Learning Aims

- Represent positive integers in binary and hexadecimal
- Convert between binary, hexadecimal and denary

Binary number system – representing integers

Binary is a base 2 number system

- The bit at the leftmost position is known as the Most Significant Bit (MSB)
- The bit at the rightmost position is known as the Least Significant Bit (LSB)

Place Values	2^8	2^7	2^6	2^5	2^4	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
	0	**0**	**0**	**1**	**0**	**1**	**0**	**1**

Adding these together gives **21**

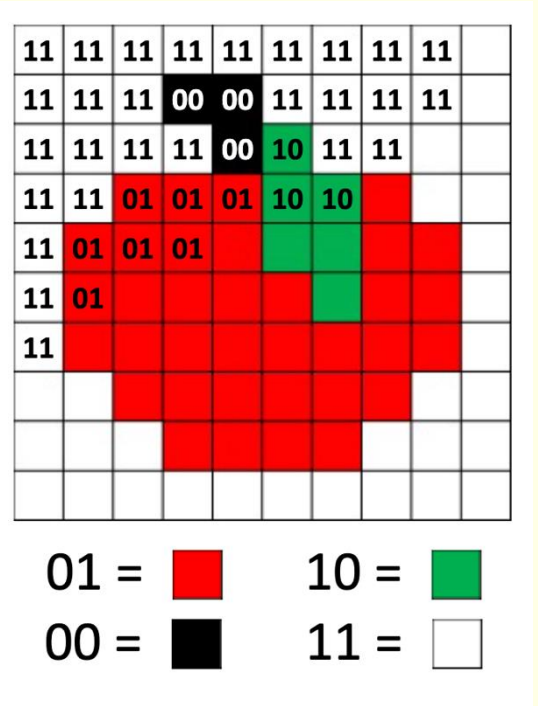
Using binary to represent data and program instructions

- Numbers, letters, images, sound and instructions have to encoded in binary
- For example, ASCII encoding system for representing text as binary numbers:

Letter	Binary
8	0011 1000
<space>	0010 0000
b	0110 0010
i	0110 1001
t	0111 0100
s	0111 0011

When we display a binary pattern as characters on the screen, it looks up the graphic representation for it in the font definition and sends it to the screen

To encode an image, each tiny picture element (pixel) is allocated its own binary pattern, for example 2 bits per pixel can generate 4 different possible colours.



Hexadecimal number system

The hexadecimal system, often referred to as simply ‘hex’

Base of 16.

Denary	Binary (4-bit)	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Converting from binary to hexadecimal and vice versa

To convert a binary number to hexadecimal, split the binary number into groups of 4 binary digits.

Binary	0011	1010	1111	1010
Hex	3	A	F	9

The hex representation of 0011 1010 1111 1010 is therefore 3AF9.

To convert from hex to binary, perform this operation in reverse by grouping the bits in groups of 4 and
translating each group into binary. For example, to convert the number 23

HEX	2	3
Binary	0010	0011

= 00100011

Q1: Convert the hexadecimal number A7 into binary.

Q2: What is 1111 1111 in hexadecimal?

Converting from hexadecimal to denary and vice versa

To convert from hexadecimal to denary, remember that the left column now represents 16s and not tens.

For example, to convert 27_{16}

	16 s	1s
HEX	2	7

= 2 x 16 + 7 = 39

To convert a denary number to hex, the easiest way is to first convert the denary number to binary and

then translate from binary to hex. For example, to convert 75_{10}

	128	64	32	16	8	4	2	1
Binary	0	1	0	0	1	0	1	1
Split into 2 nibbles	8	4	2	1	8	4	2	1
Hex	4				B			

Therefore 75 = 4B

Another way

75/16 = 4 remainder 11, or 4B, since 11 is B in hexadecimal

A2C as a denary number is:

Column value	$256 = 16^2$	$16 = 16^1$	$1 = 16^0$
Hexadecimal number	A	2	C
Denary value	10*256	2*16	12

A2C is 2560+32+12 = 2604 in denary

Why the hexadecimal number system is used

- The hexadecimal system is used as a **shorthand for binary** since it is simple to represent a byte in just two digits

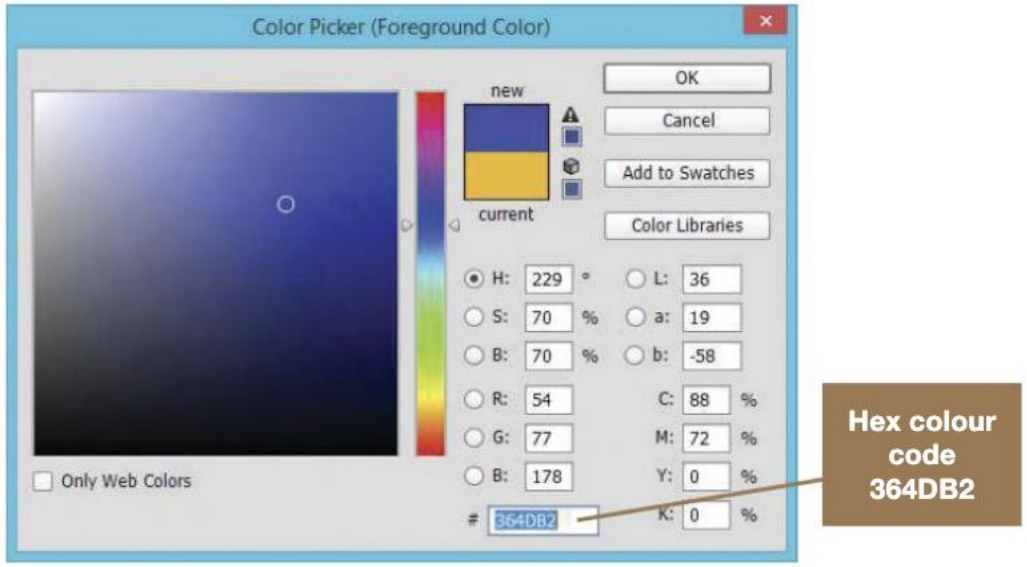
- Fewer mistakes are likely to be made in writing a hex number than a string of binary digits.

- It is easier for technicians and computer users to write or remember a hex number than a binary number.

Colour codes in images often use hexadecimal to represent the RGB values, as they are much easier to remember than a 24-bit binary string. In the example overleaf #364DB2 represents 3616

36 for Red,
4D for Green and B2 for Blue

Can be displayed or printed in the Colour Picker window more compactly than in binary.



