

SAT-Based Design Debugging

Sharad Malik
Georg Weissenbacher
Princeton University

+ Verification vs. Debugging



- Verification
 - Make sure that there are no bugs
- Debugging
 - Observe error
 - What went wrong?
 - Root-cause analysis
- Understanding bugs is often more time consuming than finding them

+ Hardware bugs are expensive



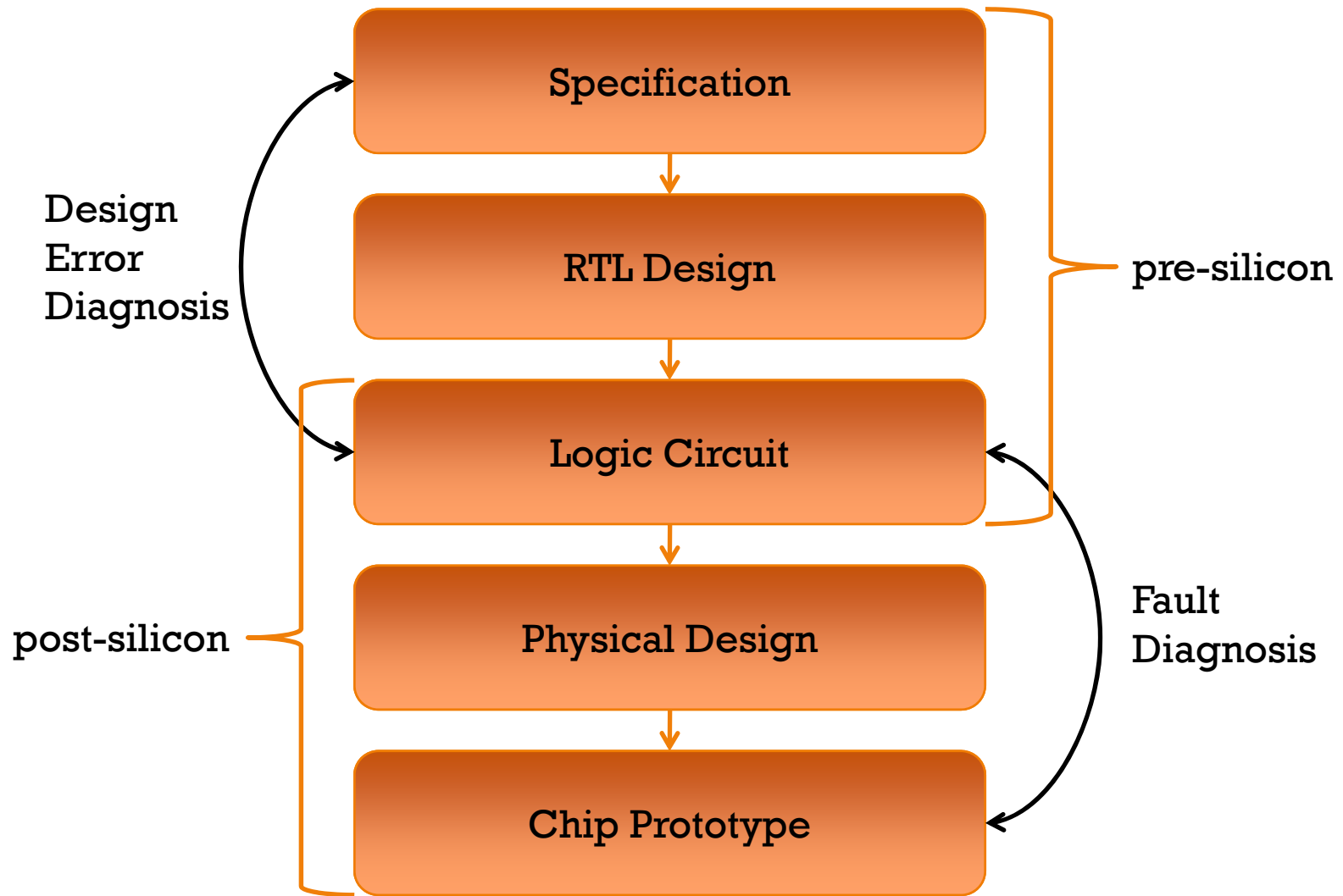
$$\frac{4195835}{3145727} = 1.333829068908037089$$

- Intel Pentium FDIV Bug 1994
 - incorrect results for division
 - due to errors in the entries in the lookup table used by the digital divide operation algorithm
- Total cost associated with replacement: \$475 million

+ Outline



- Hardware Design/Verification Flow
- SAT Background
 - SAT Encoding of Logic Designs
 - Satisfying Assignments for Debugging
 - Unsatisfiable Instances for Debugging
- Pre-Silicon Debug
 - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
 - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary



[Smith, Veneris, Ali, Viglas 2004]

+ Verification Techniques



- Testing
 - White Box Testing
 - Logic Simulation
 - Black Box Testing
- Formal Verification
 - Automated Theorem Proving (ATP)
 - Model Checking
 - (Partial) Specifications in Temporal Logic
 - Bounded Model Checking
 - Equivalence Checking
- ...

+ Applicability of Verification Techniques



	logic simulation	model checking
pre-silicon (full observability)	✓ + ease of use - limited coverage - simulation is slow	✓ + complete coverage - scalability issues - complex formalisms
post-silicon (limited observability)	✓ + real-time execution - still limited coverage - manufacturing cost	✗ not applicable

+ Find Errors, Localize Faults



- Error:
Discrepancy between observed and expected behavior
- Fault:
Abnormal condition, may cause an error
 - Electrical: signal interference in manufactured prototypes
 - Logical: missing case statement in hardware design
- Temporal and spatial localization of faults
“typically dominate[s] the effort expended during the debug process for a bug” [Josephson 2006]
- This talk’s focus:
Automated Fault Localization

+ Outline



- Hardware Design/Verification Flow
- **SAT Background**
 - **SAT Encoding of Logic Designs**
 - Satisfying Assignments for Debugging
 - Unsatisfiable Instances for Debugging
- Pre-Silicon Debug
 - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
 - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary

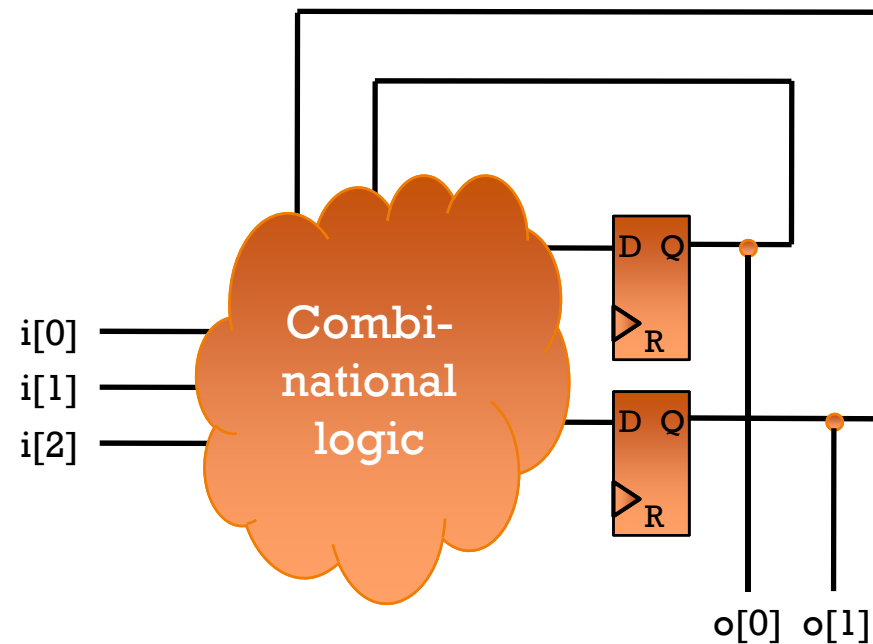
+ Register Transfer Level

```
module seq_moore(i, o, clk, reset)
  input [2:0] i;
  input clk;
  input reset;
  output o;
  reg [1:0] o;

  reg [1:0] Q; // state variables
  reg [1:0] D; // next state output

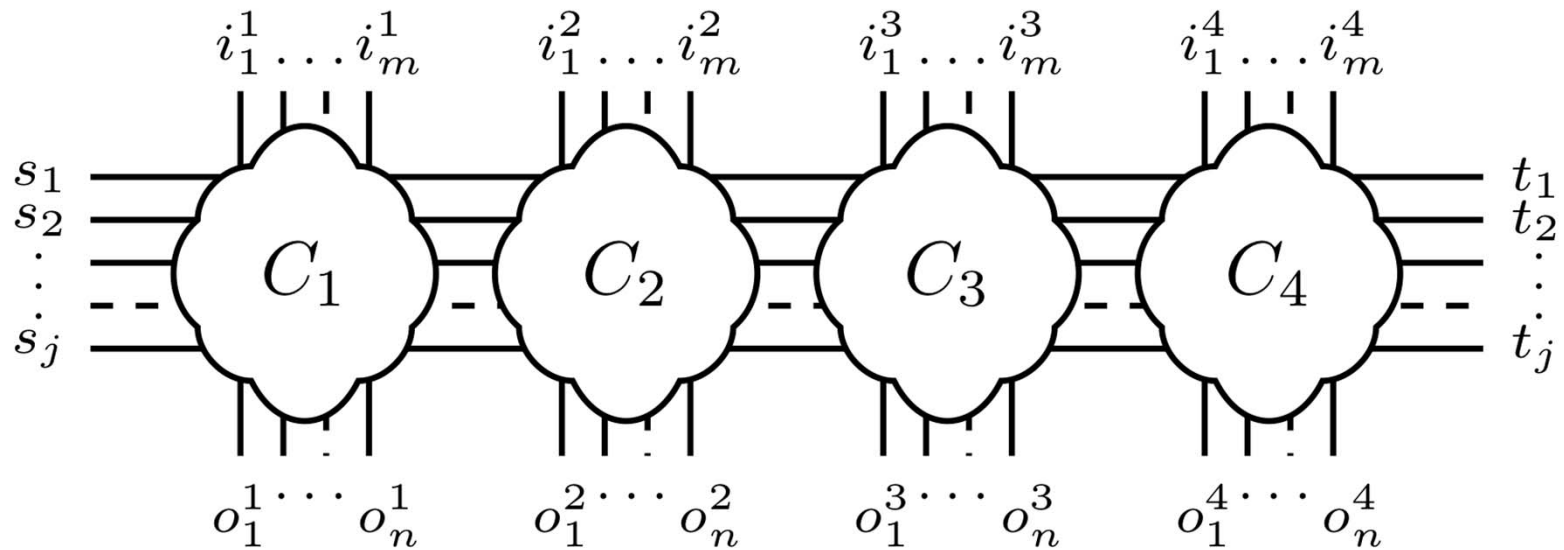
  always @(posedge clk)
    Q = D;

  always @(Q or reset or ...)
  begin
    case (Q)
    ...
```



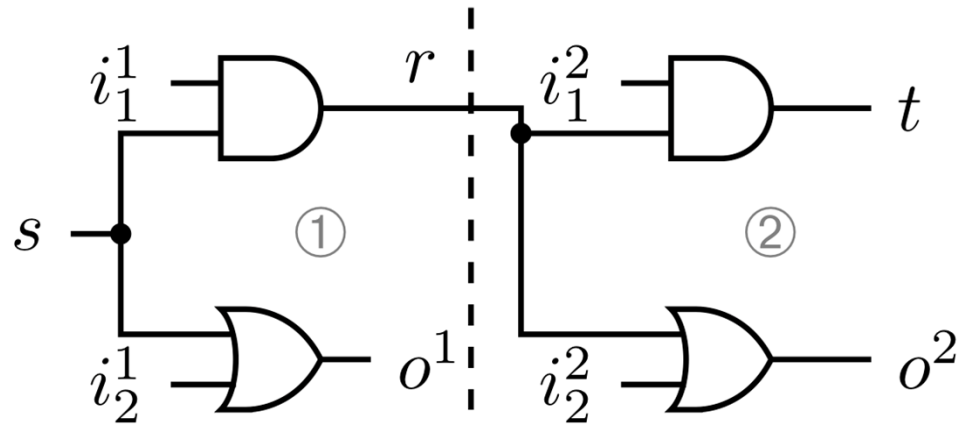
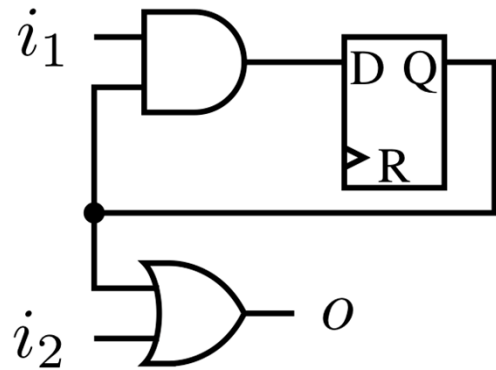
Huffman Model

