What is verification?

Verification: A process that determines if the design conforms to the specification.

It answers the question, ”Does the design do what is intended?”

It is more than a testbench.

Applies to both hardware and software.
Why do verification as early as possible?

Bugs become much more costly, the longer they go unfound.

Bugs found at unit level can often be fixed cheaply.
Bugs found at the system-level may affect the time-to-market.
Bugs found after fabrication require an expensive respin.
Bugs found by customers can result in potentially company crushing recalls and a bad reputation.
Who does verification?

Designers may begin the process.
Verification engineers manage and complete the process.
Verification engineers may outnumber designers 2-1.
Verification cost may dominate the overall cost of a project.
Is this good or bad?
“Validation is so complex that even though it consumes the most computational resources and time it is still the weakest link in the design process. Ensuring functional correctness is the most difficult part of designing a hardware system.”
–Serdar Tasiran and Kurt Keutzer
Why is verification hard?

Getting a detailed specification of correct behavior may be very difficult.

Proving conformance between the system and its specification is usually impossible.

For example, Windows XP specification may say “system is secure from network intrusion”

Verification tools may be buggy.

Often we settle for weak forms of verification: system seems to conform to specification in cases we looked at.
Common types of verification

- Functional verification.
  - Simulation.
    - Directed tests.
    - Constrained pseudorandom tests.
  - Formal methods.
    - Model checking.
    - Equivalence checking.
    - Automated theorem proving.

- Timing verification.

- Performance verification.
Functional verification is primarily done via simulation.

- Black-box.
- White-box.
- Gray-box.
Black-box Verification

The verifier has access to input, outputs, and device function. Given a set of inputs the verifier checks for correct outputs. To fully verify a black-box you must show that function is correct for all combinations of inputs. Full verification via black-box testing is impractical for any real design.
The verifier has access to (and uses) internal signals during verification.
This is common during block-level verification.
The verifier has access to (and uses) a limited number of internal signals during verification. This is the reality for most verification. Predicting the correct output value without viewing an internal signal is often difficult. Knowing the architecture of the DUT (device under test) enables you to write better tests.
Does the verification cover all possible inputs?
How can the verifier detect a failure?
Three Commandments of Simulation

Thou shalt stress thine logic harder than it will ever be stressed again.
Thou shalt place checking upon all things.
Thou shalt not move onto a higher platform until the bug rate has dropped off.
Independent Verification is Key

The verification engineer should not participate in the logic design of the DUT.

Designers may not think of all failing scenarios.

Verification engineers have a different perspective on the design.

Verification engineers must understand the function, but not the implementation.
Verification Do’s

Talk to the designer to understand the function of the design.
Think of situations the designer neglected to consider.
Focus on the corner cases or exotic scenarios.
Focus on concurrent events.
Try everything that isn’t explicitly forbidden.
Think about the all the pieces of the design you need to verify.
Talk to designers regarding the interface to your DUT.
Verification Don’ts

Don’t take the designer’s word for anything.
Don’t weaken your test plan in order to meet a time schedule.
A Typical Verification Flow

Devise a potential bug.
Write a test bench to expose the bug.
Run the simulator with the test bench.
Check the simulation result.
If the test has uncovered a bug:
  Verify the bug.
  Work with the designer to fix the bug.
To be practical this flow needs to be automated.
Devising spots for potential bugs is best not automated.
Determining which parts of the design have been explored can be automated...we will get to that later.
Writing test benches to expose bugs can be automated via constrained pseudorandom testing.
Constrained Pseudorandom Testing

Produces test cases that would be difficult to generate by hand.
Facilitates the generation of combinations of concurrent events that would be difficult to explicitly devise.
Often done using a specialized language like Specman’s e.
Good verification engineers are still needed to efficiently drive the constraints and develop the tests.
How do you check the correctness of a pseudorandom test?
   Reference models.
   Assertions.
Reference Models

An abstraction of the design implementation. Should be fast, correct, and represent all the design details. In practice, reference models are fast and correct yet lack detail. The simulator and the reference model use the same stimulus. The final result of each is compared. It can be useful to do intermediate comparisons for long tests.
Two types:

- Built-in checkers.
- Post-processing checkers.

Built-in checkers are always running during every simulation. Anyone can add them and everyone reaps the benefits. Built-in checkers can increase simulation time. Complex checkers (computationally intensive) can be moved to post-processing checkers and only run on selected tests.
Constrained pseudorandom tests cover many cases. Which cases do they cover? How many more vectors should I run? Coverage analysis provides metrics to answer these questions.
Coverage Metrics

Serve two main purposes:

Act as heuristic measures that quantify verification completeness.
Identify inadequately exercised design aspects and guide future input stimulus generation.
Theory vs. Reality

Theory: Increasing the coverage increases confidence in the design’s correctness.

Reality: At best, there is an intuitive connection between coverage metrics and bugs.

Design errors are more difficult to characterize.

A formal error model for design bugs hasn’t been found.
Inputs guided by coverage information commonly detect more bugs than conventional methods.
Coverage metrics provide a better measure of verification adequacy than bug statistics alone.
Requirements

Measuring a coverage metric must be efficient.
Generating inputs that improve coverage should not require an unreasonable effort.
Measuring coverage should not require large changes in tools and methodology.
Code Coverage Metrics

Line: Is every line of code executed?
Branch: Is the positive and negative of every branch taken?
Expression: Are all legal expression values executed?
Path: Are all paths in the CFG executed?
Measurement is low overhead.
Achieving full code coverage for HDLs is a minimum requirement.
Hardware is highly concurrent.
Code coverage doesn’t address the concurrency aspect.
Toggle coverage: How many bits toggle (0→1 and 1→0)?
More sophisticated structure metrics involve the division of data path and control.
Many more sophisticated method are structure specific.
Registers should test initialization, loads, and reads.
Counters should be tested for their min and max values.
FSMs for the design are obtained in two ways:
   Hand written at a high level of abstraction.
   Auto-extracted from the design description.

Given an FSM some coverage metrics are:
   State.
   Transition.
   Path.

FSMs for the entire design are prohibitively large, so only portions of the design space are covered using FSM metrics.
Maintaining FSM accuracy during design evolution is difficult. Writing coverage-directed tests for FSMs is very difficult. FSM state variable may be deep within the design making it nonobvious how to toggle them.
Typically consist of design-specific classes of errors.

- Bus protocols → Exercise specific transactions.
- Pipelines → Check hazard logic.

Monitors are often used to check for specific functional bugs.

Commercial test suites are available for functional verification of industry standard protocols.
Mathematically-based techniques use to prove that the design conforms to a given specification.

Powerful results.

Computationally intensive.

In practice, often not used to prove systems correct but to find “deep” or difficult bugs.
Exhaustively explores the state space of the design to prove that the system model conforms to the given specification. State space exploration is an exponential problem. Many techniques have been developed to help combat the state space explosion problem.

- Bounded model checking.
- Partial order reduction.
- Counter-Example Guided Abstraction Refinement (CEGAR).
Different state space representation methods also attempt to combat the state space explosion problem.

- Explicit state.
- BDDs.
- SAT.
- etc.

Writing a complete and correct specification is key.
Historically, have been based on temporal logics (LTL, CTL). Temporal logics are notoriously difficult to use efficiently. Industry standard property specification languages have been developed that are easier to use.

PSL/Sugar.
SVA.

PSL Examples:
assert always CONDITION;
assert always a;b |− > c;d;
Equivalence Checking

Proves that two different representations of the circuit have exactly the same behavior.
Traditionally done between RTL and the netlist.
RTL is considered to be the golden or reference model.
Equivalence checking ensures that the synthesis and other DFM related changes don’t introduce any errors to the circuit.
Automated Theorem Proving

Proving mathematical theories with the help of a computer. Automated theorem provers are notoriously difficult to use. Expert users can provide very strong results using these tools. Results are often at very high levels of abstraction.
In many cases, creating the hardware/software is not the hard part.
Hard part is ensuring high enough quality that the product can be successful.