Assembleur Intel avec NASM

par Etienne Sauvage (Assembleur Intel : découverte.)

Date de publication : 18 avril 2011

Dernière mise à jour :

A la découverte de l'assembleur.

I - De l'assembleur

Relu par **ClaudeLELOUP**.

Aujourd'hui, il existe une multitude de langages de programmation aux caractéristiques fort différentes. Des langages-objets, interprétés, portables, etc. se rencontrent quotidiennement. A tel point qu'il faut faire une totale confiance au compilateur pour traduire cela dans la langue de la machine. Car bien évidemment, un ordinateur ne comprend ni le C, ni le C++, encore moins le java, le CamL et autres PHP, Python et consorts. Aussi le curieux peut légitimement se poser cette question : que se passe-t-il vraiment dans le ventre de la machine ? Je joue en ce moment avec le langage de programmation le plus proche de la machine, celui à l'origine de tout : le langage assembleur. Je vous propose de jouer avec moi.

I.1 - Du langage machine

Un ordinateur ne comprend rien, à l'origine. Que se passe-t-il à la mise sous tension ? L'ordinateur vient brutalement à la vie. Et sa vie, à l'ordinateur, consiste à lire un livre. A la mise sous tension, il ouvre la première page de son livre et lit le premier mot. Et lire, pour un ordinateur, consiste à se modifier, à changer son état. Or donc, il lit son premier mot, et s'en trouve changé : son mot à lire n'est plus le premier, il devient le deuxième. Il a certaines de ses variables modifiées en fonction du mot lu. Un mot du livre de l'ordinateur, c'est quoi ? C'est un ensemble de fils électriques, sous tension ou pas. Vu qu'on ne s'amuse pas à ajouter ou enlever des fils à tout bout de champ, il s'ensuit que tous ces mots ont le même nombre de lettres, c'est-à-dire de fils. Nous avons ordonné ces fils en leur donnant un numéro. Et on dit que 1 est un fil électrique sous tension, et 0 un fil électrique sans tension. Un mot de l'ordinateur est donc représenté par la suite ordonnée des 0 et des 1 représentant l'état de tension de chaque fil électrique. Et une suite ordonnée de chiffres, ça donne un nombre. 86 est un mot du livre de l'ordinateur. D'un strict point de vue formel, chaque type de processeur doit avoir son propre dictionnaire. 86 n'a a priori aucune chance de signifier la même chose pour un processeur de marque AMD et pour un autre de marque Intel. 86 peut faire une addition de deux variables chez l'AMD, et afficher à l'écran la biographie des rois de France sur l'Intel. Certains se sont alors mis d'accord sur (on a aussi la version : certains ont imposé) un dictionnaire commun. Ce dictionnaire commun, c'est (une partie) de la norme Compatible PC. 86 veut dire (à peu près) la même chose chez tous les ordinateurs compatibles PC. Ca implique quelques bricoles, comme :

- n'importe qui peut écrire un livre pour n'importe quel ordinateur ;
- on n'est pas obligé de réécrire toutes les bibliothèques pour tous les ordinateurs ;
- on ne peut pas mettre n'importe quelle fonction dans un processeur.

Bien. L'informatique a avancé. Chaque nombre est une fonction à peu près partagée par un sous-groupe d'ordinateurs. Extraordinaire. Et si, je dis ça je ne dis rien, mais si, comme ça, histoire que moi je comprenne l'histoire qu'il y a dans le livre de l'ordinateur, on remplaçait, discrètement, tous ces nombres par les noms des fonctions qui leur correspondent ? Dites ? Et de ce jour, le désassembleur était né : traduire en mots plus humains les mots-nombres de l'ordinateur. Et si on faisait l'opération inverse ? Et si j'écrivais les noms des fonctions, et qu'après, on les traduisait en langue ordinateur ? Hou là, très compliqué, c'est de l'analyse de texte, on ne sait pas faire. Ce qu'on peut faire, par contre, c'est associer chaque mot ordinateur avec un tout petit mot d'anglais abrégé. Là, on devrait s'en sortir. C'est de ce modeste objectif que naquit l'assembleur. L'assembleur, c'est le premier langage de programmation différent du langage de la machine.

I.2 - De l'assembleur par rapport au langage machine

L'assembleur est donc une traduction du langage machine. Et comme toutes les traductions depuis la tour de Babel, elle n'a aucune raison d'être la même suivant les traducteurs. Et, de fait, l'assembleur diffère selon les programmes de traduction. Ceci implique deux trois petits soucis, notamment le fait que pour que l'ordinateur fasse exactement la même chose, il faut lui demander parfois différemment. C'est ennuyeux. Aussi aime-t-on, comme les chefs d'Etat, garder le même traducteur, qui traduira toujours de la même façon. Ces traducteurs traduisent une langue constituée de mots d'anglais abrégés en une langue constituée de nombres. Ces deux langues sont définies et existent indépendamment du traducteur. Celle que nous voulons traduire, celle constituée de petits mots d'anglais

abrégés, est un dictionnaire de mnémoniques. Ces mnémoniques font aussi partie de la norme Compatible PC. La différence entre les traducteurs tient à la façon de les arranger entre eux. Des traducteurs, il en existe quantité. Il faut en choisir un. Affaire de goût. Mais si on souhaite coller au plus près de la machine, et tout produire soi-même parce qu'on fait mi-moyen confiance aux autres compilos, ou parce qu'on veut bien comprendre ce qui se passe dans la machine, il faut en choisir un qui ne fait qu'assembler (i.e. traduire), sans autres fioritures. Personnellement, j'ai choisi NASM. Donc la suite se passera avec NASM. NASM peut, si j'ai tout bien compris, produire des fichiers exécutables pour Windows et Linux.

I.3 - Des fichiers exécutables

Justement, qu'est-ce qu'un fichier exécutable ? Sous le prétexte un peu subtil d'organiser tout le schmilblick qu'on peut croiser dans le disque dur d'un ordinateur (données d'Excel, vidéo du petit dernier bavant sur mamie, code de calcul de la bombe atomique), de fourbes ingénieurs en informatique (dont certains étaient même ingénieurs de recherche !) ont décidé, comme ça, qu'on allait avoir des fichiers en lieu et place d'une bouillie d'octets, et que le seul livre que l'ordinateur ouvrirait jamais s'appellerait "le système d'exploitation". Ces gens-là n'ont aucune poésie. Le système d'exploitation sait quels sont les fichiers disponibles sur l'ordinateur, et quels sont ceux qui sont des programmes. Il peut aussi ajouter des chapitres dans son propre livre, c'est-à-dire faire exécuter d'autres programmes à l'ordinateur. Comme tout est fichier (en gros) pour le système d'exploitation, cela signifie que certains fichiers sont des chapitres à ajouter à son livre : les fichiers exécutables. Afin que le système d'exploitation sache quelques bricoles quant à ces chapitres à ajouter, ces fichiers exécutables ont un en-tête donnant des informations aussi futiles que le nombre de pages du chapitre à ajouter, par exemple. Le traducteur ne sait pas écrire cet en-tête, à part pour un seul type de fichiers exécutables : le format COM, disponible sous DOS. Pour les gens qui n'ont pas eu la chance de naître avant Windows 95, DOS est le système d'exploitation sur lequel est construit Windows. Il est toujours accessible, bien que de plus en plus caché dans les tréfonds de l'ordinateur sous Windows. Donc, pour partir du plus simple possible, j'utilise **NASM** pour générer des fichiers COM.

I.4 - Des outils à utiliser

- **NASM**, donc.
- Un éditeur de texte.
  Personnellement, j'ai une tendresse particulière pour celui-ci, qui fait de la coloration syntaxique pour maints langages : **Notepad++**.
- Une fenêtre de commande DOS.
  Menu "Démarrer" ou "Windows", "Exécuter un programme", "cmd".

I.5 - Se jeter à l'eau du haut d'une falaise venteuse

Je n'y tiens plus, écrivons notre chapitre du livre de l'ordinateur. Ouvrons notre éditeur préféré, créons un nouveau fichier texte nommé comme on veut (personnellement il s'appelle "Carabistouille.txt") et tapons, avec la force de conviction que donne le bon droit :

```
org 0x0100 ; Adresse de début .COM
;Ecriture de la chaîne hello dans la console
mov dx, hello
mov ah, 0x9
int 0x21
ret
hello: db 'Bonjour papi.', 10, 13, '$'
```

Une fois sauvegardé, on lance une console, on se met dans le bon répertoire, et on commande :

```
nasm -o [un nom].com [nom complet du fichier texte]
```

Puis, en étant fou :

```
[un nom].com
```

Et boum. Ca formate le disque dur. Non, je blague. On a eu "Bonjour papi." affiché dans la console, une ligne vide, puis le retour de l'invite de commande.

Ceci est le plus petit programme qui fasse quelque chose : afficher une chaîne de caractères. Regardez la taille du fichier [un nom].com : 24 octets. On ne peut pas beaucoup plus petit : la chaîne de caractères comporte 16 caractères (Bonjour papi. + caractère 10 + caractère 13 + $), il ne reste que 8 octets de programme à proprement parler.

Maintenant, étudions ce que nous avons écrit ligne par ligne :

```
org 0x0100 ; Adresse de début .COM
```

Tout de suite, dès le départ, cette ligne n'est pas vraiment une ligne de code : c'est un message envoyé à NASM. **org** veut dire que tout ce qu'on va écrire n'est pas vraiment le chapitre : on laisse de la place pour l'en-tête du système d'exploitation. Un peu comme si on laissait de la place au début du chapitre pour la préface. **0x0100** est, aussi curieux que cela puisse paraître, un nombre. Il se décompose comme suit :

- **0x** indique qu'il s'agit d'un nombre écrit en hexadécimal ;
- **0100** est le nombre en hexadécimal.

En notation décimale, ce nombre est 256, soit la taille de l'en-tête. D'ailleurs, on peut tout à fait remplacer "org 0x0100" par "org 256", ça marche pareil. Ce qui nous apprend qu'on peut écrire un nombre en décimal en indiquant simplement le nombre, et l'écrire en hexadécimal en le préfixant par "0x". 0 = zéro, je précise.

Après, on a un point-virgule. Ceci indique à NASM que le reste de la ligne est du commentaire, du gloubi-boulga dont il n'a que faire mais qui peut nous aider, nous autres pauvres êtres biologiques. On notera également que visiblement, le nombre d'espaces entre les choses d'une ligne n'a pas d'importance. Toujours ça de pris.

```
;Ecriture de la chaîne hello dans la console
```

Commentaire, donc.

```
mov dx, hello
```

MOV, première instruction processeur. C'est L'instruction processeur, la grande, celle qui est utilisée à tout bout de champ. Elle déplace un nombre d'une case à une autre. Ici, on utilise la syntaxe Intel : destination, source. Si vous êtes passé par de l'assembleur Motorola ou AT&T, c'est l'inverse. On met donc le nombre contenu dans la case "hello" dans la case "dx". Trépidant. Alors, oui, les cases ont des noms. Ou pas. Disons que certaines cases particulières ont des noms, et qu'on peut affecter un nom aux autres, mais sinon, elles ont des numéros. Le nom "hello" correspond en fait à un numéro : le 264. Vous pouvez remplacer "hello" par 264, ça marche tout aussi bien. Pourquoi 264 ? C'est la taille du fichier exécutable moins la longueur de la chaîne de caractères plus la taille de l'en-tête : 24-16+256. Autrement dit, le numéro de l'octet de début de la chaîne une fois le programme chargé en mémoire. **DX** n'a pas de numéro, parce que c'est une case dans le processeur. Elle a un nom. C'est un registre. Cette instruction met donc dans le registre **DX** l'adresse du départ de la chaîne de caractères.

```
mov ah, 0x9
```

On met dans le registre AH la valeur 9, codée en hexadécimal. On peut la coder directement en décimal, ça marche aussi.

```
int 0x21
```

INT, deuxième instruction processeur. On demande au processeur d'appeler un petit programme stocké ailleurs, qui porte le numéro 0x21, mais bien connu du processeur. Il fait partie d'une catégorie de programmes qu'on appelle interruptions, d'où le mnémonique. C'est un programme du système d'exploitation, donc ça ne marche que sous

DOS. C'est SPE-CI-FIQUE à DOS. Ce programme fait des choses différentes selon la valeur stockée dans le registre AH. Si AH vaut 9, alors ce programme affiche la chaîne de caractères dont l'adresse du début est stockée dans DX. Voilà pourquoi on a mis l'adresse de notre chaîne de caractères dans DX, et 9 dans AH. Mais c'est bien gentil d'écrire la chaîne dont on a le début, mais la fin, elle est où ? Hé bien, la fonction 0x9 de l'interruption 0x21 s'arrête quand elle rencontre le caractère "$". C'est pour ça que notre chaîne se termine par un $ qu'on ne voit pas à l'écran.

```
ret
```

Fin du programme. C'est tout. Fin d'un programme COM.

```
hello: db 'Bonjour papi.', 10, 13, '$'
```

hello: marque une adresse. Cette case mémoire porte le nom "hello", dorénavant. Utile à NASM, pas au processeur. C'est pour nous éviter de nous balader avec des numéros de cases mémoire un peu partout. "hello" est quand même plus clair que **264**, non ? **db** indique qu'il s'agit de Data Bytes, d'octets de données. Pas d'autres choses : d'octets. Ensuite vient "Bonjour papi.", une virgule pour dire qu'on continue, le caractère numéro 10 (caractère ASCII = octet), le 13, et le $ terminal. Les guillemets servent à demander à NASM de traduire le caractère ASCII en sa valeur numérique. "B" est plus lisible que 66. Comme les caractères 10 et 13 ne sont pas affichables (il s'agit des caractères affichant un saut de ligne), on a indiqué directement leur valeur numérique.

Voilà, je crois qu'on a tout.

II - Assembleur : suite

Relu par **ClaudeLELOUP**.

II.1 - De la directive org

La première ligne de notre programme n'est pas, comme nous l'avons vu, une instruction du processeur. C'est un message destiné à NASM. On appelle ça une directive de compilation. Cette directive va demander à NASM d'ajouter un nombre (celui écrit juste après la directive) à toutes les adresses qu'on utilise dans le code. On peut donc, si on veut, le faire à la main et supprimer cette directive. Ca nous donne donc :

```
mov dx, hello + 256
```

Et on supprime la ligne de la directive. En compilant, on obtient le même résultat.

Néanmoins, cette directive est utile. Si on veut compiler notre programme dans un autre format que COM, ce décalage d'adresses n'aura plus lieu d'être. Plutôt que d'enlever dans toutes les adresses ce décalage, on modifiera uniquement la directive : beaucoup plus facile. Nous garderons donc cette directive, surtout qu'elle servira à introduire des décalages autres que celui du format COM.

