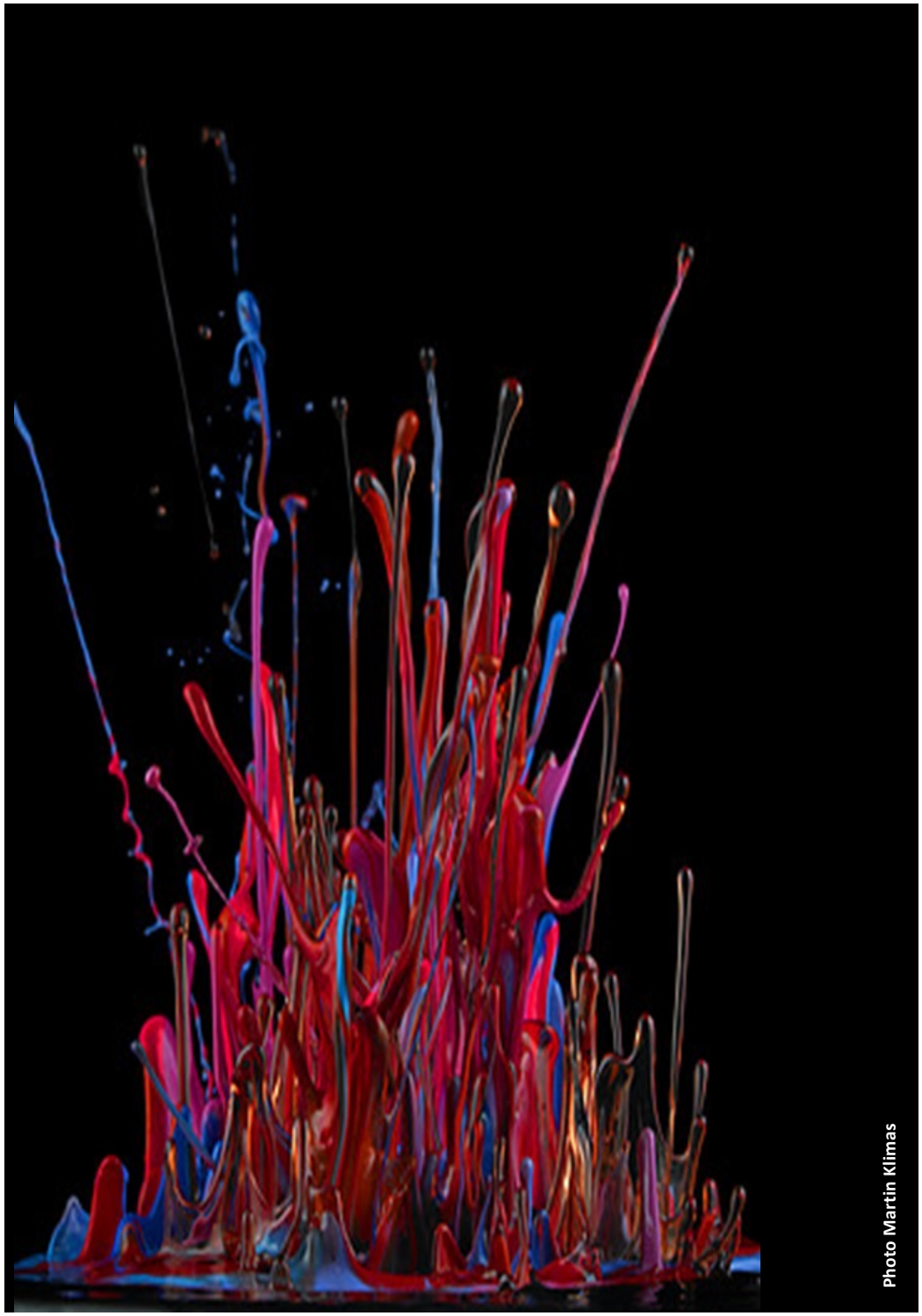


Photo Martin Klimas

EPFL

Programmation pour Ingénieur

Représentation des nombres et opérations bit à bit

ME 3e semestre

rev. 2025.3

Christophe Salzmann

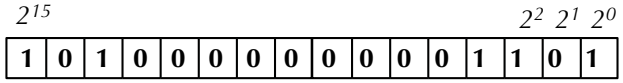
Laboratoire d'Automatique

modifications

- 17.09.2025, r2 – ajout slide 14 sélection/extraction
- 22.09.2025, r3 – slides 16,17 update var k

