SoC Verification Methodology

Prof. Chien-Nan Liu

TEL: 03-4227151 ext:4534

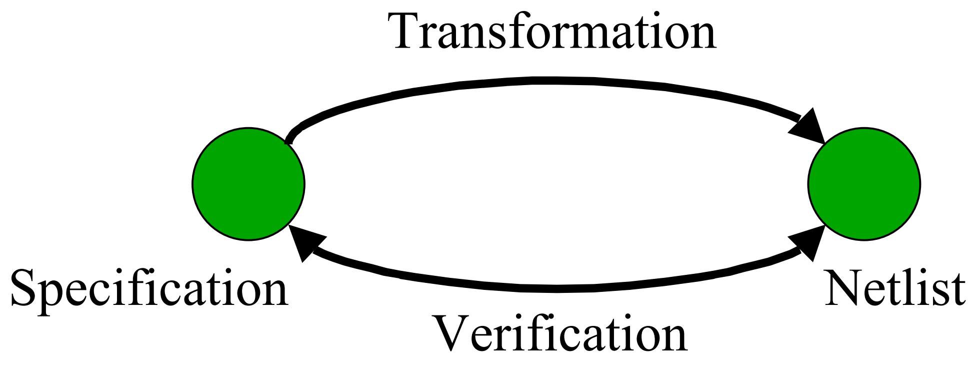
Email: jimmy@ee.ncu.edu.tw

Outline

- $\boxed{\text{Verification Overview}}$
- Verification Strategies
- Tools for Verification
- SoC Verification Flow

What is Verification ?

- A process used to demonstrate the functional correctness of a design
- To making sure your are verifying that you are indeed implementing what you want
- To ensure that the result of some transformation is as expected



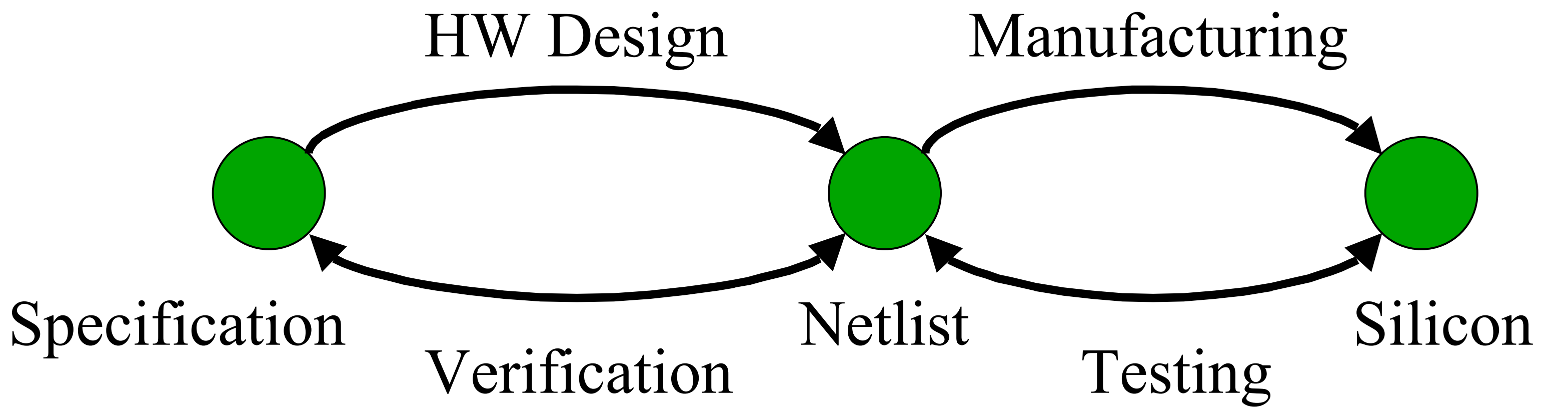
Source : “Writing Test Benches – Functional Verification of HDL Models” by Janick Bergeron, KAP, 2000.

Verification Problems

- Was the spec correct ?
- Did the design team understand the spec?
- Was the blocks implemented correctly?
- Were the interfaces between the blocks correct?
- Does it implement the desired functionality?
- ……

Testing v.s. Verification

- Testing verifies manufacturing
  - Verify that the design was manufactured correctly

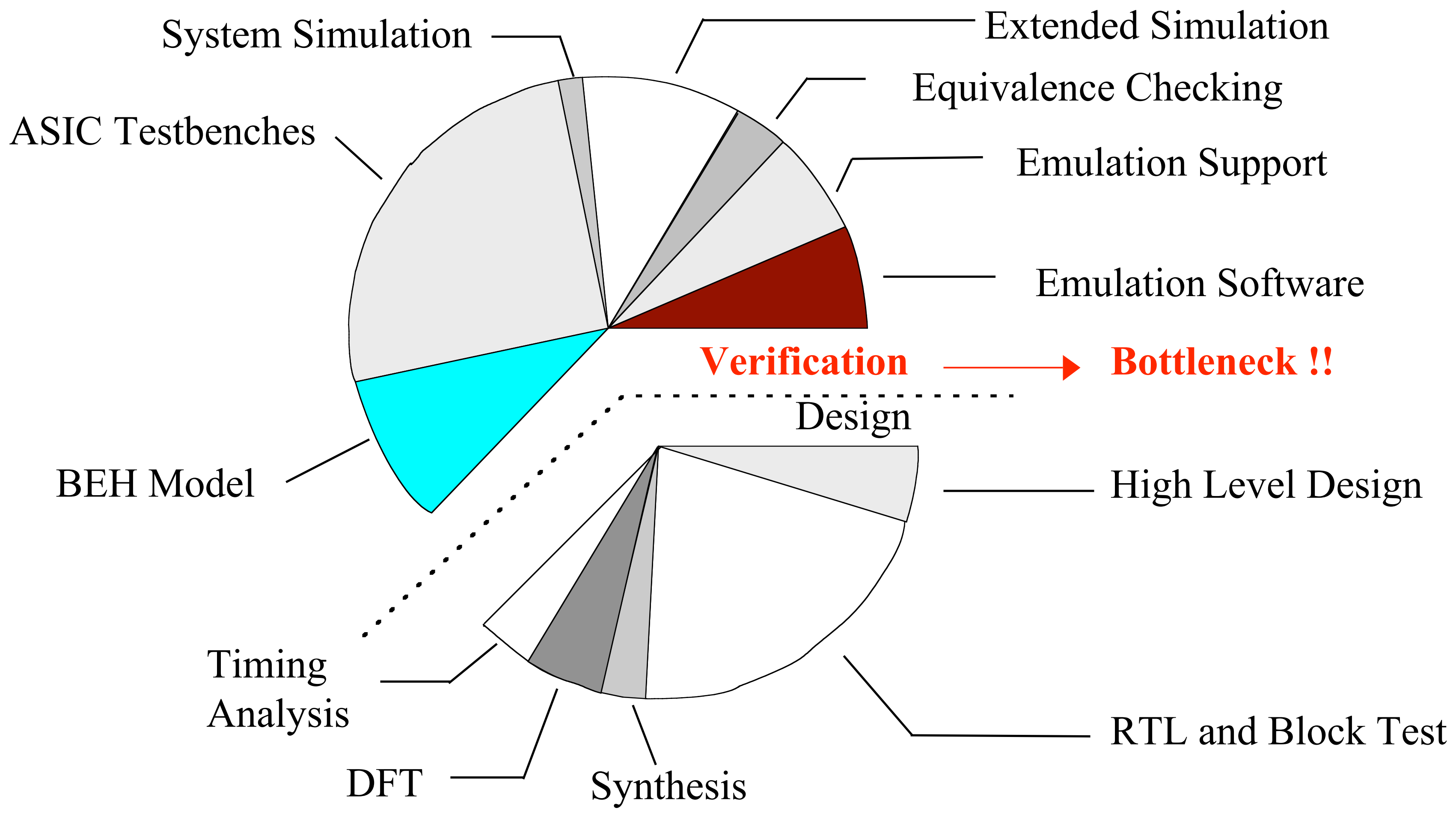


Source : "Writing Test Benches – Functional Verification of HDL Models" by Janick Bergeron, KAP, 2000.

SoC Design Verification

- Using pre-defined and pre-verified building block can effectively reduce the productivity gap
  - Block (IP) based design approach
  - Platform based design approach
- But **60 % to 80 %** of design effort is now dedicated to verification

An Industrial Example



Source : “Functional Verification on Large ASICs” by Adrian Evans, etc., 35th DAC, June 1998.

