Physics 364, Fall 2014, reading due 2014-11-02.

Email your answers to ashmansk@hep.upenn.edu by 11pm on Sunday

Course materials and schedule are at http://positron.hep.upenn.edu/p364

Assignment: (a) First read (or at least skim) Eggleston's chapter 8 (Digital circuits and devices, pages 200–233). For this week, you can stop at page 220; we'll cover the rest in later weeks. (b) Then read through my notes (starting on next page), which directly relate to what we will do in Lab 19. (c) Then email me your answers to the questions below.

1. What is the key advantage of representing information in digital form?

2. Write out the truth table for a two-input NAND gate. Is it possible to combine several NAND gates to perform the two-input AND function? Is it possible to combine several AND gates to perform the two-input NAND function? (NAND and NOR are known as "universal gates," because either one of them can be used to implement arbitrary logical functions.)

3. What is the key property of a flip-flop (also called an "information register" in the textbook) that makes it so useful for solving a whole new class of problems?

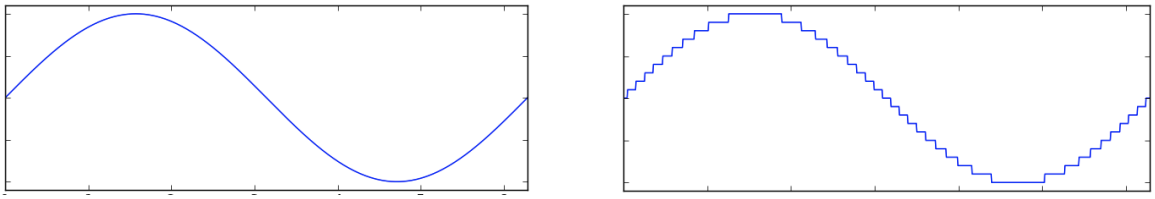
4. Is there anything from this reading assignment that you found confusing and would like me to try to clarify? If you didn't find anything confusing, what topic did you find most interesting?

5. How much time did it take you to complete this assignment? Also, I continue to welcome suggestions for ways in which I might adapt the course to help you to learn most efficiently.

Lab 19 begins the **digital** section of the course, covering roughly the last $\frac{1}{3}$ of the semester. Lab 19 itself will be a very brief introduction to digtial logic. Then we will spend about two weeks on Arduino programming. Finally the last two weeks of the semester will include four days on flip-flops programmable logic (FPGAs). A key goal of the first $\frac{2}{3}$ of the course was for you first to see how handy opamps can be, and then to learn enough about transistors to understand in principle how an opamp works. A key goal for this last $\frac{1}{3}$ will be for you first to learn to program tiny computers (called Arduinos) to perform various tasks, and in parallel to learn enough about digital logic to understand in principle how a computer works — how to go conceptually from FETs to logic gates to flip-flops to memories, and then on to state machines that go step-by-step through a sequence of operations, and then finally to a microprocessor that reads and executes instructions stored in its Random Access Memory.

So far in this course we have dealt with **analog** signals: an analog signal (voltage, current, wave intensity, etc.) is a continuous quantity, represented by a real number. If a and b are both allowed values for an analog signal, then $\frac{1}{2}(a+b)$ is also an allowed value, as is any other value between a and b.

By contrast, a **digital** signal takes on only **discrete** values. Any allowed value of a digital signal can be represented by an integer, once some real-valued scale factor is chosen. The left figure below shows an analog signal vs. time, and the right figure shows a digital representation of the same signal, in which only discrete signal values are permitted. (At this stage, time is still a continuous variable in both graphs.)

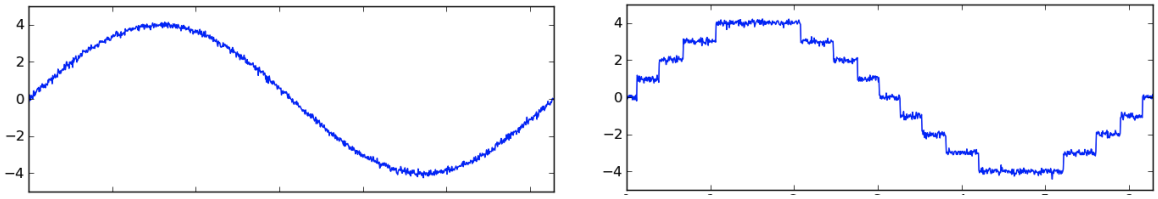


Many interesting quantities are inherently discrete in nature: your bank account balance (quantized in hundredths of a dollar), your birthday (quantized in days), the number of long-sleeved black jersey shirts you own, etc. Others, like voltages, currents, wave intensities, etc., are more naturally represented as continuous quantities, but can be approximated by a suitable choice of discrete values.

