Initiation aux fonctionnalités des microprocesseurs, illustré par i80x86 et debug

Patrick Cegielski
cegielski@univ-paris12.fr

Mai 2006

Pour Irène et Marie

Legal Notice

Université Paris XII - IUT
Route forestière Hurtault
F-77300 Fontainebleau
cegielski@univ-paris12.fr

Le constituant essentiel d'un micro-ordinateur est son microprocesseur, qui est le plus important de ses circuits intégrés. Nous pouvons aborder l'étude du microprocesseur par son *aspect électronique* (on parle plutôt de sa *description matérielle* ou de sa *description physique*), en voyant comment il est conçu et comment il est construit. Mais ceci n'est pas l'approche la plus efficace pour une première vue sur le microprocesseur. En effet un microprocesseur est un circuit intégré d'une très grande complexité. La meilleure approche en est donc sa *description fonctionnelle*, en indiquant quelles sont ses fonctions (que l'on voit à travers son *jeu d'instructions*).

D'un point de vue physique, vu de l'extérieur un microprocesseur est une boîte (appelée *boitier*) d'où sort un certain nombres de *broches*, que l'on relie électriquement aux autres composants du micro-ordinateur. Il y a trois sortes de broches : celles en entrée uniquement, celles en sortie uniquement et celles en entrée-sortie. On applique ou non un courant à une broche en entrée ; on capte ou non un courant d'une broche en sortie ; on peut faire les deux, mais cela dépend du moment et on l'indique explicitement en appliquant ou non un courant à l'une des broches en entrée uniquement, pour le troisième type de broches. D'un point de vue physique, on pourrait indiquer quelles doivent être les sorties (les courants correspondants à telle ou telle broche) pour telle série d'entrée (les courants émis à telle ou telle broche). Malheureusement il s'agit bien de série d'entrée qui peut être d'une longueur conséquente, car un microprocesseur n'est pas un circuit combinatoire mais un circuit séquentiel à un très grand nombre d'états. Par conséquent, cette façon d'aborder un microprocesseur n'est pas très efficace dans une première étape. Nous allons donc considérer un microprocesseur déjà relié à d'autres matériels pour obtenir un micro-ordinateur et nous allons étudier sa réaction dans un tel contexte. Bien entendu nous reviendrons plus tard, dans le cadre du cours d'architecture, à une étude matérielle du microprocesseur.

Pour aborder les fonctionnalités d'un microprocesseur, nous avons intérêt à considérer le microprocesseur le plus simple possible et non le dernier sorti.

Le premier microprocesseur, le `i4004`[1] d'Intel date de 1971. C'est certainement le plus simple des microprocesseurs mais il est difficile à trouver de nos jours et encore plus dans un environnement qui permettrait de faire des tests.

1. À la vérité il s'agissait du circuit intégré `4004` d'une famille qui en comptait quatre, numérotés de `4001` à `4004`. Le nom de microprocesseur ne viendra que plus tard. On lui adjoint un 'i' de nos jours pour indiquer qu'il s'agit d'un circuit intégré d'Intel.

Bien entendu on pourrait utiliser un émulateur mais l'expérience montre que dans ce cas le néophyte est moins intéressé car il n'est pas sûr que tout se passerait de la même façon avec un microprocesseur réel.

Dans la longue série des microprocesseurs conçus par Intel[2], un est privilégié : il s'agit du `i8086` car une version bridée (mais équivalente du point de vue fonctionnel), le `i8088`, fut choisie par IBM pour être le cœur de son premier micro-ordinateur en 1981, appelé tout simplement **PC** pour *Personal Computer*. On peut encore en trouver et cet environnement (avec le système d'exploitation MS-DOS) est le moyen le plus pratique de s'initier aux fonctionnalités des microprocesseurs. Cependant nous n'exigerons pas d'acquérir un tel équipement. Intel a fait le choix de la **compatibilité ascendante** de ses microprocesseurs : cela signifie que toute fonctionnalité de la n-ième génération de microprocesseurs se retrouve sur la (n+1)-ième (avec des fonctionnalités nouvelles). Donc tout microprocesseur de Intel est compatible avec le `i4004`, ce qui ne nous intéressera pas vraiment mais nous verrons que ceci a certaines conséquences, et surtout avec le `i8086`. Ce choix a évidemment des aspects négatifs mais également, et nous en utiliserons un, des aspects positifs. La plupart des micro-ordinateurs vendus de nos jours utilisent un microprocesseur Intel.

