Physics 364, Fall 2014, reading due 2014-11-16.

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

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

Assignment: (a) First read this week's notes, starting on the next page. (b) Then email me your answers to the questions below.

1. When two 16-bit numbers are added together in combinational logic (using a bunch of AND, OR, XOR gates), why do some bits of the sum require more time to update (time w.r.t. the time at which the input bits are updated) than others?

2. What role does the clock play in a synchronous digital logic circuit?

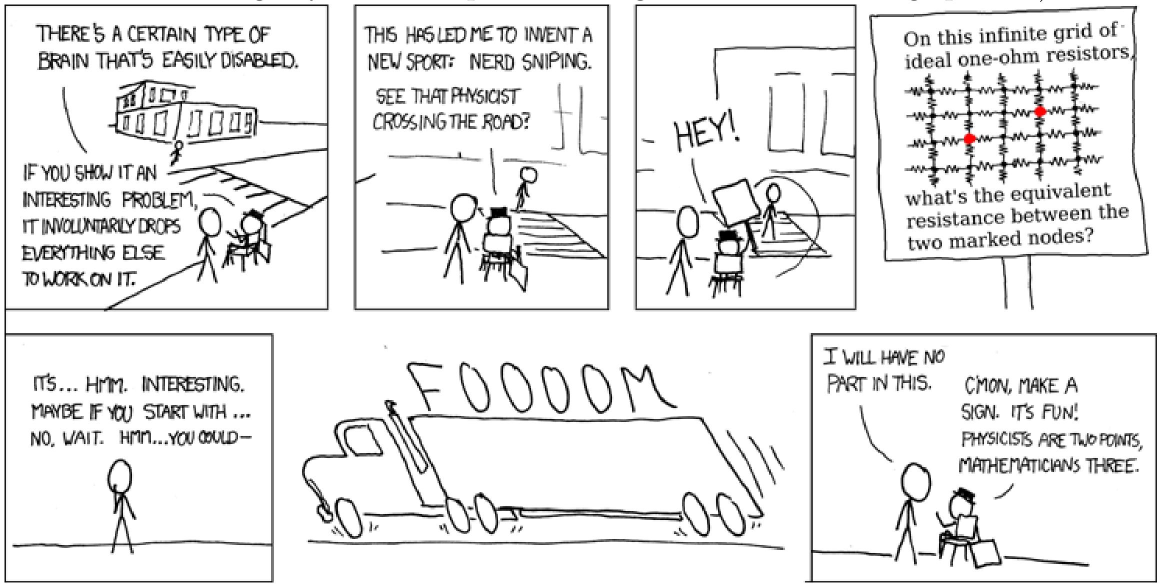
3. What are the roles of the **D**, **Q**, and **clk** pins on an edge-triggered D-type flip-flop? What happens on the $0 \to 1$ transition of the clock?

4. Convert these three numbers into binary, decimal, and hexadecimal: 255_{10}, 80_{16}, 00001000_2. Optional (avoid this one if you are vegetarian): convert the binary number 1101 1110 1010 1101 1011 1110 1110 1111 (base 2) into hexadecimal (base 16).

5. 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?

6. How much time did it take you to complete this assignment?

My dad emailed me this cartoon (from `xkcd`), which reminded me that I had once spent an entire winter break thinking about a similar infinite-grid-of-one-ohm-resistors problem. Since it never occurred to me to solve it using linear superposition, I instead wrote a computer program to find the answer numerically. (This problem was given to me as a challenge by the same person who gave me the four-bugs puzzle.)


THERE'S A CERTAIN TYPE OF BRAIN THAT'S EASILY DISABLED.
IF YOU SHOW IT AN INTERESTING PROBLEM, IT INVOLUNTARILY DROPS EVERYTHING ELSE TO WORK ON IT.
THIS HAS LED ME TO INVENT A NEW SPORT: NERD SNIPING.
SEE THAT PHYSICIST CROSSING THE ROAD?
HEY!
On this infinite grid of ideal one-ohm resistors, what's the equivalent resistance between the two marked nodes?
IT'S... HMM. INTERESTING. MAYBE IF YOU START WITH ... NO, WAIT. HMM...YOU COULD—
FOOOOM
I WILL HAVE NO PART IN THIS.
C'MON, MAKE A SIGN. IT'S FUN! PHYSICISTS ARE TWO POINTS, MATHEMATICIANS THREE.

Let's step back for a moment for an overview of where the course has gone. After a quick introduction to the lab equipment and some basic components (resistors, capacitors, inductors, diodes, LEDs), we studied the remarkable things one can do with opamps (voltage amplification, current amplification, mathematical operations on voltage signals, ...). Then we learned enough about transistors to have some feeling for (though not a detailed knowledge of) what makes an opamp's amazing behavior possible: we built and analyzed a transistor-based differential amplifier that had reasonably high gain, and then we built a multi-stage high-gain differential amplifier that functioned as a simplified home-made opamp. We didn't study the inner workings of bipolar transistors in any detail, but the working principle of field-effect transistors was straightforward enough to illustrate how a small signal can be used to control a much larger signal (voltage or current). So in essence, we decomposed the opamp into building blocks that we can understand.

Most of our study of opamps and transistors concerned *linear* circuits, in which a small change in input signal yields a proportional change in output signal; an exception was our study of comparators, whose output takes on only two values — one value for $V_{\text{in}} < V_{\text{threshold}}$ and another value for $V_{\text{in}} > V_{\text{threshold}}$. When we studied field-effect transistors, we started out with linear circuits (amplifiers, followers), then moved on to using FETs (built into the DG403 component) as *analog switches*, in which a small control signal opens or closes a circuit, analogous to your finger flipping a light switch on and off — for analog switch applications, the controlled circuit is an analog signal, but the control signal itself is digital: indicating either ON or OFF.

Then we used CMOS FETs to make *digital logic gates,* whose inputs and outputs take on only two possible values: LOW (near ground) or HIGH (near V_{DD} (or V_{CC}), the positive supply voltage, typically +5 V). In Lab 19, we first used push-button switches then used CMOS FETs to implement a few logic gates. So I think you can now understand how logic gates are implemented with CMOS FETs and how these gates could be used, for example, to make a seatbelt buzzer sound IF [car engine is on] AND [([driver is present] AND NOT [driver seatbelt is fastened]) OR ([passenger is present] AND NOT [passenger seatbelt is fastened])].

Finally, after that very brief introduction to the digital world of ones and zeros, we dove into programming tiny Arduino computers to carry out a variety of tasks, with the goal of giving you a flavor for the low-level mechanisms involved in making a computer interact with the real world. You made your own computer blink LEDs, respond to button presses, measure time intervals, and synthesize waveforms. You heard some of your classmates' Arduinos sing familiar tunes — in some cases quite loudly after amplification with a transistor-based circuit that you know how to build. You taught your computer to measure analog signals, converting a real-world voltage into a proportional integer (represented with ones and zeros) inside the computer, and in Lab 22 you will make your Arduino periodically pulse an electromagnet, to provide a driving force to a harmonic oscillator, whose motion your Arduino will measure by reading out an accelerometer. OK, we didn't build our own iPhone, but we got some sense of how computers can interact with the physical world.

For the rest of the term, I want to try to fill in the conceptual gap between basic logic gates (that can e.g. output the NAND of two inputs) and a simple microprocessor (that can do the sorts of things an Arduino does). I hope to show you enough of the building blocks of digital logic to convince you that you understand, at least in principle, how a bunch of humble transistors can work together to form something as capable as a computer.

Let's get started.