Leaning Aims

ASCII and Unicode

- Define a bit as a 1 or a 0, and a byte as a group of eight bits
- Know that 2^n
- different values can be represented with n bits
- Use names, symbols and corresponding powers of 2 for binary prefixes e.g. Ki, Mi
- Differentiate between the character code of a denary digit and its pure binary representation
- Describe how character sets (ASCII and Unicode) are used to represent text

Unit nomenclature

- Although we frequently refer to 1024 bytes as a kilobyte, it is in fact a kibibyte.
- To avoid any confusion between references to 1024 bytes rather than 1000 bytes, an international collaboration between standards organisations decided in 1996 that kibi would represent 1024, and kilo would represent 1000.
- Kibi is a combination of the words kilo and binary.
- The same is true of the other familiar names Mega, Giga and Tera being replaced by mebi, gibi and tebi
- The tab which a KiB is a kibi

Name	Symbol	Power	Value
kibi	Ki	2^{10}	1024
mebi	Mi	2^{20}	1,048,576
gibi	Gi	2^{30}	1,073,741,824
tebi	Ti	2^{40}	1,099,511,627,776
pebi	Pi	2^{50}	1,125,899,906,842,624
exbi	Ei	2^{60}	1,152,921,504,606,846,976
zebi	Zi	2^{70}	1,180,591,620,717,411,303,424
yobi	Yi	2^{80}	1,208,925,819,614,629,174,706,176

Name	Symbol	Power
Kilo	K or k	10^3
Mega	M	10^6
Giga	G	10^9
Tera	T	10^{12}
Peta	P	10^{15}
Exa	E	10^{18}
Zetta	Z	10^{21}
Yotta	Y	10^{24}

The ASCII code

Historically, the standard code for representing the characters on the keyboard was ASCII (American
Standard Code for Information Interchange).

This uses seven bits which form **128 different bit combinations**, more than enough to cover all of the characters on a standard English-language keyboard.

The first 32 codes represent non-printing characters used for control:

- backspace (code 8)
- Enter or Carriage Return key (code 13)
- Escape key (code 27).
- Space character code 32
- Delete as code 127.

ASCII	DEC	Binary	ASCII	DEC	Binary	ASCII	DEC	Binary	ASCII	DEC	Binary
NULL	000	000 0000	space	032	010 0000	@	064	100 0000	`	096	110 0000
SOH	001	000 0001	!	033	010 0001	A	065	100 0001	a	097	110 0001
STX	002	000 0010	"	034	010 0010	B	066	100 0010	b	098	110 0010
ETX	003	000 0011	#	035	010 0011	C	067	100 0011	c	099	110 0011
EOT	004	000 0100	$	036	010 0100	D	068	100 0100	d	100	110 0100
ENQ	005	000 0101	%	037	010 0101	E	069	100 0101	e	101	110 0101
ACK	006	000 0110	&	038	010 0110	F	070	100 0110	f	102	110 0110
BEL	007	000 0111	'	039	010 0111	G	071	100 0111	g	103	110 0111
BS	008	000 1000	(	040	010 1000	H	072	100 1000	h	104	110 1000
HT	009	000 1001	)	041	010 1001	I	073	100 1001	i	105	110 1001
LF	010	000 1010	*	042	010 1010	J	074	100 1010	j	106	110 1010
VT	011	000 1011	+	043	010 1011	K	075	100 1011	k	107	110 1011
FF	012	000 1100	,	044	010 1100	L	076	100 1100	l	108	110 1100
CR	013	000 1101	-	045	010 1101	M	077	100 1101	m	109	110 1101
SO	014	000 1110	.	046	010 1110	N	078	100 1110	n	110	110 1110
SI	015	000 1111	/	047	010 1111	O	079	100 1111	o	111	110 1111
DLE	016	001 0000	0	048	011 0000	P	080	101 0000	p	112	111 0000
DC1	017	001 0001	1	049	011 0001	Q	081	101 0001	q	113	111 0001
DC2	018	001 0010	2	050	011 0010	R	082	101 0010	r	114	111 0010
DC3	019	001 0011	3	051	011 0011	S	083	101 0011	s	115	111 0011
DC4	020	001 0100	4	052	011 0100	T	084	101 0100	t	116	111 0100
NAK	021	001 0101	5	053	011 0101	U	085	101 0101	u	117	111 0101
SYN	022	001 0110	6	054	011 0110	V	086	101 0110	v	118	111 0110
ETB	023	001 0111	7	055	011 0111	W	087	101 0111	w	119	111 0111
CAN	024	001 1000	8	056	011 1000	X	088	101 1000	x	120	111 1000
EM	025	001 1001	9	057	011 1001	Y	089	101 1001	y	121	111 1001
SUB	026	001 1010	:	058	011 1010	Z	090	101 1010	z	122	111 1010
ESC	027	001 1011	;	059	011 1011	[	091	101 1011	{	123	111 1011
FS	028	001 1100	<	060	011 1100	\	092	101 1100	|	124	111 1100
GS	029	001 1101	=	061	011 1101	]	093	101 1101	}	125	111 1101
RS	030	001 1110	>	062	011 1110	^	094	101 1110	~	126	111 1110
US	031	001 1111	?	063	011 1111	_	095	101 1111	DEL	127	111 1111

Character form of a denary digit

Although numbers are represented within the code, the number character is not the same as the actual
number value.

The ASCII value 0110111 will print the character ‘7’, even though the same binary value equates to the denary number 55.

Therefore ASCII cannot be used for arithmetic and would use unnecessary space to store numbers.

Numbers for arithmetic are stored as pure binary numbers.

‘7’ + ‘7’ (i.e. 0110111 + 0110111 in ASCII) would be 77, not 14 or 110.

The development of ASCII

ASCII originally used only 7 bits, but an 8-bit version was developed to include an additional 128 combinations to represent symbols such as æ, © and ƒ.

You can try holding down the ALT key and typing in the code number using the number pad to type one of these symbols.

For example, ALT+130 will produce é, as used in café.

The 7-bit ASCII code is compatible with the 8-bit code and simply adds a leading 0 to all binary codes.

Unicode

- By the 1980s, several coding systems had been introduced all over the world that were all incompatible with one another.
- This created difficultly as multilingual data was being increasingly used and a new, unified format was sought. As a result, a new **16-bit code** called Unicode (UTF-16) was introduced.
- This allowed for 65,536 different combinations and could therefore represent alphabets from dozens of languages including Latin, Greek, Arabic and Cyrillic alphabets.
- The first 128 codes were the same as ASCII so compatibility was retained.
- A further version of Unicode called **UTF-32** was also developed to include just over a million characters, and this was more than enough to handle most of the characters from all languages, including Chinese and Japanese.
- This meant that whilst there is now just one globally recognised system to maintain, one character in this scheme uses four bytes instead of two, significantly increasing file sizes and data transmission times.

Learning Aims: Binary arithmetic

- Use sign and magnitude to represent negative numbers in binary
- Use two's complement to represent negative numbers in binary
- Add and subtract binary integers
- Represent fractions in fixed point binary

Signed vs unsigned binary

Signed binary number can be positive or negative vs **unsigned binary number** can **only be positive**.

If a binary number is signed, then the first bit (the most significant bit, the leftmost) will be a 1 if it is a negative number, or a 0 if it is positive, e.g. 10010111 will be a negative number, whilst 01011011 will be positive.

