ECSE 323
Digital System Design

Combinational Circuits

Prof. Warren Gross

Material used in this set of slides was based on “*Fundamentals of Digital Logic with VHDL Design*”
by S. Brown and Z. Vranesic

Combinational Circuits

- The values of the outputs of a *combinational circuit* depend only on the values of the signals applied to the inputs
  - (In later chapters we will study circuits whose output depend not only on the values of the input, but also on the past behavior of the circuit….)

Multilevel Synthesis (4.6/4.7)

- We have so far seen two-level realizations of logic functions (assuming the availability of complements at the input)
  - AND stage followed by an OR stage (SOP)
  - OR stage followed by and AND stage (POS)
  - Equivalent NAND-NAND or NOR-NOR networks

- In many technologies, it is not efficient to build the wide gates necessary to implement large functions
  - the "*fan-in*" of the gates available to the designer is not high enough
  - FPGA: array of simple elements that usually have at most 2 - 5 inputs
  - CMOS: Can only put a few transistors in series, say at most 5

Factoring

Implementation technology dictates strategy

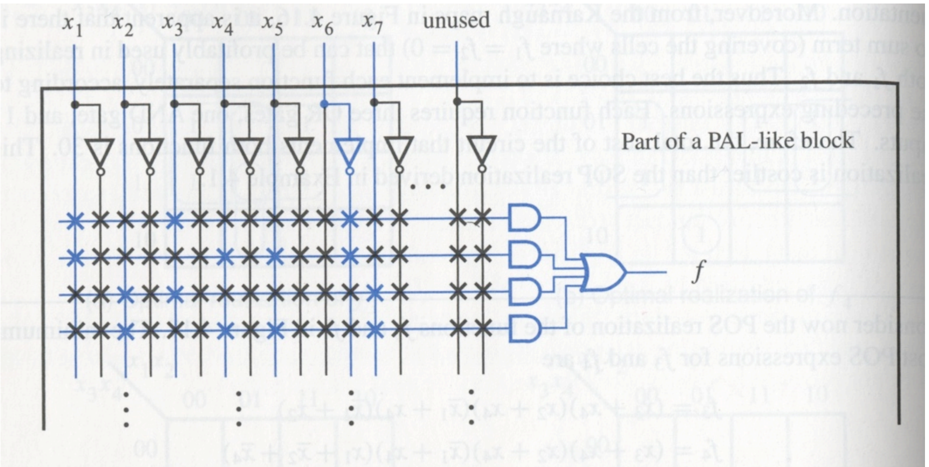


Figure 4.18 Implementation in a CPLD.

Factoring

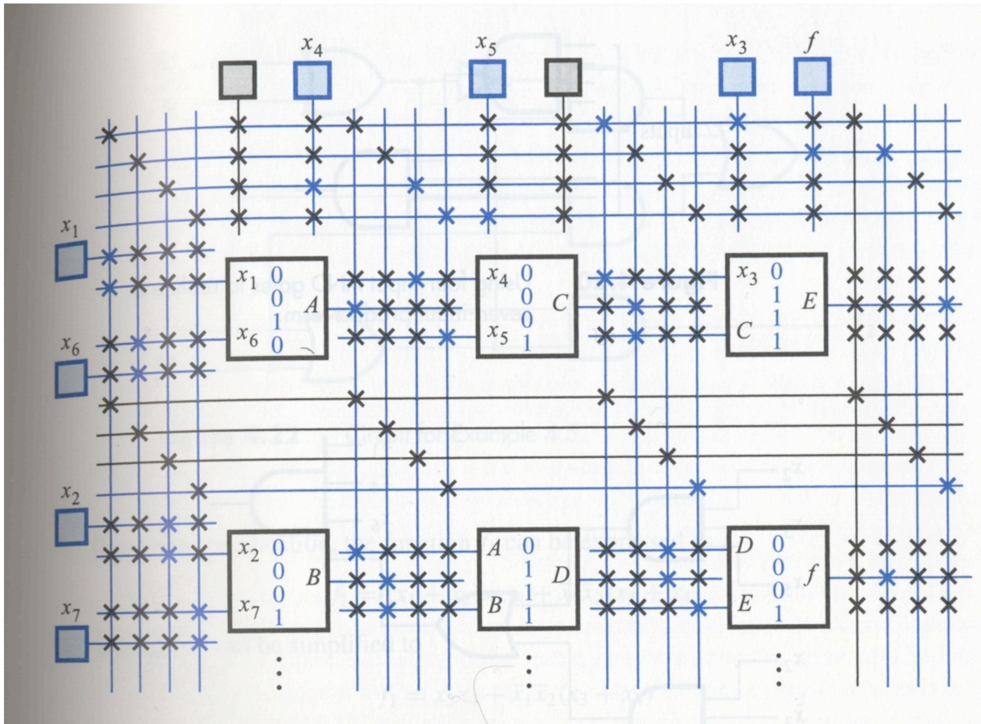


Figure 4.19 Implementation in an FPGA.

Each LUT has a fan-in of 2; need to factor the expression accordingly.

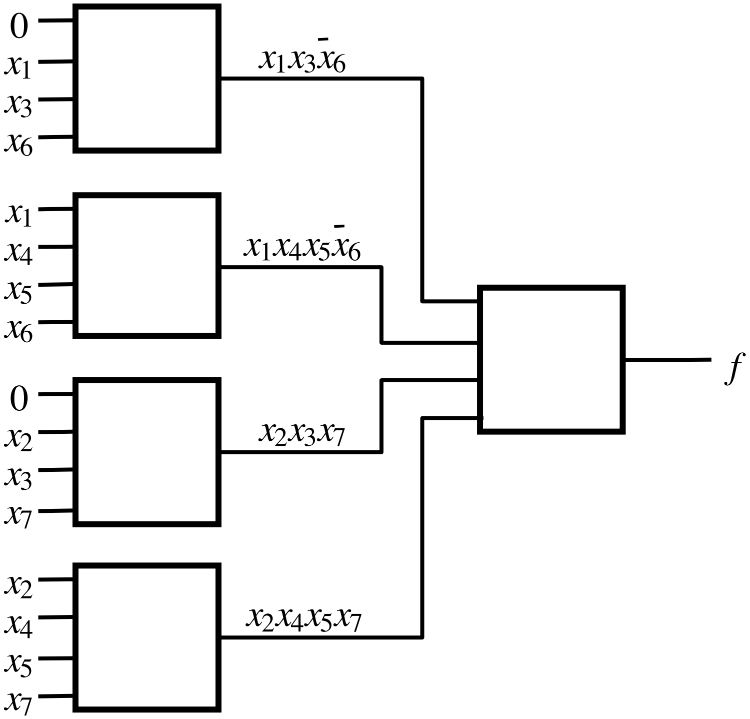
Impact on Wiring Complexity

(same example as previous, fan-in = 4)

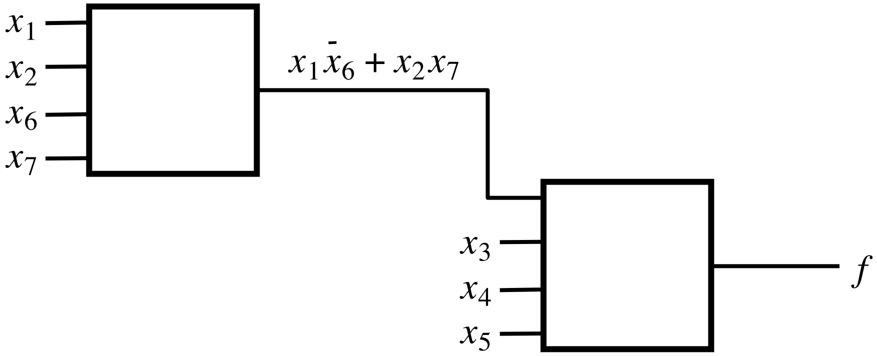
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func3 IS
            PORT ( x1, x2, x3, x4, x5, x6, x7 : IN      STD_LOGIC ;
                     f                                  : OUT  STD_LOGIC ) ;
END func3 ;