Représentation des nombres

- Un ordinateur/CPU comprend et travaille avec des 1 et 0
- Pour pouvoir être employés par le CPU, les nombres sont donc convertis de base 10 en base 2
- Le CPU gère des nombres binaires de taille donnée, ex: 32bit, 64bit
- Les entiers (base 10) représentés en base 2 ont donc une taille fixée par le nombre de bits à disposition, le nombre de valeurs possibles est également fixé par le nbr. de bits.



$1 \times 2^{15} + 0 \times 2^{14} + 1 \times 2^{13} + 2^3 + 2^2 + 2^0 = 32768 + 40960 + 8 + 4 + 1 = 73741$

- Les compilateurs définissent des constantes pour ces nombres maximums, ils peuvent également être calculé si l'on connaît la taille des nombres.

Ex: entier **non signé**, représenté sur 16 bit => 1111111111111111 => 0 .. 65535 = 2^{16}-1

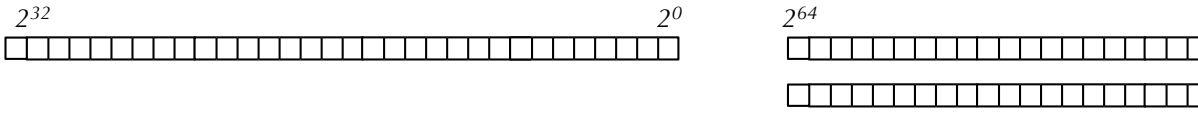
pour les entiers **signés** sur 16 bit => 1111111111111111 => -32768 .. 32767

C -> `UINT_MAX` (`#include <limits.h>`)

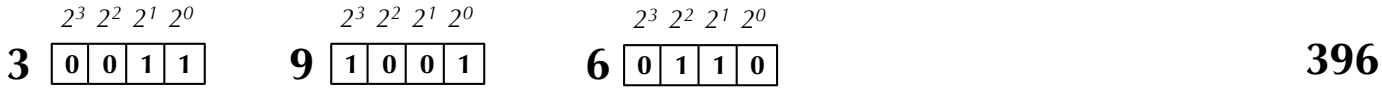
Comment faire pour représenter les nombres entre 0 et 1
ou des nombres plus grand qu'`INT_MAX` ?

Représentation des nombres

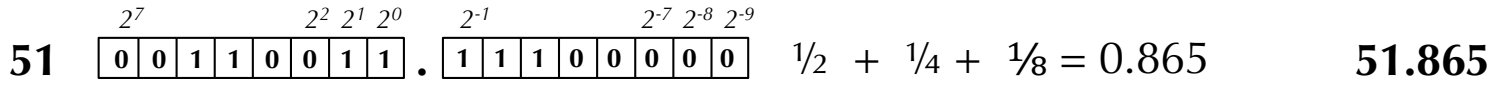
- Augmenter le nombre de bits employés pour représenter un nombre (32 -> 64)



- Encodage ***BCD***, chaque chiffre (0 à 9) est codé sur 4 bit (0000 à 1001), nécessite des traitements spécifiques



- Format ***fixed point*** définissant une partie entière et une partie décimale encodé sur un nombre fixe de bit, -> rapidité de calcul similaire aux entiers combinée avec la possibilité de représenter des nombres entre 0 et 1.

