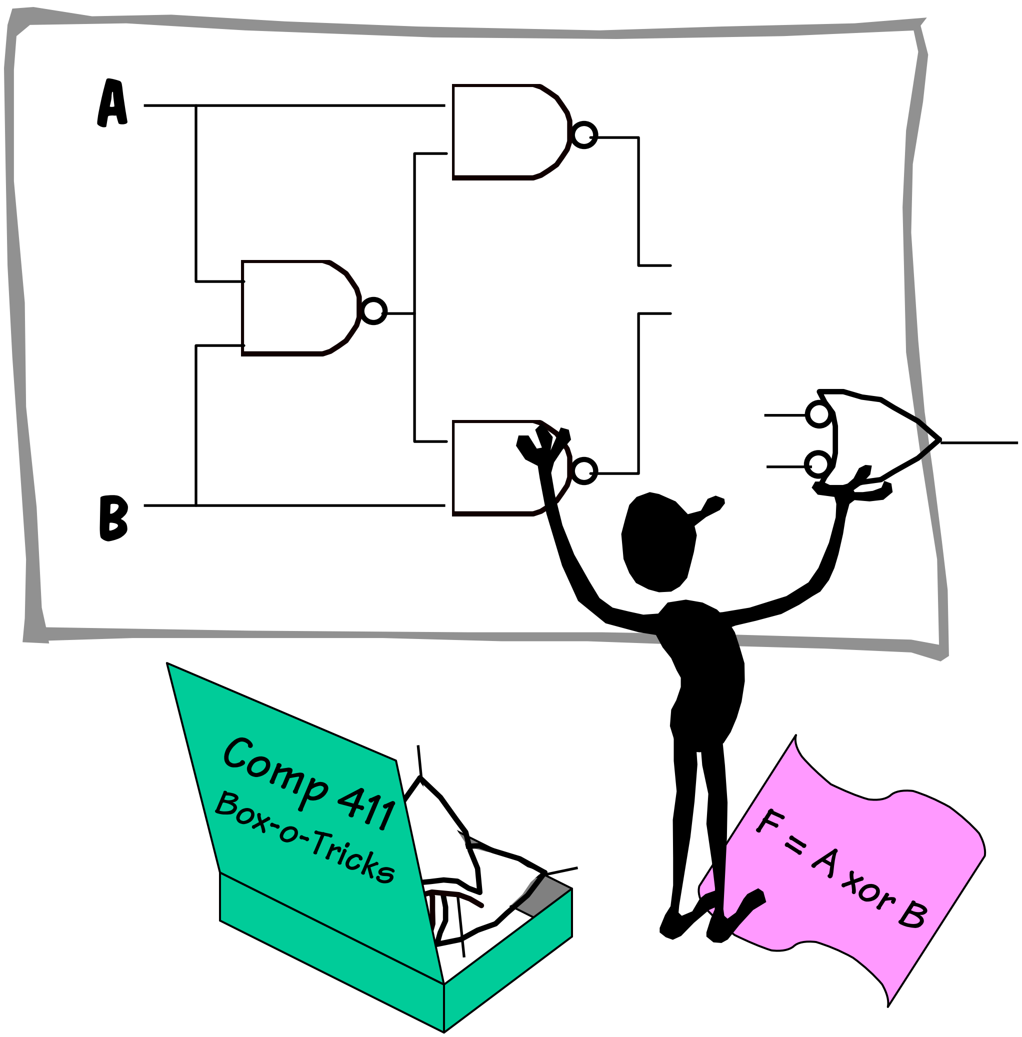
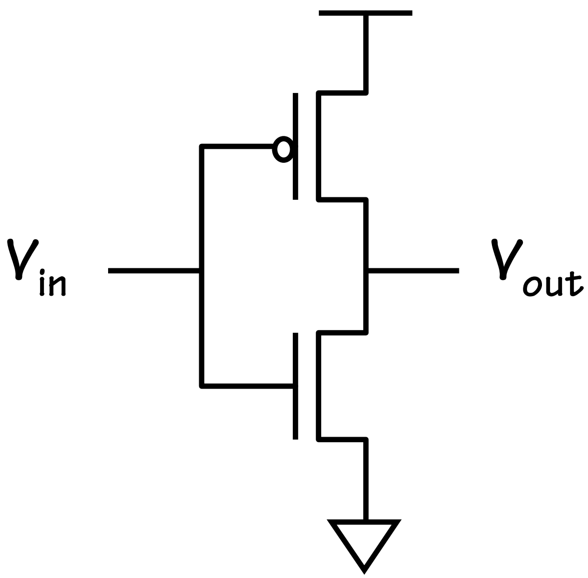
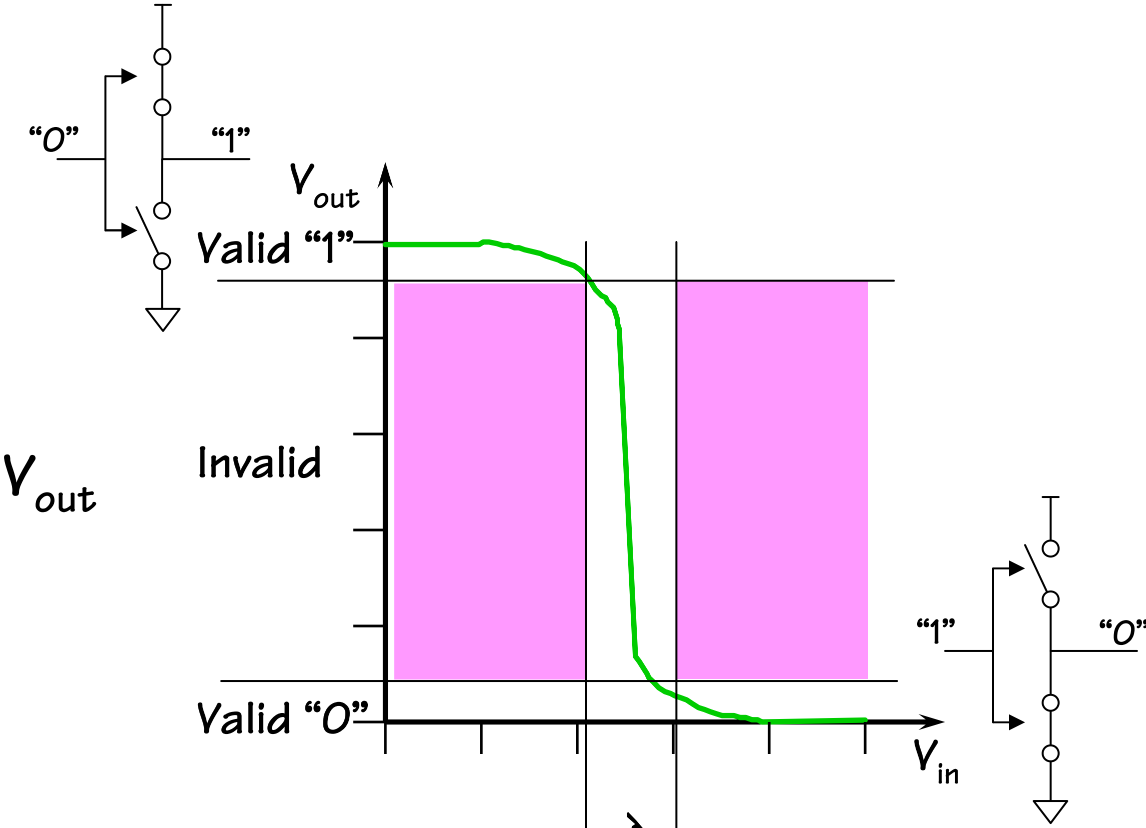
Transistors, Logic, and Math

