Complementary Pullups and Pulldowns

This is what the "C" in CMOS stands for!

We design components with *complementary* pullup and pulldown logic (i.e., the pulldown should be "on" when the pullup is "off" and vice versa).

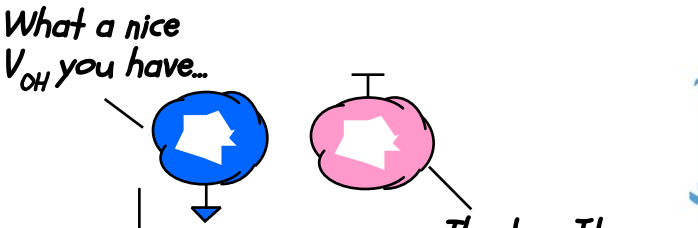
pullup	pulldown	$F(I_p,\ldots,I_n)$
on	off	driven "1"
off	on	driven "0"
on	on	driven "X"
off	off	no connection

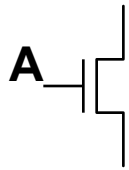
Convention: In general, let's avoid these last two cases.

When they are used, the resulting device is not STRICTLY following our STATIC DISCIPLINE (eg. Pass gates and storage devices).

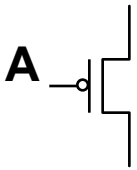
Such devices are only QUASI-DIGITAL!

CMOS Complements



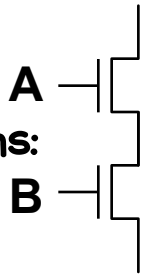


On when A is "1"

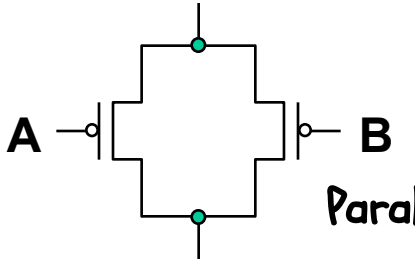


On when A is "0"

Series N connections:



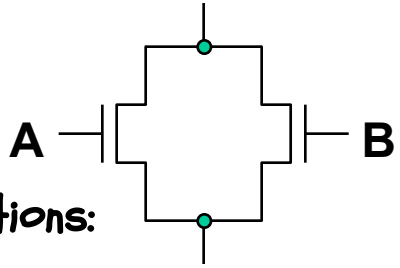
On when A is "1" and B is "1": A && B



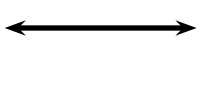
Parallel P connections:

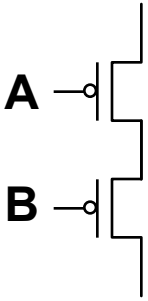
On when A is "0"or B is "0": (!A || !B)

Parallel N connections:



On when A is "1" or B is "1": A || B

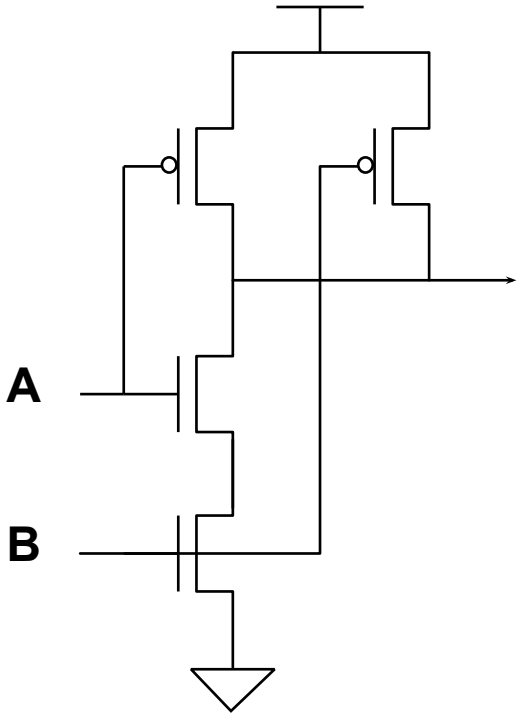




Series P connections:

On when A is "0" and B is "0": (!A && !B)

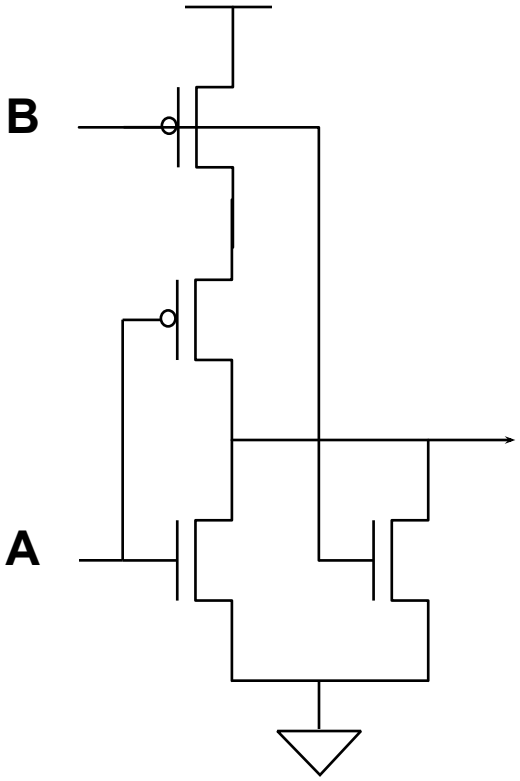
A Two-Input Logic Gate



What function does this gate compute?

