Synthesis of Digital Systems

COL 719

Part 5: Logic Synthesis

Preeti Ranjan Panda
Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Logic Synthesis

- Introduction
- Logic Optimisation (technology independent)
  - Two-level optimisation
  - Multi-level optimisation
- Technology Mapping / Cell library binding (technology dependent)
  - ASIC
  - FPGA

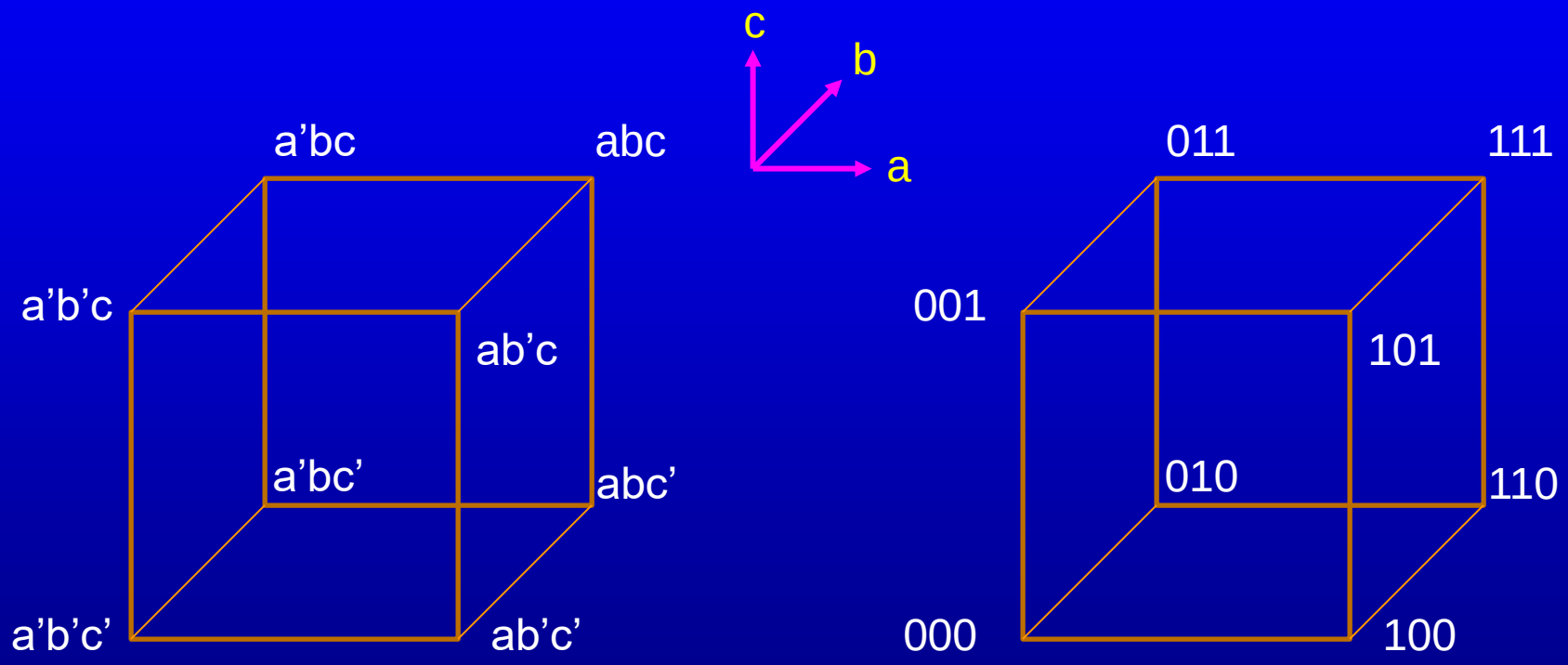
Boolean Algebra and Functions

- Set B = {0,1} (binary boolean algebra)
- Operations AND/product (.), OR/sum (+)
  - commutative
    - a.b = b.a
    - a+b = b+a
  - distributive
    - a.(b+c) = a.b + a.c
    - a+(b.c) = (a+b).(a+c)
- Complement
  - a.a’ = 0
  - a+a’ = 1

Boolean Algebra Properties

- Associativity
  - a+(b+c) = (a+b)+c
  - a.(b.c) = (a.b).c
- Idempotence
  - a+a = a
  - a.a = a
- Absorption
  - a + ab = a
  - a (a+b) = a
- De Morgan's Laws
  - (a+b)' = a'b'
  - (ab)' = a'+b'
- Involution
  - (a')' = a

3-D Boolean Space



2^n vertices (corners) for n variables

Boolean Functions

Mapping $f : B^n \rightarrow B^m$

n inputs
m outputs

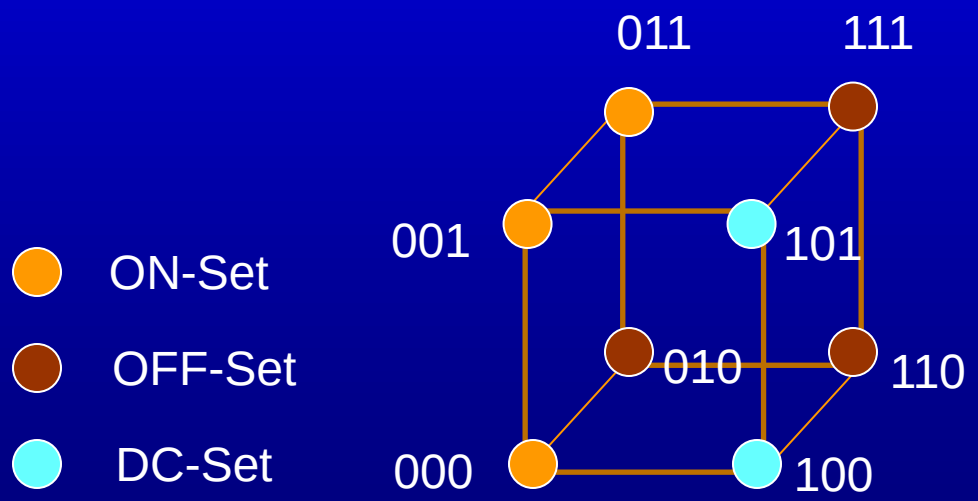
Incompletely Specified Function:
$f : B^n \rightarrow \{0,1,-\}^m$

- '-' represents DON'T CARE
- Output not relevant for specified point
- Can be exploited during optimisation

ON, OFF, DC Sets

- ON-Set: Subset of domain for which function value is 1
- OFF-Set: Subset of domain for which function value is 0
- DC-Set: Subset of domain for which function value is '-'

a b c	y
0 0 0	1
0 0 1	1
0 1 0	0
0 1 1	1
1 0 0	-
1 0 1	-
1 1 0	0
1 1 1	0



Multiple Output Functions

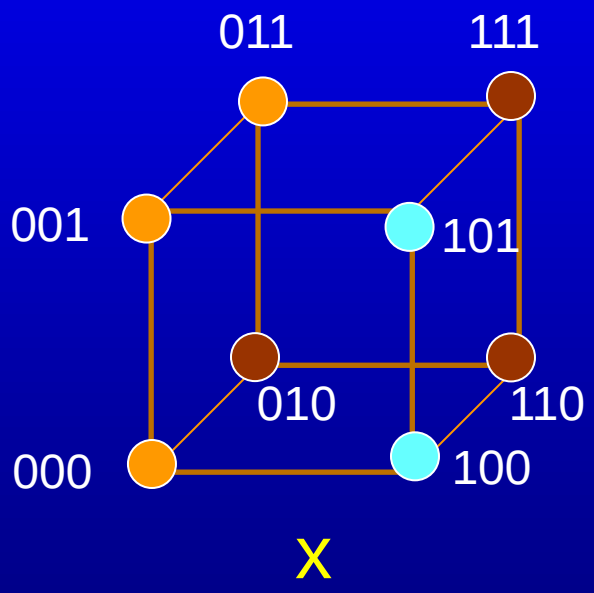
a b c	x y
0 0 0	1 1
0 0 1	1 0
0 1 0	0 1
0 1 1	1 1
1 0 0	- 0
1 0 1	- 0
1 1 0	0 -
1 1 1	0 -

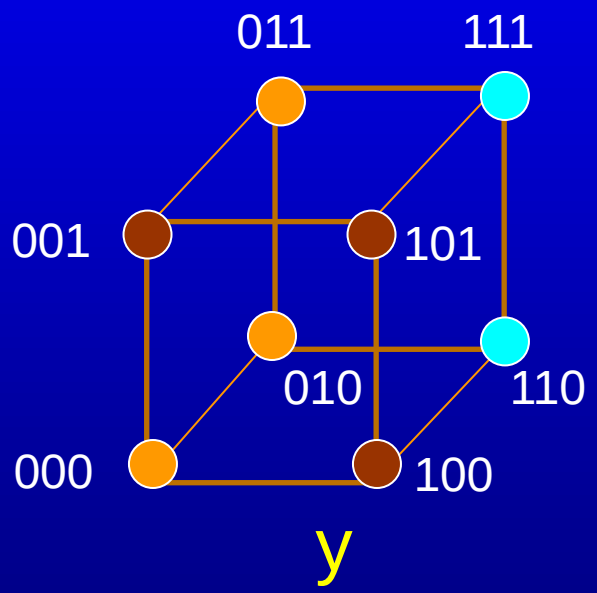
ON-Set

OFF-Set

DC-Set

One cube (n-dim) per output





Co-factors

- For function $f(x_1, x_2,\ldots, x_i,_{..}, x_n)$
- Co-factor of f with respect to variable x_i $f_{xi} = f(x_1, x_2,\ldots, 1,_{..}, x_n)$
- Co-factor of f with respect to variable x_i' $f_{xi'} = f(x_1, x_2,\ldots, 0,_{..}, x_n)$

Example

$f(a,b) = ab + a'b'$

$f_a = f(1,b) = b$

$f_{a'} = f(0,b) = b'$

Shannon's Expansion

- For any Boolean function $f : B^n \rightarrow B$

$$f(x_1, x_2, \ldots, x_i, \ldots, x_n) = x_i.f_{xi} + x_i' f_{xi'}$$

Example

$f(a,b) = ab + a'b'$

$f_a = f(1,b) = b$

$f_{a'} = f(0,b) = b'$

$a.f_a + a'.f_{a'} = ab + a'b' = f(a,b)$

Shannon's Expansion: Informal derivation

$$f(x_1, x_2, \ldots, x_i, \ldots, x_n) = x_i.f_{xi} + x_i' f_{xi'}$$

Group into
terms with a
terms with a'
terms with no a,a'

f (a,b,c,...)
= abc + a'b + b'c' + ac + d +...
= a (bc + c + ...) + a'(b + ...) + (b'c' + d +...)
= a (bc + c + ...) + a'(b + ...) + (a+a')(b'c' + d +...)
= a (bc + c + b'c' + d +...) + a'(b + b'c' + d +...)
= a.f (1,b,c,...) + a'.f (0,b,c,...)
= $a.f_a + a'.f_{a'}$

Representation of Boolean Functions

- Tabular forms
  - Truth table
  - Multiple output implicant
- Expression forms
- Binary Decision Diagrams

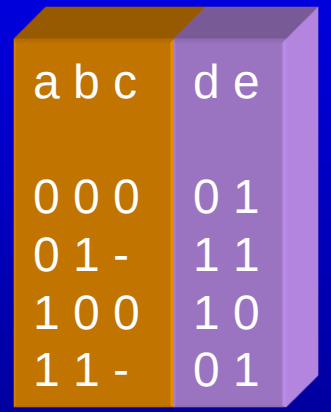
Tabular Forms: Truth Table

- 2-D table - input and output parts
- Complete listing of all points in the input space and corresponding outputs
- n-input, m-output combinational function
  - Input part - set of all row vectors in $\{0,1\}^n$
  - Output part - set of corresponding row vectors in $\{0,1,-\}^m$
- Exponential Size (2^n rows)

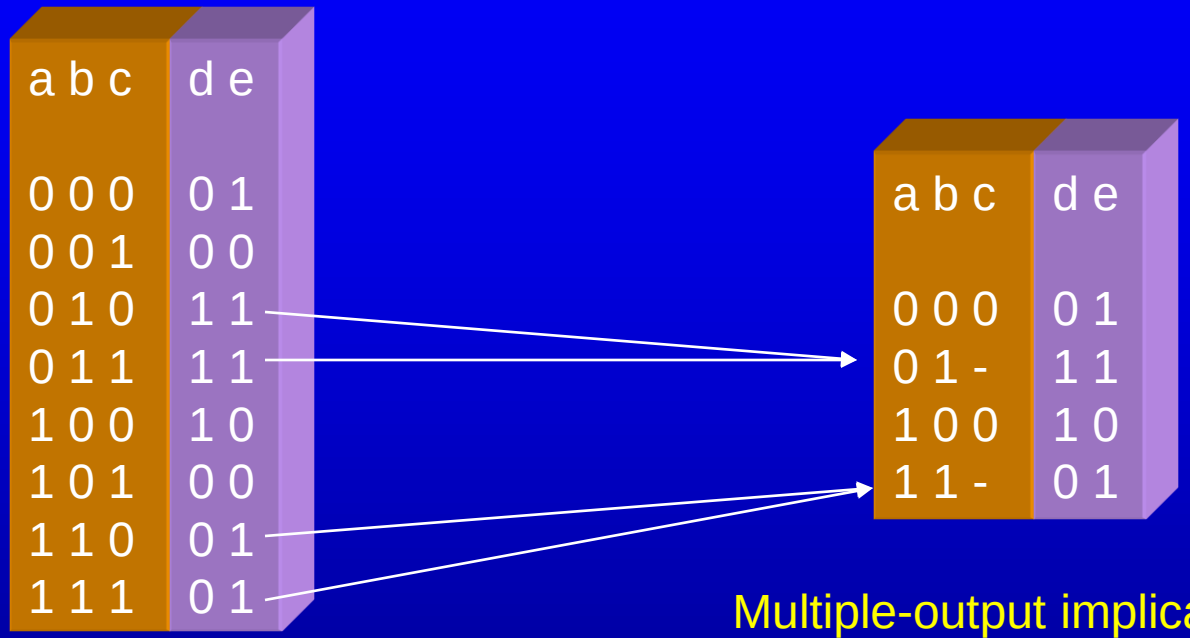
n=3 i/p	m=2 o/p
a b c	d e
0 0 0	0 1
0 0 1	0 0
0 1 0	1 1
0 1 1	1 -
1 0 0	1 0
1 0 1	- 0
1 1 0	0 1
1 1 1	0 1

Tabular Forms: Multiple-Output Implicant

- 2-D table - input and output parts
- n-input, m-output combinational function
  - Input part - set of cubes, i.e., row vectors in $\{0,1,-\}^n$
  - Output part - set of corresponding row vectors in $\{0,1,-\}^m$
- Size not necessarily exponential
  - each row now represents set of points in Boolean space (not just one point)



Tabular Forms - Example



Multiple-output implicant

Truth Table

One minterm per row
All input combinations

One input cube per row
Retain cubes that are implicants
  Drop minterms (001, 101) leading to 00 outputs
Combine rows leading to identical outputs using don't care
Table size can reduce

Expression Forms

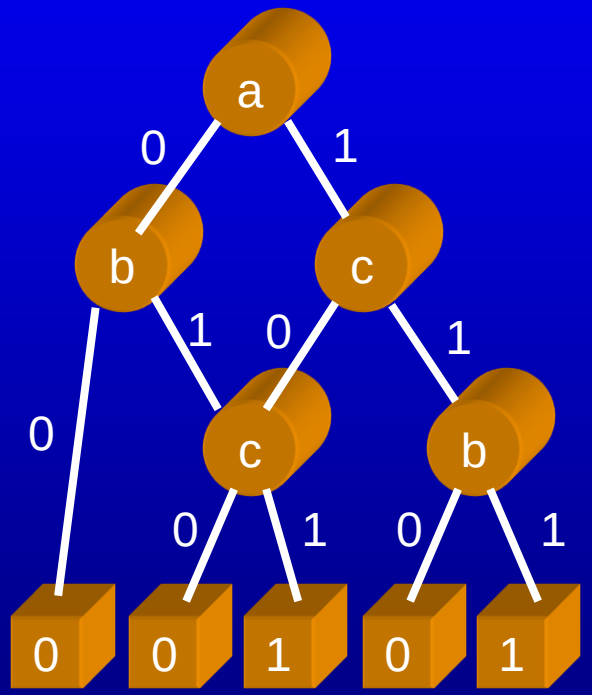
- Boolean functions represented as expression of literals combined by OR (+) or AND(.)
- Single level - only one operator
  - E.g., (1) abc’ (2) a+b+c+d
  - cannot represent all boolean functions
- Two level
  - sum of products (ab+a’b’c)
  - product of sums (a+b+c’)(a’+c)
  - can represent all boolean functions
  - sum of minterms/product of maxterms are canonical forms

Expression Forms...

