Tutorial

Formal Assertion based Verification in Industrial Setting

Alok Jain

Cadence Design Systems
Noida

Raj S. Mitra

Texas Instruments
Bangalore

Jason Baumgartner

IBM
Austin

Pallab Dasgupta

Indian Institute of Technology
Kharagpur

Design Automation Conference, San Diego, June 8, 2007

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Agenda

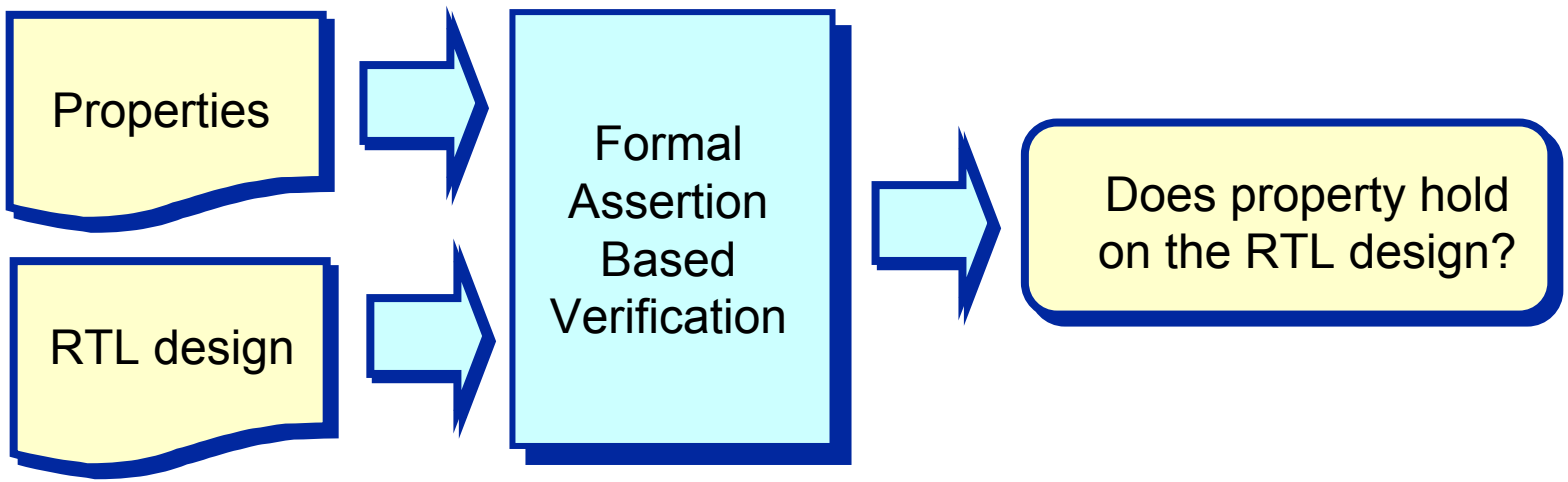
- Introduction to Formal Assertion Based Verification
  - Temporal Logics
    - LTL and CTL
    - PSL and SVA
  - Model Checking
    - CTL Model Checking
    - LTL Model Checking
  - Symbolic Model Checking
    - BDD and SAT based techniques
  - Abstractions
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Problem Statement

- Problem Statement
  - Verify an RTL design satisfies its functional requirements
- Traditional Solution – Logic Simulation
  - RTL design specified in HDLs
  - Requirements specified as test vectors
- Limitations of Logic Simulation
  - Limited set of test vectors
  - Exercise only a small fraction of design
  - Leads to undetected bugs
- Example – Floating point division bug in Pentium
  - 1012 test vectors
  - Cost $470 million

Alternative Solution – Formal ABV

- Formal Assertion Based Verification
- Mathematically reason about correctness of RTL design
- Properties specified in some form of “Temporal Logic”



- Benefits
  - Does not require user to provide test vectors
  - Does exhaustive verification

Agenda

- Introduction to Formal Assertion Based Verification
  - What is the problem statement?
  - LTL and CTL
  - PSL and SVA
  - Model Checking
    - CTL Model Checking
    - LTL Model Checking
  - Symbolic Model Checking
    - BDD and SAT based techniques
  - Abstractions
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

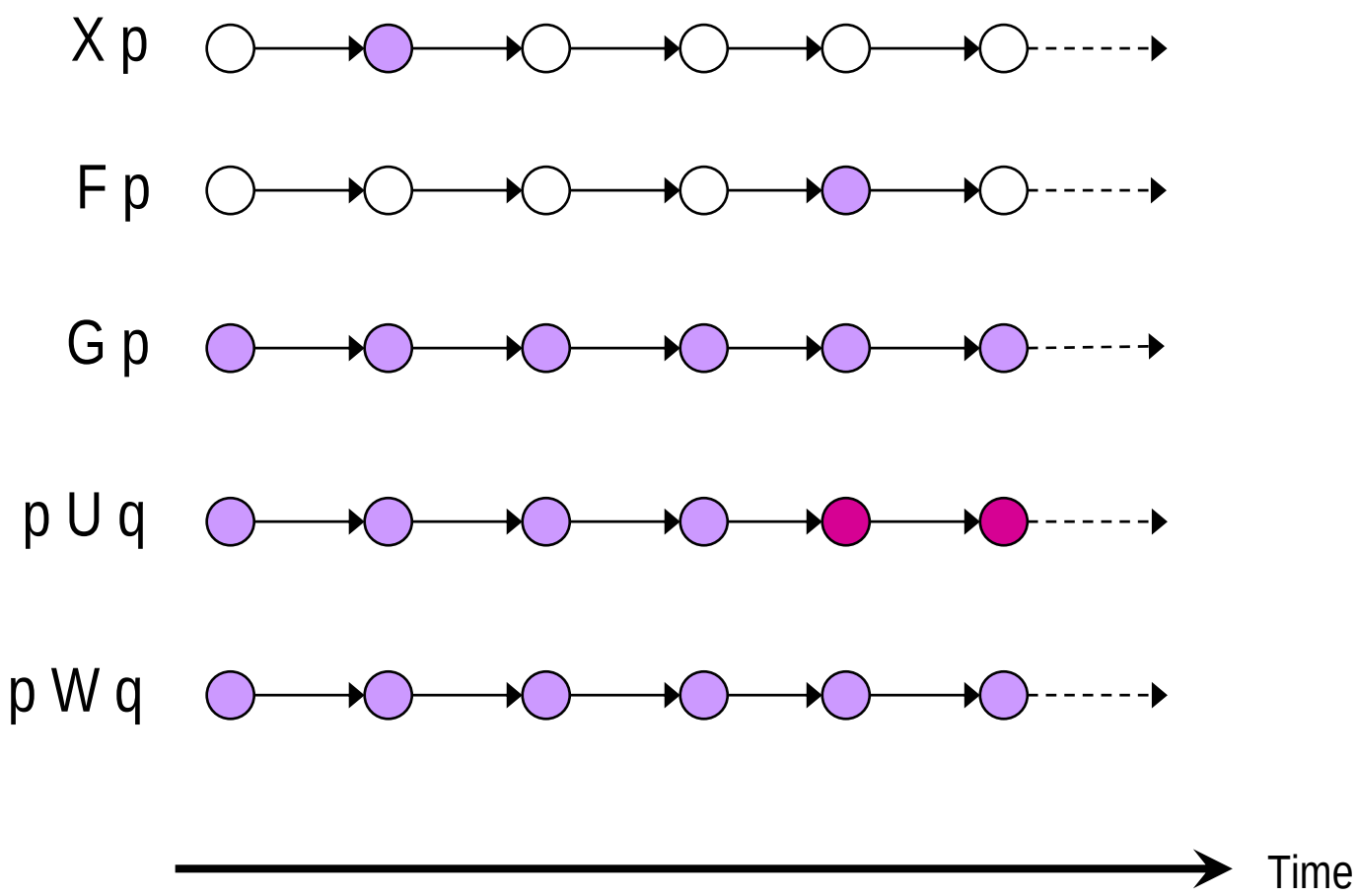
Temporal Logics

- Logic extended with the notion of time
- Reason about propositions qualified in terms of time
- Tradeoff between expressibility and complexity of verification
- Two popular forms of temporal logic for formal verification
  - Linear Temporal Logic (LTL)
  - Computation Tree Logic (CTL)

Linear Temporal Logic (LTL)

- Introduced by Pnueli in 1977
- Propositional Logic + discrete time
- Time is viewed as a single linear sequence of events
- Properties specified over a single path
- Temporal operators to represent discrete tim
  - p is a proposition – p should hold at current time
  - X p – p should hold at next time
  - F p – p should hold in the future
  - G p – p should hold globally

LTL Formulas



LTL - Examples

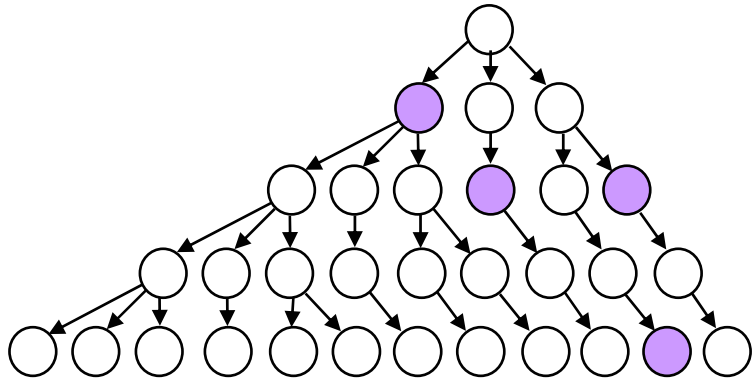
- Safety Properties - G $\neg$(Critical1 $\wedge$ Critical2)
  - Something bad never happens
- Liveness - F (Req1 $\vee$ Req2)
  - Something Good will eventually happen
- Fairness - G (Req1 → F Critical1)
  - If something is requested, it eventually gets scheduled
- Strong Fairness - GF (Req1 $\wedge$ $\neg$ Critical2) → GF Critical1
  - If something is repeatedly requested, it repeatedly gets scheduled

Computation Tree Logic (CTL)

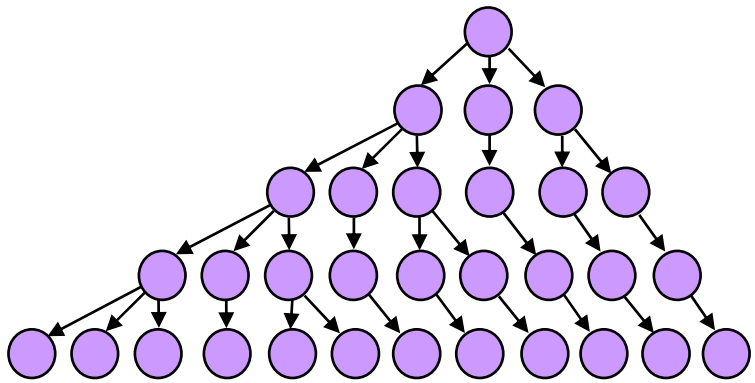
- A form of Branching Time Temporal Logic
- Introduced by Clarke and Emerson
- Temporal Operators as in LTL
  - X p
  - F p
  - G p
- In addition, path operators
  - A – Over all paths
  - E – There exists a path

Computation Tree Logic

AF p - On **all paths**, the property holds in the **future**

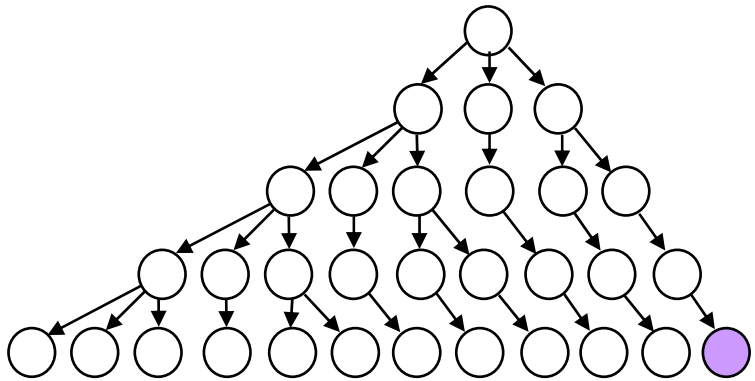


AG p - On **all paths**, the property holds **globally**

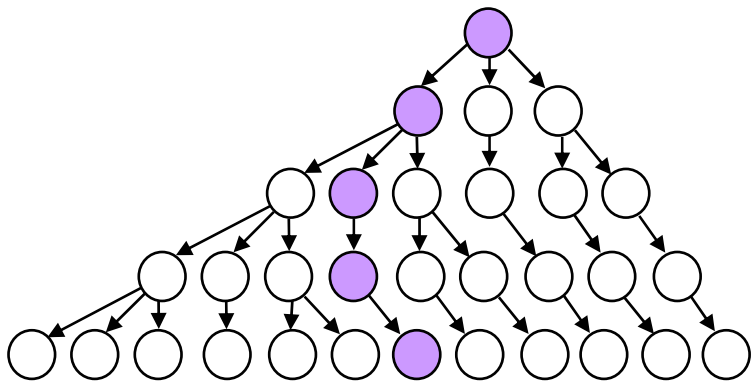


Computation Tree Logic

EF p - There **exists a path** on which the property holds in the **future**



EG p - There **exists a path** on which the property **globally** holds



CTL Examples

- Safety Properties
  - AG ¬(Critical1 ∧ Critical2)
- Fairness
  - AG (state1 → EF (¬ state1))
  - Self deadlock check for state1
  - AGEF (resetState)
  - A powerful and useful resetability test
- Strong fairness
  - Cannot be specified in CTL

Comparison between LTL and CTL

- Expressibility
  - Neither is superior to the other
  - LTL - Strong fairness CTL - Deadlock
  - CTL* - Superset of both LTL and CTL
- Verification Complexity
  - LTL - Exponential in size of formula
  - CTL – Linear in size of formula
- Compositional Verification
  - LTL – Yes CTL – No
- Amenable to Simulation
  - LTL – Yes CTL – No
- Preference in the industry – LTL flavor

Comparison between LTL and CTL

	LTL	CTL
Expressibility (CTL* superset of LTL and CTL)	Strong Fairness	Deadlock
Verification Complexity (in size of formula)	Exponential	Linear
Compositional Verification	Yes	No
Amenable to Simulation	Yes	No

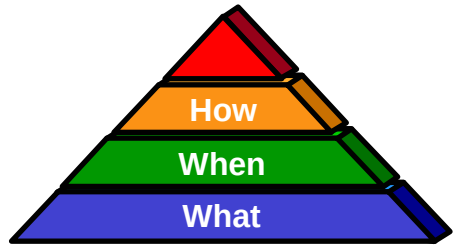
- Preference in the industry – LTL flavor

Industrial Languages

- Limitations of LTL and CTL
  - Mainly academic languages for experts
  - Unreadable by average user
  - Open to mis-interpretation
  - Sometimes un-intuitive to express properties
- Industrial Languages
  - Property Specification Languages (PSL)
  - System Verilog Assertions (SVA)
  - Extending LTL with notion of sequences
    - PSL also has a CTL flavor - Not used very often

Overview of PSL

- Boolean Expressions
  - HDL expressions Verilog xor VHDL
  - PSL/Sugar functions: rose(), fell(), prev(), ...

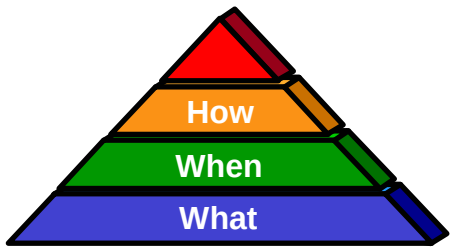


- Temporal Operators
  - always, never, next, until, before, eventually, abort, ...
  - @ -> |-> |=> ; { } [*] [=] [->] && & | :
- Verification Directives
  - assert, assume, restrict, cover, ...
- Modeling Constructs
  - HDL statements used to model the environment

Invariants

Something that should never happen!

For example: An underflow should never occur



Verification Directive

Temporal Operator

Boolean Expression

assert_no_underflow : assert never (read && empty);

label

How (to apply)

When (expression should be true)

What (to check)

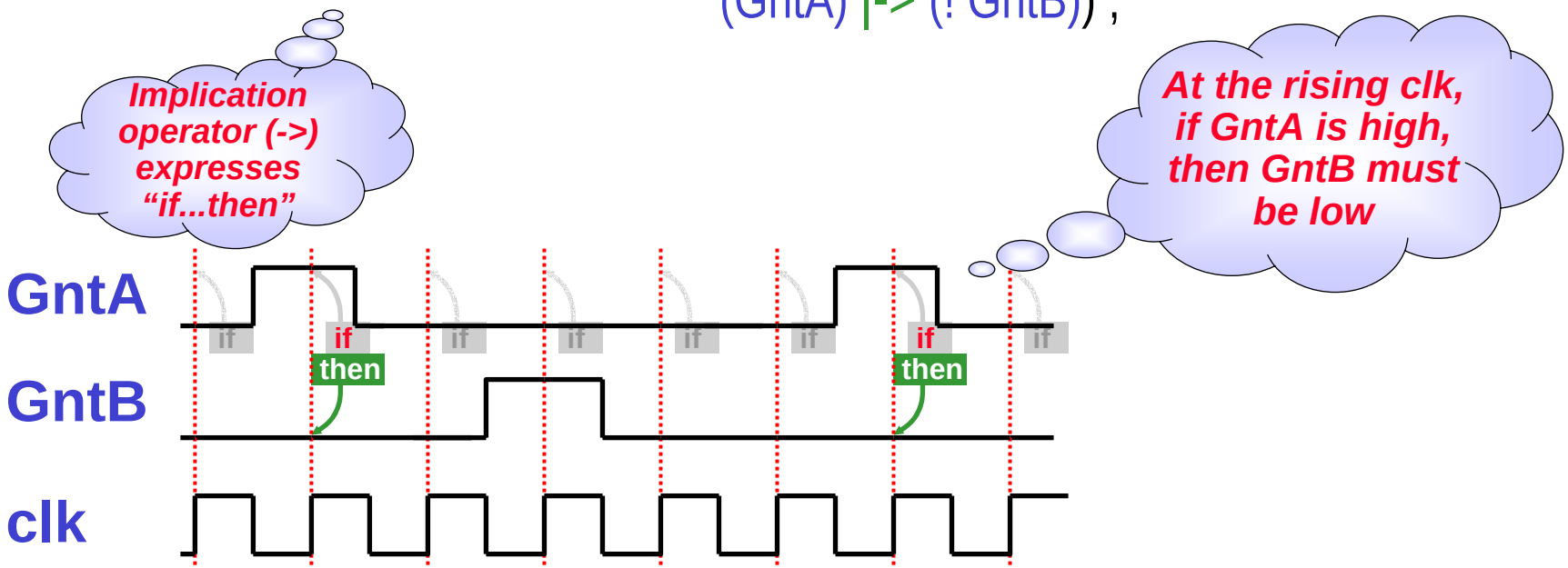
This assertion should hold at every time step

assert_no_underflow : assert never (read AND empty);

Conditional Behavior

Example: If A receives a Grant, then B does not

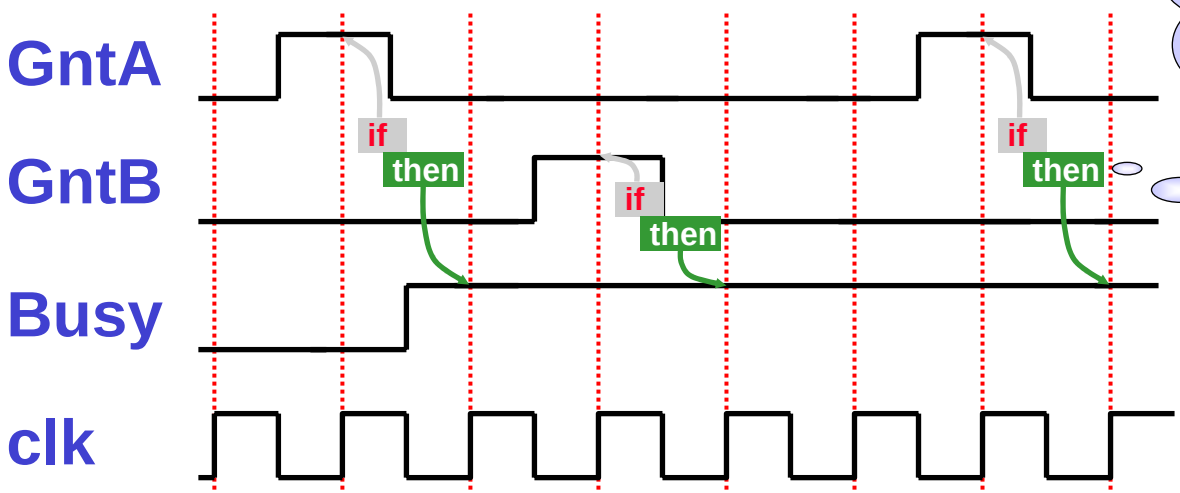
```
assert_if_GntA_no_GntB : assert
always (GntA -> ! GntB) @(posedge clk) ;
assert_if_GntA_no_GntB : assert property (@(posedge clk)
                                (GntA) |-> (! GntB)) ;
```



Multi-Cycle Conditional Behavior

Example: A Grant is always followed by Busy

```
assert_busy_after_Gnt : assert
always (GntA OR GntB) -> next Busy @(rising_edge (clk)) ;
assert_busy_after_Gnt : assert property (@(posedge clk)
                        (GntA || GntB) |=> (Busy)) ;
```



Implication (->) and 'next' together express multi-cycle conditional behavior

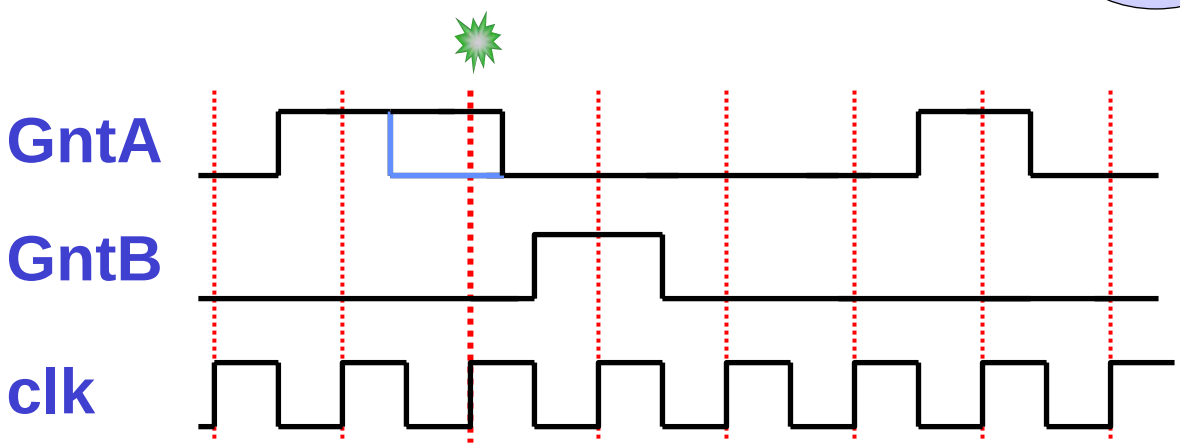
Now there is a one-cycle delay between 'if' and 'then'

Multi-Cycle Behavior Using Sequences

Example: A never receives a Grant in two successive cycles

```
assert_no_2_GntA : assert
                   never {GntA ; GntA} @(posedge clk) ;
assert_no_2_GntA : assert property (@(posedge clk)
                   (GntA ##1 GntA) |-> (0)) ;
```

A sequence is a shorthand for a series of 'next's

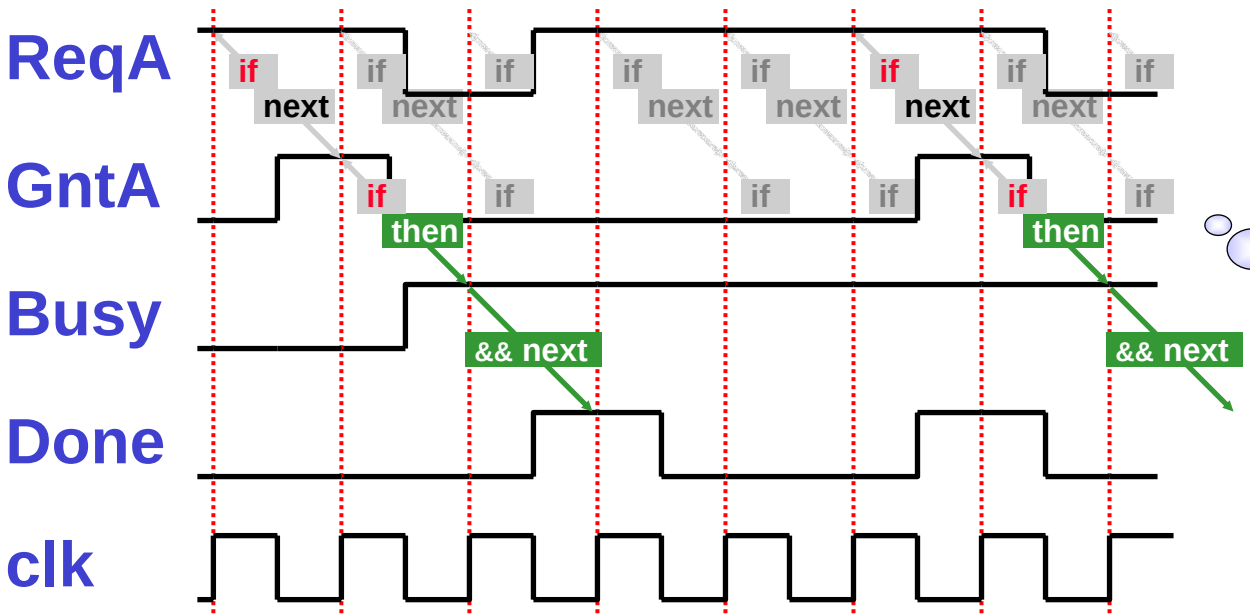


Compound Assertions

Example: If Request is followed by Grant, then next is Busy, and next is Done

```
assert_GntA_ReqA_next_Busy_Done : assert always
(ReqA -> next (GntA -> next (Busy && next Done))) @(posedge clk) ;
```

```
assert_GntA_ReqA_next_Busy_Done : assert always
{ReqA; GntA} |=> {Busy; Done} @(posedge clk) ;
```



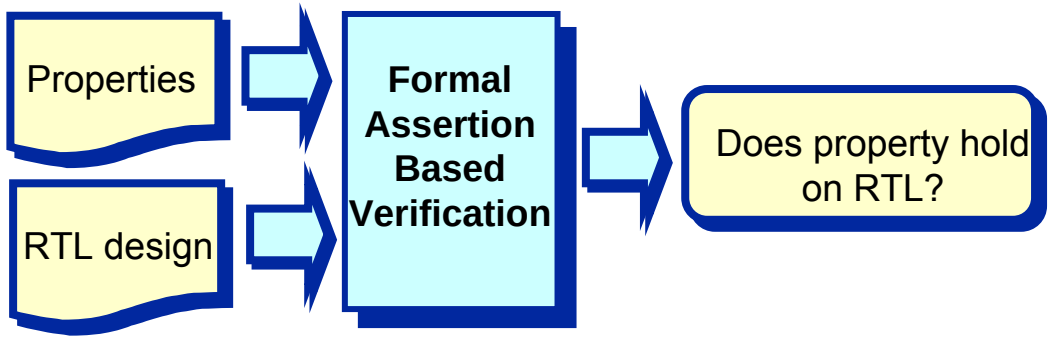
The two assertions are equivalent.

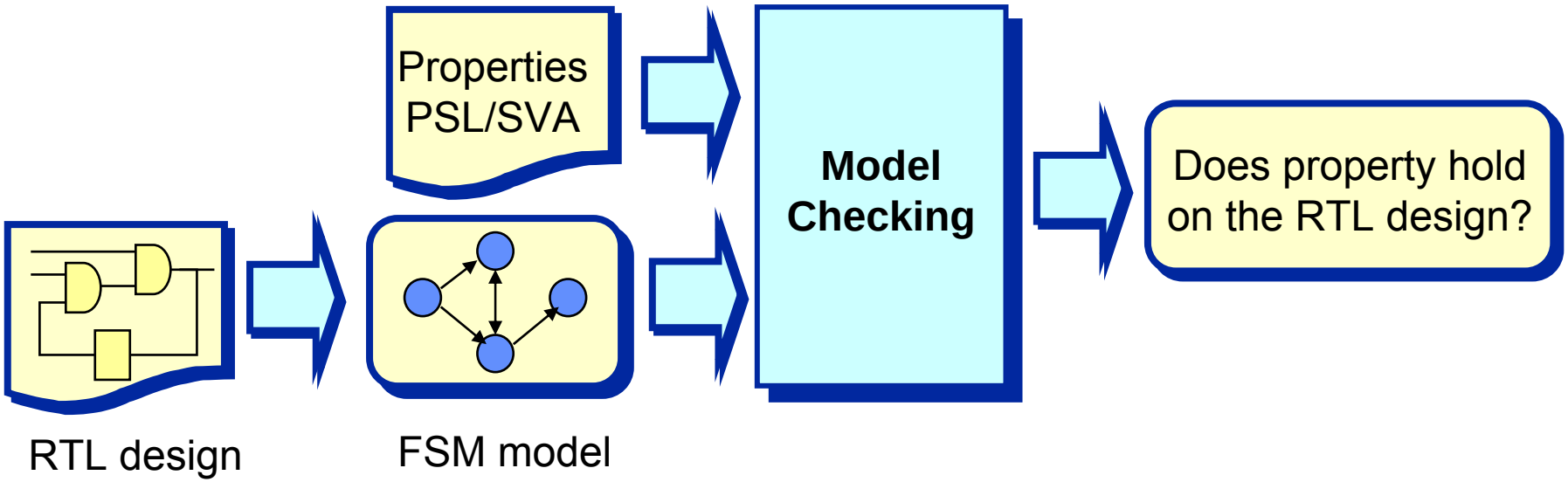
Evaluation starts again in each cycle, overlapping previous evaluations

Agenda

- Introduction to Formal Assertion Based Verification
  - What is the problem statement?
  - Temporal Logics
    - LTL and CTL
    - PSL and SVA
  - CTL Model Checking
  - LTL Model Checking
  - Symbolic Model Checking
    - BDD and SAT based techniques
  - Abstractions
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

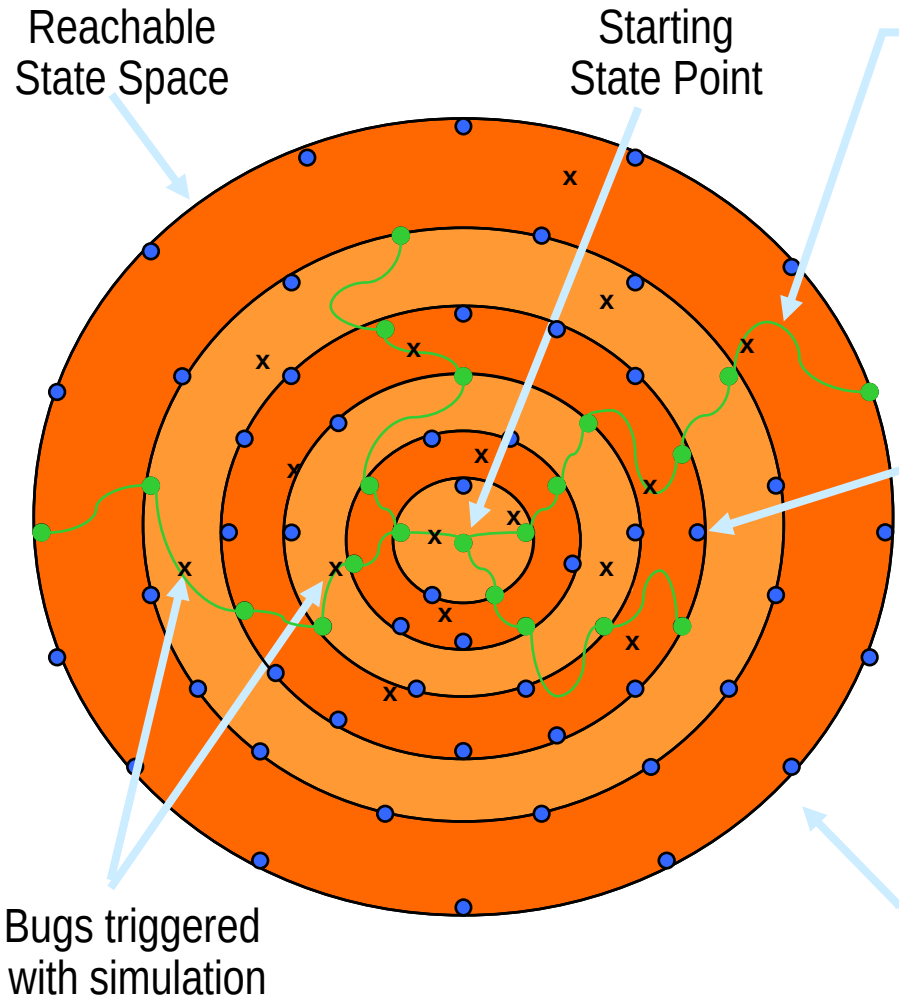
Model Checking





- Model Checking
  - Does the model satisfy the property?
  - Does not require any test vectors
  - Does exhaustive verification

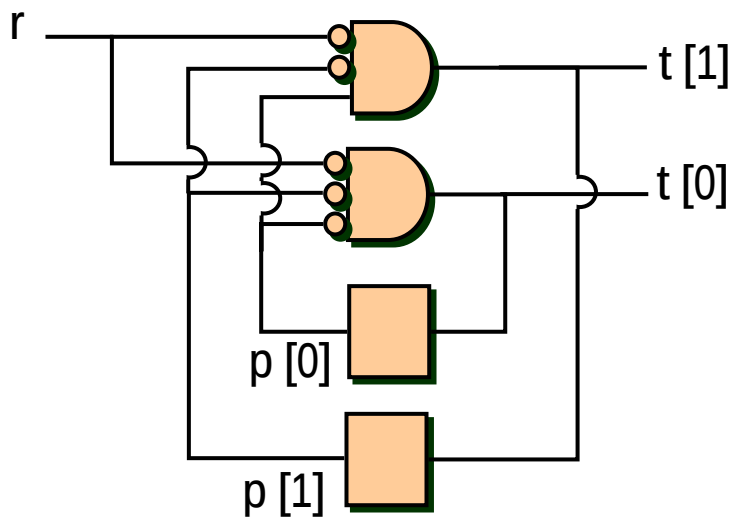
How does Model Checking work?