II.2 - Du mnémonique INT

Le mnémonique **INT** appelle un sous-programme connu du processeur, appelé une interruption. Ces interruptions sont stockées en mémoire, quelque part. Le où ne m'intéresse pas pour l'instant. Il y en a de différentes sortes, dont deux qui m'intéressent tout de suite.

- Les interruptions fournies par le DOS.
- Les interruptions fournies par le BIOS.

L'interruption **0x21**, celle que nous avons utilisée, est une interruption DOS. Donc, ce code ne fonctionne pas sur un autre système d'exploitation. Par contre, il existe des interruptions BIOS. Le BIOS est le système d'exploitation fourni avec tous les PC sans exception. BIOS signifie, si je ne m'abuse, Basic Input Output System, système basique d'entrée et de sortie. C'est lui qui démarre l'ordinateur, parce qu'il est directement stocké sur la carte mère, et que quand on démarre l'ordinateur, le malheureux ne sait même pas utiliser son disque dur, sur lequel est stocké le vrai système d'exploitation. Donc, que vous ayez un ordinateur sous Windows, Linux, BeOS ou autre, le BIOS est là au démarrage. Et il est suffisamment standard pour en tirer parti, ce que nous allons faire pas plus tard que maintenant.

Alors, l'interruption du gestionnaire d'affichage vidéo porte le numéro **0x10**. Il existe une fonction qui permet d'afficher un seul caractère à l'écran. Hé oui, un seul. Il faut lui remplir plein de registres :

- le registre **AH** porte le numéro de la fonction, **0x0A** ;
- le registre **AL** contient le numéro du caractère à afficher ;
- le registre **BH** contient la page d'affichage, mettons 0 pour l'instant ;
- le registre **CX** contient le nombre de fois que l'on va afficher le caractère.

Pour l'essayer, tapons dans un fichier :

```
    mov ah, 0x0A
    mov al, 'B'
    xor bx, bx
    mov cx, 1
    int 0x10
    ret
```

Compilons, exécutons. Un B s'affiche. Mais ce n'est pas fabuleux. On a un B, mais pas de chaîne entière. La source de joie est que l'interruption fonctionne, et que nous nous sommes libérés de l'étreinte du système d'exploitation.

Néanmoins, nous avons là une nouvelle instruction : **XOR**, mnémonique de la fonction "ou exclusif". On l'utilise à la place de mov bx, 0 parce qu'elle permet une mise à zéro légèrement plus rapide.

II.3 - Des boucles

Ce qui serait bien, à présent, ce serait qu'on mette dans le registre **AL** le premier caractère de notre chaîne "Bonjour papi.", qu'on affiche, puis qu'on mette le deuxième caractère, puis qu'on affiche, puis qu'on mette le troisième et ainsi de suite. Ce qu'il ne faut pas oublier quand on programme, c'est : quand est-ce qu'on s'arrête ?

On ne va pas faire comme le DOS, qui s'arrête quand il rencontre un "$", parce que ça implique qu'on aura quelques difficultés à afficher un $. On va utiliser ce qui se fait dans d'autres langages plus évolués : on va marquer la fin de la chaîne par un caractère non affichable, le caractère justement appelé NULL, de valeur numérique 0.

Voyons le code :

```
org 0x0100 ; Adresse de début .COM

    ;Ecriture de la chaîne hello dans la console
    mov si, hello; met l'adresse de la chaîne à afficher dans le registre SI
    xor bh, bh; RAZ de bh, qui stocke la page d'affichage
    mov ah, 0x03
    int 0x10; appel de l'interruption BIOS qui donne la position du curseur, stockée dans dx
    mov cx, 1; nombre de fois où l'on va afficher un caractère
    affiche_suivant:
    mov al, [si];on met le caractère à afficher dans al
    or al, al;on compare al à zéro pour s'arrêter
    jz fin_affiche_suivant
    mov ah, 0x02;on positionne le curseur
    int 0x10
    mov ah, 0x0A;on affiche le caractère courant cx fois
    int 0x10
    inc si; on passe au caractère suivant
    inc dl; on passe à la colonne suivante pour la position du curseur
    jmp affiche_suivant
    fin_affiche_suivant:
    ret
    hello: db 'Bonjour papi.', 0
```

Plus dur, non ? Plusieurs choses ont fait leur apparition :

- des labels en plein milieu du code ;
- des crochets ;
- des instructions ;
- de nouveaux registres.

Je n'ai pas trouvé plus concis comme code. Si on peut, je suis preneur.

Les labels suivis par ":" sont des marqueurs d'adresse. NASM va les remplacer par leur véritable adresse, mais pour écrire dans le code, c'est bien plus clair comme ça.

Les crochets indiquent qu'il ne faut pas prendre la valeur du label, mais la valeur du contenu de la case mémoire dont l'adresse est la valeur du label. Il faut considérer en fait les labels comme des adresses. Quand on met des crochets, on indique qu'il faut prendre ce qu'il y a à cette adresse, et non l'adresse elle-même.

- L'instruction **OR** : c'est un "ou" logique. La seule valeur pour laquelle le "ou" logique vaut 0 avec deux fois le même opérande, c'est quand cet opérande est zéro. C'est la façon la plus rapide pour tester si une valeur est égale à zéro : si le résultat de ce "ou" vaut zéro, un bit spécial va être mis à 1 quelque part dans la machine, et pourra être utilisé par l'instruction suivante.
- L'instruction **JZ** : Jump if Zero. Si le bit évoqué au-dessus est à 1, le programme va continuer à l'adresse indiquée en paramètre. Sinon, on continue normalement.
- L'instruction **INC** : INCrémente. Ajoute 1 au registre indiqué en paramètre.
- L'instruction **JMP** : JuMP. Continue le programme à l'adresse indiquée en paramètre. Cette instruction est la base des boucles infinies tant redoutées des programmeurs.

Un processeur dispose d'une tripotée de registres, qui sont les seules cases mémoire qui n'ont pas d'adresse et sur lesquelles le processeur peut faire des opérations. Nous utilisons les registres généraux, qui contiennent des nombres et uniquement des nombres, et, ici, un registre d'adressage qui contient des adresses. Les registres généraux sont **A**, **B**, **C** et **D** pour l'instant. Ils se décomposent en registres haut et bas, en leur ajoutant **H** pour haut et **L** pour low (bas in eunegliche). Le regroupement des parties haute et basse se suffixe par **X**, pour euh... x (eXtended). On a donc **AX** composé de **AH** et **AL**, **BX** composé de **BH** et **BL**, etc. Le registre **SI** est un registre d'adressage, utile pour faire des calculs sur les adresses. On ne peut pas utiliser directement l'adresse, puisque "hello" ne contient que l'adresse du premier caractère de la chaîne. Il faut l'adresse de tous les caractères de la chaîne, et donc ajouter 1 à l'adresse courante à chaque fois qu'on affiche un nouveau caractère.

II.4 - Des soucis

Vous avez essayé de mettre un retour chariot ? C'est pas beau, non ? Pourquoi ? Parce que le BIOS est bête, et ne fait que ce qu'on lui demande : il affiche un caractère à l'écran. Si ce caractère n'est pas affichable, il fait ce qu'il peut ! Le caractère 13, notamment, est un caractère de mise en forme (dans l'esprit des gens, mais rien n'interdit de le considérer comme affichable). Donc, on peut s'améliorer en déplaçant le curseur d'une ligne vers le bas et en RAZant (de l'informaticien RAZer : effectuer une RAZ, Remise A Zéro) le numéro de colonne quand on rencontre ce caractère.

Ca donne (à insérer après le **ret**, histoire que ça ne s'exécute pas n'importe comment) :

```
nouvelle_ligne:
inc dh; on passe à la ligne suivante
xor dl, dl; colonne 0
jmp positionne_curseur
Et juste après le JZ, on insère :
cmp al, 13
je nouvelle_ligne
positionne_curseur:
```

Et ce sera tout.

Je vous refais le code en entier, pour la forme :

```
org 0x0100 ; Adresse de début .COM

;Ecriture de la chaîne hello dans la console
mov si, hello; met l'adresse de la chaîne à afficher dans le registre SI
xor bh, bh; RAZ de bh, qui stocke la page d'affichage
mov ah, 0x03
int 0x10; appel de l'interruption BIOS qui donne la position du curseur, stockée dans dx
mov cx, 1; nombre de fois où l'on va afficher un caractère
affiche_suivant:
mov al, [si];on met le caractère à afficher dans al
or al, al;on compare al à zéro pour s'arrêter
jz fin_affiche_suivant
cmp al, 13
je nouvelle_ligne
positionne_curseur:
```

```
    mov ah, 0x02;on positionne le curseur
    int 0x10
    mov ah, 0x0A;on affiche le caractère courant cx fois
    int 0x10
    inc si; on passe au caractère suivant
    inc dl; on passe à la colonne suivante pour la position du curseur
    jmp affiche_suivant
    fin_affiche_suivant:
    ret
    nouvelle_ligne:
    inc dh; on passe à la ligne suivante
    xor dl, dl; colonne 0
    jmp positionne_curseur
    hello: db 'Bonjour papi.', 13, 0
```

III - Assembleur : on continue

Relu par **ClaudeLELOUP**.

Petit rappel : nous en sommes à avoir un programme qui affiche une chaîne de caractères à l'écran quel que soit le système d'exploitation.

C'est, mine de rien, un excellent point de départ. Mais ça manque de... comment dire... interactivité. Ce qui serait mieux, s'pas, ce serait de dire des choses à l'ordinateur, comme lui nous en dit. Or, l'ordinateur est mal fourni : il est principalement perdu dans ses propres pensées, peu lui chaut le monde extérieur. Et donc, a fortiori, moi. Mon égoïsme est blessé, mon amour-propre traîné dans la boue, et je vais remédier à cela, non de moi de bordel de moi ! Or, autant en possibilités d'actions sur le monde, l'ordinateur est bien loti, autant en possibilités de ressentir le monde, c'est lamentable. Pensez donc, un malheureux clavier et une souris ! L'ordinateur est capable de nous balancer 2000 caractères d'un coup, alors que nous ne pouvons lui donner qu'un seul caractère à la fois ! Scandaleux. Mais on va essayer de faire quelque chose quand même.

III.1 - De l'entrée clavier

Monsieur BIOS a un gestionnaire de clavier, de son petit nom **INT 0x16**, qui a une fonction qui teste si un caractère a été entré au clavier : **0x01**. On va donc boucler sur cette fonction tant qu'on n'a pas de caractère à lire. Quand on a quelque chose dans le buffer du clavier, on va le lire avec la fonction **0x00**. Histoire de s'en souvenir pour un usage ultérieur, on va le stocker en mémoire. Pour voir ce qu'on tape au clavier, on affiche ce caractère lu. On décale donc le curseur à sa position suivante, et on repart tester s'il n'y aurait pas d'autres caractères à lire. Pour s'arrêter, je décide que la touche *"Entrée"*, caractère **13**, sera la marque de la fin de l'entrée clavier. A chaque caractère, on teste alors ce marqueur de fin. S'il est atteint, on passe à la ligne suivante. Puis j'affiche la chaîne entrée. Et pour ce faire, voyons les fonctions.

III.2 - Des fonctions

Dans un programme, on a souvent besoin de faire la même chose plusieurs fois. Plutôt que de réécrire l'ensemble du code à chaque fois, nous avons la possibilité de le mettre à un seul endroit, et de l'appeler au besoin : c'est une fonction. Faisons une fonction qui affiche une chaîne de caractères. Elle aura besoin de savoir quelle est la chaîne à afficher : ce sera l'adresse contenue dans **SI**. J'appelle cette fonction *"affiche_chaine"* : une fois que l'affichage sera fait, il faut revenir au programme qui appelle cette fonction : instruction **RET**, qu'on place à la fin du code d'affichage de la chaîne. On décale tout ce code à la fin du programme, juste avant les données, histoire qu'il ne soit pas exécuté n'importe comment. Cette fonction utilise les registres **AX**, **BX**, **CX** et **DX**. Pour se faciliter le travail, nous allons sauvegarder les valeurs que ces registres avaient avant l'appel à la fonction. Comme ça, l'appel à la fonction est neutre au niveau des registres. Pas la peine de sauvegarder **SI**, c'est un paramètre : on doit s'attendre à ce qu'il soit modifié par la fonction, ça permettra en outre de savoir combien de caractères ont été écrits. Pour sauvegarder les registres, nous allons utiliser la pile : à nous les messages d'insulte *"stack overflow"*. La pile est un espace mémoire dont l'adresse de fin est fixe, et dont l'adresse de début décroit à mesure qu'on y met des données. La dernière valeur qui y est stockée est donc au début, et sera récupérée en premier. La pile commence par défaut à la fin de l'espace mémoire disponible dans notre programme. Pour y mettre une valeur, c'est l'instruction **PUSH**, qui met 16 bits dans la pile. Pour récupérer une valeur, **POP** prend 16 bits de la pile pour les mettre dans un registre.

Pour appeler la fonction, c'est **CALL nom_de_la_fonction**.

Voici maintenant notre programme :

```
org 0x0100 ; Adresse de début .COM

;Ecriture de la chaîne hello dans la console
mov si, hello; met l'adresse de la chaîne à afficher dans le registre SI
```

```
call affiche_chaine
mov si, hello; met l'adresse de la chaîne à lire dans le registre SI
mov ah, 0x03
int 0x10; appel de l'interruption BIOS qui donne la position du curseur, stockée dans dx
mov cx, 1
attend_clavier:
mov ah, 0x01;on teste le buffer clavier
int 0x16
jz attend_clavier
;al contient le code ASCII du caractère
mov ah, 0x00;on lit le buffer clavier
int 0x16
mov [si], al;on met le caractère lu dans si
inc si
cmp al, 13
je fin_attend_clavier
;al contient le code ASCII du caractère
mov ah, 0x0A;on affiche le caractère courant cx fois
int 0x10
inc dl; on passe à la colonne suivante pour la position du curseur
mov ah, 0x02;on positionne le curseur
int 0x10
jmp attend_clavier
fin_attend_clavier:
inc dh; on passe à la ligne suivante pour la position du curseur
xor dl, dl
mov ah, 0x02;on positionne le curseur
int 0x10
mov byte [si], 0;on met le caractère terminal dans si
mov si, hello; met l'adresse de la chaîne à afficher dans le registre SI
call affiche_chaine
ret

affiche_chaine:
push ax
push bx
push cx
push dx
xor bh, bh; RAZ de bh, qui stocke la page d'affichage
mov ah, 0x03
int 0x10; appel de l'interruption BIOS qui donne la position du curseur, stockée dans dx
mov cx, 1; nombre de fois où l'on va afficher un caractère
affiche_suivant:
mov al, [si];on met le caractère à afficher dans al
or al, al;on compare al à zéro pour s'arrêter
jz fin_affiche_suivant
cmp al, 13
je nouvelle_ligne
mov ah, 0x0A;on affiche le caractère courant cx fois
int 0x10
inc dl; on passe à la colonne suivante pour la position du curseur
positionne_curseur:
inc si; on passe au caractère suivant
mov ah, 0x02;on positionne le curseur
int 0x10
jmp affiche_suivant
fin_affiche_suivant:
pop dx
pop cx
pop bx
pop ax
ret
nouvelle_ligne:
inc dh; on passe à la ligne suivante
xor dl, dl; colonne 0
jmp positionne_curseur
;fin de affiche_chaine

hello: db 'Bonjour papi.', 13, 0
```

III.3 - De la machine

On arrive à faire des choses, mais ce qui serait bien, maintenant, c'est d'avoir une idée de la machine sur laquelle le programme s'exécute. C'est plus compliqué, parce que toutes ces informations ne vont pas nécessairement être stockées aux mêmes endroits selon, j'imagine, le constructeur, le type de matériel, ce genre de choses.

