Topic III

Verification Techniques Overview (I)

Functional Verification

系統晶片驗證

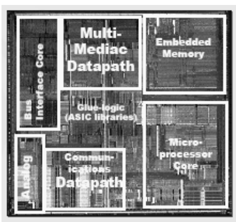
SoC Verification

Sep, 2004

What is Verification?

◆ What's the problem?

- We have learned that modern designs are
  - Extremely complex
  - With very high time-to-market pressure



☞ Are you sure your design is correct under all scenarios? (Have you fixed all the bugs?)

A high stake game

US ‘02 wireless telecom revenue: $76B (CNet)

- 7 M gates
- 500K lines of code

- DoCoMo: 23 M consumers with Java based phones.
- Mandatory recall cost: $4.2B (2001.06)

Kashai/Verisity, HLDVT 2003

And of course,

don’t forget the infamous

Pentium division bug…

Where is the problem?

☞ Let's review a little bit about the typical design process…

Typical Design Process (1)

- Market research
- Feasibility study
- Custom feedback
  - ➔ Market Requirement Documents (MRDs)
  - ➔ Product Requirement Documents (PRDs)
  - ➔ Functional Specification Documents (FSDs)

(English, block diagram, timing diagram,… etc)

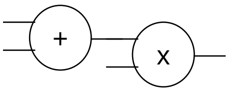
```
for (i = 0; i < d; i= i+2) {
    if (y > 3)
        p = p * 3;
    else
        q = q + r;
}
```

- Algorithmic-level design
- System-level design
- Behavior-level design
  - ➔HW/SW co-design/co-verification
  - ➔Transaction-based models
  - ➔Performance analysis

(C/C++, SystemC, RTOS,… etc)

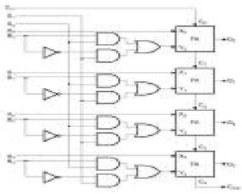
Typical Design Process (2)

```
always @ (posedge ck)
   count <= count + 1;
```



- Block-level design
- Register Transfer Level (RTL) design
  - ➔ Manually composed HDLs
  - ➔ Cycle accurate model
  - ➔ Precise hardware model

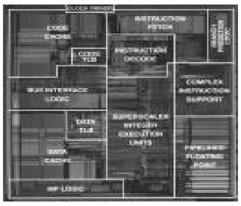
(Verilog, VHDL, design library,… etc)



- Automatic logic synthesis into gate-level design
  - ➔ Sea of gates
  - ➔ More detailed performance analysis
  - ➔ Technology dependent

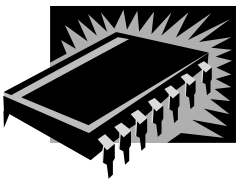
(Verilog, VHDL, design library,… etc)

Typical Design Process (3)



- Transistor-level design
- Automatic placement and routing
  - ➔ Polygons
  - ➔ Mask data
  - ➔ Technology files

(Spice, GDS-II,… etc)



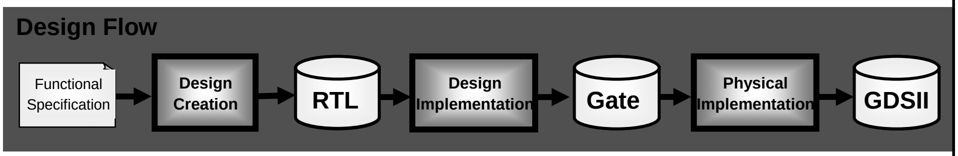
- Manufacturing
- Packaging
  - ➔ICs
  - ➔Printed Circuit Board

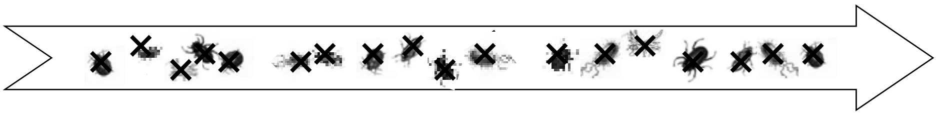
(Hardware)

What is Design Verification?

