



# The Best Verification Strategy You've Never Heard Of

David Aerne, Amir Attarha, Harry Foster, Kurt Takara  
Siemens EDA

# SIEMENS



# Introduction

Harry Foster



# The Crisis

# Productivity Gap



1997 SEMATECH sets off an alarm about the Design Productivity Gap

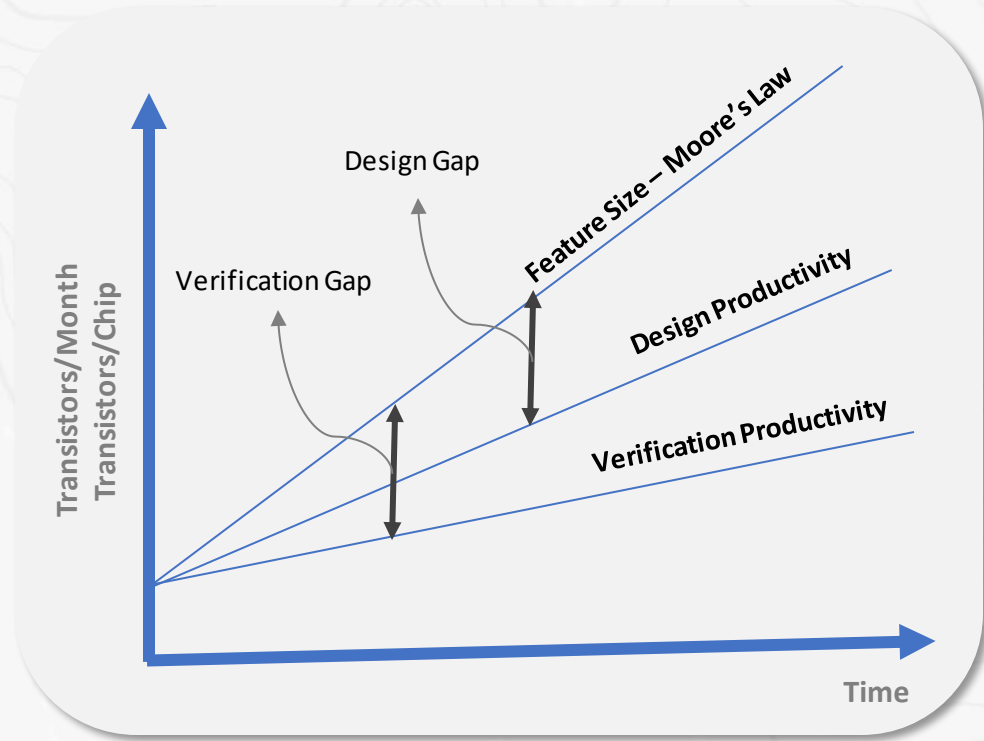
- IC manufacturing productivity gains increased 40% CAGR
- IC design productivity gains increased 20% CAGR



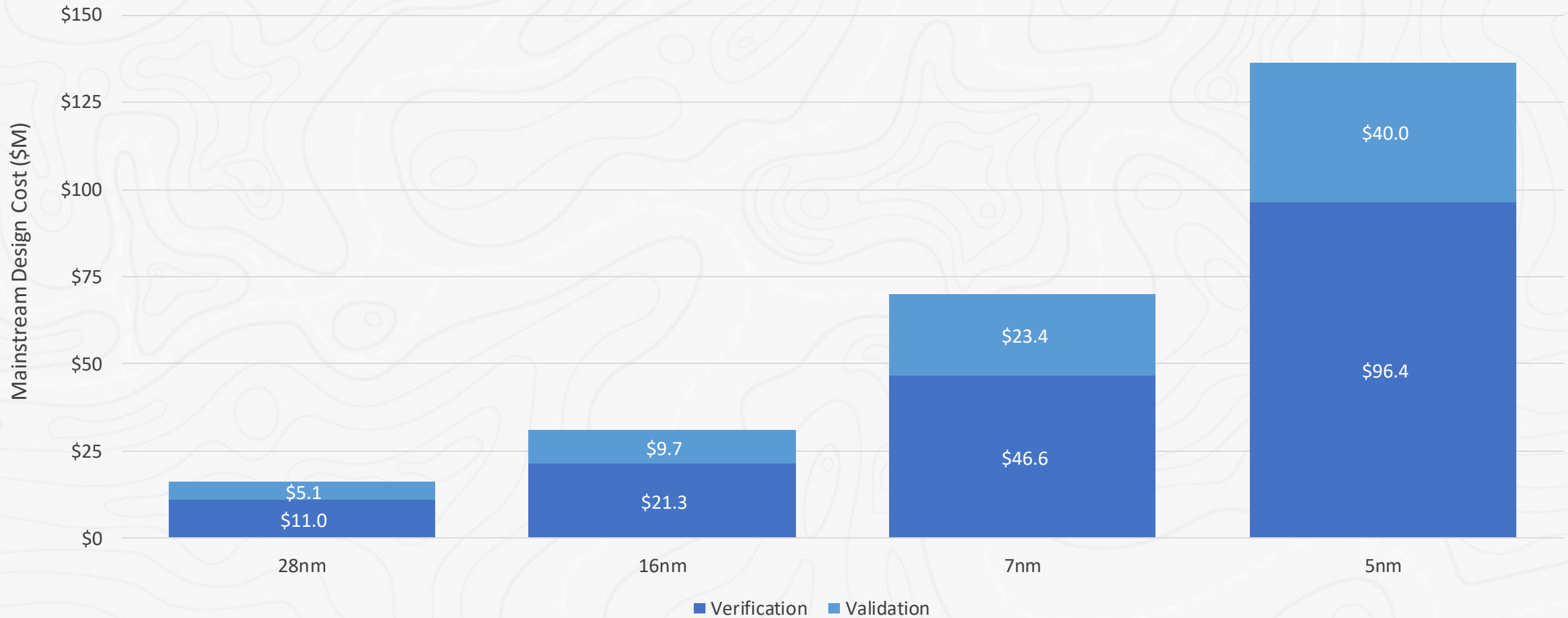
Design Productivity Gap solved through EDA improvements and IP Reuse



A more ominous productivity gap has emerged with respect to verification



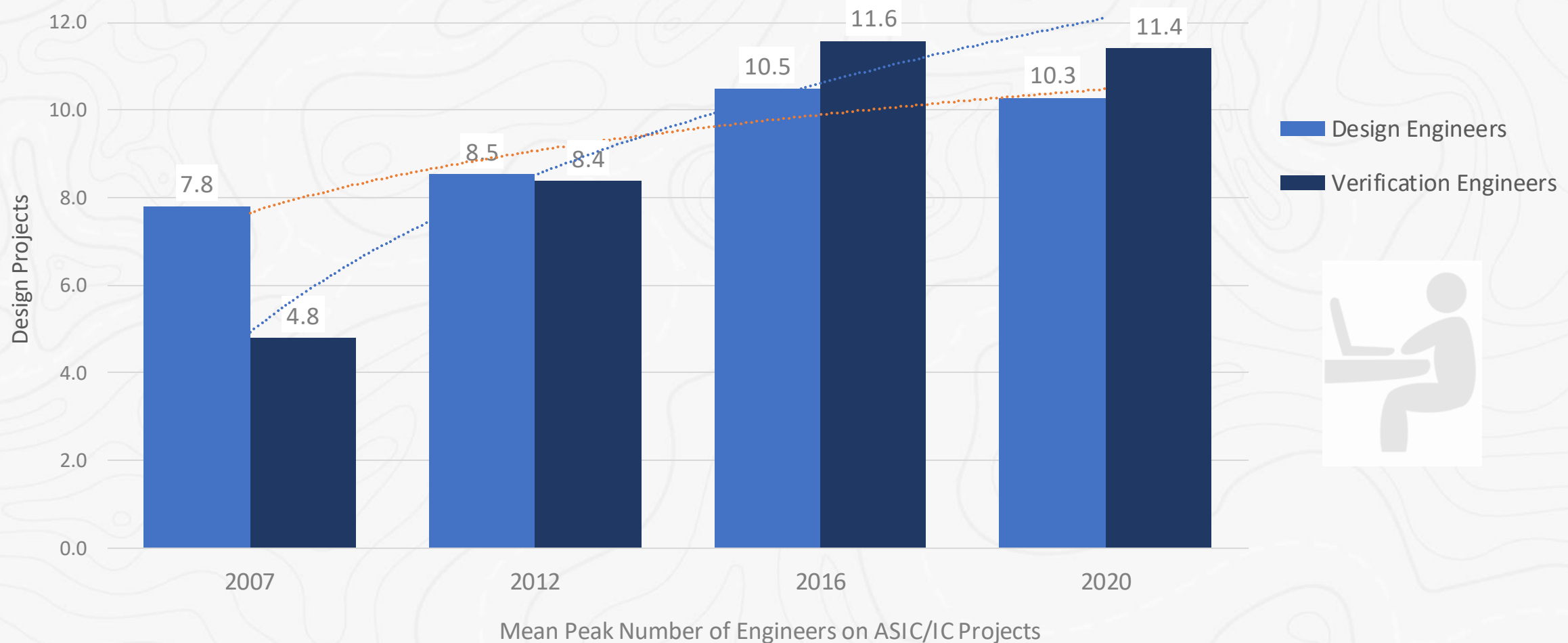
# IC Verification & Validation Cost by Process Feature Size



Source: IBS Report, Design Activities and Strategies Implications, July 2020

# Mean Peak Number of Engineers on an ASIC/IC Project

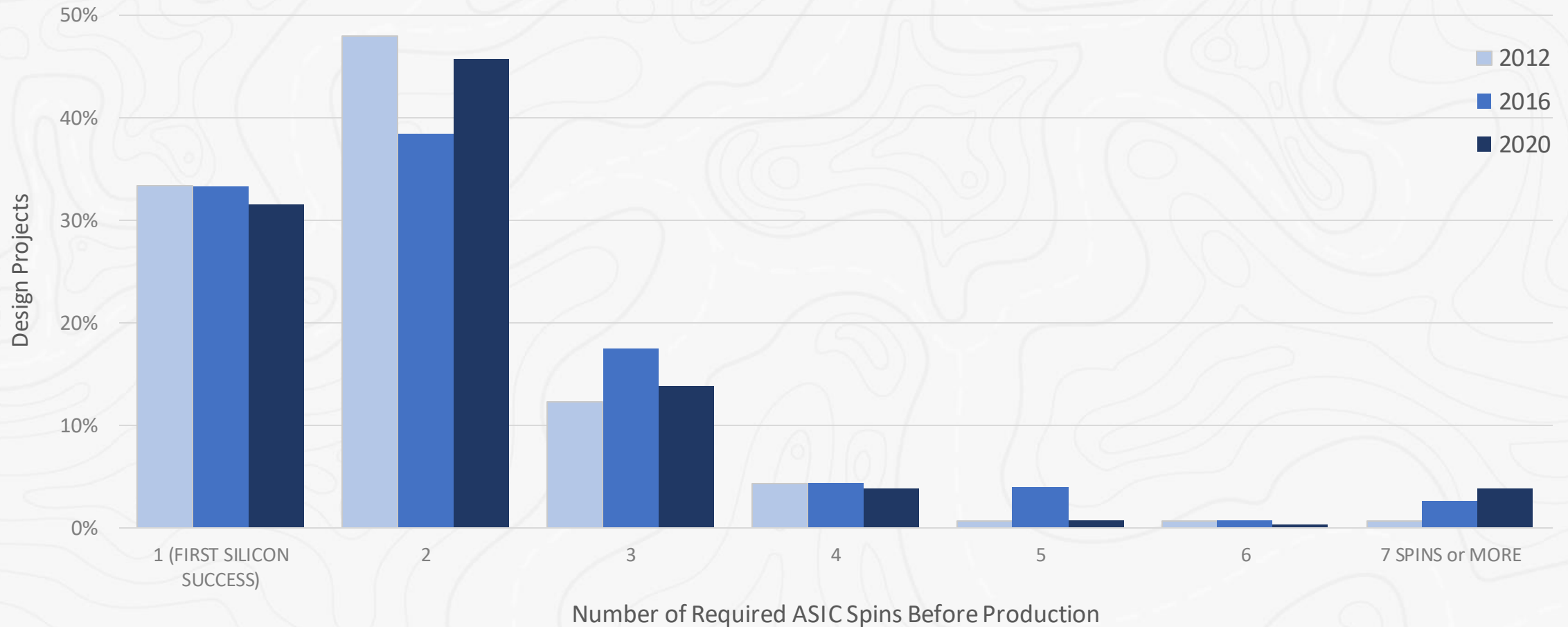
*Design engineers has increased by 32%, verification engineers has increased by 143%.*



Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

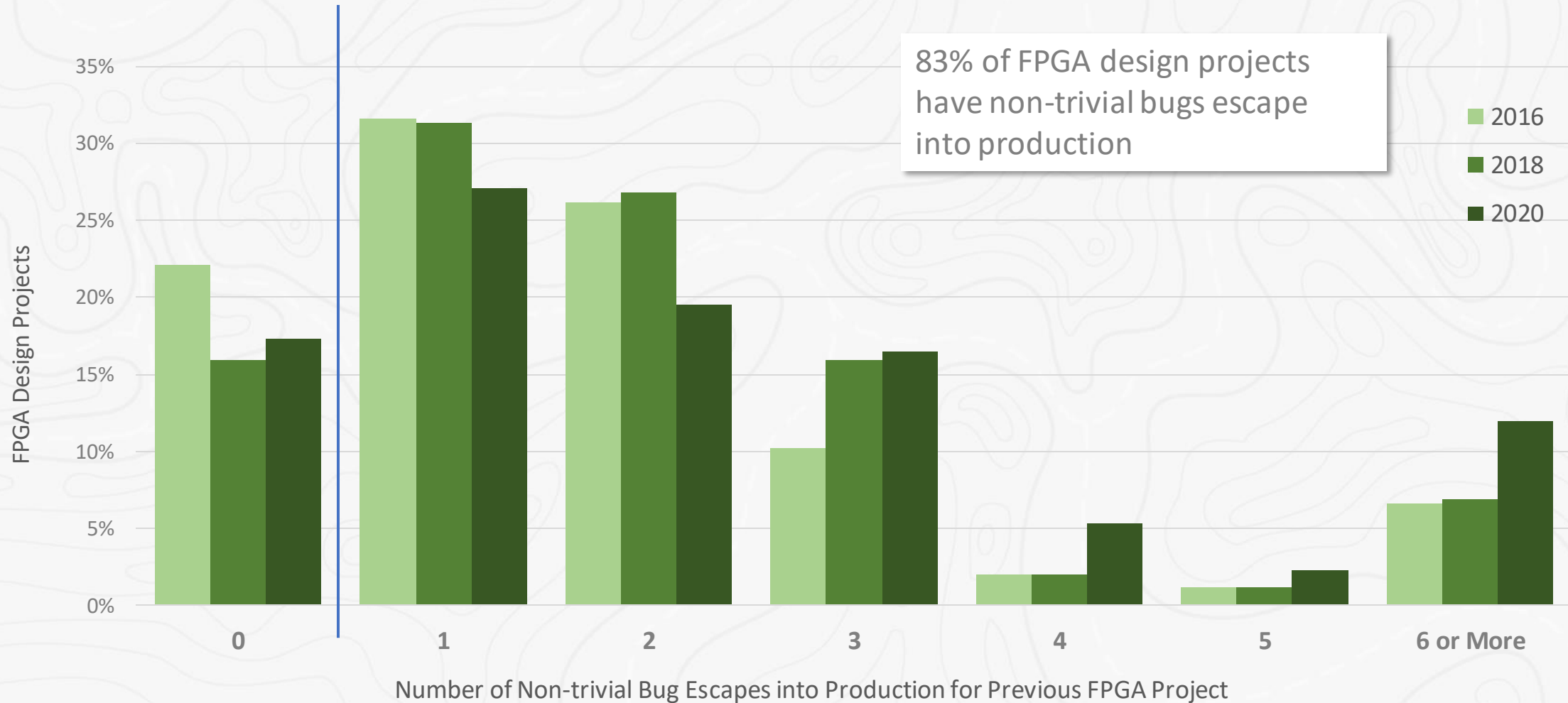


# ASIC Number of Required Spins Before Production



Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

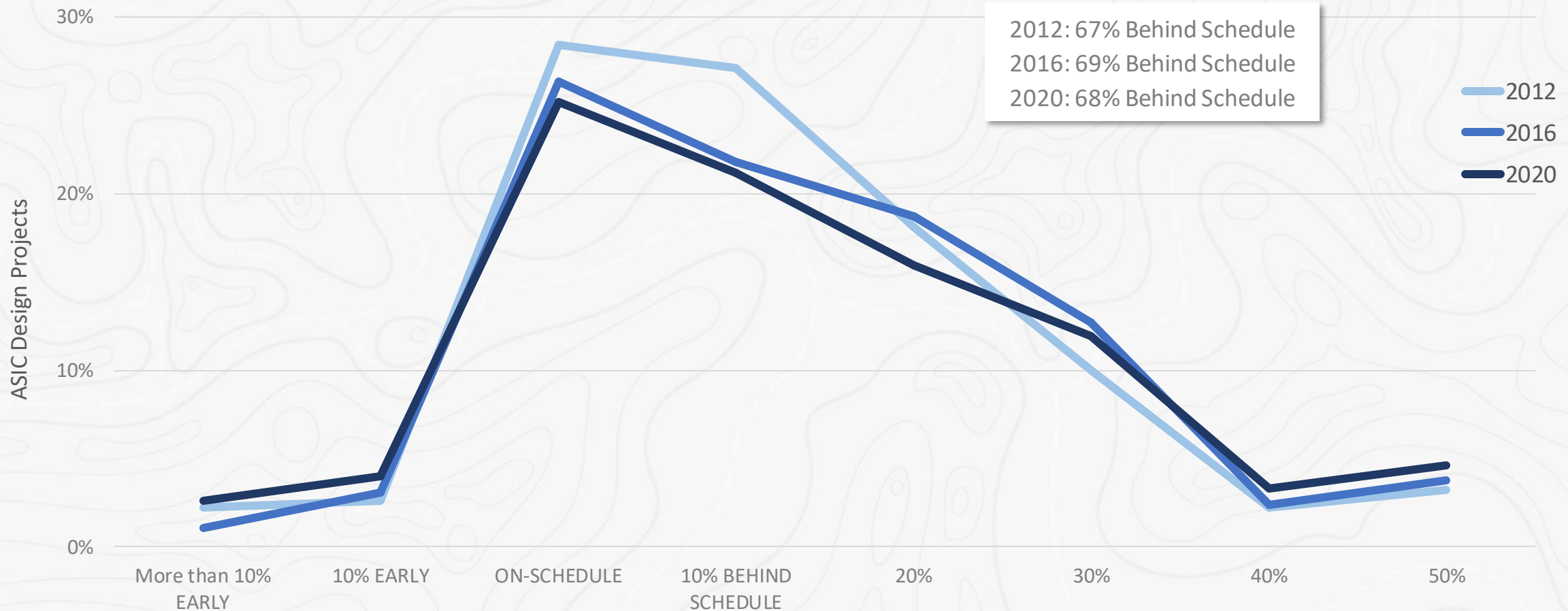
# Number of Non-trivial FPGA Bug Escapes into Production



Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study



# ASIC Completion to Project's Original Schedule



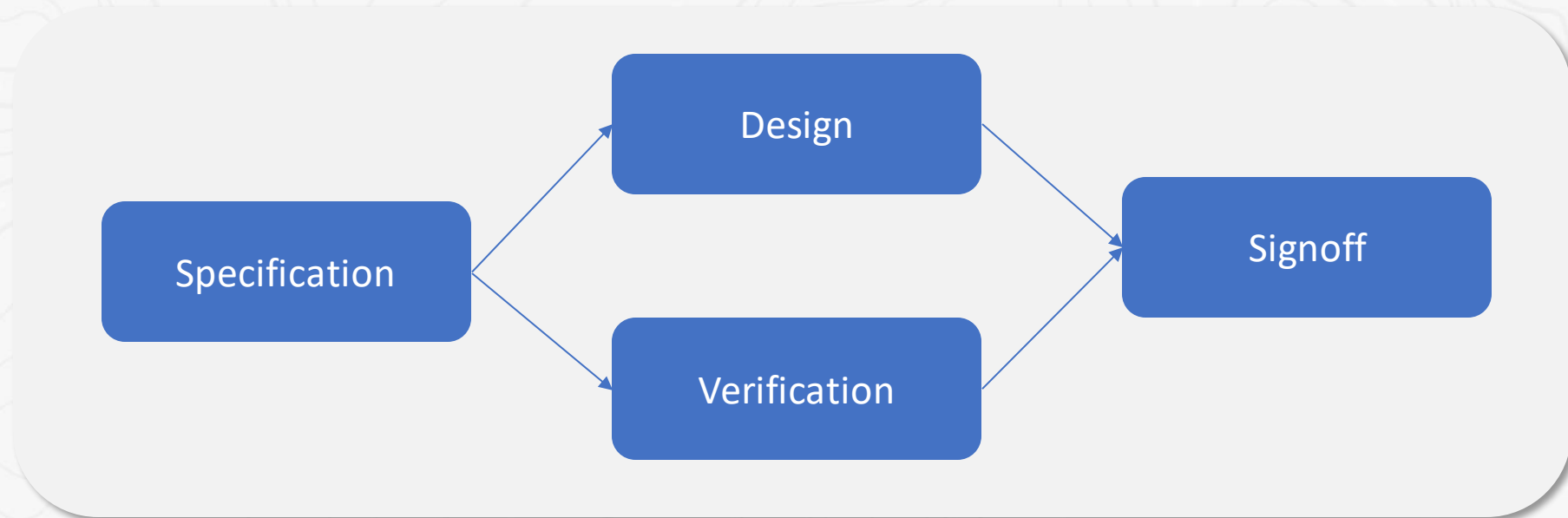
Actual ASIC design completion compared to project's original schedule

Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

# The Problem

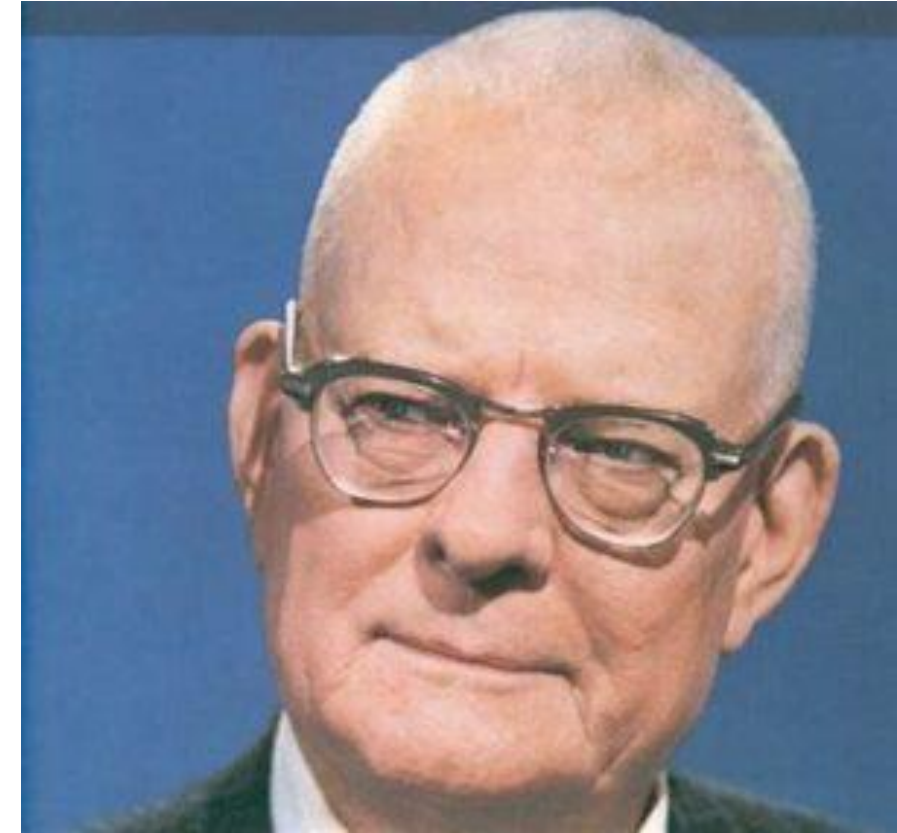
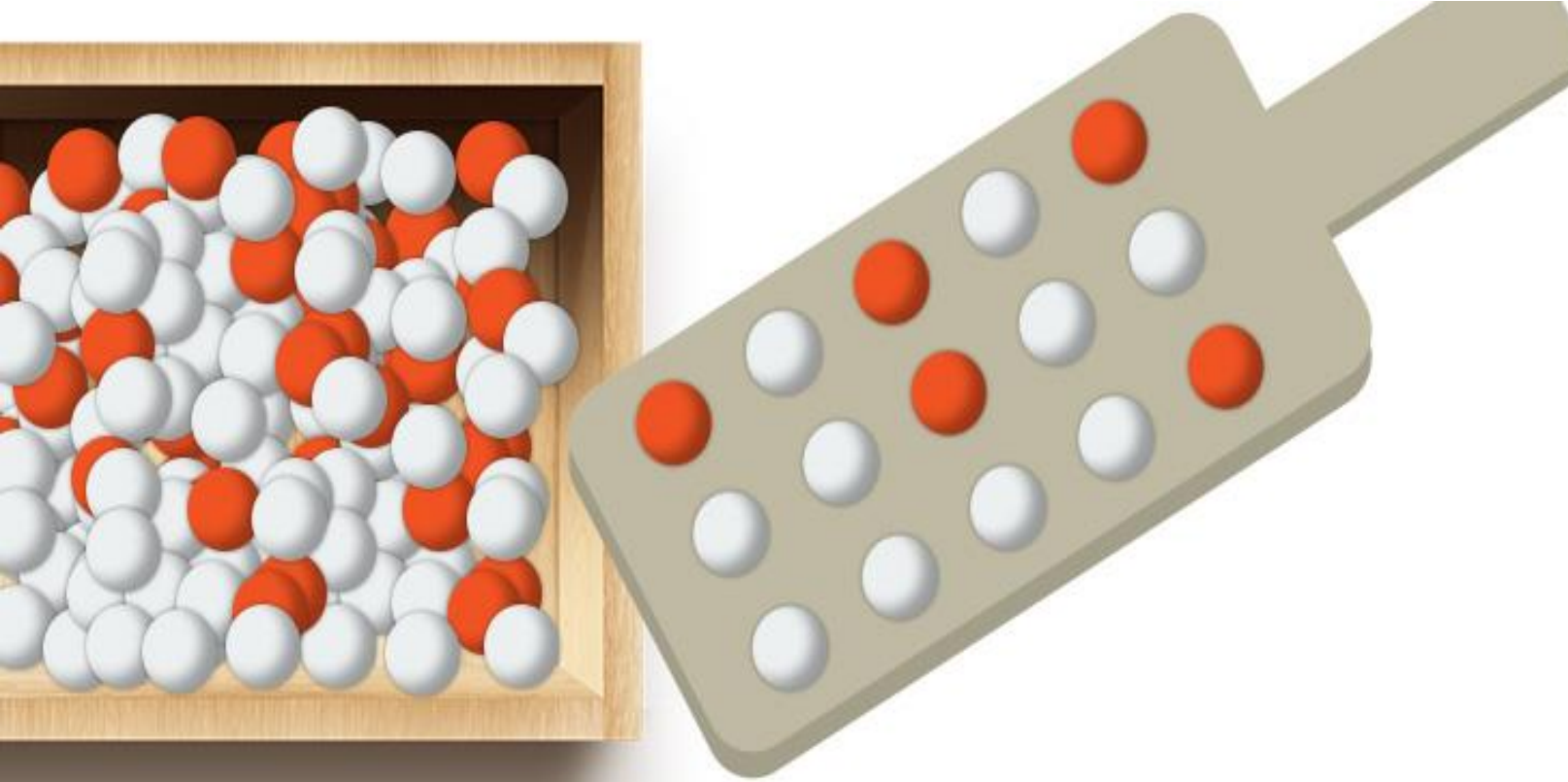
# Separation of teams

- Ensure an independent interpretation of the specification that would assist in flushing out design errors
- Increased complexity of verification environments required unique engineering skills



- Fallacy that quality can be verified into a product

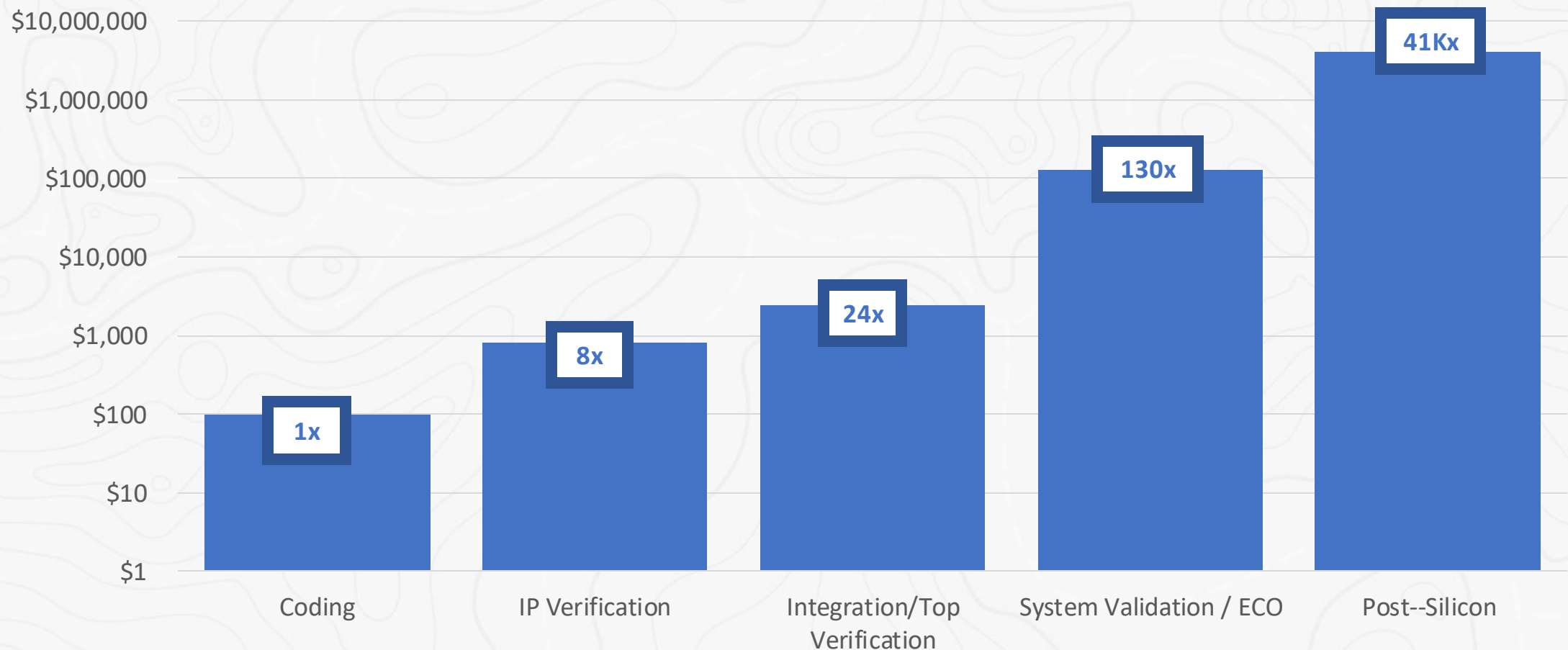
Quality cannot be inspected into a product; it must be built into it.



W. Edwards Deming  
*Father of Statistical Process Control*

# Cost Multipliers

*Finding and Fixing a Bug at Various Development Stages for a 5 nm ASIC*





# The Prescription





# Improve RTL Quality While Reducing Bug Density with Intent Focused Insight

*Design+Intent*

# Three Pillars of a Design+Intent Methodology



## Produce

Produce correct intent by construction



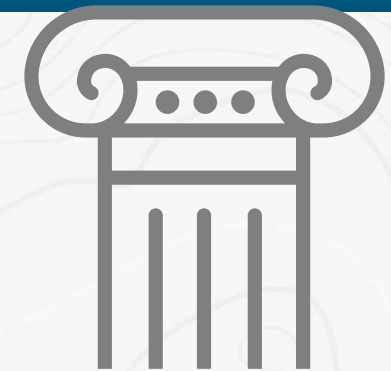
## Prove

Prove intent is met



## Protect

Protect intent throughout development lifecycle



# Bug Prevention through HLL Design

*15-50 Bugs per 1000 Lines of Code*

100 lines of HLL is equivalent to 1000 lines of RTL



Expect a 10x reduction in the average number of bugs

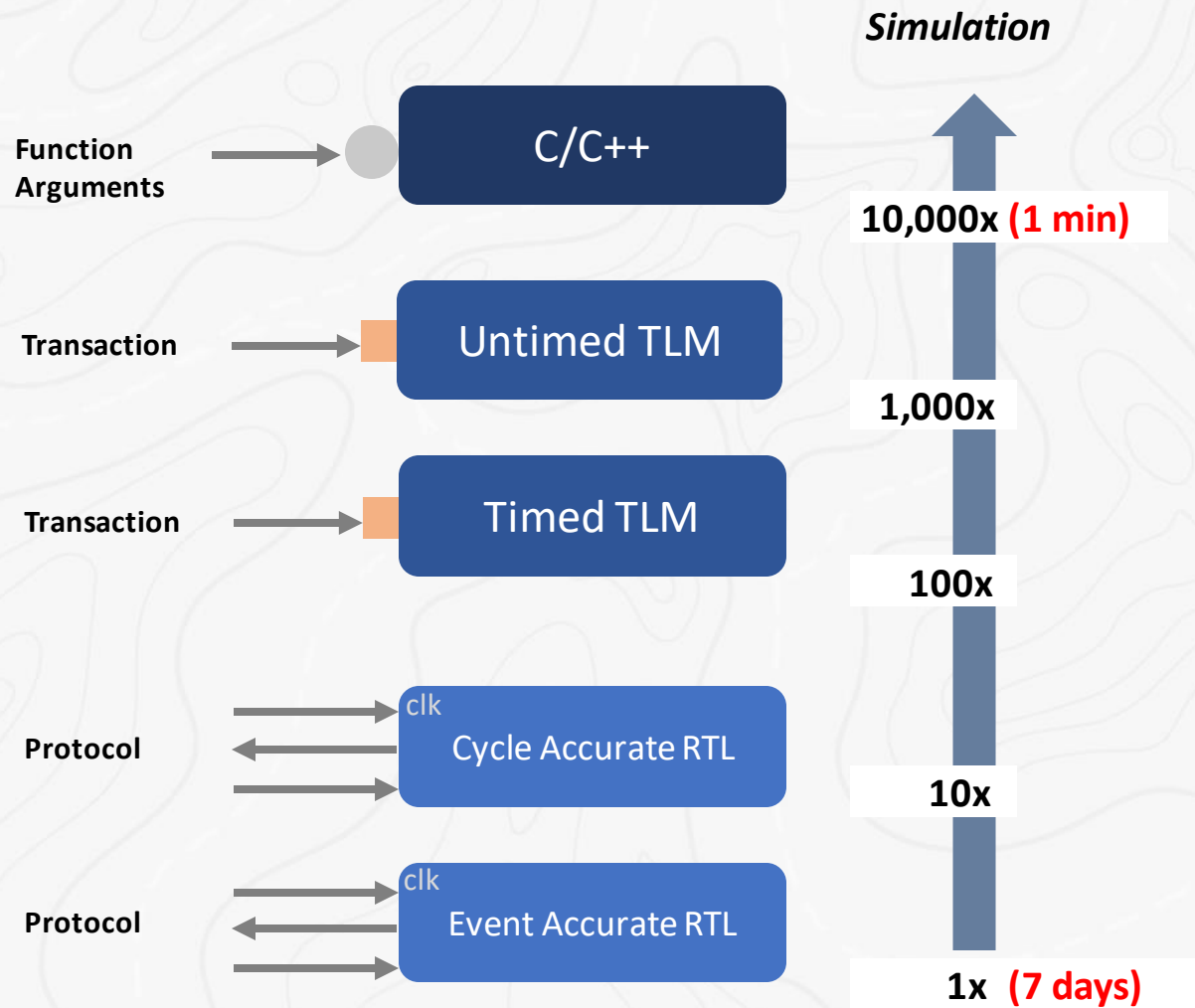


1–5 bugs for HLL design vs. 15–50 bugs for equivalent RTL design

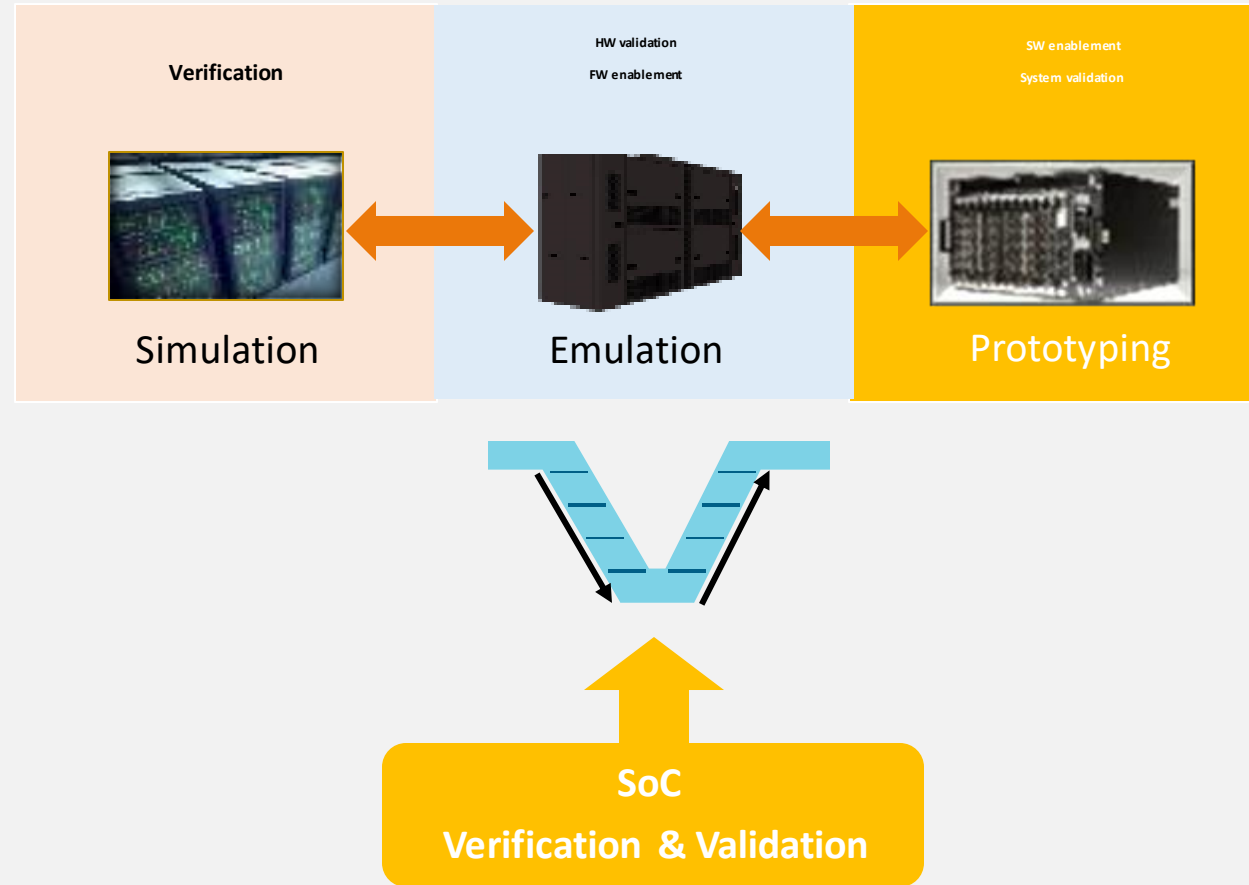


# Closing The Verification Gap

## Abstraction

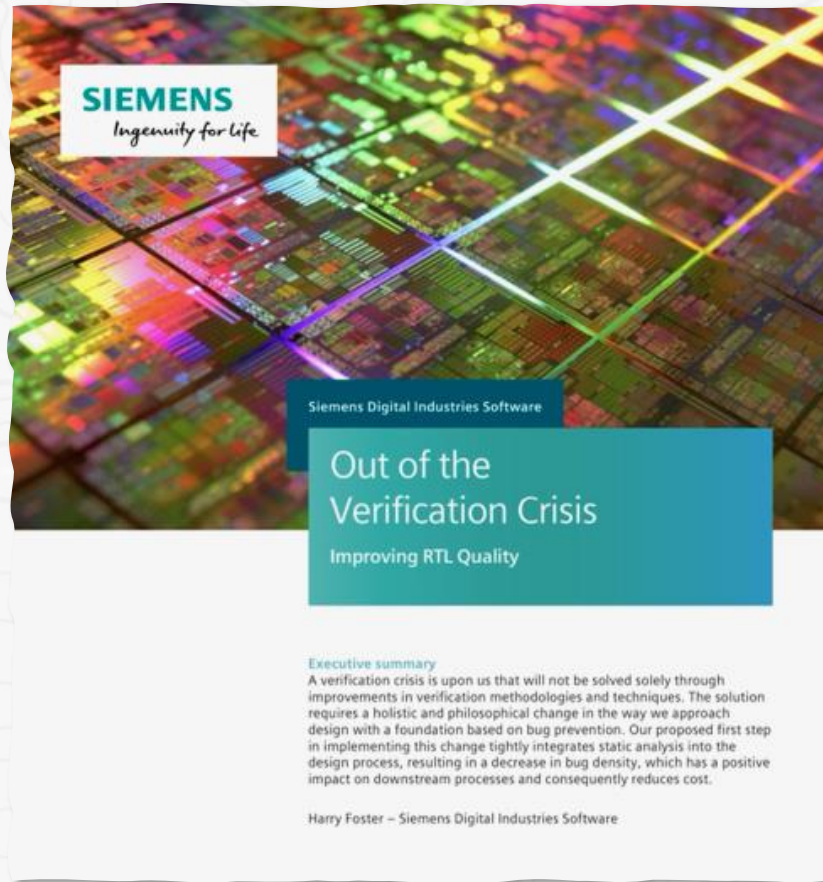


# Beyond Verification: The SoC Lifecycle



# Siemens EDA Whitepaper:

## *Out of the Verification Crisis: Improving RTL Quality*



<https://resources.sw.siemens.com/en-US/white-paper-out-of-the-verification-crisis-improving-rtl-quality>





# Produce

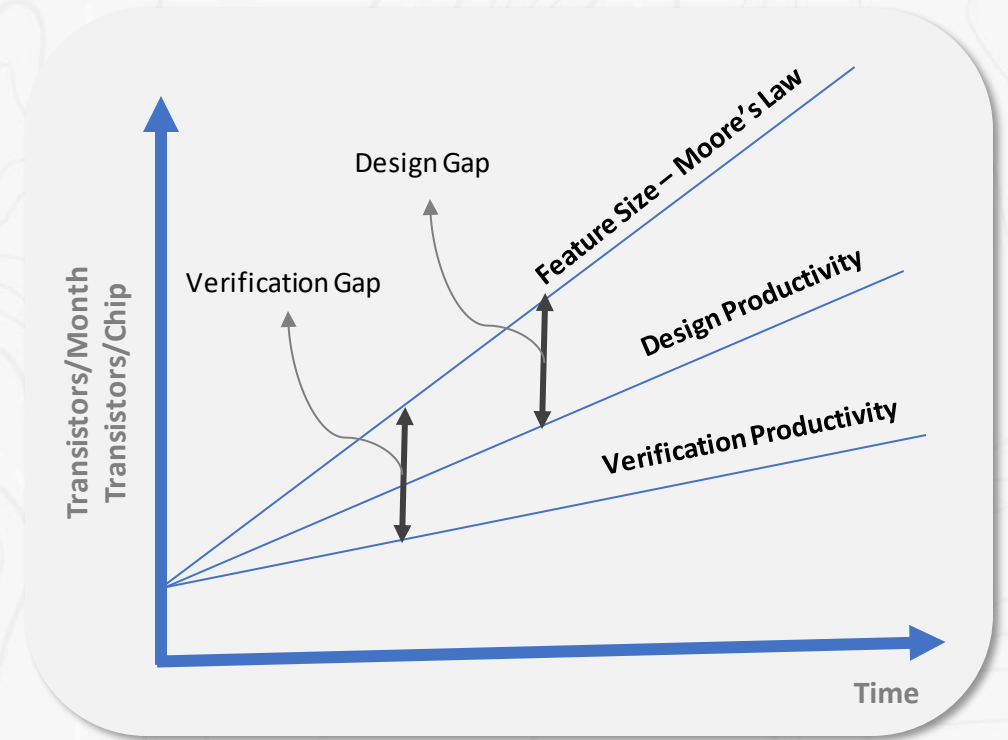
Produce correct intent by construction

David Aerne



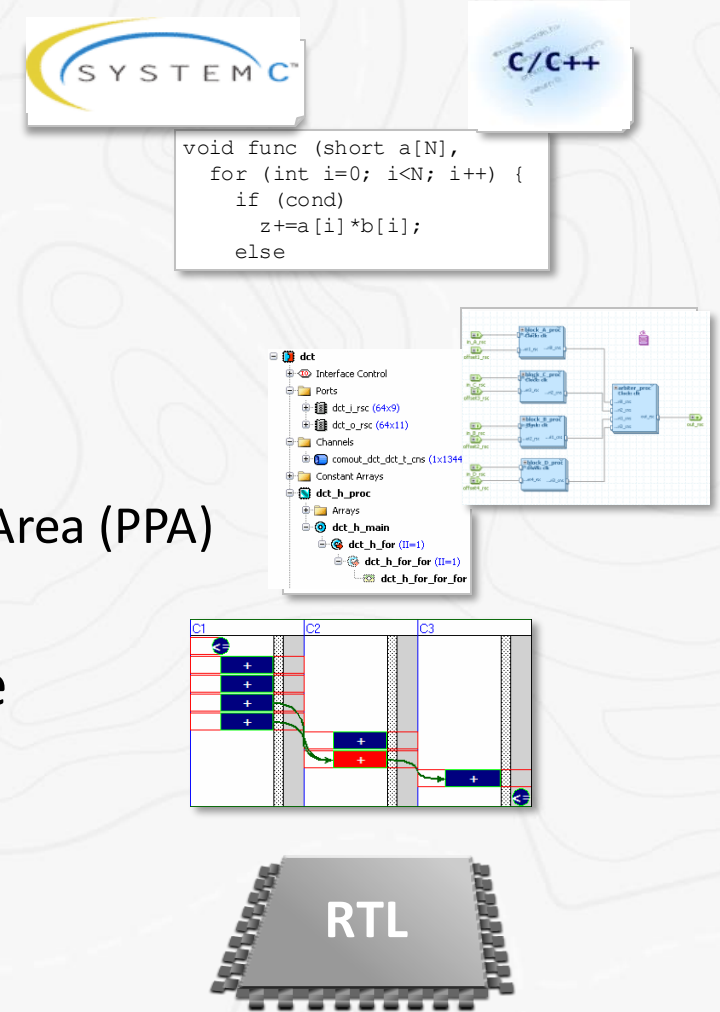
# Addressing the Productivity Gaps

- High-Level Synthesis (HLS)
  - Design at a higher level of abstraction
  - Rapid architecture exploration
  - Target Technology Library to meet PPA goals
- High-Level Verification (HLV)
  - Using known and trusted techniques
  - Speedup compared to RTL
  - Efficient & Predictable post-HLS RTL verification signoff



