Principles Of
Digital Design

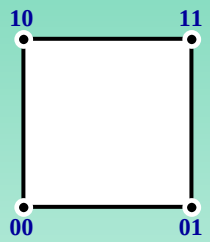
Chapter 4

Simplification of Boolean Functions

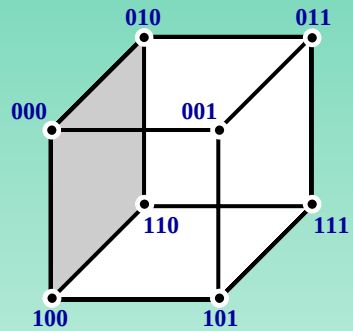
- *Karnaugh Maps*
- *Don’t Care Conditions*
- *Technology Mapping*
- *Optimization, Conversions, Decomposing, Retiming*

Boolean Cubes for n = 1, 2, 3, and 4

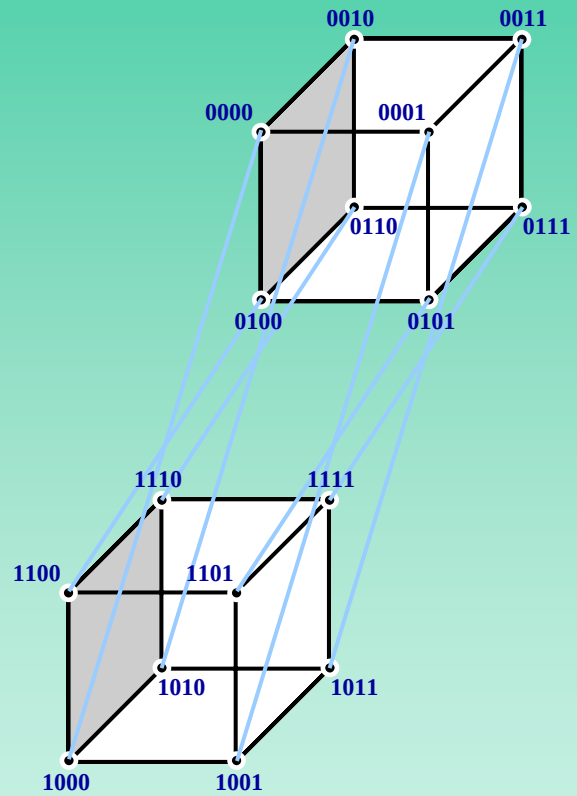
n = 1



n = 2



n = 3



n = 4

Boolean Functions and Boolean Cubes

- Each Boolean n-cube represents a Boolean function of n variables
- Each vertex represents a minterm
- Each m-subcube represents 2^m minterms, $m < n$, with the same $n - m$ literals
- Each m-subcube of 1-minterm represent a product of $n - m$ literals

$$= l_1 l_2 \ldots l_{n-m} (x'_{n-m+1} x'_{n-m+2} \ldots x'_n + x'_{n-m+1} x'_{n-m+2} \ldots x_n + \ldots + x_{n-m+1} x_{n-m+2} \ldots x_n)$$

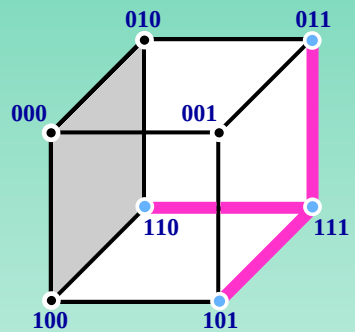
$$= l_1 l_2 \ldots l_{n-m}$$

- For any Boolean function a prime implicant is a subcube not contained in any other prime implicant
- As essential prime implicant is a subcube that contains a 1-minterm that is not included in any other prime implicant

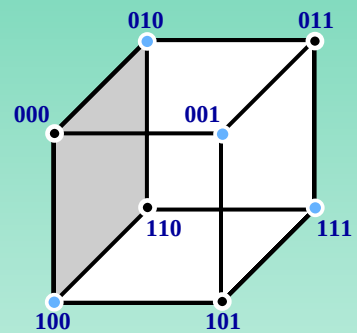
Representation of Carry and Sum Functions with Boolean Cubes

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Truth Table



Carry Function c_{i+1}

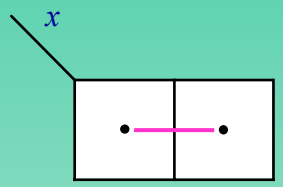


Sum Function s_i

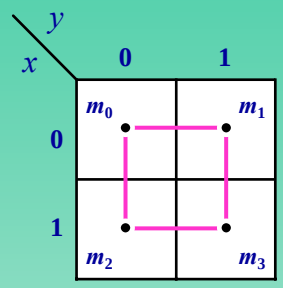
Map Representation

- (Karnaugh) maps define Boolean functions
- Map representation is equivalent to truth tables, Boolean expressions and Boolean cube representation
- Map aid in visually identifying prime implicants and essential prime implicants in each Boolean function
- Maps are used for manual optimization of Boolean functions

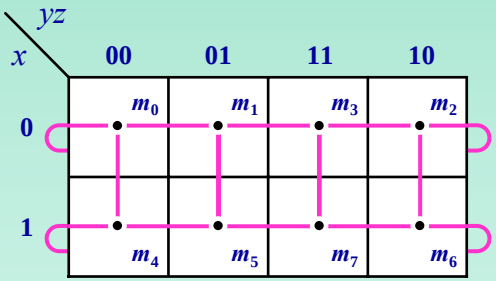
Boolean Subcubes and Corresponding Karnaugh Maps for $n = 1, 2, 3,$ and 4



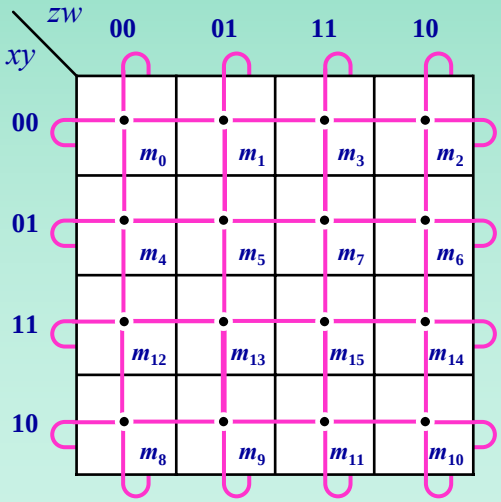
$n = 1$



$n = 2$

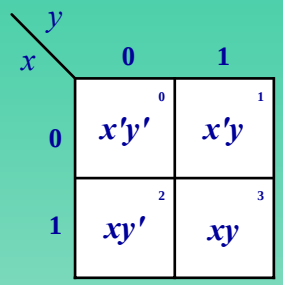


$n = 3$

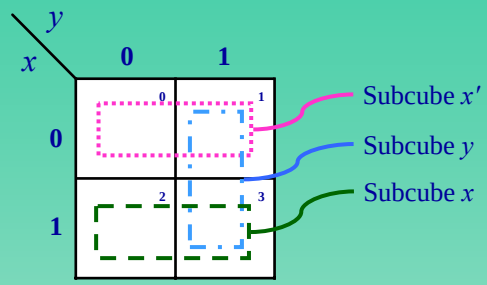


$n = 4$

2–variable Map



Map Organization

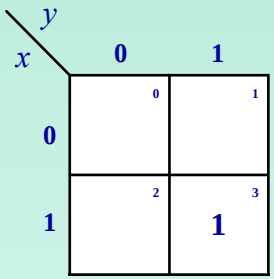


Example of 1-subcubes

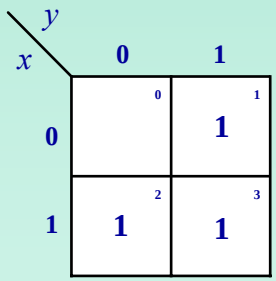
Example:

x	y	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

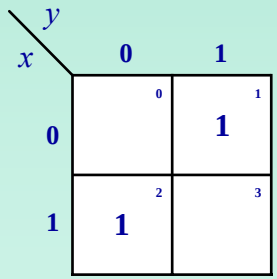
Truth Table



AND

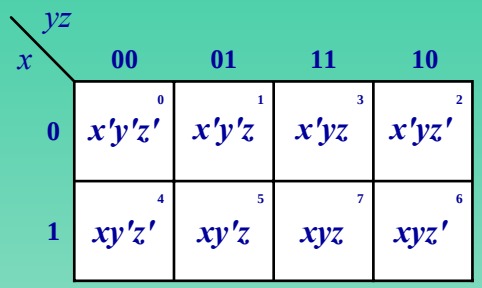


OR

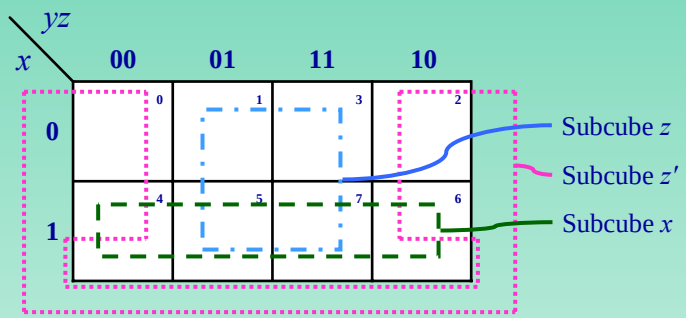


XOR

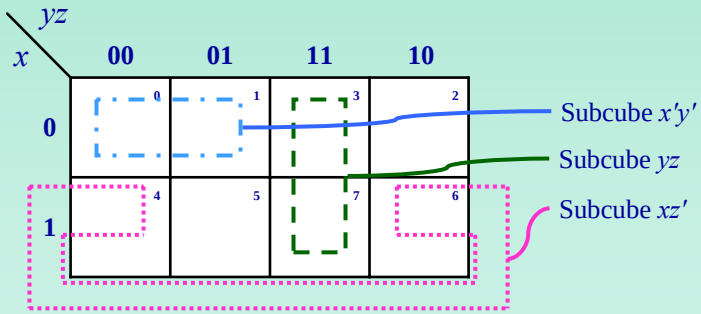
Three–variable Map



Map Organization



Example of 2-subcubes

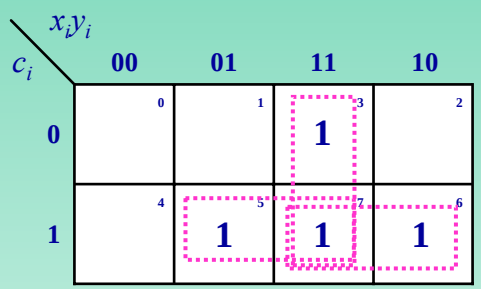


Example of 1-subcubes

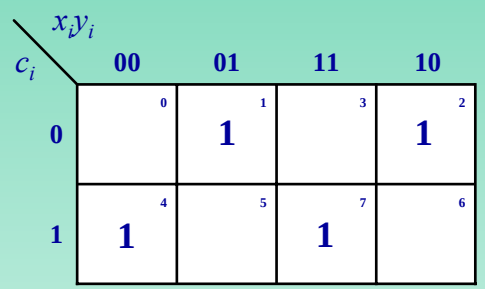
Map Representation of Carry and Sum Functions

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Truth Table



Carry Function c_{i+1}

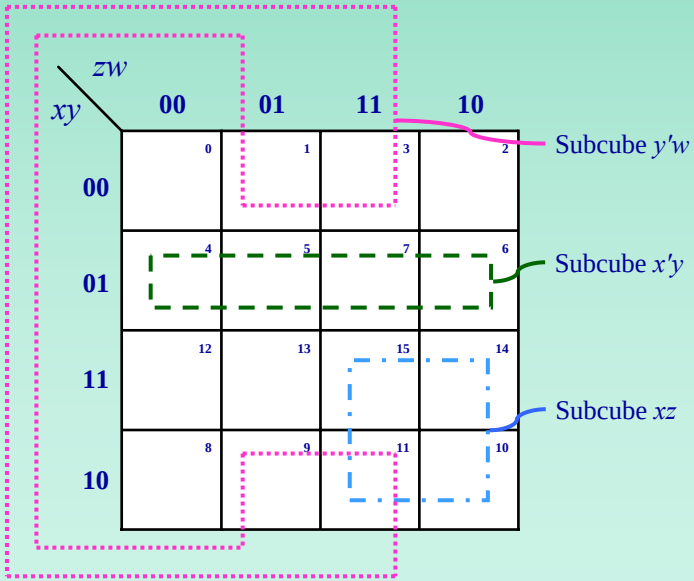


Sum Function s_i

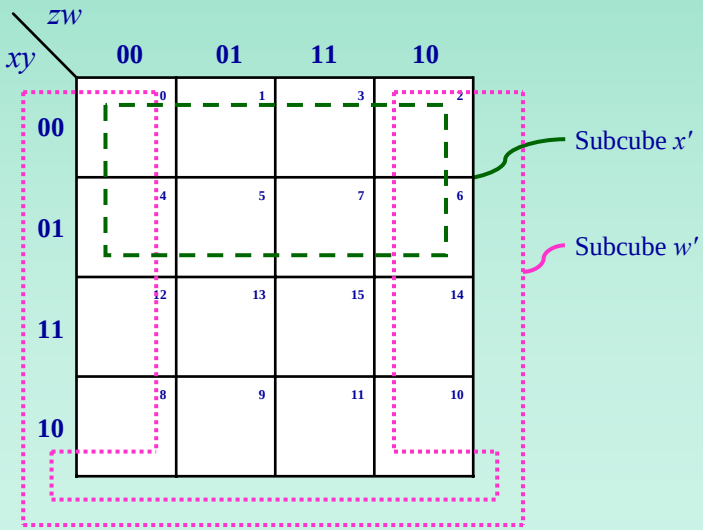
Four–variable Map

xy \ zw	00	01	11	10
00	0 $x'y'z'w'$	1 $x'y'z'w$	3 $x'y'zw$	2 $x'y'zw'$
01	4 $x'yz'w'$	5 $x'yz'w$	7 $x'yzw$	6 $x'yzw'$
11	12 $xyz'w'$	13 $xyz'w$	15 $xyzw$	14 $xyzw'$
10	8 $xy'z'w'$	9 $xy'z'w$	11 $xy'zw$	10 $xy'zw'$

Map Organization



Example of 2-subcubes

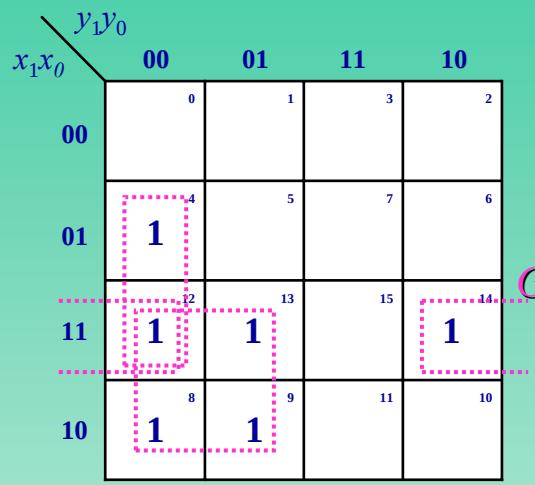


Example of 3-subcubes

Representation of Greater-than and Less-than Functions in Maps

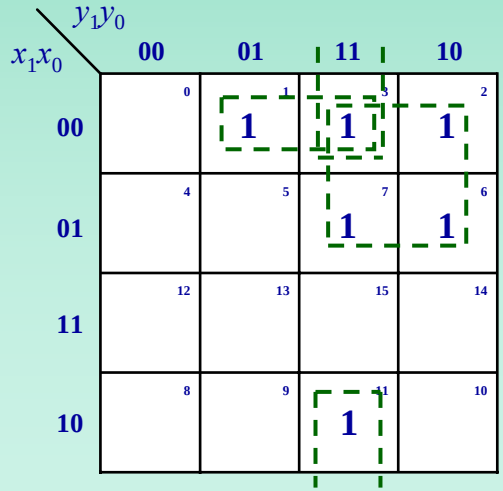
x_1	x_0	y_1	y_0	Greater Than	Equal	Less Than
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