ARCHITECTURE LogicFunc OF func3 IS
BEGIN
            f <= (x1 AND x3 AND NOT x6) OR
                 (x1 AND x4 AND x5 AND NOT x6) OR
                 (x2 AND x3 AND x7) OR
                 (x2 AND x4 AND x5 AND x7) ;
END LogicFunc ;
```

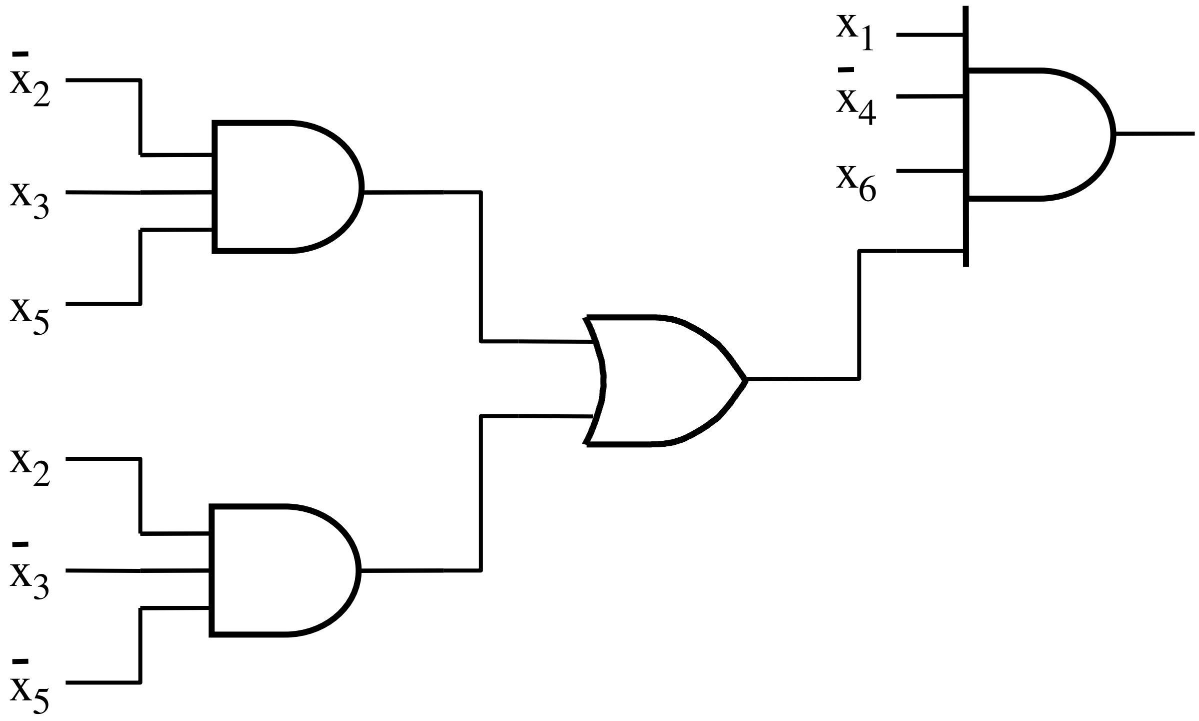


(a) Sum-of-products realization



(b) Factored realization

Another Example



e.g. Maximum fan-in of 4

See Example 4.5 in the text for another example of factorization

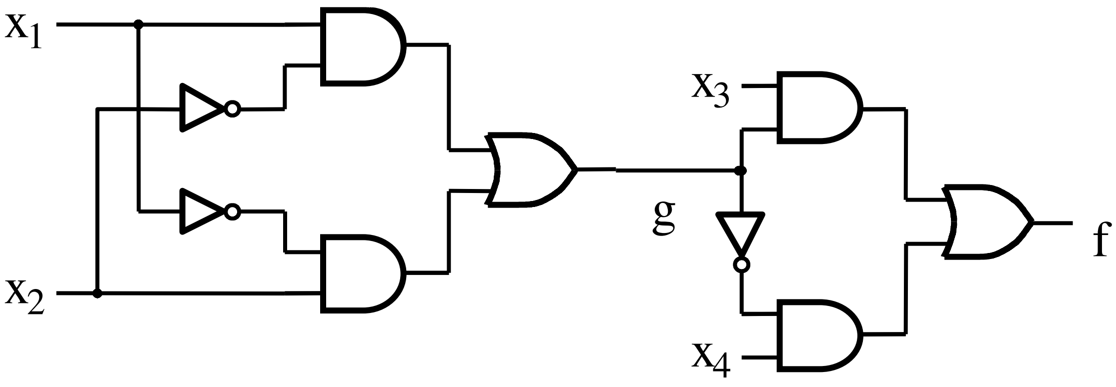
Multilevel circuits usually imply longer propagation delay

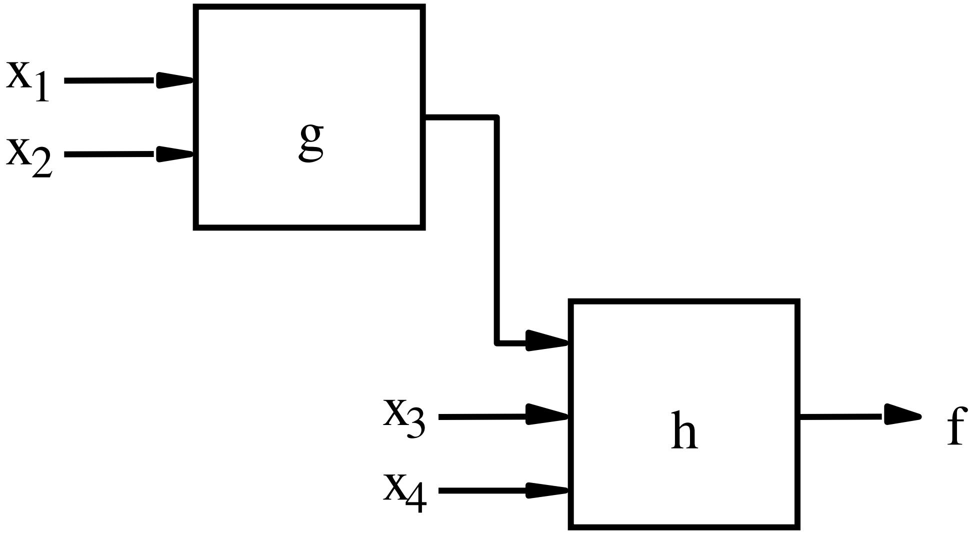
Decomposition

$$f = \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 x_4 + \overline{x}_1 \overline{x}_2 x_4$$

Factor common terms…

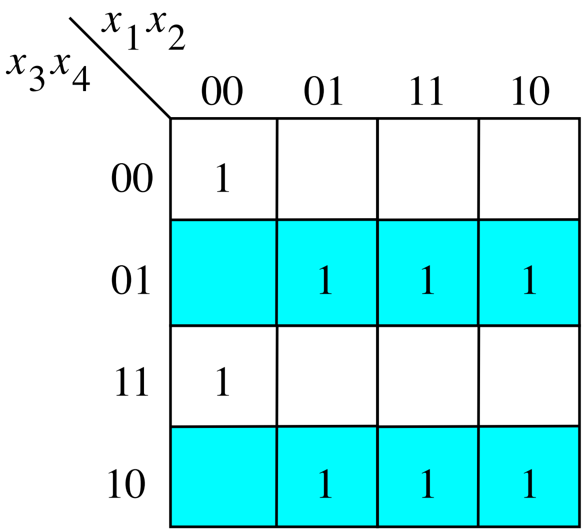
$$f = (\overline{x}_1 x_2 + x_1 \overline{x}_2) x_3 + (x_1 x_2 + \overline{x}_1 \overline{x}_2) x_4$$





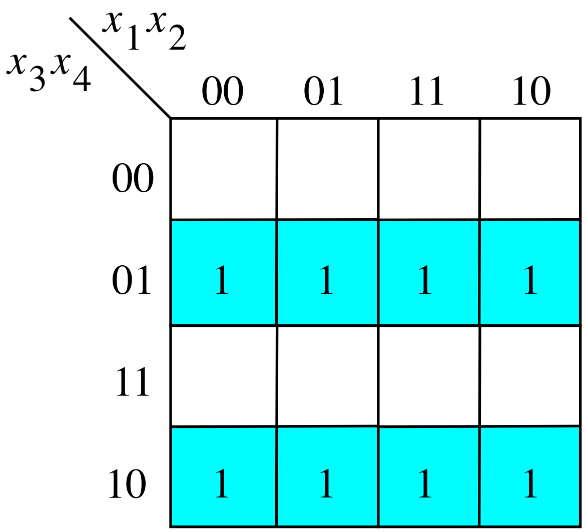
Disjoint decompostion

Another Example



$x_3x_4 \backslash x_1x_2$	00	01	11	10
00	1			
01		1	1	1
11	1			
10		1	1	1

$x_5 = 0$



$x_3x_4 \backslash x_1x_2$	00	01	11	10
00				
01	1	1	1	1
11				
10	1	1	1	1

$x_5 = 1$

Look for patterns in Karnaugh map

Each pattern only depends on the variables that define columns in each row

e.g. Blue pattern is a subfunction

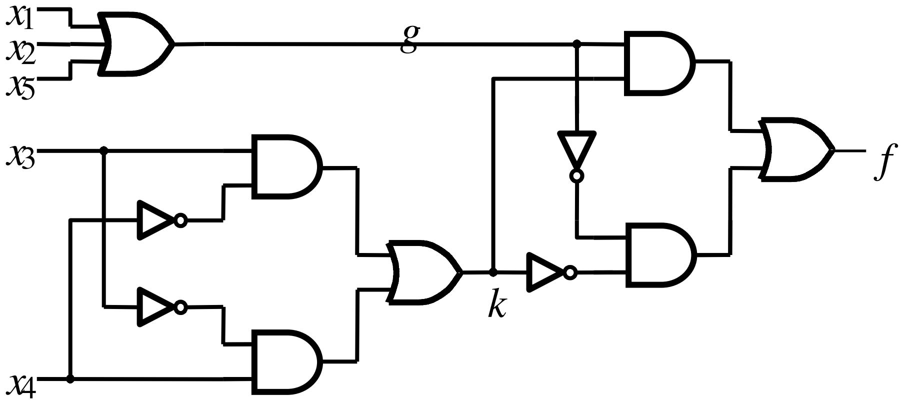
Another Example

x_3x_4 \ x_1x_2	00	01	11	10
00	1			
01		1	1	1
11	1			
10		1	1	1

$x_5 = 0$

x_3x_4 \ x_1x_2	00	01	11	10
00				
01	1	1	1	1
11				
10	1	1	1	1

$x_5 = 1$



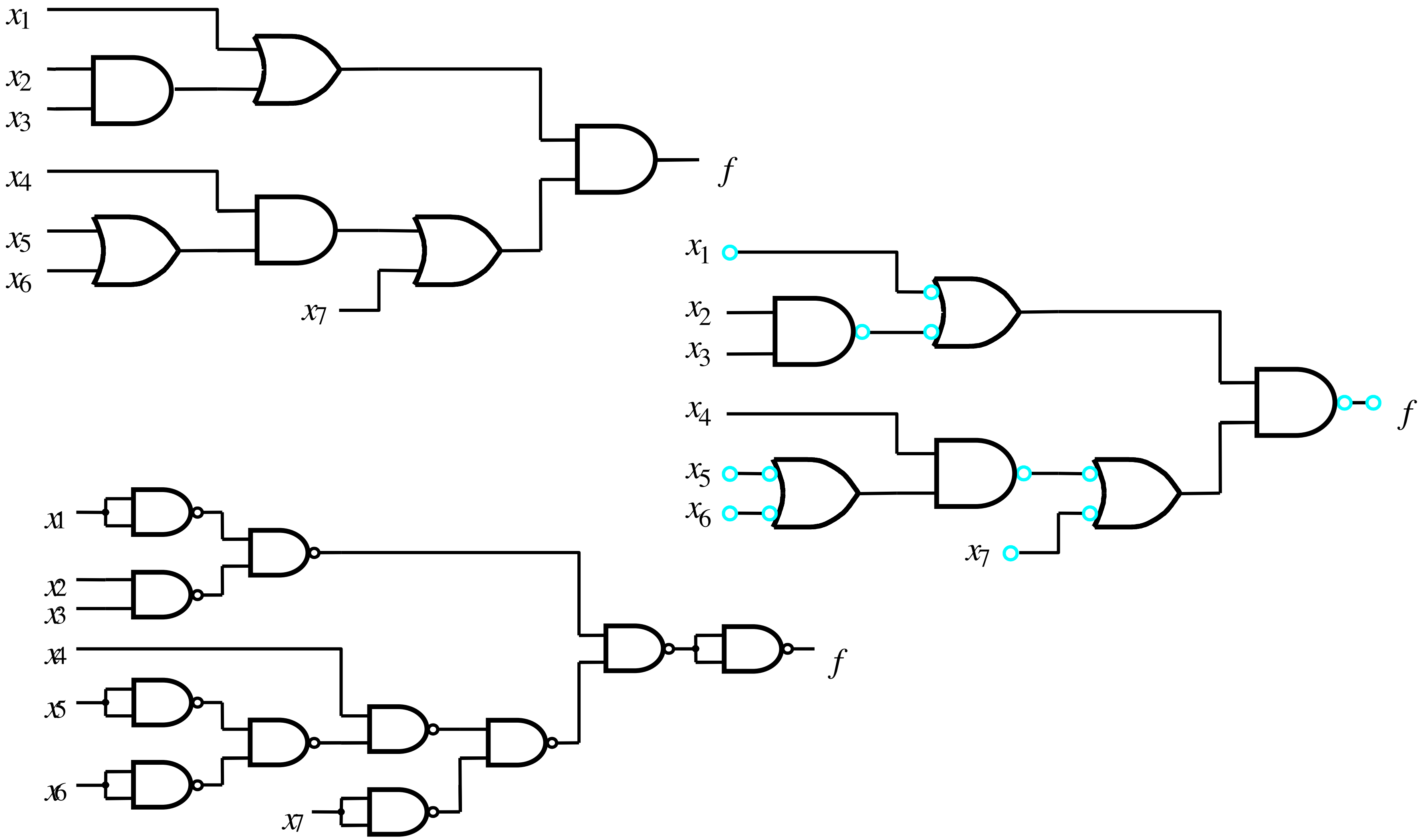
$g(x_1,x_2,x_5) = x_1 + x_2 + x_5$

$k(x_3,x_4) = x_3'x_4 + x_3x_4'$