There are a number of different ways to represent a negative number:

- **Sign and magnitude**
- **Two's complement**

In an examination you will be told what form the binary is in, whether it is signed or unsigned, or if it is in two's compliment or sign and magnitude. If you are not told anything, then treat it as an unsigned binary number.

Sign and magnitude

- Sign and magnitude is the simplest way of representing negative numbers in binary.
- The most significant bit (MSB) is simply used to represent the sign: 1 means the number is negative, 0 means it is positive.
- So to represent −4 in 8-bit binary you would write out the 4 in binary, 00000100, then change the MSB to a 1, creating 10000100.
- In another example, −2 would be 10000010 in the sign and magnitude notation.
- -103 would be:

Column Value	Sign bit	64	32	16	8	4	2	1
Binary number	1	1	1	0	0	1	1	1

Sign and magnitude

- This notation isn't used very often as using the MSB as a sign bit means that the largest number that can be represented using 8 bits is **127, much less than 255.**
- It also makes it **harder to do calculations** as different bits mean different things; some represent numbers, others represent signs.
- Also, their **value of zero is represented twice** as both positive and negative 0.
- MSB notation can represent numbers in the range −127 to 127.

Two's complement

- Two's complement is a much more useful way of representing binary numbers as it allows you to use negative numbers without reducing the range of numbers you can use (being able to represent −128 to 127).
- This method makes the MSB a negative value.

- The easiest way to show a negative number using two's complement is to write it out as a positive number using the usual binary method. Then, starting at the right-hand side, leave every bit up to and including the first 1 along, but invert all the bits after this.

Column Value	-128	64	32	16	8	4	2	1

- For example, to show −86 in binary, first work out 86. 86 is 01010110 because 64 16 4 2 equals 86:

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

- Then start at the right-hand side and leave everything alone up to and including the first 1:

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

- Finally invert everything after this, so all the 1s become 0s and vice versa:

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

Column Value	-128	64	32	16	8	4	2	1
Binary number	1	0	1	0	1	0	1	0

Two's complement

- Another way
- To store -103 we record -128 + 25 or

Column Value	-128	64	32	16	8	4	2	1
Binary number	1	0	0	1	1	0	0	1

Representing Positive number using 2's complement

32	16	8	4	2	1
0	1	0	0	1	0

MSB

IF the MSB is a 0 then it will be a positive number

16 + 2 = 18

-32	16	8	4	2	1
1	1	0	0	1	0

MSB

IF the MSB is a 1 then it will be a negative number

-32 + 18 = -14

Adding integers in binary

- In binary when we add 0s and 1s we have the following possible outcomes.

	Carry	Sum
0 + 0	0	0
0 + 1	0	1
1 + 0	0	1
1 + 1	1	0
1 + 1 + 1	1	1

Example

	Carry	Sum
0 + 0	0	0
0 + 1	0	1
1 + 0	0	1
1 + 1	1	0
1 + 1 + 1	1	1

00001011 + 10011011

	128	64	32	16	8	4	2	1	Denary Check
	0	0	0	0	1	0	1	1	11
+	1	0	0	1	1	0	1	1	155
=	1	0	1	0	0	1	1	0	166
			1	1		1	1		

Adding using two's complement numbers

- Adding two's complement values is the same process as adding standard binary integers, but adding two large numbers together does throw up an interesting phenomenon.

- The two large numbers when added together are too large to store in the 8-bit two's complement integer and the value **overflows** the available bits, creating a negative number.

- If the calculation were to result in a number that was too small to represent, then this would be called **underflow**.

Adding together 2 two's complement numbers

Add together these 2 two's complement numbers:

0010 1010 and 1101 0110

Ignoring the overflow, what result do you get?

The result is 1 0000 0000, which is −256. Ignoring the overflow gives 0000 0000.

Why do you think this is the case?

These numbers represent +42 and −42. They add to zero. It does this with an overflow bit, which is not stored in the result.

Binary subtraction using two's complement

- Subtracting two's complement numbers is a relatively straightforward process.
- We convert the number to be subtracted into a negative two's complement number and add them together.
- If you get an overflow when subtracting you lose the 1 value. This will still give the correct positive two's complement value in the 8-bits.
- You can use one's complement to change the 0s to 1s and 1s to 0s in a binary number when subtracting.

Example

Binary subtraction is best done by using the negative two's complement number and then adding the second number.

73 – 58 or 01001001 - 00111010

	-128	64	32	16	8	4	2	1	Denary	Notes
	0	0	1	1	1	0	1	0	58	Work out + 58
	1	1	0	0	0	1	1	0	-58	Flip after first 1
+	0	1	0	0	1	0	0	1	73	Add 73
(1)	0	0	0	0	1	1	1	1	15	Overflow (1) is lost

Key points

- When adding two binary numbers, we only have a small range of possibilities
- When we add two relatively large values, it is possible there is a carry in the MSB, leading to overflow, that is, the number is too big to store in the space allocated.
- When adding two two's complement values, this overflow can carry into the MSB and turn a positive value into a negative value.
- To subtract one two's complement number from another, simply take the two's complement form of the number to be subtracted and add.

Did you know?

The digital electronics that makes a computer work doesn't know (or care) if 10001000 is 2s complement or not.

It could just as easily represent the positive number 136 (128 + 8).

The computer processes the binary data (for addition, subtraction etc…) in exactly the same way.

It's down to the software rather than the hardware to remember if the binary data is a signed (2s complement) number or an unsigned (positive) number.

This can sometimes lead to some pretty catastrophic software failures when programmers aren't aware of how computers handle negative numbers.

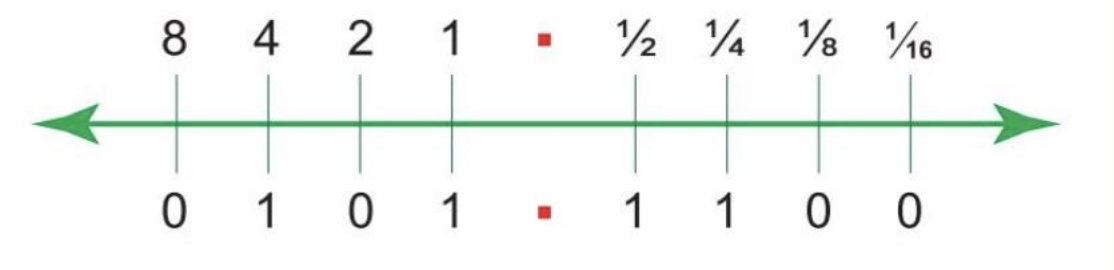
Learning Aims

Floating point arithmetic

- Represent positive and negative numbers with a fractional part in **fixed point** and **floating point** form
- Normalise un-normalised floating point numbers with positive or negative mantissas
- Add and subtract floating point numbers
- Explain underflow and overflow and describe the circumstances in which they occur

Fixed point format

- Fixed point binary numbers can be a useful way to represent fractions in binary. A binary point is used to
- separate the whole place values from the fractional part on the number line:



In the binary example above, the left hand section before the point is equal to 5 (4+1) and the right hand section is equal to ½ + ¼ (¾), or 0.5 + 0.25 = 0.75. So, using four bits after the point, 0101 1100 is 5.75 in denary.

