VERIFICATION: OUTLINE

- Introduction
- Static timing analysis
- Coding style check
- Equivalence checking for combinational logic
- Code coverage
- Assertions
- Classical simulation
- Transaction-level modeling
- Advanced simulation

FURTHER READING


TERMINOLOGY (1)

- **Verification**: make sure that the design is correct before tape-out, that no design errors have been made.
- Not to be confused with:
  - **Evaluation**: checking the correctness of the hardware, including those features that could not be verified because of limitations of verification (simulation costs time and is not always accurate as far as analog design is concerned).
  - **Characterization**: a more systematic version of evaluation directed to determine the operational limits of a design (voltage levels, temperature range, maximal clock frequency, etc.). Can take days, weeks.

TERMINOLOGY (2)

- Also not to be confused with:
  - **Testing**: making sure that nothing went wrong during production; targeted to detect as many errors as possible in as short as possible time (tester time is expensive).
  - Note: in the field of software the term **testing** is used for verification, for detecting design errors; also the word **testbench** used in hardware verification is misleading.
IMPORTANCE OF VERIFICATION

• Production masks are expensive (one set of masks costs in the order of $100,000).
• The production time is long (1 to 2 months); redesigns significantly delay the time to market.
• Sometimes errors can be fixed by replacing a subset of the masks. Such a reparation is called a metal fix.
• The possibility of replacing a subset of masks can be exploited in design, e.g. to change the contents of a ROM.
• In order to shorten the production cycle, one can put aside wafers halfway, e.g. before metallization.

METHODS OF VERIFICATION

• Two aspects:
  – Check for design consistency (no reference design necessary)
  – Check for conformance with some reference design
• Examples of design consistency checks:
  – Static timing analysis
  – Coding style check
• Conformance check by means of:
  – Simulation
  – Formal methods

STATIC TIMING ANALYSIS

• Based on delay information as obtained after:
  – Logic synthesis, or
  – Layout (much more accurate)
• Standardized delay formats include:
  – SDF (Standard Delay Format): gives time values
  – SPEF (Standard Parasitic Extended Format): gives R and C values
• Used for a multitude of checks:
  – Clock-frequency requirements
  – Setup-time and hold-time requirements
  – Clock-skew requirements
• Extensively used in practice, e.g. Synopsys Primetime

SETUP AND HOLD TIMES (1)

• Consider the positive edge-triggered flipflop.
• The data signal should be stable around the rising clock edge:
  – Setup time: safety margin before rising edge
  – Hold time: safety margin after rising edge
SETUP AND HOLD TIMES (2)

- Check for setup-time violation requires computation of *longest path* through the logic (from primary input or flipflop output to flipflop input).
- Check for hold-time violation requires *shortest path*.

CLOCK SKEW (1)

- Clock must arrive at all memory elements in time to load data.
- The maximum difference between clock arrival times is the clock skew.

CLOCK SKEW (2)

- Clock skew values larger than the flip-flop input-output delay lead to malfunctioning: some computations will be based on the next state rather than the current state.

CLOCK TREE

- In order to balance the delay from the clock source to the flip-flops, clock trees are used.
- In current-day practice clock trees are generated during layout as wiring delay is significant.
CODING STYLE CHECK

- Analyze RTL code (e.g. VHDL) for issues as:
  - Clocking all flipflop on the same edge (rising or falling)
  - Consistent use of either asynchronous or synchronous resets
  - Combinational logic defined for all combinations of conditions (forgetting an “else” branch in an “if statement” leads to an unintended latch after synthesis)
  - Primary outputs directly connected to flipflop outputs (to avoid glitching)
- Used in practice, e.g. Synopsys Leda, Atrenta tools

CONFORMANCE CHECKING

- What is the specification?
  - Formal specification, expressed in appropriate language (not yet common practice)
  - Informal document, written in natural language (practice)
- How to check?
  - Using formal methods
  - By means of simulation

INTRODUCTION TO FORMAL VERIFICATION

- Simulation is a limited method for verification of design correctness:
  - The number of simulation patterns grows exponentially with the number of input and memory bits.
- Mathematics provides methods to prove properties of even infinite sets without the need to enumerate all elements of a set (think of proofs by induction).
- Formal verification is about applying mathematical techniques to prove the correctness of a design (without or with simulation).

EQUIVALENCE CHECKING FOR COMBINATIONAL LOGIC (1)

- One of the formal verification techniques that has reached sufficient maturity for industrial application (e.g. Synopsys tool Formality)
- It compares two circuits at the register-transfer (RT) or gate level and tries to prove equivalence: the reference and the implementation.
EQUIVALENCE CHECKING FOR COMBINATIONAL LOGIC (2)

- Main steps:
  - Identify all primary outputs and flipflop inputs in reference and implementation: each is a Boolean function.
  - Match them: e.g. by name.
  - Show equivalence of each matched pair of functions by converting the function to a canonical representation.
  - If all matched pairs are equivalent, the reference and implementation are equivalent.

