# Architecture des ordinateurs - Codage et manipulation de l'information

Didier Teifreto

Université de Franche Comté

9 septembre 2015

# Objectif du cours

Connaissances utiles dans les domaines suivants :

1. Filière matérielle
   - Acheter un ordinateur
   - Conception de nouveaux processeurs - ordinateurs (rare)
   - Nouvelle version de processeurs - ordinateurs
   - Concevoir un système embarqué
2. Filière logicielle
   - Optimiser les performances logicielles
   - Concevoir un système embarqué

# Utilité du cours

- Comprendre le fonctionnement d'un système informatique,
- Optimiser les performances d'un système,
- Optimiser les performances d'un programme,
- Culture général concernant votre outil de base.

# Architecture des ordinateurs - Définition

- Architecture des bâtiments : Conception des structures, des parties d'un bâtiment - évaluer cout, durabilité... (génie civile)
- Architecture des ordinateurs (computer architecture) : Conception des circuits électroniques de l'ordinateur - cout, performance ... (génie électrique)

# Abstraction

- Centaines de millions de transistors - Difficile de travailler à ce niveaux
- Couches logiques et physiques nous aident à masquer la complexité (les détails)
- Plus nous irons au fond de l 'architecture, plus ce sera complexe.

# Abstraction logicielle - 1

Langage de haut niveaux

```
int somme(int x, int y, int z){
int t;
t = x+y+z;
return t;
```

Langage assembleur

```
somme:
iadd $t0, $zero, $zero
iadd $t0, $a0, $a1
iadd $v0, $t0, $a2
ijr $ra
```

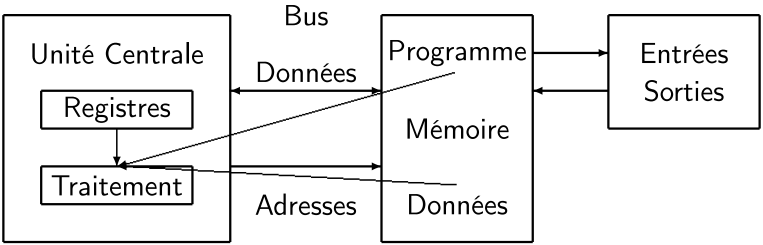
Code hexadécimal

- **0x**0040 0000 : **0x**2008 0000
- **0x**0040 0004 : **0x**0085 4020
- **0x**0040 0008 : **0x**0106 4020
- **0x**0040 000C : **0x**0106 1020

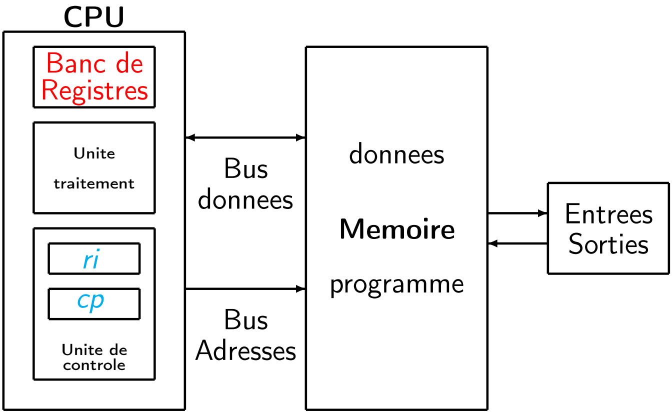
# Abstraction logicielle - 2

- Le code source de haut niveau est compilé pour une obtenir un code source de bas niveau (assembleur MIPS ici) : un programme nommé compilateur réalise ce travail.
- Le code source de bas niveau est assemblé en nombres positif compréhensible par l'ordinateur (code hexadécimal) : un programme nommé assembleur réalise ce travail.
- Le code hexadécimal peut être aussi interprété : un programme nommé machine virtuelle (Java) réalise ce travail.

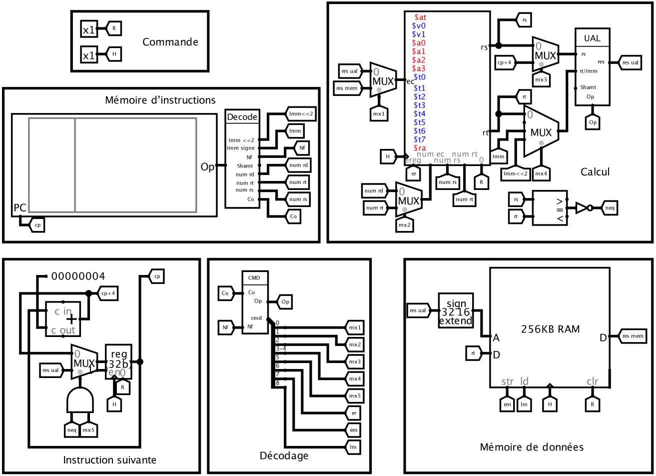
# Abstraction matérielle - 1



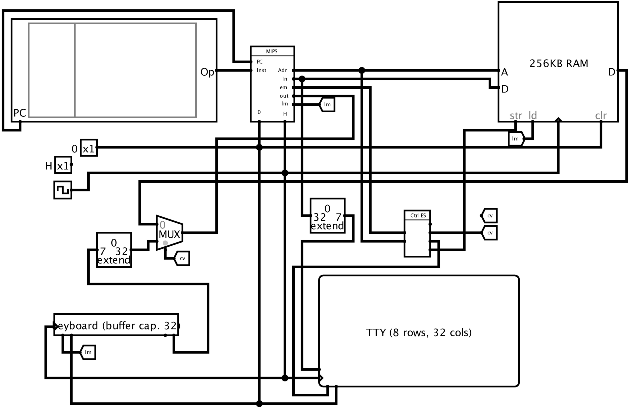
# Abstraction matérielle - 2



# Abstraction matérielle - 3



# Abstraction matérielle - 4



# Organisation du cours

1. Partie 1 : Codage et manipulation de l'information
   - Entiers signés et non signés,
   - Nombres réels standardisés,
   - Application en Java et implémentation de structure de calcul
2. Partie 2 : Programmation et Organisation MIPS
   - Programmation MIPS,
   - Application assembleur MIPS,
   - Organisation des processeurs MIPS et de la mémoire,
   - Systèmes d'exceptions et d'entrées/sorties,
   - Application assembleur MIPS et simulation de circuits
3. Partie 3 : Accélération des processeurs
   - Hiérarchie mémoire,
   - Notion de pipeline d'instructions,
   - Simulation cache et pipeline

