Algorithmique & Langage C
IUT GEII S1

Ports d'entrées/sorties
Opérations bits à bits

Notes de cours

(sixième partie)

cours_algo_lgc6.01.odp

Licence

La mémoire

Une mémoire est un circuit à semi-conducteur permettant d'enregistrer, de conserver et de restituer des informations.

Elle est utilisée pour stocker les instructions d'un programme ainsi que les données associées (variables, constantes).

Une mémoire peut être représentée comme une armoire de rangement constituée de différents tiroirs.

Chaque tiroir représente alors une case mémoire qui peut contenir un seul élément : des données.

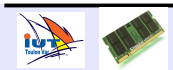
Chaque tiroir possède un numéro appelé adresse.

Chaque donnée devient alors accessible grâce à son adresse.

La mémoire

0xFFFFFFFF	10011001
0xFFFFFFFE	00001000
0xFFFFFFFD	01111111
0xFFFFFFFC	01111001
0x00000003	01001111
0x00000002	00001011
0x00000001	11001011
0x00000000	01001001

Adresses (ici 32bits) Données (ici 8bits)



Little Endian / Big Endian

Pour stocker une donnée comportant plusieurs octets dans une mémoire dont la taille des données sont des octets, il y a deux manières de procéder :

- **Little Endian** : le poids faible dans l'adresse de plus faible valeur (the little end comes first)

µP : Intel , Microchip, ...

- **Big Endian** : le poids faible dans l'adresse de valeur la plus forte (the big end comes first)

µP : Freescale (Ex Motorola), ...

Little Endian / Big Endian

La valeur entière 109243=0x0001AABB est stockée à partir de l'adresse 0x00221234 :

Little Endian :

0x00221237	0x00
0x00221236	0x01
0x00221235	0xAA
0x00221234	0xBB

} int i = 109243 ;

Big Endian :

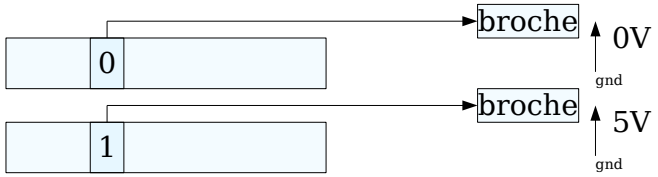
0x00221237	0xBB
0x00221236	0xAA
0x00221235	0x01
0x00221234	0x00

} int i = 109243 ;

Port d'entrées/sorties

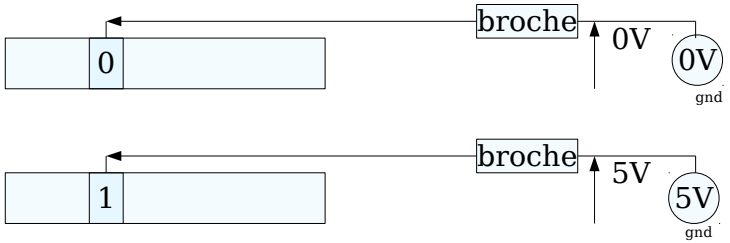
Les microcontroleurs disposent de broches (physiques) en correspondance avec la mémoire.
La correspondance est configurable en entrée ou en sortie.

Sortie : la valeur (0 ou 1) d'un bit de la mémoire détermine l'état logique (la tension) de la broche correspondante.



Port d'entrées/sorties

Entrée : un bit de la mémoire est positionné à 0 ou à 1 en fonction de l'état logique (la tension) présent sur la broche correspondante.



Port d'entrées/sorties

Les entrées/sorties sont regroupées souvent par 8, 16 ou 32.
Un tel regroupement est nommé **port**.
A un port de 8 broches correspond un octet de la mémoire.

Sur le PIC 18F2550 par exemple, on trouve :

PORTA : 7 broches
PORTB : 8 broches
PORTC : 7 broches

Chacune de ces broches est configurable indépendamment en entrée ou en sortie.

Au reset, toutes les broches bidirectionnelles sont toujours configurées en entrées.


PORTA

Trois registres (case mémoires dédiées) importants :

PORTA : permet de lire l'état des broches associées

LATA : permet de fixer l'état des broches en sorties

TRISA : permet de configurer le sens de la broche
Bit à 0 : la broche correspondante est une sortie
Bit à 1 : la broche correspondante est une entrée

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTA	—	RA6(1)	RA5	RA4	RA3	RA2	RA1	RA0
LATA	—	LATA6(1)	LATA5	LATA4	LATA3	LATA2	LATA1	LATA0
TRISA	—	TRISA6(1)	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0


Lecture d'une entrée

Configuration en entrée :

```
TRISAbits.TRISA2 = 1 ; // inutile au reset
```

Lecture :

```
if ( PORTAbits.RA2 == 1 ) {
            // la broche RA2 est à 1
      }
  else {
         // la broche RA2 est à 0
  }
```


Positionnement d'une sortie

Configuration en sortie :

```
TRISAbits.TRISA2 = 0 ;
```

Écriture :

Mise à 1 :

```
LATAbits.LATA2 = 1 ;
```

Mise à 0 :

```
LATAbits.LATA2 = 0 ;
```


PORTB et PORTC

Les port B et C fonctionnent de manière similaire :

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
LATB	LATB7	LATB6	LATB5	LATB4	LATB3	LATB2	LATB1	LATB0
TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTC	RC7	RC6	RC5(1)	RC4(1)	—	RC2	RC1	RC0
LATC	LATC7	LATC6	—	—	—	LATC2	LATC1	LATC0
TRISC	TRISC7	TRISC6	—	—	—	TRISC2	TRISC1	TRISC0
UCON	—	PPBRST	SE0	PKTDIS	USBEN	RESUME	SUSPND	—

Legend: — = unimplemented, read as '0'. Shaded cells are not used by PORTC.

Note 1: RC5 and RC4 are only available as port pins when the USB module is disabled (UCON<3> = 0).

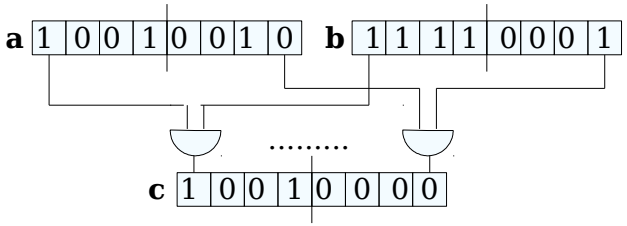
Opérateur ET bit à bit

Opérateurs ET bit à bit : **&**

(à ne pas confondre avec le ET logique déjà vu : **&&**)

```
char  a = 0x92 , b = 0xF1 , c ;

c = a & b ;
```



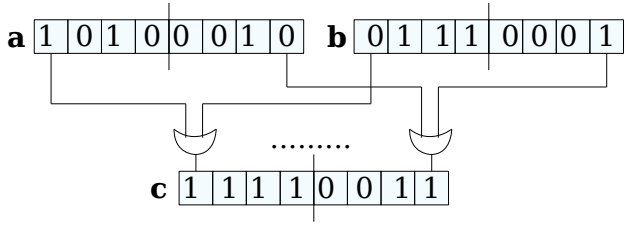
Opérateur OU bit à bit

Opérateurs OU bit à bit : **|**

(à ne pas confondre avec le OU logique déjà vu : ||)

```
char  a = 0xA2 , b = 0x71 , c ;

c = a | b ;
```

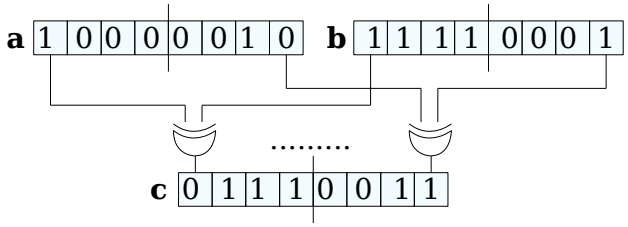


Opérateur OU Exclusif bit à bit

Opérateurs OU Exclusif bit à bit : ^

```
char  a = 0x82 , b = 0xF1 , c ;

c = a ^ b ;
```

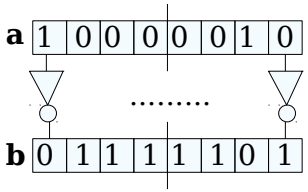


Opérateur d'inversion bit à bit

Opérateurs d'inversion bit à bit : ~

```
char  a = 0x82 ;

b= ~a ;
```



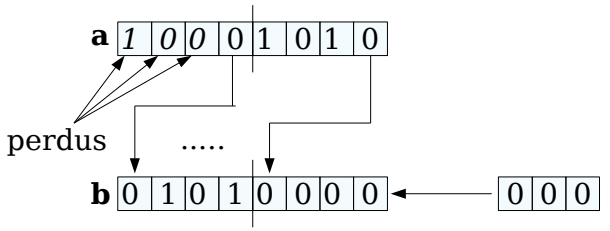
Tous les bits de a sont inversés.
b vaut donc ici 0x7D

Attention à ne pas confondre avec la négation logique **!**
!(x) est vrai (et vaut 1) quand x est faux et inversement.
Comme toute valeur non nulle est vraie, ici avec b=!a;
On aurait b qui vaudrait 0x01.

Opérateur de décalage à gauche

Décalage à gauche : **<<**
Tous les bits sont décalés vers la gauche, les bits qui "entrent" sont des zéros, les bits qui "sortent" sont perdus.

char a=0x8A , b ;
b = a << 3 ; // décalage de trois positions vers la gauche

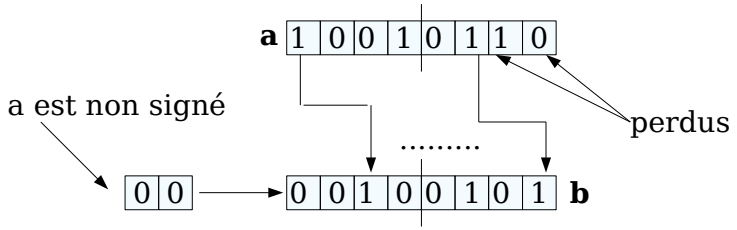


Mathématiquement, un décalage à gauche de n revient à multiplier par 2^n.
Si des bits non nuls sont perdus c'est que le résultat ne tient pas dans le format.

Opérateur de décalage à droite

Décalage à droite : **>>**
Tous les bits sont décalés vers la droite, les bits qui "sortent" sont perdus. Lorsque la valeur est non signée, les bits qui "rentrent" sont égaux à zéro.

unsigned char a=0x96 , b ;
b = a >> 2 ; // décalage de deux positions vers la droite.

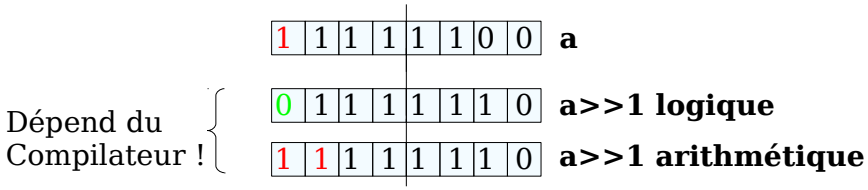


Le décalage à droite de n positions correspond à une division par 2^n.
Les bits perdus correspondent à 0.5,0.25,0.125 etc...

Opérateur de décalage à droite

En C, si on décale à droite une valeur signée, le résultat dépend du compilateur utilisé !
Les bits qui rentrent sont soit des zéros (décalage logique), soit la valeur du bit de signe (décalage arithmétique).
Il vaut donc mieux éviter d'utiliser l'opérateur >> sur des valeurs signées pour conserver la portabilité.

Ex :
signed char a = -4 ;



Masque d'un bit

Les masques de bits sont utilisés pour isoler un bit (ou un sous-ensemble de bit). Ce sont des quantités comportant tous les bits à zéros sauf celui (ou ceux) que l'on veut isoler. Les masques permettent de tester ou de modifier des bits sans influencer les autres.

masque du bit 3 : 0000 1000
masque des bits 1 à 4 : 0001 1110

valeur : 0101 1101
masque : 0000 1000

Test d'un bit :
La quantité **valeur&masque** est non nulle ssi le bit correspondant au masque vaut 1

Tester un bit

On fait un ET avec son masque :

Ex : Si le bit 4 de x vaut 1 ....

```
mask = 0x10;   // 0001 0000

if ( x & mask ) {

   // le bit 4 de x vaut 1
} else
{
   // le bit 4 de x vaut 0
}
```


Tester un bit

En C, le masque du bit n° i peut s'écrire simplement

```
1<<i
```

Donc pour tester le bit i on peut écrire :

```
if ( x & (1<<i) ) {


}
```


Mettre un bit à 1 (sans modifier les autres)

On fait un OU avec le masque du bit

int x ;

x = x | 0x10 ; // met le bit 4 de x à 1

x	1	0	1	0	1	0	0	1
masque	0	0	0	1	0	0	0	0
x | masque	1	0	1	1	1	0	0	1

Rappel de logique
a+1=1
a+0=a

Mettre un bit à 0 (sans modifier les autres)

On fait un ET avec l'inverse du masque du bit

int x ;

x = x & (~0x10) ; // met le bit 4 de x à 0

x	1	0	1	1	1	1	0	1
~masque	1	1	1	0	1	1	1	1
x & ~masque	1	0	1	0	1	1	0	1

Rappel de logique
a.1=a
a.0=0


Inverser un bit (sans modifier les autres)