In Lab 19, you saw how complementary pairs of nMOS and pMOS FETs (CMOS = "complementary MOS") could be used to implement an inverter, a NAND gate, an AND gate, a NOR gate, etc. You can combine these to form "exclusive OR" like this:

$$A \oplus B = (A + B) \cdot \overline{(A \cdot B)}$$

or spelled out in English,

$$A \text{ XOR } B = (A \text{ OR } B) \text{ AND NOT } (A \text{ AND } B)$$

or in the syntax of mathematical logic,

$$A \oplus B = (A \vee B) \wedge \overline{(A \wedge B)}$$

or in C-like syntax, `(A ^ B) == (A | B) & !(A & B)`. (It's annoying that there are so many different and conflicting sets of symbols to represent the same operations.) It turns out that combinations of these basic logic functions can implement an arbitrary *truth table,* i.e. an arbitrary mapping of some set of input ones and zeros to some desired corresponding set of output ones and zeros.

For example, suppose that I want a circuit that can add two 4-bit integers a and b: $s = a + b$, where in this case the "+" sign means addition, not the OR operation.[1] We have $0 \le a \le 15$, $0 \le b \le 15$, and $0 \le s \le 30$, so we need 5 bits to represent the sum of two 4-bit integers. We write $a = 2^3 a_3 + 2^2 a_2 + 2^1 a_1 + 2^0 a_0$, etc., so e.g. s_i represents bit i of the sum.

$$\begin{array}{ccccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline s_4 & s_3 & s_2 & s_1 & s_0 \end{array}$$

When we add a_0 and b_0, there are three possibilities: 0, 1, or 2. Remember that s_0 (a binary digit, or "bit") is only allowed to be 0 or 1. So if the answer is 2, then we write $s_0 = 0$ and we carry the 1 into the next column. Let's define the carry bits to be c_0, c_1, c_2, c_3, and re-write the sum like this:

$$\begin{array}{ccccc} c_3 & c_2 & c_1 & c_0 & \\ & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline s_4 & s_3 & s_2 & s_1 & s_0 \end{array}$$

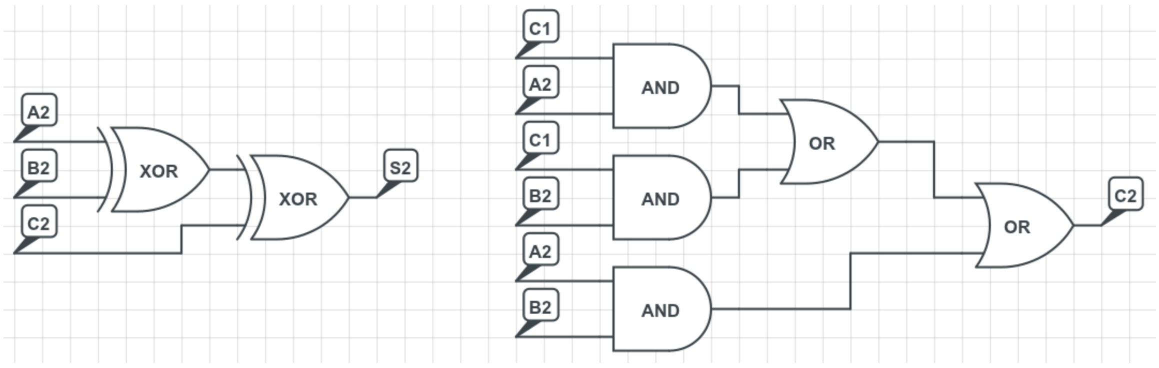
Now we can write boolean logic equations for s_0 and c_0 like this: $s_0 = a_0 \oplus b_0$, and $c_0 = a_0 \wedge b_0$. The sum bit s_0 is 1 if one (but not both) of a_0 and b_0 is 1: that's an XOR operation. The carry bit c_0 is 1 if both a_0 and b_0 are 1: that's an AND operation.

The equations for s_1 and c_1 are complicated by the possibility that c_0 might be 1. We want s_1 to be 1 if an odd number of (c_0,a_1,b_1) are 1, otherwise 0. And we want c_1 to be 1 if two or more of (c_0,a_1,b_1) are 1. We can do that with this boolean logic: $s_1 = c_0 \oplus [a_1 \oplus b_1]$, and $c_1 = [(c_0 \wedge a_1) \vee (c_0 \wedge b_1)] \vee (a_1 \wedge b_1)$. I wrote the square brackets [] only to emphasize that you can get the answer using ordinary two-input logic gates (AND, OR, XOR); but in fact there exist n-input AND, OR, and XOR gates. (The XOR of n inputs is 1 iff an odd number of inputs are 1.)

We could continue: $s_2 = c_1 \oplus a_2 \oplus b_2$ and $c_2 = (c_1 \wedge a_2) \vee (c_1 \wedge b_2) \vee (a_2 \wedge b_2)$, and then $s_3 = c_2 \oplus a_3 \oplus b_3$ and $c_3 = (c_2 \wedge a_3) \vee (c_2 \wedge b_3) \vee (a_3 \wedge b_3)$, and finally $s_4 = c_3$.

Just to emphasize that these equations represent logic gates, which you now know how to build out of CMOS transistors, the figure below shows graphically the equations for s_2 and c_2.

[1] Note to self: next time consistently use $\vee$ and $\wedge$ instead of $+$ and $\cdot$ for OR and AND, to avoid this ambiguity when talking about actual addition. Or else just use C-like syntax throughout.



How do you subtract two integers? Well, to compute the difference $a - b$, you instead compute the sum $a + (-b)$. How do you find $-b$? You flip all of the bits and then add 1. I'll try to illustrate. A 4-bit *unsigned* integer a can take on values $0 \le a \le 15$. A 4-bit *signed* integer b can take on values $-8 \le a \le +7$. The table below shows the unsigned and signed interpretations of the 16 possible 4-bit values.

bits	unsigned	signed
3210	interpretation	interpretation
0000	0	0
0001	1	+1
0010	2	+2
0011	3	+3
0100	4	+4
0101	5	+5
0110	6	+6
0111	7	+7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

To compute $5 - 3 = 5 + (-3)$, we write 5 as binary 0101 and write -3 as binary 1101 and add. (The carry bits are in grey on top.)

$$\begin{array}{ccccc} (1) & 1 & & 1 & \\ & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline (1) & 0 & 0 & 1 & 0 \end{array}$$

Interpreted as a 4-bit signed number, this is 0010 in binary or 2 in decimal, as expected. The meaning of the leftmost bit, in parentheses, is harder to explain.[2]

[2]If you're adding two n-bit signed integers, the result will properly fit into n bits if the top two

This representation of n-bit signed integers is called **two's complement** representation. It is how all modern computers (e.g. since the 1970s) represent negative integers. Nowadays most computers use 32-bit integers: with unsigned interpretation (e.g. `unsigned int` in the C programming language), they can represent values from $0 \ldots 4{,}294{,}967{,}295$, and with signed interpretation (e.g. `int` in C), they can represent values from $-2{,}147{,}483{,}648 \ldots +2{,}147{,}483{,}647$. On an Arduino Due board, an `int` uses 32 bits, as on a typical computer. On an Arduino Uno board, an `int` uses only 16 bits, and thus can represent values from $-32768 \ldots +32767$.

