Introduction au langage « assembleur » (Intel en particulier)

Par haypo

Date de publication : 11 avril 2001

Dernière mise à jour : 28 décembre 2012

DÉBUTANT

J'ai écrit cette « introduction » à l'assembleur assez rapidement, et croyant que vous vous y connaissiez un peu dans ce domaine. Alors si vous trouvez une erreur dans ce cours, ou quelque chose vous est incompréhensible, écrivez-moi : vous m'aiderez pour mon cours, et vous aiderez les prochains lecteurs de ce cours.

Si vous avez des remarques concernant ce tutoriel, un espace de dialogue vous est proposé sur le forum. **Commentez**

I - Introduction

Le processeur d'un ordinateur ne comprend aucun langage de programmation ni Turbo Pascal, ni assembleur, ni même C++ ou Java (du moins aucun des processeurs compatibles Intel 80x86). Il ne connait qu'une chose : **le langage machine**. C'est une liste de nombre 8 bits en hexadécimal du style : « B0h 12h », pas très parlant. L'assembleur c'est la version « humaine » du code machine, pour notre exemple cela donne : « mov al, 12h » (comprendre « copie la valeur 12h dans le registre AL »). Comme vous pouvez le constater, c'est le **langage de programmation le plus proche du processeur** (mis à part si vous arrivez à programmer en langage machine).

Question qui vous saute à l'esprit : **à quoi ça sert ? L'intérêt premier est la vitesse**, car c'est le langage avec lequel on peut faire les programmes les plus rapides. Le deuxième intérêt, qui n'est peut-être plus d'actualité aujourd'hui est l'accès aux interruptions DOS qui permettent un accès direct au matériel tels la souris, l'écran, ou la carte vidéo. Certains vétérans de la programmation n'utilisent que l'assembleur, certains pirates en particulier. Pourquoi ? **Pour faire les plus petits programmes possible**, car le **compilateur** ne fait que traduire l'assembleur en code machine (ce qu'on pourrait presque faire manuellement avec de l'entrainement). Et en même temps savoir exactement ce que contiendra notre fichier binaire (fichier exécutable, « .exe » ou « .com » sur les plates-formes Microsoft).

Personnellement, je vous déconseille vivement du faire du 100 % ASM (ASseMbleur), car on se perd vite dans le code source (qui atteint souvent plusieurs milliers de lignes, car chaque instruction prend une ligne), et ce n'est pas très parlant comme langage. Je vous le recommande (uniquement) pour **optimiser des fonctions déjà existantes dans votre langage de prédilection** (Pascal, C++…) demandant une vitesse optimale, souvent pour les fonctions les plus souvent appelées ; tracé de ligne, copie d'un bloc mémoire, etc.

II - Commençons par le vocabulaire

Compilateur - **Hexadécimal** - **Bit** - **Octet** - **Mot** - **Double mot** - **Segment et offset** - **Interruption** - **Pile** - **Flags**

