

Opérations binaires, décalages, masques et formats binaires compacts

Les méthodes de conversions décimal->hexa et décimal->binaire sont supposées connues.

Les conversions hexa->binaire peuvent être réalisées rapidement avec la table de correspondance ci jointe.

Un mot binaire est un ensemble de n bits, typiquement une puissance de 2 : 4, 8(octet), 16, 32 ou 64 bits.

Le processeur de la console de jeu portable Game Boy Advance (GBA) est 32 bits, sans autre précision "un mot" sera synonyme de "mot 32 bits"

En C (et assembleur) les constantes entières écrites en hexadécimal sont préfixées par 0x. Le préfixe 0b indique le binaire (non reconnu par le C).

Les bits de poids fort sont les plus gauche, les bits de poids faibles sont à droite.

Opérateurs Logiques

Ce sont les opérateurs usuels de la logique booléenne : ET & OU | Oux (Exclusif) ^ NON (complément à 1) ~
L'UAL d'un processeur peut appliquer **en parallèle** ces opérations entre tous les bits de 2 mots opérands (ici des octets)

1 0 1 0	1 0 1 0
& 1 1 0 0	1 1 0 0
1 0 0 0	1 1 1 0
1 0 1 0	~ 1 0
^ 1 1 0 0	0 1
0 1 1 0	

Exemples 0xE3 = 0b11100011 0xE3 = 0b11100011 0xE3 = 0b11100011 NON 0x25 = 0b00100101
 ET 0x25 = 0b00100101 OU 0x25 = 0b00100101 Oux 0x25 = 0b00100101 0xDA = 0b11011010
 0x21 = 0b00100001 0xE7 = 0b11100111 0xC6 = 0b11000110

Tables de vérité

Exercices 0x5A = 0b..... 0x5A = 0b..... 0x5A = 0b..... NON 0x79 = 0b.....
 ET 0x79 = 0b..... OU 0x79 = 0b..... Oux 0x79 = 0b..... 0x.. = 0b.....
 0x.. = 0b..... 0x.. = 0b..... 0x.. = 0b.....

pour info : printf("%x\n", 0xE3 ^ 0x25); // Affiche C6 (%x affiche un entier en hexadécimal)

Décalages et rotations. Opérateurs << : décalage à gauche >> : décalage à droite

Le décalage logique à gauche déplace les valeurs des bits à l'intérieur du mot vers la gauche de n positions. Les valeurs précédentes dans les n bits de poids forts sont perdues et les n bits de poids faibles sont mis à 0. Le décalage logique à droite réalise l'opération en sens inverse.

Exemple : Valeur (en hexa) du mot 0xF0ABCD0E après un décalage logique de 4 bits à gauche : 0X0ABCD0E0

Que vaut (en hexa) le mot 0xF0ABCD0E après un décalage logique de 8 bits à droite ?

Que vaut (en hexa) le mot 0xF0ABCD0E après un décalage logique de 7 bits à droite ?

Que vaut (en hexa) le mot 0xF0ABCD0E après une rotation de 24 bits à droite ?

pour info : printf("%x\n", 0xF0ABCD0E << 4); // Affiche 0ABCD0E0 << décalage à gauche

Masques binaires

Les masques binaires permettent d'extraire l'état de certains bits d'un mot, ou de modifier l'état de certains bits sans modifier les autres.

Exemples : On dispose d'un octet déclaré en C par

```
unsigned char p;
...
if (p & 0x20) { traitement ... }
p = p | 0x06;
p = p & 0xF9; // 0xF9 = ~0x06
```

On souhaite tester le bit 5 de l'octet et déclencher un traitement si il vaut 1

On souhaite mettre à 1 (set) les bits 1 et 2 de l'octet

On souhaite mettre à 0 (reset) les bits 1 et 2 de l'octet

Exercices :

On souhaite tester le bit 0 de l'octet et déclencher un traitement si il vaut 1

On souhaite mettre à 1 (set) les bits 3, 4 et 5 de l'octet

On souhaite mettre à 0 (reset) les bits 3, 4 et 5 de l'octet

On souhaite inverser (invert, toggle) le bit 7 de l'octet

pour info (à lire après le TD) :

p&=0xF9; est équivalent à p=p&0xF9; Ceci est valable pour les opérateurs & | ^

D'autre part il est fréquent en programmation "bas niveau" d'avoir dans un fichier en-tête (un #include <bidule.h>) une liste de #define (ou un enum) qui associent les puissances de 2 correspondant aux positions de bits qu'on veut tester ou positionner :

```
#define PLANE_HIGH 0x01 // Le bit 0 correspond à l'avion en altitude
#define PLANE_FIRE 0x02 // Le bit 1 correspond à l'avion en feu
#define PLANE_FUMES 0x04 // Le bit 2 correspond à la cabine pleine de fumées
#define PLANE_LGEAR 0x08 // Le bit 3 correspond au train d'atterrissage sorti
#define PLANE_WARNING 0x10 // Le bit 4 déclenche l'allumage des lumières rouges qui clignent
#define PLANE_ALARM 0x20 // Le bit 5 déclenche une sirène stridente
```

```
// Si en altitude et train d'atterrissage sorti, allumer lumières rouges
if ( (plane & PLANE_HIGH) && (plane & PLANE_LGEAR) ) plane |= PLANE_WARNING;
```

```
// Si l'avion est en feu ou la cabine pleine de fumée, allumer lumières rouges et sirène
if ( plane & (PLANE_FIRE | PLANE_FUMES) ) plane |= PLANE_WARNING|PLANE_ALARM;
```

ce qui est plus lisible que if((plane&1)&&(plane&8)) plane|=16; if(plane&6) plane|=48;

