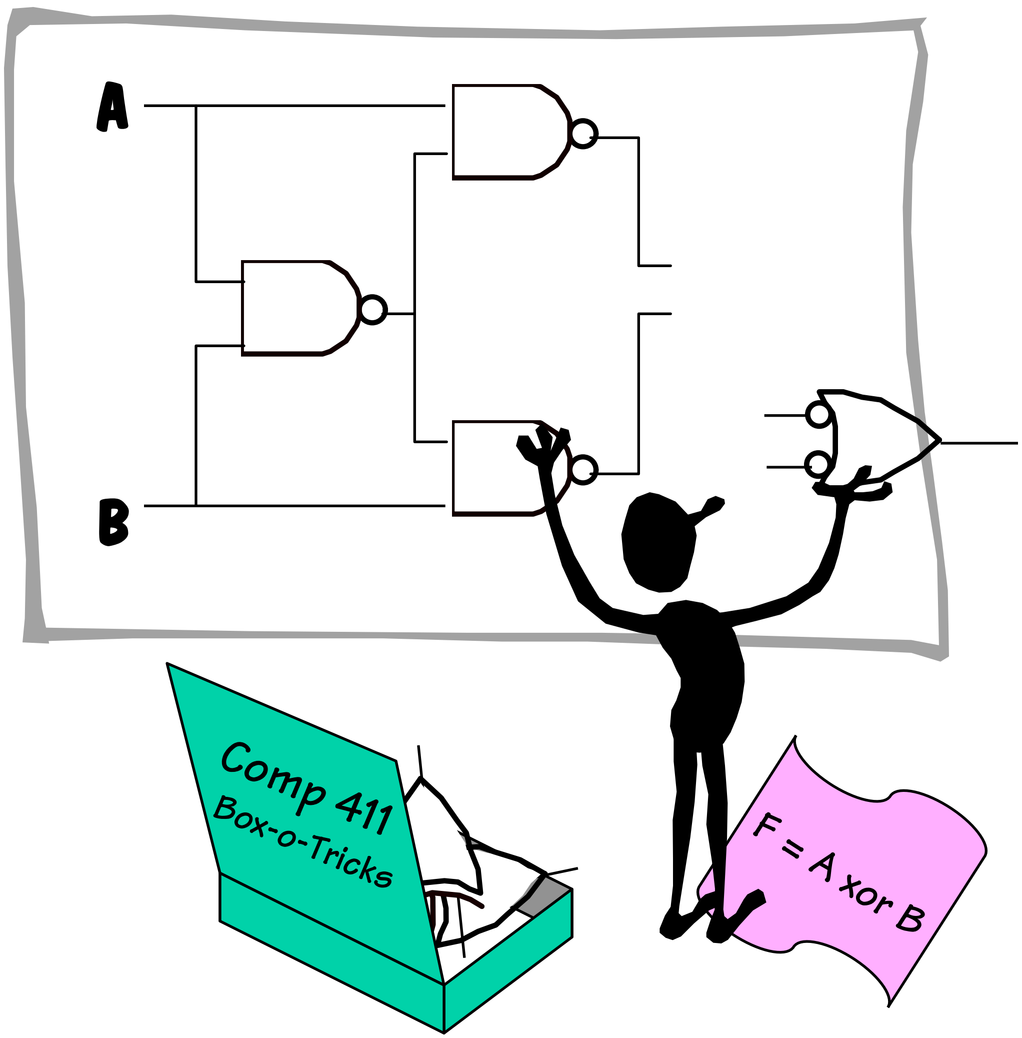
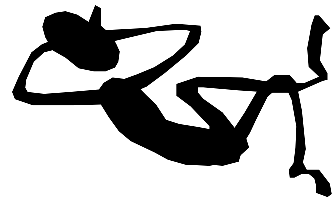
Transistors and Logic



1) The digital contract
2) Encoding bits with voltages
3) Processing bits with transistors
4) Gates
5) Large fanout gates
6) Truth-table SOP Realizations
7) Multiplexer Logic

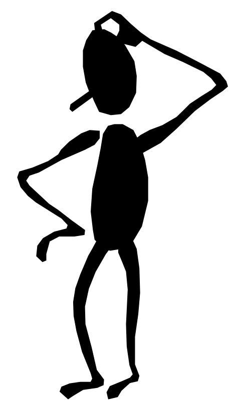
Where Are We?

Things we know so far -



1) Computers process information
2) Information is measured in bits
3) Data can be represented as groups of bits
4) Computer instructions are encoded as bits
5) Computer instructions are just data

6) We, humans, don't want to deal with bits…
   So we invent ASSEMBLY Language
   even that is too low-level so we invent
   COMPILERs, and they are too rigid so …

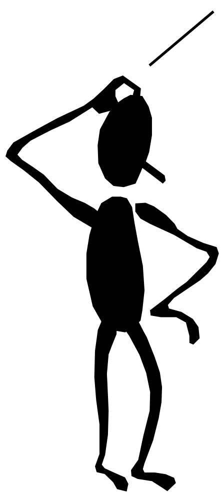


But, what PROCESSES all these bits?

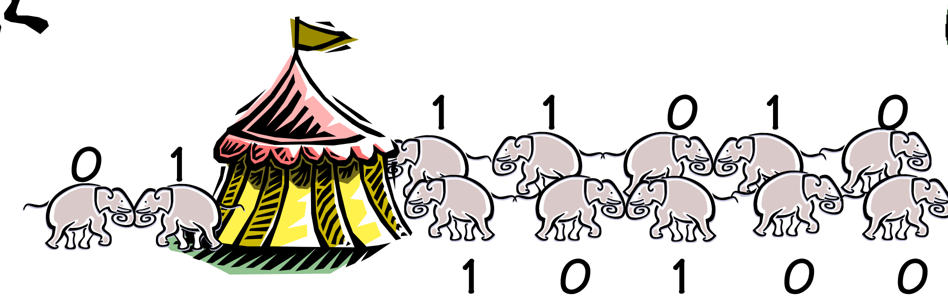
A Substrate for Computation

We can build devices for processing and representing bits using almost any physical phenomenon

Wait! Those last ones might have potential...



~~neutrino flux~~
~~trained elephants~~
~~engraved stone tablets~~
~~orbits of planets~~
~~sequences of amino acids~~
~~polarization of a photon~~



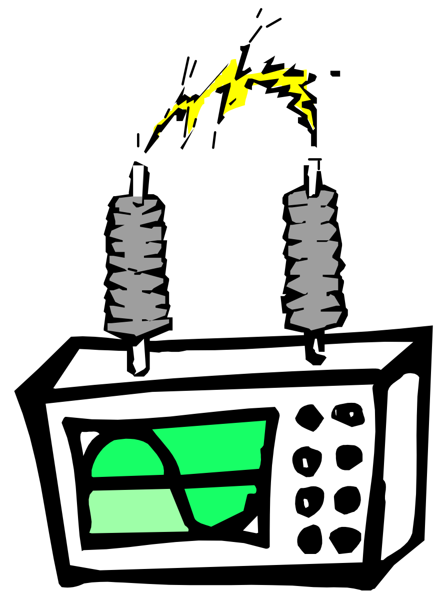
Using Electromagnetic Phenomena

Things like:

- voltages
- currents
- phase
- frequency

For today let's discuss using voltages to encode information.

- Voltage pros:
  - easy generation, detection
  - voltage changes can be very fast
  - lots of engineering knowledge
- Voltage cons:
  - easily affected by environment
  - DC connectivity required?
  - R & C effects slow things down



Representing Information with Voltage

Representation of each point (x, y) on a B&W Picture:

0 volts:	BLACK
1 volt:	WHITE
0.37 volts:	37% Gray
etc.	

Representation of a picture:

Scan points in some prescribed raster order… generate voltage waveform

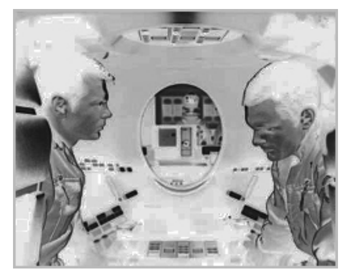
How much information at each point?

