

CHAPTER 1.2 : DATA REPRESENTATION

Data Representation Assembly language programmers deal with data at the physical level, so they must be adept at examining memory and registers.

Often, binary numbers are used to describe the contents of computer memory; at other times, decimal and hexadecimal numbers are used.

You must develop a certain fluency with number formats, so you can quickly translate numbers from one format to another.

Each numbering format, or system, has a base, or maximum number of symbols that can be assigned to a single digit.

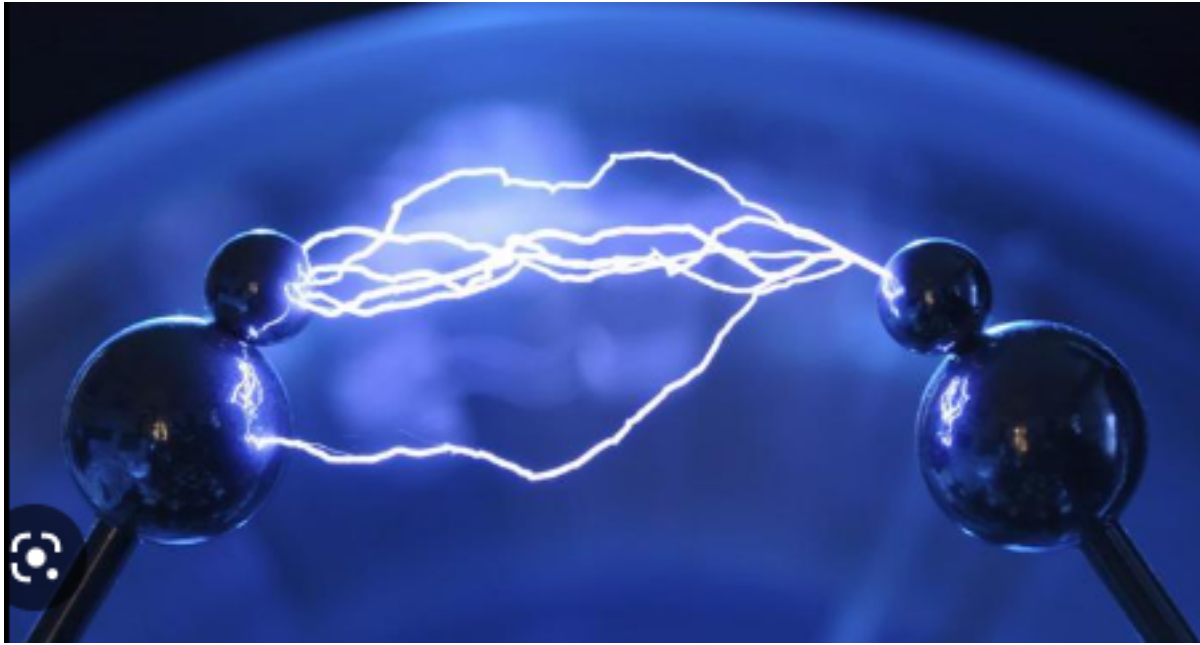
Table 1-2 shows the possible digits for the numbering systems used most commonly in hardware and software manuals. In the last row of the table, hexadecimal numbers use the digits 0 through 9 and continue with the letters A through F to represent decimal values 10 through 15.

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

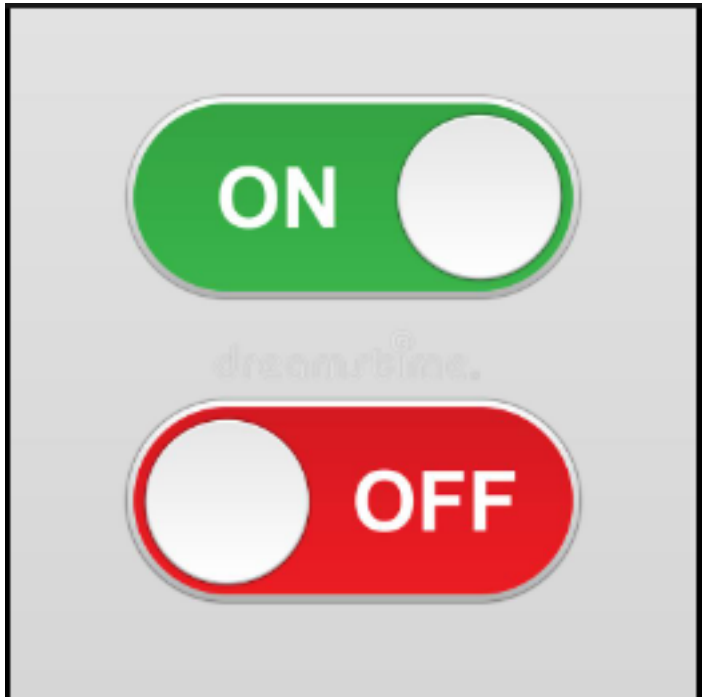
It is quite common to use hexadecimal numbers when showing the contents of computer memory and machine-level instructions.

BINARY INTEGERS

A computer stores instructions and data in memory as collections of electronic charges.

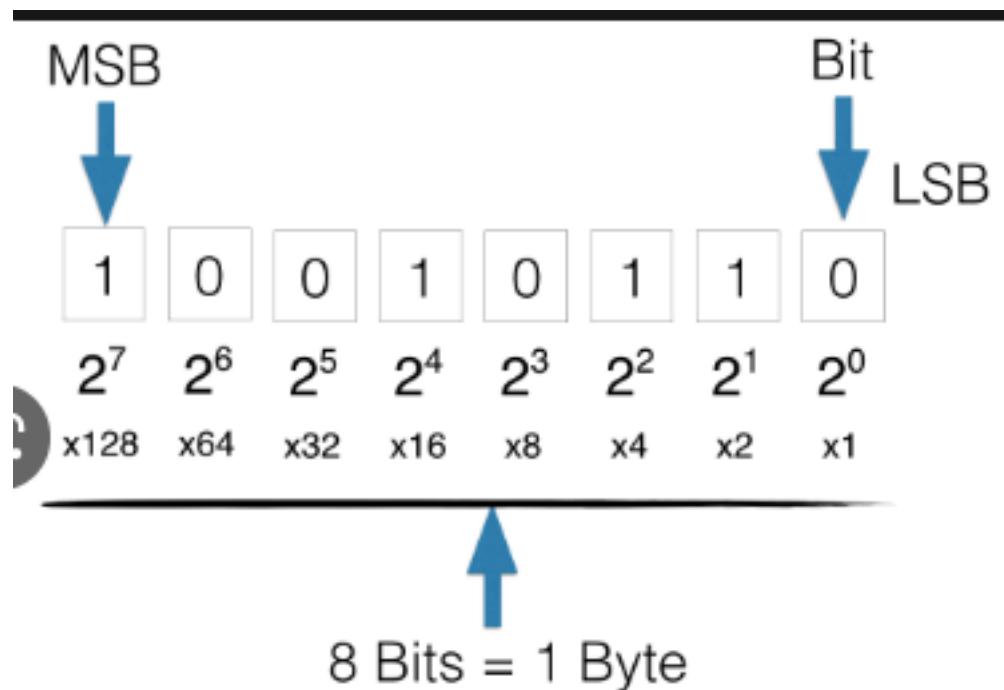


Representing these entities with numbers requires a system geared to the concepts of on and off or true and false.