By the way, one common point of confusion, if you're new to programming, is the use of the `&` operator (bitwise AND) in C or Java or on an Arduino. If you do a bit-by-bit AND of two integers a and b (e.g. by writing

```
int a = 1234; int b = 4321; int d = a & b;
```

on your Arduino), here's how it works. The decimal (base-ten) value 1234_{10} has binary (base-two) representation 10011010010_2, and decimal 4321_{10} is binary 1000011100001_2. (The subscript 2 or 10 after a numeral indicates base-two vs. base-ten.) To compute `d = a & b`, we AND the bits one-by-one, i.e. $d_i = a_i \wedge b_i$:

$$\begin{array}{cccccccccccccc}
 & & & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
\& & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

The result is $11000000_2 = 192_{10}$. So it turns out that `(1234 & 4321)` is `192`. This particular example is completely frivolous, but the bit-by-bit AND operation is often useful for picking out one bit of an integer: e.g. the Arduino expression `(i & 8)` picks out bit 3 from the integer `i`, since $8 = 2^3$. So `(5 & 8)` is `0`, but `(12 & 8)` is `8`.

It's sort of a digression, but while we're talking about binary arithmetic, let me also mention multiplication. Suppose we want to multiply two 4-bit unsigned integers, e.g. to calculate $13 \times 5 = 65$.

$$\begin{array}{ccccccc}
 & & & 1 & 1 & 0 & 1 \\
\times & & & 0 & 1 & 0 & 1 \\
\hline
 & & & 1 & 1 & 0 & 1 \\
 & & 0 & 0 & 0 & 0 & \\
 & 1 & 1 & 0 & 1 & & \\
0 & 0 & 0 & 0 & & & \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

This works just like long multiplication in decimal, except that the multiplication table for single bits requires no memorization. Since the 1's bit of 0101 is 1, the first row of the result is 1101. Since the 2's bit of 0101 is 0, the second row of the result is

carry bits are equal: $c_n = c_{n-1}$. For signed addition, if $c_n \neq c_{n-1}$, then you have an *overflow* condition. For instance, if you try to compute $(-2) + (-8)$ using 4-bit signed integers, the correct result is -10 (decimal), which doesn't fit into 4 bits — that's an overflow. Incidentally, n-bit unsigned addition overflows if $c_n \neq 0$, for example if you add the unsigned 4-bit numbers 13 and 13 (decimal), the result is 26 (decimal), which doesn't fit into 4 bits.

0000 (shifted left one place). Since the 4's bit of 0101 is 1, the third row of the result is 1101 (shifted left two places). Since the 8's bit of 0101 is 0, the fourth row of the result is 0000 (shifted left three places). In case you're confused by the final addition step, I'll rewrite it to show (in grey) where you carry the 1's:

$$\begin{array}{cccccccc}
 & 1 & 1 & 1 & 1 & & & \\
 & & & & 1 & 1 & 0 & 1 \\
 & & & 0 & 0 & 0 & 0 & \\
 & & 1 & 1 & 0 & 1 & & \\
+ & 0 & 0 & 0 & 0 & & & \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

Notice that $1000001_2 = 65_{10}$, i.e. $13 \times 5 = 65$. If you multiply two n-bit unsigned integers, in general the result is a $2n$-bit unsigned integer. While it might be fun for me to show you in detail how to use logic gates to multiply a general 4-bit integer $a_3a_2a_1a_0$ by another general 4-bit integer $b_3b_2b_1b_0$, I think you can already grasp the idea well enough that the details would be a waste of time. Suffice it to say that the general case can be reduced to a bunch of ANDs, ORs, XORs, etc., as in the case of addition.

So I hope that I have now convinced you that (a) you understand how transistors are put together to make simple logic gates, and (b) you understand in principle how simple logic gates can be put together (perhaps in large quantities) to perform arithmetic operations.

One last small digression: You might wonder what happens when you ask a computer to use a real number, like π, or to evaluate the sine function. Amazingly enough, when you use a `float` in C, the computer internally represents the number in a binary version of scientific notation, called *floating point* representation. On most computers, a `float` is 32 bits: one bit for the sign (s), eight bits for the (signed) exponent (e), and 23 bits for the fraction (f). The sign s represents ± 1. The exponent e has possible values $-126 \le e \le +127$ (because the values -127 and -128 are reserved to flag errors, infinities, etc.). The fraction f has possible values $\frac{1}{8388608} \le f \le \frac{8388607}{8388608}$, where $8388608 = 2^{23}$. Combining the pieces, you get $s \times (1 + f) \times 2^e$. So computers internally use a form of scientific notation for real-valued (`float`) quantities.[3]

Now that you know how to make a bunch of transistors do arithmetic, you might wonder what it takes to make a bunch of transistors count from zero to ten. If you start counting 0, 1, 2, and around the time you reach 3, a Boeing 747 flies directly overhead just 100 meters above the ground, you will probably be too distracted to remember that the next value should be 4 — unless perhaps you were using your

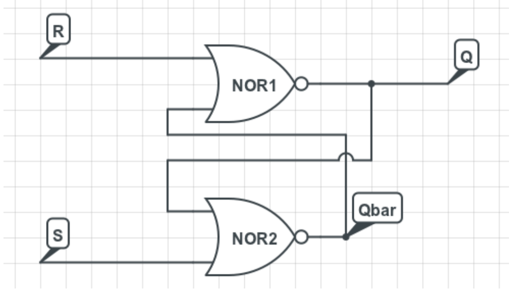
[3]There is one special-case value of e to represent numbers very close to zero (and zero itself), for which the $1 + f$ becomes $0 + f$. Also it turns out that instead of storing e as an 8-bit signed integer, the value $e + 127$ is stored as an unsigned integer. In the unlikely event that these details matter to you, see en.wikipedia.org/wiki/Single_precision_floating-point_format .

fingers or a pad of paper to keep track of your current position in the sequence. An operation like counting requires an internal state: not only do you need to know how to go from i to $i+1$ (using addition); you also need to keep track of what i is.

So far, the (various combinations of) digital logic gates that we have considered are capable of mapping the present values of n input bits into the present values of m output bits. This is called **combinational logic.** The output depends only on the present input.

To do something like counting, we need a digital device whose output depends not only on the present input, but also on past inputs. It maintains an internal state. As the inputs change, the device makes transitions through a sequence of internal states. This is called **sequential logic.** Sequential logic can remember things: it is capable of storing information to be retrieved at a later time.

The most fundamental sequential logic device is called a **flip-flop.** The simplest (though not the most useful) flip-flop can be made from two NOR gates:



S	R	Q_{next}	action
0	0	Q	hold present state
1	0	1	reset
0	1	0	set
1	1	?	(not allowed)

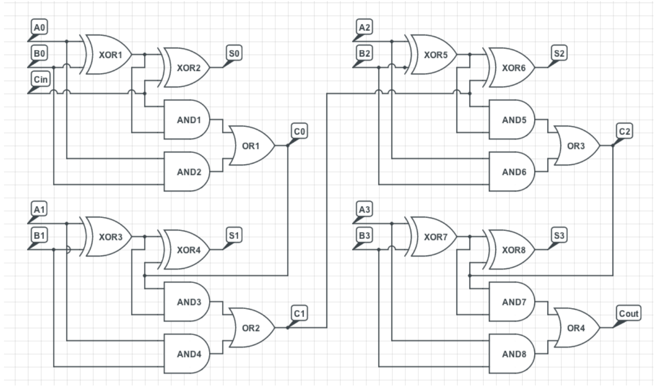
This device has two inputs, called **S**et and **R**eset, and an output called Q. (I don't know why it's called Q.) Also the output of the lower NOR gate is called $\overline{Q}$, because it contains the opposite of Q. Suppose that initially $R=1$ and $S=0$. Since at least one of the upper NOR gate's inputs is 1, its output must be 0, so $Q=0$. Thus both of the lower NOR gate's inputs are 0, so its output must be 1, so $\overline{Q}=1$. Now if we deassert R, so that $R=0$ and $S=0$, then nothing changes, because the top NOR gate has a single 1 input (so we still have $Q=0$), and the bottom NOR gate has two 0 inputs (so we still have $\overline{Q}=1$). If we then assert S, so that $R=0$ and $S=1$, then the bottom NOR gate has at least one 1 input, so $\overline{Q}=0$; then the top NOR gate has two 0 inputs, so $Q=1$. If we then deassert S, so that once again $R=0$ and $S=0$, then we stay in the $Q=1$ state, because the bottom NOR gate has at least one 1 input, so $\overline{Q}=0$, and then the top NOR gate has two 0 inputs, so $Q=1$. By asserting R (but not S), the output is *reset* to $Q=0$. By asserting S (but not R), the output is *set* to $Q=1$. In the normal state, neither S nor R is asserted, i.e. both of them are held at 0: in this case, Q keeps whatever value it had before. (The input combination $R=S=1$ is forbidden, because it leaves the outputs in the inconsistent state $Q=\overline{Q}=0$.) You can try this circuit out with mouse clicks at www.play-hookey.com/digital/sequential/rs_nor_latch.html .