Information Processing = Computation

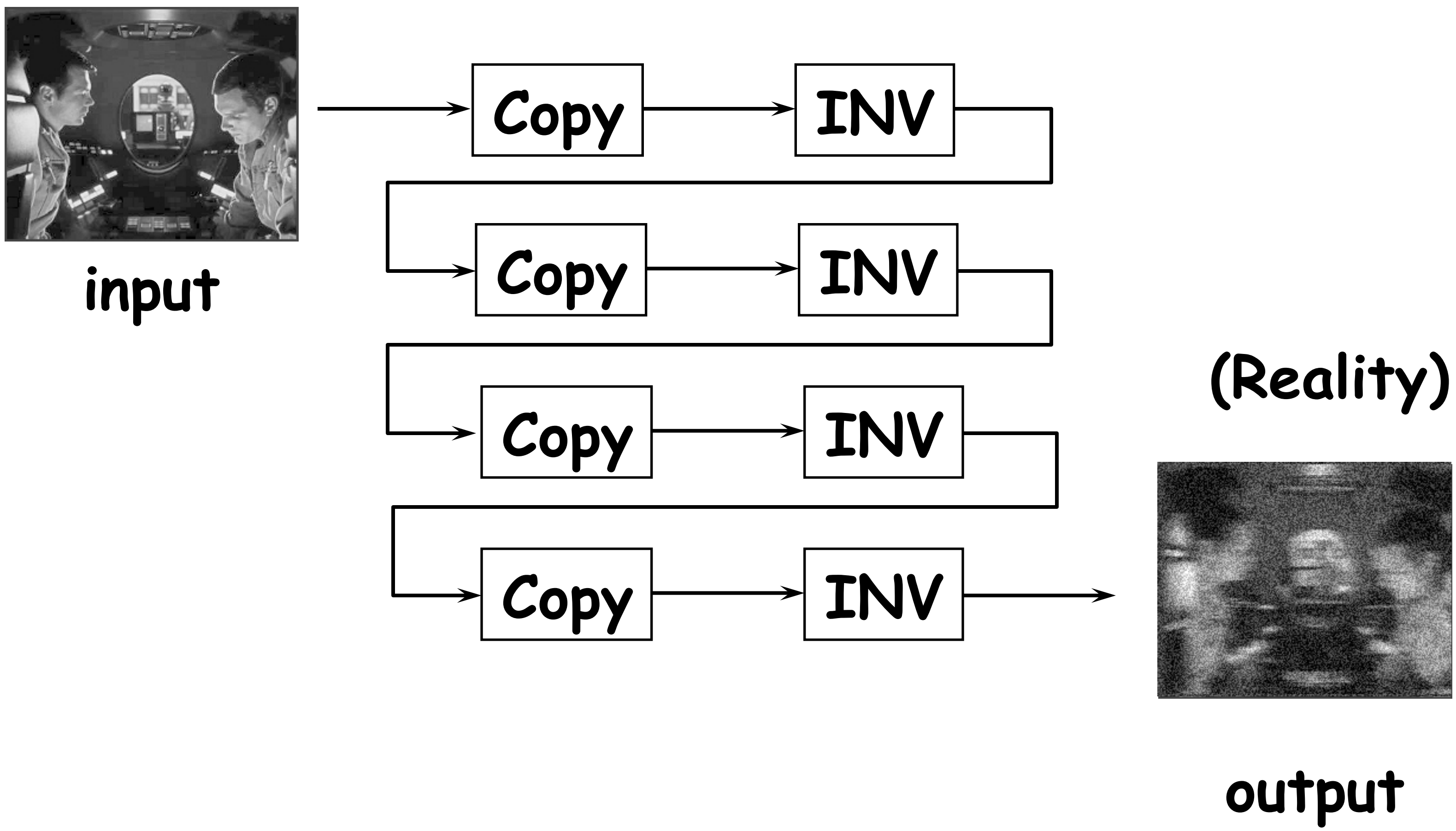
First, let's introduce some processing blocks:

v → **Copy** → v

v → **INV** → 1-v



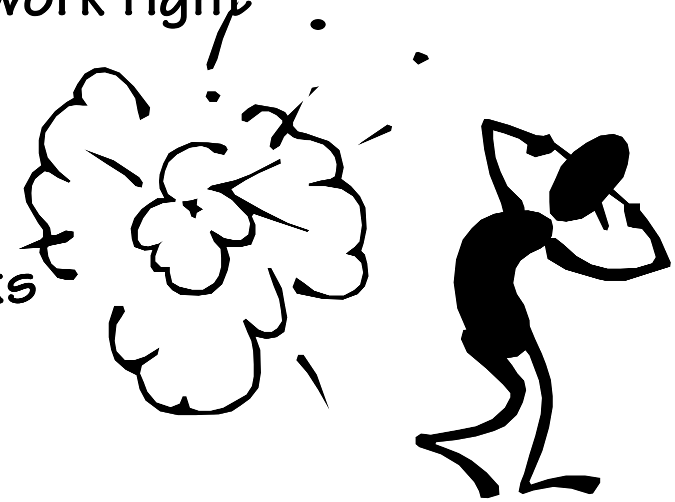
Let’s build a system!



Why Did Our System Fail?

Why doesn't reality match theory?

1. COPY Operator doesn't work right
2. INVERSION Operator doesn't work right
3. Theory is imperfect
4. Reality is imperfect
5. Our system architecture stinks

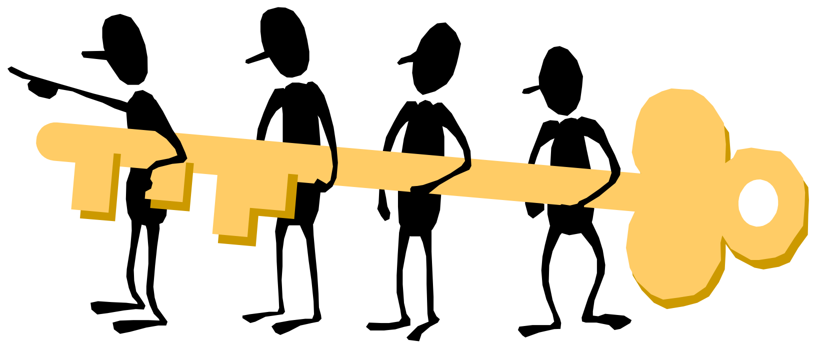


ANSWER: all of the above!

Noise and inaccuracy are inevitable; we can't reliably reproduce infinite information-- we must design our system to tolerate some amount of error if it is to process information reliably.

The Key to System Design

A SYSTEM is a structure that is guaranteed to exhibit a specified behavior, assuming all of its components obey their specified behaviors.



How is this achieved? Contracts

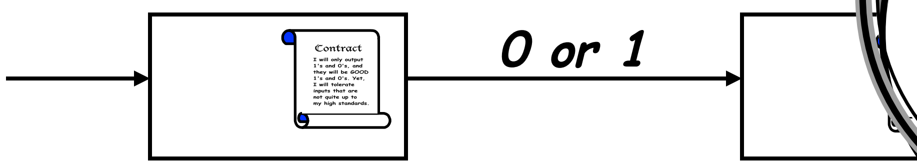
Every system component will have clear obligations and responsibilities. If these are maintained we have every right to expect the system to behave as planned. If contracts are violated all bets are off.

The Digital Panacea ...

Why DIGITAL?

... because it keeps the contracts SIMPLE!

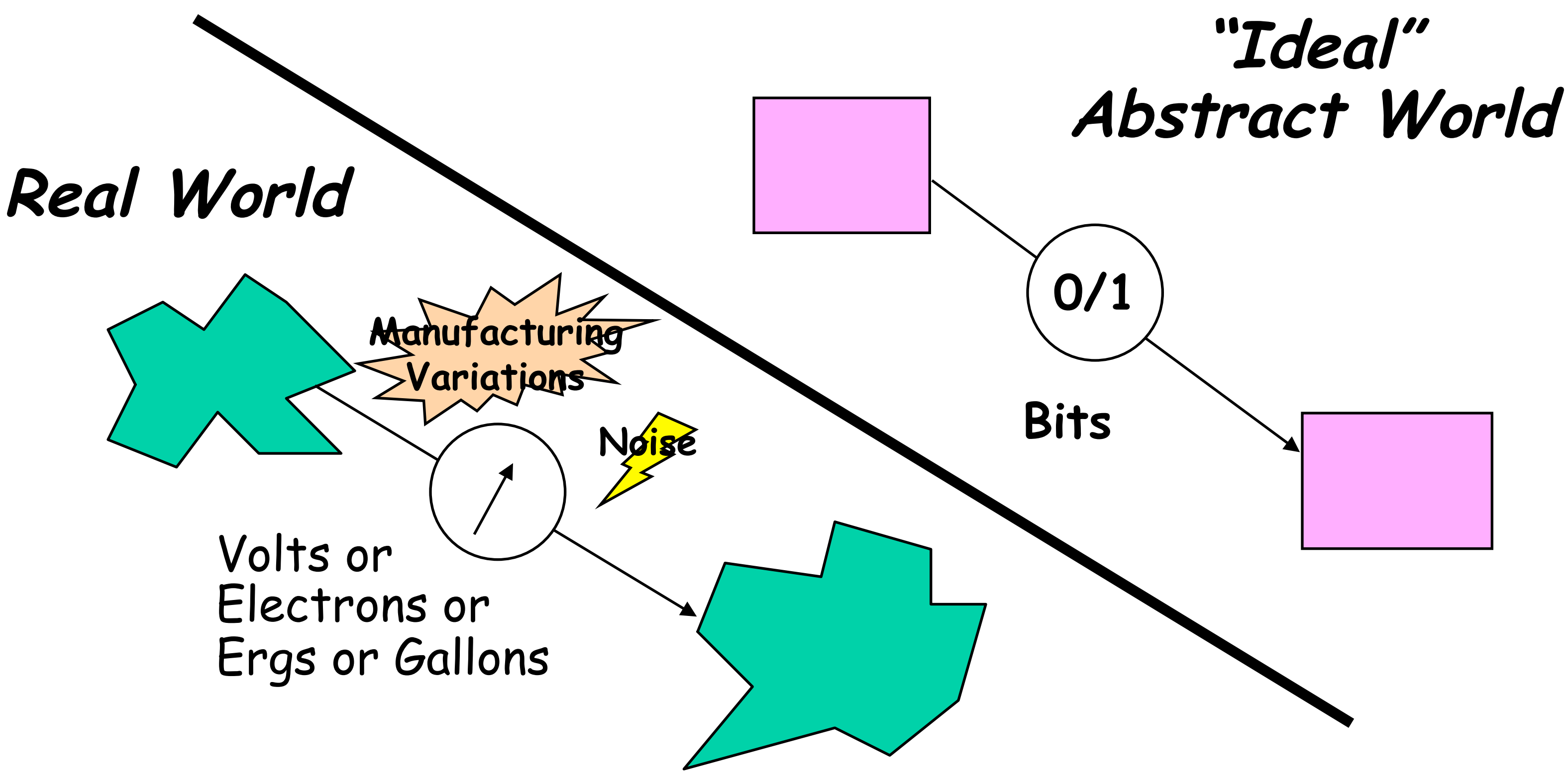
The price we pay for this robustness?



All the information that we transfer between components is only 1 crummy bit!

But, in exchange, we get a guarantee of a reliable system.

