



Representing numbers on the computer.

Computer memory/processors consist of items that exist in one of two possible states (binary states).

These states are usually labeled 0 and 1.

Each item in memory stores one "bit" of information. (whether it is a 0 or a 1).

How can we combine these "bits" into something useful?

We can let the two states represent the digits 0 and 1 of a positional, base 2 system.

(similar to our base 10 system, but with only 2 digits, 0 and 1, rather than 10 digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

This leads to a simple way to represent integers - - just use base 2.

Base 10

Base 2

0

0

1

1

2

10

3

11

4

When you run out of symbols, combine using idea of 0 and positional value

100

5

101

6

110

7

111

8

1000

9

1001

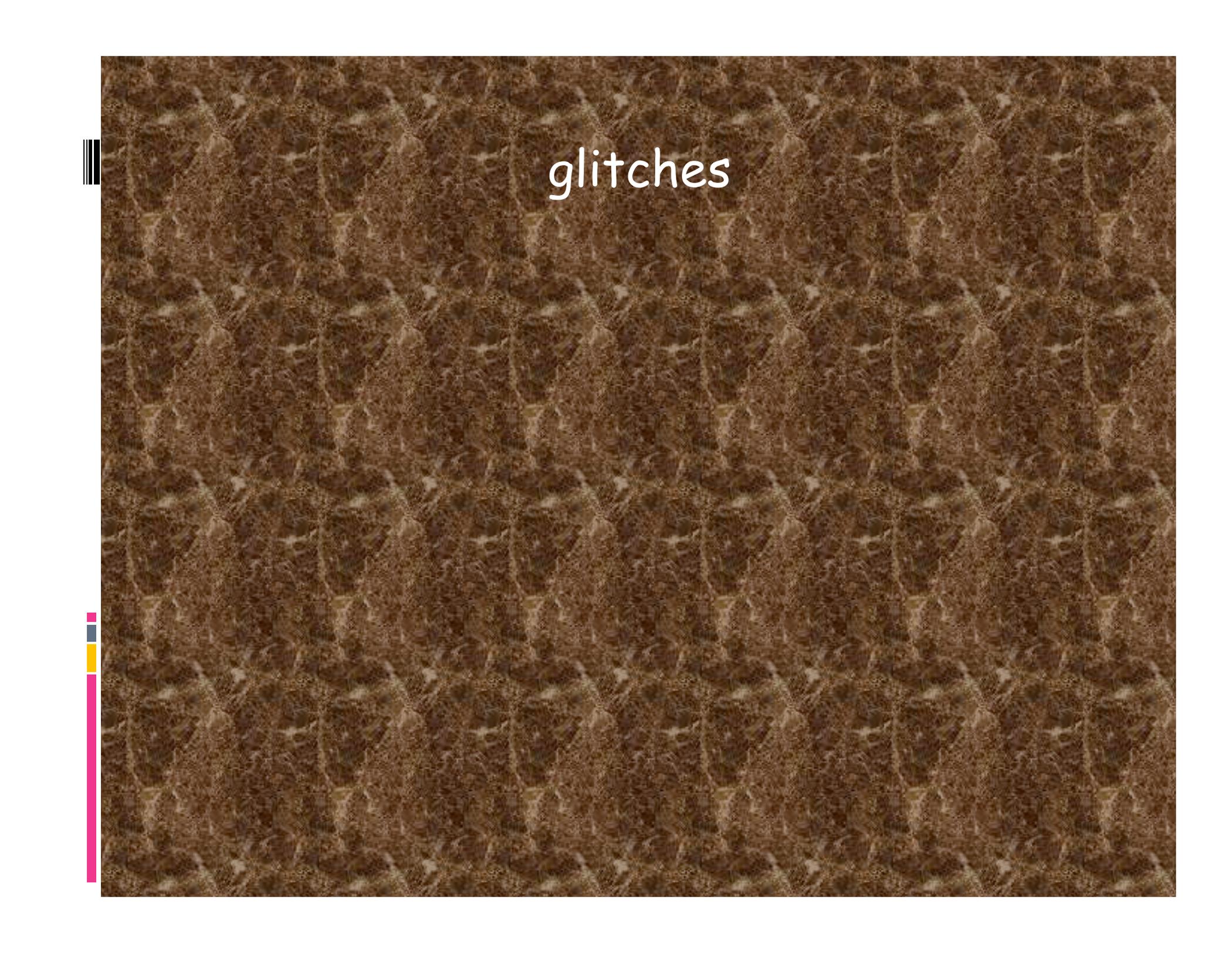
10

1010

11

1101

$a_n b^n a_{n-1} b^{n-1} \dots a_2 b^2 a_1 b^1 a_0 b^0$, "b" = base, "a" element of set of symbols for base $\{0, 1, \dots, b-1\}$



