Learning Aims

- Explain why computers use binary to represent data and program instructions
- Determine the max number of states that can be represented by a binary pattern of a given length
- Describe how computers represent and manipulate unsigned integers (positive numbers)
- Convery between denary and 8 bit binary numbers (0-255)

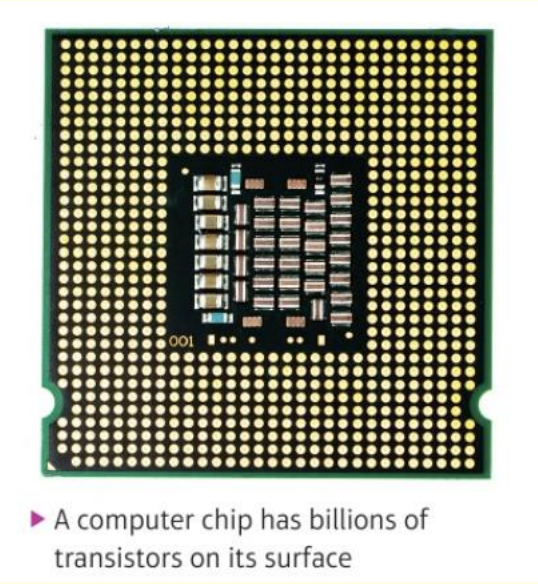
Why binary?

- **Binary** is a sequence of 0's and 1's, it can read by a computer processor, decoded and executed (BASE 2)

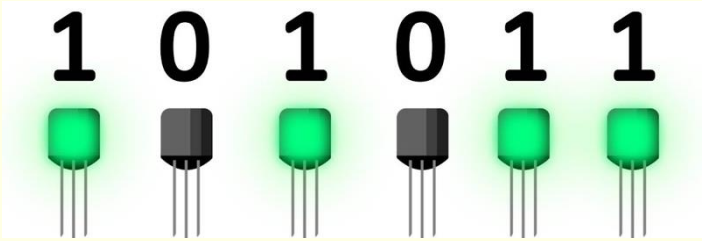
- Computers can store a sequence of 0 and 1's using tiny switches (on and off) known as **micro transistors** built into an **integrated circuit**.

- The circuits in a computer's processor are made up of billions of **transistors**.

- **A bit is a single binary digit. (0 or 1)**



A computer chip has billions of transistors on its surface



Binary Patterns

A single bit has only one state 1 or 0

2 bits produces 4 different binary patterns 00 01 10 11

3 bits = 8 binary patterns 000 001 011 111 011 101 110 111

The number of bit patterns can be expressed as:

$$2^{numberofbits}$$

Bytes & Nibbles

Byte is 8 bits

With 8 bits = 256 binary pattens = 2^8

Nibble is 4 bits

With 4 bits = 16 binary patterns = 2^4

Number of bits	2^n	Number of binary patterns	Bit patterns
1	2^1	2	0 1
2	2^2	4	00 01 10 11
3	2^3	8	000 001 010 011 100 101 110 111
4	2^4	16	0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

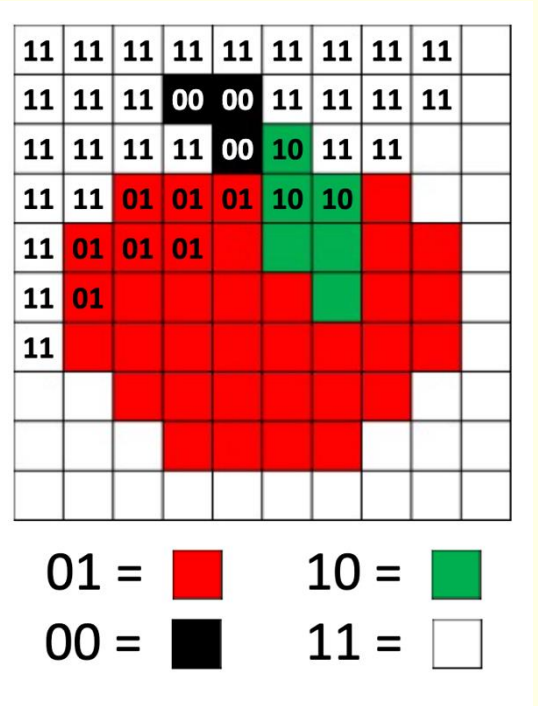
Using binary to represent data and program instructions

- 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.



