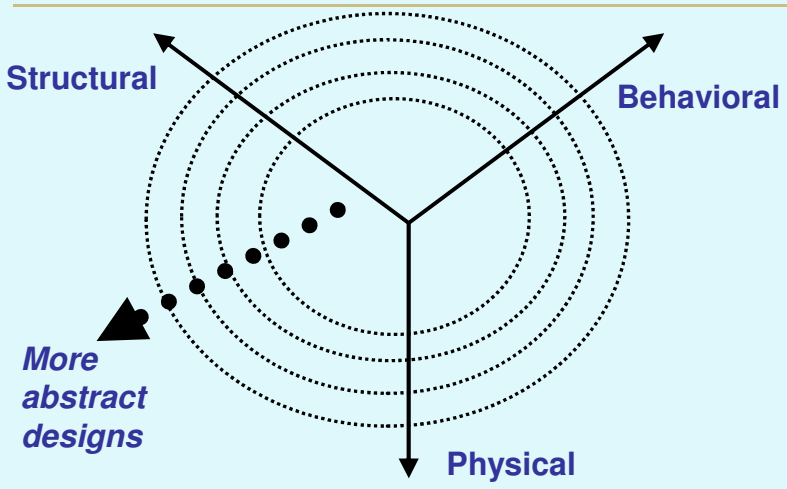
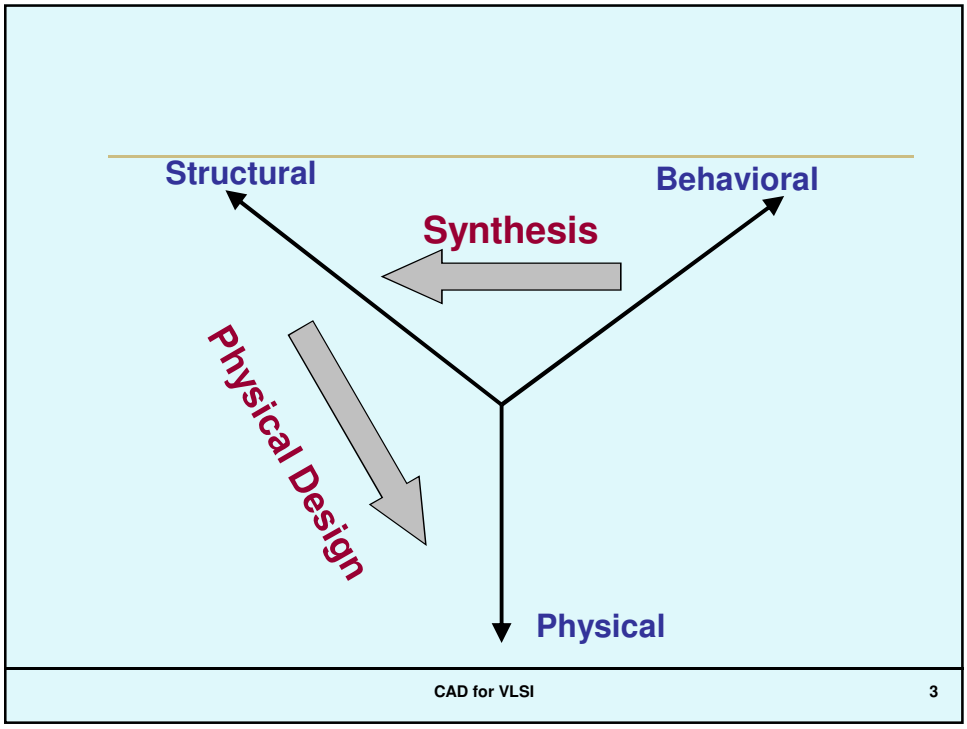
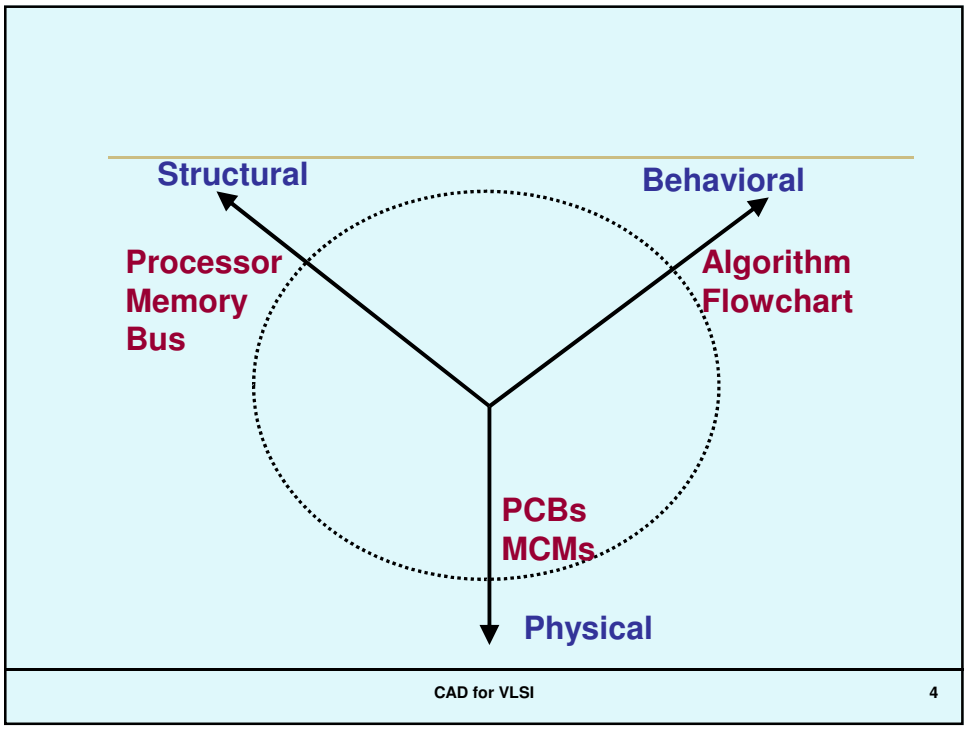
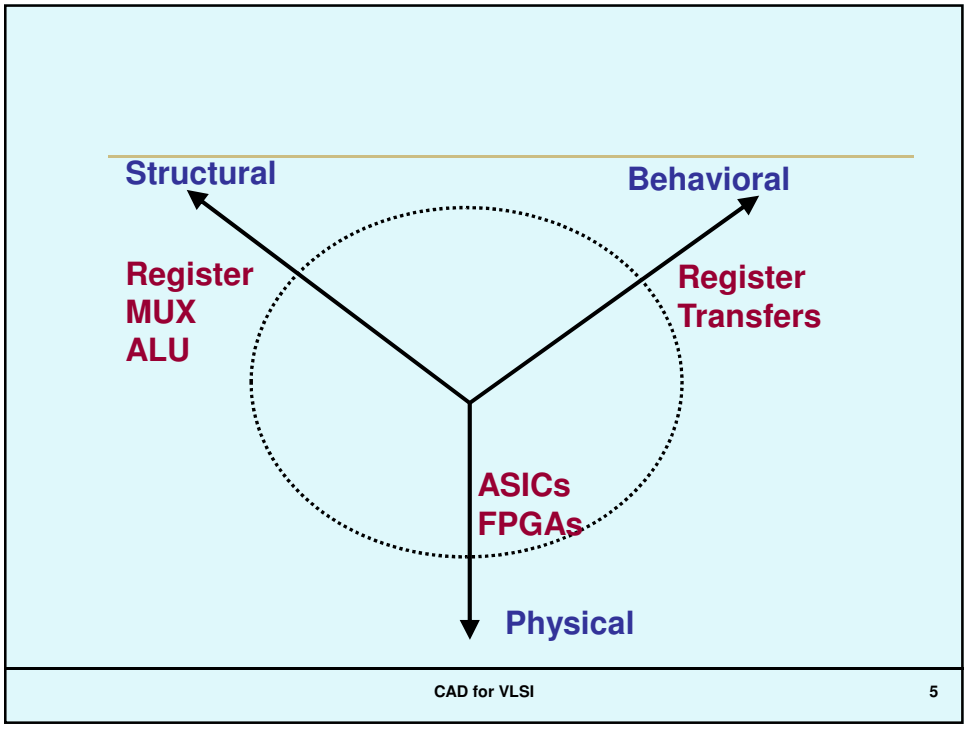
Synthesis

