Présentation du module + Représentation des données dans un système à microprocesseur

Présentation du module

- Coefficient du module : 2,5

- Cours -TDs => 11 séances (16h30)
  - Représentation des données et opérations arithmétiques de base
  - Architecture d'un système à microprocesseur
  - Evolution des microprocesseurs
  - Programmation bas niveau: assembleur
  - Les microcontrôleurs: structure et interfaçage

- TDAO => 6 séances (9h)
  - Programmation des cartes microcontrôleurs

- Evaluations
  - Contrôle
  - Examen
  - TDAOs notés

Présentation du module

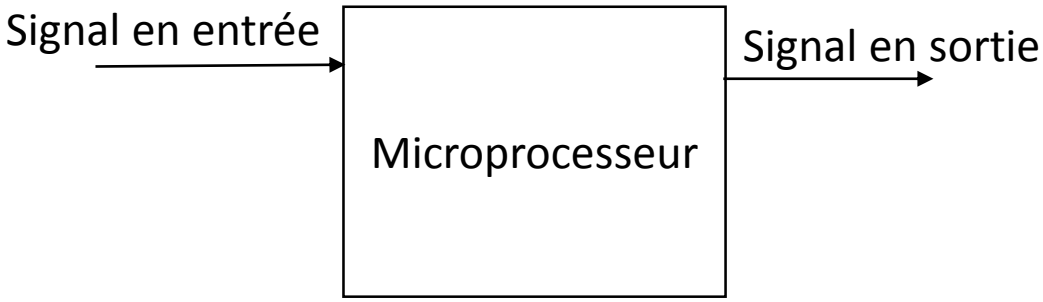
- Pourquoi on étudie l'architecture des systèmes à microprocesseurs?
  - Permettre aux programmeurs
    - Prendre en compte les coûts des opérations quand ils font leurs choix de programmation
    - D'écrire des programmes qui sont:
      - Plus rapides
      - Plus petits
    - Un débogage plus facile
    - Comprendre entre autres
      - Ce qui se passe dans la machine lors de l'exécution d'un programme
      - Où les instructions et les variables d'un programme sont stockées
      - Comment le processeur fait pour appeler une fonction et retourner au programme principal
      - Comment le processeur fait pour communiquer avec les périphériques

Représentation des données

- Le système binaire
  - Un microprocesseur est un système électronique numérique

➔les signaux ont deux valeurs possibles: (bas: **0**) ou (haut:**1**)

- Ces deux états correspondent à des tensions (liés aux transistors: états passant ou bloqué)



- Comment savoir si le signal en entrée correspond à un 1 ou 0 en se basant sur sa tension?

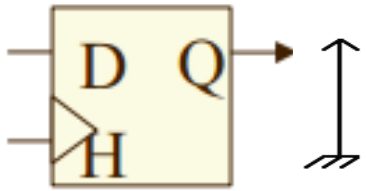
- Quelle tension générer en sortie pour avoir un signal de sortie à 1 ou 0?

Représentation des données

- Le système binaire

  Exemple: pour une bascule D alimentée à 5V:

  - une sortie de la bascule=1 correspond à une tension de 5V
  - une sortie de la bascule=0 correspond à une tension de 0V

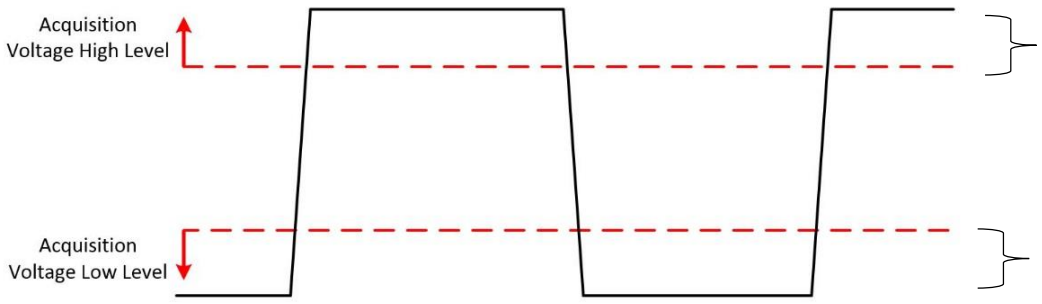


U= tension de sortie de la bascule

- Le problème est que la tension peut avoir n'importe quelle valeur entre 5 et 0V (1V, 4.5V, etc.)

  →Comment savoir si un signal correspond à 1 ou 0?

