

ECE120: Introduction to Computer Engineering **Fall 2014**
Notes Set 1.5

Redundancy and Coding

This set of notes introduces the idea of using sparsely populated representations to protect against accidental changes to bits. Today, such representations are used in almost every type of storage system, from bits on a chip to main memory to disk to archival tapes. We begin our discussion with examples of representations in which some bit patterns have no meaning, then consider what happens when a bit changes accidentally. We next outline a general scheme that allows a digital system to detect a single bit error. Building on the mechanism underlying this scheme, we describe a distance metric that enables us to think more broadly about both detecting and correcting such errors, and then show a general approach that allows correction of a single bit error. We leave discussion of more sophisticated schemes to classes on coding and information theory.

Sparse Representations

Representations used by computers must avoid ambiguity: a single bit pattern in a representation cannot be used to represent more than one value. However, the converse need not be true. A representation can have several bit patterns representing the same value, and not all bit patterns in a representation need be used to represent values.

Let's consider a few example of representations with unused patterns. Historically, one common class of representations of this type was those used to represent individual decimal digits. We examine three examples from this class.

The first is Binary-coded Decimal (BCD), in which decimal digits are encoded individually using their representations in the unsigned (binary) representation. Since we have 10 decimal digits, we need 10 patterns, and four bits for each digit. But four bits allow $2^4 = 16$ bit patterns. In BCD, the patterns $1010, 1011, \ldots, 1111$ are unused. It is important to note that BCD is not the same as the unsigned representation. The decimal number 732, for example, requires 12 bits when encoded as BCD: 0111 0011 0010. When written using a 12-bit unsigned representation, 732 is written 001011011100. Operations on BCD values were implemented in early processors, including the 8086, and are thus still available in the x86 instruction set architecture today!

The second example is an Excess-3 code, in which each decimal digit d is represented by the pattern corresponding to the 4-bit unsigned pattern for $d + 3$. For example, the digit 4 is represented as 0111, and the digit 7 is represented as 1010. The Excess-3 code has some attractive aspects when using simple hardware. For example, we can use a 4-bit binary adder to add two digits c and d represented in the Excess-3 code, and the carry out signal produced by the adder is the same as the carry out for the decimal addition, since $c + d \geq 10$ is equivalent to $(c + 3) + (d + 3) \geq 16$.

The third example of decimal digit representations is a 2-out-of-5 code. In such a code, five bits are used to encode each digit. Only patterns with exactly two 1s are used. There are exactly ten such patterns, and an example representation is shown to the right (more than one assignment of values to patterns has been used in real systems).

digit	a 2-out-of-5 representation
1	00011
2	00101
3	00110
4	01001
5	01010
6	01100
7	10001
8	10010
9	10100
0	11000

Error Detection

Errors in digital systems can occur for many reasons, ranging from cosmic ray strikes to defects in chip fabrication to errors in the design of the digital system. As a simple model, we assume that an error takes the form of changes to some number of bits. In other words, a bit that should have the value 0 instead has the value 1, or a bit that should have the value 1 instead has the value 0. Such an error is called a **bit error**.

Digital systems can be designed with or without tolerance to errors. When an error occurs, no notification nor identification of the error is provided. Rather, if error tolerance is needed, the system must be designed to be able to recognize and identify errors automatically. Often, we assume that each of the bits may be in error independently of all of the others, each with some low probability. With such an assumption, multiple bit errors are much less likely than single bit errors, and we can focus on designs that tolerate a single bit error. When a bit error occurs, however, we must assume that it can happen to any of the bits.

The use of many patterns to represent a smaller number of values, as is the case in a 2-out-of-5 code, enables a system to perform **error detection**. Let's consider what happens when a value represented using a 2-out-of-5 code is subjected to a single bit error. Imagine that we have the digit 7. In the table on the previous page, notice that the digit 7 is represented with the pattern 10001.

As we mentioned, we must assume that the bit error can occur in any of the five bits, thus we have five possible bit patterns after the error occurs. If the error occurs in the first bit, we have the pattern 00001. If the error occurs in the second bit, we have the pattern 11001. The complete set of possible error patterns is 00001, 11001, 10101, 10011, and 10000.

Notice that none of the possible error patterns has exactly two 1s, and thus none of them is a meaningful pattern in our 2-out-of-5 code. In other words, whenever a digital system represents the digit 7 and a single bit error occurs, the system will be able to detect that an error has occurred.