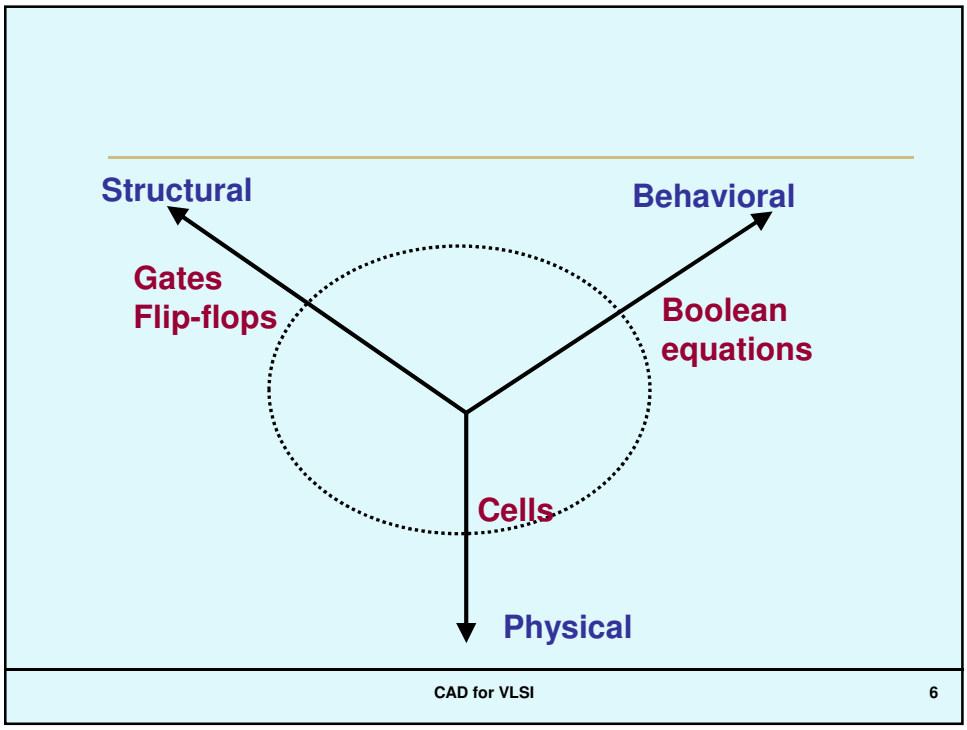
The Y-diagram Revisited

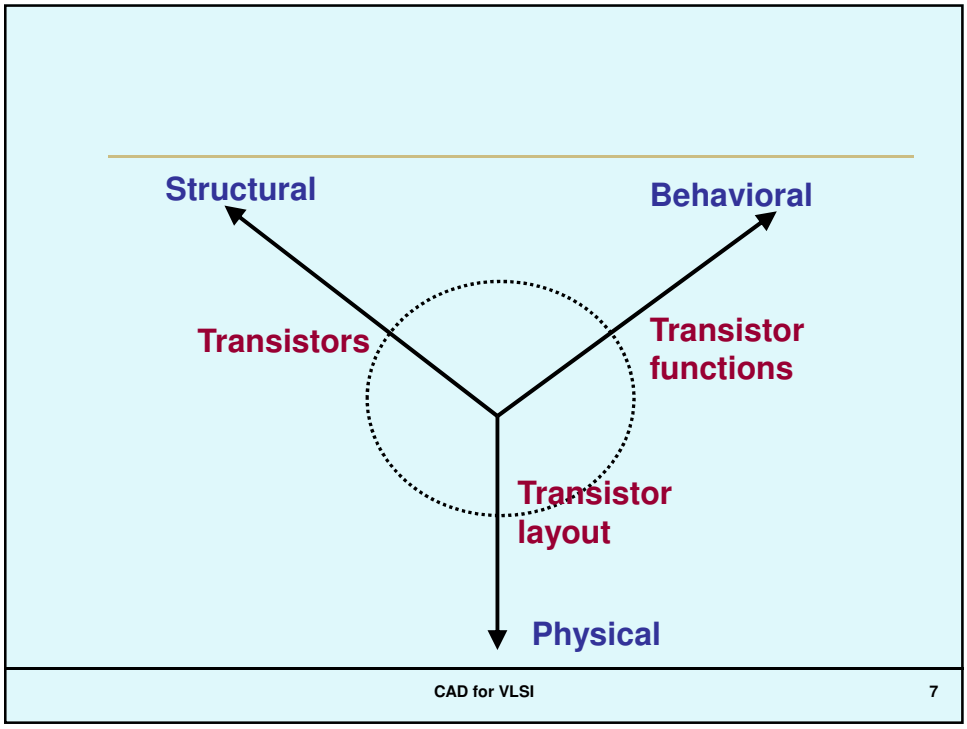



Structural
Behavioral
Synthesis
Physical Design
Physical
CAD for VLSI
3

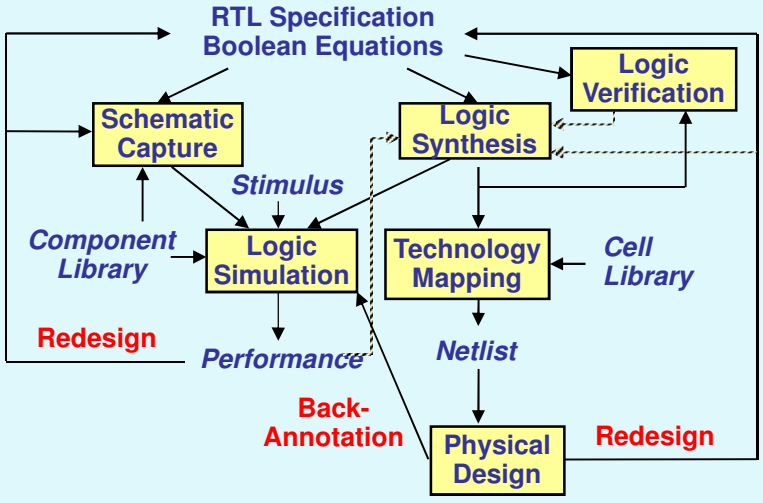

Structural
Behavioral
Processor
Memory
Bus
Algorithm
Flowchart
PCBs
MCMs
Physical
CAD for VLSI
4


Structural
Behavioral
Register
MUX
ALU
Register
Transfers
ASICs
FPGAs
Physical
CAD for VLSI
5


Structural
Behavioral
Gates
Flip-flops
Boolean
equations
Cells
Physical
CAD for VLSI
6



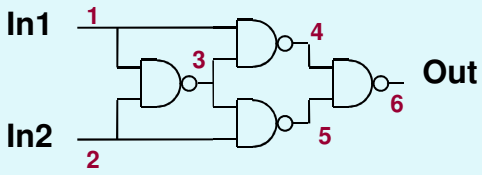
Logic Design



Schematic Capture

- Schematic
  - Graphical representation of a netlist of components.
- Schematic Capture
  - Interactive creation of a schematic
  - Using a schematic editor
    - Uses component icons
    - Picks up components from library
    - Creates netlist
  - Input to simulation & synthesis tools

Schematic



Netlist

nand2	1	2	3
nand2	1	3	4
nand2	2	3	5
nand2	4	5	6

Logic Simulation

- Takes a logic level netlist as input, and simulate functional behavior.
  - "Netlist" obtained from schematic capture or synthesis.
  - For simulation, the behavior of components is used.
    - Available from component library
    - Gates, flip-flops, MUX, registers, adder
- Ability to handle large circuits (millions of gates)
  - Should be very fast
  - Hardware accelerators

- Simulation Objectives
  - Functional correctness of the netlist
    - Requires application of a set of test vectors ➔ test bench
  - Timing analysis
    - Estimation of delay, critical paths
    - Hazards, races, etc.
  - Test generation
    - Required for manufacture test
    - To be discussed later

Logic Synthesis

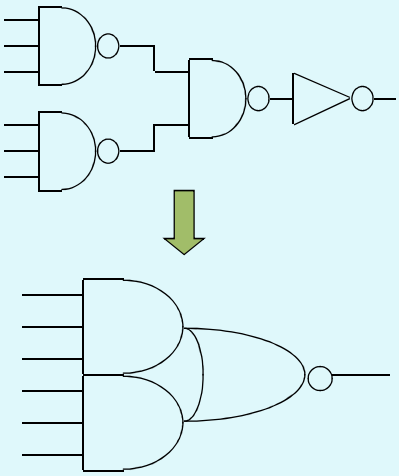
- Input: Boolean equations and FSMs
- Output: A netlist of gates and flip-flops
  - Combinational circuits and sequential circuits are typically handled separately
- Design Goals:
  - Minimize number of levels (delay)
  - Minimize number of gates (area)
  - Minimize signal activity (power)
- Typical Constraints:
  - Target library (say, only NAND and NOT gates)

- Special Considerations
  - Ability to handle large circuits within a reasonable amount of time.
    - Problem is known to be NP-complete
  - Ability to handle mutually conflicting requirements (area & delay)
  - Typically a fully automated process
    - Algorithms/heuristics well understood
    - Do not need user intervention
  - Use technology dependent considerations
    - Break a 20-input gate into smaller gates
    - Use gates available in the library

Technology Mapping

- **Basic Concept**:
  - During logic synthesis, map portions of the netlist to "cells" available in the cell library
    - Standard library (NAND, NOR, NOT, AOI, etc)
    - FPGA cells, standard cells
- **Objectives**:
  - Minimize area, delay, power
  - Should be fast
    - Able to handle large circuits, and large technology libraries

An Example



Logic Verification

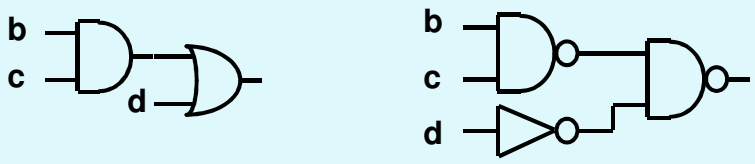
- Verify that the synthesized netlist matches the original specification
  - Detect design errors, also synthesis errors
  - Basic objective is to ensure functional correctness, and to locate errors, if any
- Broadly two approaches:
  1. Simulation
     - Fast, incremental, can handle large circuits
  2. Formal verification
     - Slow, exhaustive, for small circuits only

Logic Synthesis

The Basic Problem

- Convert from logic equations to gate-level netlists (assume combinational logic).
  - Maximize speed
  - Minimize area, power

a′bc + abc + d → bc + d



Logic Specification

- PLA Format

```
.i 3
.o 3
.p 4
1x1 011
x00 010
1x0 100
x11 011
.e
```

- Sum-of-product form

  x = ac’
  y = ac + b’c’ + bc
  z = ac + bc

Logic Synthesis Problem

1. Simplification of logic equations
   - Reduce number of literals (and operands)
2. Synthesis
   - Map logic equations to gates (AND, OR, etc)
3. Gate-level optimization
   - Replace OR-NOT by NOR, for example
   - Delay, power, area
4. Technology mapping
   - Map from gates to technology library
   - FPGA, TTL chips, standard cells, etc

Two Level Logic Minimization

Basic Approaches

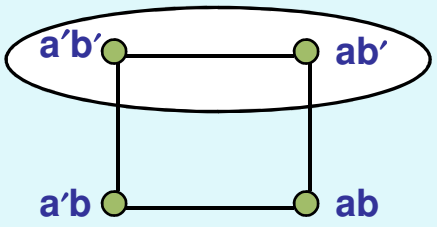
- Karnaugh Maps
  - For n inputs, the map contains 2^n entries
  - Objective is to find minimum prime cover
    - Minimum ➔ fewest terms
    - Prime ➔ choose only maximal covers
  - Don't care terms are used to advantage
  - Difficult to automate
    - Minimum cover problem is NP-complete
    - Process can get into a local minima

- Problems with K-maps:
  - Number of cells is exponential in the number of input variables.
    - Imagine a 50-input circuit
  - Requires efficient data structures
    - For representing the function
    - For searching for minimal prime cover
  - Quine-McCluskey method
    - Easy to implement in software
    - Computational complexity remains high

Espresso: A 2-level logic optimizer

- Some notations:
  - For an n-input function, n-dimensional Boolean space
    - Each point mapped to a unique combination of the n literals
    - Entries in K-map, minterm
  - Cube
    - Conjunction (AND) of literals in an n-dimensional space
    - Points on the n-dimensional hypercube that are "1"

- Expression
  - Disjunction (OR) of cubes
- Don't cares
  - Literals that are missing from a cube



Cube: b′
Don't care: a

- **Basic Approach**
  - Minimize cover of "ON-set" of the function
    - ON-set ➔ set of vertices that correspond to "1" minterms
    - Minimum set of cubes
    - Size of the cubes can be increased by exploiting don't care literals

- **The Espresso Algorithm (Outline)**
  - Start with the sum-of-products form (i.e., cubes covering the ON-set).
  - Expand, remove redundancy (irredundant) and reduce cubes in an iterative loop, until no further improvement is possible.
  - Perturb the solution, and repeat the previous iterative steps, as long as the time budget permits.
    - For each cube, add a subcube not covered by any other cube.
    - Expand subcubes and add them if they cover another cube.