Nous travaillerons donc sur un micro-ordinateur muni d'un microprocesseur Intel ou compatible (AMD entre autres). Nous aurons également besoin de feu le système d'exploitation MS-DOS de Microsoft. Là encore Windows du même Microsoft (mais pas Linux ou MacOS) suffira la plupart du temps. Le système d'exploitation MS-DOS possède un utilitaire, appelé `debug` (car il servait à déboguer les programmes), qui permet de s'initier à un langage symbolique du `i8086` très proche du langage machine du microprocesseur. Nous utiliserons cet outil pour notre exploration des fonctionnalités des microprocesseurs. Cet outil se retrouve également sous Windows.

2. Bien sûr ce n'est plus le seul fabricant (on dit **fondeur**) de microprocesseurs de nos jours mais il demeure le plus important.

Table des matières

Table des figures

Première partie

Fonctions essentielles d'un microprocesseur

Chapitre 1

Accès interne

Ce que l'on espère d'un microprocesseur, c'est de pouvoir faire des calculs, très rapidement et de nature diverse. Mais à quoi bon faire des calculs si on ne peut pas entrer de données ou sortir les résultats ? Les premières fonctions importantes d'un microprocesseur concernent donc les accès. C'est ce que nous allons voir dans ce chapitre et les deux suivants ; nous verrons les façons de calculer ensuite.

Nous allons dire en quoi consiste ces instructions de façon générale, puis comment on les utilise pour le microprocesseur que nous avons choisi de prendre en exemple, à savoir le `8086` d'Intel. Nous commencerons par les instructions concernant l'accès interne aux registres.

1.1 Étude générale

1.1.1 Première hiérarchie des instructions d'un microprocesseur

Une modèlisation des ordinateurs.- Un micro-ordinateur peut être modélisé de la façon suivante, pour la programmation qui nous intéresse. La partie essentielle en est le *microprocesseur* qui permet d'effectuer des calculs. Celui-ci contient des **registres internes** : ce sont des éléments de mémoire de capacité très limitée qui permettent de sauvegarder quelques valeurs. Le microprocesseur a donc besoin d'accéder à de la **mémoire vive** (ou **mémoire centrale**), de plus grande capacité, dans laquelle sont stockées les données, le programme lui-même et les données auxiliaires. Il a également besoin d'**entrée-sortie** pour pouvoir entrer les données et sortir les résultats (et également pour communiquer avec la **mémoire de masse**).

Mémoire de masse.- Votre expérience de la manipulation des ordinateurs, même si elle est très courte, montre que sur nos ordinateurs actuels même la mémoire vive n'est pas suffisante. Et ceci pour deux raisons : d'une part, certaines applications, telles que les bases de données, exigent une très grande capacité mémoire et, d'autre part, la technologie utilisée fait que le contenu de la mémoire vive est perdue après chaque session de travail. Il faut donc utiliser des **mémoires de masse**. Le microprocesseur doit donc posséder un jeu d'instructions pour accéder à celle-ci. Nous verrons que, du point de vue du microprocesseur, il s'agit du même jeu d'instructions que pour les entrée-sortie.

Remarque sur la terminologie.- Nous avons conservé le nom de *mémoire vive* qui est traditionnel mais le qualificatif de *vive* fait référence à une caractéristique des technologies utilisées actuellement. En effet les éléments de mémoire sont réalisés de façon telle que lorsqu'il n'y a plus de courant électrique, ils perdent leur contenu. Ce n'est évidemment pas ce qui était recherché, mais c'est comme ça. Rien ne dit que cette caractéristique perdurera dans l'avenir, par contre on aura toujours besoin de mémoire centrale.

Le qualificatif '*de masse*' correspond également à une caractéristique : la capacité d'une mémoire de masse est plus importante que celle de la mémoire vive (mais d'accès beaucoup plus lent).

On parle donc aussi de **mémoire primaire** et de **mémoire secondaire** au lieu de mémoire vive et de mémoire de masse, mais cela ne change pas grand chose. On ne voit pas *a priori* pourquoi utiliser deux types de mémoire.

Hiérarchie des instructions.- On peut donc considérer que les instructions d'un microprocesseur sont de trois sortes :

- les **instructions d'accès** au microprocesseur, à la fois en entrée, en sortie et en ce qui concerne les *registres insternes* ;
- les **instructions de calcul**, c'est en général pour elles que l'on utilise un ordinateur ;
- d'autres instructions, plus spécialisées et non théoriquement indispensables, dont le jeu dépend fortement du microprocesseur utilisé.

1.1.2 Les instructions d'accès

Hiérarchie des instructions d'accès.- La modélisation des ordinateurs que nous avons donnée ci-dessus nous explique qu'il y ait trois types d'instructions d'accès :

- les **mouvements entre registres** qui permettent de transférer une valeur d'un registre dans un autre ;
- les **accès à la mémoire vive** qui permettent de transférer une valeur d'un registre vers un élément de mémoire vive, ou *vice-versa* ;
- les **entrées-sorties** qui permettent de transférer une valeur d'un registre vers un **périphérique**, ou *vice-versa*.

On remarquera qu'un registre est toujours impliqué et qu'il n'est pas possible, *a priori*, de transférer, par exemple, directement une valeur d'un élément de mémoire vive à un autre.

Entrées/sorties mappées en mémoire ou indépendantes.- Il existe deux techniques pour accéder aux périphériques :

- La première consiste à accéder à un périphérique à travers un petit nombre d'éléments mémoire. On parle alors d'**entrée-sortie mappée en mémoire**.
- La seconde utilise des adresses particulières pour les entrées-sorties, indépendantes de celles pour la mémoire. On parle alors d'**entrée-sortie indépendante**.

Les microprocesseurs Motorola qui équipaient les premiers Macintosh, par exemple, utilisent la première technique alors que les microprocesseurs d'Intel utilisent la seconde.

1.2 Les registres

1.2.1 Notions générales

Notion.- La notion de *mémoire* est importante pour un ordinateur. Comme nous l'avons vu, on distingue, d'un point de vue matériel, les *registres* qui sont les éléments de mémoire situés sur le microprocesseur lui-même, les éléments de *mémoire vive* installés sur la carte mère et la *mémoire de masse* située sur des périphériques (disquettes, disque dur, cd-rom...). Bien entendu cette distinction, d'origine matérielle, va avoir des retombées sur le logiciel, tout au moins pour les langages à bas niveau tels que les langages d'assemblage.

Nombre de registres.- Seuls deux registres sont indispensables en théorie : l'**accumulateur**, qui contient les résultat des calculs, et le **compteur ordinal** (ou **pointeur d'instruction**), qui contient le numéro de la prochaine instruction à exécuter. Mais en général un microprocesseur comporte beaucoup plus de registres pour accélérer la vitesse de l'ordinateur (ceci évite le passage sans cesse de l'accumulateur à un élément de mémoire vive).

Numérotation des bits.- Par convention, les bits d'un élément de mémoire, et donc d'un registre, sont numérotés de 0 à N de façon telle que le bit i contienne le i-ième chiffre binaire (celui correspondant à 2^i) si le contenu est un entier naturel.

1.2.2 Les registres du `i8086`

Le microprocesseur `i8086` possède quatorze registres, soit douze de plus que le minimum nécessaire.

1.2.2.1 Les quatorze registres

Désignation des registres.- Les registres du microprocesseur `i8086` sont désignés de la façon suivante :

- quatre **registres de données** dont les noms sont **ax**, **bx**, **cx** et **dx** ;
- deux **registres de pointeurs** dont les noms sont **sp** et **bp** ;
- deux **registres d'index** dont les noms sont **si** et **di** ;
- quatre **registres de segment** dont les noms sont **cs**, **ds**, **ss** et **es** ;
- le *pointeur d'instruction* **ip** ;
- le **registre des indicateurs** (indicateur se dit *flag* en anglais).

Les huit premiers registres sont appelés **registres généraux** car ils peuvent être utilisés pour la manipulation des données alors que les autres sont spécialisés. Pour le moment il suffit de retenir qu'il y a quatorze registres ainsi que leurs noms (bien que le dernier registre n'ait pas de nom).

Origine des noms des registres.- Les noms des registres ne sont pas donnés par hasard. Ils rappellent les contextes essentiels dans lesquels ils sont utilisés.

Les lettres **a**, **b**, **c** et **d** des registres de données, outre le jeu sur l'ordre alphabétique, sont les initiales de 'Accumulateur', 'Base', 'Compteur' et 'Données'. Le 'x' est pour 'eXtend' puisque le microprocesseur antérieur, avec lequel le `i8086` est compatible, avait des registres de 8 bits et non de 16 bits.

Les registres de pointeur servent à repérer quelque chose, plus précisément l'instruction en cours (d'où `ip` pour *Instruction Pointer*), l'élément en haut de la pile[1] (d'où `sp` pour *Stack Pointer*, pile se disant *stack en anglais*) et la base dans un mode d'adressage (d'où `bp` pour *Base Pointer*).

Les registres d'index servent à désigner chaque lettre d'une chaîne de caractères lors d'une copie ou d'une comparaison, d'où les noms `si` pour *Source Index* et `di` pour *Destination Index*.

