Layers of abstraction and managing complexity.

One of the most important parts of engineering, if not the single most important part, is managing abstraction. What I mean is that when we work on a task, we might work at different levels of abstraction at different times as the problem demands. Perhaps the best example involves programming. When program we use "functional decomposition" to design a solution for the problem. That is, we break the problem into parts in a way that each part (function) can be written without really keeping track of what the other functions are going to do. Function "A" might read a file into a structure, function B might process that structure, and function C might display the results. Further, function B is probably complex enough that it needs to be broken into parts. All this serves to let us solve a big problem while only having to work on a small part at a time. It also makes it easier to split the work up among different people.

Other forms of engineering have similar properties. When designing a complex mechanical system, or trying to improve throughput on a factory floor (I've done both), you are best served by breaking the problem into parts that are largely independent and solving them each. You'll find that you can break things down into too small of bits (what threading pitch of the screws would work best here? What brand of forklift should we buy?), but often you are best off ignoring those issues when engineering a solution: there are just too many details one *could* examine and you need to stop decomposing the problem at some point.

In digital design we've already seen some of this. We've shown that MSI devices can be a really useful way to solve problems. You may well know how to implement each MSI device, but we *think* about each device as a "magic box", not worrying about exactly how they are implemented until later. (So functional decomposition again.)

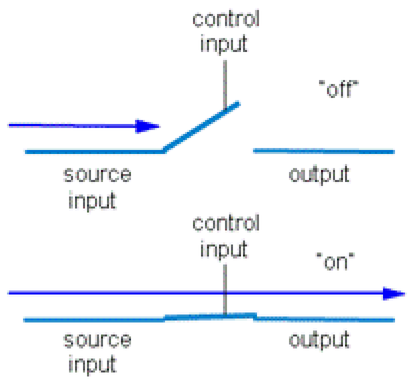
All that said: *It is useful to be familiar with the issues at a level of abstraction* ***above****, and* ***below*** *the one(s) you normally work at*. The levels of abstraction are rarely so rigid that you don't need to communicate with those at the levels above and below you. So, it is useful to learn at least enough to talk to them. So today we're going to spend some time on *a* level below digital logic and, if we have time, a short bit on a level above us.

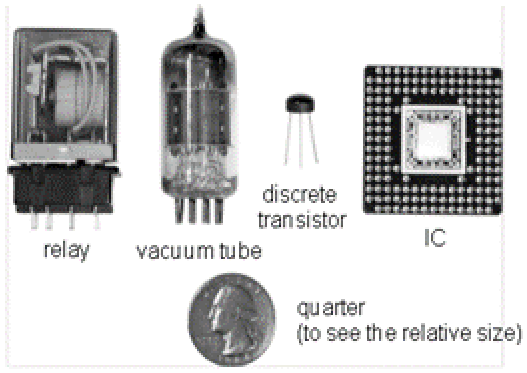
TL;DR

Problems get broken into levels of abstraction. That's a good thing as it makes it easier to do the job and/or split it up. But you need to know something about the levels above and below where you work. Today we'll talk about CMOS logic. Later we'll look above this level (computers) and later still at FPGAs.

I. Transistors and switches[i] (2.2-2.4 of the text)

Gates can, and have, been made up of various devices. Some examples include: Relays (1930s), vacuum tubes (40's), discrete transistors (that is a single transistor in a package) (50s), and integrated circuits (lots of transistors in a package) (60s). There have also been mechanical, water, and light-based switches.





Let's try to use switches to design the simplest gate—an inverter. Say we apply a "1" to the control input the switch closes (that is conducts from the source input to the output as shown in the lower diagram above) and if we apply a "0" to the control input the switch is open (as in the top part of the above figure). Given:

- One or more switches
- A resistor
- Full access to ground (0) and Vcc (1)

Can you build an inverter? (Those without Phy 240 or EECS 215 are excused!)

