Mémoire : disposition et accès Chapitre 4

Dans le premier chapitre, on a vu les formats de base pour les données en mémoire. Le troisième chapitre montre comment un ordinateur organise physiquement ses données. Le but de ce chapitre est d'illustrer l'accès à la mémoire des processeurs 80x86.

4.0 Vue d'ensemble du chapitre

Ce chapitre constitue un point important entre les sections Une et Deux ("Organisation de la machine" et "Bases du langage assembleur", respectivement). Du point de vue de l'organisation matérielle, il aborde l'adressage et l'organisation de la mémoire, les modes d'adressage du CPU et les représentations des données dans la mémoire. Du point de vue du programmeur, il traite du jeu de registres 80x86, des adressages 80x86 et des types de données composites. C'est un chapitre pivot. Si vous ne comprenez pas le contenu de ce chapitre, vous aurez des difficultés à comprendre les chapitres qui suivent. Par conséquent, vous devrez l'étudier soigneusement avant de continuer.

Ce chapitre commence par examiner les registres des processeurs 80x86. Ces processeurs se composent d'un ensemble de registres généraux, de registres de segment et de certains registres dont les fonctions sont spéciales. Certains membres de la famille 80x86 fournissent des registres additionnels, même si les applications ne s'en servent pas généralement.

Après avoir présenté les registres, ce sera le tour de l'organisation de la mémoire et de la segmentation. La segmentation de la mémoire est un sujet difficile pour beaucoup de débutants. C'est pourquoi ce livre n'en parle pas au cours de ses premiers chapitres. Néanmoins, il s'agit d'un concept puissant avec lequel vous devrez vous familiariser afin de pouvoir écrire des programmes 80x86 autres qu'élémentaires[1].

Les modes d'adressage 80x86 sont probablement le sujet le plus important de ce chapitre. Tant que vous ne les aurez complètement maîtrisés, vous ne serez pas capable d'écrire des programmes raisonnables en assembleur. Donc, n'allez pas au-delà de ce chapitre avant de vous sentir complètement à l'aise avec ces modes. On abordera également les modes étendus du processeur 80386 (et ultérieurs). Les connaître n'est pas vraiment important pour le moment, mais si vous les apprenez, vous pourrez économiser beaucoup de temps quand vous écrirez du code pour ces processeurs[2].

Et finalement nous présenterons une poignée d'instructions 80x86. Bien qu'elles soient insuffisantes pour écrire des véritables programmes en assembleur, elles vous seront utiles pour manipuler des variables et des structures de données – le sujet du prochain chapitre.

4.1 Les CPU 80x86 : Un point de vue de programmeur

Le temps d'aborder de véritables processeurs est venu : 8088/8086, 80188/80186, 80286 et 80386/80486/Pentium. Le Chapire Trois a traité plusieurs aspects matériels d'un système informatique. Même si ces aspects affectent la manière d'écrire les programmes, il y a plus que des bus, des cycles et des pipelines dans un processeur. Il est temps de regarder les composants du CPU qui sont les plus visibles pour un programmeur.

La plus visible de ces composantes est le jeu de registres. Comme dans le cas de nos processeurs hypothétiques, les puces 80x86 ont un jeu de registres incorporé. Pour chacun de ces processeurs, ces jeux constituent un surensemble des registres vus au chapitre précédent. Le meilleur point de départ est le jeu des puces 8088, 8086, 80188 et 80186, étant donné qu'elles ont les mêmes registres. Dans la discussion qui suit, le terme "8086" signifiera n'importe laquelle de ces puces.

Les concepteurs Intel ont classifié les registres du 8086 en trois catégories : registres généraux, registres de segment et registres divers. Les registres généraux sont ceux qui peuvent être utilisés comme opérandes dans

[1]Cependant, la segmentation ne concerne que la programmation 16 bits, l'espace d'adressage en 32 bits est de 4 Giga-octets, ce qui est largement suffisant, n.d.t.
[2]Ce qui sera presque toujours le cas, car, à l'heure de la traduction de ce chapitre on est désormais au Pentium IV, n.d.t.

des instructions arithmétiques, logiques, etc. Bien qu'ils soient "généraux", chacun d'eux a sa fonction spéciale. Intel utilise très largement le terme "général". Les registres de segment servent à accéder à des blocs de mémoire, appelés justement *segments*. Pour plus de détails sur la nature exacte de ce ces registres, voir "Segments 80x86" au paragraphe 4.3. La dernière catégorie constitue un pêle-mêle de registres 80x86 de différentes natures. Il y en a deux vraiment spéciaux qui feront l'objet des prochaines pages.

4.1.1 registres généraux 80x86

Il y a huit registres généraux de 16 bits dans le processeur 8086 : ax, bx, cx, dx, si, di, bp et sp. Alors que vous pouvez les utiliser à loisir dans des instructions de calcul, certaines instructions fonctionneront mieux avec certains registres ou même demanderont exclusivement un registre spécifique. Ce qui est tout dire pour des registres généraux.

Le registre ax (Accumulateur) est l'endroit où la plupart des opérations arithmétiques et logiques ont lieu. Bien que vous puissiez utiliser également d'autres registres pour faire ces choses, il est souvent plus efficace d'utiliser l'accumulateur. Le registre bx (Base) a également ses fonctions spéciales. Il est généralement utilisé pour charger des adresses indirectes, de façon très semblable au registre bx qu'on a vu avec les processeurs x86. Le registre cx (Compteur), comme son nom l'implique, sert à compter des choses. Vous l'utiliserez souvent pour décompter le nombre d'itérations d'une boucle, ou pour spécifier le nombre de caractères dans une chaîne. Le registre dx (Données) a deux fonctions spéciales : il charge le débordement de certaines opérations arithmétiques et il charge des adresses E/S pendant l'accès à des données sur le bus des entrées/sorties 80x86.

Les registres si et di (*Source Index* et *Destination Index*), ont également leurs fonctions spéciales. On peut les utiliser comme des pointeurs (de façon très analogue au registre bx), pour accéder indirectement à la mémoire. On peut également les utiliser avec les instructions 80x86 pour le traitement des chaînes de caractères.

Le registre bp (Base Pointer) ou pointeur de base, est semblable au registre bx. On l'utilise généralement pour accéder aux paramètres et aux variables locales d'une procédure.

Le registre sp (Stack Pointer) ou pointeur de pile, a une fonction très spéciale : il maintient la pile du programme. Normalement, il ne faudrait pas l'utiliser pour des opérations arithmétiques. Le bon fonctionnement des programmes dépend souvent de l'usage judicieux de ce registre.

A côté des registres de 16 bits, les CPU 8086 disposent aussi de registres de 8 bits. Intel les appelle al, ah, bl, bh, cl, ch, dl et dh. Vous aurez peut-être remarqué une similarité entre ces noms et les noms des registres de 16 bits (ax, bx, cx et dx, pour l'exactitude). Les registres de 8 bits ne sont pas indépendants. al veut dire « AX low order byte » (octet le moins significatif de de ax) et ah veut dire « Ax high byte » (octet le plus significatif de ax). Les noms des autres six registres ont la même signification, mais par rapport à bx, cx et dx. La figure 4.1 donne un aperçu visuel du jeu des registres généraux.

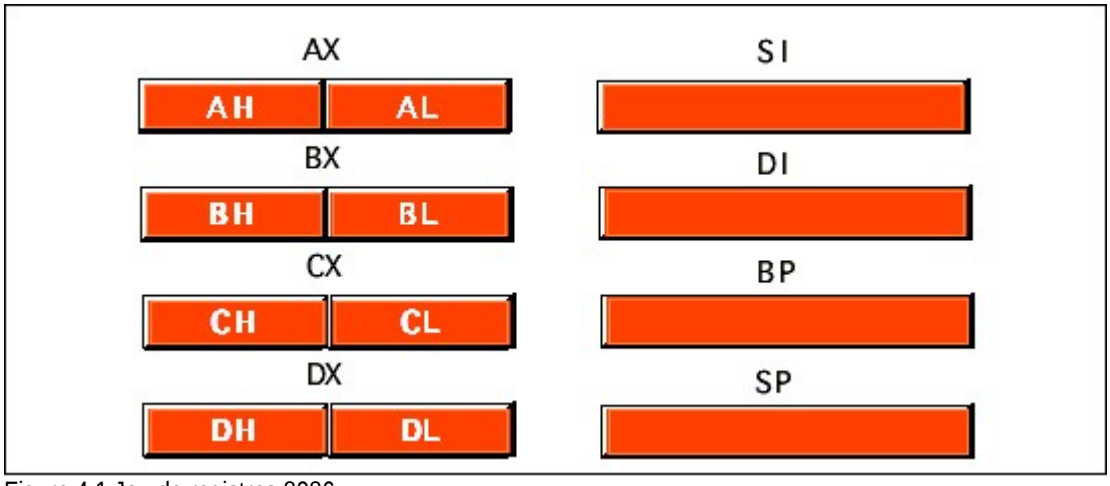

Figure 4.1 Jeu de registres 8086

Comme on vient de le dire, les registres de huit bits ne constituent pas un jeu indépendant de registres. Un changement dans al, modifiera la valeur de ax. Et la même chose s'applique pour ah. La valeur de al correspond

exactement au contenu des bits 0 à 7 de ax. Et la valeur de ah correspond aux bits 8 à 15 de ax. Par conséquent, toute modification apportée au registre al affectera le registre ax. De même, modifier ax signifie changer également le contenu des deux registres al et ah. Notez, cependant, que changer al n'affectera pas ah et vice-versa. Ces règles s'appliquent à tous les autres registres de huit bits.

Les registres si, di bp et sp sont seulement de 16 bits. Il n'y a pas de moyen d'accéder directement à leurs octets individuels.

4.1.2 Les registres de segment 8086

Les processeurs 8086 ont quatre registres de segment spécialisés : cs, ds, es et ss. Ils s'appellent respectivement *Code Segment* (segment de code), *Data Segment* (segment de données), Extra Segment (segment extra) et Stack Segment (segment de pile). La taille de ces registres est toujours de 16 bits. Ils ont pour fonction de sélectionner des blocs (segments) dans la mémoire principale. Un registre de segment, par exemple cs, pointe sur le début d'un segment de la mémoire.

Sur les processeurs 8086, la longueur d'un segment ne peut pas dépasser les 65536 octets. Cette maudite limitation de 64 Ko a irrité plus d'un programmeur. Plus tard, on sera en mesure de mesurer l'impact des problèmes qu'elle peut occasionner et de certaines solutions qu'on peut y apporter.

Le registre cs pointe sur le segment contenant les instructions machine en cours d'exécution. Notez que, malgré la limite des 64 Ko, les programmes 80x86 peuvent dépasser cette taille. Pour de tels programmes, on doit simplement multiplier les segments de code dans la mémoire. Puisqu'on peut changer la valeur du registre cs, on peut passer à un nouveau code de segment quand on veut exécuter le code qui s'y trouve.

Le registre de segment de données ds, pointe généralement sur les variables globales d'un programme. La limitation des 65536 octets persiste. Mais on peut toujours changer la valeur de ds pour accéder à des données additionnelles sur d'autres segments.

Le segment extra, es, est exactement ceci, un registre additionnel. Les programmes 8086 l'utilisent souvent pour gagner l'accès aux segments qui seraient difficiles ou même impossibles à modifier avec d'autres registres de segment.

Et, finalement, le registre ss pointe sur le segment contenant la pile, c'est-à-dire l'endroit où le processeur stocke d'importantes informations machine, des adresses de retour des sous-routines, des paramètres de procédure ou des variables locales. En général, vous ne devrez pas modifier ce registre, car le segment où il pointe contient des informations desquelles tout le système dépend.

Bien que stocker des données sur des registres de segment soit théoriquement possible, ce n'est jamais une bonne idée. Ces registres ont des fonctions vraiment spécialisées, ils pointent sur des blocs de mémoire accessibles. Toute tentative de les utiliser pour un autre but que celui pour lequel ils sont désignés peut donner lieu à des problèmes considérables, spécialement si vous programmez pour un meilleur CPU, comme le 80386.

4.1.3 Les registres 8086, dits « spécialisés »

Dans les CPU 8086 il y a deux registres particuliers : le registre ip - (instruction Pointer) ou pointeur d'instructions et le registre Flags (drapeaux). On ne peut pas accéder à ces registres de la même façon dont on accède aux autres. En général, seul le CPU peut les manipuler directement.

Le registre ip est l'équivalent du registre ip des processeurs x86 : il contient l'adresse de l'instruction suivante à exécuter. Il s'agit en fait d'un registre de 16 bits qui pointe sur le segment de code courant (il peut donc pointer jusqu'à 65536 adresses de mémoire différentes). On reparlera de ce registre en abordant les instructions de transfert de contrôle.

Le registre Flags est un registre à part, différent de tous les autres. Les autres registres peuvent contenir des valeurs de huit ou de seize bits. Mais le registre flags n'est rien d'autre qu'une collection éclectique de valeurs d'un bit aidant à déterminer l'état courant du processeur. Bien que flags ait une capacité de 16 bits, seuls neuf de ces bits sont effectivement utilisés. Et de ces neuf, il y en aura quatre que vous utiliserez tout le temps : *zero*, *carry* (retenue), *sign* et *overflow* (dépassement de capacité). Ces drapeaux sont les indicateurs de condition du 8086. Tout le contenu du registre es visible à la figure 4.2.

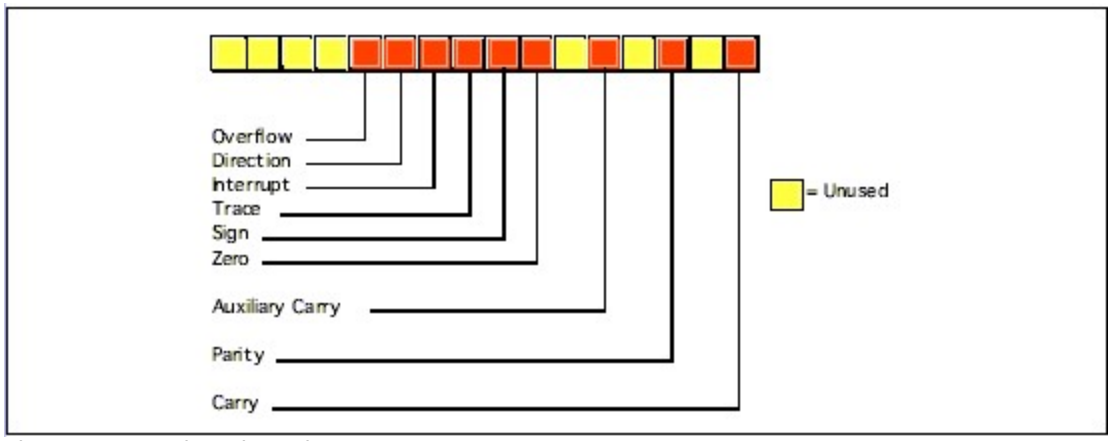

Figure 4.2 8086 : le registre Flags

4.1.4 Les registres du 80286

Le processeur 80286 ajoute une caractéristique de programmation majeure : le mode protégé. Mais nous ne couvrirons pas la version 80286 du mode protégé pour plusieurs raisons. D'abord, il a été médiocrement conçu et ensuite, il ne peut intéresser que des programmeurs écrivant leurs propres systèmes d'exploitation pour le 80286 ou des programmes système de bas niveau pour ces systèmes. Même en ayant besoin d'écrire des programmes pour des systèmes d'exploitation fonctionnant sous ce mode, comme UNIX ou OS/2, vous n'utiliserez pas le mode protégé du 80286. Il sera donc plus intéressant de se pencher sur les nouveaux registres et les nouveaux indicateurs du registre flags qui ont été ajoutés avec ce processeur.

Dans le registre flags du CPU 80286, il y a trois bits additionnels. Le niveau de privilège des E/S est un drapeau de deux bits (les bits 12 et 13). Il spécifie un des quatre niveaux de privilège nécessaires pour effectuer des opérations d'E/S. Ces deux bits contiennent généralement la valeur 00_b quand le processeur fonctionne en mode réel (le mode d'émulation du 8086). Le drapeau NT (*Nested Task* ou *tâche imbriquée* en français) contrôle l'opération d'une instruction de retour d'interruption (IRET). NT est normalement à zéro pour les programmes qui courent en mode réel.

À côté des bits additionnels dans le registre flags, le 80286 a ajouté aussi cinq registres utilisables par un système d'exploitation pour supporter la gestion de la mémoire et des processus multiples : msw (Machine Status Word), gdtr (Global Descriptor Table Register), idtr (Interrupt Descriptor Table Register) et tr (Task Register).

La seule utilisation réelle du mode protégé d'un programme pour le 80286 est l'accès à plus d'un Mo de RAM. Cependant, les processeurs 80286 sont maintenant obsolètes et les processeurs ultérieurs présentent de meilleurs moyens de dépasser cette limite. Par conséquent, les programmeurs ne se servent que rarement du mode protégé.

4.1.5 Les registres 80386/80486

En plus de tous les registres qui existaient dans le 80286 (et donc aussi le 8086), il a ajouté un nombre de nouveaux registres et étandu la définition des registres existants. Le 80486 n'a pas ajouté de nouveaux registres au jeu de base du 80386, mais il a défini certains bits dans certains registres qui n'étaient pas définis dans le 80386.

Le changement le plus significatif du point de vue du programmeur apporté par le 80386 a été l'introduction de registres de 32 bits. ax, bx, cx, dx, si, di, bp, sp, flags et ip ont tous été étendus à 32 bits. Ces nouvelles versions ont par conséquent pris les noms suivants : eax, ebx, ecx, edx, esi, edi, ebp, esp, eflags et eip, pour être différenciées des versions 16 bits de ces registres (encore disponibles sur le 80386). En plus des registres de 32 bits, le 80386 comprend aussi deux nouveaux registres de segment, fs et gs, qui permettent au programmeur d'accéder simultanément à six segments de mémoire différents sans avoir à recharger un registre de segment. Notez que tous les registres de segment sur le 80386 sont encore de 16 bits. Ils n'ont pas été étendus à 32 bits.

Le 80396 ne présente aucun changement dans le registre flag, mais ce registre a été étendu à 32 bits (en devenant le registre "eflags"), où les bits 16 et 17 ont été définis. Le bit 16 est le drapeau de reprise du débogage (*debug resume flag*, RF), utilisé avec l'ensemble des registres de débogage du 80386. Le bit 17 est un drapeau de mode virtuel (*virtual mode flag*, VM), qui détermine si le processeur est en train de fonctionner en mode *86 virtual* (qui simule un 8086), ou s'il travaille en mode protégé standard. Le 80486, de sa part, définit un dix-huitième bit, le drapeau de contrôle d'alignement (*algnment check flag*). Combiné avec le registre CR0 (*control register zero*), ce drapeau provoque une interruption du programme (*trap*) quand le processeur accède à des données non alignées (par exemple, un mot dans une adresse impaire ou un double-mot dans une adresse qui n'est pas multiple de 4).

Le 80386 a, en outre, ajouté quatre registres de contrôle : CR0-CR3. Ces registres constituent une extension du registre msw du 80286 (le 80386 émule encore msw pour des questions de compatibilité, mais les informations se trouvent réellement dans les registres CRx). Sur les CPU 80386 et 80486, ces registres ont des fonctions de gestion de mémoire paginée, d'opération d'activation/désactivation/opération du cache (80486 seulement), d'opération en mode protégé et plus.

Les processeurs 80386-406 ajoutent aussi huit registres de débogage. Un programme débogueur comme *CodeView* de Microsoft ou *Turbo Debugger* peut se servir de ces registres pour fixer des points d'arrêt pendant les opérations de localisation des erreurs dans un programme. Même si ces registres ne seront pas utilisés pour écrire des applications, vous trouverez souvent qu'utiliser des débogueurs permet de réduire de beaucoup le temps employé pour éradiquer les bogues dans un programme. Naturellement, un débogueur qui accède à ces registres fonctionnera correctement seulement sur des processeurs 80386 et ultérieurs.

Finalement, les processeurs 80386/486 ajoutent un ensemble de registres de test qui vérifient le bon fonctionnement du CPU au démarrage du système. Très probablement, Intel a placé ces registres dans la puce pour permettre des vérifications immédiates après la fabrication, mais les concepteurs des systèmes en profitent également pour effectuer des tests de démarrage.

En général, les programmeurs en assembleur n'ont pas à se préoccuper des nouveaux registres ajoutés aux 80386/486/Pentium, sauf s'il s'agit de registres qui les concernent directement comme les registres de 32 bits ou les registres de segment additionnels. Pour les programmeurs d'applications, le modèle de registres des processeurs 80386/486/Pentium ressemble à ce qui est montré à la figure 4.3.

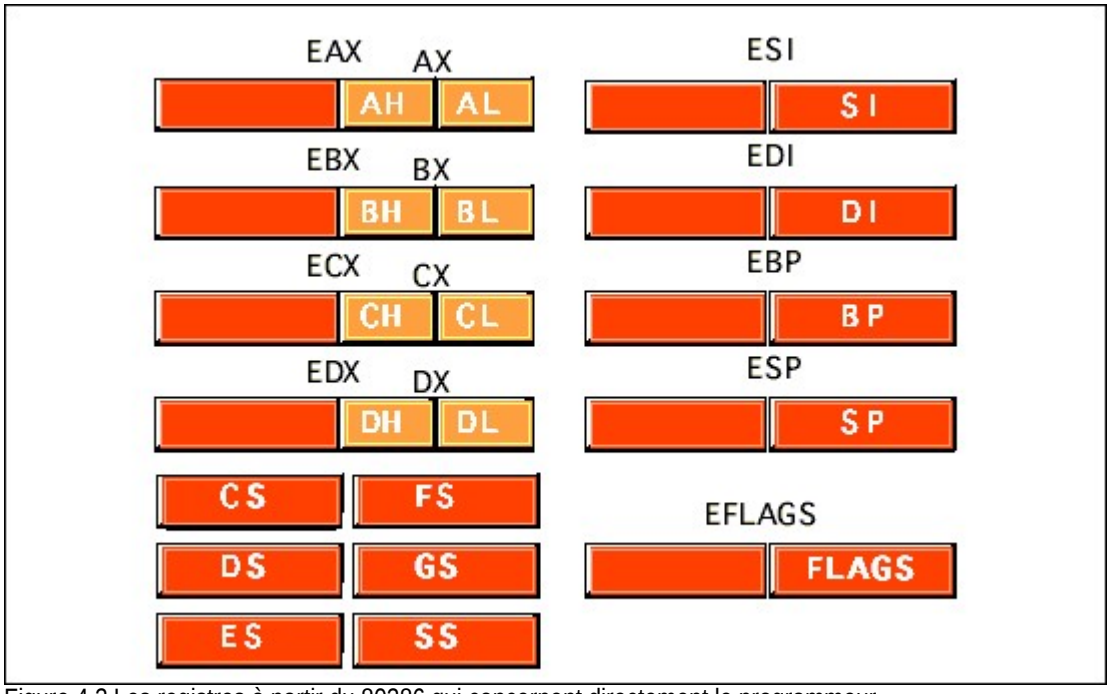

Figure 4.3 Les registres à partir du 80386 qui concernent directement le programmeur

4.2 80x86 : Organisation physique de la mémoire

Le troisième chapitre a traité l'organisation de base de l'architecture Von Neumann (VNA). Dans une machine VNA, le CPU communique avec la mémoire par le biais du bus. Les CPU 80x86 font référence à un élément mémoire en plaçant un nombre binaire dans le bus des adresses. Une autre façon de voir la mémoire est de la représenter par un tableau d'octets. Une structure de données Pascal qui correspondrait très grosso modo à la mémoire serait :

```
Memoire : Array[0...MaxRAM] of bytes;
```

La valeur dans le bus des adresses correspondrait à l'index fourni à ce tableau. Autrement dit, écrire une donnée dans la mémoire est équivalent à :

```
Memoire[adresse] := valeur_a_ecrire;
```

Et lire une donnée de la mémoire correspondrait à :

```
valeur_lue := Memoire[adresse];
```

Différents CPU 80x86 ont des bus d'adresses différents déterminant le nombre maximal d'éléments dans le tableau de la mémoire (voir paragraphe 3.1.1.2 "Le bus des adresses" du chapitre 3). Cependant, indépendamment du nombre des lignes d'adresses sur le bus, beaucoup d'ordinateurs ne disposent pas d'un octet de mémoire pour chaque emplacement adressable. Par exemple, les processeurs 80386 ont 32 lignes d'adresses permettant d'adresser jusqu'à 4 Go de RAM. Mais très peu d'ordinateurs 80386 disposent de 4 Go de mémoire. Normalement, les tailles actuelles de la RAM des processeurs 80x86 arrivent tout au plus à 256 Mo.