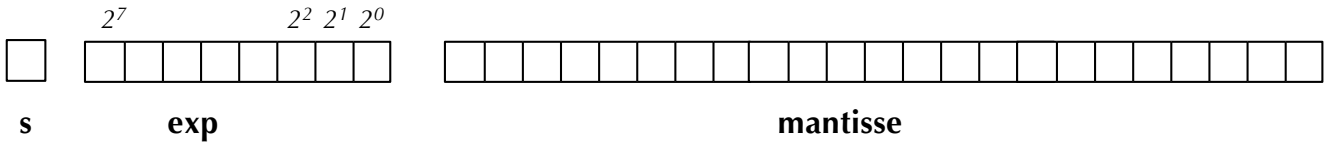


Représentation des nombres

- Format ***floating point*** définissant un nombre réel à l'aide d'un signe, d'une mantisse et d'un exposant. Ce format permet de placer la virgule n'importe où dans le nombre, d'où son nom. Le format *floating point* a été normalisé (IEEE 754)

Ex. réel simple précision *(single/float*-32 bits):

1 bit de signe (**s**), 8 bits d'exposant (**exp**), 23 bits de **mantisse**, avec **'1.'** implicite -> 1.xxxx)



$valeur = (-1)^s, 1.\text{mantisse} \times 2^{(\text{exp} - \text{décalage})} = 1.\text{mantisse} \times 2^{(\text{exp } -127)}$

déclage: $2^{\text{nbr de bit de l'exposant}}$, $2^{8-1}-1 = 2^{8-1}-1 = 127$ => décalage de -126 à 127

```
0  00000000  00000000000000000000001 =>    2^0-127 x 2^-23 x = 1/2^127 x = 1.4e-45
0  11111111  00000000000000000000000 =>   NaN
```

Représentation des nombres

- De par sa construction, un nombre réel ne peut pas forcément être représenté dans le format *floating point,* il est alors **approximé** en utilisant à la place l'**arrondi** le plus proche.
- L'opération d'arrondir un nombre est donc **forcément entachée d'erreur!**
- Il y a un également le risque d'un **effet cumulatif** des erreurs à prendre en compte. Il est dépendant de l'algorithme utilisé (itérations)

Ex.

```
float x= 0.1;   //  ->0.100000001490116119
```

- Par construction, les nombres en virgule flottante ne sont **pas espacés de façon identique** dans l'ensemble des réels : plus l'on est proche du plus petit nombre représentable, plus ils sont denses.

Représentation des nombres

- En raison du risque d'erreur d'arrondi, **ne jamais faire de test d'égalité stricte** '==' ou '!=' sur des nombres représentés en virgule flottante.

ex.

```
float x;
for(x=0.0; x != 10; x += 0.1)   // boucle infinie
        printf("%f\n",x);
```

- La **précision** des nombres représentés en virgule flottante **étant finie**, l'addition d'un nombre très petit à un nombre très grand (dans les bornes des nombres valides) peut n'avoir aucun effet et retourner le nombre très grand.
- De même l'ordre des opérations peut jouer un rôle.

ex.

avec `float a=10000, b=0.000001;`

`a` + `b` = `10000` = `a` !!!

même si `b` >> epsilon!

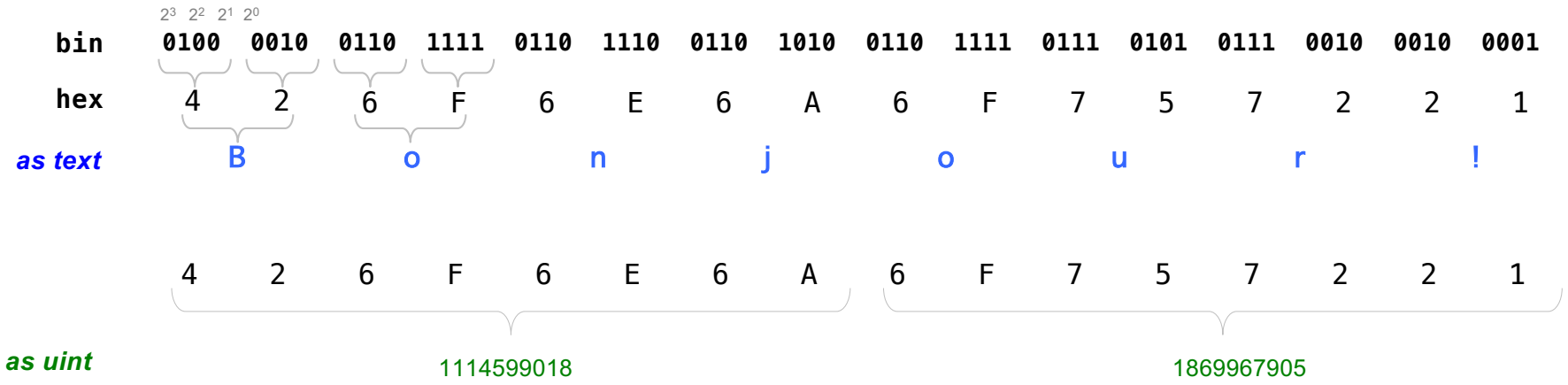
Représentation des nombres (et variables)

