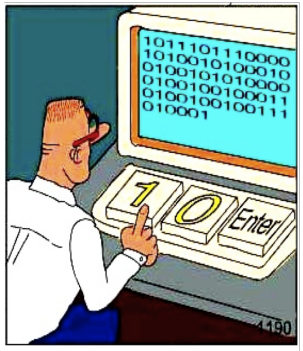
Numbers



are abstract objects used for counting and measuring.

There are several number types:

- Natural numbers $(0, 1, 2, 3, 4, 5, \ldots$ *ad infinitum*$)$
- Integer numbers $(\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots)$
- Rational numbers $(-\frac{7}{3}, \frac{4}{9}$, etc.;

  formally, they are of the form $\frac{m}{n}$, where m and n are integers and $n \neq 0$)
- Real numbers (like $\pi = 3.14159265358\ldots$, $\sqrt{2} = 1.41421356237\ldots$, etc.)

  see https://en.wikipedia.org/wiki/Irrational_number
- ...

How can we represent all these numbers in computers?

- Computer words are **binary** (no problem: binary can encode everything)
- Computer words are **finite**, usually of 32 or 64 bits

  (can lead—actually has already led—to disaster)

Numeral systems

A **numeral** is a symbol or group of symbols that represents a number

- **Unary:** numeral $|||||||$ means 7
- **Decimal:** numeral 456 means $(4 \times 10^2) + (5 \times 10^1) + (6 \times 10^0)$

  numeral 101_{10} means $(1 \times 10^2) + (0 \times 10^1) + (1 \times 10^0)$
- **Binary:** numeral 101_2 means $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$, i.e., decimal 5
- d-ary, for any $d > 1$: $a_{n-1}a_{n-2}\ldots a_1a_0$ means

$$a_{n-1} \times d^{n-1} + a_{n-2} \times d^{n-2} + \cdots + a_1 \times d^1 + a_0 \times d^0$$

From now on we assume that computer words are 32 bits long.

This means that we can represent the natural numbers from 0 to $2^{32} - 1$:

$$\begin{aligned}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 &= 0_{10} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 &= 1_{10} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 &= 2_{10} \\
\ldots & \quad \ldots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 &= 2^{32} - 2 = 4,294,967,294_{10} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 &= 2^{32} - 1 = 4,294,967,295_{10}
\end{aligned}$$

Converting decimal numbers to binaries: divide by 2

Rule: divide repeatedly by 2, keeping track of the remainders as you go

Example: covert 123_{10} to binary

$$
\begin{aligned}
123/2 &= 61 && \text{remainder } = 1\\
61/2 &= 30 && \text{remainder } = 1\\
30/2 &= 15 && \text{remainder } = 0\\
15/2 &= 7 && \text{remainder } = 1\\
7/2 &= 3 && \text{remainder } = 1\\
3/2 &= 1 && \text{remainder } = 1\\
1/2 &= 0 && \text{remainder } = 1
\end{aligned}
$$

The result is read from the last remainder upwards:

$$123_{10} = 1111011_2$$

why does it work?

A few binary numbers

Decimal Number	Binary Number
0	0
1	1
2=2^1	10
3	11
4=2^2	100
5	101
6	110
7	111
8=2^3	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16=2^4	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11011
26	11100
27	11011
28	11100
29	11101
30	11110
31	11111
32=2^5	100000

Binary addition and subtraction (for unsigned numbers)

- The four basic rules for adding binary digits are as follows:

$0 + 0 = 0$	sum of 0 with a carry of 0
$0 + 1 = 1$	sum of 1 with a carry of 0
$1 + 0 = 1$	sum of 1 with a carry of 0
$1 + 1 = 10$	sum of 0 with a carry of 1

Example:

Carry Carry Carry

$$\begin{array}{cccc} & \color{blue}{1} & \color{blue}{1} & \color{blue}{1} \\ & 0 & 1 & 1 \\ + & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \end{array}$$

- Formulate four rules for subtracting bits.
- What about multiplication?

Negative numbers: sign-magnitude representation

How to represent negative integer numbers?