# Divers

1. Volume horaire :
   - Cours 18 heures,
   - TD 18 heures,
   - TP 18 heures
   - Travail personnel 79 heures (6 heures par semaine)
2. Examen :
   - Interrogation Cours / TD coefficient 1/4,
   - Examen Cours/TD 1 heure 30, coefficient 1/4
   - Examen TP 1 heure 30 sur la partie codage de l'information et programmation MIPS 1/4
   - Projet sur les parties architecture MIPS et accélération, coefficient 1/4
3. Notions étudiées en cours, approfondies en TD et implémentées en TP.

# Planning

| Date | cours | TD | TP |
|---|---|---|---|
| 7 sep | 1 et 2 : Nb non signé/signé | 1 : non signé expression | |
| 14 sep | 3 : Nombres Flottants | 2 : Non signé Algo | 1 : Non signé |
| 21 sep | 4 : Instructions MIPS | 3 : Signé | 2 : Non signé |
| 28 sep | 5 : Instructions MIPS | 4 : Flottant | 3 : Signé |
| 5 oct | 6 : Architecture MIPS | 5 : Flottant | 4 : Flottant |
| 12 oct | 7 : Architecture MIPS | 6 : Codage Instruction | 5 : Assembleur |
| 19 oct | 8 : Partiel | 7 : Algo assembleur | 6 : Assembleur |
| 2 nov | 9 : Exceptions | 8 : Etude archi | 7 : Exam Nombres |
| 9 nov | 10 : Cache | 9 : Conception cmd | 8 : Archi MIPS v1 |
| 16 nov | 11 Fin pipeline | 10 Exceptions | 9 Exception |
| 23 nov | | 11 : Cache | 10 Cache |
| 30 nov | | 12 : Pipeline | 11 : Exam asm |
| 7 dec | 12 : Examen final | | 12 : Soutenance |
| 14 dec | | | |

# Philosophie et apprentissage

- Apprentissage demande investissement, pas juste attendre que le temps passe.
- Présentiel pour savoir ce que vous avez à apprendre à la maison.
- Étude du cours avant de venir en TD et avant le cours suivant.
- Exercice à **préparer/finir** avant les TD/TP.

Quatre facteurs d'importance égales pour réussir :

- Enseignant,
- Travail personnel,
- Amis,
- Conditions temps,lieux, économiques.

# Bibliographie

1. Architecture de l'ordinateur, **Nicolas P Carter Schaum's**,
2. Architecture des ordinateurs, **J Henessy, A Patterson**,
3. Organisation et conceptions des ordinateurs, **J Henessy, A Patterson**.

# Codage et manipulation l'information

*Le rapide élimine le lent, même si le rapide à tort* **W Kahan**

# Codage binaire

- Dériver du *I-Ching*
- Formaliser par **Gottfried Leibniz** en 1679

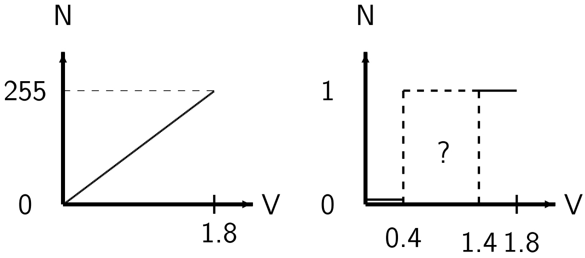
*Il n'est pas facile de donner ... l'idée de la création ex nihilo .... Maintenant, on peut dire que rien au monde ne peut mieux présenter et de démontrer ce pouvoir que l'origine des chiffres, tel qu'il est présenté ici à travers la présentation simple et sans fioritures de l'un et de zéro ou rien.*
**Lettre-Leibniz au duc de Brunswick sur les hexagrammes du I-Ching**

# Codage binaire

- Codage digital VRAI ou FAUX ou 0,1,
- Groupement de bits ayant différentes **interprétations**,

| *Taille de l'ensemble* | *Nom de l'ensemble* |
|---|---|
| 4 bits | Digit hexadécimal |
| 8 bits | Octet ou Byte |
| 16 bits | Demi-Mot ou HalfWord |
| 32 bits | Mot ou Word |
| 64 bits | Double Mot ou un Double Word |
| 128 bits | Quadruple mot ou Quad Word |

# Codage physique




Codage analogique Codage digital

- Codage analogique : Une grandeur physique pour coder un ensemble de valeurs. Sensible aux variations de la grandeur physique
- Codage digital ou numérique : Un grandeur physique pour coder deux valeurs. Insensible aux variations de la grandeur physique (dans une certaine mesure)

# Portes logiques

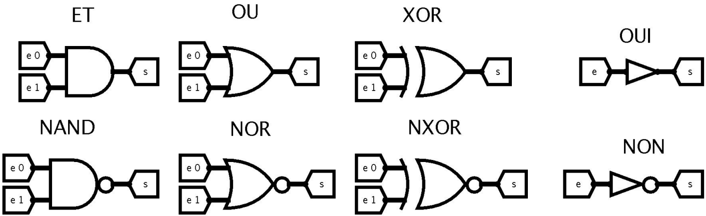



Figure 1 : Portes logiques

# Système complet NAND

$$\overline{a} = \overline{a.a} = \overline{a.1}, a.b = \overline{\overline{a.b}.1}, a + b = \overline{\overline{a + b}} = \overline{\overline{a.1}.\overline{b.1}}$$

- Deux barres de négation et théorème de De Morgan.
- Porte NON : une porte NAND
- Porte ET : deux portes NAND
- Porte OU deux entrées avec trois portes NAND.

