Layers of abstraction and managing complexity.

As I've discussed throughout the class, 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.

The table on the right is one way to think about the abstractions that go into a computer system. At the top we have application programming (which is what you are doing for the project!) and at the lowest level we've got physics.

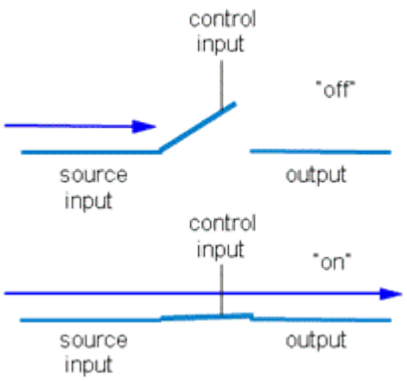
Programming	*Application*
	Operating System
	High-level language
	Assembly and machine language
Computer Organization	*Instruction set architecture (ISA)*
	Processor implementation
Digital logic	*Datapath and control*
	Combinational logic; registers
Solid State	Transistors
	Solid state physics

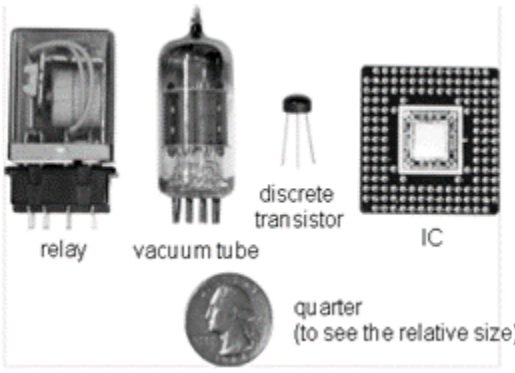
TL;DR

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. We're going to focus on the level below digital logic for a bit.

I. Transistors and switches

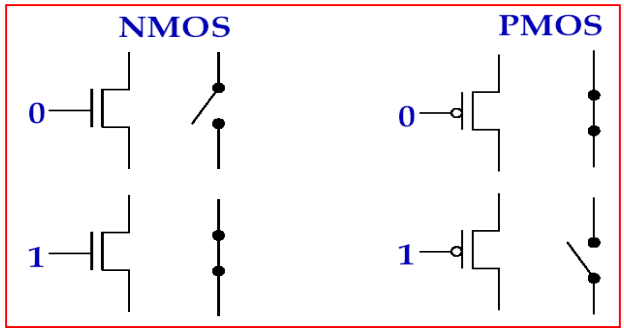
Gates can, and have, been made up of various devices. Some examples include: Relays (1930s), vacuum tubes (40's), discrete transistors (that is a single transistor in a package) (50s), and integrated circuits (lots of transistors in a package) (60s).
There have also been mechanical, water, and light-based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





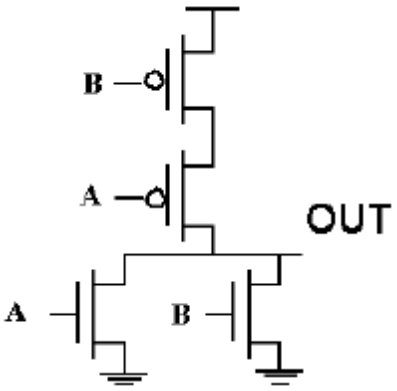
Transistors as switches

Transistors can basically act as switches. There are two types, NMOS and PMOS. NMOS transistors open if their control input is a 0 and close if their input is a 1. PMOS open if their input is a 1 and close if their input is a zero.



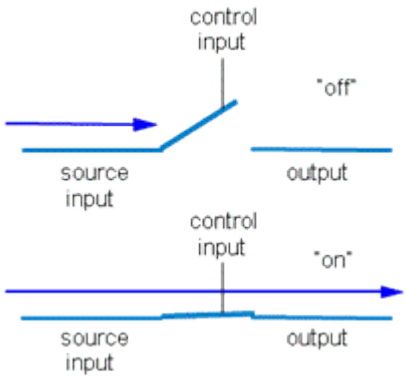
NMOS and CMOS as switches

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



A	B	OUT
0	0	
0	1	
1	0	
1	1	

Next, draw an inverter using only NMOS and PMOS transistors (as well as ground and Vcc of course).



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

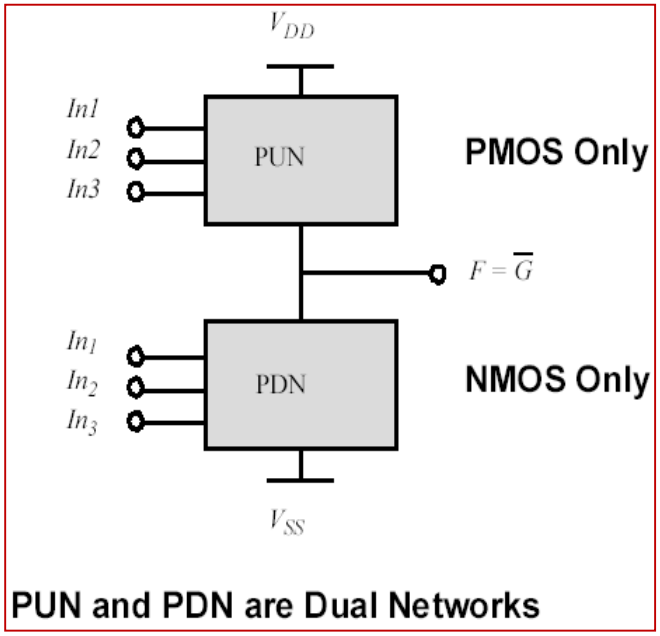
Some further restrictions

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_{NN} 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 devices will consist of a pull-up and pull-down network as shown. The networks are "duals" of each other.

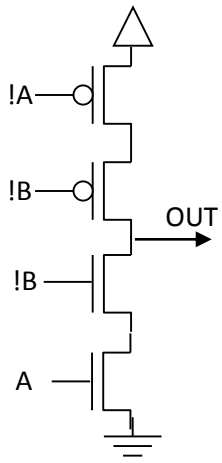
Design a NAND gate while following that restriction.

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



AND and OR gates are actually a bit more tricky. The basic theme is that we design an NAND or NOR gate and then invert the output.

What is the figure below? Recall that !A is "not A". (Also, be aware that if an output isn't connected to either Vcc or Ground, it is HiZ.)



A	B	OUT
0	0	
0	1	
1	0	
1	1	

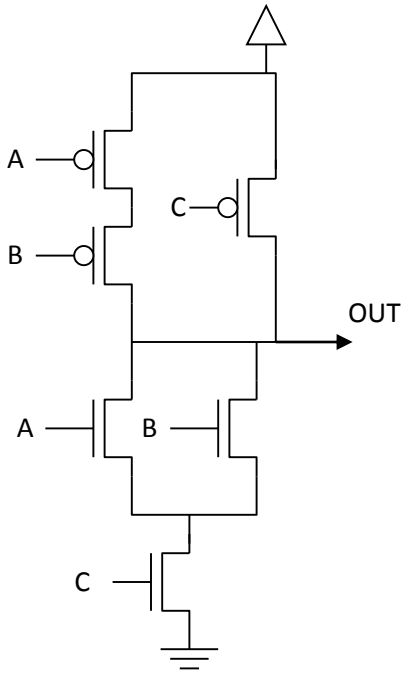
[1] Complementary metal–oxide–semiconductor. Basically means uses both NMOS and CMOS transistors as described here.

Summing up transistors and practice time

Transistors are the way a modern ASIC (application specific integrated circuit) is implemented. We've covered the standard way of building gates out of transistors (called static CMOS).

Practice problem: 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	



Next, we are going to look at how to build more complex systems including converting from a truth table to gates as well as the design of sequential logic devices.

From any truth table to gates

We can represent logic functions in a number of ways at this point. We can write logic equations, draw gates, and write truth tables. Going between these methods of representation (mostly to/from truth tables) can be interesting. Consider the following truth table. Write a logical formula that is equivalent.

A	B	Out
0	0	0
0	1	0
1	0	1
1	1	0

The basic trick is that we can take any truth table that has a single "1" and write it using an AND gate and (perhaps) some NOT gates. In this case we don't need a NOT gate for "A" because on the line where Out=1, A=1. But we do need one for B because on that same line B=0.

The next step is dealing with multiple "1s" in a truth table. In this case, we just make the gate for each individual "1" and OR them together. Try this truth table:

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

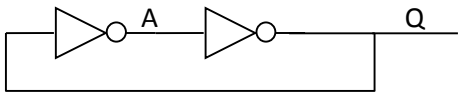
Now there may be simpler ways to represent this (in this case, a single XOR!), but this scheme will always work. Try it with the following truth table:

S	C	B	Out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

The cool thing is that because we can write any combination logic statement as a (potentially large) truth table, we can generate any combinational logic statement with just AND, OR and NOT gates!

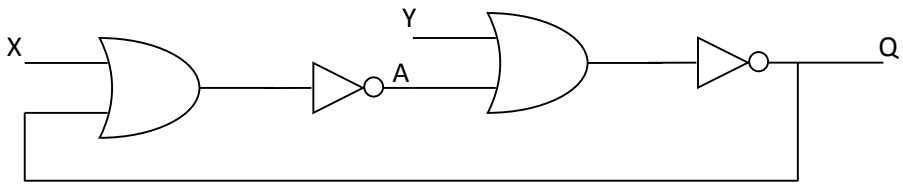
Sequential logic

So that works for combinational logic, but what about sequential logic?
One thing we have carefully avoided so far is feedback—all of our signals have gone from left-to-right. What if we don't do that?



Consider the figure on the left. It is a "bi-stable" device. That means there are two ways it could be stable. If A=0 then Q=1 and you'd expect that to hold its value (because A is then driven to 0 by Q. In the same way A=1, Q=0 is stable. So we've got two different sets of values for which this could be stable.

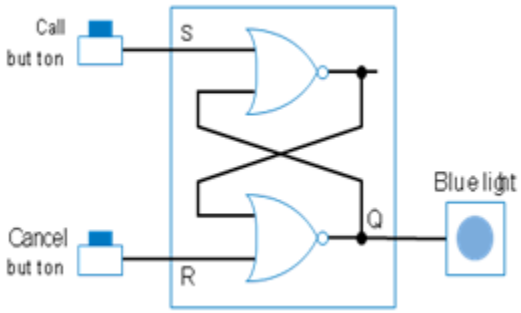
So is such a device useful? It would then *seem* like a good candidate for a memory device—after all it can keep one of two values. The problem is that we can't write to that device as both A and Q are already being driven by something else. So let's consider the following device instead



Now consider the above diagram. Recall that 0+X=X. So if X = Y = 0, those OR gates act like wires. So *in that case* the device functions exactly the same as the pair of inverters. What about the other cases?

S-R latch

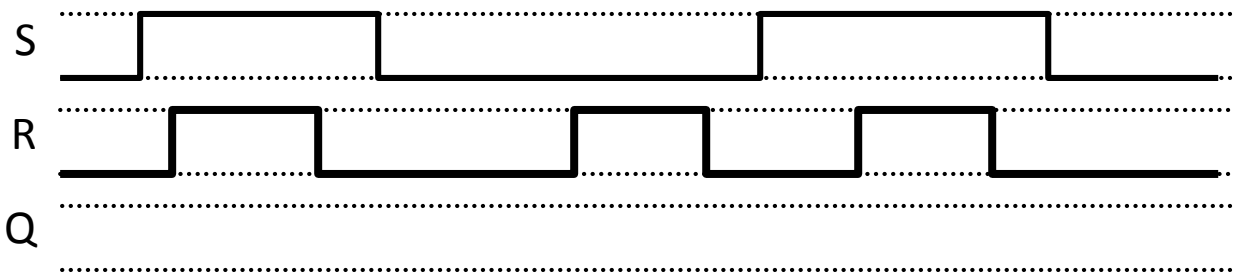
The above figure is redrawn on the right (figure from text), replacing the OR and NOT gates with NOR gates. And some wires moved around. This is the most common way to draw this device. It is called an SR latch.



Questions:

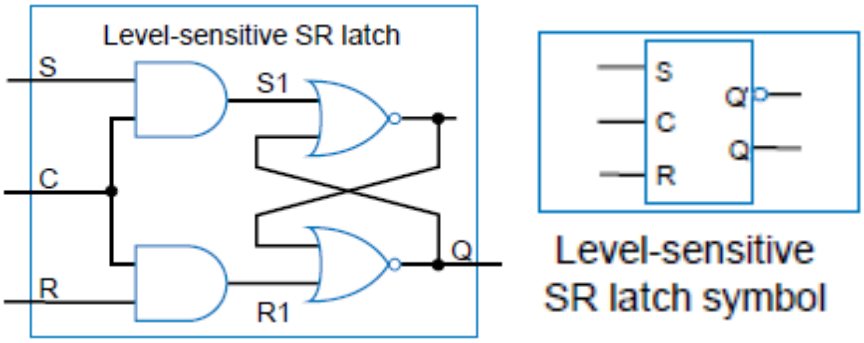
1. Fill in the truth table below
2. Complete the timing diagram.

