9618 Syllabus Content:

1.1 Data Representation

Candidates should be able to:

- Show understanding of binary magnitudes and the difference between binary prefixes and decimal prefixes
- Show understanding of the basis of different number systems
- Show understanding of the basis of different number systems
- Describe practical applications where Binary Coded Decimal (BCD) and Hexadecimal are used
- Show understanding of and be able to represent character data in its internal binary form, depending on the character set used
- Understand the difference between and use:
  - kibi and kilo
  - mebi and mega

gibi and giga

  - tebi and tera
- Use the binary, denary, hexadecimal number bases and Binary Coded Decimal (BCD) and one's and two's complement representation for binary numbers
- Convert an integer value from one number base / representation to another
- Using positive and negative binary integers Show understanding of how overflow can occur
- Familiar with ASCII (American Standard Code for Information Interchange), extended ASCII and Unicode. Students will not be expected to memorise any particular character codes.

Denary Number System:

We know decimal or denary number system has (base 10).This uses digits 0 to 9 and has place values below

10000	1000	100	10	units
3	1	4	2	1

Binary number system:

The **binary** system on computers uses combinations of 0s and 1s and has (base 2).

128	64	32	16	8	4	2	1
(2^7)	(2^6)	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)

A typical binary number would be:

1 1 1 0 1 1 1 0

Binary place values

You can also break a **binary** number down into place-value columns, but each column is a power of two instead of a power of ten.
For example, take a binary number like **1001**. The columns are arranged in multiples of 2 with the binary number written below: **PLACE VALUE**

Eights 8s (2^3)	Fours 4s (2^2)	Twos 2s (2^1)	Ones 1s (2^0)
1	0	0	1

By looking at the place values, we can calculate the equivalent denary number.

That is: $(1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8+0+0+1$
$(1 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1) = 8 + 1$
$= 9$

Converting binary to denary

To calculate a large **binary** number like **10101000** we need more place values of multiples of 2.

- $2^7 = 128$
- $2^6 = 64$
- $2^5 = 32$
- $2^4 = 16$
- $2^3 = 8$
- $2^2 = 4$
- $2^1 = 2$
- $2^0 = 1$

In **denary** the sum is calculated as:

$(1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 168$
$(1 \times 128) + (0 \times 64) + (1 \times 32) + (0 \times 16) + (1 \times 8) + (0 \times 4) + (0 \times 2) + (0 \times 1) = 128 + 32 + 8 = 168$

The table below shows denary numbers down the left with their equivalent binary numbers marked out below the base 2 columns. Each individual column in the table represents a different **place value** equivalent to the base 2 powers

Computers(9618) with Majid

Convert between denary

	Binary pattern							
	Place value 128	Place value 64	Place value 32	Place value 16	Place value 8	Place value 4	Place value 2	Place value 1
Denary number								
0								0
1								1
2							1	0
3							1	1
4						1	0	0
5						1	0	1
6						1	1	0
7						1	1	1
8					1	0	0	0
9					1	0	0	1
10					1	0	1	0
....								
255	1	1	1	1	1	1	1	1

Converting denary to binary: Method 1

There are two methods for converting a **denary** (base 10) number to **binary** (base 2). This is method one.

Divide by two and use the remainder

Divide the starting number by 2. If it divides evenly, the binary digit is 0. If it does not - if there is a remainder - the binary digit is 1.

A method of converting a denary number to binary

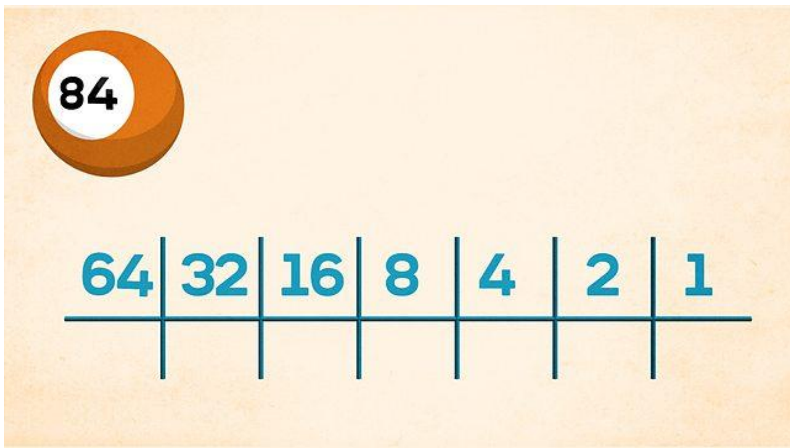
Worked example: Denary number 83

- **83** ÷ 2 = 41 remainder **1**
- **41** ÷ 2 = 20 remainder **1**
- **20** ÷ 2 = 10 remainder **0**
- **10** ÷ 2 = 5 remainder **0**
- **5** ÷ 2 = 2 remainder **1**
- **2** ÷ 2 = 1 remainder **0**
- **1** ÷ 2 = 0 remainder **1**

Put the remainders in **reverse** order to get the final number: **1010011**.

