Binary and Hexadecimal
(A chapter from “A Book About Waves”)

by Tim Warriner

www.timwarriner.com
[Last edited on 2024-02-25]

To see if this is the latest version of this chapter, visit www.timwarriner.com

Binary and hexadecimal

This excerpt from my book about waves explains how computers store and use numbers.

Binary

Computers count in binary, and therefore, their methods of storing numbers are based around binary. Binary is a counting system where every digit is either zero or one. Normally, we count in decimal, where every digit is 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9. As we know, the progression of counting in decimal is as so:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 and so on.

The basic rule for counting in decimal is that when the rightmost digit is going to pass 9, we reset the digit to 0, and add 1 to the digit to the left. If *that* digit is going to pass 9, then we set that to 0, and add 1 to the digit to the left, and so on.

Counting in binary works in a similar way, but as binary has only two different digits (zero and one), counting progresses as so:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100 and so on.

When the rightmost digit of a number is going to be above 1, that digit becomes 0, and the digit to the left is raised by 1. If *that* digit is going to be above 1, then it becomes 1, and the digit to the left is raised by 1, and so on. Counting in binary is based on identical ideas as counting in decimal.

When we have a number in *decimal*, we can think of each digit as being in a "column" representing, from right to left, ones, tens, hundreds, thousands, ten thousands and so on. Each column is ten times the one before it. For example, the number "245" has 2 in the hundreds column, 4 in the tens column, and 5 in the ones column. The number "245" is really: 2 * 100, added to 4 * 10, added to 5 * 1.

We can think in the same way about binary. From right to left in binary, the columns are ones, twos, fours, eights, sixteens, thirty-twos, sixty-fours, one hundred and twenty eights, and so on. Each column is twice the one before it. As an example, the binary number "1100" has 1 in the eights column, 1 in the fours column, 0 in the twos column, and 0 in the ones column. The binary number "1100" is really:
1 * 8
... added to:
1 * 4
... added to:
0 * 2
... added to:
0 * 1
... which makes 12 in decimal.

The decimal numbers 0 to 15 and their binary equivalents are as so:

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

A binary digit is called a “bit”. Therefore, we can say that the number “111001” contains 6 bits. This is the same as saying that the number “111001” has 6 digits.

Electronics can use the binary counting system because the system works well with switches, whether they are real mechanical ones or ones controlled by circuitry. We can portray a binary number by having a series of switches set to on or off. For example, we could portray the number “1010” by having four switches, with the first and third switched on, and the second and fourth switched off. In digital electronics, we can portray binary numbers in a similar way by having parallel lines with different voltages. We can have a voltage being set to, say, 5 volts to represent a one, and a voltage set to 0 volts to represent a zero. By using just two different states, we avoid any confusion because a state represents either a one or a zero. There is nothing between the states.

Groups of bits

Although digital electronics can use binary numbers of any chosen length, when it comes to computers, binary numbers are grouped into particularly sized sets of digits. Computer processors are designed to operate most quickly when dealing with a particular sized group of bits. The size of the group depends on the processor. [Away from computers, you can use any grouping that you want.]

[The following size names are those used by Intel when referring to 80x86 and 64-bit computer processors. These are common modern definitions of the terms, but some other, especially older, computer architectures define things differently. If you ask someone how big a “word” is, an Intel assembly language programmer will say 16 bits. Someone else might say, “It depends...”]

Bytes

A “byte” is a group of 8 bits. In other words, a byte is a binary number that is 8 digits long. To put this another way, a number that is stored as a byte will always be 8 bits long, even if the higher bits, or even all the bits, are zero. When a computer reads one byte, it will always read 8 bits. It will never read fewer or more. Given that, if we are writing out a binary number as a byte, it makes sense to include any preceding zeroes. If a decimal number is converted to a byte, the result will end up as an 8-bit number.

As a byte is always 8 bits long, the binary numbers that can be contained within a byte are as follows:

Binary number	**Decimal Equivalent**
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
00000101	5
00000110	6
00000111	7
00001000	8
00001001	9
00001010	10
00001011	11
00001100	12
00001101	13
00001110	14
00001111	15
00010000	16
00010001	17
... and so on until:	
11111100	252
11111101	253
11111110	254
11111111	255

The highest number that can be stored as a byte is 11111111 in binary, which is 255 in decimal. As we start at 00000000 in binary, this means that we can actually store 256 different values as a byte. [If we were using a byte to indicate the position of something in a list, it makes sense to treat the first item in the list as item 0, instead of item 1, as that means we can distinguish between 256 different items instead of 255.]

Supposing we were dealing with binary numbers on a piece of paper, if we were to add 1 to the binary number 11111111, we would end up with the number 100000000, which is 9 digits long. However, if we are dealing with binary numbers on a computer, and dealing with bytes, if we added 1 to 11111111, we would end

up with 00000000, which is zero. This is because we cannot have more than 8 digits in a byte. The 1 that would be the ninth digit becomes lost, and the whole number rolls over to zero. This is the same effect as when a milometer [odometer] in a vehicle, or a meter on a fuel pump, reaches its maximum value and rolls over to zero.

By only using numbers with a set number of digits, computers can work more efficiently. A byte is generally the smallest grouping of bits with which a computer processor can work quickly. It is usually possible for a processor to read individual bits or half bytes, but to do so, it would first have to load the number into a byte or larger number grouping, and the process is slower than dealing with bytes.

Half bytes

A “half byte” is a 4-bit number – it is a grouping of 4 bits. A half byte is always 4 bits long. It is half the length of a byte. A half byte is sometimes called a “nibble”. The binary numbers that can fit into a half byte are as follows:

Binary number	**Decimal Equivalent**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

A half byte can contain the binary digits 0000 to 1111. The maximum number it can hold is the binary equivalent of the decimal number 15. As it can hold the number 0, it is capable of holding 16 different values.

On Intel 32-bit and 64-bit processors, it is possible to access a half byte easily, but the number must first be placed into a byte or a larger number type. It is slower to use half bytes than it is to use bytes.

Words

A "word" is a grouping of 16 bits as one entity. A word is twice as long as a byte. The binary numbers that can be contained in a word are as follows:

Binary number	**Decimal Equivalent**
0000000000000000	0
0000000000000001	1
0000000000000010	2
0000000000000011	3
0000000000000100	4
0000000000000101	5
0000000000000110	6
0000000000000111	7
0000000000001000	8
0000000000001001	9
0000000000001010	10
0000000000001011	11
... and so on until:	
1111111111111100	65,532
1111111111111101	65,533
1111111111111110	65,534
1111111111111111	65,535

A word can hold values from 0 up to 65,535. This means that it can hold 65,536 different values.

[Note that a few people use the term "word" to describe *any* grouping of bits. You might see this in older books. Intel have been using the term "word" to mean a grouping of 16 bits for at least thirty years.]

The rule for finding out how many different values can be stored in a binary number, based on the bit size, is to raise 2 to the power of the bit size. For example, a 16-bit binary number can hold: 2^{16} = 65,536 different values. Note, however, that this is not the maximum value because we start at zero. The maximum value is always 1 less than the number of available values. For example, an 8-bit binary number can hold 2^8 = 256 values, but the maximum value will be $2^8 - 1 = 255$.

Computer processors can be categorised according to the maximum number of bits they can deal with in one go. For example, if we had a processor that could only operate with bytes (8-bit numbers), it would be referred to as an "8-bit processor". The longest number an 8-bit processor can read in one go is 8 bits long. Generally, an 8-bit processor will be fastest at dealing with 8-bit numbers. It will be able to read 8 bits in one go, store 8 bits in one go, and perform calculations on 8 bits in one go. If a processor is a 16-bit processor, the largest and optimal size of number it can deal with is 16 bits long. The processor will be best at operating with 16-bit numbers. It will be able to read and work with numbers most efficiently if they are 16 bits long. Note that a 16-bit processor will still be able to work with shorter numbers, but it will take slightly longer to do so. It will also be able to work with longer numbers, but it will require being told how to do so via a computer program, and it will take much longer. [This is analogous to using a calculator that can only count up to one hundred to perform maths with numbers over one hundred. We would have to split calculations into parts, and write down partial calculations on a piece of paper. We could do it, but it would be more effort.]

A common mistake that programmers used to make was to use of one of the binary number types, either to count with or as an indicator to a place in a list, and to forget about the limits of the number type they were using. For example, if we are using a 16-bit word to count the number of letters in a piece of text, and we count more than 65,535 letters, the number will roll around to zero, and carry on from there. This means that the total will be wrong. Similarly, if we had a list of names and we used a 16-bit word to identify the different entries, with the first entry being entry 0, we would only be able to have 65,536 different entries. If we tried to access the 65,537th entry, we would end up accessing the very first entry [entry 0] instead because we cannot count that high with 16-bit numbers. As modern computers more commonly use 64-bit processors, where the highest number is considerably larger, this is now a rarer problem. However, it still happens if the programmer, for some reason, chooses a smaller number type.

Doublewords

A “doubleword” or “dword” is 32 bits long. A dword can hold the binary numbers from:
00000000000000000000000000000000
... to:
11111111111111111111111111111111
... in binary.

The number of different values that can be expressed by a dword is:
2^{32} = 4,294,967,296

The highest number that can be expressed by a dword is $2^{32} - 1$ = 4,294,967,295

The binary numbers that can be held in a dword are as follows:

Binary number	**Decimal Equivalent**
00000000000000000000000000000000	0
00000000000000000000000000000001	1
00000000000000000000000000000010	2
00000000000000000000000000000011	3
... and so on until:	
11111111111111111111111111111100	4,294,967,292
11111111111111111111111111111101	4,294,967,293
11111111111111111111111111111110	4,294,967,294
11111111111111111111111111111111	4,294,967,295

An Intel 32-bit 80x86 processor works most efficiently with 32-bit numbers. It is fastest when reading, working on, and storing 32-bit numbers. Although the processor has commands to deal with words (16-bit numbers), bytes (8-bit numbers) and half bytes (4-bit numbers), and it can test if individual bits are one or zero, it is most efficient when handling 32-bit numbers. [Other 32-bit processors are likely to be similar.] If we wanted a 32-bit processor to work with a 64-bit number, we would have to do it in a slower, more contrived way by splitting the number into two 32-bit numbers and then using slightly more awkward maths.

Quadwords

A “quadword” or “qword” is 64 bits long. A qword can hold the binary numbers from:
0000000000000000000000000000000000000000000000000000000000000000
... to:
1111111111111111111111111111111111111111111111111111111111111111
... in binary.

The number of values that can be expressed by a qword is:
2^{64} = 18,446,744,073,709,551,616

The highest number that can be expressed by a qword is:
$2^{64} - 1$ = 18,446,744,073,709,551,615
This is roughly 18.4 million million million, which is $18.4 * 10^{18}$.

A 64-bit processor works most efficiently with 64-bit binary numbers. Intel 64-bit processors have commands that work with 64-bit qwords, 32-bit dwords, 16-bit words, and 8-bit bytes, and can test whether individual bits are 1 or 0, but they work best with 64-bit qwords. [Intel 64-bit processors can work with 4-bit half bytes but those half bytes must already be within a byte, word, dword or qword.]

Although a 64-bit processor might seem best because it works with large numbers, a consequence is that the storage of numbers requires more space. If we wanted to store the number “3” as a 64-bit qword, it would require 64 bits. If we wanted to store it as a 32-bit dword, it would require 32 bits. If we wanted to store it as a 16-bit word, it would require 16 bits. If we wanted to store it as an 8-bit byte, it would require just 8 bits. Storing smaller numbers as 64-bit values uses 8 times as much space as using 8-bit bytes. On the other hand, we cannot store a number larger than 255 in a byte.

If we wanted to store the number “3” with the minimum size possible, it would require only 2 bits because the decimal number “3” is “11” in binary. However, it is rare that a computer processor would have a grouping that was only two bits long. We could write the 2-bit number on a piece of paper, though, as then we would be unconstrained by the groupings of bits.

Double quadwords

It is also possible to have a 128-bit number, which is called a “double quadword” or “dqword”. Such numbers work in the same way as other numbers, but just contain more bits. To keep things simple, we will focus on bytes, words, dwords and qwords.

Programming

In higher-level programming languages such as C, C++, Java, Python and so on, the actual bit group sizes are usually hidden from the programmer. An advantage of this is that the programmer can use the same code on different types of processors. A disadvantage is that useful underlying aspects of the processor are hidden from the programmer, so software might not run as efficiently as it could. It is possible to be reasonably good at programming, but without knowing anything about bytes, words, dwords, qwords, or even binary.

Converting binary to decimal

It is rare that you will ever need to convert long binary numbers to decimal. You are much more likely to need to convert binary numbers to hexadecimal, which we will look at later in this chapter. It is a lot of effort to convert any remotely long binary number directly to decimal, and it is much easier to convert it to hexadecimal first, and then to decimal. Therefore, there is not much point in learning how to convert binary to decimal. Having said that, if you want to become proficient in using binary and hexadecimal, it is very helpful to know every 4-bit binary number’s decimal equivalent. Learning them is easy with practice, but whether you *need* to learn them depends on what you want to do. If you frequently use binary, you will end up learning the numbers anyway from seeing them so often.

The numbers are as follows:

Binary	Decimal	How to remember
0000	0	This is easy to remember.
0001	1	This is easy to remember.
0010	2	This is a 1 in the twos column.
0011	3	This is two 1s to the right.
0100	4	This is a 1 in the fours column.
0101	5	This is 4 + 1
0110	6	This is two 1s in the middle.
0111	7	This is three 1s, all to the right.
1000	8	This is a 1 in the eights column.
1001	9	This is 1 more than 8.
1010	10	This is 8 + 2
1011	11	This is 8 + 3
1100	12	This is 8 + 4, or two 1s to the left.
1101	13	This is 12 + 1
1110	14	This is three 1s, all to the left.
1111	15	This is four 1s.

If you can remember 0100 (four), 0110 (six), and 1100 (twelve), then it is easy to calculate the others in your head. When you are completely used to binary, it will become irrelevant to you whether a 4-bit binary number is written in binary or its decimal equivalent. You will see them as the same thing.

We will see how to convert from binary to decimal and back in the “Binary and hexadecimal” section later in this chapter.

Hexadecimal

Decimal is a counting system based around the number ten. Binary is a counting system based around the number two. Now we will look at hexadecimal, which is a number system that is based around the number sixteen. Each digit of a hexadecimal number can have 16 different values. The first ten, representing the decimal numbers from 0 to 9, are also the numbers 0 to 9. The next digits are “a”, “b”, “c”, “d”, “e” and “f”. Therefore, the letter “a” represents the decimal number ten, but in the form of only one digit. The letter “b” is the digit that represents the decimal number eleven. The letter “c” is the digit that represents the decimal number twelve, and so on.

The term “hexadecimal” is often abbreviated to the word “hex”.

The hexadecimal numbers from the equivalent of the decimal number 0 up to the decimal number 15 are as so:

Hexadecimal number	**Equivalent decimal number**
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
a	10
b	11
c	12
d	13
e	14
f	15

When we count in *decimal*, we have a ones column, a tens column, a hundreds column, a thousands column and so on. Each new column is ten times the previous one. When we count in binary, we have a ones column, a twos column, a fours column, an eights column and so on. Each new column is twice the previous one. When we count in hexadecimal, we have a ones column, a sixteens column, a 256s column, a 4096s column and so on. Each new column is 16 times the previous one. When the ones column in hexadecimal gets past "f", we set the ones column to zero, and add 1 to the sixteens column. If that makes the sixteens column go past "f", then that is set to zero, and 1 is added to the 256s column and so on.

Hexadecimal is easier to use than to explain. Here is a long list of hexadecimal numbers and their decimal equivalents so we can see how hexadecimal works. [It is worth reading this, but it is not worth trying to remember which hexadecimal value is equal to which decimal value]:

Hexadecimal number	**Equivalent decimal number**
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
a	10
b	11
c	12
d	13
e	14
f	15
10	16 [This is 1 times 16]
11	17 [This is 1 times 16, with 1 added on]
12	18
13	19
14	20
15	21
16	22

Hexadecimal number	**Equivalent decimal number**
17	23
18	24
19	25
1a	26
1b	27
1c	28
1d	29
1e	30
1f	31
20	32 [This is 2 times 16]
21	33
22	34
23	35
24	36
25	37
26	38
27	39
28	40
29	41
2a	42
2b	43
2c	44
2d	45
2e	46
2f	47
30	48 [This is 3 times 16]
31	49
... and so on, until:	
99	153
9a	154
9b	155
9c	156
9d	157
9e	158
9f	159
a0	160 [This is ten times 16]

Hexadecimal number	**Equivalent decimal number**
a1	161
a2	162
... and so on, until:	
f9	249
fa	250
fb	251
fc	252
fd	253
fe	254
ff	255
100	256 [This is 16 times 16]
101	257
102	258

Notation

Using decimal, binary and hexadecimal numbers together can be confusing if we do not identify in which number system a sequence of digits belongs. For example, without any context, the sequence of digits “111” could mean the decimal number “one hundred and eleven”, the binary number equivalent to the decimal number “seven”, or the hexadecimal number equivalent to the decimal number “273”.