What if the system needs to represent a different digit? Regardless of which digit is represented, the pattern with no errors has exactly two 1s, by the definition of our representation. If we then flip one of the five bits by subjecting it to a bit error, the resulting error pattern has either one 1 (if the bit error changes a 1 to a 0) or three 1s (if the bit error changes a 0 to a 1). In other words, regardless of which digit is represented, and regardless of which bit has an error, the resulting error pattern never has a meaning in the 2-out-of-5 code. So this representation enables a digital system to detect any single bit error!

Parity

The ability to detect any single bit error is certainly useful. However, so far we have only shown how to protect ourselves when we want to represent decimal digits. Do we need to develop a separate error-tolerant representation for every type of information that we might want to represent? Or can we instead come up with a more general approach? The answer to the second question is yes: we can, in fact, systematically transform any representation into a representation that allows detection of a single bit error. The key to this transformation is the idea of **parity**.

Consider an arbitrary representation for some type of information. For each pattern used in the representation, we can count the number of 1s. The resulting count is either odd or even. By adding an extra bit—called a **parity bit**—to the representation, and selecting the parity bit's value appropriately for each bit pattern, we can ensure that the count of 1s is odd (called **odd parity**) or even (called **even parity**) for all values represented. The idea is illustrated in the table to the right for the 3-bit unsigned representation. The parity bits are shown in bold.

value represented	3-bit unsigned	number of 1s	with odd parity	with even parity
0	000	0	000**1**	000**0**
1	001	1	001**0**	001**1**
2	010	1	010**0**	010**1**
3	011	2	011**1**	011**0**
4	100	1	100**0**	100**1**
5	101	2	101**1**	101**0**
6	110	2	110**1**	110**0**
7	111	3	111**0**	111**1**

Either approach to selecting the parity bits ensures that any single bit error can be detected. For example, if we choose to use odd parity, a single bit error changes either a 0 into a 1 or a 1 into a 0. The number of 1s in the resulting error pattern thus differs by exactly one from the original pattern, and the parity of the error pattern is even. But all valid patterns have odd parity, so any single bit error can be detected by simply counting the number of 1s.

Hamming Distance

Next, let's think about how we might use representations—we might also think of them as **codes**—to protect a system against multiple bit errors. As we have seen with parity, one strategy that we can use to provide such error tolerance is the use of representations in which only some of the patterns actually represent values. Let's call such patterns **code words**. In other words, the code words in a representation are those patterns that correspond to real values of information. Other patterns in the representation have no meaning.

As a tool to help us understand error tolerance, let's define a measure of the distance between code words in a representation. Given two code words X and Y, we can calculate the number $N_{X,Y}$ of bits that must change to transform X into Y. Such a calculation merely requires that we compare the patterns bit by bit and count the number of places in which they differ. Notice that this relationship is symmetric: the same number of changes are required to transform Y into X, so $N_{Y,X} = N_{X,Y}$. We refer to this number $N_{X,Y}$ as the **Hamming distance** between code word X and code word Y. The metric is named after Richard Hamming, a computing pioneer and an alumnus of the UIUC Math department.

The Hamming distance between two code words tells us how many bit errors are necessary in order for a digital system to mistake one code word for the other. Given a representation, we can calculate the minimum Hamming distance between any pair of code words used by the representation. The result is called the **Hamming distance of the representation**, and represents the minimum of bit errors that must occur before a system might fail to detect errors in a stored value.

The Hamming distance for nearly all of the representations that we introduced in earlier sections is 1. Since more than half of the patterns (and often all of the patterns!) correspond to meaningful values, some pairs of code words must differ in only one bit, and these representations cannot tolerate any errors. For example, the decimal value 42 is stored as 101010 using a 6-bit unsigned representation, but any bit error in that pattern produces another valid pattern corresponding to one of the following decimal numbers: 10, 58, 34, 46, 40, 43. Note that the Hamming distance between any two patterns is not necessarily 1. Rather, the Hamming distance of the unsigned representation, which corresponds to the minimum between any pair of valid patterns, is 1.

In contrast, the Hamming distance of the 2-out-of-5 code that we discussed earlier is 2. Similarly, the Hamming distance of any representation extended with a parity bit is at least 2.

Now let's think about the problem slightly differently. Given a particular representation, how many bit errors can we detect in values using that representation? *A representation with Hamming distance d can detect up to $d-1$ bit errors.* To understand this claim, start by selecting a code word from the representation and changing up to $d-1$ of the bits. No matter how one chooses to change the bits, these changes cannot result in another code word, since we know that any other code word has to require at least d changes from our original code word, by the definition of the representation's Hamming distance. A digital system using the representation can thus detect up to $d-1$ errors. However, if d or more errors occur, the system might sometimes fail to detect any error in the stored value.