Number Systems

- Computers can handle different formats of numbers
- Positive, negative numbers and decimal numbers
- GCSE Level: Only how positive and negative numbers are represented
- Unsigned numbers = only positive numbers

  For example: 1 byte can represent 0-255

- Signed numbers = range of numbers that include negative and positive

  For example: 1 byte can represent -128 to 127

Binary 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**

Alternative way of converting Denary to Binary

Worked example

Convert the denary number 213 to binary.

Denary number	÷ 2	Remainder	
213	106	1	LSB
106	53	0	
53	26	1	
26	13	0	
13	6	1	
6	3	0	
3	1	1	
1	0	1	MSB

read this way

The denary number 213 = 1101 0101 in binary.

Worked example

Convert the denary number 13 to an 8-bit binary number.

Denary number	÷ 2	Remainder	
13	6	1	LSB
6	3	0	
3	1	1	
1	0	1	
	0		
	0		
	0		
	0		MSB

read this way

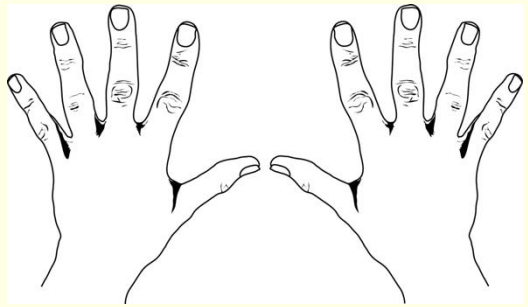
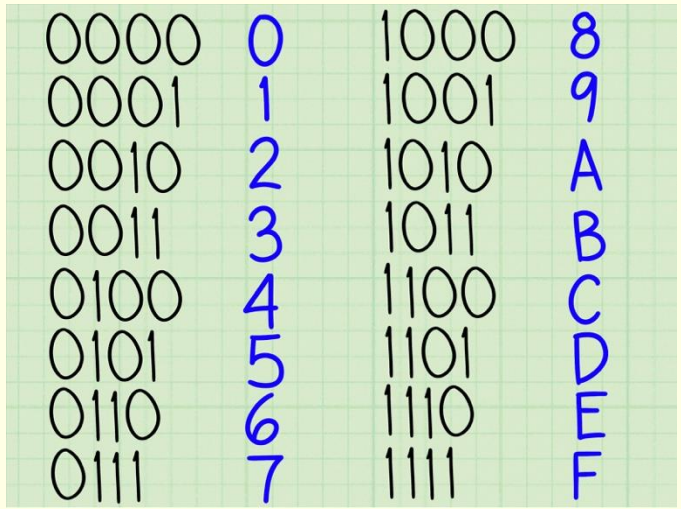
The denary number 13 = 0000 1101 in 8-bit binary.

Learning Objectives

Learn about the HEX number system and how it is used in computing

Be able to convert HEX to binary/ HEX to Decimal and vice versa

Number Systems

BASE 2	Base 10	Base 16
Binary 0 1 	Decimal (Denary) 0 1 2 3 4 5 6 7 8 9 	HEXADECIMAL 0 1 2 3 4 5 6 7 8 9 A B C D E F 

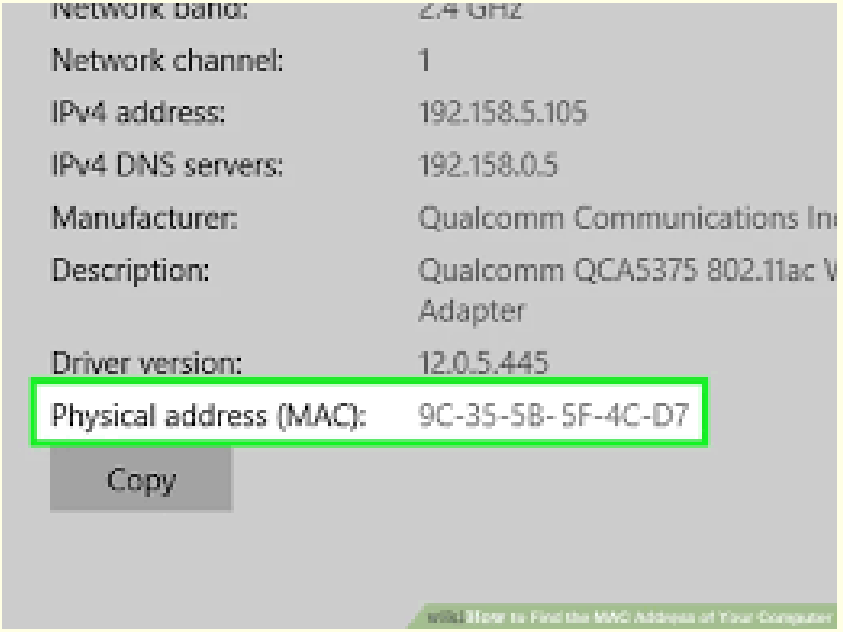
Why use Hexadecimal Numbers?