There are three main ways to avoid such confusion. The first way is the most common, and is used in higher-level programming languages such as C. For this way, a hexadecimal number is prefixed with “0x”, a binary number is prefixed with “0b”, and a decimal number is left alone. Therefore, the hexadecimal number 111 would be written as so:
0x111

The binary number 111 would be written as so:
0b111

The decimal number 111 would be written as:
111

The second way is used in assembly language programming, specifically in programming for Intel and AMD processors, but probably for other processors too. This involves putting the letter “h” after a hexadecimal number, and if that number starts with a letter, putting a zero in front. [The zero stops it being confused with the name of a variable.] If we have a binary number, we put the letter “b” at the end. If we have a decimal number, we put the letter “d” at the end. Therefore, the hexadecimal number 111 is written as:
111h

The hexadecimal number “ff” becomes:
0ffh
... because the number starts with a letter, so we prefix it with a zero.

The binary number 111 becomes:
111b
[This cannot be confused with the hex number “111b” because, if we were using this system, all hex bytes would have an “h” after them. The hexadecimal number “111b” would be written as “111bh”.]

The decimal number 111 becomes:
111d
[Again, this cannot be confused with the hex number “111d” because the hex number would be “111dh”.]

The third way of distinguishing between the types of numbers is to put a subscript on each number to indicate the base. For example, 1011_2 for binary, $f5_{16}$ for hex, 23_{10} for decimal. The downsides to this method are:

- It is an effort to type subscripted text on computers.
- Sometimes the subscript can be misread as part of the number.
- Copying and pasting subscripted text sometimes loses the attribute of being subscripted, thus turning, say, 1011_2 into 10112.

Most people use the “0x” and “0b” method. In this book, I will use the 0x method for hex numbers, and rely on the context for binary and decimal numbers.

The first few hexadecimal numbers written with the “0x” prefix are as so:

0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xa
0xb
0xc
0xd
0xe
0xf
0x10
0x11
... and so on.

Sometimes, if the context is clear, it is tidier and easier to have the values without the prefix. For example, if we know that a list of values is in hexadecimal, then there is no need to distinguish each value with a prefix.

Binary and hexadecimal

There is a connection between binary and hexadecimal that makes hexadecimal useful. A group of 4 binary bits can be represented by exactly one hexadecimal digit. Therefore, instead of dealing with very long sequences of binary digits, we can group them into fours, and deal with shorter sequences of hexadecimal digits.

The way that hexadecimal can be used to represent binary numbers more succinctly is clearer in the following table.

Decimal	Binary	Hexadecimal
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xa
11	1011	0xb
12	1100	0xc
13	1101	0xd
14	1110	0xe
15	1111	0xf

When we convert from binary to hexadecimal, we turn 4 binary digits into just 1 hexadecimal digit.

The usefulness of hexadecimal is more obvious if we have a long sequence of binary digits such as this:
10110010000011010011111110100110

To convert this into hexadecimal, we first split it into 4-bit sections:
1011 0010 0000 1101 0011 1111 1010 0110

Then, we convert each 4-bit section into the relevant hexadecimal digit:
b 2 0 d 3 f a 6

Then, we join up the hexadecimal digits to create the final value:
0xb20d3fa6

We have turned a binary number with 32 digits into a hexadecimal number with 8 digits. To put this another way, we have turned a 32-bit binary number into a 32-bit hexadecimal number. By calling the hex number "32-bit", despite it containing 8 digits, we are still acknowledging that it represents 32 binary bits. We are really just using hexadecimal to make long binary numbers more palatable. Hexadecimal is easier to read and remember than binary, but its ultimate meaning is the same.

Converting between hex and decimal

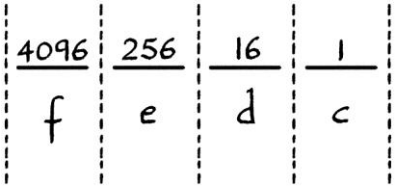
Depending on what it is we are trying to do, generally, there is less need to convert from hexadecimal to decimal and back, than there is to convert from hexadecimal to binary and back. In most situations, it is easier to remain in the world of hexadecimal than it is to swap to decimal and back. It is easy to know if one hexadecimal number is higher than another, which is what is usually needed. If you need to convert a long hexadecimal number to decimal, it is easiest to use a calculator to do it, than to bother trying to do it in your head. However, even if you always intend to use calculators, it is good to know the hexadecimal values 0x0 to 0xf by heart.

Converting between decimal and hexadecimal is not in any way difficult, but it takes more time than using a hex calculator. Before we do the conversion, we first have to be aware of the digit columns for a hexadecimal number, which from right to left are:

- 1s
- 16s
- 256s
- 4096s
- 65,536s

... and so on. Each column is 16 times the one before it.

If we had the number 0xfedc, it would have 0xf in the 4096s column, 0xe in the 256s column, 0xd in the 16s column, and 0xc in the 1s column.



The simplest way to convert from *hexadecimal to decimal* is to read each digit from each column, convert it to decimal, multiply it by the column it is in (1s, 16s, 256s, 4096s and so on), and then add up all the multiplications. For the number 0xfedc, we would proceed as follows:

The digit "f" is in the 4096s column. 0xf in hex is 15 in decimal. Therefore, we multiply 15 by 4096 to get 61,440.

The digit "e" is in the 256s column. 0xe in hex is 14 in decimal. Therefore, we multiply 14 by 256 to get 3,584.

The digit "d" is in the 16s column. 0xd in hex is 13 in decimal. Therefore, we multiply 13 by 16 to get 208.

The digit "c" is in the 1s column. 0xc in hex is 12 in decimal. Therefore, we multiply 12 by 1 to get 12.

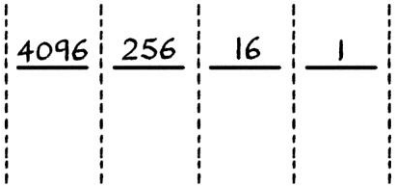
We then add up the results of the four multiplications:
61,440 + 3,584 + 208 + 12 = 65,244

We now know that 0xfedc is equal to 65,244 in decimal.

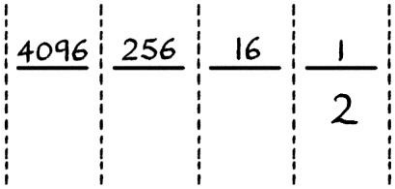
Converting from *decimal to hexadecimal* is easiest using a different type of method. The process is as follows:

- We divide the number by 16. We take the remainder, convert it into a hex digit, and put it into the 1s column.
- We then divide the integer part of the previous result by 16. We take the remainder, convert it into a hex digit, and put it into the 16s column.
- We then divide the integer part of the previous result by 16. We take the remainder, convert it into a hex digit, and put it into the 256s column.
- We continue in this way until we reach a time when the integer part of the previous division was zero.

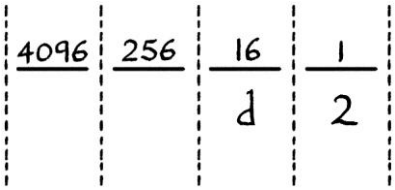
As an example, we will convert the decimal number 1234 to hexadecimal. We start with our empty hexadecimal number columns:



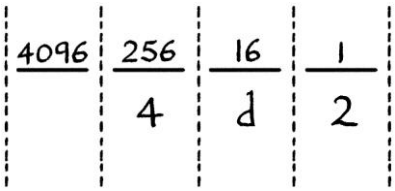
We divide 1234 by 16, and we get 77.125. This is 77 with a remainder of 2. [To calculate the remainder, we multiply the part after the decimal point by 16]. We convert the remainder to hex – it is 0x2 – and put it into the 1s column:



Then, we divide the integer part of the result from the last division (77 in decimal) by 16. This gives us 4.8125, which is 4 with a remainder of 13. We convert the remainder to a hex digit – it is 0xd – and put it in the 16s column:



Then we divide the integer part of the result from the last division (4) by 16. This gives us 0.25, which is 0 with a remainder of 4. We convert the 4 to a hex digit – it stays as 4 – and put it in the 256s column:



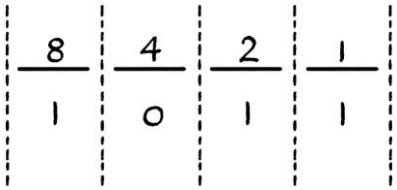
As the integer part of the result of the previous division was zero, we have finished. Our converted number is 0x4d2.

Converting between binary and decimal

Converting between *binary* and decimal uses a similar method as that for converting between hex and decimal. To convert from binary to decimal, we imagine the number columns for binary, which are, from right to left, ones, twos, fours, eights, sixteens, 32s, 64s, 128s and so on.

We go through the binary number, and add up all the number column values for whenever the bit is 1. [For example, if the number column is eights and the bit in that column is 1, then we add eight to our total.] The final total will be the binary number in decimal.

As an example, we will convert the binary number 1011 to decimal. [This is actually a number that, with practice, you will be able to do in your head instantly.] The number shown with the binary number columns is as so:

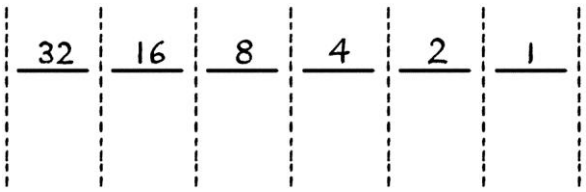


We will arbitrarily start at the leftmost bit, but it does not matter where we start. The leftmost bit is 1, which is in the eights column. Therefore, our running total will start with 8 in it. The next 1 is in the twos column. Therefore, we add 2 to our running total. The next 1 is in the ones column. Therefore, we add 1 to our running total. Our full sum is 8 + 2 + 1 = 11 in decimal.

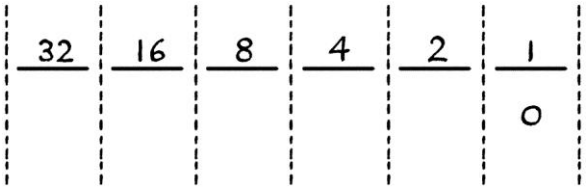
To convert from decimal to binary, we use a similar method to the one for converting from decimal to hex. If we have a decimal number to convert, we proceed as follows:

- We divide the number by 2. We take the remainder, which will be 0 or 1, and put it into the ones column.
- We then divide the integer part of the previous result by 2. We take the remainder, which again, will be 0 or 1, and put it into the twos column.
- We then divide the integer part of the last result by 2. We take the remainder, which again, will be either be 0 or 1, and put it into the fours column.
- We continue in this way until we reach a time when the integer part of the previous division was zero.

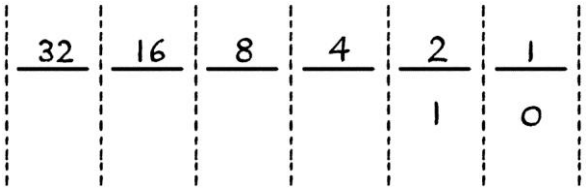
As an example, we will convert the decimal number 26 to binary. We will start with the blank number columns, which for this example, we will extend up to the 32s column. [We will only know how many columns we will need after we have performed the conversion.]



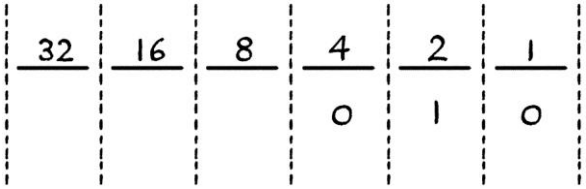
We divide our number by 2, and we end up with 13 and no remainder. Another way of saying this is that the remainder is 0. We put the remainder in the ones column:



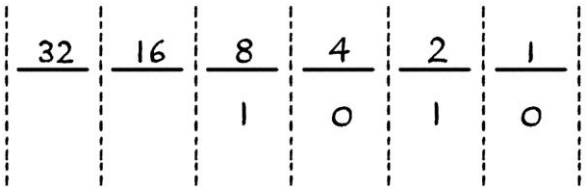
We then divide 13 by 2, and end up with 6.5, which is the same as 6 and a remainder of 1. We put the 1 in the twos column:



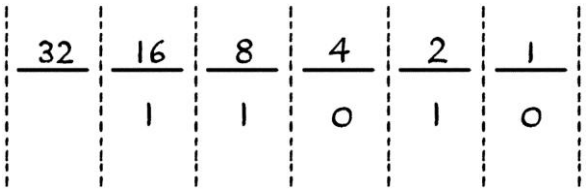
We then divide 6 by 2 and get 3 with a remainder of 0. We put the 0 in the fours column.



We then divide 3 by 2 and get 1.5, which is 1 with a remainder of 1. We put the one in the eights column.



We then divide 1 by 2, and we get 0 with a remainder of 1. We put the 1 in the sixteens column:



As our integer result was zero, we can stop here. Our final binary number (remembering to read from the sixteens column to the right) is 11010. In this example, we did not need to use the 32s column (or the 64s column, the 128s column and so on). If we wished, we could prefix our resulting binary number with zeroes to make it into a byte (00011010), word (0000000000011010), dword or qword. Alternatively, we can choose to leave it as it is.

When converting long binary numbers to decimal, it is easiest to convert the number to hex first, then and convert that to decimal. Conversely, if it looks like a conversion from decimal to binary will produce a long binary number, it is easier to convert from decimal to hex, and then from hex to binary.

Converting between hex and binary

Converting from hexadecimal to binary and back is easy because we know that 4 binary digits will end up as 1 hex digit, and that 1 hex digit will end up as 4 binary digits. A sequence of binary digits can be split up into fours (starting from the right-hand side), and each group of four can be converted into a hex digit. [If a binary number has too few digits to be split into groups of 4, we can prefix it with one or more zeroes to make it the correct length.] Similarly, a sequence of hex digits can be converted digit by digit into four binary bits.

Given all of that, if you want to become proficient at converting between hexadecimal and binary, it pays to learn the following table. The table is easiest to learn with practice – if you frequently need to convert between binary and hexadecimal, you will remember it quickly. If you seldom need to convert, it will be harder to learn, but, on the other hand, there will be less reason to learn it.

Binary	**Hexadecimal equivalent**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

If we have the binary number 1011110000000001, we first separate it into 4-bit sections to make it easier to read:
1011 1100 0000 0001
... then, we convert each 4-bit section to the single hex digit it represents:
b c 0 1
... which means that our hex number is:
0xbc01

If we have the hex number 0xfc23, we convert each digit to the 4-bit binary number it represents:
1111 1100 0010 0011
... and then we remove the spaces (which should only really exist to make the number easier to read):
1111110000100011

Groups of bits

As we saw earlier in this chapter, when dealing in binary, there are predefined groups of bits. These are half bytes, bytes, words, dwords and qwords. These groups also apply to hexadecimal, but hexadecimal groups are easier to read.

A half byte, being 4 bits, is represented by just one hexadecimal digit. For example, the binary number 1101 is 0xb in hexadecimal.

A byte, being 8 bits, is represented by two hexadecimal digits. Despite this, it will still be referred to as an 8-bit number. As it will always be two hexadecimal digits, if the number is less than 16 (in decimal), it will have a preceding zero in front. Examples of hexadecimal bytes are:

```
0x00
0x01
0x02
0x03
0x04
0x05
0x0a
0x0b
0x0c
0x0d
0x0e
0x0f
0x10
0x11
0x21
0xa0
0xb7
0xdd
0xfe
0xff
```

If we have the hexadecimal *byte* 0xff, and add 1 to it, we will end up with the byte 0x00. This is because 0xff is the highest possible number in a byte. It is 255 in decimal or 11111111 in binary. Adding 1 to 0xff makes it roll over to zero. If we added 2 to the byte 0xff, we would end up with the byte 0x01. If we added 3 to the byte 0xff, we would end up with the byte 0x02. [Remember that the rolling over at 0xff only occurs when we are using *bytes* to store a number. For other number groupings, or away from computers, adding 1 to 0xff would result in 0x100.]

As we know, a word is 16 bits long. This means that it has 16 binary digits if we are portraying it in binary. A word, written as hexadecimal, has 4 hexadecimal digits. Despite this, it will still be referred to as a 16-bit number. Some examples of words in hexadecimal are as follows:

```
0x0000
0x0001
0x0002
0x000f
0x0010
0x0011
0x0012
0x056a
0x1679
0x2acd
0x9abb
0xde01
0xdfff
0xe000
0xfffe
0xffff
```

If we are working with words, and we added 1 to 0xffff, we would end up with 0x0000. The number 0xffff is the highest number that we can represent with a word (16 bits).

If we had a word in binary as 1111000001100110, we could write it in hexadecimal as 0xf066.

A doubleword, or dword, is 32 bits long. When we write a dword as binary, we have 32 binary digits in the number. When we write it in hexadecimal, we have 8 hexadecimal digits. Despite this, it would still be referred to as a 32-bit number. Examples of hex dwords are as follows:

```
0x00000000
0x00000001
0x0000000f
0x00000010
0x0000ffff
0x12345678
0xe0000000
0xefffffff
0xf0000000
0xfffffffd
```

A quadword, or qword, is 64 bits long. In binary, such a number would require 64 digits. In hexadecimal, the same number would require 16 hexadecimal digits. Despite this, it would still be referred to as a 64-bit number.

Whereas a dword in hexadecimal is reasonably easy to read and remember, a qword is an awkward size. Therefore, for the purposes of this explanation, I will put a space in the middle of the number. You can think of this as similar to putting a comma in a long decimal number such as 1,000,000. If we were writing a qword in a computer program, we would not be able to put a space in the middle.

