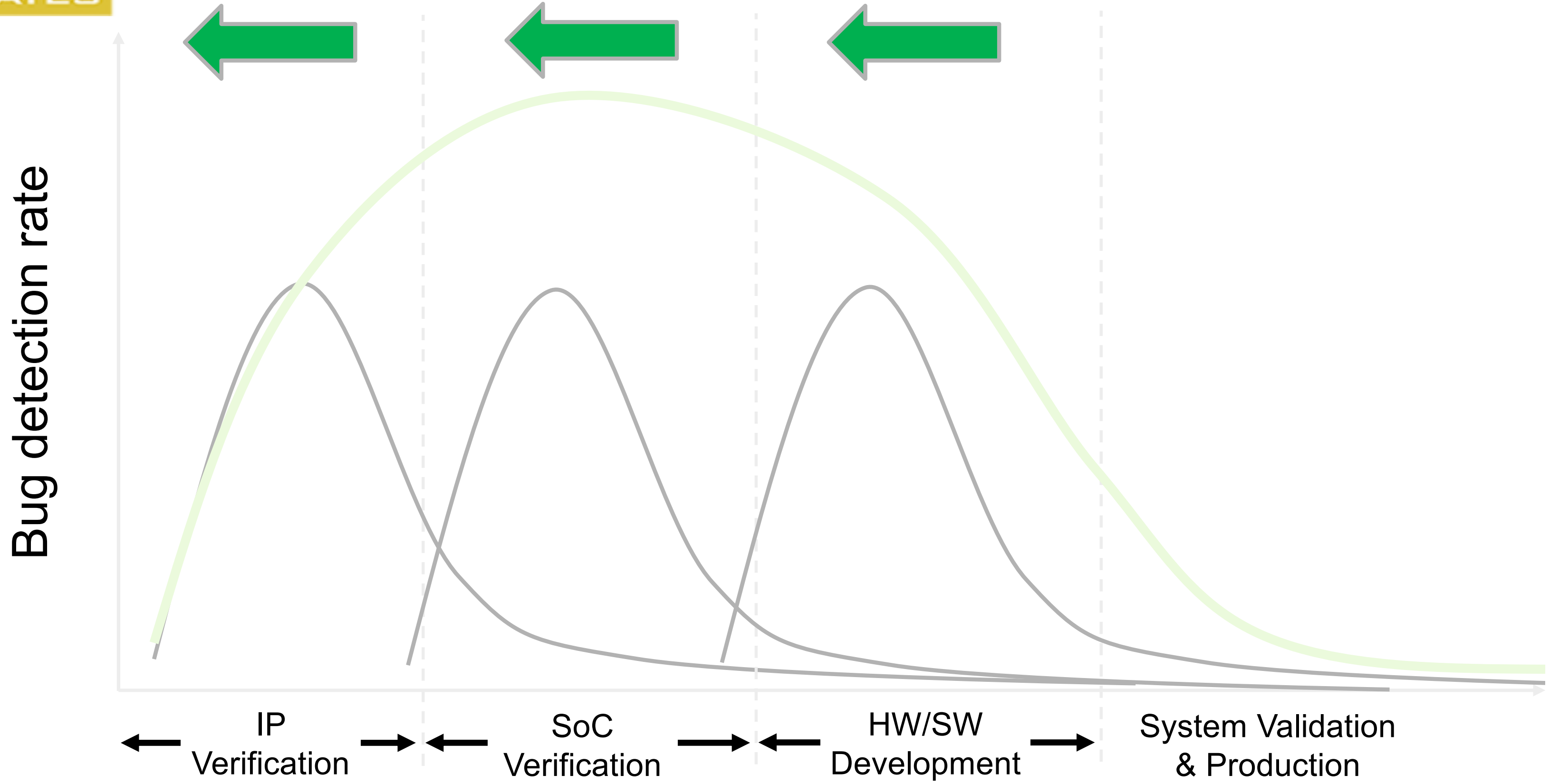
SoC Verification Speed – More is Better

Fernanda Braga - Cadence Design Systems, Inc.
John Rose - Cadence Design Systems, Inc.
William Winkeler - Cadence Design Systems, Inc.
Sharon Rosenberg - Cadence Design Systems, Inc.
Frank Schirrmeister - Cadence Design Systems, Inc.

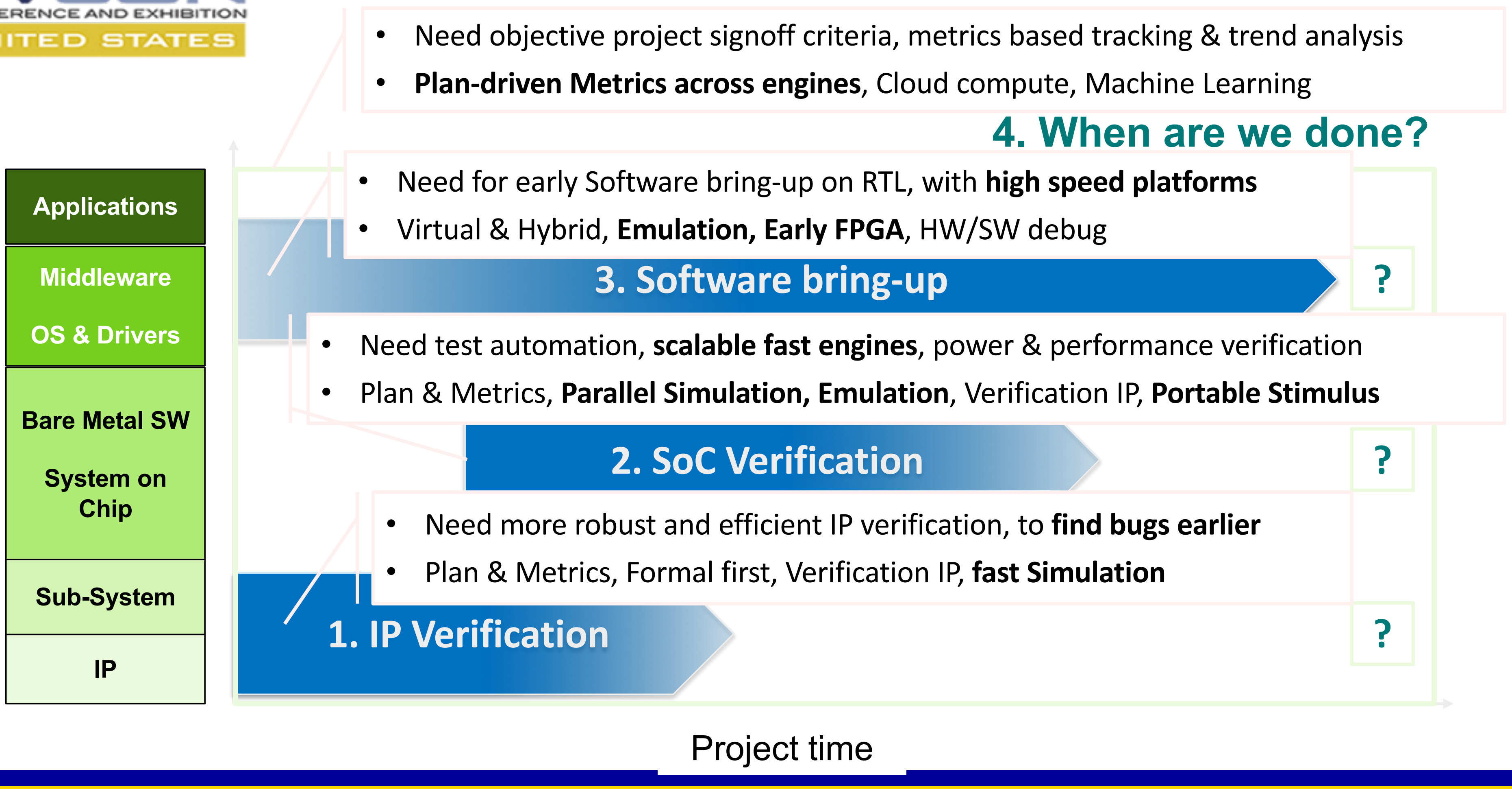
Agenda

- The Need for Speed
- Formal methods to avoid sim cycles
- Coding for max sim speed
- Speeding power + mixed-signal SoC
- Break
- Portable Stimulus for faster verification
- Applying hardware to speed system verification
- Summary and call to action

Bug Detection Still not as Early as Possible



Key Verification Challenges to Address



- Need objective project signoff criteria, metrics based tracking & trend analysis
- **Plan-driven Metrics across engines**, Cloud compute, Machine Learning

4. When are we done?

- Need for early Software bring-up on RTL, with **high speed platforms**
- Virtual & Hybrid, **Emulation, Early FPGA**, HW/SW debug

3. Software bring-up

?

- Need test automation, **scalable fast engines**, power & performance verification
- Plan & Metrics, **Parallel Simulation, Emulation**, Verification IP, **Portable Stimulus**

2. SoC Verification

?

- Need more robust and efficient IP verification, to **find bugs earlier**
- Plan & Metrics, Formal first, Verification IP, **fast Simulation**

1. IP Verification

?

Applications

Middleware

OS & Drivers

Bare Metal SW

System on Chip

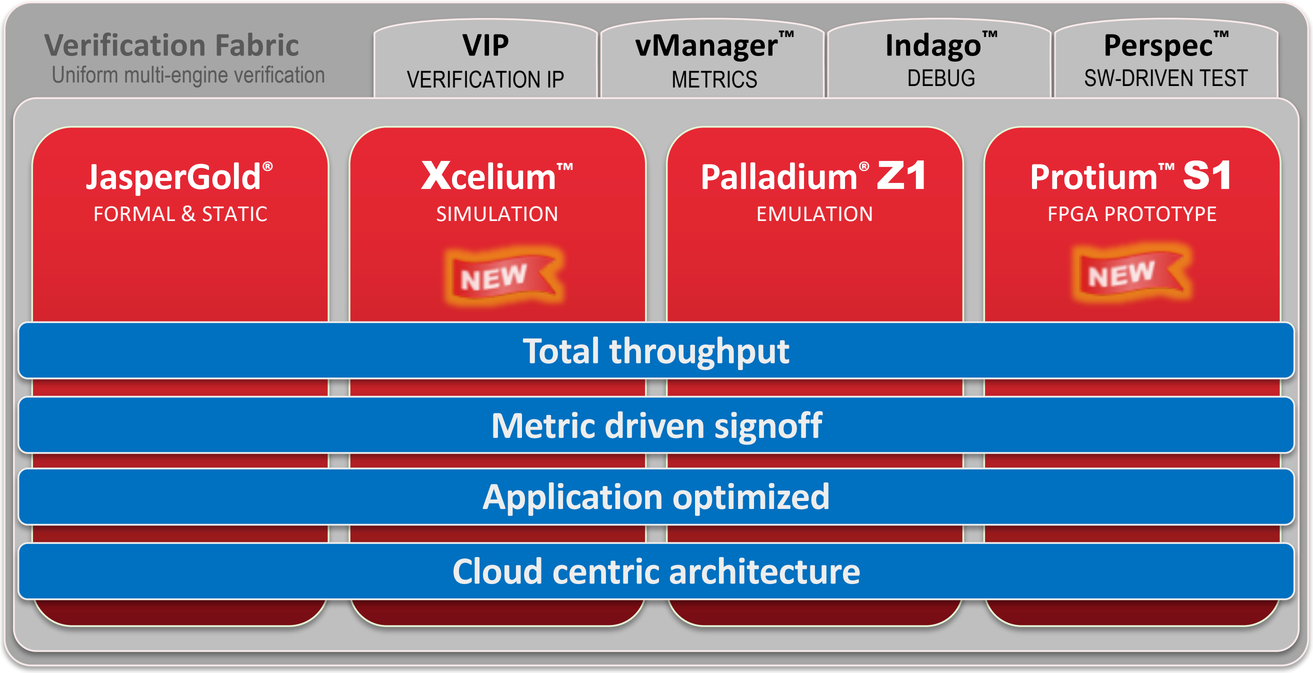
Sub-System

IP

Project time

Verification Suite

Technology innovation leadership: *Fast, Smart, and Optimized*



Verification Fabric
Uniform multi-engine verification

VIP
VERIFICATION IP

vManager™
METRICS

Indago™
DEBUG

Perspec™
SW-DRIVEN TEST

JasperGold®
FORMAL & STATIC

Xcelium™
SIMULATION
NEW

Palladium® Z1
EMULATION

Protium™ S1
FPGA PROTOTYPE
NEW

Total throughput

Metric driven signoff

Application optimized

Cloud centric architecture

- ***Fast*** **Best-in-class engines**
- ***Smart*** **Flow-driven engine integrations**
- ***Optimized*** **comprehensive solutions**

Agenda

• The Need for Speed	
• Formal methods to avoid sim cycles	-Fernanda Braga
• Coding for max sim speed	-John Rose
• Speeding power + mixed-signal SoC	-William Winkeler
• Break	
• Portable Stimulus for faster verification	-Sharon Rosenberg
• Applying hardware to speed system verification	-Frank Schirrmeister
• Summary and call to action	

Session Objectives

- Overview on how formal can speed up verification process
- Introduce Designer Formal Verification flow
- Discuss when to use formal for maximized productivity
- Introduce methodology to address Formal IP Signoff

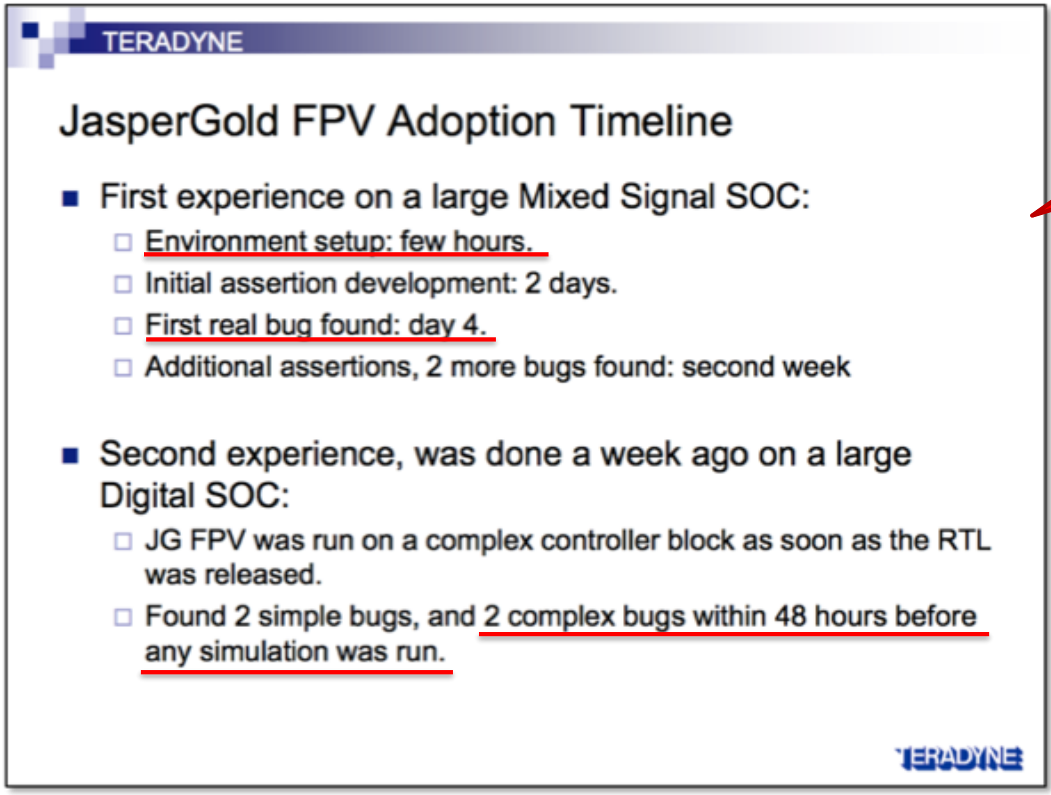
How Can Formal Help

- For many DV engineers their preferred verification method (simulation) is a hammer and everything looks like a nail
- The reality is
  - Many users are already using formal as a sign-off tool for certain blocks and problems
  - There are categories of designs which favor simulation and others which favor formal

- Formal, applied to the right designs and problems, can achieve significant ***productivity*** and ***quality*** gains in the overall verification flow
  - Especially when simulation-like rigorous verification planning and coverage closure methodologies are applied

"FV wherever we can, simulate where we must" – Erik Seligman, JUG 2016

Case Study: Teradyne



TERADYNE

JasperGold FPV Adoption Timeline

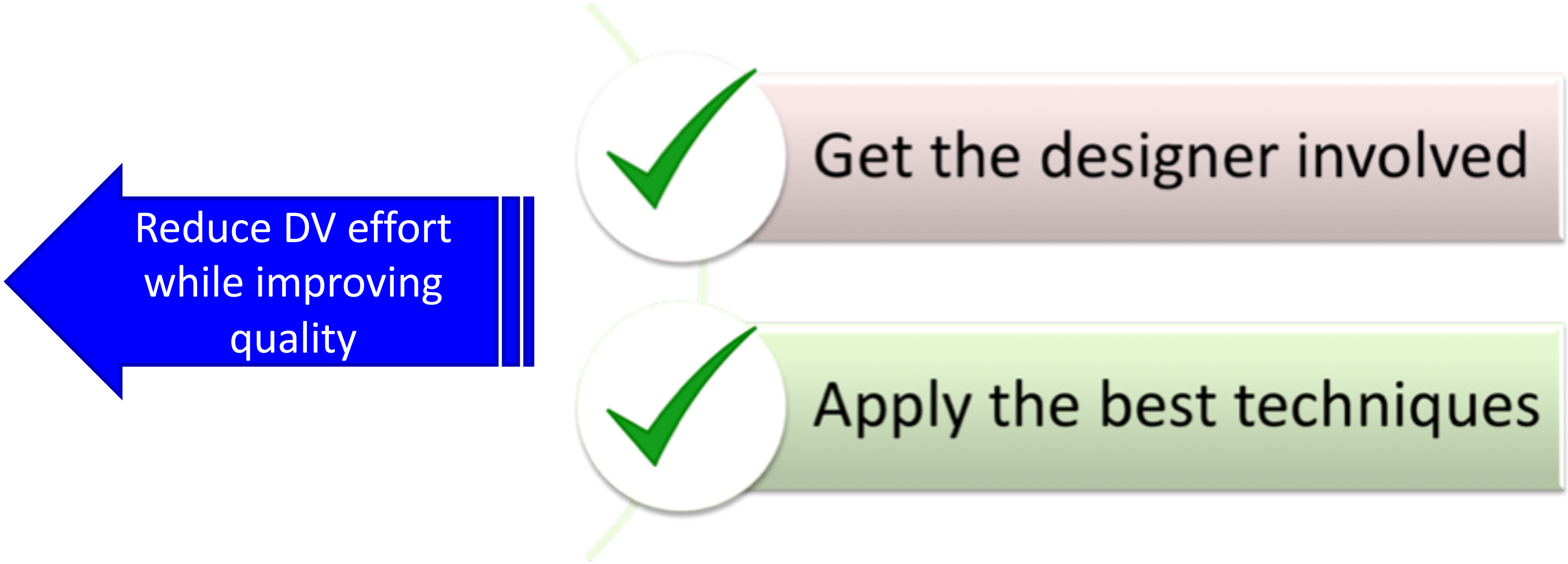
- First experience on a large Mixed Signal SOC:
  - Environment setup: few hours.
  - Initial assertion development: 2 days.
  - First real bug found: day 4.
  - Additional assertions, 2 more bugs found: second week
- Second experience, was done a week ago on a large Digital SOC:
  - JG FPV was run on a complex controller block as soon as the RTL was released.
  - Found 2 simple bugs, and 2 complex bugs within 48 hours before any simulation was run.

TERADYNE

Speed!

Source: Teradyne presentation at CDNLive Boston, Nov 2017

Formal Speeds Verification



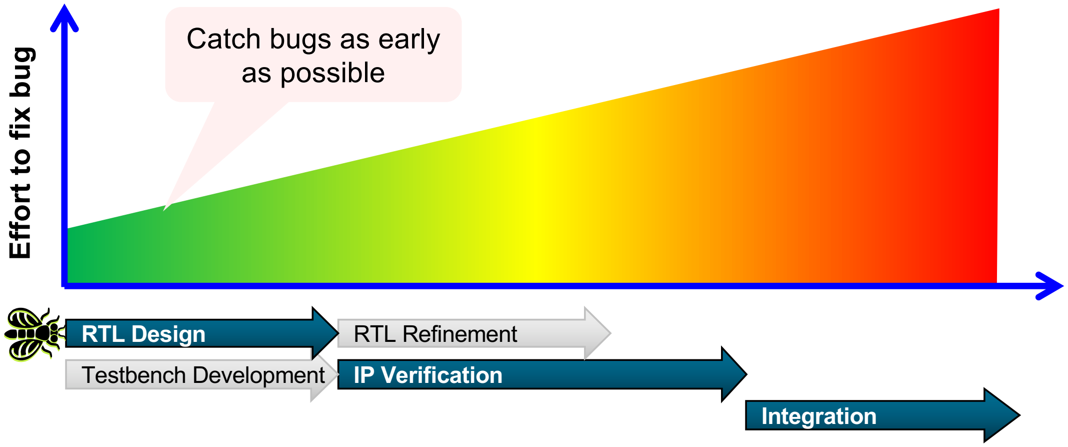
Reduce DV effort while improving quality

✓ Get the designer involved

✓ Apply the best techniques

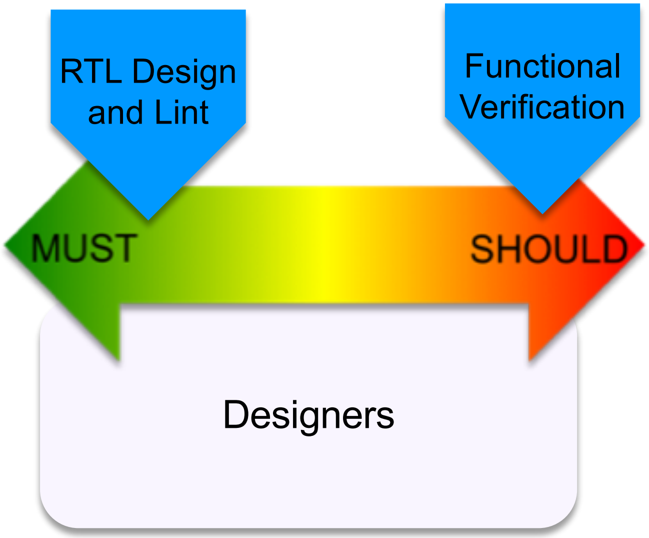
Cost of Finding Bugs

- Effort to fix a bug increases significantly the further into the development cycle



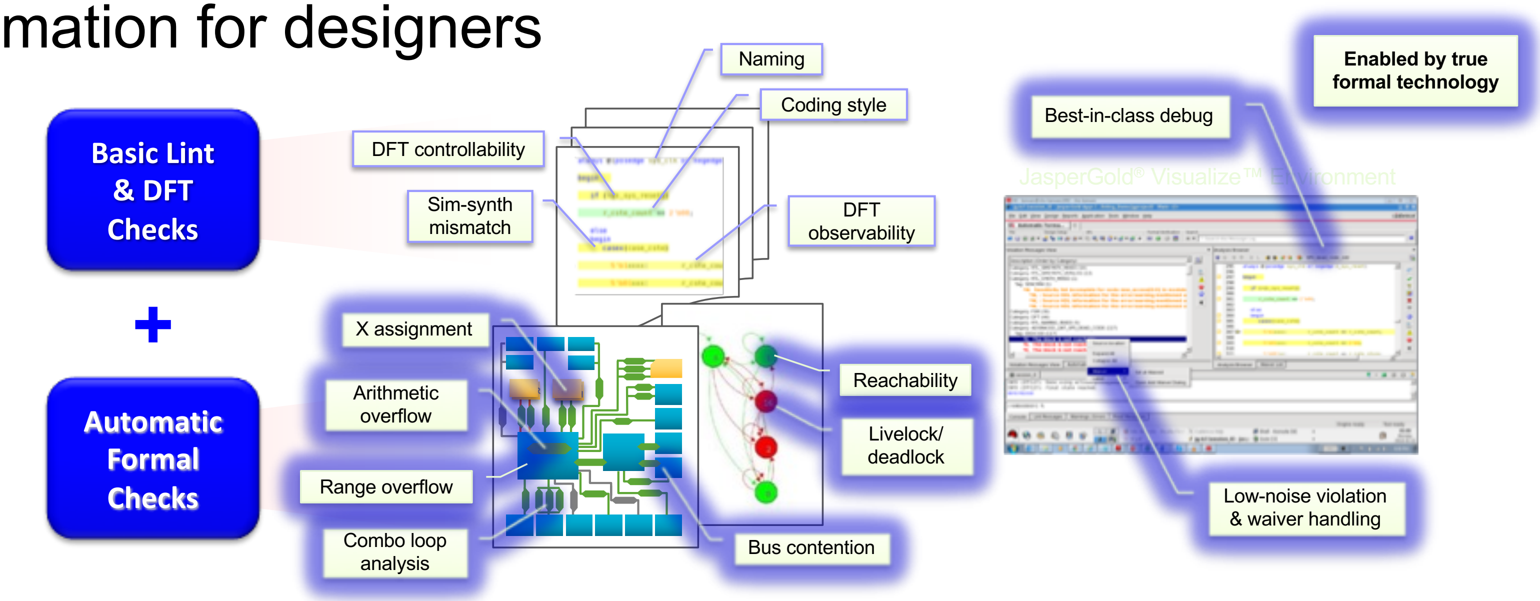
The Challenge With Designers

- Managers, verification engineers and even designers all agree that designers ***SHOULD*** get more involved in verification
- Reality is that RTL design, implementation tasks, etc. ***MUST*** get done
- Conclusion: Only successful way to get designers involved in functional verification is automation



JasperGold Superlint: Hand-off *Robust Reusable RTL*

- Automation for designers



Basic Lint & DFT Checks

+

Automatic Formal Checks

Naming

Coding style

DFT controllability

Sim-synth mismatch

DFT observability

X assignment

Arithmetic overflow

Range overflow

Combo loop analysis

Reachability

Livelock/ deadlock

Bus contention

Enabled by true formal technology

Best-in-class debug

JasperGold® Visualize™ Environment

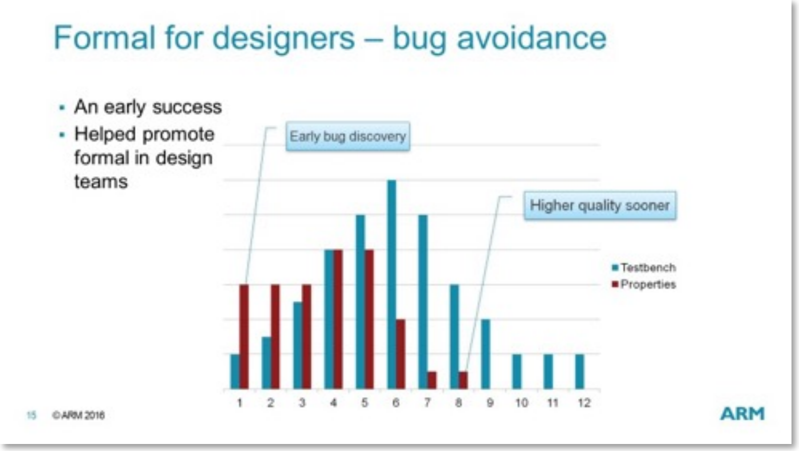
Low-noise violation & waiver handling

Comprehensive functional checks, violation debug & waiver handling based on best-in-class formal analysis

Superlint App: Success Story

- ARM

"We've been using the JasperGold Superlint App at ARM for more than a year, and we've had success with improving RTL signoff and shortening time to market. With the ability to find bugs weeks earlier in the design process, we've reduced late-stage RTL changes, which enables the team to save additional time when we get to the functional verification stage."