Les registres de segment servent à désigner les quatre segments[2] prévus pour le `i8086`, comme nous le verrons, d'où les noms `cs` pour *Code Segment*, `ds` pour *Data Segment*, `ss` pour *Stack Segment* et `es` pour *Extra Segment*.

Taille des registres.- Tous les registres du `i8086` ont une taille de seize bits (soit deux octets).

1. Qu'importe de quoi il s'agit pour l'instant.
2. Qu'importe également de quoi il s'agit pour l'instant.

Décomposition des registres généraux.- Pour des raisons de compatiblité avec les registres du `i8080`, on peut accéder aux registres de données octet par octet. On a donc huit registres d'un octet : **ah**, **al**, **bh**, **bl**, **ch**, **cl**, **dh** et **dl**. Par exemple le registre **ax** est la concaténation des deux registres **ah** et **al** : le *h* est là pour *High-order byte*, c'est-à-dire les bits de 8 à 15 du registre **ax**, et le *l* pour *Low-order byte*, c'est-à-dire les bits de 0 à 7 du registre **ax**.

1.2.2.2 Structure du registre des indicateurs

Le registre des indicateurs n'a pas de mnémonyme en langage symbolique car il ne peut pas être adressé directement par le programmeur.

Nous avons déjà vu que c'est un registre à seize bits. Chacun de ces bits est un **indicateur** à lui tout seul. En fait seuls neuf bits (sur les seize possibles) sont utilisés pour le microprocesseur `i8086`. Ces neuf bits sont divisés en deux groupes, trois sont des **indicateurs de contrôle** et six des **indicateurs de statut**. Un indicateur est **positionné** (en anglais *a flag is set*) s'il a la valeur 1.

Chacun des neuf bits utilisés porte un nom, rappelant sa fonction essentielle (que nous détaillerons au fur et à mesure de l'avancée du cours). Donnons ces noms en indiquant rapidement, pour l'instant, son rôle.

- Le bit 0 est l'**indicateur de retenue CF** (pour l'anglais *Carry Flag*). Il représente le dix-septième bit d'une addition ou d'une soustraction de deux nombres de seize bits, le neuvième bit dans le cas de deux nombres de huit bits. Il sert aussi dans les opérations de rotation.
- Le bit 2 est l'**indicateur de parité PF** (pour l'anglais *Parity Flag*). Il est égal à 1 lorsque le résultat d'une opération sur 8 ou 16 bits présente un nombre pair de bits égaux à 1.
- Le bit 4 est l'**indicateur de retenue auxiliaire AF** (pour l'anglais *Auxiliary Flag*). Il fonctionne comme `CF` mais il réagit aux opérations sur un demi-octet (ou **quartet**, *nibble* en anglais) ; il est utilisé avec le format de nombres appelé BCD.
- Le bit 6 est l'**indicateur de zéro ZF** (pour l'anglais *Zero Flag*). Il est positionné lorsque le résultat d'une opération sur 8 ou 16 bits donne zéro.
- Le bit 7 est l'**indicateur de signe SF** (pour l'anglais *Sign Flag*). Il est positionné lorsque le résultat d'une opération sur 8 ou 16 bits est négatif.
- Le bit 8 est l'**indicateur de trappe TF** (pour l'anglais *Trap Flag*, dit aussi *Single Step Flag*). Cet indicateur est prévu pour la recherche des erreurs dans les programmes : lorsqu'il est positionné, une interruption se déclenche automatiquement à la suite de chaque instruction machine.
- Le bit 9 est l'**indicateur d'interruption IF** (pour l'anglais *Interrupt Enable Flag*). Lorsqu'il est positionné, les interruptions dites masquables s'exécutent normalement. Si l'indicateur est égal à 0, elles sont inhibées.
- Le bit 10 est l'**indicateur de direction DF** (pour l'anglais *Direction Flag*). Il sert à piloter les traitements de chaînes de caractères qui utilisent les registres **DI** et **SI**. Lorsqu'il est positionné, les compteurs sont incrémentés, sinon ils sont décrémentés.

– Le bit 11 est l'**indicateur de débordement OF** (pour l'anglais *Overflow Flag*). Cet indicateur intervient lorsqu'on effectue des calculs sur des nombres signés et lorsque le résultat d'une opération sur 8 ou 16 bits dépasse le domaine autorisé. L'interprétation dépendra du programmeur : **OF** ne fait que suggérer une erreur potentielle.

1.3 Manipulation des registres avec `debug`

1.3.1 Le logiciel `debug`

La commande `debug` du système d'exploitation MS-DOS[3] permet d'examiner et de changer le contenu de la mémoire, d'entrer un programme (en langage d'assemblage) et de faire exécuter un tel programme. Commençons par voir comment activer `debug` et comment en sortir. Nous verrons tout au long des sections suivantes les options de cette commande.