Le premier Mo de mémoire (plage 0-0FFFFFh) est spécial et il correspond à la totalité de l'espace d'adressage des microprocesseurs 8088, 8086, 80186 et 80188. Beaucoup de programmes DOS limitent leurs adresses de code et de données à des emplacements se trouvant dans cette plage. Ces adresses sont nommées *adresses réelles*, d'après le *mode réel* 80x86.

4.3 Segments 80x86 (programmation 16 bits) :

Au sein de la famille 80x86, on ne peut pas parler d'adressage de mémoire sans d'abord mentionner la segmentation. Parmi d'autres choses, cette dernière fournit aussi un mécanisme puissant de gestion de la mémoire permettant aux programmeurs de diviser leurs programmes en modules qui fonctionnent indépendamment l'un de l'autre. Les segments offrent aussi une moyen aisé d'implémenter des programmes orientés objet. Ils permettent notamment à deux processus de se partager facilement leurs données. D'autre part, si vous demandez à dix programmeurs qu'est-ce qu'ils pensent de la segmentation, au moins neuf vous diront qu'elle est une chose terrible. Pourquoi une telle réponse ?

Eh bien, il se trouve que la segmentation présente aussi une autre caractéristique élégante : elle permet d'élargir l'adressabilité des processeurs. Dans le cas du 8086, la segmentation permet aux concepteurs Intel de porter le maximum de mémoire adressable de 64 Ko à 1 Mo. Ce qui paraît bon. Mais alors pourquoi tout le monde se plaint ? Une petite digression historique aidera peut-être à comprendre que, somme toute, il y a quelque chose qui ne va pas.

En 1976, quand Intel commença à concevoir son processeur 8086, la mémoire était vraiment coûteuse. Les ordinateurs personnels, tels qu'ils étaient à l'époque, avaient ordinairement quatre mille octets de mémoire. Même quand IBM introduisit son PC, cinq années plus tard, 64 Ko étaient une quantité appréciable et 1 Mo était considéré comme une taille légendaire. Les concepteurs Intel trouvaient que 64 Ko de mémoire vive c'était déjà beaucoup et le resterait, au moins pendant toute la durée commerciale des processeurs 8086. La seule erreur d'Intel a été de sous-estimer complètement la durée de vie du modèle 8086. Ils ont cru qu'il n'allait pas dépasser les cinq ans, comme cela avait été le cas avec son prédécesseur, le 8080. Ils avaient des plans pour beaucoup d'autres processeurs à cette époque et "86" n'était destiné à être le suffixe d'aucun. Intel imagina que tout était en règle et qu'un Mo serait plus que suffisant, au moins jusqu'au jour où ils sortaient quelque chose de mieux[3].

Malheureusement, Intel n'avait pas prévu le PC IBM et le nombre massif de programmes qui allaient être écrits pour lui. A partir de 1983, il fut évident qu'Intel ne pouvait plus abandonner l'architecture 80x86. Ils étaient forcés

[3]A cette époque, Intel considérait le processeur iapx432 comme son prochain gros produit. Personne ne pouvait prévoir qu'il allait périr, au contraire, d'une lente et horrible mort.

de la maintenir, mais en attendant, les programmeurs butaient sur la limite de 1Mo imposée par le 8086. Donc, Intel donna au monde le 80286, qui pouvait adresser jusqu'à 16 Mo de mémoire. Sûrement plus que suffisant. Le seul ennui était que tous les merveilleux programmes écrits pour le 8086 ne pouvaient profiter de la nouvelle extension et aller au delà du méga-octet.

Il s'avère que le maximum de mémoire adressable n'était pas la cause principale des plaintes du public. Le problème véritable était que le 8086 était un processeur de 16 bits, avec des registres et des adresses de 16 bits. Ce qui le limitait à adresser des blocs de mémoire de 64 Ko. L'usage intelligent qu'Intel avait fait de la segmentation permit d'élargir cette limite à 1 Mo, mais adresser plus de 64 Ko à la fois demandait de l'effort. Et adresser plus de 256 Ko à la fois demandait *beaucoup* d'effort.

Contrairement à ce que vous avez appris, la segmentation n'est pas une mauvaise chose. En effet, il s'agit d'un schéma de gestion de mémoire réellement valable. Ce qui est mauvais, c'est l'implémentation qu'Intel a fait en 1976 et qui est encore utilisée aujourd'hui. Vous ne pouvez pas accuser Intel pour cela. Ils ont réglé le problème dans les années 80 avec la sortie du 80386. Le véritable coupable est MS-DOS qui oblige encore les programmes à utiliser le style de segmentation de 1976. Heureusement, UNIX et les nouveaux systèmes d'exploitation comme LINUX, Windows 9x, Windows NT et OS/2 ne souffrent pas des mêmes problèmes que MS-DOS. De plus, les utilisateurs passent de plus en plus vers ces nouveaux systèmes d'exploitation, de sorte que les programmeurs peuvent profiter des nouveaux avantages offerts par la famille 80x86.

Après cette leçon d'histoire, il serait probablement une bonne idée d'illustrer ce qu'est la segmentation. Considérez l'apparence normale de la mémoire : elle ressemble à un tableau linéaire d'octets. Un index unique (adresse) permet de sélectionner un octet particulier du tableau. Appelons cet adressage *linéaire* ou *plat*. L'adressage segmenté utilise deux éléments pour spécifier un emplacement de mémoire : une valeur de segment et une valeur d'offset à l'intérieur de ce segment. Idéalement, le segment et l'offset sont des valeurs indépendantes entre elles. La meilleure façon de décrire l'adressage segmenté est de l'illustrer via un tableau à deux dimensions, où le segment constitue un index et l'offset en constitue un autre (voir figure 4.4).

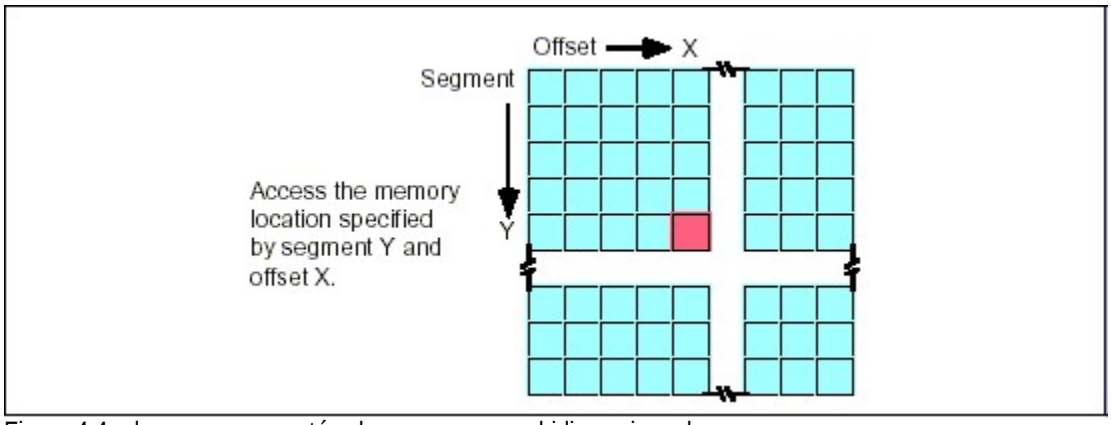

Figure 4.4 adressage segmenté selon un processus bidimensionnel

Maintenant, tout ceci vous étonnera sans doute. « Pourquoi rendre le procédé plus complexe ? ». L'adressage linéaire a l'air de bien fonctionner, pourquoi donc se tracasser avec ces modes bidimensionnels ? Eh bien, considérons la façon dont on écrit un programme. Si on écrit, par exemple, une routine SIN(X) et qu'on a besoin de certaines variables temporaires, on ne voudra probablement pas utiliser des variables globales, mais des variables locales connues seulement à l'intérieur de la fonction. Dans un sens large, c'est l'une des caractéristiques que la segmentation offre, la possibilité d'attacher des blocs de variables (des segments) à une séquence particulière de code. On pourrait, par exemple, avoir un segment contenant les variables locales de SIN, un segment pour SQRT (racine carrée), un segment pour DRAWWindow, etc. Puisque les variables de SIN apparaissent dans le segment de SIN, il est moins probable que cette routine affecte les variables reliées à la routine SQRT. En fait, sur les 80286 et ultérieurs, le CPU peut empêcher qu'une routine modifie accidentellement les variables d'un segment différent.

Une adresse segmentée complète contient un composant de segment et un composant d'offset. Ce livre utilisera la notation *segment : offset*. Sur le 8086 jusqu'au 80286, ces deux valeurs sont des constantes de 16 bits. Sur le 80386 et ultérieurs, l'offset peut être une constante de 16 ou de 32 bits.

La taille de l'offset détermine la taille maximale d'un segment. Sur le 8086, avec des offsets de 16 bits, un segment ne peut pas dépasser les 64 Ko ; il peut être plus court (ce qui est le cas pour la plupart des segments), mais jamais plus long. Alors que les 32 bits d'offset permis sur le 80386 et ultérieurs permettent des tailles maximales de 4 Go.

La partie segment est toujours de 16 bits sur tous les processeurs. Ce qui permet à chaque programme d'avoir jusqu'à 65536 segments différents. La plupart des programmes n'utilisent pas plus de 16 segments, donc ce n'est pas une limitation alarmante dans la pratique.

Naturellement, malgré le fait que les processeurs 80x86 utilisent l'adressage segmenté, la véritable mémoire physique connectée au CPU est encore un tableau linéaire d'octets. Il y a une fonction permettant de convertir une adresse segmentée en une adresse physique. Le processeur lui ajoute ensuite l'offset et obtient l'adresse effective de la donnée. En général, les programmes de ce livre se réfèreront aux adresses selon la convention des adresses segmentées ou *logiques*. Alors que l'adresse linéaire apparaissant dans le bus des adresses sera référée comme *adresse physique* (voir figure 4.5).

Sur les 8086, 8088, 80186, 80286 et d'autres processeurs fonctionnant en mode réel, la fonction convertissant une adresse logique en une adresse physique est très simple. Le CPU multiplie la valeur du segment par 16 (10h) et lui ajoute la partie de l'offset. Par exemple, considérez l'adresse logique[4] : 1000:1F00. Pour convertir celle-ci en adresse physique, tout ce qu'on doit faire est de multiplier la valeur du segment (1000h) par seize. Et multiplier par la base est très simple : il suffit d'ajouter un zéro à la fin du nombre. L'ajouter à 1000h produit 10000h. Et après avoir additionné 1F00h à ce résultat, on obtient 11F00h. Par conséquent, 11F00h est l'adresse physique de 1000:1F00 (voir figure 4.6).

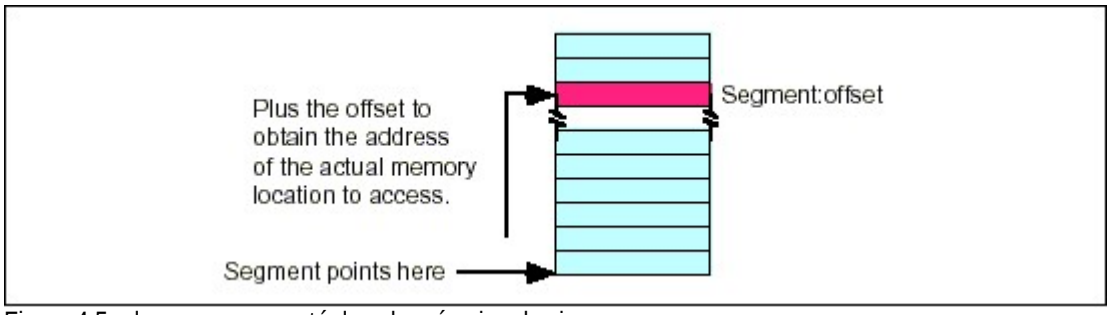

Figure 4.5 adressage segmenté dans la mémoire physique

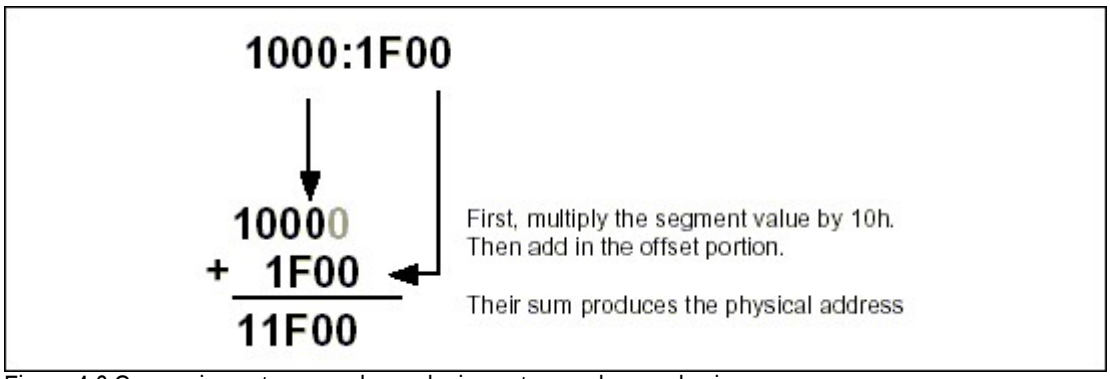

Figure 4.6 Conversion entre une adresse logique et une adresse physique

Attention : une erreur vraiment courante dans ce genre d'opérations est oublier que le travail se fait en hexadécimal. C'est surprenant de voir combien de gens, additionnant 9 + 1 trouvent 10h au lieu de 0Ah, qui est la réponse correcte.

Quand Intel a conçu le 80286 et les processeurs ultérieurs, il n'a pas étendu l'adressage en ajoutant plus de bits aux registres de segment, mais ils ont changé la fonction que le CPU utilise pour convertir une adresse

[4]Toutes les adresses segmentées, dans ce livre, seront en base hexadécimale. Puisque ce sera la base utilisée pour désigner toutes les adresses, on n'aura pas besoin du suffixe h.

logique en une adresse physique. Par conséquent, si vous écrivez du code qui dépend de « multiplie par seize et ajoute l'offset », votre programme fonctionnera seulement sur les processeurs qui fonctionnent en mode réel et vous serez également limités à un Mo de mémoire[5].

Sur les processeurs 80286 et ultérieurs Intel a introduit les *segments en mode protégé*. Entre les autres changements, il a complètement réécrit l'algorithme de transposition des segments dans l'espace d'adressage linéaire. Au lieu d'utiliser une fonction (comme celle de multiplier la valeur du segment par 10h), pour calculer les adresses physiques, les nouveaux processeurs font usage d'une *table de correspondance (lookup table)*. En mode protégé, la valeur du segment est utilisée comme un index de tableau. Son contenu est capable de donner (entre autres) l'adresse de début du segment. Le CPU additionne alors cette valeur à l'offset et il obtient l'adresse physique (voir figure 4.7).

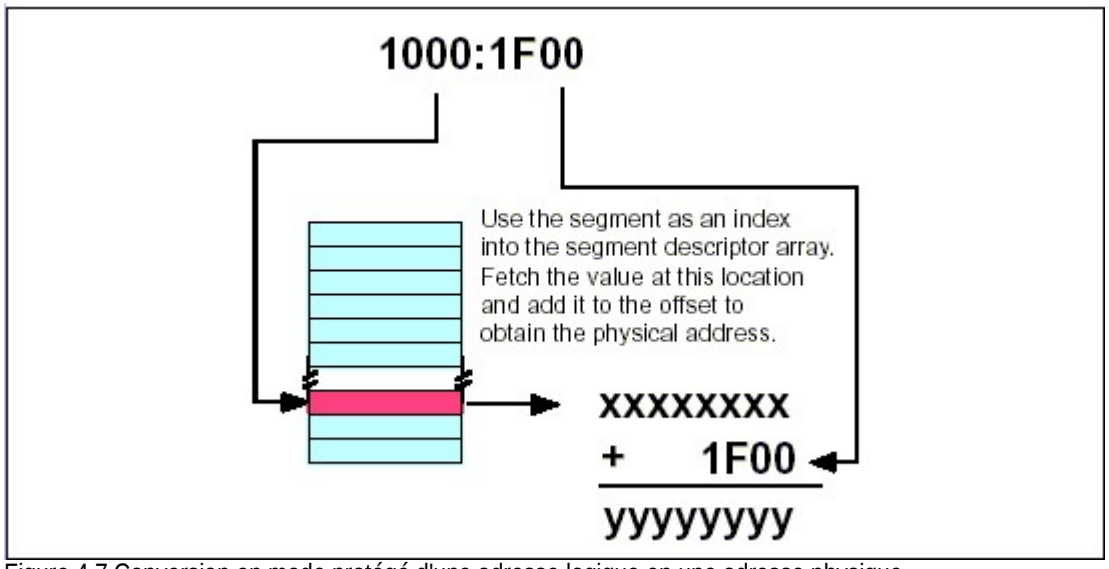

Figure 4.7 Conversion en mode protégé d'une adresse logique en une adresse physique

Notez que vos propres applications ne peuvent pas modifier la table de description des segments (la table de correspondance). Ce sont les systèmes d'exploitation basés sur le mode protégé (UNIX, Linux, Windows, OS/2, etc.), qui effectuent cette opération.

Les meilleurs programmes ne présument jamais qu'un segment est situé à un endroit particulier de la mémoire. Il faut laisser au système d'exploitation la tâche de placer un programme dans la mémoire ; ce n'est pas bon de générer toute adresse de segment par soi-même.

4.4 Adresses normalisées

Quand on travaille en mode réel, il y a un problème intéressant qui surgit. Vous pouvez faire référence à un objet en mémoire en utilisant plusieurs adresses *différentes*. Considérez l'adresse de l'exemple précédent, 1000:1F00. Plusieurs adresses logiques peuvent représenter cette adresse. Par exemple 11F0:0, 1100:F00 ou même 1080:1700 correspondent toutes à l'adresse physique 11F00h. En travaillant avec certains types de données, spécialement si on compare des pointeurs, il est pratique que les adresses segmentées pointent sur des objets différents en mémoire quand leurs représentations en bits sont différentes. Ce n'est sans doute pas toujours le cas en mode réel.

Heureusement, il existe un moyen facile d'éviter ce problème. Si vous avez besoin de comparer deux adresses vous pouvez utiliser des adresses *normalisées*. Elles ont un format spécial, de sorte qu'elles sont toutes uniques. A moins que deux valeurs segmentées normalisées ne soient absolument les mêmes, elles ne pointent pas sur le même objet en mémoire.

[5]Actuellement, vous pouvez aussi travailler en mode V86 (virtual 86) sur les processeurs 80386 et ultérieurs, mais vous serez encore limités à un Mo de mémoire adressable.

Il y a plusieurs manières (16 en fait) de créer des adresses normalisées. Par convention, la plupart des programmeurs (ainsi que les langages de haut niveau), définissent une adresse normalisée comme suit :

- La portion "segment" de l'adresse peut être toute valeur de 16 bits.
- La portion "offset" doit être une valeur de la plage 0..0Fh.

Convertir une adresse normalisée en adresse physique est très facile. La seule chose qu'il faut faire est d'ajouter, comme chiffre additionnel, la portion "offset" de l'adresse normalisée à la valeur du segment. Par exemple, la forme normalisée de 1000:1F00 est 11F0:0. On peut obtenir l'adresse physique, en ajoutant l'offset (qui vaut 0 dans ce cas) à la fin de 11F0, pour obtenir 11F00.

Convertir une adresse segmentée arbitraire en adresse normalisée est également très facile. En premier lieu, faire la conversion en adresse physique, en multipliant l'adresse logique du segment par 16 et en lui ajouter l'offset. Ensuite, placer un " : " entre les deux derniers chiffres du résultat :

1000:1F00 ⇒ 11F00 ⇒ 11F0:0

Notez que cette discussion s'applique seulement aux processeurs 80x86 qui fonctionnent en mode réel. En mode protégé, il n'y a pas de correspondance directe entre les adresses segmentées et les adresses physiques, donc cette technique ne pourrait pas marcher. Cependant, ce texte s'occupe surtout de programmes fonctionnant en mode réel, donc, on verra encore des pointeurs normalisés.

4.5 Les registres de segment 80x86

Quand Intel conçut le 8086 en 1976, la mémoire était un luxe précieux. Le jeu d'instructions a été conçu de sorte à assurer l'emploi du moins possible d'octets pour chaque instruction. Ceci garantissait aussi des programmes petits, puisque les processeurs Intel utilisaient moins de mémoire. De plus, ces systèmes étaient moins coûteux à produire. Sans doute, le coût de la mémoire est tombé jusqu'au point où ceci n'est plus une préoccupation[6]. Une des choses qu'Intel voulait éviter, c'était l'ajout d'adresses de 32 bits (segment:offset) à la fin des instructions qui faisaient référence à la mémoire. Il pouvait réduire ceci à 16 bits (offset seul) en faisant certaines hypothèses sur quels segments en mémoire pouvaient être accédés par une instruction.

Les processeurs 8086 - allant jusqu'à 80286 – ont quatre registres de segment : cs, ds, ss et es. Les processeurs à partir du 80386 ont aussi deux registres additionnels : fs et gs. cs (*code segment*) pointe sur le segment qui contient le code couramment exécuté. Le CPU charge toujours les instructions de l'adresse donnée par cs:ip. Par défaut, il s'attend à recevoir la plupart des variables du segment des données (ds). D'autres variables et d'autres opérations se produisent dans le segment de pile. Quand on accède à des données qui se trouvent dans ces secteurs spécifiques, la valeur du segment n'est pas nécessaire. Pous accéder à une donnée dans l'un des segments extra (es, fs ou gs), seulement un octet est nécessaire : celui permettant de choisir le registre de segment approprié. Seules quelques instructions de transfert de contrôle requièrent des adresses complètes de 32 bits.