Formal for designers – bug avoidance

- An early success
- Helped promote formal in design teams

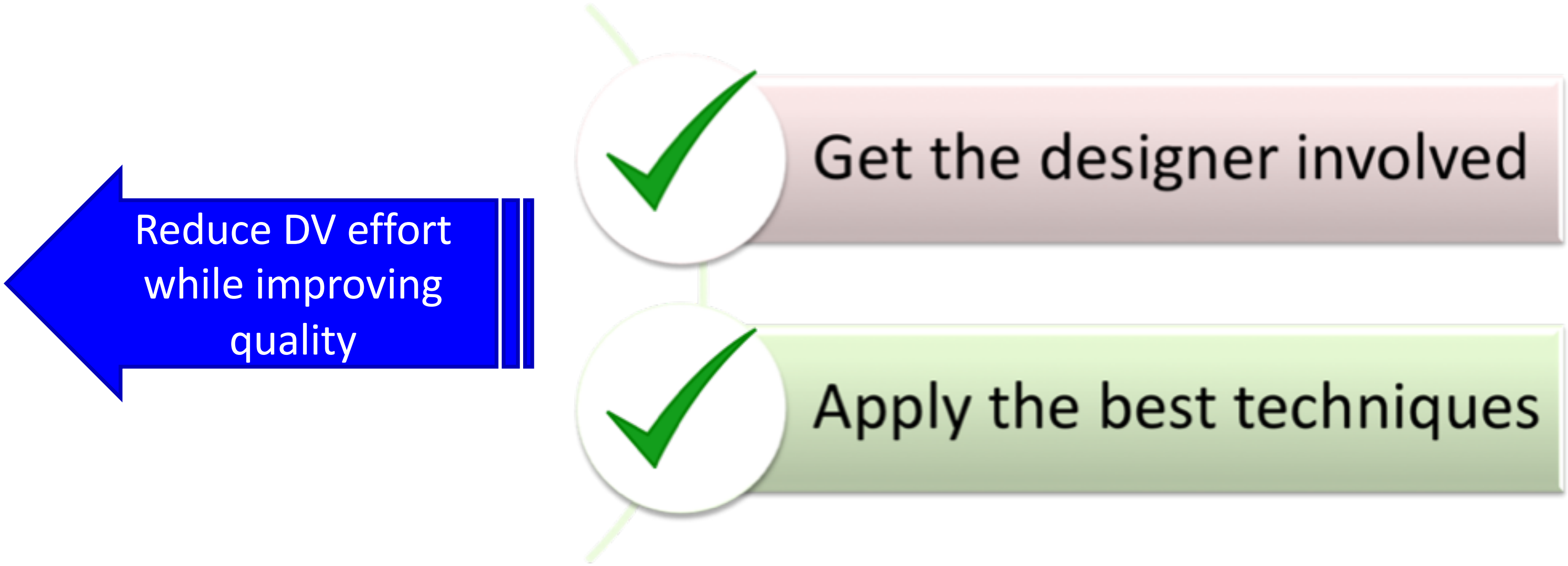
15 ©ARM 2016

Hobson Bullman
Vice President and General Manager
Technology Services Group, ARM

ARM

Source: ARM keynote presentation at Jasper User Group, Nov 2016

Formal Speeds Verification

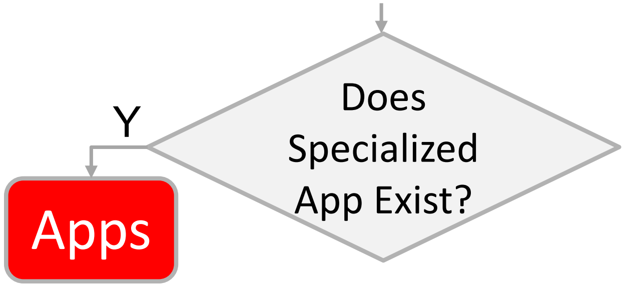


Reduce DV effort while improving quality

✓ Get the designer involved

✓ Apply the best techniques

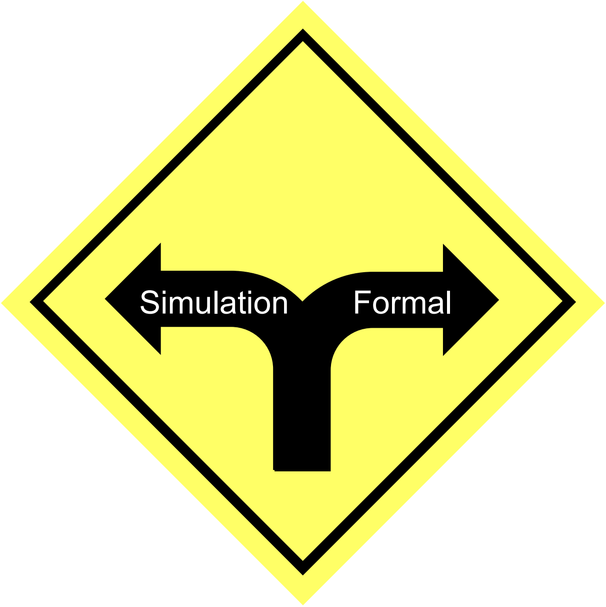
Apply The Best Techniques



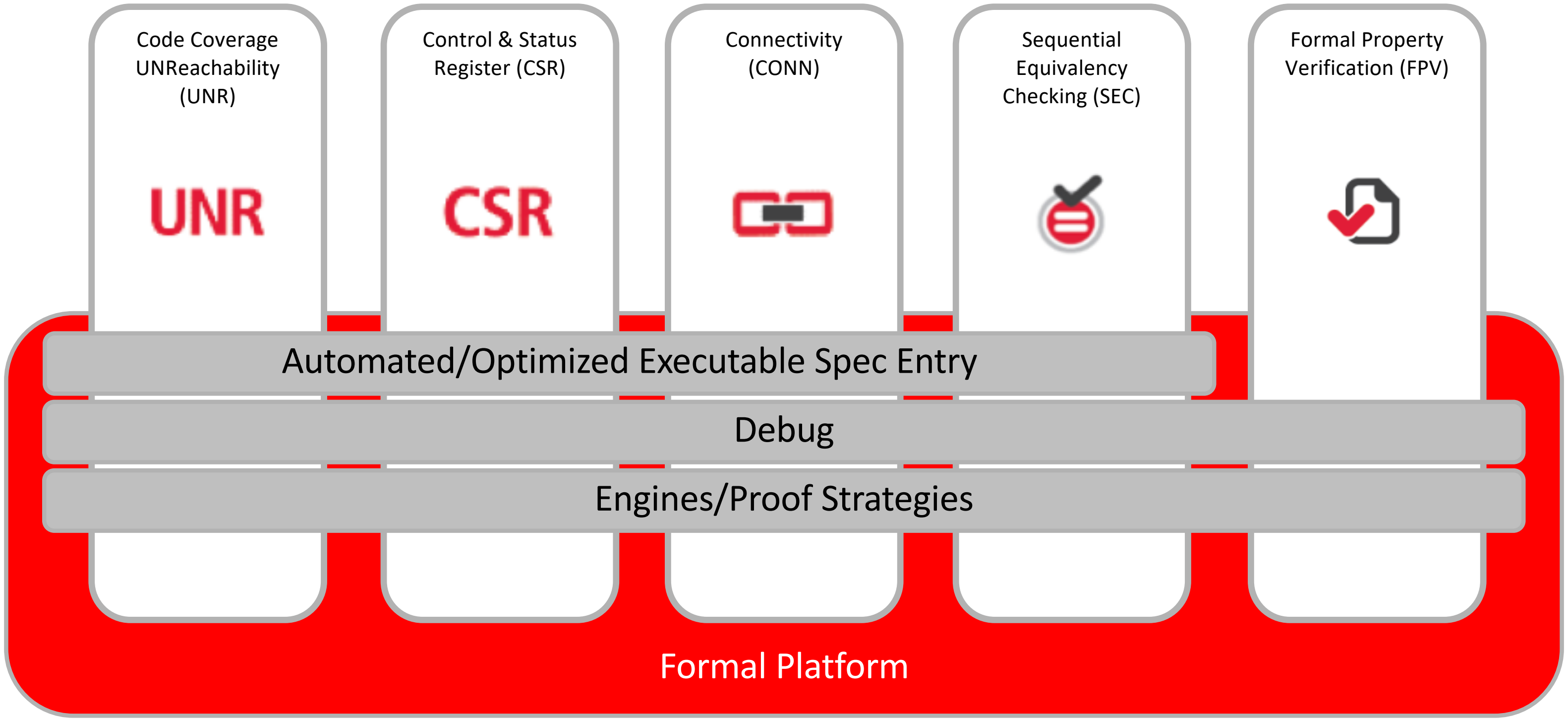
Does Specialized App Exist?

Y

Apps

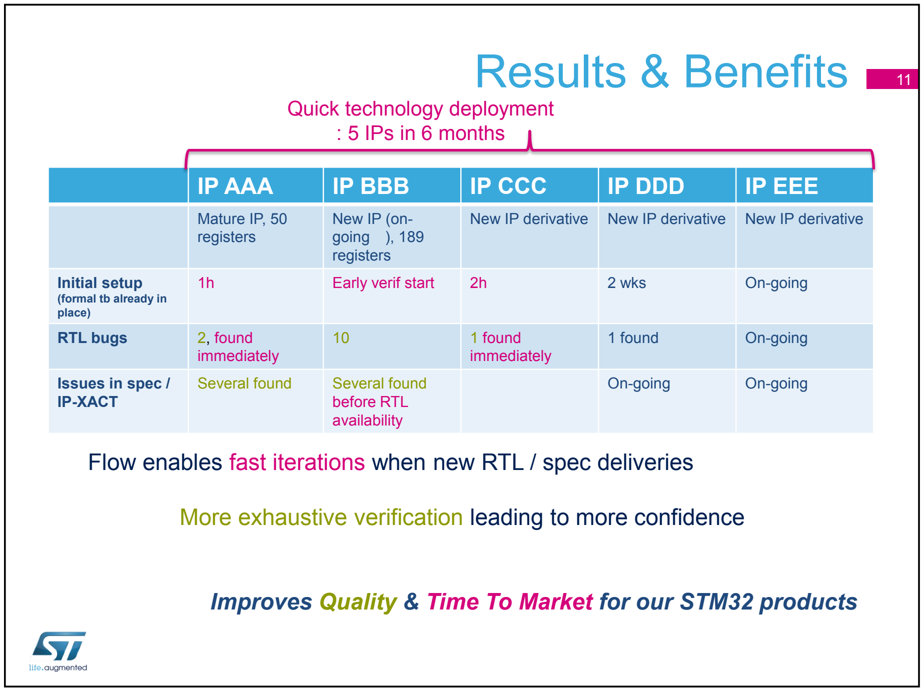


What Is A Formal App?


Code Coverage UNReachability (UNR)
UNR
Control & Status Register (CSR)
CSR
Connectivity (CONN)
Sequential Equivalency Checking (SEC)
Formal Property Verification (FPV)
Automated/Optimized Executable Spec Entry
Debug
Engines/Proof Strategies
Formal Platform

2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

CSR App: Success Story



Results & Benefits

11

Quick technology deployment
: 5 IPs in 6 months

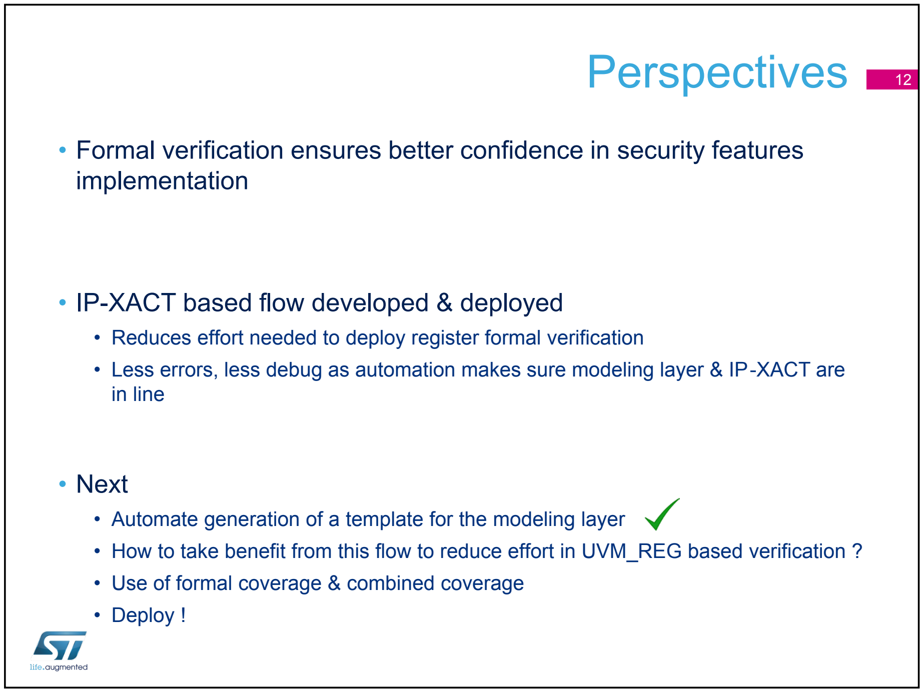
	IP AAA	IP BBB	IP CCC	IP DDD	IP EEE
	Mature IP, 50 registers	New IP (on-going), 189 registers	New IP derivative	New IP derivative	New IP derivative
Initial setup **(formal tb already in place)**	1h	Early verif start	2h	2 wks	On-going
RTL bugs	2, found immediately	10	1 found immediately	1 found	On-going
Issues in spec / IP-XACT	Several found	Several found before RTL availability		On-going	On-going

Flow enables fast iterations when new RTL / spec deliveries

More exhaustive verification leading to more confidence

Improves Quality & Time To Market for our STM32 products

life.augmented



Perspectives

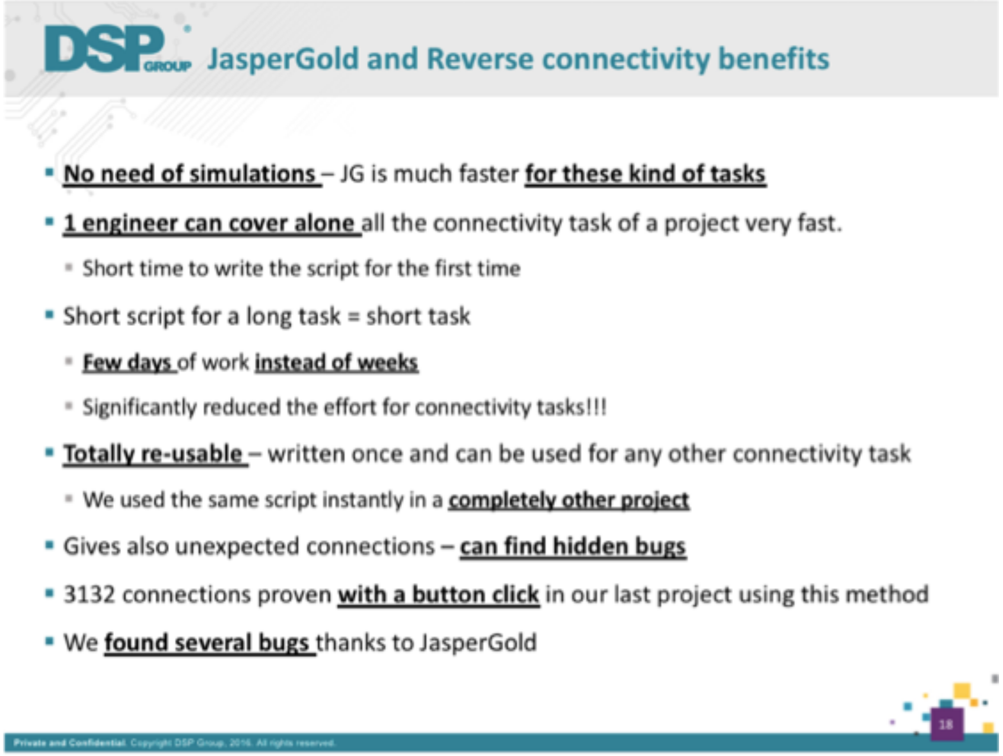
12

- Formal verification ensures better confidence in security features implementation
- IP-XACT based flow developed & deployed
  - Reduces effort needed to deploy register formal verification
  - Less errors, less debug as automation makes sure modeling layer & IP-XACT are in line
- Next
  - Automate generation of a template for the modeling layer ✓
  - How to take benefit from this flow to reduce effort in UVM_REG based verification ?
  - Use of formal coverage & combined coverage
  - Deploy !

life.augmented

Source: STMicroelectronics presentation at CDNLive 2017

CONN App: Success Story



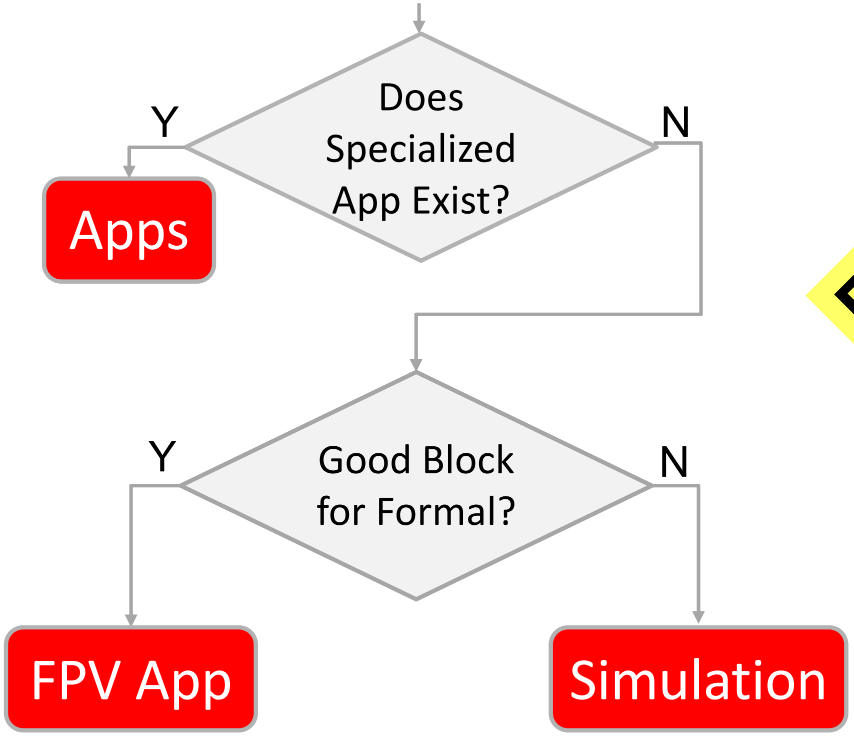
DSP GROUP JasperGold and Reverse connectivity benefits

- **No need of simulations** – JG is much faster **for these kind of tasks**
- **1 engineer can cover alone** all the connectivity task of a project very fast.
  - Short time to write the script for the first time
- Short script for a long task = short task
  - **Few days** of work **instead of weeks**
  - Significantly reduced the effort for connectivity tasks!!!
- **Totally re-usable** – written once and can be used for any other connectivity task
  - We used the same script instantly in a **completely other project**
- Gives also unexpected connections – **can find hidden bugs**
- 3132 connections proven **with a button click** in our last project using this method
- We **found several bugs** thanks to JasperGold

Private and Confidential. Copyright DSP Group, 2016. All rights reserved. 18

Source: DSP Group presentation at CDNLive 2016

Apply The Best Techniques



Does Specialized App Exist?

Y

Apps

N

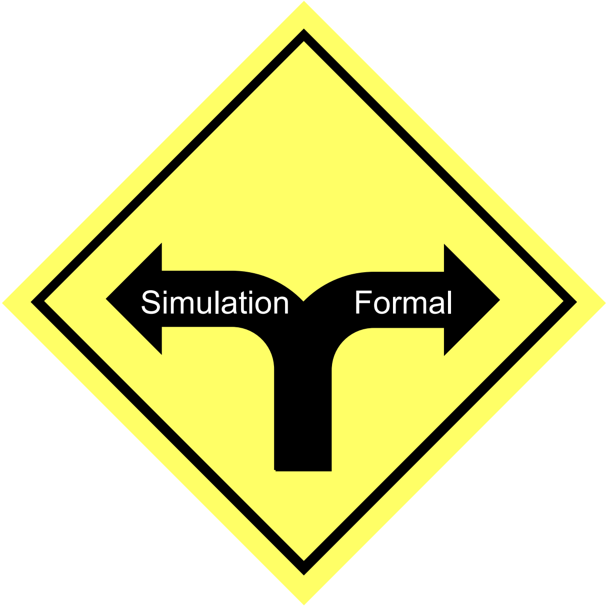
Good Block for Formal?

Y

FPV App

N

Simulation

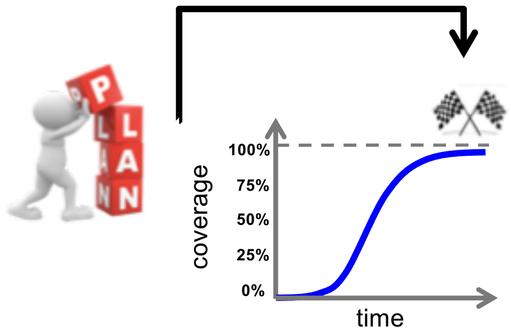


Formal IP Signoff

Metric-Driven Verification!

- Definition already well established in industry

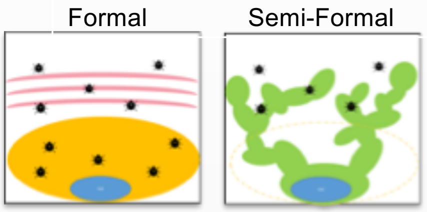
No checks fail while reaching all coverage



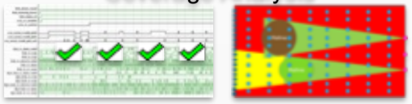
- Signoff is all about confidence
  - IMPORTANT: Finding bugs
  - **CRITICAL**: Finding no bugs while reaching a measurable, planned set of coverage
- Required:
  - Verification plan specifying checks and coverage to measure progress and define done
  - Technology and methodology to achieve signoff

Formal Signoff Summary

Technology

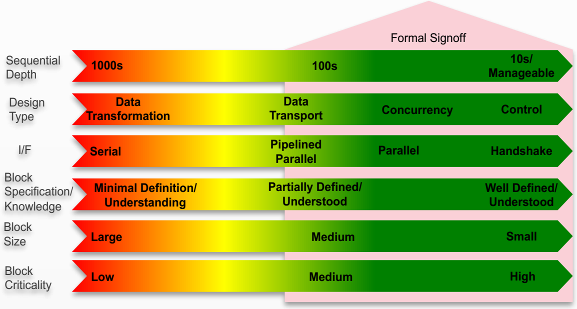


Coverage Analysis



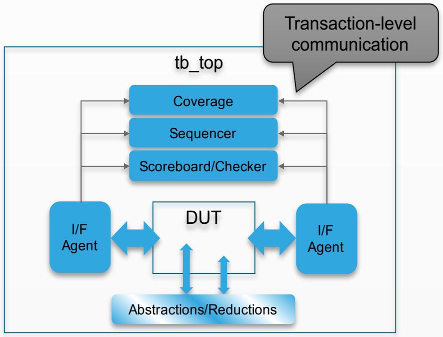
+

Repeatable Methodology



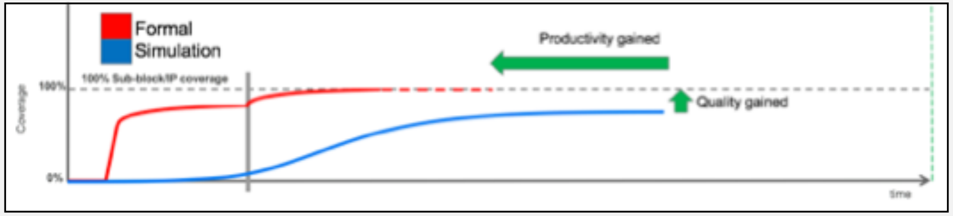
Formal Signoff

Sequential Depth	1000s	100s		10s/ Manageable
Design Type	Data Transformation	Data Transport	Concurrency	Control
I/F	Serial	Pipelined Parallel	Parallel	Handshake
Block Specification/ Knowledge	Minimal Definition/ Understanding	Partially Defined/ Understood		Well Defined/ Understood
Block Size	Large	Medium		Small
Block Criticality	Low	Medium		High



=

Metric-Based Sign-off Solution



For amenable blocks

Quality: formal >> sim coverage

Productivity: Time to signoff << sim

vPlan

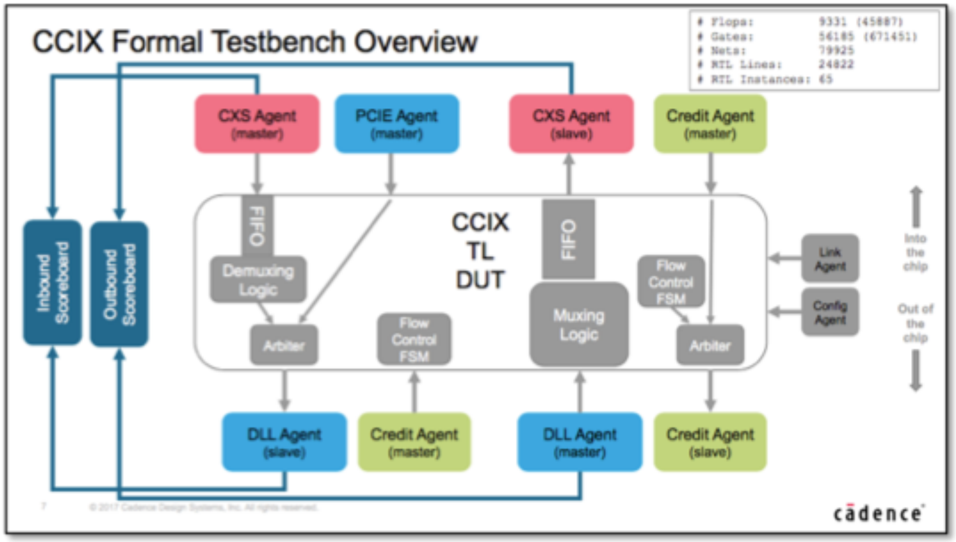
No checks fail while reaching all coverage

JUG: Coverage-Driven Formal Verification Signoff on CCIX Design

- Partnership with IP Group at Cadence

Repeatable methodology applied to create testbench

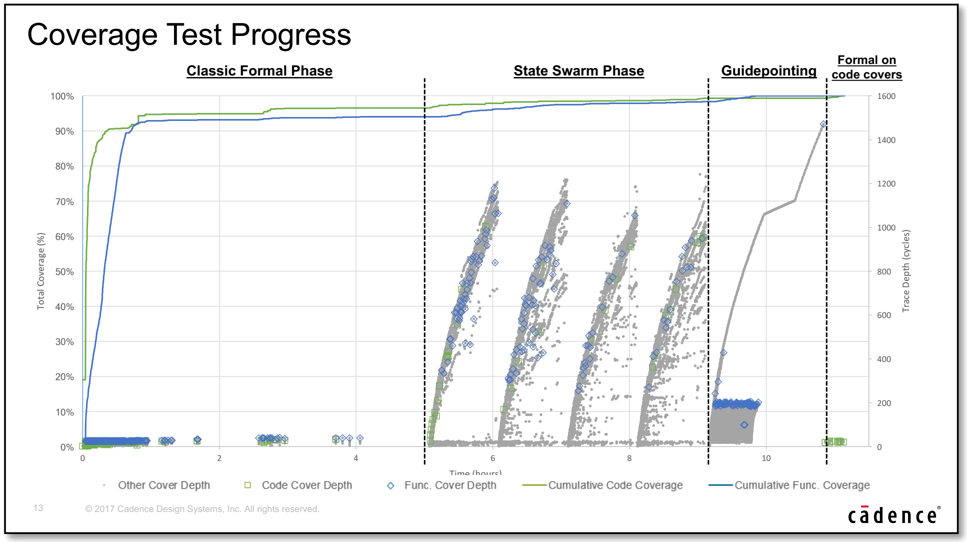
Metric-Driven Verification approach: Coverage Closure!



CCIX Formal Testbench Overview

```
# Flops:          9331 (45887)
# Gates:          56185 (671451)
# Nets:           79925
# RTL Lines:      24822
# RTL Instances:  65
```

© 2017 Cadence Design Systems, Inc. All rights reserved.