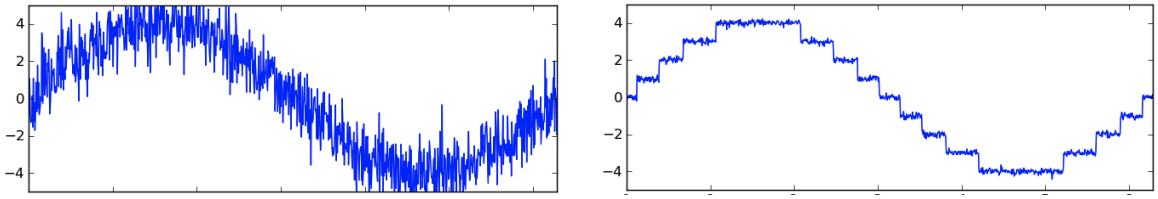
Why would you want to replace the continuous analog signal in the above-left graph with the discrete digital signal in the above-right graph? There are basically two reasons. First, **digital values can be replicated perfectly**, while analog values always degrade when copied or transmitted. Second, **digital values can be manipulated by a digital computer** such as your notebook computer, an Arduino board, or the processor on your smartphone.

The reason why analog signals always degrade with replication is that any transmis-

sion or replication process will superpose some random noise on top of the desired signal. In the left and right graphs below, I superpose a small random noise voltage on top of an analog signal (left) and a digital representation of that signal (right).



While it is easy for me to remove the noise from the digital signal (i.e. recovering the original digital signal) just by rounding each value to the nearest integer, I have no way *a priori* to remove the noise from the analog signal, unless I am given additional information about the original form of the signal. Suppose I want to send each of these two signals such a great distance that the message will need to be repeated (e.g. amplified and retransmitted) 100 times between the origin and the destination. In the case of the digital signal, as long as the noise added at each intermediate step is much smaller than the spacing between permitted signal values, I will be able at each stage to recover the original digital message before retransmission. Hence, the 100th copy will look no worse than the 1st copy. By contrast, the analog signal steadily degrades at each step. After 100 retransmissions, the two signals look like this:



The analog signal continues to degrade with each successive copy, while the digital copy always permits the original signal to be recovered, as long as the noise level between copies is small enough to allow every point to be assigned unambiguously to the correct discrete signal value.

Imagine seeing a 100th-generation copy (i.e. a copy of a copy of a copy of . . . a copy) of each of the following:

- An old family photo (where copy n is made by running copy $n-1$ through a xerox machine): you might have occasionally seen really old course handouts that suffer from a similar problem, because the original document is no longer available for re-printing.
- A house key (where copy n is made by taking copy $n-1$ to the hardware store): typically the key doesn't work unless you jiggle it for a long time in the lock.
- A 1970s cassette audio tape (where copy n is made by running copy $n-1$ through the sort of boombox that has two tape decks): it will sound like music

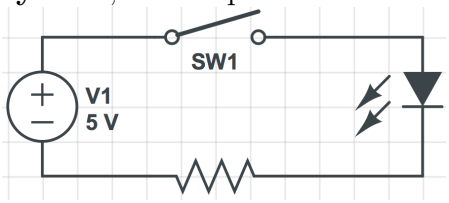
played over a poor telephone connection.