Converting denary to binary: Method 2

There are two methods for converting a **denary** (base 10) number to **binary** (base 2).
This method uses **Place Values**



Method:2 - Converting a denary number to binary

Worked example: Denary number 84

We need to check which numbers place values can be added to make 84. We will put 1 under the numbers to e added and 0 under the numbers which are not added.

1. We select Place value 64 so we put 1 under it.
2. We select place value 16 and put 1 under it
3. We selected place value 4 and put 1 under it.
4. Adding 64+16+4 gives us 84 so our number becomes:

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

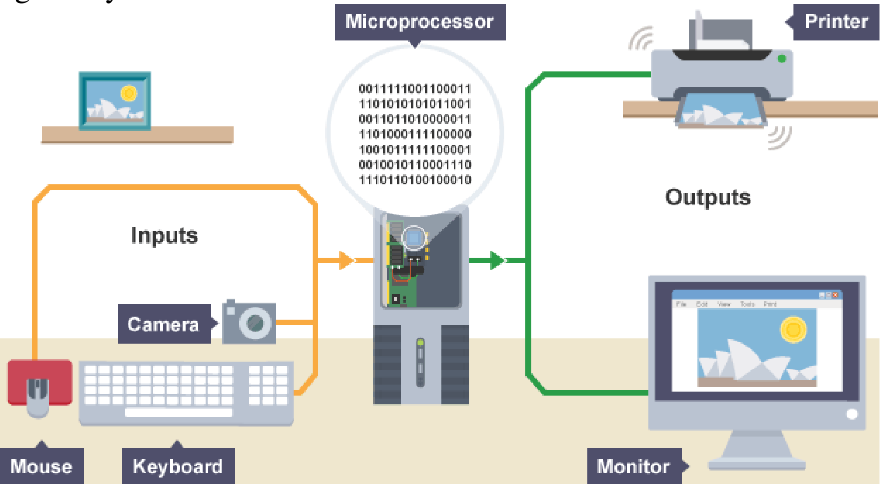
Result: **84** in denary is equivalent to **1010100** in binary.

Bits and binary:

Computers use **binary** - the digits 0 and 1 - to store data. A binary digit, or **bit**, is the smallest unit of data in computing. It is represented by a 0 or a 1. **Binary numbers are made up of binary digits (bits)**, e.g. the binary number **1001**.

The circuits in a computer's processor are made up of billions of **transistors**. A transistor is a tiny switch that is activated by the electronic signals it receives.
The digits 1 and 0 used in binary reflect the on and off states of a transistor.

All **software**, music, documents, and any other information that is processed by a computer, is also stored using binary.



Bits and bytes

Bits can be grouped together to make them easier to work with. **A group of 8 bits is called a byte**.

Other groupings include:

- **Nibble** - 4 bits (half a byte)
- **Byte** - 8 bits
- **Kilobyte** (KB) - 1024 bytes (or 1024 x 8 bits) $= 2^{10}$
- **Megabyte** (MB) - 1024 kilobytes (or 1048576 bytes) $= 2^{20}$
- **Gigabyte** (GB) - 1024 megabytes $= 2^{30}$
- **Terabyte** (TB) - 1024 gigabytes $= 2^{40}$
- **Petabyte (PB) -**1024 Terabytes $= 2^{50}$
- **Exabyte (EB**) -1024 Petabytes $= 2^{60}$
- **Zettabyte (ZB**)- 1024 Exabytes $= 2^{70}$
- **Yottabyte (YB**) 1024 Zettabytes $= 2^{80}$

The IEC convention for computer internal memories (including RAM) becomes:

- **1 kilobyte = 1000 byte**
- **1 megabyte = 1000000 bytes**
- **1 gigabyte = 1000000000 bytes**
- **1 terabyte = 1000000000000 bytes and so on.**

VS

- **1 kibibyte (1 KiB) = 1024 bytes**
- **1 mebibyte (1 MiB) = 1048576 bytes**
- **1 gibibyte (1 GiB) = 1073741824 bytes**
- **1 tebibyte (1 TiB) = 1099511627776 bytes and so on.**

Computer Science (9618) with Majid Tahir

However, the IEC terms are not universally used and we still use the more conventional terms shown above. This also ties up with the Cambridge International Examinations computer science syllabus which uses the same terminology as in example above.

