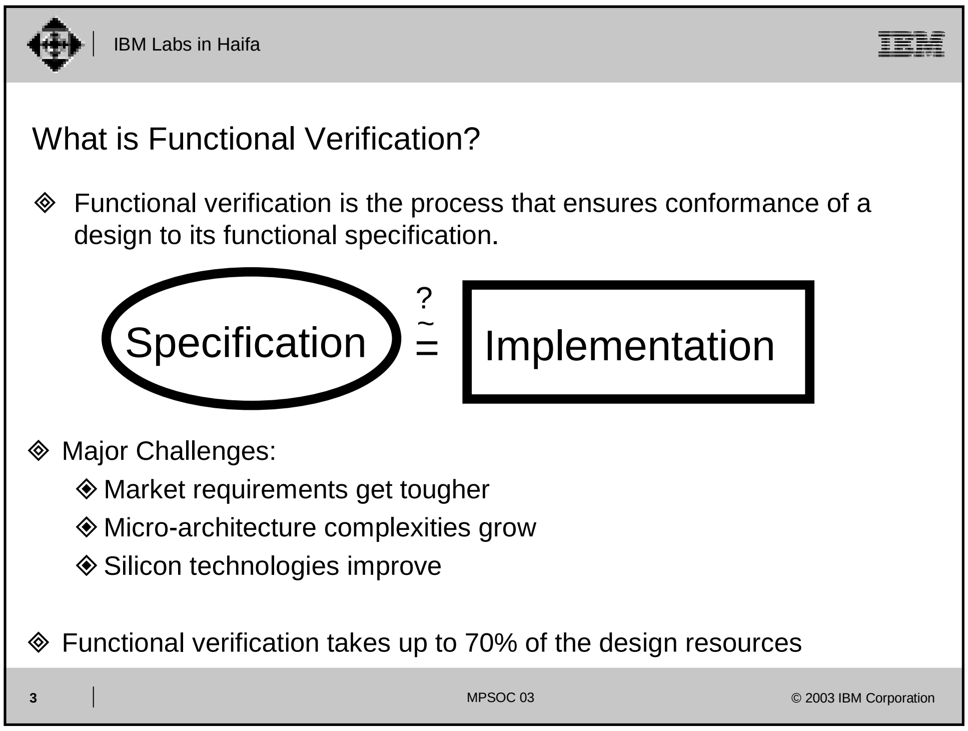
Functional Verification and the SoC Challenge

Avi Ziv
Simulation Based Methods
IBM Research Lab in Haifa

Outline

- Introduction to functional verification
  - What is functional verification?
  - Leading functional verification techniques
- The SoC challenge
  - What's new in SoC design?
  - Why is verification difficult for SoCs?
- Possible solutions
  - Raise the abstraction level
  - Test generation examples



IBM Labs in Haifa

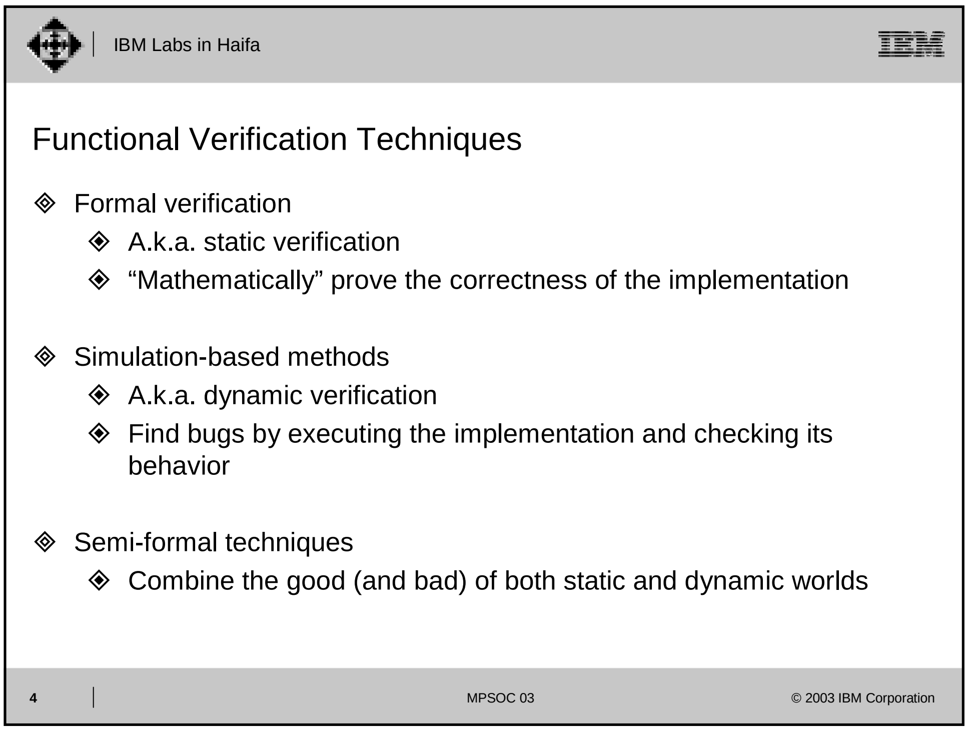
What is Functional Verification?

- Functional verification is the process that ensures conformance of a design to its functional specification.