A	B	C
0	0	
0	1	
1	0	
1	1	

Here's Another...



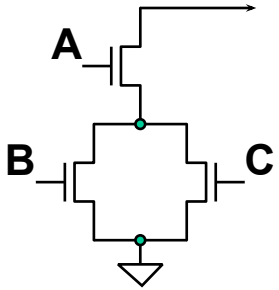
What function does this gate compute?

A	B	C
0	0	
0	1	
1	0	
1	1	

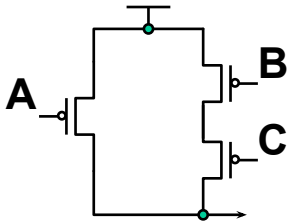
General CMOS Gate Recipe

Step 1. Figure out pulldown network that does what you want (i.e the set of conditions where the output is '0')

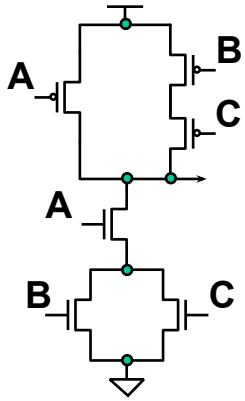
e.g., F = ~~A && (B || C)~~



Step 2. Walk the hierarchy replacing nfets with pfets, series subnets with parallel subnets, and parallel subnets with series subnets



Step 3. Combine pfet pullup network from Step 2 with nfet pulldown network from Step 1 to form fully-complementary CMOS gate.



But isn't it hard to wire it all up?

One Last Exercise

Let's construct a gate to compute:

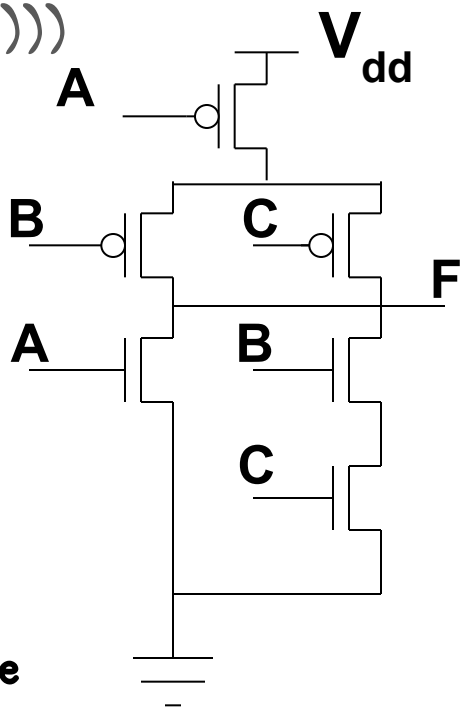
$$F = \overline{A \,\|\, (B \,\&\&\, C)} = \text{NOT}(\text{OR}(A,\text{AND}(B,C)))$$

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

Step 1: The pull-down network

Step 2: The complementary pull-up network

OBSERVATION: CMOS gates tend to be inverting! Precisely, one or more "0" inputs are necessary to generate a "1" output, and one or more "1" inputs are necessary to generate a "0" output. Why?



Next time

Now that we can see what goes on inside of a single gate, we'll next use several them to compose larger systems that compute other logic functions.

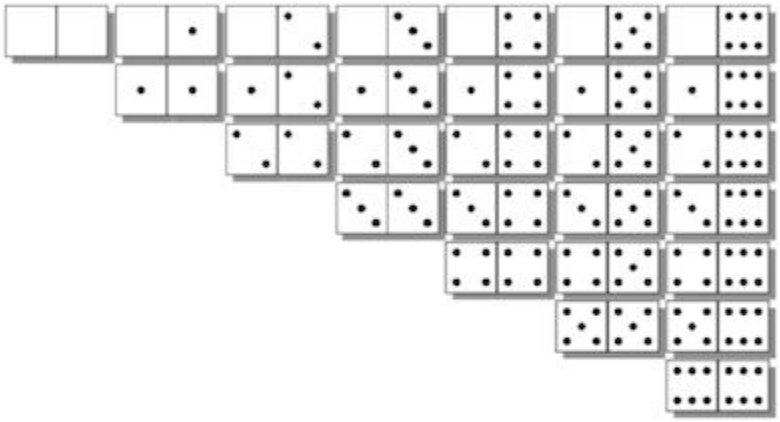
Midterm practice

Which of the following ARM instructions might be encoded as: 0x03A01008?

- ldreq r1,[r3,#10]
- ldreq r1,x
- bxeq r1
- bleq main
- moveq r1,#x

Midterm practice

Shown below is a complete set of double-six domino tiles. 4 points
Suppose someone tells you that a tile they are holding has a 5 spot. How many bits of information are conveyed?