Maintenant, ceci pourrait sembler limité. Après tout, avec seulement quatre registres de segment sur le 8086 on peut adresser un maximum de 256 kilooctets (64Ko par segment) et non tout le méga-octet promis. Cependant, on peut toujours changer de registre de segment sous le contrôle du programme, donc on peut accéder à tout octet en changeant la valeur du registre de segment.

Naturellement, pour changer la valeur d'un des registres de segment 80x86 il faut au moins deux instructions. Celles-ci consomment de la mémoire et demandent du temps pour s'exécuter. Donc, économiser deux octets par accès de mémoire ne serait pas payant si vous accédez tout le temps à des données en différents segments. Heureusement, presque tous les accès mémoire consécutifs ont lieu dans le même segment. Donc, charger des registres de segment n'est pas quelque chose que vous ferez souvent.

4.6 Les modes d'adressage 80x86

Tout comme les processeurs x86 du chapitre précédent, les processeurs 80x86 permettent d'accéder à la mémoire de différentes façons. En particulier, les modes d'adressage de ce processeur permettent un accès

[6]Néanmoins, les petits programmes sont encore importants. Plus un programme est petit, plus il exécutera rapidement, parce que le CPU aura à charger moins d'octets de la mémoire et les instructions ne prendront pas beaucoup de place dans le cache.

flexible et facile pour des variables, des tableaux, des enregistrements, des pointeurs et d'autres types de données complexes. Maîtriser les modes d'adressage 80x86 est le premier pas à faire vers la maîtrise de l'assembleur.

Quand Intel a conçu le processeur 8086, il l'a muni d'un ensemble flexible mais limité de modes d'adressage. Avec le processeur 80386, cet ensemble s'est élargi, mais tous les anciens modes ont été conservés. Ces ajouts donc, n'ont constitué qu'un supplément. Logiquement, on ne peut pas tirer profit de ces nouveautés si on programme pour un processeur 80286 ou antérieur. Étant donné que parfois certains programmeurs ont encore besoin d'écrire leurs programmes sur des anciens microprocesseurs[7], c'est important d'exposer ces deux ensembles de modes séparément, afin d'éviter toute confusion.

4.6.1 Les modes d'adressage des registres 8086

La plupart des instructions 8086 peuvent opérer sur le jeu des registres généraux. En spécifiant le nom du registre comme opérande de l'instruction, on peut accéder au contenu de ce registre. Considérez l'instruction mov :

```
mov    destination, source
```

Cette instruction copie le contenu de l'opérande *source* et le place dans l'opérande *destination*. Les registres de 8 bits ou de 16 bits sont certainement des opérandes valides pour cette instruction. La seule condition est que les deux opérandes soient de la même taille. Voici certaines instructions mov :

```
mov     ax, bx          ;place la valeur de bx dans ax
mov     dl, al          ;place la valeur de al dans dl
mov     si, dx          ;place la valeur de dx dans si
mov     sp, bp          ;place la valeur de bp dans sp
mov     dh, cl          ;place la valeur de cl dans dh
mov     ax, ax          ;oui ceci est permis !
```

Souvenez-vous que les registres sont le meilleur endroit pour garder des variables fréquemment utilisées. Comme vous le verrez un peu plus loin, les instructions faisant usage des registres sont plus courtes et plus rapides que celles qui accèdent à la mémoire. Au long de ce chapitre, vous verrez les opérandes abrégées *reg* et *r/m* (register/memory) utilisées à tous les endroits où l'on peut faire usage des registres généraux. En plus de ces derniers, beaucoup d'instructions 8086 (en incluant mov) permettent de spécifier un des registres de segment en tant qu'opérande. Il ya deux restrictions dans l'usage de ces registres avec l'instruction mov. Avant tout, vous ne pouvez pas spécifier cs comme opérande de destination et en second lieu, seulement une des deux opérandes de l'instruction mov peut être un registre de segment. Vous ne pouvez pas, par le biais d'une seule instruction mov, transporter des données d'un registre de segment à un autre. Pour copier la valeur de cs dans ds, il faut se servir de séquences comme celle-ci :

```
mov    ax, cs
mov    ds, ax
```

Il en faudrait jamais utiliser des registres de segment en tant que registres de données pour y stocker des données arbitraires. Ces registres ne devraient contenir que des adresses de segment. Mais on en reparlera plus tard. Dans ce livre, nous allons faire usage de l'abréviation *sreg* partout où les opérandes pour les registres de segment sont permises (ou requises).

4.6.2 Les modes d'adressages de la mémoire 8086

Le 8086 offre dix-sept manières d'accéder à la mémoire. Ceci peut paraître encombrant au début[8], mais, heureusement la plupart de ces modes sont de simples variantes d'autres modes, donc ils sont faciles à apprendre. Et, c'est important de les apprendre ! La clé de la bonne programmation en assembleur est l'usage correct des modes d'adressage de la mémoire.

[7]Les ordinateurs modernes n'utilisent que rarement des processeurs antérieurs au 80386, mais les systèmes intégrés en font encore usage.
[8]Attendez à voir le 80386 !

4.6.2.1 Le mode d'adressage *déplacement seul*

Le mode d'adressage le plus commun, mais aussi celui qui est le plus facile à apprendre est le mode d'adressage qu'on nomme *déplacement seul (displacement-only*) ou *mode d'adressage direct*. Ce mode consiste en une constante de 16 bits qui spécifie l'adresse de l'emplacement cible. L'instruction mov al, ds:[8088h] charge le registre al avec une copie de l'octet se trouvant à l'adresse 8088h[9] (voir figure 4.8).

La syntaxe de MASM pour les modes d'adressage 8086

Microsoft Assembler utilise plusieurs variantes pour indiquer les modes d'adressage indexé, basé/indexé, et *déplacement + basé/indexé*. Dans ce livre, vous verrez toutes ces formes utilisées interchangeablement. La liste qui suit, illustre certaines des combinaisons possibles dans les divers modes d'adressage 80x86 :

```
disp[bx], [bx][disp], [bx+disp], [disp][bx] et [disp+bx][10]

[bx][si], [bx+si], [si][bx] et [si+bx]

disp[bx][si], disp[bx+si], [disp+bx+si], [disp+bx][si], disp[si][bx],
[disp+si][bx], [disp+si+bx], [si+disp+bx], [bx+disp+si], etc.
```

Masm traite le symbole "[" de la même façon qu'il traite l'opérateur "+ ". Cet opérateur est commutatif, tout comme l'opérateur "+ ". Naturellement, cette discussion s'applique à tous les modes d'adressage 8086 et non seulement à ceux qui impliquent BX et SI. Vous pouvez utiliser tous les registres permis dans les combinaisons montrées ci-dessus.

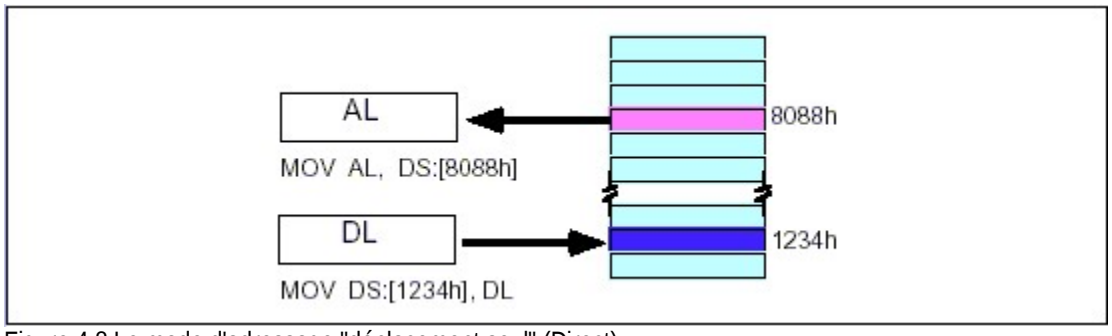

Figure 4.8 Le mode d'adressage "déplacement seul" (Direct)

Le déplacement seul est parfait pour accéder à des variables uniques. Certes, vous préférez sans doute utiliser des noms, comme "i" ou "j", au lieu de "DS:[1234h]" ou "DS:[8088h]". Vous verrez bientôt que'il est possible de le faire.

Intel donne ce nom à ce mode d'adressage parce qu'une constante de 16 bits (le déplacement) suit l'opcode de l'instruction mov en mémoire. Sous cet aspect, c'est très semblable au mode d'adressage direct qu'on a vu en étudiant les processeurs x86. Il y a cependant, certaines différences mineures. Avant tout, un "déplacement" est exactement ce qu'il décrit, une distance par rapport un point donné. Sur les processeurs x86, une adresse directe pouvait être considérée un déplacement à partir de l'adresse zéro. Sur les processeurs 80x86, par contre, un déplacement est un offset par rapport au début d'un segment (le segment des données, dans cet exemple). Ne vous inquiétez pas si cela n'a pas encore beaucoup de sens à vos yeux. Vous aurez l'opportunité d'étudier le coeur de ce sujet un peu plus loin dans ce chapitre. Pour l'instant, vous pouvez considérer le mode d'adressage *déplacement seul* comme un mode d'adressage direct. Les exemples de ce chapitre auront pour objet d'accéder à des octets en mémoire. N'oubliez pas cependant, que dans les processeurs 8086 vous pouvez accéder aussi à des mots[11] (voir figure 4.9).

[9]La fonction du préfixe "DS:" vous sera claire un peu plus tard.
[10]*disp* est l'abbréviation de *déplacement.* On utilise la notation anglaise parce que c'est plus courante et, de toute façon, on code en anglais, n.d.t.
[11]Et à des doubles-mots, à partir du processeur 80386.

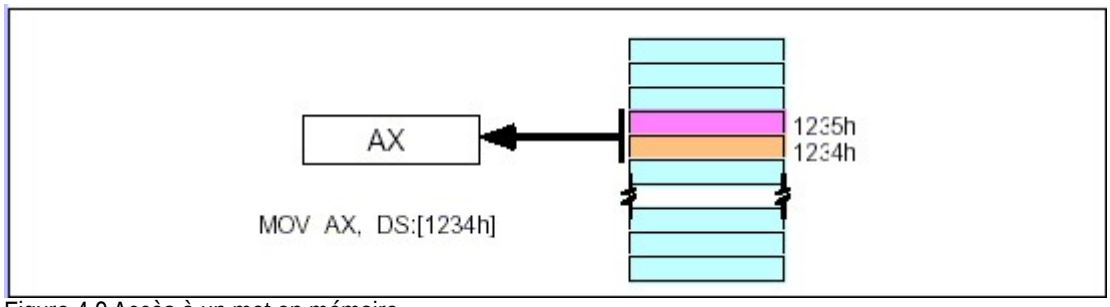

Figure 4.9 Accès à un mot en mémoire.

Par défaut, toutes les valeurs des déplacements seuls fournissent des offsets dans le segment des données. Si vous voulez produire un offset dans un autre segment, vous devez utiliser un préfixe de surcharge de segment (*segment override prefix*), avant votre adresse. Par exemple, pour accéder à l'emplacement 1234h par rapport à es, on utiliserait une instruction de la forme mov ax, es:[1234h]. De même, pour accéder à l'adresse 1234h relative au segment cs, il faudra : mov ax, cs:[1234h]. Alors que le préfixe "ds:" dans l'exemple précédent *ne constituait pas* une surcharge de segment, parce que le CPU utilise ds comme segment par défaut pour stocker des données. Certains exemples spécifiques requièrent ds: à cause de certaines limitations syntactiques de MASM.

4.6.2.2 Modes d'adressage indirect par les registres

Les CPUs 80x86 permettent d'accéder à la mémoire de façon indirecte par le biais d'un registre. Sur le 8086, il y a quatre formes de ce mode, comme montré ci-dessous :

```
mov    al, [bx]
mov    al, [bp]
mov    al, [si]
mov    al, [di]
```

Comme pour le mode d'adressage [bx] des processeurs x86, ces quatre opérations font référence à l'octet de l'offset trouvé dans les registres bx, bp, si ou di, respectivement. Les versions [bx], [si] et [di] utilisent ds par défaut. Alors que la version [bp] utilise ss par défaut.

Vous pouvez vous servir de la technique de surcharge de segments si vous voulez accéder à des données qui se trouvent à des segments différents. Les instructions suivantes montrent ceci :

```
mov    al, cs:[bx]
mov    al, ds:[bp]
mov    al, ss:[si]
mov    al, es:[di]
```

Intel se réfère à [bx] et [bp] comme à des modes d'adressage *de base* et à bx et bp comme à des registres de base (en effet bp est un sigle qui signifie *base pointer*). Alors que [si] et [di] sont considérés comme des modes d'adressage *indexés* (en effet, les sigles si et di signifient respectivement *source index* et *destination index*). Néanmoins, ces quatre modes d'adressage sont fonctionnellement équivalents. Pour une question de cohérence, ce livre appellera ces formes *modes d'adressages indirects par les registres*.

Note : les modes [si] et [di] fonctionnent exactement de la même façon, on doit juste remplacer bx par si et di dans la figure 4.10.

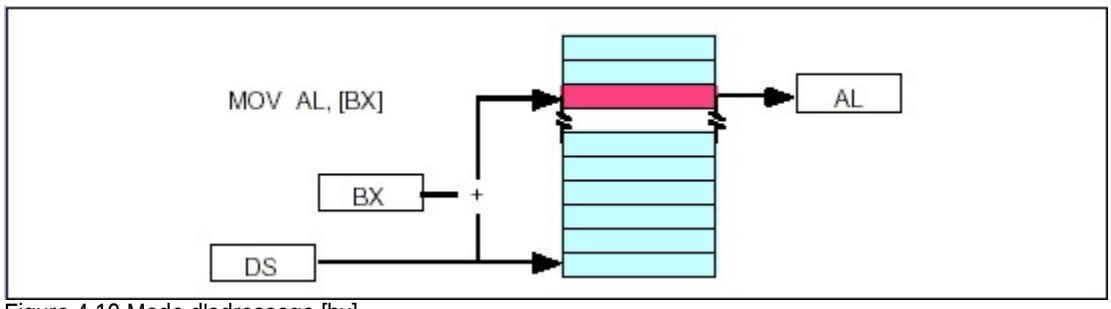

Figure 4.10 Mode d'adressage [bx]

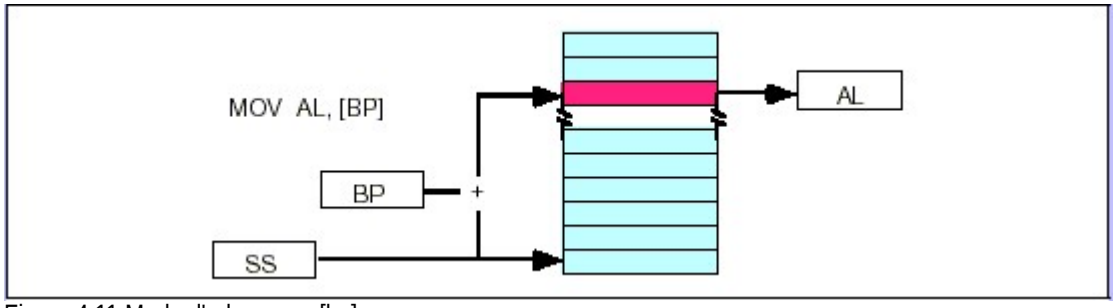

Figure 4.11 Mode d'adressage [bp]

4.6.2.3 Modes d'adressage Indexés

Les modes d'adressage indexé utilisent la syntaxe suivante :

```
mov    al, disp[bx]
mov    al, disp[bp]
mov    al, disp[si]
mov    al, disp[di]
```

Si bx contient 1000h, l'instruction mov cl, 20h[bx] chargera cl avec l'emplacement de mémoire ds:1020h. De même, si bp contient 2020h,mov dh, 1000h[bp] chargera dh avec l'emplacement ss:3020.

Les offsets générés de cette façon sont la somme entre la constante (disp) et le registre spécifié. Les modes d'adressage qui concernent bx, si et di utilisent tous le segment des données, alors que le mode concernant bp (donc disp[bp]) utilise le segment de pile par défaut. Comme dans le cas des modes d'adressage indirects, on peut utiliser les préfixes de surcharge de segment pour spécifier des segments de notre choix :

```
mov    al, ss:disp[bx]
mov    al, es:disp[bp]
mov    al, cs:disp[si]
mov    al, ss:disp[di]
```

Vous pouvez remplacer les modes d'adressage [si] ou [di] dans la figure 4.12 par les modes d'adressage [si+disp] et [di+disp].

Notez qu'Intel fait encore référence à ces modes d'adressage en termes d'adressages basés et indexés. La littérature Intel ne fait pas de distinction entre les modes avec ou sans la constante. Si vous regardez juste le fonctionnement du matériel, la définition est raisonnable. Cependant, d'un point de vue purement de programmation, cette distinction peut être parfois déterminante. C'est pourquoi, ce livre emploie différents termes pour les décrire. Malheureusement, le monde 80x86 n'est pas bien en accord sur le choix d'une terminologie appropriée.

Adressage basé versus adressage indexé

Il y a une différence subtile entre les modes d'adressage basé et indexé. Les deux constituent un déplacement additionné avec un registre. La différence principale est la taille relative aux valeurs du registre et du déplacement. Dans le mode indexé, la constante donne normalement l'adresse de la structure de données

spécifique et le registre représente l'offset par rapport à cette adresse. Dans le mode basé, le registre contient l'adresse de la structure de données et la constante constitue un index par rapport à ce point.

Puisque l'addition est commutative, les deux techniques sont essentiellement équivalentes. Cependant, puisqu'Intel supporte des déplacements d'un octet et de deux octets (voir "L'instruction MOV 80x86" au paragraphe 4.7), c'est plus logique pour eux d'appeler tout ceci mode d'adressage basé. Toutefois, vous vous servirez beaucoup plus souvent du mode indexé, d'où le changement de nom.

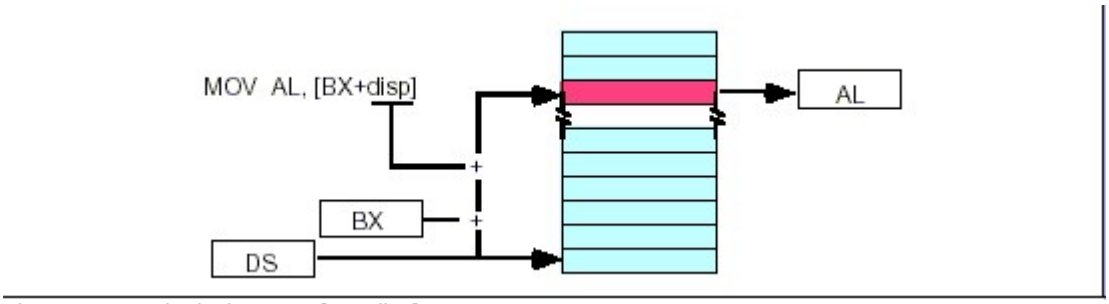

Figure 4.12 Mode d'adressage [BX+disp]

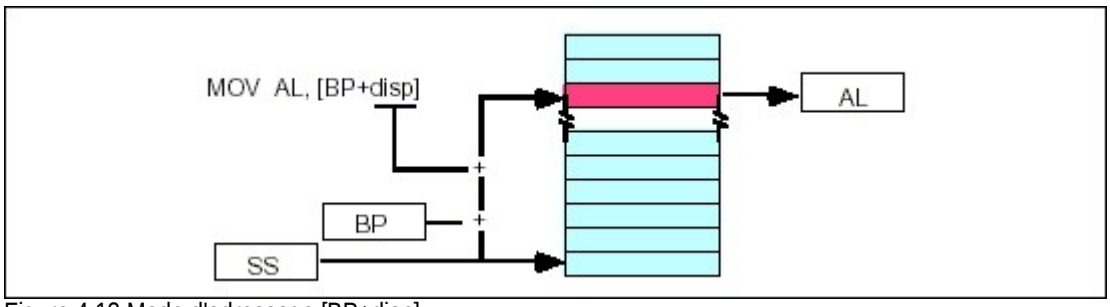

Figure 4.13 Mode d'adressage [BP+disp]

4.6.2.4 Modes d'adressage basé/indexé

Les modes d'adressage basé/indexé sont simplement une combinaison des modes indirects. Ils forment l'offset en additionnant un registre de base (bx ou bp) et un registre d'index (si ou di). Les formes permises pour ces modes sont :

```
mov    al, [bx][si]
mov    al, [bx][di]
mov    al, [bp][si]
mov    al, [bp][di]
```

Supposez que bx contient 1000h et que le registre si contient 880h. Alors, l'instruction :

```
mov al, [bx][si]
```

charge le registre al avec l'emplacement DS:1880h. De même, si bp contient 1598h et di contient 1004, mov ax, [bp+di] chargera le registre de 16 bits ax avec les emplacements SS:259C et SS:259D.

Les modes d'adressage qui ne concernent pas bp utilisent le segment des données par défaut. Ceux qui ont bp comme opérande utiliseront le segment de pile par défaut.

Dans la figure 4.14, pour obtenir le mode d'adressage [bx+di] on substitue di à si. Et on peut faire de même pour le mode d'adressage [bp+di] (figure 4.15).