Some examples of qwords in hexadecimal, with a space in the middle for clarity, are as follows:

```
0x00000000 00000000
0x00000000 00000001
0x00000000 00000002
0x00000000 00000003
0x00000000 00000004
0x00000000 00000005
0x00000000 0000000f
0x00000000 00000010
0x00000000 0000ffff
0x12345678 9abcdef0
0x2fffffff ffffffff
0x30000000 00000000
0xcccccccc cccccccc
0xf0000000 00000000
0xfffffffd fffffffd
0xfffffffe fffffffe
0xffffffff ffffffff
```

The last number in the list is the highest number that we can represent with a qword.

More on hexadecimal

Hexadecimal should really be thought of as another way of writing binary. It pays to think of binary and hexadecimal as being interchangeable, or if they were really the same thing in different forms. A number, whether written as binary or hexadecimal, is still the same number. Although computers ultimately work in binary, it is easier to think of them as working in hexadecimal. Hexadecimal is a way of making endless sequences of binary digits have a recognisable meaning. For example, it is easier to think of this 32-bit binary number:
11110001011101110110000100100011
... as being:
0xf1776123
... than it is to think of it as binary. It is much easier to read and remember hexadecimal numbers, and it is much easier to recognise patterns in hexadecimal numbers. Hexadecimal's close connection with binary is the reason that it is usually easier not to convert hexadecimal to binary, but stay in the world of hexadecimal. If we know a hexadecimal number, we know the exact position of every bit in its binary equivalent.

Hexadecimal also has a slight advantage over decimal in that it is better for larger numbers. The most extreme example is how the maximum qword value (0xffffffffffffffff) takes 16 hexadecimal digits, but its decimal equivalent is 20 digits long.

Hex editors

On a computer, every file, whether a text file, an executable, a picture, an MP3, a Microsoft Word document, or anything, is really a sequence of binary digits. When an MP3 file, for example, is opened by an MP3 player program, the sequence of bits in the file is interpreted as music. If a text file is opened by a text editor, the sequence of bits in the file is interpreted as text. When an executable is run, the operating system interprets the sequence of bits as instructions to be executed. If you open an executable file in a text editor, you will see seemingly random characters. If you try to play a text file in an MP3 player, the player will complain that it does not understand the file. To keep things straightforward, programs generally hide the underlying binary content of files from users.

It is possible to see the hexadecimal equivalent of the binary bits that make up a file by opening the file in a hex editor. A hex editor is a program that opens any file, without interpreting its contents in any way. When you open a file in a hex editor, you will see the contents of a file as a sequence of hexadecimal bytes. [It would be possible to have a program that showed the binary bits that made up a file, but it would be much harder to make sense of the data in that way. It is better to view the binary in the form of hexadecimal.] Using a hex editor is the best way to become accustomed to how computers store data. It is worth downloading a free hex editor, so you can improve your understanding of hexadecimal and files in general.

As an example of what we might see in a hex editor, here are the first 64 bytes from a text file containing this very sentence:

```
41 73 20 61 6e 20 65 78 61 6d 70 6c 65 20 6f 66
20 77 68 61 74 20 77 65 20 6d 69 67 68 74 20 73
65 65 20 69 6e 20 61 20 68 65 78 20 65 64 69 74
6f 72 2c 20 68 65 72 65 20 61 72 65 20 74 68 65
```

Because we know that these bytes are hexadecimal, there is no need to prefix them all with “0x”, and doing so would make them harder to read.

Due to the size of a typical file and the width of a typical computer screen, only so many bytes will fit neatly on a line. The start and end points of a line of bytes is solely due to how a particular hex editor has decided to fit the bytes on a line, and is not related to the nature of the bytes.

The grouping into bytes is just to make the digits easier to read. The digits could just as easily be written as one continuous block as so:

```
417320616e206578616d706c65206f662077686174207765206d696768742073656520
696e20612068657820656469746f722c20686572652061726520746865
```

Although computers work with binary, a hex editor shows the data as hexadecimal to make it easier to read and interpret. If it were to show the data in binary, we might expect to see it as so:

```
00100001 01110011 00100000 01100001 01101110 00100000 01100101
01111000
```

... and so on.

Or, if there were no arbitrary spaces between the bits, we would see it as so:

```
00100001011100110010000001100001011011100010000001100101011110000
```

... and so on.

It is much easier to read the data as hexadecimal.

Hex editors usually show where in the file each line of hex bytes starts by prefixing each line with the "offset" of the line. The "offset" is the position of a particular byte with respect to the start of the file. Our 64 bytes of text would look like this with the starts of each line written to the left of the data:

```
0000: 41 73 20 61 6e 20 65 78 61 6d 70 6c 65 20 6f 66
0010: 20 77 68 61 74 20 77 65 20 6d 69 67 68 74 20 73
0020: 65 65 20 69 6e 20 61 20 68 65 78 20 65 64 69 74
0030: 6f 72 2c 20 68 65 72 65 20 61 72 65 20 74 68 65
```

The offsets themselves are given in hexadecimal. The file starts at byte number 0x0000. To put this another way, the first byte is at offset 0x0000 in the file. The first line is sixteen (in decimal) bytes long, which is 0x10 (in hex) bytes long. This means that the first byte of the *next* line is at offset 0x0010 in the file. Byte number sixteen (in decimal) in the file has the value 0x20. The second line is 0x10 bytes long. Therefore, the third line as seen in the hex editor's display shows data from byte number 0x30 onwards in the file. Different hex editors vary in how many bytes they show on a line, but they will usually show the offset of where the bytes are. In the example above, I have given the offset as 4 hex digits to save space on the page. Most hex editors will give the offset as 8 hex digits so that they can work with very large files.

To make recognising patterns in hexadecimal easier, hex editors usually have a section to the right of each line of hex that shows if any of the bytes would be valid ASCII characters, and if so what they would be. [I will explain ASCII characters shortly, but for now, it is enough to know that ASCII is a generally accepted way of encoding letters of the alphabet with 8-bit numbers (bytes).] The bytes from our text file would look like this in a hex editor that showed the ASCII part:

```
0000: 41 73 20 61 6e 20 65 78 61 6d 70 6c 65 20 6f 66 As an example of
0010: 20 77 68 61 74 20 77 65 20 6d 69 67 68 74 20 73  what we might s
0020: 65 65 20 69 6e 20 61 20 68 65 78 20 65 64 69 74 ee in a hex edit
0030: 6f 72 2c 20 68 65 72 65 20 61 72 65 20 74 68 65 or, here are the
```

As the data in our example is all text, every byte represents a letter of the alphabet, so the whole of the right hand column is text. The very first byte of our file is 0x41, which is also the ASCII number for the letter “A”. The next byte is 0x73, which is the ASCII letter “s”. The next byte is 0x20, which is the ASCII number for a space.

ASCII

The abbreviation “ASCII” is short for the “American Standard Code for Information Interchange”. ASCII is a system that assigns a number to each letter of the alphabet, to each decimal digit, and to some punctuation. ASCII is a generally accepted standard for encoding basic text characters. By following the ASCII system of assigning particular numbers to characters, different computer programs can store and read text in the same way. It makes everything much easier to have a standard system. It is worth noting that the ASCII method of assigning numbers to characters is completely arbitrary, and its creation and its use is independent of how the processors of computers work, except for how each ASCII value is stored as one byte.

Having a general understanding of ASCII is useful in understanding what you might see in a hex editor.

In the ASCII system, each character is identified by 8 bits. Therefore, each character can fit in exactly one byte. Whether we choose to treat the value in that byte as binary, hexadecimal or decimal is a matter of choice. In some situations, it is easier to think of them as binary; in some situations, particularly when viewing them in a hex editor, it is easier to think of them as hexadecimal. Some people prefer to think of them as decimal numbers, but doing that is generally less useful.

Being an American system, ASCII prioritises characters from a subset of the modern English alphabet. As well as letters, numbers and punctuation, there are some obsolete “control” characters, which were used to control early printers among other things.

Note that a byte only represents an ASCII character if the computer program that is dealing with it chooses to interpret it as one. If a program is not specifically working with ASCII text (or it does not know that it should be), a byte will just be treated as a number.

The first 128 characters of ASCII, being the characters from 0x00 to 0x7f, are as follows. I have ignored the meanings of most of the control symbols:

ASCII number as binary	ASCII number as hex	Character or meaning
00000000	0x00	This is often used, arbitrarily, in programming languages to mark the end of a piece of text.
00000001	0x01	
00000010	0x02	
00000011	0x03	
00000100	0x04	
00000101	0x05	
00000110	0x06	
00000111	0x07	On older computers, reading this character would make the computer beep.
00001000	0x08	Backspace
00001001	0x09	Tab
00001010	0x0a	In Linux, this is treated as a new line character. It means any text starts on a new line after this appears. In Windows, this is treated as a new line character when it is preceded by 0x0d.
00001011	0x0b	
00001100	0x0c	In some text editors, this indicates a new page.
00001101	0x0d	In Windows, 0x0d followed by 0x0a is treated as a new line.
00001110	0x0e	
00001111	0x0f	
00010000	0x10	
00010001	0x11	
00010010	0x12	
00010011	0x13	
00010100	0x14	
00010101	0x15	
00010110	0x16	
00010111	0x17	
00011000	0x18	
00011001	0x19	
00011010	0x1a	
00011011	0x1b	

ASCII number as binary	ASCII number as hex	Character or meaning
00011100	0x1c	
00011101	0x1d	
00011110	0x1e	
00011111	0x1f	
00100000	0x20	Space: “ ”
00100001	0x21	!
00100010	0x22	"
00100011	0x23	#
00100100	0x24	$
00100101	0x25	%
00100110	0x26	&
00100111	0x27	'
00101000	0x28	(
00101001	0x29	)
00101010	0x2a	*
00101011	0x2b	+
00101100	0x2c	,
00101101	0x2d	-
00101110	0x2e	.
00101111	0x2f	/
00110000	0x30	0
00110001	0x31	1
00110010	0x32	2
00110011	0x33	3
00110100	0x34	4
00110101	0x35	5
00110110	0x36	6
00110111	0x37	7
00111000	0x38	8
00111001	0x39	9

[To convert the ASCII code of a number to the actual number that it represents, we can just subtract 0x30.]

ASCII number as binary	ASCII number as hex	Character or meaning
00111010	0x3a	:
00111011	0x3b	;
00111100	0x3c	<
00111101	0x3d	=
00111110	0x3e	>
00111111	0x3f	?
01000000	0x40	@
01000001	0x41	A
01000010	0x42	B
01000011	0x43	C
01000100	0x44	D
01000101	0x45	E
01000110	0x46	F
01000111	0x47	G
01001000	0x48	H
01001001	0x49	I
01001010	0x4a	J
01001011	0x4b	K
01001100	0x4c	L
01001101	0x4d	M
01001110	0x4e	N
01001111	0x4f	O
01010000	0x50	P
01010001	0x51	Q
01010010	0x52	R
01010011	0x53	S
01010100	0x54	T
01010101	0x55	U
01010110	0x56	V
01010111	0x57	W
01011000	0x58	X
01011001	0x59	Y
01011010	0x5a	Z

ASCII number as binary	ASCII number as hex	Character or meaning
01011011	0x5b	[
01011100	0x5c	\
01011101	0x5d	]
01011110	0x5e	^
01011111	0x5f	_
01100000	0x60	`
01100001	0x61	a
01100010	0x62	b
01100011	0x63	c
01100100	0x64	d
01100101	0x65	e
01100110	0x66	f
01100111	0x67	g
01101000	0x68	h
01101001	0x69	i
01101010	0x6a	j
01101011	0x6b	k
01101100	0x6c	l
01101101	0x6d	m
01101110	0x6e	n
01101111	0x6f	o
01110000	0x70	p
01110001	0x71	q
01110010	0x72	r
01110011	0x73	s
01110100	0x74	t
01110101	0x75	u
01110110	0x76	v
01110111	0x77	w
01111000	0x78	x
01111001	0x79	y
01111010	0x7a	z

[To convert a lower-case letter to upper case, we can just subtract 0x20 or set the third bit from the left to zero.]

ASCII number as binary	ASCII number as hex	Character or meaning
01111011	0x7b	{
01111100	0x7c	|
01111101	0x7d	}
01111110	0x7e	~
01111111	0x7f	

The ASCII characters from 0x80 to 0xff are called the "Extended ASCII" characters because they are an extension to the original ASCII system, which used only 7 of the available 8 bits of a byte. There are numerous variations for the meanings of the Extended ASCII bytes, depending on the encoding and language being used. These variations are called "code pages". The most commonly used of these were defined by Microsoft, and have the names "Windows-1250", "Windows-1251", "Windows-1252" and so on. Among these are the following:

- "Windows-1250" uses the bytes to represent symbols and accented letters from Eastern and Central European alphabets that are based on the Latin alphabet (such as those used with Polish and Czech).
- "Windows-1251" uses the bytes to represent symbols and letters from a subset of the Cyrillic alphabet (as used in Russian and Ukrainian).
- "Windows-1252" uses the bytes to represent symbols and accented letters from Western European alphabets that are based on the Latin alphabet (such as French).
- "Windows-1254" is for accented Turkish letters.
- "Windows-1255" is for Hebrew letters.
- "Windows-1256" is for Arabic letters.

In each system, the bytes from 0x00 to 0x7f still have the same meaning as in the long list from before.

The different interpretations of the bytes from 0x80 onwards mean that text that is supposed to be from one particular language will be displayed incorrectly if the wrong interpretation is used. If someone writes Russian text in an ASCII text file on a Russian version of Microsoft Windows, that file will appear as random accented Latin letters and symbols on an English version of Microsoft Windows. It will appear as random Arabic letters and symbols in an Arabic version of Windows. This is one of the big flaws of ASCII – it is not particularly good for non-English alphabets.

Now that memory is cheaper, and computers and networks are faster, ASCII has generally been replaced with Unicode. Unicode is a similar system, which, depending on its implementation, can use one, two, or more bytes to represent a character. Alphabets that have many characters might use three bytes or more for each character. The "0x00 to 0x7f" ASCII set of Latin characters is portrayed with one byte in the Unicode version called UTF-8, or two bytes in the Unicode version called UTF-16. As there is no limit to the number of bytes that can be used per character, each Unicode value represents a unique character or symbol, and there can never be any confusion between languages.

More hex editor examples

Here is the very start of a Microsoft Windows executable (that is to say, a program or application) as opened with a hex editor:

```
0000: 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0010: b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0030: 00 00 00 00 00 00 00 00 00 00 00 00 c8 00 00 00 ................
0040: 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
0050: 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is program canno
0060: 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS
```

The first two bytes are 0x4d, 0x5a, which are the ASCII letters "MZ". This is what is called the "signature" of a file. It gives clues to the operating system, or any program trying to open it, as to what sort of file this is. Generally, Microsoft Windows executables will start with these letters or the operating system will complain. The dots on the right hand side indicate that if the associated byte on the left hand side were interpreted as ASCII, it would not be a letter, number or punctuation. Due to the ambiguity of ASCII characters over 0x80, most hex editors do not give equivalent letters for those bytes, and just have a "." instead. Therefore, if there were Cyrillic ASCII text in a file, we would have to recognise it by the bytes themselves, and the hex editor would not help.

The "@" symbol, corresponding to the byte 0x40 is a coincidence. Hex editors cannot know if a value is intended to be a letter, number or punctuation, so they presume, usually incorrectly, that anything that could be, will be.

The part of the file that we see above is part of what is called the “header”. The header of a file is information that indicates to the operating system, or the program that opens it, certain attributes of the data within the file. For example, the full header for an executable will list where in the file the commands are, where the data is, where any resources such as icons and dialog layouts are stored, and so on. Not all files have headers – for example, a basic ASCII text file does not need one.

The following is what the start of a particular TIFF image file looks like when opened in a hex editor:

```
0000: 49 49 2a 00 08 00 00 00 0a 00 00 01 03 00 01 00 II*.............
0010: 00 00 00 08 00 00 01 01 03 00 01 00 00 00 00 02 ................
0020: 00 00 03 01 03 00 01 00 00 00 01 00 00 00 06 01 ................
0030: 03 00 01 00 00 00 01 00 00 00 11 01 04 00 01 00 ................
0040: 00 00 96 00 00 00 16 01 04 00 01 00 00 00 00 02 ................
0050: 00 00 17 01 04 00 01 00 00 00 00 00 02 00 1a 01 ................
0060: 05 00 01 00 00 00 86 00 00 00 1b 01 05 00 01 00 ................
0070: 00 00 8e 00 00 00 28 01 03 00 01 00 00 00 03 00 ......(.........
0080: 00 00 00 00 00 00 40 00 00 00 01 00 00 00 40 00 ......@.......@.
0090: 00 00 01 00 00 00 ff ff ff ff ff ff ff ff ff ff ................
00a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
```

The first two bytes are 0x49, 0x49. These indicate to any program opening the TIFF file that, first, this is a TIFF file, and second, how the data within it is stored. The bytes 0x2a, 0x00 tell the program which version of the TIFF image format is being used. The following bytes up to the byte at offset 0x96 tell the program how many rows and columns of pixels there are in the TIFF file, whether the data is compressed, what the resolution is, and where the actual image data starts. This particular file is an uncompressed black and white bitmapped TIFF file, which means that it assigns one bit to every pixel. That bit will be either a 1 to indicate a white pixel, or a 0 to indicate a black pixel. The actual image data starts at offset 0x96. The data starts as a sequence of 0xff bytes, where each byte represents 8 consecutive white pixels. The “*” sign, the bracket and the “@” sign on the right-hand side are coincidences where a hexadecimal value happens also to be a readable ASCII character.

