IEEE International SOC Conference 2003

Current Status and Challenges of SoC Verification for Embedded Systems Market

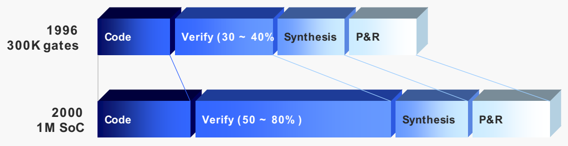
September 18th, 2003/ Portland Hilton

Chong-Min Kyung
KAIST

Agenda

Trend of Verification Effort in the Design

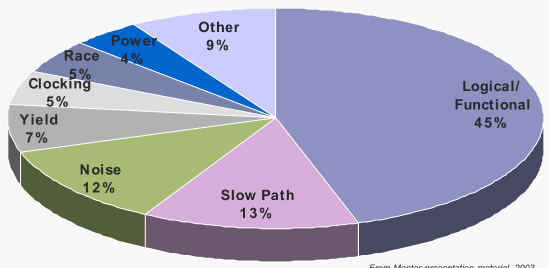
± Verification portion of design increases to anywhere from 50 to 80% of total development effort for the design.



Verification methodology manual, 2000-TransEDA

Percentage of Total Flaws

± About 50% of flaws are functional flaws.
→ Need verification method to fix logical & functional flaws



From Mentor presentation material, 2003

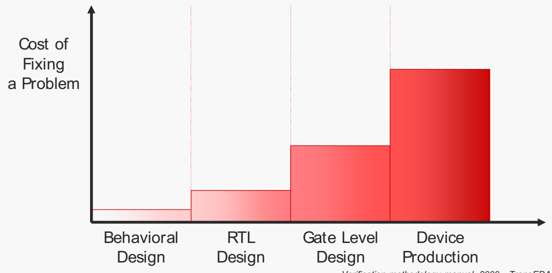
± Another recent independent study showed that more than half of all chips require one or more re-spins, and that functional errors were found in 74% of these re-spins.

± With increasing chip complexity, this situation could worsen.

± Who can afford that with >= 1M Dollar NRE cost?

Bug Fixing Cost in Time

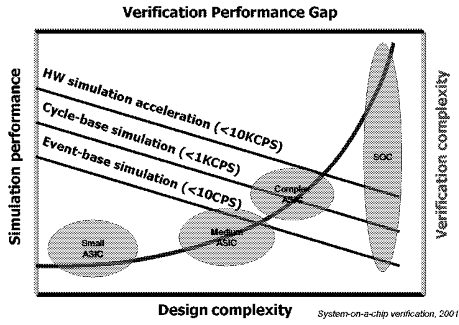
± Cost of fixing a bug/problem increases as design progresses.
→ Need verification method at early design stage



Verification methodology manual, 2000 – TransEDA

Verification Performance Gap: more serious than the design productivity gap

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



System-on-a-chip verification, 2001 – P.Rashinkar

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.

± More rigorous criteria;
→ All tests passed
→ Test Plan Coverage
→ Functional Coverage
→ Code Coverage
→ Bug Rates have flattened toward bottom.

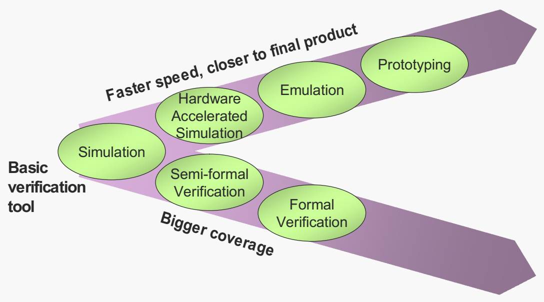
Verification Challenges

- Specification or Operating Environment is Incomplete/Open-Ended. (Verification metric is never complete like last-minute ECO.)
- The Day before Yesterday's tool for Today's Design.
- Design productivity grows faster than Verification productivity.

Agenda

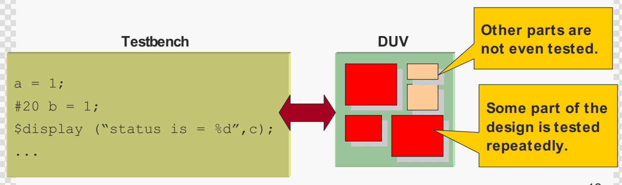
- Why Verification ?
- Verification Alternatives
  - → Simulation
  - → Hardware-accelerated simulation
  - → Emulation
  - → Prototyping
  - → Formal verification
  - → Semi-Formal (Dynamic Formal) verification
- Languages for System Modeling and Verification
- Verification with Progressive Refinement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

Overview of Verification Methodologies



Software Simulation

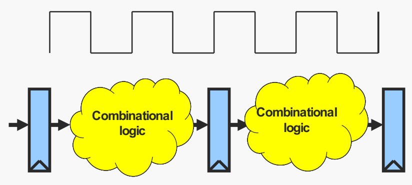
- Dynamic verification method
- Bugs are found by running the design implementation.
- Thoroughness depends on the test vector used.
- Some parts are tested repeatedly while other parts are not even tested.


Testbench
DUV
Other parts are not even tested.
Some part of the design is tested repeatedly.

```
a = 1;
#20 b = 1;
$display (“status is = %d”,c);
...
```

Cycle-Based Simulation

- Simulate the behavior of the design cycle-by-cycle.
- Cycle-accurate information is provided as a result of simulation.
- Only signals at the flip-flop input are evaluated to be stored, not internal signals of combinational logic.



Cycle-based vs. Event-driven

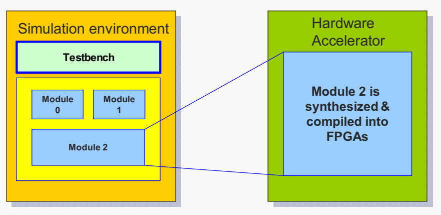
	Cycle-based	Event-driven
Timing resolution	Clock cycle	User-defined minimum delay
Evaluation time point	Rising/falling/both clock edges	At the occurrence of events
Evaluation node	Every flip-flop boundary	At the output of every logic gate on the event propagation path
Simulation time	Proportional to the (number of cycles) times (C/L size * number of F/F's)	Proportional to the number of events (circuit size* no. of cycles* event density)

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 does not exactly know how much percentage of the design have 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.