4.6.2.5 Modes d'adressage basé/indexé plus déplacement

Ces modes constituent une légère modification du mode basé/indexé. La seule différence est l'addition d'une constante de huit ou seize bits. En voici certains exemples :

```
mov     al,    disp[bx][si]
mov     al,    disp[bx+di]
mov     al,    [bp+si+disp]
mov     al,    [bp][di][disp]
```

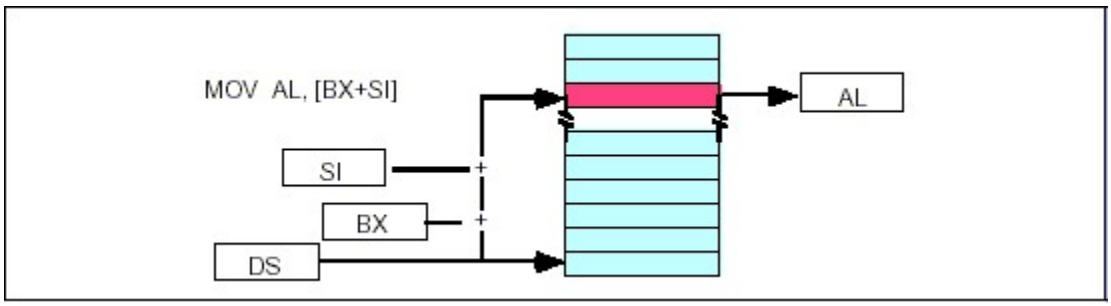

Figure 4.14 Mode [BX+DI]

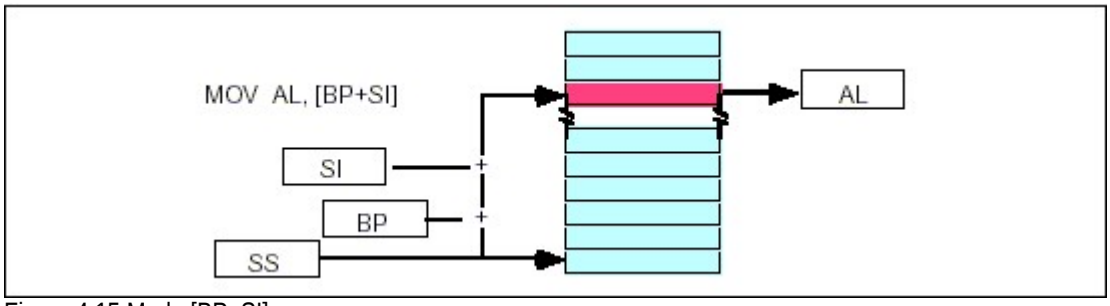

Figure 4.15 Mode [BP+SI]

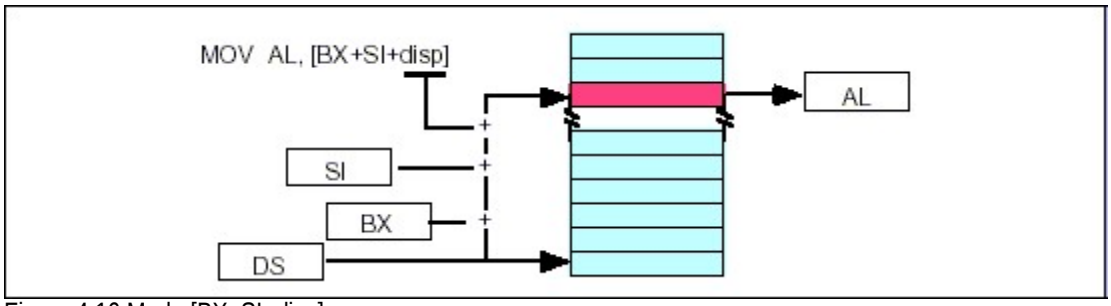

Figure 4.16 Mode [BX+SI+disp]

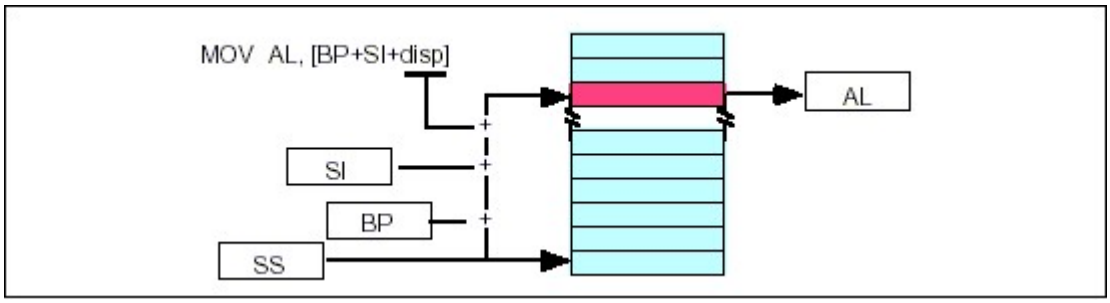

Figure 4.17 Mode [BP+SI+disp]

Dans la figure 4.16 on peut également substituer di pour produire [bx+di+disp] ou [bp+di+disp] si on se réfère au cas de la figure 4.17.

Supposez que bp contient 1000h, que bx contient 2000h, que le registre si contient 120h et que le registre di contient 5. Alors, mov al, 10h[bx+si] chargera le registre al avec l'adresse DS:2130 ; mov ch, 125h[bp+di] chargera ch avec l'emplacement SS:112A ; et, finalement, mov bx, cs2[bx][di] chargera bx avec CS:2007.

4.6.2.6 Un moyen simple de se souvenir des modes d'adressage 8086

Il y a un total de 17 différents modes d'adressage corrects sur le 8086[12] :

disp	[bx]	[bp]	[si]	[di]	disp[bx]	dips[bp]	disp[si]	disp[di]
[bx][si]	[bx][di]	[bp][si]	[bp][di]	disp[bx][si]	disp[bx][di]	disp[bp][si]	disp[bp][di]	

Vous pouvez mémoriser toutes ces formes, de sorte à savoir quelles sont les formes valides (et, par omission, lesquelles sont invalides). Cependant, il y existe un moyen plus simple de s'en souvenir. Considérez le schéma de la figure 4.18.

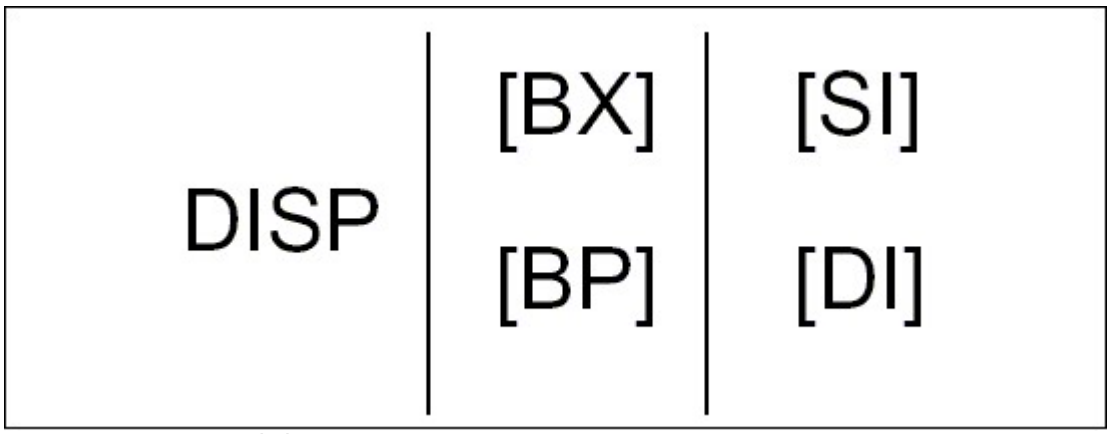

Figure 4.18 Table pour générer des modes d'adressage 8086 valides

Si vous choisissez zéro ou un élément pour chacune des colonnes et formez une expression d'au moins un terme, vous obtenez un mode d'adressage 8086 valide. Quelques exemples :

- En choisissant disp de la colonne 1, rien de la colonne 2 et [di] de la colonne 3. On obtient disp[di].
- En choisissant disp, [bx] et [di], on obtient disp[bx][di].
- En sautant les colonnes 1 et 2 et en choisissant [si], on obtient [si].
- En sautant la colonne 1, en choisissant [bx], puis [di], on a [bx][di].

Il va de soi que si vous avez un mode d'adressage qui ne peut pas être construit à partir de cette table, alors il n'est pas valide. Par exemple, disp[dx][si], n'est pas bon, car le registre dx ne se trouve dans aucune des colonnes ci-dessus.

4.6.2.7 Considérations finales à propos des modes d'adressage 8086

L'*adresse réelle* est l'offset final produit par une opération d'adressage. Par exemple, si bx contient 10h, l'adresse réelle pour 10h[bx] est 20h. Vous verrez ce terme souvent quand il sera question des modes d'adressage 8086. Il y a même une instruction spéciale *lea* (*load effective address* ou *charge de l'adresse effective* en français), qui calcule cette valeur.

Tous les modes d'adressage ne sont pas conçus de la même façon ! Différents modes ont différentes durées d'exécution. Le délai exact varie d'un processeur à l'autre. Généralement, plus un mode d'adressage est complexe plus il prendra de temps pour livrer une adresse réelle. La complexité des modes d'adressage est directement liée au nombre de termes qui les constituent. Par exemple, disp[bx][si] est plus complexe que [bx]. Voir la référence du jeu d'instructions dans les annexes pour plus d'informations à propos de la durée en cycles des modes d'adressage des processeurs 80x86.

[12]Et l'on n'a pas encore compté les variations syntaxiques !

La partie "déplacement" de tous les modes d'adressage, sauf le mode déplacement seul, peut être une constante signée de huit bits ou bien de seize bits. Si l'offset se trouve dans la plage –128...127, l'instruction sera plus courte (et donc plus rapide) par rapport à une instruction ayant un déplacement en dehors de cette plage. La taille de la valeur dans le registre n'affecte pas le temps ou la taille d'exécution. Donc, vous pouvez vous arranger pour mettre un nombre grand dans le registre et utiliser un petit déplacement, ce qui est préférable qu'à l'inverse.

Si le calcul de l'adresse réelle produit une valeur plus grande que 0FFFFh, le CPU ignorera le débordement et le résultat s'enroule pour revenir à zéro. Par exemple, si bx contient 10h, l'instruction mov al, 0FFFFh[bx] chargera le registre al de l'emplacement ds:0Fh, et non 1000Fh.

Dans cette section, vous avez vu comment ces modes d'adressage opèrent. Jusqu'à présent, nous n'avons pas dit pourquoi on les applique. On en parlera bientôt. Pour le moment, c'est déjà suffisant de savoir comment chaque mode calcule son adresse effective.

4.6.3 Les Modes d'adressage des registres sur le 80386

Le 80386 (et les processeurs ultérieurs) dispose de registres de 32 bits. Les huit registres généraux ont tous leur équivalent 32 bits. Ils sont eax, ebx, ecx, edx, esi, edi, ebp et esp. Si vous avez un de ces processeurs, vous pouvez utiliser ces registres en tant qu'opérandes de diverses instructions 80386.

4.6.4 Les Modes d'adressage de la Mémoire sur le 80386

Le processeur 80386 généralise les modes d'adressage mémoire. Là où le 8086 vous permettait seulement d'utiliser bx ou bp comme registres de base et le registre si ou di comme registres d'index, le 80386 permet d'utiliser presque tous les registres généraux de 32 bits comme registres de base ou d'index. De plus, le 80386 a ajouté un nouveau mode d'adressage, le *mode d'adressage scalaire* (*scaled idexed adressing mode*) qui simplifie l'accès aux éléments des tableaux. Au-delà de leur extensions à 32 bits les nouveaux modes d'adressage 80386 sont probablement le meilleur apport de la puce sur les processeurs plus anciens.

4.6.4.1 Les Modes d'adressage pour les registres

Sur le 80386, vous pouvez spécifier tout registre général pour les modes d'adressage indirects. [eax], [ebx], [ecx], [edx], [esi] et [edi] fournissent tous des offsets par défaut, sur le segment des données. Alors que les registres [ebp] et [esp] utilisent par défaut le segment de pile.

Notez que, tout en fonctionnant en mode réel 16 bits sur le 80386, les offsets des registres 32 bits doivent encore être dans la plage 0...0FFFFh. Vous ne pouvez pas utiliser des valeurs plus grandes pour accéder à plus de 64Ko dans un segment[13]. Notez aussi que vous devez cependant utiliser les noms des registres de 32 bits et non pas ceux de 16 bits. Les instructions suivantes montrent les formes correctes :

```
mov     al,    [eax]
mov     al,    [ebx]
mov     al,    [ecx]
mov     al,    [edx]
mov     al,    [esi]
mov     al,    [edi]
mov     al,    [ebp]   ;utilise ss par défaut
mov     al,    [esp]   ;utilise ss par défaut
```

4.6.4.2 Modes d'adressage indexés, basés/indexés et basés/indexés/disp

Les modes d'adressage indexés (registres indirects plus déplacement) permettent de mélanger des registres de 32 bits avec des constantes. Les modes d'adressage basé/indexé permettent d'apparier deux registres de 32 bits. Et, finalement, les modes basé/indexé/déplacement permettent de combiner une constante avec deux

[13]Sauf, naturellement, si vous opérez en mode protégé, où ceci est parfaitement légal.

registres pour former une adresse effective. Gardez à l'esprit que l'offset produit par une opération d'adresse réelle, doit encore être de 16 bits, si on travaille en mode réel.

Sur le 80386, les termes *registre de base* et *registre d'index* prennent réellement leur sens. Quand on combine deux registres de 32 bits dans un mode d'adressage, le premier registre est le registre de base et l'autre est le registre d'index. Ceci est vrai indépendamment des noms des registres. Notez encore que le 80386 permet d'utiliser le même registre comme base ou comme index, ce qui peut être parfois vraiment utile. Les instructions qui suivent donnent des échantillons représentatifs des divers modes d'adressage basés et indexés, ainsi que les variations syntaxiques :

```
mov     al,    disp[eax]       ;adressage indexé
mov     al,    [ebx+disp]
mov     al,    [ecx][disp]
mov     al,    disp[edx]
mov     al,    disp[esi]
mov     al,    disp[edi]
mov     al,    disp[ebp]       ;utilise ss par défaut
mov     al,    disp[esp]       ;idem
```

Les instructions qui suivent utilisent toutes le mode d'adressage basé+indexé. Le premier registre de la seconde opérande est le registre de base, alors que le second est le registre d'index. Si le registre de base est esp ou ebp, alors l'adresse réelle sera relative au segment de pile. Sinon, elle sera relative au segment de données. Notez que le choix du registre d'index n'affecte pas le choix du segment par défaut.

```
mov     al,    [eax][ebx]      ;adressages basé+indexé
mov     al,    [ebx+ebx]
mov     al,    [ecx][edx]
mov     al,    [edx][ebp]      ;utilise DS par défaut
mov     al,    [esi][edi]
mov     al,    [edi][esi]
mov     al,    [ebp+ebx]       ;utilise ss par défaut
mov     al,    [esp][ecx]      ;idem
```

Naturellement, vous pouvez ajouter un déplacement aux modes d'adressages ci-dessus, pour produire le mode basé+indexé+déplacement. Les instructions suivantes donnent encore un exemple des modes possibles :

```
mov     al,    disp[eax][ebx]
mov     al,    disp[ebx+ebx]
mov     al,    [ecx+edx+disp]
mov     al,    disp[edx+ebp]   ;utilise DS par défaut
mov     al,    [esi][edi][disp]
mov     al,    [edi][disp][esi]
mov     al,    disp[ebp+ebx]   ;utilise ss par défaut
mov     al,    [esp+ecx][disp] ;idem
```

Le 80386 pose cependant une restriction sur les registres d'index. Vous ne pouvez pas utiliser esp comme registre d'index. C'est bon de l'utiliser comme base, mais non comme index.

4.6.4.3 Modes d'adressage scalaires du 80386

Les modes d'adressage indexé, basé/indexé et basé/indexé/déplacement décrit ci-dessus sont réellement des exemples spéciaux de *modes d'adressages scalaire* du 80386. Ces modes sont particulièrement utiles dans l'accès aux éléments d'un tableau, même s'ils ne sont pas limités à cet usage. Ils permettent notamment de multiplier l'index du mode d'adressage par 1, 2, 4 ou 8. La syntaxe générale de ces modes est :

```
                disp[index * n]
                [base][index * n]
ou
                disp[base][index * n]
```

où "base" et "index" représentent n'importe quel registre général de 32 bits du 80386 et *n* est la valeur 1, 2, 4 ou 8.

Le 80386 calcule l'adresse effective en additionnant disp, base et index*n. Par exemple, si ebx contient 1000h et esi contient 4, alors :

```
mov al, 8[ebx][esi*4]   ;charge AL avec l'emplacement 1018h
mov al, 1000h[ebx][ebx*2];charge AL avec l'adresse 4000h
mov al, 1000h[esi*8]   ;charge AL avec l'emplacement 1020      h
```

Notez que les modes d'adressage étendus du 80386 indexé, basé/indesé et basé/indexé/déplacement sont vraiment des cas spéciaux de modes d'adressage scalaires, avec n=1. C'est-à-dire, les trois paires d'instructions qui suivent sont absolument identiques :

```
mov al, 2[ebx][esi*1]          mov al, 2[ebx][esi]
mov al, [ebx][esi*1]           mov al, [ebx][esi]
mov al, 2[esi*1]               mov al, 2[esi]
```

Naturellement, MASM permet un tas d'autres variations différentes de ces mêmes modes d'adressage. Ce qui suit donne un simple échantillonnage des possibilités :

```
disp[ebx][esi*1]        [bx+disp][si*2],        [bx+si*2+disp],
[si*2+bx][disp],        disp[si*2][bx],         [si*2+disp][bx], [disp+bx][si*2]
```

4.6.4.4 Quelques considérations finales à propos des modes d'adressage 80386

Puisque les modes d'adressage 80386 sont plus naturels, ils sont beaucoup plus faciles à mémoriser que ceux du processeur 8086. Les programmeurs qui travaillent sur le 80386 ont toujours la tentation d'éviter les modes d'adressage plus anciens et utiliser exclusivement les plus modernes. Cependant, comme vous verrez dans la prochaine section, les modes d'adressage 8086 sont réellement plus efficaces que leurs versions respectives 80386. Par conséquent, il est important de les connaître tous et de ne choisir que ceux qui sont le plus appropriés à résoudre un problème spécifique.

Quand sur le 80386, on utilise les modes basé/indexé et basé/indexé/disp sans options de scalabilité (autrement dit en laissant le facteur scalaire par défaut à 1), le premier registre qui apparaît dans le mode est le registre de base et le second est le registre d'index. Ceci est un point important parce que le choix du segment par défaut est déterminé par le choix du registre de base. Si le registre de base est ebp ou esp, alors le segment par défaut est ss, tandis que dans tous les autres cas, le segment accédé par défaut est ds, même si le registre d'index est ebp. Si on applique l'opérateur d'index scalaire (*n) à un registre, alors ce registre sera toujours un registre d'index, peu importe à quelle place il apparaît dans le mode d'adressage :

```
[ebx][ebp]      ;Utilise DS par défaut
[ebp][ebx]      ;Utilise SS par défaut
[ebp*1][ebx]    ;Utilise DS par défaut
[ebx][ebp*1]    ;Utilise DS par défaut
[ebp][ebx*1]    ;Utilise SS par défaut
[ebx*1][ebp]    ;Utilise SS par défaut
es:[ebx][ebp*1] ;Utilise ES
```

4.7 L'instruction MOV 80x86

Les exemples à travers de ce chapitre font un grand usage de l'instruction mov (abréviation de *move*). En outre, cette instruction est définitivement la plus commune des instructions de la famille 80x86. Il vaut donc la peine de s'attarder un peu sur son fonctionnement.

Comme sa version x86 le mov des processeurs 80x86 est très simple :

```
mov    Destination,    Source
```

Mov, fait une copie de *Source* et l'enregistre dans *Destination*, sans affecter le contenu de *Source*, mais en écrasant la valeur que *Destination* contenait avant l'usage de mov. En gros, mov peut être complètement décrite par l'instruction Pascal suivante :

```
Dest := Source ;
```

Cependant, mov a plusieurs limitations et vous vous en rendrez compte au fur et à mesure que votre progrès en asembleur avance. Pour comprendre pourquoi ces limitations existent il faut jeter un œil sur le code machine des diverses formes de cette instruction. Une mise en garde : on n'appelle pas le 80386 un processeur CISC (Complex instruction Set Computer[14]) pour rien. L'encodage de l'instruction mov est probablement le plus complexe de tous les encodages existants. Néanmoins, on ne pourra pas évaluer pleinement les possibilités de cette instruction sans l'étude de son code machine et, également, on ne peut pas vraiment avoir une idée exacte de la façon d'écrire du code optimal avec cette instruction. Vous commencez à voir maintenant pourquoi, dans le chapitre précédent, on a travaillé avec les processeurs x86 au lieu de passer directement à la vraie programmation 80x86.

L'instruction mov existe sous plusieurs versions. Sur le 80386, le seul mot-clé mov représente une douzaine d'instructions différentes. La plus commune a l'encodage binaire illustré par la figure 4.19.

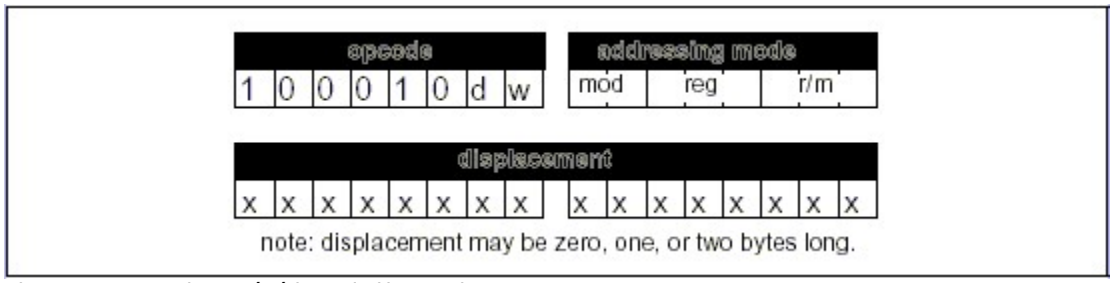

Figure 4.19 Encodage générique de l'instruction MOV

Le premier octet de l'instruction est son *opcode*. Les bits 0 (w) et 1 (d) définissent respectivement la largeur de l'instruction (8, 16 et 32 bits) et la direction du transfert. À l'heure d'expliquer des instructions spécifiques, ce livre vous indiquera toujours de façon explicite les valeurs *d* et *w,* ceci parce que les autres livres ne le font pas en s'attendant que vous même les déduisiez.

Le second octet représente le mode d'adressage, appelé familièrement "mod-reg-r/m". Cet octet choisit entre 256 possibles combinaisons d'opérandes permises pour l'instruction mov générique, qui peut prendre une des ces trois formes :

```
mov     reg, adr
mov     adr, reg
mov     reg, reg
```

Notez qu'au mons une des opérandes est un registre général. Le champ *reg* dans l'octet mod/reg/rm indique justement le registre à utiliser (ou un des registres s'il s'agit de la troisième instruction de l'exemple). Le bit d (direction) spécifie si les données doivent être stockées dans le registre (d = 1) ou dans la mémoire (d = 0).

Les bits dans le champ reg permettent de sélectionner un parmi huit différents registres. Le 8086 supporte huit registres de 8 bits et huit registres de 16 bits. Le 80386 supporte aussi huit registres de 32 bits (dans les trois cas, on parle de registres généraux). Le CPU décode la signification du champ reg comme suit :

Table 23 : Les encodages du champ REG

		mode 16 bits	mode 32 bits
reg	w = 0	w = 1	w = 1
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP

[14]Ou *Ensemble Complexe d'instructions d'Ordinateur* en français, n.d.t.

101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Pour différencier les registres de 16 bits des registres de 32 bits, les processeurs 80386 et ultérieurs utilisent un octet spécial servant de préfixe d'opcode avant les instructions qui utilisent les registres de 32 bits. Sinon, l'encodage est le même pour les deux types d'instruction.