Coverage Test Progress

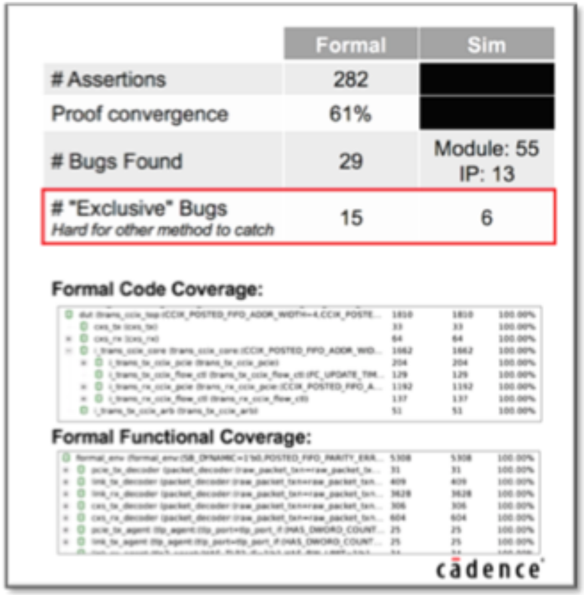
© 2017 Cadence Design Systems, Inc. All rights reserved.

Source: Cadence IPG presentation at JUG 2017

JUG: Coverage-Driven Formal Verification Signoff on CCIX Design

- Partnership with IP Group at Cadence

High quality bugs found



	Formal	Sim
# Assertions	282	[redacted]
Proof convergence	61%	[redacted]
# Bugs Found	29	Module: 55 IP: 13
# "Exclusive" Bugs *Hard for other method to catch*	15	6

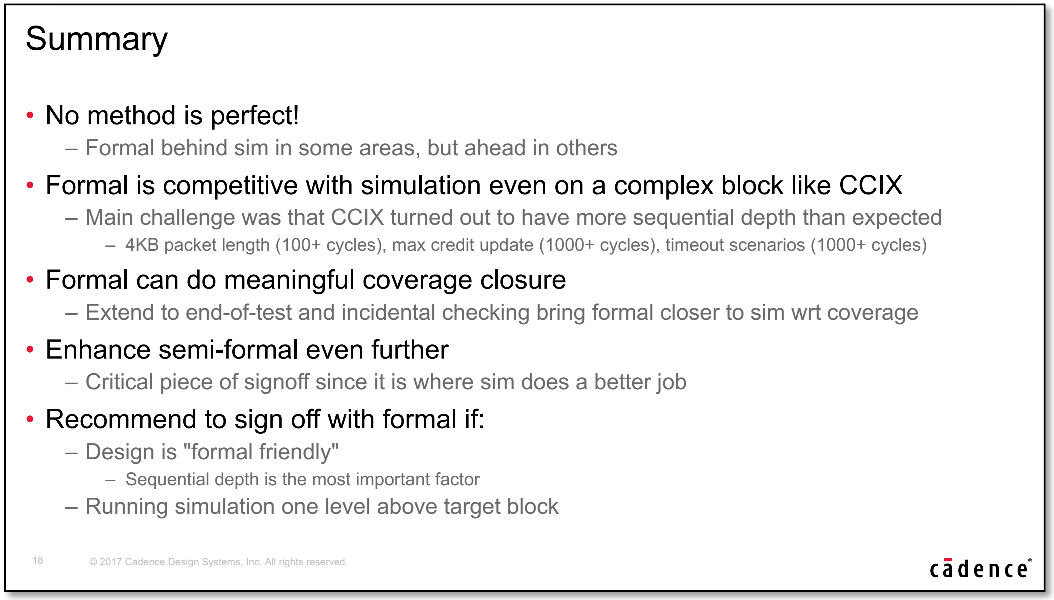
Formal Code Coverage:

dut (trans_ccix_top:(CCIX_POSTED_FIFO_ADDR_WIDTH=4,CCIX_POSTE...	1810	1810	100.00%
cxs_tx (cxs_tx)	33	33	100.00%
cxs_rx (cxs_rx)	64	64	100.00%
i_trans_ccix_core (trans_ccix_core:(CCIX_POSTED_FIFO_ADDR_WID...	1662	1662	100.00%
i_trans_tx_ccix_pcie (trans_tx_ccix_pcie)	204	204	100.00%
i_trans_tx_ccix_flow_ctl (trans_tx_ccix_flow_ctl:(FC_UPDATE_TIM...	129	129	100.00%
i_trans_rx_ccix_pcie (trans_rx_ccix_pcie:(CCIX_POSTED_FIFO_A...	1192	1192	100.00%
i_trans_rx_ccix_flow_ctl (trans_rx_ccix_flow_ctl)	137	137	100.00%
i_trans_tx_ccix_arb (trans_tx_ccix_arb)	51	51	100.00%

Formal Functional Coverage:

formal_env (formal_env:(SB_DYNAMIC=1'b0,POSTED_FIFO_PARITY_ERR...	5308	5308	100.00%
pcie_tx_decoder (packet_decoder:(raw_packet_tx=raw_packet_tx...	31	31	100.00%
link_tx_decoder (packet_decoder:(raw_packet_txn=raw_packet_txn...	409	409	100.00%
link_rx_decoder (packet_decoder:(raw_packet_txn=raw_packet_txn...	3628	3628	100.00%
cxs_tx_decoder (packet_decoder:(raw_packet_txn=raw_packet_txn...	306	306	100.00%
cxs_rx_decoder (packet_decoder:(raw_packet_txn=raw_packet_txn...	604	604	100.00%
pcie_tx_agent (tlp_agent:(tlp_port=tlp_port_if:(HAS_DWORD_COUNT...	25	25	100.00%
link_tx_agent (tlp_agent:(tlp_port=tlp_port_if:(HAS_DWORD_COUNT...	25	25	100.00%

Formal as viable option for IP signoff in amenable targets



Summary

- No method is perfect!
  - Formal behind sim in some areas, but ahead in others
- Formal is competitive with simulation even on a complex block like CCIX
  - Main challenge was that CCIX turned out to have more sequential depth than expected
    - 4KB packet length (100+ cycles), max credit update (1000+ cycles), timeout scenarios (1000+ cycles)
- Formal can do meaningful coverage closure
  - Extend to end-of-test and incidental checking bring formal closer to sim wrt coverage
- Enhance semi-formal even further
  - Critical piece of signoff since it is where sim does a better job
- Recommend to sign off with formal if:
  - Design is "formal friendly"
    - Sequential depth is the most important factor
  - Running simulation one level above target block

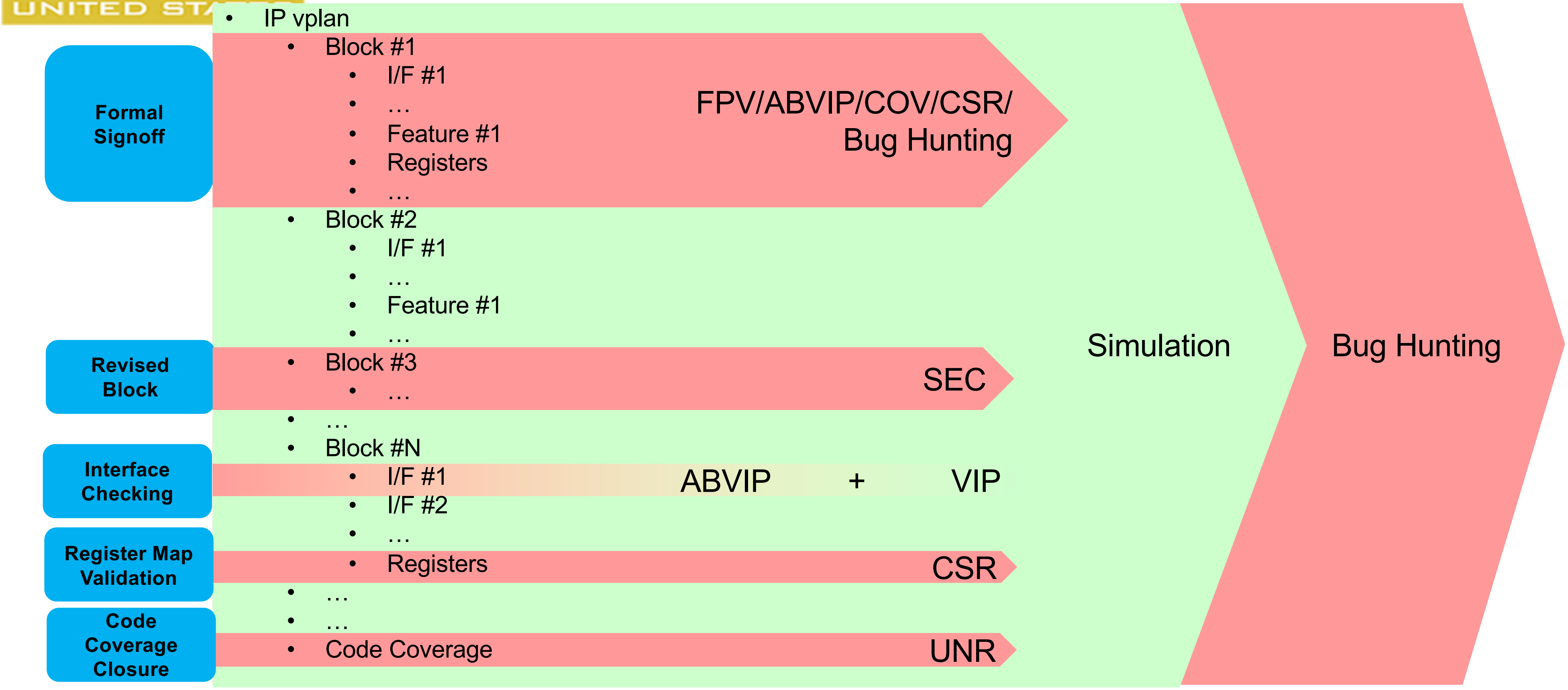
18 © 2017 Cadence Design Systems, Inc. All rights reserved.

Source: Cadence IPG presentation at JUG 2017

Maximum Speed

🟩 Simulation

🟥 Formal



Formal Signoff

Revised Block

Interface Checking

Register Map Validation

Code Coverage Closure

- IP vplan
  - Block #1
    - I/F #1
    - …
    - Feature #1
    - Registers
    - …
  - Block #2
    - I/F #1
    - …
    - Feature #1
    - …
  - Block #3
    - …
  - …
  - Block #N
    - I/F #1
    - I/F #2
    - …
    - Registers
  - …
  - …
  - Code Coverage

FPV/ABVIP/COV/CSR/
Bug Hunting

SEC

ABVIP + VIP

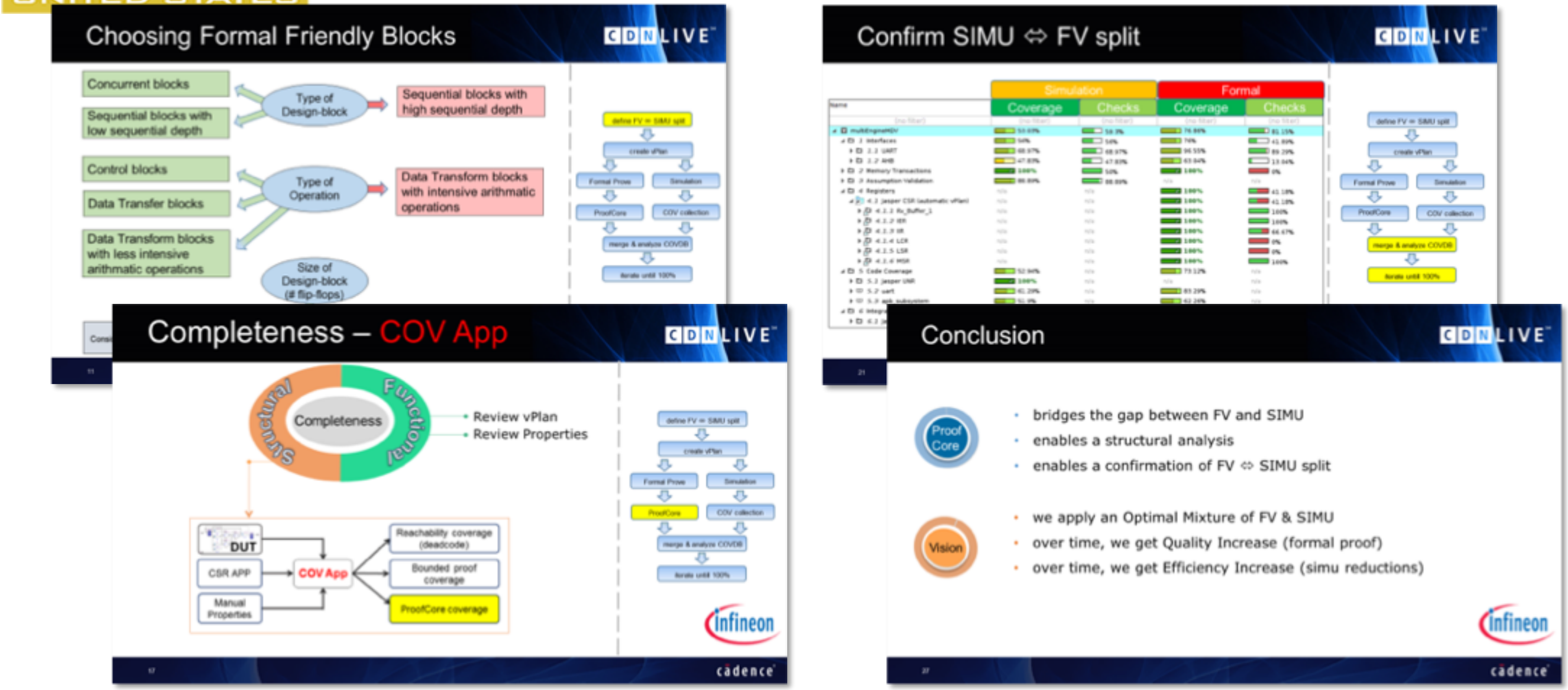
CSR

UNR

Simulation

Bug Hunting

Case Study: Infineon



Choosing Formal Friendly Blocks

CDNLIVE
Concurrent blocks
Sequential blocks with low sequential depth
Type of Design-block
Sequential blocks with high sequential depth
Control blocks
Data Transfer blocks
Data Transform blocks with less intensive arithmatic operations
Type of Operation
Data Transform blocks with intensive arithmatic operations
Size of Design-block (# flip-flops)
define FV ⇔ SIMU split
create vPlan
Formal Prove
Simulation
ProofCore
COV collection
merge & analyze COVDB
iterate until 100%
11

Confirm SIMU ⇔ FV split

	Simulation		Formal	
Name	Coverage	Checks	Coverage	Checks
multiEngineMDV	53.03%	58.3%	76.86%	81.15%
1 Interfaces	54%	54%	74%	41.89%
1.1 UART	68.97%	68.97%	96.55%	89.29%
1.2 AHB	47.83%	47.83%	63.04%	13.04%
2 Memory Transactions	100%	50%	100%	0%
3 Assumption Validation	88.89%	88.89%	n/a	n/a
4 Registers	n/a	n/a	100%	41.18%
4.1 Jasper CSR (automatic vPlan)	n/a	n/a	100%	41.18%
4.1.1 Rx_Buffer_1	n/a	n/a	100%	100%
4.1.2 IER	n/a	n/a	100%	100%
4.1.3 IIR	n/a	n/a	100%	66.67%
4.1.4 LCR	n/a	n/a	100%	0%
4.1.5 LSR	n/a	n/a	100%	0%
4.1.6 MSR	n/a	n/a	100%	100%
5 Code Coverage	52.94%	n/a	73.12%	n/a
5.1 Jasper UNR	100%	n/a	n/a	n/a
5.2 uart	41.29%	n/a	83.29%	n/a
5.3 apb_subsystem	51.9%	n/a	62.24%	n/a
6 Integra				
6.1 Ja				

Completeness – COV App

Conclusion

- bridges the gap between FV and SIMU
- enables a structural analysis
- enables a confirmation of FV ⇔ SIMU split

- we apply an Optimal Mixture of FV & SIMU
- over time, we get Quality Increase (formal proof)
- over time, we get Efficiency Increase (simu reductions)

Source: Infineon presentation at CDNLive 2017

Agenda

- The Need for Speed
- Formal methods to avoid sim cycles
- Coding for max sim speed
- Speeding power + mixed-signal SoC
- Break
- Portable Stimulus for faster verification
- Applying hardware to speed system verification
- Summary and call to action

Coding for Max Simulation Speed

- General SystemVerilog Coding
- Coding for Multi-Core Simulation
- UVM Save / Restart Methodology

General SystemVerilog Coding

- SystemVerilog is BIG (800+ pages)
  - Lots of opportunity to improve performance
- Focus today on a few of high-level concepts for making environments faster
  - Caching data (results, objects, etc.)
  - Focus on efficient algorithms
  - Choosing correct data-structures
- What is not being discussed today
  - Basic code optimization
  - Assertion / Functional coverage coding
  - Efficient randomization / constraint creation
  - Comprehensive coding guidelines (including these topics and more) available at http://support.cadence.com/ -- “Simulation Performance Coding Guidelines for SystemVerilog”

Caching Data

- Examples of when caching is effective
  - The same inputs always produce the output
    - When the calculation is done often (e.g. every cycle)
    - The same inputs will often be repeated over the short term
    - The calculation is expensive with respect to other things at the same time
  - A class object can be reused
    - When consumers of an object will take only what they want (won't keep a reference)
    - When the object is heavy to create
    - When the object has complex constraints (reuse of constraint construction)

Example Algorithm Caching

```
function int unsigned hash(string key);
  hash = 0;
  for(int i=0; i<key.len(); ++i) begin
    hash += key[i];
    hash += (hash<<10);
    hash ^= (hash>>6);
  end
  hash += (hash<<3);
  hash += (hash>>11);
  hash += (hash<<15);
endtask
```

What’s the problem?

Example Algorithm Caching

```
function int unsigned hash(string key);
  hash = 0;
  for(int i=0; i<key.len(); ++i) begin
    hash += key[i];
    hash += (hash<<10);
    hash ^= (hash>>6);
  end
  hash += (hash<<3);
  hash += (hash>>11);
  hash += (hash<<15);
endtask
```

Nothing (it is simple and fast), but, it is a linear algorithm so there is a possibility for improvement

Example Algorithm Caching

```
function int unsigned hash(string key);
  static int unsigned cache[string];
  //if you need to manage the cache size. Use a static
  //array as will be the fastest.
  static string aged_list[MAXSIZE];
  static int oldest = 0;
  // This is the savings if the same key gets used alot
  if(cache.exists(key)) return cache[key];

  ... //normal cache algorithm

  // This is the cache overhead.
  cache[key] = hash;
  cache.delete(aged_lsit[oldest%MAXSIZE]);
  aged_list[oldest%MAXSIZE] = key;
  ++oldest;
endfunction
```

A simple cache may make things faster

Example Caching an Object

```
task mycomp::run(uvm_phase_object phase);
  forever begin
    @(posedge vif.clk);
    if(txstart) begin
      local_data = mydata::create(“data”,this);
      data.randomize();
      send_recv_data(data); //some time consuming work
      $cast(shared_data, local_data.clone()); //copy it
      txport.write(shared_data); //send it on
    end
  end
endtask
```

What’s the problem?

Example Caching an Object

```
task mycomp::run(uvm_phase_object phase);
  forever begin
    @(posedge vif.clk);
    if(txstart) begin
      local_data = mydata::create("data",this);
      data.randomize();
      send_recv_data(data); //some time consuming work
      $cast(shared_data, local_data.clone()); //copy it
      txport.write(shared_data); //send it on
    end
  end
endtask
```

Data is created every time through the loop, but only used locally.

Example Caching an Object

```
task mycomp::run(uvm_phase_object phase);
  local_data = mydata::create("data",this);
  forever begin
    @(posedge vif.clk);
    if(txstart) begin
      data.randomize();
      send_recv_data(data); //some time consuming work
      $cast(shared_data, local_data.clone()); //copy it
      txport.write(shared_data); //send it on
    end
  end
endtask
```

Move data creation to only happen once.

Efficient Algorithms

- Know the complexity of your algorithm
  - Constant (O(1)), logarithmic (O(logn)), linear (O(n)), quadratic (O(n2)) ...
  - Watch out for loops in loops
    - A loop is O(n)
    - A loop inside a loop is O(n2)
    - A loop inside a loop inside a loop is O(n3) ...
- Watch out how often you are doing work
  - A linear algorithm executed every cycle will likely be problematic
- Watch what you do when operating on larger data sets (higher values of n)
  - Can algorithm be changed to be constant or logarithmic?
  - Can executions of the algorithm be minimized?

Example of a Problematic Algorithm

```
input real vin;
output real vout;
real vdata[512];
logic[8:0] ptr;
always@(posedge clk)
  ptr<=ptr+1;
real sum;
always@(posedge clk) begin
  vdata[ptr] <= vin;
  sum=0.0; foreach(vdata[i]) sum+=vdata[i];
  vout <= sum/512;
end
```

What’s the problem?

Example of a Problematic Algorithm

```
input real vin;
output real vout;
real vdata[512];
logic[8:0] ptr;
always@(posedge clk)
  ptr<=ptr+1;
real sum;
always@(posedge clk) begin
  vdata[ptr] <= vin;
  sum=0.0; foreach(vdata[i]) sum+=vdata[i];
  vout <= sum/512;
end
```

Every edge we sum the array even though only one element changes

Example of a Problematic Algorithm

```
input real vin;
output real vout;
real vdata[512];
logic[8:0] ptr;
real curr=0.0;
always@(posedge clk)
  ptr<=ptr+1;
always@(posedge clk) begin
  vdata[ptr] <= vin;
  vout <= curr/512;
  curr <= curr-vdata[ptr]+vin;
end
```

Better to only do what is required each cycle

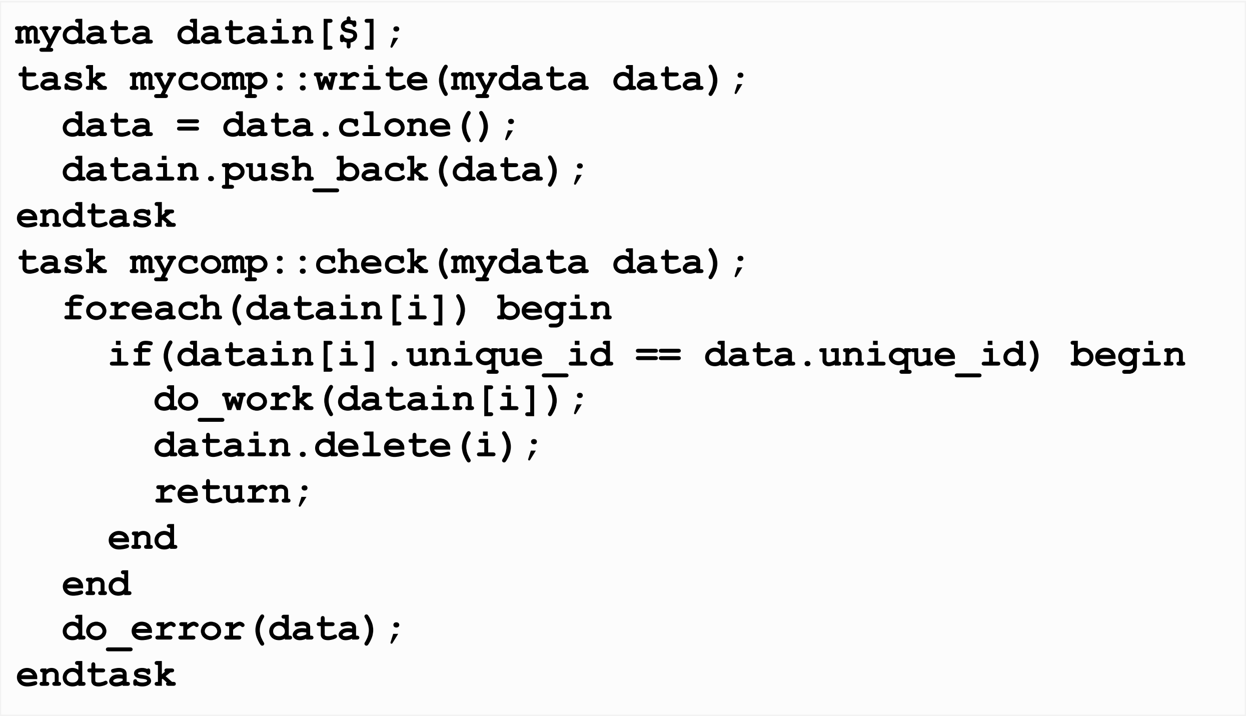
Choosing the Best Data Structure

- This is related to memory management and algorithms
- Memory management
  - Dynamic data structures (dynamic arrays, queues, associative arrays, classes) have heap management overhead.
  - Static arrays and structs are pass by value (no heap management)
  - This overhead can be significant depending on how an object is used
- Basic QDA Algorithms
  - Search
    - Associative arrays are O(logn)
    - Everything else is O(n)
  - Front/back insertion
    - Associative arrays are O(logn)
    - Queues are O(1)
    - Static and dynamic arrays are O(n) (must be done manually)
    - Queues auto-size when needed
  - Random insertion
    - Associative arrays are O(logn)
    - Everything else is O(n)

Choosing the Best Data Structure

- General recommendations
  - Use associative arrays when searches dominate
  - Use queues for most dynamically sizeable random access objects
  - Use static arrays anytime it is reasonable
  - Use structs instead of classes for tuples (or simple metadata)

Data Structure Example



```
mydata datain[$];
task mycomp::write(mydata data);
  data = data.clone();
  datain.push_back(data);
endtask
task mycomp::check(mydata data);
  foreach(datain[i]) begin
    if(datain[i].unique_id == data.unique_id) begin
      do_work(datain[i]);
      datain.delete(i);
      return;
    end
  end
  do_error(data);
endtask
```

What is the problem with this?

Data Structure Example

We are using the wrong data structure.
Queues are not good with random deletion and lookup!

```
mydata datain[$];
task mycomp::write(mydata data);
  data = data.clone();
  datain.push_back(data);
endtask
task mycomp::check(mydata data);
  foreach(datain[i]) begin
    if(datain[i].unique_id == data.unique_id) begin
      do_work(datain[i]);
      datain.delete(i);
      return;
    end
  end
  do_error(data);
endtask
```

Access is constant, deletion is linear

Data Structure Example

Use an associative array instead
Lookup and deletion are
O(log(n)) instead of O(n)!

```
mydata datain[int];
task mycomp::write(mydata data);
  data = data.clone();
  datain[data.unique_id] = data;
endtask
task mycomp::check(mydata data);
  if( datain.exists(data.unique_id) ) begin
      do_work(datain[data.unique_id]);
      datain.delete(data.unique_id);
      return;
  end
  do_error(data);
endtask
```

Lookup and deletion are logN
with # of elements

Another Data Structure Example

```
class data;
  int aval;
  int bval;
  int extra;
endclass
data sparse_memory[int];

function add_elem(int addr, data d);
  sparse_memory[addr] = d;
endfunction
function data get_elem(int addr);
  data rval;
  if(sparse_memory.exists(addr))
    rval sparse_memory[addr];
  else
    rval = new;
  return rval;
endfunction
```

What is the problem with this?