Memory

When programmers run programs they have written, it can be useful to know exactly what is going on in the computer, so they can find mistakes or improve their code. To do this they will use a program that steps through each command in turn, while showing what is happening inside the processor, and what is stored in the relevant part of memory. Such a program is called a "debugger" because it is used to debug software. When viewing what is stored in memory, a debugger will show a very similar view to what we might see in a hex editor. The main difference will be that the offsets will not generally start at zero, but instead at an offset from the start of the computer's memory being assigned to the program. For a Windows program intended for a 32-bit Intel processor, we might see something such as this:

```
040ca100: c3 eb 12 90 90 90 90 00 48 33 C0 48 8B 74 24 40 ........H3.H.t$@
```

In this example, the ASCII characters on the right-hand side are just coincidences where the bytes happen to have values equal to valid ASCII characters.

Other information

Here is some more information about hexadecimal and binary.

Bit number

When identifying entries in a list using binary or hexadecimal, it makes sense to treat the first entry as item zero, instead of item one. This is because, by doing so, we are able to count higher. For example, a list of names might start as so:

Name number 0: Gertrude
Name number 1: Gladys
Name number 2: Gwen
Name number 3: Gwynne
... and so on.

If we are constrained by using bytes, we can distinguish 256 different names if we start at zero. We would have name number 0x00 up to name number 0xff. If we start at 1, we can only distinguish 255 different names – we would have name number 0x01 up to name number 0xff.

This way of counting from zero is also used when identifying a particular bit in a binary number. When identifying the bits of a binary number, we count from the right hand side. Bit 0 is the rightmost bit. With a 8-bit byte, bit 7 is the leftmost bit. With a 16-bit word, bit 15 is the leftmost bit. With a 32-bit dword, bit 31 is the leftmost bit. With a 64-bit qword, bit 63 is the leftmost bit.

In the following binary number as a byte:
00000001
... bit 0 has the value “1”. All the other bits have the value “0”. Bit 7 (the leftmost bit) is “0”.

With this binary number:
10000000
... bit 0 (the rightmost bit) has the value “0”, and bit 7 (the leftmost bit) has the value “1”.

With the binary number:
01001110

- bit 7 is 0
- bit 6 is 1
- bit 5 is 0
- bit 4 is 0
- bit 3 is 1
- bit 2 is 1
- bit 1 is 1
- bit 0 is 0

As well as identifying bits in binary numbers, we can take advantage of how there is a one-to-one relationship between binary and hexadecimal, and identify the bits within hexadecimal numbers. For example, in the hex byte 0xf1:

- bit 7 is 1
- bit 6 is 1
- bit 5 is 1
- bit 4 is 1
- bit 3 is 0
- bit 2 is 0
- bit 1 is 0
- bit 0 is 1

This is all because 0xf1 is 11110001 in binary. Despite speaking about a hexadecimal number and not a binary number, we can still treat it as consisting of bits.

Significant bits

When dealing in binary and hexadecimal, it is common to see the phrases “most significant bit” and “least significant bit”.

The most significant bit is the leftmost bit of a byte, word, dword or qword. It is called this because the leftmost bit is the bit whose presence or absence has the greatest effect on the overall *size* of a number. For example, the leftmost bit in the binary number 10000001 makes the difference between the number being 0x81 or 0x01. This is a difference of 128 in decimal. The term “most significant bit” is often abbreviated to “msb”.

The least significant bit is the rightmost bit. This is because the presence of the rightmost bit has the least effect on the overall *size* of a number. The rightmost bit of the binary number 11111111 is the difference between the number being 0xff and 0xfe (255 and 254 in decimal). There is only a difference of 1. The term “least significant bit” is often abbreviated to “lsb”.

It is obviously quicker and more descriptive to say “leftmost bit” and “rightmost bit”, but some people prefer the terms “most significant” and “least significant”. One problem with the term “significant” is that it only makes sense if the byte, word, dword or qword is being used to count. In an ASCII byte, for example, every bit is just as significant as any other because the byte is not being used as a number, but as an index to an item in a table of characters.

Storing data in memory or files

We will imagine a computer processor is dealing with the 32-bit dword, 0x87654321. When it has finished working with the dword and it wants to store it for later, perhaps in memory or to disk, there are two ways that it can lay down the number.

The first way is for the processor to place the number down in the order of the digits. This is the most intuitive way, and is how most people would expect it to do it. This means that if we were looking at a file in a hex editor, with the dword placed at the very start, we would see this:

```
0000: 87 65 43 21 00 00 00 00 00 00 00 00 00 00 00 00 .eC!............
```

[Note that each 0x00 is just filler that I have put in to make up a full line. The letters “eC!” are there because, by chance, the bytes 0x65, 0x43, and 0x21 are also valid readable ASCII characters.]

If the computer processor then wanted to read the dword from the file or memory, it would read the whole dword in one go. The reason it would work in this way hardly needs explaining, as it is exactly what one would expect it to do.

The second way a computer processor might place a dword into memory or to a file is by splitting the dword into bytes and putting them into *reverse* order. As we are starting with the dword 0x87654321, this means that the dword becomes the bytes 0x87, 0x65, 0x43, 0x21, and each byte is put down in reverse order. A file containing the result of such a thing as the very first few bytes would look like this:

```
0000: 21 43 65 87 00 00 00 00 00 00 00 00 00 00 00 00 !Ce............
```

[Again, the zeroes are filler to make up a full line, and the letters “!Ce” are there because the bytes are, by coincidence, valid readable ASCII characters.]

Although this method seems to overly complicate matters, it can be useful. If we want to find the first byte of the stored dword 0x87654321, we look at the offset in the file of where we put the whole dword. The first byte of 0x87654321 is 0x21, and it is at offset 0x0000 in the file. Similarly, if we wanted to find the first *word* of 0x87654321, we would look at offset 0x0000 in the file, and read the two bytes there (0x21 and 0x43), which because we are using this backwards ordering system, we rearrange to be 0x4321. Therefore, this backwards method allows us easily to retrieve the lower bytes of words, dwords and qwords without having to know whether we are using words, dwords and qwords. It is a useful method, but it is mostly hidden from anyone who is not programming in assembly language, examining files with hex editors, or debugging software.

As with a dword, if a computer processor were working in this system and storing a qword or a word in memory or a file, it would similarly put its bytes in reverse order. If the processor were storing a byte, then the byte would be the same within the computer processor as it would be stored in a file or memory.

If the processor stored the qword, 0x8070605040302010 into memory or a file, it would appear as so:

```
0000: 10 20 30 40 50 60 70 80 00 00 00 00 00 00 00 00 ."3DUfw.........
```

[Note that the offset would vary depending on where in the file or memory the number was placed.]

If the processor stored the dword, 0x40302010 it would appear as so:

```
0000: 10 20 30 40 00 00 00 00 00 00 00 00 00 00 00 00 ."3D............
```

If the processor stored the word, 0x2010 it would appear as so:

```
0000: 10 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ."..............
```

If the processor stored the byte, 0x10 it would appear as so:

```
0000: 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

If we wanted to store a half byte (4 bits), we would first have to turn it into a whole byte, perhaps with the first 4 bits set to zero. The result of storing it would be the same as if we were storing a byte.

This “backwards” method of storing data is called the “little-endian” method of storing data. The “lowest” bytes are placed first, which we can rephrase as the “littlest end of the word, dword or qword is placed first”. Given that Intel processors use this method, at the time of writing, this might be as common as the ostensibly more sensible “forwards” method, which is called “big-endian”. We can think of big-endian as placing the “highest” or “biggest” bytes first. Although little-endian seems unintuitive, the more you are exposed to it, the less puzzling it will seem.

When something is stored in *memory*, it is stored using the system that the processor uses. Therefore, (at the time this is being written) an Intel or AMD processor will always use the “backwards” little-endian system when putting a number in memory. When something is stored in a *file*, the system that is used depends on the program that is storing the data. However, it is much easier and quicker to store data in a way that is consistent with the processor. Doing so means that something can be copied directly from memory into a file, without needing rearranging. Other processors might use the “forwards” big-endian system, and some processors are able to use both systems.

In a TIFF image file, the first two bytes of the header tell the image-viewing software whether the data has been stored with the big-endian or little-endian system. If the first two bytes are the ASCII letters “II”, then the data in the file is little-endian; if the bytes are “MM”, the data is big-endian. This allows TIFF files to be viewed on computers that operate in either the big-endian system or the little-endian system, without reading the data the wrong way around. If a program is going to be reading data in group sizes larger than bytes, it needs to have a way of knowing which system is being used, or it might misinterpret the data.

If we have several numbers in sequence, then we store them all in either the little-endian way or the big-endian way. As an example, we will look at this list of dwords:

```
0x0000ffff
0x11223344
0x55667788
0xfefeffff
```

Stored in the little-endian way, they would look like this in a file viewed in a hex editor:

```
0000: ff ff 00 00 44 33 22 11 88 77 66 55 ff ff fe fe ....D3"..wfU....
```

Stored in the big-endian way, they would look like this:

```
0000: 00 00 ff ff 11 22 33 44 55 66 77 88 fe fe ff ff .....”3DUfw.....
```

When it comes to storing ASCII text, each letter is represented by a single byte, so the bytes are laid down in their original order.

