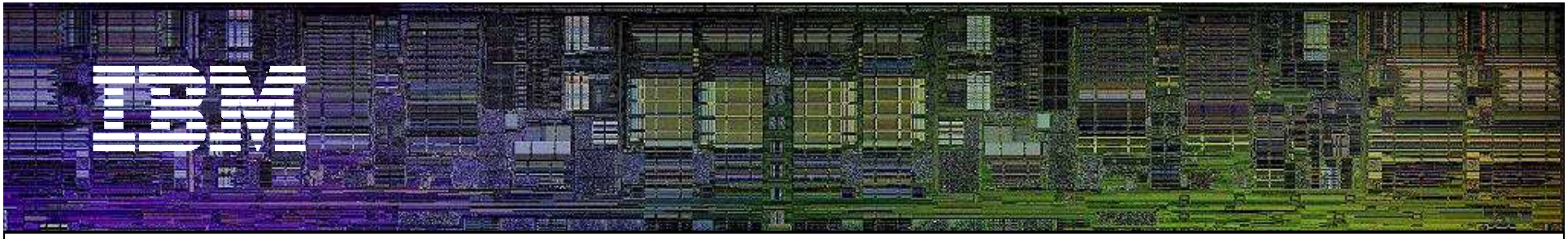


Scalable Conditional Equivalence Checking: An Automated Invariant-Generation Based Approach

Jason Baumgartner, Hari Mony, Michael Case, Jun Sawada and Karen Yorav

IBM Corporation

Anecdote: My Favorite Book Title

- Computer-Aided Reasoning: *An Approach*
  - Matt Kaufmann, Panagiotis Manolios, J Moore
- *Informative* title
- ***Unassuming*** title
  - Don’t even claim that it is a *good* approach
  - Though of course, it is!

Motivation for Our Title

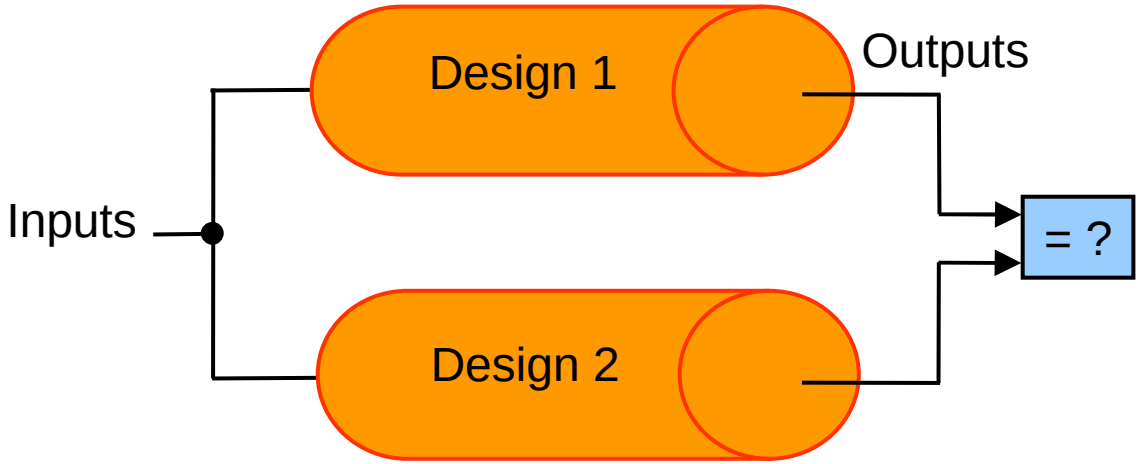
- Scalable Conditional Equivalence Checking: *An* Automated Invariant-Generation Based *Approach*
- ***Even more*** informative
- Comparably unassuming
  - Brute-force, eager technique
  - Relies upon heuristics to avoid exorbitant resources
    - *Is it a good approach??*
  - Nonetheless, *the only method* we have to solve certain problems

Outline

- Equivalence Checking
  - Combinational Equiv Checking (CEC)
  - Sequential Equiv Checking (SEC)
  - Conditional SEC (CSEC)
- Traditional SEC Algos
- CSEC Algos
- Experiments + Conclusion