+ Iterative Logic Array



[Abramovici, Breuer, Friedman 1990]

+ Iterative Logic Array (Example)



D

D

cycle ①	$(\bar{r} + i_1^1) \cdot (\bar{r} + s) \cdot (\overline{i_1^1} + \bar{s} + r)$	$(\overline{i_2^1} + o^1) \cdot (\bar{s} + o^1) \cdot (\overline{o^1} + i_2^1 + s)$
cycle ②	$(\bar{t} + i_1^2) \cdot (\bar{t} + r) \cdot (\overline{i_1^2} + \bar{r} + t)$	$(\overline{i_2^2} + o^2) \cdot (\bar{r} + o^2) \cdot (\overline{o^2} + i_2^2 + r)$

+ Outline

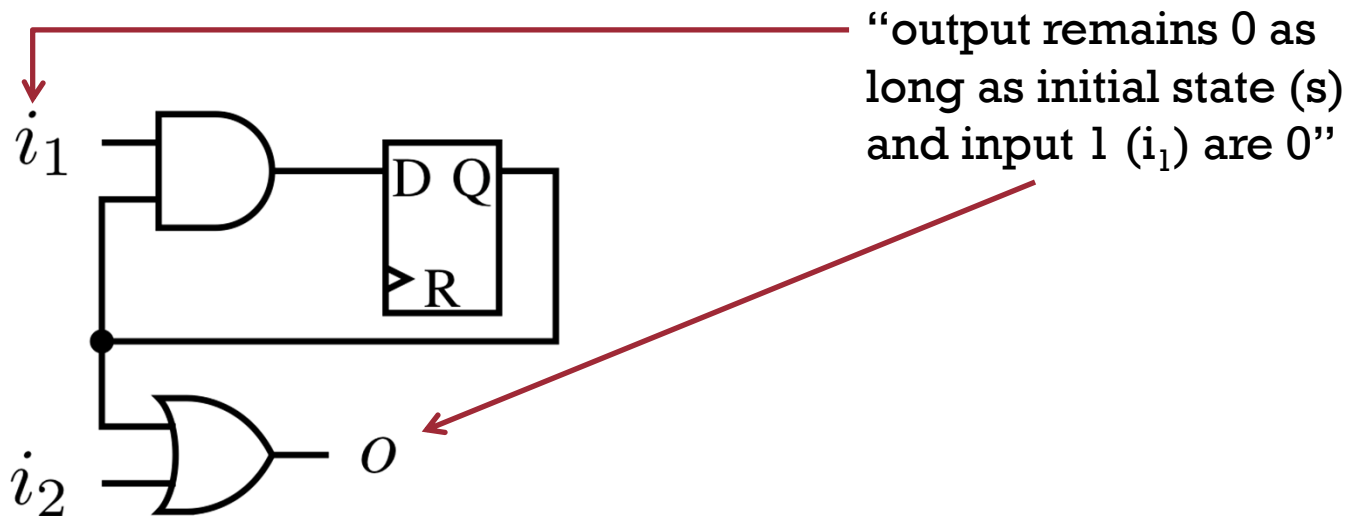


- Hardware Design/Verification Flow
- **SAT Background**
 - SAT Encoding of Logic Designs
 - **Satisfying Assignments for Debugging**
 - Unsatisfiable Instances for Debugging
- Pre-Silicon Debug
 - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
 - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary

+ Applications of SAT Encoding

Property Checking

- How do we know that the RTL design is correct?
- Check whether certain specified properties hold
- If not, we want a **counterexample**

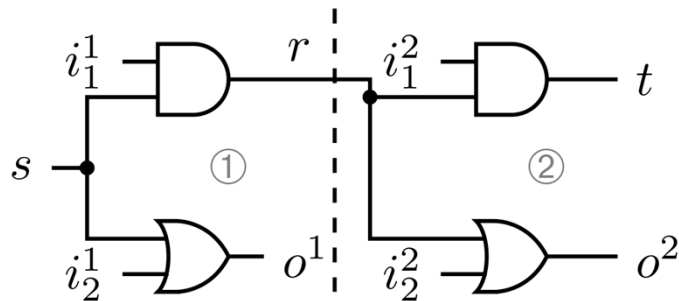


+ Applications of SAT Encoding

Property Checking

- Bounded Model Checking

Check whether given property holds for k cycles



“output remains 0 as long as initial state (s) and input 1 (i_1) are 0”

$$\left(\overline{s} \cdot \overline{i_1^1}\right) \rightarrow \overline{o_1^1}$$

$$\left(\overline{s} \cdot \overline{i_1^1} \cdot \overline{i_1^2}\right) \rightarrow \overline{o_2^2}$$

+ Applications of SAT Encoding

Property Checking

- Property holds if unfolding and negated property **UNSAT**
- Any satisfying assignment is a **counterexample** to the claim

	\sqsupset	\supset
cycle ①	$(\bar{r} + i_1^1) \cdot (\bar{r} + s) \cdot (\bar{i}_1^1 + \bar{s} + r)$	$(\bar{i}_2^1 + o^1) \cdot (\bar{s} + o^1) \cdot (\bar{o}^1 + i_2^1 + s)$
cycle ②	$(\bar{t} + i_1^2) \cdot (\bar{t} + r) \cdot (\bar{i}_1^2 + \bar{r} + t)$	$(\bar{i}_2^2 + o^2) \cdot (\bar{r} + o^2) \cdot (\bar{o}^2 + i_2^2 + r)$

$$\left[(\bar{s} \cdot \bar{i}_1^1) \rightarrow o \right]$$

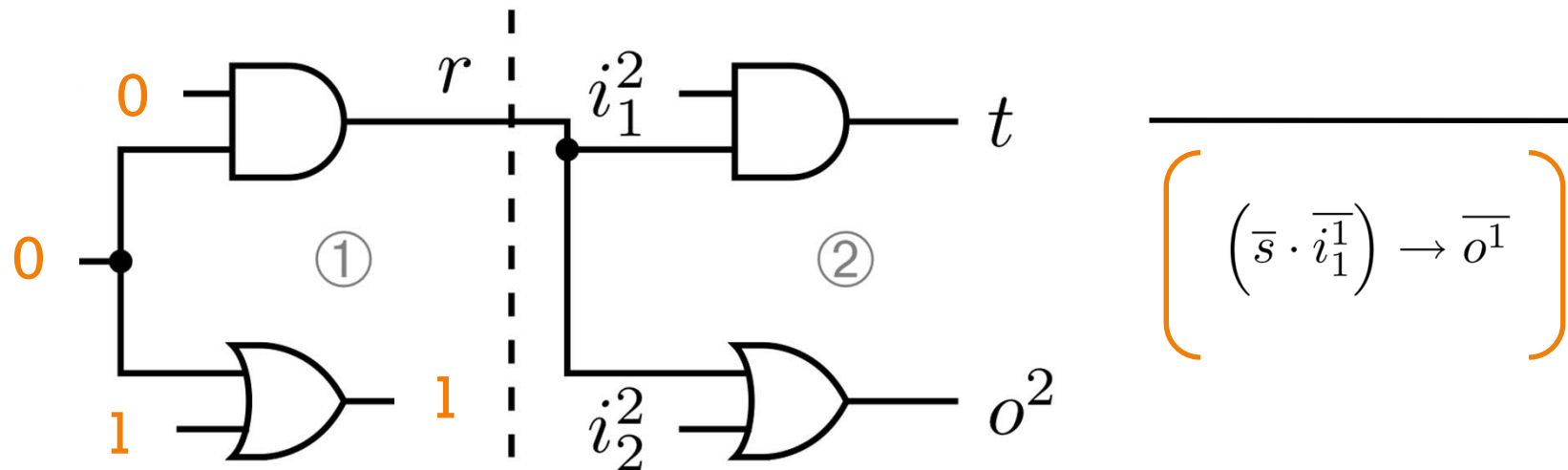
X

$$\left[(\bar{s} \cdot \bar{i}_1^1 \cdot \bar{i}_1^2) \rightarrow \bar{o}^2 \right]$$

+ Applications of SAT Encoding

Property Checking

- Any satisfying assignment is a **counterexample** to the claim



+ Counterexamples

More valuable than correctness proofs?

“One of the most important advantages of model checking [...] is its counterexample facility. [...] The counterexamples can be essential in finding subtle errors in designs.”

[Clarke, Grumberg, McMillan, Zhao 1995]

- However, a counterexample still doesn't tell us
 - *what* went wrong, and
 - *where* it went wrong.

Root Cause Analysis or Fault Localization

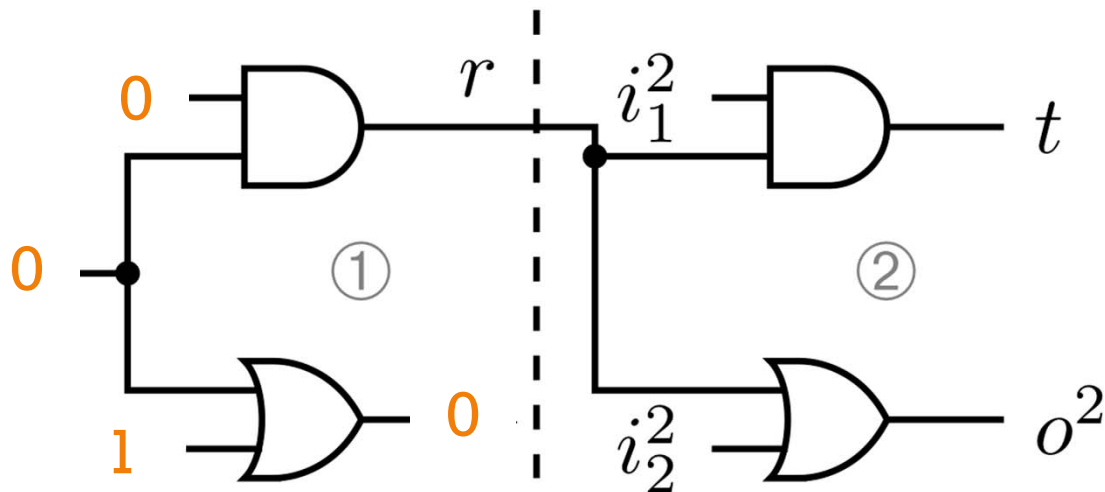
+ Outline



- Hardware Design/Verification Flow
- **SAT Background**
 - SAT Encoding of Logic Designs
 - Satisfying Assignments for Debugging
 - **Unsatisfiable Instances for Debugging**
- Pre-Silicon Debug
 - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
 - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary

+ Unsatisfiable Instances

- Constrain a faulty design with the *correct* input/output
- The resulting formula is unsatisfiable
- Locate gates that are inconsistent with desired behavior



+ Minimal Correction Set (MCS)

- Given an **UNSAT** instance
 - Minimal subset of clauses that must be dropped to make instance satisfiable
 - Any subset of an MCS is not a correction set

$$(\bar{r} + \bar{s} + t) \quad (\bar{r} + s) \quad (r) \quad (s) \quad (\bar{t})$$

dropping *both* $(\bar{r} + s)$ and (s) “corrects” the formula

- Remaining clauses are consistent
- Is there more than one MCS in our example?

+ Minimal Correction Set (MCS)



- The following formula has a single minimum MCS:
 - Least cardinality

$$(\bar{s}) \quad (\bar{r} + s) \quad (r) \quad (s)$$

- $\{(r), (s)\}$ is minimal but not minimum.
- The formula has three different MCSes.

+ MAX SAT



- The complement of a minimum correction set
 - Largest subset of clauses that can be satisfied
- Given an **UNSAT** instance

$$(\bar{r} + \bar{s} + t) \quad (\bar{r} + s) \quad (r) \quad (s) \quad (\bar{t})$$



What is the largest subset of clauses that can be satisfiable?