The Digital Abstraction

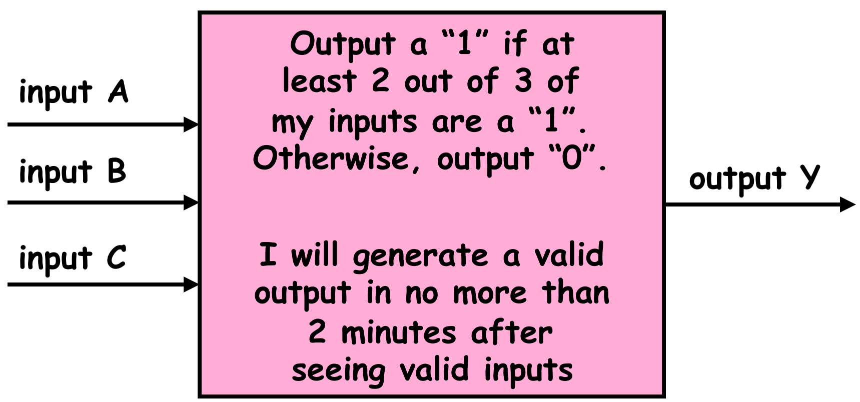


Keep in mind, the world is not digital, we engineer it to behave that way.
We must use real physical phenomena to implement digital designs!

A Digital Processing Element

Static Discipline

- A *combinational device* is a circuit element that has
  - one or more digital *inputs*
  - one or more digital *outputs*
  - a *functional specification* that details the value of each output for every possible combination of valid input values
  - a *timing specification* consisting (at minimum) of an upper bound propagation delay, t_{pd}, on the required time for the device to compute the specified output values from an arbitrary set of stable, valid input values



A Combinational Digital System

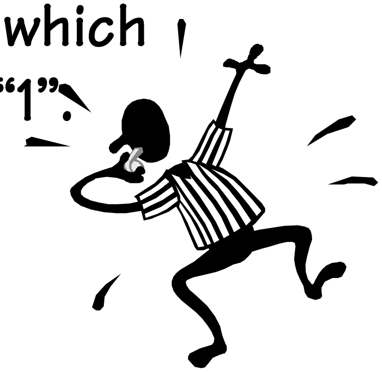
- A *system* of interconnected elements is *combinational* if
  - each circuit element is combinational
  - every input is connected to exactly one output or directly to some source of 0's or 1's
  - the circuit contains no directed cycles

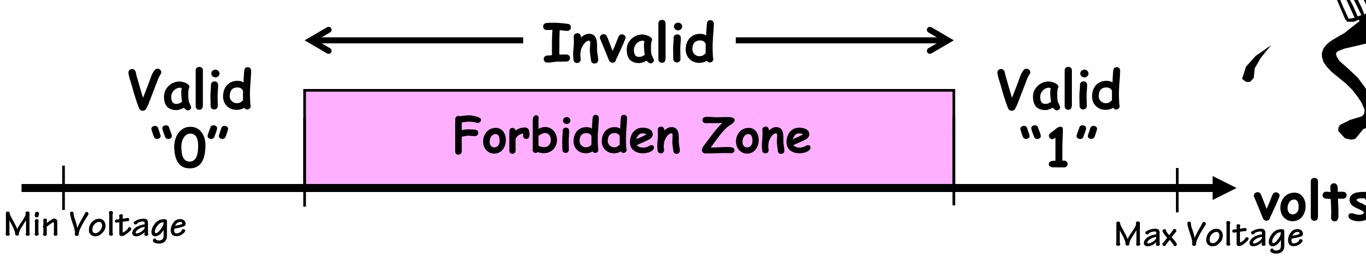
No feedback (yet!)

- But, in order to realize digital processing elements we have one more requirement!

Noise Margins

- Key idea:
  Don't allow "0" to be mistaken for a "1" or vice versa
- Use the same "uniform representation convention", for *every* component in our digital system
- To implement devices with high reliability, we outlaw "close calls" via a representation convention which forbids a range of voltages between "0" and "1".



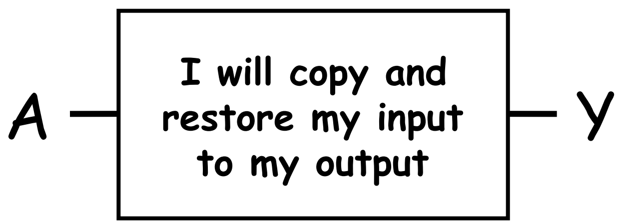


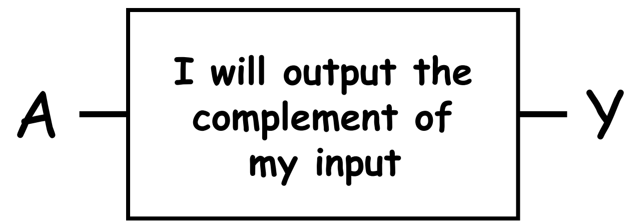
CONSEQUENCE:

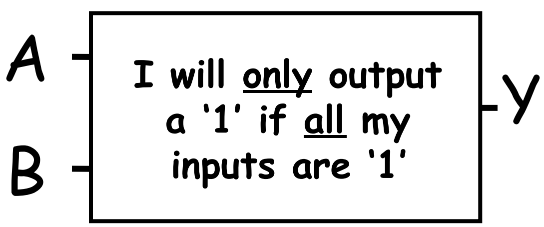
Notion of "VALID" and "INVALID" logic levels

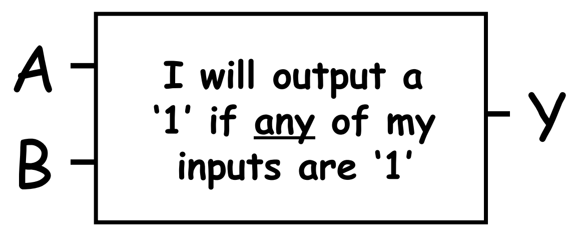
Digital Processing Elements

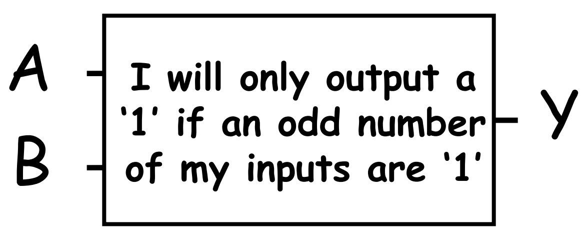
Some digital processing elements occur so frequently that we give them special names and symbols





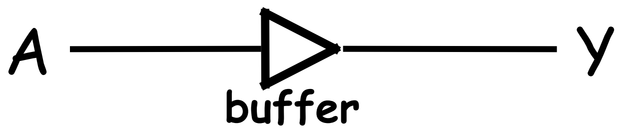


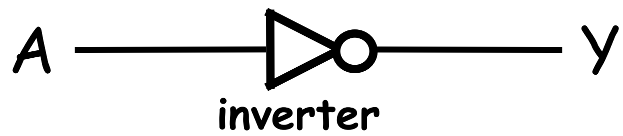


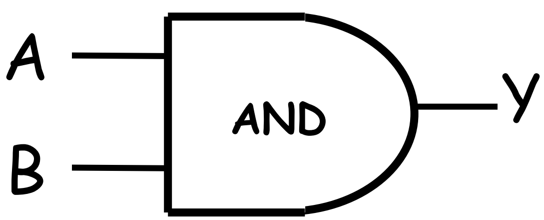


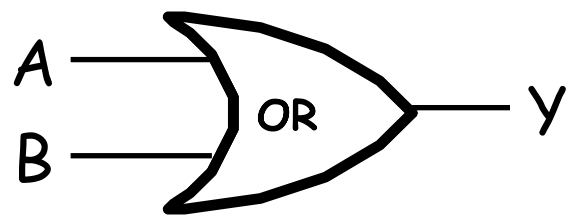
Digital Processing Elements

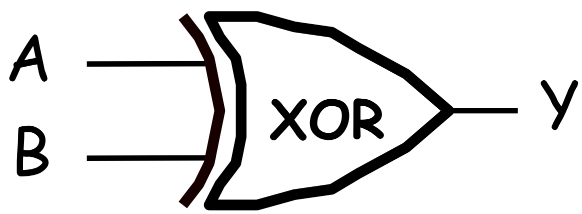
Some digital processing elements occur so frequently that we give them special names and symbols







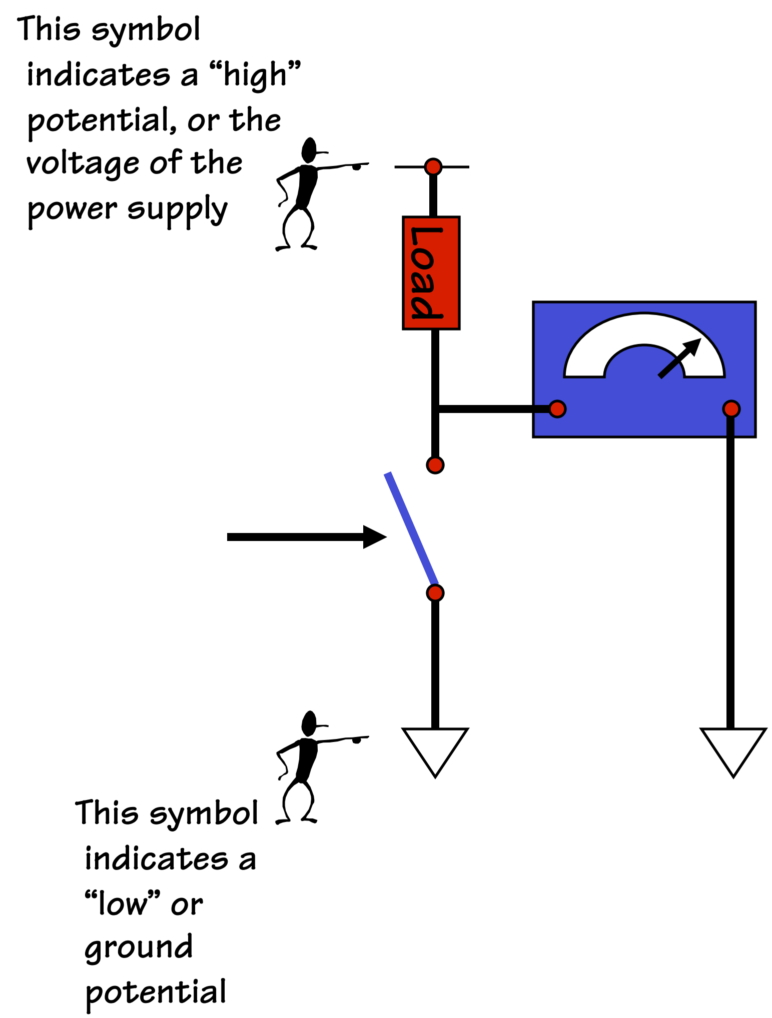




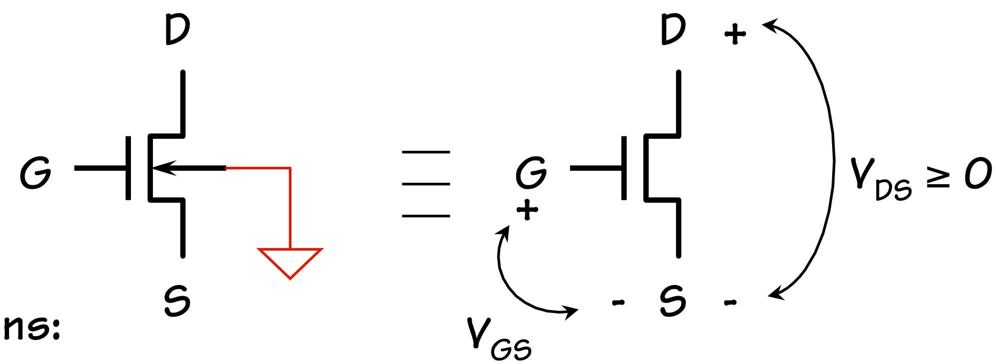
In honor of the richest man in the world we will henceforth refer to digital processing elements as "GATES"