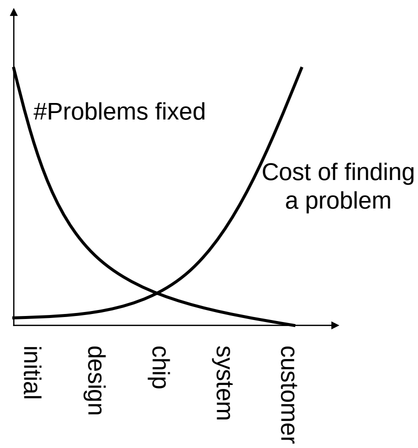
<u>To verify the correctness of your design</u>

(Find as many design bugs as possible)



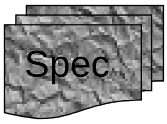


Verify Early

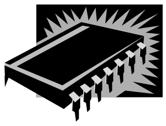


Definition of Verification

- Strictly speaking, verification is usually referred only to "*functional verification*"

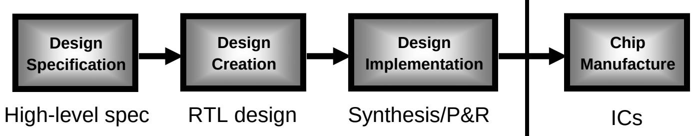


- Functionally correctness
- Spec fully implemented



- Generally speaking, verification also covers
  - Timing verification
  - Physical verification
  - Manufacturing defect testing

(Functional) Verification vs. Testing



Verification

- Object → design (SW)
- Methodologies
  - Simulation
  - Emulation
  - Formal techniques

Testing

- Object → chip (HW)
- Methodologies
  - ATPG
  - Fault Simulation
  - Scan / BIST

In short,

verification

is to

make sure

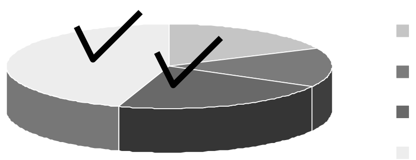
your design

is

good.

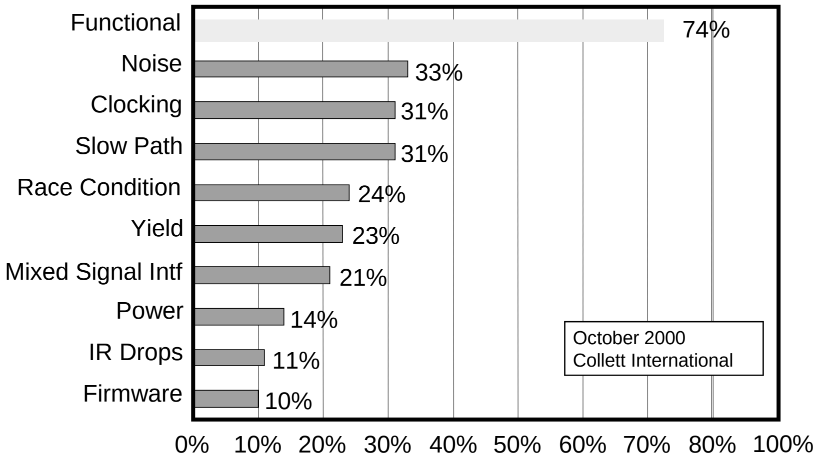
Importance of Verification

- By most statistics, verification consumes about 70% of total design efforts



- Half of chips today require 1+ re-spins
  - Mask cost ➔ .18 (350K), .13 (500K), .10 (1M), .07 (4M)

Functional Error %



See what they said…

"My biggest problem about design verification is that *the time is never enough*. I can only do my best to run more simulation, but I don't know whether it is sufficient."

--- Broadcom designer (similar comments from many others)

"We actually spent more time in *fixing the bugs* than finding them. It takes almost 50% of the total design time."

--- HP verification engineer (similar comments from many others)

How do you

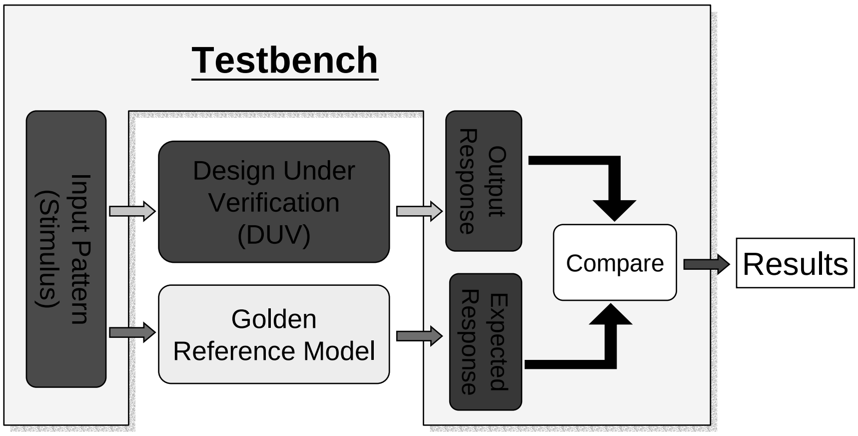
verify

your design

?