- The complement of any MAX-SAT solution is an MCS
- Converse doesn't hold:
Complement of MCS is *maximal* set of satisfiable clauses

+ Partial MAX SAT



- Maximum subset of clauses that can be satisfied
given certain clauses cannot be dropped
- Given an UNSAT instance

$$(\bar{r} + \bar{s} + t) \quad (\bar{r} + s) \quad (r) \quad (s) \quad (\bar{t})$$


which clauses do we have to drop to make it satisfiable?
(the *pinned* clauses can't be dropped)

+ Minimal Unsatisfiable Subsets (MUS) and Unsatisfiable Cores

- An *unsatisfiable subset* is an inconsistent subset of the clauses of the original formula
 - Also, referred to as an UNSAT core

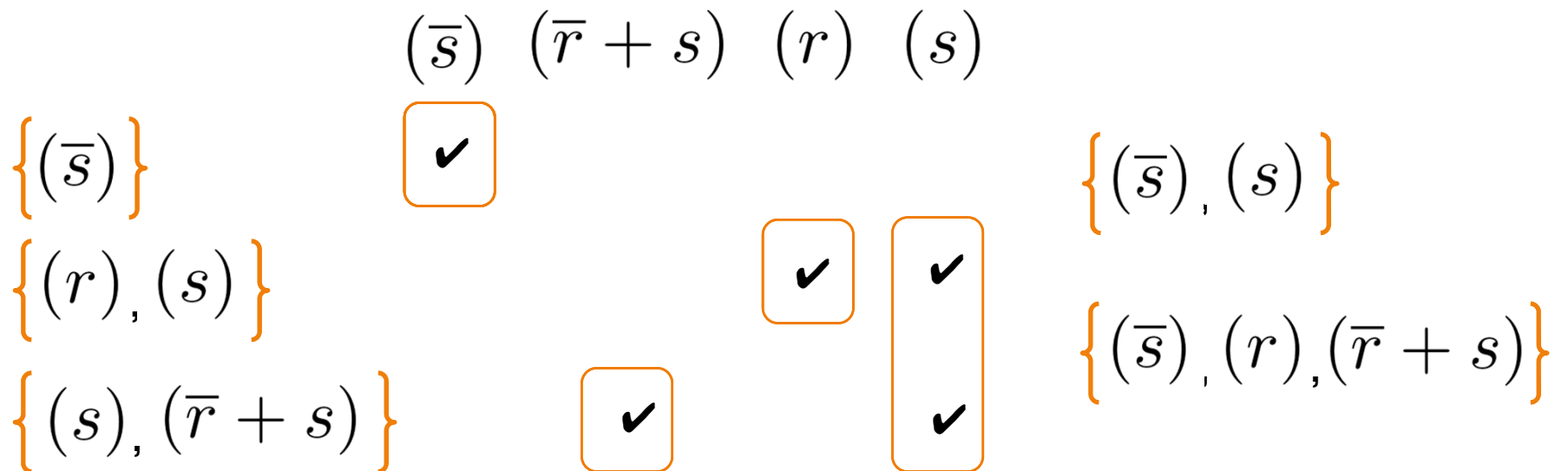
$$\boxed{(\bar{s})} \quad (\bar{r} + s) \quad (r) \quad \boxed{(s)}$$

- An unsatisfiable subset is *minimal* if dropping one of its clauses makes it satisfiable

+ MCSes and MUSes



- Generate *all* MCSes for a set of clauses

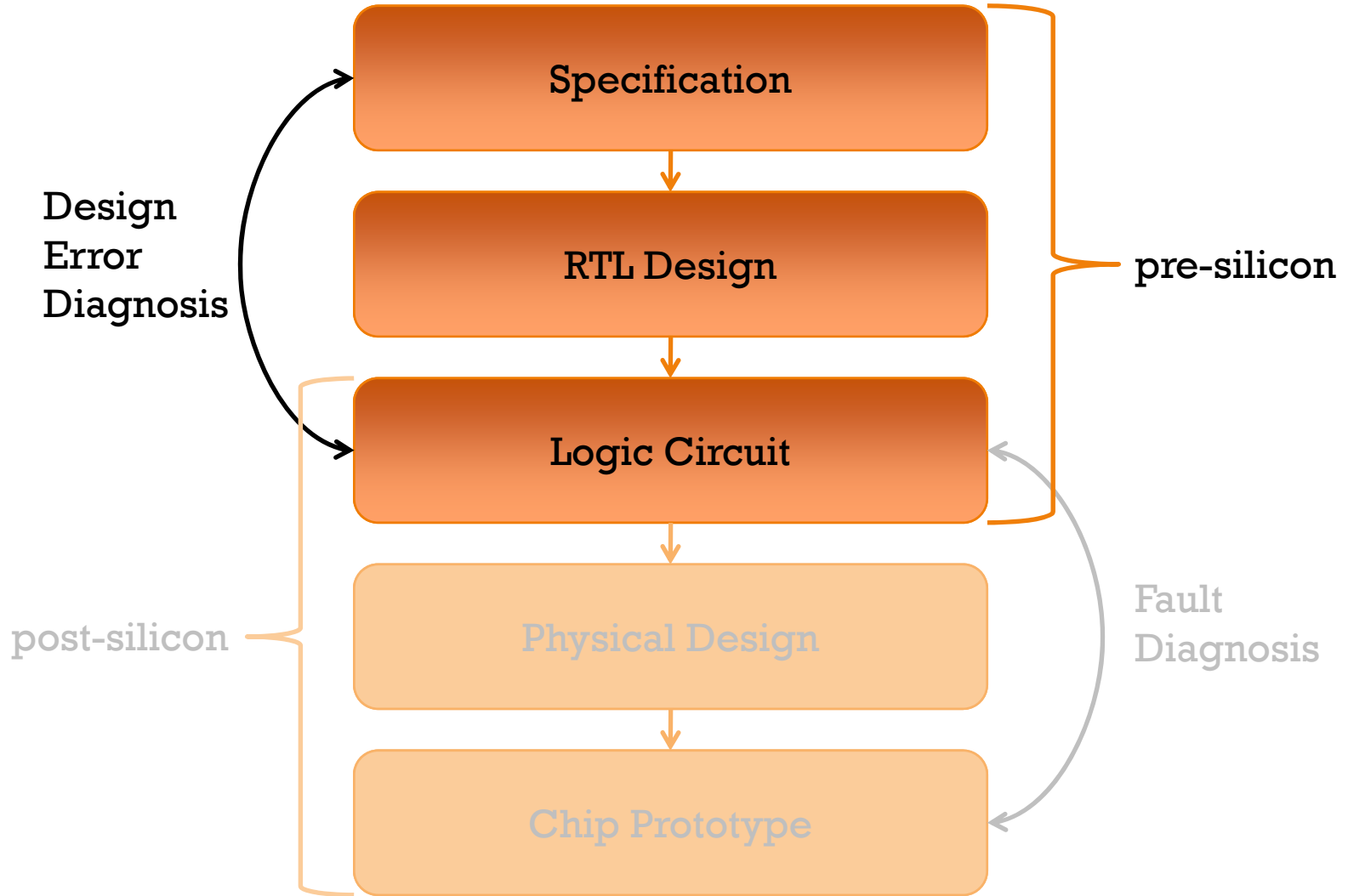


- Each *hitting set* of the MCSes is an MUS
- Each hitting set of all MUSes is an MCS
- Therefore, dropping the clauses of an MCS “deactivates” all cores

+ Outline

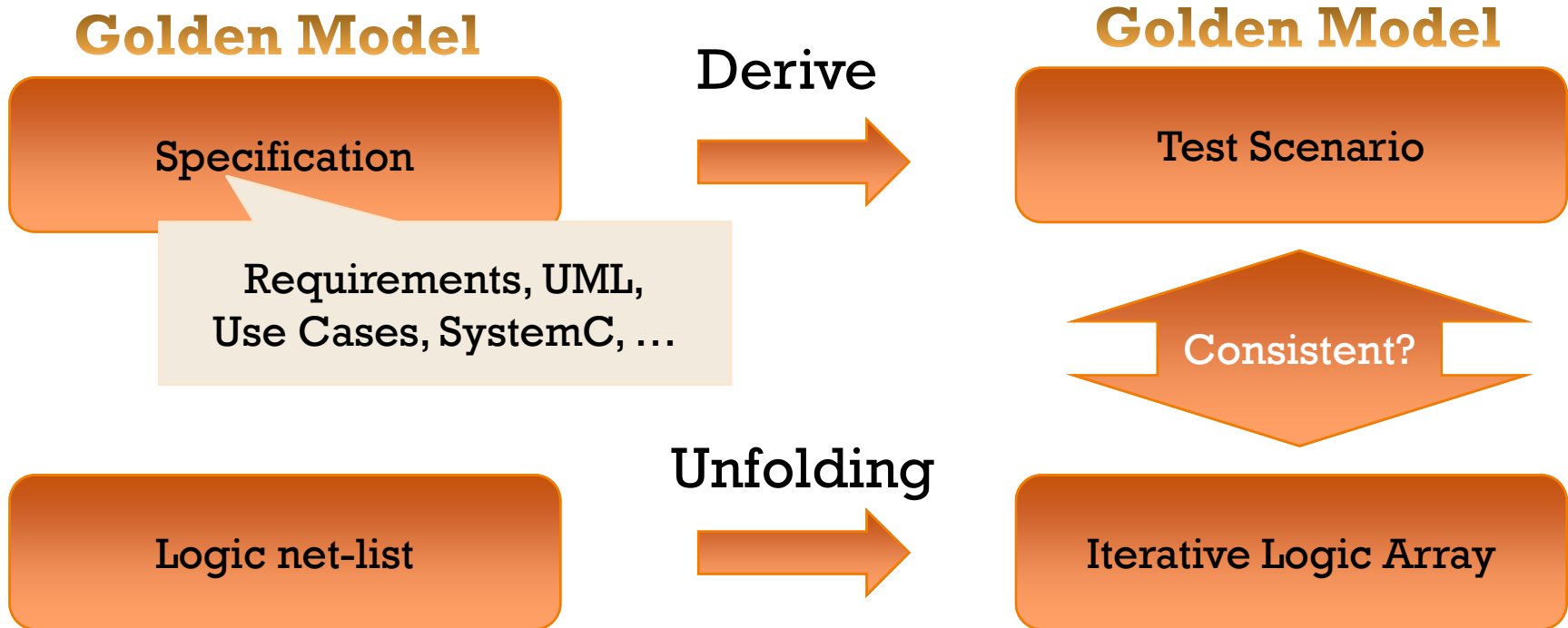


- Hardware Design/Verification Flow
- SAT Background
 - SAT Encoding of Logic Designs
 - Satisfying Assignments for Debugging
 - **Unsatisfiable Instances for Debugging**
- **Pre-Silicon Debug**
 - **Localizing faults in Register-Transfer-Level (RTL) designs**
- Post-Silicon Validation
 - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- Summary



