Current Status and Challenges of SoC Verification for Embedded Systems Market

October 8th, 2003 in Kyushu University

Chong-Min Kyung
KAIST
Agenda

- Why Verification?
- Verification Alternatives
- Languages for System Modeling and Verification
- Verification with Progressive Refinement
- Concluding Remarks
Trend of Verification Effort in the Design

- **Verification** portion of design increases to anywhere from 50 to 80% of total development effort for the design.

Verification methodology manual, 2000-TransEDA
A recent study shows that more than half of all chips require one or more re-spins, and that in 74% of these re-spins functional errors were found (to be responsible).

With increasing chip complexity, this situation can only become worse.

Who can afford to risk $\geq 1\text{M Dollar}$ NRE(non-recurring engineering) cost?
Cost of fixing a bug/problem increases as design progresses.
- Need verification method at early design stage
‘Verification Performance Gap’; more serious than the ‘design productivity gap’

Growing gap between the demand for verification and the simulation technology offered by various options.

Verification Performance Gap

![Diagram showing simulation performance and verification complexity](image-url)
Completion Metrics; How do we know when the verification is done?

- Emotionally, or Intuitively;
  - Out of money? Exhausted?
  - Competition’s product is there.
  - Software people are happy with your hardware.
  - There have been no bugs reported for two weeks.

- We need more rigorous criteria.
Verification Challenges

- Sometimes, specification is Incomplete or Operating Environment is Open-Ended.
- Verification can never become complete if ECO’s (engineering change orders) keep knocking at your door.
- Basically you’re using Yesterday’s tool for Today’s Design.
- Verification productivity grows slower than Design productivity.
Agenda

- Why Verification?
- Verification Alternatives
  - Simulation
  - Hardware-accelerated simulation
  - Emulation
  - Prototyping
  - Formal verification
  - Semi-Formal (Dynamic Formal) verification
- Languages for System Modeling and Verification
- Verification with Progressive Refinement
- Concluding Remarks
Overview of Verification Methodologies

- **Simulation**
  - Faster speed, closer to final product

- **Hardware Accelerated Simulation**

- **Emulation**

- **Semi-formal Verification**

- **Formal Verification**

- **Prototype**

- **Basic verification tool**

  - Bigger coverage
Software Simulation

- Dynamic verification method
- Bugs are found by running the design implementation.
- Thoroughness depends on the test vector used.
- Some parts are tested repeatedly while other parts are not even tested.

```
a = 1;
#20 b = 1;
$display("status is = \%d", c);
...
```
Software Simulation

Pros

- The design size is limited only by the computing resource.
- Simulation can be started as soon as the RTL description is finished.
- Set-up cost is minimal.

Cons

- Slow (~100 cycles/sec); Speed gap between the speed of software simulation and real silicon widens. (Simulation speed = size of the circuit simulated / speed of the simulation engine)
- The designer often has no idea of what percentage of the design has been tested.
Hardware-Accelerated Simulation

- Simulation performance is improved by moving the **time-consuming part** of the design to hardware.
- Usually, the software simulation communicates with FPGA-based hardware accelerator.

![Diagram](image)

- Simulation environment
  - Testbench
  - Module 0
  - Module 1
  - Module 2

- Hardware Accelerator
  - Module 2 is synthesized & compiled into FPGAs
Hardware-Accelerated Simulation

Pros
- **Fast** (100K cycles/sec)
- **Cheaper** than hardware emulation
- Debugging is easier as the circuit structure is unchanged.
- Not an Overhead: Deployed as a step stone in the gradual refinement

Cons (Obstacles to overcome)
- **Set-up time** overhead to map RTL design into the hardware can be substantial.
- **SW-HW communication** speed can degrade the performance.
- **Debugging** of signals within the hardware can be difficult.
Hardware-Accelerated Simulation

ächt

- Challenges
  - Overall speed depends on the communication between simulator and hardware.
  - Execution time decomposition in a typical case of a PCI-based hardware accelerator;
    - SW simulator + PLI/FLI + Driver overhead : 38% → It is desirable to reduce the driver call overhead.
    - PCI overhead : 44% → Can be reduced by using DMA data transfer.