Le champ *r/m*, combiné avec le champ *mod* indique le mode d'adressage. L'encodage du champ mod est le suivant :

MOD	Signification
00	Quand MOD vaut 00, le champ r/m indique un mode d'adressage indirect (voir la table d'encodage du champ r/m), *sauf* si r/m contient 110. Si MOD = 00 et r/m = 110, alors, il s'agit d'un déplacement seul.
01	Quand MOD vaut 01, le champ r/m indique un mode d'adressage de type *indexé* ou de type *basé / indexé / déplacement*. Le déplacement est une valeur signée de huit bits et suit l'octet mod-reg-r/m.
10	Quand MOD vaut 10, le champ r/m indique un mode d'adressage de type *indexé*, ou de type *basé / indexé / déplacement*. Le déplacement est une valeur signée de seize bits (en mode 16 bits), ou bien une valeur de trente-deux bits (en mode 32 bits) et elle suit l'octet mod-reg-r/m.
11	Quand MOD vaut 11, le champ r/m indique un registre et utilise le même encodage du champ reg.

Table 24 : L'encodage MOD

Le champ mod doit, en outre, choisir entre une opération de type registre-registre et une de type registre-mémoire ou mémoire-registre. Il doit également déterminer la taille du déplacement (0, 1, 4 ou 8 octets) qui suit l'instruction du mode d'adressage. Si MOD = 00, alors on a un mode d'adressage sans déplacement (registre indirect ou basé/indexé). Notez le cas spécial où MOD=0 et r/m = 110. Ceci correspondrait normalement au mode d'adressage [bp], mais le 8086 se sert de ce type d'encodage pour définir le mode d'adressage du déplacement seul. Ce qui veut dire que, *sur le 8086, il n'y a pas un vrai mode d'adressage [bp]*.

Pour comprendre pourquoi vous pouvez utiliser le mode d'adressage [bp] dans vos programmes, observez les cas où MOD=01 et MOD=10. Ces dispositions activent les modes d'adressage *disp[reg]* et *disp[reg][reg]*. Et alors ?, vous direz. Ce n'est pas la même chose que le mode d'adressage [bp]. Et vous avez raison. Néanmoins, considérez les instructions suivantes :

```
mov     al,     0[bx]
mov     ah,     0[bp]
mov     0[si],  al
mov     0[di],  ah
```

Ces instructions, qui utilisent le mode d'adressage indexé, effectuent la même opération que celles qui utilisent les modes analogues de registre indirect (obtenues en enlevant la partie *déplacement* des instructions ci-dessus). La seule véritable différence entre les deux versions est que le mode d'adressage indexé est d'un octet plus long (si MOD = 01 ou deux octets plus long si MOD = 10), pour stocker le déplacement de 0. Puisqu'elles sont plus longues, ces versions s'exécutent un peu plus lentement.

Cette caractéristique du 8086 - fournir deux ou plusieurs façons de faire la même chose - se reflète dans tout le jeu d'instructions. Vous verrez bien d'autres exemples de ce fait avant de terminer avec l'instruction mov. MASM utilise automatiquement la meilleure forme de l'instruction. En asemblant le code ci-dessus, MASM générera le mode d'adressage registre indirect pour toutes les instructions, excepté mov ah, 0[bp]. Cependant, il émettra seulement un déplacement d'un octet, ce qui est plus court et plus rapide que la même instruction avec un

déplacement zéro de deux octets. Notez que MASM ne requiert pas que vous entrez 0[bp], vous pouvez entrer [bp] et MASM placera automatiquement le zéro pour vous.

Si MOD ne vaut pas 11b, le champ r/m encode les modes d'adressage de la manière suivante :

R/M	Mode d'adressage (en supposant que MOD = 00, 01 ou 10)
000	[BX+SI] ou DISP[BX][SI] (dépend de MOD)
001	[BX+DI] ou DISP[BX][DI] (dépend de MOD)
010	[BP+SI] ou DISP[BP][SI] (dépend de MOD)
011	[BP+DI] ou DISP[BP][DI] (dépend de MOD)
100	[SI] ou DISP[SI] (dépend de MOD)
101	[DI] ou DISP[SI] (dépend de MOD)
110	Déplacement seul ou DISP[BP] (dépend de MOD)
111	[BX] ou DISP[BP] (dépend de MOD)

Table 24 : L'encodage R/M

N'oubliez pas que les modes d'adressage impliquant bp utilisent le segment de pile (ss) comme segment par défaut. Ceux qui impliquent tout autre registre utilisent le segment de données (ds) par défaut.

Si cette explication vous a complètement dérouté, vous n'avez encore vu le pire. Gardez à l'esprit que ces modes examinés ne sont qu'une partie des modes d'adressage du 8086. *Il vous reste encore à observer les modes d'adressage du 80386*. Vous êtes probablement en train de commencer à comprendre qu'est-ce qu'Intel veut dire quand ils parlent de *jeu d'instructions complexe*. Cependant, le concept important à noter est qu'on peut construire des instructions 80x86 de la même façon qu'on l'a fait avec les versions x86 au Chapitre 3 – en construisant l'instruction bit à bit. Pour des détails complets sur la façon dont les processeurs 80x86 encodent les instructions, voir les annexes.

4.8 Considérations finales à propos de l'instruction MOV

Il y divers facteurs importants que vous devez toujours garder à l'esprit à propos de l'instruction MOV. Avant tout, il n'y a pas de *mov adresse, adresse*. Pour des raisons obscures beaucoup de débutants ont du mal à saisir ce point. Même s'il existe tout quand même une paire d'instructions permettant d'effectuer cette opération, c'est toujours plus efficace de charger un registre et ensuite garder la valeur. Un autre fait important est qu'il y a plusieurs versions de l'instruction mov permettant de faire les mêmes choses. De même, il y a plusieurs modes d'adressage différents qu'on peut utiliser pour avoir accès au même emplacement de mémoire. Si vous avez intérêt à écrire les programmes les plus petits et les plus courts possibles, vous devez toujours être vigilant dans le choix entre versions différentes d'instructions équivalentes.

L'explication de ce chapitre se limite principalement à l'instruction mov générique et vous donne une idée sur le moyen utilisé par les processeurs 80x86 pour encoder la mémoire et les modes d'adressage. D'autres versions de l'instruction mov, vous permettent de transférer les données entre les registres généraux de 16 bits et les registres de segment. D'autres encore, vous permettront de charger un registre de 16 bits ou un emplacement de mémoire avec une constante. Ces versions utilisent un opcode qui n'est plus comme celui de l'instruction mov générique. Pour plus de détails, donnez un coup d'œil à l'annexe D, qui traite l'encodage des instructions.

Sur le processeur 80386 il y a encore plusieurs versions additionnelles de l'instruction, vous permettant de charger des données dans des registres réservés. Ce livre ne les prendra pas en compte. Il y a également certaines instructions de chaînes de caractères qui permettent d'effectuer des opérations de transfert de type mémoire-mémoire. Vous verrez ces instructions dans le prochain chapitre, mais elles ne sont pas une bonne alternative à l'instruction mov.

4.9 Exercices de laboratoire

Il est heure désormais de commencer à travailler avec le vrai langage assembleur des processeurs 80x86. Pour ce faire, il vous faudra aussi un aperçu sur le support logiciel auxiliaire. Dans cet ensemble d'exercices vous apprendrez à utiliser les outils de base pour éditer votre code source, l'assembler, le déboguer et l'exécuter. Ces exercices supposent que vous ayez déjà installé MASM (Microsoft Macro Assembler). Si vous ne l'avez pas encore fait, faites-le maintenant (en suivant les directives de Microsoft). Avant de commencer les exercices de ce laboratoire, il faut absolument que ce programme soit présent.

4.9.1 La Bibliothèque standard UCR pour les programmeurs 80x86

La plupart des programmes exposés dans ce livre utilisent un ensemble de routines de bibliothèque standard créées à l'*Université Riverside de Californie*. Ces routines fournissent des E/S standardisées, et du support pour les chaînes de caractères, pour les opérations arithmétiques et pour d'autres fonctions utiles. La bibliothèque en soi est très semblable à celle d'usage commun chez les programmeurs C/C++. Des chapitres ultérieurs de ce livre décrivent un grand nombre de ces routines, il n'y a pas besoin d'y aller tout de suite. Cependant, plusieurs exemples de ce chapitre et des chapitres suivants utilisent certaines de ces routines aussi, donc la bibliothèque doit être présente dans votre disque dur et activée maintenant.

Elle apparaît sur le CD qui accompagne ce livre[15]. Vous aurez besoin de la copier sur votre disque dur. Un ensemble de commandes comme le suivant (avec les ajustements appropriés selon la lettre de votre lecteur de CD) feront le travail :

```
c:
cd \
md stdlib
cd stdlib
xcopy r:\stdlib\*.* . /s
```

où r représente la lettre de votre CD ou tout autre répertoire source d'emplacement de la bibliothèque. Une fois que vous avez copié la bibliothèque sur votre disque dur, des commandes additionnelles sont encore nécessaires et doivent être utilisées avant d'assembler les programmes qui utilisent la bibliothèque :

```
set include=c:\stdlib\include
set lib=c:\stdlib\lib
```

Ce serait peut-être une bonne idée de placer ces commandes dans votre fichier AUTOEXEC.BAT, de façon qu'elles s'exécutent automatiquement à chaque démarrage de votre ordinateur. Mais si vous n'avez pas effectué ces commandes, d'une façon ou de l'autre, MASM émettra des messages d'erreur lors de l'assemblage des programmes qui utilisent la bibliothèque.

4.9.2 Éditez vos fichiers source

Avant d'assembler et d'exécuter vos programmes, vous avez évidemment besoin de les éditer. MASM accepte tous les fichiers texte, le type d'éditeur que vous utilisez n'est pas important, l'important est que le contenu des fichiers source soit de type ASCII. Notez que la plupart des traitements de texte normalement ne produisent pas des fichiers textes ASCII, donc vous ne devriez pas les utiliser pour écrire vos fichiers en assembleur.

MS-DOS, Windows et MASM sont fournis tous les trois avec de simples éditeurs standard que vous pouvez utiliser pour créer et modifier des fichiers source. Le programme EDIT.EXE est livré avec MS-DOS ; NOTEPAD.EXE avec Windows ; et, finalement, PWB (Programmer's Work Bench) avec MASM. Si vous n'avez pas un éditeur de texte favori, utilisez un de ces programmes. Si vous avez certains compilateurs installés (comme Borland C++, Delphi ou MS Visual C++) vous pouvez utiliser leurs éditeurs également.

Étant donné la grande variété des éditeurs disponibles, ce chapitre ne donnera pas d'instructions à propos de comment les utiliser. Si vous n'avez jamais utilisé un éditeur de texte ou un ordinateur[16], consultez les documentations appropriées.

[15]Mais vous pouvez aussi la télécharger de l'internet dans le site https://www.plantation-productions.com/Webster/, n.d.t.

[16]Cependant, il est fort improbable que ceux qui n'ont jamais touché à un ordinateur se mettent à faire de l'assembleur. C'est improbable qu'ils en connaissent, voire, l'existence, n.d.t.

4.9.3 Le fichier SHELL.ASM

Bien que vous pouvez écrire un programme d'assembleur à partir d'une feuille blanche et en utilisant l'éditeur de texte de votre choix, les programmes en assembleur, même les plus simples, contiennent un grand nombre d'instructions communes. Dans le répertoire du chapitre quatre (CH04) il y a un fichier nommé "SHELL.ASM". Ce fichier constitue la squelette d'un programme assembleur ordinaire[17]. Il contient toutes les instructions d'en-tête ("overhead") nécessaires pour mettre en place un programme fonctionnel en assembleur, mais sans les instructions et les variables qui permettent à ce programme de faire quelque chose. Sous certains aspects, cela est comparable au programme Pascal suivant :

```
program shell(input, output);
begin
end.
```

SHELL.ASM est un programme valide. Vous pouvez l'assembler et l'exécuter, mais il ne fera rien.

La principale finalité de ce programme est qu'il contient beaucoup de lignes de code qui doivent paraître dans un programme assembleur vide, seulement pour respecter les exégences du langage. Malheureusement, comprendre la signification de ces instructions demande une étude considérable et suppose donc une expérience en programmation que le débutant n'a certainement pas. Donc, au lieu de n'écrire aucun programme jusqu'à la compréhension de tout ce qui est nécessaire, vous utiliserez aveuglement le fichier SHELL.ASM comme modèle sans vous poser des questions sur la signification de ces lignes. Dans les deux prochains chapitres, toutes ces lignes commenceront à avoir du sens. Mais pour maintenant, utilisez simplement le fichier. La seule chose que vous avez besoin de connaître maintenant à propos de SHELL.ASM est où placer votre code source. Ceci est indiqué par trois lignes de commentaires qui vous indiquent où placer vos variables (s'il y en aura), vos sous-routines/procédures/fonctions (s'il y en aura) et les instructions de votre programme principal. Ce qui suit, constitue le listing complet du fichier SHELL.ASM, qui pourra être utilisé par ceux qui n'ont pas d'accès à la version électronique.

```
                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

dseg            segment         para public 'data'

;C'est l'endroit où mettre les variables globales

dseg            ends
cseg            segment         para public 'code'
                assume          cs:cseg, ds:dseg

;Les variables qui apparaissent ci-dessous sont utilisées par les
;routines de la bibliothèque standard. La routine MemInit utilise
;les étiquettes "PSP" et "zzzzzzseg". Si vous voulez utiliser
;les routines getenv, memInit, malloc et free, elles doivent être
;présentes.

                public          PSP
PSP             dw              ?
;-----------------------------------------------------------------
;Ici, c'est un bon endroit pour placer d'autres routines :
;-----------------------------------------------------------------
;Main est le programme principal. L'exécution des programmes
;commence toujours ici.

Main            proc
                mov     cs:PSP, es      ;enregistre le préfixe du
                                        ;segment du programme
```

[17]Ce fichier est également disponible dans le CD qui accompagne ce livre.

```
                mov     ax, seg dseg    ;installe le segment
                mov     ds, ax
                mov     es, ax

                mov     dx, 0
                meminit
                jnc     GoodMemInit
                print
                db      "Error initializing memory manager", cr, 1f, 0
                jmp     Quit
GoodMemInit:
;*****************************************************************
;Placez votre programme principal ici
;*****************************************************************

Quit:           ExitPgm
Main:           endp
cseg            ends

;Allouer un montant raisonnable d'espace pour la pile (2k)

sseg            segment         para stack 'stack'
stk             db              256 dup ("stack ")
sseg            ends

;zzzzzzseg doit être le dernier segment à être chargé en mémoire !

zzzzzzseg       ssegment        para public 'zzzzzz'
LastBytes       db              16 dup(?)
zzzzzzseg       ends

                end             Main
```

Même si vous devez simplement accepter ce code sans question, certaines explications sont toutefois nécessaires. Ce programme commence par deux instructions "include" et "includelib". Ces instructions indiquent à l'assembleur et à l'éditeur de liens que ce code utilisera certaines des routines de la bibliothèque UCR Standard.

Notez que les phrases qui commencent par un point-virgule sont des commentaires. L'assembleur ignore tout le texte qu'il trouve entre le ";" et la fin de la ligne. Tout comme les commentaires sont très importants dans les langages de haut niveau pour expliquer certaines opérations d'un programme, ici aussi ils sont nécessaires pour décrire certaines parties importantes du fichier SHELL.ASM[18].

La section d'intérêt suivante est la ligne qui commence avec dseg segment .... C'est le commencement de votre zone de données globales. Cette ligne définit le début de votre segment de données (*dseg* veut dire *data segment*), qui termine avec la ligne *dseg ends*. Vous devez placer toutes vos variables globales entre ces deux instructions.

Ensuite, c'est le tour du segment de code (nommé *cseg*), endroit où vous écrirez les instructions 80x86. La chose importante à noter ici est le commentaire "Placez ici votre programme principal". Pour le moment, vous ignorerez toute autre chose dans le code s'occupant du segment de code, hormis ce commentaire. La séquence des programmes que vous créez doit se trouver entre les lignes des astérisques qui l'entourent. Ne vous intriguez pas pour le moment. Vous comprendrez tout bientôt. Tout expliquer maintenant serait une digression trop longue.

Finalement, on trouve dans le programme deux segments additionnels : sseg et zzzzzzseg. Ces segments sont absolument nécessaires (*sseg* est requis par le système, alors que *zzzzzzseg* est requis par la bibliothèque standard UCR). Vous ne devrez en aucun cas les modifier.

[18]En fait, quand vous écrirez vos propres programmes, ce sera utile d'enlever certains commentaires, comme "insérez vos variables globales ici". Ces commentaires servent uniquement pour ceux qui sont en train d'apprendre à utiliser le fichier SHELL.ASM. Un commentaire semblable résulterait très gauche à l'intérieur d'un vrai programme.

En écrivant un nouveau programme, *ne modifiez pas* SHELL.ASM directement. Vous devrez d'abord en faire une copie. Par exemple, si vous pouvez copier le fichier dans PROJECT1.ASM et ensuite apporter toutes vos modifications à ce dernier fichier, vous resterez avec un SHELL.ASM intact et disponible pour vos projets ultérieurs.

Il y a aussi une version spéciale de SHELL.ASM : X86.ASM, qui contient quelques codes additionnels pour donner appui aux projets de programmation de ce chapitre. Veuillez visiter la section concernant ces programmes pour plus de détails.

4.9.4 Assemblez votre code avec MASM

Pour lancer MASM, il faut utiliser le programme ML.EXE (et aussi le programme qui édite les liens (linker)). Normalement, un tel fichier devrait se trouver dans un répertoire comme C:\MASM611\BIN. Vous devriez vérifier si vous avez ce chemin dans les variables d'environnement. Si vous ne l'avez pas, vous devrez ajuster les chemins DOS de façon à y inclure des variables contenant les programmes ML.EXE, LINK.EXE, CV.EXE et d'autres programmes relatifs à MASM.

MASM est un programme basé sur DOS. Il faut donc l'exécuter à partir de la ligne de commande ou bien à partir d'une fenêtre de ligne de commande. La commande de base de MASM prend la forme suivente :

```
ml {options} nomfichier.asm
```

Notez que ML requiert que vous incluez l'extension ".asm" au nom du fichier que vous voulez assembler.

La plupart du temps, vous utiliserez seulement l'option "/Zi". Ceci indique à MASM d'inclure dans le fichier .exe des informations symboliques pour le débogage avec le programme CodeView. Ceci crée souvent un fichier plus grand, mais il facilite aussi considérablement la possibilité de tracer dans un programme avec CodeView (voir "Débogueurs et CodeView[tm]" au prochain paragraphe). Normalement, on utilise toujours cette option pendant le développement et on la saute à l'heure de l'assemblage final, quand tous les problèmes ont été réglés.

Une autre option utile à utiliser est "/?", utilisée sans nom de fichier, la commande *help* permettant d'afficher l'aide en ligne. Si ML trouve cette option, il affichera alors la liste de toutes les options que ML.EXE accepte. Beaucoup de ces options sont rarement, sinon jamais, utilisées. Pour avoir plus de détails, consultez la documentation de MASM à ce propos.

Taper une commande de la forme "ML /Zi myprgr.asm" produit deux nouveaux fichiers (en supposant qu'il n'y avait pas d'erreurs dans le code source) : myprgr.obj et myprgr.exe. Le fichier OBJ (*object code file* ou *fichier de code objet* en français), est un fichier intermédiaire utilisé par l'assembleur et l'éditeur de liens. La plupart du temps, vous pouvez supprimer ce fichier si votre programme ne contient qu'un fichier source. Le fichier myprgr.exe est la version exécutable du programme. Pous pouvez le lancer directement à partir du DOS ou à travers le débogueur CodeView (c'est souvent le meilleur choix).

4.9.5 Débogueurs et CodeView[tm]

Le programme SIMx86 est un exemple de programme de débogage très simple. L'existence de plusieurs programmes de débogage disponibles pour les assembleurs 80x86 ne devrait pas vous surprendre. Dans ce chapitre, vous apprendrez le fonctionnement de base du débogueur CodeView. Il s'agit d'un produit professionnel avec beaucoup d'options et de caractéristiques différentes. Certes, ce court chapitre ne peut pas décrire toutes les façons possibles de l'utiliser. Cependant, vous apprendrez comment utiliser certaines des commandes et des techniques de débogage les plus communes.

Une des majeures difficultés pour décrire un système comme CodeView est que Microsoft le met à jour continuellement, ce qui provoque des changements sur les images des écrans et le fonctionnement des diverses commandes. Il est tout à fait possible que vous utilisiez une version de CodeView plus ancienne que celle présentée ici, ou bien que ce chapitre décrive une version plus ancienne par rapport à la vôtre (en fait, on utilise ici CodeView 4.0). Bon, ne vous en faites pas. Les principes de base restent les mêmes et vous ne devriez pas avoir de problèmes à vous adapter aux différentes versions.

Note : ce chapitre suppose que vous êtes en train d'exécuter la version DOS de CodeView. Si vous utilisez une fenêtre DOS sous Windows, les écrans peuvent avoir une apparence différente.

4.9.5.1 Un aperçu rapide de CodeView

Pour exécuter CodeView, tapez simplement cette commande à l'invite DOS :

```
CV      programme.exe
```

programme.exe est le nom du programme que vous voulez déboguer (le suffixe .exe est optionnel). Mais CodeView requiert, pour fonctionner, un programme de type .exe ou .com. Si vous ne spécifiez pas un programme de ce type, CodeView continuera toujours à vous demander un nom de programme à chaque lancement.

CodeView demande un programme exécutable comme paramètre de ligne de commande. Puisque vous n'avez probablement pas encore écrit de tel programme en assembleur, vous n'avez pas d'exécutable à fournir à CodeView. Mais, pour y remédier, vous pouvez utiliser SHELL.EXE, disponible dans le sous-répertoire du chapitre 4. Pour exécuter CodeView avec le programme SHELL.EXE, tapez à l'invite DOS "CV SHELL.EXE". Ceci fera apparaître un écran comme celui de la figure 4.20.

