Floating-point arithmetic and error analysis (AFAE)

Tutorial n° 1 - Introduction to computer arithmetic

Exercise 1 (Radix conversions)**.** In a classic position system in radix β, a number x is represented by digits $x_{n-1}\, x_{n-2} \,\ldots\, x_0$ with $x_i \in \{0, \ldots, \beta - 1\}$, as follow:

$$x = \sum_{i=0}^{n-1} x_i \beta^i.$$

It is often necessary to convert such a representation of a number in radix β into another radix β' by example for the decimal input-output of a computer that works in binary.

1. As an input to a computer system, converting decimal numbers to binary is relatively simple.

   Assume that in a C program, a number x is represented in radix $\beta = 10$ with each digit x_i being an element of an array `uint32_t x[9]`.

   Describe an algorithm to convert this number into binary ($\beta' = 2$). Show that the result is always representable with a 32 bits unsigned integer `uint32_t`. Write the algorithm in C as a function `uint32_t convertFromDecimal(uint32_t x[9])`.

2. The conversion from binary to decimal is slightly more complicated. It is useful for decimal display.

   Assume that x is stored in radix $\beta = 2$ as a 32 bits unsigned integer `uint32_t`.

   Provide an algorithm to convert this number into decimal (radix $\beta' = 10$). Show that a binary 32 bits integer will never have more than 10 decimal digits. Write an algorithm in C as a function `void convertToDecimal(uint32_t res[10], uint32_t x)`.

   Why is this binary-to-decimal conversion generally more expensive than the reverse conversion?

3. It is not always necessary to represent numbers, that is to say quantities, in a single base or with units whose ratio is always the same (as in 1 m = 10 dm = 100 cm = 1000 mm). It is possible to use several radix, respectively units with different ratios.

   US are specialists in this way of representing quantities. For example, for the volume of liquids, the following units are used:

   — 1 US gallon corresponds to 4 US quarts;

   — 1 US quart corresponds to 2 US pints;

   — 1 US pint corresponds to 16 US fluid ounces .

   — 1 US fluid ounce corresponds to $3785411 \mu\ell = 3785411 \cdot 10^{-6}$ liters.

   Propose an algorithm for converting a metric representation in micro-liters ($\mu\ell$) into a US representation. What do you notice?

4. Explain how to write a function that converts a number in a non-redundant radix into binary. We recall that a redundant radix is a radix where all the digits c_i can have some values greater (in magnitude) than the ratio of the weight of the digits. For example, in a classic redundant radix, the digits c_i can be $c_i \in \{-\alpha, -\alpha + 1, \ldots, 0, \ldots, \alpha - 1, \alpha\}$ with $\alpha > \beta$ where β the ratio of the weight of the digits.

Exercise 2 (Additions)**.** Addition is one of the most basic operations. As an operation, the addition uses numbers written in a appropriate format.

We assume in the sequel that the numbers x and y we wish to add are represented in radix $\beta = 2^{32}$ as 2 arrays `uint32_t x[N]` and `uint32_t y[N]`. We recall that

$$x = \sum_{i=0}^{n-1} 2^{32i}\, x_i \qquad\qquad y = \sum_{i=0}^{n-1} 2^{32i}\, y_i$$

where x_i (resp. y_i) are elements `x[i]` (resp. `y[i]`) of the array `uint32_t x[N]` (resp. `uint32_t y[N]`).

1. Show that for any non-redundant radix β used to represent the numbers, the carry is less than 1 for the addition of 2 numbers. It means that the carry is always either 0 or 1. The proof must be by induction.
2. Let a and b be 2 unsigned integers on 32 bits `uint32_t a, b`. Write a C function `void fulladder(uint32_t *s, uint32_t *c_out, uint32_t a, uint32_t b, uint32_t c_in)` that computes the sum s and the carry c_{out} of the 2 numbers a, b in radix 2^{32} and the carry c_{in}. The numbers s and c_{out} computed by the function will satisfy

   $$2^{32}\, c_{out} + s = a + b + c_{in}$$

   where $0 \leq s < 2^{32}$, $c_{out} \in \{0, 1\}$.
   We recall that the unsigned integer arithmetic with 32 bits is done in C modulo 2^{32}. We assume that we do not have other unsigned integer types (like `uint64_t`).
3. Write a C function `void addition(uint32_t s[], uint32_t *c, uint32_t x[], uint32_t y[], int n)` that performs an addition of the numbers x and y, represented on n unsigned 32 bits integers. The function returns the array `s` the sum of x and y modulo 2^{32n} and in `c` the final carry. What is the complexity of an addition?

Exercise 3 (Multiplication)**.** When multiplying two numbers x and y, written with non-redundant radix β, with a result r always with this radix,

$$\sum_{l=0}^{2n-1} r_l\, \beta^l = \sum_{i=0}^{n} \sum_{k=0}^{n} \beta^{i+k}\, x_i \cdot y_k,$$

we need to solve two problems:
- the computation of all the *partial products* $x_i\, y_k$ corresponding to r_{i+k} and,
- the dealing of the carry when adding these partial products.