- Multi-level
  - arbitrary nesting of boolean operators
  - E.g. a(b+c) + (a'+b)(c+d)'
- Factored form
  - literal (f = a or a')
  - sum of factored forms (f = f + f)
  - product of factored forms (f = f . f)
  - Constrained form of multi-level
    - complementation allowed only for variables

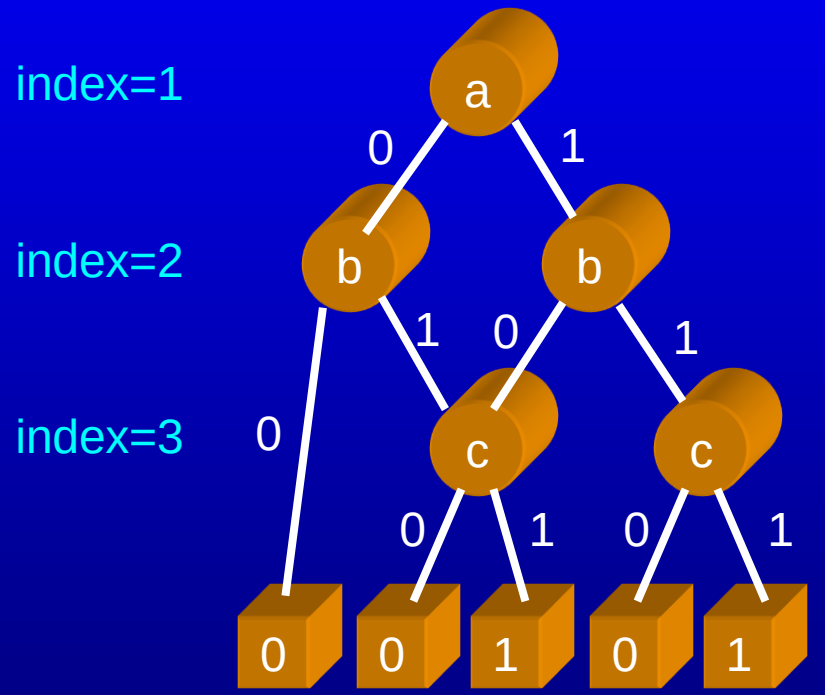
Binary Decision Diagrams (BDD)

- Set of binary (true/false) decisions
- Ending in true/false for whole function
- Represented by Tree/DAG structures
  - nodes represent decisions
    - single variables or complex functions



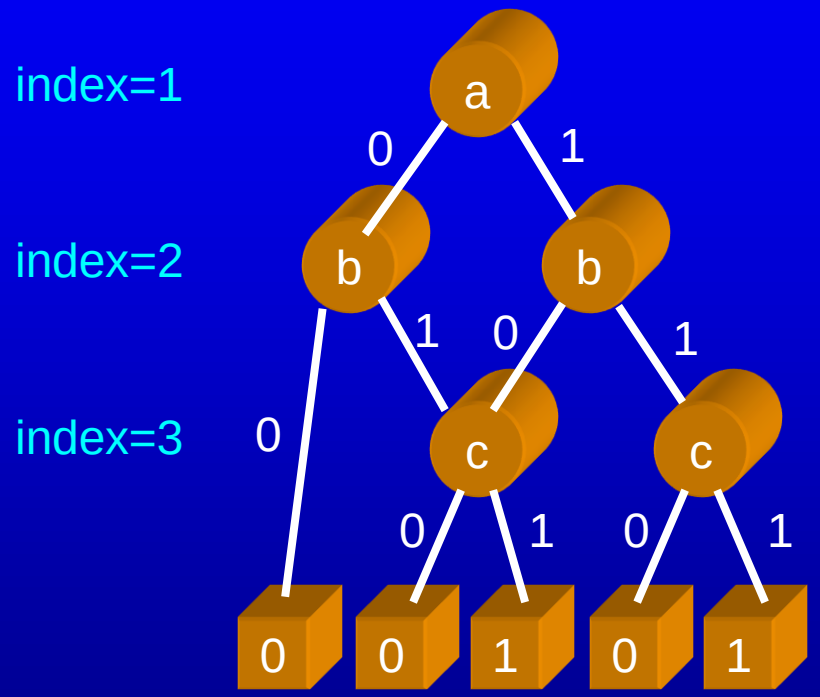
Ordered Binary Decision Diagrams (OBDD)

- Associate an ordering (index value) with the decision variables
- OBDDs can be made canonical
  - representation of a function is unique
- Efficient algorithms can operate on OBDDs
  - polynomial in # nodes
  - ...but # nodes can be exponential
  - in practice, we can often find orderings for which # nodes is not large



OBDD Definition

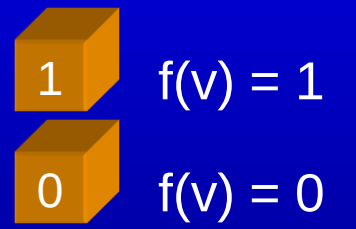
- Rooted directed graph G(V,E)
- Node v has attribute index(v) corresponding to an input variable
- Leaf node has attribute value (v) $\in$ {0, 1}
- Each node v has 2 children low(v) and high(v)
- If v, low(v), and high(v) are non-leaf
  - index (v) < index (low (v))
  - index (v) < index (high (v))
- Graph is acyclic

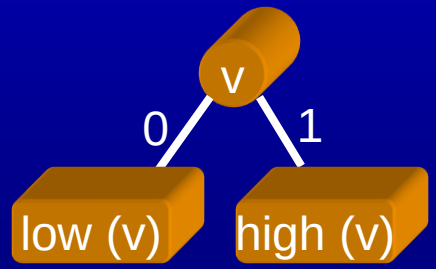


Boolean Function associated with OBDDs

- OBDD with root v
- index (v) represents variable x
- OBDD represents function f
  - if v is a leaf with value(v)=1 then f(v)=1
  - if v is a leaf with value(v)=0, then f(v)=0
  - if v is not a leaf, then:

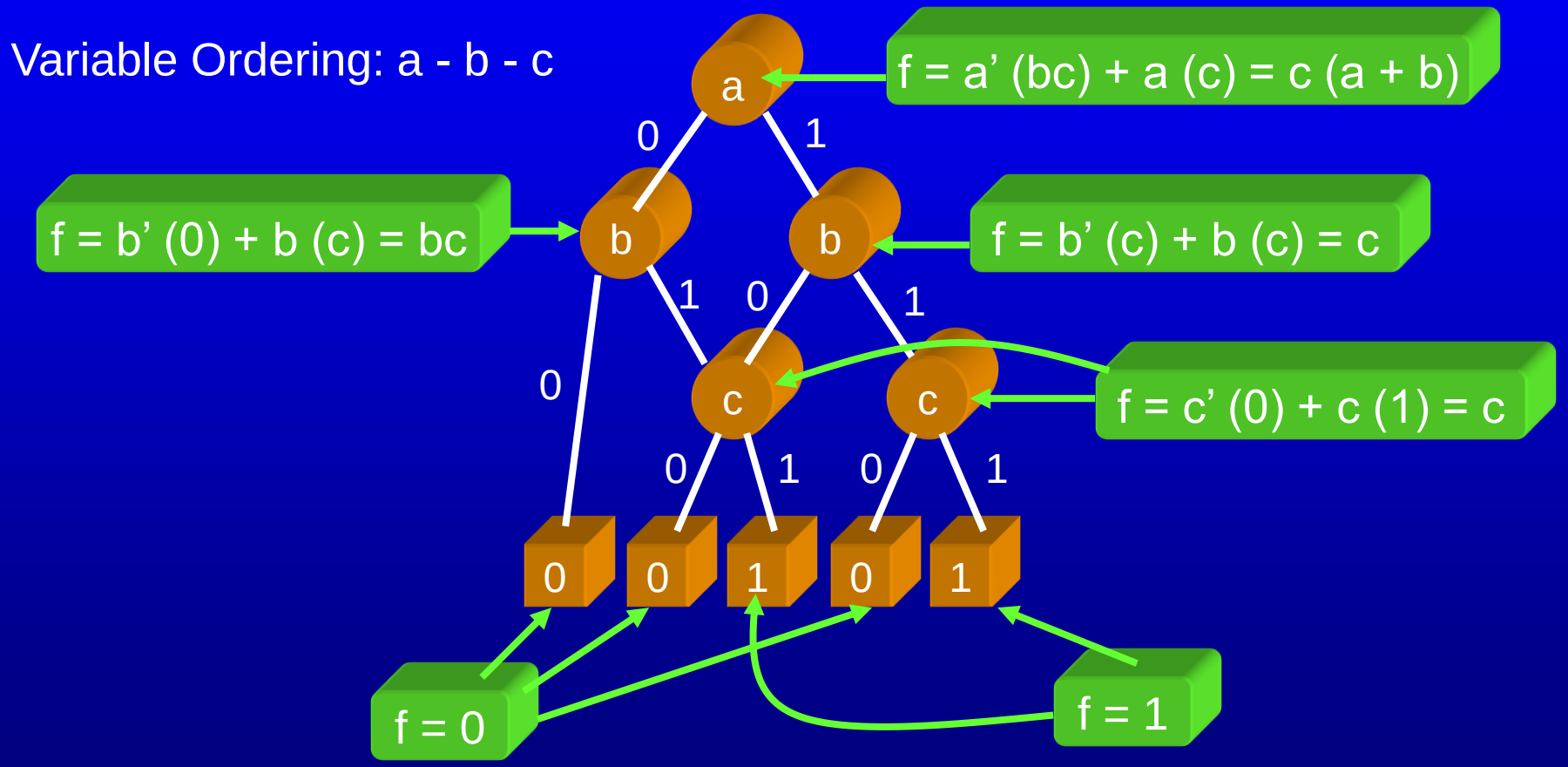
$$f(v) = x' f(low(v)) + x f(high(v))$$



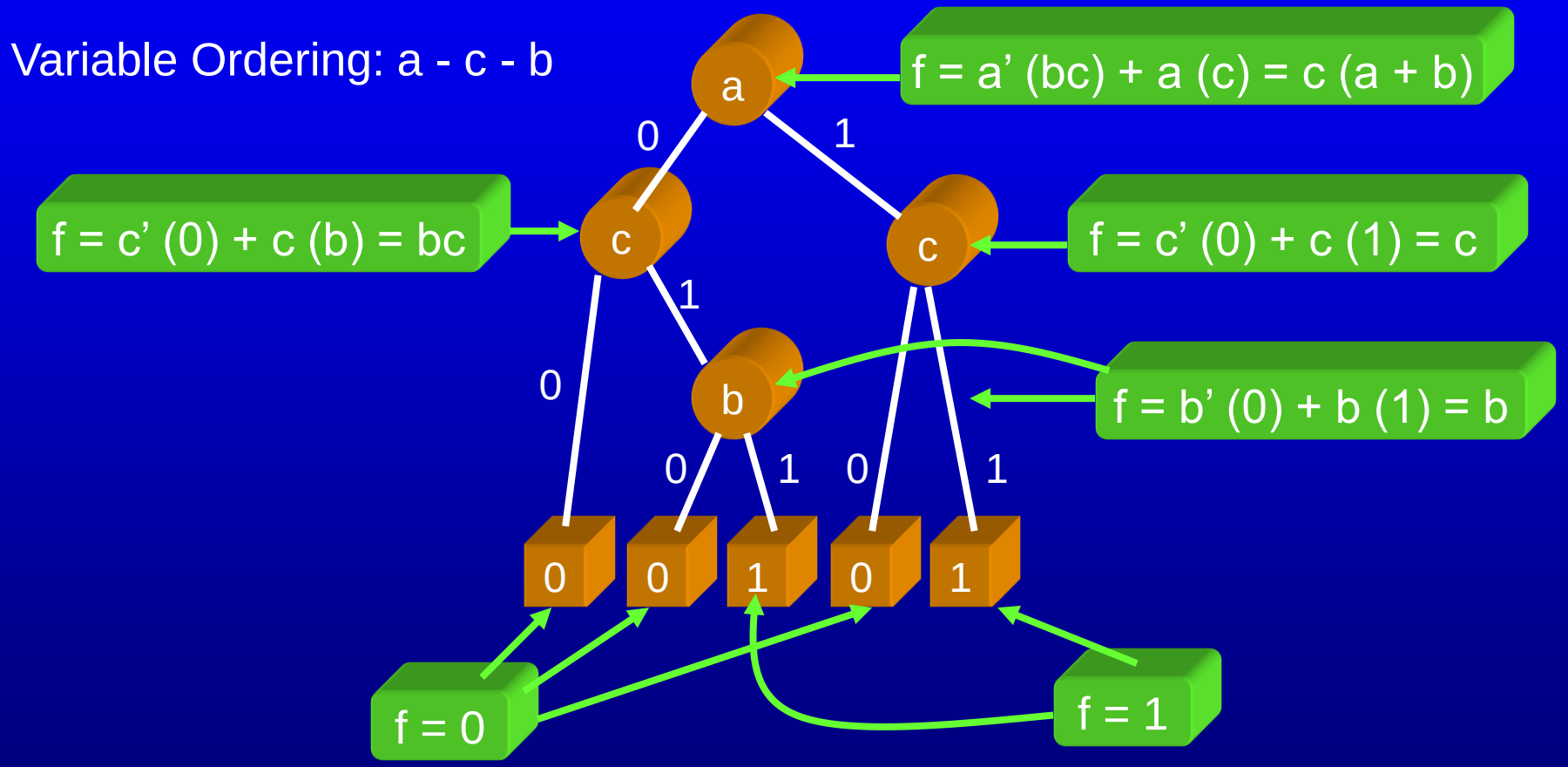


OBDD Example

Variable Ordering: a - b - c



OBDD Example with Different Ordering

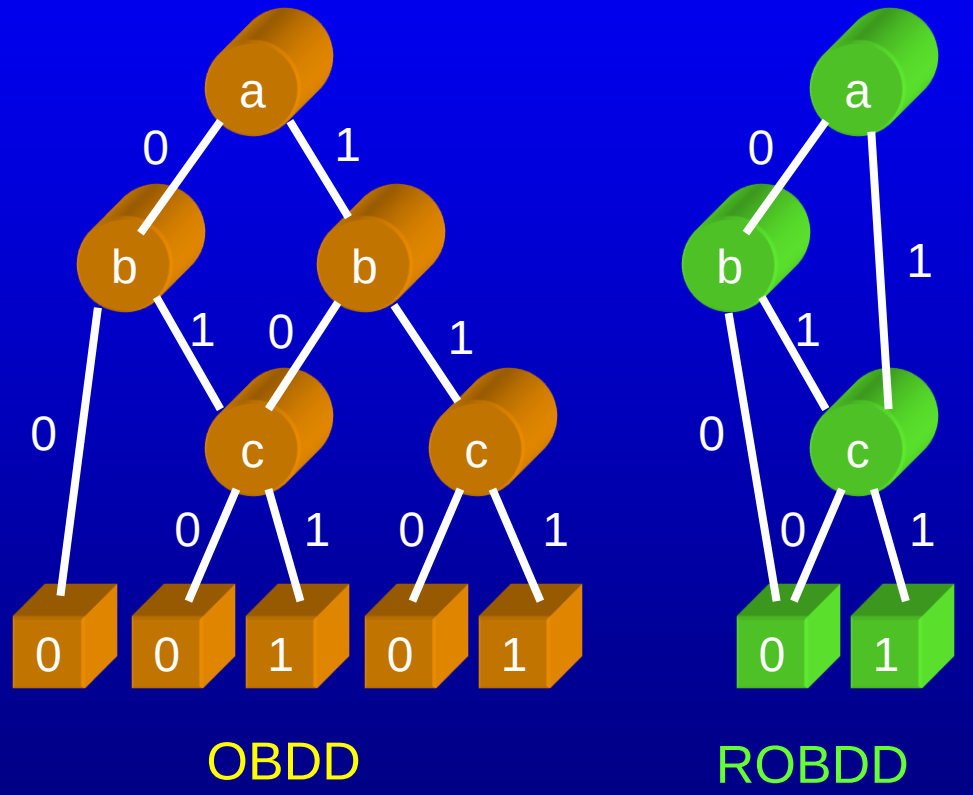


Isomorphism of OBDDs

- Isomorphic OBDDs
  - 1-1 mapping of nodes preserving indices, children, leaf values
- Isomorphic OBDDs represent same boolean function
- Same function can have different OBDDs
- Need to make OBDDs canonical
  - so that a boolean function has unique OBDD representation (for given variable ordering)
  - sufficient to remove redundancy in OBDD

