211: Computer Architecture
Spring 2021

Instructor: Prof. David Menendez

Topics:

- Digital Logic

- Reading material available on Sakai

Logic Design

How does your processor perform various operations?

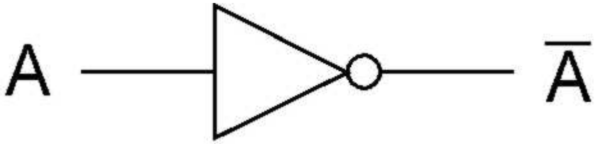
Logic Gates

Transition from representing information to implementing them

Logic gates are simple digital circuits

- Take one or more binary inputs
- Produce a binary output
- Truth table: relationship between the input and the output

Not Gate



Truth table

In	Out
0	1
1	0

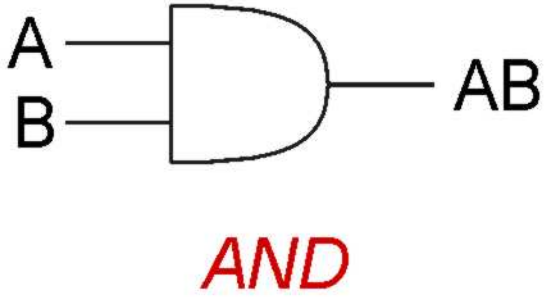
Simplest Gate

AND Gate

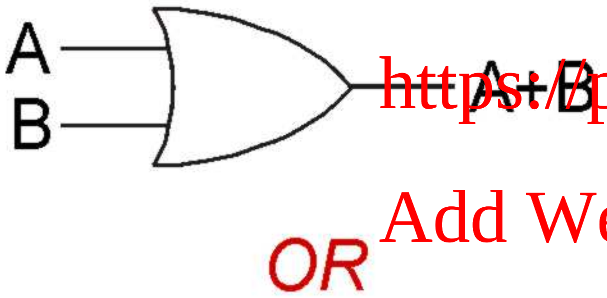
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Two inputs, One output

Result is 1 only if both the inputs are 1.



OR Gate

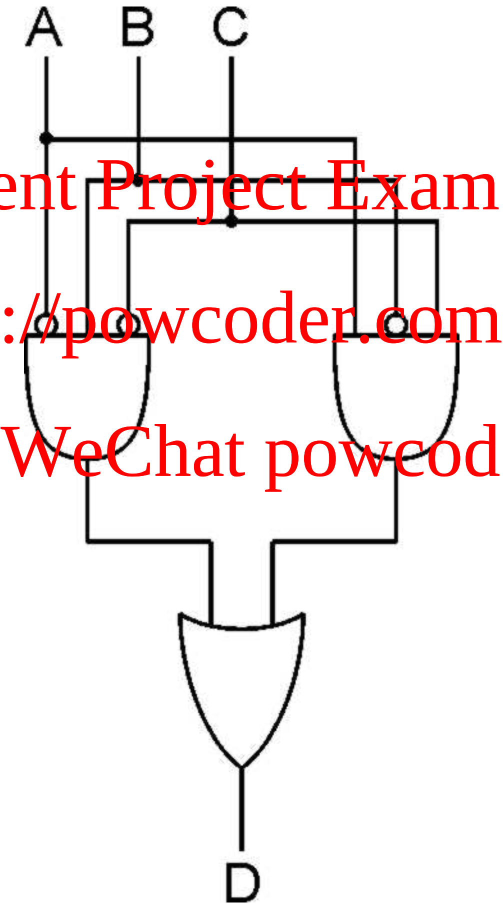


A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Logical Completeness

Can implement ANY truth table with AND, OR, NOT.

A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

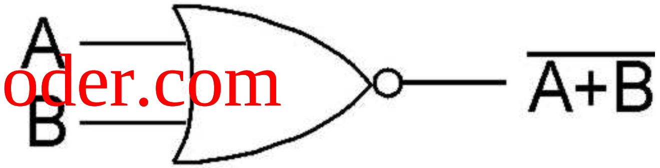


1. AND combinations that yield a "1" in the truth table.

2. OR the results of the AND gates.

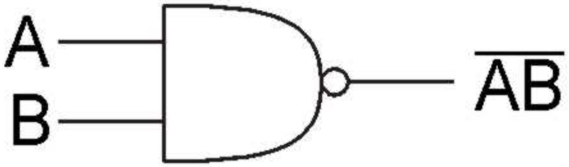
NAND and NOR Gate

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



NOR

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



NAND

Beneath the Digital Abstraction

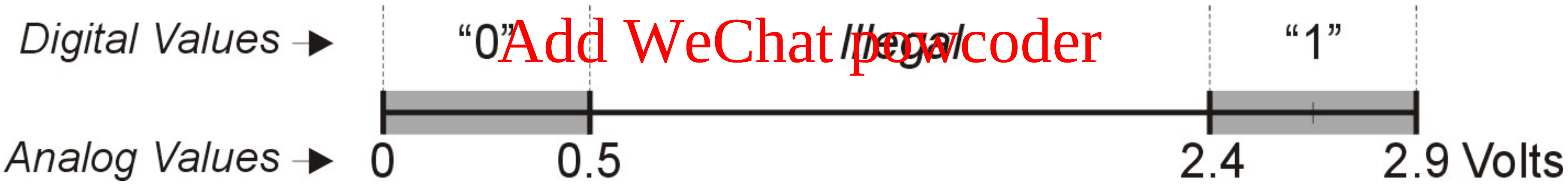
A digital system uses discrete values

- Represent it with continuous variables (eg, voltage), handle noise

Use transistors to implement logical functions: AND, OR, NOT

Digital symbols:

- recall that we assign a range of analog voltages to each digital (logic) symbol



- assignment of voltage ranges depends on electrical properties of transistors being used
  - typical values for "1": +5V, +3.3V, +2.9V
  - from now on we'll use +2.9V

Transistor: Building Block of Computers

Microprocessors contain millions (billions) of transistors

- Intel Pentium 4 (2000): 48 million
- IBM PowerPC 750FX (2002): 38 million
- IBM/Apple PowerPC G5 (2003): 58 million

Logically, each transistor acts as a switch

Combined to implement logic functions

- AND, OR, NOT

Combined to build higher-level structures

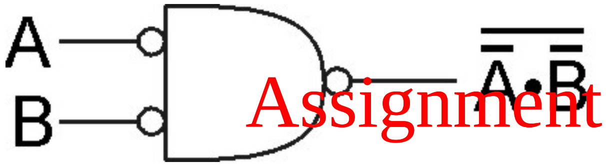
- Adder, multiplexer, decoder, register, …

Combined to build processor

DeMorgan's Law

Converting AND to OR (with some help from NOT)

Consider the following gate:



To convert AND to OR (or vice versa), invert inputs and output.

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

Same as A+B!

Generally, DeMorgan's Laws:

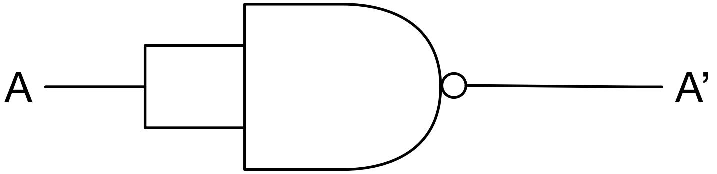
1. $\overline{PQ} = \bar{P} + \bar{Q}$
2. $\overline{P + Q} = \bar{P} \; \bar{Q}$

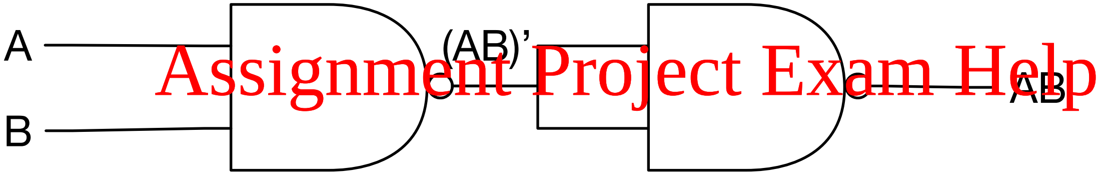
NAND and NOR Functional Completeness

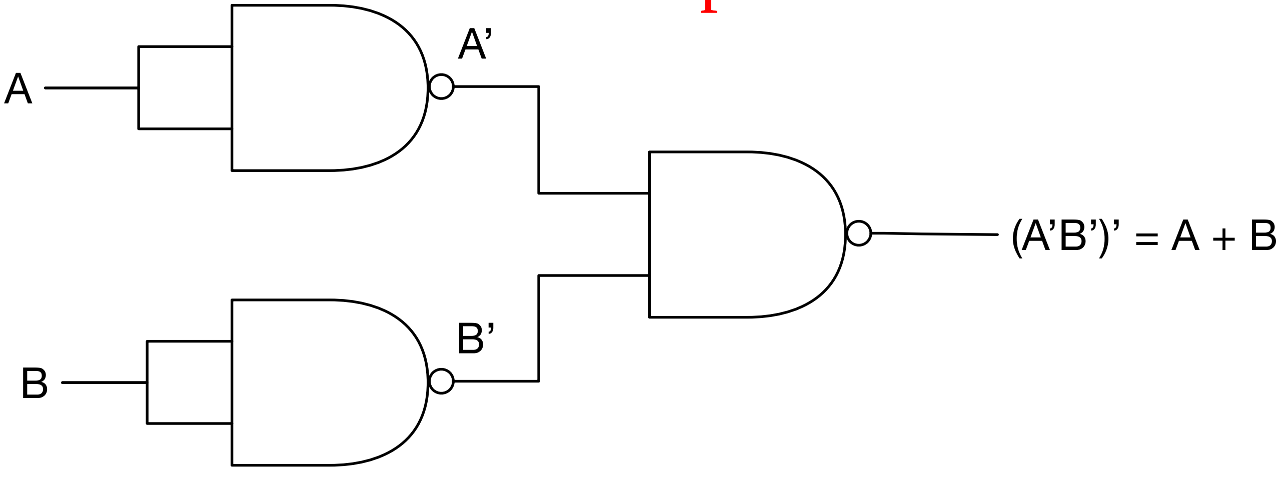
Any gate can be implemented using either NOR or NAND gates.

Why is this important?

- When building a chip, easier to build one with all of the same gates.


A
A’


A
B
(AB)’
AB

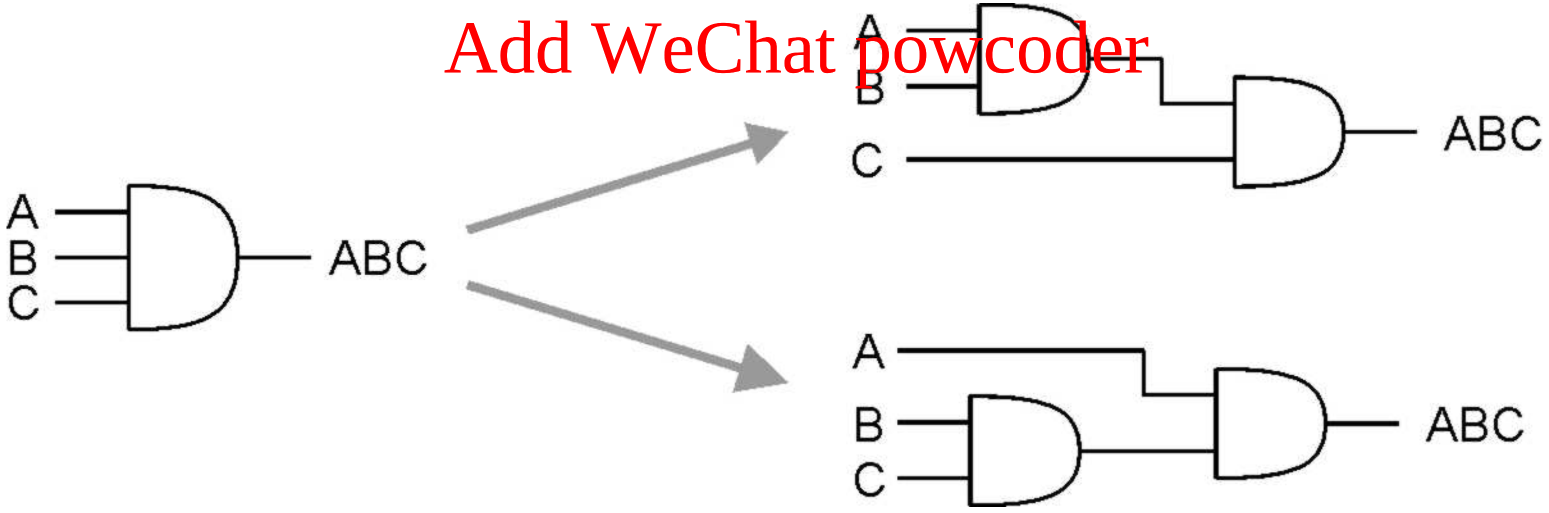

A
A’
B
B’
(A’B’)’ = A + B

More than 2 Inputs?

AND/OR can take any number of inputs.

- AND = 1 if all inputs are 1.
- OR = 1 if any input is 1.
- Similar for NAND/NOR.

Can implement with multiple two-input gates or with single CMOS circuit.



Circuit Design

Have a good idea. What kind of circuit might be useful?

Derive a truth table for this circuit

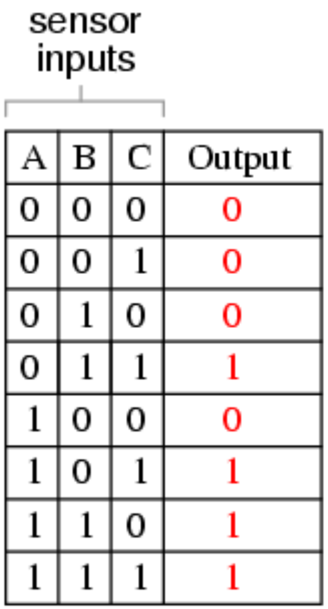
Derive a Boolean expression for the truth table

Build a circuit given the Boolean expression

- Building the circuit involves mapping the Boolean expression to actual gates. This part is easy.

- Deriving the Boolean expression is easy. Deriving a good one is tricky.

Converting Truth Table to Boolean Expression

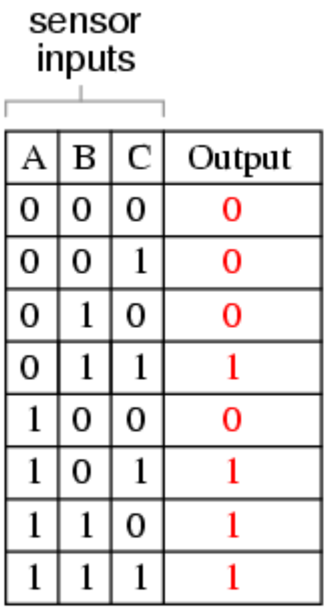


sensor inputs

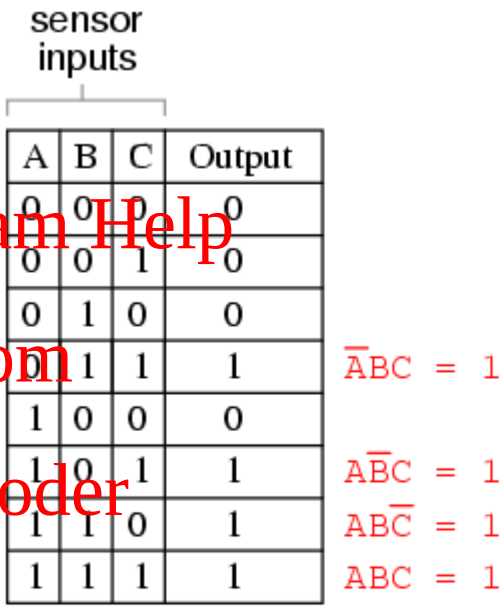
A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Given a circuit, isolate the rows in which the output of the circuit should be true

Converting 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

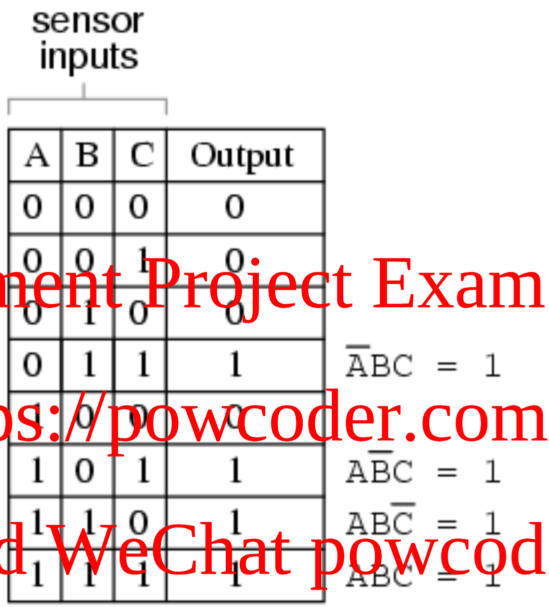

sensor inputs

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 a circuit, isolate that rows in which the output of the circuit should be true

A product term that contains exactly one instance of every variable is called a minterm

Converting 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	$\overline{A}BC = 1$
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	$ABC = 1$

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

Given the expressions for each row, build a larger Boolean expression for the entire table.

- This is a sum-of-products (SOP) form.

Canonical Forms

We have studied two canonical forms

1. Sum of Products (SoP)
2. Product of Sums (PoS)

How to convert to SoP from PoS (multiple through)

How to convert to PoS from SoP (complement, multiply through, complement via DeMorgan's)

Note: $X' = \overline{X}$

$$F = Y'Z' + XY'Z + XYZ'$$

$$F' = (Y+Z)(X'+Y+Z')(X'+Y'+Z)$$

$$= YZ + X'Y + X'Z \qquad \text{(after lots of simplification)}$$

$$F = (Y'+Z')(X+Y')(X+Z')$$

Formal Definition of Minterms

e.g., Minterms for 3 variables A,B,C

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

- A product term in which all variables appear once, either complemented or uncomplemented (i.e., an entry in the truth table).
- Each minterm evaluates to 1 for exactly one variable assignment, 0 for all others.
- Denoted by mX where X corresponds to the variable assignment for which mX = 1.

Converting 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

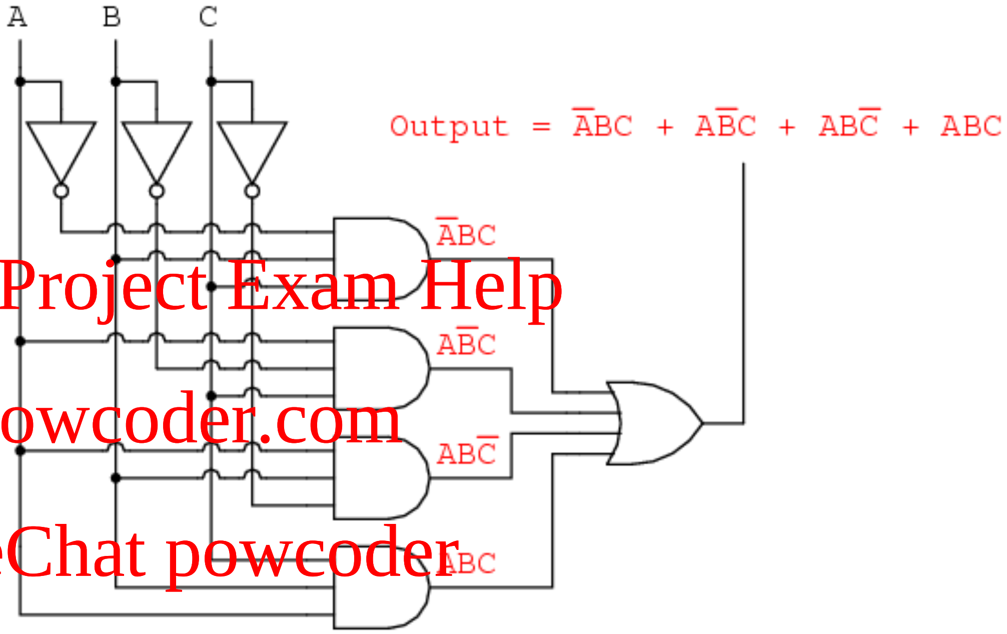
$\overline{A}BC = 1$

$A\overline{B}C = 1$

$AB\overline{C} = 1$

$ABC = 1$

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



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

Finally build the circuit.

