Number Systems

Inside today's computers, data is represented as 1's and 0's. These 1's and 0's might be stored magnetically on a disk, or as a state in a transistor. To perform useful operations on these 1's and 0's we have to organize them together into patterns that make up codes. This lecture is about ways to represent data in the codes and to illustrate different number systems that are used today. This is how data is fundamentally stored in the computer.

Each 1 and 0 is called a **bit**. 4 bits is called a **nibble**. 8 bits is called a **byte**. 16/32/64 bits is called a **word**. The number depends upon the number of bits that can simultaneously be operated upon by the processor.

Since we only have 1's and 0's available, the number system we will use inside the computer is the **binary** number system. We are used to the **decimal** number system. In decimal, we have ten symbols (0-9) that we can use. When all of the symbols are used up, we need to combine together two symbols to represent the next number (10). The most significant digit becomes our first digit, and the least significant digit cycles back to 0. The least significant digit increments until the symbols are used up, and which point the most significant digit is incremented to the next larger digit. This is just the familiar process of counting!

The same process applies to the binary number system, but we only have two symbols available (0 and 1). When we run out of symbols, we need to combine them and increment the digits just as we would in a decimal, base 10, number system: Let's start counting to illustrate:

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001

Binary to Decimal Conversion

The binary number system is positional where each binary digit has a weight based upon its position relative to the least significant bit (LSB). To convert a binary number to decimal, sum together the weights. Each weight is calculated by multiplying the bit by 2^i, where i is the index relative to the LSB.

1	1	0	1	1
2^4	2^3	2^2	2^1	2^0

Weight = $1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 + 1*2^4$

= 1*1 + 1*2 + 0*4 + 1*8 + 1*16 = 1+2+8+16 = 27 (base 10, decimal)

What is 11111 in base 10?
What is 10110101 in base 10?

Decimal to Binary Conversion

To convert a decimal number to its equivalent binary number representation, the easiest method is to use repeated division by 2. Each time, save the remainder. The first remainder is the least significant bit. The last remainder is the most significant bit.

What is 27 in binary?

27 / 2 = 13	remainder 1	LSB
13 / 2 = 6	remainder 1	
6 / 2 = 3	remainder 0	
3 / 2 = 1	remainder 1	
1 / 2 = 0	remainder 1	MSB

Our final number is 11011.

If using a calculator, watch to see if the quotient has a fraction. If it is 0.5 then the remainder was 1, and repeat the process with the whole part minus the fraction.

What is 83 in binary?
What is 255 in binary?

Hexadecimal Numbering

The hexadecimal numbering system is the most common system seen today in representing raw computer data. This is because it is very convenient to represent 8 bits, or one byte of data.

Hexadecimal uses a base 16 numbering system. This means that we have 16 symbols to use for digits. Consequently, we must invent new digits beyond 9. The digits used in hex are the letters A,B,C,D,E, and F. If we start counting, we get the table below:

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
18	…	

Hex to Decimal Conversion

Converting hex to decimal is just like converting binary to decimal, except instead of powers of 2, we use powers of 16. That is, the LSB is 16^0, the next is 16^1, then 16^2, etc.

To convert 35F in hex to decimal:

3 5 F
16^2 16^1 16^0

Weight = $15*16^0 + 5*16^1 + 3*16^2$
= $15*1 + 5*16 + 3*256$
= 863 base 10

What is F00D in octal converted to decimal?

Decimal to Hex Conversion

Converting decimal to hex is just like converting decimal to binary, except instead of dividing by 2, we divide by 16. To convert 254 to hex:

254 / 16 = 15 remainder 14 = E
15 / 16 = 0 remainder 15 = F
So we get FE for 254.

Again, note that if a calculator is being used, you may multiple the fraction remainder by 16 to produce the remainder.

What is 423 in hex?

Hex to Binary Conversion, Binary to Hex

Going from hex to binary is similar to the process of converting from octal to binary. One must simply look up or compute the binary pattern of 4 bits for each hex code, and concatenate the codes together.

To convert FE to binary:

F = 1111
E = 1110
So FE in binary is 1111 1110

The same process applies in reverse by grouping together 4 bits at a time and then look up the hex digit for each group.

Binary 11000100101 broken up into groups of 4:
0110 0010 0101 (note the 0 added as padding on the MSB to get up to 4 bits)
6 2 5
= 625 Hex

Alphanumeric Codes

So far we have only been discussing the representation of numbers. We would also like to represent alphanumeric characters. This includes letters, numbers, punctuation marks, and other special characters like % or / that appear on a typical keyboard.

The most common code in the United States is ASCII, the American Standard Code for Information Interchange. ASCII is comprised of a 7 bit code. This means that we have 2^7 or 128 possible code groups. This is plenty for English as well as control functions such as the carriage return or linefeed functions.

