Tackling the Complexity Problem in Control and Datapath Designs with Formal Verification

Ravindra Aneja
Synopsys

Ashish Darbari
Axiomise

Nitin Mhaske
Synopsys

Per Bjesse
Synopsys

Agenda

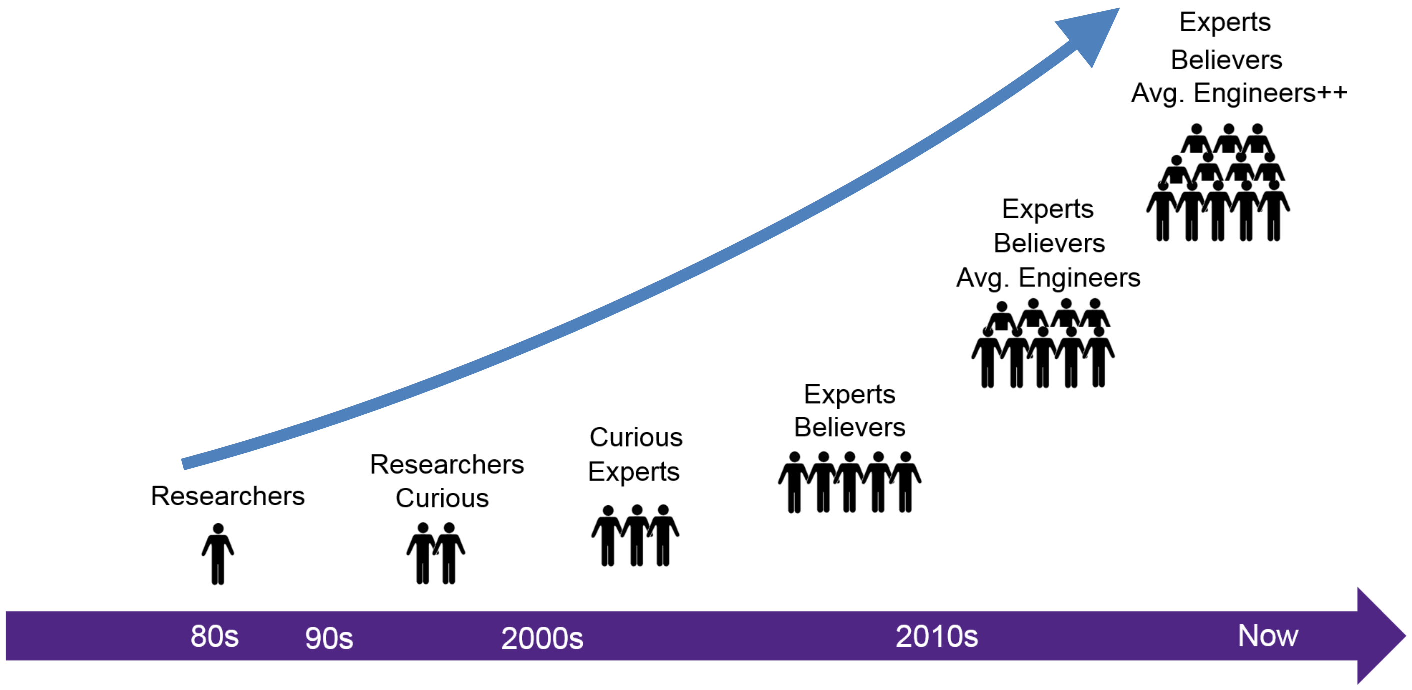
- Introduction
- Formal verification for SoC designs
- Formal verification for control paths
- Formal verification for data paths
- Summary

Current Formal Landscape

- Large number of companies are deploying formal
- Formal has become critical aspect of verification strategy
- Formal Apps are acting as catalyst

Formal Adoption

- Number of formal papers at DVCon/DAC/SNUG has gone up
- Number of users with formal expertise are growing
- Introduction of “Formal Signoff” flow is accelerating the adoption



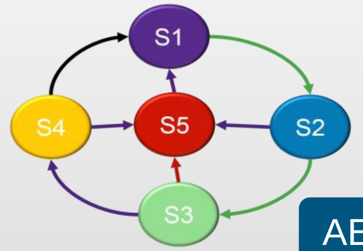
Formal is Central to Verification Strategy

- **Simulation cycles aren't scaling**
  - Need to look at each problem differently

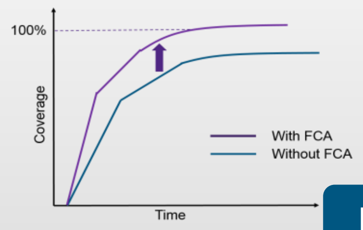
- **Let's break down the verification problem**
  - Verification plan consists of individual tasks
  - Some well suited for simulation
  - Some well suited for emulation
  - Some well suited for static/formal verification
  - Use the right task for the right problem

- **Consider multiple tools in the verification flow**
  - Not all problems can be solved by the same approach
  - Use the right tool for the right problem
    - Find bugs, saves time and $$$

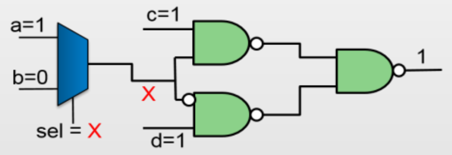
VC Formal Apps

Auto Checks
Functional Checks for RTL Structures


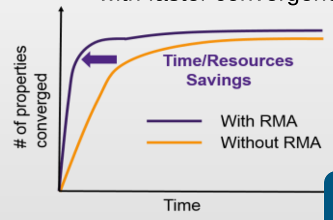
AEP

Formal Coverage Analyzer
Achieve Faster Coverage Closure


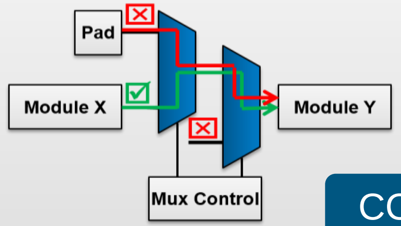
FCA

X-Propagation Verification
Detects Effects of "X"


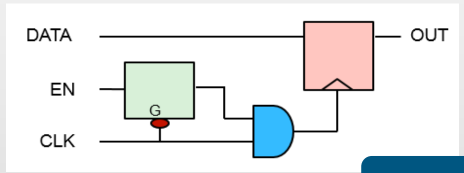
FXP

Regression Mode Accelerator
Increases verification throughput
with faster convergence


RMA

Connectivity Checking
Verify IP/SoC Connections


CC

Sequential Equivalence
Verify Clock gating and RTL optimizations


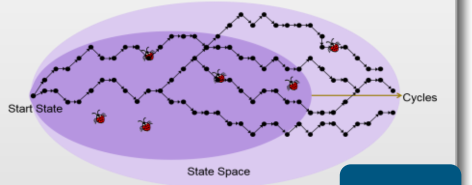
SEQ

Security Verification
Identify Data Leak/Integrity Issues

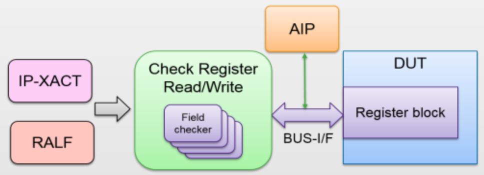
Secure Destination
Integrity Violation
OK
Unsecure Source
Secure Source
OK
Leak Violation
Unsecure Destination
FSV

Datapath Validation
Verify Datapath Designs
against the Specification

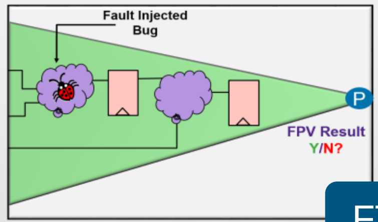
DPV

Property Verification
Verify User Defined Properties


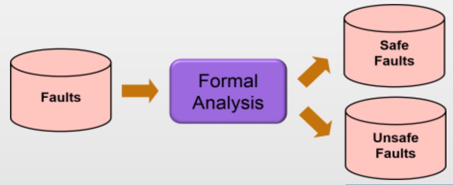
FPV

Register Verification
Verify Registers against IP-XACT/RALF


FRV

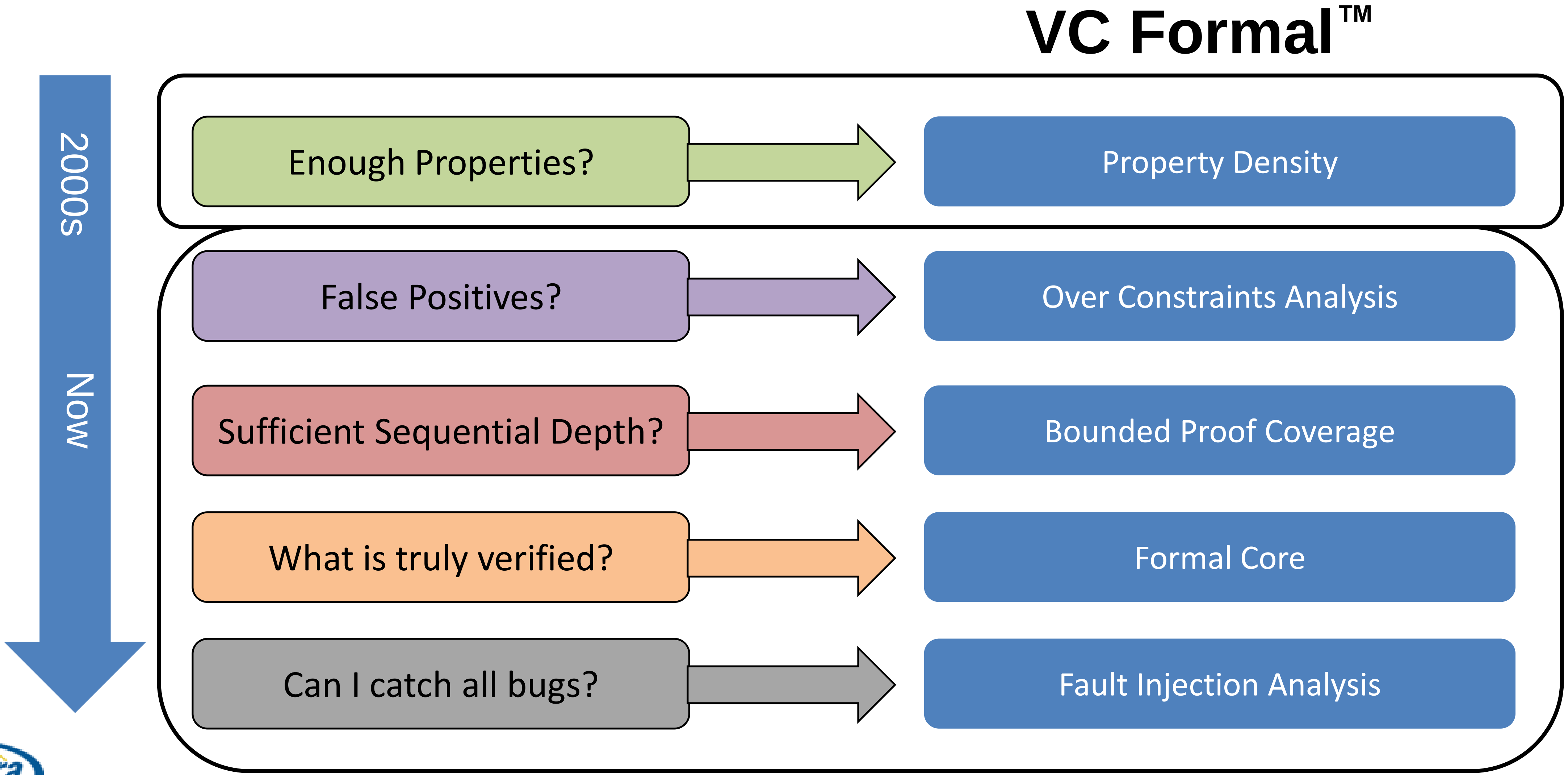
Formal Testbench Analyzer
Achieve Formal Signoff with Faults Analysis


FTA

Functional Safety Verification
Detectable and Diagnosable Faults


FuSa

Formal Signoff Criteria



Design/Verification Team’s Challenges

- Applicability
  - Class of verification problems?
  - Control path
  - Data path?
    - Data Transport, Data Transformation?
- Scalability
  - Module, block, subsystem level or chip level?
- Savings
  - Can we replace anything with Formal?
  - Can Formal compress overall verification time?
  - Can we do more with less resources?

This tutorial will answer some of these questions

Presenters: Ashish Darbari

Dr Ashish Darbari is the founder and CEO of Axiomise - a formal verification training, consulting, and services company.

Ashish obtained his DPhil from the University of Oxford in formal. Before starting Axiomise, Ashish worked at Intel, ARM, GM, Imagination Technologies, and OneSpin Solutions.

Ashish has been working in formal verification for over 20 years and has 18 patents and over two dozen research papers. Ashish is a Senior member of IEEE and ACM; and a Fellow of British Computing Society, and Fellow of IETE.

Presenters: Nitin Mhaske

Nitin Mhaske is a Senior Staff AE in VC Formal team with special interest in developing Apps that makes formal easy to apply and solve hard problems.

He has 18 years of experience in semiconductor and EDA companies. Prior to Synopsys, he was verification architect at Altera and Senior AE Manager at Atrenta for assertion based verification products. He holds 3 patents in assertion synthesis technology domain.

Presenters: Per Bjesse

Per Bjesse is a Synopsys Scientist with a PhD in Computer Science from Chalmers Technical University in Sweden.

Per has worked on formal verification at Synopsys for 15+ years on tools and applications ranging from equivalence checking, symbolic simulation, and software verification to standard model checking.

Per is the backend architect for all formal products in Synopsys Verification Group.

FORMAL VERIFICATION FOR SOC DESIGNS

OVERVIEW

SoC VERIFICATION

FORMAL VERIFICATION FLOW

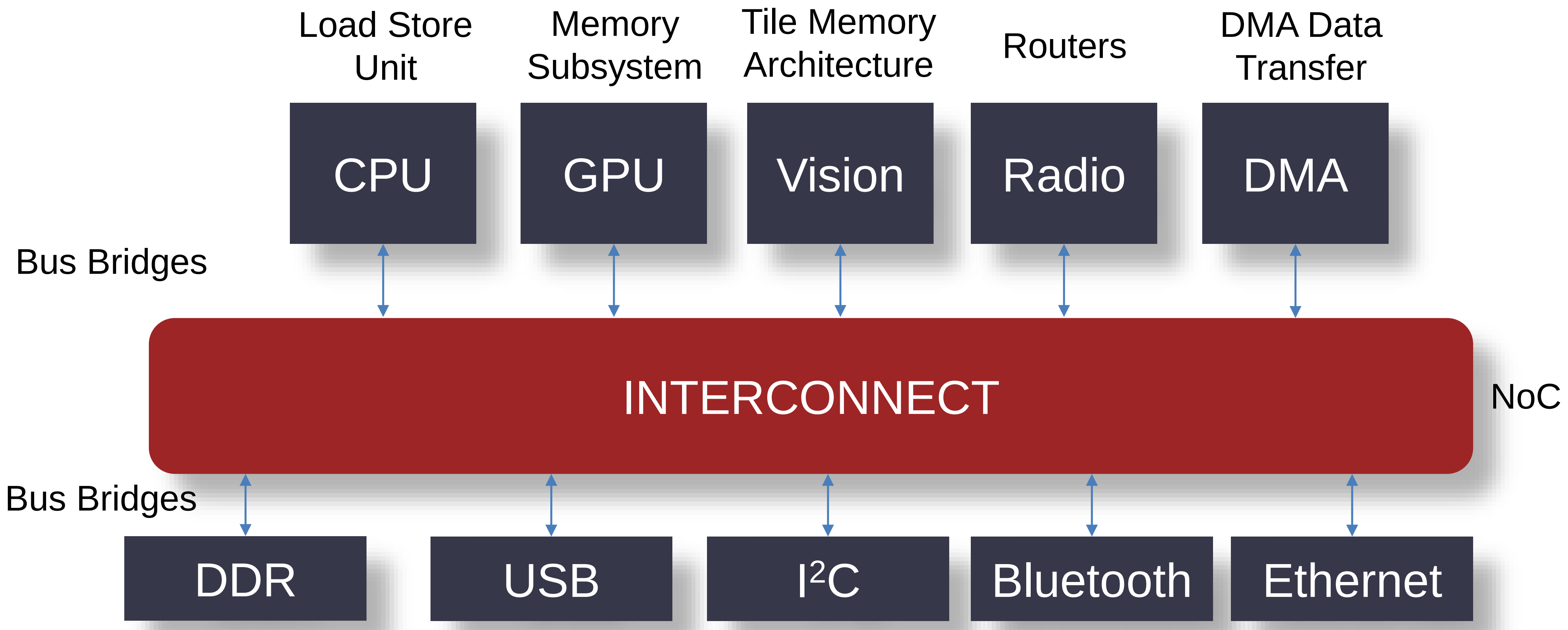
MODELING

SCALABILITY

SMART TRACKER

CASE STUDIES

SoC Architecture



Sequential designs are the root cause for verification complexity

SoC Verification

Verification

=

Design ⊨ Requirements

SoC verification

=

IP verification + Interfaces

Functional bugs

imply

security and safety bugs

SoC Verification Challenges

Functional

Safety

Security

Power

Performance

SoC Verification Challenges

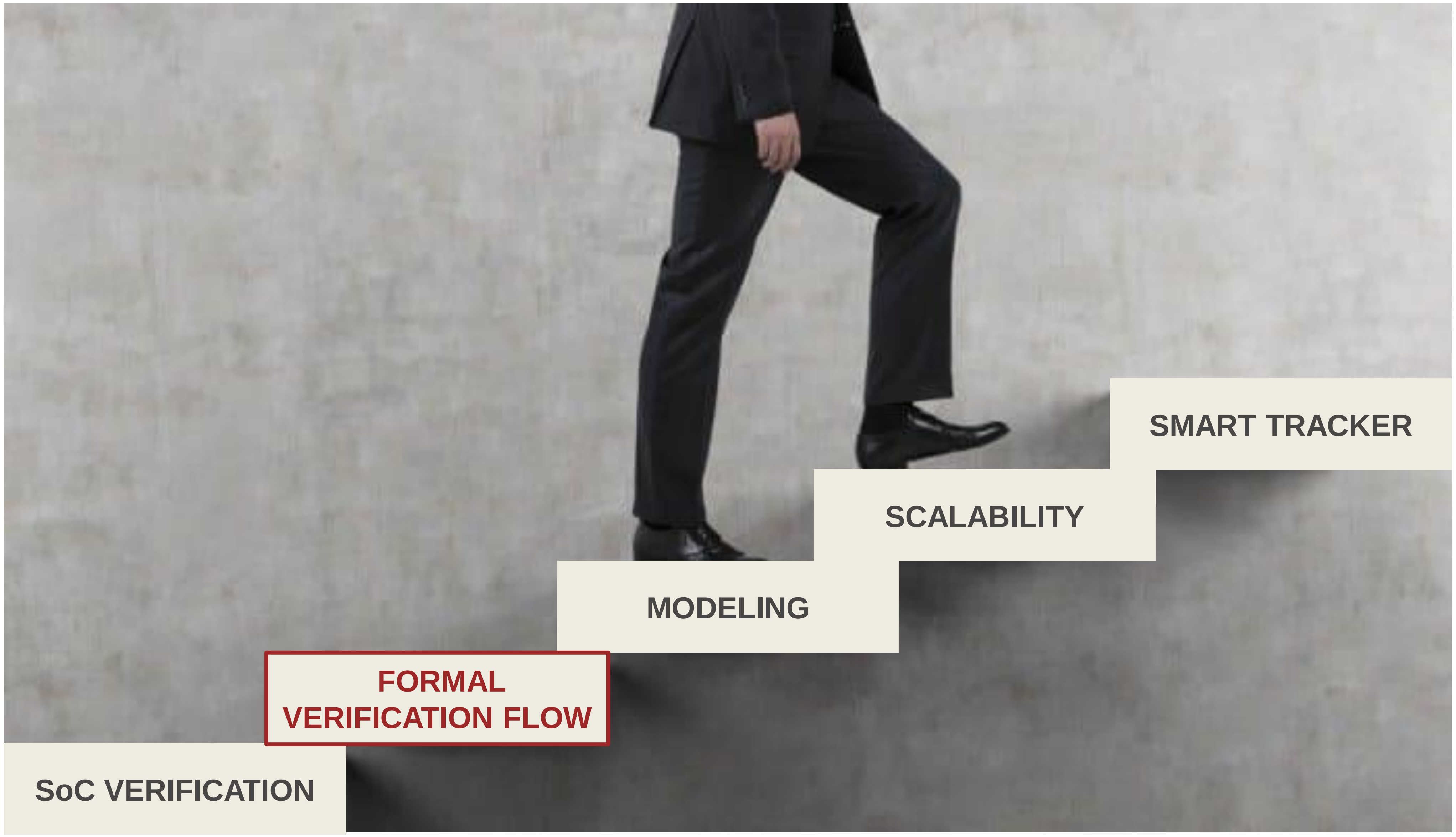
Clocks

Resets

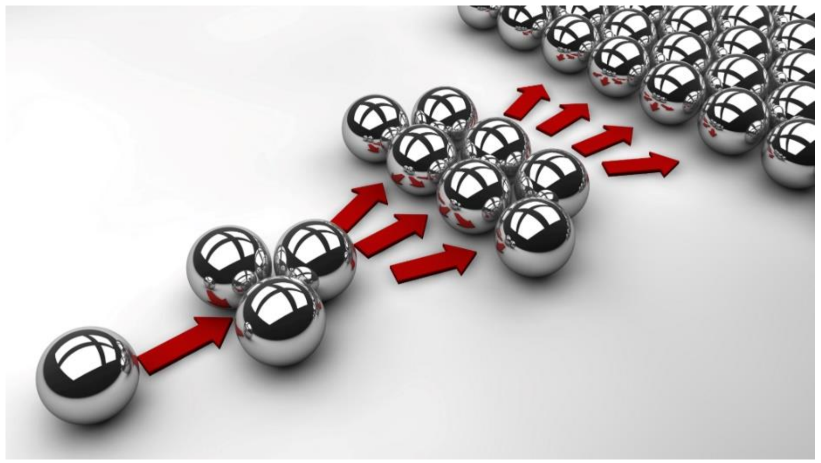
Timing

Synthesis

Layout


SMART TRACKER
SCALABILITY
MODELING
FORMAL VERIFICATION FLOW
SoC VERIFICATION

What is Formal Verification?



