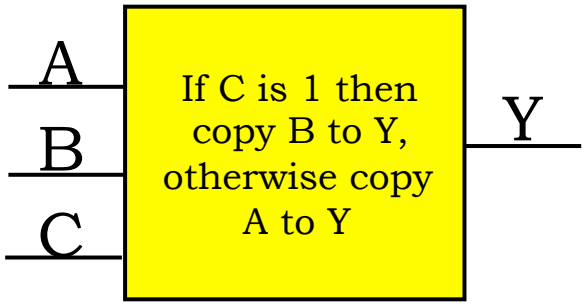
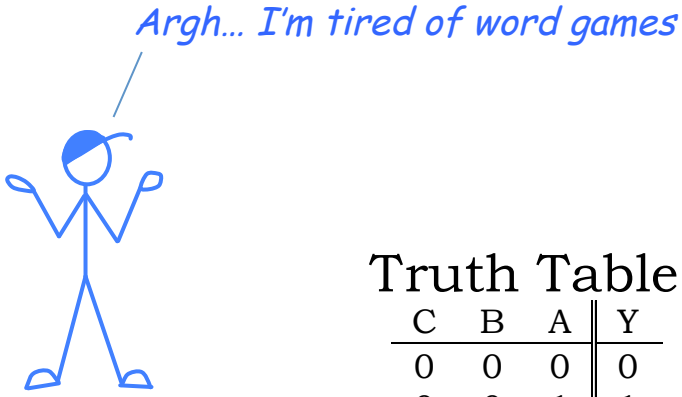
4. Combinational Logic

6.004x Computation Structures
Part 1 – Digital Circuits

Functional Specifications

There are many ways of specifying the function of a combinational device, for example:





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

Concise alternatives:

- *truth tables* are a concise description of the combinational system's function.
- *Boolean expressions* form an algebra whose operations are AND (multiplication), OR (addition), and inversion (overbar).

$$Y = \overline{C} \cdot \overline{B} \cdot A + \overline{C}BA + CB\overline{A} + CBA$$

Any combinational (Boolean) function can be specified as a truth table or an equivalent sum-of-products Boolean expression!

Here's a 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

1. Write out our functional spec as a truth table
2. Write down a Boolean expression with terms covering each '1' in the output:

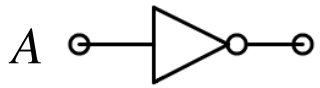
$$Y = \bar{C}\bar{B}A + \bar{C}B\bar{A} + CB\bar{A} + CBA$$

3. We'll show how to build a circuit using this equation in the next two slides.

This approach will always give us Boolean expressions in a particular form: SUM-OF-PRODUCTS

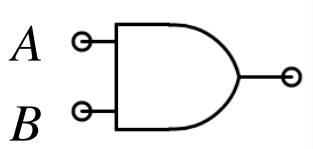
Sum-of-products Building Blocks

INVERTER:



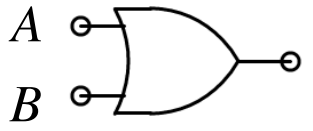
A	Z
0	1
1	0

AND:



A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR:

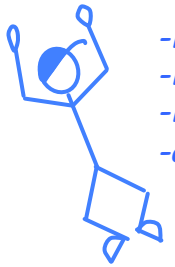


A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

Straightforward Synthesis

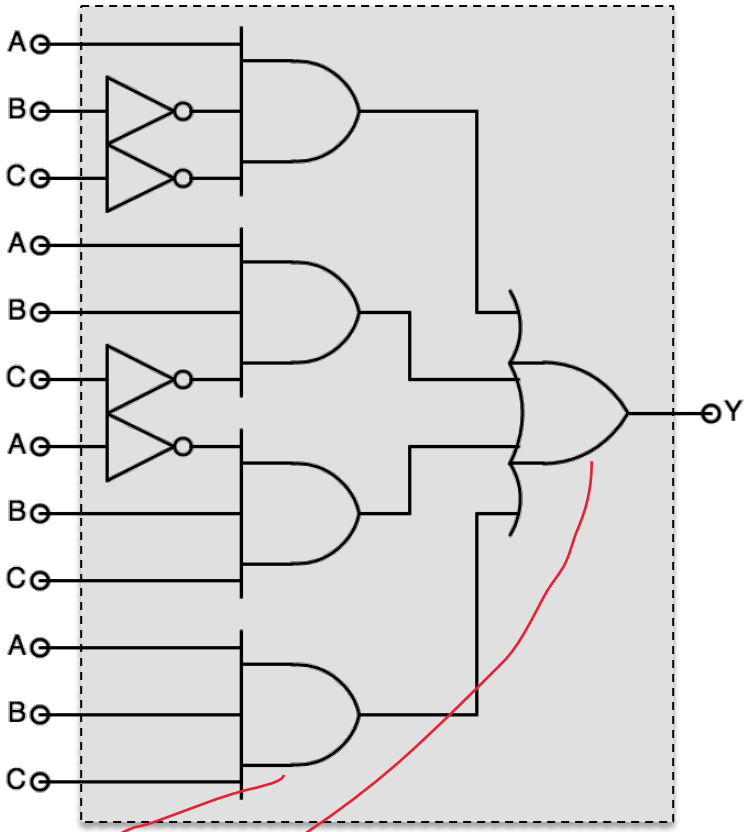
We can implement SUM-OF-PRODUCTS with just three levels of logic:

1. Inverters
2. ANDs
3. OR



-it's systematic!
-it works!
-it's easy!
-are we done yet???

$$Y = \overline{C}\overline{B}A + \overline{C}BA + C\overline{B}\overline{A} + CBA$$



Propagation delay -- No more than 3 gate delays?*

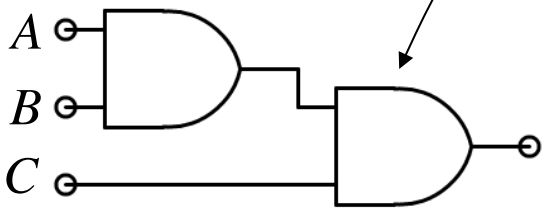
(这里每个AND gate**可以有三个**input，OR gate**可以有四个**input)

*assuming gates with an arbitrary number of inputs, which, as we'll see, isn't a good assumption!

然而实际1**个**gate**一般只能有**2**个**input

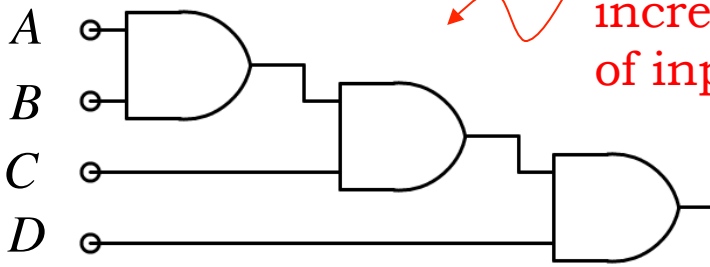
ANDs and ORs with > 2 Inputs

Replace 2-input AND gates with 2-input OR gates to create large fan-in OR gates.



