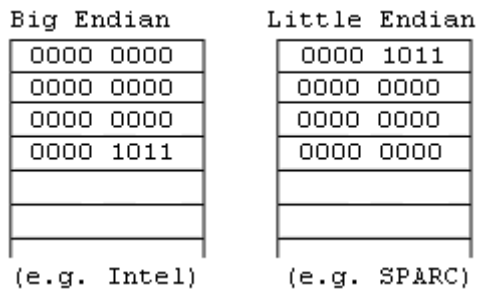
Data Representation

This is how `.byte 11` (in MIPS) is stores in memory:

0000 0000 0000 0000 0000 0000 0000 1011



We need a way to map: **data → binary**

Data Types → Number → Integer → Signed/Unsigned
→ Real
→ Char
→ Other (Picture, etc.)

Integers

A decimal example: $2734 = 2 * 10^3 + 7 * 10^2 + 3 * 10^1 + 4 * 10^0$

A binary example: $[01011]_2 = 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = [11]_{10}$

Unsigned

Unsigned integers are positive (or to be more precise, do not have a sign).

Size	Limit (low, high)	
8 bits	0, 255	($255 = 2^8 - 1$)
16 bits	0, 64k	($64k = 2^{16} - 1$)
…	…	

n bits $0, (2^n - 1)$

For the case of n = 4 we use hexadecimal shorthand.

So for example:

0000 0000 0000 0000 0000 0000 0000 1011

becomes

0x0000000B

0000	0
0001	1
0010	2
0011	3
..	
1010	A
..	
1111	F

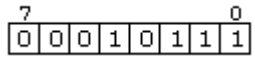
Signed

Size	Limit (low, high)
8 bits	-128, 127
16 bits	$-2^{15}, (2^{15} - 1)$
…	…
n bits	$-2^{n-1}, (2^{n-1} - 1)$

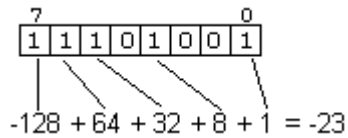
Positive numbers are like unsigned. But how do we store negative numbers?

Let's represent -23:

- Start by representing the absolute value which is 23 in this case.
  (We use an 8-bit number instead of a 32-bit integer to get rid of all the leading zeros.)
-



-
- We *could* just leave the leftmost bit for sign (0 for positive, 1 for negative) but then we would have problems: For instance, there would be two zeros (0 and -0) which can be a big problem. E.g. could create infinite loops.
- So we set the leftmost bit to 1 (-2^7 = -128) *AND* add as many 1's so that the sum of their decimal value and -128 equals the negative number we are representing:



- We can get the same result with a better algorithm:
  - Note that -1 is represented by 1111 1111

- We show the complement of a binary number by flipping all digits (replacing 0's and 1's).

$$\overline{23} \quad \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}$$

- See that $x + \overline{x} = -1 \longrightarrow -x = 1 + \overline{x}$

In our example

$$\begin{array}{rl} \overline{23} & \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0} \\ + \quad 1 & \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1} \\ \hline -23 & \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1} \end{array}$$

- This method is called "The 2's Complement"

MIPS has different instructions for signed and unsigned integers (e.g. `add` vs. `addu`). It was previously mentioned that signed and unsigned integers have the same binary representation and their binary addition will output the same binary result regardless of the sign.

Then what are the instructions with the u suffix?

In case of `addu` the answer is that it will ignore an overflow.
In case of an exception → `add` → crash
→ `addu` → ignore
However in case of `mult`, `div`, `lb` and `slt` the u suffix will use a *different algorithm* to do the operation.
For example see how `x` in this example is stored in register `t0`:

```
        .data
x:      .byte -1     # This is represented as 1111 1111 in memory
        .text
        lb $t0,x($0)
```

Result with lbu

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Result with lb

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

`lbu` adds zero to the left. `lb` adds the sign extension to the left.

→ So in case of `add` and `sub` it is not a big deal to forget the u, but be careful with the others.

Real Numbers

There are many standards to represent real numbers. We use the IEEE 754 standard.

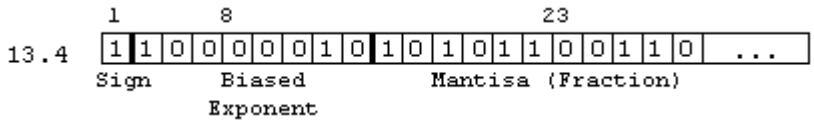
Let's say we want to represent -13.4:

- Keep the sign
- Represent 13 in binary (which is 1101)
- Keep the decimal point (for now)
- Multiply the decimal by 2, then multiply the decimal part of the result by 2 and so on until you reach the same pattern. Keep track of the digits (0's and 1's) before the decimal point and write them as a binary number:

  0.4 * 2 = **0**.8
  0.8 * 2 = **1**.6
  0.6 * 2 = **1**.2
  0.2 * 2 = **0**.4
  0.4 * 2 = 0.8 ← Same as the first line (and it repeats)

  → 0110 0110 0110 ….
- The number is now: -1101.011001100110…
- *Normalize* by moving the decimal point to left or right until there is only a '1' before it. Then multiply by 2 to the power of n with n being the number of digits we have shifted. (Positive if the decimal point is shifted to the left, i.e. numbers shifted to the right.)

  $-1.101011001100110\ldots * 2^3$
- And finally it is stored in memory:



The first bit is the sign, the next 8-bit portion is the Biased Exponent (the exponent of 2 plus 127 to ensure we get a positive number) and the last 23-bit portion is Mantisa (The binary representation of the fraction – excluding the 1 before the decimal point) as mentioned in the previous step.

<u>*Note*</u> that there are 2^{24} digits in the fraction part (including the 1 that is not stored in memory) which represents 6 digits. This is the number of significant figures in a floating point number. Most CPUs have 6-7 significant figures.

E.g. in Java floating point numbers have 6 significant figures:

```
// This java code:
float f = 17534.283;
System.out.print(f);
// will output 17534.2
```

Characters

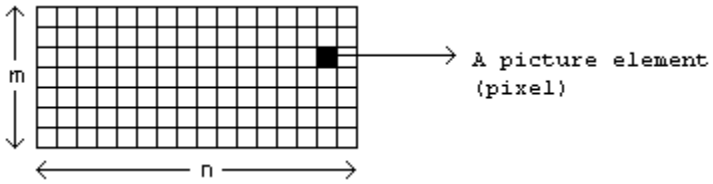
We use a mapping system to convert between ASCII characters and their value that is stored in memory as an 8 bit unsigned integer.

Other Data

Other types of data can include pictures, video, sound etc.

This is how **pictures** are stored in memory:

- Generally every picture is made of picture elements (or pixels). Pixels are the smallest part of a picture and only contain color information.



- A picture has m rows and n columns of pixels.
- These are some of the common types of pictures:

Size of each pixel	Number of Colors	Total Size (for a m x n picture)
1 bit	2 (Black & White)	1 x m x n bits
2 bit	4 (Gray Shades)	2 x m x n bits
8 bit	256	8 x m x n bits
24 bit	16 million	24 x m x n bits