Data-Driven Verification: Driving the next wave of productivity improvements

Cadence Presenters
Larry Melling, Director Product Management
Chris Komar, Group Director Product Engineering

Sharon Rosenberg, Solutions Architect
Michael Young, Group Director Product Management

UltraSoC Presenter
Hanan Moller, Systems Architect
The Problem

- Verification cost growing exponentially with complexity
- Finite budget
- Finite resources
- Compromise quality/Increase risk
# The Problem

<table>
<thead>
<tr>
<th>Integration</th>
<th>Testbench Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS IP Integration</td>
<td></td>
</tr>
<tr>
<td>SoC IP Integration</td>
<td></td>
</tr>
<tr>
<td>Sub-System IP Integration</td>
<td></td>
</tr>
<tr>
<td>IP Integration</td>
<td></td>
</tr>
<tr>
<td>IP Verification</td>
<td></td>
</tr>
</tbody>
</table>

- Complexity
- Data Size
- Compile
- Runtime
- Debug

Increasing Complexity Data Size Compile Runtime Debug
What is Data-Driven Verification?

- **Use-case-based**
  - Define legal operations
  - Workload matters: must represent real operation

- **Data Collection**
  - Non-intrusive data collection
  - Use the right execution platform

- **Analysis**
  - Correlate, filter, learn, predict
  - Anomaly Detection

- **Goal-based**
  - Verification throughput
  - Smarter bug hunting
VERIFICATION THROUGHPUT

Smart Bug Hunting
- VIP
- Automatic Test Gen
- Formal & Lint
- Coverage & Metrics
- Debug

Multi-Level Abstraction
- Software Level
- RTL Level
- Gate Level
- Transistor Level

Raw Performance
- Bare Metal Compute
- Scalable Architecture
- Optimization Algorithms

Cycles per $ per day

Bugs per $ per day

Data-Driven
Bug detection still not as early as possible

Feedback post silicon data to improve verification
PORTABLE STIMULUS:
USE-CASE-BASED VERIFICATION
Portable Stimuli Standard (PSS)

- Behavioral standard language to express scenarios
  - Parallelism with fork and join
  - Control flow with loops, conditionals
  - Data path via memory buffers and streams
- Powerful built-in system-specific semantic for
  - Resource availability and distribution
  - Configuration, and operation modes
- Codified in two equally powerful input formats:
  - C++ library – appeals to C++ users
  - PSS – a Domain Specific Language (DSL) – easier to read and better error messages
- Standard is defined by PSWG in Accellera
Capturing Legal Behaviors

Capture test intent, analyze legal paths, generate tests randomizing options

The producer can produce a buffer size smaller than 15

The consumer can consume buffer of size bigger than 10 bytes

System assumptions – IPs can be programmed to communicate if they connected to the same memory, and no restriction prevents that communication (e.g. data size or data kind mismatch)

Question: What will be a proper data size to enable communication between them?

Answer: 11..14
Capturing Legality Rules

Capture test intent, analyze legal paths, generate tests randomizing options

buffer data_buff_s {
    rand uint[1..20] size;
    rand uint data;
};

component producer_c {
    action produce {
        output data_buff_s buf;
        constraint buf.size < 15;
    }
};

compone st consumer_c {
    action consume {
        input data_buff_s buf;
        constraint buf.size > 10;
    }
};

PSS allows capturing the dependencies in a special flow-object struct

This association of activity to legality rule is a revolution!

A constraint solver solved the scenarios to achieve a legal programming

Each sub-system model captures it’s own dependencies according to it’s specifications
Capturing Legality Rules

Capture test intent, analyze legal paths, generate tests randomizing options

```plaintext
buffer data_buff_s {
    rand uint[1..20] size;
    rand uint data;
};
```

Resource pool

Power-domain A

Power-domain B

Consume in all legal configuration and operation modes to automatically fill your coverage goals!
Generating Test Scenarios

- Generating Test Scenarios
  - Legal peripherals and DMA channels were assigned
  - DMA actions were added to serve the read and write requests
  - Legal peripherals and DMA channels were assigned
  - Initialization action was added
  - As many as desired solutions with different timing can be randomized to serve the same original request
  - Concrete solution #1
  - Concrete solution #2
Test can be Generated to Run on Any Platform

My first test: load the memory with data and use the DMA to copy it to a different location

```c
// my first test
int main_core3()
{
    tb_initial_mem(0x5000, my_data);
    signal_core(1);
    done(1);
}

int main_core1()
{
    wait_for_core(1);
    dma_program(2, 0x5000, 0x700, 20);
    dma_start(2);
    dma_wait_for_done(2);
    done(1);
}
```

User firmware code

- Tool generated code
- Synchronizations, loops, fork and joins, all are done by the PSS tool

This might be a UVM virtual sequence creating the same test
PSS Impact on Stimulus

Existing stimulus

- Post-silicon:
  - Generally OS based tests. Long test consume valuable recourses.
  - Longer debug time.
  - Failures are difficult to bring to emulation or simulation for debug.