Truth Table



$$G = x_1y_1' + x_0y_1'y_0' + x_1x_0y_0'$$

Greater-than Function



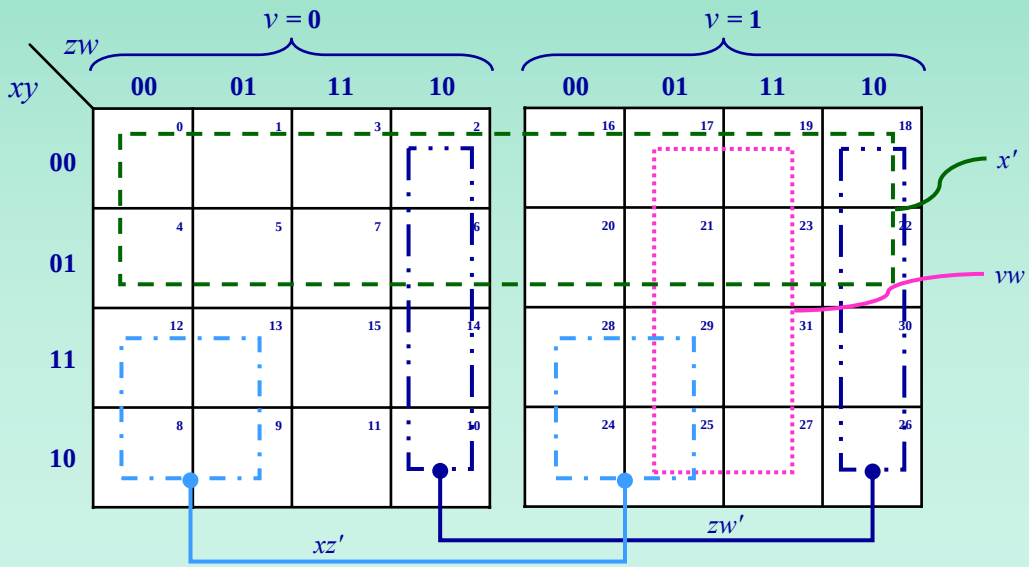
$$L = x_1'y_1 + x_1'x_0'y_0 + x_0'y_1y_0$$

Less-than Function

Five–variable Map

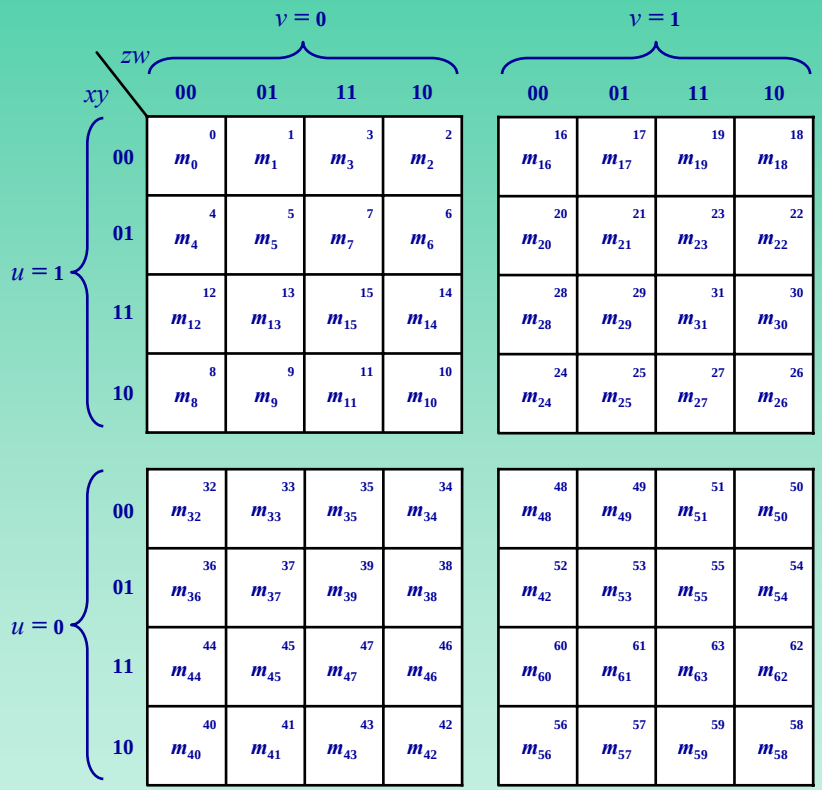
xy \ zw	v = 0: 00	01	11	10	v = 1: 00	01	11	10
00	0 x′y′z′w′v′	1 x′y′z′wv′	3 x′y′zwv′	2 x′y′zw′v′	16 x′y′z′w′v	17 x′y′z′wv	19 x′y′zwv	18 x′y′zw′v
01	4 x′yz′w′v′	5 x′yz′wv′	7 x′yzwv′	6 x′yzw′v′	20 x′yz′w′v	21 x′yz′wv	23 x′yzwv	22 x′yzw′v
11	12 xyz′w′v′	13 xyz′wv′	15 xyzwv′	14 xyzw′v′	28 xyz′w′v	29 xyz′wv	31 xyzwv	30 xyzw′v
10	8 xy′z′w′v′	9 xy′z′wv′	11 xy′zwv′	10 xy′zw′v′	24 xy′z′w′v	25 xy′z′wv	27 xy′zwv	26 xy′zw′v

