

Introduction to Electronic Design Automation

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University



Spring 2013

1

Logic Synthesis

High-level synthesis



Logic synthesis



Physical design

Part of the slides are by courtesy of Prof. Andreas Kuehlmann

2

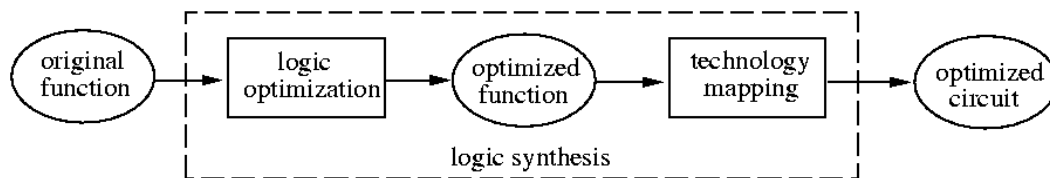
Logic Synthesis

□ Course contents

- Overview
- Boolean function representation
- Logic optimization
- Technology mapping

□ Reading

- Chapter 6



3

High-Level to Logic Synthesis

□ Hardware is normally partitioned into two parts:

- **Data path:** a network of functional units, registers, multiplexers and buses.
- **Control:** the circuit that takes care of having the data present at the right place at a specific time (i.e. FSM), or of presenting the right instructions to a programmable unit (i.e. microcode).

□ High-level synthesis often focuses on data-path optimization

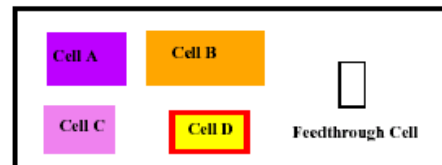
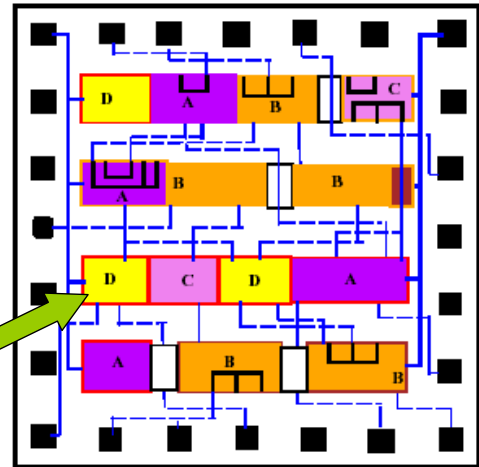
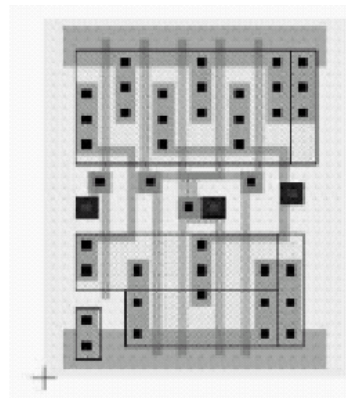
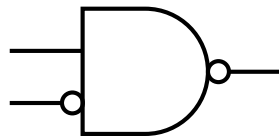
- The control part is then realized as an FSM

□ Logic synthesis often focuses on control-logic optimization

- Logic synthesis is widely used in application-specific IC (ASIC) design, where standard cell design style is most common

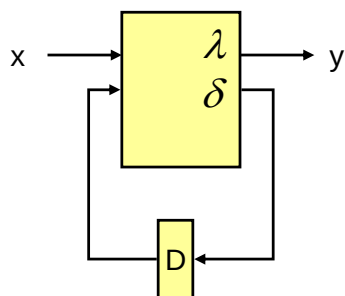
4

Standard-Cell Based Design



5

Transformation of Logic Synthesis



Given: Functional description of finite-state machine $F(Q, X, Y, \delta, \lambda)$ where:

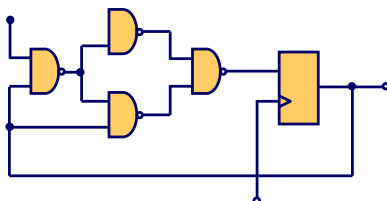
Q : Set of internal states

X : Input alphabet

Y : Output alphabet

$\delta: X \times Q \rightarrow Q$ (next state *function*)

$\lambda: X \times Q \rightarrow Y$ (output *function*)



Target: Circuit $C(G, W)$ where:

G : set of circuit components $g \in \{\text{gates, FFs, etc.}\}$

W : set of wires connecting G

6

Boolean Function Representation

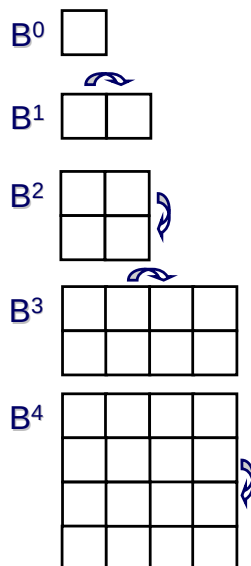
- Logic synthesis translates **Boolean functions** into **circuits**
- We need representations of Boolean functions for two reasons:
 - to represent and manipulate the actual circuit that we are implementing
 - to facilitate *Boolean reasoning*

7

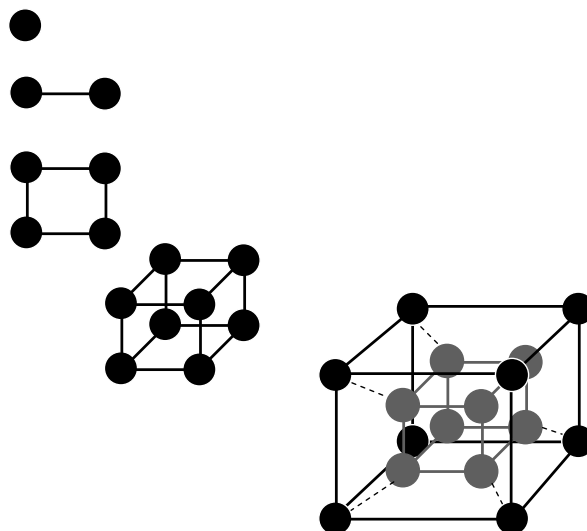
Boolean Space

- $B = \{0,1\}$
- $B^2 = \{0,1\} \times \{0,1\} = \{00, 01, 10, 11\}$

Karnaugh Maps:



Boolean Lattices:



8

Boolean Function

- A Boolean function f over input variables: $x_1, x_2, \dots, x_m$, is a mapping $f: \mathbf{B}^m \rightarrow Y$, where $\mathbf{B} = \{0,1\}$ and $Y = \{0,1,d\}$
 - E.g.
 - The output value of $f(x_1, x_2, x_3)$, say, partitions $\mathbf{B}^m$ into three sets:
 - **on-set** ($f=1$)
 - E.g. $\{010, 011, 110, 111\}$ (characteristic function $f^1 = x_2$)
 - **off-set** ($f=0$)
 - E.g. $\{100, 101\}$ (characteristic function $f^0 = x_1 \neg x_2$)
 - **don't-care set** ($f=d$)
 - E.g. $\{000, 001\}$ (characteristic function $f^d = \neg x_1 \neg x_2$)
- f is an **incompletely specified function** if the don't-care set is nonempty. Otherwise, f is a **completely specified function**
 - Unless otherwise said, a Boolean function is meant to be completely specified

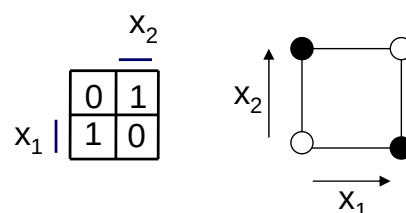
9

Boolean Function

- A Boolean function $f: \mathbf{B}^n \rightarrow \mathbf{B}$ over variables $x_1, \dots, x_n$ maps each Boolean valuation (truth assignment) in $\mathbf{B}^n$ to 0 or 1

Example

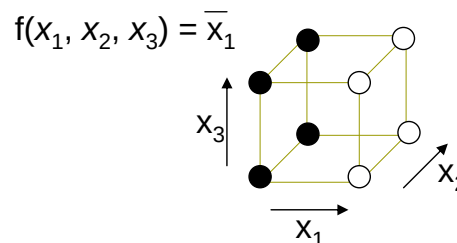
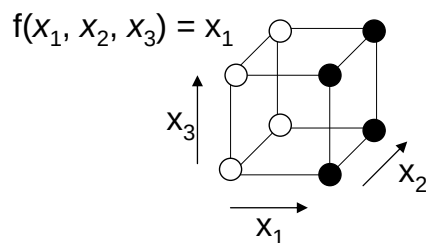
$f(x_1, x_2)$ with $f(0,0) = 0$, $f(0,1) = 1$, $f(1,0) = 1$, $f(1,1) = 0$



10

Boolean Function

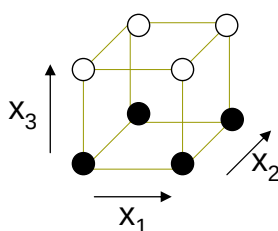
- **Onset** of f , denoted as f^1 , is $f^1 = \{v \in \mathbf{B}^n \mid f(v)=1\}$
 - If $f^1 = \mathbf{B}^n$, f is a **tautology**
- **Offset** of f , denoted as f^0 , is $f^0 = \{v \in \mathbf{B}^n \mid f(v)=0\}$
 - If $f^0 = \mathbf{B}^n$, f is **unsatisfiable**. Otherwise, f is **satisfiable**.
- f^1 and f^0 are sets, not functions!
- Boolean functions f and g are **equivalent** if $\forall v \in \mathbf{B}^n. f(v) = g(v)$ where v is a truth assignment or Boolean valuation
- A **literal** is a Boolean variable x or its negation x' (or $x, \neg x$) in a Boolean formula



11

Boolean Function

- There are 2^n vertices in $\mathbf{B}^n$
- There are 2^{2^n} distinct Boolean functions
 - Each subset $f^1 \subseteq \mathbf{B}^n$ of vertices in $\mathbf{B}^n$ forms a distinct Boolean function f with onset f^1



$x_1x_2x_3$	f
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	0
1 0 0	$\Rightarrow 1$
1 0 1	0
1 1 0	1
1 1 1	0

12

Boolean Operations

Given two Boolean functions:

$$f : \mathbf{B}^n \rightarrow \mathbf{B}$$

$$g : \mathbf{B}^n \rightarrow \mathbf{B}$$

□ $h = f \wedge g$ from **AND** operation is defined as

$$h^1 = f^1 \cap g^1; h^0 = \mathbf{B}^n \setminus h^1$$

□ $h = f \vee g$ from **OR** operation is defined as

$$h^1 = f^1 \cup g^1; h^0 = \mathbf{B}^n \setminus h^1$$

□ $h = \neg f$ from **COMPLEMENT** operation is defined as

$$h^1 = f^0; h^0 = f^1$$

13

Cofactor and Quantification

Given a Boolean function:

$$f : \mathbf{B}^n \rightarrow \mathbf{B}, \text{ with the input variable } (x_1, x_2, \dots, x_i, \dots, x_n)$$

□ **Positive cofactor on variable x_i**

$$h = f_{x_i} \text{ is defined as } h = f(x_1, x_2, \dots, 1, \dots, x_n)$$

□ **Negative cofactor on variable x_i**

$$h = f_{\neg x_i} \text{ is defined as } h = f(x_1, x_2, \dots, 0, \dots, x_n)$$

□ **Existential quantification over variable x_i**

$$h = \exists x_i. f \text{ is defined as } h = f(x_1, x_2, \dots, 0, \dots, x_n) \vee f(x_1, x_2, \dots, 1, \dots, x_n)$$

□ **Universal quantification over variable x_i**

$$h = \forall x_i. f \text{ is defined as } h = f(x_1, x_2, \dots, 0, \dots, x_n) \wedge f(x_1, x_2, \dots, 1, \dots, x_n)$$

□ **Boolean difference over variable x_i**

$$h = \partial f / \partial x_i \text{ is defined as } h = f(x_1, x_2, \dots, 0, \dots, x_n) \oplus f(x_1, x_2, \dots, 1, \dots, x_n)$$

14

Boolean Function Representation

□ Some common representations:

- Truth table
- Boolean formula
 - SOP (sum-of-products, or called disjunctive normal form, DNF)
 - POS (product-of-sums, or called conjunctive normal form, CNF)
- BDD (binary decision diagram)
- Boolean network (consists of nodes and wires)
 - Generic Boolean network
 - Network of nodes with generic functional representations or even subcircuits
 - Specialized Boolean network
 - Network of nodes with SOPs (PLAs)
 - And-Inv Graph (AIG)

□ Why different representations?

