Le processeur 386

L'acteur principal d'un ordinateur est bien sur le processeur. L'Intel 386 est un processeur 32-bits moderne. Il dispose de deux modes de fonctionnement radicalement différents, répondant chacun à deux besoin précis.
Le premier mode, et certainement le plus employé, du moins jusqu'à présent, s'appelle le mode réel (***Real mode*** en Anglais). La raison d'être de ce mode est la sacro-sainte compatibilité ascendante. Il est hérité des premiers jours du PC et du DOS. Dans ce mode le 386 se comporte comme un 8086 très rapide.
Le deuxième mode est le mode protégé (***Protected mode*** en Anglais). Ce mode est destiné à permettre la réalisation de systèmes d'exploitation modernes, où multitâche, capacité et performances en sont les caractéristiques essentielles.
Nous examinerons dans ce chapitre comment utiliser de manière experte les capacités du 386. Nous parlons du 386 mais ne délaissons pas pour autant la générations suivante que sont les 486. Ces deux processeurs sont fonctionnellement identique. Toutefois, le 486 possède un surcroît de puissance dû, entre autre, à 8Ko de mémoire cache qui lui permettent de diminuer le nombre de cycles machine nécessaires à l'exécution de la plupart des ses instructions.
D'autre part, développer du code 386 rapide et compact nous obligera à optimiser les algorithmes et le code. Le moteur exécuté sur 486 ne posera pas de problème de performance!

Détection du 386

Afin d'être sûr que nous sommes bien en présence d'un PC muni d'un 386 au minimum, nous allons écrire une routine de détection de processeur.
Depuis le 8086, toutes les générations suivantes ont bénéficié de nouvelles fonctionnalités, et donc denouvelles instructions tout en restant compatible avec les générations précédentes. De ce fait, le 286 possède des instructions que le 86 ne connait pas, le 386 exécute des instructions que ne connait pas le 286, etc...
Le principe de détection est basé sur les instructions nouvelles disponibles dans le 386. A partir du 80186, les processeurs possède un mécanisme de déclenchement d'exception qui leurs permettent de signaler au système d'exploitation une erreur de code. Charge à lui de stopper l'application fautive ou éventuellement de corriger le problème. Cette exception est matérialisée par l'interruption numéro 6.
Afin de savoir si le processeur en cours est bien un 386 ou un modèle supérieur, nous allons lui faire exécuter une instruction spécifique au 386 et regarder si une interruption 6 a été déclenchée.
Toutefois, seul le 8088 ne possède pas ce type de mécanisme. En conséquence, si vous lui faites exécuter une instruction invalide, la seule chose qu'il sache faire est de planter! Un filtre particulier doit lui être imposé.
Pour savoir si le processeur est un 8088 ou 8086, ...
Voici donc le code de détection complet. Il retourne 0 si un 386 n'est pas présent.

```
;short CPU386 ()
;////////////

global @CPU386$qv : proc

proc    @CPU386$qv

 pushf
```

```
 xor     dx,dx

 ; 486, 386 or 286.

 xor     ax,ax
 push    ax
 popf
 pushf
 pop     ax
 and     ax,0f000h
 cmp     ax,0f000h
 je      @@_Exit

 ;-- Tester s'il s'agit d'un i486, i386 ou 80286  ----

 mov     dl,CPU286
 mov     ax,07000h
 push    ax
 popf
 pushf
 pop     ax
 and     ax,07000h
 je      @@_Exit; Ce n'est pas un 386.

 inc  dl             ; C'est un 386 ou superieur.

@@_Exit:
 xor     dh,dh
 mov     ax,dx
 popf
 ret
endp
```

16-bits, 32-bits? C'est quoi la différence?

Afin de mettre en évidence les problèmes inhérents aux codes 16-bits, nous allons écrire une routine d'affichage simplifiée de sprite.

Plantons le décor. On va afficher un gros sprite, un boss de fin de niveau par exemple. Supposons que l'on travaille en résolution SVGA 640x480 en 256 couleurs. Dans une telle résolution, un pixel est codé sur 8-bits. Chaque écran comporte 640x480 octets soit 300Ko. Notre boss, occupe disons 1/4 d'écran (un gros sprite), soit 300Ko/4 = 75Ko.