  Il faut fixer une marge de tension qui correspond à l'état 0 et une marge qui correspond à l'état 1



Marge de tension correspondante à l'état 1

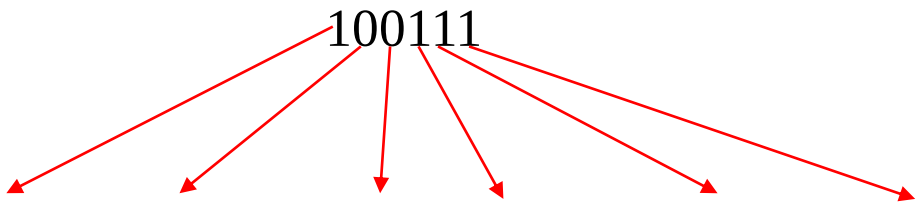
Marge de tension correspondante à l'état 0

Si le signal se trouve en dehors des deux marges, alors il est en état transitoire entre 0 et 1

Les marges sont différentes pour le TTL et le CMOS, cette information est disponibles dans les feuilles techniques du système.

Représentation des données

- Représentation binaire d'un nombre: suite de bits
  - Représentation par un ensemble de 0 et de 1
  - Conversion binaire → décimale
    - Chaque chiffre représente une puissance de 2



$(1x2^5) + (0x2^4) + (0x2^3) + (1x2^2) + (1x2^1) + (1x2^0)$

= 1x32 + 0x16 + 0x8 + 1x4 + 1x2 + 1x1

= 32 + 0 + 0 + 4 + 2 + 1

= 39

Représentation des données

- Conversion binaire → décimale
  - Nombres particuliers
    - Les puissances de 2
      - $(10)_2=(2^1)_{10}=(2)_{10}$

      On fait 2 nombre des zéros
      - $(100000)_2=(2^5)_{10}=(32)_{10}$
    - Les suites de 1
      - $(111)_2= (2^3 -1)_{10} = (7)_{10}$

      On fait 2 nombre des 1 -1

Représentation des données

- Conversion décimale → binaire

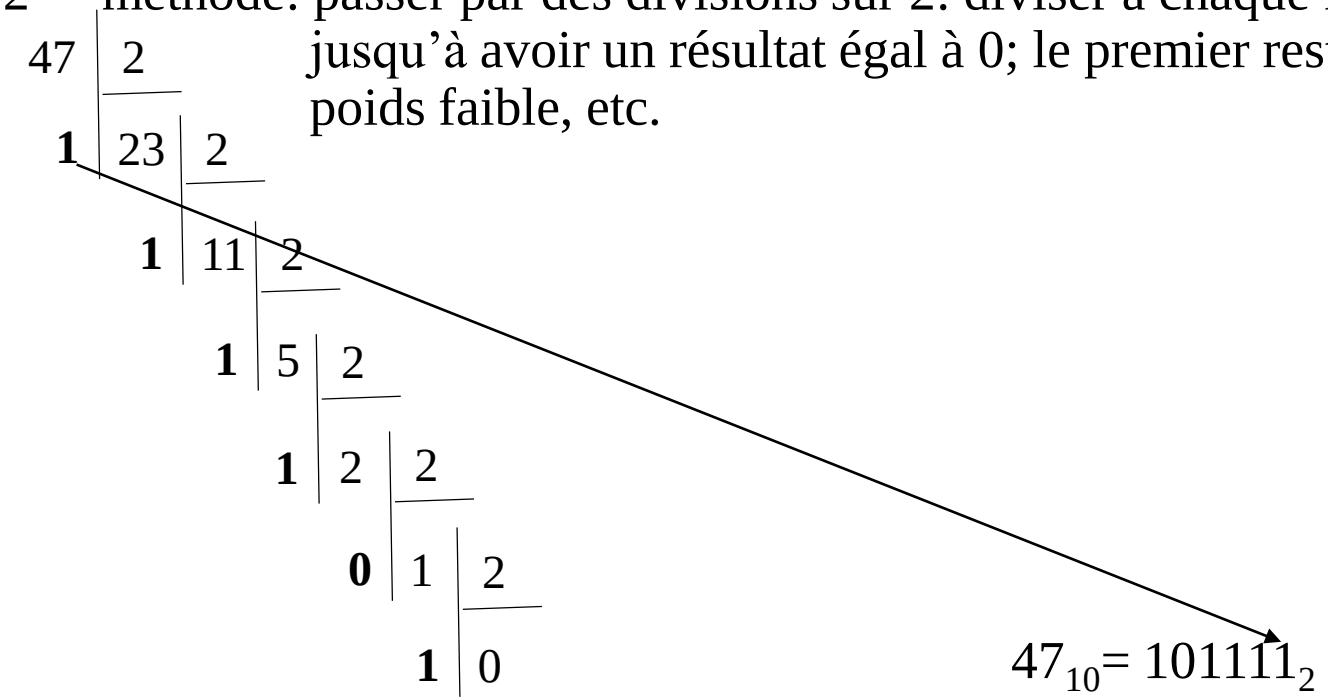
  - 1ère méthode: méthode directe: décomposer en une somme de puissances de 2