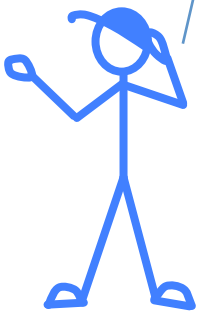
$$Z = A \cdot B \cdot C = (A \cdot B) \cdot C$$

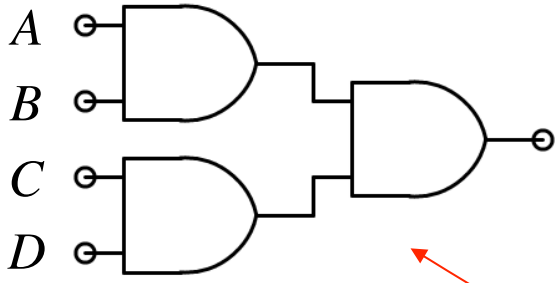
Chain: Propagation delay increases linearly with number of inputs



$$Z = ((A \cdot B) \cdot C) \cdot D$$

Which one should I use?



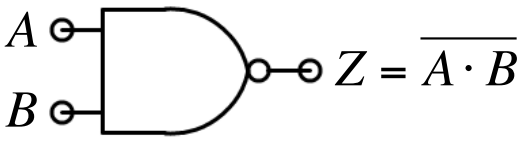


$$Z = (A \cdot B) \cdot (C \cdot D)$$

Tree: Propagation delay increases logarithmically with number of inputs

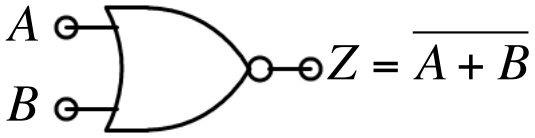
More Building Blocks

NAND (not AND)



A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

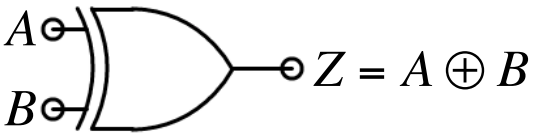
NOR (not OR)



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

In a CMOS gate, rising inputs lead to falling outputs and vice-versa, so CMOS gates are naturally inverting. Want to use NANDs and NORs in CMOS designs… But NAND and NOR operations are not associative, so wide NAND and NOR gate can't use a chain or tree strategy. Stay tuned for more on this!

XOR (exclusive OR)



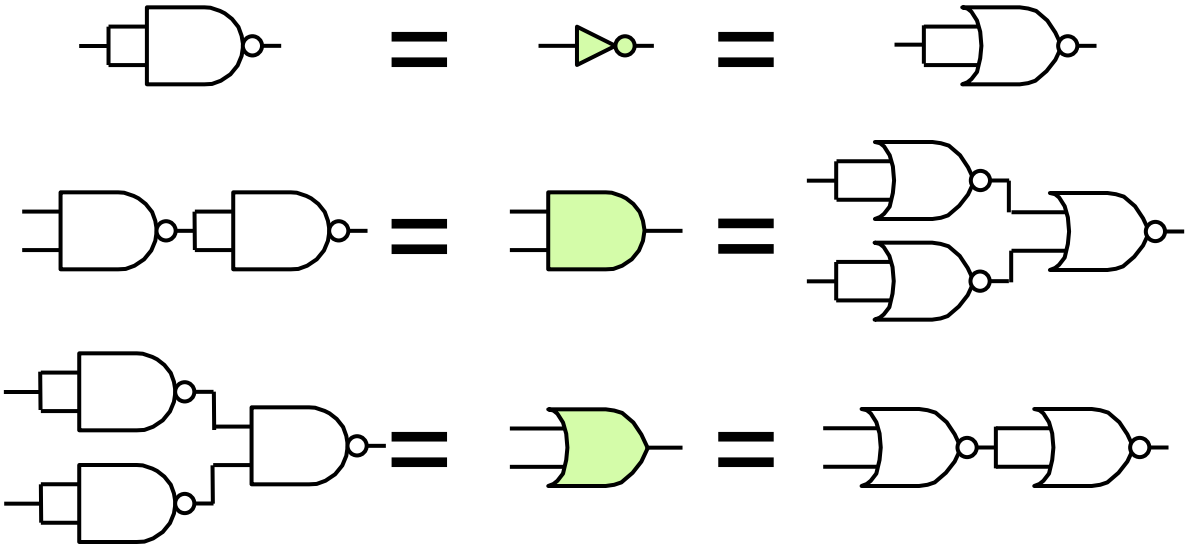
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

XOR is very useful when implementing parity and arithmetic logic. Also used as a "programmable inverter": if A=0, Z=B; if A=1, Z=~B

Wide fan-in XORs can be created with chains or trees of 2-input XORs.

Universal Building Blocks

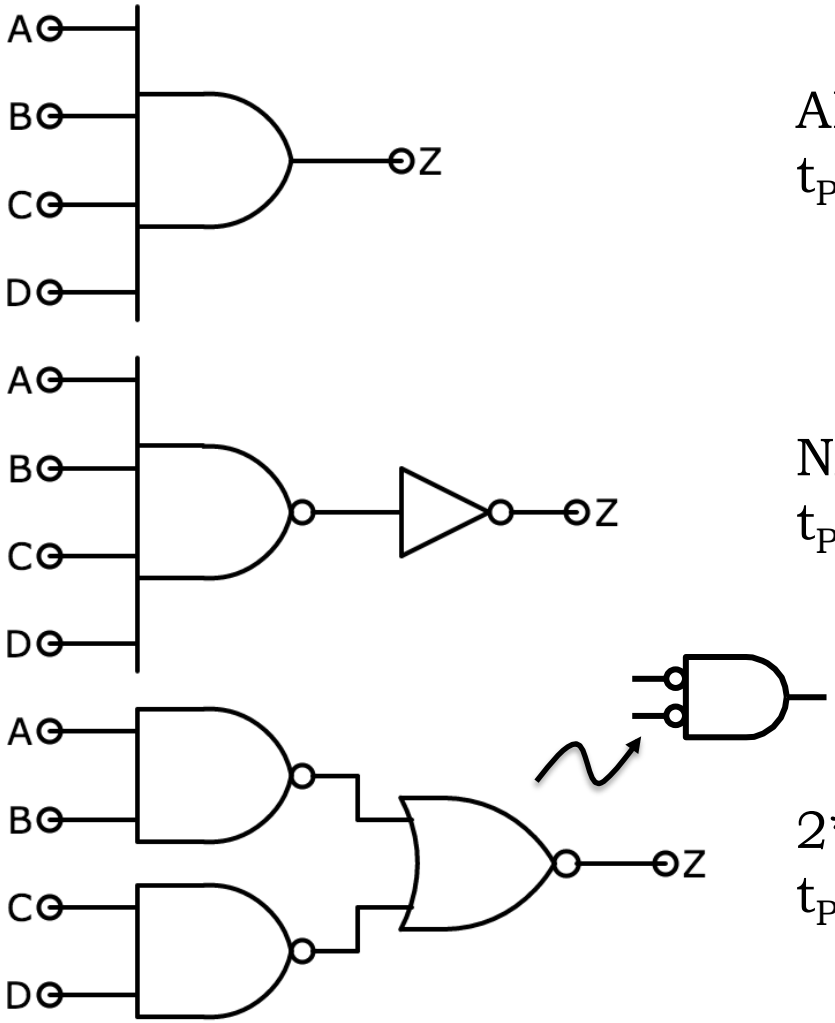
NANDs and NORs are universal:



Any logic function can be implemented using only NANDs (or, equivalently, NORs). Good news for CMOS technologies!