Dans le mode réel on adresse les données via le segment de données DS. Il doit donc pointer sur le paragraphe correspondant au début de la zone mémoire où est stockée l'image bitmap de notre boss. A l'aide du registre SI on va parcourir l'image et afficher tel quel les pixels à l'écran. SI peut balayer une plage de 0 à 0ffffh octets (valeur maxi d'un registre 16-bits). Or notre sprite dépasse 64Ko. Que faire alors des pixels restants?

La méthode traditionnelle, imposée par l'architecture de segmentation d'Intel, est de modifier le registre DS en cours de route après l'affichage des 64Ko premiers pixels du sprite.

Figure 2: Code 16-bits d'affichage d'un sprite

```
; Code   Obs                        Cycles|Taille

dataseg
 Boss    dw ??   ;Segment.
         dw ??   ;Offset.
```

```
codeseg
...
;...DS:SI doit pointer sur l'image.
lds     si,[dword ptr Boss]     ;                       7|2
mov     ax,0A000h               ; VGA                   2|2
mov     es,ax                   ;                       2|2
xor     di,di                   ; di=0                  2|2
mov     cx,10000h/4             ; 64Ko                  2|2
rep     movsd                   ; Copie          . 4*4000|2
mov     ax,ds                   ;                       2|2
add     ax,0fffh                ; Pointe seg            3|2
mov     ds,ax                   ; suivant.              2|2
xor     si,si                   ; offset à 0.           2|2
xor     di,di                   ; Debut écran.**        2|2
mov     cx,2C00h/4              ; Le reste.             2|2
rep     movsd                   ; Copie.  5+4*B00|2
...
end
```

*** Nota: Dans cet exemple on ignorera que l'écran SVGA n'adresse que 64Ko à la fois (un prochain chapitre discutera spécialement des problèmes d'affichage et de banking). Ici on revient simplement au début de l'écran, et on écrase le début du sprite.*

La Figure 2 représente le code de la routine d'affichage en 16-bits. On y vérifie que la gestion des segments incombe bien au programmeur et que ce travail s'ajoute à celui nécessaire à la gestion des adresses.
Que peut apporter le modèle 32-bits? Imaginons qu'Intel ai conçu non pas le 386, mais le 386/32! Ce 386/32 posséderai comme avantage sur le 386 standard d'avoir des segments adressables sur 32-bits, d'un coup d'un seul, et ceci sans quitter ce bon vieux mode réel. Réécrivons donc, en Figure 3, notre code 32-bits pour le processeur 386/32.

Figure 3: Code 32-bits d'affichage de sprite

```
; Code              Obs       Cycles|Taille
dataseg
 Boss    dd ??    ; Adresse 32-bits.

codeseg
 ...
 xor     ax,ax                                          ;                       2|2
 mov     es,ax                                          ;es=0.                  2|2
 mov     esi,[Boss]                                     ;                       2|2
 mov     edi,0A0000h                                    ;VGA 32-bits.           2|2
 mov     ecx,12C00h/4                                   ;75Ko d'1 coup.         2|2
 rep     movs [dword ptr es:esi],[dword ptr es:edi]     ;Copie.          5+4*1B00|2
 ...
end
```

** Nota: Le nombre de cycles correspondant au temps passé par le processeur pour transférer les octets de l'image n'est pas pris en compte. Il ne dépend pas de l'efficacité du code, mais seulement de la taille du sprite (Identique dans les deux sources).*

Nous avons utilisé réellement des registres 32-bits, avec une segmentation 32-bits afin de recopier en une seule instruction de chaîne le Boss en entier. Nous n'avons plus eu à gérer les segments et les résultats s'en ressentent aussi bien en terme de performance que de taille de code (Voir Figure 4).

Figure 4: Comparaison des performances 16/32-bits

Type de Code	Taille en octets	Vitesse* en cycles
16-bits	26	33
32-bits	12	15

Du fait de la disparition de la gestion des segments et de l'utilisation des aptitudes 32-bits du 386 nous gagnons plus du double de vitesse et le tiers de taille de code!

Mais, je m'emballe, je m'emballe... c'est beau, c'est performant, mais le 386/32 n'existe pas !?!.. Pourtant, un bug dans le 386, relaté par Al Williams en Janvier 1990 dans *l'excellentissime* revue internationale Doctor Dobb's Journal (DDJ), va nous permettre de réaliser le 386/32, et dans tirer tous les bénéfices.