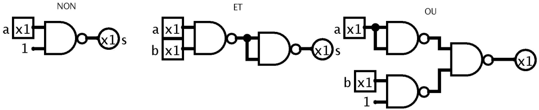



Figure 2 : Système complet

# Multiplexeur 1 bit

1. Sélection d'une information
2. Table de vérité (sélection 1 bit)

| $a$ | $b$ | $s$ | $c$ |
|---|---|---|---|
| 0 | 0/1 | 0 | 0 |
| 1 | 0/1 | 0 | 1 |
| 0/1 | 0 | 1 | 0 |
| 0/1 | 1 | 1 | 1 |

3. Équation

$$c \quad = \quad a.(b+\overline{b}).\overline{s} + (a+\overline{a}).b.s = a.\overline{s} + b.s$$

4. Schéma électrique

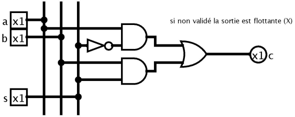



Figure 3 : Multiplexeur 1 bit

# Multiplexeur 1 bit - vue externe

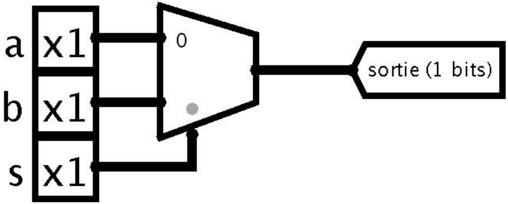



Figure 4 : Multiplexeur 1 bit vue externe

- Si s est codé sur $n$ bits, sélection d'une entrée par $2^n$.

# Décodeur

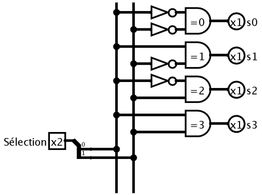
1. Table de vérité

| $a_1$ | $a_0$ | $s_3$ | $s_2$ | $s_1$ | $s0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

2. Équation

$$s_3 = a_1.a_0, s_2 = a_1.\overline{a_0}, s_1 = \overline{a_1}.a0, s_0 = \overline{a_1}.\overline{a_0}$$

3. Circuit



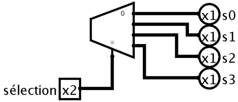




Figure 5 : Décodeur 2 bits

# Comparateur 1 bit

1. Table de vérité

| $a_1$ | $a_0$ | $a_1 < a_0$ | $a_1 > a_0$ | $a_1 = a_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |

2. Équation

$$a_1 < a_0 = \overline{a_1}.a_0, a_1 > a_0 = \overline{a_0}.a_1, a_1 = a_0 \quad = \quad \overline{a_0 \oplus a_1}$$

3. Circuit

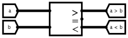



Figure 6 : Comparateur vue externe

# Entiers non signés $\in \mathbb{N}$

*Toute chose est nombre* **Pythagore**

*Signe : Ce qui permet de connaître ou de reconnaître, de deviner ou de prévoir quelque chose : Quand les hirondelles volent bas, c'est signe de pluie.* Larousse

# Polynôme non signé

Soit le nombre $d_{n-1}, d_{n-2}...., d_i, ...d_2, d_1, d_0$ avec $0 \leq d_i \leq b-1$

$$N = d_{n-1} \times b^{n-1} + ...d_i \times b^i + ... + d_1 \times b^1 + d_0 \times b^0$$

avec $i \subset [0, n-1]$ nommé le rang du chiffre

et $b^i$ nommé poids du chiffre de rang $i$.

$$\begin{aligned}
N_{max} &= (b-1) \times (b^{n-1} + b^{n-2} + ...b^i ... + 1) \\
&= b \times (b^{n-1} + b^{n-2} + ...b^i ... + b^1 + 1) \\
&\quad -1 \times (b^{n-1} + b^{n-2} + ...b^i ... + b^1 + 1) \\
&= b^n - 1 \\
N_{min} &= 0
\end{aligned}$$

Nombre de Valeurs $= b^n - 1 + 1 = b^n$

# Polynôme non signé binaire

- Les poids sont : 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 ...
- Le bit de poids fort est nommé MSB (Most Significant Bit),
- Le bit de poids faible est nommé LSB (Less Significant Bit),

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| poids | $2^{n-1}$ | $2^{n-2}$ | ... | $2^i$ | ... | $2^2$ | $2^1$ | $2^0$ |
| rang | $n-1$ | $n-2$ | ... | $i$ | ... | 2 | 1 | 0 |
| nom | MSB | | | | | | | LSB |