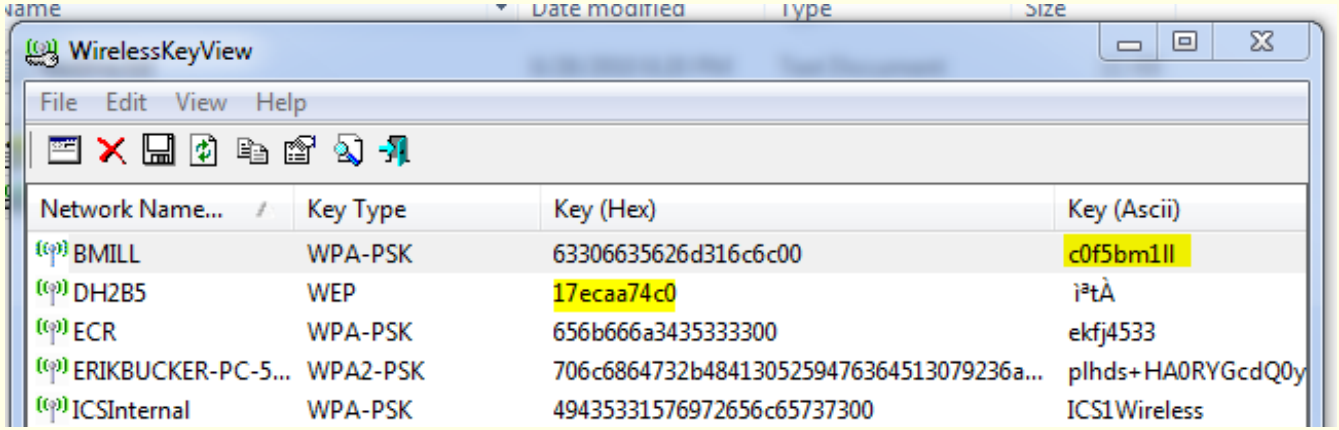
Hexadecimals are used by computer scientists for the following reasons:

Binary produces long strings and can be difficult to work with. Hex is shorter.

