FLOATING POINT REPRESENTATION

What about the other numbers?

So far we know how to store **integers** **Whole Numbers**

But what if we want to store **real numbers** Numbers **with decimal fractions**

Even 27.5 needs another way to represent it.

This method is called **floating point representation**

Fixed Notation

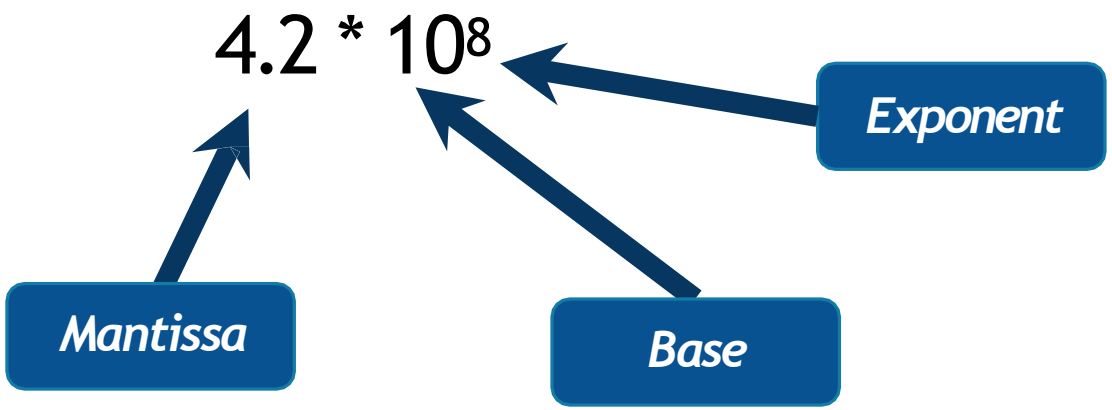
We are accustomed to using a **fixed notation** where the decimal point is fixed and we know that any numbers to the right of the decimal point are the decimal portion and to the left is the integer part

E.g. 10.75

10 is the Integer Portion and 0.75 is the decimal portion

Floating Point Representation

The structure of a **floating point**(real) number is as follows:



Only the **mantissa** and the **exponent** are **stored.** The **base is implied** (known already)
As it is not stored this will save memory capacity

IEEE standard

There is a **IEEE** standard that defines the **structure of a floating point number**
IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)

It defines 4 main sizes of floating point numbers
16, 32, 64 and 128 bit

Sometimes referred to as Half, Single, Double and Quadruple precision

A 32 bit floating pointnumber

Sign	Exponent	Mantissa
1bit	**8 bits**	**23 bits**

S is a sign bit

0 =positive

1 =negative

23 bits for the **mantissa**

8 bits for the **exponent**

Example

We want the format of a number to be in

$$\mathbf{m \times b^{e}}$$

We want the mantissa to be a single decimal digit

Example

$$3450.00 = 3.45 \times 10^{3}$$

The **exponent** is **3** as the decimal place has been moved 3 places to the left

Decimal fractions

First we will look at how a decimal number is made up: **173.75**

Hundreds	Tens	Units	Decimal place	Tenths	Hundredths
1	7	3	.	7	5

10^2	10^1	10^0	Decimal place	10^{-1}	10^{-2}
1	7	3	.	7	5

Binary fractions

Then look at how the same number could be stored in binary: 1010 1101

128	64	32	16	8	4	2	1	.	0.5	0.25
1	0	1	0	1	1	0	1		1	1

This number is constructed as shown above (in a fixed point notation).

These values come from

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}
1	0	1	0	1	1	0	1		1	1

But the problem is

We don't actually have a decimal point in binary...

Example

In decimalfirst

250.03125

First convert the **integer** part of the **mantissa** into binary (as you have done previously)

250 =**1111 1010**

Now to convert the **decimal portion** of the mantissa

.03125

Example (cont)

Decimal fraction => **.03125**

Multiply and use any remainder over 1 as a carry forward. Continue until you reach **1.0 with no carry over**