    $47 = 32+8+4+2+1$

    $= (1x2^5) + (0x2^4) + (1x2^3) + (1x2^2) + (1x2^1) + (1x2^0)$

    $= 101111$

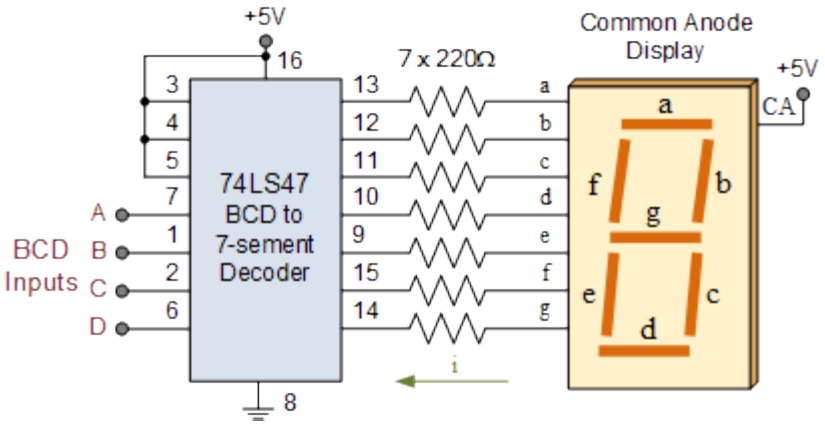
  - 2ème méthode: passer par des divisions sur 2: diviser à chaque fois le reste de la division précédente jusqu'à avoir un résultat égal à 0; le premier reste de la division correspond au bit de poids faible, etc.



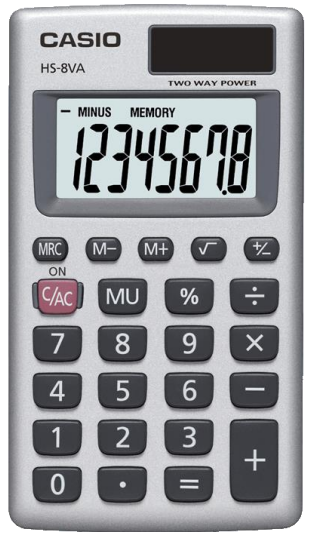
Représentation des données

- BCD: Binary Coded Decimal
  - Il s'agit d'un nombre décimal où chaque chiffre est codé sur 4 bits

Décimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	0001 0000

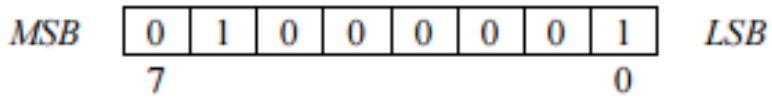


- Ainsi un mot de 16 bits permet de représenter des nombres de 0 à 9999 en BCD
- Utilisé dans les anciens processeurs mais rarement dans les processeurs modernes
- Plus utilisé dans les circuits électroniques nécessitant des affichages des chiffres décimaux (afficheur 7 segments, stockage de la date et du temps dans le bios, horloge, etc.)
- La conversion en un caractère pour l'affichage est très simple à implémenter c'est pour cette raison, que ce codage est utilisé dans la plupart des calculatrices
- Le codage BCD occupe plus d'espace que le codage binaire pur



Représentation des données

- Groupe de bits:
  - 4bits = 1 quartet (nibble en anglais, apparu avec les premiers ordinateurs qui utilisaient des groupes de 4 bits)
  - 8bits= 1 octet (1 byte en anglais)



MSB: Most Significant Bit=bit de poids fort

LSB: Least Significant Bit=bit de poids faible

  - 16bits= 1 doublet
  - Selon l'architecture du processeur, on peut trouver d'autres groupements tels que 24, 32 et 64 bits

Représentation des données

- Multiples de l'octet

Nom	Symbole	Valeur dans l'usage traditionnel
kilooctet	ko	2^{10}
mégaoctet	Mo	2^{20}
gigaoctet	Go	2^{30}
téraoctet	To	2^{40}
pétaoctet	Po	2^{50}
exaoctet	Eo	2^{60}
zettaoctet	Zo	2^{70}
yottaoctet	Yo	2^{80}

Représentation des données

- D'autres représentations des données
  - L'hexadécimal
    - Un chiffre hexadécimal a 16 valeurs possibles: 0 à 9, A, B, C, D, E, F
    - Conversion binaire → hexadécimale
      - $(10001111)_2 = (8F)_{16}$ (8 F) → Pour convertir du binaire à l'hexadécimal, on fait des groupements de 4 bits de droite à gauche
      - $(101110)_2 = (2E)_{16}$ (2 E)
    - Conversion hexadécimale → binaire

      $(2E)_{16}$

      0010 1110

Représentation des données

- D'autres représentations des données
  - L'hexadécimal
    - Conversion hexadécimale → décimale

