Unité d'Enseignement :

Concours Programmation

Gilles Menez

Université de Nice – Sophia-Antipolis
Département d'Informatique
email : menez@unice.fr
www : www.i3s.unice.fr/~menez

27 septembre 2018: V 1.1

Table of contents

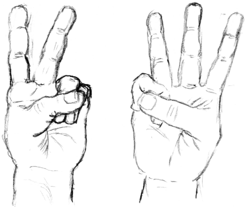
Part 1 : Introduction

Computers and Data abstraction

- An old history.
- Abstraction remains abstractions.
- How to make by himself ?

Counting

Soon after language develops, it is safe to assume that humans begin counting to represent , amount of cows, volume of wine, area of fields, . . .
It is safe to assume too that fingers and thumbs provide nature's abacus.



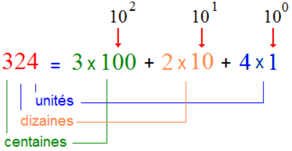
Rmks :

- The decimal system is no accident. Ten has been the basis of most counting systems in history.
- Abacus from greek "dust table" is the generic name given to "planar mechanical instrument" for computation and counting.

Modern Numbers

Arab Numbers (and 10 basis digit system) are a modern representation : 4th century AC

	1	2	3	4	5	6	7	8	9	0
Arabes :	١	٢	٣	٤	٥	٦	٧	٨	٩	٠
Indiens :										
Devanāgarī	१	२	३	४	५	६	७	८	९	०
Bengālī	১	২	৩	৪	৫	৬	৭	৮	৯	০
Gurūmukhī	੧	੨	੩	੪	੫	੬	੭	੮	੯	੦



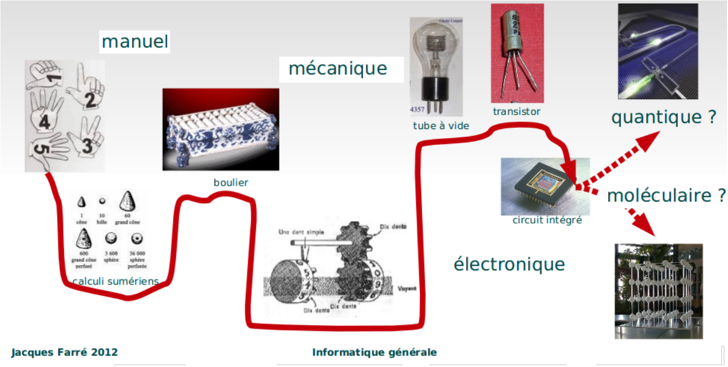
They :

- ➢ still are abstractions of volume, amount, and so on. in $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, ...
- ➢ allow counting, writing **and computing** !

First machines - Numbers first

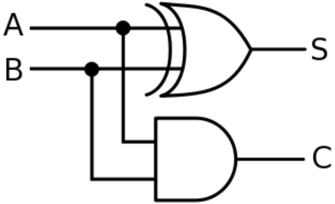
It is unknown when exactly were developed first devices to facilitate calculation, such as the counting board, or abacus.

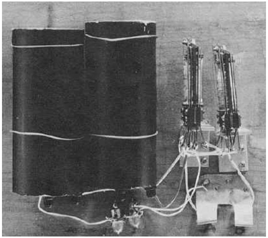
- but probably very shortly after counting.



First electronic digital computer

So first machines and more particularly first electro*.* machines were always dedicated to numbers processing.



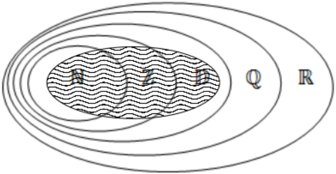


G.R. Stibitz - first electronic digital computer (half-adder 1 bit)

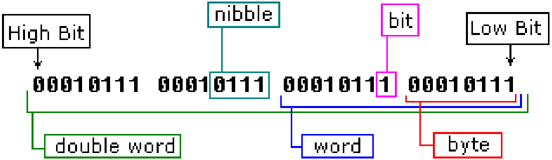
➢ Characters will be introduced in modern computing machines when a dialog will be possible between the machine and its user.

Data words

Machine numbers are abstractions of mathematical numbers :



- They are supported by a bit, a group/words (finite) of bits whose logical nature is compatible with electronic processing (on-off).



Any coding your are using, as the software representation is finite, we can only represent a subset of values : 2^n !

Coding numbers

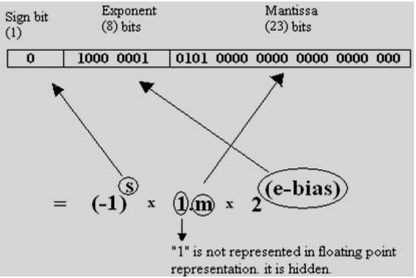
Most of machines will offer two well known coding methods :

① Two's Complement for signed integers : $v = \sum_{i=0}^{i<n} v_i * 2^i$

• $12_{ten} - 5_{ten} = 12_{ten} + (-5_{ten})$

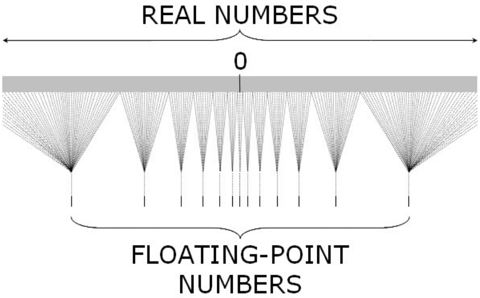
	0000 0000 0000 0000 0000 0000 0000 1100	(12_{ten})
+	1111 1111 1111 1111 1111 1111 1111 1011	(-5_{ten})
=	0000 0000 0000 0000 0000 0000 0000 0111	(7_{ten})

② IEEE 754 for floats



Abstractions : holes and bounds

These codes remain abstractions (size limited) and are inevitably **weaker in range and in precision** . . . than real world quantities/numbers (and mathematic abstractions)



Density of floats : http://jasss.soc.surrey.ac.uk/9/4/4.html

➢ using the types float or double in C, roughly half of all representable floating-point numbers are between -1 and 1