+ Pre-Silicon Debug

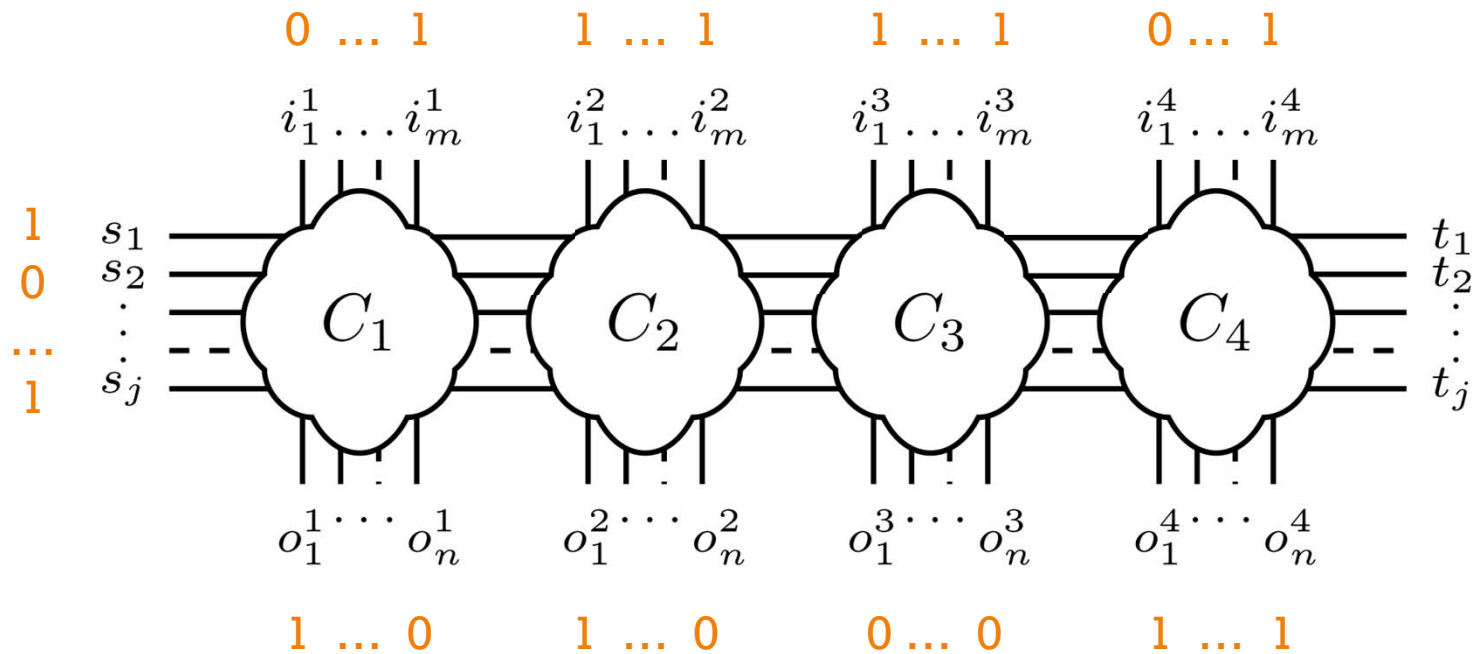
Problem Definition



+ Test Case as Circuit Constraints



- Test scenario is modeled as constraint for iterative logic array



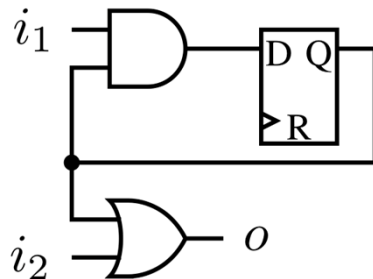
+ Test Scenarios as Constraints

Example

Specification

“output remains 0 as long as initial state (s) and input 1 (i_1) are 0”

Logic net-list

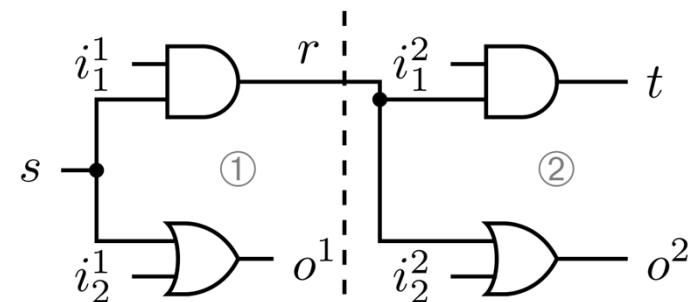


time-frame 1

$s = 0$
 $i_1 = 0$
 $i_2 = 0$
 $o = 0$

time-frame 2

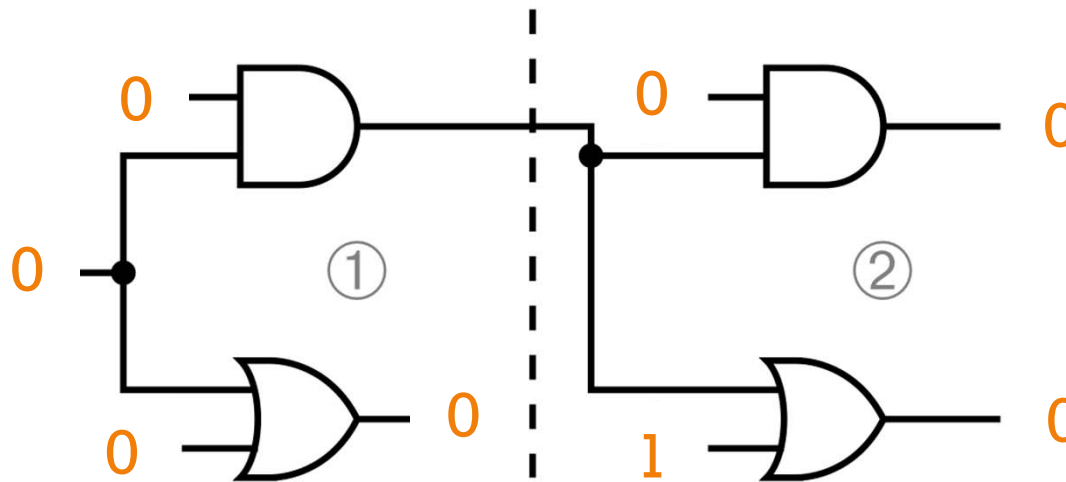
$i_1 = 0$
 $i_2 = 1$
 $o = 0$
 $t = 0$



+ Test Scenarios as Constraints

Example

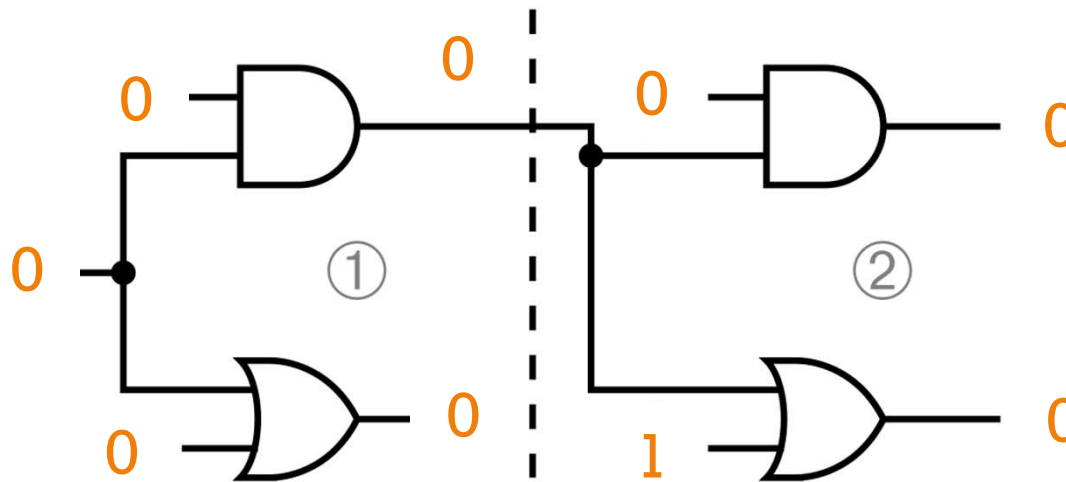
- Add test-scenario as constraints to circuit
- The corresponding CNF formula is **inconsistent**



+ Test Scenarios as Constraints

Example

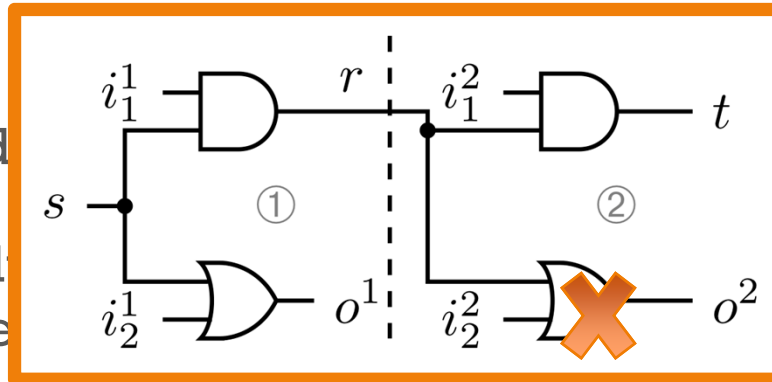
- *Detecting* the error is only half the story
- Manually **localizing** the fault causing a known error is tedious



+ Fault Localization Using MCSes

Example

- Use MCSes to identify faults
- Input/output values (we're not interested in internal values)



$$(\bar{s}) \quad (\bar{i}_1^1) \quad (\bar{i}_2^1) \quad (\bar{o}^1) \quad (\bar{t}) \quad (\bar{i}_1^2) \quad (i_2^2) \quad (\bar{o}^2)$$

D

cycle ① $(\bar{r} + i_1^1) \cdot (\bar{r} + s) \cdot (\bar{i}_1^1 + \bar{s} + r)$

cycle ② $(\bar{t} + i_1^2) \cdot (\bar{t} + r) \cdot (\bar{i}_1^2 + \bar{r} + t)$

D

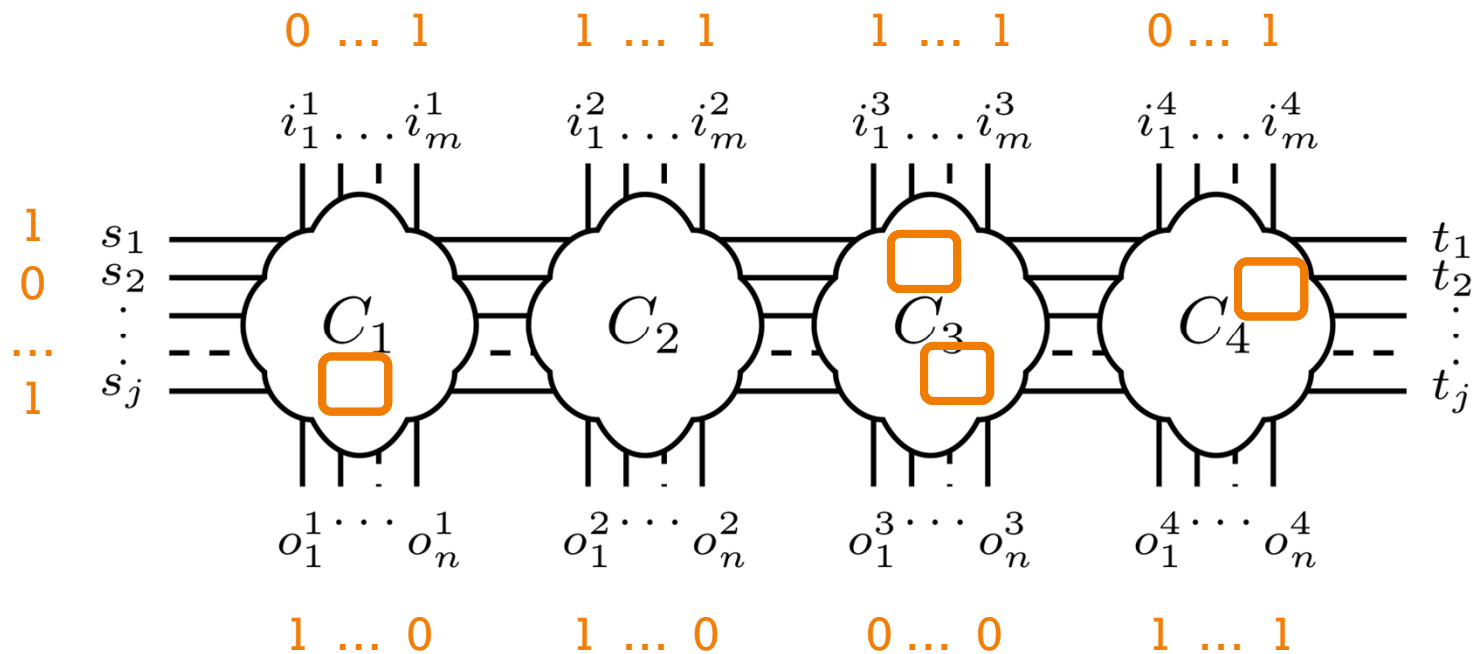
$(\bar{i}_2^1 + o^1) \cdot (\bar{s} + o^1) \cdot (\bar{o}^1 + i_2^1 + s)$