# Intervalles

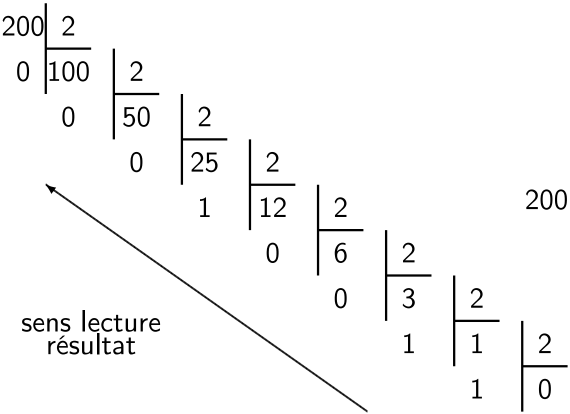
$$
\begin{aligned}
4\ bits\ N_{max} &= 2^4 - 1 = 16^1 - 1 = 15 \\
8\ bits\ N_{max} &= 2^8 - 1 = 16^2 - 1 = 255 \\
9\ bits\ N_{max} &= 2^9 - 1 = 511 \\
10\ bits\ N_{max} &= 2^{10} - 1 = 1023 = 1K - 1 \Rightarrow \color{red}{1k \Leftrightarrow 1024} \\
16\ bits\ N_{max} &= 2^{16} - 1 = 2^{10} \times 2^6 - 1 = 16^4 - 1 = 65\,535 \\
&= 64k - 1 \\
20\ bits\ N_{max} &= 2^{10} \times 2^{10} - 1 = 1\,048\,575 = 1M - 1 \\
& \Rightarrow \color{red}{1M \Leftrightarrow 1\,048\,576} \\
30\ bits\ N_{max} &= 2^{10} \times 2^{10} \times 2^{10} - 1 = 1\,073\,741\,823 = 1G - 1 \\
& \Rightarrow \color{red}{1G \Leftrightarrow 1\,073\,741\,823} \\
32\ bits\ N_{max} &= 2^{32} - 1 = 2^{30} \times 2^2 - 1 = 4\,294\,967\,295 = 4G - 1
\end{aligned}
$$

# changement de base

**Algorithme 1** divisions successives

**Variables**
    *nombre*, *base* : entier
**Début**
**Lire** (*nombre*, *base*)
**Répéter**
    **Afficher** ($nombre \mod base$)
    $nombre \leftarrow nombre \; / \; base$
**Jusqu'à** $Nombre = 0$
**Fin**

# Exemple de conversion binaire




$$
\begin{aligned}
200 &= \mathbf{0b}1100\,1000 & (1)\\
&= \color{red}{2^7} + \color{red}{2^6} + \color{red}{2^3} & (2)\\
&= 128 + 64 + 8 & (3)
\end{aligned}
$$

# Codage hexadécimal

- Codage machine : binaire exclusivement
- $16 = 2^4$, 1 digit est représenté sur 4 bits.
- Chiffre de 0 à 9,puis **0x**A,**0x**B,**0x**C,**0x**D,**0x**E,**0x**F,
- Conversion binaire $\Longleftrightarrow$ hexadécimale : regroupement d'ensemble de 4 bits.
- Notation binaire et hexadécimale : **0x**... et **0b**...
- Exemple :

$$\begin{aligned} 200 &= 128 + 64 + 8 = \mathbf{0b}1100\,1000 \\ &= 16 \times 12 + 8 = \mathbf{0x}C8 \end{aligned}$$

# Opérateurs arithmétiques

- **Addition 2 bits :** 0+0=0 ;0+1=01 ;1+1=10
- **Addition 3 bits :** 00+1=01 ;01+1=10 ;10+1=**0b11**
- **Addition de 2 x n bits :**
  - Propagation de retenue de droite à gauche.
  - Si $(n_1 + n_2) < 2^n - 1$ Résultat codé sur n bits
  - sinon Résultat codé sur n bit et la **retenue non signée**.
  - Addition à propagation de retenue

# Opérateurs arithmétiques - exemple

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 253 | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| +2 | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 255 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 253 | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| +3 | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 256 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- l'addition de deux nombres codés sur $n$ bits peut générer un résultat sur $n+1$ bits.
- La retenue sortante non signée ($R_{ns}$ (en rouge) est appelée aussi Carry flag.

# Addition de deux bits

1. Table de vérité

| $a$ | $b$ | $c$ | $r$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

2. Équations

$$c = a.b = \overline{\overline{a.b}} = \overline{\overline{(a.b)}.\overline{(a.b)}} \quad , \quad r = \overline{a}.b + a.\overline{b} = a \oplus b$$

3. Circuit électrique

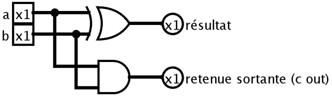



Figure 7 : Demi additionneur

# Additionneur complet

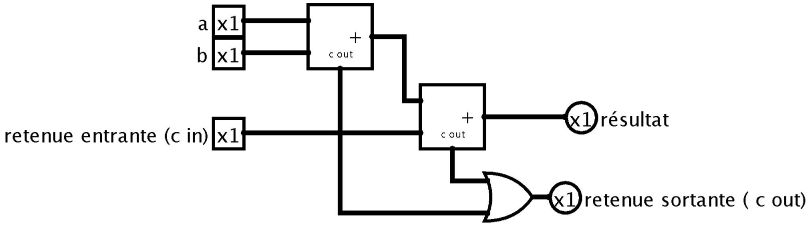



Figure 8 : Additionneur complet

- Deux demi-additionneurs
- Plus simple si nous écrivons la table de vérité puis les équations

# Propagation de retenue

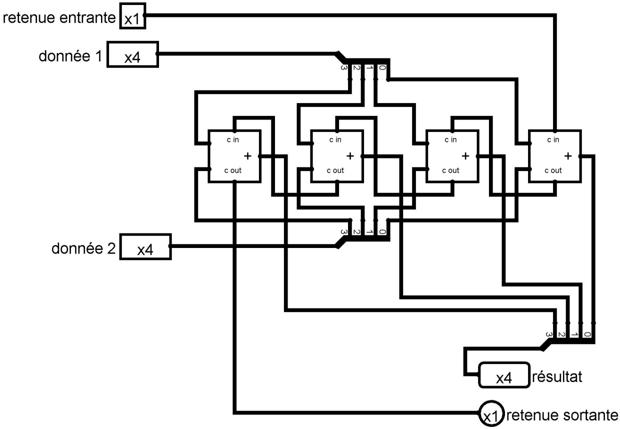



Figure 9 : Additionneur à propagation de retenue

# Opérateurs bit à bit

1. $\sim$ Non bit à bit : $\sim (x)$
2. $|$ Ou bit à bit : $b \mid 0 = b$, $b \mid 1 = 1$ ; force à 1
3. $\&$ Et bit à bit : $b \,\&\, 1 = b$, $b \,\&\, 0 = 0$ ; force à 0
4. ^ Ou exclusif bit à bit : $b \,\hat{}\, 0 = b$, $b \,\hat{}\, 1 = \sim (b)$ ; inverse
5. $<<$ Décalage logique à droite : $x >>> n = \frac{x}{2^n}$
6. $>>>$ Décalage logique à gauche : $x << n = x2^n$

