Sign-off with Bounded Formal Verification Proofs

NAMDO KIM
SAMSUNG
JUNHYUK PARK
SAMSUNG
HARGOVIND SINGH
OSKI TECHNOLOGY
VIGYAN SINGHAL
OSKI TECHNOLOGY

Oski TECHNOLOGY

Agenda

- What Does Formal Sign-off Mean
- Why Bounded Proof is Necessary in Formal Sign-off
- How to Compute Required Proof Bound
- Case Study 1: CCX design
  - Determining Required Proof Bound
  - Use Abstraction Models to reduce Required Proof Depth to Achieve Sign-off
- Case Study 2: NOC design
  - Determining Required Proof Bound
  - Results

What Does Formal Sign-off Mean

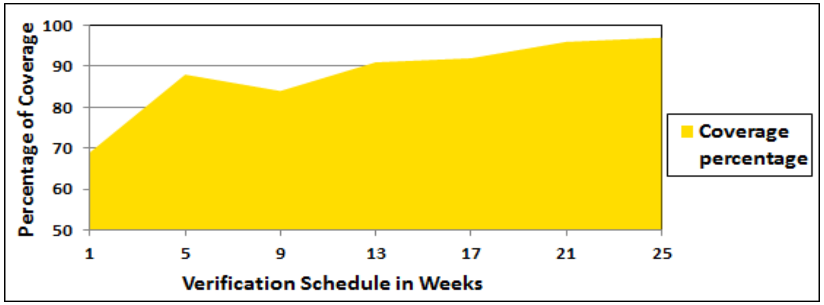
Unique Methodology. Highest Coverage. Fastest Time to Market.

Sign-offs

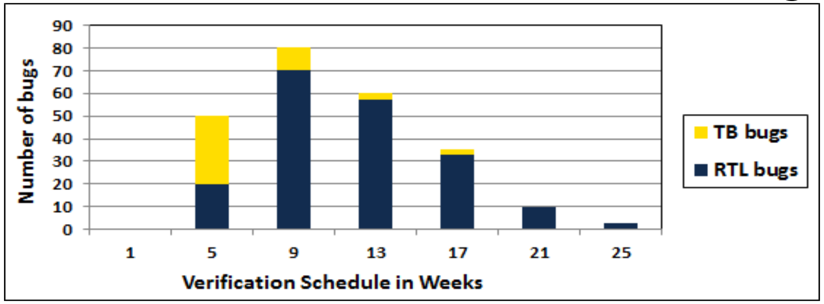
- What does sign-off mean to program managers?
  - Ready for tape-out
- Sign-off requires
  - Commitment to finish – else task is optional, and may be killed
  - Metrics to measure progress
  - Nightly/weekly regression runs
- Common sign-off flows
  - Static timing
  - Simulation (spreadsheet and coverage)
  - Power
  - RTL-vs-gates LEC

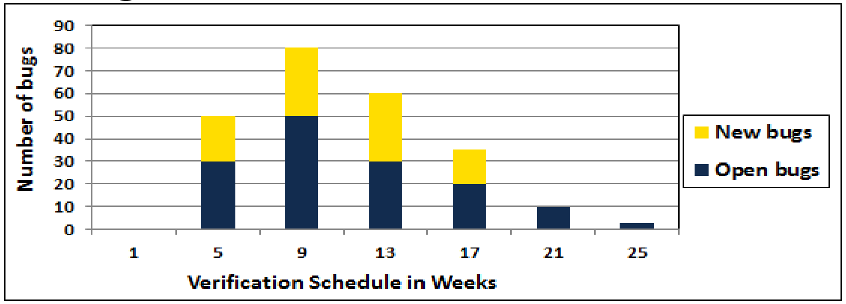
Verification (Simulation) Manager's Dashboard

Coverage tracking

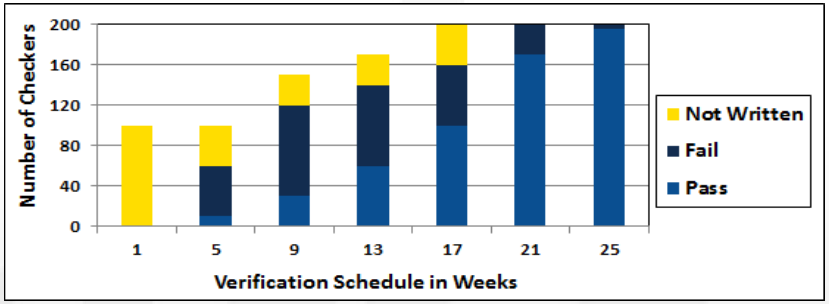


Bug tracking

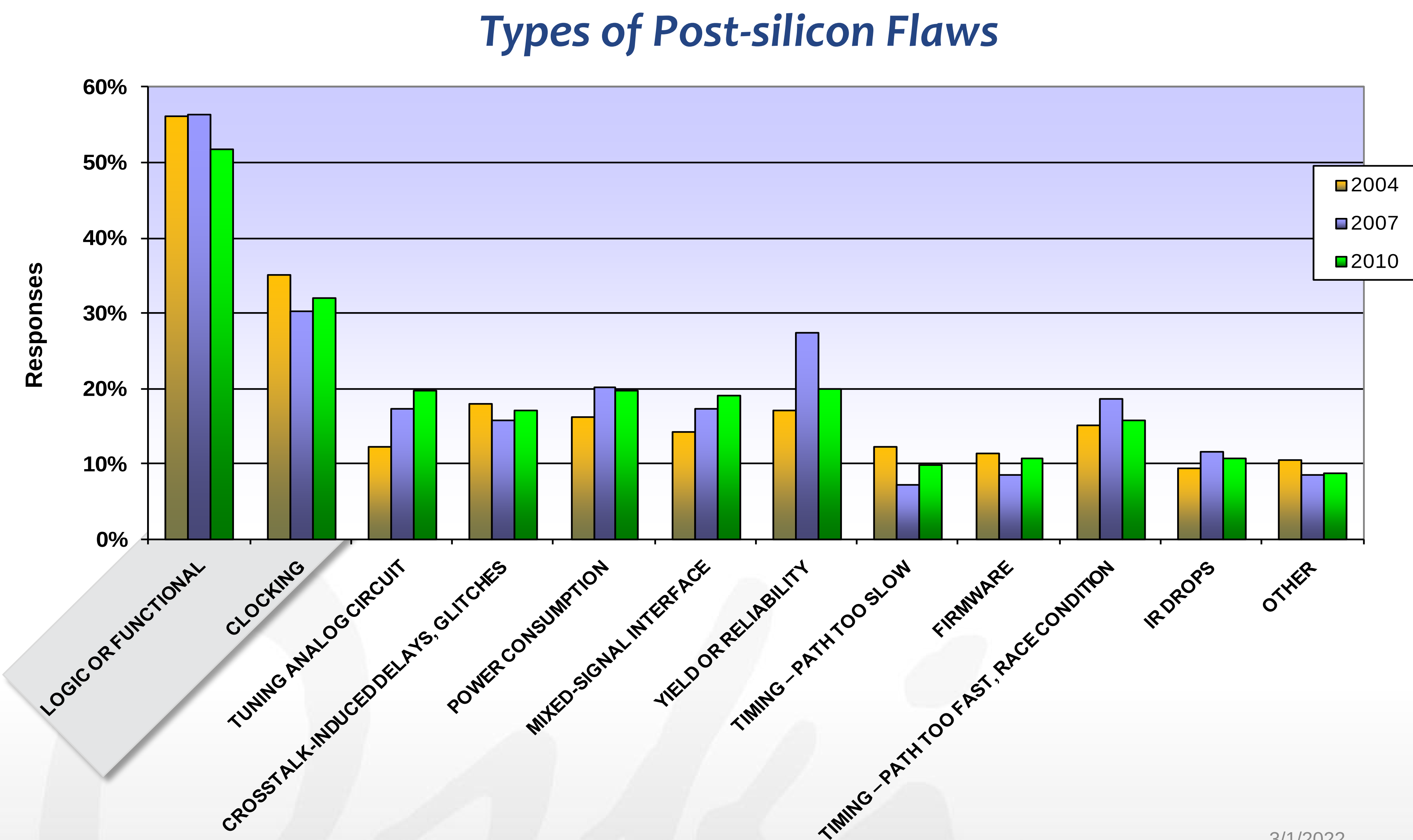




Runtime status

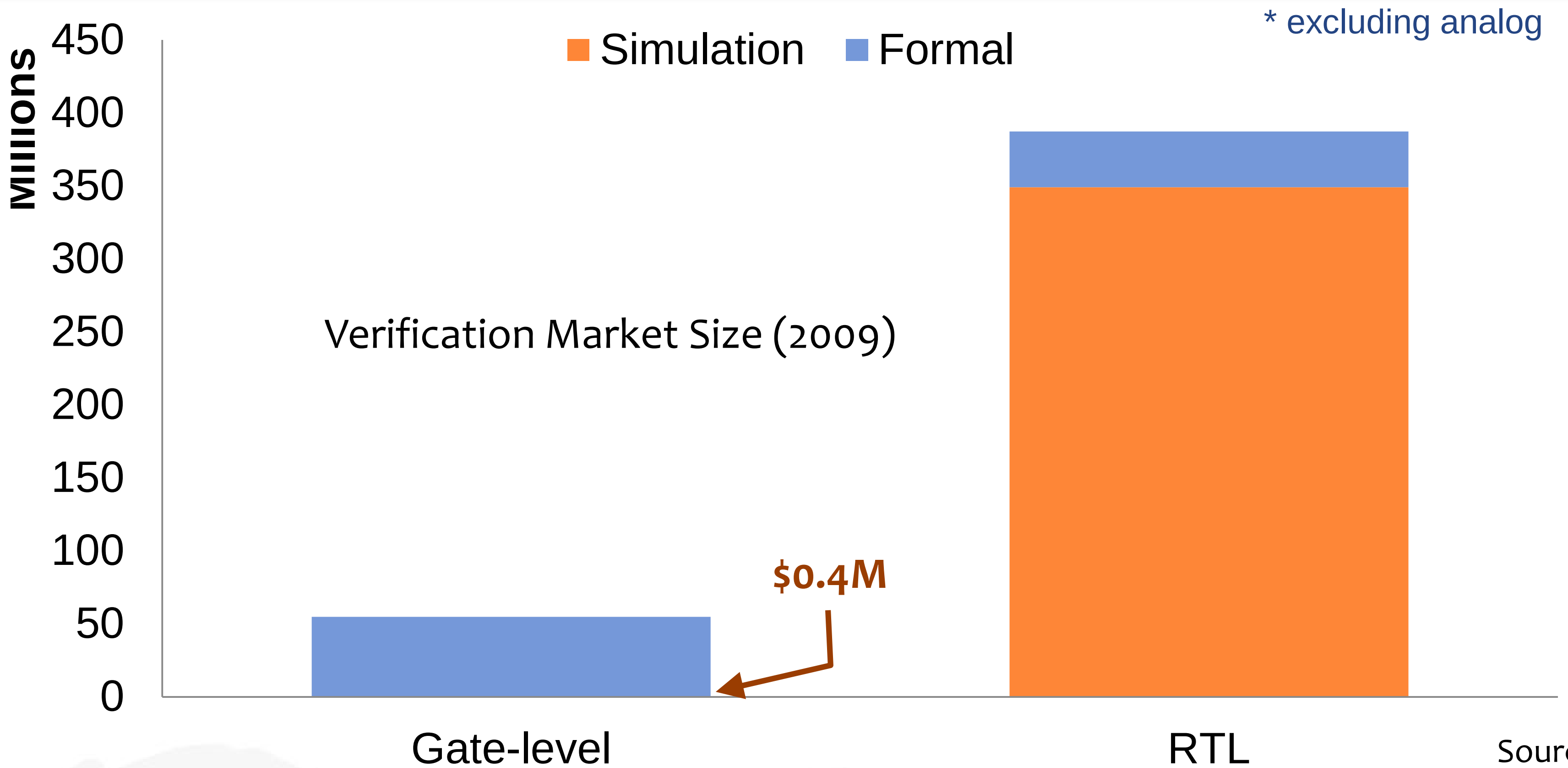


Simulation Sign-off Alone Not Sufficient



Wilson Research Group and Mentor Graphics
2010 Functional Verification Study. Used with permission.