Binary numbers are base 2 numbers, in which each binary digit (called a bit) is either 0 or 1.

S



Bits are numbered sequentially starting at zero on the right side and increasing toward the left. The bit on the left is called the most significant bit (MSB), and the bit on the right is the least significant bit (LSB). The MSB and LSB bit numbers of a 16-bit binary number are shown in the following figure:

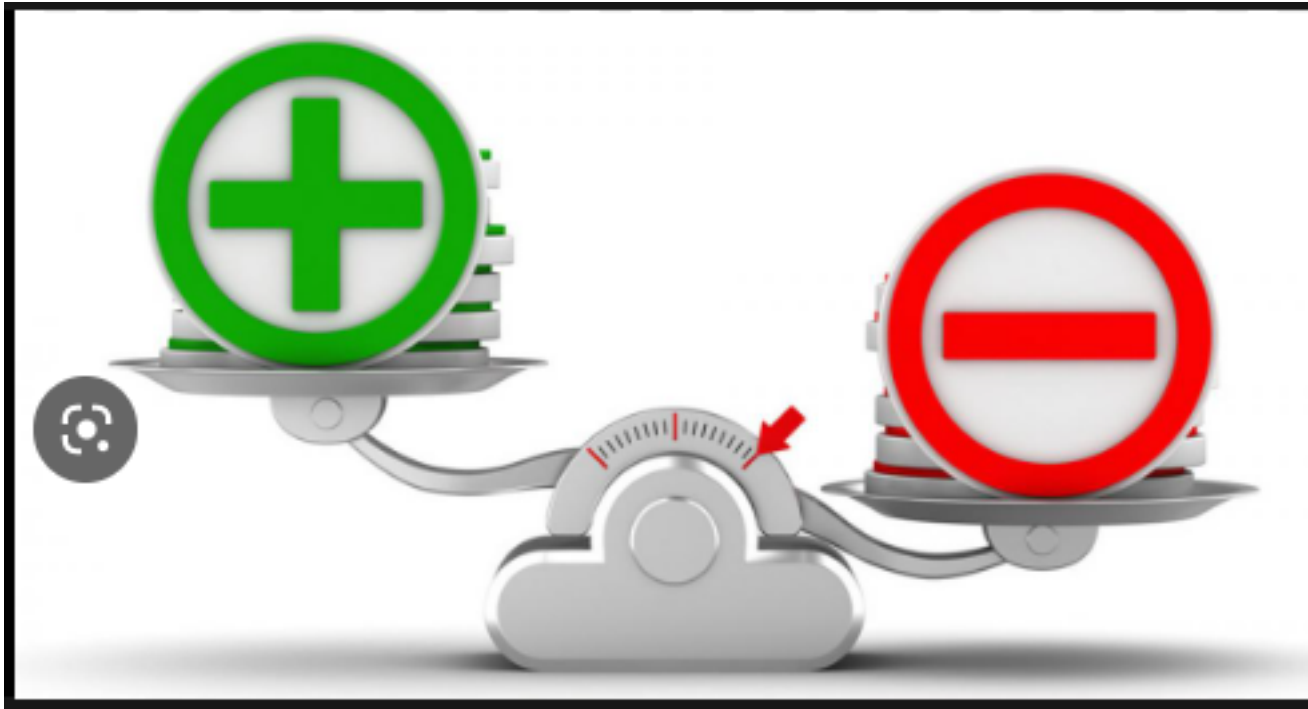


Table 1-2 Binary, Octal, Decimal, and Hexadecimal Digits.

System	Base	Possible Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Binary integers can be signed or unsigned.

A **signed integer** is positive or negative.



An **unsigned integer** is by default positive. Zero is considered positive.



When writing down large binary numbers, many people like to insert a dot every 4 bits or 8 bits to make the numbers easier to read. Examples are 1101.1110.0011.1000.0000 and 11001010.10101100.

Unsigned Binary Integers Starting with the LSB, each bit in an unsigned binary integer represents an increasing power of 2.

The following figure contains an 8-bit binary number, showing how powers of two increase from right to left:

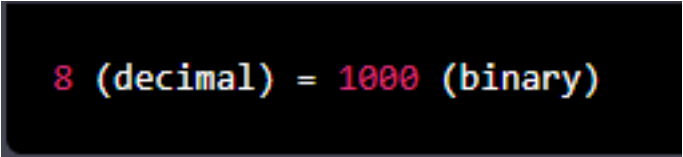
QUICK QUESTION

We use 3 bits to represent a number like 8. (2^3)?

Using 3 bits, we can represent $2^3=8$ different values.

The range of values we can represent with 3 bits is from 000 (0 in decimal) to 111 (7 in decimal).

Here's how we can represent the number 8 in binary using 3 bits:



8 (decimal) = 1000 (binary)

In binary, each digit can have only two possible values: 0 or 1.

Each digit in a binary number represents a power of 2, starting from the rightmost digit with 2^0 , then 2^1 , 2^2 , and so on, doubling each time. In the example above, the leftmost digit represents 2^3 , or 8, which is why the value is 1 and the other digits are 0.

so, the answer is no. We use 4 bits?

Yes, that's correct. To represent the decimal number 8 in binary, we actually need 4 bits, not 3.