      $(2F)_{16} = 2x16^1 + Fx\ 16^0 = 2x16 + 15 = (47)_{10}$

    - Conversion décimale → hexadécimale

      $(47)_{10} = 2x16^1 + 15x\ 16^0 = (2F)_{16}$

47	16	
F	2	16
	2	0

Représentation des données

- D'autres représentations des données

  - L'octal
    - Conversion binaire → octale
      - $(10001111)_2 = (217)_8$, on fait des groupements de 3 bits de droite à gauche

        2 1 7

    - Conversion octale → binaire
      - 56= 101110, on écrit chaque chiffre sur 3 bits

        101 110

    - La conversion octale/décimale suit le même principe des conversions binaire/décimale et hexadécimale/décimale

Représentation des données

- Utilisation des différentes représentations dans les langages C/C++, Java

  - Voici quatre représentations différentes de la même valeur 42 (en décimal)

```
int i;

i=42; /* décimal */

i=0b101010; /*binaire*/

i=052; /* octal */

i=0x2a; /*hexadécimal */
```

Formats de printf/scanf correspondants:
%d: décimal
%x: hexadecimal
%o: octal
Pas de format direct pour le binaire

Représentation des données

- Exercice

  - Donner le résultat affiché par le programme suivant:

```
#include<stdio.h>
void main(){

int a=50;
int b=0b100101;

printf (« %o %o\n », a,b);
printf (« %x %x\n », a,b);

}
```

Représentation des données

- Solution

  - Donner le résultat affiché par le programme suivant:

```
#include<stdio.h>
void main(){

int a=50;
int b=0b100101;

printf (« %o %o\n », a,b);
printf (« %x %x\n », a,b);

}
```

```
62 45
32 25
```

Opérations arithmétiques

- Addition et soustraction

Opération	Décimal	Binaire	Octal	Hexadécimal
Addition	37 + 31 = 68	1 1 1 1 1 : 10 0101 + 01 1111 = 1000100	1 : 45 + 37 = 104	1 : 25 + 1F = 44
Soustraction	37 - 31 = 06	1 1 1 1 1 : 10 0101 - 01 1111 = 000110	1 : 45 - 1 : 37 = 06	1 : 25 - 1 : 1F = 06

Opérations arithmétiques

- Multiplication et division

	Décimal	Binaire	Octal	Hexadécimal
Multiplication	1 2 37 x 3 111	100101 x 11 100101 100101 1101111	1 1 45 x 3 157	25 x 3 6F
Division	37 | 3 -3 | 12 07	100101 | 11 -11 | 1100 11 -11 00 -00 01	45 | 3 -3 | 14 15	25 | 3 C

Opérations arithmétiques

- Exercice
  - Donner le résultat affiché par le programme suivant:

```
#include<stdio.h>
void main(){
int i=046;

printf (« %o \n», i*2);
printf (« %x \n», i*2);
printf (« %d \n», i*2);

}
```

Opérations arithmétiques

- Solution

  - Donner le résultat affiché par le programme suivant:

```
#include<stdio.h>
void main(){
int i=046;

printf (« %o \n», i*2);
printf (« %x \n», i*2);
printf (« %d \n», i*2);

}
```

```
114
4c
76
```

Les entiers signés

- 3 représentations possibles
  - En signe-valeur absolue → Utilisée par les vieux processeurs
  - En complément à 1 → Quelques utilisations: exemple: le checksum sur un paquet IP
  - En complément à 2 → La représentation la plus utilisée de nos jours (les types signés: int, short et long sont représentés en complément à 2)

Les entiers signés

1) Représentation signe-valeur absolue

- Le bit de poids fort est le bit de signe: il indique s'il s'agit d'un nombre positif (0) ou négatif (1)
- Exemple: représentation sur 8 bits
  - 00000111= +7
  - 10000111=-7

- Pour une représentation sur 8 bits les valeurs vont de -127 à 127 (-2^{n-1} à 2^{n-1})

- 2 représentations possibles de zéro: 00000000 et 10000000 (zéro positif et zéro négatif)

- Exemples d'ordinateurs utilisant cette représentation
  - IBM7090 année 1959

Les entiers signés

1) Représentation signe-valeur absolue

- Inconvénients
  - Deux représentations possibles du zéro → une des représentations aurait pu être utilisée pour représenter un nombre supplémentaire
  - Des opérations d'addition et de soustraction assez compliquées (nécessitent plusieurs opérations)
    - Par exemple:
      - L'addition (ou la soustraction) toute simple de deux nombres de signes opposés ne fournit pas le bon résultat. Exemple: 7+(-6)= 00000111 + 10000110→ 10001101 ≠ 1
      - L'addition de deux nombres de signes opposés sur n bits consiste à :
        - Comparer les représentations sur n-1 (valeurs positives)
        - Soustraire la plus petite valeur de la plus grande
        - Ajouter le bit de signe adéquat