The Solution – Formal Needs to Be Widely Adopted



Source:
Gary Smith EDA, October 2010

- **Gate-level formal (equivalence checking)**
  - **Then (1993):** Chrysalis; **Now:** Cadence (Verplex), Synopsys
- **RTL formal (model checking)**
  - **Then (1994):** Averant, IBM; **Now:** Cadence, Jasper, Mentor, Synopsys

Formal Reality in Industry

- Around for 20+ years
- Expectations has been set high
  - Low efforts for constraints
  - Tools will complete proofs
- Expectations have been set low
  - Only verify local assertions
  - No End-to-End proofs
- Perception
  - Low bang-for-the-buck
  - Not worthy of "sign-off"
- Training and staffing
  - Few places to learn formal application
  - How to build a productive formal team

Explored

Undetermined

The biggest challenges is users don't know what to do with inconclusive results

Inconclusive

Proof Terminated

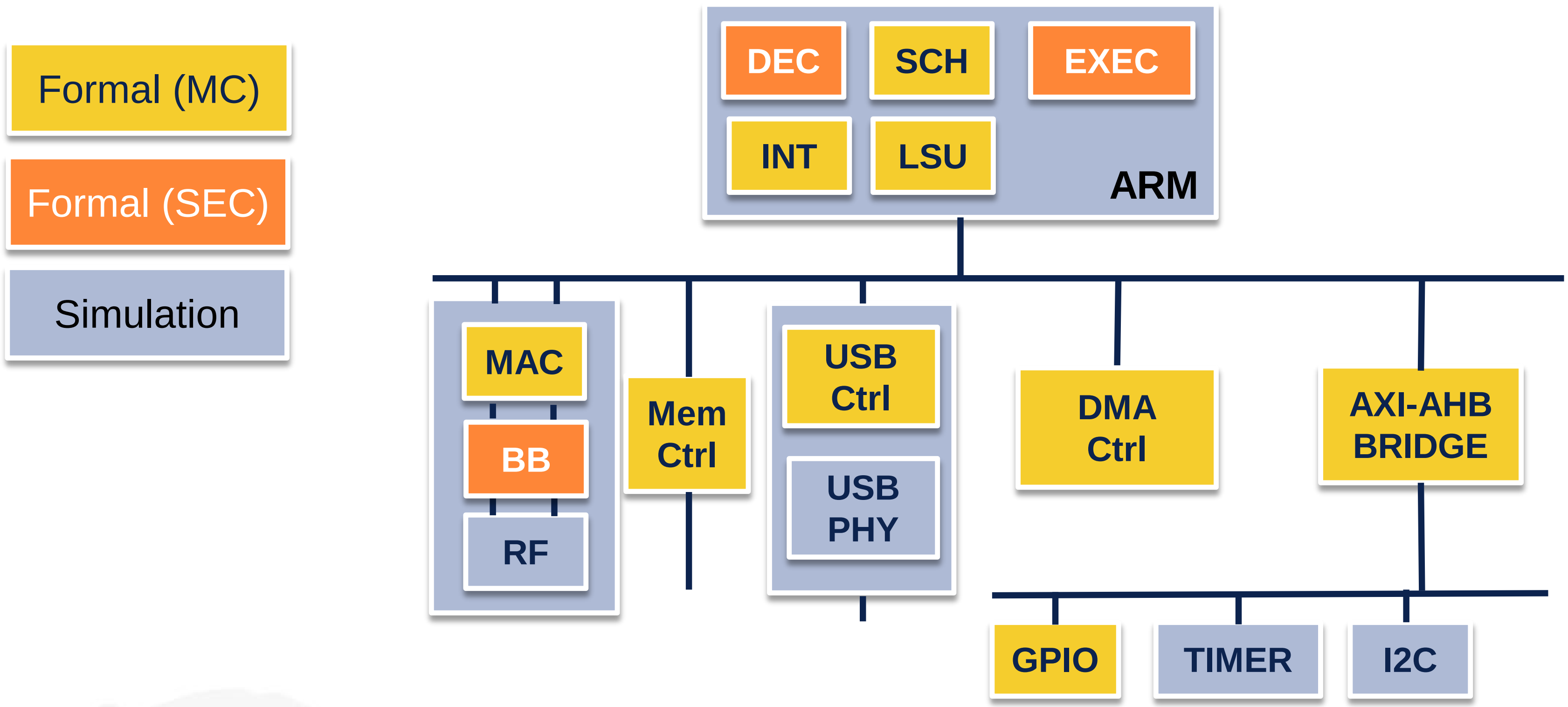
The Facts About Inconclusive Results

Myth: Inconclusive proof results are not useful

- Facts:
  - Most "End-to-End" proofs will results in Bounded Proofs
  - Bounded Proofs are reported with a proof depth
  - Formal guarantees exhaustive search up to proof depth
  - Using Abstraction Models, required proof bounds can be minimized
  - Formal coverage validates proof depths and formal efforts

End-to-End Formal Enables Sign-off

End-to-End Formal Complements Simulation



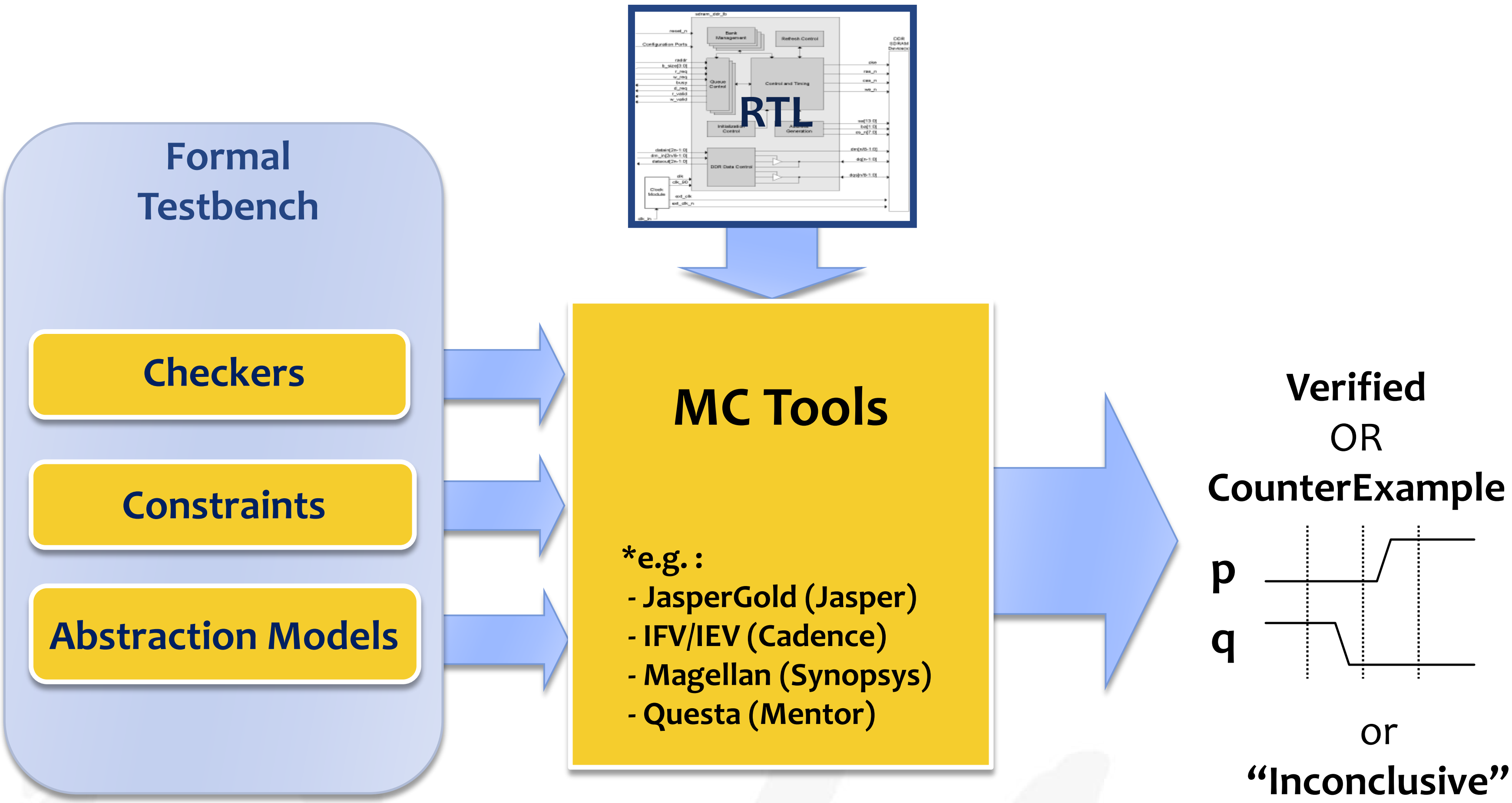
- Different designs suited for different methods (MC, SEC or simulation)
- Planning at the micro-architectural design stage is critical
- Formal delivers verified IPs for SOC integration

Why Bounded Proof Is Necessary in Formal Sign-off

Unique Methodology. Highest Coverage. Fastest Time to Market.

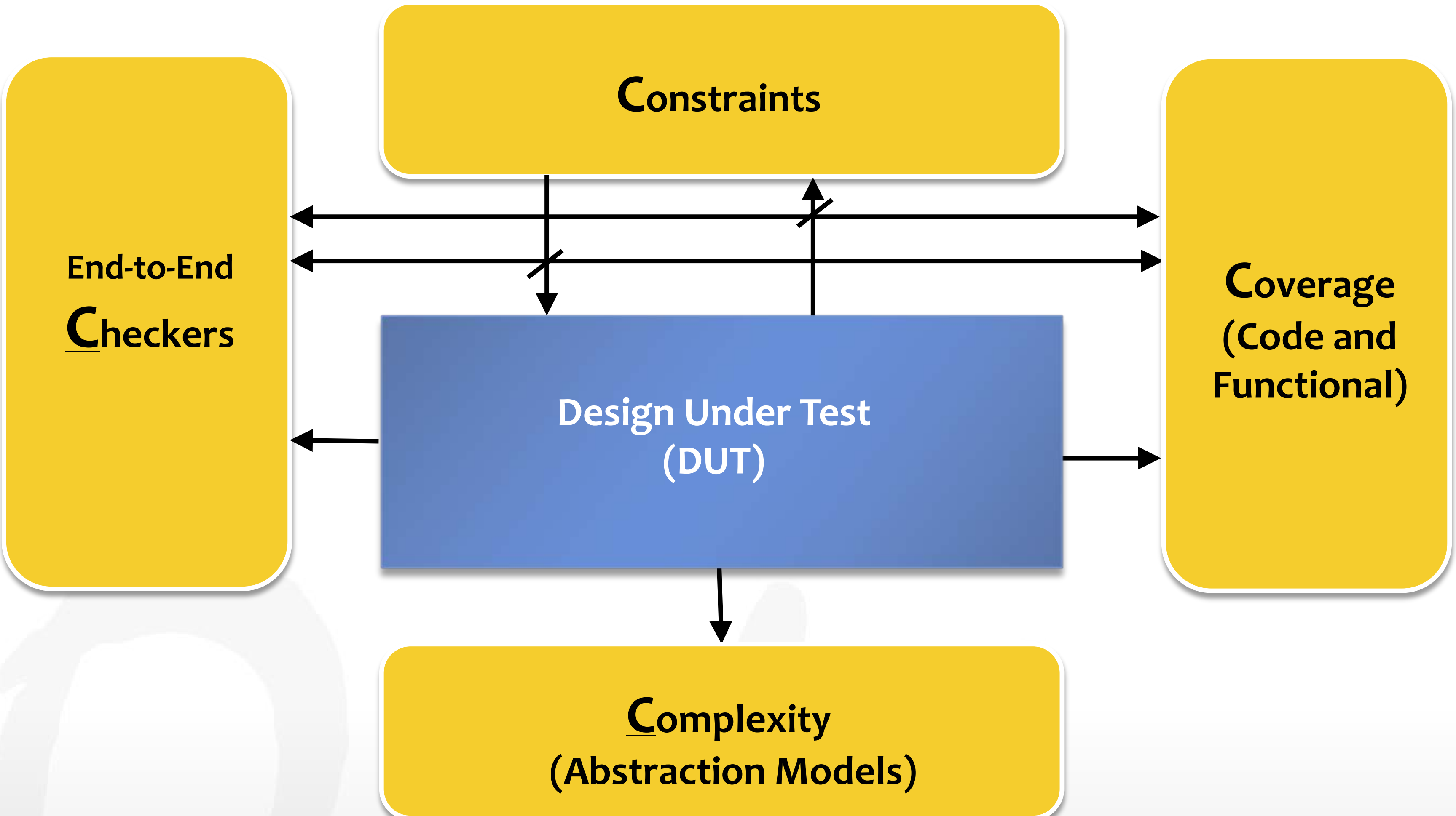
Oski TECHNOLOGY

Model Checking (MC)