Most obvious idea: treat the most significant (left-most) bit in the word as a **sign**:
if it is $\mathbf{0}$, the number is positive; if the left-most bit is $\mathbf{1}$, the number is negative;
the right-most $\mathbf{31}$ bits contain the **magnitude** of the integer

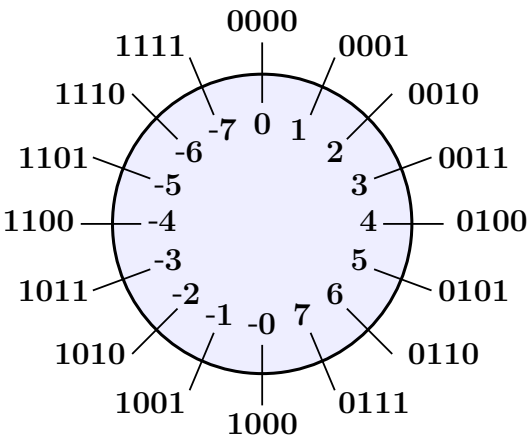
Example:

$$+18_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0010_2$$
$$-18_{10} = 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0010_2$$

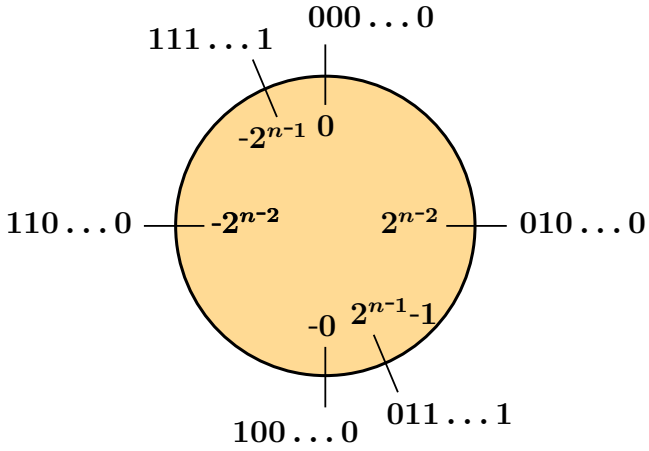
This representation was used in early machines, but several shortcomings have been revealed

- Awkward arithmetic
- Two zeros: $+0$ and -0
- ...

Sign-magnitude representation



4-bit numbers



n-bit numbers

Two's complement representation

- The most significant bit represents the **sign**, as before
- The positive numbers are also represented as before

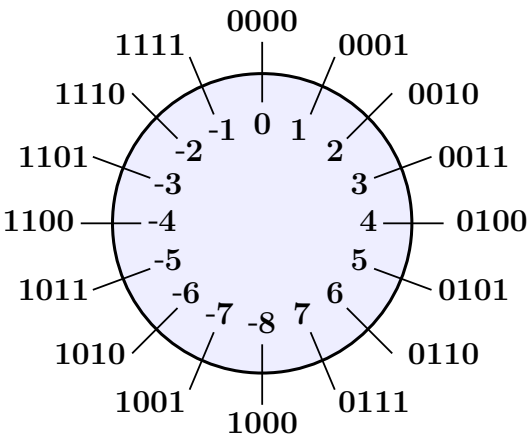
$$0a_{30}\ldots a_0 \quad \text{means} \quad 0\times 2^{31} + a_{30}\times 2^{30} + \cdots + a_1\times 2 + a_0$$

- To represent a negative number, take its **complement** to 2^{31}:

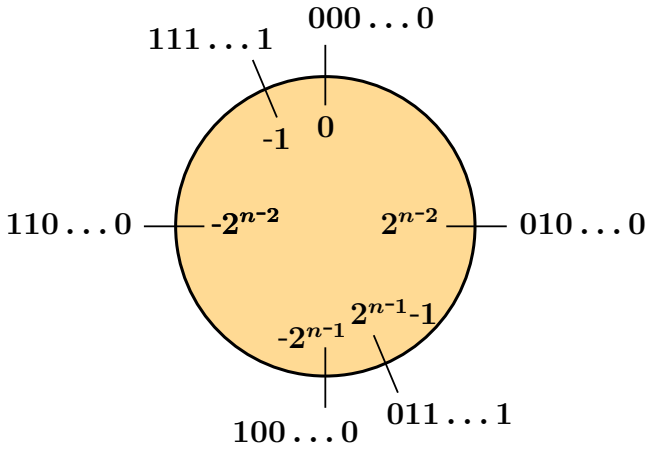
$$1a_{30}\ldots a_0 \quad \text{means} \quad -1\times 2^{31} + a_{30}\times 2^{30} + \cdots + a_1\times 2 + a_0$$

$$\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 &&= 0_{10}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 &&= 1_{10}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 &&= 2_{10}\\
&\cdots && \cdots\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 &&= 2,147,483,645_{10}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 &&= 2,147,483,646_{10}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 &&= 2,147,483,647_{10}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 &&= -2,147,483,648_{10}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 &&= -2,147,483,647_{10}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 &&= -2,147,483,646_{10}\\
&\cdots && \cdots\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 &&= -3_{10}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 &&= -2_{10}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 &&= -1_{10}
\end{aligned}$$

Two's complement representation (cont.)



