Decimal, Binary, Octal and Hex

Richard Han
richard.han@mq.edu.au

Revised 26 July 2021

Introduction

People use decimal numbers but computers work in binary. Programmers use hexadecimal (base 16) and octal (base 8) as a convenient representation of binary data. How to convert between these number representations is something that every systems programmer should know. Of course, there are calculators that can convert between decimal and hexadecimal, but systems programmers need to be able to look at hexadecimal data dumps (in particular) and be able to recognise the binary patterns in the data.

This document presents the arithmetic skills of converting between the different numbers bases that programmers use. This material is relevant to Data File Lab, and also to lectures in COMP2100.

Number bases

Decimal is convenient for us because we have ten fingers[1]. If we had only four fingers on each hand then we would probably live in an octal world where numbers are made of the digits 0-7.

Decimal

In our decimal number system, the digits of a number have place values. The bottom digit of an integer counts as units, the next digit as tens, then hundreds and so on. When we write "123" what we mean is: 3 units, 2 tens and 1 hundred. It can be written something like the following.

100	**10**	**1**
1	2	3

The value of each place is a power of ten – each place further to the left has a value ten times the place to its right. The value of this number can be computed (in decimal) by multiplying each digit by its place value and adding them up. Working from right to left (because that is easier in other number bases), we see that the value of the number is:

$$\begin{aligned}&3 * 1 + 2 * 10 + 1 * 100\\&= 3 + 20 + 100\\&= 123\end{aligned}$$

There is nothing surprising here. The number was originally decimal, we broke it down into the individual digits and place values, then recombined it into a decimal number. Our arithmetic would be faulty if the answer was not the same decimal number that we started with!

In the C programming language, numbers are normally written in decimal – special notations are used to indicate other number bases.

Octal

Other number bases work similarly to decimal, except that the value of each place is a power of the base of the new number system. For example, in an octal world, the values of the digits (from right to left) are 1, 8, 8^2, 8^3, and so on. In our decimal number system, 8^2 is 64, and 8^3 is 512. If we write

[1] Fingers including thumbs.

the octal number “0123” what we mean is 3 units, 2 eights and 1 sixty-four. So the breakdown of the octal number “0123” looks something like the following.

64	**8**	**1**
1	2	3

The meaning of “0123” as an octal number is calculated as follows, again working from right to left.

3 * 1 + 2 * 8 + 1 * 64
= 64 + 16 + 3
= 83

What we have just done is to convert the octal number into decimal by performing the place computation in decimal.

In the C programming language, we indicate that an integer is octal by prefixing the number with a zero digit (as in the example above). Octal numbers can only contain the digits 0-7, because 7 is the maximum place value.

Binary

Computers actually work in binary, so the most important number system to understand is binary. In binary, each digit (called a bit) is either 0 or 1, and the place values are powers of two. Because there are so few digits, binary numbers tend to be quite long – it takes 10 bits to represent the decimal number 999 and 20 bits to represent 999999 so we will typically work with numbers up to 8 bits in length. Here is one such number: “0b10110100”. The prefix “0b” indicates binary. The breakdown of this binary number into its digits looks like the following.

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	0	1	1	0	1	0	0

The meaning of this number is 0 units, 0 twos, 1 four, 0 eights, 1 sixteen, 1 thirty-two, 0 sixty-fours, and 1 one-hundred-and-twenty-eight. We can calculate the value as follows. Working from right to left is much easier on binary numbers because you can count off your powers of two as you read off the digits from right to left.

0 * 1 + 0 * 2 + 1 * 4 + 0 * 8 + 1 * 16 + 1 * 32 + 0 * 64 + 1 * 128
= 4 + 16 + 32 + 128
= 180

The first line of this calculation is actually unnecessary. Since we are multiplying the place values by digits that are either 0 or 1, it is just as easy to write down the simple summation on the second line – adding together the place values of all the 1 bits in the binary number. Because we are performing our calculations in decimal, the final result is that we have converted from binary to decimal.