III.3.a - Des drapeaux

Le processeur dispose d'un registre de drapeaux. Il faut voir un drapeau comme celui d'un arbitre ou d'un commissaire aux courses. Si le drapeau est invisible, il n'y a rien à signaler. Un drapeau levé signifie qu'il y a quelque chose : le vert à pois bleus signifie qu'un martien atterrit, le jaune que le n°13 vient de taper le n°6 d'en face, celui à damier que Senna a rencontré un mur, etc. En informatique, un drapeau est représenté par un bit. Ce bit à 0 signifie qu'il n'y a rien à signaler, le drapeau est invisible. Ce bit à 1 signifie que le drapeau est visible, levé. Non, malheureusement, le processeur n'a pas de drapeau pour les aliens. Voici des exemples de drapeaux dont le processeur dispose :

- **CF**, Carry Flag, drapeau de retenue : cela symbolise la retenue, comme à l'école ;
- **ZF**, Zero Flag, drapeau de zéro : vous venez de calculer la tête à toto ;
- **PF**, Parity Flag, drapeau de parité : votre résultat est pair. Le virus Parity Boot changeait la valeur de ce drapeau, le canaillou ;
- **SF**, Sign Flag, drapeau de signe : votre résultat est négatif ;
- **DF**, Direction Flag, drapeau de direction : vous allez dans le sens des adresses décroissantes, c'est le voyant de recul de votre voiture ;
- **OF**, Overflow Flag, drapeau de dépassement de capacité : votre résultat ne tient plus dans le type que vous avez choisi.

Ces drapeaux sont utilisés par les instructions de saut conditionnel. **JZ**, par exemple, signifie Jump if Zero et saute (à l'adresse donnée en paramètre) si le drapeau **ZF** est levé. Certains appels de fonction changent l'état des drapeaux selon une logique qui leur est propre, et les drapeaux perdent ainsi leur signification première, un peu comme si l'arbitre sortait, au lieu du carton rouge attendu, un ananas violet. Ca met le bazar, c'est drôle, les informaticiens adorent. Oui, nous aussi on jouera avec. Des drapeaux, le processeur en a tout un registre, ce qui fait, au maximum ...

...

...

...

16 ! Nos registres font 16 bits, ce qui fait 16 drapeaux possibles au maximum. Et le registre des drapeaux s'appelle, je vous le donne en mille : **FLAGS**. Quelque chose me dit que le processeur est anglo-saxon.

III.3.b - De la configuration système

Alors, comme point de départ, j'ai trouvé l'interruption **0x11**, qui remplit **AX** avec des flags et des nombres. Appelons donc **0x11** et analysons **AX**. On utilise l'instruction **TEST**, qui lève des drapeaux en fonction du résultat du **ET** logique entre les opérateurs. **TEST** ne change pas les valeurs des opérandes, ce qui est pratique, ça évite d'appeler l'interruption à chaque fois qu'on teste une partie de la valeur de retour. On teste avec un nombre binaire, préfixé *0b* : ça permet de voir facilement le bit qui nous intéresse. Par exemple, si on teste le premier bit, on teste avec *0b0001* : le résultat sera zéro, soit **ZF** à *1*, ou bien un, soit **ZF** à *0*. Une instruction de saut conditionnel fera le reste, et **AX** sera prêt à être testé avec le deuxième bit, soit *0b0010*.

Les troisième et quatrième bits commencent à être tordus : il s'agit de la mémoire disponible, comptée en nombre de blocs de 16 kilooctets. Mais comme un ordinateur sans mémoire vive ne sert à rien (merci monsieur Turing), le premier bloc n'est pas compté : il y a au moins 16 ko (kilooctets) de mémoire dans un ordinateur. Il faut donc ajouter

1 au nombre de pages de 16 ko. Les valeurs disponibles sur 2 bits sont : 0, 1, 2 et 3. Notre nombre de pages est alors de 1, 2, 3 ou 4, soit 16, 32, 48 ou 64 ko.

Il n'y a pas de test à faire, il faut récupérer le nombre. Pour le récupérer, il faut transformer notre retour afin qu'il soit notre nombre de pages, i.e. décaler ses bits de deux bits vers la droite (après suppression des bits qui ne nous intéressent pas par un **AND**). Il s'agit de l'instruction **SHR**, pour SHift Right (décalage à droite). Après, on incrémente pour prendre en compte la page gratis, et il faudrait multiplier par 16 pour avoir le nombre de ko disponibles. Une multiplication, c'est bien. Mais mieux, c'est le décalage de bits. Pour la faire courte, un décalage de 1 bit à droite correspond à une division par 2, tandis qu'un décalage de 1 bit à gauche fait une multiplication par 2. Pour multiplier par 16, il faut multiplier par 2 quatre fois, donc décaler de 4 bits vers la gauche, soit SHL ax, 4.

Ayant récupéré notre quantité de mémoire disponible, nous souhaiterions l'afficher. Or, il s'agit d'un nombre, pas d'une chaîne de caractères. "R" ne nous apprendrait rien. Il faut transformer notre nombre en chaîne de caractères.

III.4 - De l'affichage des nombres

Faisons une fonction de transformation de nombre en chaîne de caractères. AX contient le nombre à transformer, et SI l'adresse du début de la chaîne. On utilise l'algorithme appelé "algorithme des divisions successives". On va diviser notre nombre par 10 jusqu'à ce que le quotient soit 0, et stocker chaque reste comme étant un chiffre à afficher. Les chiffres ASCII ayant le bon goût d'être dans l'ordre, il suffit d'ajouter "0" à un chiffre pour avoir son caractère.

Alors, à la main : prenons 32. Divisons 32 par 10 : quotient 3, reste 2. 2 est le chiffre à afficher. Stockons 2 + "0". Divisons 3 par 10 : reste 3, quotient 0. Stockons 3 + "0". Le quotient est 0, fin de l'algorithme. La chaîne est dans l'ordre inverse. Si on l'a stockée dans la pile, en dépilant, on sera dans le bon ordre. Ne reste plus qu'à ajouter le zéro terminal et à l'afficher.

Une division se fait par l'instruction DIV suivie du diviseur, qui doit être dans un registre. DIV divise AX par le diviseur, et stocke le résultat dans AH pour le reste et AL pour le quotient. Note : avec ça, on ne pourra pas traiter de nombre plus grand que 2550.

Le code complet :

```
org 0x0100 ; Adresse de début .COM

;Ecriture de la chaîne hello dans la console
mov si, hello; met l'adresse de la chaîne à afficher dans le registre SI
call affiche_chaine
mov si, hello; met l'adresse de la chaîne à lire dans le registre SI
mov ah, 0x03
int 0x10; appel de l'interruption BIOS qui donne la position du curseur, stockée dans dx
mov cx, 1
attend_clavier:
mov ah, 0x01;on teste le buffer clavier
int 0x16
jz attend_clavier
;al contient le code ASCII du caractère
mov ah, 0x00;on lit le buffer clavier
int 0x16
mov [si], al;on met le caractère lu dans si
inc si
cmp al, 13
je fin_attend_clavier
;al contient le code ASCII du caractère
mov ah, 0x0A;on affiche le caractère courant cx fois
int 0x10
inc dl; on passe à la colonne suivante pour la position du curseur
mov ah, 0x02;on positionne le curseur
int 0x10
jmp attend_clavier
fin_attend_clavier:
inc dh; on passe à la ligne suivante pour la position du curseur
```

```
    xor dl, dl
    mov ah, 0x02;on positionne le curseur
    int 0x10
    mov byte [si], 0;on met le caractère terminal dans si
    mov si, hello; met l'adresse de la chaîne à afficher dans le registre SI
    call affiche_chaine

    int 0x11
    test ax, 0b0001
    jnz lecteurs_disquette
    mov si, pas_disquette
    call affiche_chaine
    test_coprocesseur:
    test ax, 0b0010
    jnz coprocesseur_present
    mov si, pas_coprocesseur
    call affiche_chaine
    test_memoire:
    and ax, 0b1100
    shr ax, 2
    inc ax; une zone mémoire est donnée gratis.
    shl ax, 4; les zones mémoires sont comptées par paquets de 16 ko
    mov si, hello
    call nombre_vers_chaine
    mov si, hello
    call affiche_chaine
    mov si, memoire_dispo
    call affiche_chaine
    ret

    lecteurs_disquette:
    mov si, disquettes
    call affiche_chaine
    jmp test_coprocesseur

    coprocesseur_present:
    mov si, coprocesseur
    call affiche_chaine
    jmp test_memoire

    nombre_vers_chaine:
    push bx
    push cx
    push dx
    mov bl, 10
    mov cx, 1
    xor dh, dh
    stocke_digit:
    div bl
    mov dl, ah
    push dx ;sauve le reste dans la pile
    inc cx
    xor ah, ah
    or al, al
    jne stocke_digit

    ;Affichage du chiffre
    boucle_digit:
    loop affiche_digit
    mov byte [si], 0
    pop dx
    pop cx
    pop bx
    ret

    affiche_digit:
    pop ax
    add ax, '0'
    mov [si], al
    inc si
    jmp boucle_digit
    ;fin nombre_vers_chaine
```

```
affiche_chaine:
push ax
push bx
push cx
push dx
xor bh, bh; RAZ de bh, qui stocke la page d'affichage
mov ah, 0x03
int 0x10; appel de l'interruption BIOS qui donne la position du curseur, stockée dans dx
mov cx, 1; nombre de fois où l'on va afficher un caractère
affiche_suivant:
mov al, [si];on met le caractère à afficher dans al
or al, al;on compare al à zéro pour s'arrêter
jz fin_affiche_suivant
cmp al, 13
je nouvelle_ligne
mov ah, 0x0A;on affiche le caractère courant cx fois
int 0x10
inc dl; on passe à la colonne suivante pour la position du curseur
positionne_curseur:
inc si; on passe au caractère suivant
mov ah, 0x02;on positionne le curseur
int 0x10
jmp affiche_suivant
fin_affiche_suivant:
pop dx
pop cx
pop bx
pop ax
ret
nouvelle_ligne:
inc dh; on passe à la ligne suivante
xor dl, dl; colonne 0
jmp positionne_curseur
;fin de affiche_chaine

disquettes: db 'Lecteur(s) de disquette', 13, 0
pas_disquette: db 'Pas de lecteur de disquette', 13, 0
coprocesseur: db 'Coprocesseur arithmétique', 13, 0
pas_coprocesseur: db 'Pas de coprocesseur', 13, 0
memoire_dispo: db ' ko.', 13, 0
hello: db 'Bonjour papi.', 13, 0
```

IV - Assembleur : des graphiques !

Relu par **ClaudeLELOUP**.

Aux dernières nouvelles, nous en étions à lire des choses au clavier et à regarder un peu ce qu'il y a dans la machine. C'est bien. On peut mieux. On peut mieux comment ? Allez, si on mettait à profit l'extraordinaire capacité de l'ordinateur à faire de l'affichage ?

IV.1 - De Windows Vista

Je ne veux pas être mauvaise langue, mais Windows Vista, entre nous soit dit, ce n'est pas fabuleux. On ne pourra plus exécuter notre programme sous Vista, parce que Môssieur Vista a décidé que non, c'était pas bien d'accéder au BIOS, surtout pour faire de l'affichage. Si bien que quand on va appeler une certaine fonction du gestionnaire d'affichage du BIOS (la fonction du pilote VESA), on va se faire jeter comme des malpropres. Ce n'est pas chouette de sa part, alors on va ruser : DOSBOX est un émulateur DOS qui fonctionne sous Vista.

Doit-on considérer qu'on est mauvais si on se fait renvoyer dans nos 6 mètres par Vista ? Je ne crois pas : VESA est un standard, je ne vois pas pourquoi on n'aurait pas le droit de l'utiliser.

Pour nous enquiquiner encore plus, NASM ne fonctionne pas avec DOSBOX. Donc, deux fenêtres : une en cmd normal, et une de DOSBOX. Je VEUX y arriver, et c'est pas Kro$oft qui va me l'interdire !

IV.2 - Le refacteur sonne toujours une paire de fois (refactoring)

Quand le code évolue, on change des trucs et des machins. On refait d'autres trucs. "Cent fois sur le métier remets ton ouvrage", disait Penelope Cruz, à moins que je n'aie pas tout saisi (note du correcteur : ça doit remuer dans les allées du Père Lachaise. Pauvre Boileau !). J'ai amélioré la fonction d'écriture de texte, histoire qu'on n'écrive pas n'importe où : quand on a plus de 80 caractères à afficher, elle faisait un peu n'importe quoi. Mais pas que. Une nouvelle instruction a fait son apparition : **LODSB**, comme LOaD String Byte. C'est l'équivalent de MOV [DS:SI], AL INC SI. Et pour être cohérent, plus facile à utiliser, l'adresse de la chaîne à afficher se passe maintenant dans le registre **SI (Source Index)** pour son offset, et **DS (Data Segment)** pour son segment.

J'ai mis la lecture au clavier dans une fonction, ça aura l'avantage de pouvoir être appelé n'importe quand, et donc de servir d'arrêt dans l'exécution du programme, ah là là, je suis d'une fourberie sans nom. Là aussi, une nouvelle fonction a fait son apparition : **STOSB**, comme STOre String Byte. Exactement l'inverse de **LOSB**, mais avec des registres différents : **DI (Destination Index)** comme offset d'adresse et **ES (Extra Segment)** comme segment d'adresse. Instructions équivalentes : MOV AL, [ES:DI] INC DI. Donc, on passe l'adresse de la chaîne de destination dans ces registres.