Another data Structure Example

```
class data;
  int aval;
  int bval;
  int extra;
endclass
data sparse_memory[int];

function add_elem(int addr, data d);
  sparse_memory[addr] = d;
endfunction
function data get_elem(int addr);
  data rval;
  if(sparse_memory.exists(addr))
    rval sparse_memory[addr];
  else begin
    rval = new;
    sparse_memory[addr] = rval;
  end
  return rval;
endfunction
```

There is no need for a class (no polymorphism or any class behaviors)

Another Data Structure Example

```
typedef struct packed{
  int aval;
  int bval;
  int extra;
} data;

data sparse_memory[int];

function add_elem(int addr, data d);
  sparse_memory[addr] = d;
endfunction
function data get_elem(int addr);
  return sparse_memory[addr];
endfunction
```

Change to a struct

Coding for Max Simulation Speed

- General SystemVerilog Coding
- Coding for Multi-Core Simulation
- UVM Save / Restart Methodology

Coding for Multi-core Simulation

- Multi-core simulation is similar to hardware acceleration except
  - Uses standard servers
  - Achieves acceleration by sending concurrent work to separate cores
  - Some applications (such as wave dumping) also lend themselves to running in separate cores
- The same coding that works for acceleration works for multi-core
  - Synthesizable code
- General guidelines
  - Signal level activity should be in synthesizable bfms
  - Reduce activity between accelerated and non-accelerated sections maximizes speed up
  - Synchronized designs speed up the best but are not required

Coding for Multi-core Simulation

```
module somemod1 (input clk, ...)
  always@(posedge clk)
    ...//complex expressions and assignments
  always@(posedge clk)
    ...//more complex expressions and assignments
  always_comb
    ...//best is to not have any timing
  assign ... //best is to not have any timing
endmodule
```

```
module connector(input clk, ...)
  clkgater g1(gclk,clk,cenable);
  somemod1(gclk, ...);
  othermod(clk, ...);
  ...
endmodule
```

Will attempt to associated processes with clock or gated version of clock

- What multi-core wants is
  - Lots of independent processes active at the same time

Coding for Multi-core Simulation

```
module behav (myinterface mif, ...)
  import vepkg::*;
  //some rtl stuff
  //some ve stuff
endmodule
```

```
module timedblock(...)
  assign #1.1 w1 = ...
  assign #0.3 w2 = ...
  assign #2.6 w3 = ...
  ...
endmodule
```

All happen in different time slots so may not be able to be in parallel if there are interdependency

- What multi-core doesn’t want is
  - Behavioral code (things it can’t synthesize)
  - Lots of independent timing (very few events at a given time slot)

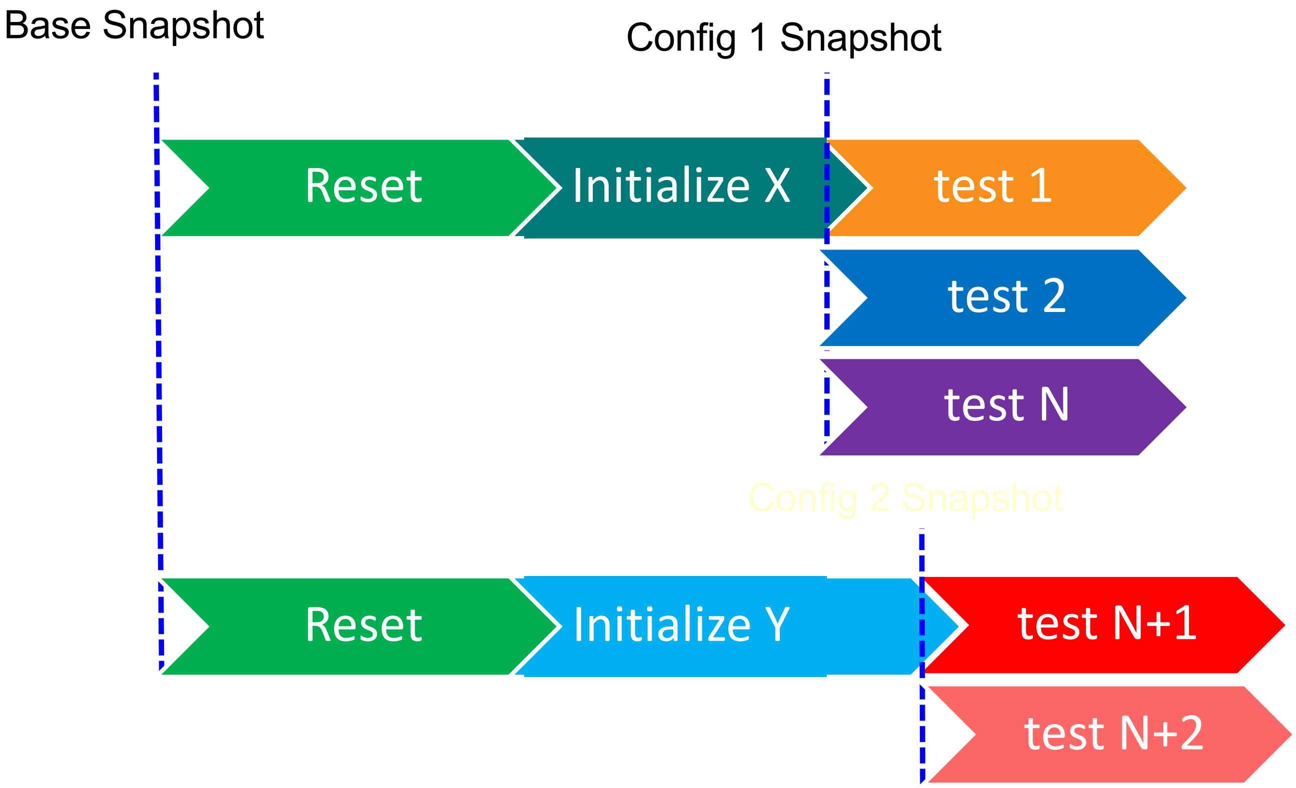
Coding for Max Simulation Speed

- General SystemVerilog Coding
- Coding for Multi-Core Simulation
- UVM Save / Restart Methodology

UVM Save/Restart Concept

- Test sets tend to do the same initialization work prior to doing test specific work
- Device setup may take as much as 80% of the simulation time
- Treat the device setup as an extension of the build
  - Build a base simulation snapshot
  - Run the simulation to time N (when device setup is complete)
  - Save the simulation snapshot at time N
  - Run the test set using the saved simulation snapshot
  - Make use of reseeding to run the same tests with different seeds

UVM Save/Restart Concept



Mechanics

```
class base_restart_test extends uvm_test;
  ...
  task run(uvm_phase_object phase);
    init_seq init_seq = init_seq::type_id::create(“init_seq”,null);
    test_seq seq;
    init_seq.start(null, null);
    $save(init_seq.get_type_name());
    $value$plusags(“SEQUENCE=%s”,restart_seq_str);
    seq  = test_seq::type_id::create(restart_seq_str,null);
    void’(seq.randomize());
    seq.start(seqr, null, -1, 0);
  endtask
endtask
```

- Each configuration is a UVM test
  - +UVM_TESTNAME=config_1
- Tests are virtual sequences loaded from command line arg
  - +SEQUENCE=testseq

Questions?

Agenda

- The Need for Speed
- Formal methods to avoid sim cycles
- Coding for max sim speed
- Speeding power + mixed-signal SoC
- Break
- Portable Stimulus for faster verification
- Applying hardware to speed system verification
- Summary and call to action

Low Power Mixed Signal Simulation Tutorial

- Discuss verification of mixed signal SoC that is powered by an off-chip regulator driving on-chip supplies
  - Processor based design powered by on-chip power supplies
  - Verify SPICE, RNM, AMS, Verilog models in the same environment
  - All IP developed by Cadence
  - Power intent specified in UPF 2.0
- Low Power Mixed Signal simulation run in UVM
  - LDO (SPICE) driving UPF Power Supply Network
  - Isolation, state retention, power shutoff
- Target Technology – Cadence 45 nm – GSCLIB045

Low Power Basics

Concept	Description
Power Domains	Group the elements of logic hierarchy that share the same primary power supply
Supply Ports	Provide the supply interface to power domains and switches
Supply Nets	Connect supply ports
Power Switch	Based on the value of the power control signal, the Power Switch connects / disconnects the input supply port to the output supply port of the switch
HDL Supply Net Control Functions	UPF provides functions which enable the user to drive Supply Ports in low power simulation: supply_on, supply_off, supply_partial_on
Power Supply Network	Consists of supply ports, supply nets and power switches and their interconnections
LDO	Low-dropout regulator. DC/DC converter used for on-chip power supplies

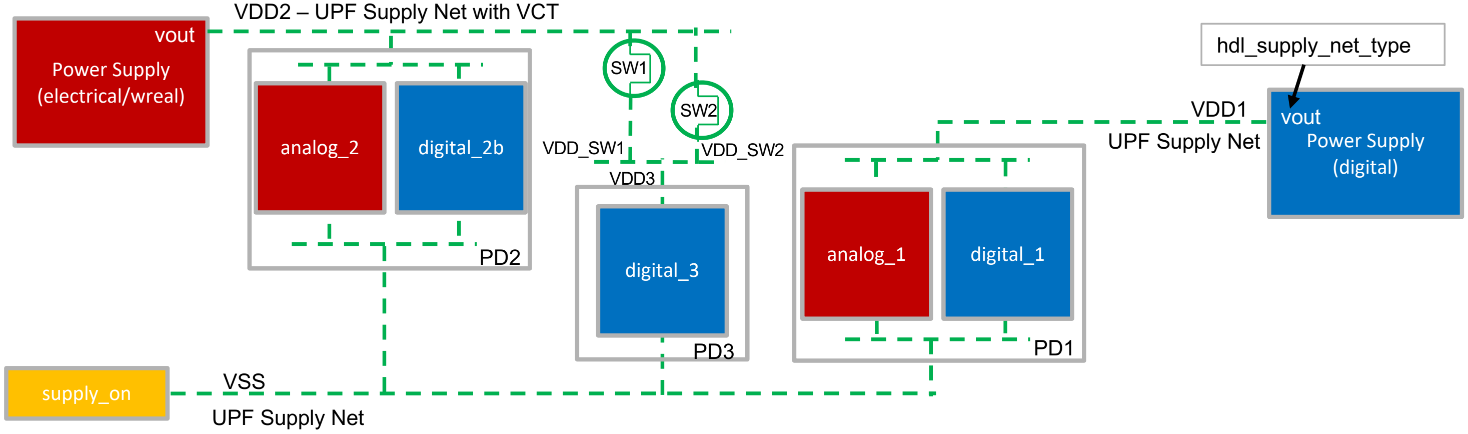
Low Power Basics

Concept	Description
State Retention	Allows the contents of registers to be saved prior to power shutoff and recovered when is power is restored Usually performed on key control registers
Isolation	Prevents corrupted values from propagating from shutoff power domains to power domains which are powered up
Power Shutoff (PSO)	Power reduction method where power domains are shutoff. Shutoff can be performed by Power Switch or by turning off the power to the supply ports. Isolation and State Retention are often used in Power Shutoff Domains

Power Digital Logic

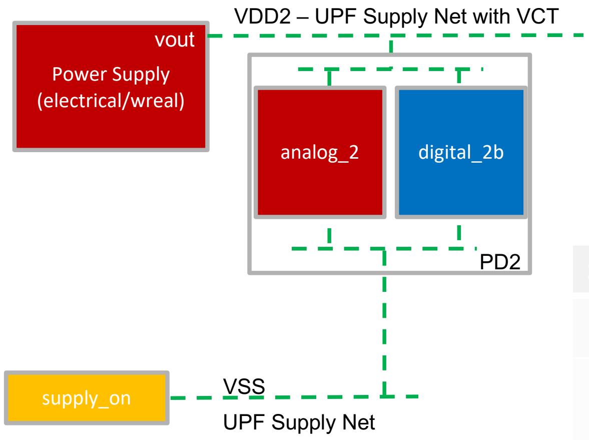
- Most Common
  - HDL Supply Net Control Functions (supply_on)
  - UPF Power Switches
- hdl_supply_net_type
  - Drive UPF supply nets from HDL models
  - UPF Package
- electrical / wreal

SW1, SW2 – UPF Power Switches
VDD_SW1 – UPF Supply Net
VDD_SW2 – UPF Supply Net
VDD3 – UPF Supply Net with Resolution Function



Driving PSN with electrical / wreal Ports

- UPF Supply Nets require a STATE and VOLTAGE
  - STATE – UNDETERMINED, PARTIAL_ON, FULL_ON, OFF
- wreal / electrical ports provide the VOLTAGE, but no STATE
- Add STATE through VCT (Value Conversion Table)

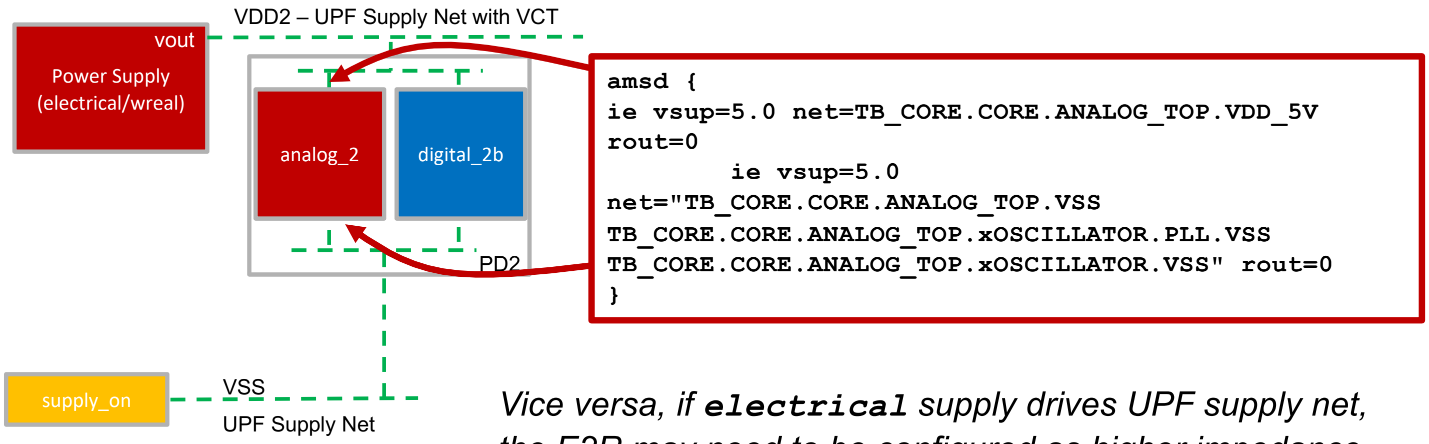

vout
Power Supply (electrical/wreal)
VDD2 – UPF Supply Net with VCT
analog_2
digital_2b
PD2
supply_on
VSS
UPF Supply Net

```
create_supply_net VDD2
create_hdl2upf_vct VCTwr2upf_VDD2 \
-hdl_type {sv cds_rnm} \
-table {{>=4.8 FULL_ON} \
        {>=4.5 PARTIAL_ON} \
        {<4.5 OFF}}
```

STATE	Digital Logic
FULL_ON	Does not cause corruption
PARTIAL_ON	Enable / Disable corruption through UPF command
OFF	Corrupt
UNDETERMINED	Corrupt

Driving PSN with Electrical / wreal Ports

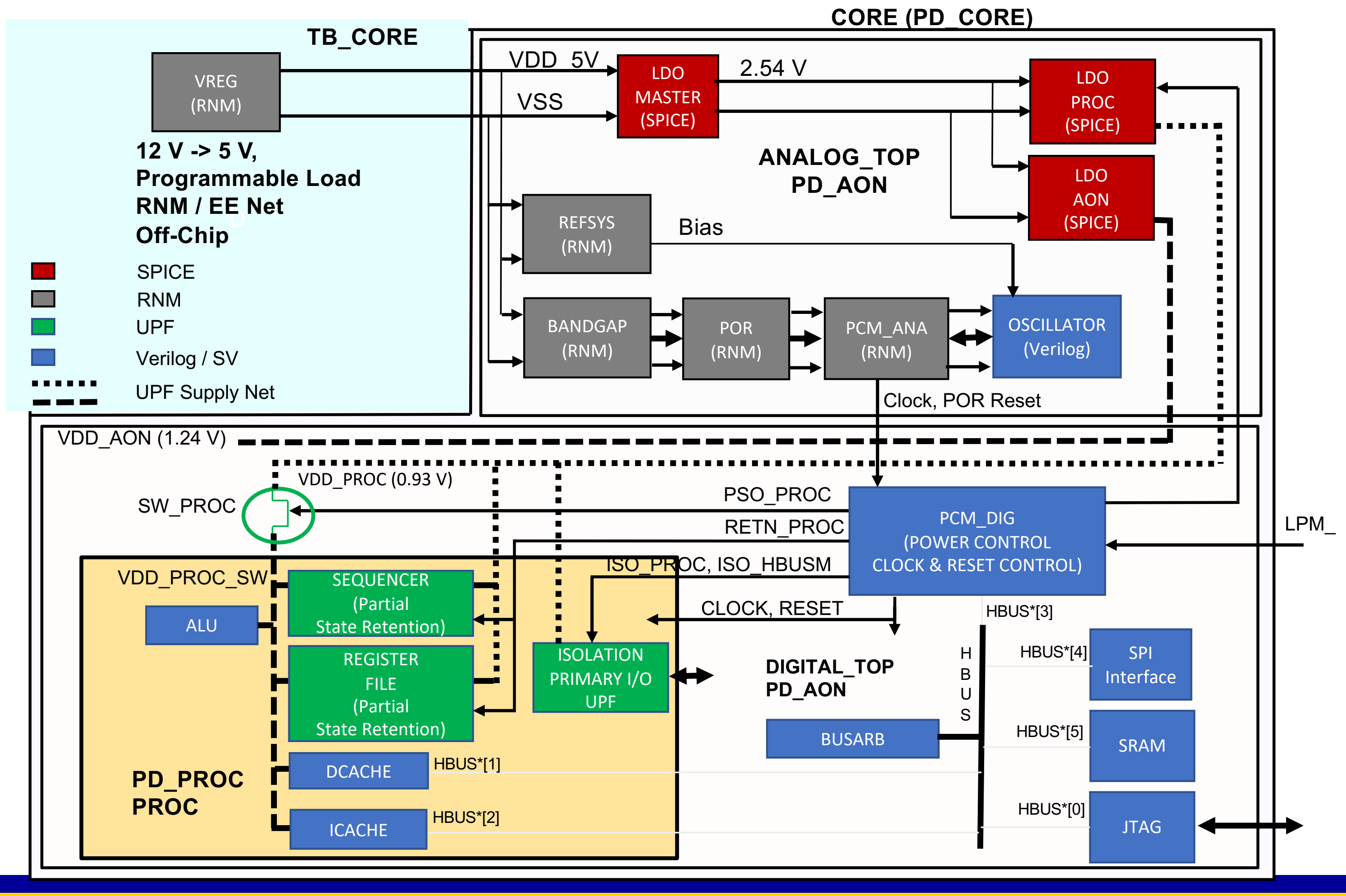
- The VOLTAGE from the UPF supply net is connected to the electrical signal by an internal R2E connect module
- The impedance of the R2E connect module is critical for analog block simulation



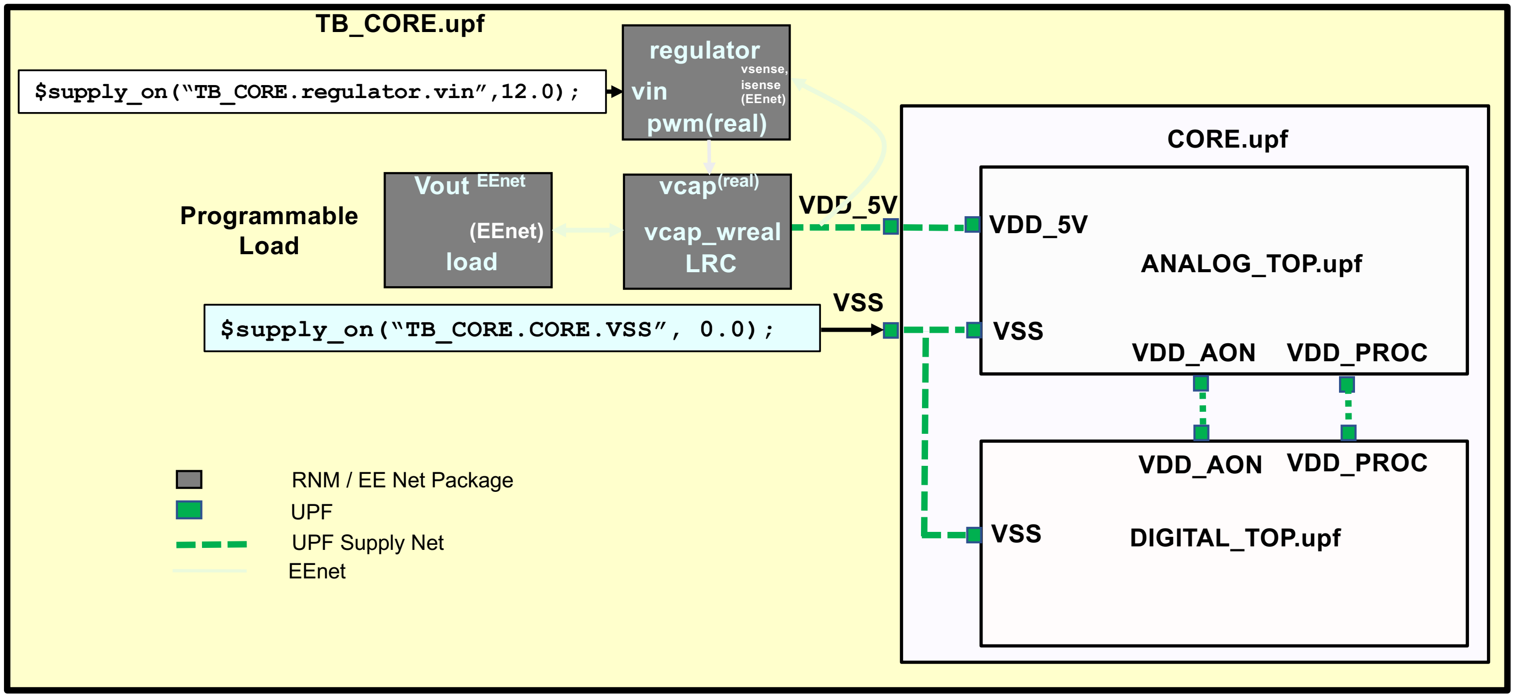
```
amsd {
ie vsup=5.0 net=TB_CORE.CORE.ANALOG_TOP.VDD_5V
rout=0
        ie vsup=5.0
net="TB_CORE.CORE.ANALOG_TOP.VSS
TB_CORE.CORE.ANALOG_TOP.xOSCILLATOR.PLL.VSS
TB_CORE.CORE.ANALOG_TOP.xOSCILLATOR.VSS" rout=0
}
```

Vice versa, if ***`electrical`*** *supply drives UPF supply net, the E2R may need to be configured as higher impedance than default (200 Ohms) especially for non-ideal supply source*

Block Diagram & Power Architecture



Hierarchical UPF & Power / Ground Connections



```
$supply_on(“TB_CORE.regulator.vin”,12.0);
```

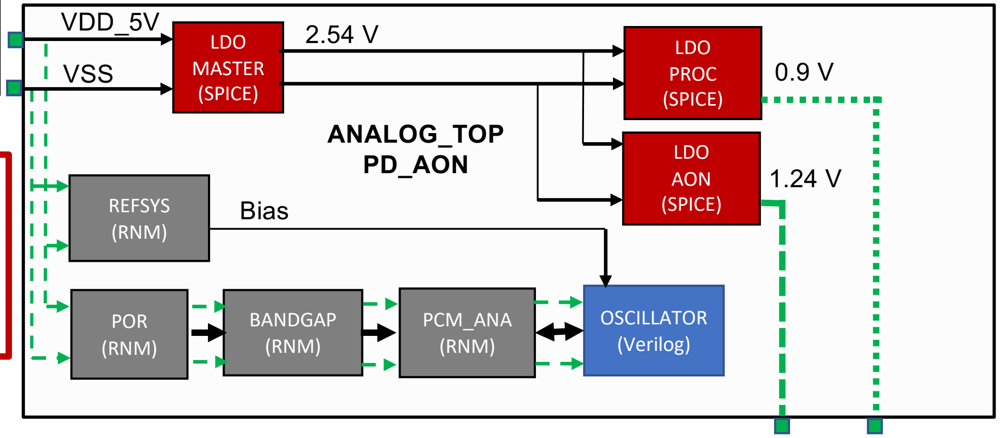
```
$supply_on(“TB_CORE.CORE.VSS”, 0.0);
```

ANALOG_TOP (PARTIAL UPF)

```
connect_supply_net VDD_5V
          -ports {VDD_5V}
connect_supply_net VSS
          -ports {VSS}
```

Adds STATE to UPF Supply Net

```
create_hdl2upf_vct
VCTwr2upf_VDD_5V \
-hdl_type {sv cds_rnm} \
-table {{>=4.8 FULL_ON} \
        {>=4.5 PARTIAL_ON} \
        {<4.5 OFF}}
```



```
connect_supply_net VDD_AON
         -ports {VDD_AON}
connect_supply_net VDD_PROC
         -ports {VDD_PROC}
```

🟥 SPICE

▨ RNM

🟩 UPF

🟦 Verilog / SV

┅┅ UPF Supply Net

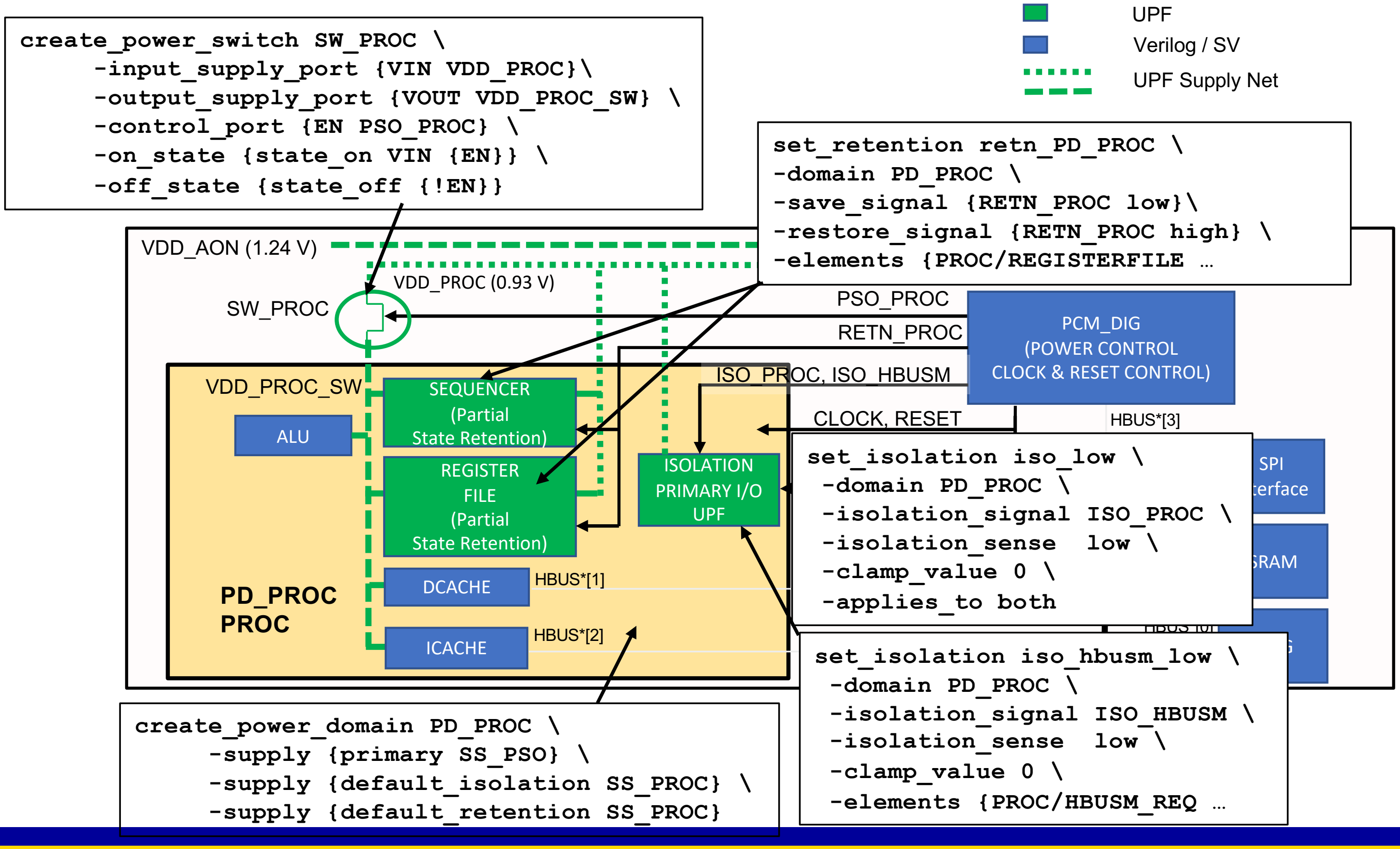
Adds STATE to UPF Supply Net

```
create_hdl2upf_vct
VCTwr2upf_VDD_AON \
-hdl_type {sv cds_rnm} \
-table {{>=1.2 FULL_ON} \
        {>=1.1 PARTIAL_ON} \
        {<1.1 OFF}}
```

```
create_hdl2upf_vct
VCTwr2upf_VDD_PROC \
-hdl_type {sv cds_rnm} \
-table {{>=0.9 FULL_ON} \
        {>=0.7 PARTIAL_ON} \
        {<0.7 OFF}}
```

DIGITAL_TOP UPF (PARTIAL UPF)

