

Summer Formal 2011

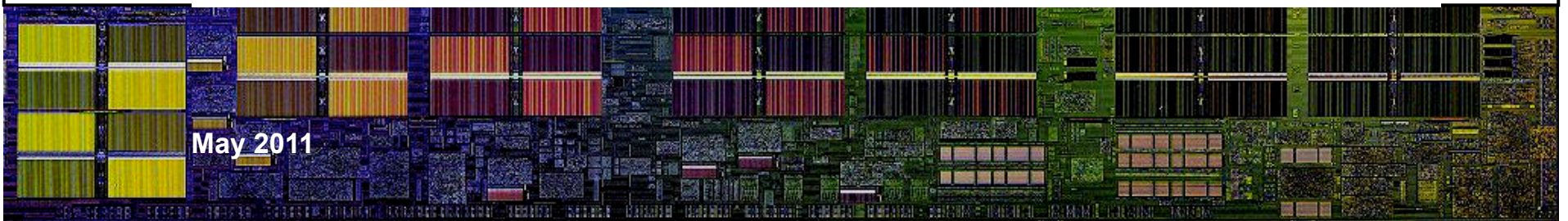
Hardware Verification 101

Hardware Verification Foundations

Jason Baumgartner

www.research.ibm.com/sixthsense

IBM Corporation



Outline

■ Class 1: Hardware Verification Foundations

- Hardware and Hardware Modeling
- Hardware Verification and Specification Methods
- Algorithms for Reasoning about Hardware

■ Class 2: Hardware Verification Challenges and Solutions

- Moore's Law v. Verification Complexity
- Coping with Verification Complexity via Transformations

■ Class 3: Industrial Hardware Verification In Practice

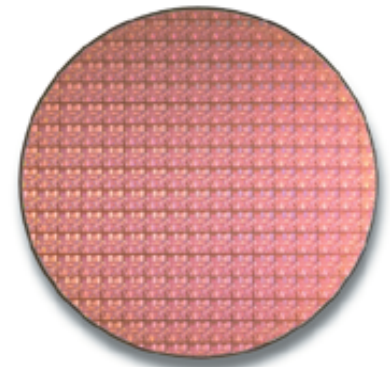
- Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies

Outline

- Hardware and Hardware Modeling
- Hardware Verification and Specification Methods
- Algorithms for Reasoning about Hardware
 - Falsification Techniques
 - Proof Techniques
 - Reductions

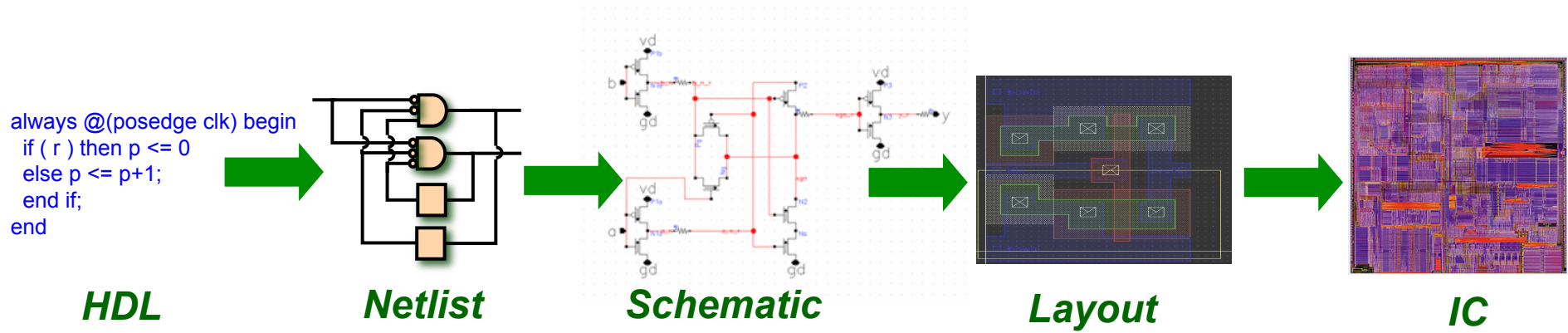
Introduction to *Hardware*

- Integrated circuits (ICs) are ubiquitous in modern life
 - Computers, Audio/Video devices, Transportation, Medical devices, Communications, Appliances, ...
- Many types of ICs
 - Processors, GPUs, RAM, Caches, Networking / Data Routing, Digital Signal Processors, Encryption, ...
- *Hardware* refers to fabricated ICs – *or their origins*



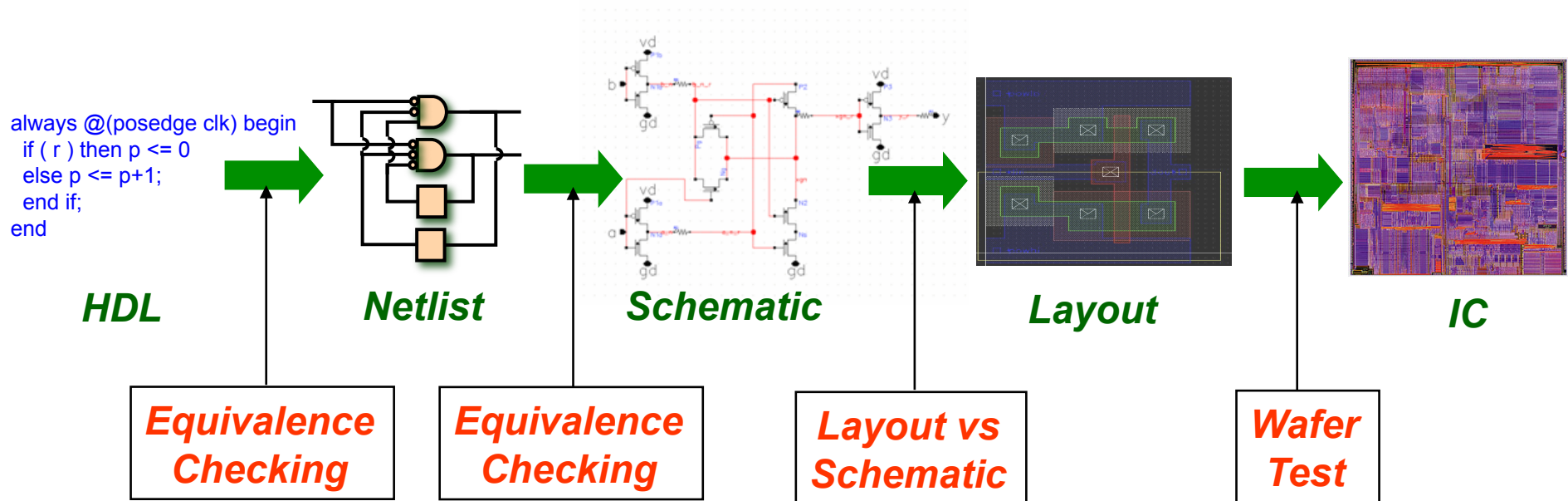
Introduction to *Hardware*

- Contemporary hardware design often begins as a Hardware Description Language (Verilog, VHDL)
- Taken through a series of synthesis steps into gate-level netlist representation, then a transistor-based representation
- Mapped to a physical layout, lithography masks, ... finally silicon!



Introduction to *Hardware Verification*

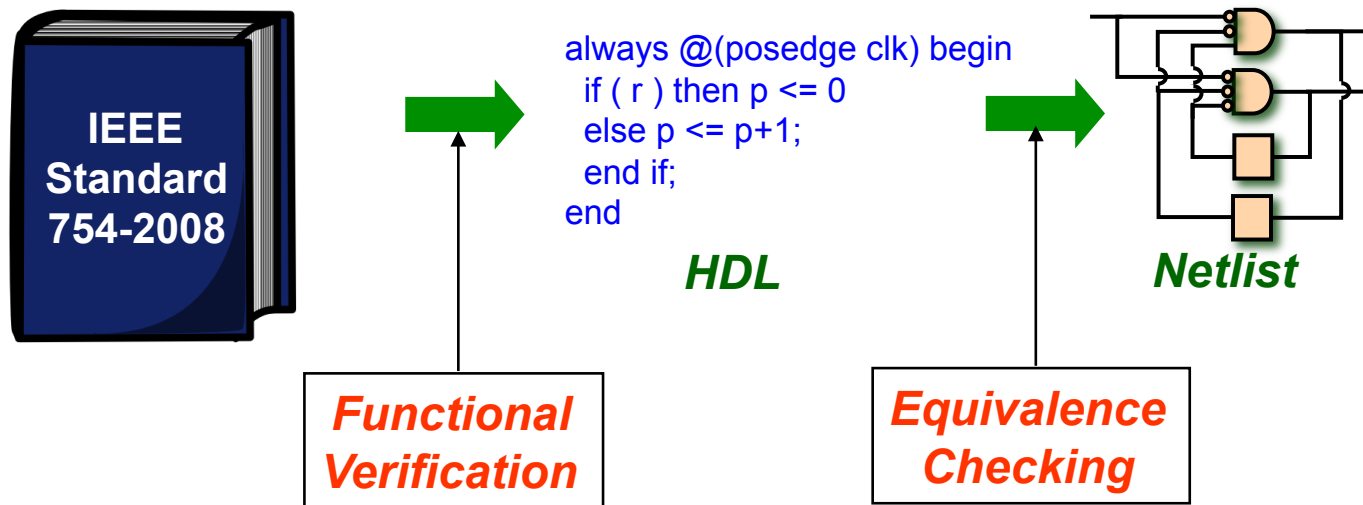
- Numerous types of **verification** relevant to hardware design



- Also timing analysis, circuit analysis, protocol analysis, ...

Introduction to *Hardware Verification*

- We focus solely upon *logical implementation verification*
- Including *functional verification*, i.e. model checking



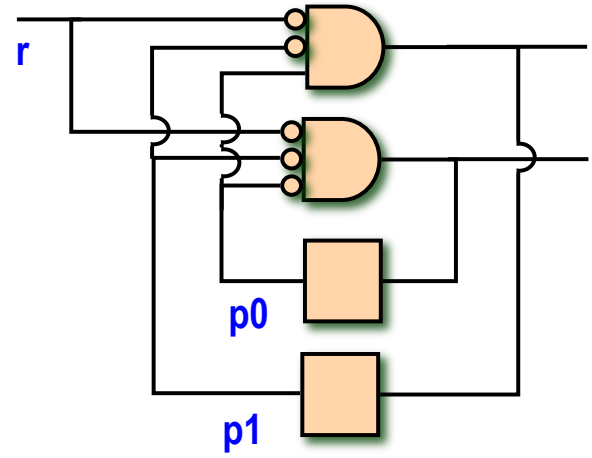
- Though the techniques we discuss may also be applied to architectural models, protocol models, software-like models ...
 - As long as they are *synthesizable*

Introduction to *Hardware*

- Hardware may be represented as a *program*

```

always @(posedge clk) begin
  if ( r ) then p <= 0
  else p <= p+1;
end if;
end
                    
```

- Or as a gate-level *netlist*


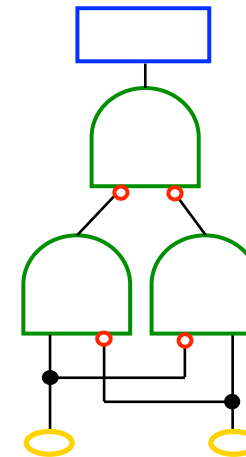
- We hereafter assume a *netlist view of hardware*
 - Finite, discrete (Boolean), no combinational cycles

Netlist Formats: Numerous Types

- Logic synthesis usually follows a structured flow
 - Word-level netlist directly correlating to HDL constructs
 - Adder: `a <= b + c;`
 - Multiplexor: `a <= if sel then data1 else data0;`
 - ...
 - Then a sequence of steps into simpler logic primitives
 - For silicon flows, primitives are dictated by fabrication technology
 - Various libraries are possible; often NAND and NOR gates
 - For verification flows, the *And / Inverter Graph* is popular

And / Inverter Graph (AIG)

- **Registers:** state-holding elements
- 2-input **AND** gates
- **Inverters:** implicit as edge attributes
- **Primary inputs:** nondeterministic Boolean values
- A constant 0 gate



a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

- Registers have associated:
 - Initial values, defining time-0 behavior
 - Next-state functions, defining time $i+1$ behavior
 - Value of next-state function at time i is applied to register at time $i+1$
 -

And / Inverter Graph (AIG)

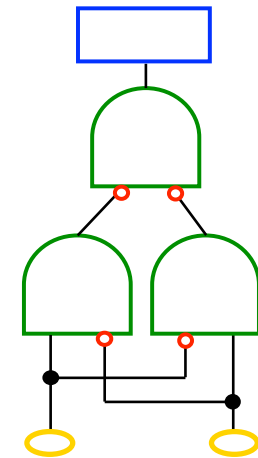
- + **Compact representation:** very few bytes / gate
- + Enables efficient processing due to monotonic types
 - Including more primitives such as XORs may reduce gate count, though often disadvantageous in other ways
 - Easy to infer XORs if desirable
- + Common format for model checking and equiv checking
 - AIGER format; Hardware Model Checking Competition <http://fmv.jku.at/aiger>
- Loss of higher-level primitives may entail overhead, preclude higher-level techniques such as SMT
 - Netlists are sometimes augmented with other gate types, e.g. with *arrays*

Netlist Flexibility

- Boolean netlists are able to model a rich variety of problems
- Higher-level gates such as multipliers? Bit-blast
- **Three-valued** reasoning? Use dual-rail encoding
 - Each gate g mapped to a pair $\langle g_l, g_h \rangle$ encoding

g	$\langle g_l, g_h \rangle$
0	0 0
1	1 1
X	0 1

a	b	a&b
0	0	0
0	1	0
0	X	0
1	0	0
1	1	1
1	X	X
X	0	0
X	1	X
X	X	X



- Multi-value reasoning? Discretely encode into Boolean gates

Outline

■ Hardware and Hardware Modeling

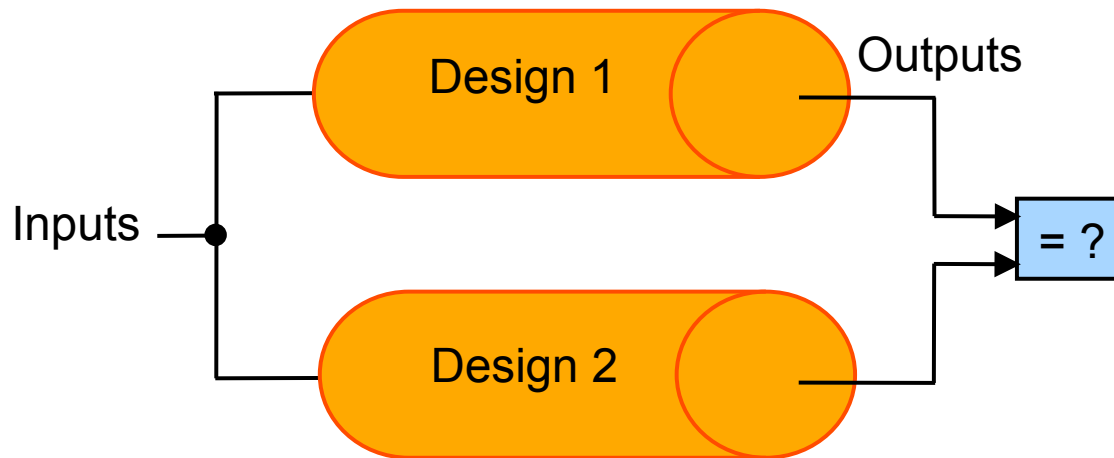
■ Hardware Verification and Specification Methods

■ Algorithms for Reasoning about Hardware

- Falsification Techniques
- Proof Techniques
- Reductions

Equivalence Checking

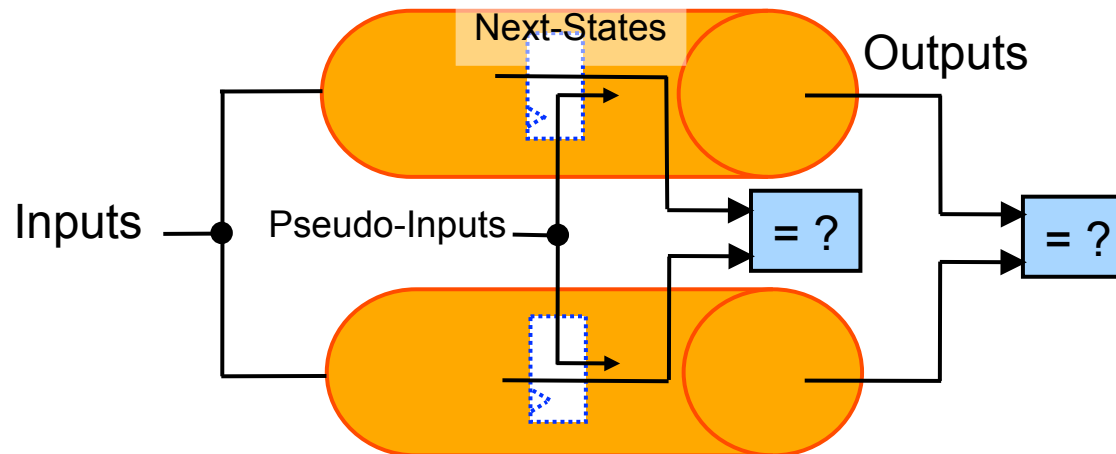
- A method to assess behavioral equivalence of two designs



- Validates that certain design transforms preserve behavior
 - E.g., logic synthesis does not introduce bugs
 - Design1: pre-synthesis Design2: post-synthesis

Combinational Equivalence Checking (CEC)

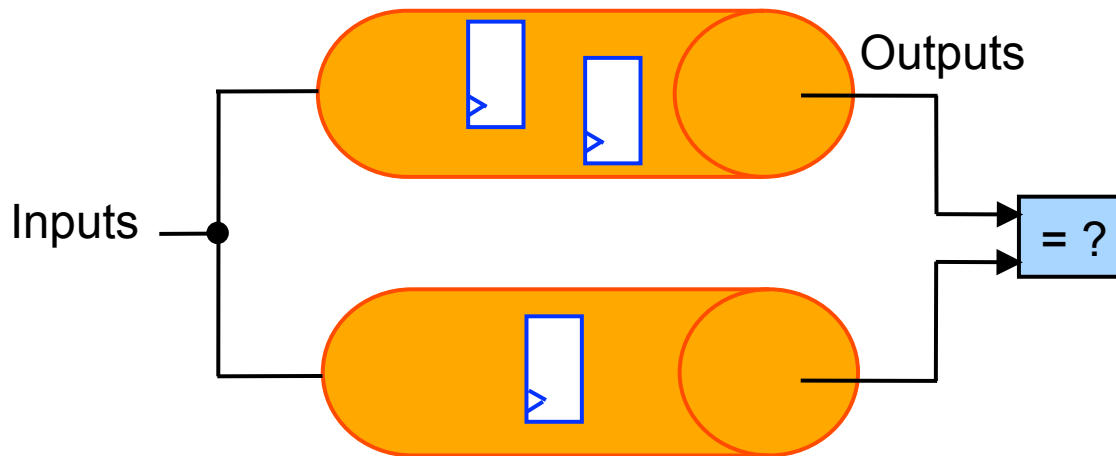
- No sequential analysis: state elements become *cutpoints*



- Equivalence check over outputs + next-state functions
 - + While **NP-complete**, a mature + **scalable** + pervasively used technology
 - Requires 1:1 state element correlation

Sequential Equivalence Checking (SEC)

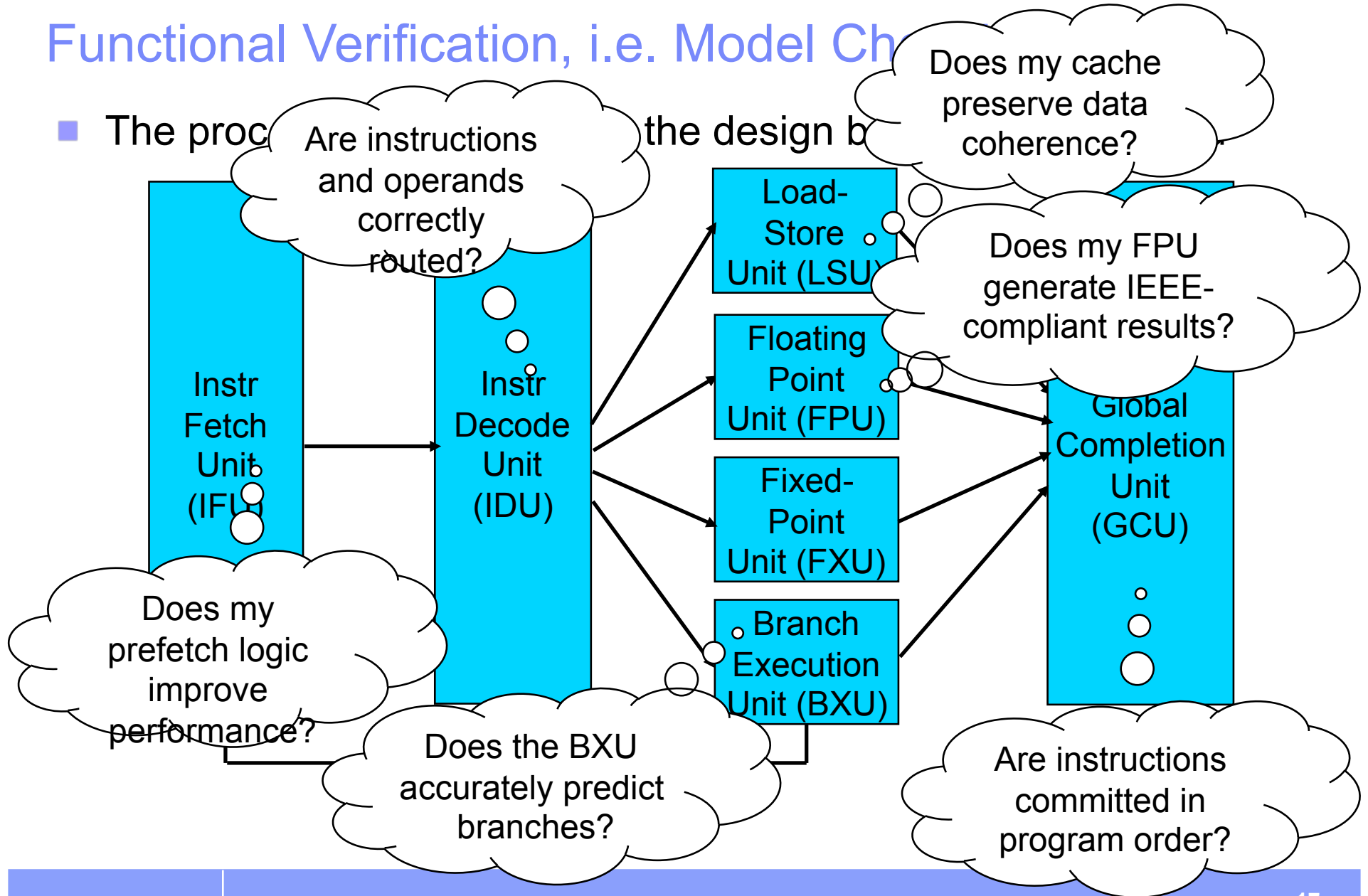
- No 1:1 state element requirement: generalizes CEC



- Greater applicability: e.g. to validate *sequential* synthesis
- Generality comes at a computational price: **PSPACE vs NP**
 - Superficially harder than model checking since **2 models**
 - + Though exist techniques to often **dramatically enhance scalability**

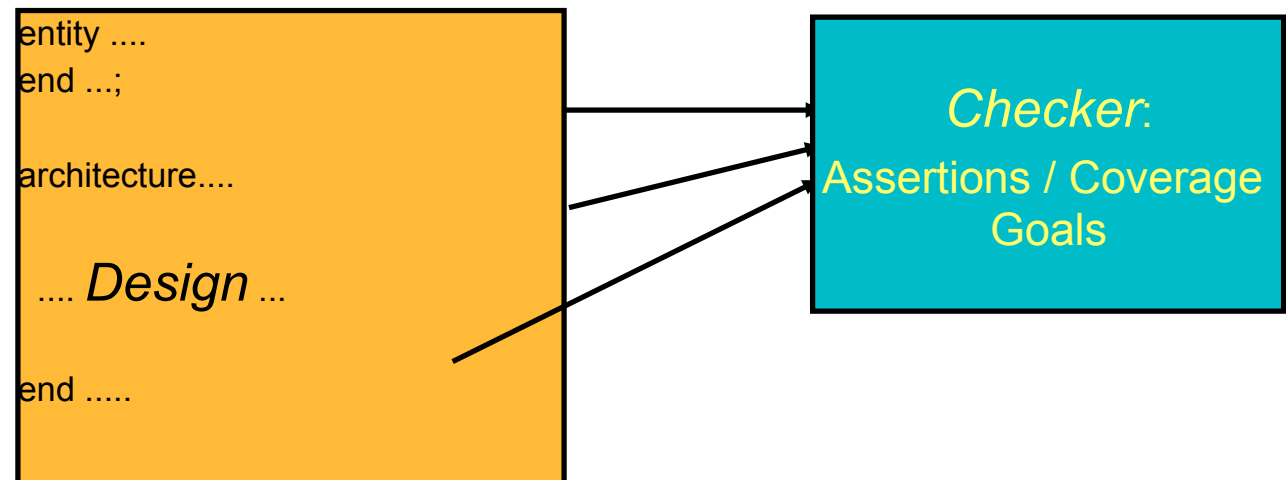
Functional Verification, i.e. Model Ch

- The proc the design b



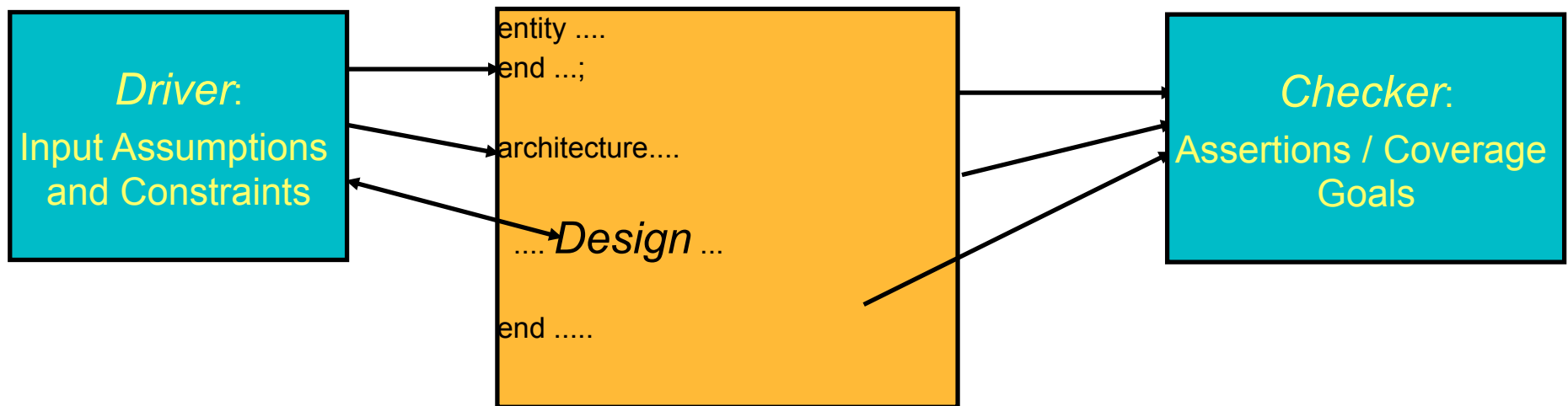
Functional Verification Testbench

- A *testbench* is used to define the properties to be checked
 - Correctness properties, or *assertions*
 - Coverage goals, to help assess completeness and correctness of the verification process



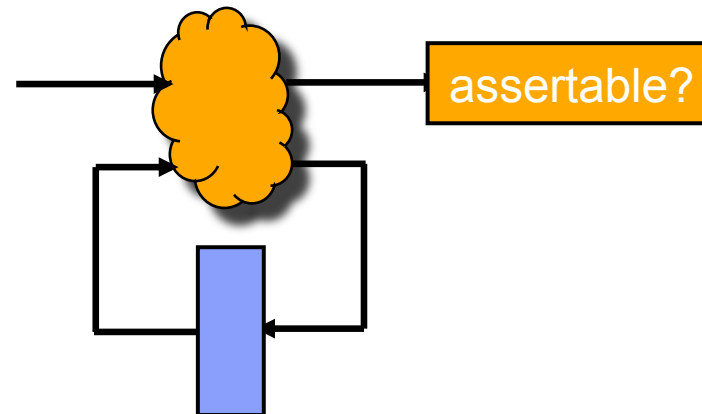
Functional Verification Testbench

- A *testbench* also defines constraints over input stimulus
 - Designs generally assume certain conditions over their inputs
 - E.g., only valid *opcode* input vectors
 - No *req_valid* input if design is asserting *busy*
 - Generally, some handshaking protocol with adjacent logic



Netlist-based Verification

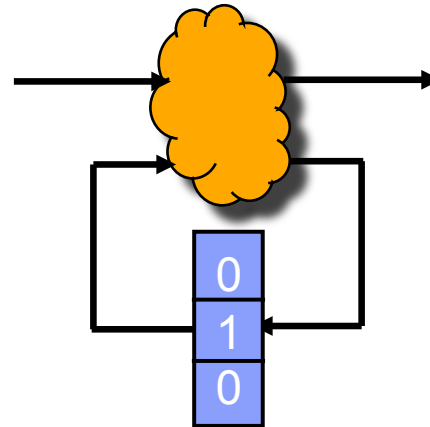
- A verification problem may often be cast as a *sequential netlist*
 - *Correctness properties* may be synthesized into simple assertion checks
 - *Drivers or assumptions* may also be synthesized into the netlist



