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

Bug detection rate

- IP Verification
- SoC Verification
- HW/SW Development
- System Validation & Production

© 2018 Cadence Design Systems, Inc. All rights reserved.
Key Verification Challenges to Address

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

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

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

4. When are we done?
   - Need objective project signoff criteria, metrics based tracking & trend analysis
   - Plan-driven Metrics across engines, Cloud compute, Machine Learning

Project time

© 2018 Cadence Design Systems, Inc. All rights reserved.
Verification Suite
Technology innovation leadership: Fast, Smart, and Optimized

- **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 Schirrmieister

• 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

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.

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

Catch bugs as early as possible

RTL Design
Testbench Development
RTL Refinement
IP Verification
Integration
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

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

Hobson Bullman
Vice President and General Manager
Technology Services Group, 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

Simulation

Formal
What Is A Formal App?

- Code Coverage UNReachability (UNR)
- Control & Status Register (CSR)
- Connectivity (CONN)
- Sequential Equivalency Checking (SEC)
- Formal Property Verification (FPV)

Automated/Optimized Executable Spec Entry

Debug

Engines/Proof Strategies

Formal Platform
**Results & Benefits**

<table>
<thead>
<tr>
<th>IP AAA</th>
<th>IP BBB</th>
<th>IP CCC</th>
<th>IP DDD</th>
<th>IP EEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mature IP, 50 registers</td>
<td>New IP (ongoing), 189 registers</td>
<td>New IP derivative</td>
<td>New IP derivative</td>
<td>New IP derivative</td>
</tr>
<tr>
<td>Initial setup (formal tb already in place)</td>
<td>1h Early verify start</td>
<td>2h 2 wks On-going</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTL bugs</td>
<td>2 found immediately</td>
<td>10 1 found immediately</td>
<td>1 found On-going</td>
<td></td>
</tr>
<tr>
<td>Issues in spec / IP-XACT</td>
<td>Several found</td>
<td>Several found before RTL availability</td>
<td>On-going</td>
<td>On-going</td>
</tr>
</tbody>
</table>

*Quick technology deployment: 5 IPs in 6 months*

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**

**Perspectives**

- 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!

Source: STMicroelectronics presentation at CDNLive 2017
CONN App: Success Story

- **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

Source: DSP Group presentation at CDNLive 2016
Apply The Best Techniques

Does Specialized App Exist?

 Apps

Good Block for Formal?

 FPV App

Simulation

Y N

Y N
Formal IP Signoff

• 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**
- Formal
- Semi-Formal

**Coverage Analysis**

**Repeatable Methodology**

**Metric-Based Sign-off Solution**

**For amenable blocks**
- Quality: formal >> sim coverage
- Productivity: Time to signoff << sim

No checks fail while reaching all coverage

vPlan
JUG: Coverage-Driven Formal Verification Signoff on CCIX Design

- Partnership with IP Group at Cadence

Repeateable methodology applied to create testbench

Metric-Driven Verification approach: Coverage Closure!

Coverage Test Progress

Source: Cadence IPG presentation at JUG 2017
JUG: Coverage-Driven Formal Verification Signoff on CCIX Design

• Partnership with IP Group at Cadence

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

Source: Cadence IPG presentation at JUG 2017
Maximum Speed

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

FPV/ABVIP/COV/CSR/Bug Hunting

SEC

ABVIP + VIP

CSR

UNR

Simulation

Formal

Bug Hunting
Case Study: Infineon

Choosing Formal Friendly Blocks

- Concurrent blocks
- Sequential blocks with low sequential depth
- Control blocks
- Data Transfer blocks
- Data Transform blocks with less intensive arithmetic operations
- Data Transform blocks with intensive arithmetic operations

Type of Design-block

- Type of Design-block
- Sequential blocks with high sequential depth

Type of Operation

- Data Transform blocks with intensive arithmetic operations

Size of Design-block (A vs. B)

Completeness – COV App

- Review vPlan
- Review Properties

Conclusion

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

```plaintext
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

```plaintext
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

```c
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_list[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
Example Caching an Object

```verilog
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

```plaintext
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(log n)), linear (O(n)), quadratic (O(n^2)) ...
  - Watch out for loops in loops
    - A loop is O(n)
    - A loop inside a loop is O(n^2)
    - A loop inside a loop inside a loop is O(n^3) ...
- 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

```vhdl
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

```verilog
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

```vhdl
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 $O(\log n)$
    • Everything else is $O(n)$
  - Front/back insertion
    • Associative arrays are $O(\log n)$
    • 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 $O(\log n)$
    • 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

```cpp
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
```
Data Structure Example

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

```plaintext
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 $O(n)$!