🟥 SPICE
▨ RNM
🟩 UPF
🟦 Verilog / SV
┅ UPF Supply Net



```
create_power_switch SW_PROC \
     -input_supply_port {VIN VDD_PROC}\
     -output_supply_port {VOUT VDD_PROC_SW} \
     -control_port {EN PSO_PROC} \
     -on_state {state_on VIN {EN}} \
     -off_state {state_off {!EN}}
```

```
set_retention retn_PD_PROC \
-domain PD_PROC \
-save_signal {RETN_PROC low}\
-restore_signal {RETN_PROC high} \
-elements {PROC/REGISTERFILE …
```

```
set_isolation iso_low \
 -domain PD_PROC \
 -isolation_signal ISO_PROC \
 -isolation_sense  low \
 -clamp_value 0 \
 -applies_to both
```

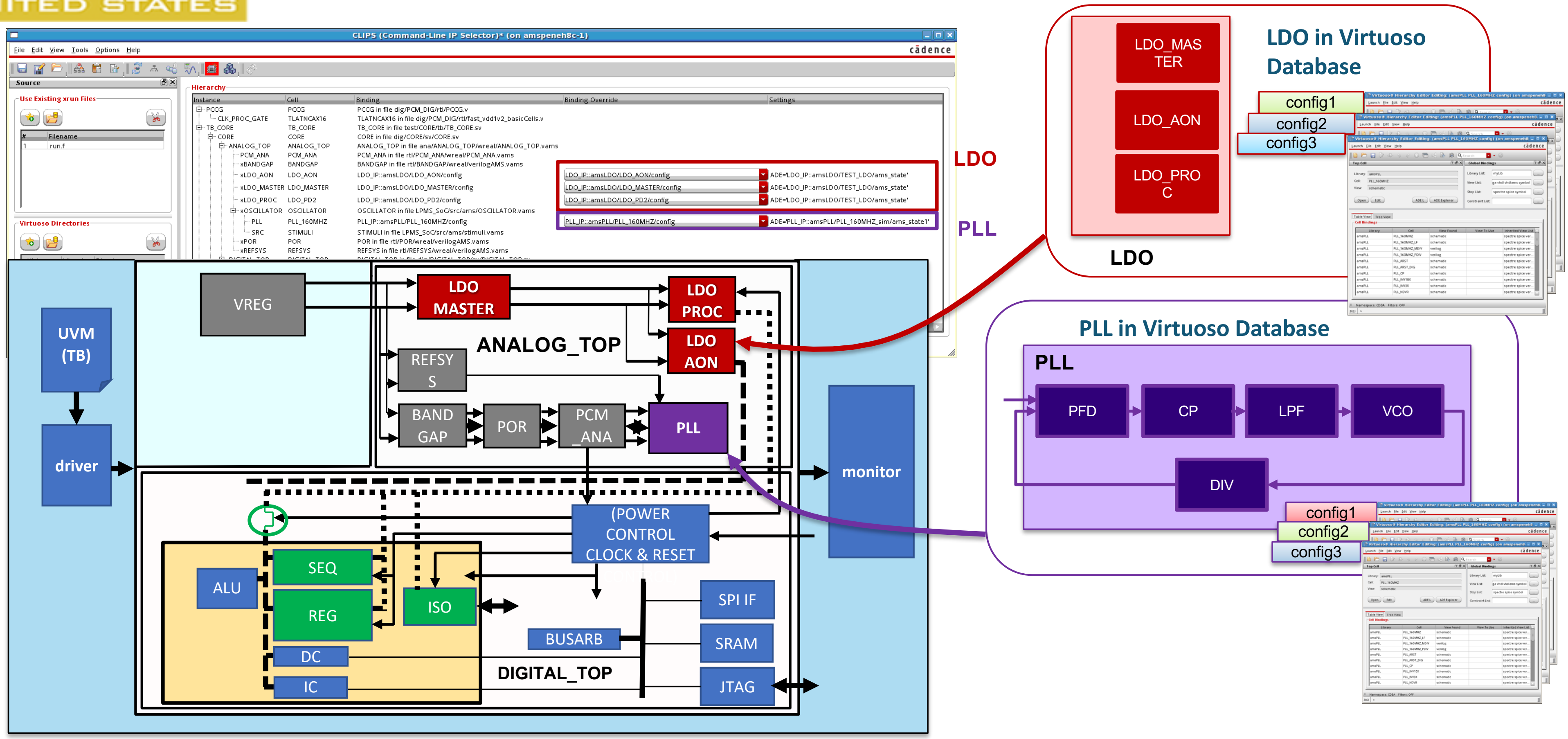
```
set_isolation iso_hbusm_low \
 -domain PD_PROC \
 -isolation_signal ISO_HBUSM \
 -isolation_sense  low \
 -clamp_value 0 \
 -elements {PROC/HBUSM_REQ …
```

```
create_power_domain PD_PROC \
     -supply {primary SS_PSO} \
     -supply {default_isolation SS_PROC} \
     -supply {default_retention SS_PROC}
```

Power States

Power State	Description
Power Up	LDO AON powers up, Power On Reset, Clock Enabled, JTAG, BUSARB, PCM_DIG on
Load SRAM	JTAG loads PROC object code LDO_PROC powers up PROC executes instruction thread
Power Down PROC	LPM_ asserted. Cache flush started, clock gated, state saved, isolation enabled. Power Shutoff by Power Switch SW_PROC
Power Up PROC	LPM_ released. PROC power on – SW_PROC turned on. Restore state, release state.
LDO Shutdown	Output of VREG is heavily loaded, causing LDO_AON and LDO_PROC to be shutoff. Load is removed, enter Power Up State. After POR, enter into LOAD SRAM state After LOAD SRAM – PROC executes instruction thread

Switch in AMS Design Configuration



CLIPS (Command-Line IP Selector)* (on amspeneh8c-1)

Hierarchy

Instance	Cell	Binding	Binding Override	Settings
PCCG	PCCG	PCCG in file dig/PCM_DIG/rtl/PCCG.v		
CLK_PROC_GATE	TLATNCAX16	TLATNCAX16 in file dig/PCM_DIG/rtl/fast_vdd1v2_basicCells.v		
TB_CORE	TB_CORE	TB_CORE in file test/CORE/tb/TB_CORE.sv		
CORE	CORE	CORE in file dig/CORE/sv/CORE.sv		
ANALOG_TOP	ANALOG_TOP	ANALOG_TOP in file ana/ANALOG_TOP/wreal/ANALOG_TOP.vams		
PCM_ANA	PCM_ANA	PCM_ANA in file rtl/PCM_ANA/wreal/PCM_ANA.vams		
xBANDGAP	BANDGAP	BANDGAP in file rtl/BANDGAP/wreal/verilogAMS.vams		
xLDO_AON	LDO_AON	LDO_IP::amsLDO/LDO_AON/config	LDO_IP::amsLDO/LDO_AON/config	ADE='LDO_IP::amsLDO/TEST_LDO/ams_state'
xLDO_MASTER	LDO_MASTER	LDO_IP::amsLDO/LDO_MASTER/config	LDO_IP::amsLDO/LDO_MASTER/config	ADE='LDO_IP::amsLDO/TEST_LDO/ams_state'
xLDO_PROC	LDO_PD2	LDO_IP::amsLDO/LDO_PD2/config	LDO_IP::amsLDO/LDO_PD2/config	ADE='LDO_IP::amsLDO/TEST_LDO/ams_state'
xOSCILLATOR	OSCILLATOR	OSCILLATOR in file LPMS_SoC/src/ams/OSCILLATOR.vams		
PLL	PLL_160MHZ	PLL_IP::amsPLL/PLL_160MHZ/config	PLL_IP::amsPLL/PLL_160MHZ/config	ADE='PLL_IP::amsPLL/PLL_160MHZ_sim/ams_state1'
SRC	STIMULI	STIMULI in file LPMS_SoC/src/ams/stimuli.vams		
xPOR	POR	POR in file rtl/POR/wreal/verilogAMS.vams		
xREFSYS	REFSYS	REFSYS in file rtl/REFSYS/wreal/verilogAMS.vams		

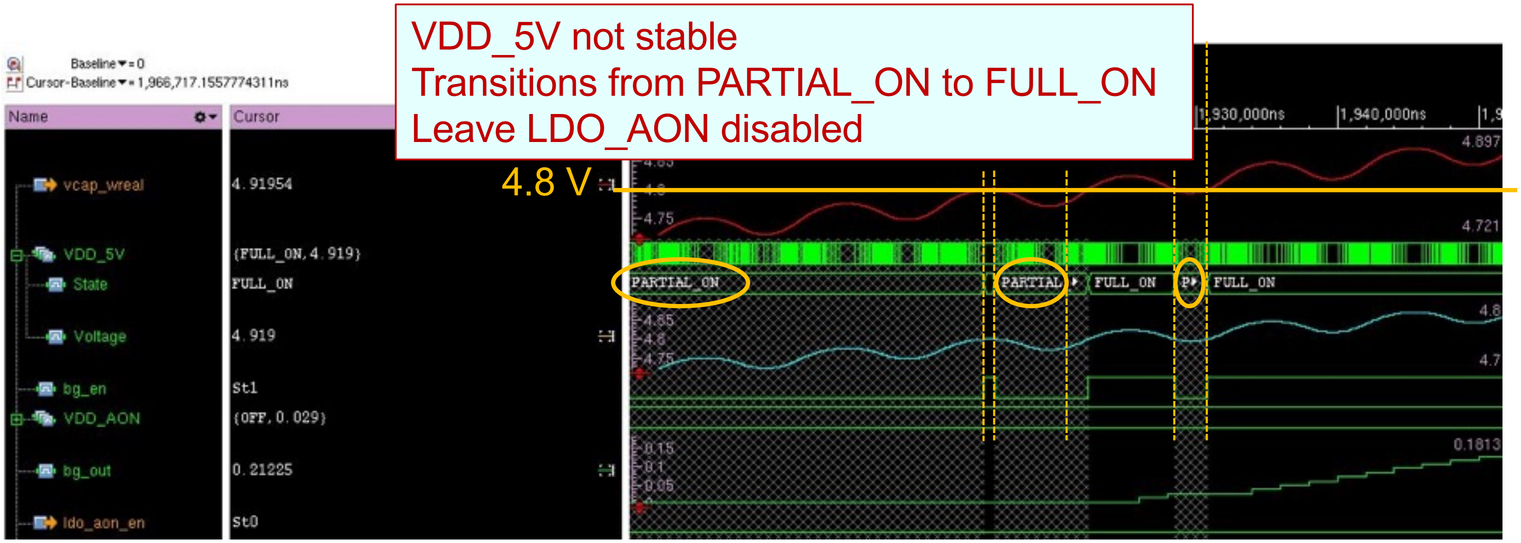
Source

Use Existing xrun Files

#	Filename
1	run.f

Virtuoso Directories

VDD_5V TRANSITIONS TO FULL_ON



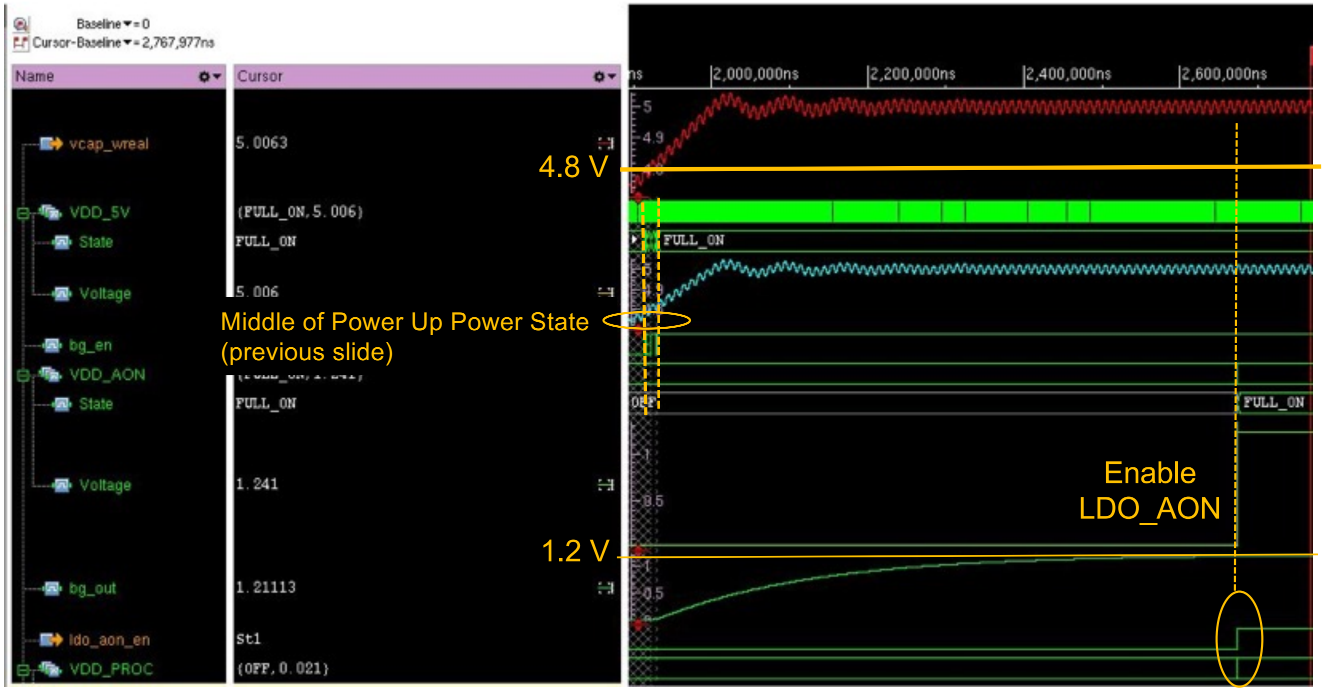
Baseline ▾ = 0
Cursor-Baseline ▾ = 1,966,717.1557774311ns

Name	Cursor
vcap_wreal	4.91954
VDD_5V	{FULL_ON, 4.919}
State	FULL_ON
Voltage	4.919
bg_en	St1
VDD_AON	{OFF, 0.029}
bg_out	0.21225
ldo_aon_en	St0

VDD_5V not stable
Transitions from PARTIAL_ON to FULL_ON
Leave LDO_AON disabled

Power Up Power State	Description
Beginning	VDD_5V ramps for 0V (OFF) to < 4.8 V (PARTIAL_ON) (PARTIAL_ON -> OFF). Waveform not shown
Middle	VDD_5V has multiple transitions between FULL_ON and PARTIAL_ON Disable LDO_AON until VDD_5V is FULL_ON and BANDGAP output >= 1.2 V

VDD_5V TRANSITIONS TO FULL_ON



Baseline ▾ = 0
Cursor-Baseline ▾ = 2,767,977ns

Name	Cursor
vcap_wreal	5.0063
VDD_5V	(FULL_ON,5.006)
State	FULL_ON
Voltage	5.006
bg_en	
VDD_AON	(FULL_ON,1.241)
State	FULL_ON
Voltage	1.241
bg_out	1.21113
ldo_aon_en	St1
VDD_PROC	(OFF,0.021)

4.8 V

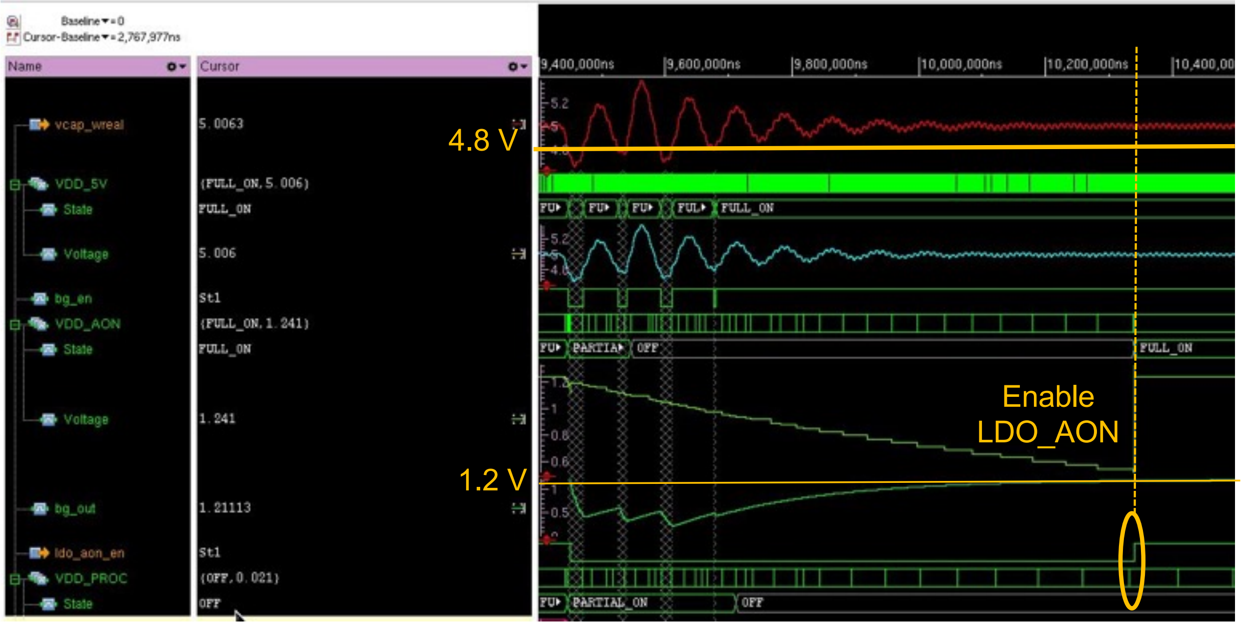
Middle of Power Up Power State (previous slide)

Enable LDO_AON

1.2 V

Power Up Power State	Description
End	VDD_5V FULL_ON and bg_out >= 1.2 V LDO_AON is powered up, enter LOAD SRAM Power State

LDO SHUTDOWN POWER STATE



Baseline ▾ = 0
Cursor-Baseline ▾ = 2,767,977ns

Name	Cursor
vcap_wreal	5.0063
VDD_5V	{FULL_ON, 5.006}
State	FULL_ON
Voltage	5.006
bg_en	St1
VDD_AON	{FULL_ON, 1.241}
State	FULL_ON
Voltage	1.241
bg_out	1.21113
ldo_aon_en	St1
VDD_PROC	{OFF, 0.021}
State	OFF

4.8 V

1.2 V

Enable LDO_AON

LDO Shutdown Power State	Description
Entire Range	VDD_5V transitions between FULL_ON and PARTIAL_ON. LDO_AON gets disabled LDO_AON is enabled after VDD_5V is FULL_ON and bg_out > 1.2 V

Summary 1 of 2

- Low Power MS simulation allows user to verify the operation of on-chip power supplies, clock generation, reset, and digital logic concurrently

Issues Found	Description
PSN Errors	Debugged using SimVision Power Supply Network. Usually found at start of LP verification cycle
Incorrect parameter setting on POR cell	VREF_LDO parameter was initially set to 2.4 v instead of 4.8 v. Result – LDO_AON turned on too soon produced wrong output voltage. Digital logic did not function.
Incorrect registers for state retention	PROC / UPF developers worked together to determine correct registers for state retention strategy
Staggering isolation enable signals	PROC / UPF developers decided to disable HBUS access (through isolation) until PROC was running. Prevent accidental HBUS traffic

Summary 2 of 2

- Cache Flush
  - Contents of Data Cache had to be transferred to SRAM before power shutoff
  - Required changes to Instruction / Data Cache and PCM_DIG

- Based on accuracy / simulation performance requirements – swap models
  - Oscillator – replaced Verilog AMS model of PLL with Verilog model
  - LDO – replaced RNM model with SPICE model

- Recommendation - isolate handshaking signals (bus request, bus grant …) to their inactive state to prevent locking HBUS when PROC is shutoff

Agenda

- The Need for Speed
- Formal methods to avoid sim cycles
- Coding for max sim speed
- Speeding power + mixed-signal SoC
- Break
- Portable Stimulus for faster verification
- Applying hardware to speed system verification
- Summary and call to action

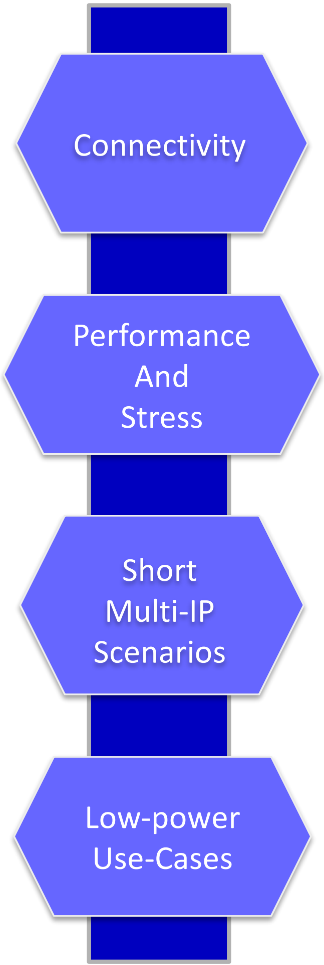
BREAK!

Agenda

- The Need for Speed
- Formal methods to avoid sim cycles
- Coding for max sim speed
- Speeding power + mixed-signal SoC
- Break
- Portable Stimulus for faster verification
- Applying hardware to speed system verification
- Summary and call to action

SoC HW Verification Next Level of Challenges For PSS

- Simulation speed
  - A UVM TB, and in specific randomization, cannot be expedite
  - Low ROI on multi-core simulation and emulation
  - SV re-elaboration is a concern
- Coverage closer requires lots of work
  - Virtual sequence creation is manual
  - Example: cannot ask a tool “try all possible traffic in all legal configurations modes”
  - Coverage holes requires reachability analysis
- UVM test creation requires expertise
  - UVM sequences introduce a learning curve and protocol VIPs comes with manuals
  - Debug contradictions or illegal tests analysis are time consuming
- Self-checking becomes a challenge
- Portability and reuse
  - Vertical reuse is a challenge
  - Cannot leverage the efforts in terms of registers sequences, tests and coverage



Connectivity

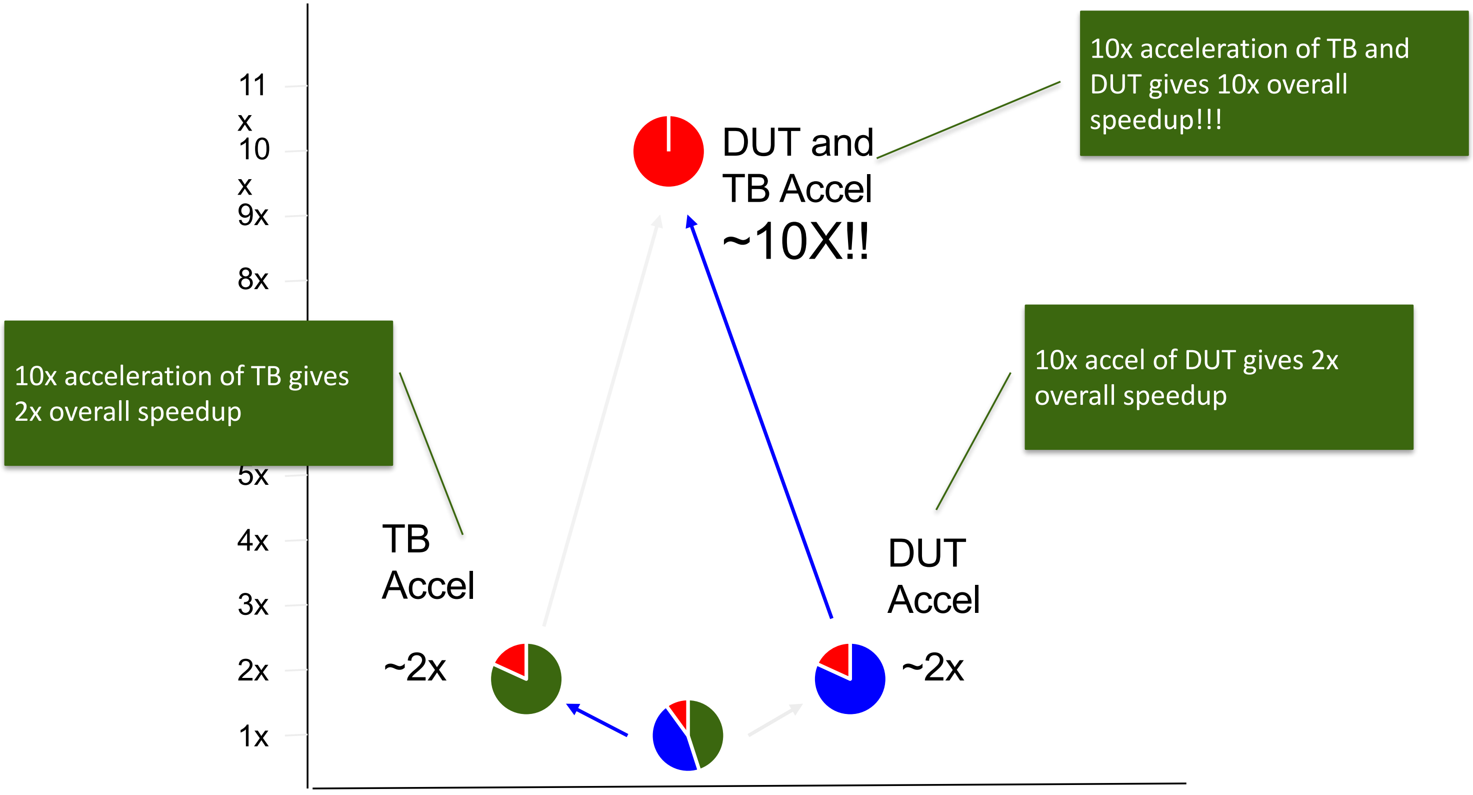
Performance And Stress

Short Multi-IP Scenarios

Low-power Use-Cases

Is it Important Speeding-up the TB?

Amdhal's Law - Amdahl's law is a formula used to find the maximum improvement improvement possible by improving a particular part of a system.



DUT and TB Accel ~10X!!

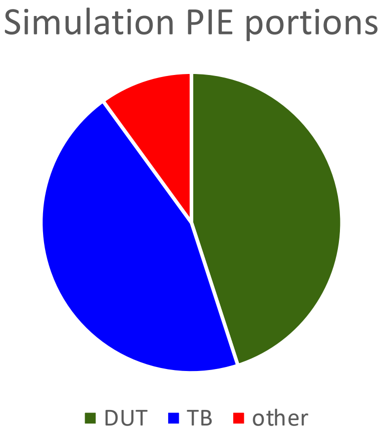
10x acceleration of TB and DUT gives 10x overall speedup!!!

10x acceleration of TB gives 2x overall speedup

TB Accel ~2x

10x accel of DUT gives 2x overall speedup

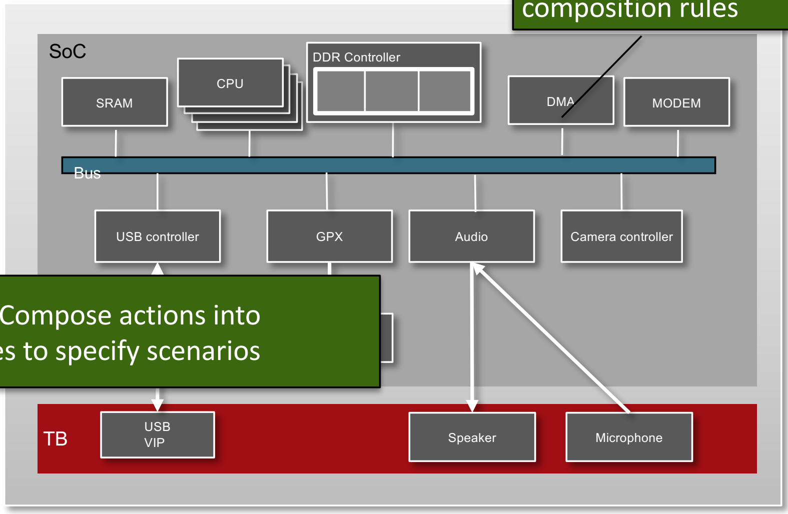
DUT Accel ~2x



How Can PSS Solution help?

The PSS Input Format and Modeling Intuition

- Create an abstract behavioral model to capture the legal scenario space
  - Automated self-checking test creation, coverage and debug
- Parsing the model allows leveraging it multiple ways:
  - Example #1: portability and reuse
  - Example #2: time and resource aware solving (virtual sequence)
  - Example #3: coverage reachability
  - Example #4: speed…
- Consider the challenge of randomizing 1M packets
  - Randomization consumes time
  - Do you really need 1M variations??