- Problem: SOP forms are often not minimal.
- Solution: Make it minimal. We’ll go over two ways.

First Approach: Algebraic

Simply use the rules of Boolean logic

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

$\downarrow$ Factoring **BC** out of 1st and 4th terms

$BC(\bar{A} + A) + A\bar{B}C + AB\bar{C}$

$\downarrow$ Applying identity $\mathbf{A + \bar{A} = 1}$

$BC(1) + A\bar{B}C + AB\bar{C}$

$\downarrow$ Applying identity $\mathbf{1A = A}$

$BC + A\bar{B}C + AB\bar{C}$

$\downarrow$ Factoring **B** out of 1st and 3rd terms

$B(C + A\bar{C}) + A\bar{B}C$

$\downarrow$ Applying rule $\mathbf{A + \bar{A}B = A + B}$ to the $C + A\bar{C}$ term

$B(C + A) + A\bar{B}C$

$\downarrow$ Distributing terms

$BC + AB + A\bar{B}C$

$\downarrow$ Factoring **A** out of 2nd and 3rd terms

$BC + A(B + \bar{B}C)$

$\downarrow$ Applying rule $\mathbf{A + \bar{A}B = A + B}$ to the $B + \bar{B}C$ term

$BC + A(B + C)$

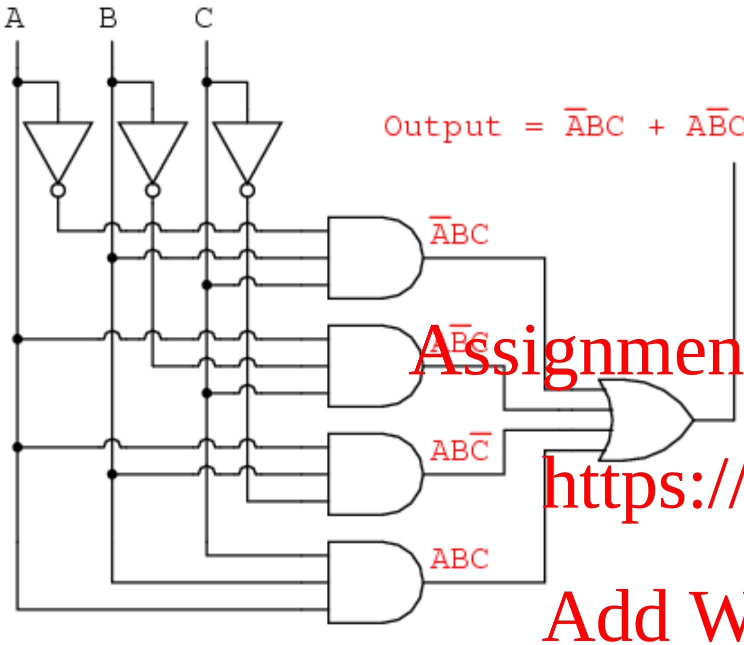
$\downarrow$ Distributing terms

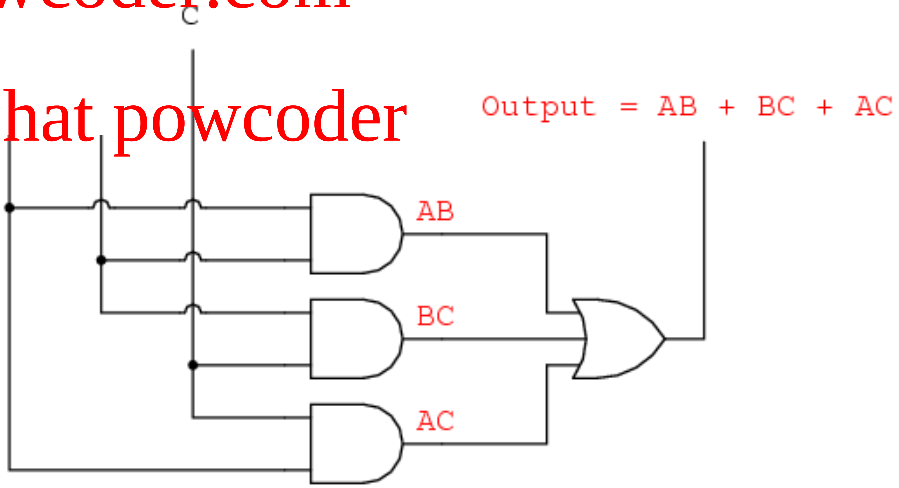
$BC + AB + AC$

or Simplified result

$AB + BC + AC$

The Result

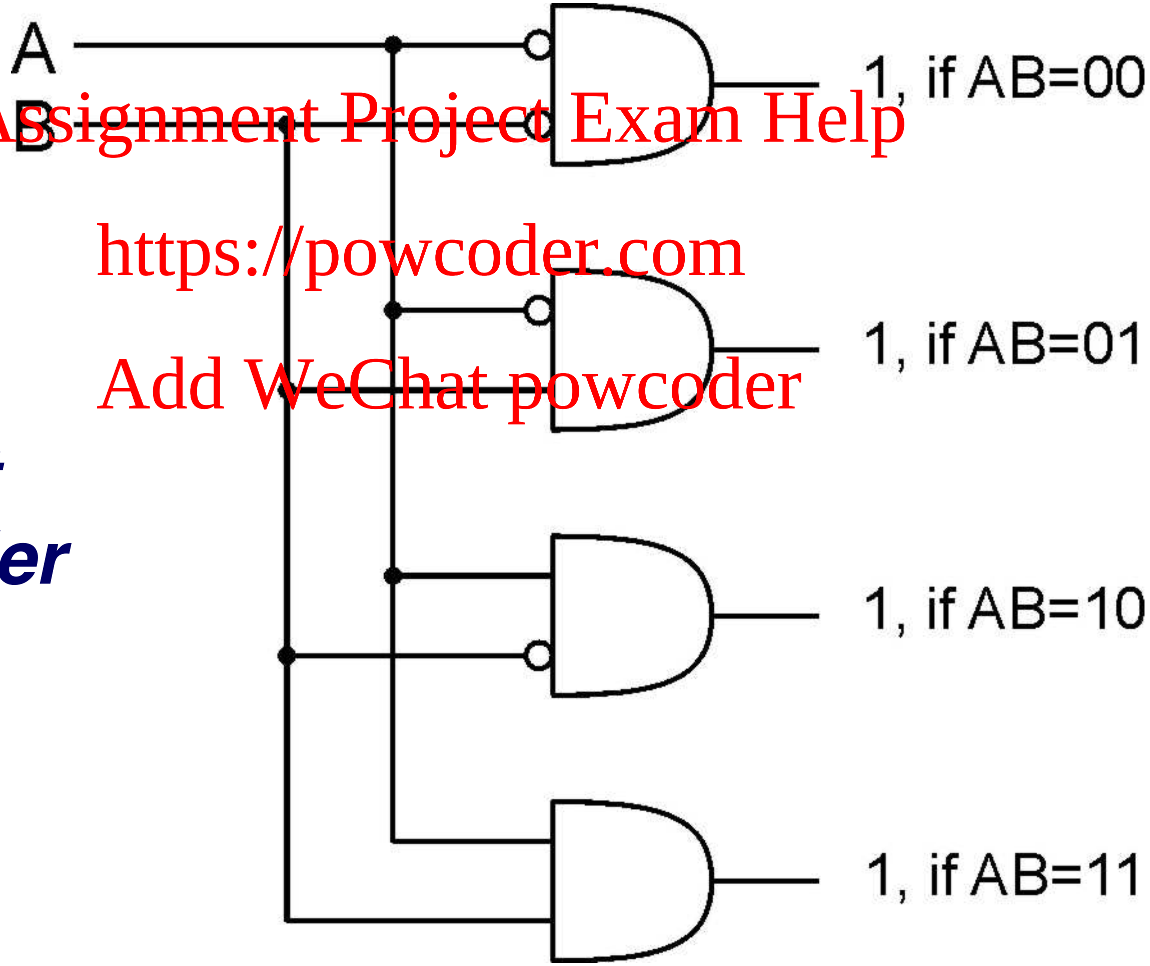

A
B
C
Output = $\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$
$\bar{A}BC$
$A\bar{B}C$
$AB\bar{C}$
ABC


C
Output = AB + BC + AC
AB
BC
AC

Decoder

n inputs, *2ⁿ* outputs

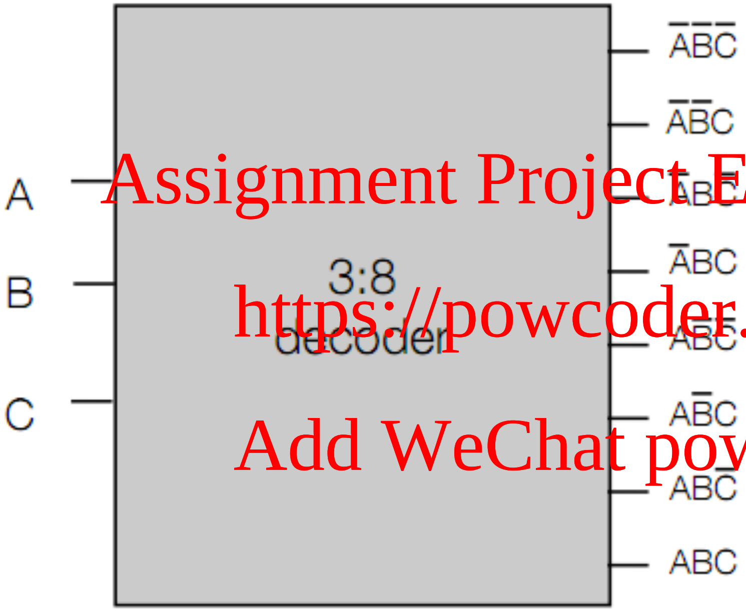
- exactly one output is 1 for each possible input pattern



2-bit decoder

Decoder Circuits

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

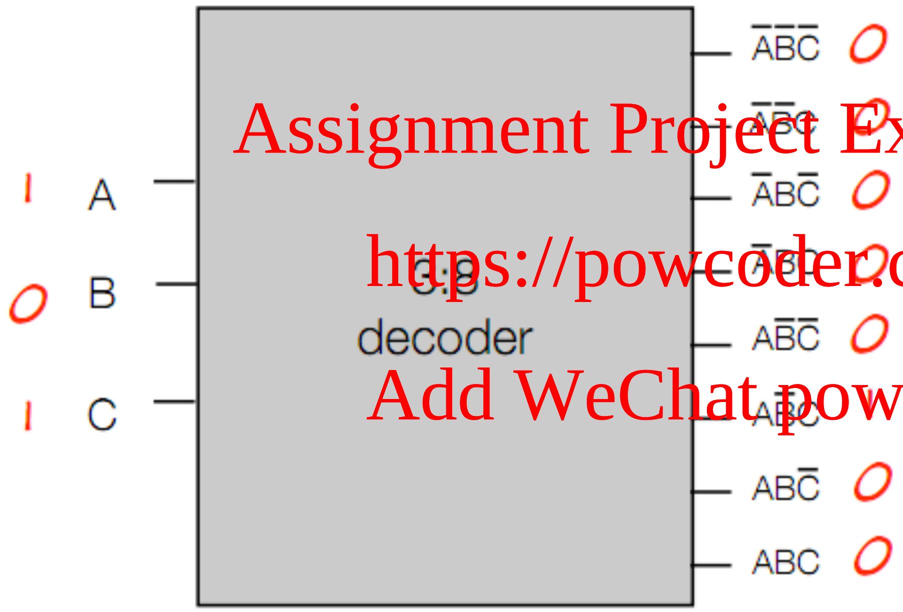

A
B
C
3:8
decoder
$\overline{A}\,\overline{B}\,\overline{C}$
$\overline{A}\,\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$A\overline{B}\,\overline{C}$
$A\overline{B}C$
$AB\overline{C}$
ABC

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

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

Decoder Example

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

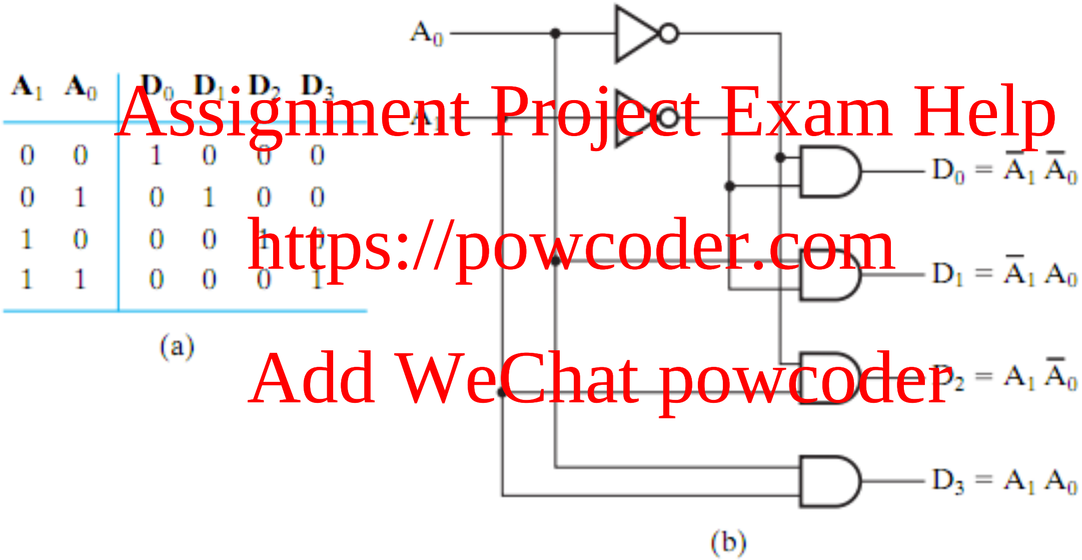

1
A
0
B
1
C
3:8
decoder
$\overline{A}\overline{B}\overline{C}$ 0
$\overline{A}\overline{B}C$ 0
$\overline{A}B\overline{C}$ 0
$\overline{A}BC$ 0
$A\overline{B}\overline{C}$ 0
$A\overline{B}C$
$AB\overline{C}$ 0
ABC 0

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

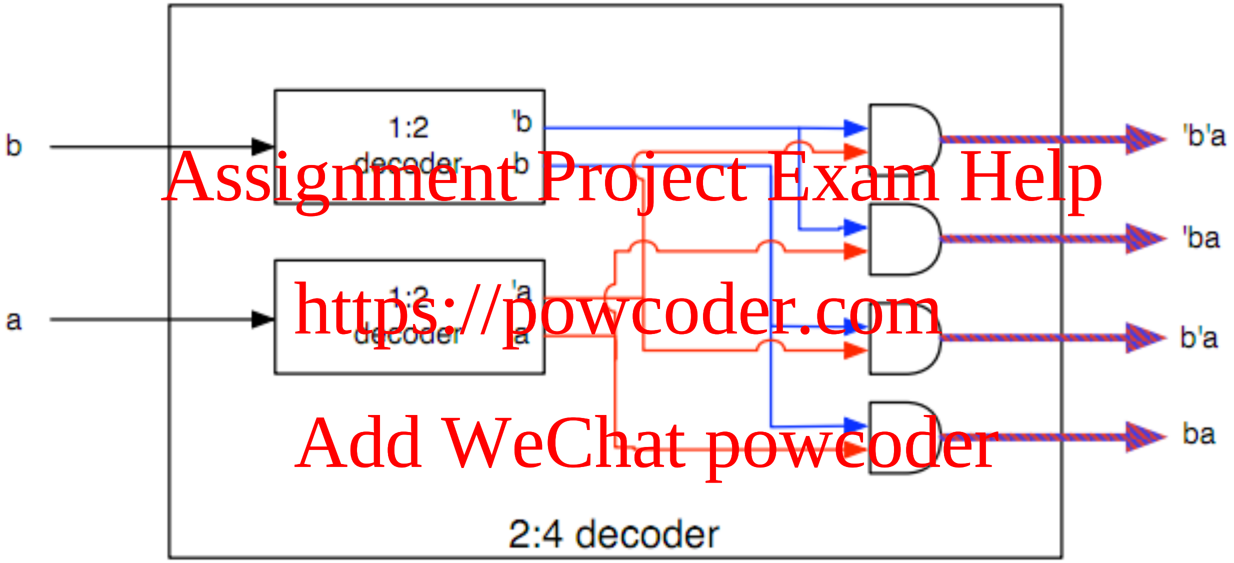
(a)

A_0
A_1
$D_0 = \overline{A}_1 \overline{A}_0$
$D_1 = \overline{A}_1 A_0$
$D_2 = A_1 \overline{A}_0$
$D_3 = A_1 A_0$

(b)

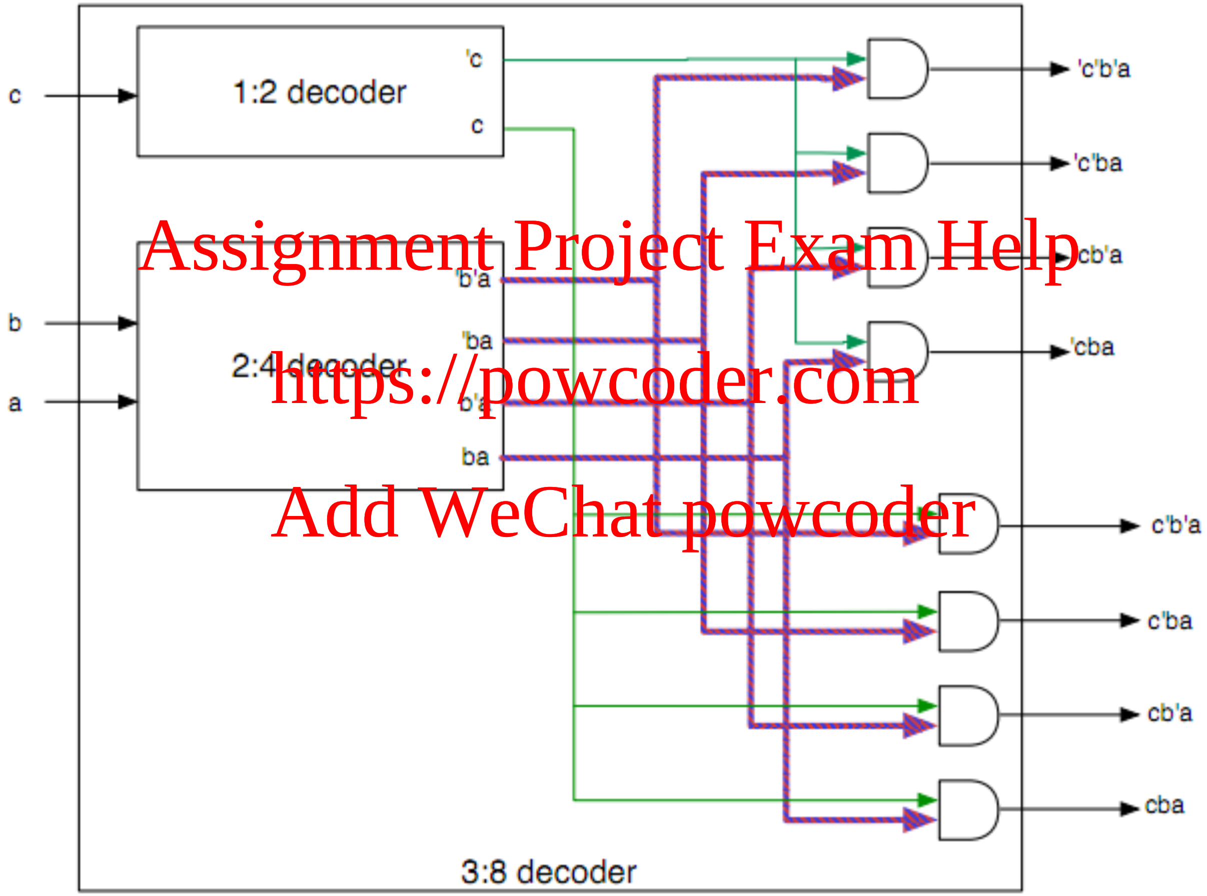
M. Morris Mano & Charles R. Kime
LOGIC AND COMPUTER DESIGN FUNDAMENTALS, 4e

2:4 Decoder from 1:2 Decoders



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

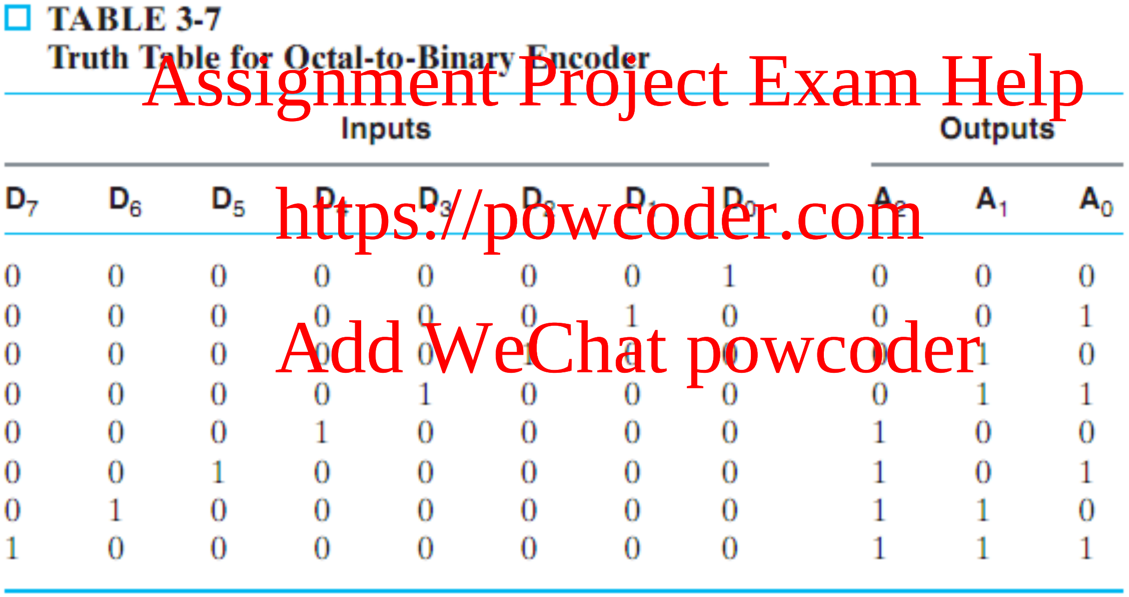
Hierarchical 3:8 Decoder



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

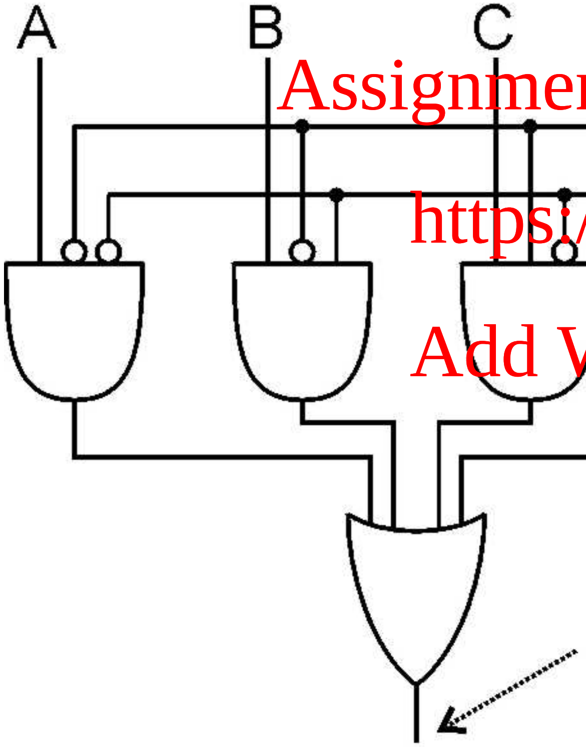
Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

M. Morris Mano & Charles R. Kime
LOGIC AND COMPUTER DESIGN FUNDAMENTALS, 4e

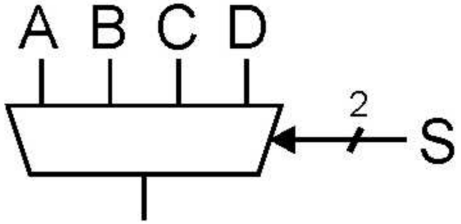
Multiplexer (MUX)

n-bit selector and *2ⁿ* inputs, one output

- output equals one of the inputs, depending on selector



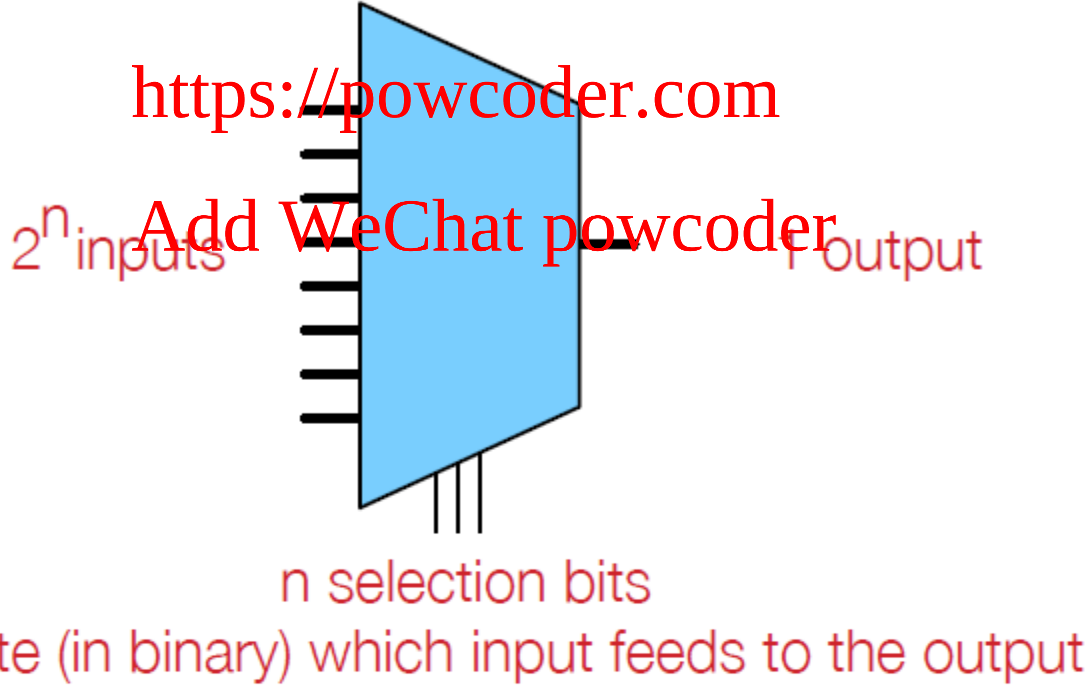
A, if S=00
B, if S=01
C, if S=10
D, if S=11



4-to-1 MUX

Multiplexers (Muxes)

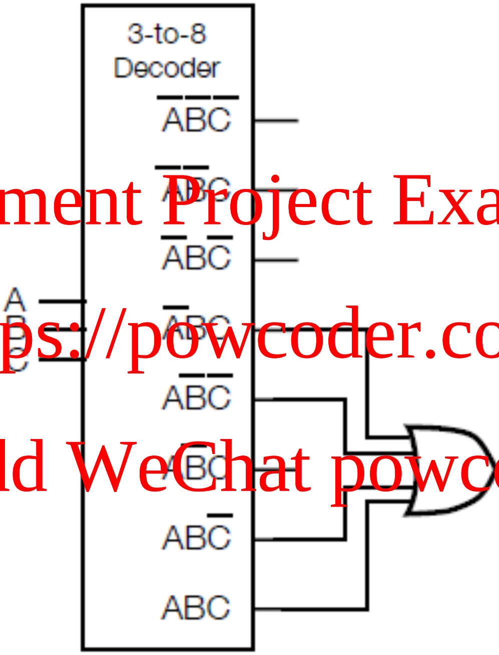
Combinational circuit that selects binary information from many inputs to one output

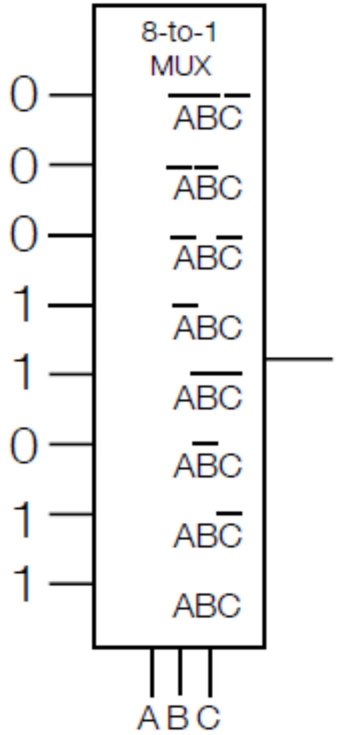


Functions with Decoders or Muxes

- 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


3-to-8
Decoder
$\overline{A}\,\overline{B}\,\overline{C}$
$\overline{A}\,\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$A\overline{B}\,\overline{C}$
$A\overline{B}C$
$AB\overline{C}$
ABC
A
B
C


8-to-1
MUX
0
0
0
1
1
0
1
1
$\overline{A}\,\overline{B}\,\overline{C}$
$\overline{A}\,\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$A\overline{B}\,\overline{C}$
$A\overline{B}C$
$AB\overline{C}$
ABC
A B C

- Decoder: OR minterms for which F should evaluate to 1

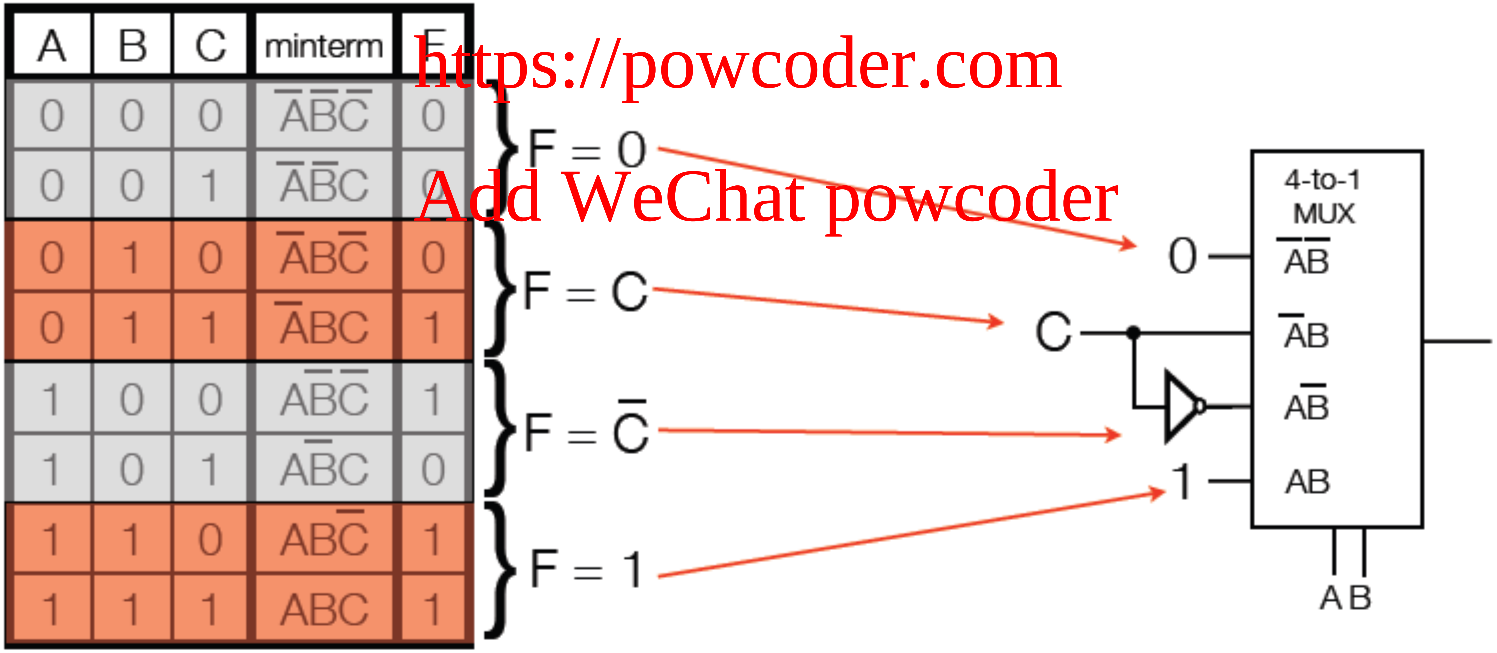
- MUX: Feed in the value of F for each minterm

Can we do it a Smaller Mux?

Can actually use a smaller mux with a trick: $F = A\overline{C} + BC$

Look at the rows below, A & B have the same value, C iterates between 0 & 1

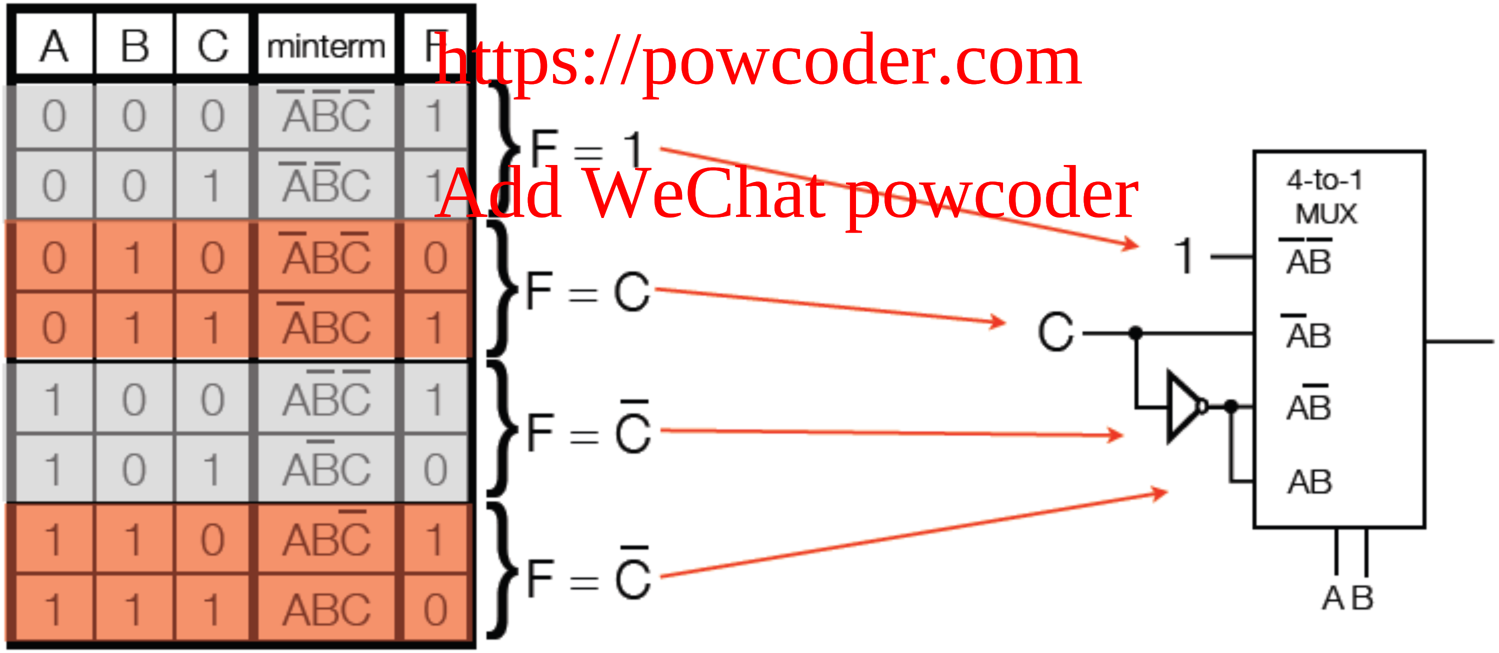
For the pair of rows, F either equals 0 or 1, C or not(C)



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

Another Example

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



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

Where are we?

We have already seen

-- Basic gates: AND, NOT, OR

-- Building blocks: Decoder and Multiplexer

-- Implement circuits from truth tables

-- We know: (a) minterm (b) Sum of products

-- We know basic identities

Implement A+B

With Multiplexers

(1) Using 2:1 mux

(2) Using 4:1 mux

With Decoders

(1) Using a 2:4 decoder

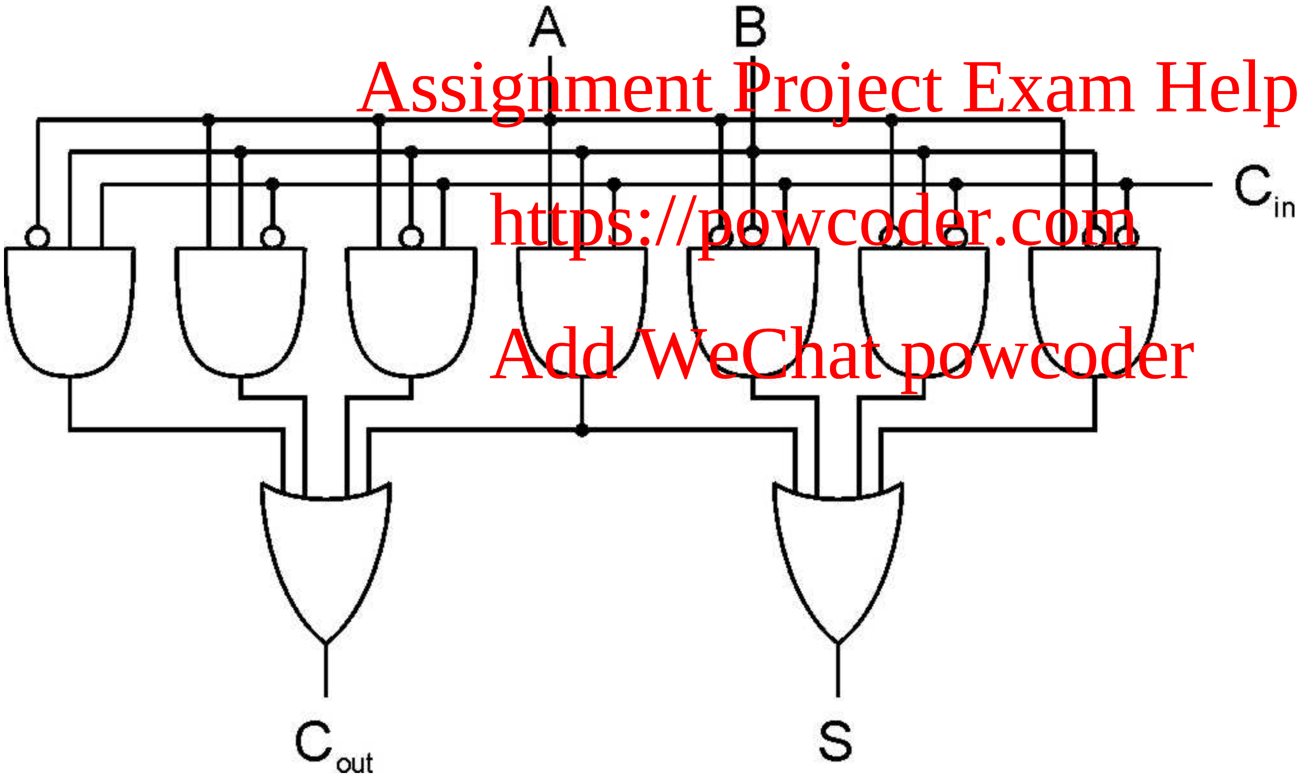
Half Adder

Add two bits and produce a sum and a carry.

How do we go about building the circuit?

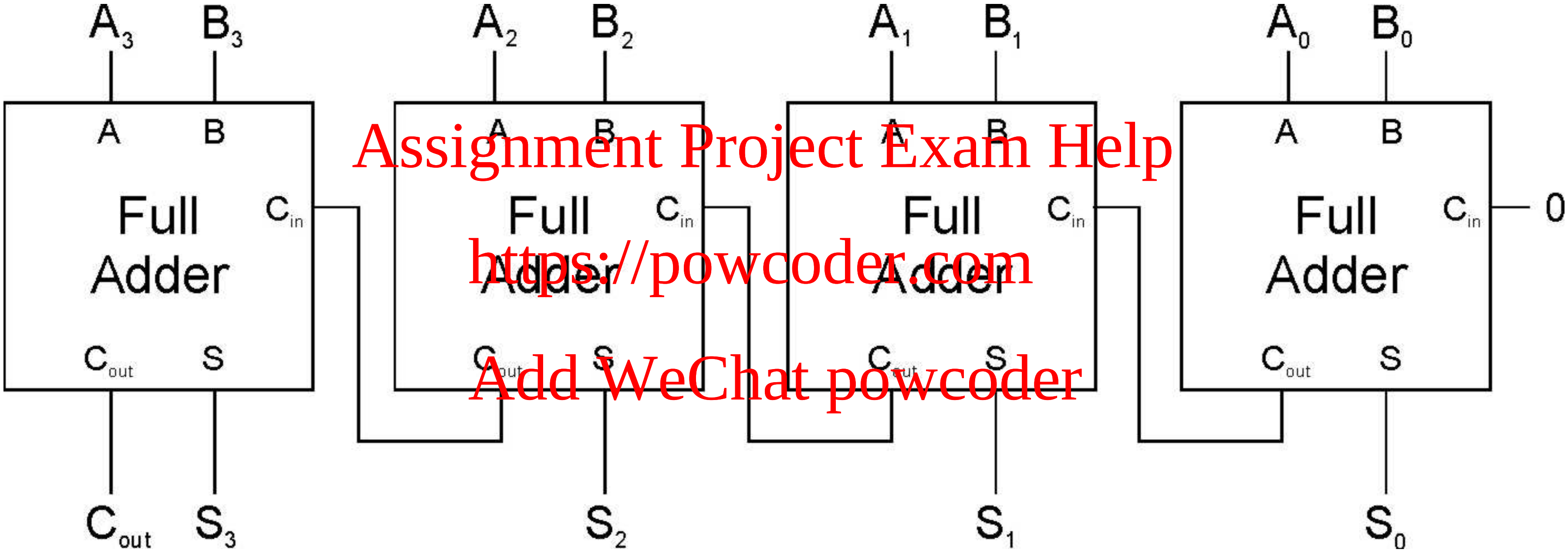
Full Adder

Add two bits and carry-in,
produce one-bit sum and carry-out.



A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Four-bit Adder



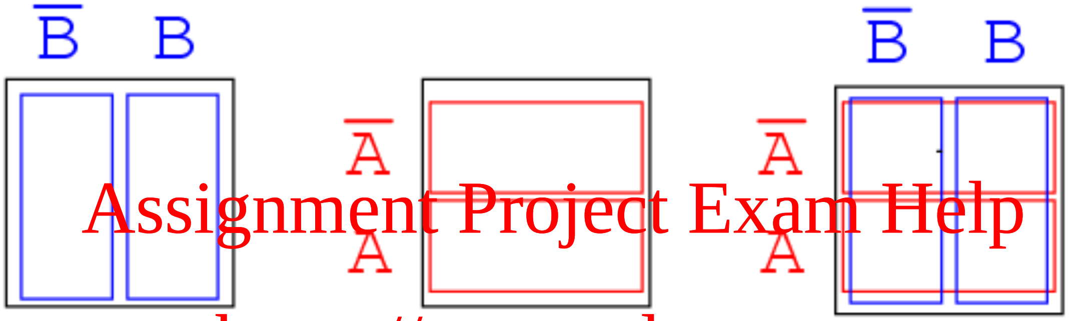
Karnaugh Maps or K-Maps

K-maps are a graphical technique to view minterms and how they relate.

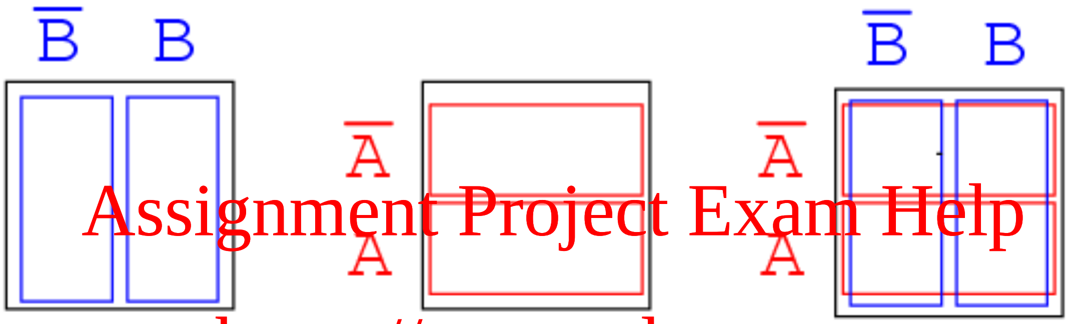
The “map” is a diagram made up of squares, with each square representing a single minterm.

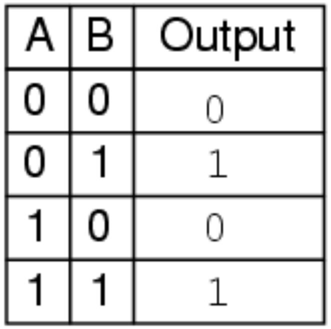
Minterms resulting in a “1” are marked as “1”, all others are marked “0”

2 Variable K-Map

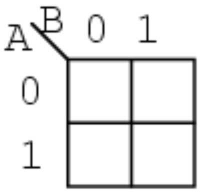


2 Variable K-Map

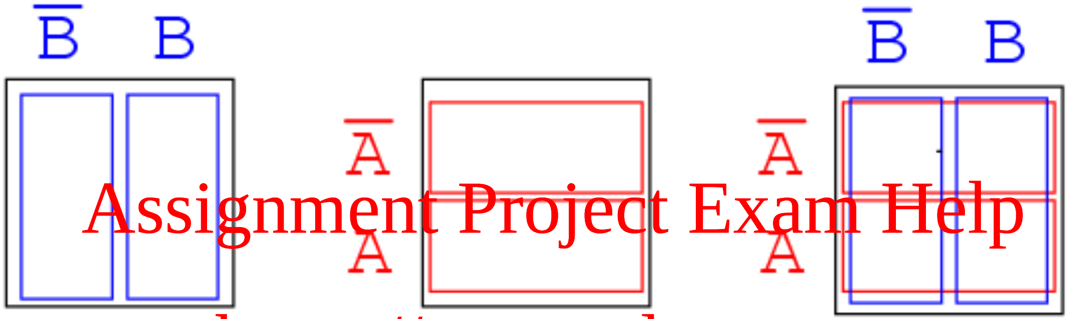




A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1



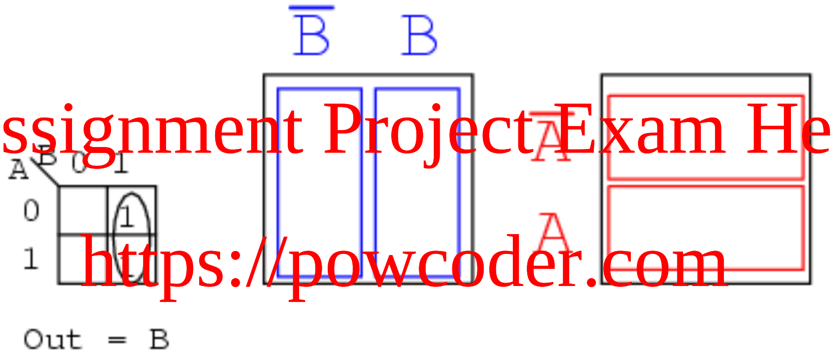
2 Variable K-Map

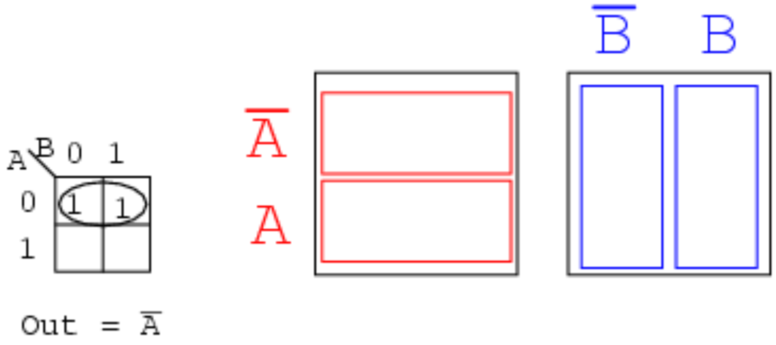


A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1

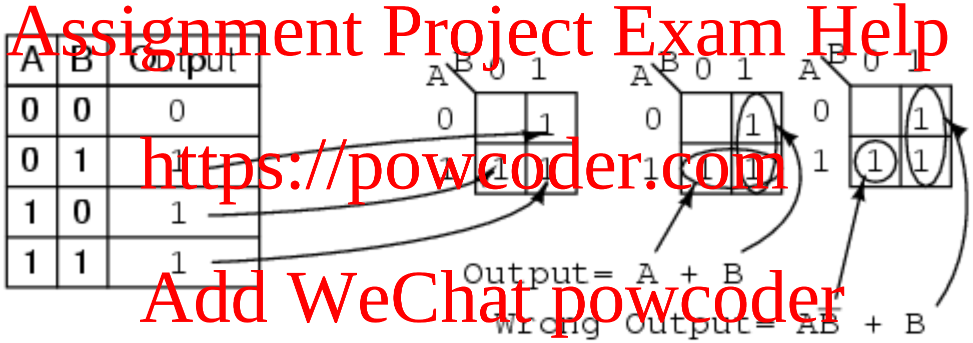
A \ B	0	1
0	0	1
1	0	1

Finding Commonality





Finding the “best” solution

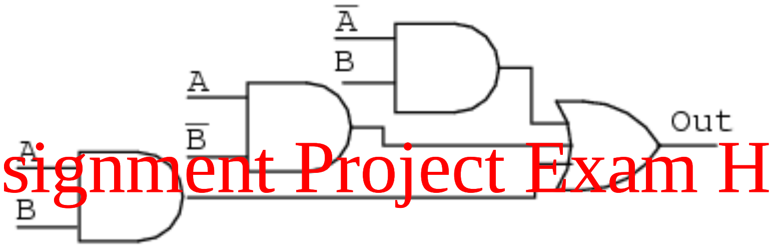


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

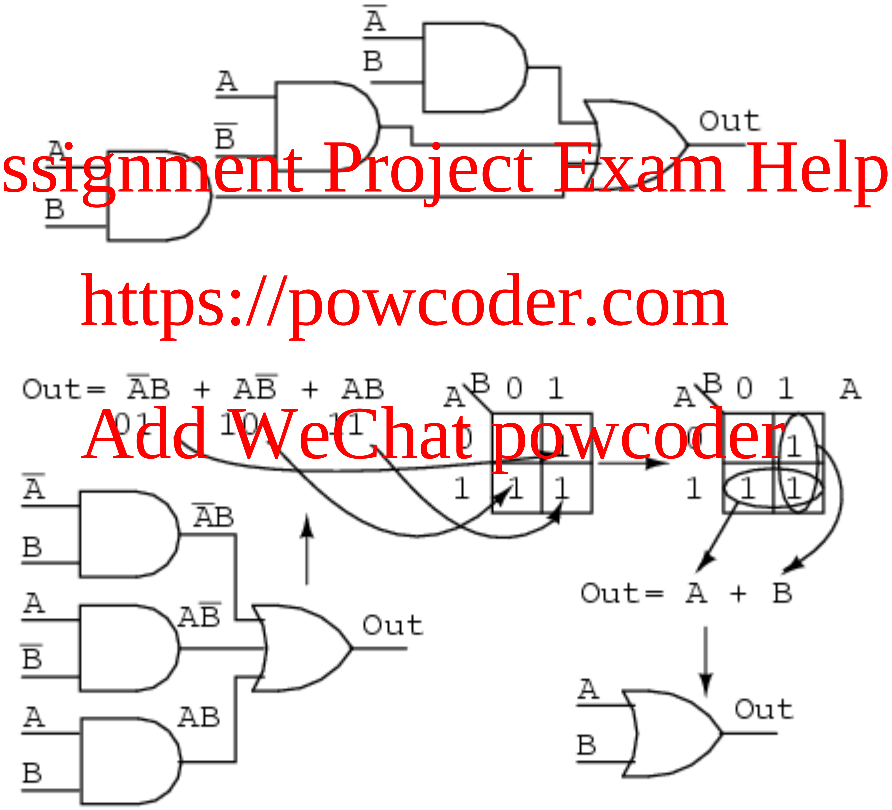
Grouping become simplified products.

Both are “correct”. “A+B” is preferred.

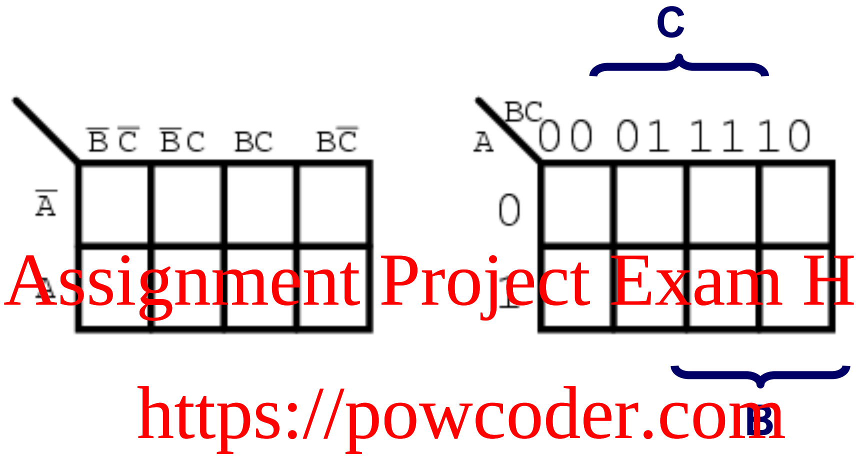
Simplify Example



Simplify Example



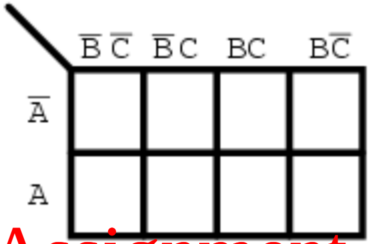
3 Variable K-Maps

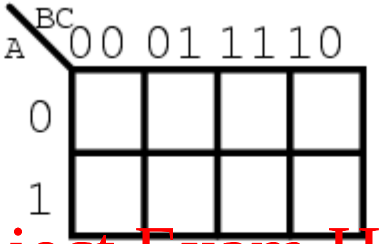


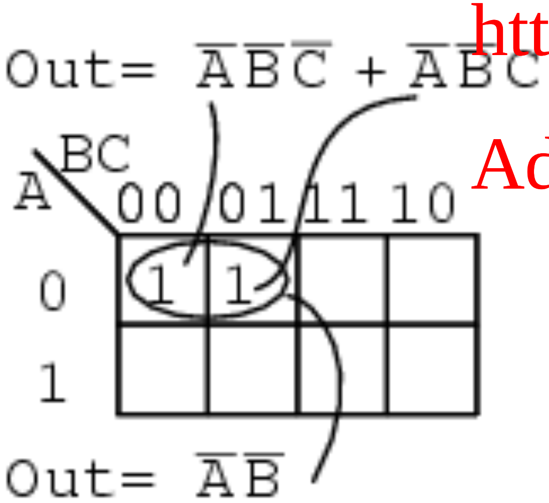
- Note in higher maps, several variables occupy a given axis
- The sequence of 1s and 0s follow a Gray Code Sequence.