- Different representations have their own strengths and weaknesses (no single data structure is best for all applications)

15

Boolean Function Representation Truth Table

□ Truth table (function table for multi-valued functions):

The **truth table** of a function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ is a tabulation of its value at each of the 2^n vertices of $\mathbf{B}^n$.

In other words the truth table lists all **minterms**

Example: $f = a'b'c'd + a'b'cd + a'bc'd + ab'c'd + ab'cd + abc'd + abcd' + abcd$

The truth table representation is

- impractical for large n
- canonical

If two functions are the equal, then their **canonical** representations are isomorphic.

	abcd	f		abcd	f
0	0000	0	8	1000	0
1	0001	1	9	1001	1
2	0010	0	10	1010	0
3	0011	1	11	1011	1
4	0100	0	12	1100	0
5	0101	1	13	1101	1
6	0110	0	14	1110	1
7	0111	0	15	1111	1

16

Boolean Function Representation

Boolean Formula

- A **Boolean formula** is defined inductively as an expression with the following formation rules (syntax):

formula ::=	'(' formula ')'	
	Boolean constant	(true or false)
	<Boolean variable>	
	formula "+" formula	(OR operator)
	formula "." formula	(AND operator)
	¬ formula	(complement)

Example

$$f = (x_1 \cdot x_2) + (x_3) + \neg(\neg(x_4 \cdot (\neg x_1)))$$

typically "." is omitted and '(', ')' are omitted when the operator priority is clear, e.g., $f = x_1 x_2 + x_3 + x_4 \neg x_1$

17

Boolean Function Representation

Boolean Formula in SOP

- Any function can be represented as a **sum-of-products (SOP)**, also called **sum-of-cubes** (a **cube** is a product term), or **disjunctive normal form (DNF)**

Example

$$\phi = ab + a'c + bc$$

18

Boolean Function Representation

Boolean Formula in POS

- Any function can be represented as a **product-of-sums (POS)**, also called **conjunctive normal form (CNF)**
 - Dual of the SOP representation

Example

$$\varphi = (a+b'+c) (a'+b+c) (a+b'+c') (a+b+c)$$

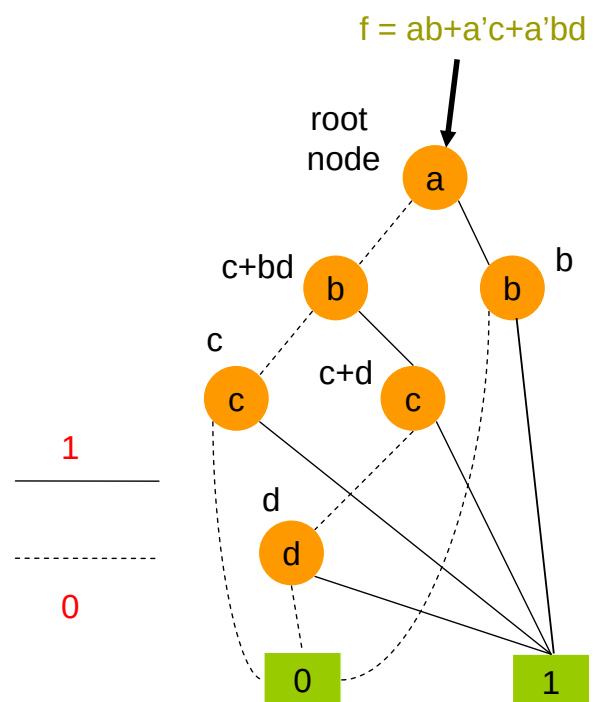
- Exercise: Any Boolean function in POS can be converted to SOP using De Morgan's law and the distributive law, and vice versa

19

Boolean Function Representation

Binary Decision Diagram

- BDD – a graph representation of Boolean functions
 - A **leaf node** represents constant 0 or 1
 - A **non-leaf node** represents a decision node (multiplexer) controlled by some variable
 - Can make a BDD representation **canonical** by imposing the **variable ordering** and **reduction** criteria (ROBDD)



20

Boolean Function Representation

Binary Decision Diagram

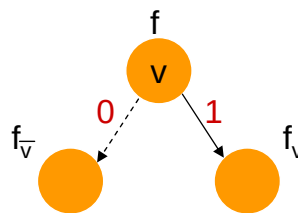
- Any Boolean function f can be written in term of **Shannon expansion**

$$f = v f_v + \neg v f_{\neg v}$$

- Positive cofactor: $f_{x_i} = f(x_1, \dots, x_i=1, \dots, x_n)$
- Negative cofactor: $f_{\neg x_i} = f(x_1, \dots, x_i=0, \dots, x_n)$

- BDD is a compressed Shannon cofactor tree:

- The two children of a node with function f controlled by variable v represent two sub-functions f_v and $f_{\neg v}$



21

Boolean Function Representation

Binary Decision Diagram

- Reduced** and **ordered** BDD (**ROBDD**) is a **canonical** Boolean function representation

- Ordered:**

- cofactor variables are in the **same order along all paths**

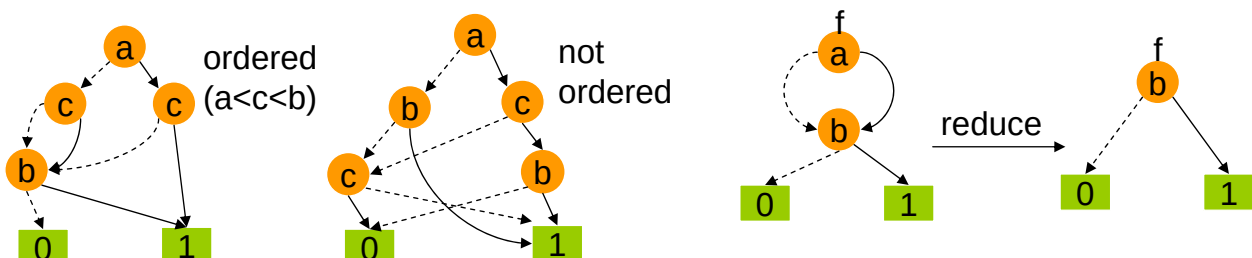
$$x_{i_1} < x_{i_2} < x_{i_3} < \dots < x_{i_n}$$

- Reduced:**

- any node with two identical children is removed

- two nodes with isomorphic BDD's are merged

These two rules make any node in an ROBDD represent a distinct logic function

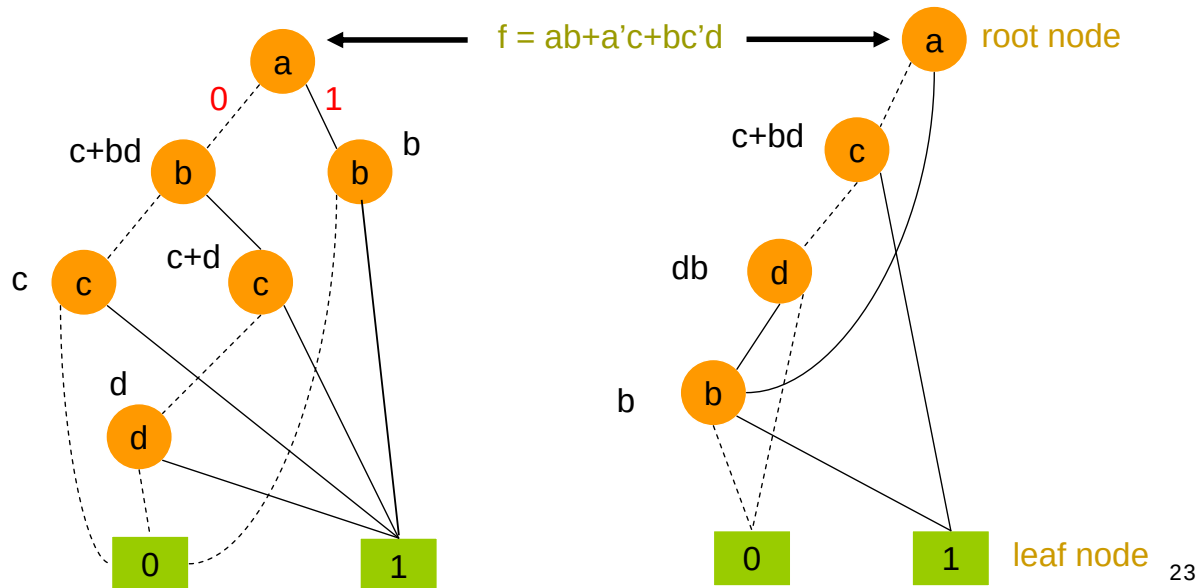


22

Boolean Function Representation

Binary Decision Diagram

- For a Boolean function,
 - ROBDD is unique with respect to a given variable ordering
 - Different orderings may result in different ROBDD structures



Boolean Function Representation

Boolean Network

- A **Boolean network** is a directed graph $C(G, N)$ where G are the gates and $N \subseteq (G \times G)$ are the directed edges (nets) connecting the gates.

Some of the vertices are designated:

Inputs: $I \subseteq G$

Outputs: $O \subseteq G$

$I \cap O = \emptyset$

Each gate g is assigned a Boolean function f_g which computes the output of the gate in terms of its inputs.

Boolean Function Representation

Boolean Network

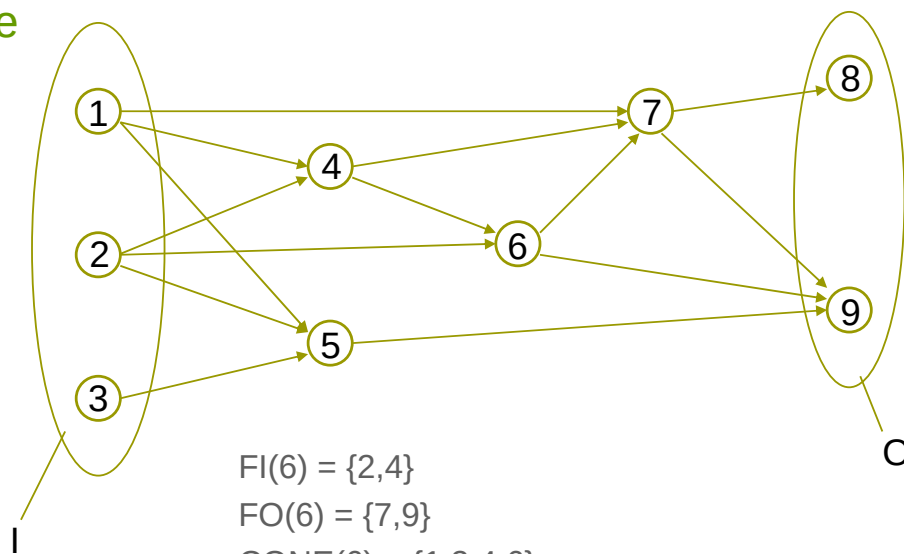
- The **fanin** $FI(g)$ of a gate g are the predecessor gates of g :
 $FI(g) = \{g' \mid (g', g) \in N\}$ (N : the set of nets)
- The **fanout** $FO(g)$ of a gate g are the successor gates of g :
 $FO(g) = \{g' \mid (g, g') \in N\}$
- The **cone** $CONE(g)$ of a gate g is the **transitive fanin (TFI)** of g and g itself
- The **support** $SUPPORT(g)$ of a gate g are all inputs in its cone:
 $SUPPORT(g) = CONE(g) \cap I$

