Lecture 9
RTL Verification

Sildes adapted from Ofer Shacham

Optional Related Reading

- J. Bergeron, *Writing testbenches: functional verification of HDL models*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.
- T. Fitzpatrick, A. Salz, D. Rich, and S. Sutherland, *System Verilog for Verification*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
- P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip verification: methodology and techniques*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.
- *OpenVera*, Synopsys, http://www.open-vera.com/
- *Specman Elite - Testbench Automation*, Cadence, http://www.verisity.com/products/specman.html
- S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for System Verilog Assertions*. New York, NY, USA: Springer Science+Business Media, Inc., 2005.
- Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "Focs: Automatic generation of simulation checkers from formal specifications," in *International Conference on Computer Aided Verification*, 2000, pp. 538–542

Why is Verification Important?

- A design alone is of no use
  - Design is used in an actual "system"
    - e.g., a microprocessor in a computer system
    - e.g., a flight controller in an aircraft
- BIG question: Does the design work in an actual system?
  - Why won't it work?
    - e.g., Simple goofs, incorrect understanding of specs, ambiguous specs, late feature definitions, etc.
- Consequences of incorrect design
  - Catastrophic (human lives), financial losses, bad reputation

Incorrect Verification In The News

"Intel has recalled its fastest chip--the 1.13-GHz Pentium III--saying the chip could cause system errors when running certain programs and at a particular temperature." (CN News, August 2000)

*"The bug … is a specific illegal instruction ... The instruction will crash systems based on **Pentium** …"* (WIRED, October 1997)

*"**AMD** delaying **Barcelona** volume shipments to next year"* (CNET News, December 2007)

*"Taiwanese … DRAM makers have been given a new lease of life for their mainstay 64MB DRAM product **thanks to … Micron which … had to recall** a large number of defective chips ..."*(BNET, July 1999)

Toshiba to Spend $1 Billion to Settle Laptop Lawsuit

By ANDREW POLLACK

LOS ANGELES -- Toshiba Corp. said Friday th about $1 billion to settle a class action lawsuit br people charging that the world's leading maker of laptop computers sold

Business Day

D1

THURSDAY, NOVEMBER 24, 1994

The New York Times

Flaw Undermines Accuracy of Pentium Chips

By JOHN MARKOFF

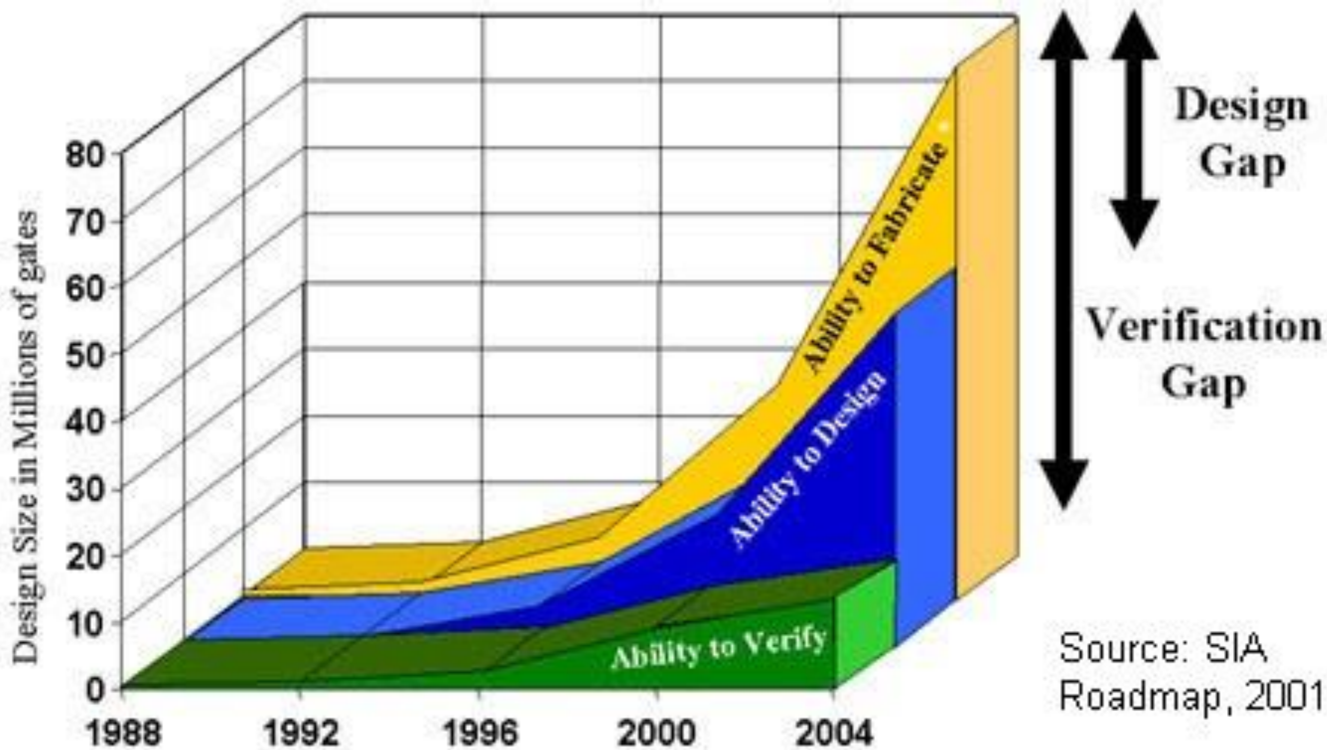
puters sold for home and business use, including those made by I.B.M.,

computers sold worldwide. Intel said yesterday that it did not

Intel's contentions that the error would only occur in extremely rare

Why Is Verification Difficult?

- Design complexity grows exponentially in time
  - Verification complexity grows at a faster rate
  - Our ability to verify is lagging behind

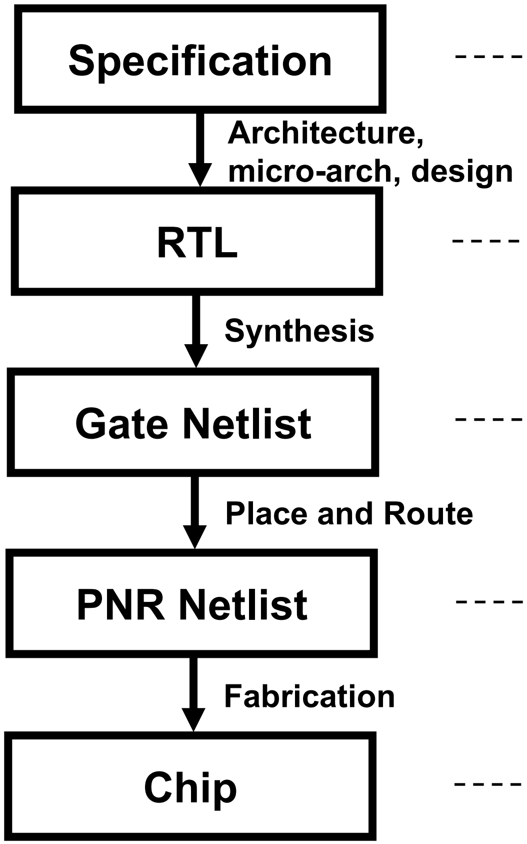


Where Can Things Go Wrong? (Everywhere…)

Chip Design Process	Typical Issues
Specification	In-correct or ambiguous specification
↓ Architecture, micro-arch, design	
RTL	Bad algorithm, wrong implementation, logic errors, connectivity mismatch, typos, …
↓ Synthesis	
Gate Netlist	Logic equivalence to RTL
↓ Place and Route	
PNR Netlist	Logic equivalence to gate netlist, setup/hold violations
↓ Fabrication	
Chip	Shorts, breaks, noise, power

Where Can Things Go Wrong? (Everywhere…)

Chip Design Process



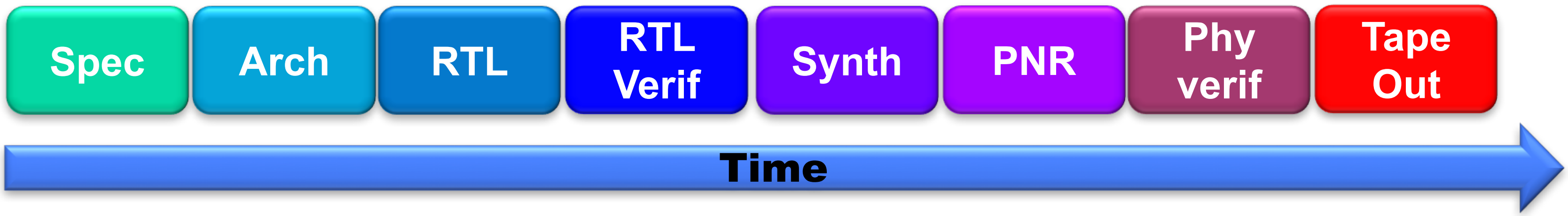
Typical Issues

- In-correct or ambiguous specification
- Bad algorithm, wrong implementation, logic errors, connectivity mismatch, typos, …
- Logic equivalence to RTL
- Logic equivalence to gate netlist, setup/hold violations
- Shorts, breaks, noise, power

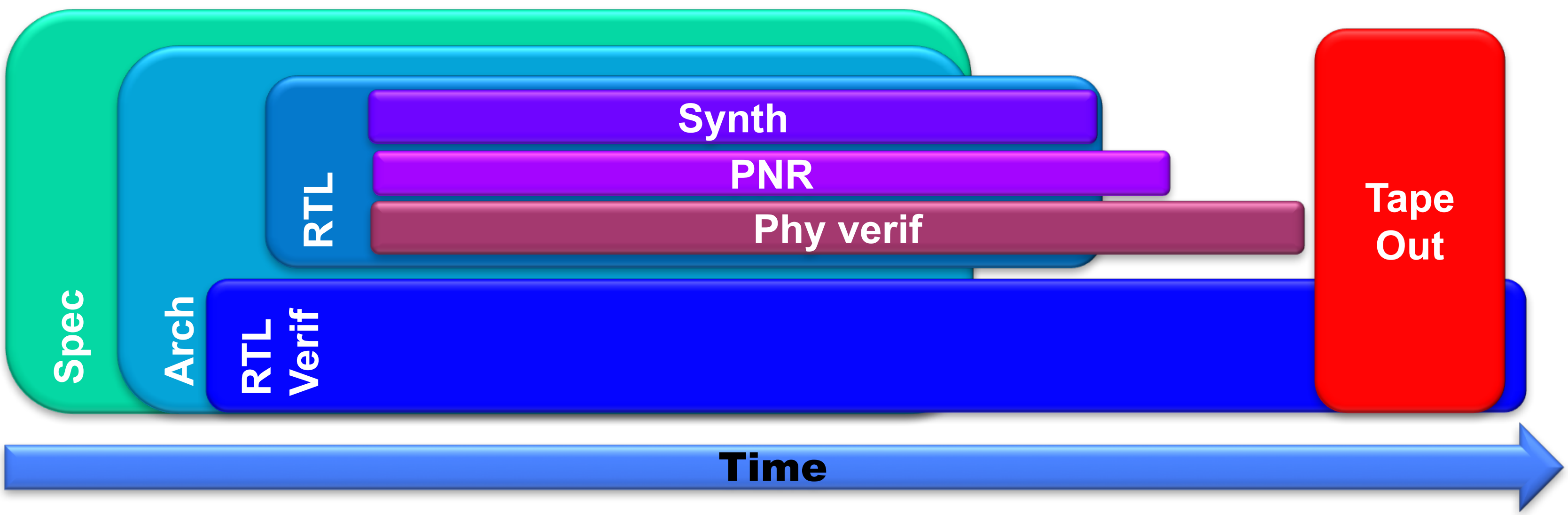
The focus of this week

Chip Design Time Chart

- In concept:



- In practice:



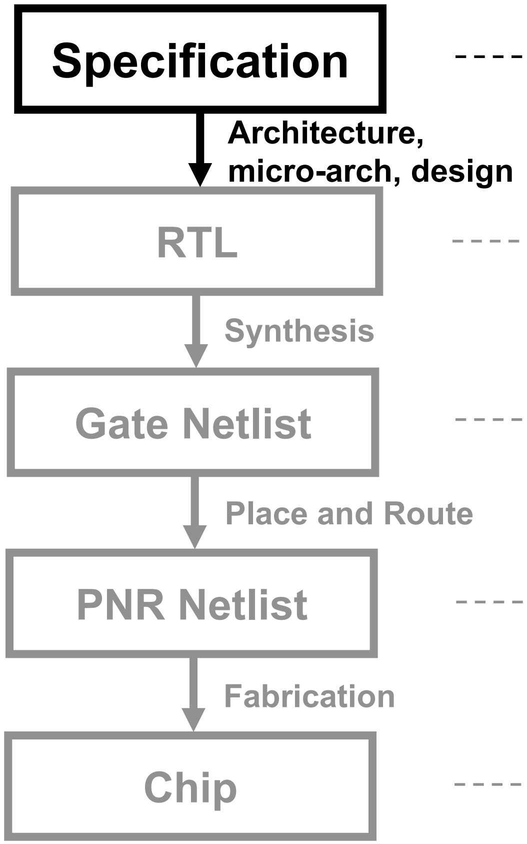
Verification

“Finding a bug should be a cause for celebration. Each discovery is a small victory; each marks an incremental improvement in the design.”

Clark, D.W., ``Large-Scale Hardware Simulation: Modeling and Verification Strategies", Chapter 9 of *CMU Computer Science: A 25th Anniversary Commemorative*, ACM Press/Addison-Wesley, 1991

Specification Bugs

Chip Design Process



Typical Issues

In-correct or ambiguous specification

Bad algorithm, wrong implementation, logic errors, connectivity mismatch, typos, …

Logic equivalence to RTL

Logic equivalence to gate netlist, setup/hold violations

Shorts, breaks, noise, power

High Level Design

- Creating executable model is critical
- Improve the ability to match specification and designer intent
  - Move to higher level abstraction
- Raising abstraction level:
  - Code is shorter and less prone to human errors

- Other means: Design reviews, block diagrams, SW simulator

Logic Bugs

Chip Design Process	Typical Issues
Specification	In-correct or ambiguous specification
↓ Architecture, micro-arch, design	
RTL	**Bad algorithm, wrong implementation, logic errors, connectivity mismatch, typos, …**
↓ Synthesis	
Gate Netlist	Logic equivalence to RTL
↓ Place and Route	
PNR Netlist	Logic equivalence to gate netlist, setup/hold violations
↓ Fabrication	
Chip	Shorts, breaks, noise, power

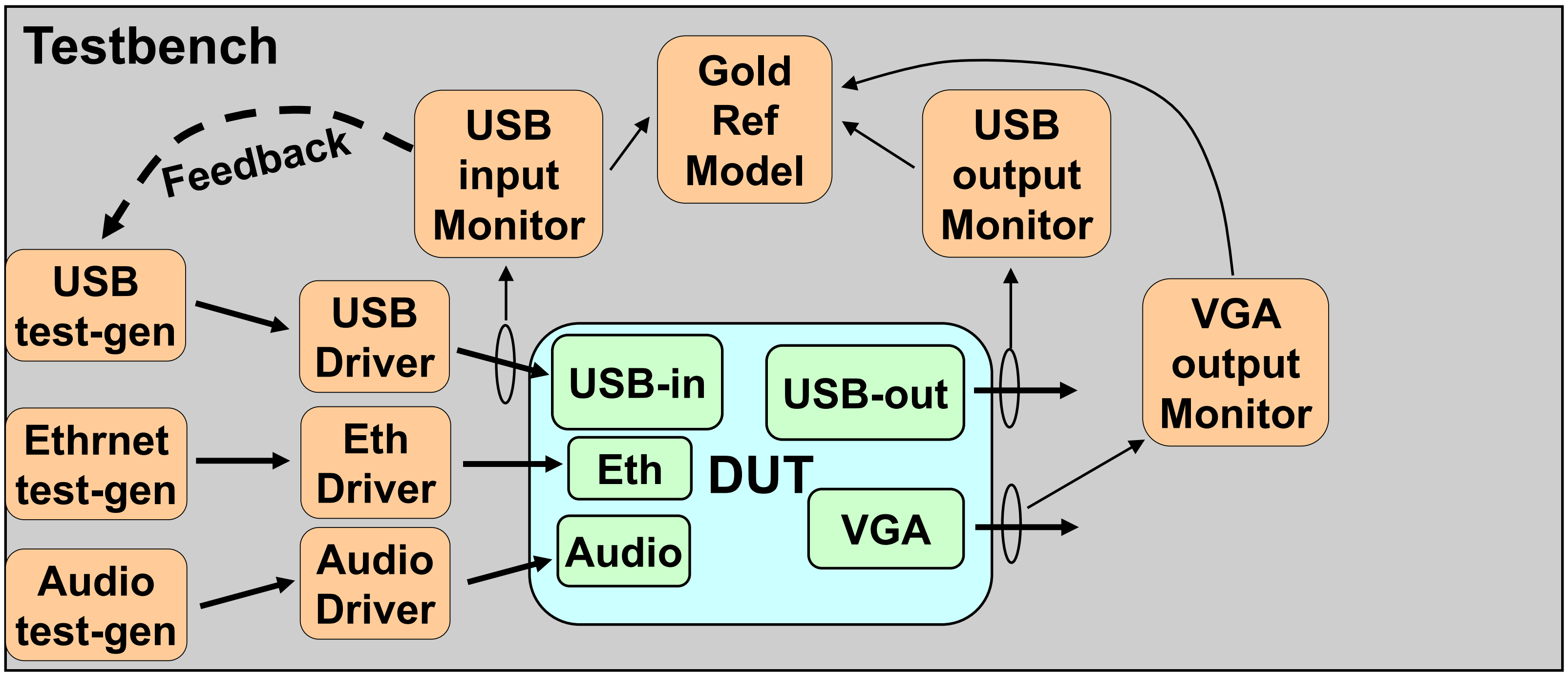
How to Find Bugs?