L'aspect 32-bit du 386

Examinons rapidement le 386 et ses registres 32-bits:
Les registres généraux sont: EAX, EBX, ECX, EDX - le E signifiant Extended. La partie basse de chacun des registre est contenue dans les registres 16-bits du x86 (Resp. AX,BX,CX,DX).

Registres généraux du 386															
	AH	AL		BH	BL		CH	CL		DH	DL				
	AX			BX			CX			DX					
EAX			EBX			ECX			EDX						

Quatre registres 32-bit à usage spécial IP, EBP, ESP et ESI et EDI.

Registres spéciaux du 386							
	IP		SI		DI		BP
EIP		ESI		EDI		EBP	

Ainsi que les registres 16-bits de segmentation/séléction: DS, ES, SS, CS, et les deux nouveaux FS et GS.

Registres de segment/selecteurs du 386					
DS	ES	SS	CS	FS	GS

Tous ces registres ont un rôle identique à celui décrit plus haut pour le 8086.
Si la valeur maxi d'un registre 16-bits est bien 0ffffh (64Ko), celle d'un registre 32-bits est 0ffffffffh (4Go).

386 en mode protégé 32-bits

Afin de mettre en oeuvre l'adressage à plat en mode réel, nous devons d'abord examiner le mode protégé du 386.

Protection des segments

Ce mode est dit protégé, car il permet de créer des segments de mémoire qui seront étanches au débordement.

Reprenons notre exemple. Nous affichons les pixels du Boss sur l'écran grâce via un segment de 64Ko réservé aux afficheurs VGA et SVGA. Ce segment se trouve au paragraphe 0A000h, correspondant à l'adresse physique 32-bits 0A0000h.

Avant et après ce segment, la mémoire disponible sur le PC est utilisée par d'autres adaptateurs de périphérique. Par exemple, juste après le segment VGA, il se trouve le segment 0B000h, qui a été réservé par la société Hercules, Inc, créateur de la célèbre carte graphique Hercules, au début du PC (Qui s'en souvient?).

Si notre programme était boggué, et que nous avions écrit après le segment 0A000h, nous aurions empiété sur la zone Hercules, et ceci avec la bénédiction du 386 en mode réel.
En mode protégé, il est possible de créer un segment VGA, débutant à l'adresse 32-bits, 0A0000h et de taille 1000h (64Ko pile). Le même bug aurai alors déclenché une exception processeur de type Faute de protection Générale (***General Protection Fault*** en Anglais) - ou ***GPF*** pour les intimes. Nous nous serions fait vidé par le 386. D'où la notion de mode protégé: chacun chez soi!

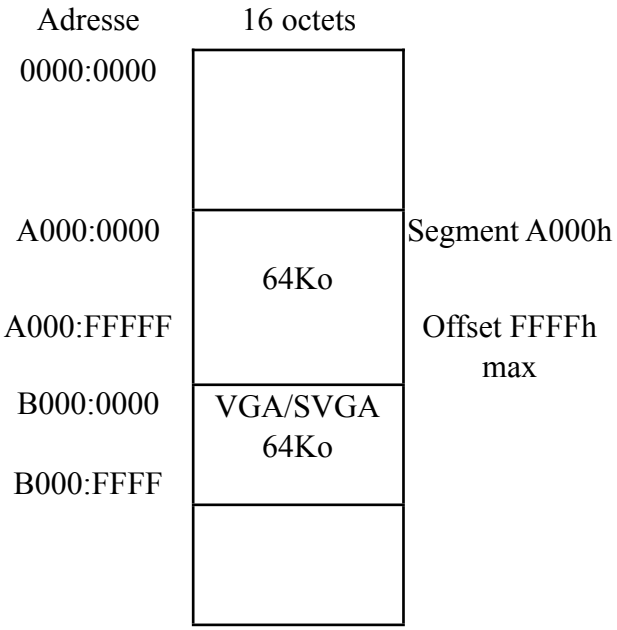
Modèle de mémoire réel

En mode réel, nous adressons la mémoire disponible grâce à un couple *Segment:Offset* appelé adresse far.