From What Do We Make Digital Devices?

- A *controllable* switch is the common link of all computing technologies
- How do you control voltages with a switch?
- By creating and opening paths between higher and lower potentials



N-Channel Field-Effect Transistors (NFETs)



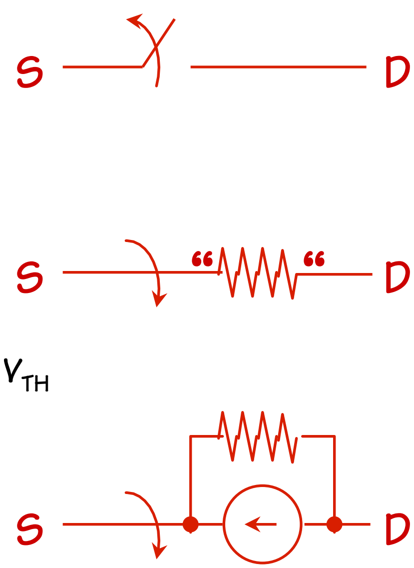
When the gate voltage is high, the switch closes. Good at pulling things "low".

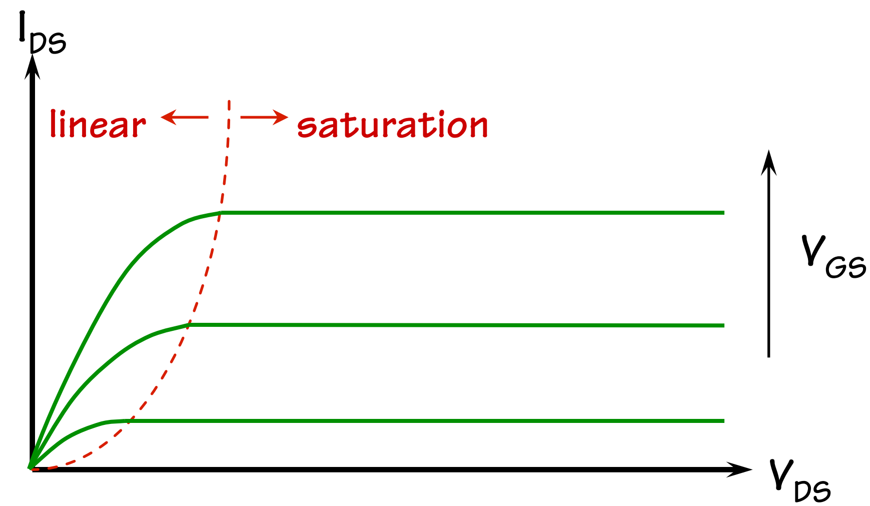
Operating regions:

cut-off:
$V_{GS} < V_{TH}$ ← 0.8V

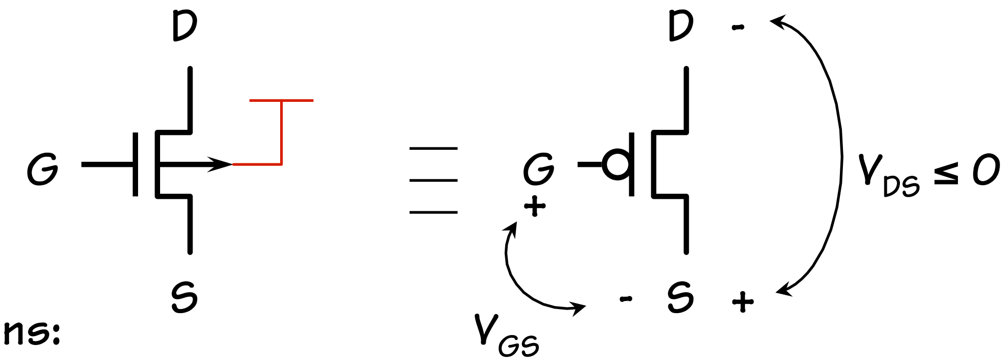
linear:
$V_{GS} \geq V_{TH}$
$V_{DS} < V_{Dsat}$ ← $V_{GS} - V_{TH}$

saturation:
$V_{GS} \geq V_{TH}$
$V_{DS} \geq V_{Dsat}$





P-Channel Field-Effect Transistors (PFETs)



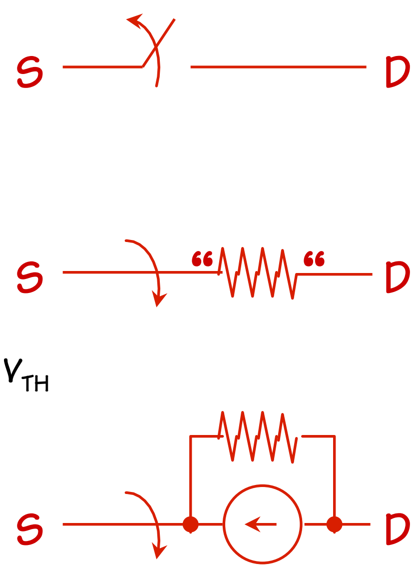
When the gate voltage is low, the switch closes. Good at pulling things “high”.

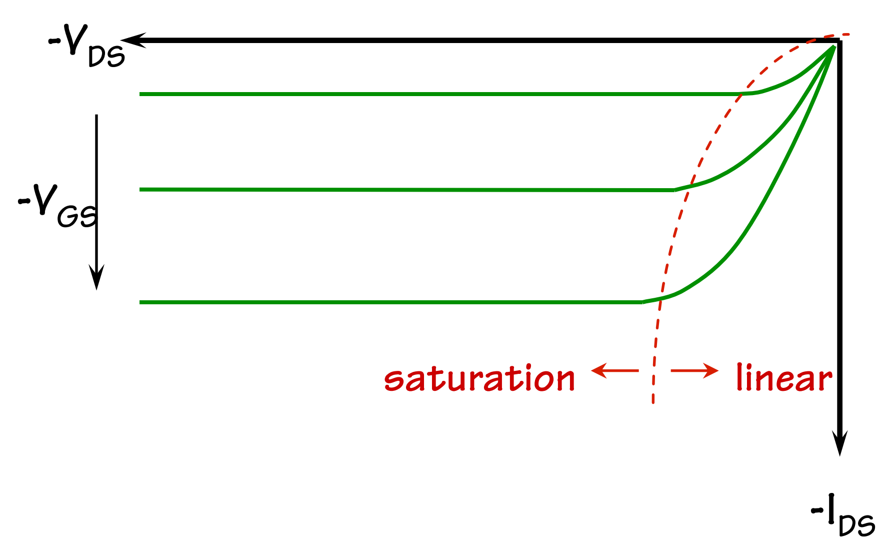
Operating regions:

cut-off:
$V_{GS} > V_{TH}$ ← −0.8V

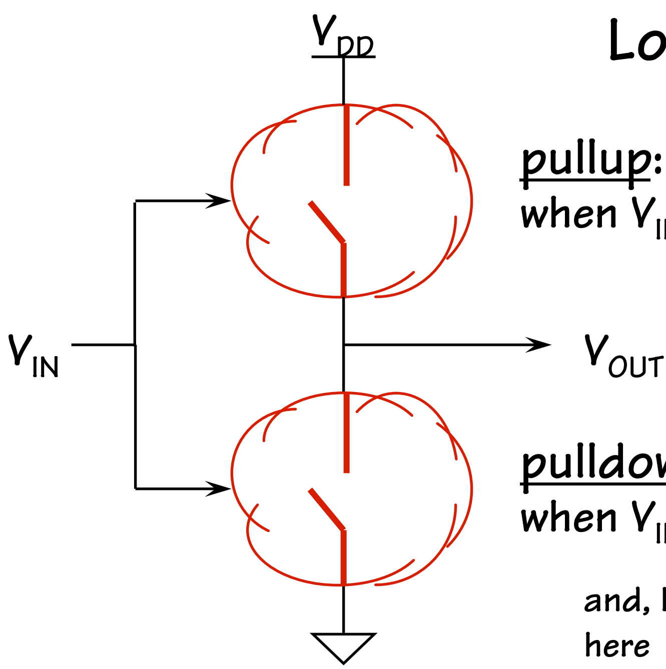
linear:
$V_{GS} \le V_{TH}$
$V_{DS} > V_{Dsat}$ ← $V_{GS} - V_{TH}$

saturation:
$V_{GS} \le V_{TH}$
$V_{DS} \le V_{Dsat}$



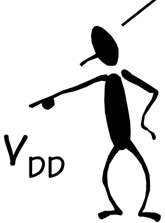


Finally… Using Transistors to Build Logic Gates!