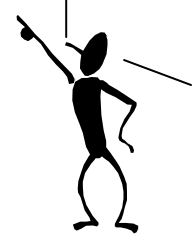


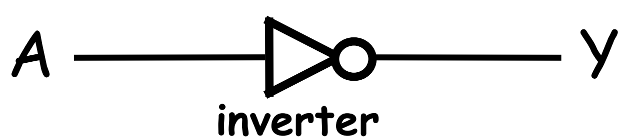
1) The digital contract
2) Encoding bits with voltages
3) Processing bits with transistors
4) Gates
5) Truth-tables
6) Multiplexer Logic

CMOS Inverter



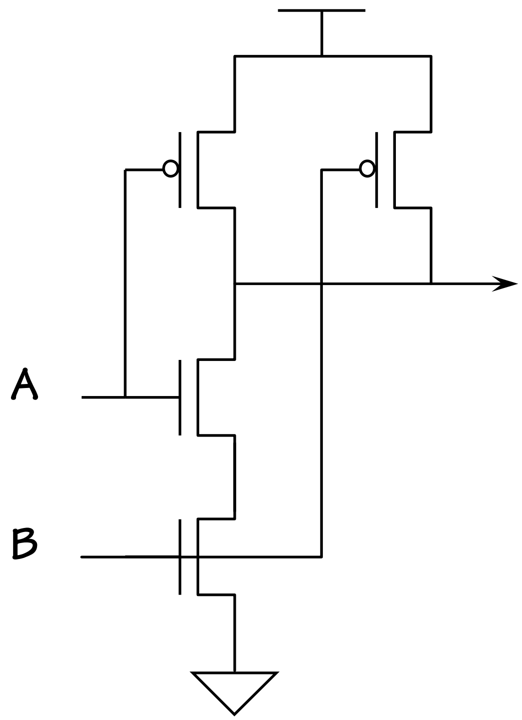






only a narrow range of input voltages result in "invalid" output values. (this diagram is greatly exaggerated)

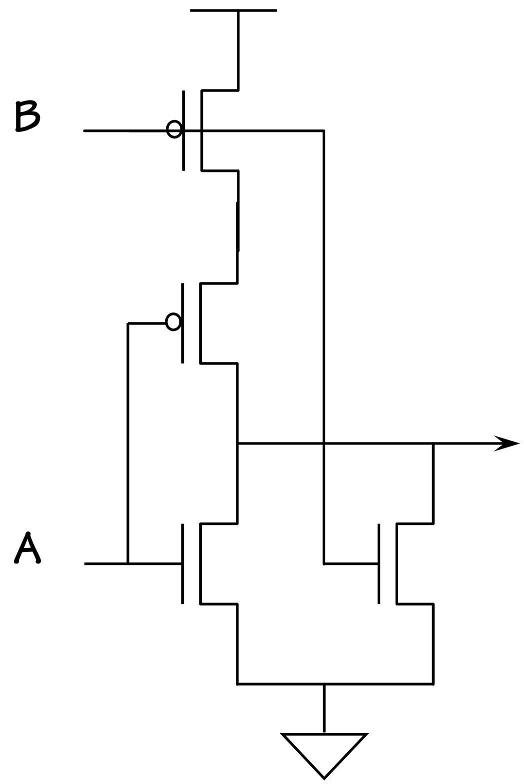
A Two Input Logic Gate



What function does this gate compute?

A	B	C
0	0	
0	1	
1	0	
1	1	

Here's Another…



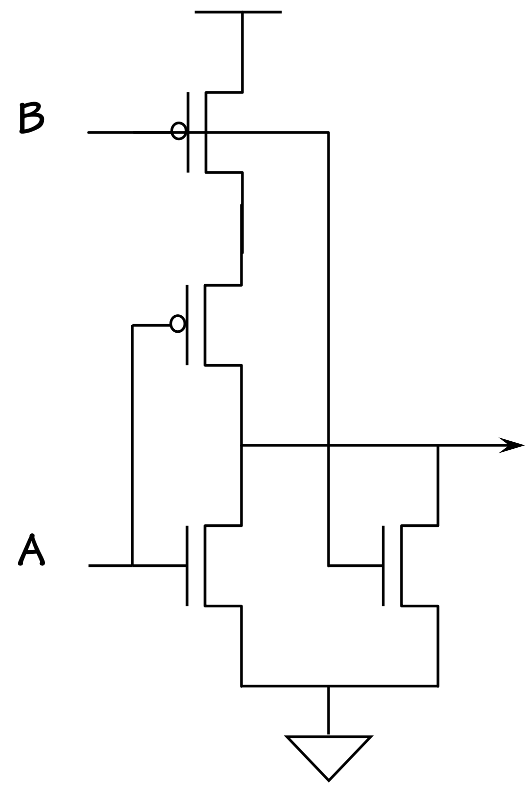
What function does this gate compute?

A	B	C
0	0	
0	1	
1	0	
1	1	

CMOS Gates Like to Invert

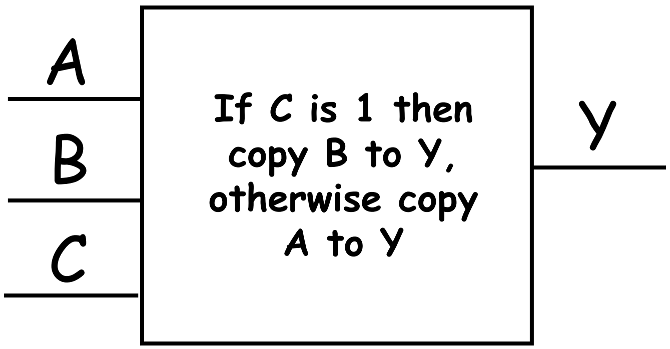
OBSERVATION: CMOS gates tend to be inverting!

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?



Now We're Ready to Design Stuff!

We need to start somewhere -- usually it's the functional specification



If you are like most engineers you'd rather see a table, or formula than parse a logic puzzle. The fact is, **any combinational function can be expressed as a table**.