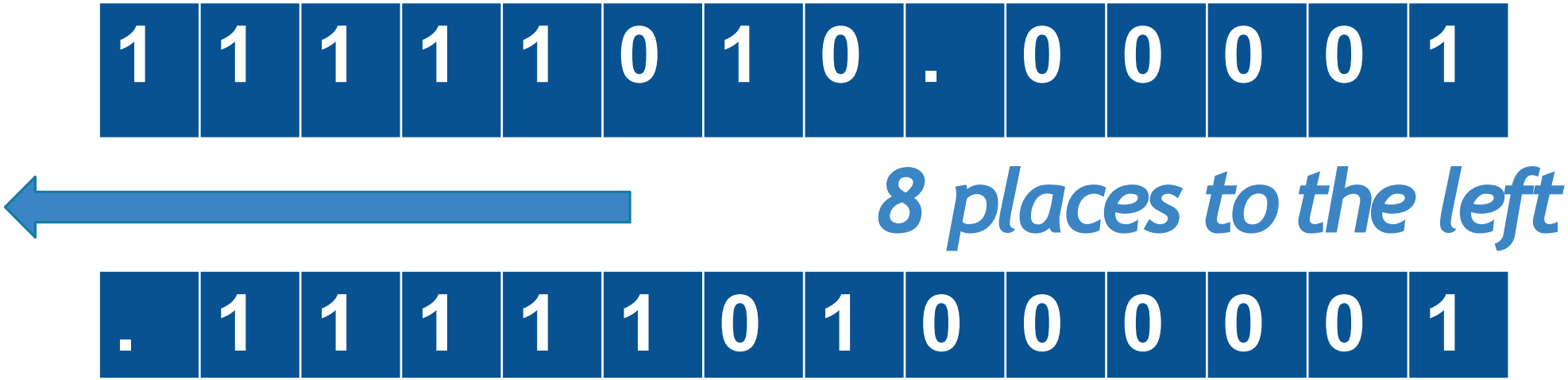
0.03125 * 2 = **0** r 0.0625
0.0625 * 2 = **0** r 0.125
0.125 * 2 = **0** r 0.25
0.25 * 2 = **0** r 0.5
0.5 * 2 = **1** r 0

Binary fraction = **0.00001**

So far….

So far wehave : **1111 1010.00001** (250.03125)

But we need it in the format :.11111 0100 0001 (the decimal point to the left of the 1)



So the exponent is **8** (1000)

Example

So back to our example

Mantissa =.11111 0100 0001 (2.5003125)

Exponent = 0000 1000 (8)

Sign Bit = 0

And the number is **positive** so the **sign bit is 0**

In 32 bit representation there is

- **8** bits for the **exponent**
- **23** bits for the **mantissa**

We will pad the **left** of the exponent with 0's up to **8** bits

We will pad the **right** of the mantissa with 0's up to **23** bits

S	Exponent	Mantissa
0	0000 1000	11111 0100 0001**00000000000**
1bit	8 bits	23 bits

Further Example 1

102.9375

Sign =0 (+ve) **Integer = 102 = 1100110**

Decimal portion = .1111 -> Number = 1100110.1111 -> Needs to be .11001101111

Exponent = 7 = 00000111

Number (32 bit Single Precision) = **0 00000111 11001101111**000000000000

Further Example 2

250.75

Sign =0 (+ve) **Integer = 250 = 11111010**

Decimal portion = .11 -> Number = 11111010.11 -> Needs to be .1111101011

Exponent = 8 = 00001000

Number (32 bit Single Precision) = 0 00001000 11111010110000000000000

What about small numbers?

What if we are storing 0.0625?

The decimal point doesn't need moved to the left it needs moved to the right…

Example (cont)

Decimal fraction => **.0625**

Multiply and use any remainder over 1 as a carry forward. Continue until you reach **1.0 with no carry over**

0.0625 * 2 = **0** r0.125
0.125 * 2 = **0** r0.25
0.25 * 2 = **0** r0.5
0.5 * 2 = **1** r0.0