Verification Complexity

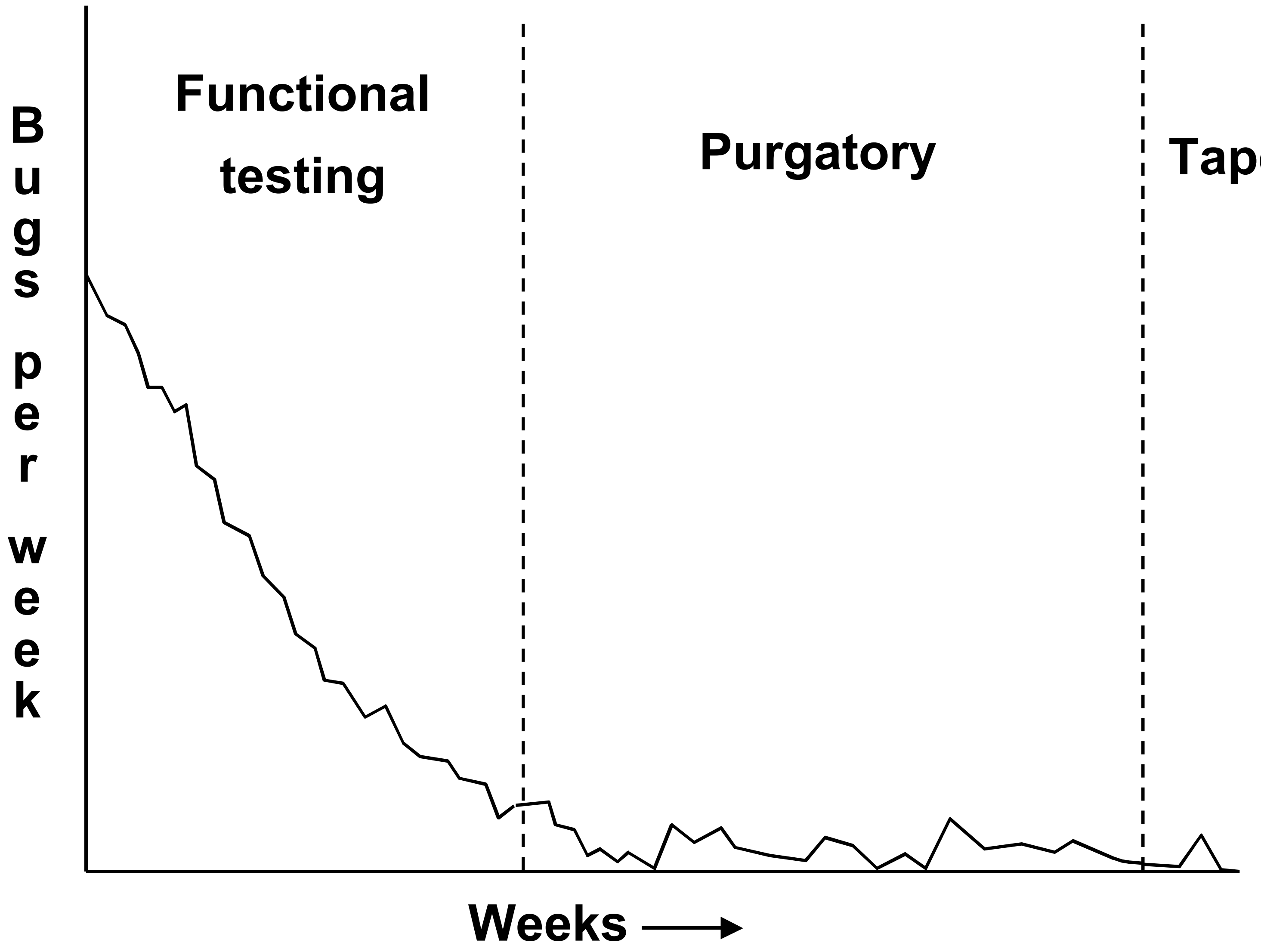
- For a single flip-flop:
  - Number of states = 2
  - Number of test patterns required = 4
- For a Z80 microprocessor (~5K gates)
  - Has 208 register bits and 13 primary inputs
  - Possible state transitions = $2^{\text{bits+inputs}} = 2^{221}$
  - At 1M IPS would take 10^{53} years to simulate all transitions
- For a chip with 20M gates
  - ??????

*IPS = Instruction Per Second

When is Verification Complete ?

- Some answers from real designers:
  - When we run out of time or money
  - When we need to ship the product
  - When we have exercised each line of the HDL code
  - When we have tested for a week and not found a new bug
  - ***We have no idea!!***
- Designs are often too complex to ensure full functional coverage
  - The number of possible vectors greatly exceeds the time available for test

Typical Verification Experience



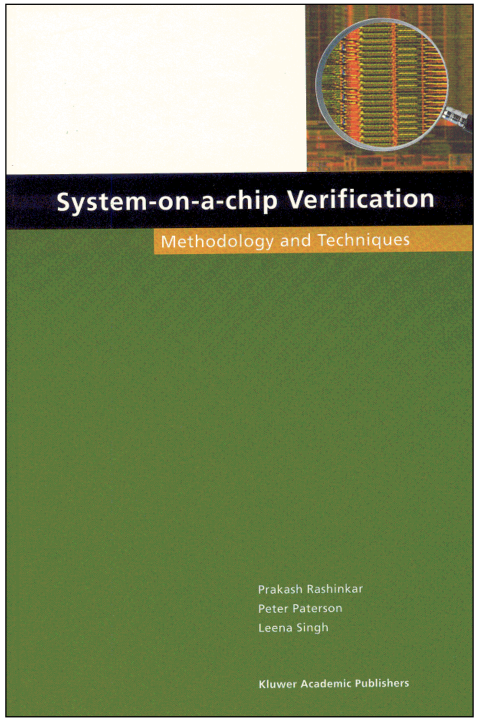
Error-Free Design ?

- As the number of errors left to be found decreases, the time and cost to identify them increases
- Verification can only show the presence of errors, not their absence
- Two important questions to be solved:
  - *How much is enough?*
  - *When will it be done?*

Reference Book

- **System-on-a-Chip Verification Methodology and Techniques**
- by
  Prakash Rashinkar
  Peter Paterson
  Leena Singh
  Cadence Design Systems Inc., USA
- published by

  Kluwer Academic Publishers, 2001


System-on-a-chip Verification
Methodology and Techniques
Prakash Rashinkar
Peter Paterson
Leena Singh
Kluwer Academic Publishers

Outline

- Verification Overview
- $\boxed{\text{Verification Strategies}}$
- Tools for Verification
- SoC Verification Flow

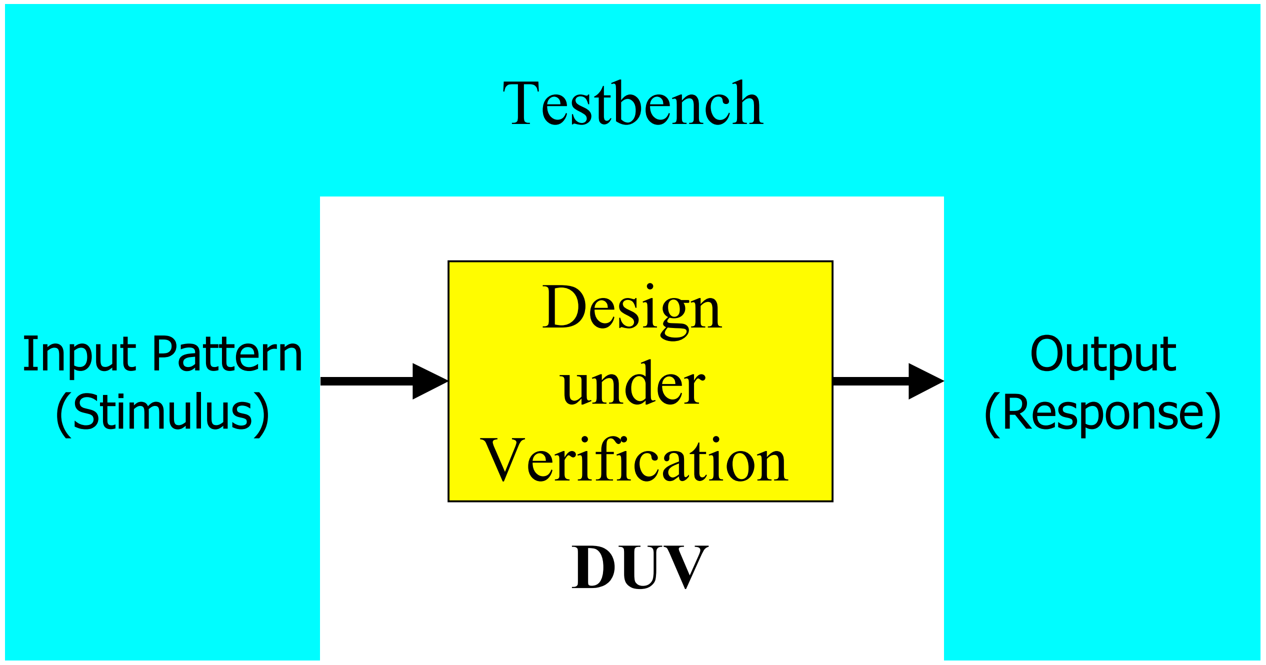
Verification Approaches

- Top-down verification approach
  - From system to individual components
- Bottom-up verification approach
  - From individual components to system
- Platform-based verification approach
  - Verify the developed IPs in an existing platform
- System interface-based verification approach
  - Model each block at the interface level
  - Suitable for final integration verification

