COURS D'ASSEMBLEUR SUR ATARI ST
SERIE NUMERO 1

SOMMAIRE

COURS NUMÉRO 1

Ce cours d'assembleur pourra sembler réduit de par sa taille. Il ne l'est cependant pas par son contenu! L'assembleur est le langage le plus simple qui soit, pourvu qu'il soit expliqué simplement ce qui n'est malheureusement pas souvent le cas. C'est un peu le pari de ces cours: vous apprendre ce qu'est l'assembleur en une quinzaine, pas plus. De toutes façons, à part grossir la sauce avec du blabla inutile, je ne vois pas très bien comment faire pour que les cours durent plus de 15 jours.

Evidement, vous comprenez bien que les cours payants ont eux, tout à fait intérêt à faire durer le plaisir et à vous faire croire que c'est très très dur à comprendre et qu'il va falloir au moins 2568478 leçons si vous voulez vous en sortir!!!

Ce cours est destiné au débutant, il est composé de plusieurs parties relativement courtes mais dans lesquelles TOUT EST IMPORTANT.

PRÉSENTATION ET AVERTISSEMENTS

Pour programmer en ASM, plusieurs habitudes sont nécessaires. Autant les prendre dès le début car, très vite, ce qui apparaissait comme de petits problèmes peut tourner à la catastrophe.

Tout d'abord avoir de l'ordre au niveau disquette:

Les sauvegardes sont très nombreuses et c'est vite la pagaille. Travailler avec soins: commentaires clairs et précis sur les listings, éviter les soit disant 'trucs' qu'on ne comprendra plus au bout de 3 jours, mettez quelques lignes explicatives au début du listing.

Au niveau outil, je conseille DEVPAC pour sa facilité d'emploi, et sa doc qui, bien qu'en Anglais et suffisamment claire pour que l'on y retrouve facilement les informations.

Si vous avez un 1040 (ou plus), n'hésitez pas à demander au niveau des 'préférences' de GENST, un chargement automatique de MONST, le débugger.

Pour ce qui est des livres de chevet (et de travail), il faut bien sur faire la différence entre 2 types d'ouvrages: ceux relatifs au 68000 Motorola et ceux relatifs à l'ATARI.

Il faut ABSOLUMENT un ouvrage de chaque. Attention, pour celui relatif au 68000, il ne doit pas s'agir d'un ouvrage de vulgarisation, mais d'un ouvrage technique, qui vous semblera certainement incompréhensible au début.

Par exemple documentation du fabricant de microprocesseur (MOTOROLA ou THOMSON pour la France). Cherchez du côté des vendeurs de composants électroniques plutôt que dans les magasins de micro-ordinateurs. En désespoir de cause,orientez-vous vers "Mise en oeuvre du 68000" aux éditions SYBEX.

Une remarque qui devra IMPERATIVEMENT guider votre choix:

Le vocabulaire informatique est composé en très grande partie d'abréviations. Or ce sont des abréviations de termes anglais. Il est donc impératif que l'ouvrage sur le 68000 donne la signification de chacune des abréviations, signification en Anglais et traduction en Français. Attention de très nombreux ouvrages ne donnent que la traduction, or autant il est difficile de se souvenir de la signification de termes tels que DATCK, BG, BGACK, MMU ou MFP, autant leurs fonctions et clairs et surtout facilement mémorisable si on connait la signification de ces abréviations dans la langue originale, la traduction coulant de source!

Pour l'ouvrage relatif au ST, le must consiste à se procurer chez ATARI la DOCumentation officielle pour les Développeurs. Sinon,"la Bible" ou "le Livre du développeur" chez Micro Application, même s'il y subsiste quelques erreurs, est un excellent palliatif.

A part cela, n'achetez aucun autre ouvrage : "le livre du GEM", "Graphismes en ASM", "cours d'ASM" etc, ne seront que des gouffres pour votre porte-monnaie et ne vous apporteront rien.

Si, après ces achats il vous reste quelque argent, je ne peux que vous conseiller

très fortement l'achat d'une calculatrice possédant les opérations logiques (AND, OR, XOR...) et les conversions entre bases utilisées en informatique (binaire, hexadécimal...).