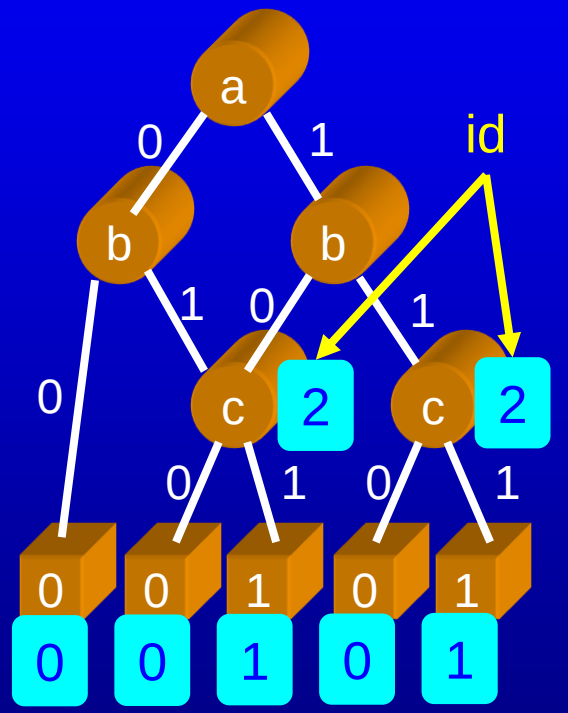
Reduced OBDD (ROBDD)

- An OBDD is reduced (ROBDD) if
  - no node v with low (v) = high (v)
  - no pair of nodes u, v such that sub-trees rooted at u, v are isomorphic
- ROBDDs are canonical
  - proof omitted



Constructing the ROBDD from OBDD

- Start from OBDD bottom up (leaf first)
- Apply label id (v) on each node v
- Resulting ROBDD will have a subset of original nodes, each with a distinct id (v)
  - if two OBDD nodes represent the same function, they will have identical labels



Removing Redundancy

- Assume all nodes with index > i are already labelled
- Consider set of nodes V with index i
- Redundancy:
  - if id (low(v)) = id (high(v)), then v is redundant
    - set id (v) = id(low (v))
  - if $\exists$ x, y $\in$ V, with
    - id (low (x)) = id (low (y))
    - id (high (x)) = id (high (y))

    then graphs rooted at x, y are isomorphic
    - set id (y) = id (x)

f(v) = v' (g) + v (g) = g

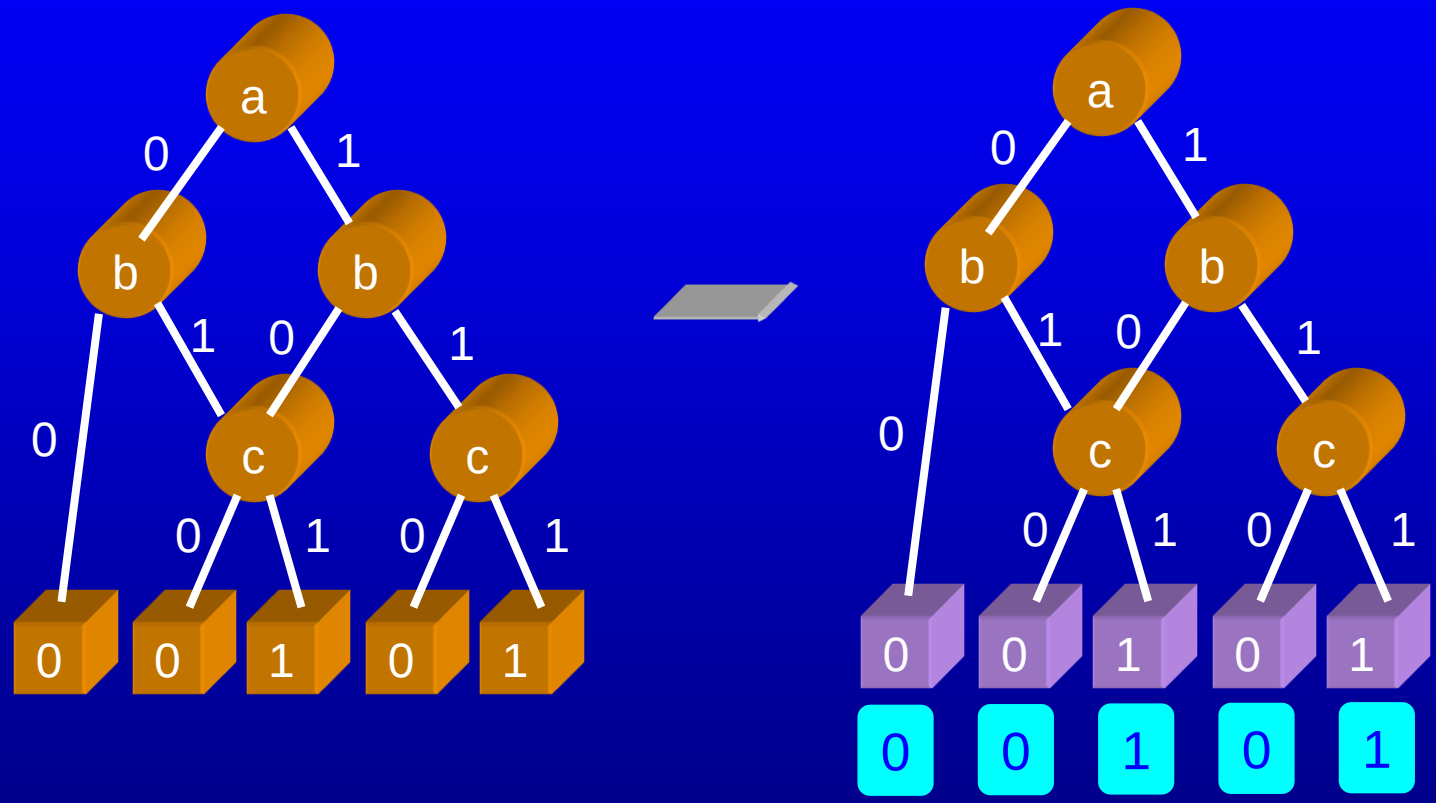
f (x) = v' (s) + v (t) = f (y)

Redundancy Removal Example

index = 1

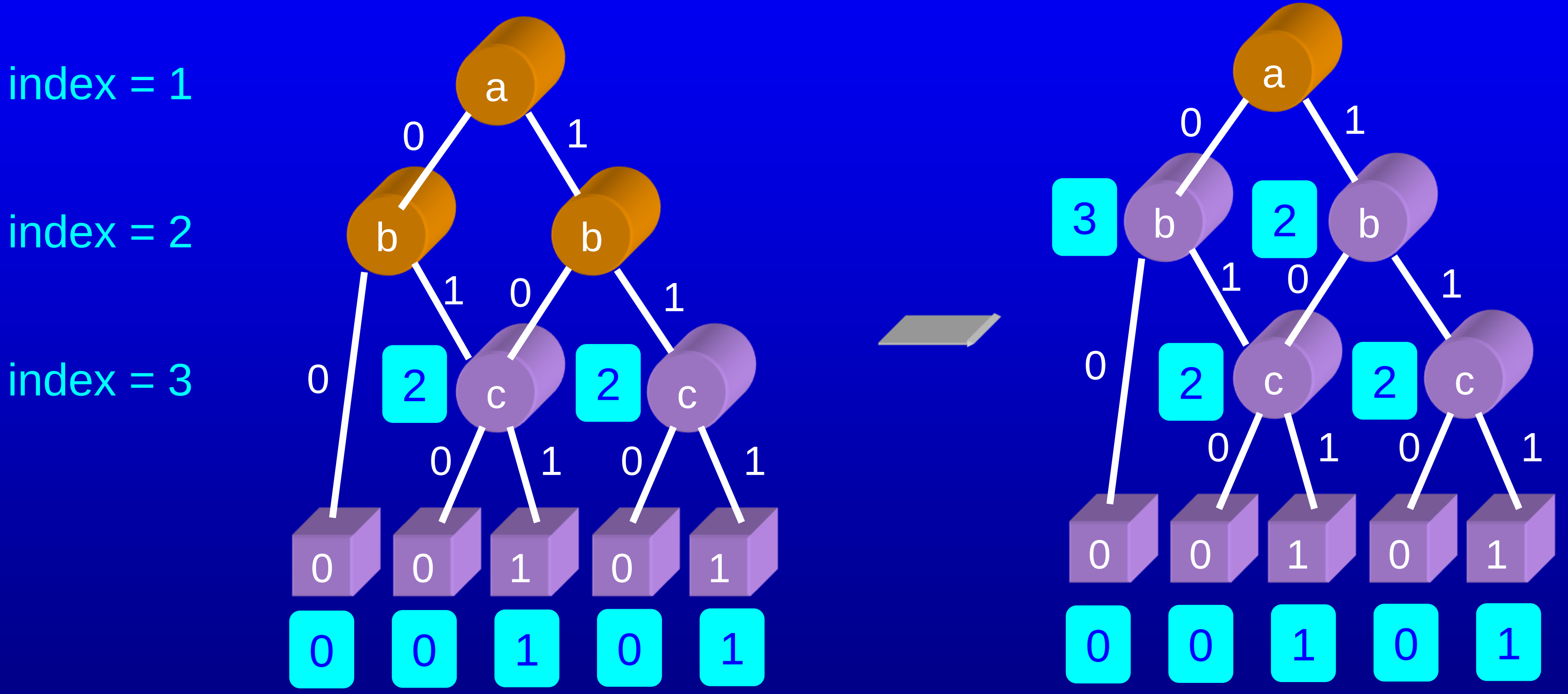
index = 2

index = 3



id represents id of node

Redundancy Removal Example

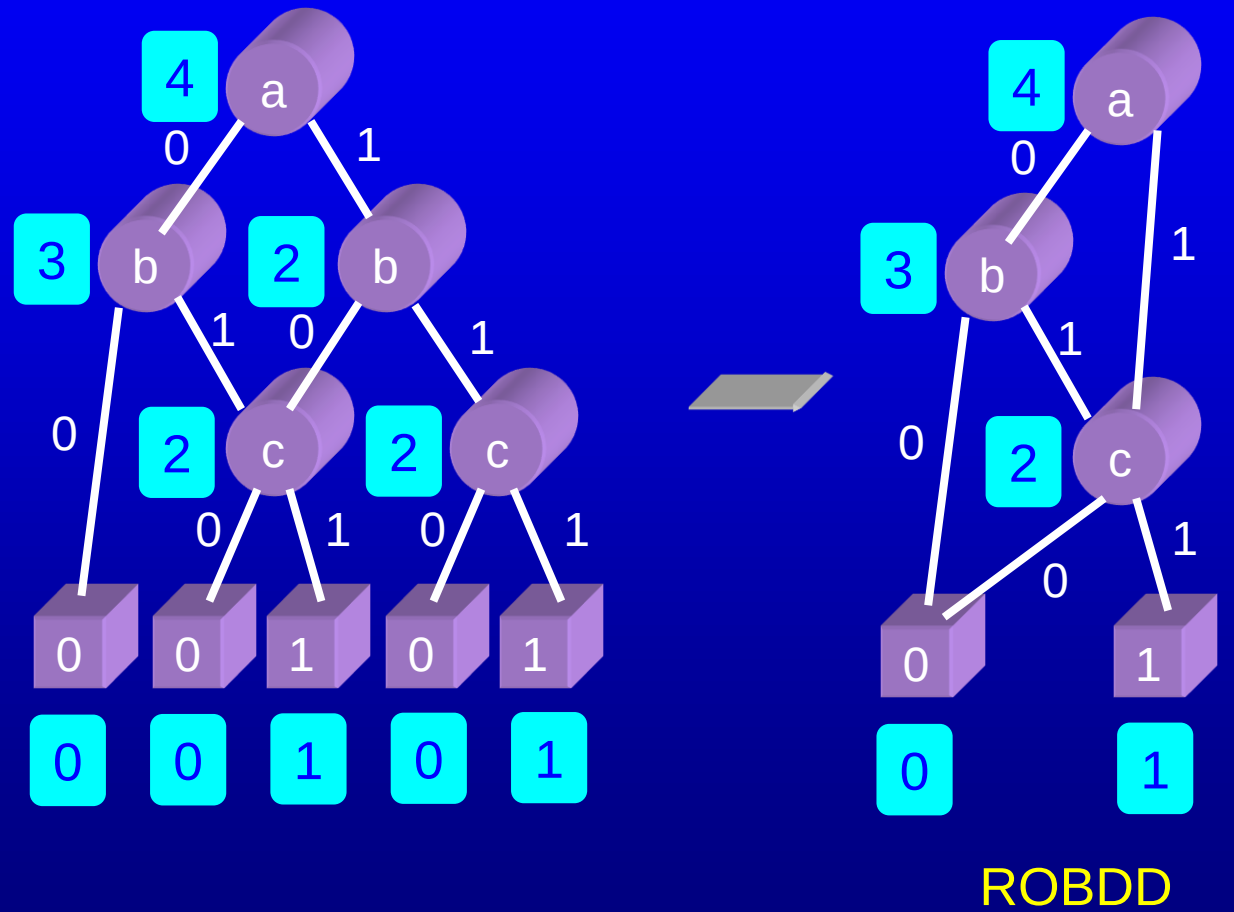


Redundancy Removal Example

index = 1

index = 2

index = 3



ROBDD

ROBDD Construction

- Build ROBDD directly from Boolean function
  - NOT generate OBDD first and then reduce
- Maintain Hash Table (called Unique Table) of triples (variable, Lid, Rid)
  - one entry for every node in ROBDD
  - when creating a new ROBDD node, first check if it already exists in Unique Table
  - (almost) constant time lookup and node creation in Unique Table