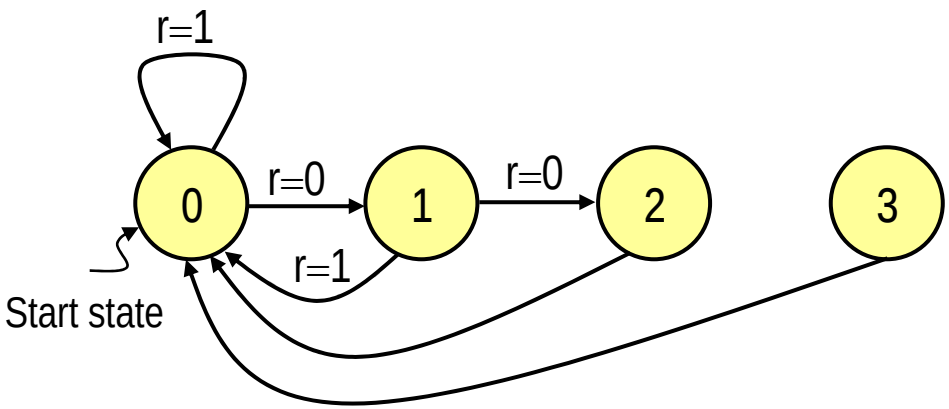
- Simulation
  - Requires test vectors
  - Follows specific path
- Model Checking
  - Requires no test vectors
  - Works on entire state space
  - Breadth first search
- Exhaustive
  - Uncovers all possible bugs

Example

- RTL Design
  - Modulo-3 counter



- FSM Model

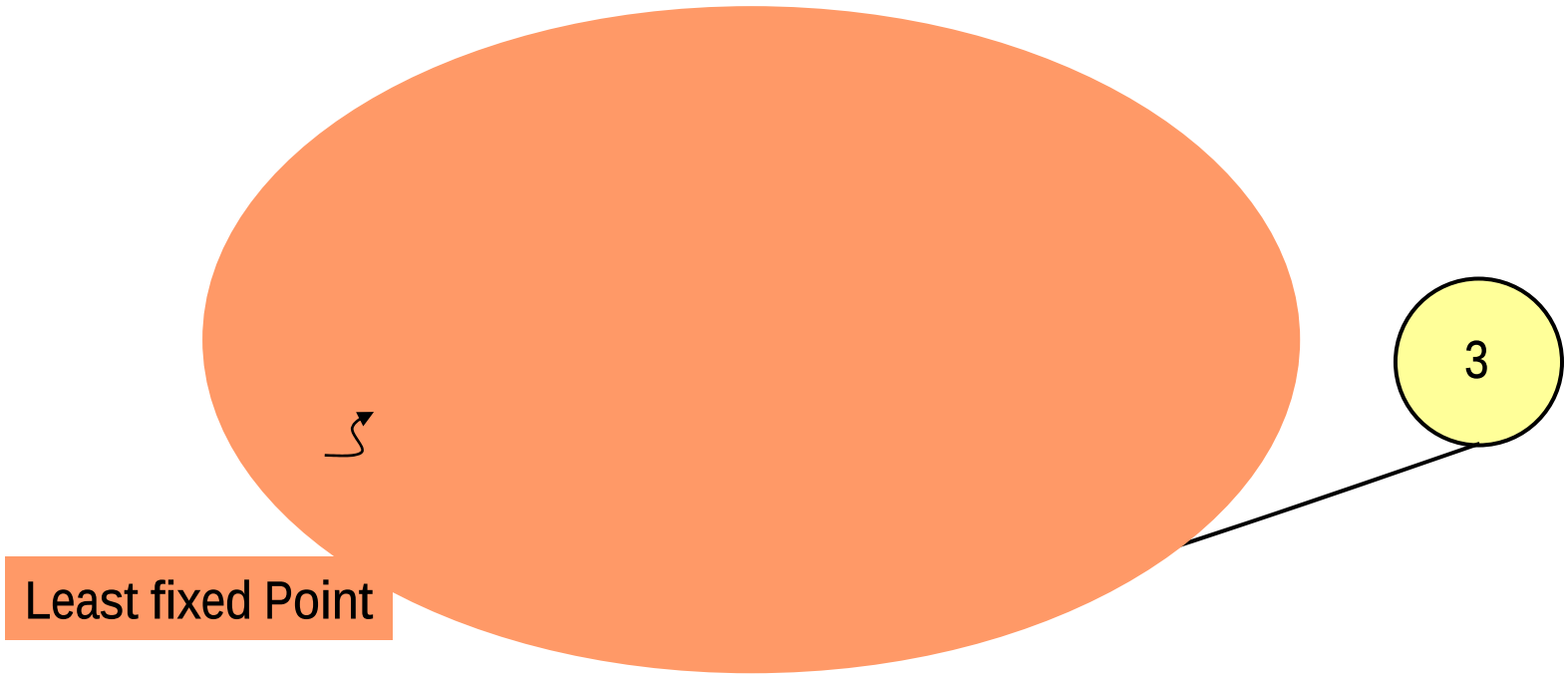


- CTL Property
  - EF (p = 2)

- Starting from start state, can the counter eventually count upto 2

CTL Model Checking

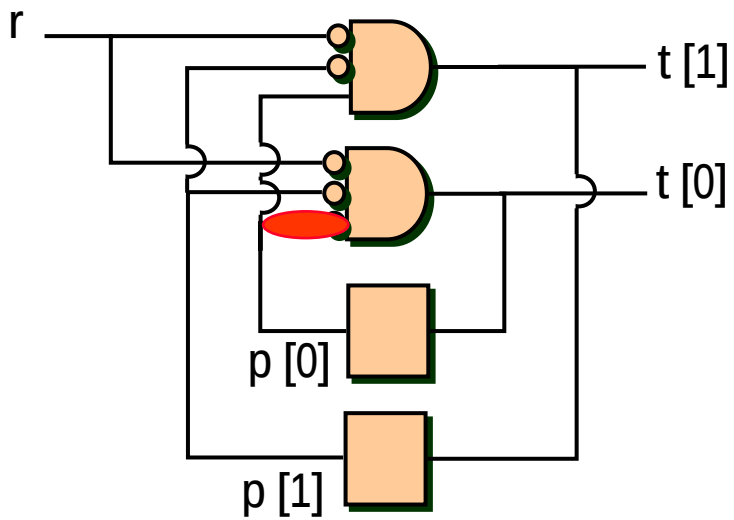
- Evaluate the CTL property on the model
  - Evaluate EF (p = 2) on the FSM



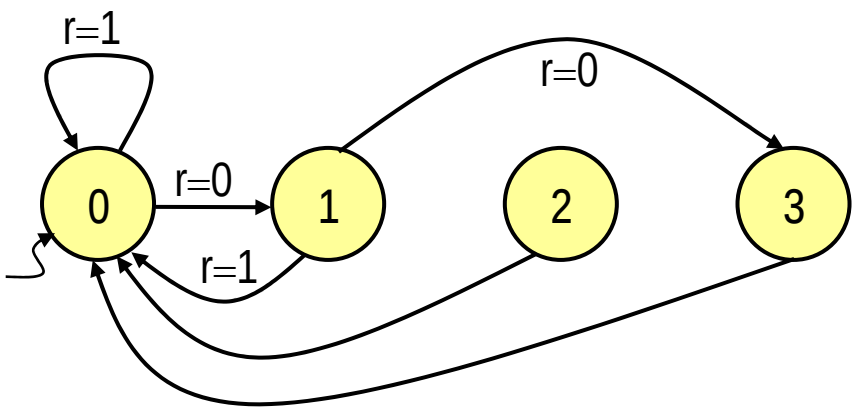
- Start state is a part of the Least Fixed Point
  - Indicates there is a path from start state to (p = 2)
  - EF (p = 2) HOLDS on the FSM model

Lets introduce a bug

- RTL Design
  - Buggy Modulo-3 counter



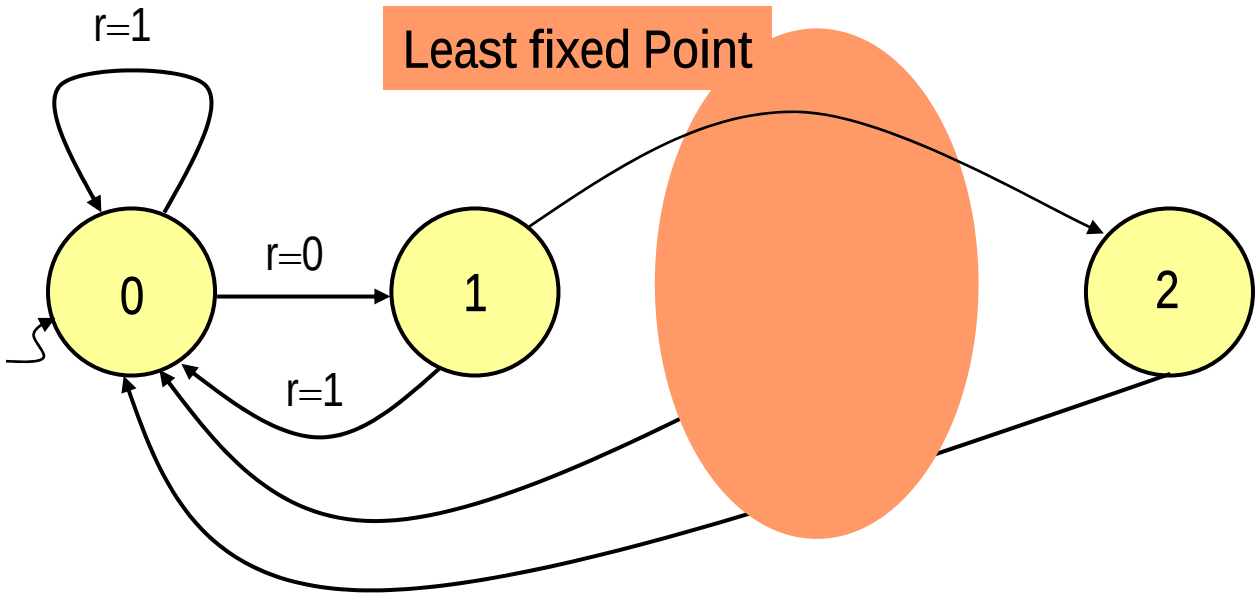
- FSM Model



- CTL Property
  - EF (p = 2)

CTL Model Checking on the buggy design

- Evaluate the CTL property on the buggy model
  - Evaluate EF (p = 2) on the buggy FSM

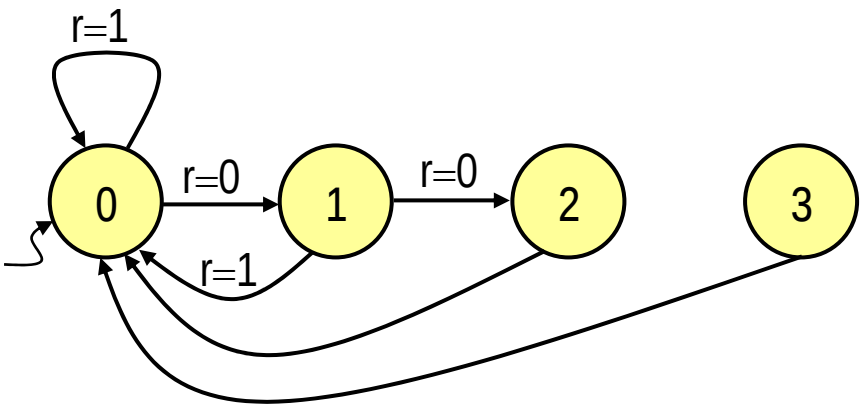


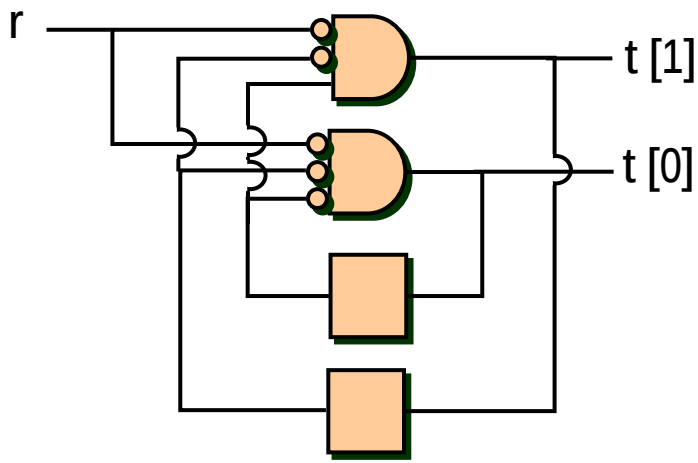
- Start state is NOT part of the Least Fixed Point
  - Indicates there is NO path from start state to (p = 2)
  - EF (p = 2) DOES NOT HOLD on the buggy FSM model

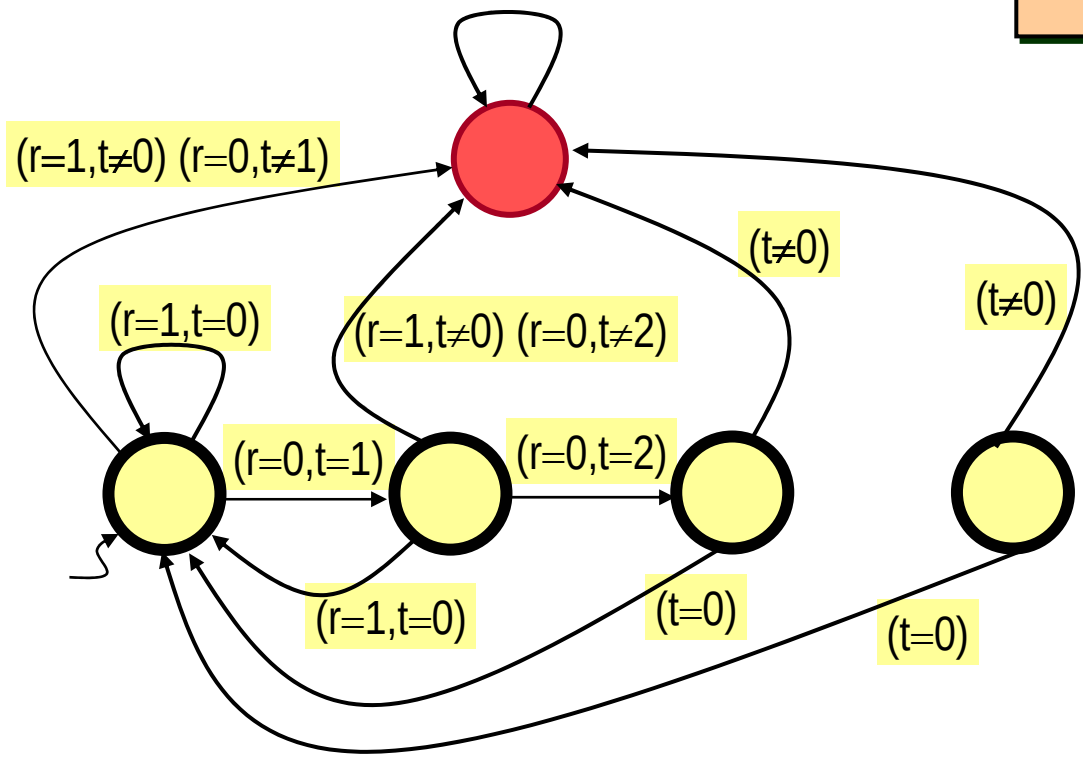
LTL Model Checking

- Requires a new mathematical formulation
  - Omega automata (ω-automata)
- ω-automata
  - Automata over infinite words
  - Infinitely running automata
  - Single framework for expressing design and property
  - Various different forms:
    - Buchi, Streett, Rabin, L-automata and L-process
- Buchi Automata
  - Accepting state is visited infinitely often

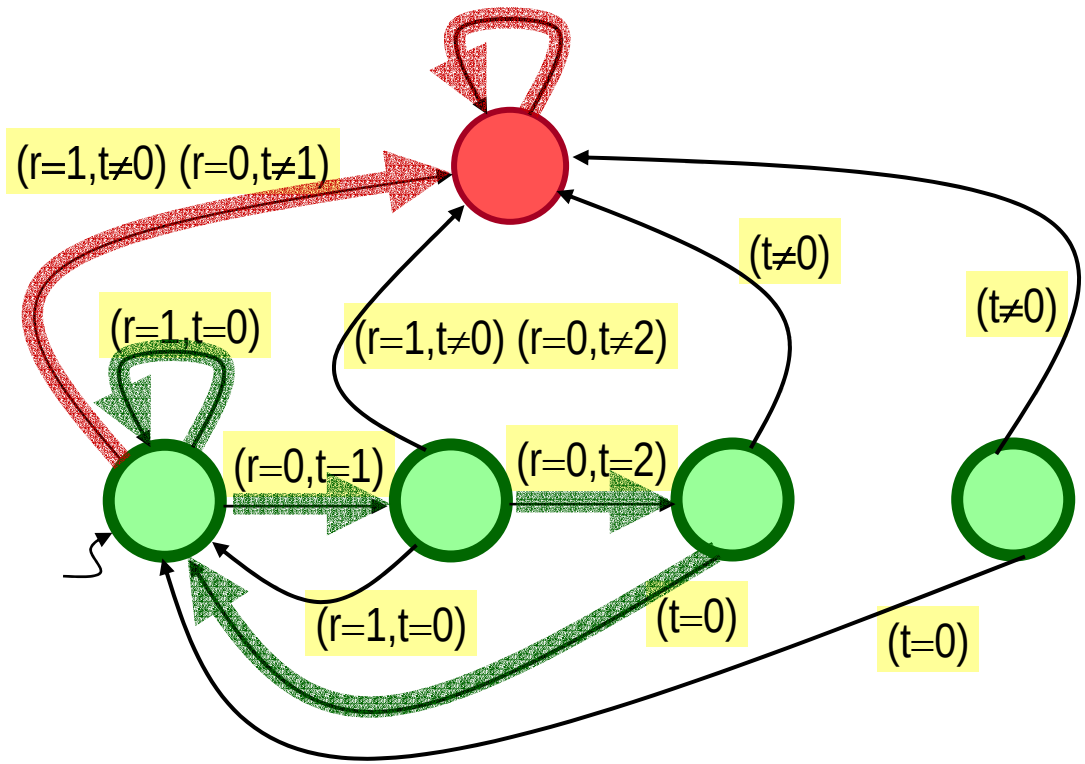
ω-automata for design







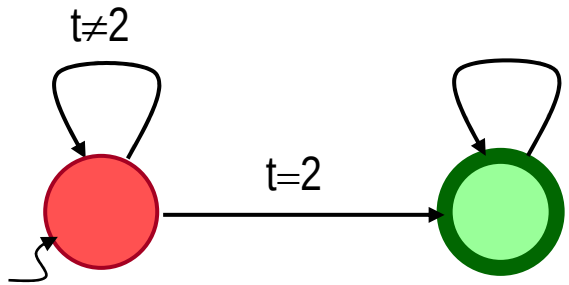
ω-automata for design



- Valid Behavior – (r=1,t=0) (r=0,t=1) (r=0,t=2) (r=0,t=0) ……
- Invalid Behavior – (r=1,t=1) ……
- L(D) – Set of all valid behaviors of the design

ω-automata for property

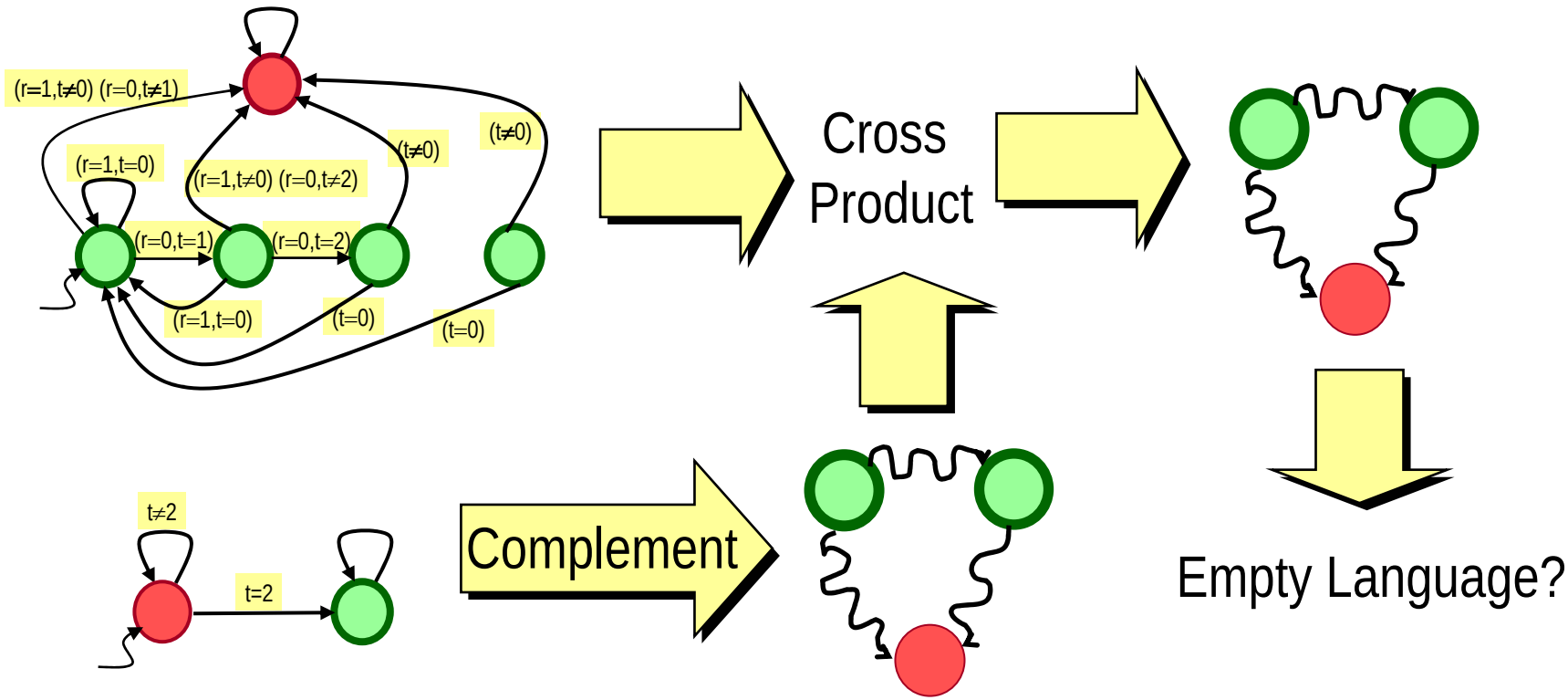
- Every LTL property can be expressed as ω-automata
- F (t = 2)
- Alternatively eventually! (t == 2)



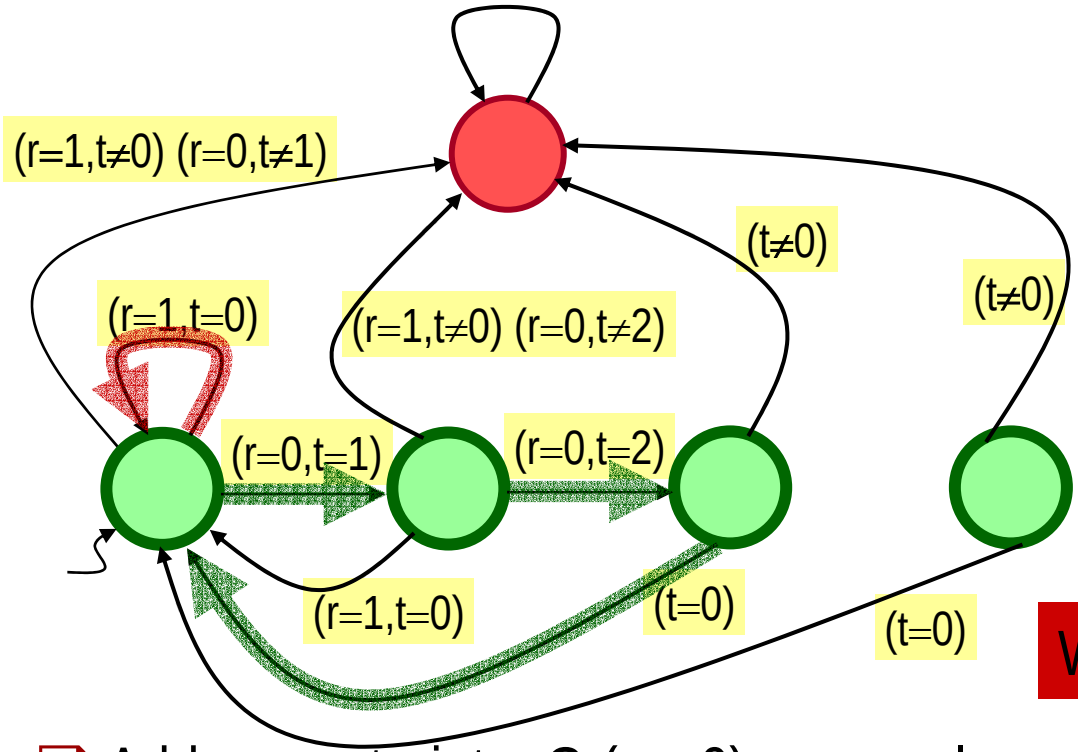
- Valid behavior of the property
  - $(t \neq 2)^{*}\ (t = 2)\ (\text{true})^{\omega}$
  - ..... (t = 2) .....
- L(P) – Set of all valid behaviors of the property

LTL Model Checking

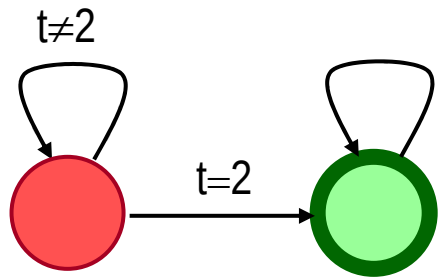
- LTL Model Checking is Language Containment
- Design behaviors is a subset of property behaviors
- $L(D) \subseteq L(P)$ or $L(D) \cap L(\overline{P}) = \varnothing$



LTL Model Checking on our Example



F (t = 2)



Will obtain a CEX – $(r=1, t=0)^{\omega}$

- Add a constraint – G (r = 0) or always(r == 0)
- Constraints are used to model environment and limit behaviors
- L(C) – Set of all behaviors of the constraint
- Check $L(C) \cap L(D) \cap \overline{L(P)} = \varnothing$.

Property will pass

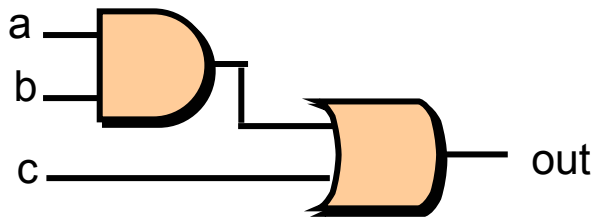
Complexity of Model Checking

- CTL Model Checking
  - Evaluate property over the FSM structure
  - Notion of fixed point computation
  - Complexity is linear in the size of the formula
- LTL Model Checking
  - Notion of ω-automata
  - Translate design and property to ω-automata
  - Then LTL model check is simply a graph problem
  - Complexity is exponential in the size of the formula
- What problem remains? The State Explosion Problem
  - Complexity is linear in the number of states in design
  - Exponential in number of state bits in the design

Agenda

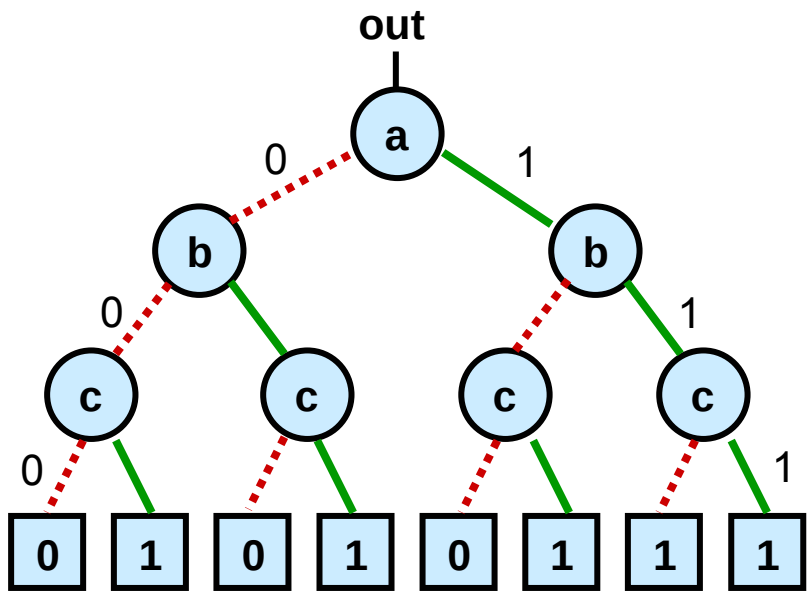
- Introduction to Formal Assertion Based Verification
  - What is the problem statement?
  - Temporal Logics
    - LTL and CTL
    - PSL and SVA
  - Model Checking
    - CTL Model Checking
    - LTL Model Checking
    - BDD and SAT based techniques
  - Abstractions
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Binary Decision Diagrams

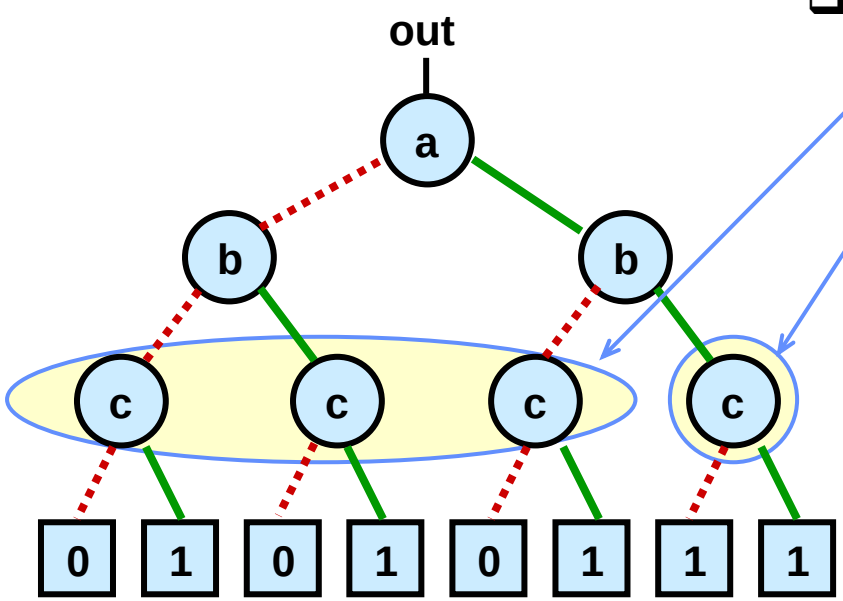


a	b	c	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

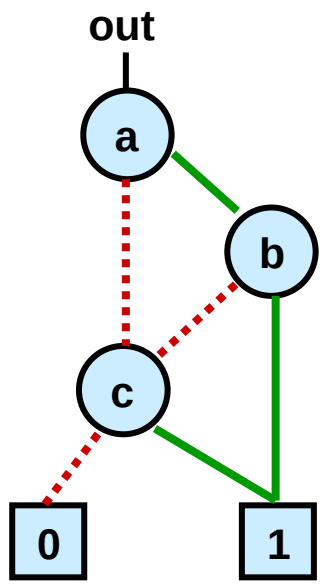
- Ordered Decision Tree



Binary Decision Diagrams

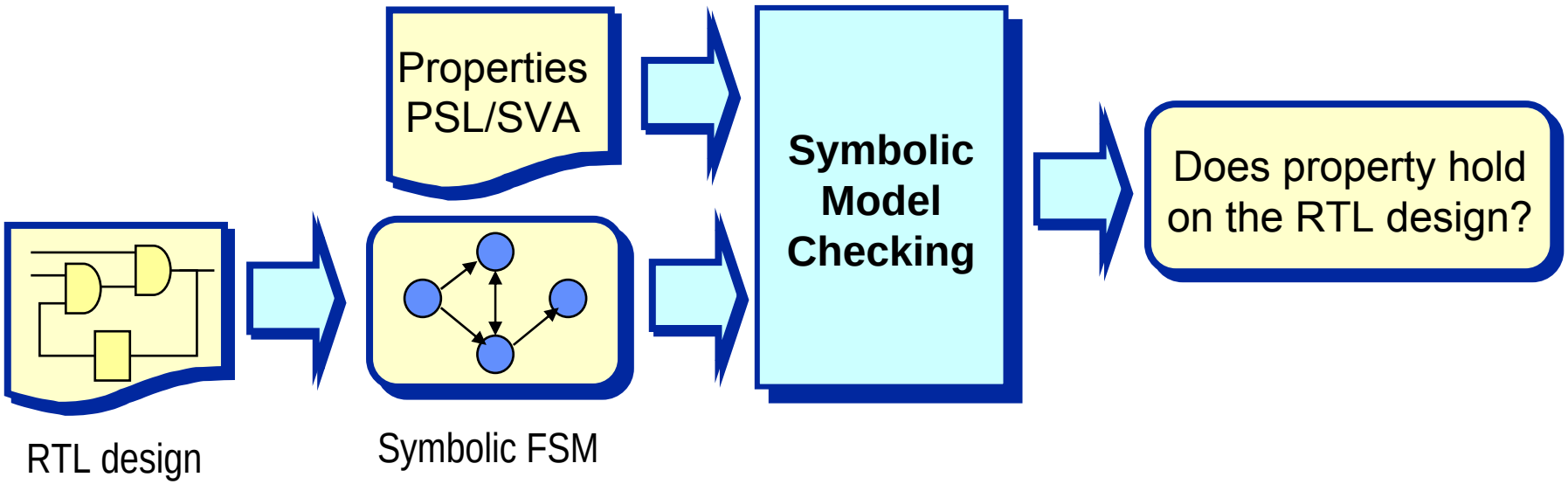


- Reduced Order BDD (ROBDD)
  - Merge isomorphic nodes
  - Remove redundant tests

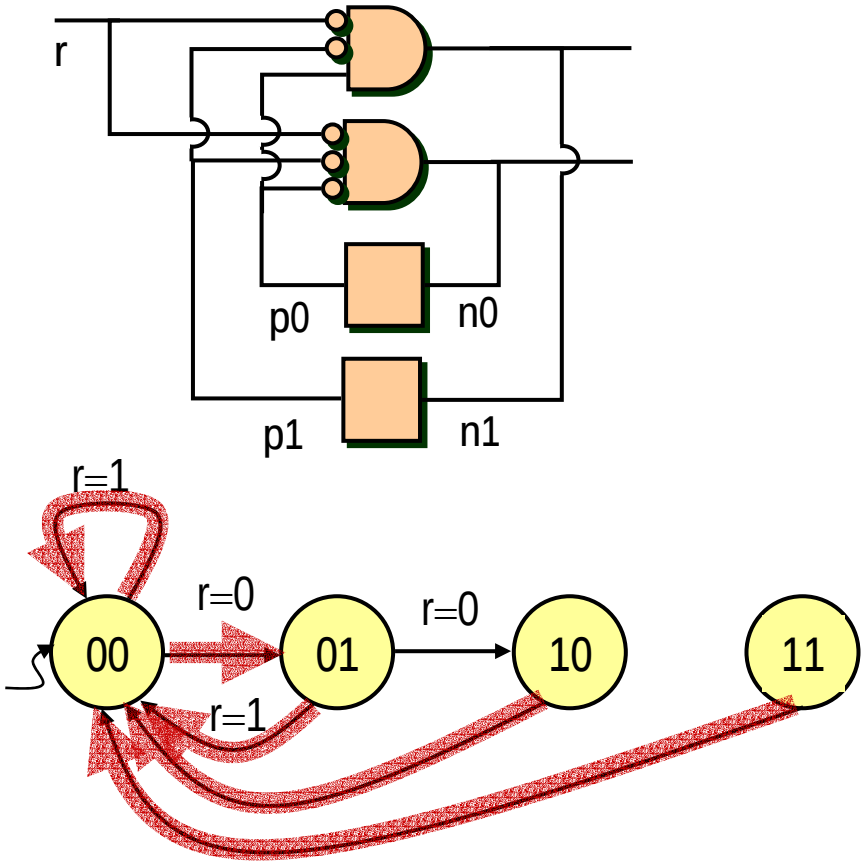


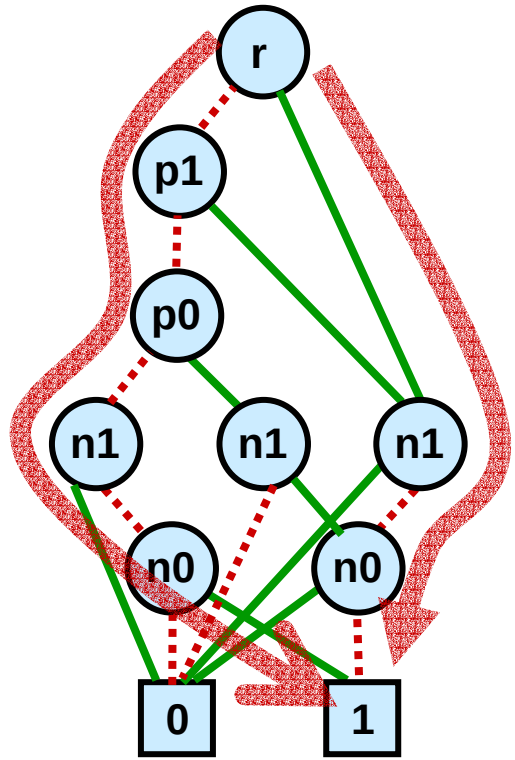
BDD based Symbolic Model Checking

- Build a symbolic FSM model using BDDs
- Perform fixed point computation on the symbolic FSM
- Determine if property holds or not on the symbolic FSM



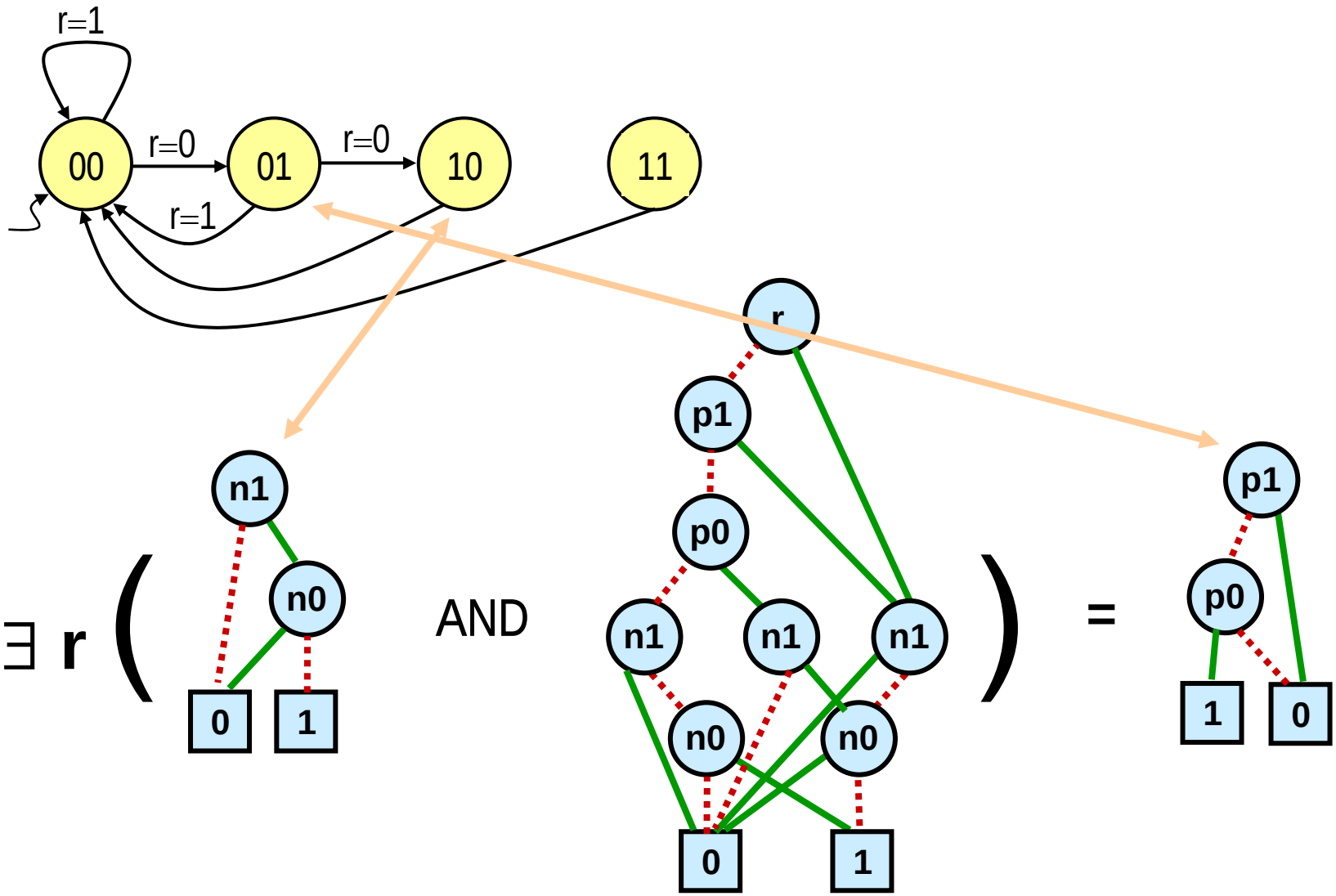
Symbolic Model Checking on our Example





- FSM next state function can be represented symbolically as a BDD
- BDD allows both forward and backward state traversal in the FSM
  - Technically called a transition relation

Symbolic Model Checking on our example



BDD Based Model Checking

- Benefits
  - All operations can be done symbolically
  - Avoids upfront state explosion of explicit FSM
- However, not a magic solution
  - Size of BDD dependent on variable order
  - Example – for an n-bit adder
    - Best order is linear, Worst is exponential
  - Finding the optimal order is a very hard problem
    - NP-complete problem
- In practice works well if
  - Transition relation compactly represented with BDDs
  - Reachability is fast - BDDs do not get stuck in reordering

SAT (Satisfiability) Problem

- Given Boolean function $f(x_1, x_2, \ldots, x_n)$
- If it is possible to find assignments:
  - $(x_1=a_1), (x_2=a_2), \ldots (x_n=a_n)$, such that
  - $f(a_1, a_2, \ldots, a_n) = 1$?
- Works on Boolean formula in CNF
  - Conjunctive Normal Form or product of sums form

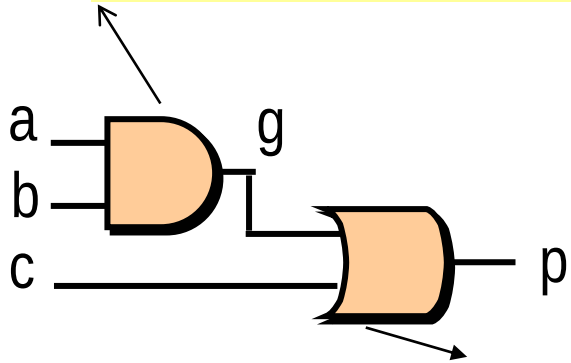
$$(\neg a \vee b) \wedge (\neg b \vee c \vee d)$$

- This is an NP-Complete problem

Transforming design into a SAT problem

- Can the output of a design take value 1?