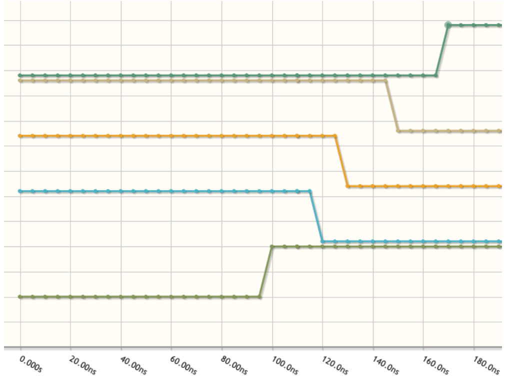
The above circuit is technically known as an "SR latch" (though you can call it a rudimentary form of an SR flip-flop). It is effectively a one-bit memory — and is in fact one possible way to implement the "memory" inside your computer. This circuit can help you to remember which of two states you are in, but it's not yet obvious how it helps you to count to ten. For instance, when you count to ten, there tends to be a fixed interval between 1 and 2, between 2 and 3, etc. If you want a large group of people to count together, to march together, or to play musical instruments together, it's helpful to have someone beating a drum or waving a baton. This is the role played by the **clock** in most sequential logic circuits. In general, sequential logic depends upon past outputs, but may or may not have a clock. But a very large fraction of the sequential logic that implements useful devices (e.g. computers) is also **synchronous** logic, meaning that it transitions from state to state in response to the beats of a drum or the $0 \to 1$ transitions of a clock signal. A clock signal is a square wave that toggles back and forth between the logic LOW and HIGH voltages, e.g. between 0 V and +5 V in the digital circuits we have considered so far. A synchronous circuit only updates its outputs immediately after the LOW→HIGH transition of the clock. This is very useful for coordinating the activites of many different components, some of which may be doing more complicated (hence slower) calculations than others. Think of the members of a marching band arranging themselves to spell out words on a football field. The clock plays the role of a camera or a strobe light periodically capturing an image of the field. If the band members move to their next positions shortly after each flash, they will be standing still in their correct locations in time for the next flash: the strobe or camera will capture the desired sequence of clear words, and will ignore the jumbled intermediate states present while members are finding their new places between words.

A feature of real-world logic gates (e.g. those made with transistors) that I neglected to mention before is that their outputs don't change instantaneously in response to their inputs. The transistors themselves, as well as the connections between them, have some finite capacitance. And only a finite current flows through an active transistor — even with V_{GS} well above $V_{\text{threshold}}$, there is still a finite resistance between drain and source. So a finite time is required for an AND gate's output to change from 0 to 1 when its inputs change e.g. from 0,0 to 1,1. This time is referred to as the logic gate's **propagation delay,**[4] sometimes also called **gate delay.** A typical propagation delay for a logic gate is on the order of a nanosecond: the very tiny CMOS logic gates used inside a modern computer are an order of magnitude faster than this, while the big 14-pin logic gates (e.g. found in our lab) that you used in Lab 19 were an order of magnitude slower than this. The finite propagation delay of each AND, OR, XOR, etc., is the reason why a complicated calculation, like adding two four-bit numbers, takes more time than a very simple calculation, like the AND of two single bits.

[4]By the way, one reason why computer chip makers keep making their transistors physically smaller each year is to reduce these capacitances.

The figure below illustrates the propagation delays of the various outputs of a four-bit adder.[5] Initially, the adder's two inputs are $A = 0111_2$, and $B = 0000_2$, so $S = A + B = 0111_2$; at time $t = 100$ ns, the second input changes to $B = 0001_2$, i.e. input B_0 changes from 0 to 1, so that $S = A + B = 1000_2$. The graph shows (vs. time) B_0, S_0, S_1, S_2, and S_3, from bottom to top.



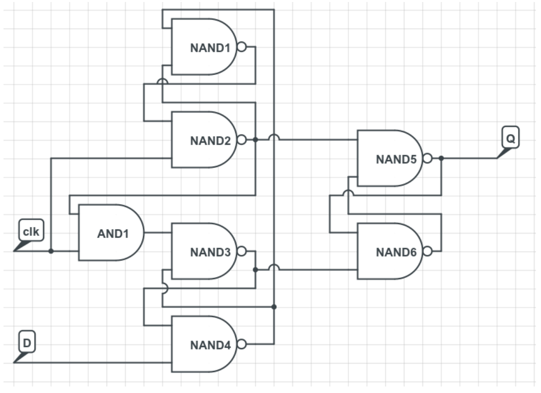


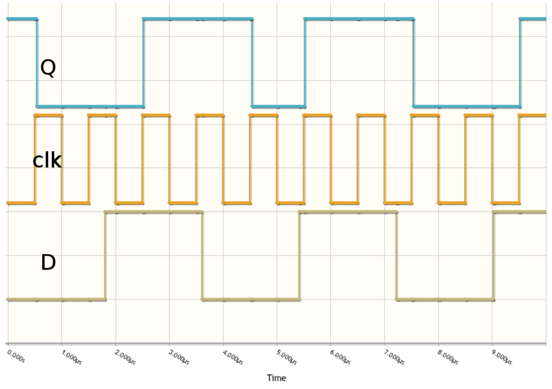
In CircuitLab, each gate has a 10 ns propagation delay. Since there are two gates between B_0 and S_0, the change in S_0 occurs at $t = 120$ ns (i.e. 20 ns after the change in B_0). Three gates between B_0 and S_1 cause S_1 to switch after 30 ns. Similarly, S_2 and S_3 change after 50 ns and 70 ns, respectively; S_3 changes last because it depends on all of the lower-order carry bits. A key point is that the output bits don't all change at the same time. Also, the times at which the various output bits will change have some uncertainties, as different types of gates will in reality have somewhat different propagation delays, and the delays will also vary with temperature, with the number of downstream gates connected to a given gate's output, and with details of how the gate is manufactured. You can put a lower and upper bound on each propagation delay, but in practice you can't predict exactly what each delay will be.

The fact that different output bits will change at slightly different times brings us back to the picture of a marching band rearranging itself on a football field between successive photographs (or strobe flashes). If I wait until, say, $t = 200$ ns to look at the output of the adder, then I don't care that S_0 updates before S_3. Since the slowest bit has settled to its new value 70 ns after the input changes, I can imagine presenting a new set of inputs at $t = 100$ ns, $t = 200$ ns, $t = 300$ ns, etc., and recording the output values at $t = 199$ ns, $t = 299$ ns, $t = 399$ ns, etc. Effectively, I use a drum beat (or a metronome) to count off ticks of 100 ns, and at each clock tick, I record the old outputs and then present the new inputs. This is how a **synchronous** digital logic system behaves. It is how your computer behaves — which is why the microprocessor's clock frequency (e.g. 1 GHz) is one way to characterize how quickly a computer helps you to get your work done.

[5]If you look carefully, you'll see that this 4-bit adder has an additional input called C_{in}, and you'll see that I now call C_{out} what I had called S_4 above. The convention is for a 4-bit adder to have both "carry in" and "carry out" pins, so that two 4-bit adders can be connected (carry-out of the first goes to carry-in of the second) together to make an 8-bit adder, etc.

The device that lets you “write down the previous outputs, and present the next inputs” once per tick of the clock is called an **edge-triggered D-type flip-flop**, which is often abbreviated as **D flop** in speech or as **DFF** in writing. You can implement a D flip-flop as shown in the figure below, though in practice a flip-flop is not something that you build for yourself — just as you don’t normally build your own NAND gate from transistors.

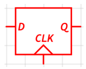




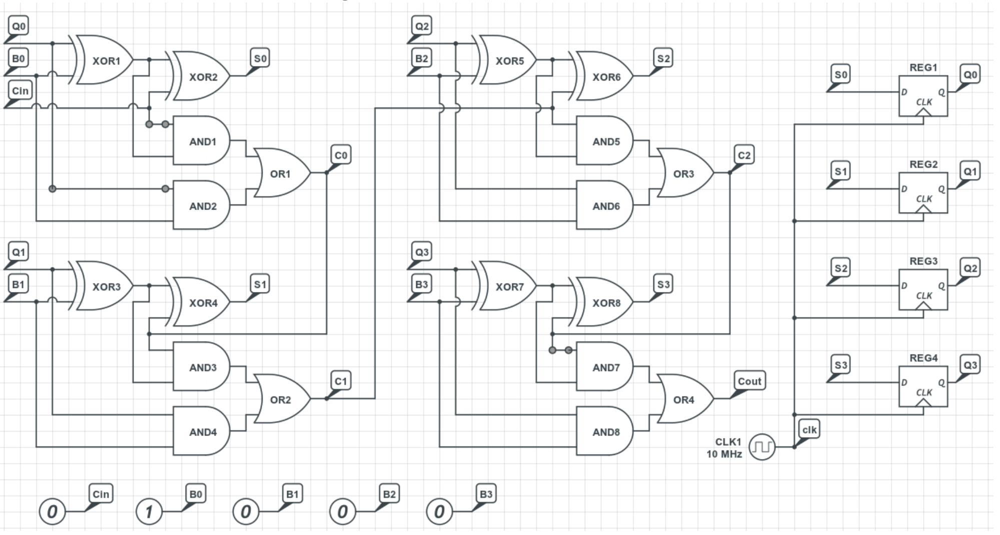
The three signals graphed (from bottom to top) in the above graph are **D**, **clk**, and **Q**. The **clk** signal is a 1 MHz square wave. The **D** signal is deliberately changing at weird times, so that it is more obvious that **Q** only updates immediately after a $0 \rightarrow 1$ clock transition. At each $0 \rightarrow 1$ edge of **clk**, the value of the **D** input from immediately before the clock edge is propagated to the **Q** output. So **Q** always reflects whatever state **D** had just before the most recent $0 \rightarrow 1$ clock transition. The effect is that even if **D** updates at an imprecise time, **Q** will update cleanly at the clock edge. There is no need for you to understand in detail how a DFF works; I just want you to know what it does. But if you’re curious, you can play with an online web-based simulation of a DFF at `www.cs.washington.edu/education/courses/cse466/11au/resources/sims/e-edgedff.html` . You can also look at my CircuitLab model of a DFF at `www.circuitlab.com/circuit/ywmw4t/d-fli` if you like. In essence, the DFF cleverly combines three SR latch circuits.[6] [7]

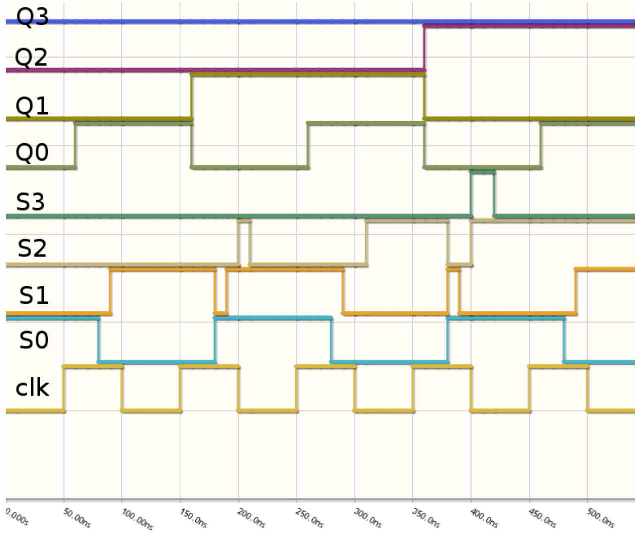
[6]You can make an SR latch either with two NOR gates or with two NAND gates; the version you see here uses NAND gates. The NOR version of an SR latch has two inputs called S and R: you put $S = 1$ to set $Q \rightarrow 1$, or you put $R = 1$ to reset $Q \rightarrow 0$, or you put $S, R = 0, 0$ to leave Q unchanged. The NAND version of an SR latch has two inputs called $\overline{S}$ and $\overline{R}$: you put $\overline{S} = 0$ to set $Q \rightarrow 1$, or you put $\overline{R} = 0$ to reset $Q \rightarrow 0$, or you put $\overline{S}, \overline{R} = 1, 1$ to leave Q unchanged. These inputs are called *active low* because their active (asserted) state is 0, and their resting (deasserted) state is 1. In the DFF schematic, the upper SR latch controls the $\overline{S}$ input of the right-hand SR latch, and the lower SR latch controls the $\overline{R}$ input of the right-hand SR latch. The upper and lower SR latches are cleverly configured so that if $D = 0$, the right-hand latch is reset shortly after the $0 \rightarrow 1$ clock transition, and if $D = 1$, the right-hand latch is set shortly after the $0 \rightarrow 1$ clock transition. The fact that this *can* be done is much more interesting than the details of *how* it is done.

[7]Note to self: next time, consider using the HADES (Hamburg Design System) interactive simulation framework to make a bunch of interactive web applets to illustrate these logic building blocks. See for example `tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/16-flipflops/20-dlatch/dff-enable.html` .

Now, if we put a DFF after each of the outputs $S_3 \ldots S_0$ of the 4-bit adder that we were discussing earlier, and then we send the Q output of each of these 4 DFFs into the inputs $A_3 \ldots A_0$, and we fix the inputs $B_3 \ldots B_0$ to the values 0001, we will have a circuit that can count up by one each time the clock ticks! The figure below shows a 4-bit counter, made by combining a 4-bit adder with 4 DFFs. Notice that the schematic symbol for a DFF looks like this: 

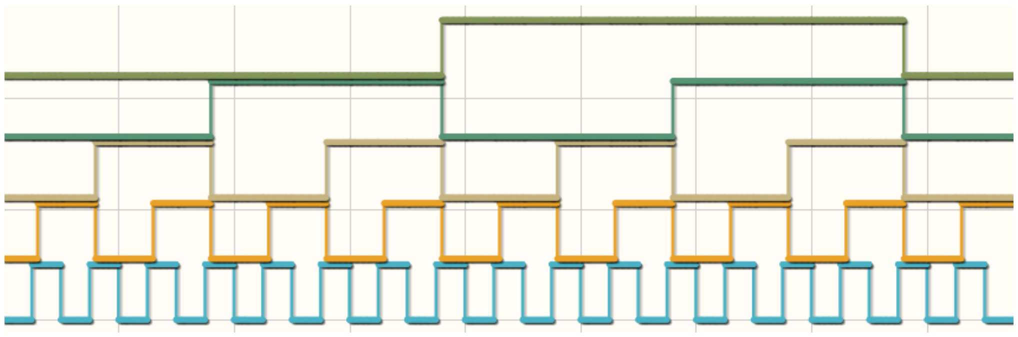
. The **D** input is on the left, and the **Q** output is on the right. The **clk** input is always drawn as a small wedge pointing into the flip-flop.





The simulation traces above are, from bottom to top: clk, S_0, S_1, S_2, S_3, Q_0, Q_1, Q_2, Q_3. Remember that the S outputs of the adder are connected to the D inputs of the D-flops. Notice that the various S_i do not all change simultaneously, because of the different numbers of gate delays; sometimes the S_i are even momentarily indecisive, because it takes a finite time for each carry bit to propagate. But the Q_i all change cleanly together just after each $0 \to 1$ clock edge. (By the way, the $0 \to 1$ clock transition is called the **positive** clock edge.) Notice that if we arrange for all of the Q_i to be 0 at time $t = 0$, then the bits $Q_3Q_2Q_1Q_0$ count out the sequence 0000_2, 0001_2, 0010_2, 0011_2, 0100_2, 0101_2, …, which in decimal is 0, 1, 2, 3, 4, 5, …, incrementing once per period of the clock. If you want to tinker with this circuit, it is online at www.circuitlab.com/circuit/qyu323/4-bit-counter/ .

By the way, if we let this counter keep running, the sequence repeats itself after 16 clock cycles: 0000, 0001, 0010, 0011, …, 1110, 1111, 0000, 0001, …, as shown in the figure below, which graphs (from bottom to top): clk, Q_0, Q_1, Q_2, Q_3.



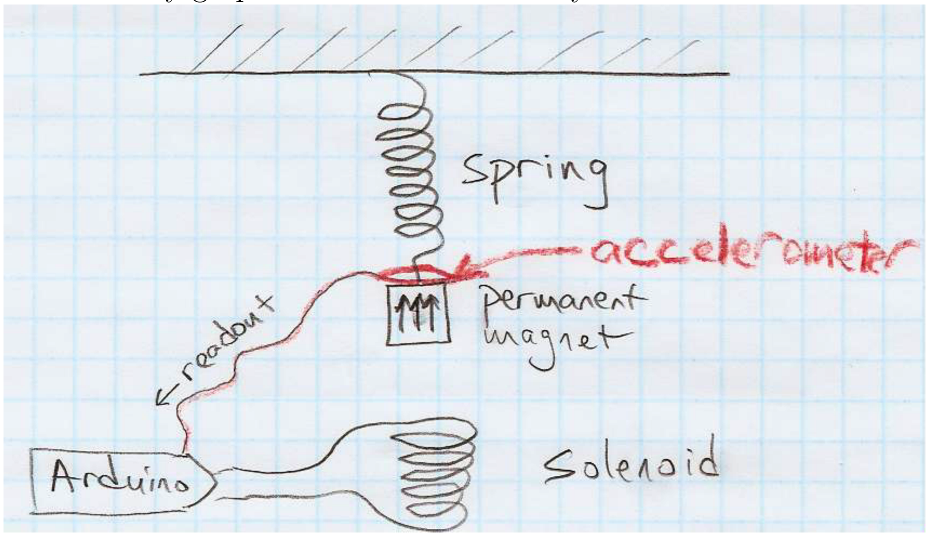
It turns out that writing numbers out in binary gets tedious, because you have to write so many digits. Imagine an 8-bit counter, which counts from 00000000_2 up to 11111111_2 and then wraps around. It is often convenient to write long binary numbers out in **hexadecimal** (i.e. base 16), which reduces by a factor of 4 the number of digits you need to write down. In hexadecimal, each digit can take on values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. So an 8-bit counter goes from 00_{16} to FF_{16} and then wraps back to 00_{16}; a 16-bit counter goes from 0000_{16} to FFFF_{16} and then wraps back to 0000_{16}. The reason hexadecimal can be more convenient than decimal is that each group of 4 bits maps to exactly one hexadecimal digit: you can easily convert 1234_{16} into binary in your head, while converting 1234_{10} into binary would require you to write $1234 = 1024 + 128 + 64 + 16 + 2$, which is a big hassle. In the C programming language (and in Arduino programming), you put `0x` in front of a number to indicate hexadecimal. So you would write `0x1234` to mean 1234_{16}. That might explain some of the confusing syntax you encounter during the Arduino labs. (We'll see a different kind of confusing syntax during the FPGA labs!)

If you didn't already do so last weekend, please **skim** *through pages 213–233 (sections 8.8 to 8.16) of Eggleston's textbook, i.e. the rest of the digital chapter that you started to read a few weeks ago.* You will probably enjoy seeing analog/digital conversion described again, now that you have worked with it. And there are a few new concepts, such as multiplexers, demultiplexers, and memories, that I have not yet described in these notes, though they will appear in next week's notes.

Before we return in Lab 23 to digital logic gates, flip-flops, etc., Lab 22 will be one final Arduino lab, in which you will program your Arduino to toggle on and off, periodically, the flow of electric current through a small solenoid.[8] This solenoid, when energized, will attract a permanent magnet that is suspended from a spring, as shown below. An accelerometer, attached to the permanent magnet, is read out by the Arduino, to monitor the up-and-down motion of the permanent magnet in response to the periodic driving force exerted by the solenoid. Zoey dreamed up this lab exercise. I like it because it connects with physics you have learned in earlier courses, and it lets your Arduino interact with the physical world. Also, since accelerometers are nowadays embedded in smartphones (mainly to sense orientation) and in computer

[8]Zoey disassembled a bunch of doorbell mechanisms to obtain these solenoids!

hard drives (to respond defensively to free-fall conditions!), you might want to see how a computer communicates with such a device. Finally, it is a chance to have your PC interactively graph some data read out by the Arduino.

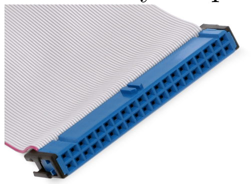


To get an idea of what you'll be doing in Lab 22, please **skim** through the current draft of Zoey's write-up: `positron.hep.upenn.edu/wja/p364/2014/files/lab22.pdf` .