ROBDD Construction Algorithm

```
node ROBDD_build (Expr f, int index)
{
  node L, R;

  if (f == '0')
    return v0;
  else if (f == '1')
    return v1;
  else {
    p = var (index);
    R = ROBDD_build (fp, index + 1);
    L = ROBDD_build (fp', index + 1);
    if (R == L) return R;
    else return UniqueTable (p, R, L);
  }
}
```

Boolean Expression in some format. Simple manipulation routines available (e.g., finding co-factor)

v_0,v_1 pre-inserted into UniqueTable

Create if node doesn't exist

ROBDD Construction Example

ROBDD_build (ac+bc, 1)

ROBDD_build (c+bc, 2)

ROBDD_build (c, 3)

ROBDD_build (‘1’, 4)

V_1

ROBDD_build (‘0’, 4)

V_0

New node in UniqueTable

V_2

ROBDD_build (c, 3)

ROBDD_build (‘1’, 4)

V_1

ROBDD_build (‘0’, 4)

V_0

(c, v1, v0) = v2 already exists in UniqueTable

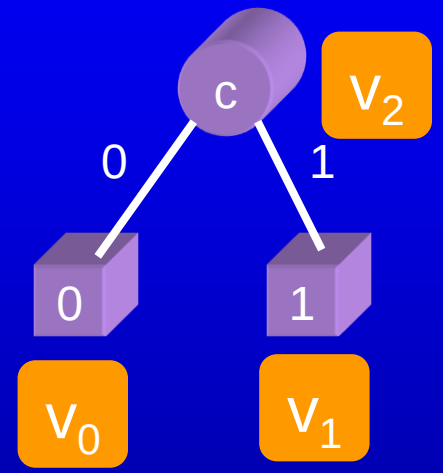
V_2

L == R

V_2

…

Order: a-b-c



ROBDD

ROBDD Construction Example

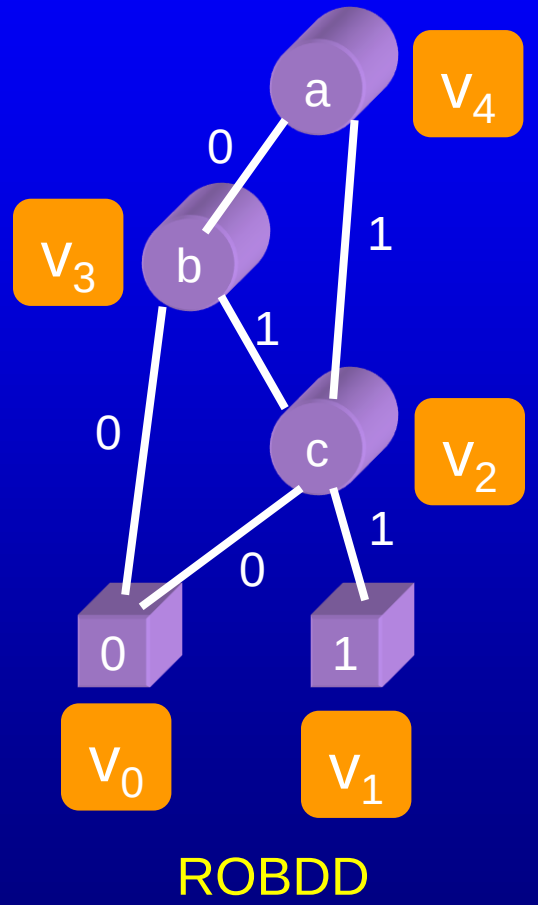
ROBDD_build (ac+bc, 1)
- ROBDD_build (c+bc, 2)
- ...
- v_2

ROBDD_build (bc, 2)
- ROBDD_build (c, 3)
  - ROBDD_build ('1', 4)
  - v_1
  - ROBDD_build ('0', 4)
  - v_0
- v_2
- ROBDD_build ('0', 3)
- v_0
- v_3

v_4

New node v3 = (b, v0, v2)

New node v4 = (a, v3, v2)



ROBDD

ROBDD Construction: Complexity

- Space optimised, but exponential time
- Solution: use another Hash Table
  - Boolean expression is the ‘key’
  - If node exists for an expression, use it instead of recursing further

Application of ROBDDs: Verification

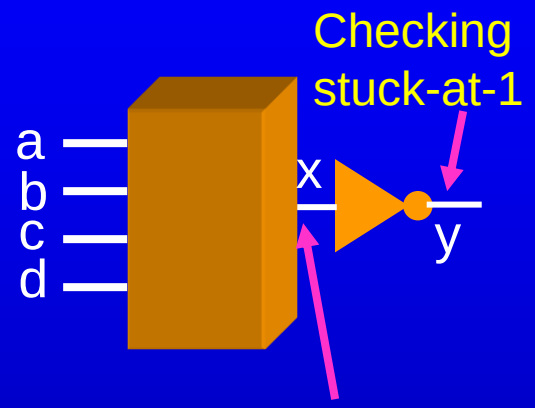
- Given two different boolean functions
  - check if they are identical
- Application: verify result of synthesis
  - does the final implementation satisfy specification?
    - is the synthesis tool correct?
    - are manual modifications made to netlist valid?
    - do optimisation passes preserve behaviour?

Checking ROBDDs for Equivalence

- Build both functions on same ROBDD structure (same UniqueTable)
  - multiple-output function
  - use same variable ordering
- Finally, just a pointer check
  - if equivalent, both functions should point to same UniqueTable entry
    - because of canonicity property

Applying ROBDDs to Testing

- Boolean satisfiability problem:
  - find an assignment of 0's and 1's to a group of variables such that a given expression is TRUE
  - NP-Complete problem
- Application in Testing
  - To determine if inverter output is stuck-at-1, set primary inputs such that inverter input is 1

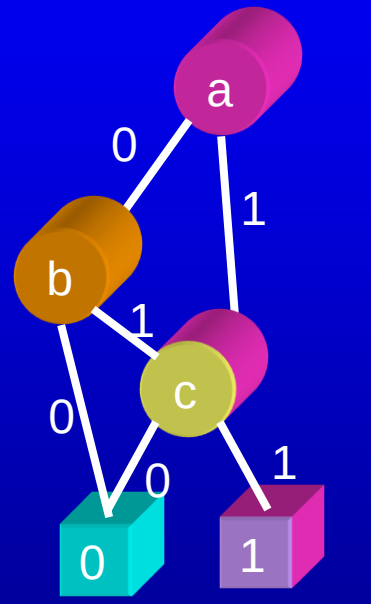


Satisfiability problem:
Can we assign values to a,b,c,d such that x is 1?

Can ROBDDs help?

- For what variable values is the function TRUE?

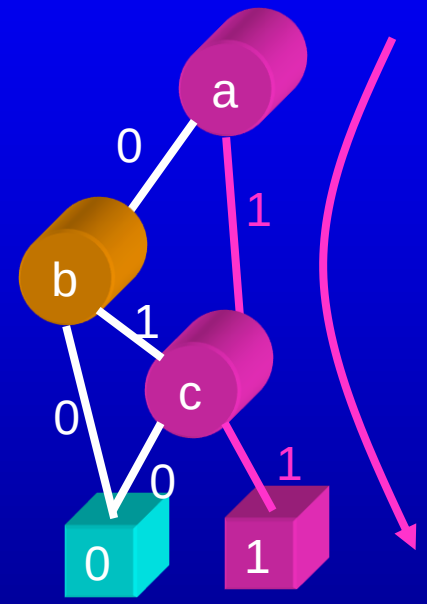
Function: ac+bc



ROBDDs in Testing

- Solving Satisfiability using ROBDDs
  - Does there exist a path in ROBDD from root to the '1' leaf node?
  - If yes, branches taken correspond to assignment of value to variables
  - If no, then no solution

Function: ac+bc



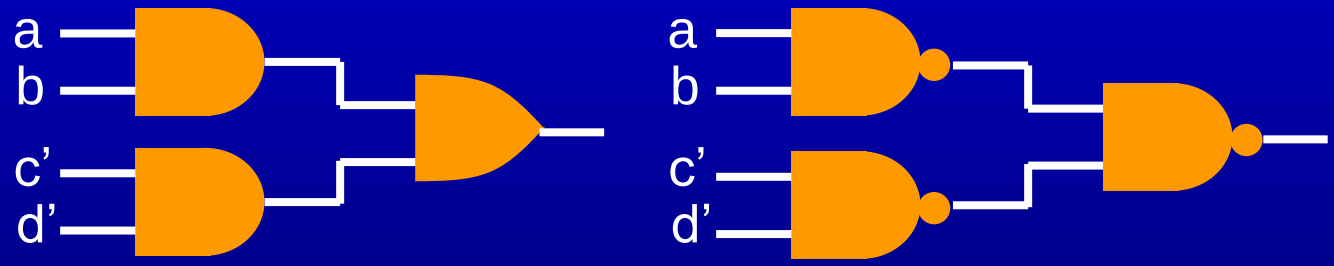
Selected path corresponds to variable assignment:
a = 1, c = 1

Logic Synthesis:
Two-Level Logic Optimisation

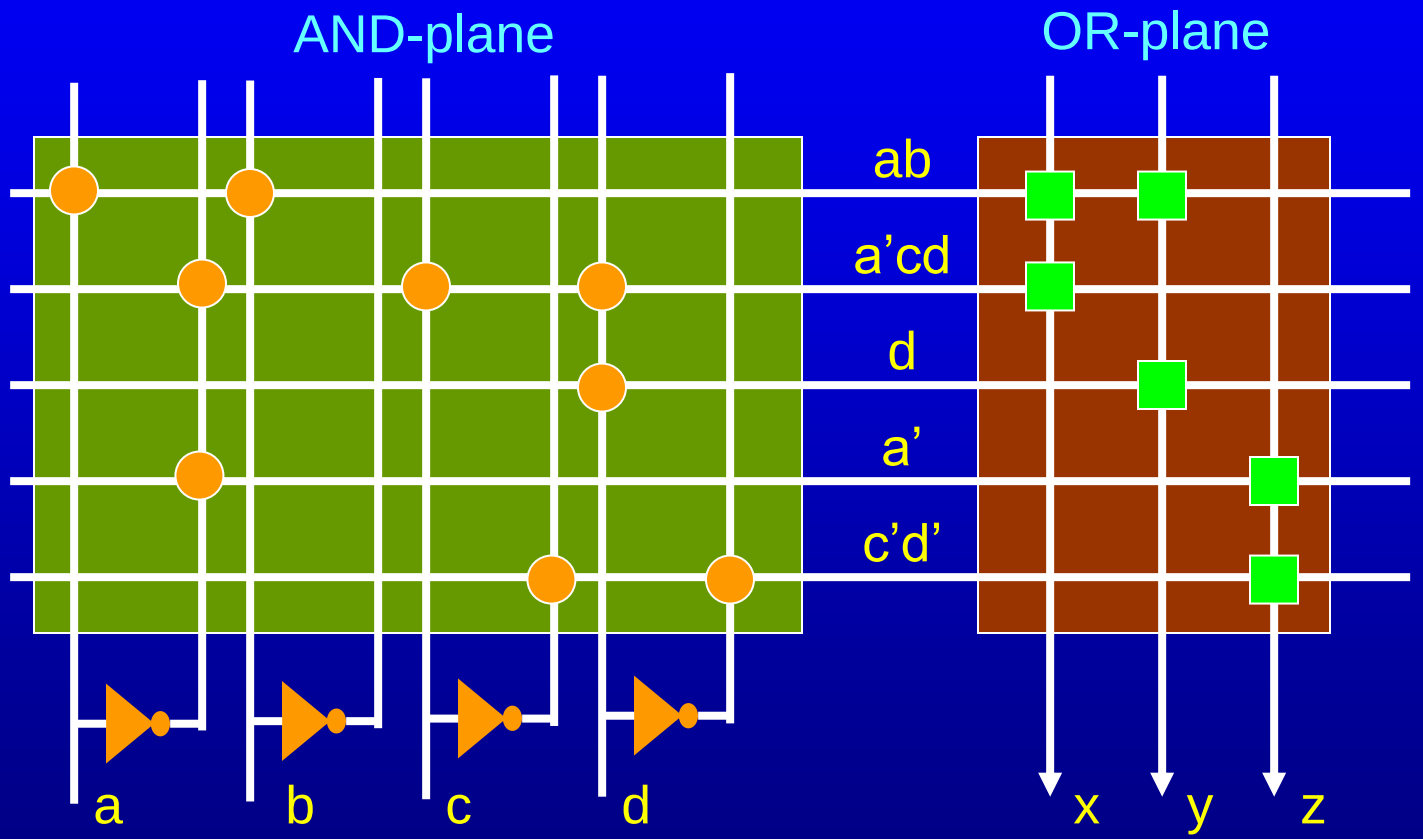
Preeti Ranjan Panda
Department of Computer Science and Engineering
Indian Institute of Technology Delhi

2-level Logic Minimisation

- AND-OR form
  - Sum of products (y = ab + c'd')
- Suitable for PLA implementation



Programmable Logic Array (PLA)

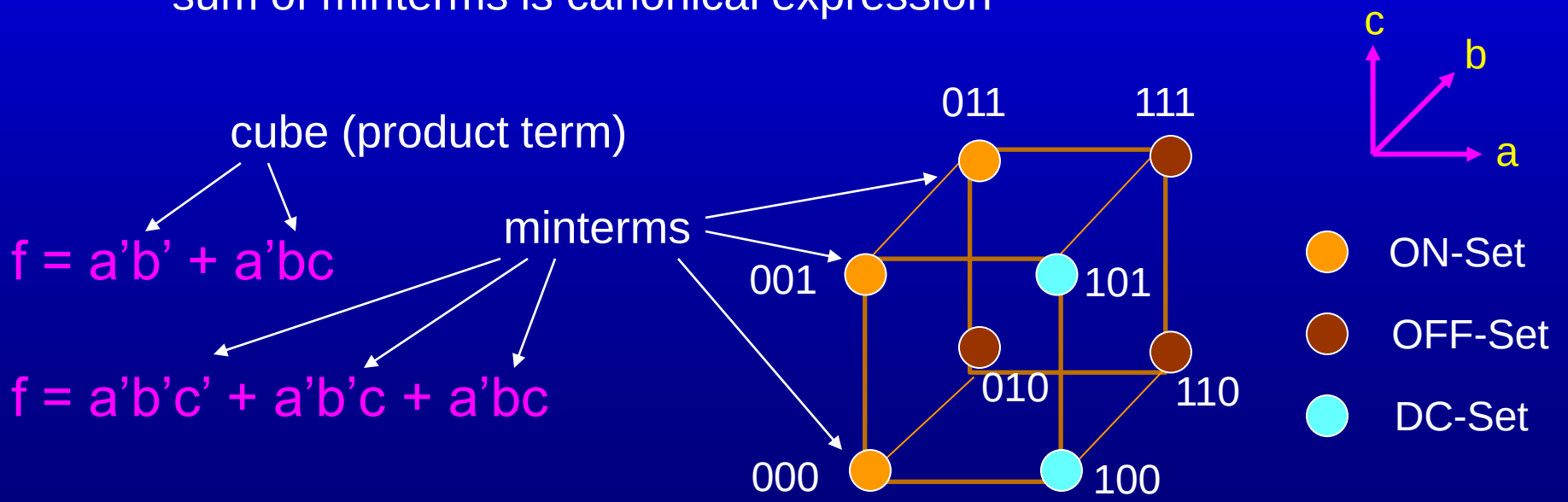


Simplified View

x = ab + a'cd
y = ab + d
z = a' + c'd'

