# Formal Equivalence Checking

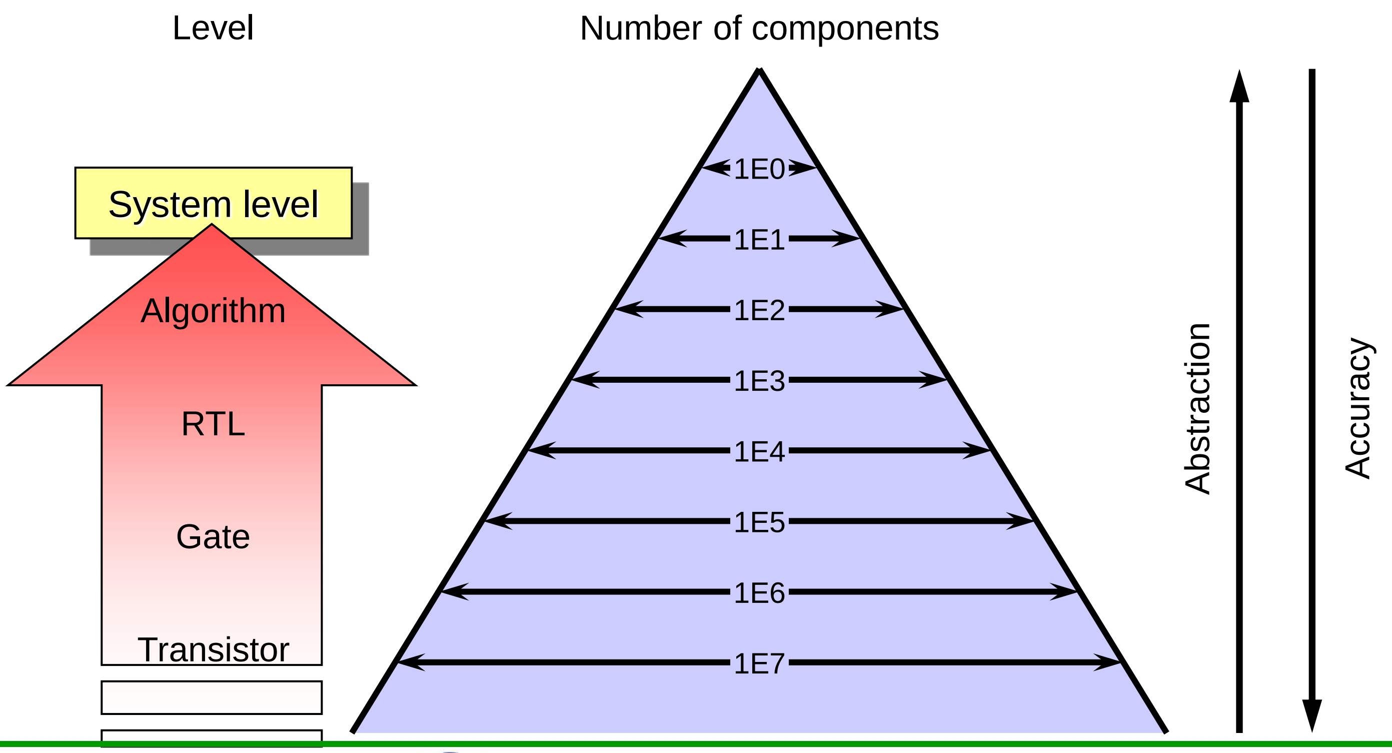
Virendra Singh
Associate Professor
Computer Architecture and Dependable Systems Lab
Dept. of Electrical Engineering
Indian Institute of Technology Bombay, Mumbai
viren@ee.iitb.ac.in

EE 709: Testing & Verification of VLSI Circuits

Lecture – 6 (Jan 17, 2012)

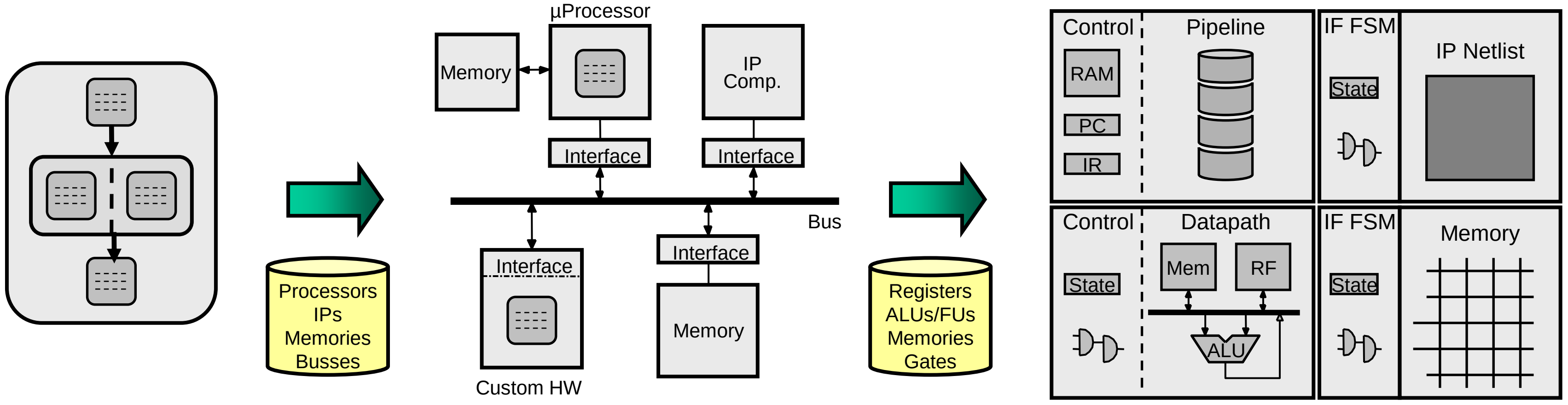
## SoC Verification

- System-on-Chip (SOC) design
- Increase of design complexity
- Move to higher levels of abstraction



# System-on-Chip (SoC) design

- Specification to architecture and down to implementation
- Behavior (functional) to structure
  - System level: system specification to system architecture
  - RT/IS level: component behavior to component micro-architecture