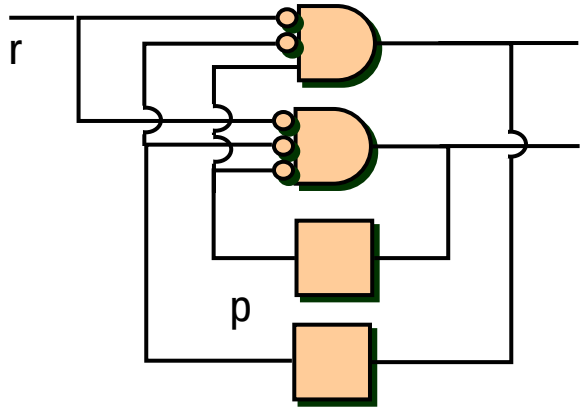
$CNF_1 = (a \vee \neg g) \wedge (b \vee \neg g) \wedge (\neg a \vee \neg b \vee g)$



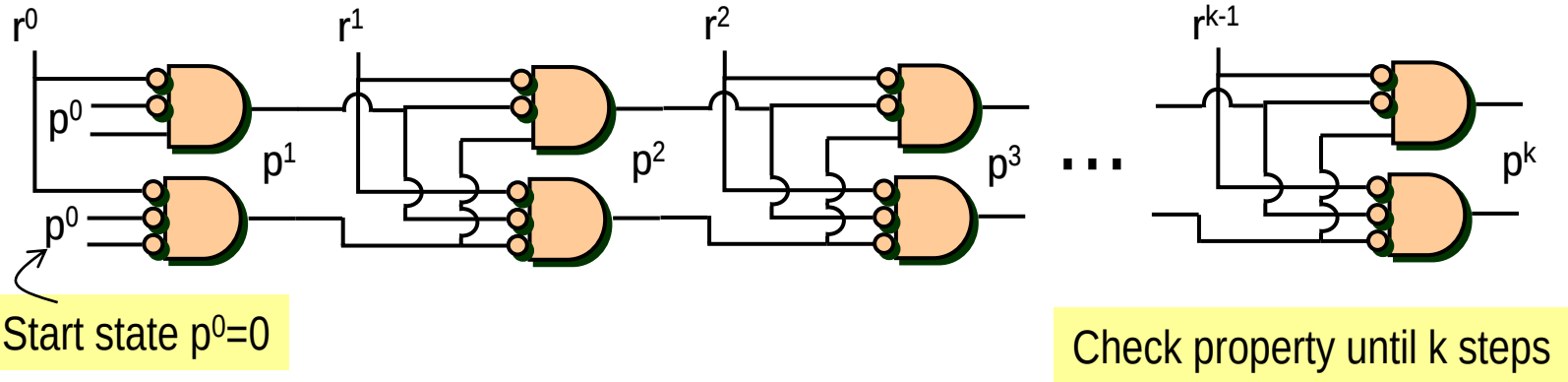
$CNF_2 = (\neg g \vee p) \wedge (\neg c \vee p) \wedge (g \vee c \vee \neg p)$

- p can take value 1 if (CNF1 $\wedge$ CNF2 $\wedge$ p) is satisfiable

Bounded Model Checking



- Unfold the circuit “k” times



- Take the unrolled circuit and convert into CNF
- Use SAT solver to check if safety property holds for k steps
- If no, then SAT solver will give a CEX of $\leq$ k steps
- Bounded Model Checking can only give failures for safety

Symbolic Model Checking Summary

- BDD based Model Checking
  - Works well for “quick” passes and failures
  - Passes and failures could be quite deep
- SAT based Bounded Model Checking
  - Good for shallow failures
  - For deep failures unrolling becomes a bottleneck
- Other techniques
  - Combining BDD, SAT and ATPG based techniques
  - Distribution
  - Abstractions

Agenda

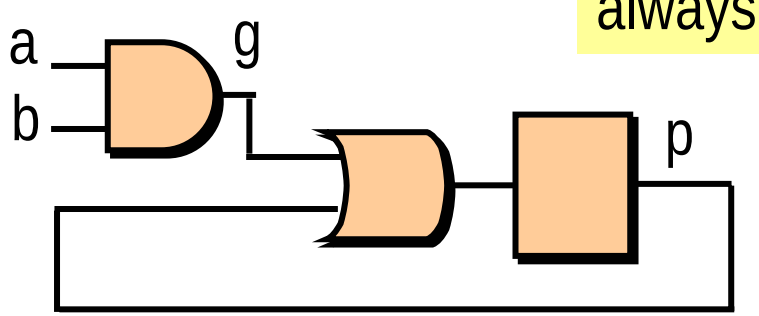
- Introduction to Formal Assertion Based Verification
  - What is the problem statement?
  - Temporal Logics
    - LTL and CTL
    - PSL and SVA
  - Model Checking
    - CTL Model Checking
    - LTL Model Checking
  - Symbolic Model Checking
    - BDD and SAT based techniques

- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Abstraction

- Throw away information that is not required for proof
- Abstraction is key to scaling capacity of model checking
- One of most important abstraction
  - “Localization" abstraction
  - Throws away information not relevant to the given property

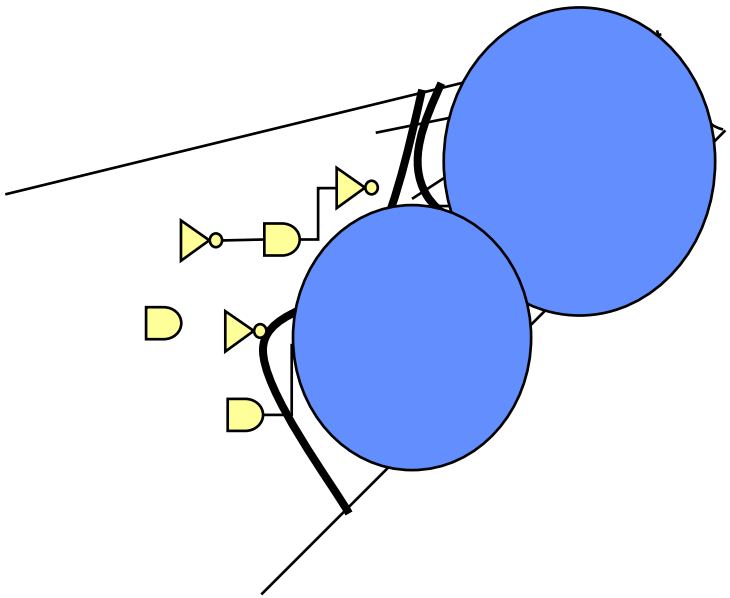
Make g a pseudo-input



G (p → Xp)

always ((p==1) → next (p==1))

Abstraction and Refinement



- Choose Initial Abstraction
- Model Check Abstracted Design
- Property Pass – It is indeed a Pass on full design!
- Justify Counter-example
- Justified? – Bug Found!
- Refine the model
  - Many ways of refinement
- Model Check Refined Design

Other Advanced Abstraction Techniques

- Proof based Abstraction-Refinement
  - Combines SAT based BMC and BDD based model checking
  - Uses SAT based BMC to find a suitable abstraction

- Interpolation
  - SAT based Unbounded Model Checking
  - Uses over-approximate reachability analysis

Industrial Formal ABV Tools

- Various Commercial Tools
  - Cadence – Incisive Formal Verifier
  - Synopsys – Magellan
  - Mentor – 0-in
  - Jasper – JasperGold
  - OneSpin, Real-Intent, Averant, Axiom
- Combination of Symbolic Model Checking and Abstractions
- Can handle local properties
  - Typically can scale to around 10K state bits
- Global properties are still difficult
  - Need advanced abstractions and compositional techniques

Agenda

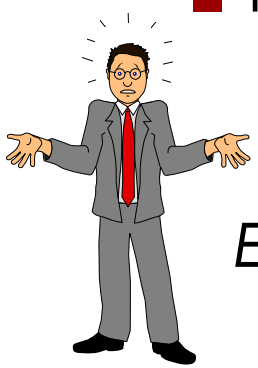
- Introduction to Formal Verification
  - Methodology
    - Planning for Formal Verification
    - Property Coding Guidelines
    - Formal Verification Flow
  - Case Studies
    - Protocol Compliance : Bridge Validation
    - Arbitration
    - Interrupt Sorter
    - Memory Controller
    - SoC Connectivity
    - SoC Performance
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Need for Efficient Methodology

- State explosion problem (insufficient resources) very real !
- Abstraction is the key
  - Automatic
  - Manual
- No golden abstraction(s) to solve state explosion problem
- Analysis is important (apriori and dynamic)
  - For handling formal proof complexity
  - For predictability of FV usage and results

Reality Bites!

- 80% of medium / large sized modules experience state explosion
  - For one or more assertions
  - No golden abstraction(s) apply across all designs
- 70% of initial counter-examples are spurious
  - More so because of incorrect constraints, rather than incorrect assertions
  - Decomposition in practical scenarios lacks formalism
- 50-70% of time is consumed in modeling / setup / coding
  - Rest is tool run time



Experience from application of FV on large set of Industrial designs

Need for Verification Planning

- Verification Objective: Close verification within a predicted schedule
  - Planning
    - Complexity Estimation, Verification Plan, Partitioning and Abstractions
  - Execution
    - Analyzing indeterminate results
    - Adding / Removing / Optimizing Constraints
    - Identification and Closure of False Failures
    - Iterations could be large – due to adhoc constraint updates
  - Closure – Signoff
    - Coverage analysis, Verification report
- All the above are inter-related
  - Closure has no meaning if results are indeterminate
  - Indeterminate results point to lack of proper abstraction
  - Lack of proper abstraction results from improper planning

Formal Verification Decisions

- Abstraction decision
  - Identify sweet spots (control versus data path)
  - What parts of the design need to be abstracted out?
- Modeling decision
  - Choosing the correct way of modeling the property
  - Complex SERE's versus simple properties
- Property decision
  - Which language ? Which AIP's ? Glue logic ?
- Tool specific Decision
  - Chose the correct FV engine
  - Chose correct tool (abstraction) options

Formal Verification Planning

- Predict the problems in advance
  - Based on design knowledge, tool, experience, etc
- Predict
  - Approximate design complexity (design analysis)
  - Approximate proof complexity (related to constraints and logic cone of assertions)
- Decide
  - Whether to partition or not ? Where to partition ?
  - Abstraction mechanism associated with the partitioning
  - How to code assertions / constraints properly?
  - What engines / methods to use for what scenarios?

Planning Step-I: Identify the sweet spots

- Sweet spots : Control logic verification
  - Bus Bridges , Arbitration Logic
  - Controllers (FSM logic, Memory Cntlr, Interrupt Cntlr)
  - Control dominated Logic (Stall, Pipeline …)
- Not so sweet spots
  - Complex Serial Protocols
  - Modules with large FIFO's
  - Protocol interfaces with huge latency parameters
- Negative candidates : Datapath intensive blocks
  - Data Transformation Blocks (Filters …)
  - Blocks with complex arithmetic units (Multipliers, Adders)

Planning Step-II: Analyze the module

- Identify the functionality of the block to be verified
- Identify the functionality of the surrounding blocks.
  - Check for well defined interfaces (standard protocols)
- Prepare detailed micro level bulleted English plan for
  - Assertions to be coded for block functionality
  - Constraints to be coded for functionality of surrounding blocks
  - Disable / Restrain any data-path interfaces (address-data bits)

Planning Step-III: Estimate Complexity

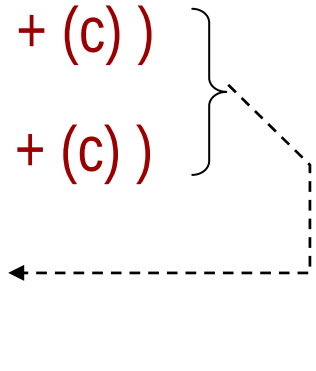
- Estimate Complexity for FV
  - (a) Number of flops in the module (first-crude estimate)
  - (b) Number of flops in logic cone of each assertion (finer estimate)
  - (c) Number of flops inferred because of constraint coding (adds to complexity)

✓ Good Candidate : < M flops ((b) + (c))

? Not so good Candidate : M – N flops ((b) + (c))

☒ Negative candidates : > N flops ((b) + (c))

M ~ 1000 , N ~ 3000



Planning Step-IV: Partition if needed

- Structural Partitioning
  - Partition a design into a set of functionally independent and spatially disjoint components
- Functional Partitioning
  - Partition the design functionality itself into mutually exclusive logical partitions

Structural Partitioning

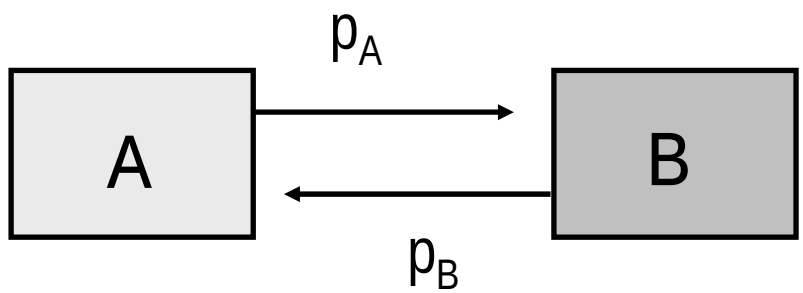
- Golden Rules for partitioning
  - Identify the sub-modules (specs must be clear) which can be verified separately
  - Identify the ‘simple / sparse’ interface at which to cut
  - Partition the design and code constraints at the cut-points
  - Verify each partition in isolation (apply assume-guarantee)
- Coding Constraints at cut-point-interface
  - Localization of the verification process
  - Abstraction is needed to benefit from decomposition
  - Golden Rule: “If surrounding RTL is replaced with equivalent constraints → No gain!”

Functional Partitioning

- Identify mutually exclusive functional partitions
  - Example: Different modes of operation
- Identify interactions between these partitions
  - Independent – do not impose restrictions on each other
    - Example: independent read and write operations
    - Use pin tie-offs to cut off the other partition
  - They follow a sequence
    - Example: a “read-exclusive” operation in OCP protocol should always be followed by a write operation
    - Identify the shared resources which play a role in one or more partitions
    - Use constraints or simulation to skip over the previous partition

Assume–Guarantee Reasoning

- Properties can be used as either assertions or assumptions
- Assume guarantee reasoning
  - DUV is block A
    - assert p_A; assume p_B
  - DUV is block B
    - assert p_B; assume p_A



- Using Protocol AIP's
  - at master interface
    - assume prop_slave* , assert prop_master*
  - at slave interface
    - assume prop_master*, assert prop_slave*

Functional Assume–Guarantee

- Properties can be coded in layers (bottom up fashion)
- Properties in a layer assume validity of properties in below layers
- Application:
  - Prove the correctness of lower level properties
  - Apply them as constraints thereafter to prove higher level properties
  - Example : Arbitration validation:
    Base level property: “The grants are zero-one-hot”
    Higher level property: “check the arbitration scheme”

Agenda

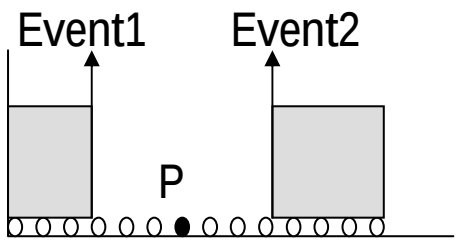
- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
  - Methodology
    - Planning for Formal Verification
    - Formal Verification Flow
  - Case Studies
    - Protocol Compliance : Bridge Validation
    - Arbitration
    - Interrupt Sorter
    - Memory Controller
    - SoC Connectivity
    - SoC Performance
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Property Coding Guidelines

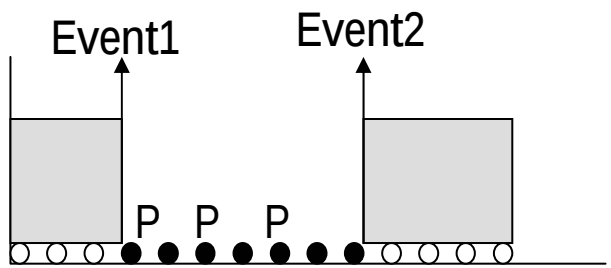
- Golden Rules:
  - Monitor style vs generator style
    - For verifying RTL, do not create another one
  - Atomic vs Expressive Sequential properties
    - Expressiveness adds proof complexity
  - Separability of functional checks
    - Verify unrelated functional aspects in isolation
    - Code separate properties for individual I/O's of module
    - Reduce input state space via pin-constraints
    - Prove partitioned spaces are mutually exclusive
  - Incremental Property specification and verification
    - Build up layers of properties
    - Usually simpler to prove properties separately
  - Avoid integers, counters

Property Coding Guidelines

- Guidelines – try these templates for each property
  - Can it be written as *"condition should/shouldn't always happen"*
  - Can it be written as
    - *prev_state →(implies) current_state*
  - Can it be written *using 'event bounded window' – something happening between two specified events*
  - Can it be written *without using number of clocks (i.e. counter)*



assert_frame



assert_window

Agenda

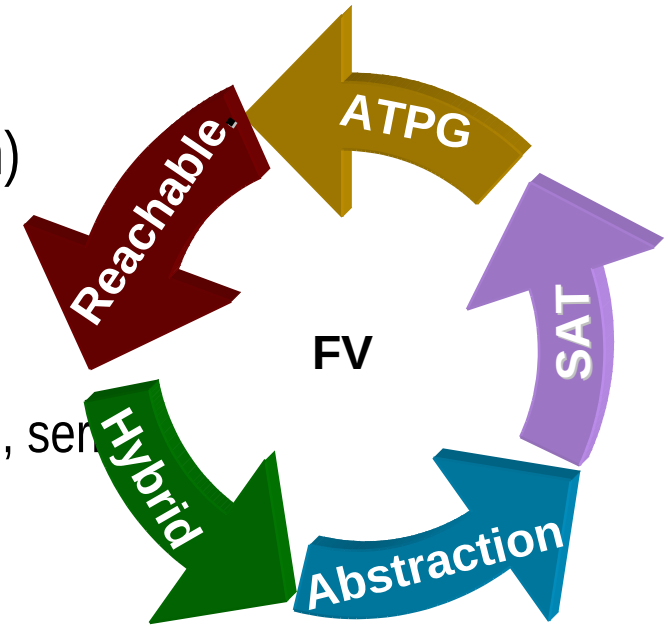
- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
  - Methodology
    - Planning for Formal Verification
    - Property Coding Guidelines
  - Case Studies
    - Protocol Compliance : Bridge Validation
    - Arbitration
    - Interrupt Sorter
    - Memory Controller
    - SoC Connectivity
    - SoC Performance
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Formal Verification Steps

- Verification Setup
  - Sanity checks
  - Cover points
  - Constraint selection (incremental constraint addition)
- Formal Engine Selection
- Managing subsequent verification setup changes
  - Use structural coverage metrics
    - Branch coverage, Expression coverage
    - FSM checks
- Putting it all together

Formal Engine Selection

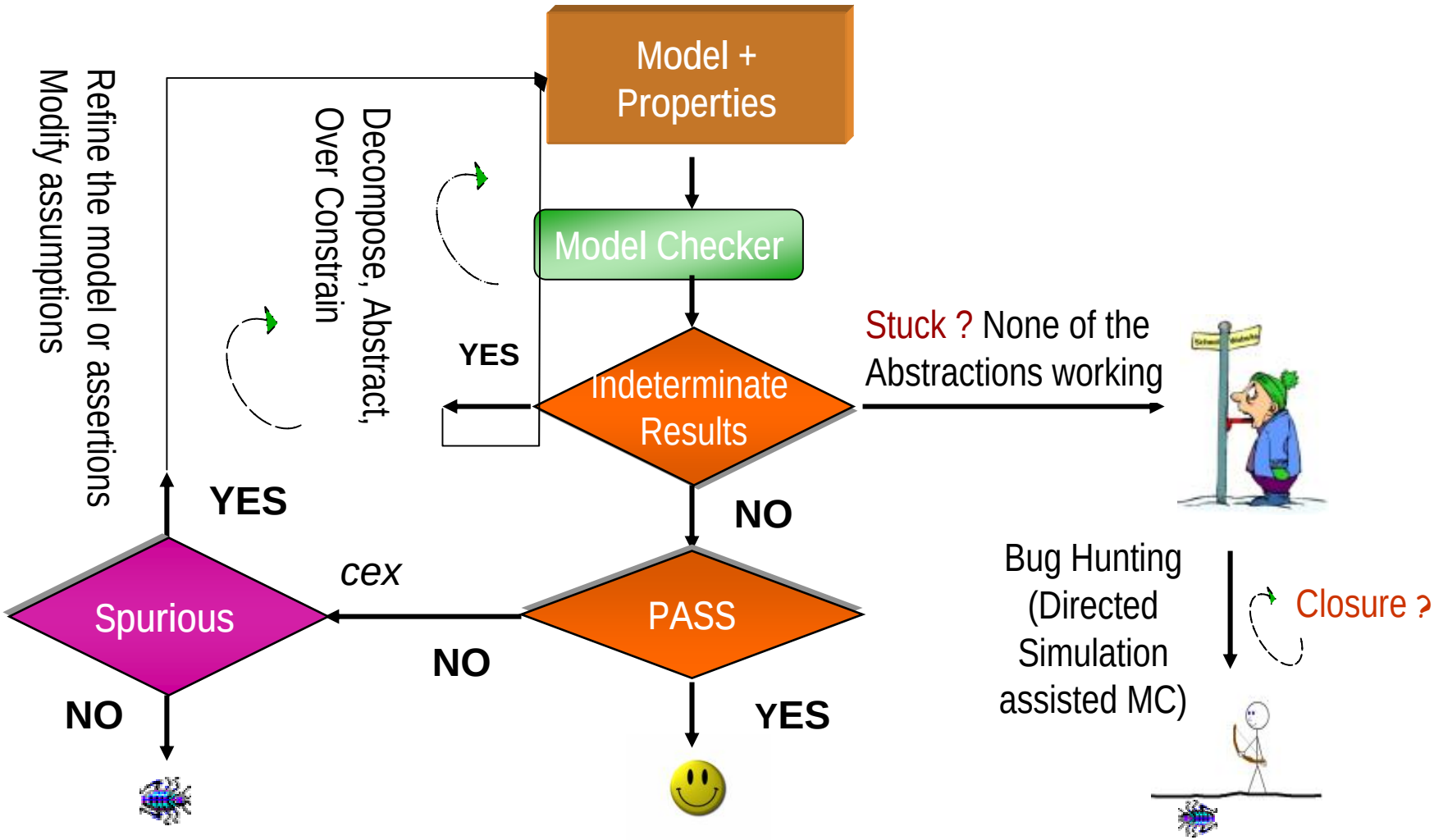
- Engines targeted towards
  - Bug Finding
  - Complete Proof
- Selection depends upon
  - Estimated Sequential Depth (user decision)
  - Design characteristics
    - More breadth than depth owing to concurrent FSM's
    - Large sequential depths owing to counters, ser shift registers



Structural Coverage Metrics

- Metrics point towards ‘unreachable’ parts of the design under the set of applied constraints – validates the ‘sanity’ of the constraints
- Various set of metrics supported by tools
  - Branch coverage (Dead code analysis)
  - FSM checks
  - Expression coverage
- Point to over-constrained, restricted verification environment
- Managing constraint addition / removal
  - For every major constraint change, coverage analysis must be done to filter out constraint related problems
  - Compare with past data

Formal Verification Flow

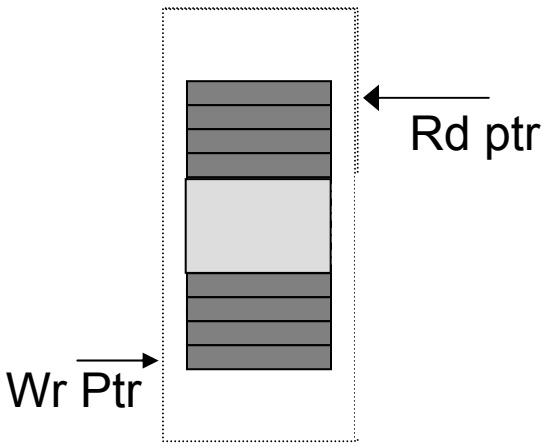


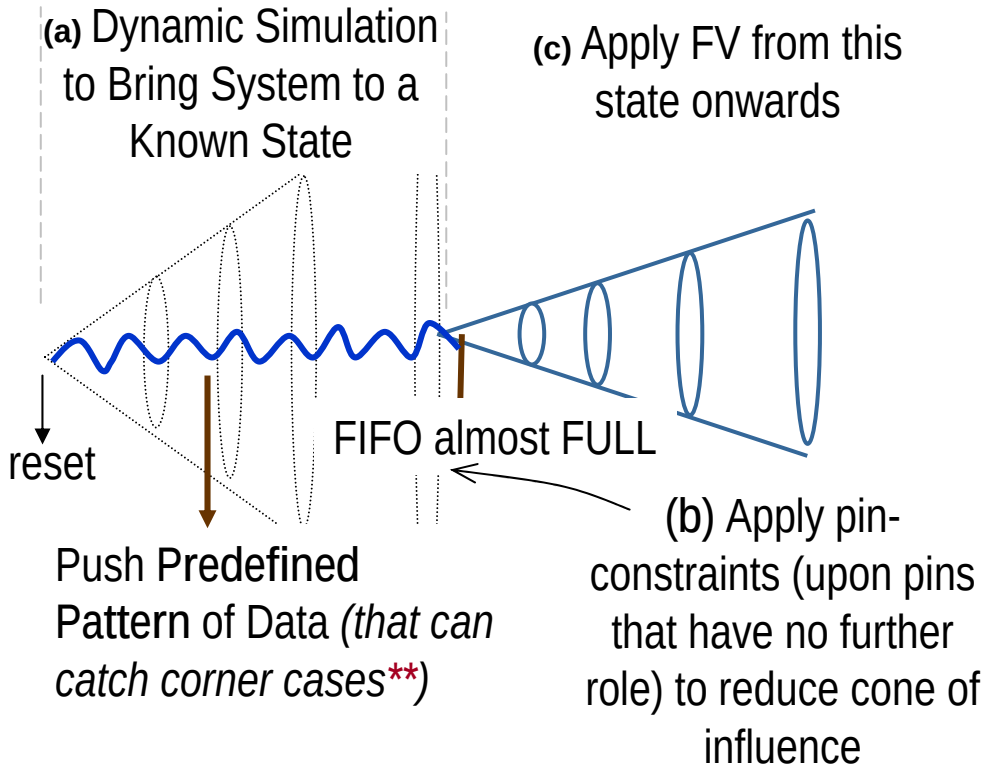
Bug Hunting Mode

- To find silicon bugs, or when the overall proof procedure does not yield results
- Functional Partitioning
  - Break design state space into logical sub-spaces
- Directed simulation
  - Guide the system to known states
- Formal Verification
  - To explore for bugs in the logical partition, separately
  - Usually apply this technique with some over-constraints

Bug Hunting Example: Dealing with FIFO's

- Interesting points to check: Overflow/Underflow
  - If possible, "reduce FIFO depth / width"
  - Else light-weight simulation to "reach near corner points"





Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
  - Methodology
    - Planning for Formal Verification
    - Property Coding Guidelines
    - Formal Verification Flow
  - Protocol Compliance : Bridge Validation
    - Arbitration
    - Interrupt Sorter
    - Memory Controller
    - SoC Connectivity
    - SoC Performance
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Bridge : Verification Targets

- Functional Coverage Points:
  - Protocol Compliance
  - Address Decoding
  - Data Integrity
  - Arbitration
  - Performance / Latency
- Structural Coverage Points
  - Dead Codes
  - FSM state / transition
  - Structural checks (Out of bounds, FIFO full / empty)

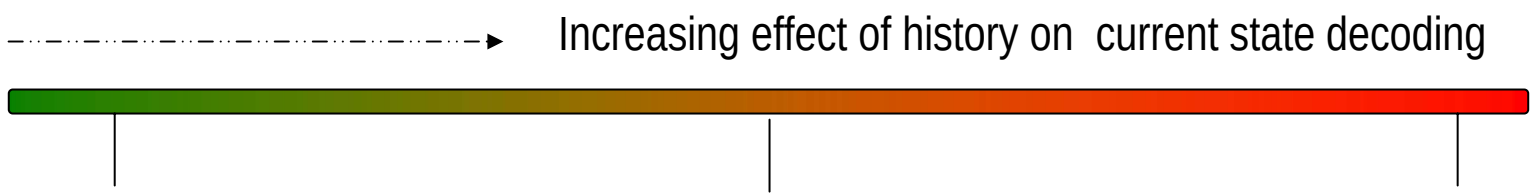
Black Box Verification	White Box Verification
Functional Intent	Implied Intent
Manual	Mostly Automated

Parallel Protocols

- Control Variables
  - Define state of transaction (Command, Burst, Response)
  - Define legal / illegal states (reachable / unreachable)
  - Define legal / illegal transitions between legal states
- Qualifier Variables
  - Add extra information to transaction (Address, Data)
- Every protocol can be decomposed into functional layers
  - Atomic Transactions: basic transactions like read / write
    - Handshake mechanism: Something holds until something else happens
    - Basic definition for start / end of request, response
  - Complex Transactions: adds extra sequential information
    - e.g. to bind atomic transactions in a “Burst mode”
    - Basic definition involves start / end of transaction ‘windows’
  - Ordering: remember history of pipelined events
  - Out of order execution: properties for inter-thread transitions

Serial Protocols

- DATA & CONTROL shared on same transmission line
  - Difficult to separate
- Data-frames are spaced over several clock-cycles
  - Requires “HISTORY” info: Detecting a pattern requires “remembering” previous train of signals – needs an FSM
  - E.g. Whether a train of 10 bits is a pre-amble or a post-amble depends on the first 3 bits in that sequence.



Simple Parallel Protocol:
CS = f(current values only)

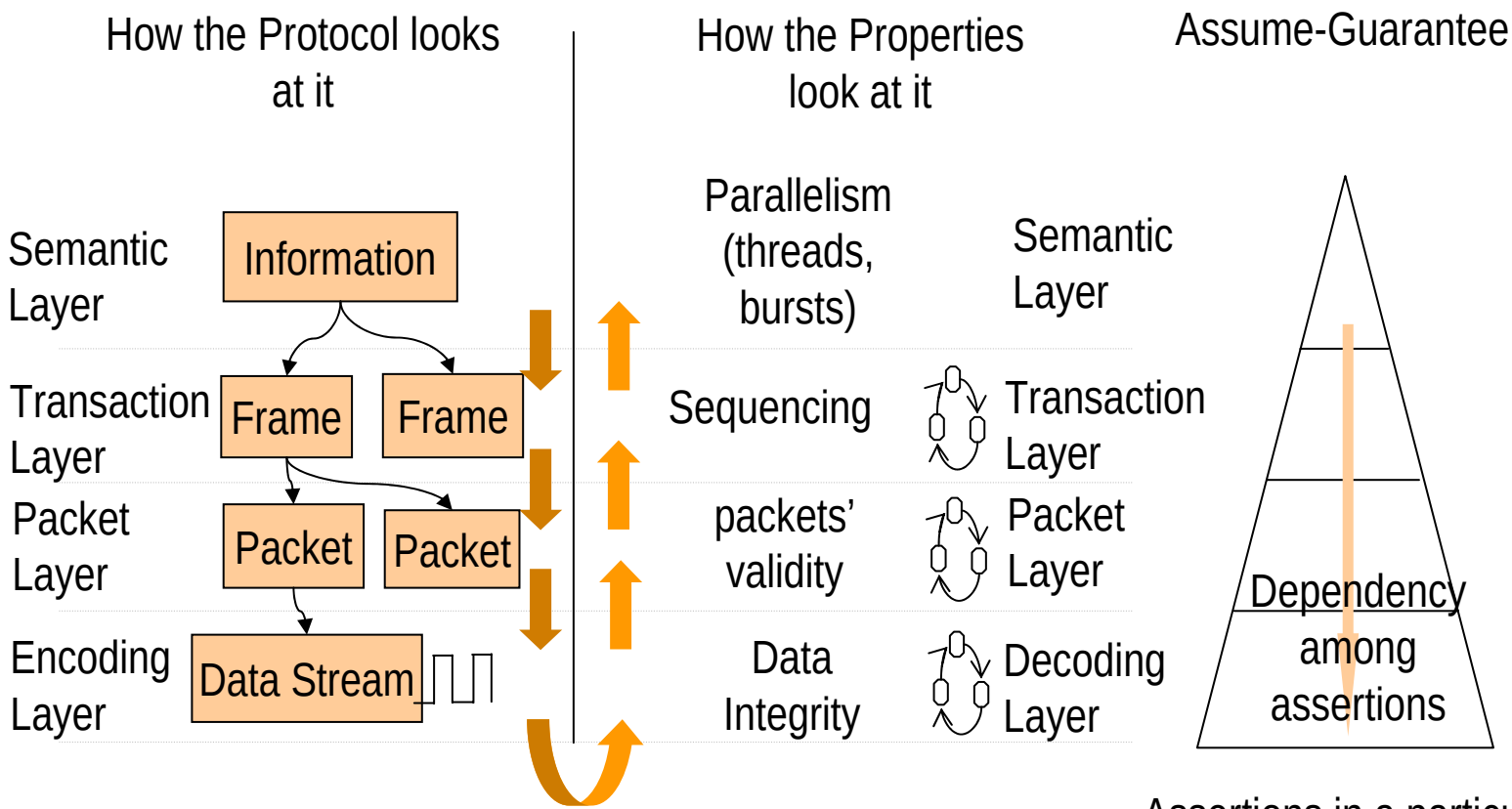
Complex Parallel Protocol
CS = f (current values, pipeline history, thread_id)

Serial protocol :
CS = f (history)

CS: Current state of protocol (transaction)

Property Coding for Protocols

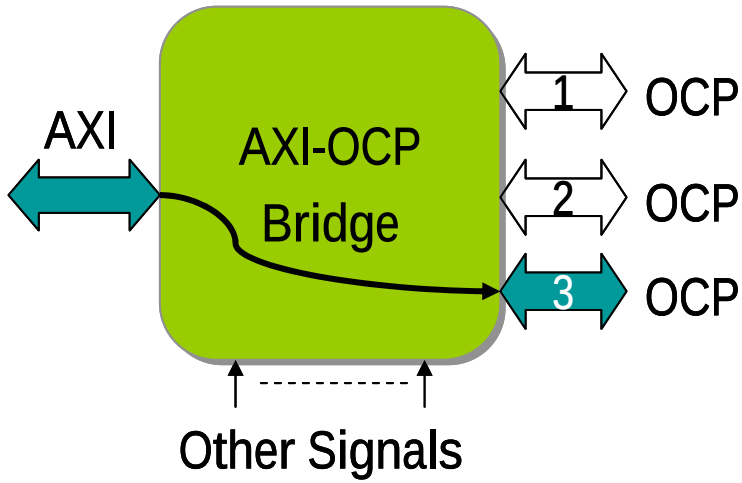
- Generic and small FSMs for assertion writing (for each layer)
- FSMs for a higher layer use output flags of lower layer FSMs
- Prove assertions bottom-up, using proven assertions as constraints



Assertions in a particular level depends on those at a lower level

AXI-OCP Bridge

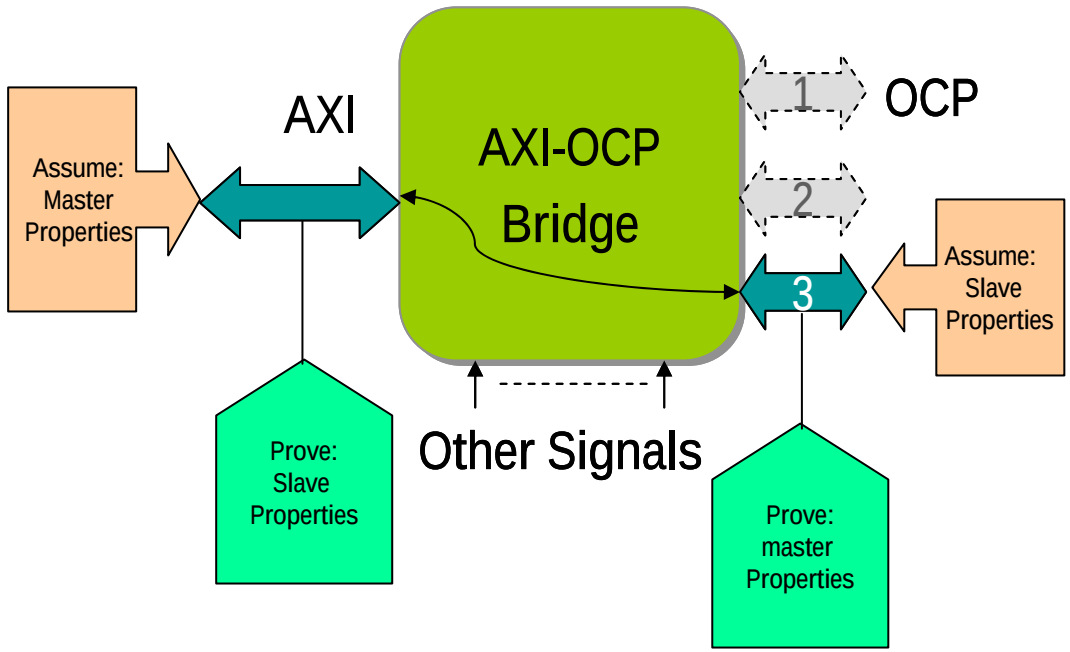
- Characteristics:
  - 1 AXI interface, 3 OCP-2.0 Interfaces
  - Flop Count : 560
  - Pipeline depth (AXI) : 5
  - Threads Supported
  - Functional reset happens 2 cycles after reset is asserted
- Scope of FV:
  - Protocol compliance check
  - No data integrity checks
- Initialization: Use Simulation
  - 2 cycles IDLE after primary reset is asserted