25

Boolean Function Representation

Boolean Network

Example



$$FI(6) = \{2, 4\}$$

$$FO(6) = \{7, 9\}$$

$$CONE(6) = \{1, 2, 4, 6\}$$

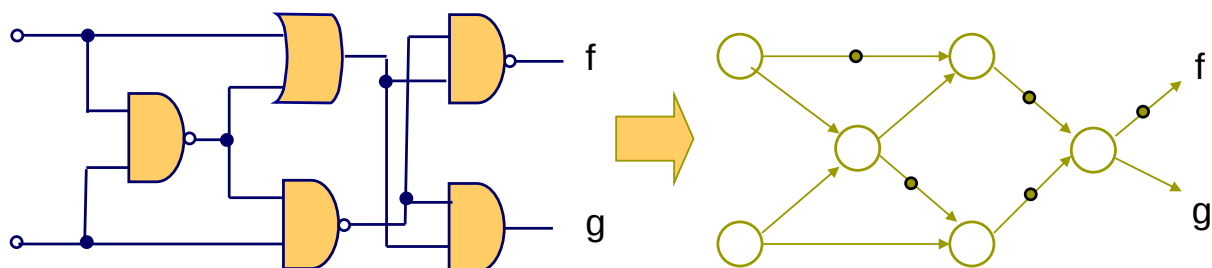
$$SUPPORT(6) = \{1, 2\}$$

Every node may have its own function

26

Boolean Function Representation And-Inverter Graph

- AND-INVERTER graphs (AIGs)
 - vertices: 2-input AND gates
 - edges: interconnects with (optional) dots representing INVs
- Hash table to identify and reuse structurally isomorphic circuits



27

Boolean Function Representation

- A **canonical form** of a Boolean function is a **unique** representation of the function
 - It can be used for verification purposes
- Example
 - Truth table is canonical
 - It grows exponentially with the number of input variables
 - ROBDD is canonical
 - It is of practical interests because it may represent many Boolean functions compactly
 - SOP, POS, Boolean networks are NOT canonical

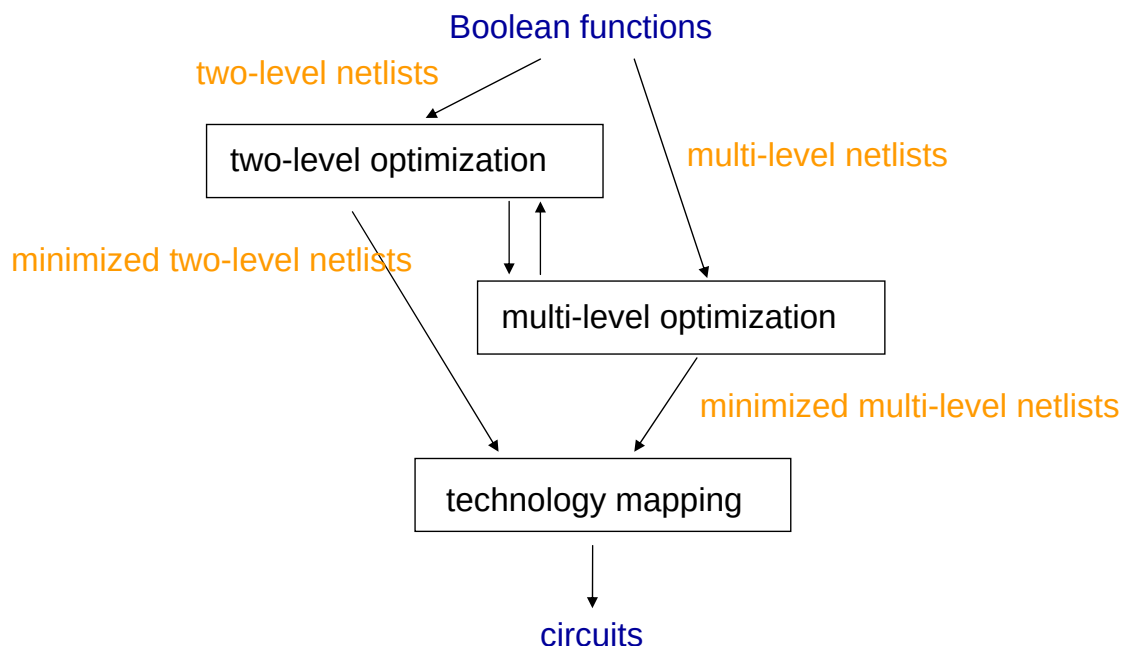
28

Boolean Function Representation

- Truth table
 - Canonical
 - Useful in representing small functions
- SOP
 - Useful in two-level logic optimization, and in representing local node functions in a Boolean network
- POS
 - Useful in SAT solving and Boolean reasoning
 - Rarely used in circuit synthesis (due to the asymmetric characteristics of NMOS and PMOS)
- ROBDD
 - Canonical
 - Useful in Boolean reasoning
- Boolean network
 - Useful in multi-level logic optimization
- AIG
 - Useful in multi-level logic optimization and Boolean reasoning

29

Logic Optimization



30

Two-Level Logic Minimization

- Any Boolean function can be realized using PLA in two levels: AND-OR (sum of products), NAND-NAND, etc.
 - Direct implementation of two-level logic using PLAs (programmable logic arrays) is not as popular as in the nMOS days
- Classic problem solved by the *Quine-McCluskey* algorithm
 - Popular cost function: #cubes and #literals in an SOP expression
 - #cubes – #rows in a PLA
 - #literals – #transistors in a PLA
 - The goal is to find a **minimal irredundant prime cover**

31

Two-Level Logic Minimization

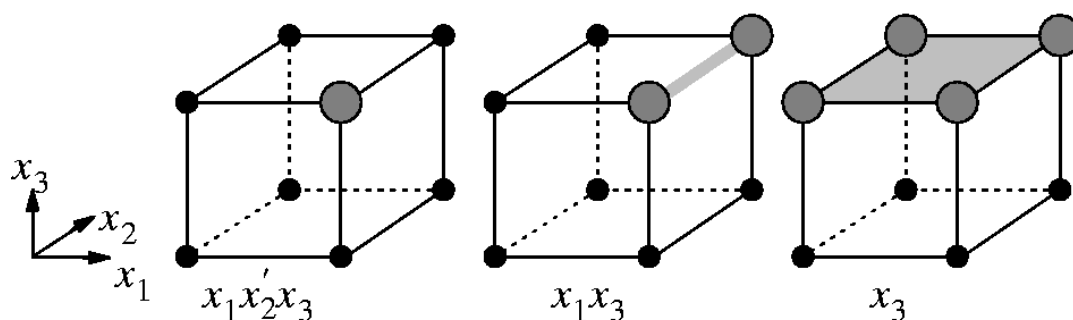
- Exact algorithm
 - Quine-McCluskey's procedure
- Heuristic algorithm
 - Espresso

32

Two-Level Logic Minimization

Minterms and Cubes

- A **minterm** is a product of **every** input variable or its negation
 - A minterm corresponds to a single point in $\mathbf{B}^n$
- A **cube** is a product of literals
 - The fewer the number of literals is in the product, the bigger the space is covered by the cube



33

Two-Level Logic Minimization

Implicant and Cover

- An **implicant** is a cube whose points are either in the on-set or the dc-set.
- A **prime implicant** is an implicant that is not included in any other implicant.
- A set of prime implicants that together cover all points in the on-set (and some or all points of the dc-set) is called a **prime cover**.
 - A prime cover is **irredundant** when none of its prime implicants can be removed from the cover.
 - An irredundant prime cover is **minimal** when the cover has the minimal number of prime implicants.
(c.f. minimum vs. minimal)

34

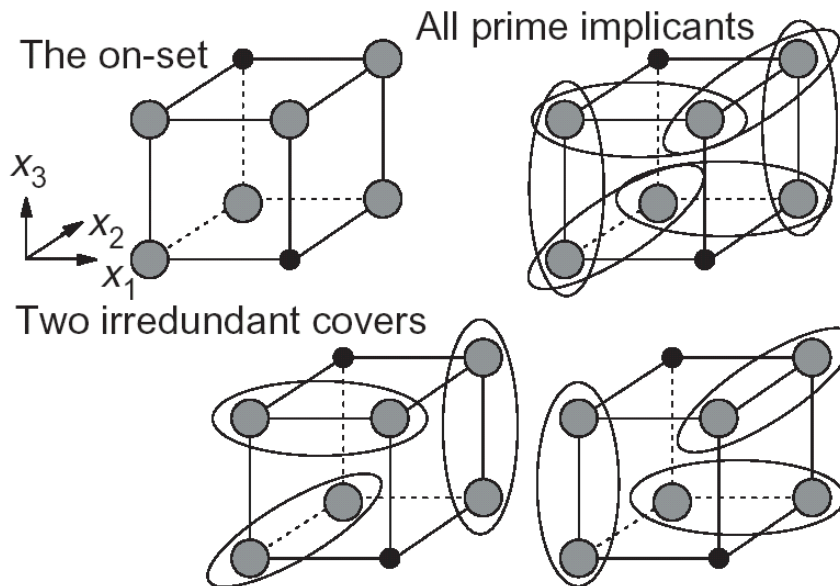
Two-Level Logic Minimization

Cover

Example

$$f = \neg x_1 \neg x_3 + \neg x_2 x_3 + x_1 x_2$$

$$f = \neg x_1 \neg x_2 + x_2 \neg x_3 + x_1 x_3$$

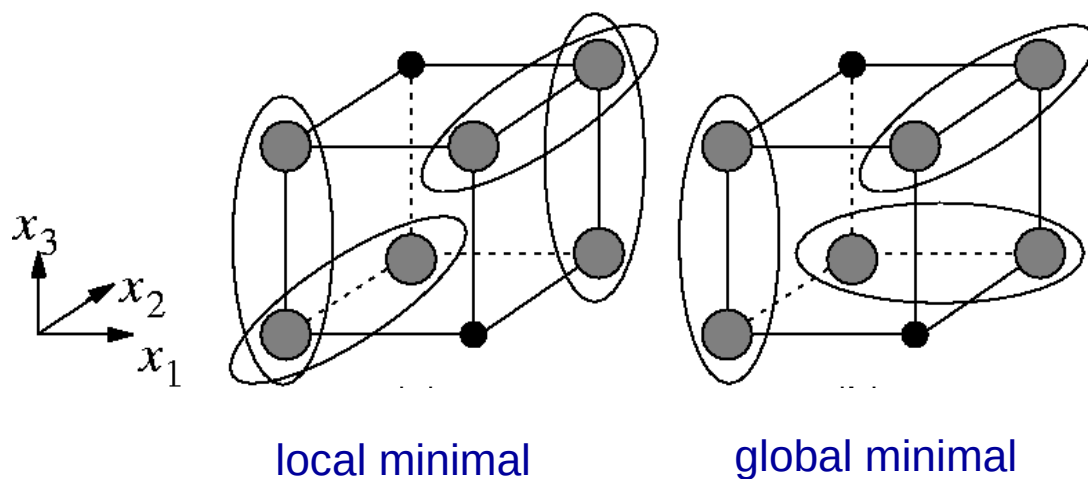


35

Two-Level Logic Minimization

Cover

Example



36

Two-Level Logic Minimization

Quine-McCluskey Procedure

- Given G and D (covers for $\mathfrak{S} = (f, d, r)$ and d , respectively), find a minimum cover G^* of primes where:

$$f \subseteq G^* \subseteq f + d \quad (G^* \text{ is a prime cover of } \mathfrak{S})$$

- f is the onset, d don't-care set, and r offset

Q-M Procedure:

- Generate all primes of $\mathfrak{S}$, $\{P_j\}$ (i.e. primes of $(f+d) = G+D$)
- Generate all minterms $\{m_i\}$ of $f = G \wedge \neg D$
- Build Boolean matrix B where

$$B_{ij} = 1 \text{ if } m_i \in P_j$$

$$= 0 \text{ otherwise}$$
- Solve the minimum column covering problem for B (unate covering problem)

37

Two-Level Logic Minimization

Quine-McCluskey Procedure

Generating Primes

Tabular method

(based on *consensus* operation):

- Start with all **minterm canonical form** of F
- Group *pairs* of adjacent minterms into cubes
- Repeat merging cubes until no more merging possible; mark (✓) + remove all covered cubes.
- Result: set of *primes* of f .