![Pie chart showing distribution of execution time]
Emulation

- Using the functionality of an imitating system to achieve the same results as the imitated system.
- Usually, the emulation hardware comprises an array of FPGA’s (or special-type processors) and interconnection scheme among them.
- About 1000 times faster than simulation.
Emulation

Pros
- **Fast** (500K cycles/sec)
- Verification on real target system.

Cons
- **Set-up time** overhead to map RTL design into hardware is very high.
- Many FPGA’s + resources for debugging → expensive
- Circuit partitioning algorithm and interconnection architecture limit the usable gate count.
Emulation

- **Challenges**
  - Efficient interconnection architecture and Hardware Mapping efficiency for Speed and Cost
  - RTL debugging facility with reasonable amount of resource
  - Efficient partitioning algorithm for any given interconnection architecture
  - Reducing development time (to take advantage of more recent FPGA’s)
Prototyping

- Special (more dedicated and customized) hardware architecture made to fit a specific application.
Prototyping

Ցանկ

Pros

- Higher (than emulation) clock rate (over 1M cycles/sec) due to specific design of prototyping board.
- Components as well as the wiring can be customized for the corresponding application.
- Can be carried along. (Hardware Emulation? Forget it!)

Cons

- Not flexible for design change
  (Every new prototype requires a new board architecture. / Even a small change requires a new PCB.)
Overview of Verification Methodologies

- **Formal verification**
  - Application of logical reasoning to the development of digital system
  - Both design and its specification are described by a language in which semantics are based on mathematical rigor.

- **Semi-formal verification**
  - Combination of simulation and formal verification.
  - Formal verification cannot fully cover large designs, where simulation can come to aid.
Formal Verification

Objective
- Check properties of model with all possible conditions

Pros
- Assures 100% coverage when applicable.
- Fast when applicable.

Cons
- Works only for small-size finite-state systems.
Formal Verification: Equivalence Checker

- Equivalence checker compares the golden model with the refined model.

- Functional representations are extracted from the designs and compared mathematically.

- Pros
  - Exhaustive design coverage
  - Very fast

- Cons
  - Memory explosion

- Tools such as LEC (Verplex), Formality (Synopsys), FormalPro (Mentor) supports Equivalence checking.
Formal Verification: Model Checking

- Model checking verifies that a design satisfies a property specified using temporal logic.

- **Computational Tree Logic**
  - Specify the temporal relationship among states in FSM with **temporal operators**;
    - A (always), E (exists) – path quantifier
    - G (global), F (future), X (next), U (until) – temporal modality
Formal Verification

- **Challenges**
  - The most critical issue of formal verification is the “state explosion” problem.
  - The application of current formal methods are limited to the design of up to 500 flip-flops.
  - Researches about complexity reductions are:
    - Reachability analysis (so that we can ignore the unreachable states)
    - Design state abstraction (to handle more gates)
    - Design decomposition (into a number of manageable sizes)
    - State projection (to reduce the dimension, and therefore the complexity without adverse effects)
Semi-Formal Verification - Assertion

 Assertion-based verification (ABV)

- “Assertion” is a statement on the intended behavior of a design.
- The purpose of assertion is to ensure consistency between the designer’s intention and the implementation.
Semi-Formal Verification - Assertion

- Simulation Quality of assertion-based verification

Diagram:
- Number of bugs found
- Time, Effort
- Setup testbench
- Describe assertions