Another code is EBCDIC, extended binary coded decimal interchange code. EBCDIC was used on some mainframe systems but is now out of favor.

Page 128 of your book has a complete listing of ASCII codes. A small sample is shown below.

Character	ASCII Hex	ASCII binary	Integer Equivalent
A	41	01000001	65
B	42	01000010	66
C	43	01000011	67
Z	5A	01011010	90

Note that the sequence of bits "01000001" could represent many things: it could be the number 65, or it might be the letter 'A'. Or it could be something else entirely different! Ultimately it is up to the software in the computer to know what a sequence of bits represents and treat it accordingly.

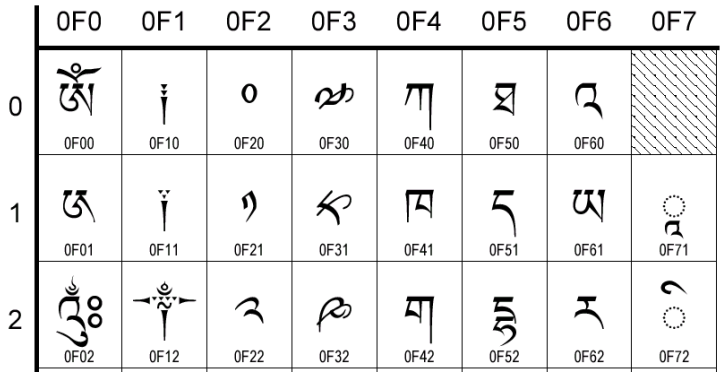
Since most computers store data in bytes, or 8 bits, there is 1 bit that may go unused (or used for parity, described below).

Could we represent something like Chinese in ASCII, which has many more than 128 characters? The answer is we can't, because there are way more than 128 characters in Chinese.

One way to represent languages such as Chinese is to use a different code. UNICODE is an international standard used to represent multiple languages. To address the shortcomings of ASCII, UNICODE uses 16 bits per character. This allows 2^{16} or 65536 characters. When this is insufficient, there are extensions to support over a million characters. Most of the work in UNICODE has gone in efforts to define the codes for the particular languages. If you would like more information, visit http://www.unicode.org.

Some example Unicode characters for Tibetan are shown below.

Tibetan



Side note: A common problem with software development occurs when a product needs to support foreign languages. Many products in the US are initially developed assuming ASCII. However, if the product is successful and needs to support something like French or German, it is often a difficult task to move the program from ASCII to UNICODE. Software developers need to carefully take into account future needs and design the software appropriately. If it is known up front that other languages may need to be supported, it is a relatively simple matter to support UNICODE up front , or to remove dependencies on ASCII.

Parity Method for Error Detection

It is very possible for errors to occur when reading or transmitting some data. A CD might have a speck of dust, turning a 0 into a 1. A radio transmission might encounter some electromagnetic interference. A magnetic disk might be going bad. There are many cases where there is a possibility that errors will occur such that the receiver does not receive the identical information that is stored or sent. Although the possibility of error in modern systems is fairly low, with gigabytes of data being sent, the probability of encountering one of these errors is high.

One of the simplest schemes to detect, and sometimes correct, errors is the parity method. The parity method simply attaches a parity bit to a code group being sent. There are two types, *even parity* or *odd parity*.

In even parity the value of the parity bit is chosen so that the total number of 1's in the group is an even number, including the parity bit.

In odd parity the value of the parity bit is chosen so that the total number of 1's in the group is an odd number, including the parity bit.

Consider the ASCII code for the letter A. The code is 1000001. If we are using even parity, we want to select a parity bit so that the total number of 1 bits is even. Since there are already an even number of 1's, we must make the parity bit 0 so that there is still an even number of 1's including the parity. The data that would be transmitted is then 01000001.

If using odd parity, we want to select a parity bit so that the total number of 1 bits is odd. This means that for 1000001 we need to make the parity bit be a 1 to make the total number of 1's be odd. The data that would be transmitted is then 11000001.

Let's say there was an error during transmission of the letter A, and the receiver got 11000011. The receiver will count up the number of 1's and see that it is even. However, using odd parity we should never have an even number of 1's. Therefore, an error occurred and the receiver may ask the sender to re-transmit the data.

Note that this parity method only works for *single bit errors*. If there are multiple errors we may be out of luck. Unfortunately, this is often the case due to the bursty nature of most errors. For these cases, we need stronger methods of error detection and correction such as a checksum, cyclic redundancy check or LRC. (which we won't describe here).

Using even parity, what would be the parity bit for 111010110100 ?

Representing images, sound

Our pattern of bits can represent more than just numbers. They can represent anything we want them to as long as the meaning can be mapped to a sequence of bits.

Images are represented by a bitmap, or a sequence of bits that represent different colors. (Todo: more detail on bitmap representation in class)

Sound is represented as a sample of intensities from a sound wave.