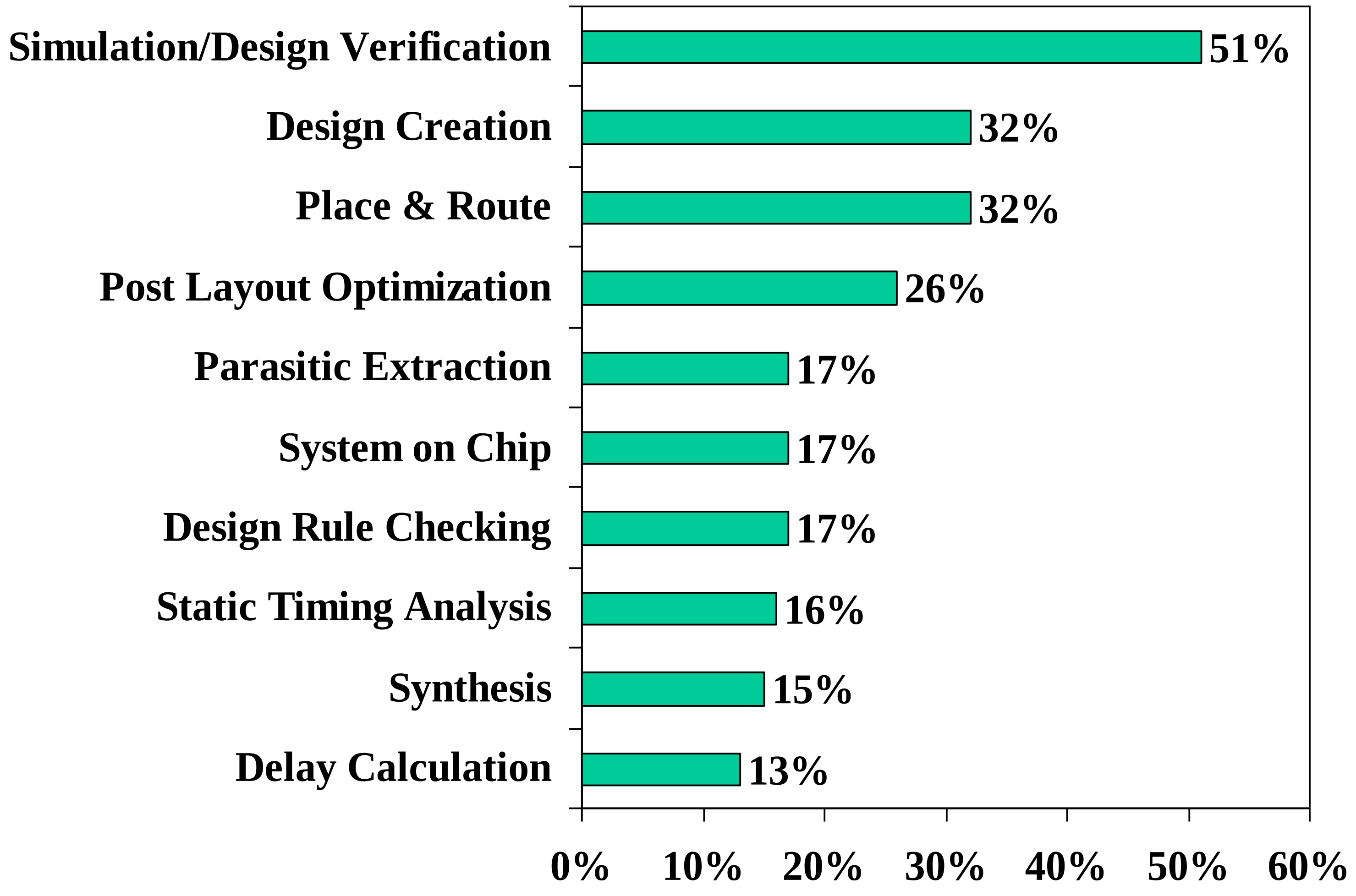


Specification
+ constraints

System architecture
+ estimates

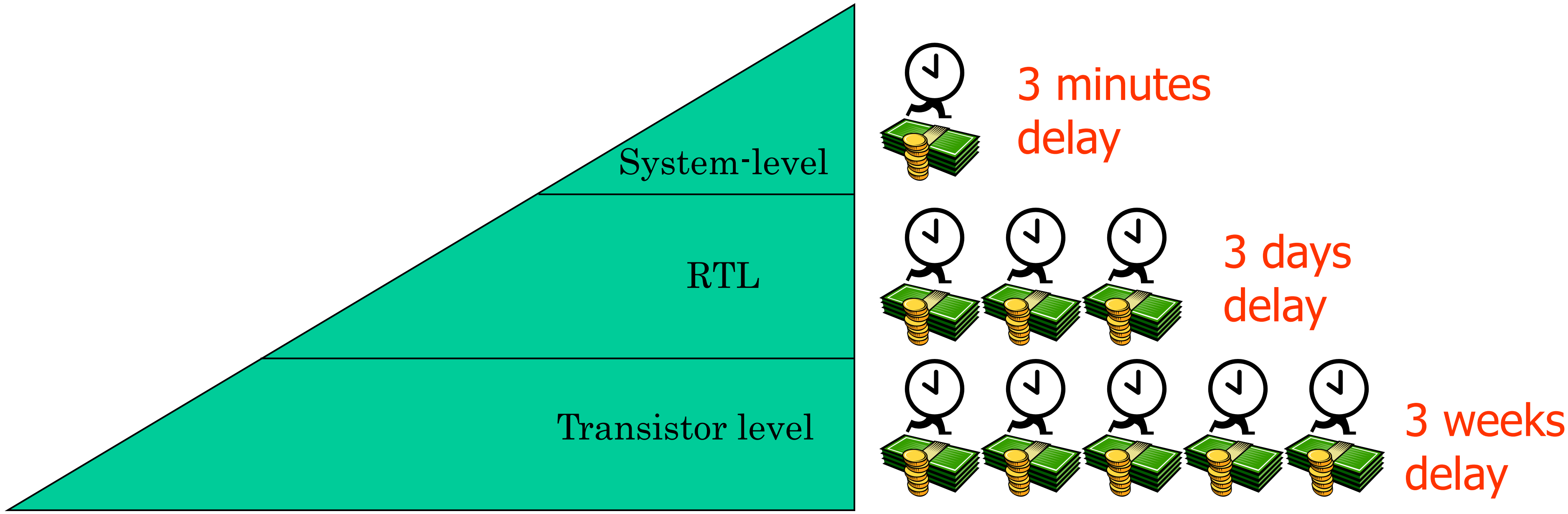
RTL/IS Implementation
+ results

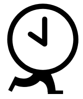
# Verification challenge



Bottlenecks in Design Cycles:
Survey of 545 engineers by EETIMES 2000

# System-level design & verification



 Bugs fix time

Cost due to the delay/late time-to-market revenue loss

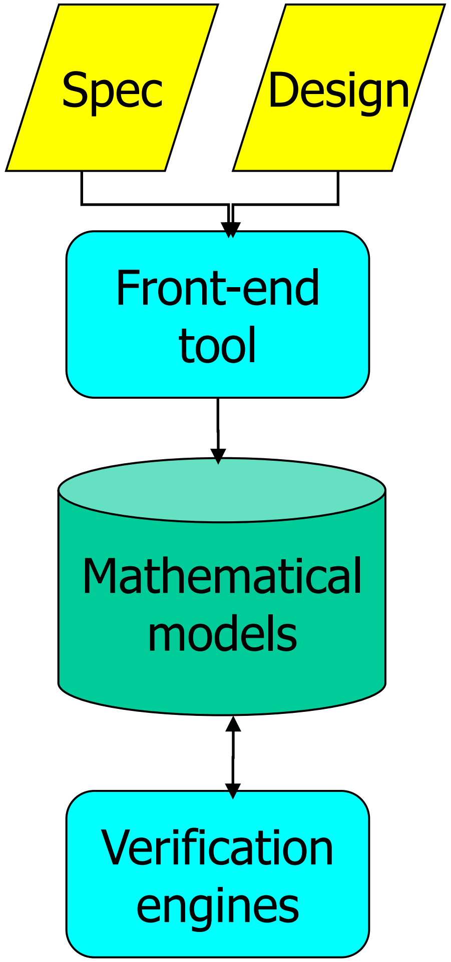
Remove as many bugs as possible in the earlier stages
Do not introduce new design errors when refining designs

$\Downarrow$

Formal verification in system-level designs:
Property checking and equivalence checking

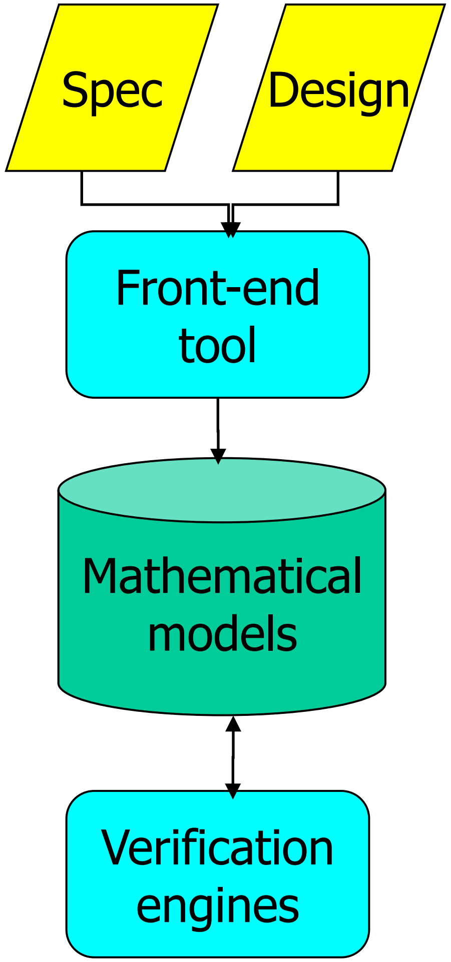
# Formal verification

- "Prove" the correctness of designs
  - Both design and spec must be represented with mathematical models
  - Mathematical reasoning
  - Equivalent to "all cases" simulations
- Possible mathematical models
  - Boolean function (Propositional logic)
    - How to represent and manipulate on computers
  - First-order logic
    - Need to represent "high level" designs
  - Higher-order logic
    - Theorem proving = Interactive method
- Front-end is also very important
  - Often, it determines the total performance of the tools