Simulation with assertions
Efficiency of assertion

By IBM in “Computer-Aided Verification” 2000
Assertion Example - Arbiter

```
for(i=0; i<arbit_channel; i++) {
    event b_req_start[i] : posedge req[i];
    event e_req_granted[i] :
        if (b_req_start[i]) then
            (#[min_lat .. max_lat] (grants[i] || !reqs[i]));
    event e_granted_only_if_req[i]:
        if (!reqs[i]) then !grants[i];
    event e_highest_grant[i] :
        if (grants[i]) then
            (priority[i] == highest_value);
    assert(e_req_granted[i]);
    assert(e_granted_only_if_req[i]);
    assert(e_highest_grant[i]);
}
```

- Request is asserted
- After request, grant should be eventually asserted within allowed latency
- Grant is asserted only when request is asserted.
- The granted one has the highest priority.
- Set assertion on the above events.
Semi-Formal Verification - Coverage

- Coverage-directed verification
  - Increase the probability of bug detection by checking the ‘quality’ of stimulus
  - Used as a guide for the generation of input stimulus
Semi-Formal Verification - Coverage

Coverage metrics for coverage-directed verification

- Code-based metrics
  - Line/code block coverage
  - Branch/conditional coverage
  - Path coverage

- Circuit structure based metrics
  - Toggle coverage
  - Register activity

- State-space based metrics
  - Pair-arcs: usually covered by Line + condition coverage

- Spec.-based metrics
  - % of specification items satisfied
Semi-Formal Verification - Coverage

- Coverage Checking tools (company)
  - VeriCover (Veritools)
  - SureCov (Verisity)
  - Coverscan (Cadence)
  - HDLScore, VeriCov (Summit Design)
  - HDLCover, VeriSure (TransEDA)
  - Covermeter (Synopsys)
Semi-Formal Verification

❖ Pros
   ◆ Designer can measure the coverage of the test environment as the formal properties are checked during simulation.

❖ Cons
   ◆ The simulation speed is degraded as the properties are checked during simulation.

❖ Challenges
   ◆ There is no unified testbench description method.
   ◆ It is sometimes difficult to guide the direction of test vectors to increase the coverage of the design.
   ◆ Development of more efficient coverage metric to represent the behavior of the design.
Speed Comparison

Speed (Cycles/sec, log scale)

0 kHz
10 kHz
100 kHz
1 MHz
10 MHz

Software Simulation
Hardware-Accelerated Simulation (from Quickturn/Dynalith Presentation)
Hardware emulation (from Quickturn presentation)
Prototyping
Semi-formal (Assertion-based verification)

100Hz
100kHz
500KHz
1~10MHz
50-70Hz
Verification Time vs. Coverage

- Simulation
- Semi-formal
- Emulation
- Prototyping

Redirection of testbench constraints
Agenda

- Why Verification?
- Verification Alternatives
- **Languages for System Modeling and Verification**
  - System modeling languages
  - Testbench automation & Assertion languages
- Verification with Progressive Refinement
- Concluding Remarks
Accellera

- Formed in 2000 through the unification of Open Verilog International and VHDL International to focus on identifying new standards, development of standards and formats, and to foster the adoption of new methodologies.
Three different ways of specifying Assertions in Verilog designs;
- OVL (Open Verification Library)
- PSL (Property Specification Language)
- Native assertion construct in System Verilog
ACCELLERA APPROVES FOUR NEW DESIGN VERIFICATION STANDARDS

June 2, 2003 - Accellera, the electronics industry organization focused on language-based electronic design standards approved four new standards for language-based design verification; Property Specification Language (PSL) 1.01, Standard Co-Emulation Application Programming Interface (SCE-API) 1.0, SystemVerilog 3.1 and Verilog-AMS 2.1.
Accellera's PSL
(Property Specification Language)

- Gives the design architect a standard means of specifying design properties using a concise syntax with clearly defined formal semantics.
- Enables RTL implementer to capture design intent in a verifiable form, while enabling verification engineer to validate that the implementation satisfies its specification with dynamic (that is, simulation) and static (that is, formal) verification.
New languages are developed to fill the productivity gap.