Simple precision IEEE754 values

Type	Exposant	Mantisse	Valeur	Ecart / préc
Zéro	0000 0000	000 0000 0000 0000 0000 0000	0,0	
Plus petit **nombre dénormalisé**	0000 0000	000 0000 0000 0000 0000 0001	$1{,}4\times10^{-45}$	$1{,}4\times10^{-45}$
Nombre dénormalisé suivant	0000 0000	000 0000 0000 0000 0000 0010	$2{,}8\times10^{-45}$	$1{,}4\times10^{-45}$
Nombre dénormalisé suivant	0000 0000	000 0000 0000 0000 0000 0011	$4{,}2\times10^{-45}$	$1{,}4\times10^{-45}$
Autre nombre dénormalisé	0000 0000	100 0000 0000 0000 0000 0000	$5{,}9\times10^{-39}$	
Plus grand nombre dénormalisé	0000 0000	111 1111 1111 1111 1111 1111	1.17549421×10^{-38}	
Plus petit **nombre normalisé**	0000 0001	000 0000 0000 0000 0000 0000	1.17549435×10^{-38}	$1{,}4\times10^{-45}$
Nombre normalisé suivant	0000 0001	000 0000 0000 0000 0000 0001	1.17549449×10^{-38}	$1{,}4\times10^{-45}$
Presque le double	0000 0001	111 1111 1111 1111 1111 1111	$2{,}35098856\times10^{-38}$	$1{,}4\times10^{-45}$
Nombre normalisé suivant	0000 0010	000 0000 0000 0000 0000 0000	$2{,}35098870\times10^{-38}$	$1{,}4\times10^{-45}$
Nombre normalisé suivant	0000 0010	000 0000 0000 0000 0000 0001	2.35098898×10^{-38}	$2{,}8\times10^{-45}$
Presque 1	0111 1110	111 1111 1111 1111 1111 1111	0,99999994	
1	0111 1111	000 0000 0000 0000 0000 0000	1,00000000	$0{,}6\times10^{-7}$
Nombre suivant 1	0111 1111	000 0000 0000 0000 0000 0001	1,00000012	$1{,}2\times10^{-7}$
Presque le plus grand nombre	1111 1110	111 1111 1111 1111 1111 1110	$3{,}40282326\times10^{38}$	
Plus grand nombre normalisé	1111 1110	111 1111 1111 1111 1111 1111	$3{,}40282346\times10^{38}$	2×10^{31}
Infini	1111 1111	000 0000 0000 0000 0000 0000	Infini	
NaN	1111 1111	010 0000 0000 0000 0000 0000	*NaN*	

http://fr.wikipedia.org/wiki/IEEE_754

Scalar abstractions : Numbers and characters

As **a conclusion to this introduction**, we know that since numbers are major (and historical) processing objects, all machines and all programming languages will provide these fundamentals of computation : scalar types !

- int, short, float, double, char, ...

These non-splittable (from a programming view) objects are matched, in the machine with memory word(s), coding standards and **dedicated operations**.

They are what a programming machine "naturally" offers :

- **"off the shelf"** ways to abstract numbers and characters.

Scalar abstractions : Limitations

Some limitations / issues come with "off the shelf" abstractions :

① **There is no "standard"** (over CPUs, compilers, languages) in this domain and for example different CPUs support different integral data types.

Typically, hardware will support both signed and unsigned types, but only a small, fixed set of widths.

② Alfred AHO (google him .. you will see) defines software science as "science of abstraction".

- ➢ You build your universe, where physicians take THE universe as it is.

But reality is often bigger and smaller than our abstraction.

- ➢ Scalar types will certainly miss range and a precision.
- ➢ Scalar operator set ('+', '-', . . .) needs to be enlarged.

Improvements

The question now is :

Can a computer process **something else than** a number or a character "off the shelf" ?

It should because :

- all potentially data to process are not "numbers" or "characters".

  For example a network address is most of the time a group of bytes whose structure/coding is not twos-complement nor IEEE754.

  All I/O peripherals have probably a specific protocol data unit. If the machine controls a peripheral (a robot ?) data units associated to the robot status will mix numbers, status bits, ...
- given pointed limitations and particularly the range, the programmer could organize several numbers to build an augmented range bignumber.
- some arithmetic operations are not provided by a machine (CPU).

  For example Fast Fourier Transform Butterfly mirror indexing or Endianess.

Solution

As a consequence, programmers **have to build <u>their</u> abstractions** and develop algorithms to access and/or give augmented semantic to bit words.

They will be given (by the hardware, the compiler and the language) memory words and dedicated operators (for example "bitwise operators").

As a result, they will augment these "off the shelf" abstractions.

- ➢ Let's try on examples !

Part 2 : Operators

Opérateurs Binaires : &, |, ^ , ~ , >>, <<

Le langage C comporte plusieurs opérateurs "bit à bit" (et, ou, ou-exclusif, complément à 1, décalages) qui permettent d'effectuer des manipulations de bits sur une expression (constante littérale, variable, ...) entière.

&, |, ^ , ~ , >>, <<

- Le langage C est, dès sa création, utilisé pour créer des applications systèmes.

  Dans le cadre de ce type d'applications, le rôle de ces opérateurs est vital.

Ces opérateurs sont aujourd'hui présents dans bien d'autres langages : Java, Python, ...

https://en.wikipedia.org/wiki/Bitwise_operation

Les différents opérateurs présentés ici **ne peuvent s'appliquer qu'aux expressions entières** (i.e. types char et int).

- On « casse » l'abstraction « nombre entier / codage complément à 2 » pour s'en servir autrement !
- Ce n'est plus qu'un **groupement de bits** dont la sémantique n'est plus nécessairement celle d'un entier.

Les opérateurs binaires peuvent se répartir en trois groupes :

① l'opérateur de complémentation à un : ˜

② les opérateurs logiques binaires : &, |, ˆ

③ les opérateurs de décalage : >>, <<

Complémentation à un : ~

Cet opérateur est aussi désigné par le « non logique ».

Exemple d'utilisation :

```
short a = 7;
                /* donc a : 0000000000000111 */
short b = 0;
b = ~a;
                /* donc b : 1111111111111000 */
```

Opérateurs Binaires : &, |, ^

① « et logique » :

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

```
short a = 7;    /* a : 0000000000000111 */
short b = 8;    /* b : 0000000000001000 */
short c;
c = a & b;      /* c : 0000000000000000 */
```

② « ou logique »

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

```
short a = 0x7;    /* a : 0000000000000111 */
short b = 0x8;    /* b : 0000000000001000 */
short c;
c = a | b;        /* c : 0000000000001111 */
```

Opérateurs Binaires : &, |, ^

③ « ou exclusif logique »

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

```
short a = 0x7;   /* a : 0000000000000111 */
short b = 0x9;   /* b : 0000000000001001 */
short c;
c = a ^ b;       /* c : 0000000000001110 */
```

Attention à ne pas confondre : && et & , ni || et |

Les opérateurs de comparaison rendent des valeurs 0 ou 1 :

2 && 1 rend 1 (vrai)

Les opérateurs bit à bit rendent des valeurs sur le domaine des entiers :

2 & 1 rend 0 (faux)

Opérateurs Binaires de Décalage : >> , <<

```
short a,b,c;
a = 0x6db7; /* = 28087 = 0110  1101  1011  0111 */
b = 0xa726; /* = 42790 = 1010  0111  0010  0110 */

/* Décalage à gauche bit à bit */
c = a << 6; /* 0110  1101  1100  0000 = 0x6dc0 */

/* Décalage à droite bit à bit */
c = a >> 6; /* 0000  0001  1011  0110 = 0x01b6 */
```

- Le premier opérande, de type entier, est le motif binaire à décaler.
- Le second est un entier **non signé**, qui précise le nombre de déplacements.
  Si le déplacement est supérieur à la largeur du motif, l'opération a un comportement "indéfini" !

Remarque : Décaler à gauche de 1 bit, c'est multiplier par 2. Décaler à droite, c'est diviser par 2.