Les entiers signés

2) Représentation en complément à 1

- Nombre positif → la même valeur qu'un nombre non signé
  - Exemple: 00000111 représente le 7 en complément à 1

- Nombre négatif → A partir de la valeur absolue, les bits à 0 deviennent à 1 et ceux à 1 deviennent à 0
  - Exemple: 11111000 représente le -7 en complément à 1

- Pour une représentation en 8 bits les valeurs vont de -127 à 127

- 2 représentations possibles de zéro: 00000000 et 11111111 (zéro positif et zéro négatif)

- Exemples d'ordinateurs utilisant cette représentation
  - PDP-1 en 1959 et UNIVAC 1100/2200 en 1962

Les entiers signés

2) Représentation en complément à 1

- Avantages par rapport à la représentation signe valeur-absolue
  - Pas de comparaison/soustractions supplémentaires si les opérandes sont de signes opposés

- Inconvénients
  - Deux représentations possibles du zéro → une des représentations aurait pu être utilisée pour représenter un nombre supplémentaire
  - Pour avoir un résultat correct, il faut additionner la retenue s'il y en a

Exemple1:

$$\begin{array}{r} 0001\ 1100 \\ +\ 1100\ 0000 \\ \hline 1101\ 1100 \end{array} \qquad \begin{array}{r} 28 \\ +\ -63 \\ \hline -35 \end{array}$$

Exemple2:

$$\begin{array}{r} 0011\ 1111 \\ +\ 1110\ 0011 \\ \hline 1\ 0010\ 0010 \\ +\ \quad\quad\quad 1 \\ \hline 0010\ 0011 \end{array} \qquad \begin{array}{r} 63 \\ +\ -28 \\ \hline 35 \end{array}$$

Les entiers signés

3) Représentation en complément à 2

- Formation:
  - Nombre positif → la même valeur qu'un nombre non signé
    - Exemple: 00000111 représente le 7 en complément à 2
  - Nombre négatif
    - 1ère méthode: Faire un complément à 1 et additionner 1
      - Exemple: -7 en complément à 2: 11111000

        \+ 1

        = 11111001
    - 2ème méthode : Inverser les bits à partir du premier 1 à droite dans la représentation positive
      - Exemple: -7 en complément à 2: 00000111 ➔ 11111001
- Pour une représentation en 8 bits les valeurs vont de -128 à 127 (-2^{n-1} à $2^{n-1}-1$)
- Une seule représentation de 0 (00000000)
- Exemples d'ordinateurs utilisant cette représentation: les PDP-5 et PDP-6 en 1963-1964, System/360 en 1964, PDP-8 en 1965, Data General Nova en 1969 et PDP-11 en 1970

Les entiers signés

3) Représentation en complément à 2

- Avantage
  - Les opérations sont plus simples
    - On fait l'addition normalement. S'il y a une retenue, on la néglige

		0	0			1	1111	1
−63	:		1100	0001	+63	:	0011	1111
+28	:		0001	1100	−28	:	1110	0100
						1	0010	0011
−35	:		1101	1101	+35	:	0010	0011

- Pour cette raison, c'est la représentation la plus utilisée (les types signés: int, short et long sont représentés en complément à 2)

Les entiers signés

- Convertir en décimal un nombre binaire signé (en complément à 2)

  - Si le nombre est positif: convertir en décimal directement
    - Exemple sur 8 bits: $(00001010)_{C2}=(10)_{10}$

  - Si le nombre est négatif
    - Calculer son complément à 2
      - inverser les bits du nombre
      - additionner 1
    - Convertir en décimal
    - Ajouter le signe -
    - Exemple sur 8 bits: $(11111010)_{C2}$ → 00000101 → 00000110=6 → $(-6)_{10}$

C1 C2

Les entiers signés

- Exercice
  - Donner le résultat affiché par le programme suivant:

```
#include<stdio.h>
void main(){
char a,b;
a=0b00000011;
b=0b11111110;
printf(«  %d %d\n », a,b);
}
```

Les entiers signés

- Solution
  - Donner le résultat affiché par le programme suivant:

```
#include<stdio.h>
void main(){
char a,b;
a=0b00000011;
b=0b11111110;
printf(« %d     %d\n », a,b);
}
```

3 -2

Les entiers signés

- Débordement/overflow
  - Il se produit si le format des opérandes ne suffit pas pour avoir un résultat correct

➔Le résultat de l'opération est en dehors de l'intervalle des opérandes

  - Exemple: sur 8 bits, il y a un débordement si le résultat est supérieur à 127 ou inférieur à -128
  - Dans les additions, cela ne peut se produire que lorsque les deux opérandes sont du même signe
  - Exemple: Si on additionne 120 + 50 sur 8 bits on a un débordement (le résultat 170 est supérieur à 127)