Equivalence Checking

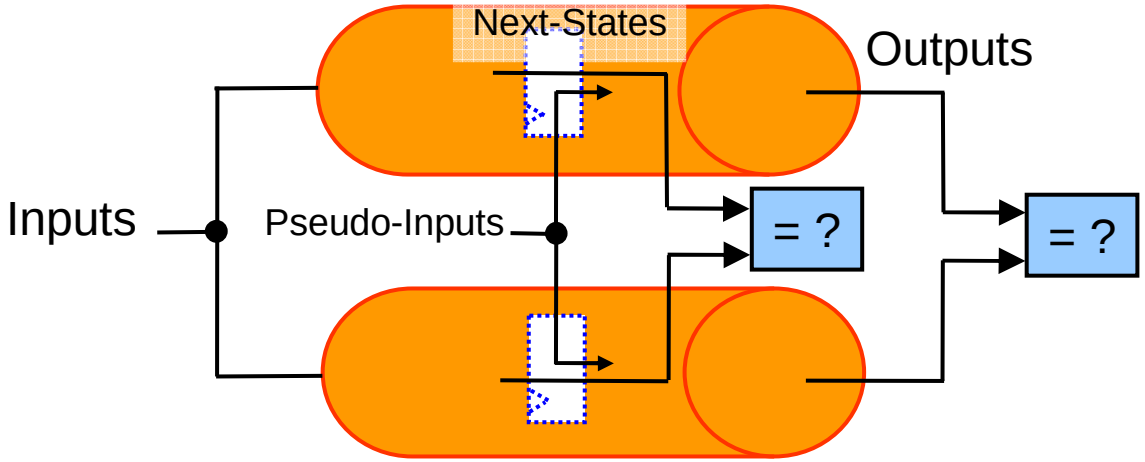
- A method to assess behavioral equivalence of two designs



- 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)

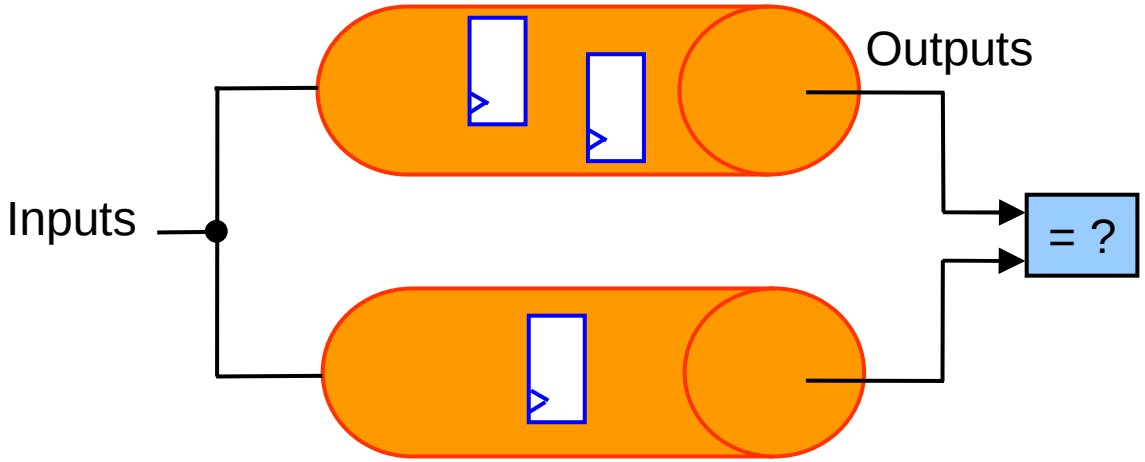
- No sequential analysis: state elements become *cutpoints*



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

Sequential Equivalence Checking (SEC)