Specification $\overset{?}{\cong}$ Implementation

- Major Challenges:
  - Market requirements get tougher
  - Micro-architecture complexities grow
  - Silicon technologies improve
- Functional verification takes up to 70% of the design resources

3 | MPSOC 03 © 2003 IBM Corporation



IBM Labs in Haifa

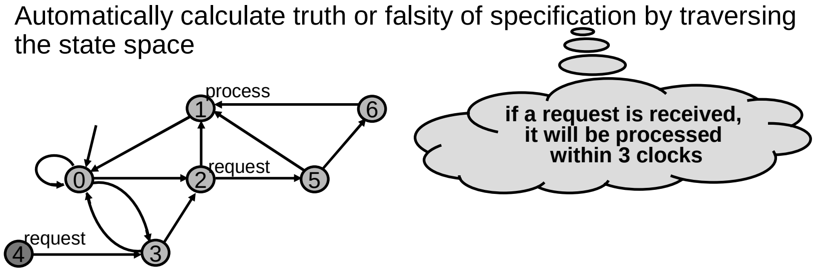
Functional Verification Techniques

- Formal verification
  - A.k.a. static verification
  - “Mathematically” prove the correctness of the implementation
- Simulation-based methods
  - A.k.a. dynamic verification
  - Find bugs by executing the implementation and checking its behavior
- Semi-formal techniques
  - Combine the good (and bad) of both static and dynamic worlds

4 | MPSOC 03 © 2003 IBM Corporation

Key Formal Verification Method: "Model Checking"

- A method for mathematically proving functional properties on the design
  - Proving a property means showing that it holds for all possible input combinations, across all execution paths
  - No tests required
- Model checking operation method:
  - Represent design as a finite state machine
  - Automatically calculate truth or falsity of specification by traversing the state space


process
request
request
0
1
2
3
4
5
6
if a request is received, it will be processed within 3 clocks

Formal Verification Limitations

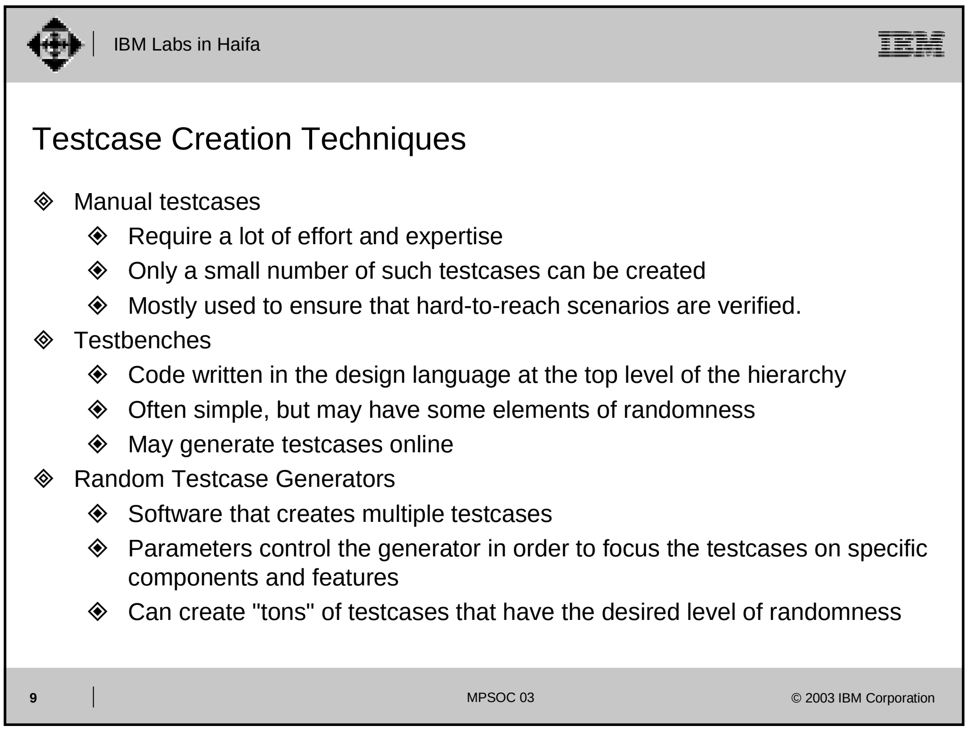
- State-space explosion
  - Need to check all possible states in the implementation
  - Number of states grows exponentially with the number of state variables
  - Current tools are limited to several hundreds or thousands of variables
- Answer only the questions it is asked
  - Translation of English specification to “formal” properties
  - Checked properties may not cover the entire specification

Dynamic Verification

- Method of operation – execute the implementation on an input testcase and check that it behaves according to the specification
  - Size of the design is not a limitation
  - Checking is limited only to the given testcase
  - Unexpected errors may be detected
- Dynamic verification is comprised of
  - Execution techniques
    - Simulation, emulation, etc.
  - Testcase generation
  - Checking
  - Coverage analysis