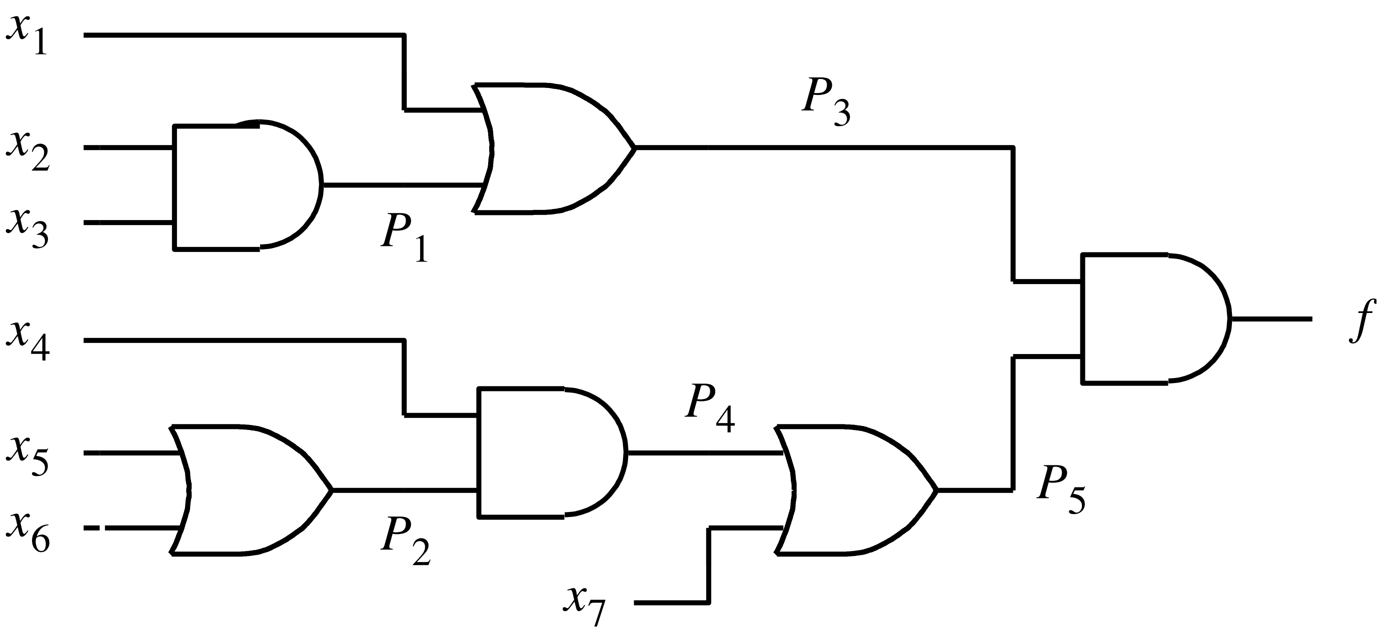
$f(x_1,x_2,x_3,x_4,x_5) = h[g(x_1,x_2,x_5),k(x_3,x_4)]$
$= kg + k'g'$

11 gates (including input inverters) and 19 inputs, max fan in = 3
2-level minimal SOP: 14 gates, 41 inputs, max fax in = 8

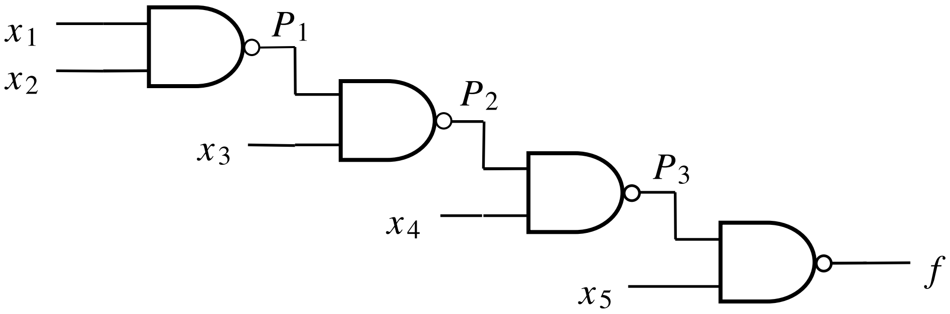
Multilevel NAND and NOR Circuits



Analysis



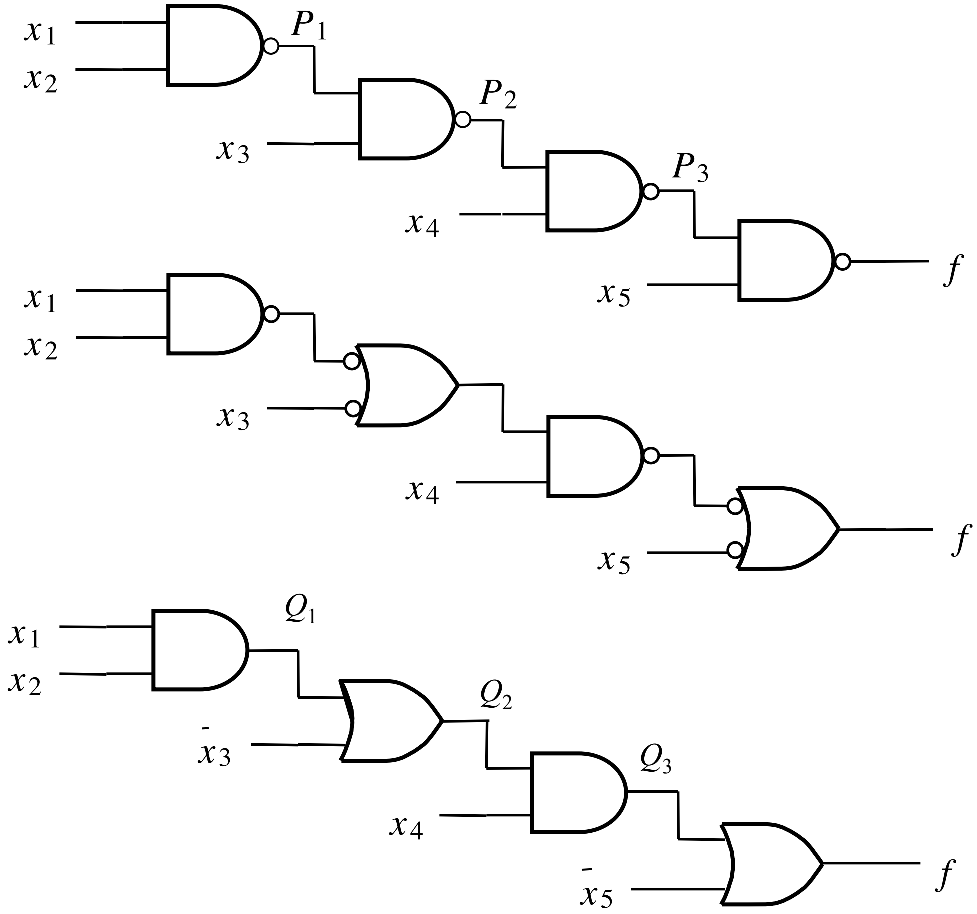
Interpretation of SOP form from NAND-NAND circuit



$$f = \overline{x_5\overline{(x_4\overline{(x_3\overline{(x_2\,x_1)}}}}$$

Successive application of de Morgan yields an unwieldy expression.

Expression greatly simplified by concerting selected NAND-NAND to AND-OR



NAND = OR with inputs inverted

Substitute

Remove redundant inverters

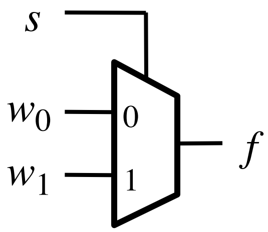
$$f = \overline{x}_5 + (x_4(\overline{x}_3 + (x_1 x_2)))$$

$$f = \overline{x}_5 + \overline{x}_3 x_4 + x_1 x_2 x_4$$

Combinational Circuit Building Blocks

- Chapter 6
- There are a few basic building blocks commonly used to build up larger circuits
  - Multiplexers / Demultiplexers
  - Decoders / Encoders
  - Arithmetic circuits

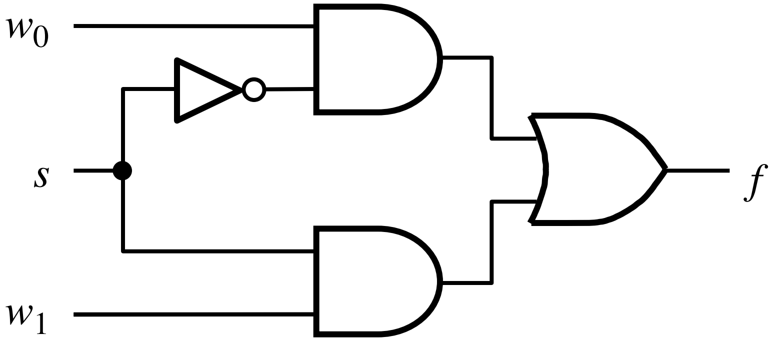
Multiplexers



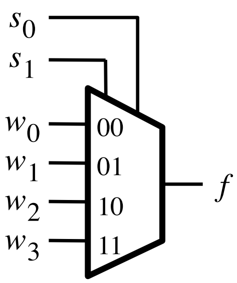
(a) Graphical symbol for 2-to-1 MUX

s	f
0	w_0
1	w_1

(b) Truth table



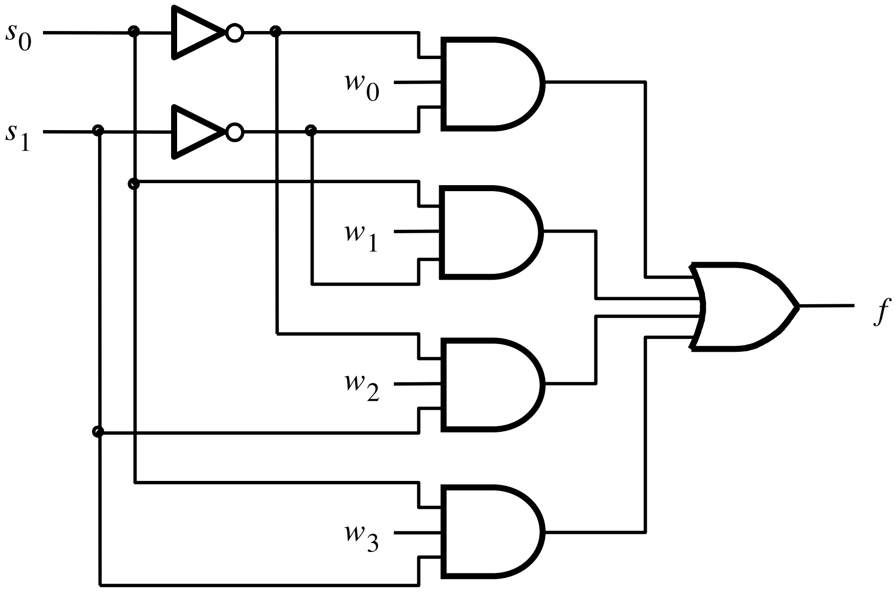
(c) Sum-of-products circuit



(a) Graphic symbol

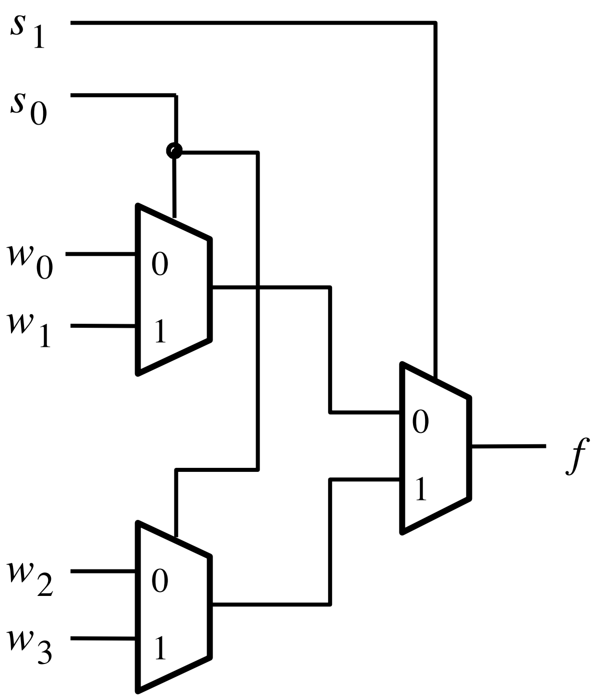
s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

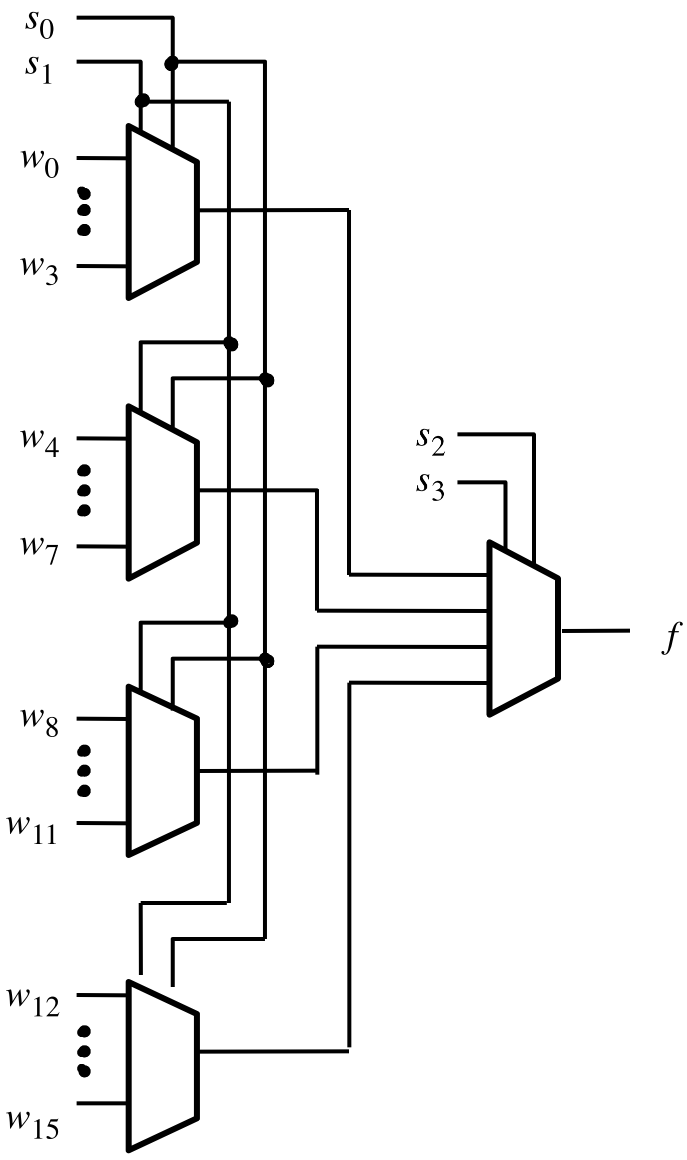


(c) Circuit

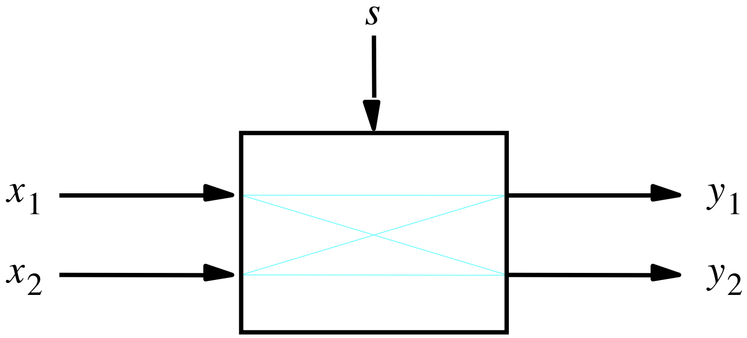
4-to-1 multiplexer.



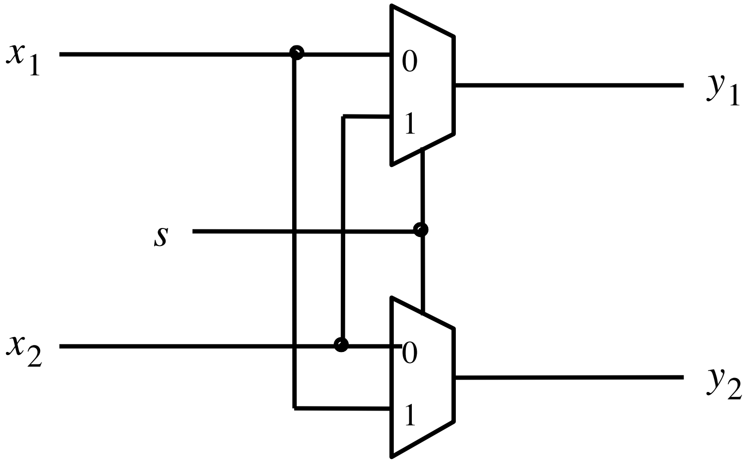
Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.



A 16-to-1 multiplexer.



(a) A 2x2 crossbar switch: connects any input to any output

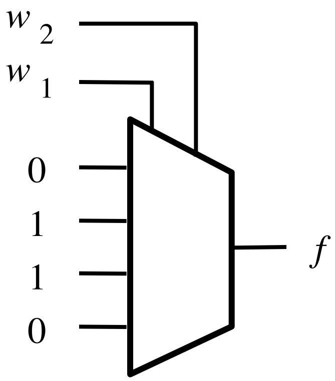


(b) Implementation using multiplexers

Synthesis Using MUXs

- E.g. f = w_1 xor w_2

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

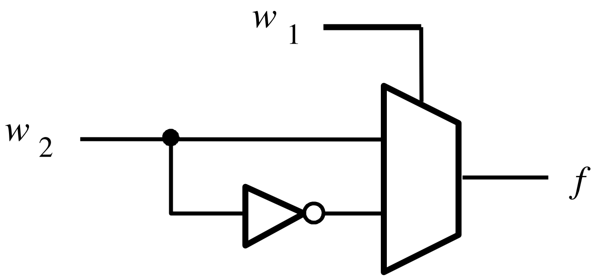


(a) Implementation using a 4-to-1 multiplexer

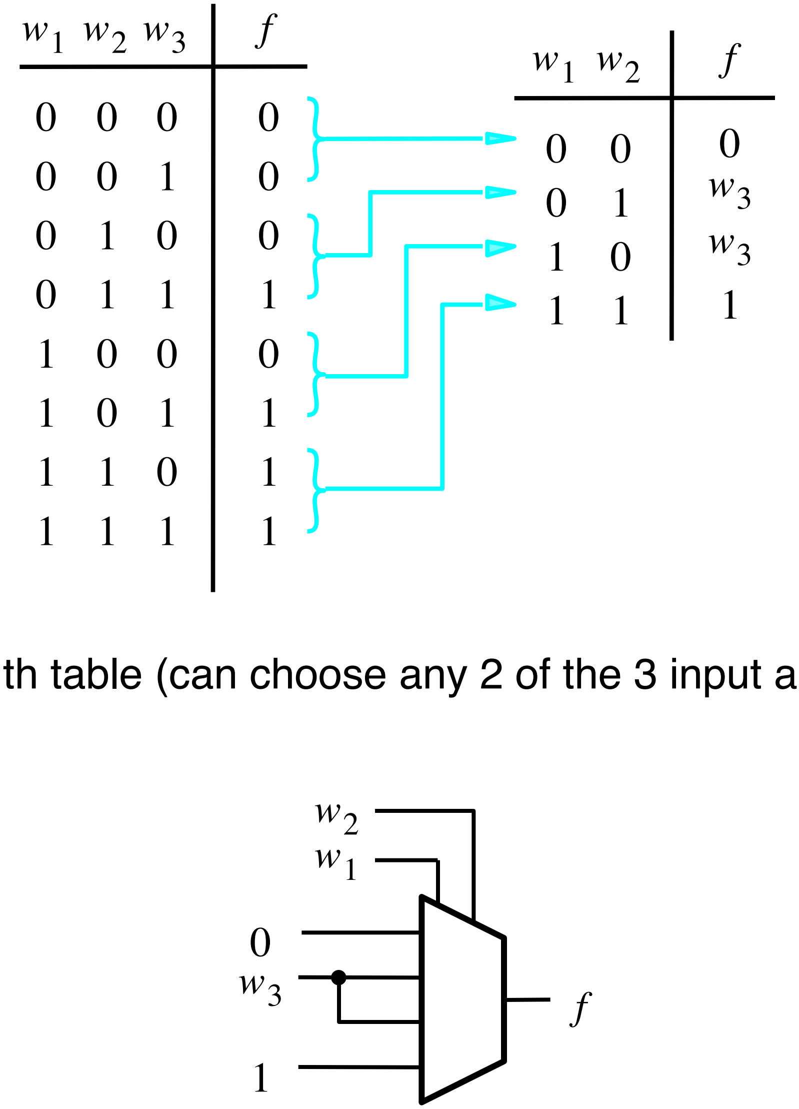
w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\overline{w}_2$

(b) Modified truth table



(c) Circuit



w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

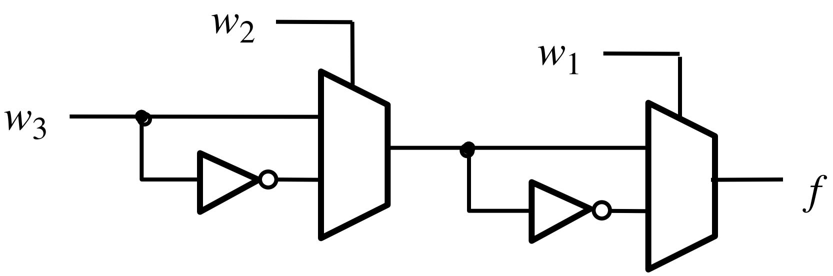
(a) Modified truth table (can choose any 2 of the 3 input as the select signals)

(b) Circuit

Implementation of the three-input majority function using a 4-to-1 multiplexer

w_1	w_2	w_3	f	
0	0	0	0	$w_2 \oplus w_3$
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	$\overline{w_2 \oplus w_3}$
1	0	1	0	
1	1	0	0	
1	1	1	1	

(a) Truth table

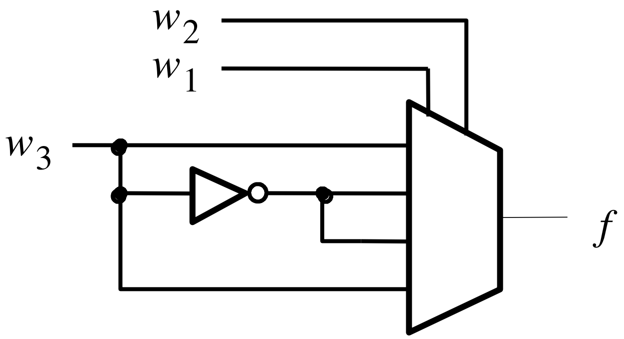


(b) Circuit

- Three-input XOR implemented with 2-to-1 multiplexers.
- Another way to derive the circuit is to write *f* = (w_2 xor w_3) xor w_1
- Other circuits are possible

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\overline{w_3}$
0	1	1	0	
1	0	0	1	$\overline{w_3}$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

(a) Truth table



(b) Circuit

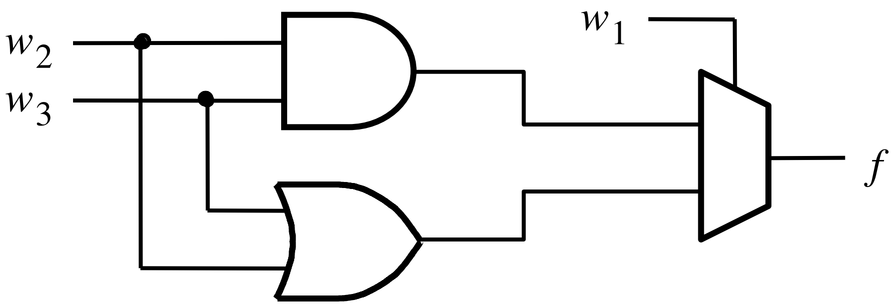
Three-input XOR function implemented with a 4-to-1 multiplexer.

MUX Synthesis: Shannon’s Expansion

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	$w_2 w_3$