Error Correction

Detection of errors is important, but may sometimes not be enough. What can a digital system do when it detects an error? In some cases, the system may be able to find the original value elsewhere, or may be able to re-compute the value from other values. In other cases, the value is simply lost, and the digital system may need to reboot or even shut down until a human can attend to it. Many real systems cannot afford such a luxury. Life-critical systems such as medical equipment and airplanes should not turn themselves off and wait for a human's attention. Space vehicles face a similar dilemma, since no human may be able to reach them.

Can we use a strategy similar to the one that we have developed for error detection in order to try to perform **error correction**, recovering the original value? Yes, but the overhead—the extra bits that we need to provide such functionality—is higher.

Let's start by thinking about a code with Hamming distance 2, such as 4-bit 2's complement with odd parity. We know that such a code can detect one bit error. Can it correct such a bit error, too?

Imagine that a system has stored the decimal value 6 using the pattern 0110**1**, where the last bit is the odd parity bit. A bit error occurs, changing the stored pattern to 0111**1**, which is not a valid pattern, since it has an even number of 1s. But can the system know that the original value stored was 6? No, it cannot. The original value may also have been 7, in which case the original pattern was 0111**0**, and the bit error occurred in the final bit. The original value may also have been -1, 3, or 5. The system has no way of resolving this ambiguity. The same problem arises if a digital system uses a code with Hamming distance d to detect up to $d-1$ errors.

Error correction is possible, however, if we assume that fewer bit errors occur (or if we instead use a representation with a larger Hamming distance). As a simple example, let's create a representation for the numbers 0 through 3 by making three copies of the 2-bit unsigned representation, as shown to the right. The Hamming distance of the resulting code is 3, so any two bit errors can be detected. However, this code also enables us to correct a single bit error. Intuitively, think of the three copies as voting on the right answer. Since a single bit error can only corrupt one copy, a majority vote always gives the right answer! Tripling the number of bits needed in a representation is not a good general strategy, however. Notice also that "correcting" a pattern with two bit errors can produce the wrong result.

value represented	three-copy code
0	000000
1	010101
2	101010
3	111111

Let's think about the problem in terms of Hamming distance. Assume that we use a code with Hamming distance d and imagine that up to k bit errors affect a stored value. The resulting pattern then falls within a neighborhood of distance k from the original code word. This neighborhood contains all bit patterns within Hamming distance k of the original pattern. We can define such a neighborhood around each code word. Now, since d bit errors are needed to transform a code word into any other code word, these neighborhoods are disjoint so long as $2k \leq d-1$. In other words, if the inequality holds, any bit pattern in the representation can be in at most one code word's neighborhood. The digital system can then correct the errors by selecting the unique value identified by the associated neighborhood. Note that patterns encountered as a result of up to k bit errors always fall within the original code word's neighborhood; the inequality ensures that the neighborhood identified in this way is unique. We can manipulate the inequality to express the number of errors k that can be corrected in terms of the Hamming distance d of the code. *A code with Hamming distance d allows up to $\lfloor \frac{d-1}{2} \rfloor$ errors to be corrected*, where $\lfloor x \rfloor$ represents the integer floor function on x, or rounding x down to the nearest integer.

Hamming Codes

Hamming also developed a general and efficient approach for extending an arbitrary representation to allow correction of a single bit error. The approach yields codes with Hamming distance 3. To understand how a **Hamming code** works, think of the bits in the representation as being numbered starting from 1. For example, if we have seven bits in the code, we might write a bit pattern X as $x_7x_6x_5x_4x_3x_2x_1$.

The bits with indices that are powers of two are parity check bits. These include x_1, x_2, x_4, x_8, and so forth. The remaining bits can be used to hold data. For example, we could use a 7-bit Hamming code and map the bits from a 4-bit unsigned representation into bits x_7, x_6, x_5, and x_3. Notice that Hamming codes are not so useful for small numbers of bits, but require only logarithmic overhead for large numbers of bits. That is, in an N-bit Hamming code, only $\lceil \log_2(N+1) \rceil$ bits are used for parity checks.

How are the parity checks defined? Each parity bit is used to provide even parity for those bits with indices for which the index, when written in binary, includes a 1 in the single position in which the parity bit's index contains a 1. The x_1 bit, for example, provides even parity on all bits with odd indices. The x_2 bit provides even parity on x_2, x_3, x_6, x_7, x_{10}, and so forth.