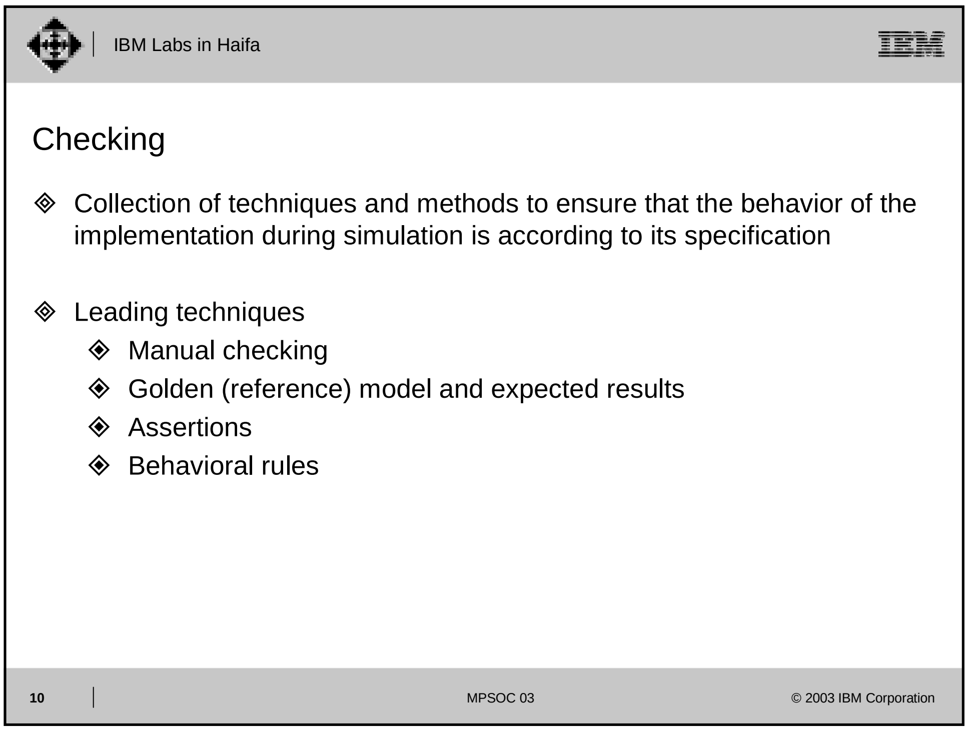
Testcase Creation

- Goal: create input pattern that exercises the design
- The main challenge:
  - Create testcases that reach all the dark corners of the design
- Test patterns need to be
  - Legal
    - Behavior of design under the test is fully specified
  - Interesting
    - Improve coverage
    - Reach corner cases
    - Find bugs
  - Meet specific user requirements



Testcase Creation Techniques

- Manual testcases
  - Require a lot of effort and expertise
  - Only a small number of such testcases can be created
  - Mostly used to ensure that hard-to-reach scenarios are verified.
- Testbenches
  - Code written in the design language at the top level of the hierarchy
  - Often simple, but may have some elements of randomness
  - May generate testcases online
- Random Testcase Generators
  - Software that creates multiple testcases
  - Parameters control the generator in order to focus the testcases on specific components and features
  - Can create "tons" of testcases that have the desired level of randomness



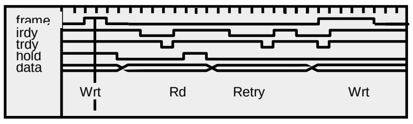
Checking

- Collection of techniques and methods to ensure that the behavior of the implementation during simulation is according to its specification

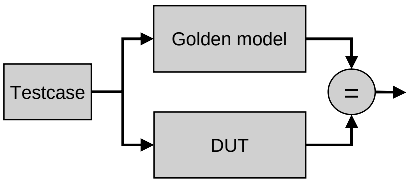
- Leading techniques
  - Manual checking
  - Golden (reference) model and expected results
  - Assertions
  - Behavioral rules

Checking Techniques

- Manual checking
  - View waveform and trace files to analyze the behavior
  - Slow, inefficient, and error prone

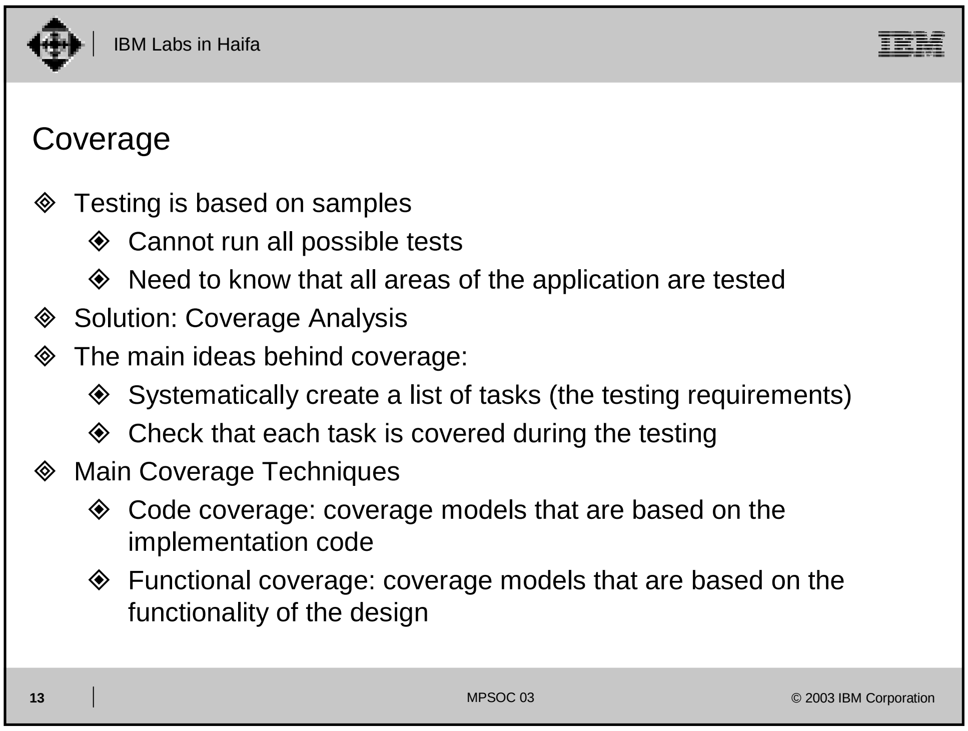


- Golden models and expected results
  - Use the behavior of a golden model to predict the behavior of the design under test
  - Usually limited to external behavior
  - Expected results may also be embedded in the testcase