ESPRESSO Algorithm

```
Forig = ON-set;              /* vertices with expression TRUE  */
R = OFF-set;                 /* vertices with expression FALSE */
D = DC-set;                  /* vertices with expression DC    */
F = expand(Forig, R);        /* expand cubes against OFF-set   */
F = irredundant(F, D);       /* remove redundant cubes         */
do {
   do {
      F = reduce(F, D);    /* shrink cubes against ON-set */
      F = expand(F, R);
      F = irredundant(F, D);
   } until cost is "stable";
   /* perturb solution */
   G = reduce_gasp(F, D); /* add cubes that can be reduced */
   G = expand_gasp(G, R); /* expand cubes that cover another */
   F = irredundant(F+G, D);
} until time is up;
ok = verify(F, Forig, D); /* check that result is correct */
```

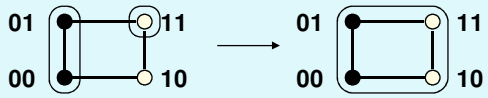

Cube operation :: expand

- Make each cube as large as possible without covering a point in the OFF-set.
  - Increases the number of literals in the cover.
  - Sets the stage for finding a new and possibly better solution.
- Example:

f = a′bc′+ bc + ab′c′ Don't care: ab′c

f = a′b + bc + ac + ab′

Expand



ON
DC
OFF

Cube operation :: irredundant

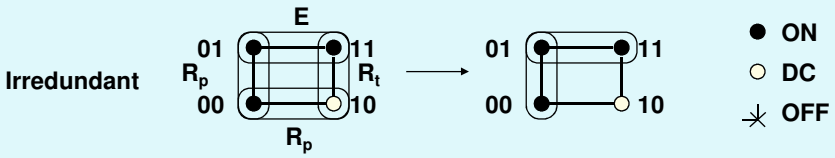
- Throw out redundant cubes.
  - Points may be covered by several cubes after the 'expand" step.
  - Remove smaller cubes whose points are covered by larger cubes.
  - There must be one cube for every essential vertex.
- Example:

  f = a′b + bc + ac + ab′

  ⇩

  f = a′b + ac + ab′

One vertex in bc is covered by a′b & the other by ac



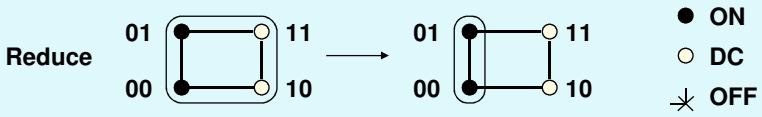
Cube operation :: reduce

- The cubes in the cover are reduced in size.
  - The number of literals in the cover is reduced.
  - Smaller cubes can expand in more directions.
  - Smaller cubes are more likely to be covered by other cubes during expansion.
- Example

  f = a′b + ac + ab′

  ⇩

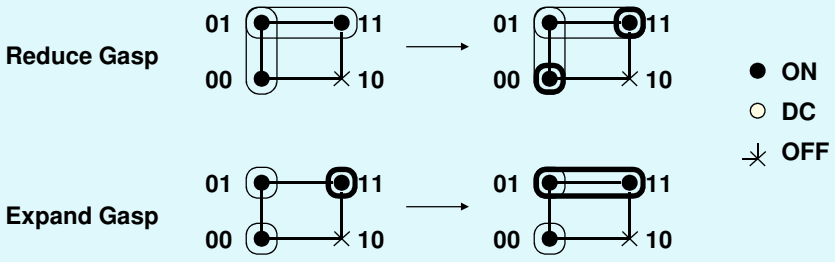
  f = a′b + abc + ab′c′



– In general, the new cover will be different from the initial cover.
  - "expand" and "irredundant" steps can possibly find out a new way to cover the points in the ON-set.
  - Hopefully, the new cover will be smaller.

Cube operation :: perturbations

- **Reduce Gasp**
  - For each cube add a subcube not covered by other cubes.
- **Expand Gasp**
  - Expand subcubes and add them if they cover another cube.
  - Later use “irredundant” to discard redundant cubes.
  - This is a “last gasp” heuristic for exploration.
    - No ordering of cube size.



- **Example:**

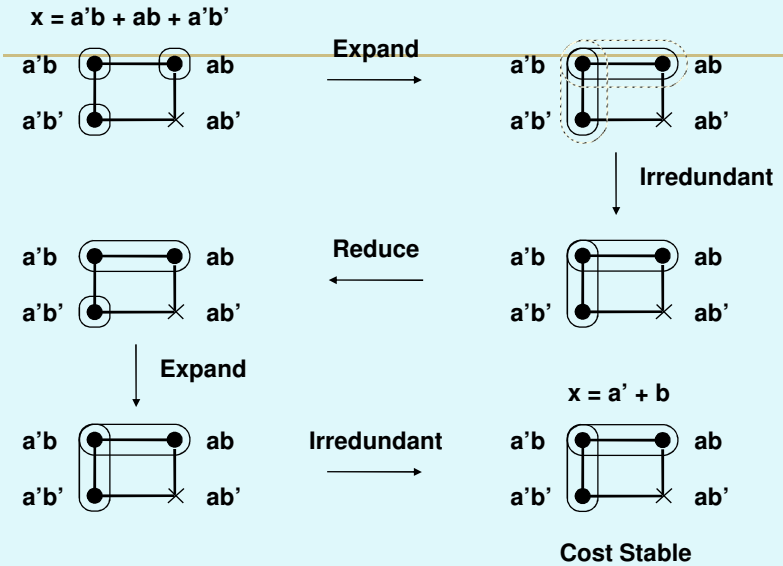
  $f = a' + b \rightarrow f = a' + b + a'b' + ab$