Binary Addition (unsigned number)

Adding **binary** numbers is similar to adding **denary** numbers.

Example: Adding the binary numbers 011 and 100

Write the numbers out using the column method. Start from the right, and simply add the numbers.

```
  011
+ 100
  ---
  111
```

111 is 7 if converted back to denary.

Example: Adding two 1s in the same column

Sometimes a binary addition will require you to carry over values into the next highest place-value column, eg when finding the sum of the binary numbers 0010 and 0111:

There is a clash when adding two ones in the same column. In binary, 1+1 is 10 - it has to become 0 with 1 carried over.

```
  0 0 1 0
+ 0₁1₁1 1
  -------
  1 0 0 1
```

1001 is 9 if converted back to denary. 2 + 7 = 9 in denary.

+ve and –ve binary numbers (signed numbers)

When computer stores binary numbers, we have to differentiate +ve binary numbers from –ve binary numbers. Unfortunately (-) or (+) sign cannot be displayed rather only **0/1** can be used in binary.

Rules of +ve and –ve binary numbers

- Positive binary number always starts with 0 in MSB (Most significant bit)
- Negative binary number always starts with 1 in MSB (Most significant bit)

e.g +ve number: **01010010**
-ve number: **10010110**

	-128	64	32	16	8	4	2	1	Place Values
Number becomes	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	so number is **0**+64+16+2 = **82**

Sign bit (0) **+ve part of number** (1010010)

Negative binary number always starts with 1 in MSB (Most significant bit)

	-128	64	32	16	8	4	2	1	Place Values
Number becomes	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	so number is **-128**+16+4+2 = -106

Sign bit (1) **+ve part of number** (0010110)

Binary addition and subtraction

Up until now we have assumed all binary numbers have positive values. There are a number of methods to represent both positive and negative numbers. We will consider:

- **One's complement**
- **Two's complement.**

In **one's complement**, each digit in the binary number is inverted (in other words, 0 becomes 1and 1 becomes 0). For example,

Step 1

65 = 01000001 in binary

Step 2: invert 1 to 0 and 0 to 1 we get: 01000001 to its one's complement as below:

01000001 = 10111110

In **two's complement**,binary digit 1 is added to one's compliment

Step 3: Convert 10111110 Binary to its two's complement by

adding 1 to the one's complement.

```
 10111110
  +     1
--------------
 10111111  =  Two's complement
```

Two's compliment of a positive number will make it a negative number

-128	64	32	16	8	4	2	1	Place values
1	0	1	1	1	1	1	1	= Two's complement

= -128 + 32+16+8+4+2+1 = -65

Computers(9618) with Majid Ta

10111111 is - 65 in binary. We know this it true because if we add

01000001 (+65) to 10111111b (-65) and *ignore the carry bit*, the sum is 0, which is what we obtain if we add +65 + (-65) = 0.

```
   01000001    +65
 + 10111111    -65
---------------------
 1 00000000      0  denary
 ^
```

Ignore the carry bit for now. What matters is that original number of bits (D7-D0) are all 0.

Two's complement sums:

Using two's complement, the CPU can perform arithmetic using binaryaddition. For example:

-7 + 7 in two's complement binary would be calculated as:

```
    1 0 0 1
 + ₁0₁1₁1₁1
 -----------
  1 0 0 0 0
```

In two's complement, if the final result overflows the remaining carry number is simply discarded. For example:

-3 + 4 in two's complement binary would be calculated as:

```
    1 1 0 1
 + ₁0₁1 0 0
 -----------
  1 0 0 0 1
```

Methods for converting a negative number expressed in two's complement form to the corresponding denary number

Consider the two's complement binary number **10110001**.

Method 1:

- Convert to the **1's compliment** gives **01001110** and keep the minus sign with it
- Converting to **two's complement** gives us **01001111**.
- You ignore the leading zero in MSB as it is not a positive number and apply one of the methods to convert the remaining binary to denary which gives 79.
- You add the minus sign to give –79.

Method 2: Sum the individual place values but treat the most significant bit as a negative value

You follow the approach illustrated in Table 1.02 to convert the original binary number 10110001 as follows:

Computers(9618) with Majid Tahir

Place value	-2^7 = −128	2^6 = 64	2^5 = 32	2^4 = 16	2^3 = 8	2^2 = 4	2^1 = 2	2^0 = 1
Digit	1	0	1	1	0	0	0	1
Product	−128	0	32	16	0	0	0	1

You now add the values in the bottom row to get −79.

Binary Subtraction

To carry out subtraction in binary, we convert the number being subtracted to its negative equivalent using two's compliment and then add the two numbers.

