State machine encoding and start on error correction.

Today we'll touch on two topics:

- Finish up the basic computer design
- The basics of error correction

For the first, we're going to be using the notes from last time (not reprinted here). For the second error detection and even correction are an important application of digital logic that shows up in a number of surprising places and gives us a nice application.

Basic Error Detection

Digital logic has applications in any number of places. One of those places is in communication technology. Most communication devices (your cell phones for example) tend to have communication errors. In the case of a cell phone, your voice is converted from an analog signal to a digital one. A primary reason for doing this is so that small errors that occur (due to interference) don't actually change what's being received. This is true for two reasons:

- A digital "bit" that gets a bit of noise in it (say is 4.8V rather than 5V) is easy to correct back to the correct binary value.
- Even if the noise is so large a "1" becomes a "0" or a "0" becomes a "1" we can add some extra bits to the message and either detect or correct the error.

Terms (a bit formally)

Let x and y be bit strings of the same length. Define h(x,y) to be the number of bits of x which need to be flipped in order to get x=y. For example, if x=1010 and y=1100 then h(x,y)=2. Notice that h(x,y)=h(y,x). The function h() is a measure of the **Hamming Distance** between two bit strings.

Now consider the function H(X) where X is a set of bit strings $\{x_1, x_2, \ldots x_n\}$ each of the same length. Let H(X) be the minimum Hamming distance between any two elements of the set X. (That is for any a and b in the range of 1 to n where $a \neq b$, $h(x_a,x_b) \geq H(X)$.) For example, if X={1111,1000, 0000},H(X)=1 because 1000 and 0000 have a Hamming Distance of 1, and no two elements of that set have a Hamming distance of 0.

In the context of error correction and detection, the notion of the Hamming distance of a set is useful because it tells us something about the redundancy of the information in the set. That is, it tells us how many bits could flip before we might confuse two bit strings in that set. For example, say we are communicating one of Y possible messages from one location to another. If the set of Y possible messages has a Hamming distance of 2, then a single bit-flip during transmission cannot confuse any

two messages. That is, we can *detect* a single bit-flip. Of course two bit-flips during transmission could still cause one message to look like the other.

Example problems:

1. Consider the set X={10000, 00110, 11111}. What is H(X)?
   Ans., h(10000,00110)=3, h(10000,11111)=4, h(00110,11111)=3. As the smallest of these is 3, H(X)=3.
2. Consider set X={1001,1111,1010). Say we were communicating between two locations, using only encodings available in X. What is the minimum number of bit flips during transmission that could cause a given message in that set to appear to be a different message in that set?
   Ans., H(X)=2. So two bit-flips are needed to cause one legal message to be mistaken for another.

Parity

Given a set X of all possible bit-strings of a certain length, that set has a Hamming distance of 1 as any given bit-flip will cause one member of that set to appear to be a different member of that set. So how do we generate a set that has a Z elements and a Hamming distance of X? In general for an arbitrary X, this is a hard problem if we want the length of the bit strings used to be minimal. However, for a Hamming distance of 2, this is fairly simple. If we have a set of Z bit strings with a Hamming distance of 1, we can simply add a single bit to the end of each bit-string. Add that last bit so that the number of 1's in the bit-string is even. Call this set of extended bit-strings Y. Now H(Y)=2. How do we know that? Well, a single bit-flip will either convert a 0 to a 1 or a 1 to a 0. In either case the number of 1's in the bit-string will no longer be even. As we know that *every* legal encoding has an even number of 1's, the bit-string with a single bit flip cannot be an element in the set Y. Thus, 1 bit flip cannot make any element of Y look like any other element of Y.

This last bit we are adding is called a *parity* bit and the exact scheme we used is called *even parity* because we made the number of 1s in the string *even.* Odd parity also works, but you can't mix and match them. Within a given set you have to consistently use either even or odd parity. Those of you who have ever played with UART may have noticed that you have to set either even or odd parity.

Example problems:

3. Add an even parity bit to the following bit-strings 11, 1110, 10101.
   Ans., 11<u>0</u>, 1110<u>1</u>, 10101<u>1</u>. Notice that in each case the number of 1's is even.

4. Using even parity come up with a set of four 3-bitencodings that has a Hamming distance of 2.
   Ans., {000, 011, 101, 110}.

Error correction

Now the above scheme gave as a method of *detecting* a one-bit flip. This is useful because we can, at the least, not use bad data, and we could perhaps even ask for the data to be resent. But what if we want to *fix* the data without needing to ask for a resend? It turns out we can do that if the set of messages we are using has a Hamming distance of 3 or more.

Why is that?

Consider a really simple example, where our set of legal messages was {000, 111}. What is the Hamming distance? If we know that no more than one bit gets flipped, do you agree we can recover the original message?

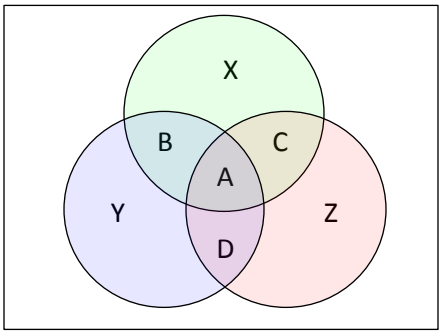
Comments/Questions:

- Obviously this "triple redundancy" has a lot of overhead. 200% in fact. That seems troubling.
- This relies on no more than 1 bit getting flipped. What if 2 bits get flipped?
    - Why isn't that something we are hugely worried about?

Let's walk a specific scheme with 4 data bits and 3 parity bits

A, B, C, D are data bits while X, Y, Z as parity bits
Let $P(x_1..x_n)$ be the function that generates even ones parity over the inputs x_1 to x_n.

- Let X=P(A,B,C)
- Let Y=P(A,B,D)
- Let Z=P(A,C,D)



Again, we are going to assume that at most one bit will get flipped and worry later about more than one bit getting flipped.
Which parity bits will be incorrect if the following bit were flipped in transmission?

A	B
C	D
X	Y
Z	

How does that help us?

Other thoughts

1. How can we use that set to detect 2-bit errors?
2. How do we build the encoder (takes in 4 data bits, generates 7 bit message) out of gates?
3. How do we build the decoder (takes in 7 bit message, generates *corrected* 4-bit data) out of gates?

2. How do we build the encoder (takes in 4 data bits, generates 7 bit message) out of gates?

3. How do we build the decoder (takes in 7 bit message, generates *corrected* 4-bit data) out of gates?