- Simulation
  - Apply a “LOT” of input sequences
  - Compare against expected outputs
- Emulation
  - Map design into FPGAs and plug into actual system setup
  - Much like simulation only 1000x faster
- Formal verification
  - “Prove” that the design really implements the specification

Simulation Based RTL Verification

- The most common approach to RTL verification
  - Roughly 50%-70% of the NRE costs of chip design
- Goal: Given an RTL description of a Design Under Test (DUT)
  - Verify that it is correct under any stimuli
- How?
  - Create a verification environment
    - (Smart) Stimulus generation
    - Result checking (reference model)
  - Application of Formal Property Verification (FPV) (Assertions)
- When do I stop?
  - Never…
  - Use coverage metrics to asses the merit of the test suite

Testbench Environment - Overview



Verification Languages

- Need languages that can “talk” to verilog
  - Drive inputs
  - “Look” at inputs, outputs and internal signals
    - And sometimes force a value on them
  - Can sometimes be done using Verilog/VHDL

- But often we need more
  - Allocate memory, remember history of events
  - Object oriented or aspect oriented for productivity
  - Verliog/VHDL cannot do that!

- Special descriptive languages that can “talk” to RTL
  - OpenVera (Synopsys), Specman E (Cadence), System Verilog

Key Principles: Reusability and Abstraction

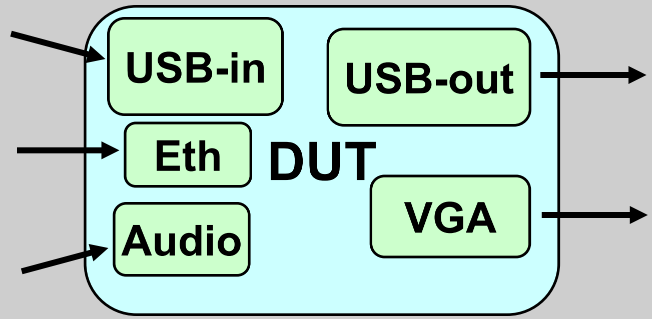
- Separate:
  - Test generation / test monitoring / test checking
- Reuse verification environment components hierarchically
  - Say you have a verification environment for a processor
  - To verify a chip multiprocessor, instantiate it multiple times
    - Might not need the processor interface drivers though
- Reuse the verification components for V2.0, V3.0 etc.
  - While the interface (driver) might change
  - The higher level abstraction are likely to be the same

Testbench Environment Creation

- Start with a design under test (DUT)

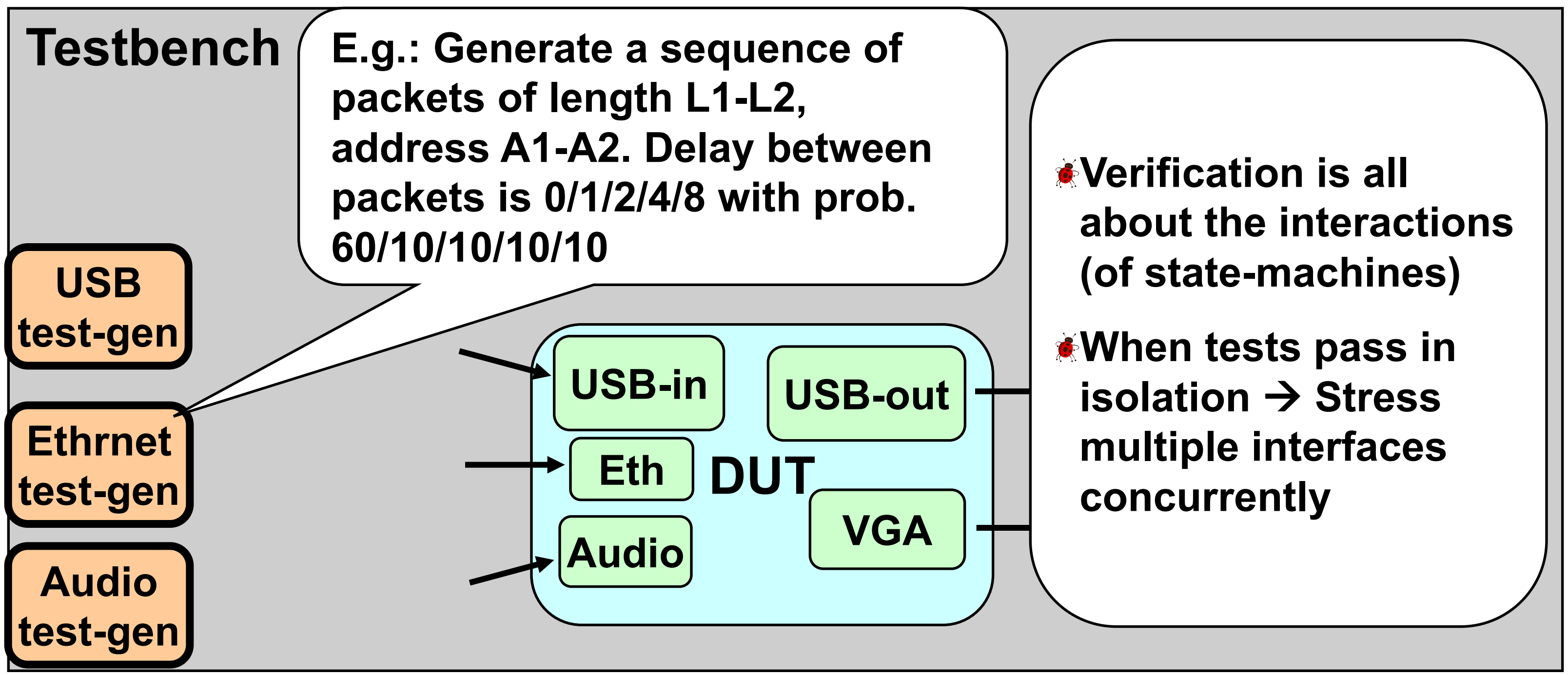
Testbench

E.g.: A design under test with three input interfaces (USB, Ethernet, Audio) and two output interfaces (USB, VGA)



Testbench Environment Creation - Stimulus

- Most critical part of verification
  - If you don't exercise a scenario, it's probably broken!
- Generate transactions (e.g. packets) not specific signal values



Stimulus Generation

- Decouple test generation for each interface
  - Increases reusability in the project and between projects
  - Increases ability to create weird scenarios
- Must also exercise all interfaces concurrently!
- Don’t settle for ONE random test
  - Create a regression suite
    - Overnight runs
    - Run many random tests, directed tests, legacy etc
- **How do we know that we covered all the interesting cases?**
  - E.g., we can’t ever cover all cases for a 64bit multiplier
  - We’ll talk more later…

Stimulus Generation, cont'd

- Pseudo-random sequences – extensive use
  - Covers lots of possibilities without spending a lot of effort
  - Requires legality checks: constraints must not be violated
    - E.g., certain addresses in a memory system might not exist
  - Also requires a good "golden model"
    - Must provide the right answer for silly input combinations …

- Issue: What about important yet improbable scenarios
  - Those are the "interesting" cases
  - E.g., suppose 100 signals must line up in a particular way? Requires a VERY LONG sequence if at all
  - E.g., exception conditions such as overflow situations

Types of Stimulus Vectors

- Random
  - Use seed based random generation.
    - Otherwise can not recreate scenarios
- Biased/Directed random sequences
  - Target improbable situations
  - Instead of equal probability, make some patterns more probable
  - Designer may help identify the “interesting” vectors
    - e.g., inputs that can cause deadlocks or livelocks
- Tests that have already “proven themselves”
  - Legacy tests that are known to hit “interesting” cases / bugs
  - Inputs captured from actual systems

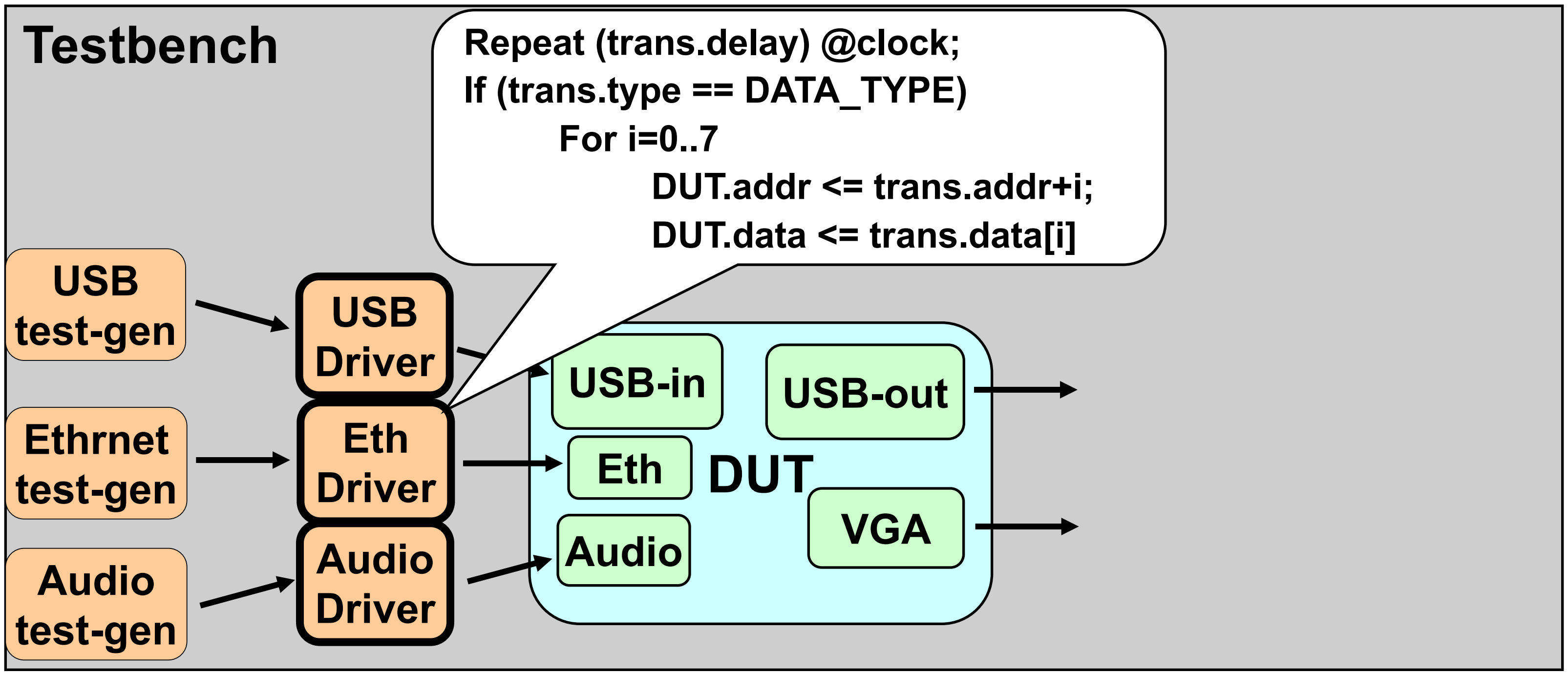
Pre-Generated Vs. On-The-Fly Generation

- Pre-generated: Input sequence generated before simulation
  - When the input does not depend on the current state
    - E.g., UDP traffic: incoming packets to a router
    - E.g., Inputs captured from actual systems
  - Sequence Generator can be decoupled from the testbench

- On-the-fly: The input is generated as the simulation executes
  - The input at time T2 depends on some previous state at T1
    - E.g., Processor's Memory interface: The input data to the processor depends on the address provided at the previous cycle, and the data saved at the memory some time before
  - Need feedback from testbench

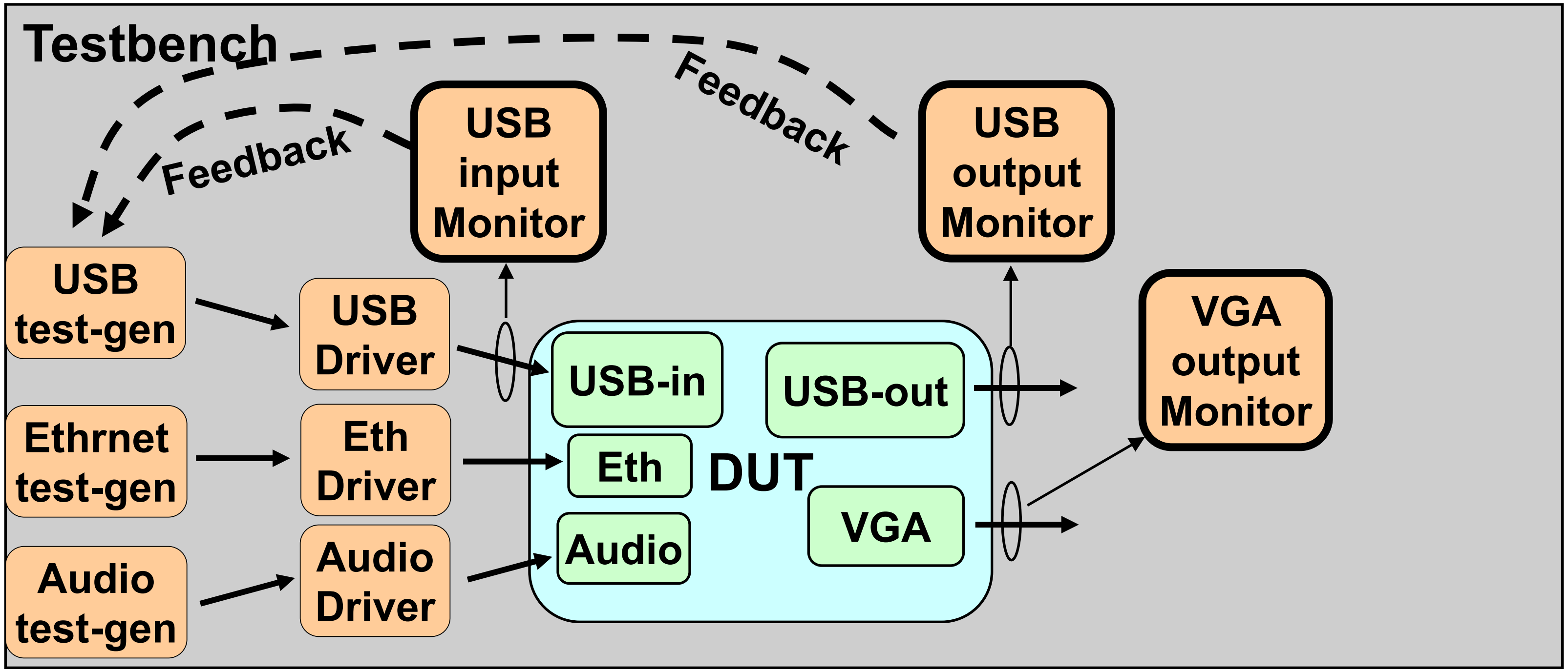
Testbench Environment Creation - Drivers

- Drive the generated transaction to the design ports
- Translate the abstracted transaction to bits on wires
  - E.g.: drive a 64 bit message as 8 chunks of 8 bit signals
  - E.g.: Translate a Cache-miss message to binary 0x00A