Example

$$F = x' y' + w x y + x' y z' + w y' z$$

$$F = x' y' + w x y + x' y z' + w y' z$$

$w' x' y' z'$ ✓	$w' x' y'$ ✓	$x' y'$
	$w' x' z'$ ✓	$x' z'$
	$x' y' z'$ ✓	
$w' x' y' z$ ✓	$x' y' z$ ✓	
$w' x' y z'$ ✓	$x' y z'$ ✓	
$w x' y' z'$ ✓	$w x' y'$ ✓	
	$w x' z'$ ✓	
$w x' y' z$ ✓	$w y' z$	
$w x' y z'$ ✓	$w y z'$	
$w x y z'$ ✓	$w x y$	
$w x y' z$ ✓	$w x z$	
$w x y z$ ✓		

Two-Level Logic Minimization Quine-McCluskey Procedure

Example

Karnaugh map

$\bar{x} \bar{z}$	$\bar{x} \bar{y}$	$\bar{x} y$	xy	$x \bar{y}$		$\bar{y}$
$\bar{z} w$	1	d	0	d		1
$\bar{z} \bar{w}$	d	1	d	1		0
zw	d	1	d	d		1
$z \bar{w}$	d	0	0	d		0

$$F = \bar{x} \bar{y} z w + \bar{x} y \bar{z} w + x \bar{y} z w + \bar{x} y z w \quad (\text{cover of } \mathfrak{F})$$

$$D = \bar{y} z + xyw + \bar{x} \bar{y} z w + x \bar{y} w + \bar{x} y \bar{z} w \quad (\text{cover of } d)$$

Primes: $\bar{y} + w + \bar{x} \bar{z}$

Covering Table

Solution: $\{1, 2\} \Rightarrow \bar{y} + w$ is a minimum prime cover (also $w + \bar{x} \bar{z}$)

39

Two-Level Logic Minimization Quine-McCluskey Procedure

Column covering of Boolean matrix

	$\bar{y}$	w	$\bar{x} \bar{z}$	Primes of f+d
$\bar{x} \bar{y} \bar{z} \bar{w}$	1	0	1	
$\bar{x} y \bar{z} w$	0	1	1	
$x \bar{y} \bar{z} w$	1	1	0	
$\bar{x} y z w$	0	1	0	Row singleton (essential minterm)

Essential prime

- Definition. An essential prime is a prime that covers an onset minterm of f not covered by any other primes.

40

Two-Level Logic Minimization

Quine-McCluskey Procedure

□ Row equality in Boolean matrix:

- In practice, many rows in a covering table are identical. That is, there exist minterms that are contained in the same set of primes.

■ Example

m_1	0101101
m_2	0101101

41

Two-Level Logic Minimization

Quine-McCluskey Procedure

□ Row dominance in Boolean matrix:

- A row i_1 whose set of primes is contained in the set of primes of row i_2 is said to **dominate** i_2 .

■ Example

i_1	011010
i_2	011110

- i_1 dominates i_2
- Can remove row i_2 because have to choose a prime to cover i_1 , and any such prime also covers i_2 . So i_2 is automatically covered.

42

Two-Level Logic Minimization

Quine-McCluskey Procedure

□ Column dominance in Boolean matrix:

- A column j_1 whose rows are a superset of another column j_2 is said to **dominate** j_2 .

■ Example

j_1	j_2
1	0
0	0
1	1
0	0
1	1

- j_1 dominates j_2
- We can remove column j_2 since j_1 covers all those rows and more. We would never choose j_2 in a minimum cover since it can always be replaced by j_1 .

43

Two-Level Logic Minimization

Quine-McCluskey Procedure

Reducing Boolean matrix

1. Remove all rows covered by essential primes (columns in row singletons). Put these primes in the cover G .
2. Group identical rows together and remove dominated rows.
3. Remove dominated columns. For equal columns, keep one prime to represent them.
4. Newly formed row singletons define **induced essential primes**.
5. Go to 1 if covering table decreased.

- The resulting reduced covering table is called the **cyclic core**. This has to be solved (**unate covering problem**). A minimum solution is added to G . The resulting G is a minimum cover.

44

Two-Level Logic Minimization

Quine-McCluskey Procedure

Solving cyclic core

□ Heuristic algorithm:

- Let $\mathcal{J} = \{I_1, I_2, \dots, I_k\}$ be the independent set of rows
- 1. choose $j \in I_i$ such that column j covers the most rows of A . Put P_j in G
- 2. eliminate all rows covered by column j
- 3. $\mathcal{J} \leftarrow \mathcal{J} \setminus \{I_i\}$
- 4. go to 1 if $|\mathcal{J}| > 0$
- 5. If B is empty, then done (in this case achieve minimum solution)
- 6. If B is not empty, choose an independent set of B and go to 1

1 11 1111 1 1	0
A	C

47

Two-Level Logic Minimization

Quine-McCluskey Procedure

□ Summary

- Calculate all prime implicants (of the union of the onset and don't care set)
- Find the minimal cover of all minterms in the onset by prime implicants
 - Construct the covering matrix
 - Simplify the covering matrix by detecting essential columns, row and column dominance
 - What is left is the cyclic core of the covering matrix.
 - The covering problem can then be solved by a branch-and-bound algorithm.

48

Two-Level Logic Minimization

Exact vs. Heuristic Algorithms

□ Quine-McCluskey Method:

1. Generate cover of all primes $G = p_1 + p_2 + \dots + p_{3^n/n}$
2. Make G irredundant (in optimum way)
 - Q-M is **exact**, i.e., it gives an exact minimum

□ Heuristic Methods:

1. Generate (somehow) a cover of $\mathfrak{F}$ using some of the primes $G = p_{i_1} + p_{i_2} + \dots + p_{i_k}$
2. Make G irredundant (maybe not optimally)
3. Keep best result - try again (i.e. go to 1)

49

Two-Level Logic Minimization

ESPRESSO

□ Heuristic two-level logic minimization

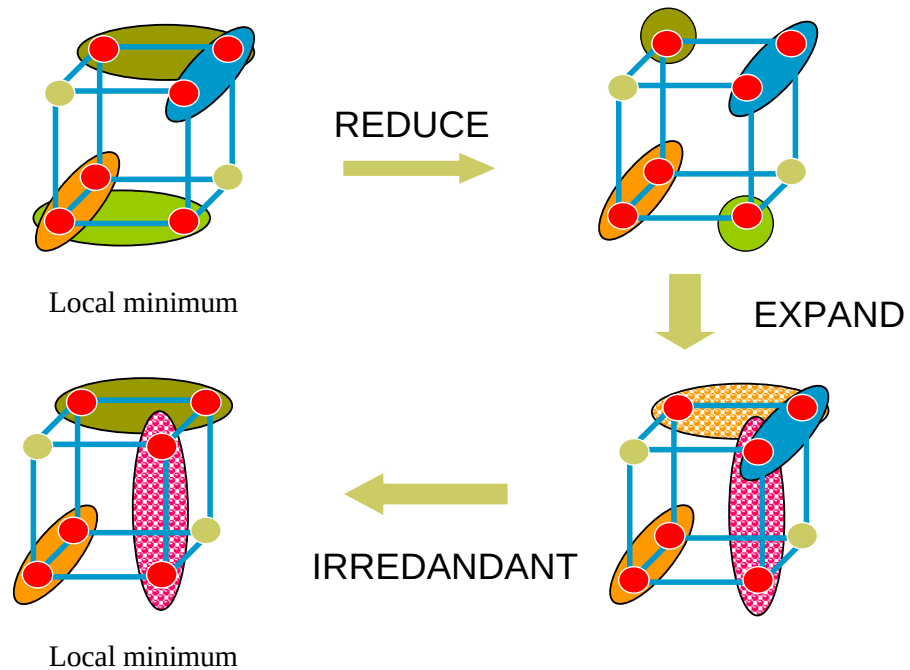
ESPRESSO($\mathfrak{F}$)

```
{
  (F,D,R) ← DECODE( $\mathfrak{F}$ )           //LASTGASP
  F ← EXPAND(F,R)                 G ← REDUCE_GASP(F,D)
  F ← IRREDUNDANT(F,D)            G ← EXPAND(G,R)
  E ← ESSENTIAL_PRIMES(F,D)       F ← IRREDUNDANT(F+G,D)
  F ← F-E; D ← D+E               //LASTGASP
  do{                             }while fewer terms in F
    do{
      F ← REDUCE(F,D)             F ← F+E; D ← D-E
      F ← EXPAND(F,R)             LOWER_OUTPUT(F,D)
      F ← IRREDUNDANT(F,D)        RAISE_INPUTS(F,R)
    }while fewer terms in F      error ← (Fold  $\not\subset$  F) or (F  $\not\subset$  Fold + D)
                                return (F,error)
}
```

50

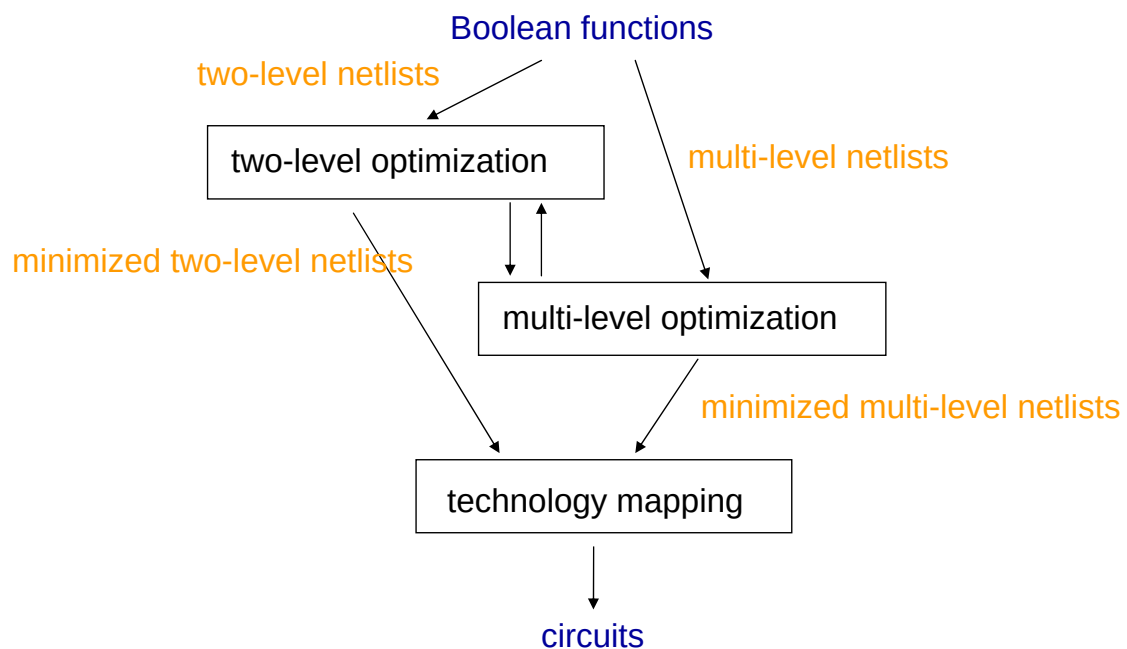
Two-Level Logic Minimization

ESPRESSO



51

Logic Minimization



52

Factor Form

Factor forms – beyond SOP

- Example:
 $(ad + b'c)(c + d'(e + ac')) + (d + e)fg$

Advantages

- good representation reflecting logic complexity (SOP may not be representative)
 - E.g., $f = ad + ae + bd + be + cd + ce$ has complement in simpler SOP $f' = a'b'c' + d'e'$; effectively has simple factor form $f = (a + b + c)(d + e)$
- in many design styles (e.g. complex gate CMOS design) the implementation of a function corresponds directly to its factored form
- good estimator of logic implementation complexity
- doesn't blow up easily