- Modeling:
  - Divide & Conquer
    - Verify each AXI-OCP path in isolation
    - Use Addresses to restrict other paths

AXI-OCP Bridge

- Verification:
  - AIPs used for AXI and OCP
  - First prove each path is independent
  - *"If address/thread-id is A3/T3, then MCmd at OCP interface 1 & 2 is always IDLE"*
  - Apply assume-guarantee
  - Prove assertions in bottom-up fashion



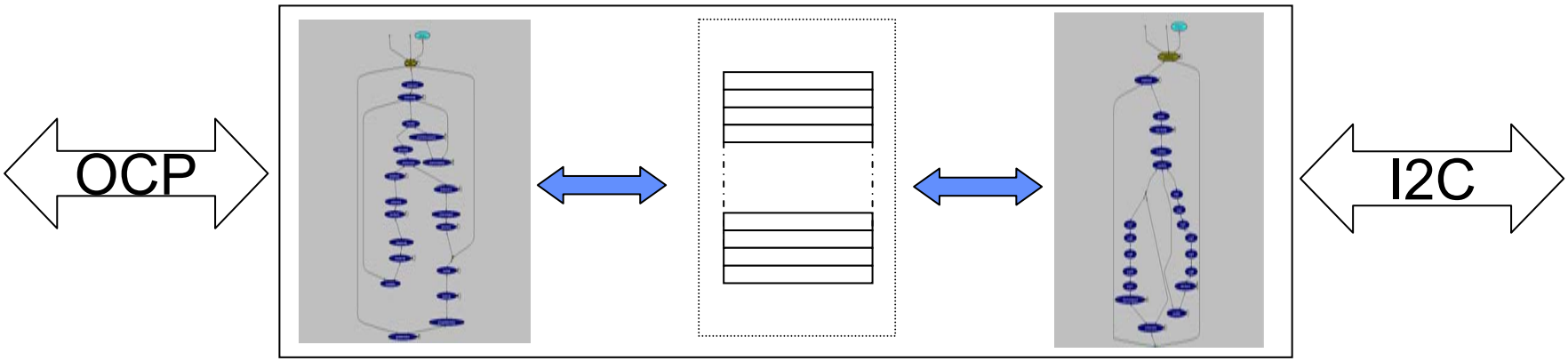
- Handling indeterminate proofs:
  - Reduce pipeline depth, # threads
  - Prove each burst type separately, using pin constraints
  - Bug hunting: initialization via simulation

AXI-OCP Bridge

- Project Statistics:
  - Number of Assertions : 125
  - Number of Constraints : 60
  - Number of Restrictions : 15
  - Total Time : 3 weeks (Tool runtime: 35%)
- Sample Results
  - BUGS / Anomalies
    - “Data was presented on the OCP port even prior to the corresponding command acceptance on AXI”
    - “Sequence of generated addresses incorrect for incrementing bursts”

I2C Controller

[Gorai et al, DAC-2006]



- Parallel to Serial Bridge:
  - 697 Flops
- Scope of Verification:
  - Root-cause a silicon bug
  - Prove that a software workaround is robust
- Result:
  - Found trace for the bug
  - Workaround valid for a specific sequence of commands only

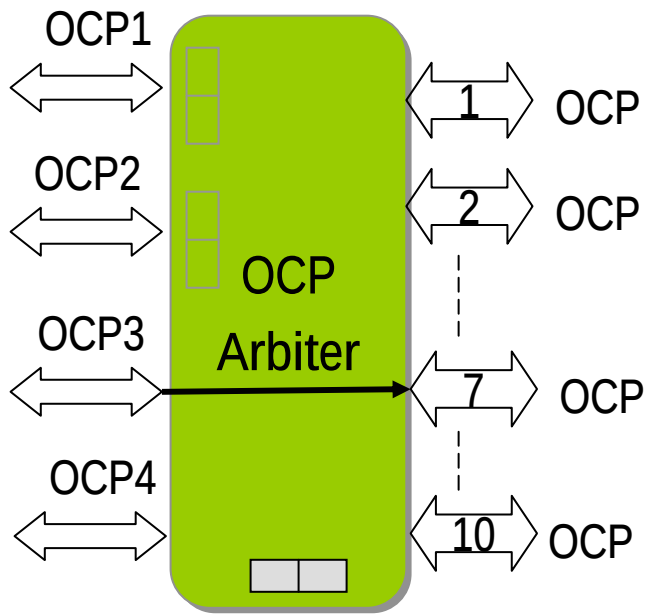
- Bug condition:
  - I2C Controller in Master Receive Mode
  - Number of bytes to be transferred *is Odd*
  - OCP fills FIFO with “addresses to be read”, and I2C transmits the address from FIFO (address phase)
  - I2C receives data from Slave (data phase) and set RRDY flag
  - OCP to read a byte from FIFO and clear RRDY flag
  - I2C detects “I2C-stop” condition and set ARDY flag
  - OCP to clear ARDY Flag
  - Bug: Near stop condition, RRDY cannot be cleared
  - Workaround: Clear ARDY and then clear RRDY flag

I2C Controller

- Verification Steps
  - OCP and I2C VIP's were configured and attached
  - Test pins, redundant OCP data pins were tied to constant values
  - Protocol Compliance checks
    - Proven I2C assertions were then used as constraints
  - Directed simulation to
    - Configured registers for MRx mode, for 1 byte transfer
    - Load transmit FIFO with data
    - Take the design to end of I2C address phase
- The Event window where RRDY could be cleared divided into 4 parts
  - Based on Set Flags & Stop conditions
- Bug located: RRDY could not be cleared in one region only
  - Effort: 2 months, mainly to write I2C AIP. Reuse: 2 weeks

OCP Arbiter

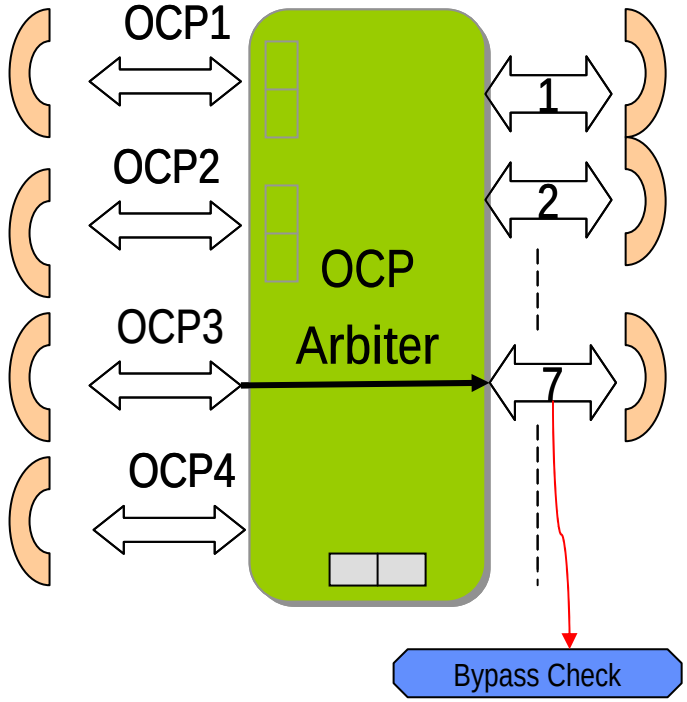
- Characteristics:
  - 4 OCP inputs (from masters)
    - ocp1, ocp2 with pipeline
    - ocp3, ocp4 no pipeline
  - 10 OCP outputs (to slaves)
  - Configurable Internal Register
  - Flops: 637
  - Round Robin Arbitration with priority-reset on 2 cycles IDLE
  - Bypass functionality
- Scope of FV:
  - Protocol compliance check
  - Arbitration Check
  - Bypass Check



- FV Complexity
  - 32 bit address lines
  - Internal Decoding Logic
  - Arbitration Exceptions:
    - Bypass
    - Configuration register

OCP Arbiter

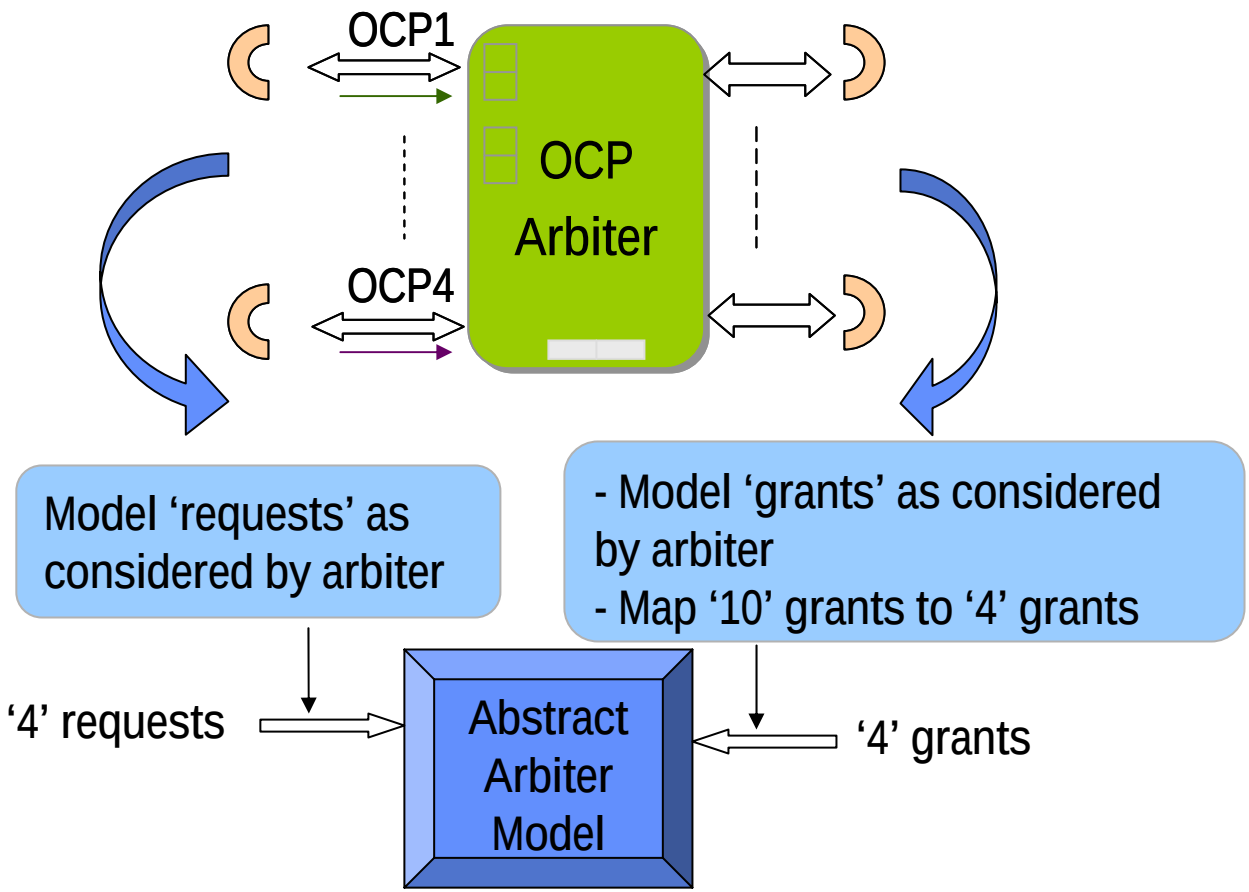
- Modeling:
  - Protocol Compliance Check
    - Assume-guarantee
  - Bypass Check
    - Whenever there is request from OCP3 to dedicated slave, it must get serviced with zero-cycle delay
  - Arbitration Check
    - Disable Configuration reg
    - Disable Bypass
  - Define events: Start / End of Requests



- Identify serviced masters:
  - Check for ByteEn pass-through
  - Tie ByteEn to unique values (to identify master requests)
  - Map the 10 requests on Slave side as 4 grants for masters

OCP Arbiter

- Create abstract arbiter model (assertions)
  - Round Robin Scheme with priority reset

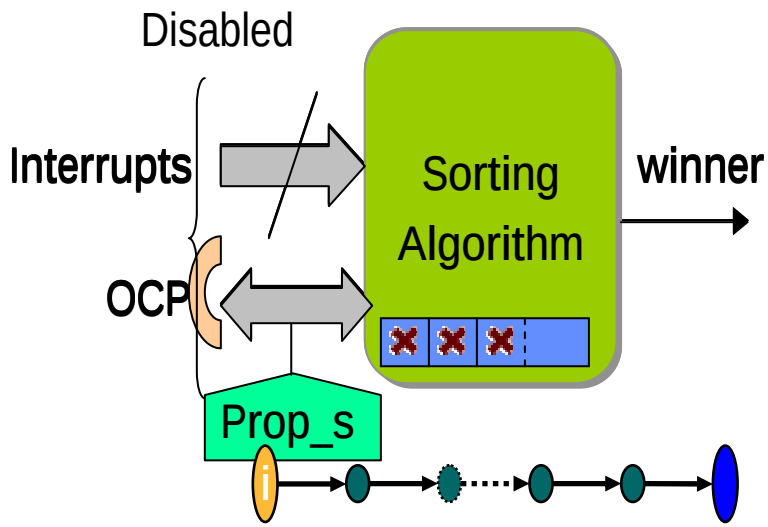


OCP Arbiter

- Challenges
  - Due to decoding logic, the proof complexity is very high
    - Address constraints: Allow only some number of active slaves
    - Restrict accesses: Take a set of 3 masters at a time
    - Using proved properties as constraints
  - Pipelining creates problems for accurate arbitration checks
    - Need to keep external history for pipeline: External buffer keeps track of 'number of pending requests' → counter implementation
- Sample BUGS / Anomalies discovered:
  - Bypass should have zero-cycle latency, whereas 1 to 2 cycle latency was possible under error conditions
  - Arbitration Error in the first cycle : uninitialized registers
  - Priority reset after idle cycles was not happening always

Interrupt Controller [Biswas et al, CDNLive-2005]

- Characteristics:
  - Sorting between 32/48/96 interrupts (configurable)
  - Mask and Priority configurable for each interrupt
  - OCP Interface (for configuration)
  - Flop Count : 785 (for 32 interrupts)
  - Output after 8 clock cycles (sorting in groups of 4)
- Scope of FV:
  - Protocol compliance check
  - Sorting Algorithm Verification



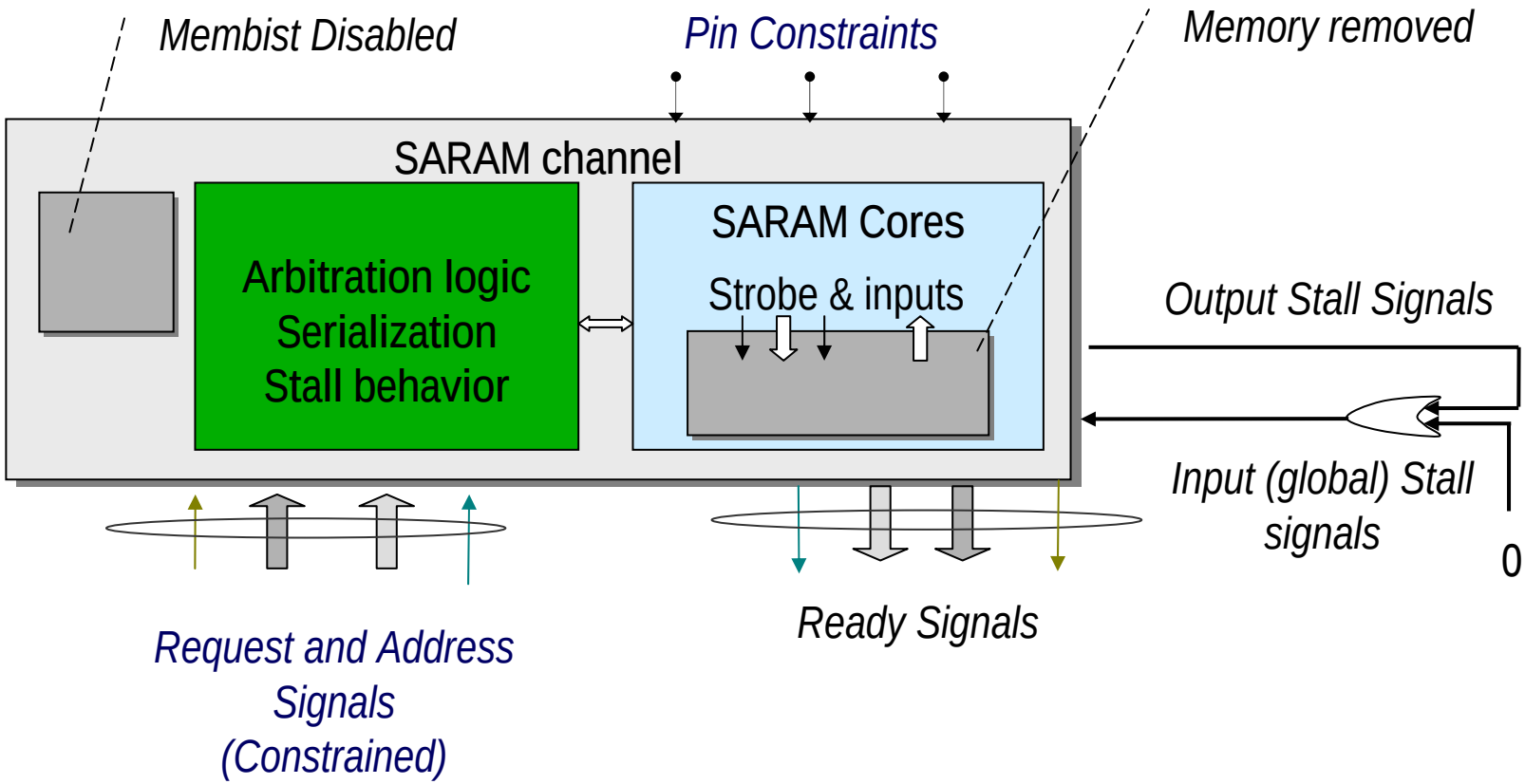
Interrupt Controller

- Challenges
  - Mask & priority registers programmable via OCP interface
    - Many combinations possible
- Solutions:
  - Prove integrity of OCP data writes to internal registers
  - Then apply stability constraints on register
  - For intra-class checks, tie-off other interrupts
  - For inter-class checks, constrain values within each class to be same
  - Compare expected winner (computed by Verilog function) with actual winner

Memory Controller

- Characteristics:
  - Interface to SARAM core
  - Handshaking (with stall) with other bank-controllers
  - Total number of request buses (cpu, dma etc.) :9
  - Fixed Intra as well inter priority between requests
  - Pipelined CPU requests, DMA burst requests
  - Flop Count : 3500
  - Embedded SARAM, BIST controller
- Scope of FV: Serialization check
  - "Order of grants must be same as the requests"
  - End-to-end property, encompasses all behavior of controller (mux, stall, arbitration, etc.)

Memory Controller



Memory Controller

- Solutions: White-box and Controlled Verification
  - Allow access to each core
    - But intelligently disable parts/bits of address bus
  - Keep out other bank controllers
    - Assume a floating primary input signaling their stall behavior
  - Identify and code assertions for internal points
    - E.g. Code and prove assertions for stall module
  - Then use them as constraints
- Results:
  - More than 30 bugs found, most of them corner cases (related to flush functionality of pending buffers and stall behavior, which in turn lead to serialization assertion failures)
  - Effort:
    - Assertions : 215, Constraints : 12
    - Total Time : 8 weeks (Tool runtime: 50%)

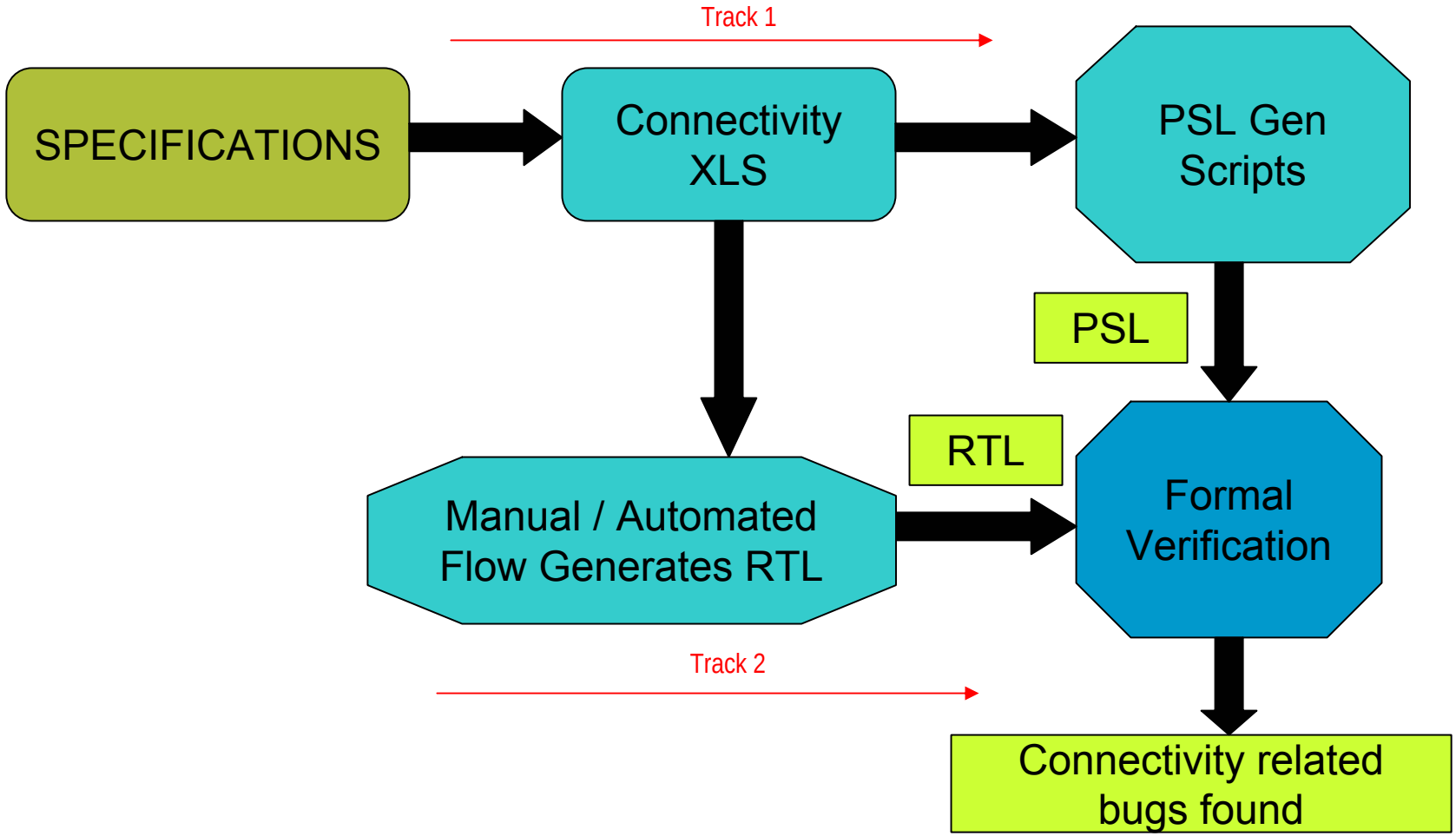
Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
  - Methodology
    - Planning for Formal Verification
    - Property Coding Guidelines
    - Formal Verification Flow
  - Case Studies
    - Protocol Compliance : Bridge Validation
    - Arbitration
    - Interrupt Sorter
    - Memory Controller
    - SoC Performance
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

SoC Connectivity Verification

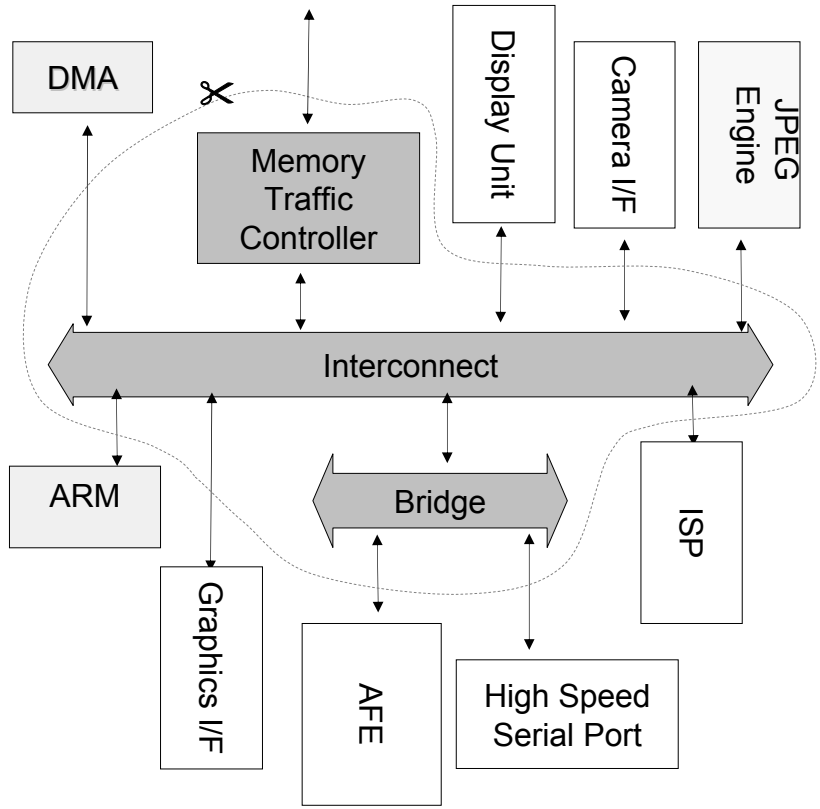
- Bus Architecture Connectivity, I/O Pin-Muxing Logic
- Errors caught easily and upfront in the design
  - Traditional verification involves exercising the complete communication protocol from one end point to find any bug
  - Verification starts much earlier, issues get resolved quicker
  - Errors detected: tieoffs, unconnected points, wrong connections
- Connectivity Description:
  - Interface Definition: Port, direction, width, high/low, etc.
  - Connectivity table: IPx-Porty to IPa-Portb
- Results:
  - Properties: ~ 5000 (at top-level of a 5M gate chip)
  - Time: 6 hours
  - Note: not checking for validity of xls itself, nor IP functionality
- To be done: Interrupt and Clock-reset connectivities

SoC Connectivity Verification



SoC Bandwidth Verification [Bhatia et al, DAC-2007]

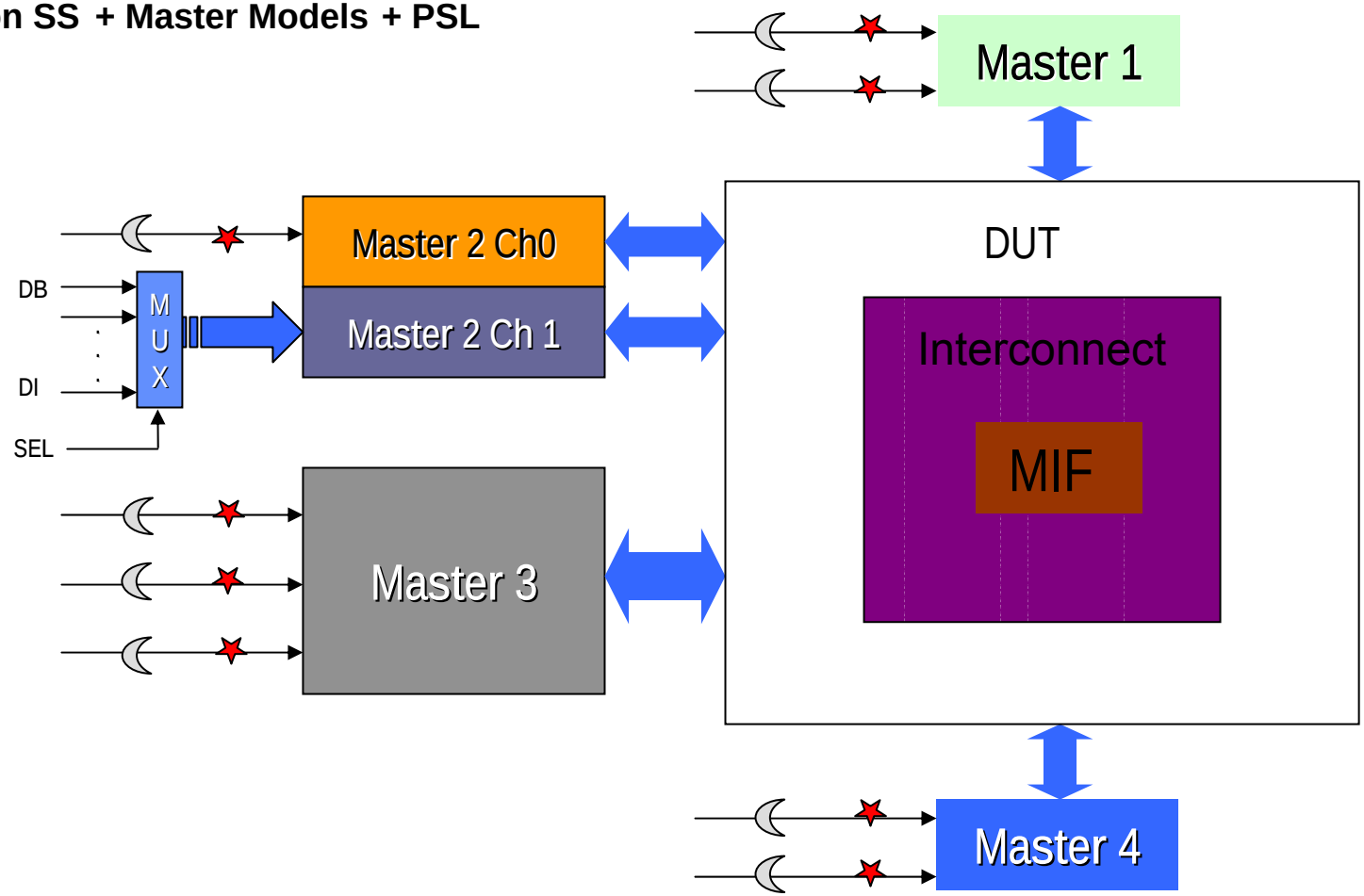
- Memory bandwidth validation:
  - Multiple masters accessing the memory subsystem
  - Master starvation affects system performance
  - Traditional methods: simulation, FGPA / emulation, running real software on real silicon
  - Formal Verification needed for corner case analysis on real RTL – early enough to fix design bottlenecks



- Properties:
  - Rate, e.g. "*Requests* are produced every 'n' time units": $t(Request[i+1]) - t(Request[i]) = n$
  - Latency, e.g. "*Response* is generated no more than 'k' time units after *Request*": $t(Response[i]) - t(Request[i]) <= k$
  - Throughput, e.g. "at least 'W' *Request* events will be produced in any period of 'T' time units": $t(Request[i+W]) - t(Request[i]) <= T$

SoC Bandwidth Verification

Verification SS + Master Models + PSL

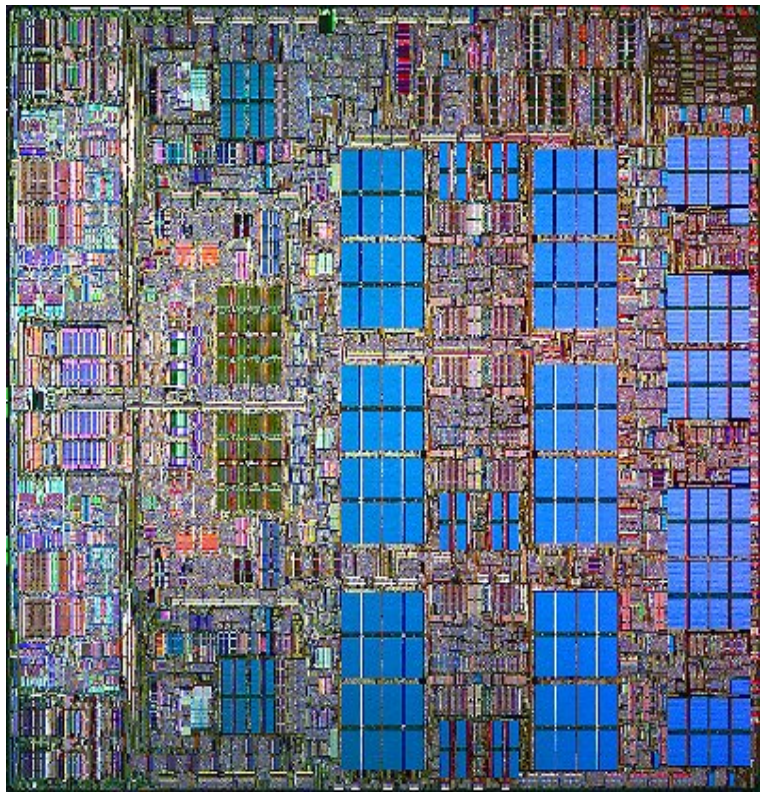


Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
  - Algorithmic Methods for Reducing Verification Complexity
  - Processor Verification Case Studies
    - Sequential Equivalence Checking (SEC)
    - Instruction Dispatch Case Study
    - Instruction Fetch-Hang Case Study
    - Floating-Point Unit Verification
    - Load-Store Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

POWER5 Chip: *It's Ugly*

- Dual pSeries CPU
- SMT core (2 virtual procs/core)
- 64 bit PowerPC
- 276 million transistors
- 8-way superscalar
- Split L1 Cache (64k I & 32k D) per core
- 1.92MB shared L2 Cache >2.0 GHz
- Size: 389 sq mm
- 2313 signal I/Os
- >>1,000,000 Lines HDL



POWER architecture:

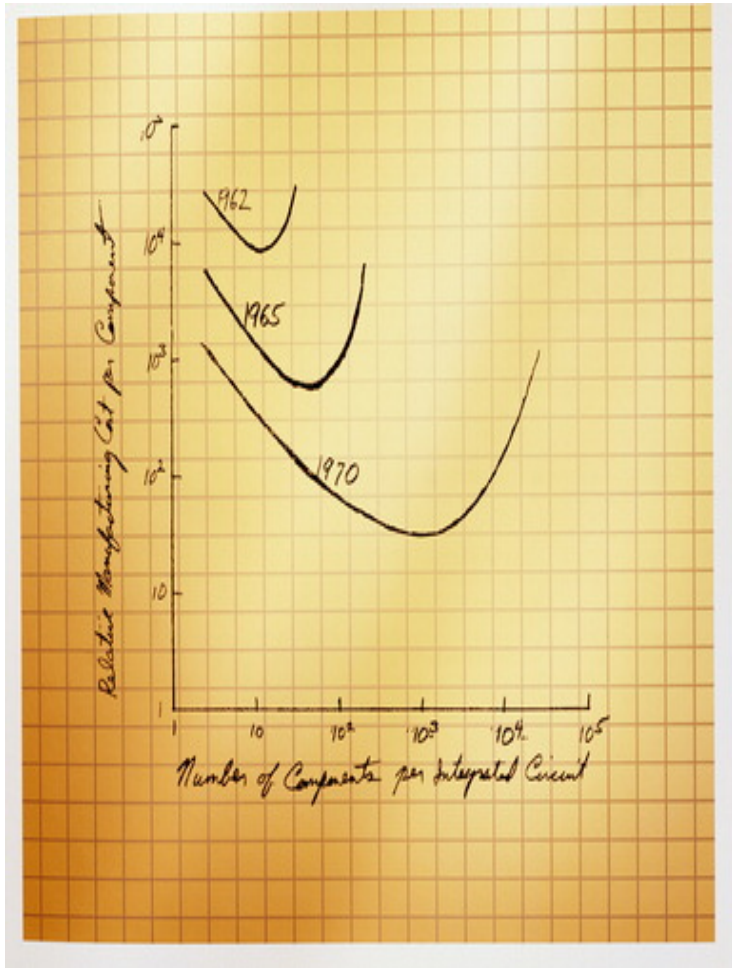
- Symmetric multithreading
- Out-of-order dispatch and execution
- Various address translation modes
- Virtualization support
- Weakly ordered memory coherency