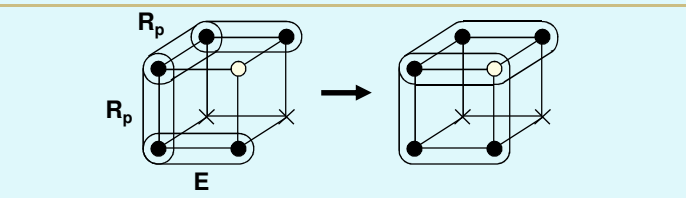
  (Reduce Gasp)

  $f = a'b' + a'b + ab \rightarrow f = a'b' + a'b + b$

  (Expand Gasp)

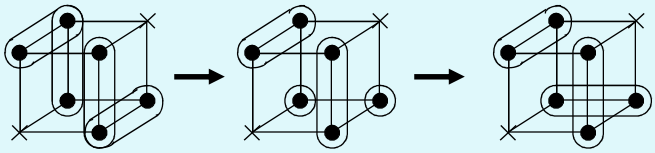
An example





Essential and Redundant Cubes

Prime & Irredundant Cover



Initial Cover

Reduce

Expand in right direction

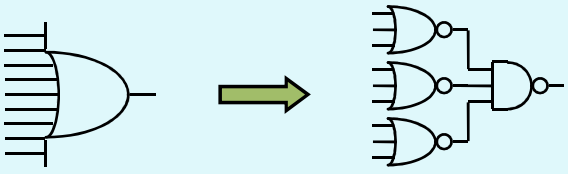
Espresso :: conclusion

- The algorithm successively generates new covers until no further improvement is possible.
- Produces near-optimal solutions.
- Used for PLA minimization, or as a sub-function in multilevel logic minimization.
- Can process very large circuits.
  - 10,000 literals, 100 inputs, 100 outputs
  - Less than 15 minutes on a high-speed workstation

Multi Level Logic Minimization

Motivation

- In many applications, 2-level logic is unsuitable as compared to random (multilevel) logic.
  - Gates with high fanin are slow, and take more area.
  - It makes sense to transform a 2-level logic realization to multi-level logic.



- **A classical example :: XOR function**
  - For an 8-input XOR function,
    - For 2-level NAND-NAND realization

      $^8C_1 + {}^8C_3 + {}^8C_5 + {}^8C_7$ = 128 NAND8 gates

      1 NAND128 gate
    - For 3-level XOR realization

      7 XOR2 gates

      ➔ 28 NAND2 gates

      Number of levels = 9

- **Multilevel logic optimization approaches:**
  1. Local optimization
     - » Rule-based transformation
  2. Global optimization
     - » Weak division

Local Optimization Technique

- Used in IBM Logic Synthesis System.
- Perform rule-based local transformations.
  - Objective ➔ reduce area, delay, power.
  - Developing a good set of rules is a challenge.
  - Should be comprehensive enough so as to completely explore the design space.
- Basic idea:
  - Apply a transformation which reduces cost.
  - Iterate and continue the transformations as long as solution keeps improving.

- AND/OR transformations
  - Reduce the size of the circuit, critical path.
  - Typical transformations:

    a . 1 = a

    a + 1 = 1

    a + a′ = 1

    a . a′ = 0

    (a′)′ = a

    a + a′ . b = a + b

    xor (xor(a1,a2,…,an), b) = xor (a1,a2,…,an,b)
- Transform the AND/OR form to NAND (or NOR) form.

- **NAND (NOR) transformations**
  - Some synthesis systems assume that all gates are of the same type (NAND or NOR).
  - Does not require technology mapping.
  - Rules framed that transform one NAND (NOR) network to another.
  - Examples:

    NAND (NOT (NAND (a,b)), c) = NAND (a,b,c)

    NAND (NAND(a,b,c), NAND(a,b,c')) = NAND (a,b)

How complex is the algorithm?

- n → number of circuit nodes

  m → number of rules
  - Ordering of rule (by cost reduction) takes *O(mn log mn)* time.
  - The process has to be repeated many times.
- To speed up, we can use *lazy evaluation*.
  - We only check those circuit nodes which were modified in the previous iteration.
  - *O(m log m)* for every rule application.

Global Optimization Technique

- Used in GE Socrates.
  - Looks at all the equations at one time.
- Perform weak division.
  - Divide out common sub-expressions.
  - Literal count gets reduced.
- The following iterative steps are carried out:
  - Generate the candidate sub-expressions.
  - Select a sub-expression to divide.
  - Divide functions by selected sub-expression.

Example

- Original equations:

  $f_1 = a.b.c + b.c.d + b.e.g$ $f_2 = b.c.f.h + d.g + a.b.g$

  ➔ No. of literals = 18

  - We find literals saved for sub-expressions:

    b.c ➔ 4 a.b ➔ 2

    a + d ➔ 2 b.g ➔ 2

    Select the sub-expression bc.

- Modified equations (after iteration 1):

  $f_1 = (a + d).u + b.e.g$

  $f_2 = u.f.h + d.g + a.b.g$

  $u = b.c$ ➔ No. of literals = 14+2

$f_1 = (a + d).u + b.e.g$
$f_2 = u.f.h + d.g + a.b.g$
$u = b.c$

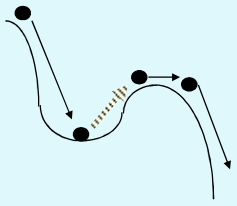
– Literals saved for the sub-expressions:
b.g ➔ 2

- Modified equations (after iteration 2):
$f_1 = (a + d).u + e.v$
$f_2 = u.f.h + d.g + a.v$
$u = b.c$
$v = b.g$ ➔ No. of literals = 12+4

- No common sub-expressions ➔ STOP

About the algorithm

- Basically a greedy algorithm
  – Can get stuck in local minima.
  – Give a "*push*" to come out of local minima.
    - Like the "gasp" function in Espresso.



  – Generation of all candidate expressions is expensive.
    - Some heuristic used.

Multilevel Logic Interactive Synthesis System (MIS)

- A very popular & widely used algorithm.
  - Uses factoring of equations.
  - Similar to weak division used in Socrates.
  - The target technology is CMOS gate.
    - Complex gates realizing any complex functions.
    - Example:

      $f' = (a + b + c)$

      $g' = (a + b) . (d + e + f) . h$

Basic Concept

- For global optimization,
  - Use algebraic factorization to identify common sub-expressions.
  - Avoid exponential search.
- For local optimization,
  - Identify 2-level sub-circuits.
  - Minimize them using Espresso, or some similar approach.

Global Optimization Approach

- Given a netlist of gates
  - Scan the network.
  - Apply simple heuristics to "clean up" the netlist.
    - Constant propagation
    - Double inverter elimination
    - Espresso minimization of each equation
  - Then proceed for global optimization with a view to minimize area.

  - Basically an iterative approach.
    - Enumerate all common factors and identify the "best" candidate.
    - Equations themselves may be common factors.
    - Invert an equation if it helps.
      - Factors may show up in the inverted form.
    - Number of literals used to estimate area.
- Factoring can reduce area.
  - An equation in simple sum-of-products form can have many literals.
    - Many transistors for CMOS realization.
  - Factoring the equation reduces the number of literals.
    - Reduces number of transistors in CMOS realization.