Disadvantages

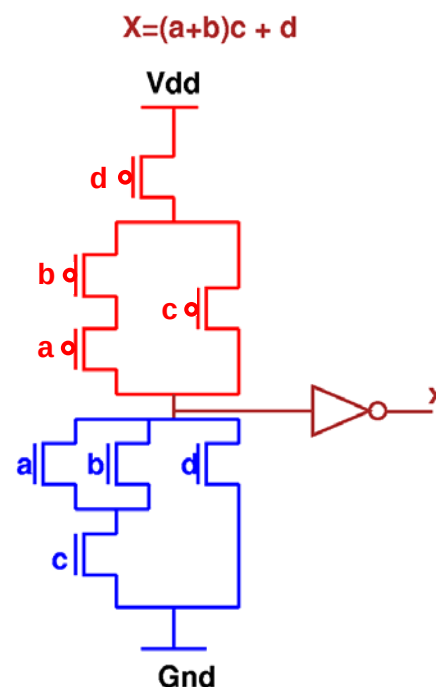
- not as many algorithms available for manipulation

53

Factor From

Factor forms are useful in **estimating** area and delay in multi-level logic

- **Note:** literal count $\approx$ transistor count $\approx$ area
 - however, area also depends on wiring, gate size, etc.
 - therefore very crude measure



54

Factor Form

- There are functions whose sizes are **exponential** in the SOP representation, but **polynomial** in the factored form

- **Example**

Achilles' heel function

$$\prod_{i=1}^{i=n/2} (x_{2i-1} + x_{2i})$$

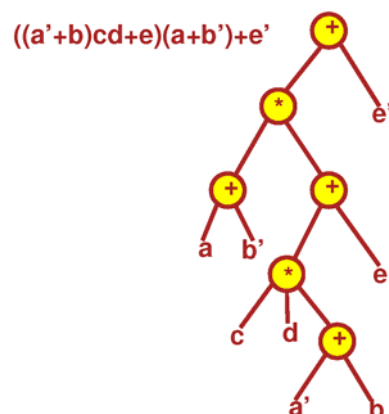
There are n literals in the factored form and $(n/2) \times 2^{n/2}$ literals in the SOP form.

55

Factor Form

- Factored forms can be graphically represented as labeled **trees**, called **factoring trees**, in which each internal node including the root is labeled with either $+$ or $\times$, and each leaf has a label of either a variable or its complement

- **Example:** factoring tree of $((a'+b)cd+e)(a+b')+e'$



56

Multi-Level Logic Minimization

- Basic techniques in Boolean network manipulation:
 - structural manipulation (change network topology)
 - node simplification (change node functions)
 - node minimization using don't cares

57

Multi-Level Logic Minimization Structural Manipulation

Restructuring Problem: Given initial network, find **best** network.

Example:

$$f_1 = abcd + abce + ab'cd' + ab'c'd' + a'c + cdf + abc'd'e' + ab'c'df'$$

$$f_2 = bdg + b'dfg + b'd'g + bd'eg$$

minimizing,

$$f_1 = bcd + bce + b'd' + a'c + cdf + abc'd'e' + ab'c'df'$$

$$f_2 = bdg + dfg + b'd'g + d'eg$$

factoring,

$$f_1 = c(b(d+e) + b'(d'+f) + a') + ac'(bd'e' + b'df')$$

$$f_2 = g(d(b+f) + d'(b'+e))$$

decompose,

$$f_1 = c(b(d+e) + b'(d'+f) + a') + ac'x'$$

$$f_2 = gx$$

$$x = d(b+f) + d'(b'+e)$$

Two problems:

- find good **common** subfunctions
- effect the **division**

58

Multi-Level Logic Minimization

Structural Manipulation

Basic operations:

1. Decomposition (for a single function)

$$f = abc + abd + a'c'd' + b'c'd'$$

⇓

$$f = xy + x'y' \quad x = ab \quad y = c + d$$

2. Extraction (for multiple functions)

$$f = (az + bz')cd + e \quad g = (az + bz')e' \quad h = cde$$

⇓

$$f = xy + e \quad g = xe' \quad h = ye \quad x = az + bz' \quad y = cd$$

3. Factoring (series-parallel decomposition)

$$f = ac + ad + bc + bd + e$$

⇓

$$f = (a + b)(c + d) + e$$

59

Multi-Level Logic Minimization

Structural Manipulation

Basic operations (cont'd):

4. Substitution

$$f = a + bc \quad g = a + b$$

⇓

$$f = g(a + c) \quad g = a + b$$

5. Collapsing (also called elimination)

$$f = ga + g'b \quad g = c + d$$

⇓

$$f = ac + ad + bc'd' \quad g = c + d$$

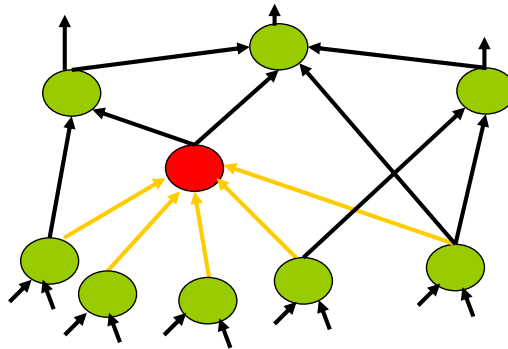
Note: “**division**” plays a key role in all these operations

60

Multi-Level Logic Minimization

Node Simplification

- Goal: For any node of a given Boolean network, find a **least-cost** SOP expression among the set of permissible functions for the node
 - Don't care computation + two-level logic minimization



combinational Boolean network

61

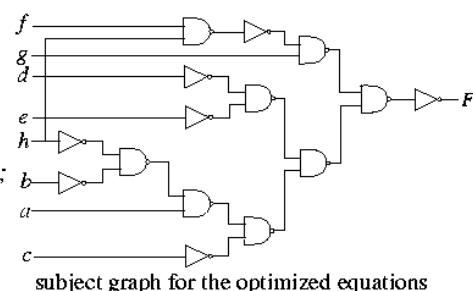
Combinational Logic Minimization

- **Two-level:** minimize #product terms and #literals
 - E.g., $F = x_1'x_2'x_3' + x_1'x_2'x_3 + x_1x_2'x_3' + x_1x_2'x_3 + x_1x_2x_3' \Rightarrow F = x_2' + x_1x_3'$
- **Multi-level:** minimize the # literals (**area** minimization)
 - E.g., equations are optimized using a smaller number of literals

$t1 = a + b \ c;$
 $t2 = d + e;$
 $t3 = a \ b + d;$
 $t4 = t1 \ t2 + f \ g;$
 $t5 = t4 \ h + t2 \ t3;$
 $F = t5';$

logic
optimization

$t1 = d + e;$
 $t2 = b + h;$
 $t3 = a \ t2 + c;$
 $t4 = t1 \ t3 + f \ g \ h;$



subject graph for the optimized equations

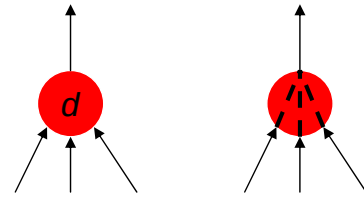
62

Timing Analysis and Optimization

□ Delay model at logic level

■ Gate delay model (our focus)

- Constant gate delay, or pin-to-pin gate delay
- Not accurate



■ Fanout delay model

- Gate delay considering fanout load (#fanouts)
- Slightly more accurate

■ Library delay model

- Tabular delay data given in the cell library
 - Determine delay from **input slew** and **output load**
 - Table look-up + interpolation/extrapolation
- Accurate

63

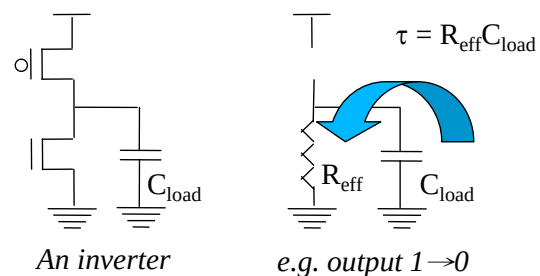
Timing Analysis and Optimization

Gate Delay

The delay of a gate depends on:

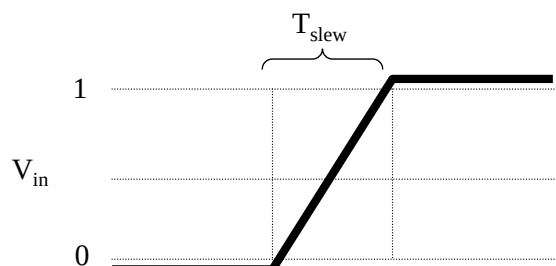
1. Output Load

- Capacitive loading $\propto$ charge needed to swing the output voltage
- Due to interconnect and logic fanout



2. Input Slew

- Slew = transition time
- Slower transistor switching $\Rightarrow$ longer delay and longer output slew

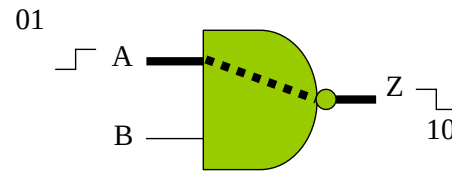


64

Timing Analysis and Optimization

Timing Library

- Timing library contains all relevant information about each standard cell
 - E.g., pin direction, clock, pin capacitance, etc.
- Delay (fastest, slowest, and often typical) and output slew are encoded for each input-to-output path and each pair of transition directions
- Values typically represented as 2 dimensional look-up tables (of output load and input slew)
 - Interpolation is used



```
Path(
  inputPorts(A),
  outputPorts(Z),
  inputTransition(01),
  outputTransition(10),
  "delay_table_1",
  "output_slew_table_1"
);
```

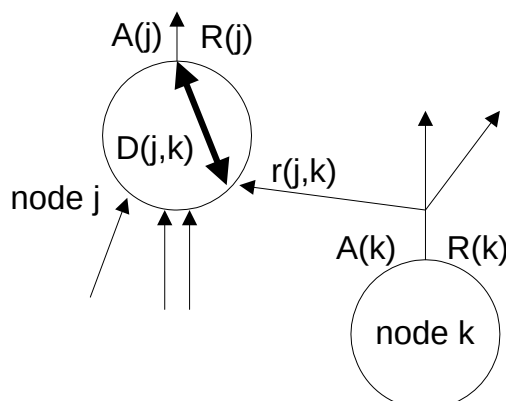
"delay_table_1"
Output load (nF)

Input slew (ns)	1.0	2.0	4.0	10.0
	0.1	2.1	2.6	3.4
	0.5	2.4	2.9	3.9
	1.0	2.6	3.4	4.0
	2.0	2.8	3.7	4.9

65

Static Timing Analysis

- **Arrival time**: the time signal arrives
 - Calculated from input to output in the topological order
- **Required time**: the time signal must ready (e.g., due to the clock cycle constraint)
 - Calculated from output to input in the reverse topological order
- **Slack** = required time – arrival time
 - Timing flexibility margin (positive: good; negative: bad)



A(j): arrival time of signal j
 R(k): required time or for signal k
 S(k): slack of signal k
 D(j,k): delay of node j from input k

$$A(j) = \max_{k \in FI(j)} [A(k) + D(j,k)]$$

$$r(j,k) = R(j) - D(j,k)$$

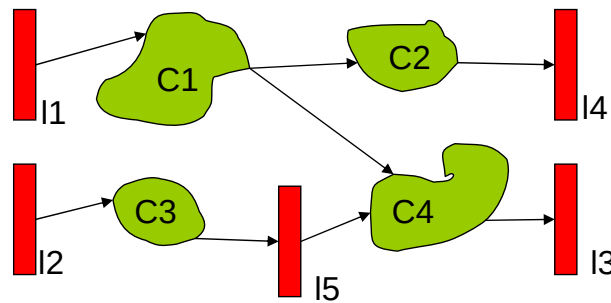
$$R(k) = \min_{j \in FO(k)} [r(j,k)]$$

$$S(k) = R(k) - A(k)$$

66

Static Timing Analysis

- Arrival times known at register outputs l_1 , l_2 , and l_5
- Required times known at register inputs l_3 , l_4 , and l_5
- Delay analysis gives arrival and required times (hence slacks) for combinational blocks C_1 , C_2 , C_3 , C_4