- Pre-silicon SoC:
  - Simple directed feature tests.
  - Difficult to manually create complex scenarios.
  - Long run time for complex scenarios.

- Pre-silicon IP:
  - Excellent UVM based constrained random testbench.
  - IP initialization sequences not easily portable to FW or post-silicon.
  - IP level tests lack system context.

Stimulus with PSS

- Post-silicon:
  - Smaller deterministic bare-metal tests.
  - Compose complex scenarios.
  - Easily bring debug to Emulation.
  - Generate large set of tests for regression.

- Pre-silicon SoC:
  - Describe test intent with PSS.
  - Automation helps with complex scenario composition.
  - Reuse tests post-silicon.

- Pre-silicon IP:
  - Reuse SoC scenarios.
  - Export initialization sequences to firmware and post-silicon.
  - Export IP specific scenarios to SoC.

Excerpt from AMD DVCON presentation
Renesas Performance Verification with Pespec Generated Use-cases

- Leading industrial and automotive MCUs
  - Number of integrated IPs is increasing
  - Switched interconnect
  - Configuration has big impact on performance

- Interconnect Workbench performance analysis
  - Early performance characterization
  - Interconnect tuning to optimize performance
  - Use case performance validation

- Palladium Z1 with Perspec use cases
  - Bring-up the entire design and software
  - Perspec generating use case tests
  - Reduce from 50 hour simulations to 12 minutes
Why PSS is Great for Data-Driven?

• Captures the verification flows
  – Allows focusing on intent
  – Abstracts away implementation details

• Automated traffic to close the coverage gap
  – UVM gives a fresh stream per seed but virtual sequences are highly directed
  – Accomplishing a goal may require coming up with different timing or test topology
  – The power of the PSS random schedule capability

• PSS captures the legality rules
  – “Don’t move the furniture and don’t clean the dog”

• Portability
  – Coverage filling task may cross platform borders
    • May not have enough cycles to be closed in a single platform

• PSS solve the entire scenario in one time
  – Can leverage data before running the simulation
Traditional MDV Flow

Coverage can be measured only after simulation is done. Manual work to achieve 100% coverage or discovering a value to be unreachable.

Need to re-do the work in case of TB changes or a different simulator release.
Data-Driven with vManager and Perspec

Implications:
- Reduced number of machines and farm size
- Less human efforts for coverage review and test creation
- Shorter cycles to meet coverage goals and project deadlines

Noticeably, Perspec adds a step of automatic test generation.

But there is much more that is added on all MDV steps!

New test topologies are automatically produced. Coverage maximization is done before running the regression per user criteria. vManager decides if a test is contributing for a plan or redundant.
Data Driven with Perspec and vManager

- Coverage on use-cases
- Reachability analysis and debug

- Runtime collection from all platforms
- Combine HW and SW coverage items

- Abstract debug and use case review

- Progress management in terms of meaningful test properties

- Use case coverage on the abstracted model
- Easy to implement
- Direct mapping to high level plan
- Contains both SW and HW
- Can be exported to all platforms

- Extra step of test generation and gen-time coverage review
- Use case visualization to approve the generated scenario

- Regression level maximization
Perspec and vManager Revolution

• Builds on the vManager flows capability
  – Allows running a multi-steps session
  – Steps can run in parallel to each other (e.g. start launching tests as soon as they are ready)

• Simplified integration scripts using the following user-defined scripts
  – ps_gen_script – a script for generating a full perspec regression
  – ps_exec_script – a script for running a single test
  – config file – lists step names, top-directories and the two scripts above

• Enhanced regression control with test tables
  – The tests to be generated and executed are coming from Perspec test tables (and not VSIF)
    • Include multiple top actions, constraint settings, counts and fill capabilities for each
  – The execution scripts are the gen and exec scripts provided above

• The MDV flow does not force usage of test tables
  – Users can use home grown scripts
  – More automation can be provided on top of test tables
Resolving vManager/Perspec terminology review

<table>
<thead>
<tr>
<th>Perspec</th>
<th>Vmanager</th>
<th>Integrated terminology</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario specification</td>
<td>NA</td>
<td>Scenario specification</td>
<td>Pure intent, partial specification</td>
</tr>
<tr>
<td>Scenario instance</td>
<td>NA</td>
<td>Scenario instance</td>
<td>Fully statically solved scenario</td>
</tr>
<tr>
<td>test</td>
<td>test</td>
<td>test</td>
<td>Code representation of scenario instance</td>
</tr>
<tr>
<td>NA</td>
<td>run</td>
<td>run</td>
<td>Test execution with a seed</td>
</tr>
</tbody>
</table>

Top partial descriptions, fill per command

Test Table

Test Table contains: Top actions, constraints, count/fill

Scenario specification

Perspec solver

Gen-time seed

Instance

Platform1 Test

Platform2 Test

Run-time seed
Perspec-vManager Solution

Perspec flow consists of two steps: Test generation & test execution with ability to analyze each.