MÉTHODE DE PENSÉE D'UN ORDINATEUR

En France nous roulons à droite. C'est simlement dans les moeurs, et tout le monde s'en sort bien. Imaginons la conduite en Angleterre. Pour un Français il existe en fait 3 solutions:

1) On ne lui dit rien sur ce type de conduite : C'est avantageux dans le sens ou notre conducteur part tout de suite sur la route, mais bien sûr le premier carrefour risque de lui être fatal.

2) On lui r,-apprend ... conduire de A ... Z : C'est long, on a l'impression de perdre son temps, mais on limite presque totalement les risques d'accidents.

3) On dit simplement au conducteur: Attention, ici on roule à gauche. Celui-ci sait conduire à droite, en le prévenant il fera attention et s'en sortira. Avantage: c'est rapide, inconvénient: un simple relachement et c'est l'accident.

Programmer, c'est comme vouloir conduire à gauche. Il suffit de penser, mais pas de penser comme nous, mais comme la machine.

Conscient de votre volonté d'aller vite, c'est la méthode 3 que nous allons utiliser, mais attention au relâchement.

Un dernier conseil avant de vous laisser aborder le premier cours à proprement parler: l'assembleur plus que tout autre langage, est assimilable à une construction en Lego. Une énorme construction en Lego n'est pourtant qu'un assemblage de petites briques. Assembler 2 briques et passer 1 ou 2 heures pour étudier cet assemblage peut paraître inutile. Pourtant c'est ce que nous allons faire: il y a peu de choses à apprendre mais elles sont très importantes. On ne le répétera jamais assez: ce ne sera pas quand notre château de Lego d'un mètre cinquante commencera à s'écrouler qu'il faudra se dire "merde ,mes 2 petites briques du début ,taient peut être mal fixées", car à ce moment-là, alors qu'une machine accepterait de tout reprendre dès le début, il y a 99% de chances pour que votre expérience en ASM s'arrête là, ce qui serait dommage!

De même, je vous déconseille fortement la chasse aux listings! Cette pratique est très courante entre autre sur RTEL et n'amène généralement que des ennuis! Il est de TRES LOIN préférable de passer pour un con parce qu'on ne sait pas faire un scrolling plutôt que de frimer alors qu'on a juste recopié un source que nous a donné un copain! A ce petit jeu là, il y a des gagnants en basic, en C ou en Pascal mais jamais en assembleur, car lorsque vous commencerez à vouloir coller des sources entre eux et que ça ne marchera pas, vous serez TOTALEMENT incapable de comprendre pourquoi, et il sera trop tard pour apprendre et vous abandonnerez. Et ne dites pas non, regarder plutôt 6 mois en arrière sur RTEL et souvenez vous de ceux qui faisaient alors de l'ASM, ils ont presque tous abandonnés! N'oubliez pas non plus une différence fondamentale entre un langage quelqu'il soit et l'assembleur: Il faut environ 6 mois pour apprendre le C ou le Pascal. Ensuite le temps sera passé à produire de bons algorithmes, et à taper les programmes.

En assembleur il en est tout autrement. En un mois maximum le 68000 ne devrait plus avoir de secret pour vous, par contre tout le temps qui suivra devra être consacr, à faire des recherches plus ou moins évidentes sur des 'trucs' à réaliser plus vite, avec plus de couleurs etc... Un programmeur en BASIC ou en C recherche des sources pour travailler. Pas un programmeur en assembleur! Le programmeur en assembleur VA FAIRE les routines! Typiquement on va demander à un programmeur en C de faire un programme et le programmeur en C va demander au programmeur en assembleur de réaliser la ou les routines soi-disant infaisables! Et bien sur pour ces routines, pas de sources de distribuées!!!! Ce que nous apprendrons donc ici, c'est à programmer comme des vrais! A chercher, à comprendre afin de pouvoir par la suite chercher tout seul.