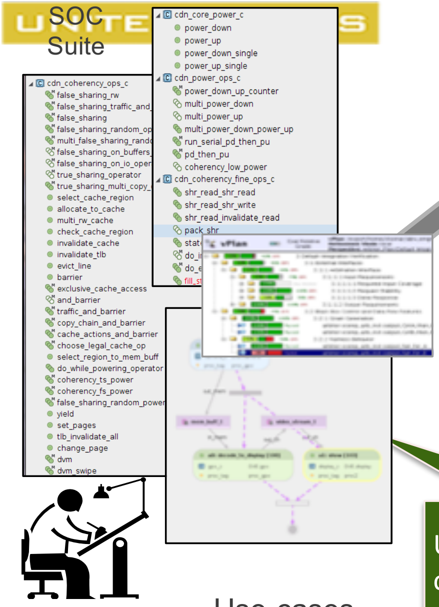


Step 1: Define component types and their operations as actions with composition rules

Step 2: Compose actions into activities to specify scenarios

Automated Hi-Speed C TestBench

Xcelium + AVIP/



SOC Suite

cdn_coherency_ops_c
- false_sharing_rw
- false_sharing_traffic_and_
- false_sharing
- false_sharing_random_op
- multi_false_sharing_rando
- false_sharing_on_buffers_
- false_sharing_on_io_oper
- true_sharing_operator
- true_sharing_multi_copy_
- select_cache_region
- allocate_to_cache
- multi_rw_cache
- check_cache_region
- invalidate_cache
- invalidate_tlb
- evict_line
- barrier
- exclusive_cache_access
- and_barrier
- traffic_and_barrier
- copy_chain_and_barrier
- cache_actions_and_barrier
- choose_legal_cache_op
- select_region_to_mem_buff
- do_while_powering_operator
- coherency_ts_power
- coherency_fs_power
- false_sharing_random_power
- yield
- set_pages
- tlb_invalidate_all
- change_page
- dvm
- dvm_swipe

cdn_core_power_c
- power_down
- power_up
- power_down_single
- power_up_single

cdn_power_ops_c
- power_down_up_counter
- multi_power_down
- multi_power_up
- multi_power_down_power_up
- run_serial_pd_then_pu
- pd_then_pu
- coherency_low_power

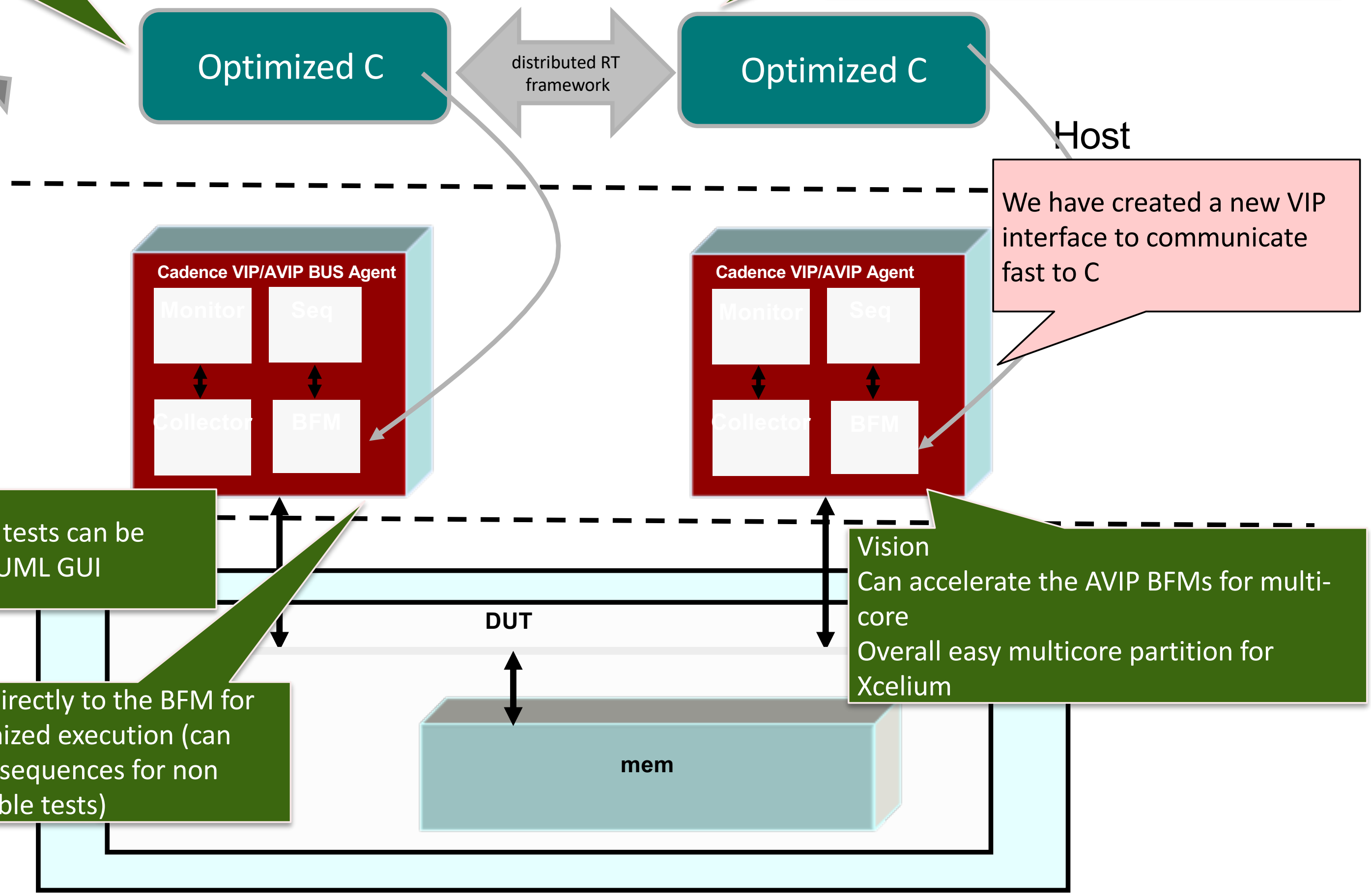
cdn_coherency_fine_ops_c
- shr_read_shr_read
- shr_read_shr_write
- shr_read_invalidate_read
- pack_shr
- state
- do_i
- do_e
- fill_st

Use-cases

We have implemented a rich SOC library that works on top of transactors as well as embedded cores.

User-defined tests can be created via a UML GUI

No re-elaboration



Fast scenarios on the host combining sophisticated constraints solving and run-time reactive repetitions

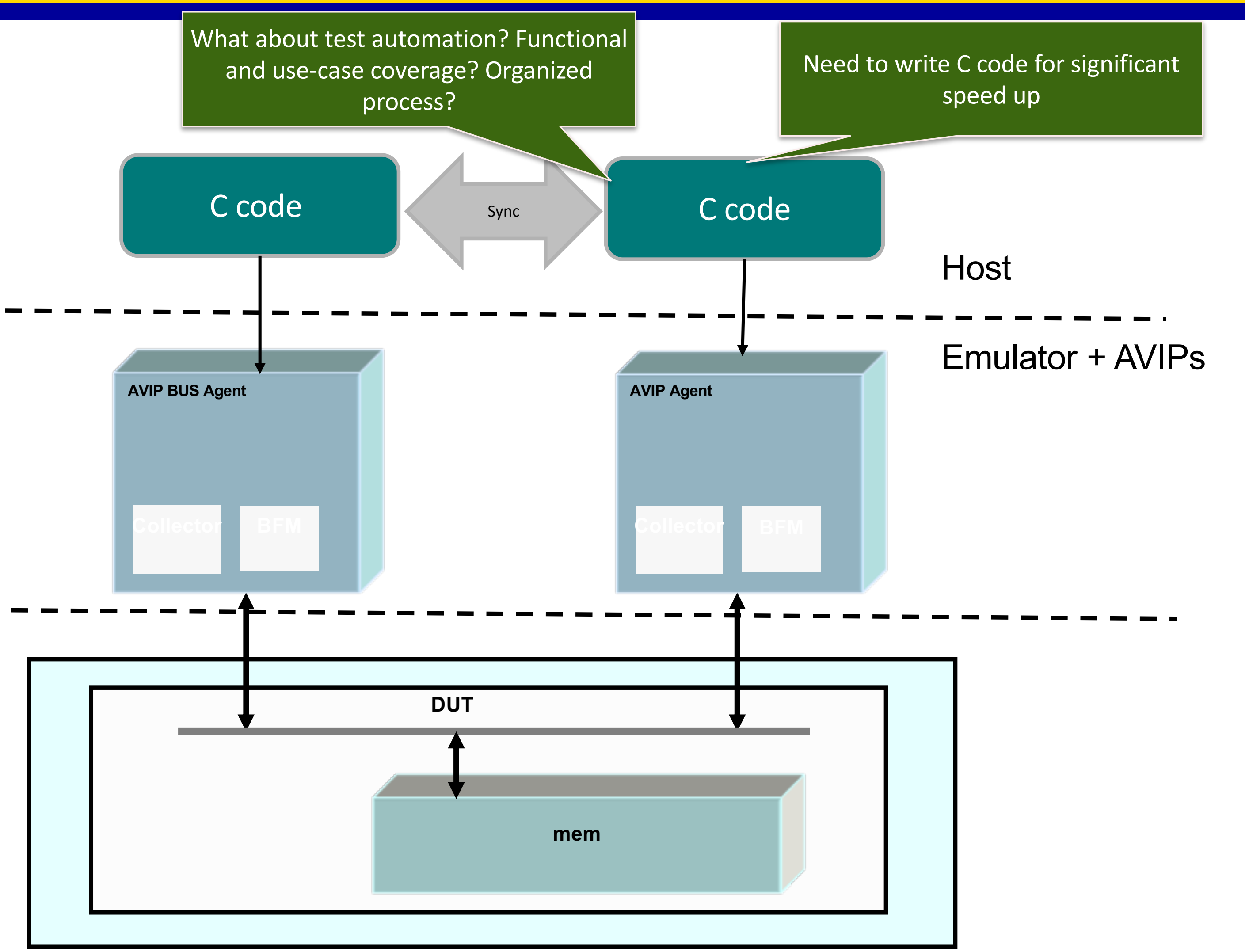
We have created a new VIP interface to communicate fast to C

Talk directly to the BFM for optimized execution (can drive sequences for non portable tests)

Vision
Can accelerate the AVIP BFMs for multi-core
Overall easy multicore partition for Xcelium

Automated Hi-Speed C TestBench *

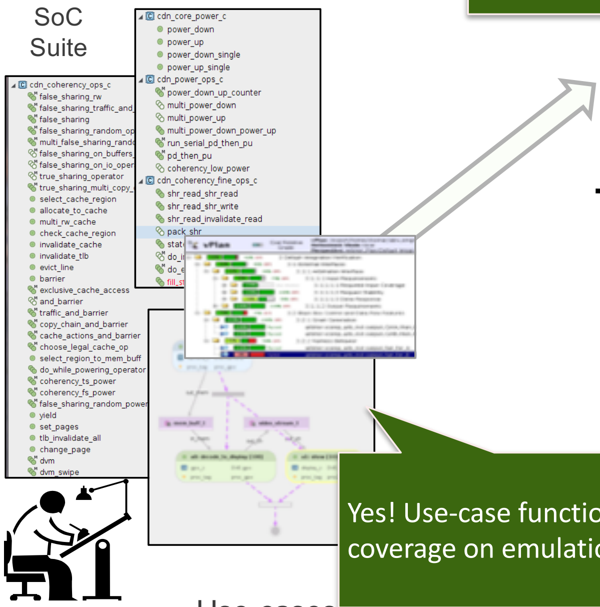
Palladium + AVIP



What about test automation? Functional and use-case coverage? Organized process?

Need to write C code for significant speed up

Automated Hi-Speed C TestBench Palladium + AVIP



SoC Suite

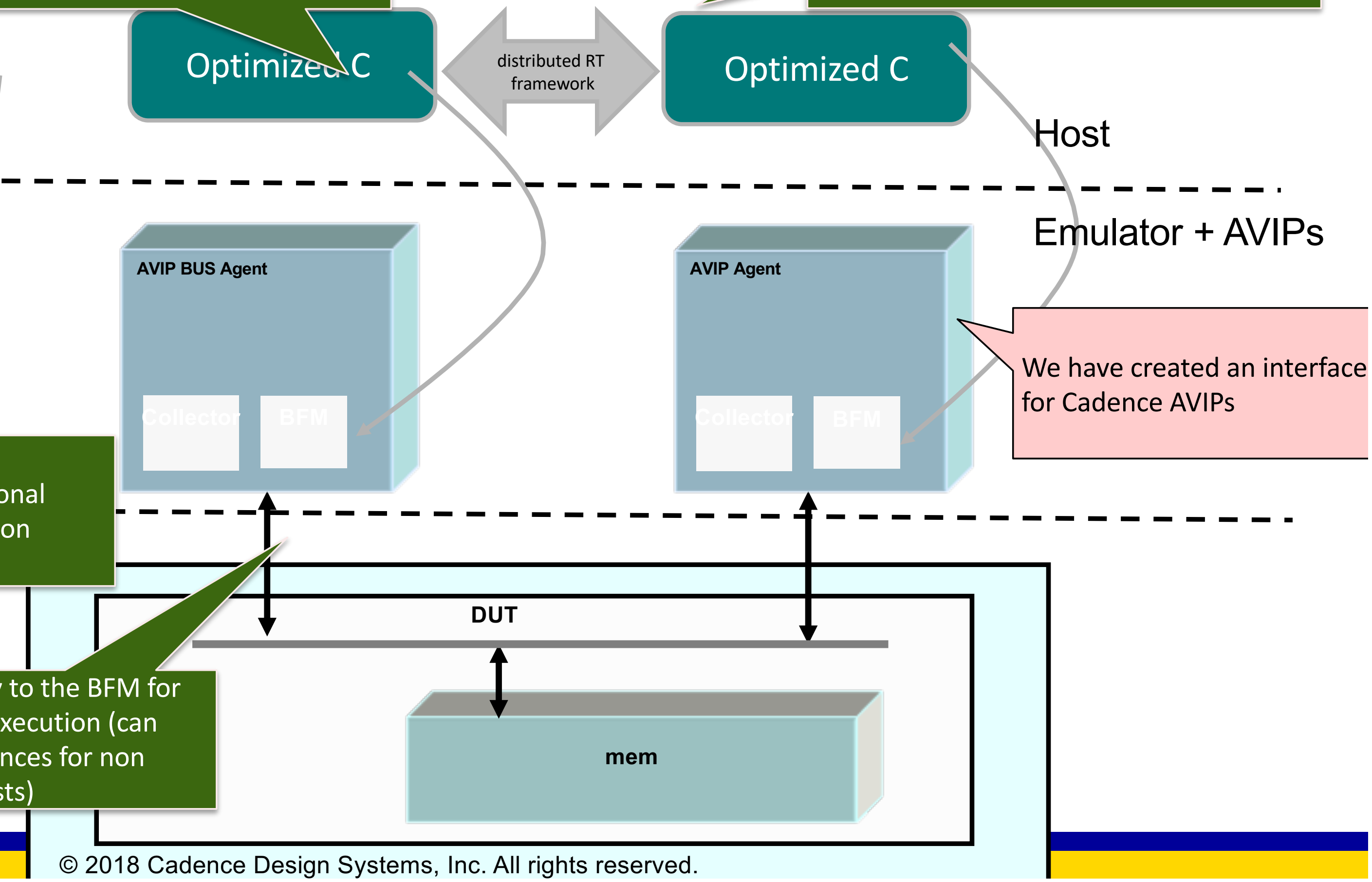
- cdn_coherency_ops_c
  - false_sharing_rw
  - false_sharing_traffic_and
  - false_sharing
  - false_sharing_random_op
  - multi_false_sharing_rand
  - false_sharing_on_buffers
  - false_sharing_on_io_oper
  - true_sharing_operator
  - true_sharing_multi_copy
  - select_cache_region
  - allocate_to_cache
  - multi_rw_cache
  - check_cache_region
  - invalidate_cache
  - invalidate_tlb
  - evict_line
  - barrier
  - exclusive_cache_access
  - and_barrier
  - traffic_and_barrier
  - copy_chain_and_barrier
  - cache_actions_and_barrier
  - choose_legal_cache_op
  - select_region_to_mem_buff
  - do_while_powering_operator
  - coherency_ts_power
  - coherency_fs_power
  - false_sharing_random_power
  - yield
  - set_pages
  - tlb_invalidate_all
  - change_page
  - dvm
  - dvm_swipe

- cdn_core_power_c
  - power_down
  - power_up
  - power_down_single
  - power_up_single
- cdn_power_ops_c
  - power_down_up_counter
  - multi_power_down
  - multi_power_up
  - multi_power_down_power_up
  - run_serial_pd_then_pu
  - pd_then_pu
  - coherency_low_power
- cdn_coherency_fine_ops_c
  - shr_read_shr_read
  - shr_read_shr_write
  - shr_read_invalidate_read
  - pack_shr
  - stat
  - do_i
  - do_e
  - fill_st

Use-cases

The same SOC tests runs efficiently with emulation

Yes! Use-case functional coverage on emulation



High-speed and does not require re-elaboration

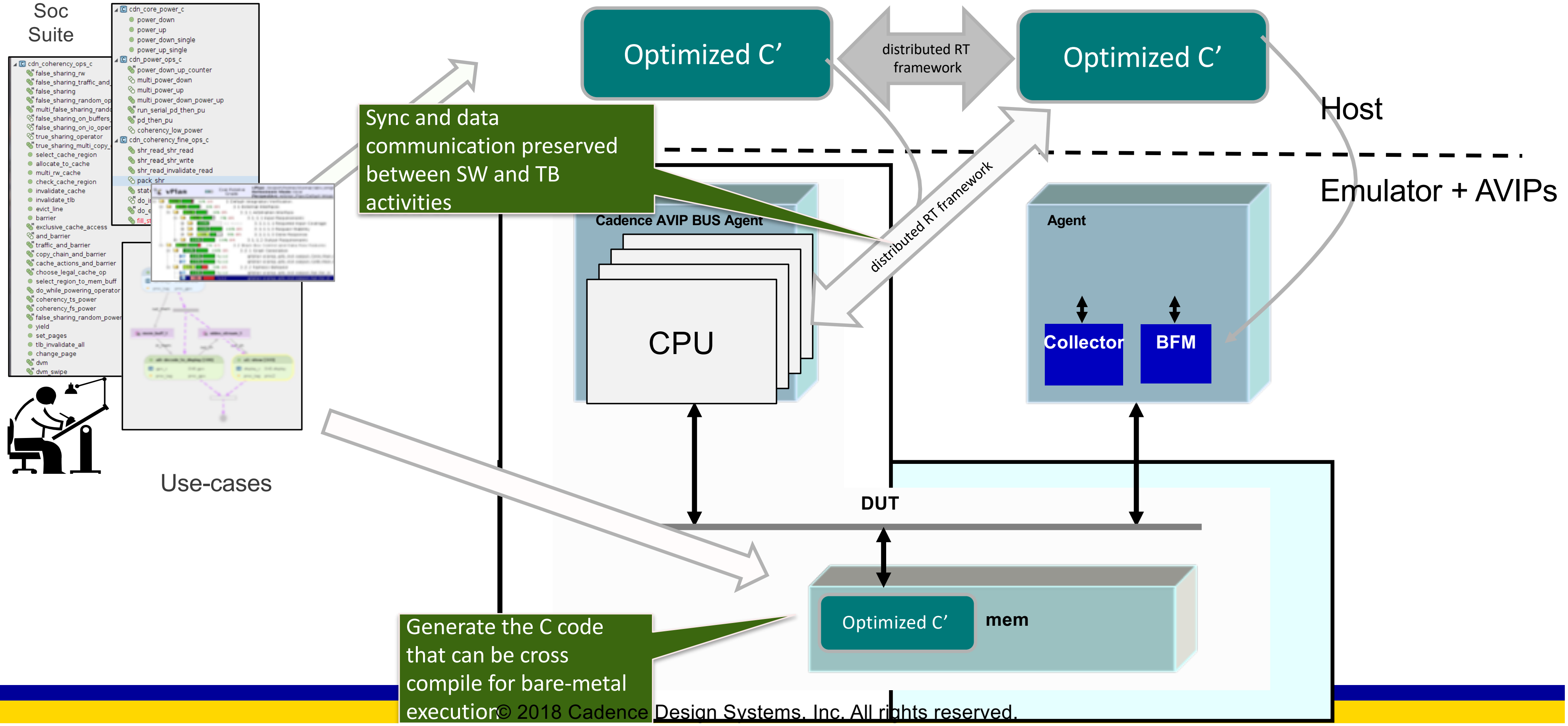
Scenarios on the host combining sophisticated constraints solving and run-time reactive repetitions

We have created an interface for Cadence AVIPs

Talk directly to the BFM for optimized execution (can drive sequences for non portable tests)

Automated Hi-Speed C TestBench

Palladium/Protium + AVIP



Soc Suite

- cdn_coherency_ops_c
  - false_sharing_rw
  - false_sharing_traffic_and_
  - false_sharing
  - false_sharing_random_op
  - multi_false_sharing_rando
  - false_sharing_on_buffers_
  - false_sharing_on_io_oper
  - true_sharing_operator
  - true_sharing_multi_copy_
  - select_cache_region
  - allocate_to_cache
  - multi_rw_cache
  - check_cache_region
  - invalidate_cache
  - invalidate_tlb
  - evict_line
  - barrier
  - exclusive_cache_access
  - and_barrier
  - traffic_and_barrier
  - copy_chain_and_barrier
  - cache_actions_and_barrier
  - choose_legal_cache_op
  - select_region_to_mem_buff
  - do_while_powering_operator
  - coherency_ts_power
  - coherency_fs_power
  - false_sharing_random_power
  - yield
  - set_pages
  - tlb_invalidate_all
  - change_page
  - dvm
  - dvm_swipe

- cdn_core_power_c
  - power_down
  - power_up
  - power_down_single
  - power_up_single
- cdn_power_ops_c
  - power_down_up_counter
  - multi_power_down
  - multi_power_up
  - multi_power_down_power_up
  - run_serial_pd_then_pu
  - pd_then_pu
  - coherency_low_power
- cdn_coherency_fine_ops_c
  - shr_read_shr_read
  - shr_read_shr_write
  - shr_read_invalidate_read
  - pack_shr
  - state
  - do_i
  - do_e
  - fill_st

Sync and data communication preserved between SW and TB activities

Generate the C code that can be cross compile for bare-metal execution

TB Speed-up Summary

- The potential of Testbench speed-up is large
  - Perspec works with post-silicon environments that are million times faster than simulation
- Speed-up is achieve using:
  - Legally parallel mixing of gen-time and with runtime reactiveness and repetition
  - Parallelizing the BFM logic
    - Cadence AVIP can be parallelized in MC
  - Running procedural C on the host
  - Eliminating re-elaboration step for new tests
- Cadence provides significant speedup on
  - On Xcelium™, Palladium®, Protium™ and Post-silicon

Agenda

- The Need for Speed
- Formal methods to avoid sim cycles
- Coding for max sim speed
- Speeding power + mixed-signal SoC
- Break
- Portable Stimulus for faster verification
- Applying hardware to speed system verification
- Summary and call to action

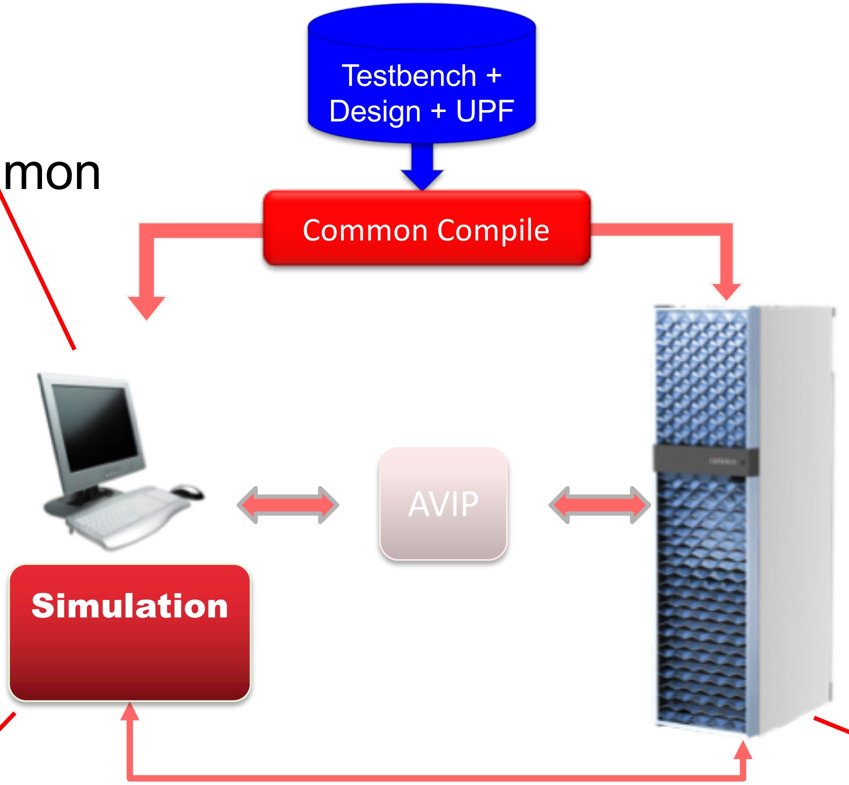
Simulation and Emulation Need to Go Hand in Hand

- C mmon c

Compile full emulation model first in simulation for initial bring-up and debug

Simulation

- Fine-grained debug
- Behavioral testbench & models
- Supports 4-state logic



- E
- H
- RTL/gates, assertions, & coverage
- Supports 2-state logic

After initial model bring-up in simulation, recompile model and co-simulate on emulation

Start simulation with 4-state logic until after reset X/Z propagation. Then hotswap to emulation .