glitches

All the integers we can write down are finite
(use some finite number digits), but size
otherwise arbitrary
(8, 147, 987346036. etc).

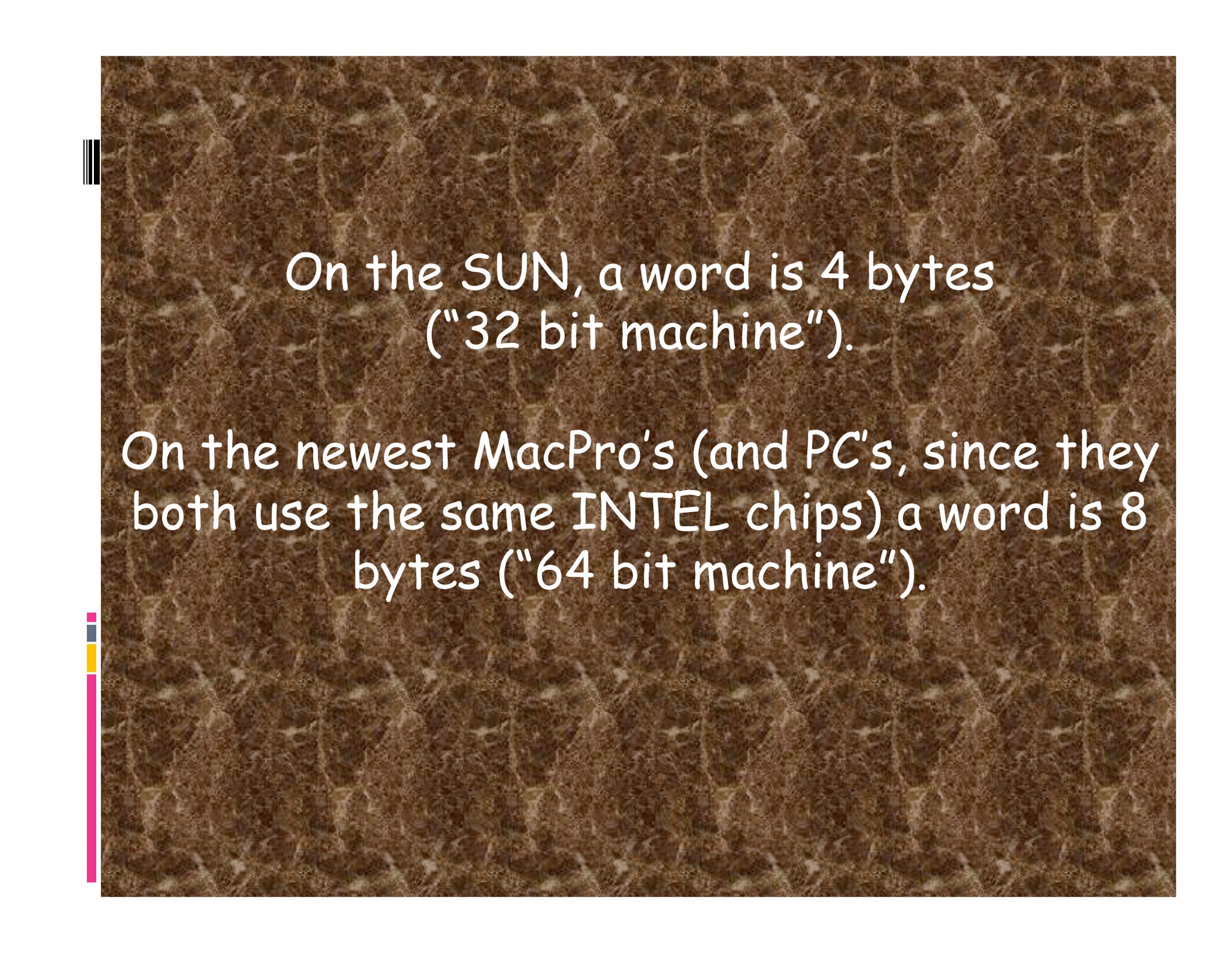
Computer takes this one step further by
stating up front the number of digits (bits)
in a number in memory. All numbers will have
this number of bits (or multiples of it).