La fonction d'affichage de nombres a subi elle aussi une cure de jouvence : on peut, pensez donc, afficher des nombres jusqu'à 65535 ! Elle prend d'autres paramètres, comme **BL**, dont le bit de poids faible est mis à un si on veut terminer la chaîne de caractères à la fin, et **BH**, qui donne le nombre minimal de caractères que doit contenir la chaîne écrite. Ah oui. J'oubliais. Cette fonction transforme **AX** en chaîne de caractères. Elle ne l'affiche pas.

Il paraît que refaire des choses, comme cela, s'appelle du refactoring. C'est un bien vilain mot qui écorche l'oreille fine du franchouillard : recodage serait mieux, de mon point de vue.

IV.3 - De VESA

Au commencement était la carte perforée. Puis vint la télévision. Ou inversement. Bref, les ordinateurs furent munis d'un écran. Monochrome. Puis on sut écrire autre chose que du texte. Puis apparut le niveau de gris, puis la couleur.

Puis tout le monde fit n'importe quoi. IBM sortit alors la spécification VGA. On progressa encore. Ce fut pire. Re standard, et VESA.

VESA est, succinctement, le pilote des cartes graphiques au-delà de VGA. Pour savoir s'il est disponible, une interruption : **0x10** (puisque c'est du gestionnaire vidéo), une fonction : **0x4F**, et une tripotée de sous-fonctions. L'idée est d'appeler la sous-fonction **0x00** qui va nous dire tellement de choses que bien des tortionnaires envieraient la place du développeur.

A cette sous-fonction, il faut lui donner un espace mémoire pour qu'elle puisse y mettre toutes ces choses qu'elle sait. Elle va les mettre à l'adresse contenue dans le registre **DI (comme Destination Index)**, et elle a besoin de 256 octets. Nous allons lui donner entière satisfaction, puisque nous allons lui donner notre adresse fourre-tout, à savoir le label hello. On dispose d'au moins 16 ko de mémoire, donc on a de la place en bout de programme pour stocker 256 octets, on ne s'occupe de rien en ce moment.

Comme nous sommes des fous parfaitement au fait du fonctionnement d'un microprocesseur, on colle directement les paramètres dans AX en une seule instruction, ah oui madame, je suis un malade des cycles d'horloge, c'est précieux ces choses-là.

Voici la structure de ce qui nous est renvoyé :

Offset	Taille	Description
0x00	4 octets (Double Mot)	Signature, doit contenir la chaîne de caractères "VESA".
0x04	2 octets (Mot)	Numéro de version du VESA. AH : numéro majeur. AL : numéro mineur.
0x06	4 octets (Double Mot)	Pointeur sur le nom de l'OEM terminé par un caractère NULL.
0x0A	4 octets (Double Mot)	Ca, je n'ai pas compris.
0x0E	4 octets (Double Mot)	Pointeur sur une liste de modes vidéo du VESA supportés par l'OEM. Se termine par le code 0xFFFF. Chaque mode est stocké comme un mot.
0x11	238 octets	Ca, je n'ai pas compris.

Nous allons lire et afficher tous les mots (mot veut dire nombre de 16 bits ici) tant qu'on ne lit pas le nombre magique **0xFFFF**. Bon, d'accord, c'est beaucoup d'efforts pour pas grand-chose, puisque après, on se mettra dans le mode 0x13. Mais on sera prêts pour la suite.

IV.4 - Du point

Ensuite, on est parti pour l'affichage d'un point. Ca consiste en l'écriture dans la mémoire vidéo d'une couleur. Une couleur, en mode 0x13, c'est un octet. Les coordonnées en abscisse et ordonnée du point à afficher doivent être transformées en coordonnée linéaire, la mémoire étant linéaire. Pour ce faire, la formule est : ordonnée * nombre de points sur la ligne + abscisse. Subtilité : plutôt que de faire une multiplication par 320, on travaille par deux décalages de bits (multiplication par puissances de 2) additionnés. On écrit à cette coordonnée linéaire la valeur de la couleur, et c'est fait.

IV.5 - De Bresenham

Maintenant, tracer des lignes. Des segments de droite, pour être exact. Il nous faut un algorithme, c'est-à-dire la marche à suivre pour parvenir à nos fins. Wikipédia nous en fournit un, qui est le top des algorithmes de tracé de segment : **l'algorithme de Bresenham**

Cet algorithme, il faut le traduire en assembleur. Ma traduction n'est certainement pas fabuleuse, mais aux dernières nouvelles, elle fonctionne.

IV.6 - Du code

Le code de ce chapitre se trouve ici : **carabistouille.asm**. Parce que bon, ça commence à faire de la ligne de code.

V - Assembleur : au delà de 0x13

Relu par **ClaudeLELOUP**.

Dessiner, c'est bien. Après tout, on commence à dessiner avant d'écrire, nous autres mortels. Continuons donc le dessin. Nous savons dessiner des droites en mode VGA, c'est à dire en 320 par 200. Mais les ordinateurs peuvent faire beaucoup mieux en résolution : du 1280 par 1024 ne devrait pas faire peur à un PC du XXI^{e} siècle. Essayons.

V.1 - De la limite

Notre processeur est utilisé en 16 bits. Sans subtilité, nous pouvons manipuler des nombres qui vont gaillardement jusque 65535. Mais pas plus. Pareil pour les adresses. On peut adresser 65536 octets en mémoire. Or, un octet étant une couleur (en mode 256 couleurs), on peut donc écrire 65536 points à l'écran en une seule passe. Quand on se rend compte que 320 par 200, ça fait 64000, on se dit qu'un bête 800 par 600 va nécessiter de la ruse.

V.2 - Au-delà des limites

M. VESA, que nous avons déjà rencontré au chapitre précédent, y a pensé. Et il s'est dit, comme ça, que nous écrirons l'écran toujours aux mêmes adresses, donc, pas plus de 65536 octets d'un coup. L'idée est donc de couper l'écran en morceaux de 64 ko, et de demander à la carte graphique de changer de morceau. Comme ça, nous, on ne fait des choses que sur un seul morceau de 64 ko, et on dit à la carte graphique : ça c'est le morceau 0, ça c'est le morceau 6, etc. Pour changer de morceau, on appelle l'interruption 0x10 (le gestionnaire d'affichage), la fonction 0x4F (VESA), sous-fonction 0x05, avec BH à zéro et le numéro du morceau dans DX.

Au fait, pourquoi le numéro du morceau dans DX ?

Tiens, oui, pourquoi ?

Vous allez voir, c'est bien fait.

Si si, c'est intelligent.

Pourquoi dans DX ?

Aucune idée ?

Comment on calcule le numéro du morceau ?

Vi. Tout à fait. On utilise la formule : Y * largeur de la ligne + X, avec (X, Y) coordonnées du point à afficher. Or Y * largeur de la ligne va donner un résultat supérieur à 65536. Et l'opération de multiplication dans le processeur va mettre dans DX, quand ça dépasse 65535, le nombre de fois qu'il y a 65536 dans notre résultat. Et ce nombre de fois qu'il y a 65536, c'est précisément notre numéro de morceau ! Il n'y a donc qu'à appeler la sous-fonction 0x05 avec le résultat de la multiplication, et le tour est joué. Elle est pas belle, la vie ?

Bon.

En fait, c'est pas vrai.

On peut changer de morceau d'écran en cours de ligne. Sinon, ce n'est pas drôle. Si on a une retenue à l'ajout de l'abscisse, c'est qu'on change de morceau en cours de route. ADC (ADd with Carry) sur DX nous mettra alors sur le bon morceau d'écran.

V.3 - D'autres ruses

Pour demander une nouvelle résolution, on passe son numéro. Il y a pourtant une petite subtilité : si on demande le mode brutalement, VESA va mettre le contenu de l'écran à 0. Si on n'en a pas besoin, il faut ajouter 0x8000 au numéro du mode : VESA ne fera pas de nettoyage.

D'autre part, on change rarement de morceau d'écran. En stockant dans une variable le numéro du morceau courant, on pourra ne demander le changement que lorsque le morceau d'écran du point à afficher est différent de celui stocké précédemment.

Plus drôle encore : il arrive que la longueur de la ligne soit, en mémoire, plus grande que celle affichée. C'est notamment le cas lorsque le nombre de points à l'écran est différent de multiples de 65536. En gros, ça veut dire qu'il y a des points non affichés. Ces points non affichés sont situés en bout de ligne. Pour faire nos calculs, nous n'allons pas utiliser le nombre de points par ligne affichés, mais le nombre d'octets par ligne.

V.4 - Du code

Le code de ce chapitre se trouve ici : **VESA.asm** Il y a des morceaux de code commentés, et des bidules que nous verrons plus tard.

VI - Assembleur : et si on en avait encore moins ?

Relu par **ClaudeLELOUP**.

On a écrit à l'écran, dessiné, lu l'entrée clavier. Nous l'avons fait sans utiliser les facilités fournies par le système d'exploitation. Mais nous utilisons toujours le système d'exploitation, parce que nous lui demandons l'exécution de notre programme. Il nous modifie notre fichier, c'est le fameux org 256 que nous devons écrire pour que nos adresses restent valides dans notre programme. Et bien, ça aussi, je n'en veux plus.

VI.1 - Des machines virtuelles

Pour que le système d'exploitation n'existe plus, un bon moyen est le "format c:". C'est très bien, mais très gênant, puisque finalement, ça supprime tout, et donc aussi notre éditeur de texte pour écrire notre programme. Et sans éditeur, nous ne pourrons pas écrire quoi que ce soit, et donc il sera impossible de refaire fonctionner l'ordinateur. C'est une mauvaise idée. Mais il se trouve que beaucoup de gens ont besoin d'un ordinateur avec rien dessus. Et à l'ère de l'internet, si beaucoup de gens ont besoin de quelque chose, alors quelqu'un l'a fait. Ca s'appelle une machine virtuelle, c'est un logiciel qui simule un ordinateur sur votre ordinateur. Oui, le concept est étrange, mais c'est bien pratique. Il en existe plusieurs, prenez celui que vous voulez tant que l'on peut y configurer un PC sans système d'exploitation. Personnellement, j'ai tenté ces deux-là :

- **VirtualBox**
- **Bochs**

VirtualBox est plus sympa à utiliser que Bochs, mais Bochs a aussi ses avantages, c'est même pour ça que les deux existent.

Ce qu'il nous faut, nous, c'est un PC avec un lecteur de disquettes avec une disquette dedans. La disquette sera de préférence un fichier ".img" mais c'est au choix. L'idée est d'avoir un ordinateur, même faux, sur lequel on va pouvoir jouer autant qu'on veut sans pour autant tout casser.

VI.2 - Du chargeur de tige de botte

Au démarrage, un ordinateur a un gros problème : il doit charger en mémoire un programme qui lui permettra de lancer d'autres programmes, qui éventuellement, lanceront d'autres programmes, etc. Mais le premier programme, comment se charge-t-il en mémoire ?

C'est un peu le problème de la poule et de l'oeuf...

L'ordinateur va utiliser la technique du baron de Münchausen : alors que le baron allait se noyer, il appelle à l'aide. Comme personne ne répond, le baron se trouve contraint, pour se sauver, de se tirer lui-même de l'eau par ses tiges de botte. En anglais, ça se dit "bootstrap". En français, on est moins poétique et on appelle le "bootstrap loader" : le chargeur de secteur d'amorçage.

Le BIOS d'un PC va, au démarrage, chercher, sur certains périphériques, un bloc de 512 octets qui se termine par la séquence 0x55AA. 0x55AA est un nombre magique : ça n'a aucune raison particulière d'être ce nombre-là plutôt qu'un autre, simplement, il en faut un, alors, on en sort un du chapeau. Et sortir du chapeau, c'est magique.

Ce bloc de 512 octets, il va le mettre à l'adresse 0x07C0:0000, et lui donner la main. C'est comme ça. C'est 0x07C0:0000 et puis c'est marre.

Ce bloc de 512 octets, nous l'allons écrire.

VI.3 - D'un secteur d'amorçage pour nous

Bien sûr, on peut utiliser des secteurs d'amorçage déjà faits. Mais mon but est de partir du plus bas. Il me faut donc le mien.

Alors, hardi, compagnons, portons haut la flamberge et écrivons :

```
    org 0x0000 ; On n'a pas de décalage d'adresse

        jmp 0x07C0:debut     ; On est chargé dans le segment 0x07C0
    debut:
        ; Met les segments utiles au segment de code courant
        mov ax, cs
        mov ds, ax
        call detect_cpu

    initialise_disque: ; Initialise le lecteur de disque
        xor ax, ax
```

Ici, une parenthèse : on est censé donner à l'interruption 0x13 l'identifiant du disque sur lequel on est chargé via le registre DL. Or, il se trouve que le BIOS, décidément conçu par des gens qui ne veulent pas s'embêter, nous donne le lecteur sur lequel il nous a trouvé dans le registre DL. Comme notre programme sera sur le même disque, on n'a rien à changer. Fin de la parenthèse.

```
        int 0x13
        jc
initialise_disque; En cas d'erreur on recommence (sinon, de toute façon, on ne peut rien faire)

    lire:
```

De nouveau une parenthèse : on va charger un autre programme que le secteur d'amorçage en mémoire. Où allons-nous le mettre ? Pour l'instant, où on veut, on est l'chef, après tout. 0x1000:0000 n'a pas l'air trop mal en première approximation. Et on va charger 5 secteurs, parce que notre programme fera moins de 2,5 ko. Ah oui, c'est comme ça.

```
     mov ax, 0x1000 ; ES:BX = 1000:0000
     xor bx, bx
     mov es, ax
     mov ah, 2 ; Fonction 0x02 : chargement mémoire
     mov al, 6 ; On s'arrête au secteur n° 6
     xor ch, ch ; Premier cylindre (n° 0)
     mov cl, 2 ; Premier secteur (porte le n° 2, le n° 1, on est dedans, et le n° 0 n'existe pas)
     ; Ca fait donc 5 secteurs
     xor dh, dh ; Tête de lecture n° 0
     ; Toujours pas d'identifiant de disque, c'est toujours le même.
     int 0x13 ; Lit !
     jc lire ; En cas d'erreur, on recommence
     mov si, sautNoyau ; Un petit message pour rassurer les troupes.
     call affiche_chaine
     jmp 0x1000:0000 ; Et on donne la main au programme que nous venons de charger
```