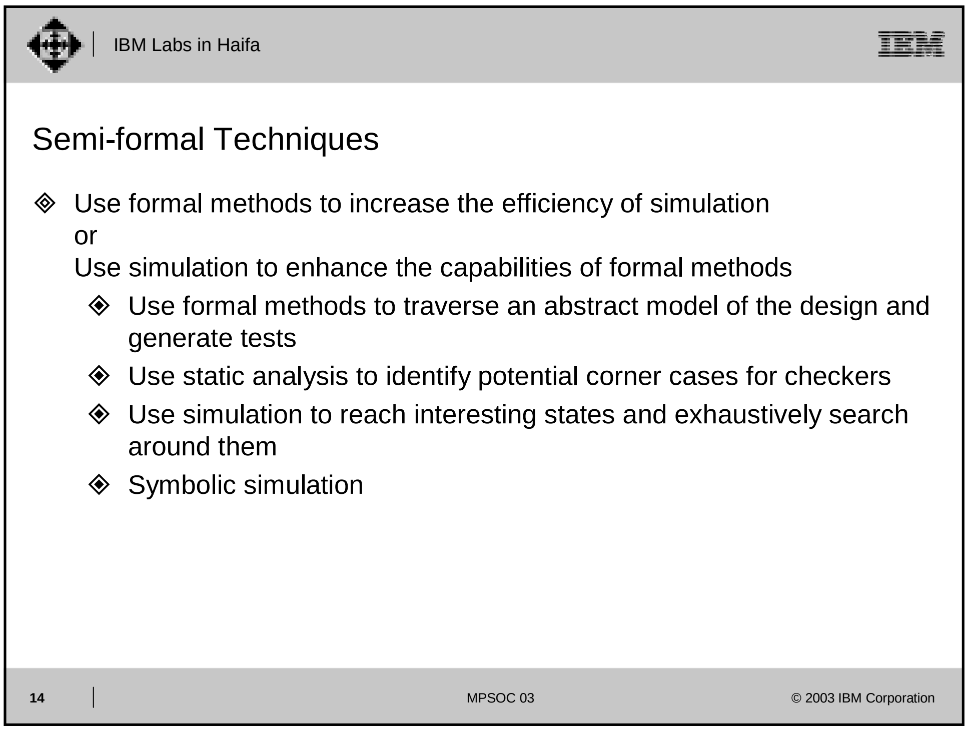
Checking Techniques (2)

- Assertions (or properties checking)
  - Starting from simple assert statements
    - **`assert (length > 0)reprot “Illegal Length”`**
  - Assertions can be manually inserted by the designer into the source code of the design, or they can be externally created and inserted by verification tools
  - Current assertion techniques use temporal property specification languages to specify complex assertions
    - **`{true[*]; req; ack} => {start; data[8]; end}`**
- Behavioral rules
  - Rules that describe the expected behavior of the design
  - Usually rules are more abstract than assertions
    - Not limited to specific facilities
  - Example: scoreboard
    - Check that everything that goes in also comes out



Coverage

- Testing is based on samples
  - Cannot run all possible tests
  - Need to know that all areas of the application are tested
- Solution: Coverage Analysis
- The main ideas behind coverage:
  - Systematically create a list of tasks (the testing requirements)
  - Check that each task is covered during the testing
- Main Coverage Techniques
  - Code coverage: coverage models that are based on the implementation code
  - Functional coverage: coverage models that are based on the functionality of the design



Semi-formal Techniques

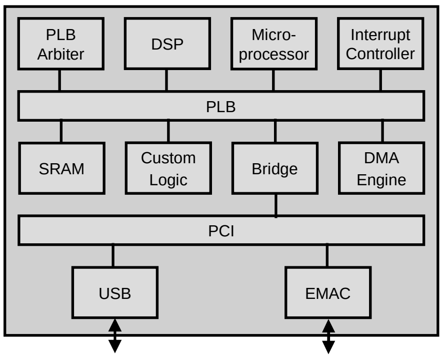
- Use formal methods to increase the efficiency of simulation
  or
  Use simulation to enhance the capabilities of formal methods
  - Use formal methods to traverse an abstract model of the design and generate tests
  - Use static analysis to identify potential corner cases for checkers
  - Use simulation to reach interesting states and exhaustively search around them
  - Symbolic simulation

Outline

- Introduction to functional verification
  - What is functional verification?
  - Leading functional verification techniques
- The SoC challenge
  - What's new in SoC design?
  - Why is verification difficult for SoCs?
- Possible solutions
  - Raise the abstraction level
  - Test generation examples

What's New in SoC Design?

- SoCs change the design world
  - Larger and more complex chips
  - Shorter time to market
  - Smaller design teams
  - Heavy use of pre-existing cores (IPs)
  - Heavy use of processors and DSPs
  - … and software
- How does this affect verification?



Large Designs, Small Teams,

- ◈ SoCs are closer to large computer systems than to ASICs
  - ◈ But they are built with small teams and short development times
- ◈ Verification team cannot gain a deep understanding of the target design
- ◈ Not enough resources to develop verification tools specific for the design

➔ Verification teams must rely on existing tools and technologies
  - ◈ Combined with generic verification knowledge of the domain
  - ◈ With small adaptations to the specific design

Heavy Use of Cores

- ☺ Core are more reliable than custom logic
  - ◈ They have been used and tested before
- ☺ Unit (core) verification may not be necessary

- ☹ Cores are often black boxes
  - ◈ Hard to look inside
- ☹ The cores may not be verified for the specific use scenario of the system
- ☹ Simulation model of the cores may not be available
- ☹ Debugging is much harder
  - ◈ Is the bug in the core or the interface?
  - ◈ How do we debug the internals of the core?
- ☹ Integration is more difficult

Processors and Software

- Processors are big and complex cores
  - … and they are programmable
- Processors provide an efficient way to irritate the rest of the system
- How do we treat the system software?
  - Ignoring it means we are not testing the entire system
  - Leaving it in means:
    - Harder to use the processor to test the rest of the system
    - Harder to stress the system
- SW / HW co-simulation is a major issue
  - Simulating the processor can significantly slow down simulation
  - Hardware and software operate at different rates
  - Modeling solutions are needed (and exist)