Once a test is generated it is added to the execution session.
Perspec-vManager Solution

Generation Steps

- Regression recipe
- Debug contradictions
- View the UML of the entire regression
- Test generation results (by top action)

Upon generation the scenario contributed coverage is evaluated against the verification plan or specific plan section.
Perspec-vManager Solution

Regression recipe
(test table)

Execution Step

Debug execution with
waveform, smart log, and
activity diagram

View the UML of
the entire regression

Link to the UML
diagram of the test.
Reflects the execution progress

Execution runs

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>Duration (sec.)</th>
<th>Top Files</th>
<th>Start Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>/cdn uart apbe tests/data_poll_vir_seq</td>
<td>passed</td>
<td>18</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:22 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/data_poll_vir_seq</td>
<td>passed</td>
<td>18</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:21 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/test_uart</td>
<td>failed</td>
<td>17</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:21 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/test_uart</td>
<td>failed</td>
<td>19</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:21 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/fill_tx_buffer</td>
<td>passed</td>
<td>18</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:20 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/fill_tx_buffer</td>
<td>passed</td>
<td>25</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:20 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/fill_tx_buffer</td>
<td>passed</td>
<td>22</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:19 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/data_poll</td>
<td>passed</td>
<td>22</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:19 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/data_poll</td>
<td>passed</td>
<td>24</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:18 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/data_poll</td>
<td>passed</td>
<td>23</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:17 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/data_poll</td>
<td>passed</td>
<td>18</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:17 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/data_poll</td>
<td>passed</td>
<td>20</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:17 PM</td>
</tr>
<tr>
<td>/cdn uart apbe tests/data_poll</td>
<td>passed</td>
<td>19</td>
<td>/grid/avs/install/indisw10.2/lates...</td>
<td>1/4/11 11:16 PM</td>
</tr>
</tbody>
</table>
PSS and Data Driven

• Use-case capture essential for Data driven flows
  – Capturing information and legality rules
  – Revolution in test generation automation
  – Can be applied to any execution platform
• Capabilities exist today
  – Used by multiple users world-wide for both sub and full systems
  – Applications include workloads for performance, power, and coherency testing
• Can feed a data-driven cognitive machine for further analysis

Thank you!
Chris Komar, Product Engineering Group Director, Cadence

DATA-DRIVEN FORMAL VERIFICATION
Data to Drive...

Use-case-based
- Define legal operations
- Workload matters: must represent real operation

Data Collection
- Non-intrusive data collection
- Use the right execution platform

Analysis
- Correlate, filter, learn, predict
- Anomaly detection

Goal-based
- Verification throughput
- Smarter bug hunting

Optimized runtime, results and resources
Formal signoff of an IP

Signoff Metrics

0%
25%
50%
75%
100%

time
Data to Drive...

Signoff Metrics

Formal signoff of an IP
Ever-Increasing Amount of Formal Coverage Data

• Formal Coverage models and types continue to expand

Design Coverage App

JasperGold® FPV w/Visualize™

Code Coverage
• Branch
• Statement
• Expression
• Toggle

Functional Coverage
• Property (SVA/PSL)
• Covergroup

Large number of new covers generated

Coverage Database
Formal-specific Coverage Types

**Stimuli Coverage**
- **Formal Setup**
- **DUT**

How restrictive is the design behavior under the formal setup? Is the design over-constrained?

**Proof Coverage**
- **Formal Setup**
- **Proven Properties (Proof Core)**

What coverage is achieved by the proven properties?

**Cone-of-Influence (COI) Coverage**
- **Formal Setup**
- **Properties (Structural COI analysis)**

How complete is my property set? Do I cover all design behaviors?
COI / Proof Core Coverage

Cone-Of-Influence Measurement
- Design fan-in is computed starting from assertion, traversing back to inputs
- The union of COIs from all assertions is reported
- Anything outside the COI region cannot influence assertion status
- Anything inside the COI region may influence assertion status
- Fast measurement – no formal engines are run

Proof Core Measurement
- The union of proof cores from all asserts is reported
- Anything outside the Proof Core region cannot influence assertion status
- Anything inside the Proof Core region may influence assertion status
- Slower measurement than COI – requires running formal engines

From COV App Rapid Adoption Kit on http://support.cadence.com
Formal-specific Coverage Types

**Stimuli Coverage**

**Formal Setup**

How restrictive is the design behavior under the formal setup?
Is the design over-constrained?

**DUT**

**Cone-of-Influence (COI) Coverage**

**Formal Setup**

How complete is my property set?
Do I cover all design behaviors?