Map Organization

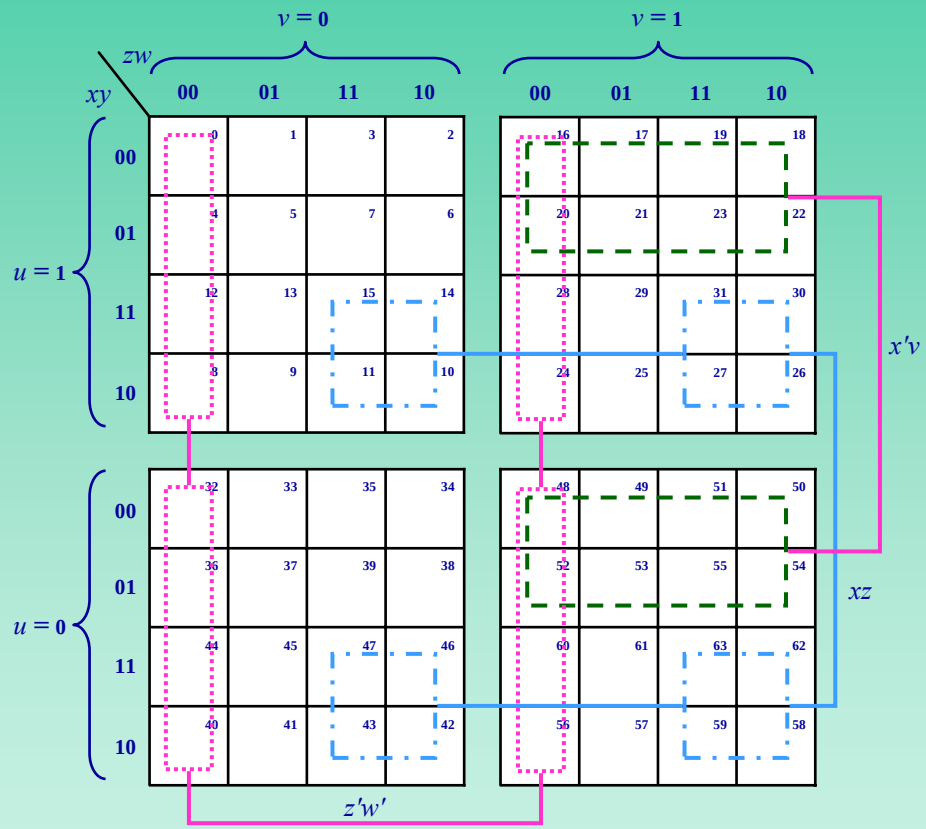


Example of 3-subsubes and 4-subcubes

Six–variable Map

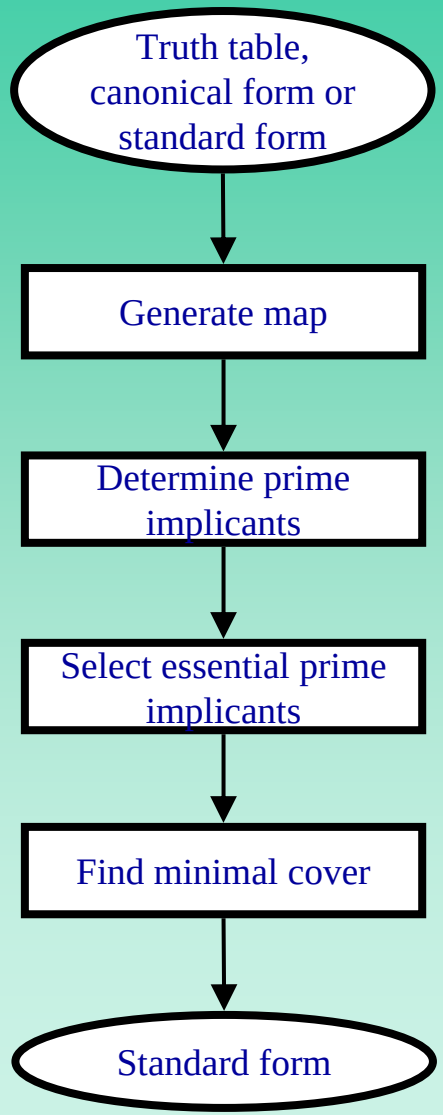


Map Organization



Example of 4-subcubes

Boolean Simplification with Map Method

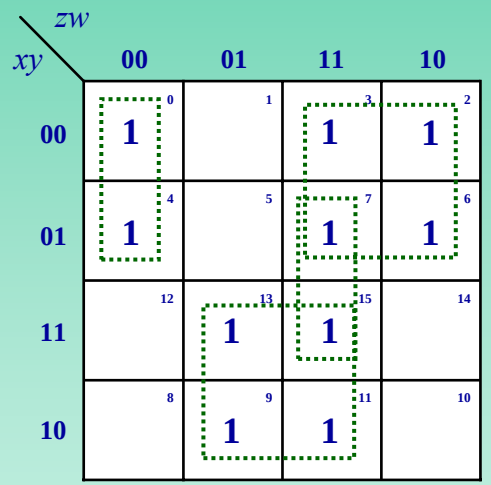


Boolean Simplification with Map Method

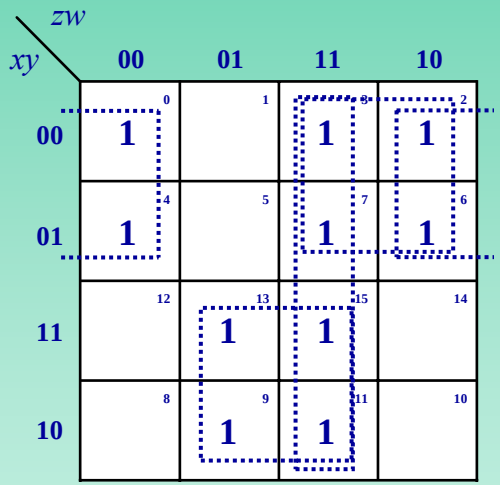
Example: Maps method

Problem: Using the map method, simplify the Boolean function

$$F = w'y'z' + wz + xyz + w'y$$



Map Organization



Prime Implicants in the Map

PI List: $w'z'$, wz, yz, $w'y$

EPI List: $w'z'$, wz

Cover List: (1) $w'z'$, wz, yz

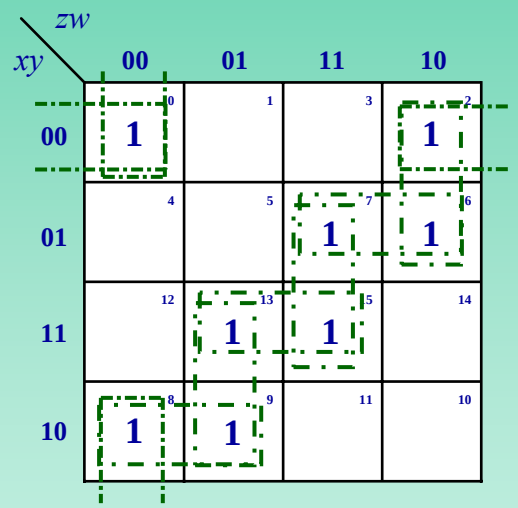
(2) $w'z'$, wz, $w'y$

Selection of Prime Implicants

Example: Selection of prime implicants

Problem: Simplify the Boolean function

$$F = w'x'yz' + w'xy + wxz + wx'y' + w'x'y'z'$$



PI List: $w'x'z', w'xy, wxz, wx'y', x'y'z', wy'z, xyz, w'yz'$

EPI List: empty

Cover List: (1) $w'x'z', w'xy, wxz, wx'y'$

(2) $x'y'z', wy'z, xyz, w'yz'$

Don’t–Care Conditions

- Completely specified functions have a value assigned for every minterm
- Incompletely specified functions do not have values assigned for some minterms which are called don’t–care minterms (d–minterms) or don’t–care conditions
- Don’t–care minterms can be assigned any value during simplifications in order to simplify Boolean expressions

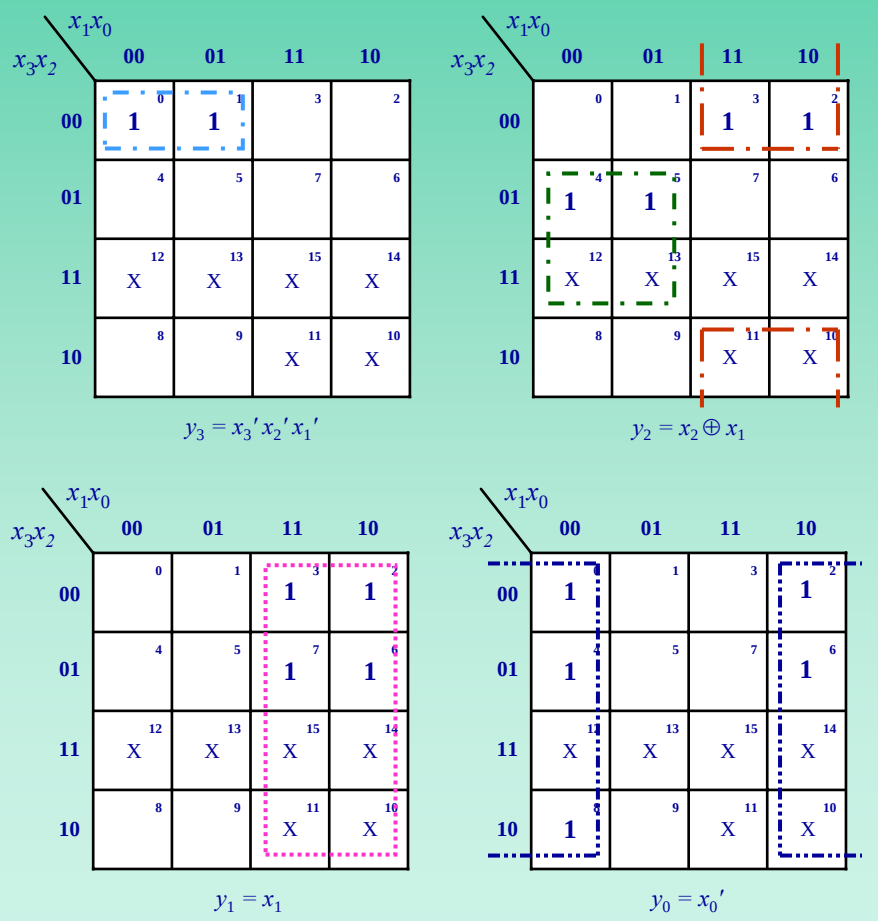
Don’t–Care Conditions

Example: Don’t–care conditions

Problem: Derive Boolean expressions for the 9’s complement of a BCD digit

Digits					Nine’s Complements				
Decimal	BCD				Decimal	BCD			
	x_3	x_2	x_1	x_0		x_3	x_2	x_1	x_0
0	0	0	0	0	9	1	0	0	1
1	0	0	0	1	8	1	0	0	0
2	0	0	1	0	7	0	1	1	1
3	0	0	1	1	6	0	1	1	0
4	0	1	0	0	5	0	1	0	1
5	0	1	0	1	4	0	1	0	0
6	0	1	1	0	3	0	0	1	1
7	0	1	1	1	2	0	0	1	0
8	1	0	0	0	1	0	0	0	1
9	1	0	0	1	0	0	0	0	0