- ○ log2(28/5)
- ○ log2(28/7)
- ○ log2(28/8)
- ○ log2(28/9)
- ○ None of the above

Midterm Practice

If an ASCII character array containing "EDIT" appears in a word of memory as 0x54494445, what can you infer? 4 points

- ○ The computer is using a convention where all strings are aligned to word boundaries
- ○ The letter "E" is encoded in ascii as 01010100
- ○ The machine is byte addressable
- ○ The machine uses little-endian addressing
- ○ All of the above

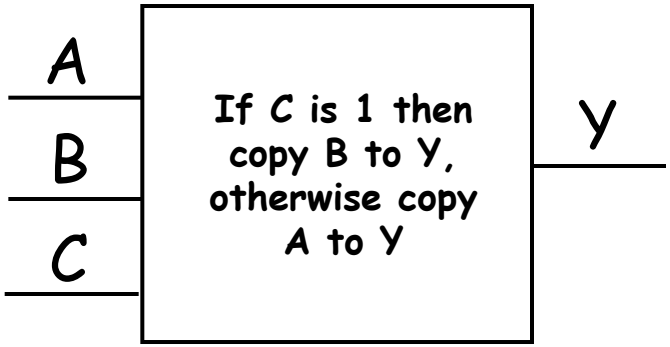
Enumerating and Composing Gates

- Combinational logic as/is truth tables
- Composing gates
- What gates do we have?
- What gates do we need?
- Making gates from others
- A systematic approach for implementing combinational logic

Midterm #1 on Friday

Now can we design larger systems

We need to start somewhere -
usually with a functional specification



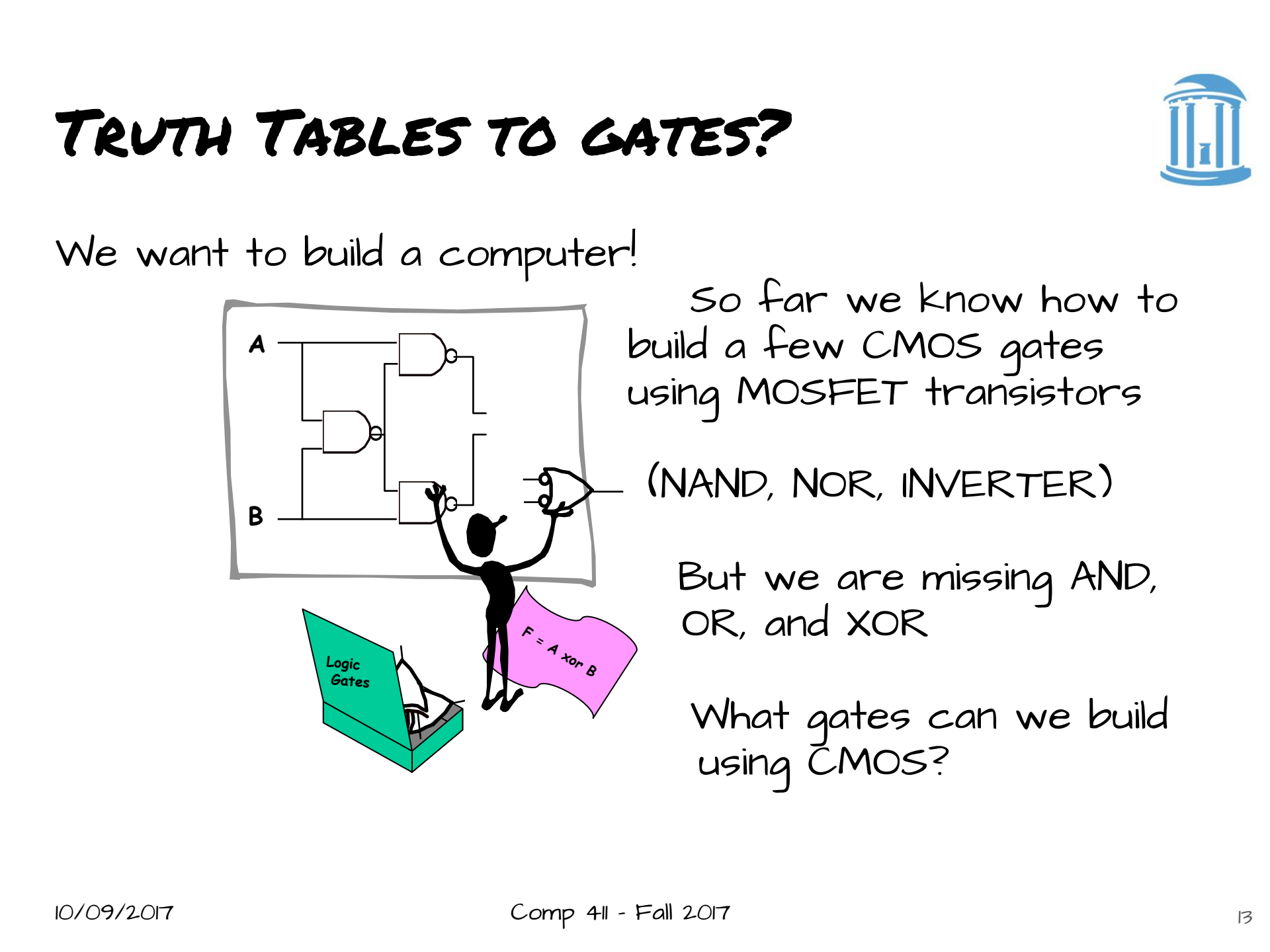
If you are like most pragmatists you'd rather be given a table or formula than solve a puzzle to understand a function. The fact is, **every combinational function can be expressed as a table**.

"**Truth tables**" are a concise description of the combinational system's function, where an output is specified for *every* input combination.

Truth Table

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

Truth Tables to Gates?



We want to build a computer!

So far we know how to build a few CMOS gates using MOSFET transistors

(NAND, NOR, INVERTER)

But we are missing AND, OR, and XOR

What gates can we build using CMOS?

What gates can we build?

Recall, we need to design our gates using a pull-up network of P-FETs and a pull-down network of N-FETs.

What gates can we
- build?
- define?

Let's start by considering only 2-input gates.

AND		OR		NAND		NOR	
AB	Y	AB	Y	AB	Y	AB	Y
00	0	00	0	00	1	00	1
01	0	01	1	01	1	01	0
10	0	10	1	10	1	10	0
11	1	11	1	11	0	11	0

How many possible 2-input gates are there?

KEY IDEA: As many as there are 2-input truth tables.

2-inputs $\rightarrow 2^2 = 4$ rows, each with an output

4-outputs $\rightarrow 2^4 = 16$ possible functions

All the Gates

There are only 16 possible 2-input gates... Let's examine all of them. Some we already know, others are just silly.

INPUT AB	ZERO	A AND B	A > B	A	B > A	B	X OR	OR	NOR	XNOR	NOT 'B'	A <= B	NOT 'A'	B <= A	NAND	ONE
00	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	~~1~~	0	~~1~~	0	~~1~~	0	~~1~~	0	~~1~~	0	~~1~~	0	~~1~~	0	~~1~~

How many of these gates can be implemented using a single CMOS gate?

N-FETs can only pull the ouput to "0", and only if one or more of their inputs is a "1".

P-FETs can only pull the ouput to "1", and only if one or more of their inputs is a "0".

Do we really need all of these gates?

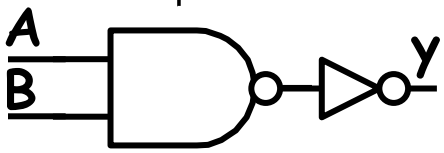
Nope! Once we realize that we can describe all of them using just AND, OR, and NOT

Composing gates to build others

Let's start with a couple of basics, AND and OR. Each can be constructed using a pair of CMOS gates, AND is just NAND with an inverter, and OR is just NOR with an inverted output.

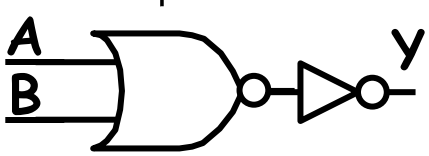
AND

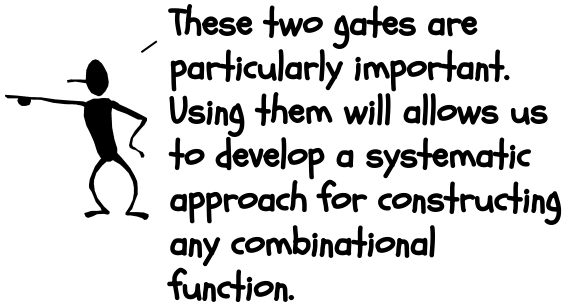
AB	Y
00	0
01	0
10	0
11	1



OR

AB	Y
00	0
01	1
10	1
11	1





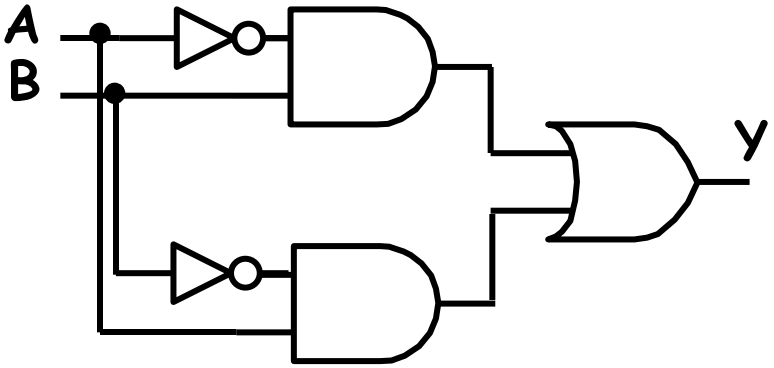
Composing Arbitrary Gates

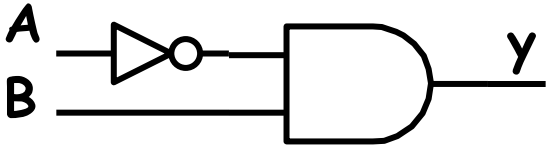
B>A

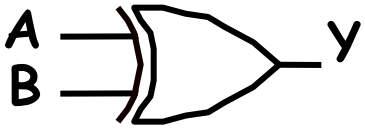
AB	Y
00	0
01	1
10	0
11	0

XOR

AB	Y
00	0
01	1
10	1
11	0







How many different gates do we really need?

We can always do it with 3 different types of gates (AND, OR, INVERT), and sometimes with 2, but, can we use fewer?

The TRICK is to OR the ANDs of all input combinations that generate an output of "1". You don't need the OR gate if only one input combination results in a "1".