**Properties** (Structural COI analysis)

**DUT**

**Proof Coverage**

**Formal Setup**

What coverage is achieved by the proven properties?

**DUT**

**Proven Properties** (Proof Core)

**Bounded Proof Coverage**

**Formal Setup**

What coverage is achieved by bounded proofs?
Is the bound enough? How to do better?

**DUT**

**Bounded Proof Analysis**

?
Multi-Dimensional Coverage Data

- Coverage data is multiplied by the unique coverage types offered by formal

<table>
<thead>
<tr>
<th>Coverage Types</th>
<th>Branch</th>
<th>Statement</th>
<th>Expression</th>
<th>Toggle</th>
<th>Property</th>
<th>Covergroup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachability</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deadcode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>COI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Proof Core</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bound</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

How to make sense of the data?
1) Abstract to more meaningful metrics
2) Provide an intuitive GUI to analyze results
3) Intelligent exclusions
# Meaningful Metrics

## Coverage Models

<table>
<thead>
<tr>
<th>Coverage Type</th>
<th>Branch</th>
<th>Statement</th>
<th>Expression</th>
<th>Toggle</th>
<th>Property</th>
<th>Covergroup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachability</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deadcode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>COI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proof Core</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bound</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Stimuli Coverage
- Stimulus exists that explores all code

### Checker Coverage
- Sufficient assertions exist that checks all code

### Bound Analysis
- In the case of an undetermined property, what is/is not covered

**Signoff?**
### Meaningful Metrics

#### Coverage Models

<table>
<thead>
<tr>
<th></th>
<th>Branch</th>
<th>Statement</th>
<th>Expression</th>
<th>Toggle</th>
<th>Property</th>
<th>Covergroup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachability</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deadcode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>COI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Proof Core</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Bound</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **Stimuli Coverage**
- **Checker Coverage**
- **Bound Analysis**

#### Coverage Types

- **Formal Coverage**
- **Design and Verification**
- **United States**

**Code cover item can be exercised by the environment/inputs AND Has been checked by the assertions**

**Signoff?**
Intuitive GUI

Formal Coverage
Stimuli Coverage
Checker Coverage
Intuitive Analysis

- Top-down navigation
  - Summary views reflect the progress of bug-hunting or signoff efforts
  - Quickly analyze the source of remaining gaps
Intelligent Exclusions Save Effort

• Auto-exclude certain covers to reduce noise
  – Reset-related unreachable covers
  – Constant-related unreachable covers
  – Deadcode

• Advanced Waiver Capability
  – Persistent waivers tolerant of design changes
    • Avoids re-analyzing previously waived items
  – Waive-multiple by expression greatly reduces the number of user actions
Data to Drive...

- Optimized runtime, results, and resources
- Formal signoff of an IP
Problem Statement

3 CPUs x 10 mins = 30 mins
3 CPUs x 20 mins = 60 mins
Total CPU time = 90 mins

Run 2: 90 mins
Run 3: 90 mins
...

Resource/Engine A
Resource/Engine B
Resource/Engine C

P1
Determined after 10 mins

Resource/Engine A
Resource/Engine B
Resource/Engine C

P2
Determined after 20 min

Time/Engine Depth
Can we use knowledge from previous runs to minimize wasted cycles?

Resource/Engine A

Resource/Engine B

Resource/Engine C

Resource/Engine A

Resource/Engine B

Resource/Engine C

Problem Statement

Time/Engine Depth

P1
Determined after 10 mins

P2
Determined after 20 min

3 CPUs x 10 mins = 30 mins
3 CPUs x 20 mins = 60 mins
Total CPU time = 90 mins

Run 2: 90 mins
Run 3: 90 mins
...

42
Simple Solutions

Need ability to learn from previous runs, to optimize subsequent proofs and smartly react to changes introduced to the design/environment.
Challenge

Use knowledge from previous run as a hint for subsequent run

<table>
<thead>
<tr>
<th>Resource/Engine</th>
<th>Time/Engine Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>P1 Determined after 10 mins</td>
</tr>
<tr>
<td>C</td>
<td>P2 Determined after 20 mins</td>
</tr>
</tbody>
</table>

1 CPUs x 10 mins = 10 mins
1 CPUs x 20 mins = 20 mins
Total CPU time = 30 mins

Run 2: 30 mins
Run 3: 30 mins
...

1 CPUs x 10 mins = 10 mins
1 CPUs x 20 mins = 20 mins
Total CPU time = 30 mins

Run 2: 30 mins
Run 3: 30 mins
...
Property Packer/Proof Flow

Proof Flow

- Target best engine inferred to properties, to save up resources
- Explore properties with new engines, using resources freed up by packer

Cache unmatched properties
Adaptive Regression (Cache “Miss”)

Design/Env → Existing Metadata for Design/Run?