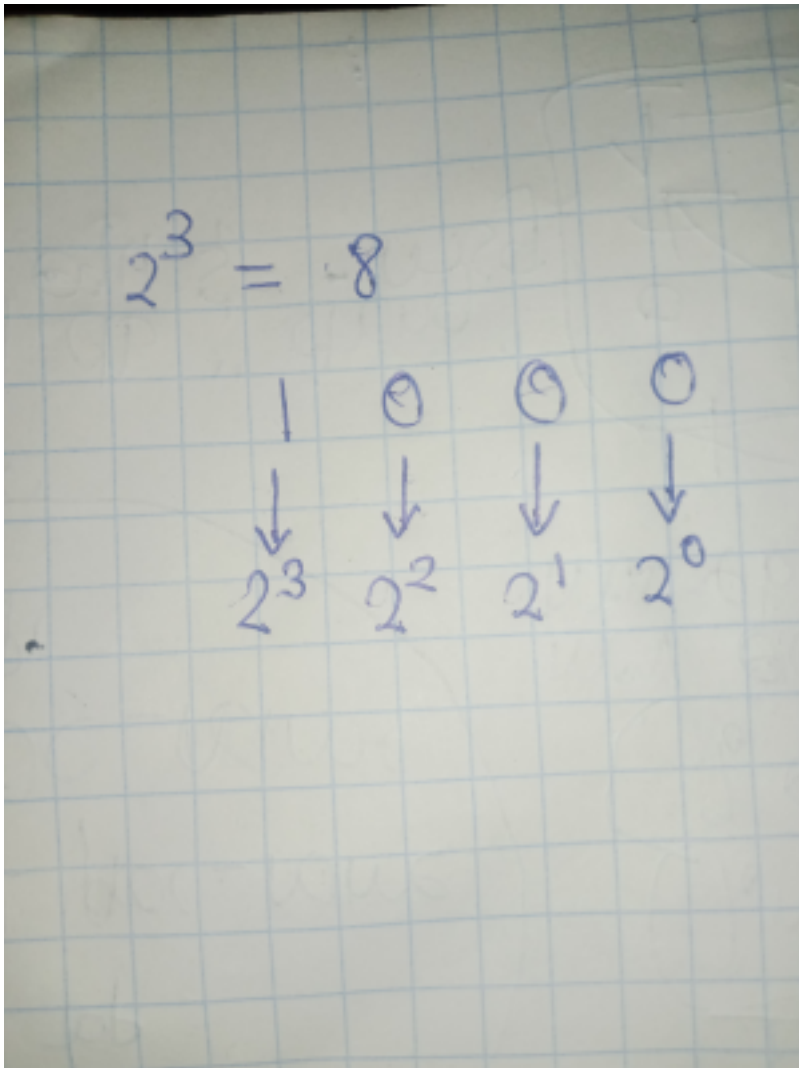
Using 4 bits, we can represent $2^4=16$ different values.

The range of values we can represent with 4 bits is from 0000 (0 in decimal) to 1111 (15 in decimal).

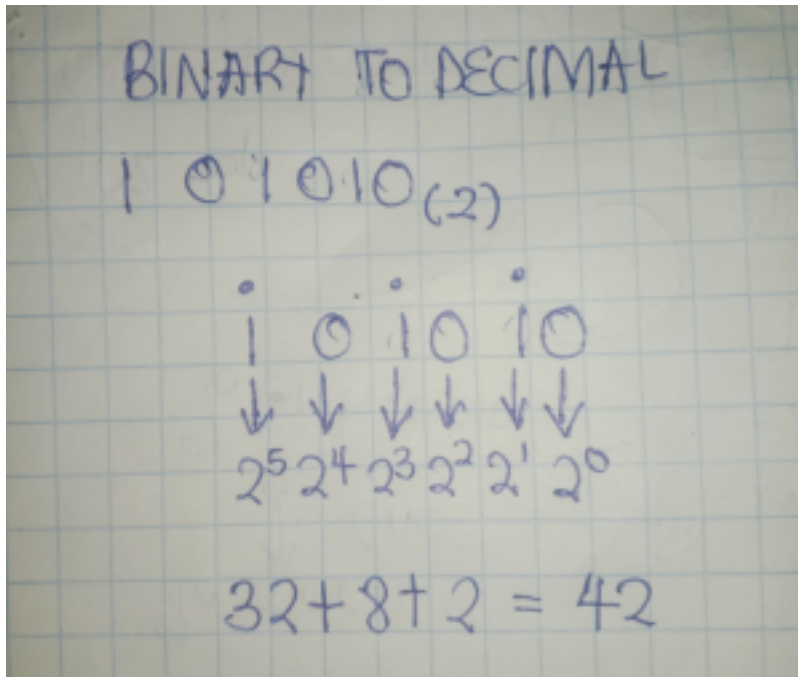
Here's how we can represent the number 8 in binary using 4 bits:

8 (decimal) = 1000 (binary)

In this case, we have one digit representing 2^3 , or 8, and the other three digits are 0. What do I mean?



BINARY TO DECIMAL



BINARY WITH DECIMAL POINTS

$$\begin{array}{cccccc}
 1 & 0 & 1 & 0 & 1 & 0.01 \\
 \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 2^5 & 2^3 & 2^1 & 2^{-1} & 2^{-2} &
 \end{array}$$

$$\begin{aligned}
 32 + 8 + 2 + \frac{1}{4} &= \frac{42}{1} + \frac{1}{4} \\
 &= 42\frac{1}{4} / 42.25
 \end{aligned}$$

DECIMAL TO BINARY

DECIMAL TO BINARY

$$100_{(10)} \rightarrow X_{(2)}$$

$$\begin{array}{rcl} 100 \div 2 & = & 50 \text{ R } 0 \\ 50 \div 2 & = & 25 \text{ R } 0 \\ 25 \div 2 & = & 12 \text{ R } 1 \\ 12 \div 2 & = & 6 \text{ R } 0 \\ 6 \div 2 & = & 3 \text{ R } 0 \\ 3 \div 2 & = & 1 \text{ R } 1 \\ 1 \div 2 & = & 0 \text{ R } 1 \end{array} \quad \uparrow$$

$$= 1100100_{(2)}$$

DECIMAL WITH DECIMAL POINTS TO BINARY

- Convert the fractional part to binary, multiply fractional part with 2 and take the one bit which appears before the decimal.
- Follow the procedure repeatedly until result ends up with 1.0

$$0.25 \times 2 = 0.5 \quad \text{take } 0$$

$$0.5 \times 2 = 1.0 \quad \text{take } 1$$

$$0.25_{(2)} = (01)_2$$

$$100.25_{(10)} = 1100100.01_{(2)}$$

BINARY TO HEX TABLE

BINARY TO HEXADECIMAL

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

HOW TO DRAW THE TABLE AT ONCE

Drawing the table at once:

8 zeros going down

8 ones going down



4 zeros going down
in column 2.

