28/11/2014

Compte-rendu [MI01]

TP n°5

MOULIN MAHTIEU
LAVIOLETTE ETIENNE

1. Le cadre de pile

	Schéma de la pile (ESP)	
	31 *0*	
	… … …	
EBP+28	img_dest	*Empilement dernier paramètre*
EBP+24	img_temp2	*Empilement avant-dernier paramètre.*
EBP+20	img_temp1	*Et*
EBP+16	img_src	*Ainsi*
EBP+12	bi_height	*De*
EBP+8	bi_width	*Suite.*
EBP+4	&ret_fonction	*Adresse de retour de la fonction (suite au call)*
-> EBP=ESP	EBP	*Création cadre de pile --> Toutes les références à la pile dans le programme se feront en fonction d'EBP*
EBP-4	ebx	
EBP-8	esi	
EBP-12	edi	
EBP-16	EAX	*Sauvegarde les registres utilisés dans la fonction*
EBP-20	EBX	…
EBP-24	EDX	…
		…

MEMO : Les pop en fin de fonction suppriment respectivement EDX, EBX, EAX, edi, esi, ebx puis ebp et le return renvoie la valeur contenue dans EAX (convention C)

```
        push    ebp                     ;empilement de ebp
        mov     ebp, esp                ; ebp=esp

        push    ebx                     ; empilement de ebx
        push    esi                     ; empilement de esi
        push    edi                     ; empilement de edi

        mov     ecx, [ebp + 8]          ; ecx = bi_width
        imul    ecx, [ebp + 12]         ; ecx = bi_width * bi_height
                                        ; ecx correspond au nbre total de pixel
        mov     esi, [ebp + 16]         ; esi = img_src
        mov     edi, [ebp + 20]         ; edi = ims_temp1
```

A l'appel de la procédure, on stocke la valeur du pointeur de pile dans ebp afin d'utiliser ce registre comme référence pour lire les données stocké dans la pile, entre autres les paramètres de la fonction comme indiqué dans le cadre de pile ci-dessus. De plus, on a :

- ECX contient le nb total de pixel de l'image (height * width)
- ESI contient les pixels de l'image source
- EDI contient les pixels de l'image temporaire 1

2. Structure itératives pour parcourir l'image

Le principe est de parcourir l'ensemble des pixels en partant du dernier jusqu'au premier. En effet, ecx contient le nombre de pixel de l'image, ce qui correspond au numéro du dernier pixel. Ainsi, à chaque itération on décrémente ecx et on stocke le pixel dans edx. In pixel étant codé sur 4 octet, il faut stocker dans EDX l'adresse ESI+ECX*4. Une fois le traitement effectué, on l'enregistre dans l'image de destination dont les pixels sont stockés dans EDI. Ainsi on remplace le pixel en EDI+ECX*4 par celui traité et contenu dans EBX

```
DEC ECX    ; on parcours les pixels du dernier au premier
MOV EDX, [ESI + ECX*4] ; on recupere l'adresse du pixel

MOV [EDI + ECX*4], EBX ; on enregistre le pixel dans l'image suivante
CMP ECX, 0 ; on test si on est arrivé au pixel numéro 0
JNE traitement  ; on repete tant qu'il reste des pixels
```

Calculs

Nous multiplions toujours un chiffre entier par un flottant dont la partie entière est nul. Pour cela, il suffit juste de multiplier la partie entière du premier par la partie décimale du second et de faire un décalage de la virgule ensuite (exemple en base 10 : 3*0.4, on fait 3*4=12 et on décale la virgule d'un cran soit 1.2).

La conversion en niveau de gris est donné par la formule suivante : I = R*0.299 + V*0.587 + B * 0.114. Les constante R, V et B étant les valeurs de rouge, vert et bleu du pixel.

Ainsi pour chaque pixel, on va traiter le résultat pour chaque couleur séparément et les multiplier au fur et à mesure.