Alors, pour la fine bouche : on va faire de l'affichage et du test. D'abord, on va tester le processeur, car depuis qu'on a commencé, j'ai appris des trucs : pour savoir si on a à faire à un 8086, 80286 ou 80386, on teste certains bits du registre de drapeaux.

```
    detect_cpu:
     mov si, processormsg ; Dit à l' utilisateur ce qu'on est en train de faire
     call affiche_chaine
```

```
 mov si, proc8086 ; De base, on considère qu'il s'agit d'un 8086
 pushf ; sauvegarde les valeurs originales des drapeaux

 ; teste si un 8088/8086 est présent (les bits 12-15 sont à 1)
 xor ah, ah ; Met les bits 12-15 à 0
 call test_drapeaux
 cmp ah, 0xF0
 je finDetectCpu ; 8088/8086 détecté

 mov si, proc286 ; On considère qu'il s'agit d'un 286
 ; teste si un 286 est présent (les bits 12-15 sont effacés)
 mov ah, 0xF0 ; Met les bits 12-15 à 1
 call test_drapeaux
 jz finDetectCpu ; 286 détecté

 mov si, proc386 ; aucun 8088/8086 ou 286, donc c'est un 386 ou plus
finDetectCpu:
 popf ; restaure les valeurs originales des flags
 call affiche_chaine
 ret

test_drapeaux:
 push ax ; copie AX dans la pile
 popf ; Récupère AX en tant que registre de drapeaux. Les bits 12-15 sont initialisés pour le test
 pushf ; Remet le registre de drapeaux sur la pile
 pop ax ; Les drapeaux sont mis dans AX pour analyse
 and ah, 0xF0 ; Ne garde que les bits 12-15
 ret
```

Et maintenant, la culte routine d'affichage en mode texte

```
affiche_chaine:
 push ax
 push bx
 push cx
 push dx
 xor bh, bh; RAZ de bh, qui stocke la page d'affichage
 mov ah, 0x03
 int 0x10; appel de l'interruption BIOS qui donne la position du curseur, stockée dans dx
 mov cx, 1; nombre de fois où l'on va afficher un caractère
affiche_suivant:
 lodsb
 or al, al;on compare al à zéro pour s'arrêter
 jz fin_affiche_suivant
 cmp al, 13
 je nouvelle_ligne
 mov ah, 0x0A;on affiche le caractère courant cx fois
 int 0x10
 inc dl; on passe à la colonne suivante pour la position du curseur
 cmp dl, 80
 jne positionne_curseur
nouvelle_ligne:
 inc dh; on passe à la ligne suivante
 xor dl, dl; colonne 0
positionne_curseur:
 mov ah, 0x02;on positionne le curseur
 int 0x10
 jmp affiche_suivant
fin_affiche_suivant:
 pop dx
 pop cx
 pop bx
 pop ax
 ret
;fin de affiche_chaine

proc8086: db '8086', 13, 0
proc286: db '286', 13, 0
```

```
    proc386: db '386', 13, 0
    processormsg: db 'Test du processeur : ', 0
    sautNoyau: db 'Saut au noyau', 13, 0
```

Petit souci : notre secteur d'amorçage doit se terminer par 0x55AA, qui est un mot et qui doit donc être à la fin des 512 octets. Pour le mettre à la fin, nous allons simplement remplir les octets libres jusque 510 avec des 0. Cela fait donc 510 octets, moins l'adresse du dernier octet de code, moins l'adresse du premier octet de la section (qui est égale à 0, mais là n'est pas le problème). NASM fournit "$" comme étant l'adresse du début de la ligne de code courante, et "$$" comme l'adresse de la première instruction de la section. NASM fournit "times", qui répète ce qui le suit un certain nombre de fois. On utilise times 510 - taille de notre code fois pour écrire 0, et le tour est joué.

```
    times 510-($-$$) db 0
    dw
0xAA55 ; Le nombre magique écrit à l'envers parce que M. Intel est grosboutiste, ce qui signifie qu'il inverse l
```

Nous avons maintenant notre secteur d'amorçage, à compiler avec : nasm -o amorce.com amorce.asm

Et on ne le lance pas bêtement. On attend d'avoir fait la suite.

VI.4 - Passer par noyau planète

La suite est un autre programme : nouveau fichier, page blanche... Noooon... pas page blanche, on va être un peu malin, on va partir du chapitre précédent. Simplement, on va, au départ, initialiser les segments utilisés par notre programme. Comme tout est dans un seul segment, on va les initialiser avec la valeur de CS, le segment de code, qui est bon, puisqu'il a été initialisé par le secteur d'amorçage. Et pour ne pas s'emberlificoter, on va utiliser un autre segment pour la pile. Au pif, encore une fois.

Ensuite, techniquement, nous développons ce qu'il est convenu d'appeler un noyau de système d'exploitation. Alors, bon, c'est pas vraiment un noyau, mais on va faire comme si. Et faire comme si, ça veut dire que notre programme ne rend jamais la main. D'ailleurs, à qui la rendrait-il ? Au secteur d'amorçage, qui a fini son travail il y a bien longtemps ? Non. Il ne rendra pas la main. Ainsi, au lieu du "ret" final, ce sera "jmp $", qui fait une boucle infinie. D'autre part, il n'y aura pas d'en-tête du système d'exploitation, vu que le système d'exploitation, c'est lui. Comme le secteur d'amorçage, nous aurons "org 0".

VI.5 - Des 45 tours

Comment va-t-on utiliser notre programme ?

Conceptuellement, nous allons en faire une disquette de démarrage, oui, une disquette dite "de boot" ! On compile notre programme normalement. Ensuite, on copie bout à bout notre secteur d'amorçage et notre programme dans un seul fichier. Sous Windows, DOS et Cie, la commande est :

```
copy amorce.com/B+programme.com/B disk.img /Y
```

"/B" signifie qu'il s'agit d'un fichier binaire, ce qui évitera au DOS de le modifier. **"/Y"** évite d'avoir la confirmation de réécriture qui ne manque pas d'arriver dès la deuxième fois.

Sous système Unix, je ne connais pas la commande.

Le fichier copié est la disquette de démarrage.

Dans le code assembleur, on termine le fichier comme pour le secteur d'amorçage, avec un **"times 4096 - ($ - $$) db 0"**. Cela sert le même objectif : que le fichier compilé fasse un nombre entier de secteurs de disquette.

Il ne reste plus qu'à configurer le PC virtuel pour qu'il lise le fichier généré comme une disquette : soit on lui donne une vraie disquette qui ne contient que notre fichier, soit on lui précise que la disquette est en fait un fichier image.

Il ne reste plus qu'à démarrer la machine virtuelle, et voilà ! On retrouve notre programme.

VI.6 - Du massacre de disque

Notre programme a été lancé sur une machine virtuelle, et surtout, qu'il en soit ainsi pour le moment : il y a un morceau de code qui écrit sur le premier disque dur. Si on l'exécutait sur un vrai ordinateur, on écrirait là où est le système d'exploitation, ce qui est toujours une mauvaise idée. Ce petit bout de code ne fait que recopier les premiers secteurs de la disquette sur les premiers secteurs du disque dur. En enlevant la disquette de la machine virtuelle, la machine virtuelle démarre toujours, et exécute notre programme.

VI.7 - Du code

Le code de ce chapitre se trouve ici :

- **amorcage.asm**
- **programme.asm**

VII - Assembleur : reconstruction

Relu par **ClaudeLELOUP**.

Depuis la création du secteur d'amorçage au chapitre précédent, je ne veux pas dire qu'on n'a plus rien, mais peu s'en faut. Les routines sont un peu bringuebalantes, tout ça. Cela nécessite une petite remise à niveau.

VII.1 - Du foutoir

Il faut avouer ce qui est : il devient rude de retrouver quoi que ce soit dans le code. NASM permet de s'y retrouver un peu mieux en séparant le code en plusieurs fichiers. La directive %include "[nom du fichier]" permet d'insérer à cet endroit le contenu du fichier spécifié.

VII.2 - De l'affichage de texte

Les routines d'affichage de texte ont été déportées dans un fichier *"affichageTexte.asm"*. Il y a deux fonctions : une fonction qui affiche une chaîne de caractères à l'écran *"affiche_chaine"*, et une fonction qui transforme un nombre en chaîne de caractères *"nombre_vers_chaine"*. Il faut garder à l'esprit que ces fonctions ont vocation à disparaître, puisqu'elles fonctionnent en mode texte, qui disparaît rapidement. Elles sont reprises du chapitre 3, clarifiées, déboguées et ainsi de suite.

VII.2.a - affiche_chaine

Cette fonction affiche la chaîne de caractères pointée par **si** et se terminant par le caractère **0**, aussi appelé caractère **nul**, de valeur numérique **0**. Elle suppose que l'écran mesure **80** caractères de large sur **25** de haut. En cas de débordement, elle efface la page et continue son écriture en haut. Elle ne connaît que deux caractères non affichables : le **0** et le **13**, respectivement marqueur de fin de chaîne et retour à la ligne.

```
;Affiche à l'écran une chaîne de caractères
affiche_chaine:
 push ax
 push bx
 push cx
 push dx
 xor bh, bh; RAZ de bh, qui stocke la page d'affichage
 mov ah, 0x03
 int 0x10; appel de l'interruption BIOS qui donne la position du curseur, stockée dans dx
 mov cx, 1; nombre de fois où l'on va afficher un caractère
affiche_suivant:
 lodsb
 or al, al;on compare al à zéro pour s'arrêter
 jz fin_affiche_suivant
 cmp al, 13
 je nouvelle_ligne
 mov ah, 0x0A;on affiche le caractère courant cx fois
 int 0x10
 inc dl; on passe à la colonne suivante pour la position du curseur
 cmp dl, 80
 jne positionne_curseur
nouvelle_ligne:
 inc dh; on passe à la ligne suivante
 xor dl, dl; colonne 0
 cmp dh, 25
 jb positionne_curseur
 xor dh, dh
 mov ah, 0x02;on positionne le curseur
 int 0x10
 mov cx, 25*80
```

```
 mov ah, 0x0A;on affiche le caractère courant cx fois
 mov al, ' '
 int 0x10
 mov cx, 1
positionne_curseur:
 mov ah, 0x02;on positionne le curseur
 int 0x10
 jmp affiche_suivant
fin_affiche_suivant:
 pop dx
 pop cx
 pop bx
 pop ax
 ret
```

VII.2.b - nombre_vers_chaine

Cette fonction écrit dans la chaîne pointée par **di** le nombre contenu dans **ax**.

```
;écrit dans la chaîne pointée par DI le nombre contenu dans AX
;si CL est à un, on écrit un caractère terminal
;CH contient le nombre minimal de caractères à utiliser
;BL contient la base
nombre_vers_chaine:
 push bx
 push dx
 push cx
 xor bh, bh
 mov cl, 1
stocke_digit:
 xor dx, dx
 div bx
 push dx ;sauve le reste dans la pile
 inc cl
 or ax, ax
 jne stocke_digit
;Les digits sont stockés dans la pile
 inc bh ;
ajout_zero:
 cmp ch, cl
 jb clearCH
 push ' '-'A'+10
 inc cl
 jmp ajout_zero
clearCH:
 xor ch, ch
;Affichage du chiffre
boucle_digit:
 dec bh
 loop affiche_digit
 pop cx ; on récupère le paramètre bx
 test cl, 0b1 ; s'il est à 1, on écrit un caractère terminal
 jz depile
 mov byte [di], 0
depile:
 pop dx
 pop bx
ret

affiche_digit:
 pop ax
 cmp ax, 10
 jae dixPlus
 add ax, '0'
 stosb ; met AL dans l'octet pointé par DI et incrémente DI
 jmp boucle_digit
dixPlus:
 add ax, 'A' - 10
 stosb
```

```
    jmp boucle_digit
;fin nombre_vers_chaine
```

VII.3 - Des modes graphiques

Pour faire une sorte de séance de révision, nous allons retravailler les graphiques du chapitre 5. Reprenons tout le déroulement du programme :

- saut au début, optionnel ;
- p'tit message pour ne pas perdre la main ;
- demande d'informations sur le VESA BIOS EXTENSION ;
- traitement de ces informations ;
- choix du mode graphique ;
- passage dans le mode choisi ;
- affichage de points ;
- affichage de droites ;
- affichage d'un alphabet.

VII.3.a - Saut au début, optionnel

```
org 0x0000 ; Adresse de début .COM

jmp start
%include "affichageTexte.asm"

start:
;On initialise Data Segment et Extra Segment à Code Segment
 call initialise_segments
```

On s'en souvient, on est en binaire pur pour être chargé depuis un chargeur d'amorçage, on a gardé la directive *org* d'une inutilité certaine ici. On fait un saut pour inclure les fonctions d'affichage de texte au début. C'est un choix, absolument pas justifiable par autre chose que "c'est comme ça". On appelle une fonction "initialise_segments" qui ne sera jamais appelée qu'une fois. On pourrait la mettre en macro, mais nous verrons les macros plus tard. C'est donc encore en fonction.

```
;Initialisation des segments
initialise_segments:
 mov ax, cs
 mov ds, ax
 mov es, ax
 mov ax, 0x8000
 cli
 mov ss, ax
 mov sp, 0xF000
 sti
 ret
;Fin initialisation segments
```

VII.3.b - P'tit message pour ne pas perdre la main

```
 mov si, hello; met l'adresse de la chaîne à afficher dans le registre SI
 call affiche_chaine
```

Bon, le message sera affiché, mais tel quel, on a d'excellentes chances de ne pas le voir, parce qu'on va afficher plus d'un écran par la suite. Il faut modifier le code qui viendra après pour le voir.

VII.3.c - Demande d'informations sur le VESA BIOS EXTENSION

```
mov ax, 0x4F00 ; demande infos sur le pilote VESA VBE
mov di, VESASignature
int 10h
cmp ax, 0x4F ; Si AL != 0x4F, on n'a pas de VESA, donc fin. Si AH != 0, erreur, donc fin.
jne fin
```

VESASignature est défini dans la partie "données" :

```
;Informations du pilote VESA
VESASignature: times 4 db 0; "VESA", signature de 4 octets
VESAVersion: dw 0; numéro de version de VBE
OEMStringPtr: dd 0; Pointeur vers le nom de l'OEM
Capabilities: dd 0; Possibilités de la carte graphique
VideoModePtr: dd 0; Pointeur vers les modes accessibles
TotalMemory: dw 0; Nombre de blocs mémoire de 64ko
reserved: times 236 db 0; Complément à 256 octets, taille du bloc
```

Ces informations, je ne les sors pas de mon chapeau, évidemment. Elles proviennent d'ici : **VESA BIOS Extension (VBE) Core Functions Standard Version 2.0**, tout ça parce que dans ma machine virtuelle, c'est VBE 2.0 qui est implanté.

VII.3.d - Traitement de ces informations

Le traitement des informations VESA consiste principalement en deux choses : gérer les cas d'erreur et pousser plus avant les investigations. Pour les erreurs, on restera très simple puisqu'on sort au moindre problème. Pour le reste, voyons cela.