- **Compilateur** : programme traduisant un code source (fichier texte, pour l'assembleur : fichiers portant l'extension « .asm ») en code machine, c'est-à-dire produisant un fichier binaire (ou fichier « exécutable », portant l'extension « .exe » ou « .com » sur les plates-formes Microsoft).
- **Hexadécimal** : dans la notation des nombres actuelle, on utilise les chiffres arabes compris entre 0 et 9, on a donc 10 chiffres, et le chiffre le plus grand est 9 (10-1). On dit de ce système de notation qu'il est en **base 10** (**décimal**, déci = 10 en grec). Pour écrire 451 par exemple, on peut le décomposer comme 4*100 + 5*10 + 1*1 = 4*(10^2) + 5*(10^1) + 1*(10^0).
  Le système hexadécimal au contraire est en **base 16**. Pour écrire 10 on utilise la lettre « a », 11 : « b », 12 : « c », 13 : « d », 14 : « e », et 15 : « f ». Donc par exemple 20 (16*1+4) s'écrira 14h. « h » est le suffixe des nombres hexadécimaux en assembleur, en Turbo Pascal on utilise le préfixe « $ » ($14) et en C le préfixe « 0x » (0x14). Pour convertir de l'hexadécimal en décimal, on procède à sa décomposition : 3DAh = 3*(16^2) + D*(16^1) + A*(16^0) = 3*256 + 13*16 + 10*1 = 986.
  Remarque : le processeur central (CPU : Central Processor Unit) travaille uniquement en binaire (uniquement avec **bits**), notation en base 2. Par exemple 9 s'écrira 1*8 + 1 = 1*(2^3) + 0*(2^2) + 0*(2^1) +1*(2^0) = « 1001b » (« b » étant le suffixe des nombres binaires). Même le coprocesseur arithmétique qui calcule le cosinus et l'exponentiel travaille en binaire !
- **Bit** : un bit est une donnée pouvant prendre soit la valeur 0 (assimilée à false, faux en anglais) ou 1 (true, vrai en anglais). C'est l'unité du système binaire (voir la remarque de l'**hexadécimal**), un **octet** par exemple est un groupement de 8 bits.
- **Octet** : un octet est une suite de 8 bits. S'il est signé (+ ou -), il peut prendre une valeur comprise entre -128 et +127 (128 = 2 puissance (8-1) et 127 = 2 puissance (8-1) -1), car le premier bit est celui du signe (TRUE = « - ») et donc la valeur est en 7 bits. S'il n'est pas signé, il peut prendre une valeur comprise entre 0 et +255 (255 = 2^8 -1, « -1 », car le zéro est une valeur à part entière !).
- **Mot** : deux octets = 16 bits. Signé : valeurs entre -32768 (2^15) et +32767 (2^15 -1). Non signé : entre 0 à +65535 (2^16 -1).
- **Double mot** : quatre octets = 2 doubles mots = 32 bits. Signé : valeurs de -2 147 483 648 (-2^31 = -2147 millions) et +2 147 483 647 (2^31 -1), non signé : entre 0 et +4 294 967 295 (quelque 4294 millions).
- **Registre** : espace mémoire placé physiquement dans le cœur du processeur dans lequel sont stockées des valeurs. Les registres étaient au début codés sur 8 bits (un octet), ils étaient nommés « ?L » (où « ? » peut-être A,B,C ou D), puis ils sont passés en 16 bits (un mot) et deviennent « ?X ». Le truc magique est que si on modifie « ?L », on modifie la « partie basse » du mot « ?X », c'est-à-dire « ?X » est en faite composé d'un octet et de « ?L ». Puis avec l'arrivée du 386, on passe en 32 bits (double mot) et les registres deviennent « E?X », là aussi, si on modifie « ?L » ou « ?X » on modifie une partie de « E?X ».
  Consultez la **Liste des registres**.
- **Segment et Offset** : les adresses mémoire (emplacement des octets dans les barrettes de mémoire) sont définies par deux registres : le segment (partie haute de l'adresse) et l'offset (partie basse de l'offset). On note l'adresse : « segment :[offset] » (« : » est le séparateur, les crochets « [» et «] » ne sont pas obligatoires), exemple : « DS:[DI] ». Avec le 386, les offsets sont passés en 32 bits pour accéder à plus de mémoire (plus de 16 Mo), les offsets avec le préfixe « E » sont apparus (ESI, EDI, ESP, EBP), et comme pour les registres 32 bits, les registres 16 bits, les offsets SI,DI,SP,BP en sont leur partie basse.
  Consultez la **Liste des registres**.
- **Interruption** : une interruption est comme un petit programme stocké en mémoire qui est appelé plus ou moins régulièrement et qui a une tâche spécifique. Il y en a 256 au maximum, et les premières sont intégrées dans le BIOS (programme lancé au démarrage de l'ordinateur qui gère le matériel : disque dur, accès mémoire…) comme les interruptions de la carte vidéo ou le clavier. Certaines sont appelées à une fréquence constante : l'interruption 1Ch par exemple qui est un compteur qui incrémente (ajoute 1) une valeur 18.6 fois/seconde. D'autres sont appelées uniquement si on a besoin d'elles : l'interruption 09h du clavier est appelée à chaque pression ou relâchement d'une touche.
  Voir la **liste complète des interruptions**.
- **Pile** : la pile pourrait être comparée à une pile d'assiettes, on entasse des assiettes (ouais bon, c'est une image) avec l'instruction « PUSH » puis on les enlève avec l'instruction « POP ». La hauteur de la pile est la valeur du registre [E]BP. On voit ce qu'il y a à l'étage X avec une instruction dans le style « MOV AX,[BP-4] », mais pas l'enlever, car si on enlève une assiette au milieu de la pile, la pile dégringole (= plantage du PC).

- **Flags (drapeaux)** : ceux-ci sont des bits internes au processeur.
  Consultez la **Liste des drapeaux**.

III - Liste des registres

Il existe plusieurs registres ayant un sens plus ou moins précis (ils peuvent avoir une utilisation différente).

- **AL/AH/EAX** : registre général, sa valeur change très vite.
- **BL/BH/EBX** : registre général, peut servir d'offset mémoire (exemple : « mov al, byte ptr ds:[bx+10] »).
- **CL/CH/ECX :** sert en général de compteur pour les boucles (exemple : « mov ecx, 5 ; rep movsd » : copie 5 doubles mots).
- **DL/DH/EDX** : registre général, obligatoire pour l'accès aux ports (moyen de communiquer avec toutes les puces de l'ordinateur, par exemple les ports 42h et 43h servent à contrôler le haut-parleur interne, voir **IN** et **OUT**).
- **CS** : segment mémoire du code.
- **DS** : segment mémoire des données.
- **ES** : segment mémoire.
- **FS** : autre segment mémoire.
- **GS** : autre segment mémoire.
- **SS** : segment mémoire de la pile (« S » = Stack = Pile).
- **BP** : offset mémoire, très souvent une copie de SP à laquelle on soustrait une valeur pour lire dans la pile (on ne doit pas modifier SP).
- **EDI/DI** : offset mémoire utilisé avec ES (ou FS ou GS si spécifié, exemple : « mov al, byte ptr gs:[10] »).
- **EIP/IP** : offset mémoire du code (inaccessible directement, modifiable indirectement avec l'instruction CALL, JMP, ou J[cas]).
- **ESI/SI** : offset mémoire utilisé avec DS.
- **ESP/S** : offset mémoire de la pile.

IV - Liste des drapeaux

- **AF** : Auxilliary Flag = indicateur de retenue auxiliaire.
- **CF** : Carry Flag = indicateur de retenue.
- **DR** : Dirrection Flag = indicateur de direction de traitement des chaînes de caractères.
- **IF** : Interrupt Flag = indicateur d'exécution des interruptions dites « masquables ».
- **OF** : Overflow Flag = indicateur de débordement.
- **PF** : Parity Flag = indicateur de parité -> PF=0 : Impaire, PF=1 : Paire.
- **SF** : Sign Flag = indicateur de signe -> SF=0 : Positif, SF=1 : Négatif.
- **TF** : Single Step Flag = indicateur de débogage.
- **ZF** : Zero Flag = indique une valeur nulle.

V - Liste des instructions

Les caractères « **{** » et « **}** » délimitent un paramètre. Il peut être un registre, une zone mémoire ou une immédiate (une valeur, la plupart du temps c'est un nombre).

Le suffixe « **[B/W/D]** » terminant une instruction doit être remplacé soit par B, soit par W ou soit par D :

- « B » signifie que l'instruction est en 8 bits (utilise si nécessaire le registre AL).
- « W » signifie que l'instruction est en 16 bits (utilise si nécessaire AX).
- « D » signifie que l'instruction est en 32 bits (utilise si nécessaire EAX).

AND - **CALL** - **CMP** - **CMPSx** - **IN** - **IRET** - **JMP** - **J[cas]** - **LEA** - **LDS** - **LES** - **LFS** - **LGS** - **LSS** - **LODSx** - **MOV** - **MOVSx** - **MOVZX** - **MUL** - **NOT** - **OUT** - **PUSH** - **POP** - **RET** - **REP** - **SHL** - **SHR** - **STOSx** - **SCASx** - **TEST** - **XOR**