Advs. of Bottom-up Approach

- Locality
- Catching bugs is easier and faster with foundational IPs (sub-blocks)
- Design the SoC chip with these highly confidence “bug-free” IPs

Verification Environment



Terminology

- Verification environment
  - Commonly referred as **testbench** (environment)
- Definition of a testbench
  - A verification environment containing a set of components [such as bus functional models (BFMs), bus monitors, memory modules] and the interconnect of such components with the design-under-verification (DUV)
- Verification (test) suites (stimuli, patterns, vectors)
  - Test signals and the expected response under given testbenches

Testbench Design

- Auto or semi-auto stimulus generator is preferred
- **Automatic response checking** is highly recommended
- May be designed with the following techniques
  - Testbench in HDL
  - Tesebench in programming language interface (PLI)
  - Waveform-based
  - Transaction-based
  - Specification-based

Types of Verification Tests (1/2)

- Random testing
  - Try to create scenarios that engineers do not anticipate
- Functional testing
  - User-provided functional patterns
- Compliances testing
- Corner case testing
- Real code testing (application SW)
  - Avoid misunderstanding the spec.

Types of Verification Tests (2/2)

- **Regression** testing
  - Ensure that fixing a bug will not introduce another bug(s)
  - Regression test system should be automated
    - Add new tests
    - Check results and generate report
    - Distribute simulation over multiple computer
  - Time-consuming process when verification suites become large

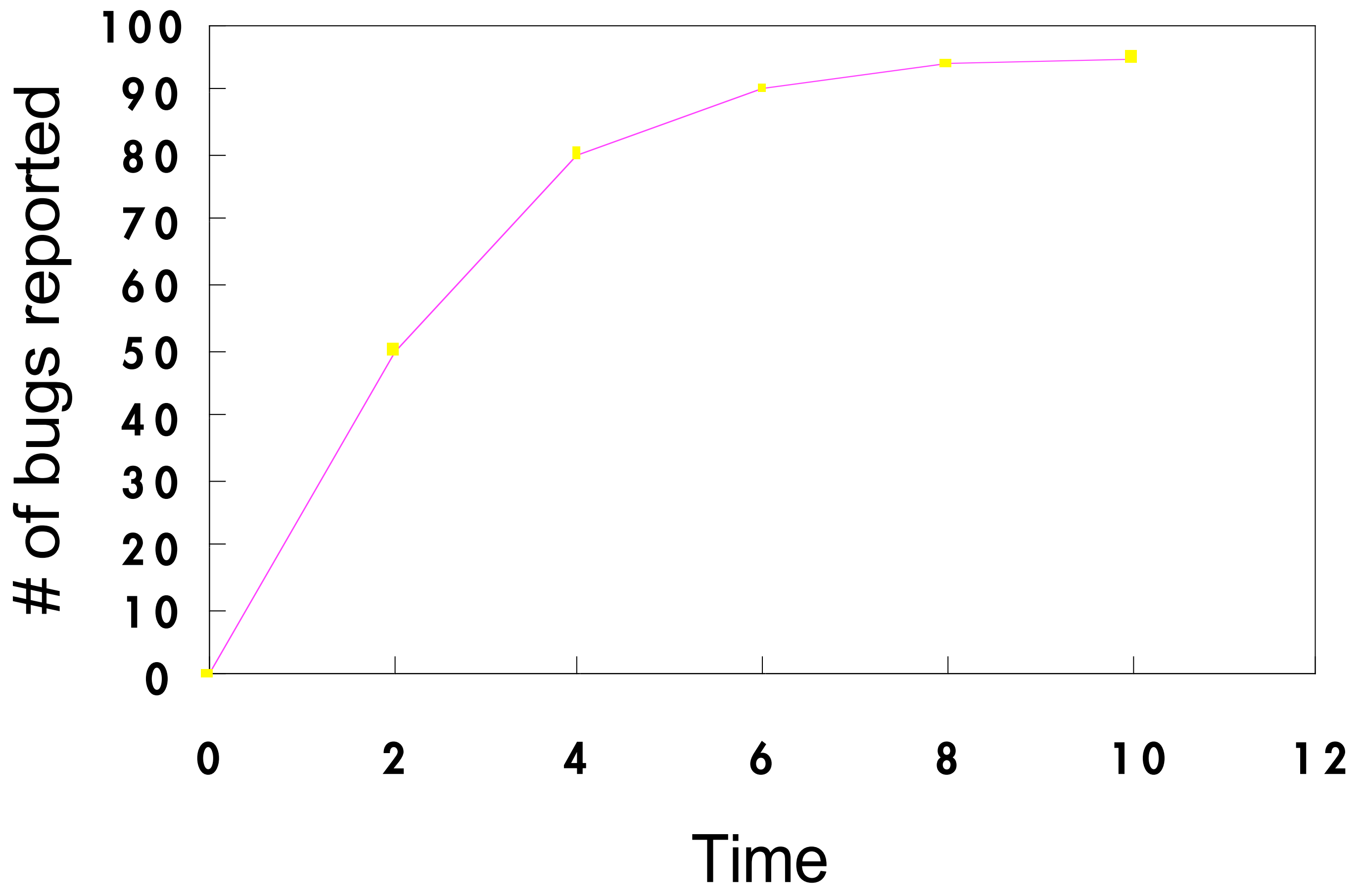
Bug Tracking

- A **central database** collecting known bugs and fixes
- Avoid debugging the same bug multiple times
- Good bug report system helps knowledge accumulation

Bug Rate Tracking

- Bug rate usually follow a well-defined curve
- The position on the curve decides the most-effective verification approach
- Help determine whether the SoC is ready to tape-out

Bug Rate Tracking Example



Adversarial Testing

- For original designers
  - Focus on proving the design works correctly
- Separate verification team
  - Focus on trying to prove the design is broken
  - Keep up with the latest tools and methodologies
- The balanced combination of these two gives the best results

Verification Plan

- Verification plan is a part of the design reports
- Contents
  - Test strategy for both blocks and top-level module
  - Testbench components – BFM, bus monitors, …...
  - Required verification tools and flows
  - Simulation environment including block diagram
  - **Key features** needed to be verified in both levels
  - Regression test environment and procedure
  - **Clear criteria** to determine whether the verification is successfully complete

Benefits of Verification Plan

- Verification plan enables
  - Developing the testbench environment early
  - Developing the test suites early
  - Developing the verification environment in parallel with the design task by a separate team
  - Focusing the verification effort to meet the product shipment criteria
  - Forcing designers to think through the time-consuming activities before performing them