1 Convert the two numbers into binary:

95 = 0 1 0 1 1 1 1 1

68 = 0 1 0 0 0 1 0 0

2 Find the two's complement of 68:

invert the digits:	1	0	1	1	1	0	1	1	
add 1:								1	
which gives:	1	0	1	1	1	1	0	0	= −68

3 Add 95 and −68:

	−128	64	32	16	8	4	2	1
	0	1	0	1	1	1	1	1
					+			
	1	0	1	1	1	1	0	0
					=			
1	0	0	0	1	1	0	1	1

The additional ninth bit is simply ignored leaving the binary number 0 0 0 1 1 0 1 1 (denary equivalent of 27, which is the correct result of the subtraction).

Overflow

A CPU with a capacity of 8 bits has a capacity of up to 11111111 in binary. **If one more bit was added there would be an overflowerror.**

Sorry, this clip is not available in your region or territory.

An explanation of binary overflow errors

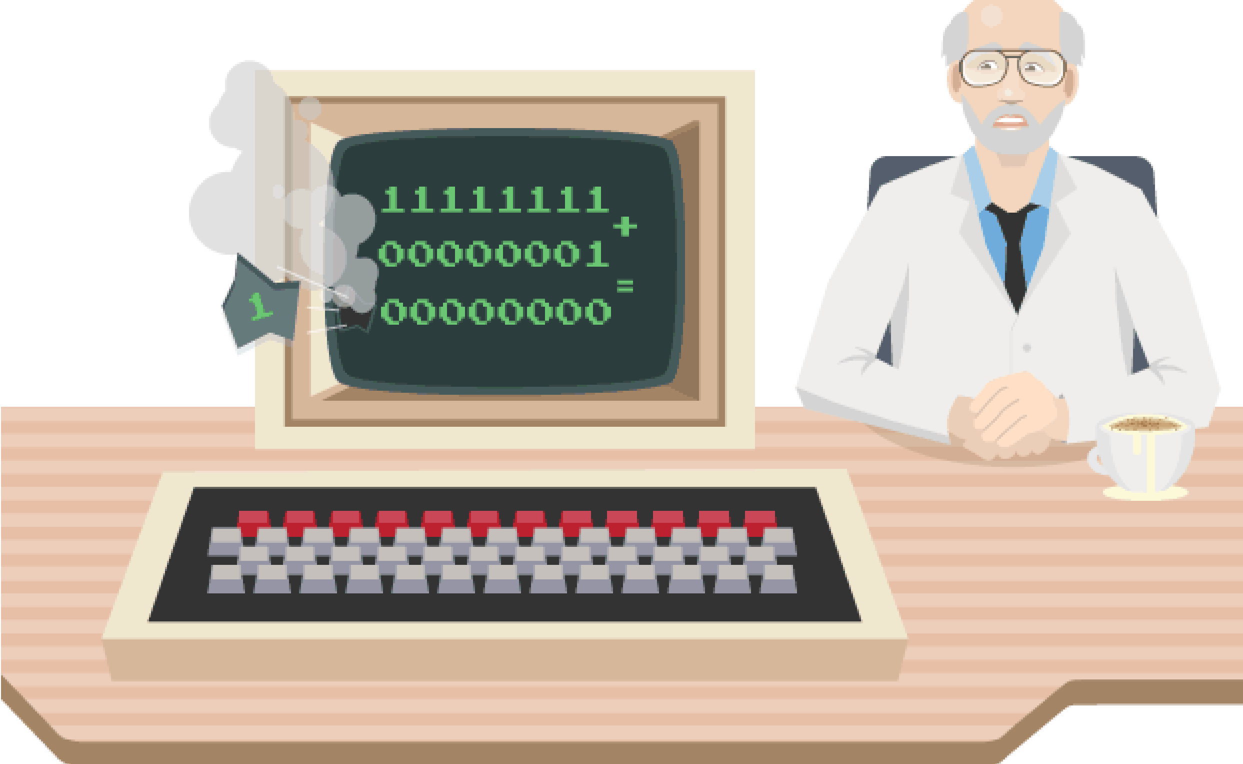
Download Transcript

Example: 8-bit overflow

An example of an 8-bit overflow occurs in the binary sum 11111111 + 1 (denary: 255 + 1).

```
   1 1 1 1 1 1 1 1
+  0 0 0 0 0 0 0 1
 1 1 1 1 1 1 1 1
 -----------------
 1 0 0 0 0 0 0 0 0
```

The total is a number bigger than 8 digits, and when this happens the **CPU drops the overflow digit** because the computer cannot store it anywhere, and **the computer thinks 255 + 1 = 0**.