Introduce the "byte" - a group of 8 bits.

In general the computer does not work with individual bits - it works with bytes (8 bits at a time) or words (some number of bytes).

Bytes can be combined into "words". Words were originally defined as 2 bytes (16 bits), but as computers got more powerful, words grew to 4 bytes (32 bits), and now 8 bytes (64 bits). (how things are combined will come back to bite us later)

Half a byte, 4 bits, is a "nibble".



On the SUN, a word is 4 bytes
("32 bit machine").

On the newest MacPro's (and PC's, since they
both use the same INTEL chips) a word is 8
bytes ("64 bit machine").



So let's say we are on a machine with 3 bit
"words".

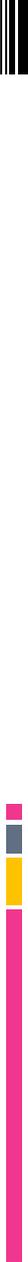
(so we can write out the numbers)

We have seen that we can represent positive
integers in base 2.



With 3 bits we can count to 7.
(with n bits we can count to $2^n - 1$)

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7



What about negative integers?

Now we have a problem.

We don't have a minus sign.

All we have are 0 and 1.

One solution is to use the most significant (MSB) or highest order bit (the leftmost one) as a "sign" bit.

000	0
001	1
010	2
011	3
100	-0
101	-1
110	-2
111	-3



This has some problems.

We have two values for zero (positive and negative, and we only have 6 distinct numbers, not 7).

This representation will also make binary arithmetic (add, subtract) on computers awkward (i.e. if numbers are expressed in this fashion).

Actual solution:

Use MSB to indicate the sign of the number
- but in a slightly funky manner.

Use the idea of the

-Additive inverse - each positive number has
an additive inverse.

combined with the fact that the computer
has

- Finite precision - uses a fixed number of
bits to represent a number.

How does this work? (using 4 digit numbers)

$$A=0101_2$$

Want additive inverse of A , a number that when added to A produces 0.

If we add 1_2 we get

$$A'=0101_2+0001_2=0110_2$$

So we have made the least significant bit (LSB) a zero.

If we add 11_2 we get

$$A' = 0101_2 + 0011_2 = 1000_2$$

Now the 3 LSBs are zero.

If we add 1011_2 we get

$$A' = 0101_2 + 1011_2 = 10000_2$$

But - our numbers are only 4 bits in size, the **1** is a carry into a 5th bit - but we don't have a 5th bit, it goes into the bit bucket.

$$A' = 0101_2 + 1011_2 = 10000_2$$

So with a limit of 4 bits we cannot add the two 4 bit positive numbers 0101_2 and 1011_2 because the answer needs 5 bits. Keeping only the 4 bits we have.

$$A' = 0101_2 + 1011_2 = 0000_2$$

we see that 1011_2 is the additive inverse of 0101_2 .

So we want 0101_2 to represent 5_{10} ,
and 1011_2 to represent -5_{10} ,
While maintaining positional notation.

$$0101_2 = 0 * \text{something} + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5_{10}$$

and

$$1011_2 = 1 * \text{something} + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = -5_{10}$$

$$1011_2 = 1 * \text{something} + 3_{10} = -5_{10}$$

$$\text{So something} = -8_{10} = -2^3$$

So now we have,

$$abcd_2 = a * (-2^3) + b * 2^2 + c * 2^1 + d * 2^0$$

$$abcd_2 = a * (-8_{10}) + b * 4_{10} + c * 2_{10} + d * 1_{10}$$

which also maintains positional notation.

We can now count from -2^{n-1} to $2^{n-1}-1$

(ex. for $n=3$, $2^3=8$, so we can represent 8 values $\{-4, -3, -2, -1, 0, 1, 2, 3\}$)



This representation of numbers is called
two's complement.

Numbers written this way are two's
complement numbers.



Two's complement numbers can be made using the "theory" presented, or by noticing that you can also form them by inverting all the bits and adding a 1!