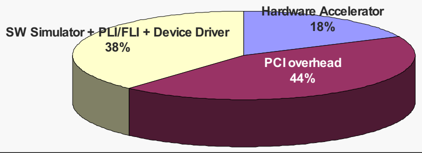


Hardware-Accelerated Simulation

- 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.

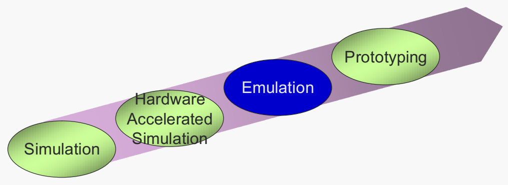
Hardware-Accelerated Simulation

- 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
    - Today (2008), SystemC has substituted PLI/FLI (Ney)



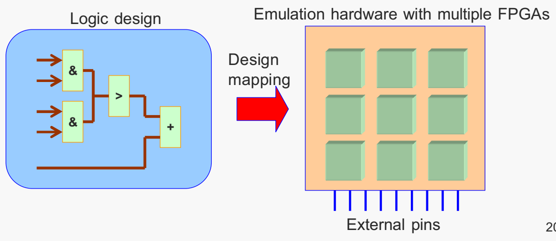
Emulation

- Emulation: Imitating the function of another system to achieve the same results as the imitated system
- Usually, the emulation hardware comprises an array of FPGAs (or special-type processors) and interconnection scheme among them
- About 1000 times faster than simulation



Emulation

- User logic design is mapped to emulation board with multiple FPGAs and/or special processors
- The emulation board has external interconnection hardware that emulates the pins of final chip.

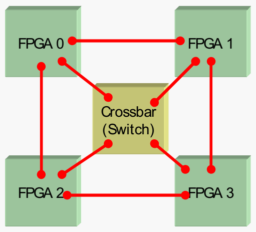


Emulation

- Pros
  - Fast (500K cycles/sec)
  - Verification on real target system
- Cons
  - Setup time overhead to map RTL design into hardware is very high
  - Many FPGAs + resources for debugging
    → high cost
  - Circuit partitioning algorithm and interconnection architecture limit the usable gate count

General Architecture of Emulation Systems

- Many FPGAs are interconnected together for large gate capacity
- Emulation systems on the market have differences in their interconnection architectures

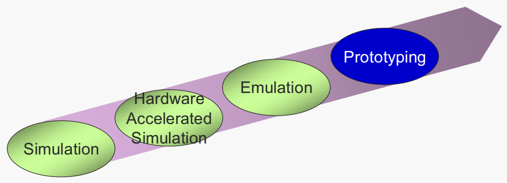


Emulation

- 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 FPGAs)

Prototyping

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



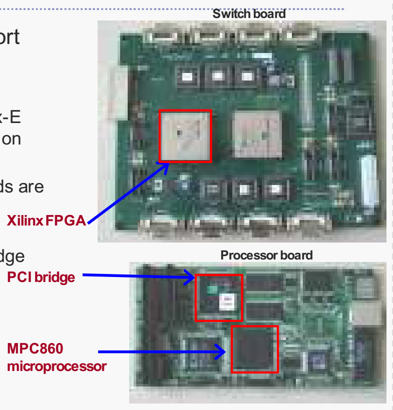
Prototyping

- 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.)

A Prototyping Example

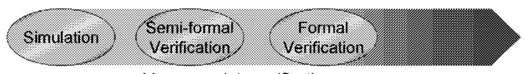
- Prototype of 4-Port Gigabit Ethernet Switch
  - → Two Xilinx Virtex-E 2000 FPGAs are on FPGA board.
  - → Four FPGA boards are used.
  - → Processor board contains PCI bridge and MPC860 microprocessor.



Courtesy of Paion, Inc.

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, and simulation can come to aid in verifying the large design



More complete verification

Formal Verification

- Objective
  - → Check properties of model with all possible conditions
- Pros
  - → Assures 100% coverage
  - → Fast
- Cons
  - → Works only for small-size finite state systems
  - → Uncomfortable due to culture difference (E.g., engineers are not familiar with the use of temporal logic used for "property" description in Model Checking)

Formal Verification: Equivalence Checker

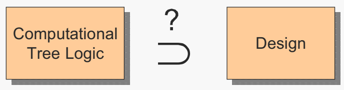
- Equivalence checker compares the golden model with the refined model.

Golden Model
?
=
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 the design satisfies a property specified using temporal logic



- Computational Tree Logic (CTL)
  - → Specify the temporal relationship among states in FSM with temporal operators:
    - → A (always/for all), E (exists/for at least one) – path quantifier
    - → G (global), F (future), X (next), U (until) – temporal modality

Formal Verification

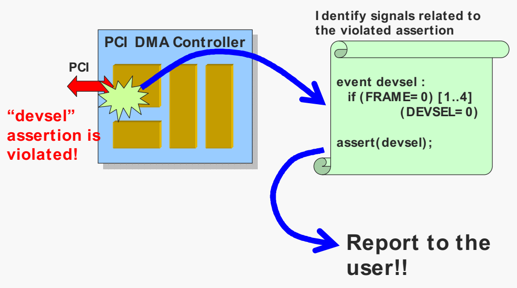
- Challenges
  - → The most critical issue of formal verification is the "state explosion" problem
  - → The application of current formal methods are limited to the design of up to 500 flip-flops
  - → Researches about complexity reductions are:
    - → Reachability analysis
    - → Design state abstraction
    - → Design decomposition
    - → State projection

Semi-Formal Verification - Assertion

- Assertion-based verification (ABV)
  - → "Assertion" is a statement on the intended behavior of a design
  - → The purpose of an assertion: to ensure consistency between the designer's intention and the implementation
- Key features of assertions
  - → *1. Error detection:* If the assertion is violated, it is detected by the simulator
  - → *2. Error isolation:* The signals related to the violated assertion are identified
  - → *3. Error notification:* The source of error is reported to the user