Local Optimization Approach

- Next step is to look at the problem locally.
  - Each equation treated as a complex gate.
    - Optimize two or more gates that share one or more literals.
    - Break a large gate into smaller gates.
  - For each equation, the don't care input set is obtained from the neighborhood gates.
    - Minimized using Espresso.
- Also an iterative step.

Binary Decision Diagrams (BDD)

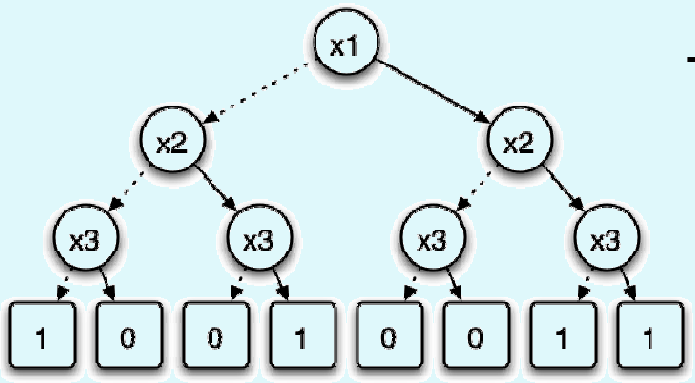
Introduction

- Representation of Boolean functions
  - Canonical
    - Truth table
    - Karnaugh map
    - Set of minterms
  - Non-Canonical
    - Sum of products
    - Product of sums
    - Factored form
    - Binary Decision Diagram (Proposed by Akers in 1978)

What is Binary Decision Diagram?

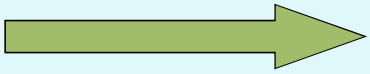
- A data structure used to represent a Boolean function.
- Represented as a rooted, directed, acyclic graph, which consists of *decision nodes* and two *terminal nodes* (0-terminal and 1-terminal).
  - Each decision node is labeled by a Boolean variable and has two child nodes called *low child*, and *high child*.
  - The edge from a node to a low (high) child represents an assignment of the variable to 0 (1).
- A BDD is said to be *ordered* if different variables appear in the same order on all paths from the root.

Example



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

- Construction of a BDD is based on the ***Shannon expansion*** of a function.



Shannon Expansion

- Given a Boolean function $f(x_1,x_2,\ldots,x_i,\ldots,x_n)$
- Positive cofactor

  $f_i^1 = f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Negative cofactor

  $f_i^0 = f(x_1,x_2,\ldots,0,\ldots,x_n)$
- Shannon's expansion theorem states that

  $f = x_i' f_i^0 + x_i f_i^1$

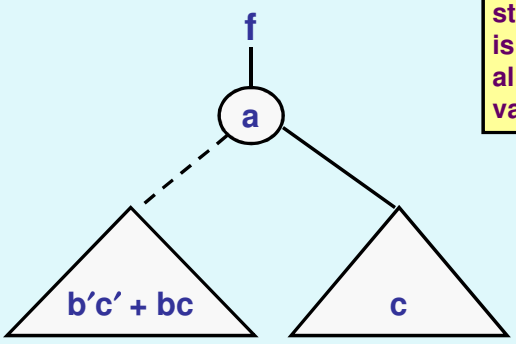
  $f = (x_i + f_i^0)(x_i' + f_i^1)$

How to construct BDD?