Yes, most people verify their designs by simulation and debug by examining the output responses

Simulation-Based Verification



Testbench Creation --- Input pattern

- HDL
  - Simulation module/wrapper
  - OK for small design
- PLI
  - C program linked to simulator
  - Can handle more complex functions
- Waveform-based
  - Tool-provided waveform editing window
- Transaction-based
  - Need “Bus Function Models (BFMs)” to translate transaction-level stimulus to and fro cycle- and pin-accurate transitions in DUT
- Specification-based
  - Testbench is created by capturing the spec in an executable form
  - Need interface to simulator

Which method is better?

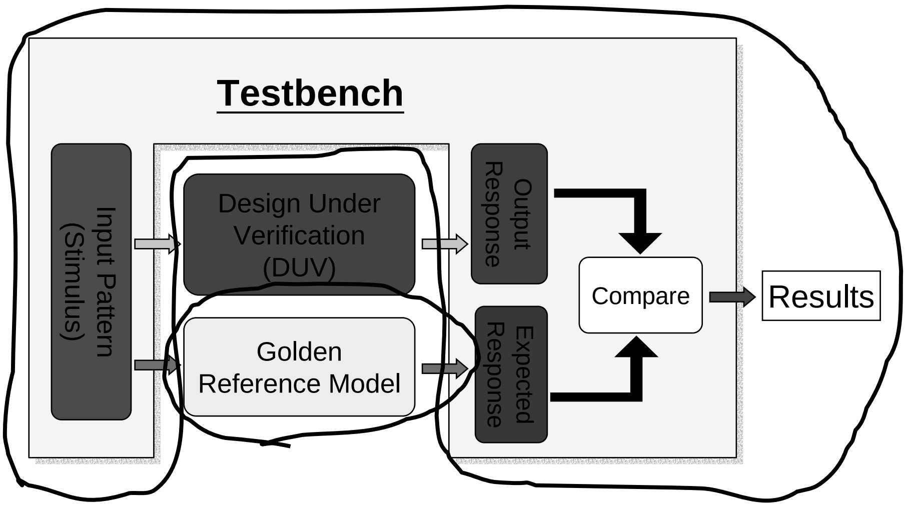
- HDL?
  - Designers most familiar with
  - But, HDL is good for design description, not for testbench description
- Others?
  - Extra interface
  - Tool support/interaction
- Environment constraints
  - Input may have some constraints
    ➔ not free variable

Testbench Authoring

- Testbench/verification Languages
  : e, vera, SystemVerilog, SystemC, etc.

1. Input stimulus, events, constraints
   - Built-in high-level constructs
   - Object-oriented: good for reuse
   - Automatic test vector generation
2. Assertion monitors
3. Coverage analysis
4. Hardware design description capability

Testbench Authoring



Any Problem? Whose Problem?

"My biggest problem about design verification is that *the time is never enough*. I can only do my best to run more simulation, but I don't know whether it is sufficient."

--- Broadcom designer (similar comments from many others)

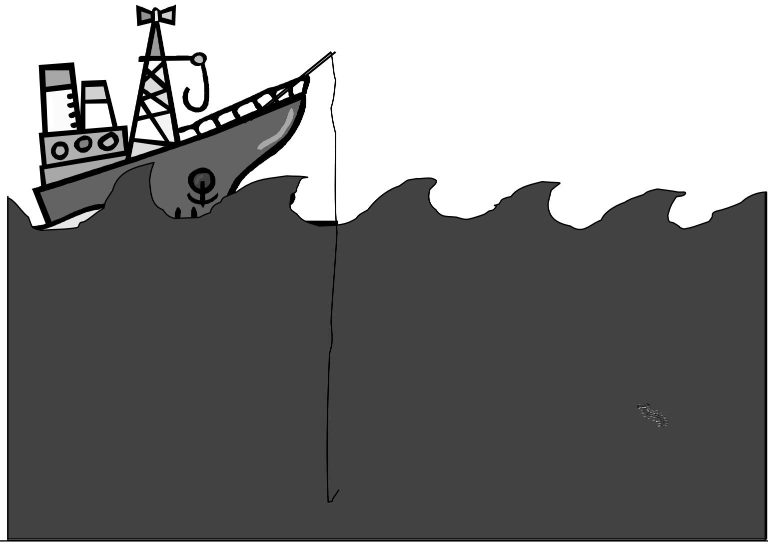
"It is very hard to write the testbenches and assertions for the design, since *I am not a designer*. Ask the designer to do it more? No way!!"