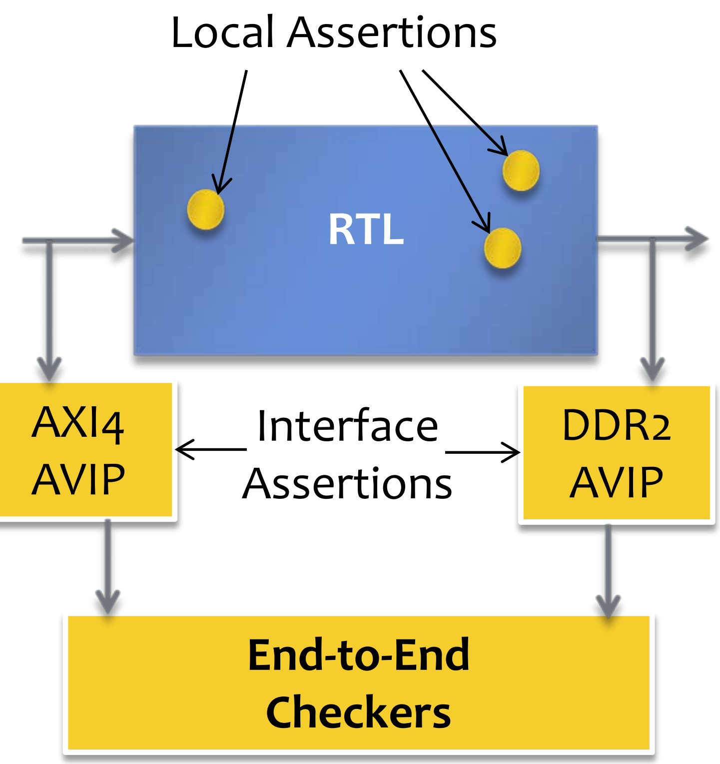
Oski Formal Sign-off Methodology

Measurable & Dependable as Simulation Sign-off



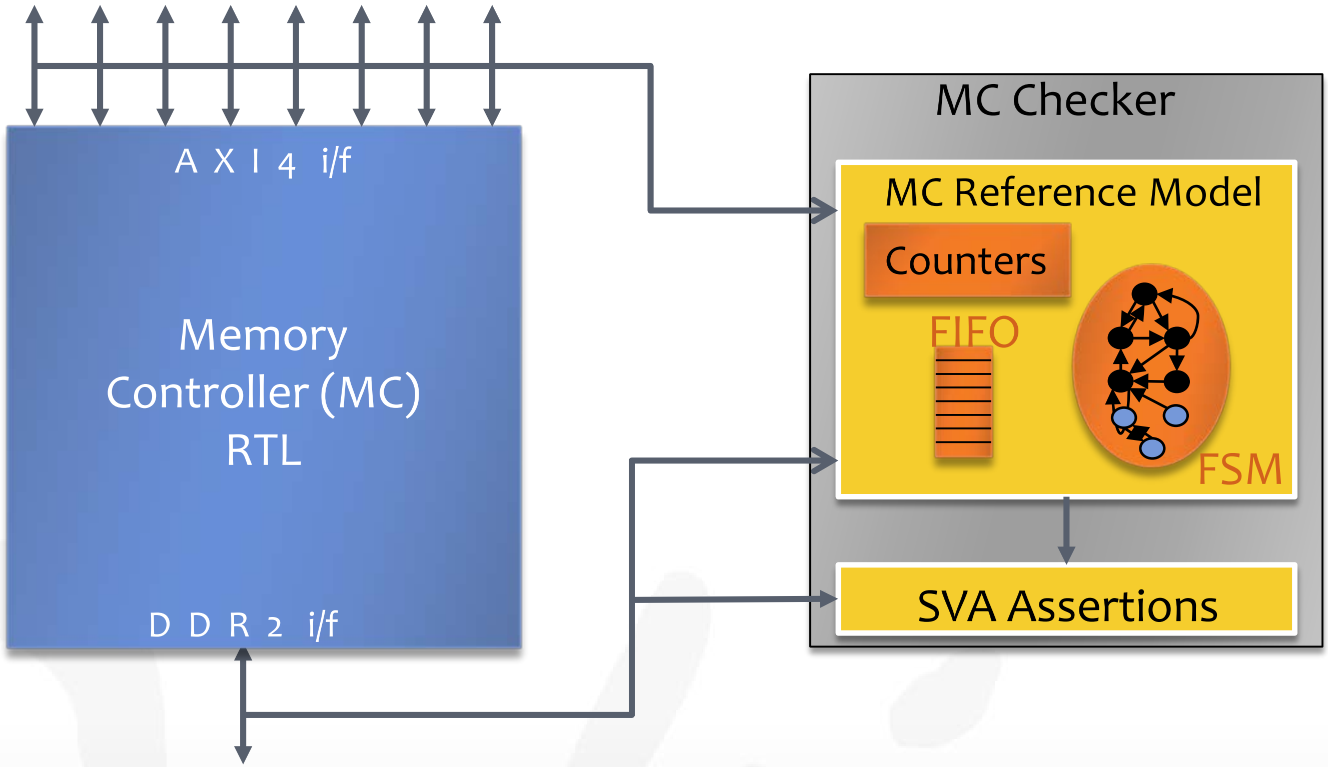
What is End-to-End Formal?

- Local Assertions – easier to verify
  - Internal RTL assertions, embedded in RTL
  - Protocol assertions
    - valid/ready handshake, AXI4, DDR2…
- Interface Assertions – harder to verify
  - Relate to inputs/outputs
- End-to-End Checkers – hardest to verify
  - Model end-to-end functionality
  - Can replace simulation
  - Often requires abstraction models to manage complexity



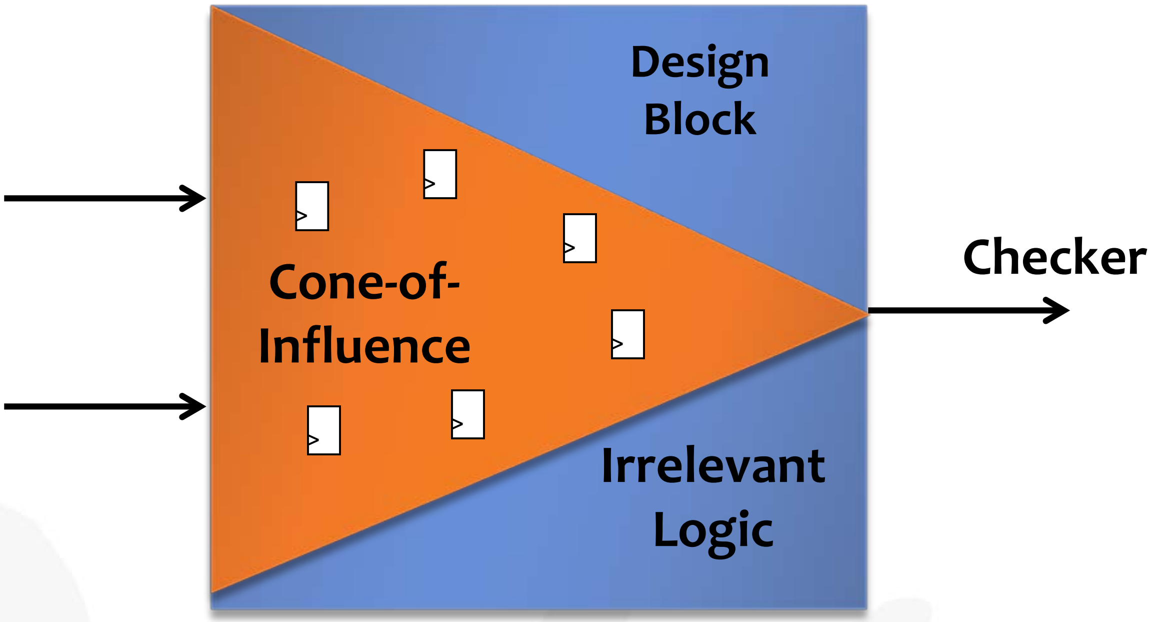
End-to-End Checkers

- 95% of End-to-End Checker is in SV or Verilog; rest is SVA
  - Developing reference model requires time and effort



Complexity – How to Measure

- One coarse measure of Complexity
  - Use number of flops/memory bits in the Cone-of-Influence of the Checker



The Larger the Cone-of-Influence, the More Complex the Proof!

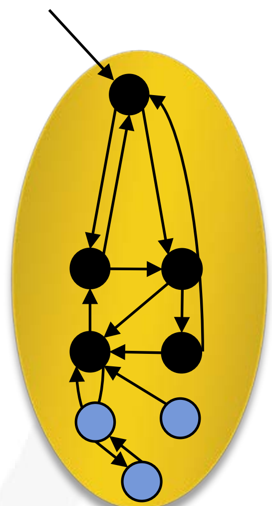
Complexity – Where It Comes From

Checker: (st == 2’b01) => ~b

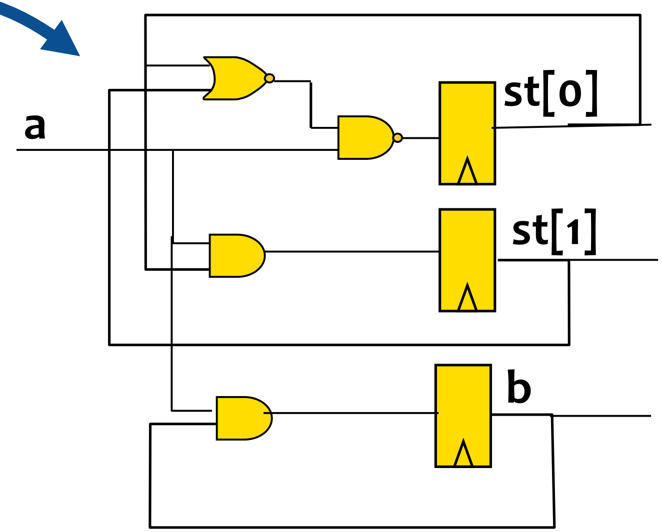
RTL

```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2’b00;
  else case( st )
        2’b00:  if (~a) st <= 2’b01;
        2’b01:  st <= 2’b10;
        2’b10:  if (a) st <= 2’b00;
        endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1’b0;
  else if (~a | b) b <= 1’b0;
  else b <= 1’b1;
```