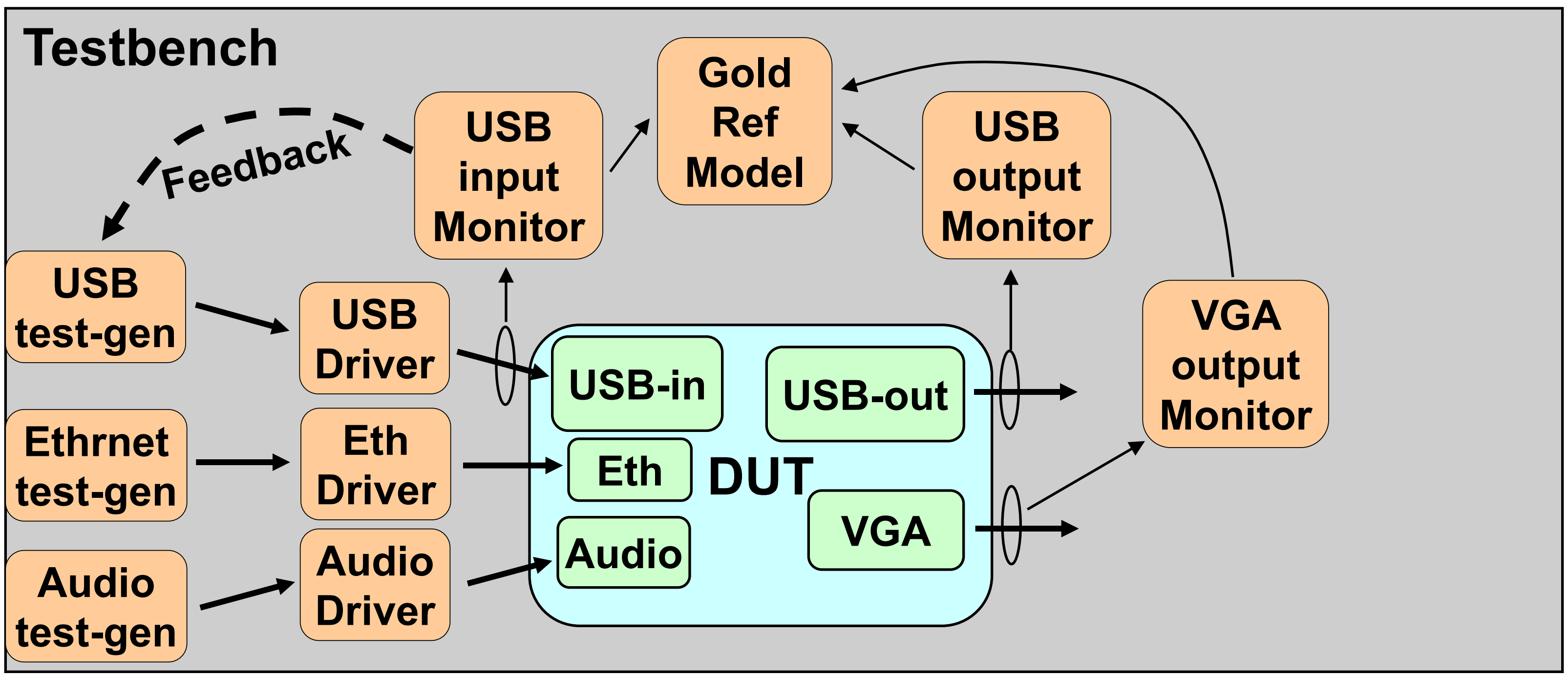
Testbench Environment Creation - Monitors

- Observe interface signals and translate to high level transactions
  - Log files; feedback for test-generator
- Check that transactions have legal structure
  - Leave the data correctness for later (assertions & reference model)



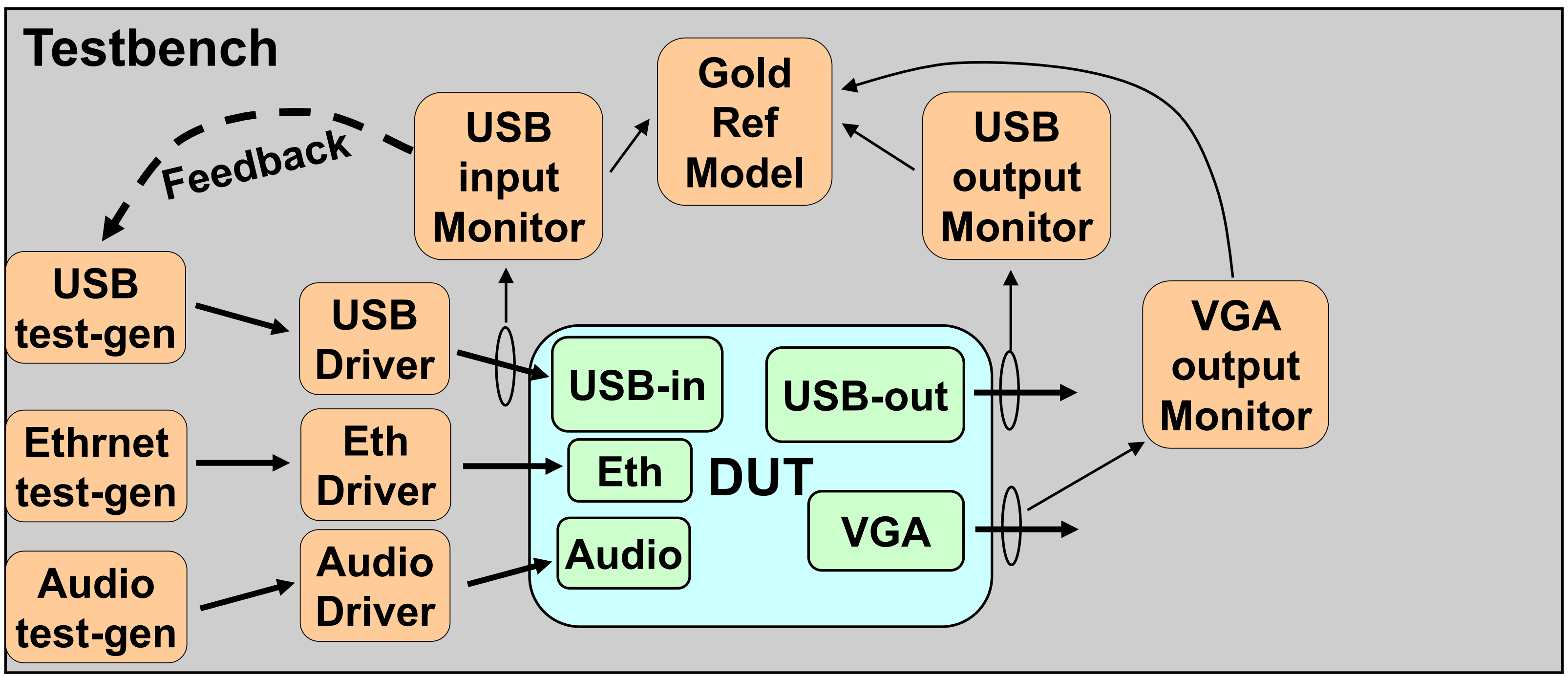
Testbench Environment Creation - Scoreboard

- A golden model of the design, also known as a "scoreboard"
  - Most likely written in a higher language (Vera, 'e', C, CPP)
- Answers the question: Is this a correct output transaction?



Testbench Environment Creation - Scoreboard

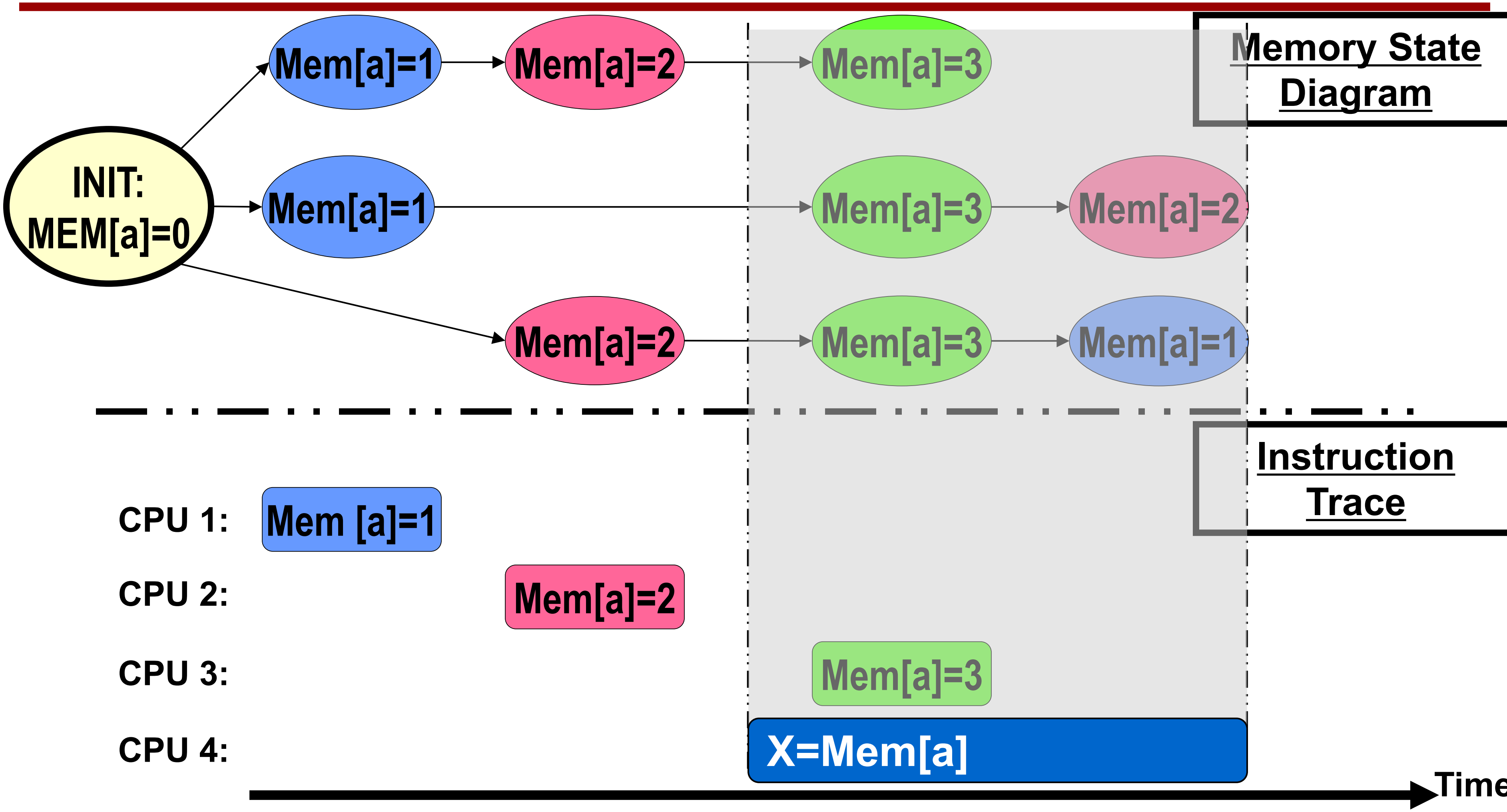
- Scoreboards are important since they enable the decoupling of the stimulus generation and the correctness checking
  - Otherwise, stimulus sequence must have a predicted result



Checking For Correctness Is Not Always Simple

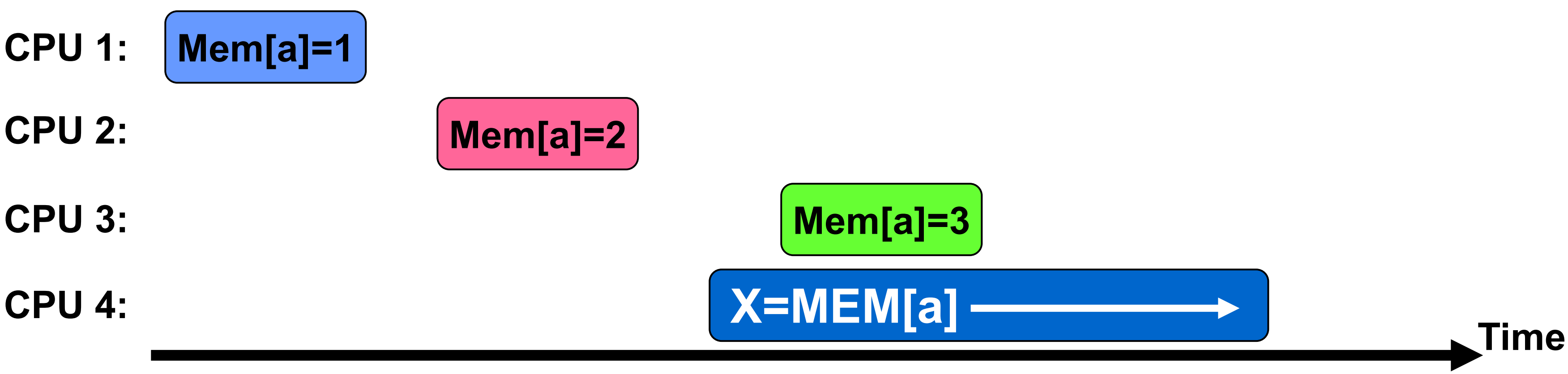
- Scoreboards may be difficult to write
  - Modeling a multiplier is one thing… modeling a chip multiprocessor with multiple interacting CPUs can get tricky
  - Cycle-by-cycle behavior can make results very different
    - But not necessarily wrong!
  - Need to “relax” the scoreboards
- What about errors that don’t (immediately) break the output?
  - A bad multiplication that is saved to the cache
  - A bad value that is not frequently used
  - Assertions!

At Times Scoreboards Are Close To Impossible
Example Of A CMP Memory System



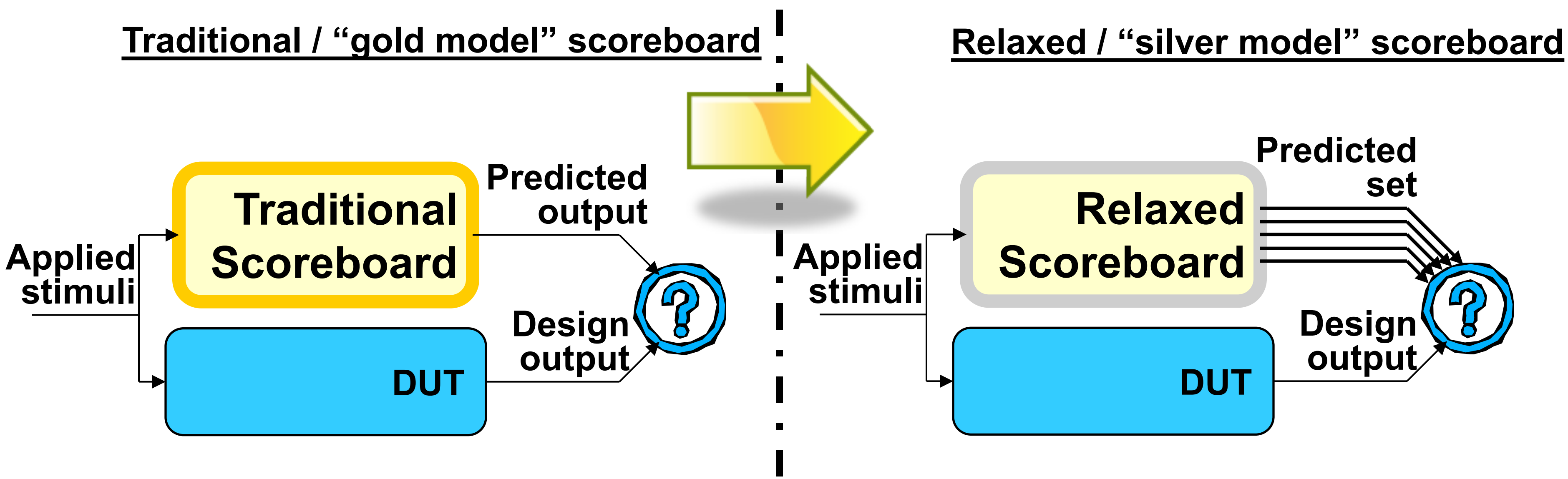
“Relaxing” The Scoreboard

- Does a scoreboard really need to predict whether X is 1, 2, or 3?
  - Aren’t they all valid answers?
  - It seems that “what is an invalid value?” is an easier question
- Option: Don’t use a scoreboard
  - Algorithmic solution: keep the complete trace and verify it



What Is A "Relaxed Scoreboard"?

