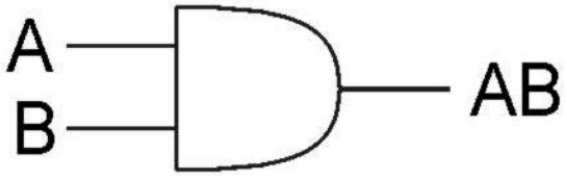
Lecture 12:
Digital Logic Cont.

Announcements

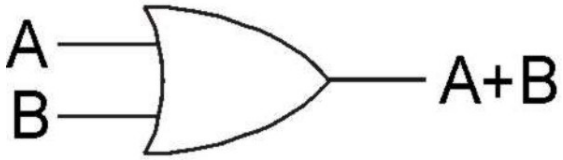
- Project 3 due in a week
  - Due August 11th 11:55pm
  - Due day before Final, so plan accordingly
  - No Extension
- Lectures until final:
  - Digital Logic (Chapter 4 in book)
  - Today: Kmaps and revisiting combinational circuits (decoders, multiplexers, and adders)
- Rest of lecture time / recitation today:
  - Questions on todays material and programming Assignment 3

Recap

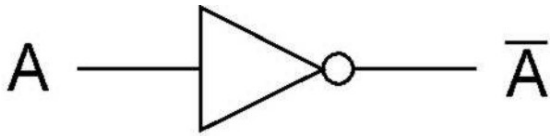
- 6 Widely used logic gates
- And Gate



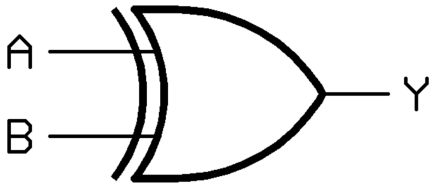
- Or Gate



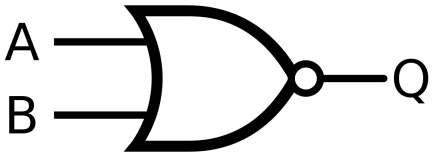
- Not Gate



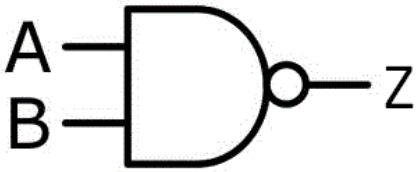
- Xor Gate



- Nor Gate

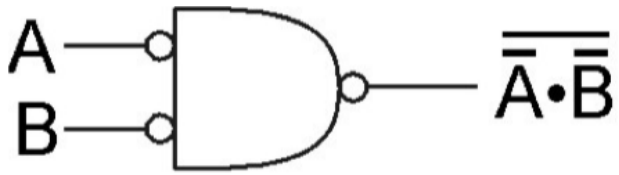


- Nand Gate



DeMorgan's Law

- Converting AND to OR (and some NOT).

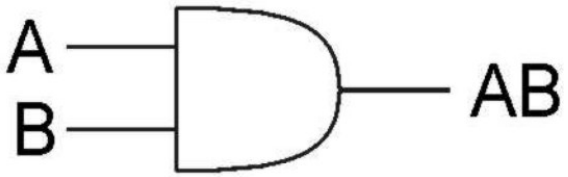


A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$	$\overline{\overline{A} \cdot \overline{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

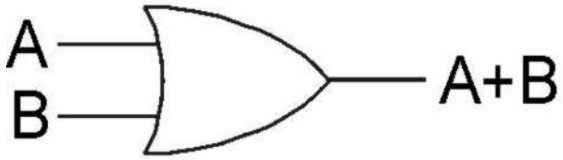
- In general,
  - (1) $\overline{PQ} = \overline{P} + \overline{Q}$
  - (2) $\overline{P + Q} = \overline{P}\,\overline{Q}$

Transformation

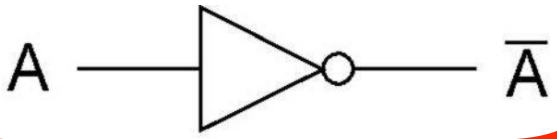
- 6 Widely used logic gates
- And Gate



- Or Gate



- Not Gate



- Xor Gate

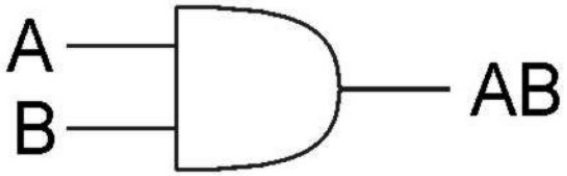
Can express any logic circuit with these as we have seen previously.

- Nor Gate

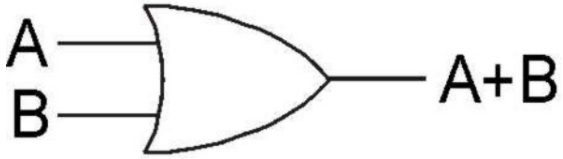
- Nand Gate

Transformation

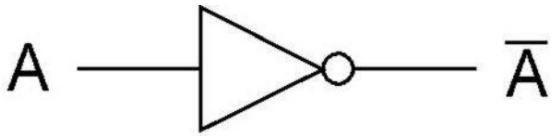
- 6 Widely used logic gates
- And Gate



- Or Gate



- Not Gate



- Xor Gate

$A\overline{B} + B\overline{A}$

- Nor Gate

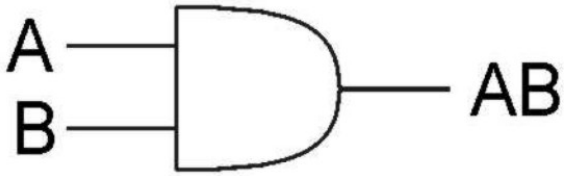
$\overline{A + B}$

- Nand Gate

$\overline{AB}$

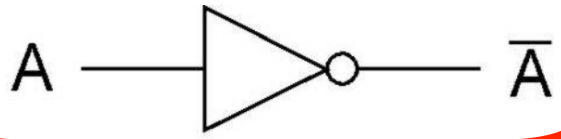
Transformation

- 6 Widely used logic gates
- And Gate



- Or Gate

- Not Gate



- Xor Gate
- Nor Gate

$\overline{A+B}$

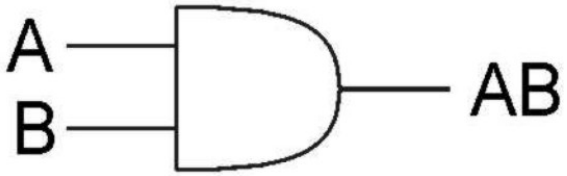
- Nand Gate

$\overline{AB}$

In fact…
You just need these two

Transformation

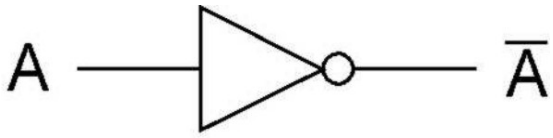
- 6 Widely used logic gates
- And Gate



- Or Gate

$$\overline{\overline{A}\,\overline{B}}$$

- Not Gate



- Xor Gate

$$\overline{\overline{A\overline{B}}\,\overline{\overline{A}B}}$$

- Nor Gate

$$\overline{A}\,\overline{B}$$

- Nand Gate

$$\overline{AB}$$

Transformation

- 6 Widely used logic gates
- And Gate

- Or Gate

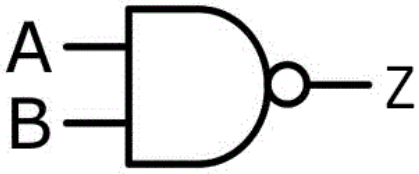
- Not Gate

- Xor Gate

Side note:
Or You just need this Nand Gate to do anything

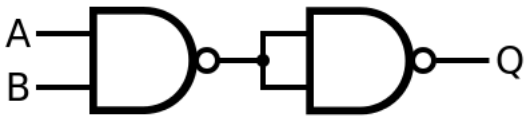
- Nor Gate

- Nand Gate

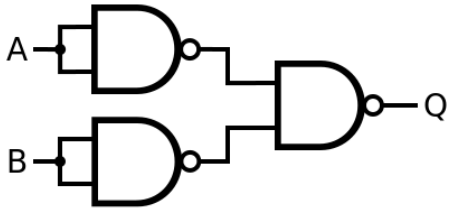


Transformation

- 6 Widely used logic gates
- And Gate

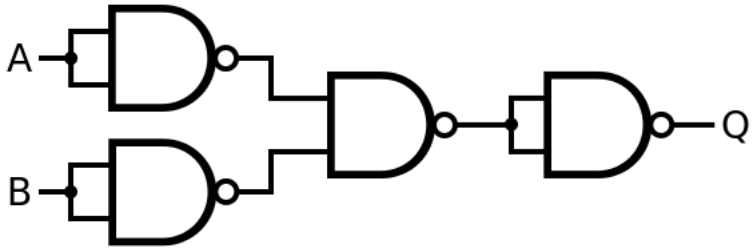


- Or Gate

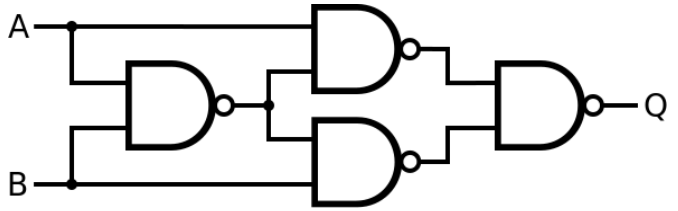


- Not Gate

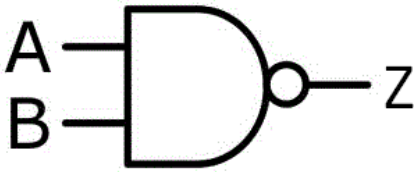
- Xor Gate



- Nor Gate



- Nand Gate



Transformation

- 6 Widely used logic gates

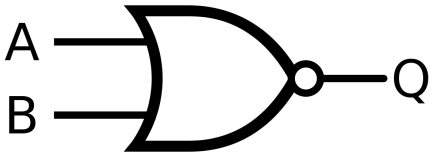
Side note:
Or You just need this
Nor Gate to do anything

- Or Gate

- Not Gate

- Xor Gate

- Nor Gate



- Nand Gate

Minterm

A	B	C	minterm	
0	0	0	m0	$\bar{A}\bar{B}\bar{C}$
0	0	1	m1	$\bar{A}\bar{B}C$
0	1	0	m2	$\bar{A}B\bar{C}$
0	1	1	m3	$\bar{A}BC$
1	0	0	m4	$A\bar{B}\bar{C}$
1	0	1	m5	$A\bar{B}C$
1	1	0	m6	$AB\bar{C}$
1	1	1	m7	ABC

- A product term in which all variables appear once.
- Each minterm evaluates to 1 in exactly one case. All other case, it evalutes to 0.

Truth Table to Boolean Expression

sensor inputs

A	B	C	Output
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

sensor inputs

A	B	C	Output	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{A}BC = 1$
1	0	0	0	
1	0	1	1	$A\overline{B}C = 1$
1	1	0	1	$AB\overline{C} = 1$
1	1	1	1	$ABC = 1$

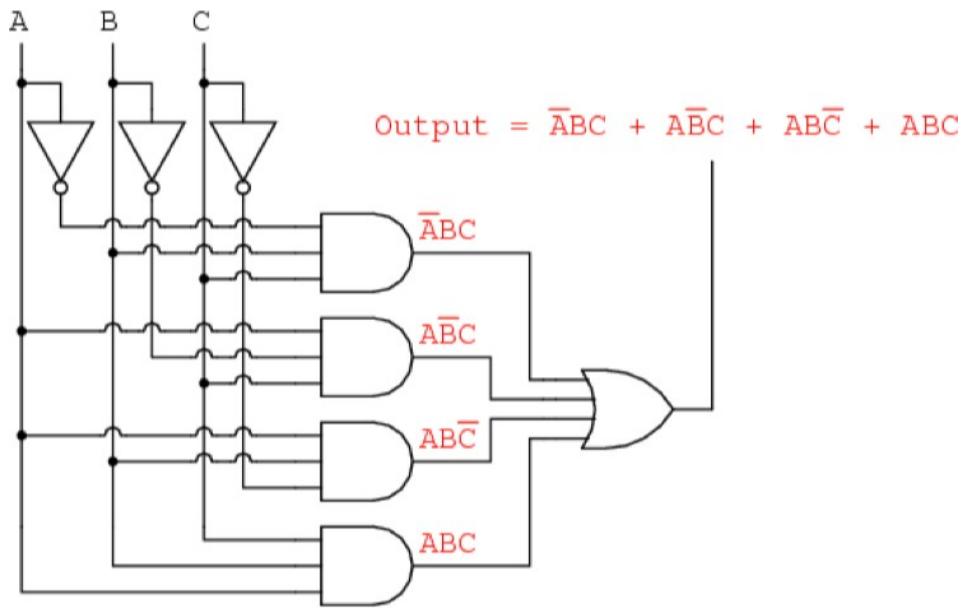
- Given expressions for each row, build a larger Boolean expression. This is called a sum-of-products (SOP) form.

$$Output = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

Truth Table to Boolean Expression

sensor inputs

A	B	C	Output	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\bar{A}BC = 1$
1	0	0	0	
1	0	1	1	$A\bar{B}C = 1$
1	1	0	1	$AB\bar{C} = 1$
1	1	1	1	$ABC = 1$



- Finally build the circuit
- However, SoP forms are usually not minimal. We must optimize.

Canonical Forms

- There are two canonical forms:
  - Sum of Products (SOP)

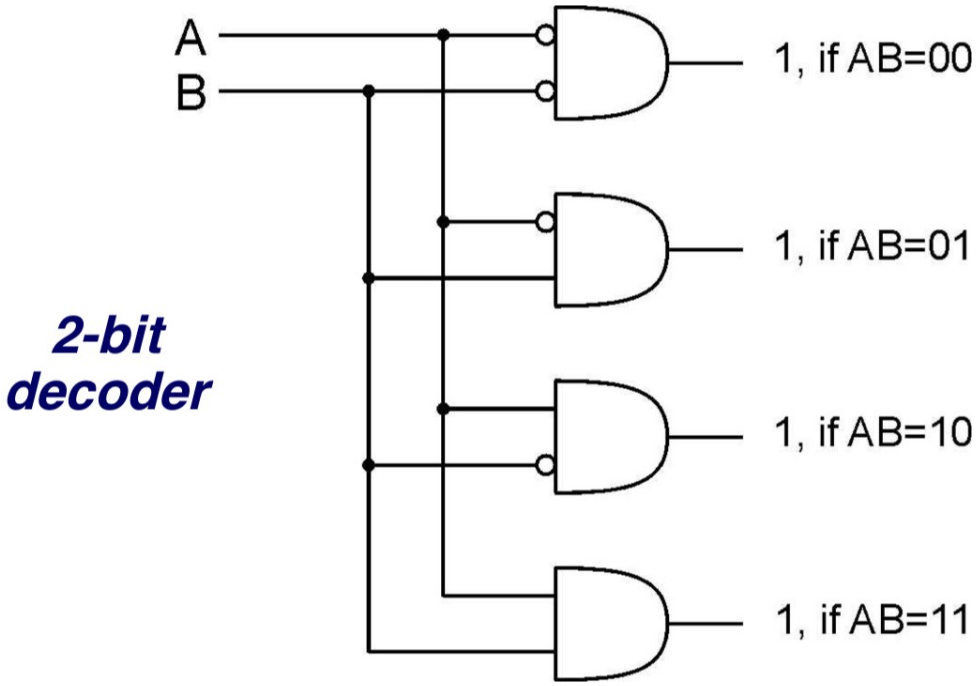
$$F = \overline{Y}\overline{Z} + X\overline{Y}Z + XY\overline{Z}$$

  - Product of Sums (POS)

$$F = (Y + Z)(\overline{X} + \overline{Y} + \overline{Z})(\overline{X} + \overline{Y} + Z)$$

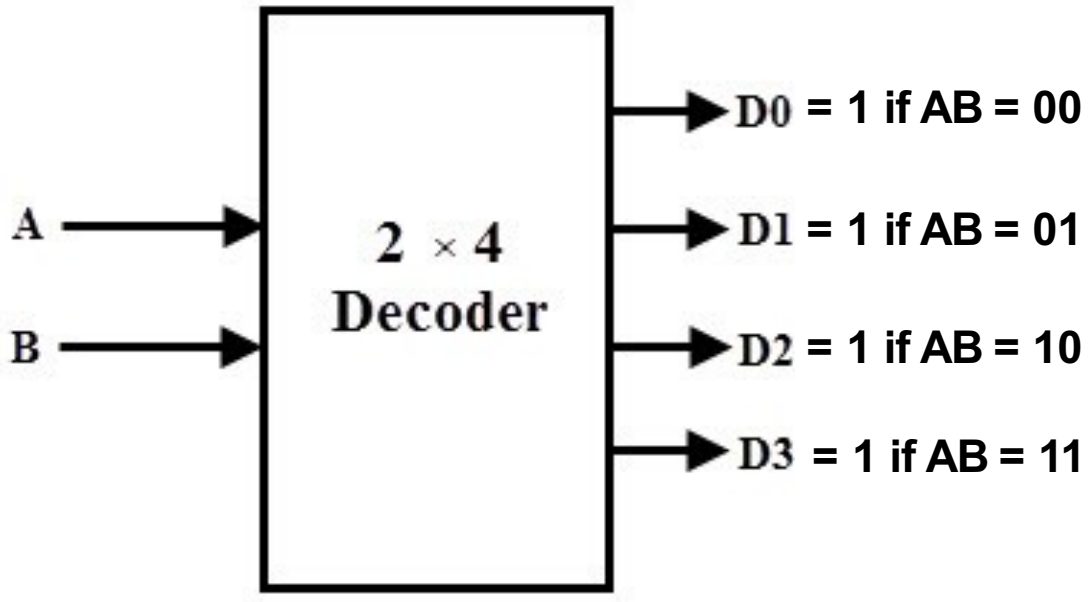
Decoder

- n inputs, 2^n outputs
- Exactly one output is 1 for a single possible input pattern



Decoder circuit

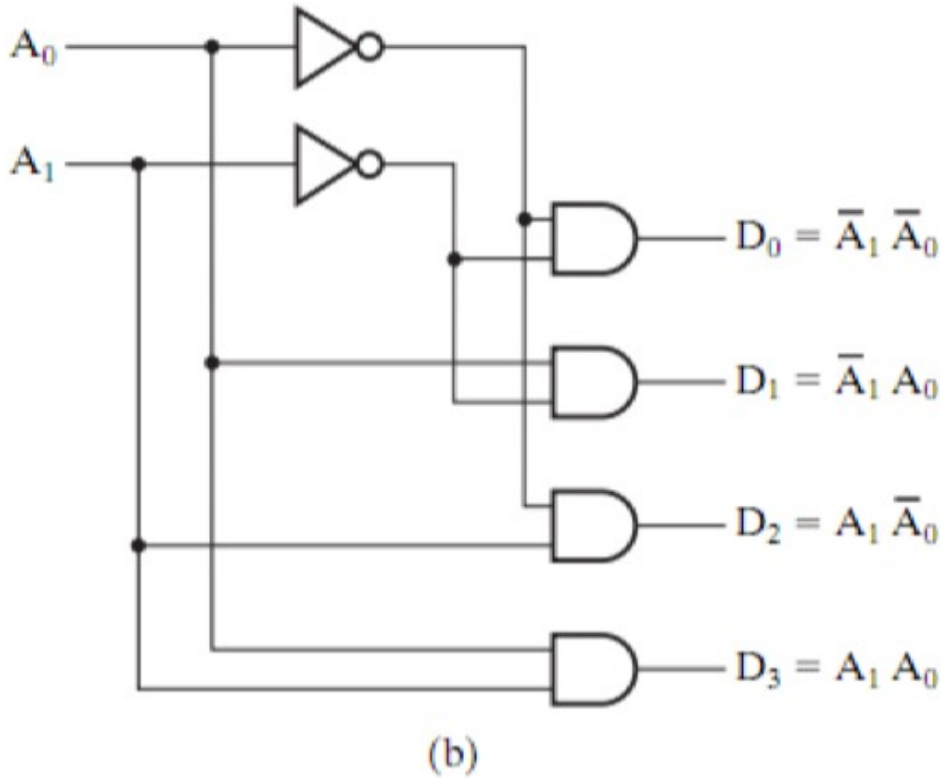
- n inputs, 2^n outputs
- Exactly one output is 1 for a single possible input pattern



Internal of 2:4 Decoder

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

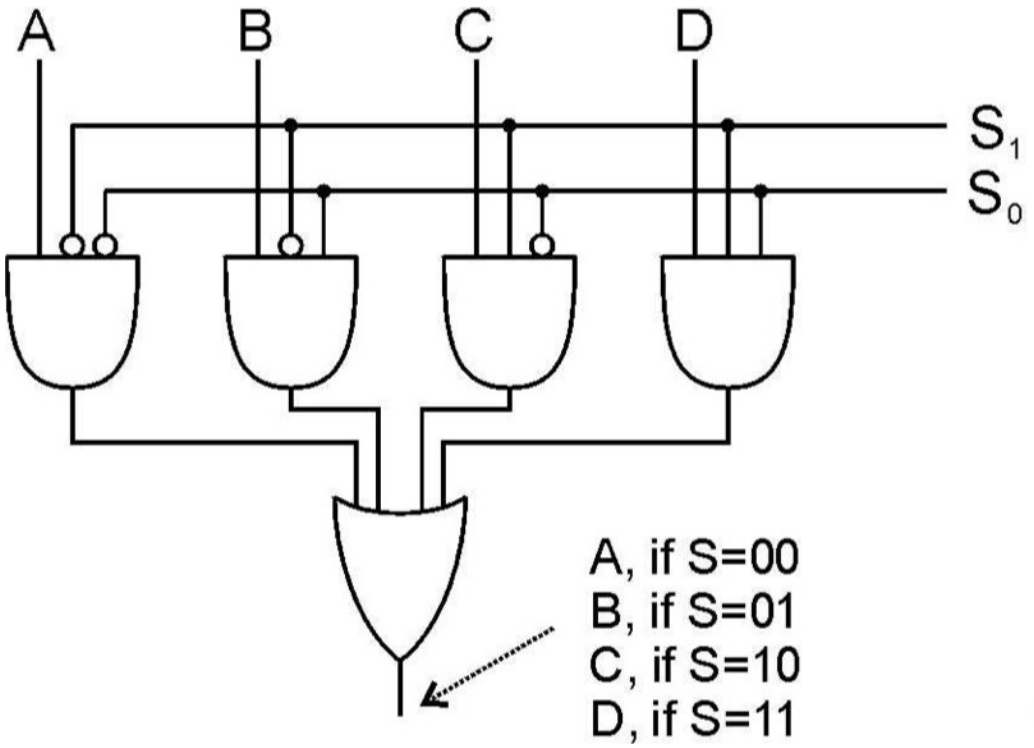
(a)

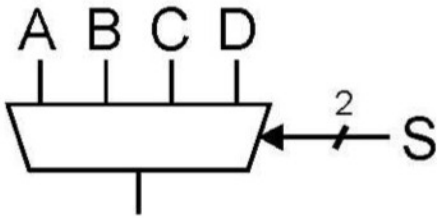


(b)

Multiplexer (MUX)

- 2^n inputs, n-bit selector, one output
  - Output equals one of the inputs, depending on the selector



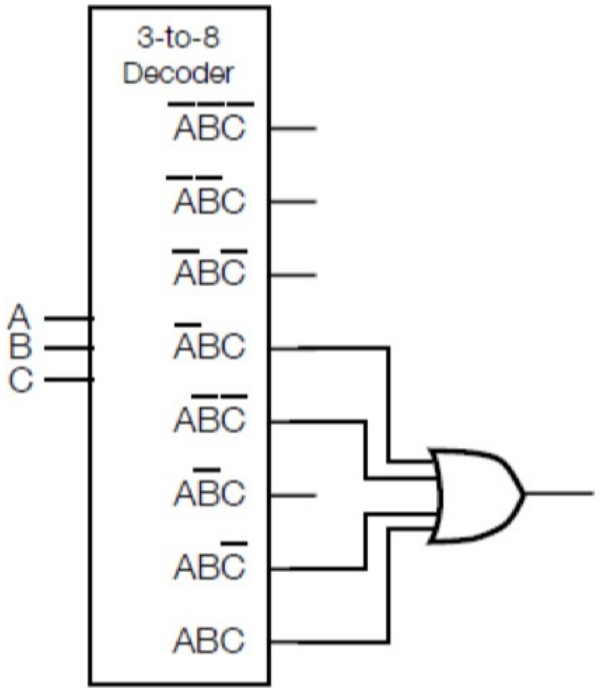


4-to-1 MUX

Functions with Decoders and Multiplexers

- e.g., $F = A\bar{C} + BC$

A	B	C	minterm	F
0	0	0	$\bar{A}\bar{B}\bar{C}$	0
0	0	1	$\bar{A}\bar{B}C$	0
0	1	0	$\bar{A}B\bar{C}$	0
0	1	1	$\bar{A}BC$	1
1	0	0	$A\bar{B}\bar{C}$	1
1	0	1	$A\bar{B}C$	0
1	1	0	$AB\bar{C}$	1
1	1	1	ABC	1

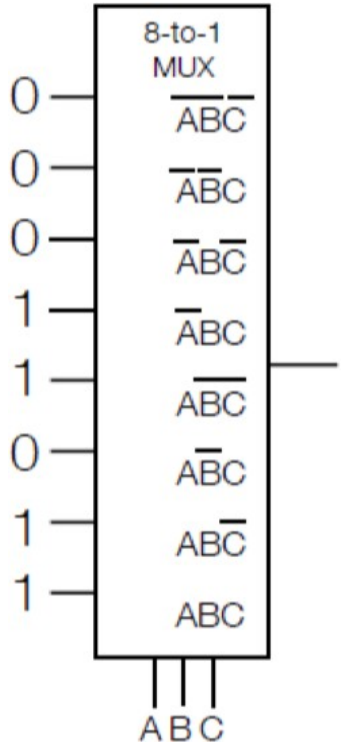


- OR minterms for which F should evaluate to 1

Functions with Decoders and Multiplexers

- e.g., $F = A\overline{C} + BC$

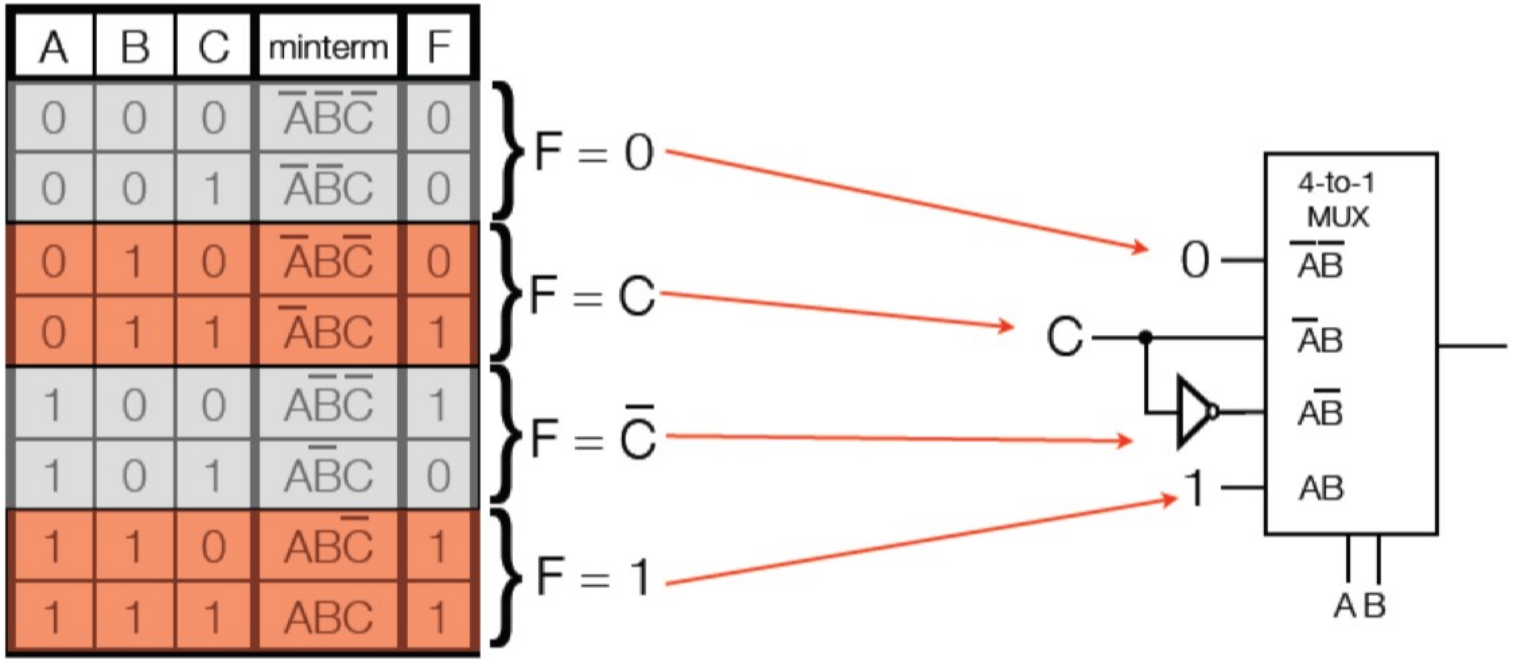
A	B	C	minterm	F
0	0	0	$\overline{A}\overline{B}\overline{C}$	0
0	0	1	$\overline{A}\overline{B}C$	0
0	1	0	$\overline{A}B\overline{C}$	0
0	1	1	$\overline{A}BC$	1
1	0	0	$A\overline{B}\overline{C}$	1
1	0	1	$A\overline{B}C$	0
1	1	0	$AB\overline{C}$	1
1	1	1	ABC	1



- Feed the value of F for each minterm in the input

Using Smaller mux:

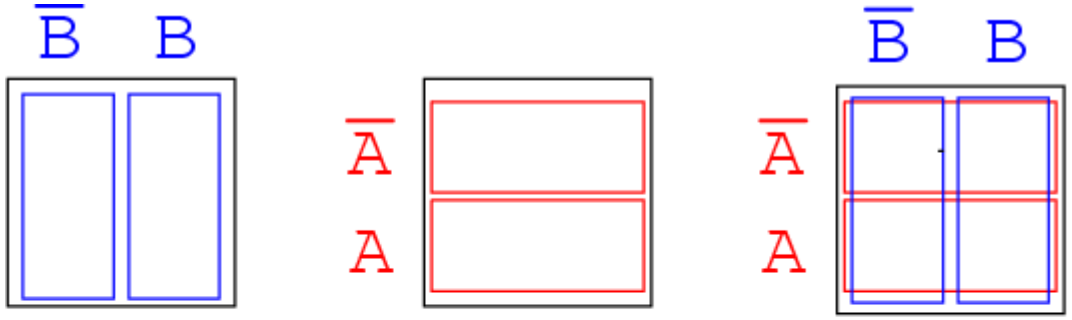
- We can use 4-to-1 mux with a trick:
- Every two rows have same A and B value. The output F depends on the value C.