Opérateurs de décalage : conservation du signe

Si le premier opérande délivre une valeur unsigned, le décalage est un décalage logique : **les bits libérés sont remplis avec des zéros**.

Sinon, le décalage peut être logique ou arithmétique (les bits libérés sont remplis avec le bit de signe), **cela dépend de l'implémentation**.

```
int main(int argc, char *argv[]){
  short a,b;

  a = -1;
  printf("%d = %x\n",a,a);
  b = a << 1; /* Décalage à gauche bit à bit */
  printf("%d = %x\n",b,b);

  a = -1;
  b = a >> 1;   /* Décalage à droite bit à bit */
  printf("%d = %x\n",b,b);
}
```

```
[!]> a.exe

-1 = ffffffff
-2 = fffffffe



-1 = ffffffff
```

Utilisations et Applications

Les opérateurs bit à bit sont utilisés dans différentes applications :

① "Anecdotiques",

- ➢ Multiplier/Diviser par 2 (<<,>>)
- ➢ Echanger deux variables avec un « ou exclusif » sans avoir besoin d'une troisième variable mémoire.

```c
int x,y;
x = x^y; /* Il faut savoir que x^x = 0 */
y = x^y;
x = x^y;
```

② D'autres vitales : les **masquages**
https://en.wikipedia.org/wiki/Mask_(computing)

Masquages

Définition :

Un processus de **transformation d'un motif binaire en un autre**, au moyen d'une opération logique binaire.

① Le motif d'origine est l'un des opérandes impliqués.

② Le second opérande, appelé le **masque**, est un motif binaire choisi spécialement pour obtenir la transformation désirée.

Le masque en « et » permet d'isoler/sélectionner une partie d'un motif binaire.

Le masque en « ou » permet de modifier/positionner à 1 une partie d'un motif binaire.

Masquage en « et » : un scénario d'utilisation

On vous a confié l'écriture du programme de gestion d'un ascenseur,

- Les 3 premier bits d'une variable entière « S » correspondent à l'état de l'ascenseur.

L'organisation de ces bits dans cette variable est la suivante :

Bit 0 et 1	Ces deux bits représentent le niveau actuel de l'ascenseur.
Bit 2	à « 1 », l'ascenseur contient des passagers sinon « 0 ».

Analyse de l'état de l'ascenseur

Vous voulez savoir si l'ascenseur contient des passagers ?

En considérant la globalité de la variable (ou plutôt ses 3 premiers bits), il faut écrire qu'il y a un passager dans l'ascenseur si :

S vaut 4 (100) ou 5 (101) ou 6(110) ou 7(111)

Il serait bien plus simple de ne regarder que le bit concerné par cette information :

- Si on teste (S&0x4) :

  Cette expression ne peut prendre que deux valeurs : 0 ou 4.
  soit celles du bit 2 de S !

- Si on veut faire encore plus commode, alors :

  $$((S >> 2)\&0x1)$$

  ne peut prendre que deux valeurs : 0 ou 1.

Masquage en « ou » : un scénario d'utilisation

Vous écrivez toujours un programme de gestion d'ascenseur,

- ➢ une variable entière « Cmd » commande son déplacement.

L'organisation des bits dans cette variable est la suivante :

Bit 0	Moteur monte
Bit 1	Moteur stoppe
Bit 2	Moteur descend
...	...
Bit 7	Allumage intérieur cabine

Vous voulez demander au moteur de descendre ?

```
Cmd= Cmd & 0xfff8; /* Met à 0 les bits du moteur*/
Cmd |= 0x4; /* Met à 1 le bit de descente */
```

Champs de bits

Il est parfois nécessaire pour le programmeur de décrire la structure d'un mot machine, cela pour plusieurs raisons :

- ➢ modéliser de façon plus fine ;
- ➢ un mot de l'espace mémoire est un registre découpé en différents champs de bits ;
- ➢ pour des raisons de gain de place, on désire faire coexister plusieurs variables à l'intérieur d'un entier.

```
/* Modélisation du bus de commande d'un ascenseur */
struct cmd_ascenseur {
  unsigned int moteur : 2;      // sur 2 bits
  unsigned int leds : 3;        // sur 3 bits
  unsigned int lumiere : 1;     // sur 1 bit
  unsigned int capteurniv : 4;  // sur 4 bits
};
```

Champs de bits : contraintes d'utilisations

Attention à la portabilité de ce concept !

① Un champ de bits **ne peut pas être d'une taille supérieure à celle d'un int**.

② Un champ de bits ne peut pas être à cheval sur deux int.

③ L'ordre dans lequel sont mis les champs de bits à l'intérieur d'un mot dépend du compilateur.

- ➢ Si on utilise les champs de bits pour gagner de la place, cela n'est pas gênant.
- ➢ Dans le cas où on les utilise pour décrire une ressource matérielle de la machine (registre, mot d'état programme, etc), il est bien sûr nécessaire de connaître le comportement du compilateur.

④ Un champ de bit déclaré comme étant de type int, peut en fait se comporter comme un int ou comme un unsigned int (cela dépend du compilateur).

- ➢ Il est donc recommandé d'une manière générale de déclarer les champs de bits comme étant de type unsigned int.

⑤ Un champ de bits n'a pas d'adresse, on ne peut donc pas lui appliquer l'opérateur adresse de (&).

Part 3 : Representation of numbers

Corrigés des exos de L2 (cours J.C. Régin) I

```c
#include<stdio.h>
int getBit(unsigned int x, unsigned int pos){
  /* pos in [31, 0] */
  return (x & (1<<pos))>>pos;
}
unsigned int setBit(unsigned int x, int pos){
  return x | (1<<pos);
}
unsigned int clearBit(unsigned int x, int pos){
  return x & (~(1<<pos));
}
unsigned int toggleBit(unsigned int x, int pos){
  unsigned int tmp = x & (1<<pos);
  if (tmp == 0)
    return setBit(x,pos);
  else
    return clearBit(x,pos);
}
unsigned int defineBit(unsigned int x, int pos, int bool){
  if (bool == 1)
    return setBit(x, pos);
  else
    return clearBit(x,pos);
}
void printbits(unsigned int x){
  int i;
  for (i=31 ; i>=0 ; i--){
    printf("%d",getBit(x,i));
  }
  printf("\n");
}
```

Corrigés des exos de L2 (cours J.C. Régin) II

```c
/*===============================================================*/
int main(void){
  unsigned int v = 5;
  printf("%x\n",getBit(v,0));
  printf("%x\n",getBit(v,1));
  printf("%x\n",getBit(v,2));
  printf("%x\n",setBit(v,1));
  printf("%x\n",clearBit(v,1));
  printf("%x\n",toggleBit(v,1));
  printbits(v);
  // bitfields!!!!
}
```

cf https://graphics.stanford.edu/~seander/bithacks.html

We keep bitwise arithmetic problems solving for the following slides.

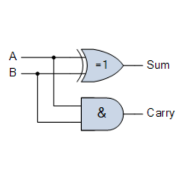
Unsigned addition

Let's see now how addition can be realized only with bitwise operators !