--- Sun Microsystems verification engineer (similar comments from many others)

Let's do some math

- Suppose a circuit has 100 inputs (this is considered tiny in modern design)
  - Total number of input combinations $= 2^{100} = 10^{30} = 10^{24}M = 1.6$ mole Mega
  - Requires (in the worse case) 10^{24}M test patterns to exhaust all the input scenarios
  - Let alone the <u>sequential combinations</u>
  - For an 1 MIPs simulator
    ➔ runtime = 10^{24} seconds = $3 * 10^{16}$ years

Like Finding a Bug in an Ocean…



What is worse, when design gets bigger…

➔Simulation runs much slower (exponential complexity)…

- e.g. Time to boot VxWorks
  - 1 million instructions, assume 2 million cycles
  - Today's verification choices:
    - 50M cps: 40 msec   Actual system HW
    - 5M cps: 400 msec   Logic emulator [1] (QT Mercury)
    - 500K cps: 4 sec   Cycle-based gate accelerator [1] (QT CoBALT)
    - 50K cps: 40 sec   Hybrid emulator/simulator [2] (Axis)
    - 5K cps: 7 min   Event-driven gate accelerator [2] (Ikos NSIM)
    - **50 cps: 11 hr   CPU and logic in HDL simulator [3] (VCS)**

1: assumes CPU chip   2: assumes RTL CPU   3: assumes HDL CPU

About VxWorks (http://www.faqs.org/faqs/vxworks-faq/part1/)

source: Kurt Keutzer, UCB

Then why is simulation still the mainstream approach in verification?

How can it be useful?

Simulation is useful because…

1. In early design cycle, most bugs are easy to find
   - Like something contaminates the ocean…
2. Use "design intent" to guide the testbench
   - Guided test pattern generation
   - Real-life stimulus
3. Make the DUV smaller
   - Lower level of hierarchy
   - Higher-level of abstraction
   - Design constraint
4. And ??

And of course,
simulation approach is
easier to learn, simpler to use,

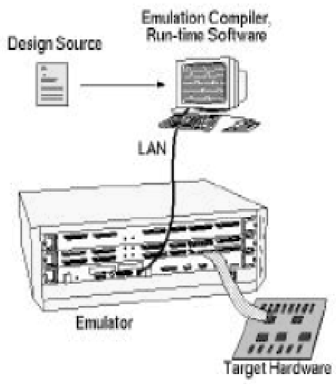
so it is the most popular.

But corner-case bugs are still very tough…

Can we do better?

(Speed up the simulation?)

Simulation Speedup (1): Emulation

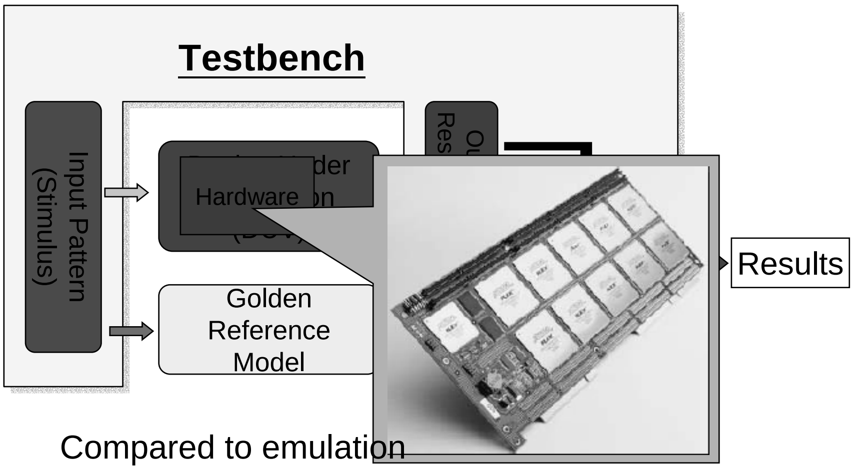


source: Kurt Keutzer, UCB

- Entire design or major module is flattened, and compiled at once into multi-FPGAs (emulator)
  - Tens to thousands of large "FPGAs"
- In-circuit or vector-driven simulation
- Regular clock rate > 1M cps