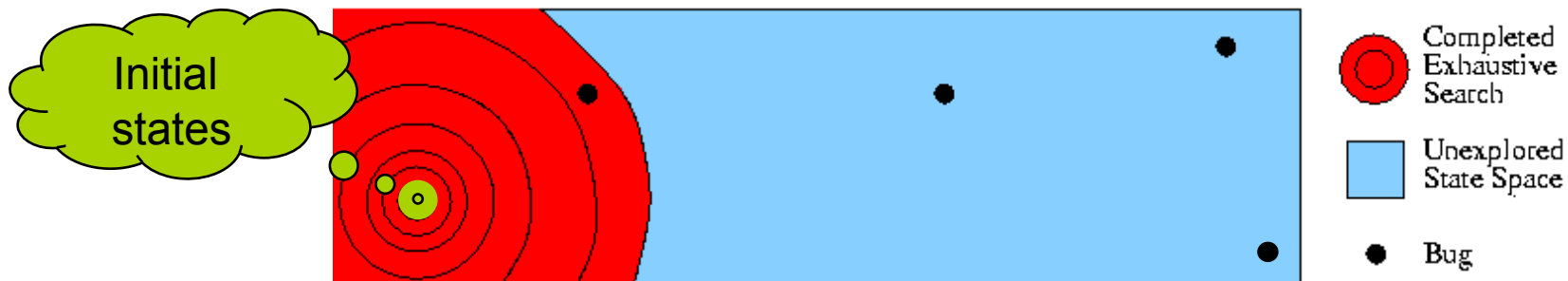
- Properties, constraints handled as specially-annotated gates
 - E.g. – properties are *outputs* in AIGER format

Netlist-Based Verification

- A *state* is a valuation to the registers

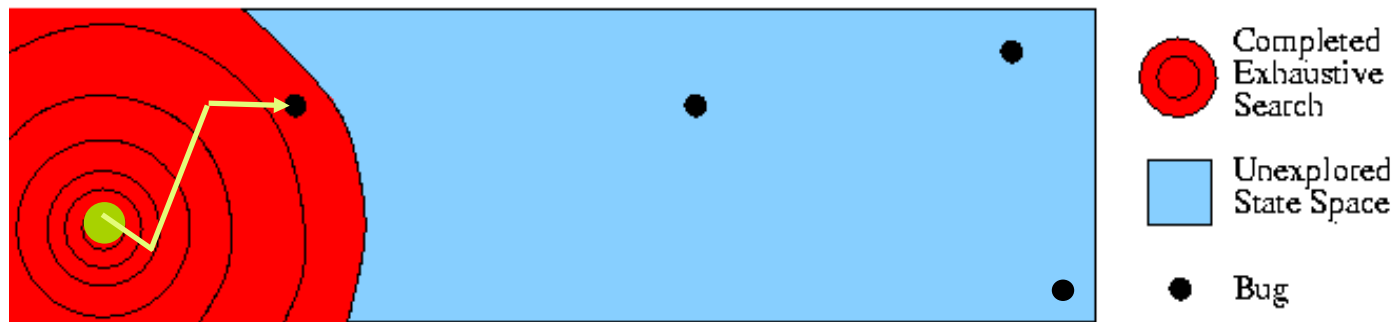


- **Formal verification** generally requires analysis of *reachable* states
 - Hence *initial states* are also part of a testbench specification



Netlist-Based Verification

- Verification goal: a **counterexample trace**, from initial state to one asserting a property



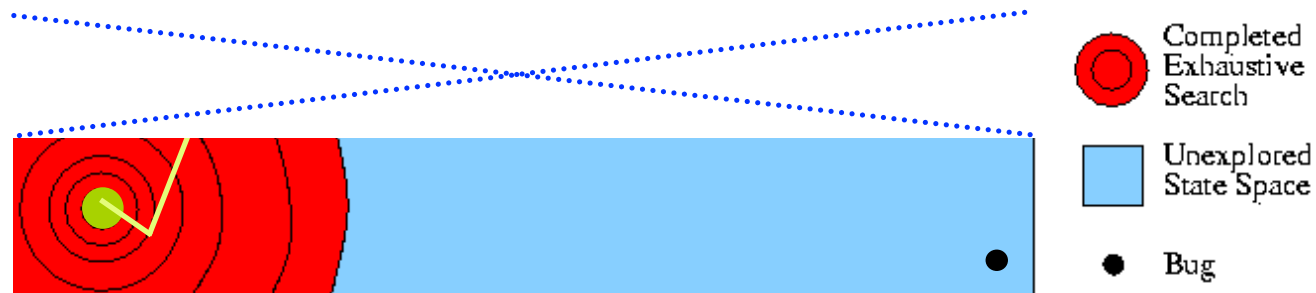
- Or a **proof** that no such trace exists

Functional Verification Testbench

- Two fundamental ways to implement an input assumption

1) Declarative approaches: *constraints* to limit design exploration

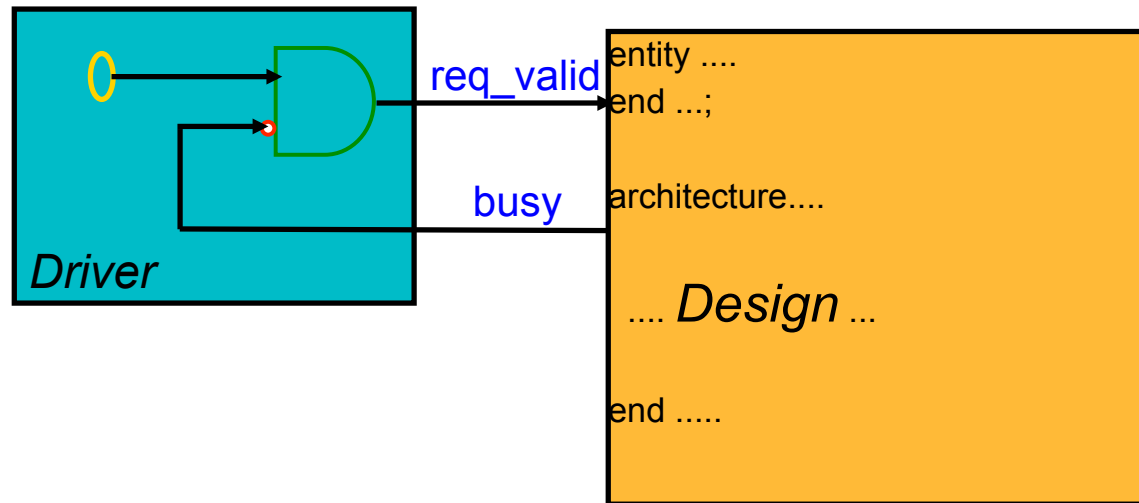
- *assume (busy → not req_valid)*
- A state is considered *unreachable* if it violates a constraint



Functional Verification Testbench

- Two fundamental ways to implement an input assumption

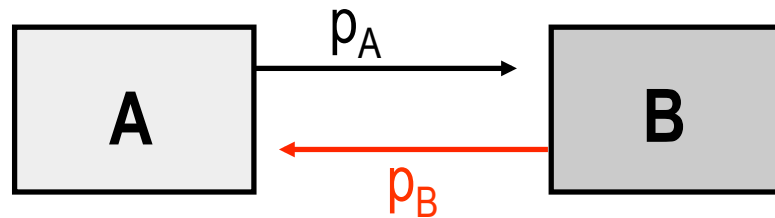
2) Imperative approaches: hardware-like *drivers* used to override inputs / internals of the design



- Strengths and weakness of both

Constraints: Benefits

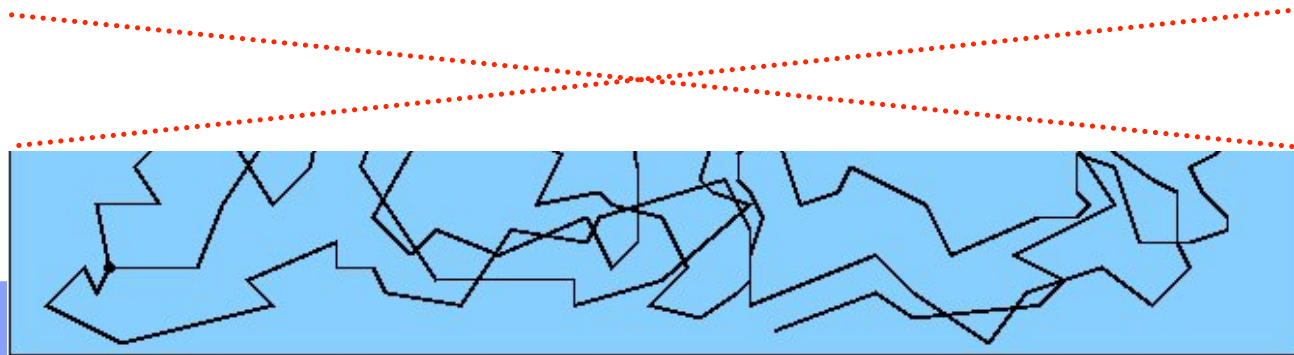
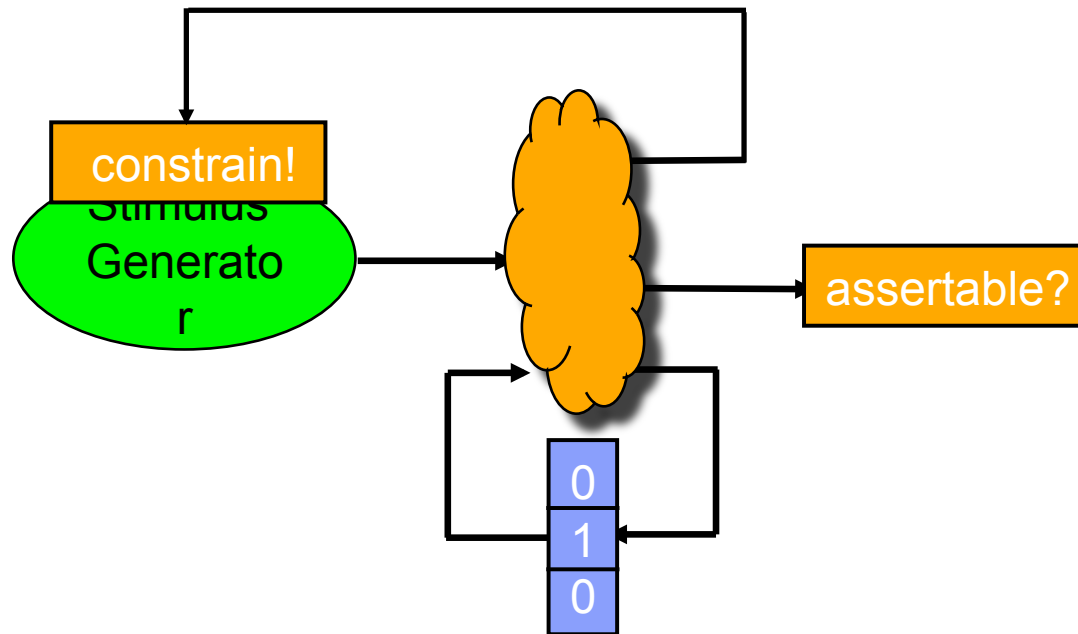
- Simplifies assume-guarantee reasoning
 - When verifying A, assume properties over B and vice-versa
 - Checker / constraint duality



- Enables rapid, incremental testbench prototyping
 - **Given** spurious counterexample: illegal stimulus X when design in state Y
 - **Add** constraint: *assume (state_Y \rightarrow not input_X)*

Constraints: Drawbacks

- Often entail an algorithmic overhead
 - Eg: need a *SAT solver* vs *random generator* to generate legal input stimulus



- Unexplored State Space
- Random Sim
- Bug

Constraints: Drawbacks

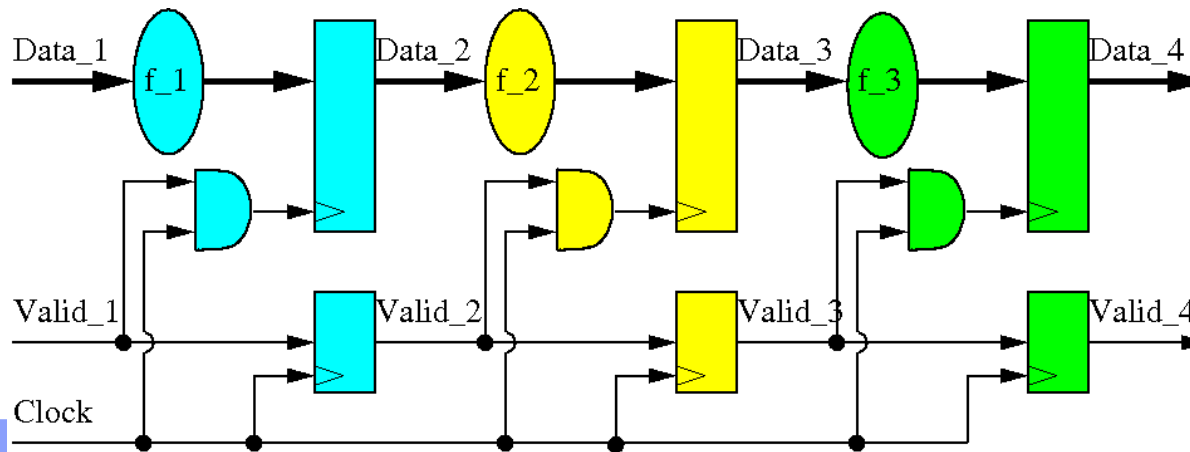
- Often entail an algorithmic overhead
 - Constraints are often of the form *assume* (*state_Y* → *input Y*)
 - Though may be arbitrary expressions: eg *assume* (*state_Y*)
 - *not* (*state_Y*) is a **dead-end state**: no stimulus is possible!
 - Simulator has no recourse but to stop or backtrack
 - While our focus is FV, random simulation is critical even in a robust FV tool
 - Semi-formal analysis, postulating invariants or reduction opportunity,...
 - Practically, the ability to reuse specification across formal and **informal** frameworks is often highly desirable
 - Constraints pose overhead to simulators
 - Fatal bottleneck to accelerators!

Drivers

- Very flexible: can override behavior of individual signals
 - *or1* <= *input1* OR *input2*;
 - Driver could map *input1* to *1*, *or1* to *0*
- Often more efficient to reason about drivers
 - Efficient random stimulus generation; *no dead-end states*
 - More portable to *informal* frameworks, especially acceleration
- Though often more laborious than constraints
 - Like logic design itself: an incremental testbench fix may require a fundamental change in implementation

Dangers of Constraints

- Both drivers and constraints limit the reachable states
 - As does the initial state specification
- *Overconstraining* risks bugs slipping through the FV process
- Constraints are bit riskier in practice due to dead-end states
 - E..g. – may contradict an initial state; rule out large portions of state space
 - **assume (valid_4 → f(data_4));** what if violated before inputs propagate?



Dangers of Constraints

- *Coverage events* are useful sanity-checks against overconstraints, incorrect constraints / initial values
 - As with simulation: used there to assess adequate state space coverage
- Another risk: missing or incorrect property

Q “Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?”

A “I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.”

- Field of *formal coverage* helps address *validity of FV results*
 - Such techniques are of limited practical use; *not* discussed herein

Specification Languages

- Hardware is often specified using VHDL or Verilog
- Testbenches are often specified using:
 - VHDL or Verilog asserts
 - Possibly using VHDL or Verilog checker or driver modules

- SystemVerilog Assertions (SVA)

`req ##[1:3] gnt`

- Property Specification Language (PSL)

`req -> next_e[1..3] gnt`

- Simple subset of PSL supported in VHDL



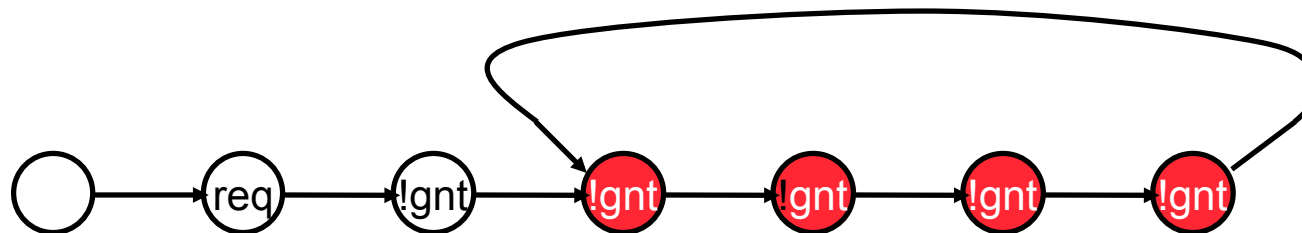
- Original model checking languages are dead in practice

Safety vs Liveness

- A safety property has a finite-length counterexample
 - E.g., every *req* will get a *gnt* within 3 timesteps



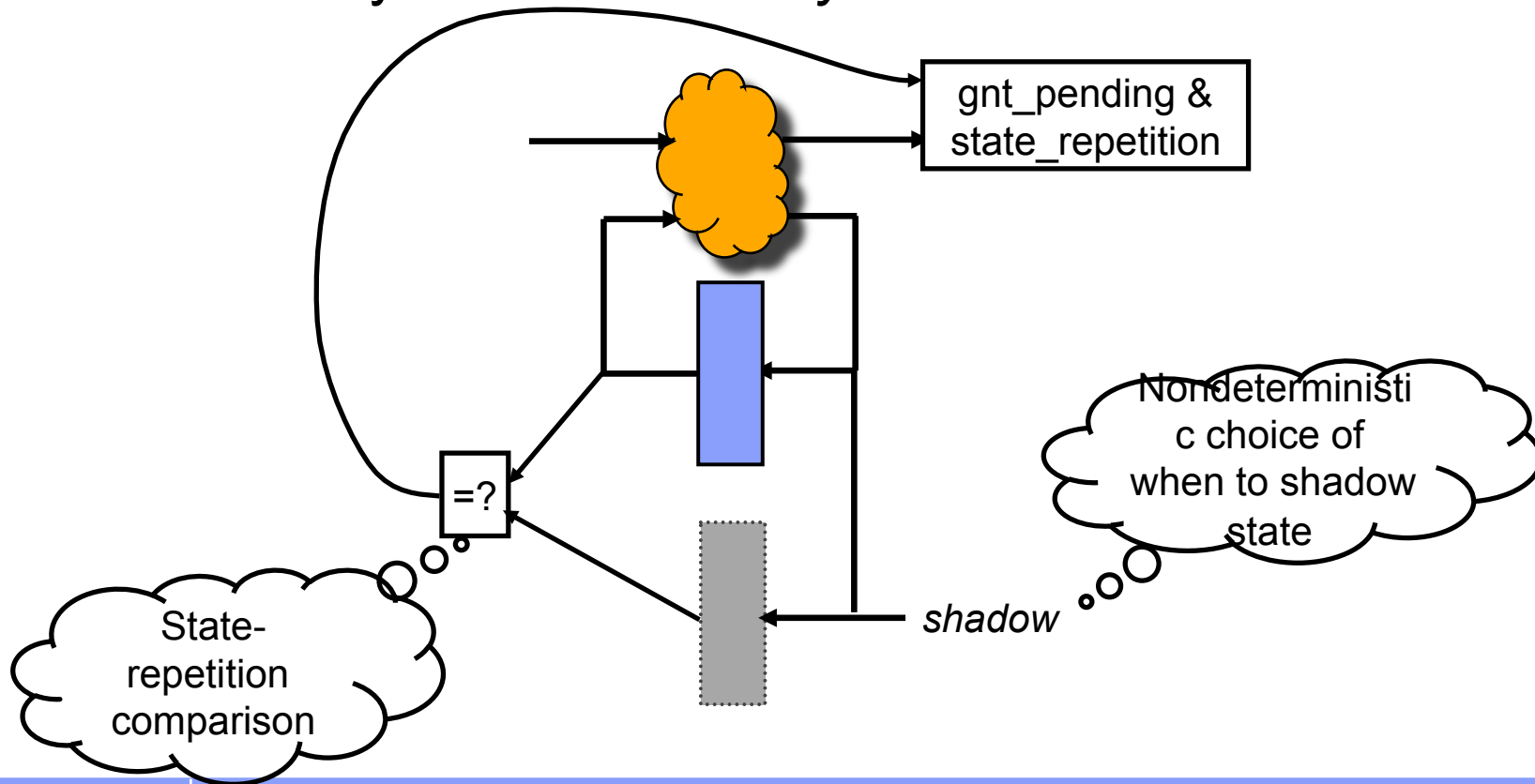
- A liveness property has an infinite-length counterexample
 - E.g. every *req* will eventually get a *gnt*



- Practically represent infinite counterexamples as a lasso
 - A prefix, followed by a state-repetition suffix

Safety vs Liveness

- Liveness checking traditionally performed via dedicated algos
- *Liveness* may be cast to *safety* via a netlist transform!



Liveness vs Bounded Liveness

+ Benefits:

- Shorter counterexamples: no need to wait for longest “valid” delay
 - May be faster to compute accordingly
- Easier to debug counterexamples
- No need to experiment to find the bound

■ Drawbacks:

- Shadow registers entail more difficult proofs
- Computed bounds are nonetheless useful for performance characterization

■ Liveness checking recently supported by most industrial tools

- Though not yet by AIGER; we focus on safety herein

Outline

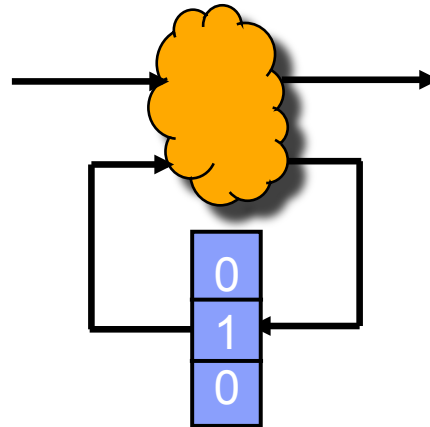
- Hardware and Hardware Modeling

- Hardware Verification and Specification Methods

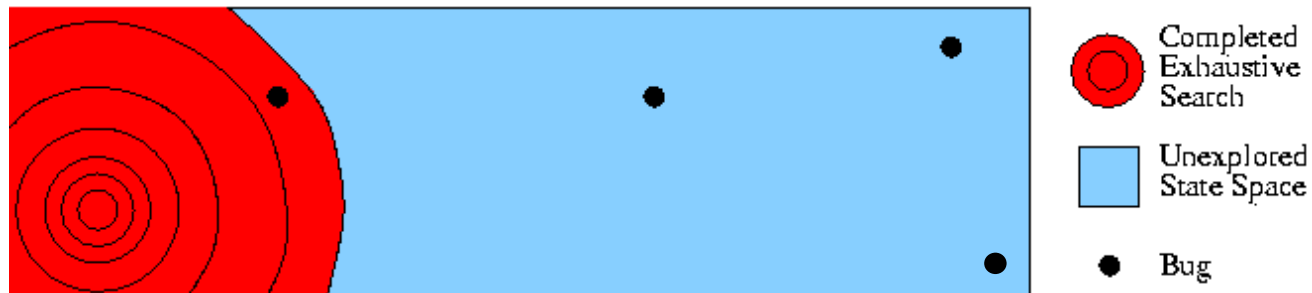
- Algorithms for Reasoning about Hardware
 - Falsification Techniques
 - Proof Techniques
 - Reductions

Verification Complexity

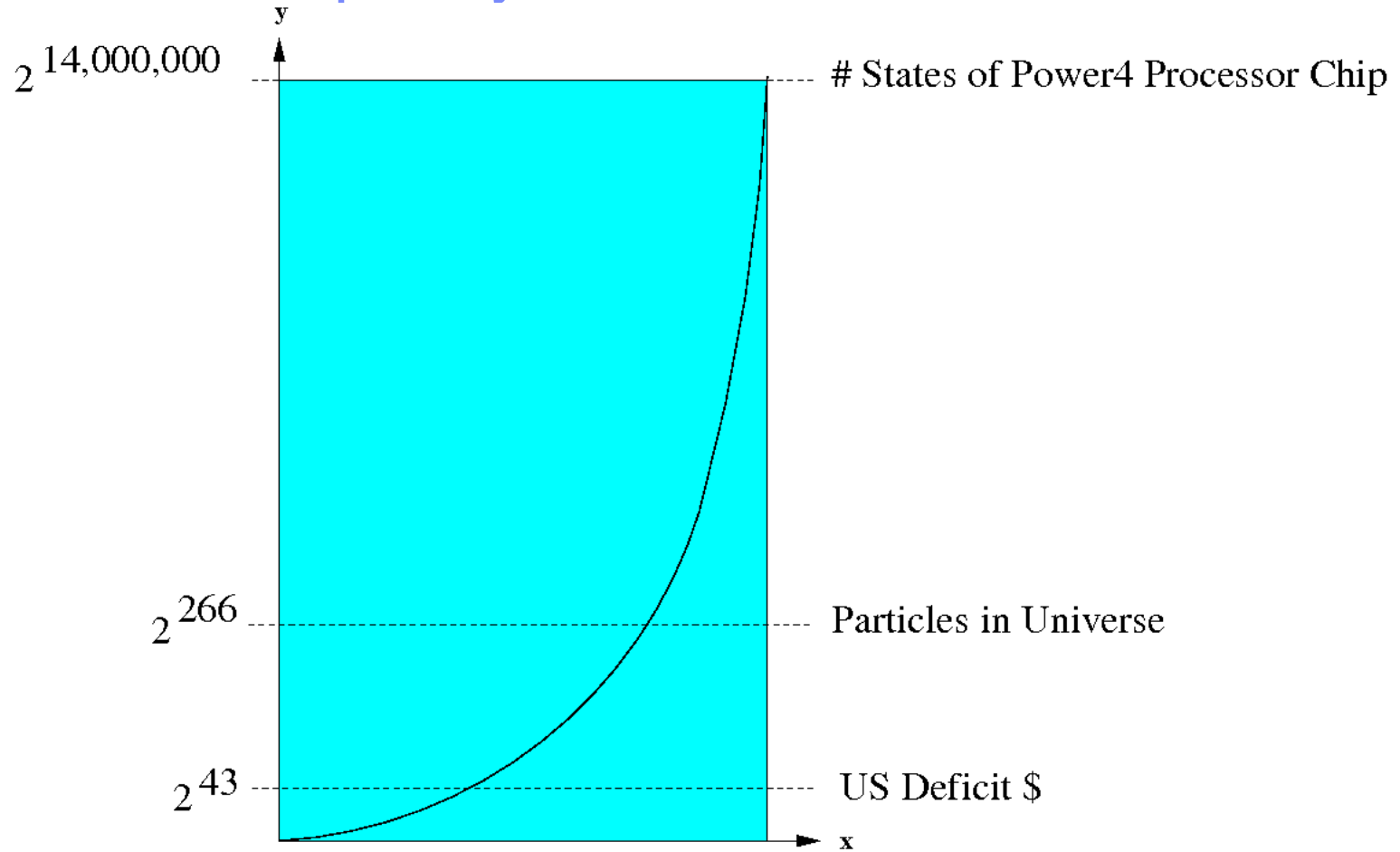
- A *state* is a valuation to the registers



- Exhaustive verification generally requires analysis of all *reachable* states

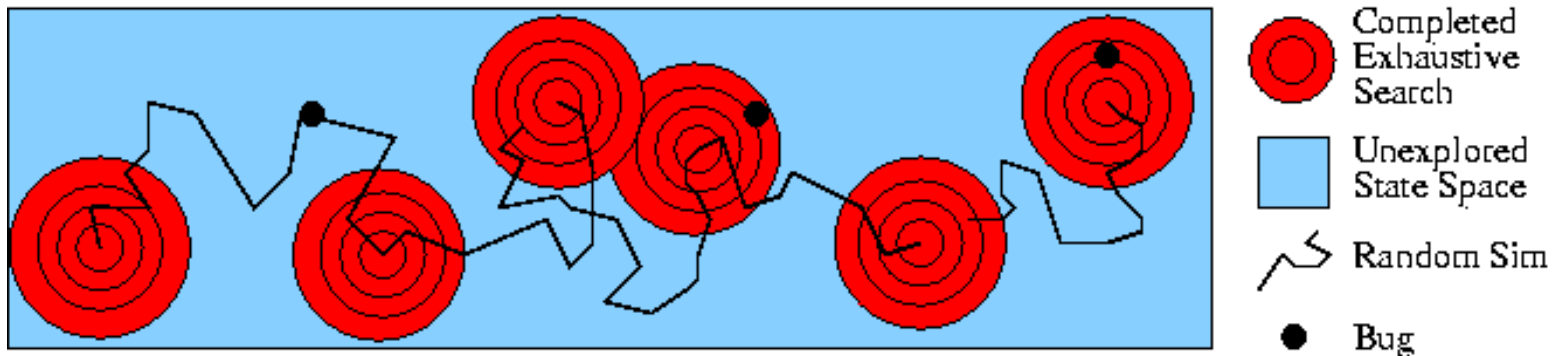


Verification Complexity: *Intractable!*



Coping with Verification Complexity: Underapproximation

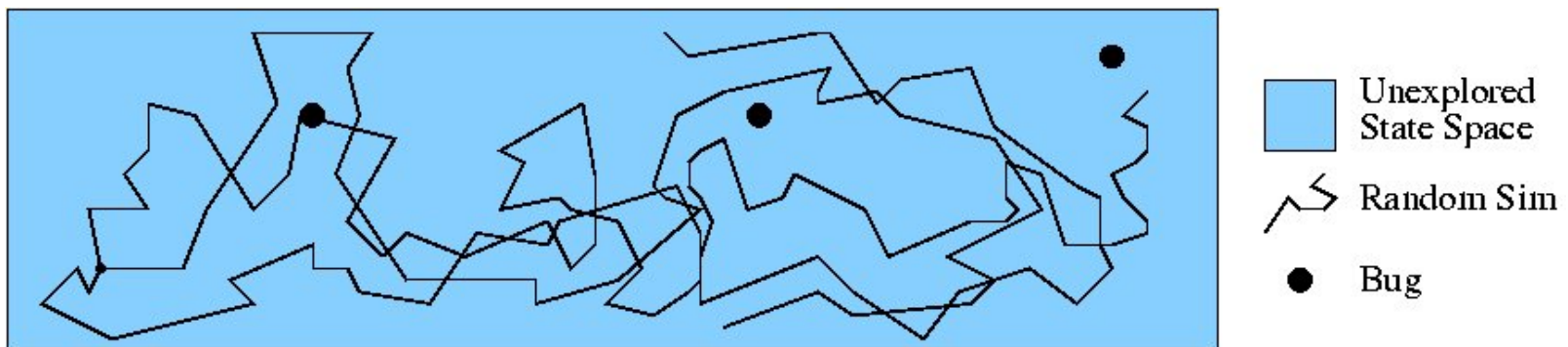
- Formal verification generally requires analysis of *reachable states*
- *Falsification* only requires exploring a *subset*