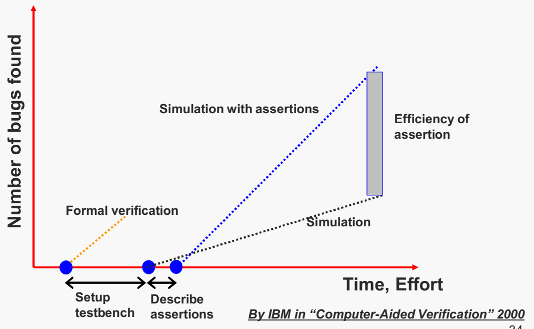
Semi-Formal Verification - Assertion

- Example of assertion-based bug detection



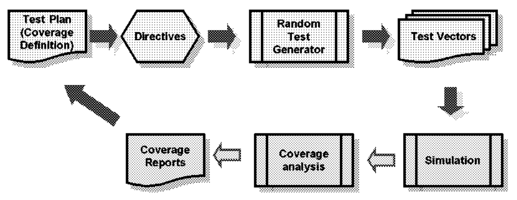
Semi-Formal Verification - Assertion

- Quality of assertion-based verification (simulation)



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

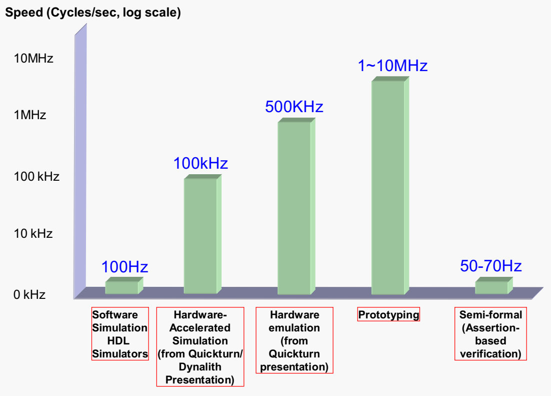
- Coverage Checking tools
  - VeriCover (Veritools)
  - SureCov (Verisity)
  - Coverscan (Cadence)
  - HDLScore, VeriCov (Summit Design)
  - HDLCover, VeriSure (TransEDA)
  - Polaris (Synopsys)
  - Covermeter (Synopsys)

  - HDL simulators (today, 2008, Ney)

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 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

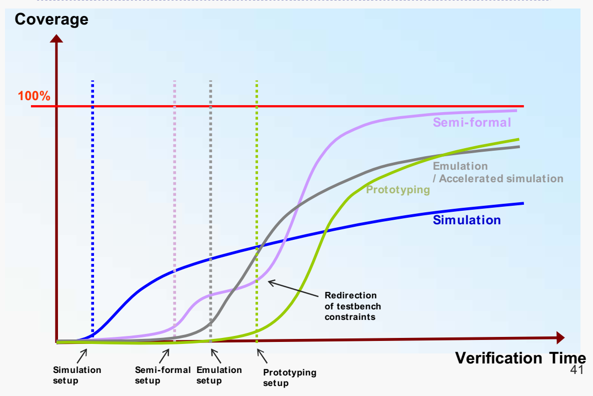
Speed Comparison



Design Complexity

	Gate counts	Comments
Simulation/Semi-formal verification	Unlimited	
Emulation/Hardware-accelerated simulation	1M~16M gates	Depends on the number of FPGAs in the architecture
Prototyping	1M~5M gates	Depends on the components on the board
Formal verification	< 10K gates	Limited to about 500 flip-flops due to state explosion

Verification Time vs. Coverage



Agenda

- Why Verification ?
- Verification Alternatives
- Languages for System Modeling and Verification
  - System modeling languages
  - Testbench automation & Assertion languages
- Verification with Progressive Refinement
- SoC Verification
- 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

Accellera

- Three different ways of specifying Assertions in Verilog designs:
  - OVL (Open Verification Library)
  - PSL (Property Specification Language)
  - Native assertion construct in System Verilog
    - (Ney) In fact, today this is a new language, SVA (SystemVerilog Assertions), complementary to PSL

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
  - **Verilog-AMS** 2.1

Accellera's PSL (Property Specification Language)

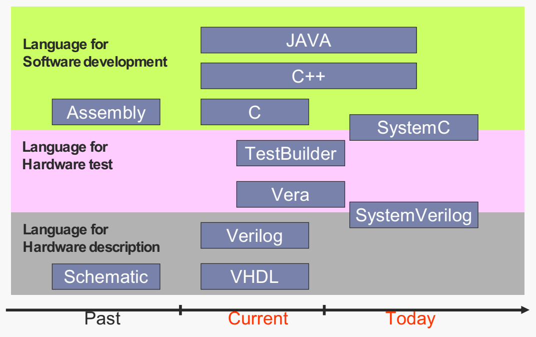
- Gives the design architect a standard means of specifying design properties using a concise syntax with clearly defined formal semantics
- Enables RTL implementer to capture design intent in a verifiable form, while enabling the verification engineer to validate that the implementation satisfies its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

SCE(Standard Co-Emulation Interface)-API

- SCE-API standard defines a high-speed, asynchronous, transaction-level interface between simulators or testbenches and hardware-assisted solutions such as emulation or rapid prototypes

Language Heritage for SoC Design

- New languages are developed to fill the productivity gap

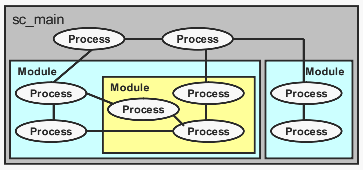

Language for Software development
JAVA
C++
Assembly
C
SystemC
Language for Hardware test
TestBuilder
Vera
SystemVerilog
Language for Hardware description
Verilog
Schematic
VHDL
Past
Current
Today

SystemC

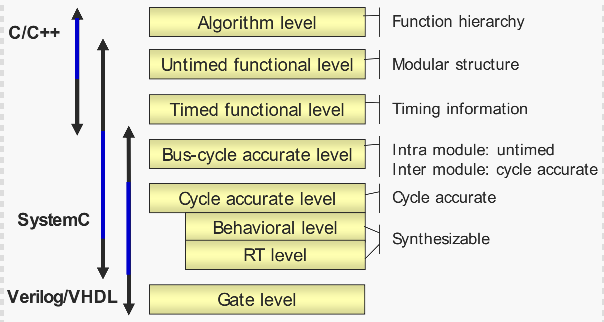
- SystemC is a modeling platform consisting of
  - A **library** of **C++ classes** for modeling hardware
  - Including a **simulation kernel** that supports hardware modeling concepts at the system level, behavioral level and register transfer level
- SystemC enables us to effectively create
  - A **cycle-accurate model** of
    - Software algorithm
    - Hardware architecture
    - Interfaces of System-on-a-Chip
- Program in SystemC can be
  - An **executable specification** of the system

