

You all know that computers represent numbers using 0's and 1's. But how exactly this works is probably a mystery for you. We start out the course with a basic example (which will come up again and again), namely how to represent integers using 0's and 1's.

Decimal vs. binary representation of positive integers

Up to now in our lives, you've represented numbers using a decimal representation (the ten digits from 0,1, ... 9). The reason 10 is special is that we have ten fingers. There is no other reason for this. There is nothing special otherwise about the number ten.

Computers don't represent numbers using decimal. Instead, they represent numbers using binary. All the algorithms you learned in grade school for addition, subtraction, multiplication and division work for binary as well. Before we review these algorithms, let's make sure we understand what binary representations of numbers are. We'll start with positive integers.

In decimal, we write numbers using *digits* $\{0, 1, \dots, 9\}$, in particular, as sums of powers of ten. For example,

$$238_{ten} = 2 * 10^2 + 3 * 10^1 + 8 * 10^0$$

In binary, we represent numbers using *bits* $\{0, 1\}$, in particular, as a sum of powers of two:

$$11010_{two} = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

I have put little subscripts (*ten* and *two*) to indicate that we are using a particular representation (decimal or binary). We don't need to always put this subscript in, but sometimes it helps to remind us of what base we are using. For example, 11010_{two} is not the same thing as 11010_{ten} .

It is trivial to write a decimal number as a sum of powers of ten and it is also trivial to write a binary number as a sum of powers of two, namely, just as I did above.

To *convert* from a binary number to a decimal number, you need to write the powers of 2 as decimal numbers and then add up these decimal numbers.

$$11010_{two} = 16 + 8 + 2$$

To do so, you need to memorize the powers of 2

$$2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32, 2^6 = 64, 2^7 = 128, 2^8 = 256, 2^9 = 512, 2^{10} = 1024, \dots$$

How do you convert from a decimal number to a binary number? On the next page, I show a simple algorithm for doing so, and an example. The algorithm repeatedly divides the decimal number by 2, and concatenates the remainders.

Why does this work? For any positive number m , let

$$m = \lfloor m/2 \rfloor * 2 + (m \bmod 2)$$

where $\lfloor m/2 \rfloor$ is the *quotient* and $\lfloor \ \rfloor$ rounds down (floor). Let $(m \bmod 2)$ is the *remainder*, i.e. division mod 2. (In the lecture slides, this was written with slightly different notation – you should be familiar with both.)

When m is represented as binary number (i.e. “base 2”), the quotient and remainder are trivial to obtain. The remainder is the rightmost bit – called the “least significant bit” (LSB). The quotient

is the number with the LSB chopped off. To convince yourself of this, note that writing a positive integer as an n bit binary number means that you write it as a sum of powers of 2,

$$(b_{n-1} b_{n-2} \dots b_2 b_1 b_0)_{two} \equiv \sum_{i=0}^{n-1} b_i 2^i = \sum_{i=1}^{n-1} b_i 2^i + b_0 = 2 \times \sum_{i=0}^{n-2} b_{i+1} 2^i + b_0$$

Here now is the algorithm:

Algorithm 1 Convert decimal to binary

INPUT: a number m expressed in base 10 (decimal)

OUTPUT: the number m expressed in base 2 using a bit array $b[]$

```

i ← 0
while m > 0 do
  bi ← m%2
  m ← m/2
  i ← i + 1
end while

```

For example,

quotient	remainder	interpretation (not part of algorithm)
241		
120	1	241 = 120*2 + 1
60	0	120 = 60*2 + 0
30	0	60 = 30*2 + 0
15	0	30 = 15*2 + 0
7	1	15 = 7*2 + 1
3	1	7 = 3*2 + 1
1	1	3 = 1*2 + 1
0	1	1 = 0*2 + 1
0	0	0 = 0*2 + 0
0	0	
:	:	

Thus,

$$241_{ten} = 11110001_{two}.$$

You need to be able to do this for yourself. You will be asked to do so in the midterm exam.

If you are not fully clear why the algorithm is correct, consider what happens when you run the algorithm *where you already have the number in binary representation*. (The quotient and remainder of a division by 2 does not depend on how you have represented the number i.e. whether it is represented in decimal or binary.)

(quotient)	(remainder)
<u>11110001</u>	
1111000	1
111100	0
11110	0
1111	0
111	1
11	1
1	1
0	1
0	0
0	0
:	:

The remainders are simply the bits used in the binary representation of the number!

Final note: The representation has an infinite number of 0's on the left which can be truncated. I mention this because (in the heat of an exam) you might get confused about which way to order the 0's and 1's. Remembering that you have infinitely many 0's when you continue to apply the trick. This tells you that the remainder bits go from right to left.

Performing addition with positive binary numbers

Doing addition with positive integers is as easy with the binary representation as it is with the decimal representation. Let's assume an eight bit representation and compute $26 + 27$.

00110100	← carry bits (ignore the 0's for now)
00011010	← 26
+ <u>00011011</u>	← 27
00110101	← 53

Its the same algorithm that you use with decimal, except that you are only allowed 0's and 1's. Whenever the sum in a column is 2 or more, you carry to the next column:

$$2^i + 2^i = 2^{i+1}$$

We would like to apply the grade school algorithm to do subtraction, multiplication and long division . There are some subtleties in doing so. For example, in subtraction, when you subtract a big positive number from a small positive number, you end up with a negative number, and I have not yet told you how to represent negative numbers. Indeed, the grade school algorithm for subtraction doesn't work, if you try subtracting a bigger number from a smaller algorithm, as I mentioned in class.

“Twos complement” representation of integers

To motivate our representation of negative numbers, let me take you back to your childhood again. Remember driving in your parent’s car and looking at the odometer. Remember the excitement you felt when the odometer turned from 009999 to 010000. Even more exciting was to see the odometer go from 999999 to 000000, if you were so lucky. This odometer model turns out to be the key to how computers represent negative numbers.

If we are working with six digit decimal numbers only, then we might say that 999999 behaves the same as -1. The reason is that if we add 1 to 999999 then we get 000000 which is zero.

$$\begin{array}{r} 999999 \\ + \quad \underline{000001} \\ 000000 \end{array}$$

Notice that we are ignoring a seventh digit in the sum (a carry over). We do so because we are restricting ourselves to six digits.

Take another six digit number 328769. Let’s look for the number that, when added to 328769, gives us 000000. By inspection, we can determine this number to be 671231.

$$\begin{array}{r} 328769 \\ + \quad \underline{651231} \\ 000000 \end{array}$$

Thus, on a six digit odometer, 651231 behaves the same as -328769.

Let’s apply the same idea to binary representations of integers. Consider eight bit numbers. Let’s look at $26_{ten} = 00011010_{two}$. How do we represent -26_{ten} ?

$$\begin{array}{r} 00011010 \\ + \quad \underline{????????} \\ 00000000 \end{array}$$

To find -26 , we need to find the number that, when added to 00011010 gives us 00000000. We use the following trick. If we “invert” each of the bits of 26 and add it to 26, we get

$$\begin{array}{r} 00011010 \quad \leftarrow 26 \\ + \quad \underline{11100101} \quad \leftarrow \text{inverted bits} \\ 11111111 \end{array}$$

This is not quite right, but its close. We just need to add 1 more. (Remember the odometer).

$$\begin{array}{r} 11111111 \\ + \quad \underline{00000001} \\ 00000000 \end{array}$$

Note we have thrown away the ninth bit because we are restricting ourself to eight bits only. Thus,

$$\begin{array}{r} 00011010 \quad \leftarrow 26 \\ 11100101 \quad \leftarrow \text{inverted bits} \\ + \quad \underline{00000001} \quad \leftarrow +1 \\ 00000000 \end{array}$$

Alternatively, we add 1 to the inverted bit representation and this must give us -26 .

$$\begin{array}{r}
 11100101 \quad \leftarrow \text{inverted bits} \\
 + \quad \underline{00000001} \quad \leftarrow +1 \\
 \hline
 11100110 \\
 \\
 00011010 \\
 \underline{11100110} \quad \leftarrow \text{This is } -26_{10}. \\
 \hline
 00000000
 \end{array}$$

Thus, *to represent the negative of a binary number, we invert each of the bits and then add 1*. This is called the *twos complement* representation.

Tricky cases

One tricky case is to check is that the twos complement of 00000000 is indeed 00000000.

$$\begin{array}{r}
 00000000 \\
 \underline{11111111} \quad \leftarrow \text{invert bits} \\
 \hline
 11111111
 \end{array}$$

And adding 1 gets us back to zero. This makes sense, since $-0 = 0$.

Another tricky case is the decimal number 128, and again we assume a 8 bit representation. If you write 128 in 8-bit binary, you get 10000000. Take the twos complement gives you back the same number 10000000.