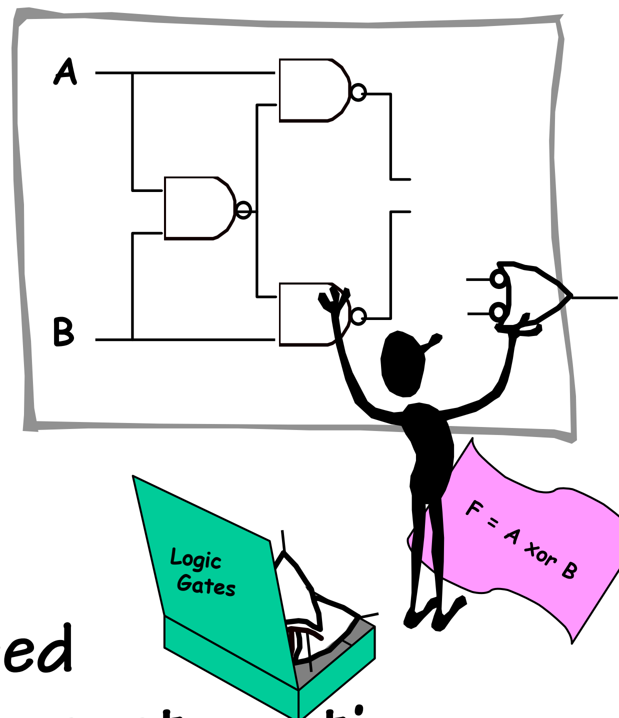
These "**truth tables**" are a concise description of the combinational system's function. Conversely, any computation performed by a combinational system can expressed as a truth table.

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

Where Do We Start?

We have a bag of gates.



We want to build a computer.

What do we do?

We need

… a systematic approach for designing logic

A Slight Diversion

Are we sure we have all the gates we need?

How many two-input gates are there?

AND AB	Y	OR AB	Y	NAND AB	Y	NOR 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

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

… 2 inputs have 4 permutations, giving 2^2 output cases

How many permutations of 4 outputs are there? 2^4

Generalizing, there are 2^{2^N}, N-input gates!

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~~

Do we need all of these gates?

Nope. After all, we describe them all using AND, OR, and NOT.

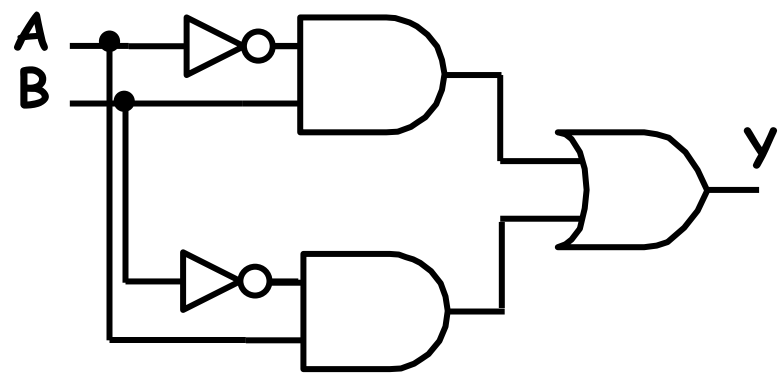
We Can Make Most Gates Out of Others

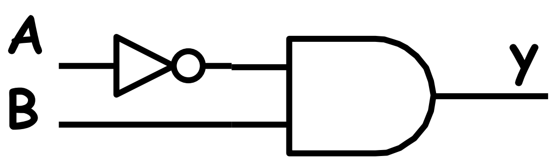
B>A

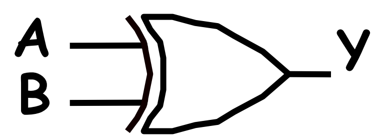
AB	Y
00	0
01	1
10	0
11	0

XOR

AB	Y
00	0
01	1
10	1
11	0



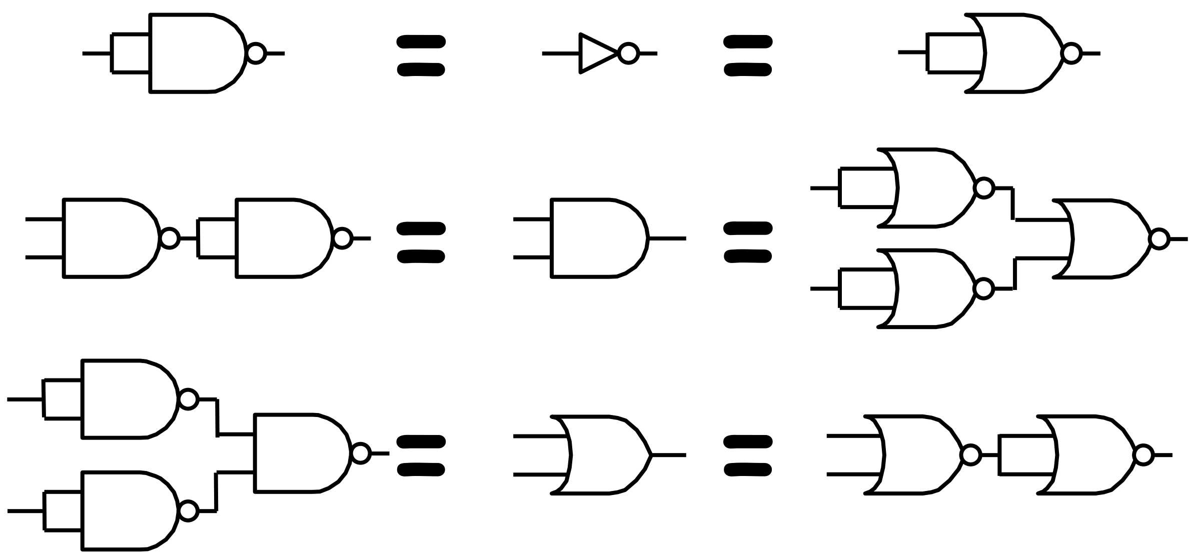




How many different gates do we really need?

One Will Do!

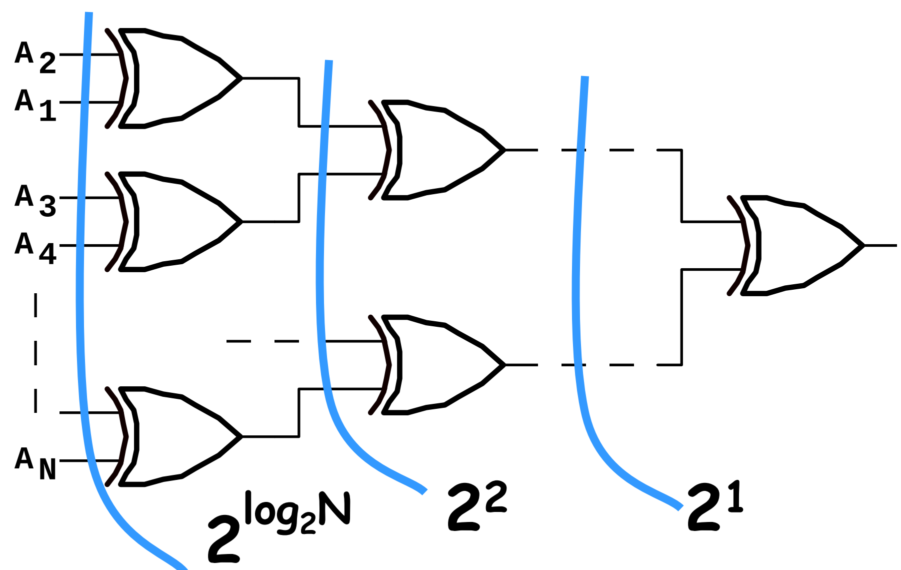
NANDs and NORs are universal



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

I Think That I Shall Never See

a Gate Lovely as a ...