Logic Gate recipe:

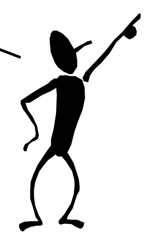
We'll use PFETs here



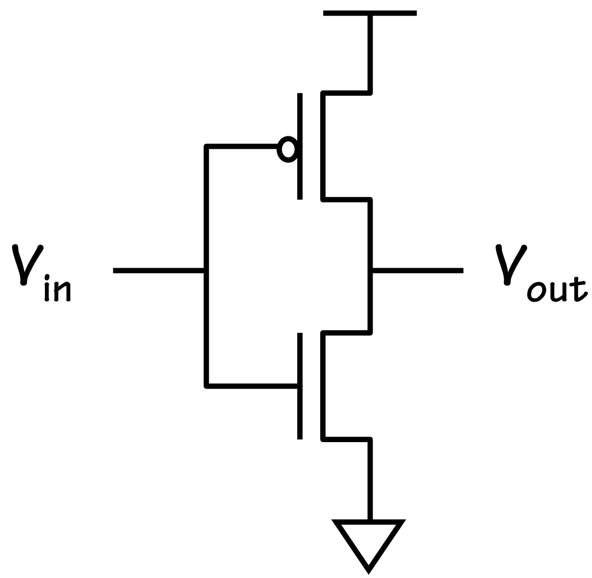
<u>pullup</u>: make this connection when V_{IN} is near 0 so that $V_{OUT} = V_{DD}$

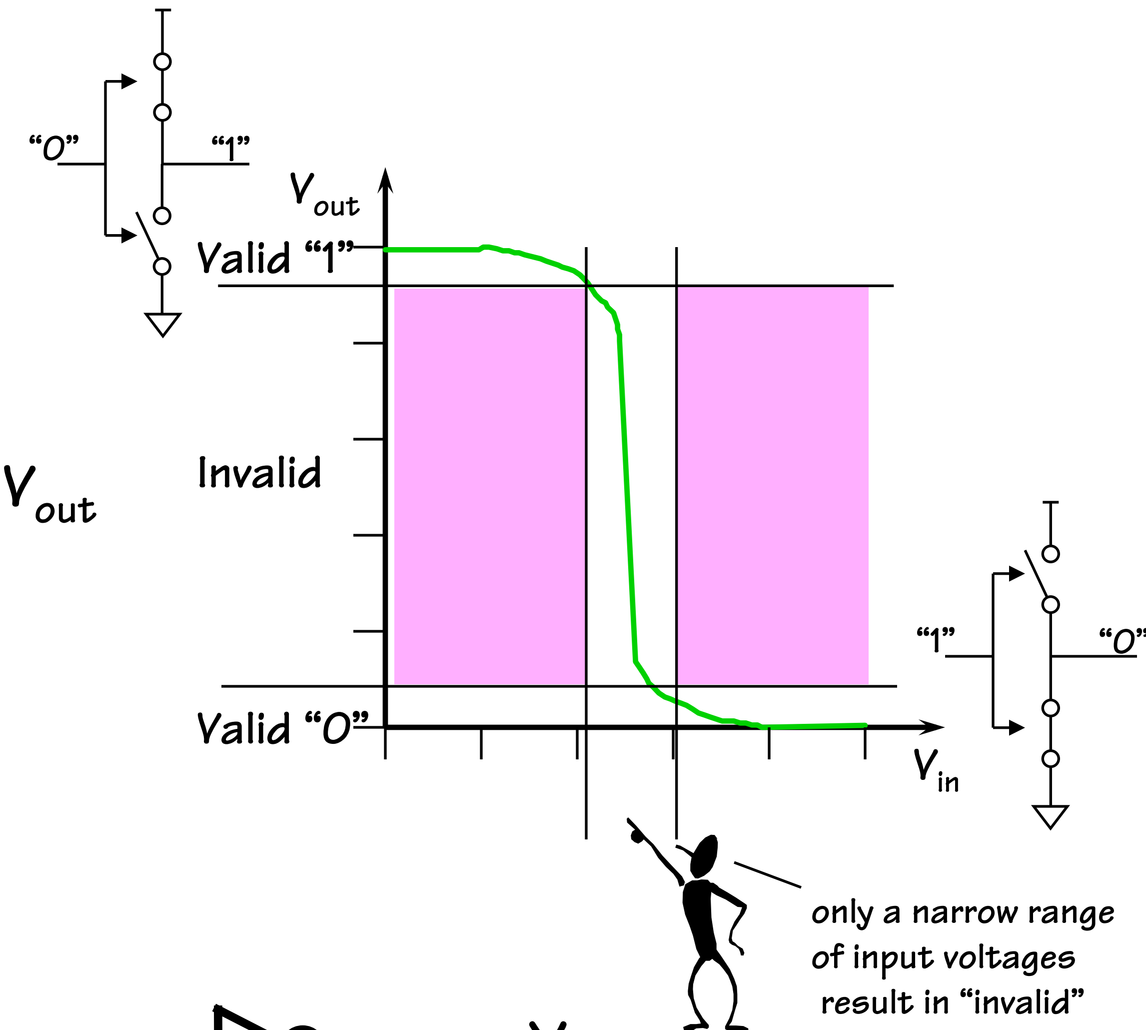
<u>pulldown</u>: make this connection when V_{IN} is near V_{DD} so that $V_{OUT} = 0$

and, NFETs here

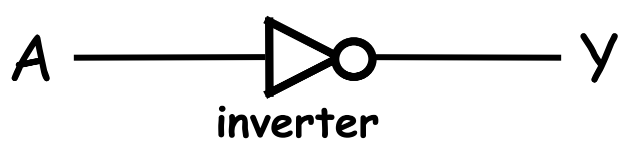


CMOS Inverter





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



Complementary Pullups and Pulldowns

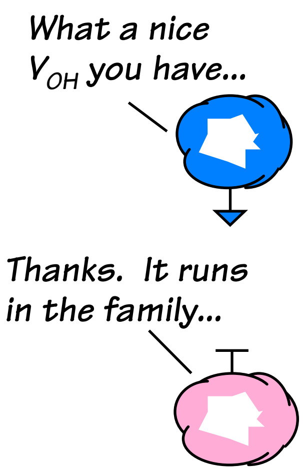
This is what the "C" in CMOS stands for!

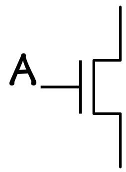
We design components with *complementary* pullup and pulldown logic (i.e., the pulldown should be "on" when the pullup is "off" and vice versa).

pullup	pulldown	$F(A_1,\ldots,An)$
on	off	driven "1"
off	on	driven "0"
on	on	driven "X"
off	off	no connection

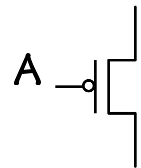
Since there's plenty of capacitance on output nodes, so when the output becomes disconnected it tends to "remember" its previous voltage– at least for a while. The "memory" is the load capacitor's charge. Leakage currents will cause eventual decay of the charge (that's why DRAMs need to be refreshed!).

CMOS Complements



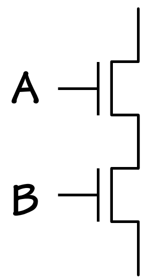


conducts when A is high

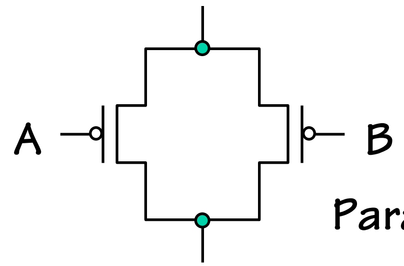


conducts when A is low

Series N connections:

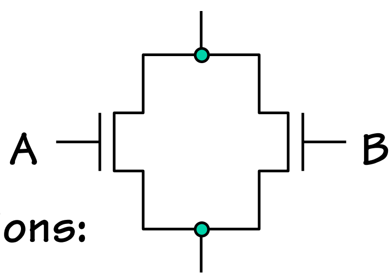


conducts when A is high
and B is high: A·B



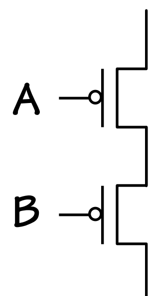
Parallel P connections:

conducts when A is low
or B is low: $\overline{A+B} = \overline{A}\cdot\overline{B}$



Parallel N connections:

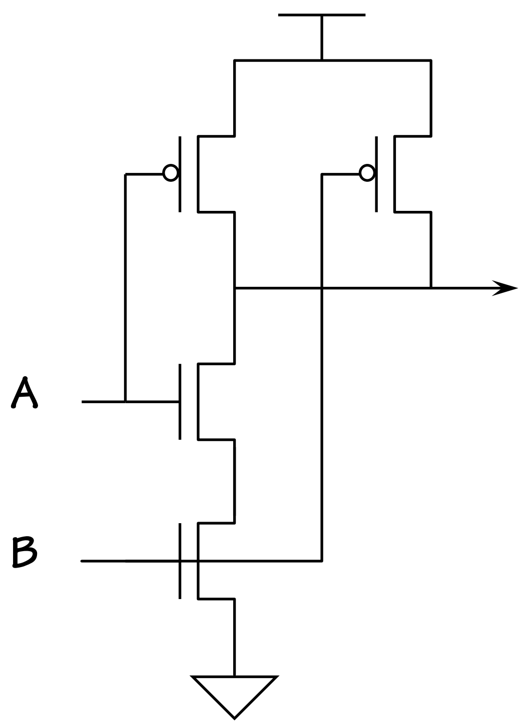
conducts when A is high
or B is high: A+B



Series P connections:

conducts when A is low
and B is low: $\overline{A}\cdot\overline{B} = \overline{A+B}$

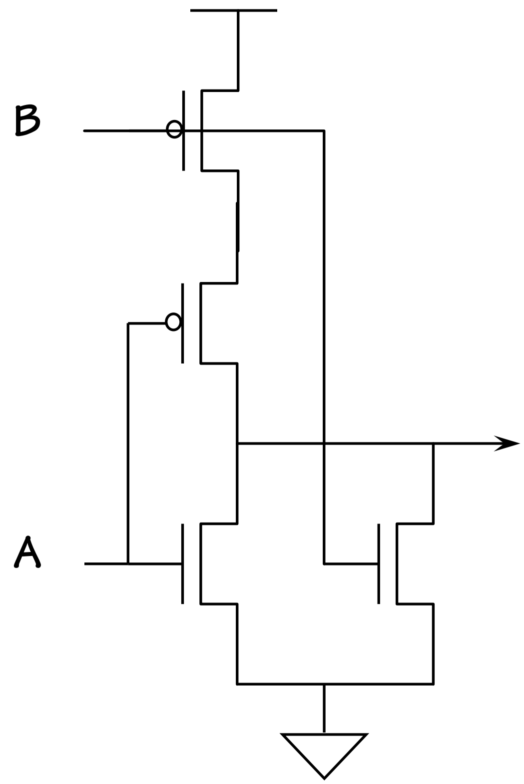
A Two Input Logic Gate



What function does this gate compute?