Powers of 2 are really important in computer hardware, so they are really important for systems programmers. You should learn the powers of 2 at least up to 2^{10} = 1024: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. Memorise them.

The standard C programming language has no notation for binary numbers, although gcc has an extension that allows you to use the prefix “0b” to represent a binary number, as in the above example.

Hexadecimal

The hexadecimal number system is base 16 and it has 16 digits. Since we only have the digits 0-9 available as digit symbols in our writing, computer scientists use the letters ‘A’, ‘B’, … ‘F’ (or their lower-case equivalents ‘a’, ‘b’, … ‘f’) to represent digits ten, eleven, etc up to fifteen. The following table shows the hexadecimal digits and their decimal values.

Digit	**Value**	**Digit**	**Value**	**Digit**	**Value**	**Digit**	**Value**
0	0	4	4	8	8	C	12
1	1	5	5	9	9	D	13
2	2	6	6	A	10	E	14
3	3	7	7	B	11	F	15

The place values of hexadecimal digits are powers of 16. From right to left, they are units, sixteens, two-hundred-and-fifty-sixes, etc. For example, consider the hexadecimal number “0x25F”. The prefix “0x” indicates hexadecimal, and the number is interpreted as follows.

256	**16**	**1**
2	5	F

Remembering that F is 15, we can calculate the value of “0x25F” as:

15 * 1 + 5 * 16 + 2 * 256
= 15 + 80 + 512
= 607

The C programming language uses the prefix “0x” to indicate a hexadecimal number. You can write a hexadecimal constant anywhere that you would use an integer constant.

Exercise: As another example, the hexadecimal number “0x123” has the value 291. Calculate it and check your answer.

Converting Binary to/from Hexadecimal and Octal

Hexadecimal and octal are convenient because their number bases are powers of two. This means that the digits of hexadecimal and octal correspond to groups of binary digits. In particular, each hexadecimal digit is equivalent to 4 binary digits while each octal digit is equivalent to 3 binary digits.

The following tables shows the equivalence between the hexadecimal number “0x25F” and the binary number “0b001001011111”.

Firstly, we group the binary digits into groups of four as “0x 0010 0101 1111” to make it easier to convert to hexadecimal. Each group of four bits corresponds to one hexadecimal digit. The equivalence of the two numbers can easily be seen by comparing each hexadecimal digit with the corresponding group of four bits.

Hex	2				5				F			
Bin	0	0	1	0	0	1	0	1	1	1	1	1

Exercise: You can also check the equivalence of these two numbers by computing their decimal values using the place values as shown in the following table.

Hex	**256**				**16**				**1**			
Hex	2				5				F			
Bin	**2048**	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Bin	0	0	1	0	0	1	0	1	1	1	1	1

Hexadecimal is particularly convenient because each hexadecimal digit represents 4 bits, so two hex digits represent an 8-bit byte. Four hex digits represent two bytes and you can convert a 16-bit integer from binary to hex by converting each byte separately and then joining them together in the correct sequence. The number of bits in primitive data types is almost always a power of 2 in modern computers, so hexadecimal digits are a good fit.

In general, you can convert a binary number to hexadecimal by grouping the binary bits into groups of 4 (from right to left) and then converting each group of 4 bits into a single hexadecimal digit. For example, here is the conversion of a large binary number.

Step	**Representation**
Original binary number	`0b101110110010111101`
Groups of four bits[2]	`0010 1110 1100 1011 1101`
Decimal value of each group[3]	2 14 12 11 13
Hexadecimal conversion of each group	`2 E C B D`

In contrast to hexadecimal, octal digits each represent three bits. A byte requires three octal digits, but it is not an exact fit and the top digit of an octal representation of a byte is always 0, 1, 2 or 3 because it represents only 2 bits. This means that you cannot easily combine the octal values of two bytes to compute the octal value of a 16-bit short integer. In contrast, the hexadecimal value of two bytes corresponds easily to the individual byte values. However, octal digits are much easier to recognise in patterns of three bits.