4 ones going down
repeats



2 zeros, 2 ones



Zero, One, Zero, One, zero

BINARY TO OCTAL TABLE

Octal	Binary
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

The table is drawn just like the hex table above.

HEX TO DECIMAL

HEX TO DECIMAL

$7CF_{(16)}$

You can convert to binary,
using the table if you have
no idea of method 2.

7 C F

↓ ↓ ↓
 16^2 16^1 16^0

$$\rightarrow (7 \times 256) + (12 \times 16) + (15 \times 16)$$

$$\rightarrow 1999_{(10)}$$

DECIMAL TO HEX

DECIMAL TO HEX

960(10)

$$960 \div 16 = 60 \text{ R } 0$$

$$60 \div 16 = 3 \text{ R } 12$$

$$3 \div 16 = 0 \text{ R } 3$$



3C0(16)

HEX TO OCTAL

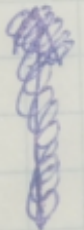
HEX TO OCTAL

1BC₍₁₆₎

Convert to binary
using table.

<u>0001</u>	<u>1011</u>	<u>1100</u>
1	B	C

Subdivide into 3's
from right to left.

000 - 0		right to left
110 - 6		
111 - 7		
100 - 4		

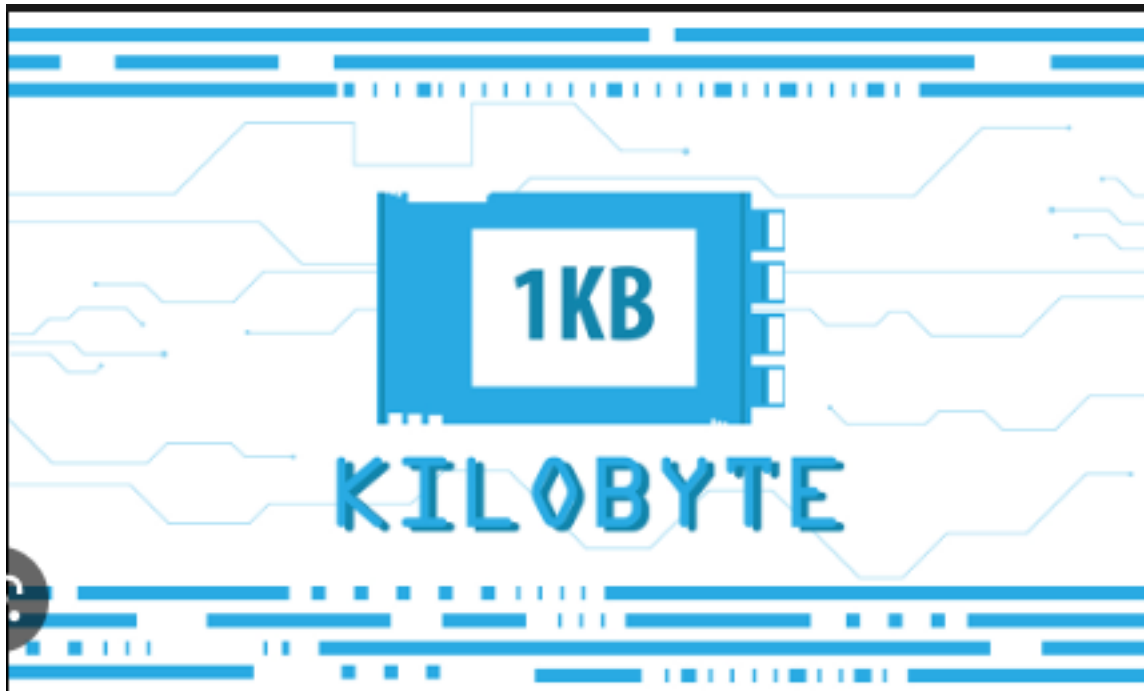
= 674₍₈₎

Zero doesn't alter
the value, so ignore

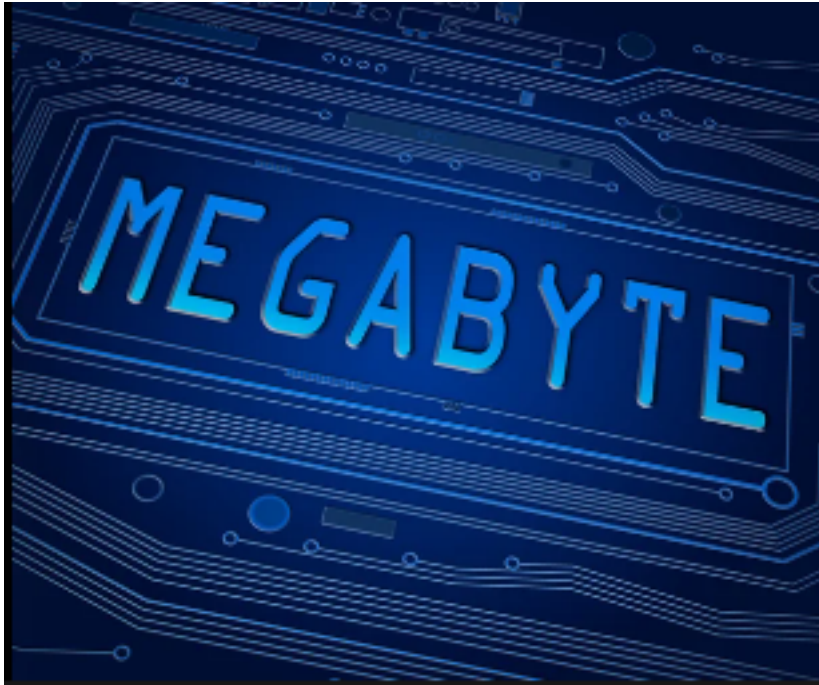
MEASUREMENTS

Yes, there are several large measurements used when referring to both memory and disk space. Here are some of the most common ones:

Kilobyte (KB): 1,000 bytes



Megabyte (MB): 1,000 kilobytes or 1,000,000 bytes



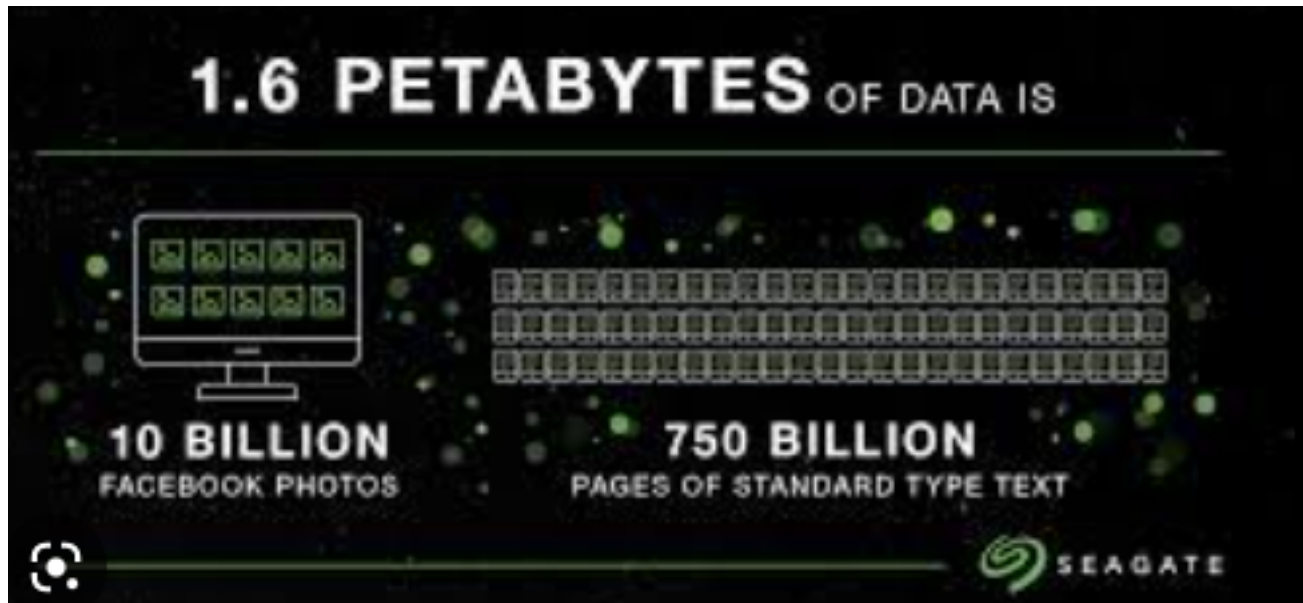
Gigabyte (GB): 1,000 megabytes or 1,000,000,000 bytes



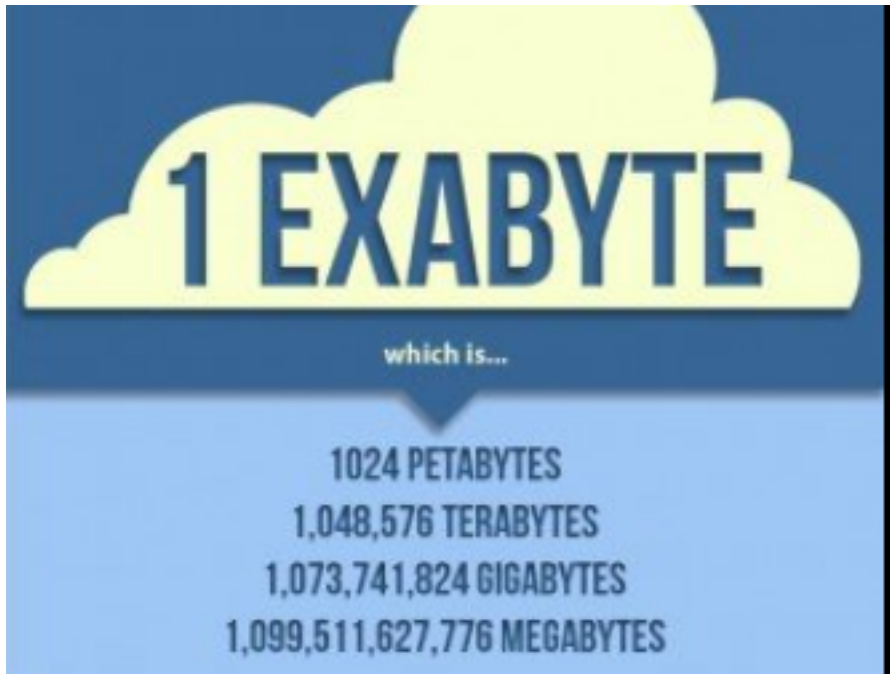
Terabyte (TB): 1,000 gigabytes or 1,000,000,000,000 bytes



Petabyte (PB): 1,000 terabytes or 1,000,000,000,000,000 bytes



Exabyte (EB): 1,000 petabytes or 1,000,000,000,000,000 bytes



Zettabyte (ZB): 1,000 exabytes or 1,000,000,000,000,000,000 bytes



Yottabyte (YB): 1,000 zettabytes or 1,000,000,000,000,000,000,000 bytes

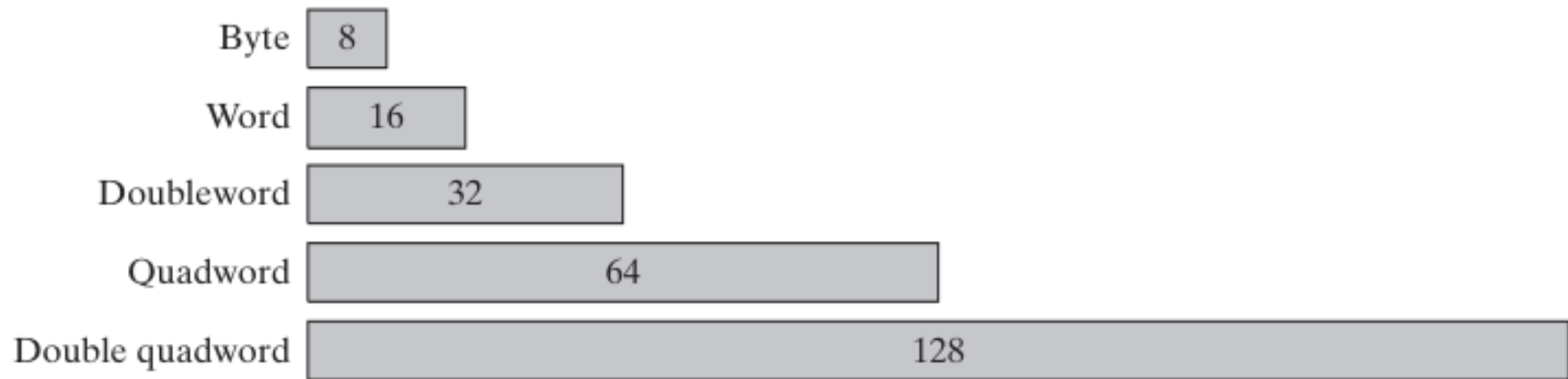


It's important to note that these measurements are based on the decimal system, which means that 1 kilobyte is 1,000 bytes, not 1,024 bytes as it would be in the binary system. Therefore, when discussing computer memory and storage, you may also encounter measurements based on the binary system, such as kibibyte (KiB), mebibyte (MiB), gibibyte (GiB), etc. These measurements are based on powers of 2 and are often used in technical contexts to **avoid confusion.**

INTEGER STORAGE

The basic storage unit for all data in an x86 computer is a **byte**, containing **8 bits**.