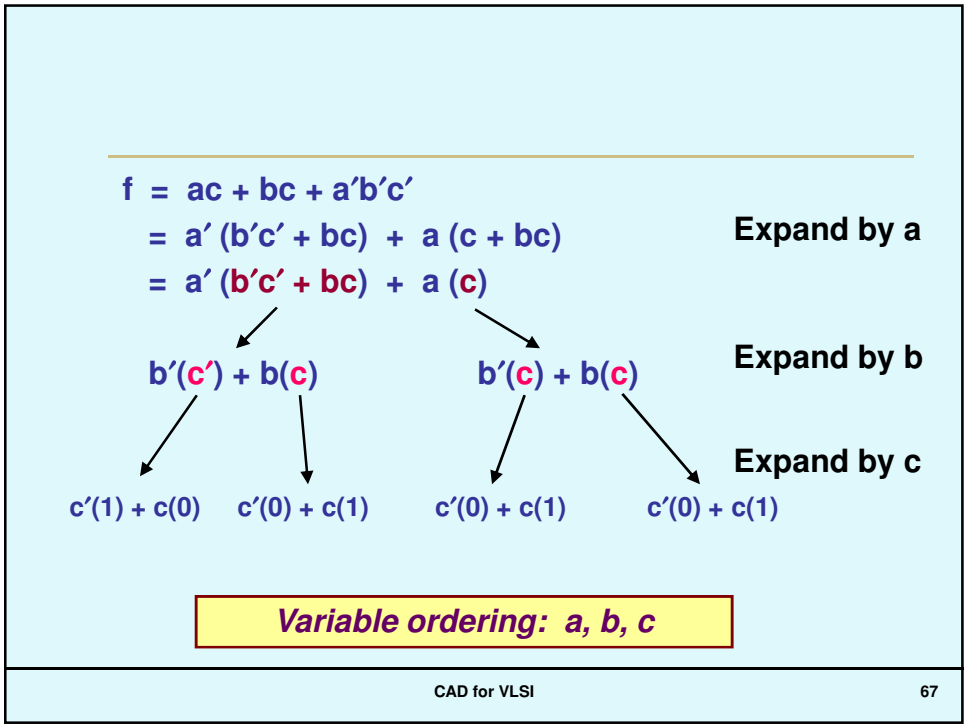
$f = ac + bc + a'b'c'$

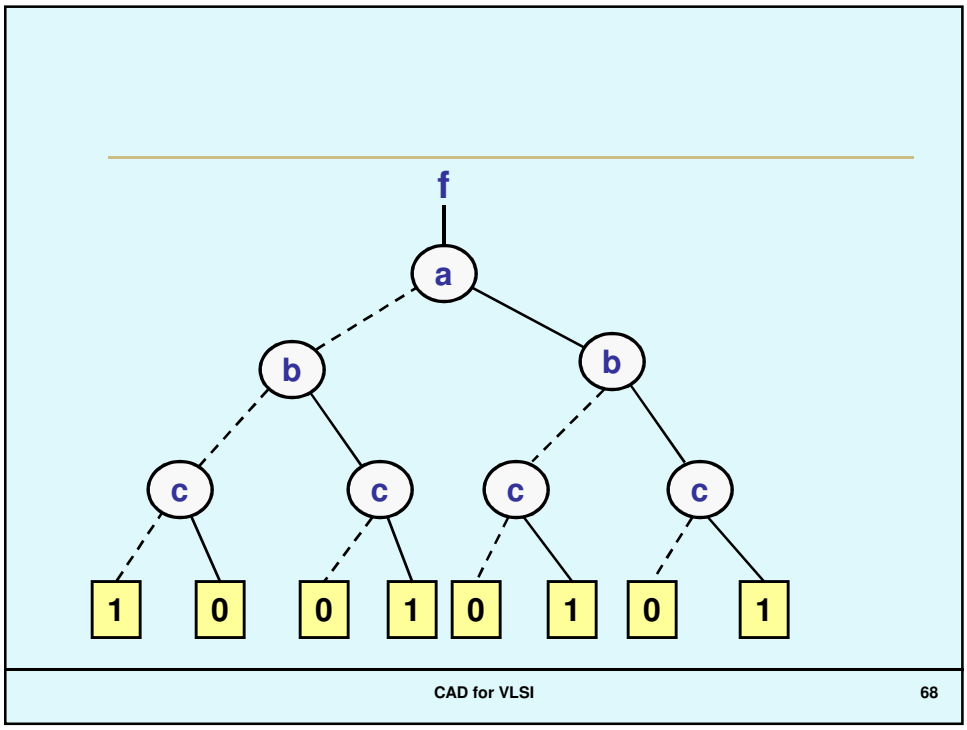
$= a'(b'c' + bc) + a(c + bc)$

$= a'(b'c' + bc) + a(c)$



This is the first step. The process is continued for all input variables.


f = ac + bc + a′b′c′
= a′ (b′c′ + bc) + a (c + bc)
= a′ (b′c′ + bc) + a (c)
Expand by a
b′(c′) + b(c)
b′(c) + b(c)
Expand by b
c′(1) + c(0)
c′(0) + c(1)
c′(0) + c(1)
c′(0) + c(1)
Expand by c
Variable ordering: a, b, c
CAD for VLSI
67


f
a
b
b
c
c
c
c
1
0
0
1
0
1
0
1
CAD for VLSI
68

Variable Ordering (OBDD)

- The size of a BDD is determined both by the function being represented and the chosen ordering of the variables.
  - For some functions, the size of a BDD may vary between a *linear* to an *exponential* range depending upon the ordering of the variables.
- An example:

$f(x_1,\ldots,x_{2n}) = x_1x_2 + x_3x_4 + \ldots + x_{2n-1}x_{2n}$

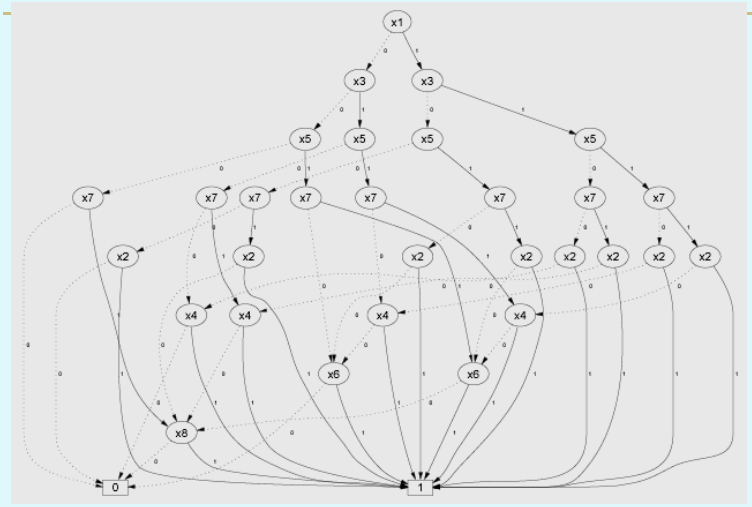
Variable ordering: $x_1 < x_3 < \ldots < x_{2n-1} < x_2 < x_4 < \ldots < x_{2n}$

BDD requires 2^{n+1} nodes to represent the function.

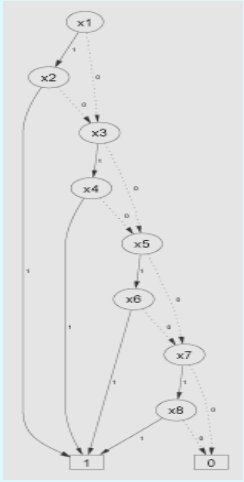
Variable ordering: $x_1 < x_2 < x_3 < x_4 < \ldots < x_{2n-1} < x_{2n}$

BDD requires 2n nodes to represent the function.

BDD for the function $f(x_1, \ldots, x_8) = x_1x_2 + x_3x_4 + x_5x_6 + x_7x_8$ using bad variable ordering



Same function using good variable ordering

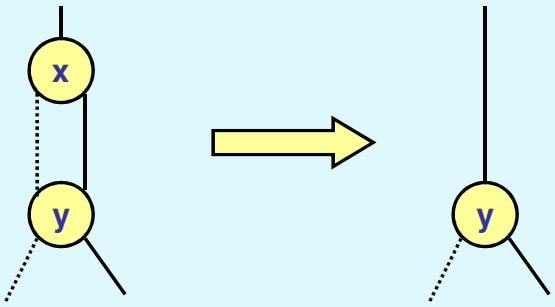


- **Important point to note:**
  - It is essential to find a good variable ordering when using the OBDD data structure in practice.
  - The problem of finding the best variable ordering is NP-hard.
  - Several heuristics for variable ordering have been proposed.

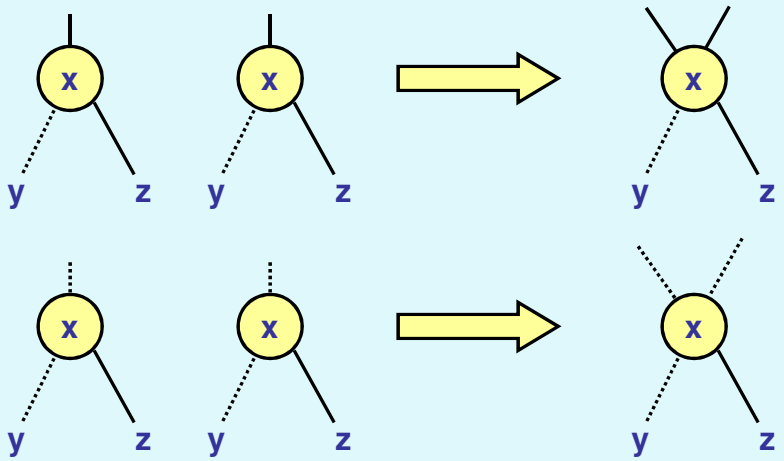
Reduced Ordered BDD (ROBDD)

- An ordered binary decision diagram is said to be reduced (ROBDD) if the following two graph reduction rules are applied:
  - Merge any isomorphic subgraphs.
  - Eliminate any node whose two children are isomorphic.
- The advantage of an ROBDD is that it is canonical (unique) for a given function.
  - This property makes it useful in functional equivalence checking.