SystemC

- ***Modules, ports***, and ***signals*** → for hierarchy
- ***Processes*** → for concurrency
- ***Clocks*** → for time
- ***Hardware data types*** → for bit vectors, 4-valued logic, fixed-point types, arbitrary precision integers
- ***Waiting*** and ***watching*** → for reactivity
- ***Channel, interface***, and ***event*** → for abstract communications



Abstraction Levels of SystemC



Test-bench automation

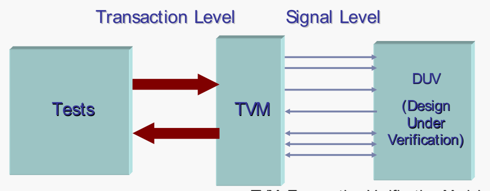
- Why is test-bench automation required?
  - Test-bench for IP can be more complex than the IP itself
  - Manual description of the test-bench is a time-consuming job
  - Simulating the whole test-bench in HDL yields excessive verification time
- Players
  - TestBuilder (Cadence)
    - Closer to C, integrated to SystemC
    - (Ney) Today, (2008) adopted as the SCV (SystemC Verification Standard)
  - VERA (Synopsys)
    - Closer to Verilog, integrated to SystemVerilog

TestBuilder

- Transaction-Based Verification

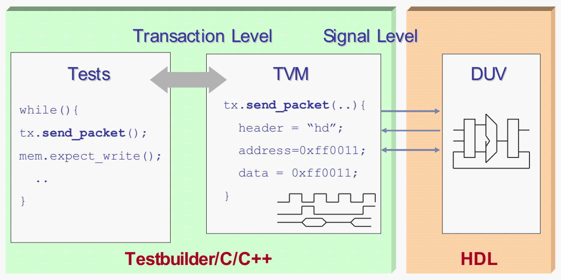
  Functional verification in higher-level abstraction

  Engineer develops tests from a system-level perspective
  - Advantages
    - Enhance reusability of each component in the test-benches
    - Improve debugging and coverage analysis



TVM: Transaction Verification Model

TestBuilder

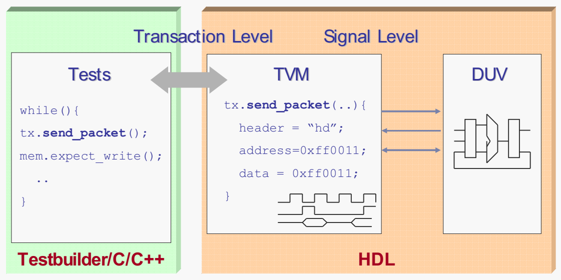


```
while(){
tx.send_packet();
mem.expect_write();
  ..
}
```

```
tx.send_packet(..){
  header = "hd";
  address=0xff0011;
  data = 0xff0011;
}
```

- TVM (Transaction Verification Model)
  - Translates a bus cycle command to a signal waveform
  - May be described in C API or Verilog PLI. (Ney) Today, (2008) this is described (mostly) in SystemC, using the SCV

TestBuilder

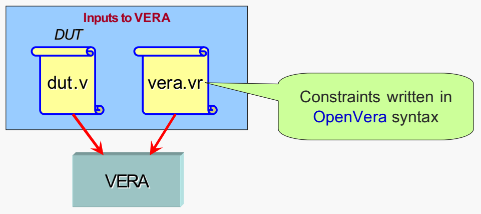


```
while(){
tx.send_packet();
mem.expect_write();
  ..
}
```

```
tx.send_packet(..){
  header = "hd";
  address=0xff0011;
  data = 0xff0011;
}
```

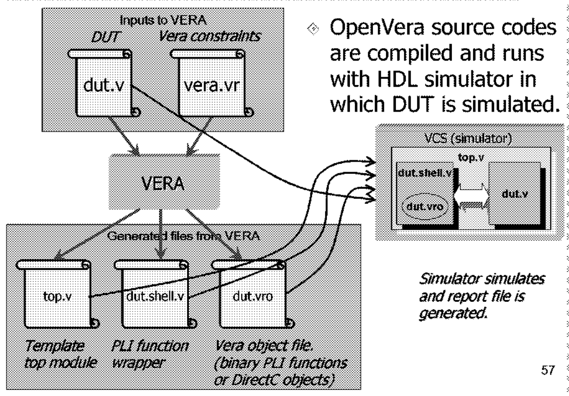
- TVM (Transaction Verification Model)
  - Translates a bus cycle command to a signal waveform
  - May be described in C API or Verilog PLI

VERA (Synopsys)



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

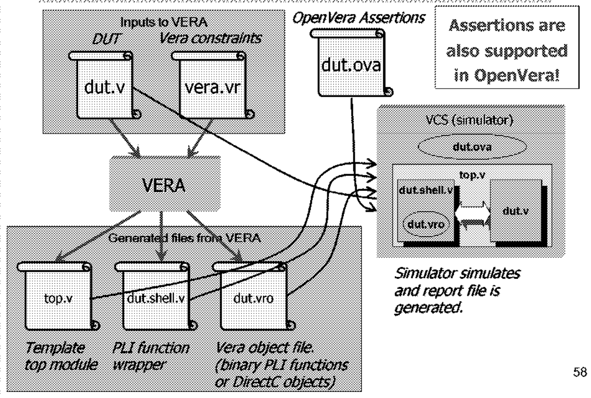
VERA



- OpenVera source codes are compiled and runs with HDL simulator in which DUT is simulated.

Simulator simulates and report file is generated.

Vera



Assertions are also supported in OpenVera!

Simulator simulates and report file is generated.

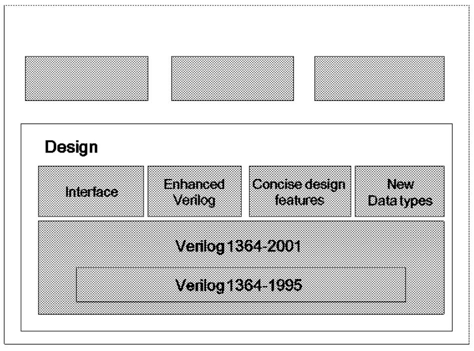
SystemVerilog

- 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
- (Ney) Today (2008), this last characteristic is also supported by SystemC!!

SystemVerilog

- New data types for data abstraction level higher 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



System Description Languages Summary

Languge	Pros	Cons
C/C++	Easy to write test vectors/environment	Unable to handle some hardware environments
HDL (Verilog, VHDL)	Familiarity Easy to describe H/W designs	Focuses on the lower-level designs Improper for system modeling
SystemC	Easily connected to C/C++ codes Easy to model system behaviors	Limited tools (simulation, synthesis, etc.) - (Ney) This is no longer true!
SystemVerilog	Easy to learn for the HDL designers Easy to model system behaviors	Few tools (simulation, synthesis, etc.)