A useful table with some denary fractions and their equivalents is given below:

Binary fraction	Fraction	Denary fraction
0.1	1/2	0.5
0.01	1/4	0.25
0.001	1/8	0.125
0.0001	1/16	0.0625
0.00001	1/32	0.03125
0.000001	1/64	0.015625
0.0000001	1/128	0.0078125
0.00000001	1/256	0.00390625

Converting a denary fraction to fixed point binary

To convert the fractional part of a denary number to binary, you can employ the same technique as you
would when converting any denary number to binary.

Take the value and subtract each point value from the amount until you are left with 0.

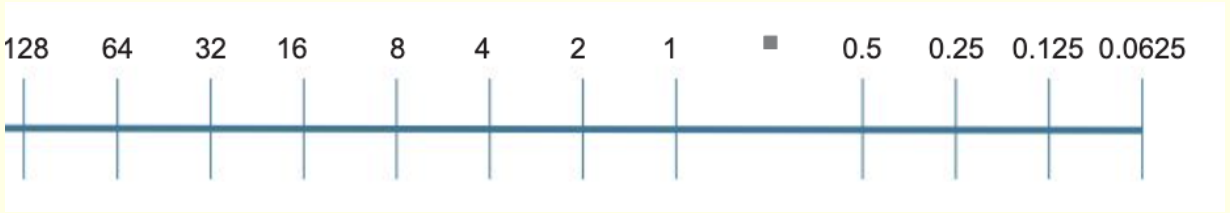
Take the example 3.5625 using 4 bits to the right of the binary point:

Subtract 0.5:	0.5625 – 0.5 = 0.0625	**1**
Subtract 0.25 from 0.0625:	Won't go	**0**
Subtract 0.125 from 0.0625:	Won't go	**0**
Subtract 0.0625 from 0.0625:	0.0625 – 0.0625 = 0	**1**

3 = 0011 in binary. 0.5625 = 1001. So 3.5625 = 0011 1001

Recap

- What are the largest and smallest unsigned numbers that can be held in two bytes with four bits after the point?



255.9375

Problem 1

It is worth noticing that this system is not only **less accurate** than the denary system, but some fractions

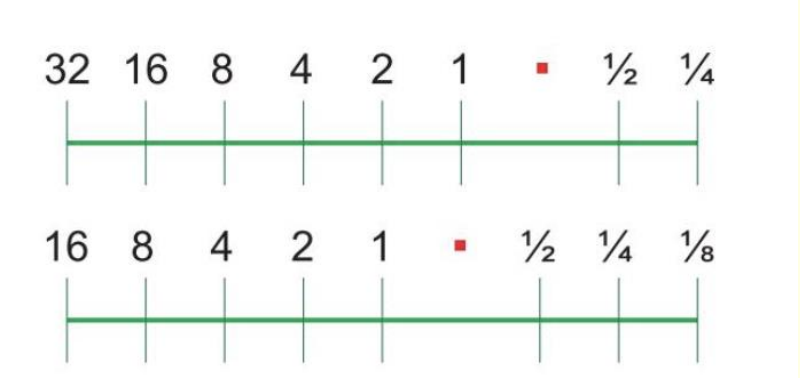
cannot be represented at all. 0.2, 0.3 and 0.4

for example, will require an infinite number of bits to the right of the point.

The number of fractional places would therefore be truncated and the number will not be accurately stored, causing **rounding errors**.

In our denary system, two denary places can hold all values between .00 and .99.

With the fixed point binary system, 2 digits after the point can only represent 0, ¼, ½, or ¾ and nothing in between.



Problem 2

- The range of a fixed point binary number is also limited by the fractional part.
- For example, if you have only 8 bits to store a number to 2 binary places, you would need 2 digits after the point, leaving only 6 bits before it.
- 6 bits only gives a range of 0-63.
- Moving the point one to the left to improve accuracy within the fractional part only serves to half the range to just 0-31.
- Even with 32 bits used for each number, including 8 bits for the fractional part after the point, the maximum value that can be stored is only about 8 million.
- Another format called floating point binary can hold much larger numbers, with greater accuracy.
- Floating point form is covered in the next lesson

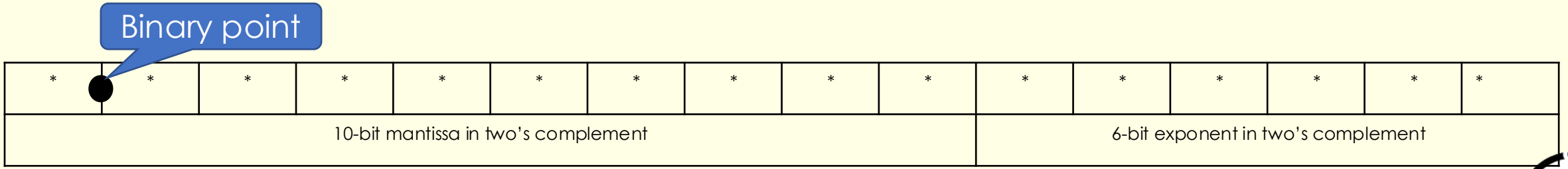
Floating point binary numbers

- Using 32 bits (4 bytes), the largest fixed point number that can be represented with just one bit after the point is only just over two billion.

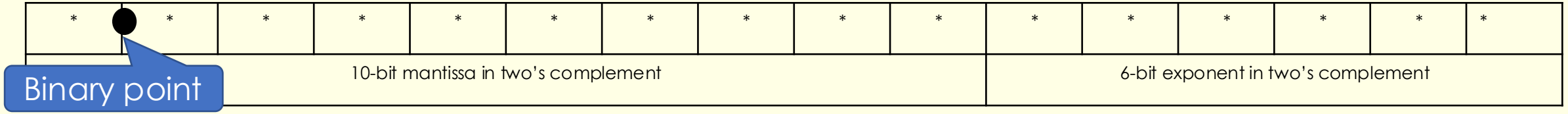
- Floating point binary allows very large numbers to be represented.

Floating Point

- To represent denary fractions (decimals), it is customary to use a standard form so 123.456 is written as 1.23456×10^{2} and 0.00167 as 1.67×10^{-3}.
- The power of 10 shows how many places the decimal point has 'floated' left or right in the number to make the **standard form**.
- The first part of these representations is called the **mantissa** and the power to which the 10 is raised, the **exponent**.
- In binary we use a similar standard form called **floating point**, for example a 16-bit floating point number may be made up of a **10-bit mantissa** and a **6-bit exponent**.