$(\bar{i}_2^2 + o^2) \cdot (\bar{r} + o^2) \cdot (\bar{o}^2 + i_2^2 + r)$

+ Fault Localization Using MCSes

General Methodology

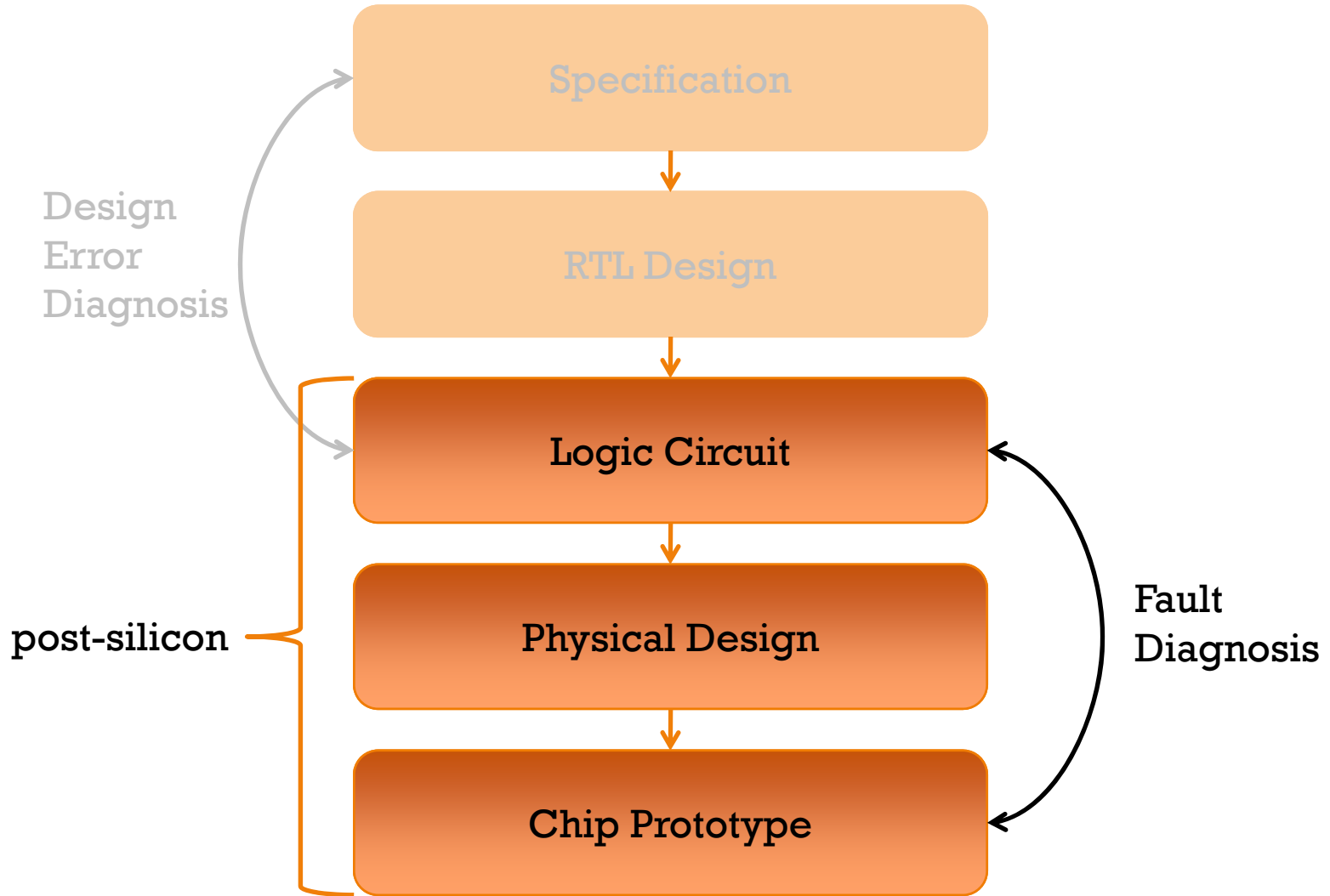
- Generate ILA constrained with test case (in CNF)
- Compute *all* MCSes:
Each MCS represents a set of potential fault locations



+ Outline



- Hardware Design/Verification Flow
- SAT Background
 - SAT Encoding of Logic Designs
 - Satisfying Assignments for Debugging
 - **Unsatisfiable Instances for Debugging**
- Pre-Silicon Debug
 - Localizing faults in Register-Transfer-Level (RTL) designs
- **Post-Silicon Validation**
 - **Localizing faults in manufactured prototypes**
- Outlook: Fault Localization in Software
- Summary



+ Post-Silicon Validation

Problem Definition



Golden Model

Logic Circuit

Unfolding



Golden Model

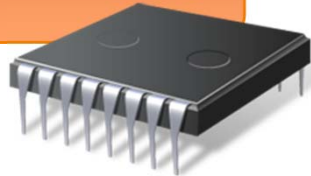
Iterative Logic Array

Consistent?

Testing



Chip Prototype



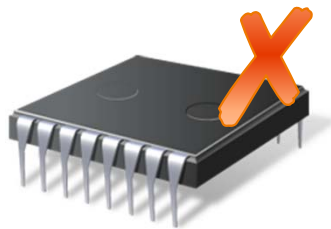
Test Result

+ Post-Silicon Validation

What has changed compared to pre-silicon?

- Include a different class of faults: electrical faults
 - Not in every time-frame: may be *transient* or *intermittent*
- The *test result* represents the erroneous behavior
- The net-list is the fault-free *golden model*
 - We assume functional correctness when considering electrical bugs

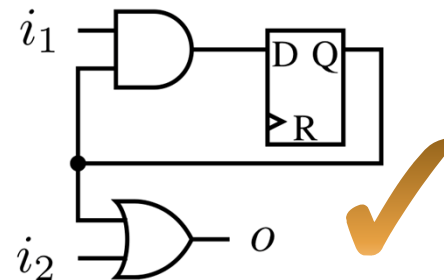
Faulty Chip



Undesired
Test Result



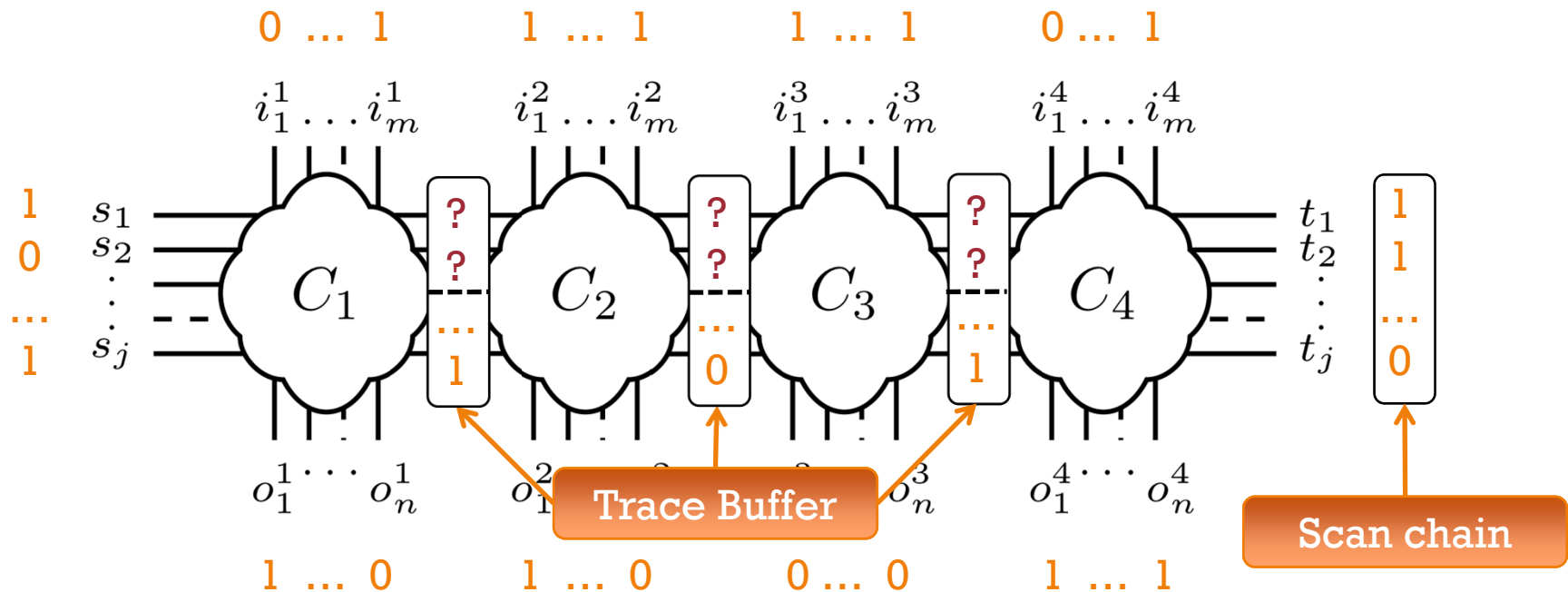
Golden Model



+ Post-Silicon Validation

What has changed compared to pre-silicon?

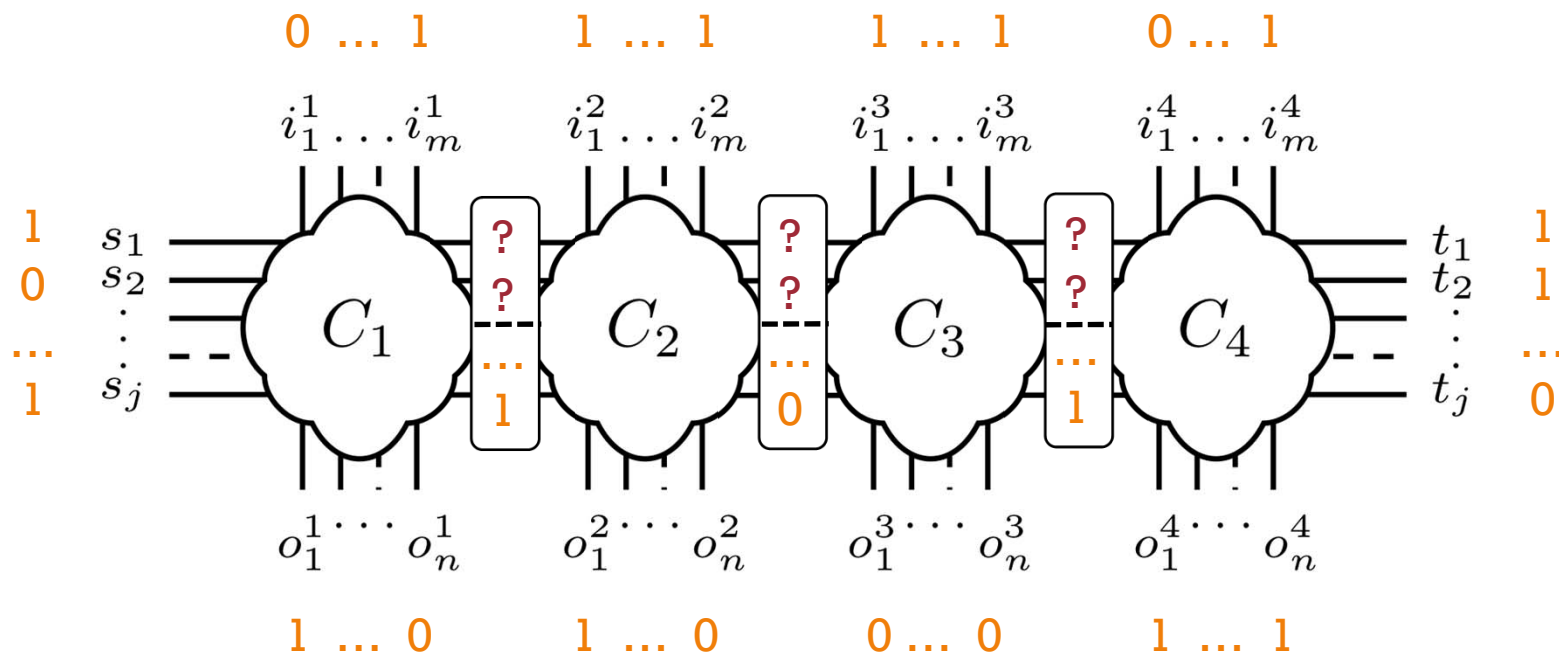
- Limited observability of signals in manufactured chip
 - Trace buffers: Limited recording of select signals
 - Scan chains: Read-out after chip execution stopped



+ Test Results as Circuit Constraints



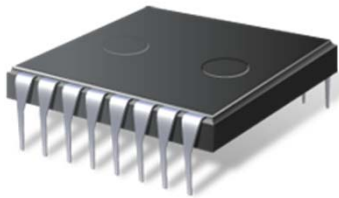
- Test results used as constraint for iterative logic array
- ? = information was not recorded