APPLICATIONS OF EQUIVALENCE CHECKING

- Show equivalence between pre-synthesis and post-synthesis description (the synthesis tool may have flaws, you may have used non-synthesizable constructs).
- Show equivalence between word-level and bit-level descriptions of the same design.
- Show equivalence of circuit before and after hierarchy manipulation (move a module across hierarchy).

REDUCED-ORDERED BINARY DECISION DIAGRAMS (ROBDD)

- Canonical and compact representation of Boolean functions.
- Relatively easy to manipulate.
- Conceptual starting point is truth table represented as a tree.
- Variables have a fixed ordering from top to bottom.

- Follow dashed edge when variable is false and solid edge when variable is true

ROBDD COMPACTION RULES (1)

- Replace all leaf vertices with identical value (‘0’ or ‘1’) with a single vertex and redirect all incoming inputs to it.
ROBDD COMPACTION RULES (2)

- Process all vertices from top to bottom: merge vertices with identical variables and identical children.

ROBDD COMPACTION RULES (3)

- Remove vertices with identical children and redirect incoming edges

REMARKS ON ROBDDS

- Reduced form can be directly built from HDL code.
- Manipulation of ROBDDS (e.g., computing the AND of two ROBDDS) can be performed efficiently.
- Often implemented as a single data structure in which multiple functions are represented and common structures are shared.
- Equivalence check amounts to a simple pointer comparison:
  - Build ROBDD of reference
  - Build ROBDD of implementation
  - Common structures will be shared; if the reference and implementation are equivalent, entire data structure will be shared.
- Some functions (e.g., multiplication) cannot be compacted.
- See also course "CAD Tools on VLSI"

MORE ON FORMAL METHODS

- Active area of research already for several decades
- Few commercial products, not widely used except for RT-level equivalence checking
- Abstraction levels above RTL are much harder to deal with
- Model checking verifies that certain properties (so-called assertions, see later) are valid.
CODE COVERAGE

• Keep track of the lines of source code that are executed during simulation and report about coverage.
• Lines that have not been executed are a potential source of error.
• Coverage of lines alone is not sufficient; one can go several steps further:
  – Have all state transitions in a state machine been visited?
  – Suppose \((a \text{ or } b)\) is the condition of an “if statement”; have \(a\) and \(b\) become true independently?
• Does not say anything about design correctness but about quality of simulation.
• Available in most commercial simulators including Modelsim/Questasim.

ASSERTIONS

• Non-synthesizable code linked to design to detect unwanted conditions.
• Example: an ‘X’ on a data bus that lasts longer than, say, 10 ns is an indication of two active drivers, while only one is allowed.
• They trigger the stimulator to print a message or to stop the simulation.
• Available in VHDL from the beginning; now also in SystemVerilog (more advanced).
• Separate assertions standard PSL (Property Specification Language), developed by Accellera (www.accellera.org) and supported by major simulators.
• Higher simulation quality at the expense of simulation time.
• Assertions can be used for formal verification.

“CLASSICAL” SIMULATION

• Based on simple generation of stimuli and designer inspection of waveforms or text output for determination of correctness.
• It is quite common to base stimuli generation and output registration on data streams read and written to a file.

SHORTCOMINGS OF CLASSICAL SIMULATION

• There is only one design, the “implementation”. The “reference” is in designer’s and verification engineer’s mind.
  – Good idea to have separate verification engineer, for a “second opinion” on the interpretation of specification.
• DUV is at RT level and becomes available in a late stage of the design:
  – Software development cannot start easily in time; verification with software will delay the tape-out.
  – RTL code is slow to simulate; it is only feasible to simulate small software programs.
TRANSACTION-LEVEL MODELING

- Abstract way of looking at hardware:
  - I/O signals not at the bit level, but as abstract data structures
  - Behavior specified in terms of transactions
  - In general, not clock-cycle accurate
- Example:
  - “Write to memory” is a transaction; its implementation will involve preparing data, address and control signals with the required timing relations.
- **Transactors** translate transactions to bit-level signal changes and back.

FEATURES OF ADVANCED SIMULATION

- Self-checking testbenches: waveform inspection only for debugging.
- Transaction-level “golden reference design” built into testbench.
- Golden reference design, being not clock-cycle accurate, executes much faster and can be used for software verification at an early stage.
- Stimuli generation makes use of constrained random pattern generation to increase code coverage.
- Transactors evolve together with RT-level implementation.
- Assertions are extensively exploited.

ADVANCED TESTBENCH STRUCTURE

![Testbench Diagram]

LAYOUT-LEVEL MODELING

- Many languages exist for dedicated tasks (think of PSL for assertions)
- Two languages with a broad application area:
  - **SystemVerilog**: extends classical Verilog, especially strong for verification, building testbenches, but also applicable to synthesis-based design.
  - **SystemC**: system-level design language (ESL = electronic system level is the buzz word), with special extensions for transaction-level modeling and verification; RTL modeling for synthesis is possible but not yet widely supported.
- What about VHDL? Used to be much more powerful than Verilog, now trying to catch up.