Another Optimization Approach: Karnaugh Maps

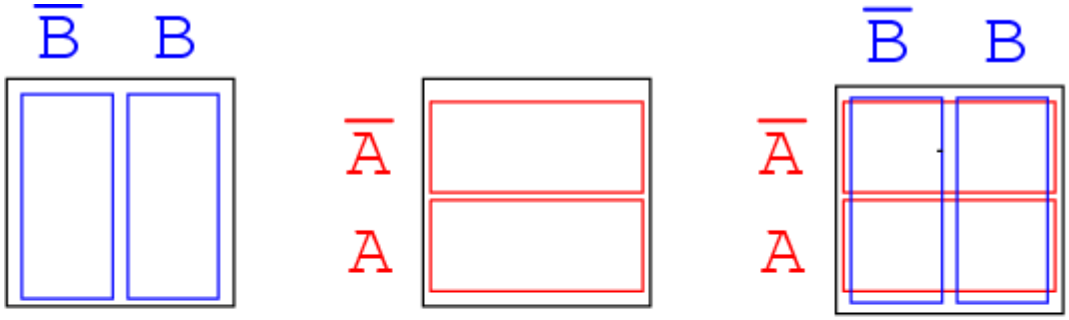
- K-maps are a graphical technique to view minterms and how they relate.
- Extremely useful to simplify boolean functions.
- The "map" is a diagram made up of squares, with each square representing a single minterm.
- Minterms resulting in a "1" are marked as "1", all others are marked "0"

2 Variable K-Map



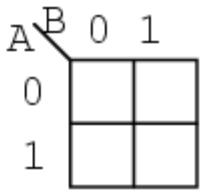
- Input B values would be the columns
- Input A values would be the rows
- The resulting matrix is a map that shows every possible input combination and the resulting output

2 Variable K-Map

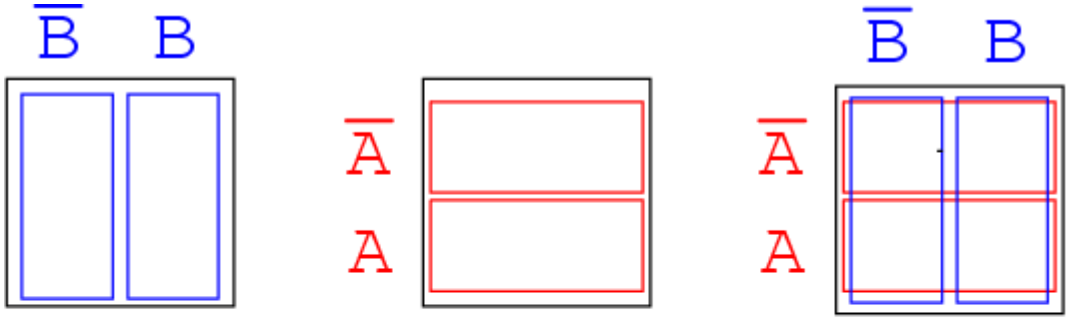


A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1

Given truth table



2 Variable K-Map



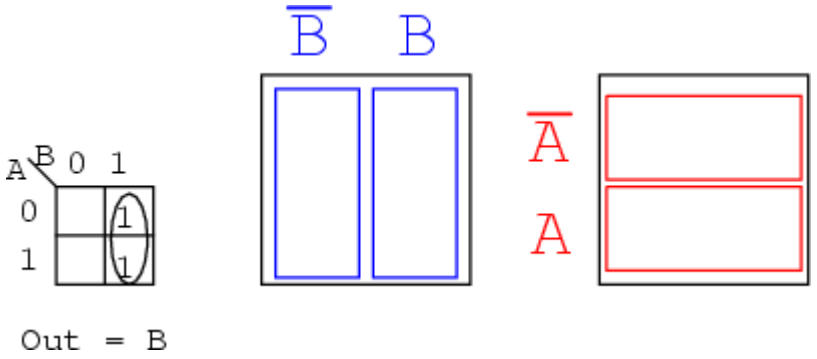
A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1

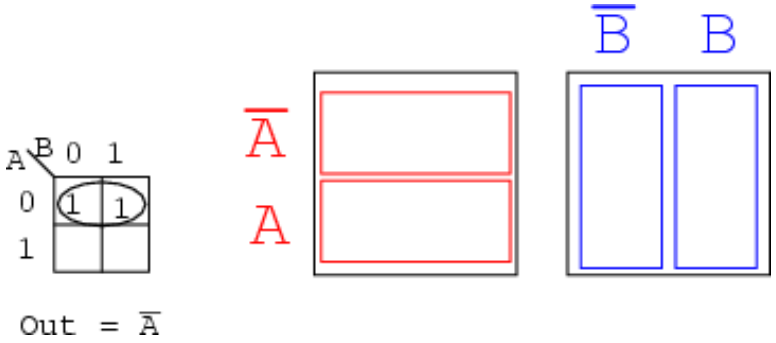
A \ B	0	1
0	0	1
1	0	1

Resulting K-map

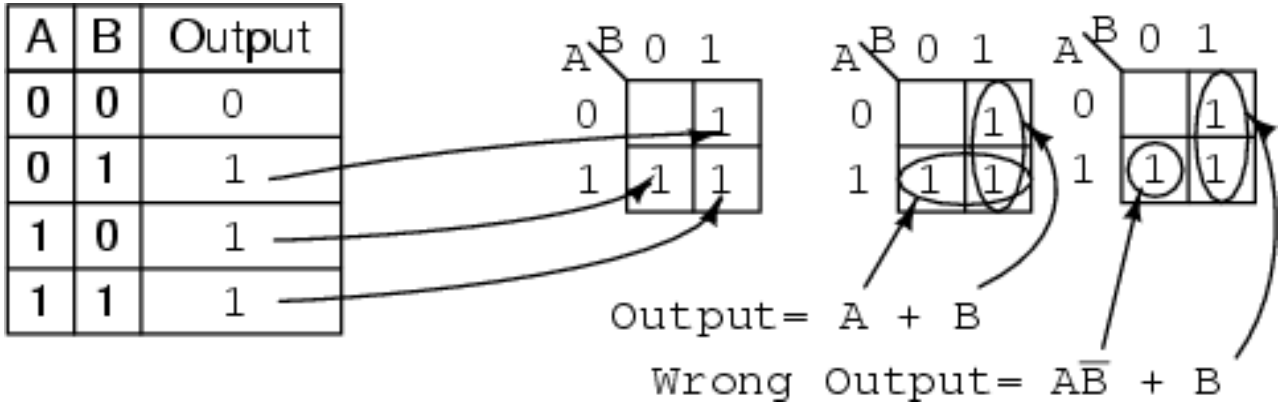
Finding Commonality

- Its usefulness come from the fact that adjacent squares correspond to minterms that differ in only ONE variable.
- Combining 2^n squares eliminate n variables.



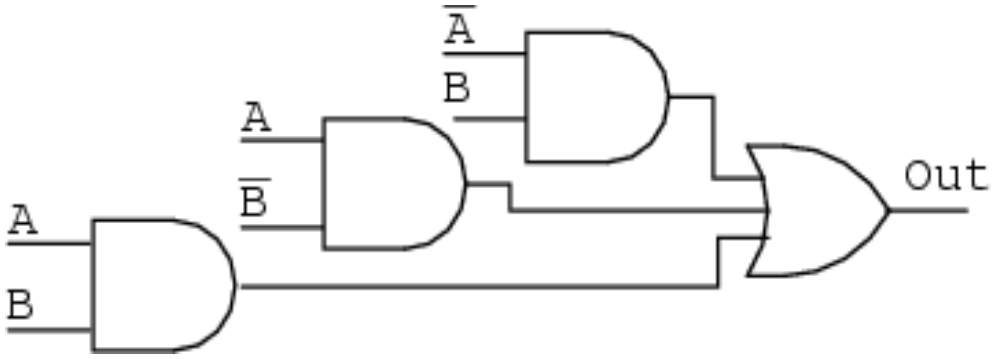


Finding the "best" solution

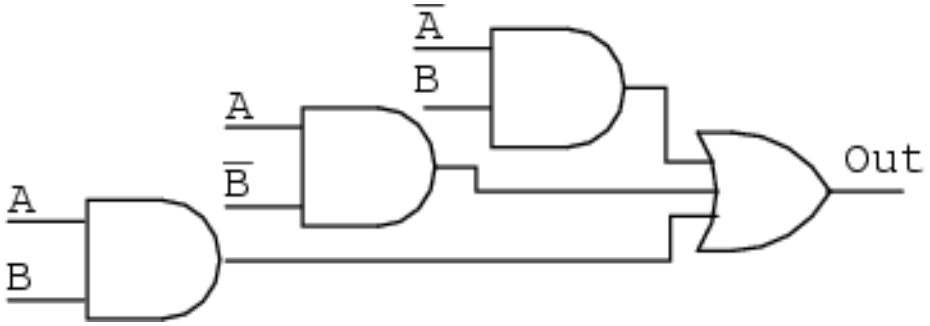


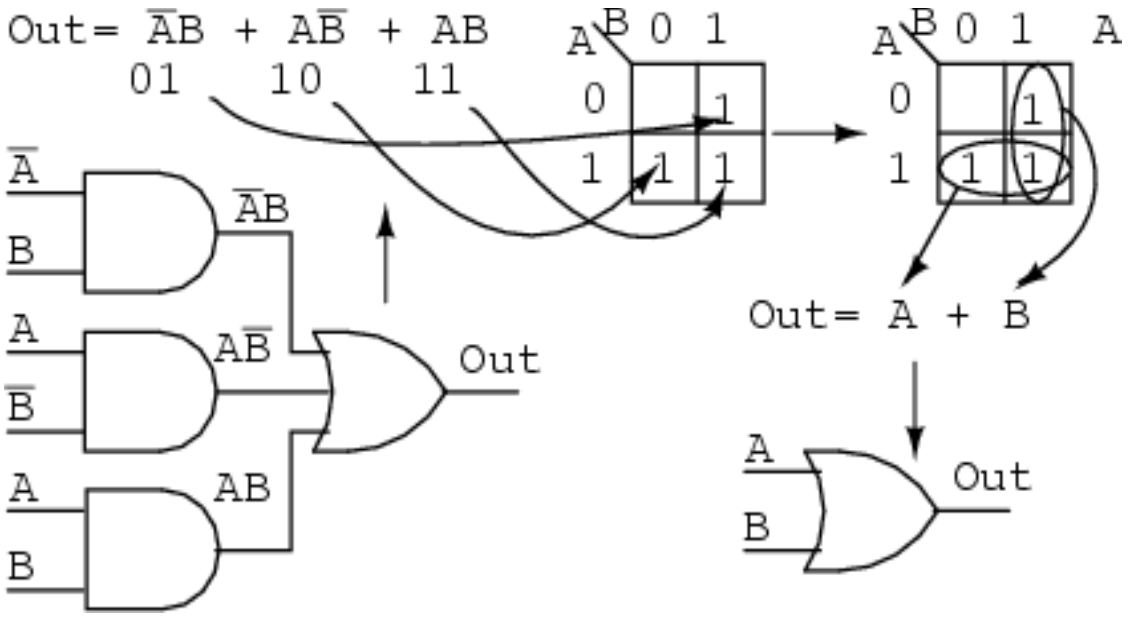
- Grouping become simplified products.
- Both are "correct". "A+B" is preferred.

Simplify Example

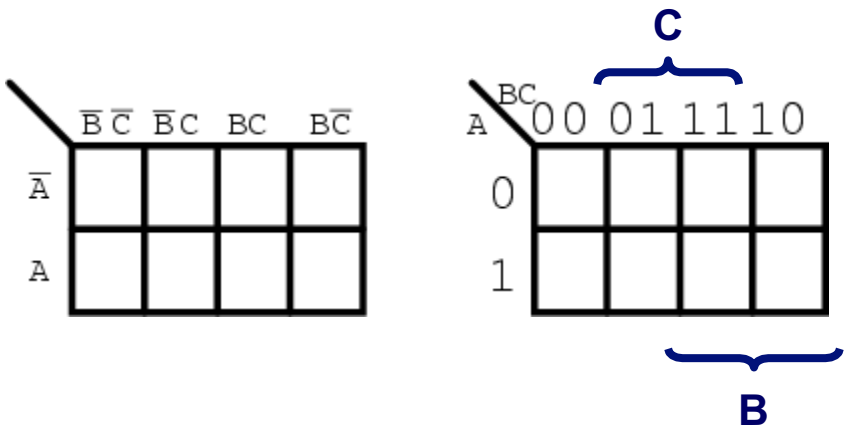


Simplify Example



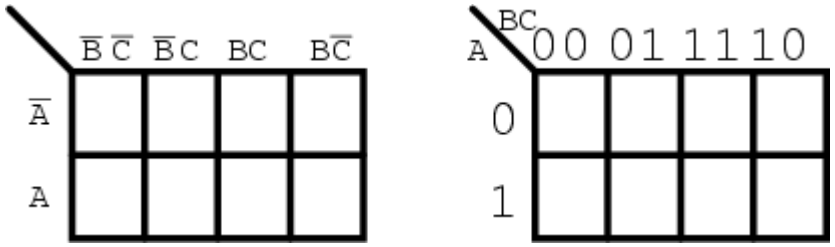


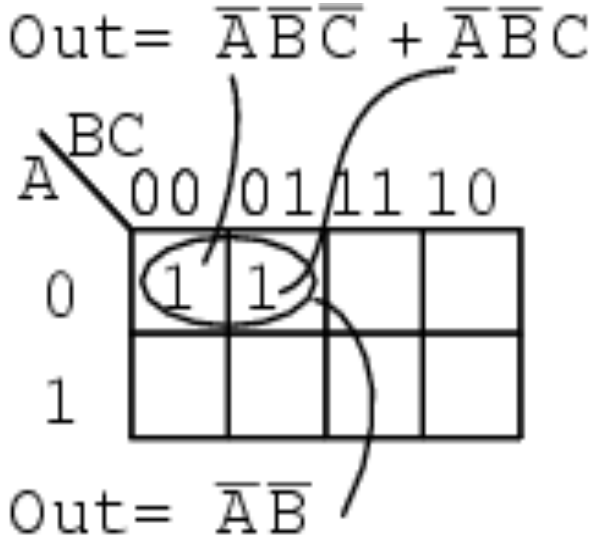
3 Variable K-Maps



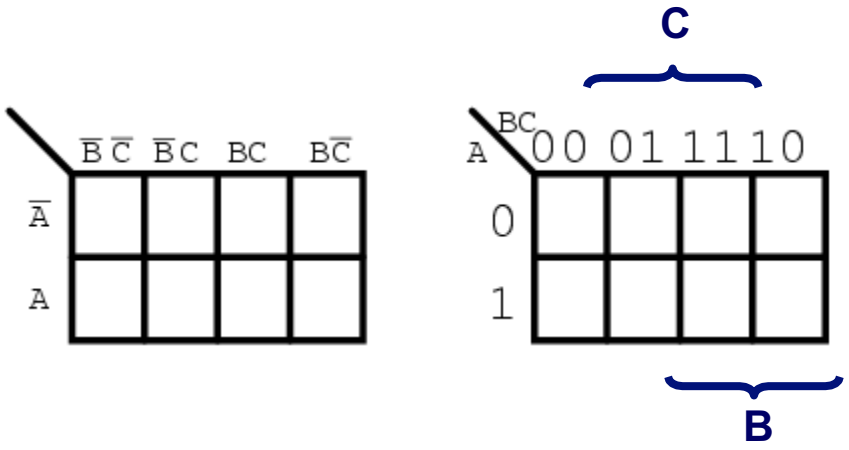
- Note in higher maps, several variables occupy a given axis
- The sequence of 1s and 0s follow a Gray Code Sequence.
- Grey code is a number system where two successive values differ only by 1-bit