67

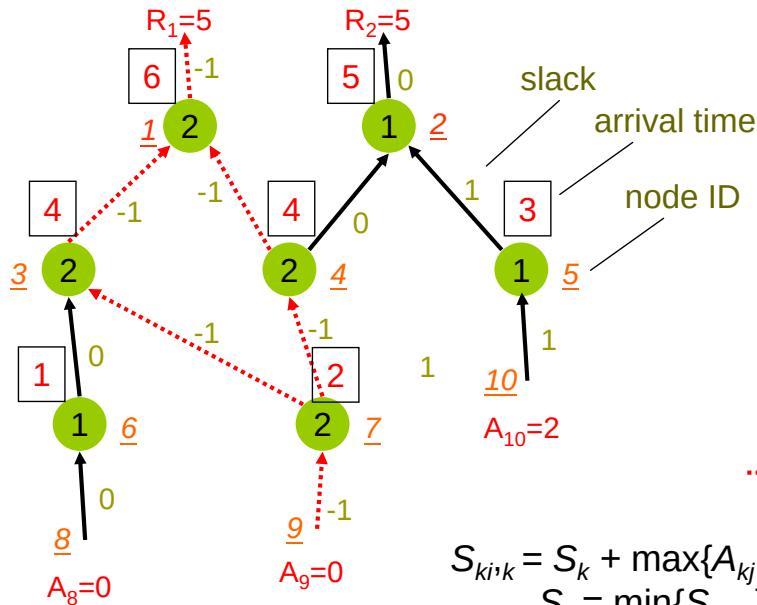
Static Timing Analysis

- **Arrival time** can be computed in the **topological order from inputs to outputs**
 - When a node is visited, its output arrival time is:
the max of its fanin arrival times + its own gate delay
- **Required time** can be computed in the **reverse topological order from outputs to inputs**
 - When a node is visited, its input required time is:
the min of its fanout required times – its own gate delay

68

Static Timing Analysis

Example



$A_1 = 6$	$R_1 = 5$
$A_2 = 5$	$R_2 = 5$
$S_1 = -1$	$R_3 = 3$
$S_2 = 0$	$R_7 = 1$
$S_{3,1} = -1$	$R_9 = -1$
$S_{4,1} = -1$	
$S_{4,2} = 0$	
$S_{5,2} = 1$	
$S_{6,3} = 0$	
$S_{7,3} = -1$	
$S_{7,4} = -1$	
$S_{7,5} = 1$	
$S_{8,6} = 0$	
$S_{9,7} = -1$	

$$S_{k_i, k} = S_k + \max\{A_{k_j}\} - A_{k_i}, k_j, k_i \in \text{fanin}(k)$$

$$S_k = \min\{S_{k, k_j}\}, k_j \in \text{fanout}(k)$$

69

Timing Optimization

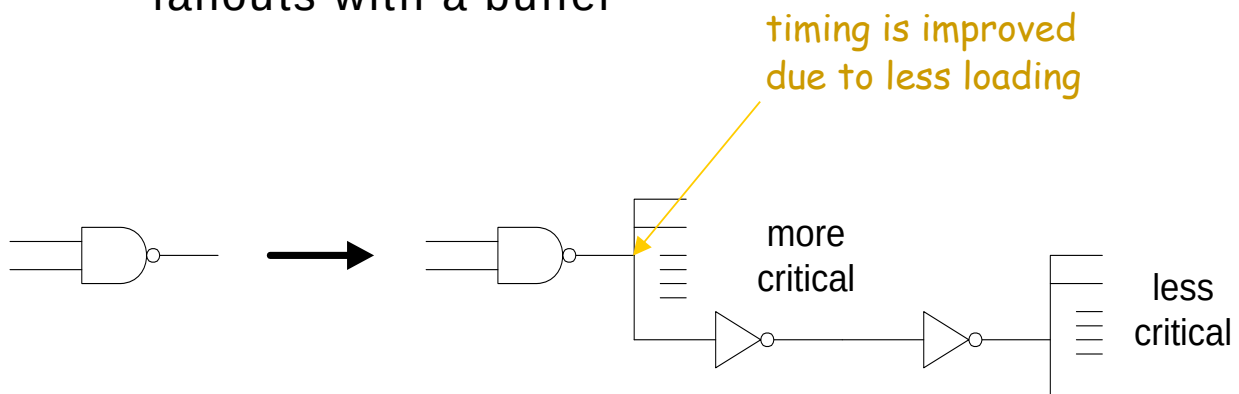
- Identify timing critical regions
- Perform timing optimization on the selected regions
 - E.g., gate sizing, buffer insertion, fanout optimization, tree height reduction, etc.

70

Timing Optimization

□ Buffer insertion

- Divide the fanouts of a gate into critical and non-critical parts, and drive the non-critical fanouts with a buffer

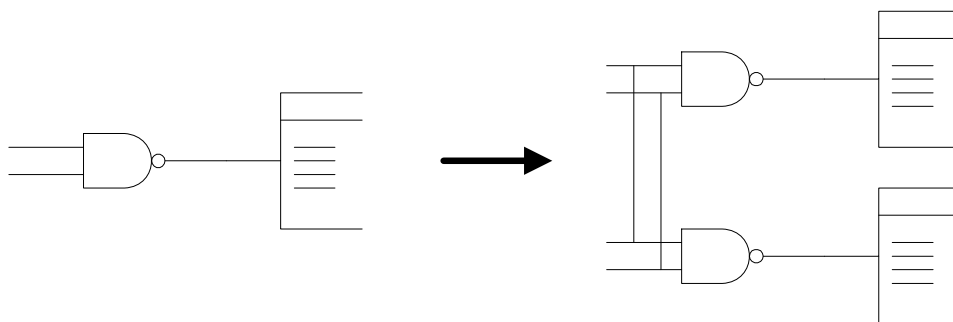


71

Timing Optimization

□ Fanout optimization

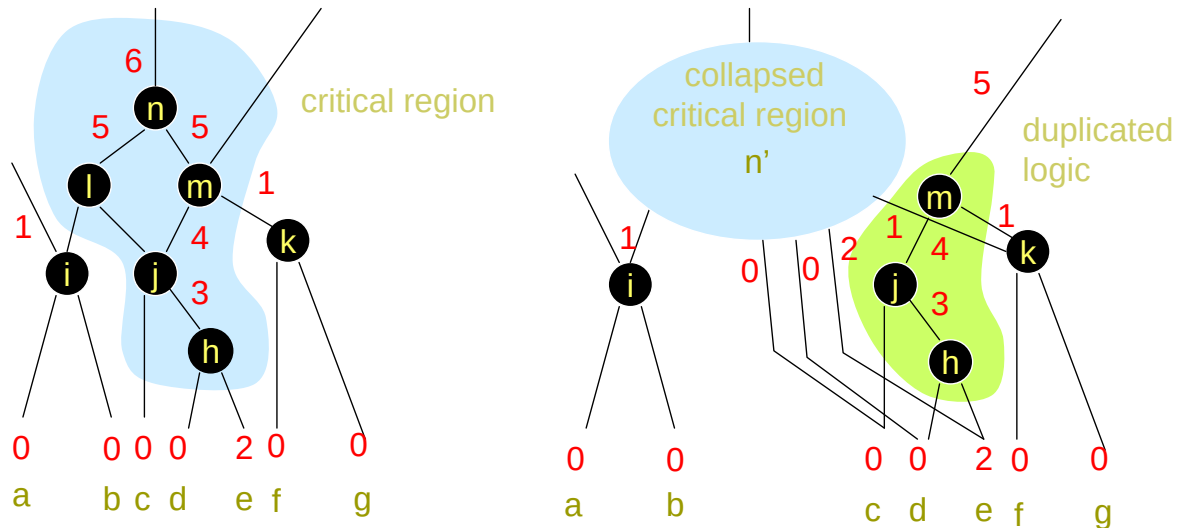
- Split the fanouts of a gate into several parts. Each part is driven by a copy of the original gate.