$(5_{10}) = 0101_2 \rightarrow$ invert bits 1010_2
then add 1

$1010_2 + 0001_2 = 1011_2 = (-5_{10})$. (compare to before)

This method is trivial to do on a computer (which can really only do logical operations [and, or, exclusive or {negate can be made from exclusive or}] on pairs of bits.)



So - to add on a computer just add the two binary numbers.

To subtract on a computer, just add the two's compliment of the number you are subtracting.



Sizes of numbers (integers)

8 bit (byte) : -128 to +127
(unsigned: 0 to +255)

16 bit (half word, word, short, int) : -32,768
to +32,767 (32K)
(unsigned: 0 to +65,535 or 64K)

32 bit (longword, word, int) : -2,147,483,648
to +2,147,483,647
(unsigned: 0 to +4,294,967,295)

64 bit (double word, long word, quadword, int68) : -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
(unsigned: 0 to +18,446,744,073,709,551,615)

128 bit (octaword) :
-170,141,183,460,469,231,731,687,303,715,884,105,728 to +170,141,183,460,469,231,731,687,303,715,884,105,727
(unsigned: 0 to +340,282,366,920,938,463,463,374,607,431,768,211,455)

{ 10^{38} - a pretty big number - but not big enough to count the atoms in the universe - estimated to be 10^{80} .}

So now we can add and subtract integers
(also known as fixed point numbers)

(and multiplying by repetitive addition, division by repetitive subtraction).

What about -

Non-integer numbers?

Numbers outside the range of integers?

Enter - floating point numbers.

Non-integer numbers on the computer are limited to rational numbers

(a/b where a and b are integers).

Akin to scientific notation

$a.cde \dots * b^n$

where b is the base.

Floating point numbers can represent a wider range of numbers (bigger range of exponents) than fixed point numbers.

As with scientific notation - floating point numbers will have a number of digits in a decimal number (with a decimal point, not base 10) plus an exponent, which is used to multiply the decimal number by the base raised to that power.

$$2.235 \times 10^6$$

But now our number and base will be binary.

Modern floating point format is IEEE 754 standard (there were at least as many as computer manufacturers for a long time).

Non-integer numbers are represented as

$$\left(1 + \sum_{n=1}^{p-1} bit_n 2^{-n} \right) 2^m$$

$$1.5707964 * 2 = \\ (1 + 1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-4} + 1 * 2^{-7} + 1 * 2^{-23}) * 2^1$$


$$1.5707964 * 2 =$$
$$(1.+1*2^{-1}+0*2^{-2}+...+1*2^{-4}+...+1*2^{-7}+...+1*2^{-23})*2^0$$

The decimal number (part in yellow) is called the mantissa.

The exponent (part in cyan) is called the exponent.





Following the same restrictions that the computer placed on integers - we will have some predefined finite size for both the mantissa and exponent.

We will also need to know where the radix (decimal) point is located (we don't have a radix point - just 0 and 1).

And a way to represent negative values for both the number and the exponent.

Typical size for floating point number is 32 bits - called single precision (double precision is 64 bits, quad precision 128 bits).

The IEEE floating point number consists of a

- 24 bit mantissa (including hidden bit),
- an 8 bit exponent (with a bias or excess to handle positive and negative values),
- and a sign bit (total 32 bits).

The decimal point will always be after (in IEEE format, it could be before) the most significant non-zero digit (which can only be a 1 in base 2).

Implicit or hidden bit.

To milk another digit out of our floating point representation use the fact that for all numbers but zero, the first binary digit will be a 1, so we can throw it out.

(i.e. - not store it in the number. We have to remember to stick the implicit or hidden bit back in for calculations. This is done automatically by the hardware in the CPU.)

Rounding is done by adding 1 to the 24th bit if the 25th bit is a 1.

(we have 23 bits in the floating point number for the mantissa after taking the sign bit into account,

but one bit [the MSB] is implicit, so the last bit is really the 24th in the number)

To handle negative exponents we will just add 127 (in the IEEE standard, some other floating point formats use 128) to the value of the exponent. The exponent is an unsigned integer.

The base for the exponent is 2

(it was 16 on the IBM, which gave a much wider range of values for the exponent, but also much bigger round off errors because every change in the exponent shifted 4 rather than 1 bits).