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

Signal propagation takes O(log N) gate delays.

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 for every '1' in the output

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

3) Wire up the gates, call it a day, and go home!

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

SUM-OF-PRODUCTS

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

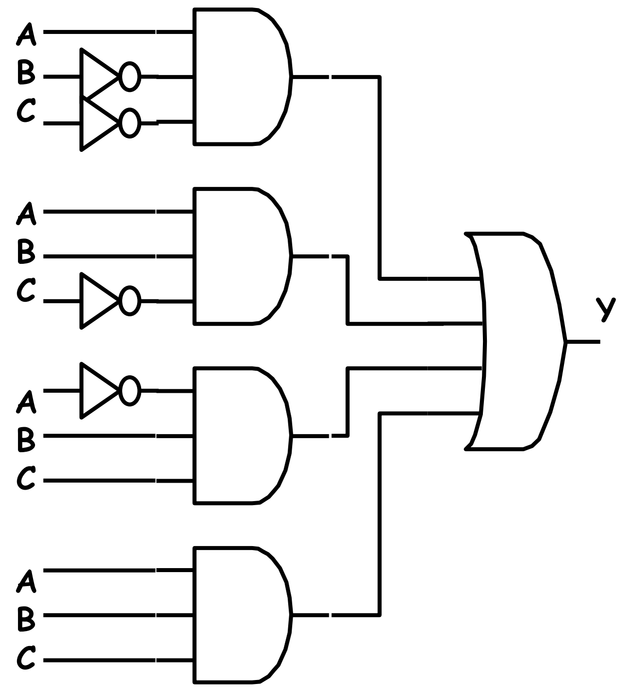
Straightforward Synthesis

We can implement

SUM-OF-PRODUCTS

with just three levels of logic.

INVERTERS/AND/OR

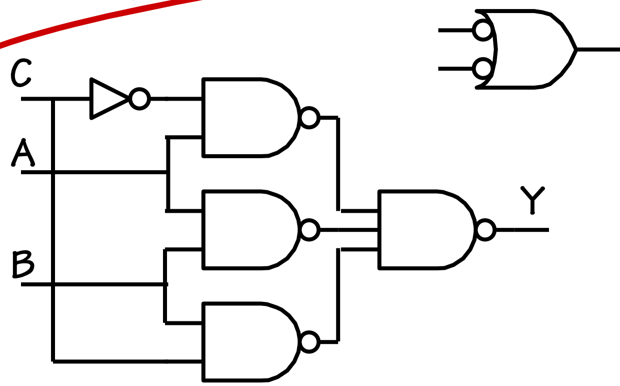


Useful Gate Structures

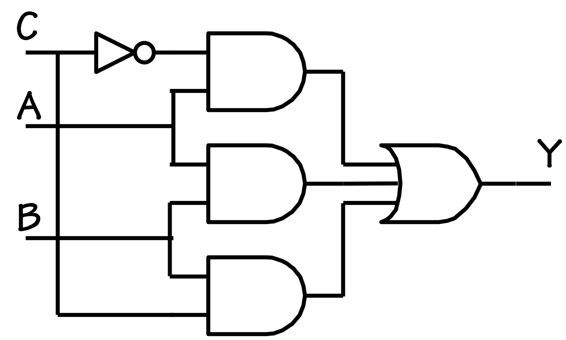
$\overline{AB} = \overline{A} + \overline{B}$

"Pushing Bubbles"

NAND-NAND



≡

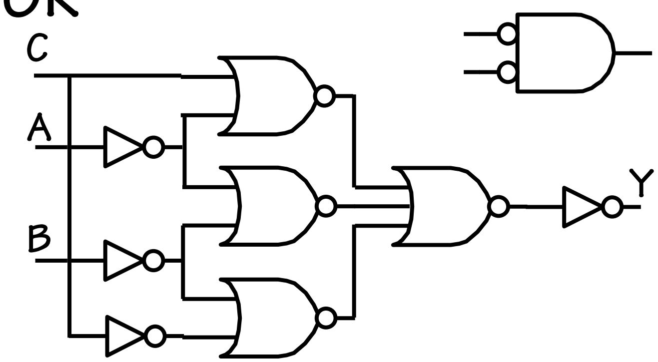


$\overline{xyz} = \overline{x} + \overline{y} + \overline{z}$

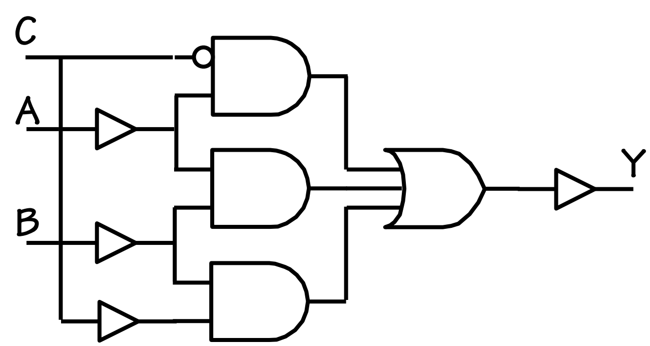
DeMorgan's Laws

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

NOR-NOR



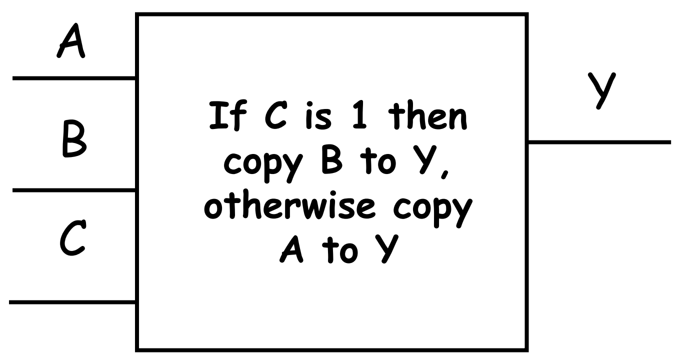
≡



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

An Interesting 3-Input Gate

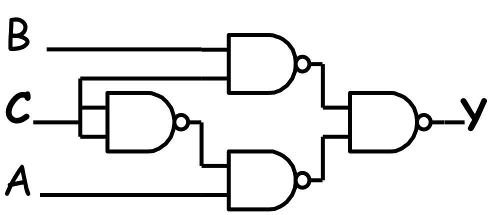
Based on C, select the A or B input to be copied to the output Y.