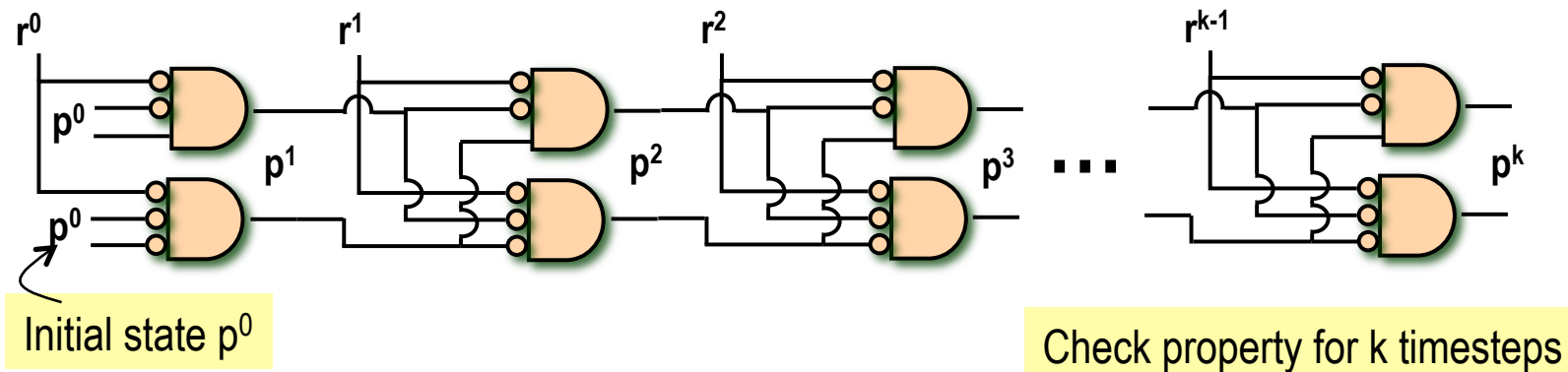
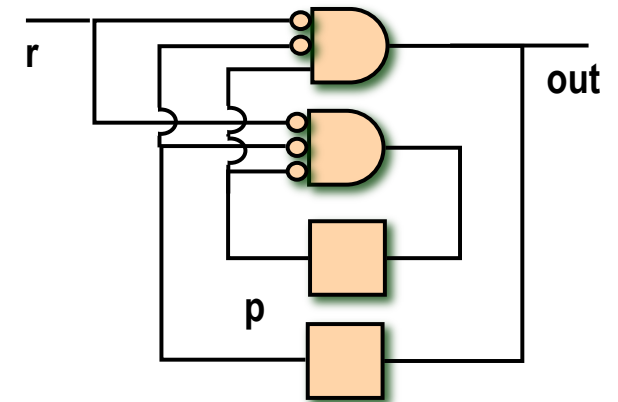
- Benefit: lower complexity class vs general unbounded techniques
 - **NP vs PSPACE**
- Drawback: *not exhaustive (unsound)*, hence proof-incapable

Simulation

- A “random walk” through the state space of the design
- + Scalable: applicable to designs of any size
- + **Very** robust set of tools & methodologies available for this technique
 - + Constraint-based stimulus generation; random biasing
 - + Clever testcase generation techniques
- Explicit one-state-at-a-time nature *severely limits attainable coverage*
 - Suffers the **coverage problem**: often fails to expose every bug

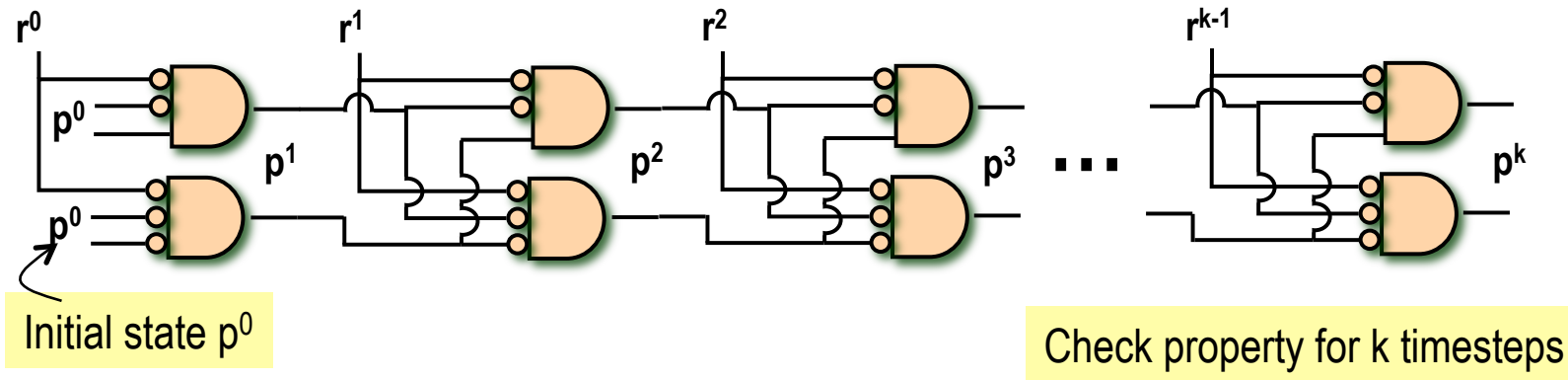


Symbolic Simulation, i.e. Bounded Model Checking



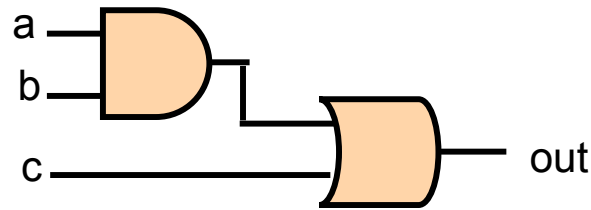
- Unfold netlist k times, representing first k timesteps of behavior
 - Time 0: instantiate initial states in place of registers
 - Time i+1: reference next-state function from time i
- A netlist unfolding sometimes referred to as a *transition relation*

Bounded Model Checking



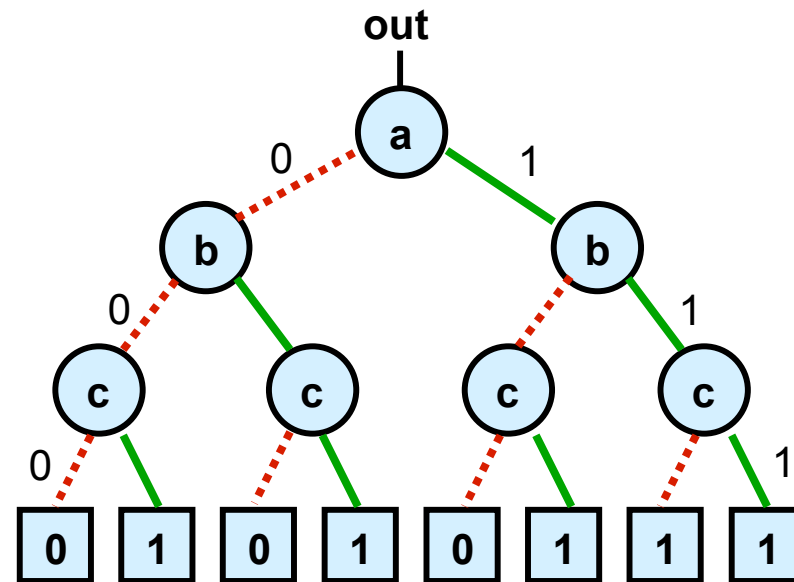
- Given k timestep unfolding, leverage some formal reasoning technique to discern whether properties falsifiable in that window
 - Usually SATisfiability solvers
 - Sometimes BDDs
 - Sometimes combinations of netlist rewriting + SAT + BDDs

Binary Decision Diagrams (BDDs)

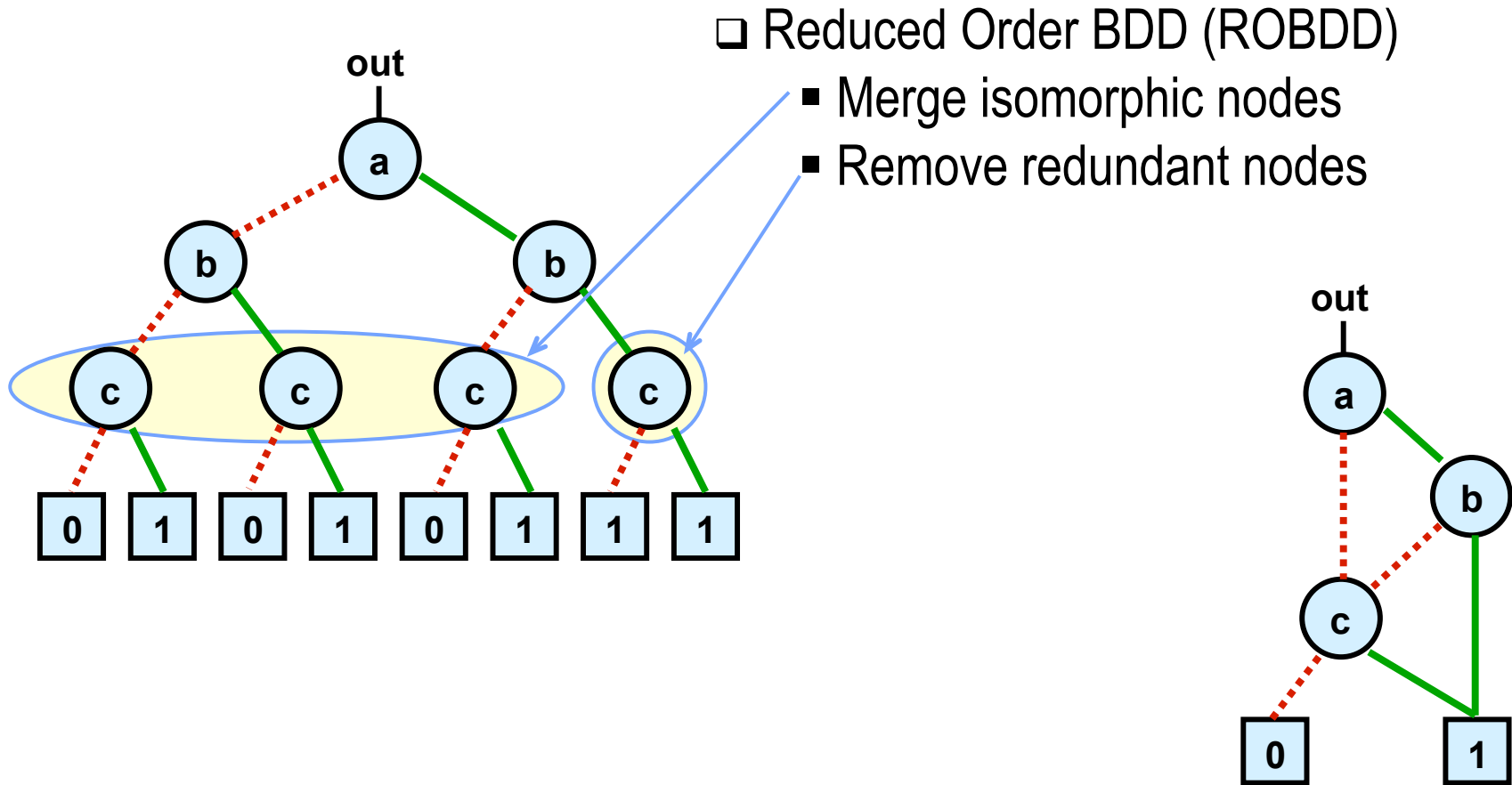


- ❑ Ordered binary trees
 - ❑ *Canonical* w.r.t. variable ordering
- ❑ Can be used to encode a *function* : truth table
- ❑ Or a *relation*

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

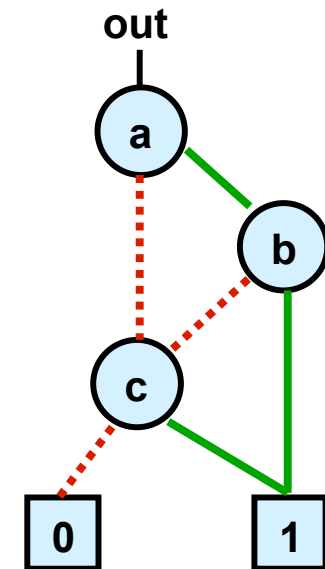
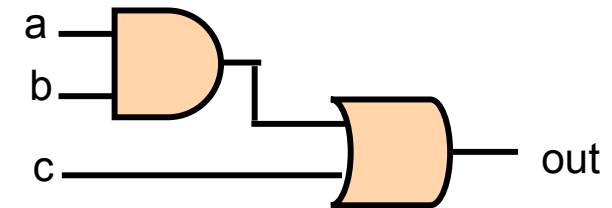


Reduced Ordered Binary Decision Diagrams (ROBDDs)



Reduced Ordered Binary Decision Diagrams (ROBDDs)

- ❑ Reduction performed on-the-fly as BDD built
 - ❑ Similar to constructing a netlist
 - ❑ Create var node
 - ❑ Create constant node
 - ❑ Create conjunction over two nodes
 - ❑ Invert a node



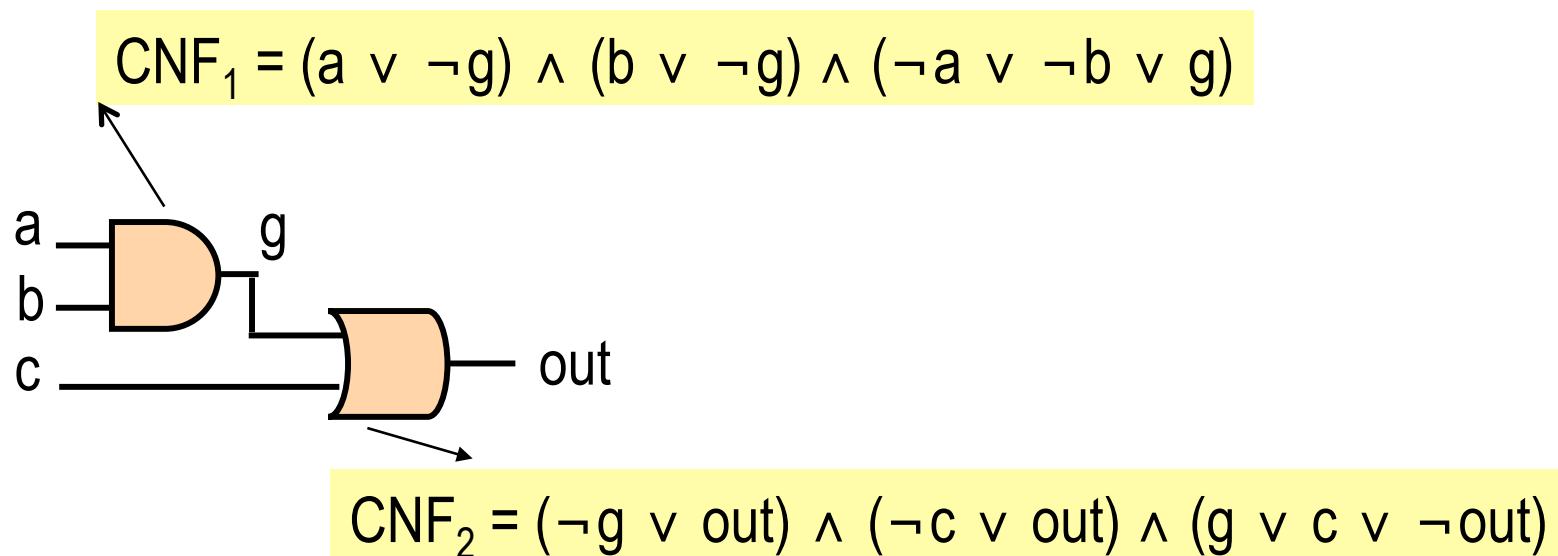
- ❑ Any root-to-1 path indicates satisfiability of **out**
 - ❑ Since canonical, unsatisfiable iff **out** is **0**

- ❑ Often compact, though risks exponential blowup netlist vs BDD

- ❑ Primary strength of BDDs: efficient quantification, unnecessary for BMC

Satisfiability Solving: Conjunctive Normal Form (CNF)

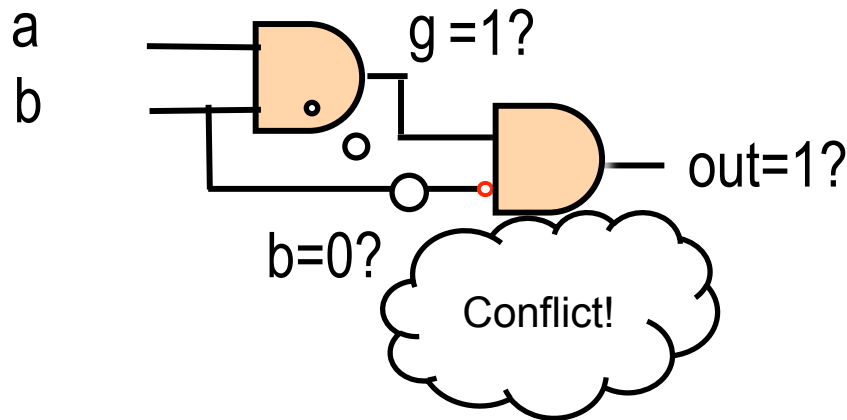
- First convert netlist to CNF formula
- Then assess whether **output** is satisfiable in CNF formula



- In particular, is $(CNF_1 \wedge CNF_2 \wedge out)$ satisfiable?
- Often more scalable than BDDs in this domain; not always
- Risk exponential runtime

Satisfiability Solving: Circuit-Based

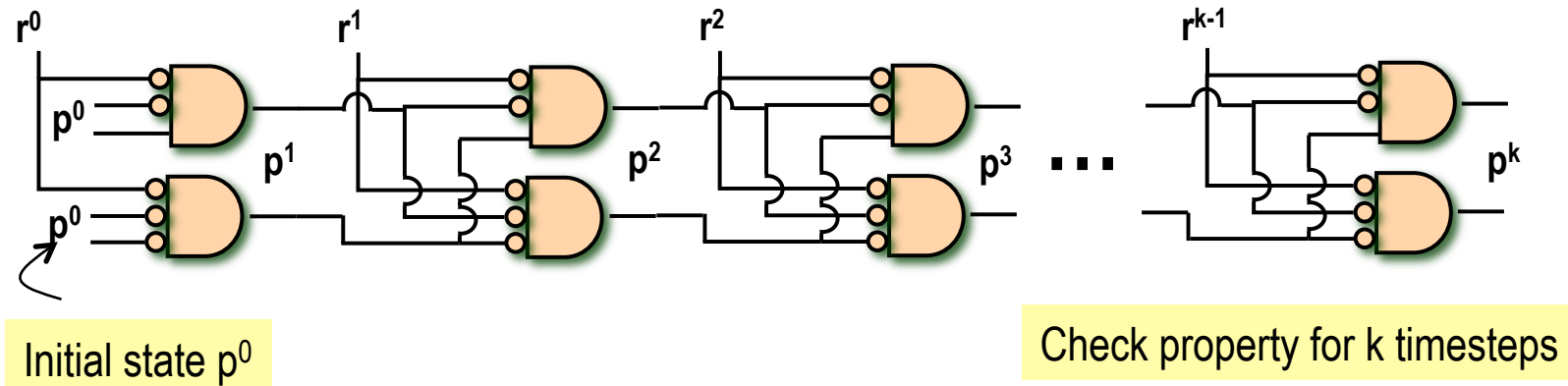
- *Circuit SAT solvers* operate directly upon netlist
- Avoid CNFization overhead; more efficient for *local* reasoning
- Though most SAT research focuses upon CNF



<i>current_state</i>	<i>next_state</i>	<i>action</i>
$1 \rightarrow X$	$1 \rightarrow X$	STOP
$X \rightarrow 1$	$X \rightarrow 1$	CONFLICT
$X \rightarrow 0$	$X \rightarrow 0$	CASE_SPLIT
$0 \rightarrow X$	$0 \rightarrow 0$	PROP_FORWARD
$X \rightarrow 1$	$1 \rightarrow 1$	PROP_LEFT_RIGHT
...

- No clear winner on Circuit vs CNF SAT: a religious debate

Bounded Model Checking (BMC)

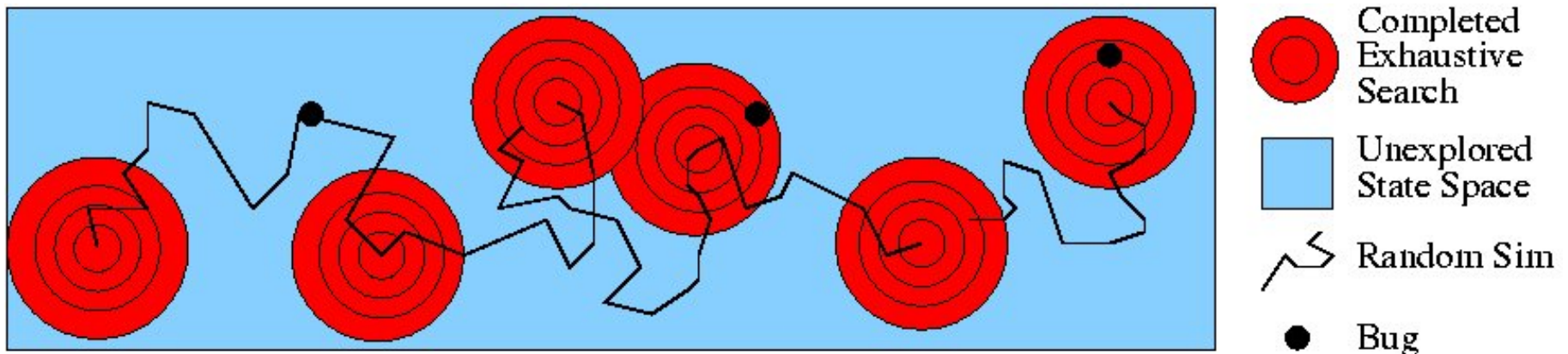


- BMC is often highly scalable; a very effective bug hunter
- Though *incomplete*
 - Checking depth k does not imply absence of a bug at $k+1$
 - Unless we know that *diameter* is $\leq k$
- May break down in practice for *deep* bugs, multipliers, ...

Semi-Formal Verification: Deeper into the State Space

- Semi-Formal Verification (SFV): a hybrid between simulation and FV
 - Uses simulation to get deep into state space
 - Uses resource-bounded formal algos to **amplify** simulation results
 - Effective at hitting deeper failure scenarios

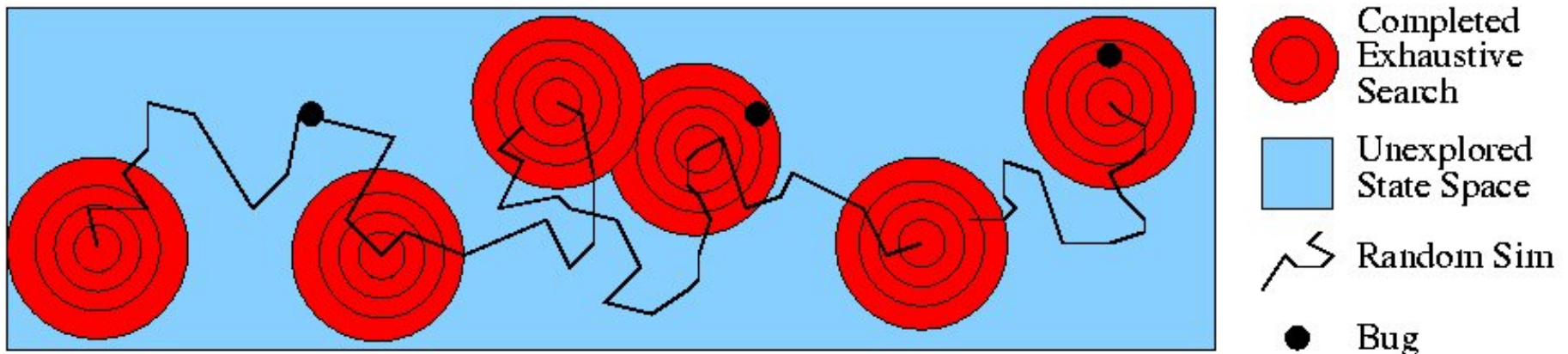
- Alternatively may overconstrain symbolic sim; usually not as scalable



Semi-Formal Verification: Deeper into the State Space

- Note: choice of initial states is somewhat arbitrary
 - No need to drive initialization sequence in the testbench!
 - Initialize testbench post-sequence; validate init mechanism separately!

- Hybrid approach enables **highest coverage**, if design too big for FV
 - **Scales to very large designs**

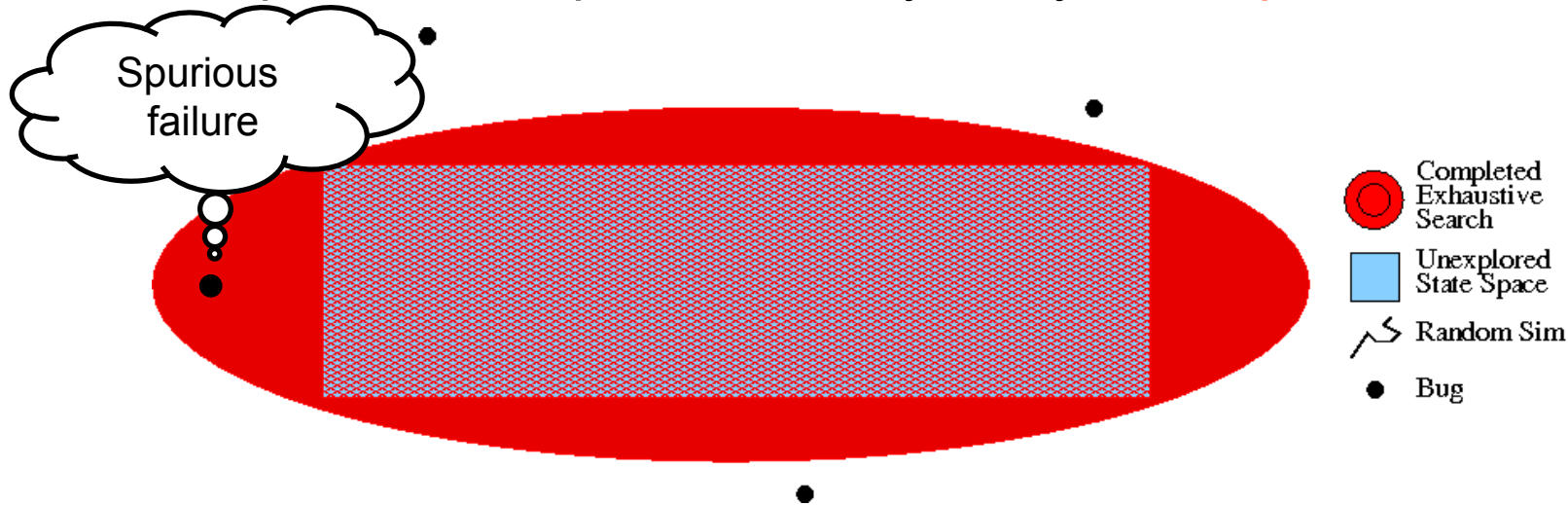


Outline

- Hardware and Hardware Modeling
- Hardware Verification and Specification Methods
- Algorithms for Reasoning about Hardware
 - Falsification Techniques
 - Proof Techniques
 - Reductions

Coping with Verification Complexity: Overapproximation

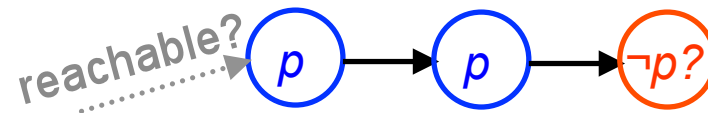
- Formal verification generally requires analysis of *reachable states*
- Some *proof techniques* efficiently analyze a *superset*



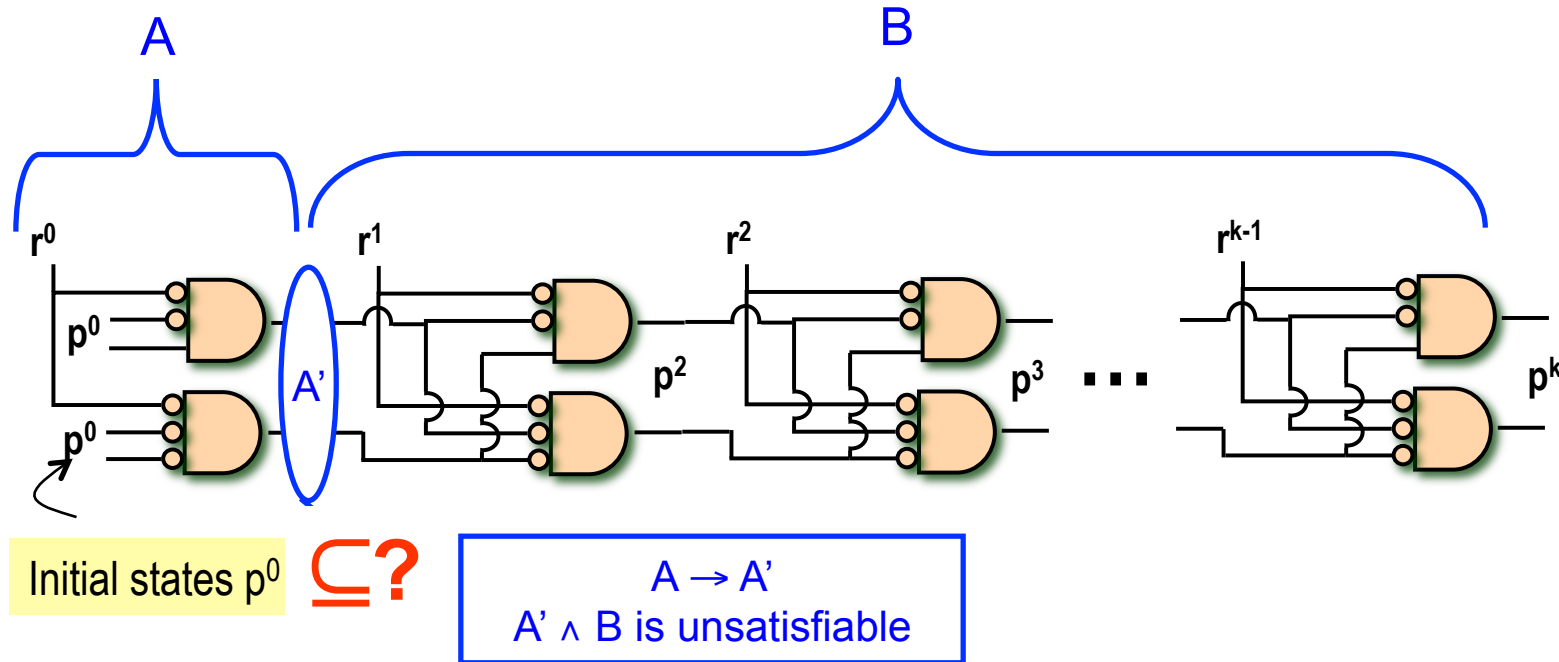
- Benefit: lower complexity class vs general unbounded techniques
 - *NP vs PSPACE* - or at least *resource-bounded*
- Drawback: *incomplete*; cannot discern validity of a counterexample

Induction

- Highly-scalable when effective, though often inconclusive
 - Unknown whether induction counterexample trace begins in reachable state



- May be strengthened by unique-state constraints: *all_different(pⁱ)*
 - Renders induction *complete*; possibly only at depth of recurrence diameter
- A very useful proof technique overall
 - Easy to disregard as “induction only solves easy problems”!
 - Luckily we have multiple under-represented algos! And we **need** more!!