So our floating point number is

1.

SEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

example

π 's first 32 bits = 11.001001 00001111 11011010 10100010₂

Round to 24 bits 1.1001001000011111011011₂ × 2¹

Take this (23 bits) less hidden bit (yellow), together with sign bit = 0 (is positive number),

exponent = 1 (11.0... = 1.10... × 2¹)₂,

excess 127₁₀ or bias 127₁₀ exponent =

127₁₀ + 1₁₀ = 128₁₀ = 10000000₂

π = 0 10000000 1.1001001000011111011011

Range of floating point numbers.

Single precision
(32 bit number, 24 in mantissa)

About 7 decimal digits.

How to approximate number of base 10
digits from number of base 2 digits

$$2^{10} = 1024 \sim 10^3$$

base 2 exponent / base 10 exponent =
 $10/3 = 3.3$

$$(24/3.3 = 7.2)$$

Ranges for various sizes of floating point numbers

Type	Sign	Exponent	Mantissa	Total bits	Exponent bias	Bits precision
Single	1	8 (± 38)	23	32	127	24_2 (7_{10})
Double	1	11 (± 308)	52	64	1023	53_2 (16_{10})
Quad	1	15 ($\pm 4,965$)	112	128	16383	113_2 (34_{10})

To add floating point numbers - have to line up the mantissas (shift based on exponent).

Potential problem when adding two numbers of very different magnitude.

$$\text{eg. } 1.0 + 0.00000001 = 1.0$$

in single precision

(does not give expected 1.00000001)

because we do not have enough bits to represent correct value (only 7 decimal digits).

(solution here is to go to double precision.)

Another potential problem

Loss of significance when subtracting two numbers that are almost the same.

$$1.234567 - 1.234566 - 0.000001$$

Start out with 7 digit numbers, end up with single significant digit in new - seemingly 7 significant digit - number (1.000000×10^{-6}).

This is not solved by increasing precision on computer.



To multiply floating point numbers - add exponents, multiply mantissas.



On computer - result has same number significant digits (7 for single precision) as the two factors.

Special values:

- Zero (no 1 bit anywhere to normalize on - all zeros)

+/- infinity

- NaN (result of operations such as divide by zero, sqrt -1 [except in matlab])

-Others

Machine precision

Characterizes accuracy of machine representation.

epsilon or ϵ_{mach}

Value depends on number bits in mantissa and how rounding is done.

With rounding to zero,

$$E_{\text{mach}} = B^{1-P}$$

With rounding to nearest,

$$E_{\text{mach}} = (1/2) * B^{1-P}$$

Where $B^M = B^M$.

$$E_{\text{mach}}$$

Quantifies bounds on the *relative error* in representing any non-zero real number x within the normalized range of a floating point system:

$$| (f(x) - x) / x | \leq E_{\text{mach}}$$

Math vs what the computer does.

Due to finite precision and rounding the computer will (generally) not give what you might expect mathematically.

Mathematically $\sin^2\theta + \cos^2\theta = 1$.

But on the computer (finite precision, rational values only, ...) the test

$\sin^2\theta + \cos^2\theta == 1$

will return FALSE!

One solution to this problem is to test against a small number - the machine precision, rather than zero.

So test

$$\text{abs}(\sin^2\theta + \cos^2\theta - 1) < \text{epsilon}$$

if this is true consider, then we can consider
 $\sin^2\theta + \cos^2\theta = 1$.

(same with test $a == b$, use $\text{abs}(a - b) < \text{epsilon}$).

One last detail

Combining bytes into words.

Many ways to do it, and all were used (of course).

Two of the most popular are both still around.

Can cause value of numbers to be interpreted incorrectly.

Can cause major headaches

(some operating systems/programs can figure it out and fix it for you, others can't and you have to do it).

Endianness

In byte (unsigned integer)

$2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0$

MSB on left, LSB on right.

What happens when I combine 2 bytes into a 16 bit number?

Two possible ways to combine.
(and also several possible ways to visualize
memory).

	MSByte	LSByte
Address	0	1

	LSByte	MSByte
Address	0	1

A number made up of just one byte would have that byte placed at address 0.