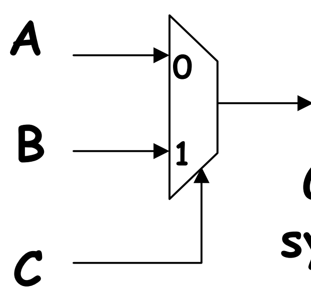
Truth Table

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



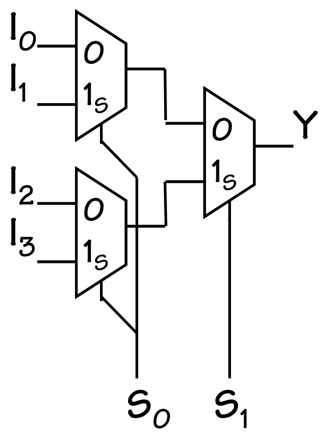
schematic

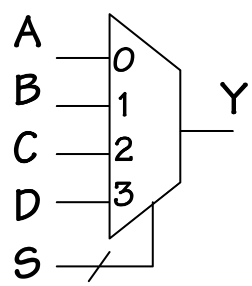


Gate symbol

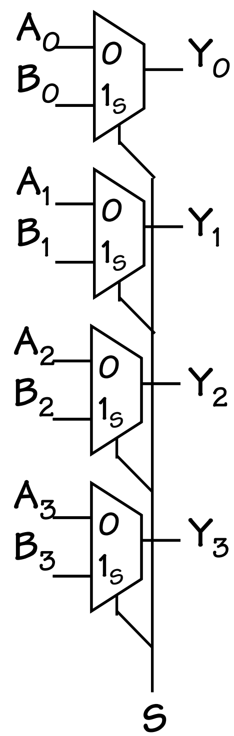
MUX Shortcuts

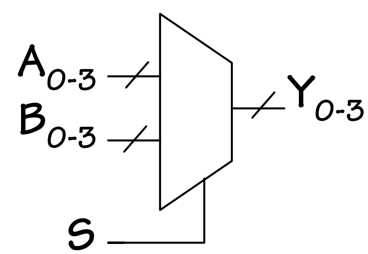
A 4-input Mux (implemented as a tree)





A 4-bit wide Mux





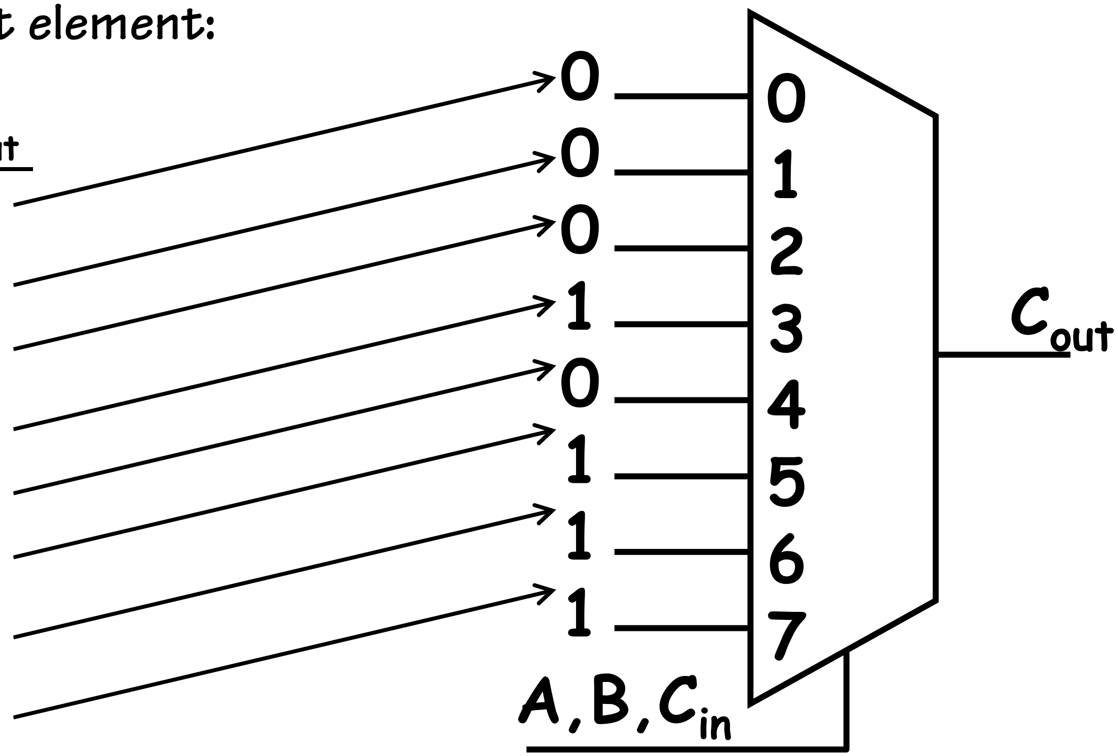
Mux Logic Synthesis

Consider implementation of some arbitrary Boolean function, F(A,B)

... using a MULTIPLEXER as the only circuit element:

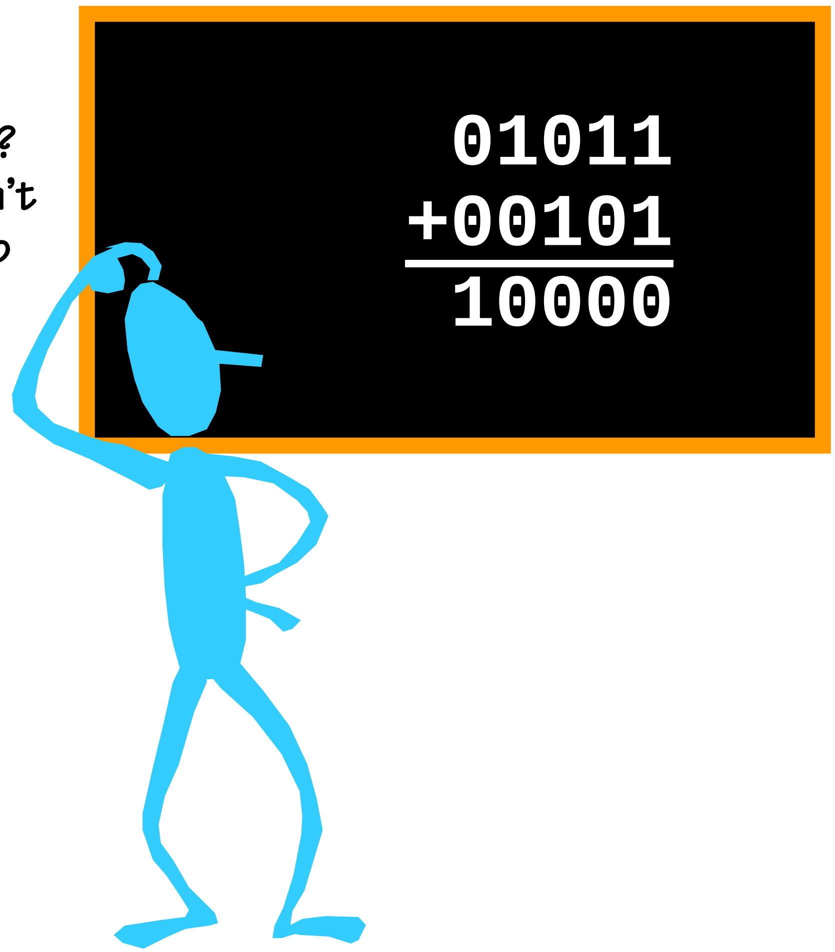
A	B	C_{in}	C_{out}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Full-Adder
Carry Out Logic



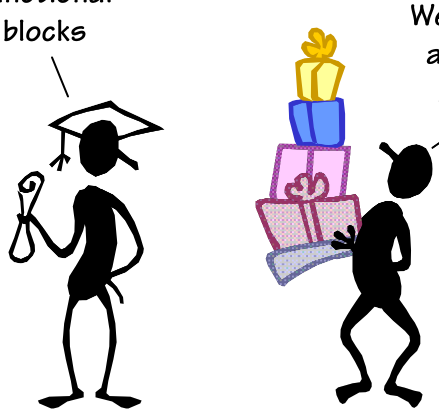
Arithmetic Circuits

Didn't I learn how to do addition in the second grade? UNC courses aren't what they used to be...