- Real numbers have fractional parts to them; in binary these fractional parts are ½, ¼, $^1/_8$, and so on.
- So the column values associated with the mantissa are:



Column value	-1	½	¼	$^1/_8$	$^1/_{16}$	$^1/_{32}$	$^1/_{64}$	$^1/_{128}$	$^1/_{256}$	$^1/_{512}$
	-1	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625	0.001953125

- The column values for the exponent are:

Column value	-32	16	8	4	2	1

Largest positive and negative number

What does the following binary number (with a 5-bit mantissa and a 3-bit exponent) represent in denary?

-1	1/2	1/4	1/8	1/16	-4	2	1
0	1	1	1	1	0	1	1

This is the largest positive number that can be held using a 5-bit mantissa and a 3-bit exponent, and represents $0.1111 \times 2^3 = 7.5$

The most negative number that can be held in a 5-bit mantissa and 3-bit exponent is:

-1	1/2	1/4	1/8	1/16	-4	2	1
1	0	0	0	0	0	1	1

This represents $-1.0000 \times 2^3 = -1000.0 = -8$

Floating Point Binary Numbers

Floating point binary allows very large numbers to be represented.

Positive Exponent

Move the point
3 places to the right

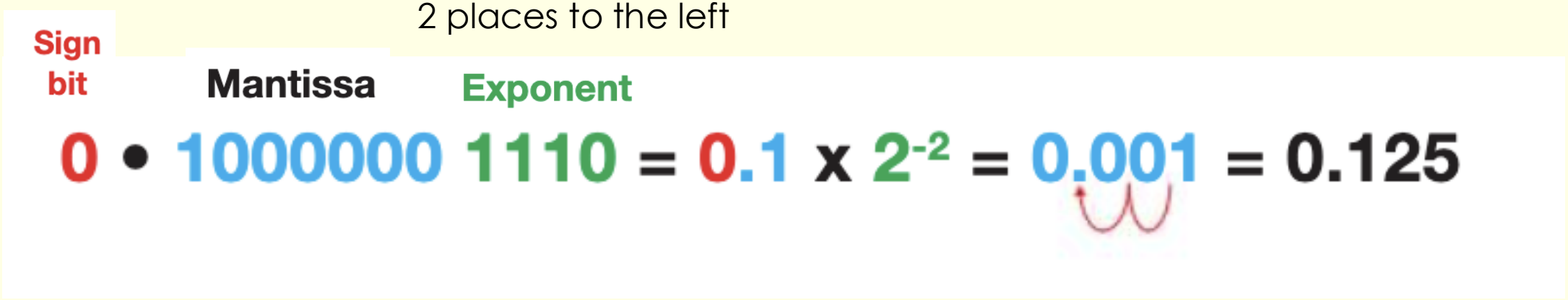
Convert the following floating point numbers to denary:

(a) 0 • 1101010 0100
(b) 0 • 1001100 0011

Floating Point Binary Numbers

- **Negative exponents**

Move the point
2 places to the left



Convert the following floating point number to denary:

0 • 1100000 1110

Floating Point Binary Numbers

Handling negative mantissas

Flip the bits from the first 1 from the right

Move point 5 places to the right

$$1 \bullet 0101101\ 0101 = -0.1010011 \times 2^{5} = -10100.11 = -20.75$$

Find the twos complement of the mantissa. It is 0.1010011, so the bits represent -0.1010011

Translate the exponent to denary, 0101 = 5

Move the binary point 5 places to the right to make it larger. The mantissa is -010100.11

Translate this to binary

The answer is -20.75.

16	8	4	2	1	1/2	1/4
1	0	1	0	0	1	1

Answer:

Step 1. Break the floating point number into its two parts and place a decimal point in the mantissa between the most significant bit and the next one:

Mantissa is 0.1100 because the question states it uses the first 5 bits.

Exponent is 010

Both are positive binary numbers because they have a **leading zero**.

So the mantissa in denary form is 1100 -> 12 denary
The exponent is 010 -> 2 denary

Step 2. Slide the decimal point of the mantissa by the exponent value i.e. 2 places to the right

The floating point numbers represents **11.00 binary or +3.0 decimal**

Range and Accuracy

- With floating point representation, the balance between the **range** and **precision** depends on the choice of numbers of bits for the mantissa and the exponent.

- A large number of bits used in the mantissa will allow a number to be represented with **greater accuracy**, but this will **reduce** the number of bits in the exponent and consequently the **range of values** that be represented.

Normalising a positive binary number

Normalisation is the process of moving the binary point of a floating point number to provide the maximum level of precision for a given number of bits

- Having a large mantissa improves the accuracy with which a number can be represented but this would be entirely wasted if the mantissa contained a number of leading 0s or leading 1s.
- For this reason, floating point numbers are normalised.

Positive numbers	Negative numbers
No leading 0s to the left of the most significant bit and immediately after the binary point. **Starts with 01** The binary fraction 0.000101 becomes 0.101×2^{-3} 8 bit mantissa and 3 bit exponent 0.1010000 101	No leading 1s to the left of the mantissa. **Starts with 10** Negative number 1.110010100 (10 bits) would become 1.00101×2^{-2} 8 bit mantissa and 3 bit exponent 1.0010100 110

Normalising a Positive Floating Point binary number

A positive number has a sign bit of 0 and the next digit is always 1.

Normalise the binary number

0.0001011 0101

8-bit mantissa and a 4-bit exponent

0.0001011	Binary point needs to move 3 places to the right so that there is no leading 0's
0.1011000	Making the mantissa larger means we must compensate by making the exponent smaller
0010	So subtract 3 from the exponent (5-3=2)

The normalised number is: 01011000 0010

Normalising a Negative Floating Point binary number

A normalised negative number has a sign bit of 1 and the next bit is always 0.

Normalise the binary number

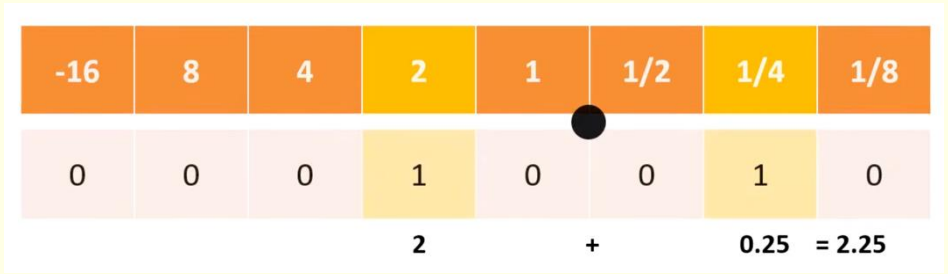
1.1110111 0001

8-bit mantissa and a 4-bit exponent

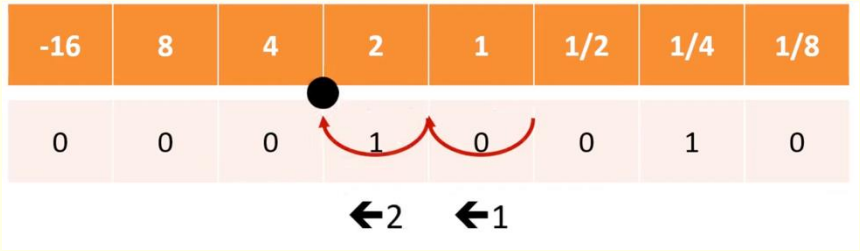
1.1110111	Move the binary point right 3 places, so no leading 1's
1.0111000	Moving the binary point to the right makes the number larger, so we must make the exponent smaller to compensate.
1110	Subtract 3 from the exponent. The exponent is now 1 – 3 = -2 = 1110