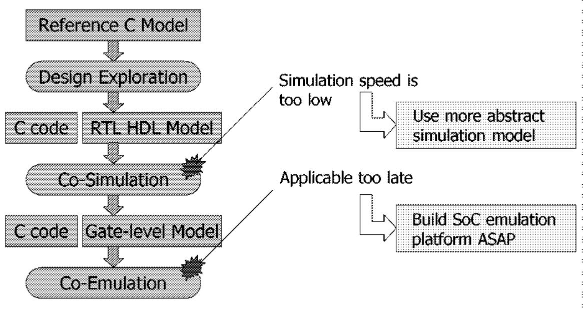
Agenda

- 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
- SoC Verification
- Concluding Remarks

Criteria for Good SoC Verification Environment

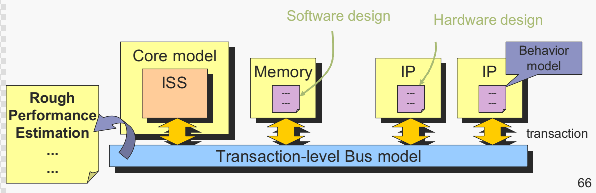
- Support various abstraction levels
- Support heterogeneous design languages
- Trade-off between verification speed and debugging features
- Co-work with existing tools
- Progressive refinement
- Platform-based design

Conventional SoC Design Flow



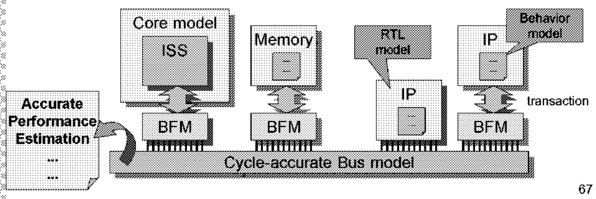
Transaction-Level Modeling

- 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



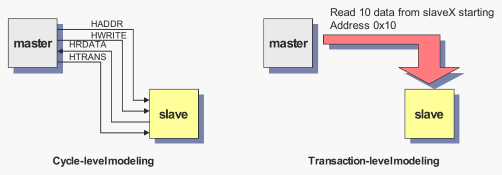
Cycle-Accurate Bus Modeling

- 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.



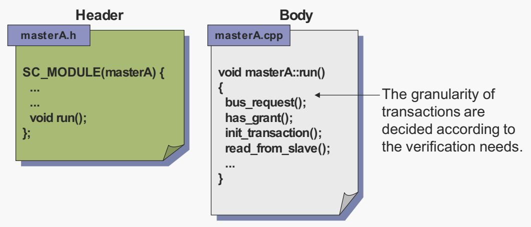
AMBA AHB CLI Specification

- 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.


master
HADDR
HWRITE
HRDATA
HTRANS
slave
Cycle-level modeling
Read 10 data from slaveX starting Address 0x10
master
slave
Transaction-level modeling

AMBA AHB CLI Specification

- Example master implementation
  - Transactions are represented as methods in transaction-level modeling.
  - The abstraction levels of each method can be decided as needed such as cycle-count accurate, cycle-accurate, transaction accurate, etc.


Header
Body
The granularity of transactions are decided according to the verification needs.

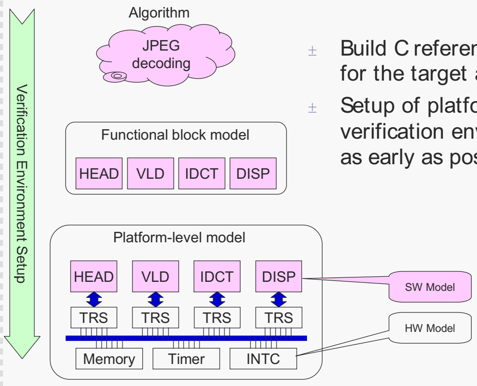
masterA.h

```
SC_MODULE(masterA) {
 ...
 ...
 void run();
};
```

masterA.cpp

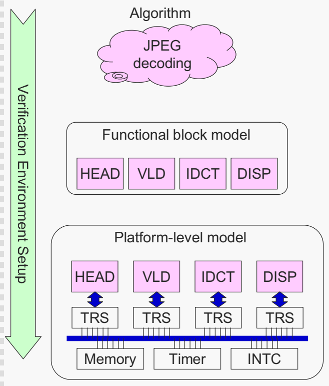
```
void masterA::run()
{
  bus_request();
  has_grant();
  init_transaction();
  read_from_slave();
  ...
}
```

Flexible SoC Verification Environment



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

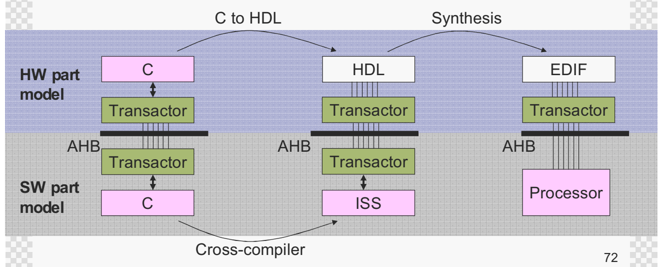
Flexible SoC Verification Environment



- Transactor connects various types of SW block models with HW bus system.
- Several types of transactors must be prepared, for example
  - AHB ←→ C model
  - AHB ←→ HDL model
  - OPB ←→ Testbuilder
  - PLB ←→ SystemC

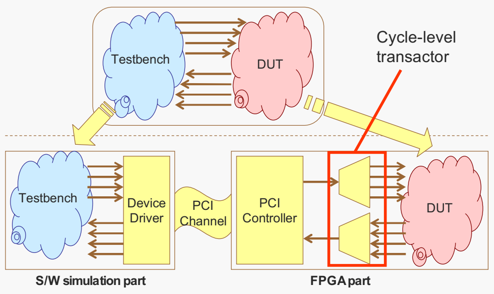
Flexible SoC Verification Environment

- Socketize IP representation
  - HW: C → HDL → EDIF
  - SW: native C → ISS → Processor Core