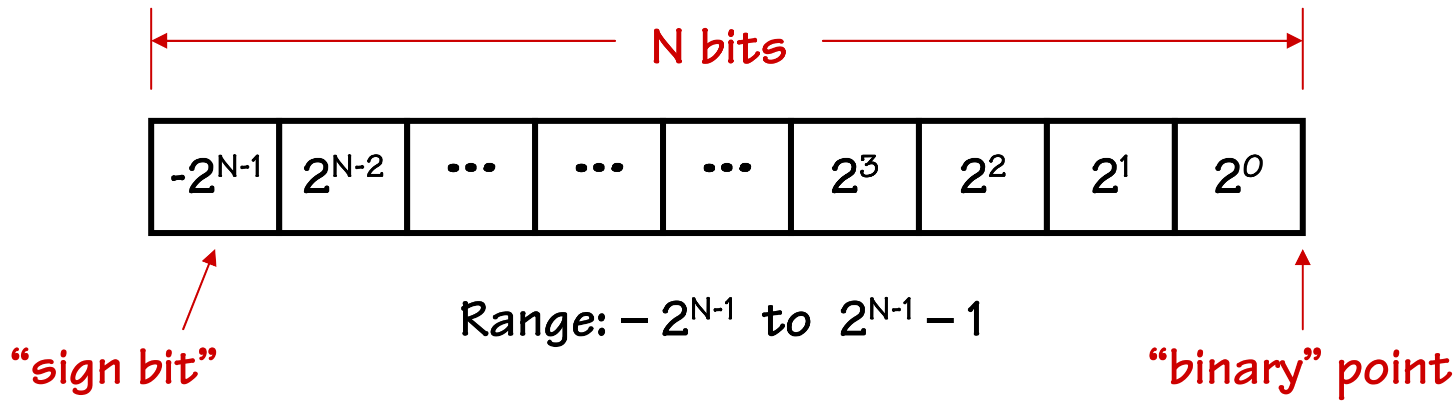


Finally; time to build some serious functional blocks

We'll need a lot of boxes



Review: 2's Complement



Range: -2^{N-1} to $2^{N-1}-1$

8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

If we use a two's-complement representation for signed integers, the same binary addition procedure will work for adding both signed and unsigned numbers.

By moving the implicit "binary" point, we can represent fractions too:

$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

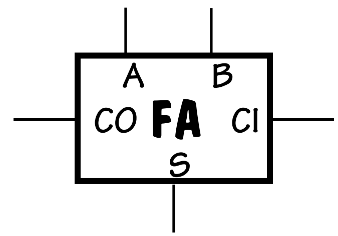
Binary Addition

Here's an example of binary addition as one might do it by "hand":

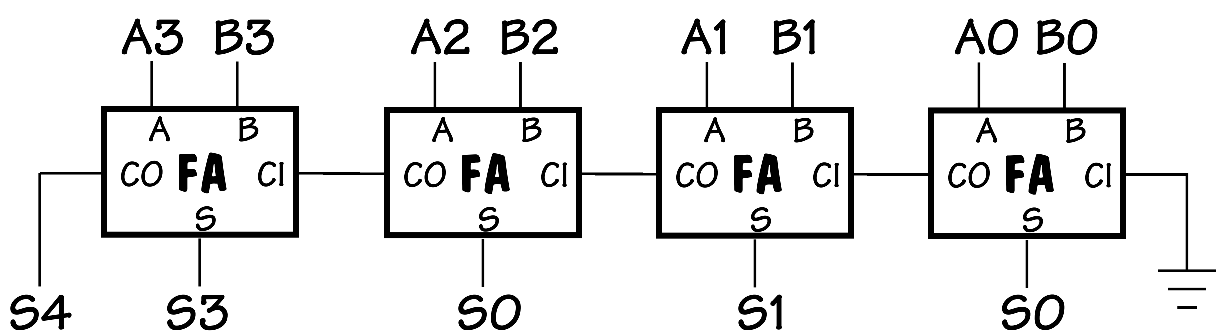
```
          1 1  0 1    <- Carries from previous column
A:      1101
B:  +   0101
    --------
       10010
```

Adding two N-bit numbers produces an (N+1)-bit result

Let's start by building a block that adds one column:



Then we can cascade them to add two numbers of any size…



Designing a Full Adder: From Last Time

1) Start with a truth table:

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

2) Write down eqns for the "1" outputs

$$C_o = \overline{C_i}AB + C_i\overline{A}B + C_iA\overline{B} + C_iAB$$
$$S = \overline{C_i}\,\overline{A}B + \overline{C_i}A\overline{B} + C_i\overline{A}\,\overline{B} + C_iAB$$

3) Simplifing a bit

$$C_o = C_i(A + B) + AB$$
$$S = C_i \oplus A \oplus B$$

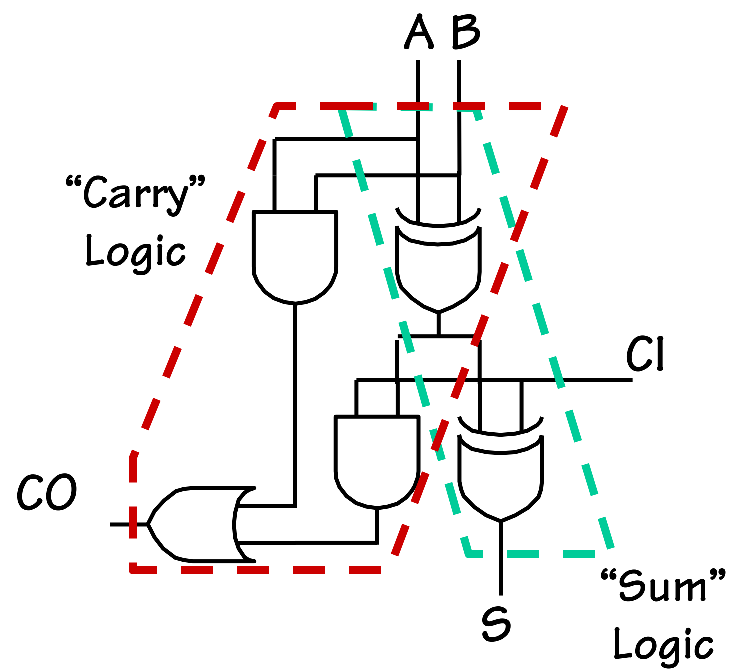
$$C_o = C_i(A \oplus B) + AB$$
$$S = C_i \oplus (A \oplus B)$$