# HLS for Rapid Algorithm to HW

- Accelerate design time with higher level of abstraction
  - 5-10X less code than RTL
  - Faster verification cycles, 30-1000x compared to RTL
  - New features added in days not weeks
- Quickly evaluate power and performance of algorithms
  - Rapidly explore multiple options for optimal Power Performance Area (PPA)
- Enable late functional changes without impacting schedule
  - Algorithms can be easily modified and regenerated
  - New technology nodes are easy (or FPGA to ASIC)



# HLS synthesizes C++ and SystemC to RTL

- SoC design is complex, one challenge is timely creation of optimal hand-crafted RTL
- Alternatively, HLS to produce correct-by-construction RTL

```
void simpleDesign(<function interface variables>){  
  <function body>  
}
```

```
class simpleDesign{  
  ...  
  public:  
  void run(<method interface variables>){  
    <method body>  
  }  
};
```

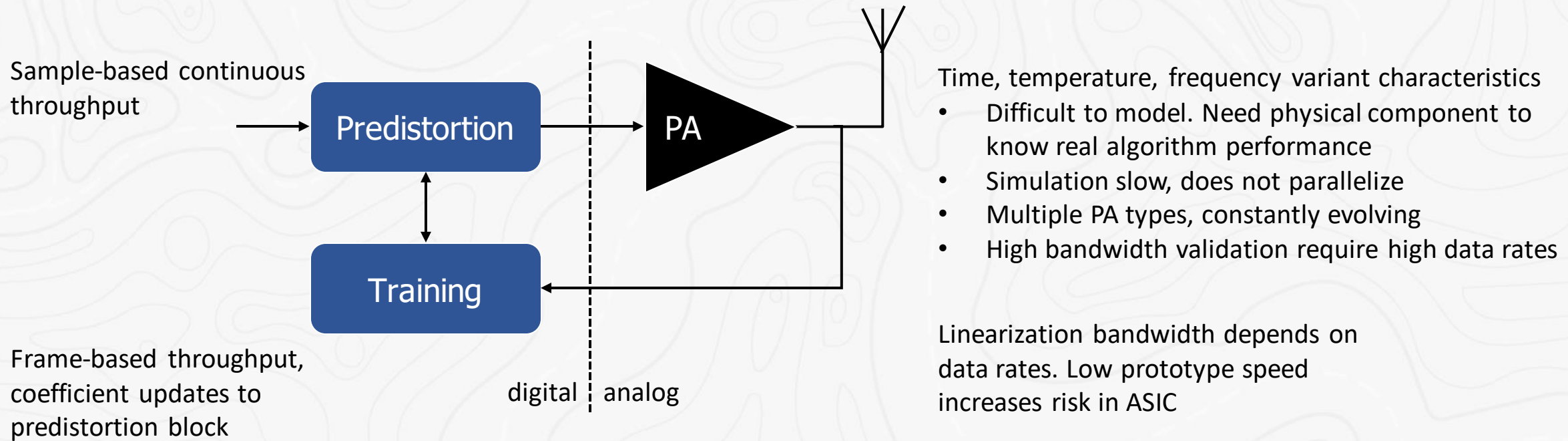
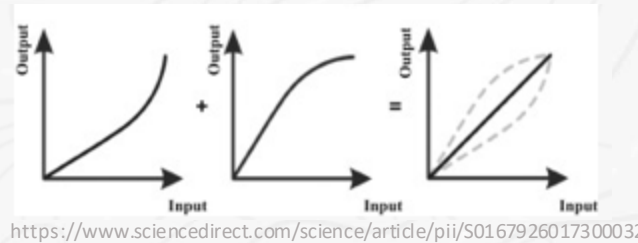
```
class simpleDesign : public sc_module {  
  <module ports>  
  SC_CTOR(simpleDesign){  
    SC_THREAD(run)  
  }  
  void run(){  
    <function body>  
  }  
};
```

```
module simpleDesign ( <module ports> );  
  always@(posedge clk)  
    begin  
      <module body>  
    end  
endmodule
```



# Demonstration Vehicle

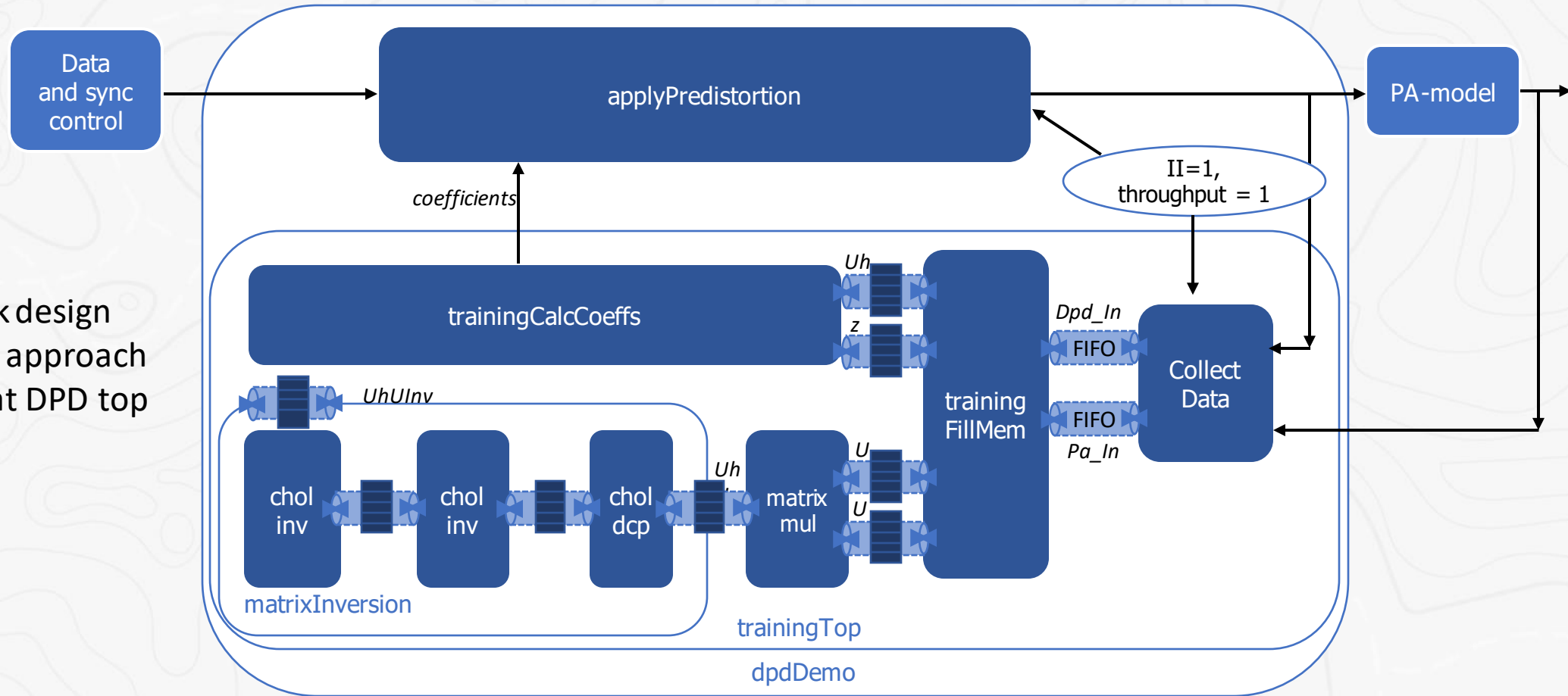
## Digital Pre-Distortion (DPD) concept



# Demonstration Vehicle

Digital Pre-Distortion (DPD) design + tb

- Multi-block design
- Bottom-up approach
- Assemble at DPD top





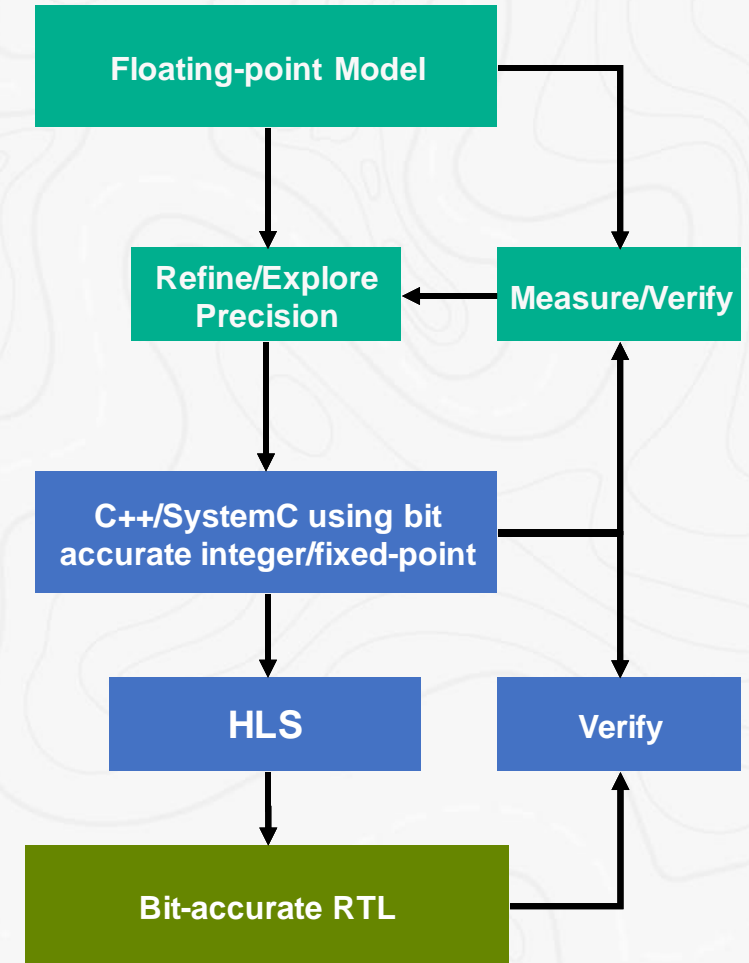
# Quickly Simulate Real HW Behavior in C++/SystemC

- Many various implementations for DPD
  - How to know which is best suited for application?
- Model bit-accurate precision in C++
  - Directly measure and observe the effects of quantization
  - Not limited to power-of-two bit-widths
  - Plug back into environment for verification
- Rapid simulation of true hardware behavior
  - 30x to 1000x faster than RTL
  - Simulate in minutes/hours vs. hours/days/weeks



Horizon  
Robotics

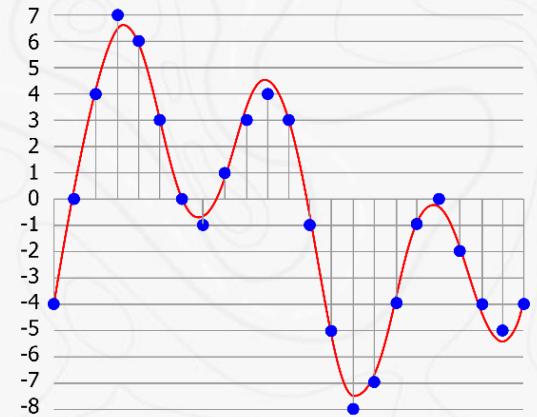
“Advantage is ability to compare C reference model with HLS C HW model”



# Bit-Accurate Data Types are a necessity

Required as modeling actual hardware

- Choice between AC Datatypes and SystemC, both are public domain
  - <https://hlslibs.org>
  - <https://github.com/accellera-official/systemc/>
- HLS Designers generally prefer AC types, even for SystemC HLS
  - AC types simulate faster than SystemC types, even in SystemC Designs
    - Especially the Fixed-Point types
- Include optional rounding and saturation modes



```
// Data types
typedef ac_fixed<12,1,true> IO_FXP_TYPE;
typedef ac_fixed<12,1,true,AC_RND_INF,AC_SAT> IO_FXP_TYPE_RND_SAT;
typedef ac_complex<IO_FXP_TYPE> IO_TYPE;
typedef ac_complex<IO_FXP_TYPE_RND_SAT> IO_TYPE_RND_SAT;

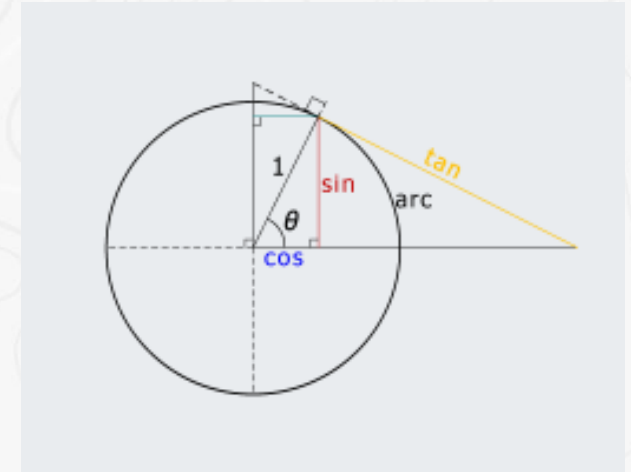
"src/dpdTypedefs.h" 239 lines --13%--
```

DPD

# HLS IP Libraries

Provide needed productivity gain

- HLS Designers rely on pre-established IP libraries
  - AC Math synthesizable C++ operations common in DSP applications
  - AC DSP synthesizable C++ objects for common DSP operations
- AC Math, AC DSP and AC ML are all public domain
  - <https://hlslibs.org>
- Customizable
  - Data type support built-in
  - PPA tradeoffs



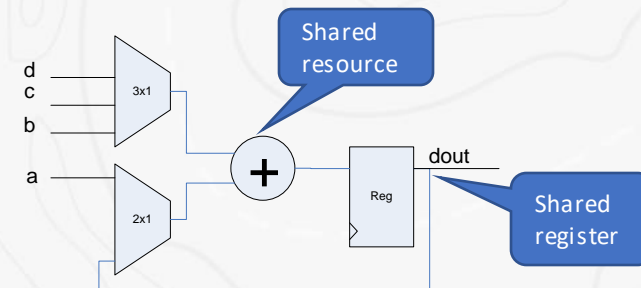
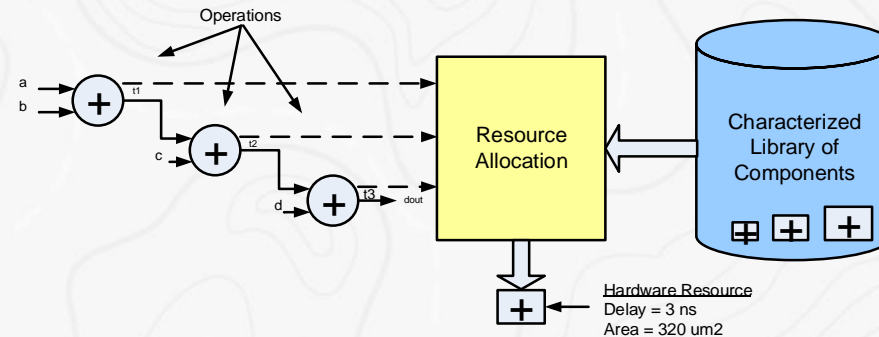
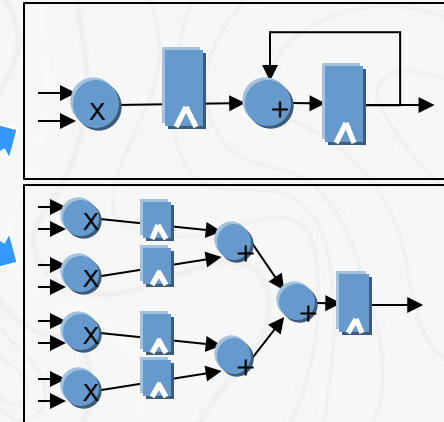
```
class complexAbs_c{
public:
    complexAbs_c(){}
    ABS_TYPE calcAbs(IO_TYPE &inputSample){
        ABS_TYPE_TEMP absSampleTemp, tempForSqrt = inputSample.mag_sqr();
        ac_math::ac_sqrt_pwl(tempForSqrt, absSampleTemp); // sqrt()
        ABS_TYPE absSample = (ABS_TYPE_SAT)absSampleTemp; // Rnd&Sat
        return(absSample);
    }
};
"src/dpdDemo.h" 622 lines --4%--
```

# HLS Makes PPA Goals Achievable

- Loop optimizations
  - Unrolling
  - Pipelining
  - Automatic merging
- Scheduling
  - Automatic timing closure based on target technology
- Register and Resource sharing
  - Automatic lifetime and mutual exclusivity analysis and optimization

```
for (int i=0;i<4;i++) {  
    acc += data_in[i] * coef_in[i] ;  
}
```

Architecture Constraints





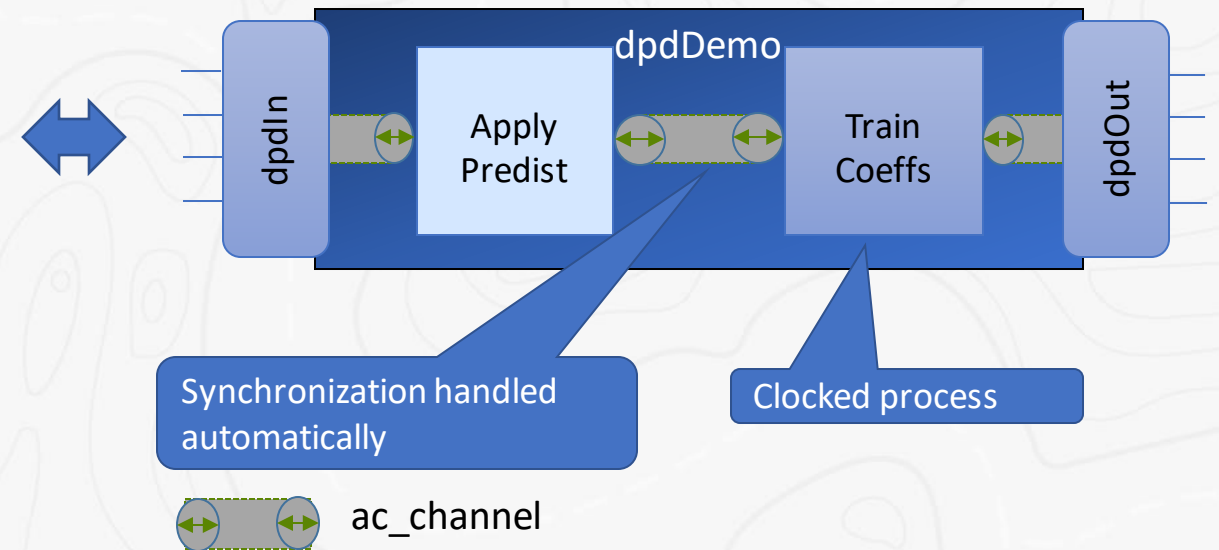
# HLS Builds Complex Multi-block Systems

- HW architectures often require multiple concurrent processes to meet performance
- Untimed HLS Builds Parallel Concurrent Processes from Sequential C++ Classes
  - Easy to design and debug
- Connect HLS blocks together using channels
  - Channels mapped to fifo's in post-HLS RTL

```
#pragma hls_design top
class dpdDemo_c{

    // Object(s)
    applyPredistortion_c applyPred;
    trainingTop_c        dpdTrain;

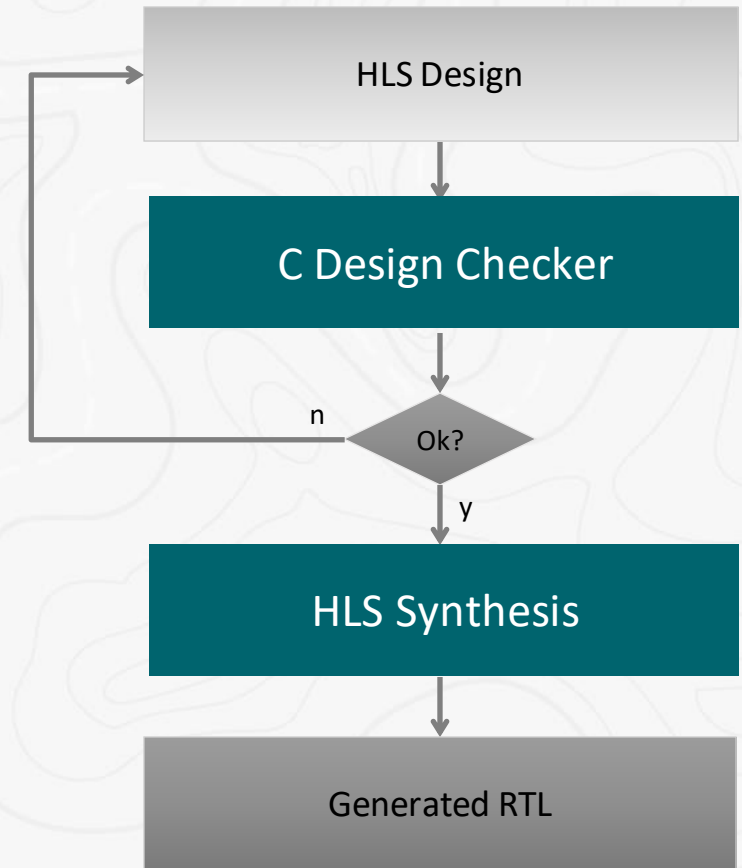
    // Interconnect(s)
    ac_channel<IO_N_TYPE> dpdOutConnect;
    ac_channel<DPD_COEFF_STRUCT> coeffConnect;
    "src/dpdDemo.h" 628 lines --92%--
}
```



# C Design Checker

Static & Formal analysis to find issues early

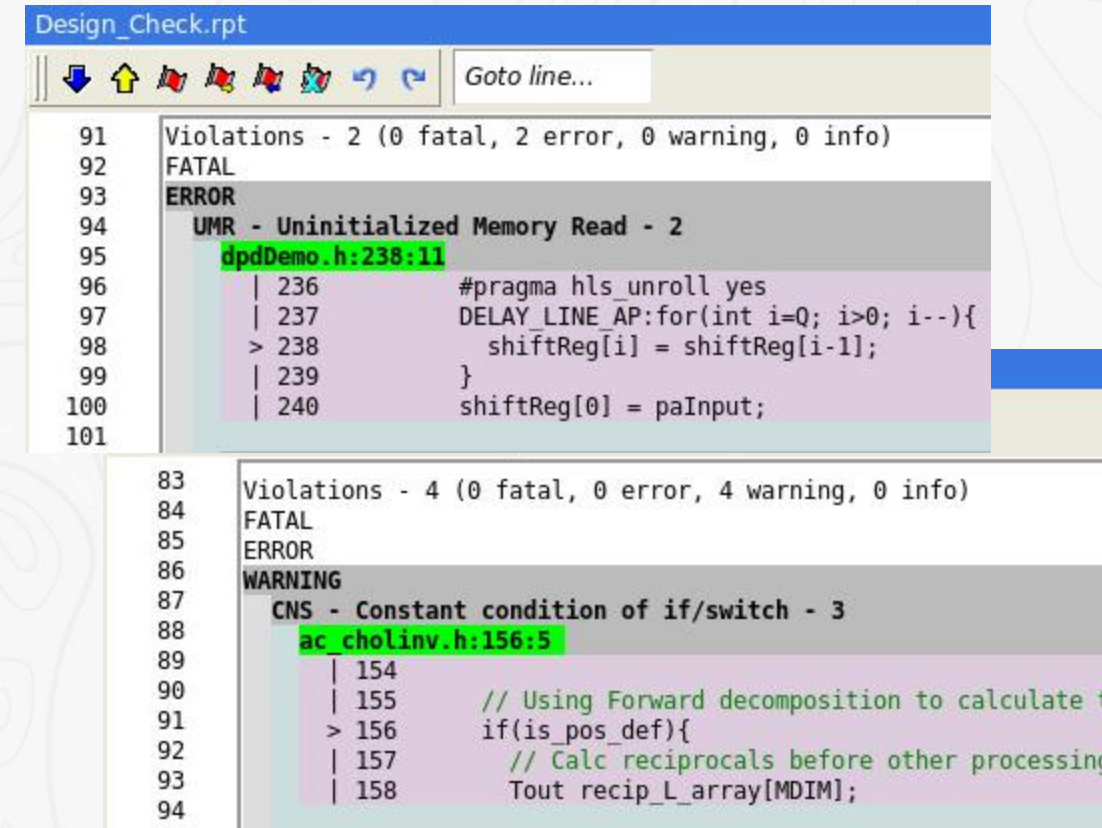
- Quickly and easily find coding bugs and errors in HLS source before synthesis or simulation
- Some C++ language behavior can be too ambiguous for describing hardware
  - Leads to mismatches between C++ and RTL sim
  - Inefficient to debug in dynamic RTL simulation
- Collection of Quality of Results (QofR) checks, static lint checks, and formal property checking
  - e.g. Out of bounds array reads and writes (ABR, ABW) and uninitialized memory reads (UMR)