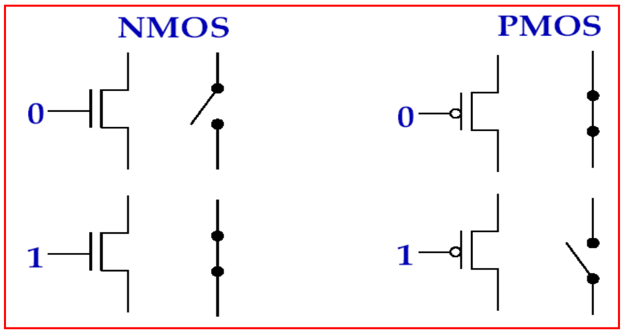
Inverter design with a switch, a resistor, and full access to ground and Vcc.

For you EE types, discuss the advantages and disadvantages of a small and large resistance. Think RC delays and power.

For everyone else, notice that the same technique could be used with, say, water. If Vcc is replaced by water under some pressure and ground by a place water can follow to the resistor just needs to be a thing that lets water slowly come in.

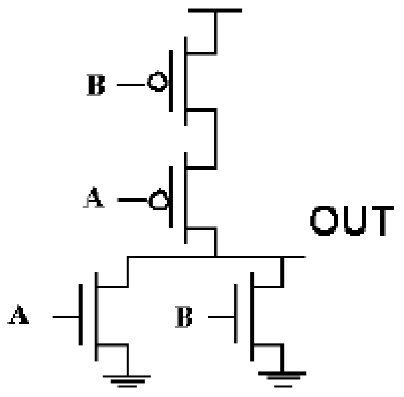
Transistors as switches

Transistors can basically act as switches. There are two types, NMOS and PMOS.



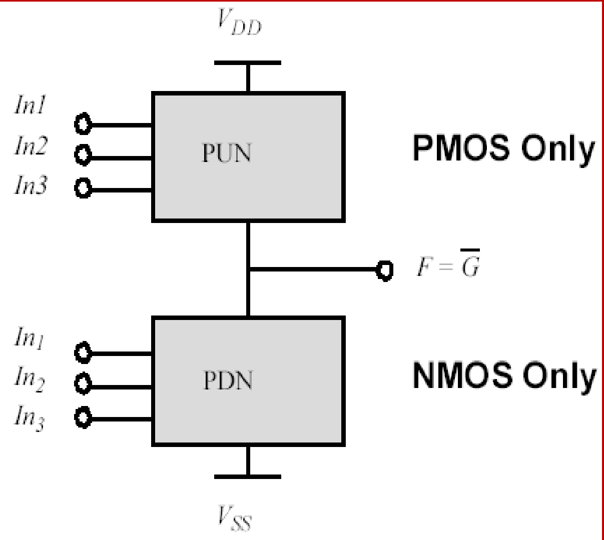
Now, let's build an inverter taking advantage of having two types of switches. Build an inverter with PMOS and NMOS transistors and no resistors.

Inverter design with NMOS, PMOS and full access to ground and Vcc.



A	B	OUT
0	0	
0	1	
1	0	
1	0	

Now, consider the above figure and write a truth table for it. What kind of gate is it?



PUN and PDN are Dual Networks

Due to the electrical properties of transistors, PMOS transistors only work well when they are connected to power (V_{DD} in the diagram to the left) and NMOS when they are connected to ground (V_{SS} in the figure on the left).

So the PUN network is a "pull-up" network and PDN is a pull-down network. In static CMOS[1] devices, all of our designs will consist of a pull-up and pull-down network as shown. The networks are "duals" of each other: exactly one will be connected to the output.

Build a NAND gate in CMOS logic. *Then* try an AND gate.

NAND gate design with NMOS, PMOS and full access to ground and Vcc.

AND gate design with NMOS, PMOS and full access to ground and Vcc.

[1] Complementary metal–oxide–semiconductor. Basically, it means uses both NMOS and CMOS transistors as described here.