For Those Who Prefer Logic Diagrams …

$C_o = C_i(A \oplus B) + AB$

$S = C_i \oplus (A \oplus B)$

- A little tricky, but only 5 gates/bit

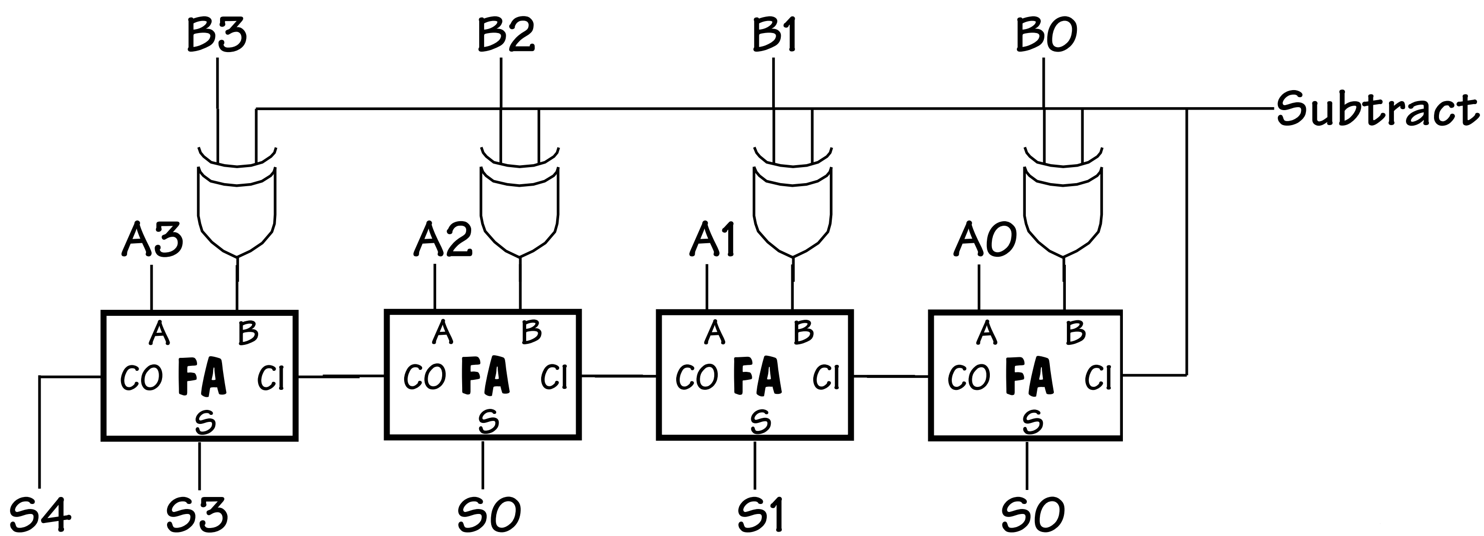


Subtraction: A-B = A + (-B)

Using 2's complement representation: $-B = \sim B + 1$

~ = bit-wise complement

So let's build an arithmetic unit that does both addition and subtraction. Operation selected by control input:



Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 — *big NOR gate*

N (negative): result is < 0 — S_{N-1}

C (carry): indicates that add in the most significant position produced a carry, e.g., "1 + (-1)" — *from last FA*

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., "$(2^{i-1} - 1) + (2^{i-1} - 1)$"

$$V = A_{i-1} B_{i-1} \overline{N} + \overline{A}_{i-1} \overline{B}_{i-1} N$$

-or-

$$V = CO_{i-1} \oplus CI_{i-1}$$

To compare A and B, perform A–B and use condition codes:

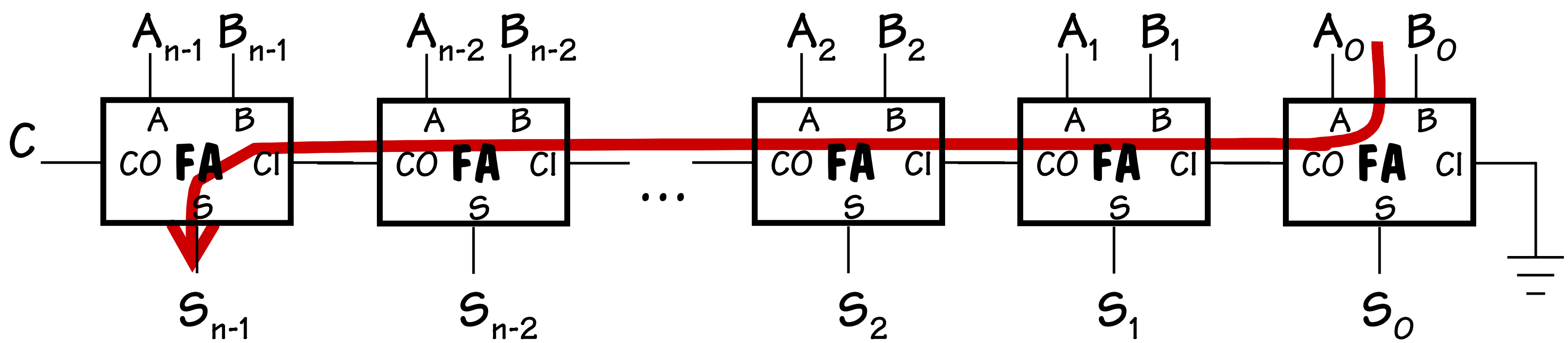
Signed comparison:

LT	N⊕V
LE	Z+(N⊕V)
EQ	Z
NE	~Z
GE	~(N⊕V)
GT	~(Z+(N⊕V))

Unsigned comparison:

LTU	C
LEU	C+Z
GEU	~C
GTU	~(C+Z)

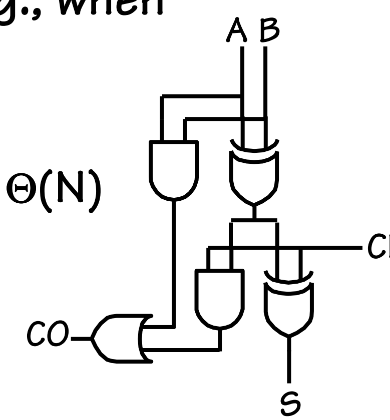
T_{PD} of Ripple-Carry Adder



Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = (t_{PD,XOR} + t_{PD,AND} + t_{PD,OR}) + (N-2)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR} \approx \Theta(N)$$