Internal STG



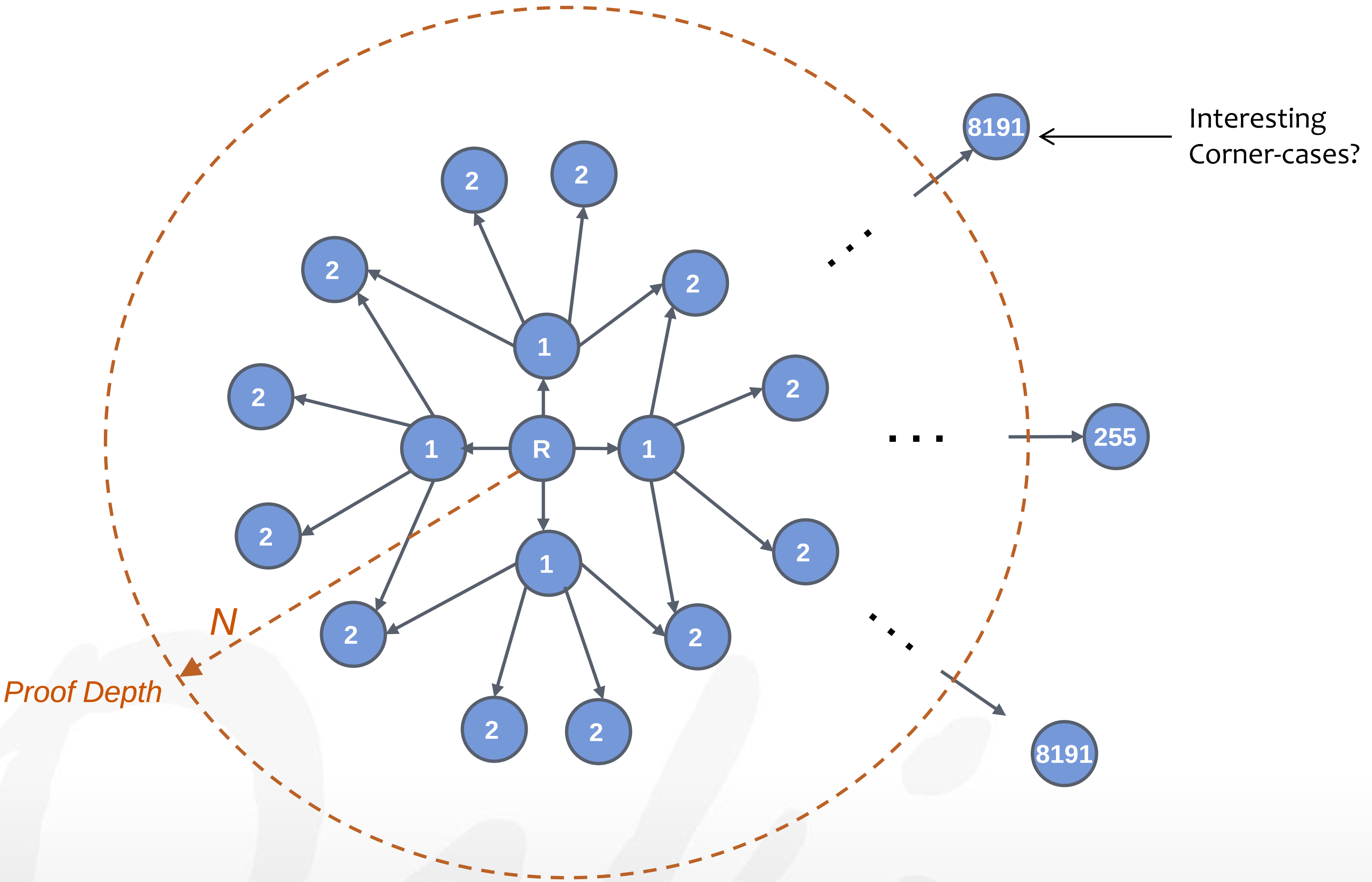
Internal Netlist



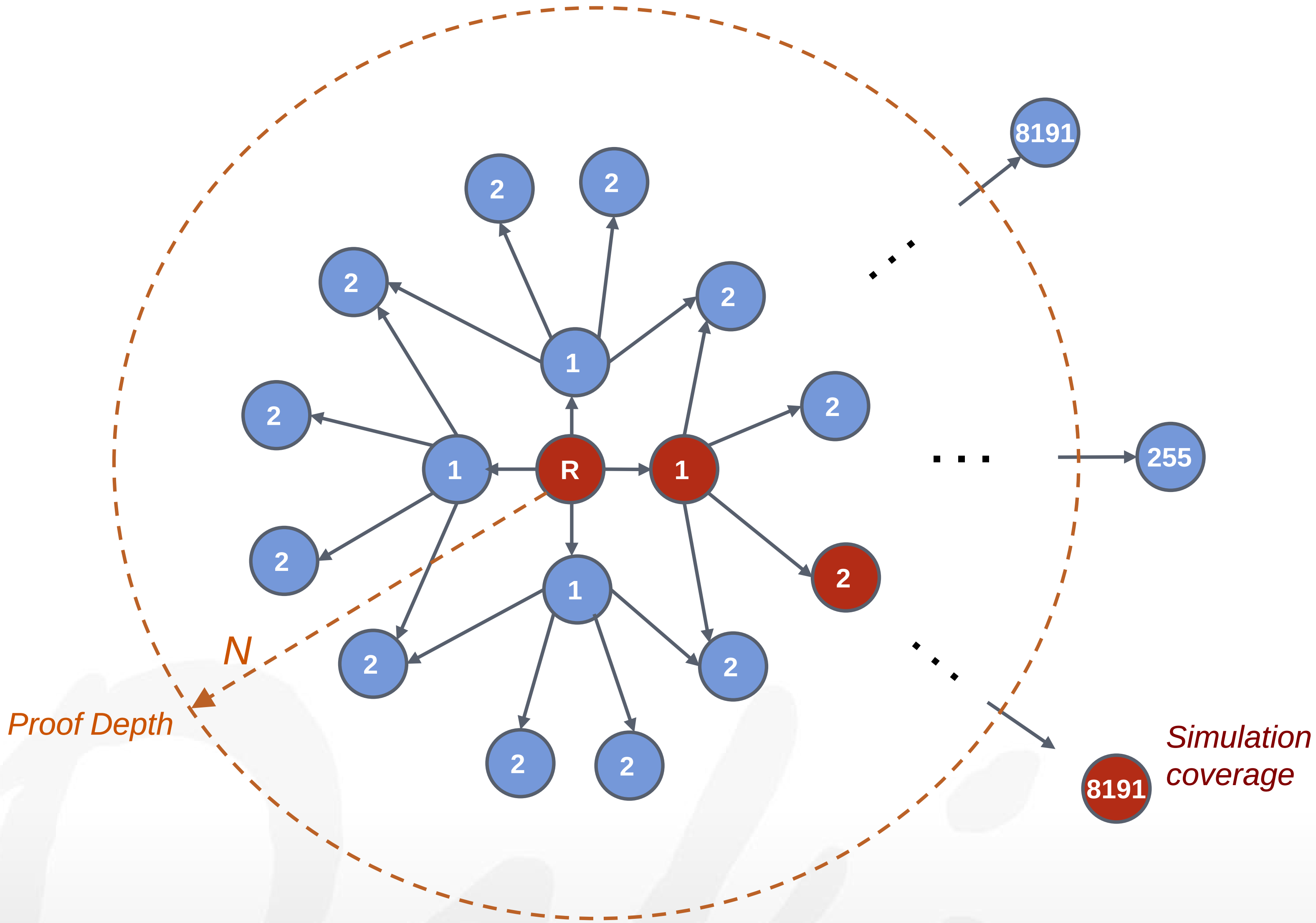
$2^3 = 8$
$2^{10} = 1,024$
$2^{20} = 1,048,576$
$2^{30} = 1,073,741,824$

Formal Complexity Comes from Search Space Explosion!

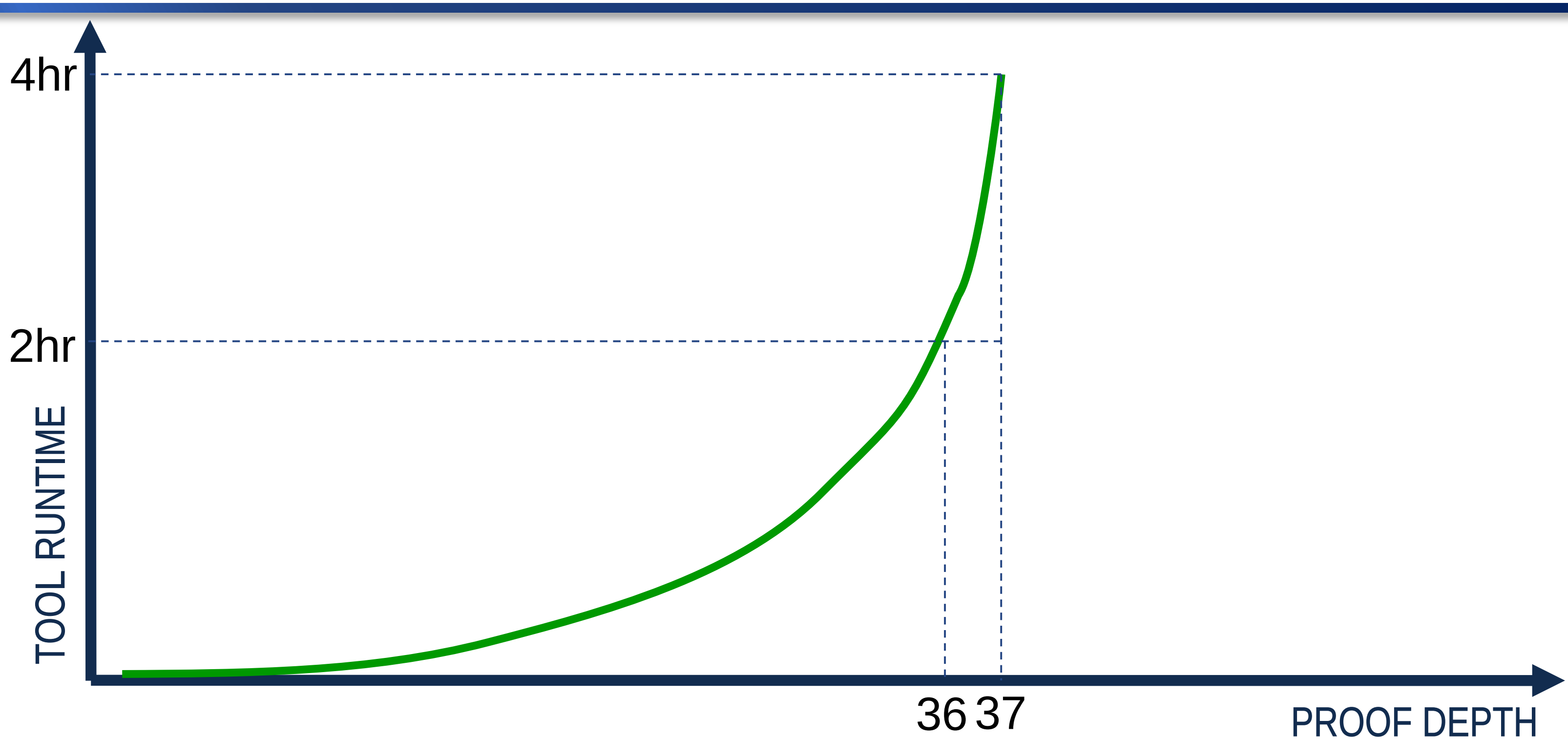
Formal Covers All State Transitions Within Proof Depth



Simulation Covers One Path of State Transition



We Can't Fight the Exponential!



- Unbounded Proof results can be unpredictable
  - Depends on engine finding an inductive invariant
- Bounded Model Checking is more predictable
  - Although, runtime comparison has a much larger variance than proof depth comparison
- Maximize engineer productivity
- Use Abstraction Models to reduce Required Proof Depth

How Perfect Does Formal Have to Be?

Graphic: MacGregor Marketing

- Formal has to be more cost-effective than the alternative
- Formal is not perfect
- Deep enough bounded proofs are good enough
- Still need to have checks and balances in place (like anything else)

How to Compute Required Proof Bound

Unique Methodology. Highest Coverage. Fastest Time to Market.