- Quel est le type d'une valeur en mémoire ?
- Pouvez-vous donner le type en regardant une suite de '1' et '0'?

  Ex: 0100 0010 0110 1111 0110 1110 …

- La réponse est **non**!
- Au mieux vous pouvez supposer le type en regardant une longue suite de '1' et '0'

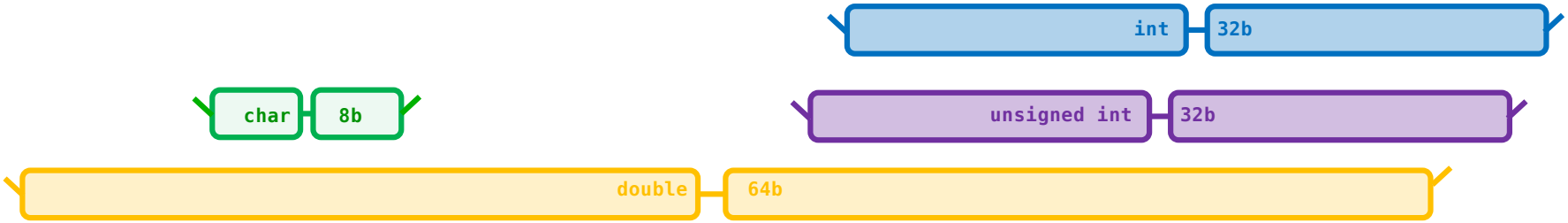
Exemple:



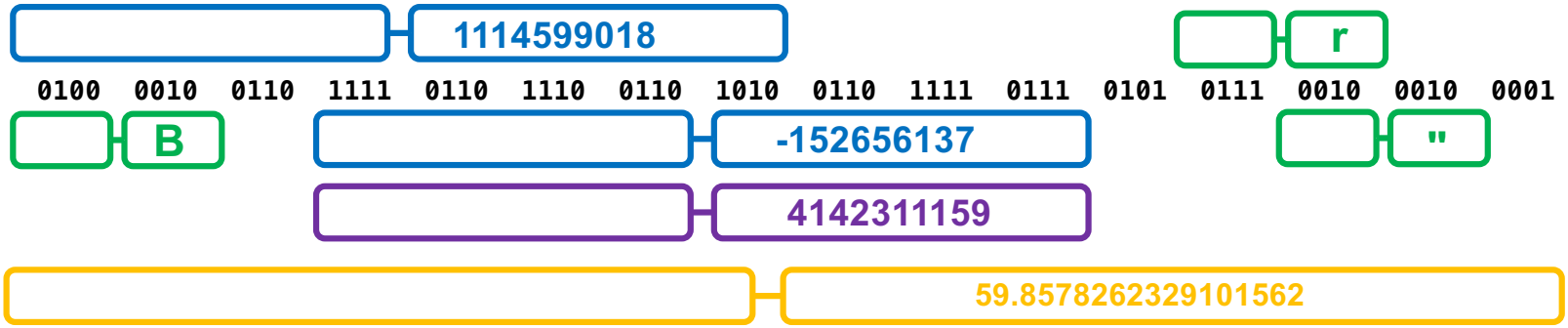
- Qui connait le type ? -> Défini lors de la déclaration de la variable

Représentation des nombres (et variables)

- Le type de la variable indique le nombre de bit à considérer
  Ex: **`char`** -> **8** bits, **`int`** -> **32** bit (dépends de la machine/OS), **`double`** -> **64** bit (dépends)
- Différentes *lunettes* pour lire le type demandé



Exemples



La même suite de '1' et '0' correspond à différentes valeurs, fonction du type de variable (=lunette)

Il est possible d'indiquer au compilateur de changer de lunettes en faisant un type cast

Ex. `printf("%d\n", (short)65535);`

demo C.1.RepresentationNombre

Où est le problème ?