On fait un Ou exclusif avec le masque du bit

int x ;

x = x ^ 0x10 ; // inverse le bit 4 de x

x	1	0	1	1	1	0	0	1
masque	0	0	0	1	0	0	0	0
x | masque	1	0	1	0	1	0	0	1

Rappel de logique
$a \oplus 1 = \bar{a}$
$a \oplus 0 = a$


Opérateurs "condensés"

En C un expression de la forme

```
x = x op y ;
```

peut s'écrire

```
x op= y ;
```

Pour

```
op ∈ { +, -, * , / , % , &,|, ^,<< ,>> }
```

Ainsi :

x = x + 3 ; <=> x += 3 ;
y = y - b ; <=> y -= b ;
t = z << a ; <=> z <<= a ;
t = t & 0x04; <=> t &= 0x04 ;
etc....


Macros de modifications de bits

Il est souvent pratique de définir les macros suivantes pour manipuler les bits :

```
// Mise à un d'un bit d'un octet
#define BIT_SET( octet, nbit )      ( octet |= (1<<(nbit))   )

// Mise à zéro d'un bit dans un octet
#define BIT_CLEAR( octet , nbit )   ( octet &= ~(1<<(nbit))  )

// Test d'un bit dans un octet
// (renvoie 1 si le bit est à 1 et 0 sinon)
#define BIT_TEST( octet , nbit  )   ( octet&(1<<(nbit))? 1:0 )
```


Application

Sur l'horloge le câblage est le suivant :
(version 2013 avec 18F2550)
PORTA : LED0 bit4 LED1 bit5
PORTB : LED 5 , 6 , 7 , 8 → bits 2 , 3, 4 , 5
PORTC : LED 2, 3 , 4 → bits 0, 1 , 2
LED 9 , 10, 11 non câblées car broches utilisées par l'USB

	7	6	5	4	3	2	1	0
PORTA	-	-	L1	L0	-	-	-	-

	7	6	5	4	3	2	1	0
PORTB	-	-	L8	L7	L6	L5	-	-

	7	6	5	4	3	2	1	0
PORTC	-	-	-	-	-	L4	L3	L2


Application

Pour pouvoir allumer facilement les LED, il est nécessaire d'avoir une correspondance simple n°bit → n° LED.

Soit v une valeur binaire, on veut que chaque bit de v commande la Led de même numéro.

Ainsi par exemple pour allumer une LED sur deux :

0x0155 → 0000 0001 0101 0101

Allumées : L0, L2 , L4 , L6 , L8 Éteintes : L1 , L3 , L5 , L7

unsigned int v =0x0155 ;

Il faut savoir prendre les bits de v pour les appliquer correctement sur les PORTA , PORTB et PORTC


PORTA

	7	6	5	4	3	2	1	0
PORTA	-	-	L1	L0	-	-	-	-

v	0	0	0	0	0	0	0	L8	L7	L6	L5	L4	L3	L2	**L1**	**L0**

Pour le PORTA :

isoler les bits 0 et 1 de v (masque)

```
v & 0x0003
```

les décaler de 4 positions vers la gauche

```
( v & 0x0003 ) << 4
```

Appliquer le résultat sur les sorties du PORTA

```
LATA = ( v & 0x0003 ) << 4 ;
```

(les autres bits du port A ne sont pas affectés car ce sont des entrées)

PORTB

	7	6	5	4	3	2	1	0
PORTB	-	-	L8	L7	L6	L5	-	-

v	0	0	0	0	0	0	0	**L8**	**L7**	**L6**	**L5**	L4	L3	L2	L1	L0

Pour le PORTB :

isoler les bits de v (masque)

```
v & 0x
```

les décaler de positions vers la droite

```
( v & 0x ) >>
```

Appliquer le résultat sur les sorties du PORTB

```
LATB =                    ;
```

(les autres bits du port B ne sont pas affectés car ce sont des entrées)

PORTC

	7	6	5	4	3	2	1	0
PORTC	-	-	-	-	-	L4	L3	L2

v	0	0	0	0	0	0	0	L8	L7	L6	L5	L4	L3	L2	L1	L0

Pour le PORTC :

isoler les bits de v (masque)

```
v & 0x
```

les décaler de positions vers la

```
( v & 0x )
```

Appliquer le résultat sur les sorties du PORTC

```
LATC =                    ;
```

(les autres bits du port C ne sont pas affectés car ce sont des entrées)