Nine’s–Complement Table

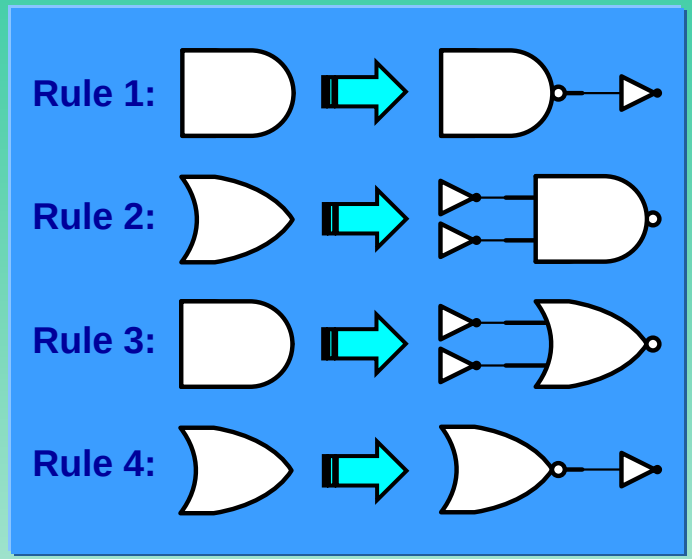


Map Representation

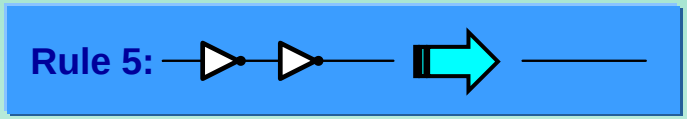
Technology Mapping for Gate Arrays

- Gate arrays contain only one type of m-input gate (such as 3-input NOR, 3-input NAND)
- Technology mapping is a transformation of Boolean expressions into a logic schematic containing only this type of gate
- Technology mapping consist of three tasks
  - Conversion replaces each operator with an operator representing the gate function given in the gate array
  - Optimization eliminates unnecessary inverters
  - Decomposition replaces a n-input gate with an m-input gate available in the gate array

Conversion and Optimization



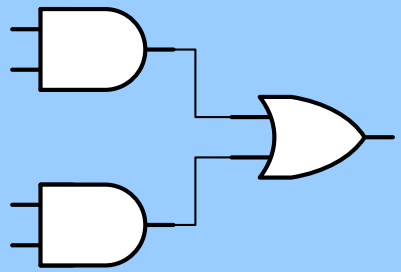
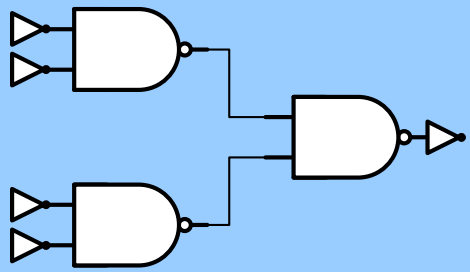
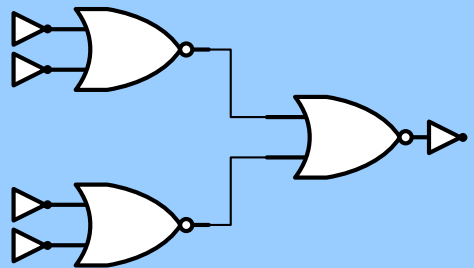
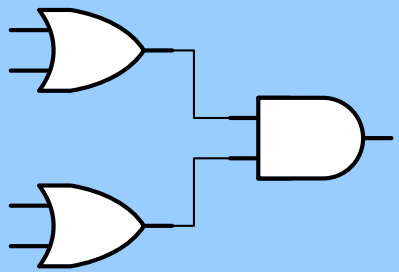
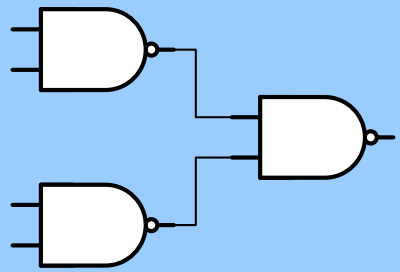
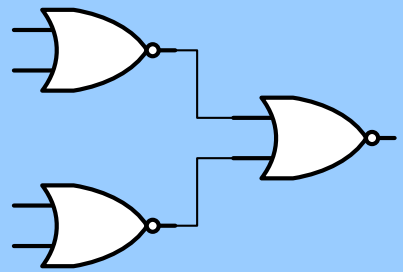
Conversion Rules



Optimization Rules

- **Conversion Procedure:**
  - **Replace AND and OR gates with NAND or NOR gates by using Rules 1 – 4, and eliminate double inverters whenever possible**

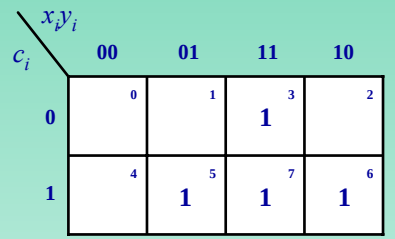
Translation of Standard Terms to NAND and NOR Schematics

Form Type	Standard Form Implementation	NAND Implementation	NOR Implementation
Sum of products			
Product of sums			

Conversion to NAND (NOR) Gates

Example: Conversion to NAND (NOR) gates

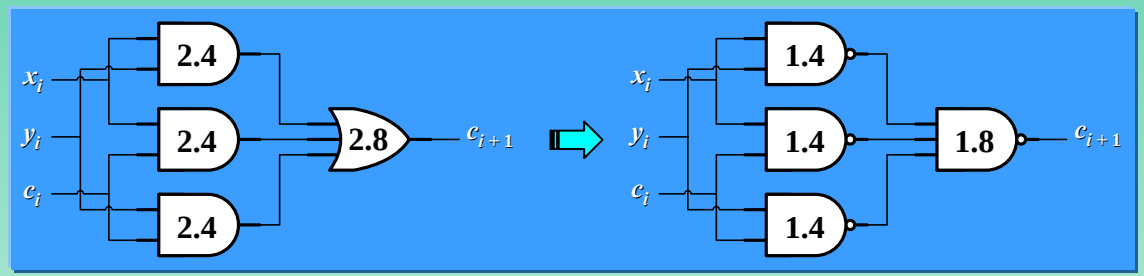
Problem: Derive the NAND and NOR implementations of the carry function



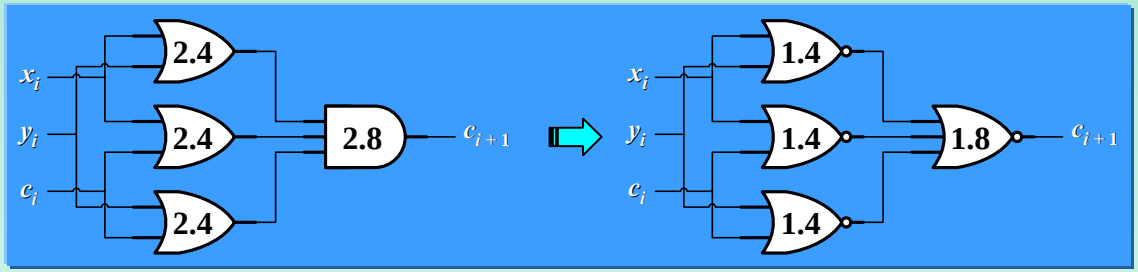
Map Definition Carry Function c_{i+1}

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$
$$c_{i+1} = (x_i + y_i)(x_i + c_i)(y_i + c_i)$$