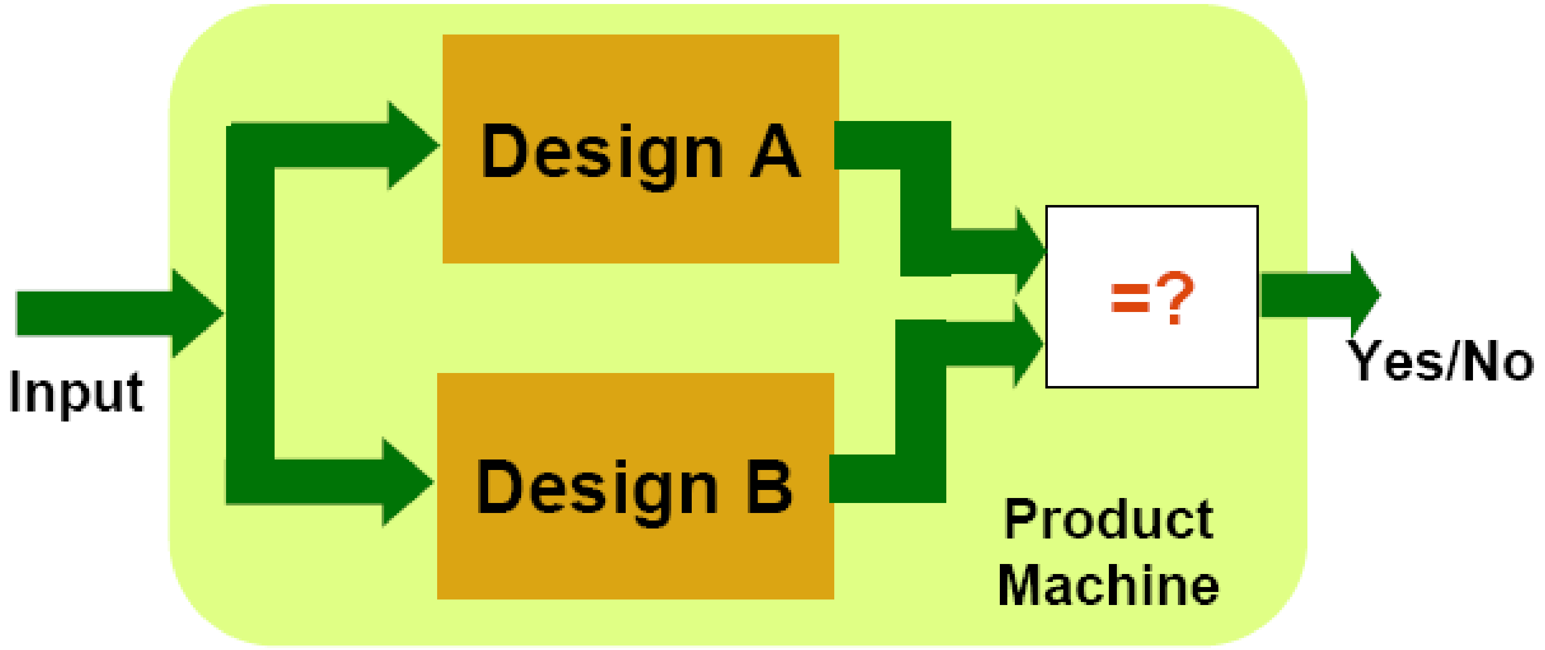
## Backgrounds technology in formal verification

- Methods for reasoning about mathematical models
  - Boolean function (Propositional logic)
    - SAT (Satisfiability checker)
    - BDD (Binary Decision Diagrams)
  - First-order logic
    - Logic of uninterpreted functions with equality
  - Higher-order logic
    - Theorem proving = Interactive method



# Formal Equivalence Checking

**Given two designs, prove that for all possible input stimuli their corresponding outputs are equivalent**



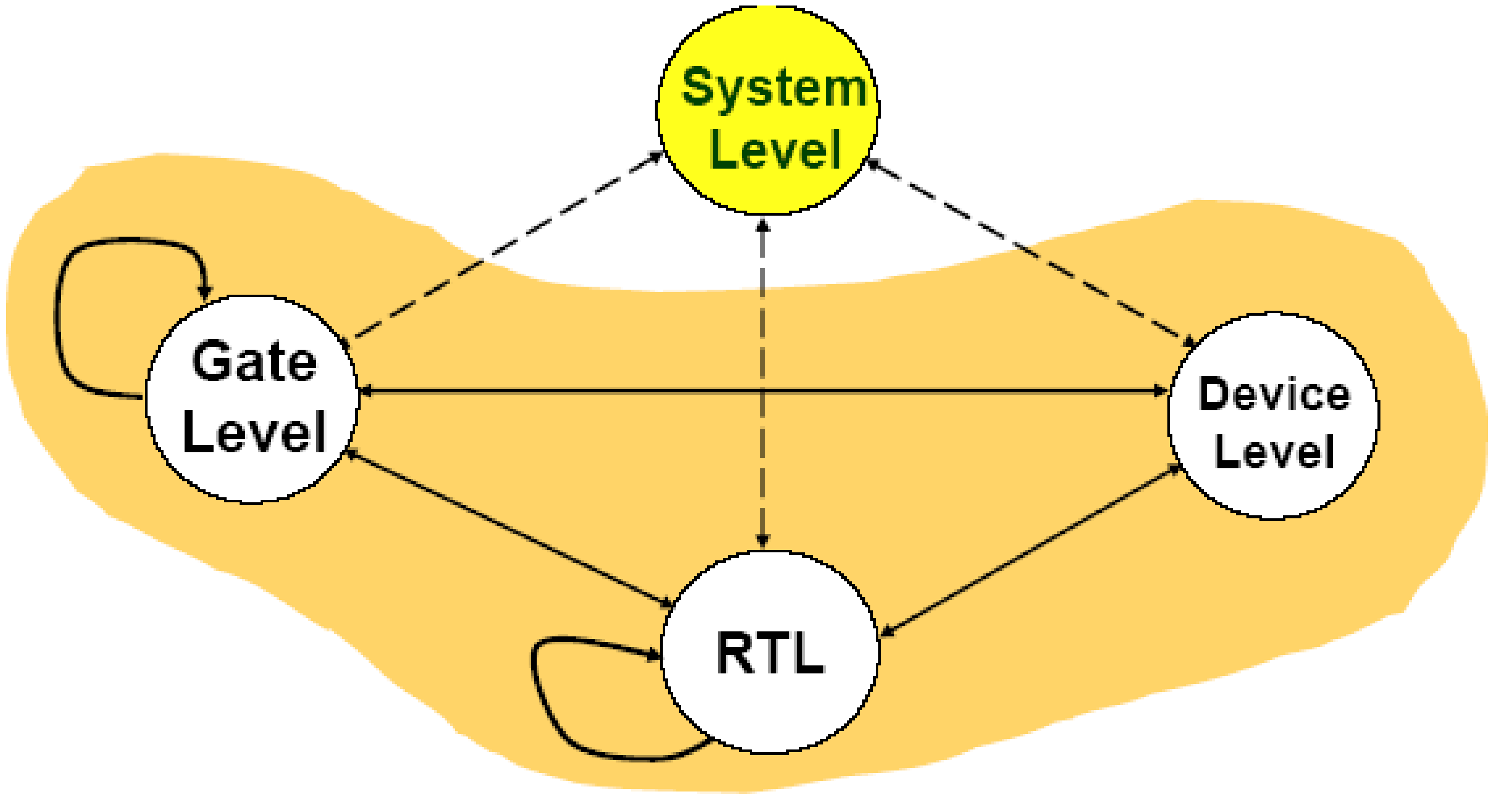
## Formal Equivalence Checking

- Scalable
  - Full chip verification possible
  - e.g. Designs of up to several million gates verified in a few hours or minutes with CEC
  - Hierarchical verification deployed
- Automatic
  - CEC: Nearly full automation possible
- High/Full Coverage