Segment est une valeur 16-bits qui représente l'emplacement du segment depuis le début de la mémoire. Ce décalage est donnée en paragraphes, ie. en unité de 16 octets, (ou granularité de 16 octet). Pour savoir à quel octet le segment se trouve il faut multiplier la valeur de segment par 16. Ainsi le segment VGA en 0A000h paragraphe correspond au 0A000h*16 = 0A0000h-ième octet depuis l'adresse 0.

Offset est le décalage à l'intérieur du segment, donné en octet (granularité de 1).
Ainsi pour calculer l'adresse 32-bit à plat correspondant au couple Seg:Ofs on applique la formule suivante: Seg*16+Ofs=Adresse 32-bits à plat. L'adressage physique effectif se fait donc sur 20-bits: c'est la fameuse barrière des 1Mo.

Figure 5: Modèle de mémoire en mode réel.



On peut remarquer que l'adressage Seg:Ofs implique une segmentation de la mémoire en bloc de 64Ko et que ces blocks peuvent se recouvrir tout les paragraphes. De ce fait, à plusieurs couple Seg:Ofs différents, peut correspondre la même adresse 32-bit à plat. Exemple: 0A000h:0000h est identique à 9001h:fffffh et correspond à A0000h.

On remarquera aussi que tout couple Seg:Ofs est toujours une adresse valide sur un PC muni de 1Mo.

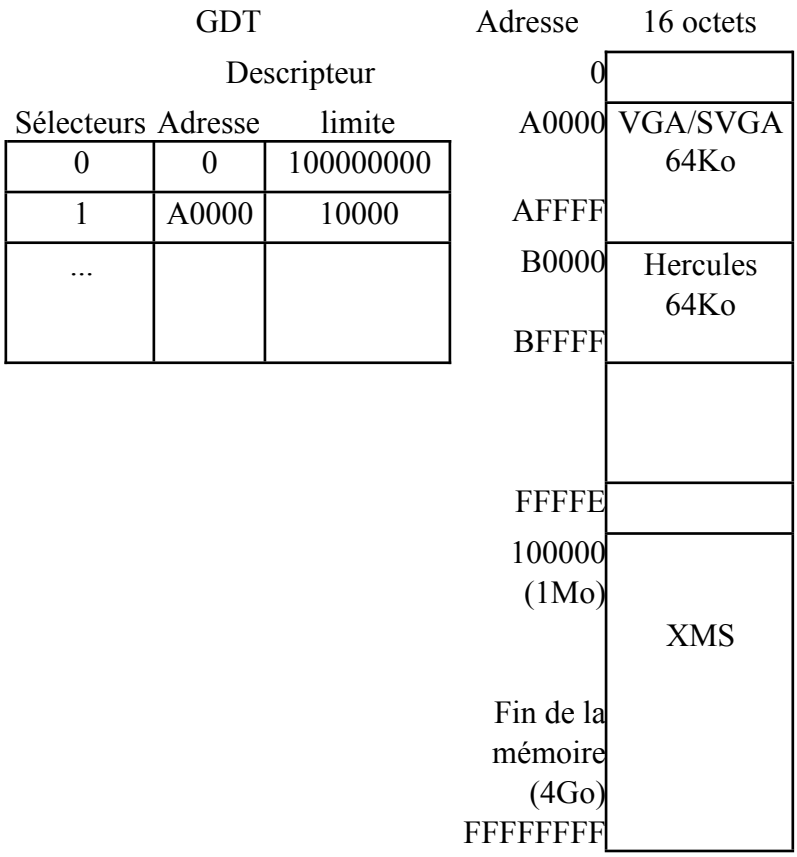
Modèle de mémoire protégé

En mode protégé, nous adressons la mémoire à l'aide du couple *Selector:Address* (voir Figure 6).

Selector est une valeur de 16-bits définissant là aussi un segment. Mais la valeur ne correspond plus à une adresse, mais à un indice de table, d'où le nom de selector. Nous en reparlerons.

Address est une valeur 16 ou 32-bits, de granularité 1, selon que le segment est un segment déclaré 16 ou 32-bits.