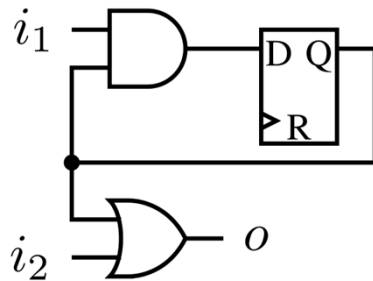
+ Test Results as Constraints

Example

Test run



Logic net-list

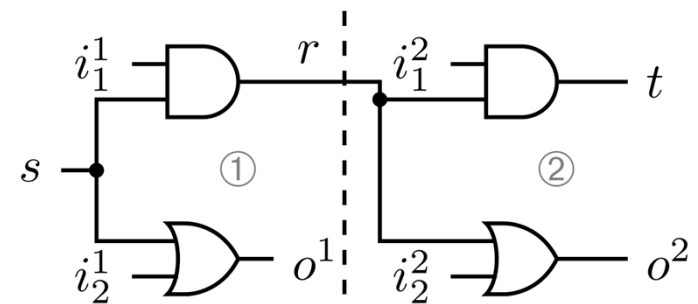


time-frame 1

$s = 0$
 $i_1 = ?$
 $i_2 = 0$
 $o = 0$
 $r = ?$

time-frame 2

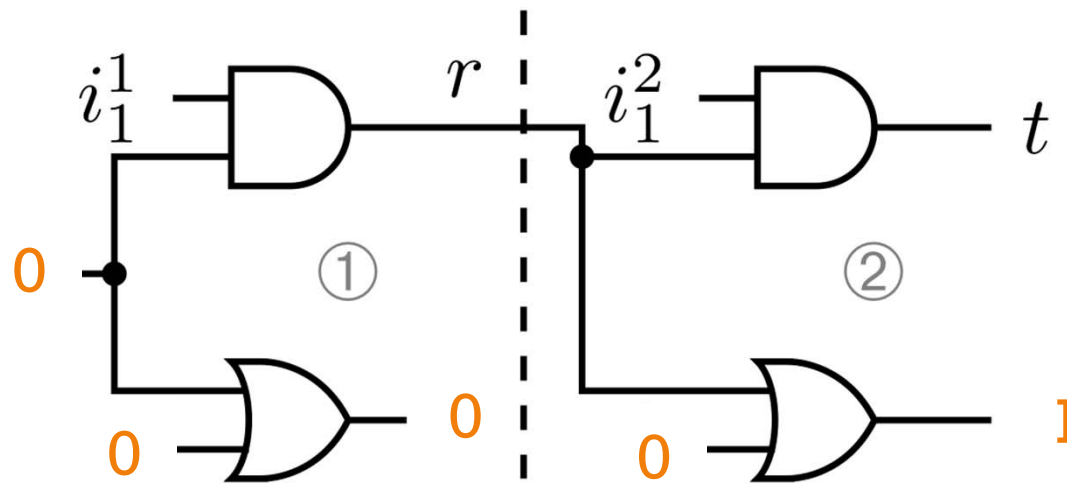
$r = ?$
 $i_1 = ?$
 $i_2 = 0$
 $o = 1$
 $t = ?$



+ Test Results as Constraints

Example

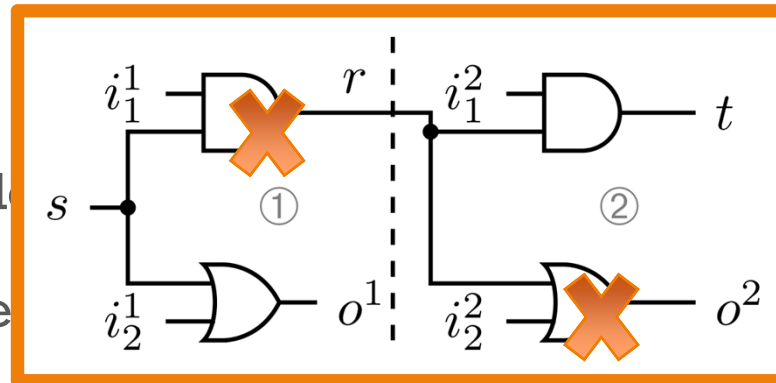
- Add test results as constraints to circuit
- The corresponding CNF formula is **inconsistent**



+ Fault Localization Using MCSes

Example

- Use MCSes to identify faults
- Recorded test results



$$(\bar{s}) (\bar{i}_2^1) (\bar{o}^1) \quad (\bar{i}_2^2) (o^2)$$

D

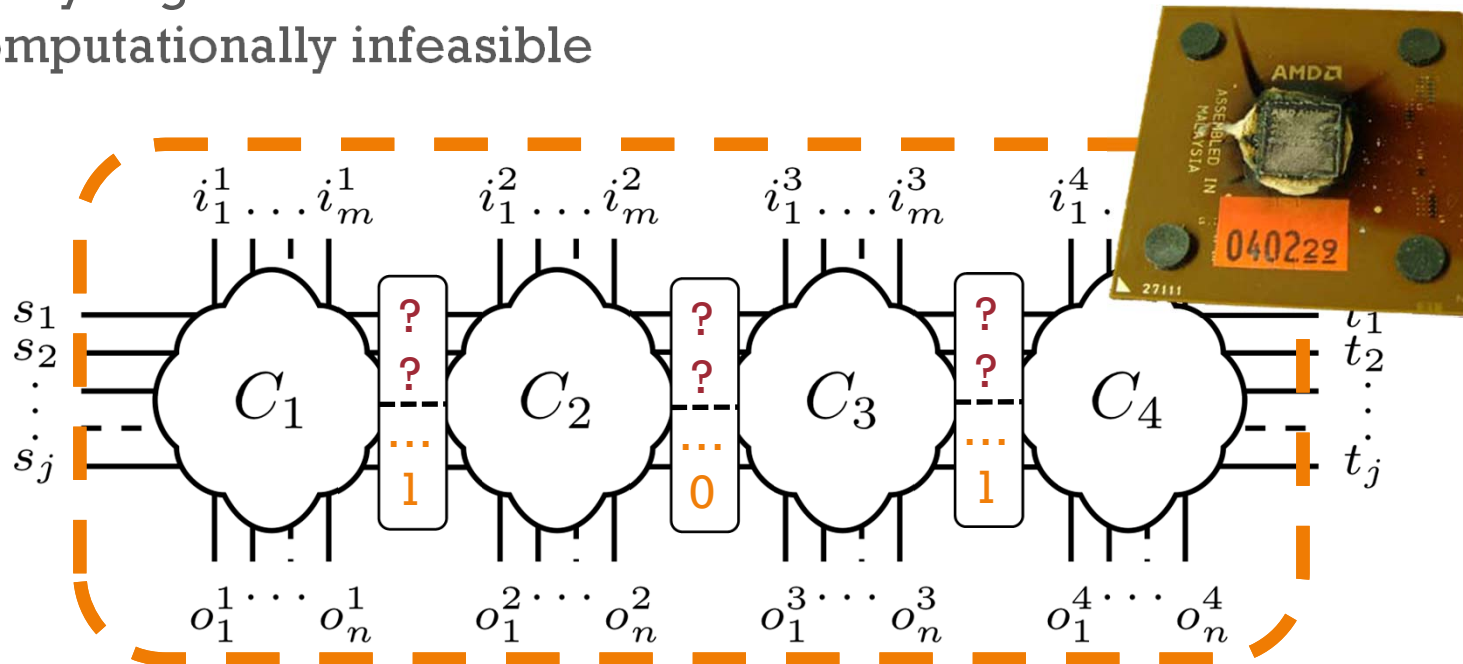
cycle ①	$(\bar{r} + i_1^1) \cdot (\bar{r} + s) \cdot (\bar{i}_1^1 + \bar{s} + r)$	$(\bar{i}_2^2 + o^1) \cdot (\bar{s} + o^1) \cdot (\bar{o}^1 + i_2^1 + s)$
cycle ②	$(\bar{t} + i_1^2) \cdot (\bar{t} + r) \cdot (\bar{i}_1^2 + \bar{r} + t)$	$(\bar{i}_2^2 + o^2) \cdot (\bar{r} + o^2) \cdot (\bar{o}^2 + i_2^2 + r)$

D

+ Post-Silicon Faults and MCSes

Limits of Scalability

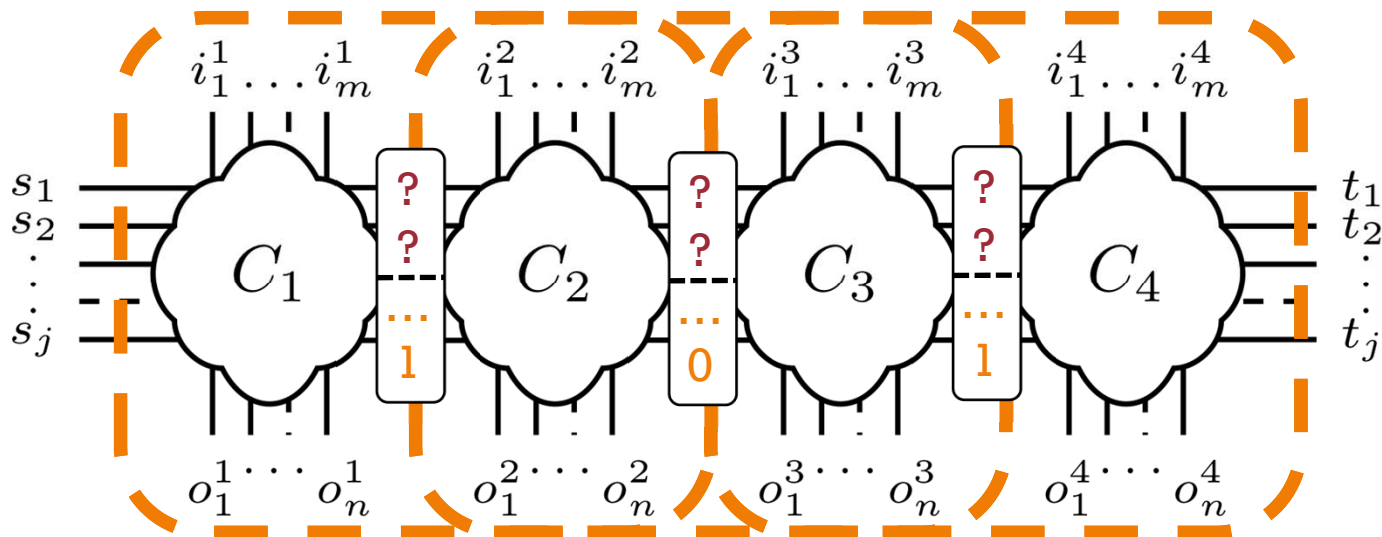
- Limited observability results in harder decision problems
 - In Pre-Silicon: Full information about signals in *each* cycle
- Analysing ILA with thousands of time-frames becomes computationally infeasible



+ Post-Silicon Faults and MCSes

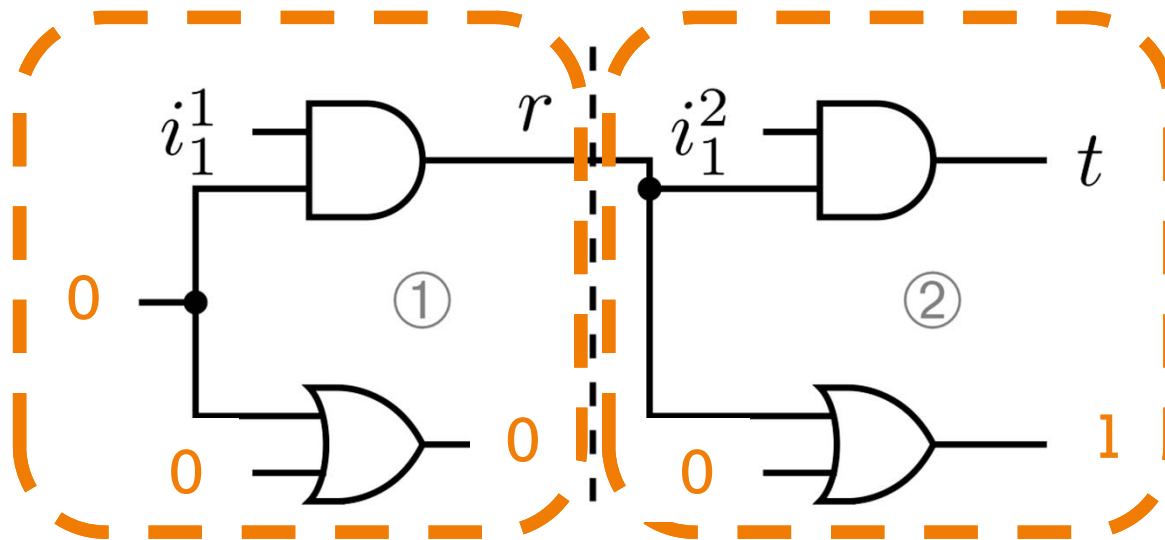
Limits of Scalability

- Analysis limited to small (contiguous) sequence of cycles
- Scalability of decision procedure determines **window size**
- **Slide window backwards in time** to cover different cycles