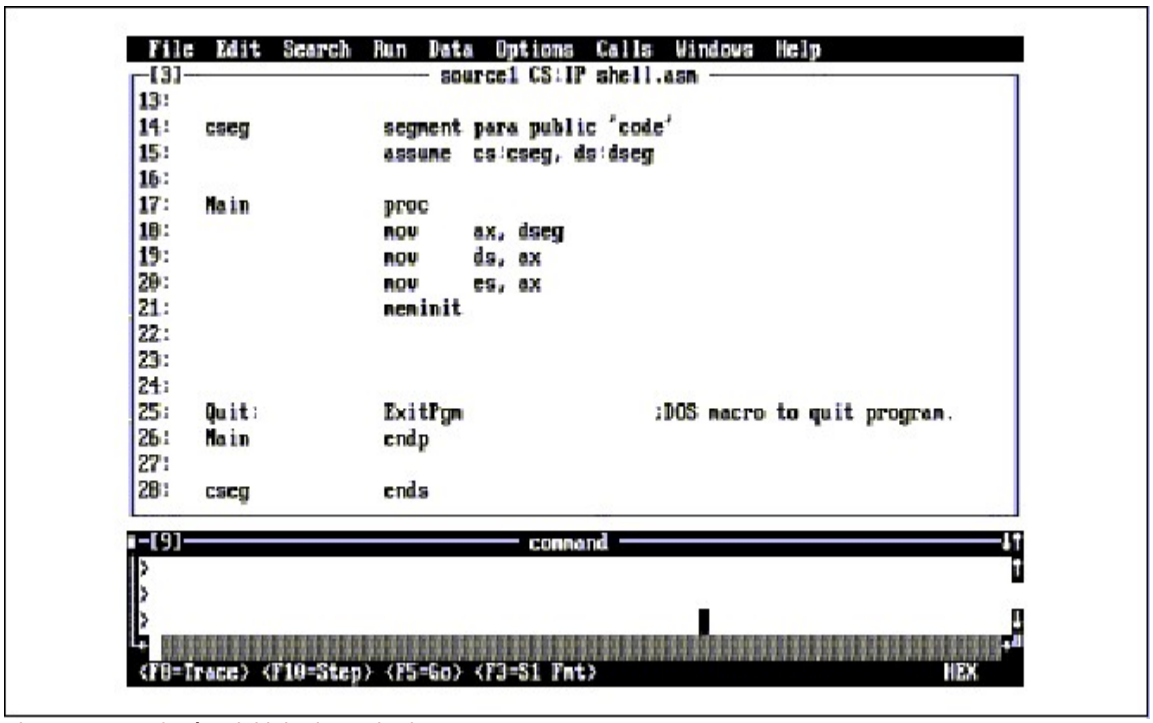

Figure 4.20 La fenêtre initiale de CodeView

Dans l'écran de la figure, il y a quatre sections : la barre des menus à la première ligne, la fenêtre du code source (Source 1), la fenêtre de commande et la ligne d'aide d'état. Notez que CodeView a également plusieurs autres fenêtres et qu'il est en mesure de se souvenir quelles fenêtres étaient ouvertes la dernière fois qu'il a été exécuté, donc, il se peut qu'il affiche des fenêtres différentes que celles qu'on montre ici dans la figure. Au début, la fenêtre de commande est la fenêtre active. Cependant, vous pouvez facilement passer d'une fenêtre à l'autre en appuyant sur la touche F6.

Ces fenêtres sont totalement configurables. Le menu *Window* vous permet de sélectionner quelles fenêtres apparaissent à l'écran. Comme dans beaucoup de produits Microsoft, vous sélectionnez des éléments dans la barre des menus en appuyant sur la touche alt et ensuite la première lettre du nom du menu que vous voulez ouvrir. La figure 4.21 montre le menu *Window* ouvert en appuyant sur alt+W.

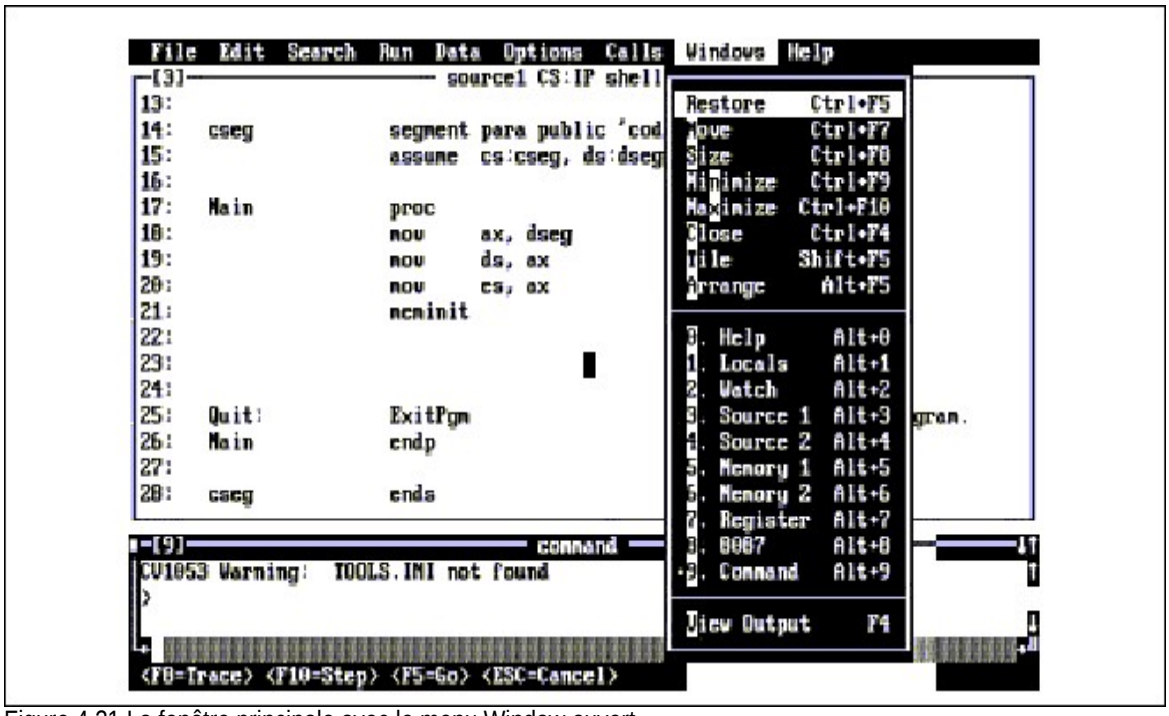

Figure 4.21 La fenêtre principale avec le menu Window ouvert

4.9.5.2 La fenêtre source

Le éléments Source1 et Source2 du menu Window vous permettent d'ouvrir des fenêtres sources additionnelles,de façon à pouvoir voir en même temps différentes sections du programme que vous êtes en train de déboguer. Les fenêtres Source sont utiles pour le débogage au niveau source.

4.9.5.3 La fenêtre Memory

L'élément Memory du menu Window ouvre un écran permettant d'afficher et de modifier des valeurs en mémoire. Par défaut, la fenêtre affiche les variables qui se trouvent dans votre segment de données, mais on peut afficher n'importe quelle valeur de n'importe quel segment, simplement en indiquant son adresse.

La figure 4.22 est un exemple d'affichage de la mémoire.

Les valeurs du côté gauche de l'écran sont des adresses de mémoire segmentée. Les colonnes hexadécimales du centre de l'écran représentent les valeurs pour 16 octets à partir de l'adresse spécifiée. Finalement, les caractères qui apparaissent à la droite de l'écran représentent le contenu ASCII de chacun des 16 octets de la portion de mémoire. Notez que CodeView affiche un point pour tous les caractères ASCII qui ne sont pas imprimables.

Quand vous l'ouvrez, cette fenêtre commence normalement par afficher les contenus de la mémoire à partir de l'offset zéro du segment de données. Il y a deux façons d'afficher des emplacements de mémoire différents. D'abord, vous pouvez utiliser les touches PgUp et PgDn pour parcourir la mémoire[19]. Une autre option est de placer le curseur sur une portion de segment ou d'offset et taper une nouvelle valeur. Dès que vous tapez un chiffre, CodeView affiche automatiquement la donnée de la nouvelle adresse.

Si vous voulez modifier des valeurs en mémoire, déplacez simplement le curseur sur l'octet désiré et tapez la nouvelle valeur hexadécimale. CodeView met automatiquement à jour l'octet correspondant.

[19]Naturellement, cela peut se faire aussi par la souris.

On peut ouvrir des fenêtres multiples de mémoire à la fois. Chaque fois que vous sélectionnez Memory depuis View Memory, CodeView ouvrira une autre fenêtre de mémoire. Avec plusieurs fenêtres de mémoire ouvertes, on peut comparer les valeurs de plusieurs emplacements non contigus. Souvenez-vous que pour passer d'une fenêtre à l'autre on peut appuyer sur F6.

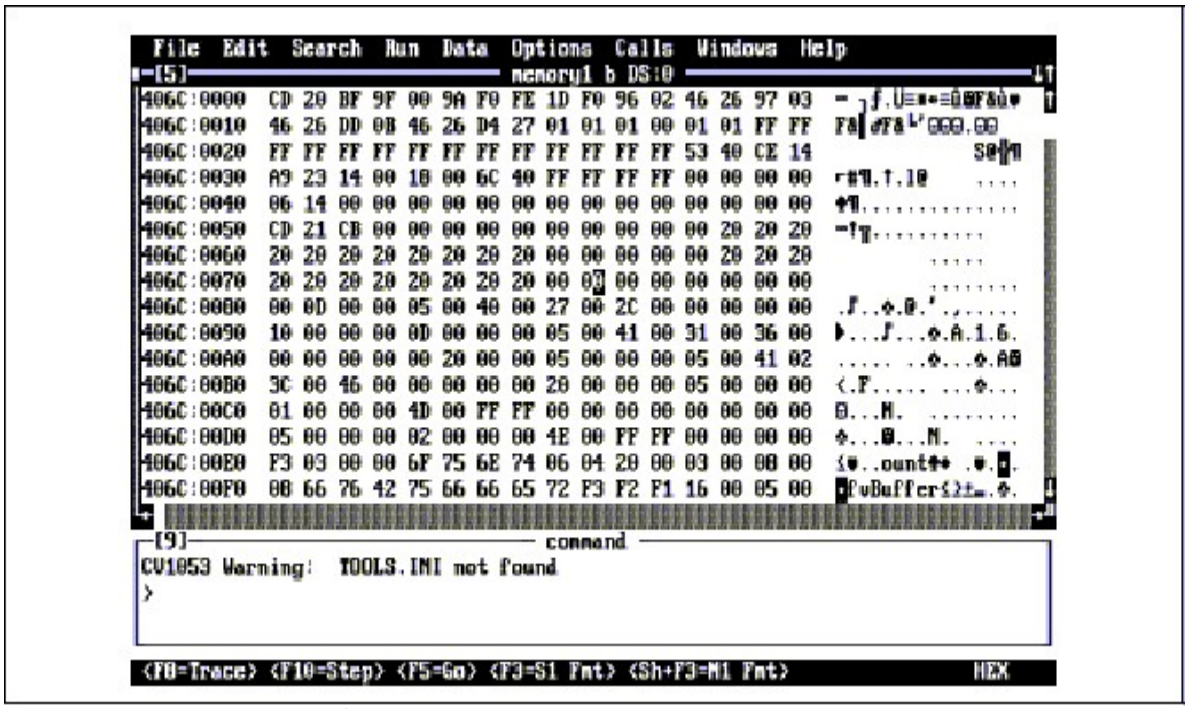

Figure 4.22 Un affichage de la mémoire

Appuyer sur Shift-F3 permet de choisir l'affichage des données en plusieurs formats, hexadécimaux, octets, caractères ASCII, mots, doubles-mots, entiers (signés), valeurs à virgule flottante et d'autres types encore. Ceci est utile quand on veut observer la mémoire de différentes façons. Vous avez seulement l'option d'afficher le contenu de la fenêtre entière dans un type de données unique. Cependant vous pouvez bien entendu ouvrir plusieurs fenêtres de mémoire et afficher dans chacune un type de données différent.

4.9.5.4 La fenêtre Register

L'élément Register du menu Window active ou annule l'affichage de la fenêtre des registres 80x86, qui affiche les valeurs actuelles de tous les registres (voir figure 4.23).

Pour modifier la valeur d'un registre, il faut activer la fenêtre des registres (via la touche F6, si elle n'a pas déjà le focus d'entrée), et placer le curseur sur la valeur que vous souhaitez changer. Tapez une nouvelle valeur et le registre sera automatiquement mis à jour. Notez que le sigle FL signifie *flags* (drapeaux d'état en français). Vous pouvez également modifier ces drapeaux en entrant une nouvelle valeur après l'entrée FL=. Un autre moyen de les changer est de placer le curseur sur une des entrées au bas de la fenêtre des registres, presser une touche alphabétique (par exemple "A") du clavier. Ceci provoque le changement d'état du drapeau spécifié. Les valeurs de ces drapeaux sont 0/1 : overflow=(OV/NV), direction=(DN/UP), interrupt=(DI/EI), sign=(PL/NG), zero=(NZ/ZR), auxiliary carry=(NA/AC), parity=(PO/PE), carry=(NC/CY).

Notez que la touche F2 fait disparaître la fenêtre des registres ou bien lui donne le focus. Cette caractéristique est très utile dans le débogage des programmes. La fenêtre des registres occupe environ 20 % de l'affichage et tend à chevaucher les autres fenêtres. Cependant, vous pouvez rapidement la rappeler ou la cacher en appuyant simplement sur F2.

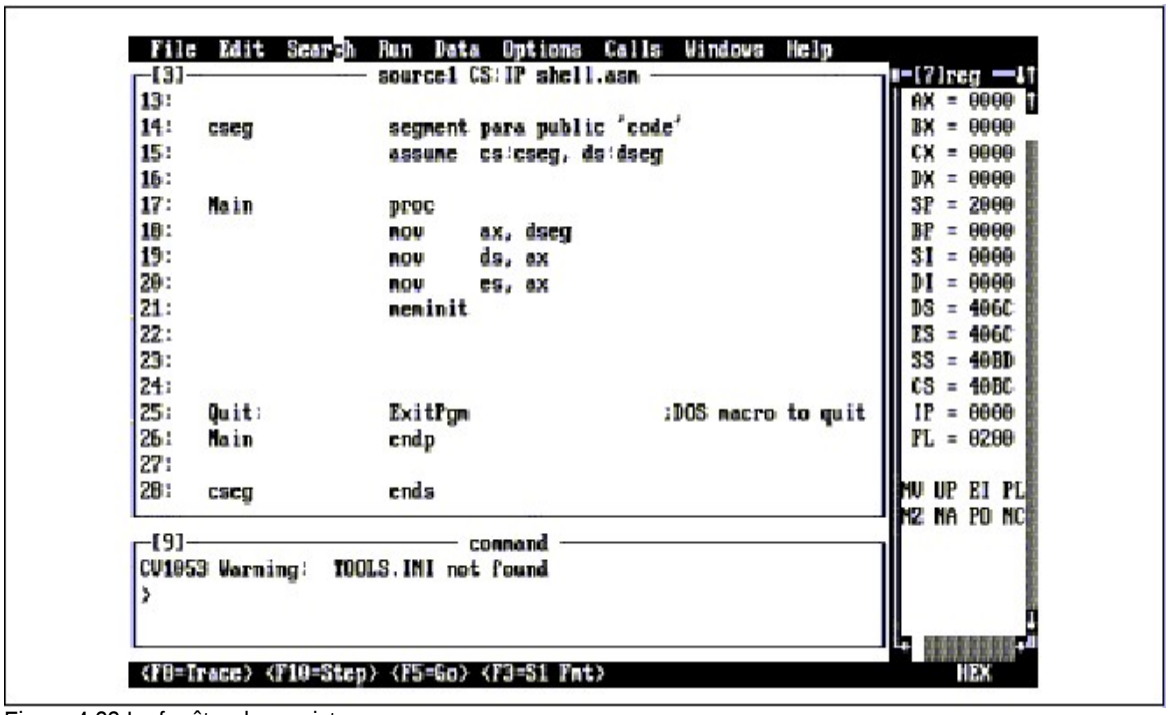

Figure 4.23 La fenêtre des registres

4.9.5.5 La fenêtre Command

Cette fenêtre vous permet d'entrer des commandes textuelles. Même si la Bien plupart des commandes disponibles dans cette fenêtre sont également disponibles ailleurs, certaines opérations sont plus faciles à accomplir par des commandes. De plus, avec la fenêtre des commandes, vous pouvez généralement exécuter une séquence de commandes complètement différentes beaucoup plus rapidement qu'en utilisant d'autres fenêtres de CodeView. Le fonctionnement de ces commandes sera le sujet de la prochaine section de ce chapitre.

4.9.5.6 L'élément output du menu Window

Le fait de sélectionner View Output du menu Window (ou le fait de presser la touche F4) fait passer l'affichage entre l'écran de CodeView et l'écran d'exécution (ou écran des sorties). Pendant l'exécution d'un programme, CodeView affiche normalement l'écran des sorties. Une fois que le contrôle revient à l'écran de CodeView, la fenêtre de débogage masque la fenêtre d'exécution. Si vous avez besoin de donner un coup d'œil rapide à l'exécution du programme pendant que vous êtes sur l'écran de CodeView, appuyez simplement sur F4.

4.9.5.7 La fenêtre Command de CodeView

CodeView est en réalité deux débogueurs en un. D'une part il constitue un système moderne de débogage utilisant des fenêtres. Et de l'autre, il peut fonctionner comme un débogueur traditionnel par ligne de commande. La fenêtre des commandes vous permet de combiner les deux caractéristiques. Si vous l'activez, vous pourrez entrer des commandes de débogage à partir du clavier. La liste suivante, énumère les commandes de CodeView les plus communes :

```
A       address          Assembler
BC      bp_number        Permet d'effacer un point d'arrêt (breakpoint clear)
```

```
BD      bp_number          Désactive les points d'arrêt (breakpoint disable)
BE      bp_number          Active les points d'arrêt (breakpoint enable)
BL                         Liste des points d'arrêt (breakpoint list)
BP      address            Place un point d'arrêt (breakpoint set)
D       range              Affiche les valeurs de l'adresse spécifiée (dump memory)
E                          Met en marche l'exécution (animate execution)
Ex      address            Entre une commande (x = "", b, w, d, etc.)
G       {address}          Aller (go) (l'adresse est optionnelle)
H                          Aide (help)
I       port               Entre des données depuis un port d'E/S (input)
L                          Recommence le programme dès le début
MC      range address      Compare deux blocs de mémoire (memory compare)
MF      range data_val(s)  Remplit la mémoire avec la valeur spécifiée (memory fill)
MM      range address      Copie un bloc de mémoire (memory move)
MS      range data_val(s)  Recherche un ensemble de valeurs dans une plage (memory search)
N       value10            Change la base par défaut
O       port value         Exporter une valeur de sortie sur un port de sortie (output)
P                          Fait la trace du programme (program step)
Q                          Permet de quitter le programme (quit)
R                          Affiche le contenu des registres (register)
Rxx     value              Modifie le contenu des registres selon la valeur xx
T                          Trace
U       address            Désassemble l'instruction à l'adresse spécifiée (unassemble)
```

Dans ce chapitre, nous prendrons en compte principalement les commandes qui permettent de gérer la mémoire. Les commandes d'exécution comme *breakpoint*, *trace*, et *go*, seront vues aux prochains chapitres. Mais si vous décidez de les apprendre tout de suite certaines d'elles, ce ne sera pas mauvais.

4.9.5.7.1 La commande Radix (N)

La première des commandes à apprendre est RADIX (sélection de base). Par défaut, CodeView fonctionne en décimal (base 10). Ceci est vraiment incommode pour des programmeurs en assembleur, donc, vous devez toujours exécuter cette commande à l'entrée de CodeView et mettre la base à 16 (système hexadécimal). Pour ce faire, utilisez :

```
N 16
```

4.9.5.7.2 La Commande Assemble

Cette commande est similaire à celle de SIMx86. La commande utilise la syntaxe :

```
A adresse
```

où *adresse* est l'adresse de départ des instructions machine. Ce qui peut être soit une adresse segmentée complète (ssss:oooo où s est le segment et o l'offset), ou tout simplement un offset direct. CodeView utilise CS comme segment de code par défaut.

Après que vous appuyez sur Entrée, CodeView vous demandera d'entrer une séquence d'instructions machine. La touche Entrée elle-même, met fin à l'entrée des instructions. La figure 4.24 montre un exemple de cette commande en action.

Cette commande est l'une des seules disponibles *uniquement* dans la fenêtre des commandes. Certainement, Microsoft n'a pas prévu que les programmeurs entrent du code assembleur en mémoire en utilisant CodeView. Ce qui n'est pas une supposition déraisonnée, étant donné que CodeView n'est rien d'autre qu'un débogueur de haut niveau.

En général, cette commande est utile seulement pour apporter de petites modifications aux programmes, mais ce n'est pas un remplaçant de MASM 6.x. Toutes les modifications que vous apporterez au programme de cette manière n'apparaîtront pas sur votre fichier source. C'est très facile de corriger une bogue dans CodeView et oublier par la suite d'apporter les modifications appropriées au fichier source original.

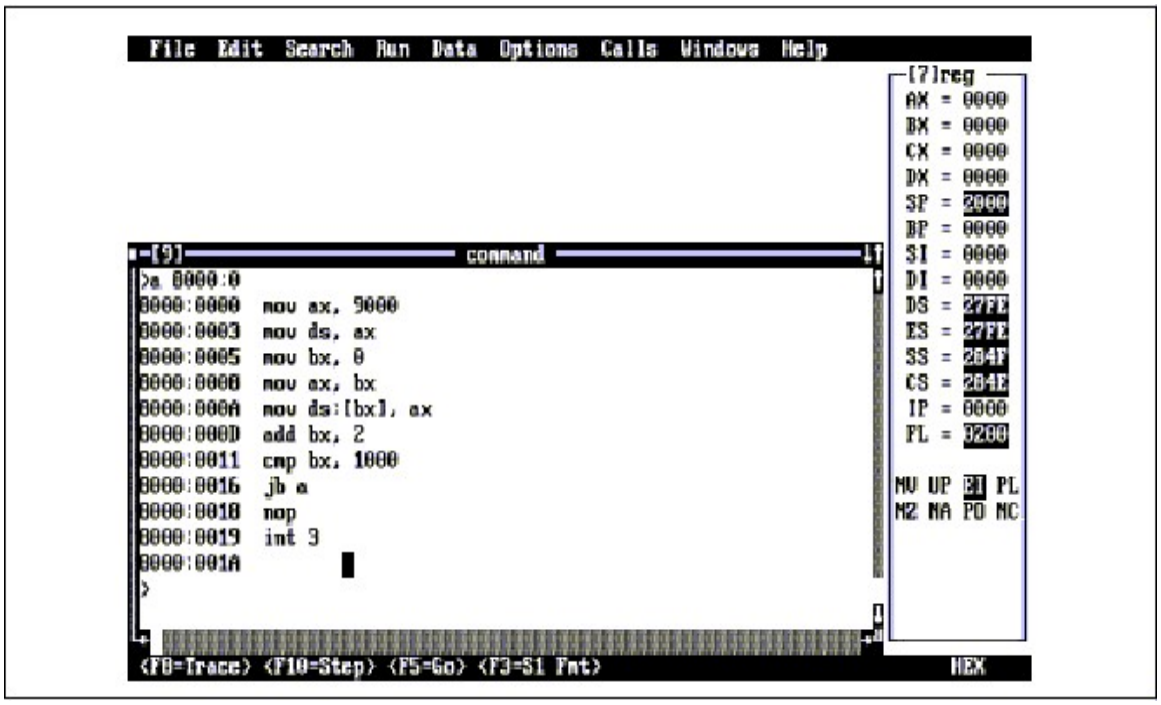

Figure 4.24 La commande Assemble

4.9.5.7.3 La commande Compare Memory

Cette commande fait la comparaison des octets appartenant à un bloc de mémoire avec les octets qui se trouvent dans un autre bloc de mémoire. Et elle signale toute différence entre les deux plages d'octets. Ceci est utile, par exemple, pour voir si un programme a initialisé deux tableaux de façon identique ou pour comparer deux longues chaînes de caractères. Cette commande prend la forme suivante :

```
MC adresse_début adresse_fin           adresse_deuxième_bloc
MC adresse_début L longueur_du_bloc    adresse_deuxième_bloc
```

La première version compare les octets spécifiés à partir de l'adresse de début jusqu'à l'adresse de fin de la plage avec les données qui commencent à partir de l'adresse du second bloc. La deuxième forme vous permet de spécifier la taille des blocs au lieu de spécifier l'adresse de fin du premier bloc. Si CodeView détecte une différence entre les deux rangées d'octets, il affiche ces différences, ainsi que leurs adresses. L'exemple qui suit, liste un exemple de commandes correctes :

```
MC 8000:0       8000:100        9000:80
MC 8000:100     L 20            9000:0
MC 0            100             200
```

Le premier bloc ci-dessus compare le bloc d'octets à partir de l'adresse 8000:0 jusqu'à 8000:100 avec le bloc d'une taille semblable qui commence à l'adresse 9000:80 (c'est-à-dire la plage 9000:80-9000:180).