Moore's Law: *AND It's Getting Uglier*

- Complexity increases for POWER5, POWER6, POWER7, …

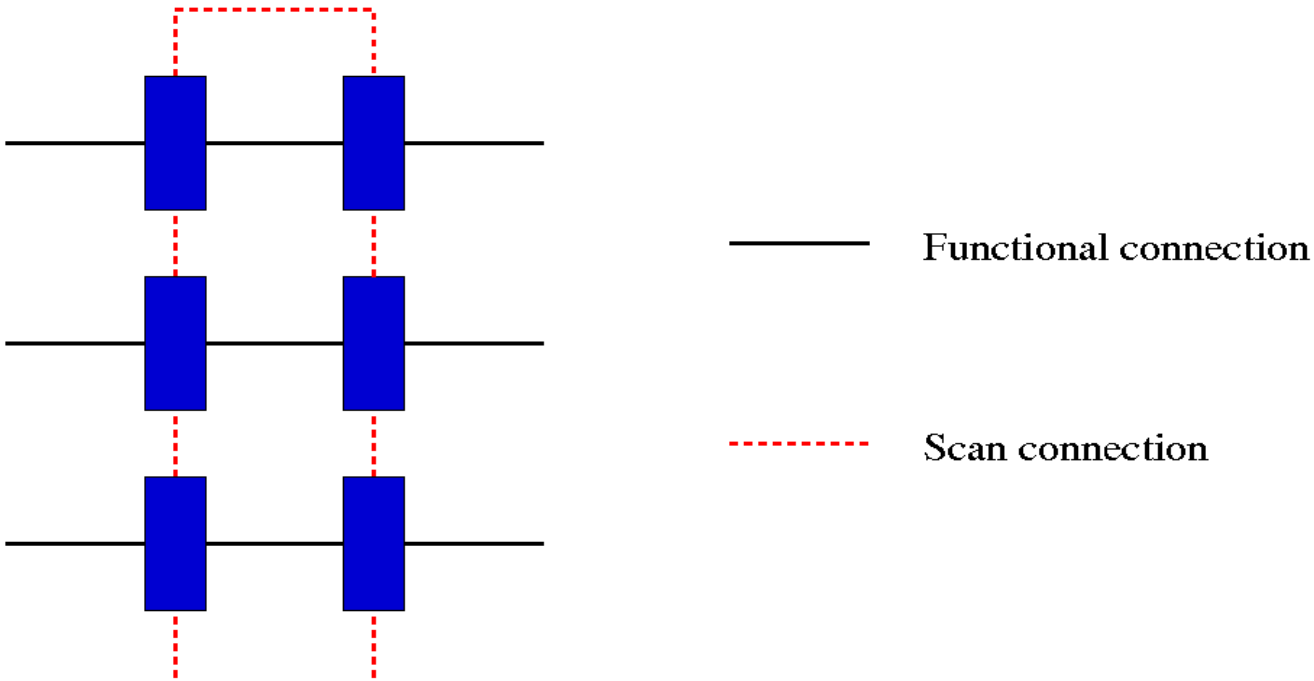
- Increase in # transistors / chip
  - Contributes *somewhat* to complexity
  - Alleviated by modularity
    - N identical proc cores on chip
  - Alleviated by increased RAM size

- Increase in speed
  - Contributes *significantly* to HDL complexity
  - CEC methodology requires 1:1 latch correspondence between circuit, HDL
    - CEC=Combinational Equiv Checking



Complexities of High-End Processors

- CEC methodology forces HDL to acquire circuit characteristics
  - Word-level operations, isomorphisms broken by self-test logic
    - Self-test logic: much more intricate than mere *scan chains*
    - Reused for functional obligations: initialization, reliability, …
  - Run-time monitoring logic entails similar complexities



Complexities of High-End Processors

- CEC methodology forces HDL to acquire circuit characteristics
  - Timing demands require a high degree of *pipelining*
    - And multi-phase latching schemes
  - *Placement issues:* redundancy added to HDL
    - Lookup queue routes data to 2 different areas of chip → replicate
  - *Power-savings logic* complicates even simple pipelines
  - Design HDL becomes difficult-to-parse bit-level representation
    - Industrial FPU: 15,000 lines VHDL vs. 500 line ref model
  - Sequential synthesis cannot yet achieve necessary performance goals
    - And need integration of *pervasive logic*: self-test, run-time monitors

Complexities of High-End Processors

- Numerous techniques have been proposed for Processor Verification
- Satisfiability Modulo Theories replaces word-level operations by function-preserving, yet more abstract, Boolean predicates
  - Replace complex arithmetic by arbitrary simple (uninterpreted) function
  - Reduce arithmetic proof to simpler data-routing check
- Numerous techniques to abstract large memories
- Numerous techniques to decompose complex end-to-end proofs
- …

Complexities of High-End Processors

- Difficult to find places to attempt such abstractions on high-end designs
  - Word-level info lost on highly-optimized, pipelined, bit-sliced designs
    - Designs are tuned to the extent that they are **"almost wrong"** R. Kaivola
  - Time-consuming + error-prone to manually abstract the implementation
- Abstractions may miss critical bugs
  - Removal of bitvector nonlinearities are lossy
    - May miss bugs due to rounding modes, overflow, … if not careful
- Need to verify a more abstract model?
  - Developing, maintaining such a model is *very expensive*
  - Emerging SEC technology: may close *abstract vs CEC model* gap

Complexities of High-End Processors

- Industrially, “Formal Processor Verification” refers to proc components
  - E.g., verification of FPU, Cache Controller, Branch Logic

- “Dispatch to Completion” proofs for processors as complex as Pentium, POWER, … are **intractable** given today’s technologies

- In this tutorial, we focus on various case studies of the verification of processor components

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
  - Complexities of High-End Processors
  - Processor Verification Case Studies
    - Sequential Equivalence Checking (SEC)
    - Instruction Dispatch Case Study
    - Instruction Fetch-Hang Case Study
    - Floating-Point Unit Verification
    - Load-Store Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

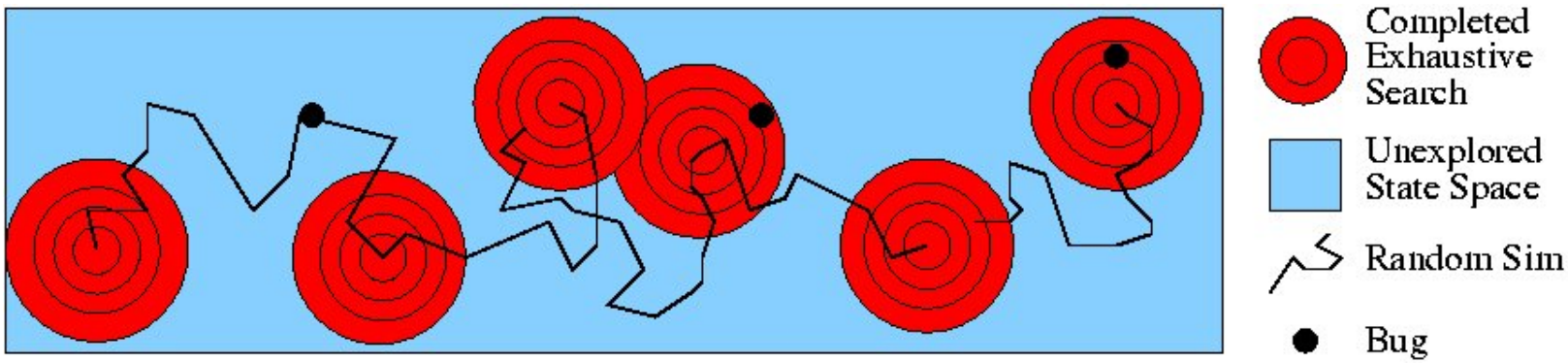
Reducing Verification Complexity

- We discussed characteristics of high-end processors that make them large, complex, *difficult* to *verify*
  - High degree of pipelining
  - Multi-phase latching schemes
  - Addition of sequential redundancy
  - Loss of word-level and isomorphic characteristics

- Some may be addressed using automatic vs. manual techniques

Semi-Formal Verification (SFV)

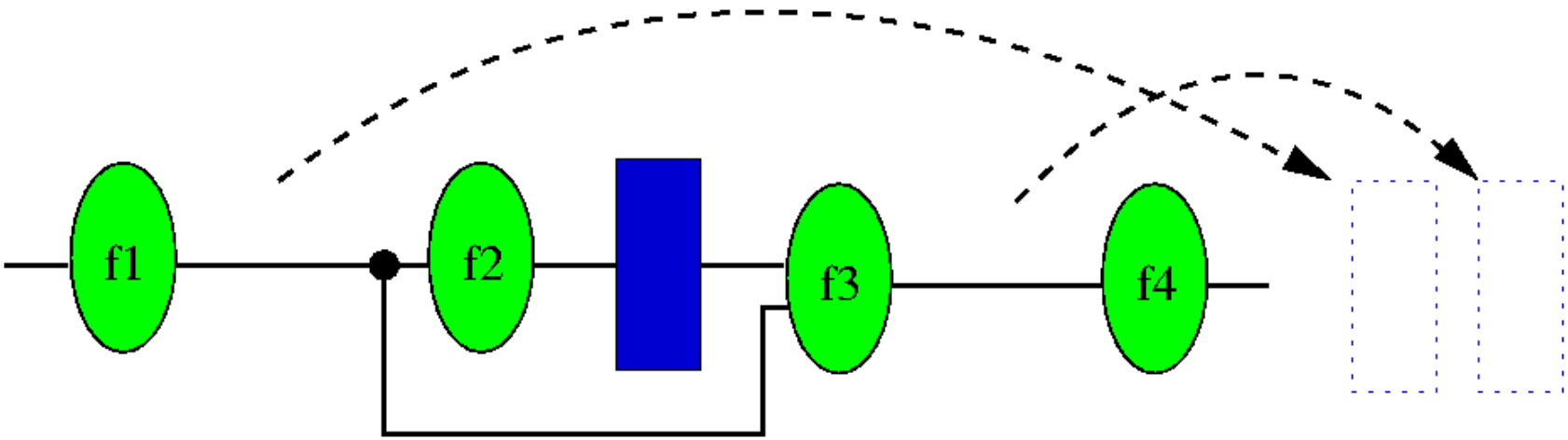
- SFV: hybrid between simulation and FV
  - Uses resource-bounded formal algos to amplify simulation results
  - Uses simulation to get deep into state space
- Hybrid approach enables highest coverage, if design too big for FV
  - Scales to large designs, without costly "manual abstraction"
- Incomplete, like simulation
  - May trade proof capability for high semi-formal coverage



Pipelining

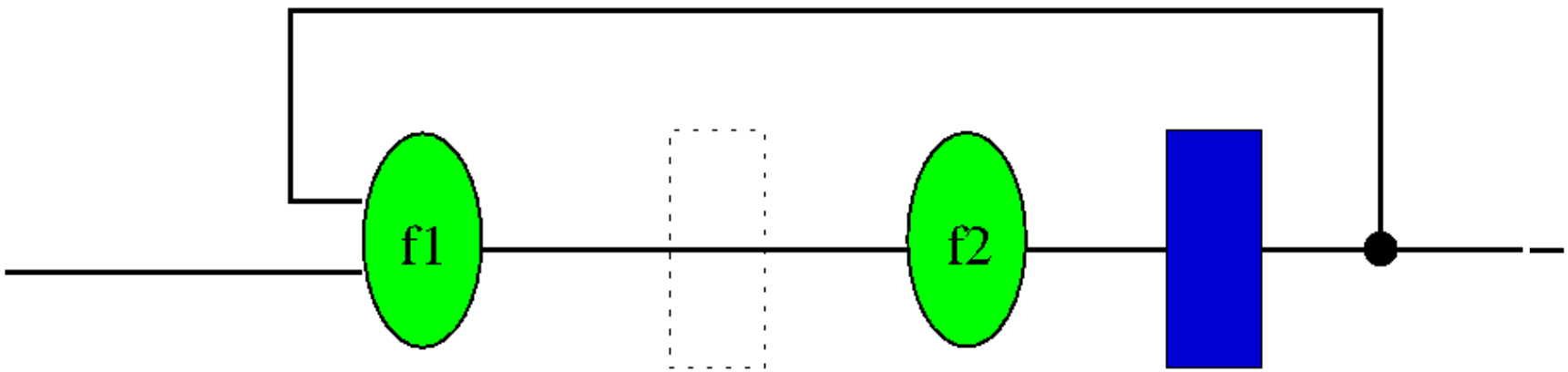
- High-end designs often use aggressive pipelining to break computations across multiple clock periods
  - Increases #state elements; diameter
- Min-area peripheral retiming may be used to drop latch count
  - Used as an automatic, property-preserving transform

"Transformation-Based Verification using Generalized Retiming" CAV01



Multi-Phase Latching Schemes

- High-end designs often use multi-phase level-sensitive latches to better distribute propagation delays vs. edge-sensitive flops
  - Increases #state elements; diameter
- Phase abstraction can automatically convert multi-phase to *simple-delay* model
  - Unfold next-state functions modulo 2, removing oscillator + ~1/2 latches

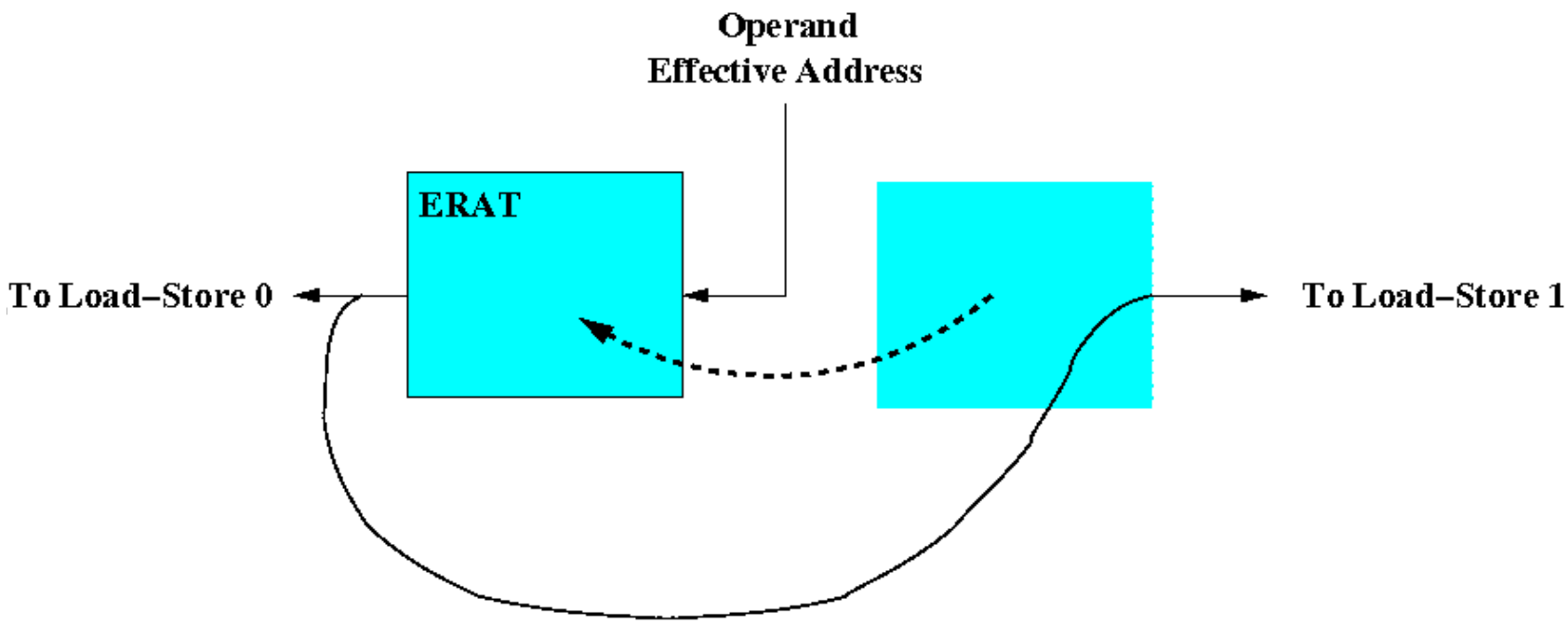


"Automatic Generalized Phase Abstraction for Formal Verification" ICCAD05

Sequential Redundancy

- High-end designs use sequential redundancy to minimize propagation delays
- Pervasive logic → sequential, yet disabled, logic intertwined with all latches
  - Increases #state elements; breaks isomorphisms, word-level properties
- Numerous techniques exist to identify, merge redundant gates

"Exploiting Suspected Redundancy without Proving It" DAC05

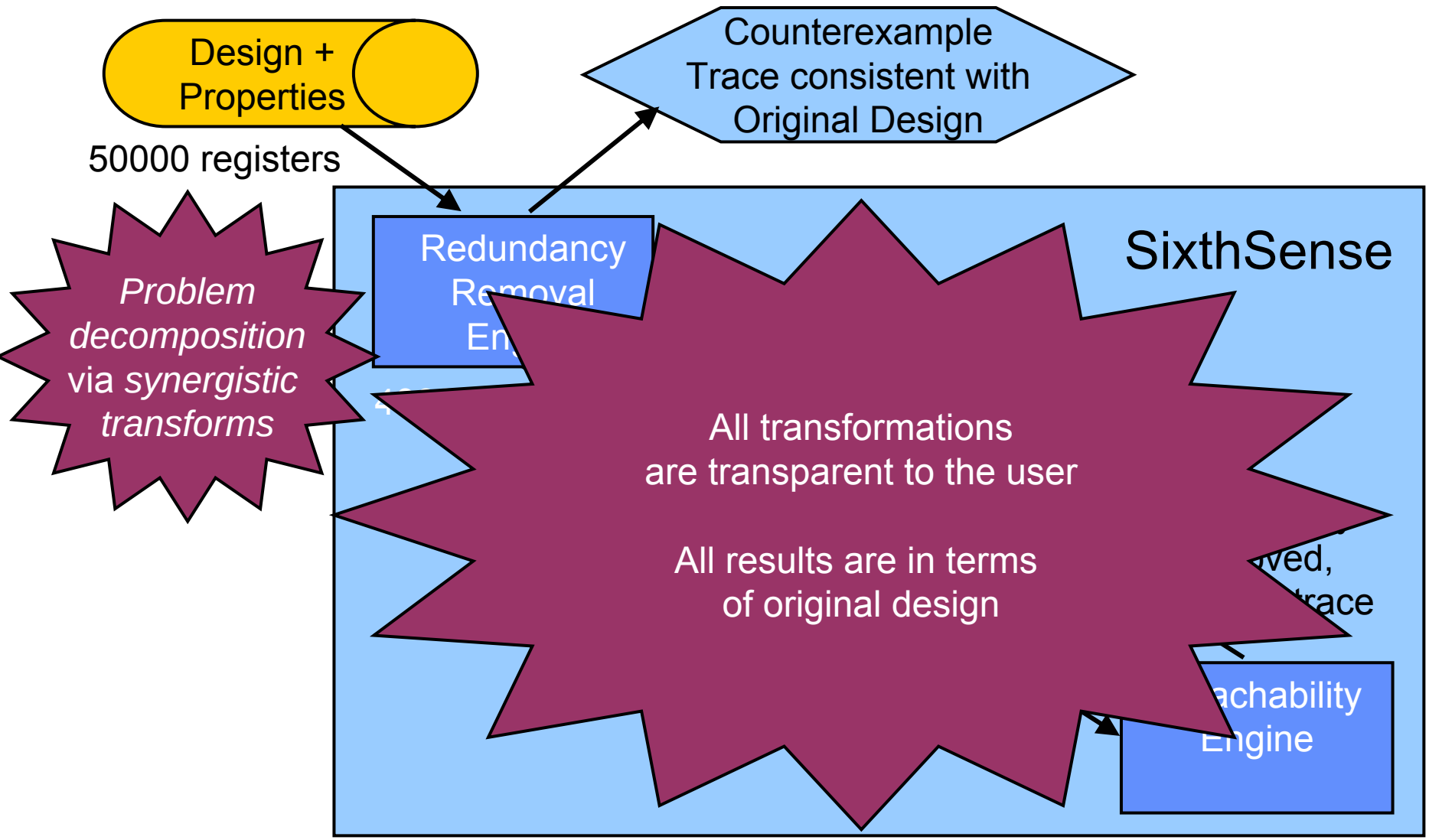


Reducing Complexity through Transforms

- High-end design complexities can be (partially) alleviated through automated transformations and abstractions

- Our solution at IBM leverages transformation-based verification
  - Fully automated, scalable (S)FV toolset *SixthSense*
  - Iterate synergistic transforms to decompose large problems into simpler sub-problems
    - Localization, retiming, phase abstraction, redundancy removal, …
  - Leverage various proof+falsification engines to solve resulting problems
    - Semi-formal search, bounded model checking, interpolation, …

Transformation-Based Verification



Reducing Verification Complexity

- Automated techniques are continually increasing in capacity
- However, for complex proofs, manual techniques are critical to push the capacity barrier
  - Choice of testbench boundaries
  - Manual abstractions to reduce design complexity
  - Underapproximations and overapproximations
  - Strategic development of constraints and properties
- The best strategy often depends upon some knowledge of available algos

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
  - Complexities of High-End Processors
  - Algorithmic Methods for Reducing Verification Complexity
    - Sequential Equivalence Checking (SEC)
    - Instruction Dispatch Case Study
    - Instruction Fetch-Hang Case Study
    - Floating-Point Unit Verification
    - Load-Store Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Processor Verification Concepts

- A verification *Testbench* comprises
  - Design components under verification
  - Input assumptions
    - May be represented as *constraints*
    - Or may be overridden by random *drivers*
      - We refer to either scheme as a *driver*
  - Properties to be checked
    - Coverage metrics, assertions, equivalence vs. ref model
  - Initialization information
    - *Initial states* impact *reachable states* which impacts *pass vs fail*

Processor Verification Concepts

- Verifying a *proc component* is basically the same as verifying *any* design
  - However, due to high-performance logic, verifying architectural properties typically requires a Testbench which is too large for proofs

- Options:
  1. Develop unit-level Testbench without worrying about proof feasibility
  2. Develop minimal Testbench encompassing only functionality to be verified

Processor Verification Concepts

1. Develop unit-level Testbench without worrying about proof feasibility
   - Unit-level testbenches often built for sim regardless
     - Use of synthesizable language → reusable for FV, emulation, …
   - Leverage semi-formal verification for *bug-hunting*
     - With luck, robust tool may yield some proofs regardless
     - But likely need hand-tweaking of Testbench for proofs

- Easier for non-experts to leverage (S)FV
  - Manual abstraction is **time-consuming and difficult**
  - Even if using experts to abstract, disperses formal spec effort

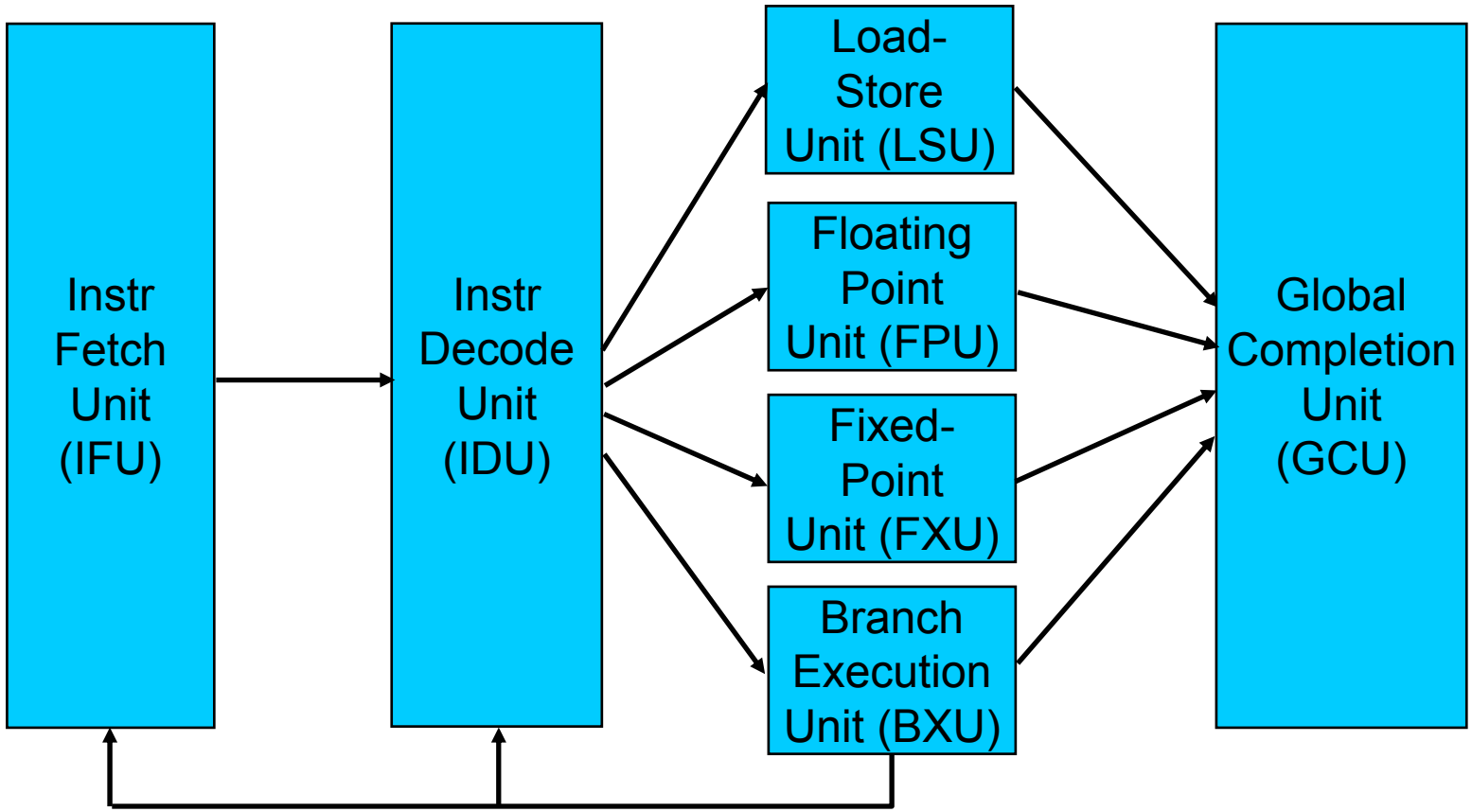
- Easier to specify desired properties at unit level
  - Verify *functionality* vs. verify *blocks*
  - Difficult to *cover* architectural properties on small blocks

Processor Verification Concepts

2. Develop minimal Testbench encompassing only functionality to be verified
   - Requires dedicated effort to enable FV
     - Checks and Assumptions *may* be reusable in higher-level sim
     - But often need to develop a higher-level Testbench for sim
   - Block-level Testbench often more complex to define than unit-level
     - More complex, prone to change, poorly documented input protocol

- Works very well if done by *designer* at *design granularity* level
  - E.g. designer of Data Prefetch building Testbench at that level
- Higher chance of *proofs*, *corner-case bugs* on smaller Testbench
  - Data Prefetch is much smaller than entire Load-Store Unit!

Basic CPU Architecture



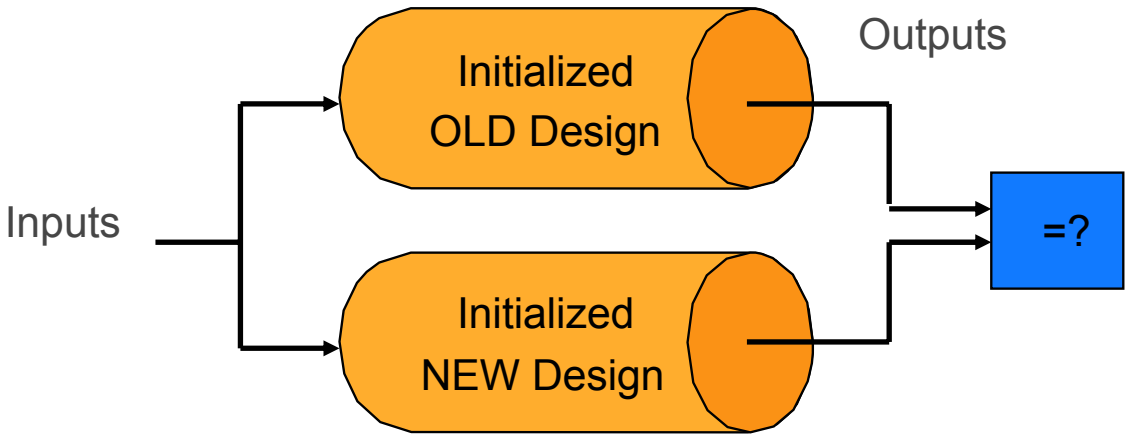
Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
  - Complexities of High-End Processors
  - Algorithmic Methods for Reducing Verification Complexity
    - Sequential Equivalence Checking (SEC)
    - Instruction Dispatch Case Study
    - Instruction Fetch-Hang Case Study
    - Floating-Point Unit Verification
    - Load-Store Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Pseudo Case-Study: SEC

- Performance-critical designs undergo *many* functionality-preserving transforms
  - Timing optimizations
  - Power optimizations
  - Addition of test and run-time monitoring logic

- Using SEC to validate such changes can **dramatically** reduce verif resources



Pseudo Case-Study: SEC

- Due to synthesis limitations, transforms often done manually
  - Entail risk of (late) error introduction
  - Suboptimalities tolerated to minimize error risk
- SEC is *very easy* to run: no difficult Testbench setup
- SEC often runs quickly, and *exhaustively* checks equivalence
  - Eliminates risk; enables more aggressive, later optimizations
- Without SEC, functional verif is rerun upon HDL modification
  - CPU-months of simulation regressions
  - FV testsuites re-run

Pseudo Case-Study: SEC

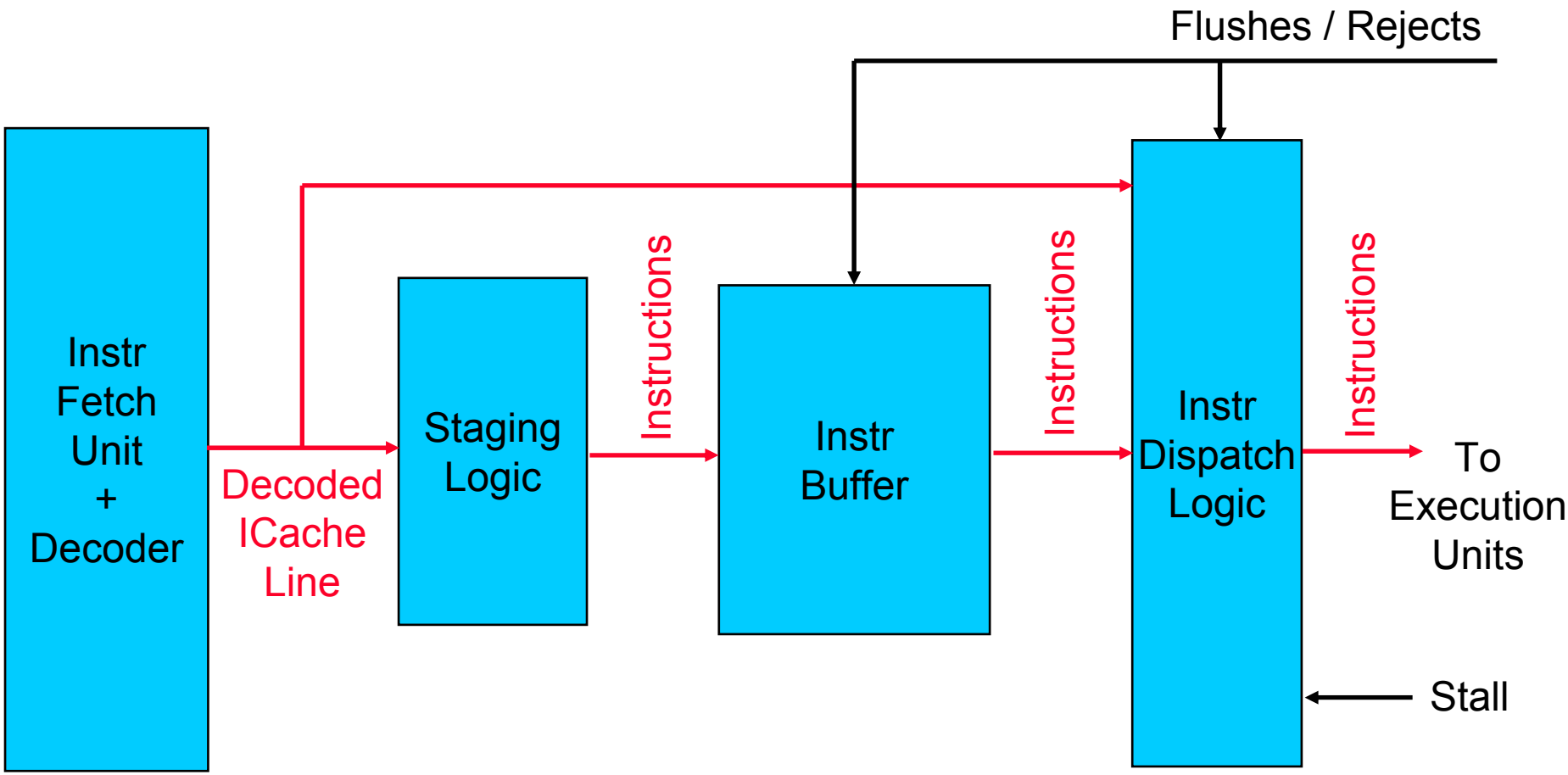
- SEC also useful for:
  - Reference-model style verification
    - Less circuit-like model serves as extensive “property set”
  - Demonstrate bwd-compatibility of design evolutions
    - E.g. if HW multithreading added to *old* design, SEC can confirm “single thread” mode of *new* matches *old*
  - Quantify: functional changes do not alter unintended behavior
    - If *opcode1* handling changed, constrain and check equivalence across all other opcodes

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
  - Complexities of High-End Processors
  - Algorithmic Methods for Reducing Verification Complexity
  - Processor Verification Case Studies
    - Sequential Equivalence Checking (SEC)
    - Instruction Fetch-Hang Case Study
    - Floating-Point Unit Verification
    - Load-Store Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Instruction Dispatch Case Study

- Concerns the portion of the Instruction Decode Unit responsible for routing valid instruction groups to execution units



Instruction Dispatch: Verification Goals

- Verify that Dispatched instructions follow program order, despite:
  - Stalls
  - Flushes (which roll back the Dispatch flow to prior *Instr* Tag)
  - Branch Mispredicts (similar to Flushes)
  - Rejects (which force re-issue of instructions)
  - Bypass path

Instruction Dispatch: Logic Boundaries

- First choice: what logic to include in Testbench?
  - Independent verif of Instr Buffer, Staging Logic, Dispatch Logic attractive from *size* perspective, but hard to express desired properties
    - Decided to include these all in a single testbench
  - Decoded instructions were mostly data-routing to this logic
    - Aside from special types (e.g. Branch), *this logic* did not interpret instructions
    - Hence drove Testbench at point of decoded instruction stream
  - Though infrequent during normal execution, this logic must react to Rejects, Stalls, Flushes at any point in time
    - Hence drove these as completely random bits

Instruction Dispatch: Input Modeling

- Second choice: how to model input behavior
  - Needed to carefully model certain instruction bits to denote *type*
    - Branch vs. Regular types
  - Other bits were unimportant to this logic
    - *Precise* modeling: allow selection of exact legal decodings
      - Manually intensive, and large constraints may slow tool
    - *Overapproximate* modeling: leave these bits free
      - Ideal since overapproximation ensures no missed bugs
      - But large buffers imply large Testbench!
    - Instead, used the bits *strategically*
      - Tied some constant, to reduce Testbench size
      - Randomized some, to help ensure correct routing
      - Drove one bit as parity, to facilitate checks
      - Encoded “program order” onto some bits, to facilitate checks

Instruction Dispatch: Property Modeling

- Third choice: how to specify properties to be checked
  - Dispatches follow instruction order:
    - Easy check since driver uses bits of instr to specify program order
    - Check for incrementing of these bits at Dispatch
  - Flushes / Stalls roll back the Dispatch to the proper instruction
    - Maintain a reference model of correct Dispatch Instr Tag
  - Dispatched instructions are valid
    - Check that output instructions match those driven:
      - Correct “parity” bit
      - Patterns never *driven* for a valid instruction are never read out
        - Drove “illegal” patterns for instructions that must not be read out

Instruction Dispatch: Proof Complexity

- Recall that driver tricks were used to entail simpler properties
  - Check for incrementing “program counter” bits in Dispatched instr

- Without such tricks, necessary to keep a reference of correct instruction
  - Captured when driven from Decoder; checked when Dispatched
    - More work to specify
    - Larger Testbench, more complex proofs, due to reference model

- Shortcut possible since this logic treated most instruction bits as data
  - If Testbench included execution units, shortcut would not be possible

Instruction Dispatch: Proof Complexity

- Philosophy: “don’t be precise where unnecessary for a given testbench” is very powerful for enabling proofs
  - Instr Dispatch requires precise *Instr Tag* modeling due to flushes; does not care about *decoded instr*
  - Some downstream Execution Units don’t care about *Instr Tag*; require precise *instr code*

- However, this occasionally runs contrary to “reusable properties”
  - E.g., “patterns which cannot be driven are not Dispatched” check cannot be reused at higher level, where overconstraints are not present

Instruction Dispatch: Proof Complexity

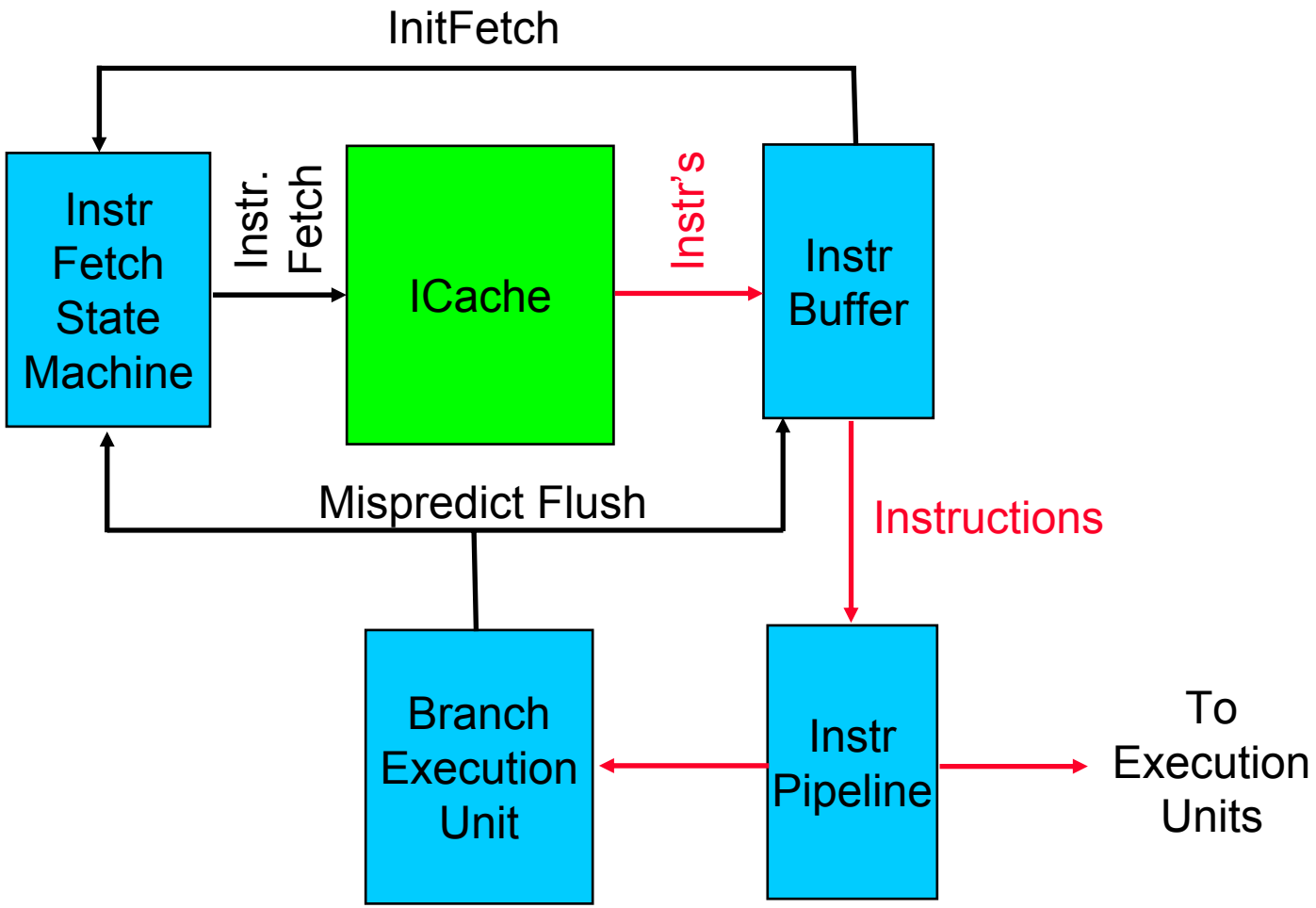
- Semi-Formal Verification was main work-horse in this verification effort
  - Wrung out dozens of bugs
    - Corner-cases due to Flushes, Stalls, Bypass, …
  - For SFV, biasing of random stimulus important to enable sim to provide a reasonable sampling of state space
    - Needed to bias down transfers from IFU, else Instr Buffer always full
- Parameterizing size of Instr Buffer smaller, setting more decoded instr bits constant helped enable proofs

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
  - Complexities of High-End Processors
  - Algorithmic Methods for Reducing Verification Complexity
  - Processor Verification Case Studies
    - Sequential Equivalence Checking (SEC)
    - Instruction Dispatch Case Study
    - Floating-Point Unit Verification
    - Load-Store Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

Instruction Fetch Case Study

- Motivated by an encountered deadlock:
  - Instruction Fetch Unit stopped fetching instructions!