+ Sliding Windows

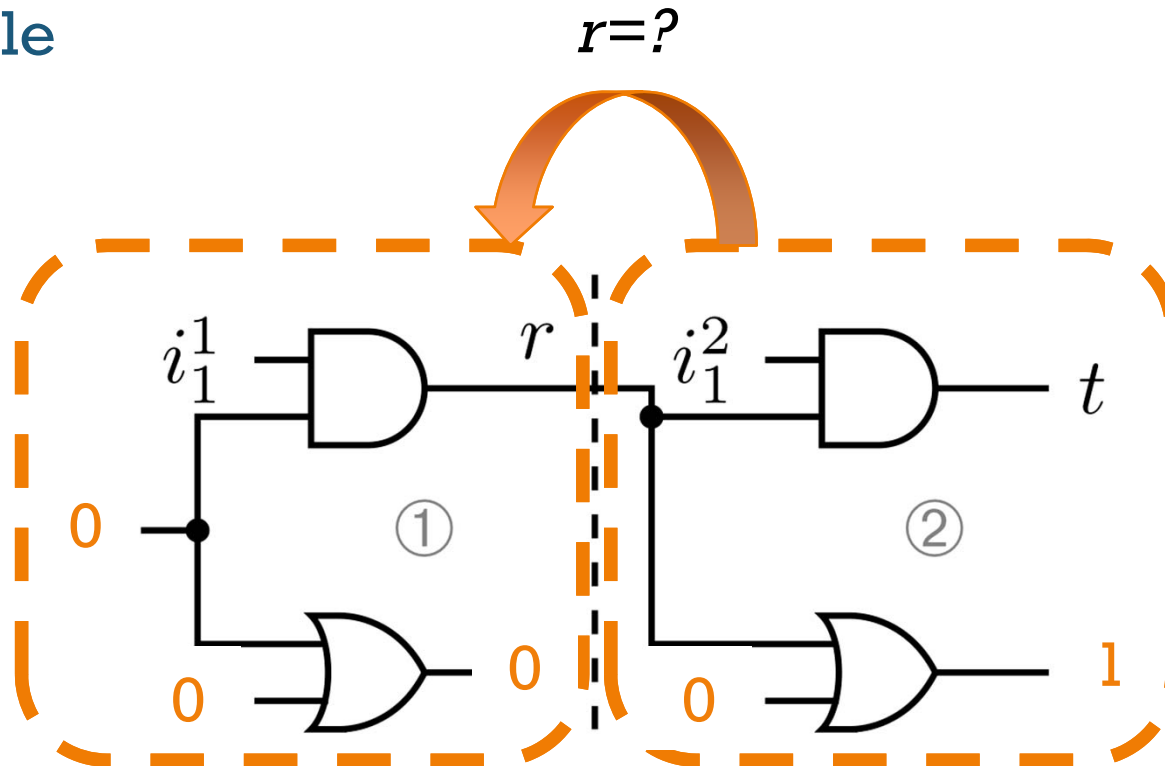
Example



- Sliding windows may fail to locate fault
- Approach is incomplete due to limited information
 - In this particular example: we don't know the value of r

+ Sliding Windows

Example

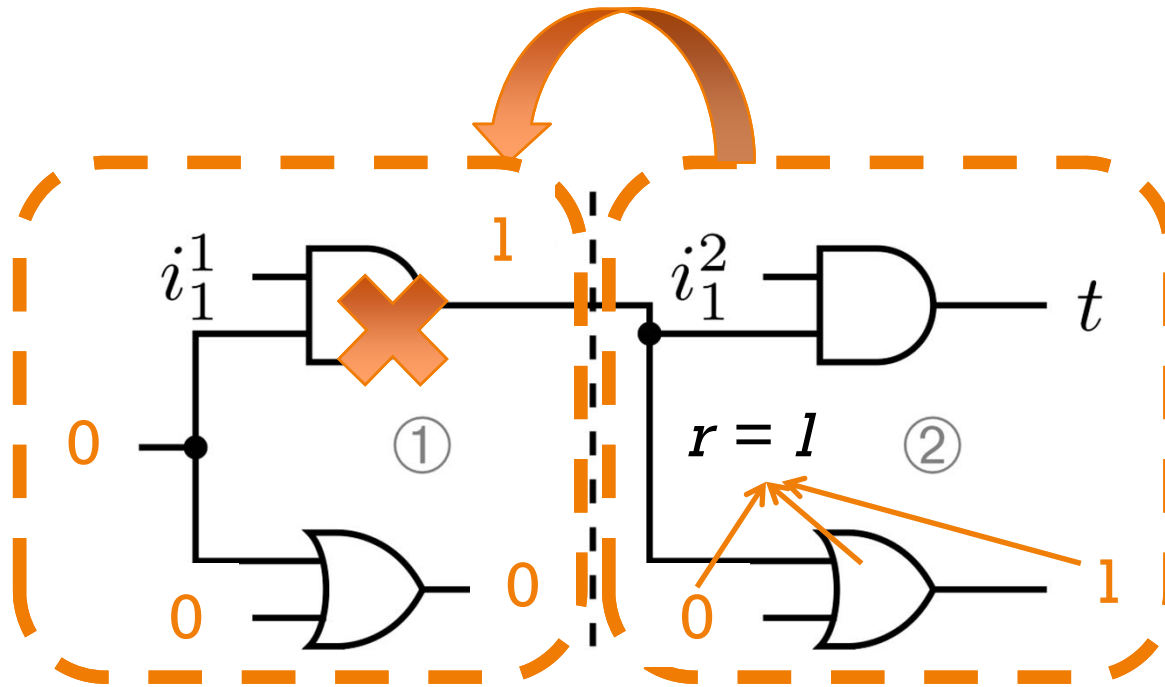


- Would like to propagate information across windows
 - At a reasonable computational cost
- Maybe we can infer the value of r in the first window?

+ Reconstructing Information

With Inferred Values

$r=l$



+ Backbones

Inferring “Fixed” Signals for Satisfiable Instances

- Backbone of a satisfiable formula:
Set of variables that have *same* value in all satisfying assignments
- Consider the satisfiable formula

$$(r \oplus t) \cdot (r + s) \cdot (r)$$

- Satisfying assignments:

r	s	t
1	0	0
1	1	0

+ Computing Backbones



Given a Boolean formula F

1. Obtain initial satisfying assignment A_0
2. For each literal p such that $A_0[p]=1$
 - variable of p is part of backbone iff $(F \cdot \bar{p})$ is *UNSAT*

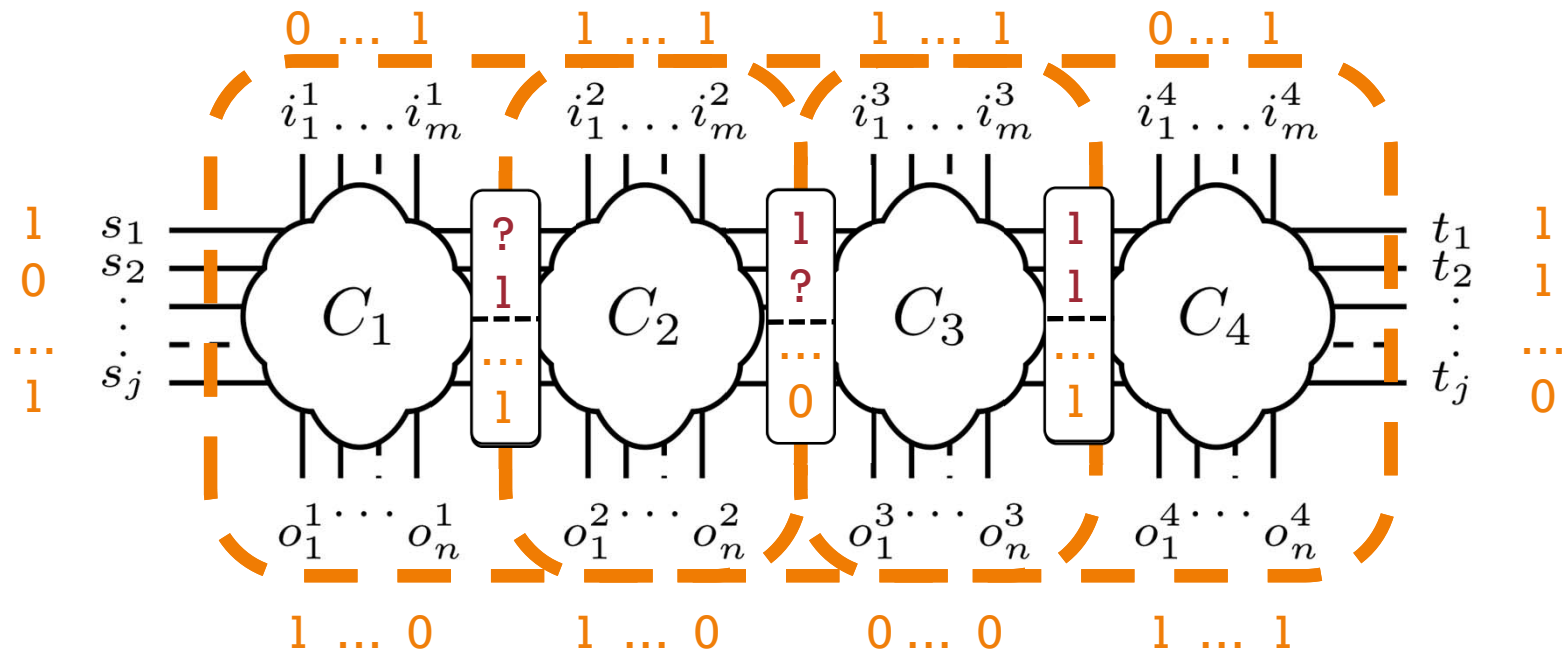
Optimization (Filtering):

1. If $(F \cdot \bar{p})$ is satisfiable, look at this satisfying assignment A_1
2. Variables differing in value in A_0 and A_1 are not in backbone

[Marques-Silva, Janota, Lynce 2010]