100x - 10000x Speedup; but very expensive

Simulation Speedup (2): Hardware Accelators



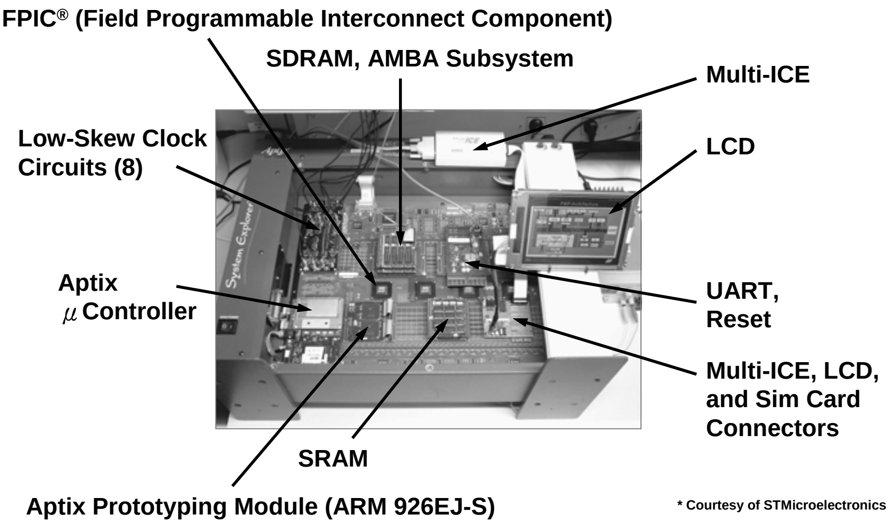
Compared to emulation
➔ lower at cost, but sacrificed in speed

e.g. ReConfigurable Computing (RCC) elements (AXIS)

Simulation Speedup (3): Rapid Prototyping

- Rapid prototyping system:
  1. Prototyping modules
     - Processor integrator core (e.g. ARM)
     - RAM, AD/DA, PLL
     - FPGA for IP cores
  2. Field Programmable Interconnect Components (FPIC)
  3. Software
     - Mapping, synthesis, optimization
  4. Debugging interfaces

A Real-World Example (Aptix)*



* Courtesy of STMicroelectronics

Other than using hardware to speed up the simulation…

Remember the difference between verification and testing...

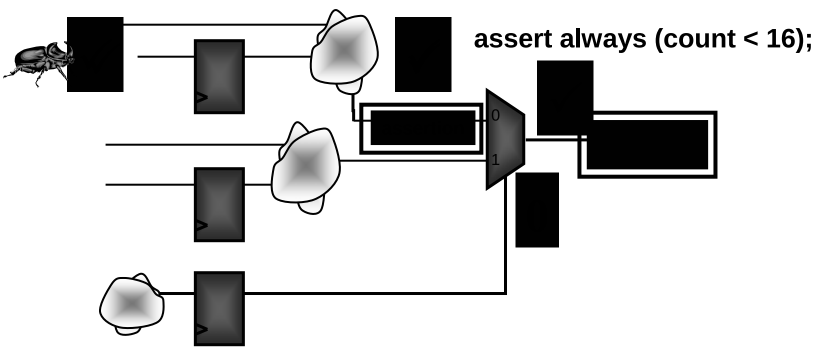
Verification vs. Testing

	Verification	Testing
Objective	Design (SW)	Chip (HW)
Environment	Simulator, debugger (tools)	Test equipments (HW)
Observation points	Any signal in the design	Chip outputs