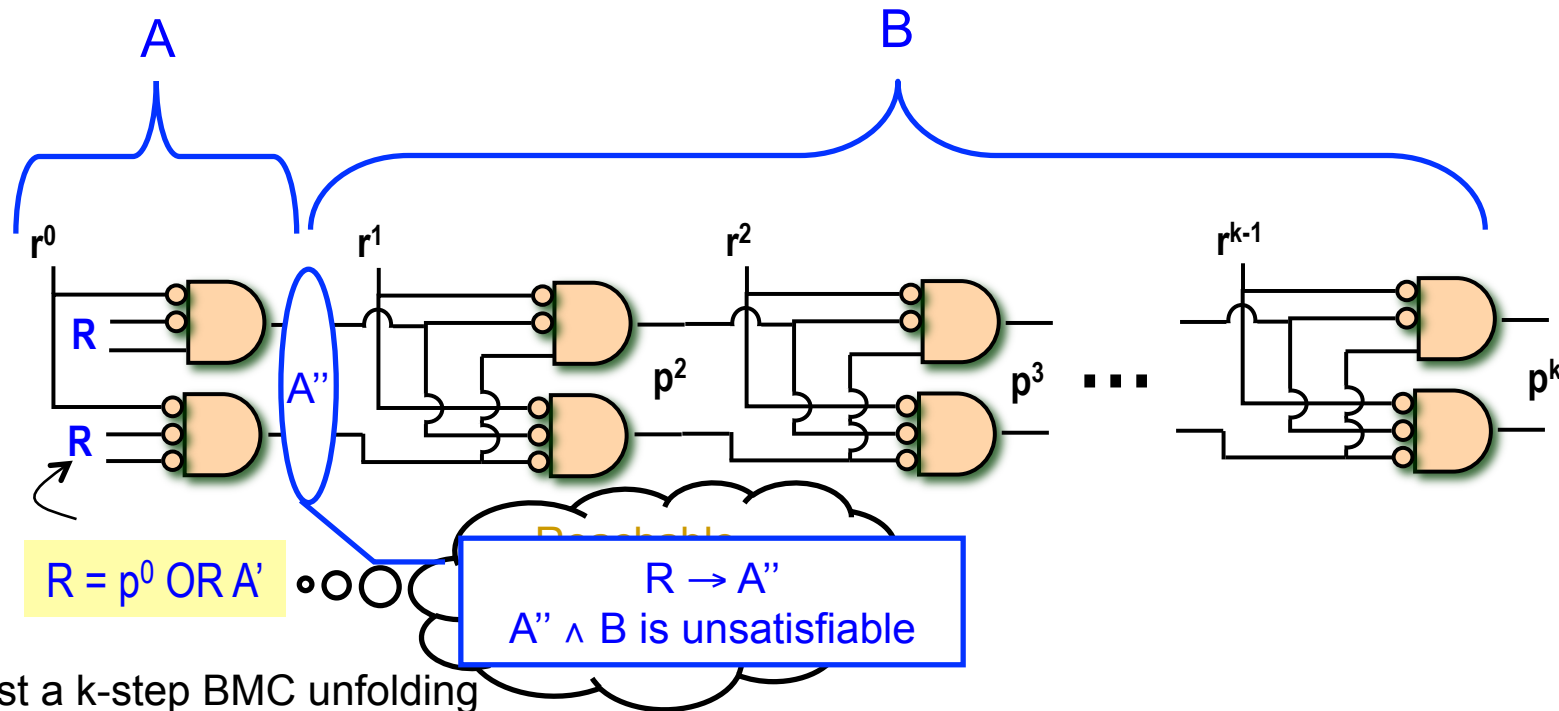


1) Cast a k-step BMC unfolding

2) If unsatisfiable, extract an **interpolant A'** from SAT proof

- A' : an overapproximate *image* of initial states
 - Key efficiency: avoid \exists since A' refers only to cut between A , B

3) Is A' **contained in** initial states? If so, A' is an invariant \rightarrow **proven!**



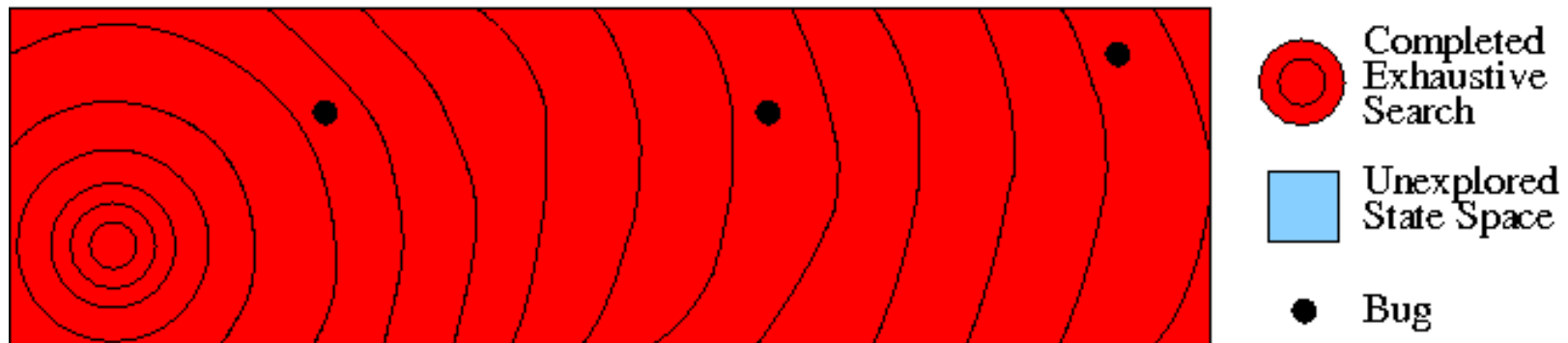
- 1) Cast a k-step BMC unfolding
- 2) If unsatisfiable, extract an interpolant A' from SAT proof
- 3) Is A' contained in initial states? If so, A' is an invariant \rightarrow **proven!**
- 4) **Add A' to initial states**; repeat BMC from resulting R
- 5) Result unsat? Goto 2) to compute another image A''
- 6) Else increase k and goto 1)

Invariant Generation

- Numerous techniques to *eagerly* compute invariants
 - Gate equivalences, gate implications, ...
 - Invariants may be *assumed* to tighten inductive states toward reachable
- And *lazy* techniques to derive invariants relevant to property
 - Bradley's IC3, on-demand checking of overapproximate counterexamples, ...
- Also techniques to strengthen the property being proven, or prove conjunctions of properties
 - Causes **induction hypothesis** to become stronger

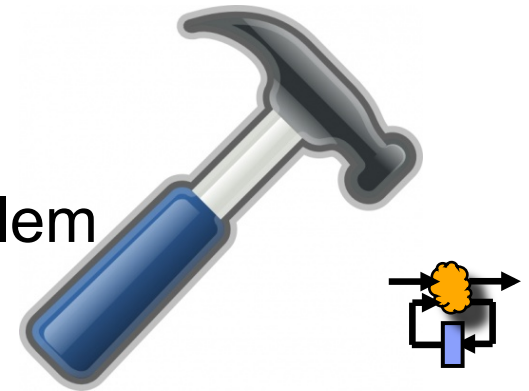
BDD-Based Reachability

- 1) Build a Transition Relation TR using BDDs
 - Current State \times Current Input \rightarrow Next State $CS \times CI \rightarrow NS$
- 3) Build a BDD S_0 for Initial States; set $i = 0$
- 4) Compute an *image*: $S_{i+1} = \exists CS, CI. (TR \wedge S_i)$; swap NS for CS variables
- 5) S_{i+1} asserts a property? **Fail**
- 6) S_{i+1} contained in $R = \{S_0, \dots, S_i\}$? **Pass**
- 7) Increment i ; goto 3

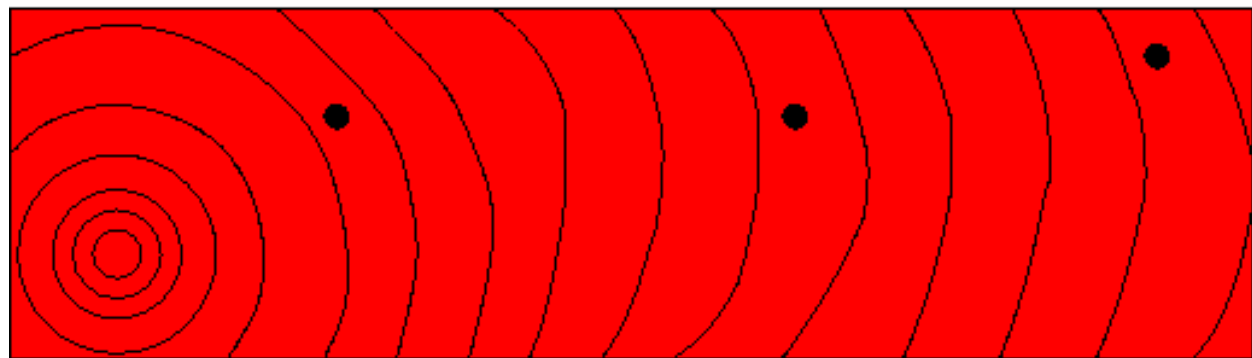


BDD-Based Reachability

- Memory-intensive algorithm for PSPACE problem
 - Prone to memout above a few hundred state variables
- Though a *reliable* proof technique for small enough netlists
- Power of \exists renders this adept at *deep* counterexamples



■ *BDDs are not dead! A critical algorithm in practice!!*



- Completed Exhaustive Search
- Unexplored State Space
- Bug

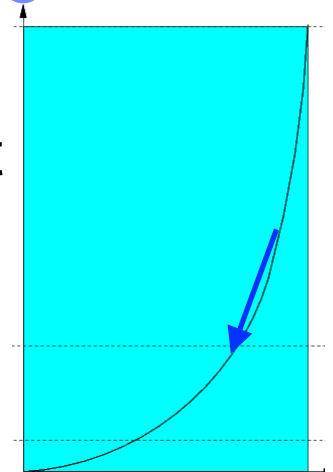
Outline

- Hardware and Hardware Modeling
- Hardware Verification and Specification Methods
- Algorithms for Reasoning about Hardware
 - Falsification Techniques
 - Proof Techniques
 - Reductions

Coping with Verification Complexity: *Reductions*

1) Transforms may **reduce** gate + state variable count

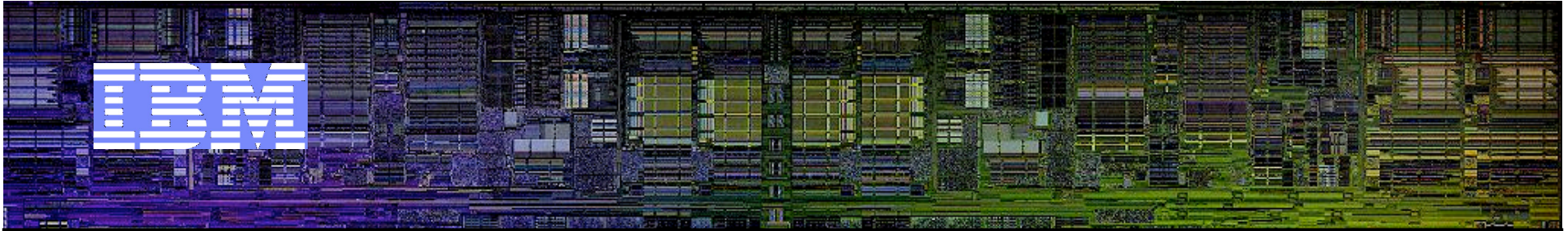
- Verification complexity may run exponential in gate count
- Gate reductions may dramatically reduce verification resources



3) Reductions are critical to eliminate circuit artifacts which preclude well-suited verification algos

5) Transforms often *enable* lightweight algos to be conclusive

- E.g., induction (NP) vs reachability computation (PSPACE)
- May set the problem on a *less-intractable curve!*



Summer Formal 2011

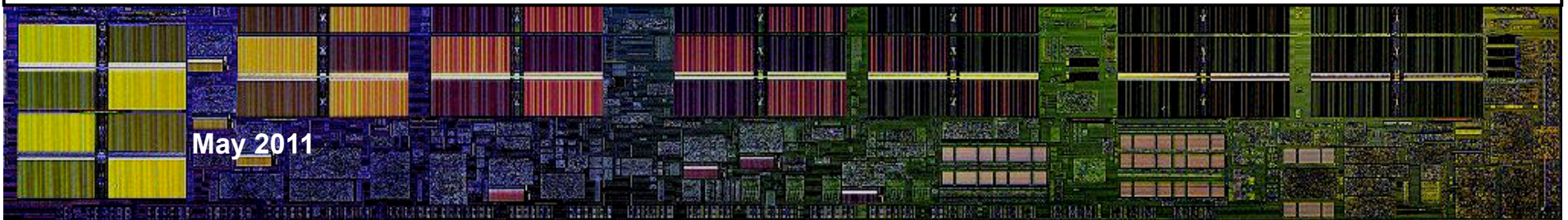
Hardware Verification 2011

Hardware Verification Challenges and Solutions

Jason Baumgartner

www.research.ibm.com/sixthsense

IBM Corporation



Outline

■ Class 1: Hardware Verification Foundations

- Hardware and Hardware Modeling
- Hardware Verification and Specification Methods
- Algorithms for Reasoning about Hardware

■ Class 2: Hardware Verification Challenges and Solutions

- Moore's Law v. Verification Complexity
- Coping with Verification Complexity via Transformations

■ Class 3: Industrial Hardware Verification In Practice

- Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies

Outline

■ Hardware Verification Challenges

- Moore's Law v. Verification Complexity

■ Coping with Verification Complexity via Transformations

■ Example Transformations

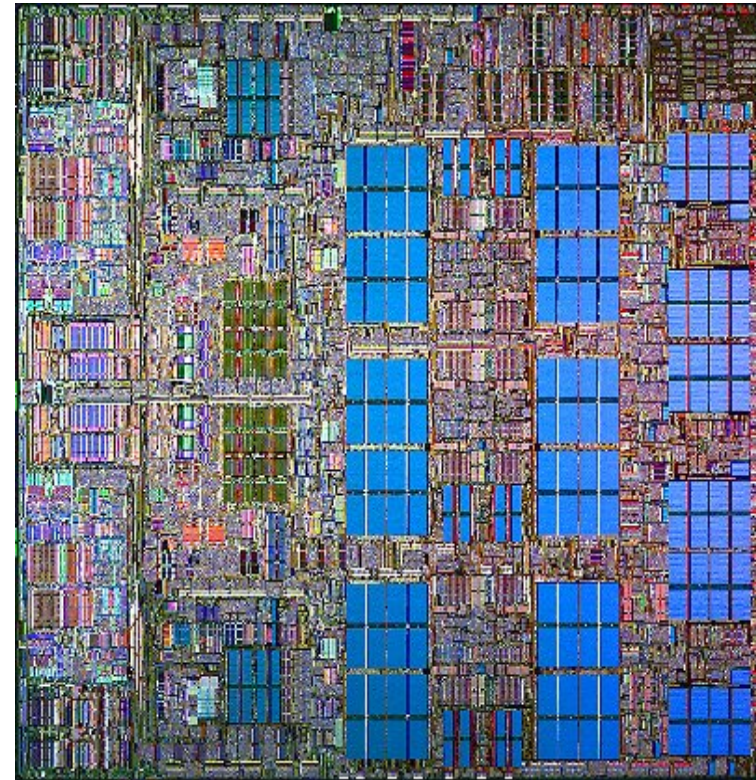
■ Benefits of TBV

POWER5 Chip: *It's Ugly*

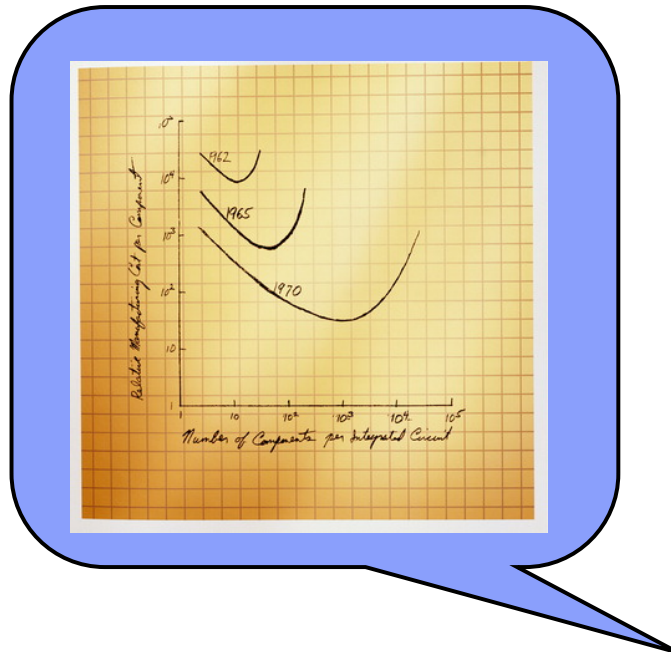
- Dual pSeries CPU
- SMT core (2 virtual procs/core)
- 64 bit PowerPC
- 276 million transistors
- 8-way superscalar
- Split L1 Cache (64k I & 32k D) per core
- 1.92MB shared L2 Cache >2.0 GHz
- Size: 389 sq mm
- 2313 signal I/Os
- >>1,000,000 Lines HDL

POWER architecture:

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



Moore's Law: *And Getting Uglier!*



transistors per IC for minimum cost has increased at roughly a factor of two per year



there is no reason to believe it will not remain nearly constant for at least 10 years

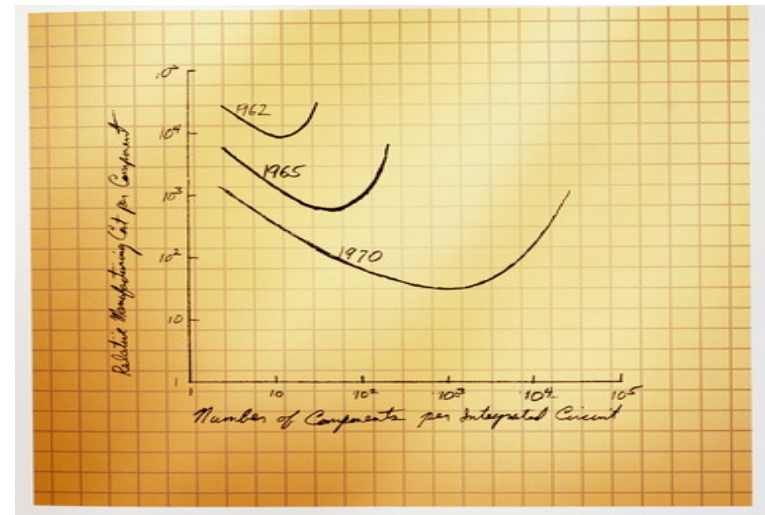
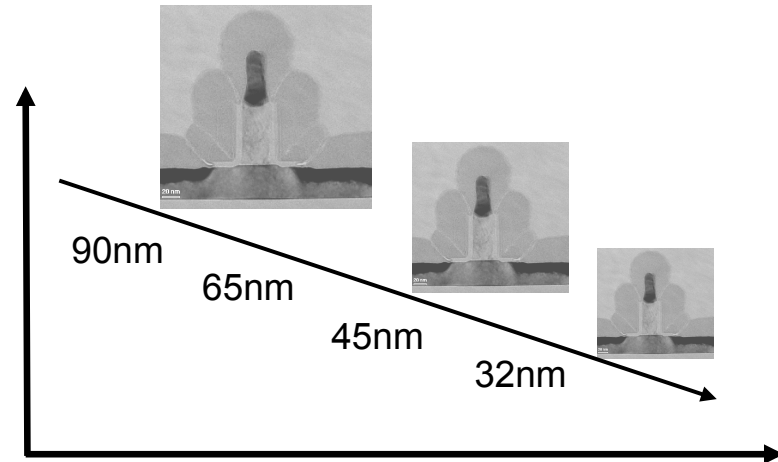
G. Moore, "Cramming More Components Onto Integrated Circuits," Electronics Magazine 1965

Design Characteristics of “Moore’s Law”

- **Smaller: *miniaturization***
 - Devices and transistors

- **Cheaper**
 - Per *transistor*
 - Not necessarily per *product*

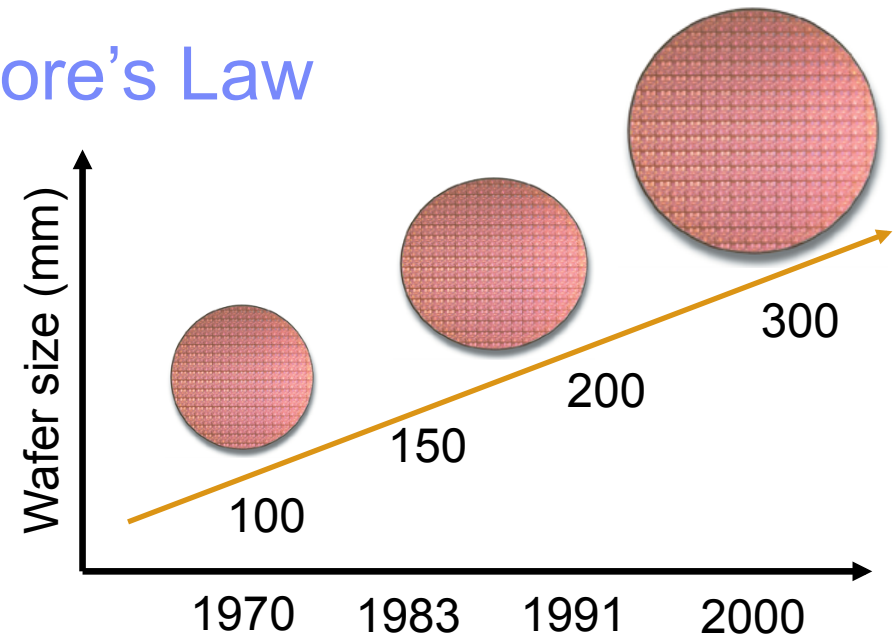
- **Faster**
 - If software developers are willing :-)



Design Characteristics of Moore's Law

- **Bigger!**

- Chips, wafers, ICs, networked systems, ...



- **More complex!!**

- More “smaller” devices crammed onto chip / IC
- Scale-up of functionality: datawidth, memory size, ...

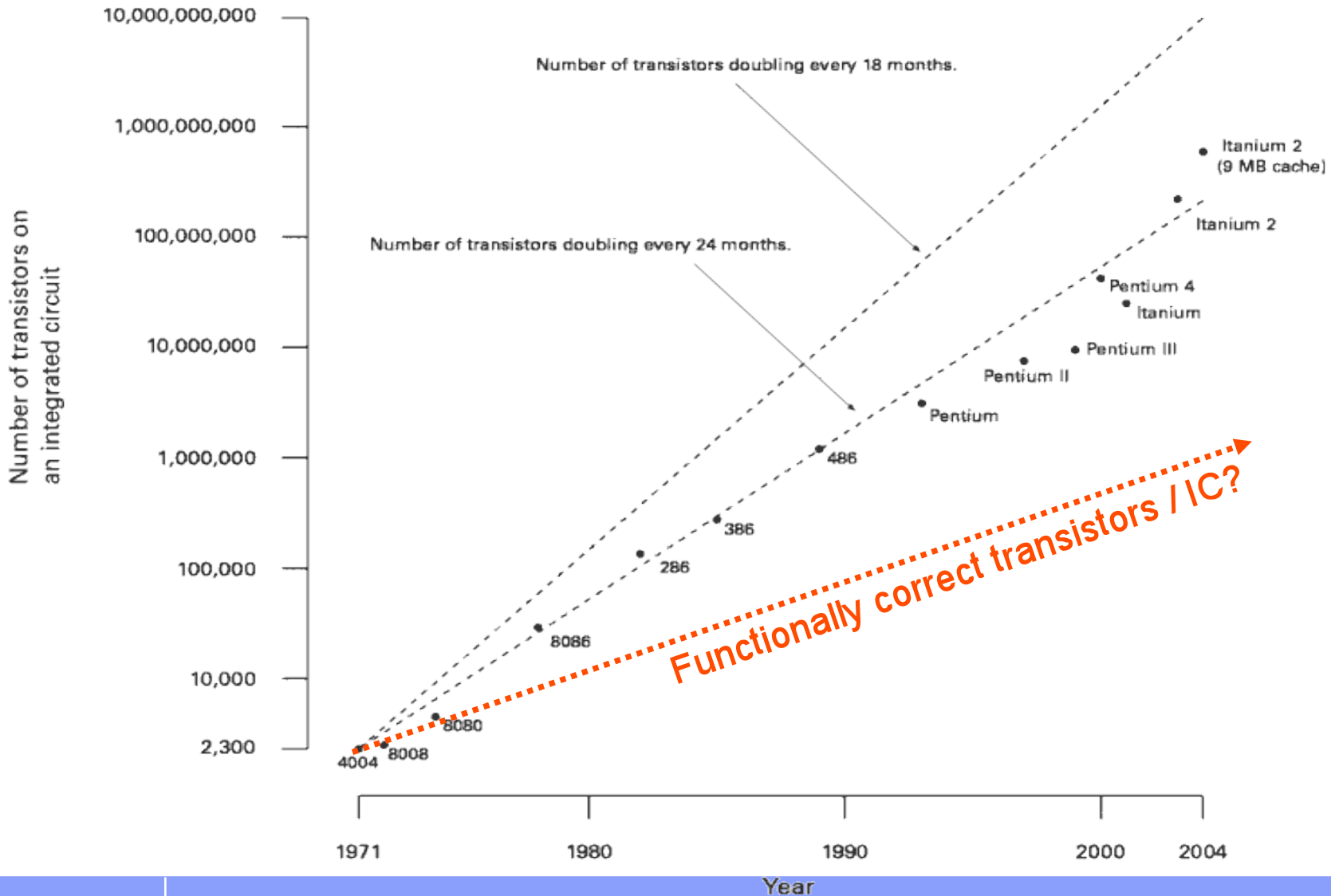
- **Hotter!!!**

- Since bigger & faster!