- Your signature (where copy n is made by a forger who is only able to look at copy $n-1$): it is unlikely to show any resemblance to your original signature.
- A sidewalk artist's drawing of your face (where copy n is made by a sidewalk artist who is only able to look at copy $n-1$, not at you): it probably won't look anything like you.
- A JPEG photograph on your computer (where copy n is made by copying byte-by-byte the contents of the disk file for copy $n-1$ to a new disk file): it will be a perfect copy, unless a hardware failure occurs.
- The lock vendor's serial number for your house key (where copy n is made by reading a 10-digit number from the slip of paper $n-1$ and writing down those ten digits onto a new slip of paper): unless someone writes down the wrong number, the information will be perfectly replicated.
- An MP3 file on your computer (where copy n is made by copying the contents of the disk file for copy $n-1$ to a new disk file): it will be a perfect copy.
- A typewritten paragraph of text (where copy n is made by re-typing copy $n-1$): as long as each typist proofreads his or her work, the 100th copy will be perfect.
- A sequence of 1000 DNA base pairs (each of which takes on one of four possible values: A, C, T, G): unless there is a transcription error or a mutation at some intermediate stage, the 100th copy will be perfect.
- A sequence of 100 decimal digits: again, can be copied perfectly, unless there is human error.
- A data file on your computer (where copy n is made by copying byte-for-byte the contents of copy $n-1$): excepting hardware failures, the copy will be perfect.

To *digitize* (could be called "discretize") means that we use a finite set of integers to represent the possible values of the physical quantity of interest. The initial discretization discards some information, but once the information is in discrete (digital) form, it can be reproduced perfectly, as long as any noise added by the copying process is much smaller than the step size between allowed discrete values.

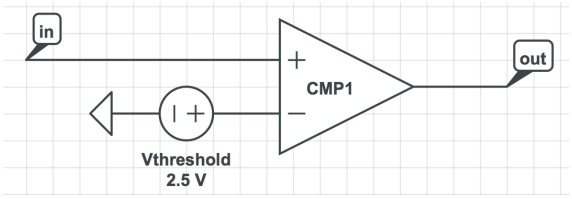
Next week, we will talk at length about methods and consequences of converting back and forth between continuous analog signals and discrete digital signals. For now, let's move on to discussing the second key property of digital information: that it can be processed by a digital computer (and the digital logic circuits of which a digital computer is composed).

First let's think about various ways that integer data can be represented. To ten-fingered humans, the most natural representation for integer data nowadays is a sequence of decimal values, e.g. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17. An N-digit decimal number is written in the form $a_{N-1}a_{N-2}\ldots a_2a_1a_0$, which represents the quantity $\sum_{n=0}^{N-1} a_n 10^n$. Each base-ten numeral can take on ten discrete values. But many useful devices, such as the following device for illuminating an LED, have only two discrete states: ON and OFF. So it turns out to be useful to represent integers in **binary** form, as a sequence of ones and zeros.

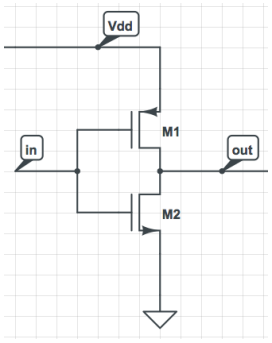


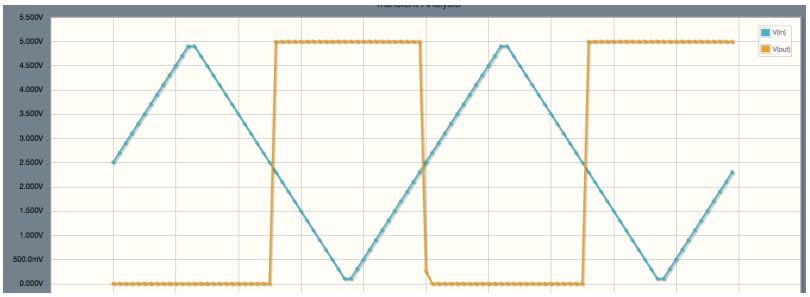
An N-bit binary number is written in the form $a_{N-1}a_{N-2}\ldots a_2a_1a_0$, which represents the quantity $\sum_{n=0}^{N-1} a_n 2^n$. Each base-two numeral can take on only two discrete values: 0 or 1. Thus, the sequence $0\ldots17$ written in decimal above would look like this in binary: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, 10001. I could represent any of these numbers (in fact any number $0 \le n \le 31$) using the ON/OFF states of five LEDs.

Binary is convenient because you can, in principle, just use a simple comparator to decide which of the two possible states you are looking at. For example, I could imagine using a circuit like the one shown below (with a +5 V power supply for the comparator) to assign inputs $0 \le V_{\text{in}} < 2.5$ V to the OFF state and inputs $2.5\text{ V} < V_{\text{in}} \le 5$ V to the ON state.