We will give solutions to those two problems in the sequel.

1. Let a, b, c be three digits in a non-redundant radix β, that is to say $0 \leq a, b, c < \beta$. Show that the partial product $a \cdot b$ increased with a carry c can always be represented as two consecutive digits $0 \leq u, v < \beta$ with this radix. To do so, show that for any choice of $0 \leq a, b, c < \beta$, we have $0 \leq u, v < \beta$ such that

   $$\beta \cdot u + v = a \cdot b + c.$$

2. Write a C function `void accuPartialProduct(uint32_t *u, uint32_t *v, uint32_t a, uint32_t b, uint32_t c)` that computes u and v from a, b and c. We always assume that we do not have access to unsigned integer type greater than `uint32_t`. You will need some shifts to implement this operation.
3. Write a function `void shortProduct(uint32_t r[], uint32_t a[], uint32_t b, int n)` that computes the operation

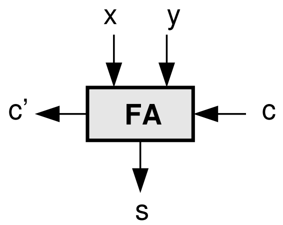
   $$\sum_{i=0}^{n} 2^{32i} r_i = b \cdot \sum_{i=0}^{n-1} 2^{32i}\, a_i.$$

4. Explain how to write a C function that computes a multiplication (naive) in $\mathcal{O}(n^2)$. This function `void multiplication( uint32_t r[], uint32_t x[], uint32_t y[], int n)` will satisfy

$$\sum_{i=0}^{2n-1} 2^{32i} \cdot r_i = \left(\sum_{i=0}^{n-1} 2^{32i}\, x_i\right) \cdot \left(\sum_{i=0}^{n-1} 2^{32i}\, y_i\right).$$

   We assume that all the arrays have been allocated with the right size and initialized to zero.

Exercise 4 (Normalization of redundant results)**.** The units of division by induction on digits often provide results in a redundant radix, that is to say, digits c_i of the quotient belong to a set $\{-\alpha, -\alpha+1, \ldots, 0, \ldots, \alpha-1, \alpha\}$ with $\alpha > \beta$, $\beta = 2$. Each of these digits are represented by a small binary number, often with an explicit bit for the sign.
In this exercise, we will see how this representation can be converted into a classic binary radix with a particular circuit.
The building block we will use is the *full-adder*:



We recall that the outputs s and c' of this operator satisfy for all $x, y, c \in \{0,1\}$

$$2\,c' + s = x + y + c.$$

1. Draw the *ripple carry adder*.
2. Propose a circuit for normalization of a number with n redundant digits in the radix $\{-3, \ldots, 0, \ldots, 3\}$.

Exercise 5 (Floating-point arithmetic)**.** The IEEE-754 double precision (known as binary64 since 2008) is the most used floating-point format in computer science. The IEEE-754 standard enforces a precision representation in memory for numbers in finite precision format.
We recall that a IEEE-754 precision number is composed of a bit for sign, 11 bits for exponent (stored with a bias $2^{11-1} - 1$ in order for it to be nonnegative) and 52 bits for the mantissa. For normalized numbers (with positive biased exponents), the first bit of the mantissa – always 1 – is not stored. For subnormal numbers, all the bits are stored.

1. Decode by hands the following double precision numbers:
   1. `0x3ff0000000000000`
   2. `0x0000000000000000`
   3. `0x8000000000000000`
   4. `0x0000000000000001`
   5. `0xc00a000000000000`
   6. `0x3ff6a09e667f3bcd`
   7. `0x7ff0000000000000`
2. On most of systems (at least for Intel/AMD, Intel/HP Itanium, IBM and ARM), floating-point number are stored in memory as if they were 64 bits words. This makes it possible to define a C type `doubleCaster` for manipulating floating-point numbers either as floating-point numbers, or as 64 bits words:

```c
#include <stdio.h>
#include <stdint.h>

typedef union {
  double d;
  uint64_t l;
} doubleCaster;

int main() {
  doubleCaster xdb;

  xdb.d = 3.125;
  printf("0x%016llx\n",xdb.l);

  return 0;
}
```

What is printed by this code?

3. By successive iterations, propose a way to decompose a double precision number $x > 2^{-1021}$ into $E \in \mathbb{Z}$ and $m \in \mathbb{F}$, $1 \leq m < 2$ such that $x = 2^E \cdot m$.
4. Using C type `doubleCaster`, propose a way to decompose a double precision number $x > 2^{-1021}$ into $E \in \mathbb{Z}$ and $m \in \mathbb{F}$, $1 \leq m < 2$ such that $x = 2^E \cdot m$.
5. Extend the method proposed in the previous question to deal with subnormals. We will assume that $x > 0$.