Outline

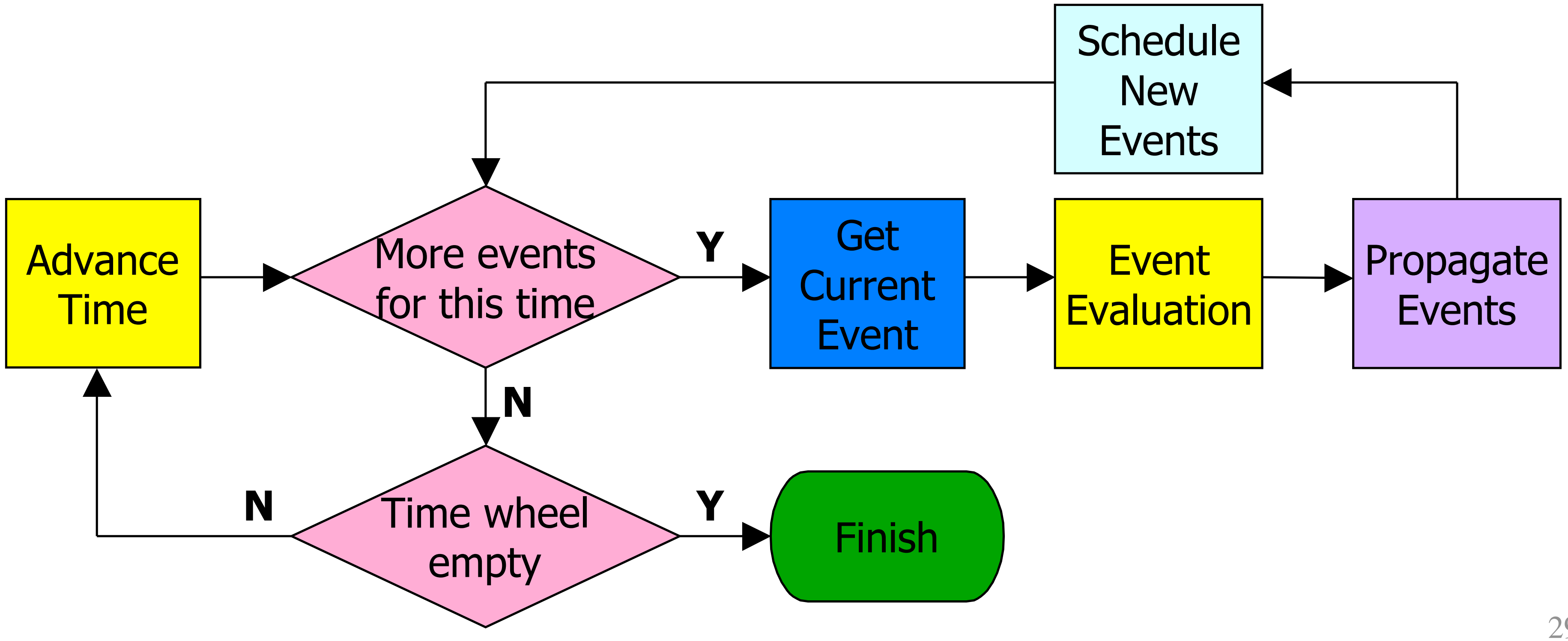
- Verification Overview
- Verification Strategies
- $\boxed{\text{Tools for Verification}}$
- SoC Verification Flow

Tools for Verification (1/4)

- Simulation
  - Event-driven:
    - Timing accurate
  - Cycle-based:
    - Faster simulation time (5x – 100x)
  - Transaction-based:
    - Require bus functional model (BFM) of each design
    - Only check the transactions between components
    - Have faster speed by raising the level of abstraction

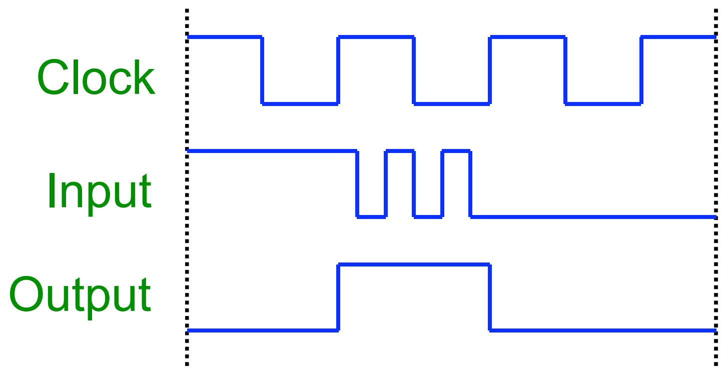
Event-Driven Simulation

- Timing accurate
- Good debugging environment
- Simulation speed is slower



Cycle-Based Simulation

- Perform evaluations just before the clock edge
  - Repeatedly triggered events are evaluated only once in a clock cycle



- Faster simulation time (5x – 100x)
- Only works for synchronous designs
- Only cycle-accurate
- Require other tools (ex: STA) to check timing problems

Simulation-Based Verification

- Still the primary approach for functional verification
  - In both gate-level and register-transfer level (RTL)
- Test cases
  - User-provided (often)
  - Randomly generated
- Hard to gauge how well a design has been tested
  - Often results in a huge test bench to test large designs
- Near-term improvements
  - Faster simulators
    - Compiled code, cycle-based, emulation, …
  - Testbench tools
    - Make the generation of pseudo-random patterns better/easier
- Incremental improvements won’t be enough

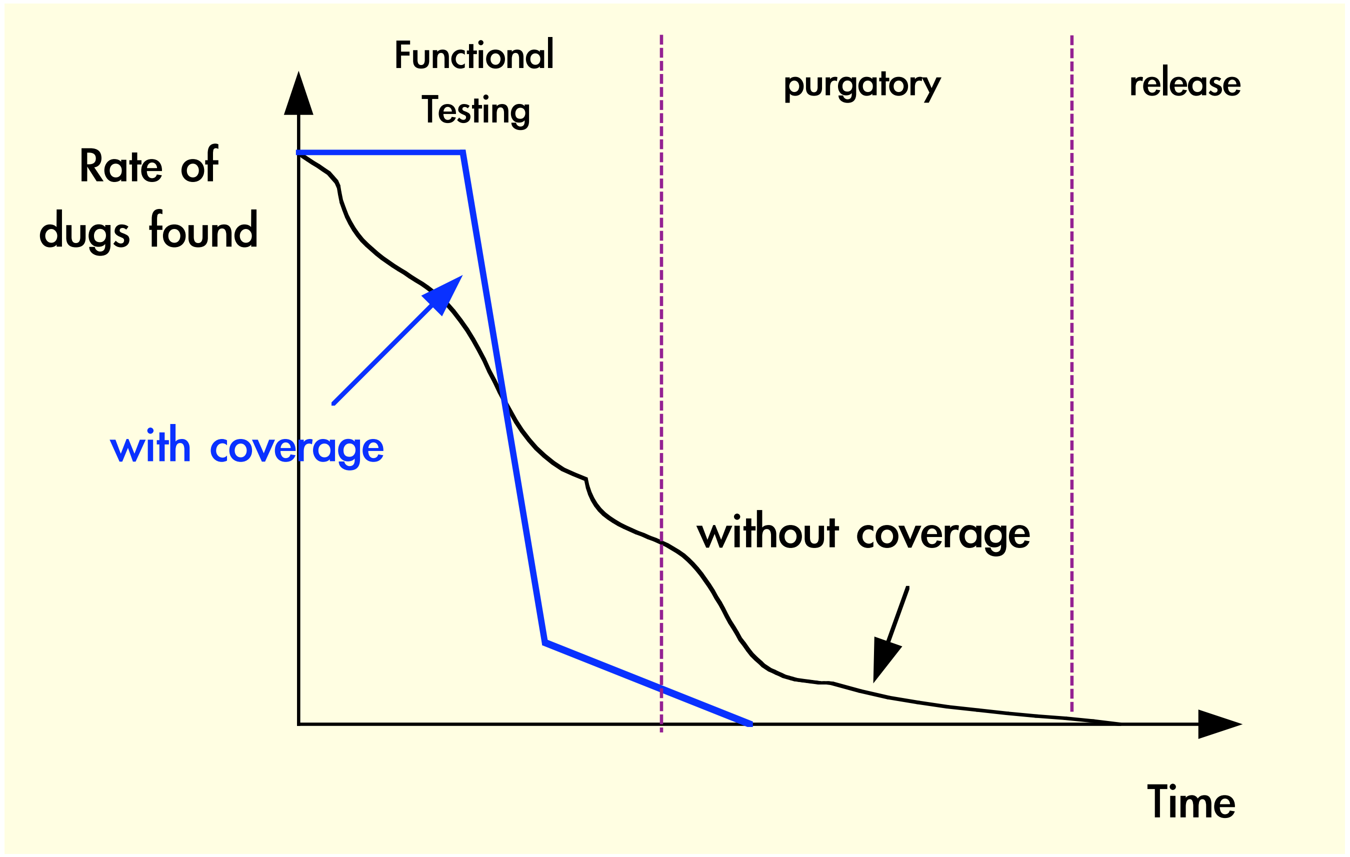
Tools for Verification (2/4)

- Code coverage
  - Qualify a particular test suite when applied to a specific design
  - Verification Navigator, CoverMeter, …
- Testbench (TB) automation
  - A platform (language) providing powerful constructs for generating stimulus and checking response
  - VERA, Specman Elite, …
- Analog/mixed-signal (AMS) simulation
  - Build behavior model of analog circuits for system simulation
  - Verilog-A, VHDL-A, …

Coverage-Driven Verification

- Coverage reports can indicate how much of the design has been exercised
  - Point out what areas need additional verification
- Optimize regression suite runs
  - Redundancy removal (to minimize the test suites)
  - Minimizes the use of simulation resources
- Quantitative sign-off (the end of verification process) criterion
- **Verify more but simulate less**