Fetch-Hang Case Study

- Suspected: Instr Fetch State Machine (IFSM) can enter illegal *hang* state

- First tried to isolate IFSM in a Testbench
  - Despite simple previous Figure, formidable to specify accurate driver due to numerous ugly timing-critical interfaces
  - With underconstrained Testbench, checked whether IFSM could enter a state where it did not initiate Instr Fetch after InitFetch command
  - Discovered a hang state – yet could not readily extrapolate the tiny counterexample to one of the entire IFU+IDU
    - Exhibited input timings thought to be illegal
    - Yet designer was able to discern a scenario which, if producible, could lead to deadlock

Fetch-Hang Case Study

- Given extrapolated scenario, next attempted to produce that scenario on larger IFU+IDU components
  - Interfaces at this level were very easy to drive
    - Abstracted the ICache to contain a small program
  - However, VERY large+complex Testbench
    - Could not get *nearly* deep enough to expose condition which could reach hang state
- Used 2 strategies to get a clean trace of failure:
  - Tried to define the property as an earlier-to-occur scenario
  - Constrained the bounded search to extrapolated scenario

Fetch-Hang Case Study

- Extrapolated scenario:
  - Stream A is being executed, encounters a *branch* to B (to be taken)
  - Instructions in-line from A are still being fetched to Instr Buffer at time of branch resolution
  - Somehow the in-line instructions are not immediately invalidated
  - Fetch to B is delayed until exactly the point that the in-line instructions are dispatched out of the Instr Buffer (InitFetch)
    - This can put the IFSM into the dangerous *hang* state
  - Somehow a Mispredict Flush does not get triggered (to squash the in-line instructions) to break IFSM out of the *hang* state

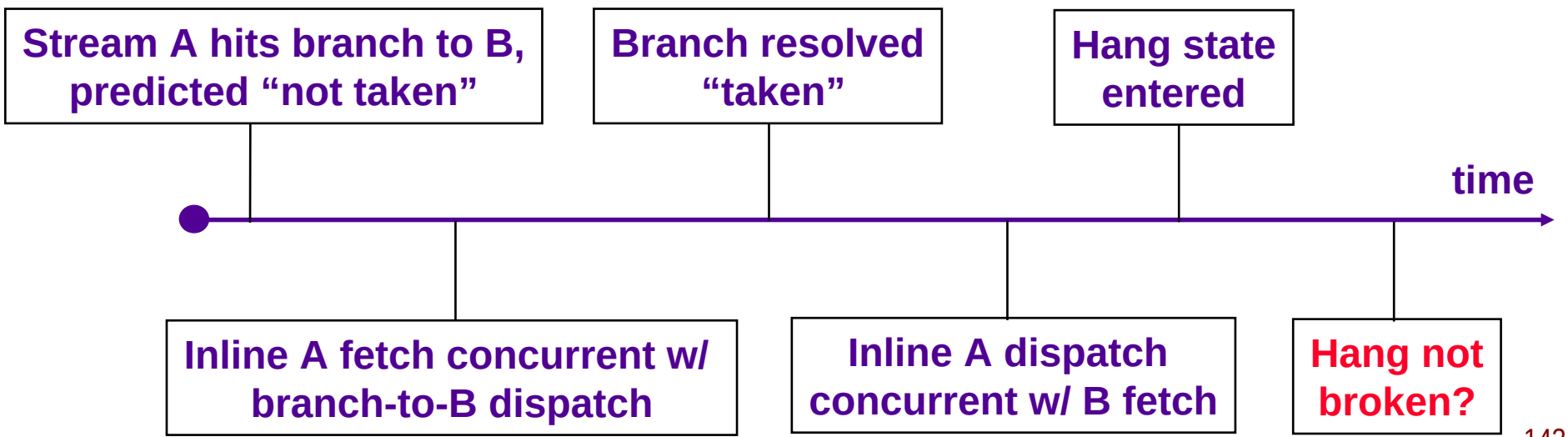
time

- Difficult to discern how to *complete* scenario to end up in deadlock

Fetch-Hang Case Study

- Reachability of *hang state* on full Testbench possible with BMC
  - However, normal execution always kicked IFSM out of hang state

- But trace provided useful insight: an in-line instruction may avoid invalidation if fetched during 1-clock window where *branch* is dispatched
  - This information, plus the timing at which activity occurred during the BMC trace, was used to constrain a deeper BMC check



Fetch-Hang Case Study

- Constrained BMC run exposed the deadlock situation!
- Address B exactly same address as in-line instructions from A which spuriously made it through Instr Buffer
  - Other conditions required, e.g. no spuriously dispatched branches
- However, removing constraints to check for alternate fail conditions (and validity of fix) became intractable even for BMC
  - Tried manual abstractions of Testbench to cope with complexity
    - Replaced Instr Buffer with smaller timing-accurate abstraction
    - Still intractable due to depth, size of Fetch logic
  - Realized we needed purely abstract model to approach a proof

Fetch-Hang Case Study

- Built a protocol-style model of entire system, merely comprising timing information and handling of relevant operations
- Validated (bounded) cycle accuracy vs. actual HDL using SEC
- Easily reproduced failures on unconstrained protocol model
  - Then verified HW fix: closing one-clock timing window
  - Also verified SW fix: strategic *no-op* injection
    - Clearly wanted to inject as few as possible for optimality
    - Modeled by adding constraints over instruction stream being executed upon abstract model
    - Re-running with constraints yielded proof

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
  - Complexities of High-End Processors
  - Algorithmic Methods for Reducing Verification Complexity
  - Processor Verification Case Studies
    - Sequential Equivalence Checking (SEC)
    - Instruction Dispatch Case Study
    - Instruction Fetch-Hang Case Study
    - Load-Store Verification
- Verification Closure: Coverage Analysis & Integration with Simulation

FPU Case Study

- Floating point number format: $M * B^E$
  - M: *Mantissa* e.g. 3.14159
  - B: *Base*, here B=2
  - E: *Exponent*, represented relative to predefined *bias*
    - Actual exponent value = *bias* + *E*

- A *normalized* FP number has Mantissa of form 1.?????
  - Aside from *zero* representation

- *Fused multiply-add* op: *A*B* + C for floating point numbers A,B,C
  - C referred to as *addend*
  - *A*B* referred to as *product*

- *Guard bits*, *rounding modes*, *sticky bits* used to control rounding errors

FPU Case Study

- Highly-reusable methodology developed for FPU verification

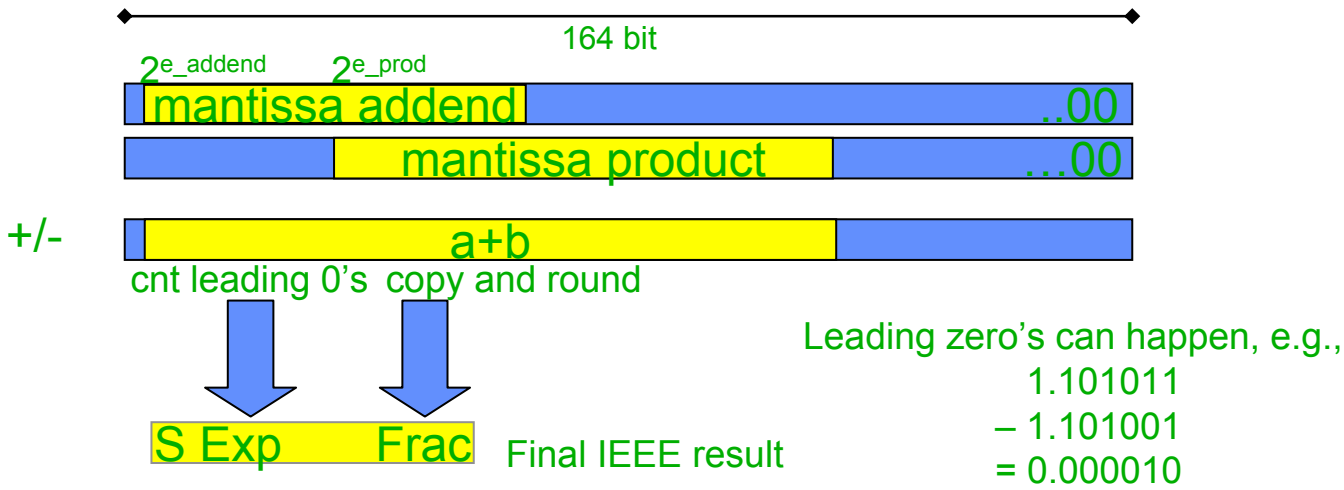
- Checks numerical correctness of FPU datapath
  - Example bugs:
    - If two *nearly equal* numbers subtracted (causing cancellation), the wrong exponent is returned
    - If result is near underflow, the wrong guard-bit is chosen

- Focused upon a single instruction issued in an empty FPU
  - Inter-instruction dependencies independently checked, conservatively flagged as error

FPU “Numerical Correctness”

- Uses a simple IEEE-compliant reference FPU in HDL
  - Uses high-level HDL constructs: + - loops to count number of zeros
  - Imp: 15000 lines VHDL; Ref-FPU: <700 lines

- Formally compare Ref-FPU vs. real FPU



FPU Complexity Issues

- Certain portions of FPU intractable for formal methods
  - E.g., alignment-shifter, multiplier

- Needed methods to cope with this complexity:
  - Black-box multiplier from cone-of-influence
    - Verified independently using “standard” techniques
    - Multipliers are fairly regular, in contrast to rest of FPU
  - Case-splitting
    - Restrict operands → each subproblem solved very fast
    - Utilize batch runs → subproblems verified in parallel
  - Apply automatic model reduction techniques
    - Redundancy removal, retiming, phase abstraction…
    - These render a combinational problem for each case

FPU Case-Splitting

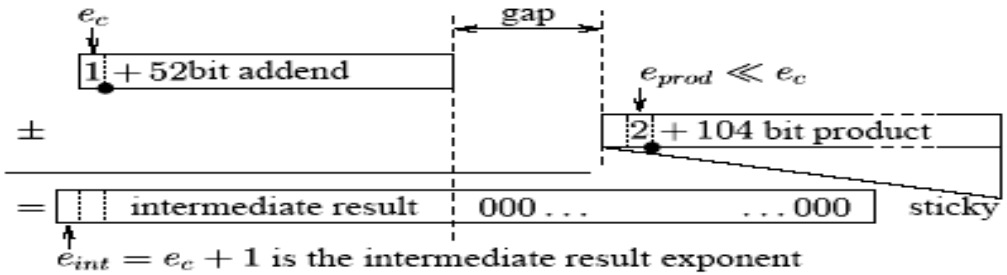
- Four distinct cases distinguished in Ref-FPU
  - Based on difference between product, addend exponent

$\delta = e_{prod} - e_c$ where e_{prod} $(= e_a + e_b - bias)$ is the product exponent and e_c is the addend exponent

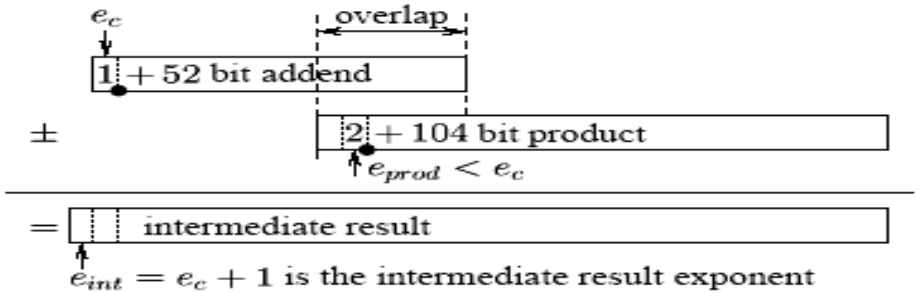
- Case splitting strategy via constraining internal Ref-FPU signals
  - Verification algos implicitly propagate these constraints to real FPU
  - Allows each case to cover large, difficult-to-enumerate set of operands

$$C_\delta := (e_a + e_b - bias = e_c + \delta)$$

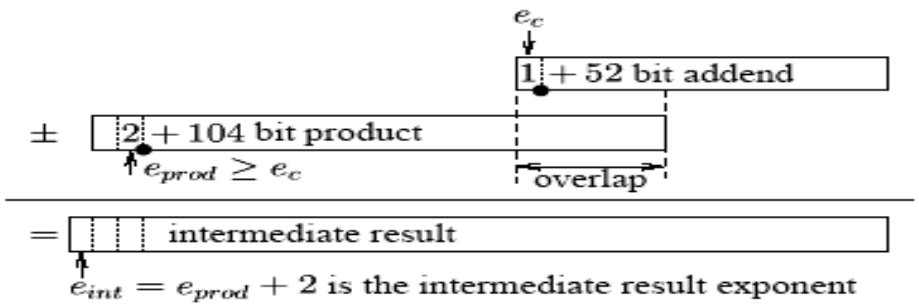
- Disjunction of cases easily provable as a tautology, ensuring completeness



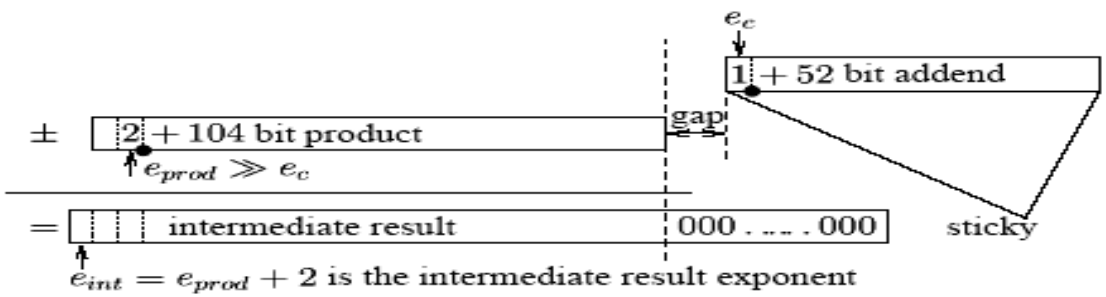
(a) Farout left: the addend is much larger than the product; there is no overlap, the product becomes a sticky bit.



(b) Overlap left: the addend is larger than the product; the product overlaps with the right part of the addend.



(c) Overlap right: the product is larger than the addend; the addend overlaps with the right part of the product.



(d) Farout right: the product is much larger than the addend; the addend becomes a sticky bit.

FPU Normalization Shift Case-splits

- Normalization shifter is used to yield a *normal* result
  - Depends upon # number of leading zeros of intermediate result

- Define a secondary case-split on normalization shift
  - Constraint defined directly on shift-amount signal (sha) of Ref-FPU
  - *Sha* is 7-bit signal (double-precision) to cover all possible shift amounts

$C_{sha} := (sha = X)$ for all 106 possible shift amounts;

$C_{sha/rest} := (sha > 106)$ to cover the remaining cases (trivially discharged)

FPU Results

- Development of methodology required nontrivial trial-and-error to ensure tractability of each proof
  - And some tool tuning…
- Resulting methodology is highly portable
  - ~1 week effort to port to new FPUs
- Numerous bugs flushed out by this process
  - In one case, an incorrect result was flushed out after **billions** of simulation patterns

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
  - Complexities of High-End Processors
  - Algorithmic Methods for Reducing Verification Complexity
  - Processor Verification Case Studies
    - Sequential Equivalence Checking (SEC)
    - Instruction Dispatch Case Study
    - Instruction Fetch-Hang Case Study
    - Floating-Point Unit Verification

- Verification Closure: Coverage Analysis & Integration with Simulation

Load-Store Unit Case Study

- Numerous properties to check of LSU and Memory Infrastructure:
  - Multiprocessor cache coherency properly maintained
  - Correctness of associativity policy
  - Proper address-data correlation and content maintained
  - Parity and data errors properly reported
  - Data prefetching stays within proper page limits
  - …

- In this case study we introduce several Testbench modeling tricks that can be used for such checks

Cache Coherence Case Study

- Cache coherence protocol requires masters to obtain a clean snoop response before initiating a *write*
  - Obtain *Exclusive snoop* to write, clean *snoop* to read
  - Otherwise data consistency will break down

- Mandatory for driver to adhere to protocol, else will spuriously break logic

- Adhering to protocol requires either:
  - Building reference model for each interface, indicating what coherence state it has for each valid address
    - Safe, but dramatically increases Testbench size!
  - Using internal cache state to decide legal responses
    - Not safe: if cache is flawed (or has timing windows due to pipelining), driver may miss bugs or trigger spurious fails

Cache Coherence Case Study

- Trick: check coherence only for one randomly-selected address
  - Reference model becomes very small
- Allow arbitrary activity to be driven to other addresses
  - Will generate illegal stimuli, but cache should still behave properly for checked address
- Other tricks:
  - Parameterize RAM! Caches often are VERY large
  - Can limit # addresses that can be written to, but need to take care that exercise sectoring, *N*-way associativity, …

Associativity Case Study

- N-way associative caches may map *M>N* addresses to *N* locations
  - When loading *N+1'th* address, need to cast a line out
  - *Victim* line often chosen using Least-Recently Used (LRU) algo

- Verify: newly-accessed entry not cast out until every other entry accessed

- Randomly choose an entry *i* to monitor; create a *N-1* wide bitvector
  - When entry *i* accessed, zero the bitvector
  - When entry *j != i* accessed, set bit *j*
  - If entry *i* is cast out, check that bitvector is all 1's

- Weaker pseudo-LRU may only guarantee: no castout until *J accesses*
  - Zero count upon access of entry *i*
  - Increment count upon access of *j != i*
  - Assert counter never increments beyond *J*

Address-Data Consistency

- Many portions of LSU need to nontrivially align data and address
  - Data prefetch, load miss queues: delay between address and data entering logic
    - Many timing windows capable of breaking logic
  - Cache needs to properly assemble *sectors* of data for writes to memory
  - Address translator logic maps *virtual* to *real* addresses

- Can either build reference model tracking what should be transmitted (remembering input stimuli)

- Or – play the trick used on Instr Dispatch example
  - *Encode* information into data

Address-Data Consistency

- Drive data as a function of addr
  - Validate that outgoing addr-data pairs adhere to encoded rule
  - Should trap any improper association and staging of data
- Encode atomicity requirements onto data
  - Tag each cache line sector with specific code, validate upon write
  - Tag each word of quad-word stores with specific code, validate that stores occur atomically and in order
- Encode a parity bit onto driven data slices
  - Can even randomize *odd vs. even* parity
  - Should trap any illegal data sampling
- Drive *poisoned* data values if known that they should not be transmitted

Parity / Error Detection Correctness

- Error code schemes are based upon algorithms:
  - Properly diagnose <*I* bit error code errors, <*J* data bit errors
  - Properly correct <*K* bit data errors
- Often use a reference model based upon error code algorithm
  - Build a Testbench for each type of *injected* error
    - Single-bit data, double-bit data, single-bit error code, …
  - Case-split on *reaction type*
    - Compare logic reaction against expected outcome
- Used to find error detection bugs; improve error detection algorithms
  - Quantify % *N*-bit errors detected using symbolic enumeration
  - Study undetected cases to tighten algorithm

Prefetch Correctness

- Prefetch logic is a performance-enhancing feature
  - Guess addresses likely to be accessed; pull into cache before needed
  - Often use a dynamic scheme of detecting access sequences:
    - Start by fetching one cache line
    - If continued accesses to prefetched stream, start fetching multiple lines

- However, faulty prefetch logic can break functionality
  - Generation of illegal prefetch addresses → checkstop
  - May be responsible for address-data propagation
  - And bad prefetching can easily *hurt* performance

Prefetch Correctness

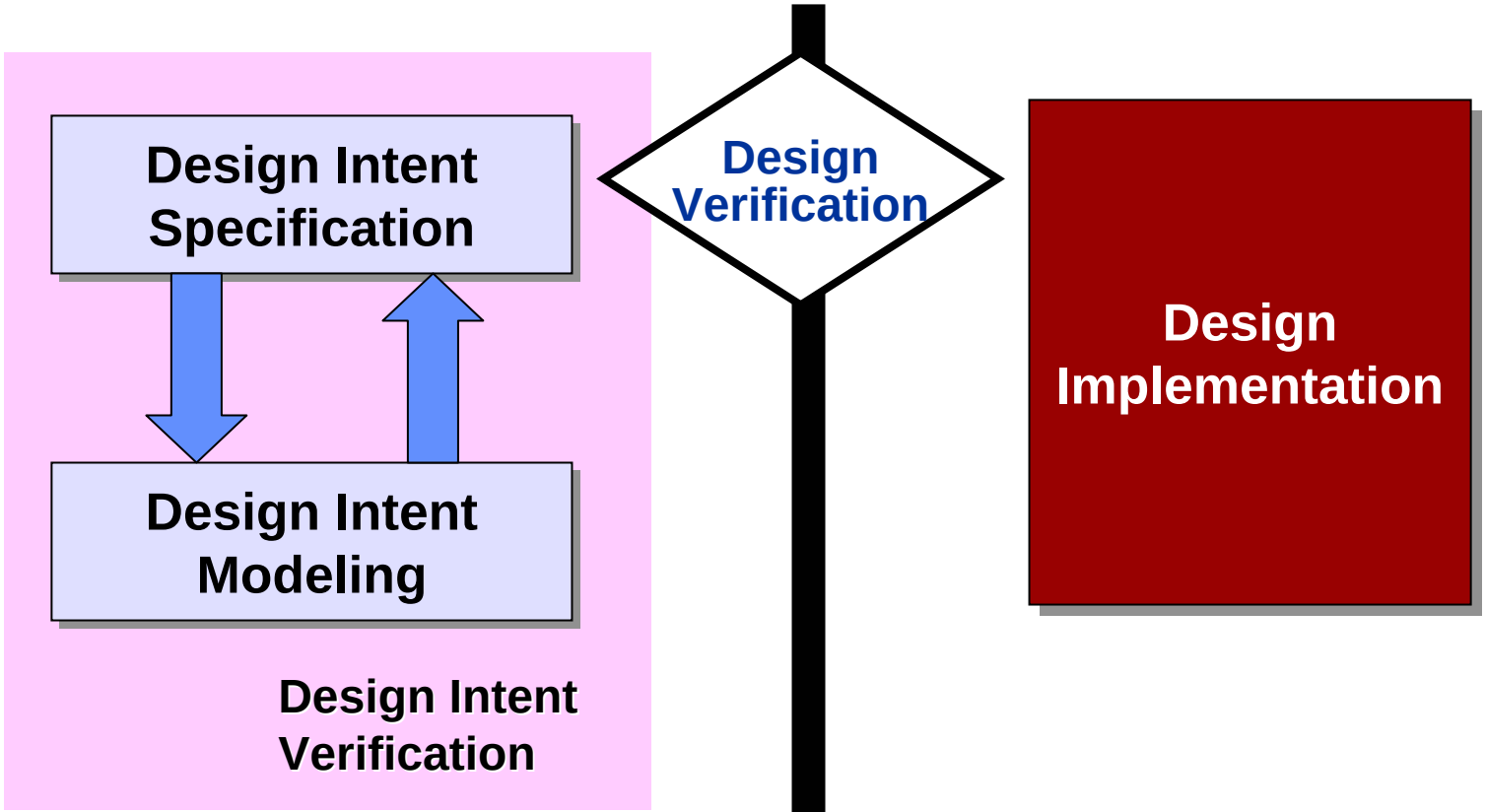
- Generation of illegal prefetch addresses → checkstop
  - Most prefetching is required not to cross address barriers
    - E.g. must be done to same page as actually-accessed addr
  - Can restrict address stream being generated, or monitor addr stream, and validate that prefetch requests stay within same page

- Also wish to verify that prefetch initiates prefetches when it should, does not when it shouldn't
  - Often done using a reference model or set of properties to encode specific prefetching algorithm

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation
  - Verification as Coverage Analysis
  - Design Intent Coverage and Specification Refinement
  - Reasoning about Specifications
  - Have I Written Enough Properties?
  - Property Directed Simulation Games
  - The Integrated Picture

Design Intent Verification

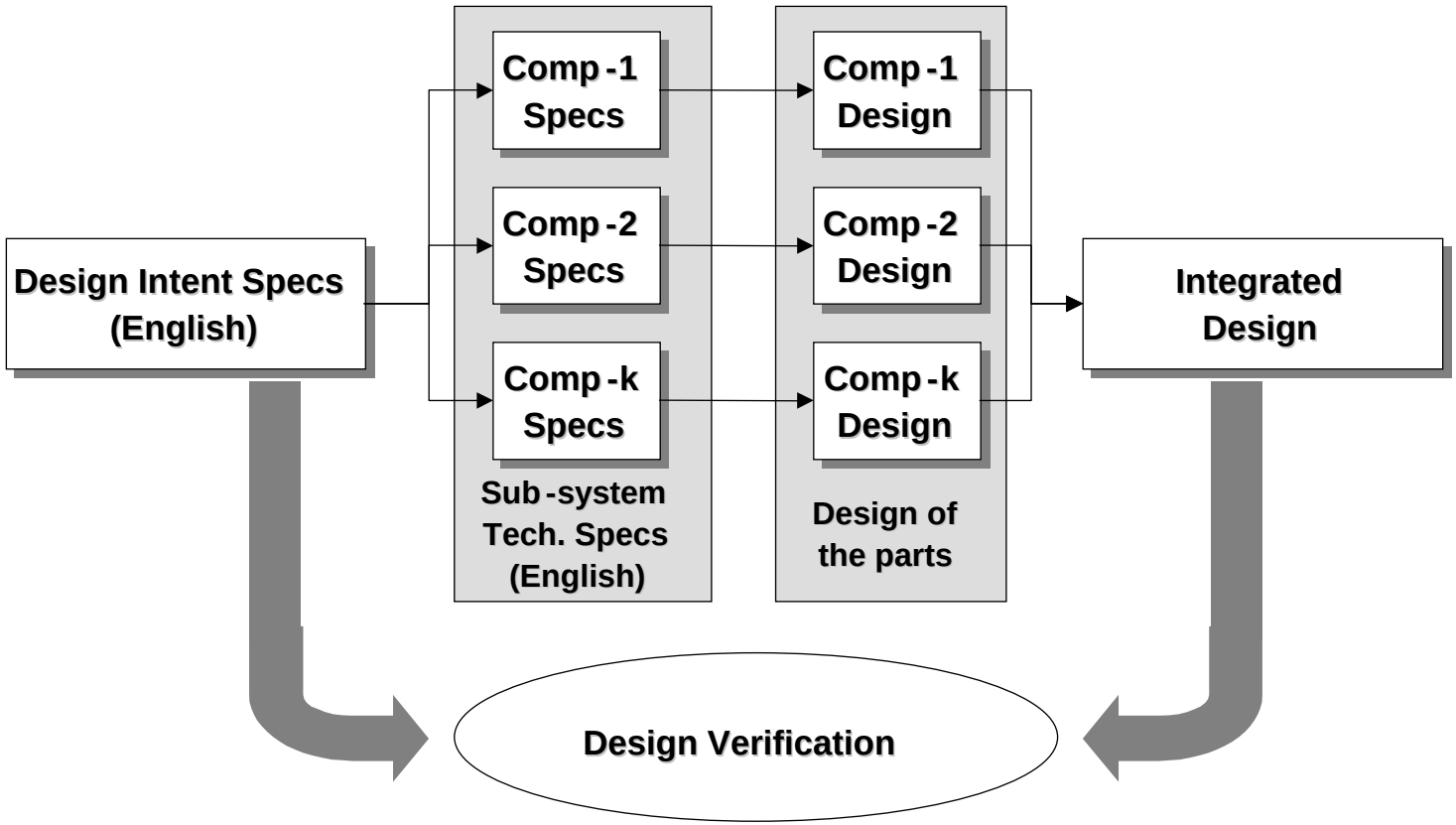


This step is becoming very important in practice – why?

What is design intent verification?

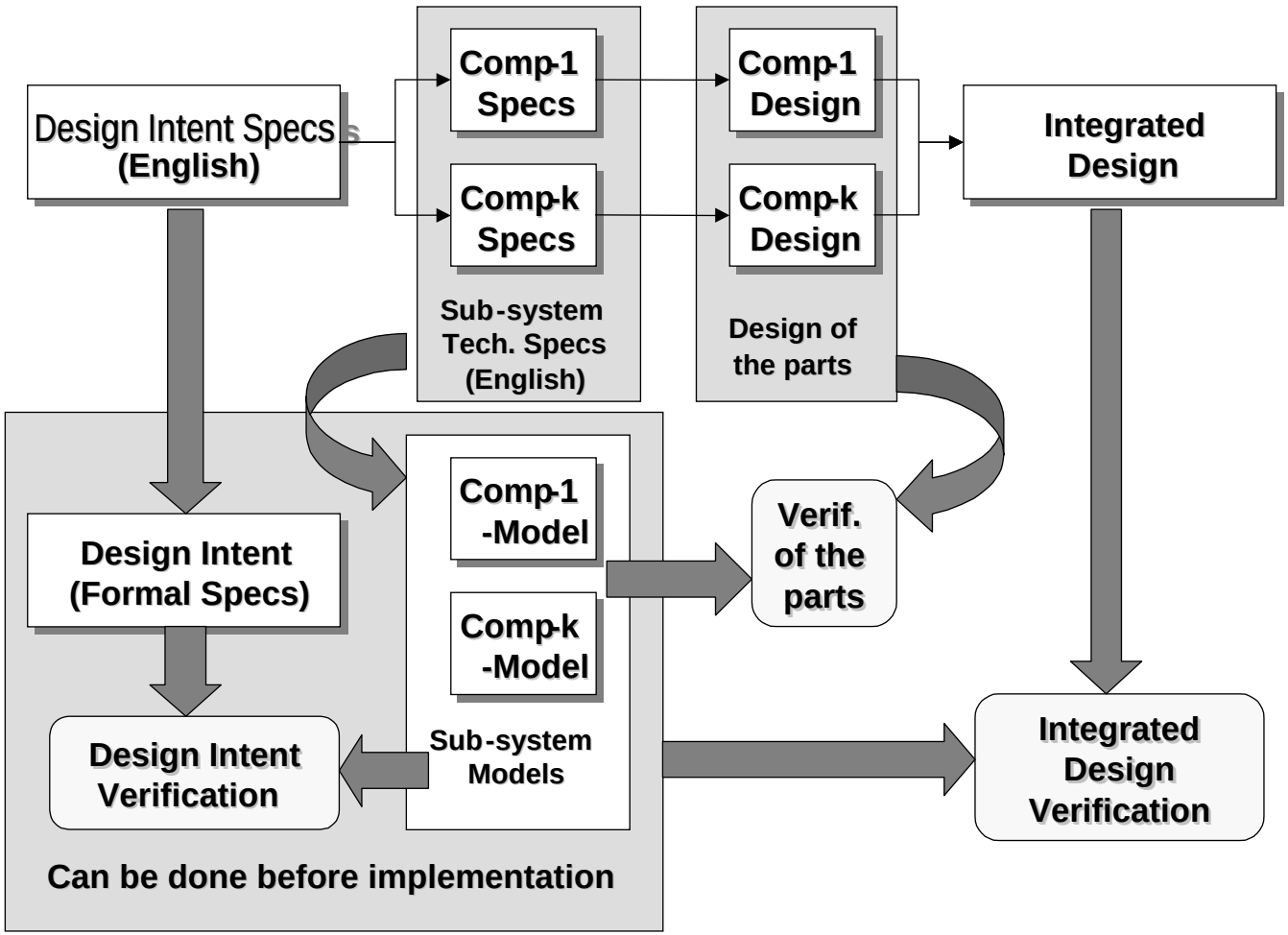
- Design Intent
  - A set of abstract high-level global safety and liveness requirements
- Design Intent Modeling
  - Developing a set of high-level component models that mimic the expected behavior of the components
- Design Intent Verification
  - The models taken together should satisfy the design intent

Traditional: *Without intent verification*



This verification task is large and complex

Emerging: *With intent verification*



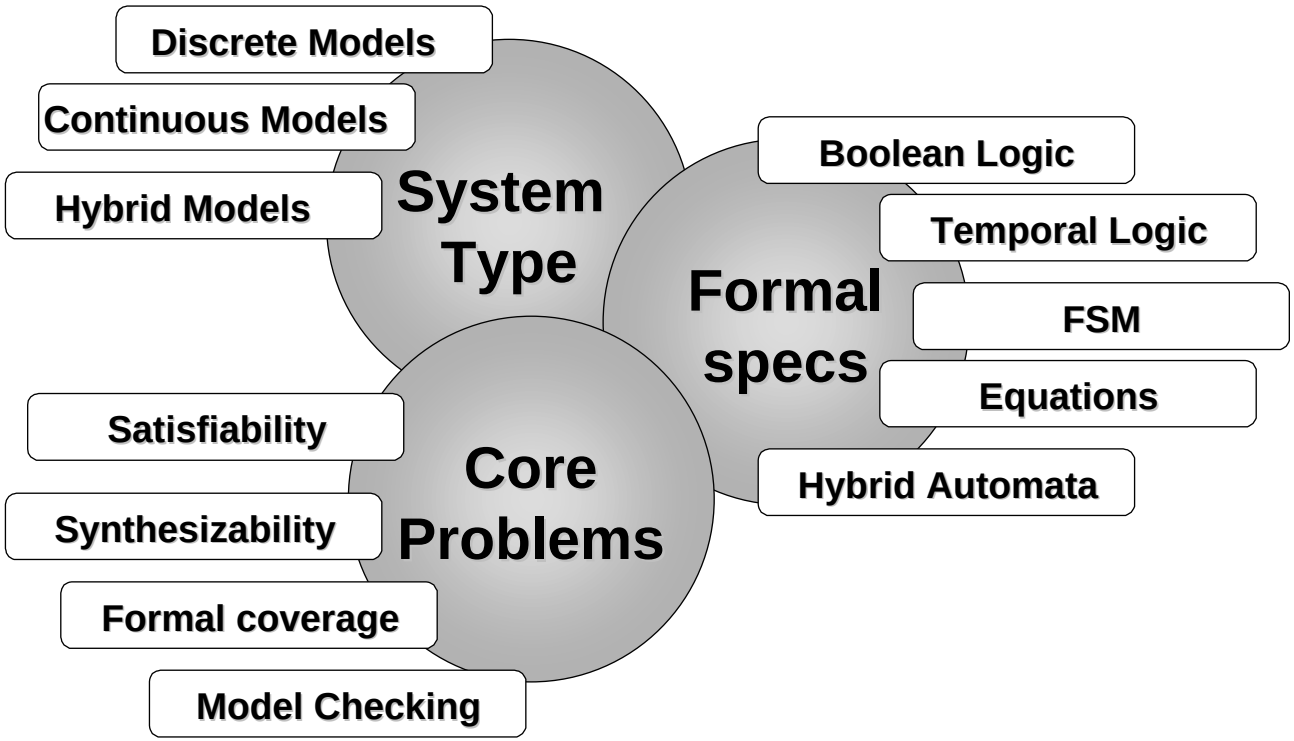
Intent Verification – *Everywhere?*

- Digital design verification
  - Micro-architectural properties (SVA / PSL)
  - RTL properties (SVA / PSL), RTL blocks
- Mixed-signal design verification
  - Integrated power mgmt chips (LDOs, buck converters, battery charger)
  - Modeling options include VerilogA, VerilogAMS
- Embedded systems
  - Safety critical properties
  - Modeling options include Matlab, UML statecharts

Design Intent Verification – *Why?*

- Practical considerations
  - Components are out-sourced, and integrated into the design at the end (sometimes as black-boxes)
- Helps in developing the component specs before implementation
- Helps in developing acceptance tests for the components and verifying the integrated design