It turns out to be quite easy (even easier than the comparator circuit above would suggest) to build circuits that are either ON or OFF, with no in-between. For example, the **CMOS inverter** shown below (left) uses just one n-channel (on the bottom) and one p-channel (on the top) MOSFET to implement the logical **NOT** function. (It's basically a CMOS push-pull with the nMOS and pMOS FETs interchanged.) Using power supply voltage $V_{DD} = +5$ V to represent ON and ground to represent OFF, this circuit will output the logical complement of its input: if the input is ON, the output will be OFF, and vice versa.



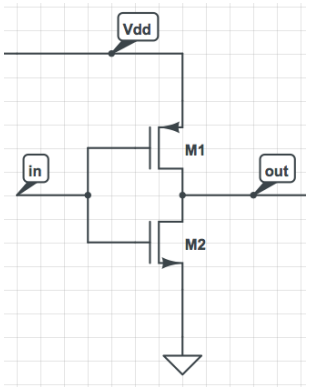


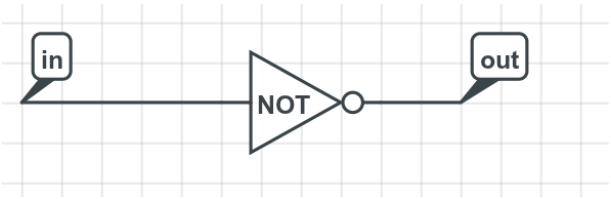
The right-hand figure above shows V_{out} as a function of V_{in} for this inverter. Normally V_{in} would be either very close to 0 V or very close to +5 V, but in this case let's look to see where the transition occurs. To be fair, the only reason that the transition happens right at $V_{\text{in}} = 2.5$ V is that I chose a pair of MOSFETs with $V_{\text{threshold}} = 2.5$ V. If you used MOSFETs whose V_{th} varied a bit from part to part, the exact threshold would vary somewhat from 2.5 V. Real CMOS logic devices are designed to accept any $V_{\text{in}} > 3.7$ V as ON and any $V_{\text{in}} < 1.3$ V as OFF. The region 1.3 V $< V_{\text{in}} <$ 3.7 V is considered ambiguous, with resulting V_{out} not guaranteed to take on one value or the other. Conversely, a CMOS logic device guarantees $V_{\text{out}} > 4.7$ V for the ON state and $V_{\text{out}} < 0.2$ V for the OFF state. Strictly speaking, the +5 V and 0 V states are called "HIGH" and "LOW," respectively, rather than "ON" and "OFF," for reasons alluded to in the next footnote.

There is a formal **Boolean algebra** for manipulating variables whose values can take on only two values. That algebra is nicely summarized in Eggleston's table 8.3 (page 209), so I won't repeat it here, except to note that he writes $A + B$ to mean "A OR B," writes $A \cdot B$ to mean "A AND B," and writes $\overline{A}$ to mean "NOT A," which is all pretty standard. Normally "1," "ON," "TRUE," "HIGH," and "+5 V all refer to one state, while "0," "OFF," "FALSE," "LOW," and "0 V" all refer to the other state.[1]

As noted above, the CMOS implementation of an **inverter** looks like the left figure below. The schematic symbol for an inverter is shown on the right figure below. The triangle indicates buffering/following a signal, and the circle indicates logical inversion. So the function performed by this circuit is OUT = $\overline{\text{IN}}$.

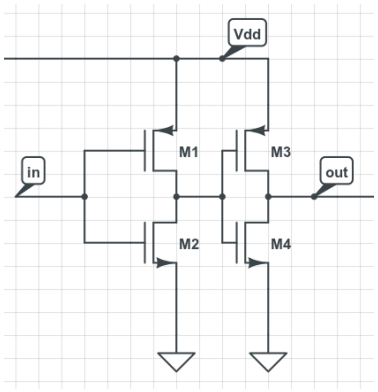
[1]There are two exceptions, which I will just gloss over now, and will mention in more detail later. The first is that not all digital logic uses 0 V and +5 V to represent the two logic states. For example, more recent CMOS logic uses either +3.3 V or +2.5 V instead of +5 V; some older families of logic circuits use both positive and negative voltages. Second, some signals are "active low," meaning that they sit at +5 V (or whatever value is used for V_{DD}) when nothing is happening, and they are momentarily moved to 0 V to indicate a special event, e.g. the completion of a requested operation. In this case, the pin would be labeled "$\overline{\text{DONE}}$" (pronounced "done bar") to indicate active-low, as opposed to being labeled "DONE" if it were (the usual case) active-high. You may see a few examples of active-low input/output pins when we work with Arduinos.