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

#define TYPE unsigned char
/*
#define TYPE unsigned long int
#define TYPE unsigned int
*/
TYPE add(TYPE a, TYPE b){
  // from : https://www.geeksforgeeks.org/add-two-numbers-without-using-arithmetic-operators/

  while (b != 0){  // Iterate till there is no carry
    // carry now contains common set bits of a and b
    TYPE carry = a & b;

    // Sum of bits of a and b where at least one of the bits is not set
    a = a ^ b;

    // Carry is shifted by one so that adding it to a gives the required sum
    b = carry << 1;
  }

  return a;
}

/*=========================================================*/

int main(void){
  printf("%x\n",add(2,2));
}
```

The logical background

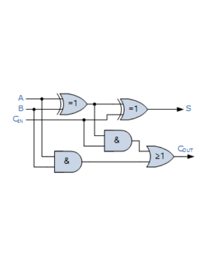
Symbol



Truth Table

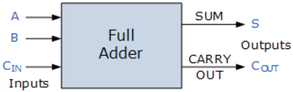
B	A	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

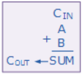
Symbol

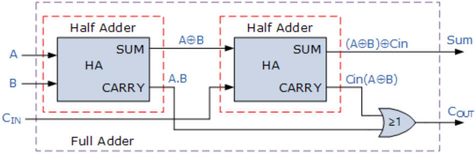


Truth Table

C-in	B	A	Sum	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1







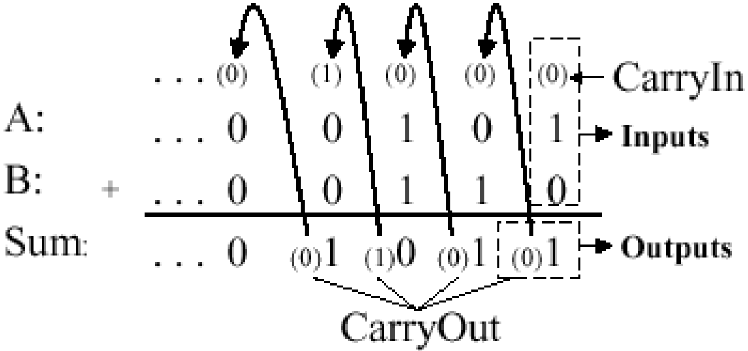
https://www.electronics-tutorials.ws/combination/comb_7.html

Half Sum $= A \oplus B$ Half Carry $= A.B$

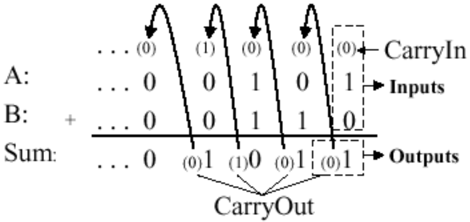
Sum $= (A \oplus B) \oplus Cin$ Cout $= C_{in}(A \oplus B) + A.B$

n-bits Unsigned addition

A classical algorithm with **propagation of the carry** between successive powers of two.



n-bits Unsigned addition : overflow !



The only difficulty adding unsigned numbers occurs when you add numbers that are resulting in a too large number :

- The last carry is 1 ! It overflows the size of a number and so it cannot be naturally represented in the machine.
- To detect and compensate for overflow (unsigned arihmetic context only !), one needs n+1 bits if an n-bit number representation is employed.
  For example, in 32-bit arithmetic, 33 bits are required to detect or compensate for overflow.

Overflow **not overflow flag** !

http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

Do not confuse the English verb "to overflow" with the "overflow flag" in the ALU. The verb "to overflow" is used casually to indicate that some math result doesn't fit in the number of bits available ; it could be integer math, or floating-point math, or whatever.

The "overflow flag" is set specifically by the ALU as described below, and it isn't the same as the casual English verb "to overflow".

In English, we may say "the binary/integer math overflowed the number of bits available for the result, causing the carry flag to come on". Note how this English usage of the verb "to overflow" is *not* the same as saying "the overflow flag is on".

A math result can overflow (the verb) the number of bits available without turning on the ALU "overflow" flag (cf two's complement representation properties).

This would result an overflow !

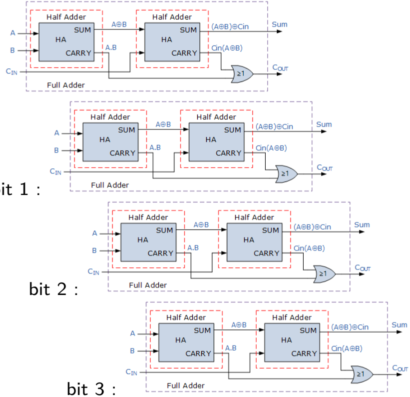
The hardware way (on 4 bits)

bit 0 :

bit 1 :

bit 2 :

bit 3 :



Ripple Carry Adder :

Cascading (by carry) full adder.

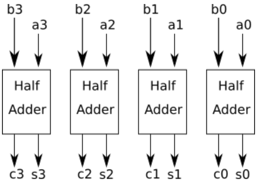
- ✘ No storage . . . just signal propagation !
  The hardware way.
- ✘ Carry propagation delay ! ! !

Many, many others/better hardware solutions . . . .

The software way (on 4 bits)

We analyze the flow of the algorithm used in the program given previously :

➢ All bits are processed in parallel.



① On the first iteration, the program computes s_i and c_i for each bit using half adder (l14, l17).

② These informations are stored in reused variable a and carry variable.

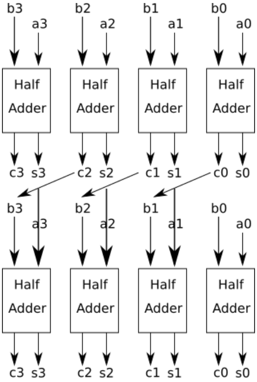
③ As all bits are processed in parallel, the propagation of carry (if any) will be realized by shifting (l20).

If there is no carry on (b==0), we can stop.

But suppose $(a_0 + b_0)$ generated a carry. Current value of s_1 is false since it did not take it into account :

➢ We have to iterate (l12) to propagate the carry to upper bits.

We need to shift left : (l20).



After second iteration s_1 is correct since it integrates the carry for the previous bit :

$$s_1 = s_1 + c_0 = (a_1 + b_1) + c_0.$$

If other carry (c_1, c_2, c_3) are 0, s_i remains s_i else they will be added.

Do you get it ?

Overflow

When overflow occurs on unsigned integer addition and subtraction, contemporary machines invariably **discard the high-order bit of the result and store the low-order bits that the adder naturally produces**.

No overflow on 4 bits :

carry		0 1 1 0
operand 1		0 1 1 0
operand 2	+	0 1 1 1
		1 1 0 0

Overflow on 4 bits :

carry		1 1 1 0
operand 1		0 1 1 0
operand 2	+	1 1 1 1
		0 1 0 0

The carry has probably been discarded : The **result is false** !

High level programming

This section discusses methods that a programmer might use to detect when overflow has occurred, without using the machine's "status bits" that are often supplied expressly for this purpose.

This is important, because some machines do not have such status bits , and even if the machine is so equipped, it is often difficult or impossible to access the bits from a high-level language.

Overflow prediction

Given the range of number (deduced from size of words used and coding), we can propose some tests to predict if an arithmetic operation will overflow :

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <signal.h>
#include <stdint.h>

int check_addition_overflow(unsigned int a, unsigned int b) {
    if (a > 0 && b > (UINT_MAX - a)) {
      // if ((a+b) < a) // make an overflow and see : unsigned only !
      // proof in : /* https://stackoverflow.com/questions/33948450/c-detect-unsigned-int-overflow-of-addition */
      printf("overflow will occur\n");
      return 1;
    }
    return 0;
}
```

Overflow Traps

In the **signed** context, one option of gcc has to be mentioned : `-ftrapv`

➢ This option generates traps for signed overflow on addition, subtraction, multiplication operations.

