O.4 — Converting between binary and decimal

ALEX AUGUST 2, 2021

Consider a normal decimal number, such as 5623. We intuitively understand that these digits mean (5 * 1000) + (6 * 100) + (2 * 10) + (3 * 1). Because there are 10 decimal numbers, the value of each subsequent digit to the left increases by a factor of 10.

Binary numbers work the same way, except because there are only 2 binary digits (0 and 1), the value of each digit increases by a factor of 2. Just like commas are often used to make a large decimal number easy to read (e.g. 1,427,435), we often write binary numbers in groups of 4 bits to make them easier to read (e.g. 1101 0101).

The following table counts to 15 in decimal and binary:

Decimal Value	Binary Value
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

Converting binary to decimal

In the following examples, we assume that we're dealing with unsigned integers.

Consider the 8 bit (1 byte) binary number 0101 1110. Binary 0101 1110 means (0 * 128) + (1 * 64) + (0 * 32) + (1 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1). If we sum up all of these parts, we get the decimal number 64 + 16 + 8 + 4 + 2 = 94.

Here is the same process in table format. We multiply each binary digit by its digit value (determined by its position). Summing up all these values gives us the total.

Converting 0101 1110 to decimal:

Binary digit	0	1	0	1	1	1	1	0
* Digit value	128	64	32	16	8	4	2	1
= Total (94)	0	64	0	16	8	4	2	0

Let's convert 1001 0111 to decimal:

Binary digit	1	0	0	1	0	1	1	1
* Digit value	128	64	32	16	8	4	2	1
= Total (151)	128	0	0	16	0	4	2	1

1001 0111 binary = 151 in decimal.

This can easily be extended to 16 or 32 bit binary numbers simply by adding more columns. Note that it's easiest to start on the right end, and work your way left, multiplying the digit value by 2 as you go.

Method 1 for converting decimal to binary

Converting from decimal to binary is a little more tricky, but still pretty straightforward. There are two good methods to do this.

The first method involves continually dividing by 2, and writing down the remainders. The binary number is constructed at the end from the remainders, from the bottom up.

Converting 148 from decimal to binary (using r to denote a remainder):

148 / 2 = 74 r0
74 / 2 = 37 r0
37 / 2 = 18 r1
18 / 2 = 9 r0
9 / 2 = 4 r1
4 / 2 = 2 r0
2 / 2 = 1 r0

1 / 2 = 0 r1

Writing all of the remainders from the bottom up: 1001 0100

148 decimal = 1001 0100 binary.

You can verify this answer by converting the binary back to decimal:

(1 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (0 * 8) + (1 * 4) + (0 * 2) + (0 * 1) = 148

Method 2 for converting decimal to binary

The second method involves working backwards to figure out what each of the bits must be. This method can be easier with small binary numbers.

Consider the decimal number 148 again. What's the largest power of 2 that's smaller than 148? 128, so we'll start there.

Is 148 >= 128? Yes, so the 128 bit must be 1. 148 - 128 = 20, which means we need to find bits worth 20 more.
Is 20 >= 64? No, so the 64 bit must be 0.
Is 20 >= 32? No, so the 32 bit must be 0.
Is 20 >= 16? Yes, so the 16 bit must be 1. 20 - 16 = 4, which means we need to find bits worth 4 more.

Is 4 >= 8? No, so the 8 bit must be 0.
Is 4 >= 4? Yes, so the 4 bit must be 1. 4 - 4 = 0, which means all the rest of the bits must be 0.

148 = (1 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (0 * 8) + (1 * 4) + (0 * 2) + (0 * 1) = 1001 0100

In table format:

Binary number	1	0	0	1	0	1	0	0
* Digit value	128	64	32	16	8	4	2	1
= Total (148)	128	0	0	16	0	4	0	0

Another example

Let's convert 117 to binary using method 1:

117 / 2 = 58 r1
58 / 2 = 29 r0
29 / 2 = 14 r1
14 / 2 = 7 r0
7 / 2 = 3 r1
3 / 2 = 1 r1
1 / 2 = 0 r1

Constructing the number from the remainders from the bottom up, 117 = 111 0101 binary

And using method 2:

The largest power of 2 less than 117 is 64.

Is 117 >= 64? Yes, so the 64 bit must be 1. 117 - 64 = 53.
Is 53 >= 32? Yes, so the 32 bit must be 1. 53 - 32 = 21.
Is 21 >= 16? Yes, so the 16 bit must be 1. 21 - 16 = 5.

Is 5 >= 8? No, so the 8 bit must be 0.
Is 5 >= 4? Yes, so the 4 bit must be 1. 5 - 4 = 1.
Is 1 >= 2? No, so the 2 bit must be 0.
Is 1 >= 1? Yes, so the 1 bit must be 1.

117 decimal = 111 0101 binary.

Adding in binary

In some cases (we'll see one in just a moment), it's useful to be able to add two binary numbers. Adding binary numbers is surprisingly easy (maybe even easier than adding decimal numbers), although it may seem odd at first because you're not used to it.