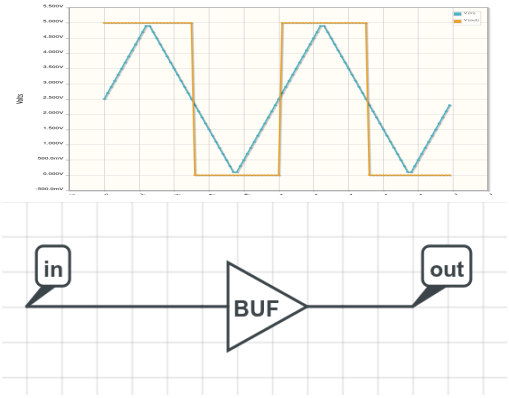




Here's how the CMOS inverter works: When $V_{\text{in}} \approx 0$ V, the nMOS FET (lower) is cut off (nonconducting) and the pMOS FET (upper) is active (conducting), so $V_{\text{out}} \approx V_{\text{DD}}$. When $V_{\text{in}} \approx V_{\text{DD}}$, the nMOS FET (lower) is active (conducting) and the pMOS FET (upper) is cut off (nonconducting), so $V_{\text{out}} \approx 0$ V.

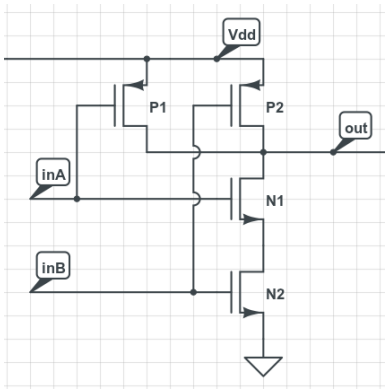
If we put two CMOS inverters in a row, we get a CMOS **buffer**, shown below (left), whose digital schematic symbol is shown below (right). Also shown on the right is V_{out} as a function of V_{in}, for the (atypical) case of V_{in} varying continuously. The logical function performed is OUT = IN. The main point of a logical buffer is largely the same as that of an analog follower: to make a stronger copy of the input signal. For example, if you need to send a digital signal a very long distance, you might want to buffer it several times along the way, so that the signal never degrades enough to become ambiguous. A more common use of logic buffering is to permit a logic signal to supply a lot of current, e.g. to illuminate an LED or to fan out to a large number of downstream logic devices. (Even if these logic devices have inputs that draw no DC current (by virtue of being the "gate" terminals of FETs), the devices and wires have a finite capacitance, requiring some current to flow during each transition when the state changes between LOW and HIGH.)

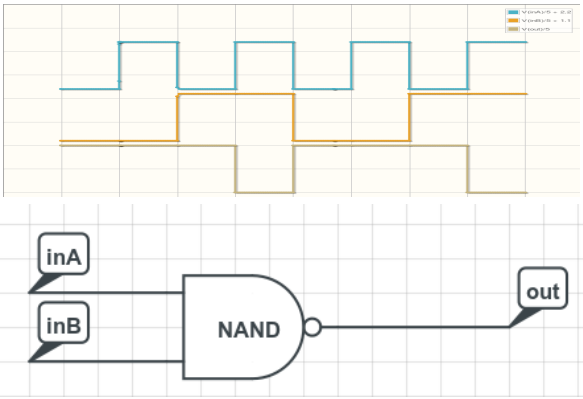




The figure below shows a CMOS **NAND** circuit. On the left is the implementation using two pMOS and two nMOS FETs. On the lower-right is the schematic symbol for the NAND function. (The D-shaped symbol represents AND and the bubble on the output represents logical inversion.) The upper-right shows (top to bottom)

input A, input B, and output, vs. time, where I've offset the three traces from each other vertically so that you can distinguish them. The logical function performed is OUT $= \overline{A \cdot B}$.





