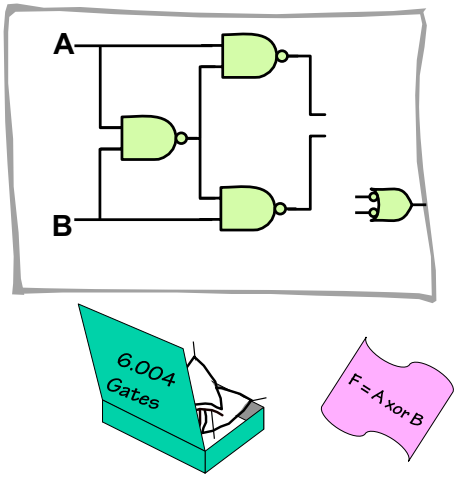
6.004 Computation Structures
Spring 2009

Synthesis of Combinational Logic



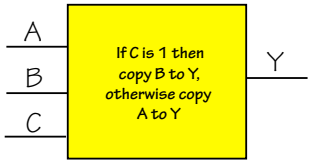
Lab 1 is due Thursday 2/19

Quiz 1 is a week from Friday (in section)

Functional Specifications

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

Argh… I'm tired of word games



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 in whose operations are **AND** (multiplication), **OR** (addition), and **inversion** (overbar).

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

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

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

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

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

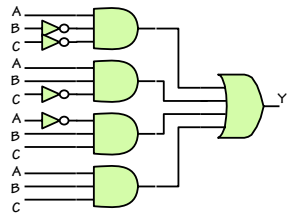
3) Wire up the gates, call it a day, and declare success!

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

Straightforward Synthesis

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

INVERTERS/AND/OR



Propagation delay --

No more than 3 gate delays

(assuming gates with an arbitrary number of inputs)

Basic Gate Repertoire

Are we sure we have all the gates we need?

Just how many two-input gates are there?

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

Hmmmm… all of these have 2-inputs (no surprise)

… each with 4 combinations, giving 2^2 output cases

How many ways are there of assigning 4 outputs? $2^{2^2} = 2^4 = 16$

There are only so many gates

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

There are only 16 possible 2-input gates

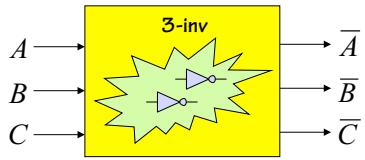
… some we know already, others are just silly

INPUT AB	ZERO	AND	A > B	A	B > A	B	XOR	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

CMOS gates are inverting; we can always respond positively to positive transitions by cascaded gates. But suppose our logic yielded cheap *positive* functions, while inverters were expensive…

Logic Geek Party Games

You have plenty of ANDs and ORs, but only 2 inverters. Can you invert more than 2 independent inputs?



CHALLENGE: Come up with a combinational circuit using ANDs, ORs, and at most 2 inverters that inverts A, B, and C !

Such a circuit exists. What does that mean?

- If we can invert 3 signals using 2 inverters, can we use 2 of the pseudo-inverters to invert 3 more signals?
- Do we need only 2 inverters to make ANY combinational circuit?

Hint: there's a subtle difference between our 3-inv device and three combinational inverters!

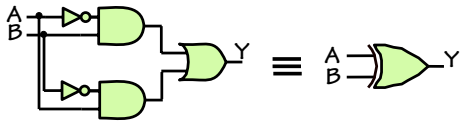
Is our 3-inv device *LENIENT*?

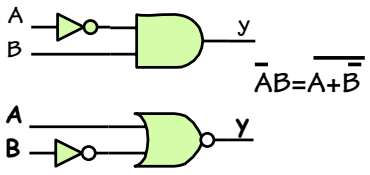
Fortunately, we can get by with a few basic gates…

AND, OR, and NOT are sufficient… (cf Boolean Expressions):

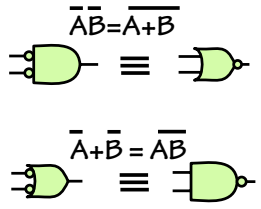
B>A	
AB	Y
00	0
01	1
10	0
11	0

XOR	
AB	Y
00	0
01	1
10	1
11	0





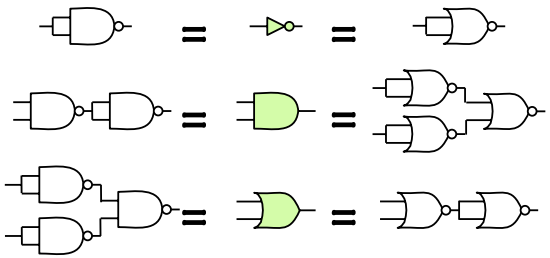
That is just DeMorgan's Theorem!



How many different gates do we *really* need?

One will do!