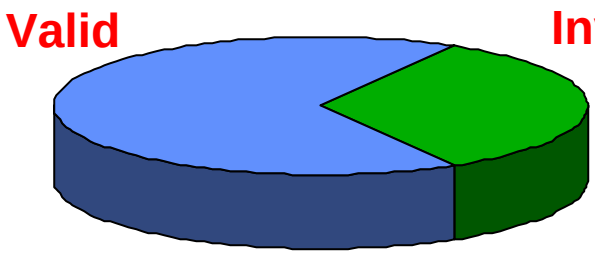
Dimensions


Discrete Models
Continuous Models
Hybrid Models
System Type
Boolean Logic
Temporal Logic
FSM
Equations
Hybrid Automata
Formal specs
Satisfiability
Synthesizability
Formal coverage
Model Checking
Core Problems

Agenda

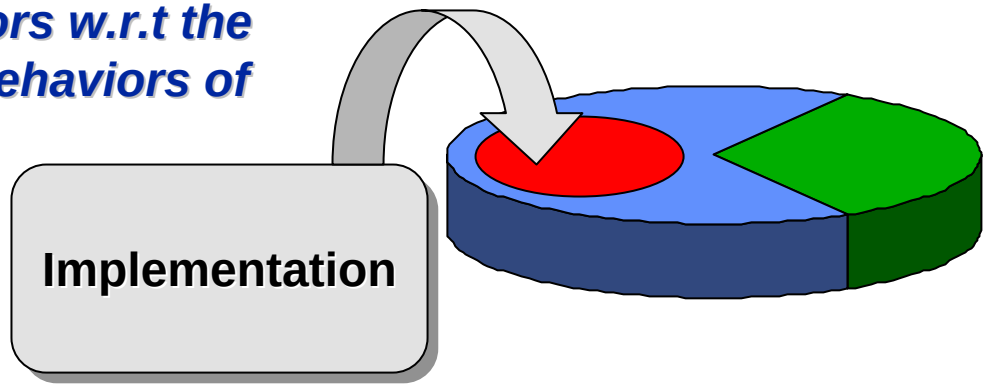
- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation
  - Design Intent Verification
  - Design Intent Coverage and Specification Refinement
  - Reasoning about Specifications
  - Have I Written Enough Properties?
  - Property Directed Simulation Games
  - The Integrated Picture

Verification is all about coverage



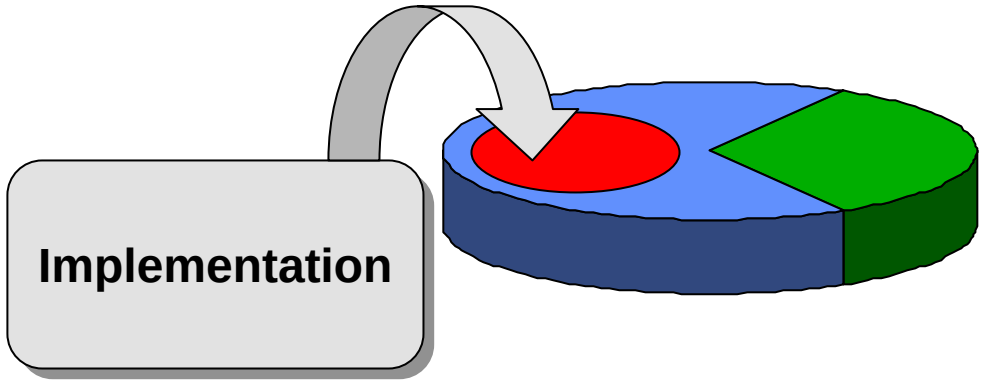
The role of a specification is to partition the set of possible behaviors into valid and invalid behaviors

The role of verification is to check whether the valid behaviors w.r.t the specification covers all behaviors of the implementation



In other words, the set of behaviors not exhibited by the implementation covers the invalid behaviors of the specification

Verification as coverage



If:

- $\mathcal{R}$ ***is a function that captures exactly the set of all behaviors of the given implementation, and***
- $\mathcal{A}$ ***is the specification,***

Then the verification problem is to check for the validity of $\mathcal{R} \Rightarrow \mathcal{A}$

A bug is a witness for $\mathcal{R} \wedge \neg \mathcal{A}$

Verification Methodologies

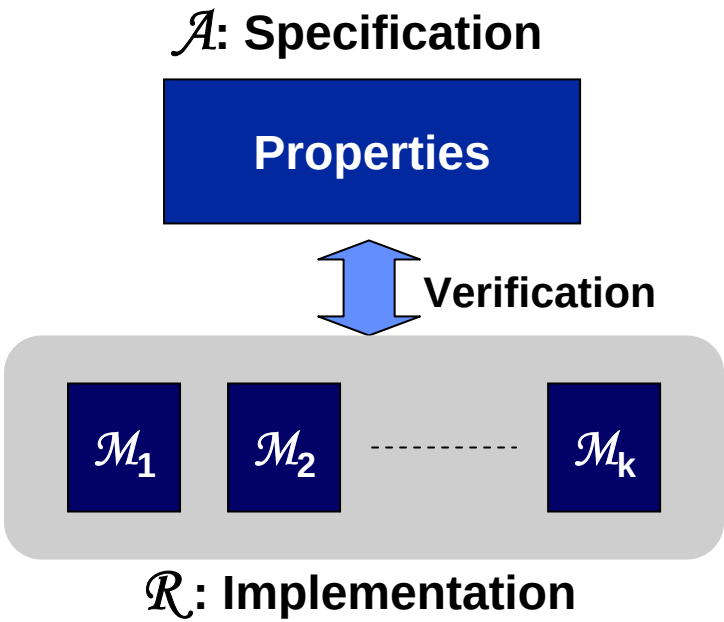
A bug is a trace that acts as a witness for $\mathcal{R} \wedge \neg \mathcal{A}$

- Simulation and dynamic ABV: *Enumerate traces treating* $\mathcal{R}$ *as an executable black box and search for* $\neg \mathcal{A}$ *on each trace*

- Model checking: *Represent* $\mathcal{R}$ *as a state machine, represent* $\neg \mathcal{A}$ *as another automaton and check whether the product of the two machines is empty*

- Design intent coverage: *Represent* $\mathcal{R}$ *as a collection of properties that are guaranteed by the components, and check the satisfiability of* $\mathcal{R} \wedge \neg \mathcal{A}$

Compositional Verification & Coverage

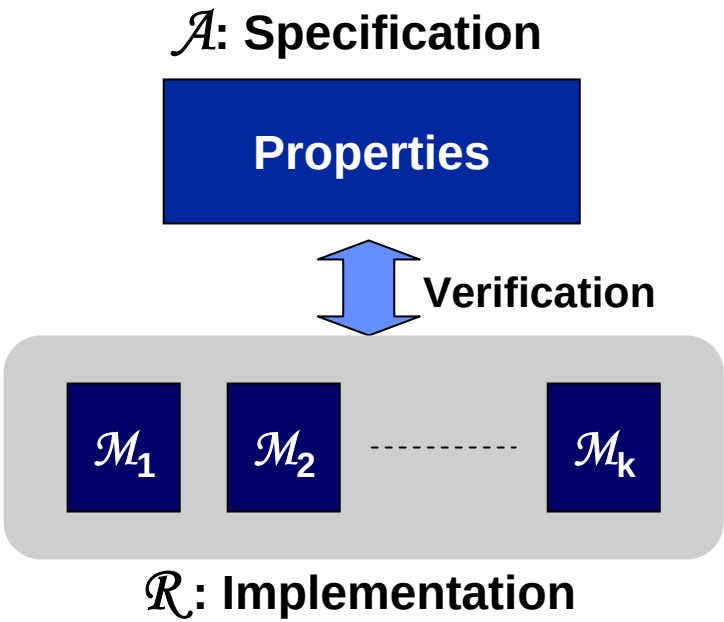


Model checking: *$\mathcal{R}$ is a set of logic blocks or state machines*

Simulation: *$\mathcal{R}$ is a set of executable black boxes*

Design intent coverage: *$\mathcal{R}$ is a set of local properties over the $\mathcal{M}_i$s*

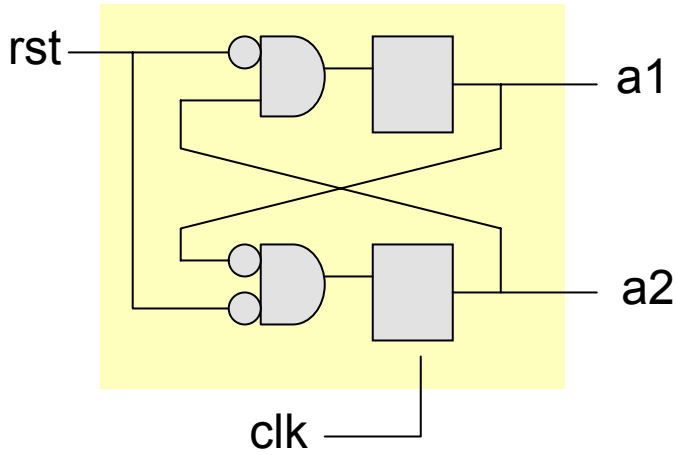
Two key problems



What should be our approach when $\mathcal{R}$ consists of state machines for some components, properties for some components and executable black boxes for the rest?

Problem-1: ***Developing a unified model for coverage analysis***
Problem-2: ***Property slicing / Specification refinement***

Model checking as coverage analysis

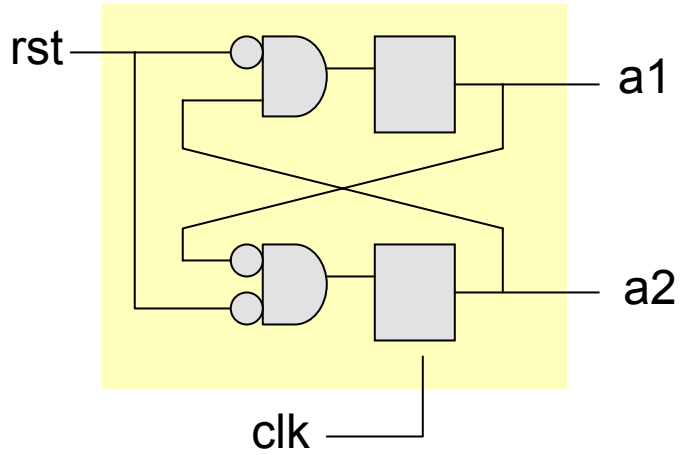



```
module GrayCounter(a1, a2, rst)
input rst;
reg a1, a2;

always @ (posedge clk)
begin
    a1 <= a2 & ~rst;
    a2 <= ~a1 & ~rst;
end
endmodule
```

Sample property: ***If the counter is not reset, then the next value of the counter differs from the present value by exactly one bit.***

$$\mathcal{A} \equiv \quad \mathbf{G(\neg rst \Rightarrow (\, a1 \oplus Xa1\,) \oplus (\, a2 \oplus Xa2\,))}$$

Model checking as coverage analysis




```
module GrayCounter(a1, a2, rst)
input rst;
reg a1, a2;

always @ (posedge clk)
begin
    a1 <= a2 & ~rst;
    a2 <= ~a1 & ~rst;
end
endmodule
```

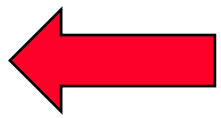
$\mathcal{R} \equiv$ $G((a2 \wedge \neg rst \Leftrightarrow Xa1) \wedge (\neg a1 \wedge \neg rst \Leftrightarrow Xa2))$

$\mathcal{A} \equiv$ $G(\neg rst \Rightarrow (a1 \oplus Xa1) \oplus (a2 \oplus Xa2))$

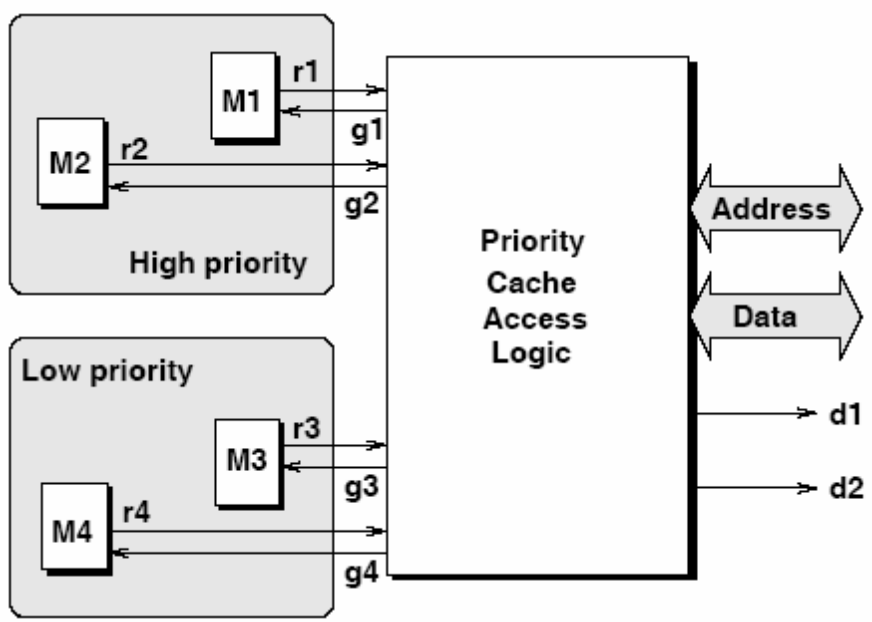
Model checking $\mathcal{A}$ on module GrayCounter is equivalent to checking the validity of $\mathcal{R} \Rightarrow \mathcal{A}$ or checking the satisfiability of $\mathcal{R} \wedge \neg \mathcal{A}$

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation
  - Design Intent Verification
  - Verification as Coverage Analysis
  - Reasoning about Specifications
  - Have I Written Enough Properties?
  - Property Directed Simulation Games
  - The Integrated Picture



Priority Cache Access



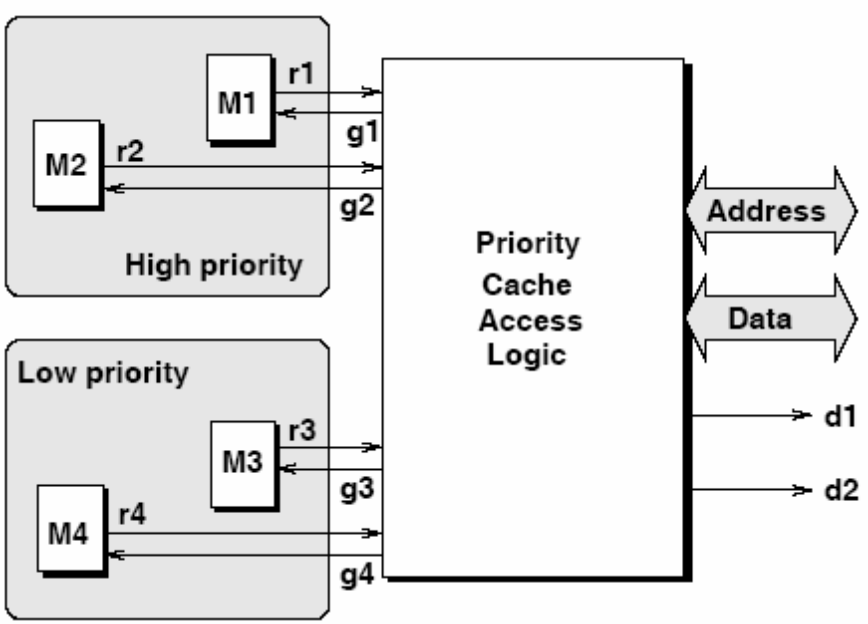
The block arbitrates between r1 and r2 to assert g1 or g2, and between r3 and r4 to assert g3 or g4

The lines d1/d2 indicate whether the page requested by the high / low priority device is available in the cache

- **If there is a cache hit then d1 is asserted within two cycles**
- **If there is a cache miss then d1 is asserted after fetching the page from memory**

[Source: *A Roadmap for Formal Property Verification, Springer, 2006*]

Architectural Property

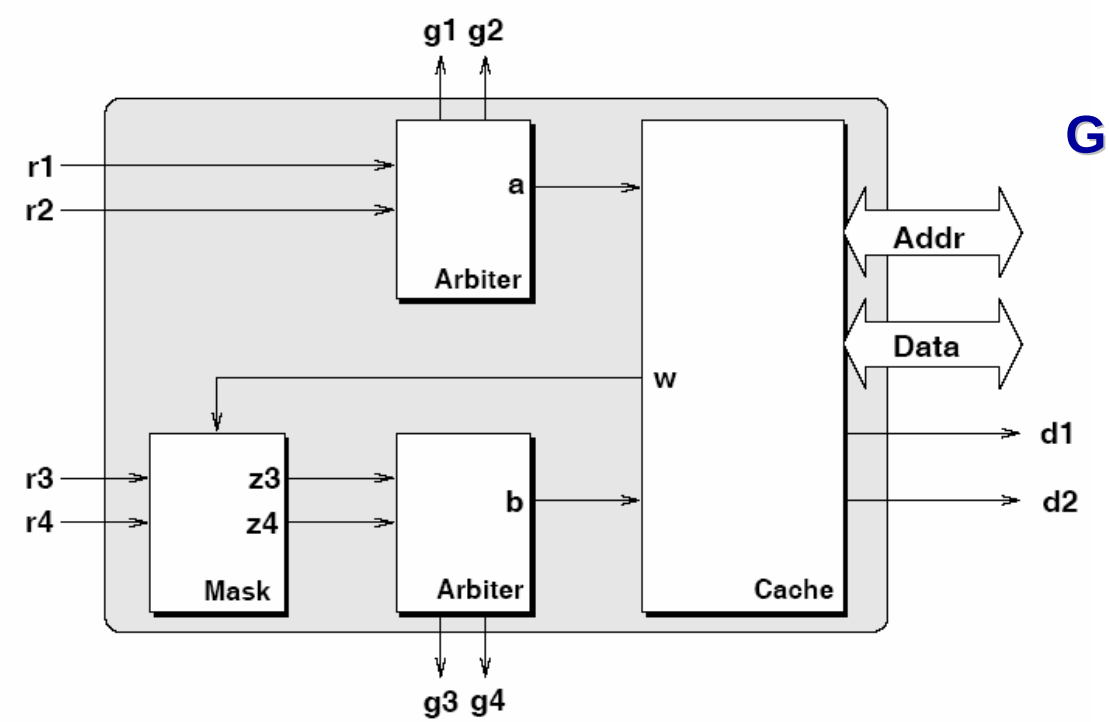


M1 and M2 have higher priority over M3 and M4

A page requested by M1/M2 is either served within two cycles (cache hit) or served when the page is ready. In the latter case, no page should be served to the low priority devices in between.

$$G[\, r1 \; V \; r2 \Rightarrow XX[\, d1 \; V \; X(\, \neg d2 \; U \; d1 \,)\,]\,]$$

One possible architecture

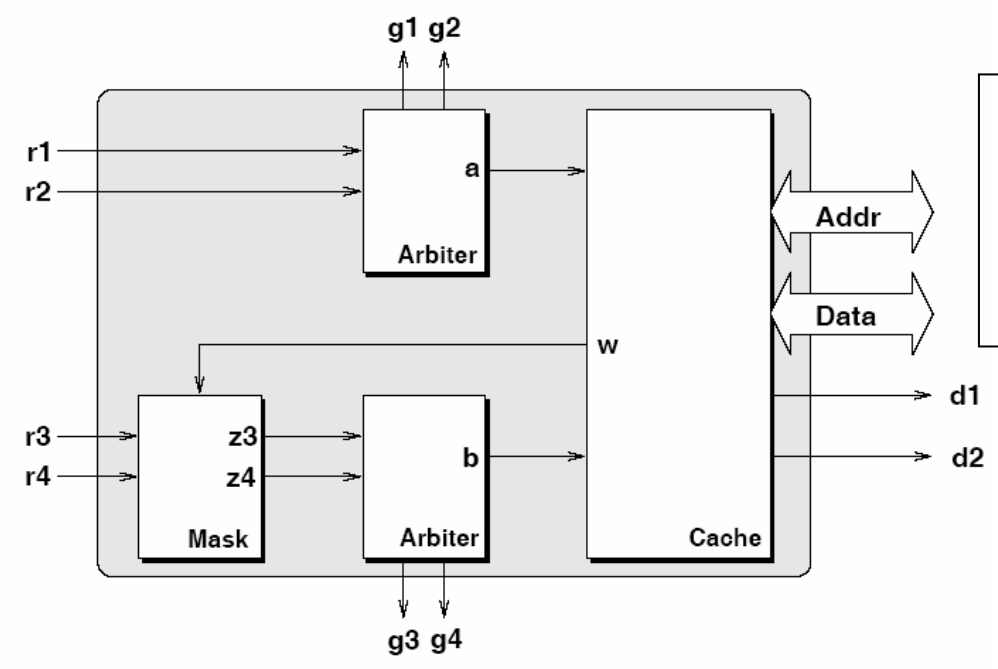


$$G[\, r1 \vee r2 \Rightarrow XX[\, d1 \vee X(\, \neg d2 \; U \; d1 \,)]]$$

What does one do if we cannot verify the property directly due to capacity limitations?

Write properties on individual blocks that together *prove* the architectural property

Developing block specs



Target:

A1: G[r1 V r2 ⇒ XX[d1 V X(¬d2 U d1)]]

Arbiter:

R1: G[r1 V r2 ⇔ Xa]

R2: G[z3 V z4 ⇔ Xb]

Cache Block:

R3: G[a ⇒ X[w U d1]]

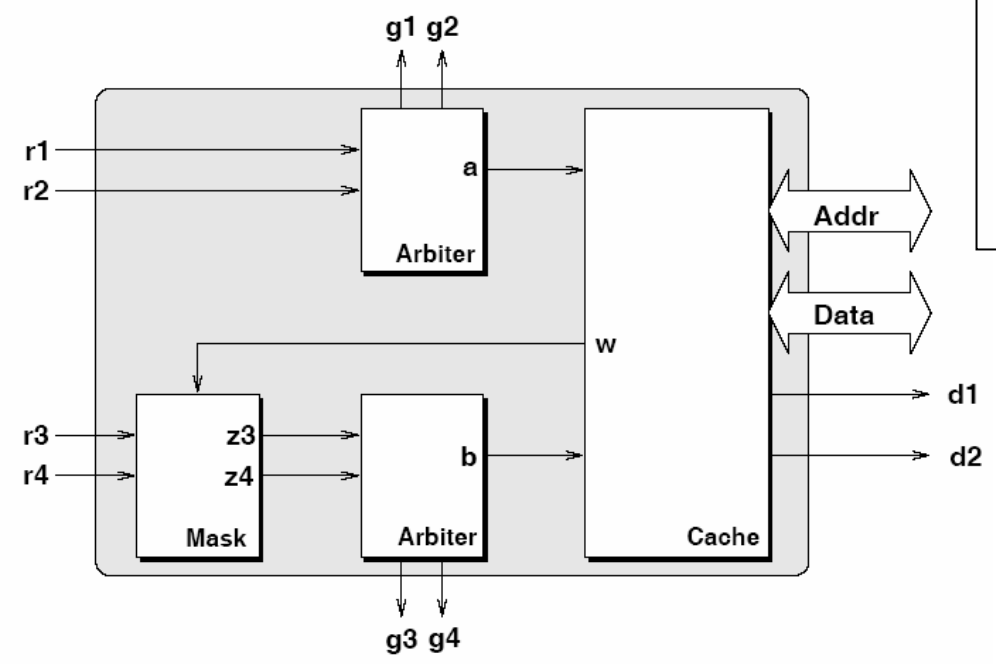
R4: G[¬b ⇒ ¬X d2]

Mask:

R5: G[r3 ∧ ¬w ⇔ z3]

R6: G[r4 ∧ ¬w ⇔ z4]

Design Intent Coverage



Arch. Specs:

A1: G[r1 V r2 ⇒ XX[d1 V X(¬d2 U d1)]]

RTL Specs:

R1: G[r1 V r2 ⇔ Xa]

R2: G[z3 V z4 ⇔ Xb]

R3: G[a ⇒ X[w U d1]]

R4: G[¬b ⇒ ¬X d2]

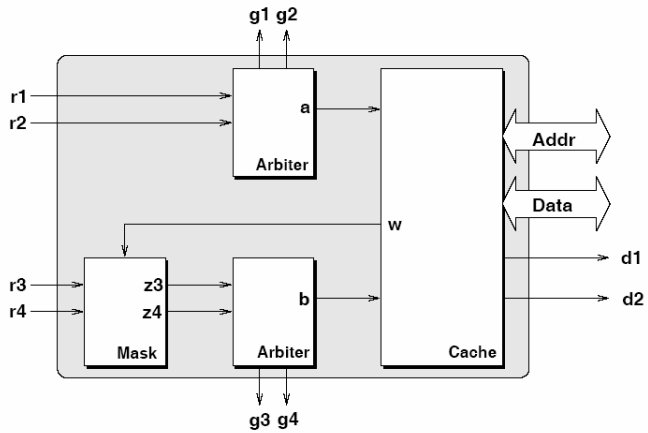
R5: G[r3 ∧ ¬w ⇔ z3]

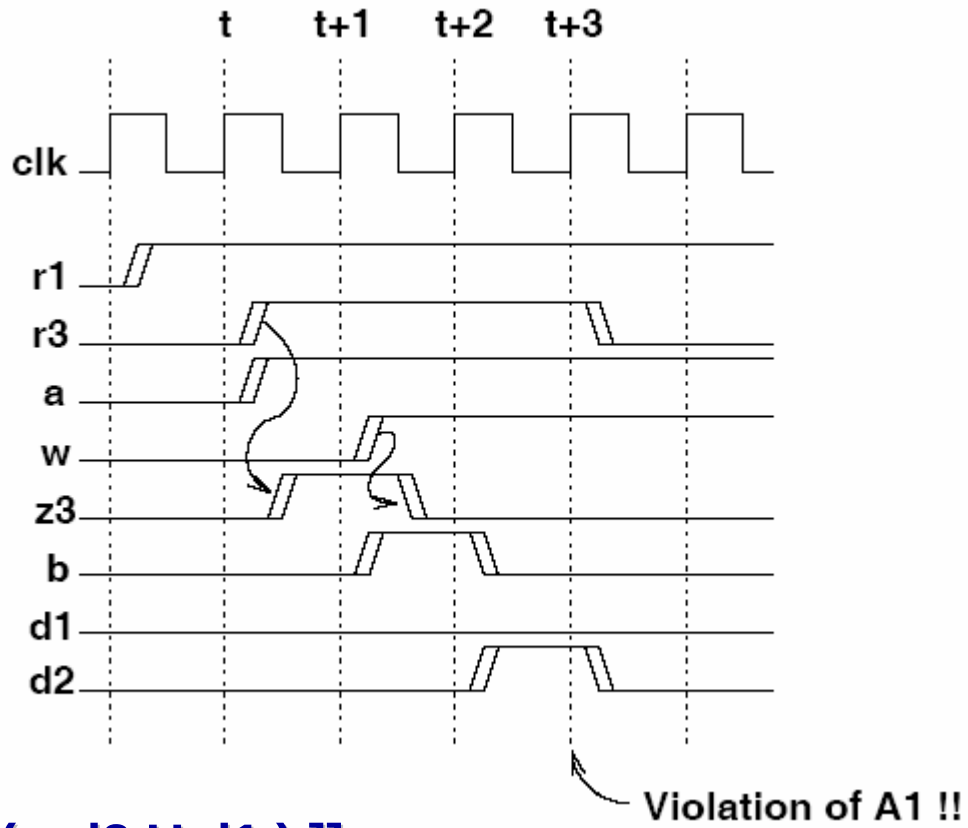
R6: G[r4 ∧ ¬w ⇔ z4]

Is it possible to have an implementation that satisfies R1, …, R6, but refutes A1?

… if not, then all invalid behaviors are not covered by the RTL specs, and we need to add more RTL properties

In this case, we have a gap!!



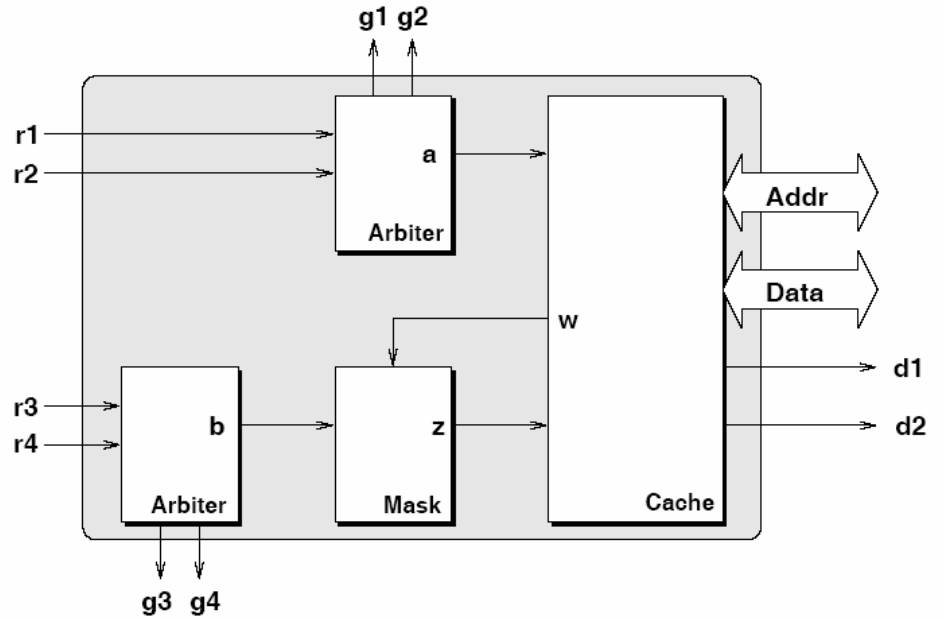


A1: $G[\, r1 \vee r2 \Rightarrow XX[\, d1 \vee X(\, \neg d2 \, U \, d1\,)\,]]$

Gap: $G[\, r1 \vee r2 \Rightarrow XX[\, (\neg d1 \wedge Xd1) \vee d1 \vee X(\, \neg d2 \, U \, d1\,)\,]]$

[Source: *A Roadmap for Formal Property Verification, Springer, 2006*]

The Correct Architecture



Target:

A1: G[r1 V r2 ⇒ XX[d1 V X(¬d2 U d1)]]

Arbiter:

R1: G[r1 V r2 ⇔ Xa]

R2: G[r3 V r4 ⇔ Xb]

Cache Block:

R3: G[a ⇒ X[w U d1]]

R4: G[¬z ⇒ ¬X d2]

Mask:

R5: G[b ∧ ¬w ⇔ z]

… this time R1,…,R5 covers A1

Intent Coverage Problem: *Formally…*

- Given:
  - *Architectural specification* $\mathcal{A}$, consisting of a set of temporal properties over a set $\mathcal{AP}_{\mathcal{A}}$ of Boolean signals
  - *RTL specification* $\mathcal{R}$, consisting of a set of temporal properties over a set $\mathcal{AP}_{\mathcal{R}}$ of Boolean signals, such that $\mathcal{AP}_{\mathcal{A}} \subseteq \mathcal{AP}_{\mathcal{R}}$

[Primary Coverage Goal] Does $\mathcal{R}$ cover $\mathcal{A}$?

[Specification Refinement] *Refinement of* $\mathcal{A}$ *to derive a property that captures the behaviors not covered by* $\mathcal{R}$.

Specification Refinement

- Consider the coverage of $\mathcal{A}$ by $\mathcal{R}$:

  $\mathcal{A}$: $G((\neg r_2) \Rightarrow X\, g_1)$

  $\mathcal{R}$: $G((r_1 \wedge \neg\, r_2) \Rightarrow X\, g_1)$

- $\mathcal{A} \vee \neg \mathcal{R}$ can be represented as:

  (a) $G(r_1 \vee r_2 \vee X\, g_1)$

  (b) $G((\neg X\, g_1 \wedge \neg r_1) \Rightarrow r_2)$

  (c) $G((\neg\, r_1 \wedge \neg r_2) \Rightarrow X\, g_1)$

  (c) is visually closest to $\mathcal{A}$ – our goal is to show the coverage gap as (c).

Spec. Refinement by Relaxation

Arch. Spec: $G((r_2 \wedge z) \Rightarrow X((g_2 \wedge \neg g_1)\ U\ \neg r_2))$

RTL Specs: $G((r_1 \wedge \neg r_2) \Leftrightarrow X\ g_1)$

$GF(\neg r_2)$

The RTL specs do not cover the architectural specs, but we can decompose the architectural specs as:

A_x: $G((r_2 \wedge z) \Rightarrow X(g_2\ U\ \neg r_2))$

A_y: $G((r_2 \wedge z) \Rightarrow X(\neg g_1\ U\ \neg r_2))$

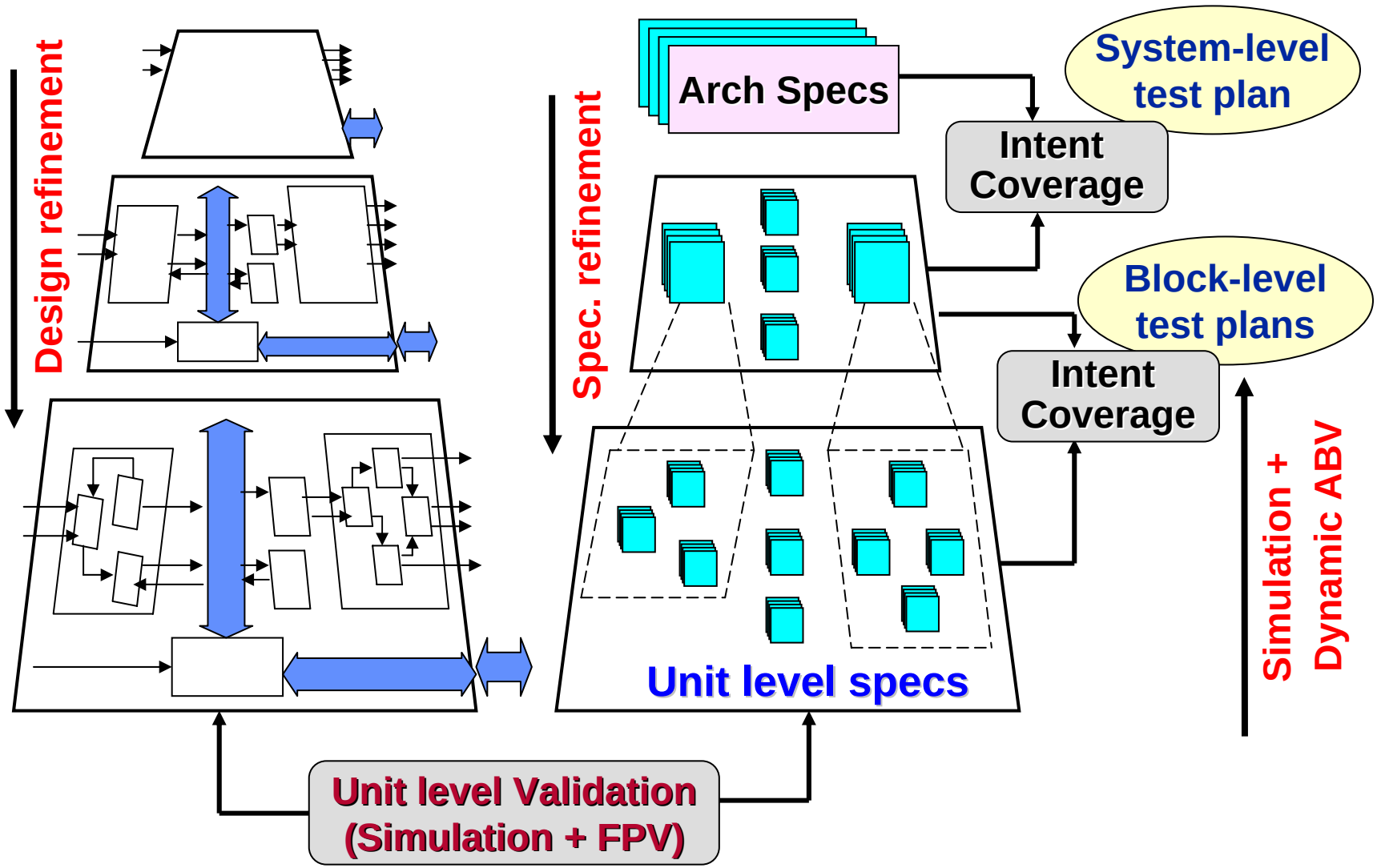
- A_y is covered by the RTL specs
- A_x represents the coverage gap more accurately

The SpecMatcher Tool

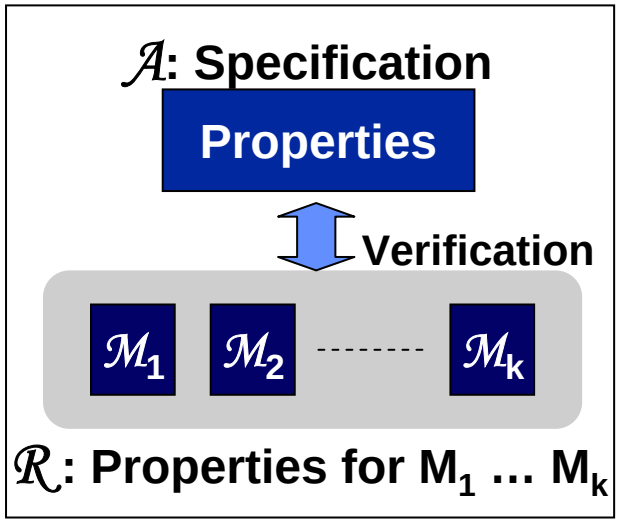
- Compares temporal specifications (in LTL)
- Reports the coverage gap in terms of *structure preserving* properties.
- Two key algorithms:
  - Push_Terms: Computes the bounded temporal terms in the gap, $\mathcal{A} \vee \neg \mathcal{R}$, and *pushes* these terms into the syntactic structure of the architectural properties.
  - Relax_ArchSpec: Systematic weakening of architectural properties having unbounded temporal operators (such as G, F, and U), and isolating the coverage gap.