◆ Unlike testing, verification does not have the observability problem

Observability Problem

◆ Bugs are detected at the observation points like POs and registers



source: Harry Foster

Observability Problem --- Solved

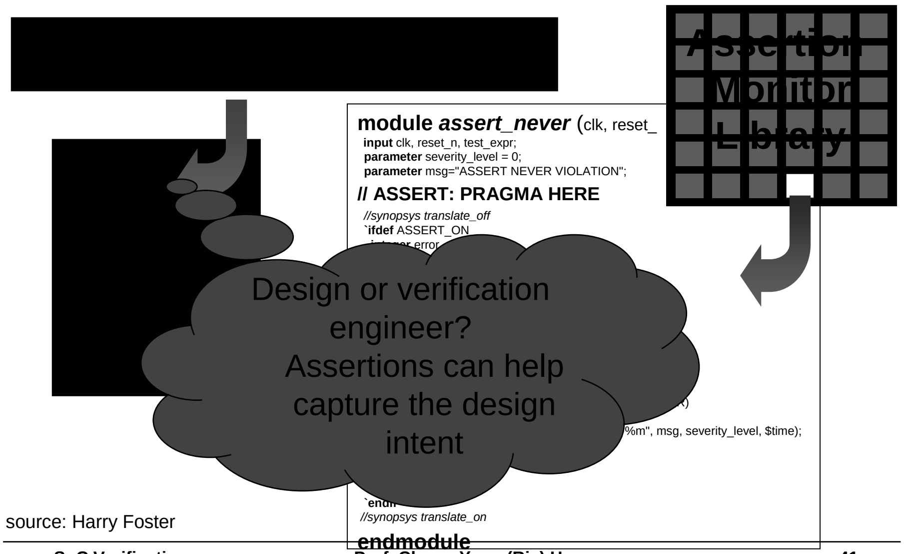
- Insert "*assertions*" in the middle of the circuit, and flag the simulator whenever the violation occurs
  - ➔ Increase the observability of the bugs

The difference between
hardware and software simulations

Assertion-Based Design/Verification (ABV)

- Embedded assertions in RTL design
  - Automatically checked by simulator
  - Nice properties for formal tools
  - ➔ Specify once!!
- Languages
  - e, OVA (vera)
  - Open Verification Library (OVL)
  - SystemVerilog, SystemC

ABV Example --- OVL



source: Harry Foster

"Where am I going to find time to write assertions? I don't even have time to write comments!"

--- Conexant design engineer

Summary of Simulation Techniques

- Advantages
  - Able to find most easy bugs
  - Easy to learn, user friendly (tools)
  - Assertions can be of great help

- Problems
  - Exhaustive simulation is impossible

  ➔ How do I know I have done enough??
    - Designer's self confidence
    - Manager's approval
    - Time is running out

    (No quantitative measure)

Coverage Metrics

DUT: usually RTL or up

1. A quantitative measure to assess the quality of a test suite
   - [Assume] Test completeness or verification confidence is proportional to the simulation coverage on certain attribute of the design
2. Can also be a hint to generate more vectors on the area where the test is not sufficient

- Examples
  - Line (statement), toggle, branch, expression, Path
  - FSM state, transistion
  - Tag

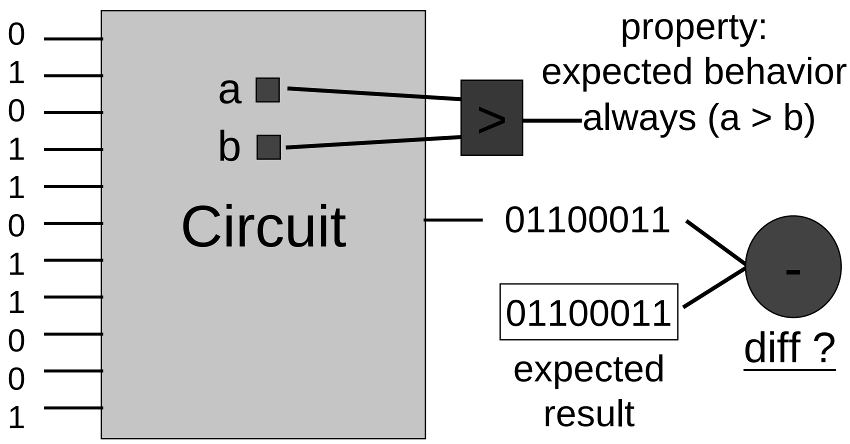
100% Coverage == 100% Verification??

- What does 100% coverage mean?
  - Every <line> has been visited by simulator
  - Can we miss any bug?
  - 100% path coverage?
- What can't simulation answer?
  - Eventuality (witness property)
    - "I will eventually be a billionaire"
  - Dead/Live lock (loop)

Any alternative to simulation/emulation?