- On récupère en premier la valeur de **bleu** correspondant au pixel en court, qui correspond au deux dernier octet. Pour le récupérer on fait donc pixel AND 0000FF.
- On multiplie cette valeur par 0.114 * 256 = 29 pour effectuer des multiplications entières
- On récupère en deuxième la valeur de **vert** correspondant au pixel. Pour le récupérer on fait donc pixel AND 00FF00.
- On décale ensuite le résultat vers la composante bleue.
- On multiplie cette 0.587 * 256 = 150 pour effectuer des multiplications entières
- On récupère en deuxième la valeur de **rouge** correspondant au pixel. Pour le récupérer on fait donc pixel AND FF0000.
- On décale le résultat vers la composante bleue
- On multiplie cette 0.299 * 256 = 77 pour effectuer des multiplications entières

3. Code assembleur correspondant

```
; IMAGE.ASM
; MI01 - TP Assembleur 2 a 5
; Realise le traitement d'une image 32 bits.
.686
.MODEL FLAT, C
.DATA
.CODE
; ************************************************************************
; Sous-programme _process_image_asm
;
; Realise le traitement d'une image 32 bits.
;
; Entrees sur la pile : Largeur de l'image (entier 32 bits)
;           Hauteur de l'image (entier 32 bits)
;           Pointeur sur l'image source (depl. 32 bits)
;           Pointeur sur l'image tampon 1 (depl. 32 bits)
;           Pointeur sur l'image tampon 2 (depl. 32 bits)
;           Pointeur sur l'image finale (depl. 32 bits)
; ************************************************************************

PUBLIC      process_image_asm

process_image_asm PROC NEAR       ; Point d'entree du sous programme

      push    ebp
      mov     ebp, esp

      push    ebx
      push    esi
      push    edi

      mov     ecx, [ebp + 8]          ; biWidth
      imul    ecx, [ebp + 12]         ; biWidth * biHeight

      mov     esi, [ebp + 16]         ; img_src
      mov     edi, [ebp + 20]         ; img_tmp1

      ;*****************************************************************
      ;*****************************************************************

      PUSH EAX                        ; sauvegarde des parametres
      PUSH EBX                        ;
      PUSH EDX                        ;

traitement:

      DEC ECX                         ; nouvel index de pixel

      MOV EDX, [ESI + ECX*4]          ; edx = @ pixel

      MOV EAX, EDX                    ; copie du pixel dans eax
      AND EAX, 000000FFh              ; masque pour B
```

```
      IMUL EAX, 29                      ; multiplication 0.114*256 = 29
      MOV EBX, EAX                      ; sauvegarde du résultat pour bleu dans ebx

      MOV EAX, EDX                      ; copie du pixel dans eax

      AND EAX, 0000FF00h                ; masque pour G
      SHR EAX, 8                        ; decalage vers le bleu
      IMUL EAX, 150                     ; multiplication 0.587*256 = 150
      ADD EBX, EAX                      ; sauvegarde dans EBX de la somme pour B + G

      MOV EAX, EDX                      ; copie du pixel dans eax
      AND EAX, 00FF0000h                ; masque pour R
      SHR EAX, 16                       ; decalage vers le bleu
      IMUL EAX, 77                      ; multiplication 0.299*256 = 77
      ADD EBX, EAX                      ; sauvegarde dans EBX = somme pour B + G + R

      SHR EBX, 8                        ; on supprime le decalage sur EBX (stockage
dans le bleu)

      MOV [EDI + ECX*4], EBX            ; enregistrement dans img_temp1
      CMP ECX, 0
      JNE traitement                    ; on repete jusqu'au dernier pixel

      POP EDX                           ; on remet les registre dans l'état
      POP EBX                           ; initiale, cad avant l'appel
      POP EAX                           ;

      ;*****************************************************************
      ;*****************************************************************

fin:
      pop     edi
      pop     esi
      pop     ebx

      pop     ebp

      ret                               ; Retour a la fonction MainWndProc

process_image_asmENDP

   END
```

4. Résultat

Le code assembleur est plus performant que le code en C. Les résultats sont :

- Assembleur = 148 ms
- Langage C = 534 ms

La différence de temps est considérable. L'assembleur étant un langage plus proche du processeur, celui interprète plus vite les instructions et l'exécution en est plus rapide. C'est très utile pour un logiciel de traitement d'image qui nécessite souvent de lourds et couteux calculs.