SPECIFICATION

Mathematical logic in specifying requirements

VERIFICATION

Verification is done by establishing a mathematical proof

Injecting Formal in the Verification Flow

VERIFICATION PLAN

APPS

USER-DEFINED PROPERTIES

COVERAGE

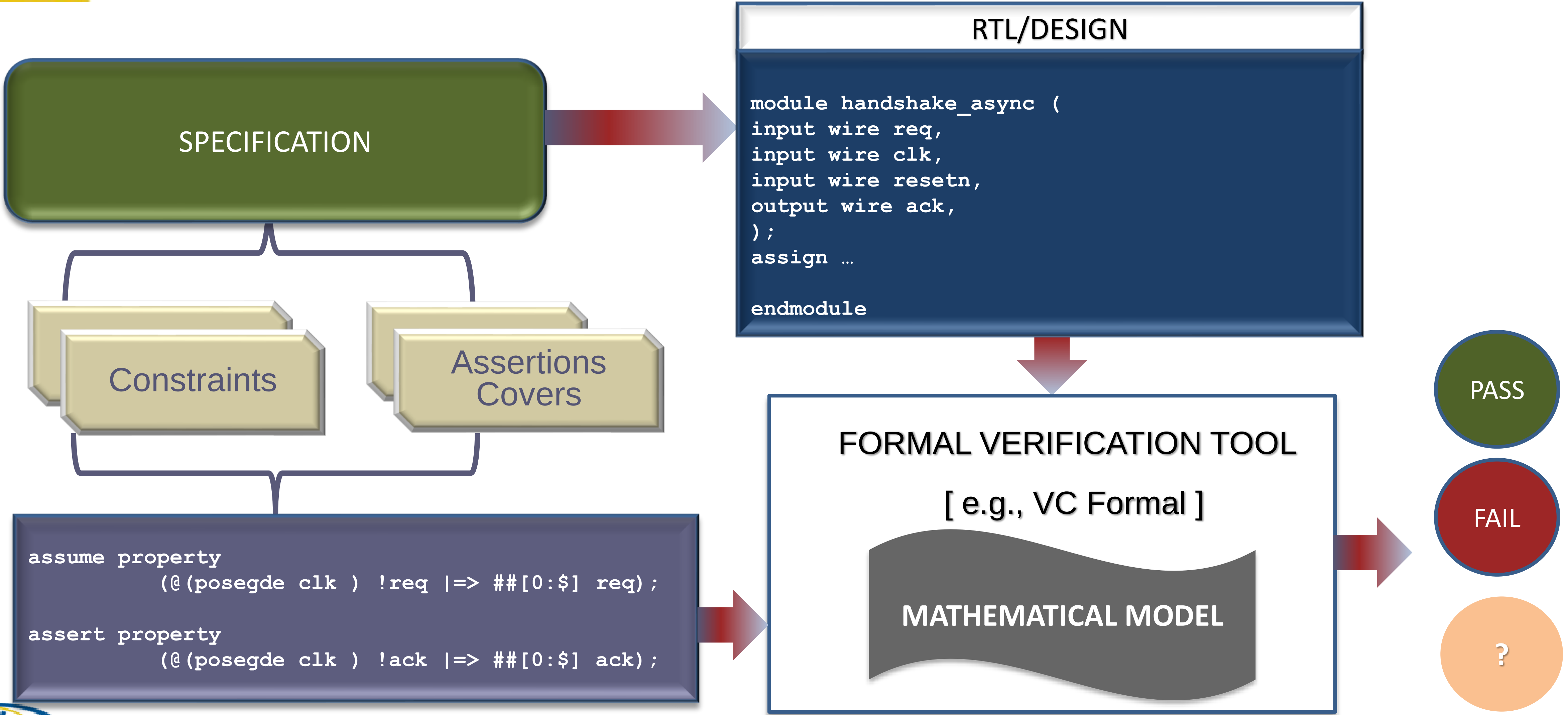
BACK ANNOTATION

Think IP think of interfaces

Think of requirements and specifications

Think of Properties (ABV)

Formal ABV in a Nutshell



```
module handshake_async (
input wire req,
input wire clk,
input wire resetn,
output wire ack,
);
assign …

endmodule
```

```
assume property
          (@(posegde clk ) !req |=> ##[0:$] req);

assert property
          (@(posegde clk ) !ack |=> ##[0:$] ack);
```

What Happens on a Pass?

✓ Property is true on all input combinations on all reachable states of the design

⚠ There are no over-constraints

What Happens on a Fail?

Bug in the design

Bug in the formalisation of the design intent (property formalisation bug)

Bug in the understanding of what the intent really is - the formalisation is correct, the intent is wrong

Missing constraint in specifying what are the legal values allowed on the inputs

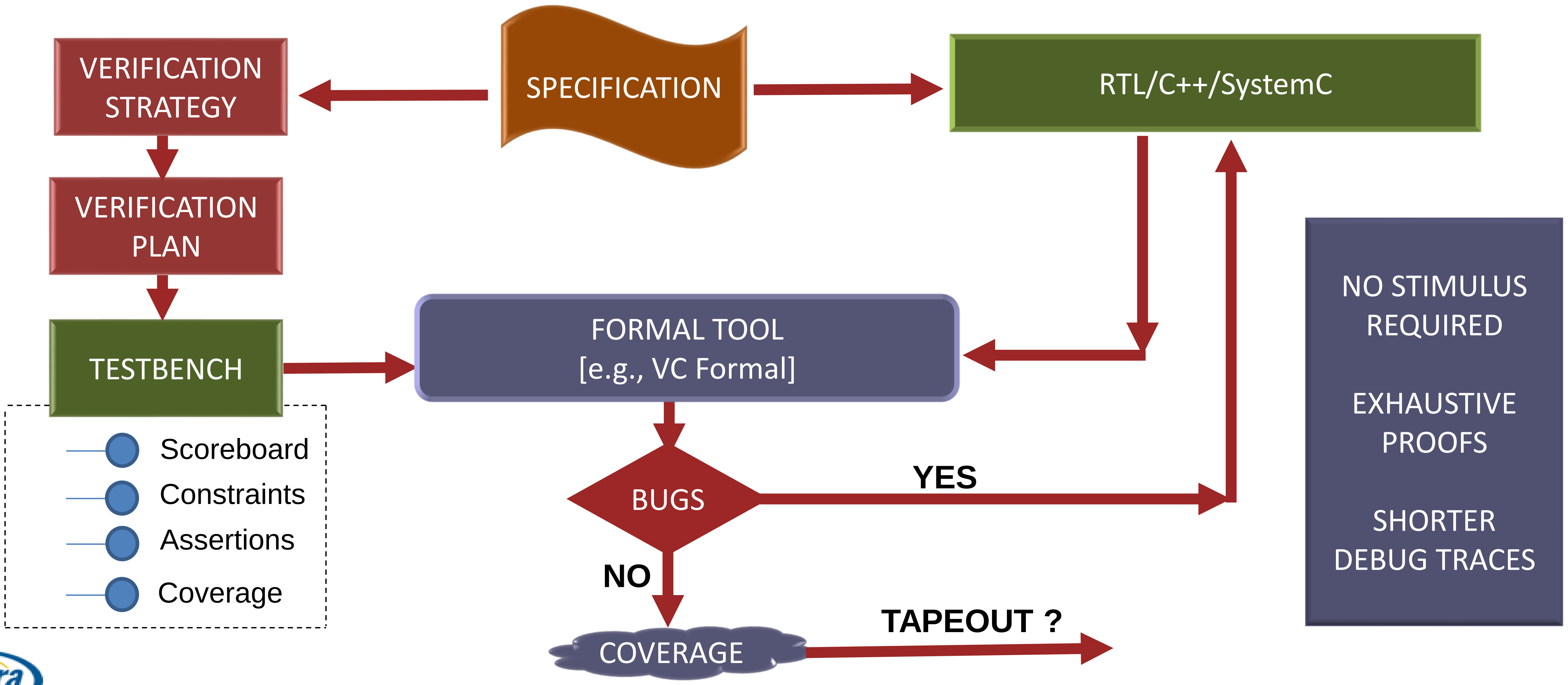
Remember with formal you get stimulus for free, so you need to ban the illegal stimulus

What Happens When You Get ?

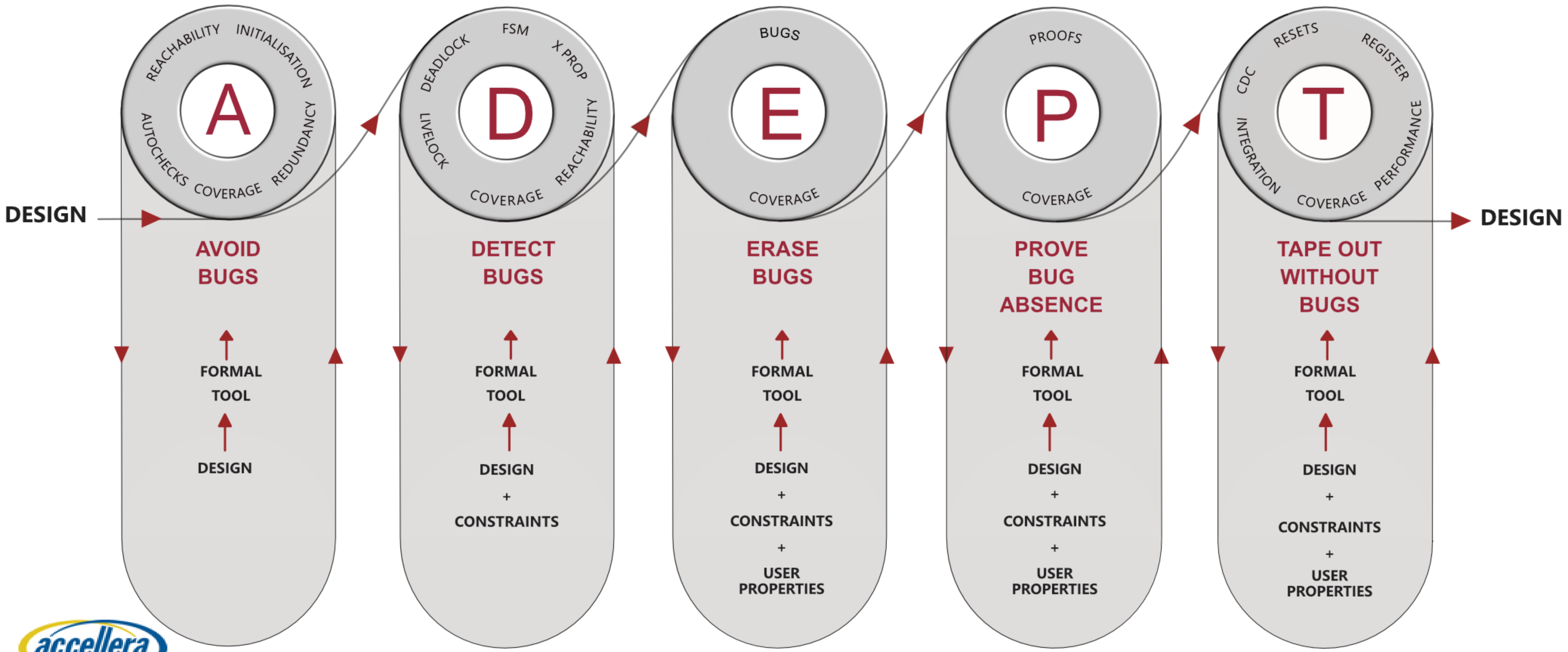
Review

- Constraints
- Assertions, covers
- Modelling code (glue logic)

Formal Verification Flow



The ADEPT FV® Agile Flow



Erase Bugs and Prove Absence

Formal Verification Testbench

- Abstractions
- Constraints

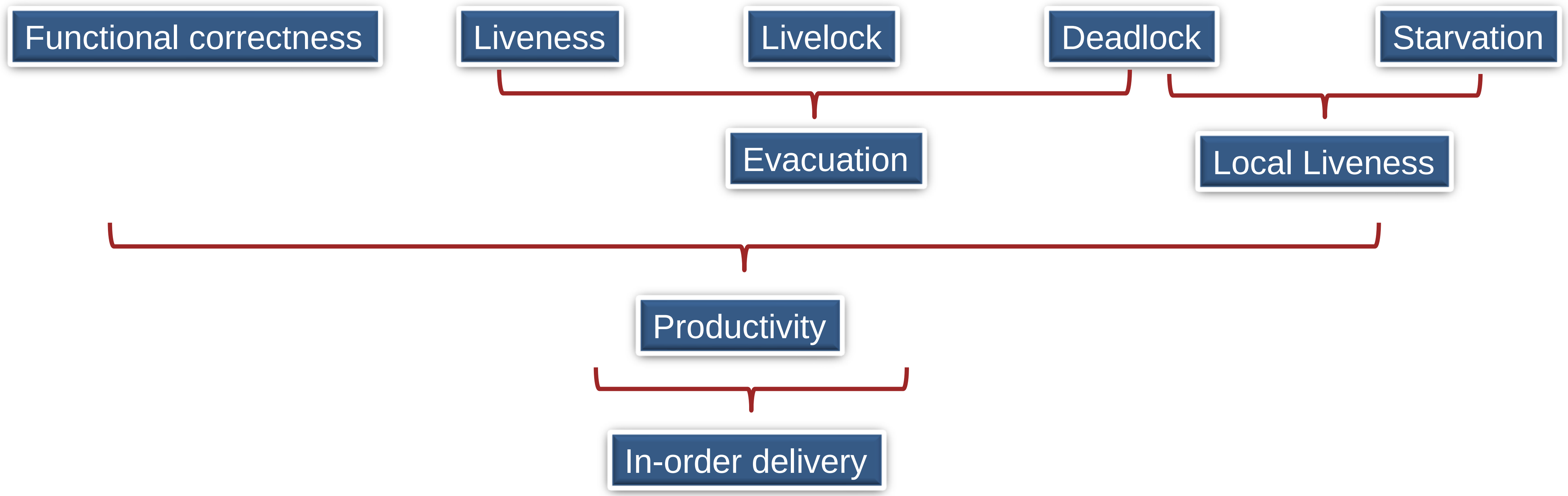
Erase bugs in both design and testbench

- Catch design bugs
- Catch testbench bugs
- Manual injection of bugs
- Run coverage analysis

Prove absence of bugs

- Invariants and assume guarantee
- Scalable results on bigger configs
- Run coverage analysis

Correctness Graph



In-order delivery is a stronger notion of correctness than productivity

Source: Formal Verification of On-Chip Communication Fabrics, Freek Verbeek, 2013, Radboud University

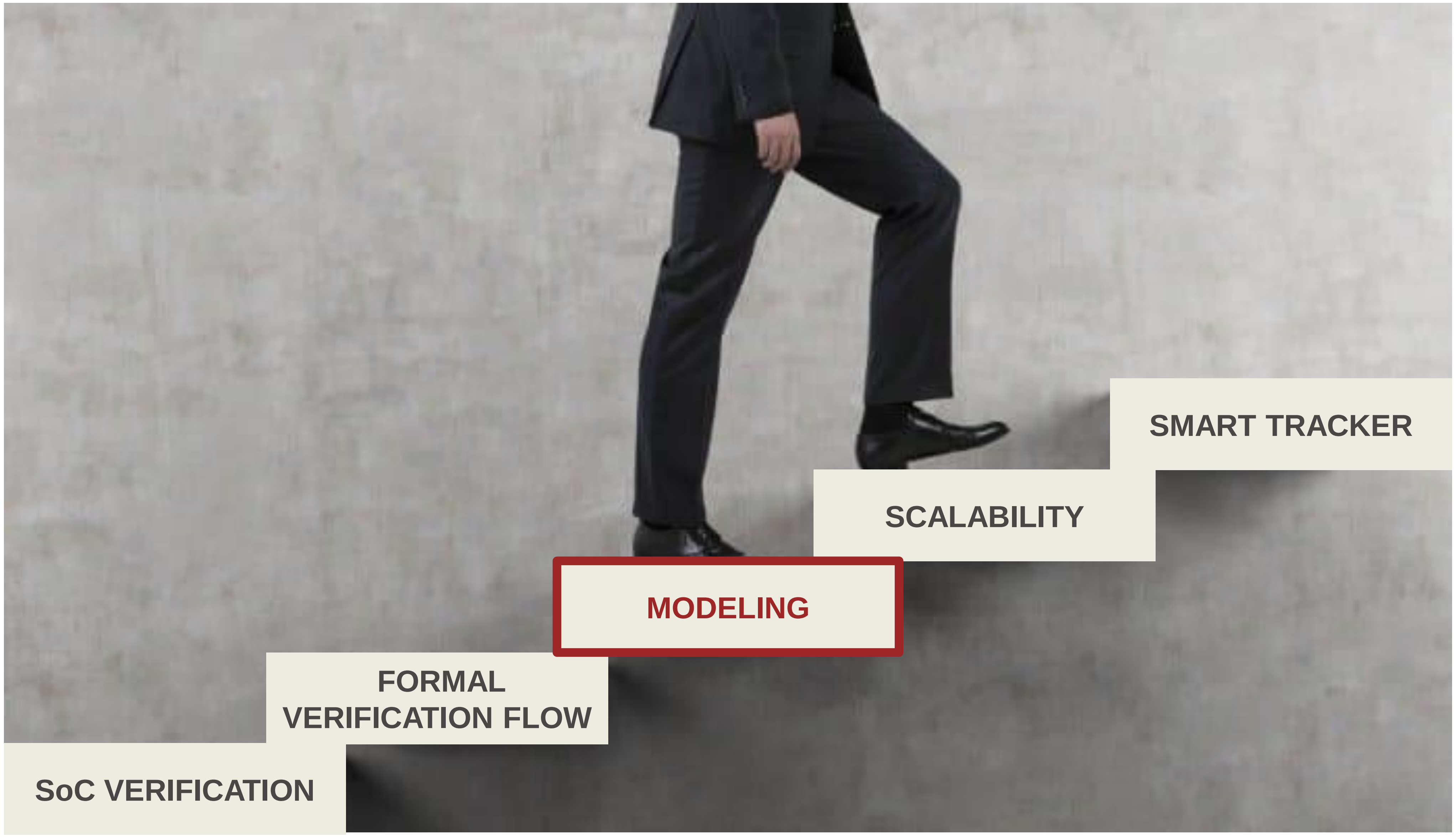
Verification Matters

Where possible proving in-order properties is sufficient to prove absence of

Liveness, Deadlock, Livelock and Starvation

Sources of Complexity

- **Control**
  - Serialisation
  - Arbitration
  - Counters
  - Arrays
  - FSMs
- **Datapath/Arithmetic**
  - Addition
  - Multiplication
  - Division
  - Multiplication/Accumulate
  - Square root


SMART TRACKER
SCALABILITY
MODELING
FORMAL VERIFICATION FLOW
SoC VERIFICATION

FORMAL MODELS

Building Blocks of Formal Testbenches

Events

- Formal models are constructed by capturing events
- Events are the right level at which we should think of verification
- An event is usually defined as some kind of asynchronous activity

Events

- Events have a start and a complete state

- We identify events by tagging them with a START and a STOP state

- Events come in two flavours
  - **WITHOUT** abstraction
  - **WITH** abstraction

- Events with abstraction provide reduction in proof complexity

Modeling Events

- The trick is to think transactional for our verification
- Leave the exact definition of a transaction somewhat abstract to begin with
- Refine it on a case-by-case basis
- The key here is to use symbolic transactions by exploiting abstractions

ABSTRACTIONS

The key to success with formal

Data Abstraction

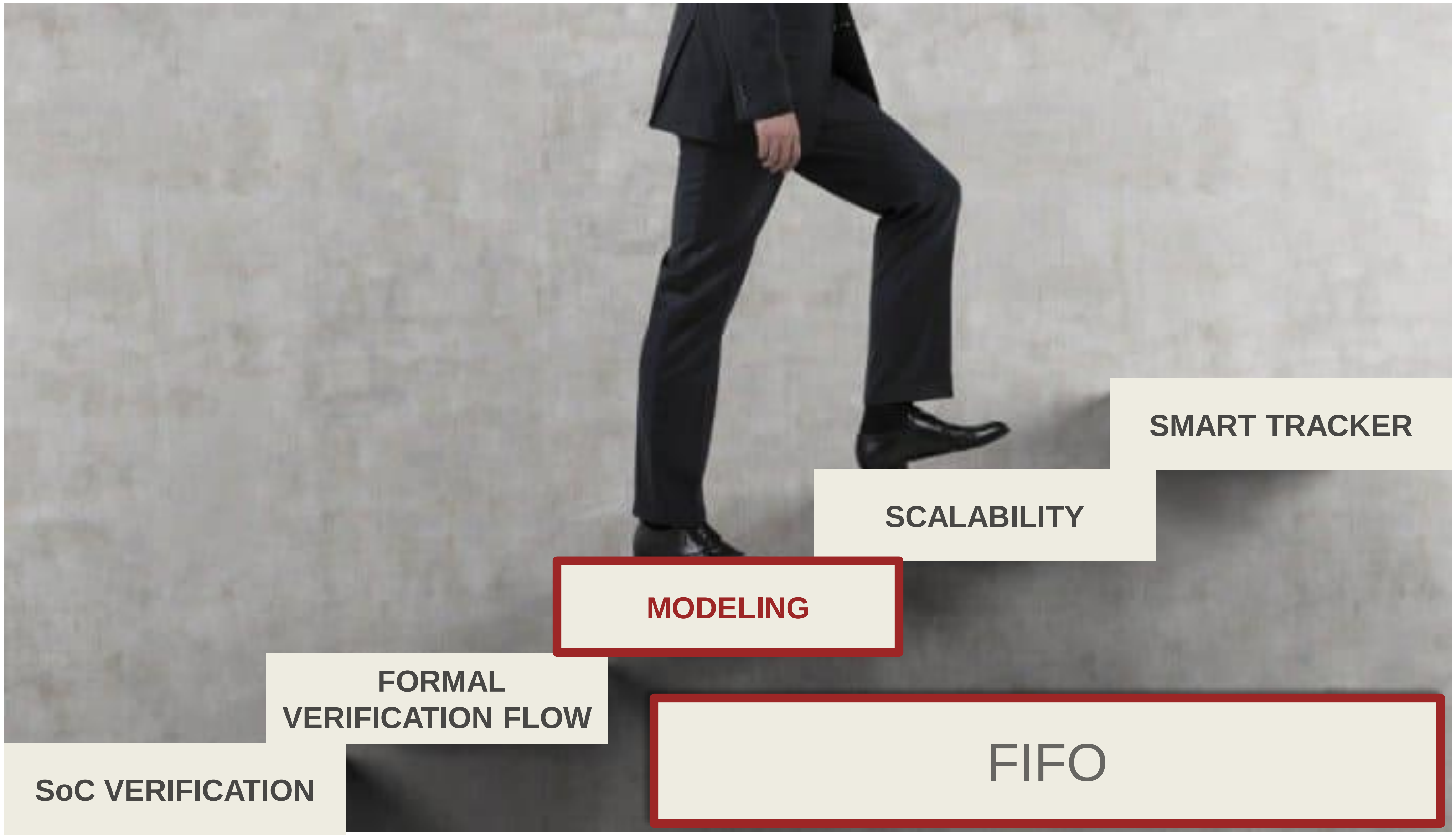
- Abstract away 0 and 1 by a new Boolean symbol
  - The result is a logarithmic reduction in state-space search

- You already use symbols without knowing
  - use of high level languages is already a symbolic step forward as
  - we don’t use truth tables for design

- Sometimes we also use an X value to get data abstraction

Temporal Abstraction

- Don’t sample the state of signals on every clock edge
- Sampling only occurring when certain key events are observed
- Use events to define “observation windows”
  - Each observation window has a start and a stop state
  - We can define multiple observation windows
  - Verification
    - Establish legal stimulus by providing constraints
    - Make claims [assert/cover] on these observation windows


SMART TRACKER
SCALABILITY
MODELING
FORMAL VERIFICATION FLOW
SoC VERIFICATION
FIFO

FIFO

Everything can be reduced to a FIFO!

Why FIFO?