The Rate of Bug Detection



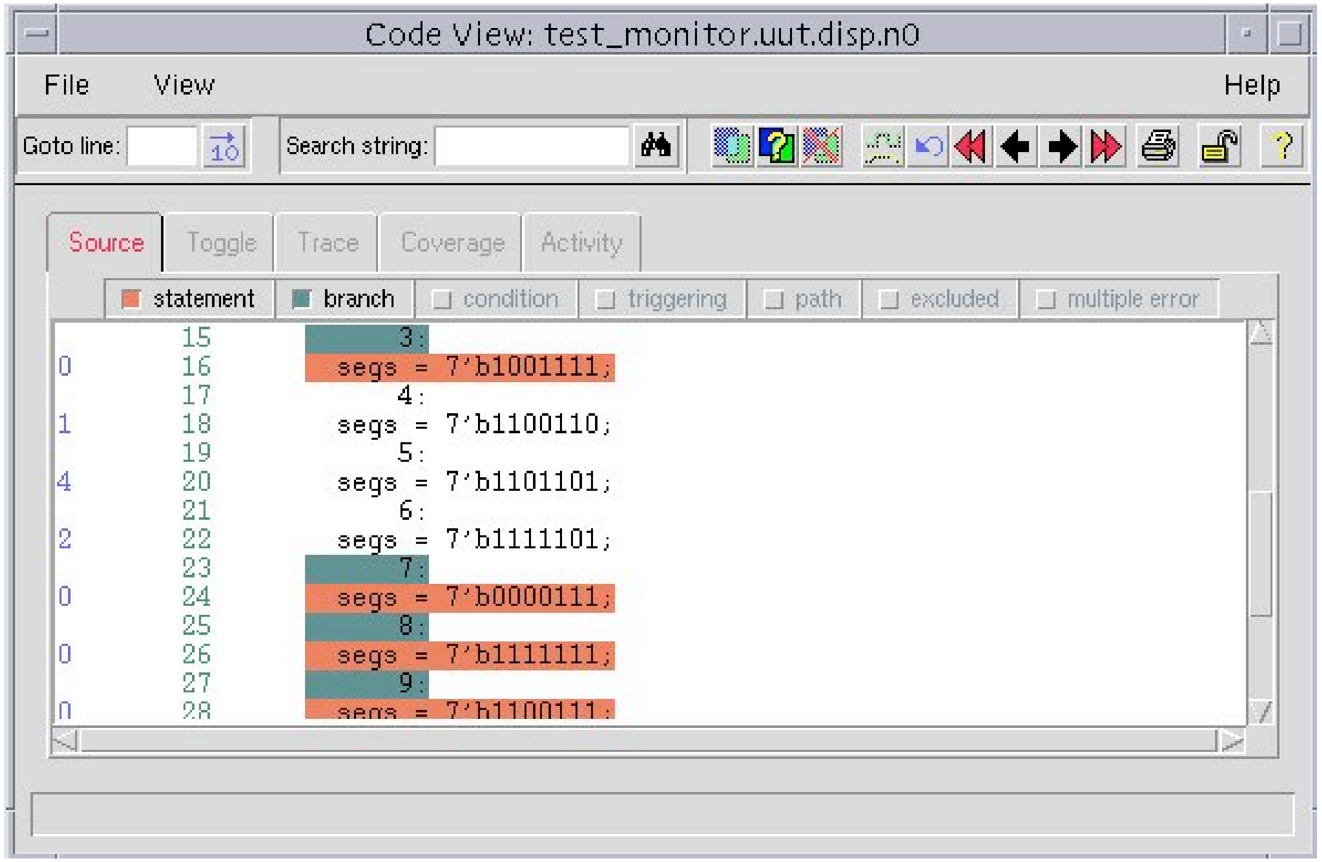
source : "Verification Methodology Manual For Code Coverage In HDL Designs" by Dempster and Stuart

Coverage Analysis

- Dedicated tools are required besides the simulator
- Several commercial tools for measuring Verilog and VHDL code coverage are available
  - VCS (Synopsys)
  - NC-Sim (Cadence)
  - Verification navigator (TransEDA)
- Basic idea is to monitor the actions during simulation
- Require supports from the simulator
  - PLI (programming language interface)
  - VCD (value change dump) files

Analysis Results

- Verification Navigator (TransEDA)



```
         3:
0        segs = 7'b1001111;
         4:
1        segs = 7'b1100110;
         5:
4        segs = 7'b1101101;
         6:
2        segs = 7'b1111101;
         7:
0        segs = 7'b0000111;
         8:
0        segs = 7'b1111111;
         9:
0        segs = 7'b1100111;
```

Untested code line will be highlighted

Testbench Automation

- Require both ***generator*** and ***predictor*** in an integrated environment
- Generator: constrained random patterns
  - Ex: keep A in [10 … 100]; keep A + B == 120;
  - Pure random data is useless
  - Variations can be directed by weighting options
  - Ex: 60% fetch, 30% data read, 10% write
- Predictor: generate the estimated outputs
  - Require a behavioral model of the system
  - Not designed by same designers to avoid containing the same errors

Analog Behavioral Modeling

- A mathematical model written in **Hardware Description Language**
- Emulate circuit block functionality by sensing and responding to circuit conditions
- Available Analog/Mixed-Signal HDL:
  – Verilog-A
  – VHDL-A
  – Verilog-AMS
  – VHDL-AMS

Mixed Signal Simulation

- Available simulators:

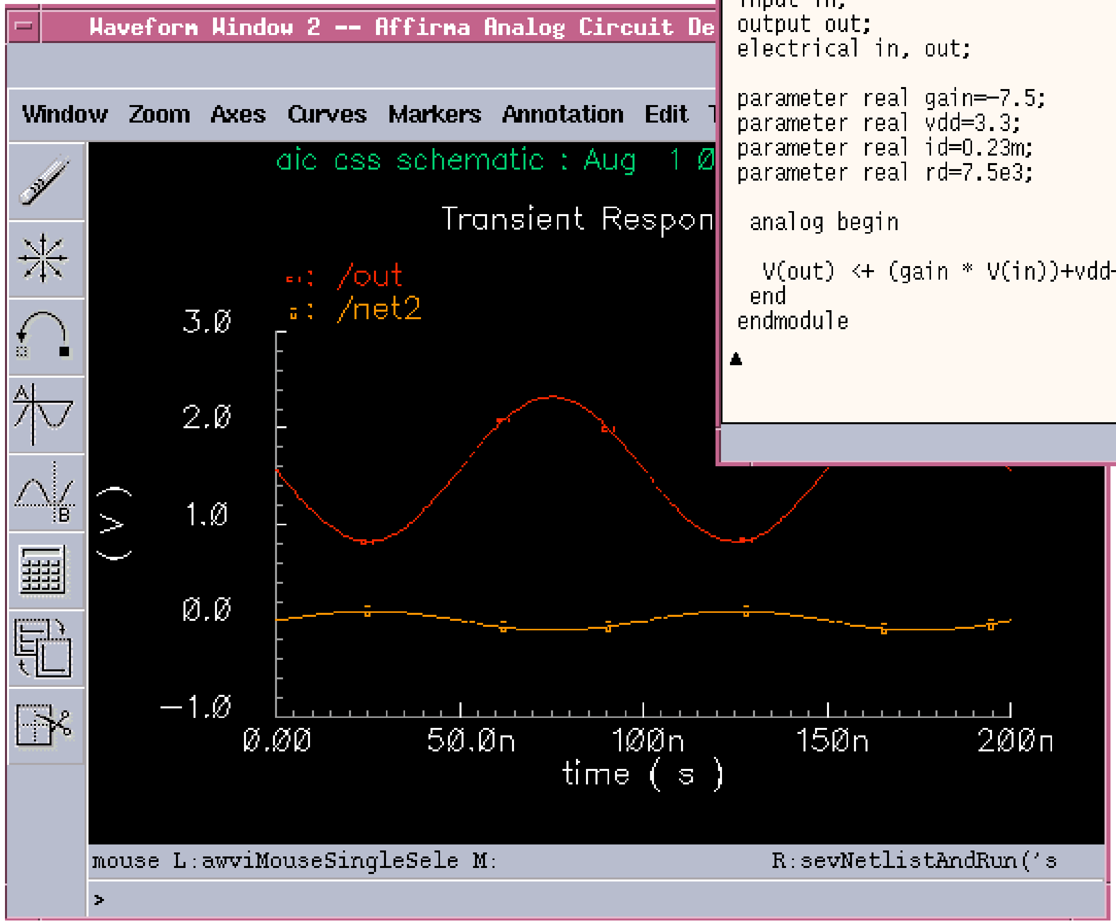
  Cadence

  Antrim

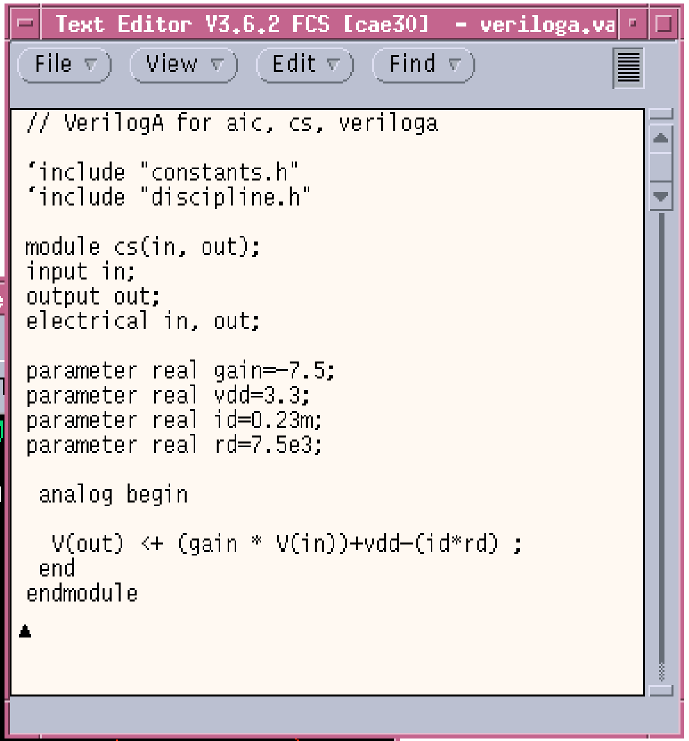
  Mentor

  Synopsys

  ……



```
mouse L:awviMouseSingleSele M:                R:sevNetlistAndRun('s
>
```



```
// VerilogA for aic, cs, veriloga

`include "constants.h"
`include "discipline.h"

module cs(in, out);
input in;
output out;
electrical in, out;

parameter real gain=-7.5;
parameter real vdd=3.3;
parameter real id=0.23m;
parameter real rd=7.5e3;

 analog begin

  V(out) <+ (gain * V(in))+vdd-(id*rd) ;
 end
endmodule
```

Tools for Verification (3/4)

- Static technologies
  - Lint checking:
    - A static check of the design code
    - Identify simple errors in the early design cycle
  - Static timing analysis:
    - Check timing-related problems without input patterns
    - Faster and more complete if applicable
  - Formal verification:
    - Check functionality only
    - Theoretically promise 100% coverage but design size is often limited due to high resource requirement

HDL Linter

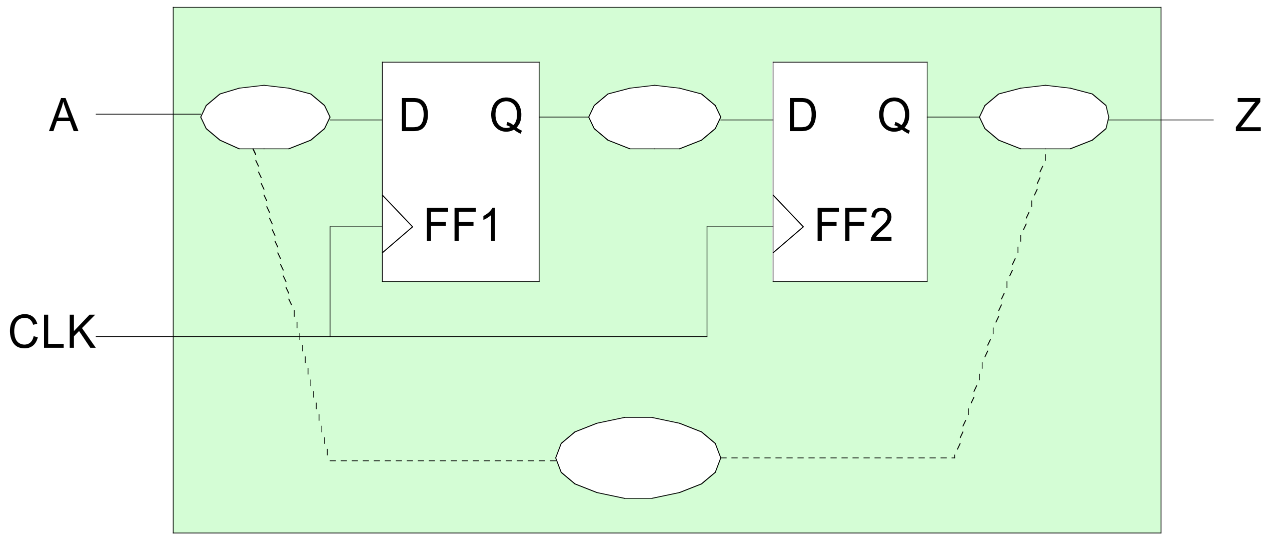
- Fast static RTL code checker
  - Preprocessor of the synthesizer
  - **RTL purification (RTL DRC)**
    - Syntax, semantics, simulation
  - Check for built-in or user-specified rules
    - Testability checks
    - Reusability checks
    - ……
  - Shorten design cycle
    - Avoid error code that increases design iterations

Inspection

- For designers, finding bugs by careful inspection is often faster then that by simulation
- Inspection process
  - Design (specification, architecture) review
  - Code (implementation) review
    - Line-by-line fashion
    - At the sub-block level
- Lint-liked tools can help spot defects without simulation
  - Nova ExploreRTL, VN-Check, ProVerilog, …

What is STA ?

- STA = static timing analysis
- STA is a method for determining if a circuit meets timing constraints without having to simulate
- No input patterns are required
  - 100% coverage if applicable

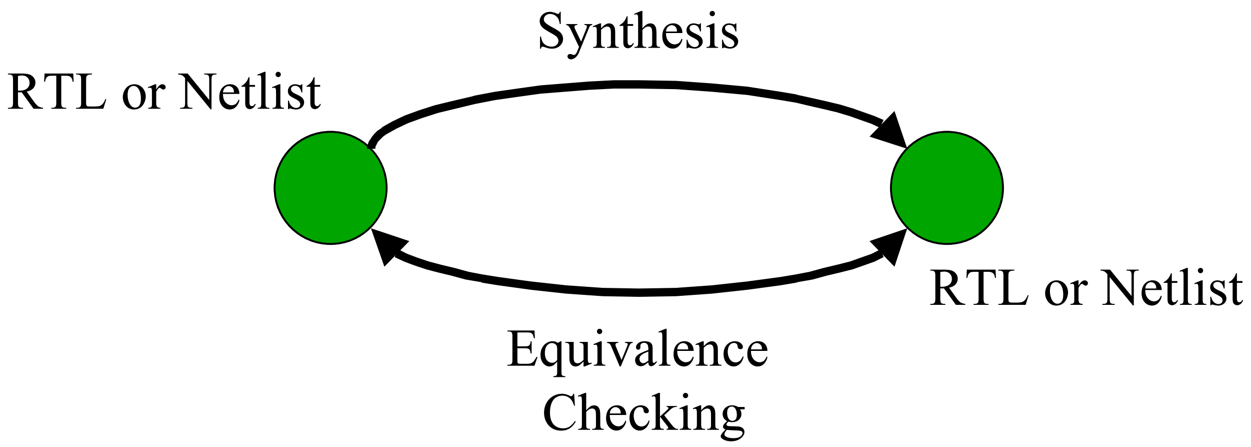


Reference :Synopsys

Formal Verification

- Ensure the consistency with specification for *all* possible inputs (100% coverage)
- Primary applications
  - Equivalence Checking
  - Model Checking
- Solve the completeness problem in simulation-based methods
- Cannot handle large designs due to its high complexity
- Valuable, but not a general solution