NANDs and NORs are universal:



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

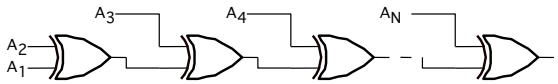
Stupid Gate Tricks

Suppose we have some 2-input XOR gates:

$t_{pd} = 1$
$t_{cd} = 0$

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

And we want an N-input XOR:



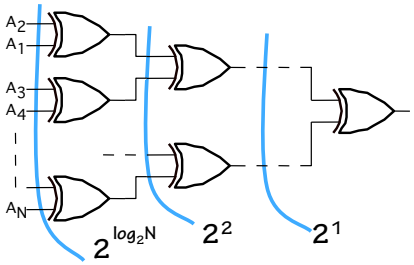
output = 1 iff number of 1s input is ODD ("ODD PARITY")

$t_{pd} = O(\underline{N})$ -- WORST CASE.

Can we compute N-input XOR faster?

I think that I shall never see

a circuit lovely as...



N-input TREE has $O(\underline{\log N})$ levels...

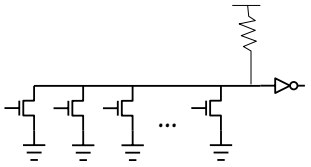
Signal propagation takes $O(\underline{\log N})$ gate delays.

Question: Can EVERY N-Input Boolean function be implemented as a tree of 2-input gates?

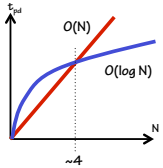
Are Trees Always Best?

Alternate Plan: Large Fan-in gates

- N pulldowns with complementary pullups
- Output HIGH if any input is HIGH = "OR"



- Propagation delay: O(N) since each additional MOSFET adds C

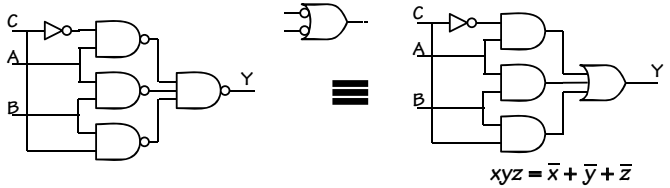


Don't be mislead by the "big O" stuff… the constants in this case can be much smaller… so for small N this plan might be the best.

Practical SOP Implementation

NAND-NAND

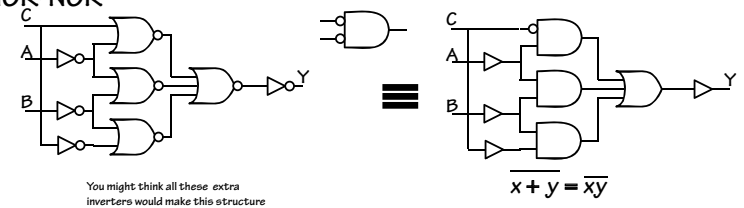
$\overline{AB}=\overline{A}+\overline{B}$ "Pushing Bubbles"



$xyz = \overline{x} + \overline{y} + \overline{z}$

NOR-NOR

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



$\overline{x + y} = \overline{x}\,\overline{y}$

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:	$a+ab=a,\quad a+\overline{a}b=a+b$
	$a(a+b)=a,\quad a(\overline{a}+b)=ab$
Reduction:	$ab+\overline{a}b=b,\ (a+b)(\overline{a}+b)=b$
DeMorgan's Law:	$\overline{a}+\overline{b}=\overline{ab},\quad \overline{a}\,\overline{b}=\overline{a+b}$

Boolean Minimization:

An Algebraic Approach

Can't he come up with a *new* example???

Lets (again!) simplify

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

Using the identity

$$\alpha A + \alpha\overline{A} = \alpha$$

For any expression α and variable A:

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

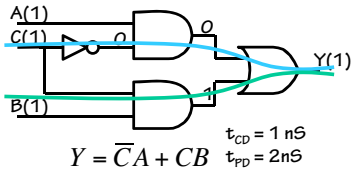
$$Y = \overline{C}\,\overline{B}A + C\overline{B} + \overline{C}\overline{B}A$$

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

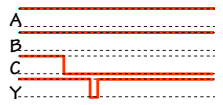
Hey, I could write A *program* to do That!

A Case for Non-Minimal SOP

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

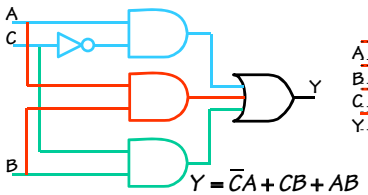


$Y = \overline{C}A + CB$ t_{CD} = 1 nS t_{PD} = 2nS

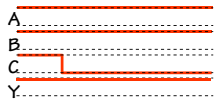


That's what we call a "glitch" or "hazard"

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



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

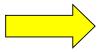


Now it's LENIENT!

Truth Tables with "Don't Cares"

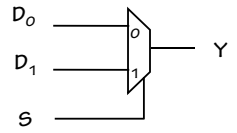
One way to reveal the opportunities for a more compact implementation is to rewrite the truth table using "don't cares" (--) to indicate when the value of a particular input is irrelevant in determining the value of the output.