Design Characteristics of Moore's Law

- Comparable functionality in smaller / cheaper package?
- No! Cram *more* into a bigger package
- Harder to verify!!! ??? !?*\$%*#@!!
 - Thankfully, “device complexity” cannot *afford* to be “as great as possible”

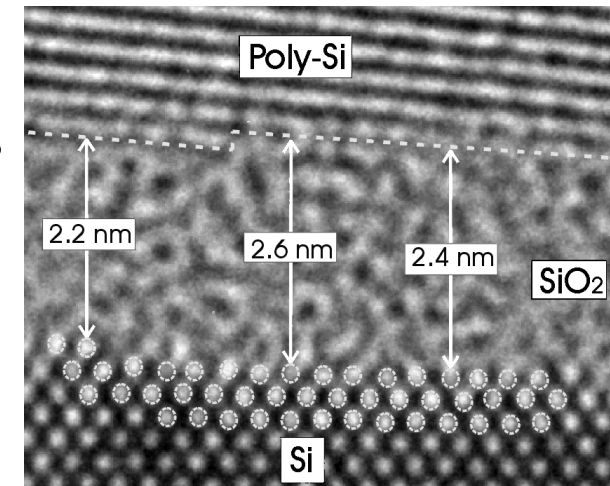
Longevity: for at least 10 years, indeed!



The End is Near! (Is it?)

- *Moore himself* was one of his harshest critics

- Disappearing “circuit cleverness” 1975
- Lack of demand for VLSI 1979
 - *The Death of Cost Effectiveness*



- *No exponential is forever* - must hit limitations of physics?

- Can we miniaturize (and design w.r.t.) quantum particles? Hmmm...

- Note trends on massive parallelization (e.g. BlueGene), 3D chips, biological computing, quantum computing, ...

- Who knows?
- Will global warming (or out-of-control particle accelerator) finitize “forever”?

Moore's Law

- Attributed to virtually all exponentially-growing computing metrics
 - Circuit speed
 - Computing power (MIPS, GFlops, ...)
 - Storage capacity
 - Network capacity
 - Pixel density
 - ...

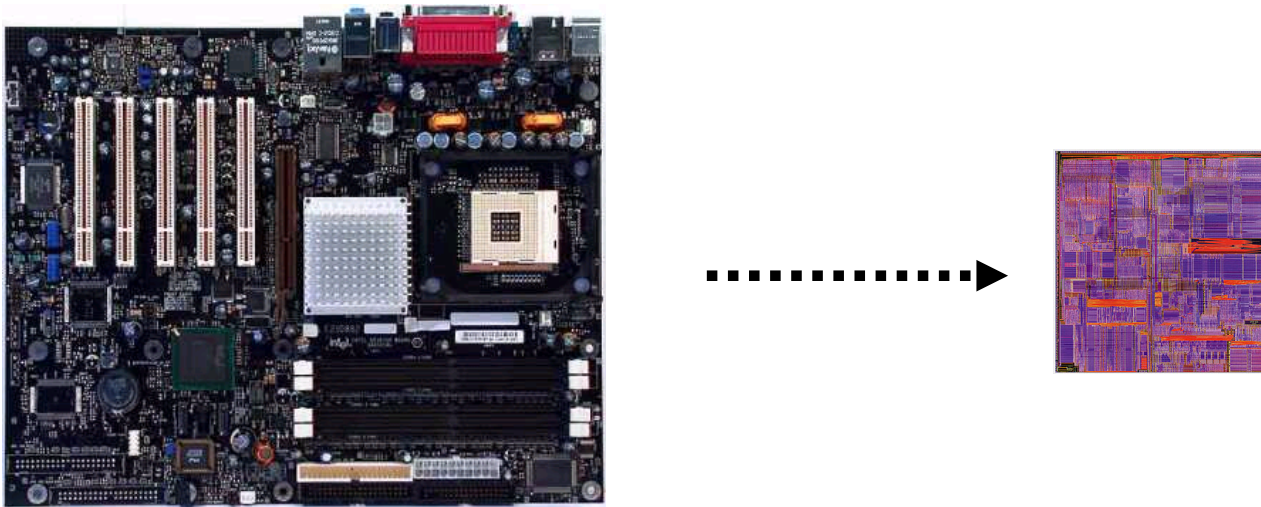
- Strictly speaking, these are *not* part of Moore's original observation
- Though all refer to the same trend; ***we abuse notation herein***

Moore's Law v. Verification Complexity

- # Components per IC doubles every ~2 years
- Verification thus *appears* to grow exponentially more complex
 - Compounded by use of *today's* computers to verify *tomorrow's* designs
- Is this *necessarily* the case?
- Let us revisit how this capacity tends to be used
 - Moore's *Heirlooms*
 - Where we are now, where we are going, and why

Moore's Heirlooms: *Integration*

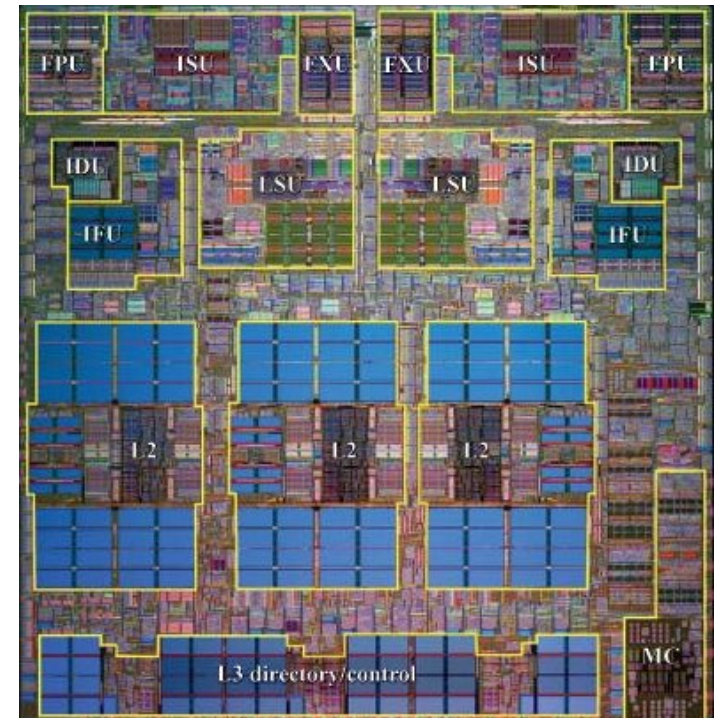
- Integration of more devices on chip
 - System on a Chip: more components+functionality moved on-chip
 - Caches are moving on-chip



- Lowers packaging costs and power, increases speed
- “Moving” components: *simplifies* verification complexity

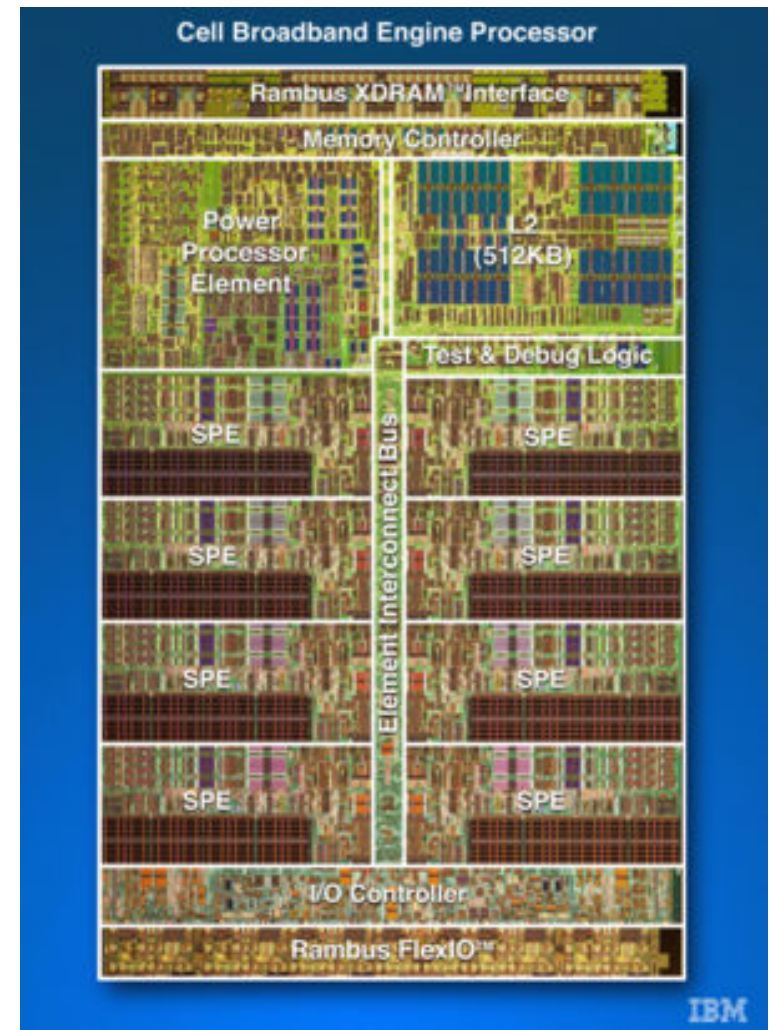
Moore's Heirlooms: *Modularity*

- Additional execution units
 - Multiple FPUs, FXUs, LSUs, ...
- Additional cores
 - POWER4 is 2 core; POWER7 is 8 core
- No additional *component* verification complexity
- Overall *system* complexity may increase
 - Hardware, software, or both
 - More concurrency, # interfaces
 - Some aspects may be covered by higher-level verification
 - More complex communication protocols



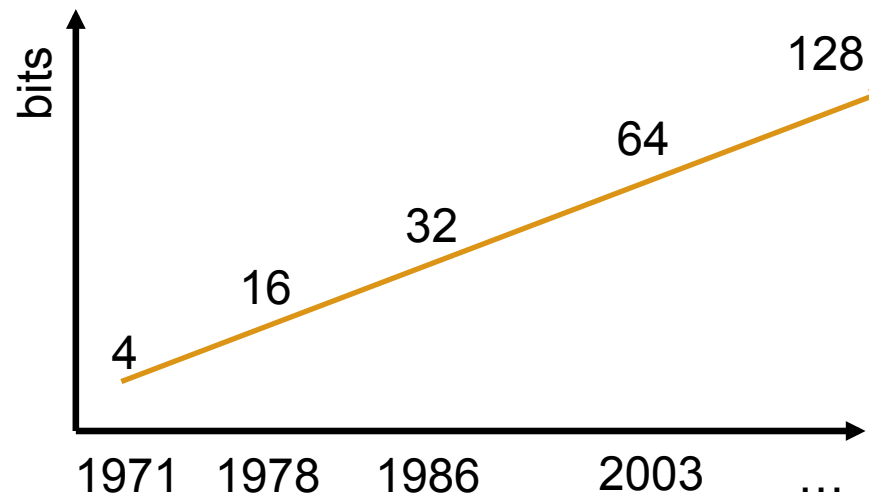
Moore's Heirlooms: *Specialized Hardware*

- SW function moves to hardware
 - Vector units, encryption, compression
- *Diversified* modularity
 - Cell processor: 8 Synergistic Processing Elements in addition to a Power processor
- May not increase verif complexity
 - “Only” *more* components to verify
- Though such HW is often *difficult to verify!*
 - Bugs more expensive to fix in HW than SW
 - HW tends to be bit-optimized
 - Move from SW to HW may hurt verification



Moore's Heirlooms: *Increased Operand / Data Width*

- Operand width has grown substantially
 - Mainstream (vs *mainframe!*) processors



- Many processors have emulated 128-bit data support for decades
 - SW + specialized HW atomically manages narrower computations

Moore's Heirlooms: *Increased Operand / Data Width*

- Does increased data width increase verification complexity?
 - Sometimes “no” !!!

- Data routing checks are not necessarily more complex
 - Some checks may be **bit-sliced**; *linear* verification scaling
 - **Word / vector** reasoning techniques scale well **when applicable**
 - UCLID, SMT, uninterpreted functions
 - Verification **reduction techniques** have been proposed to automatically shrink widths to facilitate a broader set of algorithms
 - Control / token nets, small models, domain reduction (Bjesse CAV'08),...

Moore's Heirlooms: *Increased Operand / Data Width*

- Does increased data width increase verification complexity?
 - Sometimes “yes” !!!
- What about correctness of *computations* on the operands?
 - *Optimized* arithmetic / logical computations are not simple + = * / < >
- Arithmetic is often bit-optimized in HDL itself
 - Limited by higher-level synthesis optimality
 - Limited by prevalent CEC methodology

Moore's Heirlooms: *Increased Operand / Data Width*

■ Consider IEEE Floating Point Spec: $S * 2^E$

- S: *Significand*, e.g. 3.14159
- E: *Exponent*, represented relative to predefined *bias*

	Single Precision	Double Precision	Quadruple Precision
Width	32	64	128
Exponent bits	8	11	15
Significand bits	23	52	112

■ Bit-level solvers often requires case-splitting on exponent values

- Practically ~3 orders-of-magnitude #cases increase *double-to-quad*
 - Each *quad* case is dramatically more complex than *double*
- *Double* is already **computationally expensive!!**

Moore's Heirlooms: *Increased Operand / Data Width*

- Error Code Detection / Correction (ECC) logic becomes substantially more complex w.r.t. data width
 - Byproduct of transistor miniaturization: *soft errors!*
 - Increasingly mandate ECC logic
 - Along with increasingly elaborate ECC algos to handle *more* error bits

- Emerging encryption HW similarly explodes in complexity w.r.t. data width

Moore's Heirlooms: *Increased RAM Depth*

- Often *not* a substantial cause of verification complexity
 - Most of the design is insensitive to this metric

- Verification algorithms can often treat such arrays more abstractly with *memory consistency constraints*
 - Efficient Memory Model, BAT ICCAD'07, Bjesse FMCAD'08
 - *Though bit-blasted reasoning becomes **much** more complex*

- Though with larger caches and *more elaborate associativity schemes* comes increased complexity
 - Sometimes the logic *adjacent to* memory array becomes more complex

Moore's Heirlooms: *Circuit and Device Cleverness*

- Countless tricks behind increasing MIPS and computing power
 - *Some of these are HUGE causes of verification complexity*
- Categorizable as circuit speedups vs algorithmic speedups
- First consider techniques for circuit speedups
 - Integration, interconnect speedup, miniaturization, datapath widening all eliminate speed barriers
 - Natural push to *speed up* core processing circuitry
 - How is this achieved?

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-optimized 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, ...

Simplicity vs Performance

```
begin
  res := default(queue_valid);
  res := shift(dequeue,res);
  res := enqueue(ls_tmq_cand,res);
  return res;
end;
```

This high-level initial VHDL did not have suitable timing; rewritten into...

Simplicity vs Performance

```

res.ctl.ec(0).save(0) := enqueue(0) and (not hold(0) and not valid(1));
res.ctl.ec(0).save(1) := enqueue(1) and not enqueue(0) and (not hold(0) and not valid(1));
res.ctl.ec(1).save(0) := enqueue(0) and (valid(1) xor hold(0));
res.ctl.ec(1).save(1) := (not enqueue(0) and not hold(1)) or (not valid(1) and not hold(0));
res.ctl.ec(0).hold := hold(0);
res.ctl.ec(1).hold := hold(1);
res.ctl.shift := valid(1) and not hold(0);
res.val(0) := (enqueue(0) or enqueue(1)) or (valid(1) or hold(0));
res.val(1) := (enqueue(0) and enqueue(1)) or ((enqueue(0) or enqueue(1)) and (valid(1) or hold(0))) or hold(1);
res.ctl.rej(0) := enqueue(0) and hold(1);
res.ctl.rej(1) := enqueue(1) and (hold(1) or (enqueue(0) and (valid(1) or hold(0))));
res.write(0) := (enqueue(0) or enqueue(1) or valid(1)) and not hold(0);
res.write(1) := ( enqueue(0) and enqueue(1) and not hold(1) ) or ( (enqueue(0) or enqueue(1)) and (valid(1) xor hold(0)) );

```

Quite complex to develop, maintain, verify!

Industrial hardware verification is often more complex than it intuitively should be
*Luckily, circuit speedups often coped with by **Transformation-Based Verification***

State-of-the-Art Hardware Model Checkers, Equiv Checkers

- Leverage a diversity of algos for scalability + ***robustness against implementation details***
 - 1) Various proof + falsification algos: SAT- and BDD-based, explicit search
 - 2) Synergistic transformations and abstractions
 - *Phase abstraction* copes with intricate clocking, latching schemes
 - *Retiming* eliminates overhead of pipelined designs
 - *Redundancy removal, rewriting* are key to equiv checking, eliminating high-performance hardware details pathological to verification algos
 - ...
- *HW implementation details pose barriers to well-suited algos*
 - Good for circuit performance; bad for verification!

High-End Processor Verification: Word-Level Techniques

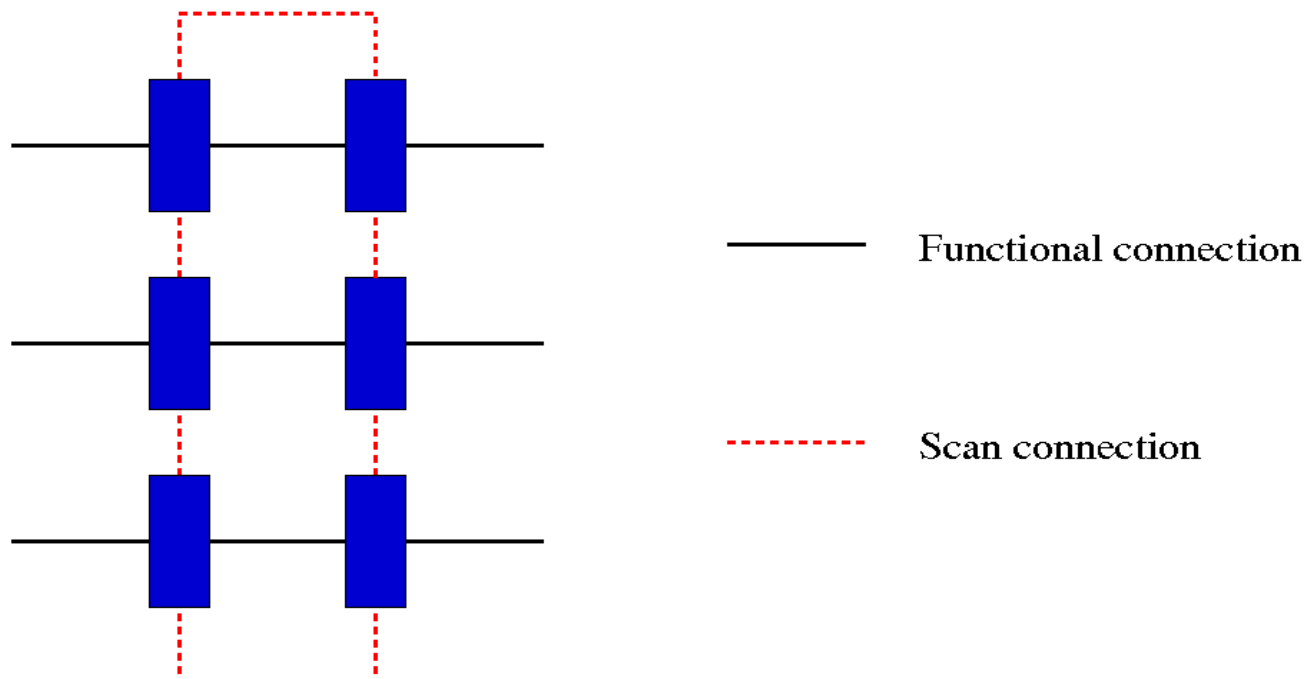
- Numerous techniques have been proposed for Processor Verification
- Satisfiability Modulo Theories replaces word-level operations by function-preserving, yet more abstract, Boolean predicates
 - Replace complex arithmetic by arbitrary simple (uninterpreted) function
 - Reduce arithmetic proof to simpler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

High-End Processor Verification: Word-Level Techniques

- **Difficult 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
 - Aggressive clock frequencies may require pipelining of comprehensible functions
 - Often a large amount of intricate bit-level logic intertwined with word-level operations
- Abstractions may miss critical bugs
 - Removal of bitvector nonlinearities are lossy
 - May miss bugs due to rounding modes, overflow, ... if not careful
- Will sequential synthesis become strong enough to enable abstract design?
 - Also mandates strong *sequential equivalence checking*
- Or need to manually create an abstract reference model??

High-End Processor Verification: Non-Functional Artifacts

- CEC methodology forces HDL to acquire circuit characteristics
 - Word-level operations, isomorphisms broken by *test* logic
 - Run-time monitoring logic, error checker / recovery, entails similar complexities
 - Reused for functional obligations: initialization, reliability, ...
 - May be suppressed for functional verif, though must be verified somehow



High-End Processor Verification: Algorithmic Speedups

- Many optimizations are beyond **automated synthesis** capability
 - Superscalar + out-of-order execution
 - Prefetching
 - Speculative execution
 - Reconfigurable hardware
 - General power optimizations

- And *quite* difficult to verify
 - Often require clever **manual** strategies for scalability
 - Many violate SEC paradigm
 - No **synthesis history** paradigm to automate / simplify SEC-style verification

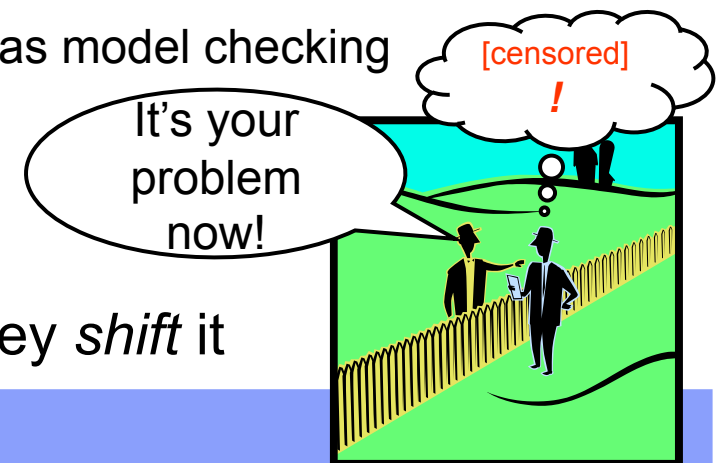
- Would complicate reference models as well!
 - Else reference model *too leaky* to expose intricate bugs!



SOTA Hardware Model Checkers, Equiv Checkers

- ***Implementation details may preclude well-suited algos***
- Robust tools *must* automatically cope with implementation details
- Behavioral reference models facilitate functional verif, though:
 - 1) Creating reference models is often prohibitively expensive
 - Semiconductor industry is forever pushing for reduced costs!
 - 3) Even if available, *equiv checking* must be used to validate ref vs imp
 - Sequential equiv checking as complex as model checking

- Ref models don't *solve* the problem; they *shift* it



Complexities of High-End Processors

- Industrially, the RTL is often the verification target
 - Ref models are rarely maintained
- Many intricate hardware artifacts that convolute “simple” behavior
- Industrially, “Formal Processor Verification” refers to proc components
 - E.g., verification of FPU, Cache Controller, Branch Prediction Logic
- Automated “Dispatch to Completion” proofs for processors as complex as Pentium, POWER, ... are **intractable** given today’s technologies

Outline

■ Hardware Verification Challenges

■ Moore's Law: *What's Next?*

■ Coping with Verification Complexity via *Transformations*

■ Example Transformations

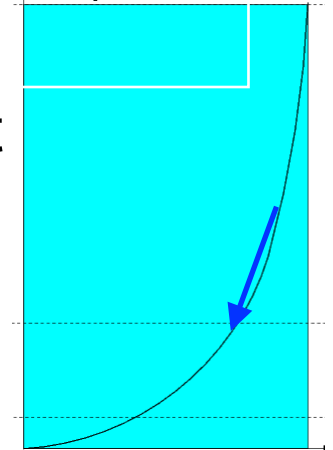
■ Benefits of TBV

Coping with Verification Complexity via Transformations

■ Key motivations for Transformation-Based Verification (TBV)

1) Transforms may **reduce** gate + state variable count

- Verification complexity may run exponential in gate count
- Gate reductions may dramatically reduce verification resources



3) Reductions are critical to eliminate circuit artifacts which preclude well-suited verification algos

- TBV is *essential* to high-end design verification
- *Phase abstraction* copes with intricate clocking, latching schemes
- *Retiming* eliminates overhead of pipelined designs
- *Redundancy removal* eliminates pervasive logic irrelevant to a given check

Coping with Verification Complexity via Transformations

3) Transforms often *enable* lightweight algos to be conclusive

- E.g., induction (NP) vs reachability computation (PSPACE)
 - May set the problem on a less-intractable curve!
- And many transforms are polynomial-time or resource-boundable

5) Transforms alone may trivialize SEC problems

- A retiming engine may reduce a retimed SEC problem to CEC
- Speculative reduction automates SEC decomposition
- And – virtually all engines useful for SEC \leftrightarrow useful for model checking
- Paradigms occasionally blur depending on nature of the specification

Coping with Verification Complexity via Transformations

5) Transforms are ***synergistic***

- Synthesis-oriented synergies have been known for more than a decade
 - *Retiming and resynthesis:*
 - Resynthesis enables more optimal register placement for retiming
 - Retiming eliminates “bottlenecks” for combinational resynthesis
- Many verification-oriented transforms have been discovered more recently
 - Localization, input elimination, temporal abstractions, ...
- ***Finding the proper sequence of transforms may be key to an automated proof***

Coping with Verification Complexity: TBV Motivations 4

6) Verification algos are *essential* to many reduction engines

- **Redundancy removal** often uses induction to prove **equivalences**
- Though sometimes requires interpolation, or reachability analysis, or ...
- And – other transforms may be critical to reduce an **equivalence proof** to be tractable for a verification algo

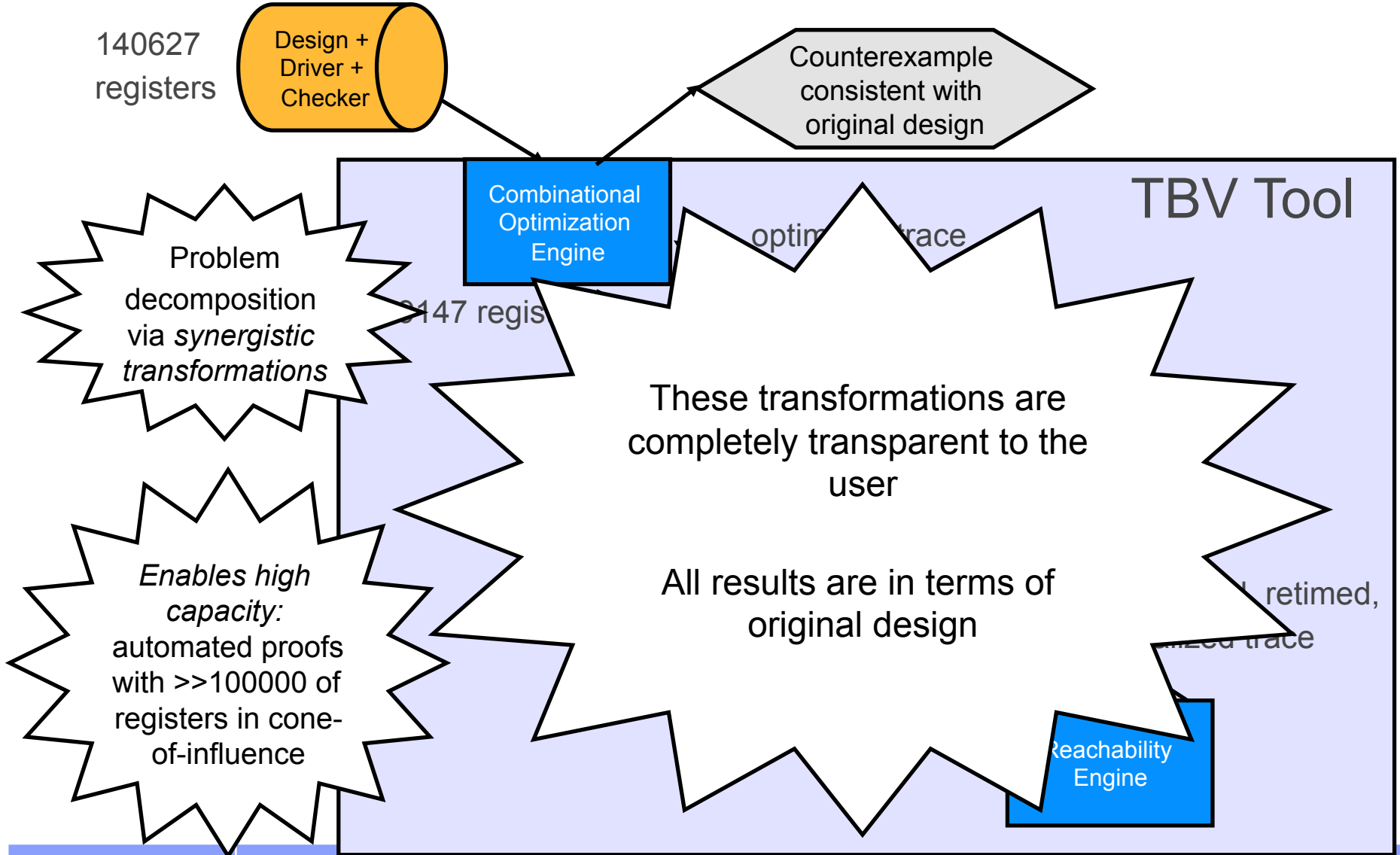
■ Scalable *verification* requires *synthesis* i.e. transforms

■ Effective *synthesis* requires *verification* algos

Transformation-Based Verification

- Encapsulates engines against a *modular API*
 - Transformation engines, proof engines, falsification engines
- Modular API enables *maximal synergy* between engines
 - Each (sub)problem may be addressed with an arbitrary sequence of algos
 - Every problem is different; different algorithm sequences may be exponentially more / less effective on a given problem
- **Incrementally chop complex problems into simpler problems, until tractable for core verification algos**