Standard Forms



NAND Implementation

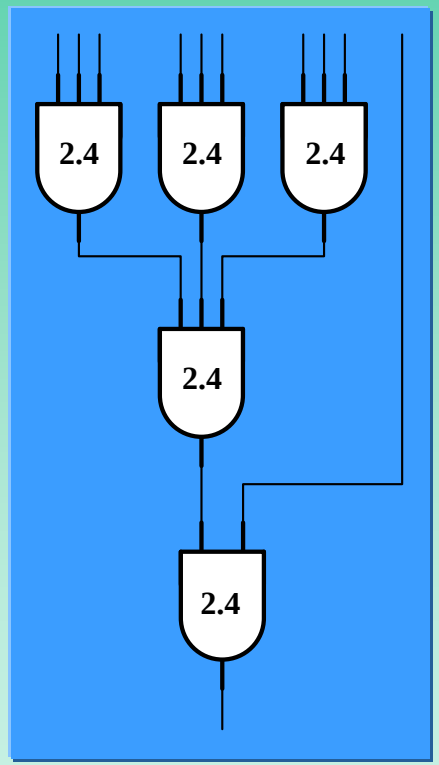


NOR Implementation

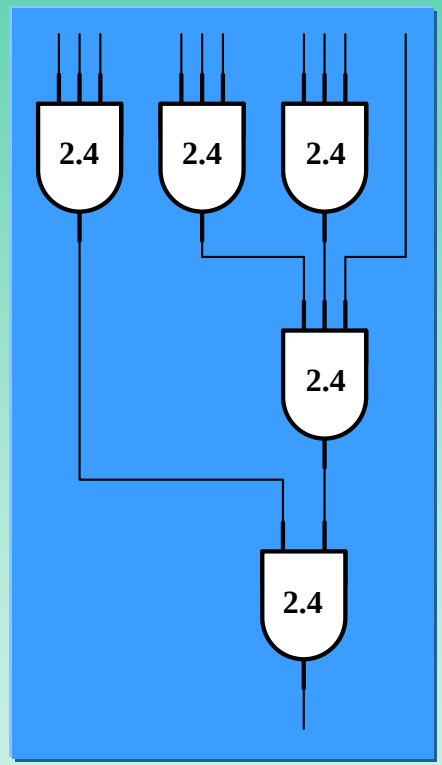
Decomposition of 10–input AND Gate into 3–input AND Gates

Level Number	Number of Inputs	Number of Gates
1	10	[10 / 3] = 3
2	3 + (10 – 3([10 / 3])) = 4	[4 / 3] = 1
3	1 + (4 – 3([4 / 3])) = 2	[2 / 3] = 1

Input and Gate Computation on Each Level



One Possible Decomposition



Alternative Decomposition

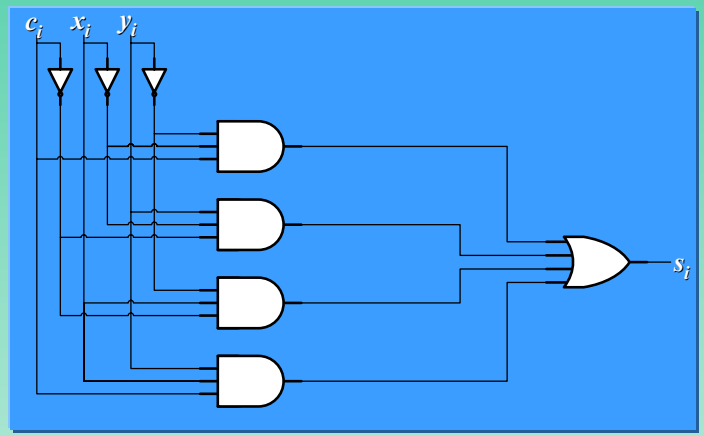
Technology Mapping for Gate Arrays

Example: Technology mapping for gate arrays

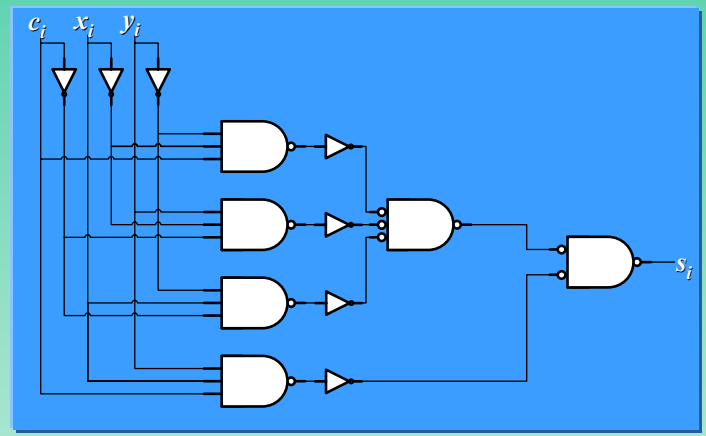
Problem: Implement the sum function using 3–input NAND gates

c_i \ $x_i y_i$	00	01	11	10
0	0	1	3	1
1	1	5	1	6

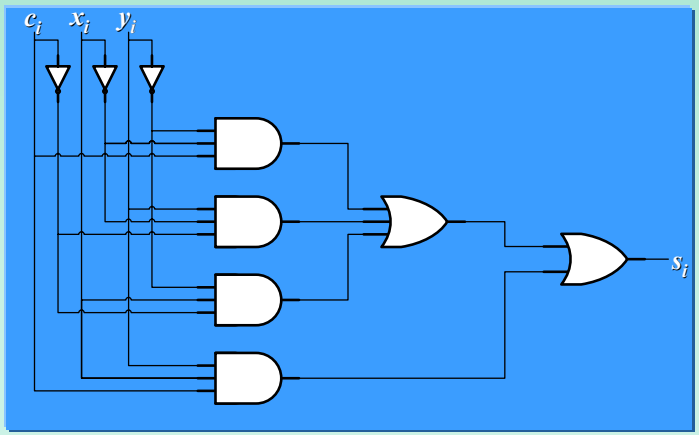
Map Definition Sum Function s_i



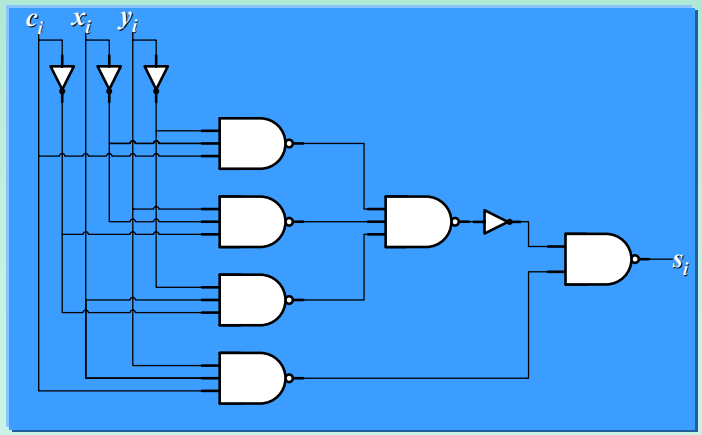
AND–OR Implementation



Conversion to NAND Network



OR Gate Decomposition



Optimized NAND Network

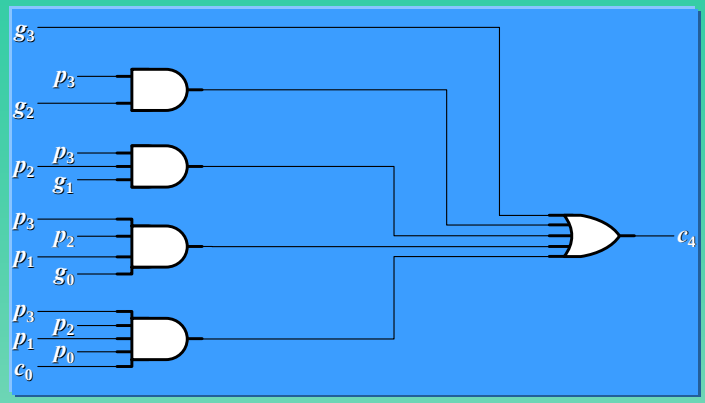
Design Retiming

Example: Design retiming

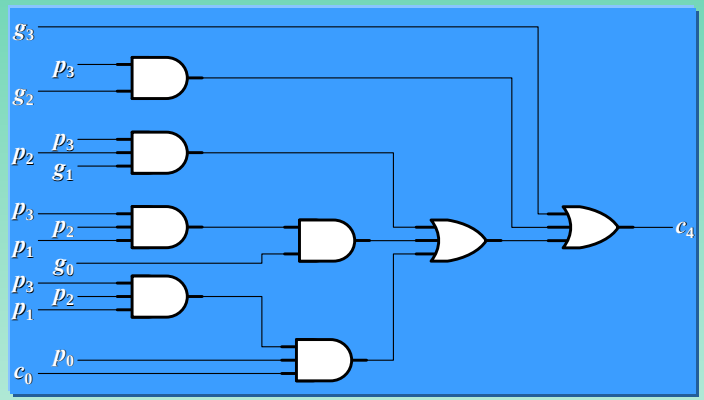
Problem: Implement 4–bit carry-look-ahead function