3 Variable K-Maps (Example 1)

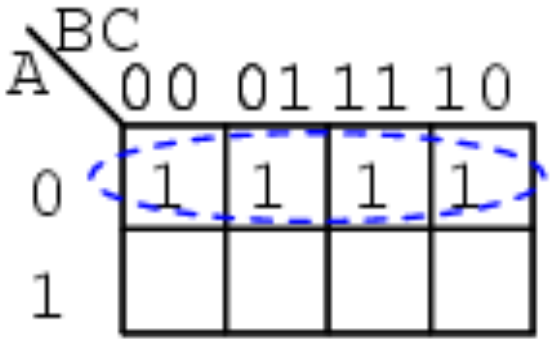




3 Variable K-Maps (Example 2)

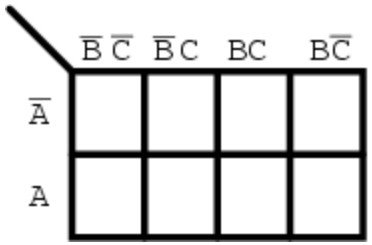


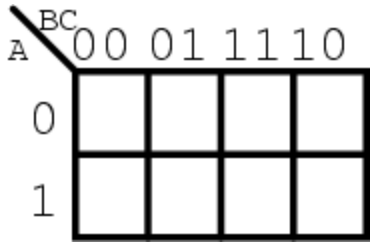
$$Out = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}BC + \overline{A}\,B\overline{C}$$



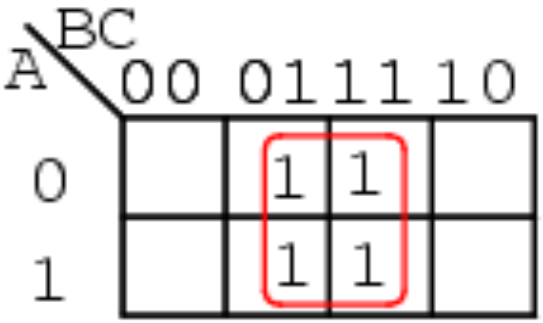
$$Out = \overline{A}$$

3 Variable K-Maps (Example 3)



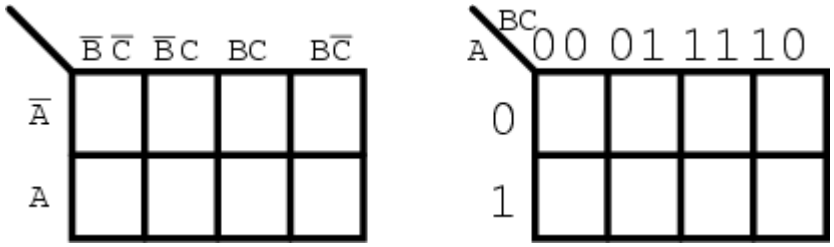


$$\text{Out} = \overline{A}\,\overline{B}C + \overline{A}\,BC + A\,\overline{B}C + ABC$$

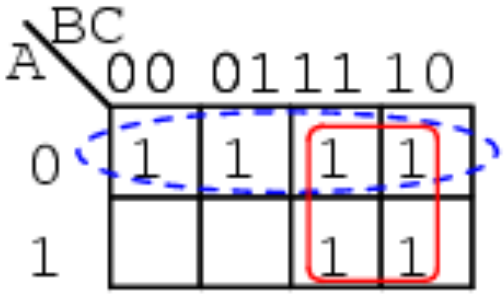


$$\text{Out} = C$$

3 Variable K-Maps (Example 4)



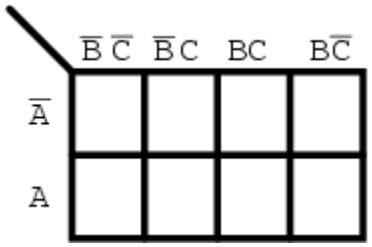
$$Out = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + \overline{A}\,B\,C + \overline{A}\,B\,\overline{C} + A\,B\,C + A\,B\,\overline{C}$$

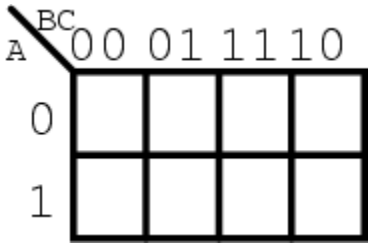


$$Out = \overline{A} + B$$

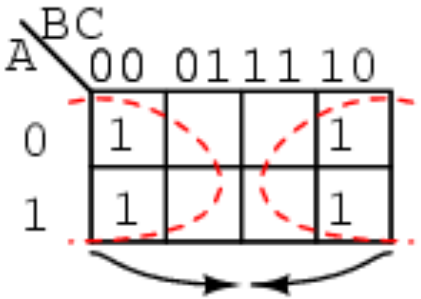
- Its usefulness come from the fact that adjacent squares correspond to minterms that differ in only ONE variable.
- **Combining 2^n squares eliminate n variables.**

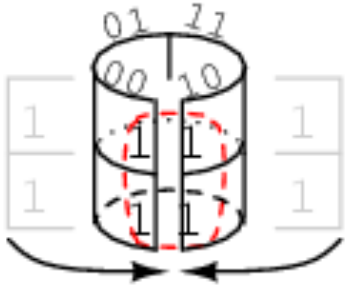
3 Variable K-Maps (Example 5)





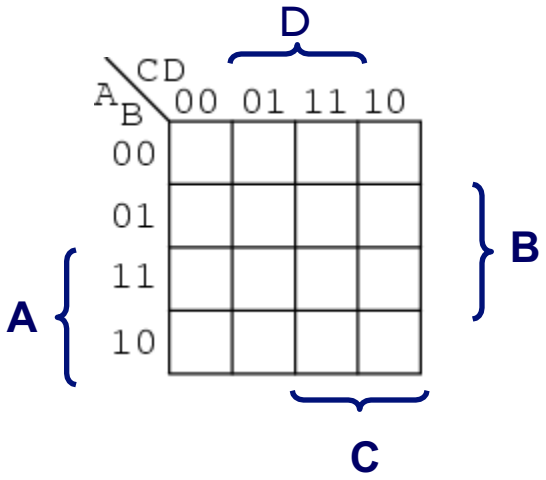
$$\text{Out} = \overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + \overline{A}\,B\,\overline{C} + A\,B\,\overline{C}$$



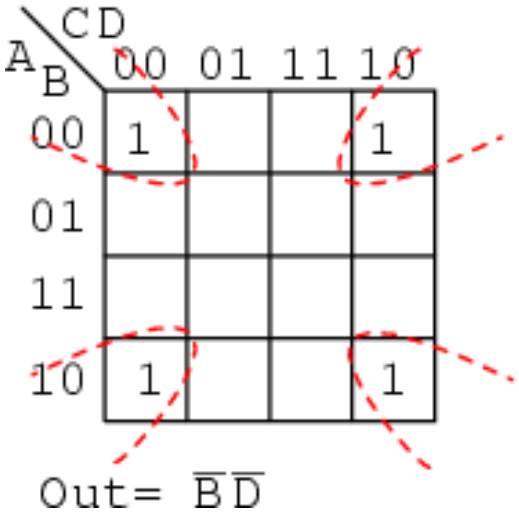


$$\text{Out} = \overline{C}$$

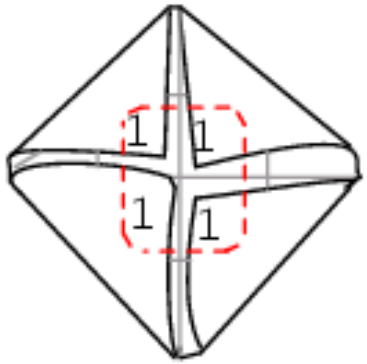
Up… up… and let's keep going



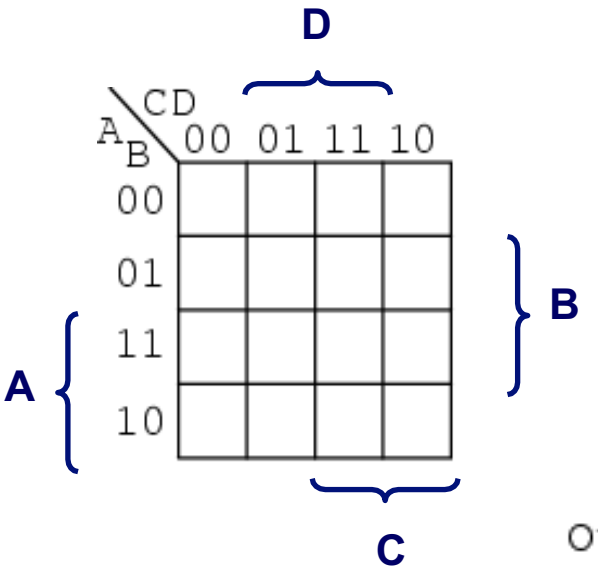
Out= $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,C\,\overline{D} + A\,\overline{B}\,\overline{C}\,\overline{D} + A\,\overline{B}\,C\,\overline{D}$



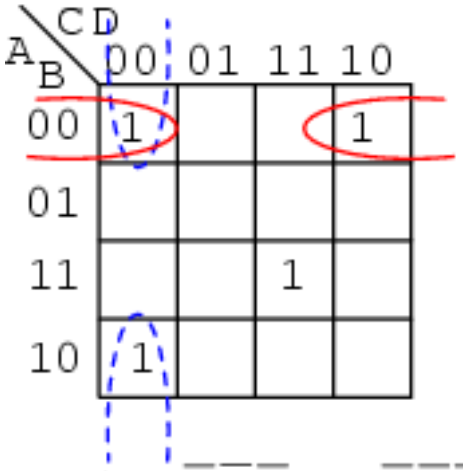
Out= $\overline{B}\,\overline{D}$



Few more examples

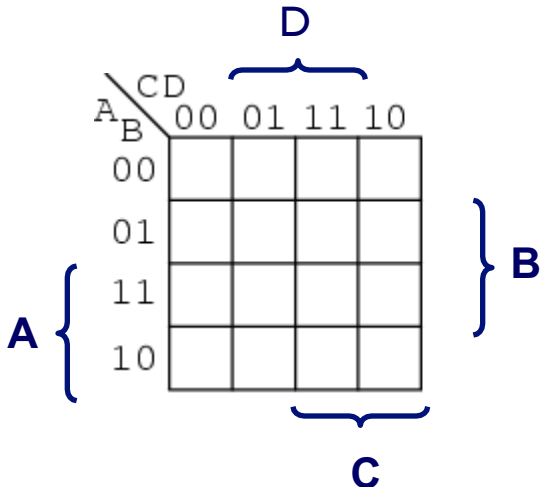


$$Out = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,C\,\overline{D} + A\,\overline{B}\,\overline{C}\,\overline{D} + A\,B\,C\,D$$

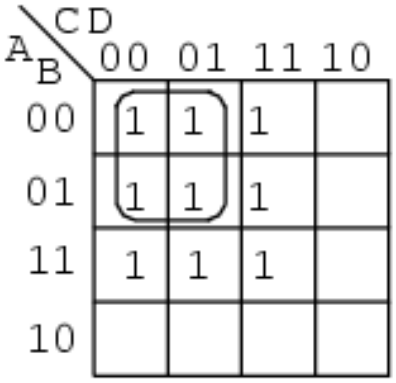


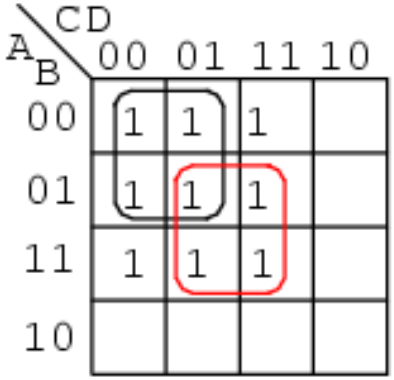
$$Out = \overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{D} + A\,B\,C\,D$$

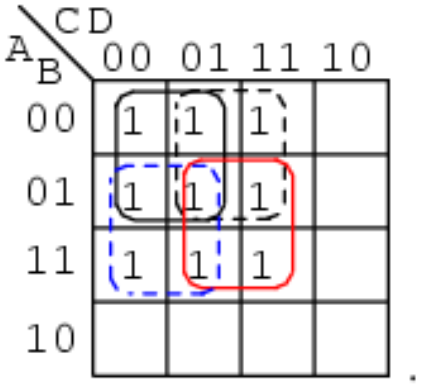
Few more examples



$$\text{Out} = \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}BCD + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}BCD + AB\overline{C}\overline{D} + AB\overline{C}D + ABCD$$







$$\text{Out} = \overline{A}\overline{C} + \overline{A}D + B\overline{C} + BD$$

Don't Care Conditions

- Let $F = AB + \overline{A}\,\overline{B}$
- Suppose we know that a disallowed input combo is A=1, B=0
- Can we replace F with a simpler function G whose output matches for all inputs we do care about?
- Let H be the function with Don't-care conditions for obsolete inputs

Inputs will not occur →

A	B	F	H	G
0	0	1	1	1
0	1	0	0	0
1	0	0	X	1
1	1	1	1	1

$$G = AB + \overline{B}$$

- Both F & G are appropriate functions for H
- G can substitute for F for valid input combinations