Si vous vous attendez à trouver dans ce cours des sources entiers de scrolling, de lectures de digits ou de gestion de souris sans le GEM, vous faites fausse route! Retourner au basic que vous n'auriez jamais dû quitter; Vous resterez a tout jamais ce que l'on appelle un lamer dans les démos, celui qui recopie mais ne comprend rien.

Si par contre vous voulez savoir, alors accrochez vous car les infos sont rares mais quel plaisir lorsqu'après plusieurs nuits blanches vous verrez apparaître votre premier scrolling dont vous pourrez dire : "c'est moa qui l'ai fait!!!", et là ce sera vrai!!

Dans ce cours nous étudierons le 68000 mais également les particularités du ST: les interruptions par le MFP68901, le son (digit ou non), les manipulations graphiques, l'interface graphique Ligne A, et enfin un gros morceau, souvent critiqué mais toujours utilisé le GEM.

BON COURAGE !

COURS NUMÉRO 2:
LES CHIFFRES 'MAGIQUES'

Voyons d'abord d'une façon simple comment marche un ordinateur, en nous plaçant dans la situation suivante: nous devons fournir des messages à une personne dont nous sommes séparés (par exemple, message de nuit entre des gens éloignés).

Nous avons une lampe de poche, que nous pouvons donc allumer, ou éteindre, c'est tout. Nous pouvons donc donner 2 messages
1)la lampe est éteinte (par ex. tout va bien)
2)la lampe est allumée (par ex. vla les flics!)

Approfondissons les 2 états de la lampe:

Allumée	du courant	1
Eteinte	pas de courant	0

Les tests seront donc notés par 0 ou 1 suivant l'allumage ou non de la lampe. Comme nous sommes riches, nous achetons une 2ème lampe.

Nous avons donc 4 possibilités de message

LAMPE 1	LAMPE2
éteinte	éteinte
allumée	éteinte
éteinte	allumée
allumée	allumée

En comptant avec 3,4,5,6 ...lampes, nous nous rendons compte qu'il est possible de trouver une relation simple entre le nombre de lampes et le nombre de possibilités. Nombre de possibilités = 2 à la puissance nombre de lampes.

Si mes 4 lampes sont éteintes (0000) je suis donc à la possibilité 0. Si elles sont allumées (1111) je suis donc à la 15 (car de 0 à 15 ça fait bien 16) donc 0000-> 0 et 1111-> 15. J'ai donc un bouquin de 16 pages donnant les possibilités des 16 allumages possibles, et mon correspondant a le même. Comment faire pour lui envoyer le message de la page 13 ?

Le chiffre le plus petit étant à droite (on note les chiffre dans l'ordre centaines, dizaines, unités), plaçons les lampes.

Lampe numéro: 4 3 2 1

a) je n'ai qu'une lampe (la 1) elle est allumée donc j'obtiens la valeur 1. (je ne peut obtenir que 0 ou 1)

b)j'ai 2 lampes (1 et 2), allumées toutes les deux, j'obtiens la 4ème possibilité . J'ai donc la valeur 3 (puisque je compte les valeurs 0,1,2 et 3, ce qui en fait bien 4). Puisque la lampe 1 vaut au maximum la valeur 1, j'en d,duis que la lampe 2 vaut à elle seule au maximum la valeur 2.

En effet lampe 1 allumée --> valeur 1
Lampe 2 allum,e --> valeur 2

Donc les 2 allumées ensemble --> valeur 3 = 4 possibilités.

La lampe 2 peut donc donner une 'augmentation' de 0 ou de 2.

Pour envoyer le message 13, il faut donc allumer la lampe 4 (valeur de 8), la lampe 3 (valeur de 4) et la 1 (valeur de 1). Nous sommes donc en train de compter en binaire.

En binaire :bi = deux car chaque chiffre ne peut prendre que 2 va-leurs (0 ou 1). Un bit peut donc être à 0 ou 1. C'est la plus petite unité informatique, car, le correspondant à qui nous envoyons des messages, c'est en fait un ordinateur. Au lieu d'allumer des lampes, nous mettons du courant sur n fil ou non. Un ordinateur 8 bits a donc 8 fil sur lesquels on met ou non du courant ! lampes.

