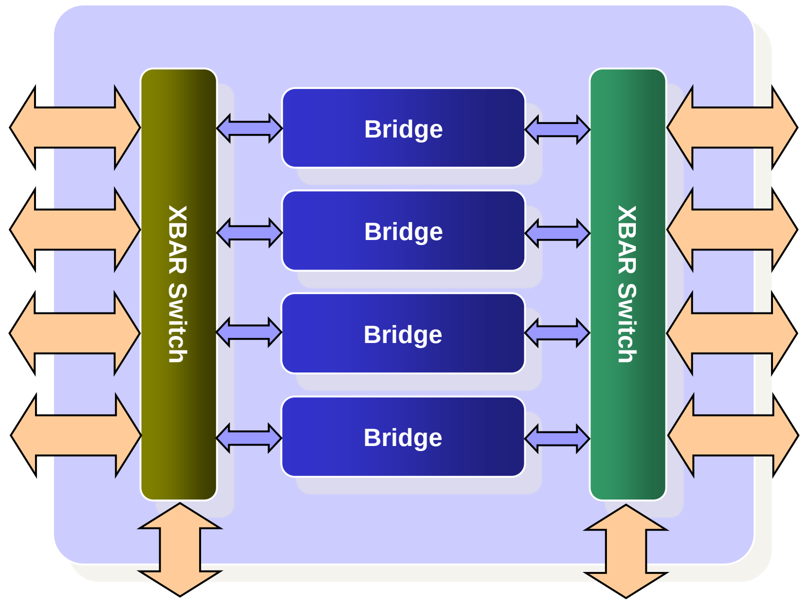


- Connectivity between blocks

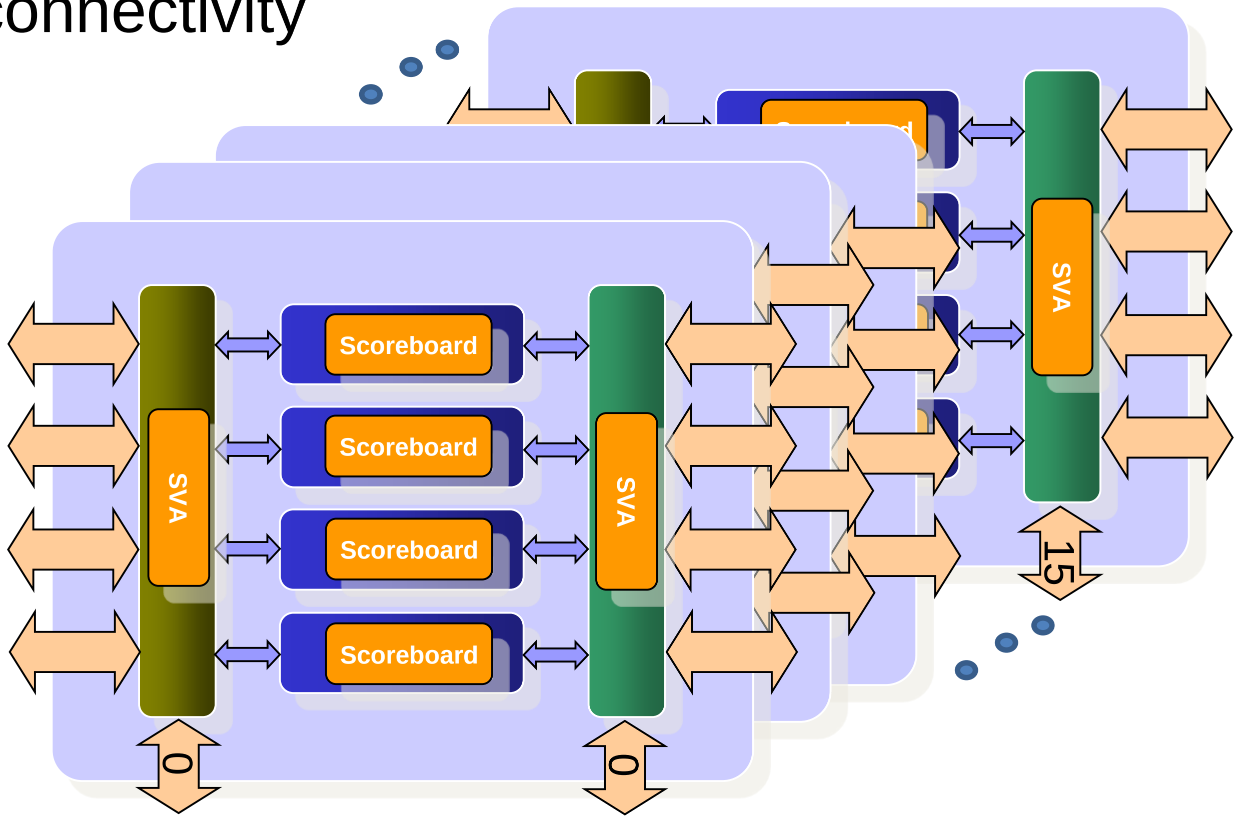


Simplify Formal Testbench

- Brute Force can be used to verify everything at the same time
  - Data integrity, functionality, connectivity

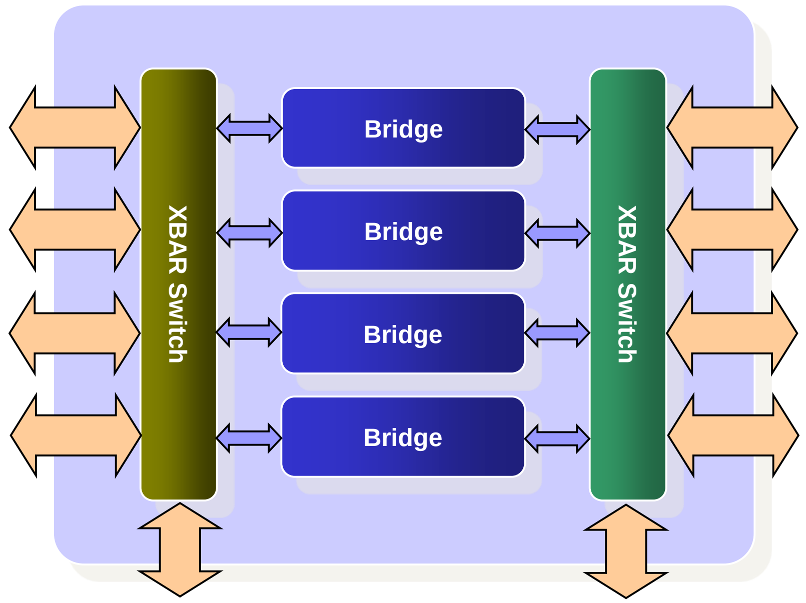


256 combinations
Selects stable during transmission



Simplify Formal Testbench

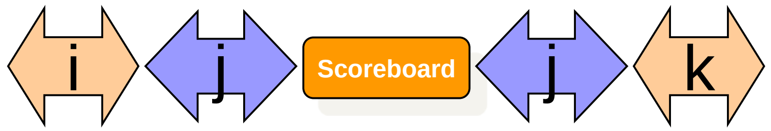
- Advanced formal techniques allow you to simplify the formal TB
  - – ND, DI, Symbolic Variables, Formal VIP, modeling code => minimize state



256 combinations
Selects stable during transmission

- Data integrity end to end
- Symbolic Variables for input/bridge/output
  - Stable - Determines select value
  - ND – formal picks the path
- Proof – all scenarios good, CEX shows bad path

```
Input  i 3 to 0
Bridge j 3 to 0
Output k 3 to 0
```



selA[j] = i

selB[k] = j

Summary

- Complete as much valuable analysis as possible in your first 24-hours

- Leverage feedback from the tool
  - Use “active logic” to identify problem constructs in the logic *being analysed*
  - Use “Engine Health” to focus on properties least likely to converge
  - Use “Run Monitor” to keep watch over all the runs

- Leverage the tool commands to reduce design size
  - Use blackbox commands to remove certain module/instance
  - Use cutpoint command to remove the fan-in logic of the specified signal
  - Use compile switches to reduce parameter sizes

- If problematic constructs are found, modify your setup and re-run
  - Add/remove/recode assumptions (a/k/a constraints)
  - Recode assertions: formal-friendly coding as per the Two Great Laws, decomposition
  - Reduce design size