1	$w_2 + w_3$

(a) Truth table for 3-input majority function



(b) Circuit

Decompose in terms of variables used for select inputs

Shannon’s Expansion Theorem

- Any Boolean Function $f(w_1, \ldots, w_n)$ can be written in the form

$$f(w_1, \ldots, w_n) = \bar{w_1} \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$

$f_{\bar{w_1}}$

Cofactor of *f* with respect to w_1’

f_{w_1}

Cofactor of *f* with respect to w_1

- In general:

- E.g. 3-input majority function

- E.g. 3-input XOR

Expansion in terms of more variables

$$\begin{aligned} f(w_1, \ldots, w_n) = {} & \bar{w_1}\bar{w_2} \cdot f(0, 0, w_3, \ldots, w_n) + \bar{w_1} w_2 \cdot f(0, 1, w_3, \ldots, w_n) \\ & + w_1 \bar{w_2} \cdot f(1, 0, w_3, \ldots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n) \end{aligned}$$

- – Implement using a 4-to-1 MUX

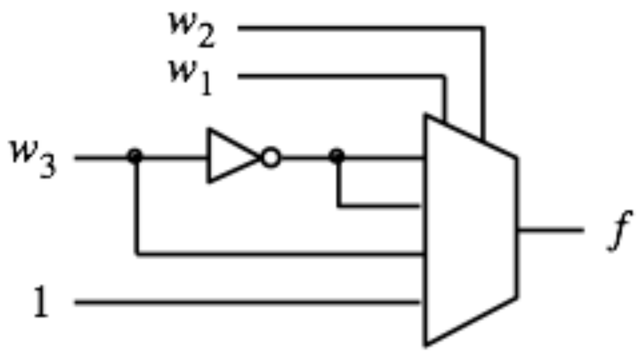
- Expansion in terms of all n variables

→ canonical sum-of-products form

Examples

$$f = \overline{w}_1\overline{w}_3 + w_1w_2 + w_1w_3$$

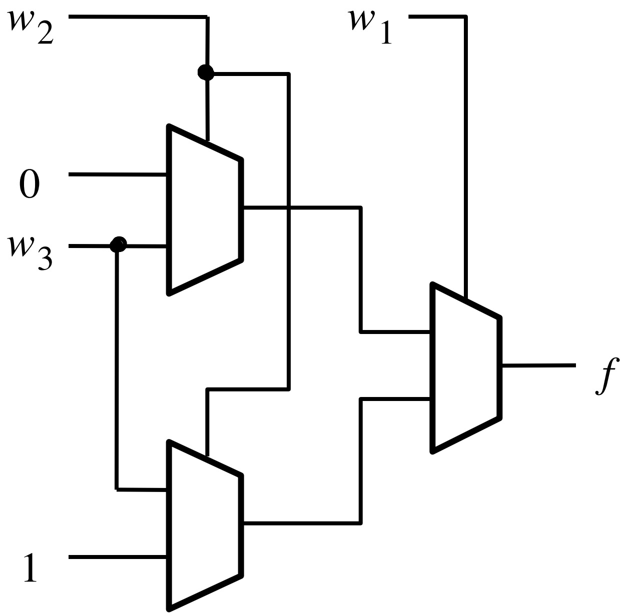
$$f = \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1w_2(1)$$



$$f = w_1 w_2 + w_1 w_3 + w_2 w_3$$

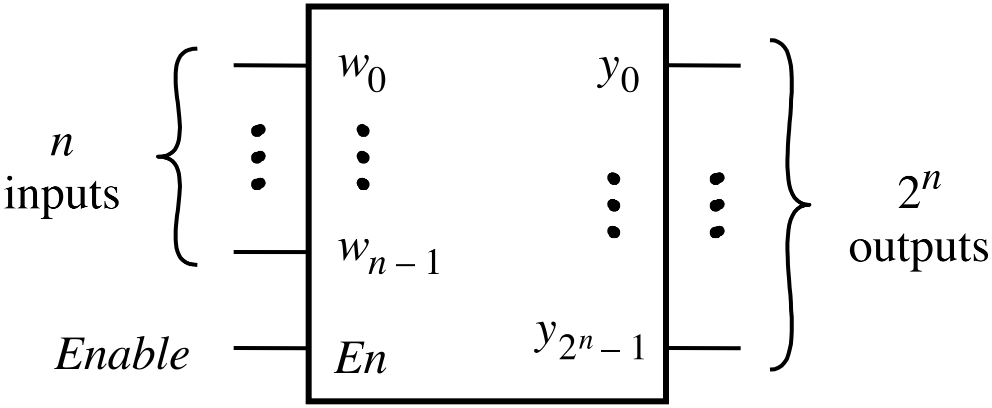
$$f = \overline{w}_1(w_2 w_3) + w_1(w_2 + w_3 + w_2 w_3)$$

$$f = \overline{w}_1(w_2 w_3) + w_1(w_2 + w_3)$$



Decoders

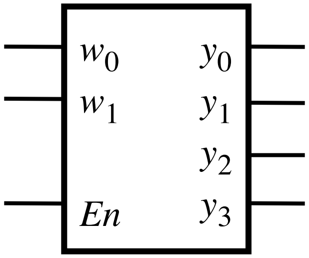
- Only one output is asserted at a time
  - *one-hot* encoding
  - Output corresponds to one valuation of the inputs
  - En = 0 → no outputs asserted



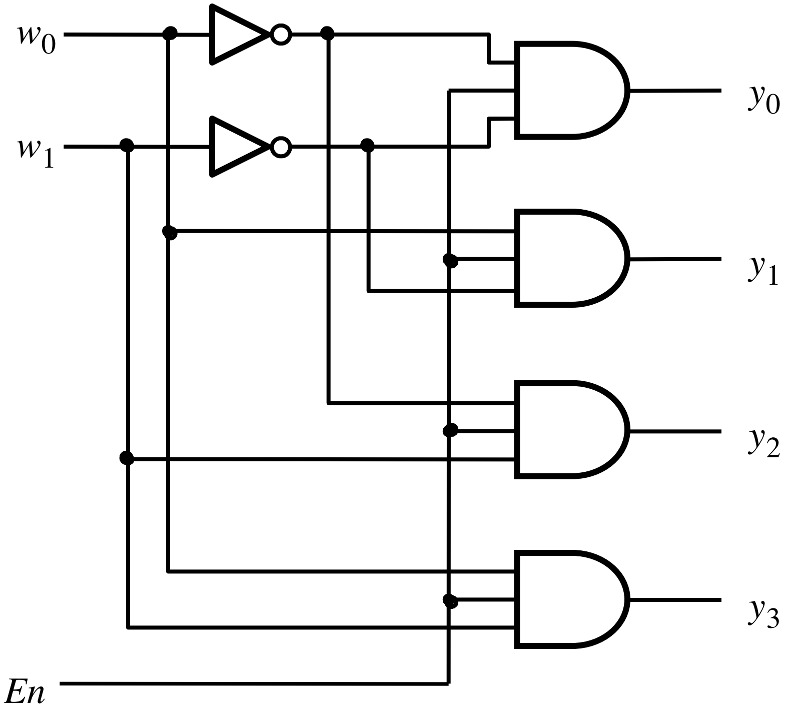
n-to-2^n binary decoder.

En	w_1	w_0	y_0	y_1	y_2	y_3