Le 68000 Motorola est un micro-processeur 16 bits. Voici donc un 'programme' (c'est-à-dire une succession d'ordres) tel qu'il est au niveau mise ou non de courant sur les 16 fils Tout ... gauche c'est la valeur du fil 16 et à droite celle du 1.

0 = pas de courant sur le fil, 1 du courant.

Le microprocesseur est entouré de multiples tiroirs (les cases mémoire) et parmi les ordres qu'il sait exécuter il y a 'va chercher ce qu'il y a dans tel tiroir' ou bien 'va mettre ça dans tel tiroir'. Chaque tiroir est repéré par une adresse (comme chaque maison), c'est-à-dire par un numéro.

Nous allons dire au microprocesseur: va chercher ce qu'il y a au numéro 24576, ajoutes-y ce qu'il y a au numéro 24578 et mets le résultat au numéro 24580. On pourrait remplacer 'au numéro' par ' l'adresse'.

Allumons donc les 16 lampes en cons,quences, cela donne:

```
0011000000111000
0110000000000000
1101000001111000
0110000000000010
0011000111000000
0110000000000100
```

Une seule biarque s'impose, c'est la merde totale! Comment faire pour s'y retrouver avec un programme comme ça, si on oublie d'allumer une seule lampe, ça ne marche plus, et pour repérer l'erreur dans un listing pareil, bonjour la merde !!!!

On a donc la possibilité de marquer ça non pas en binaire, mais en décimal. Malheureusement la conversion n'est pas commode et de toute façon, on obtient quand m^me des grands chiffres (visuellement car leur taille en tant que nombre ne change pas, bien sûr!)

Ainsi la 3ème ligne donne 53368. On va donc convertir autrement, en séparant notre chiffres binaire en groupe de 4 bits.

REMARQUE DE VOCABULAIRE

Nous ne parlerons qu'Anglais. Toutes les abréviations en informatique sont des abréviations de mots ou d'expressions anglaises.

Les lire à la Française impose d'apprendre par coeur leur signification. En les lisant telles qu'elles DOIVENT être lues (en Anglais), ces expressions donnent d'elles mêmes leur définition.

Un des exemples est T$ qui est lu syst,matiquement T dollar ! Or, $ n'est pas, dans le cas présent, l'abréviation de dollar mais celle de string. T$ doit donc se lire ET SE DIRE T string. String signifiant 'chaîne' en Anglais, T est donc une chaîne de caractère. Evident, alors que lire T dollar ne signifie absolument rien ! Le seul intérêt c'est que ça fait marrer Douglas, le joyeux britannique qui programme avec moi!

Une unité binaire se dit donc BIT (binary digit)
4 unités forment un NIBBLE
8 unités forment un octet (que nous appellerons par son nom anglais c'est à dire BYTE).
16 unités forment un mot (WORD)
32 unités forment un mot long (LONG WORD)

Revenons donc à notre conversion en groupant nos 16 lampes (donc notre WORD) en groupes de 4 (donc en NIBBLE)

0011 0000 0011 1000

Ces 4 nibbles forment notre premier word. Comptons dons les valeur possibles pour un seul nibble.

état du nibble	0000	valeur 0
	0001	valeur 1
	0010	valeur 2
	0011	valeur 3
	0100	valeur 4
	0101	valeur 5
	etc.	
	1010	valeur 10

STOP ça va plus ! 10 c'est 1 et 0 or on les a déjà utilisés! Ben oui mais à part 0,1,2,3,4,5,6,7,8,9 on n'a pas grand chose à notre disposition... Ben si, y'a l'alphabet ! On va donc écrire 10 avec A, 11 avec B, 12 avec C, 13/D, 14/E et 15 avec F. Il y a donc 16 chiffres dans notre nouveau syst˜me (de 0 à F). 'D,c' signifiant 10 et 'Hex' signifiant 6 (un hexagone) donc Hex + D,c=16. Décimal = qui a 10 chiffres (0 ... 9) hexadécimal= qui en à 16!!!

Notre programme devient donc en hexadécimal:

```
$3038
$6000
$D078
$6002
$31C0
$6004
```

Plus clair mais c'est pas encore ça.

NOTE: pour différencier un nombre binaire d'un nombre décimal ou d'un hexadécimal, par convention un nombre binaire sera précédé de %, un nombre hexadécimal de $ et il n'y aura rien devant un nombre décimal. $11 ne vaut donc pas 11 en décimal, mais 17.

Réfléchissons un peu. Nous avons en fait écrit:

'Va chercher ce qu'il y a'
'à l'adresse $6000'
'ajoute y ce qu'il y a' 'à l'adresse $6002'
'met le résultat'
'à l'adresse $6004'

Le microprocesseur peut bien sûr piocher dans les milliers de cases mémoire qu'il y a dans la machine, mais en plus il en a sur lui (des petites poches en quelque sorte, dans lesquelles il stocke temporairement des 'trucs' dont il aura besoin rapidement).

Il a 17 poches: 8 dans lesquelles il peut mettre des données, et 9 dans lesquelles il peut mettre des adresses. Donnée =DATA et adresse=ADRESS, ces poches seront donc repérées par D0,D1,D2,...D7 et par A0,A1...A7 et A7' (nous verrons plus tard pourquoi c'est pas A8, et les différences entre ces types de poches).

NOTE: le phénomène de courant/pas courant et le même pour TOUS les ordinateurs actuels. Le nombre de 'poche' est propre au 68000 MOTOROLA. Il y a donc le même nombre de 'poches' sur un Amiga ou un MacIntosh puisqu'ils ont eux aussi un 68000 Motorala. Sur un PC ou un CPC, les caractéristiques (nombre de lampes allumables simultanément, nombre de 'poches'...)sont différents, mais le principe est le même. C'est allumé OU c'est éteint.

Modifions notre 'texte', qui devient donc

'déplace dans ta poche D0'
'ce que tu trouveras à l'adresse $6000'
'ajoute à ce que tu as dans ta poche D0'
'ce que tu trouveras à l'adresse $6002'
'mets le r,sultat de l'opération'
'à l'adresse $6004'

La machine est très limitée, puisque par conception, le résultat de l'opération de la 3 ème ligne ira lui même dans D0, écrasant donc ce qui s'y trouve. Pour garder la valeur qui s'y trouvait il faudrait au préalable la recopier par exemple dans la poche D1!!!

Déplacer se dit en Anglais MOVE
Ajoute se dit en Anglais ADD

Notre programme devient donc

MOVE ce qu'il y a en $6000 dans D0
ADD ce qu'il y a en $6002 à D0
MOVE ce qu'il y a maintenant dans D0 à $6004

C'est à dire:

```
MOVE   $6000,D0
ADD    $6002,D0
MOVE   D0,$6004
```

Nous venons dé,crire en clair un programme en langage machine. La différence fondamentale avec un programme dans n'importe quel autre langage, c'est que là, chaque ligne ne correspond qu'à UNE SEULE opération du microprocesseur, alors que PRINT "BONJOUR" va lui en faire faire beaucoup. Il est évident que notre BASIC n'étant qu'un traducteur 'mécanique' sa traduction a toutes les chances d'être approximative, et, bien qu'elle soit efficace, elle utilise beaucoup plus d'instructions (pour le microprocesseur) qu'il n'en faut réellement.

Il faut bien aussi avoir une pensée émue pour les premiers programmeurs du 68000 qui ont d'abord fait un programme avec des 1 et des 0, programme qui ne faisait que traduire des chiffres hexadécimaux en binaires avant de les transmettre à la machine. Il ont ensuite réalisé en hexadécimal des programmes traduisant des instructions du genre MOVE, ADD etc... en binaire.

Il suffisait ensuite de regrouper plusieurs instructions de ce type sous une autre appellation (incomprise directement par la machine) et de faire les traducteurs correspondants, et créer ainsi les langages “évolués” (PASCAL, C, BASIC ...).