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <signal.h>
#include <stdint.h>

void h(int signal){
  printf("caught signal exiting\n");
  exit(1);
}
int main(void){
  int x = INT_MAX;
  int y;
  signal(SIGABRT,h);
  y = x+1;
  printf("%d\n",y);
  return 0;
 }
```

```
menez@vtr ~/EnseignementsCurrent/
$ gcc -ftrapv trap_overflow.c

menez@vtr ~/EnseignementsCurrent/
$ ./a.out
-2147483648

menez@vtr ~/EnseignementsCurrent/
$ gcc -ftrapv trap_overflow.c

menez@vtr ~/EnseignementsCurrent/
$ ./a.out
caught signal exiting
```

But . . .

➢ For "unsigned int", the program wraps (goes most negative).

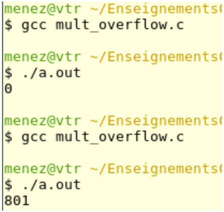
➢ If not gcc ?

Intel x86 and x64 target specific

The Intel CPUs have the so-called EFLAGS-register, which is filled by the processor after each integer arithmetic operation : http://en.wikipedia.org/wiki/EFLAGS

- The relevant flags are the "Overflow" Flag (mask 0x800) and the "Carry" Flag (mask 0x1).
- To **interpret them correctly**, one should consider if the operands are of signed or unsigned type.

```c
#include <stdio.h>
static inline size_t query_intel_x86_eflags( const size_t query_bit_mask ){
#ifdef __GNUC__
    // this code will work only on 64-bit GNU-C machines;
    size_t eflags;
    __asm__ __volatile__(
        "pushfq␣\n\t"
        "pop␣%%rax\n\t"
        "movq␣%%rax,␣%0\n\t"
        :"=r"(eflags)
        :
        :"%rax"
        );
    return eflags & query_bit_mask;
#else
#pragma message("No␣inline␣assembly␣with␣this␣compiler!")
    return 0;
#endif
}
int main(int argc, char **argv){
  int x = 10000;//00000;
  int y = 20000;
  int z = x * y;
  int f = query_intel_x86_eflags( 0x801 );
  printf( "%X\n", f );
}
```



GCC builtins

The following built-in functions of gcc allow performing simple arithmetic operations (**signed** and **unsigned**) together with checking whether the operations overflowed.

https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

```c
#include <stdio.h>
#include <assert.h>
#include <limits.h>

int main(void){
  int x = INT_MAX;
  int y;
  int ovf;
  unsigned int ux = UINT_MAX;
  unsigned int uy;

  printf("%d\n",x);
  ovf = __builtin_add_overflow(x, INT_MAX, &y);
  printf("%d\n",y);
  if (ovf)
    printf("Overflow !\n"); /* Pos + Pos => Neg ? */

  ovf = __builtin_uadd_overflow(ux, 1, &uy);
  printf("%d\n",uy);
  if (ovf)
    printf("Overflow !\n"); /* 0 ? the carry has been trapped */

  return 0;
 }
```

```
menez@vtr ~/EnseignementsCurrent,
$ gcc builtin.c

menez@vtr ~/EnseignementsCurrent,
$ ./a.out
2147483647
-2
Overflow !
0
Overflow !
```

Integers : the range problem

We know machine integers are only abstractions and as a consequence their ranges can be programming issues.

https://en.wikipedia.org/wiki/Integer_(computer_science)

In many cases, the task or the programmer can guarantee that the integer values in a specific application will not grow large enough to cause an overflow.

- ➡ Such guarantees may be based on pragmatic limits : "a school attendance program may have a task limit of 4,000 students".
- ➡ The programmer may design the computation so that intermediate results stay within specified precision boundaries.

But sometimes (more and more in modern applications), we **need to get off the range of the machine** and/or of the language.

Examples of large numbers describing everyday real-world objects are (vs $18x10^{18}$ with 64 bits) :

- The estimated number of atoms in the observable universe (10^{80})
- Earth's mass consist of about ($4x10^{51}$) nucleons
- The number of cells in the human body (more than 10^{14})
- The number of neuronal connections in the human brain (estimated at 10^{14})
- The lower bound on the game-tree complexity of chess, also known as the "Shannon number" (estimated at around 10^{120})
- ...

Big Numbers

In computer science, arbitrary-precision arithmetic, also called bignum arithmetic, multiple-precision arithmetic, or sometimes infinite-precision arithmetic, indicates that calculations are performed on numbers whose digits of precision are limited **only by the available memory** of the host system.

This contrasts with the faster fixed-precision arithmetic found in most arithmetic logic unit (ALU) hardware, which typically offers between 8 and 64 bits of precision.

- ➢ Several modern programming languages have built-in or options to support for bignums : Lisp, Python, Perl, Haskell and Ruby.
- ➢ Others (C, C++, Java,) have libraries available for arbitrary-precision integer and floating-point math.

The hint :

Rather than store values as a fixed number of binary bits related to the size of the processor register, **these implementations typically use variable-length arrays of digits**.

Although this reduces performance, it eliminates the possibility of incorrect results (or exceptions) due to simple overflow.

It also makes it possible to guarantee that arithmetic results will be the same on all machines, regardless of any particular machine's word size.

The exclusive use of arbitrary-precision numbers in a programming language also simplifies the language, because a number is a number and there is no need for multiple types to represent different levels of precision.

So when ?

As a conclusion, arbitrary precision is used in applications where the speed of arithmetic is not a limiting factor, or where precise results with very large numbers are required : programming challenges ?

Factorial overflow

The factorial developped on natural precision is rapidly overflowing :

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

static unsigned long fact (unsigned long n) {
  if (n > 1) {
    return n * fact(n - 1);
  } else {
    return 1;
  }
}

int main (int argc, char **argv) {

  /* https://en.wikibooks.org/wiki/C_Progra
  printf("ULONG_MAX = %20lu\n",ULONG_MAX);
  /* ... */
  printf("fact(20)  = %20lu\n", fact(20));
  printf("fact(21)  = %20lu\n", fact(21));
  printf("fact(22)  = %20lu\n", fact(22));
  printf("fact(23)  = %20lu\n", fact(23)); // overflow
  printf("fact(24)  = %20lu\n", fact(24)); // overflow
  return 0;
}
```

```
menez@vtr ~/EnseignementsCurrent/Concc
$ gcc fact.c

menez@vtr ~/EnseignementsCurrent/Concc
$ ./a.out
ULONG_MAX = 18446744073709551615
fact(20)  =  2432902008176640000
fact(21)  = 14197454024290336768
fact(22)  = 17196083355034583040
fact(23)  =  8128291617894825984
fact(24)  = 10611558092380307456
```