Hint: représentation des nombres

```
// Compute Packet Checksum
unsigned char checksum_Compute(unsigned char command[])
{
  unsigned char checksum;
  for (int i=0; i<(COMMANDLENGTH-1); i++)
  {
    checksum += command[i];
  }
  return checksum;
}
```

Où est le problème ?

Hint: représentation des nombres

```
// Compute Packet Checksum
unsigned char checksum_Compute(unsigned char command[])
{
  unsigned char checksum; // max 255
  for (int i=0; i<(COMMANDLENGTH-1); i++)
  {
    checksum += command[i]; // checksum might/will overflow
  }
  return checksum;
}
```

Sélection/extraction

Ex. comment extraire le jour/mois/année d'un nombre représentant une date en suivant une convention donnée?

Comment extraire le mois de **18092025**

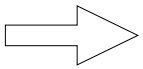
7 6 5 4 3 2 1 0 positions

Par convention les 2 chiffres représentant le mois se trouvent en position 4 et 5

18092025

00090000

Masquer pour sélectionner position 4 et 5



Décaler de 4 positions,

Note: décalage d'une position -> division par 10

09

09 -> septembre

Opérateurs *bit à bit*

&	**Bitwise AND**	`1 & 1 -> 1`
		`0 & 1 -> 0`
		`1 & 0 -> 0`
		`0 & 0 -> 0`

|	**Bitwise OR**	`1 | 1 -> 1`
		`0 | 1 -> 1`
		`1 | 0 -> 1`
		`0 | 0 -> 0`

^	**Bitwise XOR**	`1 ^ 1 -> 0`
		`0 ^ 1 -> 1`
		`1 ^ 0 -> 1`
		`0 ^ 0 -> 0`

~	**Bitwise complement**	`~1 -> 0` `~0 -> 1`

<<	**Shift left**	`4 >> 1  -> 2`
		`1 << 3  -> 8`
		`1 << 4  -> 16 (overflow)`

>>	**Shift right**	$0100_b >> 1 \rightarrow 0010_b$ $0001_b << 3 \rightarrow 1000_b$ $0001_b << 4 \rightarrow 1\ 0000_b$

Masking with &

- L'opérateur **&&** effectue l'opération AND **logic**

```
k = a && 1; // c=1 si a = 1 ou c=0 si a=0, car 0 && 1 est faux -> 0
```

- L'opérateur **&** effectue l'opération AND **bit à bit**

```
k = 12₁₀ & 10₁₀; // k = 8₁₀, car
                 //   1100b       C16        1210
                                  
// mask ->       //   1010b       A16        1010    &
                 //   ------------------------------
                 //   1000b       816         810
```

- La représentation hexadécimale **0xF** (ou **1111b**) permet de facilement sélectionner 4 bits à la fois, idem pour **0xFF** (8 bits) qui permet de sélectionner un **byte** ou **char** ou octet.
- Ex. 0x00125523 **(unsigned int 32 bits)**

  0x000000FF

  0x00000023

Fonctionne de la même manière pour :

|	*Bitwise OR*
^	*Bitwise exclusive OR*
~	*Bitwise complement*

Shifting with >>

- L'opérateur >> décale les bits vers la droite (<< vers la gauche)

```
k = 1100b >> 1; // k = 0110b décale de 1 bit vers la droite
k = 1100b >> 2; // k = 0011b décale de 2 bits vers la droite
```

Attention aux nombres négatifs! Le bit de signe peut (ou non) être étendu, cela dépend de l'implémentation 😕

- Ex.

```
0x00125523  (unsigned int 32 bits)
0x00FF0000 &
----------
0x00120000    Il faut maintenant décaler 0x00120000 de 16 bit vers la droite
0x00120000 >> 16
----------
  0x00000012
```

- En c

```
unsigned int pixel = 0x00125523;
unsigned int Red   = (pixel & 0x00FF0000)>> 16;
                                        // Δ précédence des opérations
```