CMOS ♥ Inverting Logic

See "The Standard Cell Library" handout in *Updates & Handouts*



AND4:
t_{PD} = 160 ps, size = 20 μ^2

NAND4 + INV:
t_{PD} = 90 ps, size = 27 μ^2

Demorgan's Laws:

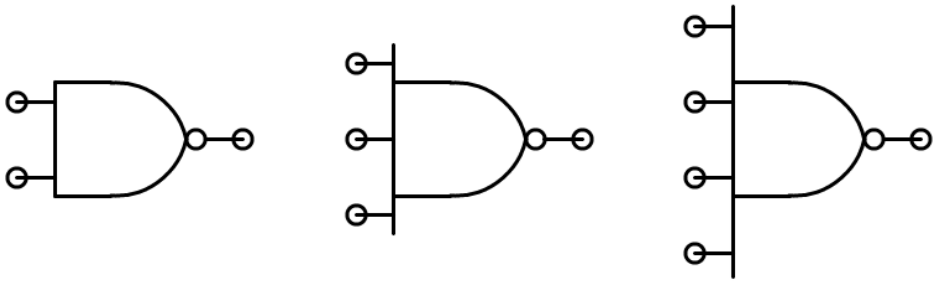
$$\overline{A} \cdot \overline{B} = \overline{A + B}$$

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

2*NAND2 + NOR2:
t_{PD} = 80 ps, size = 30 μ^2

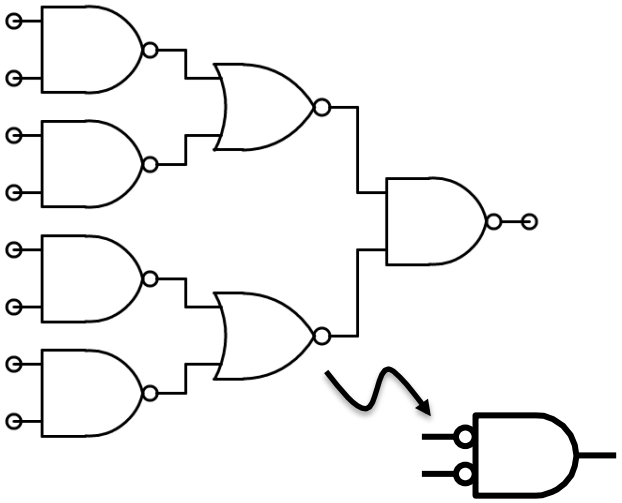
Wide NANDs and NORs

Most logic libraries include 2-, 3- and 4-input devices:

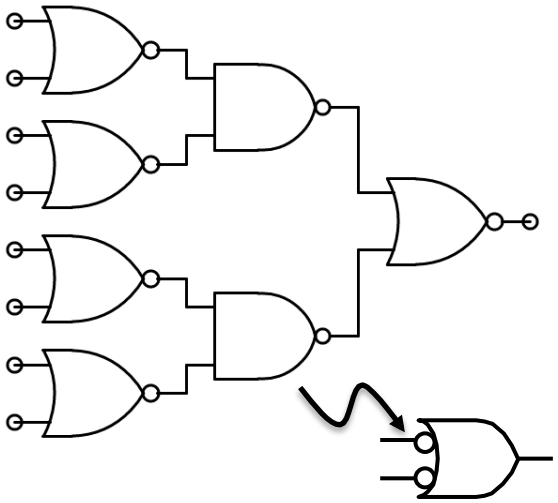


But for a large number of inputs, the series connections of too many MOSFETs can lead to very large effective R. Design note: use trees of smaller devices…

8-input NAND

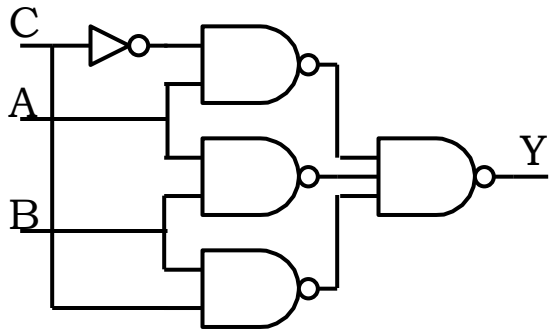


8-input NOR



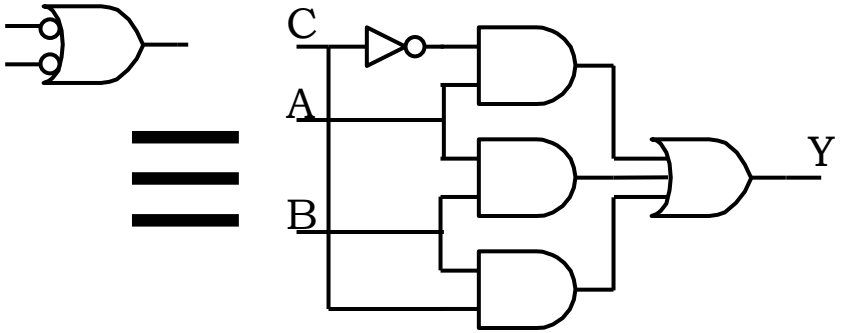
CMOS Sum-of-products Implementation

NAND-NAND



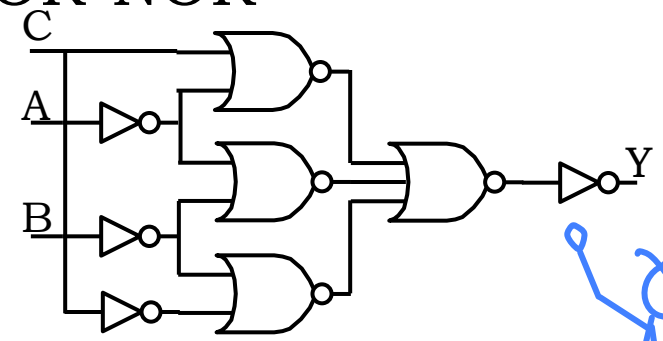
$\overline{AB}=\overline{A}+\overline{B}$

"Pushing Bubbles"

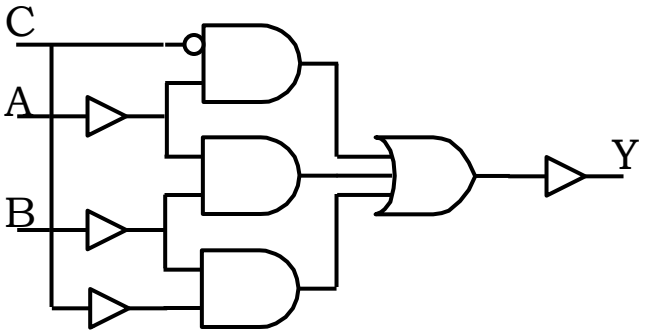


$A\overline{C}+AB+BC$

NOR-NOR



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



$A\overline{C}+AB+BC$

You might think all these extra inverters would make this structure less attractive. However, quite the opposite is true.

Logic Simplification

Can we implement the same function with fewer gates? Before trying we'll add a few more tricks in our bag.