Fast forward to point of interest in emulation without timing and then swap back to simulation and annotate timing (gate-level and RTL subset)

Emulation Market Trends

Migration to centralized emulation farm to maximize investment

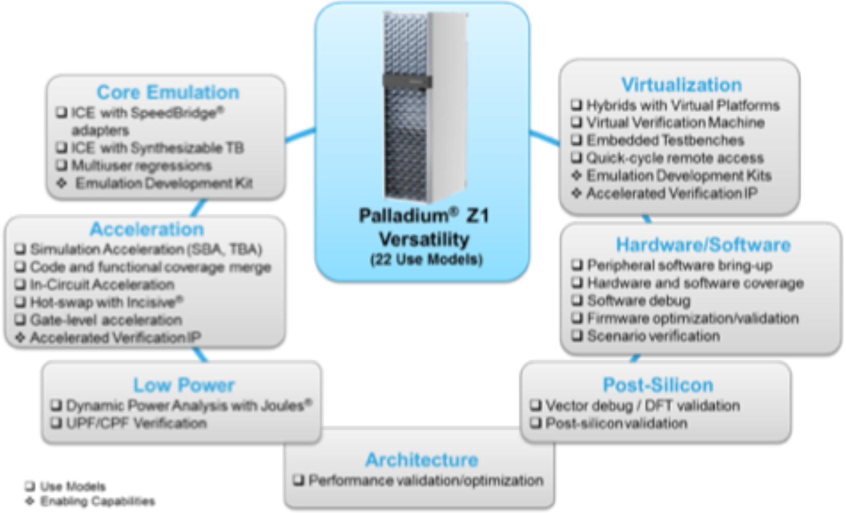
Data center ready

- Data center density and connectivity
  - Rack-based footprint
  - High-speed 56Gbs optical interfaces
  - Host expansion via Infiniband Switch
- Maximize availability and utilization
  - Power module redundancy
  - Hot-swappable power module
  - Fine-grained user granularity of 4MG increments
- Cloud Readiness
  - Enable shorter term access without the need to setup and host
  - Ease adoption via advanced virtualization features

Emulation Market Trends

Migration to centralized emulation farm to maximize investment

- Use model versatility and scalability
  - 22+ use models for RTL and netlist
  - Scaling from 4MG to 9.2BG
- Debug platform
  - FullVision, InfiniTrace, Virtual Verification Machine, Dynamic Probes, SDL,
- Applications
  - In-circuit emulation SpeedBridge® interface
  - Accelerated Verification IP (transactors for most popular interface for the purpose of acceleration)
  - Emulation Development Kit
  - Dynamic Power Analysis
  - Virtual Emulation and Debug



Palladium® Z1 Versatility
(22 Use Models)

Core Emulation
- ❑ ICE with SpeedBridge® adapters
- ❑ ICE with Synthesizable TB
- ❑ Multiuser regressions
- ❖ Emulation Development Kit

Acceleration
- ❑ Simulation Acceleration (SBA, TBA)
- ❑ Code and functional coverage merge
- ❑ In-Circuit Acceleration
- ❑ Hot-swap with Incisive®
- ❑ Gate-level acceleration
- ❖ Accelerated Verification IP

Low Power
- ❑ Dynamic Power Analysis with Joules®
- ❑ UPF/CPF Verification

Architecture
- ❑ Performance validation/optimization

Post-Silicon
- ❑ Vector debug / DFT validation
- ❑ Post-silicon validation

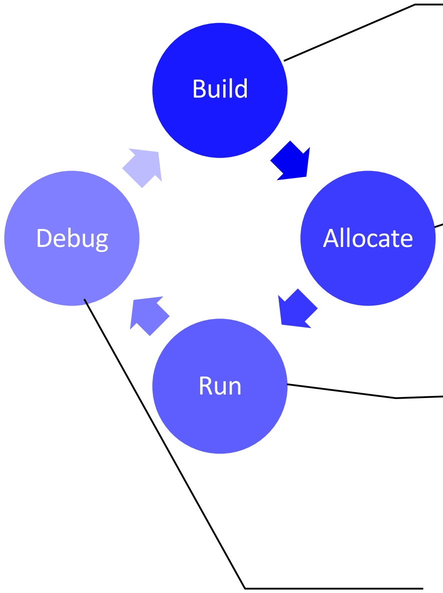
Hardware/Software
- ❑ Peripheral software bring-up
- ❑ Hardware and software coverage
- ❑ Software debug
- ❑ Firmware optimization/validation
- ❑ Scenario verification

Virtualization
- ❑ Hybrids with Virtual Platforms
- ❑ Virtual Verification Machine
- ❑ Embedded Testbenches
- ❑ Quick-cycle remote access
- ❖ Emulation Development Kits
- ❖ Accelerated Verification IP

❑ Use Models
❖ Enabling Capabilities

Way beyond emulation

HW-assisted Verification Productivity Loop



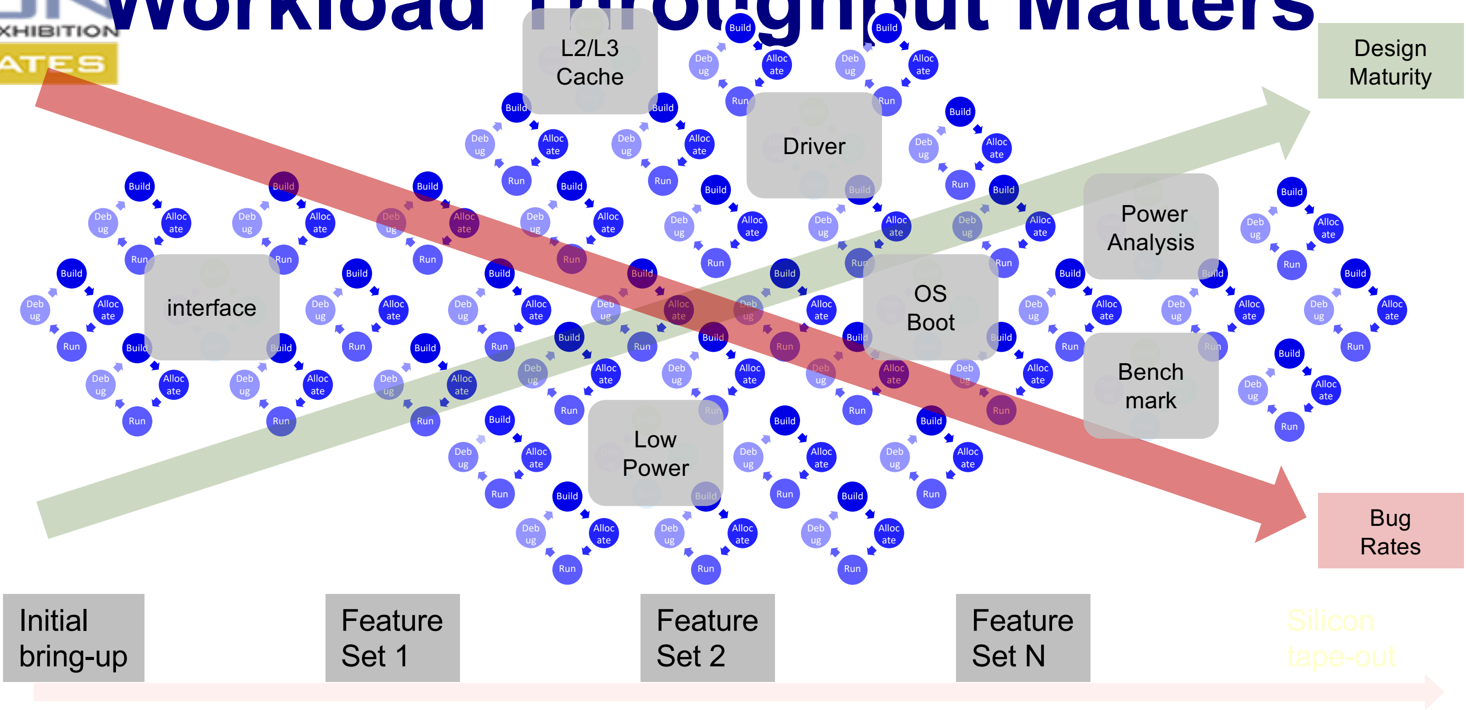
Compile databases for different workloads
(Compile: speed, automation, # of workstations)

Allocate as many workloads as possible
(Utilization efficiency: # of parallel jobs, relocation)

Run workloads based on priorities
(Speed: use models, interface solutions)

Debug for both pre & post silicon bugs
(Visibility: trace depth, dynamic trigger)

Workload Throughput Matters

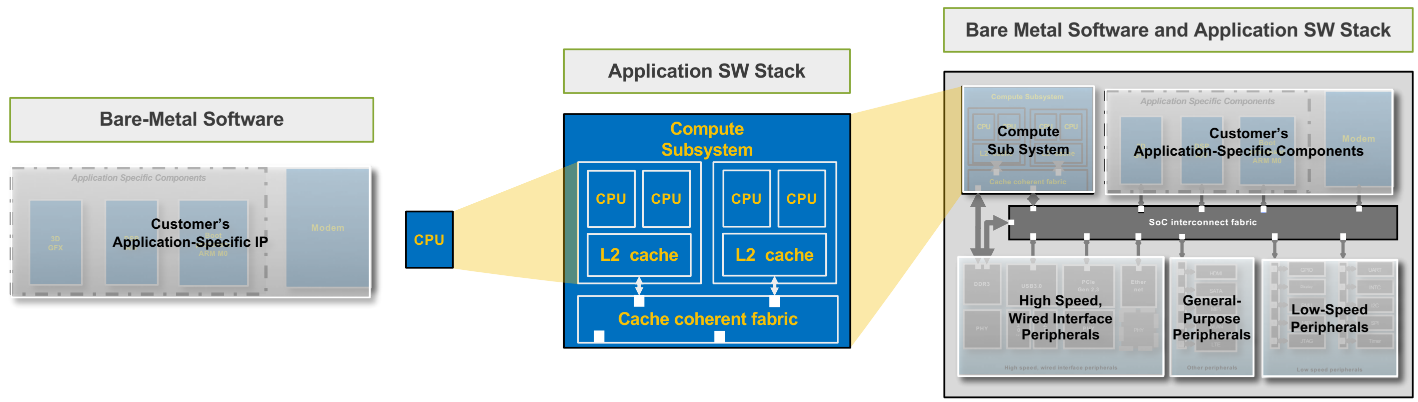


- How often will this loop be repeated over the course of a project?
- What type/size of workloads and users will be using emulation?
- How many workloads will be required to verify and validate your design?

Workloads

Scalability from Small to Large Payload Sizes

IP	Subsystem	SoC / System
2-16M Gates	32-128M Gates	128-4096M Gates or more



Bare-Metal Software

Customer's Application-Specific IP

CPU

Application SW Stack

Compute Subsystem

CPU CPU CPU CPU

L2 cache L2 cache

Cache coherent fabric

Bare Metal Software and Application SW Stack

Compute Sub System

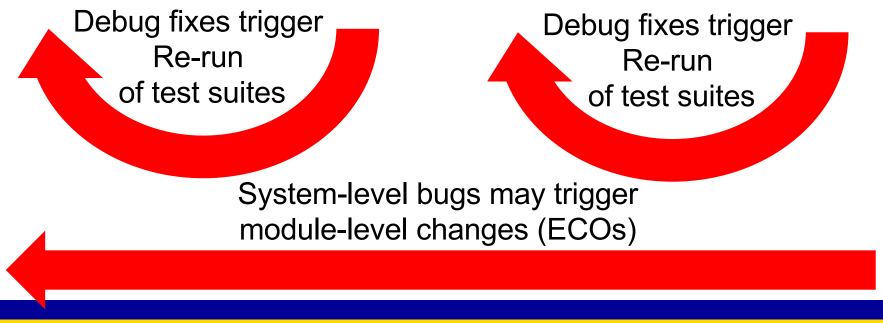
Customer's Application-Specific Components

SoC interconnect fabric

High Speed, Wired Interface Peripherals

General-Purpose Peripherals

Low-Speed Peripherals



Debug fixes trigger Re-run of test suites

Debug fixes trigger Re-run of test suites

System-level bugs may trigger module-level changes (ECOs)

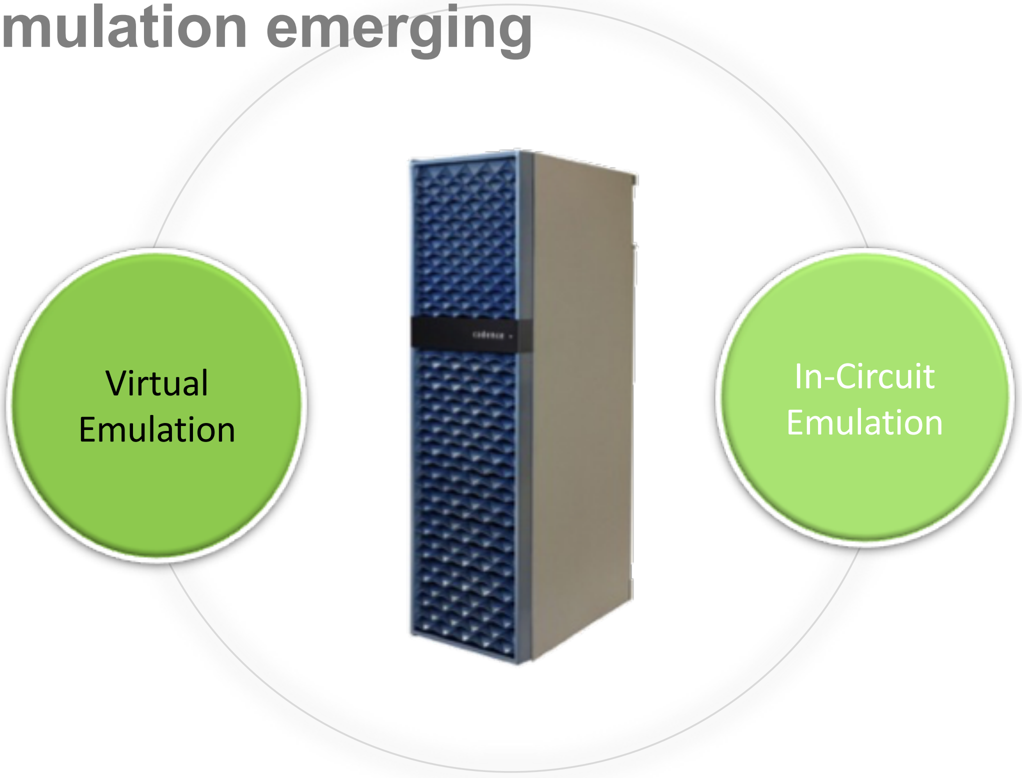
Emulation needs to scale beyond 4 billion gates while allowing resource to be shared with best user granularity of 4 million gates

Emulation Choices

In-Circuit Emulation mature and vibrant, Virtual Emulation emerging

Virtual Emulation

- Virtual enablement of SW driven HW verification
- Flexible debug
- Earlier access with Hybrid verification
- Ease of replication
- Adoption path for simulation and simulation acceleration users



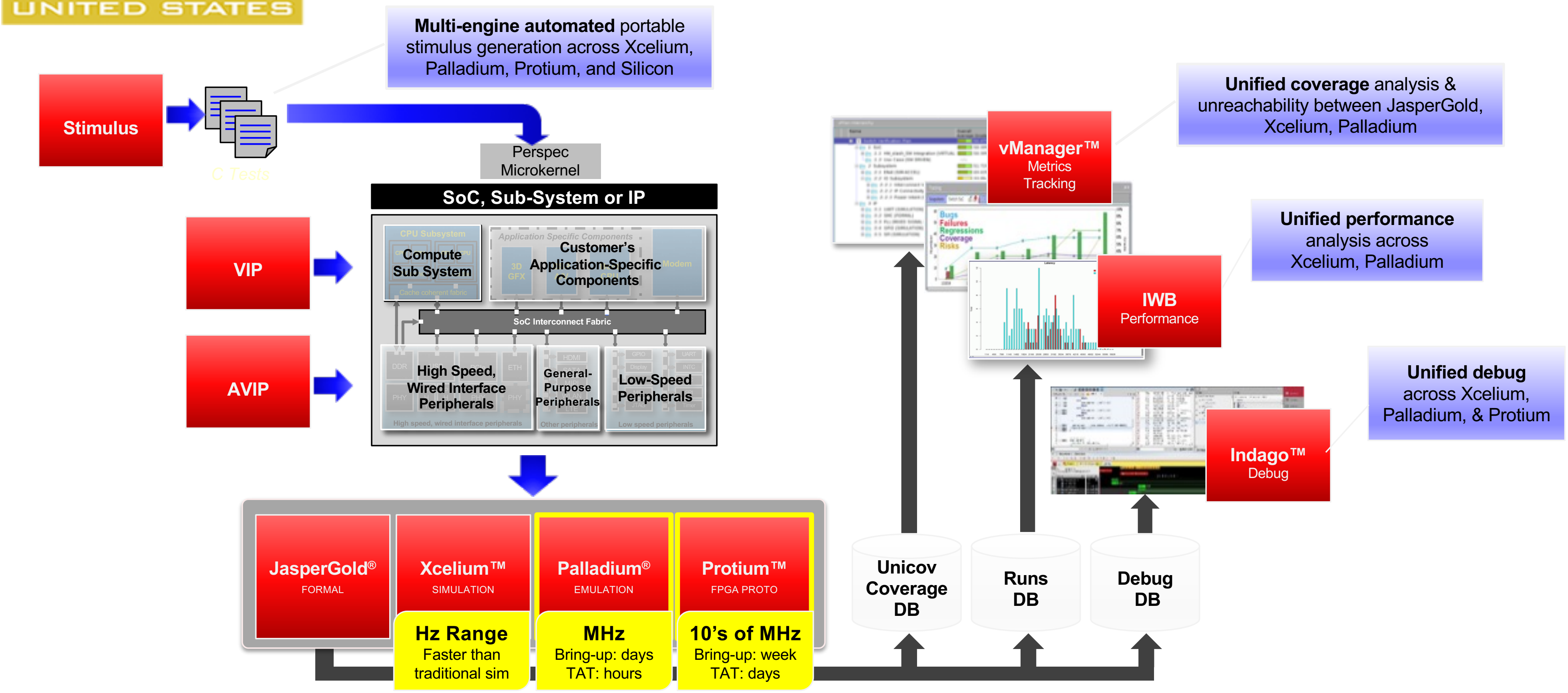
Virtual Emulation

In-Circuit Emulation

In-Circuit Emulation

- Highest performance
- High fidelity live traffic
- Traffic generation and analysis with 3rd party testers
- Remote and re-locatable access
- Post-silicon debug & reuse
- Migration path to FPGA prototyping

Verification Flow Example



Multi-engine automated portable stimulus generation across Xcelium, Palladium, Protium, and Silicon

Unified coverage analysis & unreachability between JasperGold, Xcelium, Palladium

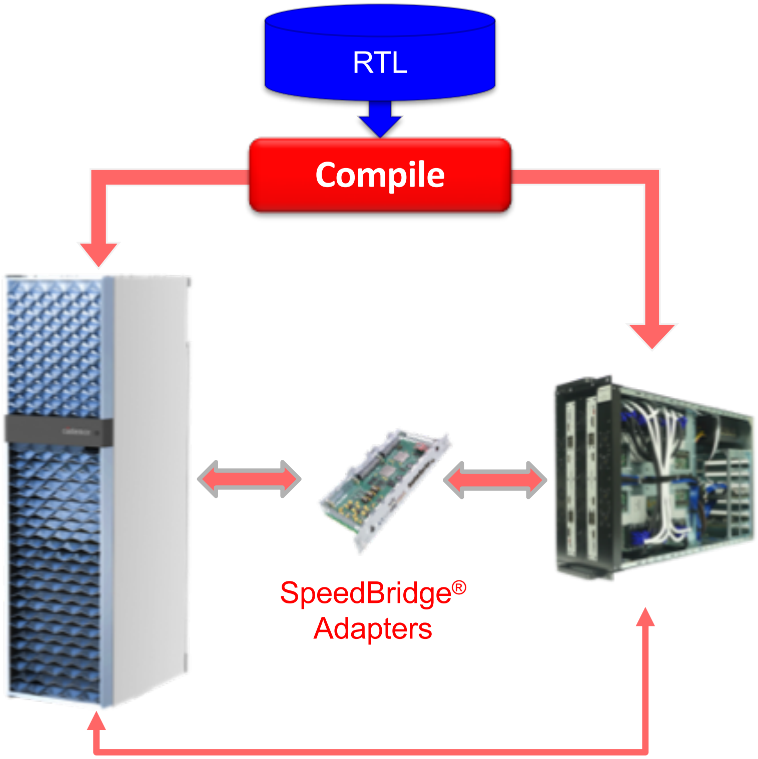
Unified performance analysis across Xcelium, Palladium

Unified debug across Xcelium, Palladium, & Protium

Emulation and FPGA Prototyping Need to Go Hand in Hand …

Emulation

- Best debug
- SoC acceleration, hardware/software
- Power and performance analysis



FPGA

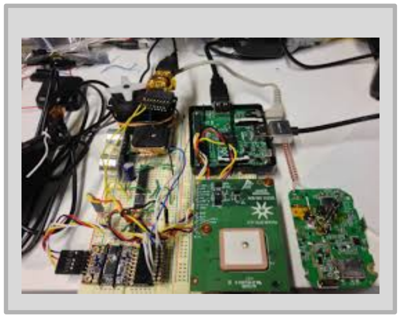
- Highest performance
- Software development
- Hardware/software regressions

Users need Congruency and a **common environment**

FPGA-Based Prototyping Is Fragmented

Disjointed, lacking integrated flow and automation

- FPGA-based prototyping

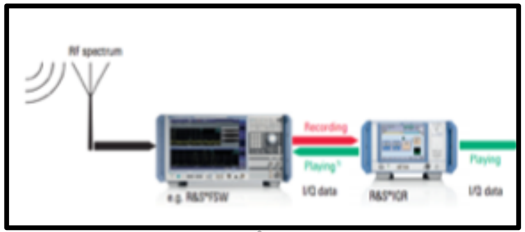


Challenges:

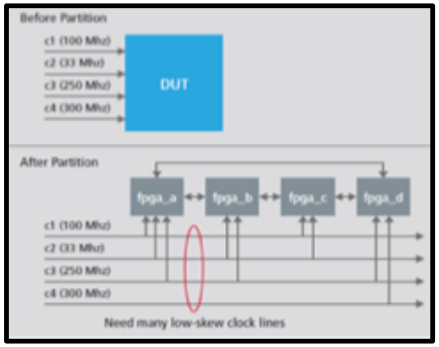
- Fragmented
  - Requires RTL modifications
- Lack auto compilation
  - Memory and clocks
  - Partitioning
- Lack of flow integration
  - Emulation and prototyping
  - Configuration reuse
  - FPGA P&R

FPGA-Based Prototyping Is Hard To Do…

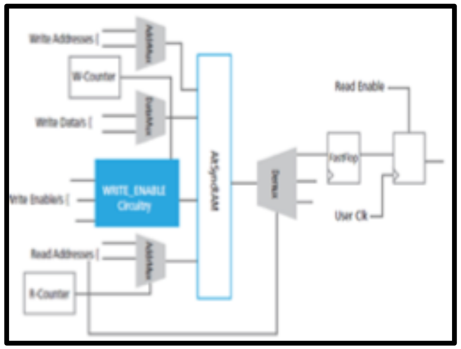
Interfaces

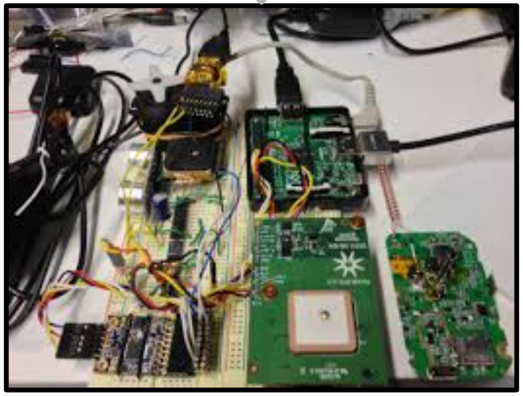


Clocking



Memories

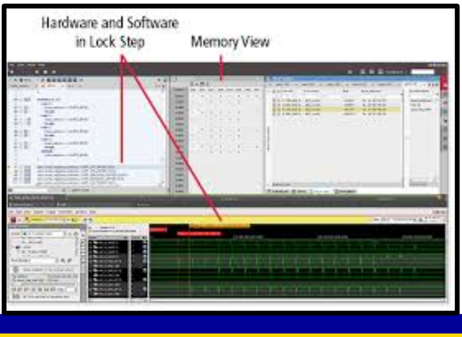




Software

Applications
Application Framework
Libraries
Android Runtime
Linux Kernel
Hardware

Debug



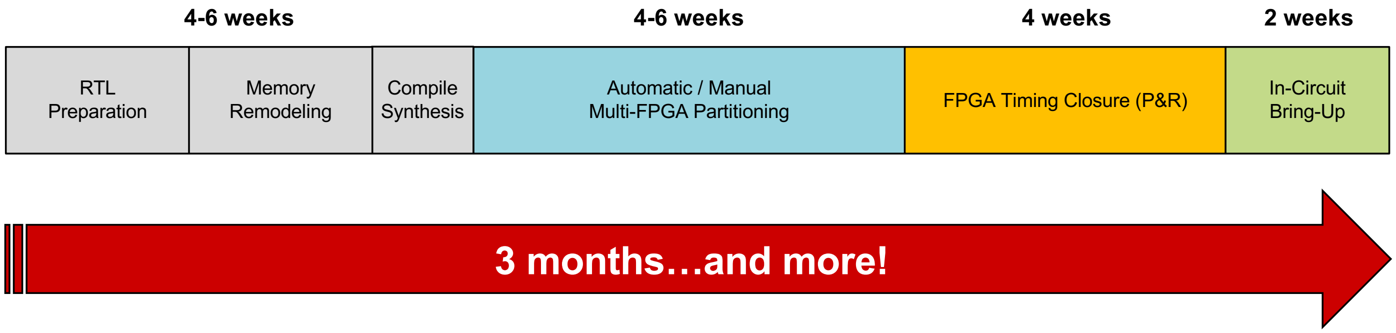
© 2018 Cadence Design Systems, Inc. All rights reserved.

Really, Really Hard To Do

FPGA-based prototyping has become the methodology of choice for early software development

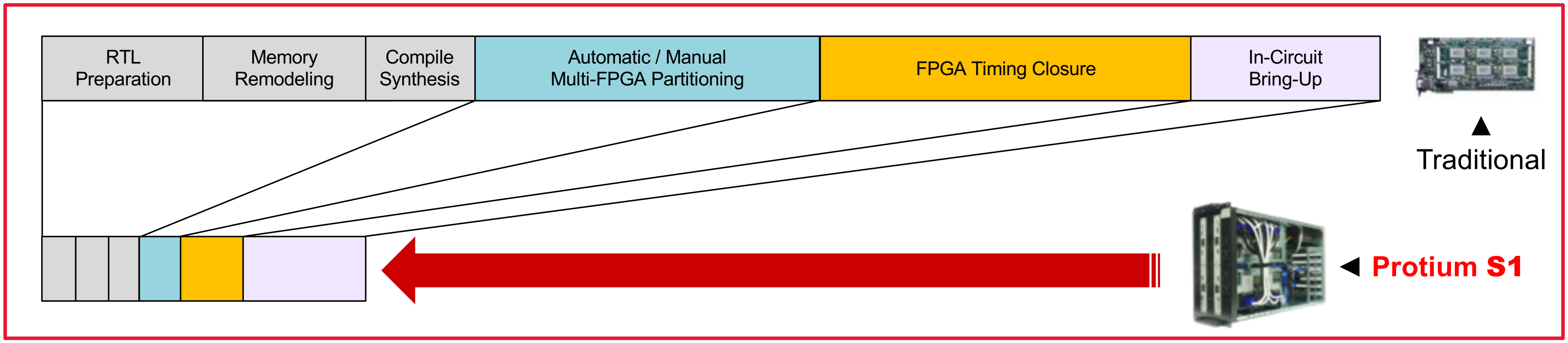
BUT…

Prototyping implementation and bring-up takes too long and there has, so far, not been any easy transition from simulation and emulation into FPGA-based prototyping



Or Is It?