Outline

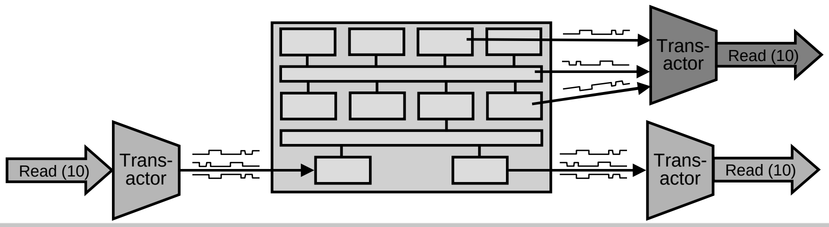
- Introduction to functional verification
  - What is functional verification?
  - Leading functional verification techniques
- The SoC challenge
  - What's new in SoC design?
  - Why is verification difficult for SoCs?
- Possible solutions
  - Raise the abstraction level
  - Test generation examples

Solution: Raise the Abstraction Level

- Raise the level of abstraction to the “right” level
  - Focus at the level in which the design complexity lies
- Advantages
  - Match shift in design paradigm
  - Improved productivity due to reasoning at the "right" level
  - Early start of the verification effort
    - Verification can start on high-level models of the design
    - But the same methodology and tools can be used at lower levels
- New building blocks
  - Signals $\Rightarrow$ packets $\Rightarrow$ complex transactions
  - Components (cores) instead of registers, FSMs, etc.

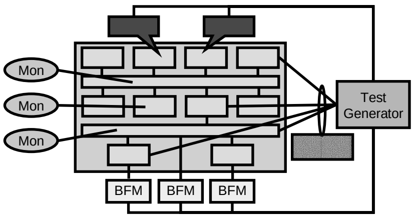
First Step - Transactors

- Translates between high-level verification environment and the design
- Three types of transactors are needed
  - Translate transactions in the testcase to signals in the design interface
  - Translate signals in the design interface to transactions
  - Translate between internal facilities and transactions
- Most existing verification environments support transactors



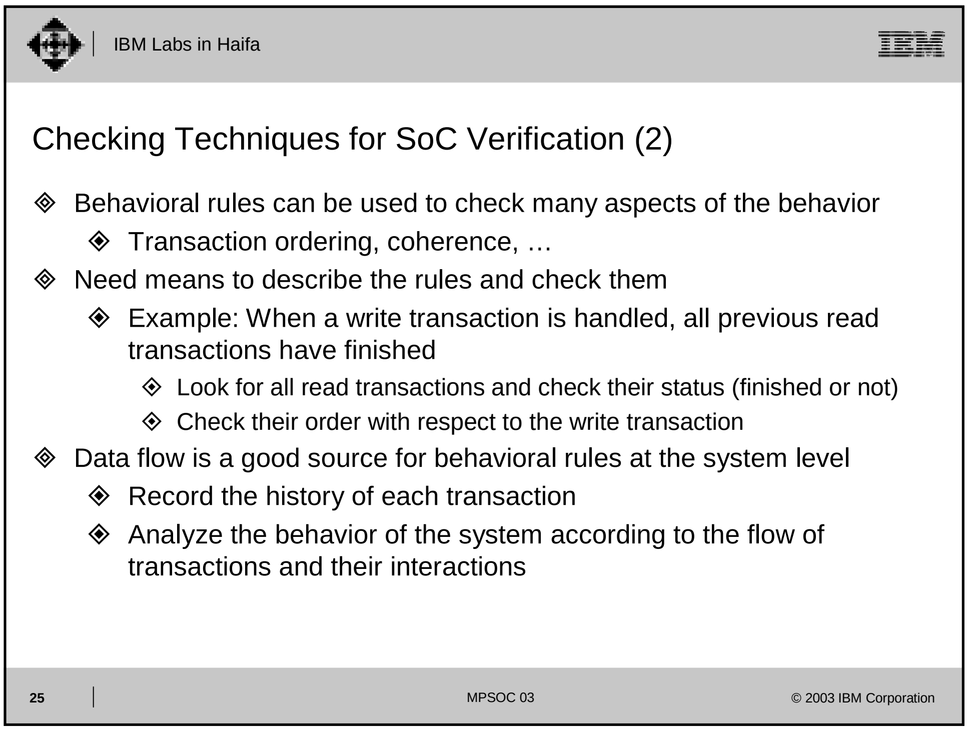
Transactors Are Not Enough

- Need new verification techniques and methodologies to address the SoC paradigm and its challenges
  - Transaction-based checking and coverage
  - System-level test generators
  - New techniques and applications for formal verification
    - Protocol verification