- Les opérateurs bit à bit s'appliquent à des ensembles de bits
- Pour les décalages $1 \leq n \leq 31$

# Opérateurs bit à bit - Test valeur bit(s)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| & 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

```
if ((x & 2)==0)
//bit de rang 1 = 0
else
//bit de rang 1 = 1
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 255 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| & 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

```
if ((x & 2)!=0)
//if ((x & 2)==2)
//bit de rang 1 = 1
else
//bit de rang 1 = 0
```

- Le ou les bits à 1 dans le masque sont conservés dans le résultat.

# Opérateurs bit à bit - Utilisation du test valeur bit(s) - 2

Test du bits de poids faible

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| & 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| res | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

res = 0 ou res = **0x1**

```
1   res = (x & 1)
```

Test du bits de poids fort

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| & **0x**80 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| res | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

res = 0 ou res = **0x**80

```
1   res=(x & 0x80)
```

# Opérateurs bit à bit - Taille donnée inconnue

mettre à 0 le bit de poids faible (utilisation de l'opérateur $\sim$ .

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 253 | 0 | ... | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | |
| & ??? | 1 | ... | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 res=(x & ? ) |
| res | 0 | ... | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 253 | 0 | ... | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | |
| & ( $\sim$ 1) | 1 | ... | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 1 | res=(x & ~1 ) |
| res | 0 | ... | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | |

# Opérateurs bit à bit - Mise à 1 bit(s)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| $\mid$ 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 255 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

$1 \quad x = (x \mid 2)$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 255 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $\mid$ 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 255 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

- Le ou les bits à 1 dans le masque sont mis à 1 dans le résultat.

# Opérateurs bit à bit - Mise à 0 bit(s)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| & 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |

```
x = (x & 0xFD)
// x = (x & ~2)
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 255 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| & 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |

- Le ou les bits à 0 dans le masque sont mis à 0 dans le résultat.

# Opérateurs bit à bit - Inversion

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| ^ 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 255 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

```
1  x = (x ^ 2)
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 255 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| ^ 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 253 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |

- Le ou les bits à 1 dans le masque sont inversés dans le résultat.

# Opérateurs bit à bit - Décalages logiques

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $125$ | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| $125 >>> 1 = 62$ | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $125$ | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| $125 << 1 = 250$ | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |

- Les bits sortants sont perdus (en rouge)
- Les bits entrants sont à 0 (en jaune)
- La division donne toujours un résultat entier.

# Processus itératif et opérateurs bit à bit

Comptage des bits à 1 d'une donnée 32 bits

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 125 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 125 & 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| res=0 +1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 125 >>> 1=62 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 62 & 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| res=1+0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 62 >>> 1=31 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 31 & 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| res=1 +1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

- Tous les bits passent dans le bits de poids faible,
- Il suffit d'additionner les bits 0 pour obtenir le nombre de bit à 1, jusqu'à ce que la donnée soit à 0,
- Processus lent car itératif.

# Implémentation comptage itératif

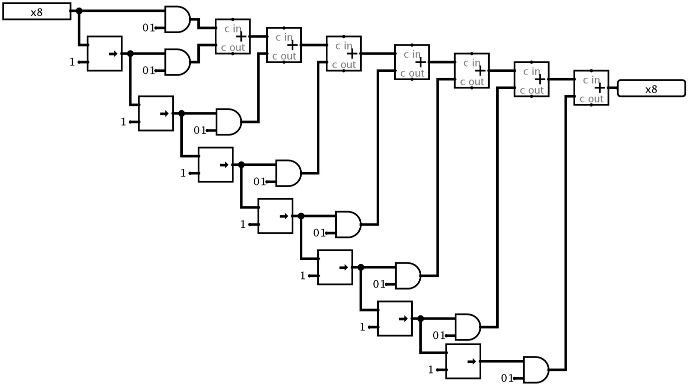



Figure 10 : Comptage itératif 8 bits

# Processus parallèle et opérateurs bit à bit

Comptage des bits à 1 d'une donnée 32 bits

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 125 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 55 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 125 & **0x55** | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| + (125 >>> 1) & **0x55** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| = 105 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 105 & **0x33** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| + (105 >>> 2) & **0x33** | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| = 51 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 51 & **0xF** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| + (105 >>> 4) & **0xF** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| = 6 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

- Addition de deux bits côte à côte, puis 4 bits, puis 8 bits.
- Processus plus rapide si taille donnée grande.

# Implémentation comptage parallèle

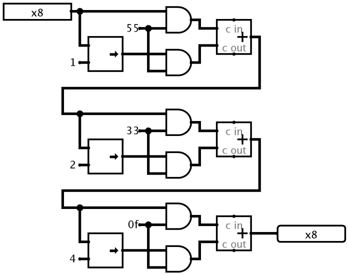



Figure 11 : Comptage rapide 8 bits

# Utilisation : Données, adresses et instructions

1. Les adresses des instructions et des données sont des entiers non signés. L'arithmétique et les opérateurs bit à bit permettent de les manipuler.
2. Les instructions sont codées dans des nombres non signés. Les opérateurs bit à bit permettent de décoder les instructions.
3. Les données de l'utilisateur sont codées sur 8, 16 ou 32 bits. Nous implémenterons uniquement les données 32 bits.

# Utilisation : Codes ascii

- Codage des caractères : octet contient le code ascii d'un caractère
- ascii : *American Standard Code for Information Interchange*

| dec | hex | Char | dec | hex | char | dec | hex | char |
|---|---|---|---|---|---|---|---|---|
| 48 | **0x30** | '0' | 65 | **0x41** | 'A' | 97 | **0x61** | 'a' |
| 49 | **0x31** | '1' | 66 | **0x42** | 'B' | 98 | **0x62** | 'b' |
| 50 | **0x32** | '2' | 67 | **0x43** | 'C' | 99 | **0x63** | 'c' |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 53 | **0x35** | '5' | 70 | **0x46** | 'F' | 102 | **0x66** | 'f' |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 59 | **0x39** | '9' | 74 | **0x4A** | 'J' | 106 | **0x6A** | 'j' |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

# Manipulation des codes ascii - 1

**Algorithme 2** Conversion d'un caractère majuscule en minuscule