- FIFOs everywhere:
  - Arbiters, UART, USB, CPUs, GPUs, Routers
- Introduces massive challenge for proof convergence
  - FIFOs introduce long latencies in other designs
  - Conceptually not very hard to understand
  - But easy to get it wrong
  - Can be extremely challenging to verify especially find corner case bugs
  - Async FIFOs

Verification Requirements

- Ordering is correct
- No duplication, No data loss, No data corruption
- Empty and Full checks
  - Empty at the right time
  - Full at the right time
  - If empty then eventually full
  - If full then eventually empty

Verification Strategy

- Build mechanisms to track data
- Provide any constraints or assumptions
- Write checks/assertions to establish “correctness always holds”
- Write cover properties to prove that behaviours can hold sometimes
- Ensure that you have not missed any bug in your test bench

Formal Verification Strategy

- We will not send any input sequences
- Let the formal tool exercise *“for free”* all input sequences
- Constrain out the illegal ones explicitly
- We track inputs going into the DUT and check if the expected ones come out
- In formal we use “symbols”
- Symbols encodes two values at once – one ‘0’ and another ‘1’
- Checking by formal tool is symbolic – covering all combinations of 0s and 1s

Formalizing Ordering

- For any **two data values** sent into a DUT in a pre-determined order, if they exit the DUT in the same order as they were sent in, then the DUT maintains ordering on the elements

- For any **two "symbolic" values** sent into a DUT in a pre-determined order, if they exit the DUT in the same order as they were sent in, then the DUT maintains ordering on the elements

$$\forall d_1\ d_2.\ (d_1\ \mathit{sampled\_in\_before}\ d_2) \implies (d_1\ \mathit{sampled\_out\_before}\ d_2)$$

Symbolic Transactions

```
logic [DATA_WIDTH-1:0] wd1;
logic [DATA_WIDTH-1:0] wd2;

am_fifo_core_d1_stable:
      assume property (@(posedge clk) ##1 $stable(wd1));

am_fifo_core_d2_stable:
      assume property (@(posedge clk) ##1 $stable(wd2));
```

Sampling Registers

SAMPLING IN

```
reg sampled_i_1;                 wire ready_to_sampled_i_1;
reg sampled_i_2;                 wire ready_to_sampled_i_2;
```

SAMPLING OUT

```
reg sampled_o_1;                 wire ready_to_sampled_o_1;
reg sampled_o_2;                 wire ready_to_sampled_o_2;
```

Watching In and Out

WATCHING WHEN TO SAMPLE IN

```
assign ready_to_sample_i_1 = data_i==wd1 && push_i && !sampled_i_1 &&
                             arbit_window;

assign ready_to_sample_i_2 = data_i==wd2 && push_i && !sampled_i_2 &&
                             arbit_window;
```

WATCHING WHEN TO SAMPLE OUT

```
assign ready_to_sample_o_1 =  sampled_i_1   && data_o==wd1 && pop_i &&
                              !sampled_o_1;

assign ready_to_sample_o_2 =  sampled_in_d2 && data_o==wd2 && pop_i &&
                              !sampled_o_2;
```

Events

SAMPLING REGISTERS

```
always @(posedge clk or negedge resetn)
    if (!resetn) begin
        sampled_i_1  <= 1'b0;
        sampled_o_1  <= 1'b0;
        sampled_i_2  <= 1'b0;
        sampled_o_2  <= 1'b0;
    end
    else begin
        sampled_i_1  <= sampled_i_1  || ready_to_sample_i_1;
        sampled_i_2  <= sampled_i_2  || ready_to_sample_i_2;
        sampled_o_1  <= sampled_o_1  || ready_to_sample_o_1;
        sampled_o_2  <= sampled_o_2  || ready_to_sample_o_2;
    end
```

Putting it altogether

INTERFACE CONSTRAINTS

```
assume property (@(posedge clk) empty_o |-> !pop_i);

assume property (@(posedge clk) full_o  |-> (!push_i || pop_i));
```

TESTBENCH CONSTRAINT

```
assume property (@(posedge clk) !sampled_i_1  |->  !sampled_i_2);
```

MASTER CHECK

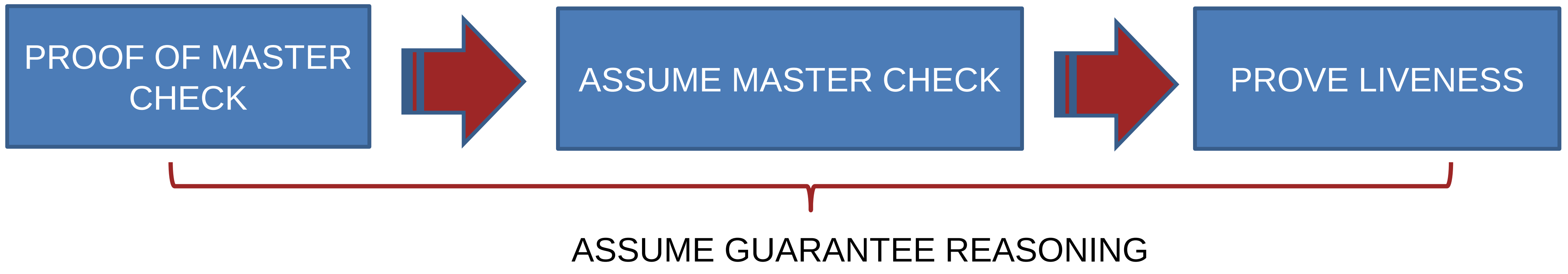
```
assert property (@(posedge clk) sampled_i_1 && sampled_i_2 && !sampled_o_1
                                  |->
                                 !sampled_o_2);
```

Liveness

LIVENESS

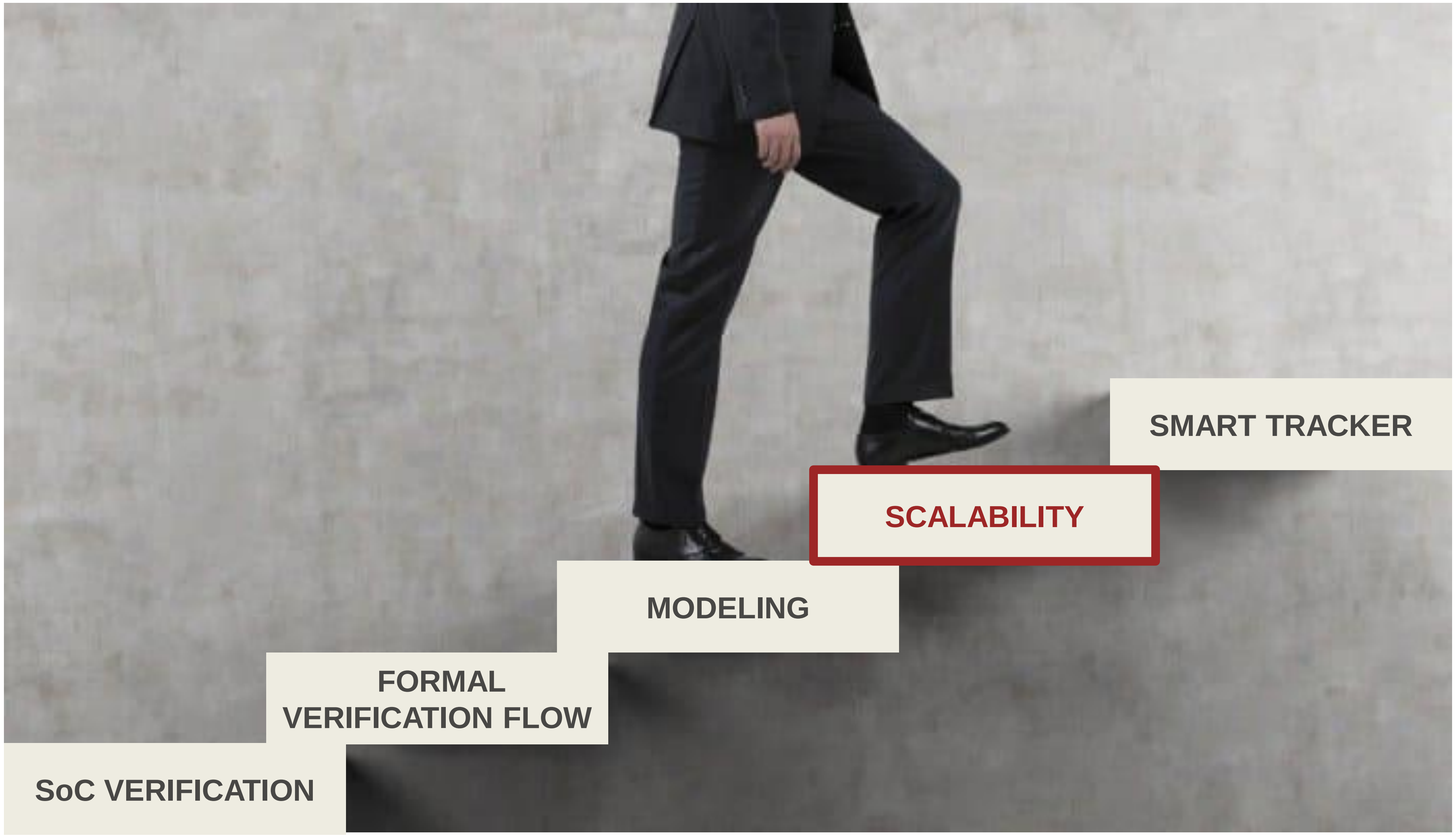
```
assume property (@(posedge clk)!pop_i |-> s_eventually (pop_i));

assert property (@(posedge clk) sampled_i_1 |-> s_eventually (sampled_o_1));
```



Analysis

- Only four registers used to model an end-to-end master check that verifies
  - Ordering
  - Data loss
  - Data duplication
  - Data corruption
- Proving then assuming the master check establishes liveness
- But there is a challenge
  - As depth increases the results degrade
- Scalability is limited


SMART TRACKER
SCALABILITY
MODELING
FORMAL VERIFICATION FLOW
SoC VERIFICATION

Proof Engineering

Scalable formal verification

=

"Proof Engineering"

Assume Guarantee

Case Splitting

Scenario Splitting

$10^{120\ \text{MILLION}}$ states; 1 billion gates and beyond

ASSUME GUARANTEE

You assume I guarantee! I assume you guarantee!!

Assume Guarantee

- Break the whole puzzle into smaller jigsaws
- Identify helper lemmas as individual components of jigsaw
- Identify how they fit together to complete the full puzzle
- PROVE helper lemmas then ASSUME them to prove other lemmas

Too Few States

- Current solution has **too few** states
- Exploit “data independence”
  - It does not matter what the specific data values are
  - It only matters how many are ahead of the watched data value
  - We will exploit this “how many are ahead” by introducing more states
  - **Yes, we bring in counters to improve performance for proof convergence!**

Increment/Decrement

INCREMENT

```
assign incr_1     = push_i && !sampled_i_1;

assign incr_2     = push_i && !sampled_i_2;
```

DECREMENT

```
assign decr_1     = pop_i  && !sampled_o_1;

assign decr_2     = pop_i  && !sampled_o_2;
```

How Many are Ahead?

TRACKING COUNTERS

```
always @(posedge clk or negedge resetn)
        if (!resetn)  begin
                tracking_counter_1 <= 'h0;
                tracking_counter_2 <= 'h0;
         end
        else begin
                tracking_counter_1 <= tracking_counter_1 + incr_1 - decr_1;
                tracking_counter_2 <= tracking_counter_2 + incr_2 - decr_2;
         end
```

Putting it altogether

INTERFACE CONSTRAINTS

```
assume property (@(posedge clk) empty_o |-> !pop_i);

assume property (@(posedge clk) full_o  |-> (!push_i || pop_i));
```

TESTBENCH CONSTRAINT

```
assume property (@(posedge clk) !sampled_i_1  |->  !sampled_i_2);
```

MASTER CHECK

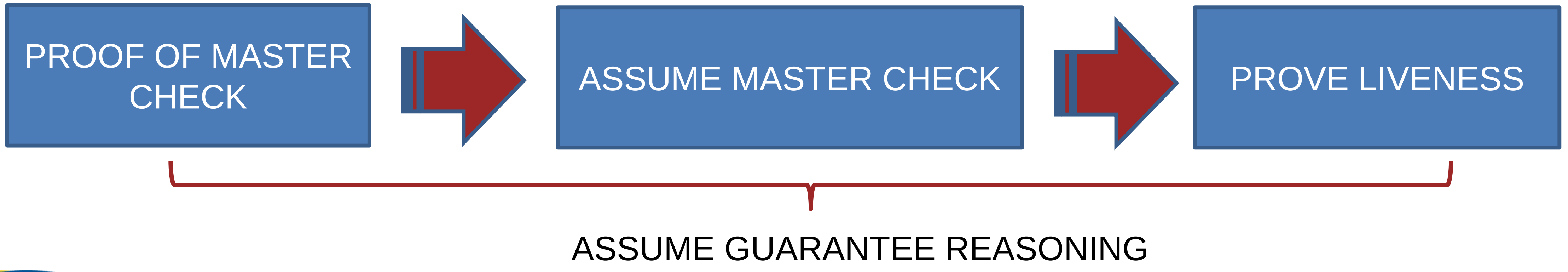
```
assert property (@(posedge clk) sampled_i_1 && sampled_i_2 && !sampled_o_1
                                  |->
                                !sampled_o_2);
```

Liveness

LIVENESS

```
assume property (@(posedge clk)!pop_i |-> s_eventually(pop_i));

assert property (@(posedge clk) sampled_i_1 |-> s_eventually (sampled_o_1));
```



Invariants and Assume Guarantee

Positional Invariant tells the tool where exactly in the DUT the tracked value is

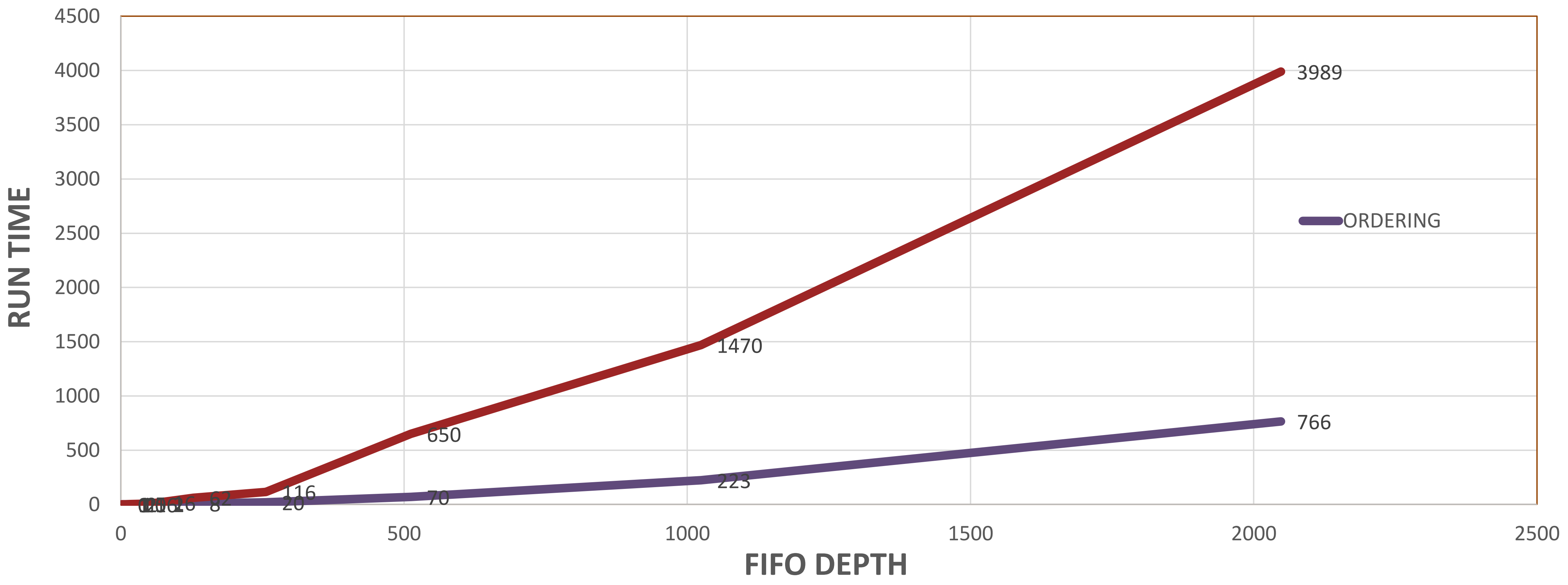
If watched value is in the DUT then the tracking counter is in between the read and the write pointers

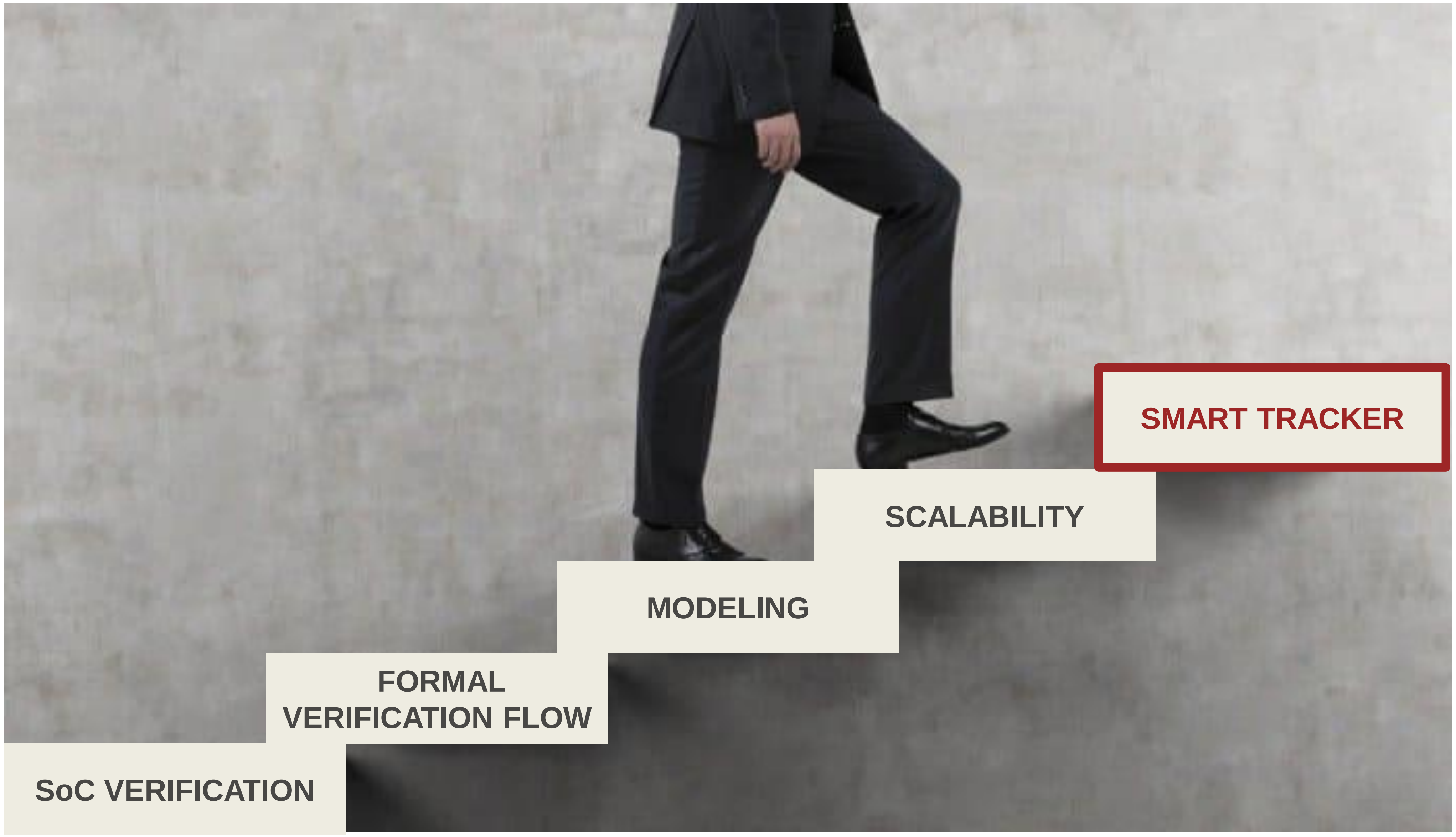
If the watched value not in the DUT then the counts between the DUT and the abstract model agree

If the watched value has not entered in the DUT then it couldn’t have left it

Once the tracking value has entered and exited the DUT then counters agree

Performance Results




SMART TRACKER
SCALABILITY
MODELING
FORMAL VERIFICATION FLOW
SoC VERIFICATION

Basic Concept

- Use one symbolic watched data value to track
- A single counter to count how many values are “ahead” of this watched data
- On every push, this counter increments, and every pop it decrements
- Once the watched data is seen on the inputs, the counter is not incremented
- When the counter is 1, expect to see the watched value appear on the output

Symbolic Transaction

```
logic [DATA_WIDTH-1:0] wd;

am_fifo_core_d1_stable:
        assume property (@(posedge clk) ##1 $stable(wd));
```

SAMPLING IN

```
reg sampled_i;                                          wire ready_to_sample_i;
```

SAMPLING OUT

```
reg sampled_o;                                          wire ready_to_sample_o;
```

Events

WATCHING WHEN TO SAMPLE IN

```
assign ready_to_sample_i = data_i==wd && incr && arbit_window;
```

WATCHING WHEN TO SAMPLE OUT

```
assign ready_to_sample_o = (tracking_counter == 1)&& sampled_i && decr;
```

SAMPLING REGISTERS

```
always @(posedge clk or negedge resetn)
  if (!resetn) begin
     sampled_i  <= 1'b0;
     sampled_o  <= 1’b0; end
  else begin
     sampled_i  <= sampled_i  || ready_to_sample_i;
     sampled_o  <= sampled_o  || ready_to_sample_o; end
```

Increment/Decrement

INCREMENT

```
assign incr    =    push_i && !sampled_i;
```

DECREMENT

```
assign decr    =    pop_i  && !sampled_o;
```

How Many are Ahead?