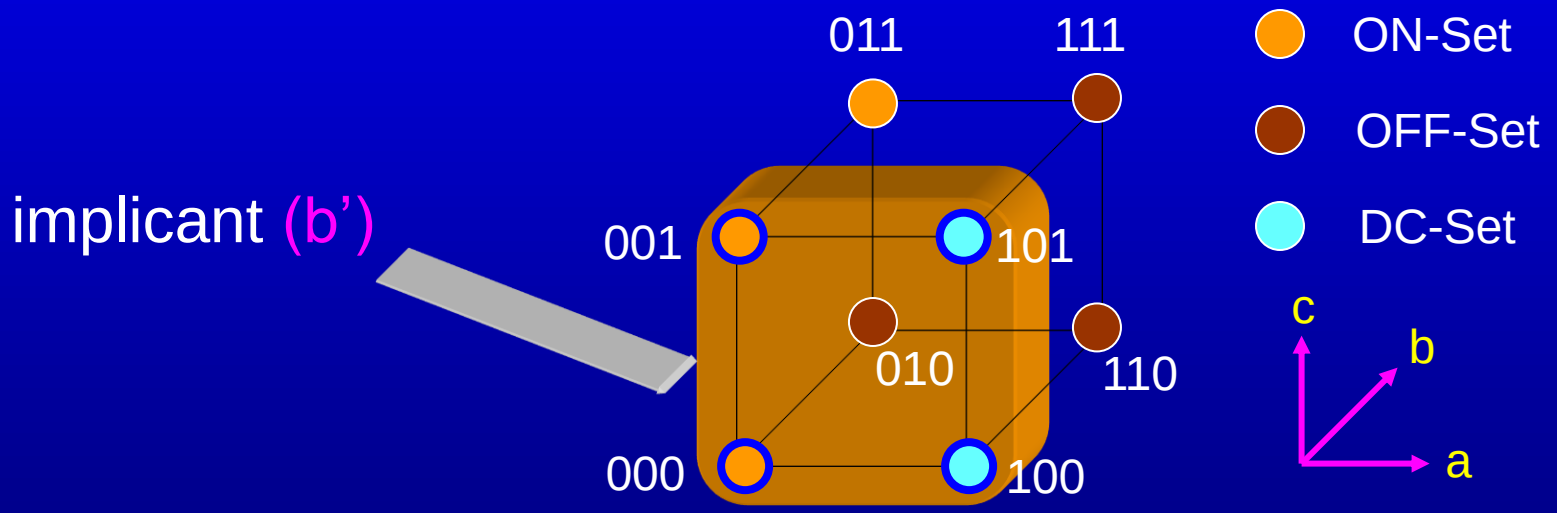
Terminology: literal, cube, minterm

- literal - a boolean variable (a) or its complement (a')
- product term or cube - product of literals (a, a'bc)
- minterm - a product of n literals for a function with n inputs
  - denotes single point in boolean space
  - sum of minterms is canonical expression



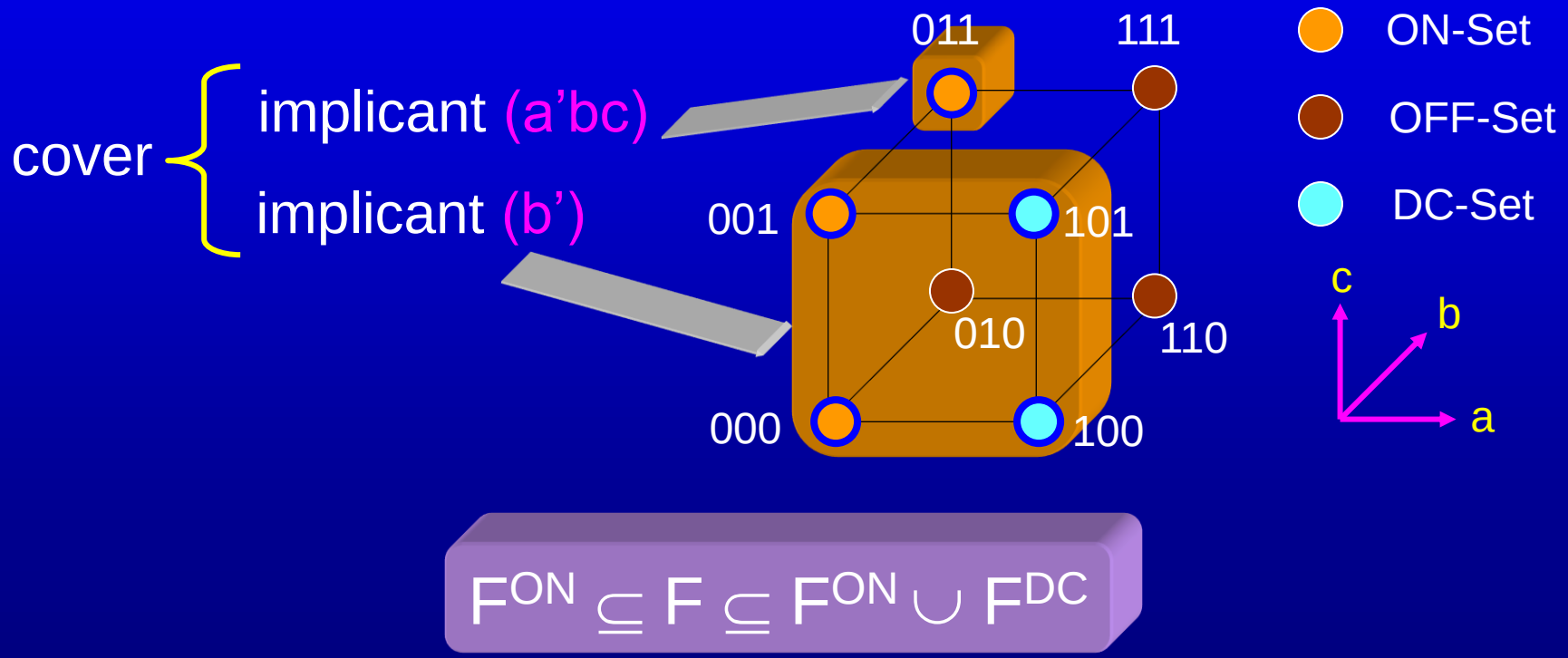
Terminology: implicant

- implicant - a cube whose points are all in the ON-set or DC-set of function



Terminology: cover

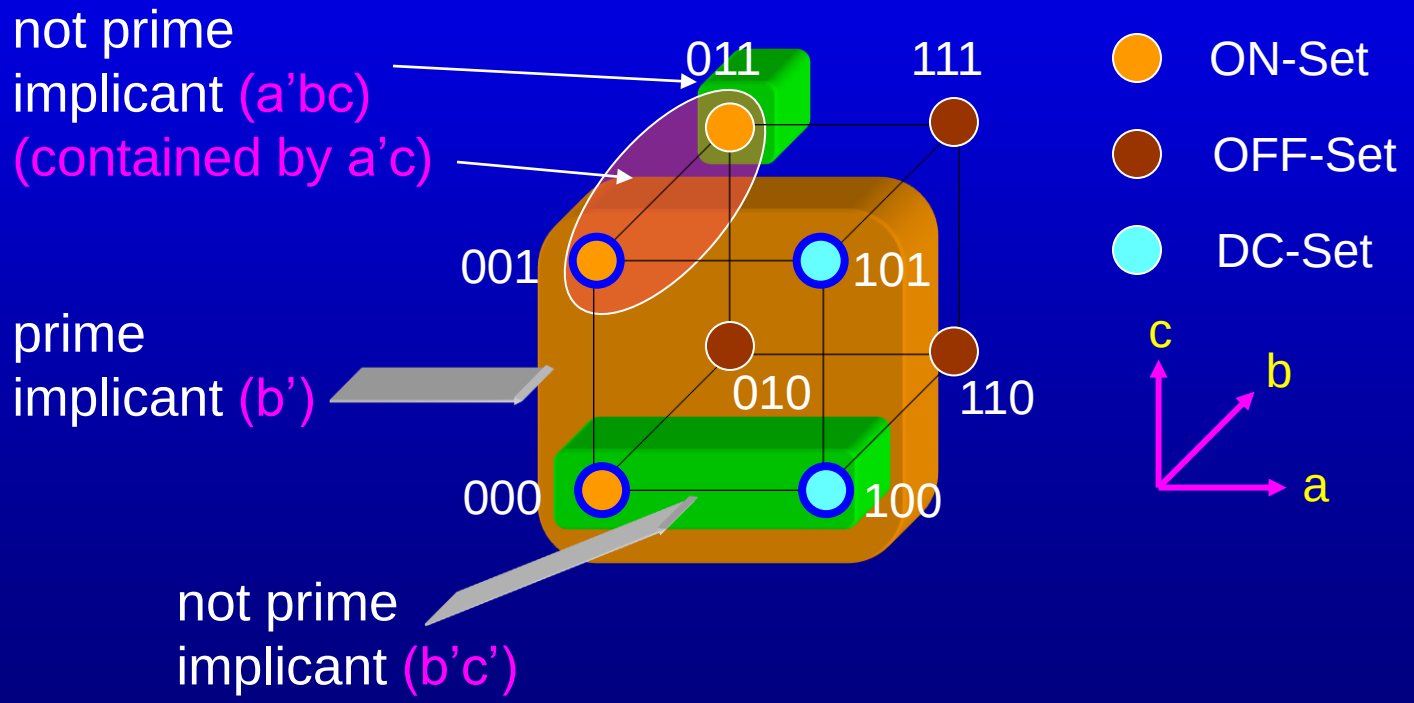
- cover - a set of implicants that includes all minterms of function



$$F^{ON} \subseteq F \subseteq F^{ON} \cup F^{DC}$$

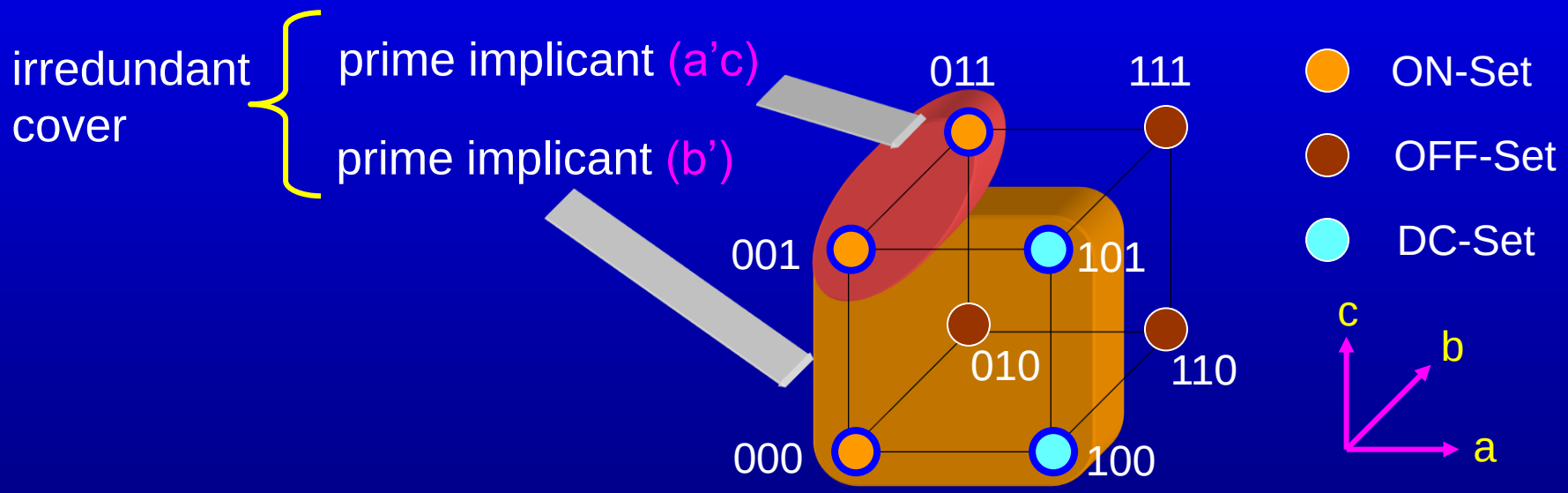
Terminology: prime implicant

- prime implicant - not contained in (not covered by) any other implicant



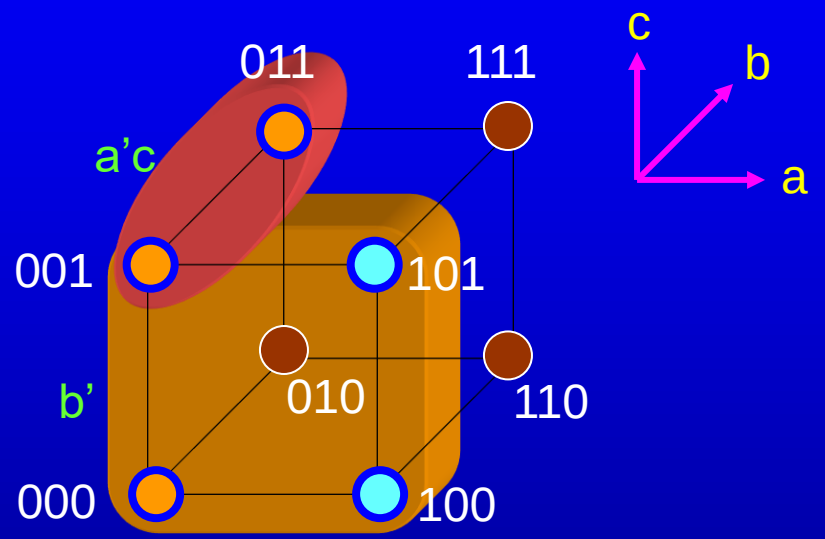
Terminology: irredundant cover

- irredundant cover - from which we cannot delete one implicant and still cover the function



Objective of 2-level Logic Minimisation

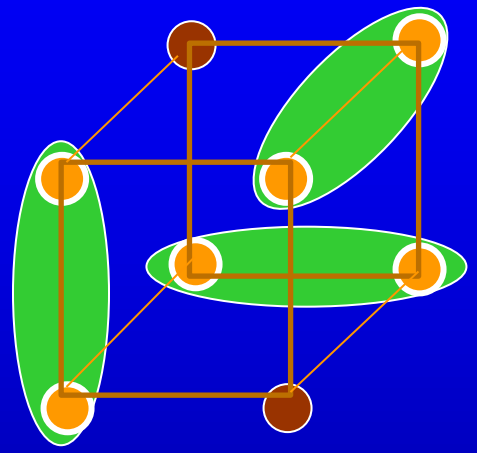
- Exact Minimisation: Finding the minimum cover
  - minimum #product terms
    - minimises #rows in PLA (i.e., reduces area)
  - minimum #literals
    - reduces #connections in PLA (i.e., reduces delay)
- Heuristic Minimisation: Finding an Irredundant cover (minimal cover)
  - May not be minimum cover
  - "Local Minimum"



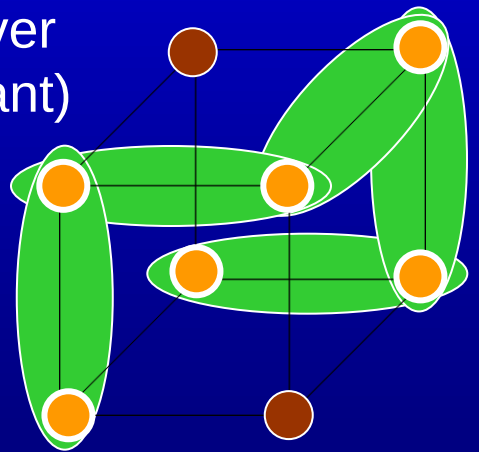
irredundant cover here is also minimum cover = {a'c, b'}

Irredundant vs. Minimum Cover

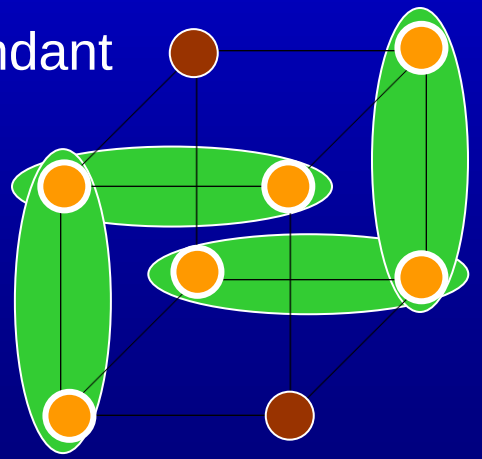
Minimum Cover



Initial Cover
(Redundant)