Stack overflow

Addresses (in program stack) too can overflow ...

```c
#include <stdio.h>
unsigned long int sum_of_first_n_natural_numbers(unsigned long int n){
  /* The recursive way ... splash ! */
  if (n == 0)
    return 0;
  else
    return n + sum_of_first_n_natural_numbers(n-1);
}
int main(void){
  unsigned long int n;
  n = 100000;
  printf("result : %lu \n", n*(1+n)/2); // closed form
  printf("result : %lu \n", sum_of_first_n_natural_numbers(n));
  n = n*10; // => stack overflow
  printf("result : %lu \n", sum_of_first_n_natural_numbers(n));
  return 0;
}
```

```
menez@vtr ~/EnseignementsCurrent/
$ gcc sumn_integers.c

menez@vtr ~/EnseignementsCurrent/
$ ./a.out
result : 5000050000
result : 5000050000
Erreur de segmentation
```

C++/Boost Lib Factorial Big Num

```cpp
#include <boost/multiprecision/cpp_int.hpp>
#include <iostream>
namespace mp = boost::multiprecision;
using namespace std;
mp::cpp_int fact(unsigned long n){
  mp::cpp_int u = 1;
  for(int i = 1; i <= n; i++)
    u *= i;

  return u;
}
//==========================================
int main(){
  mp::cpp_int u;
  u = fact(100);
  cout << "100! = " << u << '\n';
  mp::cpp_int v = u / 100;
  cout << "99! = " << v << '\n';
}
```

```
menez@vtr ~/EnseignementsCurrent/ConcoursArnaud/Src
$ g++ boost_fact.cpp

menez@vtr ~/EnseignementsCurrent/ConcoursArnaud/Src
$ ./a.out
100! = 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
99! = 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

Factorial in Python

```
menez@vtr ~/EnseignementsCurrent/ConcoursArnaud/Src
$ python3
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170118] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> print(math.factorial(100))
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
>>> print(math.factorial(99))
9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000
>>> 
```

Challenge Pb : Big addition I

We want to add large numbers (unsigned integers) stored in strings.

- The numbers may be very large (may not fit in long long int).

Your program reads (from stdin) sequences of two strings containing big numbers, computes the sum and writes (on stdout) the string of the resulting number.

- If one of the two input strings is not a number (i.e all letters of the string are not digits or the string is empty) output should be the string "?"

Examples of Inputs / Outputs :

Input stream	Output stream
3333311111111111	
4442222221111	
	3377733333332222
7777555511111111	
3332222221111	
	7780887733332222
43	
1x	
	?
00930261	
4	
	930265

Challenge Pb : Big addition II

The last example/sample shows that **non significant leading 0 should be removed** in the output string !

Two problems : with and without "Arbitrary Arithmetic".

① **With "Arbitrary Arithmetic" facilities**

In the first version of the problem, **THERE IS NO CONSTRAINT ON THE LANGUAGE** you can use.

- ➢ So as an hint you should use a language able to deal with big integers : Python ?, Java ? . . .

② **Without "Arbitrary Arithmetic" facilities**

In the second version, **ALL LANGUAGES WITH ARBITRARY ARITHMETIC FEATURES ARE PROHIBITED !**

- ➢ So as an hint you should use C or C++ . . .

Challenge Pb : Big addition

Speaking about "efficiency" :

Why this challenge is quite simple ?

How to improve the challenge and the solution ?

Where will issues appear ?

Standard mutliplication algorithm

We want to multiply the multiplicand (a) by the multiplier (b) :

$$
\begin{array}{lrl}
\text{(a) multiplicand} & 2\,3\,9\,5\,8\,2\,3\,3 & \\
\text{(b) multiplier} & 5\,8\,3\,0 & \\
\hline
 & 0\,0\,0\,0\,0\,0\,0\,0 & (= \quad 23,958,233 \times 0) \\
 & 7\,1\,8\,7\,4\,6\,9\,9 & (= \quad 23,958,233 \times 30) \\
 & 1\,9\,1\,6\,6\,5\,8\,6\,4 & (= \quad 23,958,233 \times 800) \\
+ & 1\,1\,9\,7\,9\,1\,1\,6\,5 & (= \quad 23,958,233 \times 5,000) \\
\hline
 & 1\,3\,9\,6\,7\,6\,4\,9\,8\,3\,9\,0 & (= \quad 139,676,498,390)
\end{array}
$$

Several multiply algorithms exist :

➢ We use the most classical one where each digit of b contributes (by addition if non zero), given the power matching its position (that is why there is a shift), to the final result obtained by addition.

https://en.wikipedia.org/wiki/Multiplication_algorithm :