# C Design Checker

- Results on DPD from running “Sim Mismatch” and “QofR” modes
- Reports analyzed within the Catapult GUI
  - Violations are cross-linked to design source
- Template Waiver File is automatically generated
  - Can be edited and reused for future runs
  - Can also specify waivers directly in source
- Constraints supported as ‘assume’ pragmas



The screenshot displays the 'Design\_Check.rpt' report window. It shows two sections of violations. The first section, starting at line 91, reports 2 violations (0 fatal, 2 error, 0 warning, 0 info). It highlights a 'FATAL' error at line 92 and an 'ERROR' at line 93: 'UMR - Uninitialized Memory Read - 2' in 'dpdDemo.h:238:11'. The code snippet shows a loop from line 236 to 240. The second section, starting at line 83, reports 4 violations (0 fatal, 0 error, 4 warning, 0 info). It highlights a 'FATAL' error at line 84 and a 'WARNING' at line 85: 'CNS - Constant condition of if/switch - 3' in 'ac\_cholinv.h:156:5'. The code snippet shows an if statement starting at line 156.

```
91 Violations - 2 (0 fatal, 2 error, 0 warning, 0 info)
92 FATAL
93 ERROR
94   UMR - Uninitialized Memory Read - 2
95   dpdDemo.h:238:11
96   | 236      #pragma hls_unroll yes
97   | 237      DELAY_LINE_AP:for(int i=0; i>0; i--){
98   > 238          shiftReg[i] = shiftReg[i-1];
99   | 239      }
100   | 240      shiftReg[0] = paInput;
101
83 Violations - 4 (0 fatal, 0 error, 4 warning, 0 info)
84 FATAL
85 ERROR
86 WARNING
87   CNS - Constant condition of if/switch - 3
88   ac_cholinv.h:156:5
89   | 154
90   | 155      // Using Forward decomposition to calculate
91   > 156      if(is_pos_def){
92   | 157          // Calc reciprocals before other processing
93   | 158          Tout recip_L_array[MDIM];
94
```

*Clean HLS design source results in less debug of post-HLS RTL*

# Properties in HLS

Deploy properties to catch issues early

- Catapult HLS supports immediate assertions & cover properties
  - HLS C++ and SystemC
- Properties are propagated from HLS source to RTL
- Assertions in generated RTL
  - SVA, PSL, or OVL

```
#include <ac_assert.h>

#pragma hls_design top
uint16 alu(uint8 a, uint8 b, opcode_t
opcode) {
    uint16 r = 0;
    switch(opcode) {
        case ADD:
            r = a+b;
            break;
        case SUB:
            assert(a>=b); // no negative results
            r = a-b;
            break;
        case DIV:
            assert(b!=0); // no divide-by-zero
            r = a/b;
            break;
    }

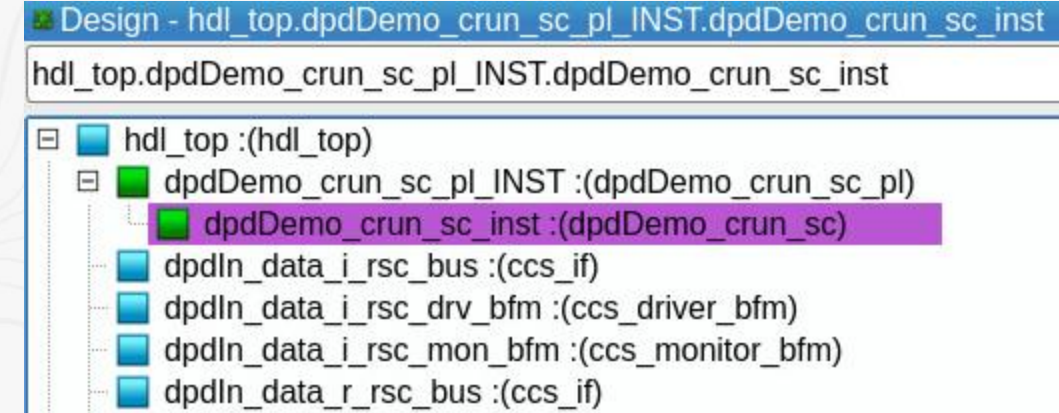
    // Cover all of the possible opcodes
    cover((opcode==ADD));
    cover((opcode==SUB));
    cover((opcode==DIV));

    return r;
}
```

*Applying common RTL debug and verification techniques to HLS design source*

# Metric Driven Dynamic HLV

- Supported in a wide range of tb environments
  - C++, OSCI, MATLAB®, Python, SV/UVM, etc.
- Using known and trusted verification techniques
  - Supporting wide range of verification requirements
- DUT at a higher level of abstraction
  - Fewer lines of code
  - Simulations run faster
- Re-use for predictable and efficient post-HLS RTL signoff



“99% of the functional bugs found in (HLS-ready) C++ before running any RTL simulation”

Hot Chips 2021

<https://hc33.hotchips.org/>



# C Coverage

RTL-like coverage for HLS Design Source

- Bring RTL coverage into HLS world
  - C++ and SystemC design source
- Match coverage concepts from RTL
  - Statement, branch, expression
  - Functional coverage including covergroups, coverpoints, bins and crosses
- HLS-aware code coverage
  - Function inlining
  - Loop unrolling
  - Array access coverage

Coverage Summary By Instance ( 78.47% )				
Instance ↑	Branches	Expressions	Statements	Total
Search...	Search...	Search...	Search...	Search...
Total	75.3%	66.66%	93.44%	78.47%
\dpdDemo_c::run#0	-	-	100%	100%
\applyPredistortion::run#0	58.82%	100%	86.6%	81.8%
\dpdTraining::run#0	87.23%	60%	98.55%	81.92%

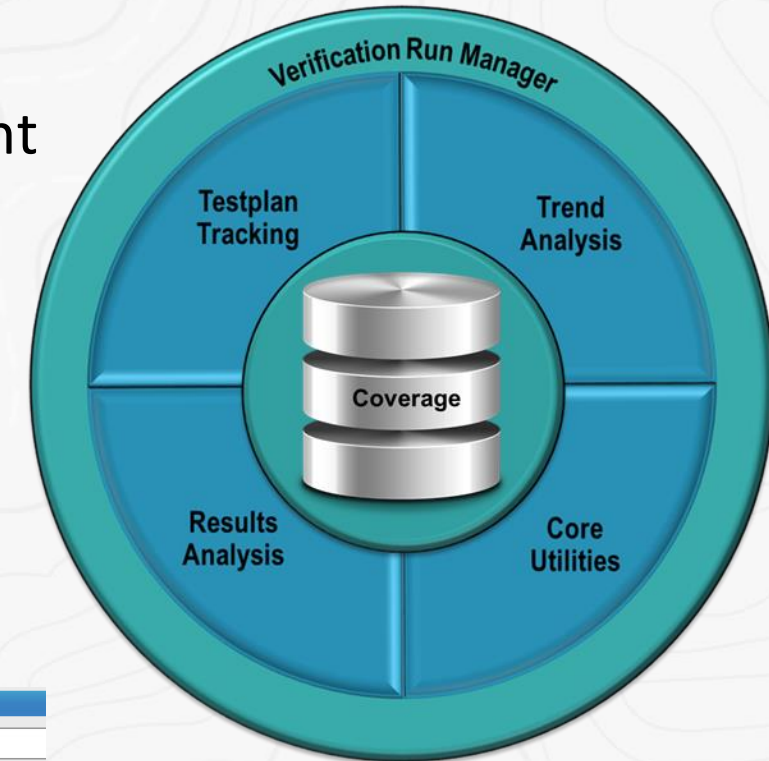
  

Covergroups Coverage ( 80% )				
Covergroups	Bins	Hits	Misses	Coverage
Search...	Search...	Search...	Search...	Search...
\applyPredistortion_c::apply...	14	10	4	80%
MyCCoverGroup_inst	14	10	4	80%

# Verification Coverage Closure

Achieve coverage closure on HLS design source

- Unified Coverage Database (UCDB)
- Coverage analysis, report generation, exclusion development
- Test merging & ranking, test plan integration and tracking



**Questa®**  
**Verification Management**

Design hierarchy	Coverage%	Statement%	Branch%	Expression%
\dvpDemo_c::run#0 :(\dvpDemo_c::run#0 )	78.47%	93.44%	75.30%	66.66%
...>&,ac_channel<IO_N_TYPE>&,ac_channel<IO_N_TYPE>& )	81.80%	86.60%	58.82%	100.00%
...nel<IO_N_TYPE>&,ac_channel<DPD_COEFF_STRUCT>& )	81.93%	98.55%	87.23%	60.00%
...ac_channel<IO_N_TYPE>&,ac_channel<IO_N_TYPE>& )	100.00%	100.00%	100.00%	100.00%
..._INV_TYPE>&,ac_channel<MATRIX_FOR_INV_TYPE>& )	95.13%	100.00%	85.41%	100.00%
..._TRIX_TYPE>&,ac_channel<MATRIX_FOR_INV_TYPE>& )	83.33%	100.00%	100.00%	50.00%
...CTOR_TYPE>&,ac_channel<DPD_COEFF_STRUCT>& )	77.77%	100.00%	100.00%	33.33%
...H_MATRIX_TYPE>&,ac_channel<Z_VECTOR_TYPE>& )	72.02%	93.84%	72.22%	50.00%
\dvpDemo_c::dpdDemo_c#0 :(\dvpDemo_c::dpdDemo_c#0 )	100.00%	100.00%		
\trainingTop_c::trainingTop_c#0 :(\trainingTop_c::trainingTop_c#0 )	100.00%	100.00%		
...CalcCoeffs_c#0 :(\trainingCalcCoeffs_c::trainingCalcCoeffs_c#0 )	100.00%	100.00%		
...matrixInversion_c#0 :(\matrixInversion_c::matrixInversion_c#0 )	100.00%	100.00%		
...abeling_c#0 :(\abeling_c::abeling_c#0 )	100.00%	100.00%		

Name	Total Bins	Missing Bins	% Hit	Coverage	Status	Goal
\applyPredistortion_c::applyPredistortion_c#0 /MyCCoverGroup	14	4	71.42%	80.00%		100%
cp_inputStruct_syncTraining	2	0	100.00%	100.00%		100%
cp_inputStruct_syncUpdate	2	0	100.00%	100.00%		100%
cp_inputStruct_io_type_data_0_real	5	2	60.00%	60.00%		100%
cp_inputStruct_io_type_data_0_imag	5	2	60.00%	60.00%		100%
MyCCoverGroup_inst	14	4	71.42%	80.00%		100%



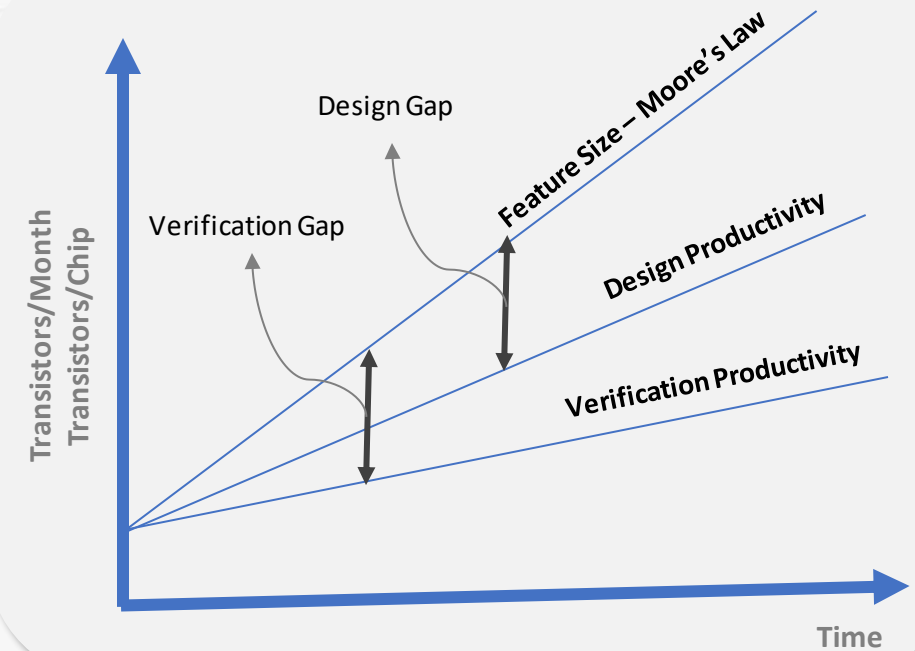
# Catapult High-Level Synthesis & Verification

Comprehensive flow providing needed productivity gains

HLL and HLS to design bit-accurate HW.  
DPD HLS 600 lines of code thus > 10x  
reduction compared to Verilog RTL

HLV using known and trusted approaches  
for functional verification of HLS source.  
DPD HLS sims > 300x speedup over RTL

Efficiencies gained by designing and verifying  
via HLL. Productivity gains via faster and  
predictable post-HLS RTL verification signoff



# Prove

Prove intent is met

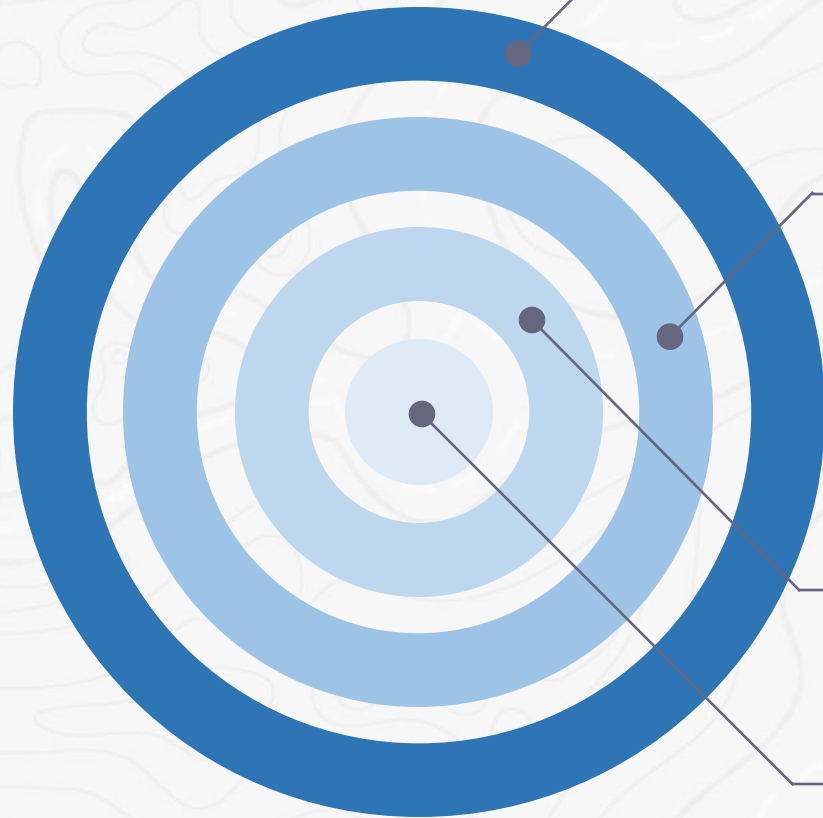
Kurt Takara



# Simulate RTL until you're done\*

The verification methodology you have heard of

Simulation-based Verification



## System design and HW/SW interaction

Unanticipated HW/SW interface bottlenecks  
Datapath and dataflow inconsistencies  
Power management functions

## Functional errors

Incorrect implementation of specified algorithms or architecture  
Unintended functional interactions  
Initialization issues

## Construction errors

Register specification and construction/connection errors  
Interconnect issues  
Clocking, testing

## Coding errors

Syntax, style, semantics, structural issues  
Cut-and-paste creation errors  
Simple mistakes

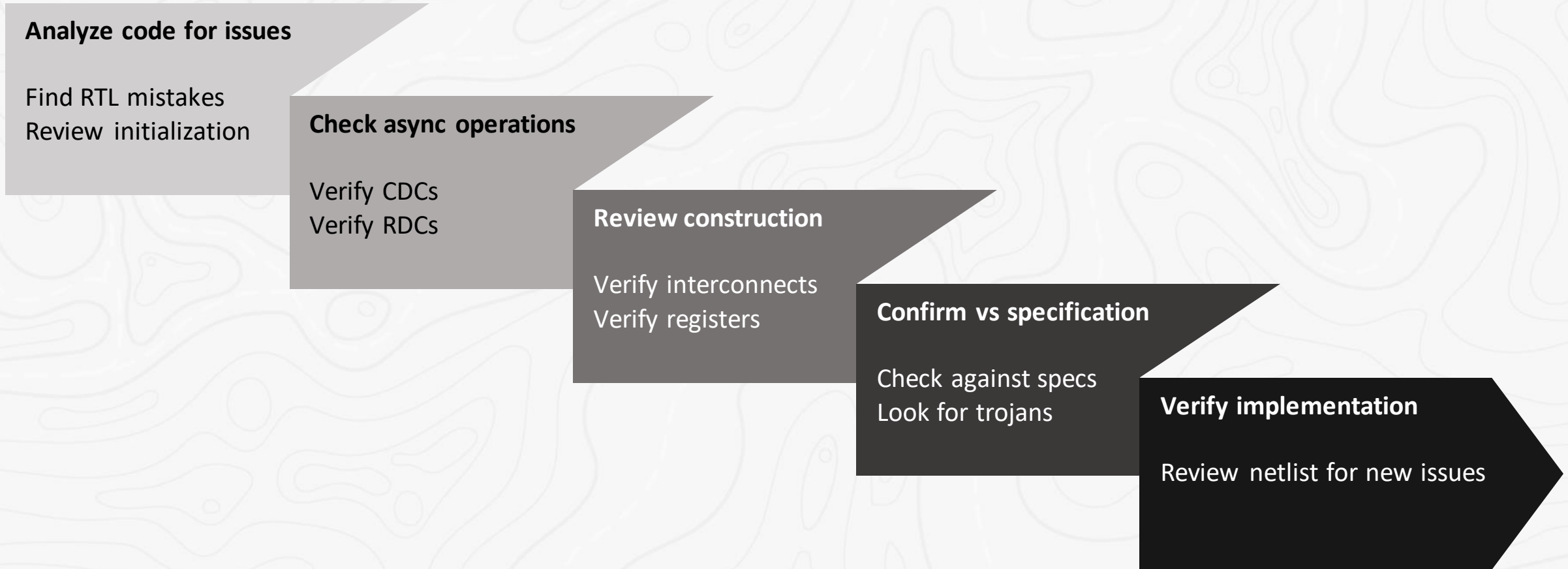
## Issues not found in simulation

Asynchronous crossing issues  
Trojan attacks and hidden functionality  
Implementation issues

\*What "done" means is an entirely different topic

# Using intent-focused insight flushes issues before simulation

Reduce bug density with a non-simulation verification methodology



 **Produce**  
Correct-by-construction

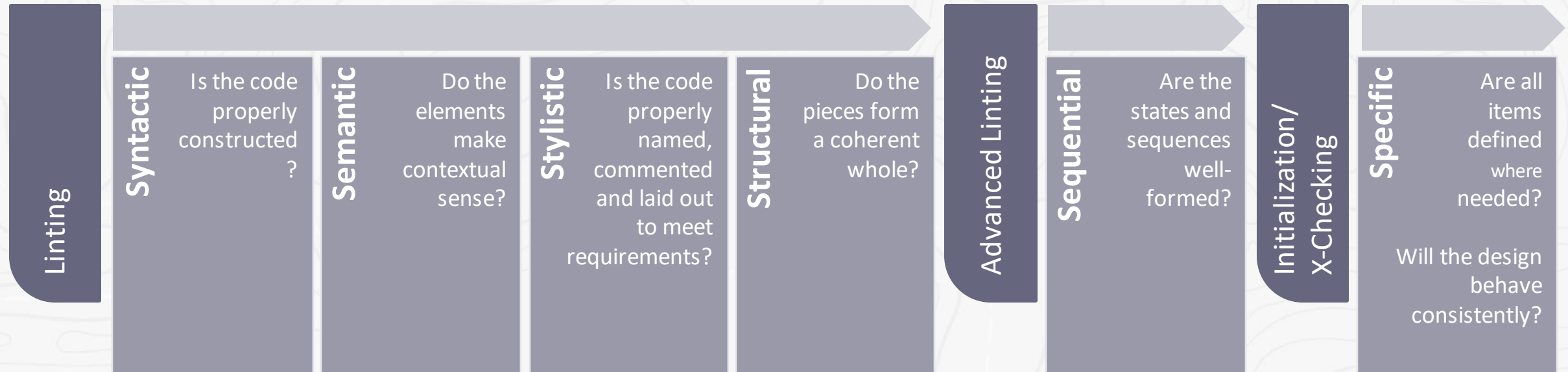
 **Prove**  
Meets intent

 **Protect**  
Retains intent



# Static code analysis finds mistakes without testing

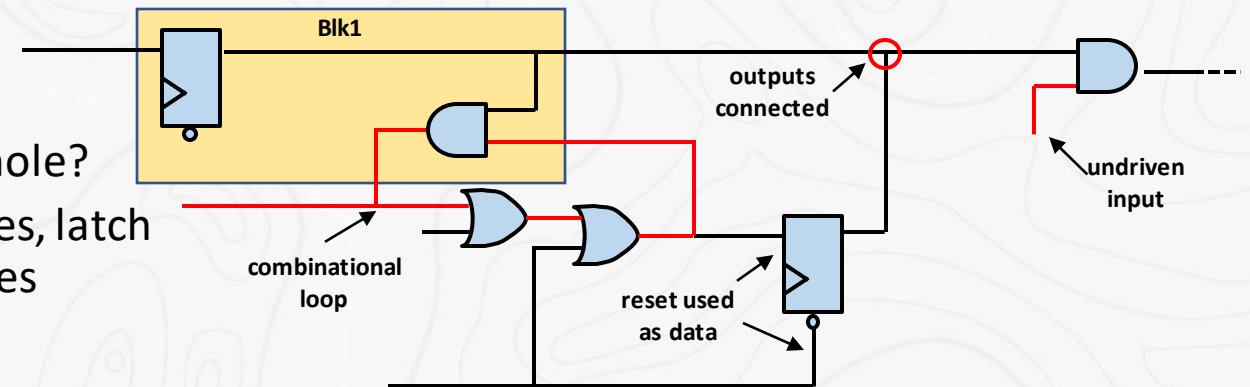
Code analyses find RTL issues early, ensure highest quality code into verification



# Linting analyzes RTL statically, finding issues quickly without simulation

Find & fix syntactic, semantic, stylistic and structural issues early during design