Irredundant
Cover



Exact 2-level Minimisation : Quine-McKluskey Algorithm

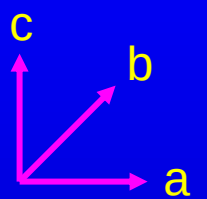
- Generate Prime Implicants
  - Quine's theorem:
    - There exists a minimum cover consisting of only prime implicants (Proof?)
- Choose a subset that covers function
  - includes all minterms
  - minimise the number of primes

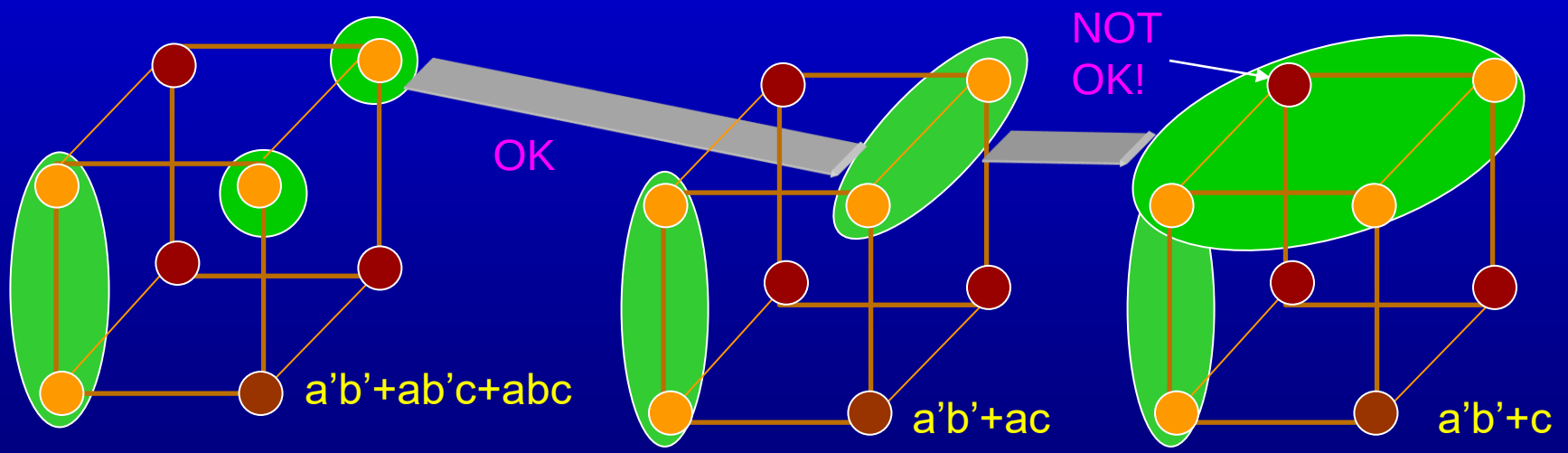
Heuristic 2-level Logic Minimisation

- Exact logic minimisation takes too much time
  - number of prime implicants may be large
- Close to minimum may be good enough
  - need for heuristic minimisation
    - start with some cover and improve it

Heuristic Operations: EXPAND

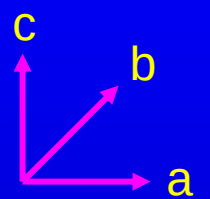
- Increase the size of each implicant
  - so that others can be covered (and deleted)
  - change one of the literals to Don't Care
  - verify validity (should not intersect OFF-set)

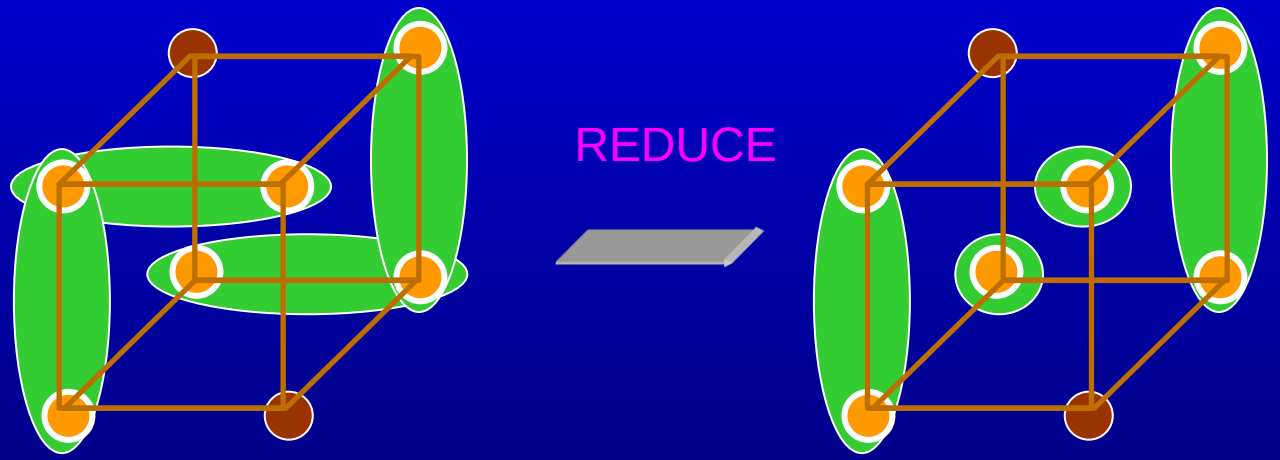




Heuristic Operations: REDUCE

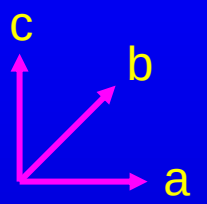
- REDUCE the size of implicant
  - subsequent EXPAND may lead to better cover
  - verify validity (function still covered)

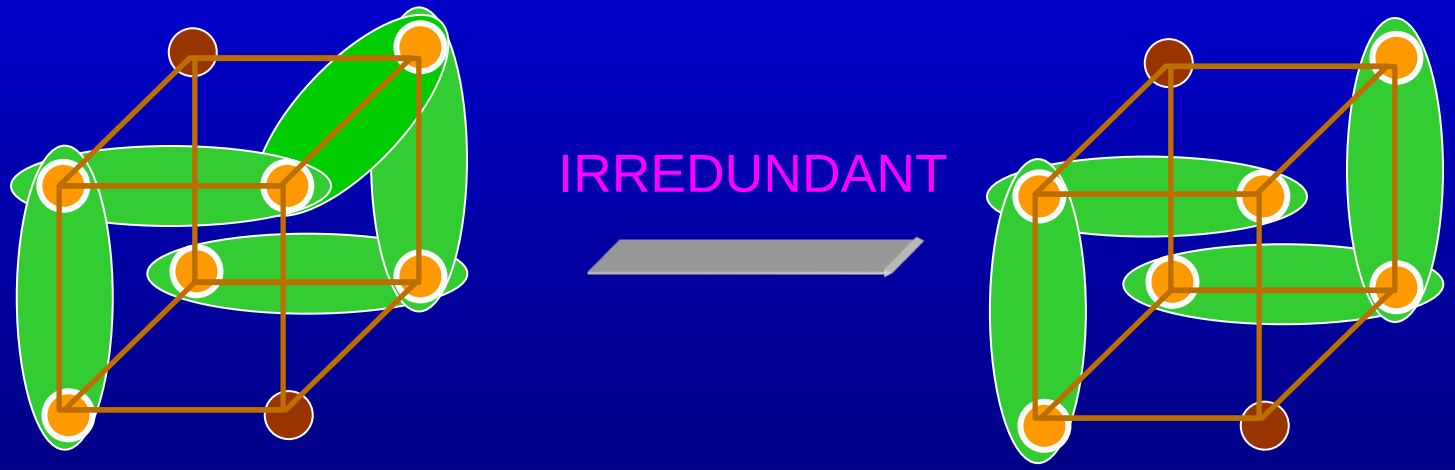




Heuristic Operations: IRREDUNDANT

- Delete redundant implicants from cover
  - check if cover is no longer valid on deletion





Example Optimisation Program: ESPRESSO

...

repeat {

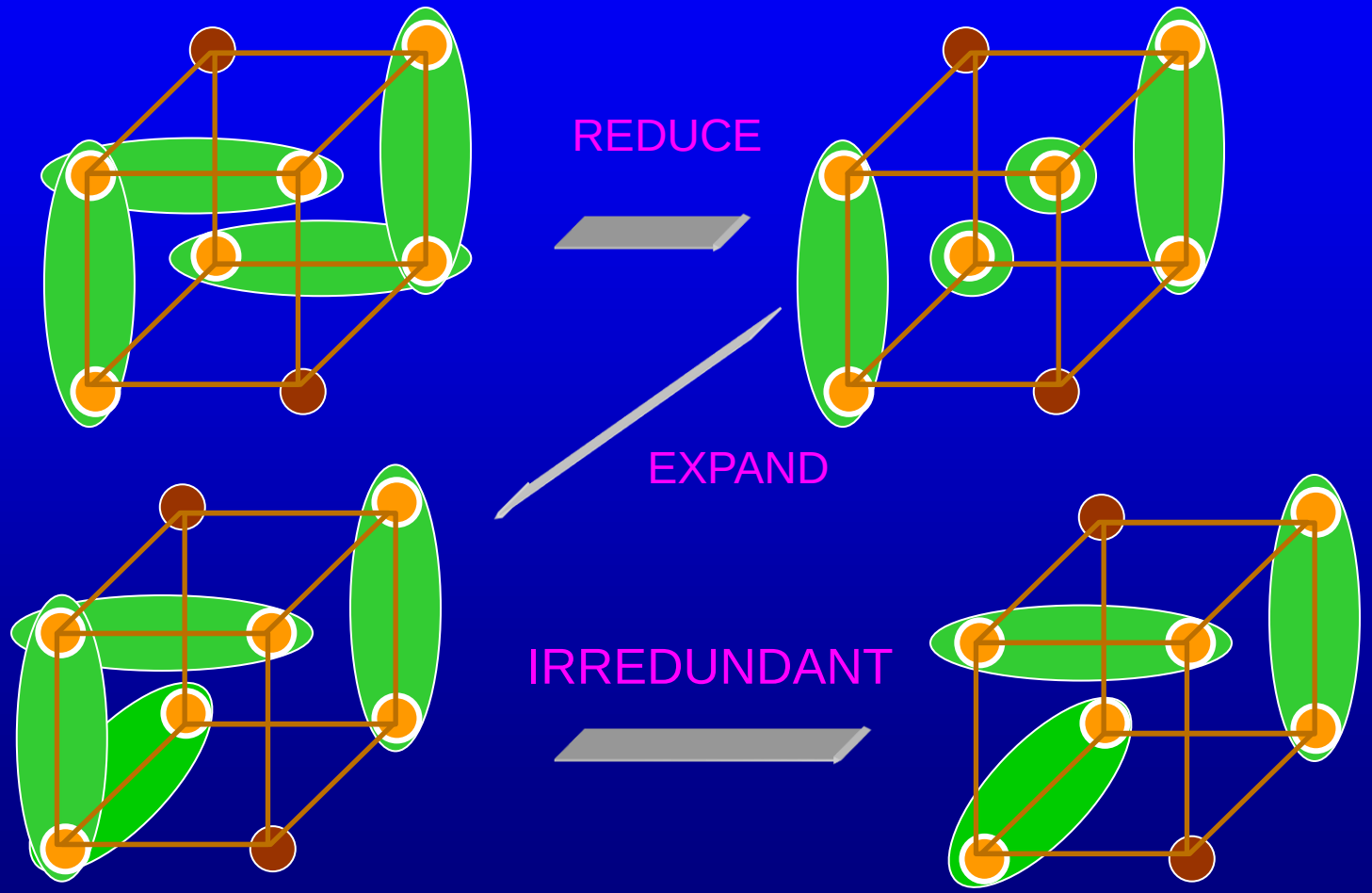
- REDUCE
- EXPAND
- IRREDUNDANT

} until no improvement

...

Example ESPRESSO Loop

Escaping from Local Minimum



Positional Cube Notation

- Binary encoding of implicants
- 2-bit encoding of boolean values
- ϕ means void
  - implicant containing ϕ should be deleted
- Boolean function represented as table
  - one row for each cube

Value	Encoding
0	10
1	01
'-'	11
ϕ	00

f = ab + a'bc + abc'

01	01	11	ab
10	01	01	a'bc
01	01	10	abc'
a	b	c	

Operations on Implicants

- Intersection of 2 implicants
  - largest cube contained in both
  - bitwise product
- Supercube of 2 implicants
  - smallest cube containing both
  - bitwise sum
- Distance between 2 implicants
  - # of ϕ's in intersection
  - if distance=0, then implicants intersect, else disjoint

abc	01	01	01
ab	01	01	11
a'bc'	10	01	10
bc'	11	01	10

$abc \cap ab = 01\ 01\ 01 = abc$

$abc \cap a'bc' = 00\ 01\ 00 = void$

supercube (abc, ab) = 01 01 11 = ab

supercube (abc, a'bc') = 11 01 11 = b

Dist (abc, ab) = 0

Dist (abc, a'bc') = 2

EXPAND Operator

- Increase size of implicants
  - so that other implicants are covered (and subsequently deleted)
- Maximally expanded implicants are Prime Implicants
- EXPAND results in cover that is minimal with respect to single-implicant containment
- Basic operation: Raise 0 to 1 in implicant
  - e.g. “01 10 10” (ab’c’) to “01 11 10” (ac’)
  - Each raise increases size of cube by factor of 2
- Check if expanded cube is VALID
  - still an implicant of the function
    - check for possible intersection with OFF-set

Issues in EXPAND

- What order to consider the implicants?
- What order to consider ‘0’ entries within an implicant?
- Quality of results depends on order
- Heuristic:
  - first expand the implicants that are unlikely to be covered by others

Implicant Order

- Vector S = column sums
- Weight to each cube C
  - dot product C • S
- Low weight for C implies
  - relatively fewer 1's in densely populated columns
  - less likely to be covered by others
- Consider implicants in increasing weight order

X	a'b'c'	10	10	10
Y	ab'c'	01	10	10
Z	a'bc'	10	01	10
W	a'b'c	10	10	01

S = [3 1 3 1 3 1]

Weight (X) = [1 0 1 0 1 0] • [3 1 3 1 3 1] = 9
Weight (Y) = [0 1 1 0 1 0] • [3 1 3 1 3 1] = 7
Weight (Z) = [1 0 0 1 1 0] • [3 1 3 1 3 1] = 7
Weight (W) = [1 0 1 0 0 1] • [3 1 3 1 3 1] = 7

Order: Y, Z, W, X

Expanding an Implicant

F^{ON} =

X	a'b'c'	10	10	10
Y	ab'c'	01	10	10
Z	a'bc'	10	01	10
W	a'b'c	10	10	01

F^{DC} =

abc'	01	01	10

F^{OFF} =

bc	11	01	01
ac	01	11	01

Original Y: [01 10 10] (Y = ab'c')
Expand Y: [11 10 10] (Y' = b'c') VALID ($Y' \cap F^{OFF} = \phi$)
Expand Y': [11 11 10] (Y'' = c') VALID ($Y' \cap F^{OFF} = \phi$)
Expand Y'': [11 11 11] (Y''' = '-') INVALID ($Y' \cap F^{OFF} \neq \phi$)
Expand W: [10 10 11] (W' = a'b') VALID ($W' \cap F^{OFF} = \phi$)