4-bit numbers



n-bit numbers

Two's complement arithmetic: negation

$a_{31}a_{30}\ldots a_1a_0$ represents $-a_{31} \times 2^{31} + a_{30} \times 2^{30} + \cdots + a_1 \times 2 + a_0$

The rules for forming the **negation** of an integer (in two's complement notation):

- Take the Boolean negation of each bit of the integer (including the sign bit)
  That is, set each 1 to 0 and each 0 to 1.
- Treating the result as an unsigned binary integer, add 1.

Example: Negate $2_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2$

Invert the bits and add 1:

$$\begin{array}{rl} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_2 \\ \hline & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2\ (= -2_{10}) \end{array}$$

- Negate -2_{10} and check whether you get 2_{10}
- Negate 0. Any problem? Overflow; but is the result correct?
- Negate $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2,147,483,648_{10}$
  Any problem? Is the result correct? Why?

Two's complement addition and subtraction

- **Subtraction** is achieved using addition $x - y = x + (-y)$:
  to subtract y from x, negate y and add the result to x
- **Addition** is the same as the addition of unsigned numbers on page 5

The result of addition can be **larger** than can be held in a 32 bit word.
This situation is called **overflow**.

- Detecting overflow: if two numbers are added, and they are both positive or both negative, then overflow occurs if the result the opposite sign

Examples of overflow: assume that we deal with 4 **bit** words only

$$\begin{array}{rl} & 0101_2 = 5_{10} \\ + & 0100_2 = 4_{10} \\ \hline & 1001_2 = -7_{10} \end{array}$$ **overflow**

$$\begin{array}{rl} & 1001_2 = -7_{10} \\ + & 1010_2 = -6_{10} \\ \hline 1 & 0011_2 = 3_{10} \end{array}$$ **overflow**

When adding numbers with different signs, overflow cannot occur. Why?

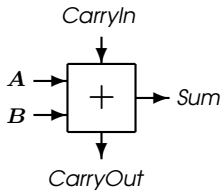
Arithmetic and logic unit (ALU)

The **ALU** is the part of computer that performs arithmetic and logical operations.
It is the workhorse of the CPU because it carries out all the calculations.

The hardware required to build an ALU is the basic gates: AND, OR and NOT
We use these gates to construct a 32-bit adder for integers
Such an adder can be created by connecting 32 1-bit adders

Inputs and outputs of a single-bit adder:

- two inputs for the operands (the bits we want to add), say, $\boldsymbol{A}$ and $\boldsymbol{B}$;
- a single-bit output for the sum;
- a second output to pass on the carry *CarryOut*
- a third input is *CarryIn* — the *CarryOut* from the previous adder



Designing a single-bit adder

We begin by giving precise input and output specifications (truth-table)

A	B	$CarryIn$	$CarryOut$	Sum	Comments
0	0	0	0	0	$0+0+0=00_2$
0	0	1	0	1	$0+0+1=01_2$
0	1	0	0	1	$0+1+0=01_2$
0	1	1	1	0	$0+1+1=10_2$
1	0	0	0	1	$1+0+0=01_2$
1	0	1	1	0	$1+0+1=10_2$
1	1	0	1	0	$1+1+0=10_2$
1	1	1	1	1	$1+1+1=11_2$

Then we turn this truth-table into logical equations:

$$CarryOut = (\neg A \wedge B \wedge CarryIn) \vee (A \wedge \neg B \wedge CarryIn) \vee (A \wedge B \wedge \neg CarryIn) \vee (A \wedge B \wedge CarryIn)$$