- Static analysis
  - Fast, requires no input apart from RTL
- Semantic issues – do the elements make sense in context?
  - Unsynthesizable code, simulation/synthesis mismatch risks, improper assignments, etc.
- Structural issues – do the elements form a coherent whole?
  - Width mismatches, unreachable or dead FSM states, latch inference, dead code, inconsistent clock/reset styles
- Stylistic issues – do the elements meet coding style requirements?
  - Adequate commenting, naming conventions, unused objects, maintainability



Advanced Linting identifies issues based on deep formal design knowledge  
Code analyses find RTL issues early, ensure highest quality code into verification

## Statically-detectable design issue

```
case (qstate)
3'b001: if (en) dstate = 3'b010;
      else dstate = 3'b001;
3'b010: dstate = 3'b100;
3'b100: if (rtn) dstate = 3'b100;
      else dstate = 3'b100;
default: dstate = 3'b001;
endcase
```

- Both next states of if-else are the same

## Formally-detectable design issue

```
case (qstate)
3'b001: if (en) dstate = 3'b010;
      else dstate = 3'b001;
3'b010: dstate = 3'b100;
3'b100: if (rtn) dstate = 3'b001;
      else dstate = 3'b100;
default: dstate = 3'b001;
endcase
```

- Formal analysis will exhaustively determine if `rtn` can ever be 1

# Initialization and X-Checking examines the design to find unintended issues

Exhaustive initialization/X analysis determines if uninitialized states cause device failure

- Verify if X values propagate to initialized registers and other control logic
  - FSM's, outputs, clocks, resets ...
    - Can be customized with SVA properties
  - All X sources are considered
    - Find and fix X source bugs, report all uninitialized registers
  - X accurate analysis (no optimism)
- Flow





# Initialization and X-Checking examines the design to find unintended issues

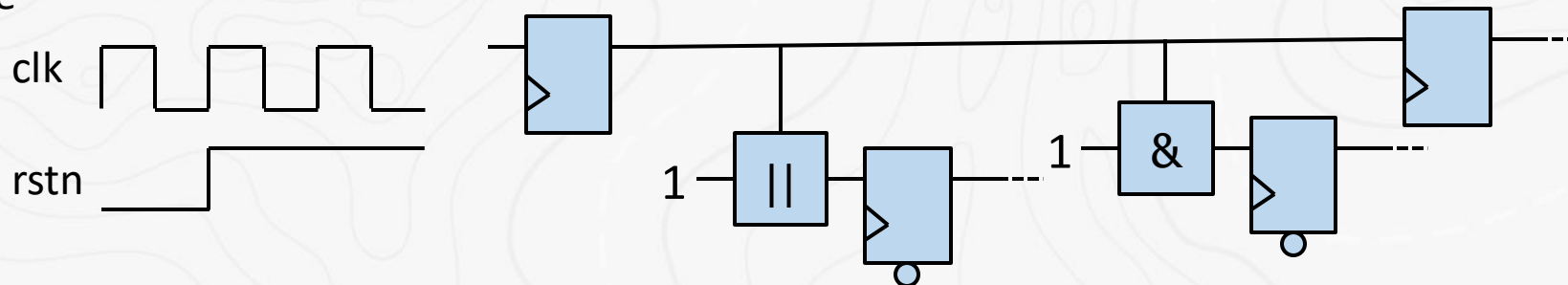
Exhaustive initialization/X analysis determines if uninitialized states cause device failure

- Verify if X values propagate to initialized registers and other control logic
  - FSM's, outputs, clocks, resets ...
    - Can be customized with SVA properties
  - All X sources are considered
    - Find and fix X source bugs, report all uninitialized registers
  - X accurate analysis (no optimism)

## Flow



## Example





# Initialization and X-Checking examines the design to find unintended issues

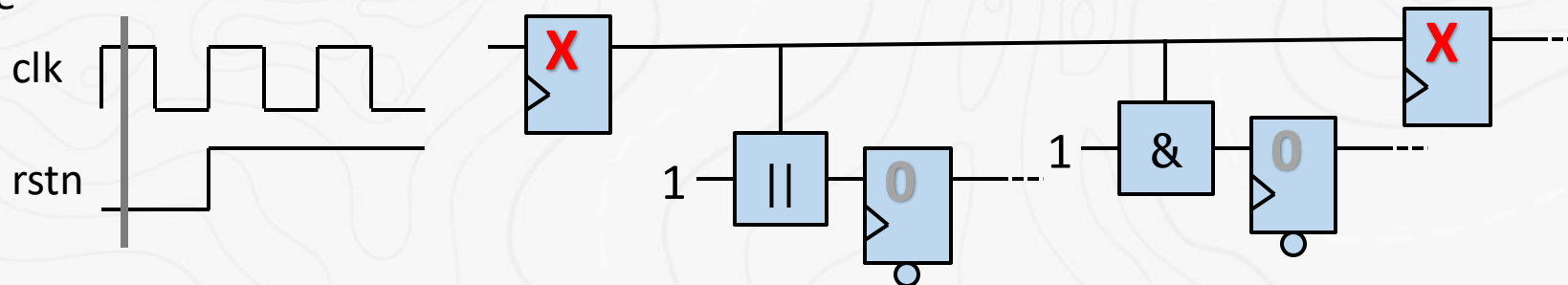
Exhaustive initialization/X analysis determines if uninitialized states cause device failure

- Verify if X values propagate to initialized registers and other control logic
  - FSM's, outputs, clocks, resets ...
    - Can be customized with SVA properties
  - All X sources are considered
    - Find and fix X source bugs, report all uninitialized registers
  - X accurate analysis (no optimism)

## Flow



## Example



# Initialization and X-Checking examines the design to find unintended issues

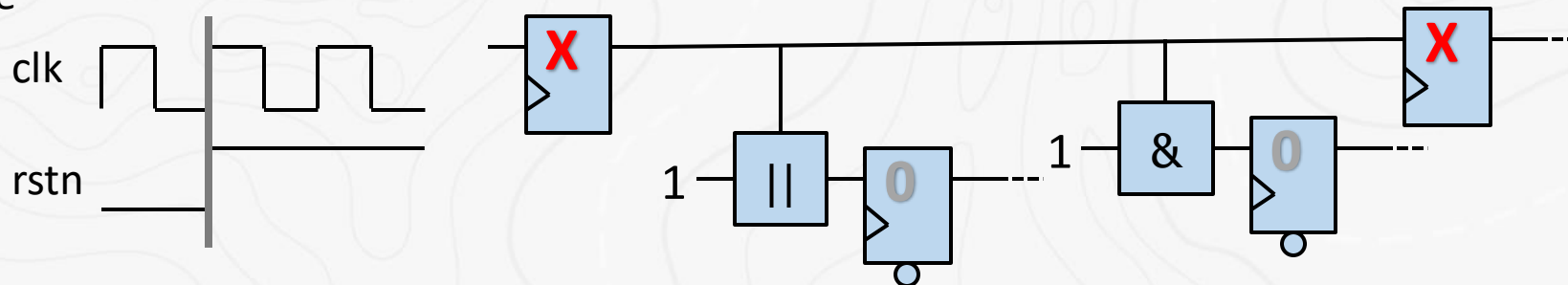
Exhaustive initialization/X analysis determines if uninitialized states cause device failure

- Verify if X values propagate to initialized registers and other control logic
  - FSM's, outputs, clocks, resets ...
    - Can be customized with SVA properties
  - All X sources are considered
    - Find and fix X source bugs, report all uninitialized registers
  - X accurate analysis (no optimism)

## Flow



## Example





# Initialization and X-Checking examines the design to find unintended issues

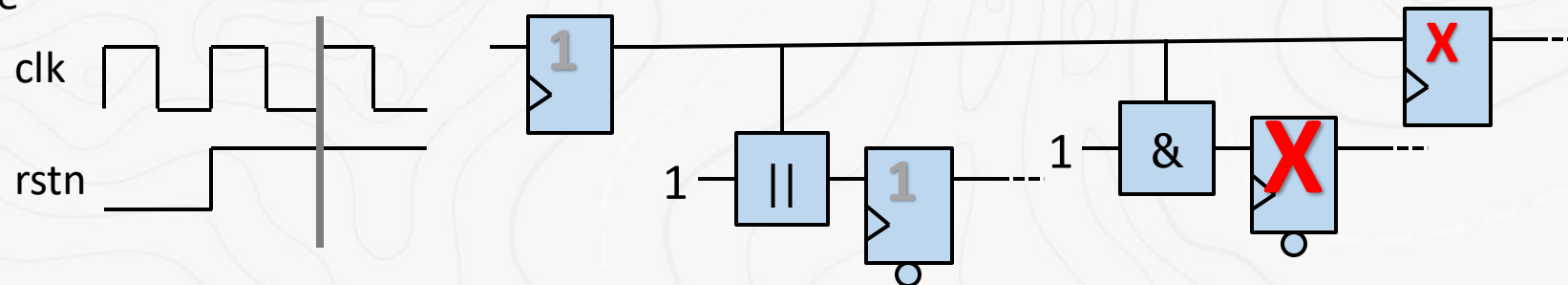
Exhaustive initialization/X analysis determines if uninitialized states cause device failure

- Verify if X values propagate to initialized registers and other control logic
  - FSM's, outputs, clocks, resets ...
    - Can be customized with SVA properties
  - All X sources are considered
    - Find and fix X source bugs, report all uninitialized registers
  - X accurate analysis (no optimism)

## Flow



## Example



# Using intent-focused insight flushes issues before simulation

Reduce bug density with a non-simulation verification methodology

## Analyze code for issues

Find RTL mistakes  
Review initialization

## Check async operations

Verify CDCs  
Verify RDCs

## Review construction

Verify interconnects  
Verify registers

## Confirm vs specification

Check against specs  
Look for trojans

## Verify implementation

Review downstream netlist  
for new issues

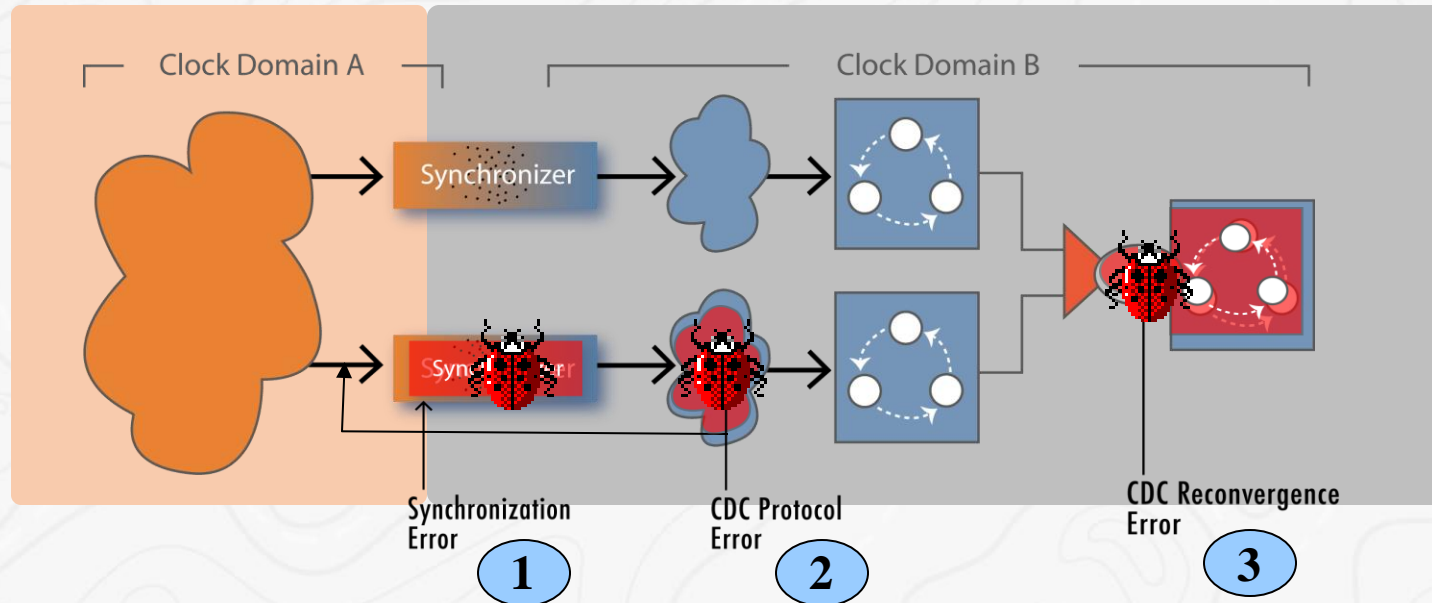
 **Produce**  
Correct-by-construction

 **Prove**  
Meets intent

 **Protect**  
Retains intent

# What is a clock domain crossing (CDC)?

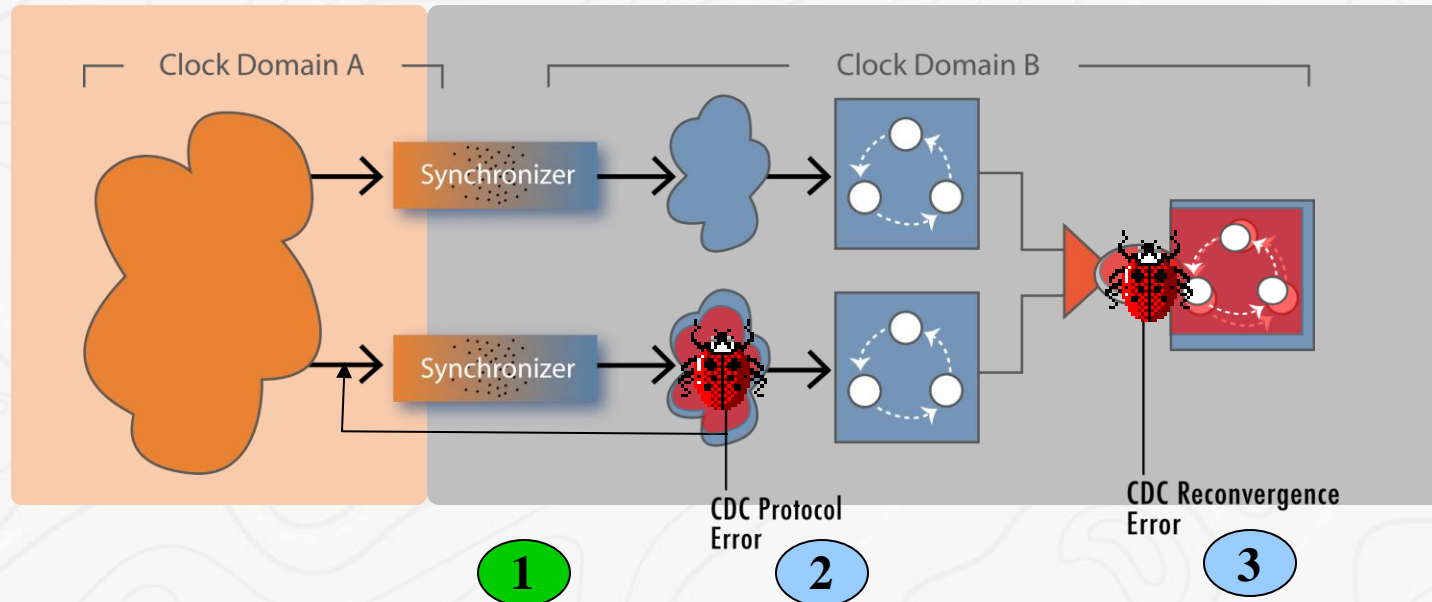
A CDC is the transit of a signal or group of signals from one clock domain to another



1. The design has missing or incorrect synchronizers
2. The design does not adhere to the required CDC protocols to ensure correct data transfer
3. The design does not account for non-deterministic delays through synchronizers

# What is a clock domain crossing (CDC)?

A CDC is the transit of a signal or group of signals from one clock domain to another

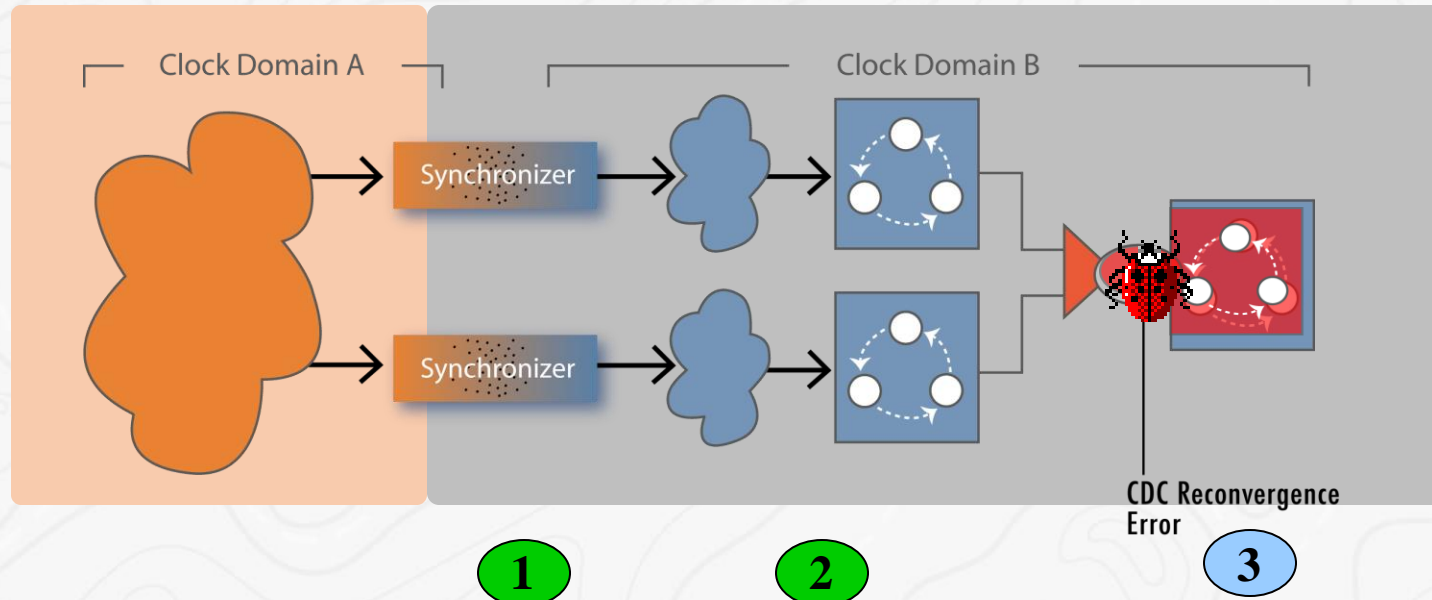


1. Complete structural analysis to find all synchronizers
1. The design does not adhere to the required CDC protocols to ensure correct data transfer
2. The design does not account for non-deterministic delays through synchronizers



# What is a clock domain crossing (CDC)?

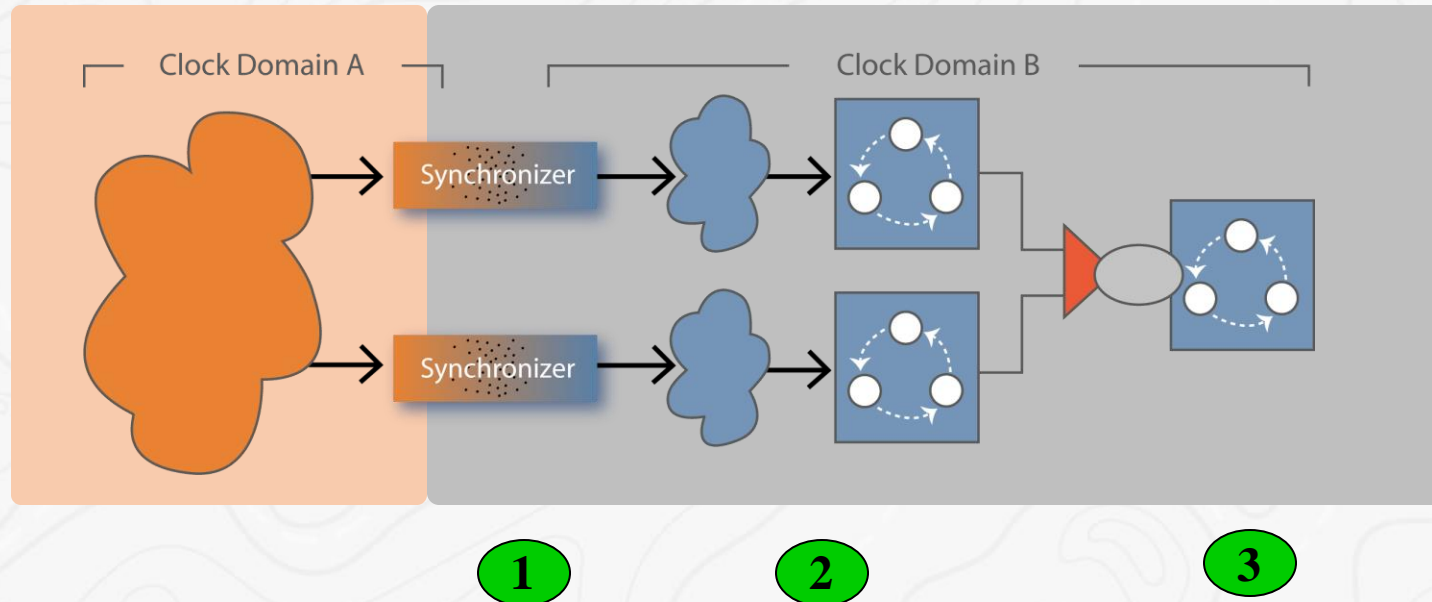
A CDC is the transit of a signal or group of signals from one clock domain to another



1. Complete structural analysis to find all synchronizers
2. Automated assertion-based verification to ensure correct implementation of CDC protocols
1. The design does not account for non-deterministic delays through synchronizers

# What is a clock domain crossing (CDC)?

A CDC is the transit of a signal or group of signals from one clock domain to another

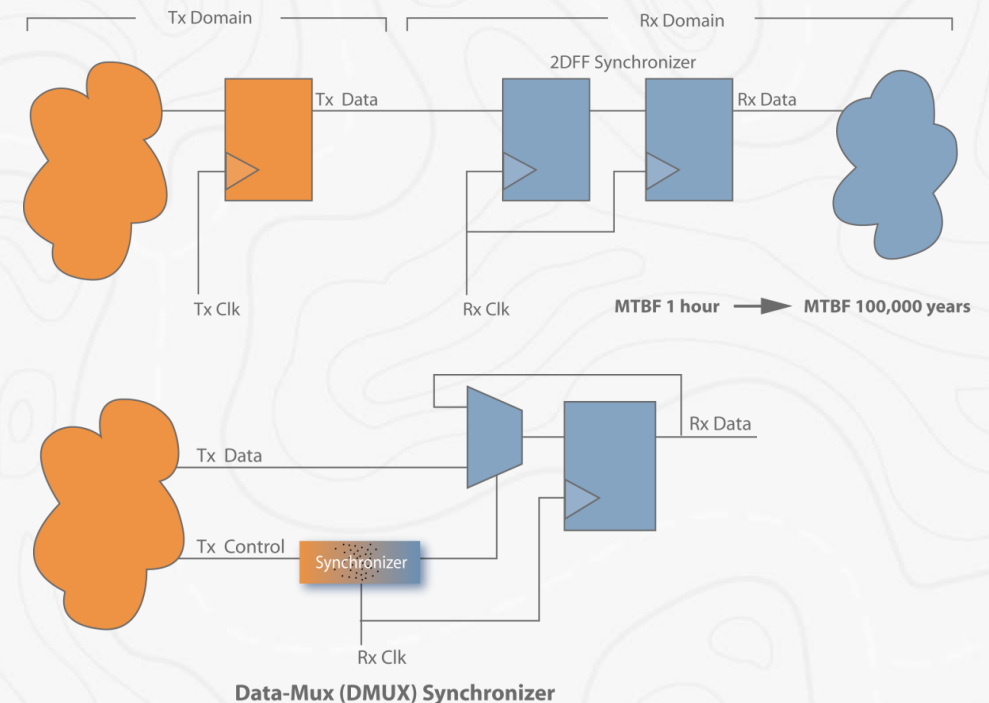


1. Complete structural analysis to find all synchronizers
2. Automated assertion-based verification to ensure correct implementation of CDC protocols
3. Accurate simulation of metastability effects in synchronizers to predict true silicon behavior