```cpp
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
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
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)
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
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

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

```verilog
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 doesn’t want is

- Behavioral code (things it can’t synthesize)
- Lots of independent timing (very few events at a given time slot)

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

```vhdl
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
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

Base Snapshot

Reset ➔ Initialize X

Config 1 Snapshot

test 1

test 2

test N

Config 2 Snapshot

Reset ➔ Initialize Y

test N+1

test N+2
Mechanics

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

```vhdl
class base_restart_test extends uvm_test;
...
  task run(uvm_phase_object phase);
  ...
  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
```

```bash
$ save(init_seq.get_type_name());
```
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

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Domains</td>
<td>Group the elements of logic hierarchy that share the same primary power supply</td>
</tr>
<tr>
<td>Supply Ports</td>
<td>Provide the supply interface to power domains and switches</td>
</tr>
<tr>
<td>Supply Nets</td>
<td>Connect supply ports</td>
</tr>
<tr>
<td>Power Switch</td>
<td>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</td>
</tr>
<tr>
<td>HDL Supply Net Control</td>
<td>UPF provides functions which enable the user to drive Supply Ports in low power simulation: supply_on, supply_off, supply_partial_on</td>
</tr>
<tr>
<td>Power Supply Network</td>
<td>Consists of supply ports, supply nets and power switches and their interconnections</td>
</tr>
<tr>
<td>LDO</td>
<td>Low-dropout regulator. DC/DC converter used for on-chip power supplies</td>
</tr>
</tbody>
</table>
## Low Power Basics

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Retention</td>
<td>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.</td>
</tr>
<tr>
<td>Isolation</td>
<td>Prevents corrupted values from propagating from shutoff power domains to power domains which are powered up.</td>
</tr>
<tr>
<td>Power Shutoff (PSO)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Isolation and State Retention are often used in Power Shutoff Domains</td>
</tr>
</tbody>
</table>
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
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)

```plaintext
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}}
```

<table>
<thead>
<tr>
<th>STATE</th>
<th>Digital Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL_ON</td>
<td>Does not cause corruption</td>
</tr>
<tr>
<td>PARTIAL_ON</td>
<td>Enable / Disable corruption through UPF command</td>
</tr>
<tr>
<td>OFF</td>
<td>Corrupt</td>
</tr>
<tr>
<td>UNDETERMINED</td>
<td>Corrupt</td>
</tr>
</tbody>
</table>

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

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

VDD_5V

TB_CORE

VREG (RNМ)

12 V -> 5 V, Programmable Load
RNМ / EE Net
Off-Chip

SPICE
RNМ
UPF
Verilog / SV
UPF Supply Net

CORE (PD_CORE)

VSS

LDO MASTER (SPICE)

2.54 V

ANALOG_TOP

PD_AON

LDO PROC (SPICE)

LDO AON (SPICE)

REFSYS (RNМ)

BANDGAP (RNМ)

POR (RNМ)

PCM_ANA (RNМ)

OSCILLATOR (Verilog)

Clock, POR Reset

VDD_AON (1.24 V)

VDD_PROC (0.93 V)

SW_PROC

PD_PROC

PROC

PSO_PROC

RETN_PROC

ISO_PROC, ISO_HBUSM

CLOCK, RESET

DIGITAL_TOP

PD_AON

HBUS[0]

HBUS[1]

HBUS[2]

HBUS[3]

HBUS[4]

HBUS[5]

SPI Interface

SRAM

JTAG

© 2018 Cadence Design Systems, Inc. All rights reserved.
Hierarchical UPF & Power / Ground Connections