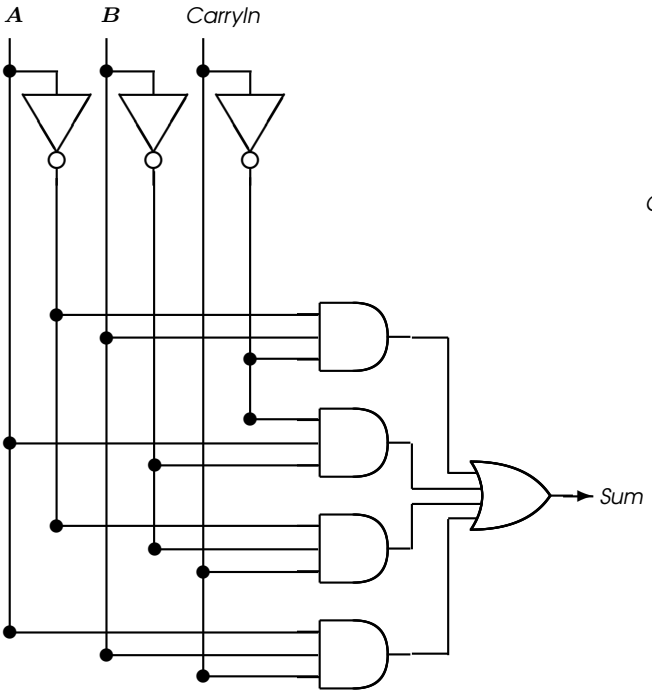
can be simplified to (remember the **majority function**?)

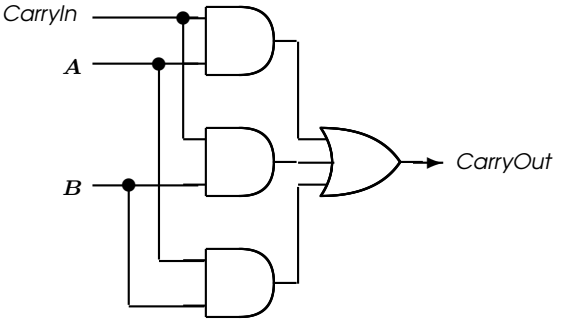
$$CarryOut = (B \wedge CarryIn) \vee (A \wedge CarryIn) \vee (A \wedge B)$$

and

$$Sum = (A \wedge \neg B \wedge \neg CarryIn) \vee (\neg A \wedge B \wedge \neg CarryIn) \vee (\neg A \wedge \neg B \wedge CarryIn) \vee (A \wedge B \wedge CarryIn)$$

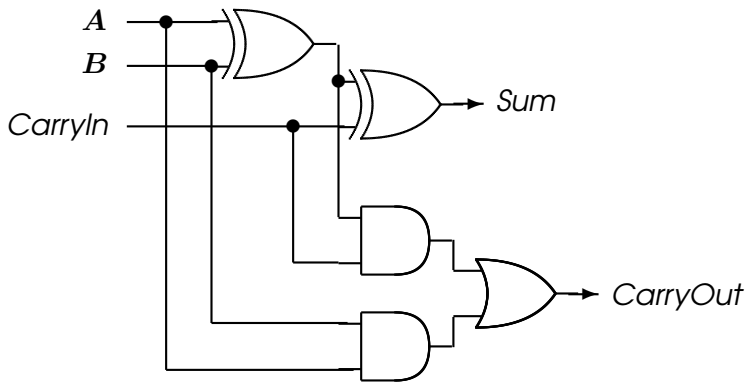
Boolean circuits for the 1-bit adder





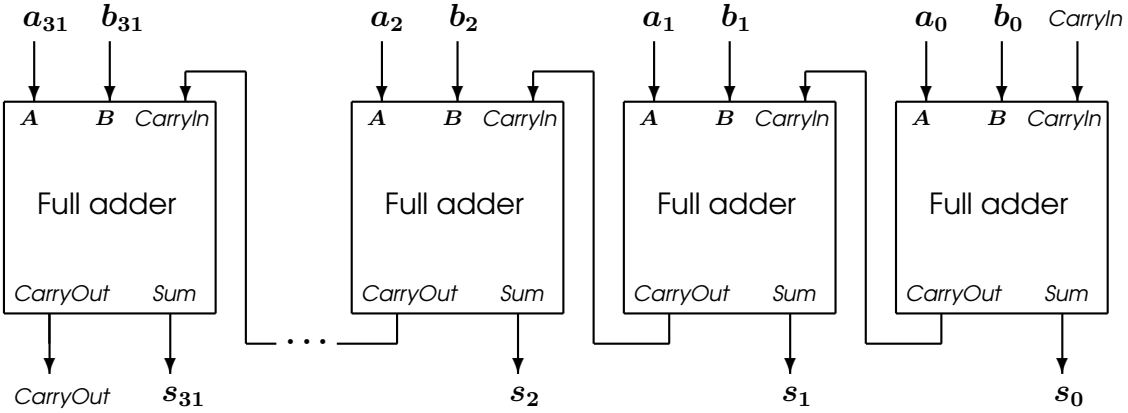
Boolean circuit for the 1-bit adder (cont.)

Check that the following circuit realises the same functions *CarryOut* and *Sum*



32-bit adder

Adding $A = a_{31}a_{30}\ldots a_2a_1a_0$ and $B = b_{31}b_{30}\ldots b_2b_1b_0$

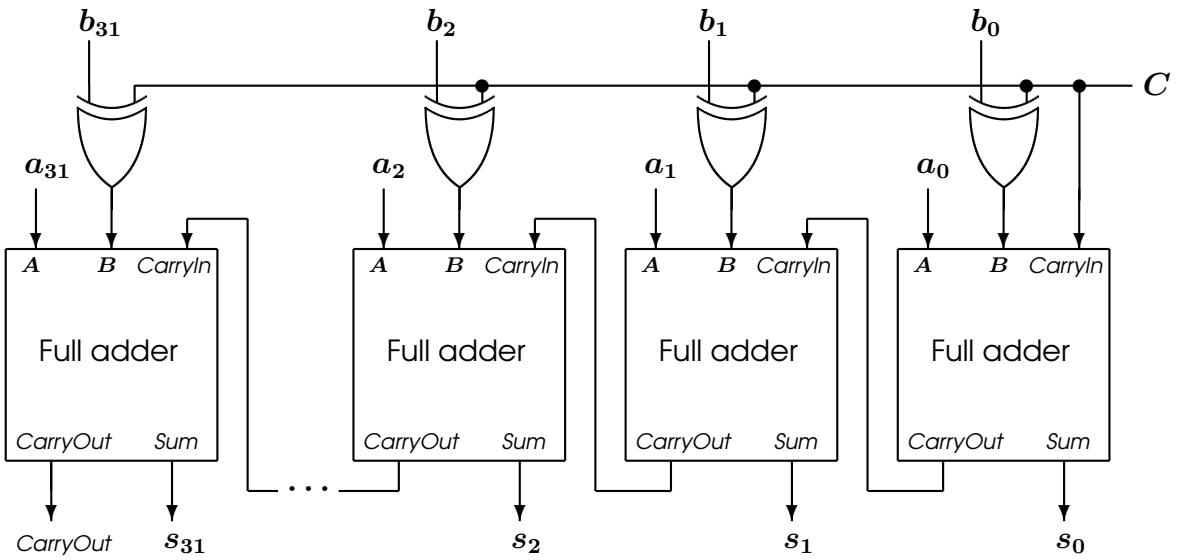


- *CarryIn* in the least significant adder is supposed to be $\mathbf{0}$
- What happens if we set this *CarryIn* to 1 instead of 0?

32-bit adder and subtractor

Control bit C:

- $C = 0$: Add $A + B$
- $C = 1$: Subtract $A - B$ (because $b_i \oplus 1 = \neg b_i$)



Scientific notation

The two's complement representation allowed us to deal with the integer numbers in the interval between -2^{31} and $+2^{31}-1$

- What if we need larger numbers (say, in astronomy)?
- How to deal with small fractions (say, in nuclear physics)?

'Scientific' notation: $976,000,000,000,000 = 9.76 \times 10^{14}$

$$0.0000000000000976 = 9.76 \times 10^{-14}$$

We dynamically slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point

Any given number can be written in the form $a \times 10^b$ in many ways; e.g.,

$$350 = 3.5 \times 10^2 = 35 \times 10^1 = 350 \times 10^0$$

In **normalised scientific notation**, exponent b is chosen such that $1 \leq |a| < 10$

Using binary numeral system instead of decimal, we can represent any real **non-zero** number in the form $a \times 2^E$ with $1 \leq |a| < 2$, or

$$(-1)^S \times (1+F) \times 2^E, \qquad 0 \leq F < 1$$

Binary fractions

$$16.625_{10} = 1 \times 10^1 + 6 \times 10^0 + 6 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}$$
$$= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$
$$= 10000.101_2$$

To convert a decimal number with both integer and fractional parts, convert each part separately and combine the answers (integer conversion is on page 3)

Example: convert 0.6875_{10} to binary (**multiply by 2**)

$$0.6875 \times 2 = 1.3750 \qquad \text{integer} = 1$$
$$0.3750 \times 2 = 0.7500 \qquad \text{integer} = 0$$
$$0.7500 \times 2 = 1.5000 \qquad \text{integer} = 1$$
$$0.5000 \times 2 = 1.0000 \qquad \text{integer} = 1$$

The result is read from the first integer downwards:

$$0.6875_{10} = 0.1011_2$$

IEEE 754 floating-point standard

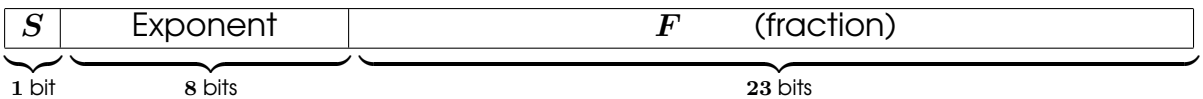
Every real number different from 0 can be represented in the form

$$(-1)^S \times (1 + F) \times 2^E, \qquad 0 \leq F < 1$$

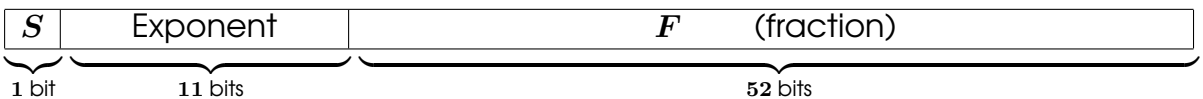
- S is the **sign** (0 for positive and 1 for negative)
- F is the **fraction** ($0 \leq F < 1$)
- E is the **exponent** (determining the actual location of the binary point)

The size of E and F may vary: a fixed word size means that one must take a bit from one to add a bit to the other. There existing compromises are:

Single precision floating-point format: 8 bits for E and 23 bits for F



Double precision floating-point format: 11 bits for E and 52 bits for F



IEEE 754 floating-point standard (cont.)

If we number the bits of the fraction F from left to right $b_1, b_2, \ldots, b_n$ (where $n = 23$ or $n = 52$), then the IEEE 754 standard gives us the numbers

$$(-1)^S \times (1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} + \cdots + b_n \times 2^{-n}) \times 2^E$$

- How to represent E? We need both positive and negative exponents.

Biased notation: fix $Bias = 127_{10} = 2^7 - 1 = 0111\ 1111_2$ for single precision

$Bias = 1023_{10} = 2^{10} - 1 = 011\ 1111\ 1111_2$ for double precision

$$E = Exponent - Bias$$

where $Exponent$ is the number **stored in the exponent field**

Examples:

- An exponent of -1 is represented by the bit pattern of the value

  $-1 + 127_{10} = 0111\ 1110_2$ (single precision)

- An exponent of $+1$ is represented by the bit pattern of the value

  $1 + 127_{10} = 1000\ 0000_2$ (single precision)

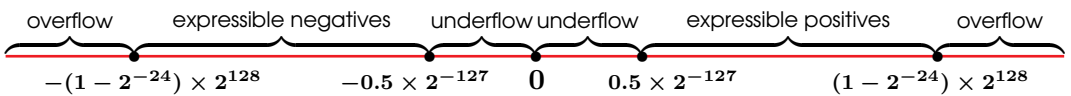
IEEE 754 floating-point standard (cont.)

The range of single precision numbers is then from as small as

$$\pm 1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{-126}$$

to as large as

$$\pm 1.1111\ 1111\ 1111\ 1111\ 1111\ 111_2 \times 2^{127}$$



Number line

- 0 is represented as both $0000\ldots0000$ $(+0)$ and $1000\ldots0000$ (-0)
- not all numbers in the intervals between $-(1-2^{-24}) \times 2^{128}$ and -0.5×2^{-127} and between 0.5×2^{-127} and $(1-2^{-24}) \times 2^{128}$ are represented (why?)