Don’t Cares can Greatly Simplify Circuits

X =0

CD \ AB	00	01	11	10
00				
01	1	1	X	1
11			X	X
10	1	1	X	X

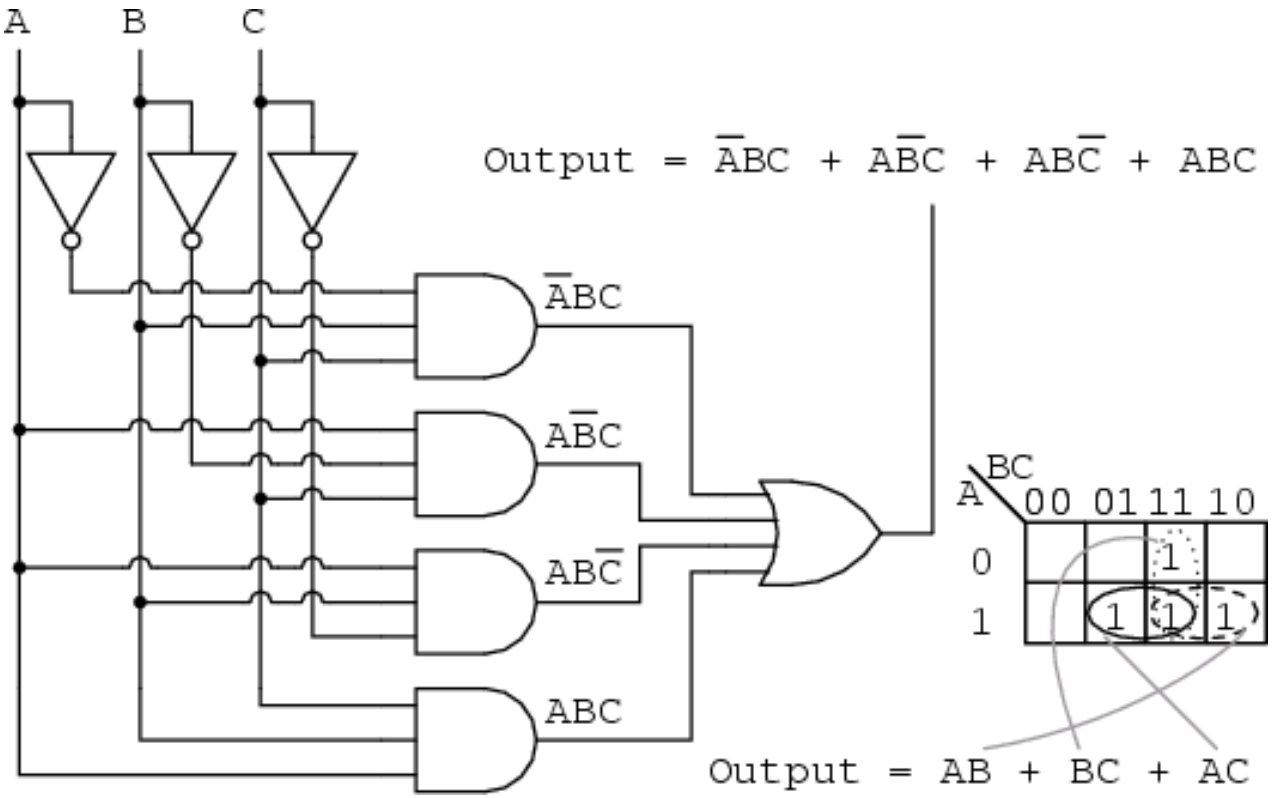
$f = \bar{A}\bar{C}D + \bar{B}\bar{C}D + \bar{A}C\bar{D}$

X=1

CD \ AB	00	01	11	10
00				
01	1	1	X	1
11			X	X
10	1	1	X	X

$f = \bar{C}D + C\bar{D}$

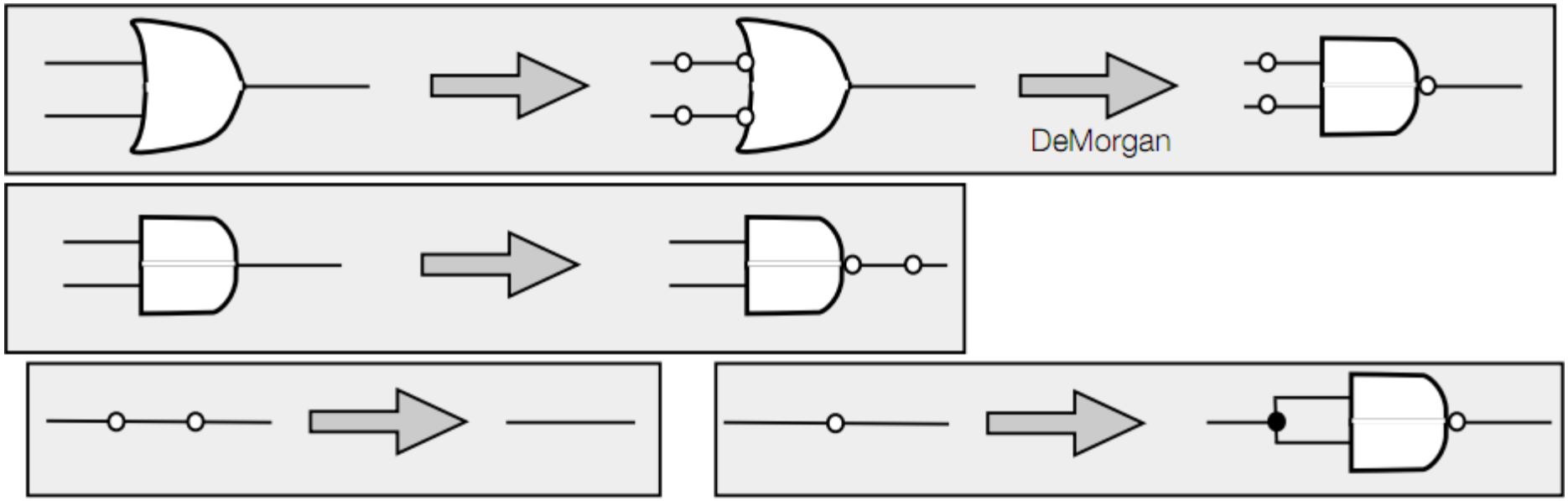
Back to our earlier example…..



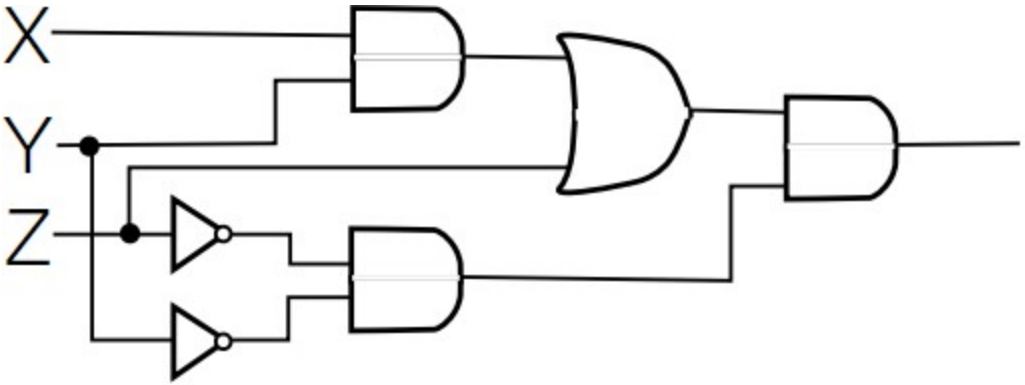
The K-map and the algebraic produce the same result.

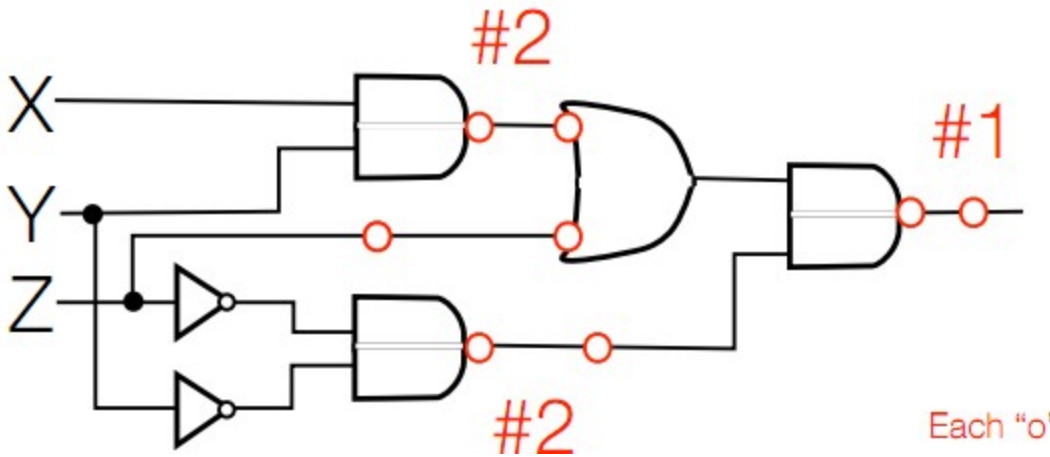
Converting Circuits to all-NAND (NOR)

- Introduce NOT gates
  - Go from left to right
  - Introduce NOT gates in pairs (does not alter logic function)
  - Isolated NOT gates can be implemented as NAND (NOR)



Example





Each “o” by itself represents a NOT gate

Put it all together…..

1. Express the truth table function in SOP (this example) or POS
2. Simplify the function using K-maps (this example) or boolean algebra

sensor inputs

A	B	C	Output
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

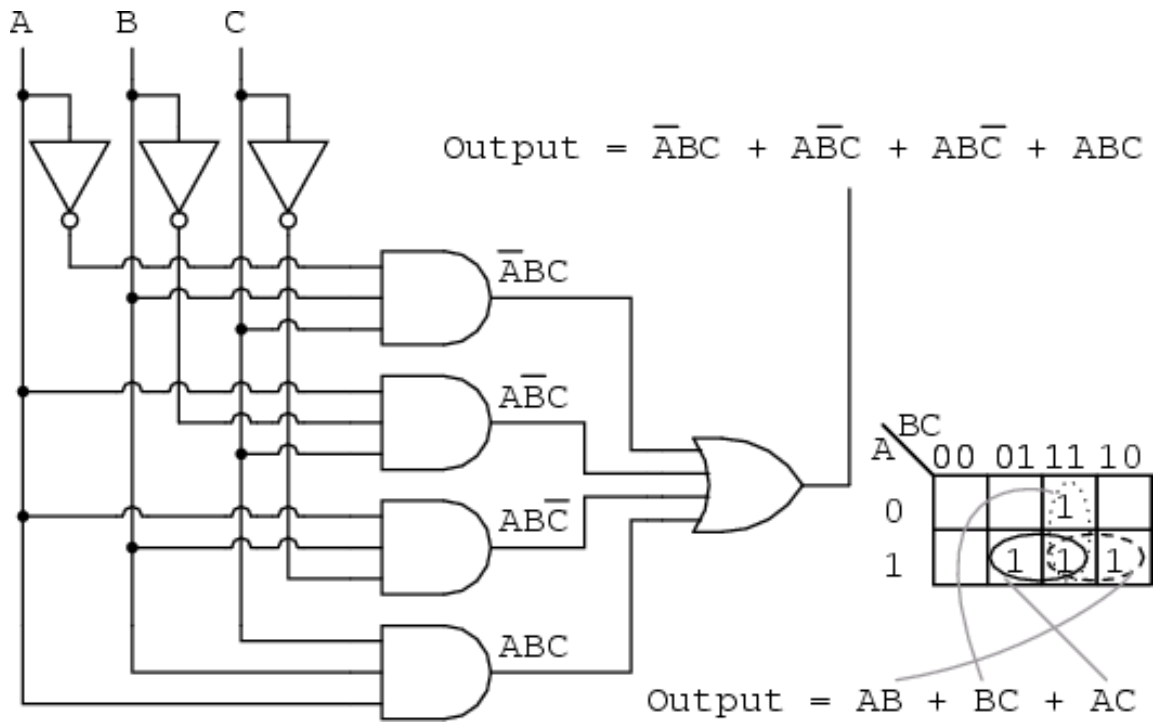
$\bar{A}BC = 1$

$A\bar{B}C = 1$

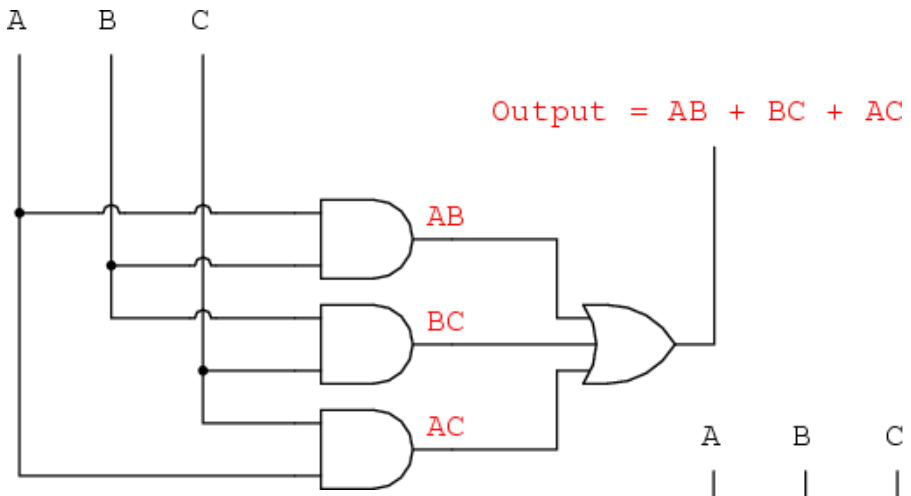
$AB\bar{C} = 1$

$ABC = 1$

$Output = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

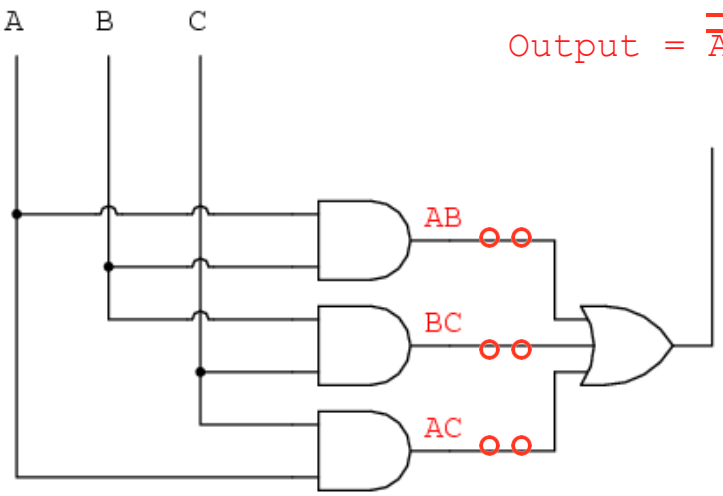


Put it all together…..