BOOLEAN ALGEBRA:

OR rules:	a + 1 = 1, a + 0 = a, a + a = a
AND rules:	a1 = a, a0 = 0, aa = a
Commutative:	a + b = b + a, ab = ba
Associative:	(a + b) + c = a + (b + c), (ab)c = a(bc)
Distributive:	a(b+c) = ab + ac, a + bc = (a+b)(a+c)
Complements:	$a+\overline{a}=1, \quad a\overline{a}=0$
Absorption:	$a+ab=a, \quad a+\overline{a}b=a+b \qquad a(a+b)=a, \quad a(\overline{a}+b)=ab$
Reduction:	$ab+\overline{a}b=b, \quad (a+b)(\overline{a}+b)=b$
DeMorgan's Law:	$\overline{a}+\overline{b}=\overline{ab}, \quad \overline{a}\overline{b}=\overline{a+b}$

Boolean Minimization

Let's (again!) simplify

$$Y = \overline{C}\,\overline{B}A + C\overline{B}\,\overline{A} + CBA + \overline{C}BA$$

Using the identity

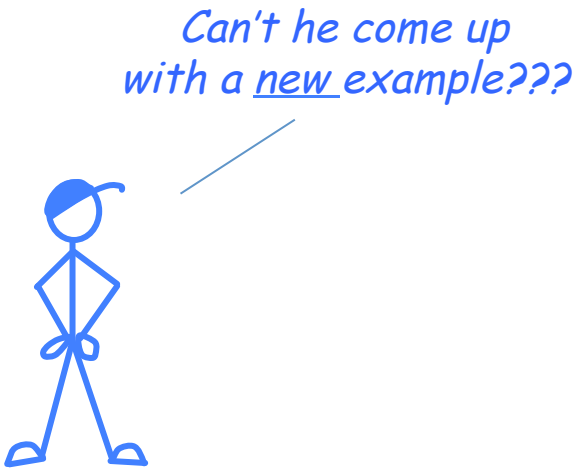
$$\alpha A + \alpha\overline{A} = \alpha(A + \overline{A}) = \alpha \cdot 1 = \alpha$$

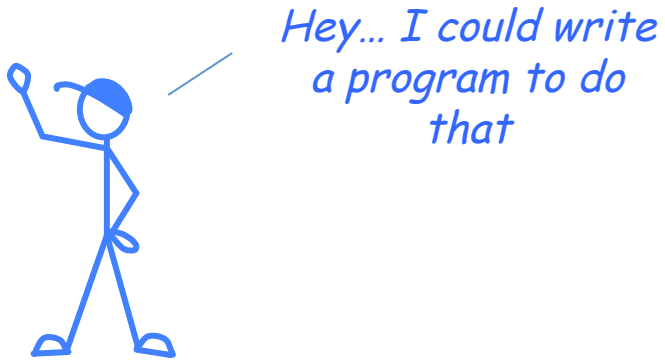
For any expression α and variable A:

$$Y = \overline{C}\,\overline{B}A + C\overline{B}\,\overline{A} + CBA + \overline{C}BA$$

$$Y = \overline{C}\,\overline{B}A + CB + \overline{C}BA$$

$$Y = \overline{C}A + CB$$

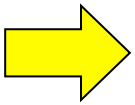




Truth Tables with "Don't Cares"

One way to reveal the opportunities for a more compact implementation is to rewrite the truth table using "don't cares" (-- or X) to indicate when the value of a particular input is irrelevant in determining the value of the output.

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



C	B	A	Y	
0	X	0	0	
0	X	1	1	$\rightarrow \overline{C}A$
1	0	X	0	
1	1	X	1	$\rightarrow CB$
X	0	0	0	
X	1	1	1	$\rightarrow BA$

Note: Some input combinations (e.g., 000) are matched by more than one row in the "don't care" table. It would be a bug if all the matching rows didn't specify the same output value!

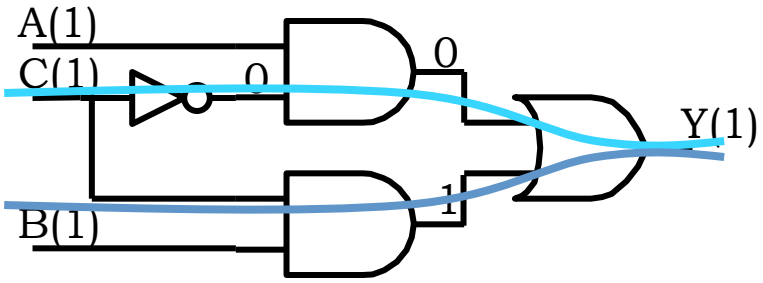
The Case for a Non-minimal SOP

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

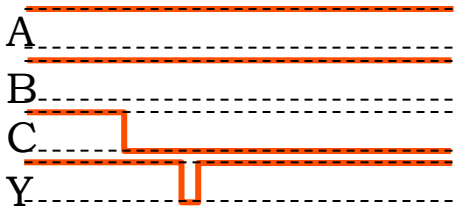
$\overline{C}A$

CB

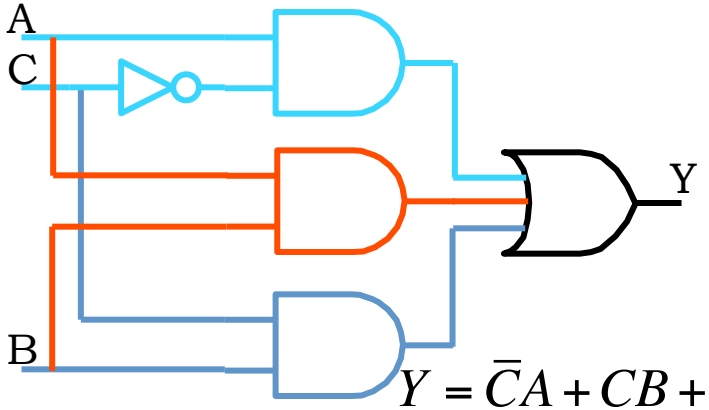
BA



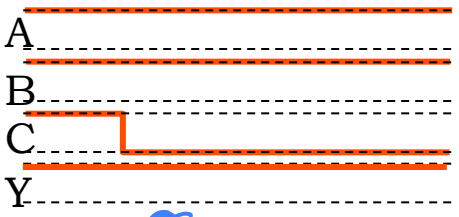
$$Y = \overline{C}A + CB$$



NOTE: The steady state behavior of these circuits is identical. They differ in their transient behavior.



$$Y = \overline{C}A + CB + AB$$



Karnaugh Maps: A Geometric Approach

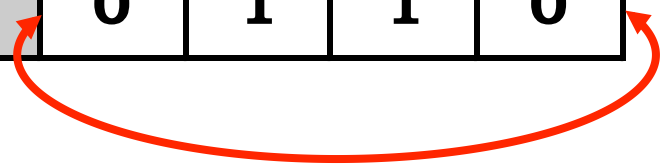
K-Map: a truth table arranged so that terms which differ by exactly one variable are adjacent to one another so we can see potential reductions easily.