***CEC Currently most practical and pervasive formal verification technology used in industry***

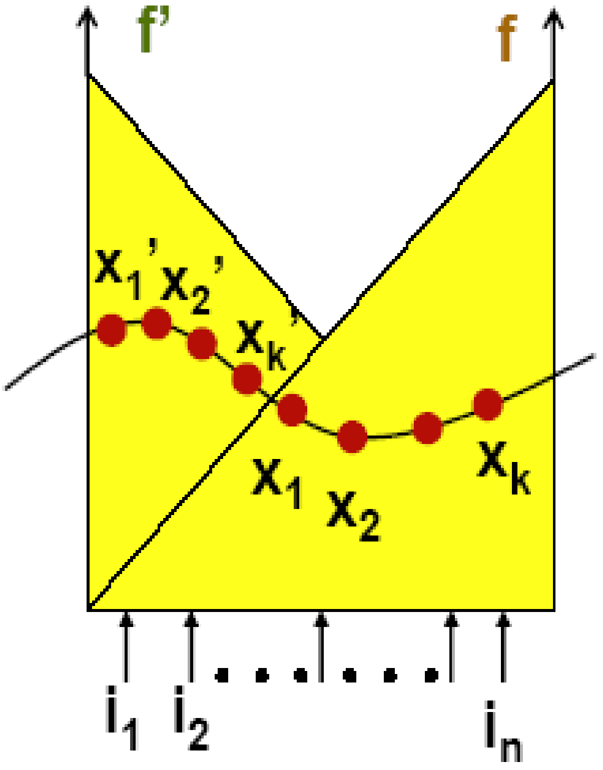
# Formal Equivalence Checking

- Equivalence checking can be applied at or across various levels



## CEC in Practice

Key observation: The circuit being verified usually have a number of internal equivalent functions



To prove $f(i_1, i_2, \ldots i_n) = f'(i_1, i_2, \ldots i_n)$

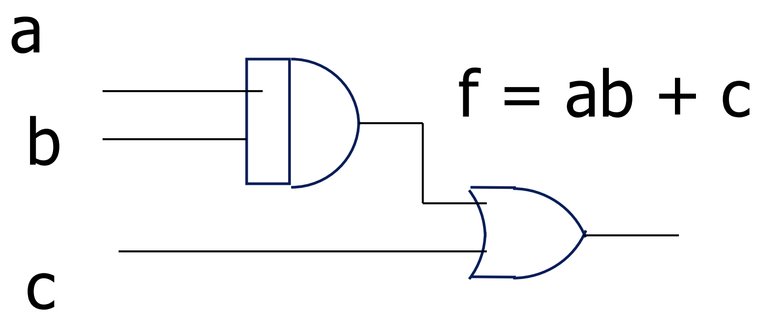
Check

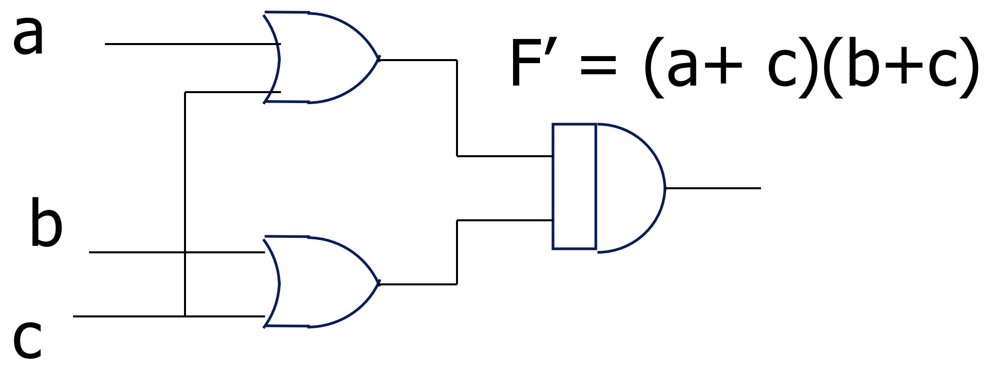
$$x_1(i_1, i_2, \ldots i_n) = x_1'(i_1, i_2, \ldots i_n)$$
$$x_2(i_1, i_2, \ldots i_n) = x_2'(i_1, i_2, \ldots i_n)$$
$$\vdots$$
$$x_k(i_1, i_2, \ldots i_n) = x_k'(i_1, i_2, \ldots i_n)$$

$$f(x_1, x_2, \ldots x_k) = f'(x_1', x_2', \ldots x_k')$$

# Formal Equivalence Checking

## Canonical Forms





| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

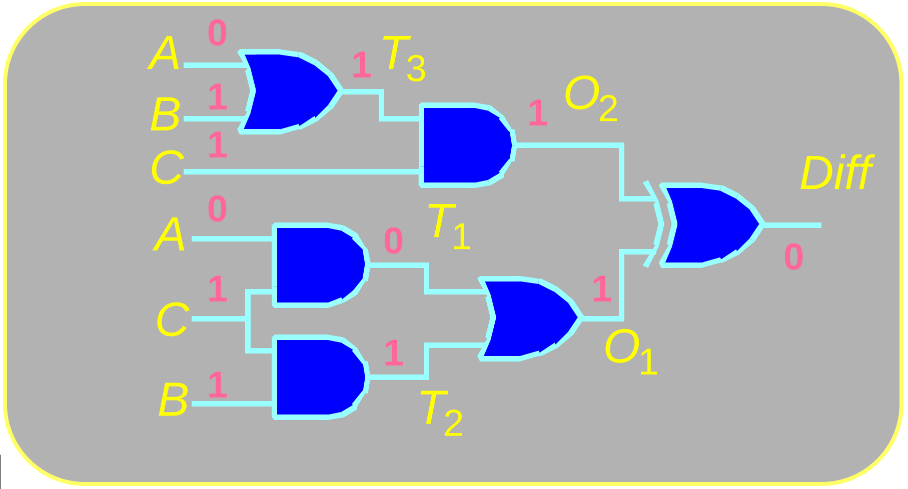
# Formal Equivalence Checking

## Complexity

- Efficiency of the conversion to canonical form
- Memory requirement
- Efficiency of the comparison of two representation of the canonical form
- Efficiency to generate the counter example in case of a miscompare

# Formal Equivalence Checking

- Satisfiability Formulation
  - Search for input assignment giving different outputs
- Branch & Bound
  - Assign input(s)
  - Propagate forced values
  - Backtrack when cannot succeed



- Challenge
  - Must prove all assignments fail
    - Co-NP complete problem
  - Typically explore significant fraction of inputs
  - Exponential time complexity

# Formal Equivalence Checking

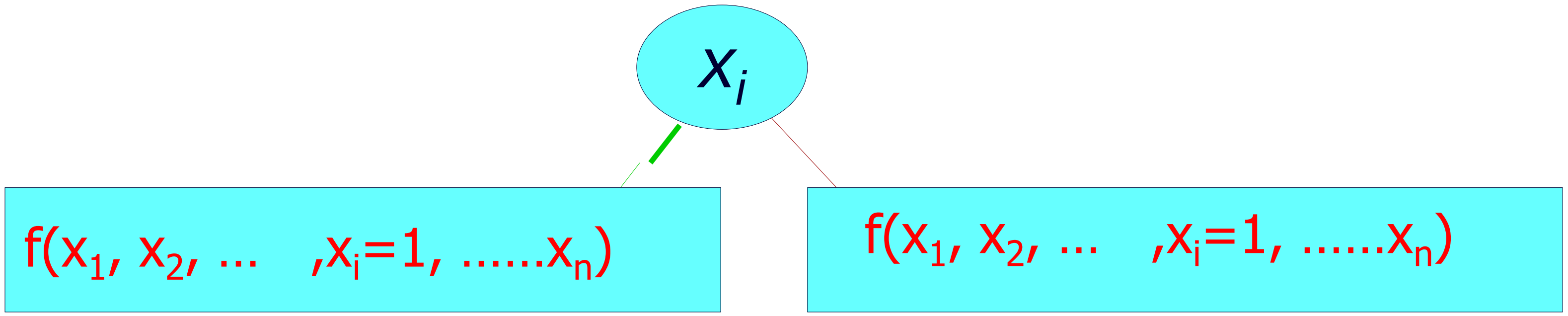
- Canonical form representation is only suitable
- DNF and CNF are not suitable
- BDD is most popular canonical form
  - graphical representation of boolean function

# Formal Equivalence Checking

- BDD is canonical form of representation
- Shanon's expansion theorem
- $f(x_1, x_2, \ldots . x_i, \ldots \ldots x_n) =$