A	B	C
0	0	
0	1	
1	0	
1	1	

Here's Another...



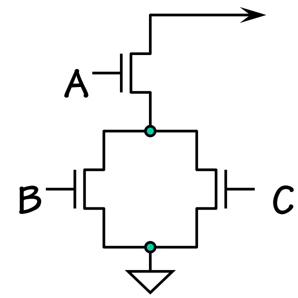
What function does this gate compute?

A	B	C
0	0	
0	1	
1	0	
1	1	

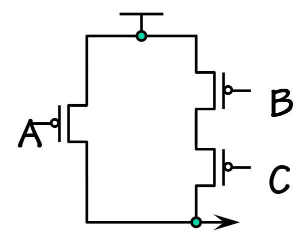
General CMOS Gate Recipe

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

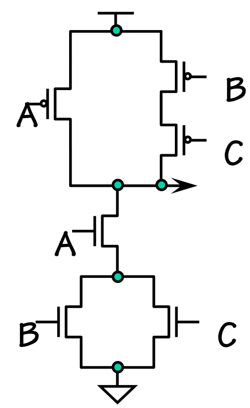
e.g., $F = \overline{A*(B+C)}$



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?

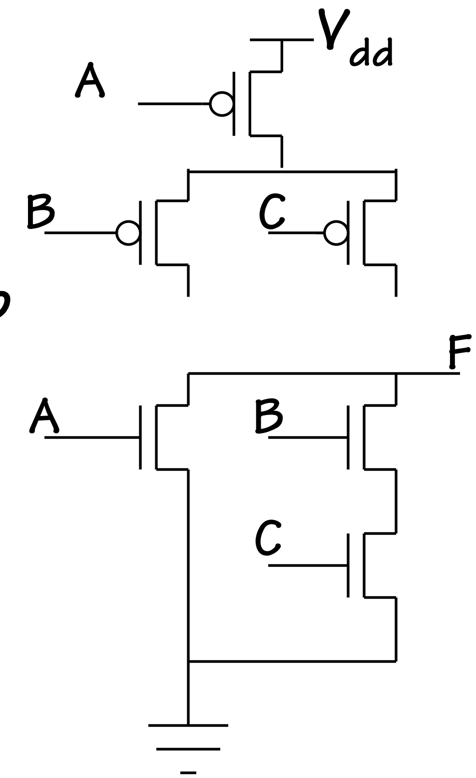
One Last Exercise

Lets construct a gate to compute:

$F = \overline{A+BC} = NOT(OR(A,AND(B,C)))$

Step 1: The pull-down network

Step 2: The complementary pull-up network



One Last Exercise

Lets construct a gate to compute:

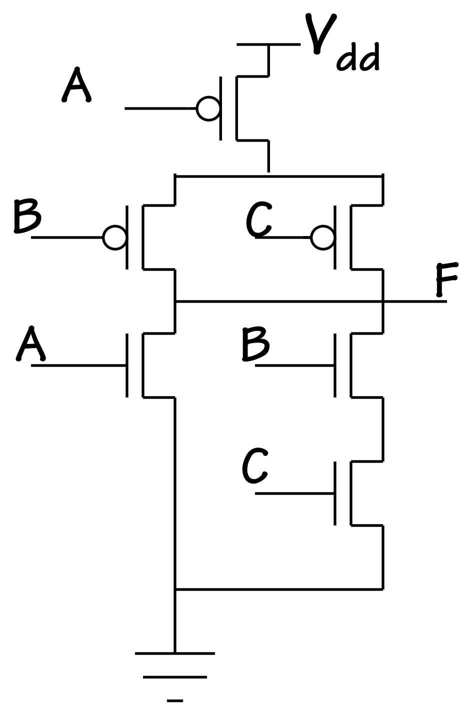
$F = \overline{A+BC} = NOT(OR(A,AND(B,C)))$

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

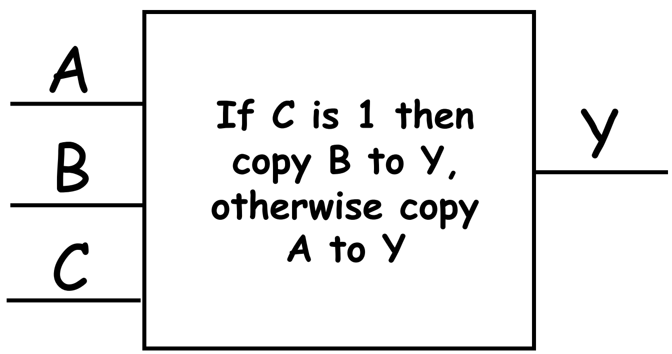
Step 3: Combine and Verify



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**.

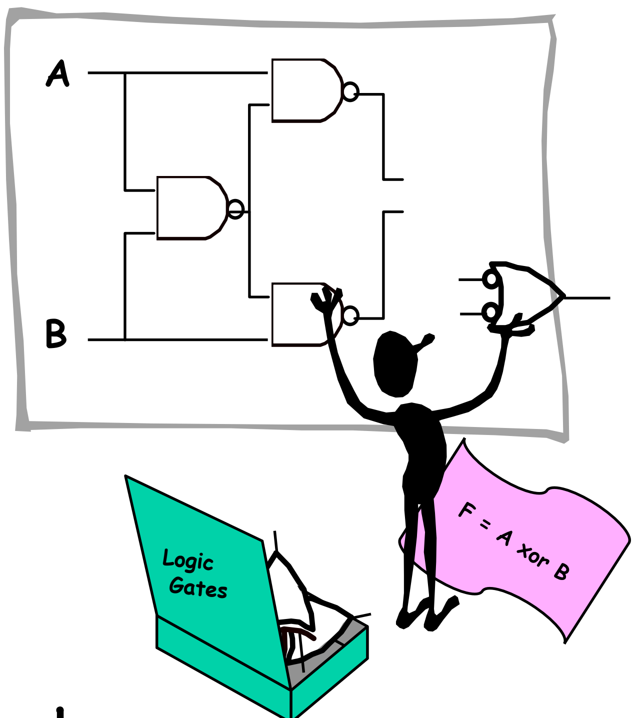
These "**truth tables**" are a concise description of the combinational system's function. Conversely, any computation performed by a combinational system can expressed as a truth table.

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?

Did I mention we have gates?

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	AND Y	OR AB	OR Y	NAND AB	NAND Y	NOR AB	NOR 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

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

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

Do we need all of these gates?

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

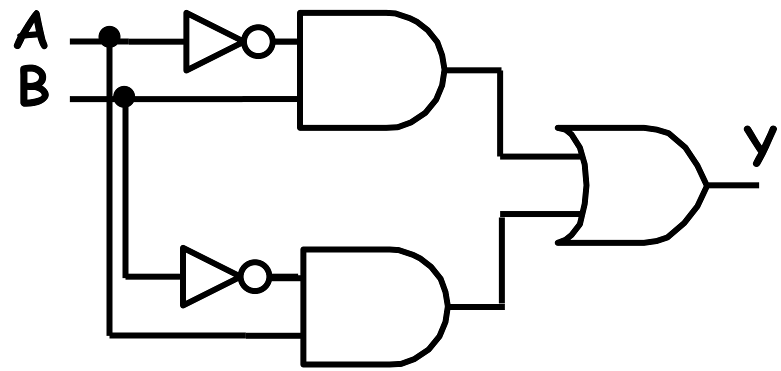
We Can Make Most Gates Out of Others

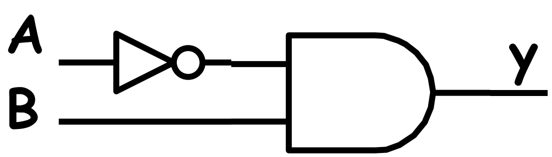
B>A

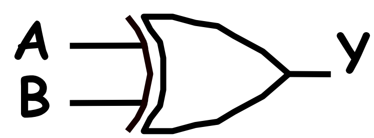
AB	Y
00	0
01	1
10	0
11	0

XOR

AB	Y
00	0
01	1
10	1
11	0



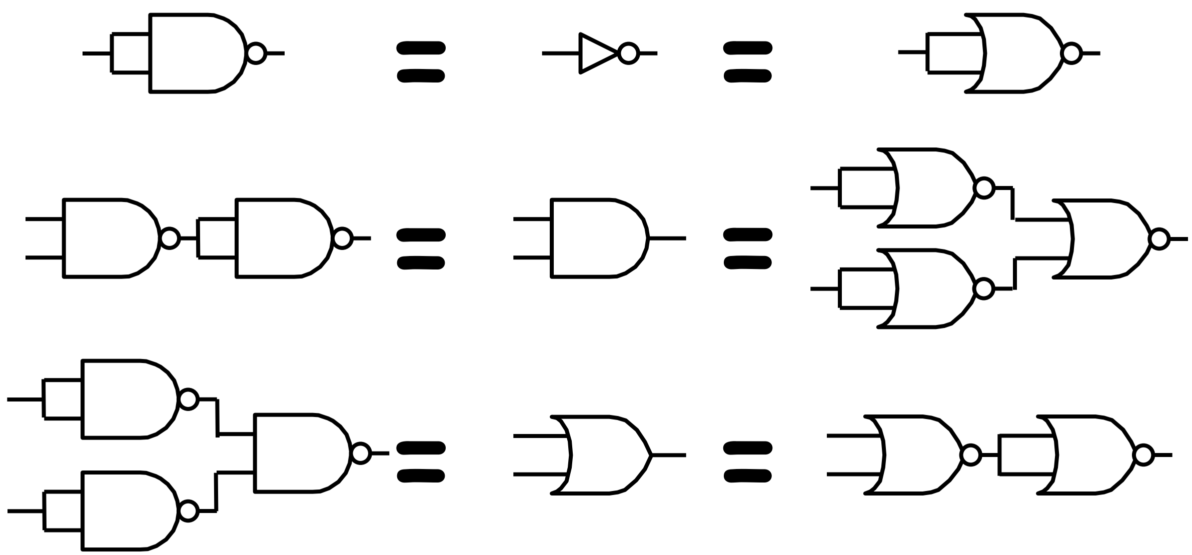




How many different gates do we really need?