- **AND {destination},{masque}** : applique un « et » à **{destination}** par **{masque}**. Tout bit de {destination} est mis à 0 si le bit correspondant de {masque} vaut 0, et est inchangé si le bit correspondant vaut 1 :
  0 AND 0 -> 0
  0 AND 1 -> 0
  1 AND 0 -> 0
  1 AND 1 -> 1
  Équivalant en Turbo Pascal : « {destination} := {destination} **and** {masque} ».
  Équivalant en C : « {destion} **&=** {masque} ».
  Voir aussi **NOT**, **OR** et **XOR**.
- **CALL {adresse}** : appelle une procédure qui est à l'adresse {adresse}. Si une ou plusieurs instructions PUSH précèdent un CALL, ceux-ci sont des paramètres qui sont stockés dans la pile. Dans ce cas la procédure commence par ;« push [e]bp ; mov [e]bp, [e]sp » et on peut trouver la lecture des paramètres avec des instructions du genre « mov {registre},[ebp-{valeur}] ». Il ne faudra surtout pas oublier le « RET [valeur] » à la fin de la procédure (voir plus bas).
- **CMP {a}, {b}** : compare les deux variables {a} et {b}. Toujours suivi d'un saut conditionnel (« J[cas] {offset} », voir sa signification plus bas).
- **CMPS[B/D/W]** : compare l'octet/le mot/le double mot DS:ESI à ES:EDI.
- **IN {destination},{port}** : lit une valeur 8 bits sur le port **{port}** (16 bits) et la stocke dans **{destination}**. Le seul registre autorisé pour **{port}** est DX. Voir aussi **OUT**.
- **IRET {valeur}** : quitte l'interruption. Est uniquement utilisé pour les programmes résidents comme le pilote de la souris par exemple. Voir aussi **RET**.
- **JMP {offset}** : va (« J » = Jump : Sauter) à l'adresse {offset}.
- **J[cas] {offset}** : va à l'adresse {offset} si la condition [cas] est exacte. [cas] est une condition relative aux « flags » (drapeaux), elle doit être traduite par « Si le résultat d'une opération logique… » (je prends comme exemple « CMP {a},{b} ») :

non signé	J**A**	est supérieur (a > b), si CF=ZF=0.
	J**AE** ou **JNB** ou **JNC**	est supérieur ou égal (a => b), si CF=0.
	J**B** ou J**C**	est inférieur (a <b), si CF=1.
	J**BE**	est inférieur ou égal (a <= b), si CF=ZF=1.
signé	J**G**	est supérieur (a > b), si SF=ZF=0.
	J**GE**	est supérieur ou égal (a => b), si SF=OF.
	J**L**	est inférieur (a < b), si SF<>OF.
	J**LE**	est inférieur ou égal (a <= b), si SF<>OF et ZF=1.
égalité	J**E** ou J**Z**	est égal (a = b), si ZF = 1.
	J**NE** ou J**NZ**	est différent (a <> b), si ZF = 0.

- **LEA {destination},{source}** : écrit l'adresse de **{source}** dans **{destination}**. Équivaut à « MOV {destination}, OFFSET {source} ».
- **LDS {destination},{adresse}** : copie l'adresse **{adresse}** en 32 bits dans le registre **DS**, son **segment**, et dans **{destination}** (16 bits), son **offset**.
- **LES {destination},{adresse}** : copie l'adresse **{adresse}** en 32 bits dans le registre **ES**, son **segment**, et dans **{destination}** (16 bits), son **offset**.
- **LFS {destination},{adresse}** : copie l'adresse **{adresse}** en 32 bits dans le registre **FS**, son **segment**, et dans **{destination}** (16 bits), son **offset**.
- **LGS {destination},{adresse}** : copie l'adresse **{adresse}** en 32 bits dans le registre **GS**, son **segment**, et dans **{destination}** (16 bits), son **offset**.
- **LSS {destination},{adresse}** : copie l'adresse **{adresse}** en 32 bits dans le registre **SS**, son **segment**, et dans **{destination}** (16 bits), son **offset**.
- **LODS[B/D/W]** : copie l'octet/le mot/le double mot ES:EDI dans AL/AX/EAX (instruction inverse de STOS[B/W/D]).
- **MOV {dst},{src}** : copie la valeur {src} dans {dst}.
- **MOVS[B/D/W]** : copie l'octet/le mot/le double mot DS:ESI dans ES:EDI.
- **MOVZX {dst},{src}** : étend à 32 bits le nombre contenu dans {src} (8 bits) et transfère le résultat dans {dst} (16 ou 32 bits).
  Exemple: MOVZX ax,al : Efface la partie haute de AX (AH).
- **MUL {source}** : multiplie la destination explicite par **{source}**, les deux nombres sont considérés comme non signés. Utiliser « JC {adresse} » pour tester le débordement. La destination est fonction de la taille de source :
  - 8 bits : la destination est **AX** (le multiplicateur est AL) ;
  - 16 bits : la destination est **DX:AX**, c'est-à-dire AX contient la partie basse et DX la partie haute (le multiplicateur est AX) ;
  - 32 bits : la destination est **EDX:EAX**, c'est-à-dire EAX contient la partie basse et EDX la partie haute (le multiplicateur est EAX).
- **NOT {destination}** : inverse les bits de {destination} :
  NOT 1 -> 0
  NOT 0 -> 1
  Équivalant en Turbo Pascal : « {destination} := not {destination} ».
  Équivalant en C : « {destion} = ~{destination} ».
  Voir aussi **AND**, **OR** et **XOR**.
- **OR {destination}, {masque}** : applique un « OU logique » à **{destination}** par **{masque}**. Tout bit de {destination} est mis à 1 si le bit correspondant de {masque} vaut 1, et laissé inchangé si le bit correspondant vaut 0.

0 OR 0 -> 0
0 OR 1 -> 1
1 OR 0 -> 1
1 OR 1 -> 1
Équivalant en Turbo Pascal : « {destination} := {destination} or {masque}; ».
Équivalent en C : « {destion} |= {masque} ».
Voir aussi **AND**, **NOT** et **XOR**.

- **OUT {source},{port}** : écrit la valeur {source} (8 bits) sur le port **{port}** (16 bits). Le seul registre autorisé pour **{port}** est DX. Voir aussi **IN**.
- **PUSH {valeur}** : met une [valeur] dans la pile.
- **POP {registre}** : sort une valeur de la pile et la stocke dans un [registre].
- **REP {instruction}**: répète l'instruction [instruction] ECX fois.
- **RET {valeur}** : quitte la procédure en cours. Si des paramètres ont été envoyés au CALL, [xxxx] est le nombre d'octets envoyés qui sont à sortir de la pile. Voir aussi **IRET**.
- **SHL {registre},{valeur}** : décalage binaire du **{registre}** de **{valeur}** vers la gauche (L = Left), les bits apparaissant à droite sont complétés par des zéros. Exemple : « mov al, 3; shl al,2 » donne al = 12, car 3 = 0011 et son décalage 1100. En fait décaler de X revient à multiplier par 2^X (3*2^2 = 3*4 = 3*2*2 = 12).
- **SHR {registre},{valeur}** : décalage binaire du **{registre}** de **{valeur}** vers la droite (R = Right), les bits apparaissant à gauche sont complétés par des zéros. Exemple : « mov al, 12; shr al,2 » donne al = 3, car 12 = 1100 et son décalage 0011. En fait décaler de X revient à diviser par 2^X (12/(2^2) = 12/4 = 3).
- **STOS[B/D/W]** : copie AL/AX/EAX dans l'octet/le mot/le double mot ES:EDI (inverse de LODS[B/W/D]).
- **SCAS[B/D/W]** : compare AL/AX/EAX à l'octet/le mot/le double mot ES:EDI (permet de rechercher une valeur dans une chaine de caractères).
- **TEST {source},{masque}** : teste si les bits **{masque}** de **{source}** sont posés ou non, et modifie ZF en conséquence (ZF posé si les bits de {source} sont posés, sinon ZF=0), ce qui sera exploitable avec « JZ »**ou**« JNZ » par la suite. L'instruction permet de tester un bit particulier de {source}.
  En particulier : **TEST {a},{a}** = Teste si la variable **{a}** est à zéro (pose ou non le drapeau ZF).
- **XOR {destination},{masque}** : applique un « ou exclusif » à **{destination}** par **{masque}**. Tout bit de {destination} est mis à 1 s'il diffère du bit correspondant de {masque}, et est mis à 0 s'il a la même valeur :
  0 XOR 0 -> 0
  0 XOR 1 -> 1
  1 XOR 0 -> 1
  1 XOR 1 -> 0
  (c'est le même système que pour la multiplication de nombres relatifs : (+2)x(-3) = (-6), et (-3)x(-3)=9 par exemple)
  XOR est utilisé en cryptographique, car appliquer deux fois XOR à un même nombre avec le même « masque » redonne le nombre. Exemple :
  24 xor 3 -> 27
  27 xor 3 -> 27
  XOR {a},{a} : met à la variable **{a}** à zéro, beaucoup plus rapidement que « MOV {a},0 », car XOR est une instruction de base du processeur.
  Équivalant en Turbo Pascal : « {destination} := {destination} xor {masque} ».
  Équivalant en C : « {destion} ^= {masque} ».
  Voir aussi **AND**, **NOT** et **OR**.

VI - Liste des interruptions

J'ai trouvé des informations par-ci, par-là, donc ne me demandez pas de tout détailler, je n'en sais pas plus.

- **01h**: lancement d'un programme en mode pas-à-pas (permet de le déboguer).
- **02h**: interruption non masquable.
- **03h**: erreur de rupture.
- **04h**: erreur de dépassement (par calcul, exemple : ;« mov al, 200 ; add al,140 » -> 340 > 255 !!!).
- **05h** « imprime écran », imprime une copie de l'écran en mode texte.
- **08h**: horloge tournant à 18.6 clics/seconde.
- **09h**: lecture du clavier. La touche est codée avec un 'code clavier' (scan code en anglais) traduit en code standard ASCII par l'interruption **16h**.
- **0Bh**: gestion du port COM2.
- **0Ch**: gestion du port COM1.
- **10h**: gestion de la carte vidéo.
- **11h** : liste de configuration (mémoire, nombre de ports COM, coprocesseur…).
- **12h** : taille de la mémoire basse (640 Ko maximum).
- **13h** : gestion des différents disques.
- **14h** : gestion de l'interface série (ports COM, voir les interruptions 0Bh et 0Ch).
- **15h** : manette de jeu, cassette et TopView.
- **16h** : conversion du code de la touche (lu par l'interruption **09h**) en code standard ASCII.
- **17h** : gestion de l'imprimante.
- **18h** : Rom BASIC.
- **19h** : Rountine de chargement du DOS.
- **1Ah** : gestion de l'heure réelle.
- **1Bh** : surveille la pression de la combinaison de touche 'CTRL + C'.
- **1Ch** : chronomètre clic/clic à la vitesse de l'horloge 08h : 18.6 Hz. Sa valeur est stockée à l'emplacement 0040h: 0060h.
- **1Dh** : la table d'initialisation vidéo.
- **1Eh** : la table de paramètres des disquettes.
- **1Fh** : la table des caractères graphiques.
- **20h** : l'interruption DOS : fin d'un programme au format COM (le format EXE est largement plus répandu aujourd'hui).
- **21h** : l'interruption DOS : fonctions universelles (disque dur, horloge……).
- **22h** : l'interruption DOS : adresse de fin de processus.
- **23h** : l'interruption DOS : surveille CTRL + PAUSE (ou CTRL + BREAK).
- **24h** : l'interruption DOS : Eerreur fatale d'un vecteur d'interruption.
- **25h** : l'interruption DOS : lecture directe d'un disque.
- **26h** : l'interruption DOS : écriture directe sur un disque.
- **27h** : l'interruption DOS : programmes résidents.
- **28h** : fin d'un programme restant résident en mémoire.
- **2Fh** : interruption pour plusieurs sous-programmes. Gestion du réseau, driver CD-Rom MSCDEX…

VII - Exemples concrets

VII-A - Lecture d'une touche au clavier

Principe : l'**interruption** 16h gère le clavier. Elle possède deux fonctions intéressantes : 00h, lecture d'une touche ; et 01h, vérification de la présence d'une touche dans le tampon clavier. Mais si une touche est étendue, par exemple les touches fléchées, ou les touches « page haut », « insertion », etc., la fonction 00h nous reverra un code null (00h) comme code ASCII, puis le code ASCII de la touche étendue. Sachant que le code ASCII est toujours inférieur à 128, on pourra ajouter 128 aux codes étendus pour finalement n'avoir à appeler notre future fonction de lecture d'une touche qu'une seule fois.

Solution Turbo Pascal

```
1. (uses Dos;)
2. const MasqueFlagsCF = 1; { Masque pour isoler le bit CF des flags } (1)
3.
4. function TouchPresse : Boolean;
5. var Regs: Registers; (2)
6. begin with Regs do begin
7.  AH := $01; { Fonction 01h: Vérification de la présence d'une touche }
8.  Intr ($16,Regs); (3)
9.  TouchPresse := (Flags and MasqueFlagsCF = MasqueFlagsCF); (1)
10. end end;
11.
12. function LitTouche : Char;
13. var Regs: Registers;
14. begin with Regs do begin
15.  AH := $00; { Fonction 00h : Lecture d'une touche }
16.  Intr ($16,Regs);
17.  if AL=0 then (4)
18.   LitTouche := Char(AH or 128) (5)
19.  else
20.   LitTouche := Char(AL);
21. end end;
```