- Grey code is a number system where two successive values differ only by 1-bit

3 Variable K-Maps

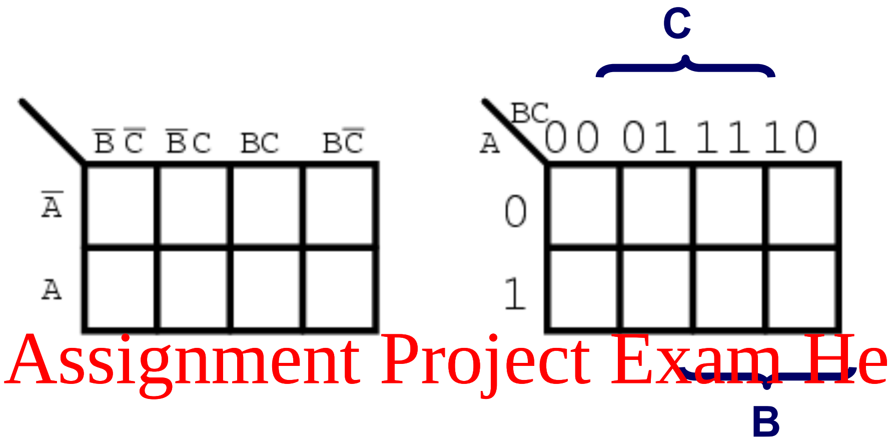




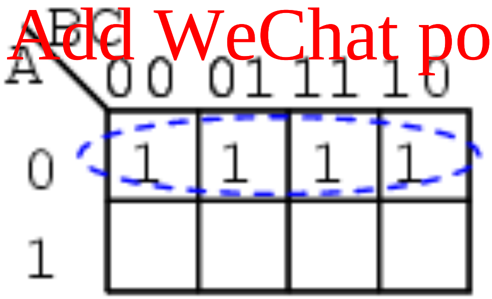

Out= $\bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C$

Out= $\bar{A}\bar{B}$

3 Variable K-Maps

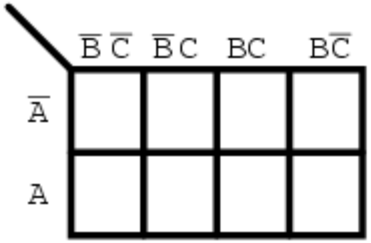


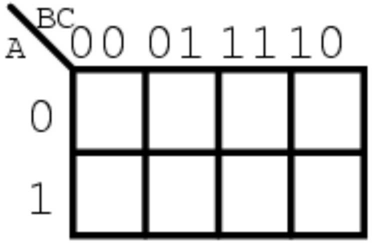
$$Out = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + \bar{A}B\bar{C}$$



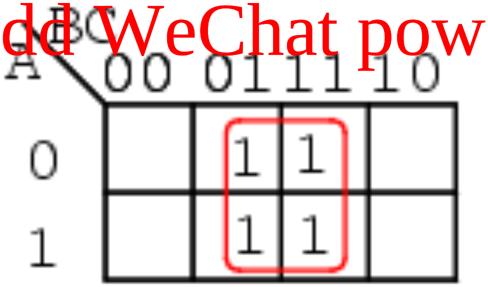
$$Out = \bar{A}$$

3 Variable K-Maps





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



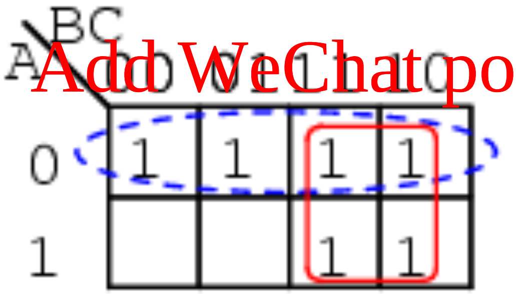
Out= C

3 Variable K-Maps

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

A \ BC	00	01	11	10
0				
1				

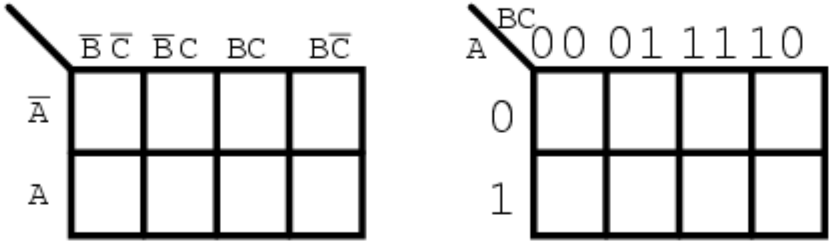
$$Out= \overline{A}\,\overline{B}\,\overline{C}+\overline{A}\,\overline{B}C+\overline{A}BC+\overline{A}B\overline{C}+ABC+AB\overline{C}$$



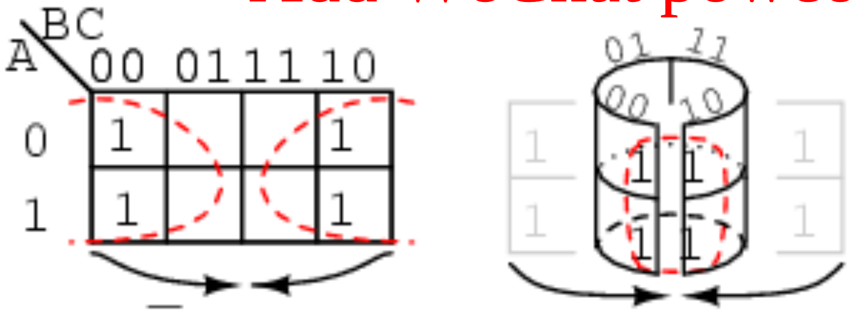
A \ BC	00	01	11	10
0	1	1	1	1
1			1	1

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

3 Variable K-Maps

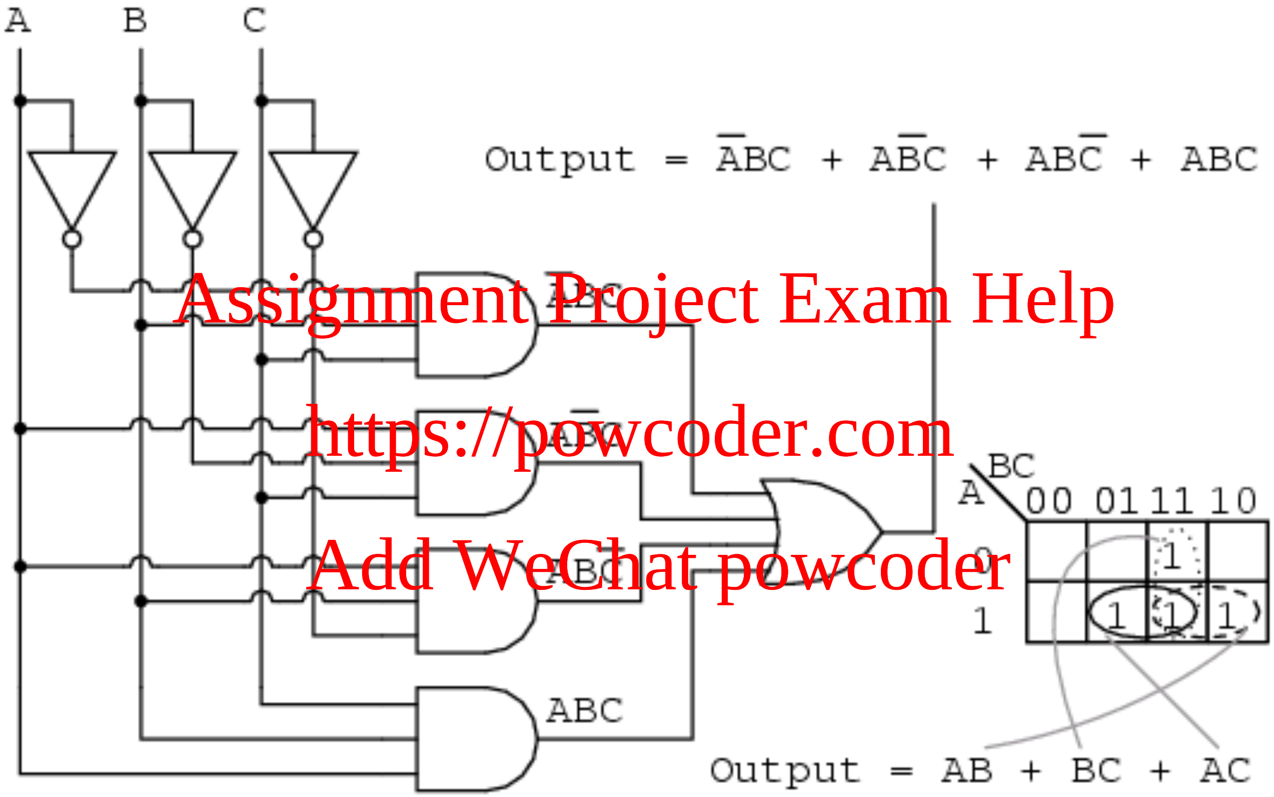


$$Out = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + \bar{A}B\bar{C} + AB\bar{C}$$



$$Out = \bar{C}$$

Back to our earlier example…..



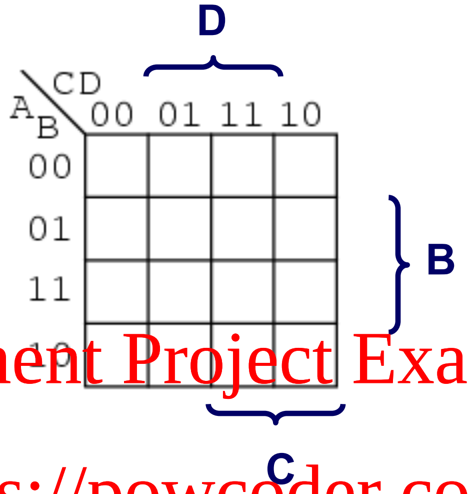
Output = $\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

A \ BC	00	01	11	10
0			1	
1		1	1	1

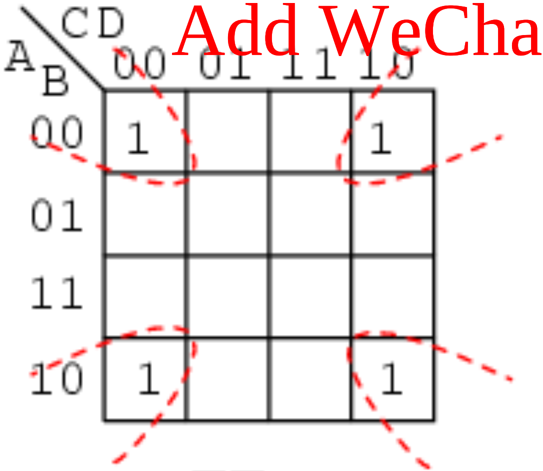
Output = AB + BC + AC

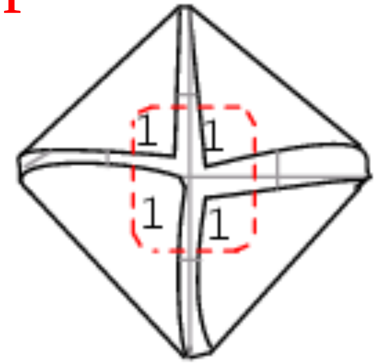
The K-map and the algebraic produce the same result.

Up… up… and let's keep going



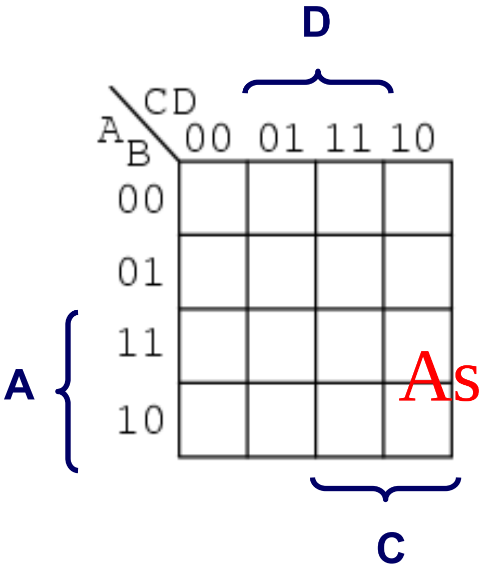
$$\text{Out} = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}C\overline{D} + A\overline{B}C\overline{D}$$



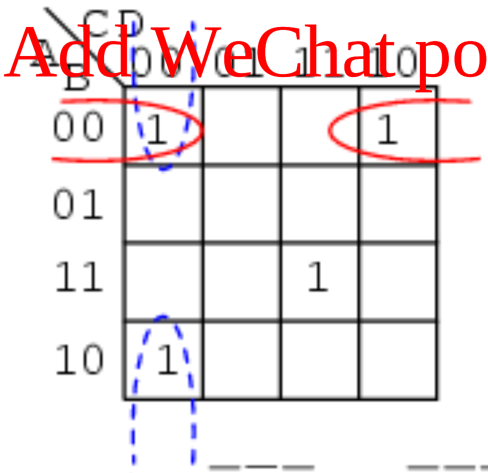


$$\text{Out} = \overline{B}\,\overline{D}$$

Few more examples



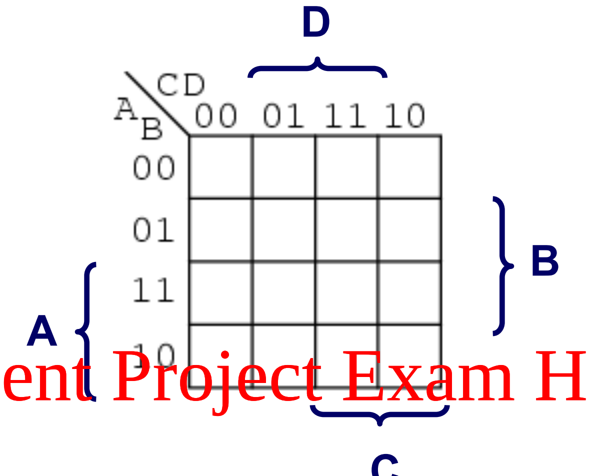
Out= $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}C\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + ABCD$



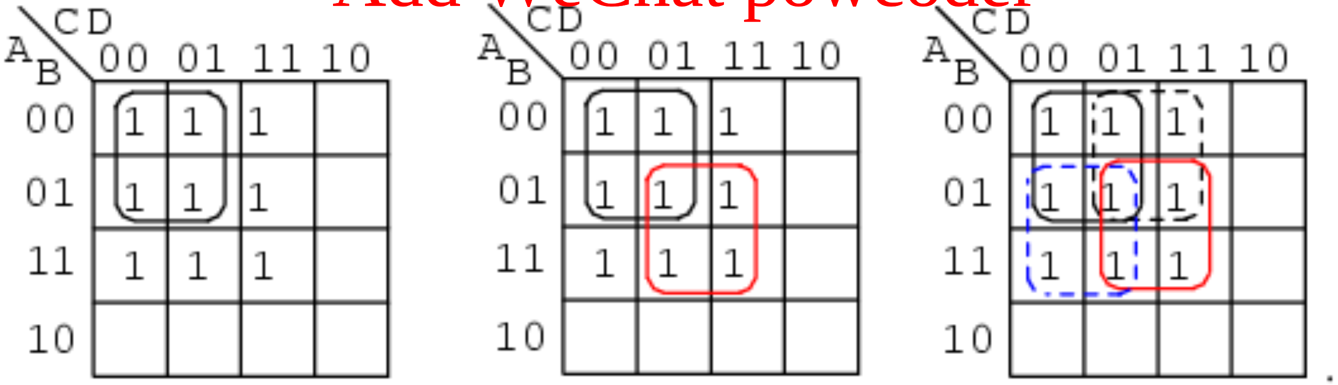
AB \ CD	00	01	11	10
00	1			1
01				
11			1	
10	1			

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

Few more examples



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



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

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

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

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

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

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

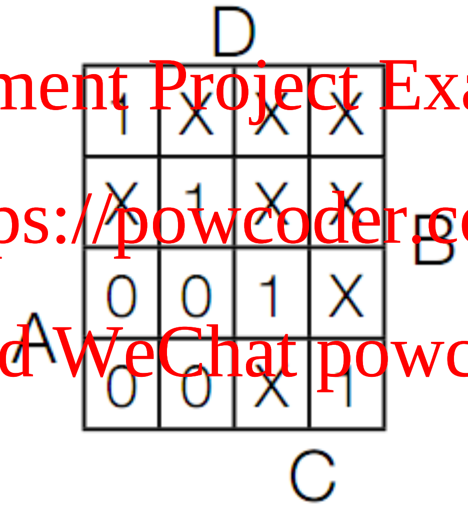
Inputs will not occur

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

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

Don’t Cares can Greatly Simplify Circuits

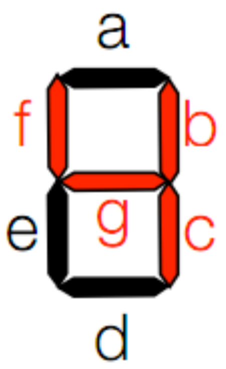
Sometimes “don’t cares” greatly simplify circuitry



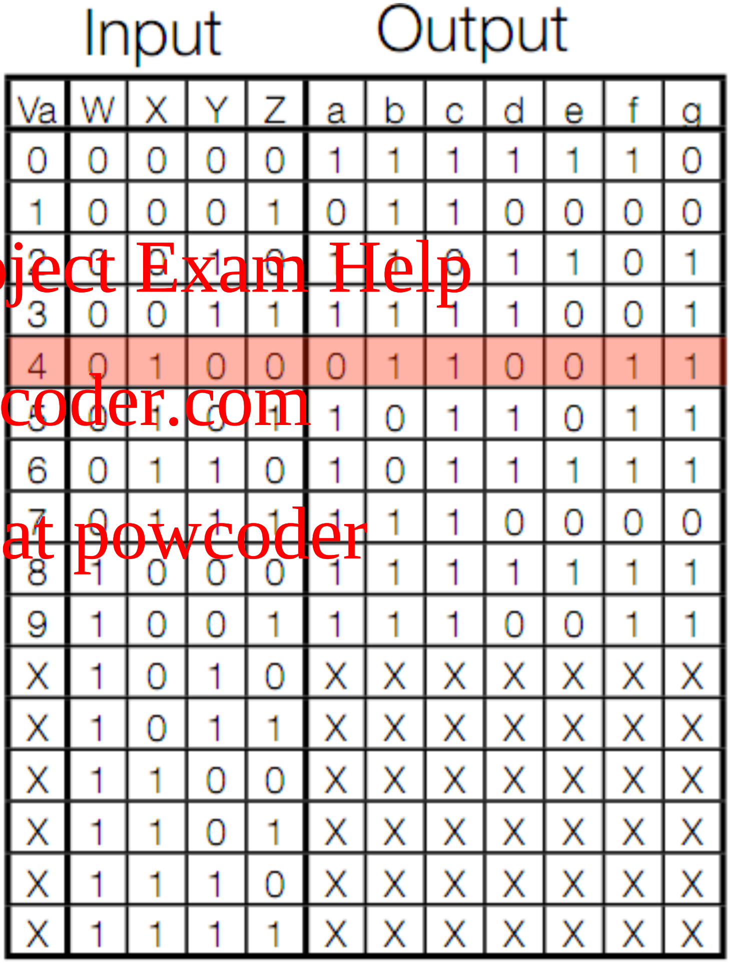
1	X	X	X
X	1	X	X
0	0	1	X
0	0	X	1

$$\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}B\bar{C}D + ABCD + A\bar{B}C\bar{D} \text{ vs. } \bar{A} + C$$

Design Example

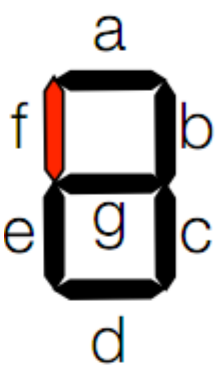


e.g., what outputs "lights up" when input V=4?

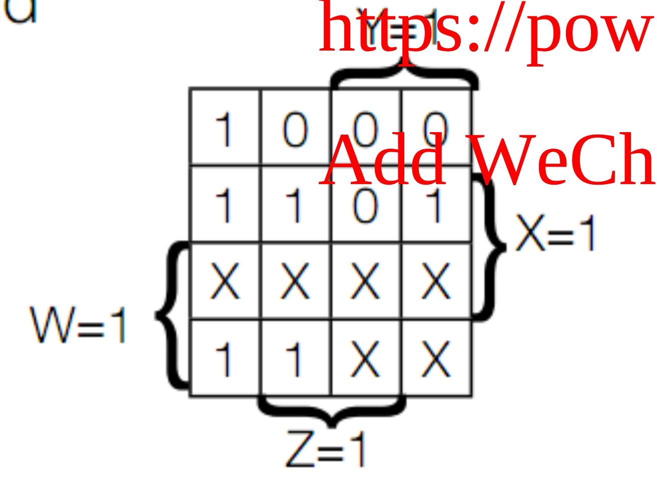


Input					Output						
Va	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
X	1	0	1	0	X	X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

Design Example



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

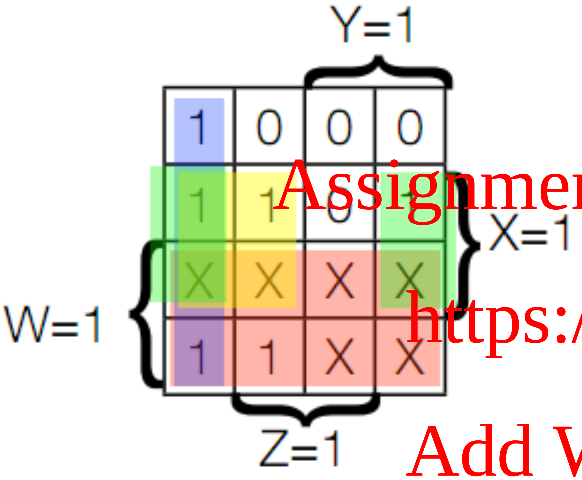


1	0	0	0
1	1	0	1
X	X	X	X
1	1	X	X

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

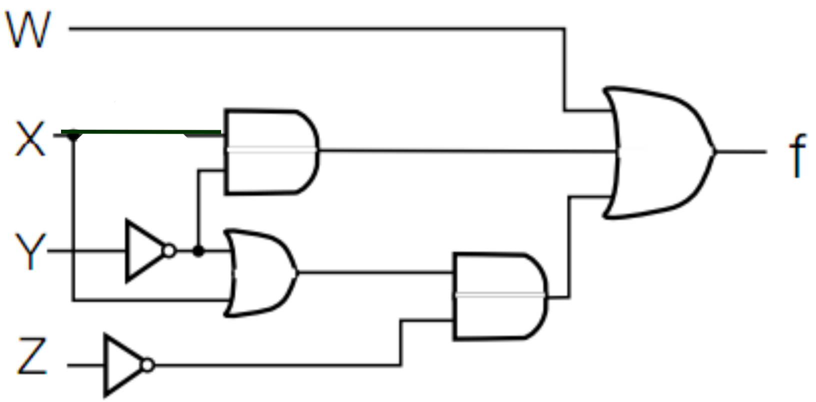
Design Example

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



1	0	0	0
1	1	0	1
X	X	X	X
1	1	X	X

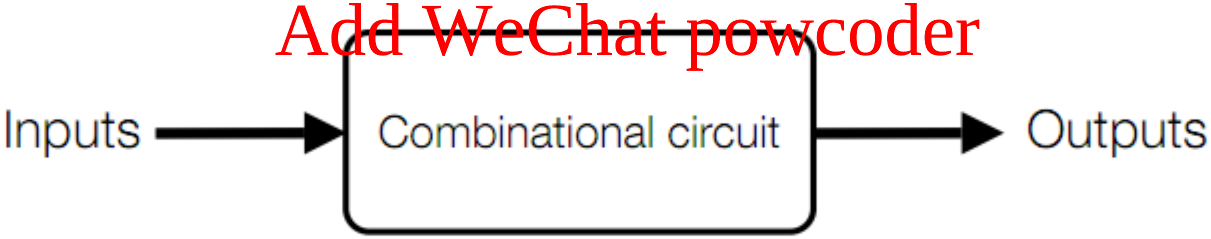
$f = W + \bar{Y}\bar{Z} + X\bar{Z}$ **$+X\bar{Y}$** $= W + (X+\bar{Y})\bar{Z}$ **$+\ X\,\bar{Y}$**



Combinational Circuits

Stateless circuits

Outputs are function of inputs only



Time and State

SEQUENTIAL CIRCUITS

How are Sequential Circuits different from Combinational Circuits?