Required Proof Bound

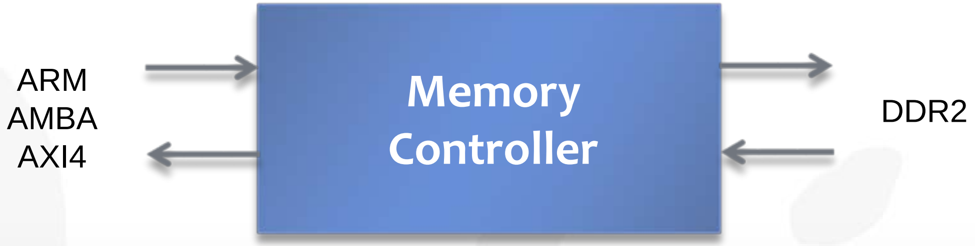
- Premise:
  - A proof depth that gives us the "coverage" that "we need"
    - Similar to simulation sign-off
    - Can be measured with commercial formal tools
  - A proof depth that will not miss any RTL bug
    - Bounded proof is as good as full proof, offering formal sign-off

Determining Required Proof Bound

- Based on:
  1. Latency analysis of design
  2. Micro-architectural analysis (with, and without designer)
  3. Covers for "interesting" corner-cases
  4. Formal coverage
  5. Failures seen during formal verification
  6. Safety nets
     a. Missed bugs found in simulation
     b. Missed bugs found in hybrid regression runs

1. Latency Analysis

- Analyze latency from one end of the design to the other end
  - Provides a lower bound, not the required proof bound
- Layer in additional proof depth due to:
  - Initialization
  - Multiple input streams
  - Long packets (if it matters to the design)
  - Error cases



2. Micro-architectural Analysis

- Analyze micro-architectural structures:
  - State machines
  - Counters
  - FIFOs
  - RAMs
  - Linked lists
- Analyze RTL code
  - Deeply nested if-then-else, or case statements
- "Determine" proof depth necessary to exercise all relevant logic

3. Covers for "Interesting" Corner-Cases

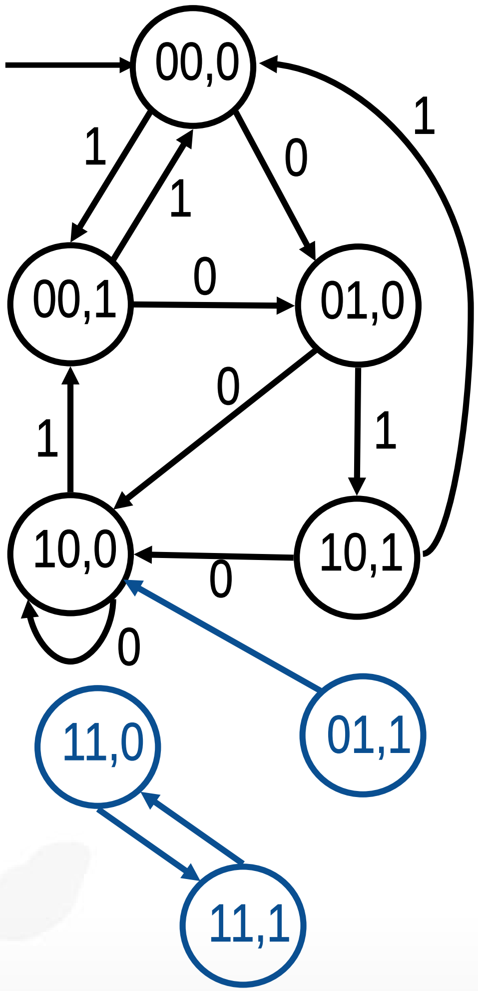
- Similar to functional coverage analysis for simulation
- Brain-storm interesting corner-cases:
  - Exercise corners of RTL
  - Ignore corner-cases that are not relevant to RTL
    - E.g. longer sequences than RTL cares about (similar to mistakes in functional coverage for simulation)
- Implement covers as properties, and run formal for a minimum required proof depth

4. Formal Coverage

- Formal Coverage measures quality of formal effort
  - Provide quantifiable measure to judge whether:
    - Constraints are complete
      - ☐ Can identify effects of over-constraints situation
    - Complexity strategy is complete
      - ☐ Coverage of proof depths with/without Abstraction Models
    - Checkers are complete
      - ☐ Requires observability coverage
        - ☐ Could be substituted by proof cores (if we get unbounded proofs!)

Code Coverage vs STG Coverage

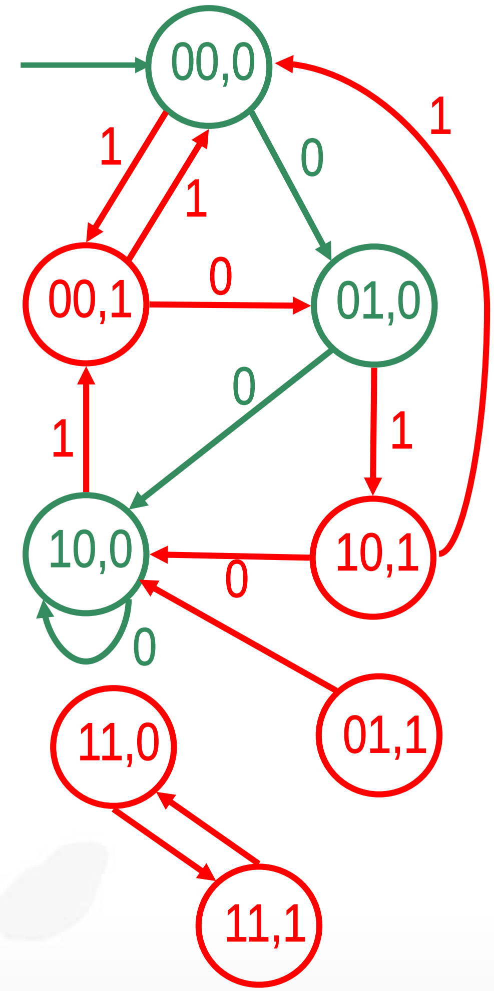
```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2'b00;
  else case (st)
    2'b00: if (~a) st <= 2'b01;
    2'b01: st <= 2'b10;
    2'b10: if (a) st <= 2'b00;
    endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1'b0;
  else if (~a | b) b <= 1'b0;
  else b <= 1'b1;
```



3/1/2022

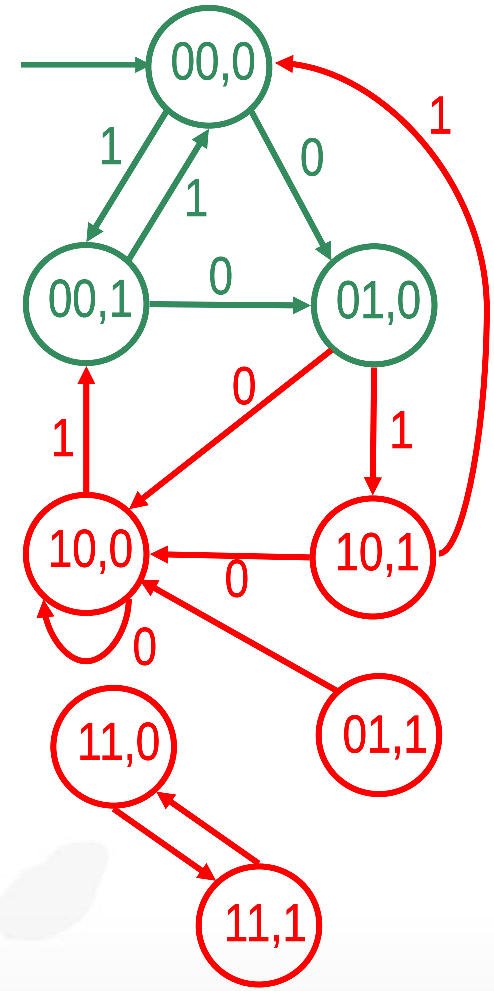
Simulation Coverage (a = 0)

```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2'b00;
  else case (st)
    2'b00: if (~a) st <= 2'b01;
    2'b01: st <= 2'b10;
    2'b10: if (a) st <= 2'b00;
    endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1'b0;
  else if (~a | b) b <= 1'b0;
  else b <= 1'b1;
```

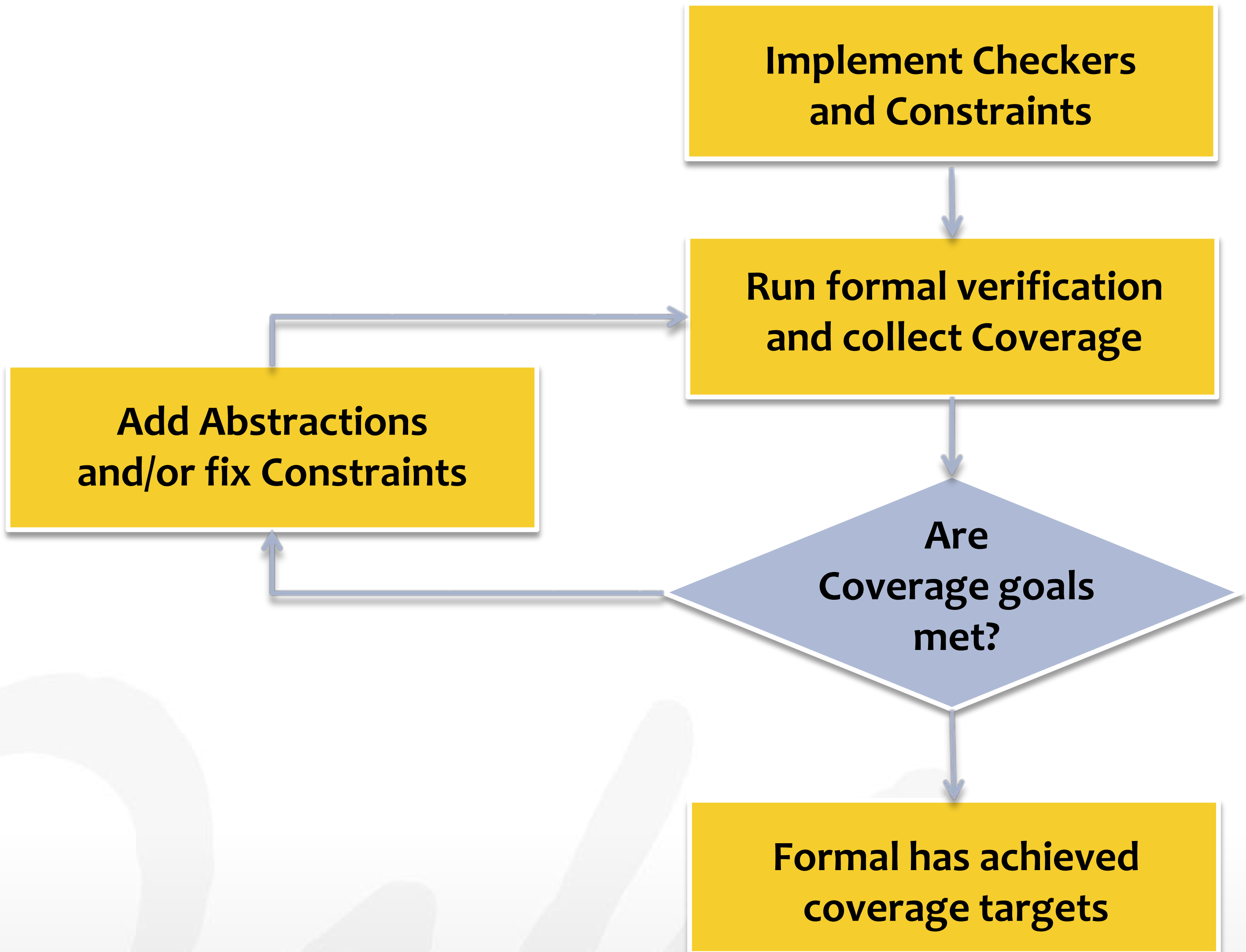


Formal Coverage (depth = 1)

```
input a;
reg b;
reg [1:0] st;

always @(posedge clk or negedge rst)
  if (~rst) st <= 2’b00;
  else case (st)
    2’b00: if (~a) st <= 2’b01;
    2’b01: st <= 2’b10;
    2’b10: if (a) st <= 2’b00;
    endcase

always @(posedge clk or negedge rst)
  if (~rst) b <= 1’b0;
  else if (~a | b) b <= 1’b0;
  else b <= 1’b1;
```

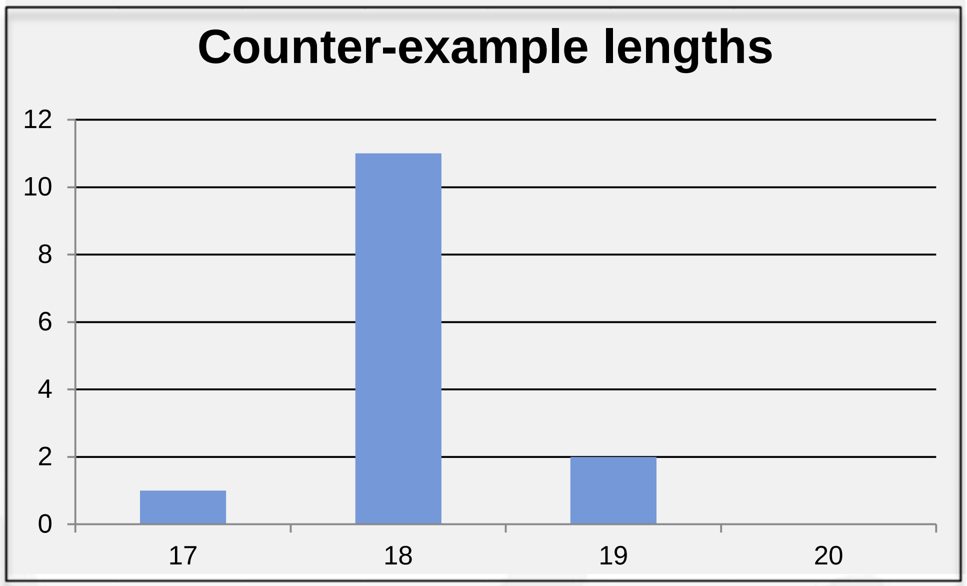


Coverage Methodology



5. Failures seen during Formal Verification

- Usually one End-to-End checker finds most (many) bugs
- Track number of bugs found for every proof depth
- Some proof depths yield a lot of bugs



- Beware of modal behavior
  - Multiple operating modes, discrete jumps in FSMs, or multiples modes of latencies

6. Safety Nets

- Bugs may be missed because:
  - Missing checkers
  - Over-constraints
  - Insufficient proof depth

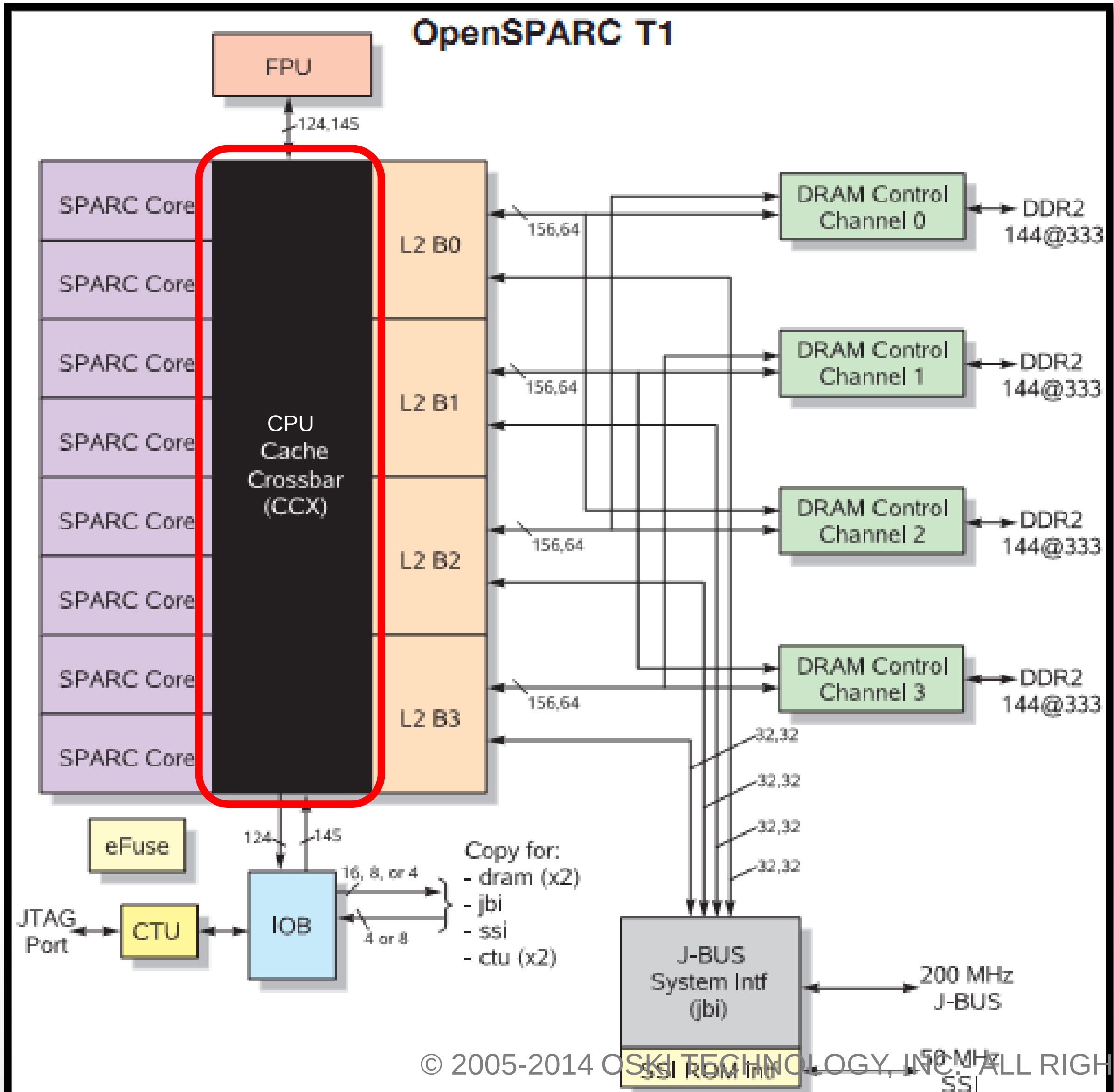
- Watch missed bugs found in simulation
- Set up formal regressions
  - Run formal search from deep states, different states on different days
  - Use bug-hunting engines that are not are exhaustive for a bound (PDR)

Case Study 1
CCX Verification

Unique Methodology. Highest Coverage. Fastest Time to Market.

Oski TECHNOLOGY

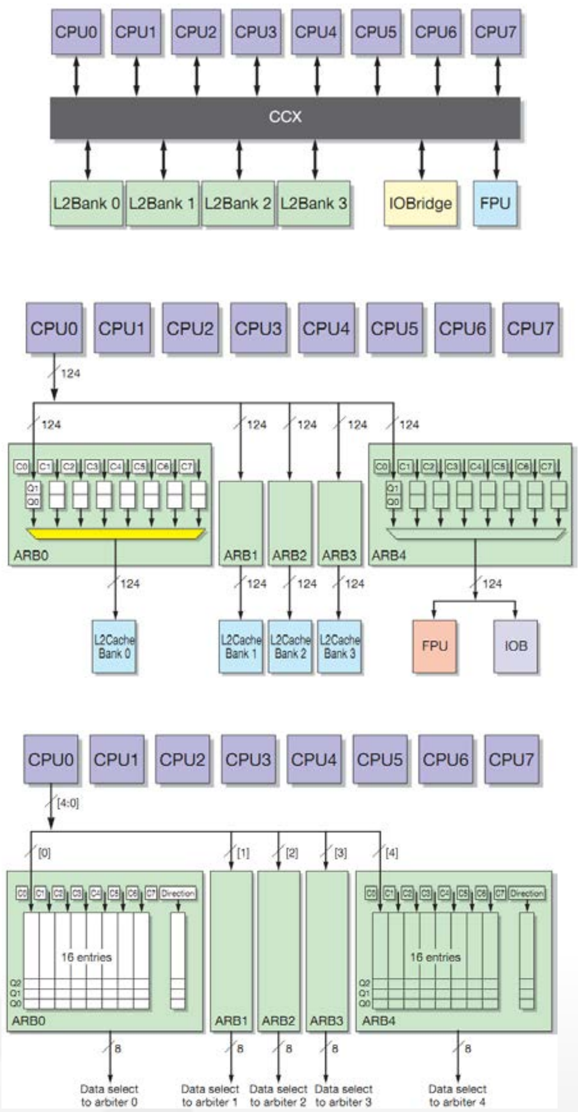
OpenSPARC T1 Design Characteristics



CCX Statistics	
RTL Lines	22,174
Instances	3,227
Inputs	1,982
Outputs	2,005
Flip-flops	30,900

CCX Details

- 8 CPU requestors
  - 1-packet or 2-packet request
- One arbiter per destination
  - 4 L2 banks
  - IOBridge
  - FPU
- 16 deep queue per destination
  - Checkerboard structure, shared across all requestors
  - 2 entries per requestor
  - 2-packet requests consume both locations reserved for a requestor
- 2 flop stages after queue
  - Effectively 18 deep storage structure



CCX Formal Testbench Summary

- 11 checkers
  - 6 end-to-end checkers
  - 5 interface checkers
- 16 constraints

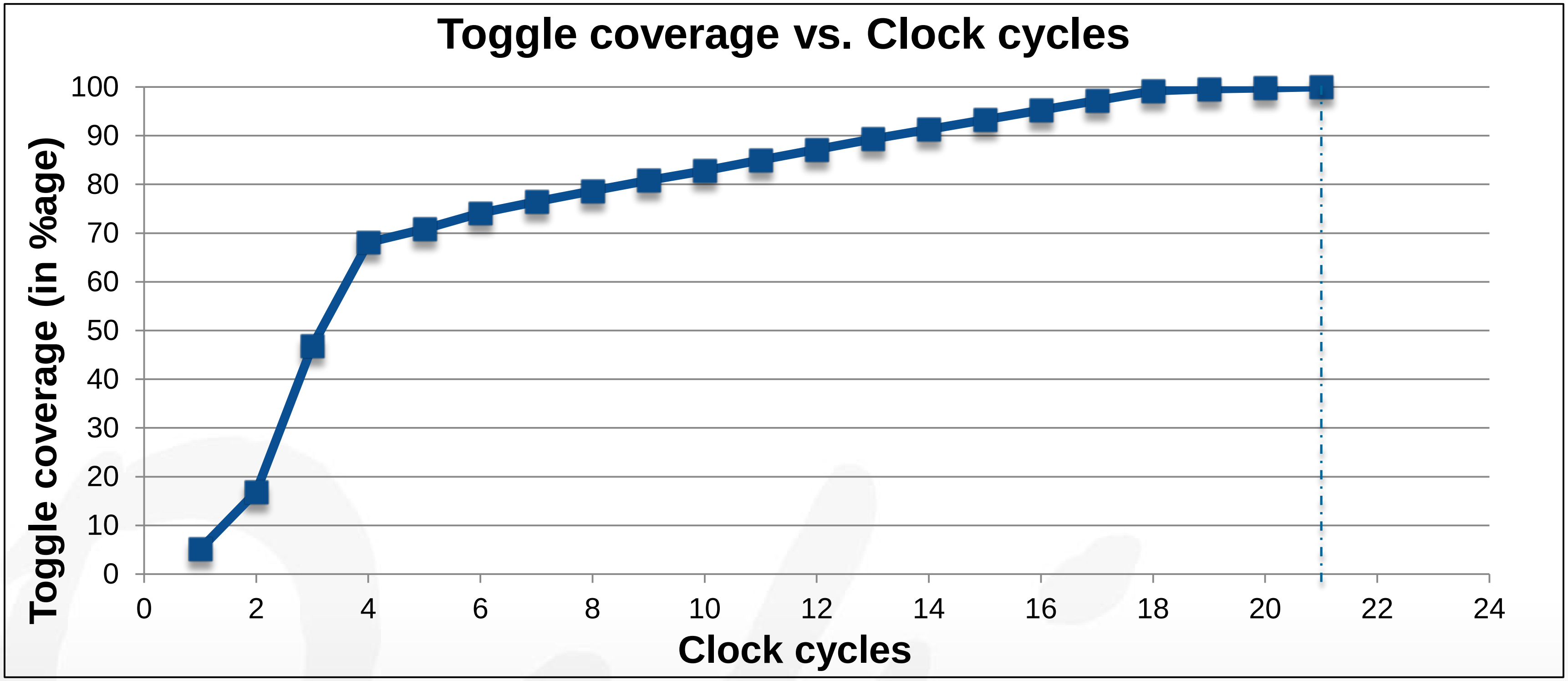
Determining Required Proof Bound (1/2)

1. Latency analysis
   - Cover on output data valid port
     - ☐ Hits in 4 cycles
   - Cover aware of 2-packet request
     - ☐ Hits in 5 cycles
2. Micro-architectural Analysis
   - Exercise all storage locations of 18 deep storage
     - ☐ 21 cycles – Initial latency of 4 cycles + 17 more requests to fill up storage
3. Cover for interesting corner-cases
   - 2-packet requests, stall conditions
     - ☐ Hits in 23 cycles

- Required proof bound – 24 cycles (added 1 cycle for margin)