Other storage sizes are **word (2 bytes)**, **doubleword (4 bytes)**, and **quadword (8 bytes)**.



Data Type	Size (bits)	Size (bytes)	Minimum Value	Maximum Value
Unsigned Byte	8	1	0	255
Unsigned Word	16	2	0	65,535
Unsigned Doubleword	32	4	0	4,294,967,295
Unsigned Quadword	64	8	0	18,446,744,073,709,551,615
Unsigned Double Quadword	128	16	0	340,282,366,920,938,463,463,374,607,431,768,211,455

Note that the minimum value for all of these data types is 0, since they are unsigned (i.e., they cannot represent negative values).

The maximum value increases as the size of the data type increases.

It's also important to note that the sizes and maximum values listed in this table are for unsigned integers in their raw binary form.

In programming languages, these values may be represented differently, depending on the data type and the language itself.

SIGN AND ZERO EXTENSION

In computing, two's complement format integers have a fixed length, which can create issues when converting values between different lengths. For example, if you need to convert an 8-bit two's complement value to a 16-bit value, or vice versa.

To solve this problem, you can use sign extension or contraction operations. Sign extension replicates the most significant (i.e., leftmost) bit of the original value to fill the additional bits in the longer value.

This ensures that the value retains its original sign (positive or negative) when it is extended. On the other hand, zero extension adds additional 0 bits to the right of the original value to fill the additional bits in the longer value. This is used for unsigned values because there is no sign bit to replicate.

Main point: In the case of the 80x86 architecture, the system works with fixed-length values, even when processing unsigned binary numbers.

This means that:

1. zero extension is used to convert small unsigned values to larger unsigned values, while
2. sign extension is used to convert signed values, replicating the sign bit.

It's worth noting that some operations, such as multiplication and division, require conversion to

32-bit values, which may involve sign or zero extension depending on the original value's sign and whether it is being extended or contracted.

TWO'S COMPLEMENT REPRESENTATION

Two's complement is a binary number representation that allows for the representation of both positive and negative integers using only 0's and 1's.

It is the most commonly used representation for signed integers in modern computing systems.

To represent a positive integer in two's complement, you simply represent it in binary form as usual. For example, to represent the number 5, you would write: 00000101.

To represent a negative integer in two's complement, you first take the binary representation of the absolute value of the number.

Then you invert all the bits (change 0's to 1's and 1's to 0's), and finally add 1 to the result.

For example, to represent the number -5, you would start with the binary representation of 5 (00000101), invert all the bits (11111010), and then add 1, resulting in the two's complement representation: 11111011.

TWO'S COMPLEMENT

- 5 (00000101)
- Getting two's complement
ie -5
- 1's become zeros, 0's
becomes 1's.

1111010

- Add 1:

$$\begin{array}{r} 1111010 \\ + 0000001 \\ \hline 1111011 \end{array}$$

Remember that 1 in the high order bit means negative.

The key advantage of two's complement is that it simplifies the implementation of arithmetic operations, such as addition and subtraction, because the same hardware can be used for both positive and negative numbers.

In addition, it eliminates the need for a separate sign bit, which would require extra operations to check and manipulate.

NOTE:

To find the negative number's binary eg negative (-6), you take its positive(+6), convert to binary(0110), invert the sign bit (1110) and that's it. Remember that 1 means negative.

To find the 1's complement you just invert the bits. To find the 2's complement, you invert, then add the binary of 1 to the result.

To subtract binary numbers, its straightfoward, but you can also convert them to decimal, subtract, then go back to binary.

To subtract 39 and 25, just do it, then convert to binary.

CALCULATING FOR 39, -25

1. Convert the absolute values of both numbers to binary using the 64, 32, 16, 8, 4, 2, 1 method.

39 in binary: 00100111

25 in binary: 00011001

2. Take the two's complement of -25 by inverting all the bits and adding 1:

25 in binary: 00011001

Inverted: 11100110

Add 1: 11100111

So, -25 in two's complement form is 11100111.

3. Make sure both binary numbers have the same number of bits (7 in this case).

00100111 (39)

11100111 (-25)

4. Add the two binary numbers:

00100111

11100111

The result in binary is **00001110**.

5. Convert the binary result to decimal:

00001110 in decimal is 14.

So, $39 - (-25) = 39 + 25 = 64$, and the binary representation of 64 is **01000000**.

TWO'S COMPLEMENT OF A HEXADECIMAL NUMBER

To create the two's complement of a hexadecimal integer in the hexadecimal number system, you would follow these steps:

1. Convert the hexadecimal integer to binary.
2. Reverse all the bits (change 0's to 1's and 1's to 0's).
3. Add 1 to the result.

An easy way to reverse the bits of a hexadecimal digit is to subtract the digit from F (15 in decimal). Example 1:

1. Convert 5A to binary:

0101 1010

2. Reverse all the bits:

1010 0101

3. Add 1 to the result:

1010 0101 + 0000 0001 = 1010 0110

So, the two's complement of 5A in hexadecimal is A6.

Here is another example to find the two's complement of the hexadecimal integer B3:

1. Convert B3 to binary:

1011 0011

2. Reverse all the bits:

0100 1100

3. Add 1 to the result:

0100 1100 + 0000 0001 = 0100 1101

So, the two's complement of B3 in hexadecimal is 4D.