Analyse :

- (1) - Le type Registre (2) nous met à disposition les **Flags**, met dans un mot d'ensemble (Word). Pour isoler le flag **CF** (bit #0), on utilisera le truc : « if Flags and MASQUE = MASQUE » où MASQUE est une puissance de 2, la puissance est la position du bit (en partant de zéro). CF étant à la position 0 (bit #0), le MASQUE est donc 1 (2^0).
- (2) - Pour appeler une interruption, Turbo Pascal nous met à disposition (par l'unité DOS) le type « Registers » qui permet d'accéder de façon virtuelle aux registres. Virtuelle, car cela ne modifie pas directement les registres, les registres sont tous envoyés comme « paramètre », puis sont lus après l'appel de « Intr » (3) (appel d'une l'interruption).
- (3) - Enfin pour appeler une interruption, on utilise la fonction « Intr » avec comme paramètre le numéro de l'interruption et les registres. Ici on utilise la fonction 00h de l'interruption 16h. Celle-ci retourne dans AL le code ASCII de la touche et AH contient le code étendu de la touche (si AL=0).
- (4) - Pour savoir si une touche est étendue, on vérifie que AL = #0.
- (5) - Pour ajouter 128 à la valeur de la touche étendue, on peut également faire « OR 128 », cela pose dans tous les cas le bit #7 de AH. Ça ne fonctionne que pour les puissances de 2 (1, 2, 4, 8, 16, 32, 64, 128, 256…).

Solution Assembleur

```
1. function TouchPresse : boolean; assembler; asm
2.  mov ah,01h { Fonction 01h = Vérifit qu'une touche soit disponible }
3.  int 16h { Appelle l'interruption clavier }
4.  mov al,1 { Une touche est présente (TRUE) }
5.  jnz @PasVide (1)
```

Solution Assembleur
```
6.  xor al,al { Pas de touche (FALSE) }
7. @PasVide:
8. end;
9.
10. function LitTouche : char; assembler; asm
11.   xor ah,ah { Fonction 00h = Lit une touche du clavier }
12.   int 16h { Appelle l'interruption clavier }
13.   or al,al { Est-ce une touche étendue ? (AL=0) } (2)
14.   jz @Etendue { Ouais -> AL = code étendu (AH) + 128 } (2)
15.   ret { Touche standard -> on se tire }
16.
17. @Etendue:
18.   mov al,ah { AL = Code ASCII étendu }
19.   or al,128 { Ajoute 128 à celui-ci pour le distinguer }
20. end;
```

Analyse :

- (1) - La fonction 01h de l'interruption 16h renvoie la présence d'une touche par le flag ZF. Pour le tester, on utilise JNZ : ZF=0, ne fait rien; ZF=1 : Saute !
- (2) - La fonction 00h de l'interruption 16h renvoie le code ASCII de la touche dans le registre AL. Si celui-ci vaut 0, alors la touche est étendue, et le code est stocké dans AH. Pour savoir si AL=0, on peut faire « cmp al,0; jz @Saut », mais il est plus rapide de faire le test par « or ». Celui-ci modifie le flag ZF : Si AL=0, ZF=1 (ZF = Zero Flag !); si AL<>0, ZF=0.

VII-B - Effacement de l'écran dans le mode VGA (320x200 pixels en 256 couleurs)

Dans le mode VGA, la mémoire vidéo est placée à l'adresse $A000:$0000, et prend 320*200=64 000 octets. Le but du jeu est de remplir ces 64 000 octets avec une couleur précise, c'est-à-dire une valeur codée sur un octet.

Solution en Turbo Pascal
```
1. fillchar (Mem[$A000:0],64000,Couleur); { Couleur étant un "Byte", un octet }
```

Cela fonctionne très bien, mais Turbo Pascal étant développé pour fonctionner avec les 286, processeur en 16 bits, fillchar fonctionnera donc en 16 bits (voire en 8 bits, je ne sais pas trop). On va donc l'optimiser en passant écrivant dans la mémoire avec les instructions 32 bits.