La seconde commande montre l'utilisation de l'option "L" qui spécifie une longueur au lieu d'une adresse de fin. Dans cet exemple, CodeView comparera les valeurs de la plage 8000:0-8000:1F (20h/32 octets), avec les données qui commencent à l'adresse 9000:0 pour une même longueur.

Et finalement, le troisième exemple montre l'utilisation des simples décalages au lieu d'adresses complètes de type segment:offset. Dans ce cas, CodeView utilise par défaut le segment de données (DS). Notez cependant que si vous fournissez une adresse de début et une de fin, *elles doivent se trouver dans le même segment*. Vous ne pouvez pas spécifier une adresse de début dans un segment et une adresse de fin dans un autre, dans aucun des trois exemples ci-dessus.

Si les deux blocs sont identiques, CodeView vous invite immédiatement à entrer d'autres plages, sans rien afficher dans la fenêtre des commandes. En cas de différences, cependant, CodeView les affiche (en incluant leurs adresses) dans la fenêtre des commandes.

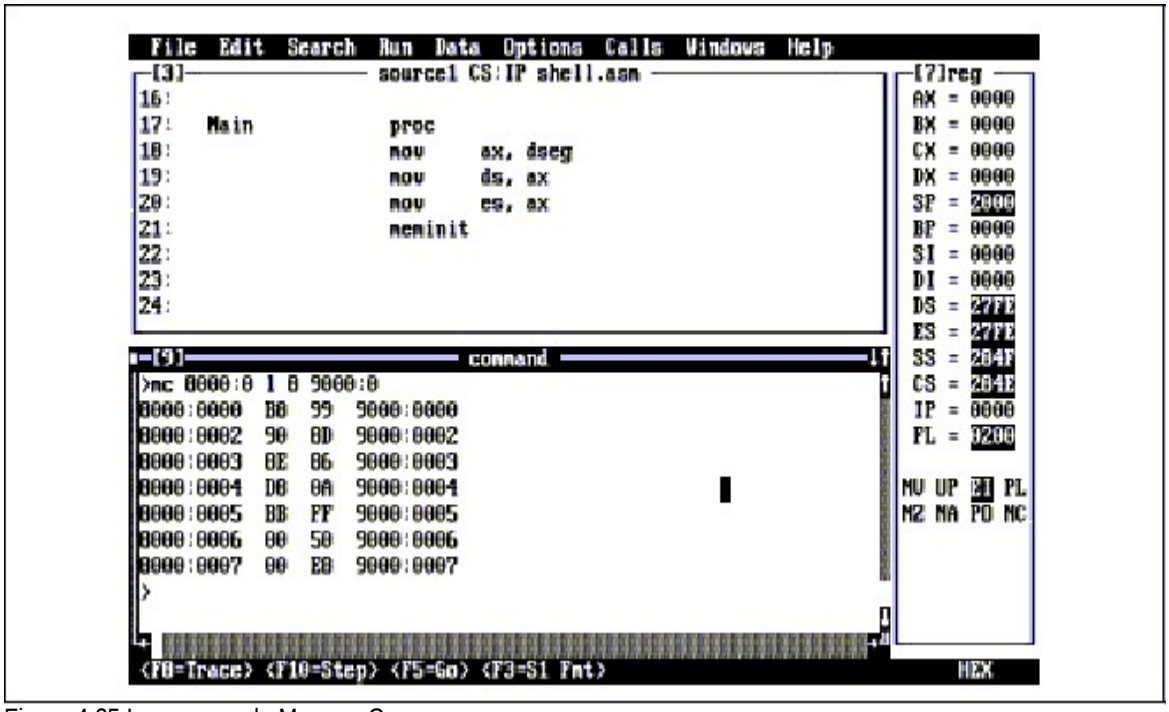

Figure 4.25 La commande Memory Compare

Dans l'exemple de la figure 4.25, les emplacements 8000:0 à 8000:200 ont été d'abord initialisés à zéro. Ensuite, les plages 8000:10 à 8000:1E ont été initialisées avec les valeurs successives 1, 2, 3, ..., 0Fh. Finalement, la commande a comparé les octets des plages 8000:0-8000:FF avec le bloc d'octets à partir de 8000:100. Et puisque les contenus respectifs étaient différents, CodeView a affiché les différences avec leurs adresses.

4.9.5.7.4 La commande Memory Dump

Cette commande vous permet d'afficher les valeurs des emplacements de mémoire sélectionnés. La fenêtre de la mémoire vous permet aussi de voir (et de modifier) la mémoire. Cependant, Dump est parfois plus pratique, spécialement pour voir le contenu de blocs de petite taille.

La commande Dump prend différentes formes, selon le type des valeurs que vous voulez visualiser à l'écran. Généralement, Dump peut être exécutée sous l'une des deux versions suivantes :

```
D adresse_début          adresse_fin
D adresse_début          L longueur
```

Par défaut, Dump affiche des valeurs hexadécimales de 16 bits, ainsi que les valeurs ASCII correspondantes pour chaque ligne (exactement comme ferait l'écran de la mémoire).

Il y a des variantes additionnelles à cette commande vous permettant de spécifier le format d'affichage des données. Cependant le format exact de ces versions change avec chaque nouvelle version de CodeView. Par exemple, dans CodeView 4.10, vous pouvez utiliser des commandes comme :

```
DA      plage_d_adressage       Affiche des caractères ASCII
DB      plage_d_adressage       Affiche des octets hexadécimaux ou ASCII (défaut)
DI      plage_d_adressage       Affiche des mots en format entier
```

```
DUI     plage_d_adressage       Affiche des mots en format entier non signé
DIX     plage_d_adressage       Affiche des valeurs de 16 bits en hexadécimal
DL      plage_d_adressage       Affiche des entiers de 32 bits
DLU     plage_d_adressage       Affiche des entiers de 32 bits non signés
DLX     plage_d_adressage       Affiche des valeurs de 32 bits hexadécimales
DR      plage_d_adressage       Affiche des valeurs réelles de 32 bits
DRL     plage_d_adressage       Affiche des valeurs réelles de 64 bits
DRT     plage_d_adressage       Affiche des valeurs réelles de 80 bits
```

Vous devriez probablement vérifier l'aide associée à votre version de CodeView pour vous assurer du format exact des commandes dump. Notez que certaines versions de CodeView vous permettent d'utiliser MDxx pour cette commande.

Une fois que vous avez exécuté une des commandes qu'on vient de voir, le nom de commande "D" affiche les données dans le nouveau format. Alors que la commande "DB" fait passer de nouveau à l'affichage octet/ASCII. La figure 4.26 en donne un exemple.

Si vous entrez une commande Dump sans adresse, CodeView affichera les données qui suivent immédiatement la dernière commande Dump. C'est parfois utile pour consulter la mémoire.

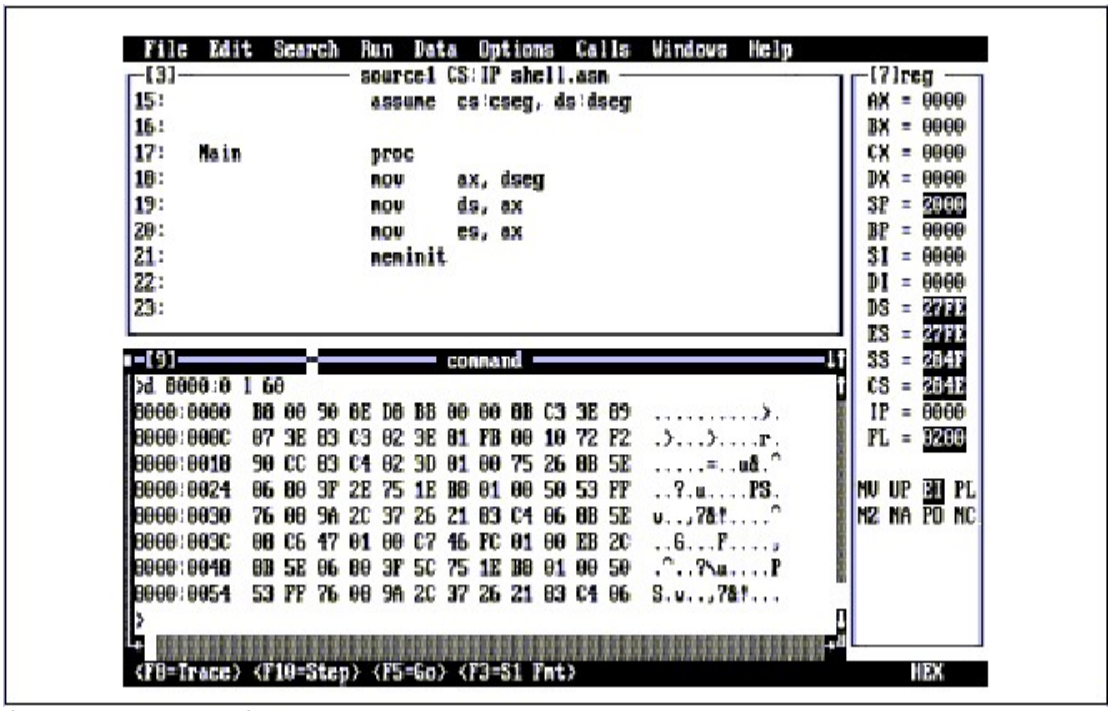

Figure 4.26 La commande Memory Dump

4.9.5.7.5 La commande Enter

La fenêtre de la mémoire de CodeView vous permet facilement d'afficher et de modifier le contenu des adresses. Mais on peut effectuer la même chose également à partir de la fenêtre des commandes ; deux commandes sont nécessaires pour le faire : d'abord Dump pour afficher les valeurs qui nous intéressent et ensuite Enter pour les modifier. Pour la plupart des tâches de modification de la mémoire, vous trouverez probablement la fenêtre de la mémoire plus facile à utiliser. Néanmoins, pour certaines tâches, c'est plus simple d'utiliser la fenêtre des commandes, plutôt que la fenêtre de la mémoire.

Comme la commande Dump, Enter vous permet aussi d'entrer des données sous différents formats. Les commandes correspondantes sont :

```
EA-             Entre les données en format ASCII
EB-             Entre des octets en format hexadécimal
```

```
ED-             Entre des doubles-mots en format hexadécimal
EI-             Entre des entiers de 16 bits en format décimal (signé)
EIU-            Entre des entiers de 16 bits en format décimal (non signé)
EIX-            Entre des entiers de 16 bits en format hexadécimal
EL-             Entre des entiers de 32 bits en format décimal (signé)
ELU-            Entre des entiers de 32 bits en format décimal (non signé)
ELX-            Entre des entiers de 32 bits en format hexadécimal
ER-             Entre des valeurs en virgule flottante de 32 bits
ERL-            Entre des valeurs en virgule flottante de 64 bits
ERT-            Entre des valeurs en virgule flottante de 80 bits
```

Comme pour la commande Dump, cette commande change régulièrement d'une version à l'autre. Donc, assurez-vous d'utiliser l'aide pour son format exact. Dans CodeView, MExx est synonyme de Exx.

La commande Enter prend deux formes possibles :

```
Ex adresse_début
Ex adresse_début liste_de_valeurs
```

La première des deux formes est la version interactive de la commande Enter. Une fois validée, elle affiche l'adresse de début et la donnée qui se trouve à cette adresse, puis elle vous invite à entrer une nouvelle valeur pour cet emplacement. Tapez-la, suivie d'un espace et CodeView vous invitera à faire de même pour l'emplacement suivant ; alors qu'en tapant sur espace, la valeur de l'adresse courante sera sautée, et la commande passera à l'emplacement suivant ; taper sur Entrée ou une valeur suivie de la touche Entrée, met fin à l'exécution du mode interactif. Notez que la commande EA ne fonctionnera pas en mode interactif. Et il en va de même pour la commande EB.

La seconde forme de la commande Enter vous permet, avec une seule opération d'entrée, d'entrer une séquence de valeurs. Dans cette version de la commande vous n'avez qu'à placer, après l'adresse de début, la liste des valeurs que vous voulez garder à partir de cette adresse. CodeView les placera automatiquement à des emplacements de mémoire successifs commençant à l'adresse de début. Avec cette méthode vous pouvez entrer des caractères ASCII en les entourant par des guillemets. La figure 4.27 montre l'usage de la commande Enter.

Il y a encore deux points à considérer sur la commande Enter. D'abord vous ne pouvez pas utiliser "E" sans aucun autre suffixe qui lui fait suite. Contrairement à ce qui arrive avec Dump, "E" ne veut pas dire "commencer à entrer la commande après la dernière adresse". Au contraire, "E" est une commande totalement différente (E = *Execution*). L'autre chose à considérer est que le mode d'affichage courant (ASCII, byte, word, double word, etc.), et le mode d'entrée courant ne sont pas indépendants. Changer le mode d'affichage par défaut pour le mode word, par exemple, a pour effet de changer aussi le mode d'entrée pour des données word.

4.9.5.7.6 La commande Memory Fill

Les commandes *Enter* et *Memory* vous permettent de changer facilement la valeur des emplacements de mémoire individuels ou d'attribuer à une plage de mémoire plusieurs valeurs différentes. Si vous voulez vider un tableau ou initialiser une plage d'adresses de mémoire pour qu'elles contiennent la même valeur, la commande Memory Fill fournit une meilleure alternative.

Memory Fill fonctionne avec la syntaxe suivante :

```
MF adresse_début adresse_fin valeurs
MF adresse_début L longueur_bloc valeurs
```

Cette commande remplit les emplacements de mémoire à partir d'une certaine adresse de début, jusqu'à une certaine adresse de fin avec les valeurs d'un octet spécifiées dans la liste des valeurs. Alors que la deuxième version vous permet de spécifier une longueur de plage, au lieu d'une adresse de fin.

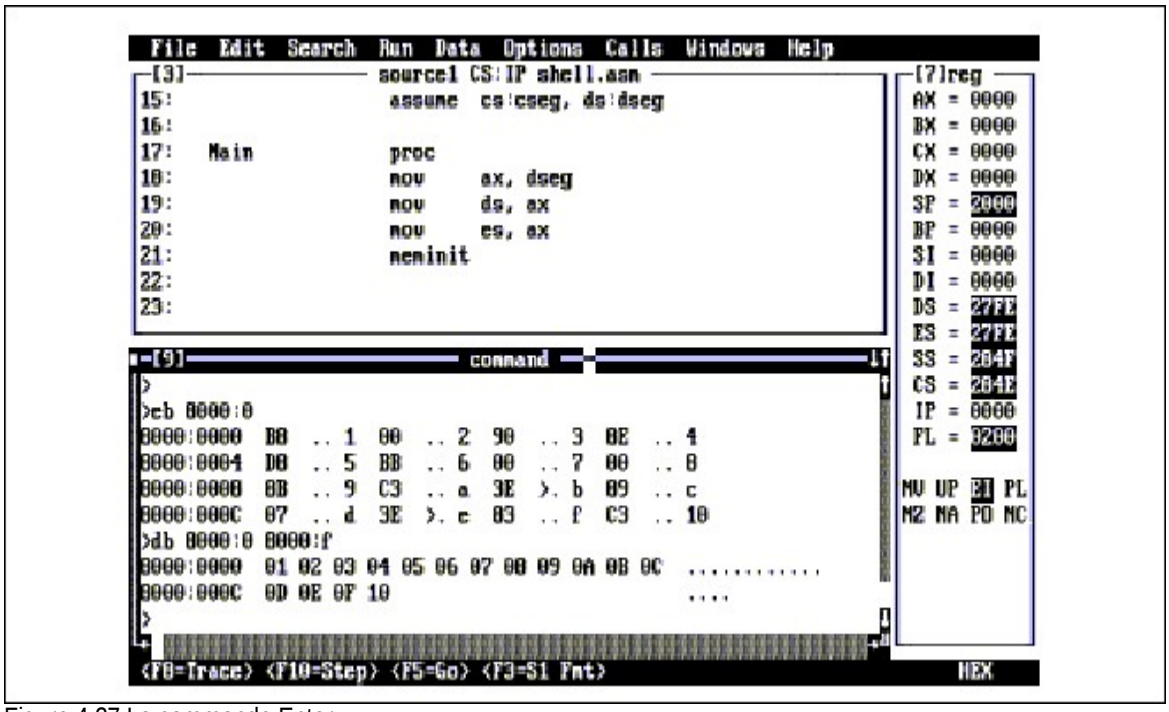

Figure 4.27 La commande Enter

La liste des valeurs peut être une valeur unique ou bien une suite de nombres. Si *valeurs* contient une seule valeur d'un octet, alors la commande initialise tous les octets de la plage avec cette valeur. Si, au contraire, il s'agit d'une liste d'octets, alors la commande répétera la séquence des octets spécifiés jusqu'à l'adresse finale de la plage. Par exemple, la commande qui suit stocke 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5... dans les 256 octets à partir de l'adresse 8000:0 :

```
MF 8000:0 L 100 1 2 3 4 5
```[20]

Malheureusement, la commande *Fill Memory* fonctionne seulement avec des octets (ou des chaînes ASCII). Néanmoins, on peut simuler des mots, des doubles-mots, etc. en les décomposant en leurs éléments octets. N'oubliez pas cependant que l'octet le moins significatif vient en premier.

4.9.5.7.7 La commande Move Memory

Cette commande copie les données d'un bloc de mémoire à un autre. Ce qui permet de copier des données d'un tableau à un autre, déplacer du code d'un côté à l'autre de la mémoire, réinitialiser un groupe de variables à partir d'un bloc enregistré et ainsi de suite. La syntaxe pour cette commande est la suivante :

```
MM adresse_début adresse_fin adresse_destination
MM adresse_début L longueur_bloc adresse_destination
```

Si les blocs de source et de destination se chevauchent, CodeView le détecte et poursuit l'opération correctement.

4.9.5.7.8 La commande Input

Cette commande permet de lire des données à partir d'un des 65536 ports d'entrée 80x86. La syntaxe est :

[20]Noter que dans l'orignal il y avait F au lieu de MF. Présumant qu'il s'agit probablement d'une erreur, j'ai corrigé avec le nom de commande approprié, ndt.

```
I       adresse_port
```

où adresse_port est une valeur de 16 bits indiquant l'adresse du port E/S à lire. Cette commande lit l'octet dans ce port et l'affiche. Remarquez que ce n'est pas une bonne idée d'utiliser cette commande pour une adresse arbitraire. Certains périphériques activent des fonctions quand on lit dans un de leurs ports. En y lisant sans raison vous risquez de faire perdre des données aux périphériques ou bien tout simplement de les déranger.

Notez aussi que cette commande lit un octet seul à partir du port spécifié. Si vous voulez y lire des types de données plus larges, vous devrez exécuter deux ou plus commandes Input successives à l'adresse du port désirée et aux suivantes.

Cette commande semble corrompue dans certaines versions de CodeView (par exemple, la version 4.01).

4.9.5.7.9 La commande Output

Cette commande fait complément à la commande Input. Elle vous permet d'écrire une valeur dans un port et utilise la syntaxe :

```
O       adresse_port    valeur_sortie
```

où *valeur_sortie* est une valeur d'un octet que CodeView écrira dans le port de sortie fourni par *adresse_port*.

Notez que CodeView utilise la commande "O" également pour activer des options. S'il ne reconnaît pas une adresse de port valide dans la première opérande, il présumera qu'il s'agit de la commande Option. Si la commande Output donne l'impression de ne pas fonctionner correctement, vous êtes passé probablement en dehors du mode assembleur (en plus de l'assembleur, CodeView supporte aussi BASIC, Pascal, C et FORTRAN), et l'adresse du port n'est pas une valeur numérique valide dans le nouveau mode. Assurez-vous d'utiliser la commande N 16 pour établir la base hexadécimale comme mode numérique par défaut avant d'utiliser la commande Output !

4.9.5.7.10 La commande Quit

Presser Q ("Quit") pour met fin à la session de débogage courante et retourner le contrôle à MS-DOS. Vous pouvez également quitter CodeView en sélectionnant l'élément Exit du menu File.

4.9.5.7.11 La commande Register

Cette commande vous permet de voir et de modifier les valeurs des registres. Pour observer la valeur courante de ces derniers, il faudra utiliser la commande suivante :

```
R
```

Cette commande affiche les registres et désassemble l'instruction de l'adresse CS:IP.

Vous pouvez aussi modifier la valeur d'un registre spécifié utilisant la commande dans cette forme :

```
Rxx
-ou-
Rxx = valeur
```

où xx représente un des registres 80x86 AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, ES, SS, IP et FL. La première version ("Rxx") affiche le registre spécifié et vous invite ensuite à entrer une nouvelle valeur. La seconde forme de cette commande modifie immédiatement le registre avec la valeur fournie (voir Figure 4.28).