Yes → Infer best engine X and time T for each property P

No → Optimize property packing and exploration

JasperGold Information Server (JGIS) → PROOF
Adaptive Regression Example

• Learn best configuration on future runs, optimize continuously according to outcome

<table>
<thead>
<tr>
<th>Prop</th>
<th>Status</th>
<th>Engine</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>determined</td>
<td>A,B,C...</td>
<td>35</td>
</tr>
<tr>
<td>p2</td>
<td>determined</td>
<td>A,B,C...</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>undetermined</td>
<td>A,B,C...</td>
<td>60</td>
</tr>
<tr>
<td>p4</td>
<td>determined</td>
<td>A,B,C...</td>
<td>30</td>
</tr>
<tr>
<td>p5</td>
<td>determined</td>
<td>A,B,C...</td>
<td>10</td>
</tr>
<tr>
<td>p6</td>
<td>undetermined</td>
<td>A,B,C...</td>
<td>60</td>
</tr>
</tbody>
</table>

Run X

<table>
<thead>
<tr>
<th>Prop</th>
<th>Status</th>
<th>AR inferred engine</th>
<th>Time</th>
<th>Parallel exploration with other engines</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>determined</td>
<td>C</td>
<td>50</td>
<td>A,B,D,E…</td>
<td>60</td>
</tr>
<tr>
<td>p2</td>
<td>determined</td>
<td>C</td>
<td>15</td>
<td>A,B,D,E…</td>
<td>15</td>
</tr>
<tr>
<td>p3</td>
<td>determined</td>
<td>B</td>
<td>20</td>
<td>A,C,D,E…</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>determined</td>
<td>B</td>
<td>30</td>
<td>A,C,D,E…</td>
<td>30</td>
</tr>
<tr>
<td>p5</td>
<td>determined</td>
<td>A</td>
<td>10</td>
<td>B,C,D,E…</td>
<td>10</td>
</tr>
<tr>
<td>p6</td>
<td>undetermined</td>
<td>A</td>
<td>60</td>
<td>B,C,D,E…</td>
<td>60</td>
</tr>
</tbody>
</table>

Run Y

Speedup: select best engine and proof time per property based on previous runs
Convergence: use saved up time to explore properties with additional engines

Design changes
Results

Efficiency = \frac{\text{CPU Time for Winning Engines}}{\text{Total CPU Time}}

6x computational efficiency improvement
Data-driven Formal Verification Summary

• Data to enable

  – User productivity
    • Analyze issues, measure formal verification progress/signoff when complemented with JasperGold COV GUI

  – Tool efficiency
    • Improve throughput and overall verification productivity with smart ML-based regression capability
DATA-DRIVEN EMULATION
Data-Driven Emulation with Palladium

- **Why emulate?**
  - Palladium enables users to verify and test with *directed, pseudo-random, random, lab-based, real-case scenarios* that are typically not practical with other verification platforms especially during heavy HW/SW integration and co-debugging stages.

- **Emulation trends**
  - Scalable models: IP to billion-class design
  - Ease of migration: simulation, prototype, etc.
  - Multi-chip and benchmark

---

- **Use-case-based**
  - Define legal operations
  - Workload matters: must represent real operation

- **Data Collection**
  - Non-intrusive data collection
  - Use the right execution platform

- **Analysis**
  - Correlate, filter, learn, predict
  - Anomaly detection

- **Goal-based**
  - Verification throughput
  - Smarter bug hunting
Bug detection still not as early as possible

- Simulation catches most IP-level bugs
- Acceleration / Emulation catches most SoC-level bugs
- Emulation / Prototyping catches most HW/SW level bugs
- Production / Live System Test catches most customer level bugs

Shift-left strategy is consuming the attention of many leading-edge companies

Reduce risk & cost
Find customer-level bugs as early as possible in the development phase
Customers Need the Fastest Engines

• **Ever-increasing verification** requirements driven by growing hardware and **software complexity**

• **Fast time to results** is essential to ensure projects can **meet schedules**

• **Right tools for the right job**: Combination of formal, simulation, emulation, and FPGA prototyping
Cadence Verification solution

- **Cadence Verification Suite**
  - JasperGold®
    - FORMAL & STATIC
  - Xcelium™
    - SIMULATION
  - Palladium® Z1
    - EMULATION
  - Protium™ S1
    - FPGA PROTOTYPE

- **VIP**
  - VERIFICATION IP
  - MEMORY
  - PROTOCOLS

- **vManager™**
  - MULTI-ENGINE
  - COVERAGE

- **Indago™**
  - DESIGN & TESTBENCH
  - DEBUG

- **Perspec™**
  - SOC TEST
  - GENERATION

**Advanced Flows**
- MIXED SIGNAL
- LOW POWER
- FUNCTIONAL SAFETY

- **CLOUD ENABLED**
  - ENABLED
  - ENABLED
  - ENABLED
Verification Acceleration
Congruency between core engines

Xcelium-Palladium congruency
- Hybrid: Accelerate software bring-up
- UVM acceleration / hot swap
- Software driven verification and debug
Platform Congruency: Game Changer
Reducing bring-up time with Multi-fabric Compiler

Palladium-Protium congruency
- Common Front-end
- Multi-fabric Compiler
- Combination enables debug and speed
Palladium Z1: core value proposition
Bridging the Productivity GAP

Palladium Z1 finds HW/SW bugs while enabling early system-level integration & validation.