The normalised number is 10111000 1110

Representing a positive normalised floating point number 2.25



Step 1: Write out 2.25 on a standard fixed point number line

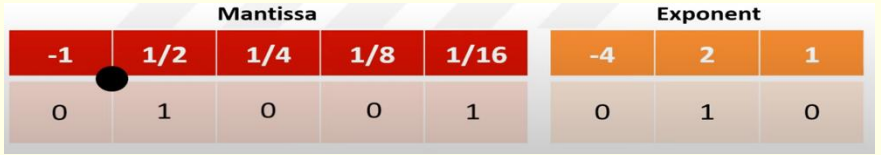


Step 2: move the binary point so it sits between the first 0 and 1

Normalised positive always starts with 01

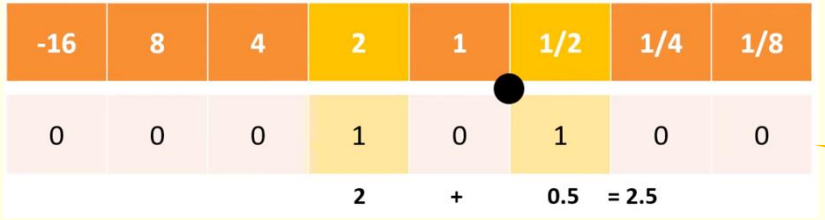
Move 2 places to the left

Step 3: Work out what the exponent should be – 2 places to the right to put the binary point back to the correct position.



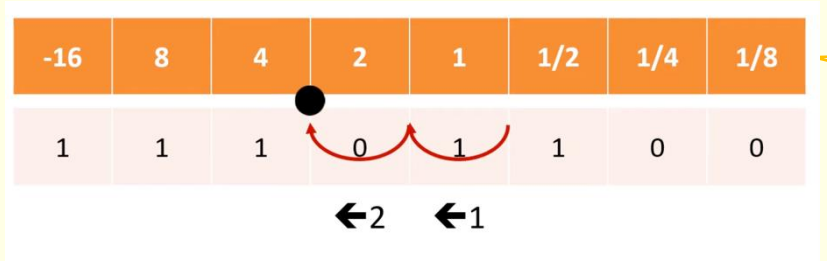
Step 4: store the mantissa

Representing a negative normalised floating point number -2.5



Step 1: Write out the positive version of the number

Step 2: convert number to negative
Flip the bits after the first 1



Step 3: next move the floating point to the first 10.

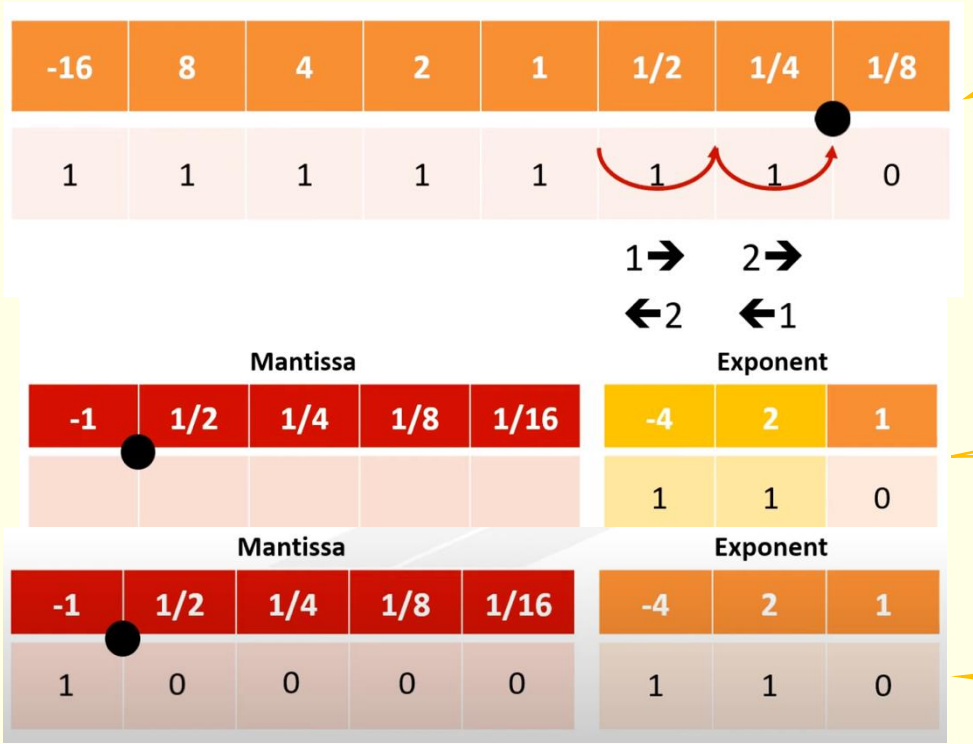
Step 4: work out exponent, 2 to the right

Step 5: store the mantissa

Representing a negative normalised floating point number -0.25

Step 1: Write out the positive version of the number

Step 2: convert number to negative
Flip the bits after the first 1



Step 3: next move the floating point to the first 10.

Step 4: work out exponent, -2 to the left

Step 5: store the mantissa

Spotting a normalised number.

Which of these are normalised numbers (8 bit scheme, 3 bit exponent, uses twos complement)

1. 00110 011
2. 01100 010

Answer:

The first one is not normalised but the second one is normalised.

They can both represent 3 decimal.

In the first example the binary number is 0.0110 and the exponent is 3 so move the binary point three places to the right. You get 11.0 which is 3 decimal

In the second example the binary number is 0.1100 with the exponent 2 so move the binary point two places to the right and you still get 11.00 which is once again 3 decimal. But note that you now have **2 binary bits after the point**, which indicates you have **more precision** available.

Notes for exam…

- Learn the approach we have used here for the exam.
- Don't check your answers using ChatGPT as it will use an alternative way of working out floating point and normalising a floating point. At the hardware level, there are different methods for storing floating-point binary numbers, including IEEE 754.
- Use https://www.advanced-ict.info/interactive/normalise.html to check your answers when doing revision.

Converting from denary to normalised binary floating point

- To convert a denary number to normalised binary floating point, first convert the number to fixed point binary.