Transformation-Based Verification



Example Engines

- *Combinational rewriting*
- *Sequential redundancy removal*
- *Min-area retiming*
- *Sequential rewriting*
- *Input reparameterization*
- *Localization*
- *Target enlargement*
- *State-transition folding*
- *Circuit quantification*
- *Temporal shifting + decomp*
- *Isomorphic property decomp*
- *Unfolding*
- *Speculative Reduction*
- *Symbolic sim: SAT+BDDs*
- *Semi-formal search*
- *Random simulation*
- *Bit-parallel simulation*
- *Symbolic reachability*
- *Induction*
- *Interpolation*
- *Invariant generation*
- ...
- *Expert System Engine may automates optimal engine selection*

Outline

■ High-End Hardware Verification Challenges

- Moore's Law: *What's Next?*

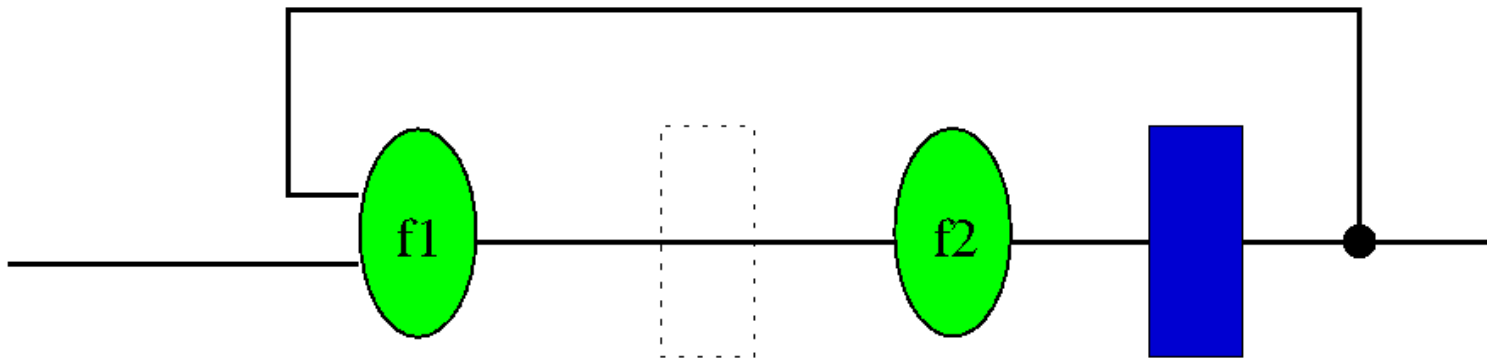
■ Coping with Verification Complexity via *Transformations*

■ Example Transformations

■ Benefits of TBV

Example Transform 1: Phase Abstraction

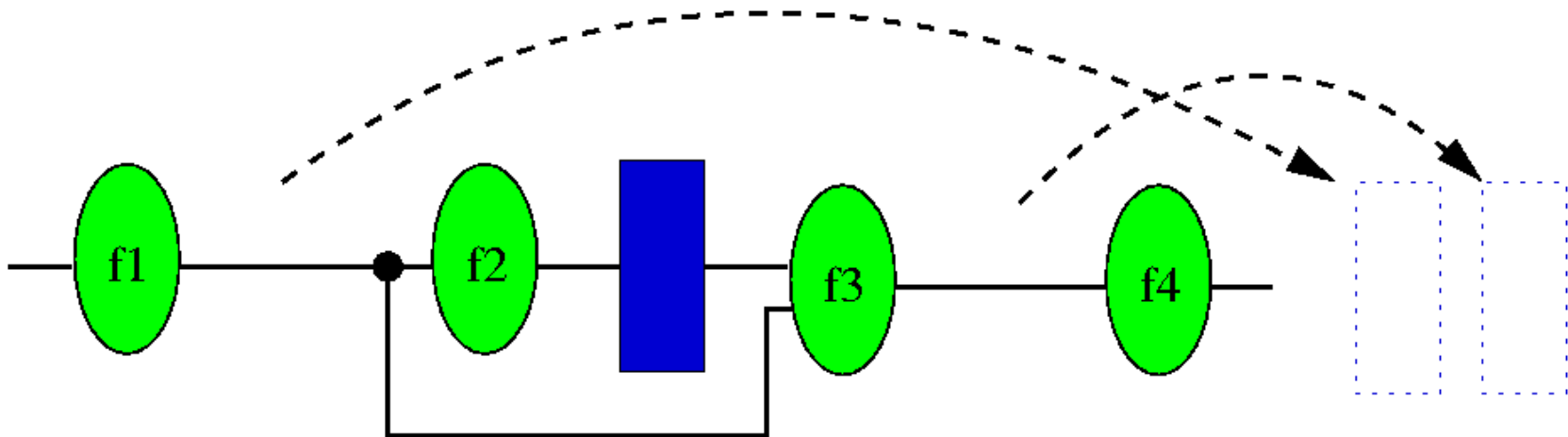
- Multi-phase latching often hurts verification
 - Increase in state element count, correlation; increase in *diameter*
- *Phase abstraction* eliminates this overhead
 - Unfold next-state functions modulo-2



- ~50% state element reduction on multi-phase designs
- Polynomial-time algorithm

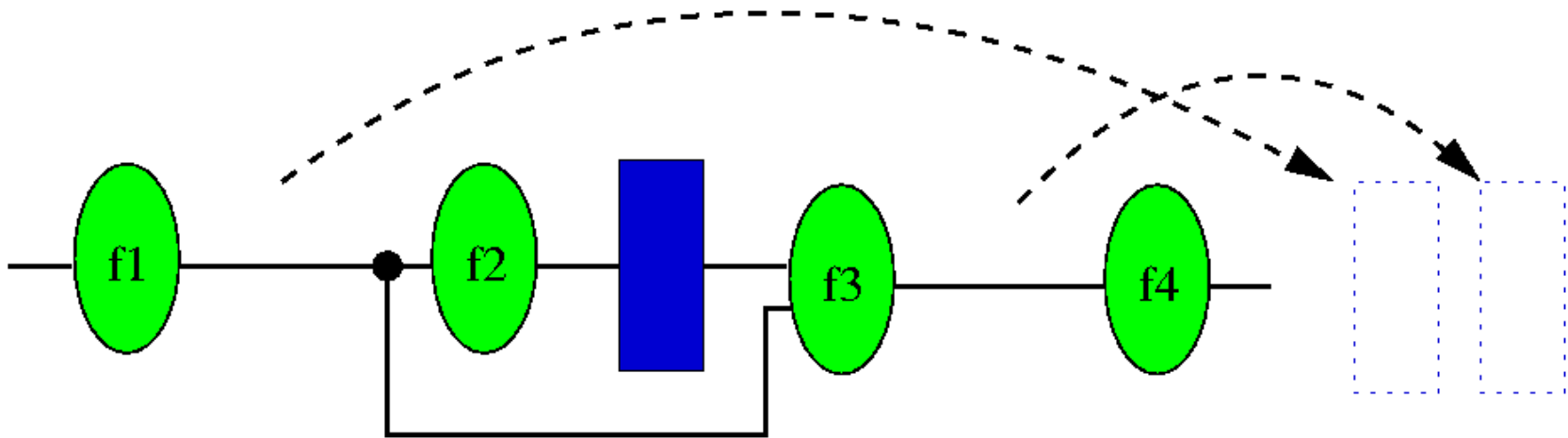
Example Transform 2: Retiming

- Retiming eliminates state elements by moving them across gates
 - Moving a state element across a gate *time-shifts* its behavior
- Very effective at reducing overhead of pipelined designs
 - 62% reduction attained on POWER4 netlists CAV2001
 - May require *phase abstraction* to enable these reductions



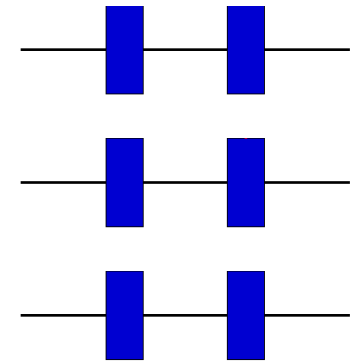
Example Transform 2: Retiming

- Min-area retiming may be cast as a min-cost flow graph algorithm
 - Solvable in polynomial time

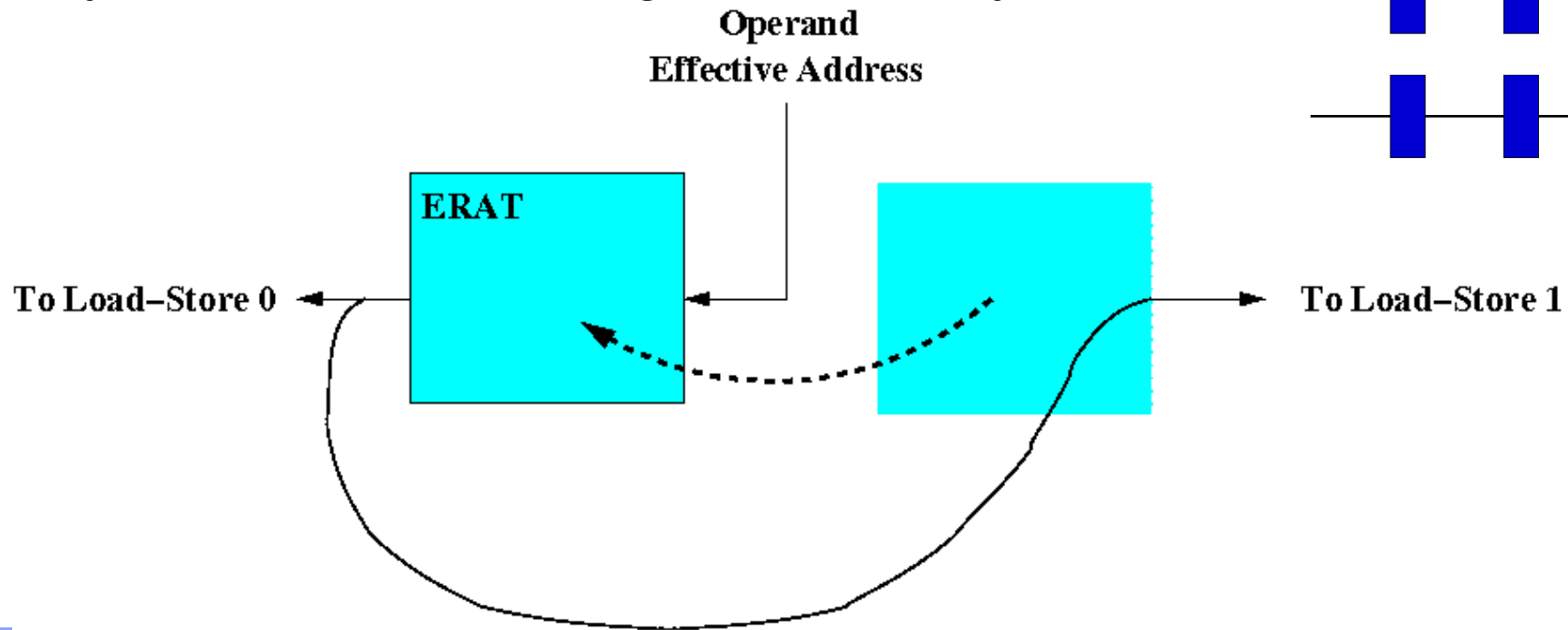


Example Transform 3: Redundancy Removal

- Redundancy is prevalent in verification testbenches, e.g.:
 - Deliberate logic replication to reduce delay (due to placement issues)
 - Disabled *pervasive logic* such as scan-chains
 - Redundancies between design + checker



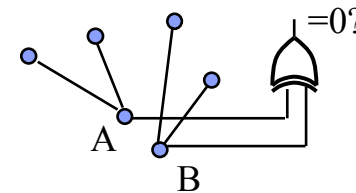
- May be eliminated through *redundancy removal*



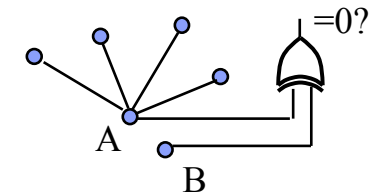
Example Transform 3: Redundancy Removal

■ *Speculative reduction* for sequential redundancy removal

- 1) Guess sets of *redundancy candidates*
- 2) Create a netlist to validate that redundancy
 - Assume all redundancies are correct: *speculative reduction*
 - Add *miters* (XORs) to *validate* assumptions
- 3) Attempt to prove miters as constant 0
- 4) If successful, exit with identified redundancies
- 5) Else, refine classes; goto step 2



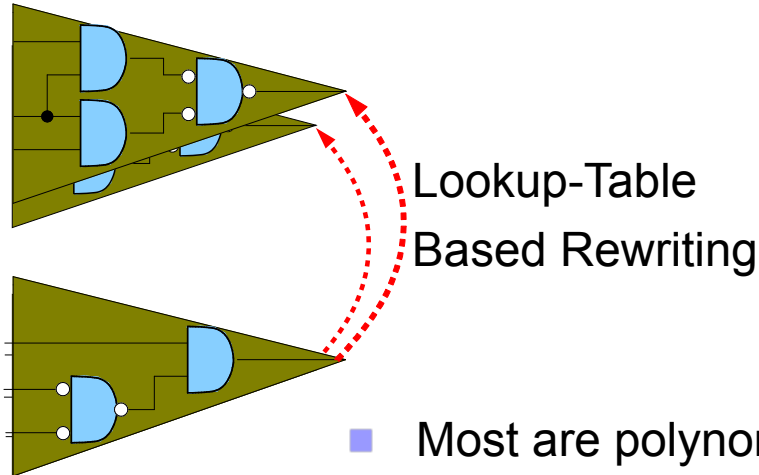
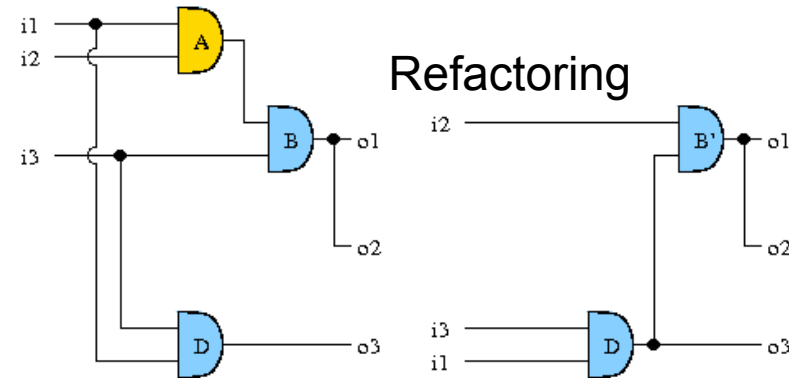
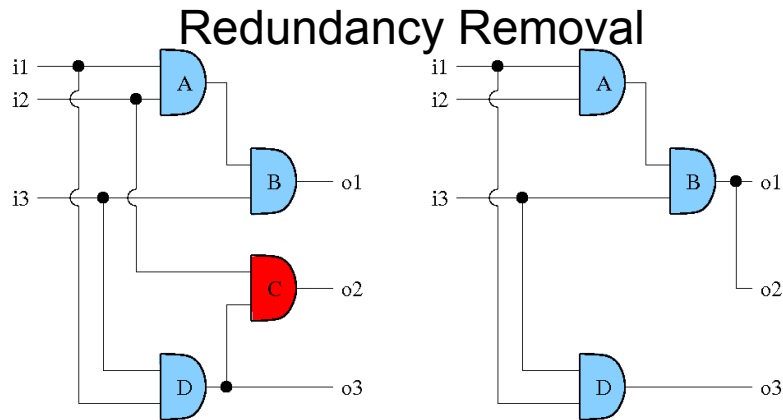
Miter without spec reduction



Miter with spec reduction

- ### ■ While relying upon proof techniques (often induction), may be resource-bounded trading optimality for runtime

Example Transform 4: Combinational Rewriting

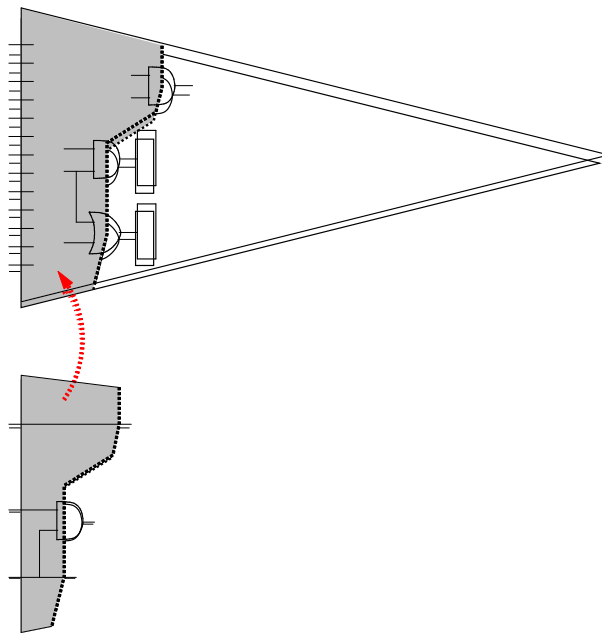


Also:
 Ternary-simulation based reductions
 ODC-based reductions
 "Dependent register" reductions
 BDD-based rewriting
 ...

- Most are polynomial-time, others are resource-boundable
- Often capable of ~50% combinational logic reductions
- Synergistically improves reductions capability of retiming

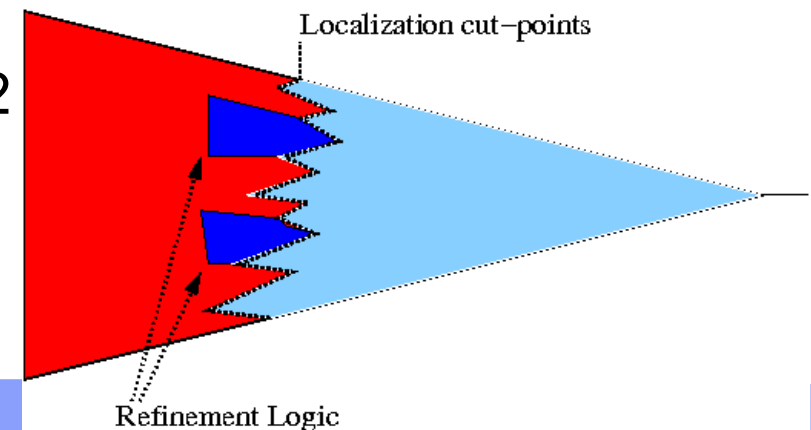
Example Transform 5: Input Reparameterization

- Abstraction technique to reduce input count
 - Identify a min-cut between primary inputs and next-state logic
 - Compute *range* of cut: function over *parametric variables* (\exists using BDDs)
 - Replace original cut by synthesis over *parametric inputs*
- Preserves design behavior; resource-boundable



Example Transformation 6: CEGAR-Based Localization

- Abstraction that reduces netlist size via cutpointing internal gates
- *Counterexample-Guided Abstraction Refinement*
 - 1) Begin with an arbitrary small abstraction
 - 2) Perform verification on the abstraction
 - 3) Proof obtained on abstract model? **Pass**
 - 4) Counterexample found? Check if valid w.r.t. original netlist
 - Yes? **Fail**
 - No? Refine the abstraction; goto 2



Example Transformation 6: Proof-Based Localization

- Abstraction that reduces netlist size via cutpointing internal gates

- 1) Perform BMC on original netlist

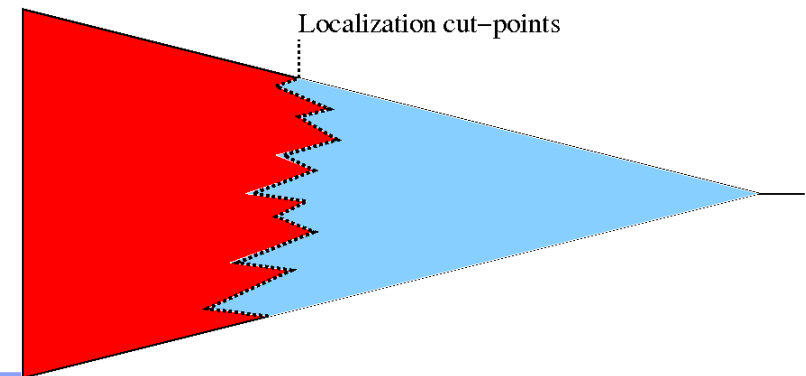
- 2) Construct abstraction to include logic referenced in BMC proof

- 3) Perform verification on the abstraction

- 4) Proof obtained on abstract model? **Pass**

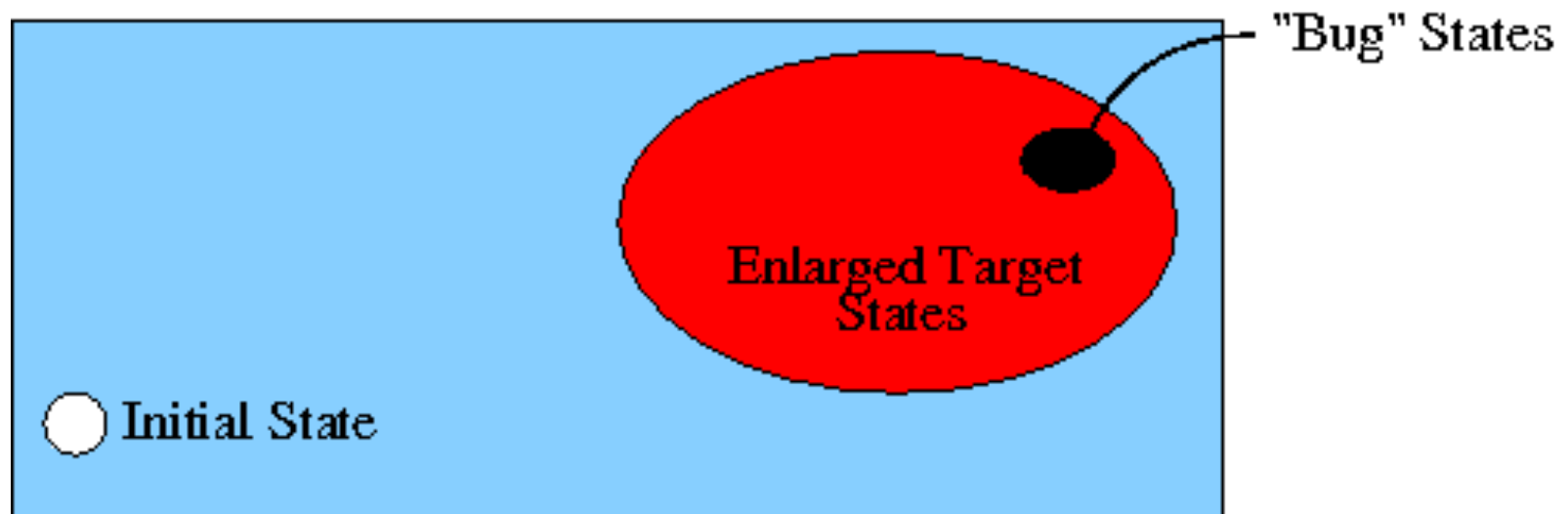
- 5) Counterexample found? Check if valid w.r.t. original netlist

- Yes? **Fail**
- No? goto 1 with deeper bound



Example Transformation 7: Target Enlargement

- Replace property p by set of states which assert p in k transitions
 - Based upon *preimage* computation; resource boundable
- Makes falsification easier
 - Eliminates probability bottlenecks, shallower fails
- May also reduce netlist size; enhance inductivity



Outline

- High-End Hardware Verification Challenges
 - Moore's Law: *What's Next?*
- What is Transformation-Based Verification (TBV)?
- Example Transformations
- Benefits of TBV

Transformation-Based Verification Generality

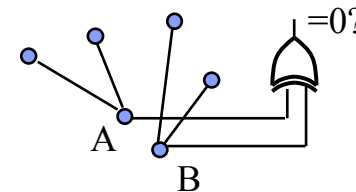
- Allows arbitrary sequencing of engines
 - Localization may be followed by retiming, rewriting, redundancy removal, reparameterization – *then further localization!*
 - Many of these are **synergistic**
 - Localization injects cutpoints deeply in the design; enhances reparameterization and retiming
 - Transforms qualitatively alter the localized design, enabling improved reductions through nested localization

- Some transforms have no counterpart to original netlist
 - Retiming a localized netlist yields reductions unachievable in original netlist
 - *Speculative reduction* yields benefits unachievable in original netlist

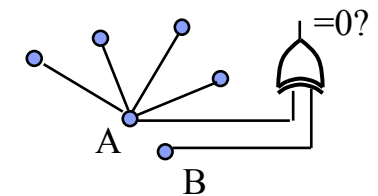
Transformation-Based Verification Generality

■ *Speculative reduction* for sequential redundancy removal

- 1) Guess sets of *redundancy candidates*
- 2) Create a netlist to validate that redundancy
 - Assume all redundancies are correct: *speculative reduction*
 - Add miters (XORs) to *validate* assumptions
- 3) Attempt to prove miters as constant 0
- 4) If successful, exit with identified redundancies
- 5) Else, refine classes; goto step 2



Miter without spec reduction

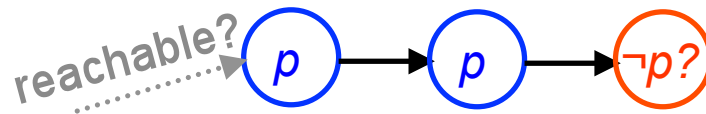


Miter with spec reduction

- Speculative reductions enable greater transform power
- Yields the benefits reduction without *first* needing to prove the suspected equivalence: they become *easier* to prove!