Traditional Flow

- Block
- Chip
- Prototype
- Silicon lab test
- Field test
- ROM
- Content
- Diagnostics & Firmware
- Drivers / RTOS / Applications

Power-aware verification & analysis
- System bugs
- HW/SW bugs

Time to market advantage of 2 to 4 months

"Easy" bugs
Bugs that take many cycles to be uncovered

Analog / RF bugs

HW/SW Spec

Sim-Acc

Emulation

Prototype / Si lab test

Field test

# of bugs in the design
Debug with Palladium
Using FullVision (FV)

- Specify time window & capture up to 2M samples (typical)
- Trigger at points of interest
- Full signal depth captured

100% signal visibility

Software

SW driven HW verification

Space

100%

Time

2M cycles

2M cycles

2M cycles

2M cycles

SoC Interconnect Fabric

CPU Subsystem

Application Specific Components

3D GFX

DSP

A/V

High speed, wired interface peripherals

DDR3

PHY

Other peripherals

SATA

MIPI

HDMI

WLAN

LTE

Low speed peripheral subsystem

PMU

MIPI

JTAG

INTC

I2C

SPI

Timer

GPIO

Display

UART

Apps

Accel

Modem

C1

L2 cache

C2

L2 cache

Cache Coherent Fabric

PHY

USB3.0

USB2.0

USB1.1

PCIe

Gen 3

PCIe

Gen 2

Ethernet

PHY

C1

C2

L2 cache

C1

L2 cache

L2 cache

C2

C2

PHY

Software

SW driven HW verification

Space

100%

signal visibility

Time

2M cycles

2M cycles
Debug with Palladium
Using Dynamic Probes (DYNP)

- Specify time window & capture up to 80M samples
- Vary sample size & probe depth
- Dynamically (at run time) choose the signals to capture
- Recompile design to change depth versus width

**Software**
SW driven HW verification

**Application Specific Components**
- 3D GFX
- DSP
- Apps Accel
- Modem

**SoC Interconnect Fabric**
- CPU Subsystem
- L2 cache
- L3 cache
- Cache Coherent Fabric

**High speed, wired interface peripherals**
- DDR3
- PHY
- USB3.0
- 3.0 PHY
- 2.0 PHY
- PCIe
- Gen 2,3 PHY
- Ethernet
- PCIe

**Software**
- SW driven
- HW verification

**Tradeoff**
- Depth vs Width at Compile Time
- Which signals are captured can change dynamically

**FullVision**
- 80M samples
- 768 probes per domain
- 2304 probes per domain
- 27M samples
- 80M samples
- 768 probes per domain
During the Prepare session
- Use all the normal commands for the run
- Snapshots captured at user specified intervals automatically
- Primary inputs / memory outputs are continuously captured
- Support included by default, just user enabled at run time
- Use in either Fullvision or Dynamic Probes mode
- Supported in all modes except with dynamic targets

Example: 70M cycles between snapshots
Debug with Palladium
Using Infinitrace – Observe (Replay)

- Jump to time window of interest using a specific time or a trigger
- Move forward and backward in time to capture window of interest
- Targets and testbench not used during the Observe session, just their recorded inputs are needed
State Description Language (SDL) - Intro

• SDL is the language you use to define a Trigger State machine. It has all the capabilities of commercial logic analyzers – plus more.

• When user-defined logic conditions are met, logic analyzer will “trigger”
  – In all modes, stop collection of trace data
  – In all modes except Logic Analyzer (LA) mode, stop the running design
    • In LA mode, trace data collection stops but design keeps running

• Trigger is like a simulation breakpoint
  – But can be more powerful, because triggering can be determined by a state machine that you define during debug

• Trigger state machine can be changed dynamically during a debug session, all signals available to SDL without recompiling the design
SDL – Basic Properties

• SDL tracks sequences of events by monitoring design objects such as signals, assertions, CPF/UPF objects using a state machine description
• Multiple instances of SDL can be used to track multiple independent sequences of events
• Each SDL instance has its own hardware resources:
  – One state machine
  – Expression evaluators (can be used inside state machines, or independently)
  – 2 general purpose counters (for counting events)
• Each SDL instance can perform, on a cycle by cycle basis, any of the following actions:
  – ACQUIRE: decide whether an individual probe sample should be acquired or rejected
  – TRIGGER: stop design clocks and/or waveform acquisition (depends on settings)
  – EXEC: Execute a TCL/XEL command/proc
  – DISPLAY: print out a formatted message, including time and signal values
  – Control internal SDL resources (go to a different state, increment/decrement/load counters, etc.)
SDL – Execution Model