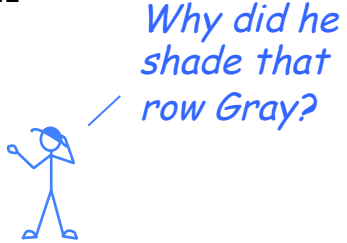
Truth Table

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

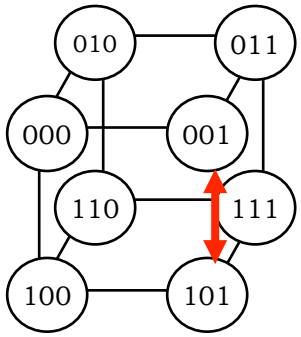
Here's the layout of a 3-variable K-map filled in with the values from our truth table:

C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0





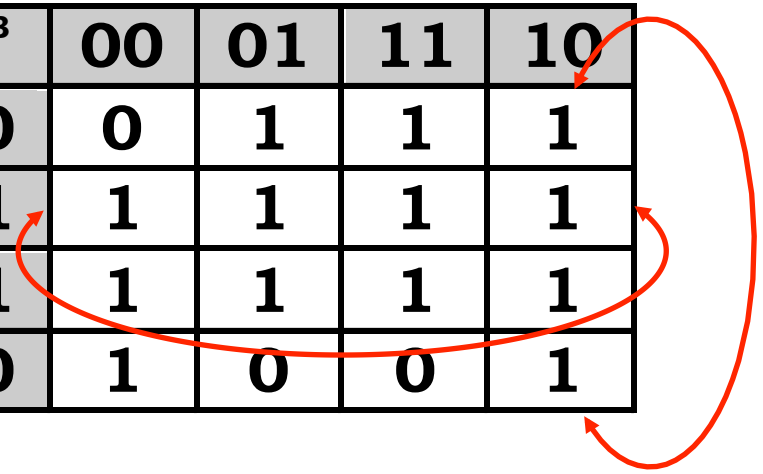
It's cyclic. The left edge is adjacent to the right edge. (It's really just a flattened out cube).



Extending K-maps to 4-variable Tables

4-variable K-map F(A,B,C,D):

\AB CD\	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1



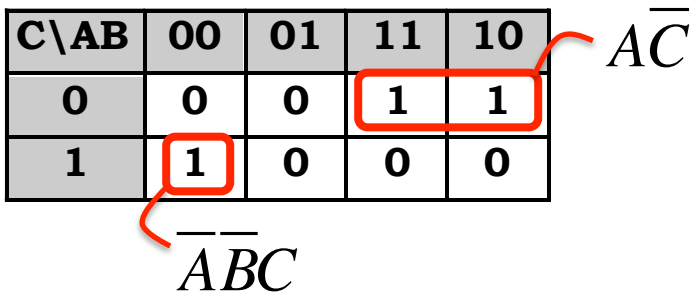
Again it's cyclic. The left edge is adjacent to the right edge, and the top is adjacent to the bottom.

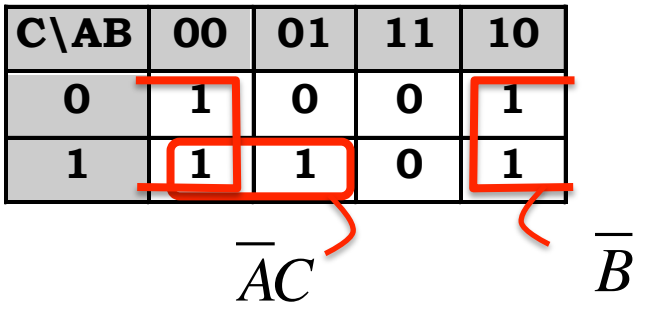
For functions of 5 or 6 variables, we'd need to use the 3rd dimension to build a 4x4x4 K-map. But then we're out of dimensions…

Finding Implicants

An implicant

- is a rectangular region of the K-map where the function has the value 1 (i.e., a region that will need to be described by one or more product terms in the sum-of-products)
- has a width and length that must be a power of 2: 1, 2, 4
- can overlap other implicants
- is a prime implicant if it is not completely contained in any other implicant.



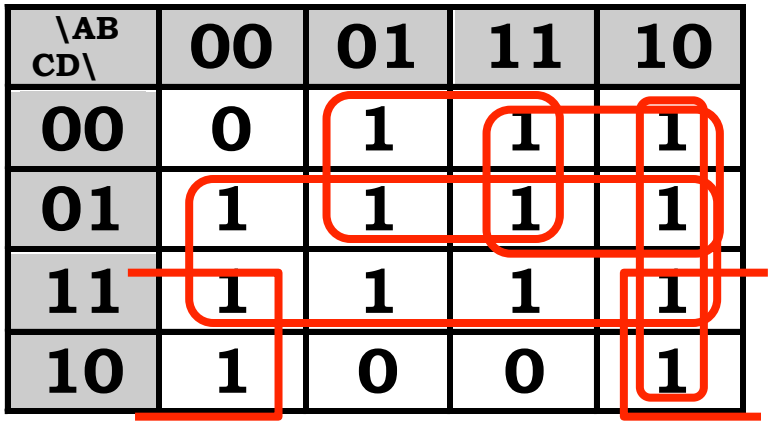


- can be uniquely identified by a single product term. The larger the implicant, the smaller the product term.

Finding Prime Implicants

We want to find all the prime implicants. The right strategy is a greedy one.

- Find the uncircled prime implicant with the greatest area
  - Order: 4x4 ⇒ 2x4 or 4x2 ⇒ 4x1 or 1x4 or 2x2 ⇒ 2x1 or 1x2 ⇒ 1x1
  - Overlap is okay
- Circle it
- Repeat until all prime implicants are circled



\AB CD\	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

Write Down Equations

Picking just enough prime implicants to cover all the 1's in the KMap, combine equations to form minimal sum-of-products.

C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0

$$Y = A\overline{C} + BC$$

\AB CD\	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

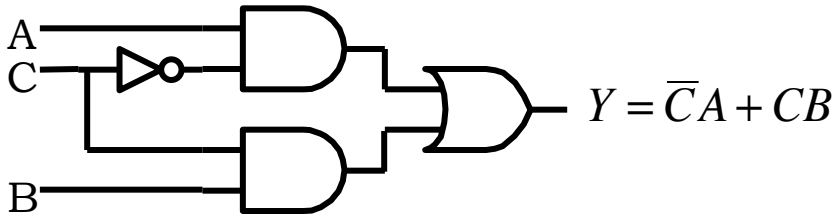
$$Y = D + B\overline{C} + A\overline{C} + \overline{B}C$$

\AB CD\	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

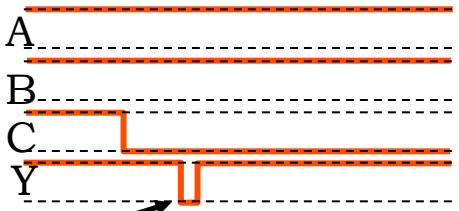
$$Y = D + B\overline{C} + A\overline{B} + \overline{B}C$$

Prime Implicants, Glitches & Leniency

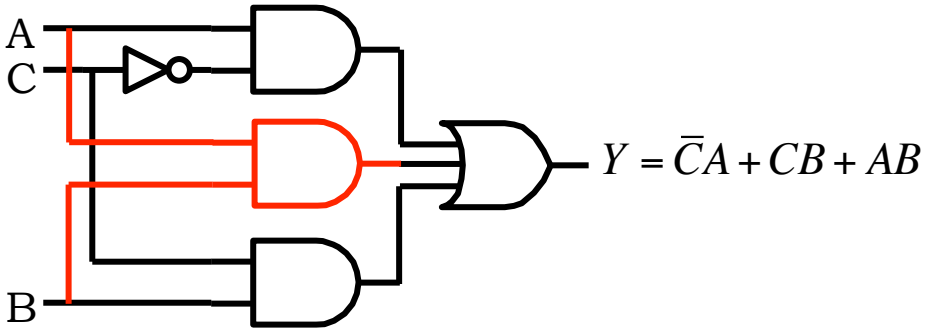
This circuit produces a glitch on Y when A=1, B=1, C: 1→0



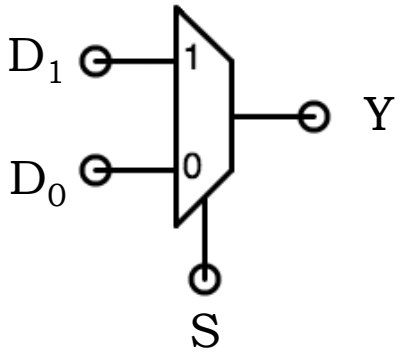
C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0



To make the circuit lenient, include product terms for ALL prime implicants.