How to address the prototyping challenges



RTL Preparation | Memory Remodeling | Compile Synthesis | Automatic / Manual Multi-FPGA Partitioning | FPGA Timing Closure | In-Circuit Bring-Up

▲ Traditional

◄ Protium S1

□ No RTL modifications needed

- Clocking / number of clocks
- Automated memory compilation and modeling

□ Fully automatic, multi-FPGA partitioning

- Optional manual optimization

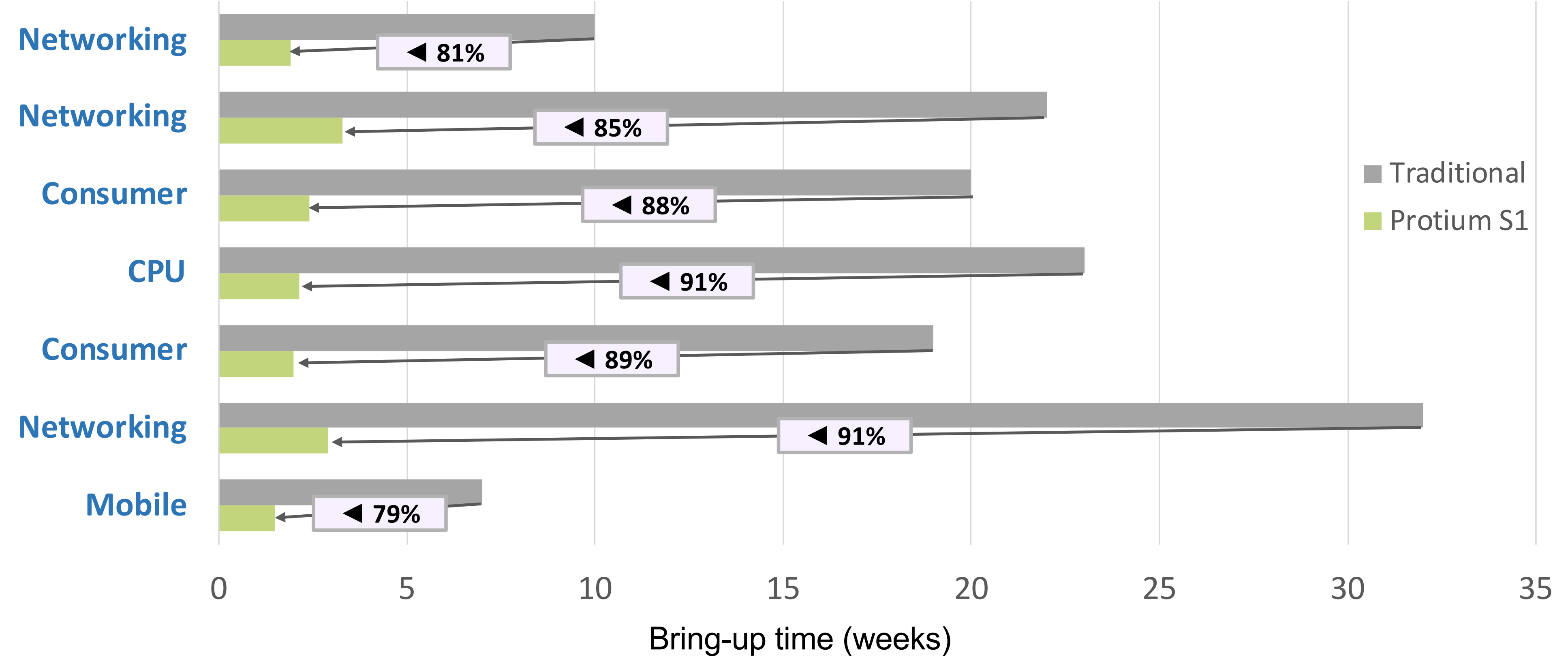
🟧 FPGA timing closure

- Multiple design integrations per day
- Avoids time-consuming FPGA P&R

□ Fully integrated FPGA P&R

- Automatic constraint generation
- Guaranteed P&R success

Fast Time-to -Prototype (TTP)

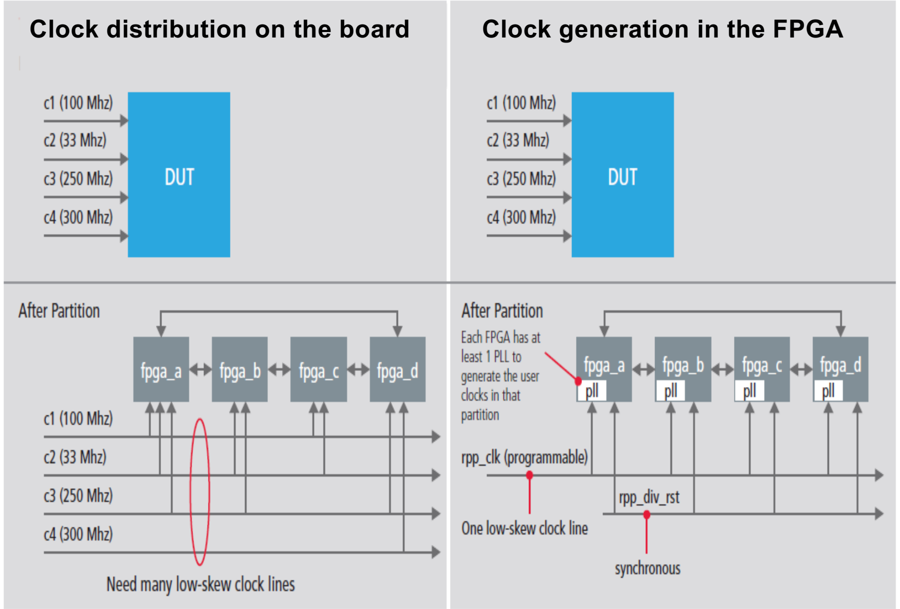


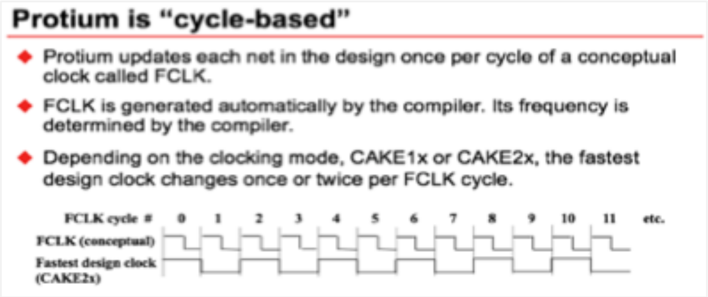
Note: Sample customer bring-up gains over traditional FPGA-based prototyping solutions

No RTL Modifications – Clocking

- Traditional imitations:
  - Gated clock, multiplexed clocks
  - # of clocks
  - **Difficult to achieve FPGA timing closure**
  - **Long iteration times / long FPGA P&R times**
  - **Unpredictable results and prototype behavior**

- Automated Clocking
  - No hold-time violations in user clock domains
  - Removes any FPGA-specific clock limitations
  - **Supports unlimited # of design clocks**
  - **Improves FPGA timing closure**
  - **Accelerates FPGA P&R times**

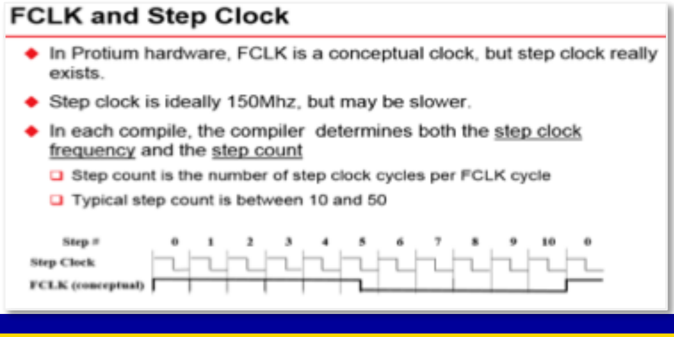




Protium is "cycle-based"

- Protium updates each net in the design once per cycle of a conceptual clock called FCLK.
- FCLK is generated automatically by the compiler. Its frequency is determined by the compiler.
- Depending on the clocking mode, CAKE1x or CAKE2x, the fastest design clock changes once or twice per FCLK cycle.

FCLK cycle #
0
1
2
3
4
5
6
7
8
9
10
11
etc.
FCLK (conceptual)
Fastest design clock (CAKE2x)



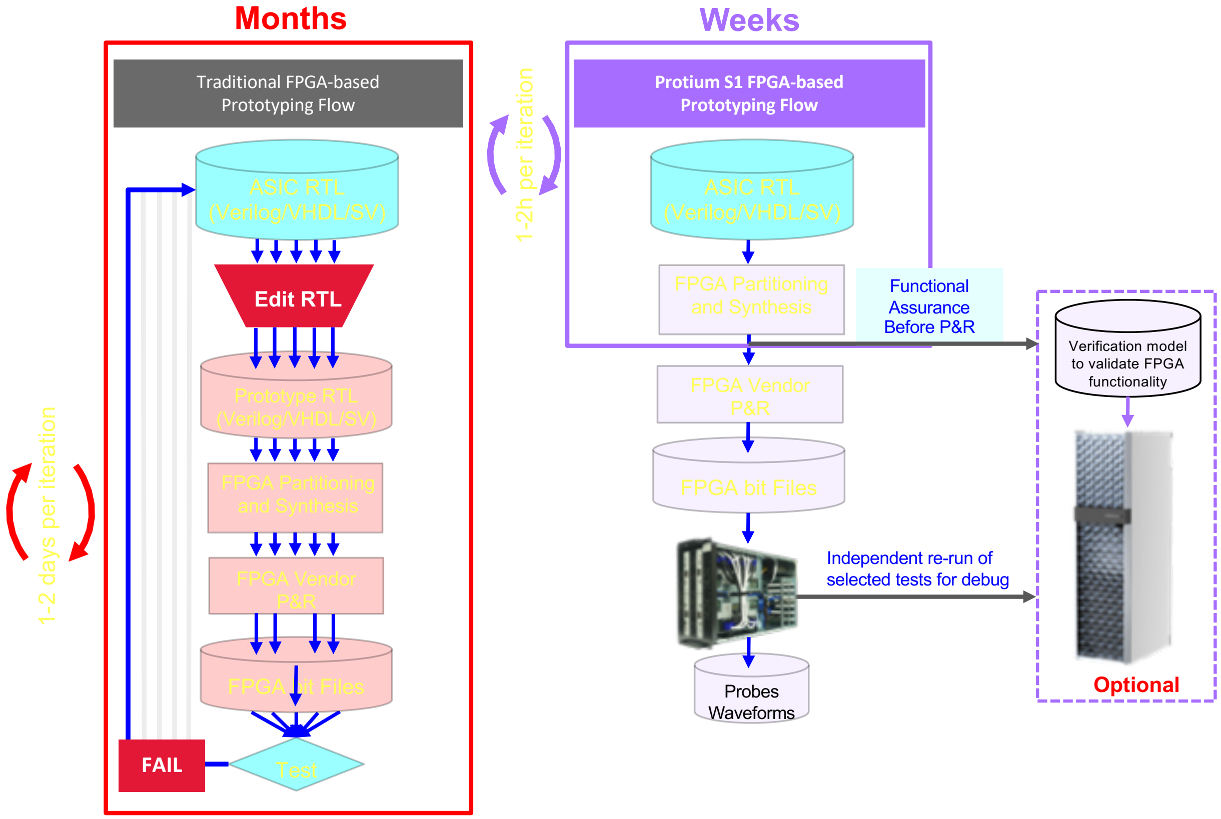
FCLK and Step Clock

- In Protium hardware, FCLK is a conceptual clock, but step clock really exists.
- Step clock is ideally 150Mhz, but may be slower.
- In each compile, the compiler determines both the <u>step clock frequency</u> and the <u>step count</u>
  - Step count is the number of step clock cycles per FCLK cycle
  - Typical step count is between 10 and 50

Step #
0
1
2
3
4
5
6
7
8
9
10
0
Step Clock
FCLK (conceptual)

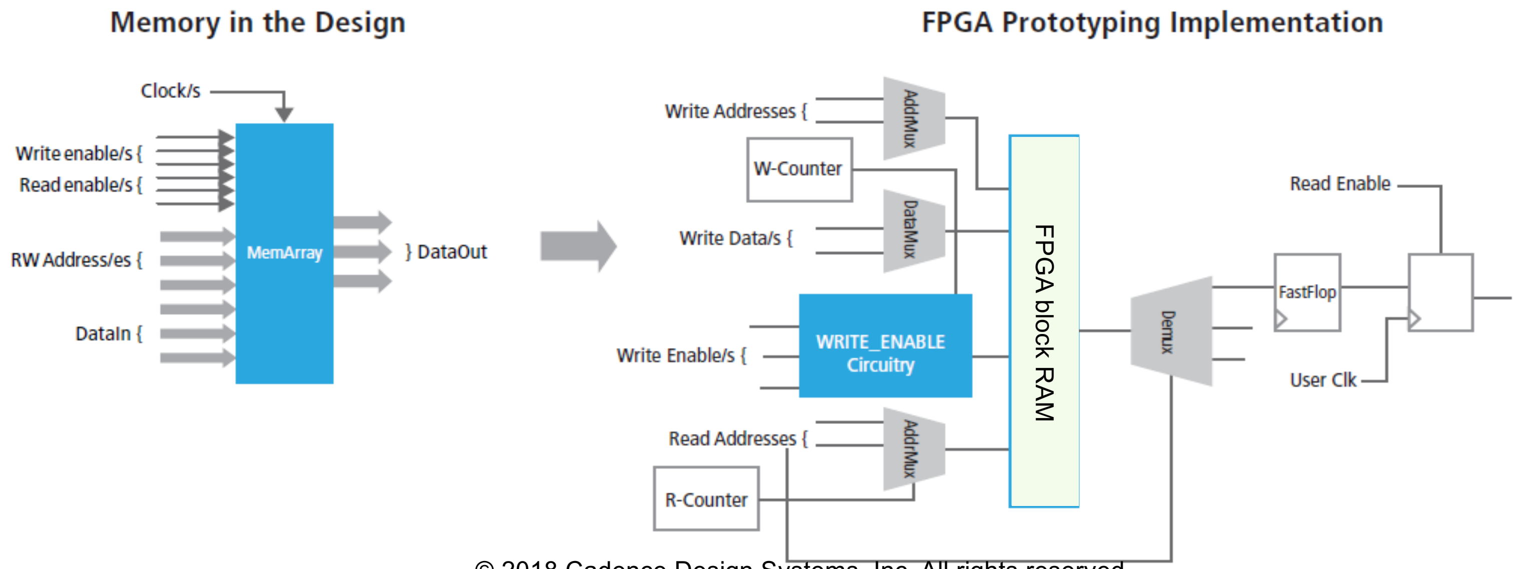
Users Need a Fully Integrated Implementation Flow

- **Automated prototyping flow reduces time-to-prototype (TTP) from months to weeks**
  - Design changes have much lesser time impact on iterations
  - Simpler single pass flow iterations are run in hours not days
  - Software development gets a head start measured in months not days


Months
Weeks
Traditional FPGA-based Prototyping Flow
Protium S1 FPGA-based Prototyping Flow
ASIC RTL (Verilog/VHDL/SV)
Edit RTL
Prototype RTL (Verilog/VHDL/SV)
FPGA Partitioning and Synthesis
FPGA Vendor P&R
FPGA bit Files
Test
FAIL
1-2 days per iteration
1-2h per iteration
ASIC RTL (Verilog/VHDL/SV)
FPGA Partitioning and Synthesis
Functional Assurance Before P&R
FPGA Vendor P&R
FPGA bit Files
Independent re-run of selected tests for debug
Probes Waveforms
Verification model to validate FPGA functionality
Optional

No RTL Modifications - Memories

- No ASIC RTL changes
  - Automatic conversion of latches and tri-states
  - Automatic memory compilation and modeling
  - Fully automated clock tree transformation
    - Automatic conversion of gated and multiplexed clocks



Comprehensive, Automated Memory Support

Conversion and **implementation** of memories is one of the **most challenging** and **time-consuming** steps in bring-up of an FPGA-based prototype (often taking many weeks to complete).

Type	Size	Palladium MMP	Upload/ Download	Perform.	Comments
FPGA-internal	~50Mbits / FPGA	Yes	Yes	Full speed	• Fully automatic compile
XSRAM (automated small external memory)	128 Mbytes per memory card	Some	Yes	<12MHz	• Fully automatic compile • Extends 'FPGA-internal' memory to external SRAM • Useful for Serial Parallel Interface (SPI)-flash and other small memories (e.g. boot ROM)
XDRAM (automated bulk memory)	16 GBytes per XDRAM card	DDR family models	Yes	<16MHz	• semi automatic compile • Leverages XDRAM hardware Support for DDR3/4, LPDDR3/4
DCMC (Direct Connected Memory Card)	x GBytes (depending on memories used)	No	No	Full design speed	• Design change may be required, depending on memory type • App notes available
FCMC (Full-custom Memory Card)	Custom	No	No	Full design speed	• Fully custom development

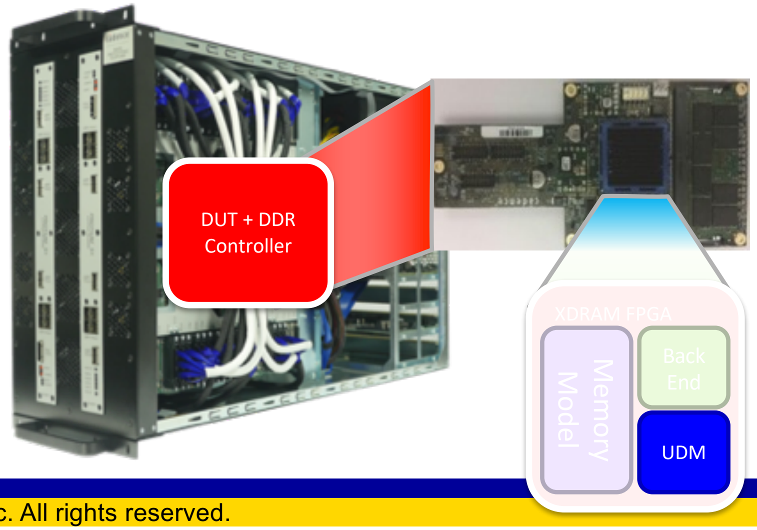
Protium S1
Memory compile capabilities :

- Smaller memories are **automatically compiled** into FPGA-internal resources
- For larger, off-FPGA memories, the Protium platform offers **several automated solutions,** see table

Innovative XDRAM & XSRAM Solution

- XSRAM
  - Benefits:
    - Increases FPGA internal memory from 80Mbits to 128MBytes (>10x)
    - Automatic mapping of any memory type
    - Support for multi-port memories
    - Support for backdoor upload/download

- XDRAM
  - Benefits:
    - Adds DDRx bulk memories
    - Supports LPDDR2/3/4; DDR3/4; HBM
    - No change to design memory controller and firmware
    - Support for backdoor upload/download
    - Acts as memory SpeedBridge (timing, refresh, etc.)



DUT + DDR Controller

XDRAM FPGA

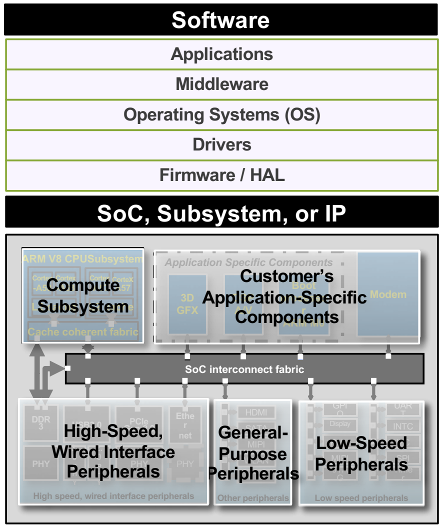
Memory Model

Back End

UDM

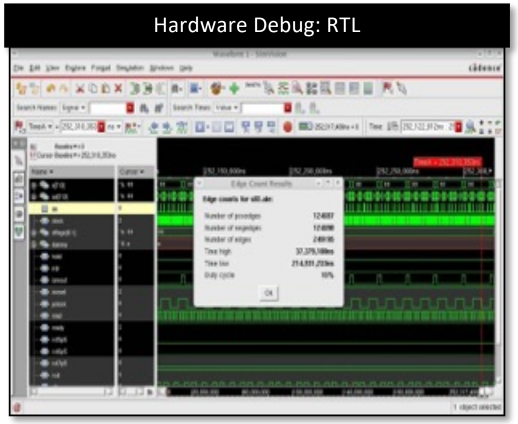
Hardware and Software Debug

- Waveforms across partitions
  - Design-centric view vs. FPGA-centric
- Force/release
  - Predefined signals (at compile time) to “0” or “1” during runtime
- Monitor signal
  - Real-time monitoring of predefined (at compile time) signals
- External data capture card
  - Thousands of signals for millions of cycles
- State read-back



- Backdoor memory access
  - Quickly change boot code, software, etc.
- Clock control
  - Start/stop the clock on demand
- Fully scriptable runtime environment
- Remote access
  - Network resource anytime from anywhere
- Assertion checkers
- High-performance link to software model

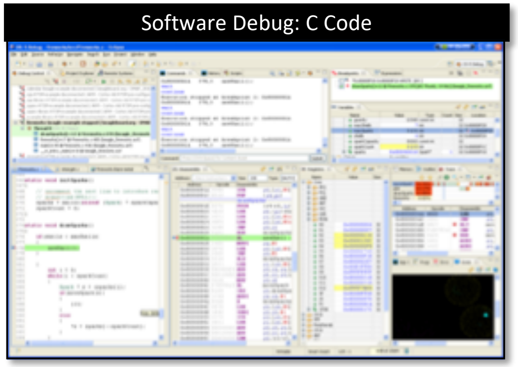
Hardware Debug: RTL



Probes

JTAG

Software Debug: C Code



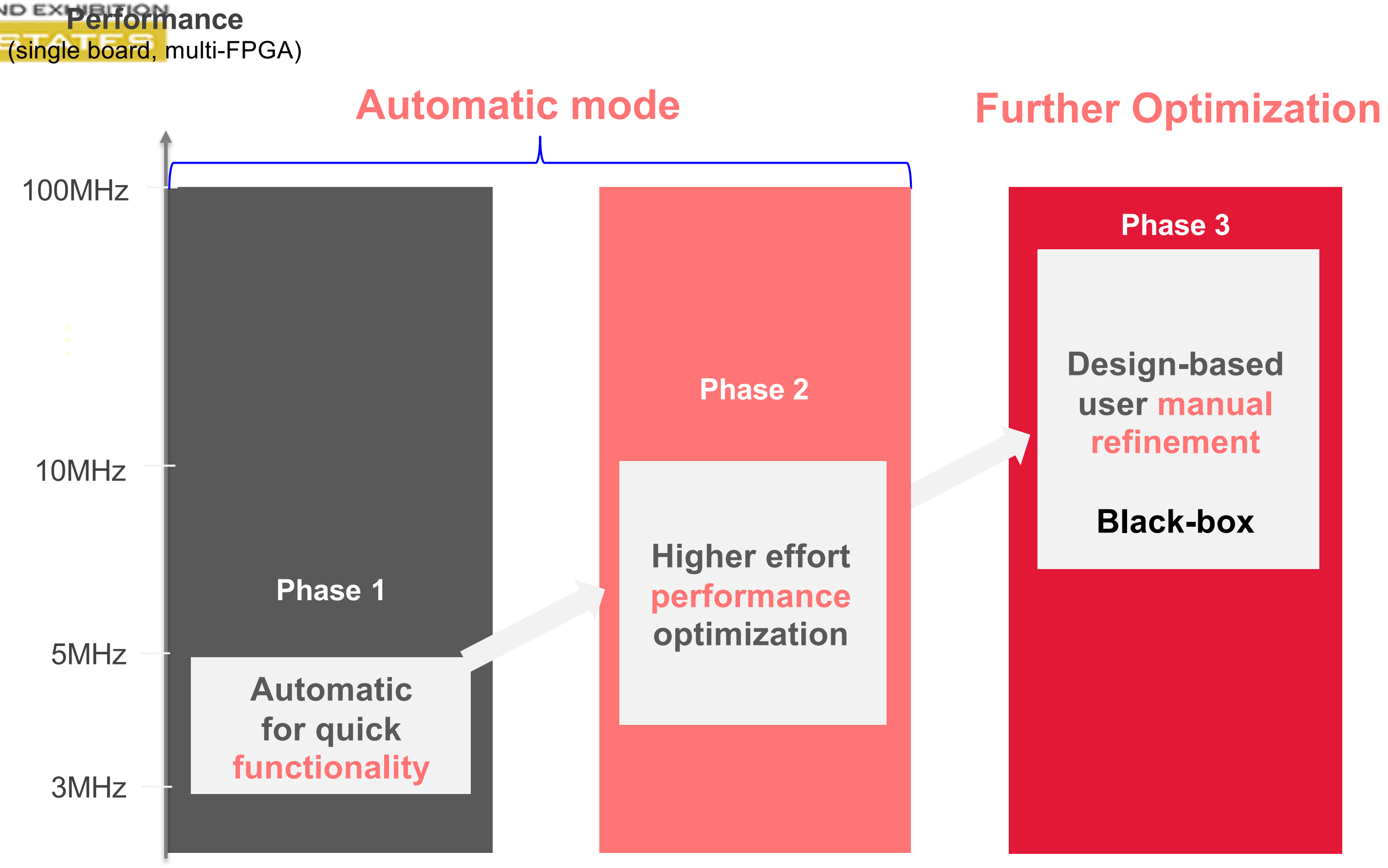
Daughtercards and peripherals

Advanced Debug

Unique to Protium™

- External data capture card
  - Thousands of signals for millions of (DUT) clock cycles
- Force/release signal
  - Forces predefined signals (at compile time) into "0" or "1" during runtime
- Memory upload and download
- Monitor signal
  - Real-time monitoring of predefined (at compile time) signals
- State read-back without recompile
- Assertion checkers
- Runtime
  - Start/stop clock capability (run "N" cycles)
- Probes
  - Runtime data capture of predefined signals for offline waveform viewing

Scalable Performance



Performance
(single board, multi-FPGA)

Automatic mode

Further Optimization

Phase 1

Automatic for quick functionality

Phase 2

Higher effort performance optimization

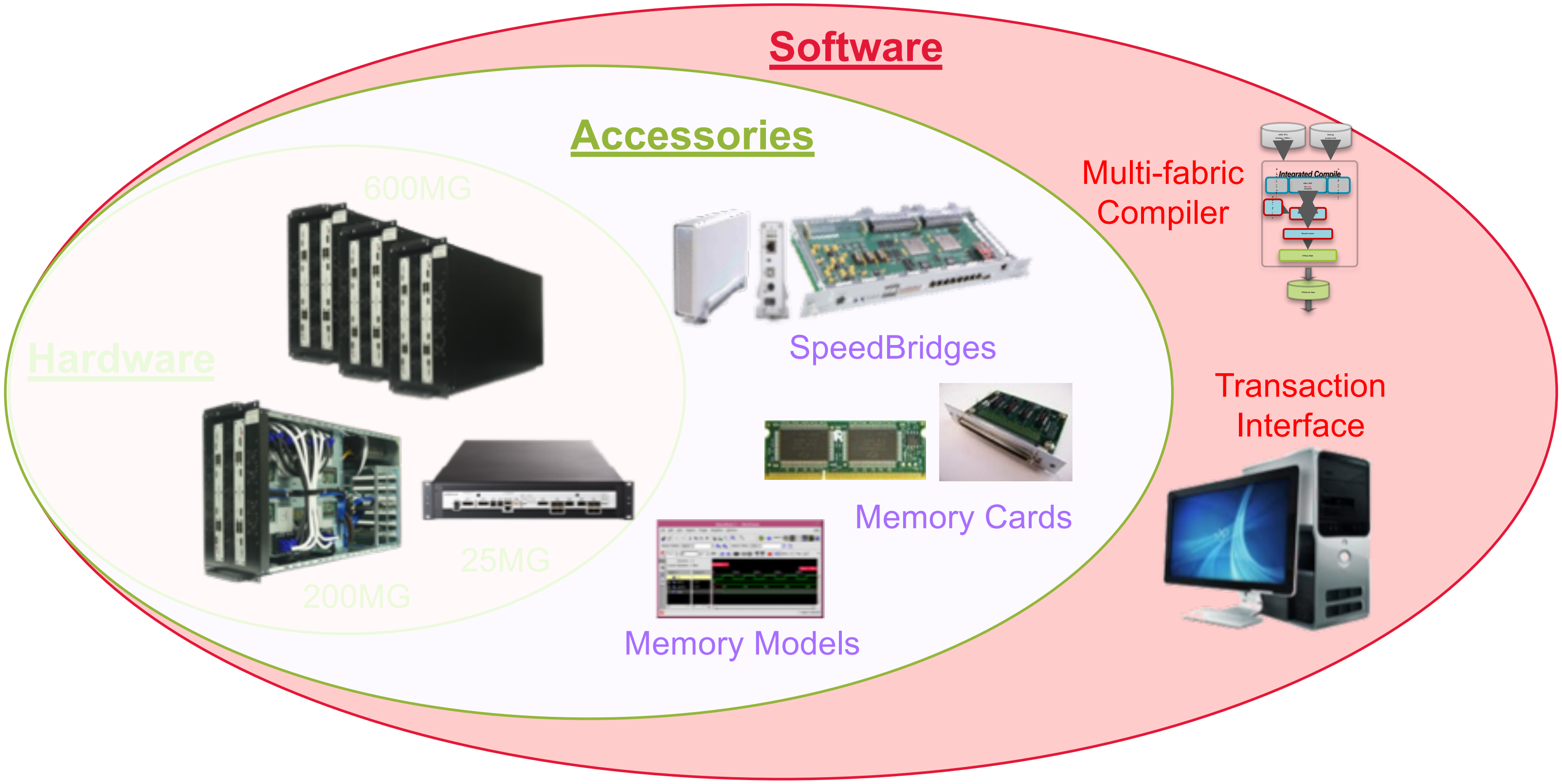
Phase 3

Design-based user manual refinement

Black-box

Protium S1 Prototyping Solution

Industry’s first comprehensive, fully integrated solution



Software

Accessories

Hardware

600MG

25MG

200MG

SpeedBridges

Memory Cards

Memory Models

Multi-fabric Compiler

Transaction Interface

Agenda

- The Need for Speed
- Formal methods to avoid sim cycles
- Coding for max sim speed
- Speeding power + mixed-signal SoC
- Break
- Portable Stimulus for faster verification
- Applying hardware to speed system verification
- Summary and call to action

Summary and Call to Action

- Every facet of SoC verification benefits from speed
- Faster engines, faster coding, more efficient cycles (MDV) and avoiding simulation cycles are all approaches to gain verification speed
- So tap into the verification speed-force today
  - Add JasperGold® Apps
  - Run more efficient code faster in Xcelium™
  - Create more efficient stimulus faster in Perspec™
  - Verify systems faster in Palladium® and Protium®

Questions?

Thank you!