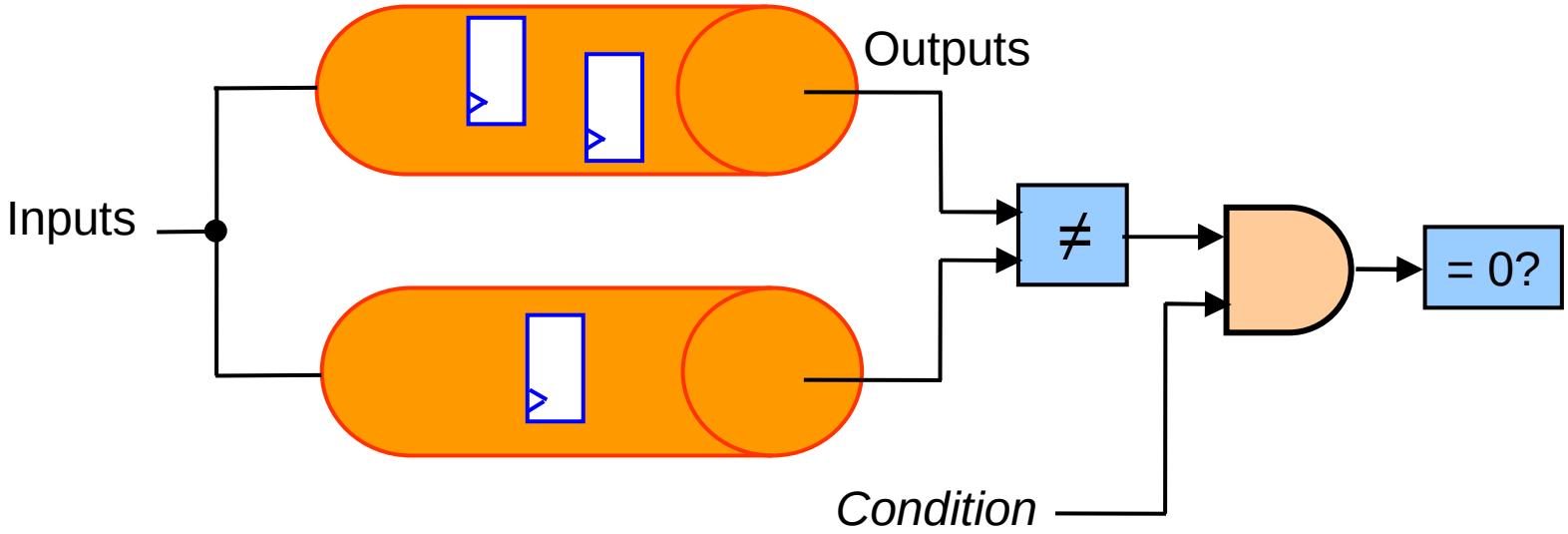
- No 1:1 state element requirement: generalizes CEC



- Greater applicability: e.g. to validate *sequential* synthesis
- Generality comes at a computational price: ***PSPACE***
  - \+ Though exist techniques to enhance ***scalability***

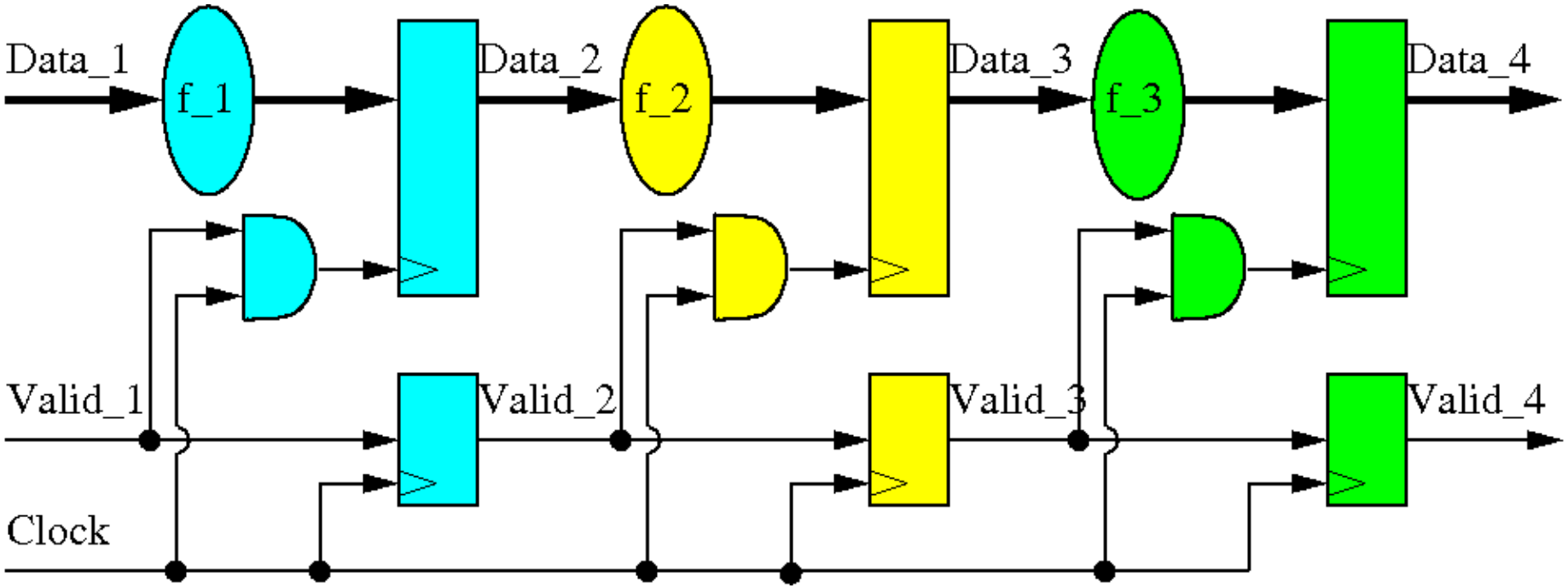
Conditional Sequential Equiv Checking (CSEC)