Convert the number 14.25 to normalised floating point binary, using an 8-bit mantissa and a 4-bit exponent.

- In fixed point binary, 14.25 = 01110.010
- Remember that the first digit after the sign bit must be 1 in normalised form, so move the binary point 4 places left and increase the exponent from 0 to 4. The number is equivalent to 0.1110010×2^4
- Using a 4-bit exponent, 14.25 = 0 1110010 0100

If the denary number is negative, calculate the two's complement of the fixed point binary:

e.g. Calculate the binary equivalent of -14.25

14.25 = 01110.010

-14.25 = 10001.110 (two's complement)

In normalised form, the first digit after the point must be 0, so the point needs to be moved four places left.

$10001.110 = 1.0001110 \times 2^4 = 10001110\ 0100$

Example: Representing a negative number

- To represent the value -0.3125 in floating point form using 10-bit two's complement mantissa and 6-bit two's complement exponent in normalised form, convert the decimal to binary:

0.3125 = 0.010100000

Flip bits 1.101100000

Now normalise by floating the binary point to remove the leading 1s in the mantissa after the binary point:

$1.011000000 \times 2^{-1}$ or 1011000000 111111

Representing 123

Step 1: Convert 123 to binary

123 in 8 bit binary (Put a 0 as the MSB and fill with zeros after the LSB)

01111011.

Normalised binary:

00111011 Move point -7 to the left

Step 2: Determine mantissa and exponent

- **Mantissa**: 01111011
- **Exponent**: 7 to the right 0111

Representing 16.75

Step 1: Convert 16.75 to binary

16.75 in binary:

010000.11 (**Put a 0 as the MSB and fill with zeros after the LSB)**

Normalised binary:

0.1000011 Move floating point -5 to left

Step 2: Mantissa and Exponent
Mantissa: 01000011
Exponent: 5 → 0101

Representing -10.5

Step 1: Convert -10.5 to binary

1010.1 **Put a 0 as the MSB and fill with zeros after the LSB**

Positive 10.5 01010.100

Flip the bits 10101.100

Normalised binary:

1.0101100 Move -4 to the left

Step 2: Determine mantissa and exponent

Mantissa: 10101100 (two's complement for negative values).
Exponent: 4 to the right in 4-bit two's complement: 0100

Representing -12.875

Step 1: Convert -12.875 to binary

12.875 in binary: 01100.111 ***(Put a 0 as the MSB and fill with zeros after the LSB)***

Flip the bits: 10011.001

Normalized binary:

1.0011001 -4 to left

Step 2: Determine mantissa and exponent

Mantissa: 1.0011001 (two's complement for negative values).
Exponent: 4 in 4-bit two's complement: 0100

Steps for normalising a floating-point number

Normalise the following numbers, using an 8-bit mantissa and a 4-bit exponent

0.0000110 0011

Step 1: Work out the exponent: + 3

Step 2: Normalise the floating point

Move floating point 4 places to the right 00000.110

As we moved it right then to put it back it would be 4 to the left -4

Step 3 Work out the new exponent

Current value (+ or -) New Exponent +3 -4 = -1 1111

Key Points

- To normalise a floating point number, we ‘float’ the binary point to be in front of the first significant digit and adjust the exponent accordingly.
- We **normalise numbers** in this way to **maximise the accuracy** of the **value stored** and to **avoid multiple representation of the same number.**
- The accuracy of a floating point number depends on the number of digits in the mantissa.
- **More digits in the mantissa** means **fewer in the exponent**, meaning a **smaller range of values can be stored**.
- There is always a trade-off between the range and the accuracy when choosing the size of the mantissa and exponent in a floating point number.

Floating point addition and subtraction

Before looking at these operations in binary, we can gain an understanding of the principles involved in floating point arithmetic by looking at equivalent calculations in denary.

In denary, when adding two numbers involving decimal points, we first have to line up the points.

$$\begin{array}{rr} & 132.156 \\ + & 1.0318 \\ \hline & 133.1878 \end{array}$$

The rules for addition and subtraction can be stated as:

1. **Line up the points by making the exponents equal**
2. **Add or subtract the mantissas**
3. **Normalise the result**

In their "normalised form", the two numbers above would be represented as:

$.132156 \times 10^{3}$

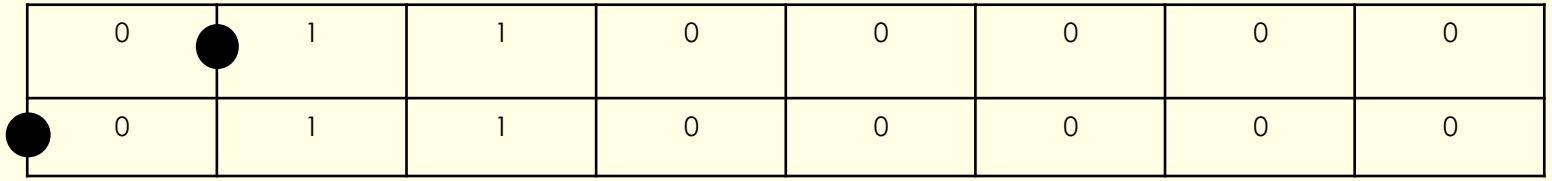
$.103180 \times 10^{1}$

Worked Example: Craig n Dave approach

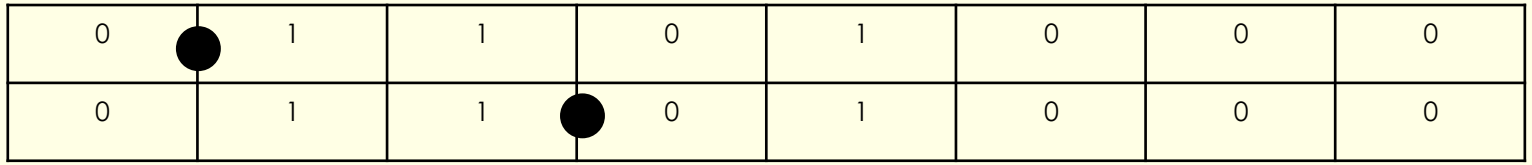
Add together the two binary numbers given below, leaving the result in **normalised** floating point binary form.