Consider two small binary numbers:
0110 (6 in decimal) +
0111 (7 in decimal)

Let's add these. First, line them up, as we have above. Then, starting from the right and working left, we add each column of digits, just like we do in a decimal number. However, because a binary digit can only be a 0 or a 1, there are only 4 possibilities:

- **0 + 0 = 0**
- **0 + 1 = 1**
- **1 + 0 = 1**
- **1 + 1 = 0, carry a 1 over to the next column**

Let's do the first column:

```
0110 (6 in decimal) +
0111 (7 in decimal)
----
   1
```

0 + 1 = 1. Easy.

Second column:

```
 1
0110 (6 in decimal) +
0111 (7 in decimal)
----
  01
```

1 + 1 = 0, with a carried one into the next column

Third column:

```
11
0110 (6 in decimal) +
0111 (7 in decimal)
----
 101
```

This one is a little trickier. Normally, 1 + 1 = 0, with a carried one into the next column. However, we already have a 1 carried from the previous column, so we need to add 1. Thus, we end up with a 1 in this column, with a 1 carried over to the next column

Last column:

```
11
0110 (6 in decimal) +
0111 (7 in decimal)
----
1101
```

0 + 0 = 0, but there's a carried 1, so we add 1. 1101 = 13 in decimal.

Now, how do we add 1 to any given binary number (such as 1011 0011)? The same as above, only the bottom number is binary 1.

```
       1  (carry column)
1011 0011 (original binary number)
0000 0001 (1 in binary)
---------
1011 0100
```

Signed numbers and two's complement

In the above examples, we've dealt solely with unsigned integers. In this section, we'll take a look at how signed numbers (which can be negative) are dealt with.

Signed integers are typically stored using a method known astwo's complement. In two's complement, the leftmost (most significant) bit is used as the sign bit. A 0 sign bit means the number is positive, and a 1 sign bit means the number is negative.

Positive signed numbers are represented in binary just like positive unsigned numbers (with the sign bit set to 0).

Negative signed numbers are represented in binary as the bitwise inverse of the positive number, plus 1.

Converting decimal to binary (two's complement)

For example, here's how we represent -5 in binary two's complement:

First we figure out the binary representation for 5: 0000 0101
Then we invert all of the bits: 1111 1010
Then we add 1: 1111 1011

Converting -76 to binary:

Positive 76 in binary: 0100 1100
Invert all the bits: 1011 0011
Add 1: 1011 0100

Why do we add 1? Consider the number 0. If a negative value was simply represented as the inverse of the positive number, 0 would have two representations: 0000 0000 (positive zero) and 1111 1111 (negative zero). By adding 1, 1111 1111 intentionally overflows and becomes 0000 0000. This prevents 0 from having two representations, and simplifies some of the internal logic needed to do arithmetic with negative numbers.

Converting binary (two's complement) to decimal

To convert a two's complement binary number back into decimal, first look at the sign bit.

If the sign bit is 0, just convert the number as shown for unsigned numbers above.

If the sign bit is 1, then we invert the bits, add 1, then convert to decimal, then make that decimal number negative (because the sign bit was originally negative).

For example, to convert 1001 1110 from two's complement into a decimal number:
Given: 1001 1110
Invert the bits: 0110 0001
Add 1: 0110 0010
Convert to decimal: (0 * 128) + (1 * 64) + (1 * 32) + (0 * 16) + (0 * 8) + (0 * 4) + (1 * 2) + (0 * 1) = 64 + 32 + 2 = 98
Since the original sign bit was negative, the final value is -98.

Why types matter

Consider the binary value 1011 0100. What value does this represent? You'd probably say 180, and if this were a standard unsigned binary number, you'd be right.

However, if this value was stored using two's complement, it would be -76.

And if the value were encoded some other way, it could be something else entirely.

So how does C++ know whether to print a variable containing binary 1011 0100 as 180 or -76?

In case the section title didn't give it away, this is where types come into play. The type of the variable determines both how a variable's value is encoded into binary, and decoded back into a value. So if the variable type was an unsigned integer, it would know that 1011 0100 was standard binary, and should be printed as 180. If the variable was a signed integer, it would know that 1011 0100 was encoded using two's complement (now guaranteed as of C++20), and should be printed as -76.

What about converting floating point numbers from/to binary?

How floating point numbers get converted from/to binary is quite a bit more complicated, and not something you're likely to ever need to know. However, if you're curious, see this site, which does a good job of explaining the topic in detail.

Quiz time

Question #1

Convert 0100 1101 to decimal.

Show Solution

Question #2

Convert 93 to an 8-bit unsigned binary number. Use both methods above.

Show Solution

Question #3

Convert -93 to an 8-bit signed binary number (using two's complement).

Show Solution

Question #4

Convert 1010 0010 to an unsigned decimal number.

Show Solution

Question #5

Convert 1010 0010 to a signed decimal number (assume two's complement).

Show Solution

Question #6

Write a program that asks the user to input a number between 0 and 255. Print this number as an 8-bit binary number (of the form #### ####). Don't use any bitwise operators. Don't use `std::bitset`.

Show Hint

Show Hint

Show Solution

Next lesson

6.1 Compound statements (blocks)

Back to table of contents

Previous lesson

O.3 Bit manipulation with bitwise operators and bit masks

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest