

TD 4

Correction du dernier exercice :
Dessin d'images bitmapsEcrire le code assembleur de **asmDrawImage** : dessiner à l'écran une image bitmap importée

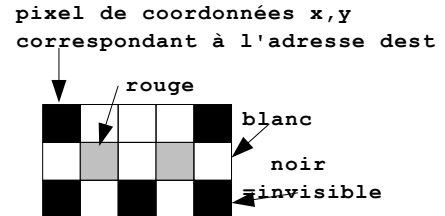
On dispose d'une image graphique importée dans le programme sous la forme d'une structure de type `t_image` initialisée.

Un utilitaire vu au TP3, basé sur allegro, réalise des conversions des formats de fichiers bmp ou pcx vers ces blocs de codes qui peuvent être ensuite inclus dans le projet. Exemple de code et image associée, à dessiner à partir de ces données :

```
typedef struct{
    int w,h;
    u16 img[];
} t_image;

// généré automatiquement à partir
// d'un fichier image invader.bmp

const t_image invader={5,3,{
    0x0000,0x7fff,0x7fff,0x7fff,0x0000,
    0x7fff,0x001f,0x7fff,0x001f,0x7fff,
    0x0000,0x7fff,0x0000,0x7fff,0x0000
}};
```



```
Prototype C : void asmDrawImage(u16 * dest, u16 * img, int w, int h);
Entrées : dest = adresse de départ du remplissage (r0)
          img = pointeur sur le tableau de couleurs (r1)
          w = largeur rectangle (r2)
          h = hauteur rectangle (r3)
```

Pour afficher le sprite (l'image) de l'invader aux coordonnées x y, l'appel depuis le C sera :

```
asmDrawImage((u16 *) (0x06000000+2*x+480*y), invader.img, invader.w, invader.h);
```

Les couleurs sont dans un tableau à une dimension, elles correspondent aux couleurs des pixels successifs du rectangle image dessiné à l'écran selon le balayage habituel de gauche à droite et de haut en bas. Inspirez vous du code de `asmRectcpy` du TD3 pour afficher ces couleurs telles quelles, puis modifiez cette 1ère version pour prendre en compte la transparence : les pixels noirs ne sont pas affichés.

```
.ARM .GLOBAL ...
```

```
asmDrawImage:
```

```
    push    {r4,r5}
```

```
diBoucleV:      @ Boucle verticale : r3 servira de compteur
    mov     r4,r2      @ Init boucle horizontale : r4 sert de compteur
```

```
diBoucleH:
    ldrrh  r5,[r1]     @ lecture depuis le tableau image
    cmp    r5,#0       @ la couleur est 0 ? -> invisible
    strneh r5,[r0]     @ transfert vers adresse destination r0 si visible
    add    r1,r1,#2    @ cellule tableau image suivante
    add    r0,r0,#2    @ dest : pixel horizontal suivant (+2 octets)
    subs  r4,r4,#1     @ décrémenter compteur boucle horizontale
    bne   diBoucleH   @ boucler pixel horizontal suivant
    sub   r0,r0,r2,lsr #1 @ dest. : Retour en début de segment horizontal
    add   r0,r0,#480   @ Puis ligne en dessous
    subs  r3,r3,#1    @ décrémenter compteur boucle verticale
    bne   diBoucleV   @ boucler ligne suivante
```

```
    pop    {r4,r5}
    bx    lr
```

```
@ Il peut être utile de regarder la doc technique (en anglais) pour voir la syntaxe du store conditionnel strneh
@ str : store register (stocker valeur de registre vers mémoire)
@ ne : not equal (seulement si cmp précédent ne signale pas une égalité)
@ h : half word (seulement 2 octets sont concernés, puisque chaque pixel vaut 2 octets)
```

Facultatif : comme pour l'exercice précédent, le choix des paramètres est fait pour faciliter votre développement en assembleur. Un format d'appel plus simple pour l'utilisateur final correspondrait au prototype `void asmDrawImage(const t_image * bmp, int x , int y)` ; Comment implémenter la réception de cet appel au niveau de la procédure assembleur ?

```
@ attention piège, bmp->img n'est pas un pointeur mais le début du tableau: en mémoire w,h,img[0],img[1]...
    push    {r4,r5}
    @ Calculer adresse dest dans r4 à partir de x(r1) et y(r2)
    @ séquence vue TD2 : rsb r4,r2,r2,lsr #4 / mov r4,r4,lsr #5 / add r4,r4,r1,lsr #1 / add r4,r4,#0x06000000

    ldr    r2,[r0]     @ bmp->w dans r2
    ldr    r3,[r0,#4] @ bmp->h dans r3 (accès avec déplacement +4, r0 non modifié)
    add    r1,r0,#8    @ adresse de bmp->img dans r1
    mov    r0,r4      @ r0 n'est plus utile : mettre adresse dest dans r0 pour pouvoir utiliser code précédent

diBoucleV: ...idem code précédent
```