$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$

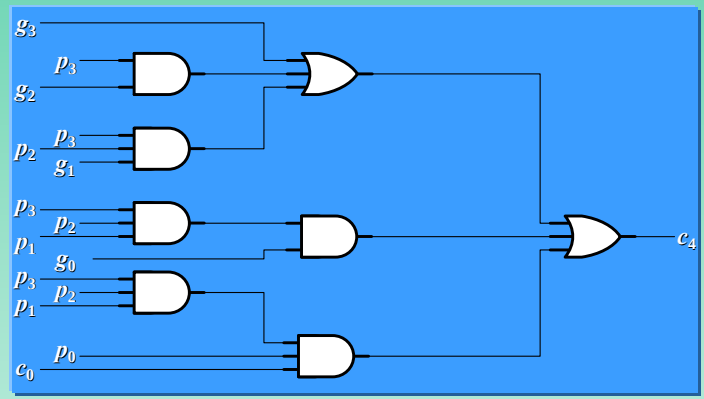
using 3–input NAND gates



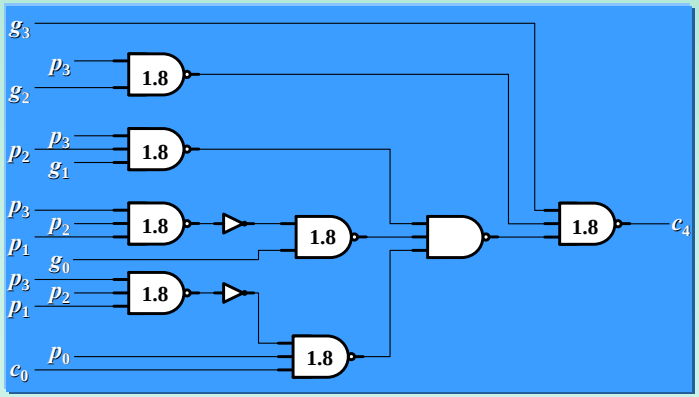
AND-OR Implementation



Decomposition of AND-OR Implementation

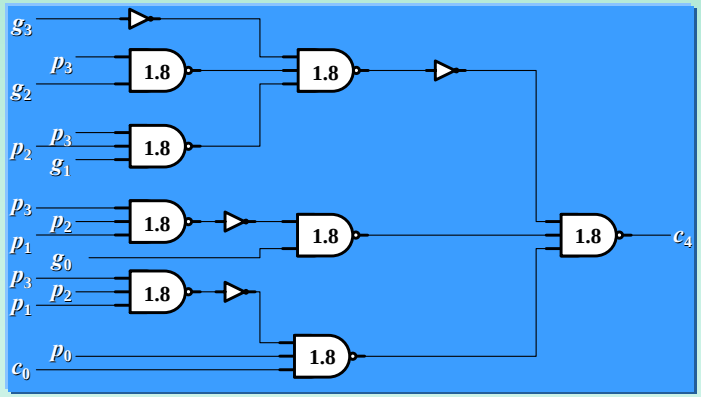


Performance Optimized Decomposition



NAND Implementation of Above

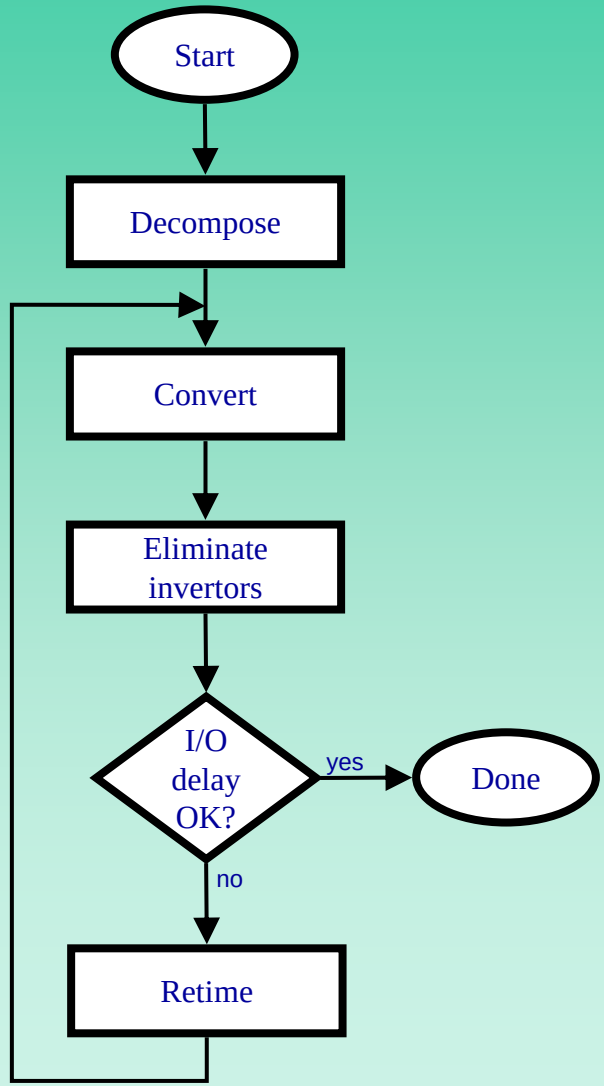
Delay = 8.2ns



Performance Optimized NAND Implementation

Delay = 6.4ns

Technology Mapping Procedure for Gate Arrays



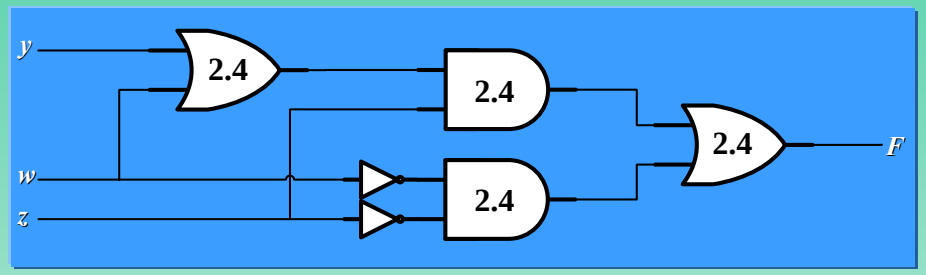
Technology Mapping for Custom Libraries

- Libraries contain gates with different functions and different delays
- Technology mapping means covering schematic with library gates
- Minimize delay on critical paths
- Minimize cost on non-critical paths