X and Z are covered by Y''

Expanded Function:

Y''	c'	11	11	10
W'	a'b'	10	10	11

Order of Raising 0's in Implicant

- Goal is to make the implicant prime
  - while covering maximal # of other implicants
- Keep set free of candidate entries that can be raised
  - initialized to all '0' entries
  - remove entries after they are considered
  - stop when set is empty

Implicant a'bc:
[10 01 01]

Initial free set:
{2, 3, 5}

Order of Raising 0's in Implicant X

- Find which entries can never be raised
  - distance = 1 from F^{OFF}
  - remove from free
- Find which entries can always be raised
  - column has only 0's in F^{OFF}
  - raise and remove from free
- If supercube (X,Y) feasible ($\cap F^{OFF}=\phi$) for some Y, raise entries in X
  - select Y covering max. # of other cubes

X = [01 10 10]
F^{OFF}: K= [01 10 01]
Distance (X, K) = 1
Column 6 of X cannot be raised

X = [01 10 01]
F^{OFF}: P = [10 01 01]
Q = [01 01 01]
R = [10 10 01]
Column 5 of X can be safely raised (won't lead to extra intersections)

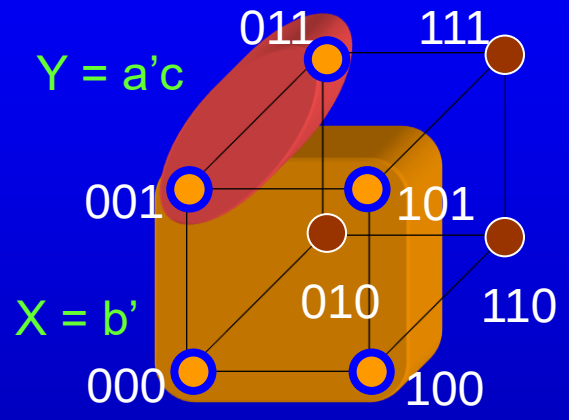
REDUCE Operator

- Convert implicant from Prime to Non-prime
- # of implicants in cover remains unchanged
- Validity check
  - (reduced implicant + other implicants) still covers function
- Find maximally reduced cube

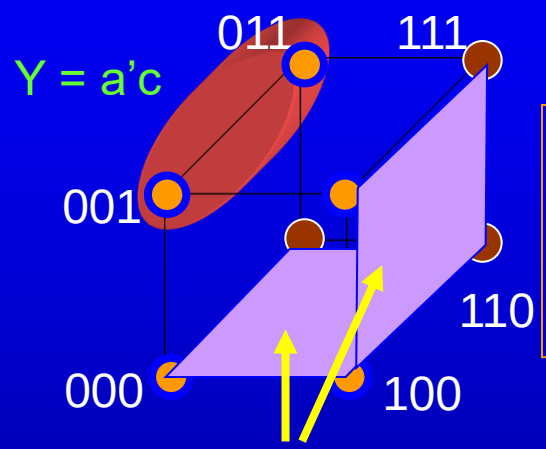
Maximally Reduced Cube

- Remove from cube X those minterms that are already covered in (F-X)
- Reduced cube = X ∩ (F-X)'
- (F-X)' might be a set of cubes
  - but we want to replace X by exactly one cube
- Reduced cube = X ∩ supercube (F-X)'
  - this is maximally reduced (proof omitted)

REDUCE Example (Try Reducing X)

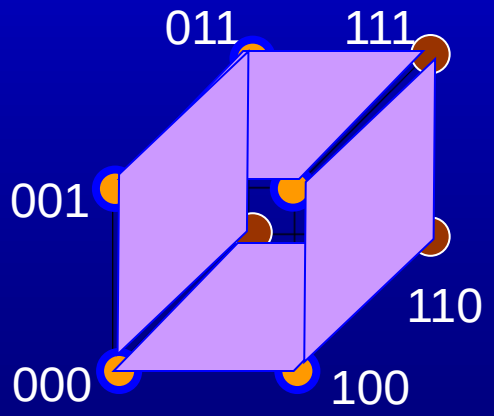


X = 11 10 11
Y = 10 11 01



(F - X)' = Y'
= (a'c)' = a+c'
= { [01 11 11],
[11 11 10] }

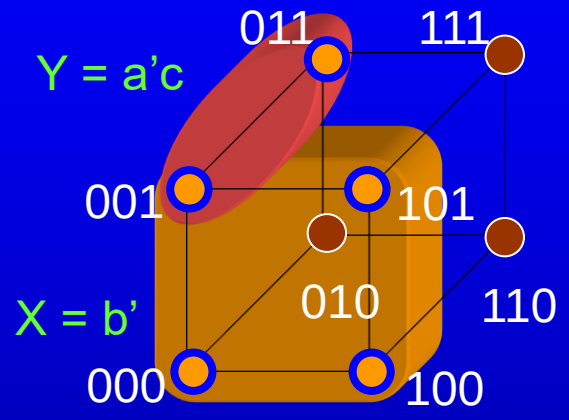
Y' = a+c' (multiple cubes)



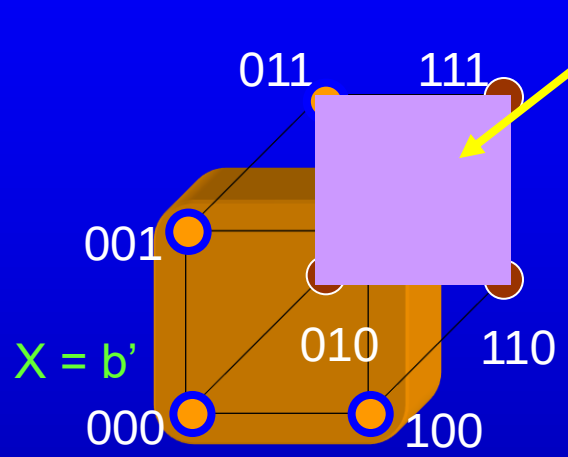
supercube (F - X)'
= [11 11 11]
(entire space)

X ∩ supercube (F - X)'
= [11 10 11] = X
X cannot be reduced further

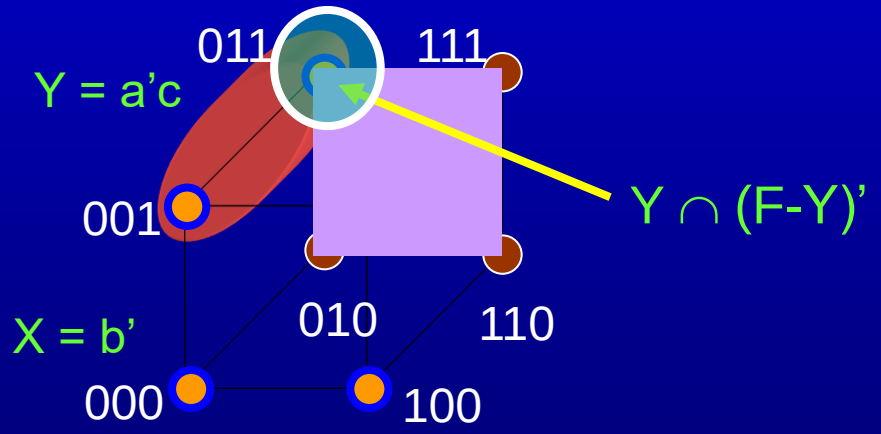
Attempt to Reduce Y



X = 11 10 11
Y = 10 11 01



(F - Y)' = X'
= (b')' = b
= [11 01 11]



supercube (F-Y)' = (F-Y)'
Y ∩ supercube (F - Y)'
= [10 01 01] = a'bc
Y can be reduced to a'bc

Multi-level Logic Optimisation

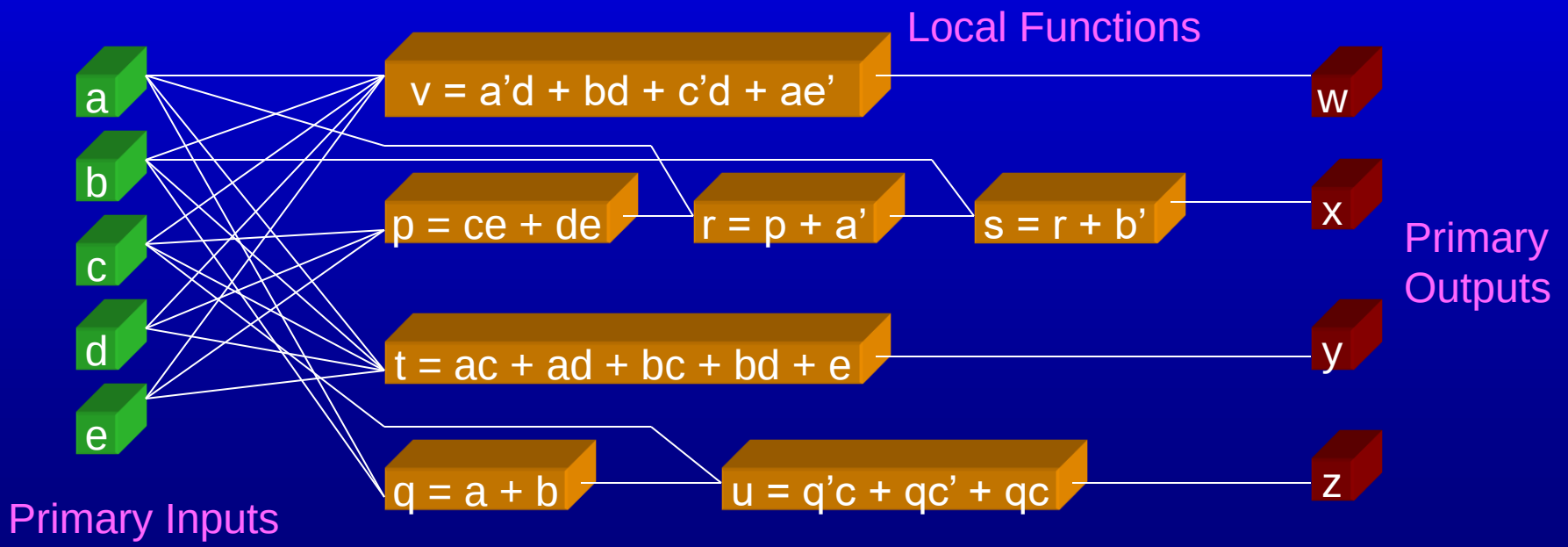
- More degrees of freedom
- Exact optimisation
  - computationally complex and impractical
- Heuristics employed
  - Phase 1 - optimise using generic gates
  - Phase 2 - technology specific mapping

Representation: Logic Networks

- Graph
- Nodes
  - Local function
    - multiple inputs, single output (y = a + bc)
  - Primary inputs
  - Primary outputs
- Edges
  - Nets
    - single source, single destination
    - if multiple dest, net represented by multiple edges

Logic Network Representation

- p = ce + de
- q = a + b
- r = p + a’
- s = r + b’
- t = ac + ad + bc + bd + e
- u = q’c + qc’ + qc
- v = a’d + bd + c’d + ae’
- w = v x = s
- y = t z = u

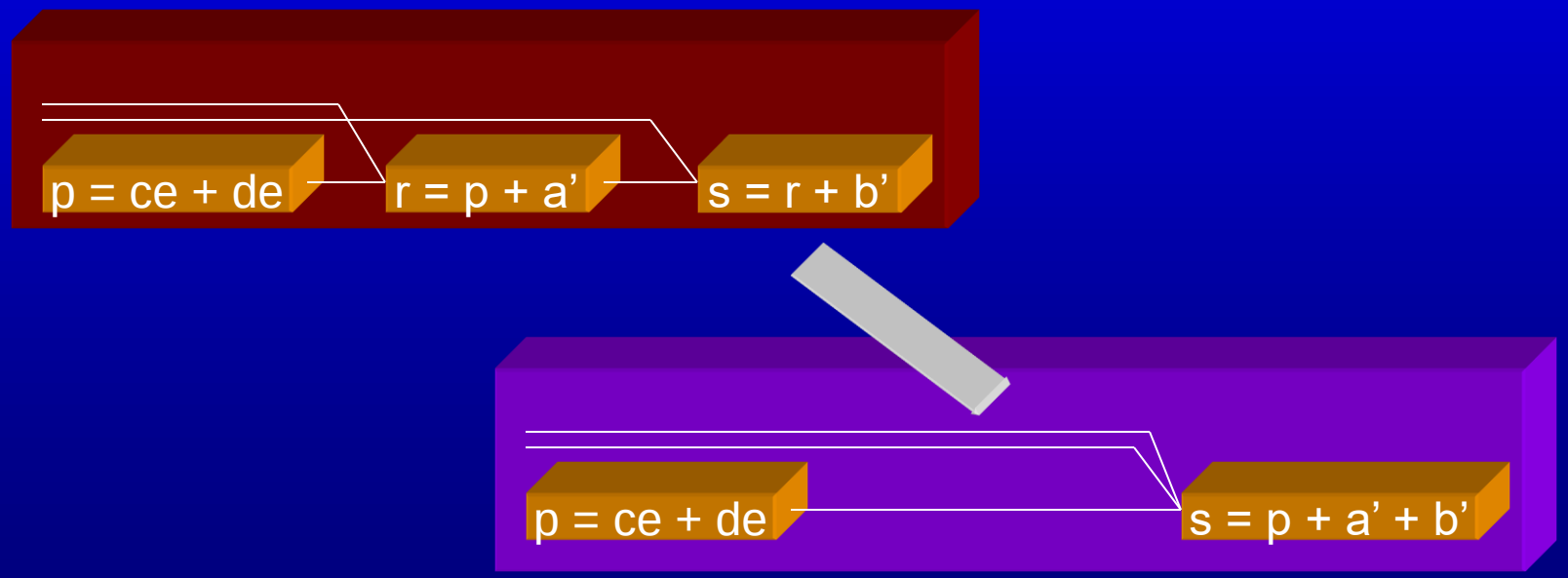


Optimisation Strategies

- Transformations
  - stepwise improvement preserving I/O behaviour
  - until no further improvement (local minimum)
- Rule-based
  - pattern matching and replacement
  - pattern pairs are stored in knowledge base
    - (common pattern, optimal pattern)
  - if subgraph matches common pattern, replace by optimal pattern

Transformations: ELIMINATION

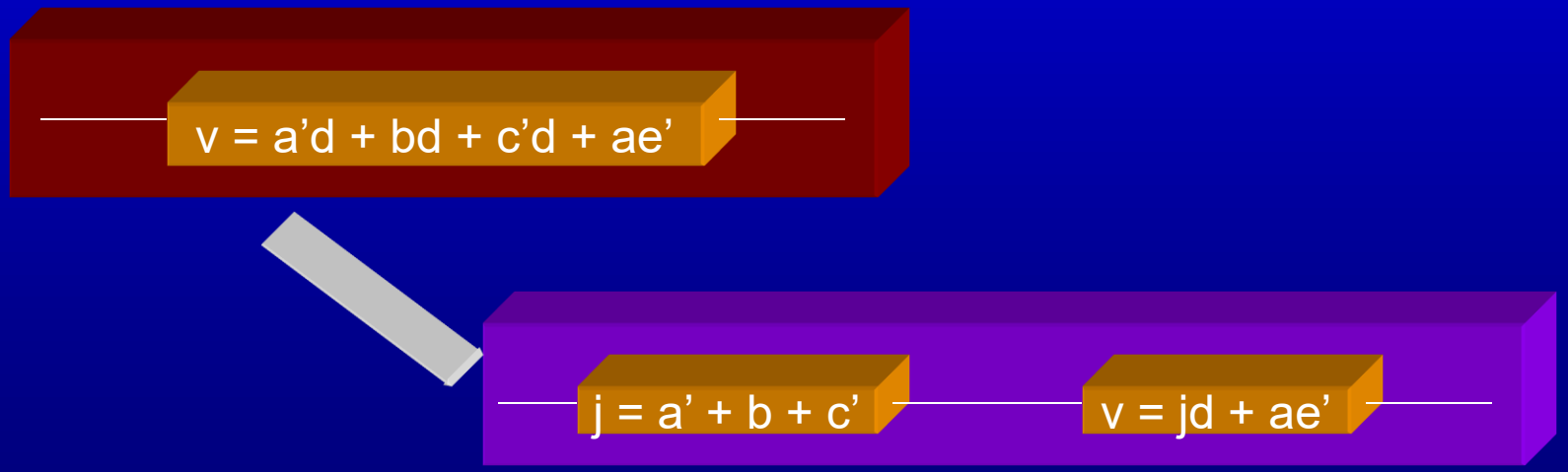
- Remove a node
- Replace its occurrences in the network by its expression
  - maybe simple local expression...optimisation opportunities if combined