Effectiveness of TBV: Enhanced Proofs

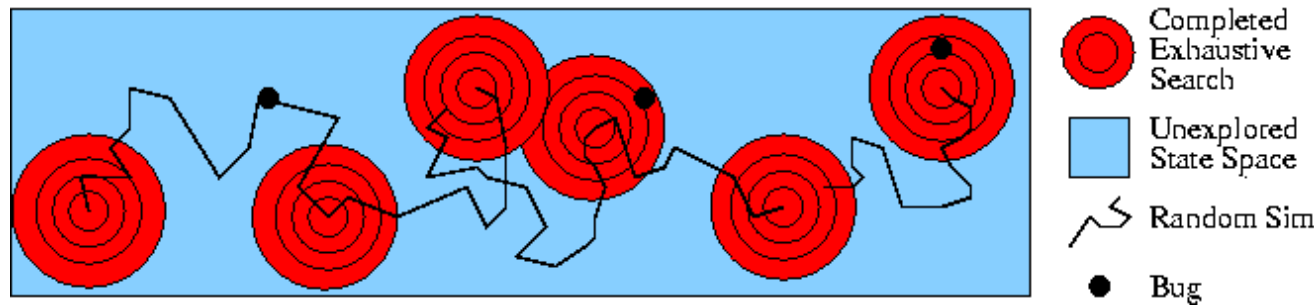
- Reduction in state variables greatly enhances reachability analysis
- “Tightened” state encoding through redundancy removal, retiming enhances inductivity
 - Inductive proof analyzes 2^n states, minus some that lead to fails
 - Transformations themselves prune some **unreachable states**



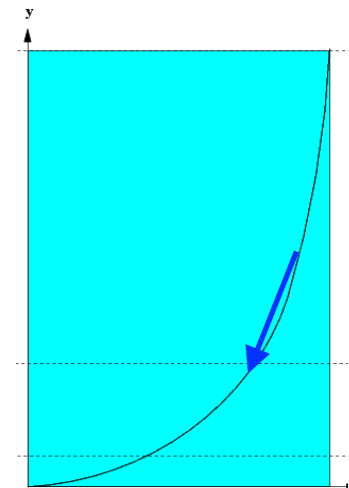
- Enhanced inductivity enhances invariant generation + transforms
- Reduction alone may solve problems
 - After all, an unreachable property is merely a redundant gate
 - Though often critical to balance reduction vs proof / falsification resources

Effectiveness of TBV: Enhanced Falsification

- Simulation + SAT (bounded model checking) are core bug-hunting techniques

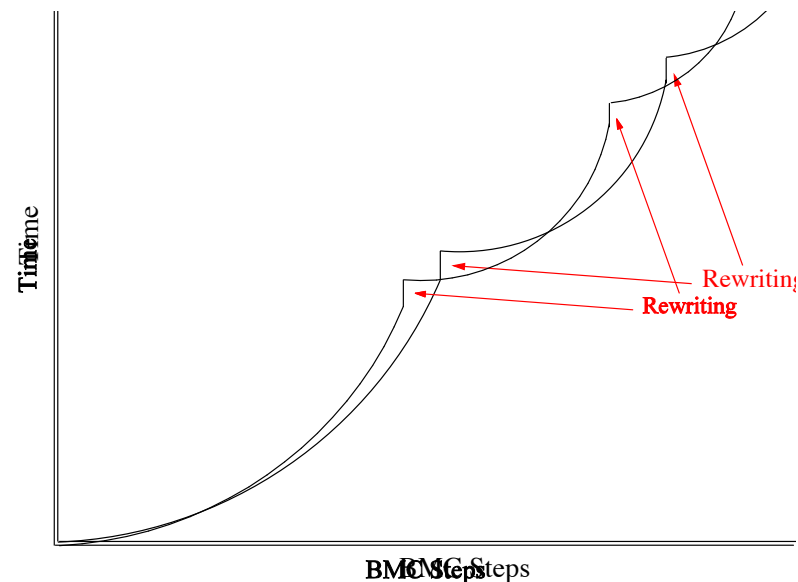


- Smaller netlist → ~linearly faster simulation
- Smaller netlist → ~exponentially faster SAT
- Reducing *sequential netlist* yields *amplified* improvement to SAT
 - Simplify once, unfold many times
 - **Transforms enable deeper exhaustive search**



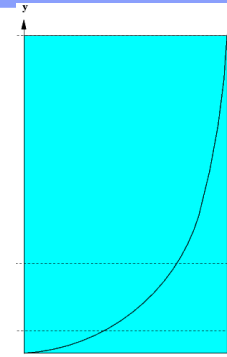
Effectiveness of TBV: Enhanced Falsification

- Reduction of sequential netlist, prior to unfolding, is very useful
- Further reducing of the unfolded netlist is also beneficial
 - Unfolding opens up additional reduction potential
 - We use a hybrid SAT solution; integrates rewriting and redundancy removal
 - All reasoning is time-sliced for global optimality



Effectiveness of TBV

- Property checking is PSPACE-complete
- TBV effectively casts *property checking as redundancy checking*
 - Though clearly this does not change its complexity
 - Certain transforms are thus also PSPACE-complete
- Though: some transforms are polynomial-time
 - Retiming, phase abstraction, ...
- Many may be applied in a resource-bounded manner
 - Redundancy removal may be time-bounded, limited to using induction (NP)
 - Trades reduction optimality for efficient run-time



Effectiveness of TBV

- Different algorithms are better-suited for different problems
 - Feed-forward pipeline can be rendered combinational by retiming
 - A *NP* problem hiding in a *PSPACE* “wrapper”

- More generally: transforms may eliminate facets of design which constitute bottlenecks to formal algorithms
 - Often a *variety* of logic within one industrial testbench
 - Arithmetic for address generation
 - Queues for data-routing
 - Arbitration logic to select among requests

 - Intuitively, optimal solution may rely upon multiple algos

Effectiveness of TBV

- Optimal solution often requires a time-balance between algorithms
 - Algorithmic *synergy* is key to difficult proofs
 - Like time-sliced integration of simplification and SAT
- Given complexity of property checking, the proper set of algos often makes the difference between *solvable* and *intractable*
- *Transforms have substantially simplified almost every verification problem we have encountered*
 - Though clearly a limit to reduction capability of a given set of transforms
 - Then rely upon a strong proof + falsification engine set

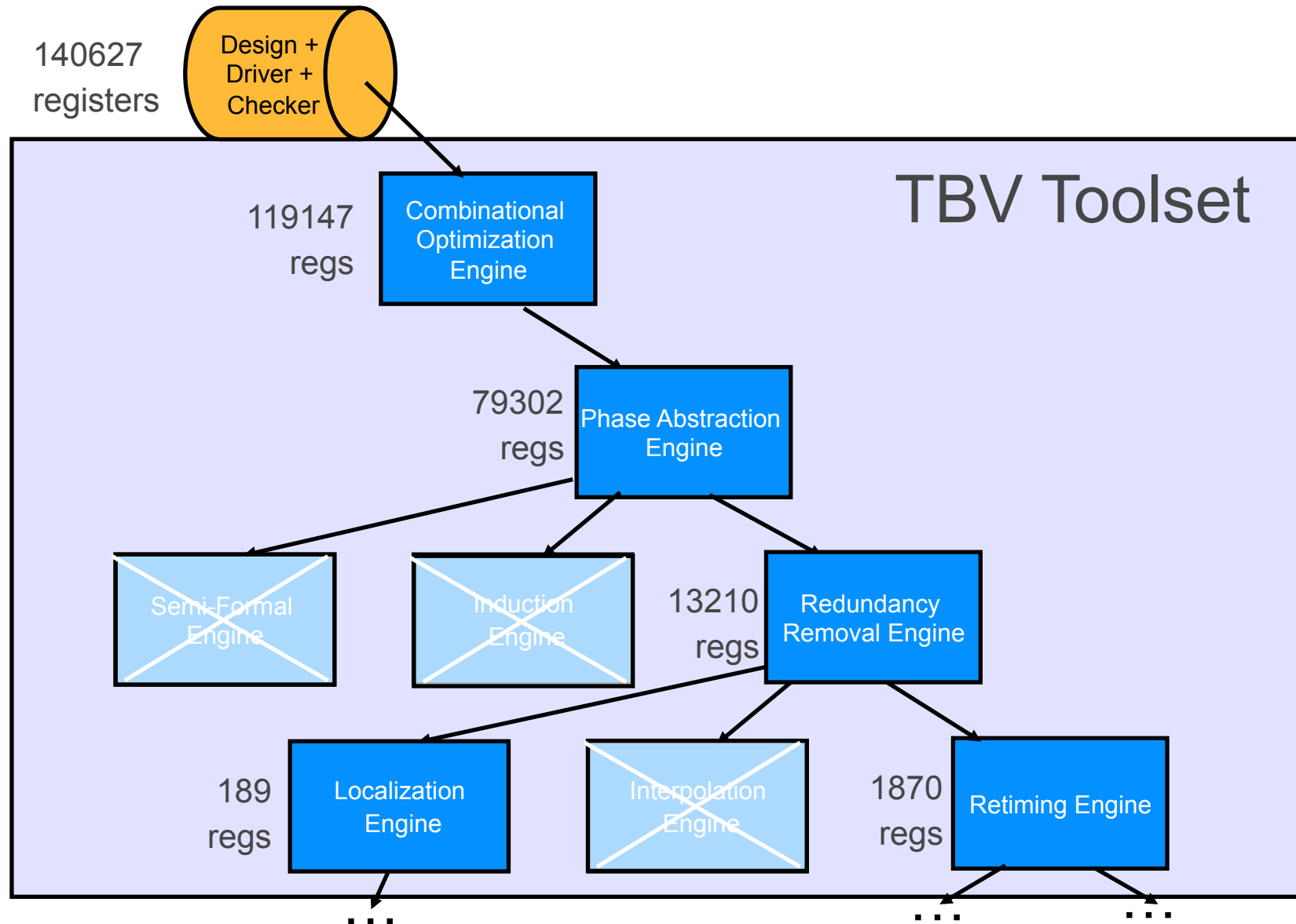
Parallel Processing

- *TBV can dramatically benefit from parallel processing*

- User specifies #machines or #processes to be used for:
 - Finding best-tuned engine flow: can yield *super-linear* speedups
 - Parallel *Expert System*
 - Partitioning many properties across multiple machines
 - Automatic case-splitting of complex properties

• IB
be
to
an

Parallel Transformation-Based Verification (TBV) Flow



• IB
be
to
an

Example Experiments

MMU	Initial*	BRN	AXE	CUT	AXE	CUT	RET	BRN	CUT	ESE
Registers	124297	67117	698	661	499	499	133	131	125	PASS
ANDs	763475	397461	9901	8916	5601	6605	16831	4645	1300	1038 sec
Inputs	1377	162	1883	809	472	337	1004	287	54	386 MB

ERAT	Initial*	BRN	EQV	RET	BRN	SEQ	AXE:50
Registers	45637	19921	419	337	273	257	FAIL
ANDs	316432	167619	3440	2679	1851	1739	2831 sec
Inputs	6874	68	63	183	126	126	884 MB

BRN: Combinational Rewriting

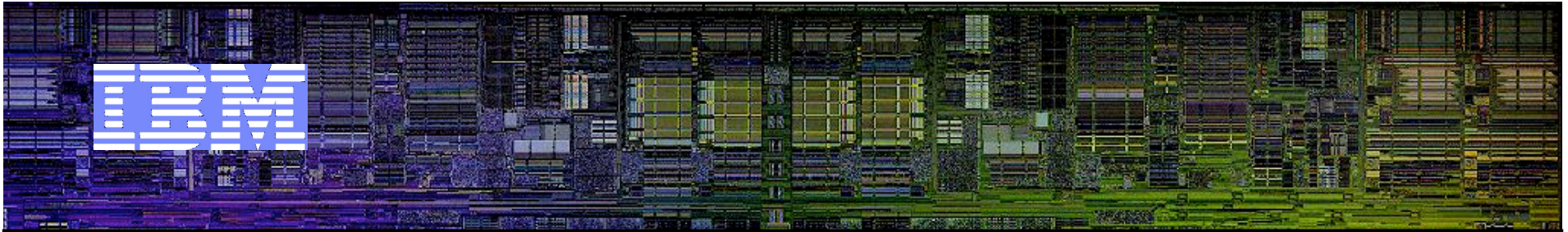
RET: Min-area retiming

AXE: Localization

CUT: Reparameterization

ESE: Reachability

EQV: Sequential Redundancy Removal



Summer Formal 2011

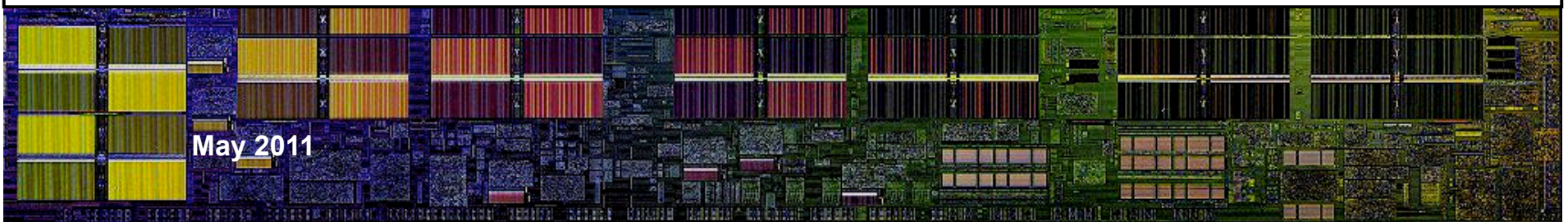
Hardware Verification 301

Industrial Hardware Verification In Practice

Jason Baumgartner

www.research.ibm.com/sixthsense

IBM Corporation



Outline

■ Class 1: Hardware Verification Foundations

- Hardware and Hardware Modeling
- Hardware Verification and Specification Methods
- Algorithms for Reasoning about Hardware

■ Class 2: Hardware Verification Challenges and Solutions

- Moore's Law v. Verification Complexity
- Coping with Verification Complexity via Transformations

■ Class 3: Industrial Hardware Verification In Practice

- Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies

Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies

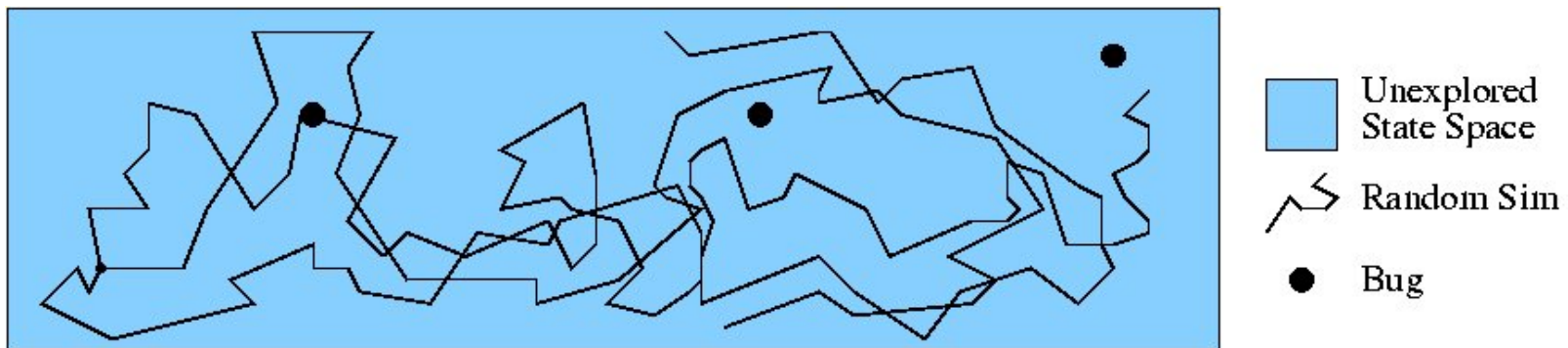
Industrial Verification

- Simulation constitutes the primary work-horse of industrial verification
- *Acceleration* is gaining prevalence: 1000s× speedup
 - Simulation model loaded into programmable hardware
- Model checking is gaining in prevalence
- Combinational equivalence checking is used everywhere

- ~\$530M annual revenues for simulation
- ~\$41M annual revenues for model checking
 - An additional ~\$12M in miscellaneous static analysis-based solutions
- ~\$85M annual revenues for equivalence checking

Simulation

- Scalable yet incomplete: only a tiny fraction of state space explored
- Considerable manual effort to squeeze the most benefit from this fraction



- 1) Build laborious model-based test generators on occasion
- 2) Intricate biasing of random stimulus to reach improbable behaviors
- 3) Manual construction of coverage models to measure progress
- 4) If insufficient, tweak 1-2 and repeat

Simulation

- Simulation requires more effort than FV, modulo “coping with capacity”
 - Due to need for testcase biasing, coverage analysis
- Nonetheless remains prominent vs FV due to
 - Scalability: even with semi-formal extensions, FV does not scale to large chips + units
 - Reusable specifications across hierarchy
 - Simulation environments often performed hierarchically
Unit-level tests, then core, then chip, then system
 - Lower levels may be attempted formally, though higher levels cannot be
 - Legacy tools + skills + verification IP still favor simulation
 - Scalable model checkers are more recent

Simulation

- A variety of specification languages can be used for simulation
- SVA, PSL are fairly common
- HDL-based testbenches may also be used
- C/C++ environments are also common
 - Unfortunately, these cannot be readily reused in model checking
 - And often considerably slow accelerators
- ***Reusable verification IP across platforms is an important goal***

Sequential Equivalence Checking (SEC)

- SEC is becoming a huge “verification win”
 - Technology remaps, “IP import” projects may forgo functional verification due to SEC
 - *Timing bringdown* design phase greatly simplified by SEC capability
 - Enables late / aggressive changes that otherwise would not be tolerated
- Equivalence checking requires *much* less effort than functional verification
 - No need to write properties; they are automatically inferred
 - Driver constraints are often trivial
 - Drive a clock, disable “scan mode”
 - Proofs are often highly-scalable; obviate manual effort for coverage analysis
- Plus, no risk of bug escapes
- Though generally does not replace functional verification

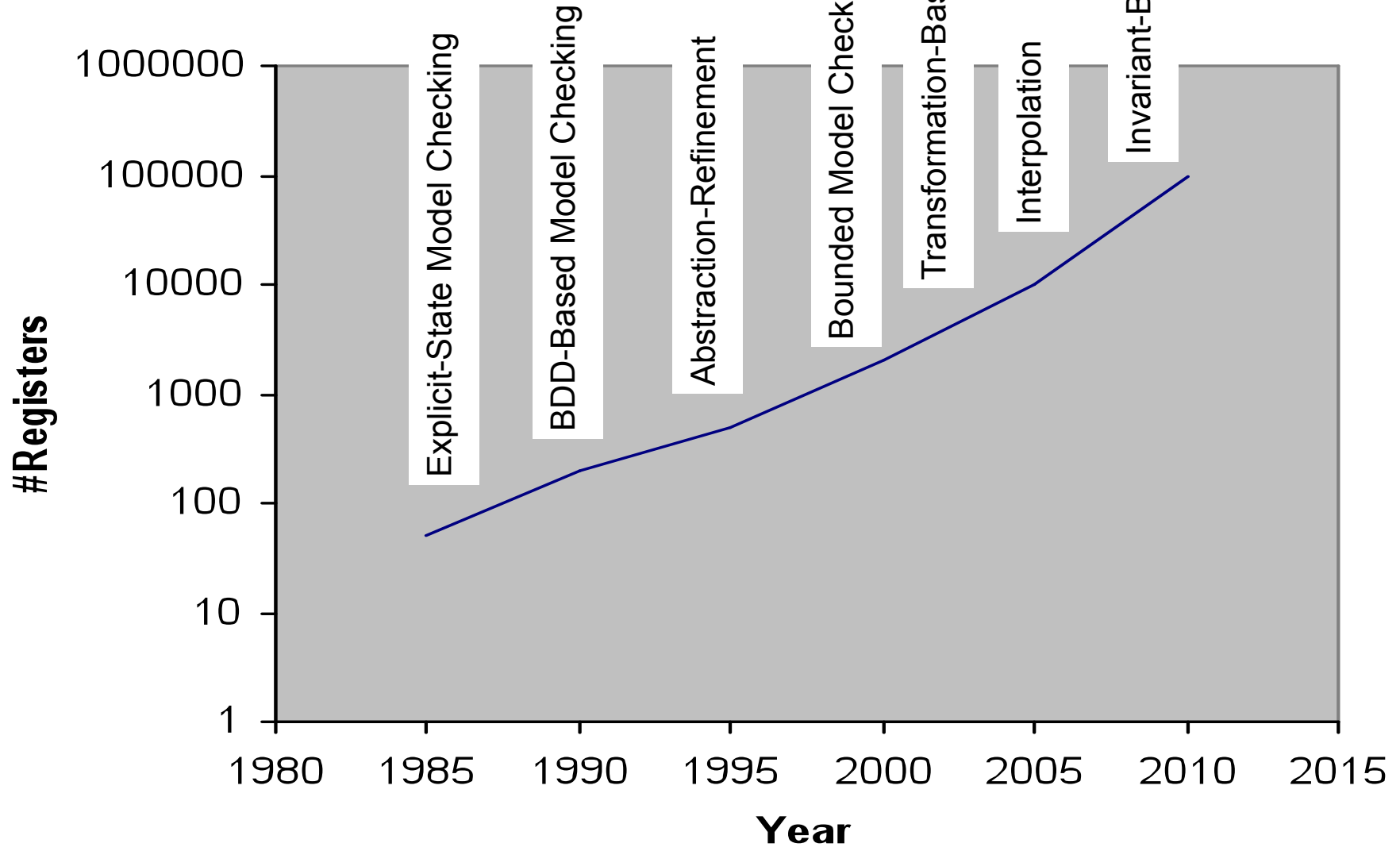
Model Checking: The Early Days

- Symbolic model checking rapidly demonstrated its value in finding corner-case bugs
- Though capacity limitations rendered it usable primarily by FV experts
- Expensive to deploy: required considerable manual abstraction
- Manual abstractions and remodeling often jeopardized its completeness
- Often degraded to verifying *blocks* vs verifying *functionality*
 - Difficult to cover (micro-)architectural properties
 - A different type of coverage problem

Model Checking: The Early Days

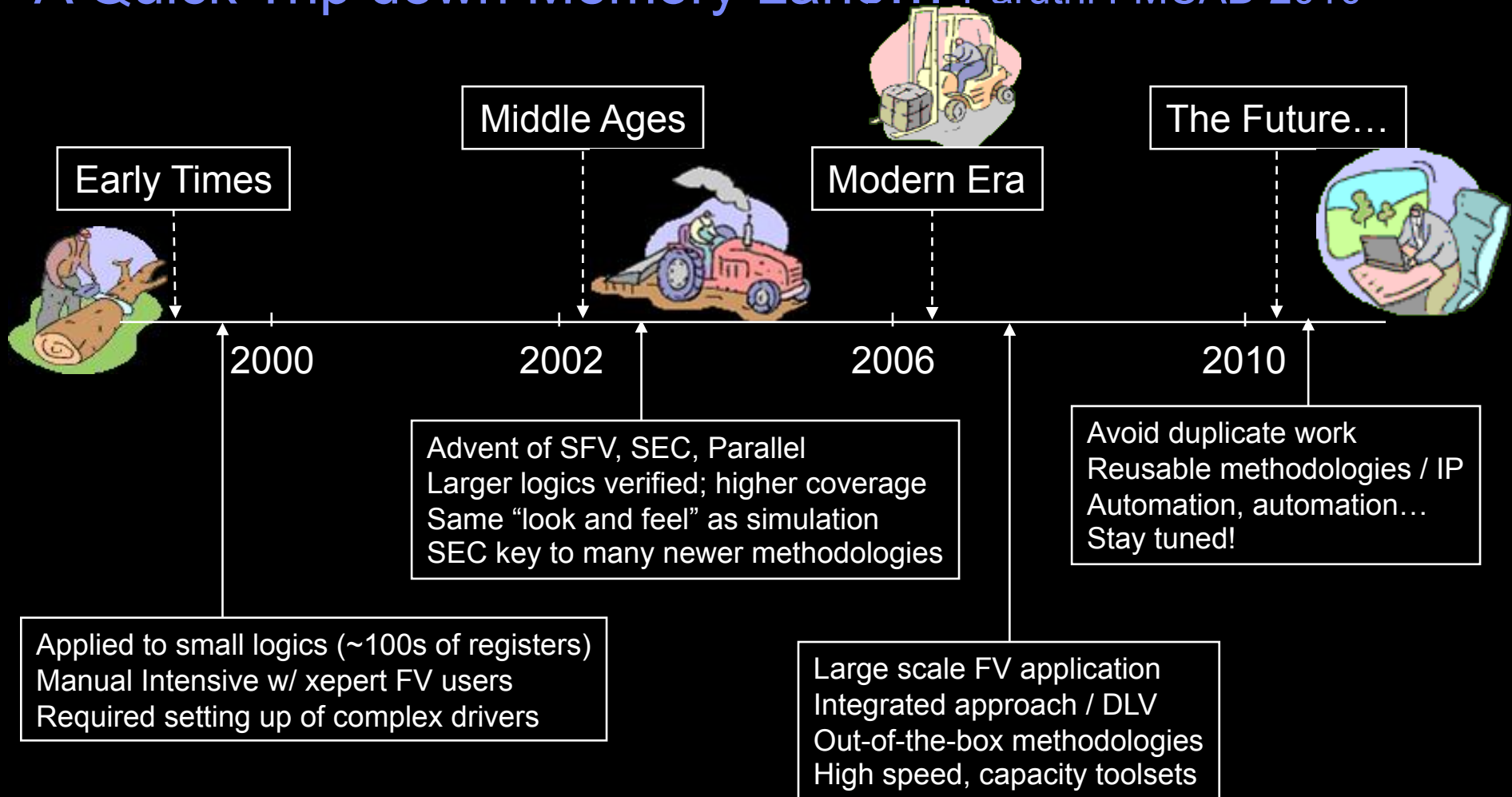
- Early IBM chips could claim ~1 saved chip fabrication due to FV
- Though given strength + challenges of model checking, initially deployed on a limited scale on most critical logic
 - FV teams often 10× smaller than sim teams
 - *Fire fighters*, rotating among most critical logic and reacting to late bugs (eg post-Si)
- Did not even *touch* each design unit with FV
 - Many could not be meaningfully abstracted anyway
 - Considerable *risk*: weeks spent developing testbench only to choke FV tool
 - **Probably unwise to significantly increase FV deployment given technology limitations**
- **Capacity gains critical to today's wider-scale deployment of FV**

Model Checking Capacity



Caveat: not *guaranteed* capacity; some problems with 0 registers are unsolvable!
Very incomplete highlight list; capacity cumulatively leverages earlier innovations + SW engineering

A Quick Trip down Memory Lane... Paruthi FMCAD 2010



SFV: Semi-formal verification
 SEC: Sequential Equivalence Checking
 DLV: Designer-level Verification

Proliferation of Model Checking

- Capacity challenges traditionally limited MC deployment
 - Risk that days / weeks spent developing a formal spec, only to choke the tool
- Several key facets have gone into more recent proliferation
 - 1) Boosts in core proof technologies: transformations, interpolation, ...
 - Though not necessarily *reliable* boosts
 - 3) Semi-formal extensions
 - At least offer high bug-hunting power; broader *non-FV expert* audience
 - 5) Reusable specifications
 - Cross-leverage designer specs, sim vs FV teams
 - 7) Improved methodologies

Proliferation of Model Checking

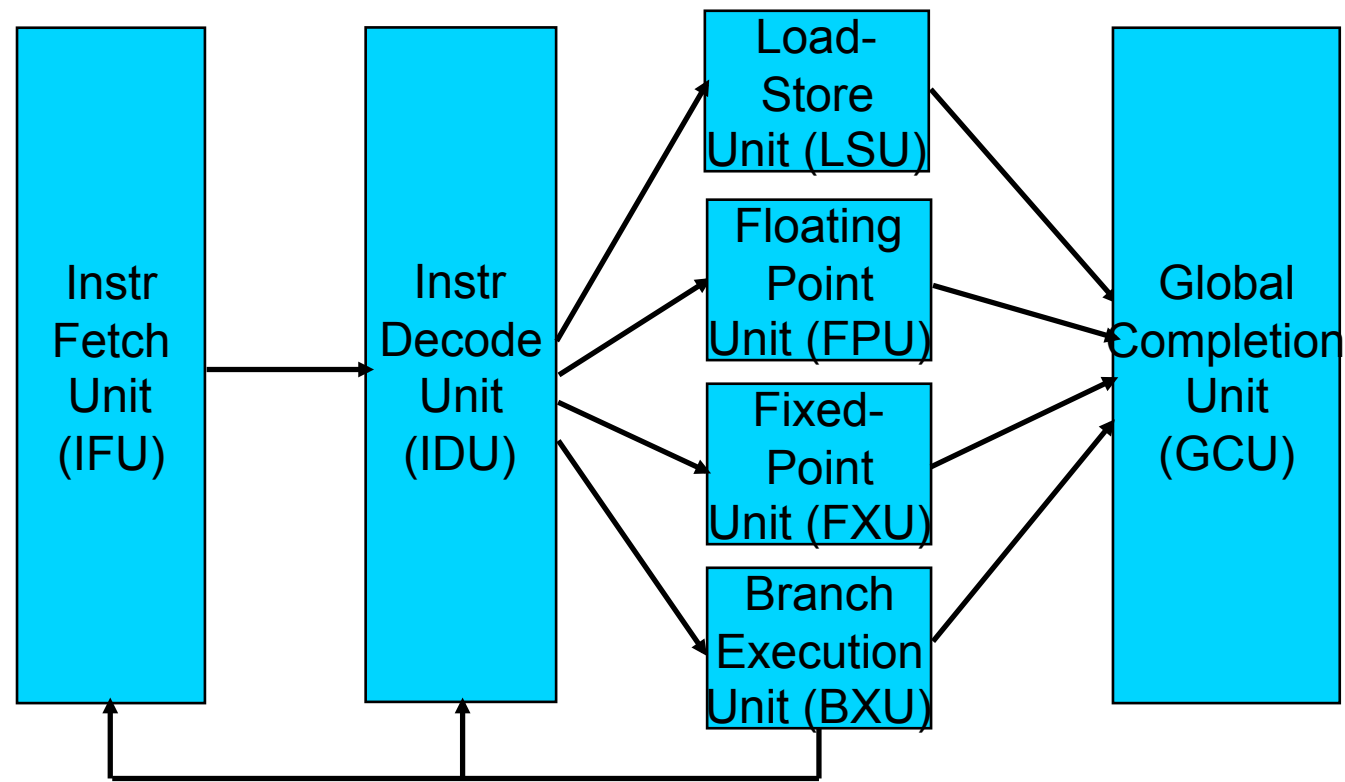
- Automated techniques are continually increasing in capacity
- However, for complex proofs, manual techniques are critical to push the capacity barrier
 - Choice of testbench boundaries
 - Manual abstractions to reduce design complexity
 - Underapproximations and overapproximations
 - Strategic development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

Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
 - Choosing Logic Boundaries
 - Overriding Internals
 - Overconstraining
 - Shortcuts
- Case Studies
- Concluding Remarks

Testbench Authoring: Logic Boundary Options

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



Testbench Authoring: Logic Boundaries

1. Develop unit-level Testbench without worrying about proof feasibility
 - Unit-level testbenches often built for sim regardless
 - Synthesizable language → reusable for FV, acceleration, ...
 - Leverage semi-formal verification for *bug-hunting*
 - Find intricate bugs quickly, not gated by time to develop proof-oriented testbench
 - With luck, a robust tool may yield proofs regardless
 - But may likely need hand-tweaking of Testbench for proofs
 - Proof effort can be done in parallel to semi-formal bug-hunting

Testbench Authoring: Logic Boundaries

1. Develop unit-level Testbench without worrying about proof feasibility

- **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**

- Interfaces are more stable, simpler and better-documented
 - Less testbench bringup effort
 - Fewer testbench vs real bugs suffered
 - Better chance of reusability of spec across projects
- *Verify functionality vs. verify blocks*
 - Difficult to *cover* architectural properties on small blocks

Testbench Authoring: Logic Boundaries

2. Develop minimal Testbench encompassing only functionality to be verified
 - Higher chance of *proofs, corner-case bugs* on smaller Testbench
 - Data Prefetch is much smaller than entire Load-Store Unit!
 - Block-level Testbench often more difficult to define than unit-level
 - More complex, prone to change, poorly documented input protocol
 - Works well if done by *designer* at *design granularity* level
 - E.g. designer of Data Prefetch building Testbench at that level

Testbench Authoring: Logic Boundaries

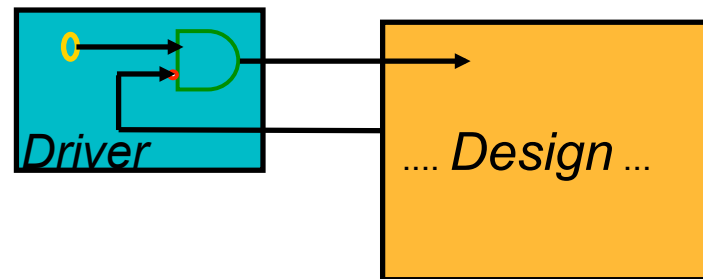
2. Develop minimal Testbench encompassing only functionality to be verified
 - Requires dedicated effort to enable FV
 - Checks and Assumptions *may* be reusable in higher-level sim
 - But often need to develop a higher-level Testbench for sim
 - Requires more Testbenches to cover a given unit
 - Load queue, store queue, prefetch, ... vs “Load-Store Unit”

Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
 - Choosing Logic Boundaries
 - Overriding Internals
 - Overconstraining
 - Shortcuts
- Case Studies
- Concluding Remarks

Testbench Authoring: Overriding

- Overriding internal signals is another method to select logic under test
 - Recall: ability to override inputs + internals using a *driver*



- *Black box* an unnecessary component, leaving its behavior nondeterministic
 - Or selectively override individual signals as cutpoints
- Occasionally a more precise behavior is necessary
 - E.g., cross-dependence among a set of cutpoints
 - E.g., an instruction tagged as a *floating-point operation* must be *valid*

Testbench Authoring: Overriding and Underconstraining

- Sometimes overriding merely serves to reduce testbench size
 - Override parity logic which is not being tested anyway
 - *Circuit optimality* does not necessarily imply *smallest netlist*
 - E.g., rewrite a one-hot state machine to use fewer state bits
 - Contemporary algos are less sensitive to this; somewhat automated
- Care must be taken not to overconstrain, or incorrectly constrain
 - Else bugs may slip through the model checking process!
- Underconstraining is desirable when possible
 - Less effort, less complex testbench
 - No missed bugs!