Equivalence Checking

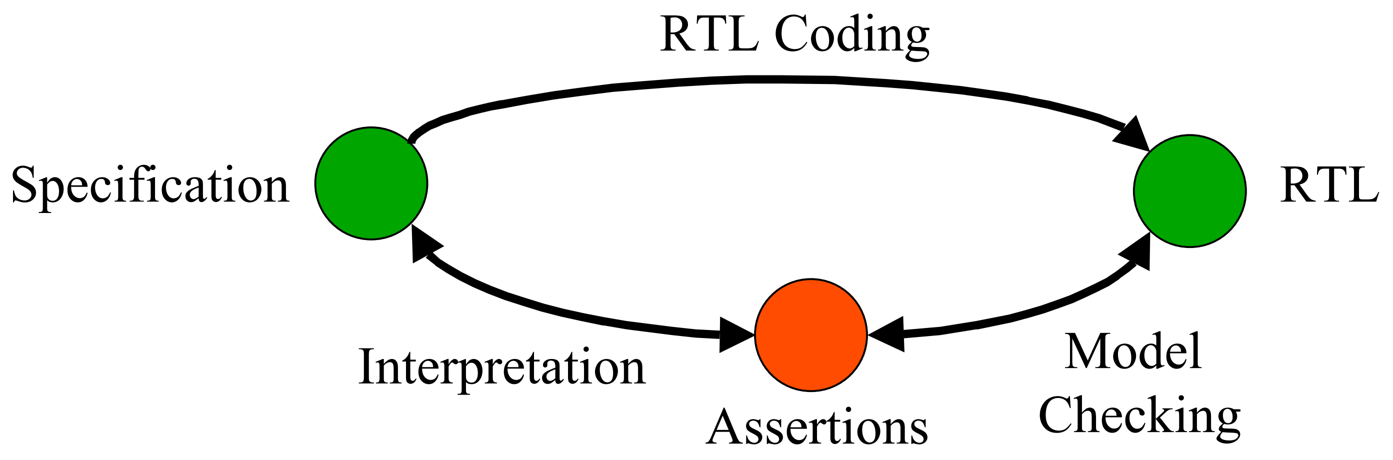


- Gate-level to gate-level
  - Ensure that some netlist post-processing did not change the functionality of the circuit
- RTL to gate-level
  - Verify that the netlist correctly implements the original RTL code
- RTL to RTL
  - Verify that two RTL descriptions are logically identical

Equivalence Checking

- Compare two descriptions to check their equivalence
- Gaining acceptance in practice
  - Abstract, Avant!, Cadence, Synopsys, Verplex, Verysys, …
- But the hard bugs are usually in both descriptions
- Target *implementation* errors, not design errors
  - Similar to check C v.s. assembly language

Model Checking



- Formally prove or disprove some assertions (properties) of the design
- The assertions of the design should be provided by users first
  - Described using a formal language

Model Checking

- Enumerates all states in the STG of the design to check those assertions
- Gaining acceptance, but not widely used
  - Abstract, Chrysalis, IBM, Lucent, Verplex, Verysys, …
- Barrier: low capacity (~100 register bits)
  - Require extraction (of FSM controllers) or abstraction (of the design)
  - Both tend to cause costly *false* errors

New Approaches

- The two primary verification methods both have their drawbacks and limitations
  - Simulation: time consuming
  - Formal verification: memory explosion
- We need alternate approaches to fill the productivity gap
  - Verification bottleneck is getting worse
- Semi-formal approaches may be the solution
  - Combine the two existing approaches
  - Completeness (formal) + lower complexity (simulation)

Tools for Verification (4/4)

- Hardware modeling
  - Pre-developed simulation models for other hardware in a system
  - Smart Model (Synopsys)
- Emulation
  - Test the system in a hardware-liked fashion
- Rapid prototyping
  - FPGA
  - ASIC test chip

Assistant Hardware

- Rule of thumb
  - 10 cycles/s for software simulator
  - 1K cycles/s for hardware accelerator
  - 100K cycles/s for ASIC emulator
- Hardware accelerator
  - To speed up logic simulation by mapping gate level netlist into specific hardware
  - Examples: IKOS, Axis, ….

Emulation

- Emulation
  - Verify designs using real hardware (like FPGA?)
  - Better throughput in handling complex designs
  - Inputs should be the gate-level netlist
  - Synthesizable testbenches required
  - Require expensive emulators
  - **Software-driven verification**
    - Verify HW using SW
    - Verify SW using HW
  - Interfaced with real HW components
  - Examples: Aptix, Quicktum, Mentor's AVS ….

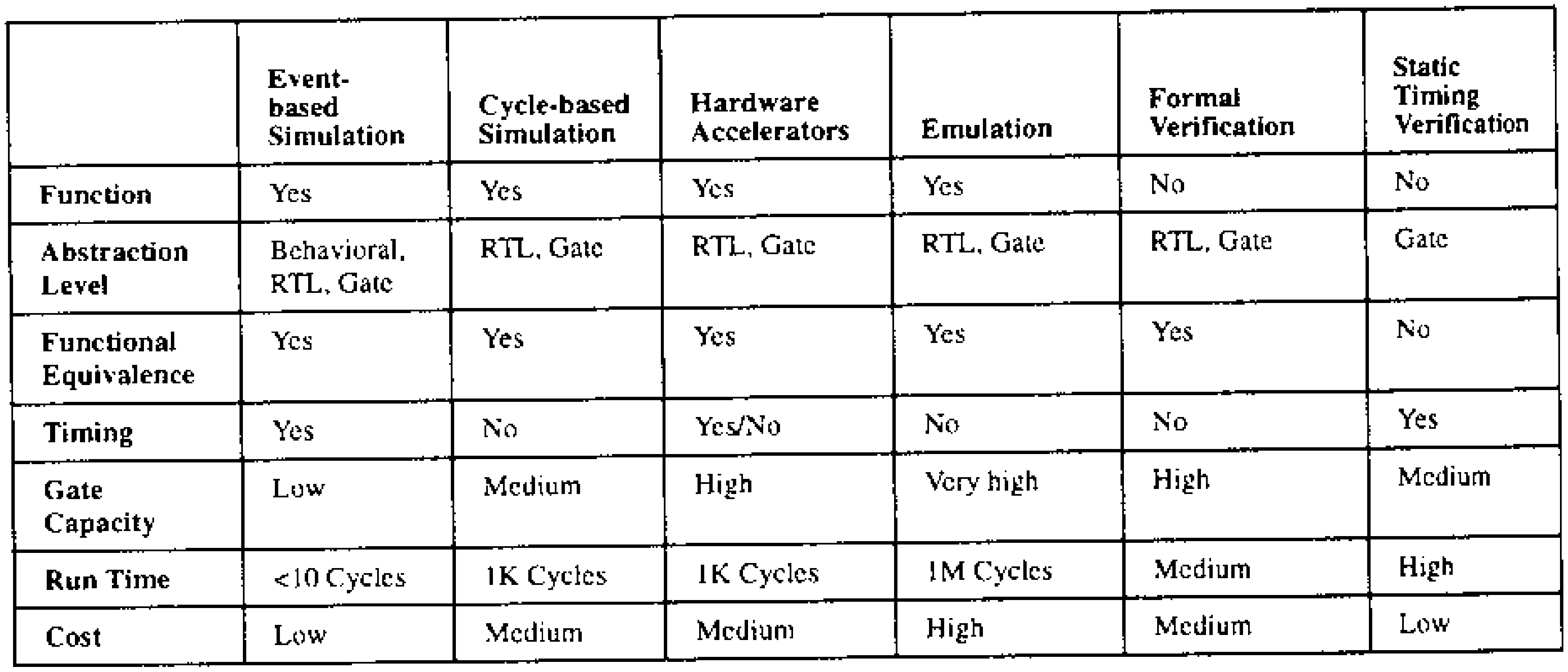
Prototyping

- FPGA
  - Performance degradation
  - Limited design capacity (utilization)
  - Partitioning and routing problems
- Emulation system
  - FPGA + Logic modeler
  - Automatic partitioning and routing under EDA SW control
  - More expensive

Limited Production

- Even after robust verification process and prototyping, it’s still not guaranteed to be bug-free
- Engineering samples
- A limited production for new macro is necessary
  - 1 to 4 customers
  - Small volume
  - Reducing the risk of supporting problems
- Same as real cases but more expensive