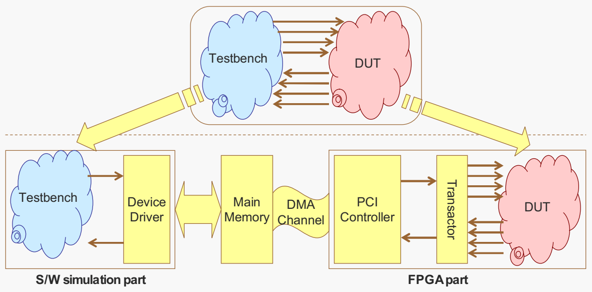
Cycle-Level Transactor

- 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

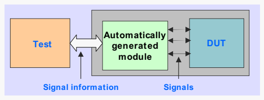


Cycle vs. Transaction-level Transactor

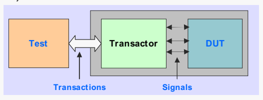
- Cycle-level transactor
  - Synchronized at every clock cycle.
  - Transactor can be automatically generated according to the pin count.
  - Operating speed depends on the number of signals to be transferred.
- Transaction-level transactor
  - Synchronized at the end of each transaction.
  - Transactor 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

Example) 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)



OpenVera (OV) verification IP

- Reusable verification modules, i.e.,

  1) bus functional models,

  2) traffic generators,

  3) protocol monitors,

  and

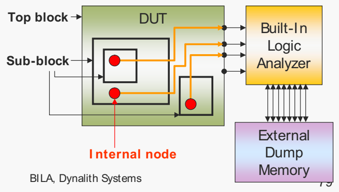
  4) functional coverage blocks.

Companies providing OpenVera Verification IP

- ControlNet India IEEE 1394, TCP/IP Stack
- GDA Technology HyperTransport
- HCL Technologies I2C
- Integnology Smart Card Interface
- nSys SIEEE 1284, UART
- Qualis Design Ethernet 10/100, Ethernet 10/100/1G, Ethernet 10G, PCI-X, PCI, PCI Express Base, PCI Express AS, 802.11b, ARM AMBA AHB, USB 1.1, USB 2.0
- Synopsys, Inc. AMBA AHB, AMBA APB, USB, Ethernet 10/100/1000, IEEE 1394, PCI/PCIx, SONET, SDH, ATM, IP, PDH

Debugging Design in the FPGA

- Embed logic analyzer with user design in EDIF format
  - Logic to store pre-registered signals into the probing memory.
  - Logic for trigger condition generation.
  - Triggering condition is dynamically configured.
- Internal node extraction
  - Sometimes the designer wants to watch internal nodes in the design.
  - Internal node probing enables this by wiring-out the internal nodes to the boundary of the DUT top block.

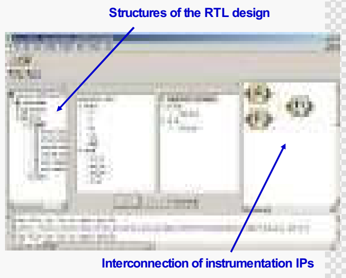


RTL Debugging Feature

- Emulation is based on gate-level netlist.
- Gate-level netlist generated from the synthesis tools has too complex name styles difficult to trace manually.
- Techniques to resolve RTL symbol names from the gate-level symbol names and to provide debugging environment in RTL name spaces is required.
- Insert RTL instrumentation IP for debugging
  - Design flow
    - Read RTL design (Verilog, VHDL)
    - Generate instrumented RTL design (spiced with triggering and dump logic)
    - Synthesis
    - Compile (mapping & PAR)
  - DiaLite (Temento), Identify (Synplicity)

RTL Debugging Feature

- Instrumentation IPs for debugging logic blocks mapped into FPGAs.
  - Trigger
  - Logic Equation Module
  - History Register
  - Transaction Register
  - Random Generator
  - Traffic Analyzer
- Instrumentation IPs are interconnected to support various configurations.



Structures of the RTL design
Interconnection of instrumentation IPs

DiaLite from Temento

FPGA-based Debuggers

Debugger	Level	Memory	Control ports	Triggering conditions
Dynalith Systems BILA	Netlist level	External memory	PCI	Dynamically configurable without recompiling FPGA
Temento DiaLite	RTL	Block memory of FPGA	JTAG signals mapped to user I/O ports	Dynamically configurable without recompiling FPGA
Synplicity Identify	RTL	Block memory of FPGA	Dedicated JTAG Ports of FPGA	Dynamically configurable without recompiling FPGA

Connecting Actual Chip to the Simulator

- Building a correct and fast reference model for the hardware is very difficult.
  → Use the actual discrete chip for the IP (or FPGA).
- Control the clock signal to the actual chip (or FPGA), i.e, slow down and synchronize with the HW simulator and SW debugger in the host machine.
- Application
  - FPGA prototyping
  - HW/SW co-verification
  - Silicon validation

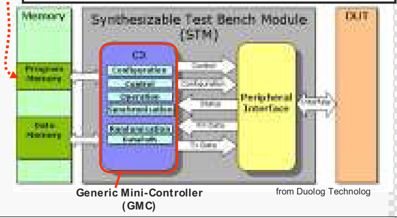
Standard Simulation Environment
Standard Software Debug Environment

from SimPOD

Synthesizable Testbench

- Prepare a set of programmable test bench module which can be also synthesized into hardware.
- To verify a DUT, build a test bench using the programmable test bench.
- The test bench is applicable to both simulation and emulation in the same fashion.

```
gmc_cfg_datasize    3 # Set to 32 Bits
gmc_cfg_msmode    0xF # Set to Master TX
gmc_cfg_datapath_0x00 # Read/Write to Mem
gmc_cmd_send_data 100 # Send 100 Bytes
uart_cfg_odd_parity 1 # Set ODD parity
```



Memory
Program Memory
Data Memory
Synthesizable Test Bench Module (STM)
Peripheral Interface
DUT
Generic Mini-Controller (GMC)

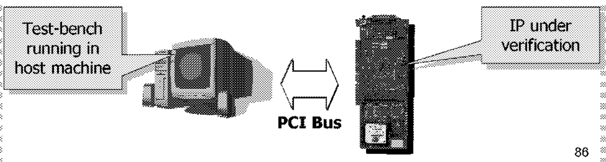
from Duolog Technolog

Large-Scale Emulators

- Celaro, Mentor
  - Massive array of FPGA's
  - Distributed compilation
  - RTL debuggability
  - Full visibility without re-compilation
- VStation, Mentor (IKOS)
  - Reduced routing problem by multiplexing multiple physical signal lines to a virtual wire.
- Palladium, Quickturn (Cadence)
  - Use custom processor-array instead of FPGA
  - Support synthesizable testbench
  - Support multi-user operation