```
multiply(a[1..p], b[1..q], base) // Operands containing rightmost
                                 // digits at index 1
  product = [1..p+q]             // Allocate space for result
  for bi = 1 to q                // for all digits in b
      carry = 0
      for ai = 1 to p            // for all digits in a
          product[ai + bi - 1] += carry + a[ai] * b[bi]
          carry = product[ai + bi - 1] / base
          product[ai + bi - 1] = product[ai + bi - 1] mod base
      product[bi + p] += carry  // last digit comes from final carry
  return product
```

Remarks and focuses :

- Size of result : $p + q$
- base parameter : $a[i]$ "range" ?
- "$ai + bi - 1$" index to "shift left"
- The carry propagation

Challenge Pb : Big multiplication

Given two numbers (unsigned integers) as strings.

The numbers may be very large (may not fit in long long int), the task is to find multiply of these two numbers.

Example :

Input :

$$
\begin{aligned}
\text{str1} &= \text{"1235421415454545454545454544"} \\
\text{str2} &= \text{"1714546546546545454544548544544545"}
\end{aligned}
$$

Output :

2118187521397235888154583183918321221520083884298838480662480

Easy to check . . . Python integers !

Challenge Pb : Big Multiplicationn

Speaking about "efficiency" :

Why this challenge is quite simple ?

How to improve the challenge and the solution ?

Where will issues appear ?

Part 4 : Representation/Abstraction

IP addresses

An Internet address uses four bytes often specified in hexadecimal :

$$C0290614_{16}$$

Because it is easier to remember (for a human), this number is often specified as 4 values of 8 bits.The split is marked by "." (dot)

8 bits	.	8 bits	.	8 bits	.	8 bits

Hence the dot notation is :

16 basis :	C0	.	29	.	06	.	14
10 basis :	192	.	41	.	6	.	20

Challenge Pb : 1/3

① Faire une fonction qui prend deux paramètres :

- ➢ un entier en **base 16** supposé être une adresse Internet (donc sur 32 bits)
- ➢ un choix de base parmi les valeurs 10 ou 16

et qui affiche cette représentation pointée dans la base souhaitée.

Examples

➡ Inputs :

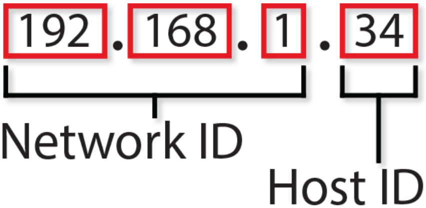
C0290614	16
C0290614	10
863B83AC	10
C0A80001	10

➡ Outputs :

C0.29.06.14
192.41.6.20
134.59.131.172
192.168.0.1

Semantic of IP addresses

To fully define the semantic of such an address, you have to know that this address contains two informations :

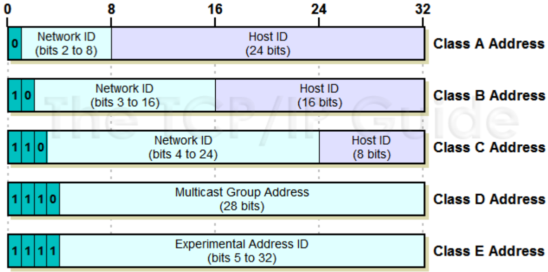


1. The **netid** identifies the network on which is the host .
2. The **hostid** identifies the host in this network.

Classfull way

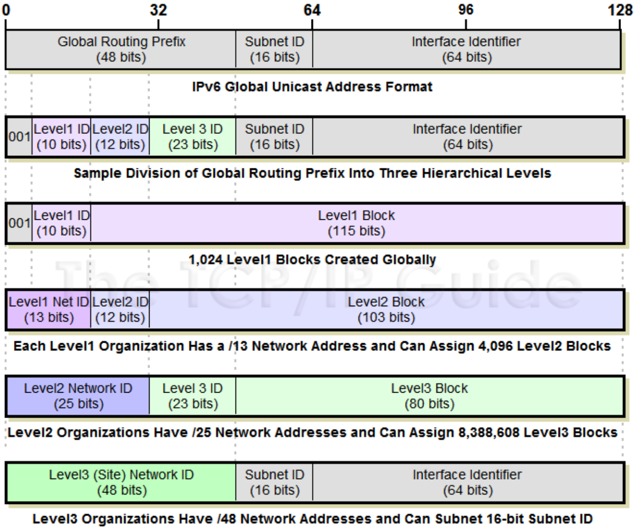
First versions of Internet suppose their will be 5 configurations for the semantic analysis of such an address : they are address classes .

- ➢ The MSB bits value decide which, among 5, configuration/semantic is used by the four bytes.
- ➢ Each configuration modifies the size of each part (net/host id) and is labeled with one of these letters **A,B,C,D ou E**.



http://www.tcpipguide.com/free/t_IPClassfulAddressingNetworkandHostIdentificationan-3.htm

IPv6 Unicast adress



Challenge Pb : 2/3

① Faire une fonction capable à partir d'une adresse Internet fournie par un entier en base 16 de rendre la classe de l'adresse sous forme d'un caractère.

Examples

➡ Inputs :

C0290614
863B83AC
C0A80001
F0040506

➡ Outputs :

C
B
C
E

Challenge Pb : 3/3

Cet exercice ne peut être réalisé que si l'exercice 1 fonctionne.

① Faire une fonction capable à partir d'une adresse Internet fournie par un entier en base 16 de donner la valeur entière en base 16 du **netid** **et** la valeur entière en base 16 du **hostid**.

Pour ce qui est des classes D et E, la valeur entière rendue (pour le **hostid** et le **netid**) est celle de l'adresse dans sa globalité.

Par exemple, si on a l'adresse $863B8317_{16}$,

Sa classe est B puisque le codage en base 2 de l'adresse commence par $1000\ldots_2$.

La fonction rendra

- un `netid` égal à la valeur $863B_{16}$ puisque ce type d'adresse code sur les 16 bits $(2 + 14)$ de poids fort le `netid`.
- un `hostid` égal à la valeur 8317_{16} puisque ce type d'adresse code sur les 16 bits de poids faible le `hostid`.

Examples

- Inputs :

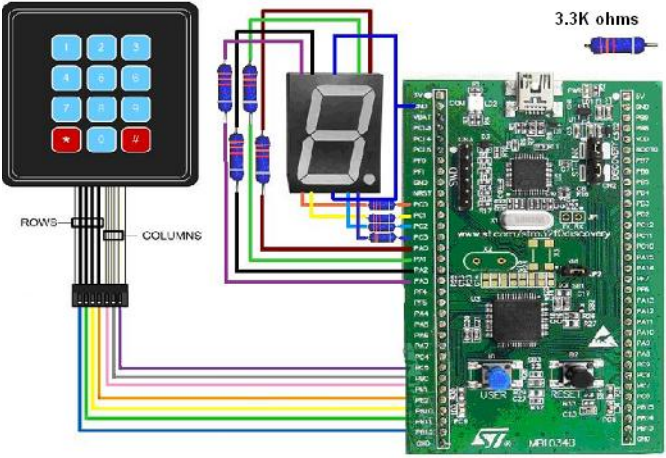
  C0290614
  863B83AC
  C0A80001
  F0040506

- Outputs :

C02906	14
863B	83AC
C0A800	01
F0040506	F0040506

System control (Embedded systems)

Program/code for STM32F0 to interface Keypad/keyboard
http://www.eeherald.com/section/design-guide/esmod6b.html



SET PINS			PRESSED KEY	OUTPUT PINS			
PC5	PB0	PB1		PB2	PB10	PB11	PB12
1	0	0	#	0	0	0	1
			9	0	0	1	0
			6	0	1	0	0
			3	1	0	0	0
0	1	0	0	0	0	0	1
			8	0	0	1	0
			5	0	1	0	0
			2	1	0	0	0
0	0	1	*	0	0	0	1
			7	0	0	1	0
			4	0	1	0	0
			1	1	0	0	0

Bitwise Access of "one by n=4" coding :

```
initgpio();