Here is an example of converting a byte from binary to octal.

Step	**Representation**
Original binary number	`0b10011101`
Groups of three bits[4]	`010 011 101`
Octal conversion of each group	`2 3 5`

A hexadecimal number is easily converted to binary by expanding each hexadecimal digit into a group of 4 bits. Similarly, octal can be converted to binary by expanding each octal digit into a group of 3 bits. The following examples show the conversion of hexadecimal "0x2ECBD" and octal "0235" to the corresponding binary numbers. You should notice that these examples are the same as the above examples with the steps reversed.

[2] Note that the left-most group has two leading zeroes added to make it a group of four bits.
[3] When you become familiar with hexadecimal and binary conversion you will learn to recognize the binary patterns corresponding to each hex digit. But, if there is any doubt, mentally convert the 4 binary digits to a decimal number and then convert to hex to be sure.
[4] Note that the left-most group has a leading zero added to make it a group of three bits.

Step	**Representation**
Hexadecimal number	`2    E    C    B    D`
Decimal value of each digit	`2   14   12   11   13`
Four bit conversion of each digit	`0010 1110 1100 1011 1101`
Binary number	`00101110110010111101`

Step	**Representation**
Octal number	`2   3   5`
Three bit conversion of each digit	`010 011 101`
Binary number	`010011101`
Binary byte – drop extra leading 0 bit	`10011101`

Converting Decimal to Binary

We have seen above how to convert from binary to decimal by using the values of the bit places. Converting from decimal to binary is a little more difficult because we do our calculations in decimal, and we need to work out the binary digits. The technique that we present here is repeated division by 2. It is worth learning this technique and using it whenever you convert decimal to binary.

Warning: Some people convert decimal to binary by guessing the powers of 2. This is an easy way to make mistakes that are costly. For reliable results, use the repeated division technique, then you can use the powers of 2 place values to convert the binary back to decimal and check your calculations.

The repeated division technique is as follows.

1. Divide the number by 2, writing the quotient below and the remainder (0 or 1) to the right.
2. Repeat step 1 until the quotient is zero.
3. Read the binary number by reading the remainders from bottom to top.

The following is an example of converting decimal 123 to binary. We write the number, draw a line under it and write the quotient after division by 2 below the line with the remainder to the right. So, 123 divided by 2 is 61 and a remainder of 1; 61 divided by 2 is 30 and a remainder of 1; 30 divided by 2 is 15 and a remainder of zero; and so on until the result of the division is 0.

```
123
61     1
30     1
15     0
7      1
3      1
1      1
0      1
```

Reading the remainders from bottom to top we get the binary number 0b1111011. Using place values, this number has the value 1 + 2 + 8 + 16 + 32 + 64 = 123.

Exercise: Convert the decimal number 75 to binary. The answer should be 0b1001011

Converting Decimal to Octal and Hexadecimal

The easiest way to convert decimal to octal or hexadecimal is by converting the decimal to binary and then grouping the bits to convert from binary to octal or hexadecimal. This is the recommended approach[5].

> **Exercise**: Convert the decimal number 157 to binary, then to octal and hexadecimal. The octal answer should be 0235 and the hexadecimal answer is 0x9D.

Some Well Known Numbers

You should know the powers of 2 up to 2^{10} = 1024. They are:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

You should also be familiar some other numbers that occur commonly in computer systems and various application areas of IT.

Decimal Number	Importance
255	This is the largest unsigned number that fits in a single byte – it is a byte full of 1 bits. In networking, 255 is commonly found in IPv4 subnet masks.
65535	This is the largest unsigned number that fits in 16 bits – two bytes full of 1 bits. It is what is meant by "64K".