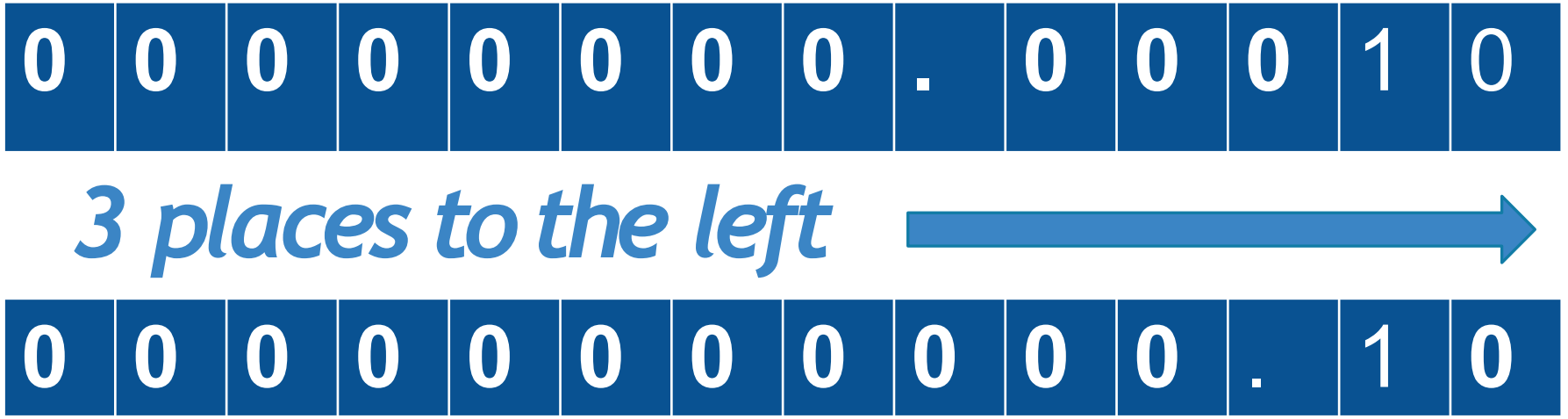
Binary fraction =
0.0001

So far….

So far we have : **0.0001** (0.0625)

But we need it in the format .10000000000000000000000

(leading bit after the . has to be a 1)



So the exponent is **-3**

Example

So back to our example

Mantissa =0.1(0.0625)

Exponent = 1111 1101 (-3)

Sign Bit = 0

In 32 bit representation there is **23** bits for the mantissa

We will pad the **right** of the number with 0's up to **23** bits

And the number is **positive** so the **sign bit is 0**

S	Exponent	Mantissa
0	1111 1101	10000 0000 00000000000000
1bit	8 bits	23 bits

If the Exponent is negative

- In reality there are other ways that this is dealt with (offset exponents for those that are interested)

- But for the purpose of the course we will store a negative exponent in 8 bit two's complement:

2	=	0000 0010
-2	=	1111 1110

What about really small numbers?

What if we are storing 0.0009765625?

The integer portion is **0**

The decimal portion is: **.0000000001**

So our number need to be **0.1**

We need to shift the exponent 10 places to the **right**

This means we need to store **-10** as the **exponent** (two's complement)

Further Example 3

0.0009765625?

Sign =0 (+ve) **Integer = 0 = 0000000**

Decimal portion = **.0000000001** **->** Number = **0.0000000001** **-> Needs to be .1**

Exponent = -10 = (**+10 = 0000 1010) -10 = 1111 0110**

Number (32 bit Single Precision) = **0 11110110 01**00000000000000000000

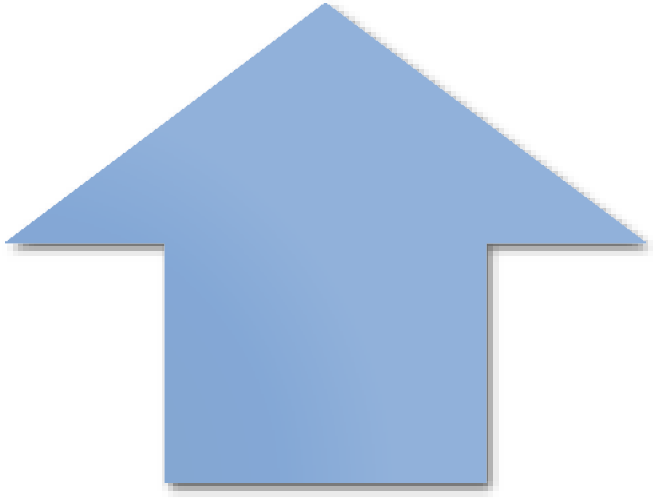
Problems with floatingpoint

What if we try to store 25.333?

We need much more bits in the mantissa to deal with this…

Increased Mantissa allocation

More bits allocated to ***Mantissa***



Increased Accuracy*/precision*

Different Precision Numbers

Single Precision (32 bit)

S	Exponent	Mantissa
1bit	8 bits	23 bits
1.18×10^{-38} to 3.40×10^{38}		

Double precision (64 bit)

S	Exponent	Mantissa
1bit	11 bits	52 bits
2.23×10^{-308} to 1.79×10^{308}		

Summary

If you increase the amount of bits allocated to the **Mantissa** you increase the **accuracy/precision**

If you increase the amount of bits allocated to the **exponent** you increase the **range** of the number

Mnemonic
MARE - Mantissa Accuracy Range Exponent

Summary - Single precision Floating point

1. Create the **mantissa** portion (The integer part)
2. Create the **decimal fraction**
3. Calculate **exponent** by moving decimal point till number is in the format 1.xxxxx
4. Convert exponent to two's complement if is negative (moving the point to the right)
5. Add the s**ign bit**
   - 0 =+ve
   - 1 =-ive
6. Write in the format **sign exponent mantissa**