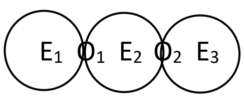
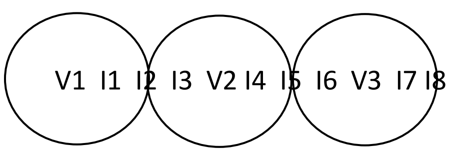
1. Parity
    a. Errors with a Hamming distance of 1 can be detected, but not located, with parity
    b. *Even parity* – count the number of 1s in the data
        i. Set or clear an additional bit so that the number of 1s is even (including the parity bit)
    c. *Odd parity* – set or clear an additional bit so that the number of 1s is odd
        i. We'll be using even parity for the rest of these examples
    d. One type of XOR gate produces a 1 whenever the number of 1s input is odd, perfect for this use
    e. Example for even parity
        i. C denotes the position of the check bit
        ii. C1001 -> 01001
        iii. C1101 -> 11101
    f. Even parity creates valid code words that have a Hamming distance of 2 between them
        i. Need valid and invalid code words so we know when something goes wrong
        ii. Invalid code words in this case are the odd parity numbers
            1. E = even (valid) codeword, O = odd (invalid) codeword
            2. Odd codewords map to multiple even codewords, so can't correct
                a. Assume only a single bit changed
                b. Does 01101 (from above) map to 11101, 00101, 01001, 01111, or 01100?
                c. Can't tell exactly which, any bit could have been changed in transit



Circles represent the space of all codewords associated with the valid one (here, even). They have a radius of one Hamming distance.

    g. Two bit errors
        i. To allow detection, need valid code words with at least a Hamming distance of 3 away
            1. Below is for some theoretical error correction scheme, **not** parity or Hamming(7,4)
            2. Hamming distance is 4 away here, 4 steps from V1 to V2



Circles represent the space of all codewords associated with the valid one (here, some arbitrary scheme we haven't defined). They have a radius of two Hamming distance.

        ii. All one Hamming distance errors are associated with exactly one valid code word
            1. Thus, can correct these ones back to the corresponding valid code word
        iii. Errors with two bits will still be detected, but may be associated with another valid code word
            1. I2 above could either be associated with V1 or V2
        iv. What happens with three-bit errors?
            1. With enough errors, *any* error correction system will fail
        v. Further reading on error correcting codes and the circles above [on Quantitative Decisions]

2. Further bit checking
    a. Increase Hamming distance between valid code words with more parity bits
    b. Will look at Hamming(7,4) error correction
        i. Hamming(7,4) is used in this class as SECSED – Single Error Correction Single Error Correction
            1. Not SECDED as I previously mentioned, see next note for why
        ii. Can also be used as solely Double Error Detection (DED)
            1. However, cannot reliably correct errors when used this way
            2. If we call it SECDED then it has to be able to correct double errors without caveat
            3. Extension to Hamming(7,4) with one more parity bit on whole codeword gives SECDED
                a. We won't talk about it or use it in this class, though

3. Hamming(7,4) examples
    a. Example: 4-bit word

111	110	101	100	011	010	001	Bit position (binary)
7	6	5	4	3	2	1	Bit position (decimal)
D3	D2	D1	C2	D0	C1	C0	Bit type (D = data, C = check / parity)

        i. C0 is the parity bit over bits 3, 5, 7
        ii. C1 is the parity bit over bits 3, 6, 7
        iii. C2 is the parity bit over bits 5, 6, 7
    b. Original data: 0110
        i. Let's calculate the code word associated with it
            1. Fill in table, then calculate check bits

111	110	101	100	011	010	001
7	6	5	4	3	2	1
0	1	1	C2	0	C1	C0

            2. For C0, bits 3, 5, 7 are 0, 1, 0. XOR(010) = 1, so C0 = 1
            3. For C1, bits 3, 6, 7 are 0, 1, 0. XOR(010) = 1, so C1 = 1
            4. For C2, bits 5, 6, 7 are 1, 1, 0. XOR(110) = 0, so C2 = 0
        ii. Putting these together, we get 0110011
    c. Let's flip one of the bits now
        i. Doesn't matter which type of bit, algorithm works identically
            1. No differentiation between check bits and data bits being flipped
            2. Only care about how many bits get flipped with this algorithm
            3. In other words, only care about difference in Hamming distance
        ii. Is 0010011 valid?
            1. Fill in the table, then verify check bits

111	110	101	100	011	010	001
7	6	5	4	3	2	1
D3	D2	D1	C2	D0	C1	C0
0	0	1	0	0	1	1

            2. For C0, bits 3, 5, 7 are 0, 1, 0. XOR(010) = 1, so C0 = 1
            3. For C1, bits 3, 6, 7 are 0, 0, 0. XOR(000) = 0, so C1 = 0
            4. For C2, bits 5, 6, 7 are 1, 0, 0. XOR(100) = 1, so C2 = 1
        iii. Was codeword valid?
            1. We have a mismatch with C1 and C2, thus the codeword was invalid
                a. We calculated C1 = 0, but the bit received was 1
                b. Same applies for C2 = 1, but bit received was 0

- d. Error correction portion
    - i. We know there was an error, how do we fix it?
        1. XOR the generated check bits with the check bits from the received word
        2. Result tells us exactly where the error occurred
            - a. This is possible because of the way we've laid out the check bits
            - b. This is also why we started with 1 instead of 0 when numbering the bits
    - ii. XORing each bit together:

Received:	011
Calculated:	<u>101</u>
XOR:	110

    - iii. So we know bit position 6 was the error, we correct that one
        1. We get 0110011, which was our original code word before we flipped anything
        2. Extracting the data, we get 0110, which was our original data