It is also important to recognise the digit F in hexadecimal as 4 bits that are all 1-bits and the digit 7 in octal as 3 bits that are all 1-bits. The important numbers listed above have very noticeable binary, hexadecimal and octal representations. In fact, the reason that these two numbers are so important is their binary representation.

Decimal	Binary	Hexadecimal	Octal
255	0b 1111 1111	0xFF	0377
65535	0b 1111 1111 1111 1111	0xFFFF	0177777

We will encounter other significant numbers (and approximations) in the lectures.

Two important properties of binary numbers are helpful to remember:

1. If the value is a power of two then the binary representation has exactly one 1-bit in the place representing that power of two. To be more mathematical, the binary representation of the number 2^n is a 1-bit followed by n 0-bits.
2. If the value is one less than a power of two then the binary representation is all 1-bits below the place of the power of two. To be more mathematical, the binary representation of the number 2^n-1 is n 1-bits.

[5] Repeated division by 8 can also be used to convert from decimal to octal directly, and repeated division by 16 would convert from decimal to hexadecimal, but division by 16 is probably more difficult for most people than repeatedly dividing by 2.

Exercise: Convert 4096 to binary, then to hexadecimal and octal. Check your answers by converting back. How does this number relate to the important properties listed above?
Exercise: Convert 511 to binary, then to hexadecimal and octal. Check your answers by converting back. How does this number relate to the important properties listed above?

Negative Numbers

In this section you will learn:

- How to compute the binary representation of a negative number
- How to negate a signed binary number
- How to recognise whether a signed binary integer is positive or negative

So far in this lab note we have focused on unsigned integers.

Aside: The C programming language has an explicit data types for unsigned integers of different sizes. The type `unsigned int` occupies the same amount of memory as a `signed int` (which we normally just call `int`) but none of the unsigned values are interpreted as negative – instead `unsigned int` supports larger positive numbers than can be stored in an `int`.
The type `unsigned short int` (or `unsigned short`) has values ranging from 0 to 65535 whereas `short int` (or short) has values ranging from -32768 to 32767. Both `unsigned short int` and `short int` occupy 2 bytes (16 bits) of memory, and have 2^{16} possible different values; the bits are just interpreted differently.

Signed integers need a special representation for negative numbers. The method of representing negative integers in computers is called 2's complement. This system is easy to implement in hardware, which is what makes it the standard approach. It is a little bit complicated to understand, but once you get it you won't find it difficult.

How to Negate a Number in 2's complement

There is a specific process for negating a number in 2's complement. The steps are as follows.

1. Start with a binary number.
2. Flip all the bits. i.e. Change every 0-bit to a 1-bit and every 1-bit to a 0-bit. Note: To do this, we need to know how many bits we are working with.
3. Add one to get the negated number. We need to perform binary addition.

Computing the Binary Representation of a Negative Number

Consider computing the binary representation of the decimal number -22. We first compute the binary representation of the positive decimal number 22, and then use the steps above to negate it.

Step	Explanation	Binary
1	Start with the positive number 22. Express it as binary in 8 bits.	`0001 0110`
2	Flip the bits	`1110 1001`
3	Add one. The result is the binary representation of -22 in 8 bits.	`1110 1010`

Notice that the highest bit of the negative number is a 1-bit. In signed computer integers, the number is negative if the highest bit is a 1-bit, and the number if positive if the highest bit is a 0-bit. This is why the highest bit is also called the sign bit.

Exercise: Convert the decimal number -11 to binary in eight bits. First convert decimal 11 to binary and then negate it. The answer should be `1111 0101`.

Negating a Negative Number

The process for negation of a negative number is actually the same, but starts with a negative binary number. Let's apply the process to the binary representation of -22 that we computed in the previous section.