Solution Assembleur (intégrée dans un programme Turbo Pascal, première version, dite "éducative")
```
1. Procedure EffaceEcran (Coul: byte); begin
2. asm (1)
3. mov ax,0A000h (2)
4. mov es,ax { ES = A000h }
5. mov di,0 { Maintenant ES:DI pointe sur l'adresse A000:0000 }
6. mov al,[coul] { On lit la couleur passée en paramètre } (3)
7. mov cx,64000 { 64 000 octets à remplir}
8. rep stosb { Ecrit CX fois l'octet AL à l'adresse ES:DI }
9. end; (1)
10. end;
```

Analyse :

- (1) - En turbo Pascal, on peut intégrer de l'assembleur n'importe où en tapant « asm { les instructions } end; ».
- (2) - Les nombres hexadécimaux débutant par des lettres peuvent être confondus avec des noms de variables, il faut alors rajouter un zéro en préfixe.
- (3) - Pour lire les paramètres, rien de plus simple : taper son nom (les crochets ne sont pas obligatoires).

Solution Assembleur (version optimisée 32 bits)

```
1. Procedure EffaceEcran (Coul: byte); Assembler; asm (1)
2. mov ax,0A000h
3. mov es,ax
4. { xor edi, edi } (2) db 66h; xor di,di (3) { ES:DI pointe sur l'adresse A000:0000 }
5. mov al,[coul] { On lit la couleur passée en paramètre }
6. mov ah, al { Copie AL dans AH, donc AX = deux fois la couleur } (4)
7. mov bx, ax { Sauve AX dans le registre BX } (4)
8. { shl eax,16 } db 66h; shl ax,16 { Déplace les deux octets de poids faible de EAX dans les
 octets de poids fort } (4)
9. mov ax, bx { Relit AX du registre BX } (4)
10. mov cx,64000/4 { 16 000 doubles mots à remplir } (5)
11. {rep stosd } db 66h; rep stosw { Ecrit ECX fois le double mot EAX à l'adresse ES:EDI } (6)
12. end; (1)
```

Analyse :

- (1) - Encore mieux, en turbo Pascal on peut déclarer une procédure qui n'utilise que l'assembleur, ce qui évite d'avoir à taper « begin » et « end; » inutiles.
- (2) - Pour mettre un registre à zéro, la méthode la plus rapide est « xor reg,reg », car XOR est une fonction de base du processeur.
- (3) - Turbo Pascal 7 ne connaissant pas les instructions 32 bits, on peut toujours les écrire en langage machine. Pour avoir le code machine d'une instruction, on peut compiler un programme en assembleur ne contenant que cette instruction en demandant au compilateur de créer un listing (avec le paramètre « /Z » pour TASM 1.0 et 2.0), c'est-à-dire un fichier contenant le code assembleur décomposé, interprété et traduit en code machine. Sinon, il existe aussi des programmes effectuant la conversion, mais je n'en connais pas directement. Pour revenir à notre « xor edi, edi », l'équivalant en code machine est « 66 33 FF », or « xor di,di » est équivalant à « 33 FF », il suffit donc de rajouter le code « 66h » en préfixe, préfixe de nombreuses instructions 32 bits en fait.
- (4) - En 32 bits, l'instruction « STOSD » écrira EAX et non AL, il faut donc répéter « Coul » quatre fois dans EAX. On commence par l'écrire dans AL, puis le copier dans AH, et enfin un « truc » pour le copier dans EAX. Le truc est en fait de décaler les octets de EAX de 16 bits vers la gauche à l'aide de l'instruction SHL, puis recopier l'ancienne valeur de AX dans AX (qui est la partie basse de EAX pour souvenir !). Remarque : l'instruction « SHL » nécessite de compiler en mode 286/287, on peut rajouter « {$G+} » au début du programme pour ce fait.
- (5) - STOSD écrit des doubles mots (4 octets), or nous voulons remplir 64 000 octets. Il nous faudra donc écrire 64 000/4 fois EAX. Remarque : pour STOSW, on divisera par 2 tout simplement :-)
- (6) - Une fois de plus, on peut étendre une instruction 16 bits à son équivalant en 32 bits en rajoutant le préfixe « 66h ». En réalité « REP STOSW » est « F3 AB » (F3 = « REP »), et « REP STOSD » est « F3 66 AB ». Mais le processeur est tolérant, et le code est d'autant plus clair avec « db 66h; stosw », alors pourquoi s'en priver ?