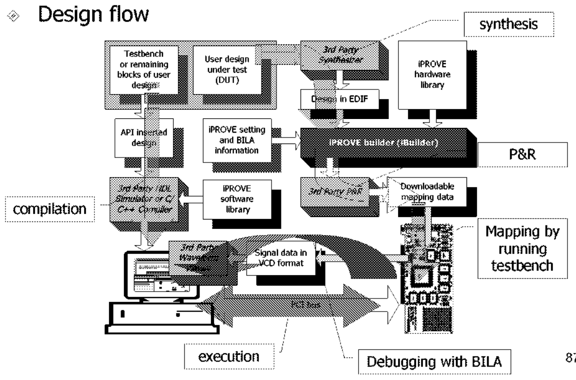
Simulation Acceleration

- Use one or several large-scale FPGA's instead of array of small FPGA's
  - Reduce pin limitation overhead between FPGA's
  - Utilize advanced features of state-of-the-art FPGA's
- Commercial products
  - Eve – ZeBu
  - ALATEK – HES
  - Dynalith – iPROVE



iPROVE Technology

- Design flow



synthesis
Testbench or remaining blocks of user design
User design under test (DUT)
3rd Party Synthesizer
iPROVE hardware library
Design in EDIF
API inserted design
iPROVE setting and BILA information
iPROVE builder (iBuilder)
P&R
compilation
3rd Party HDL Simulator or C/C++ Compiler
iPROVE software library
3rd Party P&R
Downloadable mapping data
Mapping by running testbench
3rd Party Waveform Viewer
Signal data in VCD format
PCI bus
execution
Debugging with BILA

Agenda

- Why Verification ?
- Verification Alternatives
- Languages for System Modeling and Verification
- Verification with Progressive Refinement
- SoC Verification
  - Co-simulation
  - Co-emulation
- Concluding Remarks

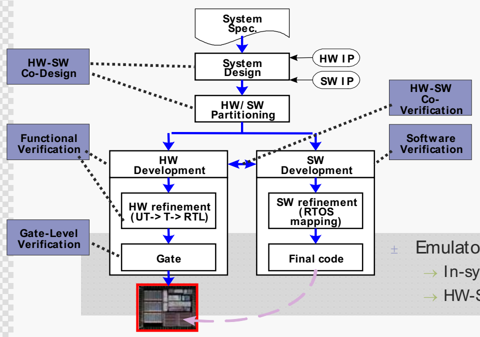
SoC Verification

- Co-simulation
  - → Connecting ISS with HDL simulation environment
  - → Seamless, N2C
- Co-emulation
  - → Emulation/rapid-prototyping equipments supporting co-emulation
  - → ARM Integrator, Aptix System Explorer, AXIS XoC, Dynalith iPROVE

What's the point in SoC Verification?

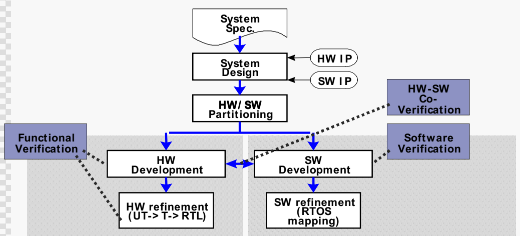
- Mixture of SW and HW
  - → Let the HW model to cooperate with Processor Model such as ISS or BFM (Bus functional model)
- Mixture of pre-verified, unverified components
  - → Utilize legacy IPs already verified
- Mixture of different language, different abstraction levels
  - → Provide common interface structure between SoC components

Canonical SoC design flow



- Emulator
  - → In-system emulator
  - → HW-SW co-debugging

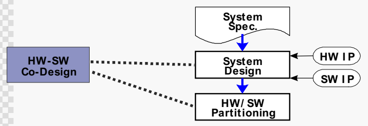
Tools for HW-SW Co-Verification



- → High-level synthesis
- → Testbench automation
- → IP accelerator

- → HW-SW co-simulation
- → ISS
- → RTOS simulator

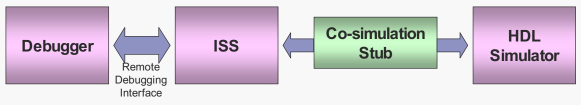
Tools for System-level Verification



- System-level design (Performance analysis tools)
  - → Hot-spot analyzer
  - → High-level cycle count estimation
  - → High-level power analysis
  - → High-level chip area estimation
  - → On-chip-bus traffic estimation

Co-Simulation Tools

- Software debugging in ISS and hardware verification in HDL simulator are done in parallel way.
- Co-simulation stub manages the communication between HDL simulator and ISS.
- The most accurate solution albeit very slow
- Commercial Products
  - → Eagle*i* (Synopsys), Seamless (Mentor)



Instruction Set Simulator

- Interpretive ISS
  - → Slow but flexible and accurate
- Compiled ISS
  - → Fast and accurate but applicable only for static programs
  - → Static vs. dynamic
    - → Depending on the code generation is done in static or dynamic due to cache miss, branch prediction and self-modifying code, etc.
- Native Code (not an ISS)
  - → Fast but not accurate
  - → I/O handling problem

```
while(true) {
  inst = fetch( pc );
  opcode=decode(inst);
  switch( opcode ){
        …
    case ADD:
        …
        break;
  }
}
```

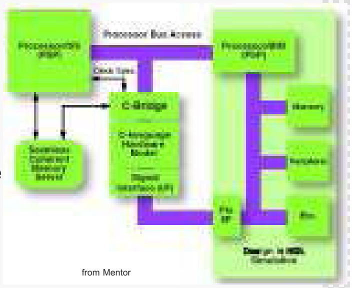
Main loop of interpretive ISS

Instruction Set Simulator

- Execution speed
  - → Native code > Static compiled ISS > Dynamic compiled ISS > Interpreted ISS
- Accuracy
  - → Native code < Static compiled ISS = Dynamic compiled ISS <= Interpreted ISS

Seamless (Mentor)

- Seamless and C-Bridge enables co-verification of ISS(Processor) + HDL + C Hardware model
- Full visibility and dynamic performance estimation.
- Supports various CPU's (over 100 models)
- The communication between the S/W and the H/W is optimized to maximize the verification performance



from Mentor

N2C

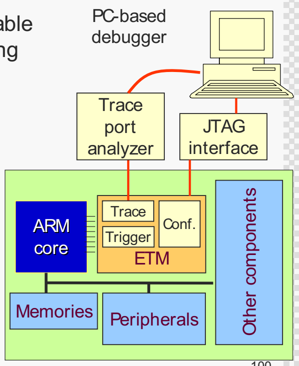
- A set of tools allowing co-simulation of the system described in various abstraction levels
  - → Un-timed C
  - → Timed functional description
  - → Behavioral bus cycle accurate description
  - → RTC (Register Transfer C)
  - → HDL with C
- Interface synthesis to enable platform exploration
  - → Interface synthesis make it possible to verify the performance of each platform efficiently in the early design stage.
    - → Solving Hardware/software partitioning problems.
    - → Deciding bus architecture of the SoC.