4.9.5.7.12 La commande Unassemble

Cette commande provoque le désassemblage d'une séquence d'instructions à l'adresse que vous spécifierez, en convertissant le code machine binaire en instructions machine lisibles (ou presque). La version de base de la commande utilise la syntaxe suivante :

```
U       adresse
```

Notez que *vous devez avoir une fenêtre source ouverte pour que cette instruction fonctionne correctement !*

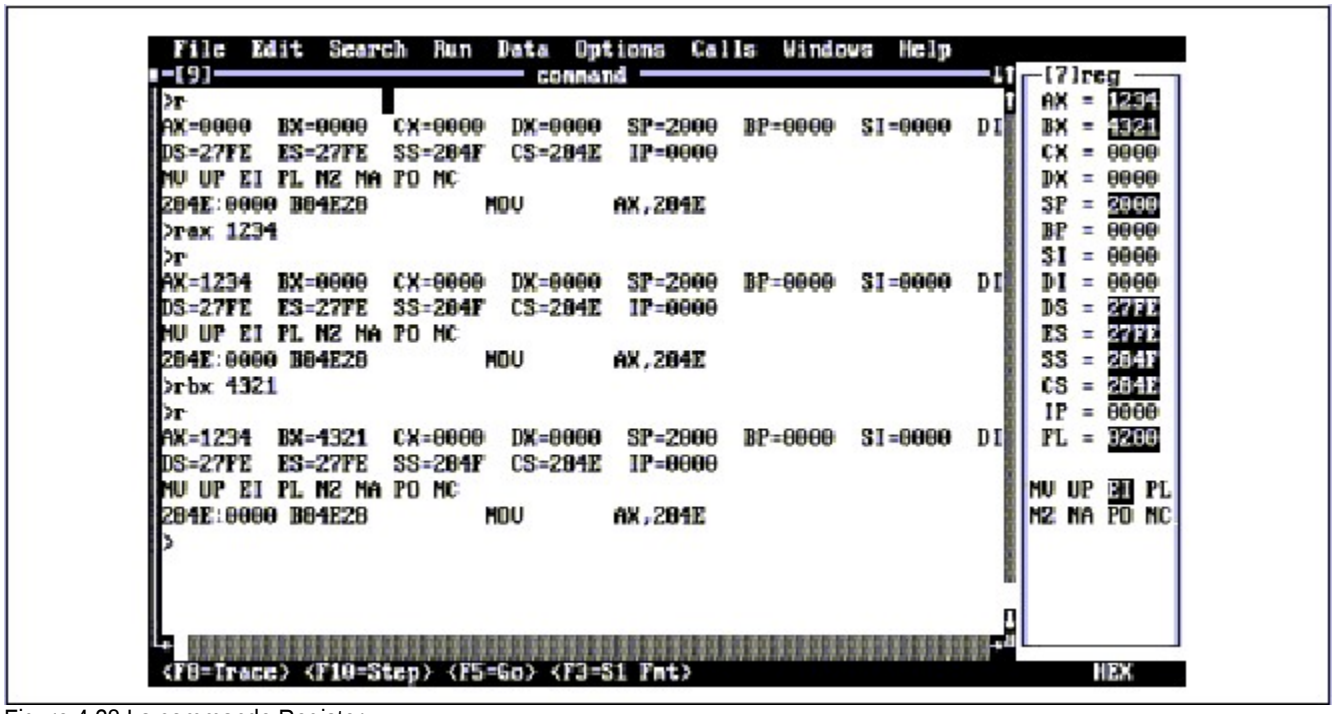

Figure 4.28 La commande Register

En général, la commande Unassemble n'est pas très utilisée parce que la fenêtre source vou permet de voir votre programme au niveau source (plutôt qu'au niveau de langage machine désassemblé). Cependant, cette commande reste d'une grande utilité pour désassembler du code en mémoire comme ceux du BIOS, DOS, TSR et d'autres programmes.

4.9.5.8 Touches de fonction de CodeView

CodeView se sert de plusieurs touches de fonction pour accélérer certaines tâches d'usage fréquent. La table suivante donne une brève description de l'usage de chacune de ces touches.

Touche de fonction :	Seule :	Avec Shift :	Avec Ctrl :	Avec Alt :
F1	Aide	Contenu de l'aide	Aide suivante	Aide précédente
F2	Fenêtre des registres			
F3	Mode fenêtre source	Mode fenêtre mémoire		
F4	Ecran de sortie		Fermeture fenêtre	
F5	Exécuter			
F6	Passer d'une fenêtre à l'autre	Fenêtre précédente		
F7	Exécute jusqu'au curseur			
F8	Trace	Historique précédent	Taille fenêtre	
F9	Point d'arrêt			

F10	Etape par étape. Exécute les appels.	Historique suivant	Maximiser la fenêtre	

La touche de fonction F3 mérite une mention spéciale. Elle permet de passer entre les modes machine langage (lire : langage machine désassemblé), *mixed* et *machine langage source*. En mode source (en supposant que vous avez assemblé votre code avec les options correctes), la fenêtre source window affiche votre code source courant. En mode mixed, CodeView affiche chaque ligne avec du code source suivi par le code machine qui a été généré pour cette ligne. Ce mode est surtout destiné aux utilisateurs des langages de haut niveau, mais il maintient son utilité même pour les utilisateurs du langage assembleur, comme on verra quand on étudiera les macros. En mode machine, CodeView ignore votre code source et désassemble simplement les opcodes binaires dans la mémoire. Ce mode est utile si vous soupçonnez des bogues dans MASM (elles existent) et vous n'êtes pas certain si MASM est en train d'assembler correctement votre code.

4.9.5.9 Quelques commentaires sur les adresses de CodeView

Les exemples donnés pour les adresses dans la section précédente sont quelque peu déroutants. Maintenant, vous aurez sans doute la certitude que les adresses doivent être entrées en forme hexadécimale, c'est-à-dire, ssss:oooo ou encore oooo. En réalité on peut spécifier les adresses de mémoire de plusieurs façons différentes. Par exemple, si vous avez une variable dans votre programme qui s'appelle maVar, vous pouvez utiliser une commande comme :

```
D maVar
```

pour afficher la valeur de cette variable[21]. Vous n'avez pas besoin de connaître l'adresse, pas plus que le segment de cette variable. Un autre moyen de spécifier une adresse est à l'aide du jeu des registres. Par exemple, si ES:BX pointe sur un bloc de mémoire que vous voulez afficher, vous pouvez utiliser la commande suivante :

```
D ES:BX
```

et CodeView utilisera les valeurs courantes des registres es et bx en tant qu'adresses du bloc de mémoire à afficher. Il n'y a rien de magique dans l'usage des registres. Vous pouvez vous en servir simplement comme toute autre composante d'adresse. Dans l'exemple ci-dessus es contient la valeur du segment et bx la valeur de l'offset - ce qui est très typique dans un programme en assembleur. Cependant, CodeView n'impose pas des combinaisons valides du jeu 80x86. Vous pourriez même faire recours à une instruction de la sorte :

```
D CX:AX
```

L'usage des registres n'est pas limité à des adresses source spécifiques. Vous pouvez aussi spécifier des adresses de destination et même des longueurs :

```
D CX:AX L BX ES:DI
```

Naturellement, on peut également mêlanger les registres et les adresses numériques dans la même commande :

```
D CX:AX L 100 8000:0
```

Et on pourrait enfin avoir recours à des expressions arithmétiques complexes pour spécifier une adresse de mémoire. En particulier, vous pouvez utiliser l'opérateur d'addition arithmétique pour calculer la somme de diverses composantes d'une adresse. Ce qui fonctionne très bien quand on veut simuler des modes d'adressage. Par exemple, si vous voulez voir quel octet se trouve à l'adresse 1000[bx], vous pouvez utiliser la commande :

```
D BX+1000 L 1
```

Pour simuler le mode d'adressage [BX][SI] et voir dans le mot de cette adresse, vous pouvez utiliser la commande :

```
DIX BX+SI L 1
```

[21]Ceci demande un assemblage assez particulier, mais on verra comment.

Les exemples présentés dans cette section utilisent tous la commande Dump, mais vous pouvez vous servir de cette technique avec toutes les commandes de CodeView. Pour plus d'informations concernant ce qui constitue une adresse valide pour CodeView et également pour une explication complète des expressions permises, consultez le système d'aide en ligne de CodeView.

4.9.5.10 Un mot final sur CodeView

On n'a pas épuisé les sujets à traiter sur CodeView. Spécialement on n'a pas mentionné les commandes d'exécution, de trace et de point d'arrêt, cruciales pourtant pour un programme de débogage. On reviendra sur ces sujets dans des chapitres ultérieurs. Déjà une matière considérable a été couverte ici, certainement assez pour faire face aux expériences proposées dans les exercices de laboratoire. Les nouvelles commandes seront traitées au fur et à mesure de leur besoin.

Il y a sans doute deux autres sources d'informations additionnelles sur CodeView - la section dédiée à CodeView du livre *Microsoft Macro Assembler Programmer's Guide* et l'aide en ligne disponible à l'intérieur de CodeView. Spécialement cette dernière source est très utile pour avoir une idée de comment une commande spécifique fonctionnera à l'intérieur de CodeView.

4.9.6 Exercices de laboratoire

Le sous-répertoire du chapitre 4 contient un fichier échantillon nommé EX4_1.ASM. Assemblez ce programme avec MASM (n'utilisez pas l'option /Zi pour ce programme). **Pour votre rapport de laboratoire :** incluez une version imprimée de ce programme. Décrivez qu'est-ce qu'il fait. Exécutez le programme et imprimez les sorties pour les inclure dans votre rapport.

Quand vous assemblez un programme, MASM écrit par défaut un octet dans le fichier .exe pour chaque octet d'instruction ou variable, même si les données ne sont pas utilisées. Si vous déclarez de gros tableaux, le fichier .exe que ML produit sera plus gros. Notez la taille du fichier .exe que vous venez juste de créer. Maintenant, assemblez le programme de nouveau en utilisant la commande suivante :

```
ml EX4_1.asm /link /exepack
```

ML passe l'option /link /exepack à l'éditeur de liens. L'option exepack indique à l'éditeur de construire le fichier .exe en éliminant les informations redondantes (spécialement les données non initialisées). Ceci engendre souvent un fichier .exe beaucoup plus petit. **Pour votre rapport de laboratoire :** après avoir assemblé le fichier en utilisant la commande ci-dessus, notez la taille du fichier .exe résultant. Comparez les deux tailles et commentez leurs différences dans votre rapport de laboratoire.

Notez que l'option exepack permet d'économiser uniquement l'espace disque. Il ne réduit aucunement l'usage que le programme fait de la mémoire. De plus, vous ne pouvez pas charger dans CodeView les programmes assemblés avec cette option. Par conséquent, vous ne devriez pas utiliser l'option exepack durant le développement et le test d'un programme, mais seulement une fois que toutes les bogues ont été éliminées et que le programme est prêt à être livré.

En utilisant l'éditeur de votre choix, éditez le fichier x86.ASM. Lisez les commentaires au début du programme qui explique comment écrire des programmes x86 qui s'assemblent et puissent tourner sur un CPU 80x86. **Pour votre rapport de laboratoire :** décrivez les restrictions auxquelles l'écriture des programmes x86 est soumise.

Le fichier source EX4_2.ASM est une copie du fichier x86.ASM, avec quelques commentaires additionnels dans le programme principal décrivant un ensemble de procédures que vous devriez suivre. Chargez ce fichier dans l'éditeur de votre choix et lisez les instructions du programme principal. Suivez-les pour produire un programme. Assemblez ce programme en utilisant ML et exécutez le fichier binaire EX4_2.EXE qui en résulte. **Pour votre rapport de laboratoire :** incluez une version imprimée de votre programme résultant et faites de même avec les sorties de celui-ci.

Essayez de charger EX4_2.EXE dans CodeView en utilisant la commande DOS suivante :

```
cv EX4_2
```

Quand CodeView s'exécutera, vous remarquerez qu'il affiche un message dans la fenêtre des commandes disant : "no CodeView information for EX4_2.EXE" ("il n'y a pas d'informations pour CodeView dans EX4_2.EXE"). Observez le code dans la fenêtre source. Cherchez les instructions que vous avez placées dans le programme principal. **Pour votre rapport de laboratoire :** comparez le listing qui apparaît dans CodeView avec celui qui est produit dans l'écran d'émulation du programme SIMx86.

Maintenant assemblez de nouveau le fichier EX4_2.asm et chargez-le dans CodeView via les commandes DOS suivantes :

```
ml /Zi Ex4_2.asm
   cv EX4_2
```

Pour votre rapport de laboratoire : décrivez la différence dans la fenêtre source de CodeView quand vous utilisez l'option /Zi et quand vous ne l'utilisez pas.

4.10 Projets de programmation

Note : vous devez écrire ces programmes en langage assembleur 80x86 en utilisant une copie du fichier X86.ASM comme point de départ de vos *listings*. Le jeu d'instructions 80x86 est pratiquement un surensemble du jeu x86. Par conséquent, vous pouvez utiliser beaucoup des instructions que vous avez apprises au cours du chapitre précédent. Lisez les commentaites au début du fichier x86.ASM pour plus de détails. **Notez en particulier que vous ne pouvez pas utiliser l'étiquette "C" dans votre programme, parce que "C" est un mot réservé dans MASM.** En soumettant une copie de vos résultats, incluez un document de spécification, un plan de test, un listing et un exemple de sortie.

1) Les projets suivants sont des modifications des exercices de programmation du chapitre précédent. Convertissez ces programmes x86 en version 80x86 respective.

1a. Le jeu d'instructions x86 n'inclut pas d'instruction de multiplication. Ecrivez un court programme qui lit deux valeurs de l'utilisateur et affiche leur produit (astuce : souvenez-vous qu'une multiplication est juste une addition répétée).

1b. Écrivez un programme qui lit trois valeurs de l'utilisateur : une adresse à placer dans BX, un compteur dans CX et une valeur dans AX. Ce programme doit écrire CX copies de AX dans les mots successifs de la mémoire à partir de l'adresse située dans BX.

1c. Écrivez la fonction logique générique du chapitre deux pour un processeur x86. Truc : *add ax, ax* fait un décalage à gauche de la valeur dans AX. Vous pouvez tester pour voir si le bit fort est activé pour vérifier si la valeur de AX est plus grande que 8000h.

1d. Écrivez un programme lisant un tableau de mots à partir de l'adresse 1000h et d'une longueur spécifiée dans la valeur contenue dans CX. Le programme doit localiser la valeur maximale dans ce tableau. Affichez la valeur après que celle-ci a été lue dans le tableau.

1e. Écrivez un programme qui effectue une opération de complément à deux sur un tableau commençant à l'adresse 1000h. CX devrait contenir le nombre de valeurs dans ce tableau. Basez-vous sur le fait que chaque élément est un entier de deux octets.

1f. Écrivez un programme simple qui trie les mots des emplacements 1000 à 10FF en ordre croissant. Vous pouvez utiliser un algorithme de tri par insertion. Le code Pascal pour un tel tri est :

```
for i := 0 to n - 1 do
        for j := i + 1 to n do
                if(memory[i] > memory[j]) then
                begin
                        temp := memory[i];
                  memory[i] := memory[j];
                  memory[j] := temp;
                end;
```

Pour le projet qui suit, ayez la liberté d'utiliser tout mode d'adressage 80x86 additionnel pouvant rendre le programme plus facile à écrire.

2) Écrivez un programme qui stocke les valeurs 0, 1, 2, 3, ..., dans des mots successifs du segment de données, à partir de l'adresse 3000h, jusqu'à l'adresse 3FFEh (la dernière valeur écrite sera 7FFh). Puis, stockez la valeur 3000h dans l'emplacement 1000h en mémoire. Ecrivez un ensemble d'instructions additionnant les 512 mots à partir de l'adresse trouvée à l'emplacement 1000h. Cette portion du programme ne peut pas présumer que l'adresse 1000h pointe sur 3000h. Finalement le programme affichera la somme avant de terminer son exécution.

4.11 Résumé

Ce chapitre s'est focalisé surtout sur l'organisation de la mémoire et les structures de données de la famille 80x86. Il ne constitue certainement pas un cours complet sur l'argument et ces concepts apparaîtront encore plus tard. Ici, on a simplement traité des types de données les plus primitifs et des types composés les plus simples. On a également vu comment les déclarer et les utiliser dans un programme. Beaucoup d'autres informations additionnelles sur la déclaration et l'usage des types primitifs pourra être trouvé dans "MASM : Directives et pseudo opcodes" au chapitre 8.

Les processeurs 8088, 8086, 80188, 80186 et 80286 partagent tous un jeu de registres commun ; ce jeu comprend les registres généraux, ax, bx, cx, dx, si, di, bp et sp, les registres de segment, cs, ds, es et ss et les deux registres spéciaux, ip et flags. Tous ces registres ont une taille de 16 bits. Ces processeurs ont aussi des versions de 8 bits des registres généraux : al, ah, bl, bh, cl, ch et dl, dh. Voir :

- "Registres généraux", au paragraphe 4.1.1
- "Les registres de segment", au paragraphe 4.1.2
- "Les registres 8086 dits « spéciaux »", au paragraphe 4.1.3

De plus, le 80286 supporte divers registres spéciaux de gestion de la mémoire, utiles dans des programmes pour les systèmes d'exploitation ou d'autres applications de niveau système. Voir :

- "Les registres du 80286", au paragraphe 4.1.4

A partir du processeur 80386, les registres généraux et certains des registres spéciaux ont été étendus à 32 bits. Deux registres de segment ont aussi été ajoutés. En plus de ces améliorations que tout programme peut exploiter, ces processeurs ont également ajouté divers registres de niveau système pour la gestion de la mémoire, le débogage et le test du processeur. Voir :

- "Les registres 80386/80486", au paragraphe 4.1.5

La famille 80x86 utilise un schéma puissant d'adressage de la mémoire, dit *adressage segmenté*, qui fournit une simulation d'un adressage à deux dimensions. Ce qui permet de relier logiquement des blocs de données à des segments. Le format exact de ces segments varie selon que le CPU fonctionne en mode *réel* ou en mode *protégé*. La plupart des programmes DOS fonctionnent en mode réel. Quand on travaille dans ce mode, il est très facile de convertir des adresses *logiques* (segmentées) en adresses *physiques*. En mode protégé, cette conversion est considérablement plus difficile. Voir :

- "Segments 80x86", au paragraphe 4.3

Puisqu'en mode réel, les adresses segmentées font en réalité référence aux adresses physiques, il est possible d'avoir diverses adresses segmentées représentant le même emplacement de mémoire. Une solution à ce problème est d'utiliser les adresses normalisées. Si deux adresses normalisées n'ont pas le même modèle binaire, alors elles pointent sur des adresses différentes. Les pointeurs normalisés sont très utiles pour comparer des pointeurs en mode réel. Voir :

- "Adresses normalisées", au paragraphe 4.4

A l'exception de deux instructions, les processeurs 80x86 ne fonctionnent pas avec des adresses de 32 bits pleines. C'est pourquoi ils utilisent les *registres de segment* pour garder les valeurs de segment par défaut. Ceci a permis aux concepteurs Intel de construire un jeu d'instructions beaucoup plus petit, étant donné que les adresses ont une longueur de seulement de 16 bits (en considérant seulement la portion de l'offset), au lieu de 32 bits. Le 80286 et les processeurs antérieurs fournissent quatre registres de segment : cs, ds, es et ss ; alors qu'à partir du 80386, les registres de segment sont six : cs, ds, es, fs, gs et ss. Voir :

- "Les registres de segment", au paragraphe 4.5

La famille 80x86 offre diverses possibilités d'accéder à des variables, à des constantes et à d'autres genres de données. Le nom du mécanisme vous permettant d'accéder a la mémoire est *mode d'adressage*. Les processeurs 8088, 8086 et 80286 fournissent un large éventail de modes d'adressage. Voir :

- "Les modes d'adressage 80x86", au paragraphe 4.6
- "Les modes d'adressage des registres 8086", au paragraphe 4.6.1
- "Les modes d'adressage de la mémoire 8086", au paragraphe 4.6.2

Les processeurs 80386 et ultérieurs fournissent un jeu étendu des modes d'adressage de la mémoire et des registres. Voir :

- "Les modes d'adressage des registres sur le 80386", au paragraphe 4.6.3
- "Les modes d'adressage de la mémoire sur le 80386", au paragraphe 4.6.4

L'instruction la plus commune de tout le jeu d'instructions 80x86 est *mov*. Elle supporte presque tous les modes d'adressage disponibles. Par conséquent, il vaut la peine d'étudier les encodages et son fonctionnement correct. Voir :

- "L'instruction mov 80x86" au paragraphe 4.7

Cette instruction prend diverses formes génériques, vous permettant de transporter des données entre les registres et d'autres emplacements. Les emplacements source/destination possibles comprennent : (1) d'autres registres, (2) des adresses de mémoire (en utilisant n'importe quel mode d'adressage), (3) des constantes (via le mode d'adressage immédiat) et (4) des registres de segment.

L'instruction mov vous permet de transférer des données entre deux emplacements (quoique vous ne pouvez pas transporter des données entre deux adresses de mémoire avec une seule instruction, à ce propos voir la discussion sur l'octet mod-reg-r/m).

4.12 Questions

1. Même si sur des machines 80x86 on utilise toujours des adresses segmentées, une instruction comme "mov AX, I" a seulement un offset de 16 bits encodé dans l'opcode. Expliquer.
2. L'adressage segmenté est mieux décrit comme un *schéma d'adressage à deux dimensions*. Expliquer.
3. Convertir les adresses logiques suivantes en adresses physiques. Considérez toutes les valeurs comme étant hexadécimales et en mode réel :

   a) 1000:1000 b) 1234:5678 c) 0:1000 d) 100:9000

   e) FF00:1000 f) 800:8000 g) 8000:800 h) 234:9843

   i) 1111:FFFF j) FFFF:10
4. Mettez les adresses ci-dessus en forme normalisée.
5. Énumerez tous les modes d'adressage 8086.
6. Énumerez tous les modes d'adressage 80386 (et ultérieurs) qui ne sont pas disponibles sur le 8086 (utilisez les formes génériques d'utilisation, comme disp[reg], n'énumérez pas toutes les combinaisons possibles).
7. A côté du mode d'adressage mémoire, quels sont les deux autres modes d'adressage les plus importants sur le 8086 ?
8. Décrivez un usage commun de chacun de ces modes d'adressage :

   a) Registre b) Déplacement seul c) Immédiat

   d) Registre indirect e) Indexé f) Indexé scalaire

   g) Indexé basé avec déplacement h) Indexé basé

9. Étant donné le modèle binaire pour l'instruction MOV générique (voir "L'instruction MOV", au paragraphe 4.7), expliquer pourquoi les processeurs 80x86 ne supportent pas l'opération mov mem, mem.

10. Lesquelles des instructions MOV suivantes ne sont pas prévues par l'opcode générique de MOV ? Expliquer.

    a) mov ax, bx b) mov ax, 1234 c) mov ax, I

    d) mov ax, [bx] e) mov ax, ds f) mov [bx], 2

11. Supposez que la variable "I" se trouve à l'offset 20h du segment de données. Donnez l'encodage binaire des instructions ci-dessus.

12. Qu'est-ce qui détermine si le champ R/M spécifie un registre ou bien une opérande de mémoire ?

13. Quel champ dans l'octet REG-MOD-R/M détermine la taille du déplacement qui suit une instruction ? Quelles tailles de déplacement le 8086 supporte-t-il ?

14. Pourquoi le déplacement seul ne supporte pas plusieurs tailles de déplacement ?

15. Pourquoi n'interchangeriez-vous pas les instructions "mov ax, [bx]" et "mov ax, [ebx]" ?

16. Certaines instructions 80x86 prennent diverses formes. Par exemple, il y a deux versions différentes de l'instruction MOV qui chargent un registre avec une valeur immédiate. Expliquez pourquoi les concepteurs ont incorporé ces redondances dans le jeu d'instructions.

17. Pourquoi il n'existe pas de véritable mode d'adressage [bp] ?

18. Énumerez tous les registres de huit bits des 80x86.

19. Énumerez tous les registres généraux de 16 bits.

20. Énumerez tous les registres de segment (ceux qui sont disponibles sur tous les processeurs).

21. Décrivez la "fonction spéciale" de chaque registre général.

22. Énumerez tous les registres généraux de 32 bits des 80386/486/586.

23. Quelle est la relation entre tous les registres généraux de 8, 16 et 32 bits ?

24. Quelles valeurs apparaissent dans le registre FLAGS du 8086 ? Et du 80286 ?

25. Quels drapeaux (flags) constituent les codes de condition ?

26. Quels sont les registres de segment additionnels apparus avec le processeur 80386 ?