In a 7-bit Hamming code, for example, x_1 is chosen so that it has even parity together with x_3, x_5, and x_7. Similarly, x_2 is chosen so that it has even parity together with x_3, x_6, and x_7. Finally, x_4 is chosen so that it has even parity together with x_5, x_6, and x_7.

The table to the right shows the result of embedding a 4-bit unsigned representation into a 7-bit Hamming code.

value represented	4-bit unsigned $(x_7x_6x_5x_3)$	x_4	x_2	x_1	7-bit Hamming code
0	0000	0	0	0	0000000
1	0001	0	1	1	0000111
2	0010	1	0	1	0011001
3	0011	1	1	0	0011110
4	0100	1	1	0	0101010
5	0101	1	0	1	0101101
6	0110	0	1	1	0110011
7	0111	0	0	0	0110100
8	1000	1	1	1	1001011
9	1001	1	0	0	1001100
10	1010	0	1	0	1010010
11	1011	0	0	1	1010101
12	1100	0	0	1	1100001
13	1101	0	1	0	1100110
14	1110	1	0	0	1111000
15	1111	1	1	1	1111111

A Hamming code provides a convenient way to identify which bit should be corrected when a single bit error occurs. Notice that each bit is protected by a unique subset of the parity bits corresponding to the binary form of the bit's index. Bit x_6, for example, is protected by bits x_4 and x_2, because the number 6 is written 110 in binary. If a bit is affected by an error, the parity bits that register the error are those corresponding to 1s in the binary number of the index. So if we calculate check bits as 1 to represent an error (odd parity) and 0 to represent no error (even parity), then concatenate those bits into a binary number, we obtain the binary value of the index of the single bit affected by an error (or the number 0 if no error has occurred).

Let's do a couple of examples based on the pattern for the decimal number 9, 1001100. First, assume that no error occurs. We calculate check bit c_4 by checking whether x_4, x_5, x_6, and x_7 together have even parity. Since no error occurred, they do, so $c_4 = 1$. Similarly, for c_2 we consider x_2, x_3, x_6, and x_7. These also have even parity, so $c_2 = 0$. Finally, for c_1, we consider x_1, x_3, x_5, and x_7. As with the others, these together have even parity, so $c_1 = 0$. Writing $c_4c_2c_1$, we obtain 000, and conclude that no error has occurred.

Next assume that bit 3 has an error, giving us the pattern 1001000. In this case, we have again that $c_4 = 0$, but the bits corresponding to both c_2 and c_1 have odd parity, so $c2 = 1$ and $c_1 = 1$. Now when we write the check bits $c_4c_2c_1$, we obtain 011, and we are able to recognize that bit 3 has been changed.

A Hamming code can only correct one bit error, however. If two bit errors occur, correction will produce the wrong answer. Let's imagine that both bits 3 and 5 have been flipped in our example pattern for the decimal number 9, producing the pattern 1011000. Calculating the check bits as before and writing them as $c_4c_2c_1$, we obtain 110, which leads us to incorrectly conclude that bit 6 has been flipped. As a result, we "correct" the pattern to 1111000, which represents the decimal number 14.

SEC-DED Codes

We now consider one final extension of Hamming codes to enable a system to perform single error correction while also detecting any two bit errors. Such codes are known as **Single Error Correction, Double Error Detection (SEC-DED)** codes. Creating such a code from a Hamming code is trivial: add a parity bit covering the entire Hamming code. The extra parity bit increases the Hamming distance to 4. A Hamming distance of 4 still allows only single bit error correction, but avoids the problem of Hamming distance 3 codes when two bit errors occur, since patterns at Hamming distance 2 from a valid code word cannot be within distance 1 of another code word, and thus cannot be "corrected" to the wrong result.

In fact, one can add a parity bit to any representation with an odd Hamming distance to create a new representation with Hamming distance one greater than the original representation. To proof this convenient fact, begin with a representation with Hamming distance d, where d is odd. If we choose two code words from the representation, and their Hamming distance is already greater than d, their distance in the new representation will also be greater than d. Adding a parity bit cannot decrease the distance. On the other hand, if the two code words are exactly distance d apart, they must have opposite parity, since they differ by an odd number of bits. Thus the new parity bit will be a 0 for one of the code words and a 1 for the other, increasing the Hamming distance to $d + 1$ in the new representation. Since all pairs of code words have Hamming distance of at least $d + 1$, the new representation also has Hamming distance $d + 1$.