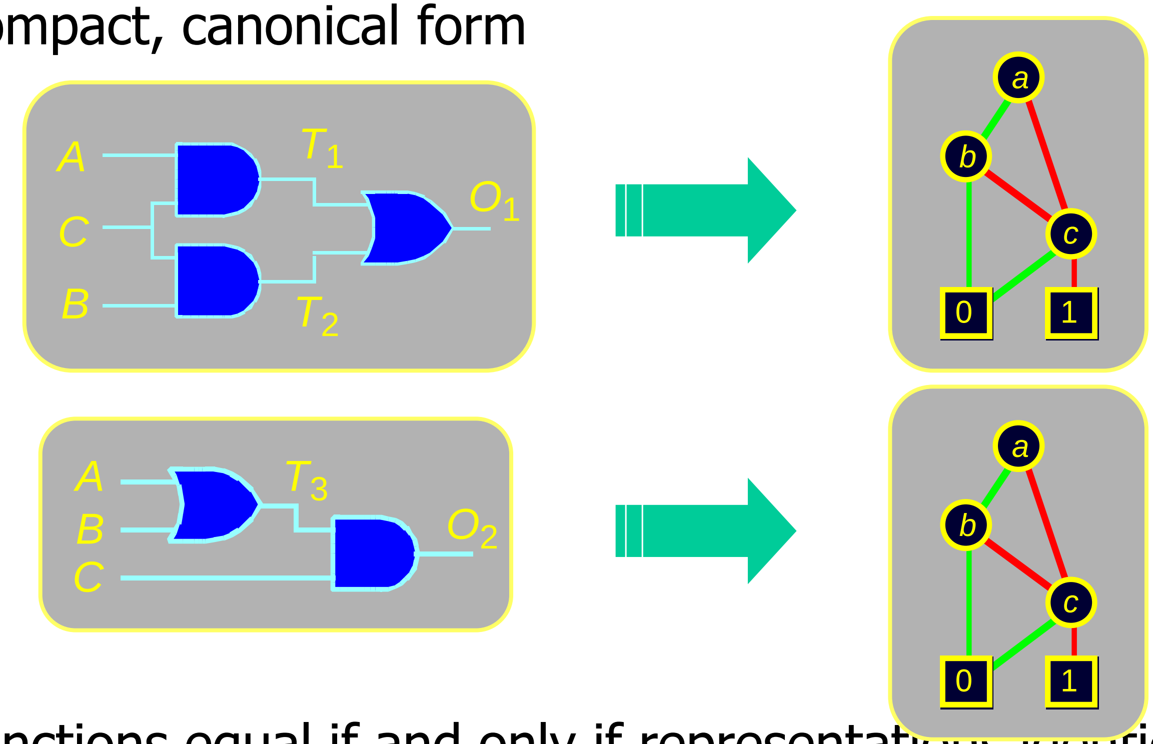
  $x_i . f(x_1, x_2, \ldots \quad , x_i=1, \ldots \ldots x_n) +$

  $x_i' . f(x_1, x_2, \ldots \quad , x_i=0, \ldots \ldots x_n)$


$x_i$
$f(x_1, x_2, \ldots \quad , x_i=1, \ldots \ldots x_n)$
$f(x_1, x_2, \ldots \quad , x_i=1, \ldots \ldots x_n)$

# Binary Decision Diagram

- Generate Complete Representation of Circuit Function
  - Compact, canonical form



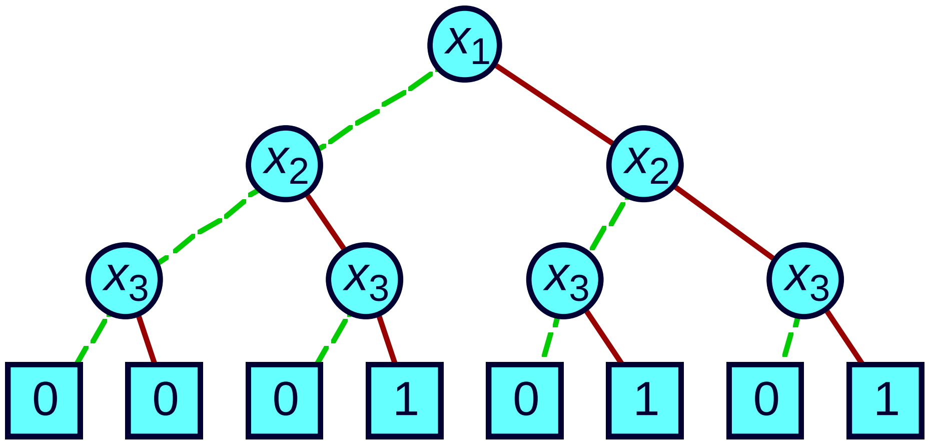
➢ Functions equal if and only if representations identical
➢ Never enumerate explicit function values
➢ Exploit structure & regularity of circuit functions

## Decision Structures

**Truth Table**

| $x_1$ | $x_2$ | $x_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

**Decision Tree**

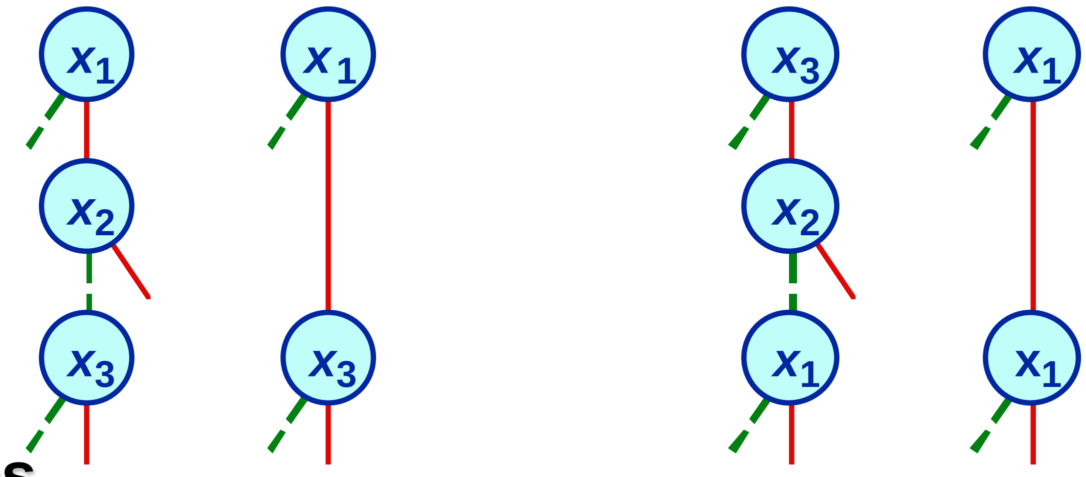


- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

# Variable Ordering

- Assign arbitrary total ordering to variables
  - e.g., $x_1 < x_2 < x_3$
- Variables must appear in ascending order along all paths