The rules for addition and subtraction can be stated as:

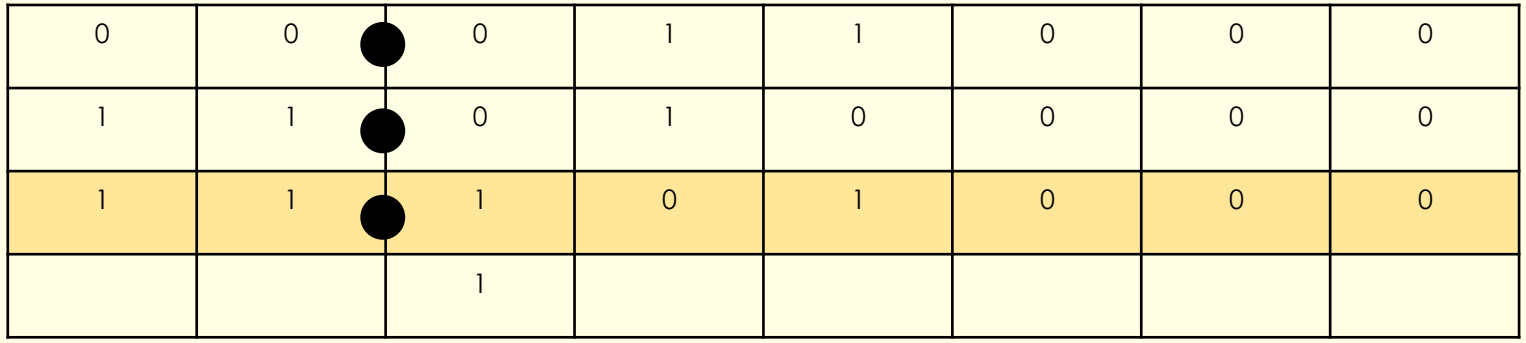
1. **Line up the points by making the exponents equal**
2. **Add or subtract the mantissas**
3. **Normalise the result**



-1 move 1 to left

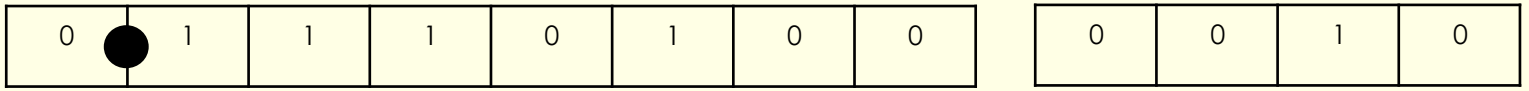


2 move 2 to right



Line up the points by making the exponents equal

Add



Normalise move 2 places to the left + 2

Worked Example – Alternative approach

- Convert the denary numbers 0.25 and 10.5 to normalised floating point binary form using an 8-bit mantissa and a 4-bit exponent.
- Add together the two normalised binary numbers, giving the result in normalised floating point binary form.

Step 1: The numbers in normalised form are:

0 • 1	0	0	0	0	0	0

1	1	1	1

0 • 1	0	1	0	1	0	0

0	1	0	0

Step 2: Write the mantissas with a binary point, and convert the exponents to denary, giving

0.1000000 exponent -1 and

0.1010100 exponent 4

Step 3: Make both exponents 4 and shift the binary points accordingly

0.0000010 (make the number *smaller* as you *increase* the exponent)

Move 5 places to the left

0.1010100

Step 4: Add the numbers, giving 0.1010110 exponent 4 (In this case it's already normalised)

Result is

0 • 1	0	1	0	1	1	0

0	1	0	0

Worked Example - Subtraction

Subtract the second of the two numbers given below from the first, giving the result in normalised floating point binary form.

Step 1: Convert the exponents to denary, giving

0.1000100 exponent 6 and

0.1000010 exponent 5

Step 2: Make both exponents 6 and shift the binary point of the second number accordingly

0.1000100 exp 6

0.0100001 exp 6 (make the number *smaller* as you *increase* the exponent)

Step 3: Find the twos complement of the second number

Step 4: Add the numbers

0.1000100

1.1011111

(1)0.0100011 exp 6 (ignore the carry)

Now normalise the number by moving the binary point right 1 place, which increases the number, and decrease the exponent by 1

Result is 

Practice Answer

Subtract the number 01001000 0011 from 01011000 0000, giving the result in normalised floating point binary form.

Step 1 Work out the exponents and move the floating points

a = 0.1001000 exponent 3

b = 0.1011000 exponent 0

(Check: a = 100.12 = 4.5 and b = 0.10112 = 0.6875

Task 2: Line up the exponents and work out the 2's complement for the second number
0100.1000
0000.1011 (Perform 2's complement to negative format)
1111.0101

0.6875 – 4.5 = -3.8125

Step 3 Add:
0100.1000
1111.0101

0011.1101

The result obtained was 1100.0011 = -8 + 4.1875 = -3.8125)

Step 4: Normalise this result by moving the binary point two places left, and adding 2 to the exponent

Result: 1.0000110 0010

Underflow and overflow

- Underflow occurs when a number is too small to be represented in the allotted number of bits. If, for example, a very small number is divided by another number greater than 1, underflow may occur and the result will be represented by 0.
- Overflow occurs when the result of a calculation is too large to be held in the number of bits allocated.

Check answer

Using floating point representation with 4 bits for the exponent and 4 bits for the mantissa, add together the following floating point binary numbers and write the answer as a **normalised floating point** number with 4 bits mantissa and 4 bit exponent.

0110 0010
Exponent 2

0100 0011
Exponent 3

Match exponent to **2**

0100 Move point 1 place to the right 1000 2^{3-1}

Mantissa 1110 Exponent 0010

Normalised

1.110 0.111 moved 1 to left 2^{2+1}

Moved to the right (- **from the exponent**)

Moved to the left (+ **to the exponent**)

0110
1000
111 0

Mantissa 0111 Exponent 0011

Show the subtraction of these two floating point binary numbers.
Both numbers are stored in a normalised floating point format, using 6 bits for the mantissa and 4 for exponent.
You should show your result in the same format.
Show your working out.

011010 0011 – 010010 0010

Step 1 work out Both exponents
0011 = 3
0010 = 2

Step 2 Both mantissas shifted
0.11010 3 to the right
0110.10
0.10010 2 to the right
010.010

Step 3 Align mantissas

0110.10
0010.01

Step 4 Binary subtraction:

0110.10

Convert 0010.01 - 2's complement
1101.11

Add:

0110.10
1101.11
+ 0100.01

Step 5 normalise

0100.01 Move 3
0011

Correct mantissa 010001

Correct exponent 0011

Comparison: Fixed-Point vs. Floating-Point Representation

Feature	Fixed-Point	Floating-Point
Definition	Represents numbers with a fixed number of decimal places.	Represents numbers using a sign, mantissa, and exponent for greater range.
Range	Limited range; cannot represent very large or very small values efficiently.	Much wider range, suitable for very large or very small numbers.
Precision	Fixed number of decimal places; can be more precise for small numbers.	Precision depends on mantissa size; can lose accuracy due to rounding errors.
Storage Size	Uses fewer bits, making it memory efficient.	Requires more bits
Speed & Efficiency	Faster for calculations, as operations are simpler.	Slower than fixed-point due to complex calculations (normalization, rounding, etc.).
Usage	Used in applications where precision is more important than range (e.g., financial systems).	Used in scientific computing, graphics, and engineering where a large range is needed.

Size of the mantissa will determine the **precision** of the number, and the size of the exponent will determine the **range** of numbers that can be held.