If both inputs are HIGH, then nMOS FETs N_1 and N_2 are both active, while pMOS FETs P_1 and P_2 are both cut off; the result is $V_{\text{out}} = 0$ V. If either input goes LOW, then at least one of N_1 or N_2 is cut off, and at least one of P_1 or P_2 is active; the result is $V_{\text{out}} = V_{DD}$. The result is HIGH unless both inputs are HIGH. By the way, the "NAND circuit" is technically called a "NAND gate." The graph shown below (borrowed from D.V. Bugg's electronics book) motivates why a circuit performing a logical function might be called a "gate:" you can see input A allowing or disallowing the passage of a stream of pulses from input B, as if it were a kind of door that you could open or close. Don't be confused by the re-use of the word "gate." A circuit that performs a NAND operation is called a "NAND gate." That usage is completely unrelated to the fact that one terminal of a FET is called the "gate."

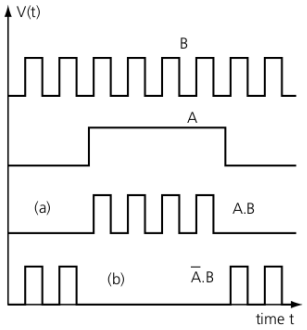
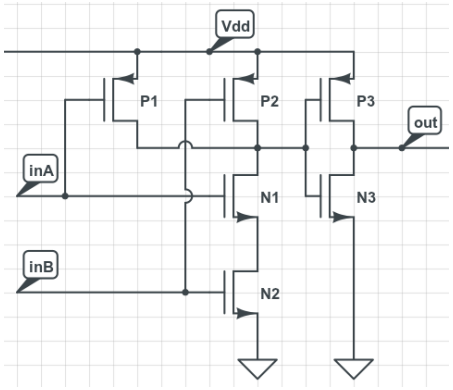
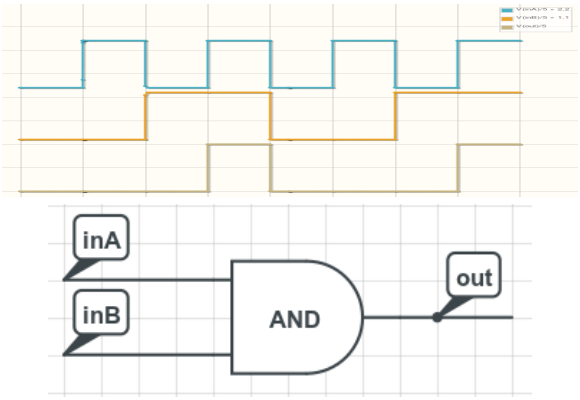


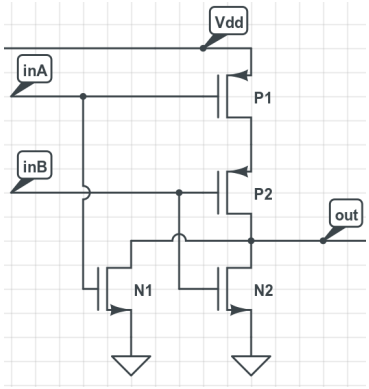
Fig. 12.4. (a) $A.B$ and $\overline{A}.B$.

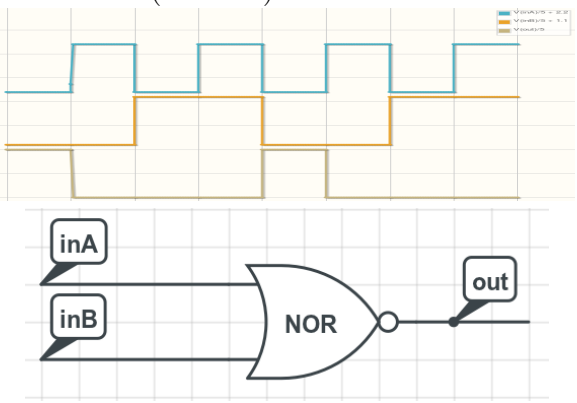
Remarkably enough, the way to make a CMOS **AND** gate is to tack an inverter onto the end of a CMOS NAND gate, as shown in the figure below (left). On the upper-right are (top to bottom) input A, input B, and output, again vertically offset so that you can see them. On the lower-right is the schematic symbol for an AND gate. It is just like the NAND symbol, but without the bubble on the output. The logical function performed is OUT $= A \cdot B$. You can see that OUT is only HIGH if both A and B are HIGH.