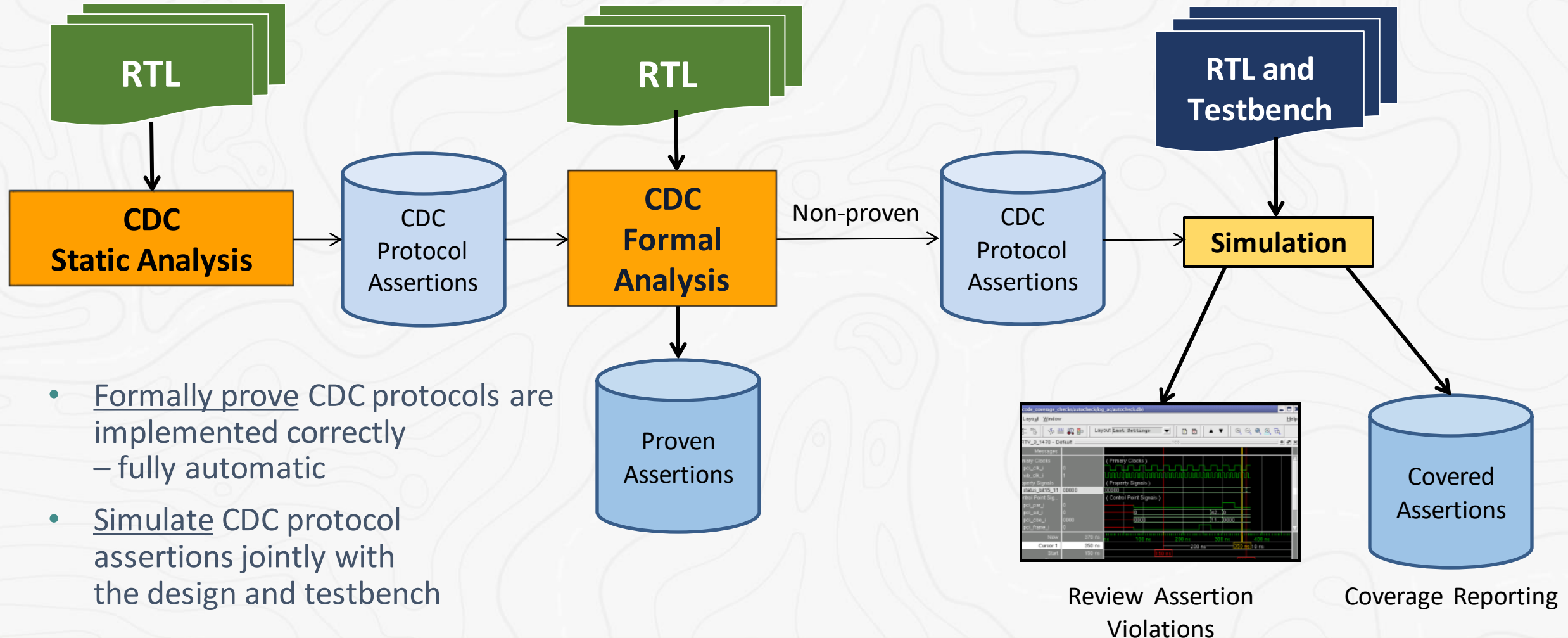
# Why does a crossing protocol matter?

Adding or checking for a synchronizer across every CDC isn't enough

- Synchronization between clock domains requires a transfer protocol
  - To ensure that data is predictably transferred between domains
  - Simplest example: Stability check on input to 2-DFF synchronizer
    - Signal must be held stable long enough in the transmitting clock domain
- These protocols must be verified
- When protocol is violated
  - Data can be lost
  - Simulation may not show a failure
  - Silicon implementation will eventually fail!



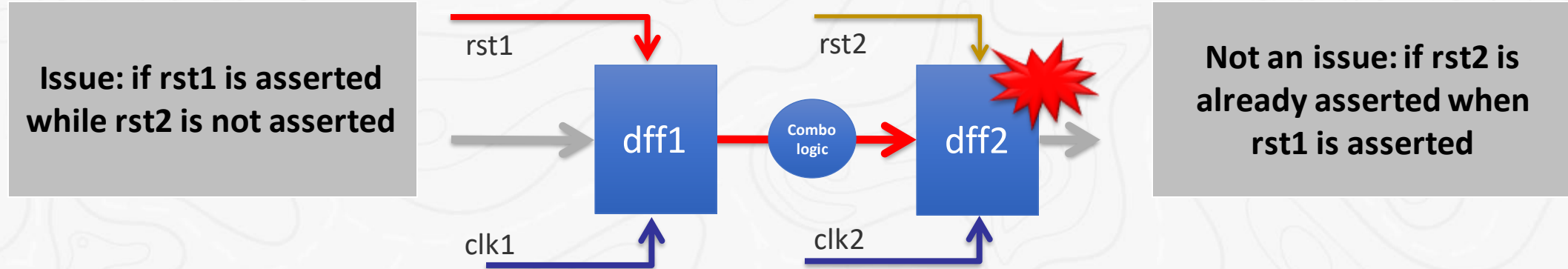
How much of the protocol can be verified without simulation?  
As many crossings as possible are verified without a testbench





## What is a reset domain crossing (RDC)?

An RDC is the transit of a signal or group of signals from one reset domain to another



Reset Condition	Clock condition	Result
rst1 and rst2 are asynchronous	clk1 and clk2 are same domain	violation
rst1 and rst2 are asynchronous	clk1 and clk2 are different domains	caution – CDC verification needed
rst1 and rst2 follow ordering constraints	NA	evaluation

# Using intent-focused insight flushes issues before simulation

Reduce bug density with a non-simulation verification methodology

## Analyze code for issues

Find RTL mistakes  
Review initialization

## Check async operations

Verify CDCs  
Verify RDCs

## Review construction

Verify interconnects  
Verify registers

## Confirm vs specification

Check against specs  
Look for trojans

## Verify implementation

Review downstream netlist  
for new issues

 **Produce**  
Correct-by-construction

 **Prove**  
Meets intent

 **Protect**  
Retains intent

# Review design construction integrity with interconnect verification

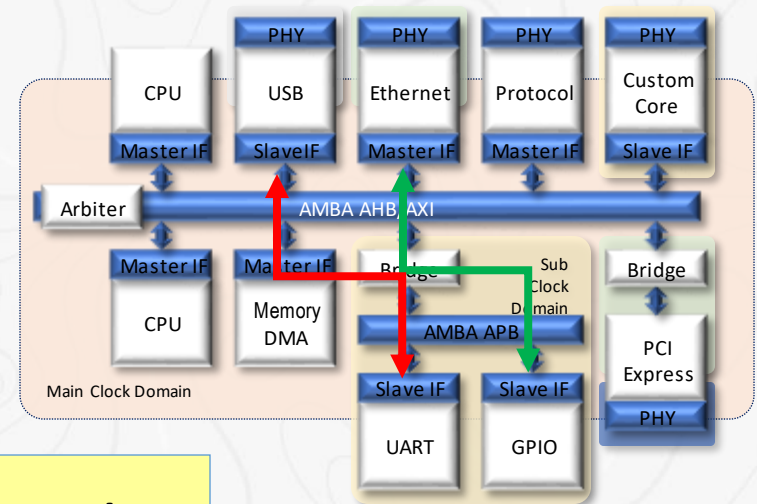
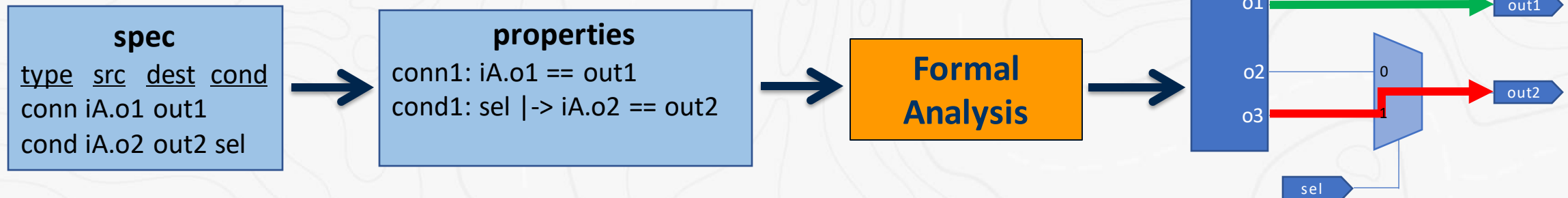
Verify the correctness of connectivity in the design

- Check that design connection implementation matches the specification
  - Endless applications: pin muxes, power rails, clock trees, etc.
  - Both functional and structural connectivity checked
  - Checks include: connect, conditional, constants, delays, etc.
  - ASIC and FPGA designs from block to SoC level verification

- Flow



- Example

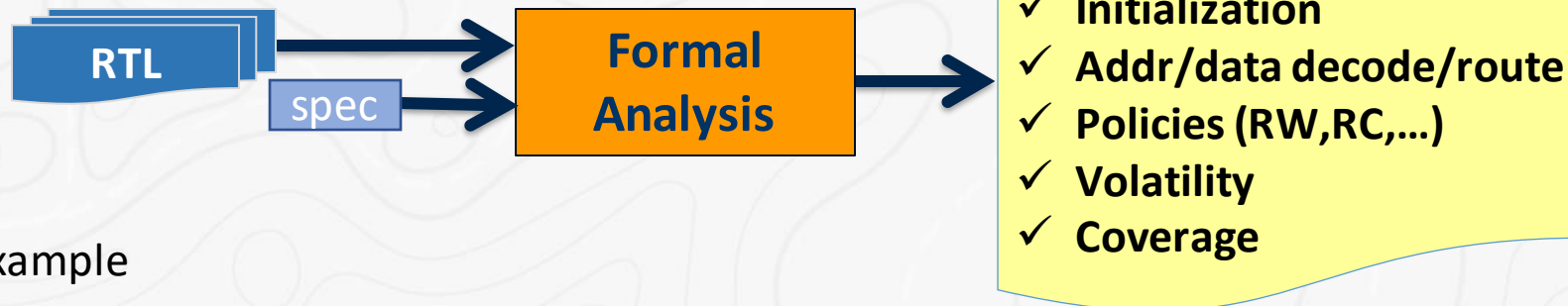


# Ensuring register implementation correctness and construction is vita

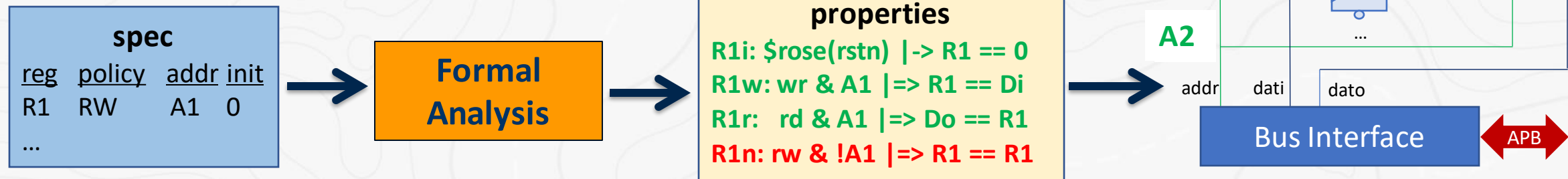
Consistency between spec and implementation aids working systems downstream

- Verify memory mapped register implementation matches the specification
  - IP-XACT, UVM, and other custom specs
  - Front door and back door
  - Common interfaces supported, can be customized

- Flow



- Example





# Using intent-focused insight flushes issues before simulation

Reduce bug density with a non-simulation verification methodology

## Analyze code for issues

Find RTL mistakes  
Review initialization

## Check async operations

Verify CDCs  
Verify RDCs

## Review construction

Verify interconnects  
Verify registers

## Confirm vs specification

Check against specs  
Look for trojans

## Verify implementation

Review downstream netlist  
for new issues

 **Produce**  
Correct-by-construction



**Prove**  
Meets intent

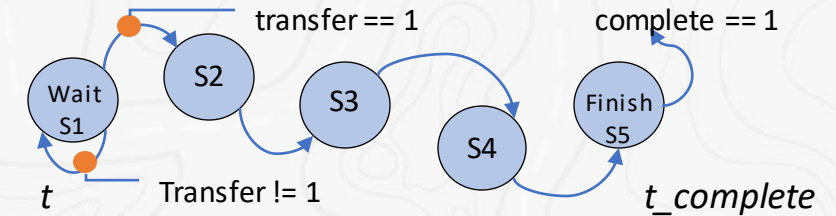


**Protect**  
Retains intent

# Operational assertions enable verification of conformance to specification

A timing diagram can be transcribed to assertions for formal verification

- Capture your design functionality as a set of operations
  - Start state, i.e. a starting condition
  - Trigger condition – event that triggers the operations
  - Expected output behavior
  - End state
- Use standard languages to write operational assertions
  - TiDAL : OneSpin SystemVerilog library
  - Allow timing diagram style assertions
  - Generate proof or CEX for operational assertions
- Enable Operational Assertion Based Verification
  - Ensure verification completeness
  - Automatically detect gaps between operations



state	Wait				Finish
transfer	1				
complete					

```
sequence t_complete; nxt(t,4); endsequence
```

```
property transfer;
```

```
t ##0 state == wait and
t ##0 transfer == 2'd1
```

Cause

```
implies
```

```
t_complete ##0 state == finish and
t_complete ##0 complete;
```

Effect

```
endproperty
```

```
transfer_a: assert property (transfer);
```

# Verification against specifications can find unanticipated functionality

## Processor verification confirms implementation against ISA

### 1 Automated Design Analysis



Arch &  $\mu$ Arch Database

Implementation Info

GUI (Visualize/ Extend)

Custom Extensions

Provide Core Specific Information

### 2

### 3 Automated Verification File Generation

Automated Assertion Generation

Verification Database (SV)

Retune Core Checker

### 4

### 5 Run Assertions/ Completeness

Debug

Proof (Assertions/ Completeness)



Signoff

Coverage Database

# Processor trojan verification shifts from verifying what is to what isn't

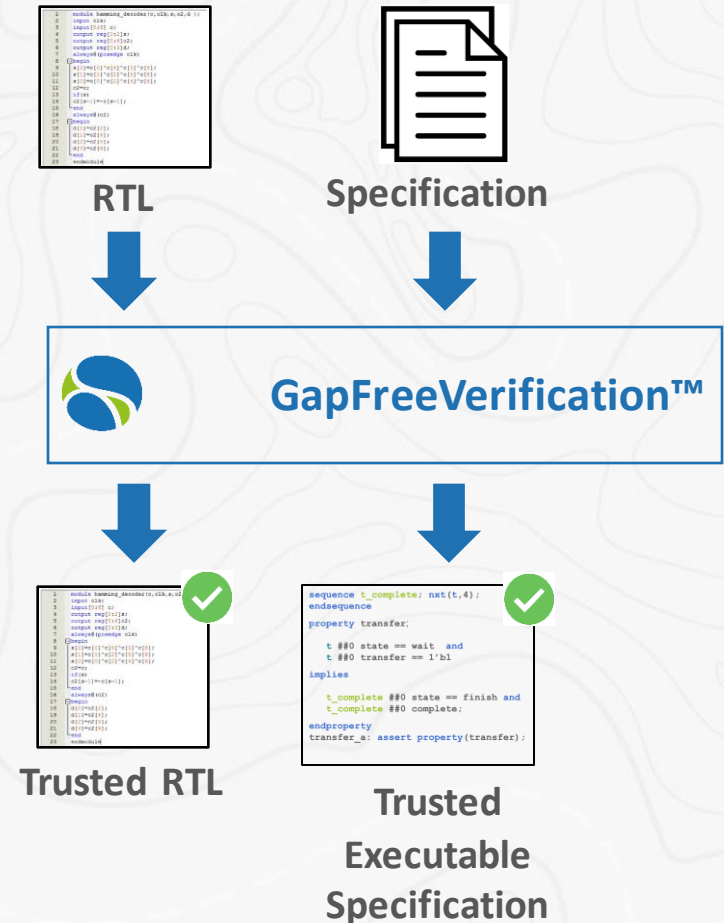
Extending a GapFree analysis to processor verification finds what shouldn't be there

- Benefits

- Detects errors and inconsistencies in the specification
- Prove 100% equivalence between specification and implementation
- Demonstrates absence of bugs/Trojans/ambiguities

- Successes

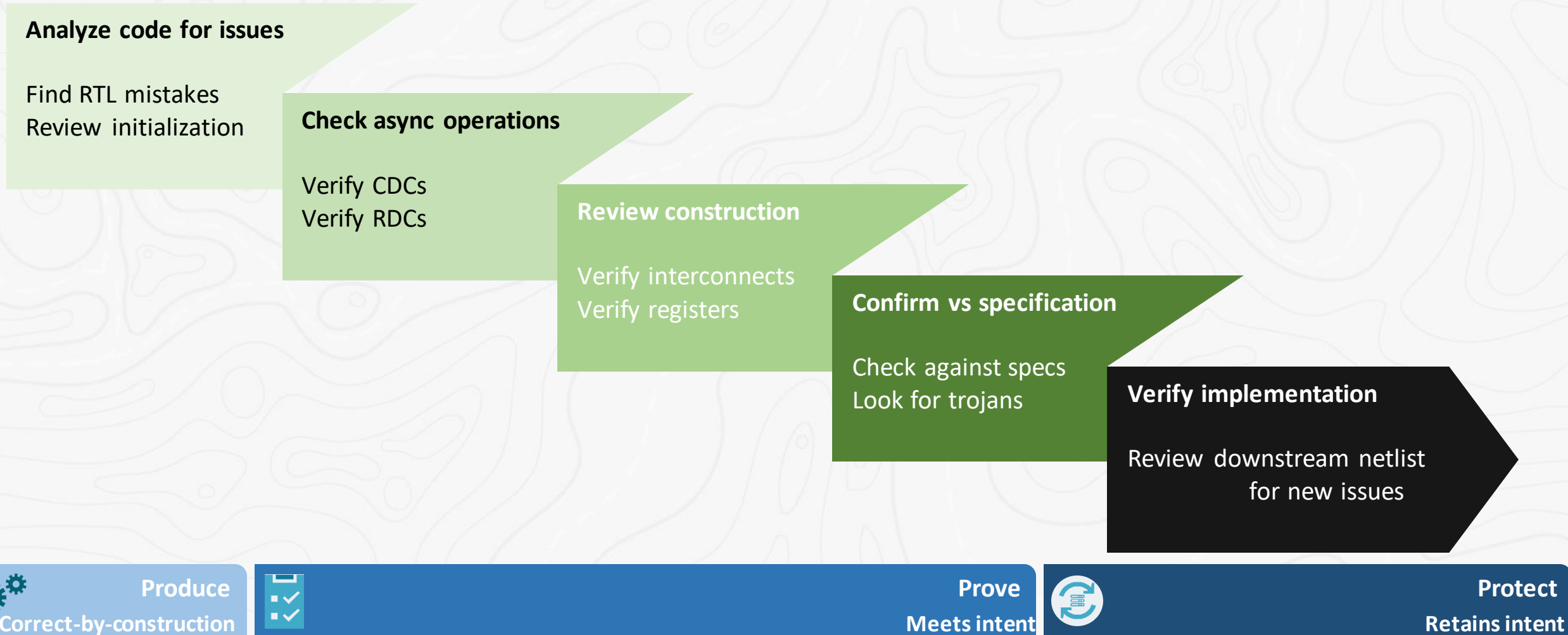
- Paper “Complete Formal Verification of RISC-V Processor IPs for Trojan-Free Trusted ICs” presented at GOMACTech 2019 identified bugs reported on GitHub (Discovered CEASE instruction – a “trojan kill switch” to those without knowledge)





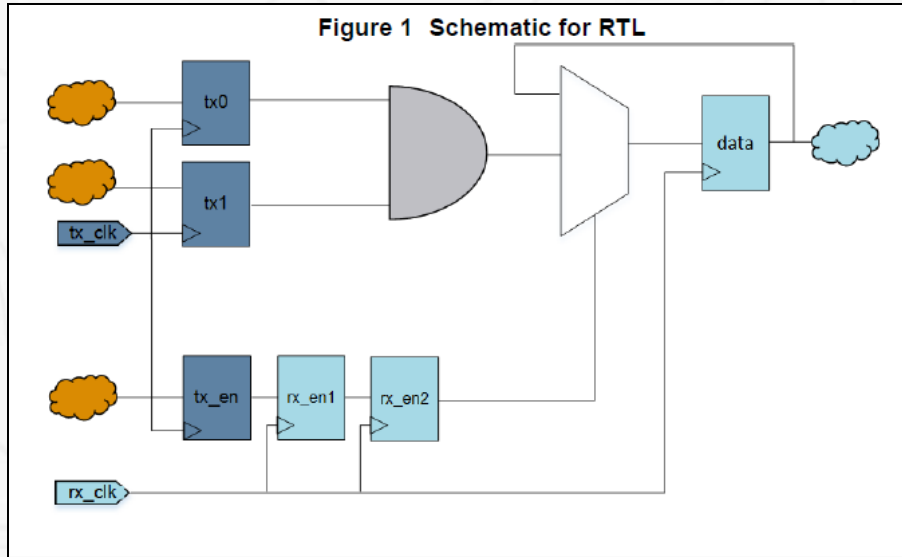
# Using intent-focused insight flushes issues before simulation

Reduce bug density with a non-simulation verification methodology

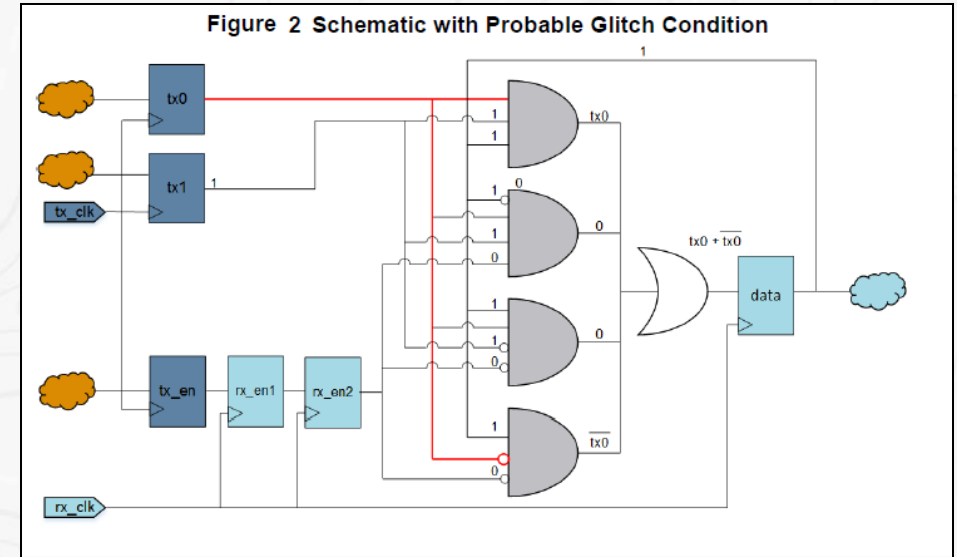


# Verifying the implementation matches the original intent is critical

Example: Synthesis (or other implementation stages) can compromise a clean RTL design