Accès.- Pour accéder à `debug`, il suffit tout simplement d'écrire son nom après le prompteur (en ligne MS-DOS, dans une fenêtre MS-DOS si on utilise Windows) :

```
C:> debug <retour>
```

Remarques.- Comme toujours avec MS-DOS on peut écrire indifféremment en minuscule ou en majuscule. Si la variable d'environnement `path` indique dans `autoexec.bat` où se trouve les commandes du MS-DOS, on peut accéder à `debug` depuis n'importe quel répertoire.

Réaction.- Le prompteur de `debug` apparaît alors, à savoir un tiret '-' :

```
C:> debug <retour>
-
```

Sortie.- La première commande de `debug` à connaître est évidemment de savoir comment sortir. Il suffit d'écrire 'q' (pour l'anglais *Quit*) :

```
C:> debug <retour>
-q <retour>
C:>
```

1.3.2 Contenu en hexadécimal

Bases utilisées en langage symbolique.- En langage symbolique, il est traditionnel d'utiliser les entiers exprimés non pas en base dix (**numération décimale**) mais en base deux (**numération binaire**, qui correspond bien au matériel) ou le plus souvent, de façon intermédiaire, en base seize (**numération hexadécimale**, bien entendu parce que cette dernière base est liée à la base deux).

Écriture des entiers.- Avec `debug` les entiers sont nécessairement exprimés en base seize. On écrit un entier en base seize en utilisant les chiffres successifs '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a' ou 'A' (pour dix), 'b' ou 'B' (pour onze), 'c' ou 'C' (pour douze), 'd' ou 'D' (pour treize), 'e' ou 'E' (pour quatorze) et

3. On retrouve cette commande, exécutable dans une fenêtre d'émulation de terminal, sous Windows.

'f' ou 'F' (pour quinze). Par exemple vingt s'écrira 14 (puisque vingt égal seize plus quatre) ou, plus exactement, pour indiquer que la base n'est pas la base dix habituelle, 14_{16} en mathématiques et `14h` (avec 'h' pour hexadécimal) ou `0x14` en informatique.

1.3.3 Examen du contenu des registres

Introduction.- Comme nous l'avons dit, `debug` permet d'examiner et de changer le contenu de la mémoire, d'entrer un programme (en langage d'assemblage) et de faire exécuter un tel programme. Par mémoire, il faut entendre mémoire vive et mémoire de masse. Si on utilise un microprocesseur, le contenu des registres change sans cesse ; on ne peut donc pas obtenir d'information (même instantanée) sur le contenu de ceux-ci.

`Debug` permet cependant d'émuler la manipulation des registres.

Syntaxe.- Pour visualiser le contenu de tous les registres (de m'émulateur) avec `debug`, il suffit d'utiliser la commande 'R' (pour *Register*).

Exemple.- On a :

```
C:> DEBUG <retour>
-R <retour>
AX=0000  BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
CS=2377 DS=2377 ES=237 SS=2377 IP=0100  NV UP DI PL NZ NA PO NC
2377:0100 D6       DB  D6
```

Commentaires.- 1^o) `Debug` répond à la commande 'R' par trois lignes : la première ligne indique le contenu des registres généraux, des registres de pointeur et des registres d'index ; la seconde ligne indique le contenu des registres de segment, la valeur du pointeur d'instruction et la valeur de huit des indicateurs ; la troisième ligne montre l'instruction pointée par `CS:IP`.

2^o) La valeur d'un registre (sauf pour le registre des indicateurs) est indiquée en hexadécimal, avec quatre chiffres de façon systématique (même si la valeur commence par des zéros).

3^o) Bien entendu, comme nous l'avons déjà dit, il ne s'agit pas des vraies valeurs des registres mais de ceux de émulateur, tout au moins en ce qui concerne la premièrre ligne. À chaque appel de `debug`, les registres de la première ligne sont remis à zéro et tous les indicateurs sont positionnés à zéro. Les contenus des registres de segment varient d'une session à l'autre mais ils ont tous la même valeur. Les instructions exécutées par `debug` changeront la valeur des registres comme le ferait le microprocesseur.

4^o) Nous analyserons plus tard la signification de la troisième ligne.

Contenu d'un seul registre.- Il est possible de ne faire apparaître le contenu que d'un seul registre en faisant suivre la commande 'R' du nom du registre. Par exemple :

```
C:> DEBUG
-R AX
AX 0000
```

:

Il suffit d'appuyer sur <retour> pour l'instant pour revenir au prompteur de `debug`.