Outputs of sequential logic depend on both current and prior values – it has memory

Definitions:

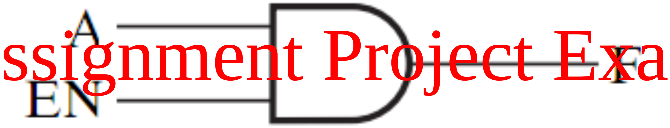
State: all the information about a circuit to explain its future behavior

Latches and flip-flops: state elements that store one bit of state

Synchronous sequential elements: combinational logic followed by a bank of flip-flops

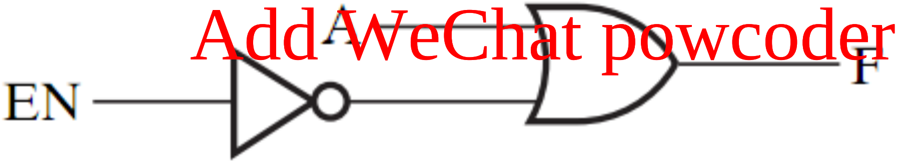
Enabler Circuits

Output is "enabled" (F=A) only when input 'ENABLE' signal is asserted (EN=1)



(a)

EN	F
0	0
1	A



(b)

EN	F
0	1
1	A

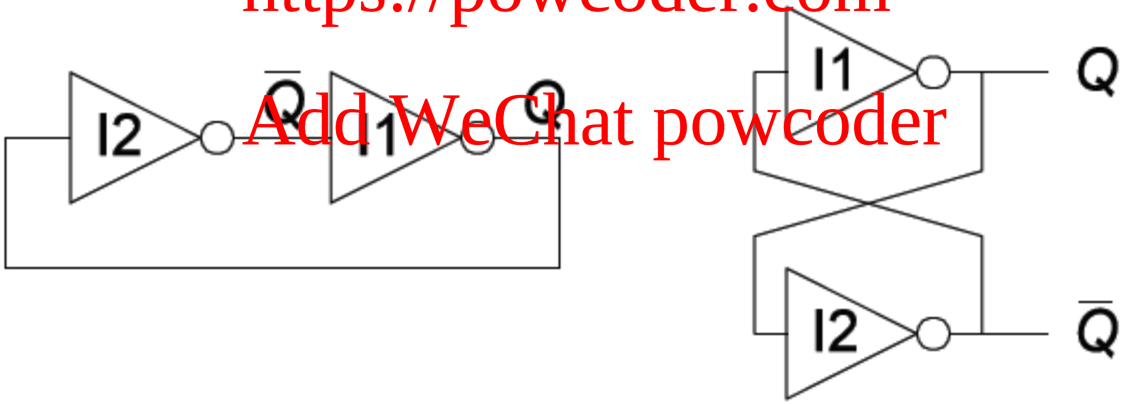
M. Morris Mano & Charles R. Kime
LOGIC AND COMPUTER DESIGN FUNDAMENTALS, 4e

Bistable Circuits

Fundamental building blocks of other elements

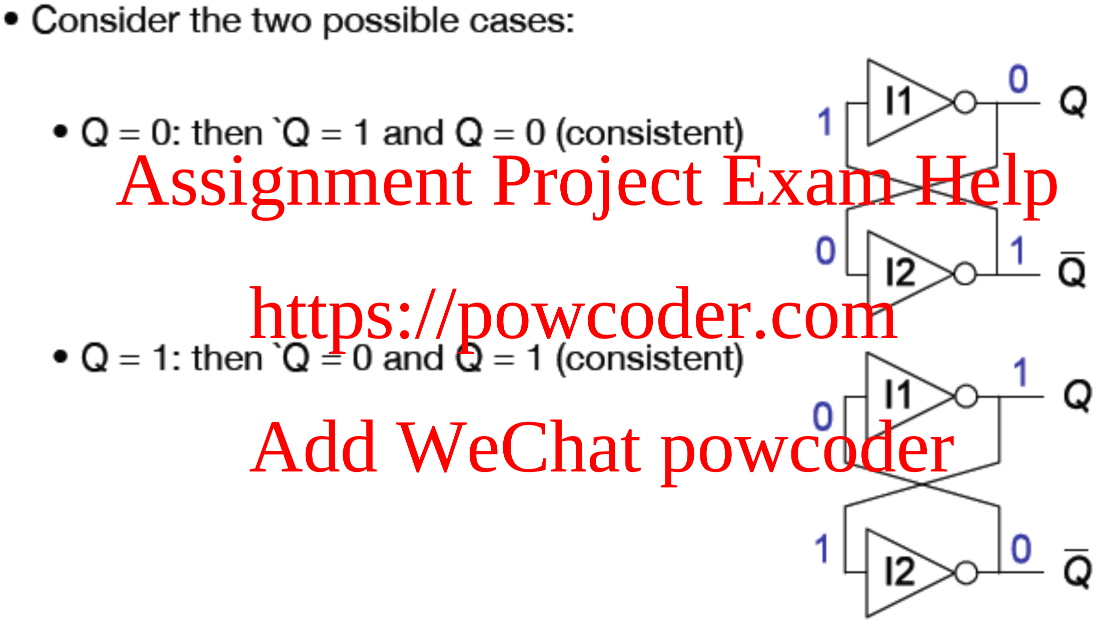
No inputs

Two outputs (Q and Q')



Bistable Circuit Analysis

Consider all the cases

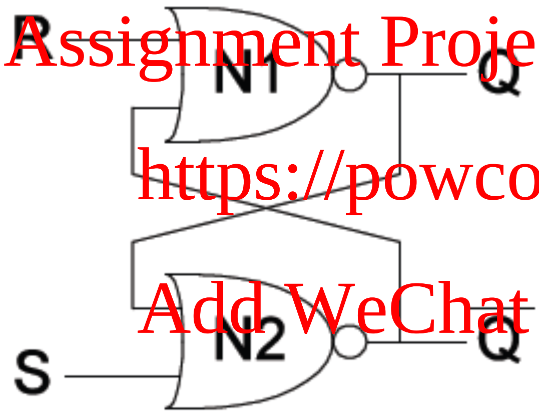


- Consider the two possible cases:
  - Q = 0: then `Q = 1 and Q = 0 (consistent)
  - Q = 1: then `Q = 0 and Q = 1 (consistent)

Bistable circuit stores 1 bit of state (Q, or Q')

But there are no inputs to control state

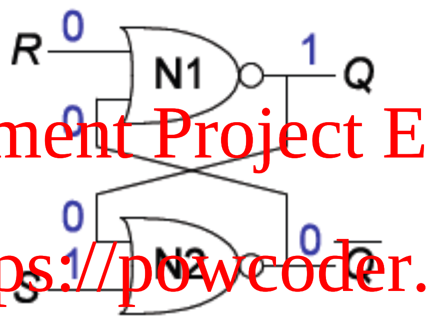
Set/Reset Latch



- S = 1, R = 0
- S = 0, R = 1
- S = 0, R = 0
- S = 1, R = 1

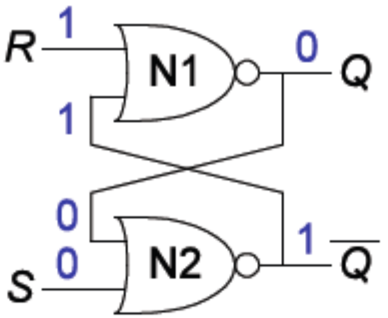
S/R Latch Analysis

- S = 1, R = 0: then Q = 1



set

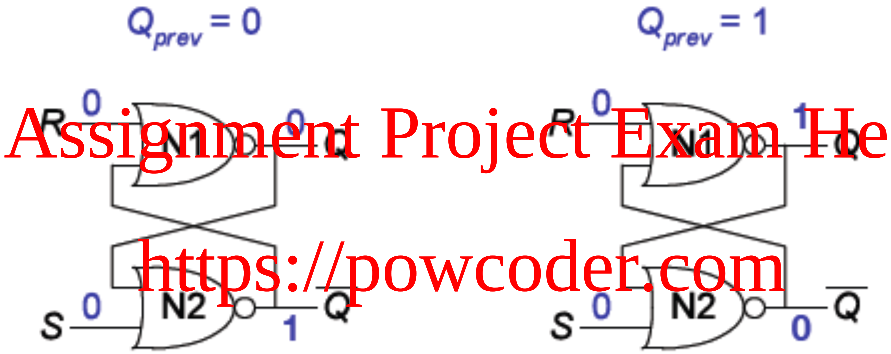
- S = 0, R = 1: then Q = 0



reset

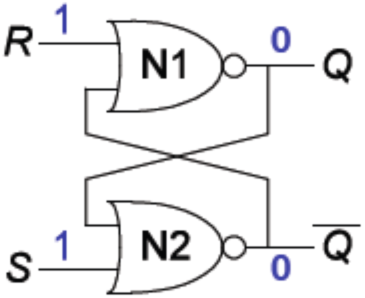
S/R Latch Analysis

- S = 0, R = 0: then Q = Q_{prev}



memory!

- S = 1, R = 1: then Q = 0 and Q = 0



Q=`Q
Invalid state

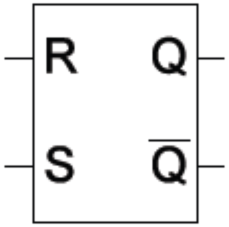
S/R Latch Symbol

Set operation – makes output 1 (S = 1, R = 0, Q = 1)

Reset operation – makes output 0 (S = 0, R = 1, Q = 0)

What about invalid state? (S = 1, R = 1)

SR Latch Symbol



D Latch

Two inputs (C and D)

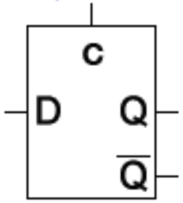
C: controls when the output changes

D (data input): controls what the output changes to

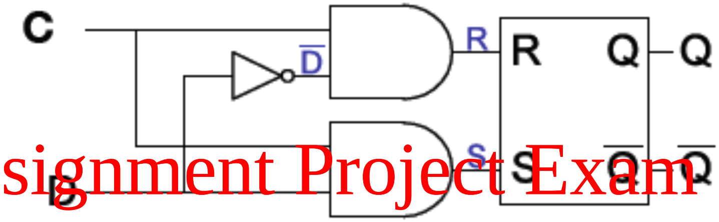
When C = 1, D passes through to Q (transparent latch)

When C = 0, Q holds previous value (opaque latch)

D Latch Symbol



D Latch Internal Circuit



C	D	`D	S	R	Q	`Q
0	X					
1	0					
1	1					

How to Coordinate with Multiple Components?

But how do we coordinate computations and the changing of state values across lots of different parts of a circuit?

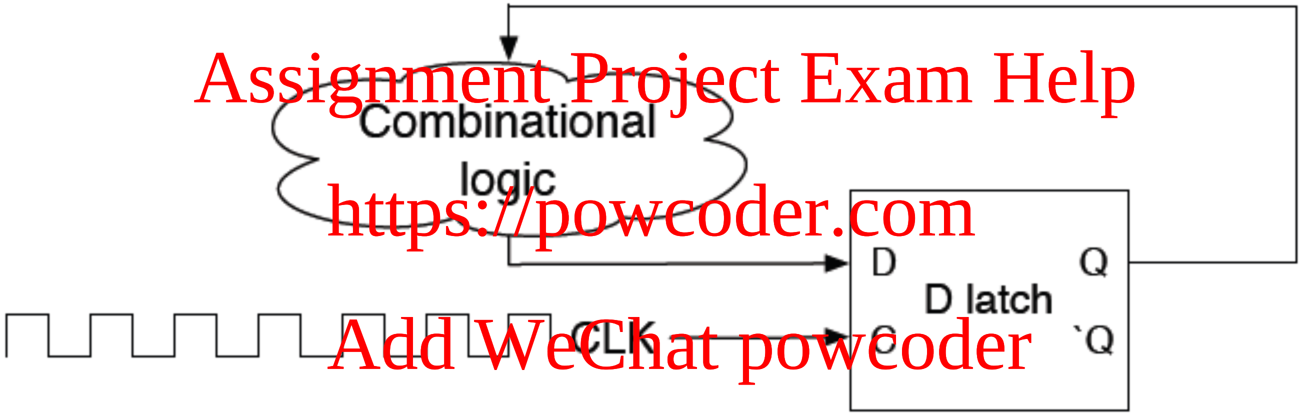
We use CLOCKING (eg. 2.6GHz clock on Intel processors)

On each clock pulse, combinational computations are performed, and results stored in latches

How to introduce clocks into latches?

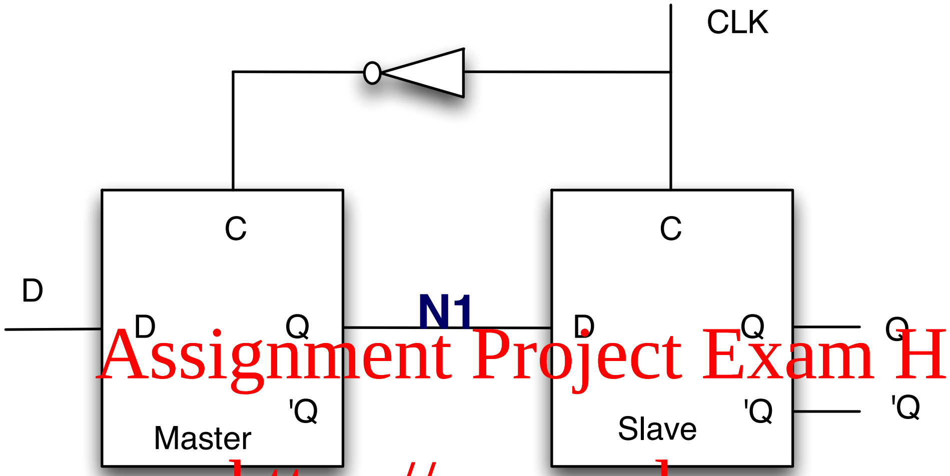
Flip-flops: Latches on a Clock

A straightforward latch is not safely synchronous (or predictably synchronous)



Flip-flops designed so that outputs will NOT change within a single clock pulse

D Flip-Flop



When CLK is 0

- master is enabled (N1 obtains the value input to the master)
- slave is disabled (Old output is still output)

When CLK is 1

- then master is disabled (N1 is the old value)
- Slave is enabled, it copies N1 into output

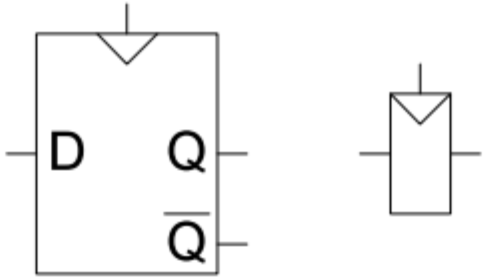
D Flip-Flop Summary

Two inputs: Clk, D

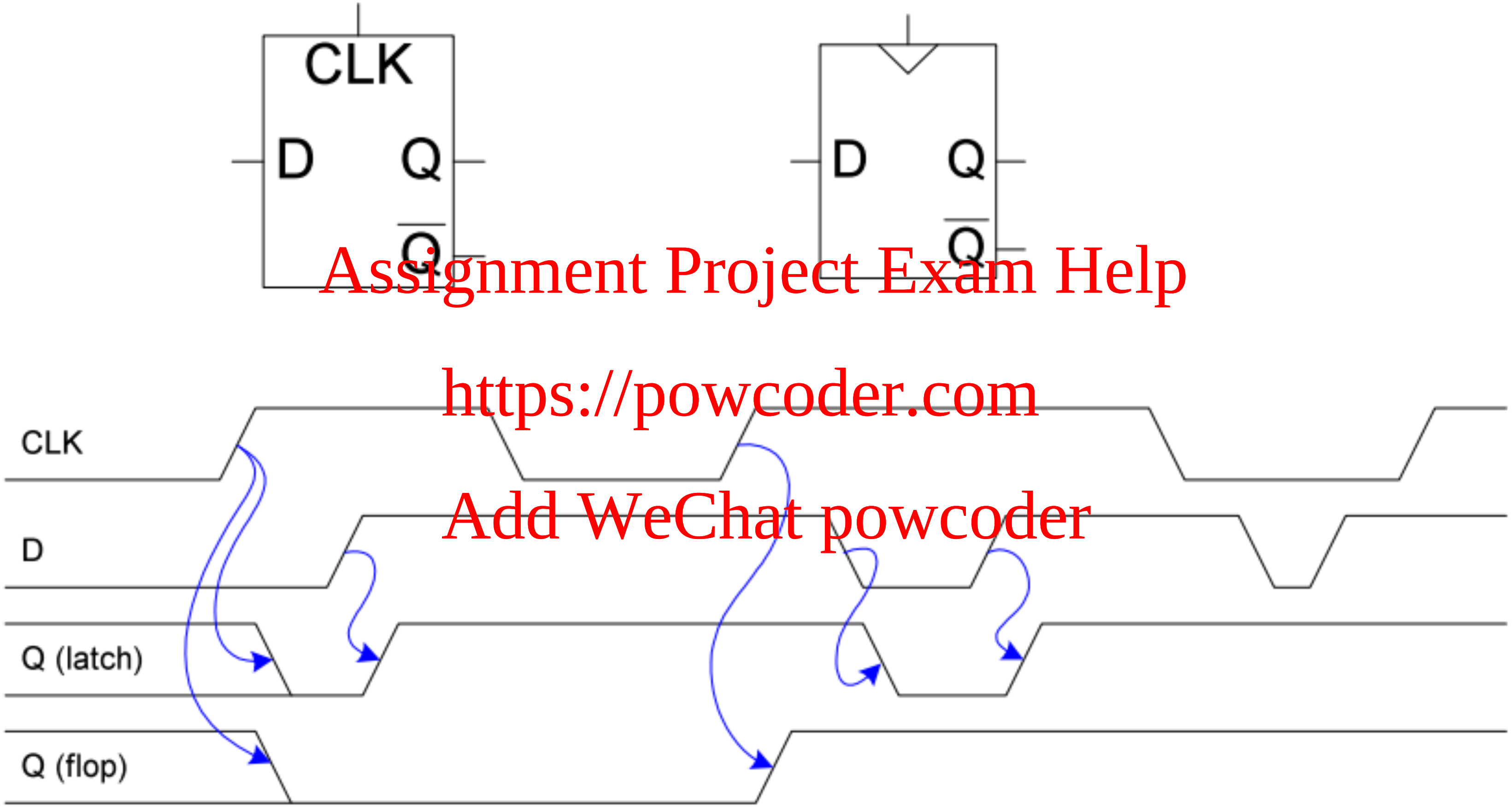
Function

- The flip-flop samples D on rising clock edge
- When clock goes from 0 to 1, D passes through Q
- Otherwise, Q holds its value
- Q only changes on rising clock edge
- Flip-flop is called “edge-triggered” because it is activated only on the clock edge