Testbench Authoring: Overconstraining

- Overconstraining may entail missed bugs
 - Though overconstraining is practically necessary in many cases
- 1) Incremental testbench authoring: first model one type of instruction, then another, then another, ...
 - Accelerates time to first bug; testbench authoring may be laborious!
 - Process of gradual elimination of constraints
 - 2) “Quick regression” flavor of a testbench
 - Model checking runs may be lengthy
 - Often good to have a set of overconstrained “faster regression” runs
 - More interactive way to identify bugs after a design or testbench change

Testbench Authoring: Overconstraining

- Though overconstraining is practically necessary in many cases
- 3) Model checking is PSPACE complete; a proof may be infeasible!
 - Better to get partial coverage via overconstraining than MEMOUT
 - As with non-exhaustive simulation, clever use of constraints may enable identifying all the bugs anyway
 - 5) Time may not permit a completely accurate testbench
 - Testbench authoring is laborious
 - A subset of functionality may be of greatest interest; constrain accordingly
 - 6) Constraints may be used in complete *case-splitting* strategies
 - Use a set of overconstrained testbenches, which collectively cover all behaviors

Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
 - Choosing Logic Boundaries
 - Overriding Internals
 - Overconstraining
 - Shortcuts
- Case Studies
- Concluding Remarks

Testbench Authoring: Shortcuts

- Writing checkers and drivers is laborious
 - Occasionally one may easily *approximate* a desired check

- Validate that the design properly associates <tag, data>?
 - *Drive* data as a function of tag, *check* that function later

- Need to check that data transfers occur in FIFO order?
 - Encode a counter into driven data; check for monotonicity at output

- Need to track progress of a set of tags (eg cache lines)?
 - Nondeterministically choose one to track; ignore the others

Testbench Authoring: Shortcuts

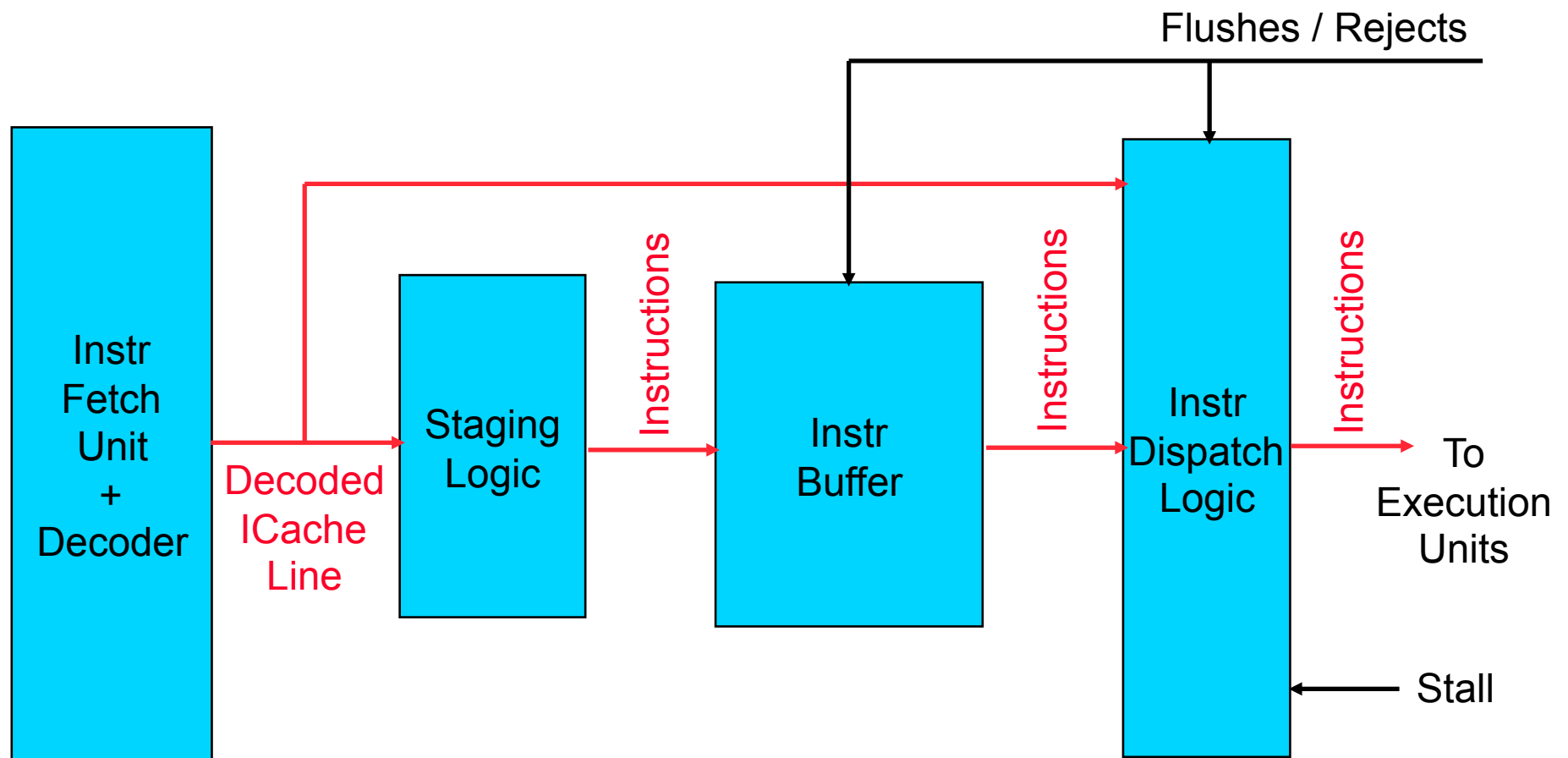
- In cases, such shortcuts are *lossless*
- If used carefully, chance of missed bugs is negligible
 - Constrain <data> as a function of <tag>? Also try a secondary function
- Though may hurt portability to simulation, acceleration
- Overall, consider your verification purpose
- Don't do more work than necessary
- Don't delay time-to-first-bug
- Though for *critical* or *reusable* tasks, may need a more structured approach

Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies
 - Instruction Dispatch Case Study
 - Instruction Fetch-Hang Case Study
 - Floating-Point Unit Verification
 - Load-Store Verification
- Concluding Remarks

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

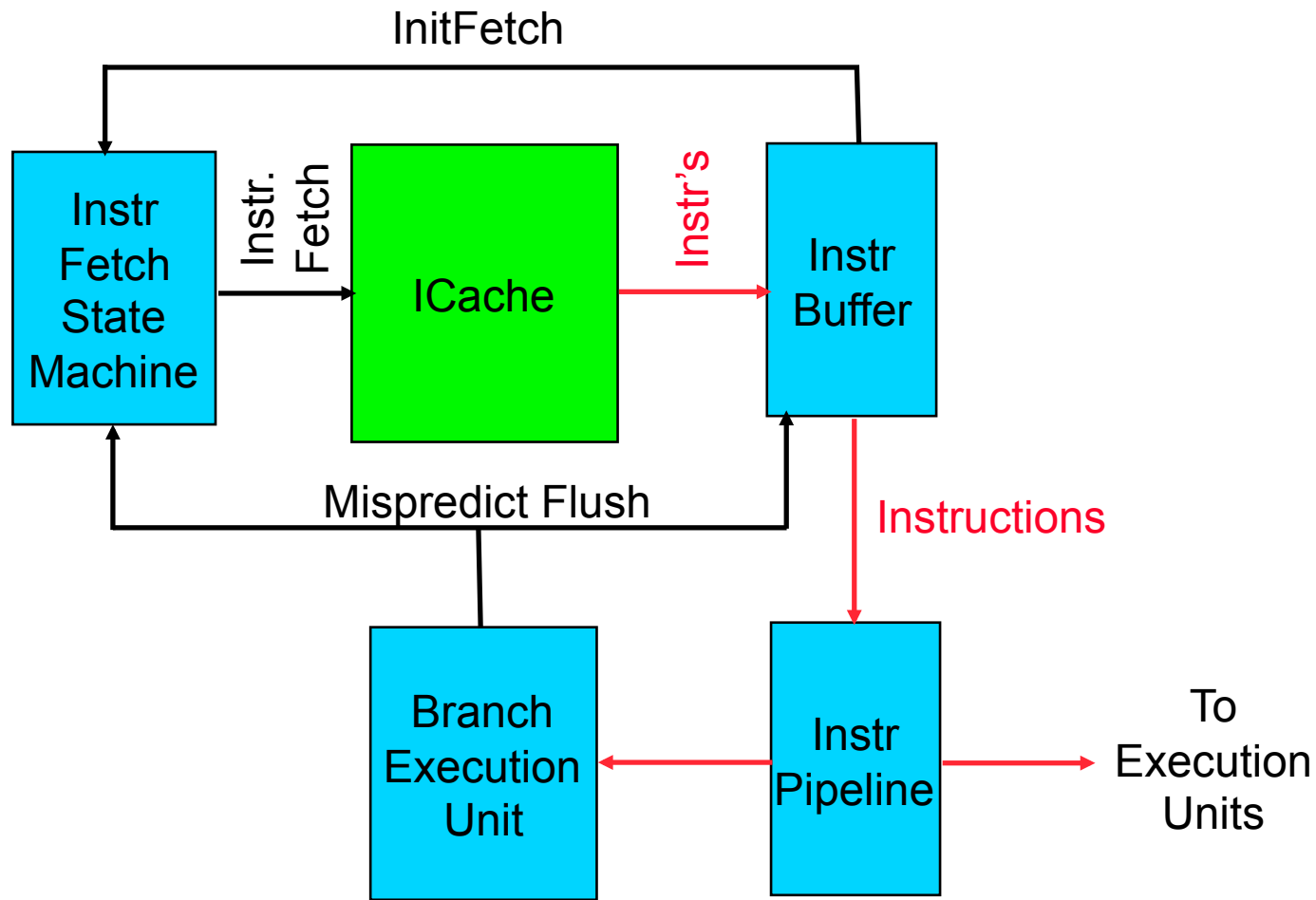
Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies
 - Instruction Dispatch Case Study
 - Instruction Fetch-Hang Case Study
 - Floating-Point Unit Verification
 - Load-Store Verification
- Concluding Remarks

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

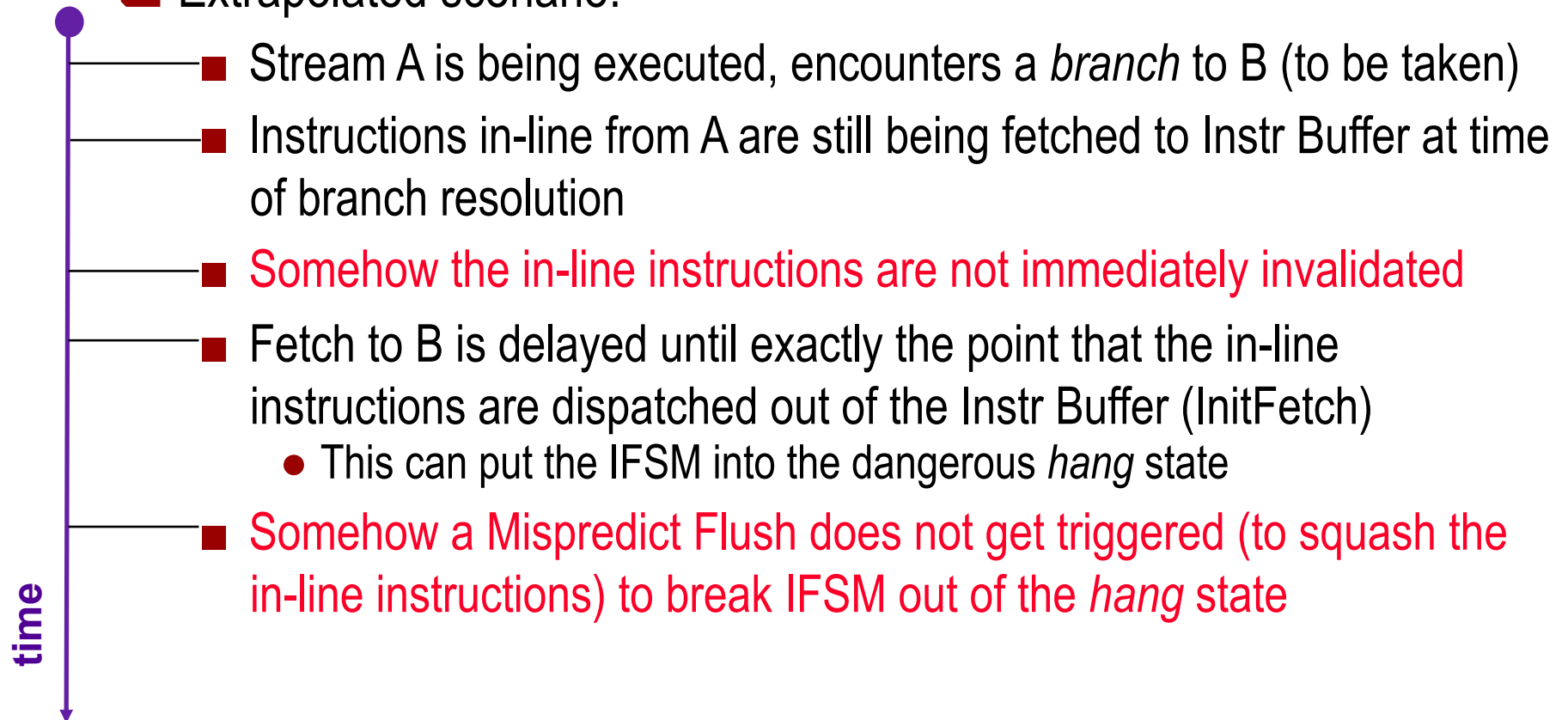
Fetch-Hang Case Study

- ❑ 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

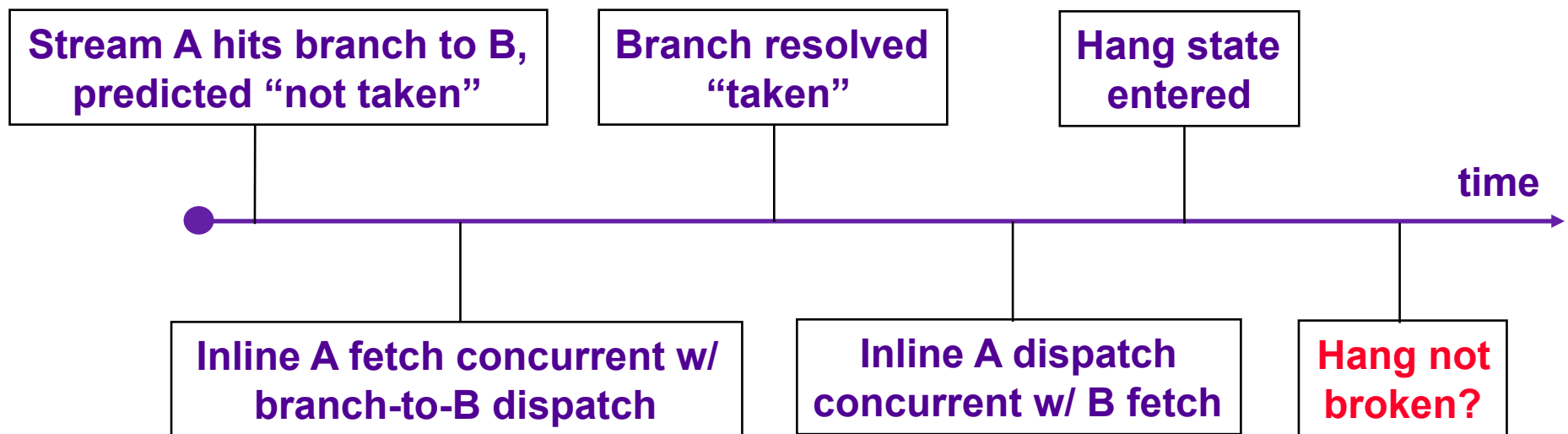
❑ Extrapolated scenario:



❑ 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

Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies
 - Instruction Dispatch Case Study
 - Instruction Fetch-Hang Case Study
 - Floating-Point Unit Verification
 - Load-Store Verification
- Concluding Remarks

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

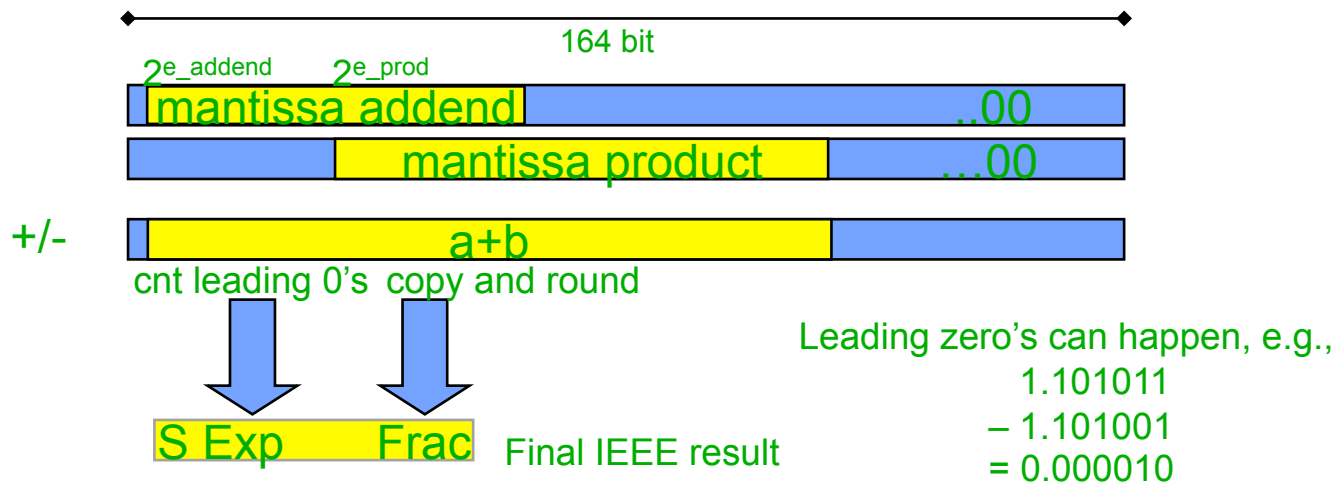
- ❑ Checks numerical correctness of FPU datapath
 - Example bugs:
 - If two *nearly equal* numbers subtracted (causing cancellation), the wrong exponent is returned
 - If result is near underflow, the wrong guard-bit is chosen

- ❑ 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

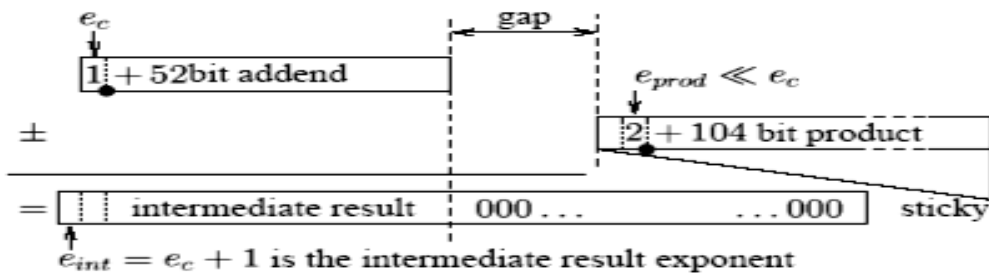
- Four distinct cases distinguished in Ref-FPU
 - Based on difference between product, addend exponent

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

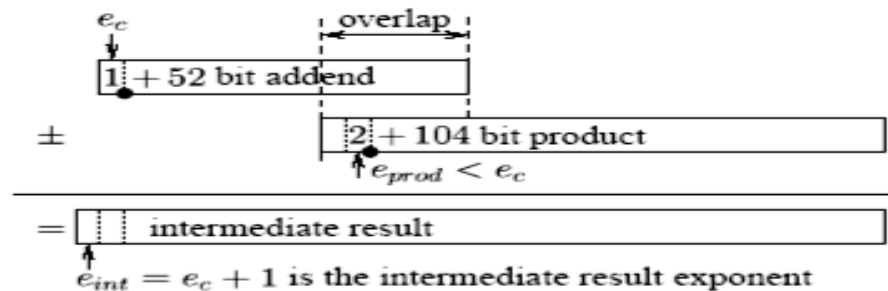
- Case splitting strategy via constraining internal Ref-FPU signals
 - Verification algos implicitly propagate these constraints to real FPU
 - Allows each case to cover large, difficult-to-enumerate set of operands

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

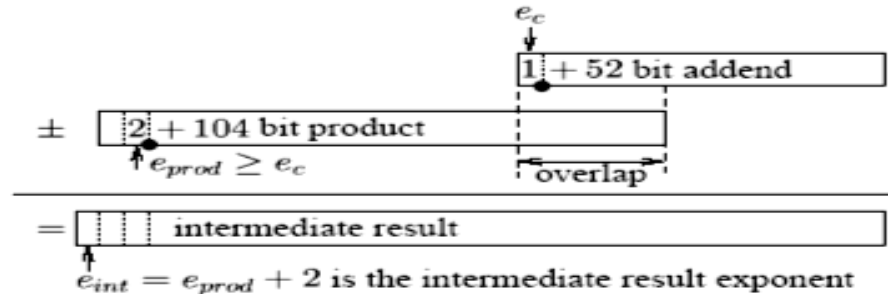
- Disjunction of cases easily provable as a tautology, ensuring completeness



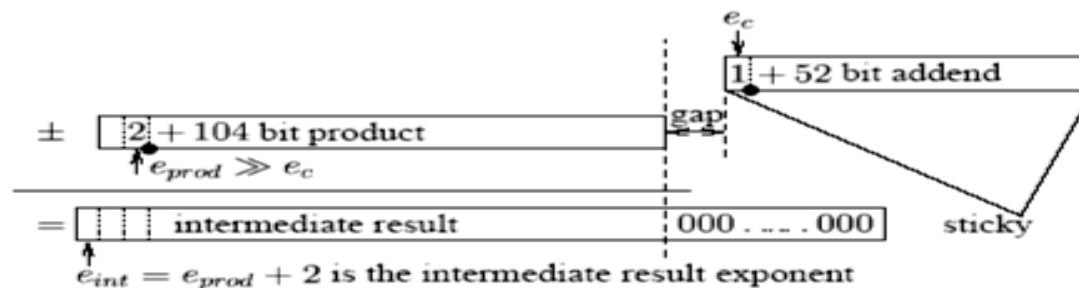
(a) Farout left: the addend is much larger than the product; there is no overlap, the product becomes a sticky bit.



(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
 - *S ha* 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

Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies
 - Instruction Dispatch Case Study
 - Instruction Fetch-Hang Case Study
 - Floating-Point Unit Verification
 - Load-Store Verification
- Concluding Remarks

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 $j \neq i$ 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 $j \neq i$
 - 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

Outline

- Industrial Verification and the Evolution of Model Checking
- Testbench Authoring Concepts
- Case Studies
- Concluding Remarks

Hardware Verification in a Nutshell

- Hardware verification is trickier than it should be!
 - ***Hardware verification is not a solved problem***
 - Many unsolvable problems; **manually-intensive** to cope with these
 - Capacity enhancements have gone a long way to automating such manual testbench tricks
- Verification *must* address all HW implementation ugliness
- Either in functional verification, or equivalence checking
 - Reference / architectural models are easier to verify
 - Though laborious to develop these, and they ultimately miss bugs

Hardware Verification Progress

- **Hardware verification is not a solved problem**
- Room for many improvements to core verification algos
 - Improvements to bit-level proof, falsification, transformation algorithms
 - Improvements to higher-level techniques (SMT, word-level, ...)
 - Improvements to theory / solver combinations to handle heterogenous designs
- Continue to see powerful improvements to core bit-level solvers
- Hardware Model Checking Competition helps to drive this research
 - <http://fmv.jku.at/hwmcc10/>
- Semiconductor Research Corporation is fostering the HWMCC winners
 - <http://www.src.org>
- **Though research in bit-level HW verification is waning**
- Encourage your group to submit to the next SRC project solicitation!

Industrial Hardware Verification Requirements

- Robust set of proof + *falsification* algos; automation + scalability
 - Must scale to simulation-sized testbenches, at least for falsification
 - Though increasingly able to complete proofs at surprisingly large scales
 - Need to **bring power of FV to designers** vs. expert FV engineers alone

- Transformation-based paradigm to cope with implementation bottlenecks
 - Critical for scalability of both proofs + falsification

- Embrace reusable specs between designer, simulation + formal teams
 - **No new languages** or interfaces; *make FV look like simulation*
 - Lesser education + “verification investment”; higher “return” + reusability

- **Trust in FV to address your HW verification needs**

- **And don't do more work than necessary to write your testbench!**



Grand Challenge

- “Someday Moore’s Law will work *for*, not *against*, the verification community” Allen Emerson
 - Requires substantial innovation! *Help us achieve this goal !!!!*

- Perhaps, if we can enable higher-level design without manually-derived reference model
 - Synthesis *must be* sequential; RTL design is too low-level
 - Though manually-designed pervasive logic interacts with state elements
 - Scan chains, debug buses, ... also ECO challenges, ... A tough problem!

- Grand challenge: application of higher-level algos to bit-level implementation-detailed designs

■ References

Hardware Verification References

- AIG tools; tools for converting to / from AIG:
 - AIGER <http://fmv.jku.at/aiger/>
 - SMV (*crude subset!*), BLIF

- Converting HDLs into benchmark formats:
 - vl2mv: somewhat limited Verilog to BLIF
 - Part of the VIS toolkit: <http://vlsi.colorado.edu/~vis>

- State-of-the-art model checker
 - ABC <http://www.eecs.berkeley.edu/~alanmi/abc/>
 - Numerous transformations, formal / semiformal engines, synthesis routines
 - Overall 1st place winner of all Hardware Model Checking Competitions

Hardware Verification References

■ AIGs and Transformation-Based Verification

- “ABC: An Academic Industrial-Strength Verification Tool” CAV 2010
- “Scalable Automated Verification via Expert-System Guided Transformations” FMCAD 2004
- IBM SixthSense Homepage <http://www.research.ibm.com/sixthsense>

■ Symbolic Simulation / Bounded Model Checking / BDDs

- “Formal Hardware Verification with BDDs: An Introduction”, PRCCC 1997
- “Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs” CAV 1999

■ BDD-Based Reachability Analysis

- “Border-Block Triangular Form and Conjunction Schedule in Image Computation” FMCAD 2000

Hardware Verification References

■ Semi-formal Verification

- “Smart Simulation using Collaborative Formal and Simulation Engines” ICCAD 2000
- “Using Counter Example Guided Abstraction Refinement to Find Complex Bugs” DATE 2004

■ Constraints vs. Drivers

- “Speeding up Model Checking by Exploiting Explicit and Hidden Verification Constraints” DATE 2009

■ Liveness vs. Safety

- “Liveness Checking as Safety Checking” ENTCS vol 66

Hardware Verification References

■ Induction, Invariant Generation

- “SAT-Based Verification without State Space Traversal” FMCAD 2000
- “Checking Safety Properties using Induction and a SAT-Solver” FMCAD 2000
- “Exploiting state encoding for invariant generation in induction-based property checking” ASPDAC 2004
- “Cut-Based Inductive Invariant Computation” IWLS 2008
- “Strengthening Model Checking Techniques with Inductive Invariants” TCAD 2009
- “SAT-Based Model Checking without Unrolling” VMCAI 2011

■ Interpolation

- “Interpolation and SAT-Based Model Checking” CAV 2003

Hardware Verification References

■ Redundancy Removal, Equivalence Checking

- “Sequential Equivalence Checking without State Space Traversal” DATE 1999
- “Speculative reduction-based scalable redundancy identification” DATE 2009

■ Abstraction-Refinement

- “Counterexample-Guided Abstraction Refinement” CAV 2000
- “Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines” DAC 2001
- “Automatic Abstraction without Counterexamples” TACAS 2004

■ Phase Abstraction

- “Automatic Generalized Phase Abstraction for Formal Verification” ICCAD 2005

Hardware Verification References

■ Retiming for Verification

- “Transformation-Based Verification Using Generalized Retiming” CAV 2001

■ Target Enlargement

- “Property Checking via Structural Analysis” CAV 2002

■ Input Reparameterization

- “Maximal Input Reduction of Sequential Netlists via Synergistic Reparameterization and Localization Strategies” CHARME 2005

Hardware Verification References

■ Logic Rewriting

- “DAG-aware AIG rewriting: A fresh look at combinational logic synthesis” DAC 2006
- “SAT Sweeping with Local Observability Don’t Cares” DAC 2006

■ SAT Solvers

- “The Quest for Efficient Boolean Satisfiability Solvers” CAV 2002
- “An Extensible SAT-solver” SAT 2003