Synthesis



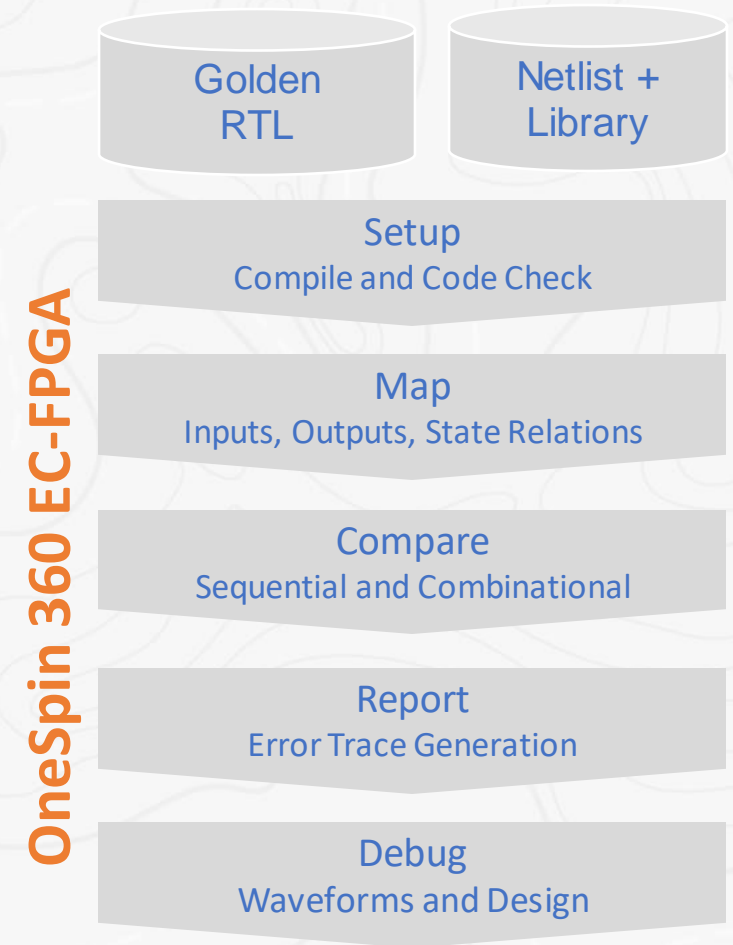
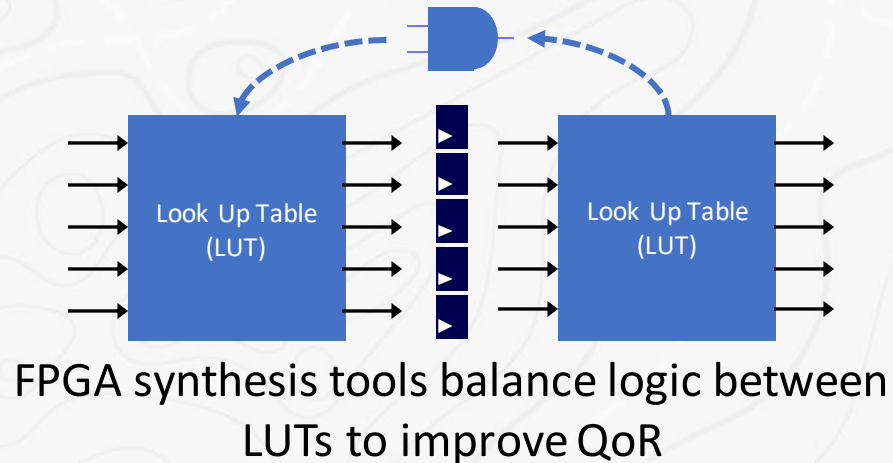
- Synthesis can break corrective circuitry or add surprise paths
- MUX used at RTL for CDC crossing
- Synthesis tool may implement combinational logic which produces glitch
  - $X+!X$  or  $X\&!X$
- Potential chip failure issue if glitch is caught by the receiving flip-flop

# Equivalence checking flows for FPGAs finds implementation-caused issues

Check for combinational and sequential equivalence from Golden RTL through final

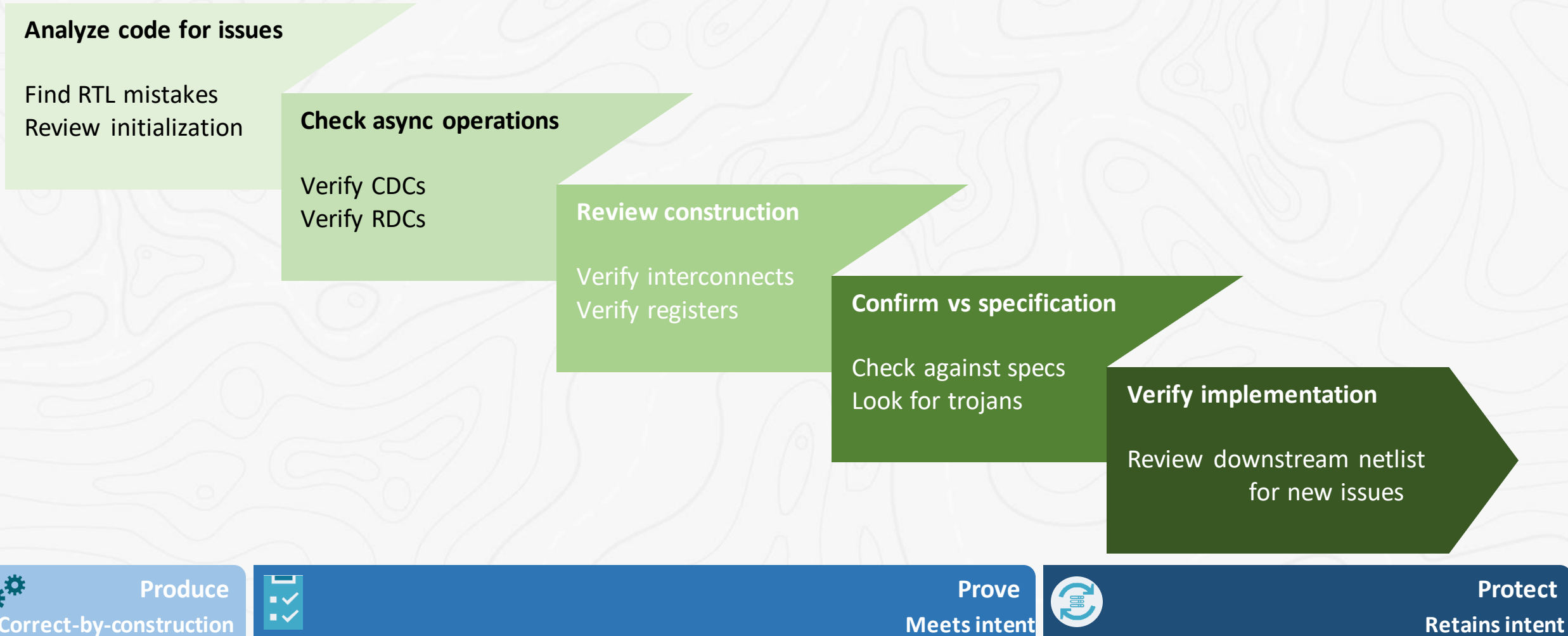
- FPGA Specifics

- Fixed interconnect grid, LUTs, shift registers, block RAMs, configurable DSP blocks, etc.
- Many timing, fan-out, capacity restrictions
- Synthesis maximizes utilization by register duplication, retiming, and other sequential optimizations



# Using intent-focused insight flushes issues before simulation

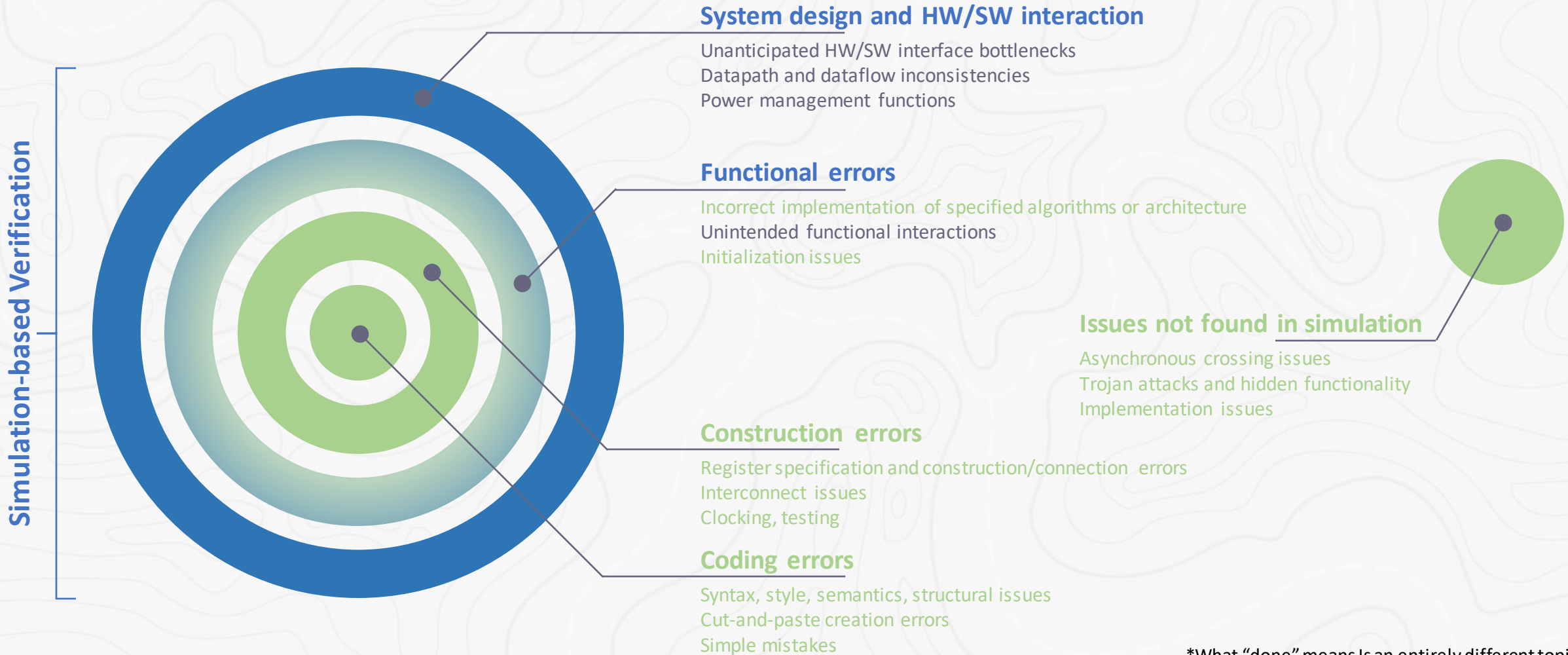
Reduce bug density with a non-simulation verification methodology





# Revisiting the verification methodology you have heard of

Static and Formal capabilities enable cleaner RTL into simulation-based verification



\*What “done” means is an entirely different topic

# Protect

Protect intent throughout development lifecycle

Amir Attarha



# Veloce emulation and prototyping

A complete and integrated verification and validation platform

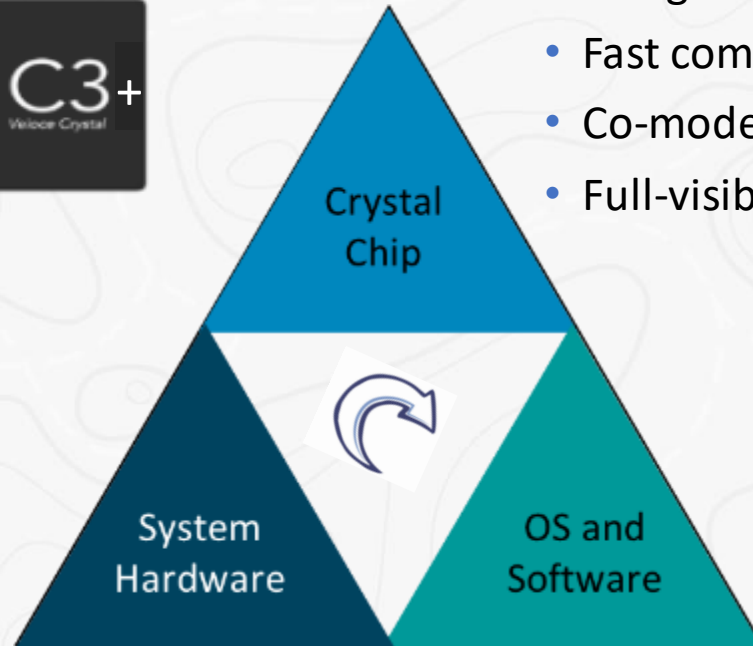


# Scalable Veloce Architecture for HW-Accelerated Verification

Custom Crystal  
device



Hardware System with backplane and co-model channel architecture



Siemens' chip/sys/SW is purpose-built for emulation

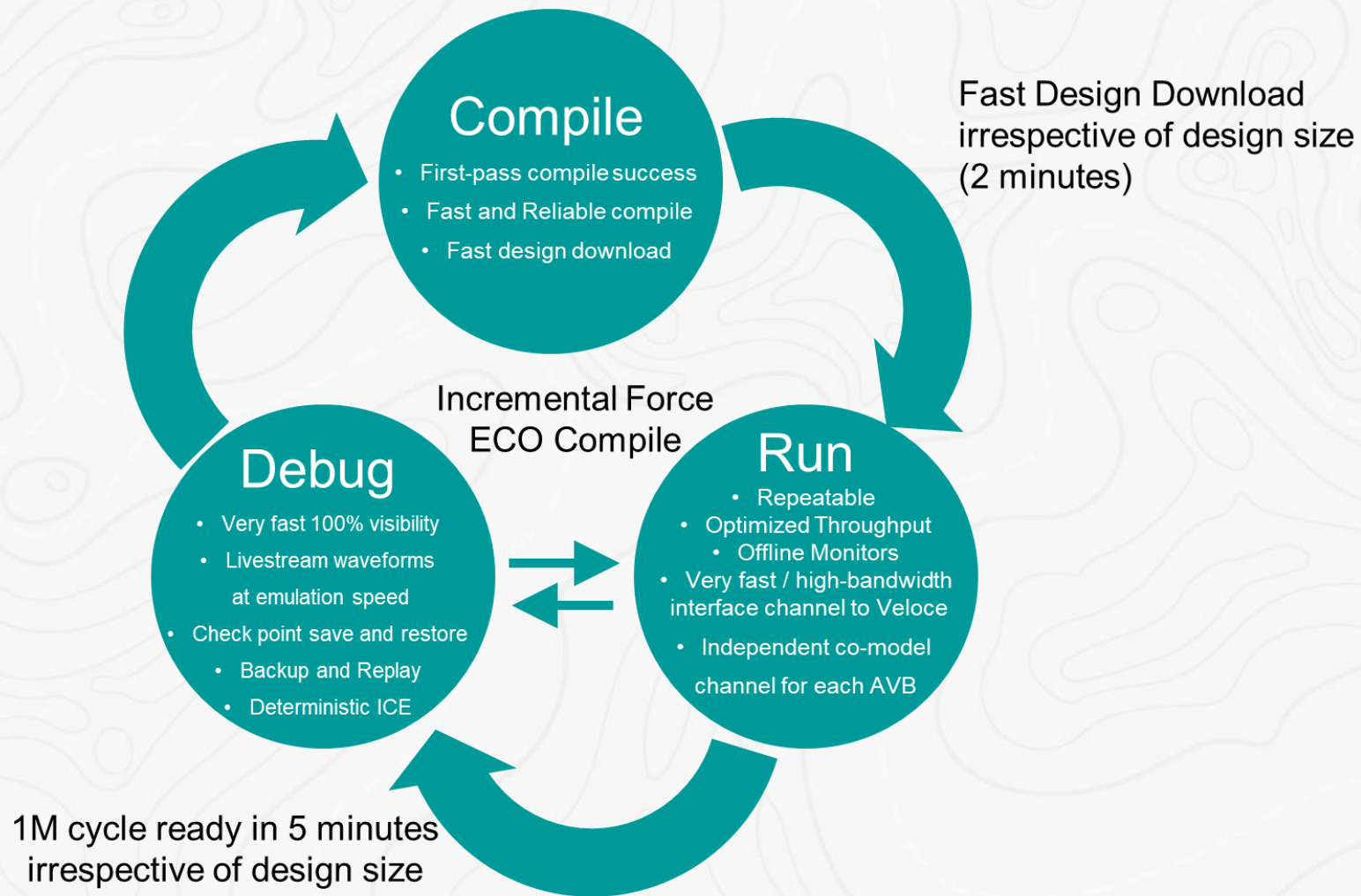
- Design scalability (12 BG, in Production Use)
- Fast compile (5 min/chip, patented VirtualWires)
- Co-model bandwidth (64 channels/StratoM)
- Full-visibility for complete debug (1M signals / 5min)



Comprehensive OS+Apps  
SW Framework



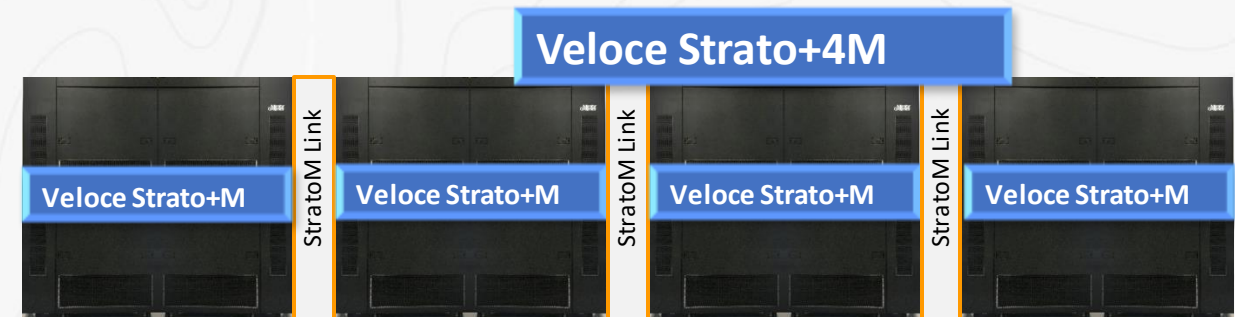
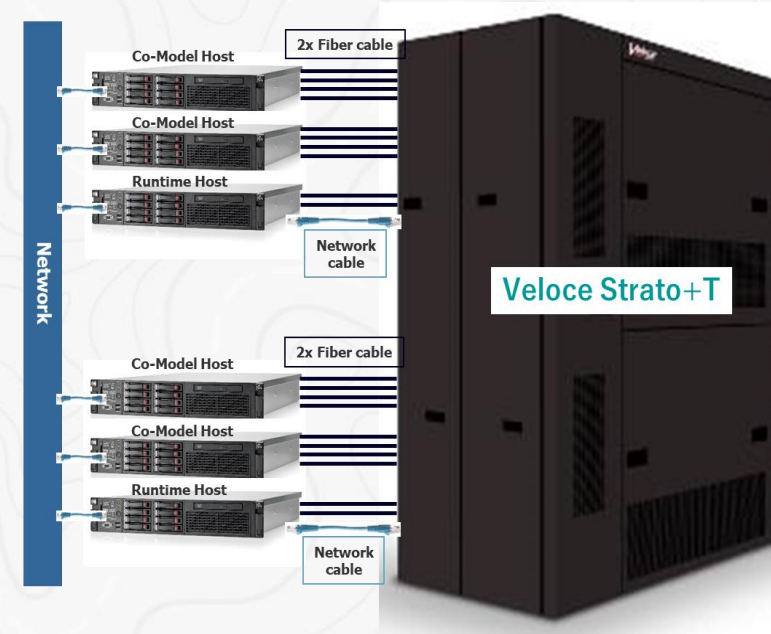
# Veloce Accelerates Progress Cycle



# Veloce Strato: Architected to scale

- Low latency, high bandwidth channel
- Independent co-model channel for each AVB
- Concurrent execution of TB, Comm, DUT
- Directly impacts on these use models
  - Virtual environments
  - Time to visibility and debug
  - Simulation acceleration and coverage closure
  - Monitors/trackers for data collection
  - Power trend analysis and measurements
  - Data analytics

**No performance-throughput degradation  
with capacity scaling**



# Clock frequency vs. throughput: An important HAV system attribute

Emulator clock is easy to report, but everyone knows that the real measure of verification throughput is wall-clock time to execute customer workloads

Design/Test	Competition Off-The-Shelf FPGA-based Emulator Reported Average Clock: 2.0 MHz	Veloce Strato Reported Average Clock: 632 KHz
Design #1	8 min	9 min
Design #2, Test #1	6 min	2 min
Design #2, Test #2	26 min	22 min
Design #3, Test #1	6 min	2 min
Design #3, Test #2	27 min	23 min
Design #4, Test #1	7 min	2 min
Design #4, Test #2	29 min	25 min

**Veloce Strato's superior system architecture delivers more SW-driven workload performance even when the competition reports faster clock speed!**

	Competition Off-The-Shelf FPGA-based Emulator	Veloce Strato
Compile Time	12 Hrs	3 Hrs
Debug (1M Cycles)	30 min	2 min

# Ensuring optimized and productive debug with Veloce Strato

## **Fast 100% visibility** built-in

- Always available for every compile
- No model performance impact
- No capacity impact
- No probes

## **AutoUpload** for long simulation time captures

- 1 Million cycle upload in 5 mins
- Unlimited trace depth for full debug
- Scalable, independent of design size

## **LiveStream** with marching waves

- Leverage multiple co-model channels

## **Advanced debug Flows**

- Checkpoint Save & Restore
- Backup Replay etc.