D Flip-Flop Symbols

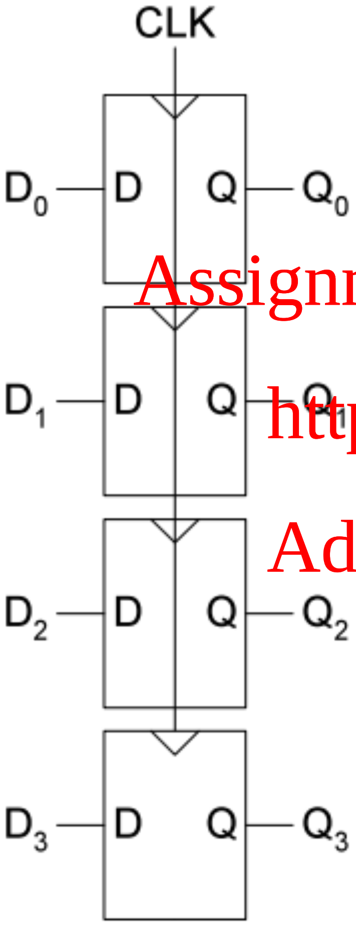


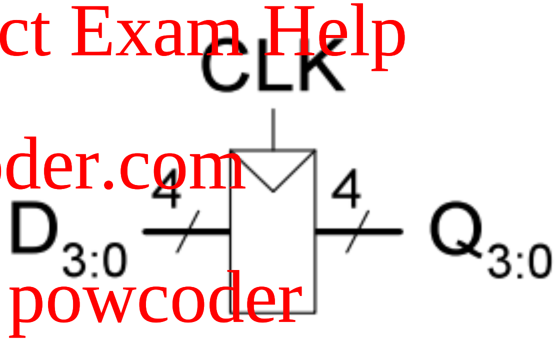
Flip-Flop versus Latch



Latch outputs change at any time, flip-flops only during clock transitions

Registers



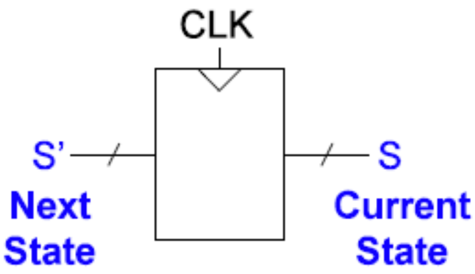


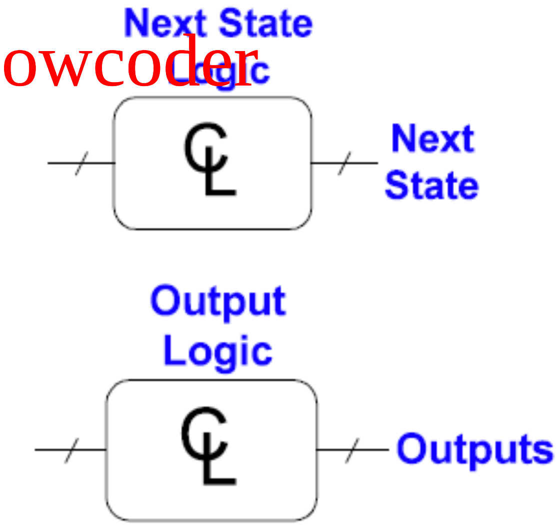
Finite State Machines

FSM = State register + combinational logic

Stores the next state and loads the next state at clock edge

Computes the next state and computes the outputs

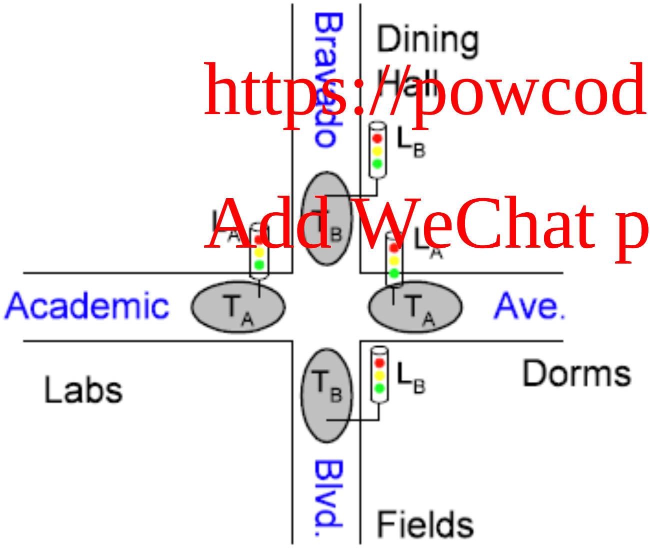


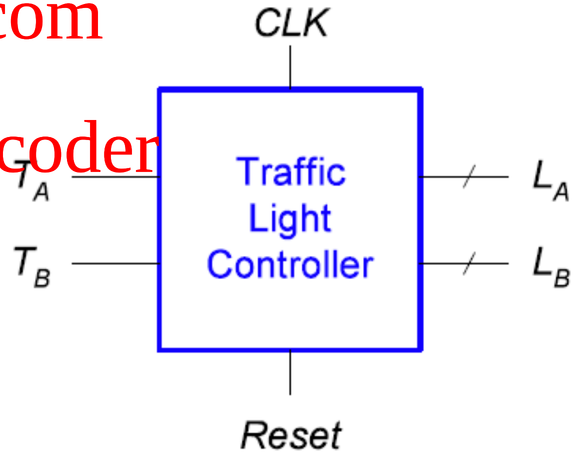


Traffic Light Controller Example

- Traffic sensors: TA, TB (TRUE when there is traffic)

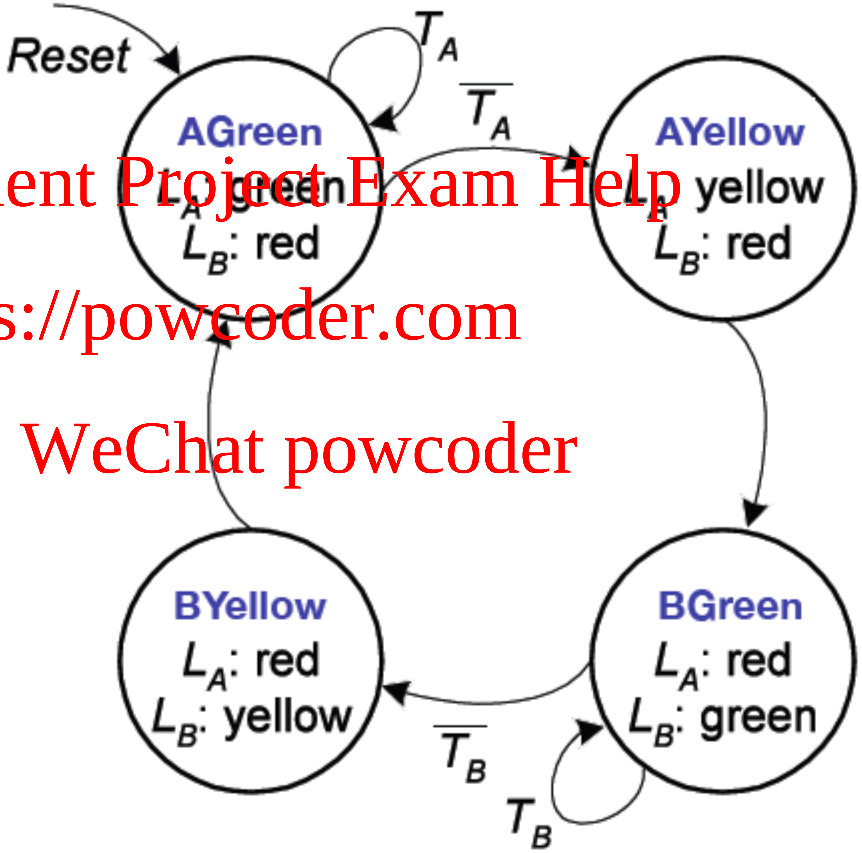
- Lights: LA, LB





FSM State Transition Diagram

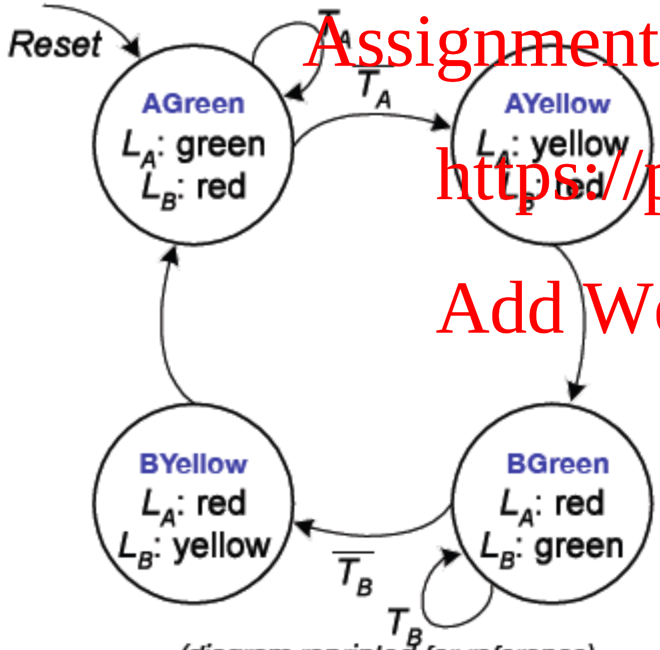
- States: Circles
- Transitions: Arcs



FSM State Transition Table

State transitions from diagram can be rewritten in a state transition table

(S = current state, S' = next state)



(diagram reprinted for reference)

Current State	Inputs		Next State
S	TA	TB	S'
AGreen	0	X	AYellow
AGreen	1	X	AGreen
AYellow	X	X	BGreen
BGreen	X	0	BYellow
BGreen	X	1	BGreen
BYellow	X	X	AGreen

Encoded State Transition Table

After selecting a state encoding, the symbolic states in the transition table can be realized with current state/next state bits

	Encoding	
State	S1	S0
AGreen	0	0
AYellow	0	1
BGreen	1	0
BYellow	1	1

Current State	Encoded Current State		Inputs		Next State	Encoded Next State	
S	S1	S0	TA	TB	S'	S1'	S0'
AGreen	0	0	0	X	AYellow	0	1
AGreen	0	0	1	X	AGreen	0	0
AYellow	0	1	X	X	BGreen	1	0
BGreen	1	0	X	0	BYellow	1	1
BGreen	1	0	X	1	BGreen	1	0
BYellow	1	1	X	X	AGreen	0	0

Computing Next State Logic

Current State	Encoded Current State		Inputs		Next State	Encoded Next State	
S	S1	S0	TA	TB	S'	S1'	S0'
AGreen	0	0	0	X	AYellow	0	1
AGreen	0	0	1	X	AGreen	0	0
AYellow	0	1	X	X	BGreen	1	0
BGreen	1	0	X	0	BYellow	1	1
BGreen	1	0	X	1	BGreen	1	0
BYellow	1	1	X	X	AGreen	0	0

From K-maps, figure out expressions for the next state:

$$S_1' = S_1 \oplus S_0$$

$$S_0' = \overline{S_1}\,\overline{S_0}\,\overline{T_A} + S_1\,\overline{S_0}\,\overline{T_B}$$

FSM Output Table

FSM output logic is computed in similar manner as next state logic

In this system, output is a function of current state (Moore machine)

Alternative – Mealy machine (output function of both current state and inputs, though we won't cover this in class)

output encoding

Output	Encoding	
Green	0	0
Yellow	0	1
Red	1	0

output truth table

	State		LA		LB	
State	S1	S0	LA1	LA0	LB1	LB0
AGreen	0	0	0	0	1	0
AYellow	0	1	0	1	1	0
BGreen	1	0	1	0	0	0
BYellow	1	1	1	0	0	1

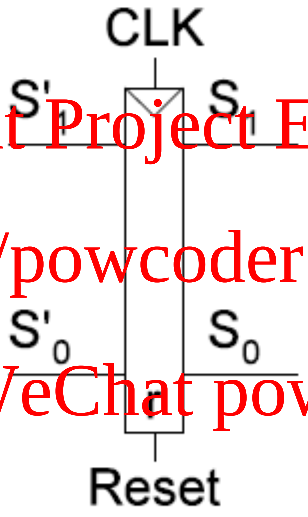
red light

- Compute output bits as function of state bits

$$L_{A1} = S_1; L_{A0} = \overline{S_1}\, S_0$$

$$L_{B1} = \overline{S_1}; L_{B0} = S_1\, S_0$$

State Register: Assume D-FF

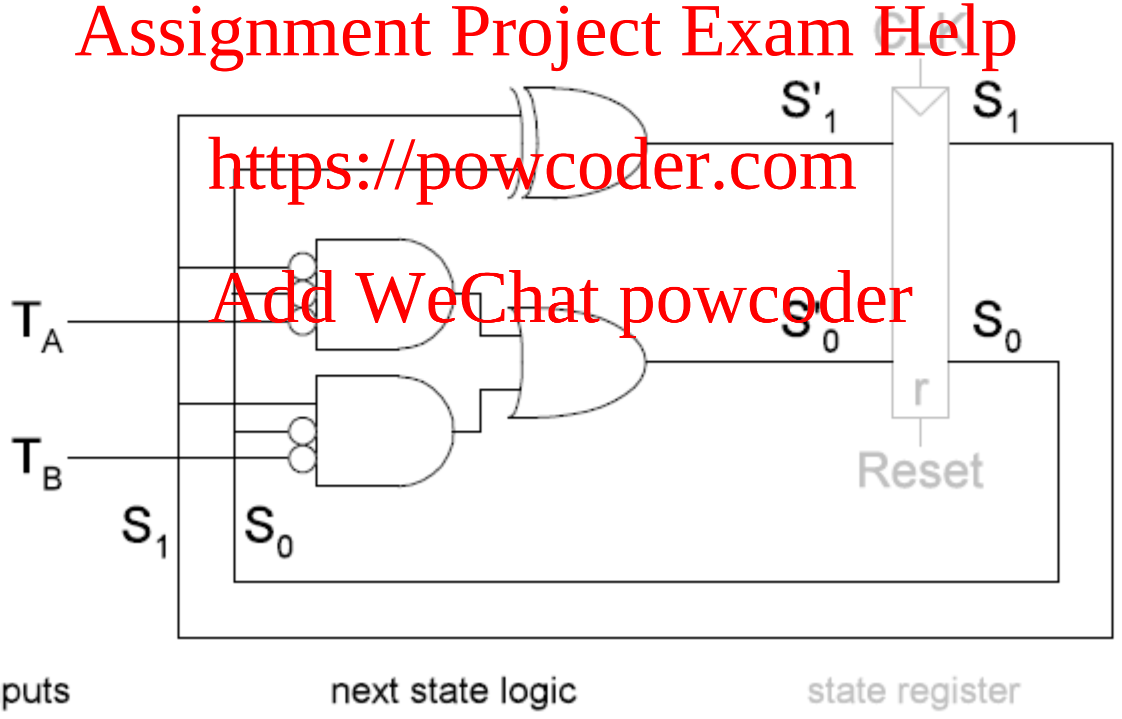


state register

FSM: Figure out Next State Logic

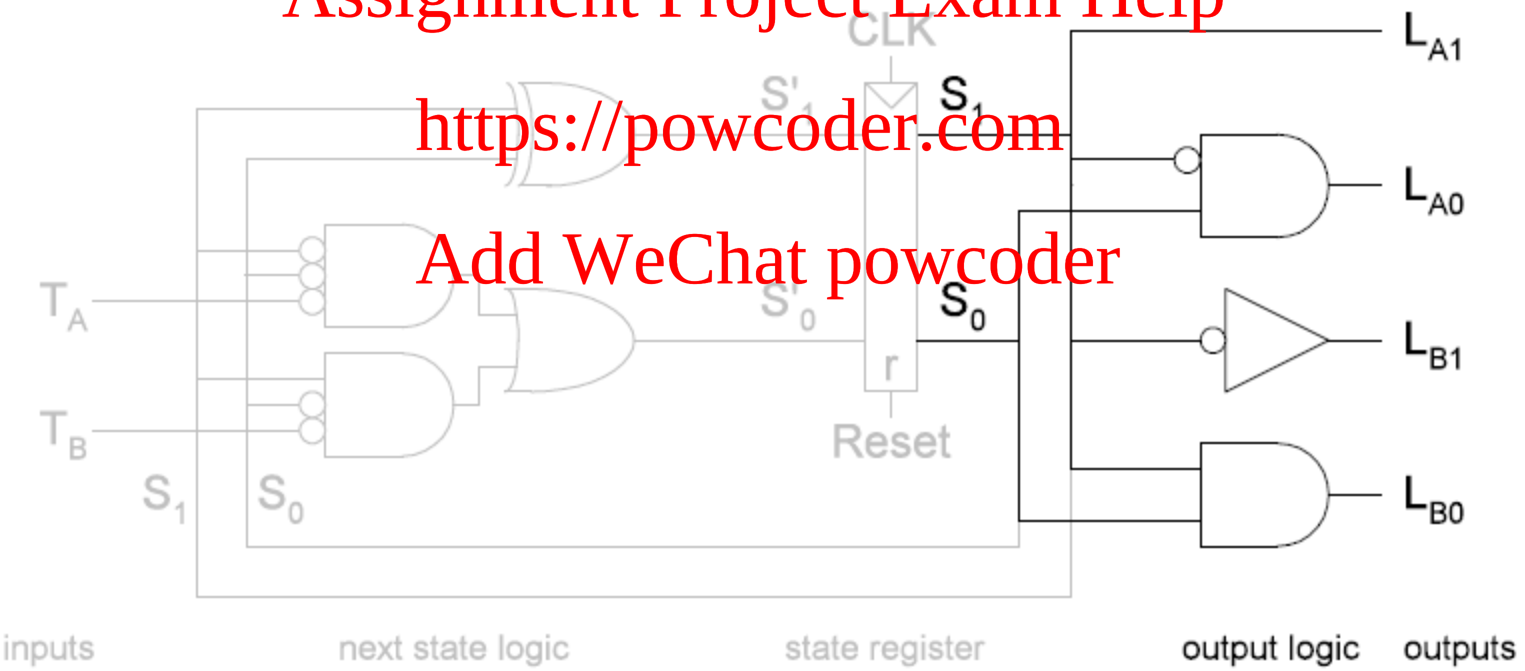
$$S'_1 = S_1 \oplus S_0$$

$$S'_0 = \overline{S_1}\,\overline{S_0}\,\overline{T_A} + S_1\,\overline{S_0}\,\overline{T_B}$$



FSM: Figure out Output Logic

$$L_{A1} = S_1; L_{A0} = \overline{S_1}\, S_0$$
$$L_{B1} = \overline{S_1}; L_{B0} = S_1\, S_0$$