- Generalizes SEC: check equiv only under specific *conditions*

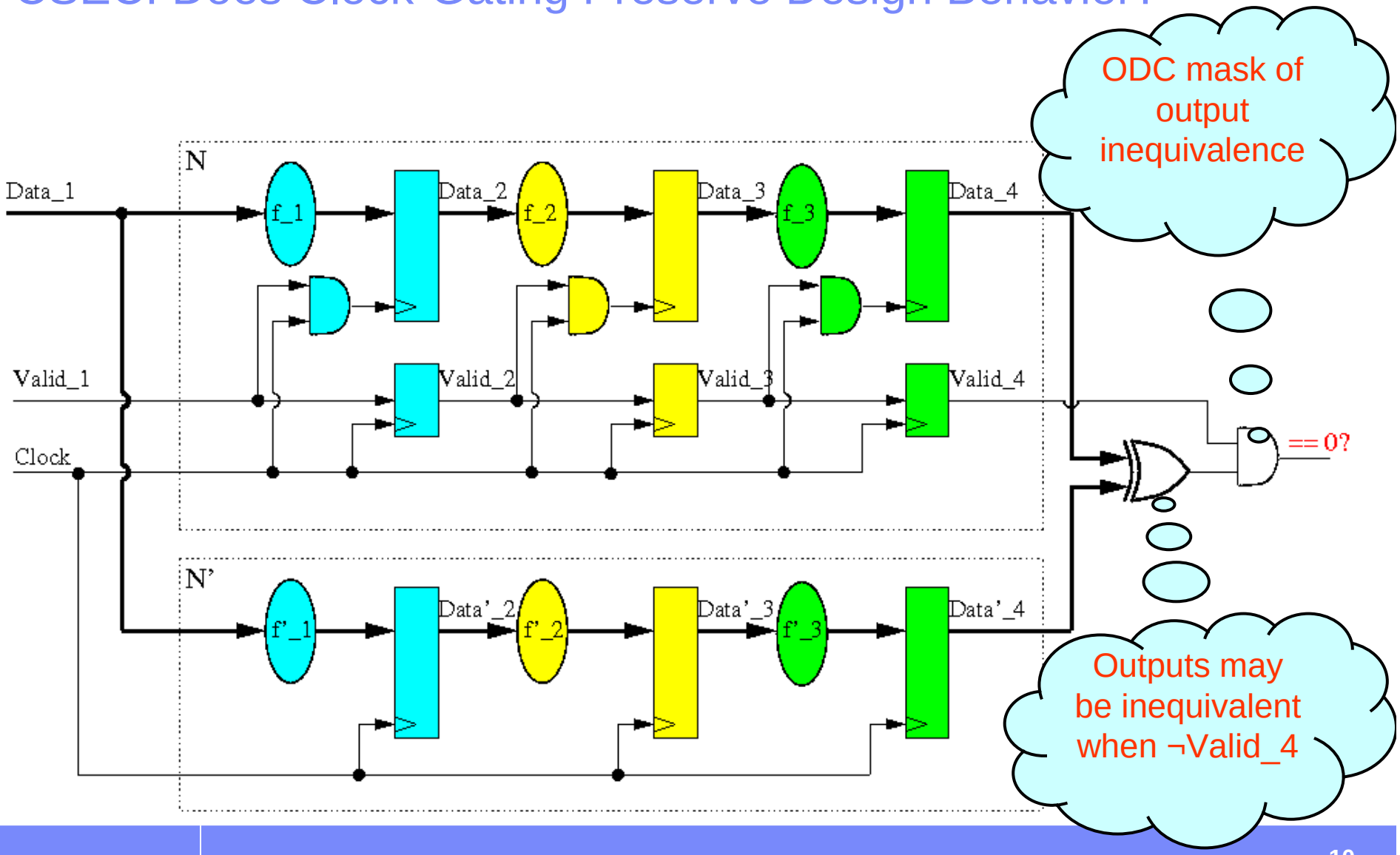


- While also PSPACE, practically ***much less scalable*** than SEC
  - *Output* inequivalence entails *internal* inequivalence
  - ***Precludes fundamental SEC scalability techniques***

Example: 3-Stage Clock-Gated Pipeline Design



CSEC: Does Clock-Gating Preserve Design Behavior?

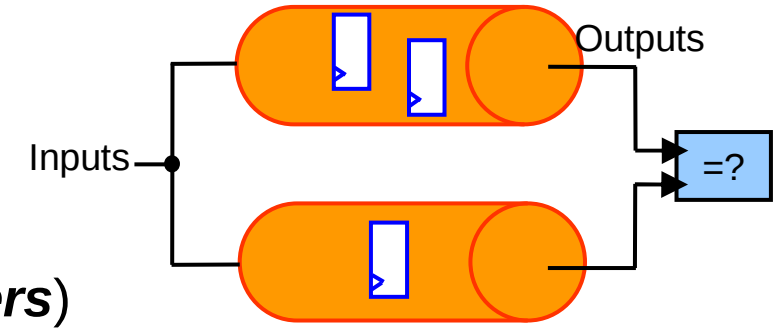


CSEC Problem Domains

- Clock gating: equivalence during valid computations
- Power gating: equivalence during power-up operation
- Post-reboot equivalence
- Generally: for sequential ODC-based optimizations
  - Equivalence during *care* conditions

- Increased demand for low-power devices +
- Increased sophistication of synthesis flows →