SIGNED BINARY TO DECIMAL

Starting value	11110000
Step 1: Reverse the bits	00001111
Step 2: Add 1 to the value from Step 1	00001111 + 1
Step 3: Create the two's complement	00010000
Step 4: Convert to decimal	16

CONVERSION OF -43 SIGNED NUMBER TO BINARY

Here are the steps to convert -43 to binary:

1. Convert 43 to binary: **101011**
2. Invert all the bits: **010100**
3. Add 1 to the result: **010101**

Therefore, the binary representation of -43 in 8-bit two's complement notation is: **10101011**. Note that the leftmost bit (1) indicates that this is a negative number.

MAXIMUM AND MINIMUM VALUES FOR SIGNED INTEGERS

Type	Range	Storage Size in Bits
Signed byte	-2^7 to $+2^7 - 1$	8
Signed word	-2^{15} to $+2^{15} - 1$	16
Signed doubleword	-2^{31} to $+2^{31} - 1$	32
Signed quadword	-2^{63} to $+2^{63} - 1$	64
Signed double quadword	-2^{127} to $+2^{127} - 1$	128

CHARACTER SET - ASCII

Computers represent characters using character sets, which **map each character to a unique integer**. ASCII is a commonly used character set that assigns a unique 7-bit integer to each character.

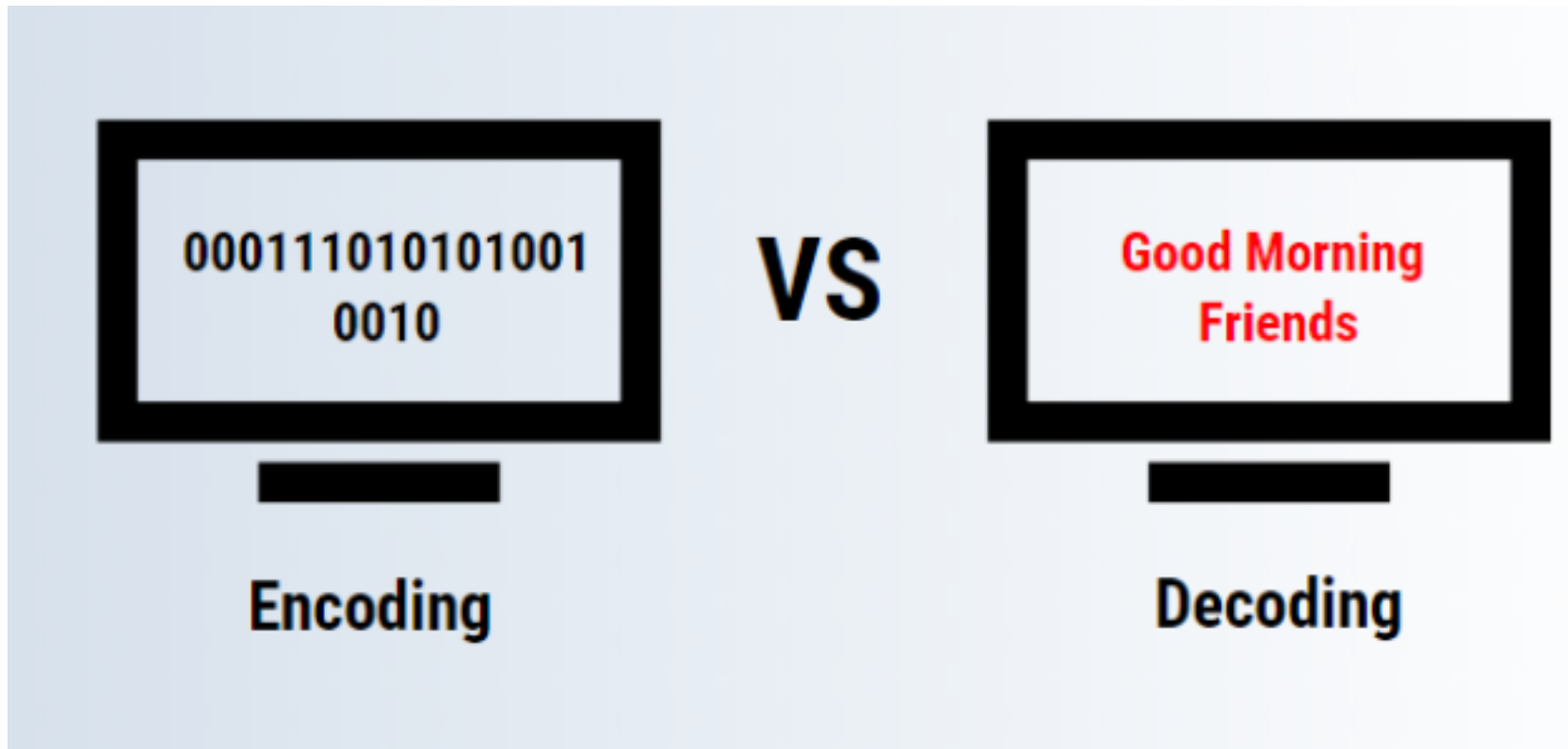
The first 128 characters of the 8-bit ANSI character set correspond to letters and symbols on a standard US keyboard, while the second 128 characters represent special characters from international alphabets, accents, currency symbols, and fractions.

Unicode is a universal standard for defining characters and symbols, which defines numeric codes for characters used in all major languages and scripts. Unicode has three transformation formats: UTF-8, UTF-16, and UTF-32.

Strings of characters are stored in memory as a sequence of bytes containing the corresponding character codes, and null-terminated strings are strings followed by a single byte containing zero.

UTF-8, UTF-16, UTF-32

In computers, **encoding** is the process of putting a sequence of characters (letters, numbers, punctuation, and certain symbols) into a specialized format for efficient transmission or storage. **Decoding** is the opposite process - the conversion of an encoded format back into the original sequence of characters.



UTF-8, UTF-16, and UTF-32 are three different encoding formats used to represent Unicode characters.

UTF-8

UTF-16
UCS-2

UNICODE ENCODINGS

UTF-32



- **UTF-8:** It is a variable-length encoding format that uses 8-bit code units to represent characters. In UTF-8, the first 128 characters (code points) are represented using a single byte. The remaining characters are represented using two, three, or four bytes, depending on their Unicode code point value. UTF-8 is the most widely used encoding format on the web and in email systems because it is backwards compatible with ASCII and can represent any Unicode character.
- **UTF-16:** It is a variable-length encoding format that uses 16-bit code units to represent characters. In UTF-16, the first 65,536 characters are represented using a single 16-bit code unit. The remaining characters are represented using two 16-bit code units (surrogate pairs). UTF-16 is used in some operating systems and programming languages, such as Microsoft Windows and Java.
- **UTF-32:** It is a fixed-length encoding format that uses 32-bit code units to represent characters. In UTF-32, each character is represented using a single 32-bit code unit. UTF-32 is less commonly

used than UTF-8 and UTF-16 because it requires more storage space than the other two formats. It is used in some programming languages, **such as C and C++.**

ASCII STRINGS

ASCII strings are sequences of characters that are represented in memory as a succession of bytes containing ASCII codes. ASCII codes are numeric values that are assigned to each character in the ASCII character set. For example, the ASCII code for the capital letter "A" is 65, and the code for the digit "0" is 48.