Formats binaires compacts, champs binaires

Pour optimiser la place occupée en mémoire par des informations de petites tailles, il est fréquent de vouloir "compacter" plusieurs champs de données dans un seul mot (4 octets ou 2 octets ou 1 octet).

Ainsi dans la mémoire vidéo de la GBA (en mode graphique 3) la couleur de chaque pixel de l'écran est représentée à l'intérieur d'un demi-mot (16 bits) par un format BGR sur 15 bits. Chaque composante couleur primaire (Blue, Green, Red) codée sur 5 bits peut prendre $2^5 = 32$ valeurs différentes.

Bleu (entre 0 et 31) $b_4 b_3 b_2 b_1 b_0$ Vert (entre 0 et 31) $g_4 g_3 g_2 g_1 g_0$ Rouge (entre 0 et 31) $r_4 r_3 r_2 r_1 r_0$

Le tout tient dans un demi mot de 16 bits : $0 b_4 b_3 b_2 b_1 b_0 g_4 g_3 g_2 g_1 g_0 r_4 r_3 r_2 r_1 r_0$

Compte tenu de ce format, combien de nuances de couleurs la GBA peut-elle utiliser dans ce mode graphique ?

Pour la plupart des systèmes graphiques actuels une couleur est un triplet Rouge Vert Bleu d'octets, c'est à dire que chaque composante primaire est représentée par une valeur entre 0 (minimum) à 255 (maximum) :

```
red=0; green=0; blue=0; // Noir
red=0; green=255; blue=0; // Vert
red=127; green=255; blue=127; // Vert clair
red=255; green=255; blue=0; // Jaune
red=255; green=255; blue=255; // Blanc
```

Pour le projet sur GBA on souhaite pouvoir passer de cette représentation usuelle des couleurs au format spécifique BGR 15 bits et réciproquement.

Ecrire en C le code pour passer de 3 composantes dans des variables séparées au demi-mot couleur dans une variable unique et réciproquement. Il sera sans doute nécessaire d'utiliser des opérateurs logiques et des opérateurs de décalage.

```
// Déclaration des variables
u8 red, green, blue; // u8 est synonyme de unsigned char : entier non signé sur 8 bits
u16 color; // u16 est synonyme de unsigned short int : entier non signé sur 16 bits
```

<pre>// composantes -> couleur red=0x45;green=0x98;blue=0xF7; . . CODE A ECRIRE . -> color vaut 0x7A68</pre>	<pre>// couleur -> composantes color=0x7A68; . . CODE A ECRIRE . -> red vaut 0x40, green vaut 0x98 et blue vaut 0xF0</pre>
--	--

Les composantes seront données dans l'intervalle 0 255 : seuls les 5 bits de poids forts sont utilisés, les 3 bits de poids faibles seront ignorés.

```
red=0b01000--- green=0b10011--- blue=0b11110--- <-> color=0b011101001101000=0x7A68
rrrrr ggggg bbbbb bbbbbbggggrrrrr
'-' indique soit un bit nul soit une valeur ignorée
```

pour info (à lire après le TD) : Champs binaires dans une structure, *bitfields*

Il est possible en C de déclarer une structure contenant des champs plus petits que l'octet
Ainsi une couleur BGR 15 bits peut être représentée par la déclaration suivante

```
typedef struct {
    unsigned int red : 5; // Champs rouge : 5 bits de poids faibles 0 à 4
    unsigned int green : 5; // Champs vert : 5 bits suivants 5 à 9
    unsigned int blue : 5; // Champs bleu : 5 bits suivants 10 à 14
} BGRcolor;
```

L'accès aux différents champs se fait comme pour une structure usuelle :

```
BGRcolor col; // Déclaration d'une variable de type bitfield BGRcolor

col.red = 8; // On affecte des valeurs entre 0 et 31 à chaque champs
col.green = 19;
col.blue = 30;
```

Cette notation est pratique pour mettre des informations de petite taille dans des espaces de stockage réduit.

Un inconvénient est que l'empreinte mémoire réelle d'un bitfield est généralement au minimum la taille d'un mot : ici 4 octets (2 suffisaient)
D'autre part le compilateur considère qu'il ne s'agit pas d'un scalaire ce qui oblige parfois à faire de laborieux transtypages :

```
u16 color; // Déclaration d'une variable 16 bits contenant une couleur utilisable sur la GBA
...
color=((u16 *)&col); // Copie du contenu du bitfield dans un scalaire de taille convenable
printf("%x\n",color); // Affichage du scalaire pour vérifier
```