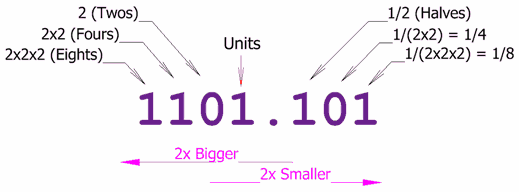
# Real numbers



[!NOTE] The purpose of this section is to …

* Know how numbers with a fractional part can be represented:
  + in fixed point form in binary in a given number of bits
  + in floating point form in binary in a given number of bits
* Be able to convert for each representation form:
  + decimal to binary of a given number of bits
  + binary to decimal of a given number of bits
* Know why floating point numbers are normalised and be able to normalise un-normalised floating point numbers with positive or negative mantissas.

The next stage is to consider those numbers with a fractional part, i.e. , etc..

## Fixed point

When representing numbers with a fractional part we make use of an additional symbol, the decimal point, which we do not have in a digital system of ‘0s’ and ‘1s’. Most modern systems will have native support through hardware for floating-point numbers (see next section) but sometimes fixed point is all that is needed as it is much faster than floating-point.

To represent binary numbers with a fractional part we need a **binary point** to separate the whole number part from the fractional part and to know how many bits to allocate for each part. Let’s assume, 8 bits, with 5 bits allocated for the whole part and to help with a later point we’ll start by converting the whole number 59 into binary:

Dividing by results in , which in binary fixed point representation is:

It should be noted that the bit pattern for each of these numbers is identical, the only difference being the position of the binary point.

The positional weights for the columns after the binary point are , , , and so on.

Fixed point numbers, signed or unsigned, are very closely related to whole number representation the only difference being the position of the binary point. All arithmetic operations carried out on whole numbers can be used to apply to real numbers using a fixed point representation. Thus, no additional circuitry is needed unlike floating point and they will perform better. The downside is the limitation on the number of bits available to represent the number.

### Converting decimal to fixed point binary

For example, the decimal value .

* Convert the whole number part, , to binary using the repeated division by two algorithm
* Convert the fractional part, , to binary using a repeated multiplication by two taking the integer part of the result (which will be either or , as the first bit and repeat until you’ve exhausted the number of bits available or hit .

Thus:

* is

is equivalent to

NB. With a fixed number of bits it may not be possible to represent the number precisely, for example , taking the part:

And, now we’ve run out of bits! So, the closest we can get with bits available is which is equivalent to .

This can be improved upon, of course, by increasing the number of bits available to increase the **range**, and increasing the number of bits available after the binary point to increase **precision**.

If the decimal number is negative, e.g. , proceed as above and then complement the number by either flipping the bits and adding 1, or copy all digits from the right until the first is met and then flipping the bits. So, becomes which is also an approximation of course.

Converting from fixed point binary into decimal is just a case of generating the sum of products by multiplying each digit by its equivalent power of two.

## Floating point

It’s important to note that floating point numbers are not real numbers. Remember real numbers are all numbers from to . Computers are finite machines so there is an obvious limit on the maximum and minimum values but worse than that there will be significant gaps between adjacent floating point numbers. These errors will be considered [later](05_DataRepresentation/errors.md).

When representing numbers with a fractional part in decimal we will often used **scientific notation** which takes the form , where is between , and is any integer. Thus, can be written as . The exponent, , tells us how many places to move the decimal point to return the mantissa, to its actual value. Scientific notation is usually reserved for numbers that are either too small or too big to represent using conveniently in decimal form.

A similar form is used when representing binary numbers with a fractional part using a floating point representation where the mantissa will be any real number greater than or equal to and less than and the exponent will be an integer. Again, the exponent tells us how many places to shift the binary point. Shifting the binary point one place to the right is equivalent to multiplying by ; shifting it one place to the left is the equivalent of dividing by .

We need to be made aware of the number of bits reserved for the mantissa, and the number of bits for the exponent [[1]](#footnote-1). The examples used here will use 10 bits for the mantissa, and 6 bits for the exponent. The most significant bit can be regarded as the **sign bit** with the implied binary point between it and the next most significant bit.

For example, consider the following 16 bit binary number, , which uses 10 bits for the mantissa and 6 bits for the exponent with an implied binary point as indicated:

|  |  |
| --- | --- |
| mantissa | exponent |
| 0.100100100 | 000100 |

The exponent has the value so the implied binary point needs to be moved places to the right which gives the following::

This can then be converted into decimal in the usual way:

If the exponent is negative the binary point needs to move to the left.

For example, consider the following 16 bit binary number, :

|  |  |
| --- | --- |
| mantissa | exponent |
| 0.101000000 | 111110 |

The value of the exponent is () so the implied binary point is moved places to the **left** which leaves the following binary number:

This can then be converted into binary in the usual way:

Finally, we’ll look at an example with a negative mantissa: :

|  |  |
| --- | --- |
| mantissa | exponent |
| 1.100100100 | 000100 |

First convert the exponent to decimal: . The exponent is positive so the implied binary point has to move to the right places:

Then convert this number into decimal in the usual way:

NB. Each of the above can also be worked out mathematically using the formula:

## Normalisation of floating point numbers

Referring back to scientific notation, it is possible to represent the decimal number in several ways, e.g.

* etc..