Step	Explanation	Binary
1	Start with the negative number -22 expressed as binary in 8 bits.	`1110 1010`
2	Flip the bits	`0001 0101`
3	Add one. The result is the binary representation of 22 in 8 bits.	`0001 0110`

Exercise: The binary representation of -11 in eight bits is `1111 0101`. Negate this number to obtain the binary representation of 11 in eight bits. Check your answer by converting your binary result to decimal.

Recognising Bit Patterns of Positive and Negative Signed Integers

In signed integers, the highest bit is the sign bit. If the sign bit is 1, the signed integer represents a negative number. If the sign bit is 0, the signed integer is positive. We can recognise a signed integer as positive or negative by looking at the sign bit. However, we will also observe common patterns in both positive and negative signed integers.

Small Numbers in Eight Bits

We often represent relatively small integers in larger numbers of bits. In the example above, we represented the numbers 22 and -22 in 8 bits. Eight bits is enough to represent signed numbers ranging from -128 up to and including 127. So the number 22 is quite a small positive number in that range, while the number -22 is quite a small negative number.

Now look at the bit pattern of the positive number 22: "`0001 0110`". What do you notice about the high bits? The sign bit is zero, indicating that the number is positive; but actually the three highest bits are all zero. This is typical of positive integers – commonly, there and zeroes in the highest bits of the number.

What do you notice about the bit pattern of the negative number -22: "`1110 1010`"? The sign bit is one, but in fact the highest three bits are all 1-bits. This is typical of negative integers – we will often see 1-bits in the highest bit positions of the number.

Exercise: Consider the binary representations of 11 and -11, as in the previous exercise. How many high bits are zero in the representation of 11? How many high bits are one in the representation of -11? Both answers should be four.

Small Numbers in 16 Bits

Let's consider the numbers 22 and -22 expressed in 16 bits. These numbers were quite small in eight bits, but in sixteen bits they are much smaller relative to the range of 16-bit signed integers.

The binary representation of 22 in 16 bits is "`0000 0000 0001 0110`". You can easily prove to yourself that this is correct. But now, notice that this positive number has very many high 0-bits – in fact, the top 11 bits are all zero. This is typical of a small positive integer expressed in a large number of bits.

Now we compute -22 in sixteen bits by negating the 16-bit representation of 22.

Step	Explanation	Binary
1	Start with the positive number 22. Express it as binary in 8 bits.	`0000 0000 0001 0110`
2	Flip the bits	`1111 1111 1110 1001`
3	Add one. The result is the binary representation of -22 in 8 bits.	`1111 1111 1110 1010`

Here we can clearly see that the negative number ends up with a large sequence of high 1-bits – eleven high 1-bits including the sign bit. Again, this is typical of small negative integers expressed in a large number of bits.

Exercise: Convert the 16-bit binary representation of 22 to hexadecimal. What do you notice about the pattern of the hexadecimal number? Do the same thing for the 16-bit binary representation of -22. What do you notice now?

When bit patterns with many leading zeroes are converted to hexadecimal or octal, there are typically 0 digits at the high end of the converted numbers. Depending on how the number is formatted for printing, these leading zeroes may not be printed. It may be useful in some circumstances to ensure that they are printed by using appropriate `printf` format options.

When bit patterns with many leading ones are converted to hexadecimal there may be many leading F digits, each representing four 1-bits. Conversion to octal is not as simple, because octal uses groups of three bits.

Increasing the Word Size of a Signed Number

Suppose we have a signed number in eight bits and we want to store it into 16 bits? What can we do to ensure that the 16-bit value still represents the same number? Or suppose we want to convert up to 32 or even 64 bits?

Exercise: Carefully consider the previous section where we expressed both 22 and -22 in eight and then 16 bits. What do you notice about the relationships between the eight-bit representations and the corresponding 16-bit representations? How might you write a simple rule to convert from eight to 16 bits?

For the positive number, the larger representation has additional 0-bits extending it at the high end. On the other hand, the negative number has additional 1-bits at the high end. So, if the sign bit is zero we extend the number at the high end with 0-bits, and if the sign bit is one, we extend the number at the high end with 1-bits. More simply, we extend the signed number by inserting copies of the sign bit into the new higher bits.

This process of extending a signed number by replicating the sign bit is called "sign extension" of a number. It is easy to implement in hardware.

> **C Language:** C automatically applies sign extension to eight-bit and 16-bit integers when it passes them to a subroutine because the C specification says that anything smaller than `int` must be passed as `int` to the subroutine. Again, this is done for reasons of hardware efficiency.

> **Exercise**: Sign extend the binary representations of 22 and -22 from 8 bits to 32 bits. Then convert each of the results to hexadecimal. The answers should be `00 00 00 16` and `FF FF FF EA`.

Reducing the Word Size of a Signed Number

We can also reduce the word size of a number. For example, suppose we have the 16-bit binary number "`1111 1111 1110 1010`". What is the eight bit equivalent number? It is easy to see from the way we obtained this sixteen bit number that the eight-bit equivalent is the low eight bits: "`1110 1010`". This is a general principle, and is easily implemented in hardware: To convert a number to fewer bits, just keep the low bits.

Unfortunately, large numbers of bits can represent numbers that cannot be represented in fewer bits. For example, the number 1000 cannot be represented in 8 bits. If we start with the 16-bit representation of 1000 and convert to eight bits, what will we get? It turns out that we get whatever is in the low 8 bits of the binary representation of 1000. It won't be an eight bit representation of 1000, because it cannot be. What it will be, you can work out for yourself.

> **Exercise**: What is the very minimum number of bits that can represent the signed numbers 22 and -22? You need to keep the sign bit, but you can throw away any high bits that replicate the sign bit. The answer is more than 5 and less than 7.

> **Exercise**: Consider the 32-bit signed number -22 which is written in hexadecimal as `FF FF FF EA`. Write this same number in 16 bits and eight bits directly from the hexadecimal. Check your answer by considering the binary.

Signed and Unsigned Interpretations of the Same Bit Patterns

The computer hardware does not know whether a number is signed or unsigned. It just works on bits. It is worth noting that the same bit patterns can be interpreted as unsigned or signed numbers.

First consider a signed positive number, such as decimal 22 in eight bits: "`0001 0110`". If this number is decoded as an unsigned number, what is the result? It is, in fact, 22.

Exercise: Prove to yourself that `0001 0110` as unsigned binary decodes to 22.

However, when we consider a signed negative number, the unsigned interpretation of the bits is not the same. In one sense, this is to be expected because unsigned numbers cannot be negative.

Consider the signed negative number decimal -22 expressed as signed binary in eight bits: "`1110 1010`". What is this number interpreted as an unsigned number? i.e. Use the bit values of eight-bit numbers that we talked about on page 2 of this document to convert the number to decimal.

Exercise: Convert the unsigned binary number `1110 1010` to decimal. The answer should be 234.

The key observation here is that signed negative numbers become very large positive numbers if they are interpreted as unsigned numbers. Conversely, very large unsigned numbers would become negative numbers if they were interpreted as signed numbers.

Since the computer does not know whether a number is signed or unsigned, the C compiler gives you some help with type naming of variables. However, it turns out to be quite easy to deliberately or even accidentally fool the C compiler and cause it to misinterpret an unsigned integer as signed or vice-versa.

Conclusion

In this document, we have learned how to convert decimal numbers to binary and then to hexadecimal, and octal. We have also studied the reverse conversions back to decimal. We have learned how negative numbers are represented using 2's complement, and how to negate a number. This means we can compute the signed binary representation, and the corresponding hexadecimal, of a negative number. We learned to recognise bit patterns of small signed numbers in larger words. Also, we learned how to extend and reduce the word size of a signed number. Finally, we considered what happens if a signed integer is interpreted as unsigned, or an unsigned integer is interpreted as signed.

Len Hamey 2020