One Will Do!

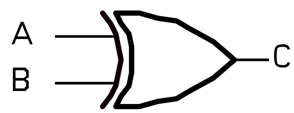
NANDs and NORs are universal



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

Stupid Gate Tricks

Suppose we have some 2-input XOR gates:

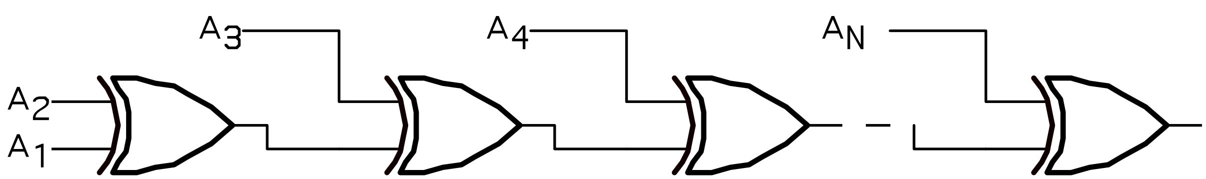


$t_{pd} = 1$

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
("PARITY")

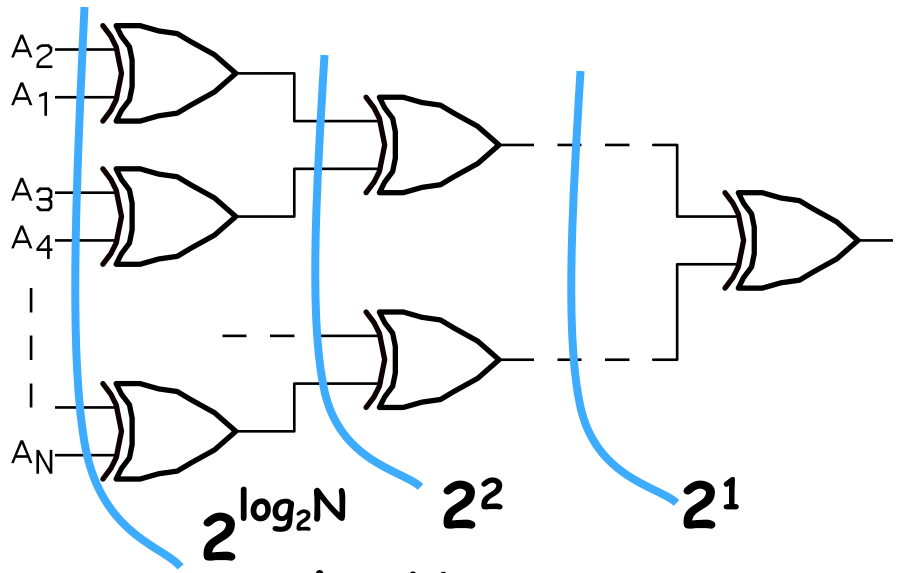


t_{pd} = O(N) -- WORST CASE.

Can we compute N-input XOR faster?

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.

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

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

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

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

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

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

SUM-OF-PRODUCTS

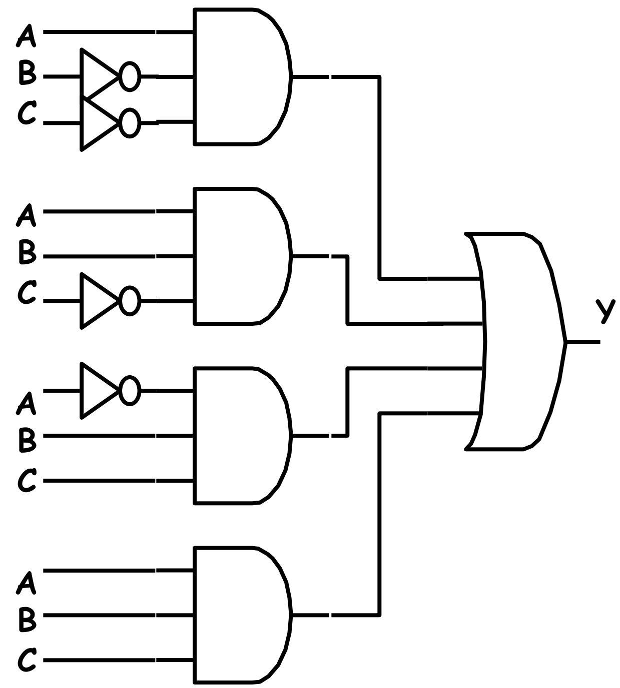
Straightforward Synthesis

We can implement

SUM-OF-PRODUCTS

with just three levels of logic.

INVERTERS/AND/OR

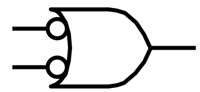


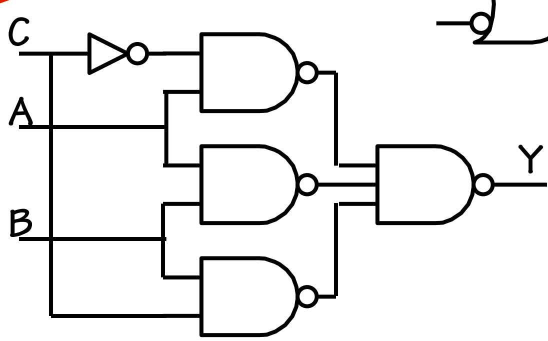
Useful Gate Structures

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

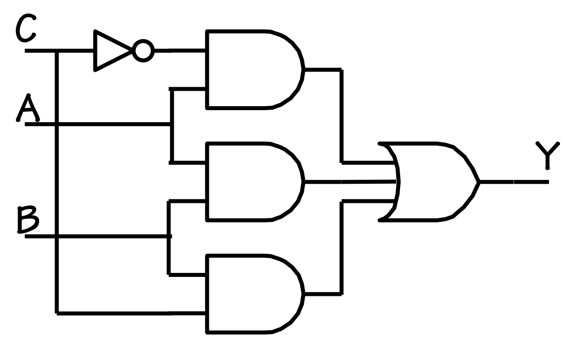
"Pushing Bubbles"