Les entiers signés

- Débordement/overflow
  - Le processeur détecte qu'il y a eu un overflow, quand la dernière et l'avant dernière retenue sont différentes
  - L'overflow est confirmé quand on regarde le bit de signe du résultat qui est erroné

		0	0000	011			0≠	1000	111
+35	:		0010	0011	+103	:		0110	0111
+65	:		0100	0001	+65	:		0100	0001
+100	:		0110	0100	+168	≠		1010	1000

report		1	1111	111	report		1≠	0111	111
−35	:		1101	1101	−103	:		1001	1001
−65	:		1011	1111	−65	:		1011	1111
		1	1001	1100			1	0101	1000
−100	:		1001	1100	−168	≠		0101	1000

Dans ces exemples, on a un overflow dans les deux cas à droite puisque 8 bits ne suffisent pas pour représenter 168 et -168

Les entiers signés

- Représentation en complément à 2
  - Débordement/overflow

testOverflow.c

```c
#include <stdio.h>
int main(){

char a=103,b=65,c;
c=a+b;
printf("%d\n",c);
return 0;
}
```

```
eisea@VM-Linux:~/AlgoC/Microcontroleur$ ./testOverflow
-88
eisea@VM-Linux:~/AlgoC/Microcontroleur$
```

Extension de format en complément à 2

- Nombres positifs
  - On ajoute des 0 à gauche
  - Représentation du nombre 7
    - 0111: format 4 bits
    - 0000 0111: format 8 bits
    - 0000 0000 0000 0111: format 16 bits
- Nombres négatifs
  - On ajoute des 1 à gauche
  - Représentation du nombre -7
    - 1001: format 4 bits
    - 1111 1001: format 8 bits
    - 1111 1111 1111 1001: format 16 bits

Extension de format

- Exercice
  - Donner le résultat affiché par le programme suivant:

```
#include<stdio.h>
void main(){

char a=0b10010100;
int b=a;
printf(« %x\n »,b);

}
```

Extension de format

- Solution
  - Donner le résultat affiché par le programme suivant:

```
#include<stdio.h>
void main(){

char a=0b10010100;
int b=a;
printf(« %x\n »,b);

}
```

ffffff94

Les nombres réels

- Deux représentations différentes

  - A virgule fixe: c'est une représentation basée sur le placement virtuel d'une virgule sur les nombres entiers

  - A virgule flottante: c'est une représentation qui utilise une partie fractionnaire et un exposant.

    c'est la représentation utilisée dans la plupart des microprocesseurs (exemple: les types float et double dans le langage C)

Les nombres réels

- Représentation en virgule fixe

  - Il s'agit d'un **entier** avec une information sur la position de la virgule
  - La position de la virgule est **fixe** lors de l'écriture d'un programme ou de la conception d'un circuit (la décision est prise lors de l'écriture du programme ou la conception d'un circuit)
  - Exemple: codage en virgule fixe 4,4
    - $(0011,1101)_2 = (0011\ 1101)_2 \ / \ (2^4)_{10}$

      $= (61/16)_{10}$

      $= (3,8125)_{10}$

    - 2ème méthode de lecture : $(0011,1101)_2 = (0x2^3 + 0x2^2 + 1x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4})_{10}$

      $= (\ 0 + 0 + 2 + 1 + 0,5 + 0,25 + 0 + 0,0625\ )_{10}$

      $= (3,8125)_{10}$

Les nombres réels

- Représentation en virgule fixe
  - Conversion décimal → virgule fixe
    - Exemple convertir 3,8125 en virgule fixe 4,4
      - La partie entière: $(3)_{10} = (0011)_2$
      - La partie fractionnaire
        - On procède par des multiplications de 2 sur la partie fractionnaire jusqu'à arriver à un résultat égal à 1 ou dépasser le nombre de chiffres après la virgule fixé

0,8125*2=**1**,625

0,625*2=**1**,25

0,25*2=**0**,5

0,5*2=**1**

→ $(3,8125)_{10} = (0011,1101)_2$

Les nombres réels

- Représentation en virgule fixe
  - Nombres signés: le même principe que pour les entiers
    - $(3,8125)_{10} = (0011,1101)_{C2}$
    - $(-3,8125)_{10} = (1100,0011)_{C2}$

Les nombres réels

- Représentation en virgule fixe
  - Avantage: Opérations simples
    - Le processeur effectue les opérations comme si elles étaient sur des entiers (addition, multiplication, etc.)