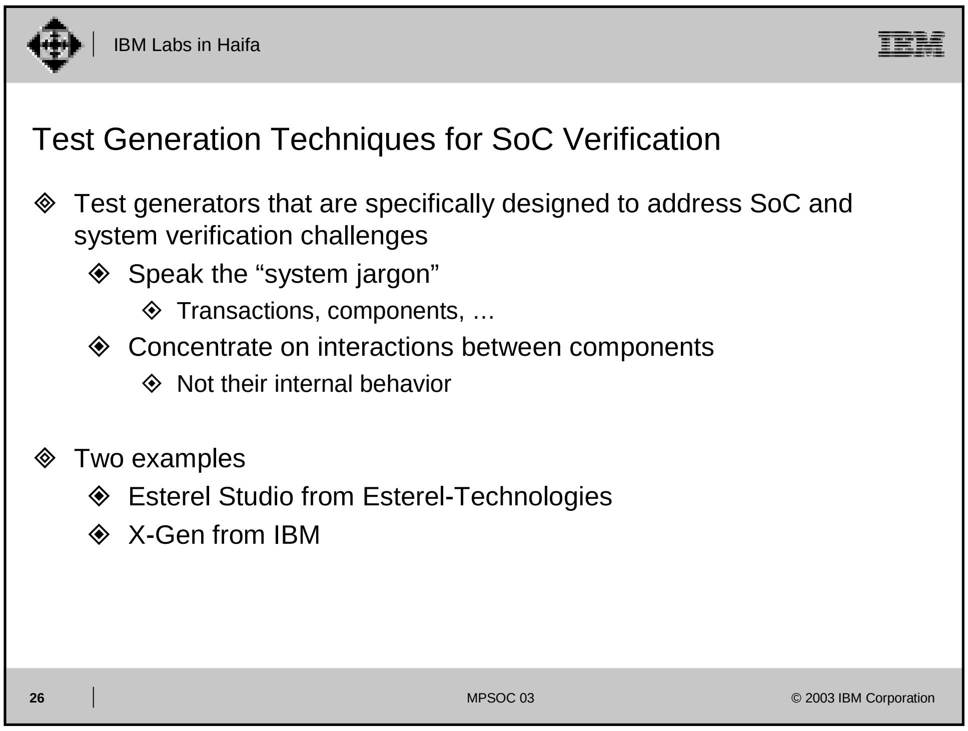
Checking Techniques for SoC Verification

- Golden model / expected results
  - Results are hard to predict because of parallel nature of systems
  - Possible solutions:
    - Cycle-accurate golden model
    - Ignore ordering
- Needed assertions
  - On the interfaces to detect protocol violations
    - Should be provided by the developers of the cores
  - Transaction level assertions
    - New assertion language with transaction vocabulary
      - Length, fields, actors, …
    - Detection of internal transactions may be difficult



Checking Techniques for SoC Verification (2)

- Behavioral rules can be used to check many aspects of the behavior
  - Transaction ordering, coherence, …
- Need means to describe the rules and check them
  - Example: When a write transaction is handled, all previous read transactions have finished
    - Look for all read transactions and check their status (finished or not)
    - Check their order with respect to the write transaction
- Data flow is a good source for behavioral rules at the system level
  - Record the history of each transaction
  - Analyze the behavior of the system according to the flow of transactions and their interactions



Test Generation Techniques for SoC Verification

- Test generators that are specifically designed to address SoC and system verification challenges
  - Speak the “system jargon”
    - Transactions, components, …
  - Concentrate on interactions between components
    - Not their internal behavior
- Two examples
  - Esterel Studio from Esterel-Technologies
  - X-Gen from IBM

Test Generation with Esterel Studio

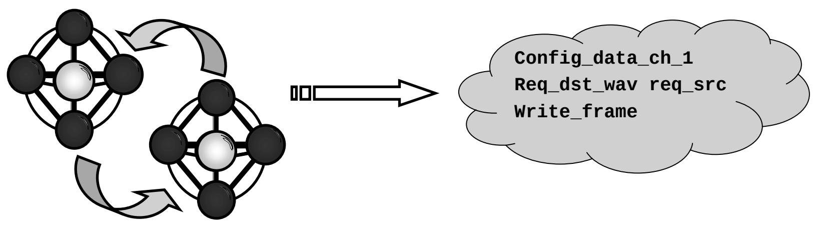
- Goal: Systematic verification of IP interaction to ensure global functional behavior
  - Assumes that each IP has been individually verified
- Generates tests that cover the interactions between IPs
  - Coverage is systematic, well-defined, and complete
- Based on the hierarchical concurrent finite state machines (HFSM) formalism
- Four step process:
  1. Model the system as an HFSM
  2. Create symbolic (abstract) tests
  3. Transform the tests to concrete tests (refinement)
  4. Simulate the concrete tests

Step 1 – Model the System

- Everything is modeled as an HFSM
  - Global HFSM for configuration and test specification
  - Environment HFSM for allowing only legal inputs
  - One HFSM per IP
- Black box model of the IPs
  - Abstract away the data computation performed in the IP
  - Model configuration and interactions with other IPs
    - Interactions are modeled at the transaction level
  - Outputs are symbolic commands of the IP
  - Inputs are arbitrary
    - Whatever is convenient to drive the tests

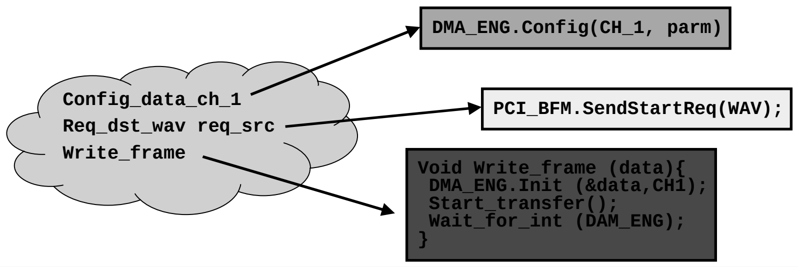
Step 2 – Create Symbolic Tests

- Use BDD-based traversal engine to compute *all* the possible paths to the test_completed state in the global test scheduling FSM
  - Each path is associated with an input sequence
- Transform the input sequences into output sequences using the Esterel Studio simulator
- The resulting sequences are the commands to, and by, the IPs that create the requested scenario



Step 3 – Create Concrete Tests

- Use library of elementary drivers and scripts that translate transactions in the symbolic test into concrete action in the design
- The drivers are typically
  - C / C++ routines for the processors and DSPs
  - Configuration commands for IPs
  - BFM controls



```
DMA_ENG.Config(CH_1, parm)
```