## **Protocol analyzer**

- Virtual and ICE
- All major protocols

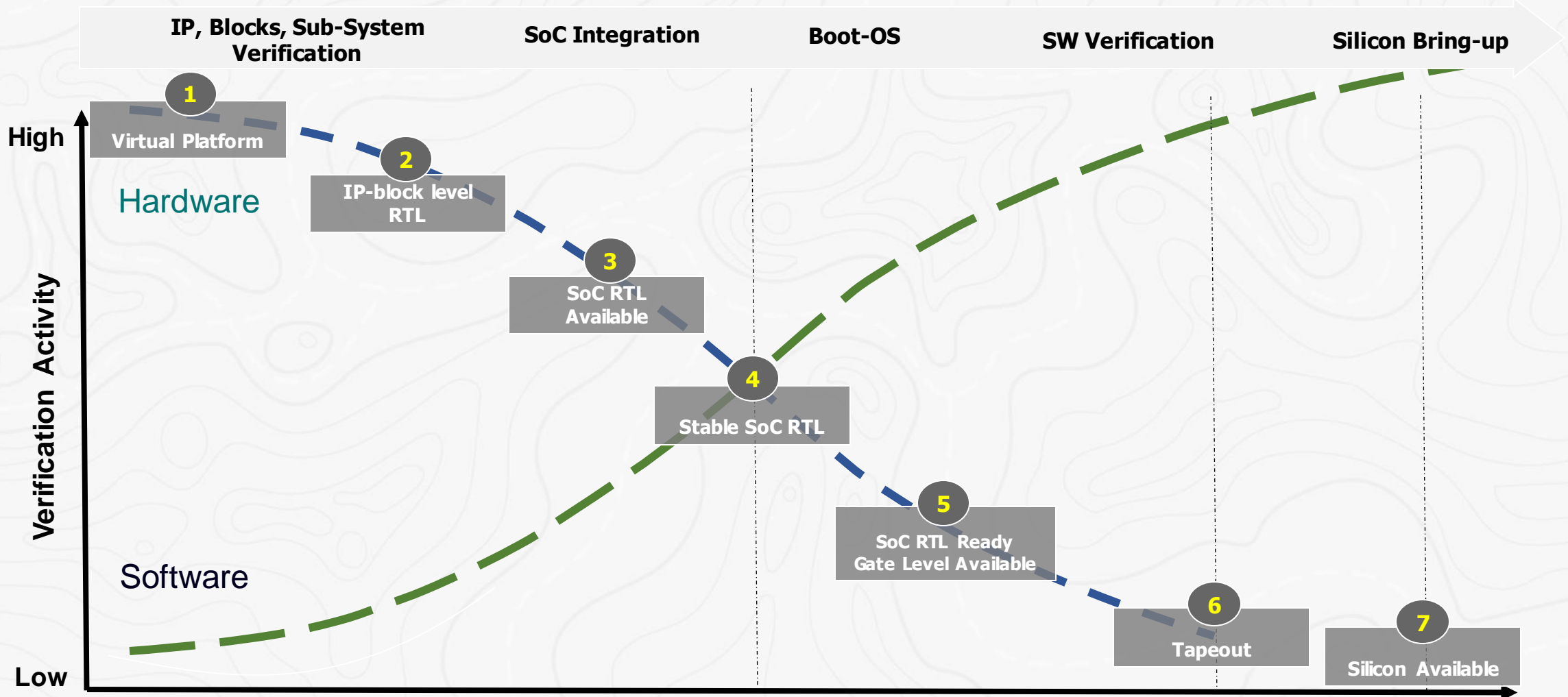
## **Triggers** (no recompile needed)

## **Virtual FSDB support (Verdi)**

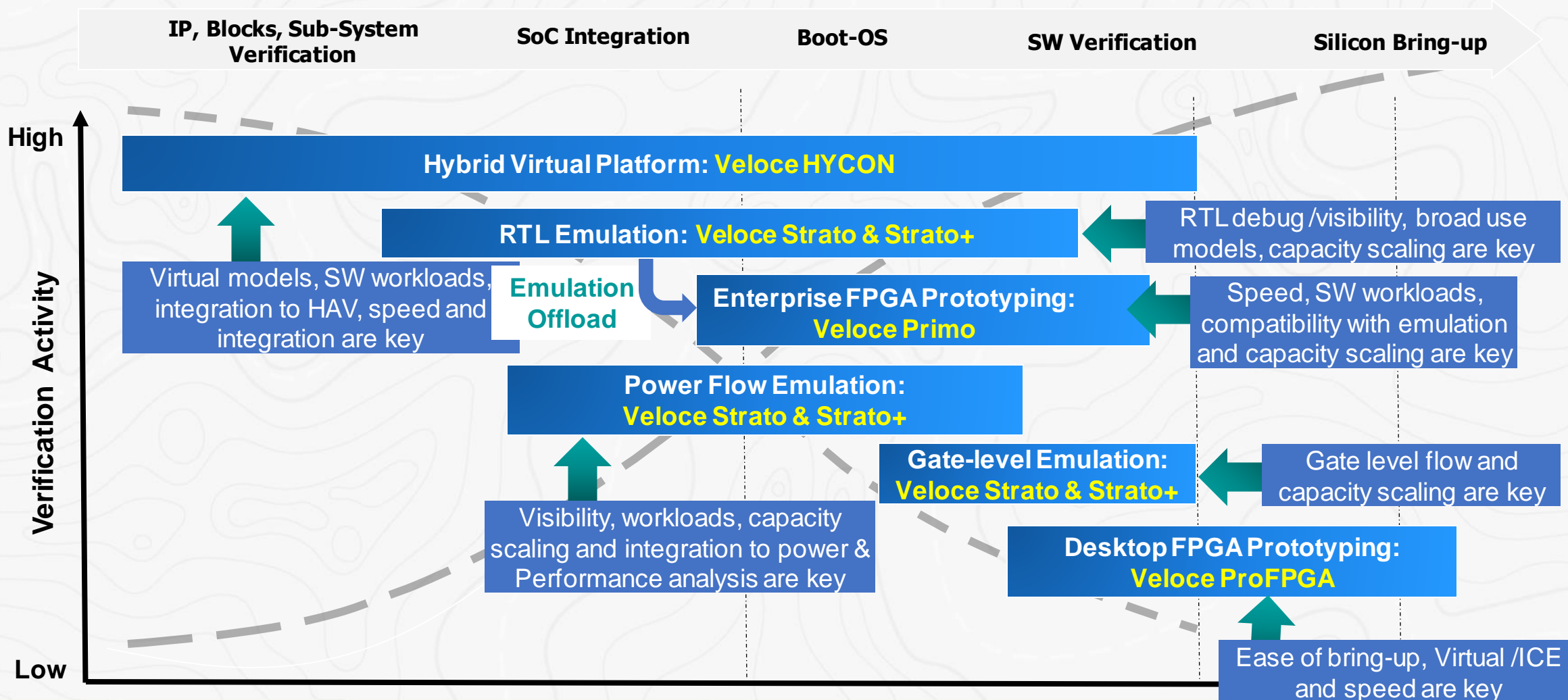
**1 million cycles with full visibility takes 5 minutes irrespective of design size**



# Design and verification milestones



# Velocite providing a complete and integrated solution 'right tool for the right task'



# Satisfying SW workload and benchmark requirements with Hardware-Assisted Verification (HAV)

## Software, Workload-based requirements

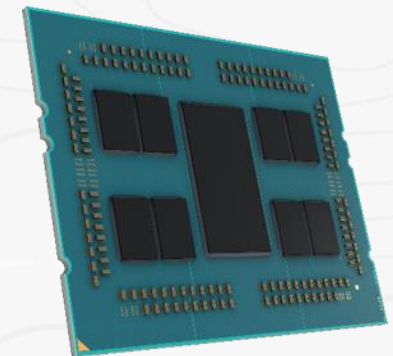
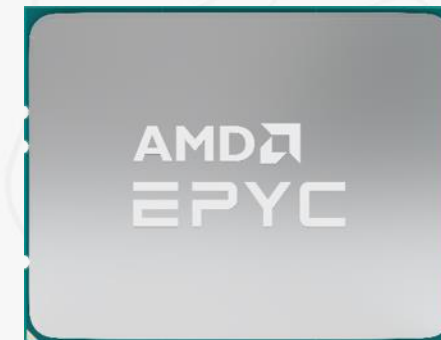
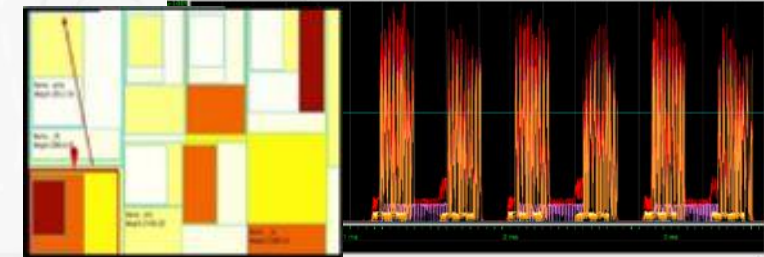
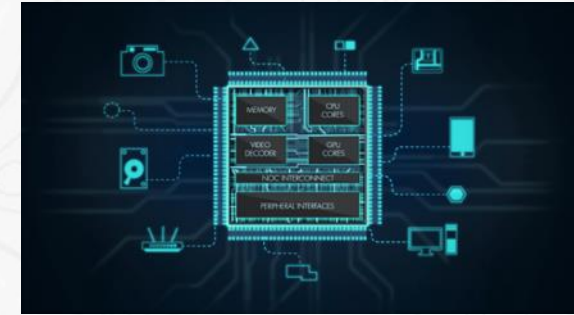
- Long software sequences take extensive verification cycles to complete
- From boot-sequences to benchmarks

## Power and performance analysis

- Accurate power/performance analysis during workload and benchmark cycles requires:
  - Visibility to power activity
  - Accurate analysis
  - Comprehensive debug tools

## Size and complexity of SoCs and Systems

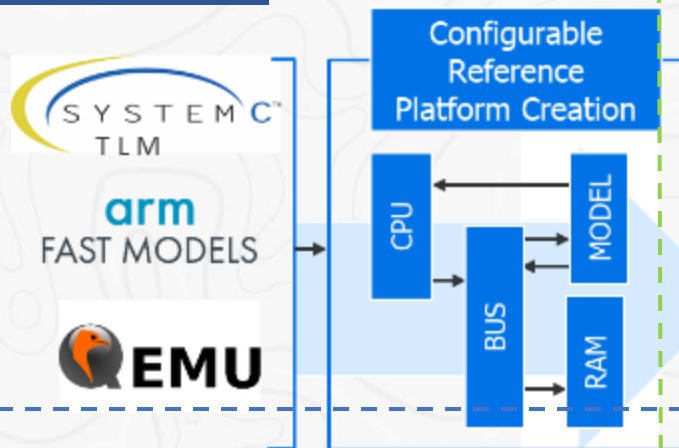
- Billion-gate designs and rapidly growing
- Latest AMD 3<sup>rd</sup> Gen EPYC as an example of SOC size and complexity





# Enabling analysis & insights based on real workloads and benchmarks

## Conf. SW Platform



## Benchmarks running on RTL Design



3DMARK



GFXBench



Geekbench



AnTuTu

GFXBench 4.0, Car Chase, Kishonti



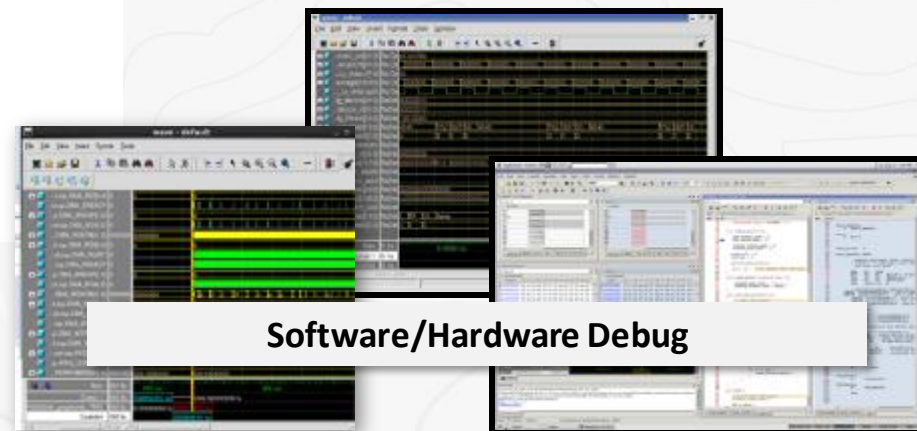
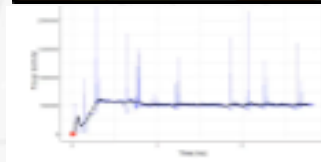
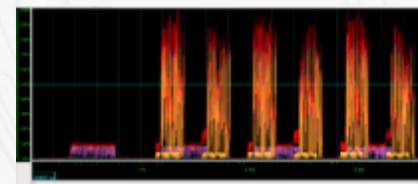
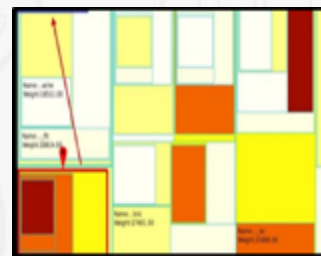
GFXBench 3.1, Manhattan, Kishonti



## HW-Assisted Verification



## Emulation identifies power peaks running real workload benchmarks



## Software/Hardware Debug



# Veloce Apps offering

*Broad portfolio of Apps to address specific verification needs*



## Power App

Early power profiling, analysis, metric tracking and UPF verification

## Coverage App

Accelerate code and functional coverage closure

## Assertion App

Efficient debug of design violations using SVA

## DFT App

Accelerate validation of zero-delay patterns prior to tape out

## Fault App

Accelerate functional safety and ISO 26262 certification

## De-ICE App

Repeatable debug for non-deterministic ICE environments

## ES App

Efficient job scheduling and management of Veloce HW

## Codelink App

Deterministic and non-intrusive offline SW-HW co-debug

## HYCON App

Configurable hybrid, high-speed, ready-to-use SoC reference platform

# Veloce Codelink App

## RTL Driver/Firmware development and debug

- Built-in PC Trace Monitor, Reg/Mem Visualization
- SW Coverage data

## Enables efficient sharing of emulation resources

- Multiple software engineers debug offline in parallel
- Freeing up valuable Emulator resource

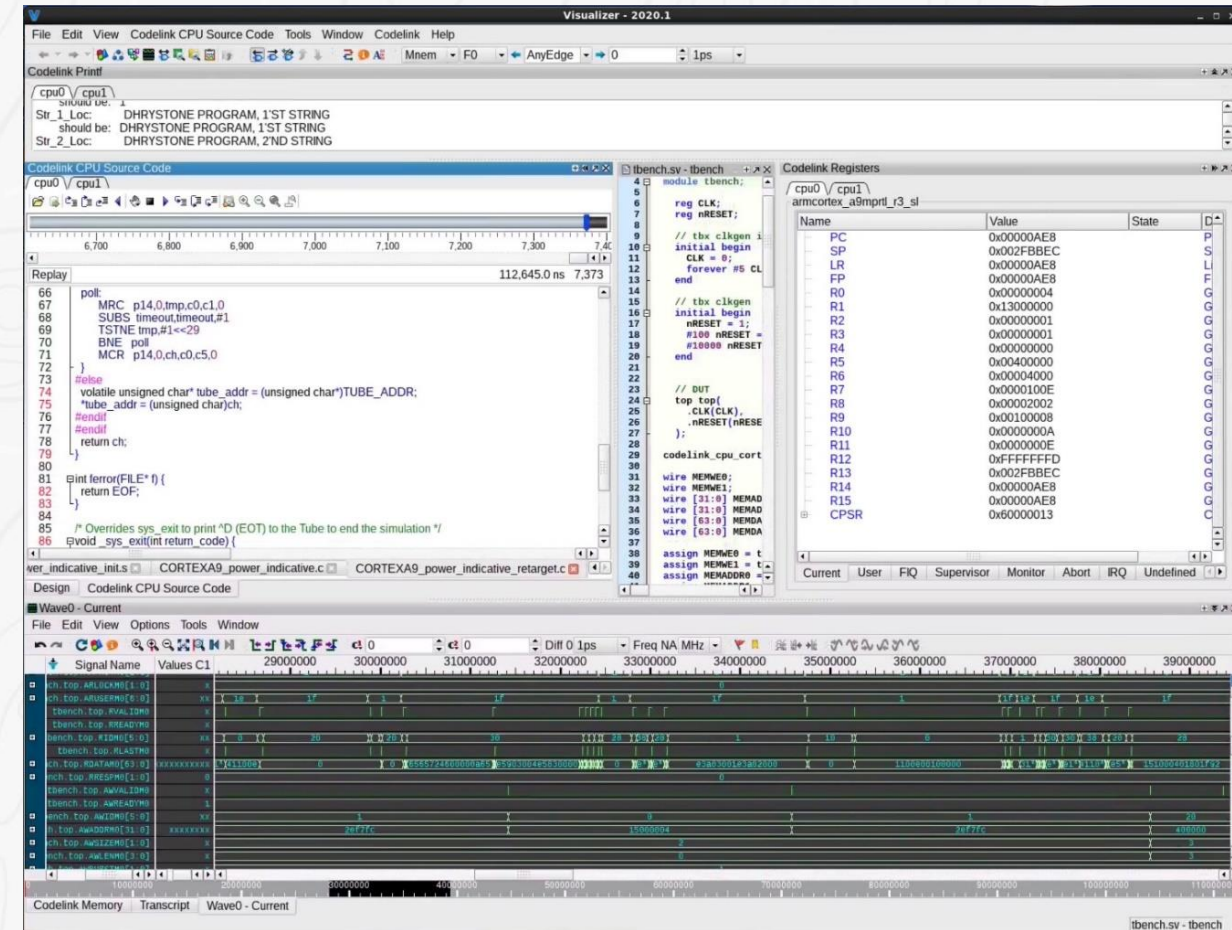
## HW/SW Correlation

- Correlate between events in HW and SW executing
- Power/performance
- Waveforms and protocol analyzer

## Deterministic and Non-intrusive Debug

- with unique forward and backward execution via replay functionality

## SW Code Coverage and SW Performance Profiling



# Veloc Power App: Real scenario and workload-based power analysis

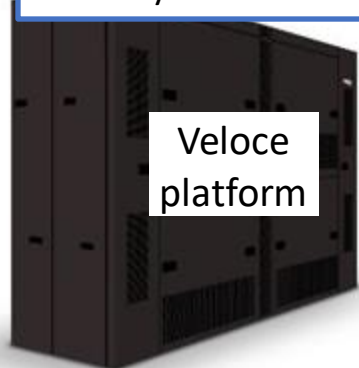
*Early power trend analysis, estimation and sign-off power*

Power profiling at 100s of KHz

Power analysis/ optimization

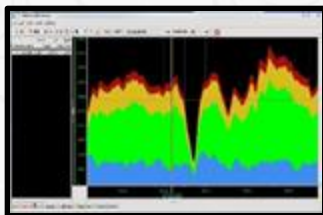
Sign-off power numbers

No compile and visibility restriction



Veloc platform

Veloc activity plot



Design hotspot



File-based

Veloc API  
10x faster  
TTP

RTL/ Gate

Power tools

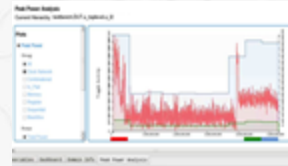
PowerArtist

PowerPro

Average power



Peak power



File-based

Veloc API

Gate-level sign off  
(PrimePower, PowerArtist, PowerPro, ...)

PrimePower (PTPX)  
(Veloc API integration)

Work in Progress

Billions of cycles

Millions of cycles

Thousands of cycles

Profile real workloads on full SoC  
Identify hotspots, peaks, di/dt  
Metric based power Tracking (**CGE/FFE**)

Analyze using Veloc API for real scenarios (FSDB/SAIF)  
Identify/optimize SoC/ IP for power

Sign off power analysis using RTL or Gate switching activity at full SoC



# Summary

Harry Foster



# Three Pillars of a Design+Intent Methodology



## Produce

Produce correct intent by construction



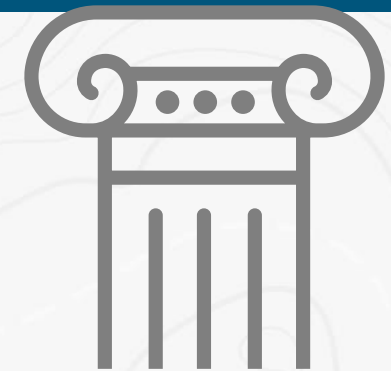
## Prove

Prove intent is met



## Protect

Protect intent throughout development lifecycle



# Three Pillars of a Design+Intent Methodology



## Produce

Produce correct intent by construction



## Prove

Prove intent is met



## Protect

Protect intent throughout development lifecycle



# Catapult High-Level Synthesis & Verification

*Comprehensive flow providing needed productivity gains*



HLL AND HLS > 10X REDUCTION IN  
CODE RESULTING IN A REDUCTION  
OF BUG DENSITY



HLV RESULTED IN TWO ORDERS OF  
MAGNITUDE SPEEDUP IN  
SIMULATION



EFFICIENCIES GAINED BY DESIGNING  
AND VERIFYING VIA HLL.  
PRODUCTIVITY GAINS VIA FASTER  
AND PREDICTABLE POST-HLS RTL  
VERIFICATION SIGNOFF

# Three Pillars of a Design+Intent Methodology



## Produce

Produce correct intent by construction



## Prove

Prove intent is met



## Protect

Protect intent throughout development lifecycle





# Using intent-focused insight flushes issues before simulation

*Reduce bug density with a non-simulation verification methodology*

## Analyze code for issues

Find RTL mistakes  
Review initialization

## Check async operations

Verify CDCs  
Verify RDCs

## Review construction

Verify interconnects  
Verify registers

## Confirm vs specification

Check against specs  
Look for trojans

## Verify implementation

Review downstream netlist  
for new issues

# Three Pillars of a Design+Intent Methodology



## Produce

Produce correct intent by construction



## Prove

Prove intent is met



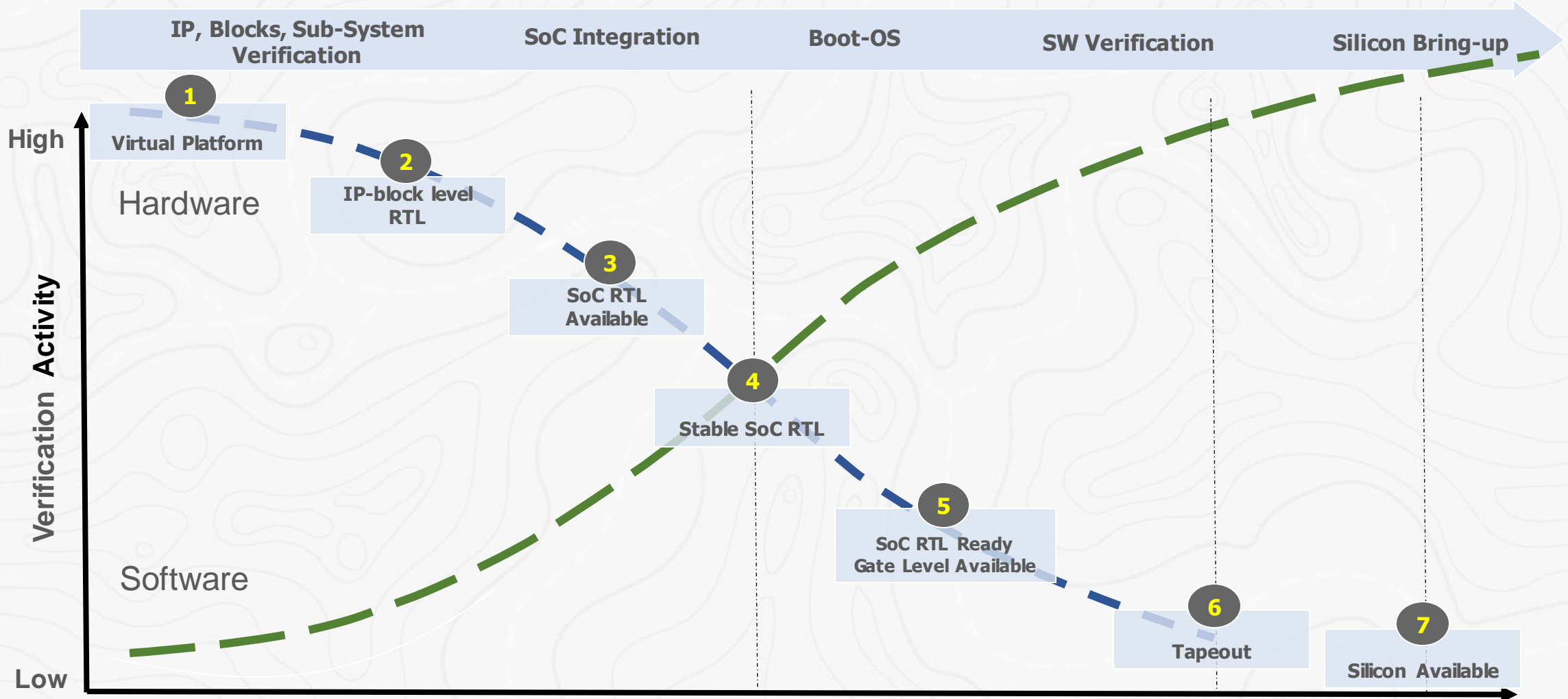
## Protect

Protect intent throughout development lifecycle



# Protect Intent Throughout the Development Lifecycle

*Emulation is critical in the age of SoC verification and validation*





# The Best Verification Strategy You've Never Heard Of

## Q & A



# Disclaimer

© Siemens 2022

Subject to changes and errors. The information given in this document only contains general descriptions and/or performance features which may not always specifically reflect those described, or which may undergo modification in the course of further development of the products. The requested performance features are binding only when they are expressly agreed upon in the concluded contract.

All product designations may be trademarks or other rights of Siemens AG, its affiliated companies or other companies whose use by third parties for their own purposes could violate the rights of the respective owner.