Première chose, afficher le nom du constructeur :

```
cmp ax, 0x4F ; Si AL != 0x4F, on n'a pas de VESA, donc fin. Si AH != 0, erreur, donc fin.
jne fin
mov si, OEMStringPtr ; pointeur vers le nom de l'OEM stocké offset:segment
lodsw; on charge l'adresse d'offset dans ax
mov bx, ax ; BX contient l'adresse d'offset
lodsw ; on charge l'adresse de segment dans ax
mov si, bx ; SI pointe sur le nom de l'OEM
push ds ; on sauvegarde DS
 mov ds, ax ; ds contient l'adresse de segment du nom de l'OEM
 call affiche_chaine
pop ds ; on restaure DS

mov cx, 18
lignes_vides:
mov si, retour_chariot
call affiche_chaine
loop lignes_vides
```

Notez qu'on ajoute 18 lignes de retour chariot. C'est bête, c'est méchant et ça ne sert qu'à décaler l'affichage pour en avoir plus à visualiser sur l'écran suivant. Cette horreur devrait partir prochainement.

Nous allons ensuite lire les informations de chaque mode supporté par notre interruption **0x10**. Ca nous permettra de choisir un bon mode vidéo.

```
mov si, VideoModePtr ; pointeur vers la liste des modes supportés
lodsw ; on charge l'adresse d'offset dans ax
```

```
     mov cx, ax ; cx contient l'adresse d'offset
     lodsw ; on charge l'adresse de segment dans ax
     mov si, cx ; si pointe sur le premier mode supporté
     mov dx, ax ; dx contient l'adresse de segment
    lit_mode_suivant:
     push ds
      mov ds, dx ; ds contient l'adresse de segment de la liste des modes
      lodsw ;charge dans ax le mode
     pop ds
     cmp ax, 0xFFFF ; Fin de la liste
     je arret_modes
     mov cx, ax
     mov ax, 0x4F01 ; demande infos sur le mode VESA
     mov di, ModeAttributes
     int 0x10
```

VideoModePtr est défini dans le bloc de données :

```
    ;Informations d'un mode vidéo
    ModeAttributes: dw 0; Attributs du mode
    WinAAttributes: db 0; Attibuts de la fenêtre A
    WinBAttributes: db 0; Attibuts de la fenêtre B
    WinGranularity: dw 0; Granularité de la fenêtre en ko
    WinSize: dw 0; Taille de la fenêtre en ko
    WinASegment: dw 0; Segment de la fenêtre A
    WinBSegment: dw 0; Segment de la fenêtre B
    WinFuncPtr: dd 0; Pointeur vers la fonction "de fenêtrage"
    BytesPerScanLine: dw 0; Octets par "scanline"
    XResolution: dw 0; Résolution horizontale
    YResolution: dw 0; Résolution verticale
    XCharSize: db 0; Largeur d'un caractère
    YCharSize: db 0; Hauteur d'un caractère
    NumberOfPlanes: db 0; Nombre de plans mémoire
    BitsPerPixel: db 0; Bits par pixel
    NumberOfBanks: db 0; Nombre de banques de style CGA
    MemoryModel: db 0; Type de modèle mémoire
    BankSize: db 0; Taille des banques de style CGA
    NumberOfImagePages: db 0; Nombre de pages image
    res1: db 0; Reservé
    RedMaskSize: db 0; Taille du masque rouge en couleur directe
    RedFieldPosition: db 0; Position du bit faible du masque rouge
    GreenMaskSize: db 0; Taille du masque vert en couleur directe
    GreenFieldPosition: db 0; Position du bit faible du masque vert
    BlueMaskSize: db 0; Taille du masque bleu en couleur directe
    BlueFieldPosition: db 0; Position du bit faible du masque bleu
    RsvdMaskSize: db 0; Taille du masque réservé en couleur directe
    RsvdFieldPosition: db 0;  Position du bit faible du masque réservé
    DirectColorModeInfo: db 0; Attributs du mode de couleur directe
    res2: times 216 db 0; Complément à 256 octets, taille du bloc
```

Le test d'erreurs :

```
     cmp ax, 0x4F ; Si AL != 0x4F, la fonction n'est pas supportée, on se contentera du VGA. Si AH !
= 0, erreur, pareil.
     jne lit_mode_suivant
     test word [ModeAttributes], 0xF
     jz lit_mode_suivant
```

On affiche les informations pertinentes du mode courant :

```
    ;On ecrit les modes
     mov di, hello ; on écrit dans hello
     mov ax, cx
     push cx
```

```
 mov ch, 3
 mov bl, 16
 call nombre_vers_chaine
 mov al, ':'
 stosb
 mov ch, 4
 mov bl, 10
 mov ax, [XResolution]
 call nombre_vers_chaine
 mov ax, ('*' << 8) + ' '
 stosw
 mov al, ' '
 stosb
 mov ax, [YResolution]
 call nombre_vers_chaine
 mov ax, ' '
 stosb
 mov ch, 2
 mov al, [BitsPerPixel]
 call nombre_vers_chaine
 mov al, ' '
 stosb
 mov ax, ';' ; on met 2 caractères d'un coup après la chaîne : un "\n" et le zéro terminal.
 stosw ; les caractères sont dépilés, c'est-à-dire qu'il faut placer le premier dans la zone basse
pop cx
    push si ;sauve si sur la pile
 mov si, hello
 call affiche_chaine
   pop si ; on récupère si
```

Les esprits chagrins remarqueront que j'écris sur l'emplacement *hello*. Alors, il est fait pour cela, et j'en suis parfaitement conscient. Vu le peu que j'écris, j'ai largement la place dans cet espace pour mettre ce qui m'intéresse.

On va déborder un peu sur le paragraphe suivant : on va chercher quelle est la meilleure résolution à utiliser. On va la définir comme étant celle maximisant largeurEnPixels * hauteurEnPixels * TailleDeCodageDesCouleurs. Ca se fait dans la même boucle, c'est pour cela qu'on en parle maintenant.

```
push dx
 mov ax, [XResolution]
 shr ax, 5
 push ax
  mov ax, [YResolution]
  shr ax, 3
  push ax
   mov al, [BitsPerPixel]
   xor ah, ah
   shr ax, 3
  pop bx
  mul bx
 pop bx
 mul bx
pop dx
cmp ax, [maxResol]
jb lit_mode_suivant
mov [maxResol], ax
mov [mode_souhaite], cx
   jmp lit_mode_suivant
```

VII.3.e - Choix du mode graphique

```
arret_modes:
 mov cx, [mode_souhaite] ; On s'enquiert du mode souhaité
    mov ax, 0x4F01 ; demande infos sur le mode VESA
 mov di, ModeAttributes
    int 0x10
```

```
    mov di, hello ; on écrit dans hello
    mov ax, cx
    push cx
     mov ch, 3
     mov bl, 16
     call nombre_vers_chaine
     mov al, ':'
     stosb
     mov ch, 4
     mov bl, 10
     mov ax, [XResolution]
     call nombre_vers_chaine
     mov ax, ('*' << 8) + ' '
     stosw
     mov al, ' '
     stosb
     mov ax, [YResolution]
     call nombre_vers_chaine
     mov ax, ' '
     stosb
     mov ch, 2
     mov al, [BitsPerPixel]
     call nombre_vers_chaine
     mov ax, 13 ; on met 2 caractères d'un coup après la chaîne : un "\n" et le zéro terminal.
     stosw ; les caractères sont dépilés, c'est-à-dire qu'il faut placer le premier dans la zone basse
    pop cx
       mov si, hello
       call affiche_chaine

    mov al, [BitsPerPixel]
    shr al, 3
    mov byte [octetsParPixel], al
       mov ax, [WinASegment]

or ax, ax ; on teste l'adresse du segment de la fenêtre. Si elle est nulle, on passe en mode 0x13
       jnz adresse_OK
  adresse_mode_13h:
       mov word [mode_souhaite], 0x0013 ; infos du mode 0x13, le mode VGA
       mov word [WinASegment], 0xA000
       mov word [YResolution], 200
       mov word [XResolution], 320
    mov byte [octetsParPixel], 1
  adresse_OK:
    mov di, hello ; met l'adresse de la chaîne à lire dans le registre SI
    call lit_chaine ; On attend l'utilisateur pour nettoyer l'écran
```

VII.3.f - Passage dans le mode choisi

```
    mov ax, 0x4F02
    mov bx, [mode_souhaite]
    int 0x10 ; Changement de mode vidéo
    call nettoyage_ecran
```

VII.3.f.1 - Nettoyage de l'écran

Derrière ce terme ménager se cache juste le remplissage de l'intégralité de l'écran avec une couleur de fond. Comme d'habitude, l'idée n'est pas l'optimisation, le code super rapide comme l'ont tous les logiciels spécialisés. L'idée est d'avoir d'abord un code fonctionnel, créé au plus évident. L'algorithme que je vous propose ne prend en compte aucun prérequis autre qu'un mode VESA disposant d'une fenêtre A en mode écriture. Enfin, il me semble.

On écrit dans la mémoire vidéo par bloc de 64 ko. L'idée est donc de remplir tous ces blocs par la couleur de fond.

```
nettoyage_ecran:
 push di
 push es
 push ax
 push bx
 push cx
 push dx
 mov es, [WinASegment]; On lit l'adresse de départ de la mémoire vidéo

 mov cx, [YResolution]
 mov ax, [XResolution]
 mul cx ; Nombre de points total
 mov ax, dx
 mov cl, [octetsParPixel]
 mul cl
 mov cx, ax
 xor dx, dx
 xor bh, bh
 xor bl, bl
boucle_fenetres:
 push cx
  mov ax, 0x4F05
  int 0x10 ; Changement de fenêtre
  mov cx, [WinSize]
  shl cx, 9 ;Passage de ko en mots.
  xor di, di
 .point:
  push cx
  push bx
   mov cl, [octetsParPixel]
   xor ch, ch
   mov bx, couleur_defaut
  .couleur:
   inc bx
   mov al, byte [bx]
   stosb
   loop .couleur
  pop bx
  pop cx
  loop .point
  inc dx
 pop cx
 loop boucle_fenetres
.depile:
 xor dx, dx
 mov [bloc_courant], dl
 mov ax, 0x4F05
 int 0x10 ; Changement de fenêtre
 pop dx
 pop cx
 pop bx
 pop ax
 pop es
 pop di
 ret
;Fin nettoyage_ecran
```

VII.3.g - Affichage de points

Avec cette fonction, vous pouvez dorénavant dessiner ce que vous voulez où vous le voulez, en, normalement, n'importe quel mode VESA ! L'idée de base reste de mettre, octet par octet, la couleur spécifiée à l'adresse **couleurPoint** dans la mémoire vidéo, et de donner le bon numéro de plage. Ce bon numéro de plage est toujours basé sur la sainte formule : ordonnée * octetsParLigne + abscisse. Je suppose que le débordement de la multiplication me donne le numéro convoité.

```
;fonction affiche_point : on est déjà dans un mode graphique
;BX : Coordonnée X du point
```

```
;AX : Coordonnée Y du point
affiche_point:
 push bx ; On sauve les registres qu'on va manipuler
 push cx
 push es
 push di
 push dx
 push ax
  mov cx, word [BytesPerScanLine]
  mul cx
  mov di, ax
  push dx
   mov ax, bx
   xor ch, ch
   mov cl, byte [octetsParPixel]
   mul cx
   add di, ax
  pop dx
 .change_fenetre:
  mov ax, 0x4F05
  xor bh, bh
  xor bl, bl
  int 0x10 ; Changement de fenêtre
  mov es, [WinASegment] ; On va dans la mémoire vidéo
  mov bx, couleurPoint
 .couleur:
  inc bx
  mov al, byte [bx]
  stosb
  loop .couleur
 pop ax ; On restaure les registres manipulés
 pop dx
 pop di
 pop es
 pop cx
 pop bx
 ret
;fin de affiche_point
```

VII.3.h - Affichage de droites

On avait déjà parlé de l'algorithme de Bresenham, le revoici à l'identique. Il a été mis à jour pour refléter l'inversion des paramètres de affiche_point. On met sur la pile X1, puis Y1, puis X2 et enfin Y2.

```
;fonction affiche_ligne : on est déjà dans un mode graphique
affiche_ligne:
 jmp depart_affiche_ligne
Y2: dw 0
X2: dw 0
Y1: dw 0
X1: dw 0
deltaX: dw 0
deltaY: dw 0
incX: dw 0
incY: dw 0
e: dw 0
depart_affiche_ligne:
 push si
 push ax
 push bx
 push cx
 push dx
 push di
 push es
 mov ax, sp
 mov si, ax
 add si, 16 ; SI pointe sur Y2
```

```
 mov di, Y2
 mov ax, ds
 mov es, ax
 mov ax, ss
 mov ds, ax
 mov cx, 4
 rep movsw
 mov ax, es
 mov ds, ax
 mov ax, [X2]
 mov bx, [X1]
 sub ax, bx
 mov [deltaX], ax
 mov cx, [Y2]
 mov bx, [Y1]
 sub cx, bx
 mov [deltaY], cx
 or ax, ax ; test deltaX
 jnz test_deltaX_positif
 or cx, cx ; test deltaY
 jnz test_deltaY_deltaX_nul
fin_affiche_ligne:
 mov bx, [X2]
 mov ax, [Y2]
 call affiche_point
 pop es
 pop di
 pop dx
 pop cx
 pop bx
 pop ax
 pop si
 ret

deltaX_positif:
 or cx, cx
 jnz test_deltaY_deltaX_positif
 ;vecteur horizontal vers la droite
 mov cx, [deltaX]
 mov word [incX], 1
 mov word [incY], 0
 jmp ligne_H_V

test_deltaY_deltaX_nul:
;cx contient deltaY
 mov word [incY], 1
 mov word [incX], 0
 cmp cx, 0
 jns ligne_H_V
 neg cx
 mov word [incY], -1
ligne_H_V:
 mov bx, [X1]
 mov ax, [Y1]
avance_H_V:
 call affiche_point
 add bx, [incX]
 add ax, [incY]
 loop avance_H_V
 jmp fin_affiche_ligne

test_deltaX_positif:
 cmp ax, 0
 jns deltaX_positif
 or cx, cx ; CX contient DeltaY
 jnz test_deltaY_deltaX_negatif
 ;vecteur horizontal vers la gauche
 mov cx, [deltaX]
 neg cx
 mov word [incX], -1
 mov word [incY], 0
 jmp ligne_H_V
```

```
charge_registres:
    shl cx, 1
    shl ax, 1
    mov [deltaY], cx
    mov [deltaX], ax
    mov bx, [X1]
    mov ax, [Y1]
    ret

charge_e_deltaX_et_cmp_X2:
    mov [e], ax
    call charge_registres
    mov cx, [X2]
    ret

charge_e_deltaY_et_cmp_Y2:
    mov [e], cx
    call charge_registres
    mov cx, [Y2]
    ret

affiche_et_charge_eY:
    call affiche_point
    add ax, [incY]
    mov dx, [e]
    ret

affiche_et_charge_eX:
    call affiche_point
    add bx, [incX]
    mov dx, [e]
    ret

octants1_et_4:
    call charge_e_deltaX_et_cmp_X2
depart_boucle1:
    call affiche_et_charge_eX
    cmp bx, cx
    je fin_affiche_ligne
    sub dx, [deltaY]
    cmp dx, 0
    jns X_pret1
    add ax, [incY]
    add dx, [deltaX]
X_pret1:
    mov [e], dx
    jmp depart_boucle1

deltaY_positif_deltaX_negatif:
    neg ax
deltaY_positif_deltaX_positif:
    mov word [incY], 1
    ;deltaY > 0, deltaX > 0
    cmp ax, cx
    jae octants1_et_4
    neg ax
    call charge_e_deltaY_et_cmp_Y2
depart_boucle2_et_3:
    call affiche_et_charge_eY
    cmp ax, cx
    je fin_affiche_ligne
    add dx, [deltaX]
    cmp dx, 0
    jns X_pret2_et_3
    add bx, [incX]
    add dx, [deltaY]
X_pret2_et_3:
    mov [e], dx
    jmp depart_boucle2_et_3

octant5:
```

```
        call charge_e_deltaX_et_cmp_X2
    depart_boucle5:
        call affiche_et_charge_eX
        cmp bx, cx
        je fin_affiche_ligne
        sub dx, [deltaY]
        cmp dx, 0
        js X_pret5
        add ax, [incY]
        add dx, [deltaX]
    X_pret5:
        mov [e], dx
        jmp depart_boucle5

    octant8:
        neg cx
        call charge_e_deltaX_et_cmp_X2
    depart_boucle8:
        call affiche_et_charge_eX
        cmp bx, cx
        je fin_affiche_ligne
        add dx, [deltaY]
        cmp dx, 0
        jns X_pret8
        add ax, [incY]
        add dx, [deltaX]
    X_pret8:
        mov [e], dx
        jmp depart_boucle8

    test_deltaY_deltaX_positif:
     mov word [incX], 1
     cmp cx, 0
     jns deltaY_positif_deltaX_positif
     ;deltaY < 0, deltaX > 0
     mov word [incY], -1
     neg cx
     cmp ax, cx
     jae octant8
     neg cx
     jmp octants6_et_7
    test_deltaY_deltaX_negatif:
     mov word [incX], -1
     cmp cx, 0 ; cx contient deltaY
     jns deltaY_positif_deltaX_negatif
     ;deltaY < 0, deltaX < 0
     mov word [incY], -1
     cmp ax, cx ; ax contient deltaX
     jbe octant5
     neg ax
    octants6_et_7:
     call charge_e_deltaY_et_cmp_Y2
    depart_boucle6_et_7:
     call affiche_et_charge_eY
     cmp ax, cx
        je fin_affiche_ligne
        add dx, [deltaX]
        cmp dx, 0
        js X_pret6_et_7
        add bx, [incX]
        add dx, [deltaY]
    X_pret6_et_7:
     mov [e], dx
     jmp depart_boucle6_et_7
    ;AFFICHE_LIGNE ENDP
```