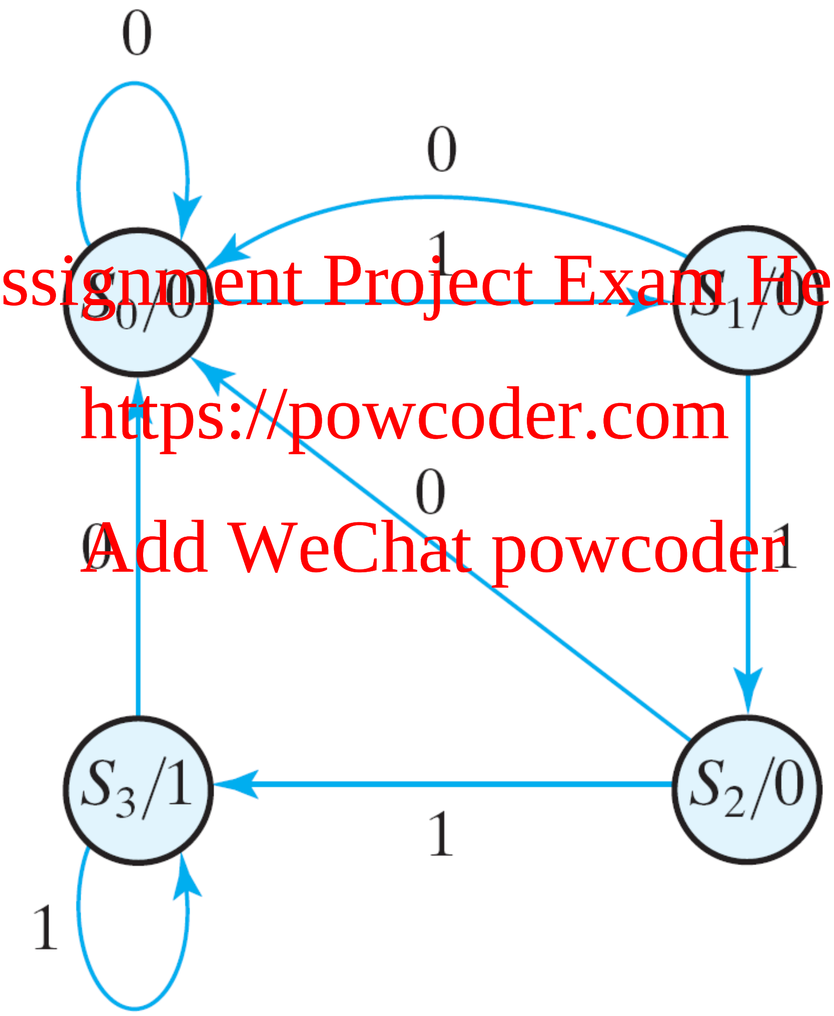
FSM Example 2

Design an FSM that detects a stream of three or more consecutive 1s on an input stream

Input: 0 1 1 1 0 1 0 1 1 0 1 1 1 0 1 …

Output: 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 …

Finite State Machine for the 3 1's problem



FSM Truth Table

Truth Table for Next State (AN and BN are next states)

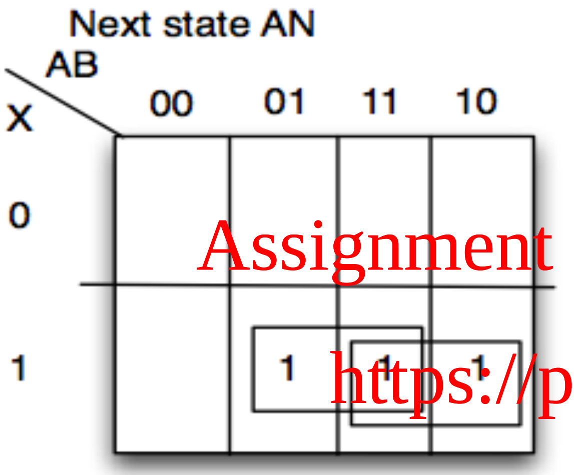
A	B	X	AN	BN
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

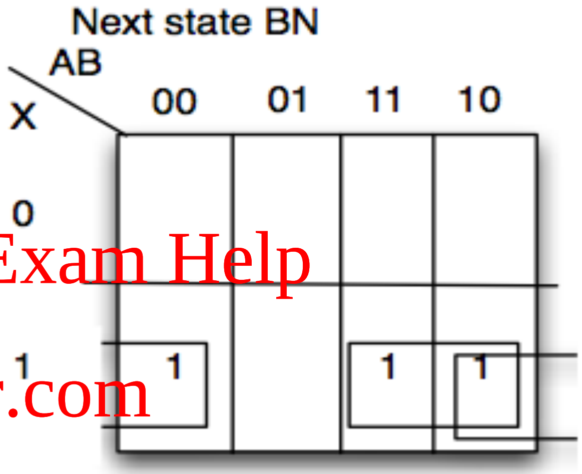
Encoding

	A	B
S0	0	0
S1	0	1
S2	1	0
S3	1	1

We need two bits
to encode 4 states
(lets call these bits A & B)

FSM with D-Flip Flops





$A' = A\,X + B\,X = (A + B)\,X$

$B' = A\,X + \overline{B}\,X = (A + \overline{B}\,)\,X$

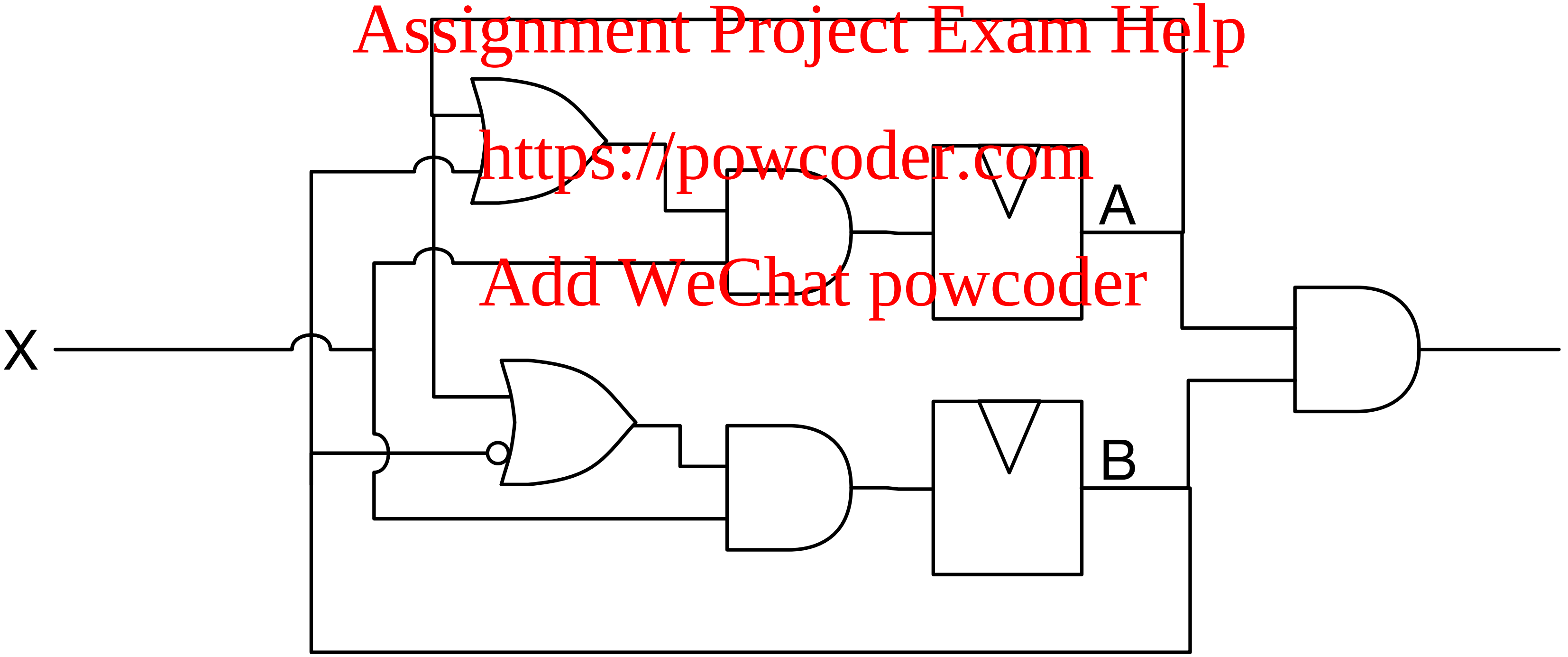
Truth Table for Output

AN	BN	Y
0	0	0
0	1	0
1	1	1
1	0	0

Y = AN. BN

FSM Circuit

$$A' = A\,X + B\,X = (A + B)\,X$$
$$B' = A\,X + \overline{B}\,X = (A + \overline{B})\,X$$



Backup

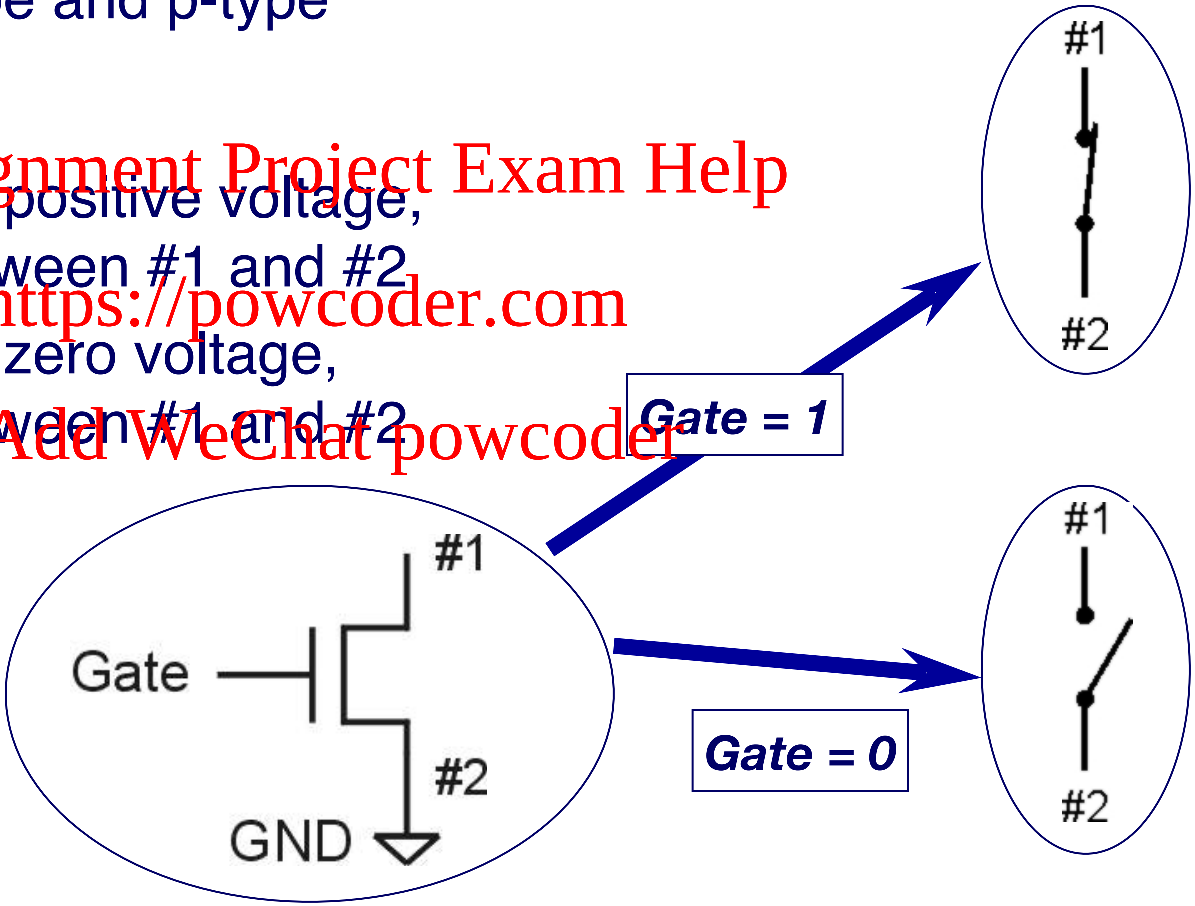
n-type MOS Transistor

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

n-type

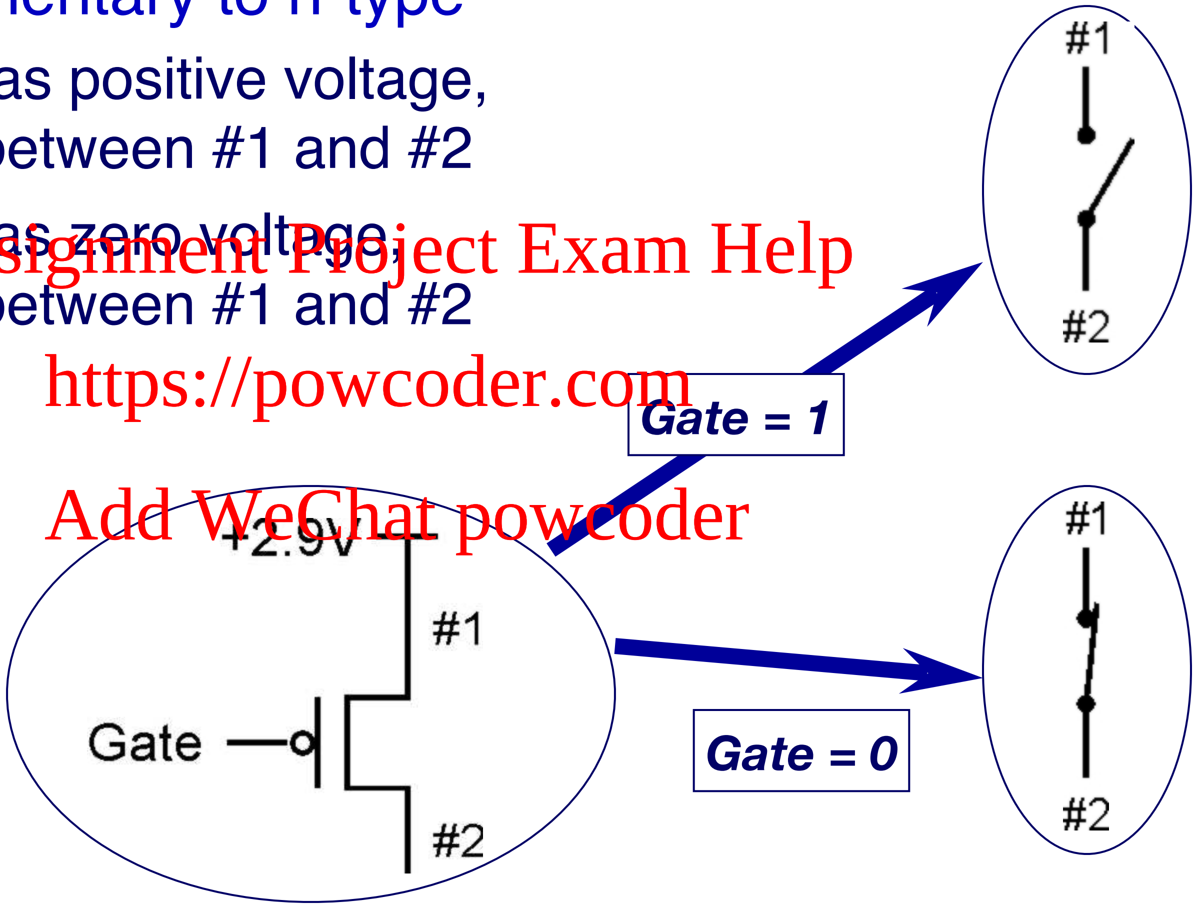
- when Gate has positive voltage, short circuit between #1 and #2
- when Gate has zero voltage, open circuit between #1 and #2



p-type MOS Transistor

p-type is complementary to n-type

- when Gate has positive voltage, open circuit between #1 and #2
- when Gate has zero voltage, short circuit between #1 and #2



CMOS Circuit

Complementary MOS

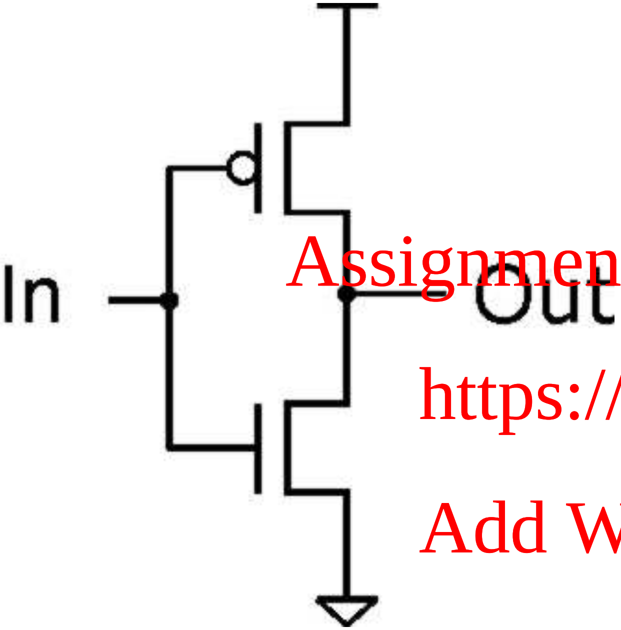
Uses both n-type and p-type MOS transistors

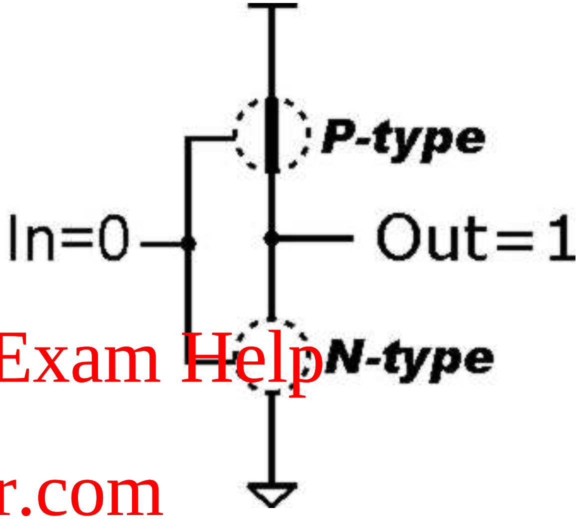
- p-type
  - Attached to + voltage
  - Pulls output voltage UP when input is zero
- n-type
  - Attached to GND
  - Pulls output voltage DOWN when input is one

MOS transistors are combined to form Logic Gates

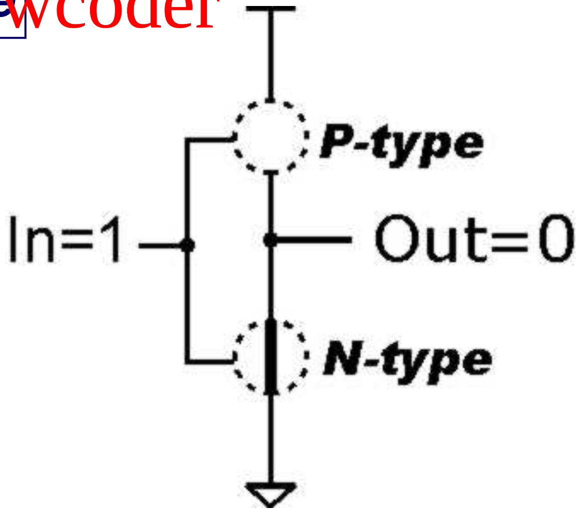
For all inputs, make sure that output is either connected to GND or to +, but not both!

Inverter (NOT Gate)





Truth table



In	Out
0 V	2.9 V
2.9 V	0 V

In	Out
0	1
1	0