Comparing Verification Options



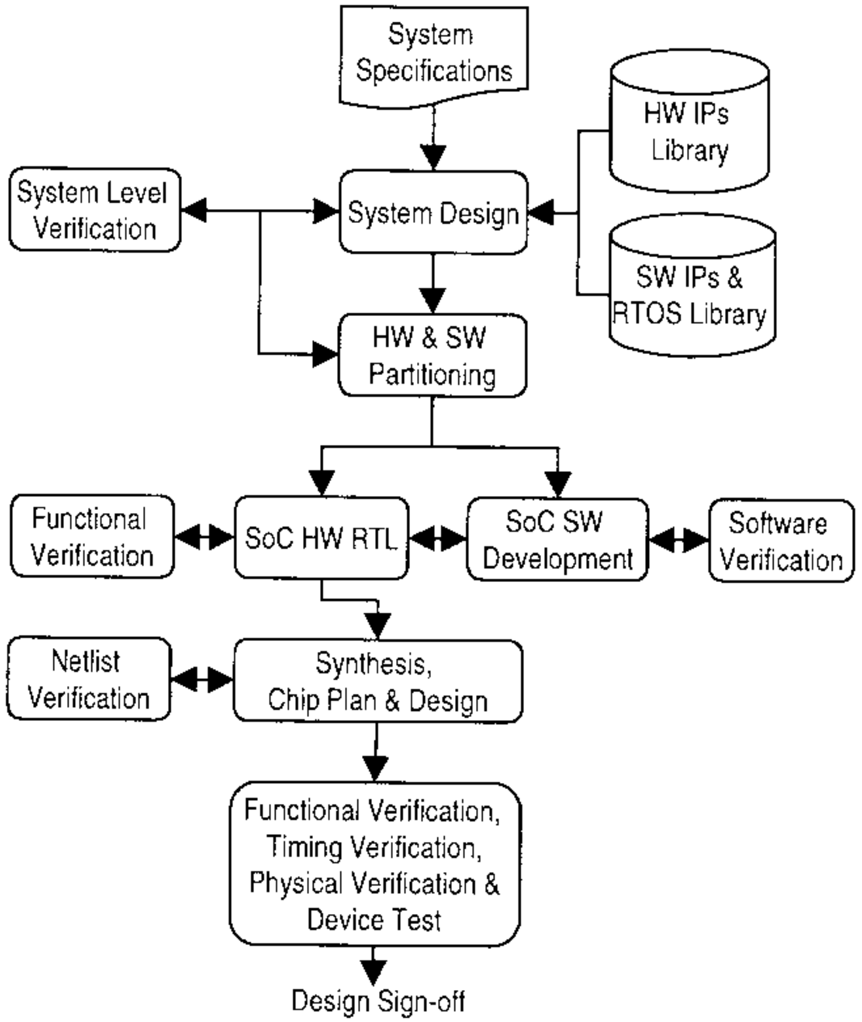
	Event-based Simulation	Cycle-based Simulation	Hardware Accelerators	Emulation	Formal Verification	Static Timing Verification
Function	Yes	Yes	Yes	Yes	No	No
Abstraction Level	Behavioral, RTL, Gate	RTL, Gate	RTL, Gate	RTL, Gate	RTL, Gate	Gate
Functional Equivalence	Yes	Yes	Yes	Yes	Yes	No
Timing	Yes	No	Yes/No	No	No	Yes
Gate Capacity	Low	Medium	High	Very high	High	Medium
Run Time	<10 Cycles	1K Cycles	1K Cycles	1M Cycles	Medium	High
Cost	Low	Medium	Medium	High	Medium	Low

Source : “System-on-a-chip Verification – Methodology and Techniques” by P. Rashinkar, etc., KAP, 2001.

Outline

- Verification Overview
- Verification Strategies
- Tools for Verification
- $\boxed{\text{SoC Verification Flow}}$

SOC Verification Methodology



Source : "System-on-a-chip Verification – Methodology and Techniques" by P. Rashinkar, etc., KAP, 2001.

System Verification Steps

- Verify the leaf IPs
- Verify the **interface** among IPs
- Run a set of complex applications
- Prototype the full chip and run the application software
- Decide when to release for mass production

Interesting Observations

- 90% of ASICs work at the first silicon but only 50% work in the target system
  - Do not perform system level verification (with **software**)
- If a SoC design consisting of 10 blocks
  - P(work)= $.9^{10}$ =.35
- If a SoC design consisting of 2 new blocks and 8 pre-verified robust blocks
  - P(work) = $.9^{2} * .98^{8}$ =.69
- To achieve 90% of first-silicon success SoC
  - P(work) = $.99^{10}$ =.90

Interface Verification

- Inter-block interfaces
  - Point-to-point
  - On-chip bus (OCB)
  - **Simplified models** are used
    - BFM, bus monitors, bus checkers

Check System Functionality (1/2)

- Verify the whole system by using full functional models
  - Test the system as it will be used **in the real world**
- Running real application codes (such as boot OS) for higher design confidence
  - RTL simulation is not fast enough to execute real applications

Check System Functionality (2/2)

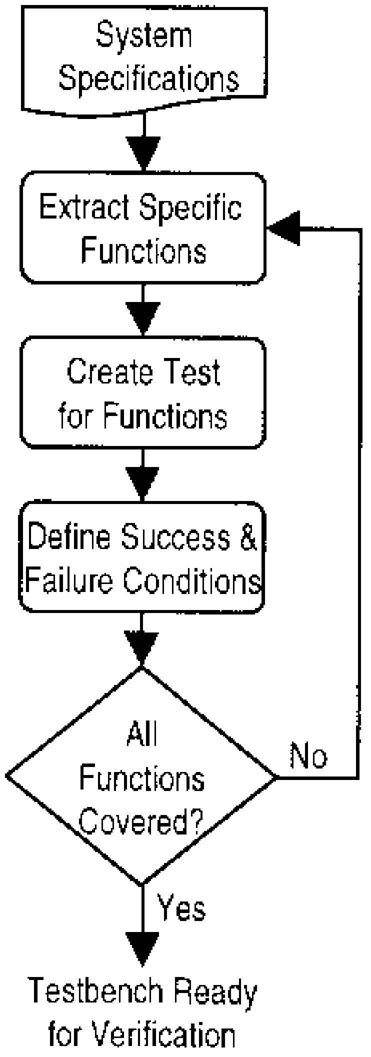
- Solutions
  - Move to a higher level of abstraction for system functional verification
  - Formal verification
  - Use assistant hardware:
    - Hardware accelerator
    - ASIC emulator
    - Rapid-prototyping(FPGA)
    - Logic modeler
    - …

HW/SW Co-Simulation

- Couple a software execution environment with a hardware simulator
- Simulate the system at higher levels
  - Software normally executed on an **Instruction Set Simulator** (ISS)
  - A **Bus Interface Model** (BIM) converts software operations into detailed pin operations
- Allows two engineering groups to talk together
- Allows earlier integration
- Provide a significant performance improvement for system verification
  - Have gained more and more popularity

System-Level Testbench

- All functionality stated in the spec. should be covered
- The success and failure conditions must be defined for each test
- Pay particular attention to:
  - Corner cases
  - Boundary conditions
  - Design discontinuities
  - Error conditions
  - Exception handling


System Specifications
Extract Specific Functions
Create Test for Functions
Define Success & Failure Conditions
All Functions Covered?
No
Yes
Testbench Ready for Verification

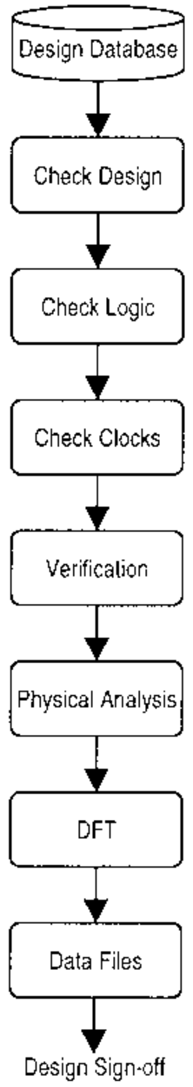
Source : "System-on-a-chip Verification – Methodology and Techniques" by P. Rashinkar, etc., KAP, 2001.

Timing Verification

- STA (static timing analysis) is the fastest approach for synchronous designs
  - Avoid any *timing exceptions* is extremely important
- Gate-level simulation (dynamic timing analysis)
  - Most useful in verifying timing of asynchronous logic, multi-cycle paths and false paths
  - Much slower run-time performance
  - Gate-level sign-off

Design Sign-off

- Sign-off is the final step in the design process
- It determines whether the design is ready to be taped out for fabrication
- No corrections can be made after this step
- The design team needs to be confident that the design is 100% correct
  - Many items need to be checked



Source : “System-on-a-chip Verification – Methodology and Techniques” by P. Rashinkar, etc., KAP, 2001.

Traditional Sign-off

- Traditional sign-off simulation
  - Gate level simulation with precise timing under a given parallel verification vectors
  - Verify functionality and timing at the same time (dynamic timing analysis, DTA)
  - Parallel verification vectors also serve as the manufacturing test patterns
- Problems
  - Infeasible for million gates design (take too much simulation time)
  - Fault coverage is low (60% typically)
  - Critical timing paths may not be exercised

SoC Sign-off Scheme

- **Formal verification** used to verify functionality
- **STA** for timing verification
- Gate level simulation
  - Supplement for formal verification and STA
- Full scan BIST for manufacturing test