```
Variables
    c : caractère
Début
c ← 'F' //Lettre majuscule
c ←(caractère)((entier)c + 0x20) //ou bien (entier)c | ('a' −' A')
Afficher (c) // Affichera le caractère f
Fin
```

# Manipulation des codes ascii - 2

**Algorithme 3** Conversion d'un nombre non signé en un caractère

```
Variables
    i : entier
    c : caractère
Début
i ← 9
c ← (caractère)(i+'0')
Afficher (c) // Afficher le caractère 9
i ← 10
c ← (caractère)(i - 0xA +'A')
Afficher (c) // Afficher le caractère A
Fin
```

# Entiers signés $\in \mathbb{Z}$

*Signe : Marque distinctive faite sur quelque chose : Marquer d'un signe les arbres à abattre.* **Larousse**

# Principe du codage binaire

1. **Valeur absolue et signe :** Bit de poids fort (MSB=1) si nombre négatif. Pratique pour multiplication et division.
2. **Complément à 1 :** Inversion des bits. Codage générant une erreur de 1.
3. **Complément à 2 :** Correction du complément à 1. Pratique pour addition et soustraction.
4. **Codage avec excédent :** Une constante positive est ajoutée pour obtenir le nombre. Ceci est utilisé pour les nombres réels.

# Polynôme binaire

$$\begin{array}{rcl} & & \pm d_{n-1}, d_{n-2}...., d_i, ...d_2, d_1, d_0 \quad \text{avec } 0 \leq d_i \leq b-1 \\ N & = & -d_{n-1} \times 2^{n-1} + ...d_i \times 2^i + ... + d_1 \times 2^1 + d_0 \times 2^0 \\ & & \text{le bit de rang } n-1 \text{ est le bit de signe} \\ N_{max} & = & 2^{n-1} - 1 \\ N_{min} & = & -2^{n-1} \end{array}$$

1. Un nombres signés est formatés : 8,16,32 bits.

# Intervalles

$$
\begin{array}{rl}
8\ bits & Nmax = 2^7 - 1 = 127 \\
 & Nmin = -2^8 = -128 \\
16\ bits & Nmax = 2^{15} - 1 = 32k - 1 \\
 & Nmax = -2^{15} = -32k \\
32\ bits & Nmax = 2^{32} - 1 = 2G - 1 \\
 & Nmin = -2^{32} = -2G
\end{array}
$$

- Soit le nombre 8 bits **0b**1111 1111,
- Interprétation non signée : 255
- Interprétation signée : -1

# Opérateurs arithmétiques :inversion de signe

- Technique du complément à 2
- Complément à 1 (Non bit à bit) + 1

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 100 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| $\sim$ (100) | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| $\sim$ (100)+1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

- $\sim$ (100) = -128+16+8+2+1= -101
- $\sim$ (100)+1 = -128+16+8+4= -100

# Opérateurs arithmétiques : addition

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| -100 | | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| +(-1) | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| -101 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |

- Dans l'exemple précédent, retenue positionnée pour un résultat correct.
- La retenue non signée ne fonctionne pas.
- La retenue signée est nommée aussi Overflow

# Opérateurs arithmétiques : soustraction

$$\begin{aligned} a - b &= a + (-b) \\ &= a + \sim (b) + 1 \end{aligned}$$

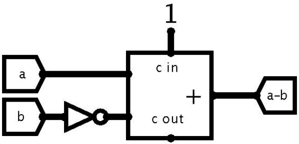



Figure 12 : Soustraction

- L'overflow n'est pas généré ici

# Opérateurs de décalage arithmétique à droite.

- Le décalage à droite est une division par deux.
- Pas de changement de signe

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $252 = -4$ | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| $252 >>> 1 = 126$ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| $-4 >> 1 = -2$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