There needs to be a standard form. For decimal the standard form for the mantissa must be at least and less than . Thus the correct form for our number is . As well as clearing up some confusion the purpose of **normalising** the format is to maximise precision by ensuring we use as many of the significant digits as possible.

Floating point binary numbers also need to be normalised to maximise precision i.e. to use as many of the significant digits as possible. It also provides a unique representation of the number. Normalising the number ensures the digit to the right of the implied binary point will be a significant digit. It will be a for a positive mantissa; and a for a negative mantissa. NB. The first two digits will always be different.

For example, using 10 bits for the mantissa and 6 bits for the exponent, written in groups of 4 bits to aid readability:

* We can see this number is not normalised because the two digits in both the mantissa and the exponent are the same. So the first step is to correct this with the mantissa moving the binary point 1 place to the right and subtracting from the exponent to compensate:
* This number is now normalised.

It’s a similar process for negative numbers:

* The binary point needs to be moved two places to the right to ensure the leading digits are different whilst preserving the sign, and the exponent is adjusted to be two less:
* This number is now normalised

Being able to convert a binary floating point normalised number into decimal (and back) is often an exam question so here is the algorithm and some further examples:

**Converting from binary to decimal**

Convert into decimal:

|  |  |  |
| --- | --- | --- |
| # | Step | Example |
| 1 | Write down the mantissa, with the point inserted after the sign bit |  |
| 2 | If negative, find the twos complement of the mantissa | (Mantissa is positive) |
| 3 | If exponent is negative, find its twos complement | (Exponent is positive) |
| 4 | Calculate the value of the exponent |  |
| 5 | Move the binary point places to the right |  |
| 6 | Convert the result into decimal |  |

Convert into decimal:

|  |  |  |
| --- | --- | --- |
| # | Step | Example |
| 2 | Mantissa negative so find the twos complement |  |
| 3 | Calculate exponent (000101) in denary |  |
| 4 | Adjust point in mantissa (move point 5 places right) |  |
| 5 | Convert mantissa to denary |  |

Convert the number 1010000000 111101 to denary

|  |  |  |
| --- | --- | --- |
| # | Step | Example |
| 1 | Write down mantissa |  |
| 2 | Mantissa negative so find the twos complement |  |
| 3 | Exponent negative so find the twos complement |  |
| 4 | Calculate exponent (- 000011) in denary |  |
| 5 | Adjust point in mantissa (move point 3 places left) |  |
| 6 | Convert mantissa to denary |  |

**Converting from Denary to Binary**

Convert to floating point form

|  |  |  |
| --- | --- | --- |
| # | Step | Example |
| 1 | Convert number (123.5) to pure binary |  |
| 2 | Normalise mantissa |  |
| 3 | If negative, convert to twos complement | (not negative) |
| 4 | How far has the point moved? | 7 places |
| 5 | In which direction? (If left, positive) | left, so exponent = +7 |
| 6 | Convert exponent to twos complement binary |  |
| 7 | Add 0’s to the mantissa to fill bits |  |
| 8 | Answer: |  |

Convert to floating point form

|  |  |  |
| --- | --- | --- |
| # | Step | Example |
| 1 | Convert number (0.1875) to pure binary |  |
| 2 | Normalise mantissa |  |
| 3 | If negative, convert to twos complement | (not negative) |
| 4 | Point has moved 2 places right | exponent = |
| 5 | Convert exponent to twos complement binary |  |
| 6 | Add 0’s to the mantissa |  |
| 7 | Answer: |  |

Convert -0.375 to floating point form

|  |  |  |
| --- | --- | --- |
| # | Step | Example |
| 1 | Convert number (-0.375) to pure binary |  |
| 2 | Normalise mantissa |  |
| 3 | Number negative so find twos complement |  |
| 4 | The point has moved 1 place right | exponent = |
| 5 | Convert exponent to twos complement binary | 111111 |
| 6 | Add 0’s to the mantissa |  |
| 7 | Answer: |  |

## Questions

1. The binary number shown below is represented as a normalised floating point number with an 8- bit mantissa and a 4-bit exponent. The mantissa and exponent are both stored using two’s complement. (The assumed binary point is between the two digits to the left of the number.)  **Convert the number into denary**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Mantissa | | | | | | | | Exponent | | | |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |

1. Convert the following twos complement floating point number into denary.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Mantissa | | | | | | | | Exponent | | | |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

(NB. The exponent is always using twos complement!)

1. Convert +0.3125 to binary, encoding the answer as a **normalized floating point number** with an 8-bit mantissa and a 4-bit exponent. Both mantissa and exponent use twos complement.
2. The denary number -0.140625 bas been converted to binary and held as a normalized floating point number with an 8-bit mantissa and a 4-bit exponent, both using twos complement. Which of the following is the correct representation?
   1. 1.0111000 0010
   2. 1.0111000 1110
   3. 1.1101110 0000
   4. 0.1001000 1110
3. The binary number below is represented as a floating point number with a 10-bit mantissa and a 4-bit exponent, both using twos complement. Convert this number into denary

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Mantissa | | | | | | | | | | Exponent | | | |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

1. NB. The standard representation has been laid down by the IEEE and you can search for this with your favourite search engine. For the A Level you only need to know, understand and use a simplified floating point representation used here. [↑](#footnote-ref-1)