TRACKING COUNTER

```
always @(posedge clk or negedge resetn)
      if (!resetn)  begin
            tracking_counter <= ‘h0;
       end
      else begin
            tracking_counter <= tracking_counter + incr - decr;
      end
```

Putting it Altogether

INTERFACE CONSTRAINTS

```
assume property (@(posedge clk) empty_o |-> !pop_i);

assume property (@(posedge clk) full_o  |-> (!push_i || pop_i));
```

MASTER CHECK

```
assert property  (@(posedge clk) ready_to_sample_o |-> data_o==wd);
```

Invariants and Assume Guarantee

Positional Invariant tells the tool where exactly in the DUT the tracked value is

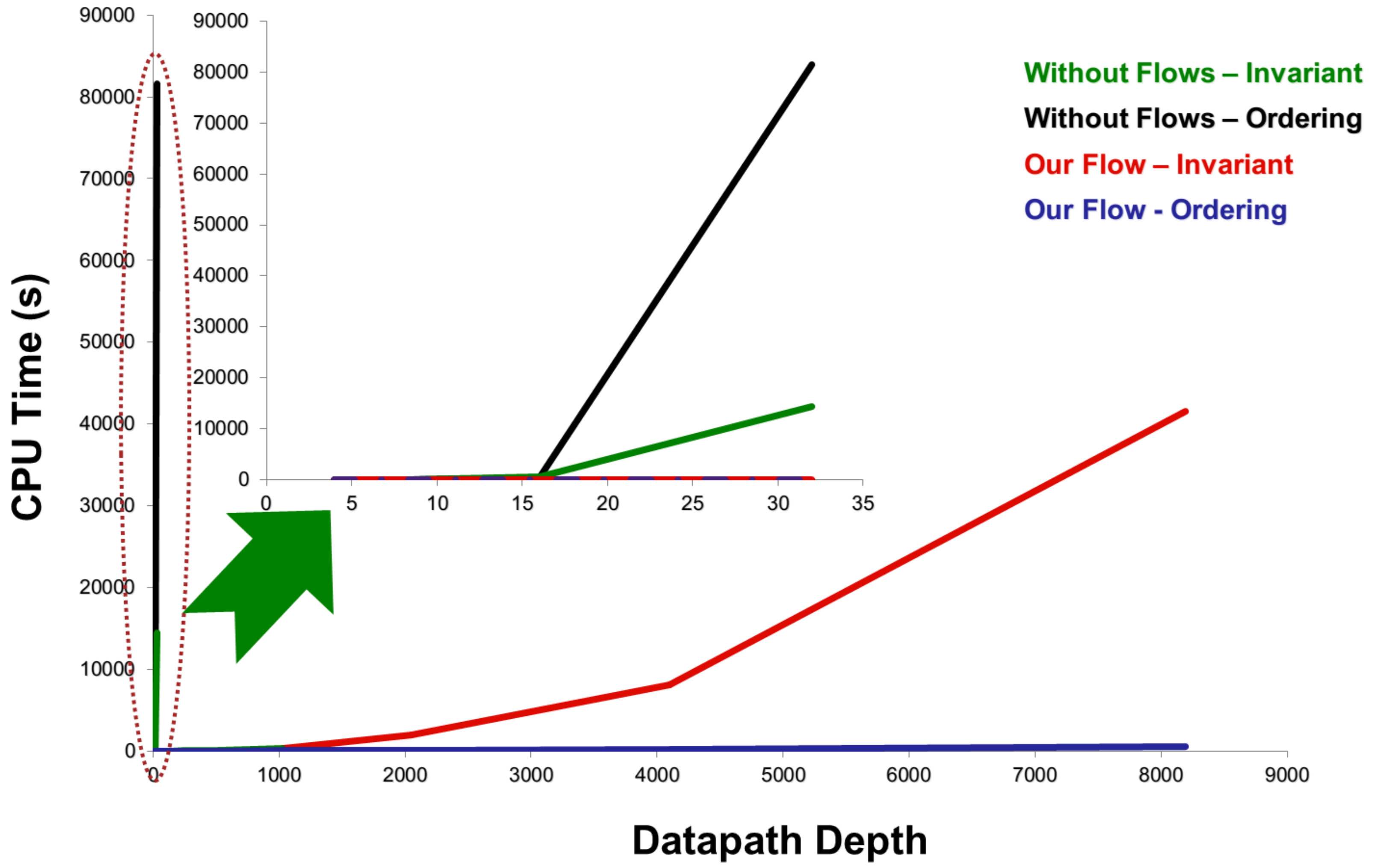
If watched value is in the DUT then the tracking counter is in between the read and the write pointers

If the watched value not in the DUT then the counts between the DUT and the abstract model agree

If the watched value has not entered in the DUT then it couldn’t have left it

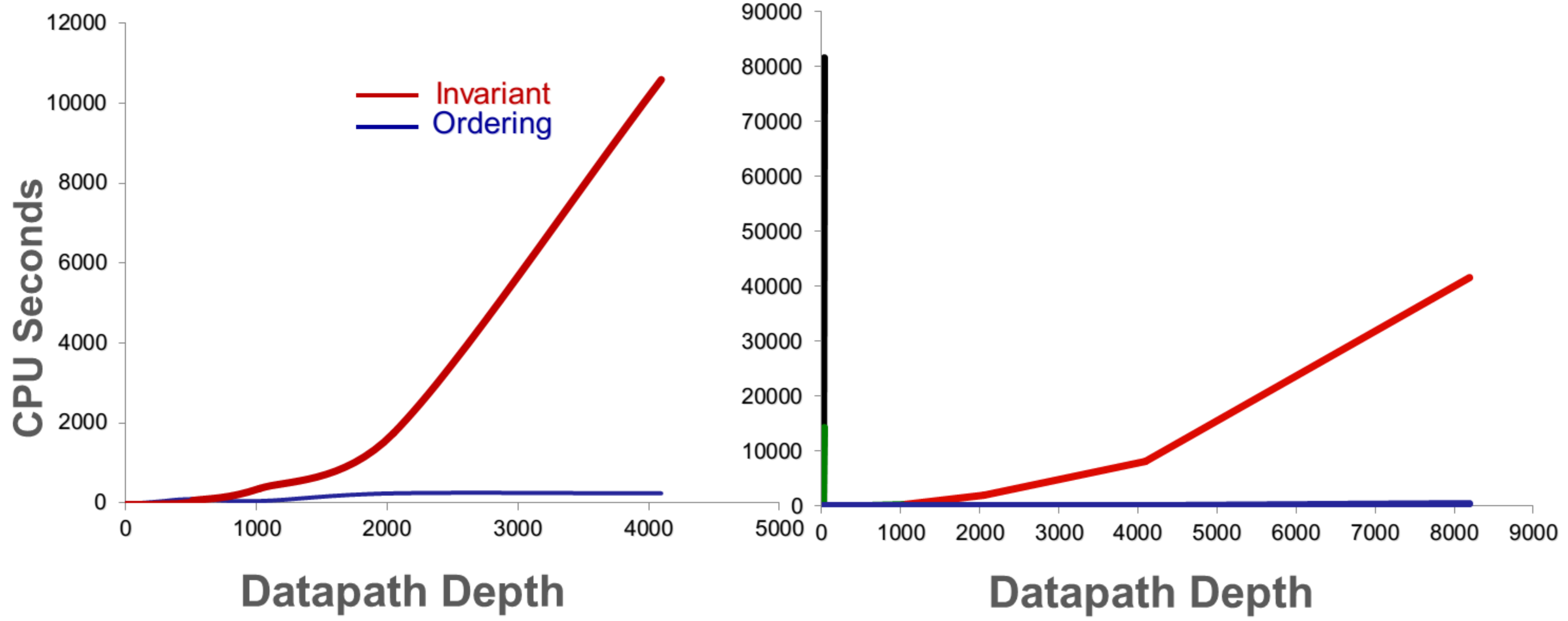
Once the tracking value has entered and exited the DUT then counters agree

Smart Tracker Performance



Source: More is Less: Exhaustive Formal Verification of Sequentially Deep Data-Transport Components, Darbari et al. DAC 2014

Two Transaction vs Smart Tracker



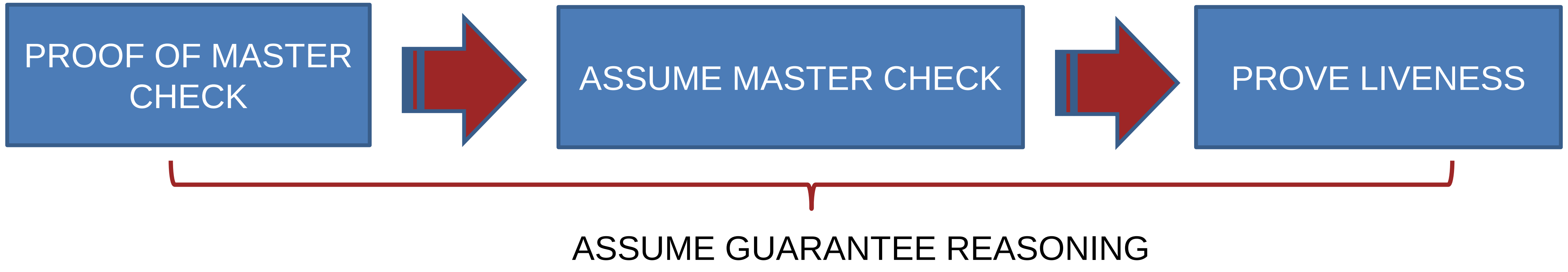
Source: Industrial Strength Formal Using Abstractions, A. Darbari & I. Singleton, Tech Report, 2016, Available from arxiv.org

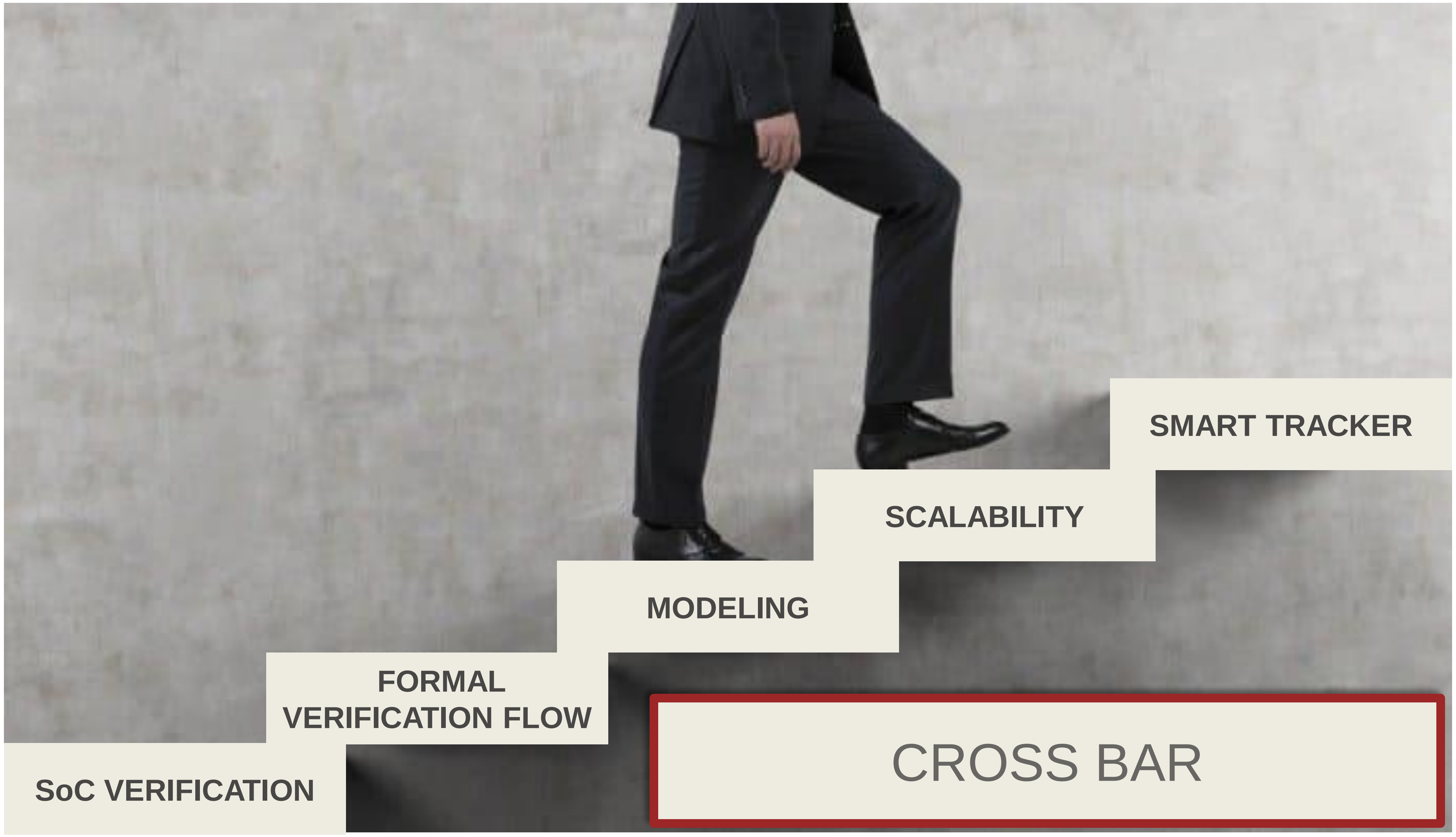
Liveness

LIVENESS

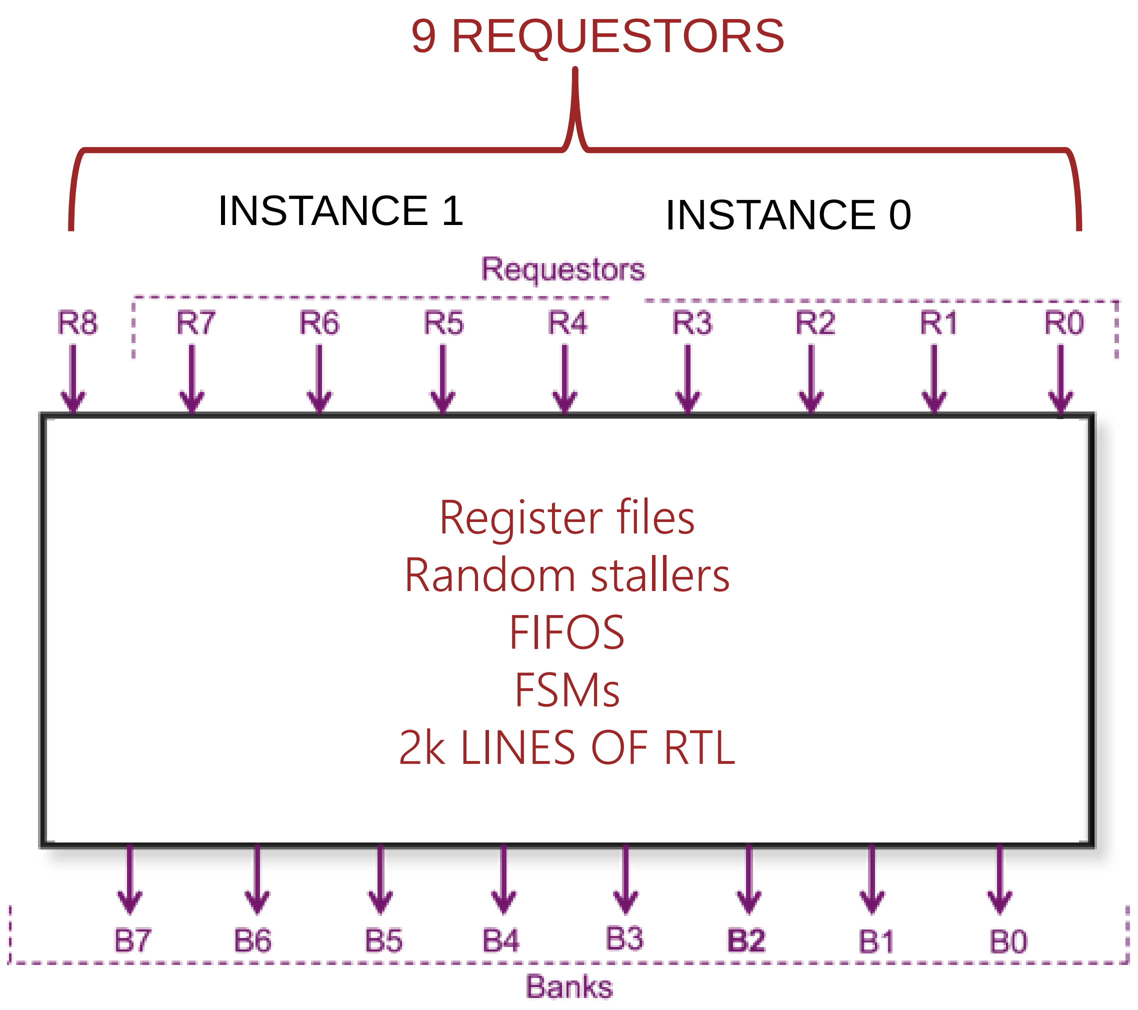
```
assume property (@(posedge clk)!pop_i |-> s_eventually (pop_i));

assert property (@(posedge clk) sampled_i |->  s_eventually (sampled_o));
```




SMART TRACKER
SCALABILITY
MODELING
FORMAL VERIFICATION FLOW
SoC VERIFICATION
CROSS BAR

Memory Subsystem Arbiter



4 GROUPS with 2 BANKS in each group

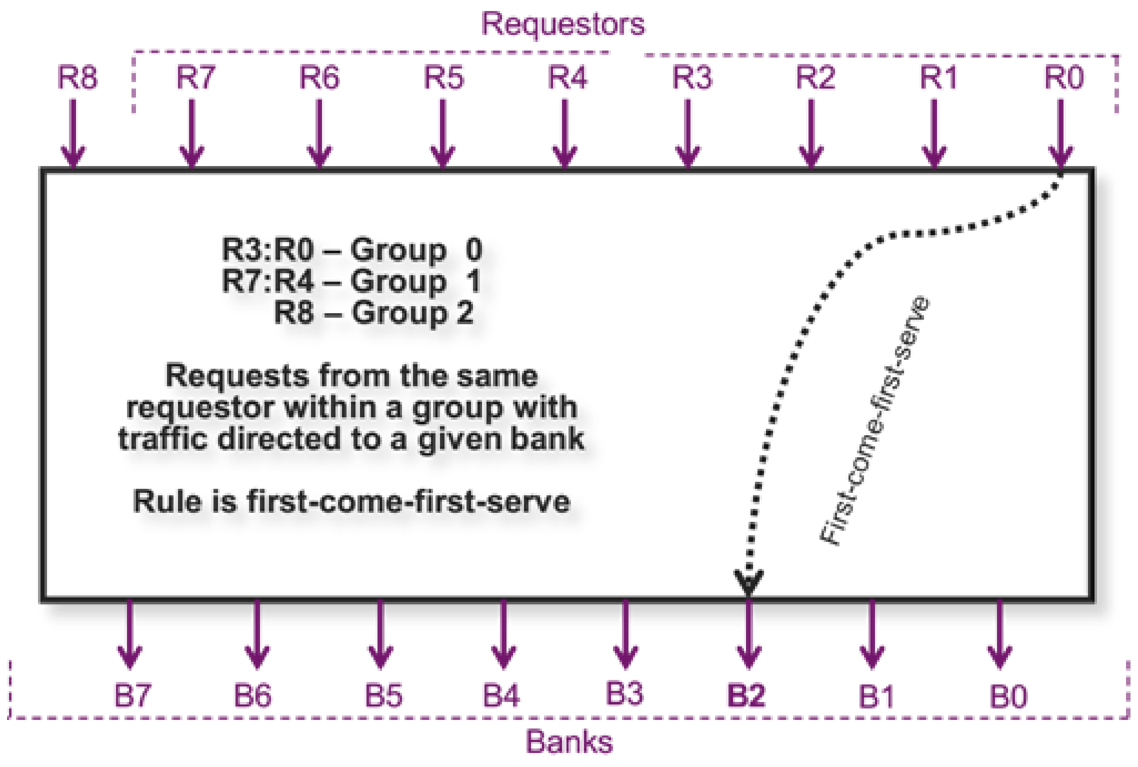
Verification Challenge

Concurrency

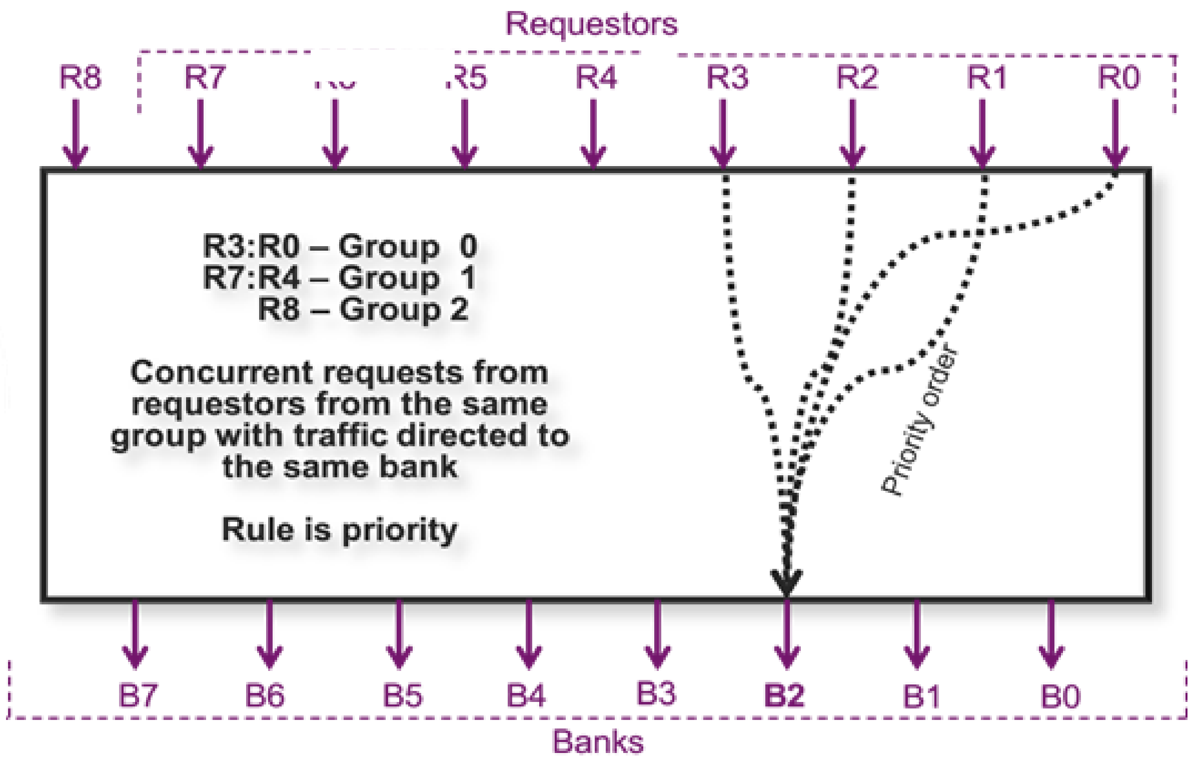
Serialisation

Priority

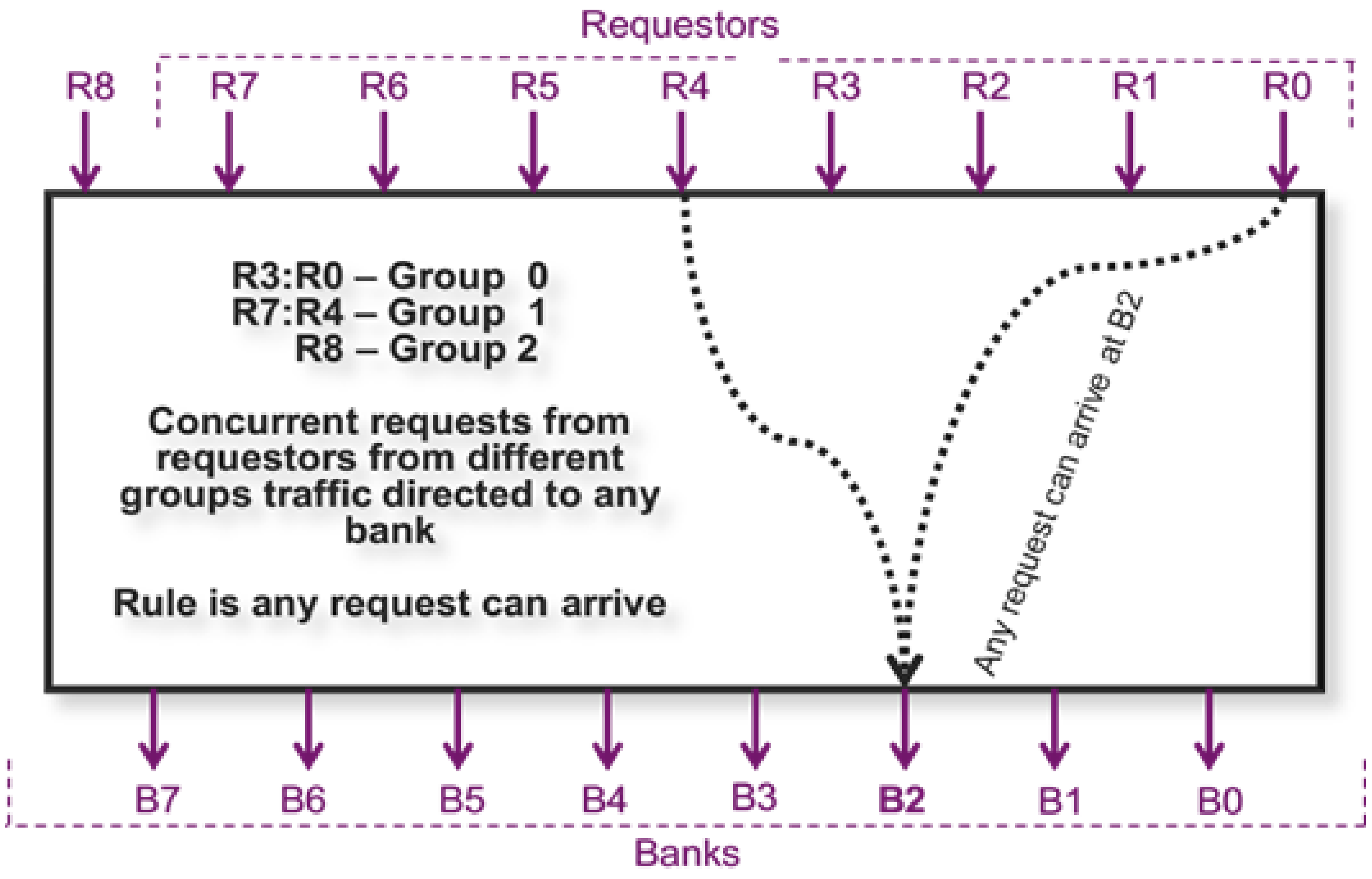
Arbitration Policy



Requestors
R8
R7
R6
R5
R4
R3
R2
R1
R0
R3:R0 – Group 0
R7:R4 – Group 1
R8 – Group 2
Requests from the same requestor within a group with traffic directed to a given bank
Rule is first-come-first-serve
First-come-first-serve
B7
B6
B5
B4
B3
B2
B1
B0
Banks



Rules of Arbitration


Requestors
R8
R7
R6
R5
R4
R3
R2
R1
R0
R3:R0 – Group 0
R7:R4 – Group 1
R8 – Group 2
Concurrent requests from requestors from different groups traffic directed to any bank
Rule is any request can arrive
Any request can arrive at B2
B7
B6
B5
B4
B3
B2
B1
B0
Banks

Proof Engineering

Scalable formal verification

=

"Proof Engineering"

Assume Guarantee

Scenario Splitting

Case Splitting

$10^{120\ \text{MILLION}}$ states; 1 billion gates and beyond

Scenario Splitting

- First scenario
  - Focus only first-come-first-serve behaviour
- Second scenario
  - Focus on multi-instance activity but narrow down the observation to only those requests that are directed to the same bank
- Last scenario
  - Observe traffic originating from any requestor going to any bank and ensuring that these requests are received and not lost

Multiple Active Requests: Case Splitting

- Two transaction abstraction + Case Splitting
- Overall 176 assertions for
  - REQ=2 [2 requestor groups]
  - GRP=4 [4 groups of mem banks]
  - BNK=2 [2 banks in each group]
  - CASES=11

CASES
R0, R1
R0, R2
R0, R3
R0, R1, R2
R0, R1, R3
R0, R2, R3
R1, R2, R3
R1, R2
R1, R3
R2, R3
R0, R1, R2, R3

Solution Mechanics

- Smart tracker abstraction for checking first-come-first-serve
- Two-transactions abstraction for establishing priority

DATA STRUCTURES

```
wire [BNK-1:0]   ready_to_sample_out [REQ-2:0][i-1:0][GRP-1:0];
wire [3:0]       decr                [REQ-2:0][i-1:0][GRP-1:0][BNK-1:0];
wire [i-1:0]     hsk_in              [REQ-2:0];
wire [BNK-1:0]   hsk_out             [GRP-1:0];

reg [MAX:0]      tracking_cnt        [REQ-2:0][i-1:0][GRP-1:0][BNK-1:0];
reg [BNK-1:0]    seen_in_watched_id  [REQ-2:0][i-1:0][GRP-1:0];
reg [BNK-1:0]    seen_out_watched_id [REQ-2:0][i-1:0][GRP-1:0];
```

64 TRACKING COUNTERS NEEDED

Master Checks: First-come-first-serve

```
generate
 for (g=0; g<GRP; g=g+1)
      for (b=0; b<BNK; b=b+1)
       assert property (ready_to_sample_out[0][0][g][b] &&
                        !other_req_active[0]
                        |->
                        (OUT_ID[g][b] == watched_id));
endgenerate
```

RUNTIME IS 1 HOUR PER PROPERTY

OVERALL 64 ASSERTIONS

Master Checks: Priority Checks

```
reg [CASES-1:0] seen_multi_inst[REQ-1:0][GRP-1:0][BNK-1:0];
generate
 for (r=0;r<REQ-1;r=r+1)
   for (g=0;g<GRP;grp_i=g+1)
       for (b=0;b<BNK;b=b+1)
        check_arbitration_i0_and_i1:
                     assert property (seen_multi_inst [r][g][b][0] &&
                                      !req_out [r][0][g][b]
                                         |->
                                      !req_out [r][1][g][b]);
       endgenerate
```

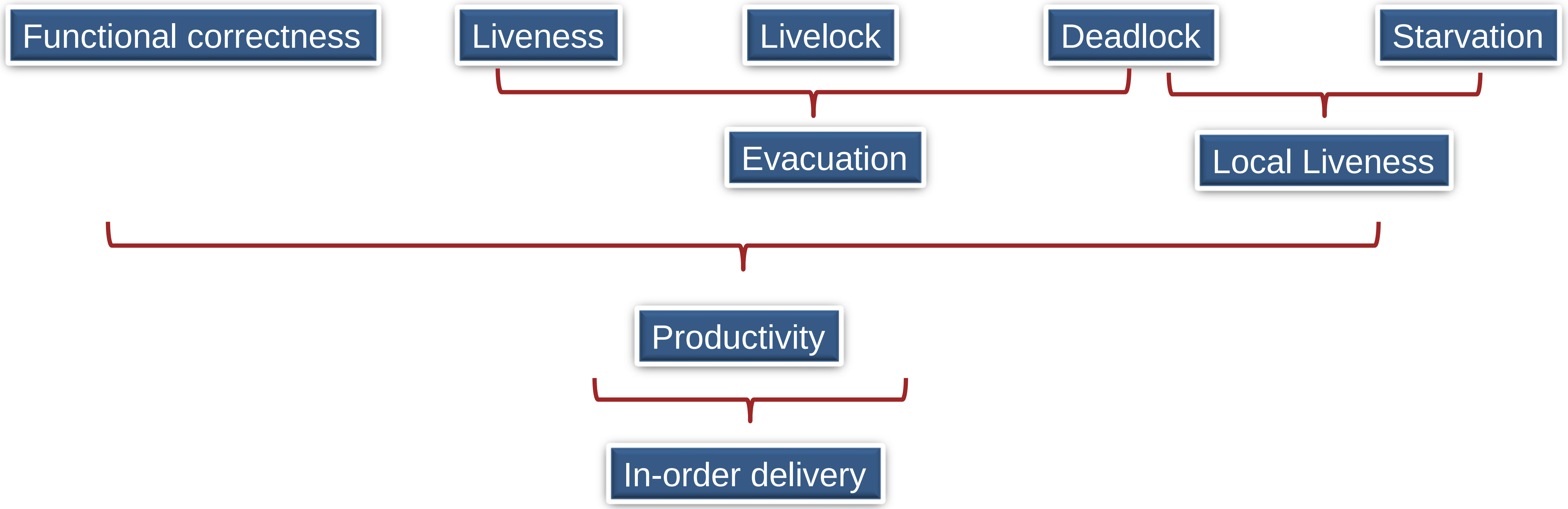
RUNTIME: 5-7 MIN PER ASSERTION

OVERALL 176 ASSERTIONS

Correctness Requirements

- Starvation
  - If any requestor was starved access it would never be seen coming out
- Fairness
  - If any requestor was serviced unfairly with respect to the arbitration scheme then the priority/ordering assertions would fail
- Liveness
  - All incoming requests granted eventually proved using assume guarantee
- Deadlock
  - No deadlock as all incoming requests are granted at correct destination

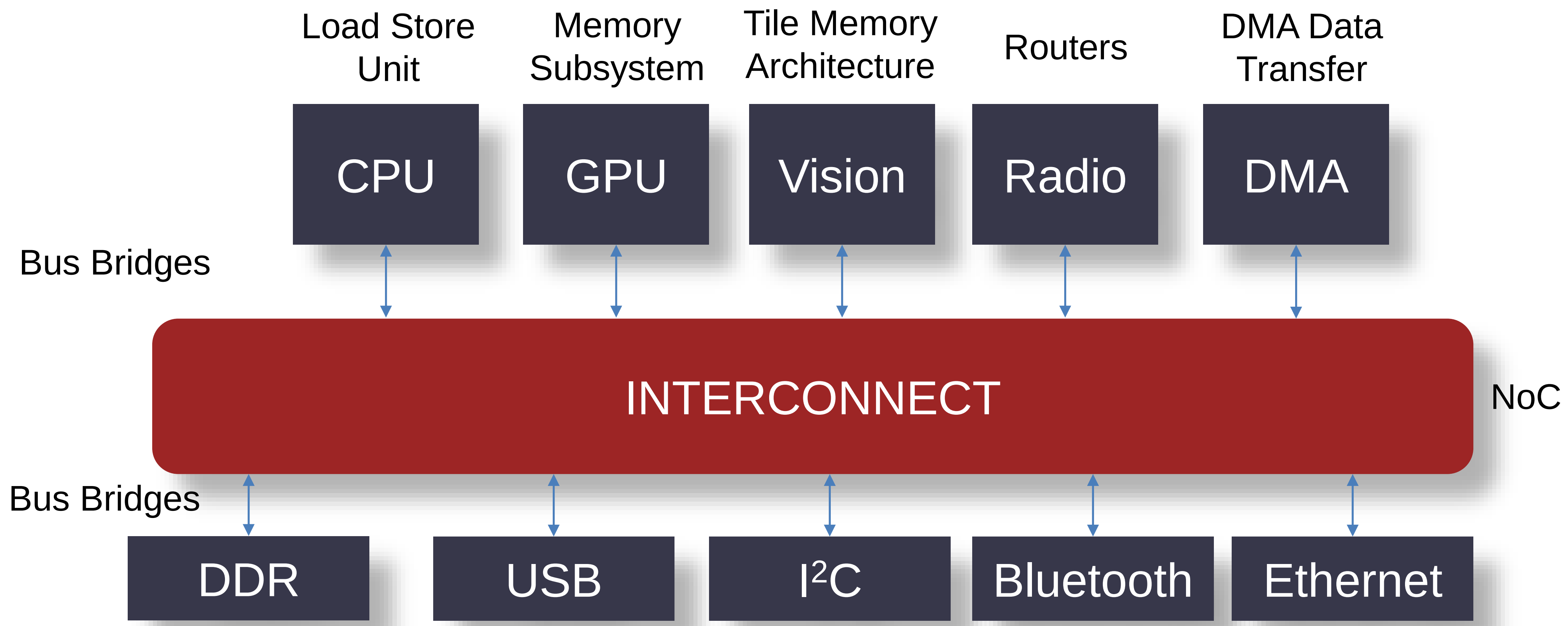
Revisiting Correctness Graph



In-order delivery is a stronger notion of correctness than productivity

Source: Formal Verification of On-Chip Communication Fabrics, Freek Verbeek, 2013, Radboud University

SoC Verification



Sequential complexity

Summary

- We described challenges with SoC Verification
- Showed how an agile formal verification flow can be used
  - Focus was on Erase and Prove phases
- Tutorial on how to build efficient and scalable formal models
- Addressed scalability aspects through case studies
  - Abstraction
  - Assume Guarantee
  - Scenario Splitting
  - Case splitting

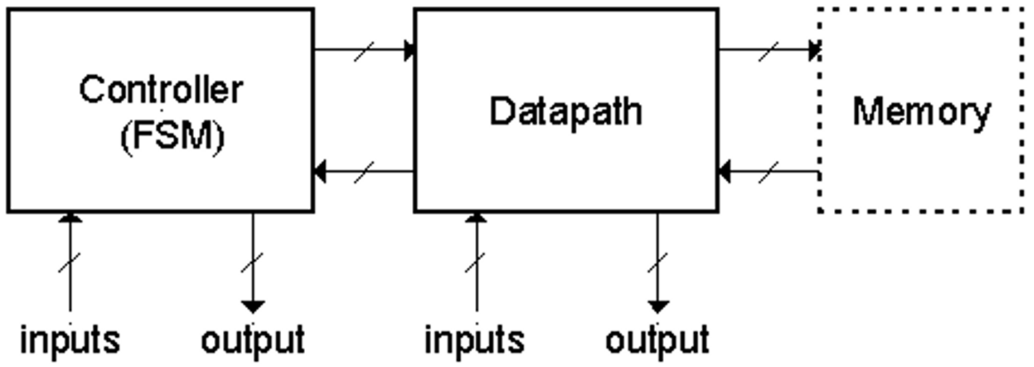
FORMAL VERIFICATION FOR CONTROL PATHS

Agenda

- Control Path Complexity
- Formal Apps for Control Path Verification
- Formal Environment Architecture
- Abstraction/Bug Hunting Techniques
- Example Bugs

CONTROL PATH COMPLEXITY

High Level Design Architecture



Controller

Accepts external and control input, generates control and external output and sequences the movement of data in the Datapath.

Datapath

Responsible for data manipulation. Usually includes a limited amount of storage.

Memory

Optional block used for long term storage of data structures.

FSMs For Control Paths

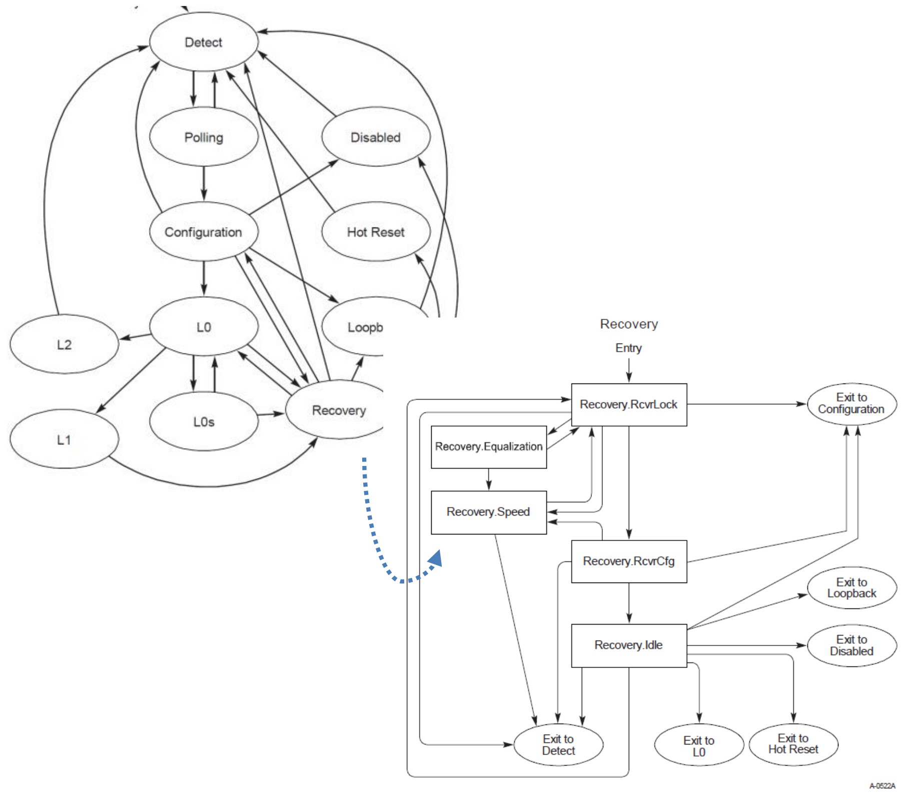
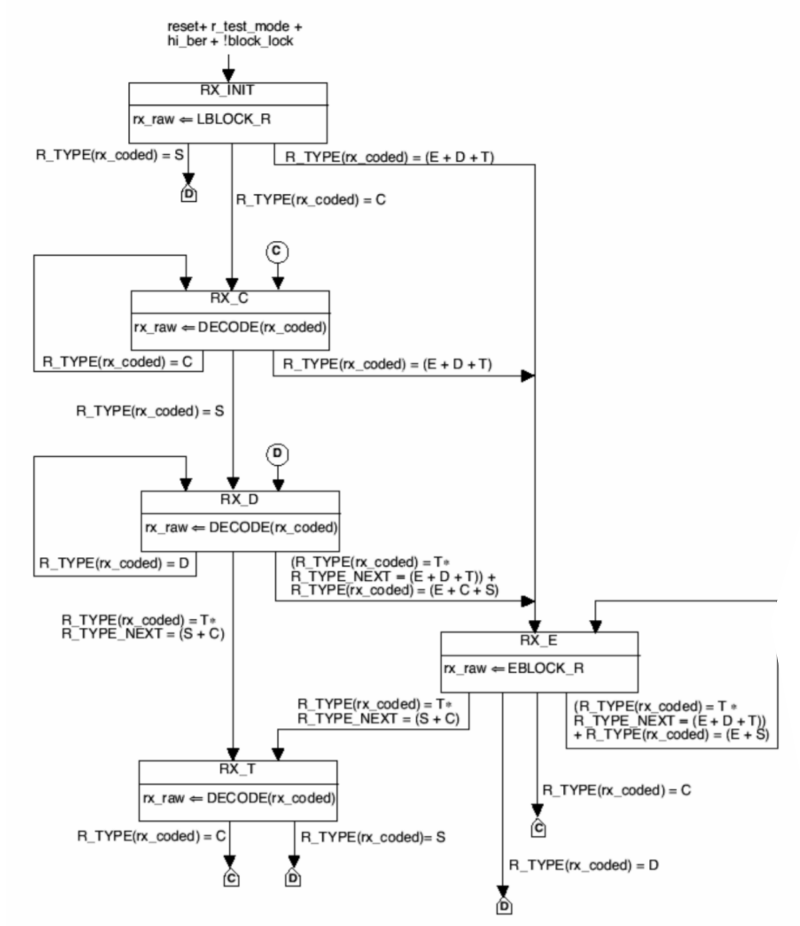


Figure 4-27: Recovery Substate Machine

PCIE/USB LTSSM



reset+ r_test_mode + hi_ber + !block_lock
RX_INIT
rx_raw ⇐ LBLOCK_R
R_TYPE(rx_coded) = S
R_TYPE(rx_coded) = (E + D + T)
R_TYPE(rx_coded) = C
RX_C
rx_raw ⇐ DECODE(rx_coded)
R_TYPE(rx_coded) = C
R_TYPE(rx_coded) = (E + D + T)
R_TYPE(rx_coded) = S
RX_D
rx_raw ⇐ DECODE(rx_coded)
R_TYPE(rx_coded) = D
(R_TYPE(rx_coded) = T • R_TYPE_NEXT = (E + D + T)) + R_TYPE(rx_coded) = (E + C + S)
R_TYPE(rx_coded) = T • R_TYPE_NEXT = (S + C)
RX_E
rx_raw ⇐ EBLOCK_R
R_TYPE(rx_coded) = T • R_TYPE_NEXT = (S + C)
(R_TYPE(rx_coded) = T • R_TYPE_NEXT = (E + D + T)) + R_TYPE(rx_coded) = (E + S)
RX_T
rx_raw ⇐ DECODE(rx_coded)
R_TYPE(rx_coded) = C
R_TYPE(rx_coded)= S
R_TYPE(rx_coded) = C
R_TYPE(rx_coded) = D
C
D

10G/40G Ethernet

Control paths commonly implemented using FSMs:

- Many states w/ sub-states M/C
- Multiple paths to reach to the same state
- Convoluted state transitions
- Priority among simultaneously occurring transition conditions

Control Path Verification Challenges

Traditional coverage metrics insufficient

Limited visibility at the interface boundary

State space explosion due to temporal input behavior

No good models to check control path accuracy

Not every control path bug manifest into scoreboard bug efficiently

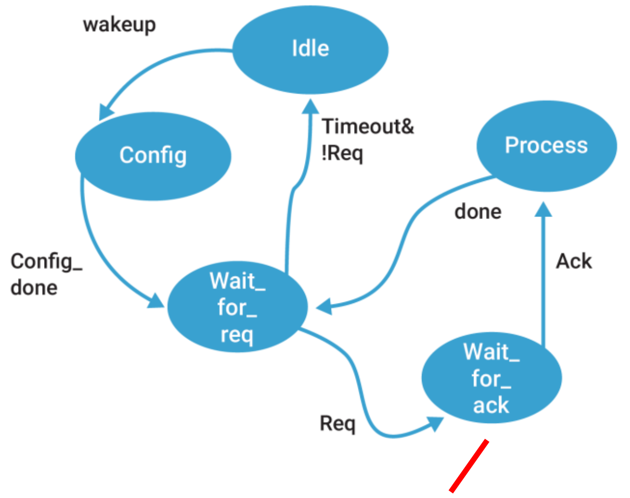
Example Bug Escapes

Bug: What happens when ack and req are asserted in the same cycle?

A packet processing engine responsible for packet transfers to/from memory and is sharing DMA

- ***DMA acknowledge the request (Req) within 0-10 cycles***
- ***Acknowledge (Ack) is asserted for one cycle***
- ***Ack is asserted in the same cycle as Req when no other high priority request is pending or for a parking master***

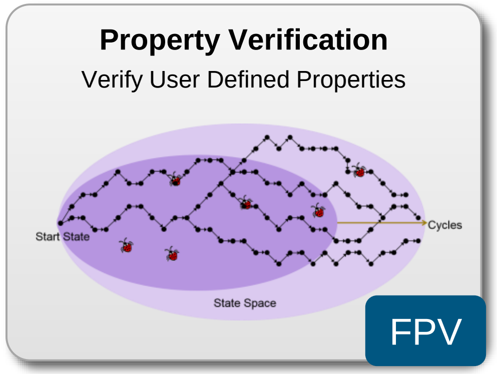
RTL Implementation of State Machine

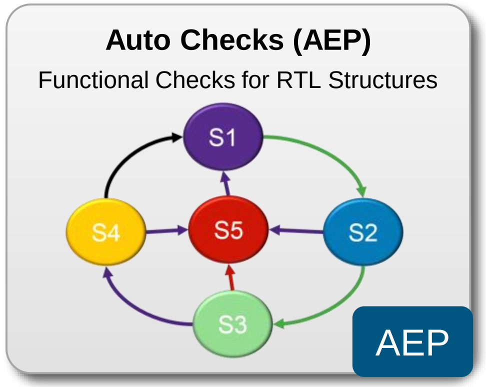


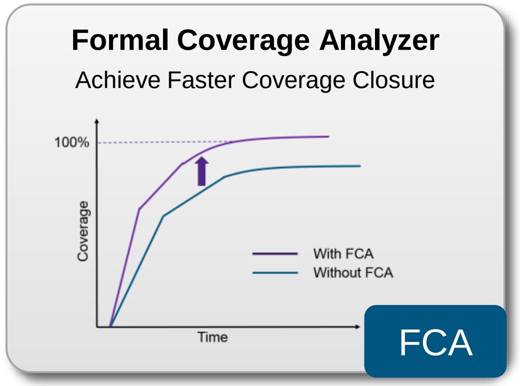
Will starve permanently waiting for ack

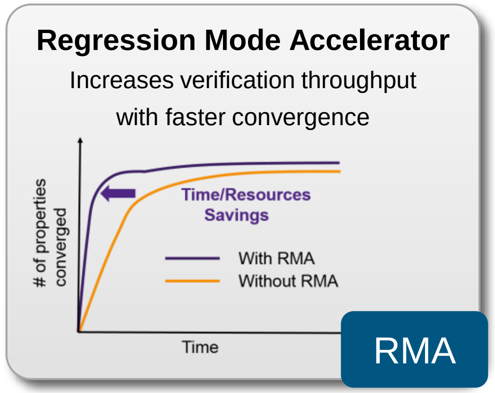
FORMAL APPS FOR CONTROL PATH VERIFICATION

Introduction to Formal Apps









Build & Verify user properties

Auto generated properties

Periodic regressions w/ faster closure

FORMAL ENVIRONMENT ARCHITECTURE

Properties for FSM

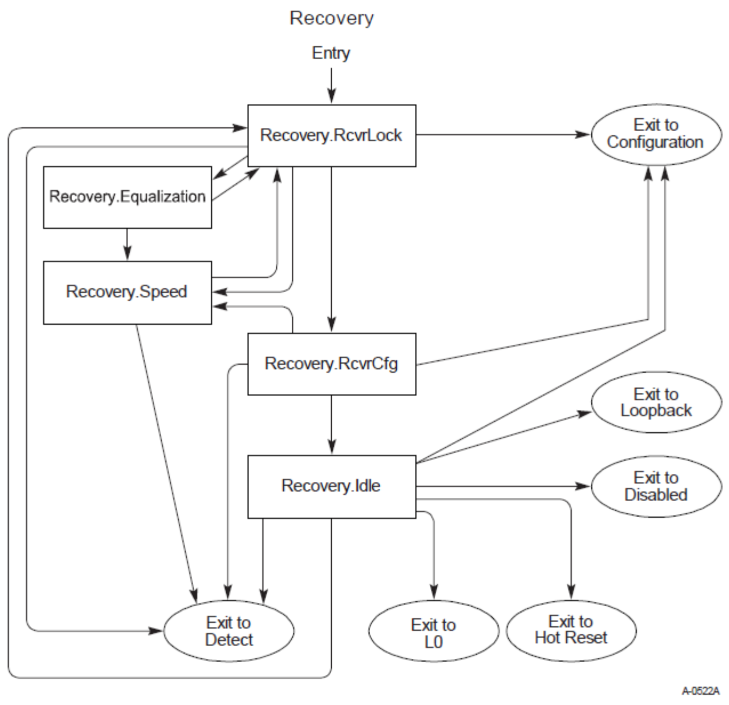


Figure 4-27: Recovery Substate Machine

State Transition (In) checks

Entry to Recovery_Speed is possible only when current states are Recovery_lock, Recovery_Config, Recovery, eq* states

State Transition (Out) checks

When current state Recovery_Speed changes, next state possible are Recovery_lock and Detect_Quiet states

State (Output) checks

Directed_speed_change if asserted, has to be go low when entered in Recovery.speed state

End-To-End checks

When targeted link speed is different than current port speed, LINK DATA RATE has to change in # time

Coding Assertion Properties

Transition (IN) checks

Transition (Out) checks

Output Signal checks

Assertions using RTL internal signals
+ Good for bug hunting
+ Typically 1/2 cycle assertions; Easier to converge
+ Lesser COI; Pinpoints to the root cause
- Needs design knowledge

End-to-end checks

Assertions using RTL boundary signals
+ Map to spec level features; hence gives higher confidence
- Tend to be long temporal; relatively Harder to converge
- Harder to code and debug

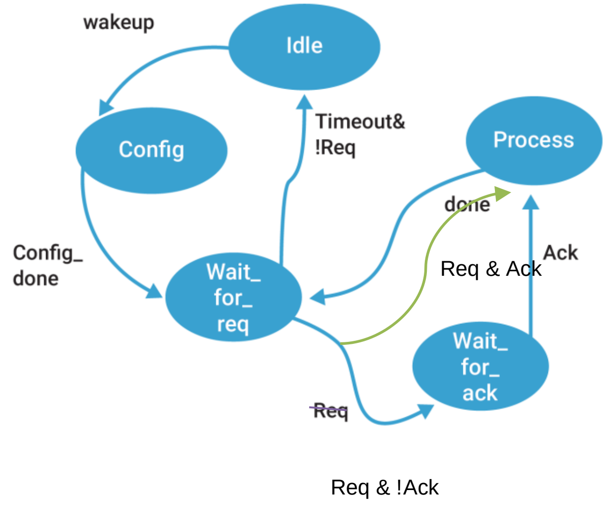
Methodology Recommendation: Begin with white-box/internal assertions and then move to end to end checks (Assume-Guarantee in upcoming slides)

Orthogonal Properties

Orthogonal properties capturing specification intent (than duplicating RTL) finds bugs

Bug: What happens when Ack and Req are asserted in the same cycle?