- Increased need for scalable CSEC techniques

Traditional SEC Flow

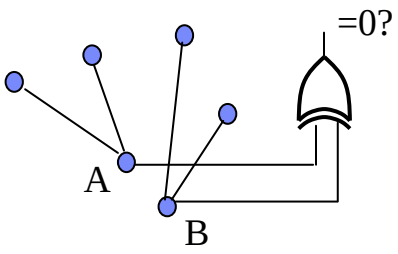


1) Postulate **internal equivalences** (***miters***)
2) Attempt to prove conjunction of miters
3) If successful, exit with proven internal equivalences
   - I/O equivalence often follows
4) Else refine unprovable miters, go to step 2

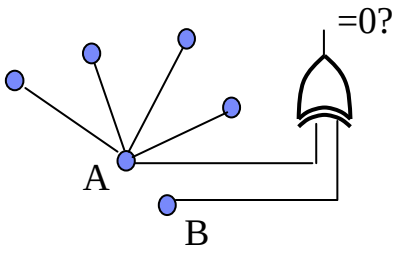
- Scalability requires ***assuming*** certain equivs while ***proving*** others
  I. *Conjunction* of miters often becomes *inductive*
  II. *Speculative reduction* enables dramatic speedup

Traditional SEC Flow

1) Postulate internal equivalences
2) ***Speculatively reduce w.r.t. postulated equivalences***
   - ***Similar to latch cutpointing in CEC – though preserves SEC results***