Code des indicateurs.- Le code pour les indicateurs est le suivant :

Indicateur	**Code lorsque positionné** (= 1)	**Code lorsque non positionné** (= 0)
OF	OV (OVerflow)	NV (No oVerflow)
DF	DN (DowN)	UP (UP)
IF	EI (Enable Interrupt)	DI (Disable Interrupt)
SF	NG (NeGative)	PL (PLus)
ZF	ZR (ZeRo)	NZ (Not Zero)
AF	AC (Auxiliary Carry)	NA (No Auxiliary carry)
PF	PE (Parity Even)	PO (Parity Odd)
CF	CY (CarrY)	NC (No Carry)

1.3.4 Changement du contenu d'un registre

Syntaxe.- Pour changer le contenu d'un registre de l'émulateur (n'importe quel registre à 16 bits sauf le registre des indicateurs) en utilisant `debug`, il suffit de l'appeler avec la commande 'R' suivi de son nom. Sa valeur est alors indiquée (comme nous l'avons vu ci-dessus) et le prompteur est ':'. Il suffit d'indiquer la valeur souhaitée (en hexadécimal, suivie d'un retour).

Exemple.- Si on veut que le registre `ax` contienne `FF01h`, on effectue les manipulations suivantes :

```
C:> DEBUG
-R AX
AX 0000
:FF01
-R AX
AX FF01
:
```

la dernière commande de `debug` permet de vérifier qu'on a bien changé la valeur du registre.

Remarques.- 1^o) Il est fortement recommandé de ne pas changer (pour l'instant) le contenu des registres de segment. En effet ces valeurs ont été données par le système d'exploitation là où il y a de la place disponible ; les changer pourrait avoir pour conséquence de modifier une zone utilisée par ailleurs.

2^o) Même si on met moins de quatre chiffres (hexadécimaux), `debug` affiche toujours avec quatre chiffres :

```
C:> DEBUG
-R CX
CX 0000
:2
-R CX
AX 0002
:
```

3^o) `Debug` indique une erreur si on essaie de placer un contenu à plus de cinq chiffres :

```
C:> DEBUG
-R CX
CX 0000
:12345
Erreur
```

ou si on essaie d'accéder à un registre à huit bits :

```
C:> DEBUG
-R AH
Erreur
```

4^o) Nous verrons plus tard comment changer le contenu du registre des indicateurs.

5^o) Rappelons qu'il s'agit d'une émulation. Le registre ne contient pas réellemnt la valeur indiquée.

1.4 Accès aux registres

Nous avons vu la notion de registre. Voyons comment initialiser un registre et comment transférer une valeur d'un registre à un autre.

1.4.1 Copie entre registres

Syntaxe.- L'instruction en langage symbolique est :

```
mov destination, source
```

où `destination` et `source` sont des registres (différents du registre des indicateurs) de même taille (seize ou huit bits). Le mnémonyme `mov` rappelle évidemment l'anglais *move*.

Sémantique.- La signification de cette instruction est la copie du contenu du registre source dans le registre destination.

Remarques.- 1^o) Attention à l'ordre, d'abord la destination, ensuite la source, qui n'est pas nécessairement l'ordre auquel on pourrait s'attendre. On peut penser à l'analogue en langage de haut niveau :

```
destination := source;
```

2^o) La dénomination anglaise est trompeuse car il s'agit bien d'une copie et non d'un déplacement (le contenu du registre source ne change pas).

1.4.2 Initialisation des registres

Syntaxe.- L'instruction en langage symbolique est :

```
mov destination, constante
```

où `destination` est un registre qui n'est ni un registre de segment ni le registre des indicateurs et où `constante` est une constante hexadécimale (de taille compatible avec celle du registre).

Sémantique.- La signification de cette instruction est l'initialisation du registre `source` par cette constante.

1.5 Codage des programmes

Nous allons voir comment coder un programme pour le microprocesseur `i8086`, toujours en utilisant le logiciel `debug`.

1.5.1 Premier programme

Nous allons voir, sur un exemple simple, comment mettre en place un programme avec `debug`.

1.5.1.1 Langage symbolique et langage machine

Le **langage machine** est celui qui est compris du microprocesseur : il s'agit d'une succession de 0 et de 1, que l'on écrit traditionnellement en hexadécimal. Un **langage symbolique** est un langage non compris par le microprocesseur mais proche du langage machine tout en utilisant des mnémoniques bien utiles pour nous, pauvres humains. Il faut évidemment, à un certain moment, traduire le **programme source** (en langage symbolique) en **programme objet** (en langage machine). Nous utiliserons, une fois de plus, le logiciel `debug` pour cela, tout au moins dans une première étape. Nous verrons plus tard comment se fait cette traduction (et nous pourrons alors l'effectuer à la main si cela nous fait plaisir).