Transformations: DECOMPOSITION

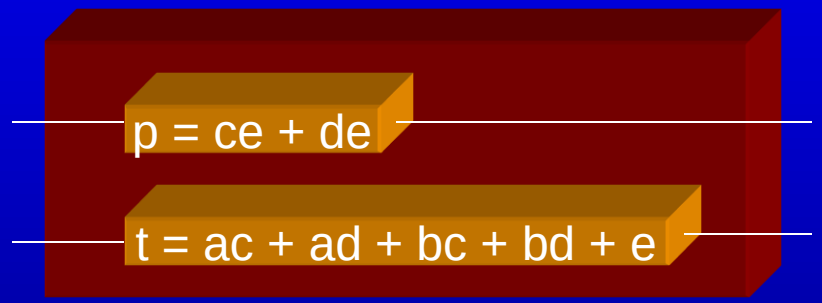
- Replace one node by two or more
- Splitting a complex function
  - may lead to common subexpression

a’d + bd + c’d = (a’ + b + c’) d

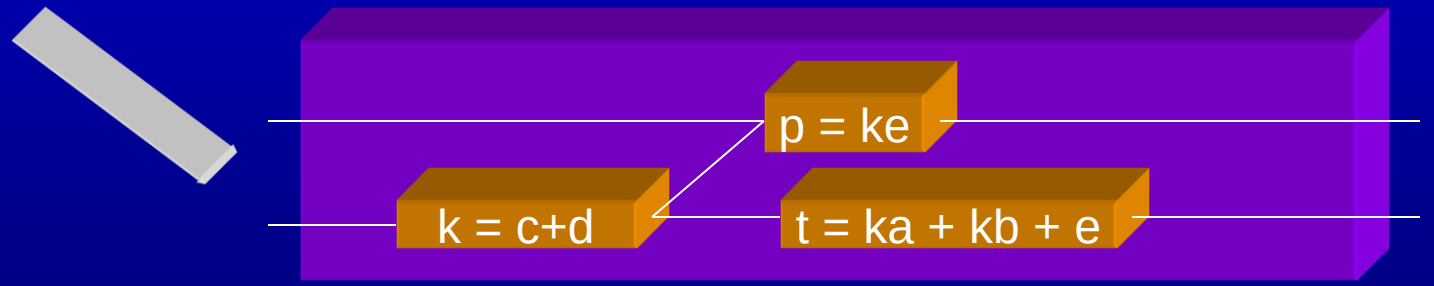


Transformations: EXTRACTION

- Common sub-expression
- Leads to simplification

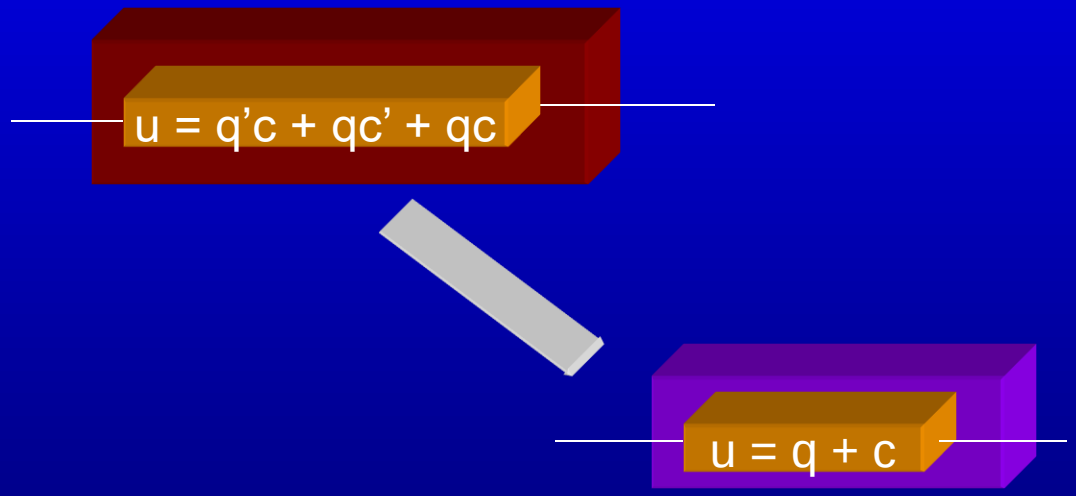


p = (c + d) e
t = (a + b)(c + d) + e



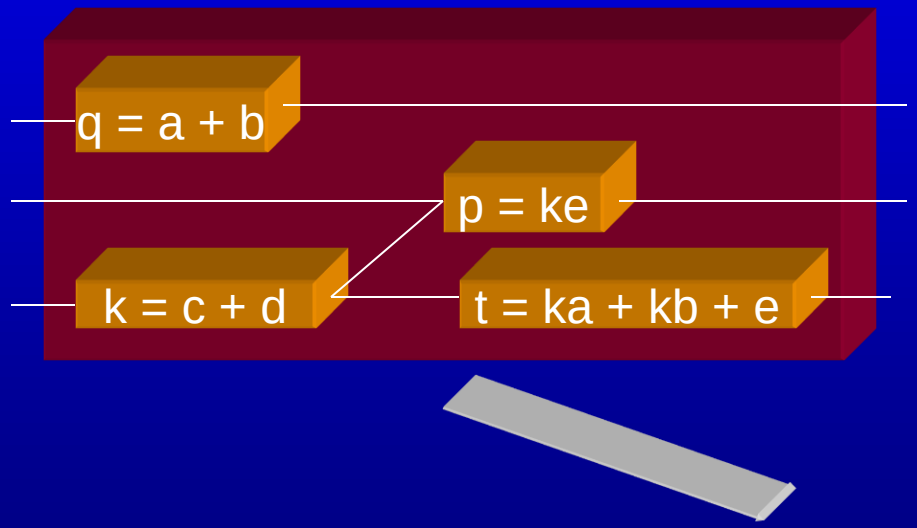
Transformations: SIMPLIFICATION

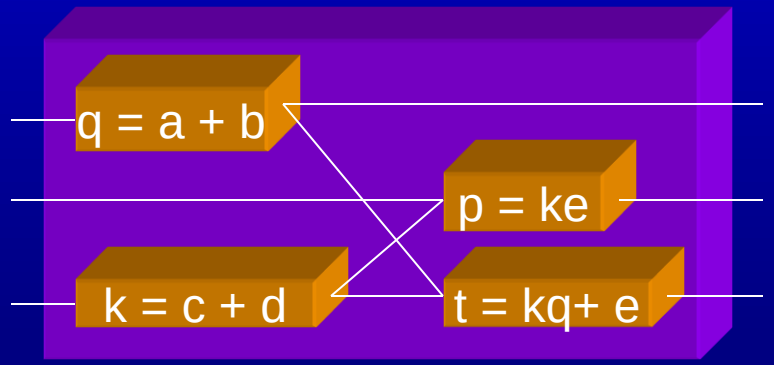
- Normal 2-level logic minimisation within an expression



Transformations: SUBSTITUTION

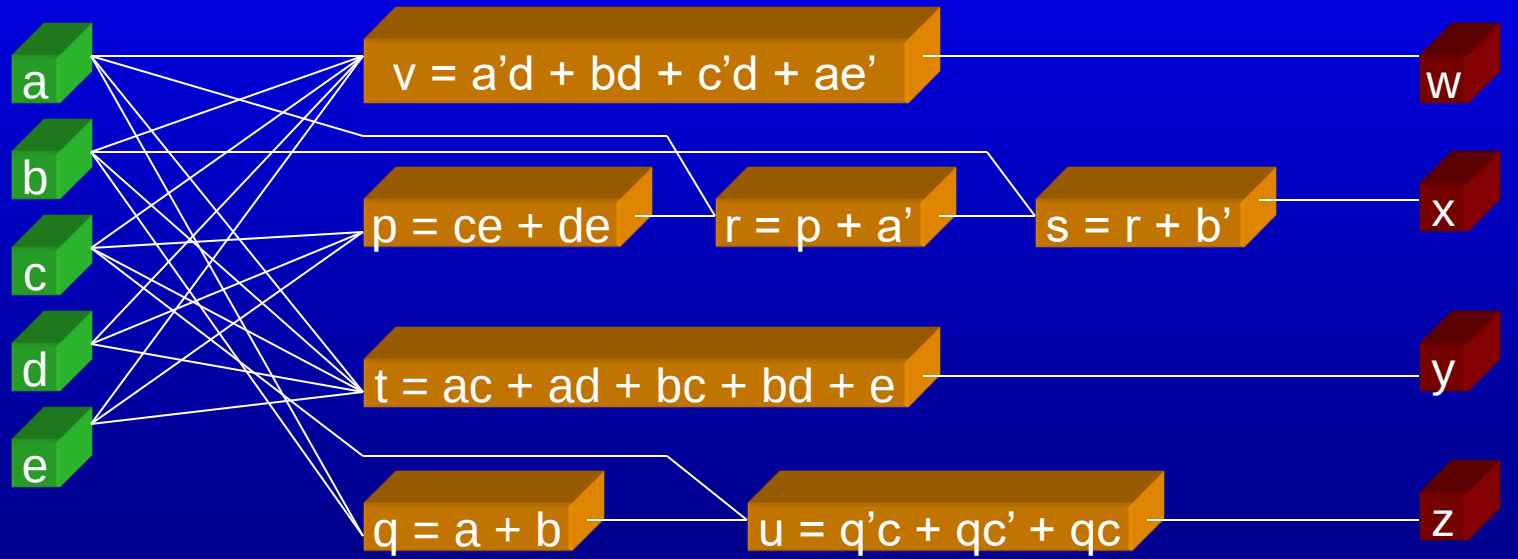
- Replacing by expression in terms of other nodes
  - creation of new dependency
  - maybe dropping of others





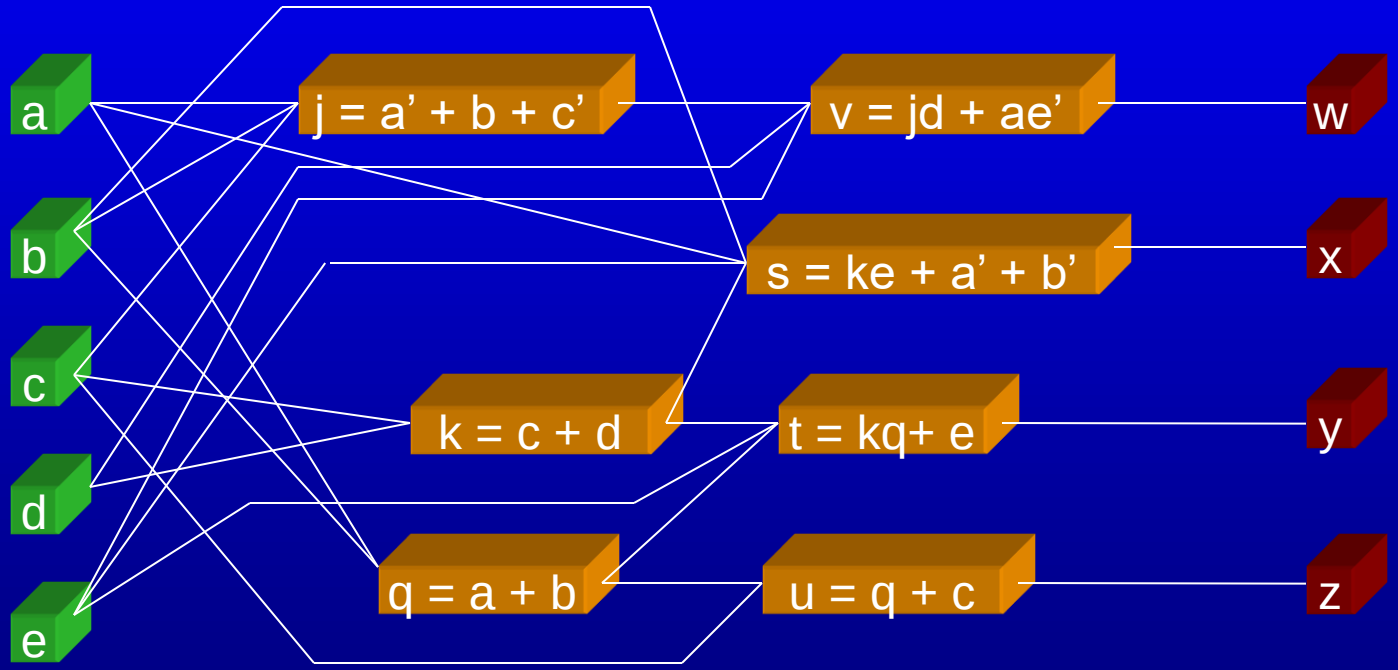
Result After Transformations

Original Circuit: 33 literals



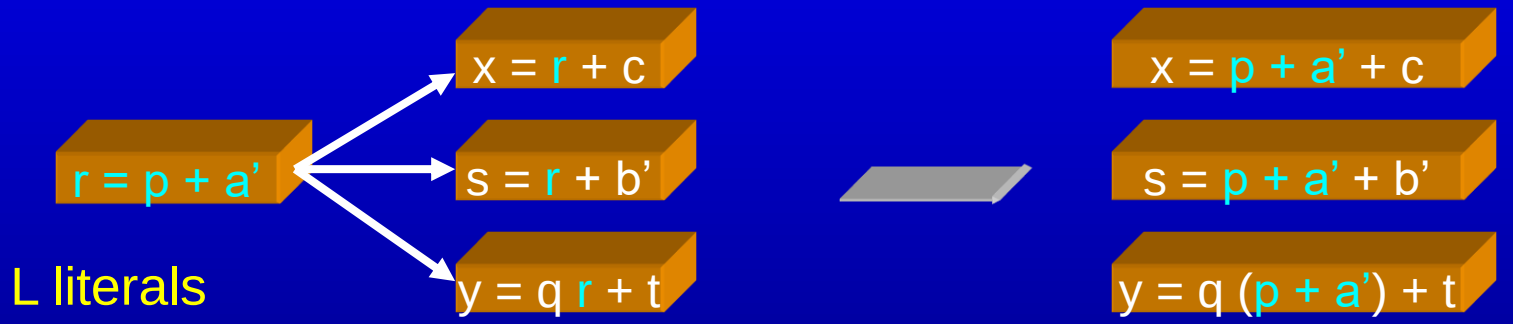
Final Circuit

20 literals



ELIMINATION Algorithm

- Assign VAL(v) to each node v
  - increase in area (literal count) caused by the elimination



L literals
(L = 2)

N occurrences of r
(N = 3)

Extra literals = NL - N - L

NL literals added
N r's dropped
L literals dropped when node is eliminated

ELIMINATION Algorithm

```
ELIMINATE (G, k)
repeat {
    v_x = select node with VAL (v_x) ≤ k
    if (v_x = Ø) return;
    Replace x by f_x in logic network
}
```

Increase literal count by no more than k per node eliminated