Output = AB + BC + AC

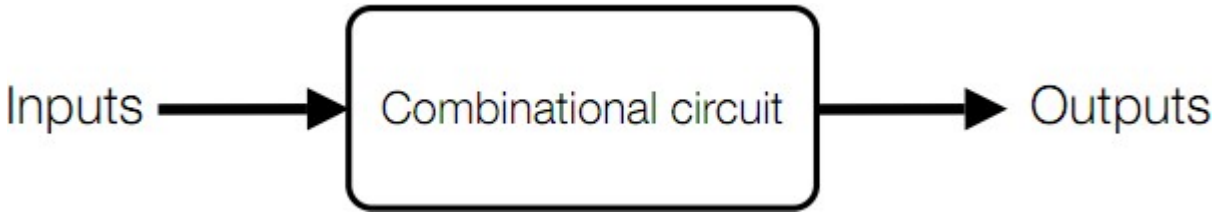
3. Convert function to use NAND or NOR gates



Output = $\overline{\overline{AB}\ \overline{BC}\ \overline{AC}}$

Combinational Circuits

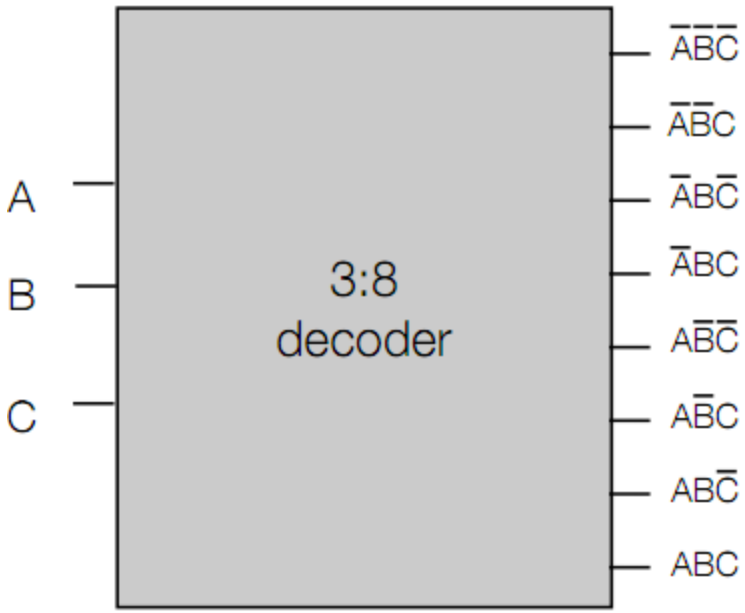
- What we've seen so far are combination circuits
- Realizes boolean functions (logical gates)
- Behavior is stateless
  - Outputs are function of inputs only



Decoder Circuits

Characteristic: for each input only one output is set

Converts n-bit input to m-bit output, where $n <= m <= 2^n$



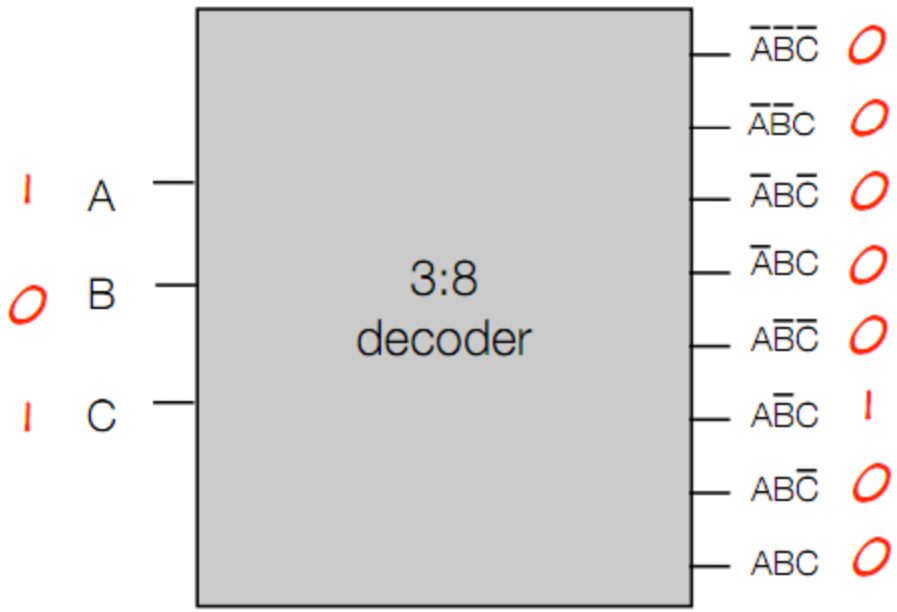
"Standard" Decoder: i^{th} output = 1, all others = 0,
where i is the binary representation of the input (ABC)

Decoder Example

Suppose each output has a numbered lamp

- the number is visible if the lamp is on

Converts n-bit input to m-bit output, where $n <= m <= 2^n$



e.g., ABC = 101 (i=5)

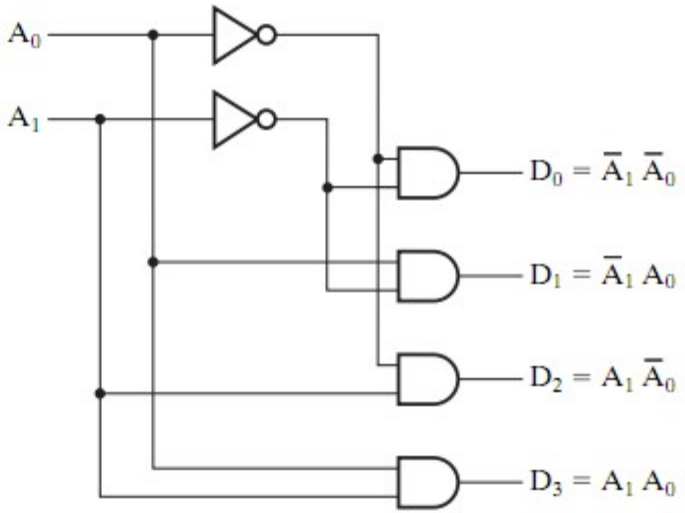
"Standard" Decoder: i^{th} output = 1, all others = 0,

where i is the binary representation of the input (ABC)

Internal 2:4 Decoder Design

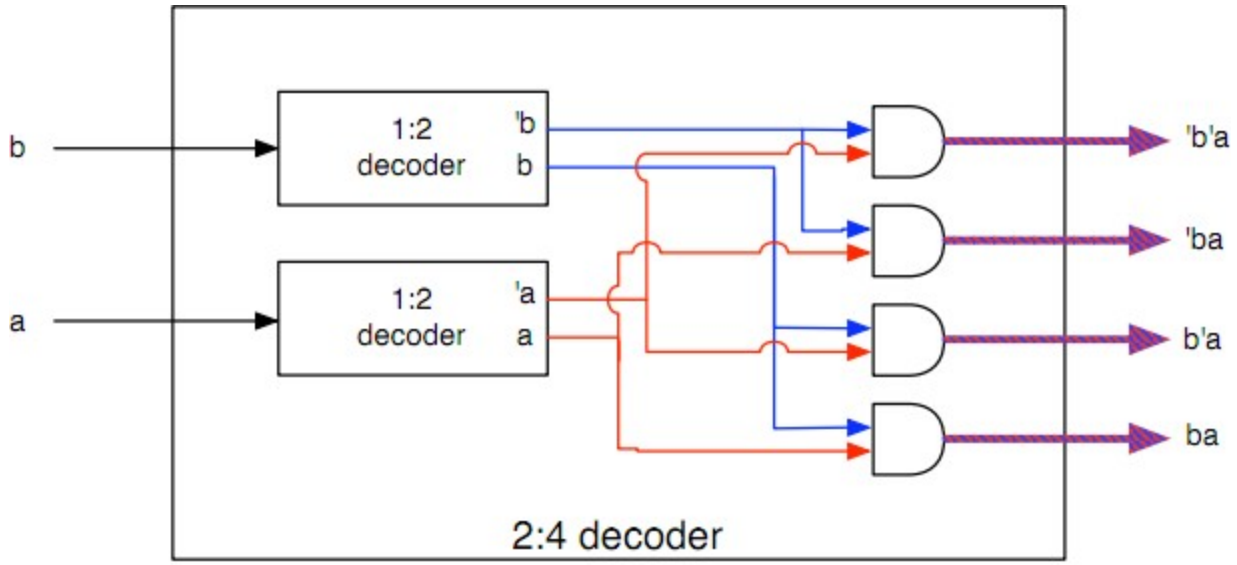
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



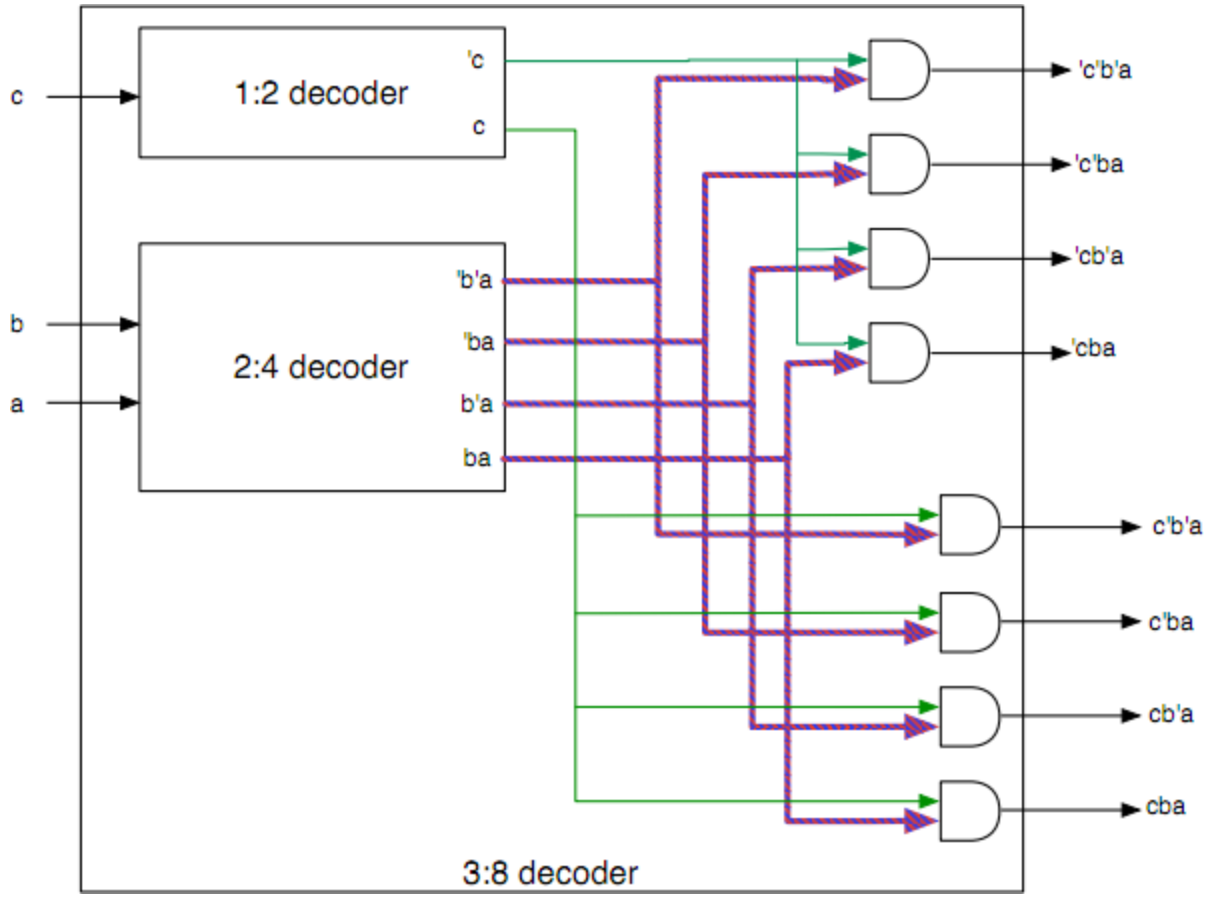
(b)

2:4 Decoder from 1:2 Decoders



Can build 2:4 decoder out of two 1:2 decoders
(and some additional circuitry)

Hierarchical 3:8 Decoder



Encoder: Inverse of Decoder

Inverse of decoder: converts m bit input to n bit output (n <= m)

TABLE 3-7
Truth Table for Octal-to-Binary Encoder

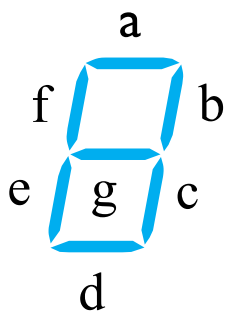
Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Code Converter

- Frequently we have some form of code and want to convert to another form of code and back
  - Encoder converts sound of voice into electrical signals (your cell phone) and a decoder converts the signals back to voice (your friend cell phone)
- Other times we just want to convert some code into another
  - Decoder and encoder are used in cascade
    - e.g. seven segment display

Code Converter Design Example

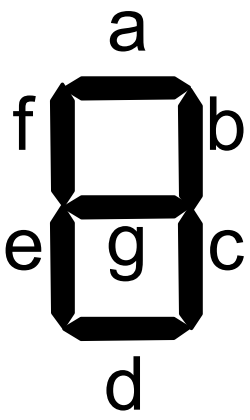
Each segment (LED) is identified by the letters from *a* to *g*, and can be lit individually



(a) Segment designation

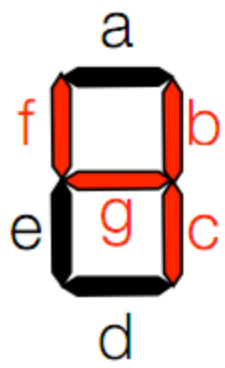
(b) Numeric designation for display

Code Converter Design Example



Input					Output						
Va	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
X	1	0	1	0	X	X	X	X	X	X	X
X	1	0	1	1	X	X	X	X	X	X	X
X	1	1	0	0	X	X	X	X	X	X	X
X	1	1	0	1	X	X	X	X	X	X	X
X	1	1	1	0	X	X	X	X	X	X	X
X	1	1	1	1	X	X	X	X	X	X	X

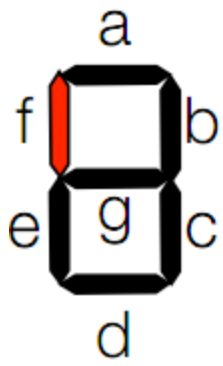
Code Converter Design Example



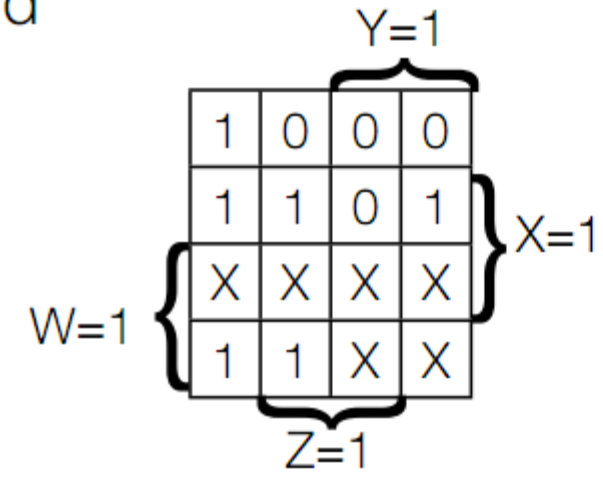
e.g., what outputs "lights up" when input V=4?

	Input				Output						
Va	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
X	1	0	1	0	X	X	X	X	X	X	X
X	1	0	1	1	X	X	X	X	X	X	X
X	1	1	0	0	X	X	X	X	X	X	X
X	1	1	0	1	X	X	X	X	X	X	X
X	1	1	1	0	X	X	X	X	X	X	X
X	1	1	1	1	X	X	X	X	X	X	X

Code Converter Design Example



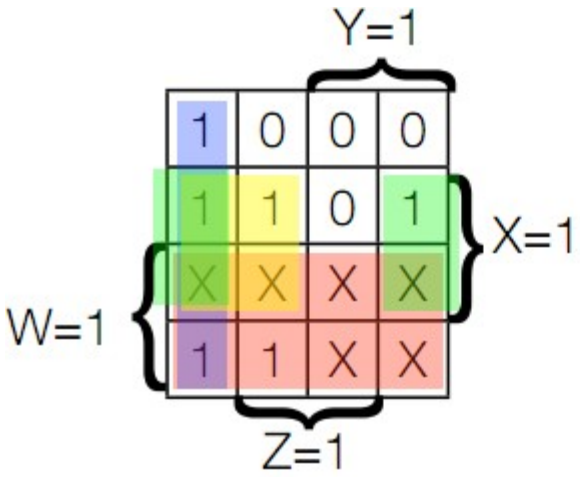
For what values does output f "light up" for?



