Chapter 2
Data Representation

2.1 Decimal and Binary Representation

- The number system we normally use is the **decimal** system.
- It uses 10 as the base.
- But a number system can use any base.
- Computers work with the **binary** system (base 2).
- Other systems used with computers are **octal** (base 8) and **hexadecimal** (base 16).

2.1 Decimal and Binary Representation

Bases and Exponents

Any number squared means that number times itself.

Example: **10** is the base and **2** is the exponent:

- $10^2 = 10*10 = 100$

When a number is raised to a positive integer power, it is multiplied by itself that number of times.

Example: the base is **5** and the exponent is **6**:

$5^6 = 5*5*5*5*5*5 = 15,625$

Exception 1: When a non-zero number is raised to the power of 0, the result is always 1.

- $5,345^0 = 1$
- $4^0 = 1$
- $(-31)^0 = 1$

Exception 2: 0^0 is undefined

The Decimal System

The first eight columns of the decimal system

10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
10,000,000	1,000,000	100,000	10,000	1,000	100	10	1
ten-millions	millions	hundred-thousands	ten-thousands	thousands	hundreds	tens	ones

Expanded Notation

The ten digits that are used in the decimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Any number in the decimal system can be written as a sum of each digit multiplied by the value of its column. This is called **expanded notation**.

The number **6,825** in the decimal system actually means:

```
  5*10⁰ = 5*1      =      5
+ 2*10¹ = 2*10     =     20
+ 8*10² = 8*100    =    800
+ 6*10³ = 6*1,000  =  6,000
                      6,825
```

Therefore, **6,825** can be expressed as: $6*10^3 + 8*10^2 + 2*10^1 + 5*10^0$

The Binary System

The binary system follows the same rules as the decimal system.

The difference is that the **binary system** uses a **base of 2** and has only two digits (**0** and **1**).

The rightmost column of the binary system is the one's column (2^0). It can contain a **0** or a **1**.

The next number after one is two; in binary, a two is represented by a **1** in the two's column (2^1) and a 0 in the one's column (2^0)

- one-hundred in decimal is represented by a **1** in the one-hundred's (10^2) column and 0s in the ten's column (10^1) and the one's column (10^0)
- in binary, a **1** in the 2^2's column represents the number **4**; i.e. **100**

The Binary System

The first eight columns of the binary system								
Power of 2	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal value	128	64	32	16	8	4	2	1
Binary representation								

Converting the Decimal Number 29_{10} to Binary

29 is less than **32** but greater than **16** so put a **1** in the **16's (2^4)** column.

29 – 16 = 13

13 is less than **16** but greater than **8** so put a **1** in the eight's **(2^3)** column

13 – 8 = 5

5 is less than **8** but greater than **4** so put a **1** in the four's **(2^2)** column

5 – 4 = 1

1 is less than **2** so there is nothing in the two's **(2^1)** column

Put a **0** in the two's column

You have **1** left so put a **1** in the one's **(2^0)** column

Therefore, $29_{10} = 11101_2$

Power of 2	2^5	2^4	2^3	2^2	2^1	2^0
Decimal value	32	16	8	4	2	1
Binary representation	0	1	1	1	0	1

Converting the Decimal Number 172_{10} to Binary

There is one **128** in **172** so put a **1** in the **128's (2^7)** column

172 – 128 = 44

44 is less than **64** so put a **0** in the **64's (2^6)** column

44 is less than **64** but greater than **32** so put a **1** in the **32's (2^5)** column

44 – 32 = 12

12 is less than **16** but greater than **8** so put a **0** in the **16's (2^4)** column and a **1** in the eight's **(2^3)** column

12 – 8 = 4

Put a **1** in the four's **(2^2)** column

4 – 4 = 0

Put **0s** in the last two columns

Power of 2	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal value	128	64	32	16	8	4	2	1
Binary representation	1	0	1	0	1	1	0	0

Converting Binary to Decimal

To convert a binary number back to decimal, just add the value of each binary column.

Convert the binary number 1011_2 to decimal

- There is a **1** in the one's column
- There is a **1** in the two's column so the value of that column is **2**
- There is a **0** in the four's column so the value of that is **0**
- There is a **1** in the eight's column so the value of that column is **8**

1 + 2 + 0 + 8 = 11

- Therefore, $1011_2 = 11_{10}$

Convert the Binary Number 10101010_2 to Decimal

There is a **0** in the one's column

There is a **1** in the two's column so the value of that column is **2**

There is a **0** in the four's column so the value of that is **0**

There is a **1** in the eight's column so the value of that column is **8**

There is a **0** in the **16**'s column

There is a **1** in the **32**'s column so the value of that column is **32**

There is a **0** in the **64**'s column

There is a **1** in the **128**'s column so the value of that column is **128**

0 + 2 + 0 + 8 + 0 + 32 + 0 + 128 = 170

Therefore, $10101010_2 = 170_{10}$

2.2 The Hexadecimal System

The hexadecimal system uses a base of **16**.

- there is a one's column (**16^0**)
- a **16**'s column (**16^1**)
- a **256'**s column (**16^2**)
- a **4,096**'s column (**16^3**)
- a **65,536**'s column (**16^4**)
- and so forth

Rarely need to deal with anything larger than the **16^4**'s column.

The hexadecimal system makes it easier for humans to read binary notation.

The Hexadecimal System

The first five columns of the hexadecimal system					
Hexadecimal column	16^4	**16^3**	**16^2**	**16^1**	**16^0**
	16*16*16*16	**16*16*16**	**16*16**	**16**	**1**
Decimal equivalent	**65,536**	**4,096**	**256**	**16**	**1**

Hexadecimal Digits

- The decimal system uses 10 digits (0 through 9) in each column (base 10)
- The binary system uses two digits (0 and 1) in each column (base 2)
- The hexadecimal system uses 16 digits in each column (base 16)
- How can you represent 10 in the one's column in base 16?
  - no way to distinguish "ten" (written as 10) from "sixteen" (also written as 10 → a one in the 16's column and a zero in the one's column)
- Use uppercase letters to represent the digits 10 through 15
- **hexadecimal digits** are 0 through 9 and A through F
  - $\mathbf{10_{10}}$ is represented as $\mathbf{A_{16}}$
  - $\mathbf{11_{10}}$ is represented as $\mathbf{B_{16}}$
  - $\mathbf{12_{10}}$ is represented as $\mathbf{C_{16}}$
  - $\mathbf{13_{10}}$ is represented as $\mathbf{D_{16}}$
  - $\mathbf{14_{10}}$ is represented as $\mathbf{E_{16}}$
  - $\mathbf{15_{10}}$ is represented as $\mathbf{F_{16}}$

Converting the Decimal Number 23_{10} to Hexadecimal

There is one **16** in $\mathbf{23_{10}}$ so put a **1** in the **16's** column

- **23 – 16 = 7** so put a **7** in the **1**'s column
- Therefore, $\mathbf{23_{10} = 17_{16}}$

Power of 16	16^3	16^2	16^1	16^0
Decimal value	4096	256	16	1
Hexadecimal representation			1	7

Converting the Decimal Number 875_{10} to Hexadecimal

875 is less than **4,096** but greater than **256** so there is nothing in the **4,096**'s (**16^3**) column

Divide **875** by **256** to see how many **256**s there are

875 ÷ 256 = 3 with a remainder of **107**

Put a **3** in the **256**'s column

107 ÷ 16 = 6 with a remainder of **11**

Put a **6** in the **16**'s column

11 in decimal notation = **B** in hexadecimal notation

Put a **B** in the one's column

Therefore, $\mathbf{875_{10} = 36B_{16}}$

Power of 16	16^3	16^2	16^1	16^0
Decimal value	4096	256	16	1
Hexadecimal representation		3	6	B

Converting the Hexadecimal Number $123D_{16}$ to Decimal

In expanded notation, this hexadecimal number is:

(1*4096) + (2*256) + (3*16) + (D*1)

D in hexadecimal is **13** in decimal, so:

4096 + 512 + 48 + 13 = 4669

Therefore, $\mathbf{123D_{16} = 4669_{10}}$

Power of 16	16^3	16^2	16^1	16^0
Decimal value	4096	256	16	1
Hexadecimal representation	1	2	3	D

Using Hexadecimal Notation

Notice:

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

Using Hexadecimal Notation

It is common to write a long binary number in hexadecimal notation.
The 15 hexadecimal digits represent all combinations of a 4-bit binary number.
Convert the following binary number to hexadecimal notation:

1110101000001111_2

1. Separate the binary number into sets of 4 digits:

   1110 1010 0000 1111

2. Refer to the table, if necessary, to make the conversions

   $1110_2 = E_{16}$
   $1010_2 = A_{16}$
   $0000_2 = 0_{16}$
   $1111_2 = F_{16}$

Therefore, 1110101000001111_2 is $EA0F_{16}$

2.3 Integer Representation

- How computers process numbers depends on each number's **type**.
- **Integers** are stored and processed in quite a different manner from **floating point** numbers.
- Even within the broad categories of integers and floating point numbers, there are more distinctions.
- Integers can be stored as unsigned numbers (all nonnegative) or as signed numbers (positive, negative, and zero).
- Floating point numbers also have several variations.

Unsigned Integer Format

- A computer processes information in the form of **bytes.**
- Bytes are normally 8 to 16 bits.
- To store $\mathbf{11}_2$ and $\mathbf{101101}_2$ both must have the same length as a byte.
- Do this by adding **0**s to the left of the number to fill up as many places as needed for a byte.
- This is called the **unsigned form of an integer.**

Unsigned Binary Integers

Store the decimal integer $\mathbf{5}_{10}$ in an 8-bit memory location:

Convert $\mathbf{5}_{10}$ to binary: $\mathbf{101}_2$

Add five **0**s to the left to make 8 bits:

$\mathbf{00000101}_2$

Store the decimal integer $\mathbf{928}_{10}$ in a 16-bit memory location:

Convert $\mathbf{928}_{10}$ to binary: $\mathbf{1110100000}_2$

Add six **0**s to the left to make 16 bits:

$\mathbf{0000001110100000}_2$

Overflow

If you try to store an unsigned integer that is bigger than the maximum unsigned value that can be handled by that computer, you get a condition called **overflow**.

Store the decimal integer $\mathbf{23}_{10}$ in a 4-bit memory location:

→ range of integers available in 4-bit location is $\mathbf{0}_{10}$ through $\mathbf{15}_{10}$

Therefore, attempting to store $\mathbf{23}_{10}$ in a 4-bit location gives an overflow.

Store the decimal integer $\mathbf{65,537}_{10}$ in a 16-bit memory location:

→range of integers available in 16-bit location is $\mathbf{0}_{10}$ through $\mathbf{65535}_{10}$

Therefore, attempting to store this number in a 16-bit location gives an overflow.

Range of Unsigned Integers

Number of Bits	Range
8	0...255
16	0...65,535
32	0...4,294,967,295
64	0...18,446,740,000,000,000,000 (approximate)

Sign-and-Magnitude Format

The simple method to convert a decimal integer to binary works well to represent positive integers and zero.

We need a way to represent negative integers.

The **sign-and-magnitude format** is one way.

The leftmost bit is reserved to represent the **sign.**

The other bits represent the **magnitude** (or the absolute value) of the integer.

Store the decimal integer $+23_{10}$ in an 8-bit memory location using sign-and-magnitude format

Convert 23_{10} to binary: 10111_2

Since this is an 8-bit memory location, 7 bits are used to store the magnitude of the number.

10111_2 uses 5 bits so add two **0**s to the left to make up 7 bits: 0010111_2

Finally, look at the sign. This number is positive so add a **0** in the leftmost place to show the positive sign.

Therefore, $+23_{10}$ in sign-and-magnitude format in an 8-bit location is 00010111_2

Store the decimal integer -19_{10} in a 16-bit memory location using sign-and-magnitude format

Convert 19_{10} to binary: 10011_2

Since this is a 16-bit memory location, 15 bits are used to store the magnitude of the number.

10011_2 uses 5 bits so add ten **0**s to the left to make up 15 bits: 000000000010011_2

Finally, look at the sign. This number is negative so add a **1** in the leftmost place to show the negative sign.

Therefore, -19_{10} in sign-and-magnitude format in an 8-bit location is 1000000000010011_2

The Problem of the Zero

(a) Store the decimal integer 0_{10} in an 8-bit memory location using sign-and-magnitude format:

Convert 0_{10} to binary: 0_2

Since this is an 8-bit memory location, 7 bits are used to store the magnitude of the number.

The number 0_2 uses 1 bit so add six **0**s to the left to make up 7 bits: 0000000_2

Look at the sign. Zero is considered a non-negative number so you should add a **0** in the leftmost place to show that it is not negative.

Therefore, 0_{10} in sign-and-magnitude in an 8-bit location is: 00000000_2

(b) ... but...given that 10000000_2 is an 8-bit binary integer in sign-and-magnitude form, find its decimal value:

First convert the rightmost 7 bits to decimal to get 0_{10}

Look at the leftmost bit; it is a **1**. So the number is negative.

Therefore, 10000000_2 represents the decimal integer -0_{10}

One's Complement Format

The fact that 0 has two possible representations in sign-and-magnitude format is one of the main reasons why computers usually use a different method to represent signed integers.

There are two other formats that may be used to store signed integers.

The one's complement method is not often used, but it is explained here because it helps in understanding the most common format: two's complement.

To **complement** a binary digit, you simply change a **1** to a **0** or a **0** to a **1**.

In the **one's complement** method, positive integers are represented as they would be in sign-and-magnitude format. The leftmost bit is still reserved as the sign bit.

$+6_{10}$, in a 4-bit allocation, is still 0110_2

In one's complement, -6_{10} is just the complement of $+6_{10}$

-6_{10} becomes 1001_2

The range of one's complement integers is the same as the range of sign-and-magnitude integers.

BUT... there are still two ways to represent the zero.

Store the decimal integer -37_{10} in an 8-bit memory location using one's complement format

Convert 37_{10} to binary: 100101_2

Since this is an 8-bit memory location, 7 bits are used to store the magnitude

The number 100101_2 uses 6 bits so add one **0** to the left to make up 7 bits:

0100101_2

This number is negative. Complement all the digits by changing all the **0**s to **1**s and all the **1**s to **0**s

Add a **1** in the 8th bit location because the number is negative

Therefore, -37_{10} in one's complement in an 8-bit location is 11011010_2

Converting One's Complement to Decimal

To convert a one's complement number back to decimal:

Look at the leftmost bit to determine the sign.

If the leftmost bit is **0**, the number is positive and can be converted back to decimal immediately.

If the leftmost bit is a **1**, the number is negative.

- Un-complement the other bits (change all the **0**s to **1**s and all the **1**s to **0**s)
- then convert the binary bits back to decimal
- Remember to include the negative sign when displaying the result!

The Problem of the Zero Again

a) Store the decimal integer 0_{10} in an 8-bit memory location using one's complement format:

Convert 0_{10} to binary: 0_2

Since this is an 8-bit memory location, 7 bits are used to store the magnitude

The number 0_2 uses 1 bit so add six **0**'s to the left to make up 7 bits: 0000000_2

Zero is considered non-negative so add a **0** in the leftmost place

Therefore, 0_{10} in one's complement in an 8-bit location is 00000000_2

(b) but... given that 11111111_2 is a binary number in one's complement form, find its decimal value:

Look at the leftmost bit. It is a **1** so you know the number is negative

Since the leftmost bit is **1**, all the other bits have been complemented.

"un-complement" them to find the magnitude of the number.

When you un-complement 1111111_2, you get 0000000_2

Therefore, 11111111_2 in one's complement represents the decimal integer -0_{10}

Why the Fuss About Nothing?

Why is there so much fuss about the zero?

Why not just define zero in binary as **0000_2** (or **00000000_2** or **0000000000000000_2**) and be done with it?

In a 4-bit allocation, the bit-pattern **1111_2** still exists. Unless the computer knows what to do with it, the program will get an error. It might even not work at all.

One possible scenario: If the result of a calculation using one's complement was **1111_2**, the computer would read this as **–0**. If you then tried to add **1** to it, what would the answer be?

- The number that follows **1111_2** in a 4-bit allocation is **0000_2**.
- That would mean, using one's complement, that **–0 + 1 = +0**. This certainly would not be an irrelevant issue!

Two's Complement Integers

To find the two's complement of an **X**-bit number:

1. If the number is positive, just convert the decimal integer to binary and you are finished.
2. If the number is negative, convert the number to binary and find the one's complement.
3. Add a binary **1** to the one's complement of the number.
4. If this results in an extra bit (more than **X** bits), discard the leftmost bit.

Rules of Binary Addition

Rule	1 + 0 = 1	1 + 1 = 10
Example 1	1 0 + 1 1 1	1 + 1 1 0
Example 2	1 0 1 + 1 0 1 1 1	1 1 + 1 1 0 0
Example 3	1 0 0 + 1 1 0 1	1 0 1 + 1 1 1 0

Finding the Two's Complement of 8-bit Binary Integers

Find the two's complement of $+43_{10}$ as an 8-bit binary integer:

Convert 43_{10} to binary: 101011_2

Add zeros to the left to complete 8 bits: 00101011

Since this is already a positive integer, you are finished.

Find the twos complement of -43_{10} as an 8-bit binary integer:

Convert 43_{10} to binary: 101011_2

Add zero's to the left to complete 8 bits: 00101011

Since the number is negative, do the one's complement to get:

11010100

Now add binary 1 to this number:

```
 11010100
+       1
 11010101
```

Therefore, -43_{10} in two's complement in an 8-bit location is 11010101

Carrying the 1 With Binary Addition

Find the two's complement of -24_{10} as an 8-bit binary integer:

Convert 24_{10} to binary: 11000_2

Add zeros to the left to complete 8 bits: **00011000**

Since the number is negative, do the one's complement to get:

11100111

Now add binary **1** to this number:

```
 11100111
+       1
 11101000
```

Therefore, -24_{10} in two's complement in an 8-bit location is **11101000**

When the Two's Complement Cannot Be Done

Find the two's complement of -159_{10} as an 8-bit binary integer:

Convert 159_{10} to binary: 10011111_2

10011111 already takes up 8 bits so there is nothing left for the sign bit

Therefore, -159_{10} cannot be represented as a two's complement binary number in an 8-bit location.

The Zero Solution

a. Find the two's complement of $+0_{10}$ as an 8-bit binary integer:

Convert 0_{10} to 8-bit binary: 00000000_2

The number is positive so nothing more needs to be done

Therefore, **+0** in two's complement in an 8-bit location is **00000000**

b. Find the two's complement of -0_{10} as an 8-bit binary integer:

Convert 0_{10} to 8-bit binary: 00000000_2

Since the number is negative, do the one's complement to get: **11111111**

Now add binary **1** to this number:

```
 11111111
+       1
---------
100000000
```

Recall that Step 4 in the rules for converting to two's complement states that, after the addition of **1** to the one's complement, any digits to the left of the maximum number of bits (here, 8 bits) should be discarded. Discard the leftmost **1**

Therefore, -0_{10} in two's complement in an 8-bit location is 00000000_2 which is exactly the same as $+0_{10}$

Why the Method Works

How in the world does flipping digits and then adding 1 somehow end up with the negative of the number you started with?

Example: using a 4-bit allocation, since it is easy to manage.

- A 4-bit allocation allows for 16 binary numbers ranging from **0000** to **1111**, or 0_{10} to 15_{10}.
- Define the "flip side" of any number between **0** and **16** to be **16** minus that number.
  - using 4 bits, there are 16 possible numbers (0_{10} to 15_{10}), so the flip side of a number between **0** and **16** would be **16** minus that number.
  - the flip side of **4** is $16 - 4 = 12$.
  - in two's complement, the negative of a number is represented as the flip side of its positive value.
  - using two's complement notation, a -3_{10} is represented as the flip side of $+3_{10}$.
  - In a 4-bit location, this would be $16 - 3 = 13$. In an 8-bit location, this would be $256 - 3 = 253$ because $2^8 = 256$.

In mathematical terms, this can be expressed as follows (assuming an **X**-bit memory allocation):

- For a number, **N**, the two's complement is:

$$2^X - |N|$$ where $|N|$ denotes the absolute value of N

2.4 Floating Point Representation

All floating point numbers have both an integer part and a fractional part, even if that fractional part is 0.

- Note: **6** is an integer but **6.0** is a floating point number

To represent a floating point number in binary:

- divide the number into its parts:
  - the **sign** (positive or negative)
  - the **whole number** (integer) part
  - the **fractional** part

The Integer Part

A specific bit is set aside to denote the sign.

To convert the integer part to binary, simply convert the same way you convert positive integers to binary.

The integer part of a floating point binary number is separated from the fractional part.

The dot (or period) between the integer and fractional parts of a binary number will be referred to as a **point.**

The point is, in effect, a **binary point**
- it does the same thing as a decimal point in the decimal system.

The Fractional Part

We know that the columns in the integer part of a binary number represent powers of **2**.
- The first column, 2^0 is the one's column; the second column, 2^1 is the two's column; and so on.

We can think of the fractional part in similar terms.
- The decimal number **0.1** represents $\frac{1}{10}$, the decimal number **0.01** represents $\frac{1}{100}$ and so on.
- As the denominators get smaller, each decimal place is **10** raised to the next power.
- In decimal notation: $\frac{1}{10^1}, \frac{1}{10^2}, \frac{1}{10^3},$ etc.
- Also can be represented as 10^{-1}, 10^{-2}, 10^{-3}, etc.

Fractional Part of the Binary System

The first six columns of the fractional part of a number in the binary system					
0.1	**0.01**	**0.001**	**0.0001**	**0.00001**	**0.000001**
$\frac{1}{2^1} = 2^{-1}$	$\frac{1}{2^2} = 2^{-2}$	$\frac{1}{2^3} = 2^{-3}$	$\frac{1}{2^4} = 2^{-4}$	$\frac{1}{2^5} = 2^{-5}$	$\frac{1}{2^6} = 2^{-6}$
0.5	0.25	0.125	0.0625	0.03125	0.015625
halves	fourths	eighths	sixteenths	thirty-seconds	sixty-fourths

Converting a Decimal Fraction to Binary

1. How many bits are allowed for the fractional part of a given number?
2. Create a chart:

Binary	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
Decimal	0.5	0.25	0.125	0.0625	0.03125	0.015625
Conversion						

3. As you work, you can fill in the boxes in the third row.
4. If the number is equal to or greater than **0.5**, put a **1** in the 2^{-1} column. Otherwise put a **0** in the 2^{-1} column. Then subtract **0.5** from the decimal number. If the result is **0**, you are done.
5. If the result is equal to or greater than **0.25**, put a **1** in the 2^{-2} column. Then subtract **0.25** from the decimal number. If the result is **0**, you are done.
6. If the result of your subtraction is less than **0.25**, put a **0** in the 2^{-2} column. Look at the next column. If your number is less than **0.125**, put a **0** in the 2^{-3} column
7. Repeat with each subsequent column until the subtraction either gives a result of **0** or until you reach the end of the bits required.

Convert the Decimal Number 0.4 to a 6-bit Binary Number

This number is less than **0.5**, so put a **0** in the 2^{-1} column

The number is greater than **0.25**, so put a **1** in the 2^{-2} column, then subtract: **0.4 – 0.25 = 0.15**

0.15 is greater than **0.125**, so put a **1** in the 2^{-3} column and subtract: **0.15 – 0.125 = 0.025**

0.025 is less than the next column, **0.0625**, so put a **0** in the 2^{-4} column

0.025 is less than the next column, **0.03125**, so put a **0** in the 2^{-5} column

0.025 is greater than the next column, **0.015625**, so put a **1** in the 2^{-6} column

Even though there is a remainder when you subtract **0.025– 0.015625 = 0.009375**, you do not need to do anything more because the problem specified that only 6 bits are needed

0.4 in decimal = **0.011001** in a 6-bit binary representation

Binary	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
Decimal	0.5	0.25	0.125	0.0625	0.03125	0.015625
Conversion	0	1	1	0	0	1

Putting the Two Parts Together: Store the Decimal Number **75.804** as a Binary Number

1. Convert 75 to binary: 1001011
2. Convert 0.804 to binary:
   - o Put a 1 in the 2^{-1} column and subtract: 0.804 – 0.5 = 0.304
   - o Put a 1 in the 2^{-2} column and subtract: 0.304 – 0.25 = 0.054
   - o Put a 0 in the 2^{-3} column.
   - o Put a 0 in the 2^{-4} column.
   - o You do not need to do anything more because the problem specified that only 4 bits are needed for the fractional part.

Binary	2^{-1}	2^{-2}	2^{-3}	2^{-4}
Decimal	0.5	0.25	0.125	0.0625
Conversion	1	1	0	0

Therefore, $\mathbf{75.804_{10} = 101011.1100_2}$

2.5 Putting It All Together

Just converting a decimal floating point number to binary representation isn't enough.

There are several concepts to understand before seeing how a floating point number is actually represented inside the computer.

While you will probably use a calculator for conversions, it is valuable to understand the process and will prove helpful when writing programs.

Scientific Notation

Computers are used for many scientific applications which often use very large or very small numbers.

Example: the distance from Earth to our nearest star is 24,698,100,000,000 miles. We would need a 49-digit binary number to represent this in a computer.

Example: The diameter of an atom would require at least a 30-digit binary number.

Humans deal with these almost-impossible-to-read numbers with **scientific notation**.

Examples of Scientific Notation

In **scientific notation**, a given number is written as a number between 1 and 9 multiplied by the appropriate power of 10.

Examples:

680,000 = 6.8×10^{5}

1,502,000,000 = 1.502×10^{9}

8,938,000,000,000 = 8.938×10^{12}

0.068 = 6.8×10^{-2}

0.00001502 = 1.502×10^{-5}

0.000000000008938 = 8.938×10^{-12}

Exponential Notation

In programming, instead of writing **10**^{power}, we use the letter **E** followed by the given power. This is called **exponential notation**. Notice, you must include the sign of the exponent.

Examples:

```
680,000 = 6.8E+5
1,502,000,000 = 1.502E+9
8,938,000,000,000 = 8.938E+12
0.068 = 6.8E-2
0.00001502 = 1.502E-5
0.000000000008938 = 8.938E-12
```


Converting a Number from Exponential Notation to Ordinary Notation

Move the decimal point the number of places indicated by the integer following **E**

Examples:

Given **1.67E–4**

- write **1.67** and move the decimal point 4 places to the left, filling in 3 zeros before **1**
- This gives **0.000167**

Given **4.2E+6**

- move the decimal point 6 places to the right, filling in 5 zeros to the right of **2**
- This gives **4200000**, or as it is usually written, **4,200,000**

Base 10 Normalization

Normalized form is similar to scientific notation.

Each normalized number has two parts: the **scaled portion** and the **exponential portion**.

In scientific notation, the decimal point was moved so the first non-zero digit was immediately to the left of it.

In normalized form, the decimal is moved so the first non-zero digit is immediately to the right of it. The value of the number is always maintained.

To normalize a decimal number, after moving the decimal point, the number is multiplied by 10 raised to whatever power is necessary to return the number to its original value.

Normalized Decimal Numbers

Number	Scaled Portion	Exponential Portion	Normalized Form
371.2	0.3712	10^3	0.3712×10^3
40.0	0.4	10^2	0.4×10^2
0.000038754	0.38754	10^{-4}	0.38754×10^{-4}
−52389.37	−0.5238937	10^5	-0.5238937×10^5

Normalizing Binary Floating Point Numbers

The **IEEE Standard** is the most widely accepted standard for representation of floating point numbers in a computer and uses normalized binary numbers.

A **normalized binary number** consists of three parts:

- the **sign** part
- the **exponential** part
- the **mantissa**.

The mantissa is the binary equivalent to the scaled portion (as in previous slide)

The **Excess_127** system is used to represent the exponential portion

Excess_127

The Excess_127 system Is used to store the exponential value of a normalized binary number.

To represent an 8-bit number in the Excess_127 system:

- Add **127** to the number
- Change the result to binary
- Add zeros to the left to make up 8 bits

Examples:

(a) To represent $+9_{10}$

→ add **9 + 127 = 136**

→ Convert **136** to binary: **10001000**

→ $+9_{10}$ in Excess_127 is **10001000**

(b) To represent -13_{10}

→ add **(−13) + 127 = 114**

→ Convert **114** to binary: **01110010**

→ -13_{10} in Excess_127 is **0111001**

Base 2 Normalization

- The process is similar to the one followed to normalize a decimal number.
- The point is moved but it is moved so that the first non-zero number (a **1**) is immediately to the left of the point.
- Then multiply the number by the power of **2** needed to express the original value of the number.
- Not necessary to worry about the sign of the number since, in normalized form, the sign bit takes care of this.

Examples

Normalize the Binary Number **+10110**

- Move the point to the left 4 places to get **1.0110**
- Since the point was moved 4 places to the left, the number is then multiplied by 2^4 to get the original number
- +**10110** in normalized form is $2^4 \times 1.0110$

Normalize the Binary Number +**0.11110011**

- Move the point to the right 1 place to get **1.1110011**
- Since the point was moved 1 place to the right, the number needs to be multiplied by 2^{-1} to get the original number
- +**0.11110011** in normalized form is $2^{-1} \times 1.1110011$

Single Precision Floating Point Numbers

IEEE has defined standards for storing floating point numbers. The most common standard is **single precision** floating point.

In **single precision** format, a normalized floating point number has three parts.

- The **sign** is stored as a single bit
- The **exponent** is stored in 8 bits
- The **mantissa** is stored in the rest of the bits (23 bits)
- Single precision uses 32 bits in total to store one floating point number

There is also a **double precision** representation which allows for a much larger range of numbers.

- The **sign** of the number still uses one bit
- The **exponent** uses 11 bits
- The **mantissa** uses 52 bits.
- An 11-bit exponent uses the Excess_1023 system and can handle exponents up to ±1023
- Double precision uses 64 bits in total to store one floating point number

Converting a Decimal Number to Single Precision Floating Point Binary

1. **The sign bit**:

    If the number is positive, put a **0** in the leftmost bit. If it is negative, put a **1** in the leftmost bit.

2. **Convert the number to binary.**

    If there is an integer and a fractional part, convert the whole number to binary

3. **Normalize the number.**

    Move the point so it is directly to the right of the first non-zero number.

4. **Count the number of places you moved the point.** This is your **exponent**.

    If you moved the point to the right, your exponent is negative.

    If you moved the point to the left, your exponent is positive.

5. **The exponent part**:

    Convert your exponent to a binary number, using the Excess_127 system.

    Store this number in the 8 bits to the right of the sign bit.

6. **The mantissa**: Use the number from Step 3 is used to find the mantissa.

    When storing the normalized part of the number, **the 1 to the left of the point is discarded.**

    Everything to the right of the point is now called the mantissa.

Represent in single precision floating point the normalized number: $-2^{-9} \times 1.00001011$

The sign is negative so the leftmost bit is a **1**

The exponent is **–9**. Convert this to Excess_127:

- Add: **(–9) + 127 = 118**
- Convert **118** to binary: **01110110**
- Store this number in the next 8 bits

The rest of the number is **1.00001011**

- Discard the leftmost **1** (the one to the left of the point) and store the remainder of the number in the last 23 bits

This number takes up 8 bits while, in single-precision floating point, the mantissa is 23 bits long. You *must* add 15 **0**'s at the end to complete 23 bits.

Therefore, $-2^{-9} \times 1.00001011$ as a single-precision floating point number is

1 01110110 00001011000000000000000

Hexadecimal Representation

It is much easier to read a hexadecimal number than to read a long string of **0**s and **1**s.

Single precision floating point numbers are often changed to hexadecimal.

It's easy to convert binary to hexadecimal:

- Divide the binary number into groups of four digits
- Convert each group to a single hexadecimal number

Example (from previous slide):

1 01110110 00001011000000000000000

is

$\mathbf{BB058000}_{16}$