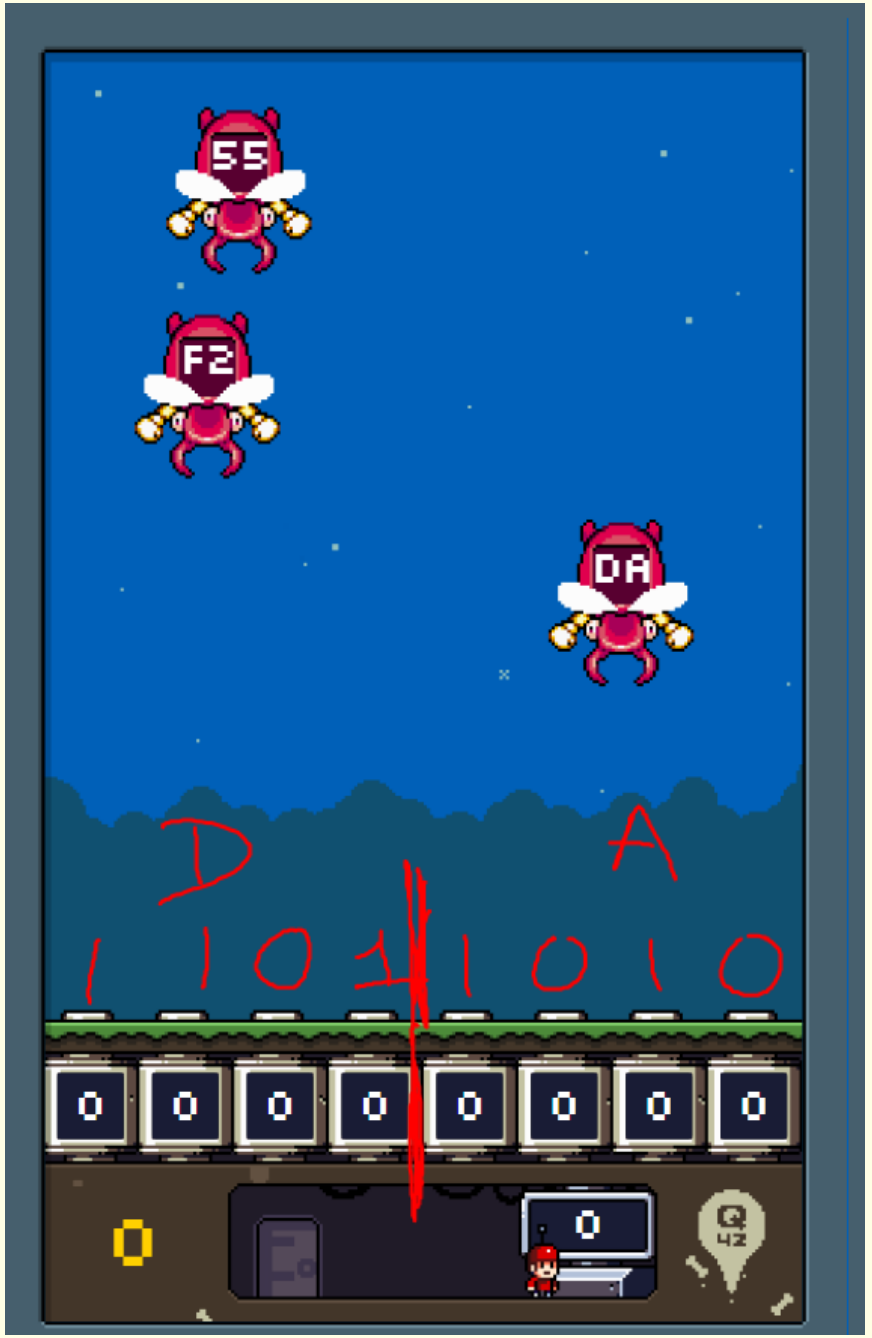
Hex can be easily converted to/from binary as there is 1 hex digit per nibble.

Hex is less susceptible to error.