RTL Implementation of State Machine



Cs == Wait_for_req && Req |=> Cs == Wait_for_ack
Cs == Wait_for_ack && Ack |=> Cs == Process

Won't find bug

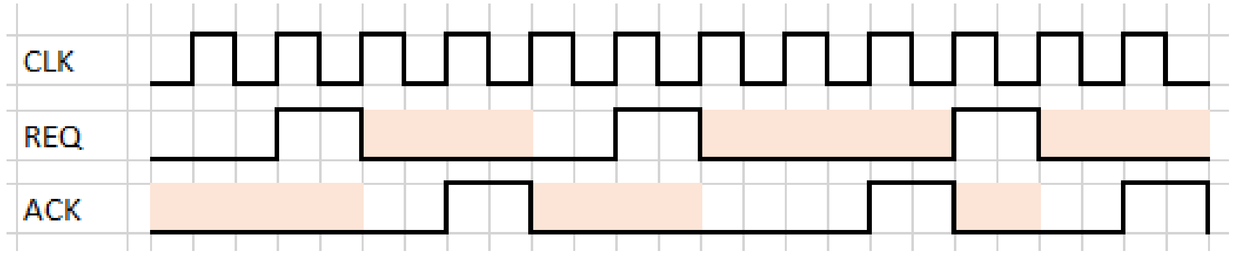
Ack |=> Cs == Process

Will find bug

RTL Helper Model for Assertions – 1/2

RTL Helper code to model assertions are easier to code, debug and better for tools to converge

- Master drives REQ as pulse signal, Slave responds with ACK as pulse signal
- The relation between REQ and ACK is 1 on 1, Slave must not assert ACK more than asserted REQ

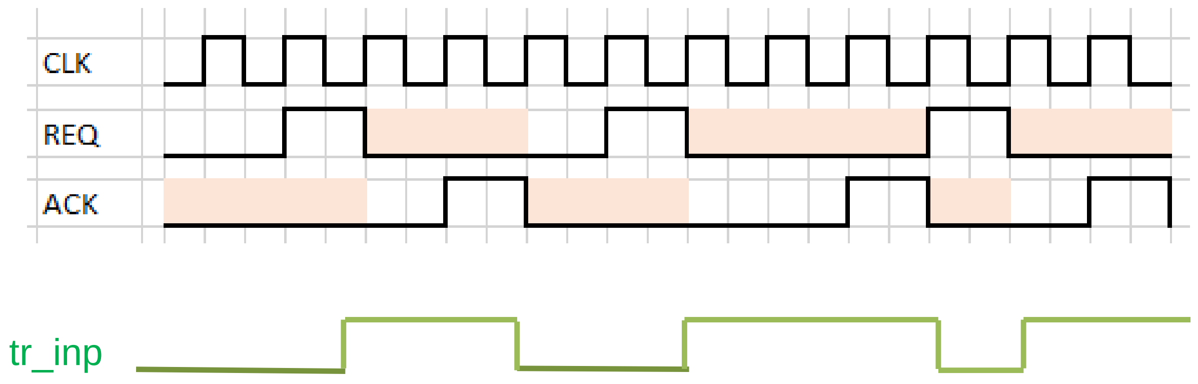


```
assert property (@(posedge CLK)   REQ |=> !REQ);
assert property (@(posedge CLK)   ACK |=> !ACK);
assert property (@(posedge CLK)   REQ |-> ##[1:$] ACK);
assert property (@(posedge CLK)   not (REQ && ACK));
….
```

Easier to model in Verilog

RTL Helper Model for Assertions – 2/2

Create a new signal, tr_inp, using RTL and write assertions using it



```
logic tr_inp;

always @(posedge CLK or negedge RSTN)
  if (!RSTN) begin
     tr_inp <= 1'b0;
  end else begin
     if (REQ)
       tr_inp <= 1'b1;
     else if (ACK)
       tr_inp <= 1'b0;
  end
```

```
property p_req_allowed;
   @(posedge CLK) disable iff (!RSTN)
     REQ |-> !tr_inp;
endproperty
property p_ack_allowed;
   @(posedge CLK) disable iff (!RSTN)
     ACK |-> tr_inp;
endproperty


ast_ack_allowed   : assert property(p_ack_allowed);
ast_ackid_valid   : assert property(p_ackid_valid);
```

Constraints Strategy

- Over-constrained environment
  - Pros:
    - Properties will pass easily
    - Easier to understand RTL behavior
  - Cons
    - Constraints will need to be reviewed and removed
    - **Chance of missing important bugs**

- Under-constrained environment
  - Pros:
    - **Will be covering a lot of RTL**
    - Highly effective for bug hunting
  - Cons:
    - Harder to get a good proof early on

Recommendations:
- Start with fewer constraint for effective bug hunting
- Add constraints one by one; constraint layering – methodical approach

Effective Use of Cover Properties

Early RTL Exploration

Early stages of verification to do how and what if analysis

Constraint Analysis

To ensure design is not getting over-constrained

Deep Bug Hunting

To guide hybrid engines to cover interesting state or scenario many cycles deep from reset

Bounded Proof Depth Analysis

Useful to decide the cycle depth when bounded proofs can be signed-off with confidence

Integration wit Verification Plan

Important cover properties can be integrated back to verification plan for coverage signoff

ABSTRACTION/BUG HUNTING TECHNIQUES

Driving Deep Sequential Inputs/States

Cut-points to drive interesting scenarios on deep sequential signals with constraints

E.g. Deep state machine w/
- TS1_64_Symbol_Rcvd signal need to be asserted to go from RcvrLock to RcvrCgf State.
- Would need 64 consecutive TS1 symbols for signal to go high

Cut the driver for TS1_64_Symb_Rcvd
assume next_state == RcvrLock |-> ##[2:64] TS1_64_Symb_Rcvd

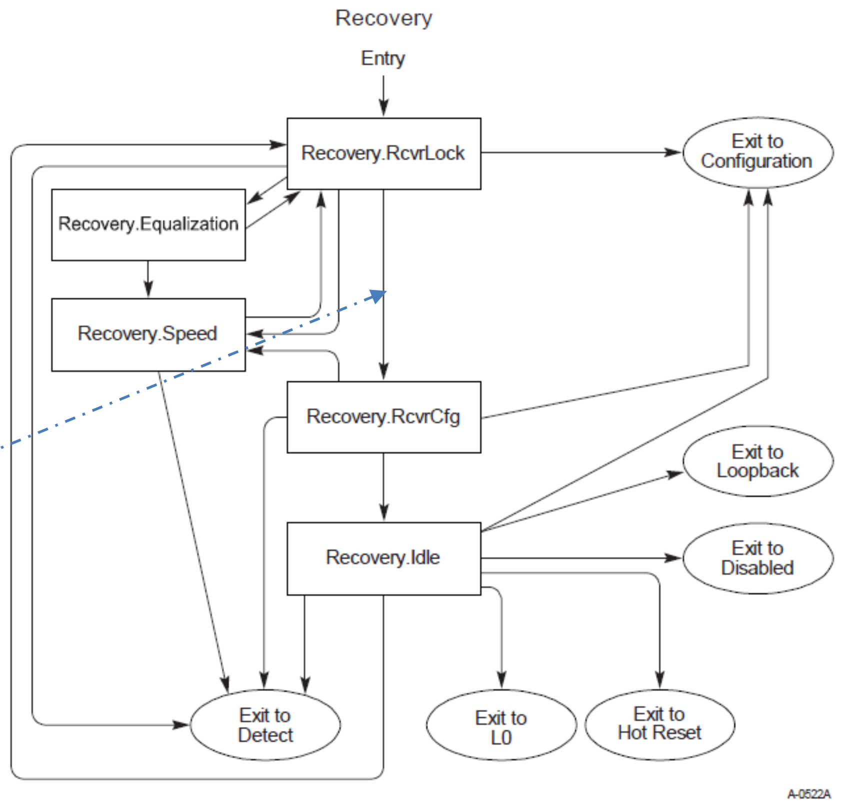


Figure 4-27: Recovery Substate Machine

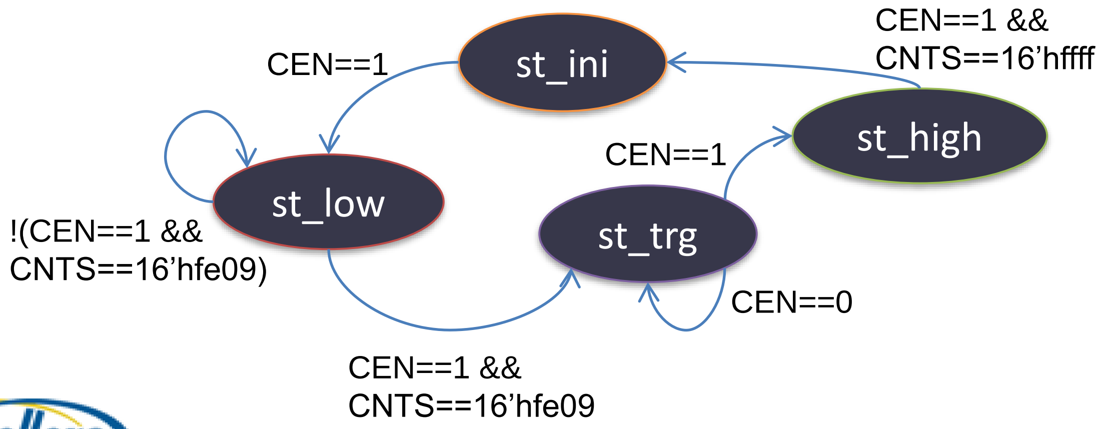
Counter/Timer Abstractions

- Reducing the length of counters/timers
  - Using parameterized or `define RTL settings

    #(.CREDIT_THRESHOLD (8)) // instead of 64
  - Using automatic or guided abstraction commands in the tool setup

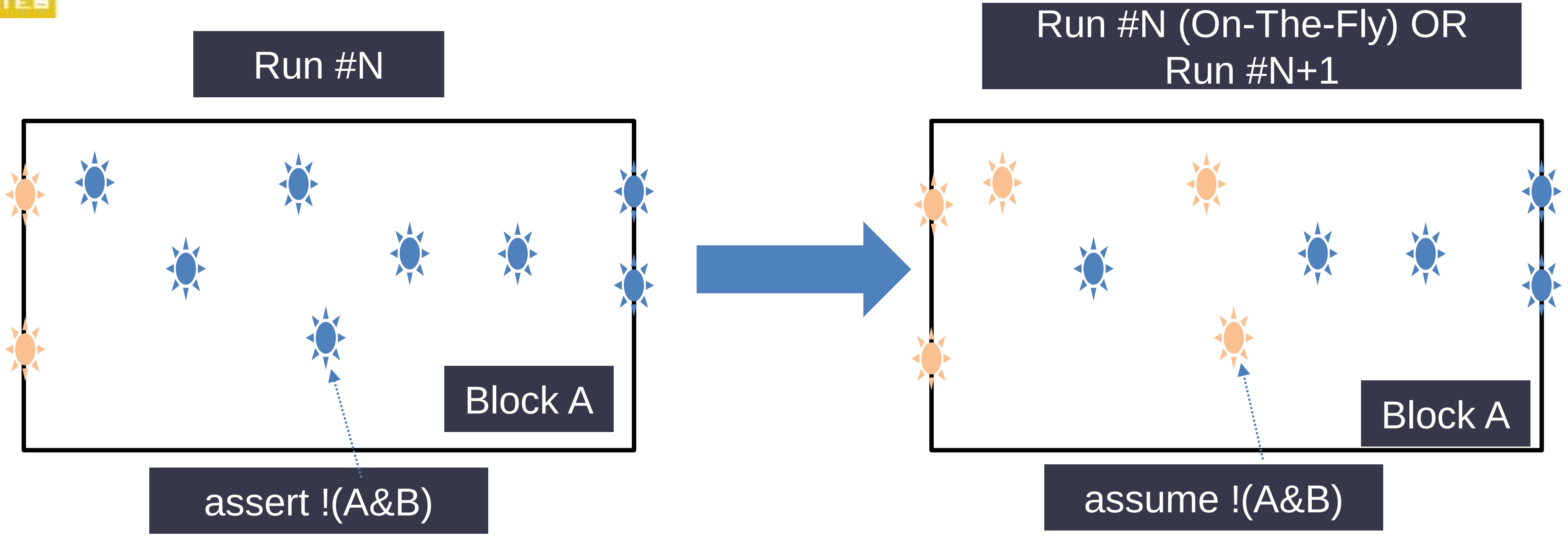
```
set_abstraction -construct count=8
```

- Cut the driver and apply additional constraints for interesting cases based on the spec



```
Snip driver to CNTS
assume –expr {abscnt==st_ini  |-> CNTS==0}
assume –expr {abscnt==st_low  |-> CNTS>0 && CNTS<16'hfe10}
assume –expr {abscnt==st_trg  |-> CNTS==16'hfe10}
assume –expr {abscnt==st_high |-> CNTS>16'hfe10 &&
CNTS<=16'hffff}
```

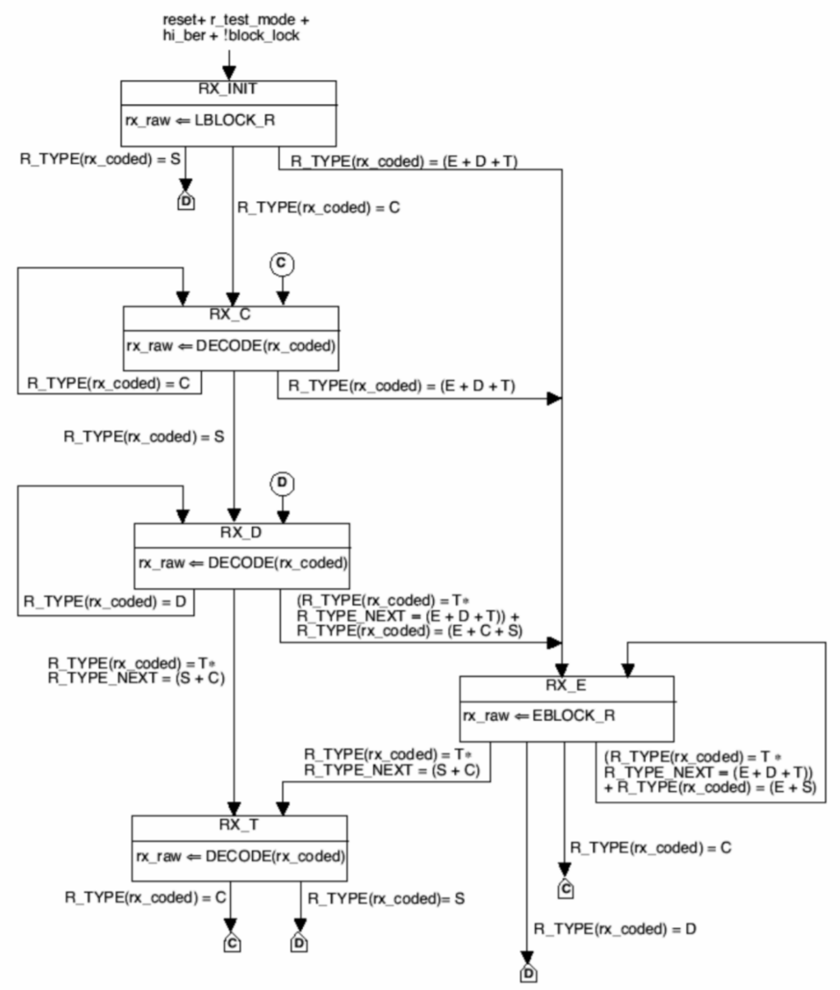
Assume Guarantee



Proven assertions are treated as assumptions for subsequent properties

- Very useful for harder to prove properties in common cone of influence
- Internal (intermediate) properties act as effective invariants and assumptions for end to end properties
- Can be applied on the fly for same run or subsequent runs

Case Splitting



10G/40G RX S/M

Independent modes/values/bit indexes to be verified can be separated into diff formal tasks; reduces complexity

WARNING: Make sure no modes are left out during case enumeration

assume mode == 10G
Check assertions

10G Mode

assume mode == 40G
Check assertions

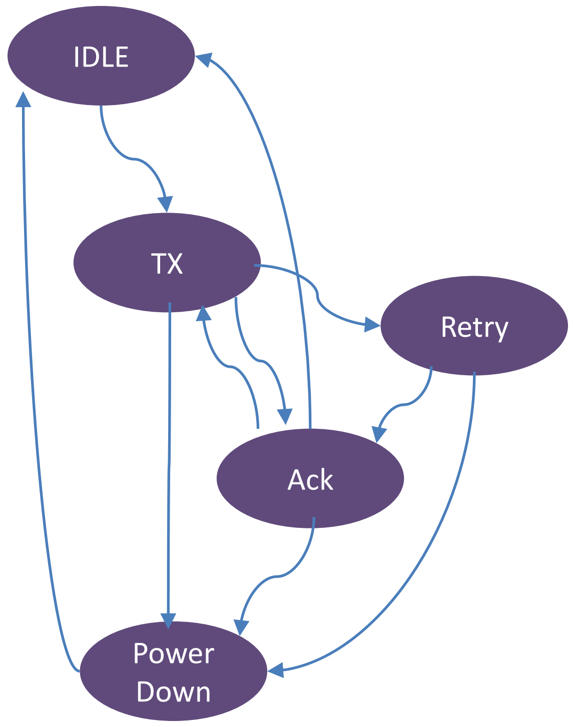
40G Mode

Enumerate different modes/scenarios for case split

EXAMPLE BUGS

Example Bugs – 1/3

States and transitions covered, but not all paths.



During transmission (TX State), a received error count is maintained.

If error count reaches certain threshold, S/M goes to retry state.

BUG: Error count is expected to reset to zero before starting new data transmission; however it is reset only in IDLE state and not Ack state (after entry from Retry)

Paths covered:

IDLE -> TX -> ACK -> TX….

IDLE -> TX -> RETRY -> ACK -> IDLE…

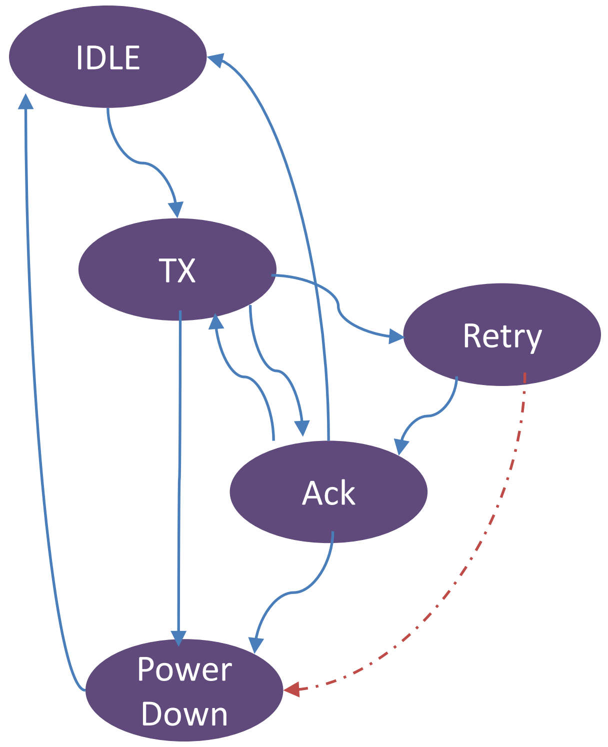
Paths not covered

IDLE -> TX -> RETRY -> ACK -> TX…

```
assert property $rose(current_state == TX) |-> error_count == 0
```

Example Bugs – 2/3

Missing state transition conditions harder to find



Whenever low power request is asserted, state machine should immediately go to Power-Down state, turn off all clocks, and move to IDLE (non-active) state.