Miter without spec reduction



Miter with spec reduction

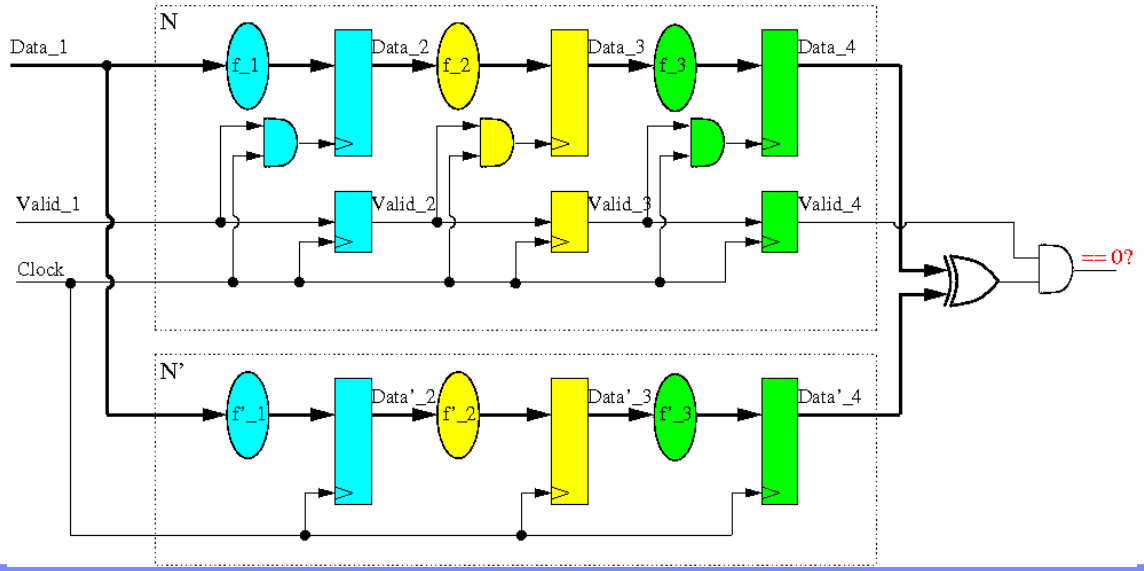
3) Attempt to prove miters on reduced design
4) If successful, exit with proven miters
5) Else refine unprovable miters, go to step 2

Speculative Reduction: Key to SEC Scalability

- Decomposes monolithic SEC problem into subproblems
- Reduces #gates in the fanin of each proof obligation
  - Many trivialized (A XOR A); all become easier to solve
- Enhances applicability of many algos vs. complex miters
  - *Structurally* tightens approximate analysis (e.g. interpolation)
  - Abstraction techniques more readily discard irrelevant logic, …
- **Enables 5 orders of magnitude speedup to SEC**
  - "Speculative Reduction-Based Scalable Redundancy Identification" DATE 2009

CSEC Precludes Speculative Reduction!

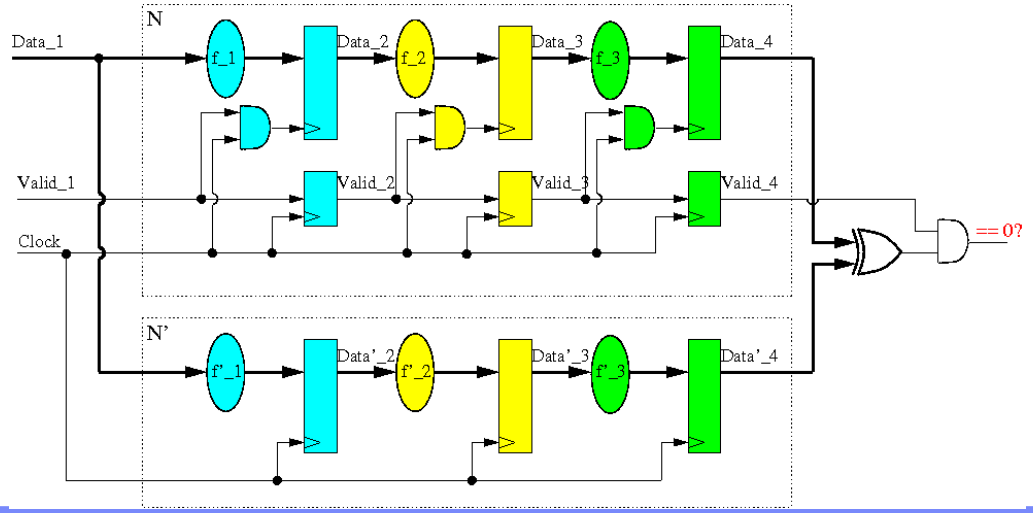
- CSEC problems exhibit little internal equivalence
  - ¬Valid_i → “probably” (Data_i ≠ Data’_i)
- *How can we approach scalability???*