A diagram showing a memory layout for a single byte. The top row contains addresses from 0 to F. The bottom row contains the corresponding hexadecimal values. A green arrow points to the value '21' at address 0.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	21	00	00	00	00	00	00	00	00	00	00	00	00	00	00

How do we expand this number to two bytes?
We have 2 options. We could allow it to grow towards the right - the little endian form).



A diagram showing a memory layout for two bytes in little-endian format. The top row contains addresses from 0 to F. The bottom row contains the corresponding hexadecimal values. A green arrow points to the value '21' at address 0.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	21	43	00	00	00	00	00	00	00	00	00	00	00	00	00

This puts the numbers "backwards", but allows us to extend the size of number to the limits of memory without having to move the least significant parts.

Alternately, we could slide the first byte to the right, changing its address, and then extend the number toward the left, the big endian form.


0 1 2 3 4 5 6 7 8 9 A B C D E F
0000 43 21 00 00 00 00 00 00 00 00 00 00 00 00 00

This keeps the digits in the "correct" order, but forces a definite size into the number (one has to move the bytes with lower significance as add more bytes).
(the arrow indicates the base address when referring to the number).

Nothing really forces us to number bytes left to right. If we wanted, we could number right to left. If we were to do so, the above exercise takes on a whole new look:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	21

grows to become either (Little Endian):

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	43	21

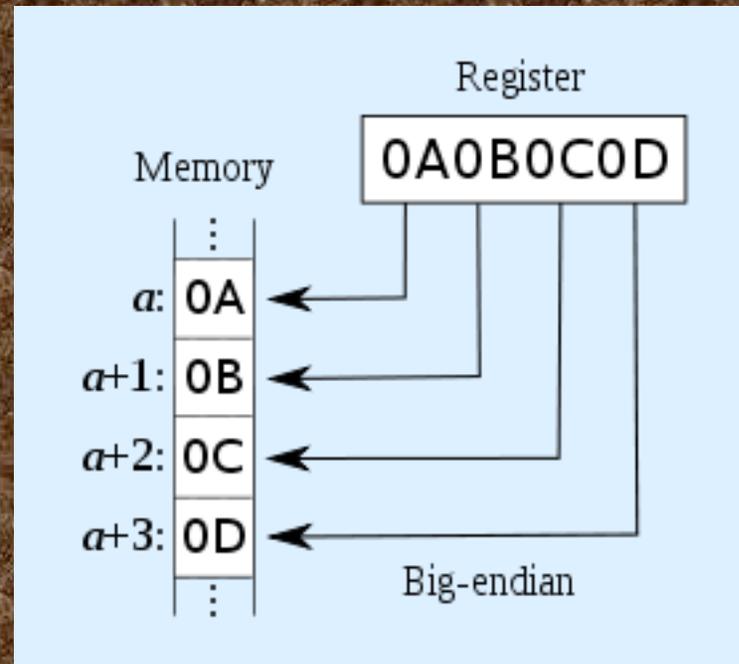
or (Big Endian)

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	21	43

Suddenly, little endian not only looks correct, but also behaves correctly, grows left without affecting existing bytes. And, just as suddenly, big endian turns onto a bizarre rogue whose byte ordering doesn't follow the "rules".

Big-endian

(Looking at memory as column going down.)



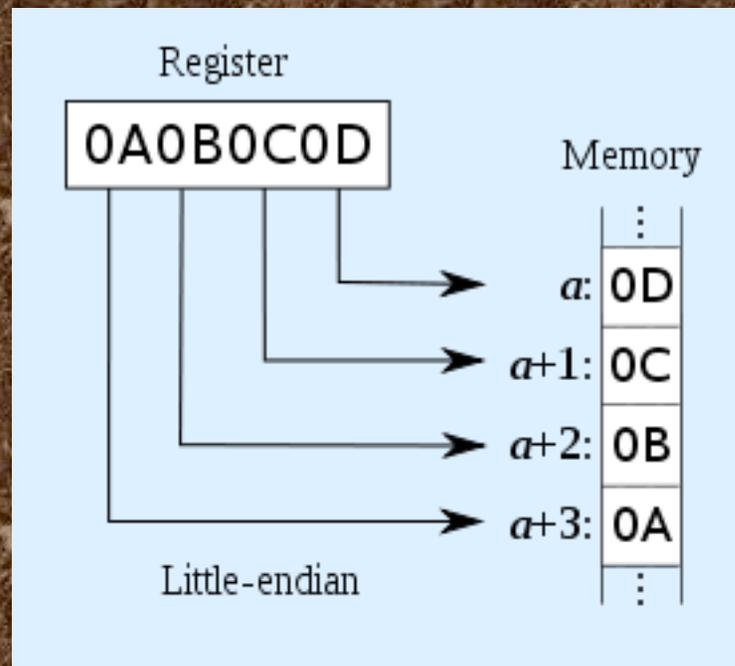
Big-endian - with 8-bit atomic element size
and 1-byte (octet) address increment:

increasing addresses →
... 0x0A 0x0B 0x0C 0x0D ...

The most significant byte (MSB) value, which is 0x0A in our example, is stored at the memory location with the lowest address, the next byte value in significance, 0x0B, is stored at the following memory location and so on. This is akin to Left-to-Right reading in hexadecimal order.

Little-endian

(Looking at memory as column going down.)



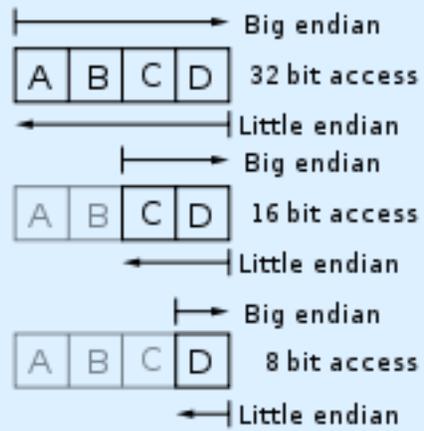
(As with left2right or right2left ordering of row form, reasonableness of behaviors would "switch" if looked at memory as column going up.)

Little-endian - with 8-bit atomic element size
and 1-byte (octet) address increment:

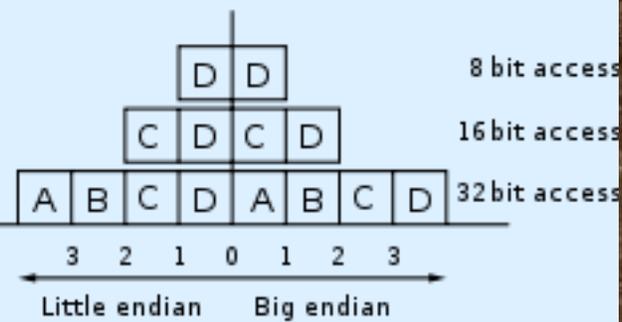
increasing addresses →
... 0x0D 0x0C 0x0B 0x0A ...

The least significant byte (LSB) value, 0x0D,
is at the lowest address. The other bytes
follow in increasing order of significance.

Register



Memory





Which way makes "more sense" depends on
how you picture memory.

As rows or columns.

Whether the rows go from left2right or
right2left, or the columns go up or down.



Machines that use little-endian format include x86, 6502, Z80, VAX, and, largely, PDP-11

Machines that use big-endian format include Motorola (pre intel macs), IBM, SUN (SPARC)

(machines/companies that started out with 8 bits typically used little-endian when they combined bytes. Machines/companies that started out with 16 bits typically used big-endian to break words into bytes.)

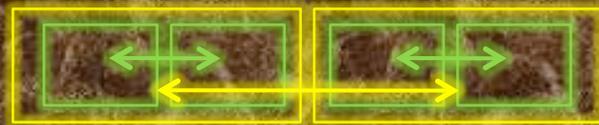
What you need to know.

For binary data (not ascii [basically letters] which is stored in a single byte) you have to know how it is stored. If it is stored the wrong way for your machine, you have to do a "byte swap" to fix it.

There are programs to do this.

(plus some programs, like the latest version of SAC, can figure it out - so you don't have to worry about it)..

When you byte swap, you also have to swap each grouping of 2^n (e.g. for 32 bit numbers you have to swap words also).



Etc. for 64 bit, 128 bit, values.

When converting floating point (assuming base 2 exponent) have to worry about

- the exponent's excess value (IEEE uses 127, some other formats use 128 - a factor of 2)
- and position of assumed decimal point (before or after most significant bit with value of 1 (another factor of 2)).

Only have to worry about this stuff when moving (usually old) binary stuff between machines/architectures.