Figure 6: Modèle de mémoire protégée



Chaque segment est décrit au processeur par un *descriptor*. C'est une zone mémoire, de 24 octets, comportant des informations sur le segment (adresse de base du segment, un indicateur 16 ou 32-bits etc...).

L'ensemble des descripteurs de segments définis et valides est maintenu dans une table spécifique accessible au processeur: la *Global Descriptor Table* ou *GDT*.

Une seule GDT est géré par processeur, par contre plusieurs table locales de descripteurs (*LDT*) peuvent coexister sur un même processeur.

Un set de fonctions sont dévolue à l'usage et la maintenance de ces tables (Par exemple LGDT charge la table globale, etc...).

Un set de fonctions sont dévolue à l'usage et la maintenance de ces tables (Par exemple LGDT charge la table globale, etc...).

Contrairement au mode réel, tout couple Sel:Adr ne correspond pas forcement à une adresse valide puisque le sélecteur doit pointer sur une entrée valide de la GDT et, chaque segment ayant une limite, toute adresse qui dépasserai la limite du segment provoquerai un splendide GPF 0, même lors d'une lecture.

A cause de - ou grâce à - la compatibilité ascendante, le 386 peut gérer deux type de segments. Les petits (16-bits, 64Ko maxi) et les grand s (32-bits, 4Go maxi et désigné par l'attribut *Big* dans un descripteur).

Exploitation des grands segments

Il est donc facile, en mode protégé, de créer des segments de grande taille. Mais qu'en est-il du mode réel?
Or il se trouve que lorsque l'on passe du mode protégé au mode réel, le processeur ne fait aucune vérification quant à la taille des segments se trouvant en GDT. En fait, la documentation d'Intel spécifie simplement que les segments définis en mode protégé pour le mode réel doivent posséder une limite de 10000h, c'est-à-dire une taille de 64Ko.
En réalité, si l'on expérimente un peu, il est clair que des segments ayant une limite supérieure à 10000h de retour au mode réel ne provoque rien de malheureux. Au contraire, avec des instructions 32-bits portant sur des registres tels que ESI ou EDI, on peut parcourir un segment jusqu'à une taille de 4 Go!
Nous allons donc mettre en oeuvre cette incroyable fonctionnalité au travers d'un set de fonctions C. Elles nous permettrons de manipuler de grande quantité de mémoire.

Détection d'un gestionnaire XMS

Afin de ne pas gêner d'autres programmes qui utiliseraient aussi de la mémoire XMS, nous allons réserver notre espace par l'intermédiaire du gestionnaire XMS. La mémoire XMS débute à l'adresse 32-bit à plat 1000000h, et est en général géré par le gestionnaire HiMem.SYS livré avec le DOS ou Windows.
Il est donc nécessaire de vérifier si un tel gestionnaire est installé. Nous allons appeler cet hypothétique gestionnaire via l'interruption réservée 2fh. Avec la fonction 43h sous fonction 0, nous saurons qu'un gestionnaire est installé si al contient la valeur 80h au retour d'interruption. Si le driver est présent, nous récupérons son adresse grâce à la fonction 43h sous-fonction 1, et la stockons pour un usage ultérieur.

```
static void (far *XMSDriver)();     // Adresse du gestionnaire XMS.

short XMSHere (void)
//////////////
{
 asm {
  mov   ax,0x4300    // Vérifier si driver XMS présent.
  int   0x2f
  cmp   al,0x80      // Retourne 80h dans al si driver présent.
  je    _XMSOk
  xor   ax,ax        // Pas de driver installé.
  jmp   _Sort
_XMSOk:
  mov   ax,0x4310    // Récupère l'adressedu gestionnaire.
  int   0x2f
  mov   word ptr XMSDriver,bx // Stocke l'offset
  mov   word ptr XMSDriver+2,es     // stocke le segment
_Sort:
 }
}
```

Deux cas de figures se présentent alors à nous.
• Si aucun gestionnaire de mémoire XMS n'est présent, ce qui est de nos jours assez rare, nous pouvons décréter que notre segment démarre au début de la mémoire XMS en 1000000h.
• Si un tel gestionnaire est présent, nous allons lui demander d'allouer un block d'une taille donnée et récupérer son adresse.
•

```
short XMSEMBAlloc(unsigned short mysize, unsigned short *handle)
/////////////////
{
 asm {
  mov   ah,XAllocEMB          // function code
  mov   dx,mysize             // number of K to allocate
  call  dword ptr XMSDriver   // call the XMS driver
  les   di,handle
  mov   word ptr es:di,dx     // save handle number
  xor   ah,ah
  mov   al,bl                 // transfer XMS error code to al
 }
}
```