$supply_on("TB_CORE.regulator.vin", 12.0);

$supply_on("TB_CORE.CORE.VSS", 0.0);

© 2018 Cadence Design Systems, Inc. All rights reserved.
connect_supply_net VDD_5V
-ports {VDD_5V}
connect_supply_net VSS
-ports {VSS}

create_hdl2upf_vct
VCTwr2upf_VDD_5V \\ -hdl_type {sv cds_rnm} \\ -table {{>=1.2 FULL_ON} \ 
{>=1.1 PARTIAL_ON} \ 
{<1.1 OFF}}

create_hdl2upf_vct
VCTwr2upf_VDD_AON \\ -hdl_type {sv cds_rnm} \\ -table {{>=0.9 FULL_ON} \ 
{>=0.7 PARTIAL_ON} \ 
{<0.7 OFF}}

create_hdl2upf_vct
VCTwr2upf_VDDPROC \\ -hdl_type {sv cds_rnm} \\ -table {{>=0.9 FULL_ON} \ 
{>=0.7 PARTIAL_ON} \ 
{<0.7 OFF}}
create_power_domain PD_PROC \
  -supply {primary SS_PS0} \ 
  -supply {default_isolation SS_PROC} \ 
  -supply {default_retention SS_PROC} \
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 ...
# Power States

<table>
<thead>
<tr>
<th>Power State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Up</td>
<td>LDO AON powers up, Power On Reset, Clock Enabled, JTAG, BUSARB, PCM_DIG on</td>
</tr>
</tbody>
</table>
| 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
### VDD_5V TRANSITIONS TO FULL_ON

<table>
<thead>
<tr>
<th>Power Up State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginning</td>
<td>VDD_5V ramps for 0V (OFF) to &lt; 4.8 V (PARTIAL_ON) (PARTIAL_ON -&gt; OFF). Waveform not shown</td>
</tr>
<tr>
<td>Middle</td>
<td>VDD_5V has multiple transitions between FULL_ON and PARTIAL_ON. Disable LDO_AON until VDD_5V is FULL_ON and BANDGAP output &gt;= 1.2 V</td>
</tr>
</tbody>
</table>

**VDD_5V not stable**
- Transitions from PARTIAL_ON to FULL_ON
- Leave LDO_AON disabled

4.8 V
### VDD_5V TRANSITIONS TO FULL_ON

<table>
<thead>
<tr>
<th>Power Up Power State</th>
<th>Description</th>
</tr>
</thead>
</table>
| End                  | VDD_5V FULL_ON and bg_out >= 1.2 V  
LDO_AON is powered up, enter LOAD SRAM Power State |
<table>
<thead>
<tr>
<th>LDO Shutdown Power State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire Range</td>
<td>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 &gt; 1.2 V</td>
</tr>
</tbody>
</table>
• Low Power MS simulation allows user to verify the operation of on-chip power supplies, clock generation, reset, and digital logic concurrently

<table>
<thead>
<tr>
<th>Issues Found</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSN Errors</td>
<td>Debugged using SimVision Power Supply Network. Usually found at start of LP verification cycle</td>
</tr>
<tr>
<td>Incorrect parameter setting on POR cell</td>
<td>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.</td>
</tr>
<tr>
<td>Incorrect registers for state retention</td>
<td>PROC / UPF developers worked together to determine correct registers for state retention strategy</td>
</tr>
<tr>
<td>Staggering isolation enable signals</td>
<td>PROC / UPF developers decided to disable HBUS access (through isolation) until PROC was running. Prevent accidental HBUS traffic</td>
</tr>
</tbody>
</table>
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
Is it Important Speeding-up the TB?

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

10x acceleration of TB gives 2x overall speedup

DUT and TB Accel ~10X!!

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

How Can PSS Solution help?