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	0	1
0	x	x	0	0	0	0

(a) Truth table



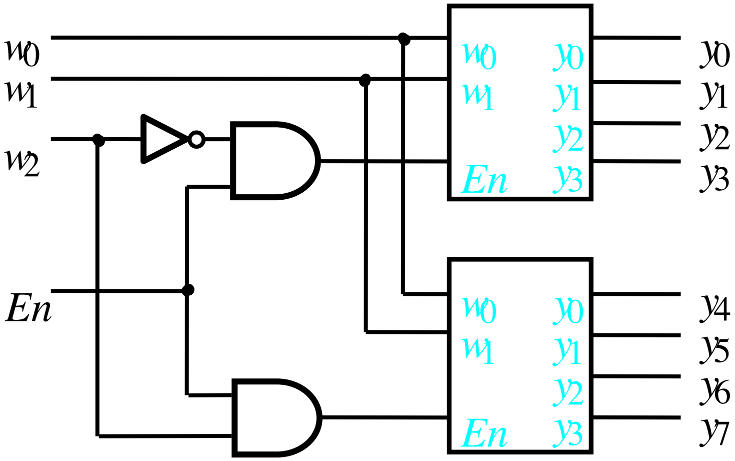
(b) Graphical symbol



(c) Logic circuit

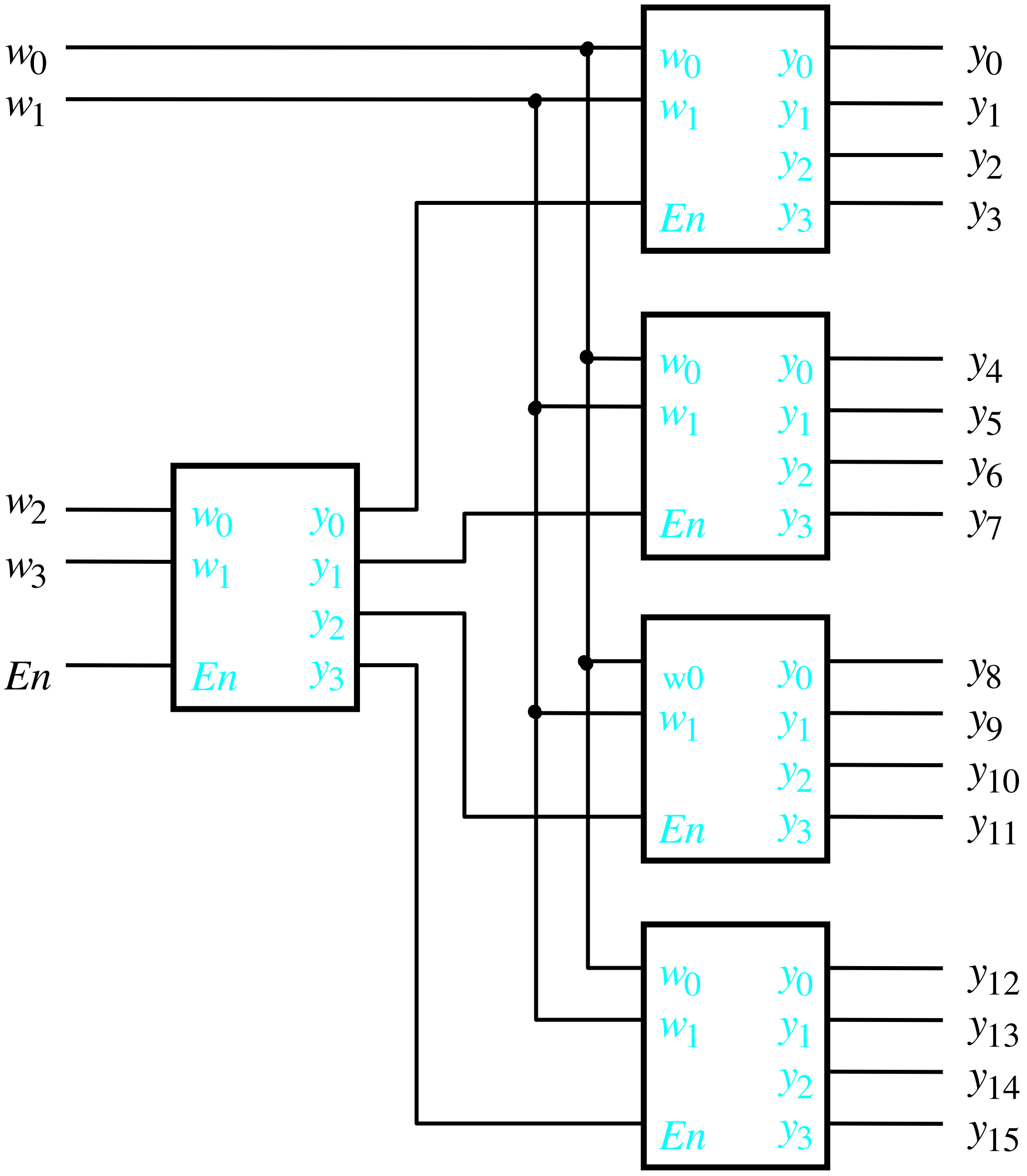
Larger decoders can be build by extending this SOP structure or by combining smaller decoders

2-to-4 decoder



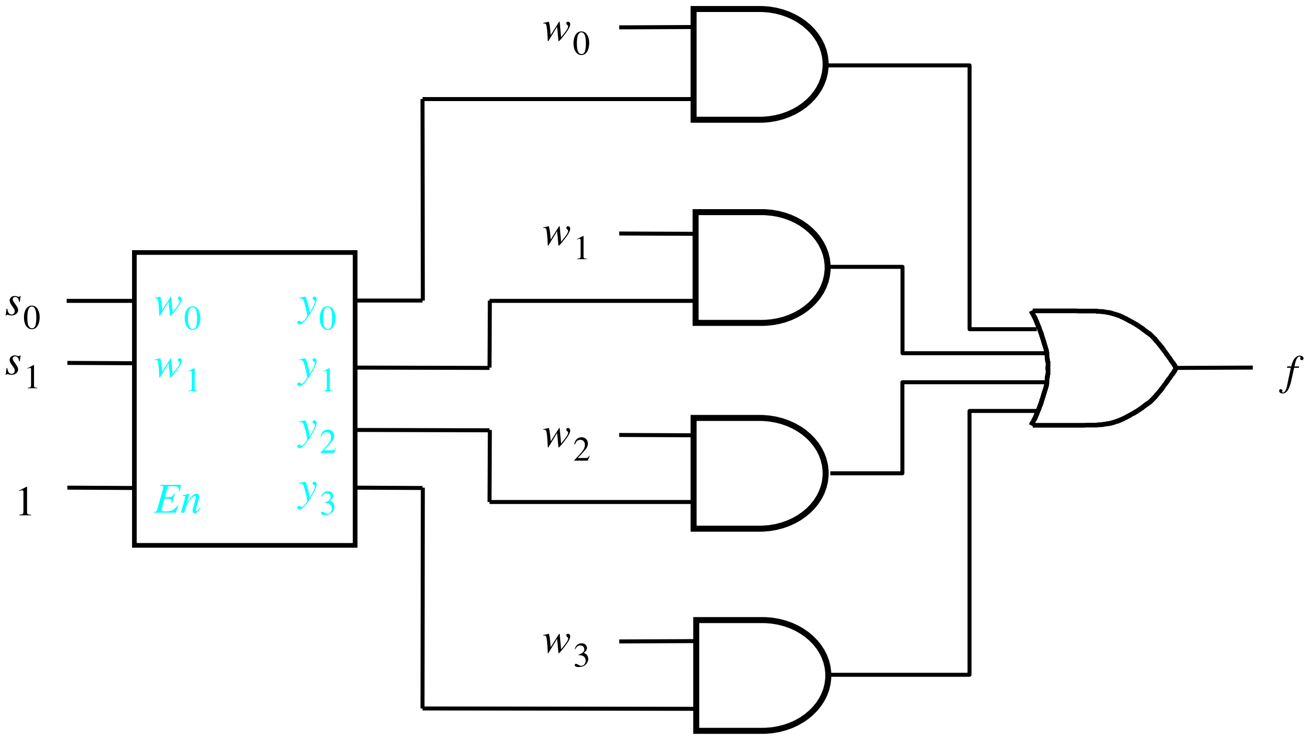
A 3-to-8 decoder using two 2-to-4 decoders.

Top decoder is enabled if $w_2 = 0$ and bottom is enabled if $w_2 = 1$



A 4-to-16 decoder built using a decoder tree

Application: MUX

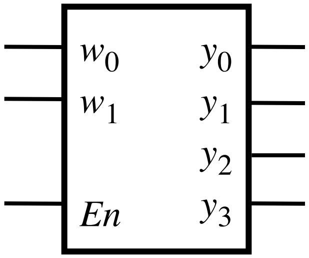


A 4-to-1 multiplexer built using a decoder.

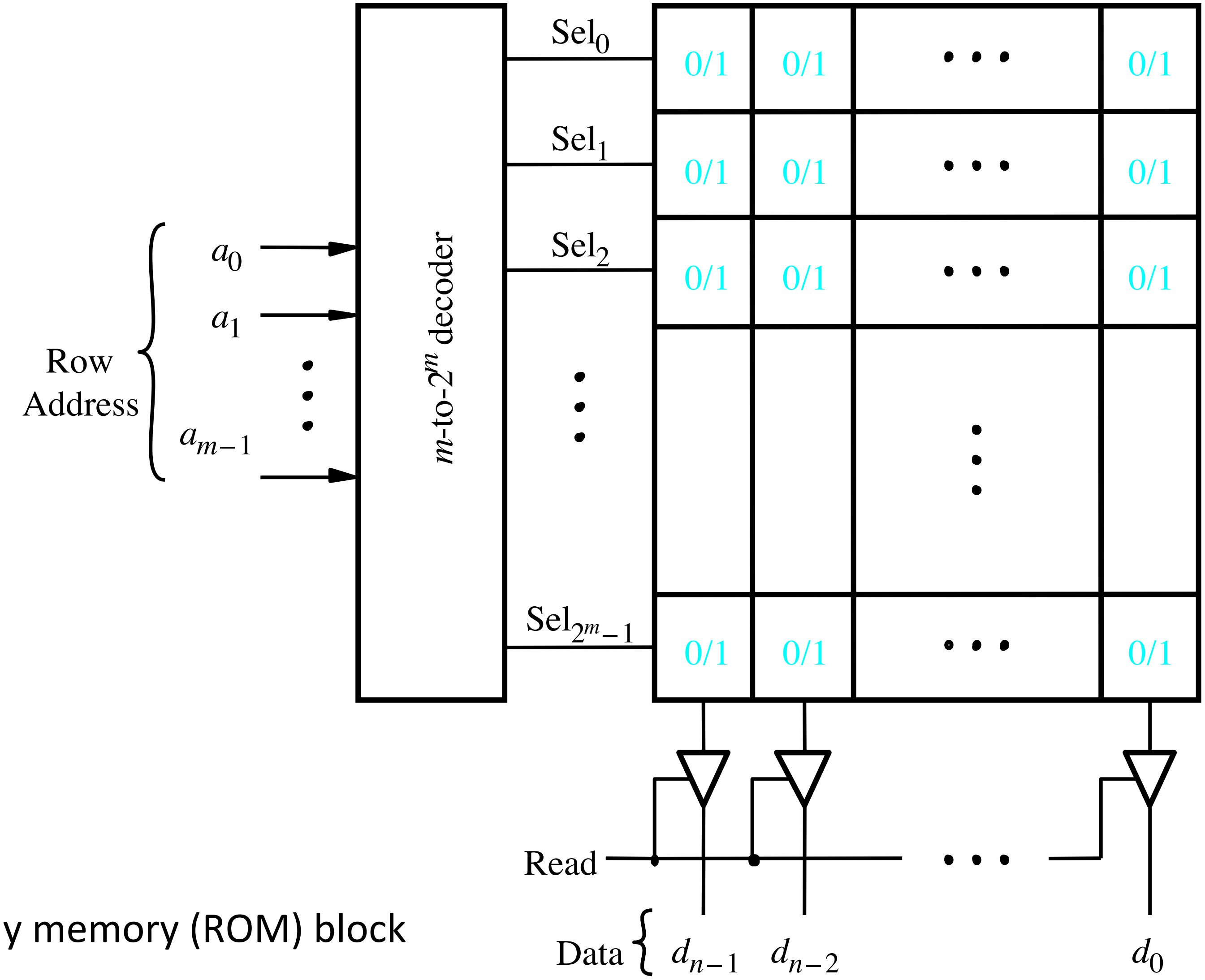
Application: Demultiplexers

- Places the value of a single input onto one of 2^n outputs, controlled by *n* select bits
  - Can be implemented using a decoder
  - e.g. 2:4 decoder can implement a 1:4 demux using the enable as data input

En	w_1	w_0	y_0	y_1	y_2	y_3
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	0	1
0	x	x	0	0	0	0



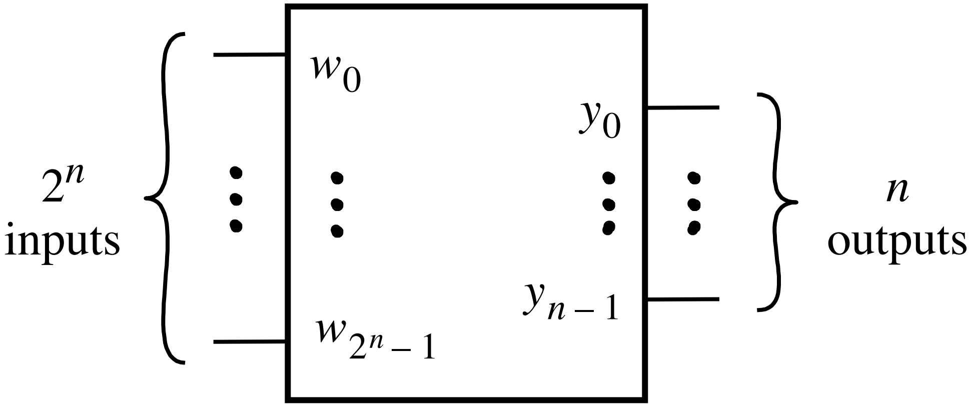
Application: Read-Only-Memory



A 2^m x n read-only memory (ROM) block

Encoders

- Used to compactly represent information
  - Exactly one of the inputs should be 1
  - Output is the binary number that identifies which input is 1



2^n-to-n binary encoder.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Truth table for 4:2 binary encoder

Priority Encoders

- Output indicates which input of the highest *priority* is active
  - Lower priority inputs are ignored
  - Output *z* indicates none of the inputs are 1