- Like other golden models, this is a global design checker but…
- It does not compare an observed value to a single known value. Instead we keep a set of "possibly correct values"



Local Scope Checkers: Assertions

- Assertion Aided Simulation

- What are assertions?
  - Properties involving internal signals (and / or primary I/Os)
  - Mostly local to (relatively) small blocks
  - Similar to C code:

    *int *ptr = (int*)malloc(5*sizeof(int)); // allocate memory*

    assert(ptr); // make sure that allocation succeeded

- Simple examples:
  - “A FIFO must never raise full and empty at the same cycle”
  - “In a state machine, if state!=IDLE, then within 10 cycles, state must be IDLE again”

Why Assertions?

- Assertions concisely describe complicated temporal scenarios
  - Formed as a mathematically phrased sentence (rule or property)
    - IEEE P1850 Property Specification Language (PSL) standard
    - IEEE P1800 System Verilog → Assertion standard

- Act as traps for “bad” or “misbehaving” scenarios

- Enables the designer to set verification “hedging” in places where manual reasoning is difficult
  - Write assertions in parallel with writing the verilog code
  - For any test fail/bug found, add an assertion so that next time it is caught faster

Assertions' Structure

- General structure:

  assert (property) [$display ("Pass...");] else $error ("Fail...");

- Immediate assertions:
  - Detects a behavior <u>at (every) time instance</u>
  - E.g., Intersection: If GreenLight_EW then !GreenLight_NS

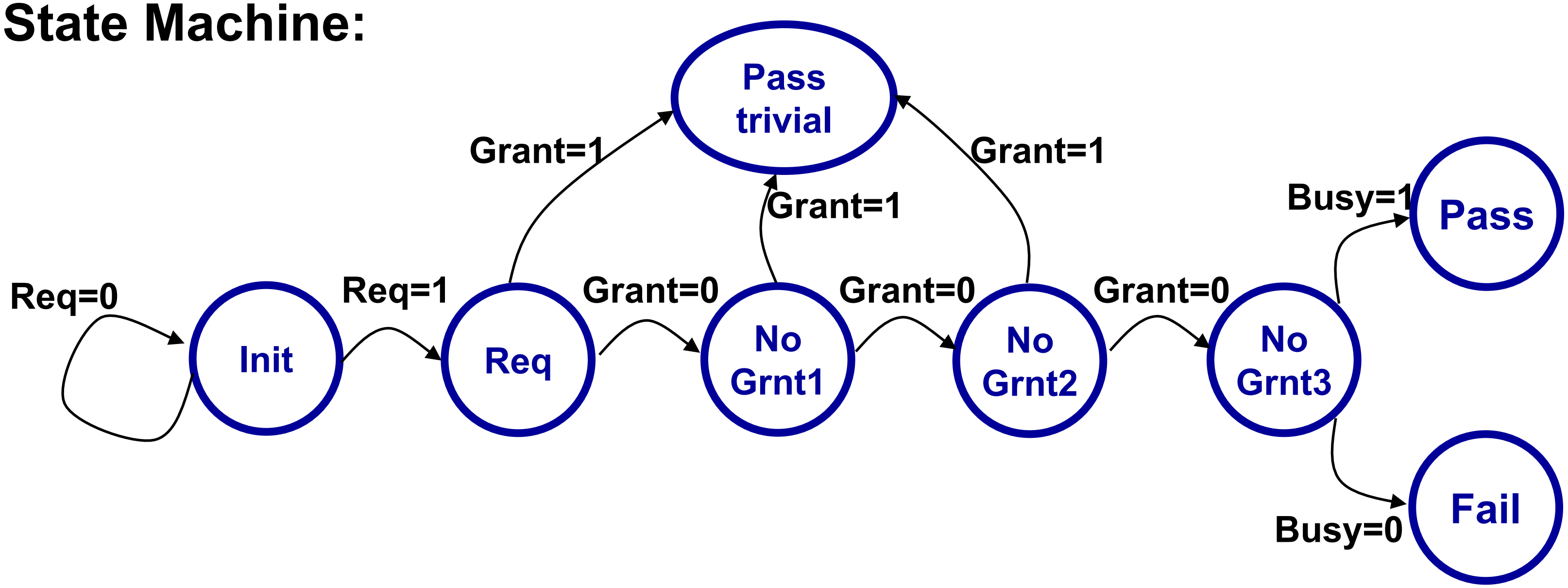
- Concurrent assertions:
  - Detects a behavior <u>over a period of time</u>
  - Given that sequence1 happened, sequence2 must happen
  - E.g., if (request at this cycle) then (ack in one-three cycles)

Assertions Are State Machines

English:

If a ‘request’ is raised, followed by 3 cycles with no ‘grant’, then at the next cycle, ‘busy’ must be on.

State Machine:

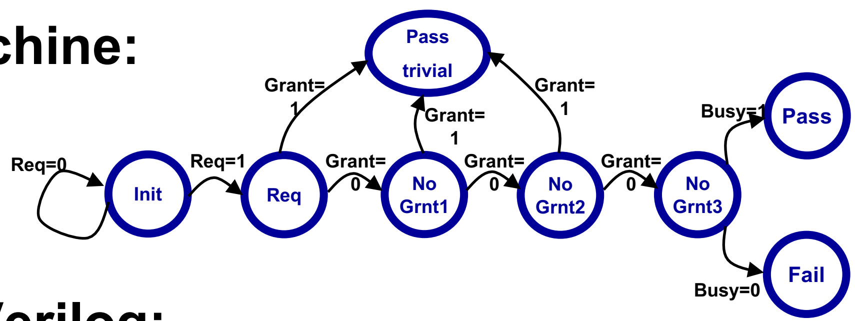


"Assertions Language" Enables Concise, Mathematical Description

English:

If a 'request' is raised, followed by 3 cycles with no 'grant', then at the next cycle, 'busy' must be on.

State Machine:



System Verilog:

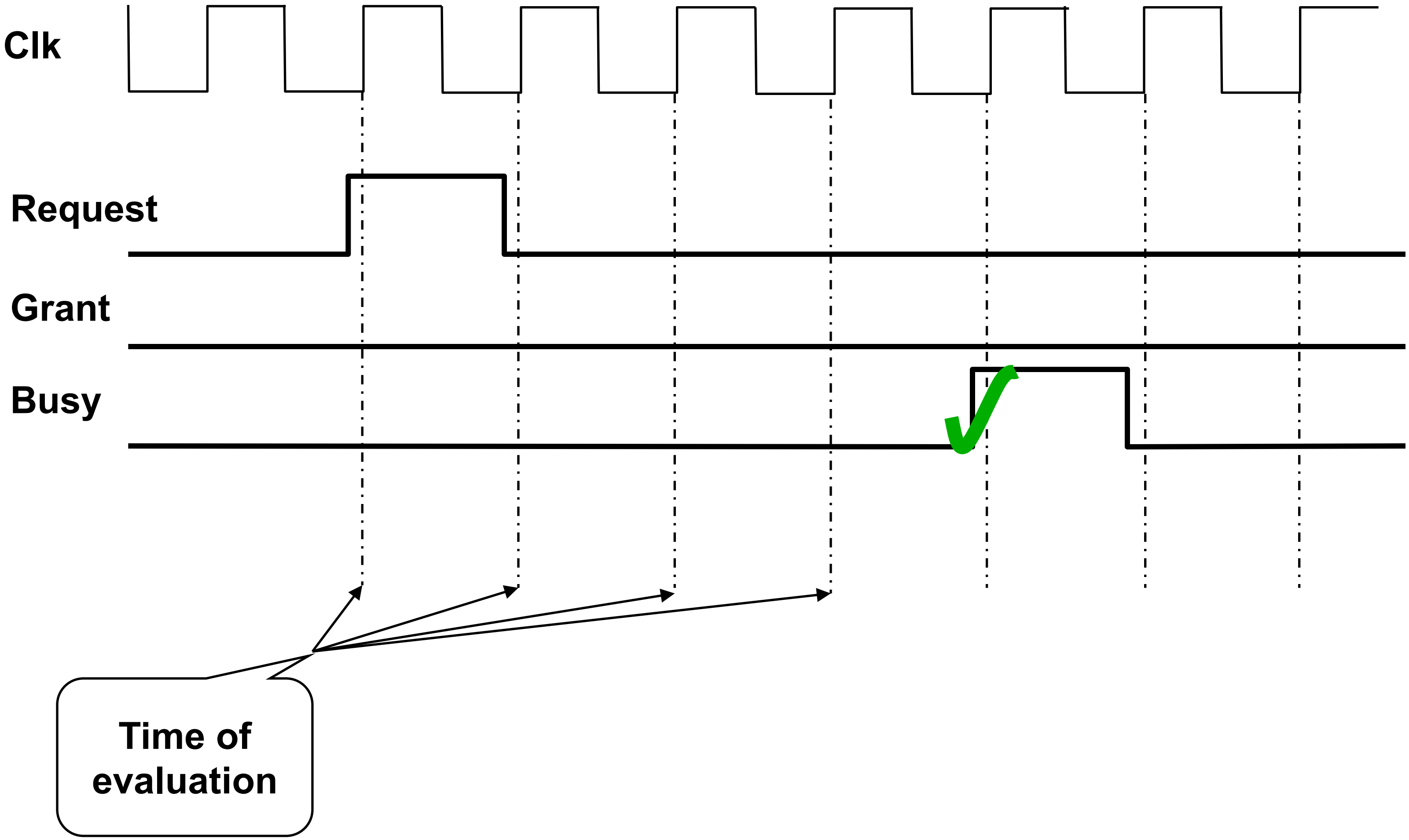
assert property (@(posedge Clk)

(request ##1 (!grant)[*3] |=> (busy)))

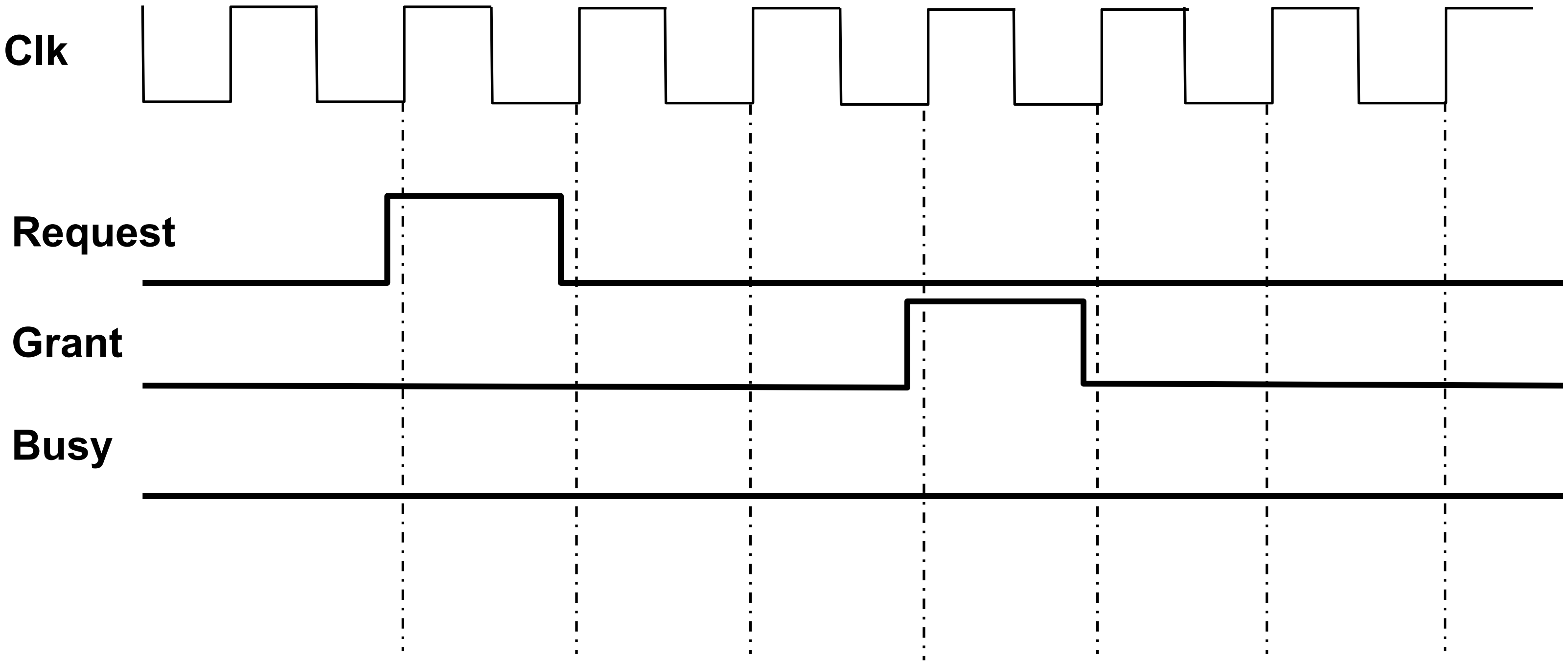
In this sequence, request is high at cycle N, then grant is low at cycles N+1, N+2 and N+3

Then at the next cycle (N+4) this event must happen

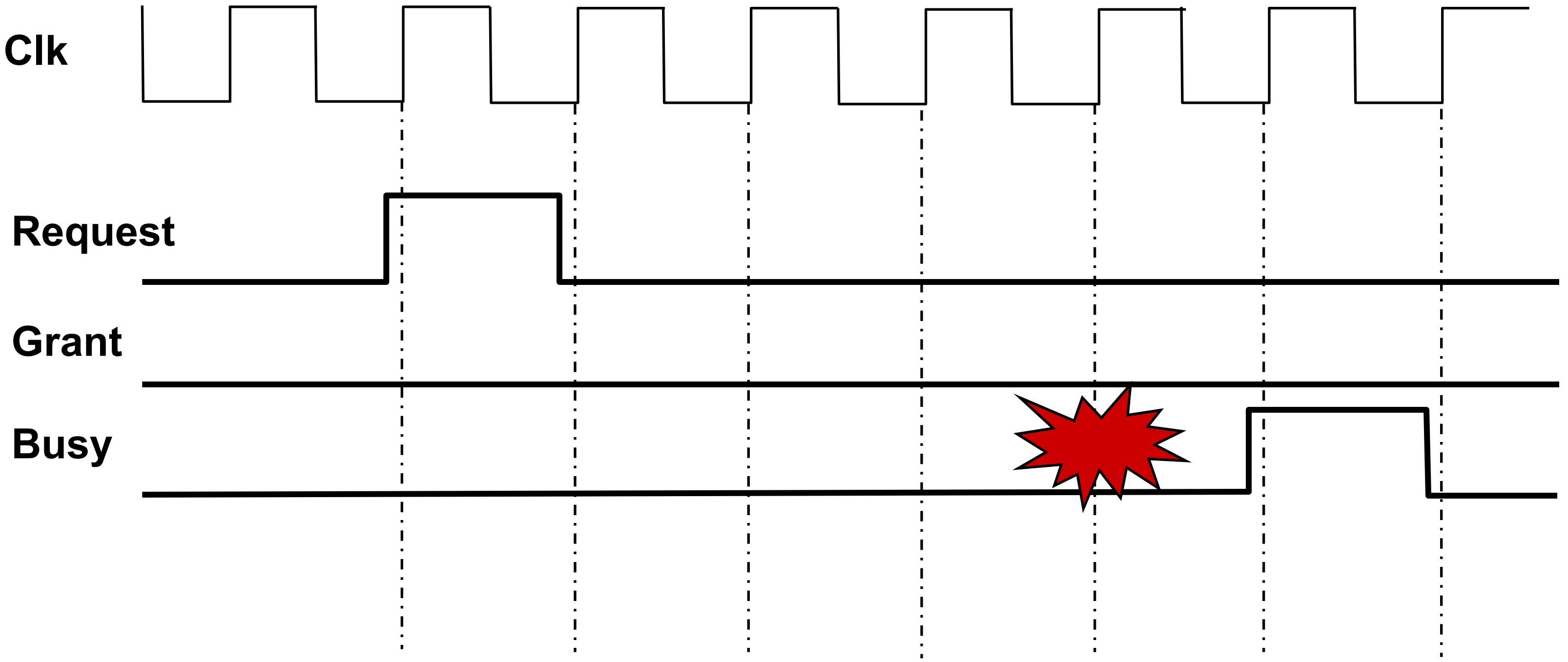
Assertions In Time Domain: Pass Instance