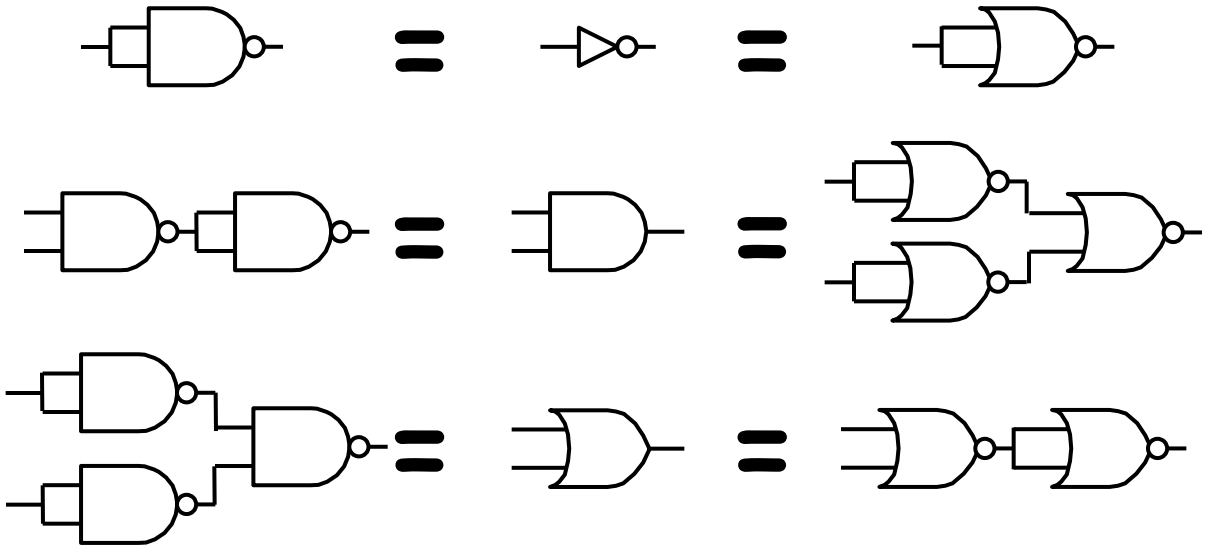
You need Inverters to handle input combinations involving "0"s, ANDs, and ORs.

ONE WILL DO!

A UNIVERSAL gate is one that can be used to implement *ANY* COMBINATIONAL FUNCTION. There are many UNIVERSAL gates, but not all gates are UNIVERSAL

Q: What is a COMBINATIONAL FUNCTION?
A: Any function that can be written as a truth table.

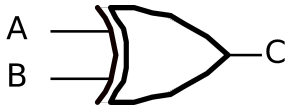
NANDs and NORs are UNIVERSAL!



Ah!, but what if we want more than 2-inputs?

Stupid Gate Tricks

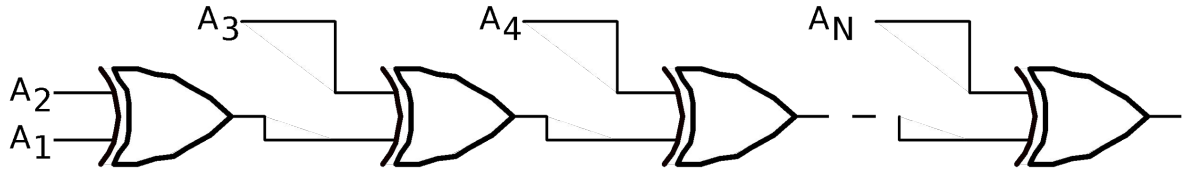
Suppose we have some 2-input XOR gates:



t_{pd} = 1 nS

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

And we want an N-input XOR:



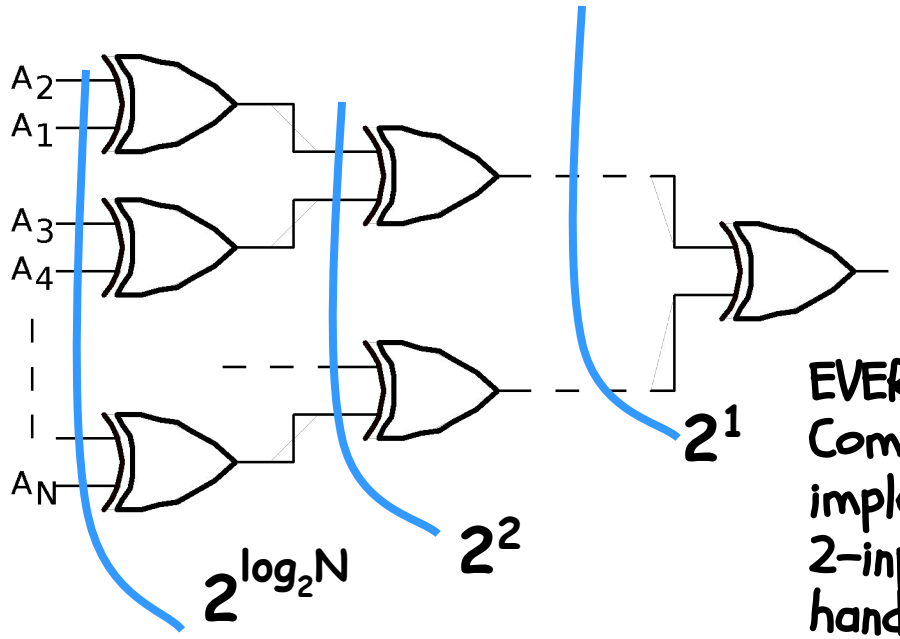
output = 1 iff number of "1"s input is ODD ("PARITY")

t_{pd} = N nS -- WORST CASE.