Formal Verification

◆Property checking: “proof” techniques

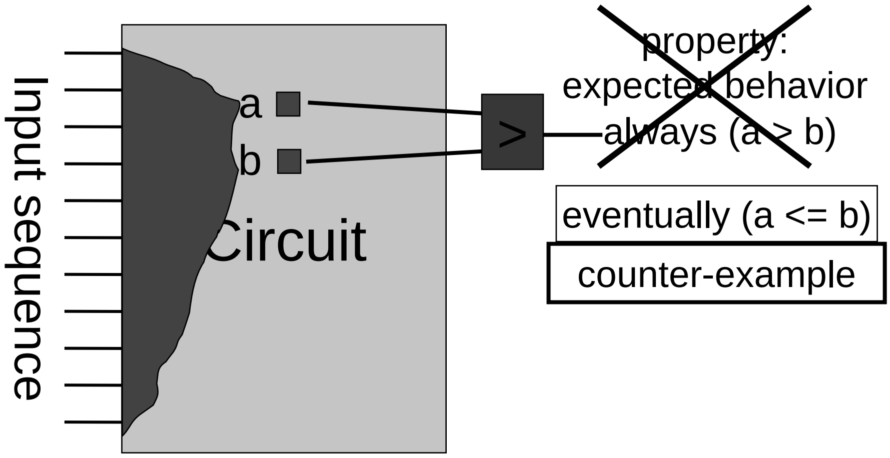


Formal Verification

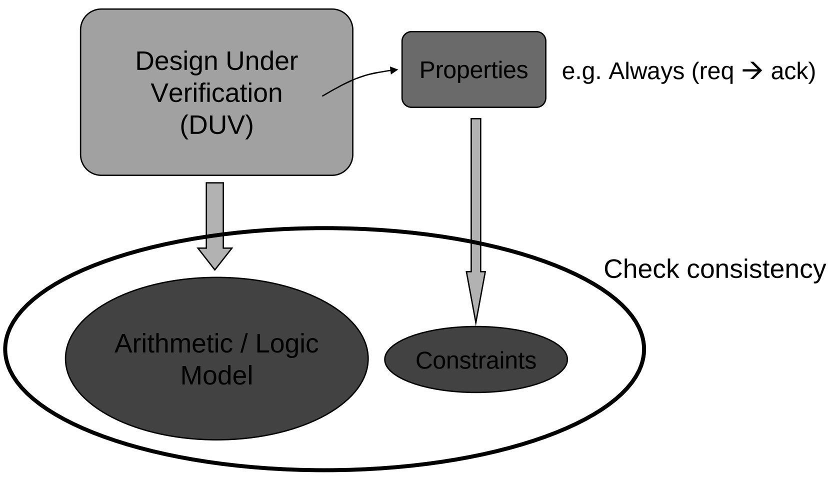
◆Given
- Property: expected behavior

◆Prove
- Property always hold under all circumstance
- No input sequence can make property fail

Finding Counter-Example



Formal Verification Technologies



For example

- Proving “always (a > b + c)”
  - Simulation needs to enumerate all the possible combinations of a, b, and c
- But, consider circuit is just a set of logic relations of input signals
  - Imagine in a math test, given a set of logic relations and asked to prove (a > b + c)
  - Try to use logic and math reasoning (e.g. induction)

Solving logic / temporal relations between circuit signals…

<u>A constraint satisfaction problem</u>

Constraint Satisfaction Problems

- Constraints
  - Logic: y = a && b;
  - Mux: y[31:0] = en? a[31:0] : b[63:32];
  - Arithmetic: y = (a > b)? (c + d) : (c * d);
  - Relational: (x1 << 1) + x0 >= 256;
- Constraint Satisfaction
  - Find an <u>input assignment</u> that satisfies all the constraints
  - Note: solutions in modular number systems

Constraint Satisfaction Solver

- Solving techniques
  - Boolean: SAT, ATPG, BDD, etc.
  - General: arithmetic solvers
  - Advanced: abstraction, genetic, probabilistic, theorem proving, etc
- Examples of Applications
  - Testbench generation (Find a solution)
  - Assertion validation (Prove no solution)
  - Optimization problems (+ Cost functions)