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

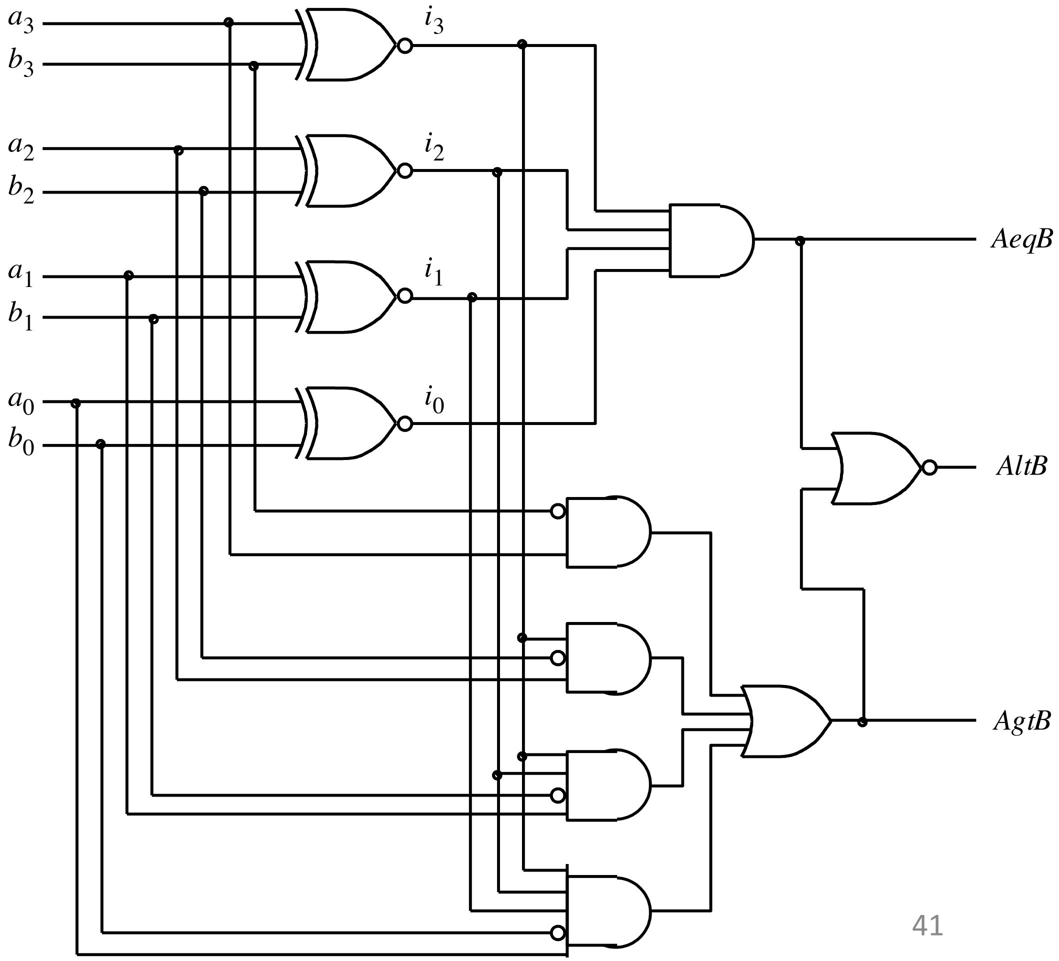
4-to-2 priority encoder.

$$w_0\text{: lowest priority}$$

$$w_{2^n-1}\text{: highest priority}$$

Arithmetic Building Blocks

- Comparators
  - Inputs A and B are unsigned *n*-bit binary numbers
  - E.g.: 4-bit comparator
- Truth table is large even for moderate values of *n*



$A = a_3 a_2 a_1 a_0$

$B = b_3 b_2 b_1 b_0$

$i_k = 1$ if $a_k = b_k$

Starting at MSB, find first position k for which $a_k \neq b_k$

If $a_k = 0$ and $b_k = 1$ then $A < B$
If $a_k = 1$ and $b_k = 0$ then $A > B$

e.g: A = 0 1 1 0
B = 0 1 0 1

$$AeqB = i_3 i_2 i_1 i_0$$

$$AgtB =$$

$$AltB =$$

Other Arithmetic Circuits

- Adders (and subtractors)
  - Many different adder circuits
- Multipliers / Dividers
  - Build out of multiple adders and subtractors
  - In general can require many logic gates to implement
- Multiply or divide by powers of 2
  - Use a *shifter* circuit
  - Shifter design including VHDL is given in Examples 6.31, 6.32, 6.34 and 6.35

Full Adder (5.2)

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

(a) Truth table

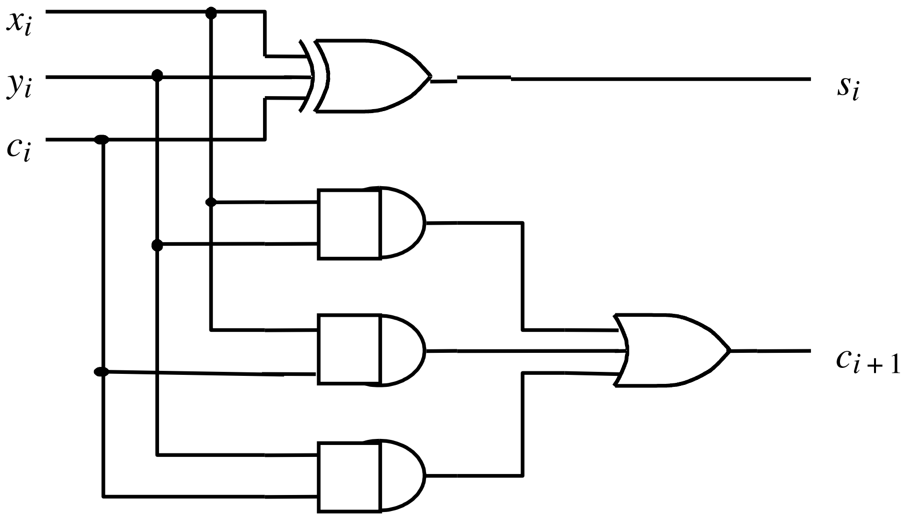
c_i \ x_iy_i	00	01	11	10
0		1		1
1	1		1	

$$s_i = x_i \oplus y_i \oplus c_i$$

c_i \ x_iy_i	00	01	11	10
0			1	
1		1	1	1

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

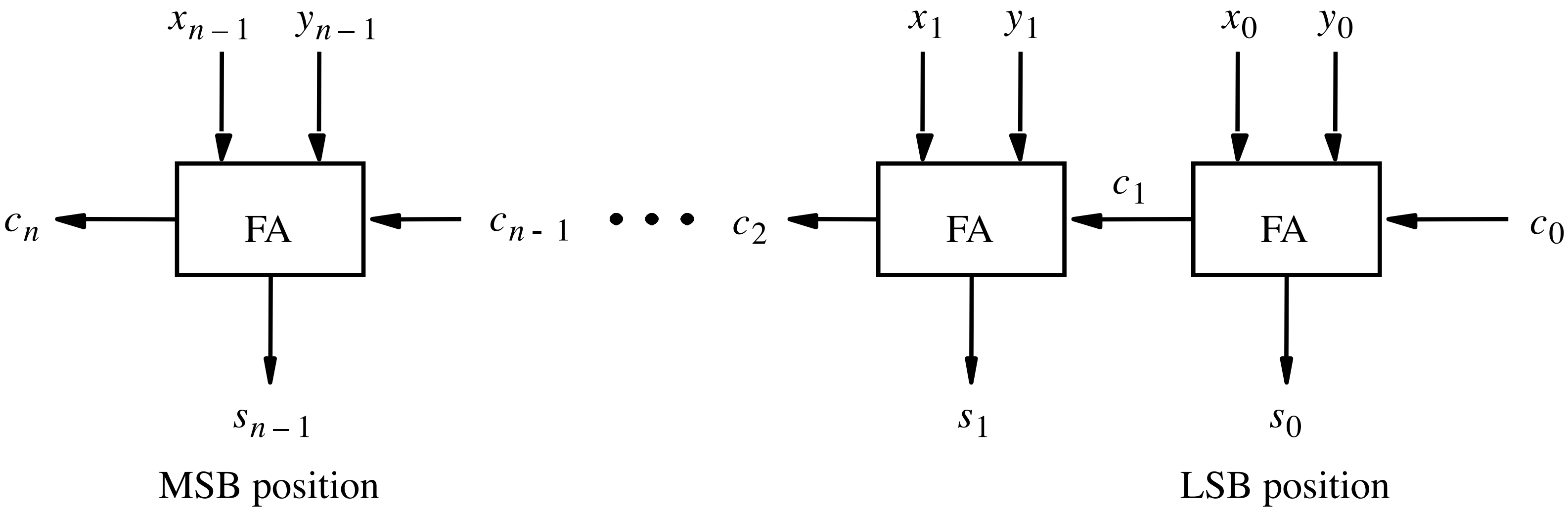
(b) Karnaugh maps



(c) Circuit

Ripple Carry Adders

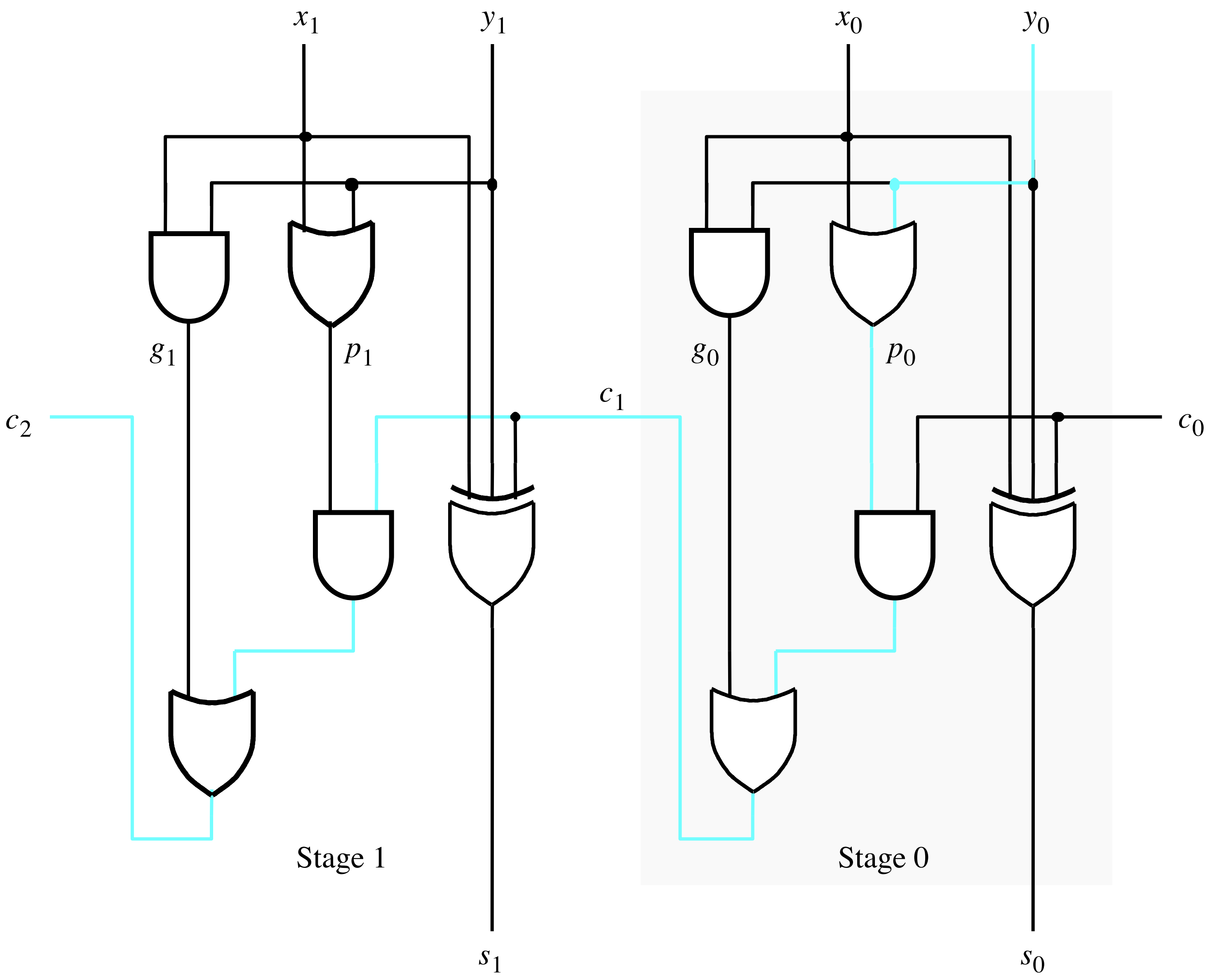
- You have seen binary *n*-bit ripple carry adders in ECSE 221 (see 5.2.2)
  - The sum is available after a delay of $n\Delta t$ where Δt is the delay of a full-adder