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	--	0	0	
0	--	1	1	$\rightarrow \overline{C}A$
1	0	--	0	
1	1	--	1	$\rightarrow CB$
--	0	0	0	
--	1	1	1	$\rightarrow BA$

We've been designing a "mux"

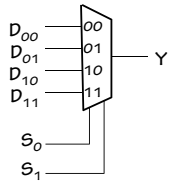


2-input Multiplexer

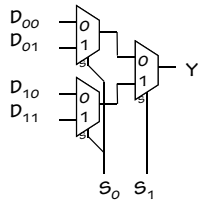
Truth Table

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

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



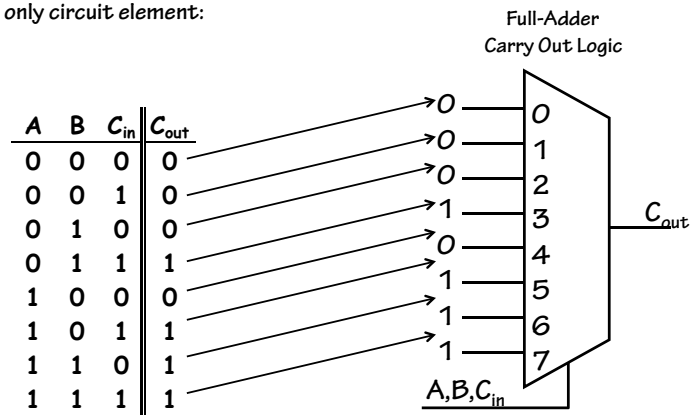
… and implemented as a tree of smaller MUXes:



Systematic Implementations of Combinational Logic

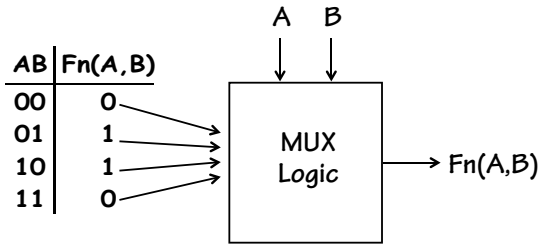
Consider implementation of some arbitrary Boolean function, F(A,B,C) … using a MULTIPLEXER as the only circuit element:

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



General Table Lookup Synthesis

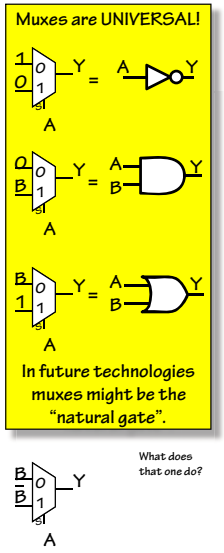
AB	Fn(A,B)
00	0
01	1
10	1
11	0



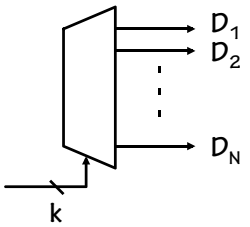
Generalizing:
In theory, we can build any 1-output combinational logic block with multiplexers.

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

BIG Multiplexers?
How about 10-input function? 20-input?



A New Combinational Device



DECODER:

k SELECT inputs,

N = 2^k DATA OUTPUTs.

Selected D_j HIGH;
all others LOW.

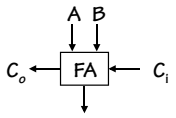
Have I mentioned that HIGH is a synonym for '1' and LOW means the same as '0'

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 ...

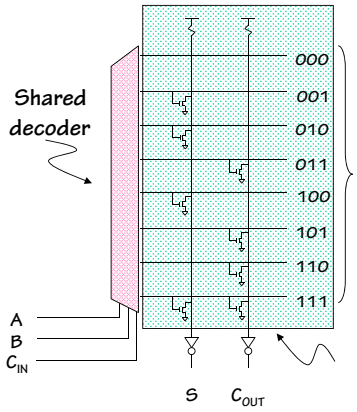
Read-only memories (ROMs)

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 "OR" as described on slide #12. Note location of pulldowns correspond to a "1" output in the truth table!

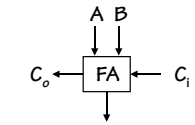


For K inputs, decoder produces 2^K signals, only 1 of which is asserted at a time -- think of it as one signal for each possible product term.

One selector for each output

Read-only memories (ROMs)

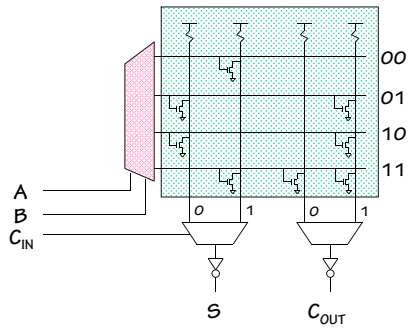
Full Adder



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 = $\frac{2^N \times \#\text{outputs}}{}$

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 p'terms are or'ed together