BUG: Design misses the transition arc from Retry state causing downstream logic to get into deadlock state

Hard to ensure from design boundary that power-down signal is asserted from every possible state (less visibility)

If the design misses the transition arc, code coverage tools cannot find it

```
assert property $rose(powerdown_request) |-> next_state == PowerDown
```

Example Bugs – 3/3

Priority between simultaneously occurring state transition conditions often harder to cover

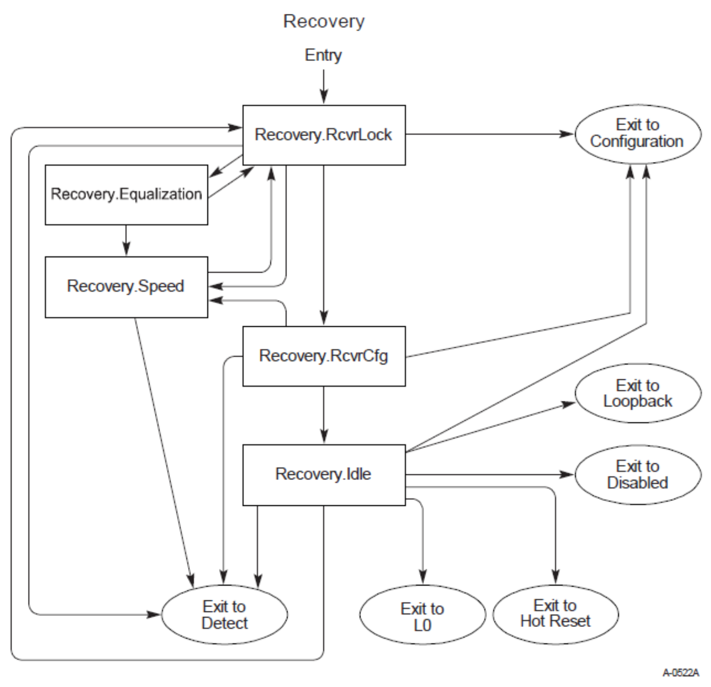


Figure 4-27: Recovery Substate Machine

case (Current_state)

…

Recovery.idle:

If(consecutive_8_symb_data_rcvd) next state == L0

….

else if(timeout_2ms_expired && no_activity) next_state = Detect

else if(!idle_to_rlock_transition_cnt_expired) next_state = Recovery.Lock

When multiple transition condition are true concurrently, priority of transition needs to be checked thoroughly

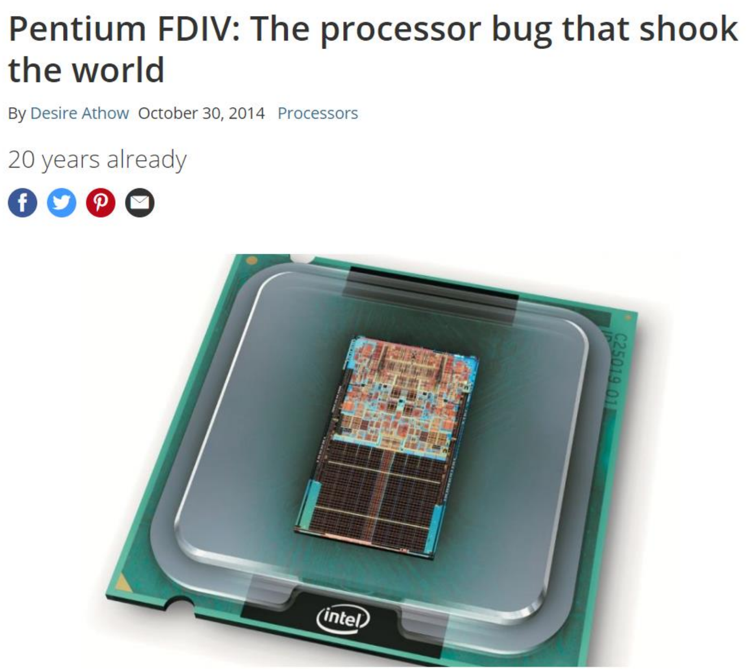
assert current_state == Recovery.Idle && *next_state == Detect |-> !idle_to_rlock_transition_cnt_expired*

Summary

- Control path complexity introduces bugs that escapes in traditional verification
- Formal apps with user and auto generated properties target control path exhaustively
- Architecting formal TB with orthogonal assertions and right set of constraints is key to success
- Abstraction techniques enable effective bug hunting to find corner case bugs

DATA PATH VERIFICATION

Datapath Correctness is Important… and Difficult!

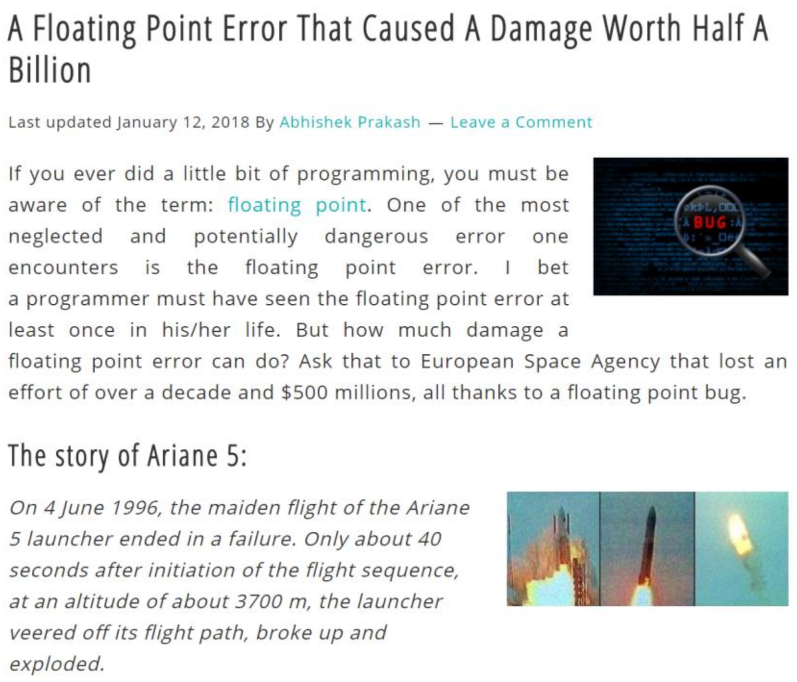


Pentium FDIV: The processor bug that shook the world

By Desire Athow October 30, 2014 Processors

20 years already

Source: https://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773



A Floating Point Error That Caused A Damage Worth Half A Billion

Last updated January 12, 2018 By Abhishek Prakash — Leave a Comment

If you ever did a little bit of programming, you must be aware of the term: floating point. One of the most neglected and potentially dangerous error one encounters is the floating point error. I bet a programmer must have seen the floating point error at least once in his/her life. But how much damage a floating point error can do? Ask that to European Space Agency that lost an effort of over a decade and $500 millions, all thanks to a floating point bug.

The story of Ariane 5:

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.

Source: https://itsfoss.com/a-floating-point-error-that-caused-a-damage-worth-half-a-billion/

KILLED BY A MACHINE: THE THERAC-25

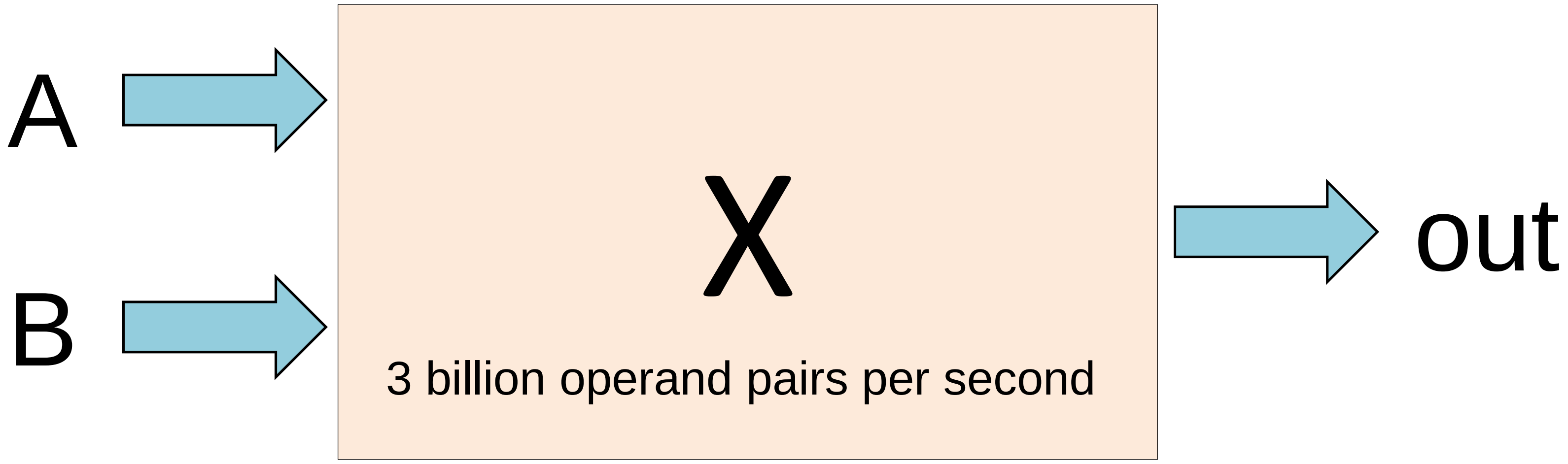
by: Adam Fabio

138 Comments

October 26, 2015

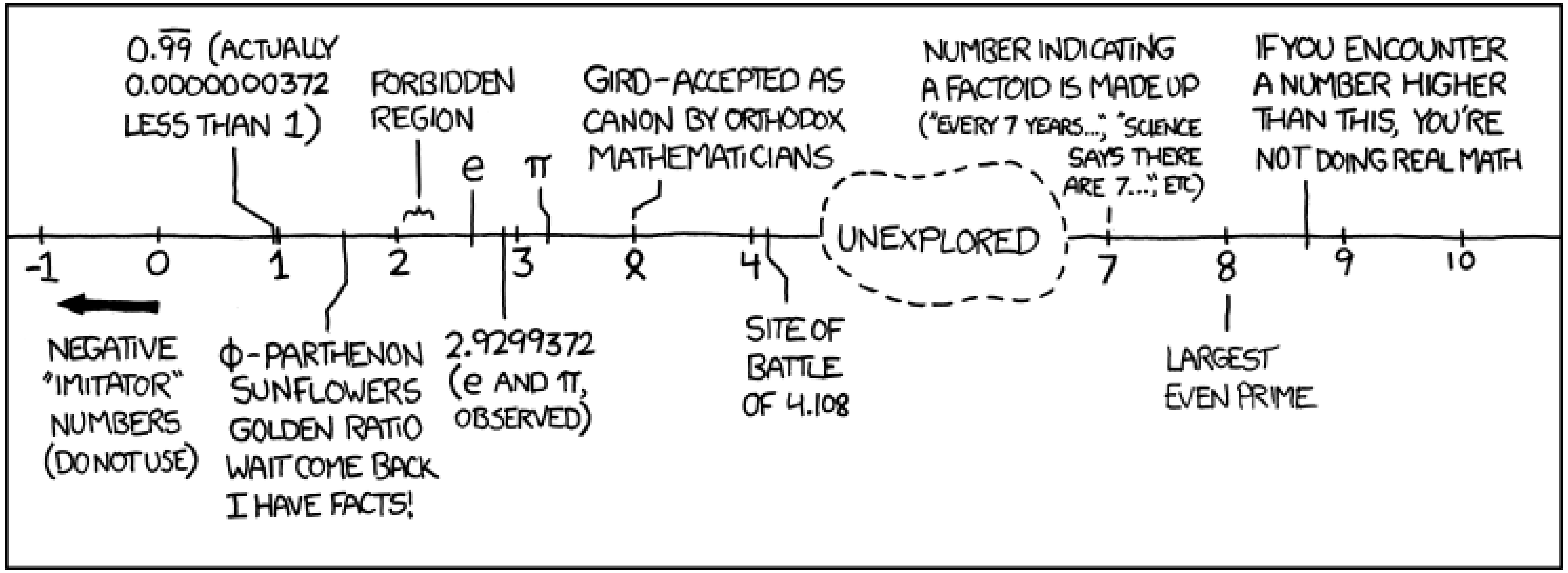
Source: https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

Exhaustive Simulation Takes Too Long



16-bit operands	32-bit operands	64-bit operands
1.5 seconds	195 years	$3.5 * 10^{21}$ years

What About Functional Coverage?


0.9̄9̄ (ACTUALLY 0.0000000372 LESS THAN 1)
FORBIDDEN REGION
e
π
GIRD-ACCEPTED AS CANON BY ORTHODOX MATHEMATICIANS
NUMBER INDICATING A FACTOID IS MADE UP ("EVERY 7 YEARS...", "SCIENCE SAYS THERE ARE 7...", ETC)
IF YOU ENCOUNTER A NUMBER HIGHER THAN THIS, YOU'RE NOT DOING REAL MATH
-1
0
1
2
3
4
UNEXPLORED
7
8
9
10
NEGATIVE "IMITATOR" NUMBERS (DO NOT USE)
ϕ-PARTHENON SUNFLOWERS GOLDEN RATIO WAIT COME BACK I HAVE FACTS!
2.9299372 (e AND π, OBSERVED)
SITE OF BATTLE OF 4.108
LARGEST EVEN PRIME

Source: xkcd.com

(Not a real data representation)

What About Functional Coverage?

Find two 32-bit floating point multiplication operands that:

- Have a positive result?
- Have the largest representable pre-rounded exponent?
- Have all 1’s in the pre-rounded mantissa?
- Round up?

Commercial Formal Verification Options

Property Verification
(VC Formal FPV)

- Familiar methodology
- Correct properties mean correct design
- Heavy use of bit-level modeling

Equivalence Checking
(HECTOR, VC Formal DPV)

- Small learning curve
- Correct properties mean equivalent designs
- Bit-level and word-level modeling

Industrially Used But Not Commercially Supported

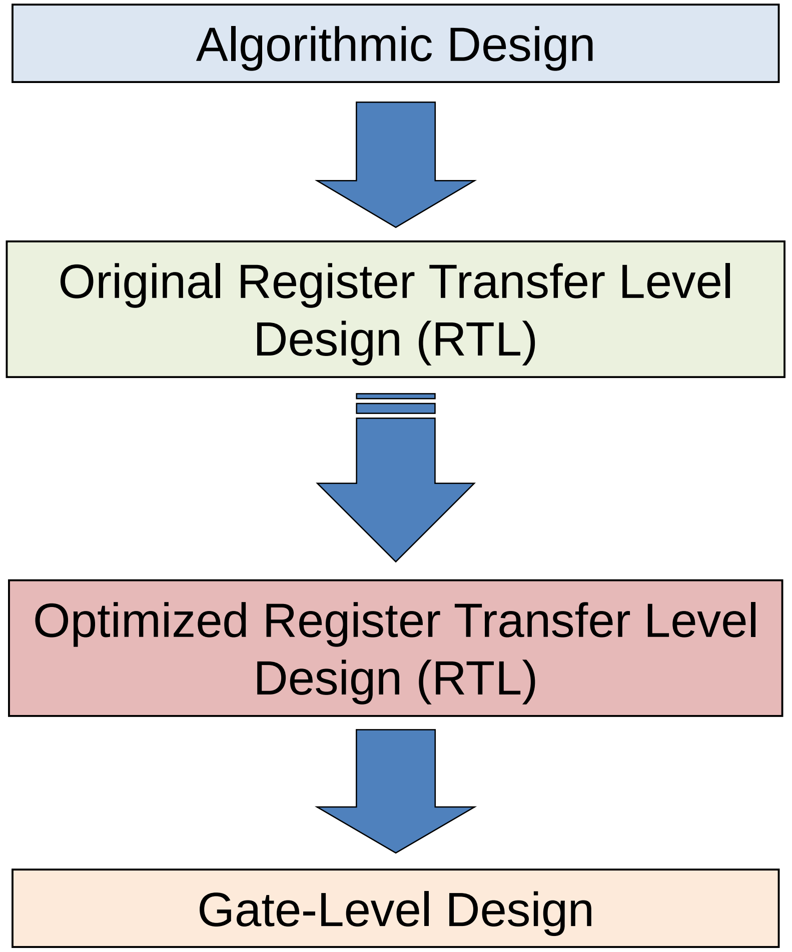
Theorem Proving

- Steep learning curve
- Correct properties mean correct design
- Bit-level and word-level modeling
- Long development time

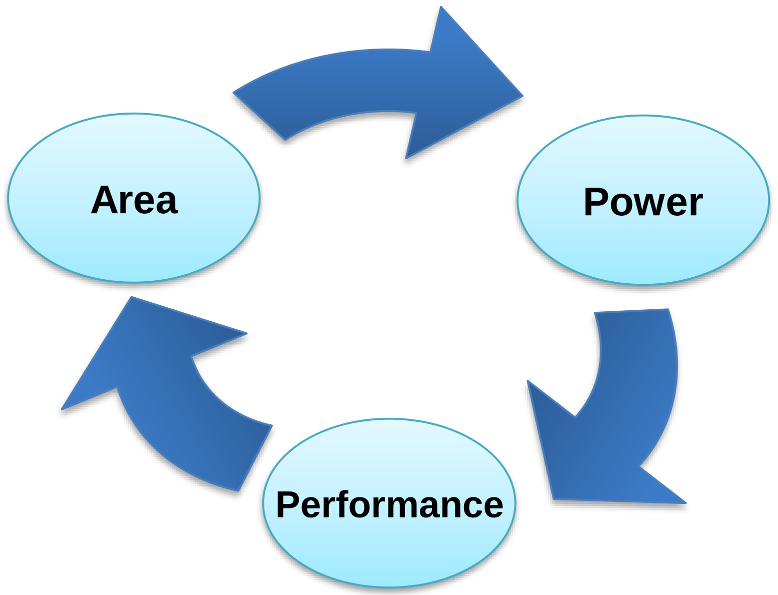
Symbolic Trajectory Evaluation

- Abstraction integrated into specification
- Potentially robust but requires manual abstraction in general.
- Proprietary and secret tooling.

Equivalence Checking Overview


Algorithmic Design
Original Register Transfer Level Design (RTL)
Optimized Register Transfer Level Design (RTL)
Gate-Level Design

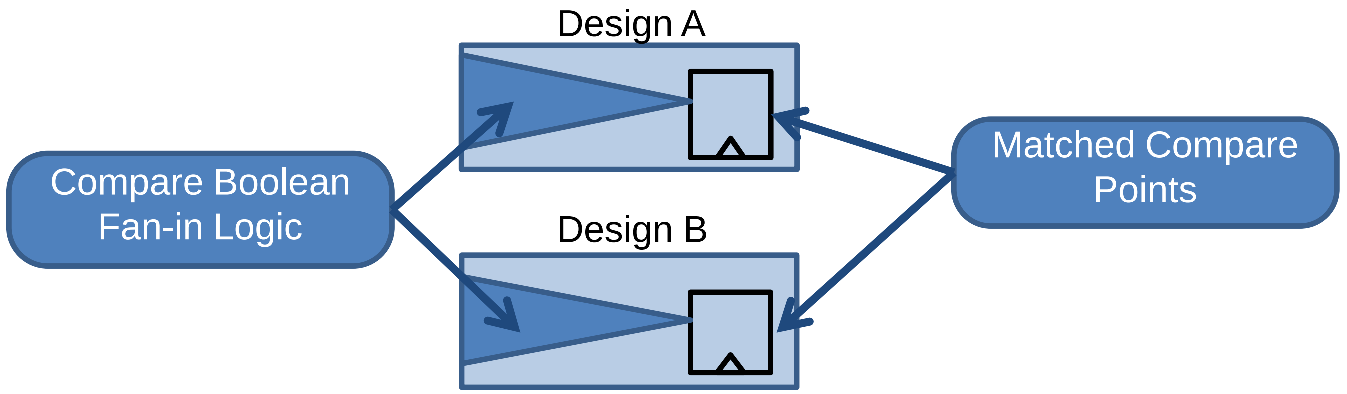
- *Untimed C/C++*
  < 100 – 10K lines >
- *Timed Verilog, VHDL*
  < 1K – 50K lines >
- *Timed Verilog, VHDL*
  < 1K – 100K lines >
- *Netlist*



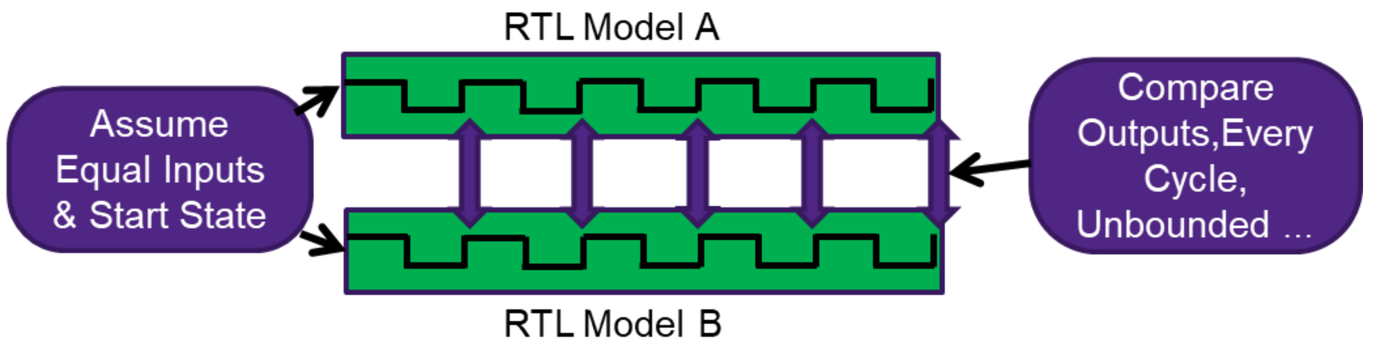
Automatic or manual refinement steps

Different Types of Equivalence Checking

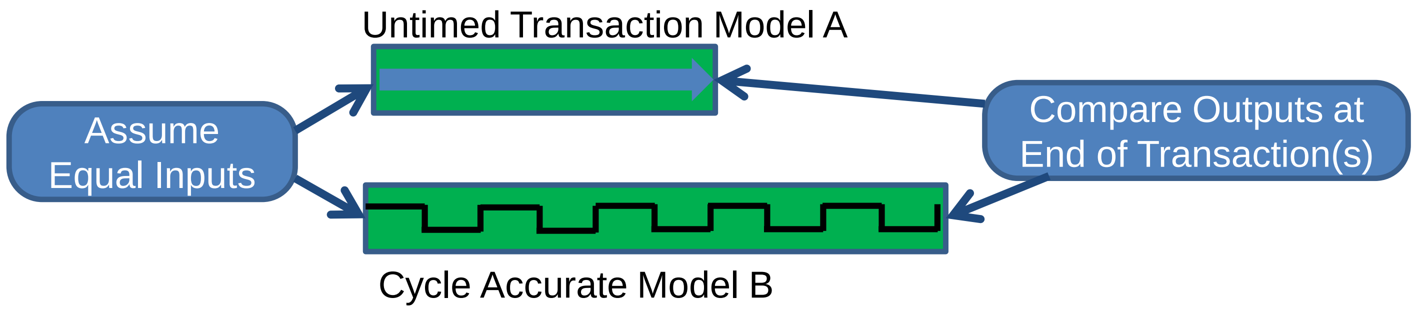
Boolean Equivalence (Formality)



Sequential Equivalence (SEQ)


RTL Model A
Assume Equal Inputs & Start State
Compare Outputs,Every Cycle, Unbounded …
RTL Model B

Transaction Equivalence (VC Formal DPV)

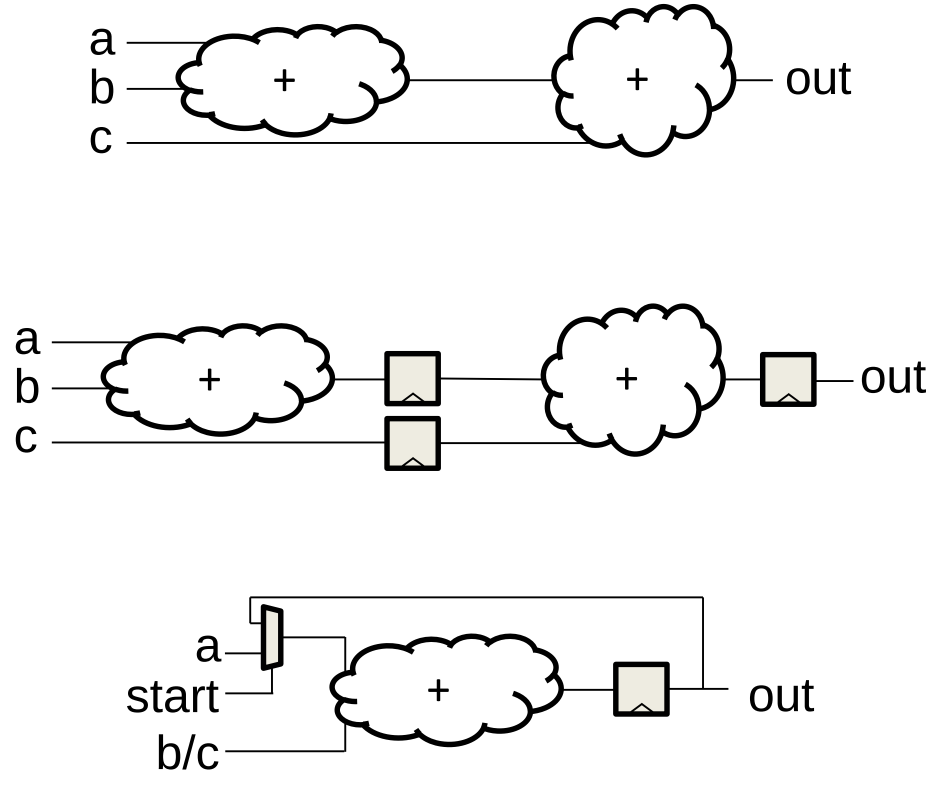

Untimed Transaction Model A
Assume Equal Inputs
Compare Outputs at End of Transaction(s)
Cycle Accurate Model B

What Is Transactional Equivalence?