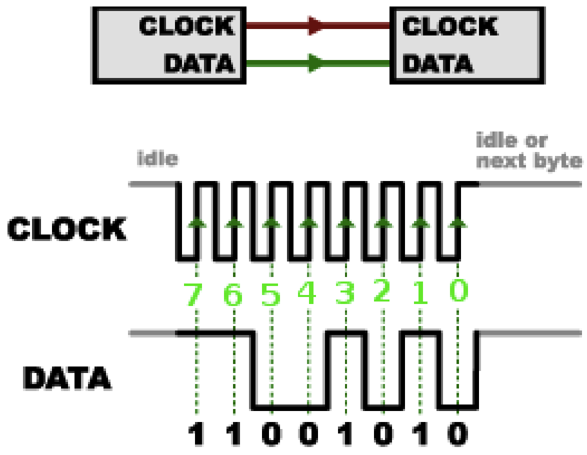
Notice first that the accelerometer uses a **serial interface** to communicate with the Arduino. What does serial mean? Imagine that you want to transmit an 8-bit binary number, e.g. $10001001_2 = 137_{10}$ from one side of the room to the other. You have a supply of placards on which the front side reads **0** and the back side reads **1**. To transmit this number in parallel, you would have eight people stand up side-by-side and simultaneously show their placards, which would read 1,0,0,0,1,0,0,1. This is analogous to running eight separate wires across the room, one for each bit of the 8-bit binary number to be transmitted. To send the number serially, instead, you would have just one person stand up, whose placard would first display **1**, then **0**, then **0**, then **0**, then **1**, then **0**, then **0**, then finally **1**, perhaps updating the placard direction once per tick of an external clock. Timing is an important part of serial communication, so that you can correctly distinguish 101 from 1001 from 10001, etc. So serial communication almost always makes use of a clock signal that can be observed by both sender and receiver — i.e. it is *synchronous*.[9] The main advantage of serial communication is that fewer wires are needed than for parallel communication. Morse-code telegraphy (in which only one "wire" is available) is an early example of serial communication. The **S** in USB (for computer peripherals) and in SATA (for disk drives) stands for *serial*. Contrast the photos below of an old "parallel ATA" disk-drive cable, a modern "serial ATA" disk-drive cable, and a USB

[9] *Asynchronous* serial communication, such as that used by a PC's `COM` port, does not use a shared clock. Instead, it requires that both sender and receiver agree in advance on the frequency at which bits will be transmitted (e.g. 9600 bits per second). In addition, they must agree to follow a convention in which the sender transmits **0** when idle between messages, each message has a fixed length (e.g. 8 bits), and each message is preceded by an additional **1** bit that serves as a *start* signal.

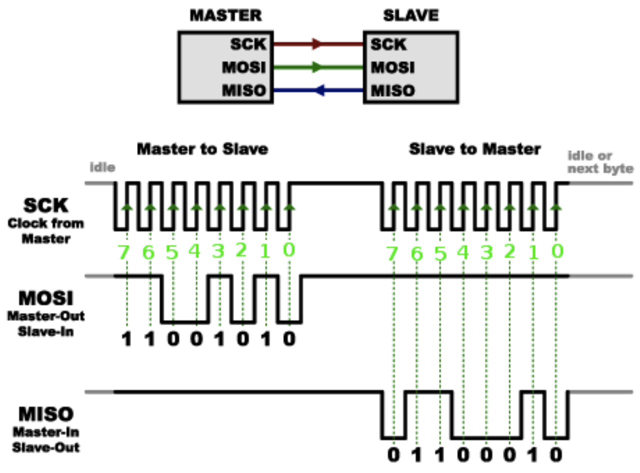
cable. The serial connections use fewer wires and are more compact. In the Arduino environment, where the total number of available input/output pins is limited, a serial interface leaves more of your precious pins free for other uses.



The serial protocol that we will use to read data from the accelerometer in Lab 22 is called SPI. The SPI interface uses separate clock and data lines. The receiver samples the state of the incoming data bit at each **rising edge** of the clock signal. Information is sent as a sequence of bytes (8 bits per byte). The bits in each byte are read out from left to right ("most-significant bit (MSb) first"), i.e. the 8-bit number b is sent in the order $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$. In the figure on the right, the byte $202_{10} = 11001010_2 =$ `0xca` (hexadecimal CA) is transmitted serially, bit by bit, in the sequence 1,1,0,0,1,0,1,0. Notice that the vertical dashed green lines in the figure indicate where each bit of the byte is sampled.

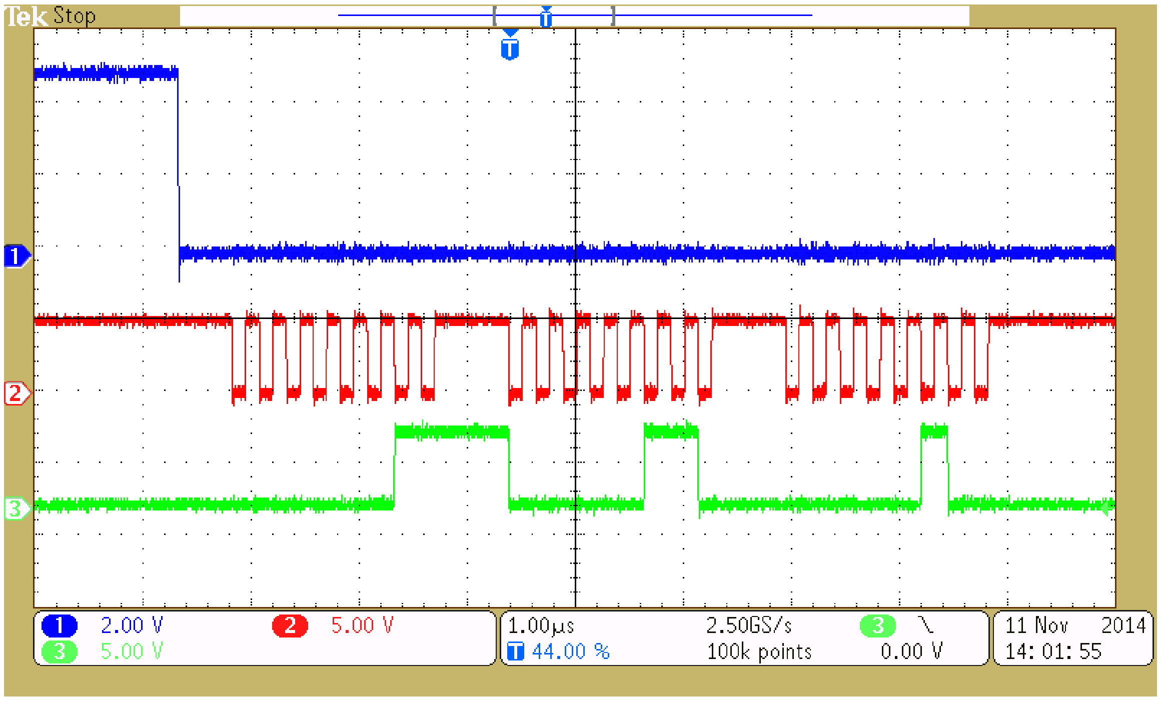


To allow for bidirectional data flow, SPI uses two data lines — **MOSI** (master out, slave in), and **MISO** (master in, slave out). One of the two sides (called the *master*) provides the clock signal (actually called **SCK** (serial clock)) and initiates the communication. The *slave* responds to commands sent by the master. (I usually prefer to say "target" rather than "slave," but the SPI nomenclature is "slave." Some authors have switched to using "leader" and "follower.") In our case, the Arduino will be the master, providing the serial clock and initiating commands to which the accelerometer



responds. So **MOSI** carries data from Arduino to accelerometer, while **MISO** carries data back from accelerometer to Arduino. In the above-right figure, the master sends out the byte $11001010_2 = 202$, and the target responds with the byte $01100010_2 = 98$.

An SPI interface normally consists of four distinct wires, not three. The fourth wire, called **SS** (target select), is also controlled by the master (i.e. the Arduino). It tells the target to wake up and get ready to receive and process a new command from the master. The **SS** line rests in the HIGH state between commands. Each new command to the target begins by setting the **SS** signal LOW, where it stays until the command is completed. An example command would be, "Send me the present value of the z component of acceleration."