72

Timing Optimization

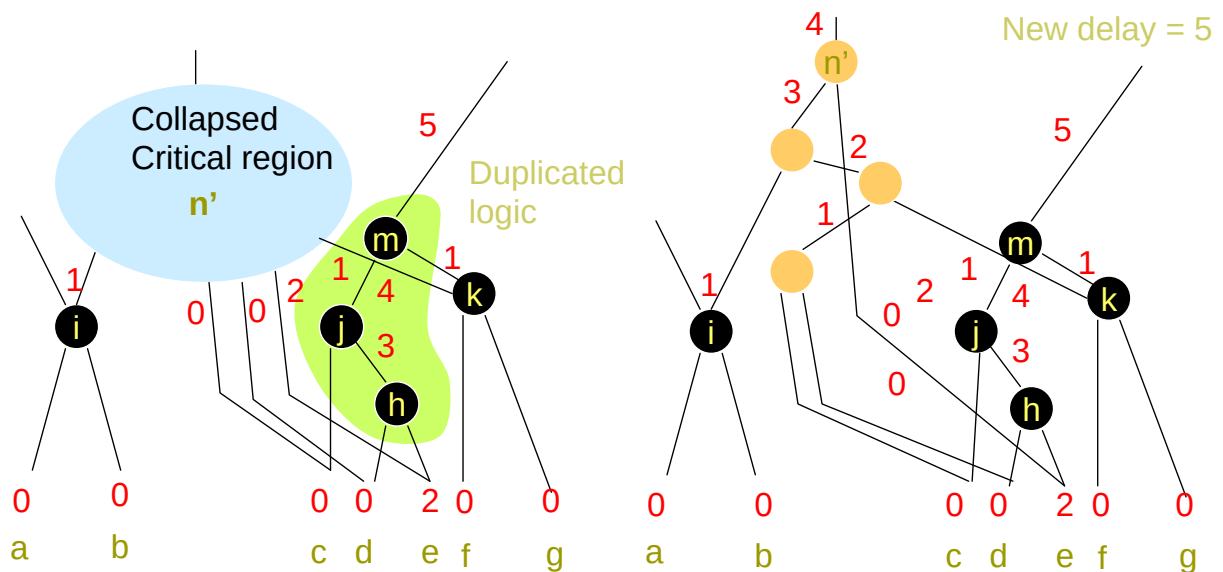
Tree height reduction



73

Timing Optimization

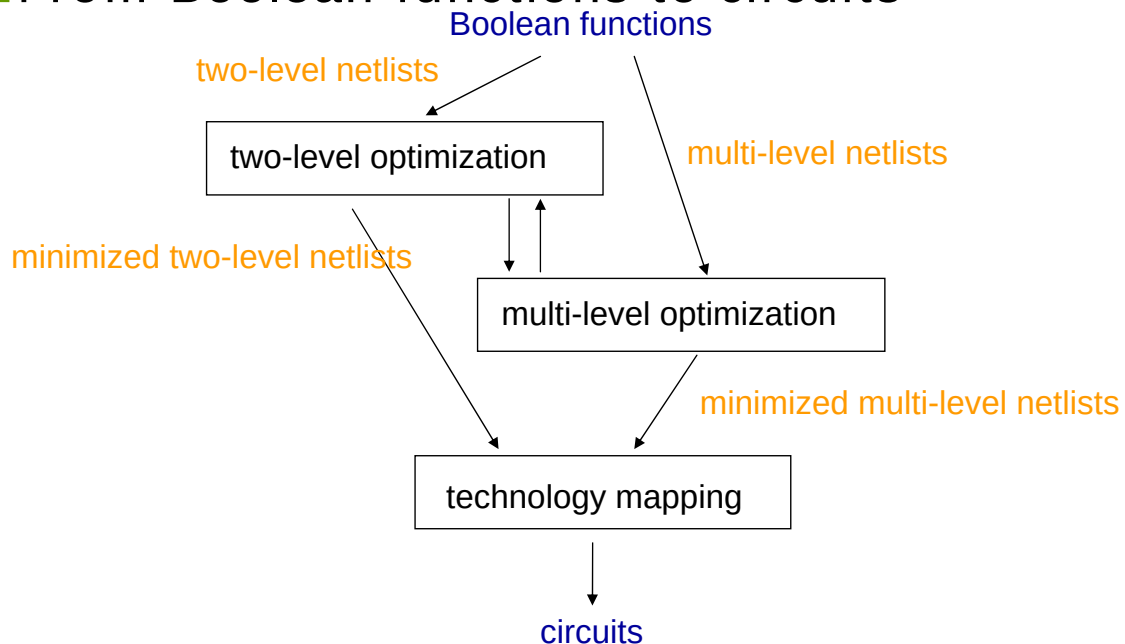
Tree height reduction



74

Combinational Optimization

□ From Boolean functions to circuits



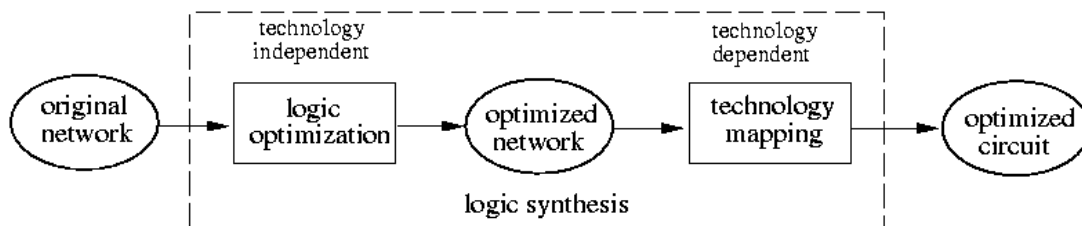
75

Technology Independent vs. Dependent Optimization

- Technology independent optimization produces a two-level or multi-level netlist where literal and/or cube counts are minimized
- Given the optimized netlist, its logic gates are to be implemented with library cells
- The process of associating logic gates with library cells is **technology mapping**
 - Translation of a technology independent representation (e.g. Boolean networks) of a circuit into a circuit for a given technology (e.g. standard cells) with optimal cost

76

Technology Mapping



■ Standard-cell technology mapping: standard cell design

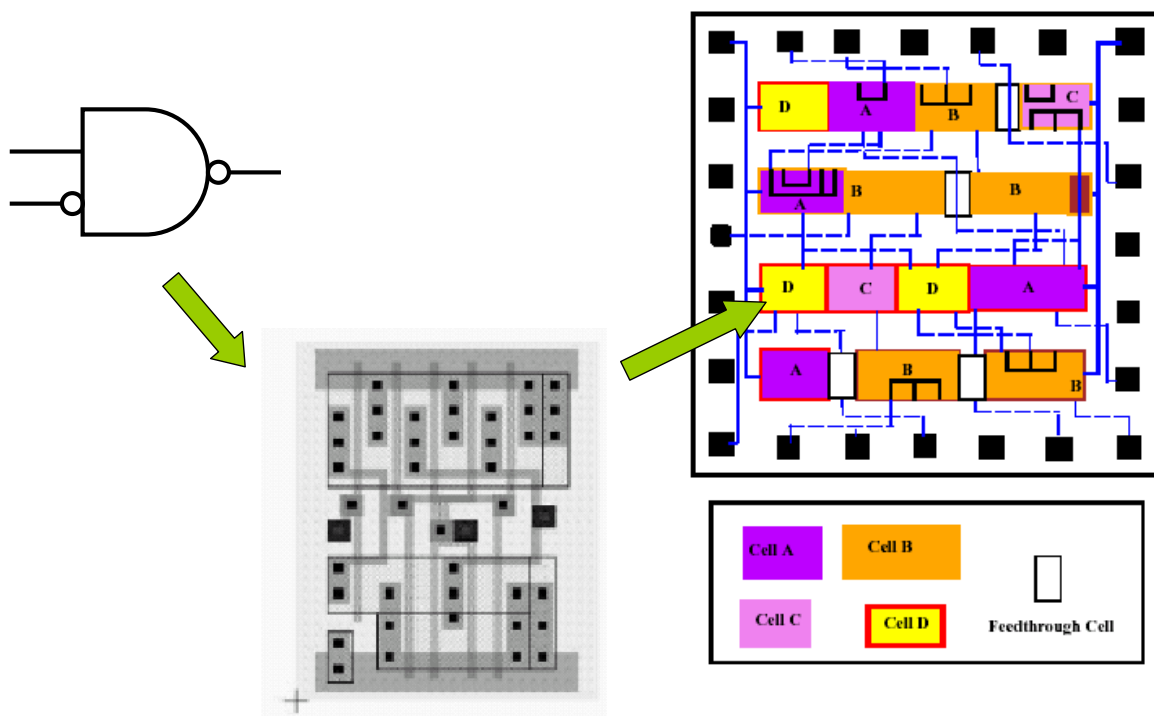
- Map a function to a limited set of pre-designed library cells

■ FPGA technology mapping

- Lookup table (LUT) architecture:
 - E.g., Lucent, Xilinx FPGAs
 - Each lookup table (LUT) can implement all logic functions with up to k inputs ($k = 4, 5, 6$)
- Multiplexer-based technology mapping:
 - E.g., Actel FPGA
 - Logic modules are constructed with multiplexers

77

Standard-Cell Based Design



78

Technology Mapping

□ Formulation:

- Choose base functions
 - Ex: 2-input NAND and Inverter
- Represent the (optimized) Boolean network with base functions
 - Subject graph
- Represent library cells with base functions
 - Pattern graph
 - Each pattern is associated with a cost depending on the optimization criteria, e.g., area, timing, power, etc.

□ Goal:

- Find a minimal cost covering of a subject graph using pattern graphs

79

Technology Mapping

- **Technology Mapping:** The optimization problem of finding a minimum cost covering of the subject graph by choosing from a collection of pattern graphs of gates in the library.

- A **cover** is a collection of pattern graphs such that every node of the subject graph is contained in one (or more) of the pattern graphs.
- The cover is further constrained so that each input required by a pattern graph is actually an output of some other pattern graph.

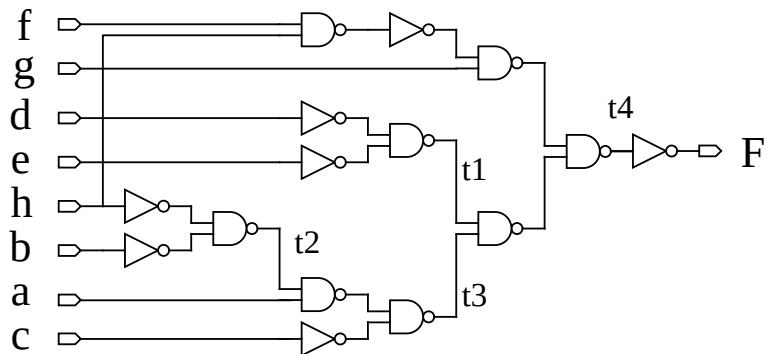
80

Technology Mapping

Example

Subject graph

$t1 = d + e$
 $t2 = b + h$
 $t3 = a t2 + c$
 $t4 = t1 t3 + f g h$
 $F = t4'$

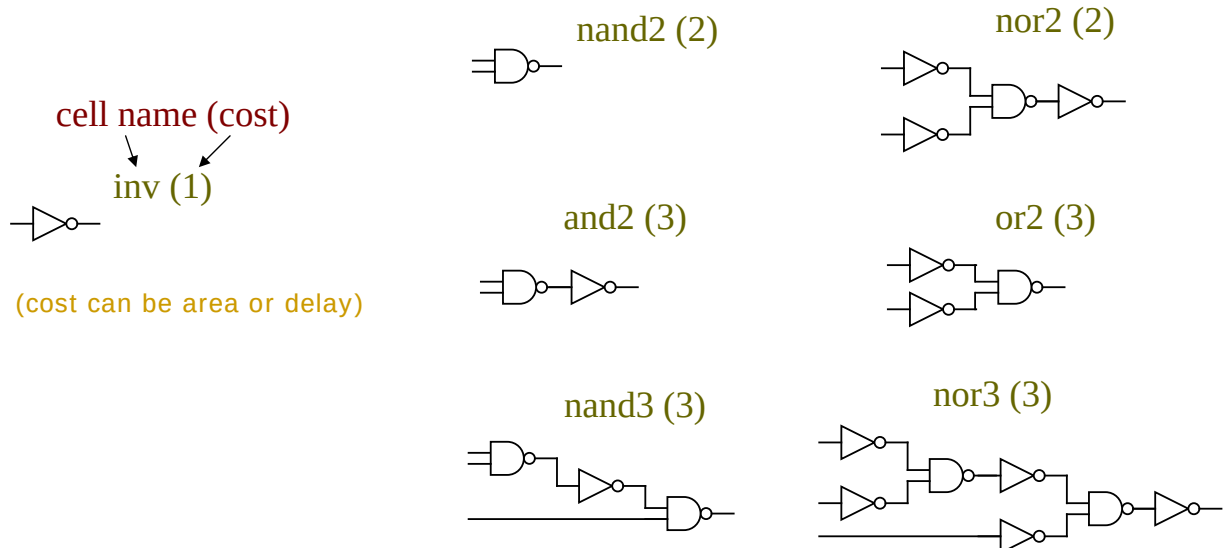


81

Technology Mapping

Example

Pattern graphs (1/3)



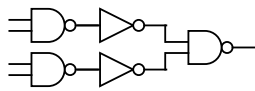
82

Technology Mapping

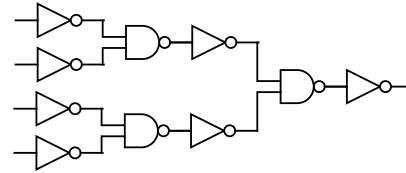
Example

Pattern graphs (2/3)

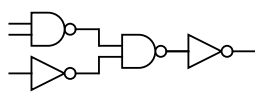
nand4 (4)



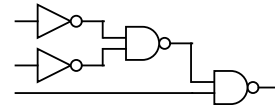
nor4 (4)



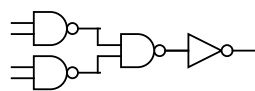
aoi21 (3)



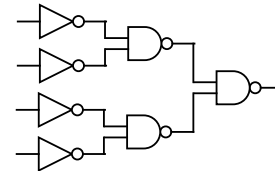
oai21 (3)



aoi22 (4)



oai22 (4)



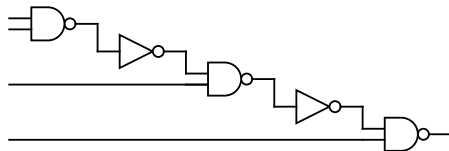
83

Technology Mapping

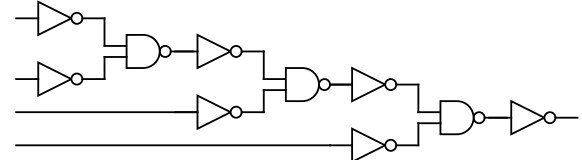
Example

Pattern graphs (3/3)

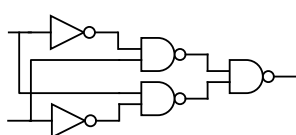
nand4 (4)



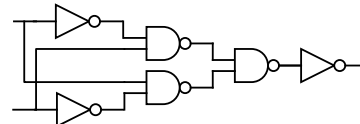
nor4 (4)



xor (5)



xnor (5)



84

Technology Mapping

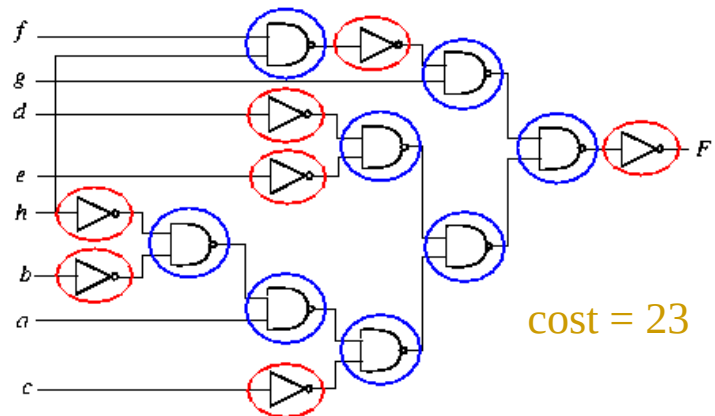
Example

A trivial covering

Mapped into NAND2's and INV's

- 8 NAND2's and 7 INV's at cost of 23

$$\begin{aligned} t1 &= d + e; \\ t2 &= b + h; \\ t3 &= a \cdot t2 + c; \\ t4 &= t1 \cdot t3 + f \cdot g \cdot h; \end{aligned}$$

