Syllabus Content:

1.1.1 Number representation

- show understanding of the basis of different number systems and use the binary, denary and hexadecimal number system
- convert a number from one number system to another
- express a positive or negative integer in two's complement form
- show understanding of, and be able to represent, character data in its internal binary form depending on the character set used (Candidates will not be expected to memorise any particular character codes but must be familiar with ASCII and Unicode.)
- express a denary number in Binary Coded Decimal (BCD) and vice versa
- describe practical applications where BCD is used

Binary, Denary & Hexadecimal

The **binary** system on computers uses combinations of 0s and 1s.

In everyday life, we use numbers based on combinations of the digits between 0 and 9.

This counting system is known as **decimal**, **denary** or **base 10**.

$$(0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)_{10}$$

A number **base** indicates how many digits are available within a numerical system. Denary is known as **base 10** because there are ten choices of digits between 0 and 9.

For binary numbers there are only two possible digits available:

$$(0 \text{ or } 1)_2$$

The binary system is also known as **base 2**.

All denary numbers have a binary equivalent and it is possible to **convert** between denary and binary.

Place values

Denary place values

Using the **denary** system, **6432** reads as six thousand, four hundred and thirty two. One way to break it down is as:

- **six** thousands
- **four** hundreds
- **three** tens
- **two** ones

Each number has a **place value** which could be put into columns. Each column is a power of ten in the base 10 system:

Thousands 1000s (10^3)	Hundreds 100s (10^2)	Tens 10s (10^1)	Ones 1s (10^0)
6	4	3	2

Or think of it as:
(**6** x 1000) + (**4** x 100) + (**3** x 10) + (**2** x 1) = **6432**

Binary place values

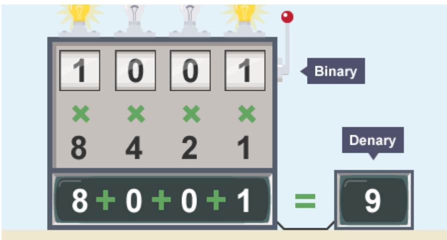
You can also break a **binary** number down into place-value columns, but each column is a power of two instead of a power of ten.
For example, take a binary number like **1001**. The columns are arranged in multiples of 2 with the binary number written below:

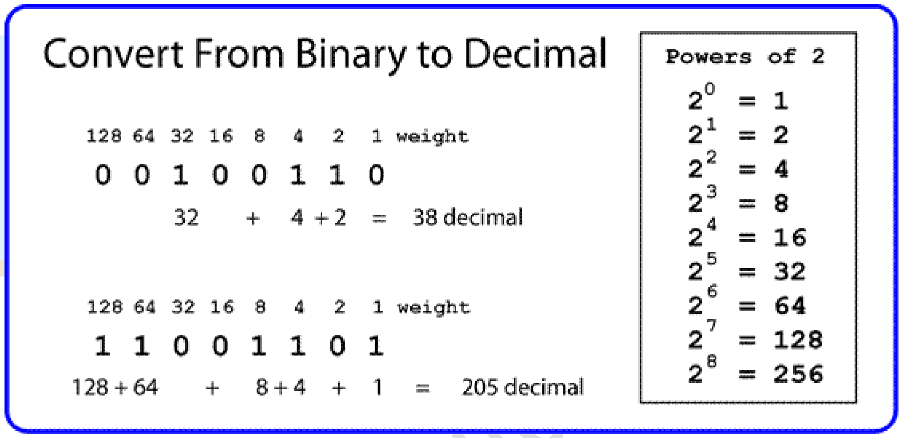
Eights 8s (2^3)	Fours 4s (2^2)	Twos 2s (2^1)	Ones 1s (2^0)
1	0	0	1

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** x 8) + (**0** x 4) + (**0** x 2) + (**1** x 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** x 128) + (**0** x 64) + (**1** x 32) + (**0** x 16) + (**1** x 8) + (**0** x 4) + (**0** x 2) + (**0** x 1) =
128 + 32 + 8 = **168**

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.

Play

A method of converting a denary number to binary

Worked example: Denary number 83

1. **83** ÷ 2 = 41 remainder **1**
2. **41** ÷ 2 = 20 remainder **1**
3. **20** ÷ 2 = 10 remainder **0**
4. **10** ÷ 2 = 5 remainder **0**
5. **5** ÷ 2 = 2 remainder **1**
6. **2** ÷ 2 = 1 remainder **0**
7. **1** ÷ 2 = 0 remainder **1**

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

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

To check that this is right, convert the binary back to denary:
(**1** x 64) + (**0** x 32) + (**1** x 16) + (**0** x 8) + (**0** x 4) + (**1** x 2) + (**1** x 1) = **83**

Worked example: Denary number 122

1. **122** ÷ 2 = 61 remainder **0**
2. **61** ÷ 2 = 30 remainder **1**
3. **30** ÷ 2 = 15 remainder **0**
4. **15** ÷ 2 = 7 remainder **1**
5. **7** ÷ 2 = 3 remainder **1**
6. **3** ÷ 2 = 1 remainder **1**
7. **1** ÷ 2 = 0 remainder **1**

Put the remainders in **reverse** order to get the final number: **1111010**.

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

To check that this is right, convert the binary back to denary:

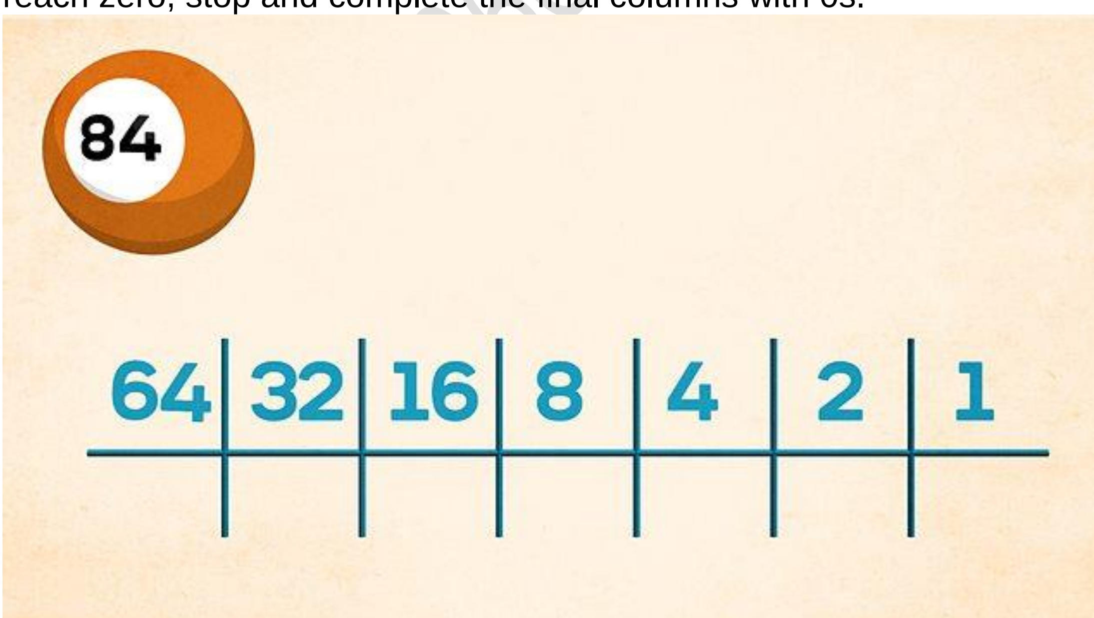
(**1** x 64) + (**1** x 32) + (**1** x 16) + (**1** x 8) + (**0** x 4) + (**1** x 2) + (**0** x 1) = **122**

The binary representation of an even number always ends in 0 and an odd number in 1.

Converting denary to binary: Method 2

There are two methods for converting a **denary** (base 10) number to **binary** (base 2). This is method two.
Take off the biggest 2^n value you can

Remove the 2^n numbers from the main number and mark up the equivalent 2^n column with a 1. Work through the remainders until you reach zero. When you reach zero, stop and complete the final columns with 0s.



Play

A method of converting a denary number to binary

Download Transcript

Worked example: Denary number 84

First set up the columns of base 2 numbers. Then look for the highest 2^nnumber that goes into 84.

1. Set up the columns of base 2 numbers
2. Find the highest 2^n number that goes into 84. The highest 2^n number is 26 = **64**
3. 84 – 64 = 20. Find the highest 2^n number that goes into 20. The highest 2^n number is 24 = **16**
4. 20 - 16 = 4. Find the highest 2^n number that goes into 4. The highest 2^n number is 22 = **4**
5. 4 - 4 = 0
6. Mark up the columns of base 2 numbers with a 1 where the number has been the highest 2^nnumber, or with a 0:

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

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

To check that this is right, convert the binary back to denary:
(**1** x 64) + (**0** x 32) + (**1** x 16) + (**0** x 8) + (**1** x 4) + (**0** x 2) + (**0** x 1) = **84**

Binary combinations

These tables show how many binary combinations are available for each bit size.

One bit

		Binary number
		Place value 1
Denary number	0	0
	1	1

Maximum binary number = 1

Maximum denary number = 1

Binary combinations = 2

Two bit

		Binary pattern	
		Place value 2	Place value 1
Denary number	0	0	0
	1	0	1
	2	1	0
	3	1	1

Maximum binary number = 11

Maximum denary number = 3

Binary combinations = 4

Three bit

Maximum binary number = 111

Maximum denary number = 7

Binary combinations = 8

		Binary pattern		
		Place value 4	Place value 2	Place value 1
Denary number	0	0	0	0
	1	0	0	1
	2	0	1	0
	3	0	1	1
	4	1	0	0
	5	1	0	1
	6	1	1	0
	7	1	1	1

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

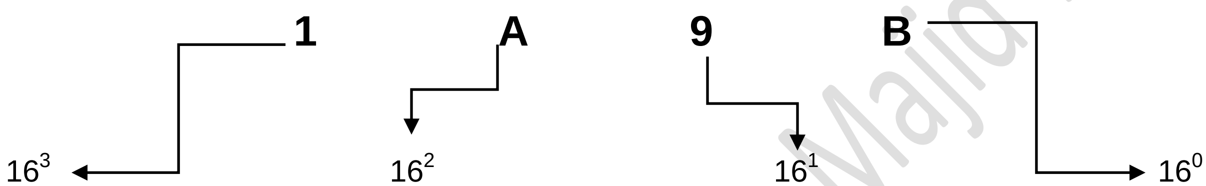
Fortunately, large binary numbers can be made much more compact—and hence easier to work with—if represented in base-16, the so-called hexadecimal number system. You may wonder: Binary numbers would also be more compact if represented in base-10—why not just convert them to decimal? The answer, as you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

symbol B is equivalent to the decimal number 11, and so forth, with the symbol F being equivalent to the decimal number 15.

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:



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). These 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) $2\,8_{16} = 2\,8_H = 2 \times 16^1 + 8 \times 16^0 = 40$

$= 32 \quad + \quad 8 \quad = \quad 40$

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

(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, $(\mathbf{A\ D\ 4\ 5})_{16}$ 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

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.

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

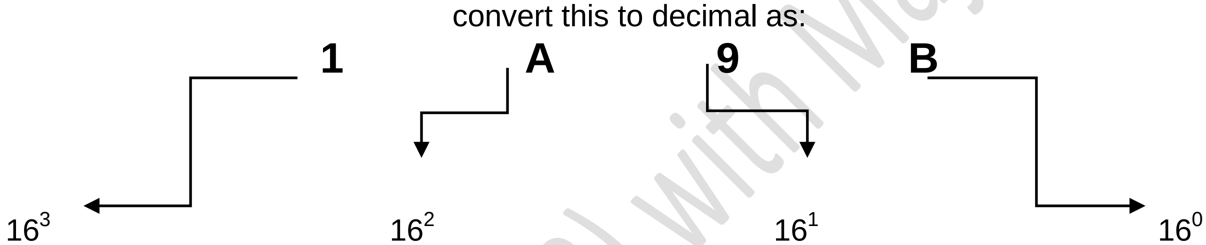
So Binary equivalent of Hexadecimal number is: 9F2= 100111110010

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:



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

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.

Step 1: Convert 65d to binary. Ignore the sign for now. Use the absolute value. The absolute value of -65 is 65.

65 --> 01000001 binary

Step 2: Convert 01000001 to its one's complement.

01000001 --> 10111110

Step 3: Convert 10111110b to its two's complement by adding 1 to the one's complement.

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

10111111b 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
---------------------
  100000000b      0  denary
  ^
```

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

We will examine signed binary values in more detail later. For now, understand the difference between one's complement and two's complement and practice converting between them.

One's Complement

If all bits in a byte are inverted by changing each **1 to 0** and each **0 to 1**, we have formed the one's complement of the number.

```
Original      One's Complement
-------------------------------
10011001  -->  01100110
10000001  -->  01111110
11110000  -->  00001111
11111111  -->  00000000
00000000  -->  11111111
```

Converting to one's complement.

And that is all there is to it! One's complement is useful for forming the two's complement of a number.

Two's Complement (Binary Additive Inverse)

The two's complement is a method for representing positive and negative integer values in binary. The useful part of two's complement is that it automatically includes the sign bit. Rule: To form the two's complement, add 1 to the one's complement.

Step 1: **Begin with the original binary value**

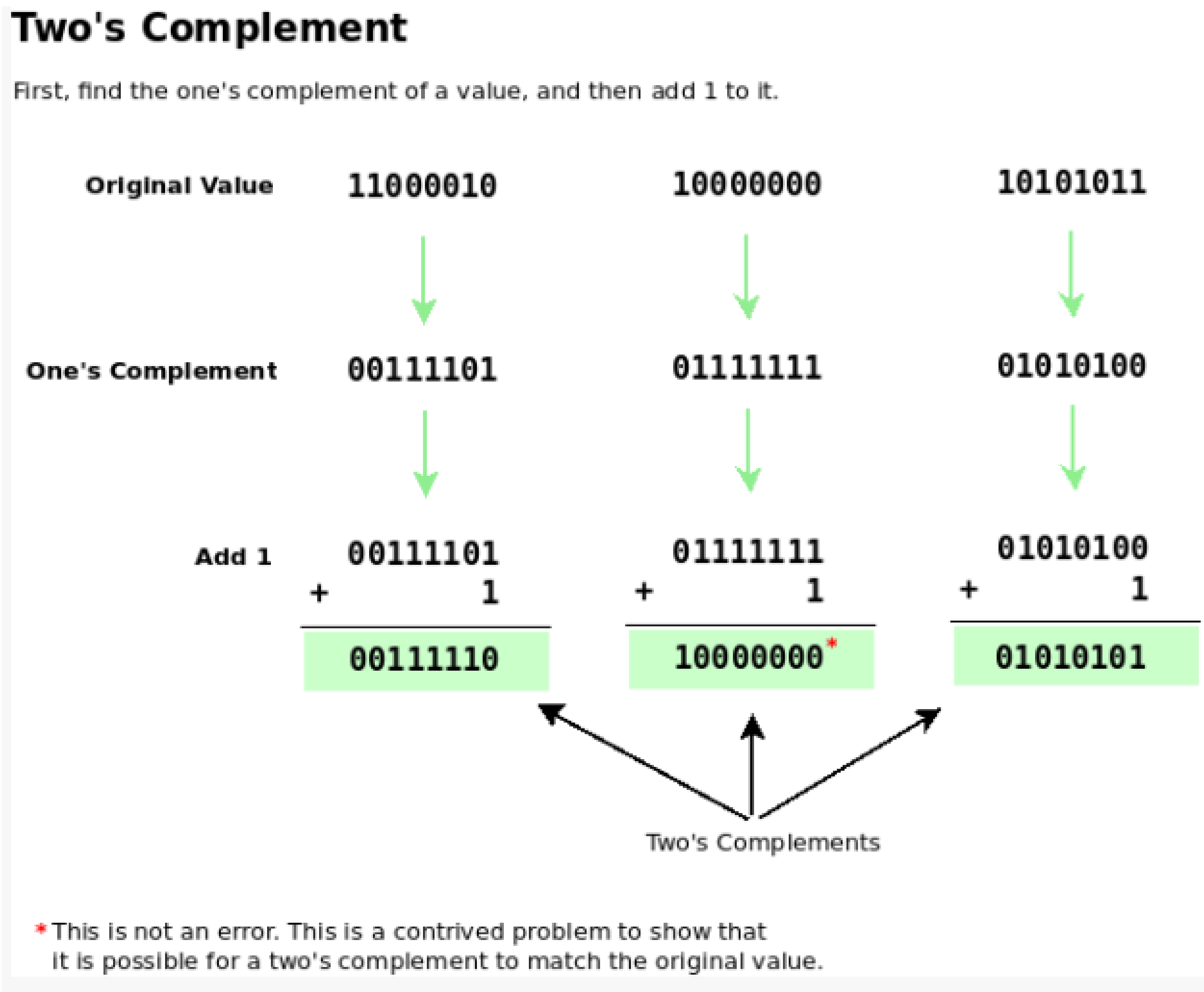
10011001 Original byte

Step 2: **Find the one's complement**

01100110 One's complement

Step 3: **Add 1 to the one's complement**

```
01100110         One's complement
  +        1     Add 1
--------------
01100111  <---  Two's complement
```



USE OF HEXADECIMAL NUMBER IN COMPUTER REGISTERS AND MAIN MEMORY:

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

Computers are comprised of chips, registers, transistors, resistors, processors, traces, and all kinds of things. To get the binary bits from one place to the next, software programmers convert binary to hex and move hex values around. In reality, the computer is still shoving 1's and 0's along the traces to the chips.

There are two important aspects to the beauty of using Hexadecimal with computers: First, it can represent 16-bit words in only four Hex digits, or 8-bit bytes in just two; thus, by using a numeration with more symbols, it is both easier to work with (saving paper and screen space) and makes it possible to understand some of the vast streams of data inside a computer merely by looking at the Hex output. This is why programs such as DEBUG, use only Hexadecimal to display the actual Binary bytes of a Memory Dump rather than a huge number of ones and zeros!

The second aspect is closely related: Whenever it is necessary to convert the Hex representation back into the actual Binary bits, the process is simple enough to be done in your own mind.

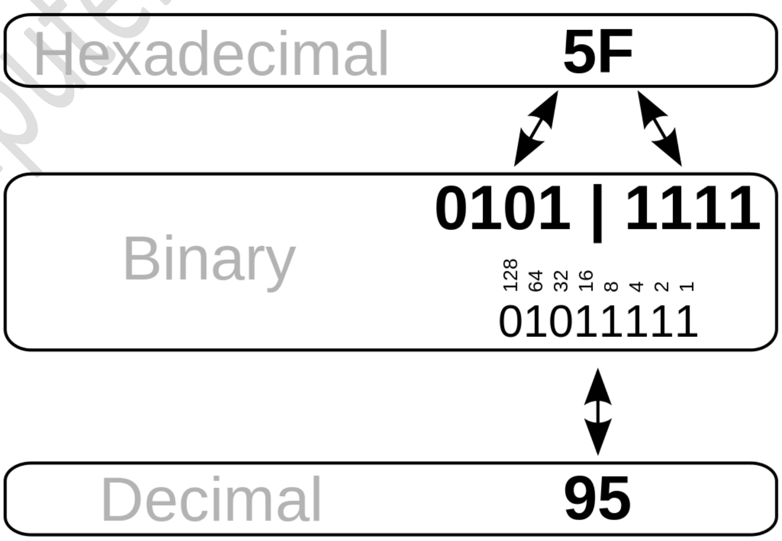
For example, **FAD7 hex is 1111 1010 1101 0111 (F=1111, A=1010, D=1101, 7=0111)** in Binary. The reason one might wish to do this is in order to work with "logical" (AND, OR or XOR) or "bit-oriented" instructions (Bit tests, etc.) which may make it easier (at times) for a programmer to comprehend.

For example, if you wanted to logically AND the hex number FAD7 with D37E, you might have a difficult time without first changing these numbers into Binary. If you jot them out in Binary on scratch paper, the task will be much easier:

FAD7(hex)	1 1 1 1	1 0 1 0	1 1 0 1	0 1 1 1
D37E(hex)	1 1 0 1	0 0 1 1	0 1 1 1	1 1 1 0
ANDing gives	1 1 0 1	0 0 1 0	0 1 0 1	0 1 1 0
Answer (in hex)	D	2	5	6

Converting Between Bases

To convert from denary to hexadecimal, it is recommended to just convert the number to binary first, and then use the simple method above to convert from binary to hexadecimal.



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).

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 as **1000010100000011**)

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

Answer: 3 = 0011 9 = 1001 8 = 1000 6 = 0110 0 = 0000 2 = 0010

So 398602 = 001110011000011000000010 (in BCD)

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

Application

The BIOS in many personal computers stores the date and time in BCD because the MC6818 real-time clock chip used in the original IBM PC AT motherboard provided the time encoded in BCD. This form is easily converted into ASCII for display.

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.

IBM and BCD IBM used the terms binary-coded decimal and BCD for six-bit alphameric codes that represented numbers, upper-case letters and special characters. Some variation of BCD was used in most early IBM computers, including the IBM 1620, IBM 1400 series, and non-Decimal Architecture members of the IBM 700/7000 series. With the introduction of System/360, IBM replaced BCD with 8-bit EBCDIC

ASCII code:

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 ind ividua l codes but the re are certain aspects of the coding scheme which you need to understand.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

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 emphasising 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 5. This makes conversion of upper case to lower case, or the reverse, a simple operation.

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.

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 lambda, or "man in business suit levitating."

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 codepoint data can go (shown here as "x"s).

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

Practice Questions

Convert to one's complement:

1. 1010
2. 11110000
3. 10111100 11000000
4. 10100001

Convert to two's complement:

1. 1010
2. 11110000
3. 10000000
4. 011111111

Convert these negative decimal values to negative binary using two's complement:

1. -192d
2. -16d
3. -1d
4. -0d

Answers

One's complement:

1. 0101
2. 00001111
3. 01000011 00111111
4. 01011110

Two's complement:

1. 0110 (1010 –> 0101 + 1 = 0110)
2. 00010000
3. 10000000 (Result is the same as the original value.)
4. 10000001

Negative decimal to negative binary:

1. 01000000b (192d = 11000000b –> 00111111 + 1 = 01000000b)
2. 11110000b (16d = 00010000b –> 11101111 + 1 = 11110000b)
3. 11111111b (1d = 00000001b –> 11111110 + 1 = 11111111b) Tricky? Before converting from binary to decimal, we must know ahead of time if the binary value is signed or not because a signed binary value will not convert properly using the place value chart we have seen so far. If seen by itself, 11111111b = 255d, not -1d. As a rule, assume that a binary value, such as 11111111b, is a positive integer unless context specifies otherwise. Since we are dealing with negative binary values in this problem set, then 11111111b is -1d, not 255d.
4. 0. There is no such thing as negative zero (-0). Nothing is always nothing and does not have a sign. We can convert anyway: 0d = 00000000 –> 11111111 + 1 = 1 00000000b. We still arrive at 0 in binary for the eight relevant bits. Ignore the ninth carry bit.

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
http://www.answers.com/Q/What_are_the_applications_of_BCD_Code
https://en.wikipedia.org/wiki/Binary-coded_decimal
https://commons.wikimedia.org/wiki/File:CPT-Numbers-Conversion.svg