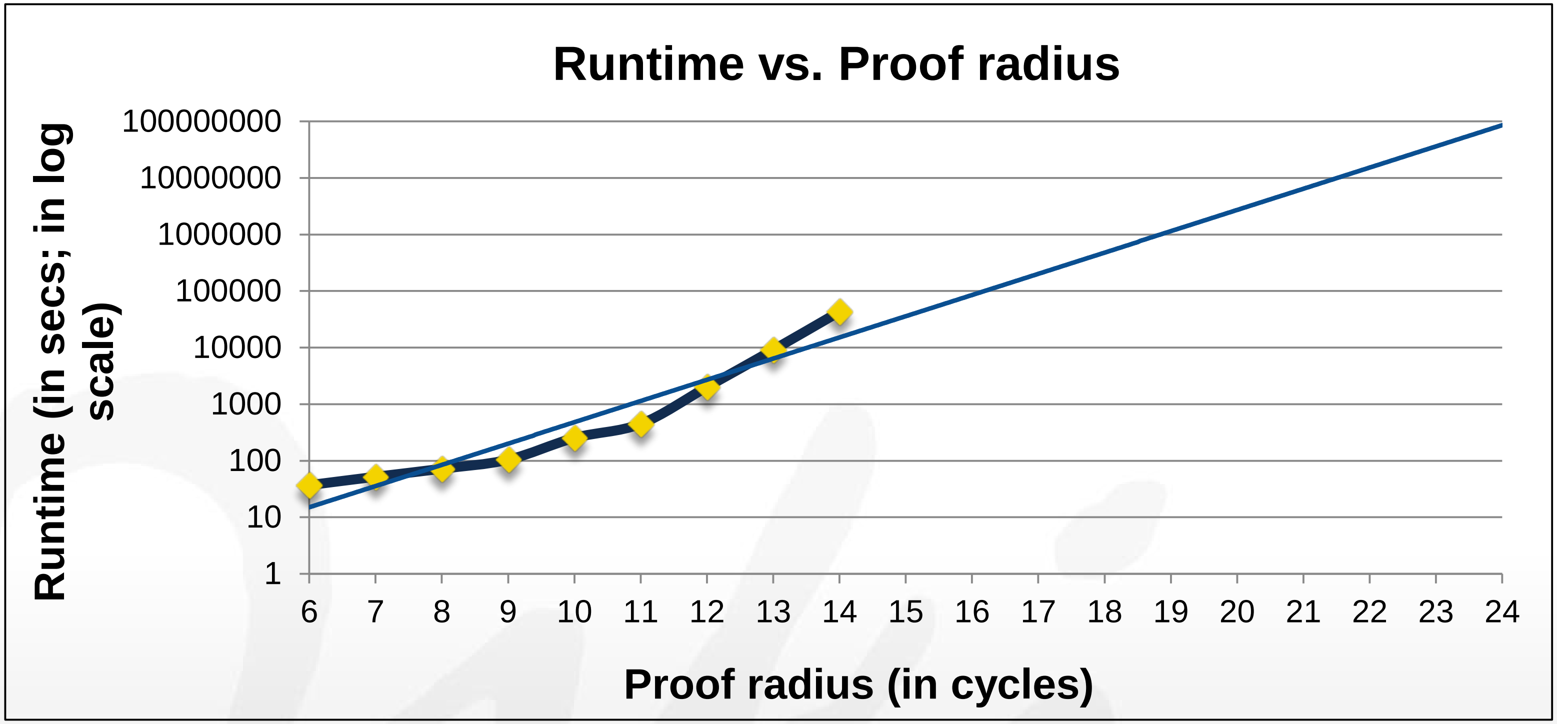
Determining Required Proof Bound (2/2)

4. Formal Coverage
   - 100% toggle coverage achieved in 21 cycles



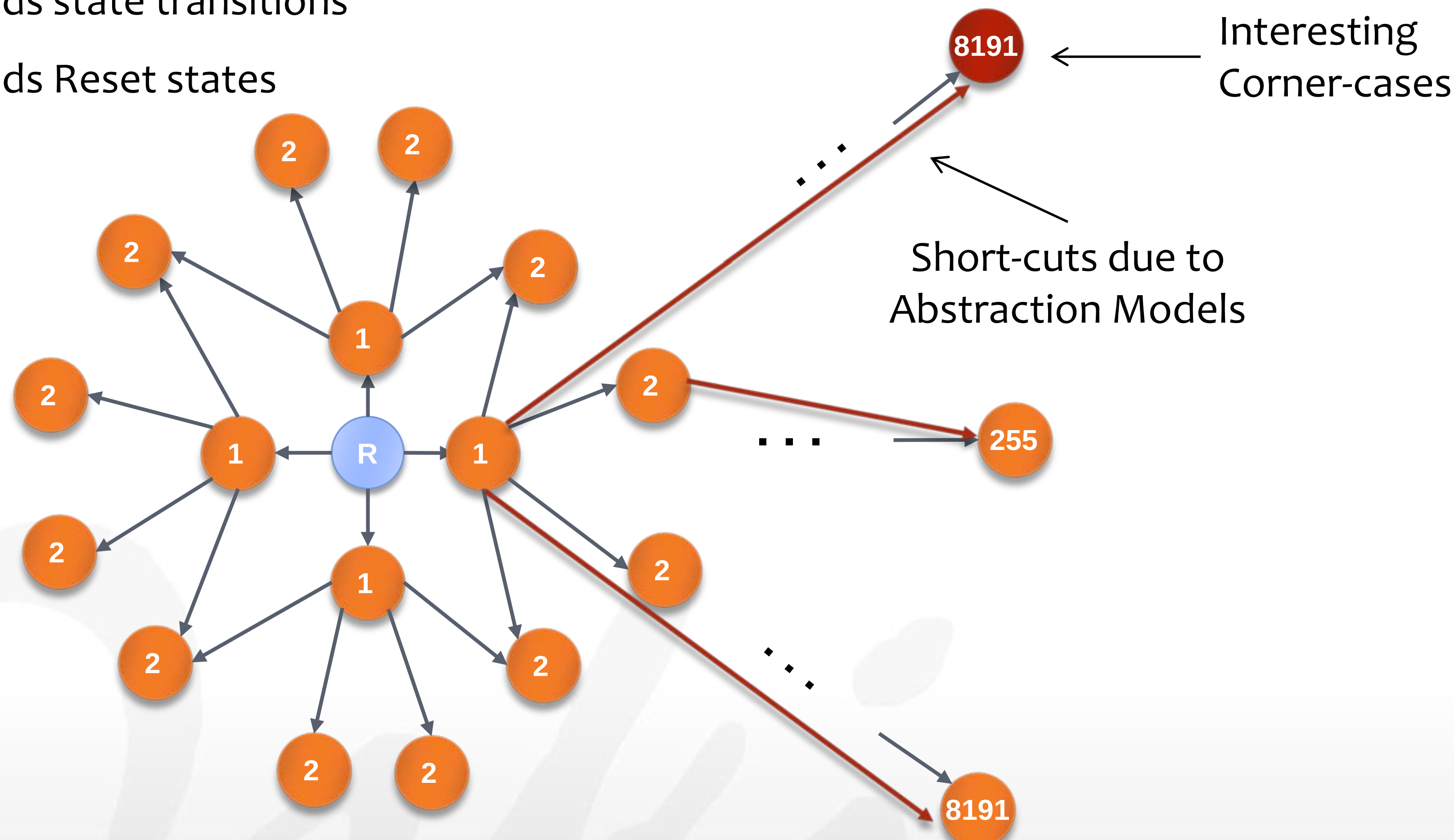
End-to-End Checker: pcx_data_match_A

- Property: when data ready is high, the output data must match the corresponding input data
- **24** cycles are necessary to achieve full proof
- **Run-time** to reach full proof is approx. **991 days** without using Abstraction Models



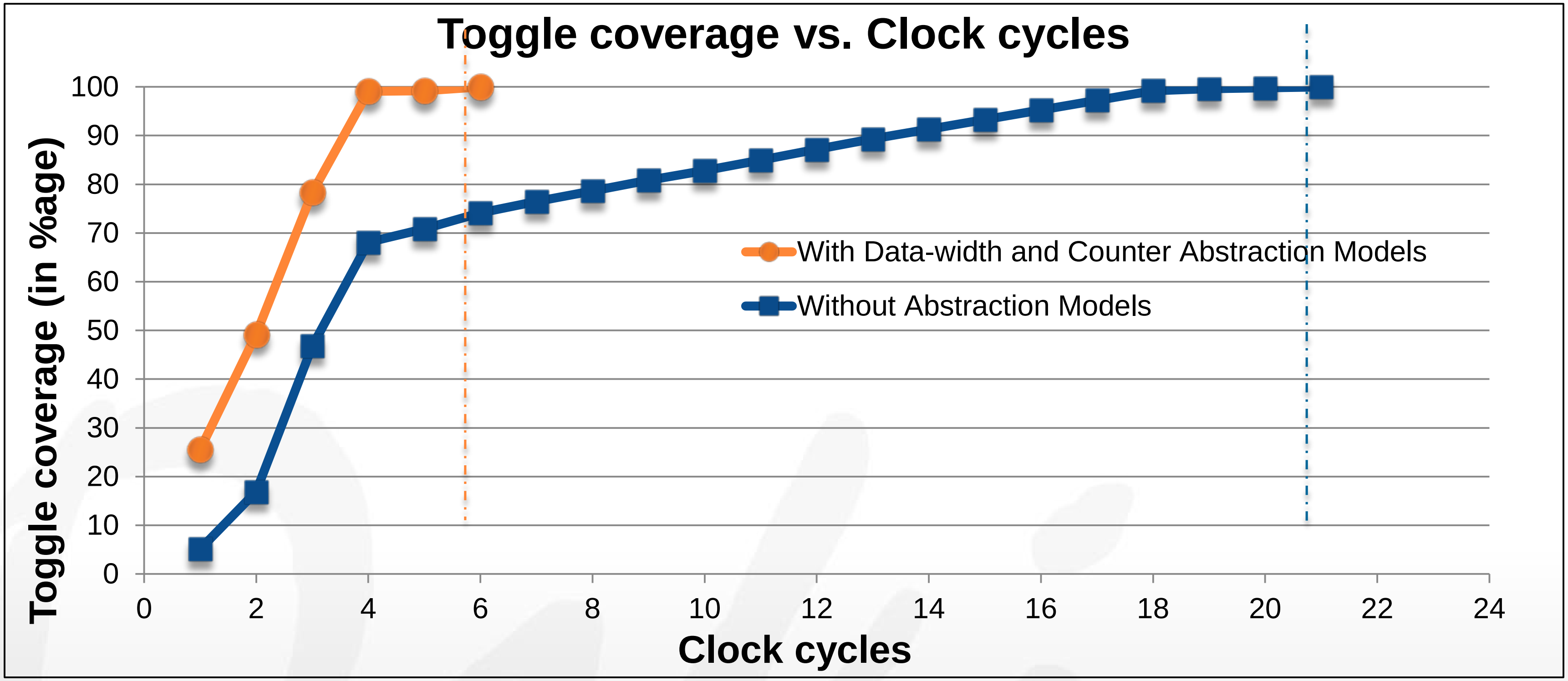
Going Deeper with Abstraction Model

- An "Abstraction Model" of a design is a superset of the design behavior
  - Reduces state space
  - Adds state transitions
  - Adds Reset states



Abstraction Models Achieve Coverage Early

- To achieve 100% toggle coverage
  - With Data-width and Counter Abstraction Models takes **6** cycles (proof depth is 9 cycles)
  - Without Abstraction Models takes **21** cycles (proof depth is 24 cycles)



Abstraction Models Enable Formal Convergence Faster

	Without Abstraction Models	With Data-width Abstraction	With Data-width and Counter Abstractions	
Proof Depth	Runtime (in sec)	Runtime (in sec)	Proof Depth	Runtime (in sec)
10	253	175		
11	447	160		
12	2,008	317		
13	9,112	841		
14	Timeout (8 hours)	1,705		
15	Timeout (8 hours)	5,265		
16	Timeout (8 hours)	25,748		
17-22	Timeout (8 hours)	Timeout (8 hours)	7	56
23	Timeout (8 hours)	Timeout (8 hours)	8	101
24 (full proof)	Timeout (8 hours)	Timeout (8 hours)	9 (full proof)	149

600,000x speedup!