# Réels $\in \mathbb{R}$

*Nombre flottant : nombre mis sous forme du produit d'un nombre décimal (appelé cofacteur, ou mantisse) par une puissance de 10 dont l'exposant est un entier.*

*Nombre flottant normalisé : nombre flottant tel que le cofacteur doit être en valeur absolue compris entre 1 et 10, sans être égal à 10. (Le nombre flottant* $42,3710^{-3}$ *est égal au nombre flottant normalisé* $4,23710^{-2}$.

# Polynôme virgule flottante

- Nombre représenté sous la forme $1, \textit{mantisse} \times 2^{exposant}$
- Nombre minimal 1,0
- Nombre de chiffres significatifs maximum
- Exemple $1{,}75 = \mathbf{0b}1{,}1100\,0000... \times 2^0$
- Exemple $3{,}00 = \mathbf{0b}1{,}100\,0000... \times 2^1$
- Impossible de représenter 0.0

$$
\begin{aligned}
\textit{mantisse} \quad & d_{n-1}, d_{n-2}...., d_i, ...d_2, d_1, d_0 \quad \text{avec } 0 \leq d_i 1 \\
N &= (1 + d_{n-1}2^{-1} + ... + d_i 2^{n-1} + ....)2^{exposant} \\
&= (1 + d_{n-1}\frac{1}{2} + d_{n-2}\frac{1}{4} + ... + d_i \frac{1}{2(n-i)} + ...)2^{exposant} \\
N_{max} &\approx 2 \times 2^{exposant} \\
N_{min} &= 1, 0
\end{aligned}
$$

# Conversion binaire de la partie fractionnaire

**Algorithme 4** Multiplications successives

**Variables**
    *nombre*, *base* : entier
**Début**
**Lire** $(nombre, base)$
**Tant Que** $nombre \neq 0$ **Faire**
    **Afficher** $(\leftarrow \lfloor nombre \times base \rfloor)$
    $nombre \leftarrow \{nombre \times base\}$
**Fin Tant Que**
**Fin**

# Exemple de conversion binaire

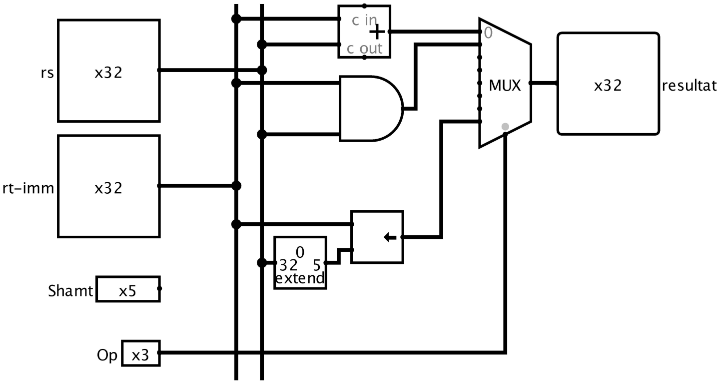
$$
\begin{array}{lll}
0.44 & 0.44 \times 2 = 0.88 & 0 \\
0.88 & 0.88 \times 2 = 1.76 & 1 \\
0.76 & 0.76 \times 2 = 1.52 & 1 \\
0.52 & 0.52 \times 2 = 1.04 & 1 \\
0.04 & 0.04 \times 2 = 0.08 & 0 \\
0.08 & 0.08 \times 2 = 0.16 & 0 \\
0.16 & 0.16 \times 2 = 0.32 & 0 \\
0.32 & 0.64 \times 2 = 0.64 & 0
\end{array}
$$

- $0,44 = 0b0,011100... = 0,4375...$
- Nombre infini de chiffre en binaire

# Unité arithmétique et logique UAL (ALU) - 1

- Circuit de calcul du résultat.
- Une opération parmi un ensemble possible
- Signal Op permet de sélectionner une opération
- Signal A et B non interchangeable.

# Unité arithmétique et logique UAL (ALU) - 2



# Unité arithmétique et logique UAL (ALU) - 3

| Op | Opération |
|---|---|
| 0 | A + B |
| 1 | A & B |
| 2 | A ^ B |
| ... | ... |
| 7 | A >>> B |
| ... | ... |

# Format IEEE 754

| Signe mantisse | Exposant | Mantisse |
|---|---|---|
| MSB | | LSB |

Format simple précision (32 bits)

| Signe mantisse | Exposant | Mantisse |
|---|---|---|
| 1 bit | 8 bits | 23 bits |

Format double précision (64 bits)

| Signe mantisse | Exposant | Mantisse |
|---|---|---|
| 1 bit | 11 bits | 52 bits |

# Spécification IEEE 754

| **Précision** | **simple** | **double** |
|---|---|---|
| TAILLE DE LA DONNEE | 32 bits | 64 bits |
| TAILLE SIGNE MANTISSE | 1 bit | 1 bit |
| TAILLE MANTISSE | 23 bits | 52 bits |
| TAILLE EXPOSANT | 8 bits | 11 bits |
| EXCEDENT | 127 | 1023 |
| EXP MINIMUM normalisé | -126 | -1022 |
| EXP MAXIMUM normalisé | +127 | +1023 |
| EXP MINIMUM Dénormalisé | -127 | -1023 |
| EXP MAXIMUM Dénormalisé | +128 | +1024 |
| $\mid$ VALEUR MINIMALE $\mid$ | $1 \times 2^{-149}$ | $1 \times 2^{-1023-52}$ |
| $\mid$ VALEUR MAXIMALE $\mid$ | 0b1,11.. $\times 2^{127}$ | 0b1,111...$2^{1023}$ |

# Codage de l'excédent

- exposant codé = exposant réel + excédent
- exemple pour le format simple précision

| Exposant codé | Expression de l'exposant réel | Exposant réel |
|---|---|---|
| 0 | $0 = 0 - 127$ | -127 |
| 1 | $1 = 1 - 127$ | -126 |
| $x$ | $= x - 127$ | |
| 254 | $254 = 254 - 127$ | +127 |
| 255 | $255 = 255 - 127$ | +128 |

# Exemple 1

$$\begin{aligned}
1 &= 1 \times 2^0 \\
\text{exposant} &= 127 + 0 = \mathbf{0b}0111\,1111 \\
\text{mantisse normalisée} &= \mathbf{0b}1{,}0 \\
\text{mantisse normalisée} &= \mathbf{0b}0\ldots \\
\text{mémoire} &= \mathbf{0b}0011\,1111\,1000\ldots \\
&= \mathbf{0x}3F80\,0000
\end{aligned}$$

| Signe mantisse | Exposant | Mantisse |
|---|---|---|
| 1 bit | 8 bits | 23 bits |
| 0 | 011 1111 1 | 000 0000... |

# Exemple 2

$$
\begin{aligned}
-2,5 &= 10,1 \times 2^0 = \mathbf{0b}1{,}01 \times 2^1 = \mathbf{0x}1{,}4 \times 2^1 \\
\text{exposant} &= 127 + 1 = \mathbf{0b}1000\,0000 \\
\text{mantisse normalisée} &= \mathbf{0b}1{,}01 \\
\text{mantisse normalisée} &= \mathbf{0b}01\ldots \\
\text{mémoire} &= \mathbf{0b}1100\,0000\,0010\,0000\ldots. \\
&= \mathbf{0x}C020\,0000
\end{aligned}
$$

| Signe mantisse | Exposant | Mantisse |
|---|---|---|
| 1 bit | 8 bits | 23 bits |
| 1 | 100 0000 0 | 010 0000... |

# Affichage hexadécimal nombres normalisés

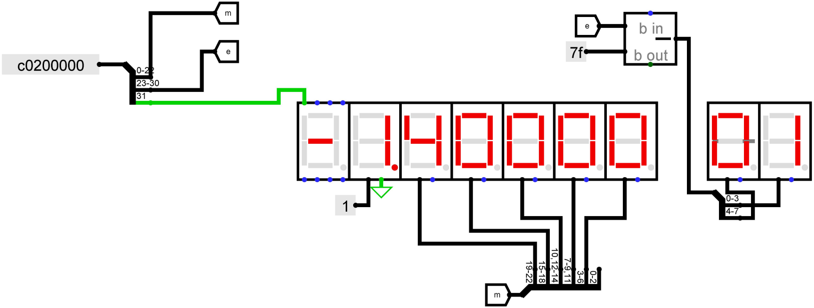



Figure 13 : Affichage réel normalisé

# Nombres dénormalisés

1. Zéro est codé **0b**0,0000 ...000 $\times 2^{-127}$ = **0x**0000 0000
2. $\infty$ est codé **0b**1,0000 ...000$\times 2^{128}$ = **0x**7F80 0000
3. $-\infty$ est codé -**0b**1,0000 ...000 $\times 2^{128}$**0x**FF80 0000
4. *NaN* est codé **0b**1,1000 ...000$\times 2^{128}$ = **0x**7FC0 0000
5. Les nombres dénormalisés < **0b**1,0000 ... $\times 2^{-126}$
   - Pas de 1 à gauche de la virgule
   - Exposant codé -127
   - **0b**0,..1.. $2^{-126}$ avec mantisse différente de zéro.
   - Valeur la plus petite = $2^{-23} \times 2^{-126} = 2^{-149}$

# Addition de deux nombres normalisés

$$
\begin{aligned}
1,0 &= \mathbf{0b}1 \times 2^0 = \mathbf{0b}0{,}1 \times 2^1 \\
2,5 &= \mathbf{0b}1{,}01 \times 2^1 \\
1,0 + 2,5 &= \mathbf{0b}0{,}1 + \mathbf{0b}1{,}01 \times 2^1 \\
&= \mathbf{0b}1{,}11 \times 2^1 = 1,75 \times 2^1 \\
&= 3,5
\end{aligned}
$$

- Rechercher l'exposant le plus grand
- Aligner les mantisses sur l'exposant le plus grand
- Additionner les mantisses

# Addition de deux nombres normalisés algorithme

**Algorithme 5** Addition deux réels $> 0$ format simple précision

$N1, N2, N$ : réel 32 bits,
$m, m_1, m_2$ : entier 24 bits, $e, e_1, e_2$ : entier 8 bits
$m_1, m_2 \leftarrow$ Mantisse de $N_1, N_2$ ,$e_1, e_2 \leftarrow$ Exposant de $N_1, N_2$
**Si** $e_1 < e_2$ **Alors**
$m_1 \leftarrow m_1 >>> (e_2 - e_1)$, $e \leftarrow e_2$
**Sinon**
$m_2 \leftarrow m_2 >>> (e_1 - e_2)$, $e \leftarrow e_1$
**FinSi**
$m \leftarrow m_1 + m_2$ //Addition $2 \times 24$ bits
**Si** $c_{out} = 1$ **Alors**
$e \leftarrow e + 1, m \leftarrow (m >>> 1) \mid$ **0x**800000 //avec $c_{out}$
**FinSi**
Coder $m$ et $e$ dans $N$

# Addition de deux nombres réels >0 - circuit simplifié

- Beaucoup plus complexe qu'une addition sur des nombres entiers
- Plus lent

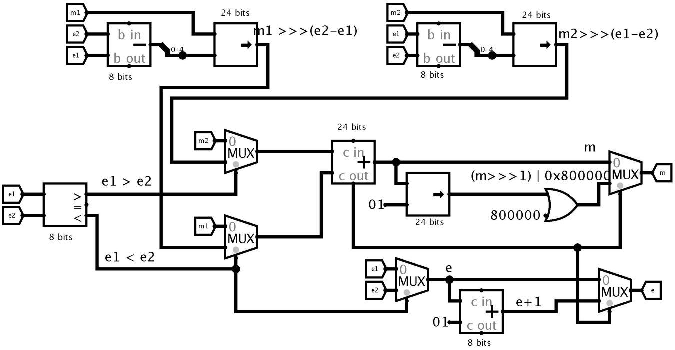



Figure 14 : Addition réel normalisé positifs

# Résumé

- Non signé : **0x**FFFF FFFF = 4G-1
- Signé **0x**FFFF FFFF = -1
- Réel IEEE 754 **0x**FFFF FFFF = NaN

- Non signé **0x**C000 0000 = 3G
- Signé **0x**C000 0000 = -1G
- Réel IEEE 754 **0x**C000 0000 = 2,0

- L'utilisateur connait le type de la valeur codée.
- Exception : 0 représente toujours la valeur 0