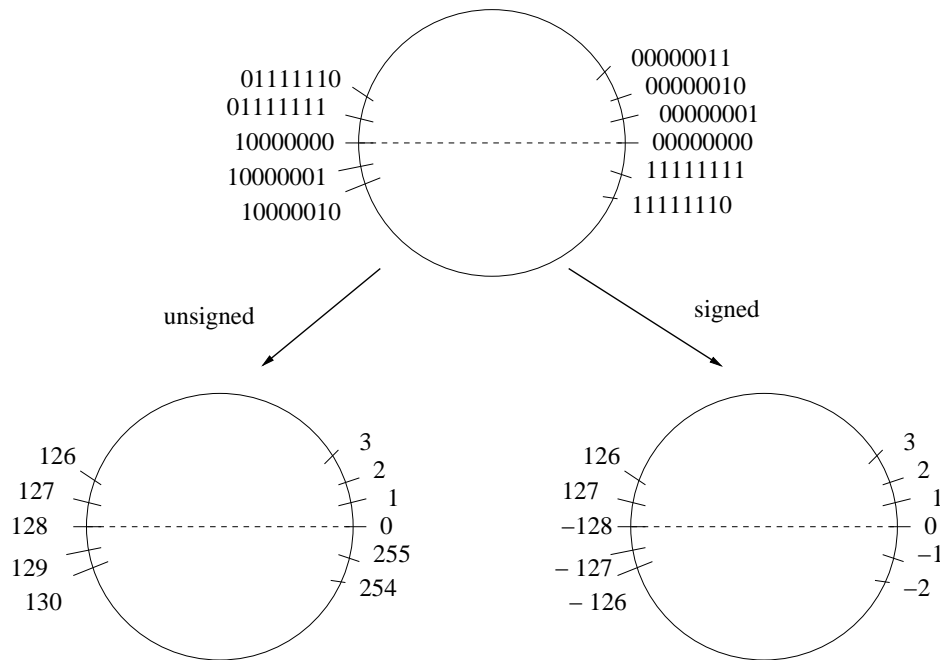
$$\begin{array}{r}
 10000000 \\
 \underline{01111111} \quad \leftarrow \text{invert bits} \\
 \hline
 11111111
 \end{array}$$

And adding 1 gets us back to zero. This implies that 128 is the same as -128 , which is strange at first. To understand what's going on here, you need to think about these numbers as living on a circle, rather than a line.

Unsigned vs. signed numbers

If treat all the numbers as positive, but we ignore the carry of the leftmost bit in our sum (the *most significant bit, or MSB*), then adding 1 to the binary number 11111111 (which is 255 in decimal) takes us back to 0. See the figure below on the left. This representation is called *unsigned*. Unsigned numbers are interpreted as positive.

To allow for negative numbers, we use the twos complement representation. Then we have the situation of the circle on the right. This is called the *signed* number representation. *Note that the MSB indicates the sign of the number. If the MSB is 0, then the number is non-negative. If the MSB is 1, then the number is negative.*



Unsigned and signed n-bit numbers

The set of *unsigned* n -bit numbers is represented on a circle with 2^n steps. The numbers are $\{ 0, 1, 2, \dots, 2^n - 1 \}$. It is common to use $n = 16, 32, 64$ or 128 , though any value of n is possible. The *signed* n bit numbers are represented on a circle with 2^n steps, and these numbers are $\{-2^{n-1}, \dots, 0, 1, 2, \dots, 2^{n-1} - 1\}$. Signed n bit numbers are represented using twos complement. For example, if $n=8$, then the signed numbers are $\{-128, -127, \dots, 0, 1, 2, \dots, 127\}$ as we saw earlier. Consider the following table for the 8 bit numbers.

<u>binary</u>	<u>signed</u>	<u>unsigned</u>
00000000	0	0
00000001	1	1
:	:	:
01111111	127	127
10000000	-128	128
10000001	-127	129
:	:	:
11111111	-1	255

If $n=16$, the corresponding table is:

<u>binary</u>	<u>signed</u>	<u>unsigned</u>
0000000000000000	0	0
0000000000000001	1	1
:	:	:

0000000001111111	127	127
0000000010000000	128	128
0000000010000001	129	129
:	:	
0111111111111111	$2^{15} - 1$	$2^{15} - 1$
1000000000000000	-2^{15}	2^{15}
1000000000000001	$-2^{15} + 1$	$2^{15} + 1$
:	:	
1111111101111111	-129	$2^{16} - 129$
1111111110000000	-128	$2^{16} - 128$
1111111110000001	-127	$2^{16} - 127$
:	:	
1111111111111111	-1	$2^{16} - 1$

Hexadecimal representation of binary strings

When we write down binary numbers with lots of bits, we can quickly get lost. No one wants to look at 16 bit strings, and certainly not at 32 bit strings. Yet we will need to represent such bit strings often. What can we do?

The most common solution is to group bits into 4-tuples ($2^4 = 16$) starting at rightmost bit (i.e. least significant bit). Each 4-bit group can code 16 combinations and we typically write them down as: 0, 1, ..., 9, a, b, c, d, e, f. a represents 1010, b represents 1011, c represents 1100, ..., f represents 1111.

You may find yourself saying a represents the decimal number 10 and is written in 4-bit binary as 1010, b represents the decimal number 11 and is written in 4-bit binary as 1011, ..., f represents the decimal number 15 and is written in 4-bit binary as 1111. If you do this, then you are first converting 4-bit binary to decimal, and then converting decimal to hexadecimal.

We commonly write hexadecimal numbers as $0x______$ where the underline is filled with characters from 0, ..., 9, a, b, c, d, e, f. For example,

$$0x2fa3 = 0010\ 1111\ 1010\ 0011.$$

Sometimes hexadecimal numbers are written with capital letters. In that case, a large X is used as in the example **0X2FA3**.

If the number of bits is not a multiple of 4, then you group starting at the rightmost bit (the least significant bit). For example, if we have six bits string 101011, then we represent it as **0x2b**.