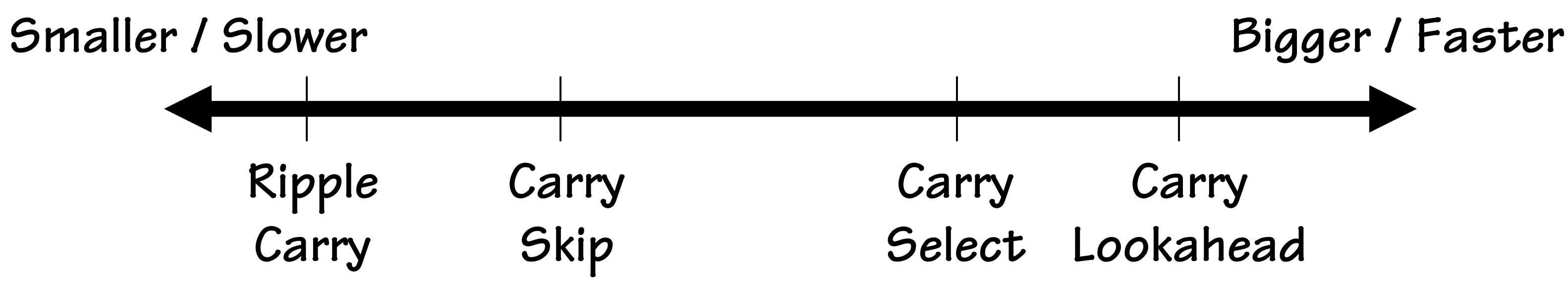
A,B to CO — CI to CO — CI_{N-1} to S_{N-1}



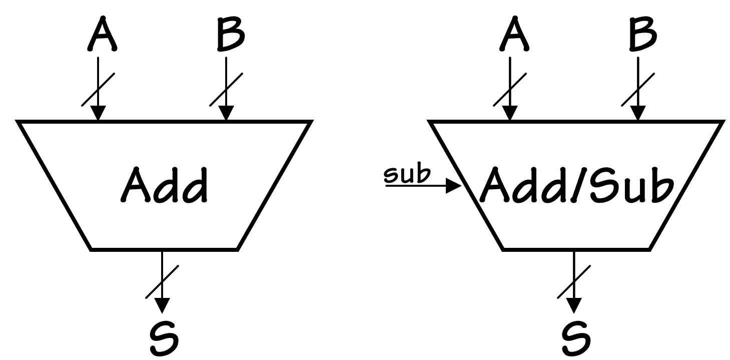
$\Theta(N)$ is read "order N" and tells us that the latency of our adder grows in proportion to the number of bits in the operands.

Adder Summary

Adding is not only a common, but it is also tends to be one of the most time-critical of operations. As a result, a wide range of adder architectures have been developed that allow a designer to tradeoff complexity (in terms of the number of gates) for performance.



At this point we'll define a high-level functional unit for an adder, and specify the details of the implementation as necessary.



Shifting Logic

Shifting is a common operation that is applied to groups of bits. Shifting can be used for alignment, as well as for arithmetic operations.

X << 1 is approx the same as 2*X

X >> 1 can be the same as X/2

For example:

$$X = 20_{10} = 00010100_2$$

Left Shift:

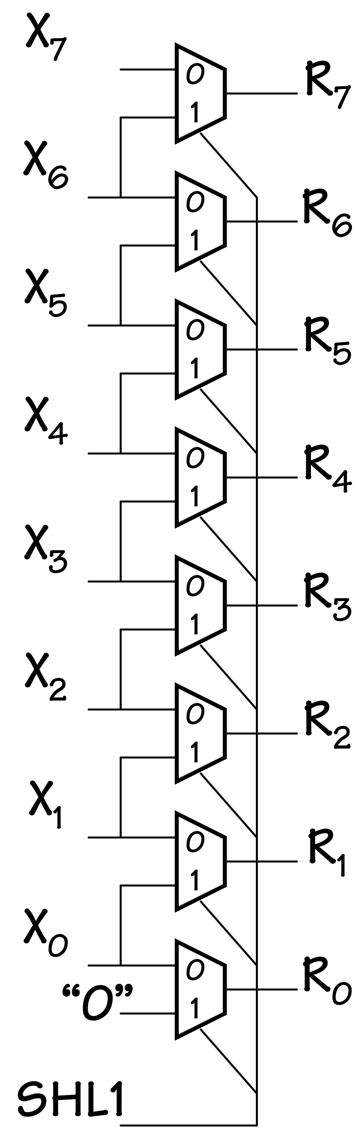
$(X << 1) = 00101000_2 = 40_{10}$

Right Shift:

$(X >> 1) = 00001010_2 = 10_{10}$

Signed or "Arithmetic" Right Shift:

$(-X >> 1) = (11101100_2 >> 1) = 11110110_2 = -10_{10}$



Boolean Operations

It will also be useful to perform logical operations on groups of bits.
Which ones?

ANDing is useful for "masking" off groups of bits.
ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

ANDing is also useful for "clearing" groups of bits.
ex. 10101110 & 00001111 = 00001110 (0's clear first 4 bits)

ORing is useful for "setting" groups of bits.
ex. 10101110 | 00001111 = 10101111 (1's set last 4 bits)

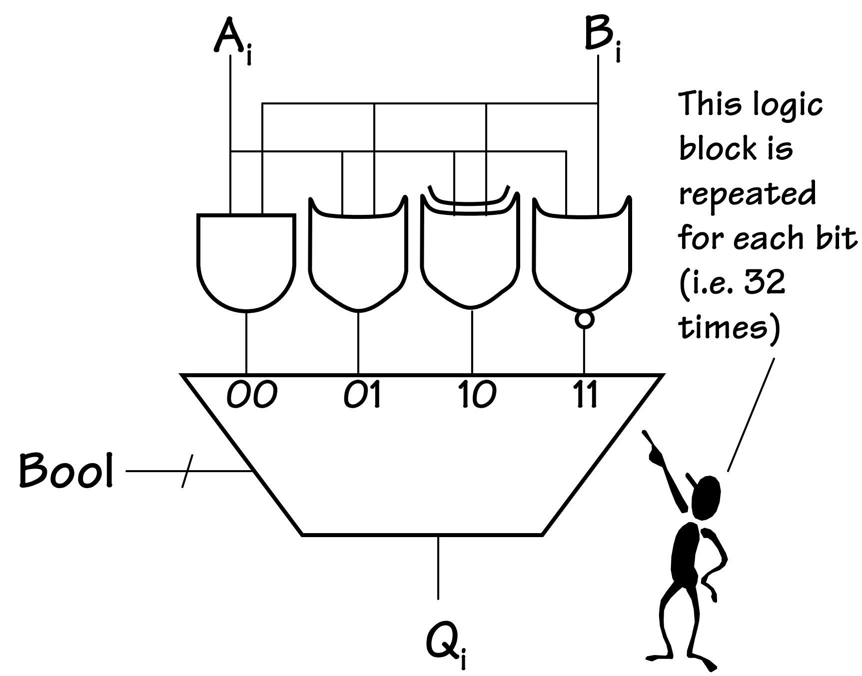
XORing is useful for "complementing" groups of bits.
ex. 10101110 ^ 00001111 = 10100001 (1's complement last 4 bits)

NORing is useful.. Uhm, *because* John Hennessy says it is!
ex. 10101110 # 00001111 = 01010000 (0's complement, 1's clear)

Boolean Unit

It is simple to build up a Boolean unit using primitive gates and a mux to select the function.

Since there is no interconnection between bits, this unit can be simply replicated at each position. The cost is about 7 gates per bit. One for each primitive function, and approx 3 for the 4-input mux.



This is a straightforward, but not too elegant of a design.

An ALU, at Last

Now we're ready for a big one! An Arithmetic Logic Unit.