- Exemple:

	0010,1101	2,8125
+	0010,0100	+ 2,25
	0101,0001	5,0625

➔ le processeur n'a pas besoin d'un module spécial pour les opérations sur des réels, il utilise les mêmes modules matériels que pour les opérations sur les entiers

Les nombres réels

- Représentation en virgule fixe
  - Inconvénients
    - Plus de complexité de programmation (exemple: décalage pour les multiplications)

- Exemple:

```
        0010,0100   2,25
    x   0010,0100   2,25
  ---------------------
          1
           00000000
          00000000
         00100100
        00000000
       00000000
      00100100
     00000000
    00000000
  ---------------------
   000010100010000
   0000101,00010000
       0101,0001 =5,0625
```

Ecriture du résultat dans le format de base 4,4
→ décalage à droite de 4 positions

Les nombres réels

- Représentation en virgule fixe
  - Inconvénients
    - Le choix de la position de la virgule a un impact sur la dynamique (intervalle entre la plus grande et la plus petite valeur) et la précision

      - La dynamique
        - Un format 6,2 permet de représenter des valeurs de 100000,00 à 011111,11 (-32,00 à 31,75)
        - Un format 2,6 permet de représenter des valeurs de 10,000000 à 01,111111 (-2 à 1,984375)

      - La précision
        - Un format 6,2 ne permet pas de représenter la valeur décimale 1,8 par exemple (la valeur après la virgule en format 6,2 a un maximum de $(0,11)_2 = (0,75)_{10}$

        ➔ **$(1,8)_{10}$** est représenté par 000001,11 dans le format 6,2, la valeur exacte de 000001,11 est **1,75**

        - Si on choisit un format 2,6, on obtient 01,110011= 1,796875 → on est plus proche de la valeur décimale

Les nombres réels

- Représentation en virgule fixe
  - Inconvénients
    - La nécessité de connaître à l'avance la précision nécessaire, et donc l'ordre de grandeur des nombres à manipuler
    - Ce type de connaissance a priori existe pour certaines applications, mais pour traiter des applications génériques, il vaut mieux avoir une représentation plus flexible.
    - Le programmeur doit gérer les overflows
    - Pour pallier ce manque de flexibilité, le concept de virgule flottante a été introduit.

Les nombres réels

- Représentation en virgule flottante
  - Avantage
    - Pas de position fixe pour la virgule → une dynamique (intervalle entre la plus grande et la plus petite valeur) plus grande → cette représentation est utilisée pour des applications avec différents niveaux de précision

Les nombres réels

- Représentation en virgule flottante

Signe	Exposant	Fraction

- A la différence de la représentation en virgule fixe, les nombres signées ne sont pas représentés en complément à 2

- Deux formats possibles
  - Simple précision sur 32 bits (ce qui correspond à float en C/Java)
  - Double précision sur 64 bits (ce qui correspond à double en C/Java)

	Encodage	Signe	Exposant	Fraction
Simple précision	32 bits	1 bit	8 bits	23 bits
Double précision	64 bits	1 bit	11 bits	52 bits

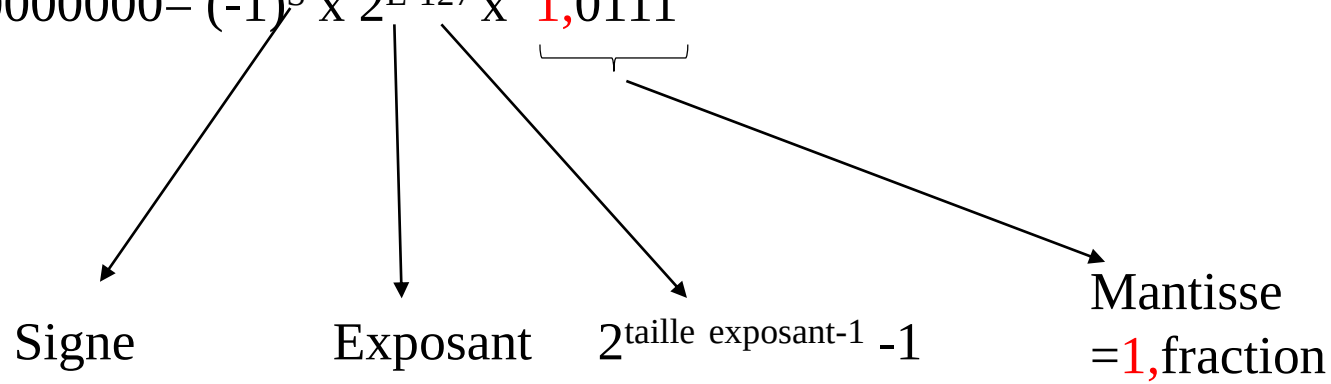
Type	Taille	Dynamique
float	4 octets	$-3,4*10^{38}$ à $3,4*10^{38}$
double	8 octets	$-1,7*10^{308}$ à $1,7*10^{308}$

Les nombres réels

- Représentation en virgule flottante
  - Conversion virgule flottante → décimal
    - Exemple: = 0 10000001 01110000000000000000000