while(1)
{
GPIOC->BSRR = GPIO_Pin_5;//set bit as high
GPIOB->BRR = GPIO_Pin_0;//set bit as low
GPIOB->BRR = GPIO_Pin_1;//set bit as low
{
if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_12))//read input bit PB12
display(3);
if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_11)) //read input bit PB11
display(6);
if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_10)) //read input bit PB10
display(9);
if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_2)) //read input bit PB2
display(11);
}
GPIOC->BRR = GPIO_Pin_5;//set bit as low
GPIOB->BSRR = GPIO_Pin_0;//set bit as high
GPIOB->BRR = GPIO_Pin_1;//set bit as low
{
if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_12))
display(2);
if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_11))
display(5);
if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_10))
```

Part 5 : Dedicated operations

Example : FFT

FFT is for "Fast Fourier Transform" : **the most frequent "machine procedure" all categories** ?

- Used in all sounds, images, videos processing.
- **In 1990,** 40% **of all Cray supercomputer cycles were devoted to the FFT**.

FFT is a rapid algorithm to compute DFT (Discret Fourier Transform) :

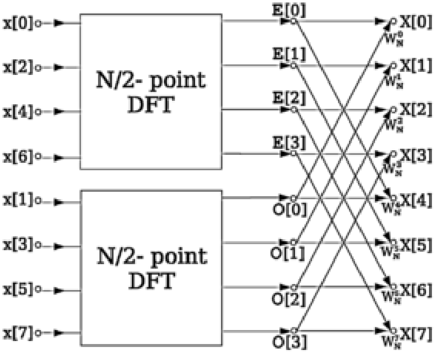
Given N signal samples : $x[0], \ldots, x[N-1]$
computes spectral representation of this signal :

$$X[k]_{(k:0\ldots N-1)} = \sum_{n=0}^{N-1} x[n] W_N^{kn} \tag{1}$$

where $W_N^{kn} = e^{-j\frac{2\pi}{N}*kn} = cos(\frac{2\pi}{N} * kn) - j * sin(\frac{2\pi}{N} * kn)$

- Many good properties on W_N^{kn} !

FFT is a divide and conquer formulation (thanks to properties on W_N^{kn})

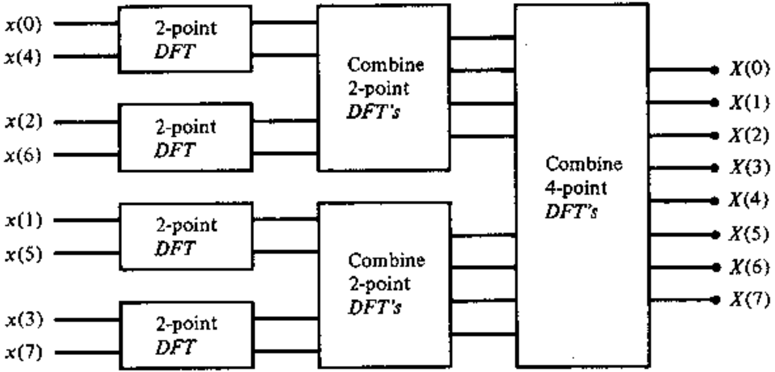


A decimation-in-time radix-2 FFT breaks a length-N DFT into two length-N/2 DFTs **followed** by a combining stage consisting of many butterfly operations.

You can find details/proofs on :

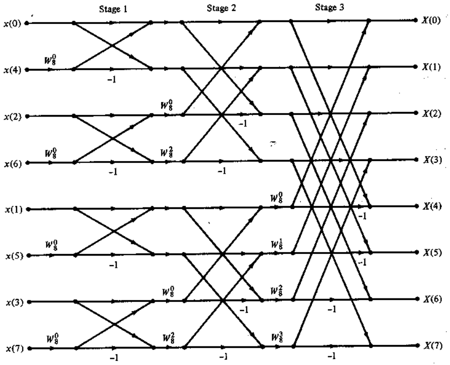
https://web.eecs.umich.edu/~fessler/course/451/l/pdf/c6.pdf

Repeat/Recurse "divide and conquer" :

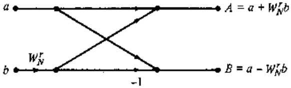


You will get $log_2(N)$ stages !

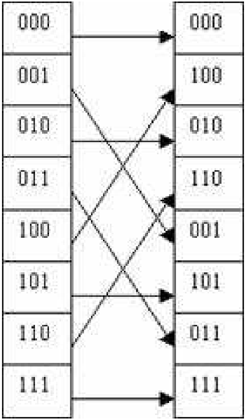
The same diagram with butterflies flow of data :



with each butterfly :



Look at the permutation needed on input samples $=>$ "mirror" or "bit reversed" !



Challenge Pb : Bit Reverse

On souhaite réaliser une fonction capable d'effectuer un "effet miroir" sur une partie de la représentation binaire d'un nombre entier.

Ainsi, si on applique cet effet sur les 3 bits de poids faible d'un nombre entier, on obtient :

$b_{31} \dots b_3 000_2$	devient	$b_{31} \dots b_3 000_2$
$b_{31} \dots b_3 001_2$	devient	$b_{31} \dots b_3 100_2$
$b_{31} \dots b_3 010_2$	devient	$b_{31} \dots b_3 010_2$
$b_{31} \dots b_3 011_2$	devient	$b_{31} \dots b_3 110_2$
$b_{31} \dots b_3 100_2$	devient	$b_{31} \dots b_3 001_2$
$b_{31} \dots b_3 101_2$	devient	$b_{31} \dots b_3 101_2$
$b_{31} \dots b_3 110_2$	devient	$b_{31} \dots b_3 011_2$
$b_{31} \dots b_3 111_2$	devient	$b_{31} \dots b_3 111_2$

L'effet miroir ne doit pas avoir d'effet de bord sur les bits de poids fort non concernés ($poids \in [31, 3]$ dans l'exemple).

① Proposer un programme réalisant cet effet sur les n bits de poids faible d'un nombre entier x.

Si n $\notin [32, 2]$ alors le nombre x n'est pas modifié. Sinon la valeur rendue est celle du nombre x avec un effet miroir sur ses n bits de poids faible.

On fournira une ligne avec x puis n (les deux en base 10).

Votre programme rend la valeur (en base 10) du nombre x avec un effet miroir sur ses n bits de poids faible.

Examples :

Inputs		Outputs
4	3	1
12	3	1
4	4	2
255	3	255
254	3	251
254	4	247

Fast(est ?) solution

I hope your are convinced this is a real problem.

May be your solution will be iterative ?

But as this is an important issue, people have proposed quite smart solutions : https://graphics.stanford.edu/~seander/bithacks.html

Reverse the bits in a byte with 4 operations (64-bit multiply, no division) :

```
 unsigned char b; // reverse this byte

b = ((b * 0x80200802ULL) & 0x0884422110ULL) * 0x0101010101ULL >> 32;}
```

Fastest solution

```
  unsigned char b; // reverse this byte

b = ((b * 0x80200802ULL) & 0x0884422110ULL) * 0x0101010101ULL >> 32;}
```

The following shows the flow of the bit values with the boolean variables a, b, c, d, e, f, g, and h, which comprise an 8-bit byte. Notice how the first multiply fans out the bit pattern to multiple copies, while the last multiply combines them in the fifth byte from the right :

```
                                                                abcd efgh (-> hgfe dcba)
*                                     1000 0000  0010 0000  0000 1000  0000 0010 (0x80200802)
-------------------------------------------------------------------------------
                           0abc defg  h00a bcde  fgh0 0abc  defg h00a  bcde fgh0
&                          0000 1000  1000 0100  0100 0010  0010 0001  0001 0000 (0x0884422110)
-------------------------------------------------------------------------------
                           0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
*                          0000 0001  0000 0001  0000 0001  0000 0001  0000 0001 (0x0101010101)
-------------------------------------------------------------------------------
                           0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
                0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
     0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
-------------------------------------------------------------------------------
0000 d000  h000 dc00  hg00 dcb0  hgf0 dcba  hgfe dcba  hgfe 0cba  0gfe 00ba  00fe 000a  000e 0000
>> 32
-------------------------------------------------------------------------------
                           0000 d000  h000 dc00  hg00 dcb0  hgf0 dcba  hgfe dcba
&                                                                      1111 1111
-------------------------------------------------------------------------------
                                                                       hgfe dcba
```

Index

Index :