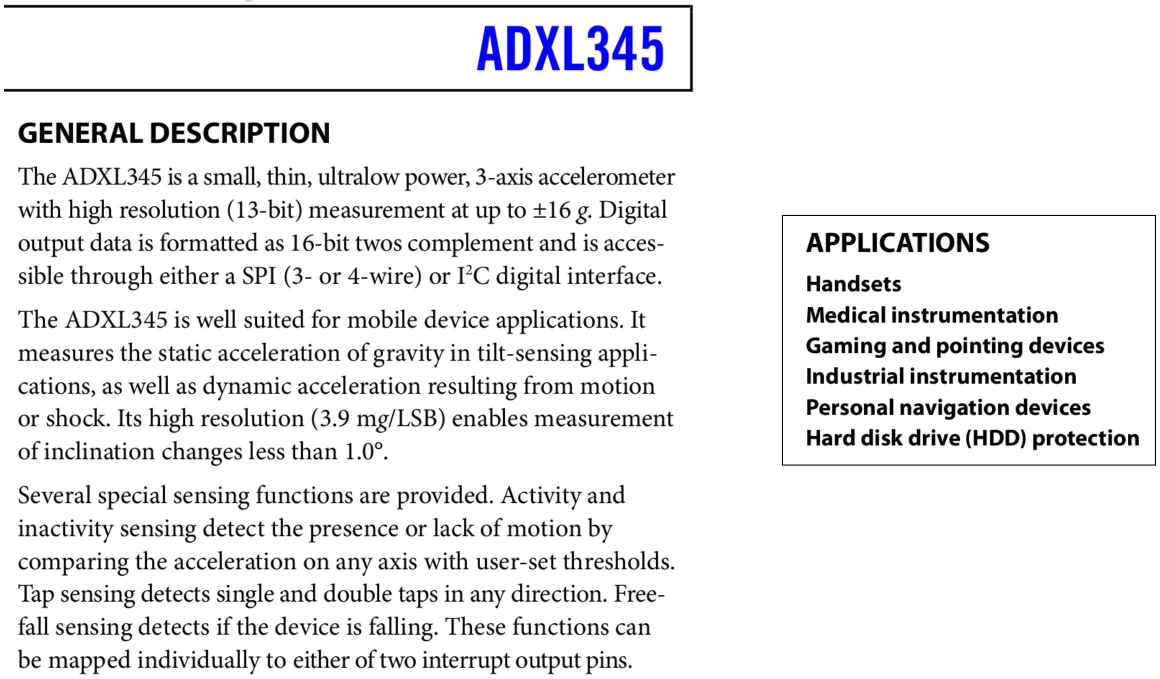


The figure above is an oscilloscope screenshot of the Arduino sending a sequence of three bytes to the accelerometer. The blue trace is **SS**, the red trace is **SCK**, and the green trace is **MOSI**. There is no response here, so **MISO** is not shown. See if you can figure out the three bytes that are sent, by looking at the value of the green data line at the times when the red clock line goes from LOW to HIGH. Here's a hint: *Laboratory Electronics.*

For more information on SPI, you can look here (from which I took the SPI figures): `learn.sparkfun.com/tutorials/serial-peripheral-interface-spi` . Beware that their SPI description reverses the order in which the bits of each byte are transmitted.[10]

[10]Some SPI devices use LSb-first ordering, but as far as I can tell, it is much more common to use MSb-first, as our accelerometer does. The Arduino is capable of handling either convention.

The accelerometer in Lab 22 will be the ADXL345, whose data sheet is at www.analog.com/static/imported-files/data_sheets/ADXL345.pdf . Below is a snippet of the manufacturer's description of what this accelerometer does. The resolution "3.9 mg/LSB" means that the digitized acceleration changes by one count (one "least-significant bit") when the acceleration changes by 0.0039×9.8 m/s^2.



ADXL345

GENERAL DESCRIPTION

The ADXL345 is a small, thin, ultralow power, 3-axis accelerometer with high resolution (13-bit) measurement at up to ±16 *g*. Digital output data is formatted as 16-bit twos complement and is accessible through either a SPI (3- or 4-wire) or I^{2}C digital interface.

The ADXL345 is well suited for mobile device applications. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock. Its high resolution (3.9 m*g*/LSB) enables measurement of inclination changes less than 1.0°.

Several special sensing functions are provided. Activity and inactivity sensing detect the presence or lack of motion by comparing the acceleration on any axis with user-set thresholds. Tap sensing detects single and double taps in any direction. Free-fall sensing detects if the device is falling. These functions can be mapped individually to either of two interrupt output pins.

APPLICATIONS

Handsets
Medical instrumentation
Gaming and pointing devices
Industrial instrumentation
Personal navigation devices
Hard disk drive (HDD) protection

In order to graph (and perhaps analyze) accelerometer data collected by the Arduino, we will need to write some short programs that run on the laptop PC. The Arduino will send out data with the usual `Serial.println()` function, but instead of simply displaying these messages as printed text, the PC program will interpret these messages as data to be graphed on the PC's screen. Zoey will provide a program that does 90% of this job for you; you will just need to fill in the missing 10% to get it working (with as much help from us as you like).

The programming language we will use on the laptop for graphing, etc., is called Processing. It is a very simplified version of Java, with a programming environment that looks just like that of the Arduino. In fact, the Arduino platform is derived from the Processing platform. Both were created with the aim of putting technology into the hands of artists and visual designers. The Processing-based parts of Lab 22 will go more quickly for you if you have had a chance to try out Processing at home first. Especially if you prefer to work on your own laptop computer rather than one of the lab's laptops, you should download the Processing programming environment and write the basic "Hello mouse" program on your machine, by following this link: processing.org/tutorials/overview/

If you do manage to install Processing on your machine, try out this short sketch that

I wrote for the students in Physics 008 (*Physics for Architects*) last year. I'll bet you can guess without even running it what it is going to do! What is amazing about Processing is that you can make pretty decent animations with minimal coding effort.

```
void setup() {
  size(900, 450);
}

void draw() {
  float t = 0.01*frameCount;
  float xsun = 0.5*width;
  float ysun = 0.5*height;
  // clear screen before each new frame
  background(128);
  // draw sun
  ellipse(xsun, ysun, 20, 20);
  float rplanet = 200;
  float xplanet = xsun + rplanet*cos(t);
  float yplanet = ysun + rplanet*sin(t);
  // draw planet
  ellipse(xplanet, yplanet, 10, 10);
  float rmoon = 30;
  float xmoon = xplanet + rmoon*cos(t*365/27.3);
  float ymoon = yplanet + rmoon*sin(t*365/27.3);
  // draw moon
  ellipse(xmoon, ymoon, 5, 5);
}
```

If you prefer not to bother installing Processing on your own machine, you can learn a bit of Processing in this one-hour-long video lesson, in which you can type your very first short Processing programs into your web browser, with no software installation at all: hello.processing.org

If you do manage to try out Processing this weekend, you can earn 10% extra credit on this week's reading by sending me (along with your answers to the usual reading questions) the source code for whatever sketch you decided to write in Processing. If possible, send it in a form that I can easily copy and paste into my own Processing sketch editor, in case I want to try it out.

Completely optional: if you're interested in a textbook-level introduction to the general topic of feedback and control systems, you can skim Chapter 1 of this book:
www.cds.caltech.edu/m̃urray/books/AM05/pdf/am08-complete_30Aug11.pdf
(Zoey found this online.) The book also includes a chapter on PID controllers (Chapter 10), in case Lab 16 left you eager to learn more. If you do decide to read Chapter 1, you can earn 10% extra credit on this week's reading by writing a couple of sentences to tell me about something that you found interesting from the chapter.