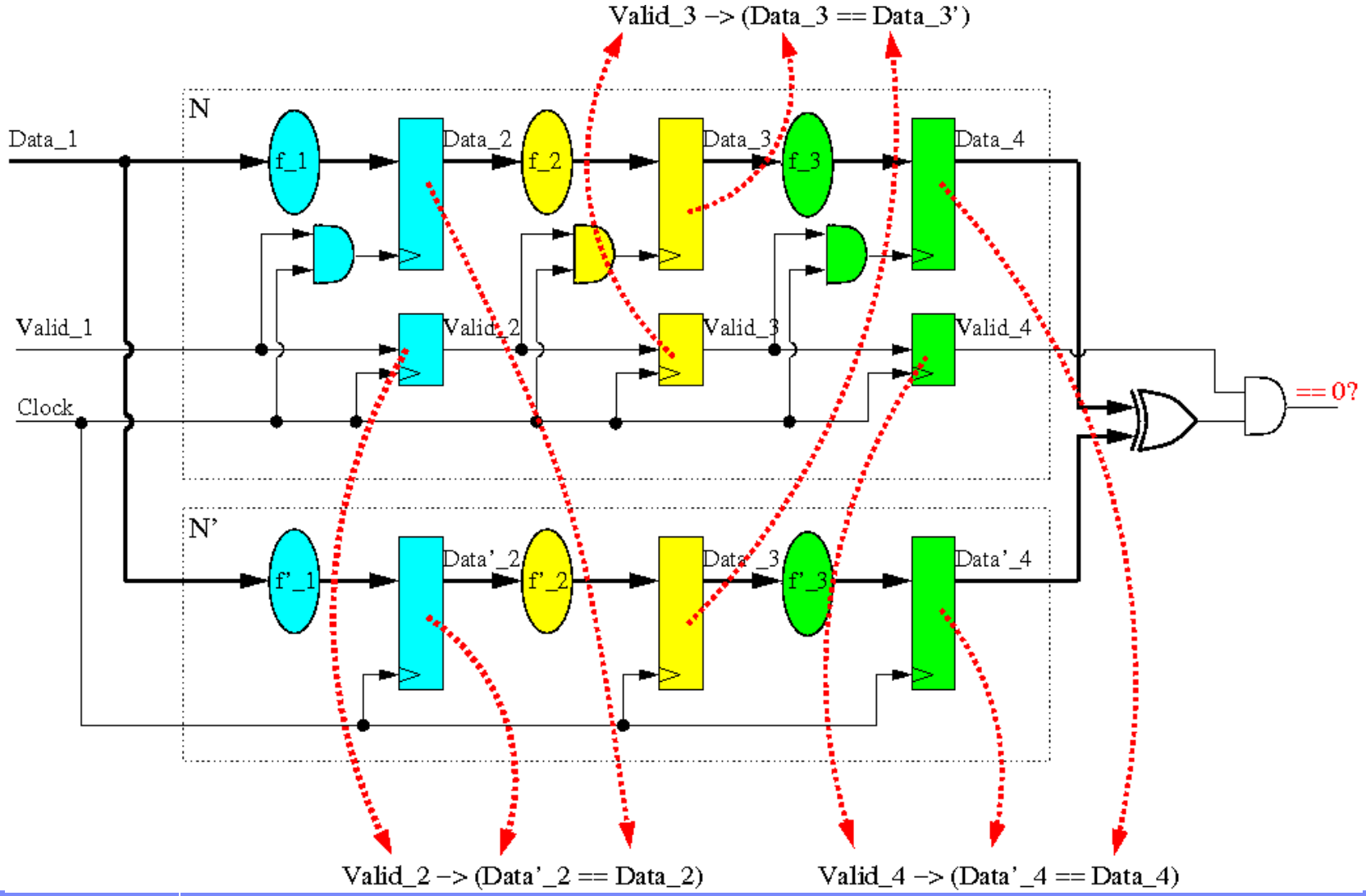
Scalability in CSEC

- Conditional *inequivalence* implies conditional *equivalence*
  - ¬Valid_i → “probably” (Data_i ≠ Data’_i)
  - Valid_i → **definitely** (Data_i = Data’_i)