We've Been Designing a Mux

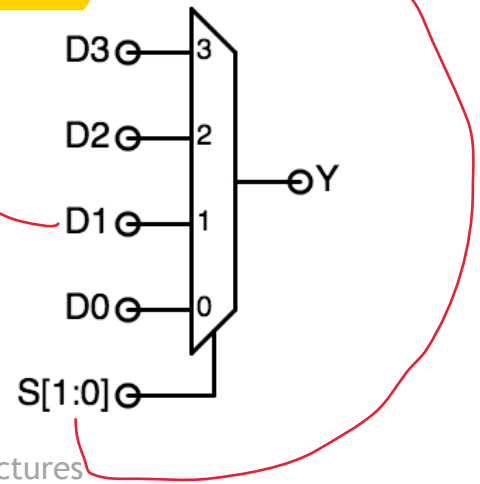


2-input Multiplexer

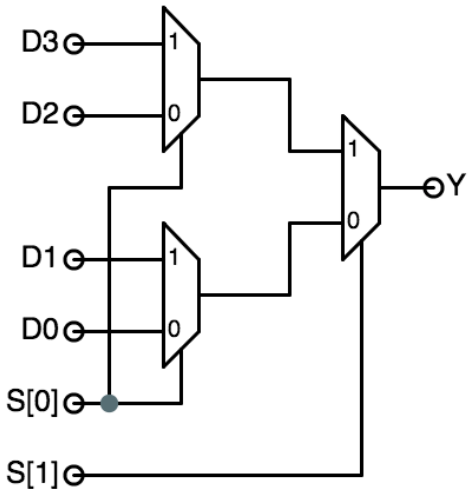
Truth Table

S	D_1	D_0	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

MUXes can be generalized to 2^k data inputs and k select inputs …



… and implemented as a tree of smaller MUXes:

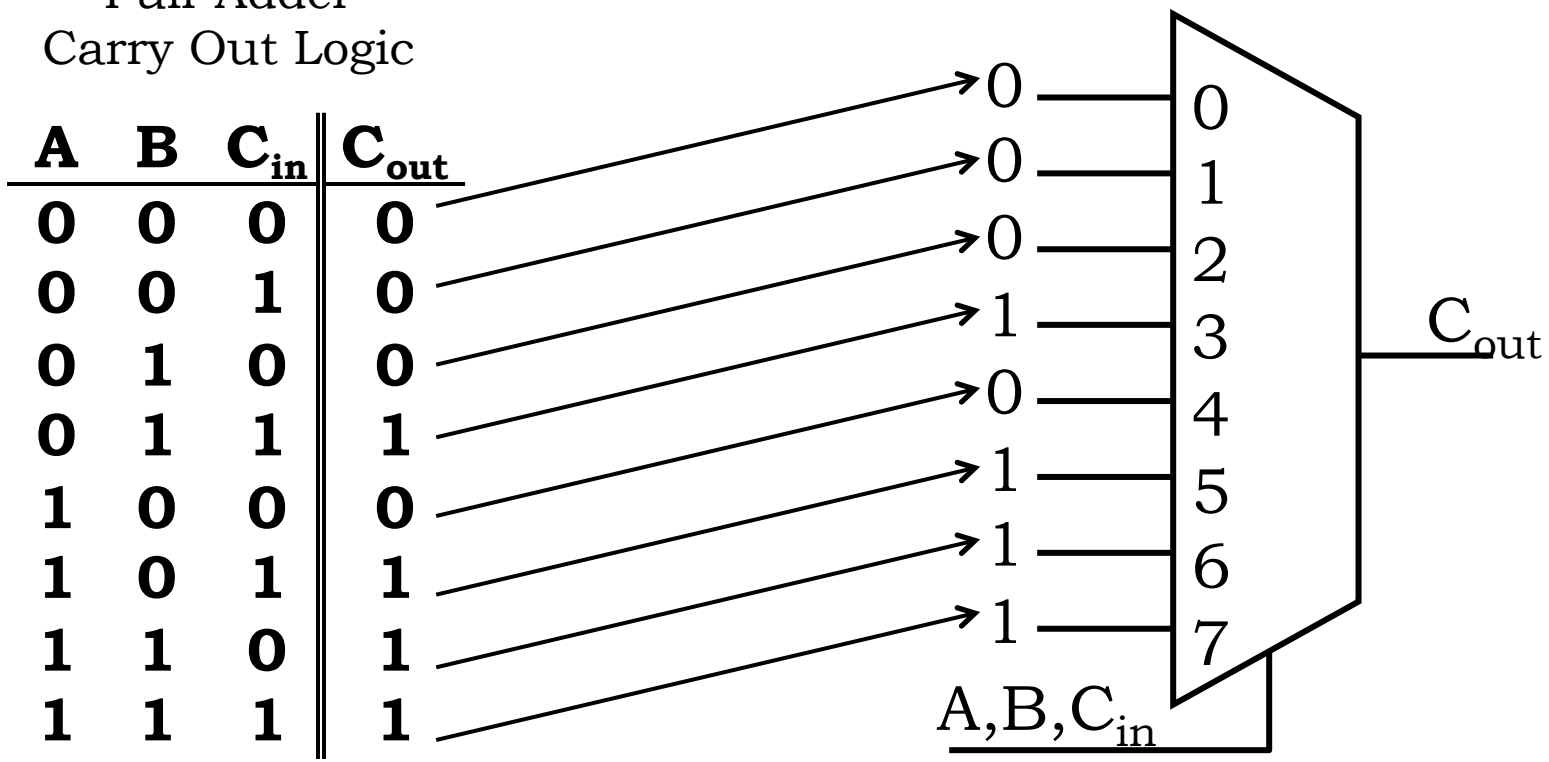


Systematic Implementation Strategies

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

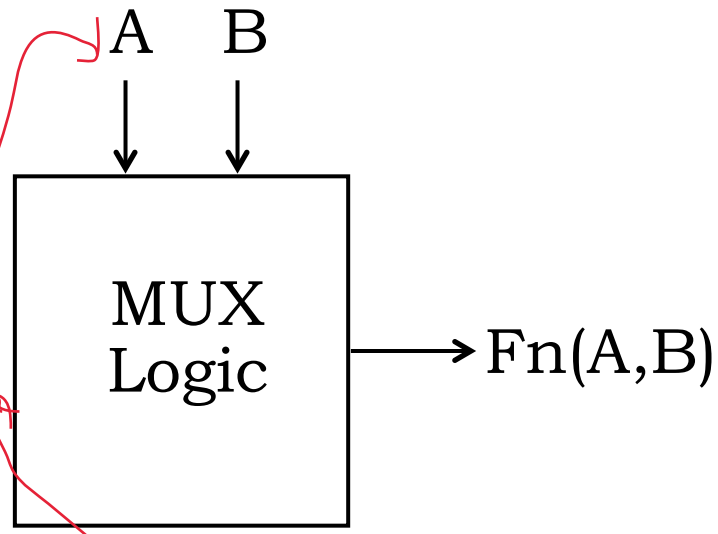
Full-Adder
Carry Out Logic

A	B	C_{in}	C_{out}
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



Synthesis By Table Lookup

AB	Fn(A,B)
00	0
01	1
10	1
11	0

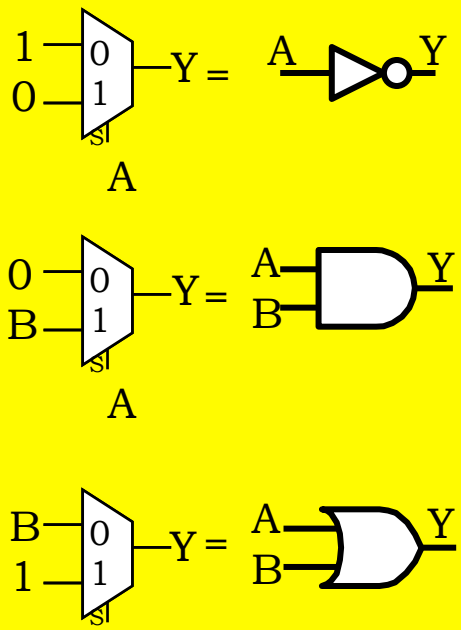


Generalizing:
In theory, we can build any 1-output combinational logic block with multiplexers.

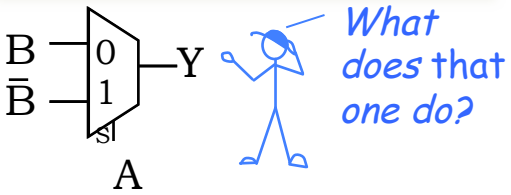
For an N-input function we need a 2^N input mux.

Is this practical for BIG truth tables?
How about 10-input function? 20-input?

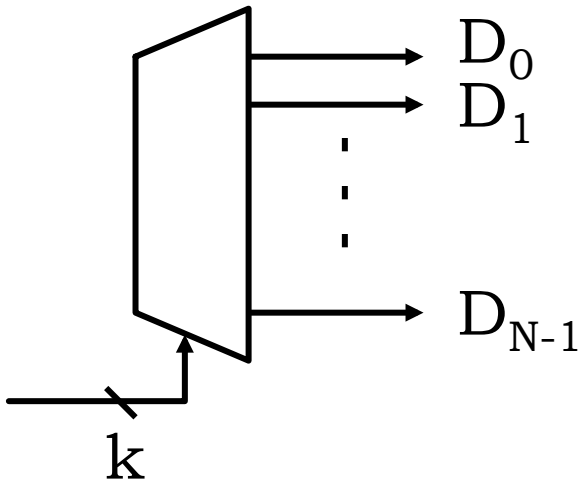
Muxes are universal!



In future technologies muxes might be the "natural gate".



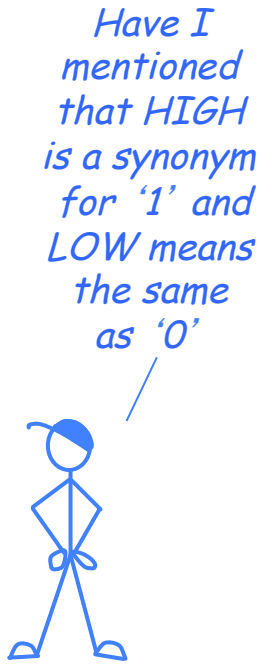
A New Combinational Device



DECODER:

- k SELECT inputs,
- $N = 2^k$ DATA OUTPUTs.

Select inputs choose one of the D_i to assert HIGH, all others will be LOW.

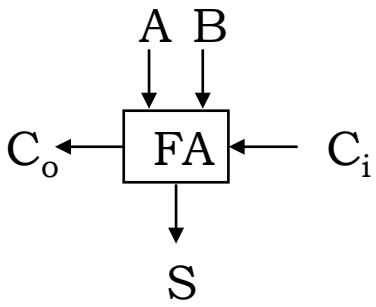


NOW, we are well on our way to building a general purpose table-lookup device.

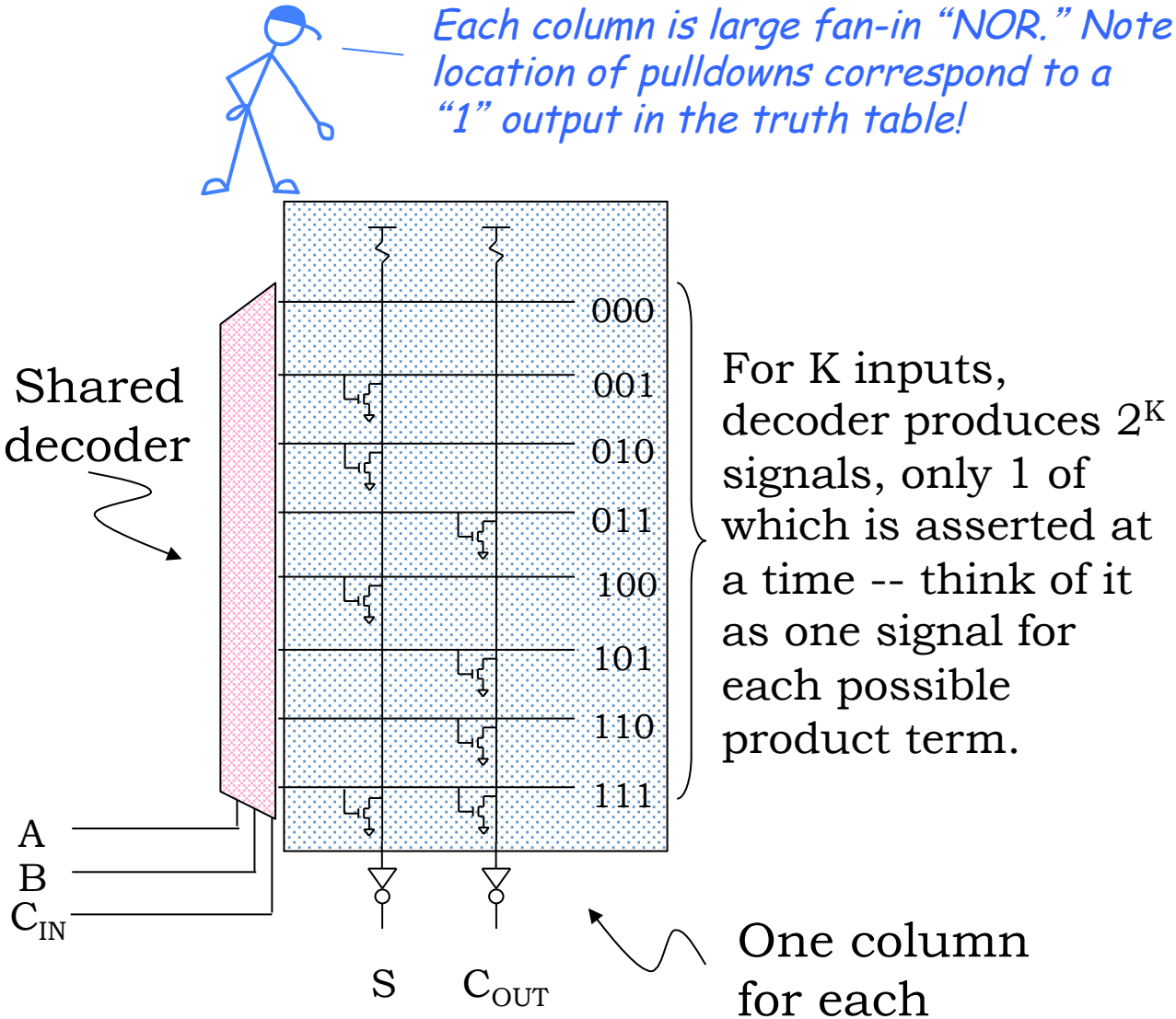
We can build a 2-dimensional ARRAY of decoders and selectors as follows …