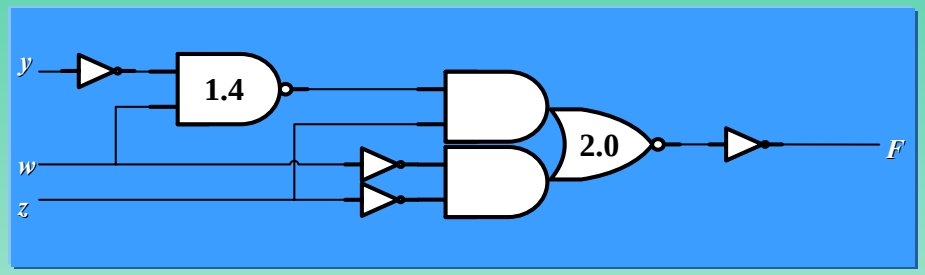
Technology Mapping for Custom Libraries

Example: Technology mapping for custom libraries

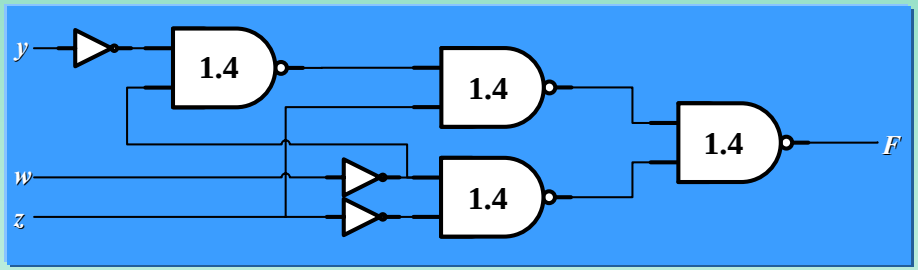
Problem: Convert the expression $w'z' + z(w + y)$ into a logic schematic using any of the gates defined in the digital logic gates, multiple-input gates, and complex gates libraries



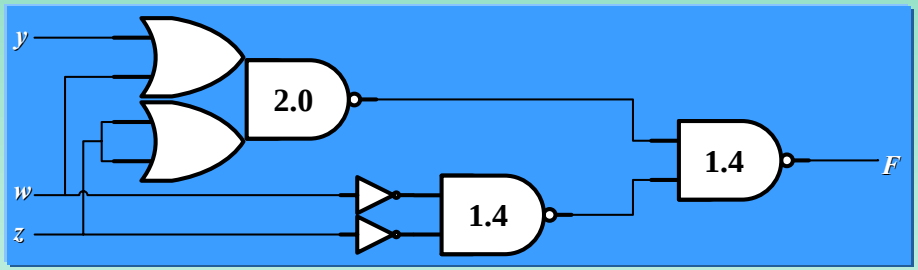
AND–OR Implementation (Delay = 7.2ns, Cost = 28)



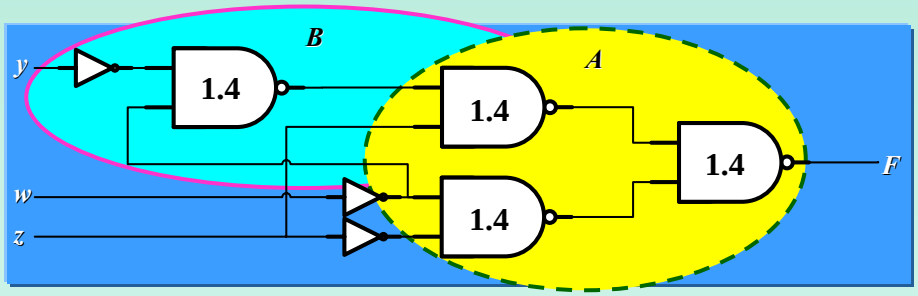
Alternative A (Delay = 5.4ns, Cost = 20)



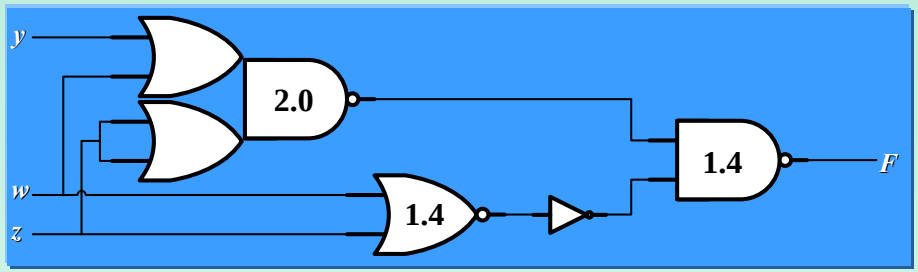
NAND Implementation (Delay =5.2ns, Cost = 22)



Alternative B (Delay = 3.8ns, Cost = 20)

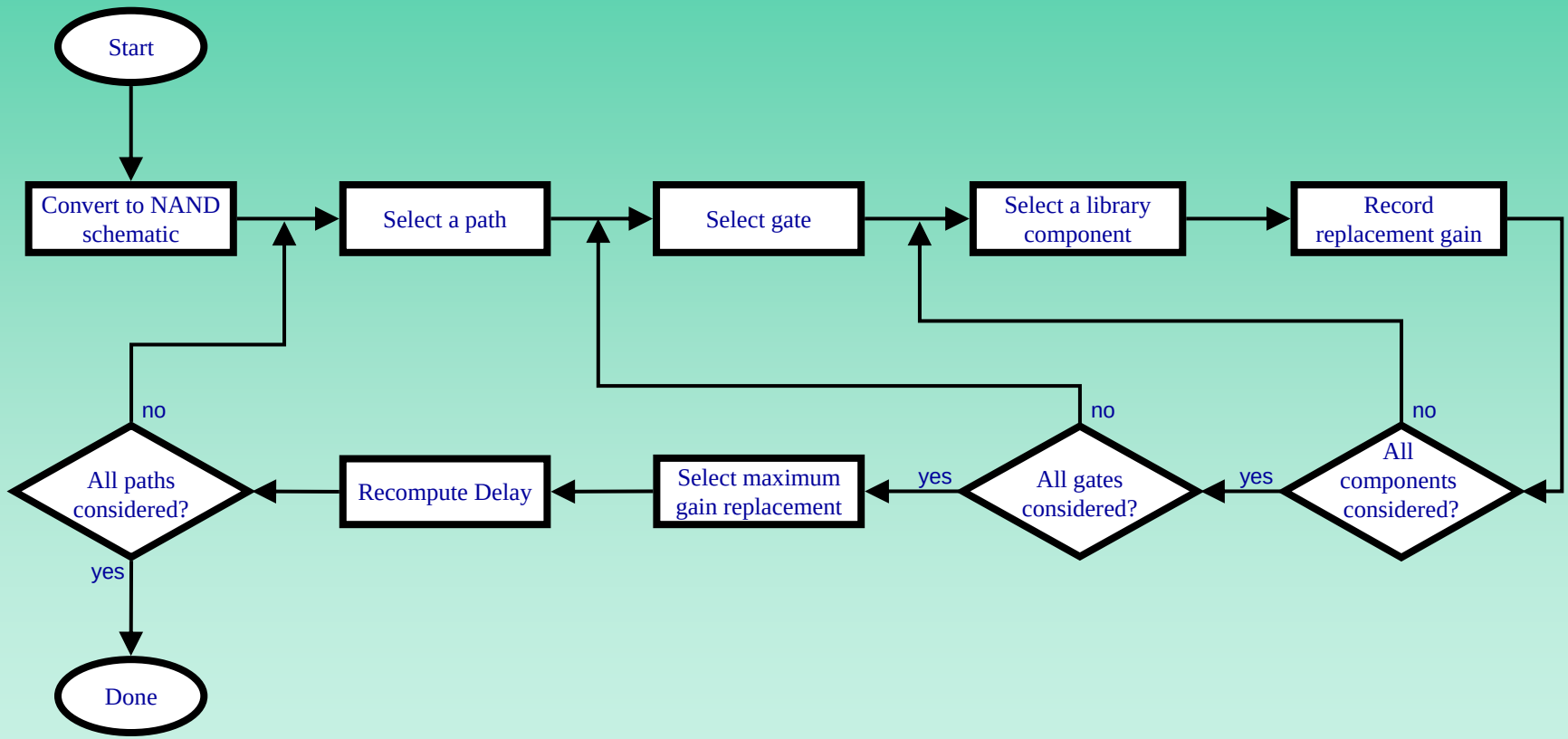


Two Possible Conversions



Cost Optimized Alternative B (Delay = 3.8ns, Cost = 18)

Conversion Procedure for Custom Libraries



Chapter Summary

- Simplification of Boolean functions by
  - Map method (visual)
- Technology mapping for gate arrays
  - Decomposition
  - Conversion
  - Optimization
  - Retiming
- Technology mapping for custom libraries by schematic covering with complex gates with
  - Time optimization on circuit paths
  - Cost optimization on non-critical paths