Simulation PIE portions

10x accel of DUT gives 2x overall speedup

© 2018 Cadence Design Systems, Inc. All rights reserved.
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??
Automated Hi-Speed C TestBench

**Xcelium + AVIP**

No re-elaboration

**Optimized C**

Distributed RT framework

**Optimized C**

Host

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

**User-defined tests can be created via a UML GUI**

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

**Vision**

Can accelerate the AVIP BFMs for multicore

Overall easy multicore partition for Xcelium

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

C code

Sync

C code

AVIP BUS Agent

Collector

BFM

AVIP Agent

Collector

BFM

DUT

mem

Host

Emulator + AVIPs
Automated Hi-Speed C TestBench Palladium + AVIP

High-speed and does not require re-elaboration

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

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

The same SOC tests runs efficiently with emulation

Yes! Use-case functional coverage on emulation

We have created an interface for Cadence AVIPs

© 2018 Cadence Design Systems, Inc. All rights reserved.
Automated Hi-Speed C TestBench
Palladium/Protium + AVIP

- Palladium/Protium
- AVIP
- DUT
- Agent
- Collector
- BFM
- Optimized C'
- Emulator + AVIPs
- Host
- Distributed RT framework
- Soc Suite
- Use-cases
- Generate the C code that can be cross compile for bare-metal execution
- Sync and data communication preserved between SW and TB activities

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

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

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

**Emulation**
- Highest performance
- RTL/gates, assertions, & coverage
- Supports 2-state logic

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

Compile full emulation model first in simulation for initial bring-up and debug.
Emulation Market Trends

Migration to centralized emulation farm to maximize investment

- 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

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

© 2018 Cadence Design Systems, Inc. All rights reserved.
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?
Scalability from Small to Large Payload Sizes

<table>
<thead>
<tr>
<th>IP</th>
<th>Subsystem</th>
<th>SoC / System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bare-Metal Software</td>
<td>Compute Subsystem</td>
<td>Bare Metal Software and Application SW Stack</td>
</tr>
<tr>
<td>Customer’s Application-Specific IP</td>
<td>CPU</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>L2 cache</td>
<td></td>
</tr>
<tr>
<td>Cache coherent fabric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-16M Gates</td>
<td>32-128M Gates</td>
<td>128-4096M Gates or more</td>
</tr>
</tbody>
</table>

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

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

© 2018 Cadence Design Systems, Inc. All rights reserved.
Verification Flow Example

**Stimulus**
- C-Tests

**VIP**

**AVIP**

**SoC, Sub-System or IP**
- High Speed, Wired Interface Peripherals
- General-Purpose Peripherals
- Low-Speed Peripherals

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

**JasperGold®**
- Hz Range
  - Faster than traditional sim

**Xcelium™**
- MHz
  - Bring-up: days
  - TAT: hours

**Palladium®**
- 10’s of MHz
  - Bring-up: week
  - TAT: days

**Protium™**

**vManager™**
- Metrics Tracking

**IWB**
- Performance

**Indago™**
- Debug

**Unified coverage** analysis & unreachability between JasperGold, Xcelium, Palladium

**Unified performance** analysis across Xcelium, Palladium

**Unified debug** across Xcelium, Palladium, & Protium

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

<table>
<thead>
<tr>
<th>4-6 weeks</th>
<th>4-6 weeks</th>
<th>4 weeks</th>
<th>2 weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Preparation</td>
<td>Memory Remodeling</td>
<td>Compile Synthesis</td>
<td>Automatic / Manual Multi-FPGA Partitioning</td>
</tr>
</tbody>
</table>

3 months…and more!

© 2018 Cadence Design Systems, Inc. All rights reserved.
Or Is It?
How to address the prototyping challenges

- 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)

- Networking: 81% faster
- Networking: 85% faster
- Consumer: 88% faster
- CPU: 91% faster
- Consumer: 89% faster
- Networking: 91% faster
- Mobile: 79% faster

Bring-up time (weeks)

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

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

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

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

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

Software

- Applications
- Middleware
- Operating Systems (OS)
- Drivers
- Firmware / HAL

SoC, Subsystem, or IP

- 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

- 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

Software Debug: C Code

- 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

Daughtercards and peripherals

Probes

JTAG
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

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

© 2018 Cadence Design Systems, Inc. All rights reserved.
Protium S1 Prototyping Solution
Industry’s first comprehensive, fully integrated solution
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!