	Encodage	Signe	Exposant	Fraction/ Pseudomantisse
Simple précision	32 bits	1 bit	8 bits	23 bits

$0\ 10000001\ 01110000000000000000000 = (-1)^S \times 2^{E-127} \times 1{,}0111$



$(-1)^S \times 2^{E-127} \times 1{,}0111 = (-1)^0 \times 2^{129-127} \times 1{,}0111 = 1{,}0111 \times 2^2 = 101{,}11 = \mathbf{5{,}75}$

Les nombres réels

- Représentation en virgule flottante

Signe (S)	Exposant (E)	Pseudomantisse (P)
1 bit	e bits	p bits

  - Conversion décimal → virgule flottante

Le nombre ainsi représenté a pour valeur :
$(-1)^S * 2^{E-(2^{e-1}-1)} * (1 + \frac{P}{2^p})$

    - Exemple: convertir 5,75 en simple précision

1) Convertir en virgule fixe
5 → 101
0,75=0,5 +0,25 → 0,11
➔5,75=101,11

2) Décaler la virgule à gauche jusqu'à avoir 1 dans la partie entière et multiplier par la puissance de 2 correspondante: $5,75=101,11=1,0111*2^2$

3) Ajouter 127 à l'exposant obtenu (2^7 -1=127 pour la simple précision et 2^{10} -1=1023 pour la double précision) ➔E=2+127=129= 10000001

4) Ecrire la pseudomantisse sur 23 bits
P= 01110000000000000000000

➔ 5,75= 0 10000001 01110000000000000000000

Les nombres réels

- Représentation en virgule flottante
  - Addition
    - Le nombre ayant le plus petit exposant est décalé à droite de façon à avoir le même exposant
    - Les résultats sont additionnées
    - Exemple:
      - 0,25+1,5 en simple précision
        - 0,25=0 01111101 00000000000000000000000=1,0x $2^{125-127} = 1,0x\ 2^{-2}$
        - 1,5= 0 01111111 10000000000000000000000=1,1x $2^{127-127} = 1,1x\ 2^{0}$

        → décaler à droite 0,25 → 0,01 x 2^0

      - Addition des résultats $1,1x\ 2^0 + 0,01x\ 2^0 = 1,10000000000000000000000x\ 2^0$
      - Résultat= 0 01111111 10000000000000000000000= 1,75

Les nombres réels

- Représentation à virgule flottante
  - Multiplication
    - Addition des exposants – (2^{e-1} -1)
    - Multiplication des deux mantisses
    - Décalage du résultat de la multiplication pour avoir un résultat de la forme 1,xxxx (ajouter le décalage au résultat d'addition des exposants)
    - Exemple
      - 2,5x0,75 en simple précision
        - 2,5= 0 10000000 01000000000000000000000
        - 0,75= 0 01111110 10000000000000000000000
      - Addition des exposants - 2^{8-1} -1 → 128+126 – 127=127=01111111
      - Multiplication des mantisses 1,01 x 1,1=1,111 (pas besoin de décalage)

      → E=01111111 et P=11100000000000000000000

      → Résultat= 0 01111111 11100000000000000000000=1,111x$2^{127-127}$=1,875

Les nombres réels

- Représentation à virgule flottante
  - Inconvénients
    - La multiplication sur les flottants est plus compliquée que celle sur les virgules fixes (entiers)
      → elle prend généralement plus de temps qu'une multiplication à virgule fixe
    - Le processeur ne peut pas utiliser la même unité arithmétique et logique pour faire ces multiplications
      →Les processeurs intègrent donc des unités de calcul flottant
      → Les processeurs à virgule flottante coûtent plus chers que ceux à virgule fixe

Les nombres réels

- Comparaison entre virgule fixe et virgule flottante

	Virgule fixe	Virgule flottante
Dynamique (intervalle entre la plus grande et la plus petite valeur)	Plus limitée	Plus grande
Temps de développement	Plus long	Plus court
Temps d'exécution	Plus court	Plus long
Précision et overflow	Moins de précision et plus d'overflow	Plus de précision et moins d'overflow
Coût du processeur	Moins cher	Plus cher
Consommation d'énergie	Consommation plus petite	Consommation plus grande

Le choix entre les deux représentations dépend de plusieurs facteurs: les applications visées, le coût du processeur, les contraintes temps-réel, l'autonomie souhaitée, etc.

Bibliographie

- Les niveaux de tension d'un signal logique
  - http://www.allaboutcircuits.com/textbook/digital/chpt-3/logic-signal-voltage-levels/
- Opérations sur nombres signés
  - http://www.math-info.univ-paris5.fr/~gk/NumLog/CM/NL_CM3.pdf
  - http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/arith.int.html
- Opérations en virgule flottante
  - http://www.sifflez.org/lectures/archi-ord/AideMemoireIEEE754.pdf