NAND-NAND





≡

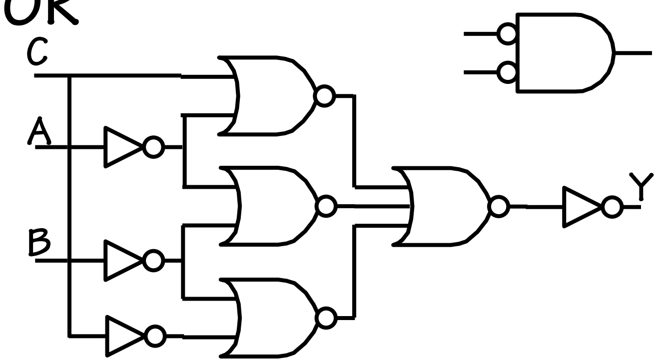


$xyz = \overline{x} + \overline{y} + \overline{z}$

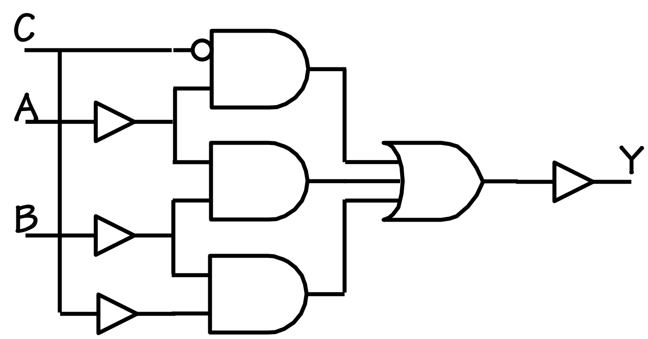
DeMorgan's Laws

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

NOR-NOR



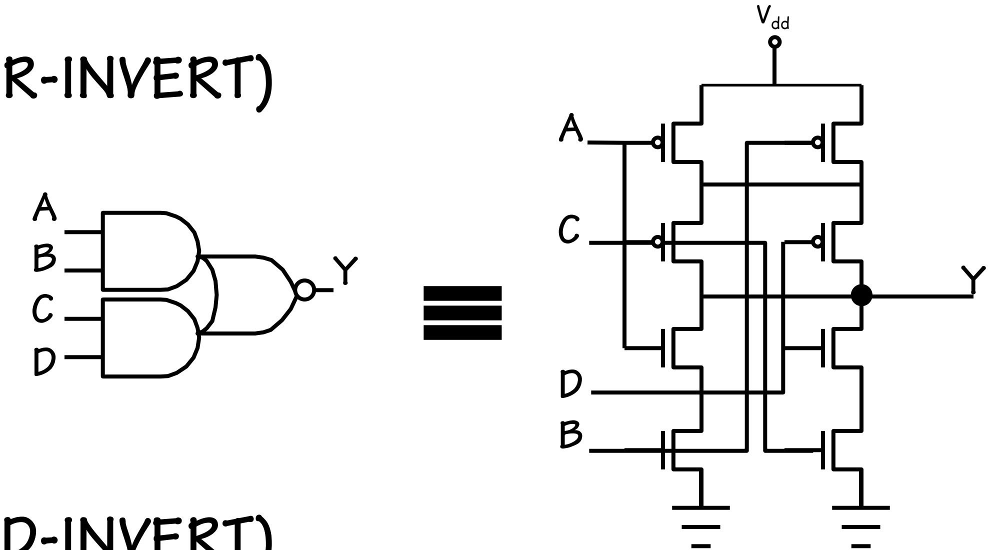
≡



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

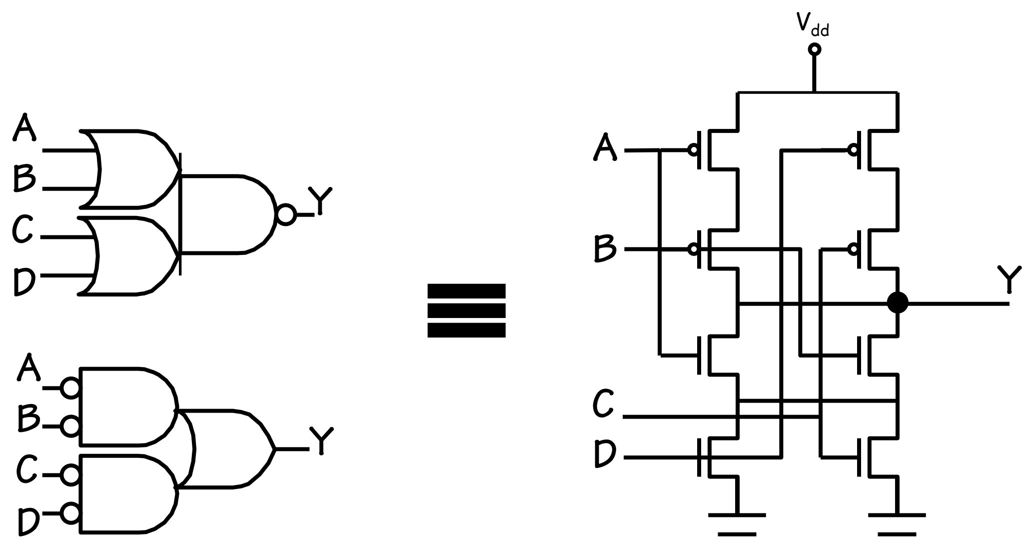
More Useful Gate Structures

AOI (AND-OR-INVERT)

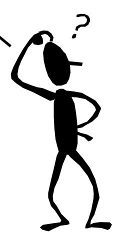


AOI and OAI structures can be realized using a single CMOS gate. However, their function is equivalent to 3 levels of logic.

OAI (OR-AND-INVERT)

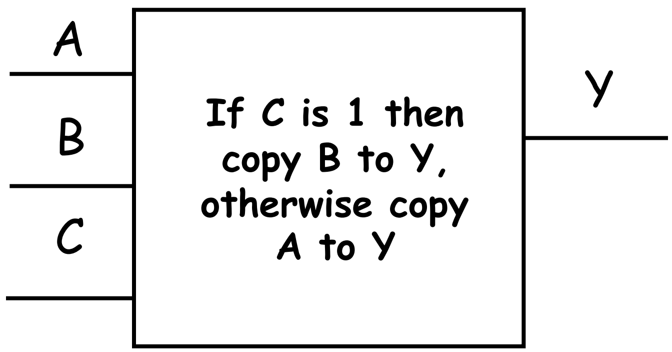


An OAI's DeMorgan equivalent is usually easier to think about.



An Interesting 3-Input Gate

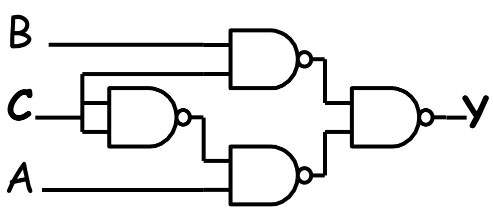
Based on C, select the A or B input to be copied to the output Y.



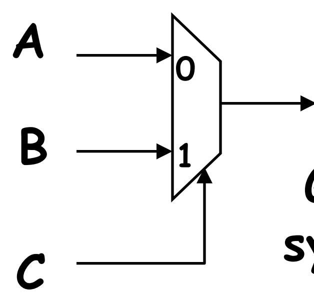
Truth Table

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



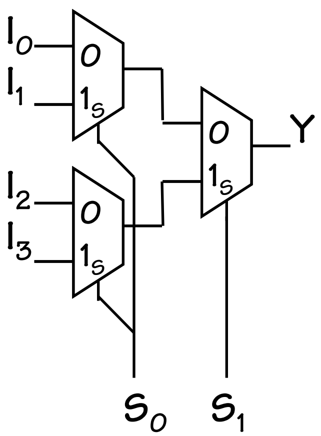
schematic

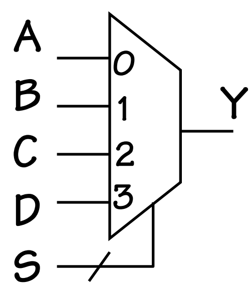


Gate symbol

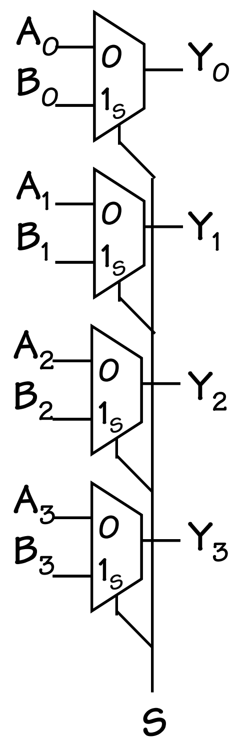
MUX Shortcuts

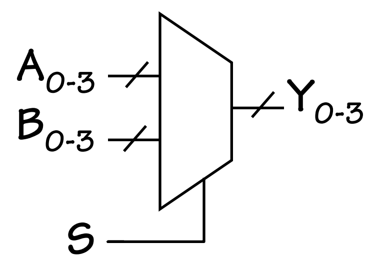
A 4-input Mux (implemented as a tree)





A 4-bit wide Mux





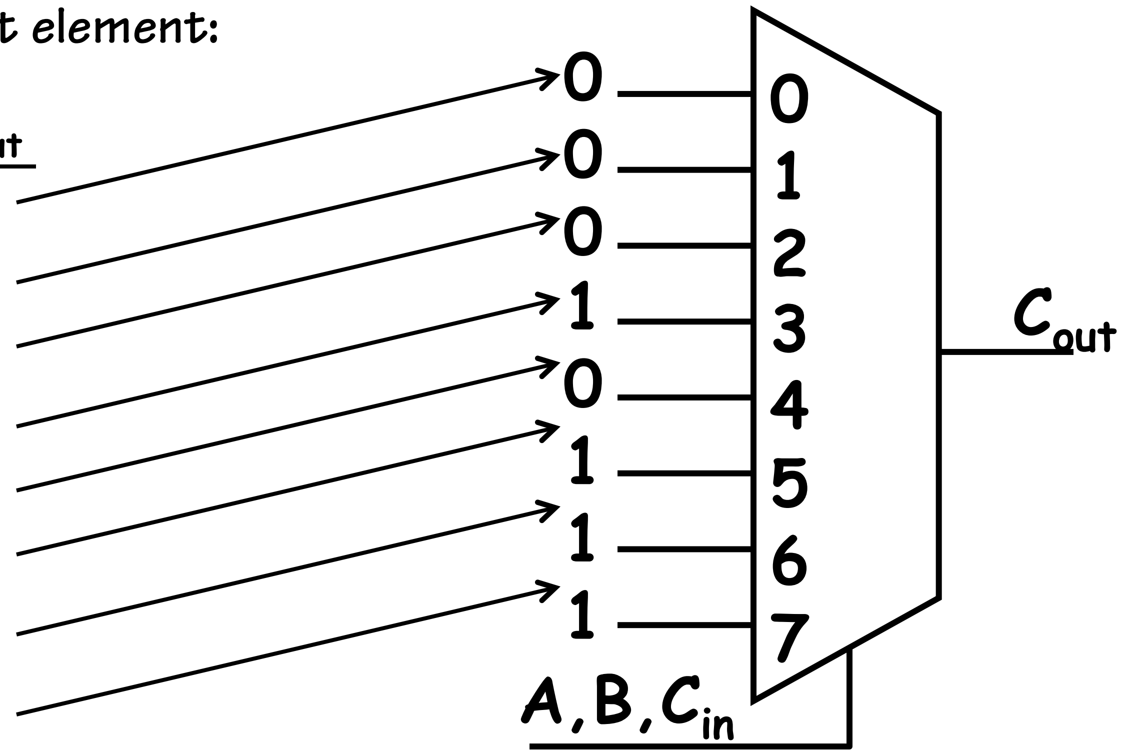
Mux Logic Synthesis

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

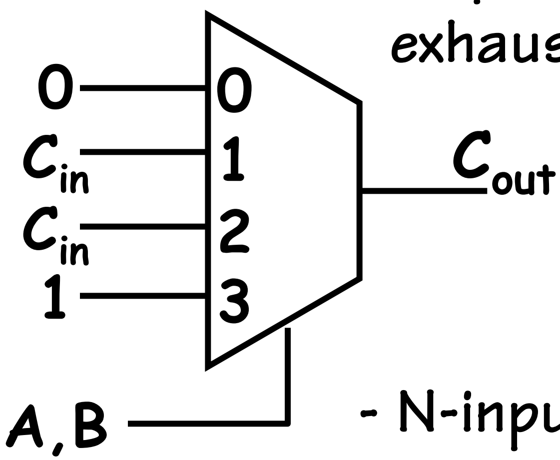


Small Improvements

We can also apply certain optimizations to MUX Logic

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



- Largely by inspection or exhaustive search

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

Next Time

Binary Arithmetic

- Circuits that:
  - ADD
  - SUBTRACT
  - SHIFT

"7x8=52, 9-3=5... things like that."