- **Assembly**
  - Language for Software development
- **C**
  - Language for Hardware test
- **C++**
- **JAVA**
- **SystemC**
- **TestBuilder**
- **Vera**
- **SystemVerilog**
- **Verilog**
- **VHDL**

**Past** | **Present** | **Future**
SystemC

- SystemC is a modeling platform consisting of
  - A set of **C++ class library**, including a **simulation kernel** that supports hardware modeling concepts at the system level, behavioral level and register transfer level.

- SystemC enables us to effectively create
  - A **cycle-accurate model** of
    - Software algorithm,
    - Hardware architecture, and
    - Interfaces of System-on-a-Chip.

- Program in SystemC can be
  - An **executable specification** of the system.
SystemC

- **Modules, ports**, and **signals** \(\rightarrow\) for hierarchy
- **Processes** \(\leftarrow\) for concurrency
- **Clocks** \(\leftarrow\) for time
- **Hardware data types** \(\leftarrow\) for bit vectors, 4-valued logic, fixed-point types, arbitrary precision integers
- **Waiting** and **watching** \(\leftarrow\) for reactivity
- **Channel, interface, and event** \(\leftarrow\) for abstract communications
Abstraction Levels of SystemC

Algorithm level
Untimed functional level
Timed functional level
Bus-cycle accurate level
Cycle accurate level
Behavioral level
RT level
Gate level

Function hierarchy
Modular structure
Timing information
Intra module: un-timed
Inter module: cycle accurate
Cycle accurate
Synthesizable

C/C++
SystemC
Verilog/VHDL
What is Transaction Level Modeling?

- A modeling style where details of communication among modules are separated from the details of the implementation of the functional units or of the communication architecture.

**Diagram:**
- **Transaction Level Modeling:**
  - Block A sends `send_data` to Block B.
  - Block B responds with an acknowledgment.

- **Traditional cycle-accurate modeling:**
  - Block A sends pin signals to Block B.
  - Block B acknowledges the receipt of data `[0]` and `[31]`.
### Transaction Level Modeling

#### Comparison with other abstraction levels

- TLM does not provide pin-accurate information.
- Yet, it can be cycle-accurate.

<table>
<thead>
<tr>
<th>Abstraction level</th>
<th>Description language</th>
<th>Block definition</th>
<th>Cycle accurate</th>
<th>Pin accurate</th>
<th>Intra-module Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional level</td>
<td>C/C++</td>
<td>△</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Transaction-Level</td>
<td>SystemC</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Behavioral-Level</td>
<td>SystemC HDL</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>RTL</td>
<td>SystemC HDL</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

![Diagram showing the comparison between abstraction levels with Transaction-Level Modeling](image.png)
Test-bench automation

Why is test-bench automation required?
- Test-bench for IP can be more complex than the IP itself.
- Manual description of the test-bench is a time-consuming job.
- Simulating the whole test-bench in HDL yields excessive verification time.

Players
- TestBuilder (Cadence)
  - Closer to C, integrated to SystemC
- VERA (Synopsys)
  - Closer to Verilog, integrated to SystemVerilog
TestBuilder

- **Transaction-Based Verification**
  Functional verification in higher-level abstraction
  Engineer develops tests from a system-level perspective
  - **Advantages**
    - Enhance reusability of each component in the test-benches
    - Improve debugging and coverage analysis

![Diagram](image.png)
TVM (Transaction Verification Model)
- Translates between bus cycle command and signal waveform
- Described in C API
TVM (Transaction Verification Model)
- Translates between bus cycle command and signal waveform
- Described in Verilog PLI
Features of VERA (Synopsys):

- Functional verification language for testbench description
  - OpenVera is a language specification.
  - VERA (Synopsys) is a testbench generation tool.

**Inputs to VERA**

- dut.v
- vera.vr

**Constraints written in OpenVera syntax**
OpenVera source codes are compiled and runs with HDL simulator in which DUT is simulated.