VII.3.i - Affichage d'un alphabet

Voici la grande nouveauté de ce chapitre : un alphabet ! En mode graphique, oui madame, dessiné par mes soins, à la va-comme-j'te-pousse. Je l'ai fait parce qu'il me semble que bientôt, nous n'en aurons plus. Et puis, ça permettra d'afficher facilement des "é", "ç" et autres "à". Dès que j'aurai mis ces caractères, bien sûr. La fonction est très simple : on lui donne le numéro du caractère, avec A = 1, dans **cx**. **ax** contient l'ordonnée du coin supérieur gauche du caractère et **bx** son abscisse.

```
affiche_caractere:
 push bx
 push dx
 push si
 push cx
 push ax
 push ax
 mov ax, cx
 mov cx, 6
 mul cx
 mov si, Alphabet; adresse de la lettre
 add si, ax; si contient l'adresse de la lettre
.colonne:
 pop ax
 push cx
 mov dl, 0b10000000
 mov cx, 8; On affiche 8 colonnes
.ligne:
 push ax
 mov al, [si]; On charge l'octet à afficher
 test al, dl
 jz .suite
 pop ax
 call affiche_point
 push ax
.suite:
 shr dl, 1
 inc bx
 pop ax
 loop .ligne
 pop cx
 inc ax; passage à la ligne suivante
 sub bx, 8
 push ax
 inc si
 loop .colonne
 pop ax
 pop ax
 pop cx
 pop si
 pop dx
 pop bx
 ret
```

VII.4 - Du code

- **amorce.asm**
- **affichageTexte.asm**
- **alphabet.asm**
- **ecran.asm**

VIII - Assembleur : passage en mode protégé

Relu par **ClaudeLELOUP**.

Au chapitre précédent, nous avons patiemment retrouvé le contrôle de l'écran, dans ses grandes lignes. A une phase de reconstruction suit une phase de destruction. Il fallait que je m'y colle il y a longtemps déjà, j'ai écumé le net et la documentation AMD, je lis l'anglais comme un américain maintenant, voilà, petits Frenchies, voilà où nous en sommes : le mode protégé !

Les plus futés auront remarqué que nous n'avons manipulé jusqu'à présent que des registres de 16 bits. Or, nous avons tous, au moins, des machines 32 bits. Nous avons utilisé des adresses de la forme segment:offset, codées sur 20 bits par une arithmétique assez odieuse, alors qu'un seul registre de 32 bits nous aurait évité cela. Il y avait une raison. Une vraie raison, officielle en diable : pour faire mieux, il faut relever le challenge des débutants, passer en mode protégé. Alors, je vais faire un petit topo sur tout ça.

VIII.1 - Histoire de l'informatique

Un bien pompeux titre, mais ne croyez pas tout ce qu'on vous raconte. Il ne s'agit ici que de prendre en compte de simples considérations historiques qui nous expliquent pourquoi on en est là.

VIII.1.a - L'architecture des premiers Personal Computers

C'est la société IBM (dont le nom signifie quelque chose comme "machines intelligentes pour les affaires") qui a gagné le marché des ordinateurs personnels. N'oublions pas qu'à cette époque (le début des années 1980), les ordinateurs existent, sont assez répandus mais ne sont pas non plus à la disposition de tout un chacun. IBM va jeter les bases de l'informatique personnelle. Bien évidemment, la concurrence est rude. IBM va choisir de fabriquer des ordinateurs de série à bas coût, avec des processeurs qu'elle peut produire en nombre.

Et il se trouve que ce sera un succès. Modeste par rapport au nombre d'ordinateurs vendus quotidiennement aujourd'hui, mais suffisamment important pour qu'IBM occupe une grosse part de marché. Les développeurs vont donc fournir des programmes développés pour cette machine, l'IBM PC, basée sur un processeur 8086 (en fait un 8088, mais c'est le 8086 qui a légué son nom à la postérité). Les utilisateurs de machines vont ensuite vouloir faire fonctionner ces mêmes programmes sur d'autres machines. Cela n'est possible que si les machines sont **compatibles**. Au vu des parts de marché d'IBM, la concurrence est obligée de s'aligner et d'adopter la même architecture que celle d'IBM.

Cette architecture permet d'accéder à 2^{20} octets de mémoire vive, plus ou moins un. Ca fait 1 mégaoctet. Mais c'est une machine 16 bits, ce qui fait qu'elle ne peut représenter que 2^{16} octets, soit 64 kilooctets. Pour adresser 1 Mo, elle doit ruser. La ruse consiste à utiliser deux zones mémoire du processeur pour adresser toute la mémoire. Du coup, on a deux fois 16 bits, ce qui couvre nos 20 bits d'adresse. Oui, mais deux fois 16 bits, ça fait 32 bits, on en a trop. Et c'est à ce moment, je ne sais pas pourquoi mais ils avaient leurs raisons, que les collègues de chez IBM ont décidé que leurs deux nombres recouvriraient en partie la zone d'adressage. Il y a 12 bits qui se recouvrent ! 0x1222:2220 correspond à exactement la même case mémoire que 0x1444:0000 ou 0x1000:4440 ou 1111:3330 ! Ce truc délirant, ça s'appelle l'arithmétique des pointeurs.

VIII.1.b - Le piège d'IBM

Peut-être que les gars qui ont pondu ça, ils étaient fatigués, sous pression, les commerciaux leur ont dit que c'était juste un petit truc comme ça, que sais-je. Mais le fait est qu'on a eu l'arithmétique des pointeurs. Et que l'IBM PC a eu un succès fou. Là est tout le drame. Car non seulement la concurrence a dû faire des machines compatibles, mais de surcroît IBM aussi ! Quand l'entreprise a voulu enlever ces zones mémoire qui se recouvraient, par souci

de compatibilité, elle n'a pas pu. Néanmoins, il fallait obligatoirement dépasser cette limite de 1 Go de mémoire, et quitter cette méchante façon d'adresser les octets. Que faire ?

VIII.1.c - Le mode protégé

IBM a inventé le mode protégé. Appelons le mode historique le mode réel. Dans un ordinateur en mode réel, n'importe quel programme peut voir l'ensemble de la mémoire et l'écrire. Cela peut poser des problèmes, notamment quand votre voisin décide d'écrire chez vous. Il faut une astuce pour rendre la chose plus difficile.

Grâce à une série d'instructions à faire dans le bon ordre, de zones mémoire judicieusement choisies et d'un processeur le permettant (donc au moins un 80286), on peut utiliser 32 bits d'adressage direct et interdire à un programme d'aller voir en-dehors de l'espace qui lui est alloué. C'est cela, le mode protégé.

VIII.2 - Problèmes avec le mode protégé

Il faut s'en douter, si on n'était pas en mode protégé jusqu'à présent, c'est que ce mode pose des problèmes qu'on peut apprécier ne pas avoir. Le plus gros et le plus velu, de mon point de vue, est celui-ci : en mode protégé, adieu les interruptions. Fini les services du BIOS. Plus de changement de mode graphique. Plus d'affichage de caractère, plus de clavier. Plus rien. Il faut tout refaire. Tout.

VIII.3 - Solutions en mode protégé

Bon, ben quand faut y aller, faut y aller.

Techniquement, pour passer en mode protégé, il suffit de ceci :

```
mov eax,cr0
or ax,1
mov cr0,eax
```

En langage humain, il suffit de passer le bit n° 0 du registre **CR0** à 1. Comme le registre **CR0** n'est pas éditable par le microprocesseur, on le passe d'abord dans **EAX**, on fait le **OR** qui permet de mettre le bit n° 0 à 1 sans changer tout le reste, et on remet EAX dans **CR0**.

Mais ce n'est pas suffisant. En effet, en mode protégé, la mémoire peut être segmentée. C'est un vilain mot qui signifie qu'il faut définir des segments de mémoire. Les anciens registres, tels que CS, DS et ES existent toujours en mode protégé. Ils font toujours 16 bits, mais ce ne sont plus les bits de poids fort de l'adresse. Ils sont devenus des offsets, des décalages. Ils correspondent au décalage nécessaire pour atteindre le descripteur de segment correspondant dans le tableau global des descripteurs, Global Descriptor Table (GDT).

En mode protégé, le processeur va regarder le segment correspondant à son instruction, ajouter cette adresse à son registre GDT, lire le descripteur de segment à cet endroit, en conclure quant à l'adresse concernée et y aller.

VIII.3.a - Le GDT (Global Descriptor Table)

Qu'on appelle, en français, le tableau global des descripteurs. C'est une zone mémoire, à une adresse spécifiable comme bon nous semble. Elle est spécifiquement liée à deux instructions particulières, mais une seule nous intéresse ici : **LGDT**, Load Global Descriptor Table. Elle prend un seul argument, l'adresse (32 bits maintenant, donc) d'une toute petite zone mémoire, que nous allons appeler pointeurGDT:, comme c'est original. Cette zone contient deux nombres dans cet ordre :

- la taille en octets du tableau global des descripteurs, sous forme de mot (16 bits) ;

- l'adresse de début du tableau global des descripteurs, sous forme de double-mot (32 bits), nécessairement.

Comme le monde entier suppose tout à fait intelligemment que si on a une adresse et une taille de tableau à donner, c'est qu'il nous faut le remplir, et que s'il s'appelle "Tableau Global des Descripteurs", c'est qu'il doit contenir des descripteurs, voyons un peu les descripteurs.

VIII.3.b - Les Descriptors

Oui, les descripteurs. Un descripteur est une structure de données qui décrit, d'où son nom, quelque chose. J'ai parlé avant de segments en mémoire, et bien mettons les deux ensemble : dans le tableau global des descripteurs, les descripteurs décrivent des segments. Il en faut au moins deux : un pour le code, un autre pour les données. C'est comme ça, c'est imposé par le processeur. Par contre, on a le droit de décrire le même segment.

Le descripteur à proprement parler a la structure qui suit.

- Base : adresse linéaire du début du segment, 32 bits : la limite peut être toute la mémoire adressable.
- Limite : taille en octets du segment, 20 bits avec astuce. Normalement, on devrait avoir 32 bits.
- G : drapeau. Si à 1, la Limite est donnée en octets. Sinon, elle est en multiples de 4096 octets. 1 bit. Permet de simuler les 32 bits attendus sur Limite.
- S : drapeau. Si à 1, il s'agit d'un segment système (inaccessible aux autres programmes). Sinon, il s'agit d'un segment de code ou de données. 1 bit.
- Type : type de segment (données, code, etc.). 4 bits.
- DPL : niveau de privilège minimal pour accéder au segment. 2 bits.
- P : Si à 1, le segment est présent dans la mémoire principale. 1 bit.
- D/B : Taille des éléments du segment : opérandes en mode Code, pile en mode Données. 0 : 16 bits ou SP, 1 : 32 bits ou ESP. 1 bit.
- AVL : champ utilisable par le programmeur. 1 bit.

Ce qui fait royalement 63 bits. Un bit ne sert à rien et porte le total à 8 octets, répartis comme suit (attention c'est stéganologique) :

```
 15                      7                    0
---------------------------------------------------
| Limite, bits 0-15                               |
---------------------------------------------------
| Base, bits 0-15                                 |
---------------------------------------------------
| P  DPL   S     Type     Base, bits 16-23        |
---------------------------------------------------
|    Base, bits 24-31     G  D/B 0 AVL Limite,fin|
---------------------------------------------------
```

Voici les valeurs possibles de Type.