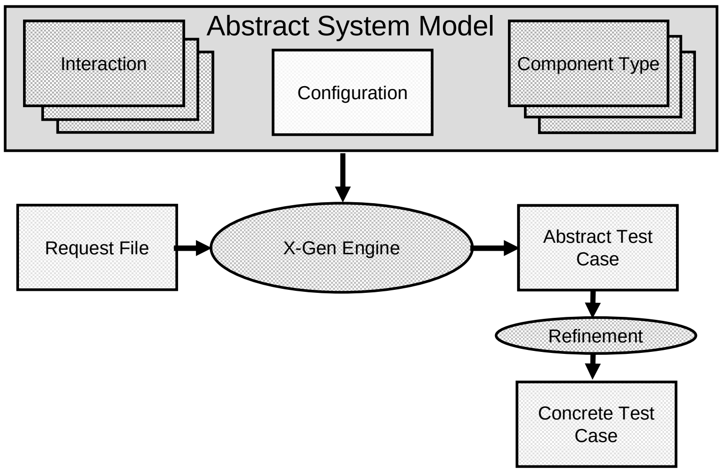
```
PCI_BFM.SendStartReq(WAV);
```

```
Void Write_frame (data){
 DMA_ENG.Init (&data,CH1);
 Start_transfer();
 Wait_for_int (DAM_ENG);
}
```

X-Gen Overview

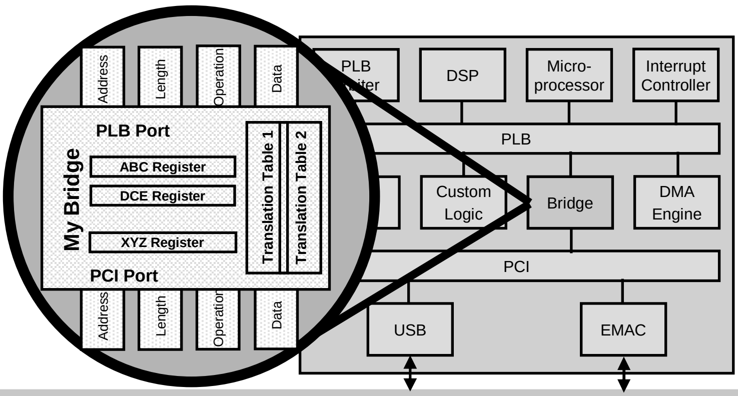
- Applicable to a large variety of systems
  - Generic engine combined with system-specific model
- Separation between system description and test description
- Strong test description capabilities
  - Request file language
- System knowledge is embedded in the tool and in the system model
  - Enables generation of interesting tests
  - Reduces user labor
  - A 'good enough' model

X-Gen Structure



Modeling a System - Component Types

- **Component types:**
  Internal resources, ports, behavior

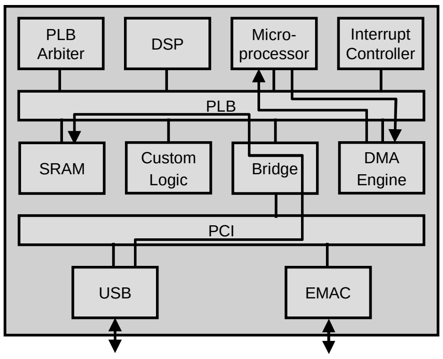


Modeling a System - Interactions

- **Interactions**: Acts, actors

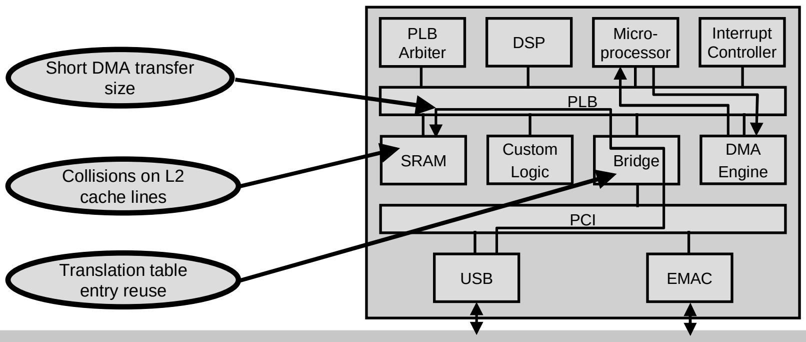
A DMA Interaction

- A CPU stores to the doorbell register of the DMA engine
- The data is moved from the USB port to a memory
- The DMA engine interrupts the initiating CPU

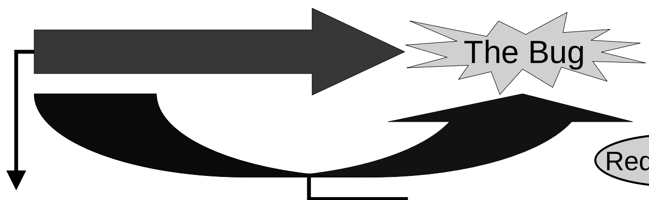


Modeling a System - Testing Knowledge

- Testing knowledge improves test quality
  - Aimed at 'interesting' events
- Testing knowledge can be generic or system dependent
- Modeling testing knowledge: For component types and interactions