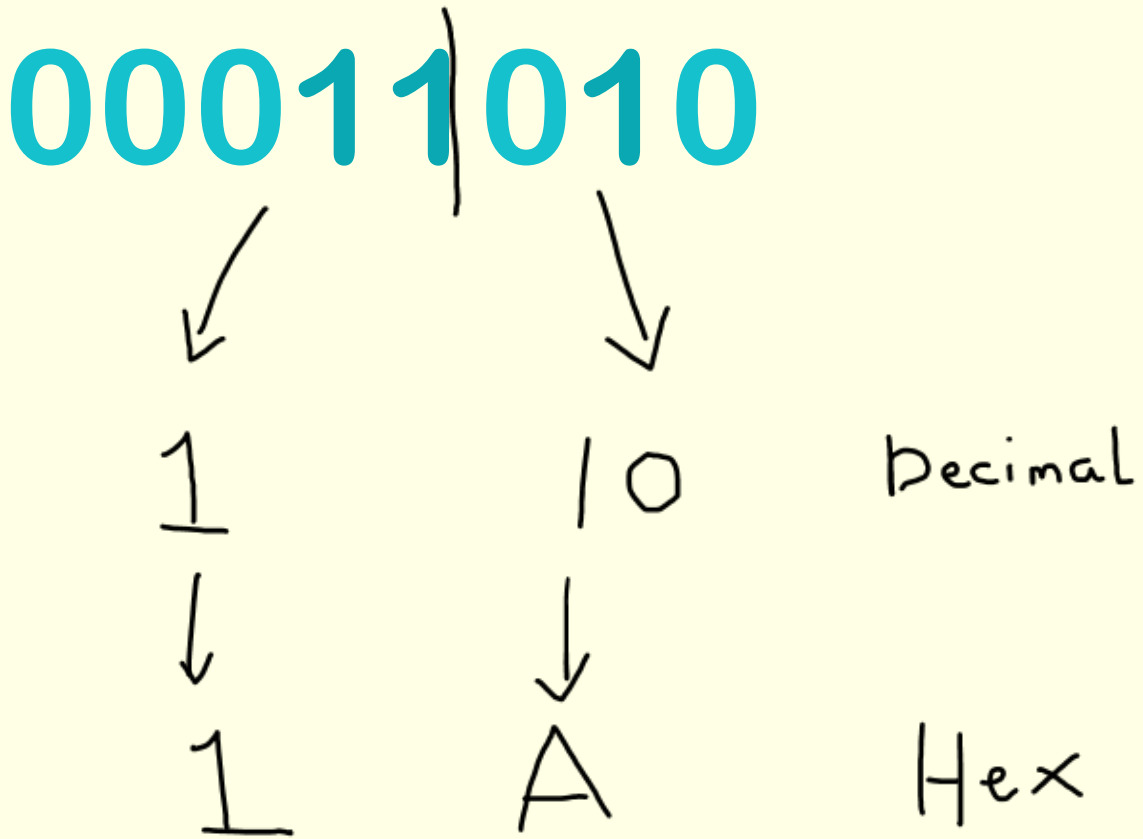




Decimal (Base 10)	Binary (Base 2)	Hexadecimal (Base 16)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

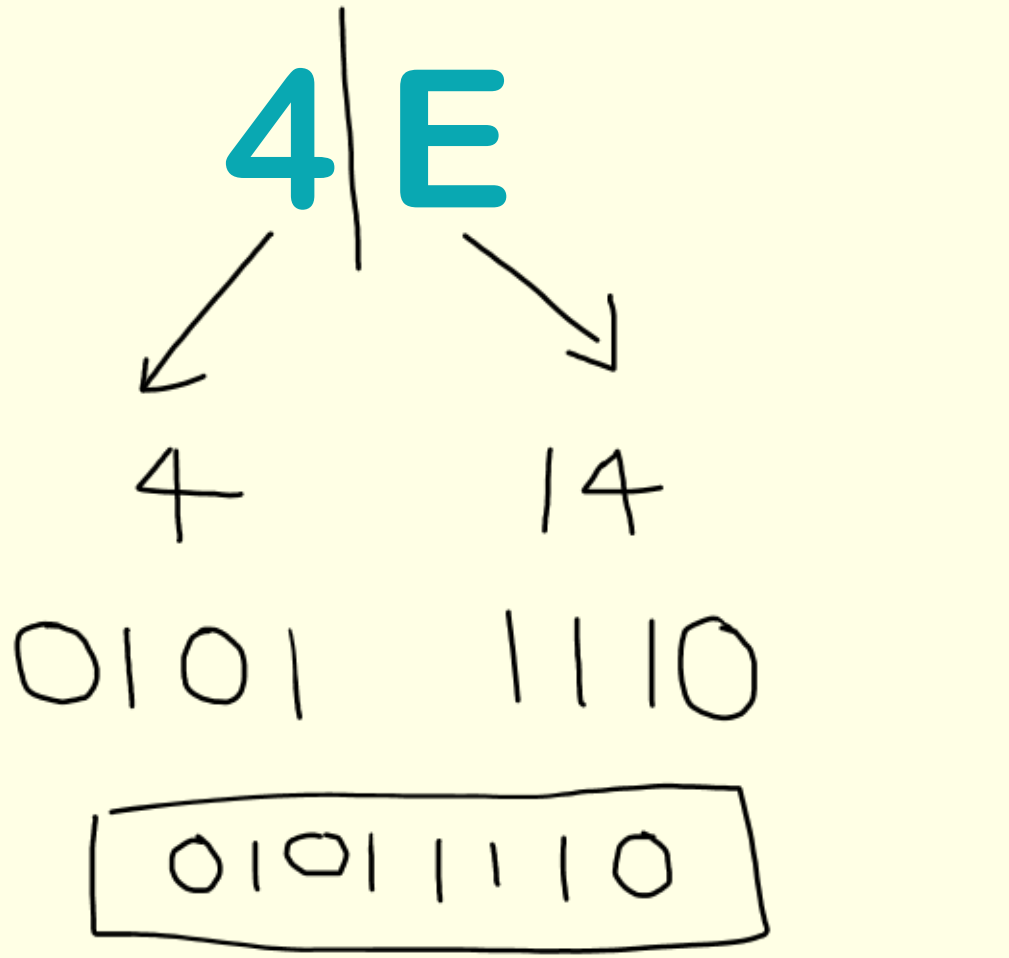


Working out the Hexadecimal



decimal	hexadecimal	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Working out the Denary



decimal	hexadecimal	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Working out the Hexadecimal

