Architecture et Système

Stefan Schwoon

Cours L3, 2014/15, ENS Cachan

Data representation

Some basic notions:

bit: most basic piece of digital information, stores either 0 o 1.

byte: smallest addressable unit in the computer's memory.
Today usually 8 bit.

word: any vectors of bits manipulated or transferred as a unit.
E.g. *register word*, *instruction word*, etc.
Without qualifier, usually a register word is meant.

Integers

Stored in a word of fixed size (usually of word length 2^{3n}, for some $n \geq 1$)

Data types in C: `char`, `short int`, `int`, `long int`, `long long int`

Sizes not fixed by C standard, only *minimal* sizes: 8, 16, 16, 32, 32, respectively, where `int` meant as register word.

Concrete data size can be found using `sizeof(char)` etc.

Big vs little endian

A word with length > 8 takes up multiple bytes in memory.

Big-endian: most significant byte first
e.g. a value $(12345678)_{16}$ is stored in four bytes, in the order 12, 34, 56, 78
(all hexadecimal)

Little-endian: least significant byte first
in the above example, the order is 78, 56, 34, 12

Both formats are common (e.g., little-endian on Intel-compatible processors)

Mostly historic reasons; choices today mostly for backward compatibility

> pro little-endian: same word can be interpreted modulo 2^8, 2^{16}, etc using same address; some arithmetic operation, e.g. with serial carry propagation became easier
>
> pro big-endian: more “natural”, division modulo 2^8 etc using same address

Beware!

Endian-ness becomes important when exchanging data (files, networks).

In these cases, the byte order must be specified in any protocol/file format.

The Internet protocol defines a byte order for IP protocols (incidentally, big endian).

C functions: `ntohl`, `ntohs`, `htonl`, `htons`

Signed and unsigned integers

A word with n bytes can be used to store values $0 \ldots 2^n - 1$.
This is called an unsigned integer.
Arithmetic operations implicitly carried out modulo 2^n.

Signed integers are usually stored as two's complement.
Here, a word stores values $-2^{n-1} \ldots 2^{n-1} - 1$.

For non-negative values, the most-significant bit (MSB) is 0.

For negative values, the MSB is 1: representation of $-i$ (for $i \geq 0$) obtained by subtracting 1, then inverting all bits.
E.g., -1 represented as $11 \cdots 1$, -2^{n-1} represented as $10 \cdots 0$.

Alternatively: signed representation of i equals unsigned representation of $2^n - i$.

Thus, adding i and $-i$ using (unsigned) addition gives 0, the right result.

Hence: at the binary level, all arithmetic operations can be carried out as for unsigned values.

Signed/unsigned merely a matter of interpretation of the results!

Floating-point numbers

Real numbers are approximated by so-called floating-point numbers.

Also stored in words of fixed size.

Hence: precision limited, only finitely many numbers can be represented.

General idea: floating-point number is a tuple (s, m, e) meaning $\pm 2^e \cdot m$.

s is the sign (one bit, usually `0` non-negative, `1` negative);

m the mantissa;

e the exponent.

Need for standardization

Problems:

size of exponent, mantissa?

not unique: $(s, m, e) \equiv (s, 2m, e - 1)$

How to treat special cases like division by zero, overflow, ...

The most important standard for dealing with these matters is IEEE 754. Here, we just discuss its 32-bit part (`float` in C, `double` = 64-bit).

IEEE 754 (32-bit variant)

IEEE 754 fixes the following conventions:

1 bit for sign, 8 for exponent, 23 for mantissa;

Interpretation of exponent: $e_u - 127$, where e_u is the unsigned interpretation of the 8 bits. Thus, the range is ± 127. Value 128 is used for special purposes.

Interpretation of Mantissa: $1 + (m_u/2^{23})$, hence range $[1, 2)$.

Remarks:

Interpretation of mantissa solves the uniqueness problem

$e = 128$ used to represent $\pm\infty$ (with $m_u = 0$)
and NaN (not a number, with $m_u \neq 0$)

Remarks

Floating-point arithmetic has certain problems that arise from the limitation to a fixed number of bits:

- limited precision
- certain laws like distribution etc do not hold
- rounding errors, etc

On modern machines, floating-point operations are not implemented in software but carried out by an FPU (floating-point unit)

Modern FPUs are specially equipped to handle complex operations like logarithms, exponenation, roots, trigonometric functions, ...

Strings

A string is chain of characters.

Suppose for the moment that each character can be stored in one byte.

Then the main problem remains to identify the length of the string.

The prevailing method is to use `0`-terminated strings, i.e. the actual string content is followed by a 0-valued byte.

In C, a string constant `"abc"` actually creates an object that is four bytes long.

Attention when allocating memory for strings, always leave room for terminating zero! `strlen` returns length without terminating byte!

Alternatives: prefix string with a byte indicating the length.

Used, e.g., in Pascal.

Advantage: can find end-of-string more quickly for, e.g., extending the string. Also, strings may contain 0-bytes.

Disadvantage: Length of string limited to 255 bytes.

Certain 80x86 assembly instructions support 0-terminating strings (`LOOPNE`, `SCASB`, etc)

Character encoding

Initially: one byte per character; encoding maps values to characters

International standardization: ISO 646 in 1963, left some positions open for national variants

ASCII: US-variant of ISO 646, becomes de facto standard.

ASCII fixes only positions 0..127. Diverse extensions, notably Latin-1 (ISO 8859-1) exist for positions 128..255.

Unicode/ISO 10646: current version provides for more than 1 million *codepoints*

ASCII / ISO 646

7-bit code, positions 0..127. Examples:

positions 0..31 used for “non-printable” characters (newline, tabstop)

positions 48..57 (hex: 30..39) used for digits 0..9

positions 65..90 (hex: 41..5a) used for capitals A..Z

positions 97..122 (hex: 61..7a) used for lowercase a..z

Attention: code 10 is called *linefeed* or *newline*. Under Unix, it is generally interpreted to end a line and go to the beginning of the next.

Under Windows/DOS, a sequence of codes 13 (carriage return) and 10 is used for this purpose!

ASCII extensions

What to do with the remaining values 128..255 in a single byte?

DOS: used for drawing boxes, plus some accented characters

Latin-1: most used “European” characters (accented letters)

many other extensions exist

Unicode

Developed by *Unicode consortium* (several companies et al) in coordination with ISO.

Compatible with ASCII and Latin-1 on positions 0..255.

Defines many more positions (more than 1 million) for many languages and scripts.

Problem: more than one byte necessary to encode a character.

$\rightarrow$ length of string no longer equals number of bytes

$\rightarrow$ multi-byte coding would be wasteful: variable-length encoding used instead

UTF-8 encodes a character with 1..4 bytes