**OK** **Not OK**

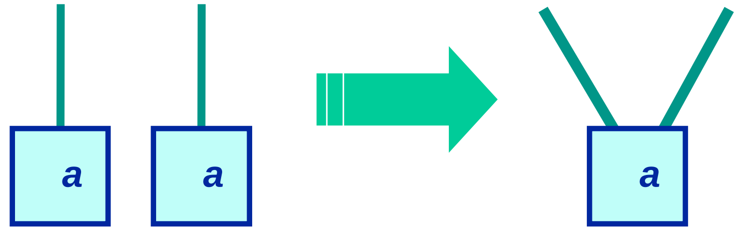


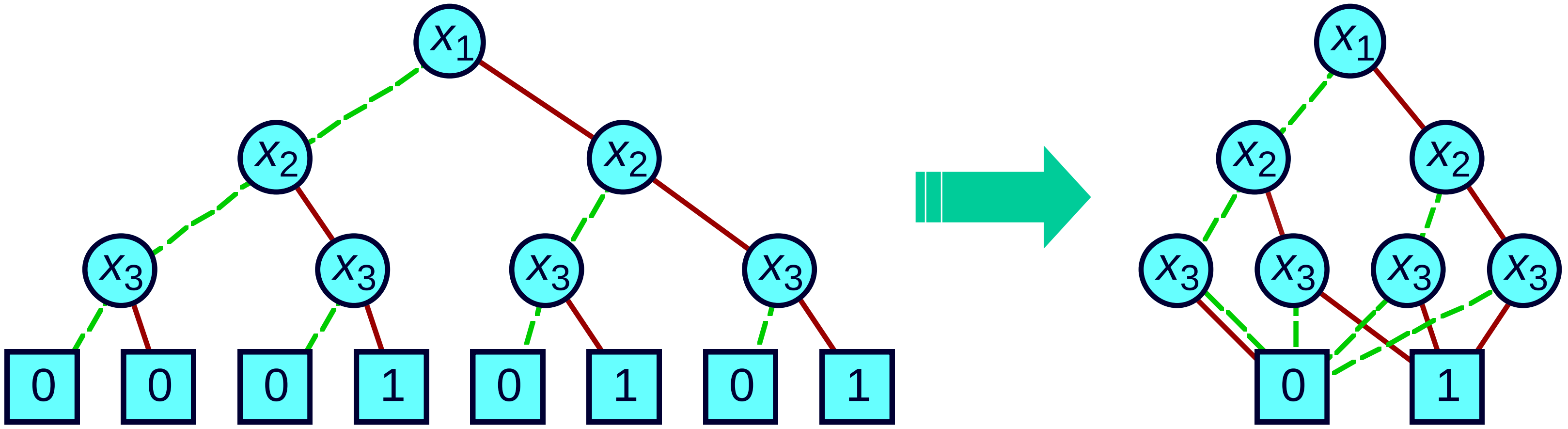
**Properties**

- No conflicting variable assignments along path
- Simplifies manipulation

# Reduction Rule #1

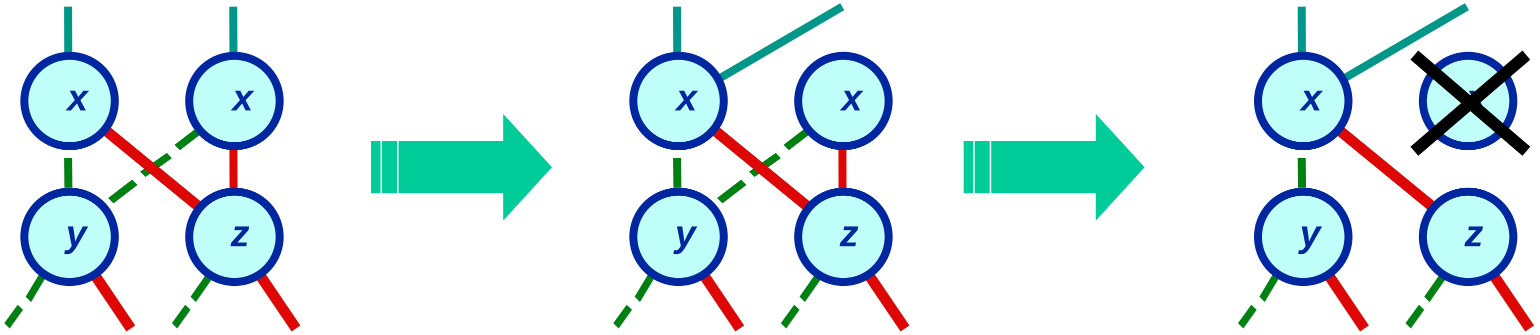
Merge equivalent leaves

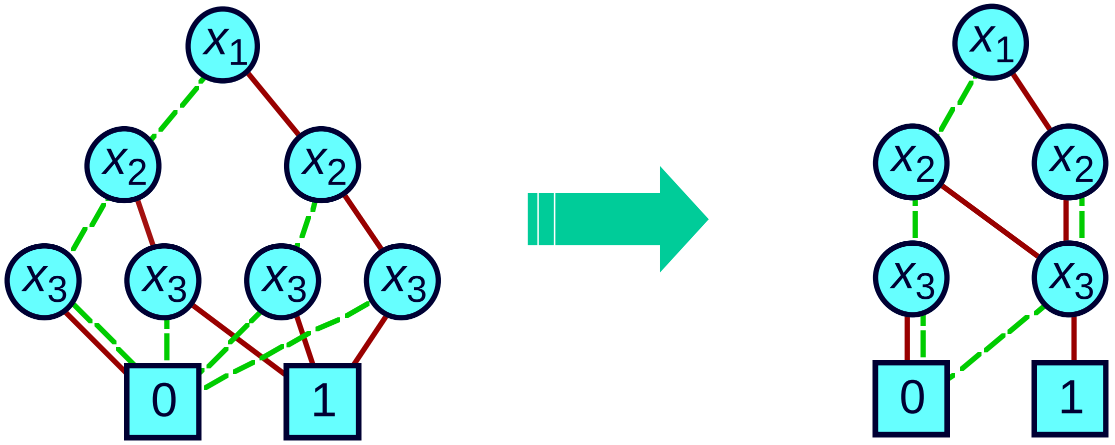




# Reduction Rule #2

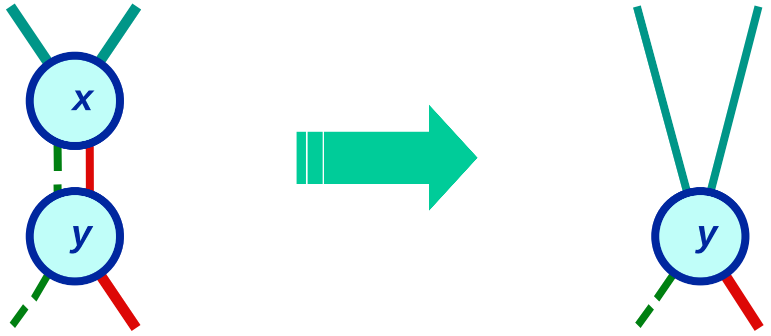
Merge isomorphic nodes

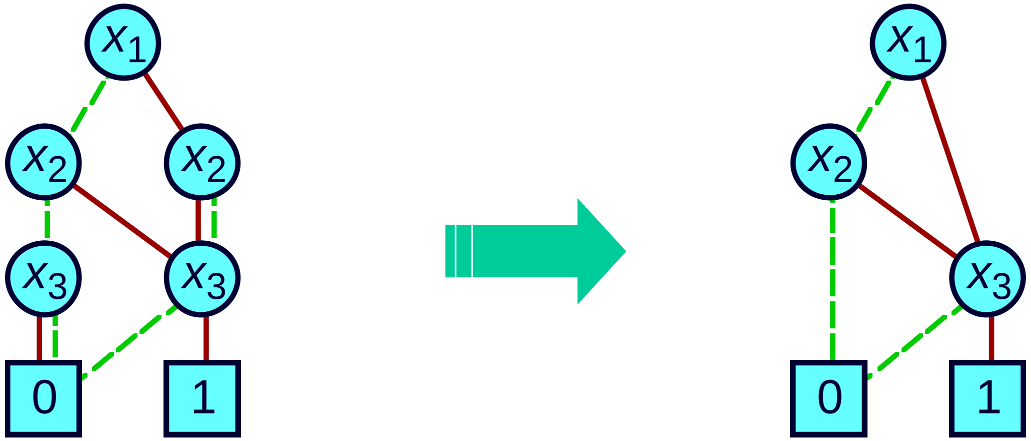




## Reduction Rule #3

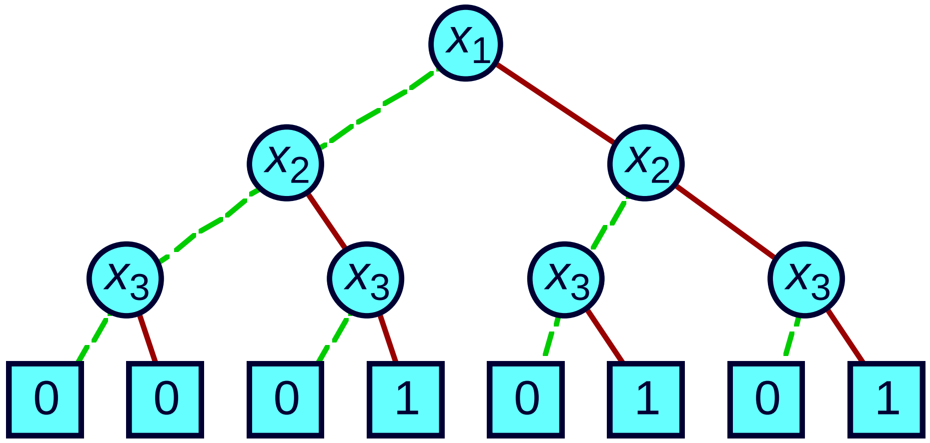
**Eliminate Redundant Tests**



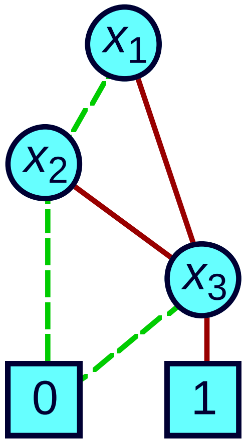


# Example OBDD

**Initial Graph**



**Reduced Graph**

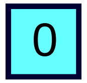


$(x_1+x_2)\cdot x_3$

- Canonical representation of Boolean function
  - ❖ For given variable ordering
  - ➢ Two functions equivalent if and only if graphs isomorphic
    - o Can be tested in linear time
  - ➢ Desirable property: *simplest form is canonical*.