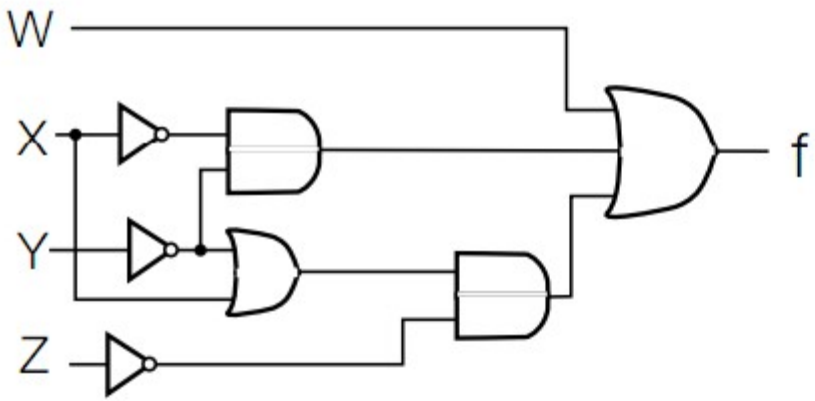
	Input				Output						
Va	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
X	1	0	1	0	X	X	X	X	X	X	X
X	1	0	1	1	X	X	X	X	X	X	X
X	1	1	0	0	X	X	X	X	X	X	X
X	1	1	0	1	X	X	X	X	X	X	X
X	1	1	1	0	X	X	X	X	X	X	X
X	1	1	1	1	X	X	X	X	X	X	X

Code Converter Design Example

We will do f, but you should be able to design a-e as well

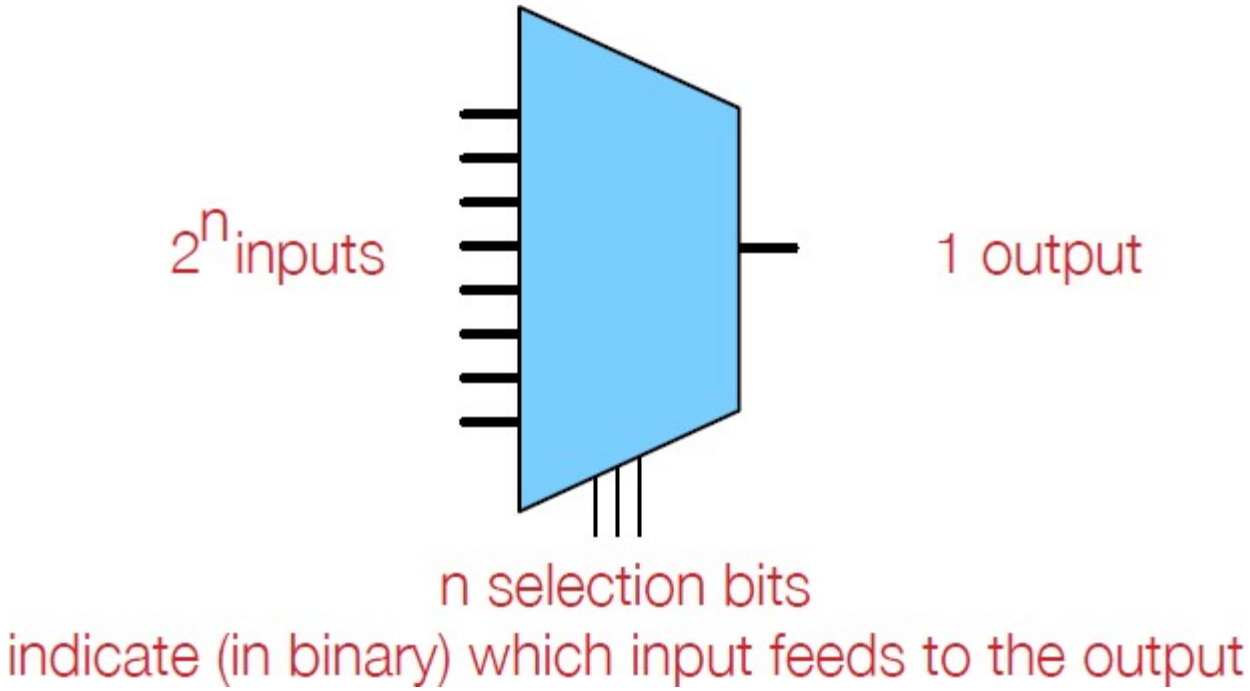


$$f = W + \overline{Y}\overline{Z} + X\overline{Z} + \overline{X}\overline{Y} = W + (X+\overline{Y})\overline{Z} + \overline{X}\overline{Y}$$



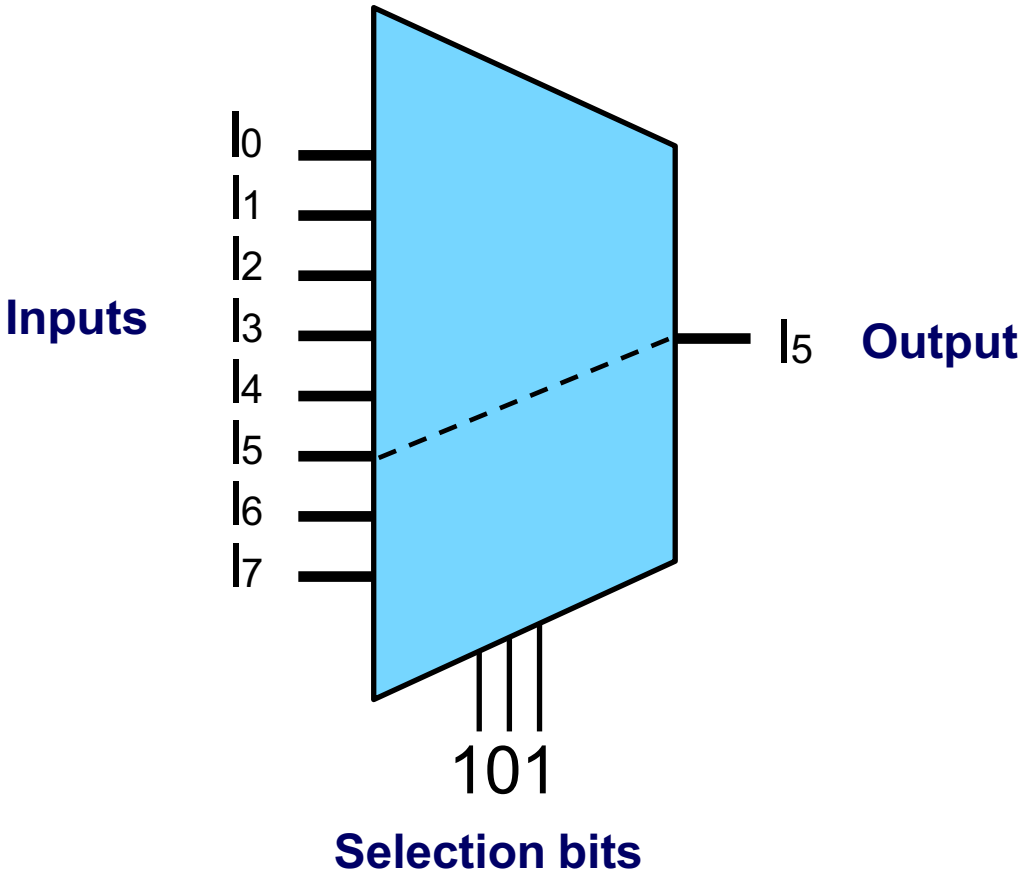
Multiplexers (Muxes)

Combinational circuit that selects binary information from one of many inputs to one output

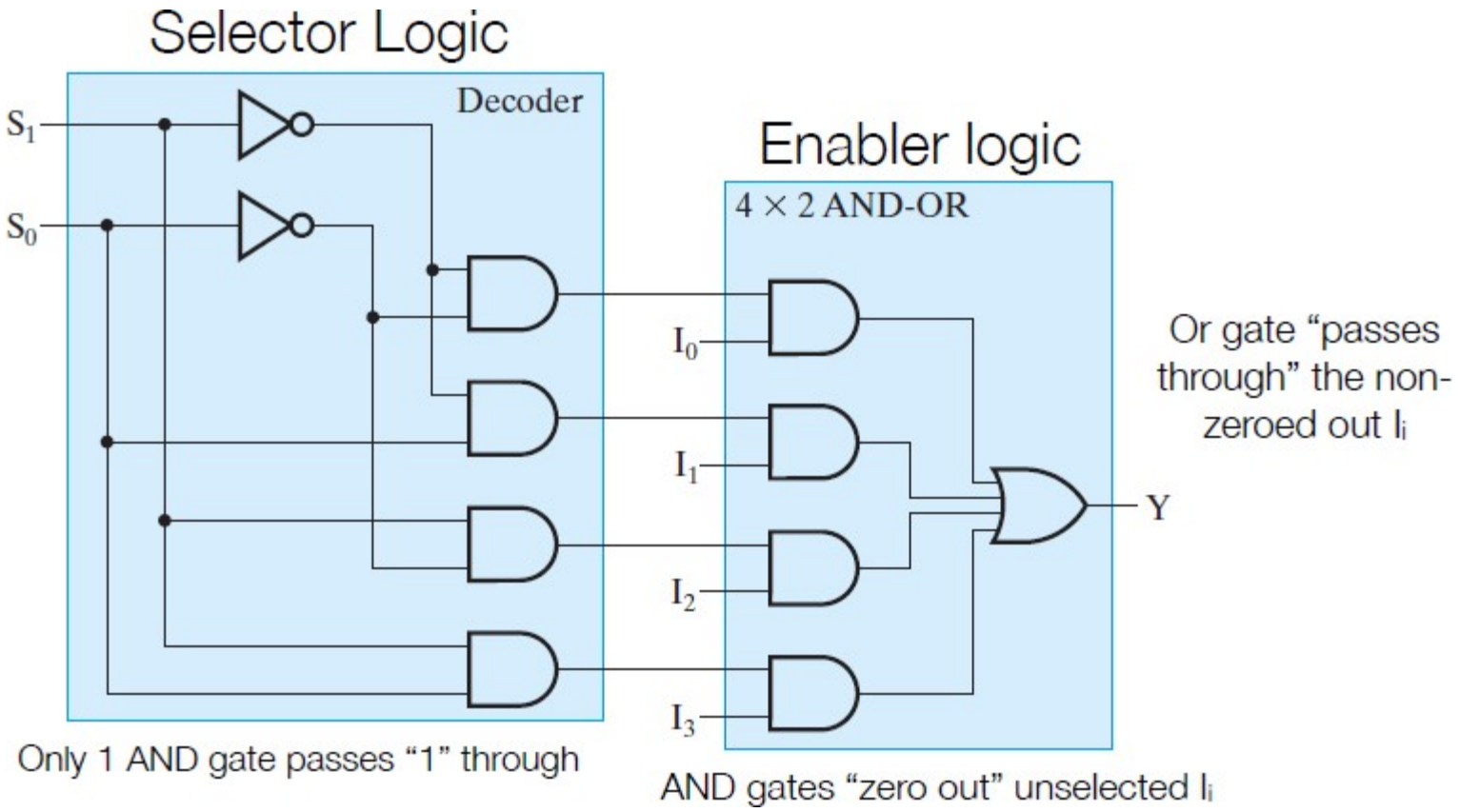


Multiplexer example

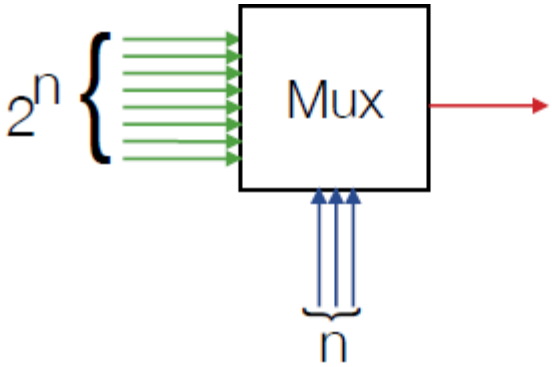
Input is seen in the output according to selector



Internal Mux Organization



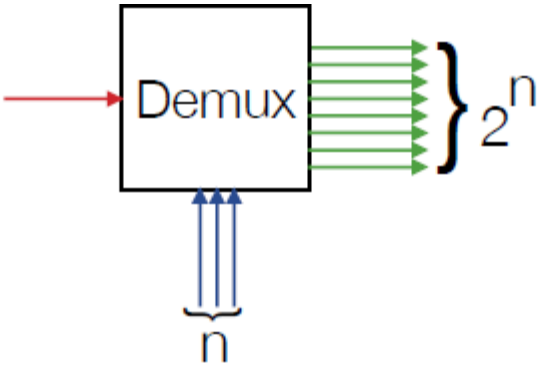
Mux Truth Table



2^n inputs								n-bit BCD value			1 output
a	x	x	x	x	x	x	x	0	0	0	a
x	b	x	x	x	x	x	x	0	0	1	b
x	x	c	x	x	x	x	x	0	1	0	c
x	x	x	d	x	x	x	x	0	1	1	d
x	x	x	x	e	x	x	x	1	0	0	e
x	x	x	x	x	f	x	x	1	0	1	f
x	x	x	x	x	x	g	x	1	1	0	g
x	x	x	x	x	x	x	h	1	1	1	h

Demultiplexer (Demux)

Selector specifies which output receives the input



1 input	n-bit BCD value			2^n outputs							
a	0	0	0	a	0	0	0	0	0	0	0
b	0	0	1	0	b	0	0	0	0	0	0
c	0	1	0	0	0	c	0	0	0	0	0
d	0	1	1	0	0	0	d	0	0	0	0
e	1	0	0	0	0	0	0	e	0	0	0
f	1	0	1	0	0	0	0	0	f	0	0
g	1	1	0	0	0	0	0	0	0	g	0
h	1	1	1	0	0	0	0	0	0	0	h

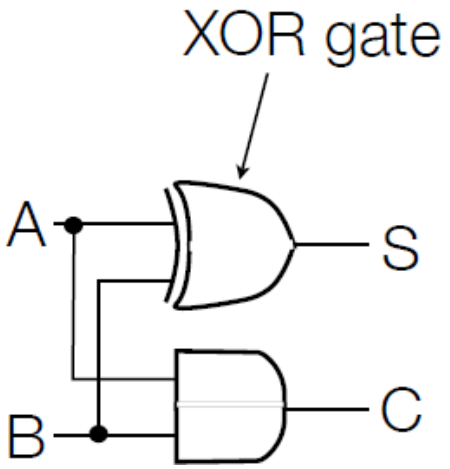
Addition: The Half Adder

Addition of 2 bits: A & B produces summand (S) and carry (C)