- **DUT**
- **Vera constraints**
- **dut.v**
- **vera.vr**

**VERA**

**Inputs to VERA**

**Generated files from VERA**

- **top.v**
- **dut.shell.v**
- **dut.vro**

**Template top module**

**PLI function wrapper**

**Vera object file. (binary PLI functions or DirectC objects)**

**VCS (simulator)**

**Simulator simulates and report file is generated.**
Vera

Inputs to VERA

DUT

Vera constraints

dut.v

vera.vr

VERA

Generated files from VERA

top.v
dut.shell.v
dut.vro

OpenVera Assertions

dut.ova

VCS (simulator)
dut.ova
dut.vro
dut.shell.v
top.v
dut.v

Simulator simulates and report file is generated.

 Assertion is also supported in OpenVera.
SystemVerilog

- SystemVerilog 3.1 provides design constructs for architectural, algorithmic and transaction-based modeling.
- Adds an environment for automated testbench generation, while providing assertions to describe design functionality, including complex protocols, to drive verification using simulation or formal verification techniques.
- Its C-API provides the ability to mix Verilog and C/C++ constructs without the need for PLI for direct data exchange.
New **data types** for higher data abstraction level than Verilog
- Structures, classes, lists, etc. are supported.

**Assertion**
- Assertions can be embedded directly in Verilog RTL.
- Sequential assertion is also supported.

**Encapsulated interfaces**
- Most system bugs occur in interfaces between blocks.
- With encapsulated interfaces, the designer can concentrate on the communications rather than on the signals and wires.

**DirectC** as a fast C-API
- C codes can be called directly from the SystemVerilog codes.
Key Components of SystemVerilog

- **Verification**
  - Testbench
  - Assertions
  - DirectC

- **Design**
  - Interface
  - Enhanced Verilog
  - Concise design features
  - New Data types

- **Verilog 1364-2001**
- **Verilog 1364-1995**
## System Description Languages Summary

<table>
<thead>
<tr>
<th>Language</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>• Easy to write test vectors/environment</td>
<td>• Unable to handle some hardware environments.</td>
</tr>
<tr>
<td>HDL (Verilog, VHDL)</td>
<td>• Familiarity</td>
<td>• Focuses on the lower-level designs.</td>
</tr>
<tr>
<td></td>
<td>• Easy to describe H/W designs</td>
<td>• Improper for system modeling.</td>
</tr>
<tr>
<td>SystemC</td>
<td>• Easily connected to C/C++ codes.</td>
<td>• Limited tools (simulation, synthesis, etc.)</td>
</tr>
<tr>
<td></td>
<td>• Easy to model system behaviors.</td>
<td></td>
</tr>
<tr>
<td>SystemVerilog</td>
<td>• Easy to learn for the HDL designers.</td>
<td>• Fewer tools (simulation, synthesis.) available</td>
</tr>
<tr>
<td></td>
<td>• Easy to model system behaviors.</td>
<td></td>
</tr>
</tbody>
</table>
Agenda

- Why Verification?
- Verification Alternatives
- Languages for System Modeling and Verification
- **Verification with Progressive Refinement**
  - Flexible SoC verification environment
  - Debugging features
  - Cycle vs. transaction mode verification
  - Emulation products
- Concluding Remarks
What are the main Criteria for Good SoC Verification Environment?