[Source: *A Roadmap for Formal Property Verification, Springer, 2006*]

The Integrated Flow



Intent Coverage and Model Checking



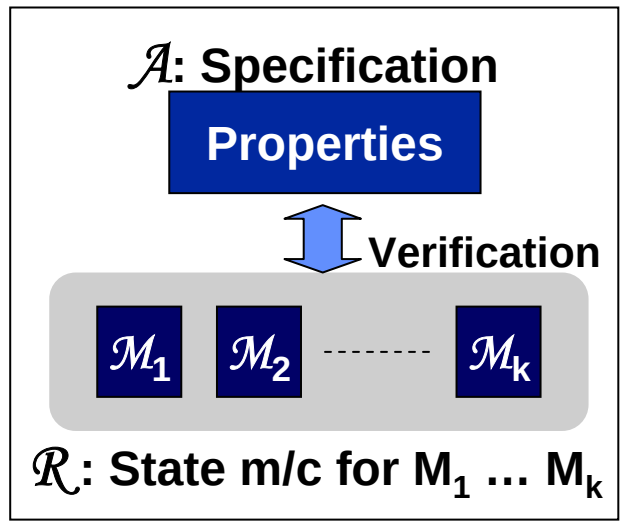
Pure Design Intent Coverage

Advantages:
- *Scalable*
- *Can be done top down, before implementation*

Disadvantages:
- *Requires more user involvement*

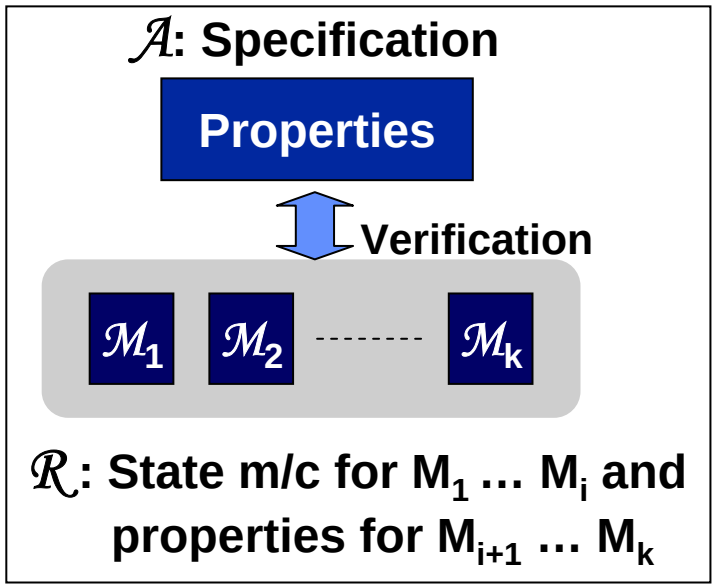
Advantages:
- *Less user involvement*

Disadvantages:
- *Capacity limitations*



Pure Model Checking

Between Intent Coverage and Model Checking



Hybrid Design Intent Coverage

Approach-1:

- *Find the gap, T, between $\mathcal{A}$ and the properties of M_{i+1} … M_k*
- *Model check T on M_1 … M_i*

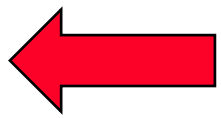
Approach-2:

- *Translate M_1 … M_i into the property domain*
- *Use pure design intent coverage*

[Source: *Das et. al.* What lies between Design Intent Coverage and Model Checking? DATE 2006]

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation
  - Design Intent Verification
  - Verification as Coverage Analysis
  - Design Intent Coverage and Specification Refinement
  - Have I Written Enough Properties?
  - Property Directed Simulation Games
  - The Integrated Picture



Consistency and Completeness

- Verification: $\mathcal{R} \Rightarrow \mathcal{A}$
- Consistency issues:
  - Verification is vacuous when $\mathcal{R}$ is unsatisfiable
  - Verification is vacuous when $\mathcal{A}$ is valid
- Completeness issues:
  - Verification closure depends on the completeness of $\mathcal{A}$ – *have I written enough properties?*

Formal Consistency Analysis – *Why?*

- Are my properties correct?
  - Debugging formal specifications can be quite hard
  - Coding errors – new languages, alien semantics
  - Logical errors
- Am I checking the right property on the right design?
  - A typical BUS protocol consists of:
    - Properties over individual components: master, slave, arbiter
    - Global properties
  - In the absence of proper assume constraints, checking global properties on individual components can lead to realizability problems.
- Reasoning about formal specifications
  - Logical implication
  - Coverage analysis

Verification is Logical Consistency

- Verification is mostly about checking logical implication
- Model checking:
  - Does the product of the component state machines logically imply each of the formal properties?
- Design intent coverage:
  - Do the properties of the component modules together imply the architectural properties of the design?
- Logical implication ≡ Satisfiability / Validity

Satisfiability and Realizability forms the basis of reasoning about specifications

Unsatisfiable Specification

- When the master is not in the IDLE or WAIT states, the request line, *req*, should be kept high
- The master lowers the request line, *req*, sometime

```
`define IDLE 3'b000
`define WAIT 3'b000

property ReqHighDuringTransfer;
        @ (posedge clk)
        (state != 'IDLE || state != 'WAIT) |-> req ;
endproperty

property ReqIsSometimesLow;
        @ (posedge clk)
        ##[0:$] !req ;
endproperty
```

The Correct Specification

- When the master is not in the IDLE or WAIT states, the request line, *req*, should be kept high
- The master lowers the request line, *req*, sometime

```
`define IDLE 3'b000
`define WAIT 3'b000

property ReqHighDuringTransfer;
        @ (posedge clk)
        (state != 'IDLE && state != 'WAIT) |-> req ;
endproperty


property ReqIsSometimesLow;
        @ (posedge clk)
        ##[0:$] !req ;
endproperty
```

Vacuity

- When the *gnt* signal is high, the master should not be in the IDLE or WAIT states

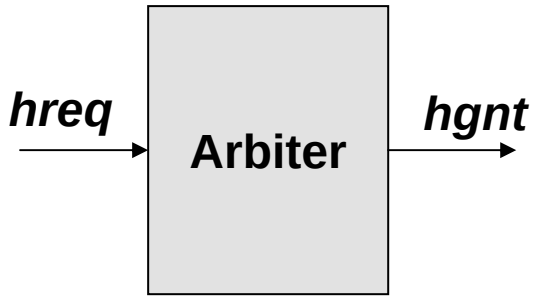
```
property UseBusWhenGranted;
        @ (posedge clk)
        gnt |-> (state != ‘IDLE || state != ‘WAIT) ;
endproperty
```

- Same mistake – *this time the property will always be true*
- *This gives us a false sense of security !!*

Realizability of Open System Specs

- Whenever the high-priority request, *hreq*, arrives, the grant, *hgnt*, is given for one cycle

```
property HighPriorityGrant ;
        @ (posedge clk)
        hreq |-> ##1 hgnt ##1 !hgnt ;
endproperty
```



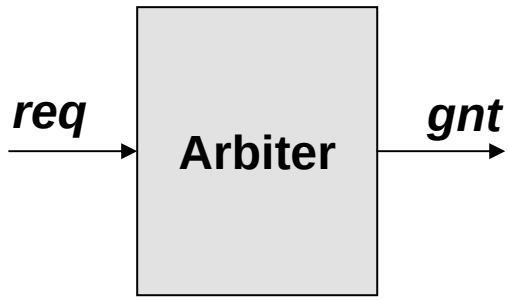
- The property cannot be satisfied if we have *hreq* in consecutive cycles
- The property is satisfiable -- consider all traces where *hreq* is not asserted in consecutive cycles
- The property is *unrealizable* because *hreq* is an input signal
  - *It will become realizable under the assumption that hreq never arrives in consecutive cycles*

[Ref: *Pnueli, Rosner,* ACM POPL, 1989]

Realizability = Satisfiability ∀ Inputs?

- Not quite. Consider the following properties:
  - Each request is eventually granted
  - The request line is lowered one cycle after the grant

```
property Gnt ;
        @ (posedge clk)
        req |-> ##[1:$] gnt ;
endproperty
property LowReqAfterGnt ;
        @ (posedge clk)
        gnt |-> ##1 !req ;
endproperty
```



- The second property can be satisfied if we know the future inputs !!

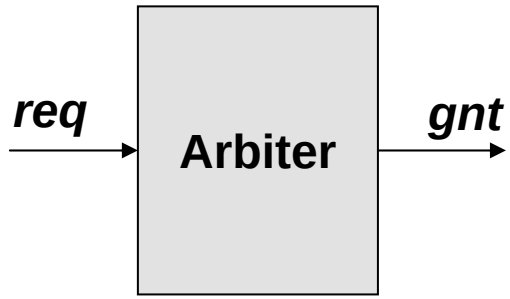
[Ref: *Pnueli*, *Rosner*, ACM POPL, 1989]

Receptiveness

- Consider the property:

  A request is always lowered after the grant is asserted.

```
property LowReqAfterGnt ;
        @ (posedge clk)
        gnt |-> ##1 !req ;
endproperty
```

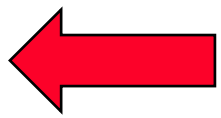


- The arbiter can realize this property by never asserting g1.
  - This is an un-receptive property – *why?*
- A module must have the freedom of choosing its outputs as long as it does not refute the property

[Ref: *Dill*, MIT Press, 1989]

Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation
  - Design Intent Verification
  - Verification as Coverage Analysis
  - Design Intent Coverage and Specification Refinement
  - Reasoning about Specifications
  - Property Directed Simulation Games
  - The Integrated Picture



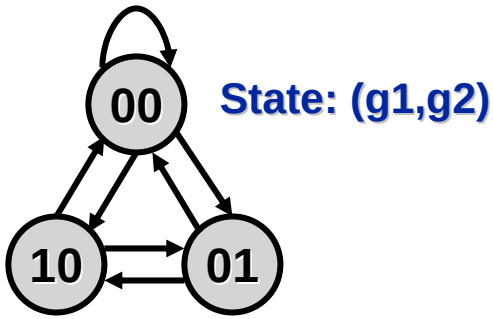
Have I written enough properties?

- Completeness can be assured against some functional coverage goal
- Paradox:
  - If I had a formal definition of the coverage goal, then that itself could become the formal specification!!
- Solution:
  - Evaluate the formal property specification with some structural coverage metric
  - The structural coverage metric should be such that:
    - Low coverage indicates gaps in the specification, and I need more properties
    - High coverage does not necessarily mean that I have enough properties

Types of Coverage Approaches

- Mutation-based Approaches
  - A given implementation is used as the reference
- Fault-based Approaches
  - A given fault model is used as the reference
- Design Intent Coverage
  - A higher level specification is used as the reference

Mutation-based Coverage



Abstract FSM of a Round-robin Arbiter

Specification:

P1: g1 is never asserted in consecutive cycles

```
property NoConsecutiveG1;
    @ (posedge clk) g1 |-> ##1 !g1 ;
endproperty
```

P2: g2 is never asserted in consecutive cycles

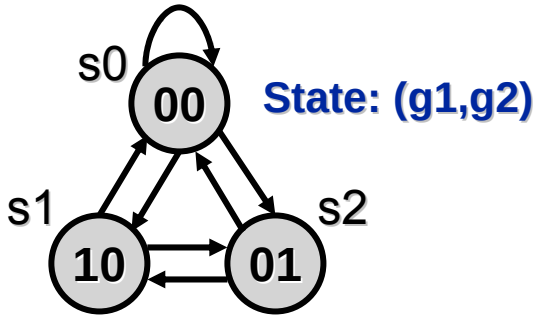
Approaches:

Falsity coverage: ***Mutate the FSM and check whether the truth of any property changes.***

Vacuity coverage: ***Mutate the FSM and check whether any property becomes vacuous.***

[Ref: *Hoskote et.al.* DAC 1999, *Chockler et.al.* CAV 2001]

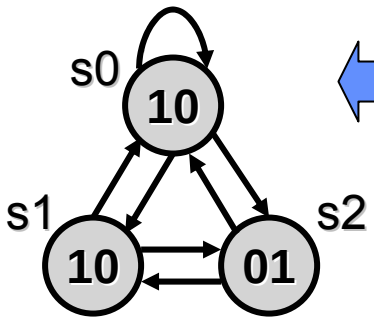
Mutation-based Falsity Coverage



Abstract FSM of a Round-robin Arbiter

Specification:

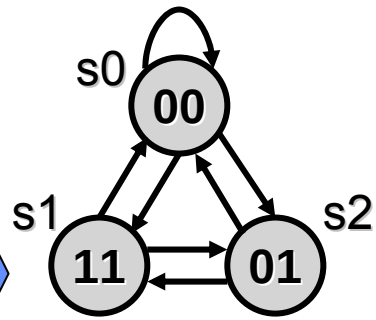
P1: g1 is never asserted in consecutive cycles
P2: g2 is never asserted in consecutive cycles



Mutant-1
(g1 flipped at s0)

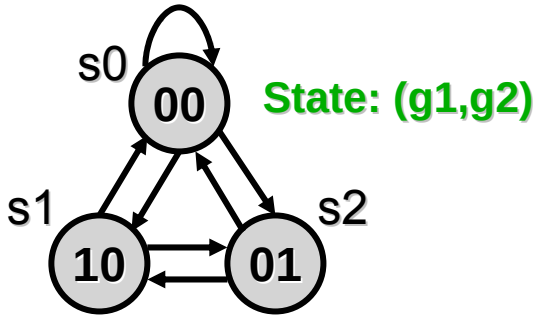
P1 fails.
⇒ The value of g1 at s0 is covered

P2 fails.
⇒ The value of g2 at s1 is covered



Mutant-2
(g2 flipped at s1)

Mutation-based Vacuity Coverage



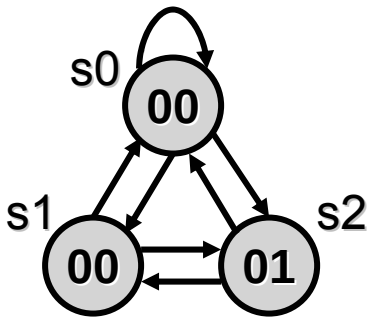
Abstract FSM of a Round-robin Arbiter

Specification:

P1: g1 is never asserted in consecutive cycles

```
property NoConsecutiveG1;
    @ (posedge clk) g1 |-> ##1 !g1 ;
endproperty
```

P2: g2 is never asserted in consecutive cycles



Mutant-3
(g1 flipped at s1)

Falsity coverage

No property fails.
⇒ The value of g1 at s1 is not covered

Vacuity coverage

P1 is satisfied vacuously.
⇒ The value of g1 at s1 is covered

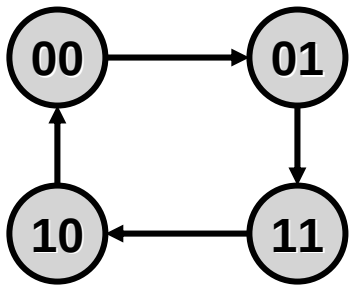
What does mutation coverage mean?

- If mutations on many parts of the implementation do not affect the truth of the properties, then it is likely that our specification does not cover those behaviors that are exhibited by those parts of the implementation
  - Extensions to transition coverage, path coverage, etc
  - Extensions to simulation coverage metrics, such as code coverage (mutations on the HDL code) and circuit coverage (toggle coverage on latches and signals)
- Does this mean that 100% coverage $\Rightarrow$ we have written enough properties?

High Coverage is not good enough

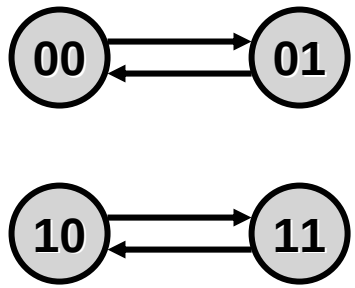
Gray Counter: Correct!!

```
module GrayCounter( x1, x2 )
reg x1, x2;

always @ (posedge clk)
begin
  x1 <= x2;
  x2 <= ~x1;
end
endmodule
```



Gray Counter: Incorrect!!

```
module GrayCounter( x1, x2 )
reg x1, x2;

always @ (posedge clk)
begin
  x1 <= x1;
  x2 <= ~x2;
end
endmodule
```



Property P: ***The next value of the counter differs from the present value in exactly one bit*** **(100% state coverage, but …)**

[Source: *A Roadmap for Formal Property Verification, Springer, 2006*]

Fault-based Coverage Analysis

- Core Idea: Inject a fault into the specification and test whether the specification remains satisfiable / realizable
- Fault Model
  - Stuck-at faults on one or more signals
  - Possible counter-example scenarios
- Good for a first-cut check on the first formal specification
  - Low coverage means we need more properties
  - High coverage does not necessarily mean we have sufficient properties

[Ref: *A Roadmap for Formal Property Verification, Springer, 2006*]

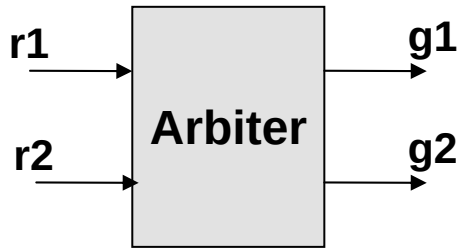
Stuck-at Fault Coverage

- Output Fault Coverage:
  - A stuck-at fault on a non-input y is covered if there exists some scenario where the specification forces y to take the opposite value
- Input Fault Coverage:
  - A stuck-at fault on a input x is covered if we cannot realize the specification without reading that input x

[Ref: *Das, et.al.*, VLSI 2005]

Example: Output Fault Coverage

Memory arbiter: mem-arbiter(input r_1, r_2 ; output g_1, g_2)



Priority of g1: $G((r1 \wedge r2) \rightarrow X\ g1)$

Mutex: $G(\neg g1 \vee \neg g2)$

- g1 s-a-0 is *directly covered* (input: r1=r2=1)
- g2 s-a-1 is *indirectly covered* – it implies g1 s-a-0, which is directly covered
- g2 s-a-0 is *not covered* – we can have a valid implementation that never asserts g2 !!
  - add a property for the scenario where g2 has to be high
    $G((\neg r1 \wedge r2) \rightarrow X\ g2)$
  - now, g2 s-a-0 is directly covered and g1 s-a-1 indirectly covered

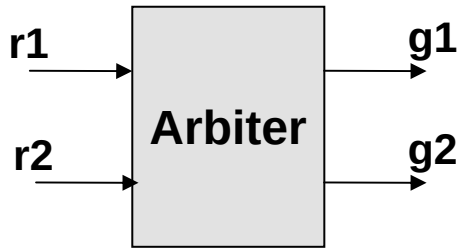
Example: Input Fault Coverage

Memory arbiter: mem-arbiter(input r_1, r_2 ; output g_1, g_2)

Priority of g1: G ((r1 ∧ r2) → X g1)

No starvation: G ((¬r1 ∧ r2) → X g2)

Mutex: G (¬g1 ∨ ¬g2)

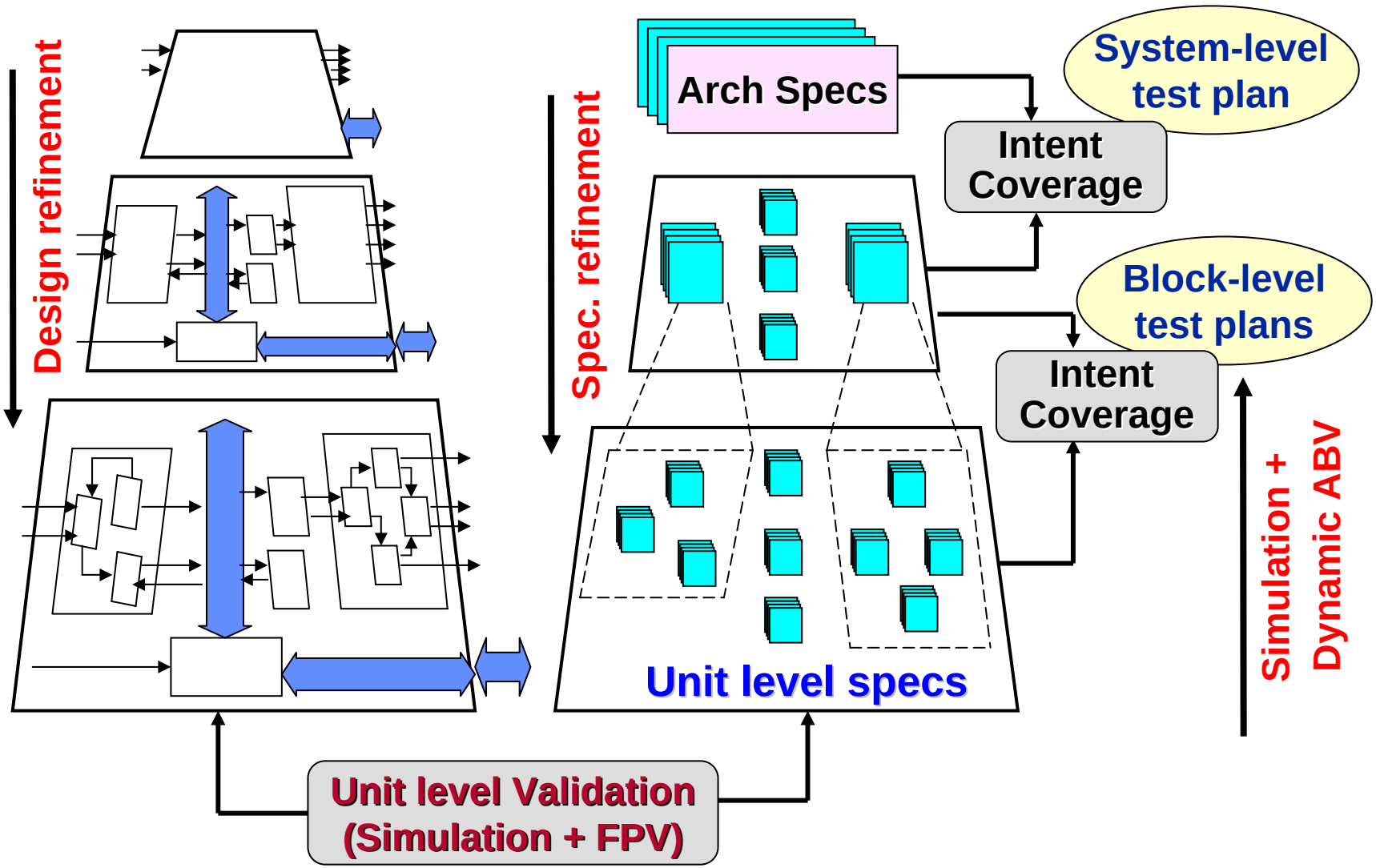


- r1 s-a-0 and s-a-1 are *covered*
- r2 s-a-0 is *covered*
- r2 s-a-1 is *not covered* – we can realize the specs without reading r2 (always assume that it is high)
  - Modified specs (substituting r2=1) covers original specs:
    G (r1 → X g1), G (¬r1 → X g2), G (¬g1 ∨ ¬g2)
  - Add a property for some scenario where r2 is low
    G (¬r2 → X (¬g2))
  - Substituting r2=1 fails to cover the above property. Hence r2 s-a-1 is covered.

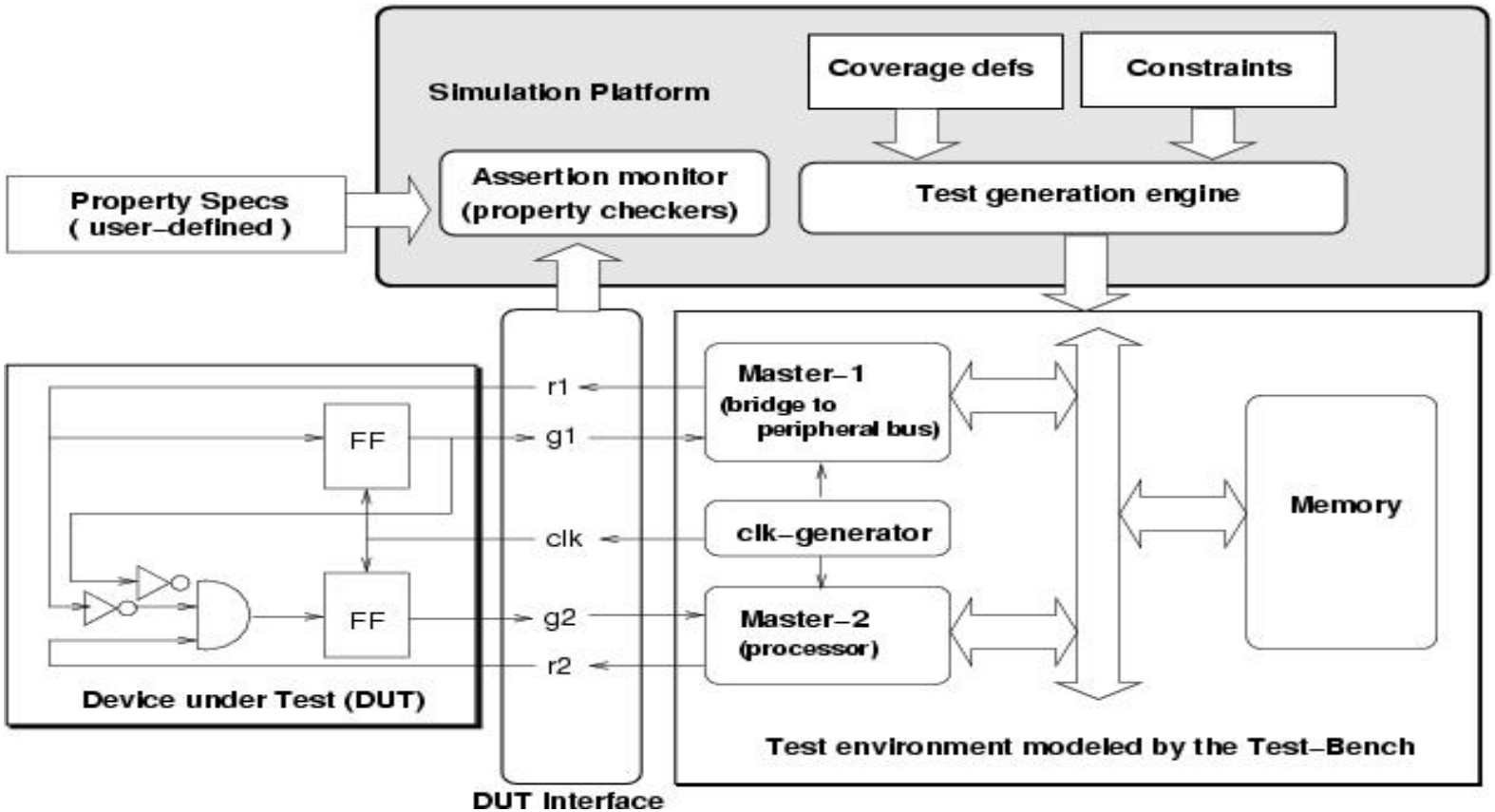
Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation
  - Design Intent Verification
  - Verification as Coverage Analysis
  - Design Intent Coverage and Specification Refinement
  - Reasoning about Specifications
  - Have I Written Enough Properties?
  - The Integrated Picture

The Integrated Flow (Recap)



Dynamic Property Verification (DPV)



[Source: *A Roadmap for Formal Property Verification, Springer, 2006*]

The notion of context

- Two types of properties:
  - Invariants – properties that must hold always
    - *Example:* An arbiter must never assert two grants at the same time (mutex)
  - Context sensitive properties
    - Example: In a burst mode transfer, addresses wrap around the 2KB address boundaries
- To verify the second property we must reach a burst transfer
  - In all other scenarios, the property is vacuous

Assertion Coverage

- Assertion coverage attempts to determine whether the simulation has covered scenarios where the assertion was checked non-vacuously
  - How do we check the vacuity?
  - Many definitions:
    - Implication vacuity
    - Explicit specification of coverage goals – such as cover properties in SVA
    - Gaming definitions

Implication vacuity

- Property: *If the request r is asserted, then the grant g must be asserted in the next two cycles, unless r is lowered in between.*

```
property P;
  @ (posedge clk)
  r |-> ##1 (g or (!g && !r) or ((!g && r) ##1 g) ;
endproperty
```

- Whenever *r* is asserted, implication vacuity will report non-vacuous interpretation

- If *r* is asserted and the DUT does not assert *g* in the next cycle, then we should drive *r* again to check the remaining part of the property
  - Otherwise, the real intent of the property is not checked

Property-driven test generation?

- It's a game …

  Player-1: *The Test Bench (TB)*
  - Drives the input signals in each round

  Player-2: *The Design Under Test (DUT)*
  - Produces the output signals in each round

- *We play for the test bench …*

- Two types of games – same players, but the winning conditions are different
  - Vacuity games
  - Realizability games

[Ref: *Banerjee, et.al.*, DAC 2006]

Vacuity Games

- Winning condition defined in terms of a formal property, P

- In any round of the game, the inputs written by Player-1 (TB) is *vacuous* iff P is satisfied under these inputs regardless of the values of the outputs in that round
  - **Player-1 (Test bench)**
    - **It loses if in any round it writes vacuous inputs**
    - **It wins if in any round it writes non-vacuous inputs and yet the property is satisfied or refuted**
  - **Player-2 (DUT) loses when Player-1 wins, and wins when Player-1 loses**

- The values written in a round re-defines the property for the next round

[Ref: *Banerjee, et.al.*, DAC 2006]

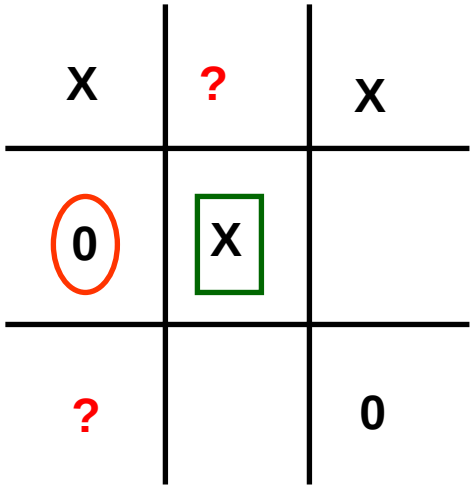
Realizability Games

- Problem with unreceptive specifications
  - In some round of the game, the property for the next round may become unrealizable, but satisfiable
  - This means that Player-1 (TB) has a winning strategy from that round – if it uses that strategy then the property for some future round will become unsatisfiable
  - This strategy is to be demonstrated by a realizability game

- Winning conditions (P is unrealizable)
  - Player-1 (Test bench)
    - Wins in a round if the property is refuted (becomes unsat)
    - Loses if the property for the next round becomes realizable
  - Player-2 (DUT) loses when Player-1 wins, and wins otherwise

[Ref: *Banerjee, et.al.*, DAC 2006]

Example – *tic-tac-toe*

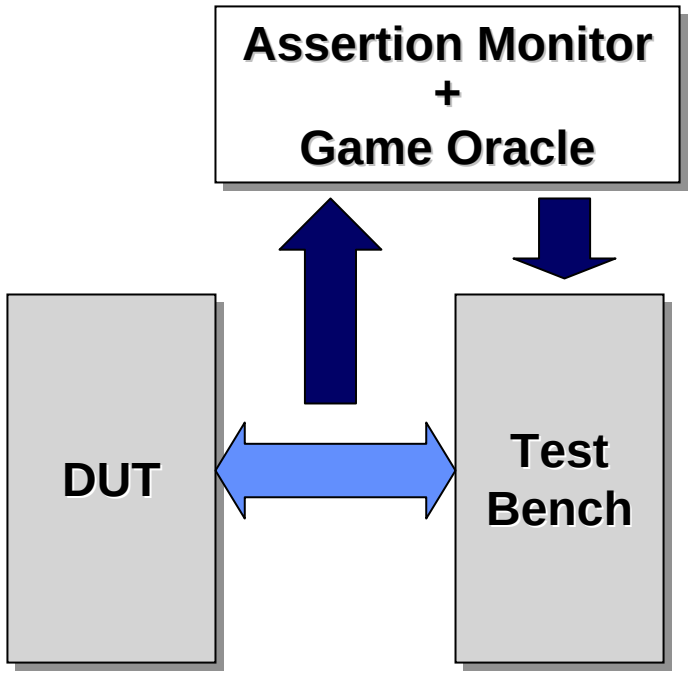


X : Environment

0 : DUT

- Open Game at start: DUT has a strategy to win
  - ✓ Specification is realizable
- DUT makes a mistake: Occupies row 2, column 1
  - ❖ Specification is unrealizable
- Intelligent Test-bench: forces DUT to defeat
  - ➢ Crucial Move: Occupying central square

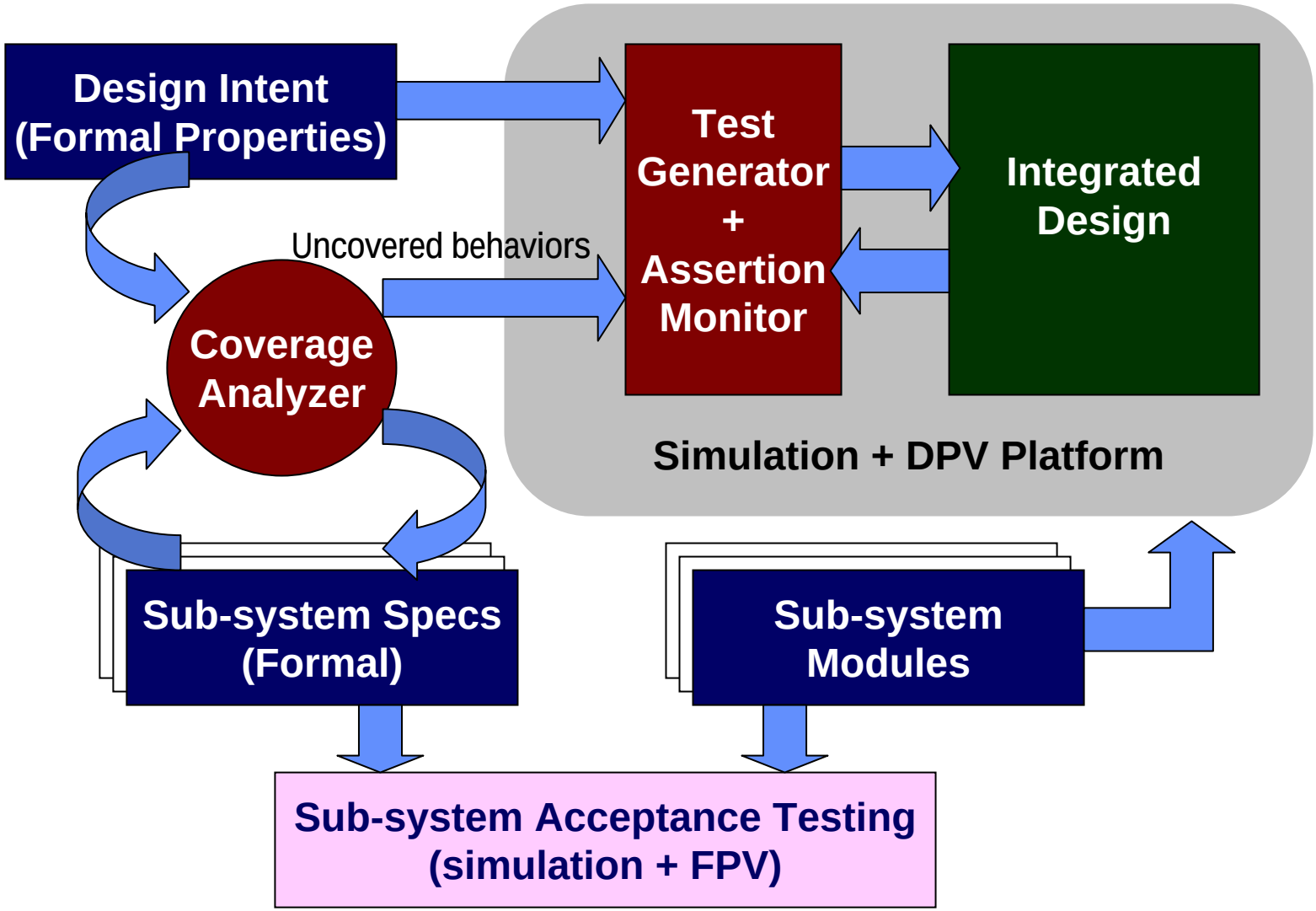
Integration into Test Environment



Agenda

- Introduction to Formal Verification
- Case Studies from TI: Protocol & Control Logic Verification
- Case Studies from IBM: Formal Processor Verification
- Verification Closure: Coverage Analysis & Integration with Simulation
  - Design Intent Verification
  - Verification as Coverage Analysis
  - Design Intent Coverage and Specification Refinement
  - Reasoning about Specifications
  - Have I Written Enough Properties?
  - Property Directed Simulation Games

The Design Intent Verification Flow



References

1. Banerjee, A., B.Pal, S.Das, A.Kumar, P.Dasgupta, *Test Generation Games from Formal Specifications,* DAC 2006.

2. Basu, P., S.Das, A.Banerjee, P.Dasgupta, P.P.Chakrabarti, C.R.Mohan, L.Fix, R.Armoni, *Design Intent Coverage – A New Paradigm for Formal Property Verification.* IEEE Trans. on CAD, Oct 2006.

3. Chockler, H., O.Kupferman, R.P.Kurshan, M.Y.Vardi, *A practical approach to coverage in model checking.* CAV 2001.

4. Chockler, H., O.Kupferman, M.Y.Vardi, *Coverage metrics for temporal logic model checking.* TACAS 2001, LNCS 2031.

5. Chockler, H., O.Kupferman, M.Y.Vardi, *Coverage Metrics for Formal Verification.* LNCS 2860, 2003.

6. Das, S., P.Basu, A.Banerjee, P.Dasgupta, P.P.Chakrabarti, C.R.Mohan, L.Fix, R.Armoni, *Formal Verification Coverage: Computing the coverage gap between temporal specifications,* ICCAD 2004.

7. Das, S., A.Banerjee, P.Basu, P.Dasgupta, P.P.Chakrabarti, C.R.Mohan, L.Fix, *Formal methods for analyzing the completeness of an assertion suite against a high-level fault model,* VLSI Design, 2005.

References

8. Das, S., P.Basu, P.Dasgupta, P.P.Chakrabarti, *What lies between design intent coverage and model checking?* DATE 2006.

9. Dasgupta, P., *A Roadmap for Formal Property Verification,* Springer 2006.

10. Dill, D.L., *Trace Theory for Automatic Hierarchical Verification of Speed-independent circuits,* ACM Distinguished Dissertations, MIT Press, 1989.

11. Hoskote, Y., T.Kam, P.H.Ho, X.Zao, *Coverage estimation for symbolic model checking,* DAC 1999.

12. Pnueli, A., R.Rosner, *On the synthesis of a reactive module,* ACM Symposium on Principles of Programming Languages, 1989.