- Idea: derive adequate *conditional equivalence invariants* to enable a scalable proof technique



Goal: Inductive Conditional Equivalence Invariant Set



CSEC Invariant Generation Flow

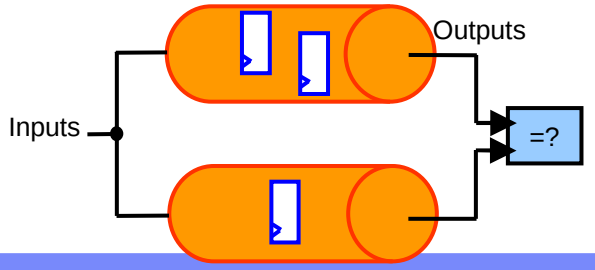
1) Postulate *conditional* equivalence invariants
2) Attempt to prove conjunction of invariants
3) If successful, exit with proven invariants
   - CSEC often becomes inductive under these invariants
4) Else refine unprovable invariants, go to step 2

Challenge 1: Huge #Candidate Invariants

- #Candidate invariants may be *cubic*: a → (b = c)

- Invariant generation is expensive
- Implication invariants a → b : quadratic #candidates
  - Often performed *lossily* to contain expense
  - "Inductively finding a reachable state space over-approximation" IWLS06

1) Leverage inherent CSEC correlation to reduce cubic → quadratic
   - a → (b = b') vs arbitrary
   - a → (b = c)



Challenge 1: Huge #Candidate Invariants

2) Leverage heuristic shortcuts to minimize #antecedents

- Limit antecendents to testbench-level signals defining ***Condition***
- 3-valued equivalence: ¬*tristated*(B) → (B = B')
- Use toggle/mismatch activity to correlate antecedent/consequent

- Different heuristics applicable to different CSEC problems
- Balancing act: efficiency vs. adequate invariants

Challenge 2: Efficiently Manage (In)valid Invariants

- Equiv class partition inadequate to represent candidates
  - Each (B = B') pair may have a distinct set of candidate antecedents

1) Represent candidates with sub-quadratic memory via *trie*

2) Use efficient bit-parallel simulator to prune large classes of invalid candidates upon each counterexample

3) Careful SW engineering between SAT, sim, trie

Experiment 1: Clock-Gated FPU

- All the bells and whistles: double-precision, 53x54 multiplier, fused multiply-add $a \times b + c$, 12 clock-period pipeline, …
  - >23k HDL lines, 21k state elements; 120k gates in CSEC formulation
- Complexity precludes single-instruction BMC in 24 hours
- Limited CSEC antecedents to testbench ***Condition*** logic
  - 11k of 254k candidate invariants proven in 4 hours, 3-step induction
  - Sim vs. SAT falsification ratio 679:1
- Could not solve otherwise without manual abstraction

Experiment 2: Power-Gated Arithmetic Unit

- 4-port out-of-order unit capable of arithmetic, ALU ops on 32-bit data, 16-entry register file
  - 13k lines RTL, 807 state elements, 22k gates
- CSEC used ternary equivalence mode
  - ¬tristated(B) → (B = B')
  - 961 of 1196 invariants proven in <3minutes, 100MB
- Could not solve otherwise
  - Manually-simplified version required >90 hours

Conclusion

- CSEC: an increasingly prevalent problem domain
- No internal equivalence!
  - Techniques to scale SEC to 1M+ gate designs *inapplicable*
- Presented an invariant generation approach tailored for CSEC
- Brute force, relies upon heuristics + careful SW engineering
- *The only* mechanism we have found for automated solution