164

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

10 → A

4 → 4

A4

decimal	hexadecimal	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Learning objectives

- To add two positive binary numbers
- Explain the concept of overflow

Binary Addition Rules

	0		0		1		1
+	0	+	1	+	1		1
	0		1	1	0	+	1
						1	1

Adding 2 Binary numbers

			1		1		1	1
	+		1		0		1	0
		1	1	1	0		0	1

Worked Example

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

A quick check…

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

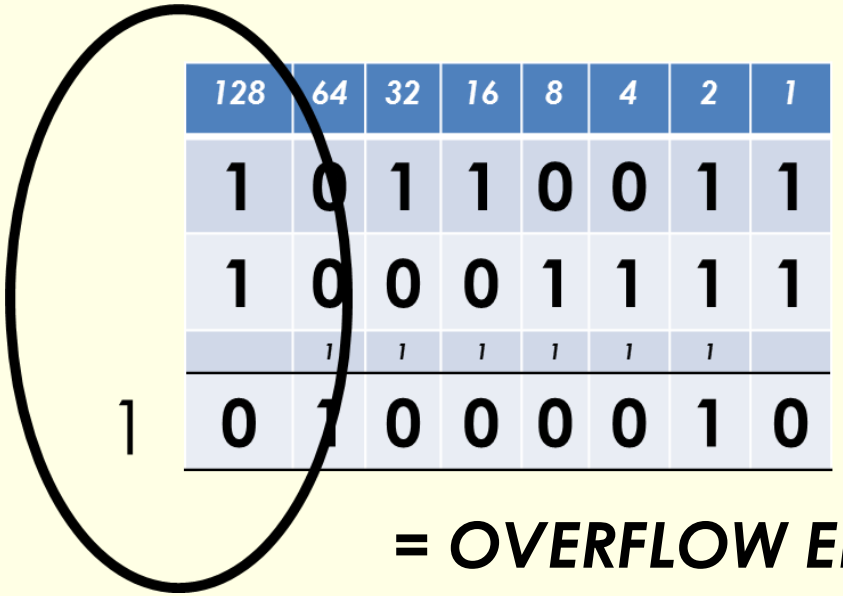
= 194

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

51 + 143 = 194

The problem with 9-bit answers

Because the largest number we can hold in a byte is 255, if we add two bytes together there is a chance that the answer will be greater than 255.



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

= OVERFLOW ERROR!

This answer will not be able to be held in a byte and so this causes an over flow error.

In modern computers, CPU's can hold much larger numbers so this is dealt with. *Ever heard of a 32 or 64 bit processor? – these can deal with larger binary numbers!*

However you do **need to know that overflow errors occur** when doing binary addition **when the answer is 9 bits** in length.

Learning Objectives

- Describe how computers represent 2's complement signed numbers
- Convert between denary and 8 bit binary numbers (-128 to +127)

Two's complement

- Two's complement is a much more useful way of representing binary numbers as it allows you to represent negative numbers
- With 8 bits you can represent −128 to 127
- Because the MSB is used to represent the sign (+ or -)
- If MSB is a 1 = negative number
- If MSB is a 0 = positive number
- This method requires the MSB place value to be negative value

Number of bits	Value of MSB	Range
4	−8	−8 to +7
8	−128	−128 to +127
16	−32,768	−32,768 to +32,767

Place Value	-128	64	32	16	8	4	2	1
	0	0	0	0	0	0	1	1

Representing numbers using 2's complement

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

MSB

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

16 + 2 = 18

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

MSB

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

-128 + 18 = -110

Worked Example

- For example in 8 bits the MSB will be -128

Place Value	-128	64	32	16	8	4	2	1
	1	0	0	0	0	0	0	1

-128 + 1 = -127

Here are two 8-bit two's complement numbers. Seven of their eight bits are the same, but they have different MSBs.

Positive number	0	1	1	0	1	1	0	1
Negative number	1	1	1	0	1	1	0	1

The one with a 0 is a positive number.
The denary equivalent is: 0 + 64 + 32 + 8 + 4 + 1 = +109.

The one with a 1 is a negative number.
Its denary equivalent is (–128) + 64 + 32 + 8 + 4 = –20.

Worked example

Convert the two's complement binary number 1110 0100 to denary.

