Is today’s design methodology a recipe for a Tacoma Narrows incident?

Carl Seger
Strategic CAD Labs
Intel Corporation
July 8, 2009
Outline

• Validation brick wall
• Two types of validation
• What is known
  – More development needed
• What is unknown
  – More research needed
• Danger of “business as usual”
Electronic Circuits

• Moore’s law drives industry
  – Number of transistors available doubles every two years
    – Over 2 billion in 2009
    – No sign of show-stoppers for next 10-15 years.

• Extremely complex systems can be designed on a single die
  – Single chip multi-core processors
  – System On a Chip

• Society increasingly depends on correctly functioning products and devices
Design Challenges

- **Complexity of design**
  - More transistors → More functionality → More design effort

- **Number & size of models**
  - Performance, ERTL, GRTL, Schematics, ...
  - Multi-million line RTL

- **Multi-objective convergence**
  - Timing, power, area, etc. feedback way too late in design schedules

- **Validation of design**
  - Bug rate rising ~4x per lead
  - Trillions of simulation cycles on a rapidly changing model

---

Plan → Design → Analyze

weeks/months
Without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry.

THE INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS: 2005/6
Two Classes of Bugs:

- **Specification** bugs
  - “What” is captured incorrectly
    - Unintended interactions
    - Communication failures
    - Deadlock
    - Livelock

- **Implementation** bugs
  - “How” is captured incorrectly
    - Refinement failed

- **Note:**
  - The more abstract the specification is, the more implementation bugs (and vice versa).
How to Address Implementation Bugs

- Formal equivalence* checking
  - Poster child of formal methods
  - Sequential checking and local property verification are still difficult and can benefit from algorithmic breakthroughs
- However, FEV is very limited in abstraction gap that can be bridged
- Integrated design and verification can solve this problem

* Should really be called Formal Refinement Checking
Integrating Design and Verification

- Start with a very high-level model description of the design
  - Validation target
- Through sequential design steps:
  - Create more detail & explore/add/remove
  - While proving that each step maintains correctness
- Additionally, start from detailed design and abstract up
  - Abstract details by transformations
  - While proving that each step maintains correctness
- System:
  - Ensures correctness
  - Automatically replays steps
Example Designs Done Using a Prototype IDV System

Bottom line: During 13 months of design effort, no RTL changes were needed because of implementation considerations.

Area for which more rD is needed

Graphics execution unit (~120,000 gates) HLM -> Placed cells
What to do for Spec-bugs?

- Create fewer bugs
  - Write significantly more abstract specs
    - Style? Methodology? Language? ...
  - Change specification infrequently
    - How to accomplish this?
    - Maybe make it easier to absorb specification changes?

- Make design easier to check
  - Focus on “what” not “how”

- Make bugs easier to find
  - Reduce specification size by at least 1-2 orders of magnitude

- Capture bugs sooner

- Reuse verification
Danger of "Business-as-Usual"
The Original Tacoma Narrows Bridge

- The first Tacoma Narrows Bridge was evolutionary in its design.
  - Third longest suspension bridge ever constructed
  - The lightest suspension bridge (considering its length) ever constructed
  - (Arguably) the most beautiful and elegant suspension bridge ever constructed.

- The original bridge was built
  - using the best available scientific knowledge
    - including self resonance and vortex induced vibrations
  - was manufactured correctly using high-quality products
But...

- The bridge collapsed four months after its opening.
  - The shape of the bridge was similar to an airplane wing and created significant lift even in modest winds.
  - Due to self-excitation (negative damping) a “cork screw” 0.2Hz oscillation grew until the bridge deck broke and the bridge collapsed.
  - This was an entirely new phenomena and required a new validation approach.

- Let us not make the same mistake in continuing today’s validation approaches blindly into the “new brave world” of multi-billion transistor system-on-a-chip designs.
Backup
Ideal Specification

- A specification of *what* you want
- Ideally, immutable and has immunity from how you use it
- But:
  - Has to change due to “above” changes (bugs, architectural feature change, environmental changes, etc.)
  - May have to change if not what you really want (e.g. “below” discovery that the idea was bad to begin with)
  - Have to change if it cannot be built (e.g., “below” discovery that spec. is not implementable)
Create fewer bugs

- Use a KISS approach (keep it simple and stupid)
- Reduce the number of lines of code
  - Higher-level modeling (powerful abstractions)
  - Focus on “what” not “how”
- Re-use already correct code
- Use experienced coders with good SW skills
- Use a structured SW development method
  - E.g., extreme programming
- Use a very small team (<10)
  - Each coder owns/understands more of the interactions
- Use a concise and efficient language to express design in
  - Rich strongly typed language
  - A language with powerful abstraction mechanisms
- Do thorough and formalized code review
Make Design Easier to Check

• Make features orthogonal
  – In the high-level model, do not use sharing even though the implementation will!
• Avoid duplication of same/similar state
• Make modules functional
  – Avoid state
  – Localize state to input and/or output delays
• “Overdesign”
  – Don’t take advantage of every don’t care
• Use standard well-defined protocols between components
  – Efficiency can be added during refinement
• Make don’t cares explicit
  – Both temporal and data
• Make environmental assumptions explicit
Make Bugs Easier to Find

- Make modules self-contained
  - Localize impact of bugs
- Make environmental assumptions explicit
- Add invariants and properties to code
- Write complex behaviors as a composition of simple ones
  - Test/verify each simple module
- Use an environment in which composition is correct by construction
  - E.g., very strong type checking (including properties and behaviors)
Capture Bugs Sooner

- **Static checks**
  - Strong typing
  - Thorough Lint type program enforcing naming and coding style
  - Formal verification of properties
    - User written properties
    - Self consistency properties (e.g., new feature did not break old functionality)
  - Formal verification of equals-for-equals

- **Symbolic Simulation**

- **Dynamic checks**
  - Faster simulation
  - HW emulation
  - Extensive coverage monitors

- **Add rigorous regression checks for checking in code into repository**
Logical Design Transformations

- Add correct-by-construction implementation details
  - Examples:
    - Bypass
    - Re-timing
    - Duplication/merging of logic
    - Changing state encoding
    - Don’t care usage
    - Introducing clock gating
    - ...

- Allow arbitrary design changes when coupled with machine-checked justification
Physical Design Transformations

• Add physical details
  – Examples:
    – Change Hierarchy
    – Re-synthesize
    – Change relative placement
    – Change overlapping region constraints
    – Replace abstract wires with sized/repeated wires

• Again, allow arbitrary design changes when coupled with machine-checked justification

Spec
  Block A: at most 40% utilized
  Block B: at most 40% utilized

Imp
  Block A: at most 60% utilized but in smaller area
  Block B: at most 80% utilized but in smaller area