```
0000: 48 65 72 65 27 73 20 73 6f 6d 65 20 74 65 78 74 Here's some text
```

When it comes to storing Unicode text, and when the characters are being portrayed by 16-bit words each, the 16-bit words are stored in either the little-endian way or the big-endian way.

Octal

Octal is a counting system that is based around the number 8. If we were to count in octal, we would proceed as follows:

0
1
2
3
4
5
6
7
10
11
12
13
14
15
16
17
20
21
22
23
24
25
26
27
30

... and so on.

For some reason, octal is often taught as if it were as useful to computers as hexadecimal or binary. Computers do not natively use octal, and it is much easier to be using hexadecimal, binary and decimal. I only mention octal here in case you should see it mentioned elsewhere.

Negative numbers

Away from a computer, we can portray negative numbers and fractions in binary and hexadecimal in the same way that we do with decimal numbers. For example, the number −12 in decimal could be written as −1100 in binary or −c [or −0xc] in hexadecimal. The decimal number 2.5 could be written as 10.1 in binary or 2.8 in hexadecimal. The number −1011.0011001 is a perfectly valid way to write a number in binary. When it comes to computers, we cannot write numbers in this way because the default system of counting in binary or hexadecimal for computers only allows positive integers. There is no place for a decimal point or a negative sign in one of the group types for bits.

To allow computers to work with negative numbers and fractions, slightly awkward workarounds were invented. These involve *interpreting* seemingly normal hexadecimal or binary values in a different way. In other words, we still use bytes, words, dwords and qwords, and these still contain binary bits. However, we alter the bits of the number in a special way before we put the number into a byte, word, dword or qword. Someone reading the byte, word, dword or qword must know that they need to interpret it differently, or else the value will be misinterpreted as a positive integer.

In this section, we will look at how computers manage negative numbers.

Two's complement

The standard way that computers express negative numbers in binary or hexadecimal is called "two's complement". The general idea is that the leftmost bit (the highest bit) is set to 0 if the number is positive, and it is set to 1 if the rest of the number is negative. This means that we have fewer bits with which to portray values, but it allows us to have both positive and negative numbers – *as long as we and anyone else using the number know that it should be interpreted in this way*. The actual method is slightly more complicated than just setting the leftmost bit, as we will see shortly.

Two's complement allows one byte to portray the numbers from −0x80 to +0x7f (−128 to +127 in decimal). It allows a word to hold the numbers from −0x8000 to +0x7fff (−32,768 to +32,767 in decimal). It allows a dword to hold the numbers from −0x80000000 to +0x7fffffff (−2,147,483,648 to +2,147,483,647 in decimal). It allows a qword to hold values from −0x8000000000000000 to +0x7fffffffffffffff (−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 in decimal).

All of this means that if we only wanted to store positive integers, using two's complement would only allow us to store 1 less than half the highest value that we could store normally.

Trying to store a value outside of the available ranges will result in the number being stored incorrectly, and later, being interpreted incorrectly.

The way to format a number as a two's complement number is fairly easy. We start by making sure that our number is not too high or too low to fit in the byte, word, dword or qword that we are using.

If the number we have is negative, then:

- We make it positive.
- We subtract 1.
- We "flip" all the bits apart from the leftmost bit. In other words, if a bit is 0, we change it to 1; if a bit is 1, we change it to 0.
- We set the leftmost bit to 1 to indicate that it is a negative number.

If the number we have is positive, then:

- We set the leftmost bit to 0 to indicate that it is a positive number.
  [Strictly speaking, it should be 0 already, because if it is not, then the number is too large to be encoded as a two's complement number.]

When we have a negative number, it will always be the case that once we have subtracted 1, the leftmost bit will be zero. Therefore, we can speed up the method by flipping all of the bits including the leftmost bit in one go – we do not need to set the leftmost bit to 1 to indicate that it is a negative number because the flipping will do that for us. Therefore, a quicker method to store a negative number is:

- We make it positive.
- We subtract 1.
- We flip all the bits. In other words, if a bit is 0, we change it to 1; if a bit is 1, we change it to 0.

The "complement" of a bit is its opposite. The complement of 0 is 1, and the complement of 1 is 0. When flipping a bit from 0 to 1, or 1 to 0, we are finding its complement. For negative numbers, we find the complement of every bit. As we are using binary and complementing the binary digits, the system is called "two's complement".

When reading a converted value, as long as the processor or software (or a person reading it) is aware that the value is a two's complement number, it will be known what the value represents. On the other hand, if a negative two's complement number is read as a normal number, it will be misinterpreted as a different value. There is no way of knowing whether a given hex or binary value is intended to be a two's complement number or not by looking at it. It needs context to be decoded correctly. Two's complement is really just a slightly awkward compromise that reuses normal binary and hex in a different way.

Examples

We will look at some simple examples involving bytes. The first thing to realise is that any *positive* number under and including 0x7f will be the same whether it is in two's complement or not. A value higher than 0x7f cannot be converted to a two's complement byte, as the maximum possible positive value is 0x7f. The byte 0x7f is 01111111 in binary. Any number higher than this would have a 1 as its leftmost digit. The number 0x23 as a two's complement byte will still be 0x23. The number 0x70 will still be 0x70 as a two's complement byte. The number 0x00 will still be 0x00.

If we want to express −1 as a negative two's complement byte, we first make it positive. It becomes:
00000001

We subtract 1 to end up with:
00000000

We then flip all the bits, which also has the effect of setting the leftmost bit to 1 to indicate that it is a negative number:
11111111

This is 0xff in hex, so we can say that −1 as a two's complement byte is 0xff.

A useful pattern to recognise is that, *in two's complement*, −1 is 0xff as a byte, 0xffff as a word, 0xffffffff as a dword, and 0xffffffffffffffff as a qword. It pays to remember that if we were not interpreting the bytes as two's complement numbers, the values would have their normal hexadecimal meanings, with 0xff being 255 in decimal, 0xffff being 65,535, and so on. There is no way of knowing by looking at a number whether it is intended to be a two's complement number or not – we have to know the context in which it is being used.

If we want −2 as a two's complement byte, we first make it positive. It becomes:
00000010

We subtract 1 to end up with:
00000001

We then flip all the bits:
11111110

This is 0xfe in hex. The number −2, *when using two's complement*, is 0xfe as a byte, 0xfffe as a word, 0xfffffffe as a dword, and 0xfffffffffffffffe as a qword. If we were not using two's complement, then we could not express −2 in hex on a computer, and the values 0xfe, 0xfffe, 0xfffffffe and 0xfffffffffffffffe would have their normal meanings.

Now, we will encode the largest possible negative number for a byte, which is −128. To do this, we first make it positive, so it becomes +128 in decimal, which, in binary is:
1000000

We might think this number is too large to be encoded because it uses the leftmost bit. However, the next step is to subtract 1, which results in:
01111111

Now the number does not use the leftmost bit. This is how the negative numbers for a byte can reach down to −128, but the positive values can only reach up to +127.

We then flip the bits:
10000000

This is 0x80, so we can say that the decimal number −128 as a two's complement number is 0x80.

Thoughts

Any hexadecimal two's complement number that starts with 8, 9, a, b, c, d, e or f must be a negative number – for it to start with one of those digits, it must have the leftmost bit set to 1:

8 is 1000
9 is 1001
a is 1010
b is 1011
c is 1100
d is 1101
e is 1110
f is 1111

Any two's complement number that starts with 0, 1, 2, 3, 4, 5, 6, or 7 must be a positive number – for it to start with one of those digits, it must have the leftmost bit set to 0:

0 is 0000
1 is 0001
2 is 0010
3 is 0011
4 is 0100
5 is 0101
6 is 0110
7 is 0111

From this we know that 0xa4, for example, must be a negative number and that 0x74, for example, must be a positive number *when those numbers are intended to be, and being interpreted as, two's complement numbers.* If the numbers are not intended to be two's complement numbers, or they are not being interpreted as two's complement numbers, then both 0xa4 and 0x74 will be positive numbers, as all non-two's complement numbers are.

More examples

Now we will look at −7 as a two's complement *dword*. First, we make it positive, and write it as a binary dword: [I have put in spaces to make it easier to read]
00000000 00000000 00000000 00000111

We subtract 1:
00000000 00000000 00000000 00000110

Then we flip the bits:
11111111 11111111 11111111 11111001

This is 0xfffffff9 in hex. Therefore, −7 in decimal is 0xfffffff9 in hexadecimal as a two's complement dword. [As a byte, it would be 0xf9. As a word, it would be 0xfff9.]

Another method

Another way to calculate a two's complement number is to take the negative number we want to express, make it positive, subtract 1, and then subtract the number from 0xff (if we are using bytes), from 0xffff (if we are using words), from 0xffffffff (if we are using dwords), or from 0xffffffffffffffff (if we are using qwords).

As an example, we will say that we want to portray the hex number −0x67 as a two's complement byte. We make it positive as +0x67, then subtract 1 to get 0x66, and then subtract that from 0xff. We end up with 0x99.

This method works because subtracting a binary number from a second binary number that is all 1s has the same effect as flipping the bits of the first number. For −0x67 as a byte, we would be calculating 0xff minus 0x66, which is:

11111111
... minus:
01100110
... which is:
10011001
... which is:
0x99 in hex.

Whether this method is better than the bit flipping method depends on what it is we are doing. If we have a hexadecimal calculator, and one that works only with positive numbers, this method might be quicker than the bit flipping method. [However, if we have a hexadecimal calculator, it is likely to be able to convert negative numbers into two's complement numbers anyway.]

Converting away from two's complement

Converting from a two's complement number to a normal number is easy. If the two's complement number is positive (it has the leftmost bit set to 0), then we leave it as it is. If the two's complement number is negative (it has the leftmost bit set to 1), we proceed as follows:

- We flip all the bits
- We add 1
- We put a minus sign in front of the number

As an example, we will convert 0xd5 to a non-two's complement number. As the hexadecimal digit "d" is 1101 in binary, we can tell that this is a negative number – the leftmost bit is 1. The number in full is 11010101. We flip all the bits to produce:
00101010
... then we add 1:
00101011
... then we put a minus sign in front of the number:
−00101011

This is the same as −0x2b in hexadecimal, which is −43 in decimal.

Why the system is how it is

If we are given a two's complement number *and we know that it is supposed to be a two's complement number*, we can tell if it is a negative number or not by whether the leftmost bit is set to 1 or not. A negative number always has the leftmost bit set to 1, and a positive number always has the leftmost bit set to 0.

When converting negative numbers to two's complement, we start by making the number positive and then subtracting 1. This is how the generally agreed system works. You might think that an easier system would keep the number the same and set the leftmost bit to 1 for a negative number, and set it to 0 for a positive

number. However, doing this would not make full use of the available numbers. For an 8-bit byte, we would be able to count down to 1111111, which in the new system would be −127, and we would be able to count up to 01111111, which in the new system (and in two's complement) would be +127. However, we would have two numbers being used for zero: 10000000 and 00000000, which would be negative zero and positive zero. The two's complement system makes the best use of the available space by not having two zeroes, and by having one extra negative number. We can count from −128 up to +127. The system is ever so slightly more complicated than it could be, but it enables us to store one more number. It also allows us to perform addition with negative numbers more easily, as we will see shortly.

Flipping the bits

The simplest way to flip the bits of a hexadecimal number is to convert it to binary [each hex digit is turned into 4 binary digits], then flip the bits, and then convert the result back to hex [each group of 4 binary bits are turned into 1 hex digit.] Alternatively, it can be quicker to remember which hex digit becomes which hex digit when the bits are flipped, as shown in this table:

Hex digit	**Hex digit after the bits have been flipped**
0	f
1	e
2	d
3	c
4	b
5	a
6	9
7	8
8	7
9	6
a	5
b	4
c	3
d	2
e	1
f	0

A rule for remembering the table is that 0x7 becomes 0x8, and 0x8 becomes 0x7. The values either side become turned into the values the other side of 7 or 8. We only really need to know half the table as the top half is mirrored in the bottom half. If we learn the table, we will be able to know instantly, for example, that if the bits are flipped in 0x053e2076, we will end up with 0xfac1df89. If you were going to be converting a lot of numbers without a calculator, it might be useful to learn the table.

Maths

One reason that two's complement is a useful system is that we can perform addition and subtraction on two two's complement numbers without needing to know if they are two's complement or not, and the results will still be correct. As a simple example, we will add −1 (0xff) and +1 (0x01). As binary, these are:
11111111
... added to:
00000001

If we were dealing with *words, dwords or qwords*, the above addition would result in the nine-digit number:
100000000
However, as we are dealing in *bytes,* and bytes only have 8 bits, the whole number rolls over to zero. Therefore, the result is 00000000 in binary, which is 0x00 in hex.

We will add the bytes −3 (0xfd) and −4 (0xfc). This addition is:
11111101
... added to:
11111100

The result is 9 bits long:
111111001
... but because we are dealing with bytes, this is truncated to 8 bits by ignoring the ninth bit, as so:
11111001
... which is 0xf9 in hex, which, *as a two's complement number*, is −7 in decimal.

We will add the two's complement numbers 0xf7 and 0x23. *As two's complement numbers*, these are −9 and +35 in decimal. In binary, the calculation is as so:

11110111
... added to:
00100011
... which is the 9-digit number:
100011010
... but as we are dealing in bytes, we truncate it to 8 bits, and have:
00011010

We can instantly tell that this is a positive number because the leftmost bit is zero. This result is 0x1a in hex, which, whether we treat it as a two's complement number or not, is +26 in decimal.

More on negative numbers

In programming, a common term for a two's complement number is a "signed integer", where the word "signed" means that the decimal equivalent would be portrayed with a plus sign or a minus sign. A normal number would be called an "unsigned integer". A signed integer might be positive or negative, and will always be portrayed using the two's complement system. On the other hand, an unsigned integer will always be positive, and will always be portrayed normally with binary or hex numbers.

In higher-level programming languages, the details of what a two's complement number is, or how it is encoded, are generally hidden from the programmer. If you are dealing with files containing data stored as two's complement numbers, it is helpful to have at least a passing understanding of how they work.

Although two's complement seems like a slightly contrived way of encoding negative integers, computer processors have specific commands that work with them, and the system is generally accepted.

There are two likely ways in which using two's complement numbers will lead to mistakes:

- The first is not knowing whether a byte, word, dword or qword should be interpreted as being a two's complement number or not. A value that should be interpreted as a negative number represents a completely different number if it is treated normally. The most extreme example of this is 0xffffffffffffffff, which, as a two's complement qword is hugely lower than it is as a normal number.

  Whether a byte, word, dword or qword is acting as a two's complement number relates only to whether someone wants to treat it as such. For a byte sitting in memory, there is literally no difference between 0xff meaning "255" in decimal, and 0xff meaning "−1" in decimal. They are both 0xff. You cannot tell if a byte is meant to be a signed integer or not by looking at it. The processor itself does not even know unless it is in the middle of operating on such a value. However, if you know that the byte is a signed integer because you created it yourself, for example, or because you know the context of how it is being used, you can perform maths on it that uses this knowledge. Similarly, once a computer processor is told to act as if the byte is a signed integer, it can treat it accordingly.

- The second way of making mistakes is to forget that the highest possible positive number in a two's complement number is 1 less than half of what it would be normally. For example, a byte can normally count up to 0xff (255 in decimal), but if it is being used to hold a two's complement number, the maximum is only 0x7f (127 in decimal).

  If 1 were added to 0x7f as a *normal* byte, we would get 0x80, which is +128 in decimal.

  If we add 1 to 0x7f, *when 0x7f is being treated as a two's complement number*, *and whatever it is that is doing the adding knows this*, we will get 0x00 because we would have gone past the maximum possible positive value, and rolled over to zero. A computer processor would keep the number positive and not let it go above 0x7f.

  If we add 1 to 0x7f, *when it is being treated as a two's complement number*, *but whatever is doing the adding does *not* know this*, we would end up with 0x80, which, in two's complement, is −128. Instead of the number increasing by 1, it has fallen by 255.

The largest positive number we can have in a two's complement signed byte is:
+127 in decimal, which is 0x7f in hex and 01111111 in binary.

The largest negative number we can have in a two's complement signed byte is:
−128 in decimal, which is 0x80 in hex and 10000000 in binary.

If we ignore zero, the *smallest* positive number we can have in a two's complement signed byte is:
+1 in decimal, which is 0x01 in hex and 00000001 in binary.

The smallest negative number we can have in a two's complement signed byte is:
−1 in decimal, which is 0xff in hex and 11111111 in binary.

Non-integers

We have just seen how computers store negative numbers in hexadecimal and binary. The method is essentially a contrived workaround that redefines how hexadecimal and binary normally work. Now we will look at the standard way that computers store non-integers (fractions) using hex and binary. This also redefines how hex and binary normally work.

A computer stores a non-integer using binary or hex by treating the number as an exponential. More specifically, it rephrases the number so that it becomes a single digit value followed by a decimal point and any number of digits afterwards, multiplied by "2 raised to a particular power". [Strictly speaking, it is not a decimal point, but a *binary* point because we will be working in binary.]

Throughout this explanation, we will get closer and closer to the actual implementation used by computer processors, however to keep the explanation simple, we will advance step by step.

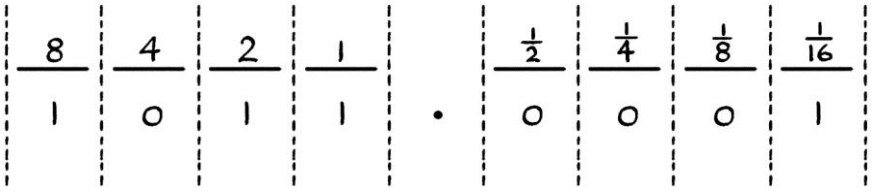
The concept of binary non-integers

Although a computer processor has to use a contrived way to deal with binary non-integers, away from computers, we can portray them in the same way that we would portray decimal non-integers – as an integer followed by a *binary* point followed by the binary digits of the fraction part. [A decimal point is used in decimal numbers and a binary point is used in binary numbers.] For example,

away from a computer, “1011.0001” would be a perfectly valid way to express a binary number. However, a computer processor would be unable to use this method because the processor works with bytes, words, dwords and qwords, and there is no way to place a binary point in the middle of one of these bit groupings.

When thinking about a number such as “1011.0001”, it helps to remember the binary number columns. We know about the integer columns that extend to the left, becoming ever higher: ones, twos, fours, eights, sixteens and so on. The “fraction” columns extend to the right, becoming ever smaller: halves, quarters, eighths, sixteenths, thirty-seconds, sixty-fourths and so on. The number columns to the right of the binary point are all 1 divided by a power of 2.

The binary number “1011.0001” can be thought of as it appears in the following picture:



When we think of a *decimal* number with digits after the decimal point, we are really saying that its fraction part is the sum of so many units of one column, added to so many units of another column, added to so many units of another column, and so on. Given that the number columns after the decimal point are tenths, hundredths, thousandths, ten thousandths and so on, the decimal number 0.7903, for example, really means the sum of 7 lots of tenths added to 9 lots of hundredths, added to zero lots of thousandths, added to 3 lots of ten thousandths. We could express the sum as so:
7 * 0.1
... added to:
9 * 0.01
... added to:
0 * 0.001
... added to:
3 * 0.0001

The nature of storing fractions in this way means that if the fraction part of a number cannot be created by adding multiples of tenths, hundredths, thousandths, ten thousandths and so on, then it cannot be expressed by a finite number of decimal places. For example, we cannot express the number created by dividing 1 by 3 using a finite number of decimal places. This is because 1 divided by 3 cannot

be expressed as a sum of tenths, hundredths, thousandths and so on. Expressing 1 divided by 3 ends up with an infinite number of decimal places: 0.3333333333333... and so on forever.

Storing *binary* numbers with fractions is similar to storing decimal numbers with fractions. If we have the binary number 0.111111, then it is really the sum of:
1 * 0.5
... added to:
1 * 0.25
... added to:
1 * 0.125
... added to:
1 * 0.0625
... added to:
1 * 0.03125
... added to:
1 * 0.015625

If we try to encode any non-integer in binary, then it must be possible to express the part after the binary point as the sum of one or more of the following:

0.5
0.25
0.125
0.0625
0.03125
0.015625
0.0078125
0.00390625
0.001953125
0.0009765625
... and so on.

If the fraction part cannot be expressed as the sum of one or more of those values, then it will require an infinite number of digits after the binary point. Even if it can be expressed as the sum of one or more of those values, it might need many more digits after the binary point than the same number expressed in decimal would require after the decimal point.

Converting decimal non-integers to binary

For this section, we need to remember how to convert a decimal *integer* into a binary *integer*, as mentioned earlier in this chapter. The method is as follows:

- We divide the integer by 2. We take the remainder, which will be 0 or 1, and put it into the ones column.
- If the integer part of the division was not zero, we divide it by 2. We take the remainder, which again, will be 0 or 1, and we put it into the twos column.
- If the integer part of the previous division was not zero, we divide it by 2. We take the remainder, which again, will be either be 0 or 1, and we put it into the fours column.
- We continue in this way until we reach a time when the integer part of the previous division was zero.

[If it seems as if the resulting binary number will be very long, it is easier to convert a decimal number into hex, and then convert the hex number into binary.]

The simplest way to convert a decimal non-integer to binary is to keep multiplying the decimal number by 2 until it becomes an integer, and then to convert that integer into a binary integer. We then divide the binary integer by the amount by which we scaled the original number. This division will always be a power of 2.

Dividing a binary number by a power of 2 is easier than it sounds – we just slide the binary point one digit to the left for each power of 2. For example, if we needed to multiply our decimal non-integer by 2 ten times to create an integer, we would need to divide the equivalent binary integer by 2 ten times, which we would do by sliding the binary point to the left by ten digits.

As an example, we will convert 34.25 to binary. First, we turn it into an integer, by multiplying it by 2 two times. We end up with 137. We then convert 137 into binary. It is 10001001. As we multiplied our original number by 2 twice to make it into an integer, we need to *divide* this number by 2 twice. This means we just move the binary point two digits to the left. We end up with 100010.01 as our result.

Although it is simple to do, the “multiplying by powers of two to turn a number into an integer, then dividing the binary equivalent by powers of two” method reveals a particular problem with converting non-integers into binary in general. For example, if we had the decimal number 34.55, it would require multiplying by two 42 times before it became an integer. It would become 151,952,506,958,643. We would have to convert that into a binary number, and then shift the binary

point 42 digits to the left. This instantly tells us that the binary number would have 42 digits after the binary point.

The seemingly simple *decimal* number 0.1 requires multiplying by 2 fifty times before it becomes an integer. This means that the binary number would need to be divided by 2 fifty times, and so the result would have 50 digits after the binary point.

Another problem is that it might not always be possible to convert a number into an integer by multiplying by 2. As a simple example, no matter how many times we multiply a third by 2, it will never become an integer.

These problems are not related to this method of converting from decimal to binary. Instead, they relate to how it is sometimes difficult or impossible to represent a non-integer in binary correctly, in the same way that it is sometimes difficult or impossible to represent a non-integer in decimal correctly.

A good example of a number that cannot be portrayed in any (sensible) number system is π. To portray the fraction part of π would require an infinite number of digits, no matter whether we used decimal, binary, hexadecimal, or any number system based on integers. [We could use a number system where the number columns were somehow based on multiples of π, but such a numbering system would be inaccurate when portraying any numbers that were not related to π.]