To store a string of ASCII characters in memory, each character is represented by its corresponding ASCII code. For example, the string "ABC123" is stored in memory as the hexadecimal values 41h, 42h, 43h, 31h, 32h, and 33h, which correspond to the ASCII codes for the characters 'A', 'B', 'C', '1', '2', and '3', respectively.

A null-terminated string is a string of characters followed by a single byte containing zero. This zero byte is called a null terminator and is used to mark the end of the string. The C and C++ programming languages use null-terminated strings as the standard way to represent strings in memory. Many Windows operating system functions also require strings to be in this format.

When working with ASCII strings, it is important to keep in mind that each character requires one byte of storage, regardless of its actual size. This can be a limitation when working with multibyte character sets such as Unicode, where some characters require more than one byte of storage.

Does 41h, 42h, represent the hex value of the memory location of that value 'a' and 'b' or it represents the hex equivalent of 'a' and 'b'??

The hexadecimal values 41h, 42h, and 43h represent the ASCII codes for the characters 'A', 'B', and 'C', respectively. These values **are not the memory locations** of the characters in memory, but rather the numeric values that represent these characters in the ASCII character set.

the numeric values that represent these characters in the ASCII character set.

When a string of characters is stored in memory, each character is represented by its corresponding ASCII code. In the example given, the string "ABC123" is stored in memory as the hexadecimal values 41h, 42h, 43h, 31h, 32h, and 33h. These values represent the ASCII codes for the characters 'A', 'B', 'C', '1', '2', and '3', respectively.

To retrieve the characters from memory, the program must interpret the bytes as ASCII codes and convert them to their corresponding characters. In the case of the string "ABC123", the program would read the bytes from memory and interpret them as the characters 'A', 'B', 'C', '1', '2', and '3', in that order.

USING ASCII TABLE

Using the ASCII Table A table on the inside back cover of this book lists ASCII codes used when running in Windows Console mode.

To find the hexadecimal ASCII code of a character, look along the top row of the table and find the column containing the character you want to translate.

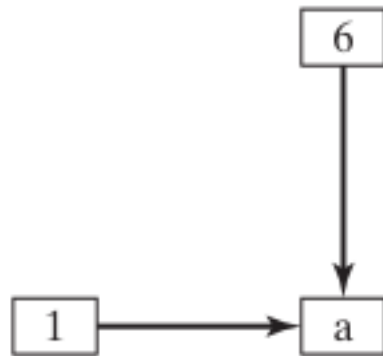
The most significant digit of the hexadecimal value is in the second row at the top of the table; the least significant digit is in the second column from the left.

For example, to find the ASCII code of the letter a, find the column containing the a and look in the second row.

The first hexadecimal digit is 6. Next, look to the left along the row containing a and note that the second column contains the digit 1.

Therefore, the ASCII code of a is 61 hexadecimal.

This is shown as follows in simplified form:



ASCII CONTROL CHARACTERS

ASCII control characters are a set of **33 non-printable characters** with ASCII codes in the range 0 through 31. These characters are often referred to as "control codes" because they are used to control various functions of the computer, such as printing, cursor movement, and data transmission.

When a program writes these control characters to standard output (or any other output device), they will carry out predefined actions instead of being displayed as visible characters.

For example, the ASCII code 10h (hexadecimal) represents the **newline character**, which causes the cursor to move to the beginning of the next line.

Similarly, the ASCII code 08h (hexadecimal) represents the **backspace character**, which moves the cursor back one position and allows overwriting of the previous character.

In C++, you can output ASCII control characters to standard output using escape sequences, such as `'\n'` for newline and `'\b'` for backspace.

ASCII table at the start of this book.

TERMINOLOGY IN DATA REPRESENTATION

It is important to use precise terminology when describing the way numbers and characters are represented in memory and on the display screen. Decimal 65, for example:

- Is stored in memory as a single binary byte as **01000001**.
- A debugging program would probably display the byte as “41,” which is the number’s hexadecimal representation.
- If the byte were copied to video memory, the letter “A” would appear on the screen because **01000001** is the ASCII code for the letter A.

Because a number’s interpretation can depend on the context in which it appears, we assign a specific name to each type of data representation to clarify future discussions:

- A **binary integer** is an integer stored in memory in its raw format, ready to be used in a calculation. Binary integers are stored in multiples of 8 bits (such as 8, 16, 32, or 64).
- A **digit string** is a sequence of characters that represent a number. In the case of your example, “123” is a digit string representing the number one hundred twenty-three.

The table you mentioned is likely a reference to the various formats that a number can be represented in. For example, the decimal number 65 can also be represented in binary as "1000001" and in hexadecimal as "41". Here are the formats for the number 65:

- Binary digit string: "01000001".
- Decimal digit string: "65".
- Hexadecimal digit string: "41".
- Octal digit string: "101".

SHIFTS AND ROTATES

Shifts and rotates are operations performed on binary values. These operations are commonly used in computer programming and digital circuit design to manipulate binary data.

A shift operation involves moving the bits of a binary value left or right by a certain number of positions. A left shift (<<) moves the bits to the left, and a right shift (>>) moves the bits to the right. For example, shifting the binary value 1010 (decimal 10) one position to the left results in the binary value 10100 (decimal 20).

A rotate operation is similar to a shift operation, but the bits that are shifted out of one end are moved to the other end. There are two types of rotate operations: a left rotate (ROL) and a right rotate (ROR). For example, rotating the binary value 1010 (decimal 10) one position to the left results in the binary value 0101 (decimal 5), with the bit that was shifted out at the left end being moved to the right end.

Shift and rotate operations can be useful for manipulating data in certain ways. For example, shifting a binary value to the left is equivalent to multiplying it by 2, and shifting it to the right is equivalent to dividing it by 2. Rotates can be useful for circular buffer operations or for

certain encryption algorithms.

More on this can be found online.