Carry-Lookahead Adder (5.4)

$$\begin{aligned}
c_{i+1} &= x_i y_i + x_i c_i + y_i c_i \\
&= x_i y_i + (x_i + y_i) c_i \\
&= g_i + p_i c_i \\
&= g_i + p_i (g_{i-1} + p_{i-1} c_{i-1}) \\
&= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \\
&= g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0
\end{aligned}$$

- g_i: carry generate = $x_i y_i$
- p_i: carry propagate = $x_i + y_i$
  - – g_i and p_i are independent of the carry-in c_i
  - – The carry-out c_{i+1} is computed with a fast two-level circuit independent of the carry-in



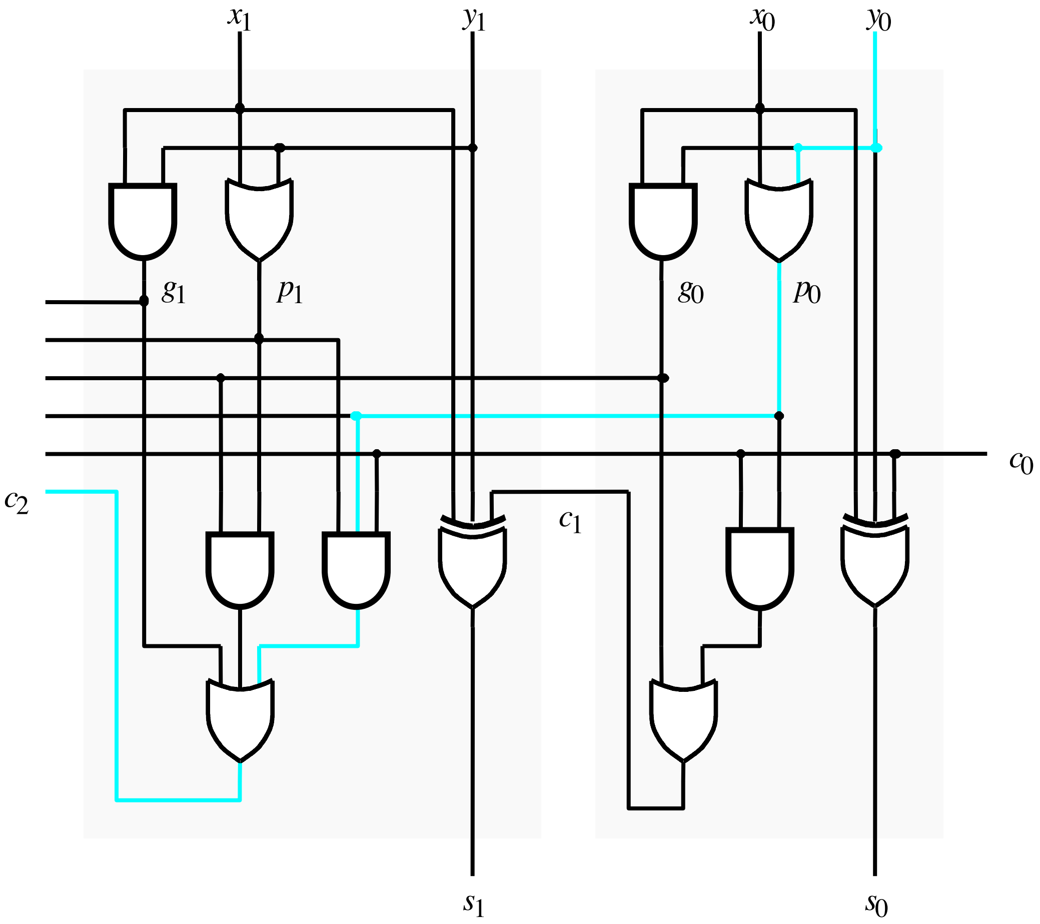
ripple-carry adder

$$\text{Delay} = 2n + 1 \text{gates}$$

- After the *g* and *p* signals settle in 1 gate delay, **all output carries are evaluated *simultaneously*** in 2 gate delays and then one additional gate delay is needed to get the sum outputs

- Total delay of the entire adder is 4 gate delays total independent of *n*

- The trade-off is higher circuit complexity as *n* increases



- *Hierarchical* designs group input bits into blocks implemented with carry-lookahead adders and using ripple carries between blocks

- There are many different fast adder structures
  - Choice of adder structure depends on the constraints (delay, area, power) and the target technology
  - e.g. FPGAs are actually rather good at implementing carry-ripple adders because of dedicated fast carry chain wires

Self-Study

- Code-converters (6.4)
- VHDL for combinational circuits (4.12, 5.5, 6.6)
- As usual, the last section in each chapter of the textbook has solved practice problems (4.14, 5.9, 6.8)