Can we compute an N-input XOR faster?

I Think That I Shall Never See

a Gate Lovely as a ...



EVERY N-Input Combinational function be implemented using only 2-input gates? But, it's handy to have gates with more than 2-inputs when needed.

N-input TREE has O(log N) levels...

Signal propagation takes O(log N) gate delays.

A SYSTEMATIC DESIGN APPROACH

Truth Table

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

-it's systematic!
-it works!
-it's easy!
-we get to go home!

1) Write the functional spec as a truth table
2) Write down a Boolean expression for every '1' in the output

```
Y = (!C && !B && A) || (!C && B && A)
  || (C && B && !A) || (C && B && A)
```

3) Wire up the ideal gates, replace them with equivalent realizable gates, call it a day, and go home!

This approach will always give us logic expressions in a particular form:

SUM-OF-PRODUCTS

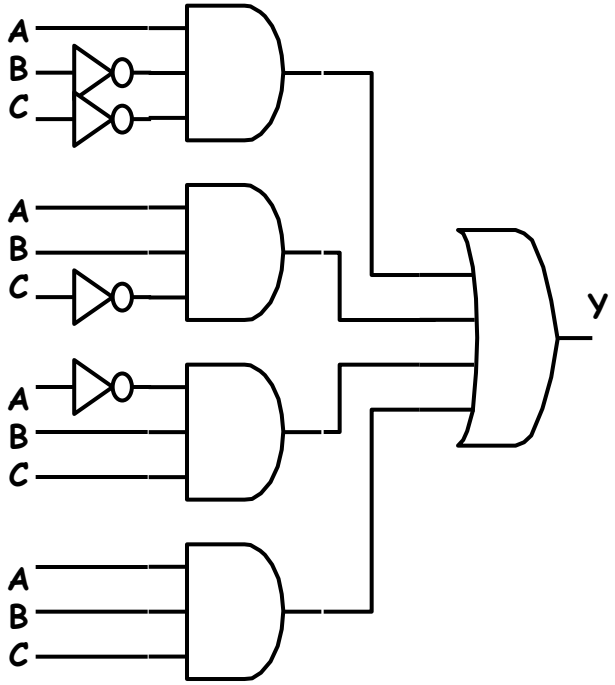
Straightforward Synthesis

We can implement

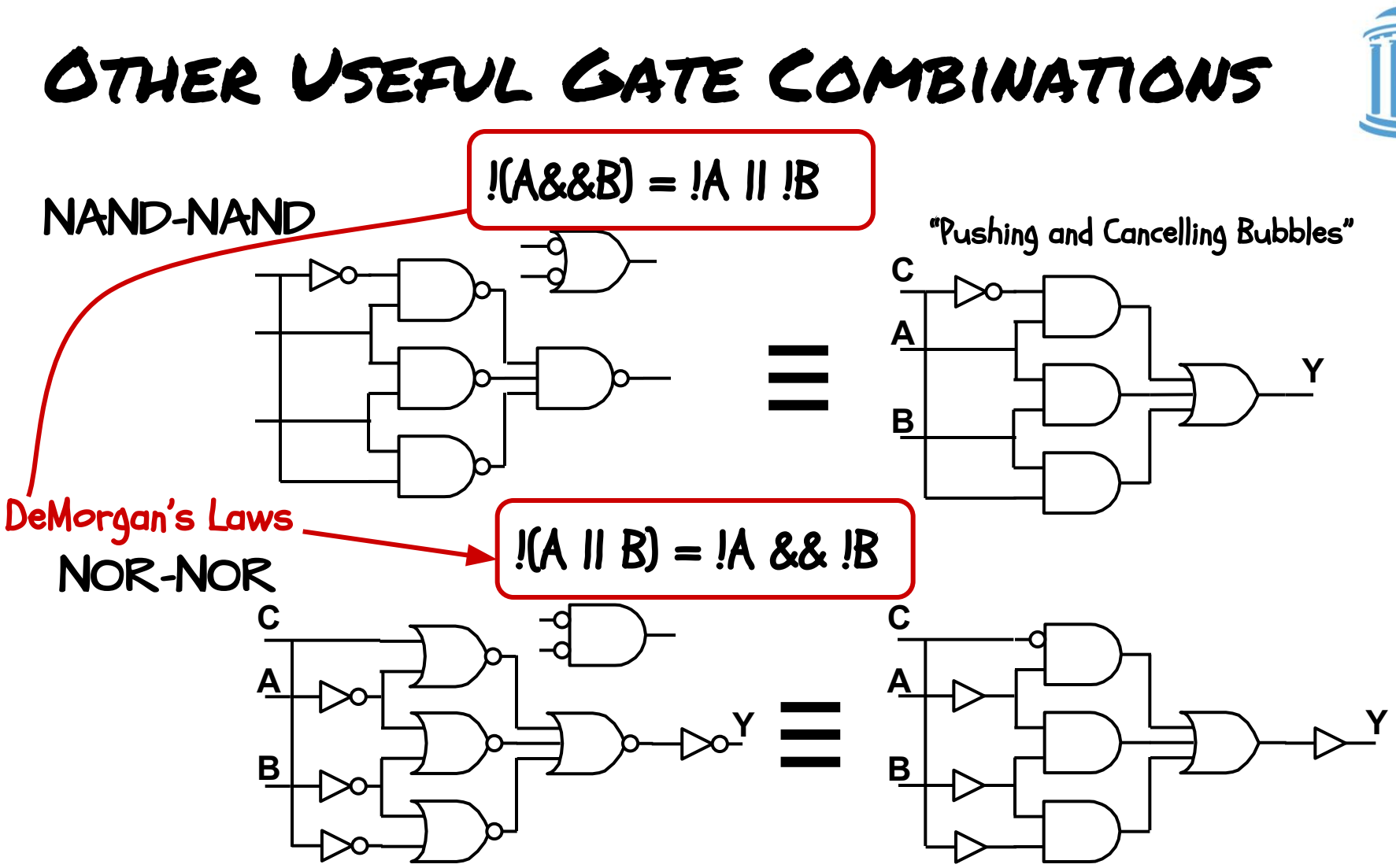
SUM-OF-PRODUCTS

with just 3 levels of logic.

INVERTERS/AND/OR

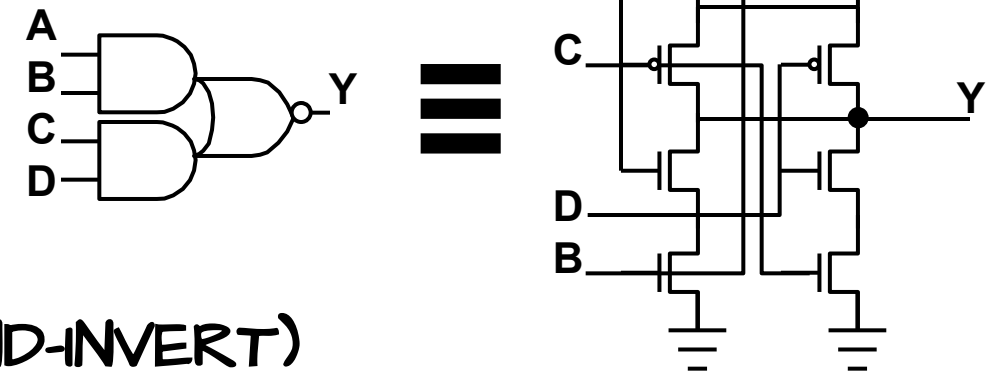


Other Useful Gate Combinations



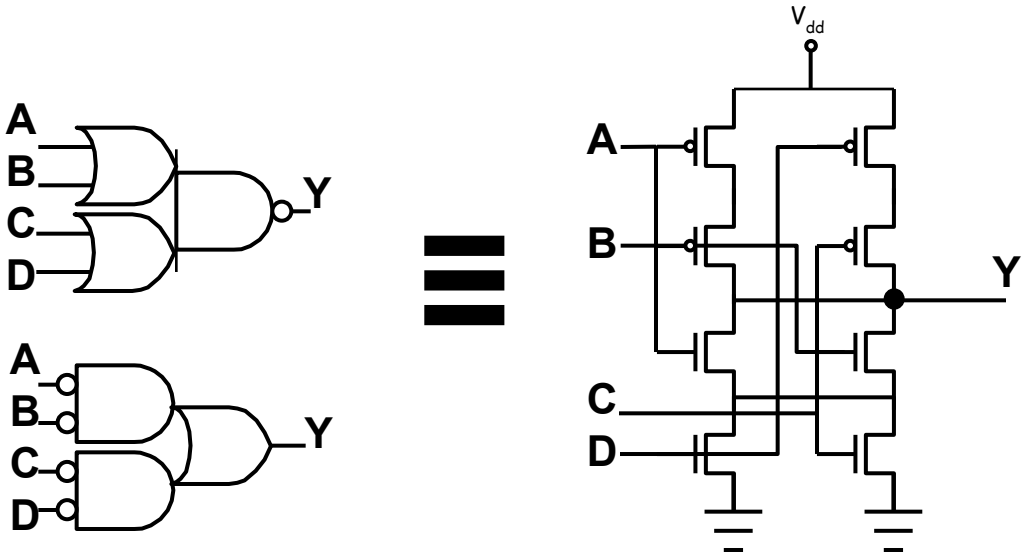
Other Useful CMOS Gates

AOI (AND-OR-INVERT)



AOI and OAI structures can be realized as a single CMOS gate. However, their function is equivalent to 3 levels of logic.

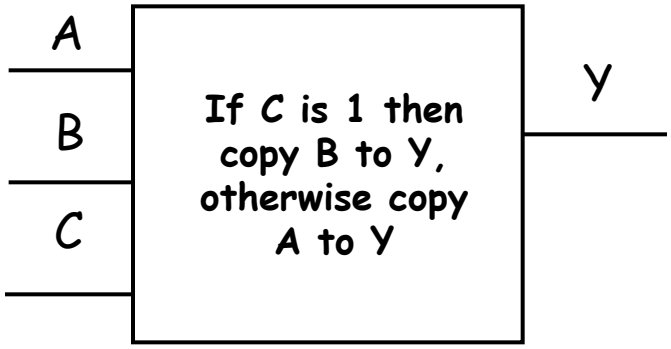
OAI (OR-AND-INVERT)



An OAI's DeMorgan equivalent is usually easier to think about.