Enfin, la fonction de libération du block permettra de relacher ce block pour pouvoir redevenir disponible pour les éventuelles autres applications.

```
unsigned char XMSFreeEMB(XMSHandle handle)
////////////////////////
{
 asm {
 XMSOk
  mov   ah,XFreeEMB                // function code
  mov   dx,handle                  // handle number to free
  call  dword ptr XMSDriver        // call the XMS driver
  mov   al,bl                      // transfer XMS error code to al
_exit:
 }
}
```

Construction du GDT

Pour nos besoins, nous allons donc créer un seul segment de données de grande taille logé au-delà de la barrière des 1Mo, dans la mémoire XMS. Nous allons nous attacher à déclarer ce segment au processeur.
La structure C++ d'un descripteur est la suivante:

```
struct TDescriptor
/////////////////
{
 unsigned short _lo_limit;         // Les deux octets de poids faible de la
limite.
 unsigned short _base;             // La base de l'adresse.
 unsigned short _access;           // Le flag d'accès
```

```
 unsigned short _hi_limit;          // Les deux octets de poids fort de la
limite.
};
```

Nous commençons par construire la nouvelle GDT, qui contiendra deux entrés. Le slot 0 est inutilisable, donc notre segment sera déclaré dans le slot numéro 1.
D'où GDT elle-même:

```
TDescriptor GDT[2] =
{
 {0,0,0,0},                    // Inutilisé.
 {0xFFFF,0,0x9200,0x8F}        // Notre Grand Segment.
};
```

Quelques explications doivent être données. Toutd'abord la limite de note segment est ici 0x8fffff donnée en 4-octets puisque nous avons mis la valeur 0x9200 dans le champs _access.

Afin de valider notre GDT nous construisons le pointeur FWORD (4-octets) du GDT.

```
//...FWORD pointer to GDT.

struct fword
////////////
{
 unsigned short _limit;             // Limite du GDT.
 unsigned long _linear_address;     // Adresse 32-bits à plat du GDT.
};



static fword  pGDT;  // pointeur FWOR sur notre GDT.
```

Nous calculons l'adresse 32-bits du GDT dans pGDT. Cette opération est réalisée par l'appel de la fonction S4Seg2Lin qui suit. Son inverse est aussi founis.

```
unsigned long S4Seg2Lin (void far *p)
//////////////////////
{
 //...Convertion d'un pointeur far en adresse 32-bit flat.

 return (((unsigned long)FP_SEG(p))<<4)+FP_OFF(p);
}

void far *S4Lin2Seg (unsigned long lin)
///////////////////
{
 //...Convertion d'une adresse 32-bits flat en pointeur far.

 void far *p;
 FP_SEG(p)=(unsigned int)(lin>>4);
 FP_OFF(p)=(unsigned int)(lin&0xF);
 return p;
}
```

Puis, pour préparer la validation du GDT, nous allons interdire à toute interruption masquable ou pas, de déranger le 386 pendant l'opération.

```
// Keyboard controller defines.

#define RAMPORT   0x70
#define KB_PORT   0x64
#define PCNMIPORT 0xA0
#define INBA20    0x60
#define INBA20ON  0xDF
#define INBA20OFF 0xDD


void S4Extend (TGDT *gdt, unsigned limit)
/////////////
//...Adjust the GS register's limit to 4GB
{
 //...Calcul de l'adresse linéaire 32-bit du GDT.

 gdtptr.linear_add=S4Seg2Lin((void far *)gdt);
 gdtptr.limit=limit;

 //...Stope les intéruptions maquables

  _disable();

 //...Stope les interrutions non masquable (NMI).

 outp(RAMPORT,inp(RAMPORT)|0x80);

 //...Basculement en mode protégé, validation du GDT et retour au mode réel.

 S4ProtSetup(&gdtptr);

 //...Autorise à nouveau les interrutions masquables..

 _enable();

 //...Ainsi que les NMIs.

 outp(RAMPORT,inp(RAMPORT)&0x7F);
}
```