S	R	Q
0	0	
0	1	
1	0	
1	1	

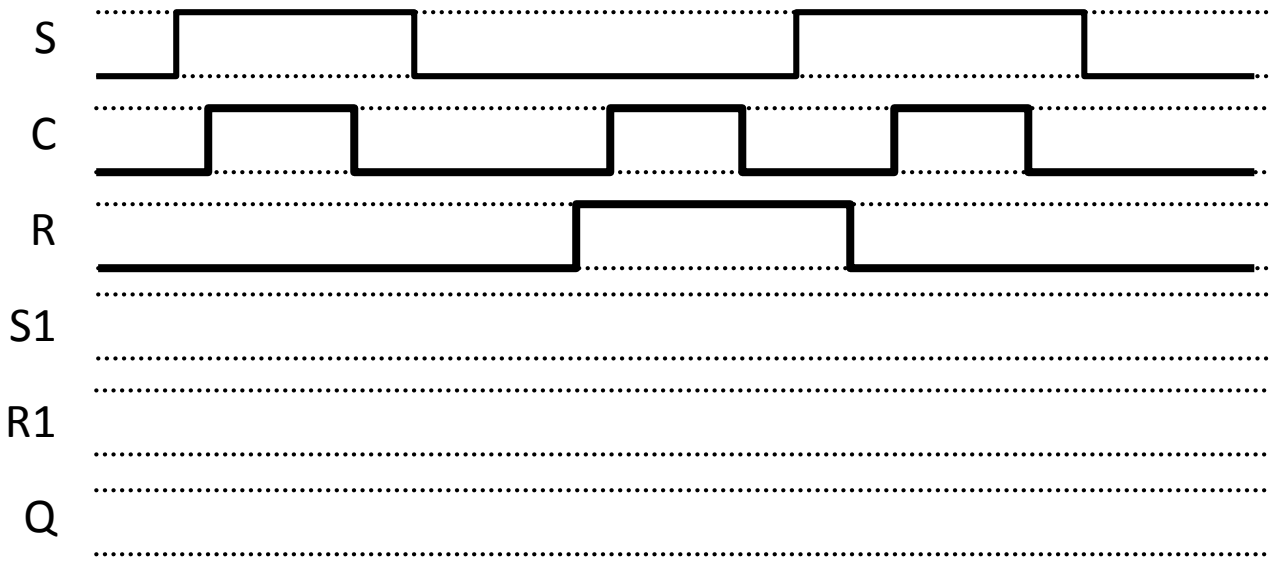


SR-latch with enable (which our text calls a level-sensitive SR latch)

One improvement: Add an "enable" or "clock". Only allow S and R to change when C=1.

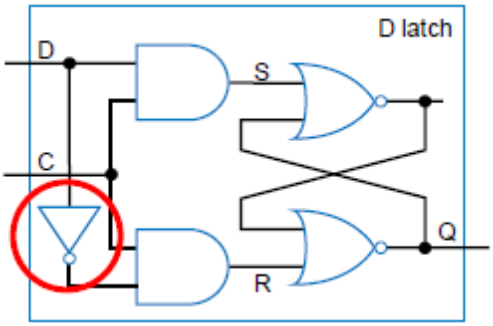


Fill in the timing diagram below.

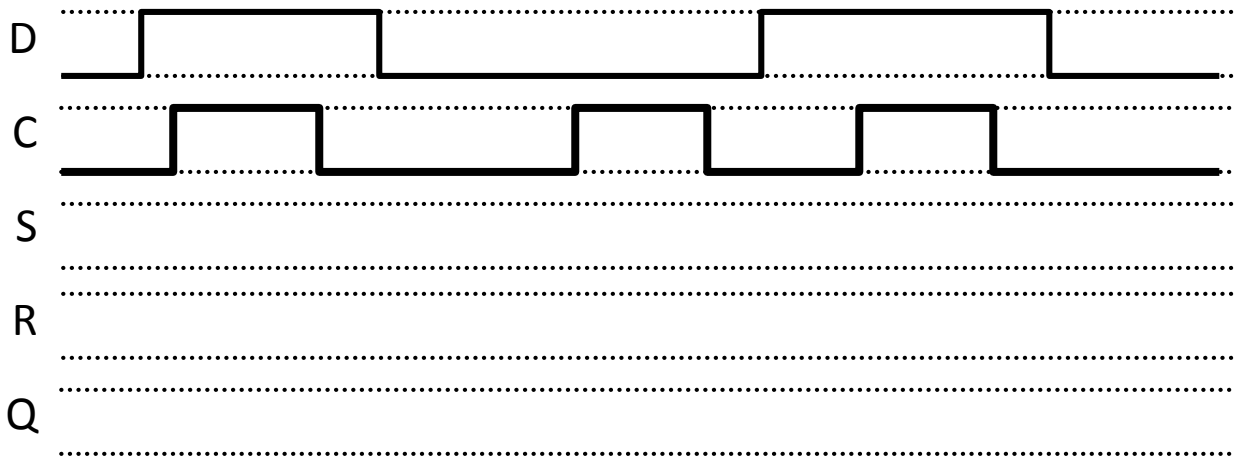


D latch

The D latch, shown below, is a very common type of latch. It has the property that S and R can't both be high at the same time.



Fill in the timing diagram below.



So this isn't edge triggered. Which is what we want for a register. We'll cover how to do that next time.

Do your peer evaluations!!!

They are due today at 7pm.

Figures in blue are taken from Vahid's book "Digital Design".