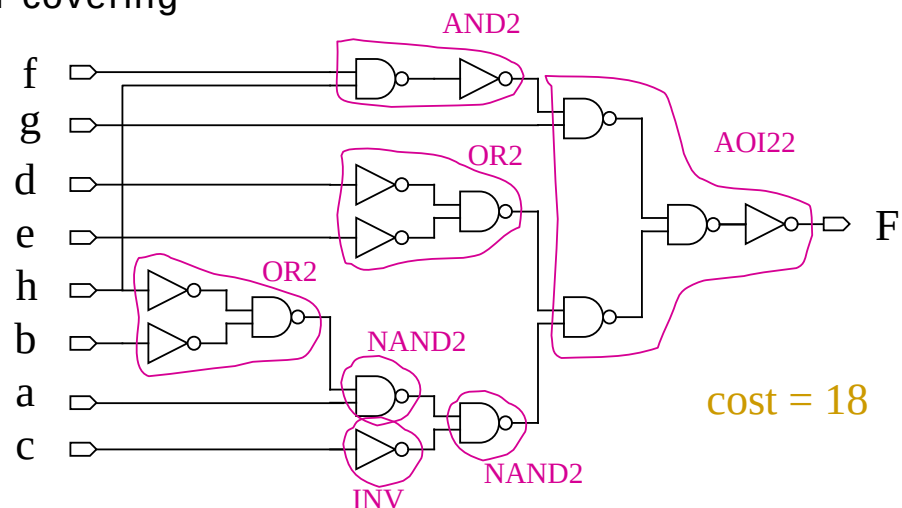


85

Technology Mapping

Example

A better covering



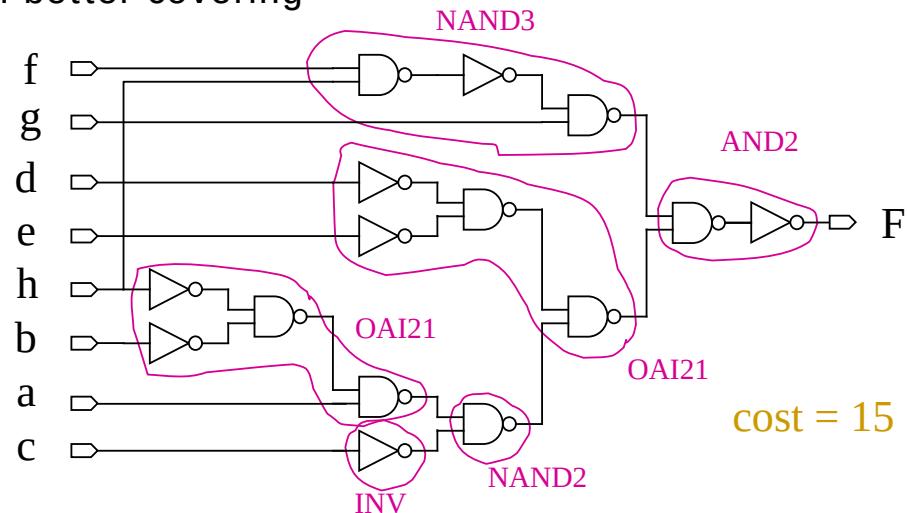
For a covering to be legal, every input of a pattern graph must be the output of another pattern graph!

86

Technology Mapping

□ Example

■ An even better covering



For a covering to be legal, every input of a pattern graph must be the output of another pattern graph!

87

Technology Mapping

□ Complexity of covering on directed acyclic graphs (DAGs)

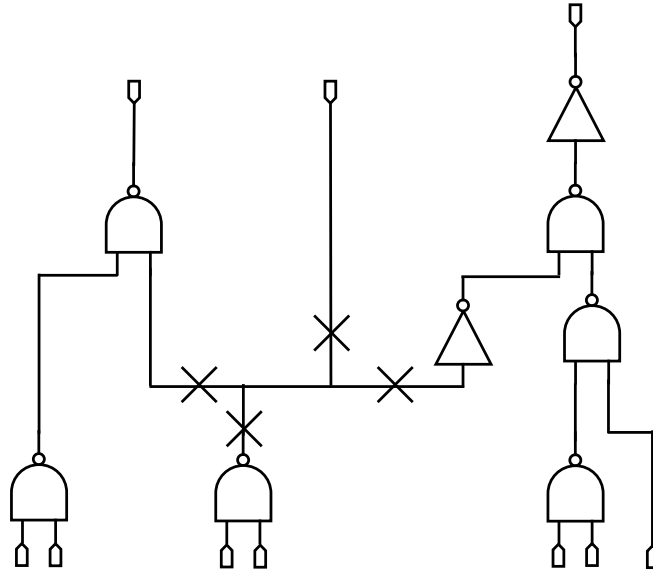
■ NP-complete

■ If the subject graph and pattern graphs are trees, then an efficient algorithm exists (based on [dynamic programming](#))

88

Technology Mapping DAGON Approach

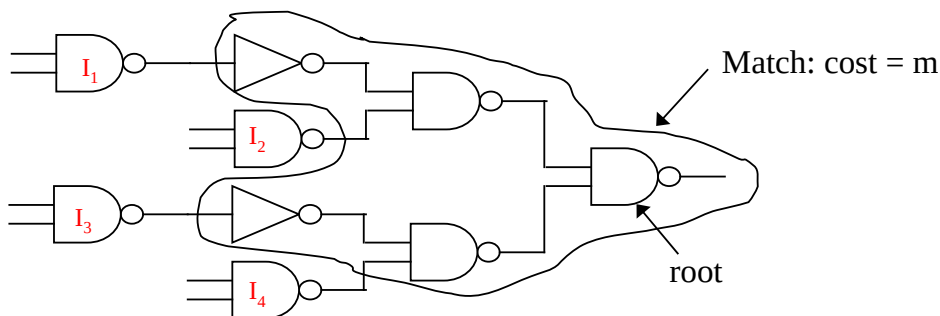
- Partition a subject graph into trees
 - Cut the graph at all multiple fanout points
- Optimally cover each tree using dynamic programming approach
- Piece the tree-covers into a cover for the subject graph



89

Technology Mapping DAGON Approach

- Principle of optimality: optimal cover for the tree consists of a match at the root plus the optimal cover for the sub-tree starting at each input of the match



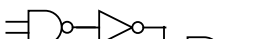
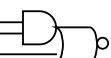
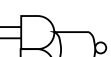
$$C(\text{root}) = m + C(I_1) + C(I_2) + C(I_3) + C(I_4)$$

cost of a leaf (i.e. primary input) = 0

90

Technology Mapping DAGON Approach

Example Library

INV	2		a'	
NAND2	3		$(ab)'$	
NAND3	4		$(abc)'$	
NAND4	5		$(abcd)'$	
AOI21	4		$(ab+c)'$	
AOI22	5		$(ab+cd)'$	

library element

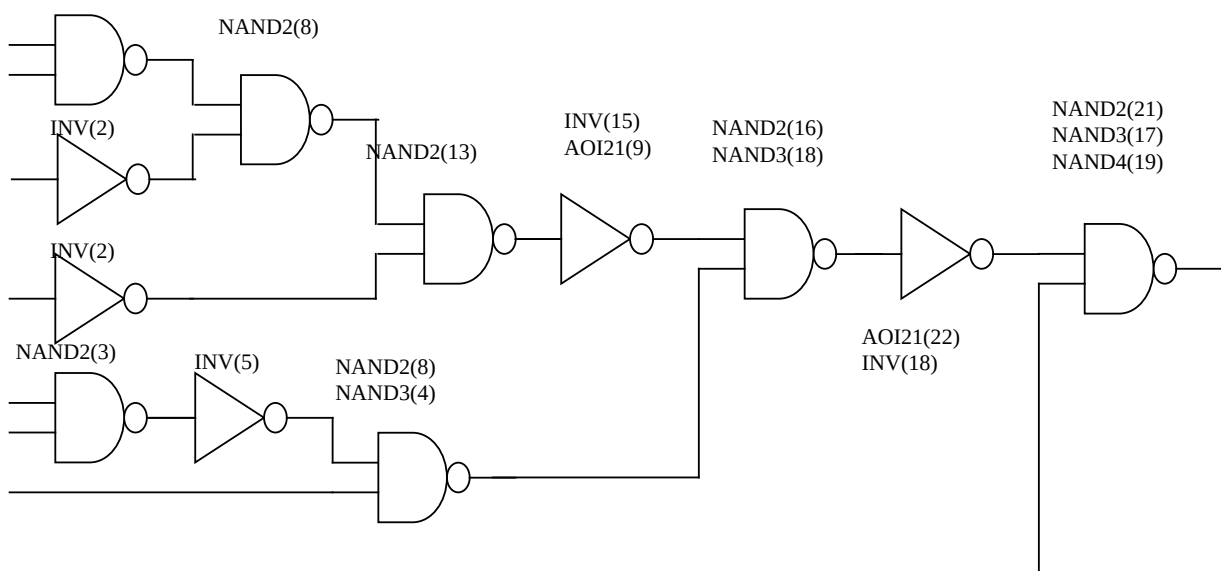
base-function representation

91

Technology Mapping DAGON Approach

Example

NAND2(3)



92

Technology Mapping DAGON Approach

- Complexity of DAGON for tree mapping is controlled by finding **all** sub-trees of the subject graph isomorphic to pattern trees
- **Linear** complexity in both the size of subject tree and the size of the collection of pattern trees
 - Consider library size as constant

93

Technology Mapping DAGON Approach

- | | |
|--|---|
| <ul style="list-style-type: none">□ Pros:<ul style="list-style-type: none">■ Strong algorithmic foundation■ Linear time complexity<ul style="list-style-type: none">□ Efficient approximation to graph-covering problem■ Give locally optimal matches in terms of both area and delay cost functions■ Easily “portable” to new technologies | <ul style="list-style-type: none">□ Cons:<ul style="list-style-type: none">■ With only a local (to the tree) notion of timing<ul style="list-style-type: none">□ Taking load values into account can improve the results■ Can destroy structures of optimized networks<ul style="list-style-type: none">□ Not desirable for well-structured circuits■ Inability to handle non-tree library elements (XOR/XNOR)■ Poor inverter allocation |
|--|---|

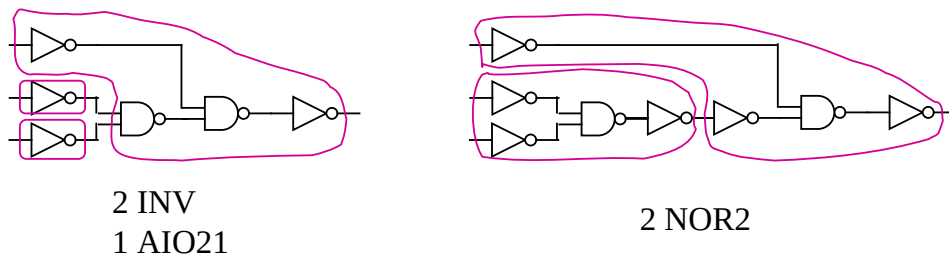
94

Technology Mapping

DAGON Approach

□ DAGON can be improved by

- Adding a pair of inverters for each wire in the subject graph
- Adding a pattern of a wire that matches two inverters with zero cost



95

Available Logic Synthesis Tools

□ Academic CAD tools:

- Espresso (heuristic two-level minimization, 1980s)
- MIS (multi-level logic minimization, 1980s)
- SIS (sequential logic minimization, 1990s)
- ABC (sequential synthesis and verification system, 2005-)
 - <http://www.eecs.berkeley.edu/~alanmi/abc/>

96