These problems are sometimes misinterpreted as a flaw in the way that computers store numbers, when in reality, they are just a consequence of using binary. Problems of this type would occur no matter which number system we used.

If it turns out that we need to multiply our decimal non-integer by 2 a huge number of times to turn it into an integer (or an infinite number of times), then we have to make the choice of where to truncate the digits after the binary point. This is always an arbitrary choice and depends on what we are doing. The rule for rounding up a *decimal* number is that if the digit to the right of where we are truncating it is 5 or more, the digit to the left has 1 added to it. Otherwise, the digit is left alone. For example:

7.00015 becomes 7.0002 to four decimal places.
7.00014 becomes 7.0001 to four decimal places.

The rule for binary is that if the digit to the right of where we are truncating is 1, we add 1 to the digit to the left. Otherwise, we leave the digit to the left alone. For example:

1.00001 becomes 1.0001 to four binary places.
1.00000 becomes 1.0000 to four binary places.
1.00011 becomes 1.0010 to four binary places.
1.00010 becomes 1.0001 to four binary places.

Non-integers for computers

We will now move on to how computer processors store and work with binary non-integers. The exact details are easiest to understand if we first go through the method with *decimal* non-integers.

When thinking in decimal, any number except zero can be rephrased to be a non-zero single digit followed by a decimal point and any number of digits afterwards, all multiplied by ten raised to a particular power. [We will think about how to encode zero later.] For example:

300 can be rephrased to be $3 * 10^2$
7000 is $7 * 10^3$
20 is $2 * 10^1$
330 is $3.3 * 10^2$
7078 is $7.078 * 10^3$
22 is $2.2 * 10^1$

1234 is $1.234 * 10^3$
12.34 is $1.234 * 10^1$
1.234 is $1.234 * 10^0$
0.1234 is $1.234 * 10^{-1}$
0.01234 is $1.234 * 10^{-2}$
0.000000001234 is $1.234 * 10^{-9}$
12,340,000 is $1.234 * 10^7$

−77 is $-7.7 * 10^2$
−314,159 is $-3.14159 * 10^5$
−11.0000001 is $-1.10000001 * 10^1$

From all of these examples, we can see that whether or not a decimal number is an integer, once we have rephrased it, there are four identifying characteristics:

- Whether it is positive or negative.
- The single digit before the decimal point.
- The digits after the decimal point.
- The exponent of the number 10.

These four attributes are enough to identify any possible number except zero.

For example, if we have the number −0.1234, it can be rephrased as:
$-1.234 * 10^{-1}$. The identifying characteristics are:

- It is negative. We could say that its “sign” is negative.
- The value of the digit before the decimal point is 1.
- The digits after the decimal point are 234.
- The exponent of the number 10 is −1.

The number 0.00000007954 can be rephrased as $7.954 * 10^{-8}$. The identifying characteristics are:

- It is positive. We could say that its “sign” is positive.
- The value of the digit before the decimal point is 7.
- The digits after the decimal point are 954.
- The exponent of the number 10 is −8.

The number 2,000,007 can be rephrased as $2.000007 * 10^{6}$. The identifying characteristics are:

- It is positive.
- The value of the digit before the decimal point is 2.
- The digits after the decimal point are 000007.
- The exponent of the number ten is 6.

The number −99,999.12 is $-9.999912 * 10^{4}$. The identifying characteristics are:

- It is negative.
- The value of the digit before the decimal point is 9.
- The digits after the decimal point are 999912.
- The exponent of the number ten is 4.

For each of these examples, we can write out the identifying characteristics as a line of text:

−0.1234: negative, 1, 234, −1
0.00000007954: positive, 7, 954, −8
2,000,007: positive, 2, 000007, 6
−99,999.12: negative, 9, 999912, 4

Knowing just "negative, 1, 234, −1" is enough to know that we are talking about the number −0.1234. There is no other number for which these four characteristics apply.

The system in binary

We can use the same idea in binary. When thinking in binary, any number except zero can be rephrased to be a single non-zero digit followed by a *binary* point and a number of *binary* digits afterwards, multiplied by 2 raised to a particular power.

[It always helps to know that we can multiply a binary number by 2 by shifting the binary point one digit to the right. We can divide a binary number by 2 by shifting the binary point one digit to the left.]

To make the idea easier to understand, we will first look at some examples with the values as binary numbers, *but with the exponentials kept as decimal numbers*. This makes it easier to see how far the binary point has been moved. [Mathematically, it could be confusing to have calculations containing both binary and decimal values, but for the sake of the explanation, it makes things easier to understand.]

1100 can be rephrased as $1.1 * 2^{3}$
10001111 can be rephrased as $1.0001111 * 2^{7}$
1011 can be rephrased as $1.011 * 2^{3}$
−11001100 can be rephrased as $-1.1001100 * 2^{7}$

1100.001 can be rephrased as $1.100001 * 2^{3}$
−0.0111 can be rephrased as $-1.11 * 2^{-2}$
−0.00000001 can be rephrased as $-1 * 2^{-8}$
11.1111 can be rephrased as $1.11111 * 2^{1}$

We can identify any binary number (except zero) using just 4 attributes:

- Whether it is positive or negative.
- The value of the digit before the binary point.
- The binary digits after the binary point.
- The exponent of the number 2.

[We will see how to encode zero later in this chapter.]

For example, −0.0111 can be rephrased as $-1.11 * 2^{-2}$. It can be identified with these four attributes:

- It is negative.
- The digit before the binary point is 1.
- The digits after the binary point are 11.
- The exponent of the number 2 is −2.

The number 11.1111 can be rephrased to be $1.11111 * 2^{1}$. It can therefore be identified by these four attributes:

- It is positive.
- The digit before the binary point is 1.
- The digits after the binary point are 11111.
- The exponent of the number 2 is 1.

One interesting consequence of doing this in binary is that the single digit before the binary point will *always* be 1. This is because we always slide the number to have one non-zero digit before the binary point, and binary only has one type of non-zero digit, which is 1. As the digit before the binary point will always be 1, we do not need to bother making a note of it. Its existence is implied. Therefore, we can encode any binary number using just three attributes:

- Whether it is positive or negative.
- The binary digits after the binary point.
- The exponent of the number 2.

For example, 1011 is also $1.011 * 2^3$. The three attributes of this are:

- It is positive.
- The digits after the binary points are 011.
- The exponent of the number 2 is 3.

We will now make this explanation one step more complicated by giving the exponentials in binary too. First, this means the base of the exponential changes from being "2" in decimal to being "10" in binary. Second, the exponent becomes a binary number too. The previous binary examples rephrased to have binary exponentials are:

1100 is $1.1 * (10)^{11}$
10001111 is $1.0001111 * (10)^{111}$
1011 is $1.011 * (10)^{11}$
−11001100 is $-1.1001100 * (10)^{111}$
1100.001 is $1.100001 * (10)^{11}$
−0.0111 is $-1.11 * (10)^{-10}$
−0.00000001 is $-1 * (10)^{-1000}$
11.1111 is $1.11111 * (10)^{1}$

[Note that because we are writing the binary numbers and exponentials out by hand, we can still use negative signs with them. A computer would not be able to do this, and would have to use two's complement. This is a potential source of confusion. Away from computers, we have much more leeway in writing binary – we can use binary points and negative signs. When a computer deals with non-integers and negative values, it has to use the contrived alternatives.]

We will look at one of these examples. The binary number 1100.001 can also be portrayed as $1.100001 * (10)^{11}$, where all the values are in binary. The three attributes that distinguish this number from any other number are therefore:

- It is positive.
- The binary digits after the binary point are 100001.
- The exponent of the binary number 10 is 11 (in binary), which is 3 in decimal.

We can summarise all of the examples with their sign (whether they are positive or negative), the digits after the binary point, and the exponent:

1100:	positive, 1, 11
10001111:	positive, 0001111, 111
1011:	positive, 011, 11
−11001100:	negative, 1001100, 111
1100.001:	positive, 100001, 11
−0.0111:	negative, 11, −10
−0.00000001:	negative, 0, −1000
11.1111:	positive, 11111, 1

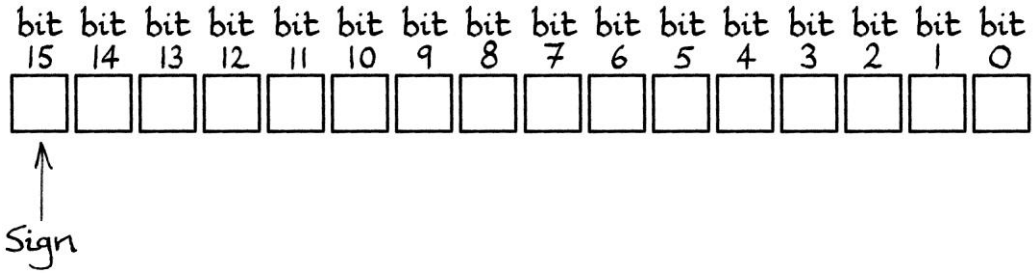
Floating-point numbers

The method of identifying any number apart from zero by the three attributes is the basis of how computer processors store and work with non-integers. Processors store the three attributes in what is called a “floating-point” number. We can think of the process of obtaining the three attributes as *sliding* the binary point so that we have one digit before the binary point. We could also think of the binary point as *floating* left or right. A floating-point number can indicate any number at all, whether it is an integer or a non-integer, and whether it is positive or negative.

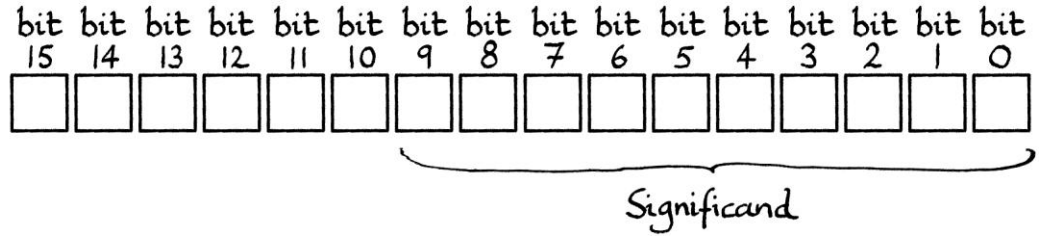
When a value is encoded as a floating-point number, the three attributes are joined together so that they fit within particularly sized groups of bits. At the time of writing, for Intel and AMD 64-bit processors, the minimum sized group of bits is a 16-bit word. As this is the shortest size to contain a floating-point number, we will look at these numbers first.

Of our three attributes of a number, the first refers to whether the number is positive or negative. This is called “the sign”, because it relates to the “plus sign” or “minus sign” that would normally be used to prefix a number. The sign is represented by the leftmost bit of the 16-bit word. If there is a plus sign, then this is set to zero, if there is a minus sign, it is set to one. Thinking of this the other way around, if the leftmost bit of the word is zero, the number is positive; if the leftmost bit is one, the number is negative.

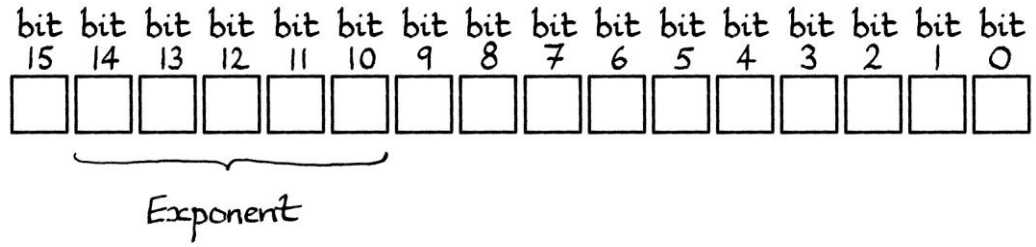
In the following picture, each of the boxes represents a binary digit. The sign is the leftmost bit (which is bit 15):



The second attribute refers to the digits after the binary point. It is called the "significand" on account of it being considered the "significant" part of the number in this system. In a floating-point number contained within a 16-bit word, this is placed in the rightmost part from bit 9 to bit 0 [Remember that we number the bits from right to left, and count from zero upwards]. This means that it will be ten bits long. To fit the significand into the space available, it might have zeroes placed after it, or it might be truncated or rounded up.

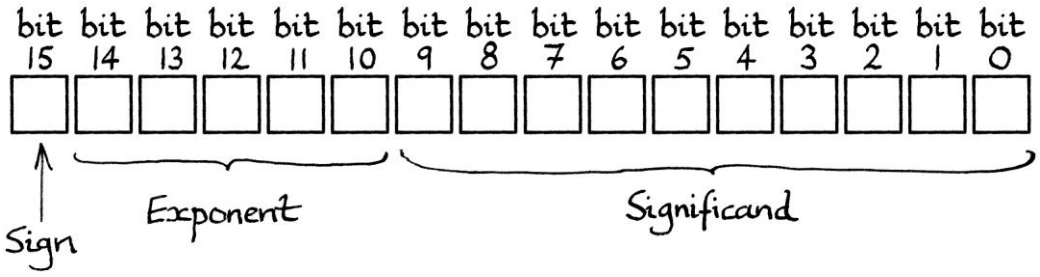


The third attribute refers to the exponent of the exponential in binary. In a floating-point number contained within a 16-bit word, this is placed in the middle, from bit 14 to bit 10. This means that it will be 5 bits long. In practice, the actual exponent is adjusted before it is stored, as we will see shortly.



The full layout of a floating-point number contains the following, in this order:

- the sign
- the exponent
- the significand (in other words, the value after the binary point).



The exponent

Among the possible problems we can see with this system so far is that the exponent might need to be negative. Therefore, we have to decide how to encode the exponent as a negative binary number. One way would be to use two's complement. In practice, however, a fixed value is added to the exponent so that it always becomes positive. [That value is subtracted when the floating-point number is decoded.] The value added to the exponent varies depending on how many bits we are using to store the floating-point number. For a 16-bit word, where the exponent is contained within 5 bits, this number will be 15 (in decimal), which is 01111 in binary. [Note that I am writing it as 01111 instead of 1111 so that it takes up 5 bits, and so removes any confusion as to what happens if we end up with just 1111, which is what would happen if the exponent were zero.]

For 16-bit words, whatever the exponent, it will have 15 added to it to make it into a positive number between, and including, 00001 and 11110 in binary [which are 0x01 and 0x1e in hex, or 1 and 30 in decimal.] It is important that the exponent ends up as a number between 00001 and 11110 and not between 00000 and 11111. This is because exponents that are all zeroes or all ones have special meanings, as we shall see later. This restriction means that the original exponent before the addition can only be between −14 and +15. Therefore, we can only encode numbers that have an exponential between 2^{-14} and 2^{15}. In binary, we would say that they can only be between $(10)^{-1110}$ and $(10)^{1111}$.

The value that is always added to the exponent is called the "bias". We can say that the "exponent is biased", where the word "bias" in this sense ultimately means a constant distortion in one direction.

Here are some examples of storing the exponent as a "biased" value – in other words, they are examples of storing the exponent with the addition of the fixed value of 15. When using 16-bit words to encode floating-point numbers, there are 5 available bits for the biased exponent.

Original number and exponential (in binary)	**Exponent (in binary)**	**Exponent after the addition of 01111 (in binary) [The biased exponent]**
$-1.1010101 * (10)^{1}$	1	10000
$1.11101 * (10)^{101}$	101	10100
$1.001 * (10)^{1000}$	1000	10111
$1.11 * (10)^{-1000}$	−1000	00111
$1.1 * (10)^{-1110}$	−1110	00001
$1.1 * (10)^{1111}$	1111	11110

The full layout

For a floating-point 16-bit word, the first bit holds the sign, the next 5 bits hold the exponent, and the next 10 bits hold the significand. We can portray this with a template that indicates the 16 bits with letters standing in for the bits:

sEEEEEnnnnnnnnnn

This is a template that we can use to fill in the bits. The "s" (for "sign") will be replaced with the sign bit. The five letters "EEEEE" (for "Exponent") will be replaced by the biased exponent. The ten letters "nnnnnnnnnn" (for "significand") will be replaced by the significand. The choice of letters is arbitrary, but these are easy to distinguish from each other. The template is just to make this explanation easier to understand, and normally such a thing would not be used.

Using the template

We will look at a complete example. We start with our empty 16-bit word template:
sEEEEEnnnnnnnnnn

We will look at the first example in the previous table: −1.1010101 * $(10)^1$. The sign is negative, so we set the leftmost bit to 1:
1EEEEEnnnnnnnnnn

The exponent of our number is 1. Therefore, we add it to 01111 (in binary), and we end up with the binary number 10000. This then goes into the exponent section ["EEEEE"] of our 16-bit word, which is from bit 14 to bit 9. Our word so far will be:
110000nnnnnnnnnn

The significand (containing the digits after the binary point) is: 1010101. There are only 7 digits, but the section for the significand for a 16-bit word is ten digits long. Therefore, we extend our binary digits to ten digits by putting zeroes *at the end*, as so: 1010101000. These extra digits do not affect the number that we are encoding in any way. These digits are then placed into the remaining bits as so:
1100001010101000

[If there had been more than ten digits after the binary point, we would have had to round up or truncate the digits.]

We now have our finished number:
1100001010101000
... which, given spaces to make it easier to read, is:
1100 0010 1010 1000
... and this is:
0xc2a8
... in hexadecimal.

Our final 16-bit floating-point number is 0xc2a8. As with two's complement numbers, this is only a floating-point number *if we treat it as such*. There is no way of knowing that this 16-bit word is a floating-point number by looking at it. Floating-point numbers require context to be interpreted correctly. With no context, we might treat this as a normal unsigned integer word, in which case, it would be 49,832 in decimal. If we mistakenly thought it was a signed two's complement integer, it would have the value −0x3d58, which is −15,704 in decimal. From the point of view of computers, 0xc2a8 only means −1.1010101 * $(10)^1$ if the computer processor is told that 0xc2a8 is a floating-point number when it is supplied with the number.

16-bit example 1

We will now look at several examples.

We will say that we want to convert the binary number:
1100.0011
... into a 16-bit floating-point number. First, we arrange it to be a single digit preceding the binary point multiplied by an exponent of 2. If we give the exponential entirely in binary, we will have:
$1.1000011 * (10)^{11}$

We split this into the three parts that make up a floating-point number:

- The sign is positive.
- The significand (the digits after the binary point) is 1000011.
- The exponent is 11 in binary (which is 3 in decimal).

We will start with our 16-bit template:
sEEEEEnnnnnnnnnn

As the sign is positive, the leftmost bit is set to zero:
0EEEEEnnnnnnnnnn

Our exponent is 11, but we need to turn this into a biased exponent by adding 01111. This gives us 10010. We can put this into the exponent section:
010010nnnnnnnnnn

Our significand is 1000011. This is 7 digits long, but we have space for 10 digits. Therefore, we extend it to ten digits by putting zeroes *at the end*. We end up with this: 1000011000. We can then place this into the significand section:
0100101000011000

We now have our finished binary word. We will convert it into hexadecimal – first, we split it into 4-bit groups to make it easier to read:
0100 1010 0001 1000
... then we convert each 4-bit group into the relevant hex digit:
4 a 1 8
... and our final hex word is:
0x4a18

16-bit example 2

Next, we will convert −1111000011 to a floating-point number. Rephrased to have a single digit integer part and to be multiplied by an exponential, we have:
$-1.111000011 * (10)^{1001}$

- The sign is negative
- The significand is 111000011. This is 9 digits long, but we need it to be 10 digits long, so we put a zero at the end: 1110000110.
- The exponent is 1001. Therefore, the biased exponent will be:
  1001 + 01111 = 11000.

We will start with our 16-bit template:
sEEEEEnnnnnnnnnn

As the sign is negative, we set the leftmost bit to 1:
1EEEEEnnnnnnnnnn

We then fill in the exponent part with our biased exponent 11000:
111000nnnnnnnnnn

We then fill in the significand:
1110001110000110

This is our final floating-point number, but we will convert it to hex. First, we split it into 4-bit sections:
1110 0011 1000 0110
... then we convert each 4-bit section into one hex digit:
e 3 8 6
... which is:
0xe386

16-bit example 3

Now we will convert the *decimal* number 123.25 into a 16-bit binary or hex floating-point number. First, we have to convert the decimal number into a binary number with a binary point. To do this, we keep multiplying the decimal number by 2 until it becomes an integer:
123.25 * 2 = 246.5
246.5 * 2 = 493

Therefore, we have to convert 493 into binary. As this looks as if it will produce a long binary number, we will convert it into hex first, and then convert the result into binary. Doing this is quicker and simpler than converting directly to binary. First, we divide the number by 16. This is 30.8125, which is 30 and a remainder of 13. We convert 13 to hex – it is 0xd – and put it into the 1s column. We then divide 30 by 16, and we get 1.875, which is 1 and a remainder of 14. We convert 14 to hex – it is 0xe – and put it in the 16s column. We then divide 1 by 16, and get 0 and a remainder of 1. We put the 1 into the 256s column, and we have finished the conversion into hex. The result is:
0x1ed
... which is:
0001 1110 1101 in binary (with spaces to make it easier to read)
... or:
000111101101 (without the spaces)
... or:
111101101 (without the preceding zeroes).

As we multiplied our original number (123.25) by two twice to turn it into an integer, we need to divide our binary number by two twice. We do this by moving the decimal point two digits to the left to produce:
1111011.01

We now have our binary number ready for conversion to a floating-point number. We alter it so the integer part is only one digit, and it is multiplied by an exponential with 2 as the base. Entirely in binary, this is:
$1.11101101 * (10)^{110}$
[where the binary number 110 is equal to the decimal number 6.]

The three attributes of this number are:

- The sign is positive.
- The significand is 11101101
- The exponent is 110

We take our 16-bit template:
sEEEEEnnnnnnnnnn

The sign bit will be 0 because the number is positive:
0EEEEEnnnnnnnnnn

The exponent needs biasing, so we add 01111 to it:
01111 + 110 = 10101

We then put that into our template:
010101nnnnnnnnnn

The significand is 11101101, which is 8 bits long. As it needs to be ten digits long, we put two zeroes on to the end to produce 1110110100. We then put this into our template, and we have the finished floating-point number:
0101011110110100

We will split this into 4-bit sections to make it easier to read:
0101 0111 1011 0100
... and then we convert each 4-bit section to hex:
5 7 b 4
... and our final floating-point number as a hex word is:
0x57b4

16-bit example 4

We will convert the *binary* number −1.1 into a floating-point number. Portraying this as a single digit integer, followed by a fraction and an exponential, it becomes:
$-1.1 * (10)^0$

- The sign is negative
- The significand is 1, which we will extend to 10 digits by putting nine zeroes after it: 1000000000
- The exponent is zero, to which we will add 01111 to become 01111

We take our 16-bit template:
sEEEEEnnnnnnnnnn

The sign is negative, so the leftmost bit is set to 1:
1EEEEEnnnnnnnnnn

We put the biased exponent in:
101111nnnnnnnnnn

We put the significand in, and we have our finished floating-point number in binary:
1011111000000000

To convert this into hex, we split the number into 4-bit sections:
1011 1110 0000 0000
... then we convert each 4-bit section to a hex digit:
b e 0 0
... and our floating-point number in hex is:
0xbe00

16-bit example 5

Now we will convert 1001.111100001010101 into a floating-point number. As a value multiplied by an exponential, this is:
$1.001111100001010101 * (10)^{11}$
... where the binary exponent 11 is in 3 in decimal.

We start with our template:
sEEEEEnnnnnnnnnn

- The sign is positive, so the sign bit will be zero:
  0EEEEEnnnnnnnnnn

- The exponent is 11 in binary. We add this to 01111, and we get 10100, which we put into the template:
  010100nnnnnnnnnn

- The significand is 0011111000010101010. The maximum number of digits we can have for the significand is ten. Our significand is eighteen digits long. Therefore, we have to round it to ten digits, which means that we will lose some accuracy. This is an unavoidable consequence of floating-point numbers, but is more likely to happen when we are using 16-bit floating-point numbers as we are doing here. A 32-bit, 64-bit, or 80-bit floating-point number can contain more digits in the significand. Our significand rounded up to ten digits is 0011111000. We put this into the template:
  0101000011111000

Our final floating-point number in binary is:
0101000011111000

To convert this into hex, we split it into 4-bit sections:
0101 0000 1111 1000
... and convert each one to a single hex digit:
5 0 f 8
... so our final floating-point number in hex is:
0x50f8

16-bit example 6

We will now encode the following binary number as a floating-point number [shown with spaces to make it easier to read]:
0.0000 0000 0000 01

This has 14 decimal places. As a value multiplied by an exponential, it becomes:
$1 * (10)^{-1110}$

As significands always show the digits to the right of the number 1, we will write it with some zeroes after the binary point to make it clearer and easier to work with:
$1.0000 * (10)^{-1110}$

- The sign is positive, so the sign bit will be zero.

- The significand is 0000 (although it would be equally valid to say that it is 0 or 00 or 000, or 000000, or any number of zeroes). We will extend it to ten digits as 0000000000.

- The exponent is −1110, to which we must add 01111. The addition is easiest if we think of it as 01111 – 1110 = 1. [If we were confused by binary maths, we could convert the numbers to decimal, then add them, then convert the result back to binary. We would be adding −14 and +15, which results in +1 in decimal, which is +1 in binary.] As the biased exponent needs to be 5 digits long, we will write this as 00001.

Out template is:
sEEEEEnnnnnnnnnn

We fill in the sign:
0EEEEEnnnnnnnnnn

We fill in the biased exponent:
000001nnnnnnnnnn

Then we fill in the significand, which is all zeroes, and our completed floating-point number is:
0000010000000000

We split this into 4-bit sections:
0000 0100 0000 0000
... and convert each one to a hex digit:
0 4 0 0
... and our floating-point number as a hex word is:
0x0400

Capacity

Example 5 showed how there is a limit to the number of digits after the binary point that we can store – the limit is ten digits. However, in example 6, we managed to store a number with 14 digits after the binary point. The difference is that in example 6, all except the last digit was zero.

It is difficult to say exactly how large or small a number we can store in a floating-point word. This is because the number of digits we can store depends on the nature of the bits in the number we have.

It is possible to store a 16 digit binary *integer* – we would actually be storing only 15 digits because the first 1 is implied. However, we would only be able to fit in the first ten digits after the first 1. The digits after the first 1 are rounded up to ten digits. If the digits after the first ten digits were already zeroes, this will not make any difference – we will be storing the number we have completely accurately. However, if the digits were not zeroes, we would be storing an approximation. This approximation can still be useful as the size of the stored number will be very similar to the number we started with – only the lower digits will be zeroed out. The idea is analogous to having, say, the decimal number 1,300,021 but only being able to store it as 1,300,000.

As an example, if we tried to store this 16 digit binary integer:
1111 0000 1111 1111
... as a 16-bit floating-point word, we would only be able to store the first ten digits after the first 1. It would end up being converted to:
1111 0000 1110 0000
This is because it would be rounded up to ten significant digits after the first 1. The first number is 61,695 in decimal. The second number is 61,664 in decimal. We still have a similarly high number, but it is not exactly the same.

We could store a 15 digit binary number if it is entirely a fraction less than 1. We would actually only need to store 14 digits because the first 1 is implied. However, we would only be able to store the first ten digits after the first 1 because the digits after the first 1 would be rounded up to ten decimal places. Again, depending on the number we are storing, this might not matter.

If we wanted to have the binary number:
0.1111 0000 1111 1111
... as a 16-bit floating-point word, it would end up being changed to:
0.1111 0000 1110 0000
... because it would be rounded up after the first ten decimal places after the first 1.

16-bit example 7

We will now decode a 16-bit floating-point number to find out the number it represents. We will decode: 0x8e23. In binary this is:
1000 1110 0010 0011 (with spaces to make it easier to read)
... or:
1000111000100011 (without spaces)

By putting the number against our template, we can see which bits make up the sign, exponent and significand:

```
sEEEEEnnnnnnnnnn
1000111000100011
```

... or to make it clearer:

```
s EEEEE nnnnnnnnnn
1 00011 1000100011
```

- The sign bit is 1, so the number is negative.

- The exponent bits are 00011. [As this is less than 01111, it means that the actual exponent must have been negative.] We have to subtract 01111 to find the actual exponent. We calculate: 00011 – 01111 = −1100. If you are still getting used to binary, this can be easier to calculate in decimal: 3 – 15 = −12.

- The significand is 1000100011. This means that the original number that was multiplied by the exponential was 1.1000100011 [We prefix the significand with the implied "1"].

Our decoded binary number is:
$1.1000100011 * (10)^{-1100}$

We need to shift the binary point of 1.1000100011 by 1100 (in binary) digits to the left. This is 12 (in decimal) digits to the left. Each shift of the binary point to the left results in another zero being placed at the start of the number. We end up with the binary number:
0.000000000011000100011

Zero and infinity

So far, we have not been able to store the number zero. This is because zero cannot be expressed as a binary number multiplied by a power of 2 if that binary number has 1 as the integer part. For this reason, the floating-point number system sets aside a special combination of significand and exponent that represents a zero. The exponent and significand are both set to zero, so the exponent becomes 00000, and the significand becomes 0000000000. The sign bit can be either 1 or 0, which means we can have positive zero or negative zero. These ultimately mean the same thing.

Positive zero in a 16-bit word looks like this in binary (with spaces added to make it easier to read):
0000 0000 0000 0000
... which is:
0x0000 in hex.

Negative zero is this in binary:
1000 0000 0000 0000
... which is:
0x8000 in hex.

We can also represent infinity, in which case, the exponent is set to all ones: 11111, and the significand is set to zeroes: 0000000000. The sign bit can be either 1 or 0, which means we can represent positive infinity and negative infinity.

Positive infinity in binary is:
0111 1100 0000 0000
... which is:
0x7c00 in hex.

Negative infinity in binary is:
1111 1100 0000 0000
... which is:
0xfc00 in hex.

32-bit floating-point numbers

Although a 16-bit floating-point word is good for explanations, on Intel and AMD 64-bit processors, a floating-point number as a 16-bit word is only useable in certain situations, and then the processor immediately converts it into a 32-bit floating-point number. Intel and AMD processors mainly use larger sizes, which have lengths of 32 bits, 64 bits and 80 bits.

A 32-bit floating-point number fits into a dword. Its formal name is “a single-precision” floating-point number. [A 16-bit floating-point number is called “a half-precision” floating-point number.] In programming languages such as C, a 32-bit floating-point number is called a “float”. Away from programming, the term “32-bit float” is more descriptive.

A 32-bit floating-point number has a similar layout to a 16-bit one, but it has more room for the exponent and the significand. Specifically, there are 8 bits for the exponent, and 23 bits for the significand. The exponent has to be “biased” by adding it to 127 (in decimal), which is 01111111 in binary or 0x7f in hex. The bits available for the exponent allow it to store the exponents of exponentials from 2^{-126} to 2^{127}. This means that a dword can store 126 binary places or 127 integer digits, but the digits will be rounded up to the first 23.

To make the dword easier to visualise, a 32-bit template would look like this:

sEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnn

... or shown with spaces to make it clearer:

s EEEEEEEE nnnnnnnnnnnnnnnnnnnnnnn

Zero is represented by setting the exponent and significand to all zeroes. Depending on the sign bit, we can have positive zero or negative zero. Positive zero is this in binary (with spaces to make it easier to read):
0000 0000 0000 0000 0000 0000 0000 0000
In hex, it is:
0x00000000

Negative zero is:
1000 0000 0000 0000 0000 0000 0000 0000
... which, in hex, is:
0x80000000

Infinity is represented by setting the exponent to all ones, and the significand to all zeroes. We can represent positive or negative infinity. Positive infinity in binary is:
0111 1111 1000 0000 0000 0000 0000 0000
In hex, it is:
0x7f800000

Negative infinity in binary is:
1111 1111 1000 0000 0000 0000 0000 0000
In hex, it is:
0xff800000

As an example of a 32-bit floating-point number, we will store the binary number:
−1101.000001111

As before, this involves turning it into a number with a single digit integer multiplied by an exponent of 2. It becomes:
$-1.101000001111 * (10)^{11}$

The sign is negative, so we will set the sign bit in our template to 1:
1EEEEEEEEnnnnnnnnnnnnnnnnnnnnnnn

The exponent is 11. We have to turn it into a biased exponent, so we add 01111111 to it, which produces 10000010. We put this into the exponent section:
110000010nnnnnnnnnnnnnnnnnnnnn

The significand is 101000001111. This is twelve digits long, but there are 23 available bits. Therefore, we put eleven zero digits after it to make it the correct length. It becomes 10100000111100000000000. We put it into the significand section, and we have our finished 32-bit floating-point number in binary:
11000001010100000111100000000000

This is a good example of a number that is much easier to read in hex. We split it into 4-bit sections:
1100 0001 0101 0000 0111 1000 0000 0000
... and we convert each 4-bit section to its relevant hex digit:
c 1 5 0 7 8 0 0
... and our number as a hex dword is:
0xc1507800

64-bit floating-point numbers

64-bit floating-point numbers are called “double-precision” floating-point numbers. They are also called “double floats” or even just “doubles”. A more descriptive short name for them is “64-bit floats”.

A 64-bit floating-point number has eleven digits for the exponent and 52 digits for the significand. The exponent of a number that is about to be stored needs first to be added to 1023 in decimal, which is 0x3ff in hex and 01111111111 in binary (this is eleven digits long). The bits available for the exponent allow it to store the exponents of exponentials from 2^{-1022} to 2^{1023}.

Although it does not really make the qword easier to visualise, a 64-bit template looks like this:

```
sEEEEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

... or like this with the parts drawn on different lines:

```
s
EEEEEEEEEEE
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

As with 16-bit and 32-bit floating-point numbers, zero is represented by setting the exponent and significand to zeroes. Positive zero looks like this *in hexadecimal*:
0x0000000000000000
... while negative zero looks like this *in hexadecimal*:
0x8000000000000000

Infinity is represented by setting the exponent to all ones and the significand to all zeroes. Positive infinity looks like this *in hexadecimal*:
0x7ff0000000000000
... while negative infinity looks like this *in hexadecimal*:
0xfff0000000000000

As an example of a 64-bit floating-point number, we will store the following binary number, which I have put into 4-bit sections to make it easier to read:
0.0000 0000 0000 1100 1111 0001 0000 1111

As a value multiplied by an exponential this is:
$1.1001\ 1110\ 0010\ 0001\ 111 * (10)^{1101}$
... where the binary exponent 1101 is 13 in decimal.

To make things simpler, we will use our 64-bit template:
`sEEEEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn`

The number is positive, so the leftmost bit is set to zero:
`0EEEEEEEEEEEnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn`

Our exponent is 1101. For 64-bit numbers, we have to add the exponent to 01111111111 before we can store it. We calculate:
01111111111 + 1101 = 10000001100

We put this into our 64-bit template:

`010000001100nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn`

Our significand is 100 1111 0001 0000 1111. This is 19 digits, but we have space for 52 digits. Therefore, we put extra zeroes after it. We can then put it into our 64-bit number, and we have the finished number, which looks like this in binary:

`0100000011001001111000100001111000000000000000000000000000000000`

To make this more palatable, we will convert it into hexadecimal. We start by splitting it into 4-bit sections:

```
0100 0000 1100 1001 1110 0010 0001 1110 0000 0000 0000 0000 0000
0000 0000 0000
```

Then we convert each 4-bit section to the relevant hex digit:
4 0 c 9 e 2 1 e 0 0 0 0 0 0 0 0
... and we end up with our 64-bit floating-point number in hex as:
0x40c9e21e00000000

80-bit floating-point numbers

The next type of floating-point number uses 80 bits. Unlike 16-bit, 32-bit and 64-bit floating-point numbers, 80-bit floating-point numbers do not fit into a standard grouping of bits. We cannot store an 80-bit number in a byte (8 bits), word (16 bits), dword (32 bits), or qword (64 bits). However, we can still store them in computer memory or in a file.

The formal name for an 80-bit floating-point number is a “double-extended-precision” floating-point number. They are also called “long doubles”, but a more descriptive name for them would be “80-bit floats”.

80-bit floating-point numbers are slightly different from 64-bit, 32-bit and 16-bit floating-point numbers. On Intel and AMD 32-bit and 64-bit processors, they can only be used in what is called the floating-point unit. To explain this most simply, the floating-point unit is the part of the processor that is dedicated solely to dealing with floating-point numbers one instruction at a time. A modern Intel or AMD processor has one floating-point unit, but it also has other parts that can use commands to operate on floating-point numbers in parallel. [Adding in parallel, for example, means that the values in two lists of numbers can be added together more quickly than if they were added one value at a time.] The set of parallel commands are called the “Single Instruction Multiple Data” instructions, or “SIMD” instructions. The commands in this group that operate on floating-point numbers in parallel are called the “Streaming SIMD *extensions*” (or “SSE” for short) because the original SIMD commands only worked on integers. The SSE instructions can work on integers or floating-point numbers, but when working on floating-point numbers, they can only work on 32-bit and 64-bit floating-point numbers. As new processor designs were released, the set of SSE instruction set was added to, and the new additions are called “SSE2”, “SSE3” and so on. 16-bit half-precision

floating-point numbers can only be used with SSE instructions, and then they are immediately converted to 32-bit single-precision floating-point numbers.

Although 80-bit floating-point numbers can store bigger and more accurate numbers, they are less versatile than 32-bit or 64-bit floating-point numbers. An Intel or AMD computer processor can only work with 80-bit floats one at a time in the floating-point unit. It is quicker not to use 80-bit floats if we need to perform the same calculations on several floating-point numbers at the same time. A computer game, for example, would be faster using 32-bit or 64-bit floats with SSE instructions than using 80-bit floats in the floating-point unit. On the other hand, a maths program would be more accurate when using 80-bit floats. Which is the best size depends on what it is we want to do, and how we want to do it. When programming in higher-level languages, if asked to use 80-bit floats, some compilers will ignore the request and use 64-bit floats instead. [A compiler is the program that takes the text of a program's source code and converts it into the finished executable program.]

When a 32-bit or 64-bit float is loaded into an Intel or AMD processor's floating-point unit, it becomes converted into an 80-bit float. When the floating-point unit stores the 80-bit float to memory, it can store it as an 80-bit float, a 64-bit float or a 32-bit float, depending on what it is asked to do. [It can also store it as a rounded up signed integer in the form of a two's complement 16-bit word, 32-bit dword or 64-bit qword.]

80-bit floating-point numbers have 15 bits for the exponent and 64 bits for the significand. What makes them different is that the significand will contain the preceding "1" that is implied with the other sizes of float. Therefore, the leftmost bit of the significand will *generally* be set to 1. [It is set to 0 to indicate zero and "denormalised" numbers, which I will explain later in this chapter.] Given that the leftmost bit is used, there are actually only 63 bits for the significand.

The 15 bits for the exponent mean that the exponent can go from −16,382 to +16,383 in decimal. These are −0x3ffe to +0x3fff in hex, or:
−11 1111 1111 1110
... to:
+11 1111 1111 1110
... in binary (with spaces to make the numbers easier to read).

The exponent needs to be biased by having 16,383 in decimal added to it. This is 0x3fff in hex or 011 1111 1111 1111 in binary

A template for an 80-bit floating-point number is as follows. It consists of 1 bit for the sign, 15 bits for the exponent and 63 bits for the significand. The number is so long that it is easier to read when split over three lines:

```
s
EEEEEEEEEEEEEEE
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

If we want to portray zero, it would have an exponent of zero, a significand of zero, and a sign of one or zero, depending on whether we want it to be positive or negative zero. In binary, positive zero is:
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
... which is:
0x00000000000000000000
... in hex, or with the lower dwords separated by spaces to make it easier to read:
0x0000 00000000 00000000

In binary, negative zero is:
1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
... which is:
0x80000000000000000000
... in hex, or with the lower dwords separated by spaces to make it easier to read:
0x8000 00000000 00000000

With 16-bit, 32-bit and 64-bit floating-point numbers, infinity was represented by setting the exponent all to ones and the significand all to zeroes. With 80-bit floating-point numbers, the exponent is similarly set to all ones, but the significand is set to all zeroes *except for the leftmost bit,* which is set to 1. Therefore, in binary, positive infinity is:

0111 1111 1111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

In hex, this is:
0x7fff 80000000 00000000
... with spaces separating the lower dwords to make it easier to read.

In binary, negative infinity is:

1111 1111 1111 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

In hex, this is:
0xffff 80000000 00000000

It is worth noting that apart from when encoding zero and certain rare exceptions (denormal numbers, which I mention later), the leftmost bit of the significand will always be set to 1. This is because the implied 1 that is ignored in 16-bit, 32-bit and 64-bit floating-point numbers is kept in 80-bit floating-point numbers.

We will convert a binary number to an 80-bit floating-point number to see how it works. We will convert the following binary number:
0.0000 0000 0000 0000 0000 0000 1100 1100 0000 1100 1100 1111 1111

This can be rephrased as:
$1.1001\ 1000\ 0001\ 1001\ 1001\ 1111\ 111 * (10)^{-11001}$
... all in binary, where −11001 is −0x19 in hex and −25 in decimal.

The attributes of this number are:

- The sign is positive.

- The exponent is −11001. We have to add this to 011111111111111 for it to become our biased exponent. This is slightly easier to do by working in decimal, in which case we add −25 to 16,383, which is 16,358. This is 0x3fe6 in hex, and 011 1111 1110 0110 in binary.

- For 16-bit, 32-bit and 64-bit floating-point numbers, we would say that the significand is 1001 1000 0001 1001 1001 1111 111. However, for 80-bit floating-point numbers, we have to include the normally implied “1” as the leftmost bit. Therefore, our significand is actually all of that number preceded by 1, which is: 1100 1100 0000 1100 1100 1111 1111. We also have to extend the number to 64 digits by putting zeroes at the end.

Our template looks like this:

```
s
EEEEEEEEEEEEEEE
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

We set the sign to 0:

```
0
EEEEEEEEEEEEEEE
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

We set the exponent to 011111111100110:

```
0
011111111100110
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

We set the significand to 11001100000011001100111111111 followed by enough zeroes to make it 64 digits long.

```
0
011111111100110
1100110000001100110011111111100000000000000000000000000000000000000000
```

We then join up the pieces, and our final binary number (with spaces to make it easier to read) is:
0011 1111 1110 0110 1100 1100 0000 1100 1100 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000
... which, in hex, is:
0x3fe6 cc0ccff0 00000000
... with spaces separating the lower dwords to make it easier to read.

Denormalised numbers

The method of encoding numbers with the floating-point system described in this chapter refers to what are called “normalised” or “normal” numbers. This is sufficient for nearly all use of floating-point numbers. Normalised numbers are so-called mainly to distinguish them from “denormalised” or “denormal” numbers. We can think of “normalised” numbers as just being numbers as encoded with the floating-point system already described where:

- With 16-bit, 32-bit, and 64-bit floats, there is an implied initial 1 digit.
- With 80-bit floats, the 1 digit is included.

“Denormalised” or “denormal” numbers are ones that are so small that they are likely to be treated as zero because of the way that the floating-point system encodes numbers – the exponent cannot go high enough to accommodate them. This can be a problem if a calculation produces a very small number and the program needs to know that the result is not exactly zero. To avoid treating tiny numbers as zero, the way they are stored is temporarily altered to accommodate

them. This involves removing the idea of an implied 1 (or an included 1 for 80-bit floats) and having an implied *zero* instead (which is an included zero for 80-bit floats). On Intel processors, the switch to a denormalised number is flagged with a warning to tell a program that it has happened. As we have seen, 80-bit floating-point *normal* numbers have the usually implied 1 kept as the leftmost bit of the significand. If a number becomes too small for the exponential to encode it, the number becomes *denormal* and the leftmost bit of an 80-bit number becomes set to zero. In 16-bit, 32-bit and 64-bit floats, the zero becomes implied instead of the usual 1 being implied.

You do not need to understand denormalised numbers for the purposes of understanding anything in this book, or even for most programming. I only mention them here because other explanations of floating-point numbers might discuss them.

More on floating-point numbers

Here are some more facts about floating-point numbers.

There is an internationally agreed system for the encoding of binary numbers with the floating-point method. It is called “IEEE 754”. Intel and AMD processors follow this system.

Values stored as floating-point numbers are limited to those that have significant digits that can fit into the significand, and also those that are of a size that can be represented by the exponent. For most purposes, the floating-point number system is perfectly suitable. As long as a number has the basic layout of:
11100000000
... or:
0.0000000111
... it can be encoded without any trouble. Problems can arise if we have a number that is both large and requires accuracy, such as this one:
1110000000000.0000000000011
[Using signal processing language, we could say that we are limited by the dynamic range of the floating-point system.]

Floating-point numbers have an unavoidable limit to their accuracy. As more calculations are performed with a number, this can become a problem and rounding errors will start to produce increasingly inaccurate results. A consequence of this is that a calculation performed with an 80-bit float might

produce slightly different results from the same calculation performed with a 64-bit or 32-bit float. An 80-bit float will be slightly more accurate. For 80-bit, 64-bit and 32-bit floats, any accumulated inaccuracies will be different, depending on which we are using.

The significand is sometimes called the “mantissa”. “Mantissa” is a Latin word that means a small amount of something used to top up a quantity to achieve a certain weight. For example, if we added a small amount of flour to a bag of flour to make it up to a 1 kilogramme, then, in the Latin sense, that small amount of flour would be a mantissa. The word has a related sense as “a worthless addition”. It is used in maths with the idea that the part after the binary (or decimal) point is not as important as the part before. Etymologically, the term “significand” really means the opposite of “mantissa” because it implies significance. The significand in a floating-point number is the most significant part. Some people prefer the word “significand”; some people prefer the word “mantissa”. In Intel’s “Intel 64 and IA-32 Architectures Software Developer’s Manual” (which is the definitive description of Intel’s processors), Intel uses the term “significand”.

As we have seen, floating-point numbers can store integers as well as non-integers. However, if we were only dealing in integers, it would be much easier not to use floating-point numbers and to use standard words, dwords and qwords instead.

Storing in memory

Floating-point numbers are stored in memory in the same way as bytes, words, dwords and qwords. In other words, they can be placed in the “backwards” little-endian way, or the more straightforward big-endian way. It is easiest to store them in files in the same way as they are stored in a computer’s memory.

A 16-bit half-precision floating-point number is a word, so it is stored as two bytes. If we store the 16-bit float 0x4a18 in the little-endian way, it would look like this in memory when viewed in a debugger or in a file as viewed in a hex editor: [I have arbitrarily added zeroes afterwards to fill a full line.]

```
0000: 18 4a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .J..............
```

In the big-endian way, it would look like this:

```
0000: 4a 18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 J...............
```

A 32-bit single-precision floating-point number is a dword, so it is stored as four bytes. If we stored the 32-bit float 0xc1507800 in the little-endian way, it would look like this in memory or a file:

```
0000: 00 78 50 c1 00 00 00 00 00 00 00 00 00 00 00 00 .xP.............
```

In the big-endian way, it would look like this:

```
0000: c1 50 78 00 00 00 00 00 00 00 00 00 00 00 00 00 .Px.............
```

A 64-bit double-precision floating-point number is a qword, so is stored as eight bytes. If we stored the 64-bit float 0x40c9e21e00000000 in the little-endian way, it would look like this in memory or a file:

```
0000: 00 00 00 00 1e e2 c9 40 00 00 00 00 00 00 00 00 .......@........
```

If we stored it in the big-endian way, it would look like this:

```
0000: 40 c9 e2 1e 00 00 00 00 00 00 00 00 00 00 00 00 @...............
```

An 80-bit double-extended-precision floating-point number is 80 bits long, so it takes up ten bytes. If we stored the 80-bit float 0x3fe6cc0ccff000000000 in the little-endian way, it would look like this in memory or a file:

```
0000: 00 00 00 00 f0 cf 0c cc e6 3f 00 00 00 00 00 00 .........?......
```

In the big-endian way, it would look like this:

```
0000: 3f e6 cc 0c cf f0 00 00 00 00 00 00 00 00 00 00 ?...............
```

Intel 64-bit processors (among other 64-bit processors) work fastest when reading numbers from memory if the numbers are aligned to an offset that is on a 64-bit (8-byte) boundary. In other words, they can read a number more quickly if the offset of the number in memory ends in 0 or 8. For example, it is quicker for the processor to read the 64-bit qword 0xffffffffffffffff in this bit of memory, where it starts on a 64-bit offset:

```
000000014000D000: ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00
```

... than it is to read it from this bit of memory, where it starts 1 byte after a 64-bit offset:

```
000000014000D000: 00 ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00
```

For this reason, in memory, numbers are frequently stored at 64-bit offsets, and if a number does not take up the full 64 bits, the bytes afterwards until the next 64-bit boundary become unused. They might be set to zero, they might be set to 0xcc, or they might be left as whatever they were before. [Setting them to 0xcc can be helpful for programmers trying to detect problems when debugging software – if the number 0xcc, 0xcc, 0xcccc or similar is read by the program when it should not be, then it is a sign that the program is reading from the wrong parts of memory. Similarly, if the processor tries to execute the command 0xcc, it will instantly pass control of the program to a debugger.] Aligning numbers to 64-bit boundaries means that in memory, the consecutive 32-bit unsigned integer dwords 0xffffffff and 0x22222222 would appear as so:

```
000000014000D000: ff ff ff ff 00 00 00 00 22 22 22 22 00 00 00 00
```

If speed is not of importance, then the numbers might appear normally as so:

```
000000014000D000: ff ff ff ff 22 22 22 22 00 00 00 00 00 00 00 00
```

The 64-bit alignment idea means that an 80-bit floating-point number, which takes up ten bytes, will often be followed by padding bytes so that the next number starts on a 64-bit offset. There will be 6 bytes of padding. Therefore, two consecutive 80-bit floating-point numbers might be stored in memory as follows: [Shown here with the padding bytes set to 0xcc to make it clearer as to what is a number and what is padding.]

```
000000014000d000: 00 30 1d df 27 58 c5 c5 00 40 cc cc cc cc cc cc
000000014000d010: 00 40 c7 f7 09 56 71 81 02 c0 cc cc cc cc cc cc
```

When storing the data into a *file*, there is less need to have padding, as the file will need to be loaded into memory before the processor can read it. Reading the data from a file into memory is so many times slower than reading from memory that there are no real *speed* advantages in having padding in a file. Therefore, in a file, the above numbers would be much more likely to appear consecutively as so:

```
0000: 00 30 1d df 27 58 c5 c5 00 40 00 40 c7 f7 09 56
0010: 71 81 02 c0 00 00 00 00 00 00 00 00 00 00 00 00
```

Sometimes, it is less effort from a programming point of view just to copy data directly from memory into a file, without looking at the data while it is being done. In such situations, a file will unnecessarily contain the padding. This is why sometimes when you look at a file in a hex editor, the numbers might appear differently from how you would expect.

Programming terms

In this chapter, we have looked at the different ways of storing binary and hex in a way that emphasises the number of bits in each number. Someone from Intel or an assembly language programmer would speak of the numbers in this way. Generally, in programming languages, the actual details of the numbers being used are hidden from the programmer. Therefore, the names that a C programmer, for example, would use can be different from those that an assembly language programmer would use. The terms used in higher-level programming tend to be much more vague in their meaning, and can even refer to different things depending on the type of processor on which the program is going to be run. When higher-level programmers discuss the types of numbers that are stored in a file, they might use higher-level programming terms instead of saying bytes, words, dwords, 80-bit floats, and so on. Therefore, it pays to know what they mean.

In this section, we will look at the more common programming terms used in C programming.

Char

The term “char” is short for “character”. A “char” is 8 bits long and is so-called due to how an ASCII character is 8 bits in length. A “char” is specifically a *signed* byte. Therefore, it can store positive and negative integers, but it can only store a positive integer up to +127.

A “uchar” is an unsigned byte, or what I would call a normal byte. This means it can store positive integers from 0 up to 255.

Int

The term “int” is short for “integer”, and specifically means a “*signed* integer”. This is an ambiguous term because it can refer to different lengths of numbers depending on the processor for which the program is being written. On a 16-bit processor, an “int” will be a 16-bit *signed* integer. For a 32-bit processor, an “int” will *usually* be a 32-bit signed dword, but not always. Similarly, for a 64-bit processor, an “int” will *usually* be a 64-bit signed qword, but not always. All of this means that if someone mentions the term “int”, its meaning is ambiguous without further information. Having a slightly ambiguous definition of an unsigned integer allows fairly simple code to be used on different types of computers without

needing to be rewritten. [It is not good if you want to tailor your code to take advantage of a particular type of processor or operating system.] The ambiguity is a nuisance when it comes to numbers stored in a file, as saying “this file contains a list of numbers as ints” is as useless as saying “this file contains numbers”. It is better either to refer to signed or unsigned bytes, words, dwords and qwords, or otherwise to specify the bit size of the int.

To have an unsigned integer, there is the term “uint”.

There is also the term “long”, which refers to a signed integer that is bigger than an “int”. This can also be ambiguous.

Float

A “float” is a 32-bit single-precision floating-point number. Outside of programming, it would be more descriptive to refer to floats as “32-bit floats” to indicate how many bits they use.

Double

A “double” is a 64-bit double-precision floating-point number. Outside of programming, it is more descriptive to refer to such numbers as “64-bit floats”, “64-bit doubles” or even “64-bit double floats”.

Long double

A “long double” is an 80-bit double-extended-precision floating-point number. Outside of programming, it is more descriptive to refer to such numbers as “80-bit floats”, “80-bit long doubles” or even “80-bit long double floats”. Some modern compilers will ignore any requests to use 80-bit long doubles, and replace them with 64-bit doubles instead. This means that there will be less accuracy, but the program might run faster because it can use a wider range of instructions to deal with the numbers.

Conclusion

This has been Chapter 40 of my book about waves. The rest of the book is available at www.timwarriner.com

It is easiest to understand and remember everything in this chapter if you have a need to do so. It is well worth downloading a free hex editor to get more used to binary and hexadecimal.