• At the beginning of the run we are in the first state of the SDL program
• At each FCLK, SDL program can only be in one state
• If in a certain FCLK we are in state S1 and we execute “Goto S2”, then in the next FCLK we will be in state S2
• At each FCLK,
  – First, all signals in the design are updated
  – Then, all the tests in the SDL for current state are evaluated concurrently
  – Then, (depending on the test results) 0 or more actions are executed concurrently

```
State s1
{
    if ( resetn == 'b0 ) {
        goto s2;
    }
}
State s2
{
    if ( resetn == 'b1 ) {
        Trigger;
    }
}
```

```
<table>
<thead>
<tr>
<th>FCLK #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>resetn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Trigger marker
```
State $s_1$
{
    if ( $A == 'b1$ ) {
        load counter1 4;
        Goto s2;
    }
}
State $s_2$
{
    if ( $A == 'b0$ ) {
        goto s1;
    } else if (counter1<=0) {
        trigger;
    } else {
        decrement counter1;
    }
}

Trigger the first time signal $A$ remains high for at least 5 consecutive FCLK cycles.
Dynamic RTL – DRTL
Alternative and Complement to SDL

• New runtime monitor functionality
  – Constructed using standard Verilog/VHDL RTL design
  – Loaded and instantiated at runtime. Fully dynamic and independent of compile.
  – Can monitor, display, trigger and provide runtime control

• Advantages of DRTL
  – Code complex monitors with state machines in a standard RTL language (Verilog or VHDL)
  – Able to Save and Load DRTL from precompiled files
    • This allows the creation of standard libraries of DRTL monitors
  – Flexible, single module can be instantiated multiple times
    • User only needs to instantiate the DRTL module and connect to the signals of interest

• Complements SDL
  – Easier to write complex logic and state machines
  – Optionally interacts with SDL to provide control of the runtime session
DRTL Independently Controlling and Monitoring

$display and $qel used within the DRTL module

DRTL code
- Complex state machine monitoring
- Read-in / compiled at runtime (similar to SDL)
- $display used to print monitoring messages
- $qel used for control such as triggering
module riscMon(clk, rst, data, ld, PC, OP);
    input clk, rst;
    input [8:0] data;
    input ld;
    output [5:0] PC;
    output [2:0] OP;
    ..... 
    assign PC = rPC;
    assign OP = rOP;
    always @(posedge clk or negedge rst)
    begin
      if (rst == 1'b0)
        currentState <= STATE_INIT;
      else
        currentState <= nextState;
    end
    always @( *)
    begin
      case (currentState)
        STATE_INIT: begin
          if (ld == 1'b1)
            nextState = STATE_LOAD;
          else
            currentState <= nextState;
        end
        .....
        endcase // case (currentState)
    end
endmodule

state begin
  if (RISC_PROCESSOR.rst == 1'b1)
  {
    goto Monitor;
  }
State Monitor
  if (RISC_PROCESSOR.id_reg == 1'b1)
  {
    display(" PC: %h  OP:%h ",
           monInst.PC[5:0],monInst.OP[2:0])
    goto L0;
  }
}

DRTL code
• Complex state machine monitoring
• Outputs available to SDL

SDL code
• Existing control mechanism for emulator
• Accesses outputs from DRTL state machine

Dynamic RTL Complements SDL
Output ports available to SDL
DRTL Usage Example
Monitoring a standard interface

Two Instances of the DRTL PCIe Monitor

module PCIeMon (state_signal, signal_width, reset, clk);
input [7:0] state_signal;
input reset;
input [2:0] signal_width;
reg [7:0] state_signal_prev;
reg [3:0] state, next_state;
input clk;

always@(posedge clk)
begin
state <= next_state;
state_signal_prev <= state_signal;
end

always@(*)
begin
    case(state)
        BEGIN: begin
            $display("state = BEGIN");
            next_state <= UPDATE;
        end
end

PCleEP_inst1

PCIeEP_inst2

Wrapper
Data type: coverage example with Palladium
All coverage in simulator and Palladium is scored
Data type: coverage example with Palladium

All coverage in Palladium is scored in emulation as well.
SoC Power Analysis Requires “Deep” Cycles
@100MHz for 10 secs → 1 Billion cycles

Additional cycles are needed for system-level power analysis

Deep cycles: Dynamic power profiling calculates average power over long run w/ “real” stimulus & SW interactions

Identify and analyze peak and average power at system level

Explore the ‘What if’s to avoid ‘What now’