Co-Emulation Tools

- Link hardware emulators with a processor model running in host machine or an actual processor core module
- Most emulation and rapid prototyping products support linkage with ISS running in host machine
- As the emulator runs very fast, speed of ISS including memory access as well as synchronization between emulation and ISS rise as new bottlenecks in verification

Example) ARM ETM (Embedded Trace Macrocell)

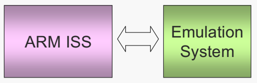
- Real-time trace module capable of instruction and data tracing dedicated to the ARM core family
- Triggering enables to focus collection around the region of interest
- Trace module controls the amount of trace data by filtering instructions or data
- Only applicable to the ARM core debugging



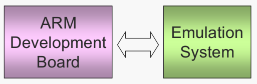
Target system with ARM based ASIC

Typical Co-Emulation Environment

- Connect ARM ISS or ARM board to the emulation system.



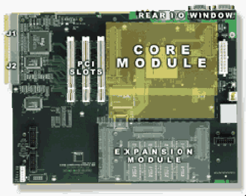
- → ARM ISS can be configured to user's target SoC architecture.
- → SW debugger can be fully utilized.



- → Faster than ISS.
- → Ready-made ARM development boards has fixed architecture and it may be different from user's desire.

ARM Integrator

- ARM prototyping platform
- Composed of the followings
  - → Platform : AMBA backbone + system infrastructure
  - → Core Module : various ARM core modules (up to four)
  - → Logic Module : FPGA boards for AMBA IP's
- Allows fast prototyping of ARM-based SoC
- Enables co-emulation of both software in ARM processor core and hardware in the FPGA
- Difficult to debug hardware logics



from ARM

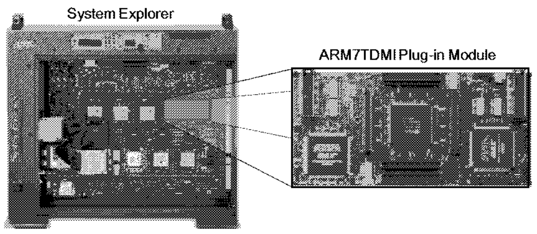
XoC (Axis)

- ARM core module is connected to Axis's FPGA arrays.
  - → Source level debugging for both hardware and software
  - → HW/SW logic simulation hot swapping → VCD on demand
  - → Software instant replay
    → Correlate bus transaction with software instruction

from Axis

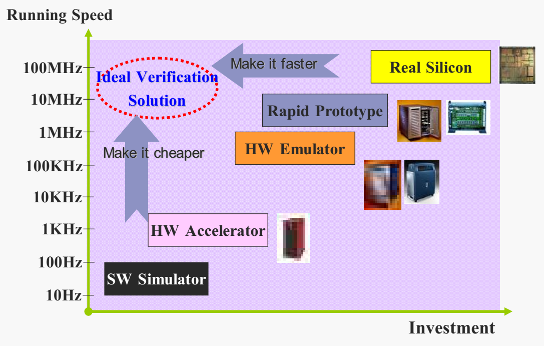
System Explorer (Aptix)

- Backplane with FPID's (Field-Programmable Interconnect Device) and connector array.
- ARM plug-in module is inserted to one portion of connector array.



from Aptix

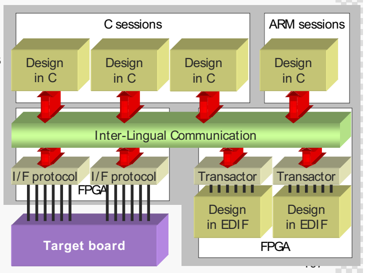
ASIC Verification Methods



iSAVE-MP (Dynalith)

iSAVE-MP (Dynalith)

- All-in-one verification :
  - → Heterogeneous models including ARM ISS, C hardware model, HDL hardware description
  - → SW models run in linux-based PC
  - → HW models run in FPGA's
- Debugging with PSA
  - → Probe FPGA internal signals values to SRAM memory on the fly.
    - → fastest operating speed, wide and deep sampling window
- Communicate with C model using PCI DMA



Tools for SoC Design

Tools	Pros	Cons
ADS	· Required tool for ARM SW development	· No consideration about HW IP's.
ARM Real-Time Debuggers	· SW programming is easy on ARM-based prototyping hardware or SoC.	· No consideration about debugging of HW IP's.
Seamless	· Cosimulation environment is supported. · Many CPU types	· Low speed
N2C	· Cosimulation environment is supported. · C and SystemC languages supported.	· Low speed
ARM Integrator	· Semi-customization using modules is possible. · HW prototyping with ARM · ARM SW debugging through ETM & ETB.	· Complete customization is not possible. · Debugging of IP embedded in FPGA is not easy.

Tools for SoC Design

Tools	Pros	Cons
XoC	· Cosimulation with ARM · ARM SW debugging though ETM & ETB · HW IP debugging	· Long compilation time
System Explorer	· HW IP debugging · Module-based customization · Cosimulation environment	· Long compilation time · Manual setup required
iPROVE/ iSAVE	· Cosimulation with ARM ISS · SW debugging through ISS · HW debugging is supported. · Low cost	· Long compilation time

Agenda

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

Concluding Remarks

- Verification is challenging; It needs strategy!
- Strategy is to apply each method when appropriate
- Verify as early as possible; Catch the bug when it is small and still isolated in a smaller region (Don't wait until it grows and kills you)
- 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

- Main differences of SoC with 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); You lose in the beginning, but will win later, like Design for Reuse