- 0 : Read Only. Lecture seule.
- 1 : Read Only, accessed. Lecture seule, accédé ?
- 2 : Read/Write. Lecture/Ecriture.
- 3 : Read/Write, accessed. Lecture/Ecriture, accédé ?
- 4 : Read Only, expand down. Lecture seule, s'accroît vers le bas.
- 5 : Read Only, expand down, accessed. Lecture seule, s'accroît vers le bas, accédé ?
- 6 : Read/Write, expand down. Lecture/Ecriture, s'accroît vers le bas.
- 7 : Read/Write, expand down, accessed. Lecture/Ecriture, s'accroît vers le bas, accédé ?
- 8 : Execute Only. Exécution seule.
- 9 : Execute Only, accessed. Exécution seule, accédé ?
- A : Execute/Read. Exécution/Lecture.
- B : Execute/Read, accessed. Exécution/Lecture, accédé ?

- C : Execute Only, conforming. Exécution seule, standard.
- D : Execute Only, conforming, accessed. Exécution seule, standard, accédé ?
- E : Execute/Read, conforming. Exécution/Lecture, standard.
- F : Execute/Read, conforming, accessed. Exécution/Lecture, standard, accédé ?

Source : **http://www.c-jump.com/CIS77/ASM/Protection/W77_0090_segment_descriptor_cont.htm**

Nous avons besoin de trois segments.

- Le segment NULL, nécessaire au processeur en cas d'erreur de segmentation. Tout à zéro, fin du match. `gdt: db 0,0,0,0,0,0,0,0`
- Le segment de code. Limite à 0xFFFFF, Base à 0, P à 1, DPL à 0, S à 1, Type à Exécution/Lecture accédé, AVL à 1, D/B à 1 et G à 1. `gdt_cs: db 0xFF,0xFF,0x0,0x0,0x0,10011011b,11011111b,0x0`
- Le segment de données. Limite à 0xFFFFF, Base à 0, P à 1, DPL à 0, S à 1, Type à Lecture/Ecriture accédé, AVL à 1, D/B à 1 et G à 1. `gdt_ds: db 0xFF,0xFF,0x0,0x0,0x0,10010011b,11011111b,0x0`

VIII.4 - De la pratique

On l'a bien mérité, voici le secteur d'amorçage qui passe en mode protégé avant de donner la main à un noyau à venir.

```
%define BASE 0x100  ; 0x0100:0x0 = 0x1000
%define KSIZE 2
%define BOOT_SEG 0x07c0

BITS  16
org  0x0000 ; Adresse de début bootloader

;; Initialisation des segments en 0x07C0
 mov  ax,  BOOT_SEG
 mov  ds,  ax
 mov  es,  ax
 mov  ax,  0x8000 ; pile en 0xFFFF
 mov  ss,  ax
 mov  sp,  0xf000

;; Affiche un message
 mov  si,  msgDebut
 call afficher

;; Charge le noyau
initialise_disque:    ; Initialise le lecteur de disque
 xor  ax,  ax
 int  0x13
 jc initialise_disque ; En cas d'erreur on recommence (sinon, de toute façon, on ne peut rien faire)

lire:
 mov  ax,  BASE  ; ES:BX = BASE:0000
 mov  es,  ax
 xor  bx,  bx
 mov  ah,  2   ; Fonction 0x02 : chargement mémoire
 mov  al,  KSIZE  ; On lit KSIZE secteurs
 xor  ch,  ch   ; Premier cylindre (n° 0)
 mov  cl,  2   ; Premier secteur (porte le n° 2, le n° 1, on est dedans, et le n° 0 n'existe pas)
 xor  dh,  dh   ; Tête de lecture n° 0
 ; Toujours pas d'identifiant de disque, c'est toujours le même.
 int  0x13    ; Lit !
 jc lire     ; En cas d'erreur, on recommence

;; Passe en mode protégé
 cli
 lgdt  [pointeurGDT] ; charge la gdt
 mov  eax, cr0
 or   ax,  1
 mov  cr0,  eax  ; PE mis a 1 (CR0)
```

```
 jmp next
next:
 mov  ax,  0x10 ; offset du descripteur du segment de données
 mov  ds,  ax
 mov  fs,  ax
 mov  gs,  ax
 mov  es,  ax
 mov  ss,  ax
 mov  esp, 0x9F000

 jmp dword 0x8:BASE << 4; réinitialise le segment de code

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Synopsis: Affiche une chaîne de caractères se terminant par NULL ;;
;; Entrée:   DS:SI -> pointe sur la chaîne à afficher    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
afficher:
 push ax
 push bx
.debut:
 lodsb   ; ds:si -> al
 cmp  al, 0 ; fin chaîne ?
 jz .fin
 mov  ah, 0x0E ; appel au service 0x0e, int 0x10 du BIOS
 mov  bx, 0x07 ; bx -> attribut, al -> caractère ASCII
 int 0x10
 jmp .debut

.fin:
 pop bx
 pop ax
 ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
msgDebut db "Chargement du kernel", 13, 10, 0

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
gdt:
 db 0x00, 0x00, 0x00, 0x00, 0x00, 00000000b, 00000000b, 0x00
gdt_cs:
 db 0xFF, 0xFF, 0x00, 0x00, 0x00, 10011011b, 11011111b, 0x00
gdt_ds:
 db 0xFF, 0xFF, 0x00, 0x00, 0x00, 10010011b, 11011111b, 0x00
gdtend:

pointeurGDT:
 dw gdtend-gdt    ; taille
 dd (BOOT_SEG << 4) + gdt ; base

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Rien jusqu'à 510
times 510-($-$$) db 0
dw 0xAA55
```

IX - Assembleur : mode protégé - Retrouver ses interruptions

Relu par **ClaudeLELOUP**.

Parce que quand même, ce n'est pas chic, c'était bien pratique, les interruptions.

IX.1 - Au fait, pourquoi a-t-on inventé les interruptions ?

C'est vrai, ça, pourquoi les interruptions ? La raison en est toute baignée de considérations calculatoires. Si le processeur devait regarder régulièrement si, par exemple, un caractère est en train d'être entré au clavier, mes pauvres enfants, on ne s'en sortirait pas et on aurait encore bien du mal à faire fonctionner un simulateur de calculette. Pour gagner énormément de temps et pour qu'aucun programmeur n'oublie de vérifier l'entrée clavier, c'est l'inverse que l'on fait : c'est le contrôleur clavier qui interrompt le processeur en lui disant : "Je ne sais pas si cela vous intéresse, mon bon ami, mais j'ai ici une touche qui vient d'être frappée." Ca, c'était pour la poésie. Dans la vraie vie, ça se passe plutôt comme "Touche. -6^{e} procédure. -STOP ! 6^{e} procédure. -6^{e} procédure faite. -OK, j'y retourne." Et la 6^{e} procédure en question, c'est l'interruption clavier. Et si un contrôleur de clavier peut appeler une interruption, alors un programme aussi. C'est ce détournement que nous avons utilisé précédemment, et c'est même prévu pour être utilisé à outrance.

IX.2 - Et une interruption, c'est quoi ?

C'est un programme, sous-programme ou routine, peu importe le nom, dont l'exécution interrompt le cours normal d'un autre programme. Un peu comme quand l'exécution automatique d'un CD se déclenche quand vous êtes en train de rédiger votre site, par exemple. On est interrompu. Cette routine a ceci de particulier qu'elle se termine par l'instruction **IRET**, comme Interrupt RETurn, instruction qu'on ne trouve qu'en langage assembleur (ou alors d'autres langages parfaitement exotiques). Voici donc notre première routine de gestion d'interruption : **entreeInterruption : IRET**. Donc, pour appeler une routine définie comme interruption, on ne peut pas utiliser **CALL**. Ce sera **INT**.

IX.3 - Comment j'en fais ?

Comme d'habitude chez nous, on va faire dans le simple, quitte à bulldozer un brin. Alors, en premier, on doit pratiquer la même chose que pour le passage en mode protégé. C'est-à-dire que l'on doit créer un **IDT** (Interrupt Descriptor Table), tableau de descripteurs d'interruption, contenant, comme c'est étrange, des descripteurs d'interruption. L'instruction INT prend un opérande de 8 bits, on peut donc avoir 256 descripteurs d'interruption. Voici comment je fais :

```
    mov eax, IDTBASE     ; Adresse de l'IDT
    mov ebx, interruptionParDefaut
  ; On va remplir toute la table avec l'interruption par défaut : bx contient les bits de poids faible
    mov ecx, IDTSIZE     ; Nombre de vecteurs d'interruption
    call metInterruptionNFois

    lidt [idtptr] ; charge l'idt
    sti

   ;;Met l'interruption n fois dans l'idt
  metInterruptionNFois:
    mov edx, ebx
    shr edx, 16
    mov [descripteurInterruption], bx ;bx contient les bits de poids faible
    mov [descripteurInterruption + 6], dx; et dx ceux de poids fort
    mov edi, eax
  .remplitIDTPIC:
    mov esi, descripteurInterruption
    movsd
    movsd
```

```
    loop .remplitIDTPIC
    ret

interruptionParDefaut:
    iret

idtptr:
    dw IDTSIZE << 3    ; limite
    dd IDTBASE      ; base

descripteurInterruption:
    dw 0, 1 * 8, INTGATE, 0
```

Notez l'instruction **LIDT** qui fait exactement la même chose que **LGDT** mais pour l'IDT, et le rétablissement des interruptions l'instant d'après. Notez aussi que nous travaillons en 32 bits, gloria alleluia.

IX.4 - Du PIC

Les périphériques d'un ordinateur compatible PC ne sont pas branchés directement sur le processeur. Ils sont branchés sur un élément qu'on appelle le PIC (Programable Interruption Controller), le contrôleur d'interruption programmable. Il y en a deux, l'un piloté par l'autre. Le pilote s'appelle le maître, l'autre l'esclave. La raison d'être du PIC est donc de déclencher des interruptions. Or, il se trouve que dans son nom, il y a "Programable", ce qui doit signifier qu'on peut faire quelques configurations. Le PIC n'étant ni la mémoire ni le processeur, pour y accéder, on va parler :

IX.4.a - Du port d'entrée/sortie

Notre ordinateur ne fait pas qu'écrire et lire de la mémoire. Il interagit aussi avec des choses, qui sont des appareils électroniques. Cette interaction se fait au travers de 8 fils qui relient tous ces appareils. Le processeur peut écrire et lire sur ce port par les instructions **OUT** et **IN**. Et il écrit et lit un octet. L'adresse de l'appareil connecté est donnée en premier paramètre : il faut la connaître. Pour le PIC, il s'agit de 0x20 et 0x21 pour le maître et 0xA0 et 0xA1 pour l'esclave.

IX.4.b - ICW1

C'est la première valeur à envoyer au PIC pour le configurer, notamment pour qu'il pointe vers nos interruptions. Première signifie ici le numéro d'ordre dans la séquence. C'est obligatoirement celle-là. ICW signifie Initialisation Command Word, mot de la commande d'initialisation. Voici sa structure :

Bit	7	6	5	4	3	2	1	0
Valeur	A5-A7			1	LTIM	ADI	SNGL	IC4

- IC4: 0 = Pas de ICW4, 1 = ICW4 requis.
- SNGL: 1 = Un seul PIC, 0 = D'autres PIC sont montés en cascade.
- ADI: Intervalle d'adressage. Utilisé uniquement dans les 8085, pas dans les 8086 (donc pas pour nous).
- LTIM: Mode de déclenchement des interruptions: 1 = toutes les lignes de requête d'interruption sont déclenchées par niveau. 0 = elles sont déclenchées par front. Pour nous, 0 c'est bien.
- A5-A7: Utilisé uniquement dans les 8085.

Le PIC occupe deux adresses de port, ICW1 est sur la première.

Source : **http://www.thesatya.com/8259.html**

Ca va nous donner : 00010001b pour le maître, et 00010001b pour l'esclave.

```
mov al, 0x11 ; Initialisation de ICW1
out 0x20, al ; maître
out 0xA0, al ; esclave
```

IX.4.c - ICW2

ICW2 est sur la seconde adresse, et correspond à l'index de la première interruption dédiée au PIC dans l'IDT. Il est indiqué nécessairement après ICW1.

```
%define INTERRUPTION_PIC_MAITRE 0x20
%define INTERRUPTION_PIC_ESCLAVE 0x70
 mov al, INTERRUPTION_PIC_MAITRE ; Initialisation de ICW2
 out 0x21, al ; maître, vecteur de départ = 32
 mov al, INTERRUPTION_PIC_ESCLAVE
 out 0xA1, al ; esclave, vecteur de départ = 96
```

IX.4.d - ICW3

Pour le PIC maître, chaque bit **x** mis à 1 indique que l'IRQ**x** est utilisée par un PIC esclave. Pour l'esclave, cela indique le numéro de l'IRQ qu'il utilise chez son maître. ICW3 est sur la seconde adresse. Le standard est de mettre l'esclave sur l'IRQ2.

```
mov al, 0x04 ; initialisation de ICW3
out 0x21, al
mov al, 0x02 ; esclave
out 0xA1, al
```

IX.4.e - ICW4

ICW4 est sur la seconde adresse.

Bit	7	6	5	4	3	2	1	0
Valeur	0			SFNM	BUF	M/S	AEOI	Mode

- SFNM: 1 = Special Fully Nested Mode, 0 = Fully Nested Mode. Pas de spécial pour moi, merci.
- M/S: 1 = Maître, 0 = eSclave
- AEOI: 1 = Auto End of Interrupt (fin automatique d'interruption), 0 = Normal.
- Mode: 0 = 8085, 1 = 8086

Dans notre cas, ce sera donc 5 pour le maître et 1 pour l'esclave.

```
mov al, 0x05 ; initialisation de ICW4
out 0x21, al
mov al, 0x01 ; esclave
out 0xA1, al
```

IX.4.f - OCW1

Notons que c'est l'ordre des opérations qui permet au PIC de comprendre de quoi on cause. Une fois l'initiation faite, on peut envoyer des OCW, Operational Command Word, mot de commande opérationnelle. OCW1 est sur la seconde adresse, et détermine quelles sont les interruptions masquées. Pour utiliser toutes les interruptions :

```
    xor al, al ; masquage des interruptions
    out 0x21, al
    out 0xA1, al
```

IX.4.g - Fin de traitement d'interruption

Lorsqu'un PIC déclenche une interruption, il faut lui dire qu'elle s'est bien passée. Il y a 8 interruptions par PIC, que l'on va traiter ainsi :

```
interruptionPICParDefaut:
 mov al,0x20 ; EOI (End Of Interrupt)
 out 0x20,al ; qu'on envoie au PIC
 iret
```

Avant de quitter l'interruption, si elle a été déclenchée par un PIC, on écrit sur le port 0x20 l'octet 0x20, qui signifie "fin de l'interruption" et qui s'appelle EOI.

Une fois tout ceci fait, on a des interruptions que l'on peut utiliser. Malheureusement, il n'y a rien dedans.

X - Remerciements

Merci à l'équipe de developpez.com, et en particulier à **ClaudeLELOUP** pour sa relecture soigneuse et à **E-Sh4rk** pour sa maîtrise des puissances de 2.