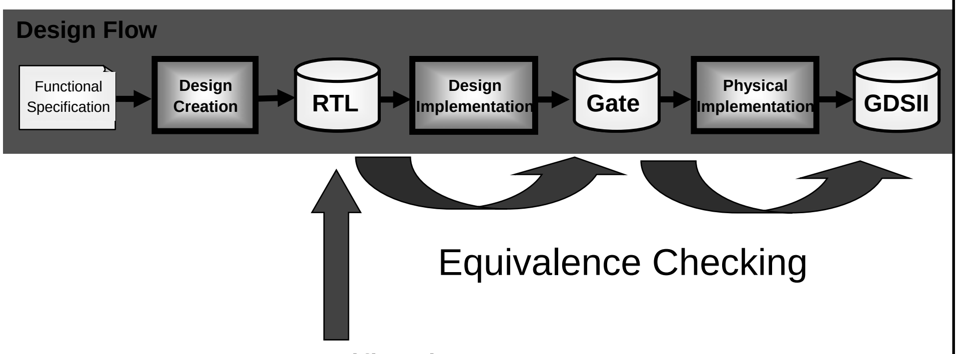
Formal Verification: 100% Verification?

- Note: formal verification can “prove” a property always true
  - For all input combinations until infinite time
- If we can prove ALL properties true (or find counter-examples for failed properties)

  ➔ Is it 100% verification??

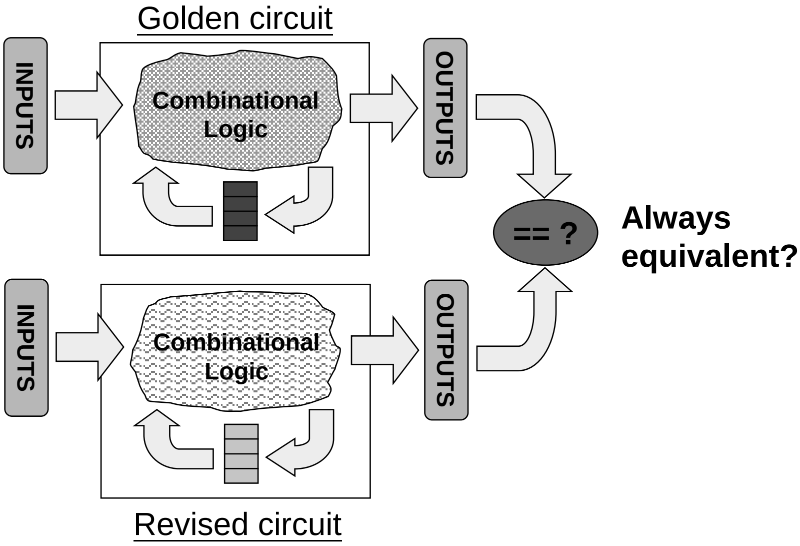
Do you write enough properties??
(who’s responsibility??)

Apply Formal Techniques in Design Flow



Intent verification
- Property checking
- Golden circuit

Equivalence Checking (EC)



Limitations of Formal Techniques

- ◆(Sounds too good to be true!!)
- ◆Complexity
  - Space (memory) explosion
  - Time: cannot complete

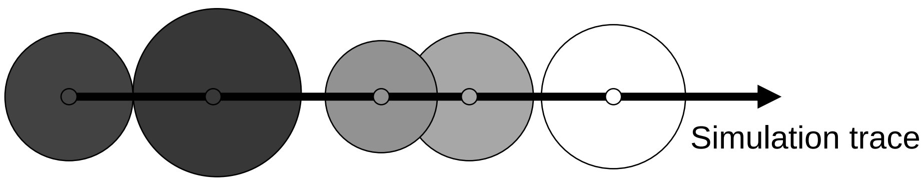
  ➔ If a proof is inclusive, what do you get??
  (no coverage metric information)
- ◆Learning curve
  - To write temporal properties
- ◆People are very used to simulation-based verification

Simulation vs. Formal

- Simulation
  - Easy to use
  - Can run on large circuit
  - Can detect easy bugs quickly
  - Almost impossible to handle corner case bug
- Formal (property checking)
  - Higher learning curve for designers
  - Cannot perform exhaustive search on large designs
  - Can target on corner case bug

Semi-formal --- combines the advantages of both

Simulation-based Semi-formal Approach



Apply formal techniques (state space exploration) around the simulation state

What we have learned so far…

- ◆Simulation-based techniques
- ◆Emulation, rapid prototyping
- ◆Formal verification

What to expect next…

- ◆Static analysis
- ◆Physical verification
- ◆Manufacturing test