# Example Functions

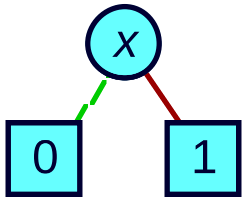
## Constants



Unique unsatisfiable function

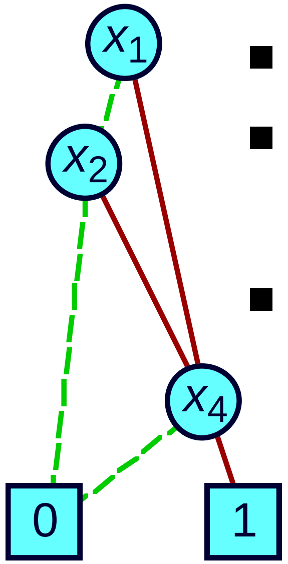
Unique tautology

## Variable



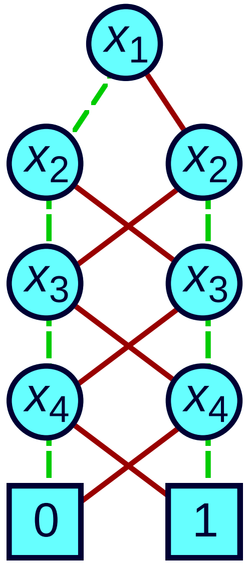
Treat variable as function

## Typical Function



- $(x_1 \lor x_2) \land x_4$
- No vertex labeled $x_3$
  - independent of $x_3$
- Many subgraphs shared

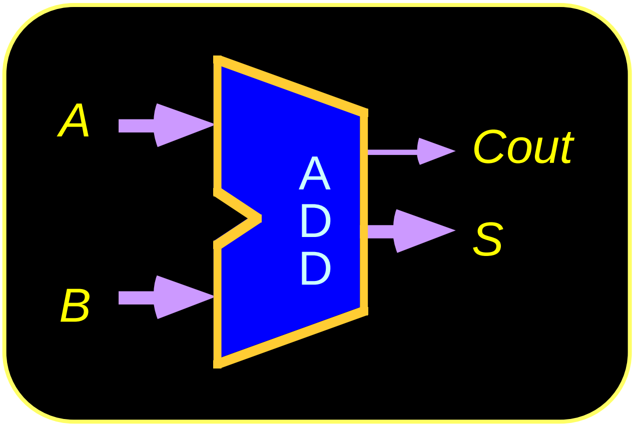
## Odd Parity



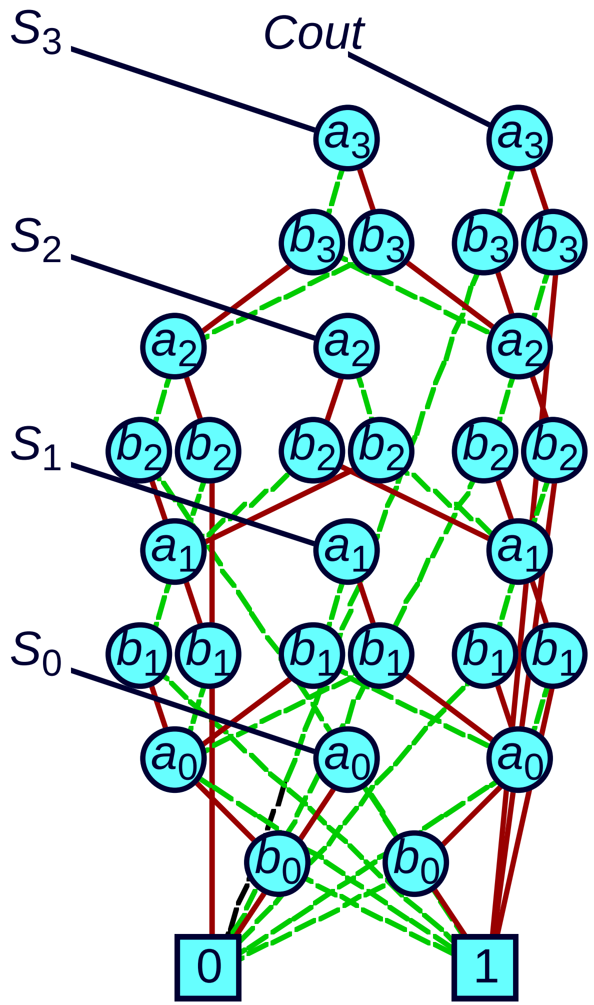
Linear representation

# Representing Circuit Functions

- Functions
  - All outputs of 4-bit adder
  - Functions of data inputs



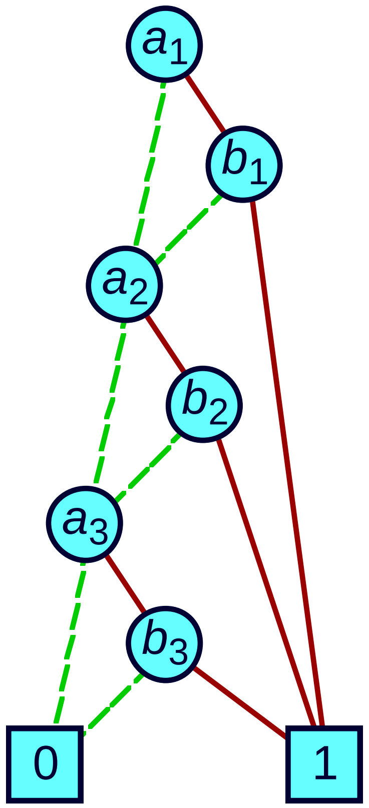
- Shared Representation
  - Graph with multiple roots
  - 31 nodes for 4-bit adder
  - 571 nodes for 64-bit adder
  - ☒ *Linear growth*



# Effect of Variable Ordering

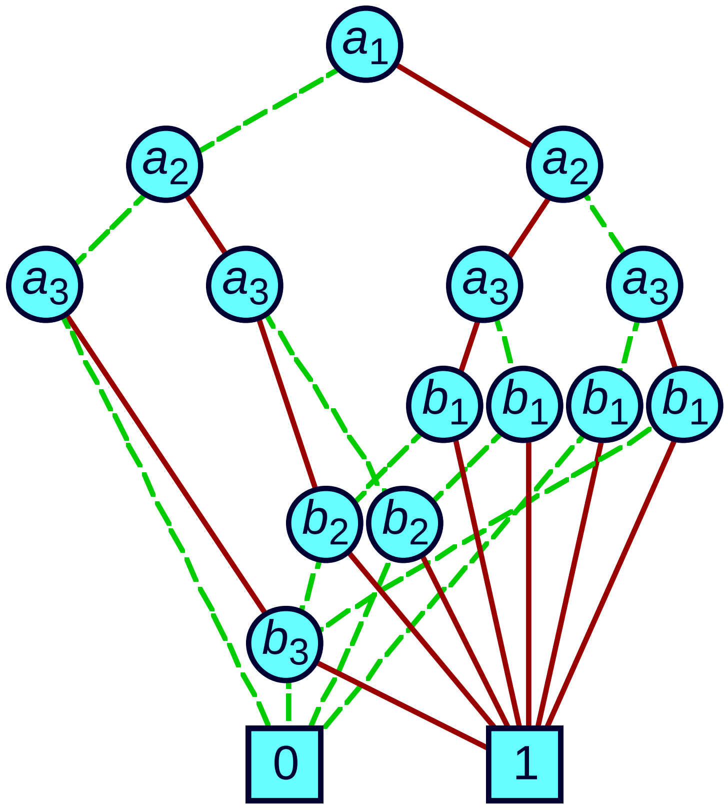
$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$$