Component-level

System-level

Sample frequency

time

Simulation Run

Indicates Block turns on/off

OFF
On
Data-driven emulation example: Power analysis

Data-driven workload can be leveraged to extract power profile: average and peak power

<table>
<thead>
<tr>
<th>Power Info</th>
<th>Inputs</th>
<th>Work model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Palladium DPA</td>
<td>Toggles</td>
<td>• Power Analysis (per hierarchy / time)</td>
</tr>
<tr>
<td></td>
<td>RTL or Gates</td>
<td>• Peak detection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Find window of interest for other tool</td>
</tr>
<tr>
<td>Joules</td>
<td>Watts</td>
<td>• RTL Power Analysis and Optimization</td>
</tr>
<tr>
<td></td>
<td>RTL or Gates</td>
<td>• Power estimation</td>
</tr>
<tr>
<td>Voltus</td>
<td>Watts</td>
<td>• Power estimation and power Integrity</td>
</tr>
<tr>
<td></td>
<td>Gates</td>
<td>• IR-drop and final Signoff</td>
</tr>
</tbody>
</table>
Summary: Data driven emulation enables system-level analysis

Palladium enterprise emulation platform excels with early HW/SW integration and co-verification with power analysis at the system-level

Palladium Series
High-performance verification platform from RTL acceleration to system emulation
POST-SILICON AND IN-LIFE ANALYTICS IN HETEROGENOUS SOCS
Problem statements

• It is not about the ISA(s)
• It is not about the core(s)
  – Compute is largely ‘solved’
• The challenge today is systemic complexity, for example:
  – Ad-hoc programming paradigms
  – Processor-processor interactions
  – HW/SW interactions
  – Interconnect, NoC & deadlock
  – System are informally architected
  – Workload details unknown in advance
  – Massive data
UltraSoC Distills Insights from Data

UltraSoC delivers actionable insights
With system-wide understanding
From rich data across the whole SoC

Objective

Actionable Analytics from any Chip for performance, safety, cyber-security
Advanced Debug/Monitoring for the Whole SoC

Interconnect (AXI, ACE, ACE-lite, OCP, NoC)

Portfolio of Analytic Modules
Flexible & Scalable Message Fabric
Family of Communicators

System Block
UltraSoC IP
Software tools for data-driven insights

Eclipse based UltraDevelop IDE

- Multiple other CPUs
- SW & HW in one tool
- Single step & breakpoint CPU code & decoded trace
- Real-time HW Data
- RISC-V instruction packets

RISC-V CPU

Script based
UltraSoC

• A coherent architecture to debug, monitor and provide rich data for run-time analytics
  – RTL IP is highly parameterizable - allows customers to trade hardware resources and thus silicon area
  – Hardware resources are configurable at runtime
  – Allows reuse of hardware resources for different scenarios and different algorithms
  – Help with security and safety of systems
  – Hardware provides rich data so CPU load for analysis is small
Analytics throughout
Simulation ➔ Emulation ➔ In-Life

- Simulation
- Emulation
- Prototype
- Lab test
- Field trial
- In Life

Tape-out

GA

CPU and other IP

- HW/SW bring-up, Initial system release
- Post-processing in software or Real-time processing in hardware

Xcelium™ Simulation
Palladium® Z1 Emulation
Protium™ S1 FPGA Prototype

ultrasoc

etc...
In-life Detection

Safety
HW “stuck pixel” detection

• **Non-intrusive**: No performance impact
• **Hardware**: Fast, react at HW timescale; invisible to software
• **Visibility**: Analyze software and system everywhere in SoC

Security
HW-based attack detection

Performance Optimization
Run-time server SW tuning / security

- Lab test
- Field trial
- In Life
Non-intrusive stuck pixels detection

Fastest time to detection

Incoming image

Detected stuck pixels
Non intrusive anomaly detection

- Three CPU plots below show CPU cache-like traffic for 3 CPUs configured with different miss rates
- Excessive (anomalous) latencies are shown in red
Non-intrusive profiling with anomaly detection

- Traditional profilers are inadequate:
  - Sampling = miss subtle or fast events (Nyquist)
  - Performance impact/intrusive
  - “Heisenbugs”
- UltraSoC is non-intrusive
- UltraSoC is wirespeed (100% coverage)
- Analytics and automated anomaly detection to make engineer more efficient
Summary

• The challenge today is systemic complexity
  – Architectural and modelling is needed but not enough
• Data analysis critical throughout product life-cycle
  – Focused, non-intrusive data collection
• Need tools that support heterogenous systems
• Complex systems may require autonomous analytics and causality detection in real-time
Data-Driven Verification

Use-case-based
- Define legal operations
- Workload matters: must represent real operation

Data Collection
- Non-intrusive data collection
- Use the right execution platform

Analysis
- Correlate, filter, learn, predict
- Anomaly detection

Goal-based
- Verification throughput
- Smarter bug hunting
Thank You!

- Q&A