FREQUENTLY ASKED QUESTIONS

April 6, 2017

Content Questions

How do we convert analog signals into digital?

There are various ways to do this. You can devise a circuit that creates a 0 or 1 level (high or low voltage) output whenever the input is below or above some predetermined threshold—this kind of circuit is called a "discriminator". Some circuits are purely digital, though, and have components that (ideally) only produce high and low voltage-level signals.

Why does the muons example work for muons and not other particles?

In today's cosmic ray veto example, you get a coincidence trigger when a muon traverses both counters. It would indeed work for other particles traversing both counters, but most particles are not as penetrating as muons and would only hit one counter.

Cosmic ray muons are quite energetic and most can easily go through two scintillator paddles in a few-meter scale setup. Most particle detector experiments are looking at beta or gamma radiation, or neutrons, etc. which will typically deposit energy over less than a few tens of cm.

Of course what the signal and background look like depend very much on what specifically you are trying to do, and it can sometimes be difficult to tell the signal from the cosmic background. And sometimes it's the cosmic rays that are the signal and the background is from something else... But in pretty much all these cases you use some kind of timed logic to "trigger" your experiment, i.e., to count a signal or tell your electronics to record the data.

How do we adjust a FET's threshold voltage?

Do you mean the voltage at which the FET is "on"? The behavior of the FET as a function of V_{gs} is a property of the FET (and depends on whether it's a JFET, an e-MOSFET or a d-MOSFET). To decide how to choose V_{dd} and the "pull-down" resistor R_d in a FET switch, you find a solution like

that described in Eggleston 5.3, using the I_d vs V_{ds} characteristic of the FET, and select voltages for which the FET is in the plateau region with small or large I_d when V_{gs} is high or low.

How do AND/OR logic gates work on a physical level?

They are made out of switches– we already saw how to make an inverter (NOT gate) out of either a bipolar transistor or a FET. We'll also see next class how to make a NOR gate out of bipolar transistors. Other gates can be made in similar ways.

Why are base 2, 8, 16 particularly easy to compute with? What about base 4?

Bases that are powers of 2 are easy to compute with because you can deal with collections of binary bits: base 2 is 1 bit (numbers 0-1), base 4 is 2 bits (numbers 0-3), base 8 is 3 bits (numbers 0-7), and base 16 is 4 bits (numbers 0-15). Base 4 would actually be fine, but you only get 2 bits to play with, so it's not so common.

In the 2's complement, how do you tell a number is negative and not a different positive number?

In 2's complement, negative numbers always have a 1 as the most significant (leftmost) bit. If you are storing positive numbers in n bits, you can store from 0 to $2^n - 1$. In contrast, in 2's complement you can only store up to $2^{n-1} - 1$ positive numbers, instead going from -2^{n-1} to $2^{n-1} - 1$.

As a concrete example of this, take 8 bits. If you are storing just positive integers, your numbers go from $0_{10} = (00000000)_2$ to $255_{10} = (111111111)_2$, i.e., 0 to 255. If you are storing numbers with a 2's complement convention, with 8 bits your numbers go from $-(128)_{10} = (10000000)_{2,2'\text{s complement}}$ to $127_{10} = (01111111)_2$ (a total of 256 numbers represented). Note that +128 can't be represented in 8 bits in 2's complement, since the meaning of $(10000000)_2$, the nominal positive representation of 128_{10}, is actually $-(128)_{10}$ in 2's complement.

How do you figure out the result after adding 1 in the "complement" convention for a negative number?

To add 1 to a binary number, you first flip the least significant bit. If this changed a 0 to a 1, you're done. Otherwise, you "carry" the 1, and add it to the next significant bit; you then repeat this process from right to left until you reach the most significant bit.

For example, in binary, 1+1 = 10, 1+10 = 11, 1+11 = 100, etc. Today's examples were: 1100000+1 = 11000001, and 10111111+1 = 11000000.

(Try a few examples if you are uncomfortable with this.)

Can you explain again how 2's complement is useful?

If negative numbers are encoded as 2's complement, addition and subtraction works much more easily. You can add 2's-complement-encoded numbers with the usually binary addition method, whereas the signed-magnitude method of encoding negative numbers (for which the leftmost bit is assigned to carry the minus sign information) is not compatible with that.

Take the example from class:

The 2's complement version of $(63)_{10}$ is 00111111, and of $(-63)_{10}$ is 11000001. Now add these (by the method in the question above, i.e., in each place "carry" a resulting 1 to the next-most-significant bit, and continue right to left). This yields 00000000, which is what you expect from adding 63 and −63.

Now suppose you try the same with signed-magnitude-encoded numbers. 63+−63 in signed-magnitude binary is 00111111+10111111, which by binary addition yields 11111110, which is not zero. So adding these requires some tricky manipulation.

(We'll see how to implement the carry-the-overflow-bit method of binary addition in digital electronics a couple of lectures from now.)

How do the numbers with the letters work?

These are *hexadecimal* numbers— they are base 16. A single character represents numbers from 0 to 15. 0 through 9 are represented by the usual digits 0 through 9. Decimal 10 is designated A, decimal 11 is designated B, etc., and decimal 15 is designated F. So 0 through 15 in decimal correspond to 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F in hexadecimal.

You can store a hexadecimal number in 4 bits.

Where do the Boolean rules come from?

Well, I guess "math"! While most of the Boolean rules are consistent with our everyday logic intution (subject to the caveat that the meaning of "+" is not the everyday meaning of exclusive OR but Boolean-defined OR), you can derive all the rules from today using the basic truth tables of AND, OR and NOT.