More about transistors

Frankly, transistors are a lot more complex than the above discussion shows. They are analog devices and have some interesting (and complex) properties. Large parts of EECS 320, 311 and 312 deal with those properties. EECS 312 in particular worries about how to build fast, small gates out of transistors. EECS 427 deals with using those transistors in a larger context (a fairly simple processor). Most of that is far out of the scope of this class. That said; let's try to get a basic idea why we need PMOS transistors in the pull-up network and NMOS transistors in the pull-down network.

(This isn't on the exam or homework and I'm not even going to include it in the notes other than the figure on the right.)

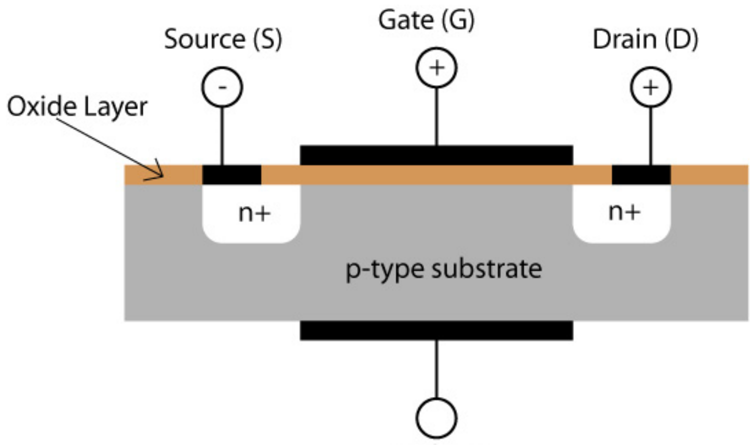


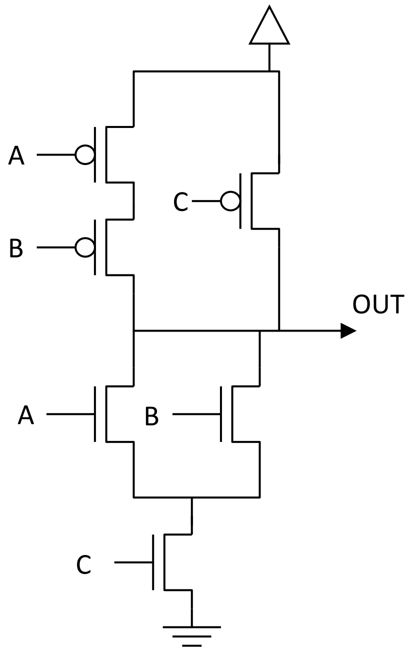
Figure 1: NMOS transistor

Summing up/practice time

Transistors are the way a modern ASIC (application specific integrated circuit) is implemented. In this lecture we've covered the standard way of building gates out of transistors (called static CMOS). We've seen why NAND and NOR gates tend to be faster than AND and OR gates.

<u>Practice problem:</u> Fill in the following truth table with either "1", "0", "Hi-Z" or "Smoke" (the last if OUT is connected to both Vcc and Ground).

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



Bonus question: Design an XOR gate using as few transistors as possible.

Higher-level abstraction (Chapter 8)

Fundamentally, we can take things like MSI devices, memories and gates and build even higher-level components. The most obvious and fundamental is a *general-purpose processor*. These devices are at the core of the modern internet age. They are covered in EECS 370, 470, 427 and 373 as well as a bit in EECS 452. If you are interested in learning the basics, chapter 8 covers this material at a fairly high level. We'll touch on it later in the term.

And a datapath problem (time allowing)

Now let's do another one. Say we have as input the signals Go, A[7:0] and B[7:0]. As outputs we have X[24:0] and Done. When Go is sampled high, you are to compute A^3+B^3. You may only use one multiplier (as large as needed but no larger) and one adder (ditto on size). When done, the output X should be set to the result and Done should be a 1. Go's value should be ignored until the cycle after Done is set to 1. The values of A and B may change after the cycle Go is first sampled high and those changes should be ignored until a new computation is requested. You should use as few MSI devices as possible.

[i] Figures taken from: Dr. David Brooks, Dr. Frank Vahid, and others.