Place values	-2^7	$+2^6$	$+2^5$	$+2^4$	$+2^3$	$+2^2$	$+2^1$	$+2^0$
	−128	**+64**	**+32**	**+16**	**+8**	**+4**	**+2**	**+1**
	1	1	1	0	0	1	0	0

The MSB is 1, indicating that it is a negative number.

The denary equivalent is: (−128) + 64 + 32 + 4 = (−128) + 100 = −28

Two's complement

- 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

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

Place Value	-128	64	32	16	8	4	2	1
Binary number	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:
- Finally invert everything after this, so all the 1s become 0s and vice versa:

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

Adding together two two's complement numbers

Add together these two 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.

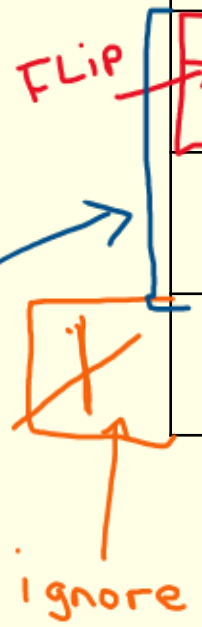
Why do you think this is the case?

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

64-12

	128	64	32	16	8	4	2	1
Step 1 work out + 12	0	0	0	0	1	1	0	0
Step 2 work out – 12	1	1	1	1	0	1	0	0
Step 3 work out 64	0	1	0	0	0	0	0	0
Step 4 Add together -12 and 64	0	0	1	1	0	1	0	0



Learning Objectives

- To understand how to use a binary shift
- Binary numbers can be shifted to multiply or divide them

Binary Shifts

- **Binary shifts** are used to **move all the bits** in a binary pattern left or right
- Logical shifts treat all the bits of binary pattern in the same way
- To multiply binary numbers, the bits are shifted to the left
- To divide binary numbers, the bits are shifted to the right

A logical shift left (LSL) moves each bit left by n positions

Worked example

Perform a logical shift left of two positions on the binary pattern 0001 0100.

Move each bit two positions to the left. Discard the two leftmost bits. Fill up the empty spaces on the right with 0s.

Before shift	0	0	0	1	0	1	0	0
After shift	0	1	0	1	0	0	0	0

The result is 0101 0000.

- Left shift is used to multiply

A logical shift right(LSR) moves each bit right by n positions

Worked example

Perform a logical shift right of three positions on the binary pattern 0001 1000.

Move all the bits three positions to the right. Discard the three rightmost bits. Fill up the empty spaces on the left with 0s.

Before shift	0	0	0	1	1	0	0	0
After shift	0	0	0	0	0	0	1	1

The result is 0000 0011.

- right shift is used to divide

Loss of precision

Sometimes a binary shift produces an imprecise result. This can happen when the pattern is interpreted as numbers. For example, if we shift 0010 0001 (+33) by one position right (a division by 2) the rightmost bit is lost, producing a result of +16, which is imprecise.

Place values	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Before shift	0	0	1	0	0	0	0	1
After shift	0	0	0	1	0	0	0	0

Figure 2.1.3 In this case shifting right produces an imprecise result

Loss of precision occurs because one or more of the right-most bits are discarded during the shift.

Rules for binary shift

Multiply or divide by	Shifts
$2 = 2^1$	1
$4 = 2^2$	2
$8 = 2^3$	3
$16 = 2^4$	4
$32 = 2^5$	5
$64 = 2^6$	6
$128 = 2^7$	7

Arithmetic shift – Right Shift

- An arithmetic shift is a binary shift used with 2's complement
- An arithmetic shift right (ASR) moves each bit right by n positions.
- **Fill rule:**
  - The **most significant bit (MSB)** — the *sign bit* — is copied into the empty positions on the left.
  - If MSB = 0 → fill with 0s (number is positive).
  - If MSB = 1 → fill with 1s (number is negative).

- This preserves the **sign** of the number.

Worked example

Perform an arithmetic shift right of one position on the binary pattern 1011 0000. Move all the bits one position to the right. Discard the rightmost bit. Put a 1 in the empty space on the left.

Before shift	1	0	1	1	0	0	0	0
After shift	1	1	0	1	1	0	0	0

The result is 1101 1000.

Arithmetic shifts – Left

Before shift	1	1	0	1	0	1	0	1
After shift	0	1	0	1	0	1	0	0

Same as left logical shift – fill with 0's

However a left shift on a 2's complement will lose the MSB (negative sign)