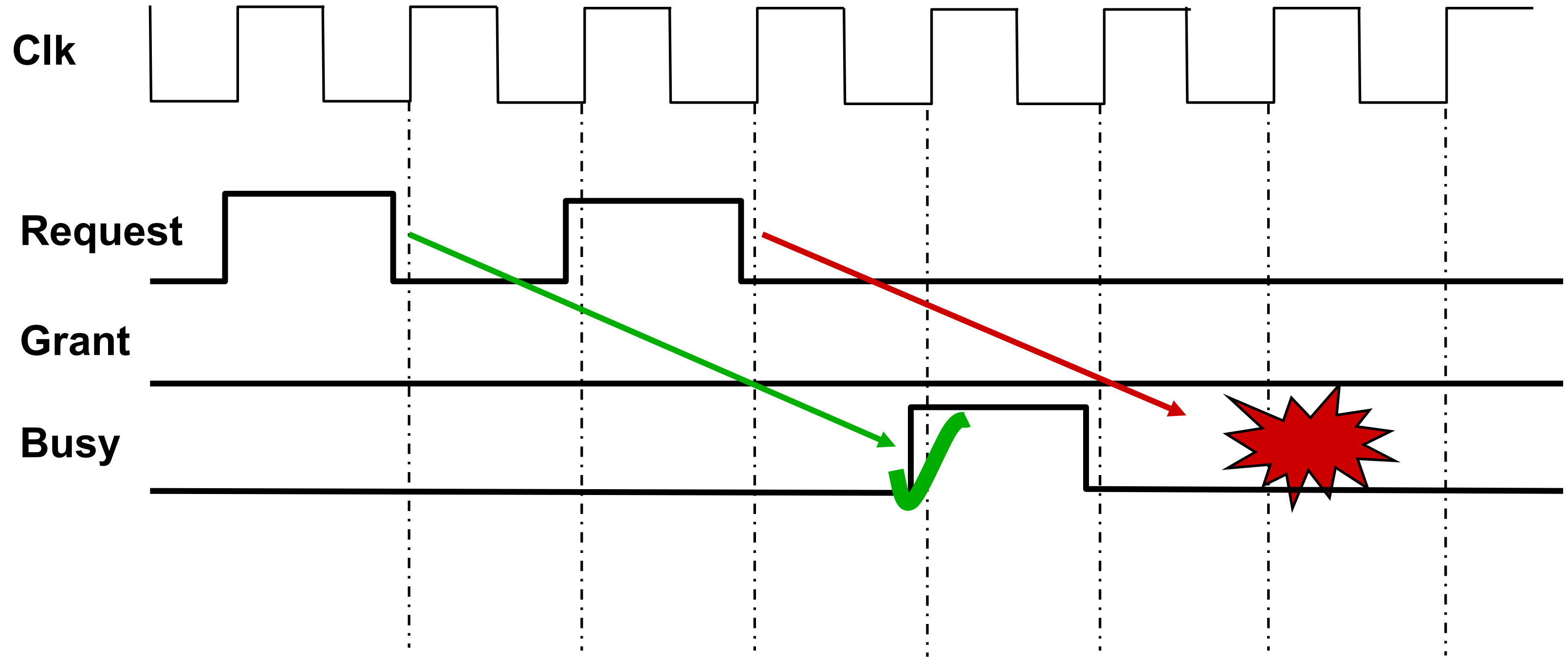
Assertions In Time Domain: Pass Trivially



Assertions In Time Domain: Fail Instance



Assertions In Time Domain: More Then One Instance



- The simulation engine must generate multiple instances of the state machine
  - The longer the pattern the more concurrent copies
  - Some “rules” might be a compute burden if not careful!

+ Tricky Class Exercise – Which is Better?

English:

If a request (‘req’) is raised, acknowledge (‘ack’) must be raised within 10 cycles

Suggested Assertion 1:

```
assert property ( @(posedge Clk)
    ($rose(req)  ##[0:10]     |=>       $rose(ack))  );
```

Suggested Assertion 2:

```
assert property ( @(posedge Clk)
    ($rose(req)       |=>       ##[0:10]  $rose(ack))  );
```

+ Tricky Class Exercise, Cont'd

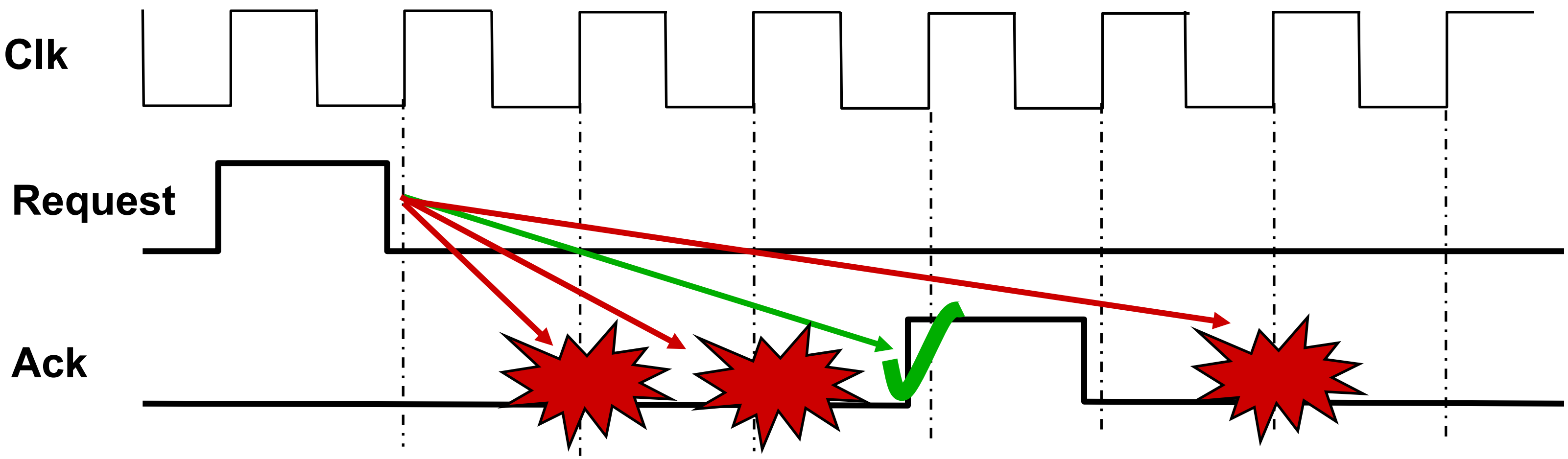
- Turns out those are VERY different assertions

- Suggestion 1 is equivalent to 10 assertions since the left hand side can be satisfied "easily":

```
assert property ( @(posedge Clk)
      ($rose(req)          |=>       $rose(ack))  );
AND
assert property ( @(posedge Clk)
      ($rose(req)  ##1   |=>       $rose(ack))  );
AND
…
AND
assert property ( @(posedge Clk)
      ($rose(req)  ##10|=>         $rose(ack))  );
```

+ Tricky Class Exercise, Cont'd

- Suggestion 1 will give false alarms. Example:



+ Tricky Class Exercise, Cont'd

- Suggestion 2 is what we wanted. It's equivalent is:

```
assert property ( @(posedge Clk)
        ($rose(req)        |=>        $rose(ack))  );
OR
assert property ( @(posedge Clk)
        ($rose(req)        |=> ##1  $rose(ack))  );
OR
…
OR
assert property ( @(posedge Clk)
        ($rose(req)        |=> ##10 $rose(ack))  );
```

Assertions Vs. Scoreboard

Assertions

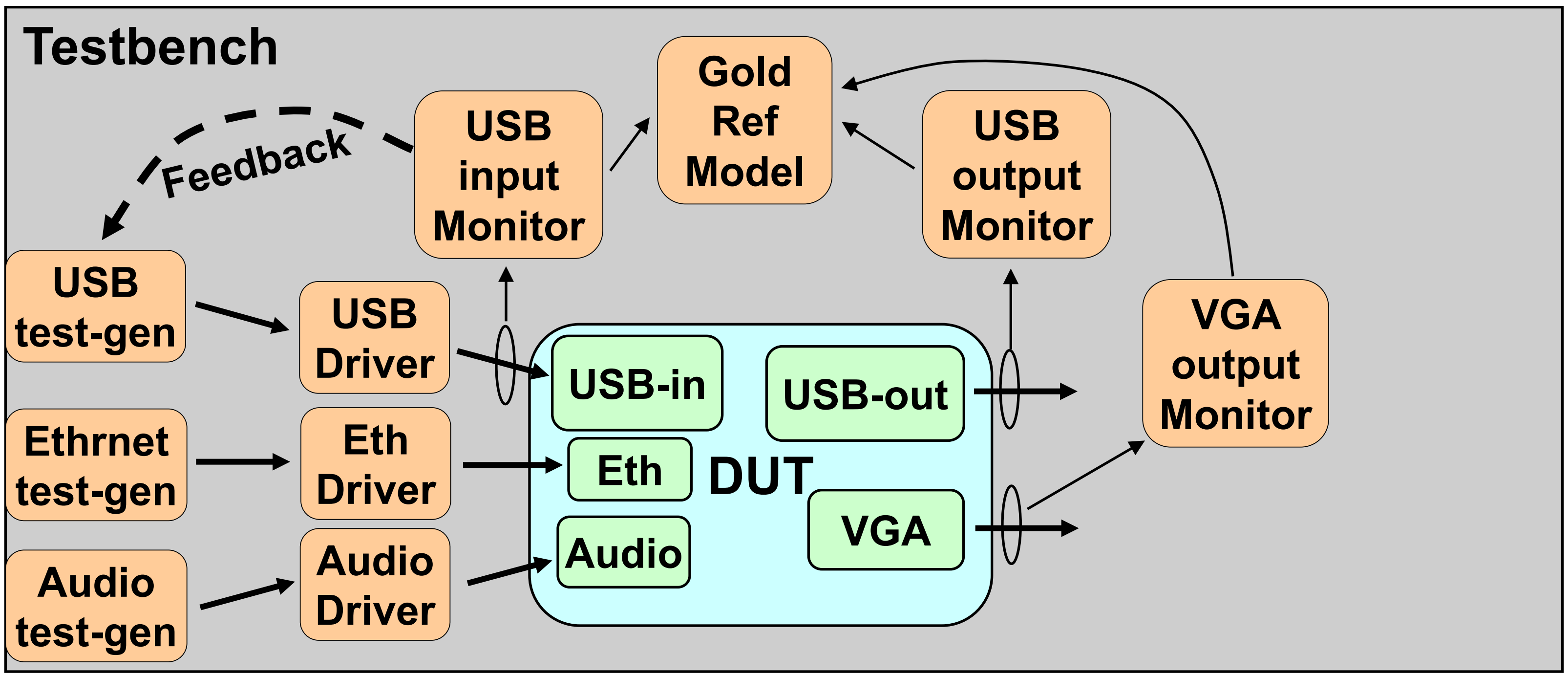
- Local scope
- Simple to develop
  - Designers can embed within the verilog/vhdl code
- Described in a formal or mathematical language
- Only capture a very small set of scenarios. Most assertions would pass trivially most of the time…
- Stimulating more scenarios leads to better verification

Scoreboard

- Global, end to end checker
- Complicated to develop
  - Can be as complicated as the DUT it self
  - Takes a lot of time to tune
- Ref. model – essentially this is a program which is as good as the engineer who wrote it
- Verifies every output (but only as good as the input given…)
- Stimulating more scenarios leads to better verification

When The Testbench Is Ready: Remember GIGO

- The best testbench, with the best scoreboard and the best assertions is only as good as the verification patterns that are sent through it



When The Testbench Is Ready: Remember GIGO

- The best testbench, with the best scoreboard and the best assertions is only as good as the verification patterns that are sent through it
- A module/state/interaction that was not exercised is broken!

Garbage In → Garbage Out

Are We There Yet?
Are We There Yet?
Are We There Yet?
Are We There Yet?

- When does the **design** part in the project end?
  - When all partitions have been designed
- When does the **verification** part in the project end?
  - NEVER
  - Revise: When the next version of the chip is taping out…
- So… how do I know that the design is ready for tape-out?
  - Use '*coverage'* metrics to estimate how well you tested it
- Coverage is the question:
  - "How much of the possible cases have I exercised?"
  - "Which scenarios am I still missing?"

Coverage Rule Of Thumb

- Low coverage $\rightarrow$
  Bad. Need to run more scenarios!

- High coverage $\rightarrow$
  Means it's time to use a better metric…

Types Of Coverage

- Toggle coverage
  - Simplest case: “Interesting” signals must have 0 and 1
  - Good measure for signal connectivity
  - High toggle coverage doesn’t mean good verification quality
    - E.g., “32bit AND-tree”: in=32’h0 and in=32’ffff_ffff would translate to 100% toggle coverage on all input & output & internal signals

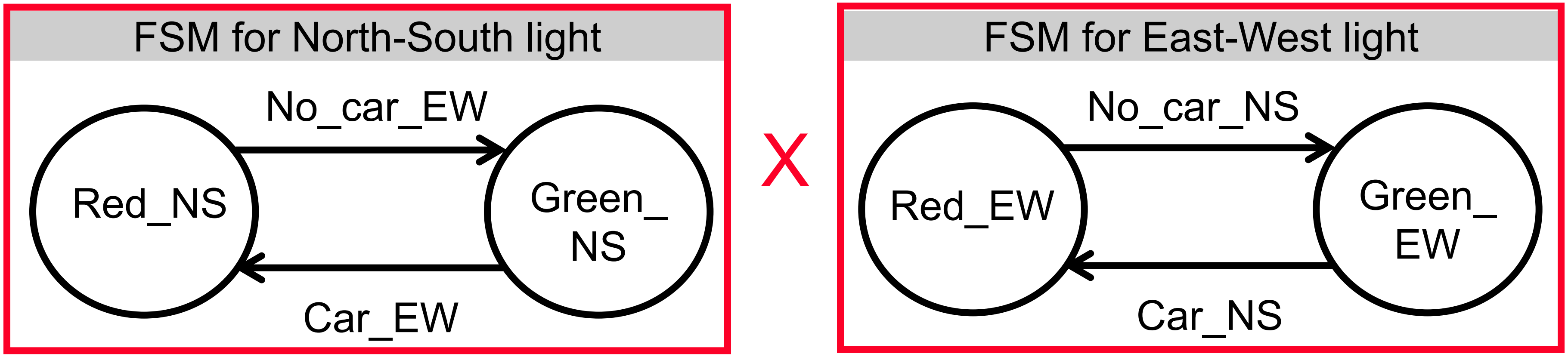
- Statement Coverage
  - High statement coverage is a MUST
  - Again, high coverage doesn’t mean good verification quality
    - Consider statement: if (x > y) then z = 1 else z = 0;
      - What if is x bigger then y for all stimuli?

Types Of Coverage (cont.)