Generation Directives: Request Files

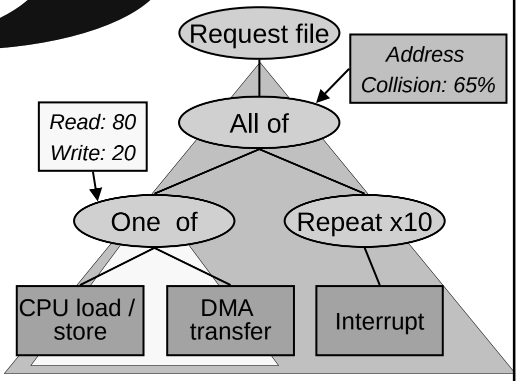


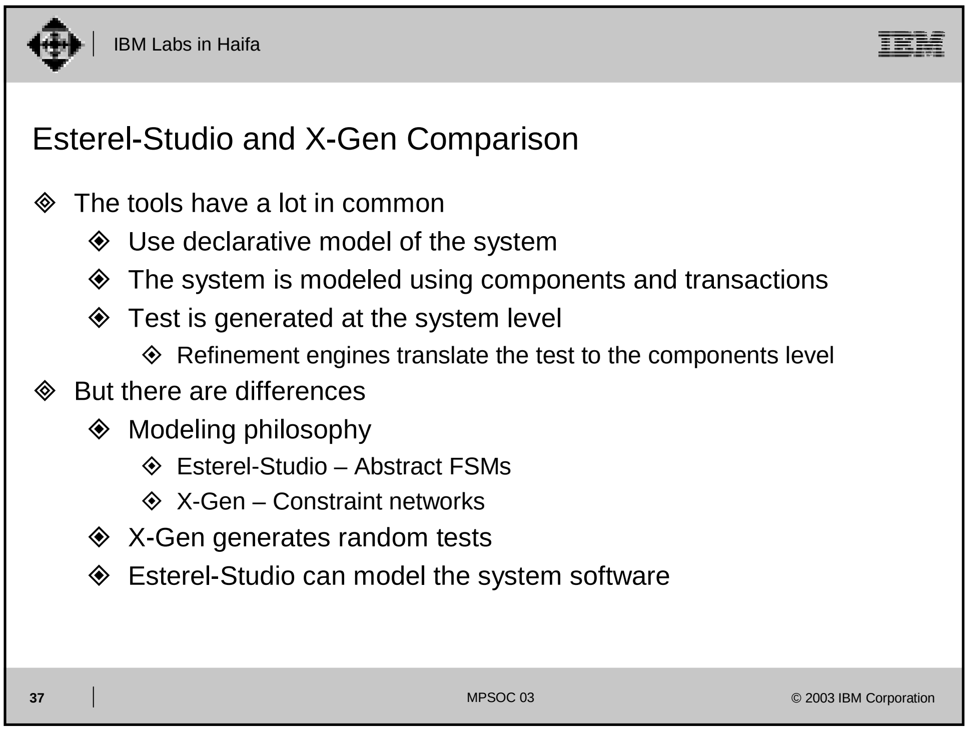
Specifying a scenario

- Interactions as building blocks
- Restrict actors, properties
- Inter-interaction relations

Intelligent background noise

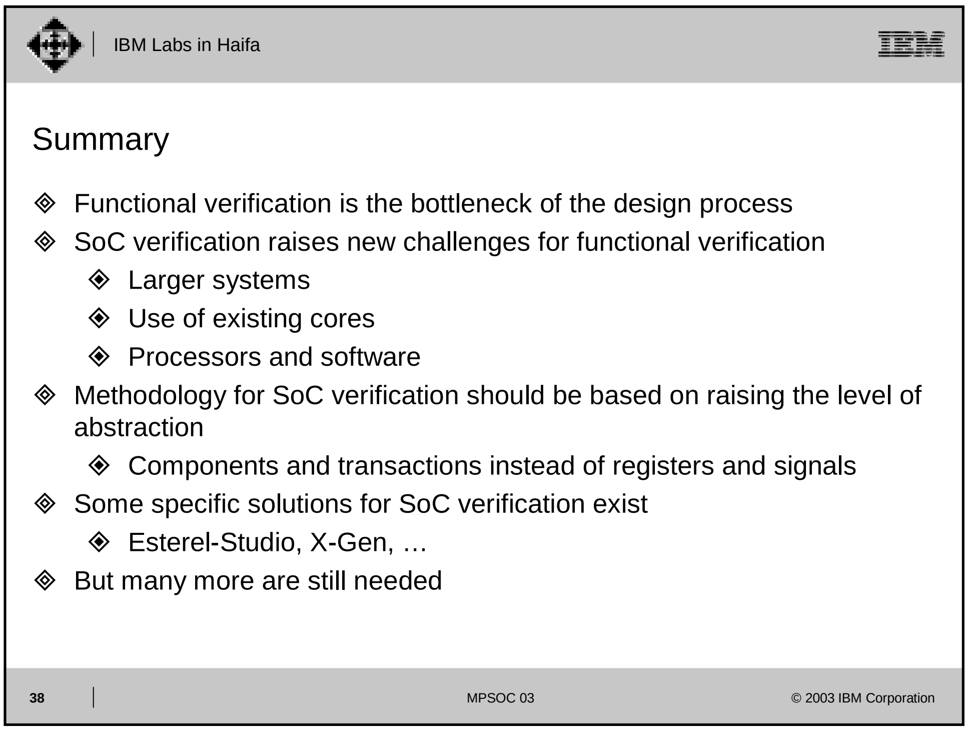
- Built-in testing knowledge
- User direction





Esterel-Studio and X-Gen Comparison

- The tools have a lot in common
  - Use declarative model of the system
  - The system is modeled using components and transactions
  - Test is generated at the system level
    - Refinement engines translate the test to the components level
- But there are differences
  - Modeling philosophy
    - Esterel-Studio – Abstract FSMs
    - X-Gen – Constraint networks
  - X-Gen generates random tests
  - Esterel-Studio can model the system software



Summary

- Functional verification is the bottleneck of the design process
- SoC verification raises new challenges for functional verification
  - Larger systems
  - Use of existing cores
  - Processors and software
- Methodology for SoC verification should be based on raising the level of abstraction
  - Components and transactions instead of registers and signals
- Some specific solutions for SoC verification exist
  - Esterel-Studio, X-Gen, …
- But many more are still needed

Questions?