Read-only Memory (ROM)

Full Adder

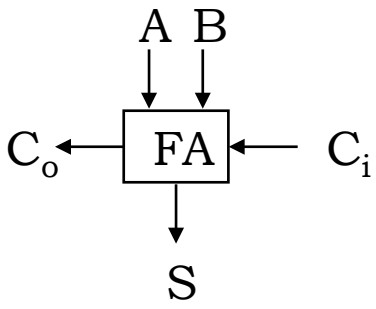


A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

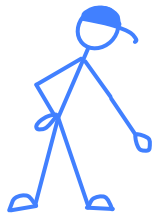


Read-only Memory (ROM)

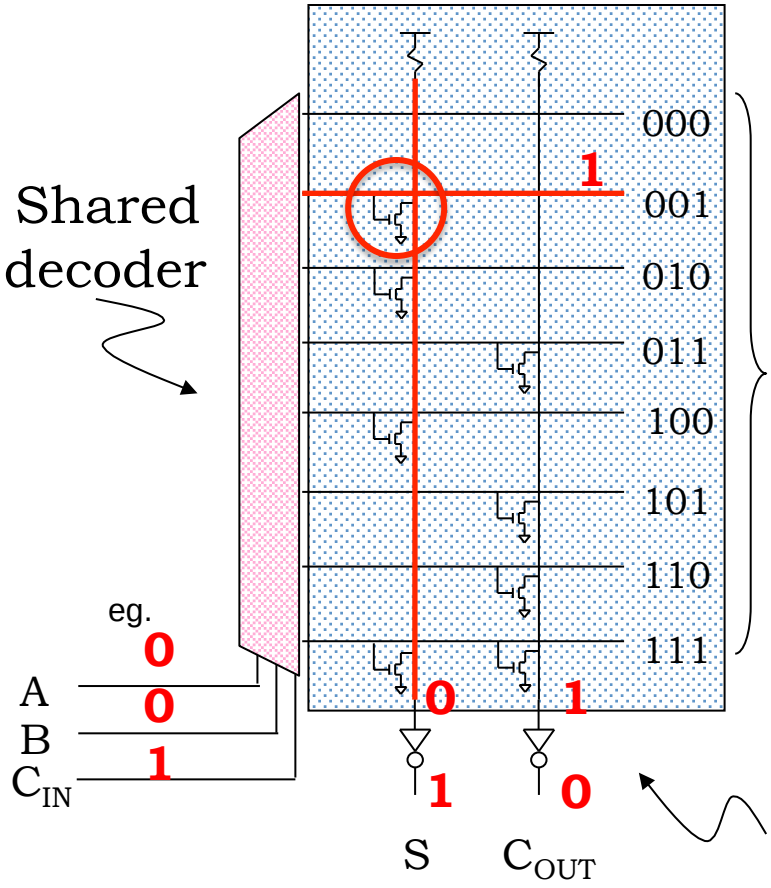
Full Adder



A	B	C_i	S	C_o
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



Each column is large fan-in "NOR." Note location of pulldowns correspond to a "1" output in the truth table!

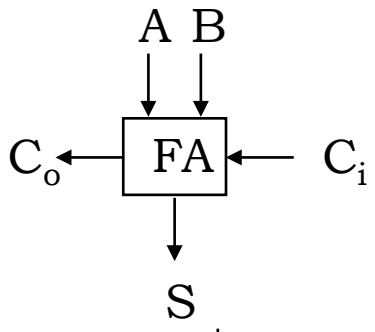


For K inputs, decoder produces 2^K signals, only 1 of which is asserted at a time -- think of it as one signal for each possible product term.

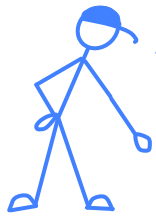
One column for each output

Read-only Memory (ROM)

Full Adder

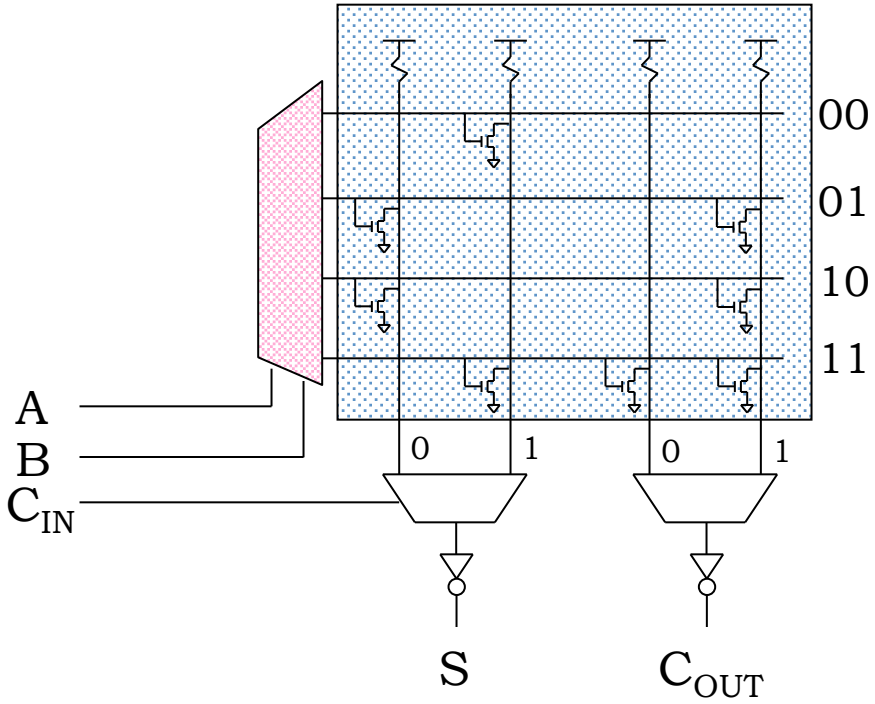


A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



LONG LINES slow down propagation times…

The best way to improve this is to build *square arrays*, using some inputs to drive output selectors (MUXes):



2D Addressing: Standard for ROMs, RAMs, logic arrays…

Logic According to ROMs

ROMs *ignore* the structure of combinational functions …

- Size, layout, and design are independent of function
- Any Truth table can be "programmed" by minor reconfiguration:

  - Metal layer (masked ROMs)
  - Fuses (Field-programmable PROMs)
  - Charge on floating gates (EPROMs)

  … etc.

ROMs tend to generate "glitchy" outputs. WHY?

Model: LOOK UP value of function in truth table…

Inputs: "ADDRESS" of a T.T. entry

ROM SIZE = # TT entries…

… for an N-input boolean function, size ≅ 2^N x #outputs

Summary

- Sum of products
  - Any function that can be specified by a truth table or, equivalently, in terms of AND/OR/NOT (Boolean expression)
  - "3-level" implementation of any logic function
    - Limitations on number of inputs (fan-in) increases depth
  - SOP implementation methods
    - NAND-NAND, NOR-NOR
- Muxes used to build table-lookup implementations
  - Easy to change implemented function -- just change constants
- ROMs
  - Decoder logic generates all possible product terms
  - Selector logic determines which terms are ORed together