Defining a transaction

- **A transaction consists of:**
  - Inputs
  - Input State (optional)
  - State Change
  - Outputs
  - Output State (optional)

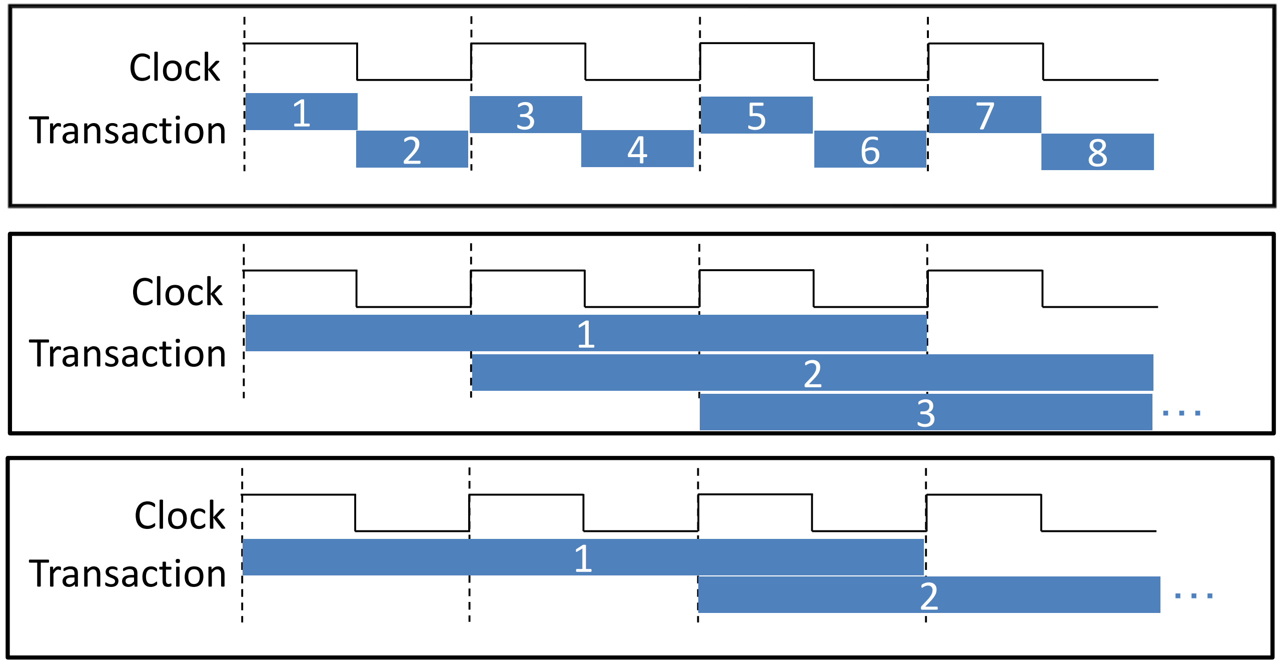
- **A transaction can be:**
  - Combinational
  - Sequential Overlapping / Pipelined
  - Sequential Non-Overlapping

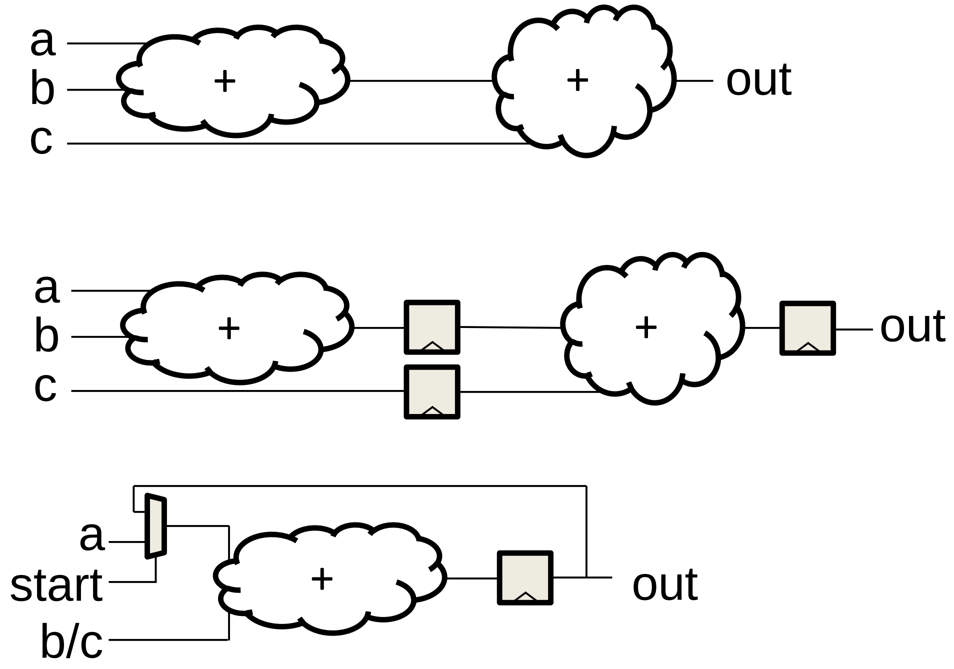


a + b + c = out

What is Transactional Equivalence?

Transaction boundaries

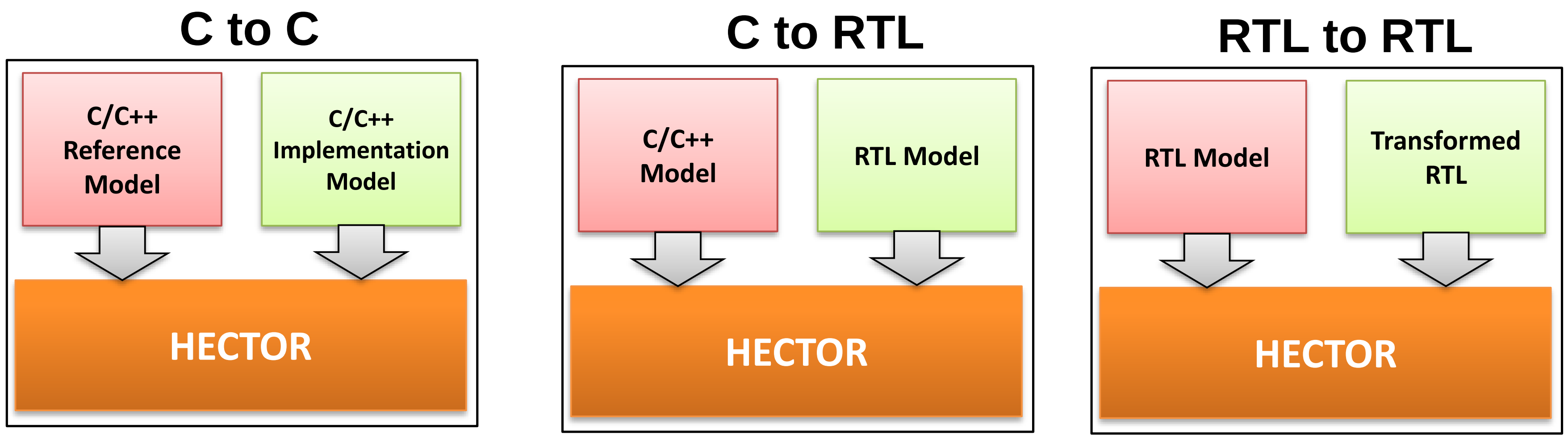




Flops capture on positive edge

Datapath Verification using HECTOR Technology

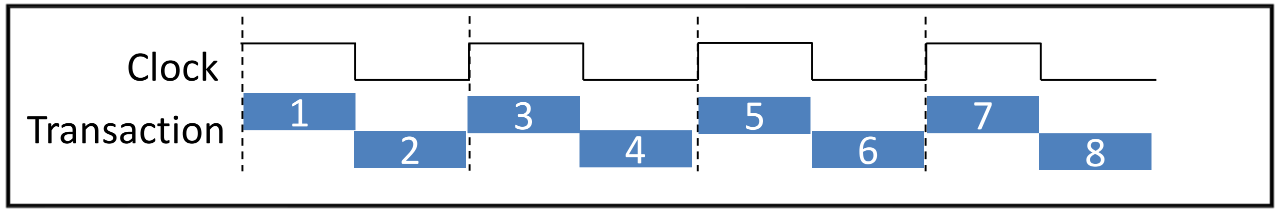
Formal Block-level Transaction Equivalence Checking

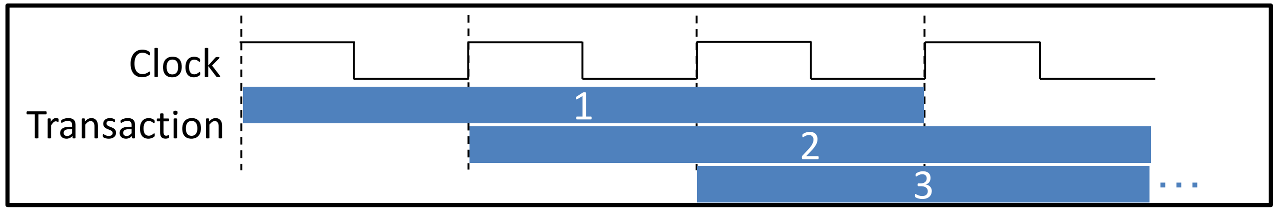


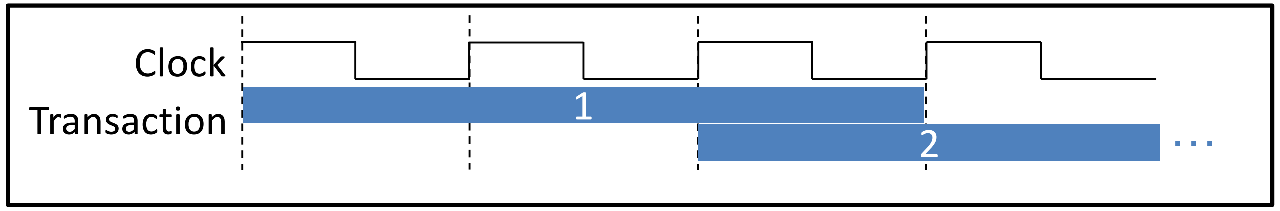
- ✓ Proves consistency of independently developed models
- ✓ Exhaustively verifies successive design refinements
- ✓ Great for corner case bugs
- ✓ Does not require testbenches, assertions, coverage

Building a Proof

- Relating transactions







- Model: Picking an arbitrary matched transaction
- Combinational - No state, just need to match inputs
- Fully Pipelined - State represented by previous transactions can be “All X’s”. Memories need to be mapped, but can be unconstrained
- Iterative - Memories need to be mapped, input state needs to be constrained and checked

Common Functions Verified with HECTOR Technology

Low effort		Medium effort		High effort
Integer	Logical operations Addition/Subtraction Absolute value Multiply (result < 20 bits)	**Integer**	Multiply Multiply accumulate	High radix SRT dividers
Cryptography	SHA Primitives AES single round	**Floating Point**	Multiply Multiply accumulate	Systolic array multiplier matrices
Floating Point	Convert Addition/Subtraction Multiply (half precision) Compare	**Integer / Floating Point**	Divide (SRT radix 2/4) Square root (SRT radix 2/4)	Streaming data operations
		May require advanced techniques **Implementation specific results**		**Requires advanced techniques** **Implementation specific results**

Case Study 1: 64-bit Floating Point

Function	**Approximate Runtime (minutes)**
Opcode 1	2
Opcode 2	1
Opcode 3	1
Opcode 4	3
Opcode 5	4
Opcode 6, ver. 1	410
Opcode 6, ver. 2	103

- All ISA defined rounding modes
- Numeric results, NaN handling, and exceptions
- Verified RTL equivalence to C simulation reference model

”In the case of **math modules…**, the formal verification work has **found more than 100 bugs that might not otherwise have been found until silicon!”**

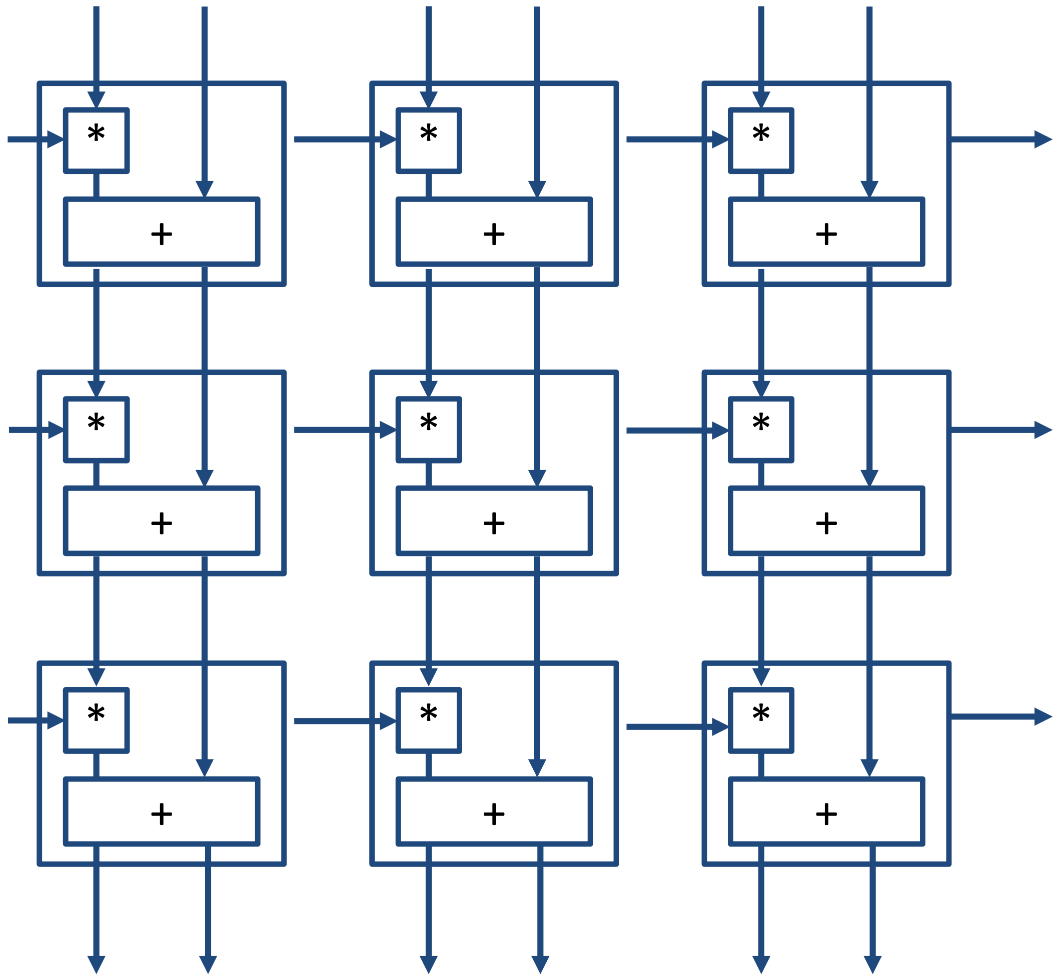
Case Study 2: 32-bit Floating Point

Function	Approximate Runtime (minutes)
FP ADD	5
FP SUB	5
FP MUL	5
FP MUL ADD	120
FP DIV	240
FP SQRT	240

- Verified RTL equivalence to SoftFloat reference model
- Multiple RTL bugs discovered
- HDPS, solveNB_division/solveNB_sqrt accelerates convergence of traditionally difficult problems

Case Study 3: AI Inference

- Complex MAC arrays
- Verify scalable from 3x3 to 128x128
  - Multiplier inputs 16 bit
  - Adder inputs 32 bit
- Datapath verification results
  - Matrix 8 x 8: proved in seconds
  - Matrix 32 x 32: proved in minutes
  - Matrix 64 x 64: proved in a few hours
  - Matrix 128 x 128: proved in 30hrs



What is the Manual Effort to Build a Proof?

– Easier problems just run.
  - First time users pleasantly surprised with quick setup

– Harder problem needs some manual guidance
  - Case split, abstractions, ….
  - Expert design domain knowledge is required
  - Formal background helps but not necessary

DATAPATH VERIFICATION BEHIND THE SCENES

How is DPV Different than Model Checking?

- Problems are often nonlocal
  - Model checking properties often only need a small part of the design for the proof

- Different engines are needed
  - Model Checking
    - The large number of control behaviors are typically the hardest problem
    - Individual computation steps are often not that hard to check

  - DPV: there is often a relatively manageable control aspect, but the correctness of single step computations can be really hard to show

- The need for multiprocessing is different
  - Model Checking: The main complexity is finding a chain of (optimizations, abstractions, final solver) that can deal with a particular problem

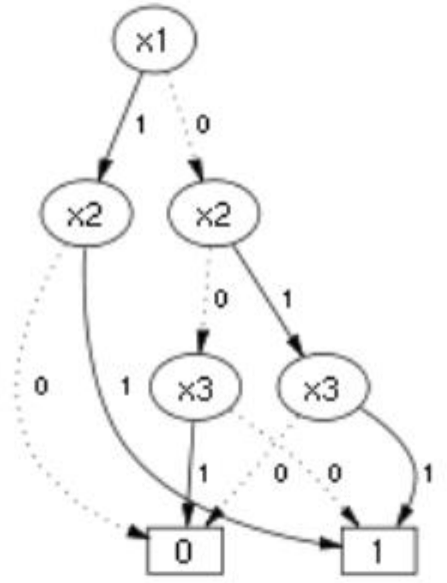
  - DPV: Typically have fewer engines but lots of case splitting and optimization

Fundamental Technology Building Blocks

- Binary Decision Diagrams and variants (BDDs)
- Satisfiability (SAT) and Satisfiability modulo theories (SMT) checkers.
- Polynomial verification procedures/Groebner bases
- Rewriting
- Proprietary leaf level solvers

Binary Decision Diagrams (BDDs)

- Graph data structures that represent Boolean functions
- Invented in the mid 1980s, enabled first wave of fc verification tools
- Unable to deal with whole larger multipliers
  - But extensions such as *BMDs tried to solve that
- Still relevant and useful if used judiciously



Satisfiability (SAT) solvers

- Takes as input a description of a single output circuit in some syntactic form
- Tries to find one way to the assign all leaf level inputs some Boolean value that makes the output evaluate to true
- Lots of design automation problems can be cast into this form
  - In particular, the comparison of two implementations of a multiple output circuit
- Unable to prove that two larger multiplier implementation are equivalent, if used monolithically
- Enabled the second wave of formal verification tools in the early 2000s

SMT

- SAT solvers can be extended into a platform for plugging together solvers for custom theories
  - Memories
  - Infinite precision arithmetic
  - Bit vectors
  - …..
- This is called "Satisfiability Modulo Theories" (SMT)
- This enabled boom of interest in the field of formal software verification that started in the early 2000s

Polynomial Verification

- Takes two polynomials as inputs and decide if they are the same

- Typically
  - one is a polynomial modelling low-level gates
  - the other one a polynomial expressing a higher level datapath specification

Rewriting

- Takes as input expressions in some formal language
- Generates a rewritten, hopefully simpler, expression
- Examples:
  - A(32) * (B(32) + C(32)) -> A(32)*B(32) + A(32)*C(32)
  - a&b | a -> a&b

Putting Leaf Level Solvers Together

- Proof tactics:
  - Case splitting
  - Abstraction
  - ….
- Speciality engines
  - Deep integrations of leaf level solvers
- Multi processing
  - Many of our users run a single check on a grid with 100s of processors

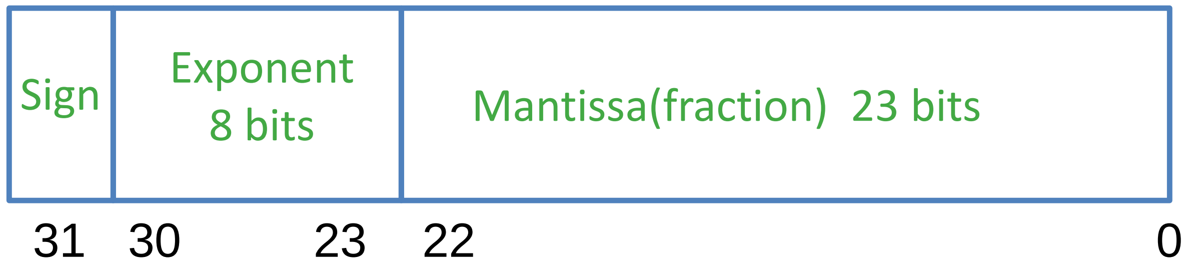
Case Splitting

- Most common technique to solve hard problems

- Breaks original proof into sub-proofs
  - Failing sub-proof is failure for larger proof
  - All passing sub-proofs true mean larger proof is true

- Useful case splits can be based on operation or based on microarchitecture

Brief Overview: 32-bit Floating Point Format



IEEE 754

exp	mantissa	Meaning
0	0	Zero +0 and -0 are possible.
0xFF	0	Infinity +inf and -inf are possible
0xFF	!= 0	NaN (Not a Number) Created by invalid operations
!= 0 != 0xFF		Normal number $1.Mant \times 2^{(exp-127)}$
0	!= 0	Denormal number $0.Mant \times 2^{-126}$

The value 1 is represented as: exp = 127 mantissa = 0 1.0×2^{0}
The value 0.5 is represented as: exp = 126 mantissa = 0 1.0×2^{-1}

Case Split Example - FP Multiply Accumulate

- Operation: (A * B) + C

- Possible case splits
  - Any input Infinity or NaN
  - Product term zero
  - Addend term zero
  - All combinations of subnormal/normal non-zero numbers

Other Convergence Techniques

- Internal equivalence points or relationships
  - Tool will look for these, but may not recognize complex relationships
- Assume-Guarantee
  - Split design up into stages to reduce complexity
- Bit-level to Word-level abstraction
  - Particularly on complex operations (multiply, divide, etc)
- Blackbox/cutpoint
  - Remove logic that does not participate in the result

Thank You