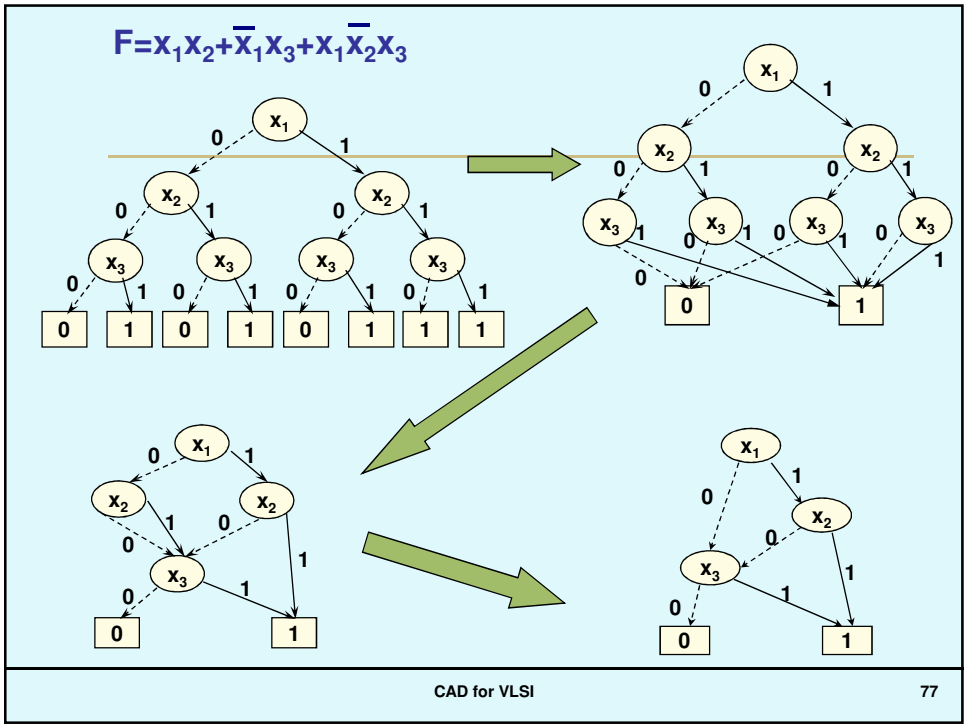
Some Reduction Rules



Some Reduction Rules (contd.)



Construction of ROBDD: an example

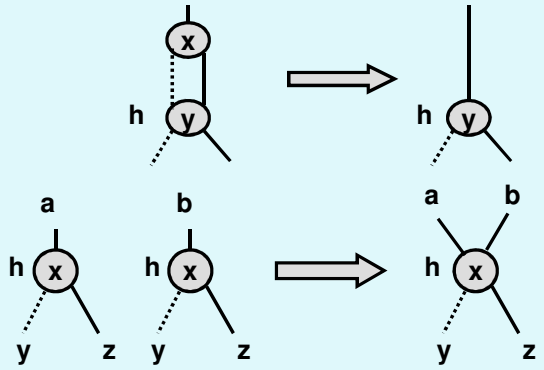


Some Benefits of BDD

- Checking for tautology is trivial.
  - BDD is a constant 1.
- Complementation.
  - Given a BDD for a function f, the BDD for f' can be obtained by simply interchanging the terminal nodes.
- Equivalence check.
  - Two functions f and g are equivalent if their BDDs (under the same variable ordering) are the same.

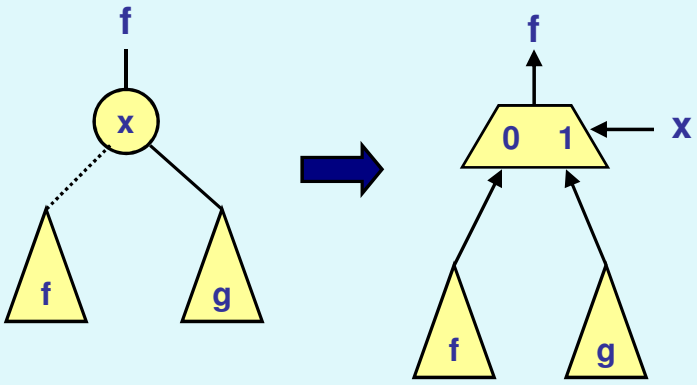
Use of BDD in Synthesis

- BDD is canonical for a given variable ordering.
- It implicitly uses factored representation:

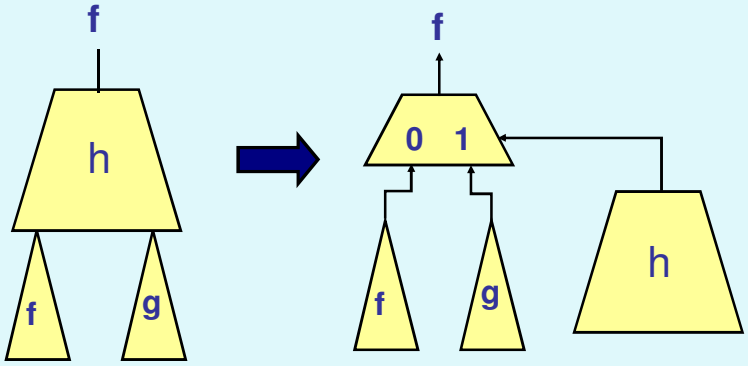


- Variable reordering can reduce the size of BDD.
  - Implicit logic minimization.
- Some redundancy is also removed during the construction of BDD itself.

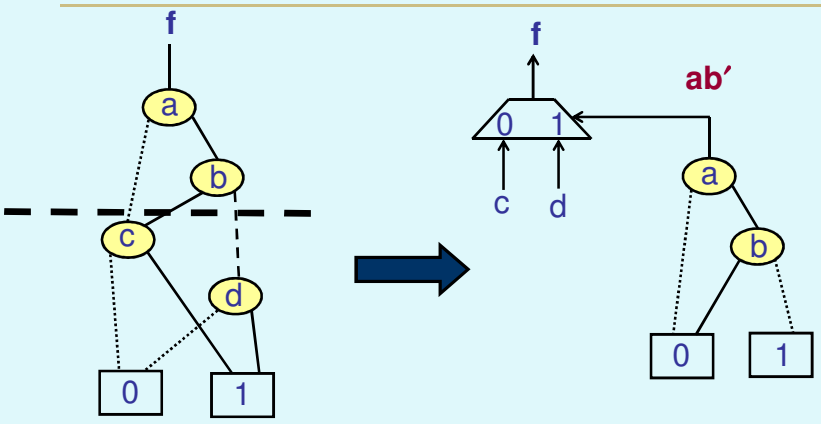
MUX realization of functions



MUX-based Functional Decomposition



An example ===>



To Summarize

- BDDs have been used traditionally to represent and manipulate Boolean functions.
  - Used in synthesis systems.
  - Used in formal verification tools.
  - Efficient packages to manipulate BDDs are available.