- It should support various abstraction levels, and heterogeneous design languages
- Trade-off between verification speed, accuracy and debugging features is possible.
- Co-work with existing tools
- Progressive refinement
- Platform-based design
Conventional SoC Design Flow

Reference C Model

Design Exploration

C code

RTL HDL Model

C code

Gate-level Model

C code

Co-Simulation

Co-Emulation

Simulation speed is too low

Use more abstract simulation model

Applicable too late

Build SoC emulation platform ASAP
Transaction-Level Modeling

- Model the bus system in transaction-level
  - No notion of exact time.
  - But precedence relation of each functional block is properly modeled.
  - Rough estimation on performance is possible.
  - Used as the fastest reference model by each block designer
Cycle-Accurate Bus Modeling

- For more accurate modeling
  - Build a cycle-accurate system model in C or SystemC
  - Replace the transaction-level bus model with a cycle-accurate bus model
- ARM released a “Cycle-Level Interface Specification” for this abstraction level.
AMBA AHB CLI Specification

- AMBA AHB Cycle Level Interface (CLI) Specification
  - Released on July 23, 2003 by ARM.
  - CLI spec defines guidelines for TLM of AHB with SystemC.
    - Interface methods
    - Data structures
    - Header files for SystemC models
  - CLI spec leaves the detailed implementation of the AHB bus model to the reader.

- Cycle-level modeling
- Transaction-level modeling

[Diagram showing cycle-level modeling and transaction-level modeling]
Flexible SoC Verification Environment

- Build C reference model for the target application.
- Setup of platform-level verification environment as early as possible.

Algorithm

Functional block model

- HEAD
- VLD
- IDCT
- DISP

Platform-level model

- HEAD
- VLD
- IDCT
- DISP

TRS: Transactor

- SW Model
- HW Model

Verification Environment Setup
Flexible SoC Verification Environment

- **Socketize IP representation**
  - HW: C → HDL → EDIF
  - SW: native C → ISS → Processor Core
Cycle-Level Transactor

- Generate stimulus at every clock cycle
- Check the result of DUT at every clock cycle
Transaction-Level Transactor

- Only information to generate transaction is transferred to DUT, i.e., address and data
- No need to synchronize at every clock cycle
Cycle vs. Transaction-level Transactor

- Cycle-level transactor
  - Synchronized at every clock cycle.
  - Operating speed depends on the number of signals to be transferred.

- Transaction-level transactor
  - Synchronized at the end of each transaction.
  - Transactor generates all necessary signals for DUT to properly transfer the data.
  - Transactor must be designed for each interface standard
    ex) AHB transactor, SDRAM transactor, IIS transactor
Dynalith iPROVE Technology

- **PCI-based Simulation Accelerator**
  - Cycle-level verification
    - Seamless integration with the HDL testbench.
    - Up to 100K cycles/sec speed. (1000 times faster than SW simulation)

- **Transaction-level verification**
  - Up to 33M cycles/sec speed. (330K times faster than SW simulation, or 330 times faster than Cycle-Level Verification)
Agenda

- Why Verification?
- Verification Alternatives
- Languages for System Modeling and Verification
- Verification with Progressive Refinement
- Concluding Remarks
Concluding Remarks

- Verification is challenging; It needs strategy!
- The first principle of the strategy is to apply the most appropriate method for each case.
- The second principle is to verify as early as possible; Don’t wait until it grows big and kills you.
- Try to apply the cheapest and most reliable method first
  - 1st step: Apply formal methods
    - Static formal verification
    - Assertion-based verification
  - 2nd step: Simulate IP with transaction level test-bench
    - Test-bench automation tools
  - 3rd step: Emulate design
    - Emulate IP operation in FPGA
    - In-system IP verification
    - Cycle-level vs. transaction level test-bench
Concluding Remarks

- Note that main differences of SoC against old ASIC design are
  - Planned IP-reuse
  - Reuse of pre-verified platform
  - Focus on co-verification with software

- Newly added IP’s must be thoroughly verified utilizing automated testbench and formal methods, if possible.

- Well-established emulation platform helps,
  - Progressive refinement of newly added SoC components
  - Early development and verification of software

- Powerful debugging features handling both hardware part and software part are required.

- Language, Tool/Data Interfaces need standardization.

- DFV (Design for Verification) is becoming a must like Design for Reuse.
Thank You for Your Kind Attention!