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S = A \oplus B$

$C = AB$



But to do addition, we need 3 bits at a time (to account for carries)

```
  011   <-- carry bits
+ 1011
  1001
 10101
```

The Full Adder

Takes as input 2 digits (A&B) and previous carry (P)

P	A	B	C	S
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

S:

P \ B	0	1	0	1
1	1	0	1	0

A

XOR:checker-board pattern

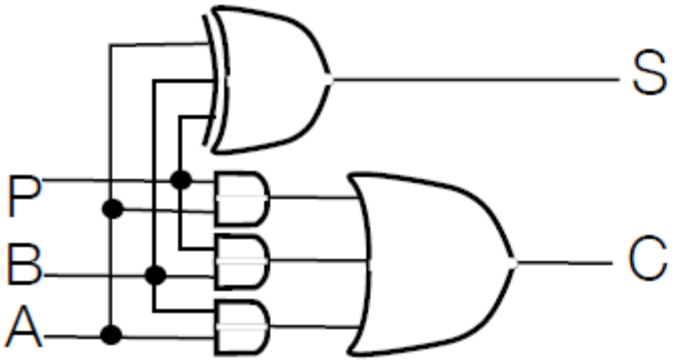
$$S = A \oplus B \oplus P$$

C:

P \ B	0	0	1	0
0	0	1	1	1

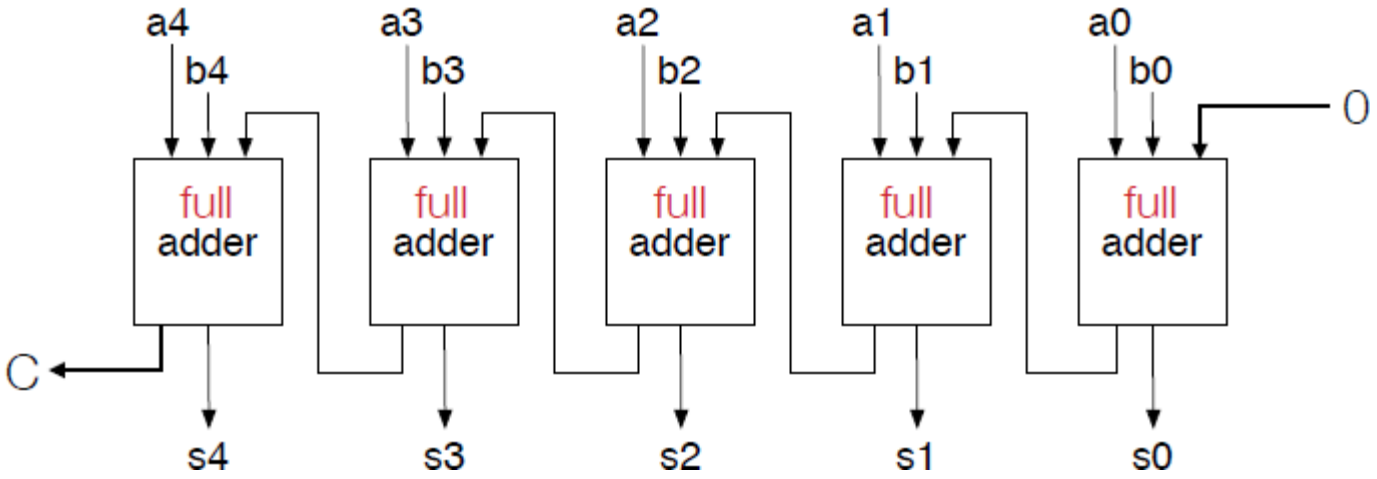
A

$$C = AB + AP + BP$$



5-bit Ripple Carry Adder

Computes $a_4a_3a_2a_1a_0 + b_4b_3b_2b_1b_0$



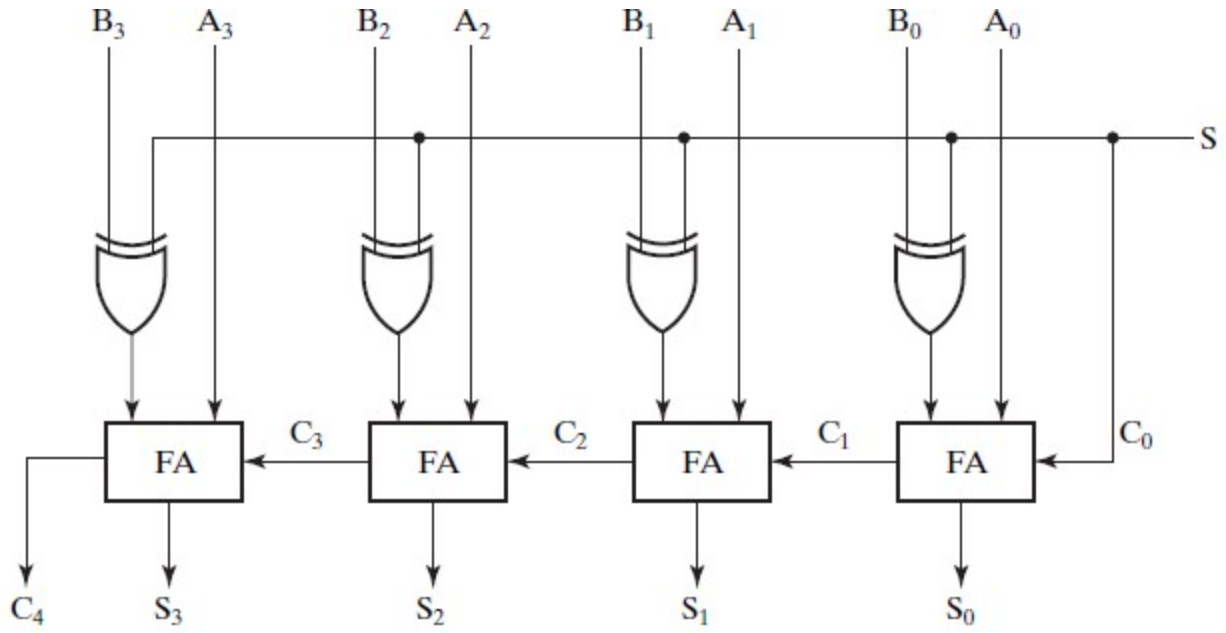
Note how computation ripples from left to right

- Each adder has depth 2 (input passes through 2 gates to reach output)
- Full adder that computes s_i cannot start its computation until previous full adder computes carry
- The longest depth in a k-bit ripple carry adder is 2k

Adder-Subtractor circuit

How to change the adder to add and subtract using same circuit?

- Add extra bit S to switch between addition and subtraction
- Add XOR gate at the second input to full-adder



S=0: B unchanged, C_0=0: add

S=1: B complemented, C_0=1 (bits flipped and 1 added): subtract

- Addition: A + B
- Subtraction: A – B = A + (complement of B + 1)

Handling 2's Complement Overflow

Adding 2 positive numbers ➔➔ negative result

```
      1
  6   0110
+ 5   0101
-----------
 -5   1011
```

Adding 2 negative numbers ➔➔ positive result

```
      1 1
 -6   1010
+ -6  1010
------------
  4  10100
```

Handling 2's Complement Overflow

- Two cases (look at two most significant carries):
  1. 1 carried in (c3), and 0 carried out (c4)
     - Only if a3 = 0 and b3 = 0
  2. 0 carried in (c3), and 1 carried out (c4)
     - Only if a3 = 1 and b3 = 1

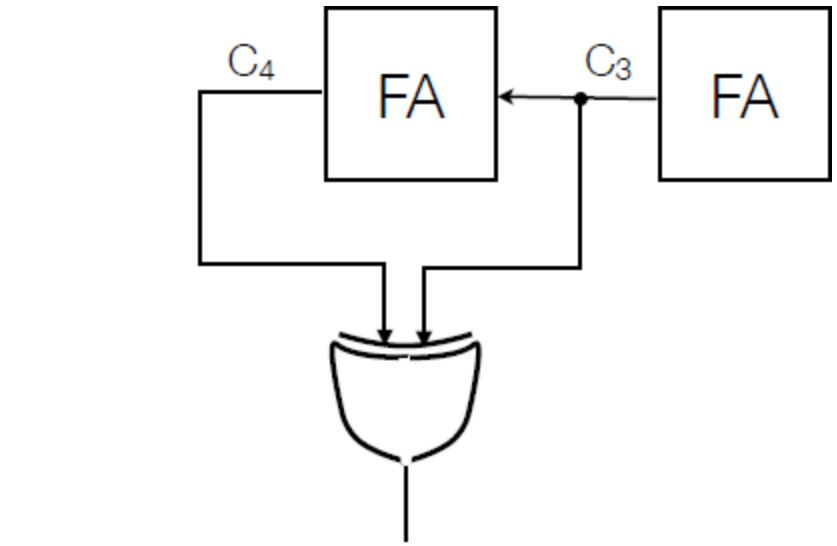
```
c4  c3  c2  c1  c0
    a3  a2  a1  a0
    b3  b2  b1  b0
    ---------------
    s3  s2  s1  s0
```

a3	b3	c3	c4	s3	overflow	
0	0	0	0	0	0	sum of two pos is pos
0	0	1	0	1	1	
0	1	0	0	1	0	
0	1	1	1	0	0	
1	0	0	0	1	0	
1	0	1	1	0	0	
1	1	0	1	0	1	sum of two negs is neg
1	1	1	1	1	0	

n bit two's comp: -2^n <---> 2^n - 1
split into pos and neg ranges and find smallest and largest possible results. show that they're in range for twos comp.

Overflow Computation in Adder/Subtractor

For 2s complement, overflow if 2 most significant carries differ

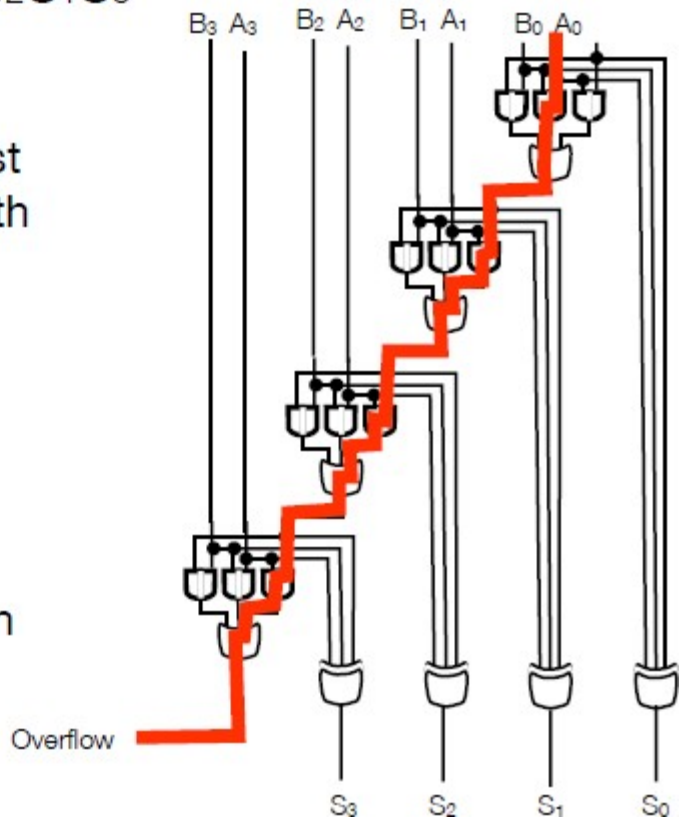


Ripple Carry Adder Circuit Depth

Have to wait for the carry to be computed

$A_3A_2A_1A_0 + B_3B_2B_1B_0 = S_3S_2S_1S_0$

- Depth of a circuit is the longest (most gates to go through) path
- Overflow has depth 8
- S_3 has depth 7
- In general, S_i has depth 2i+1 in Ripple-Carry Adder



Carry Lookahead Adder (CLA)

Goal: produce an adder circuit of shorter depth

Mechanism: rewrite the carry function

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$

$$c_{i+1} = a_i b_i + c_i (a_i + b_i)$$

$$c_{i+1} = g_i + c_i (p_i)$$

Generates carry out ($c_{i+1}=1$) if $a_i = b_i = 1$

Propagates carry in ($c_i = 1$) If either $a_i = 1$ or $b_i = 1$

carry generate

$g_i = a_i b_i$

carry propagate

$p_i = a_i + b_i$

Now, look at g_i and p_i

They both use **only** a_i and b_i, neither depend on the carry. If we write the carry function in terms of g_i and p_i we might avoid waiting for the carry.

Carry Lookahead Adder (CLA)

Recursively define carries in terms of propagate and generate signals

$$c_1 = g_0 + c_0 p_0$$

$$c_2 = g_1 + c_1 p_1$$

$$= g_1 + (g_0 + c_0 p_0) p_1$$

$$= g_1 + g_0 p_1 + c_0 p_0 p_1$$

$$c_3 = g_2 + c_2 p_2$$

$$= g_2 + (g_1 + g_0 p_1 + c_0 p_0 p_1) p_2$$

$$= g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$

We can compute carries in terms of a_i's, b_i's and c0. This bits are available right away, we don't have to wait for them.

i^{th} carry has i+1 product terms, the largest of which has i+1 literals

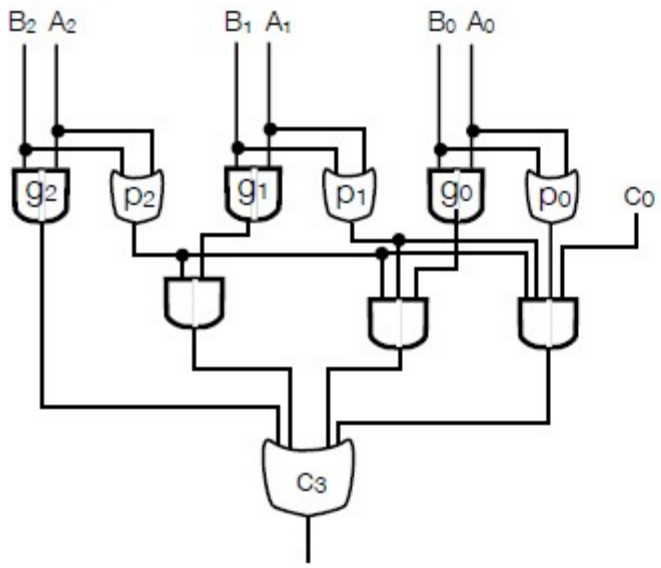
Carry circuit depth is 3 (SoP form)

If gates take 2 inputs, total circuit depth is 1 + $\log_2(k)$ for k-bit addition

Carry Lookahead Adder (CLA)

$$c_0 = 0$$
$$c_1 = g_0 + c_0 p_0$$
$$c_2 = g_1 + g_0 p_1 + c_0 p_0 p_1$$
$$c_3 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$
$$c_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$$

$$s_0 = a_0 \oplus b_0 \oplus c_0$$
$$s_1 = a_1 \oplus b_1 \oplus c_1$$
$$s_2 = a_2 \oplus b_2 \oplus c_2$$
$$s_3 = a_3 \oplus b_3 \oplus c_3$$
$$s_4 = a_4 \oplus b_4 \oplus c_4$$



Depth of 3 for all c_i

Depth of 4 for all s_i, $i>0$

Note: gates take only 2 inputs: depth increases by a log2 factor: still much less than linear of ripple-carry adder

Next Topic

Sequential circuits