Enfin nous passons au mode protégé puis validons la GDT et retournons au mode réel.

```
;void S4ProtSetup (fword *gdt)
;////////////////
proc S4ProtSetup
 arg    @@fpointer:dword

 push   ds
 lds    bx,[@@fpointer]
 lgdt   [fword ptr bx]             ; Charge la GDT
 pop    ds

 mov    eax,cr0                   ; Bascule en mode protégé.
 or     al,1
 mov    cr0,eax
```

```
 jmp    short @@_PurgePrefetch     ; Vide le pipe de décodage d'instruction

@@_PurgePrefetch:                  ;du 386. Ceci évite un crash éventuel.

 mov    bx,8                       ; Charge GS et ES.
 mov    gs,bx
 mov    es,bx

 and    al,0feh                   ; Retourne au mode réel.
 mov    cr0,eax
 ret
endp S4ProtSetup
```

Notons toutefois, que le 386 possède un pipeline destiné à accélérer ses performances globale. Il procède en effet au chargement de la prochaine instruction depuis la mémoire, pendant qu'il exécute l'instruction en cours. Afin de ne pas provoquer un crash du a un mauvais décodage, nous effacons l'instruction préchargée simplement en effectuant un saut à l'instruction suivante.
Nous sommes donc maintenant muni de notre Grand Segment où nous allons pouvoir stocker toutes sortes de ressouces, comme les images, les sons échantillonnées, etc.
Afin de faciliter l'accés à cette zone depuis le langage C++, nous allons écrire deux fonctions de recopie entre la mémoire conventionnelle et la mémoire XMS.

```
;void S4CopyLin2Seg (void far *dest,unsigned long src, unsigned long sz)
;//////////////////
proc S4CopyLin2Seg
 arg    @@dest:dword, @@source:dword, @@count:dword
 uses   es, si, di

 mov    esi,[@@source]

 xor    edi,edi      ; Converition far pointeur à 32-bit flat.
 les    di,[@@dest]
 xor    eax,eax
 mov    ax,es
 shl    eax,4
 add    edi,eax

 mov    ecx,[@@count]

 xor    ax,ax        ; Zero es for 32-bit programming..
 mov    es,ax

 cld
 rep    movs [byte ptr es:esi],[byte ptr es:edi]
ret
endp S4CopyLin2Seg

;void S4CopySeg2Lin (unsigned long dst,void far *,unsigned long sz)
;//////////////////
proc S4CopySeg2Lin
 arg    @@dest:dword, @@source:dword, @@count:dword
 uses   es, si, di

 mov    edi,[@@dest]
```

```
 xor    esi,esi      ; Convertion de l'adresse far en 32-bit flat.
 les    si,[@@source]
 xor    eax,eax
 mov    ax,es
 shl    eax,4
 add    esi,eax

 mov    ecx,[@@count]

 xor    ax,ax        ; Adressons notre segment.
 mov    es,ax

 cld
 rep    movs [byte ptr es:esi],[byte ptr es:edi]
 ret
endp S4CopySeg2Lin
```

Afin de nous permettre d'atteindre les adresses au delà de la barrière de 1Mo, nous devons annuler l'adressage en rouleau de la mémoire. A l'époque où l'AT est sortie, pour conserver la compatibilité totale avec le 8086, les ingénieurs de chez Intel ont annulé, par défaut, la possibilité d'adressage du 286 au delà les 1Mo. Il nous faut donc rétablir cette option, qui est piloté par la ligne A20 du bus d'adressage à travers le controleur de clavier! La fonction S4A20() rétabli le mode rouleau si le paramètre est 0, sinon permet l'adressage au delà des 1Mo.

```
void S4A20 (int flag)
//////////
{
 while (inp(KB_PORT)&2);      // Attend un caractère disponible.

 outp(KB_PORT,0xd1);
 while (inp(KB_PORT)&2);      // Attend un caractère disponible.

 outp(INBA20,flag?0xdf:0xdd);
 while (inp(KB_PORT)&2);      // Attend un caractère disponible.

 outp(KB_PORT,0xff);
 while (inp(KB_PORT)&2);      // Attend un caractère disponible.
}
```