**Good Ordering**



**Linear Growth**

**Bad Ordering**



**Exponential Growth**

# Selecting Good Variable Ordering

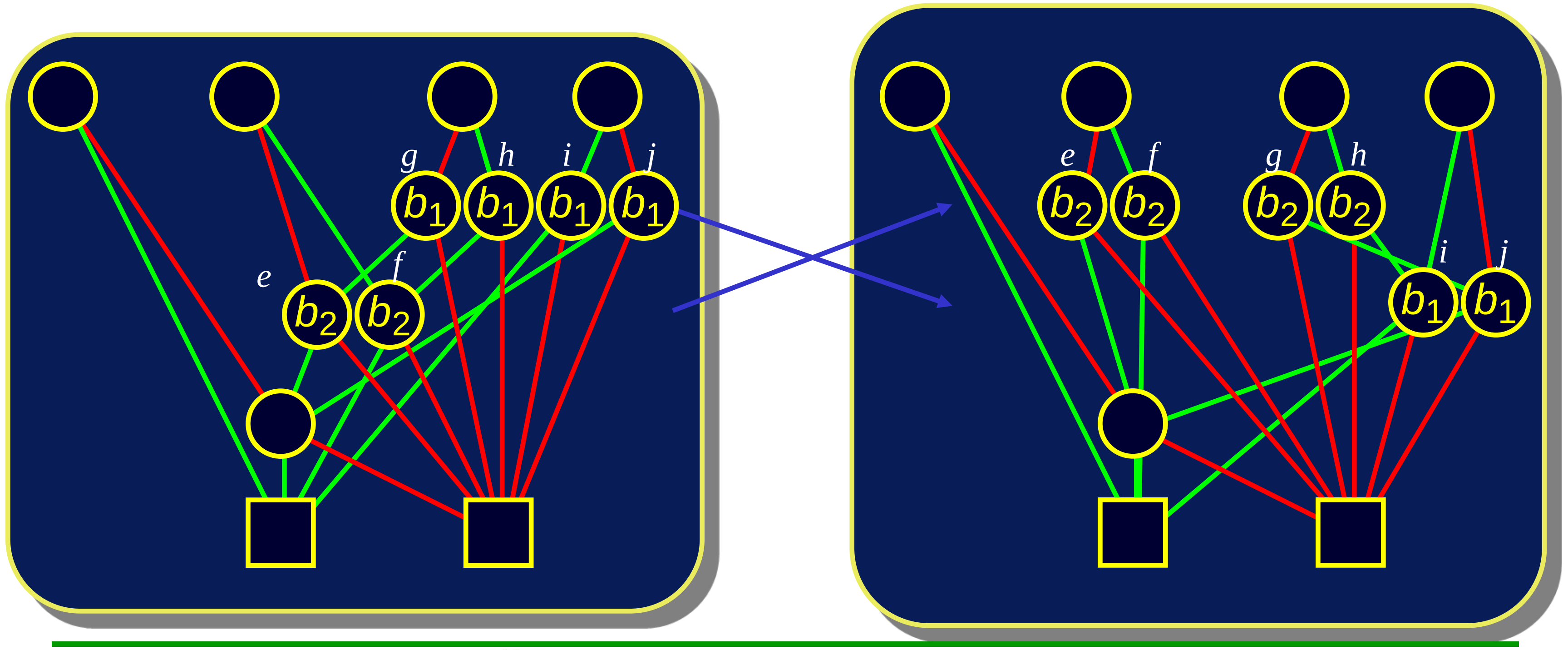
- Intractable Problem
  - Even when problem represented as OBDD
    - i.e., to find optimum improvement to current ordering
- Application-Based Heuristics
  - Exploit characteristics of application
  - e.g., Ordering for functions of combinational circuit
    - Traverse circuit graph depth-first from outputs to inputs
    - Assign variables to primary inputs in order encountered

## Selecting Good Variable Ordering

- Static Ordering
  - Fan In Heuristic
  - Weight Heuristic
- Dynamic Ordering
  - Variable Swap
  - Window Permutation
  - Sifting

# Swapping Adjacent Variables

- Localized Effect
  - Add / delete / alter only nodes labeled by swapping variables
  - Do not change any incoming pointers

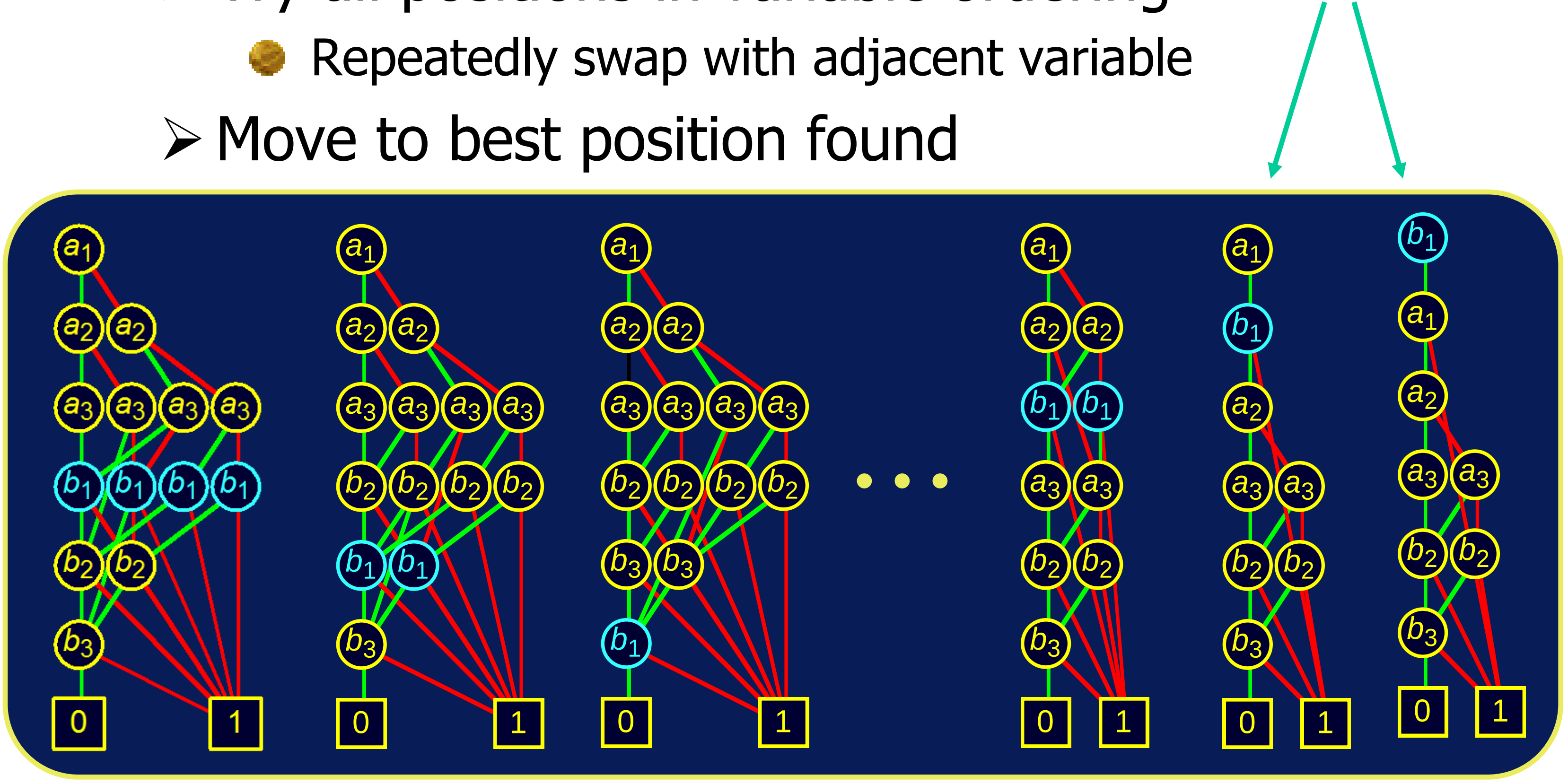


# Dynamic Variable Reordering

- Richard Rudell, Synopsys
- Periodically Attempt to Improve Ordering for All BDDs
  - Part of garbage collection
  - Move each variable through ordering to find its best location
- Has Proved Very Successful
  - Time consuming but effective
  - Especially for sequential circuit analysis

# Dynamic Reordering By Sifting

- Choose candidate variable
- Try all positions in variable ordering
  - Repeatedly swap with adjacent variable
- Move to best position found



# ROBDD sizes & variable ordering

- Bad News 💣
  - Finding optimal variable ordering NP-Hard
  - Some functions have exponential BDD size for all orders *e.g.* multiplier
- Good News ☺
  - Many functions/tasks have reasonable size ROBDDs
  - Algorithms remain practical up to 500,000 node OBDDs
  - Heuristic ordering methods generally satisfactory
- What works in Practice ☜
  - Application-specific heuristics e.g. DFS-based ordering for combinational circuits
  - Dynamic ordering based on variable sifting *(R. Rudell)*

**Thank you**