1.5.1.2 Le programme

Écrivons un programme en langage symbolique qui initialise les registres `ax` et `bx` à 16 (ou 10h) :

```
mov ax, 10
mov bx, ax
```

Remarque.- On aurait pu, évidemment, initialiser directement le registre `bx` mais notre façon de faire nous permet de voir deux formes de l'instruction `mov`.

1.5.1.3 Codage du programme

Méthode.- Pour coder le programme, il suffit d'utiliser la commande 'A' (pour *assemble*) de `debug` : on fait suivre 'A' de l'adresse de départ (en hexadécimal) ou, mieux, de rien (l'adresse `100h` est prise par défaut) ; on répond à chaque adresse par une instruction en langage symbolique (celui compris par `debug`, évidemment) ; on termine par l'instruction `int 3` et par un retour chariot lorsqu'au prompteur on a l'adresse suivante.

Exemple.- Dans notre cas cela va donner :

```
C:>debug
-a
249c:0100 mov ax,10
```

```
249c:0103 mov bx,ax
249c:0100 int 3
249c:0106
-
```

Remarques.- 1^o) Si vous écrivez une instruction (syntaxiquement) incorrecte que `debug` ne peut pas assembler, il indique une erreur.

2^o) L'instruction `int 3` est ce qu'on appelle un **point d'arrêt**. Si on l'oublie, lors de l'exécution, les instructions suivantes (dans l'ordre de la mémoire) seront également exécutées. Ceci n'est pas ce qui est voulu. De plus il y a toutes les chances qu'on arrive de cette façon à un certain moment à un blocage du système ; on n'a plus alors qu'à redémarrer l'ordinateur.

1.5.1.4 Exécution d'un programme

Principe.- Pour exécuter un programme avec `debug`, on utilise la commande `G` (pour *go*). Son format complet est :

```
G <= adresse de départ> <adresse d'arrêt>
```

Si aucune adresse n'est précisée, `debug` commence l'exécution à l'adresse `CS:IP` jusqu'à ce qu'un point d'arrêt soit trouvé. Un **point d'arrêt** est généré par l'interruption `int 3`, d'où l'intérêt de terminer un programme de cette façon. Lorsqu'un point d'arrêt est atteint, `debug` affiche le contenu des registres et un prompteur "-".

On peut placer jusqu'à dix points d'arrêt. `Debug` s'arrête au premier qu'il rencontre. Nous verrons que plusieurs points d'arrêt peuvent être utiles lorsque le programme contient des branchements.

Exemple.- On peut faire exécuter le programme ci-dessus de plusieurs façons.

- 1^o) La première façon consiste à utiliser la commande `G` sans autre indication :

```
-g
AX=0010 BX=0010 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=249C ES=249C SS=249C CS=249C IP=0105 NV UP EI PL NZ NA PO NC
249C:0105 CC INT 3
```

- 2^o) La seconde façon consiste à utiliser la commande `G` en ne donnant que l'adresse de départ :

```
-g =100
AX=0010 BX=0010 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=249C ES=249C SS=249C CS=249C IP=0105 NV UP EI PL NZ NA PO NC
249C:0105 CC INT 3
```

- 3^o) La troisième façon consiste à utiliser la commande `G` avec son format complet (le point d'arrêt n'est pas nécessaire dans ce cas-là) :

```
-g =100 103
AX=0010 BX=0010 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=249C ES=249C SS=249C CS=249C IP=0105 NV UP EI PL NZ NA PO NC
249C:0103 89C3 MOV BX,AX
```

- 4^o) La quatrième façon consiste à utiliser la commande `G` avec seulement l'adresse d'arrêt (le point d'arrêt n'est pas nécessaire non plus dans ce cas-là) :

```
-g 103
AX=0010 BX=0010 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
```

```
DS=249C ES=249C SS=249C CS=249C IP=0105 NV UP EI PL NZ NA PO NC
249C:0103 89C3 MOV BX,AX
```

1.5.2 Trace d'un programme

1.5.2.1 Notion

L'inconvénient avec l'exécution d'un programme, c'est qu'on obtient les résultats au premier point d'arrêt rencontré sans donner les détails de ce qui se passe à chaque instruction. Ceci est suffisant en général, sauf lorsqu'il y a une erreur et qu'on ne sait pas à quel endroit. Dans ce dernier cas, on utilise alors la **trace du programme**, c'est-à-dire que l'on effectue une seule instruction à la fois.