+ Propagating Backbones

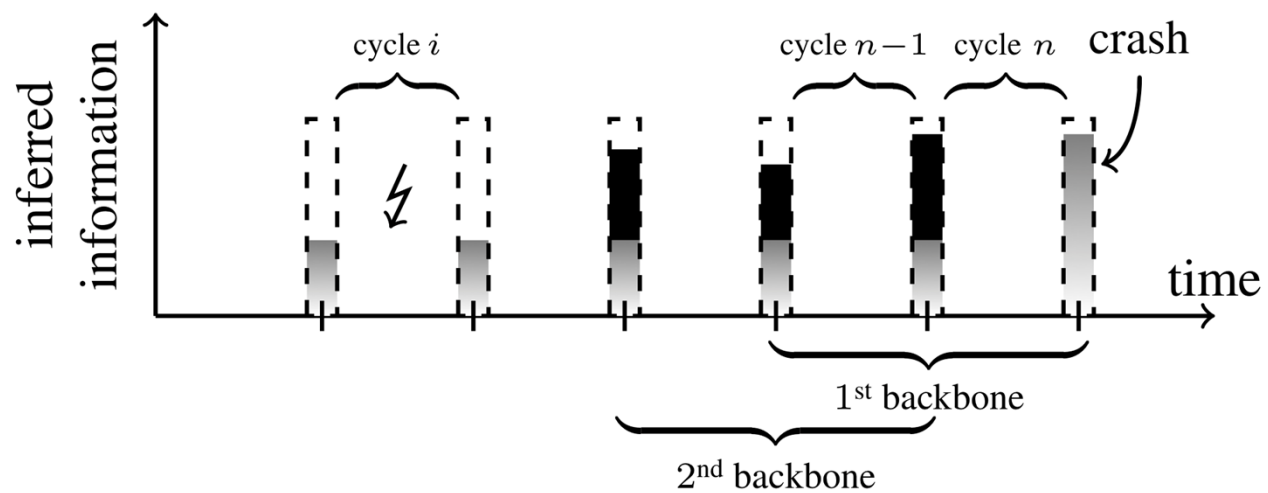
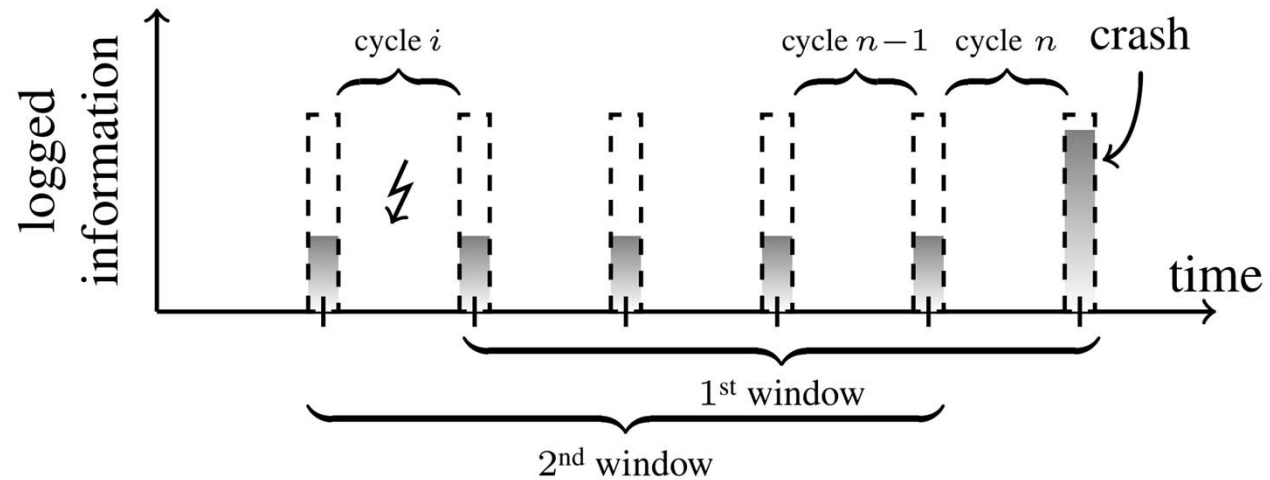
Across Sliding Windows



[Zhu, Weissenbacher, Sethi, Malik 2011]

+ Propagating Backbones

Across Sliding Windows



+ Outline



- Hardware Design/Verification Flow
- SAT Background
 - SAT Encoding of Logic Designs
 - Satisfying Assignments for Debugging
 - **Unsatisfiable Instances for Debugging**
- Pre-Silicon Debug
 - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
 - Localizing faults in manufactured prototypes
- **Outlook: Fault Localization in Software**
- Summary

+ Fault Localization in Software

Methodology

1. Observe an assertion violation
2. Unwind loops of the program, obtain symbolic representation
3. Constrain program with *expected* input/output values
4. Compute MCSes to locate faults

[Jose, Majumdar 2011]

+ Fault Localization in Software

Problem Definition

Golden Model

Specification

e.g., requirements

Assertions in Program

Consistent?

Software Implementation




Unwinding

Symbolic Representation



+ Symbolic Representation of Software

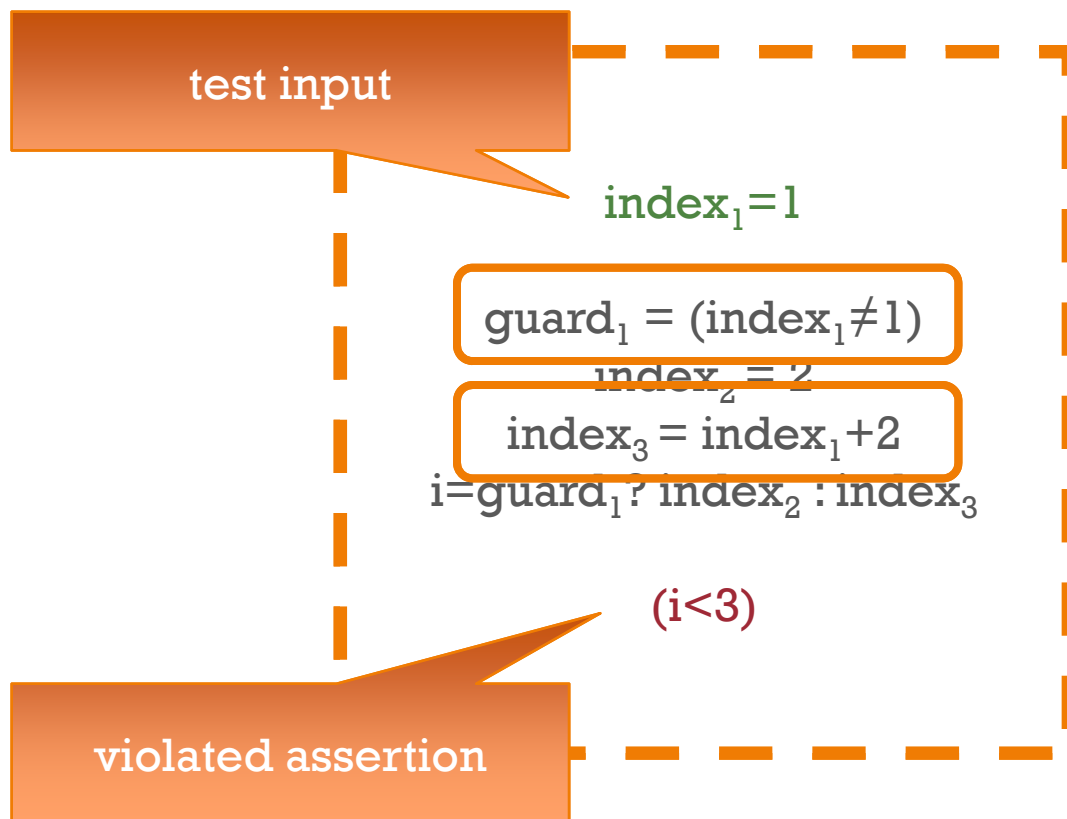


```
main.c     
  
int Array[3];  
  
int testme(int index)  
{  
    if (index != 1)  
        index = 2;  
    else  
        index = index + 2;  
  
    i = index;  
    assert (i >= 0 && i < 3); //   
    return Array[i];  
}
```

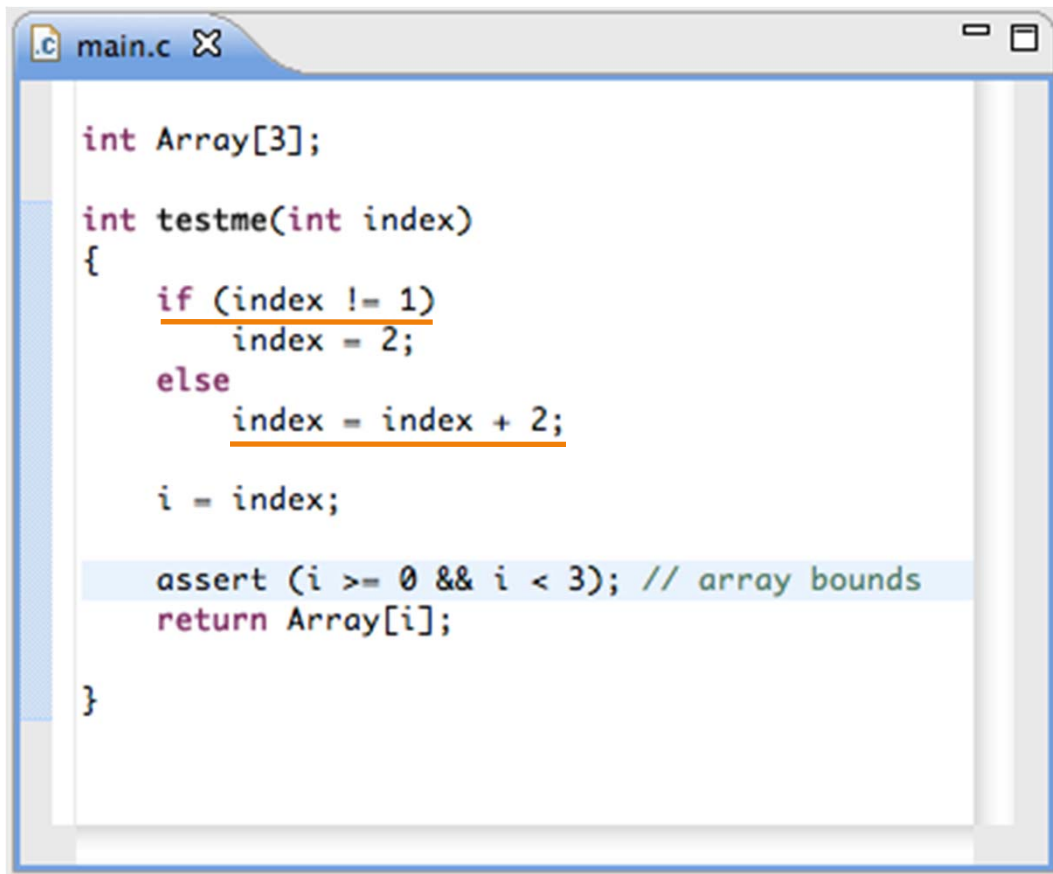
Static Single Assignment Form
[Cytron, Ferrante, Rosen,
Wegman, Zadeck 1991]




$guard_1 = (index_1 \neq 1)$
 $index_2 = 2$
 $index_3 = index_1 + 2$
 $i = guard_1 ? index_2 : index_3$

+ Computing MCSes for Program



+ MCSes Indicate Potential Errors



```
main.c     
  
int Array[3];  
  
int testme(int index)  
{  
    if (index != 1)  
        index = 2;  
    else  
        index = index + 2;  
  
    i = index;  
  
    assert (i >= 0 && i < 3); // array bounds  
    return Array[i];  
}
```

+ Outline



- Hardware Design/Verification Flow
- SAT Background
 - SAT Encoding of Logic Designs
 - Satisfying Assignments for Debugging
 - **Unsatisfiable Instances for Debugging**
- Pre-Silicon Debug
 - Localizing faults in Register-Transfer-Level (RTL) designs
- Post-Silicon Validation
 - Localizing faults in manufactured prototypes
- Outlook: Fault Localization in Software
- **Summary**

+ Summary



- Locating faults is tedious
- Minimal Correction Sets enable fault localization in
 - logic design
 - manufactured chip prototypes
 - software
- Backbones enable partial recovery of information



References (for Hardware)



- [Abramovici, Breuer, Friedman 1990]
Digital Systems Testing and Testable Design
Computer Science Press
- [Smith, Veneris, Ali, Viglas 2004]
Fault Diagnosis and Logic Debugging Using Boolean Satisfiability
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
- [Josephson 2006]
The Good, the Bad, and the Ugly of Silicon Debug
Design Automation Conference
- [Liffiton, Sakallah 2009]
Generalizing Core-Guided MAX-SAT
Theory and Applications of Satisfiability Testing
- [Chen, Safarpour, Marques-Silva, Veneris 2009]
Automated Design Debugging with Maximum Satisfiability
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
- [Marques-Silva, Janota, Lynce 2010]
On Computing Backbones of Propositional Theories
European Conference on Artificial Intelligence
- [Zhu, Weissenbacher, Sethi, Malik 2011]
Post-Silicon Fault Localisation Using Maximum Satisfiability and Backbones
Formal Methods in Computer-Aided Design, 2011

+ References (for Software)

- [Clarke, Kroening, Lerda 2004]
A Tool for Checking ANSI-C Programs
Tools and Algorithms for the Construction and Analysis of Systems
- [Groce, Chaki, Kroening, Strichman 2006]
Error Explanation with Distance Metrics
International Journal on Software Tools for Technology Transfer
- [Jose, Majumdar 2011] (Tool paper)
BugAssist: Assisting Fault Localization in ANSI-C Programs
Computer Aided Verification
- [Jose, Majumdar 2011]
Clause Cue Clauses: Error Localization Using Maximum Satisfiability
Programming Languages Design and Implementation