An Interesting 3-Input Gate

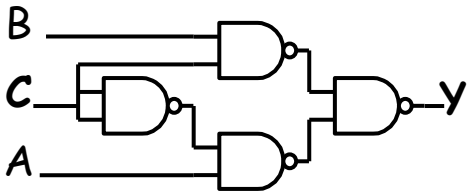
Based on C, select the A or B input to be copied to Y.



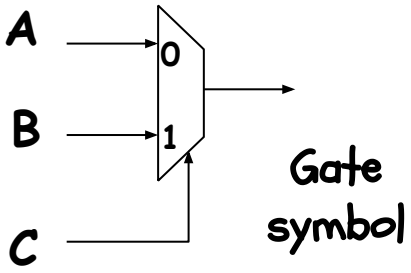
Truth Table

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

2-input Multiplexer



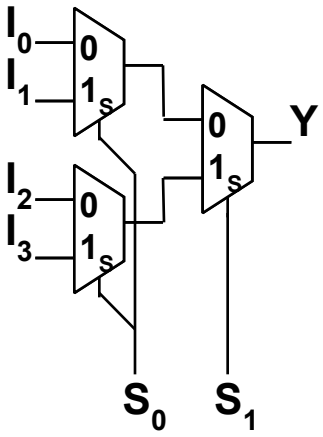
schematic

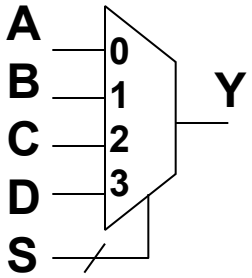


Gate symbol

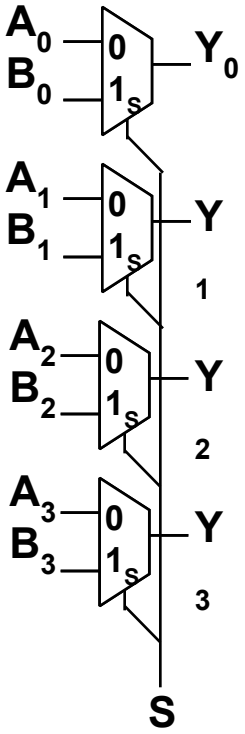
MUX Compositions and Shortcuts

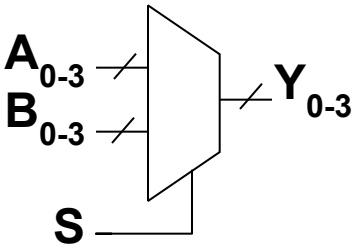
A 4-input Mux
(implemented as a tree)





A 4-bit wide 2-input Mux



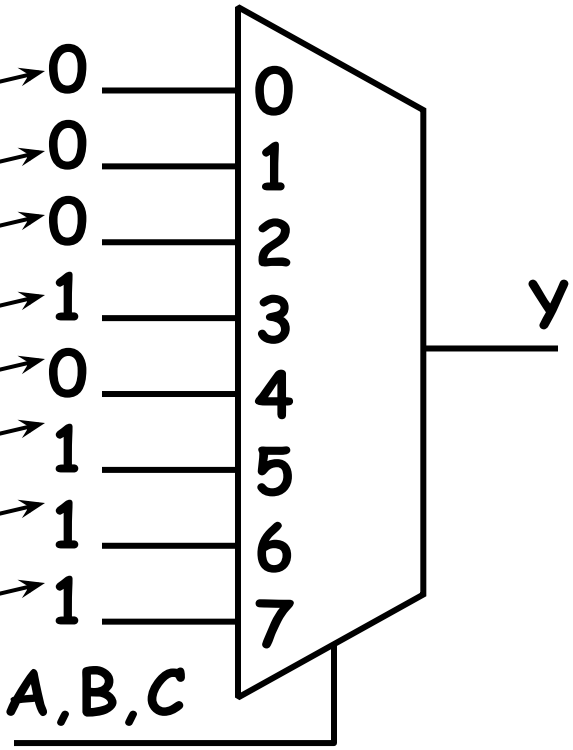


MUX Function Synthesis

Consider implementation of some arbitrary Combinational function, F(A,B,C)… using a MULTIPLEXER as the only circuit element:

Mux Logic: An example "configurable" logic element

A	B	C	Y
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

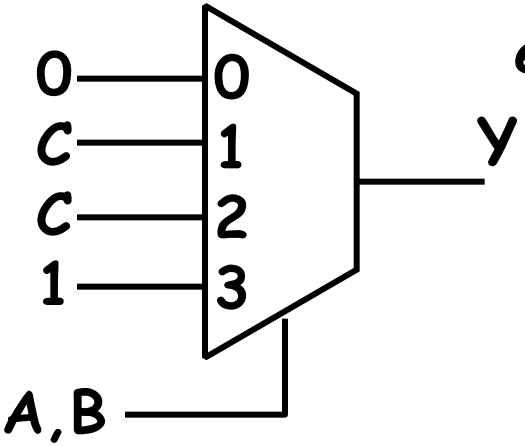


MUX Logic Tricks

We can apply certain optimizations to MUX Function synthesis

Desired Logic Function

A	B	C	Y
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



- Largely by inspection or exhaustive search

- N-input gate with an N-1 input MUX & one inverter

Next Time

Binary Circuits that:

- ADD
- SUBTRACT
- SHIFT