Example: floating point representation

Show the IEEE 754 binary representation of the number -0.75_{10} in single and double precision

$$-0.75_{10} = -(3/4)_{10} = -(3/2^2)_{10} = -11_2/2^2_{10} = -0.11_2 =$$
$$= -0.11_2 \times 2^0 = -1.1_2 \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \textit{Fraction}) \times 2^{\textit{Exponent}-127}$$

So, in our case $\textit{Fraction} = 0.1, \quad \textit{Exponent} = -1 + 127 = 126$

The single precision binary representation of -0.75_{10} is therefore

$$\underbrace{1}_{1 \text{ bit}}\ \underbrace{01111110}_{8 \text{ bits}}\ \underbrace{10000000000000000000000}_{23 \text{ bits}}$$

- What is the double precision representation?

Example: converting binary to decimal floating point

What decimal number is represented?

$$\underbrace{1}_{\text{1 bit}}\ \underbrace{10000001}_{\text{8 bits}}\ \underbrace{01000000000000000000000}_{\text{23 bits}}$$

The sign bit is 1, the exponent field contains $10000001_2 = 129_{10}$,
and the fraction field contains $0.01_2 = 1 \times 2^{-2} = \frac{1}{4} = 0.25$

$$(-1)^S \times (1 + \textit{Fraction}) \times 2^{\textit{Exponent}-\textit{Bias}} = (-1)^1 \times (1 + 0.25) \times 2^{129-127} =$$
$$= -1 \times 1.25 \times 2^2 = -1.25 \times 4 = -5$$

Accurate arithmetic

– All integers in the interval between -2^{-31} and $2^{31} - 1$ are represented **exactly**

– Floating-point numbers are normally **approximations** for a number they can't really represent (there are infinitely many numbers between, say, 0 and 1, but no more than 2^{53} can be represented exactly in double precision floating point. The best we can do is get the floating-point representation **close** to the actual number

Rounding in IEEE 754: always keeps two extra bits on the right during intermediate computations, called **guard** and **round**

Example: add $2.56_{10} \times 10^0$ to $2.34_{10} \times 10^2$ assuming we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

To add $2.56_{10} \times 10^0$ to $2.34_{10} \times 10^2$, we have to shift the smaller number to the right to align the exponents, so $2.56_{10} \times 10^0$ becomes $0.0256_{10} \times 10^2$. The guard digit holds 5 and the round digit holds 6. As $2.3400 + 0.0256 = 2.3656$, the sum is 2.3656×10^2. Since we have two digits to round, we obtain 2.37×10^2. Doing this withot guard and round would give 2.36×10^2. Does it matter? Well it does...

Finite precision can lead to disaster

The Patriot missile failure

http://www.ima.umn.edu/~arnold/disasters/disasters.html

American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.

Cause: 'software problem,' namely, inaccurate calculation of the time since boot due to computer arithmetic errors.

Problem specifics: Time in tenths of second as measured by the system's internal clock was multiplied by $1/10$ to get the time in seconds. Internal registers were 24 bits wide.

$1/10 = 0.0001\ 1001\ 1001\ 1001\ 1001\ 100$ (chopped to 24 bits)

Error $\approx 0.1100\ 1100 \times 2^{-23} \approx 9.5 \times 10^{-8}$

Error in 100 hour operation period: $\approx 9.5 \times 108 \times 100 \times 60 \times 60 \times 10 = 0.34s$

Distance traveled by Scud $= (0.34s) \times (1676m/s) \approx 570m$. This was far enough that the incoming Scud was outside the 'range gate' that the Patriot tracked.

Finite range can lead to disaster

Explosion of Ariane rocket 4/06/1996

http://www.ima.umn.edu/~arnold/disasters/disasters.html

Unmanned Ariane 5 rocket of the European Space Agency veered off its flight path, broke up, and exploded only $\mathbf{30}s$ after lift-off (altitude of $\mathbf{3700}m$)

The $500 million rocket (with cargo) was on its first voyage after a decade of development costing $7 billion

Cause: 'software error in the inertial reference system'

Problem specifics:

A $\mathbf{64}$ bit floating point number relating to the horizontal velocity of the rocket was being converted to a $\mathbf{16}$ bit signed integer.

An SRI[1] software exception arose during conversion because the 64-bit floating point number had a value greater than what could be represented by a $\mathbf{16}$-bit signed integer ($\mathbf{max} = \mathbf{32,767}$)

[1] SRI = Système de Référence Inertielle or Inertial Reference System