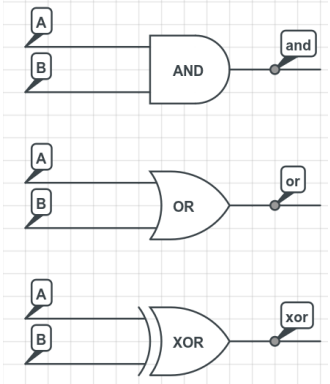


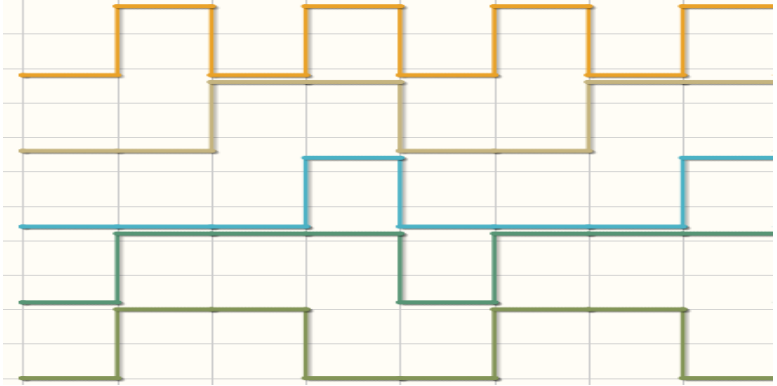
The figure below shows a CMOS **NOR** gate. On the left is the implementation using two pMOS and two nMOS FETs. On the lower-right is the schematic symbol for the NOR function. (The Star-Trek-esque symbol represents OR and the bubble on the output represents logical inversion.) The upper-right shows (top to bottom) input A, input B, and output, vs. time, where I've offset the three traces from each other vertically so that you can distinguish them. The logical function performed is OUT $= \overline{A + B}$. The output is LOW if either A or B (or both) is HIGH.





I think you can easily imagine tacking an inverter onto the end of a NOR gate to form an OR gate. Another interesting function is XOR (exclusive or). XOR is HIGH if either A or B (*but not both!*) is high; another way to look at it is that XOR is HIGH if $A \neq B$, i.e. if A and B differ.





The above-left graph shows the circuit symbols for AND, OR, and XOR. The way I remember AND vs. OR is that the OR symbol looks like an AND symbol that has been coerced into being more accomodating. The way I remember the XOR symbol is that it looks like an OR symbol with an additional barrier. The above-right graph shows (from top to bottom) input A, input B, $A + B$ (A OR B), $A \cdot B$ (A AND B), and finally $A \oplus B$ (A XOR B).

It turns out that you can make an inverter out of a NAND by wiring its two inputs together: $\overline{A \cdot A} = \overline{A}$. By putting this inverter after a NAND, you have an AND. If you invert both inputs of a NAND, you have OR, because $\overline{\overline{A} \cdot \overline{B}} = A + B$. If you invert the output of that OR, you have NOR. And so on. So it turns out that using only NAND gates, you can build up any desired logical function. It turns out that you can also do this using only NOR gates. So NAND and NOR are called "universal gates," because you could in principle build up any desired logical function by using an arbitrary number of this one building block. If you have a recipe for making unlimited quantities of NAND gates, you can perform any digital function you wish.

I really want to move on to talking about mathematical operations such as addition, subtraction, and multiplication. I also want to mention two's complement representation of negative numbers and hexadecimal representation of binary numbers. I also want to hint at how "real" numbers are stored in floating-point representation within a computer. And then I absolutely want to discuss so-called "sequential logic," which uses **flip-flops** to remember previously stored information. Eggleston's chapter mentions all of these things, but I think I can do better. In any case, I'm out of time, so I will have to write up these descriptions for you for next week's reading. The whole new class of problems enabled by flip-flops is the ability to have a circuit's output depend not just on the present values of the inputs, but to have it also depend on previous inputs. A flip-flop gives a circuit the ability to remember previous inputs (or outputs). This makes it possible, for example, to count from zero to ten. (How can you count from zero to ten without remembering the last number that you counted?) It also makes it possible to build a machine that steps through a sequence of states, such as a traffic light, a washing machine, or a coin-counting vending machine. This idea, in turn, can be generalized to the idea of a computer that executes a stored sequence of instructions. During the next few weeks, the elucidation of these ideas will be interleaved with the Arduino material that you will learn.