Formal tools **timeout** on original RTL

Abstraction Models **reduce runtime**

Abstraction Models **reduce proof depth**

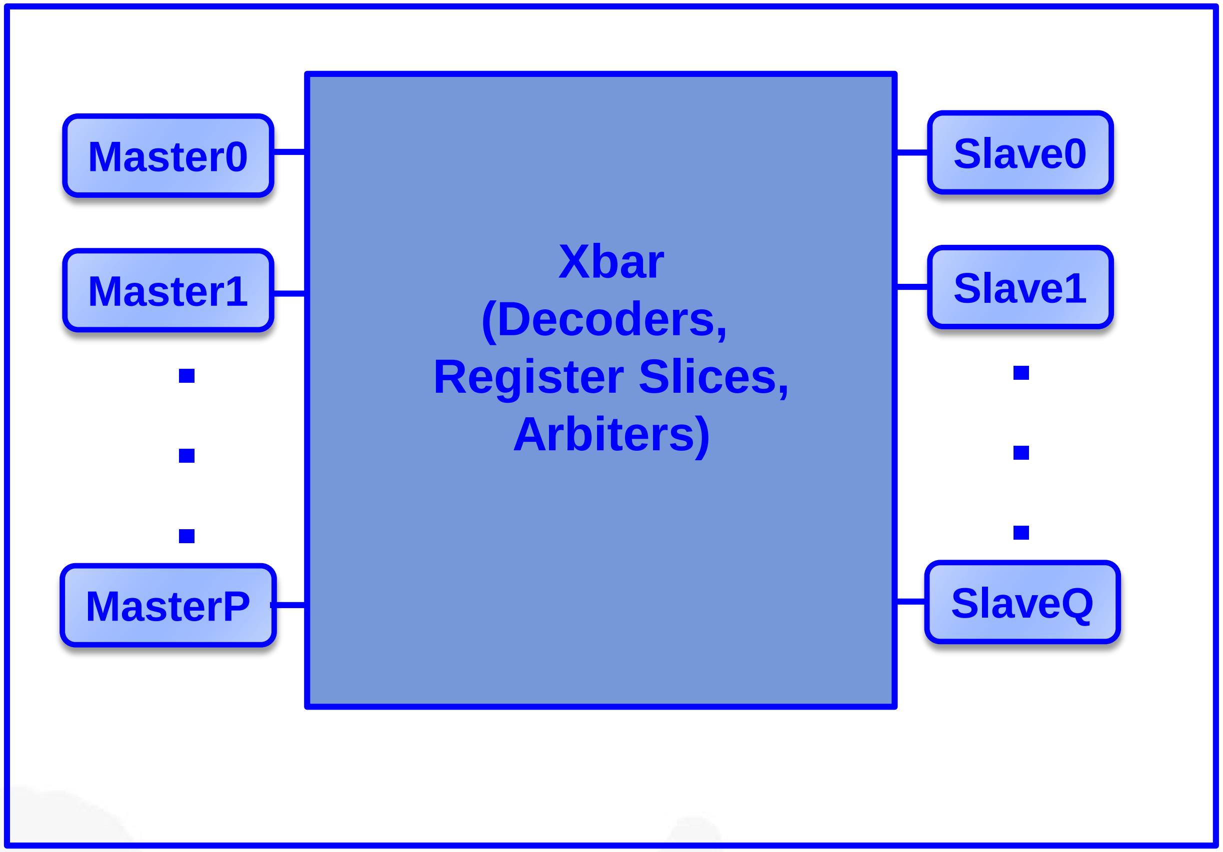
Abstraction Models enable **early formal convergence**

Case Study 2
Deadlock verification in NOC

Unique Methodology. Highest Coverage. Fastest Time to Market.

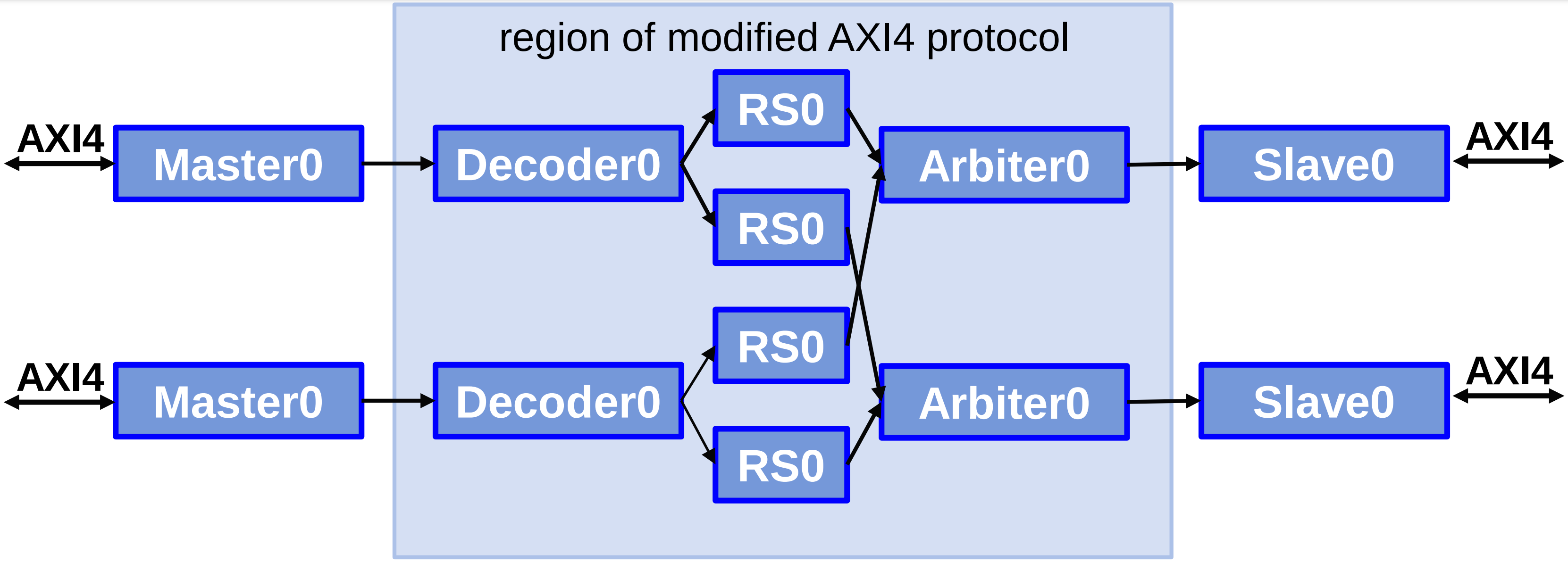
Oski TECHNOLOGY

NOC Block Diagram



- P master and Q slaves
  - Interconnect having decoders, register slices (RS) and arbiters

2-Master, 2-slave DUT



2x2 NOC flop Count	
Master	10,446
Decoder, RS, Arbiter	2,104
Slave	7,486
Total	20,036

Verification Strategy

- Slave buffers and re-orders transactions to obey AXI4
- Master can never contribute to deadlock
- Deadlock problem divided in 3 parts
  - Decoder-RS-Arbiter network doesn't deadlock
  - Decoder-RS-Arbiter network follows modified AXI4 protocol
  - Slave doesn't deadlock
- DUT - Decoder-RS-Arbiter network

Determining Required Proof Bound (1/3)

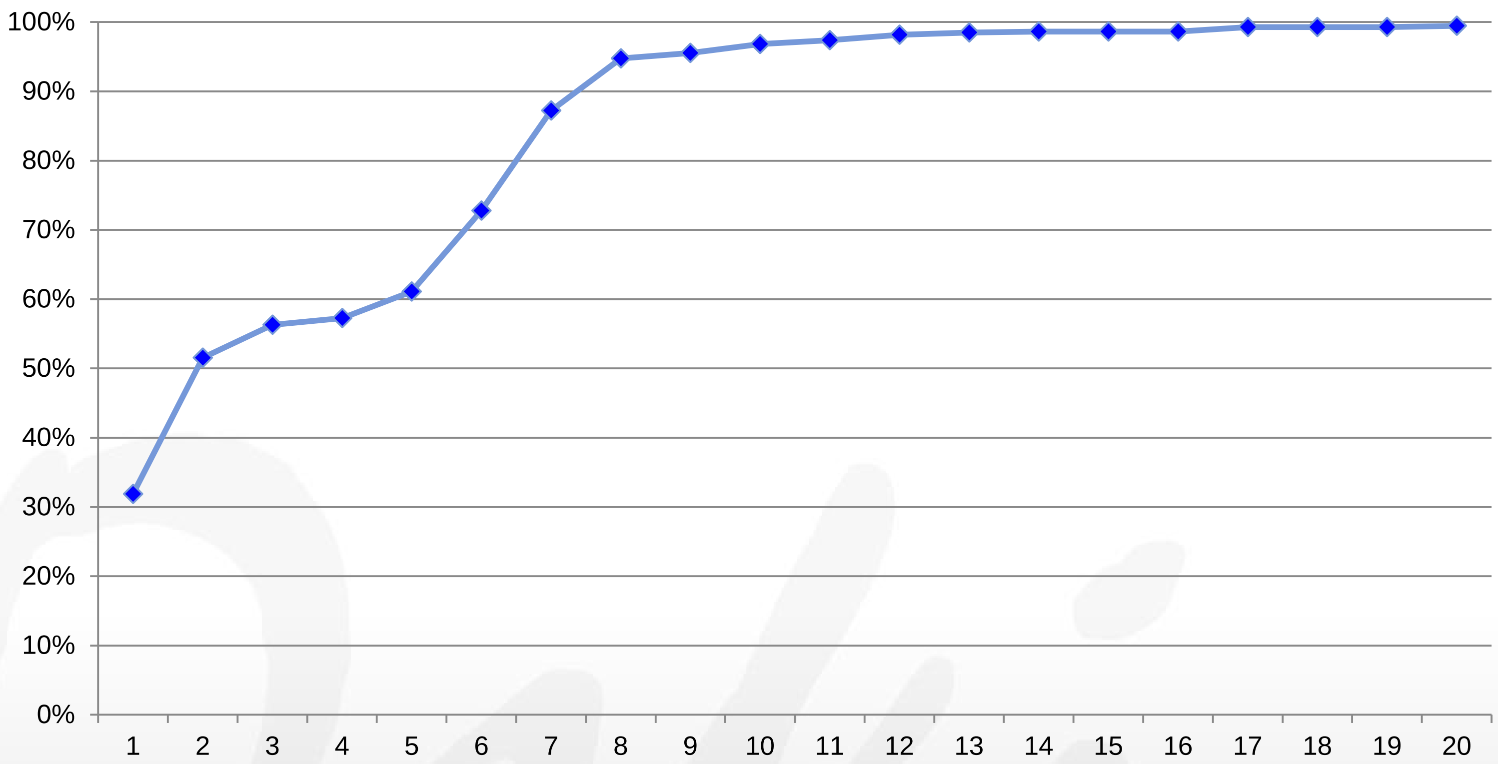
1. Latency analysis
   - Cover on output valid ports of arbiters (e.g. AWVALID)
     - Hits in 3 cycles
2. Micro-architectural Analysis
   - Analyzed depth of FIFOs and counters in Arbiters
   - Cover on simultaneous occurrence of FIFO full and counter empty condition
     - 13 cycles
3. Cover for interesting corner-cases
   - 29 cycles

- Proof bound unaffected by AXI4 support of 256 transfers
- Required proof bound – 29 cycles

Determining Required Proof Bound (2/3)

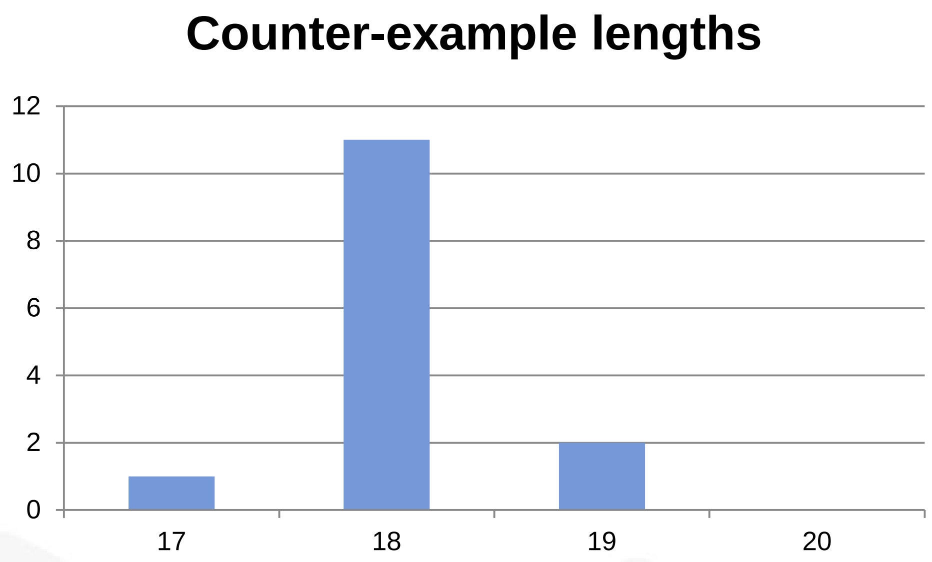
4. Formal Coverage

- 100% branch, expression and toggle coverage achieved in 20 cycles



Determining Required Proof Bound (3/3)

5. Failures seen during formal verification
   - No counter-example from 20th cycle onwards



- Deadlock checker reached 31 cycles with 12 hours of effort