- Branch coverage
  - Keep track of branch conditions exercised
  - Better than statement coverage but still has several problems
    - e.g., combinational logic may be specified as either "if-else", "case" statements or simple AND-OR expressions

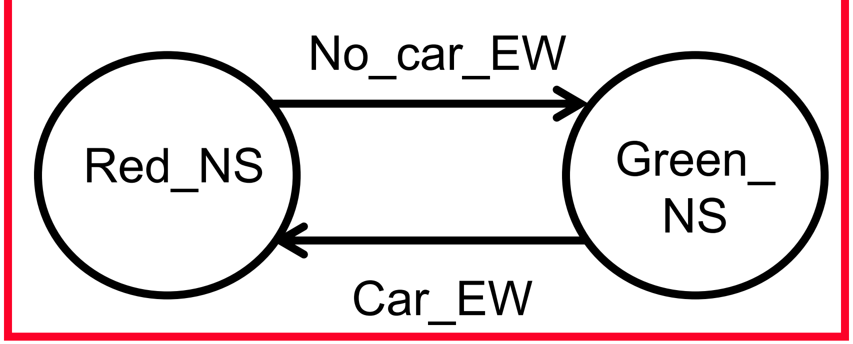
- Finite State Machine (FSM) coverage
  - State coverage – % states visited
  - Transition (arc) coverage – % transitions exercised
  - Issue
    - Complexity increases with "interacting" FSMs
    - Traffic light controller example (next)

Example: Traffic Light's FSM Interactions



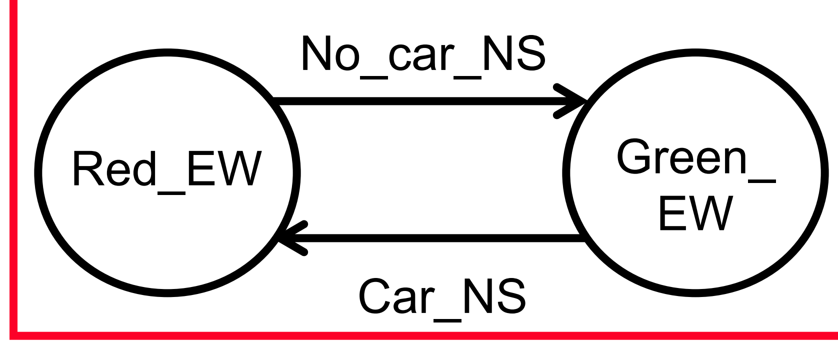
Traffic Light Example to Illustrate FSM Interactions

FSM for North-South light

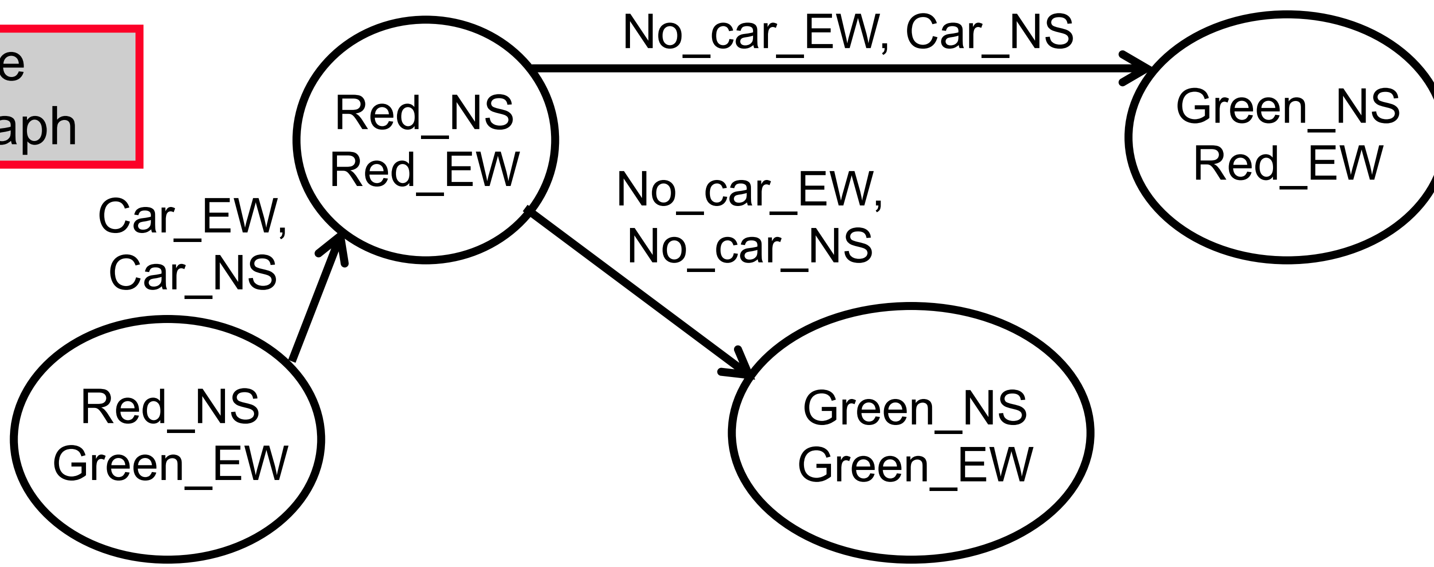


X

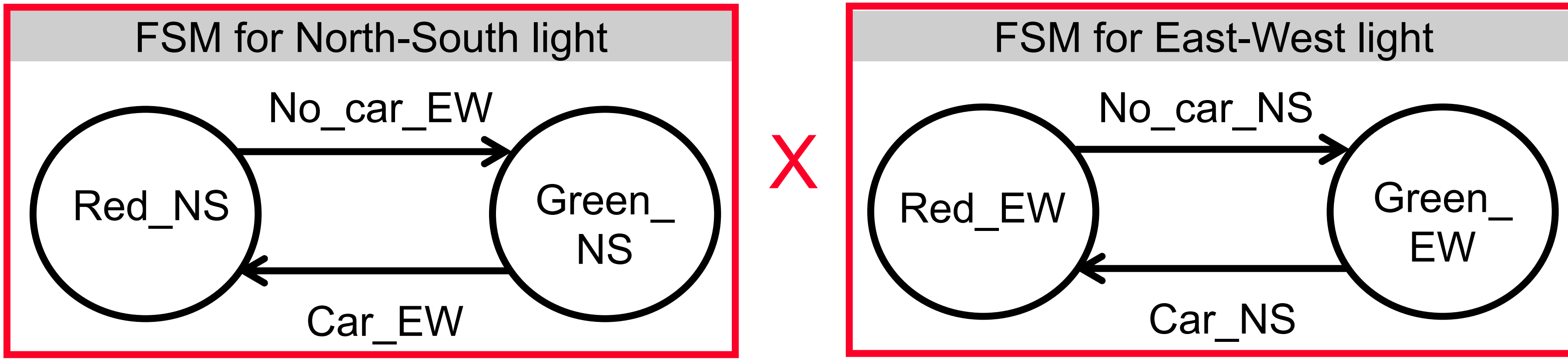
FSM for East-West light



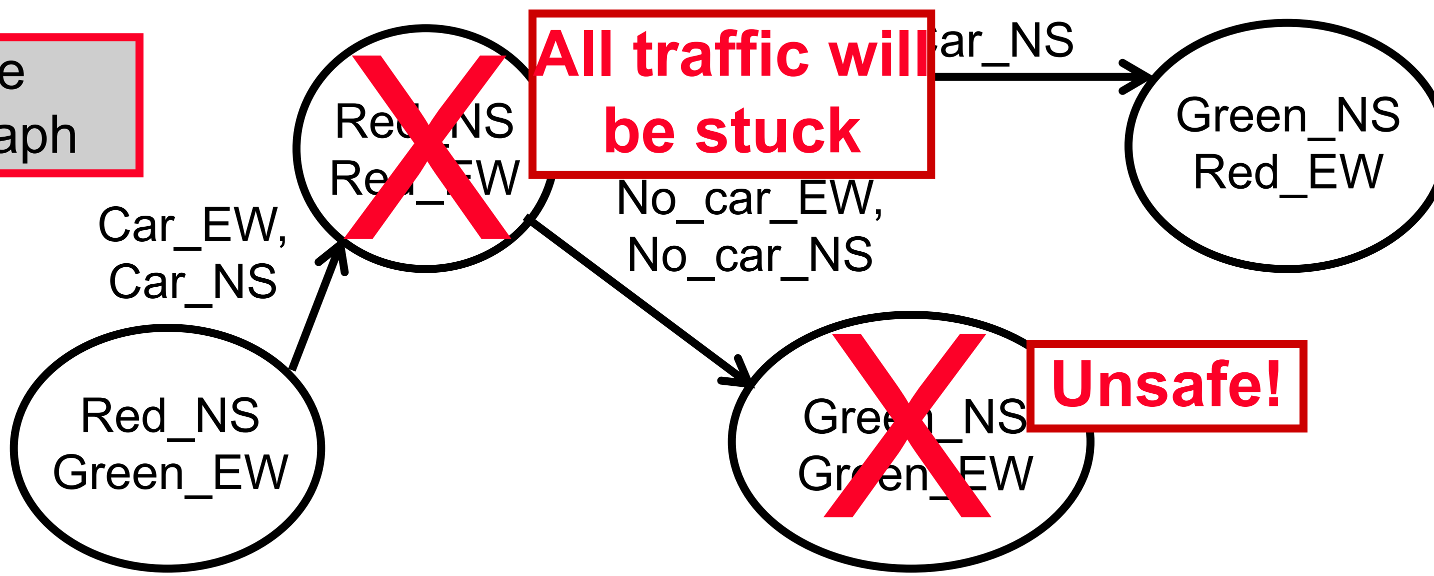
Partial State Transition Graph



Traffic Light Example to Illustrate FSM Interactions



Partial State Transition Graph



Types Of Coverage (cont.)

- Coverage Assertions
- Assertions capture events and cross product of events
  - Can be used to monitor interesting events
  - (In system verilog) Simply use the ‘cover’ key word instead of the ‘assert’ key word

- Coverage assertion examples (in English):
  - Cover the case of two packets arriving back to back
  - Cover the case of packet request while FIFO is full
  - Cross-product of the two

- **Need to be very familiar with the design to define the “interesting” coverage events**
  - **I consider it as being “The Devil’s Advocate”**

Aside: Assertions Can Find Traffic Light Bug

Public_safety_assertion:

```
Assert property (
    @(posedge Clk)
    (Green_NS  |-> !Green_EW)
    )else $error(Boom!)
```

No_deadlock_assertion:

```
Assert property (
    @(posedge Clk)
    (Red_NS  |-> !Red_EW)
    )else $error(Deadlock!)
```

This seems like a trivial property. Can we mathematically prove/disprove that this assertion never fires and be done with it?

Aside: Assertions Can Find Traffic Light Bug

```
Public_safety_assertion:
Assert property (
    @(posedge Clk)
    (Green_NS  |-> !Green_EW)
    )else $error(Boom!)

No_deadlock_assertion:
Assert property (
    @(posedge Clk)
    (Red_NS  |-> !Red_EW)
    )else $error(Deadlock!)
```

This seems like a trivial

In some cases the answer is YES!

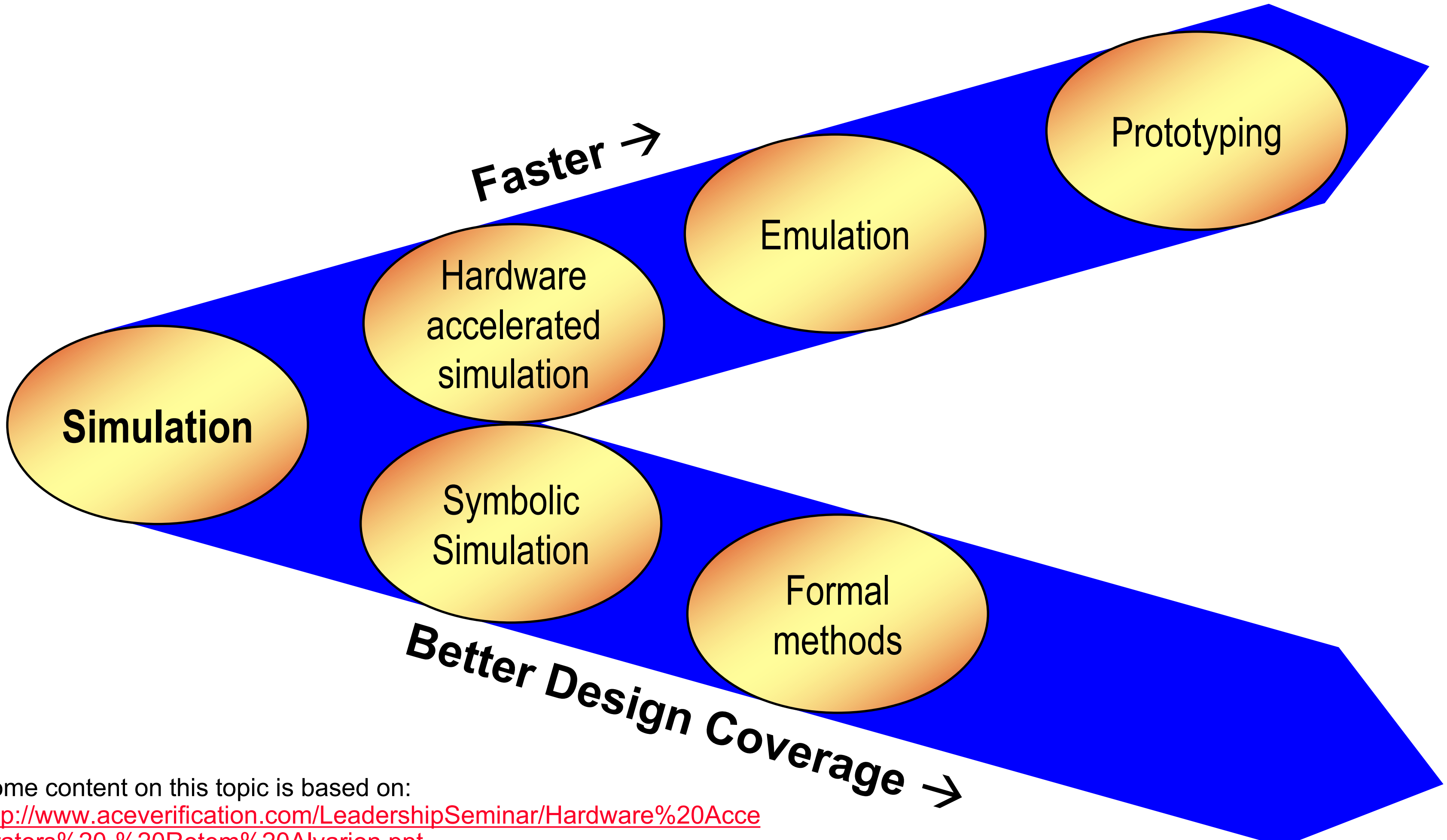
We'll see more details in "Formal Verification"

Getting More/Faster Coverage

- Logic simulation is slow
  - It will never be fast enough as designs keep growing in size
  - E.g., Smart Memories chip multiprocessor statistics:
    - Silicon runs at **200,000,000** cycles/sec (200MHz)
    - RTL simulation runs at **150** cycles/sec
    - A 5M cycles test run finishes in 0.025sec on silicon and in 9hr on RTL simulation (gate level simulation takes ~10x longer)

- Question: Can we do better/faster?
  - Yes, by using specialized hardware
  - Yes, by mathematically proving our design is correct

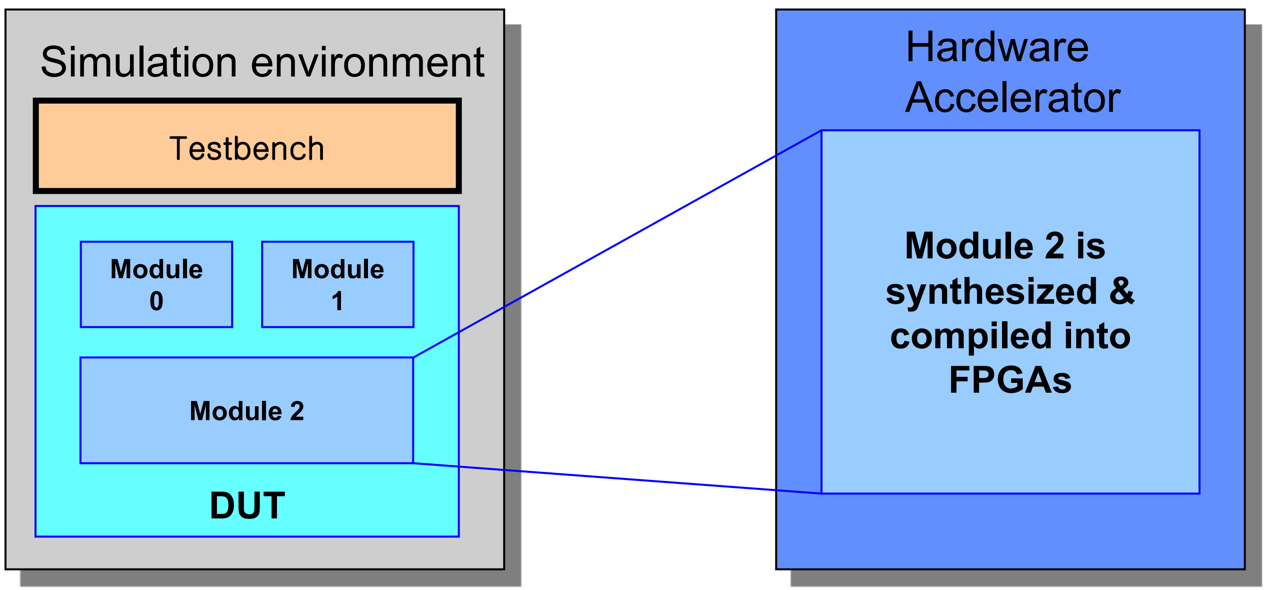
Faster Coverage vs. More Coverage



Some content on this topic is based on:
http://www.aceverification.com/LeadershipSeminar/Hardware%20Accelerators%20-%20Rotem%20Alvarion.ppt

Hardware-Accelerated Simulation

- Simulation performance is improved by moving the time-consuming part of the design to hardware
  - Specialized logic processors with custom instruction sets
  - Design mapped into Programmable logic (e.g., FPGAs)

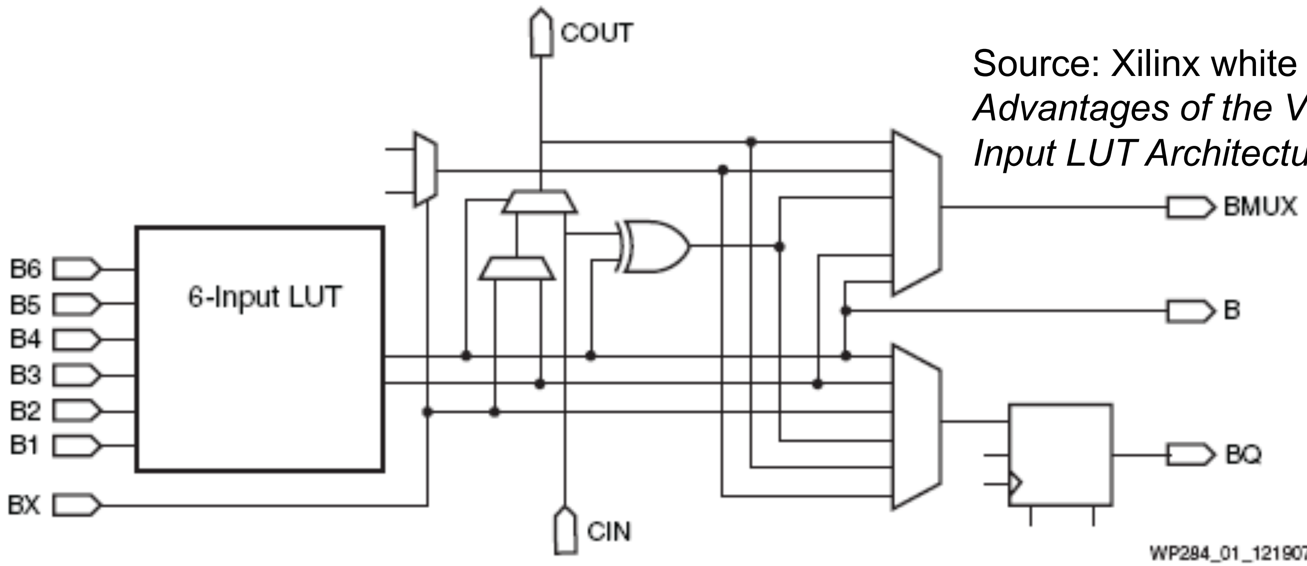


Hardware-Accelerated Simulation / Emulation

- Pros
  - Fast (10x-1000x faster then simulation)
  - May enable verification on real target system
    - E.g., boot operating system

- Cons
  - Setup time overhead to map RTL design to hardware is high
    - weeks to set the working environment
    - hours for every mapping
  - SW-HW communication speed can hurt performance
  - Low visibility to signals within the hardware

+ Hardware Emulation Example: Basic FPGA Unit



Source: Xilinx white paper, *Advantages of the Virtex-5 FPGA 6-Input LUT Architecture (2007)*

Figure 1: **Virtex-5 FPGA 6-Input LUT Architecture**

- FPGA = Field Programmable Gate Array
- Capable of implementing any function of 6 inputs and numerous combinations of one or two smaller functions

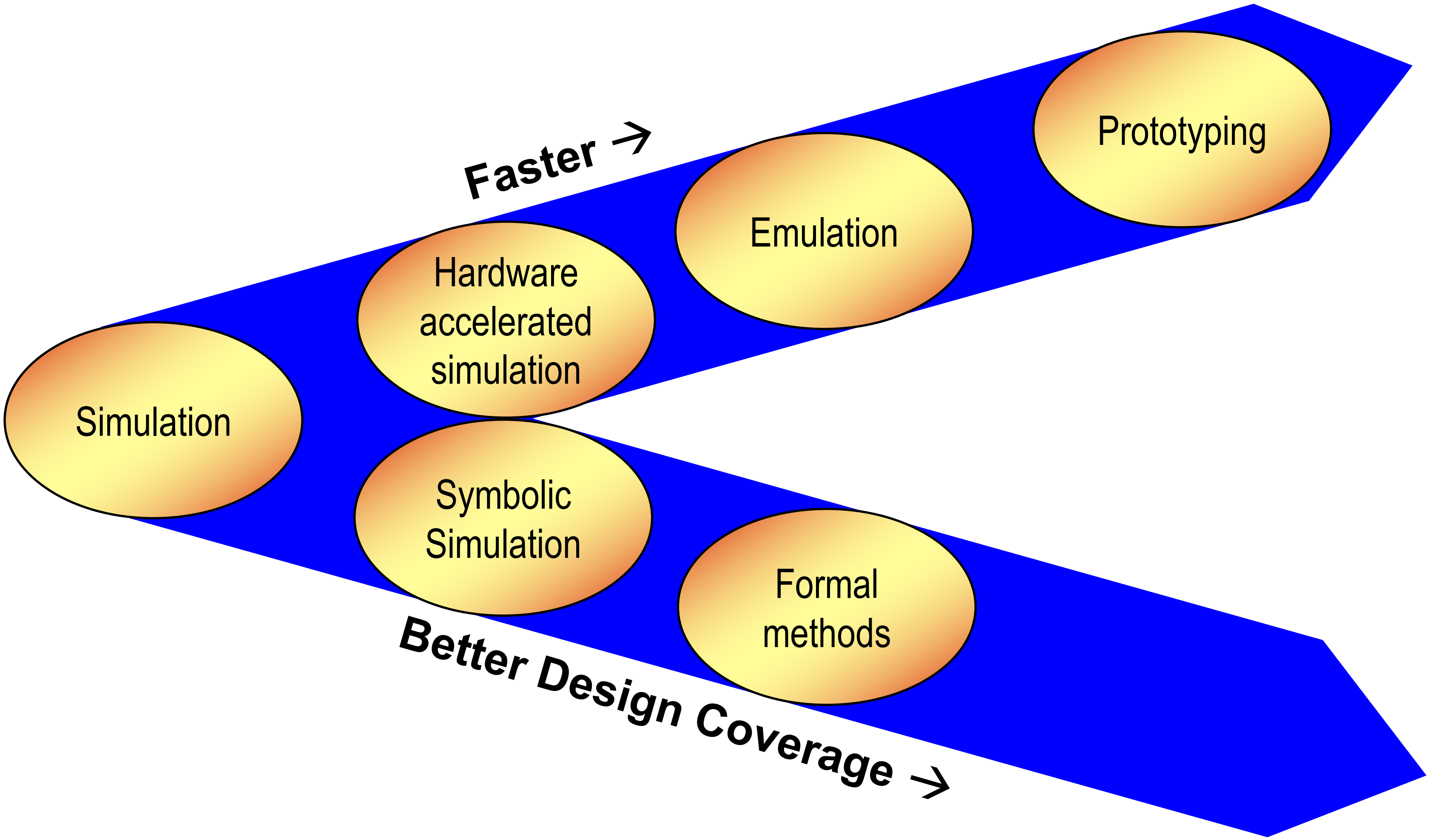
Prototyping

- Special (more dedicated and customized) hardware architecture made to fit a specific application.

- Pros
  - Higher (than emulation) clock rate
  - Small form factor (can bring along for a demo in the office ☺)

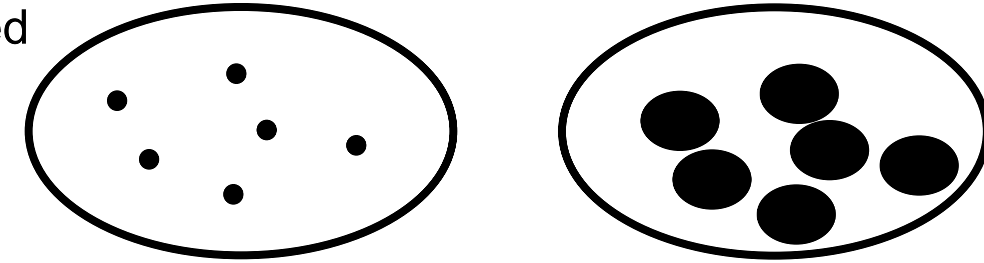
- Cons
  - Huge overhead in developing the prototype
  - Not flexible for design change

Faster Coverage vs. More Coverage



Exercising More Of The Design In Less Tests: Symbolic Simulation

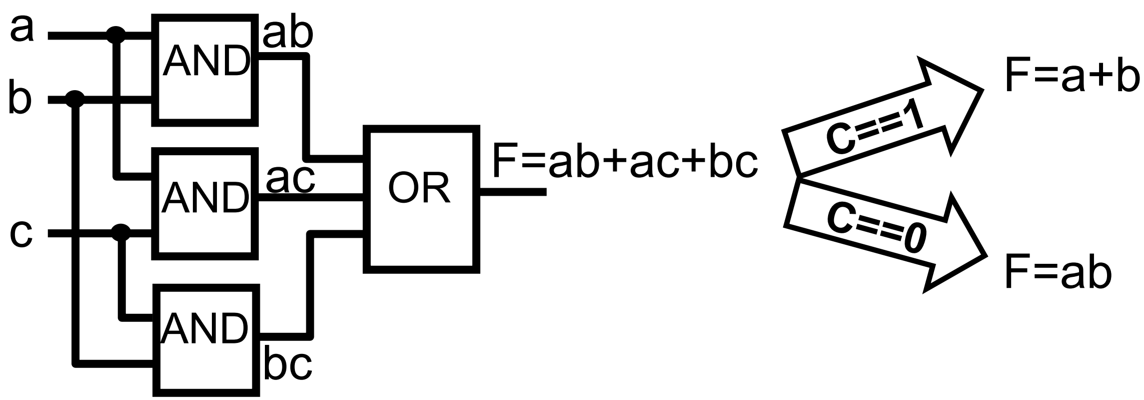
- At the end of a simulation run we know how the system respond to one test pattern
- Question: Can we simulate multiple runs at once ?
- Symbolic simulation idea
  - Keep some variables “symbolic”: Which variables? – heuristic
  - Apply 0s and 1s to other variables
- Advantage:
  - for *k* symbolic variables, 2^k possibilities simulated at once



Symbolic Simulation Example

- Symbolic simulation with a and b as symbolic inputs and c=1
- Suppose objective is to show that this circuit produces:

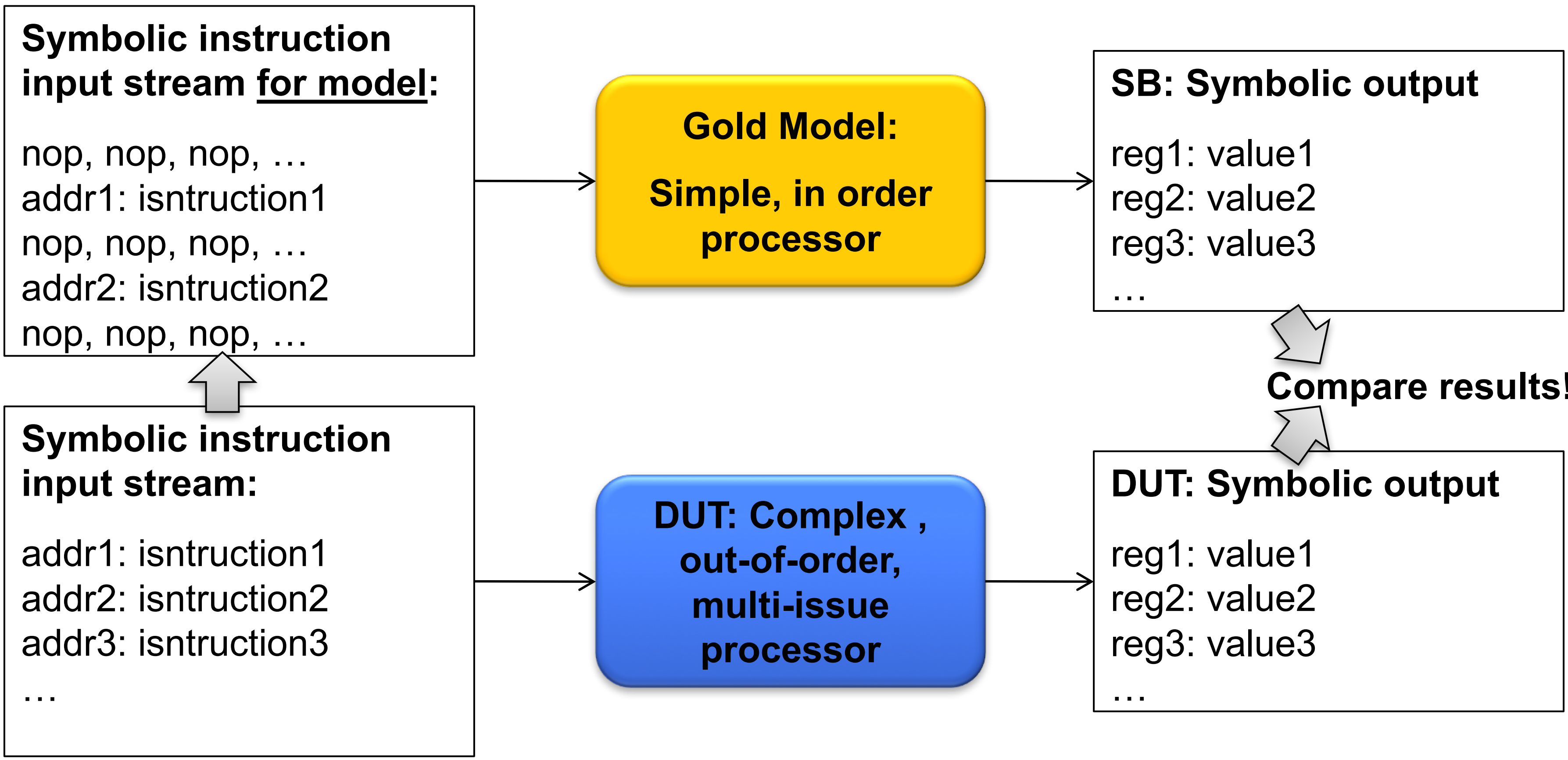
$$F(a,b,c) = ab + c\ (a \text{ XOR } b)$$



  - Simulate with c==1 → F=a+b (compare with ab+ab'+a'b)
  - Simulate with c==0 → F=ab (compare with ab)

Key Issue: If the result is symbolic, how do we know it's correct?

A Symbolic Scoreboard



Exercising The Entire Design Space: Formal Property Verification (FPV)

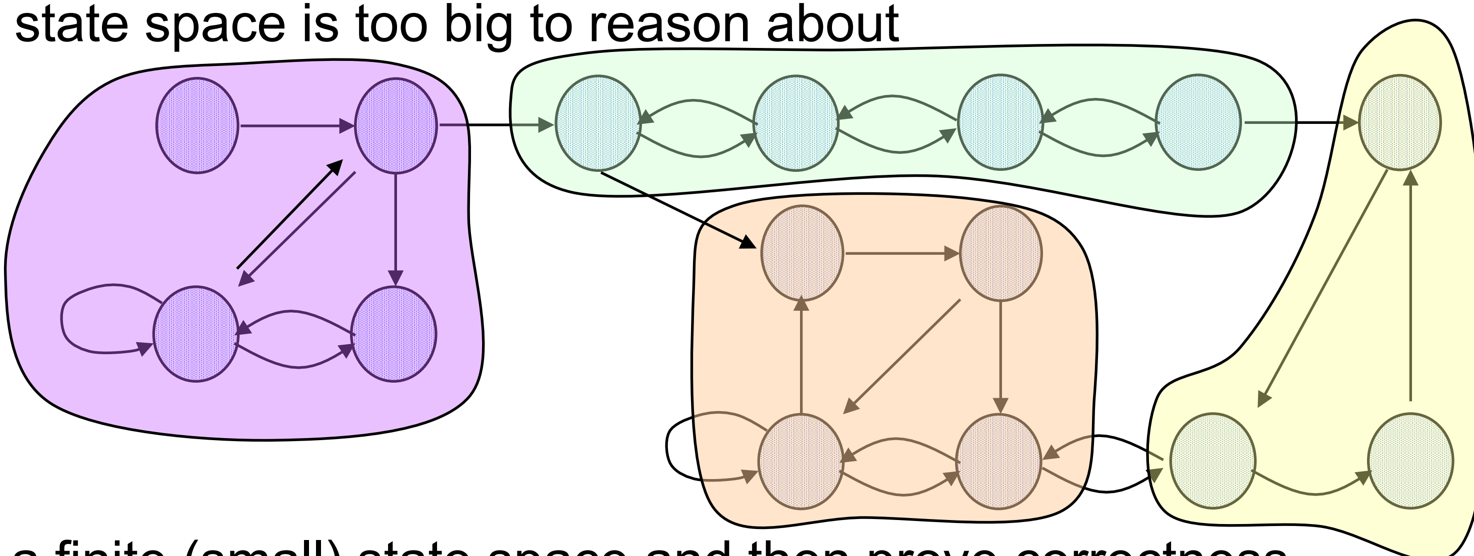
- Mathematically proving the correctness of the design
- "Prove" that a property holds for the ENTIRE state space
  - Unlike simulation where no guarantees can be made
  - Or symbolic simulation where partial space is covered
- Advantages
  - Can find corner cases not exposed by simulation
  - Identifies hidden assumptions
  - Bug hunting: generally provides counter-examples
- Shortcomings
  - Problem size (potentially) grows exponentially with state

Formal Verification Methods

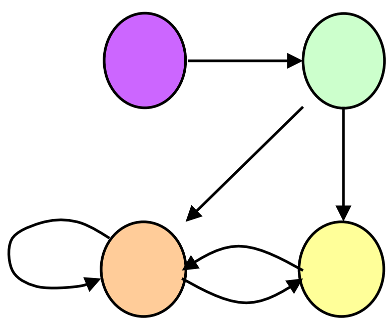
- **Model checking**: "... the process of checking whether a given structure is a model of a given logical formula..." (Wikipedia)
- **Computation tree logic (CTL)**: "…its model of time is a tree-like structure … there are different paths to the future…" (Wikipedia)
- **Equivalence checking**: Are two circuits performing the same functionality, for all allowed inputs?
- Layman: Construct a graph or a formula and (dis)prove it!

Model Checking

- Design state space is too big to reason about

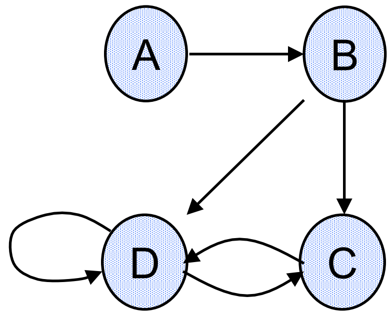


- Map to a finite (small) state space and then prove correctness

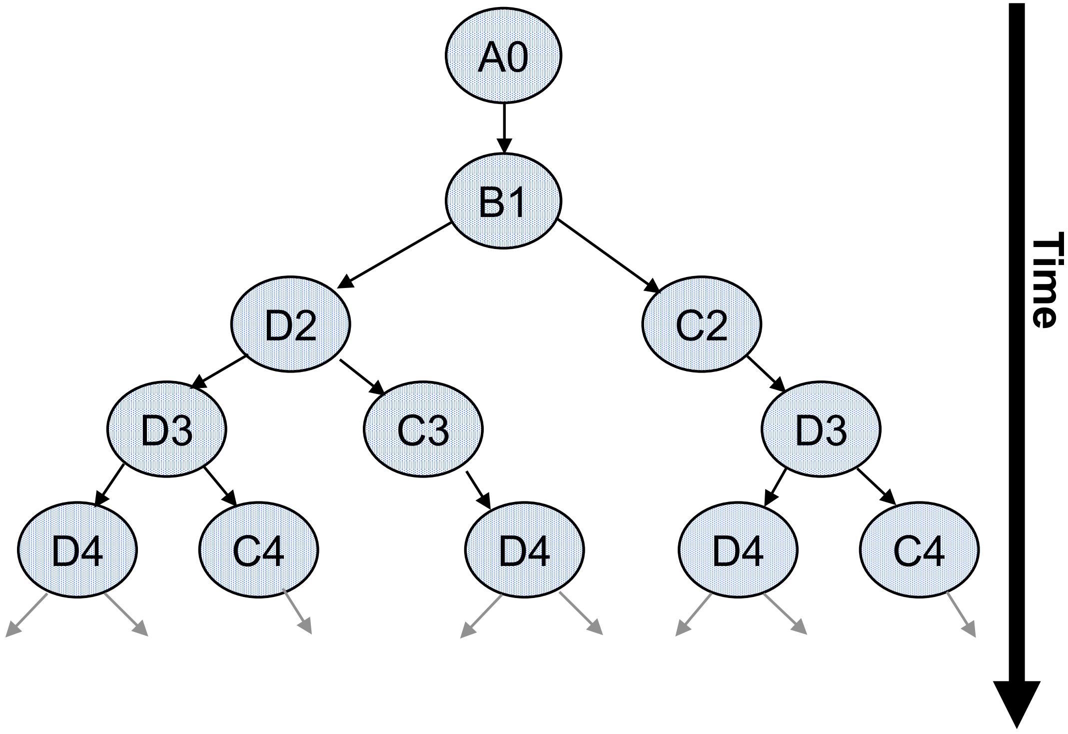


Computation Tree Logic

Final State Machine



CTL = Span FSM in time as a tree graph

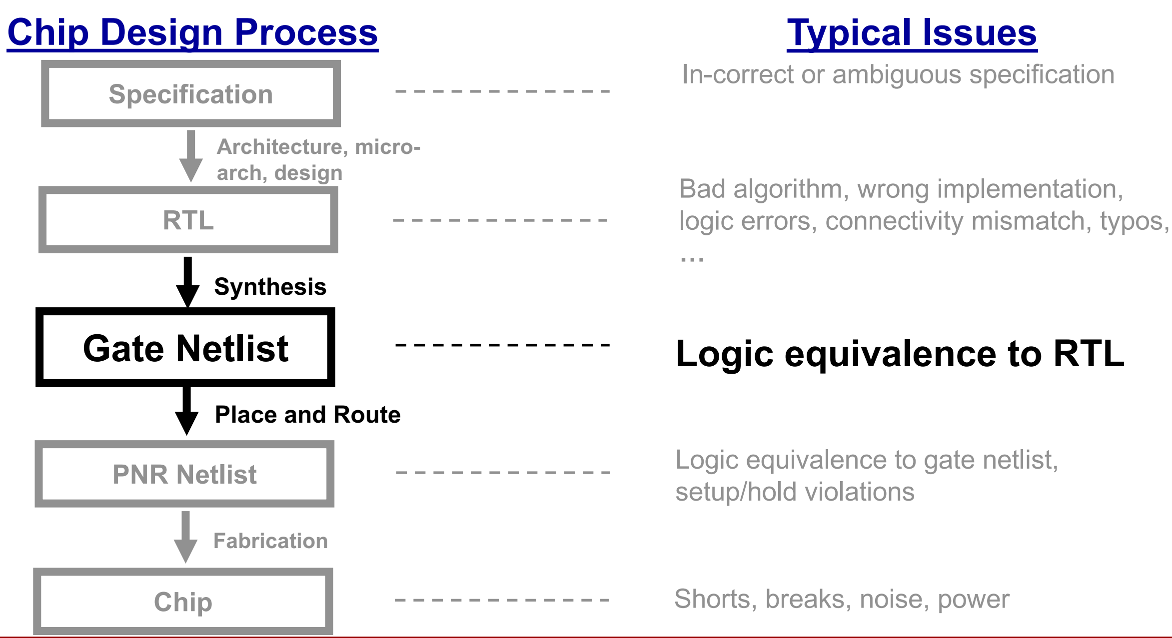


Property Examples in CTL

- Property p must hold <u>globally</u> (for all states): Gp (Safety property)
  - e.g, traffic light must not be green on 2 conflicting directions
- Property p must hold for at least some <u>future</u> state: Fp (Liveness property)
  - e.g., traffic light must become green eventually
- <u>For all paths</u> starting from now property p is true at some <u>future</u> time: AFp
  - e.g., If state != IDLE, it will eventually go back to IDLE
- There <u>exists</u> a path for which property p is true at some <u>future</u> time: EFp
  - E.g., if state != IDLE, there is a possible path for which it goes to IDLE

Formal Verification For Equivalence Checking

- When does this problem arise?
  - RTL vs. gate-level netlist vs. another netlist…
  - Custom designs for data-path circuits
  - Various flavors of implementations for the same logic function with various power, performance and area trade-offs



Formal Verification For Equivalence Checking

- When does this problem arise?
  - RTL vs. gate-level netlist vs. another netlist…
  - Custom designs for data-path circuits
  - Various flavors of implementations for the same logic function with various power, performance and area trade-offs

- Given
  - An *n*-input combinational logic function *f*
  - An implementation

- Verification question
  - Does the implementation A correspond to function *f*
  - Alternatively, does implementation A match implementation B?

SAT

- Boolean Satisfiability Problem (SAT)
  - Given combinational function f, is there an assignment of 0s & 1s to input variables such that f is 1 ?
  - In other words: Is f Satisfiable ?

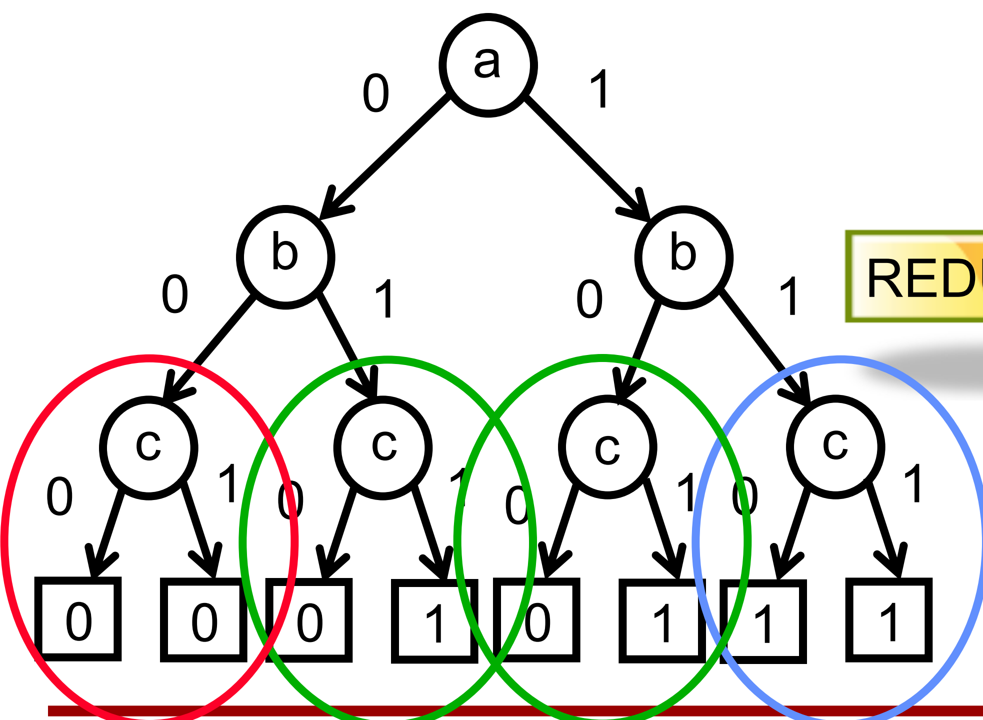
- Fundamental CS problem: theoretically difficult (NP Complete)

- Several well-researched efficient heuristic algorithms exist
  - Combination of BDDs, ATPG (Automatic Test Pattern Generation), efficient data structures

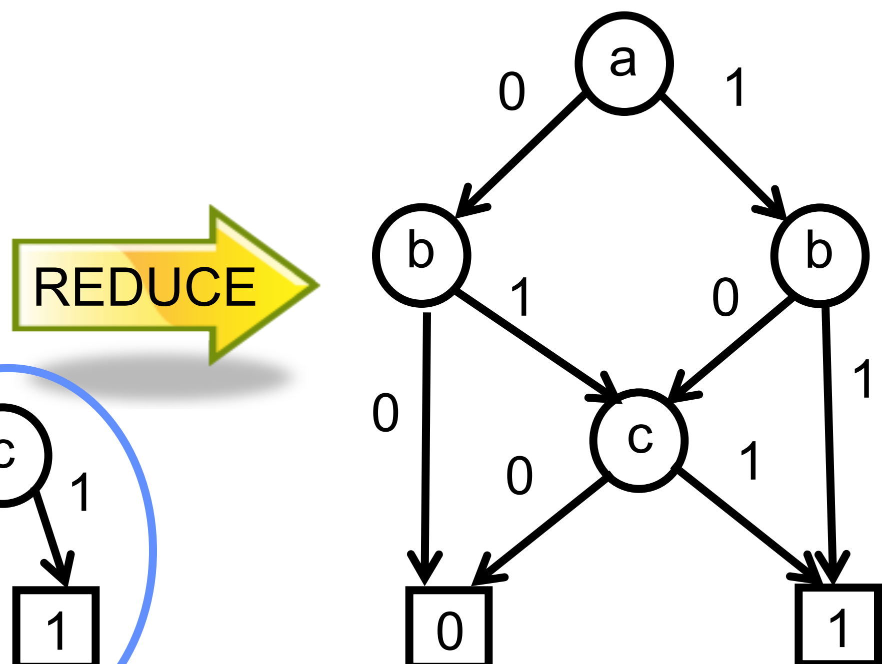
Binary Decision Diagrams (BDD)

- BDDs are an efficient representation of Boolean logic functions
- Given a function: f=ab+bc+ac and an order of computation: a → b → c

ORDERED BDD (OBDD)



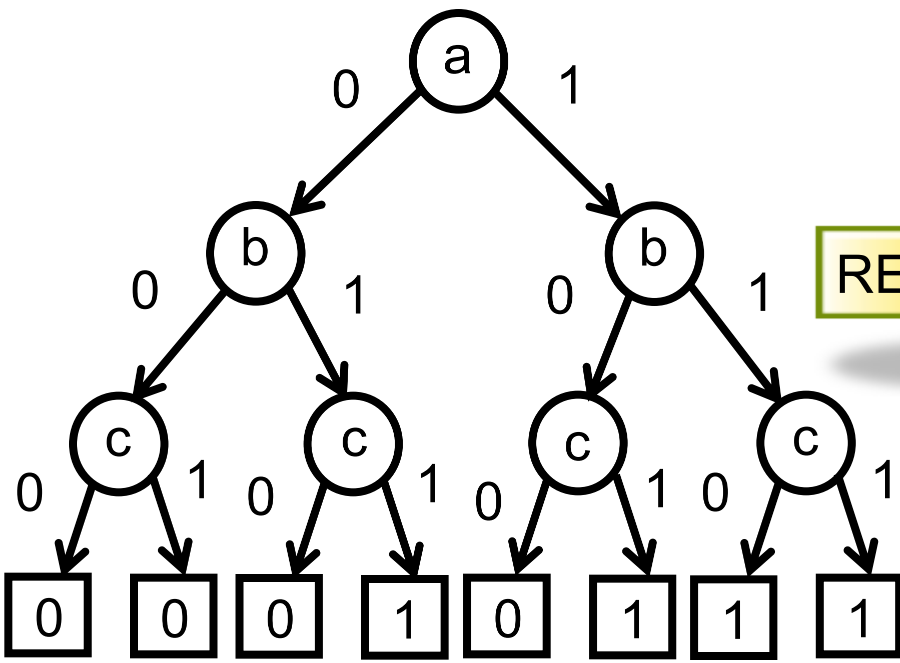
REDUCED ORDERED BDD (ROBDD)



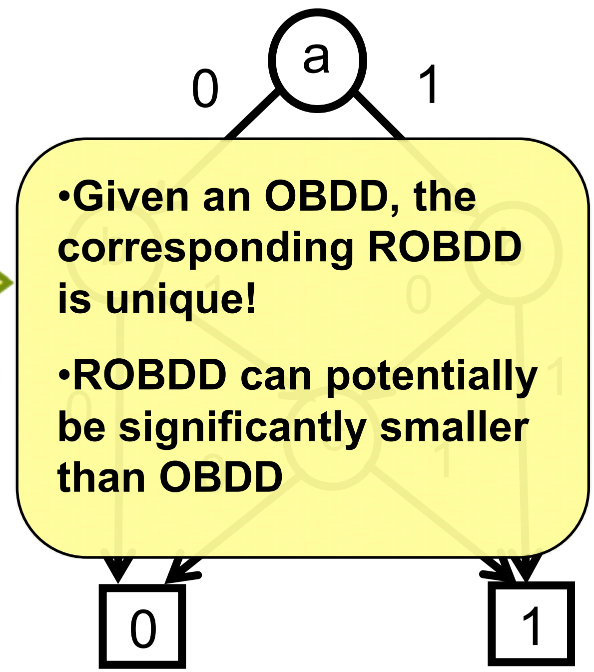
Binary Decision Diagrams (BDD)

- BDDs are an efficient representation of Boolean logic functions
- Given a function: f=ab+bc+ac and an order of computation: a → b → c

ORDERED BDD (OBDD)



REDUCED ORDERED BDD (ROBDD)



Using BDDs for Combinational Equivalence

Two logic functions: *f* , *g*

↓

Pick variable ordering (same order for both functions!)

↓

Create Ordered Binary Decision Diagram (OBDD) for each function

↓

Reduce each OBDD to its canonical form (ROBDD)

↓

Function *f* is equivalent to function *g* ←IFF→

their ROBDDs are identical

The End