Overflow errors happen when the largest number that a **register** can hold is exceeded. The number of bits that it can handle is called the **word size**.
Most CPUs use a much bigger word size than 8 bits. Many **PCs** have a 64-bit CPU. A 64-bit CPU can handle numbers larger than 18 quintillion (18,446,744,073,709,551,615 to be precise).

Negative numbers: Sign and magnitude

Computers sometimes need to work with **negative numbers**.
Integers can be encoded so that they can be positive or negative numbers. Integers that can be either positive or negative are **signed**numbers.
One way to represent negative numbers is through **sign and magnitude**. In this method, the **bit** at the far left of the bit pattern - the **sign bit** - indicates whether the number is positive or negative. The rest of the bits in the pattern store the size of the number (called its **magnitude**).
For example, with an 8-bit pattern, the first bit would be used to indicate positive or negative.**0** can indicate a **positive** number and a **1** can indicate a **negative** number. The other seven bits would be used to store the actual size of the number.

For example, **1**0001001 could represent -9:

the first bit, **1**, indicates a **negative** number

the other seven bits indicate the number, 0001001 = 9

The smallest possible number using this method of representation is -127 (or 11111111) and the largest possible number is +127 (or 01111111).

Negative numbers: Two's complement

Another method of representing **signed** numbers is **two's complement**. Most computers use this method to represent negative numbers. This method can be more effective when performing mathematical operations like adding and subtracting.

With two's complement, the **bit** at the far left of the bit pattern - the **most significant bit** or **MSB** - is used to indicate positive or negative and the remaining bits are used to store the actual size of the number. Positive numbers always start with a 0.

Four-bit, positive, two's complement numbers would be 0000 = 0, 0001 = 1, up to 0111 = 7. The smallest positive number is the smallest **binary**value.

Negative numbers always start with a **1**. The smallest negative number is the largest binary value. **1**111 is -1, **1**110 is -2, **1**101 is -3, etc down to **1**000 which represents -8.

Using two's complement for negative numbers

1. Find the positive binary value for the negative number you want to represent.
2. Add a 0 to the front of the number, to indicate that it is positive.
3. Invert or find the complement of each bit in the number.
4. Add 1 to this number.

Examples

Find -1 using two's complement numbers

1. 1 = 001
2. Adding 0 to the front becomes **0001**
3. 'Inverted' becomes **1110 (one's compliment)**
4. Add 1 = **1111** (**-8 + 4 + 2 + 1 = -1**)

Find -4 using two's complement numbers

1. 4 = 100
2. Adding 0 to the front becomes **0100**
3. 'Inverted' becomes **1011**
4. Add 1 = **1100** (**-8 + 4 = -4**)

This table shows the two's complement set for 4-bit numbers.

Denary	4-bit binary
-8	1000

Computers(9618) with Majid Tahir

Denary	4-bit binary
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Conversion of –Ve Denary number to Binary:

What is – 65 $_{10}$ in binary?

Two's complement allows us to represent signed negative values in binary, so here is an introductory demonstration on how to convert a negative decimal value to its negative equivalent in binary using two's complement.

Hexadecimal Number System:

We often have to deal with large positive binary numbers. For instance, consider that computers connect to the Internet using a Network Interface Card (NIC). Every NIC in the world is assigned a unique 48-bit identifier as an Ethernet address. The intent is that no two NICs in the world will have the same address. A sample Ethernet address might be:
000000000100011101011110011111111001001000110110

- Fortunately, large binary numbers can be made much more compact—and hence easier to work with—if represented in base-16, the so-called hexadecimal number system.
- You may wonder: Binary numbers would also be more compact if represented in base-10—why not just convert them to decimal?
- The answer, as you will soon see, is that converting between binary and hexadecimal is exceedingly easy—much easier than converting between binary and decimal.

The Hexadecimal Number System

The base 16 hexadecimal has 16 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F). Note that the single hexadecimal symbol A is equivalent to the decimal number 10, the single

Just as with decimal notation or binary notation, we again write a number as a string of symbols, but now each symbol is one of the 16 possible hexadecimal digits (0 through F). To interpret a hexadecimal number, we multiply each digit by the power of **16** associated with that digit's position.

For example, consider the hexadecimal number 1A9B. Indicating the values associated with the positions of the symbols, this number is illustrated as:

Hexadecimal Place value:

The **hexadecimal** system is very closely related to the binary system. Hexadecimal (sometimes referred to as simply hex) is a base 16 system with the weightings:

1048576	65536	4096	256	16	1
(16^5)	(16^4)	(16^3)	(16^2)	(16^1)	(16^0)

Because it is a system based on 16 different digits, the numbers 0 to 9 and the letters A to F are used to represent hexadecimal digits.

A = 10, B = 11, C = 12, D = 13, E = 14 and F = 15.

- The one main disadvantage of binary numbers is that the binary string equivalent of a large decimal base -10 number, can be quite long.
- When working with large digital systems, such as computers, it is common to find binary numbers consisting of 8, 16 and even 32 digits which makes it difficult to both read and write without producing errors especially when working with lots of 16 or 32-bit binary numbers.
- One common way of overcoming this problem is to arrange the binary numbers into groups or sets of four bits (4-bits).
- hese groups of 4-bits use another type of numbering system also commonly used in computer and digital systems called Hexadecimal Numbers.

REPRESENTING INTERGERS AS HEXADECIMAL NUMBERS:

The base 16 notational system for representing real numbers. The digits used to represent numbers using hexadecimal notation are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

—H denotes hex prefix.

Examples:

(i) $28_{16} = 28_H = 2 \times 16^1 + 8 \times 16^0 = 40$

$= 32 + 8 = 40$

(ii) **2 F16 =** $2F_H = 2 \times 16 + 15 \times 1 = 47$

(iii) **BC1216 =** $BC12_H = 11 \times 16^3 + 12 \times 16^2 + 1 \times 16^1 + 2 \times 16^0 = 48146$

Hexadecimal Numbers in Computing

There are two ways in which hex makes life easier.

- The first is that it can be used to write down very large integers in a compact form.
- For example, $(\mathbf{AD45})_{16}$ is shorter than its decimal equivalent $(\mathbf{44357})_{10}$ and as values increase the difference in length becomes even more pronounced.

Converting Binary Numbers to Hexadecimal Numbers.

Let's assume we have a binary number of: 01010111

The binary number is 01010111

We will break number into 4 bits each as

0101 0111

Then we will start with the right side 4 bits

Starting from extreme right number

for 0101	for 0111
$0X2^3+1X2^2+0X2^1+1X2^0$	$0X2^3+1X2^2+1X2^1+1X2^0$
0X8+1X4+0X2+1X1	0X8+1X4+1X2+1X1
0+4+0+1=5	0+4+2+1=7
5	7

So Hexadecimal number is 57

Computers(9618) with Majid Tahir

Converting Hexadecimal Numbers to Binary Numbers

To convert a hexadecimal number to a binary number, we reverse the above procedure. We separate every digit of hexadecimal number and find its equivalent binary number and then we write it together.

Example 1.2.4

To convert the hexadecimal number 9F216 to binary, each hex digit is converted into binary form.

$$9\ F\ 2_{16} = (1001\ 1111\ 0010)_2$$

9 =1001 F=1111 2=0010

So Binary equivalent of Hexadecimal number is: 9F2= 100111110010

Problems 1.2.6

Convert hexadecimal 2BF9 to its binary equivalent.

Convert binary 110011100001 to its hexadecimal equivalent. (Below is working area)

Converting a Hexadecimal Number to a (Denary) Decimal Number

To convert a hexadecimal number to a decimal number, write the hexadecimal number as a sum of powers of 16. For example, considering the hexadecimal number 1A9B above, we convert this to decimal as:

1	**A**	**9**	**B**
16^3	16^2	16^1	16^0

1A9B = 1(16^3) + A (16^2) + 9(16^1) + B (16^0)

= 4096 + 10(256) + 9(16) + 11(1) = 6811

So $1A9B_{16} = 6811_{10}$

Converting a (Denary) Decimal Number into Hexadecimal Number

The easiest way to convert from decimal to hexadecimal is to use the same division algorithm that you used to convert from decimal to binary, but repeatedly dividing by 16 instead of by 2. As before, we keep track of the remainders, and the sequence of remainders forms the hexadecimal representation.

For example, to convert the decimal number **746** to hexadecimal, we proceed as follows:

		Remainder	
16 | 746			
|46	**-**	**10 =**	**A**
|2		**14 =**	**E**
|0			**2**

We read the number as last is first and first is last.

2EA

So, the decimal number 746 = 2EA in hexadecimal

Computers(9618) with Majid Tahir

BCD Binary Coded Decimals:

In computing and electronic systems, **binary-coded decimal** (**BCD**) is a class of binary encodings of decimal numbers where each decimal digit is represented by a fixed number of bits, usually four or eight. Special bit patterns are sometimes used for a sign or for other indications (e.g., error or overflow).

0 0 0 0 = 0	0 1 0 1 = 5
0 0 0 1 = 1	0 1 1 0 = 6
0 0 1 0 = 2	0 1 1 1 = 7
0 0 1 1 = 3	1 0 0 0 = 8
0 1 0 0 = 4	1 0 0 1 = 9

BCD was used in many early decimal computers, and is implemented in the instruction set of machines such as the IBM System/360 series and its descendants

and Digital's VAX. Although BCD *per se* is not as widely used as in the past and is no longer implemented in computers' instruction sets[*dubious* – *discuss*], decimal fixed-point and floating-point formats are still important and continue to be used in financial, commercial, and industrial computing

As most computers deal with data in 8-bit bytes, it is possible to use one of the following methods to encode a BCD number:

Unpacked: each numeral is encoded into one byte, with four bits representing the numeral and the remaining bits having no significance.

Packed: two numerals are encoded into a single byte, with one numeral in the least significant nibble (bits 0 through 3) and the other numeral in the most significant nibble

The Denary number **8 5 0 3** could be represented by **one BCD digit per byte**

00001000 00000101 00000000 000000011(Unpacked)Denary

Number **8 5 0 3** represented by **One BCD per nibble**

1000 0101 0000 0011 (Packed)

e.g. **398602** in **BCD**

Answer: 3 = 0011 9 = 1001 8 = 1000 6 = 0110 0 = 0000 2 = 0010 So
398602 = 001110011000011000000010 (in BCD)

Method 1: four single bytes

0	0	0	0	0	0	1	1	3
0	0	0	0	0	0	0	1	1
0	0	0	0	0	1	1	0	6
0	0	0	0	0	1	0	1	5

Method 2: two bytes

0	0	1	1	0	0	0	1	3	1
0	1	1	0	0	1	0	1	6	5

Note: All the zeros are essential otherwise you can't read it back.

But do not get confused, ***binary coded decimal*** is not the same as hexadecimal. Whereas a 4-bit hexadecimal number is valid up to F_{16} representing binary 1111_2, (decimal 15), binary coded decimal numbers stop at 9 binary 1001_2

Uses of BCD:

There are a number of applications where BCD can be used.

- The obvious type of application is where denary digits are to be displayed, for instance on the screen of a calculator or in a digital time display.

- A somewhat unexpected application is for the representation of currency values. When a currency value is written in a format such as $300.25 it is as a fixed-point decimal number (ignoring the dollar sign). It might be expected that such values would be stored as real numbers but this cannot be done accurately.

ASCII code:

Computers(9618) with Majid Tahir

If text is to be stored in a computer it is necessary to have a coding scheme that provides a unique binary code for each distinct individual component item of the text.

Such a code is referred to as a **character code**.

- The scheme which has been used for the longest time is the ASCII (American Standard Code for Information Interchange) coding scheme.
- This is an internationally agreed standard. There are some variations on ASCII coding schemes but the major one is the 7-bit code. It is customary to present the codes in a table for which a number of different designs have been used.
- The full table shows the 27 (128) different codes available for a 7-bit code. You should not try to remember any of the individual codes but there are certain aspects of the coding scheme which you need to understand.

- Computers store text documents, both on disk and in memory, using ASCII codes. For example, if you use Notepad in Windows OS to create a text file containing the words, "Four score and seven years ago," Notepad would use 1 byte of memory per character (including 1 byte for each space character between the words

- It is worth emphasizing here that these codes for numbers are exclusively for use in the context of stored, displayed or printed text.

- All of the other coding schemes for numbers are for internal use in a computer system and would not be used in a text.

- There are some special features that make the coding scheme easy to use in certain circumstances.

    - The first is that the codes for numbers and for letters are in sequence in each case so that, for example,
    - if 1 is added to the code for seven the code for eight is produced.
    - The second is that the codes for the upper-case letters differ from the codes for the corresponding lower-case letters only in the value of bit 6.
    - This makes conversion of upper case to lower case, or the reverse, a simple operation.

Computers(9618) with Majid Tahir

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	<SPACE>	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	“	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	‘	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o

Notice the storage of characters with uppercase and lowercase. For example:

a 1 1 0 0 0 0 1 hex 61 (lower case)
A 1 0 0 0 0 0 1 hex 41 (upper case)
y 1 1 1 1 0 0 1 hex 79 (lower case)
Y 1 0 1 1 0 0 1 hex 59 (uppercase)

Unicode

Despite still being widely used, the ASCII codes are far from adequate for many purposes.

Unicode is an international encoding standard for use with different languages and scripts.

It works by providing a unique number for every character, this creates a consistent encoding, representation, and handling of text.

Basically Unicode is like a Universal Alphabet that covers the majority of different languages across the world, it transforms characters into numbers.

It achieves this by using character encoding, which is to assign a number to every character that can be used.

What’s an example of a Unicode?

Unicode has its own special terminology. For example, a character code is referred to as a **'code point'**.

In any documentation there is a special way of identifying a code point. An example is U+0041 which is the code point corresponding to the alphabetic character A.

The 0041 are hexadecimal characters representing two bytes. The interesting point is that in a text where the coding has been identified as Unicode it is only necessary to use a one-byte representation for the 128 codes corresponding to ASCII. To ensure such a code cannot be misinterpreted, the codes where more than one byte is needed have restrictions applied.

ا FE8D	ـا FE8E	آ FE81	ـآ FE82
ب FE8F	بـ FE91	ـبـ FE92	ـب FE90
پ FB56	پـ Fb58	ـپـ FB59	ـپ FB57

0000	00D0	00F0	0141	0142	0160	0161	00DD	00FD	0009	000A	00DE	00FE	000D	017D	017E
	Đ	ð	Ł	ł	Š	š	Ý	ý			Þ	þ		Ž	ž
0010	0011	0012	0013	0014	00BD	00BC	00B9	00BE	00B3	00B2	00A6	2212	00D7	001E	001F
					½	¼	¹	¾	³	²	¦	−	×		
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
00C4	00C5	00C7	00C9	00D1	00D6	00DC	00E1	00E0	00E2	00E4	00E3	00E5	00E7	00E9	00E8
Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è
00EA	00EB	00ED	00EC	00EE	00EF	00F1	00F3	00F2	00F4	00F6	00F5	00FA	00F9	00FB	00FC
ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
2020	00B0	00A2	00A3	00A7	2022	00B6	00DF	00AE	00A9	2122	00B4	00A8	2260	00C6	00D8
†	°	¢	£	§	•	¶	ß	®	©	™	´	¨	≠	Æ	Ø
221E	00B1	2264	2265	00A5	00B5	2202	2211	220F	03C0	222B	00AA	00BA	03A9	00E6	00F8
∞	±	≤	≥	¥	µ	∂	∑	∏	π	∫	ª	º	Ω	æ	ø
00BF	00A1	00AC	221A	0192	2248	2206	00AB	00BB	2026	00A0	00C0	00C3	00D5	0152	0153
¿	¡	¬	√	ƒ	≈	∆	«	»	…		À	Ã	Õ	Œ	œ

At its core, Unicode is like ASCII: a list of characters that people want to type into a computer. Every character gets a numeric codepoint, whether it's capital A, lowercase or lambda.

A = 65

λ = 923

So Unicode says things like, ―Allright, this character exists, we assigned it an official name and a codepoint, here are its lowercase or uppercase equivalents (if any), and here's a picture of what it could look like. Font designers, it's up to you to draw this in your font if you want to.

Just like ASCII, Unicode strings (imagine ―codepoint 121, codepoint 111…‖) have to be encoded to ones and zeros before you can store or transmit them. But unlike ASCII, Unicode has more than a million possible codepoints, so they can't possibly all fit in one byte. And unlike ASCII, there's no One True Way to encode it.

What can we do? One idea would be to **always** use, say, 3 bytes per character. That would be nice for string traversal, because the 3rd codepoint in a string would always start at the 9th byte. But it would be inefficient when it comes to storage space and bandwidth.Instead, the most common solution is an encoding called UTF-8.

UTF-8 :

UTF-8 gives you four templates to choose from: a one-byte template, a two-byte template, a three-byte template, and a four-byte template.

0xxxxxxx

110xxxxx 10xxxxxx

1110xxxx 10xxxxxx 10xxxxxx

11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Each of those templates has some headers which are always the same (**shown here in red**) and some slots where your code point data can go (shown here as black).

The four-byte template gives us 21 bits for our data, which would let us represent 2,097,151 different values. There are only about 128,000 codepoints right now, so UTF-8 can easily encode any Unicode codepoint for the foreseeable future.Unicode to represent any possible text in code form.

Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.

Developed in conjunction with the Universal Coded Character Set (UCS) standard and published as *The Unicode Standard*, the latest version of Unicode contains a repertoire of more than 128,000 characters covering 135 modern and historic scripts, as well as multiple symbol sets..

As of June 2016, the most recent version is *Unicode 9.0*. The standard is maintained by the Unicode Consortium.

Unicode's success at unifying character sets has led to its widespread and predominant use in the internationalization and localization of computer software. The standard has been implemented in many recent technologies, including modern operating systems, XML, Java (and other programming languages), and the .NET Framework

References:

http://www.bfoit.org/itp/ComputerContinuum/RobotComputer.html

https://en.wikibooks.org/wiki/GCSE_Computer_Science/Binary_representation
http://bssbmi.com/olevel/computer-science-2210/class-9/binary-systems/
http://www.math10.com/en/algebra/systems-of-counting/binary-system.html
Reference: http://www.bbc.co.uk/education/guides/zjfgjxs/revision/6

Computers(9618) with Majid Tahir