1.5.2.2 Trace avec `debug`

Syntaxe.- On utilise la commande `T` qui permet d'exécuter une instruction et qui affiche ensuite le contenu des registres.

Exemple.- Reprenons le programme précédent (il n'y a plus besoin du point d'arrêt) et effectuons la trace du déroulement du programme. On commence en général par demander l'affichage des registres pour visualiser leur état initial.

```
C:\>debug
-a
17A4:0100 mov ax,10
17A4:0103 mov bx,ax
17A4:0105
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=17A4  ES=17A4  SS=17A4  CS=17A4  IP=0100   NV UP EI PL NZ NA PO NC
17A4:0100 B81000        MOV     AX,0010
-t

AX=0010  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=17A4  ES=17A4  SS=17A4  CS=17A4  IP=0103   NV UP EI PL NZ NA PO NC
17A4:0103 89C3          MOV     BX,AX
-t

AX=0010  BX=0010  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=17A4  ES=17A4  SS=17A4  CS=17A4  IP=0105   NV UP EI PL NZ NA PO NC
17A4:0105 7422          JZ      0129
-q
```

1.5.3 Sauvegarde et récupération d'un programme

On peut sauvegarder un programme sur disque, pour pouvoir le récupérer plus tard.

1.5.3.1 Sauvegarde d'un programme avec `debug`

Pour sauvegarder un programme avec `debug`, il faut nommer un fichier qui le contiendra et écrire le programme dans ce fichier.

Nommer un fichier.- On peut indiquer directement l'adresse de début du fichier mais une meilleure façon (pour l'instant) consiste à donner un nom au fichier de sauvegarde. On se sert pour cela de la commande `N` (pour *name*) dont la syntaxe est :

```
-N <nom du fichier>
```

où le nom du fichier vérifie les spécifications de MS-DOS. Ce fichier sera placé dans le répertoire depuis lequel on a lancé `debug`.

Écriture.- Pour sauvegarder un programme (écrit avec la commande `A`), il suffit d'indiquer le nom du fichier (à l'aide de la commande `N`), d'indiquer le nombre d'octets à sauvegarder à l'aide de `BX:CX` puis d'écrire ce programme sur le disque avec la commande `W` (pour *write*).

Dans le cas ci-dessus on aura :

```
C:>debug
-a
249c:0100 mov ax,10
249c:0103 mov bx,ax
249c:0100 int 3
249c:0106
-R BX
BX 0000
:
-R CX
CX 0000
:6
-N EX1.COM
-W
Ecriture de 00006 octets
-Q
```

Remarque.- Vous remarquerez l'extension classique d'un fichier nommé avec la commande `N`, à savoir ".com".

1.5.3.2 Récupération d'un programme avec `debug`

Introduction.- Pour lire un programme (sauvegardé antérieurement), il suffit d'indiquer le nom du fichier (toujours à l'aide de la commande `N`), puis de demander de charger (*to load* en anglais) ce programme grâce à la commande `L`.

Syntaxe du chargement.- Si le fichier a été nommé, il suffit d'utiliser :

```
L [adresse]
```

pour charger le programme à une adresse déterminée. Si aucune adresse n'est indiquée, le programme est chargé à `CS:100`.

Exemple.- Dans le cas ci-dessus on aura :

```
C:>debug
-N EX1.COM
-L
```

Il y a un petit problème à ce moment-là puisque nous ne savons pas comment visualiser le programme (action que nous allons voir dans la sous-section suivante).

Autre façon.- Une autre façon de faire (il s'agit d'un raccourci) consiste à indiquer le nom du programme à l'appel de `debug` :

```
C:>debug EX1.COM
```

1.6 Code machine du programme

La commande `U` (pour *unassemble*, c'est-à-dire *désassembler* en anglais) permet à la fois de visualiser le code machine d'un programme et l'équivalent de ce code en langage symbolique de `debug`.

Syntaxe.- On peut utiliser l'un des formats suivants :

```
U <adresse de départ> <adresse de fin>
U <adresse de départ> <L nombre d'octets>
```

où le nombre d'octets est exprimé en hexadécimal. Si on n'indique rien, 32 octets sont affichés en commençant à l'adresse `CS:IP`.

Exemple.- Dans le cas de l'exemple ci-dessus on a :

```
C:>debug EX1.COM
-U 100 105
24C3:0100 B81000 MOV AX,0010
24C3:0103 89C3   MOV BX,AX
24C3:0105 CC     INT 3
-
```

Sans entrer dans les détails pour l'instant, on remarquera que le code pour `mov AX,`, à savoir `B8h`, n'est pas le même que pour `mov BX,`, qui lui est `89h`.