Nous allons donc nous intéresser à la programmation ou plutôt à la transmission d'ordre au 68000 Motorola. Combien d'ord-

res peut-il exécuter. Uniquement 56 !!!! (avec des variantes quand même mais ça fait pas beaucoup).

Des recherches (à un niveau bien trop haut pour nous!) on en effet montrées qu'il était plus rapide d'avoir peu d'instructions faisant peu de chose chacune et donc exécutables rapidement les unes après les autres, plutôt que d'avoir beaucoup d'instructions (le microprocesseur perdant sans doute du temps à chercher celle qu'on lui a demandé de faire) ou bien des instructions complexes.

Travail à faire: relire tout ça au moins 2 fois puis se reposer l'esprit avant de lire la suite.

CONSEIL :

Ne commencez pas la suite tout de suite. Avalez parfaitement TOUT ce qui est marqué, car la compréhension du moindre détail vous servira.

Une lampe, ce n'est pas grand chose, mais une de grillée et vous comprendrez la merde que ça amène.

Là, c'est pareil. La plus petite chose incomprise et vous n'allez rien comprendre à la suite. Par contre si tout est compris, la suite sera aussi facile, et surtout aussi logique.

COURS NUMÉRO 3

Si vous avez correctement étudié les deux premières leçons, vous devez normalement avoir un peu plus d'ordre qu'au départ, et le binaire et l'hexadécimal ne doivent plus avoir de secret pour vous.

Avant de commencer je dois vous rappeler quelque chose d'essentiel: Il est tentant de réfléchir en chiffre alors que bien souvent il serait préférable de se souvenir qu'un chiffre n'est qu'une uite de bits. Ainsi imaginons un jeu dans lequel vous devez coder des données relatives à des personnages. En lisant ces données vous saurez de quel personnage il s'agit, et combien il lui reste de point de vie.

Admettons qu'il y ait 4 personnages. Combien faut-il de bits pour compter de 0 à 3 (c'est-à-dire pour avoir 4 possibilités) seulement 2 bits. Mes personnages peuvent avoir, au maximum, 63 points de vie (de 0 à 63 car à 0 ils sont morts), il me faut donc 6 bits pour coder cette vitalité. Je peux donc avoir sur un seul byte (octet) 2 choses totalement différentes: avec les bits 0 et 1 (le bit de droite c'est le bit 0, le bit le lus à gauche pour un byte est donc le 7) je code le type de mon personnage, et avec les bits 2 à 7 sa vitalité.

Ainsi le chiffre 210 en lui même ne veut rien dire. C'est le fait de le mettre en binaire: 11010010 et de penser en regroupement de bits qui va le rendre plus clair.

Séparons les 2 bits de droite:
10 ce qui fait 3 en décimal, je suis donc en présence d'un personnage de type 3.

Prélevons maintenant les 6 bits de gauche: 110100 et convertissons.

Nous obtenons 52. Nous sommes donc en présence d'un personnage de type 3, avec 52 points de vitalité.

Ceci devant maintenant être clair, passons à une explication succinte concernant la mémoire, avant d'aborder notre premier programme.

STRUCTURE DE LA MÉMOIRE

La mémoire, c'est un tube, très fin et très long Il nous faut distinguer 2 choses:

1) Ce qu'il y a dans le tube.
2) La distance par rapport au d,but du tube.

***ATTENTION:** CETTE NOTION DOIT ÊTRE PARFAITEMENT COMPRISE CAR ELLE EST PERPÉTUELLEMENT SOURCE D'ERREUR.*

Grâce à la distance, nous pourrons retrouver facilement ce que nous avons mis dans le tube. Cette distance est appelée, 'adresse'.

Le tube a un diamètre de 1 byte (octet). Lorsque je vais parler de l'adresse $6F00 (28416 en décimal), c'est un emplacement. A cet emplacement je peux mettre un byte. Si la donnée que je veux mettre tiens sur un word (donc 2 bytes car 1 word c'est bien 2 bytes accolés), cette donnée occupera l'adresse $6F00 et l'adresse $6F01.

Imaginons que je charge une image (32000 octets) à partir de l'adresse $12C52. Je vais donc boucler 32000 fois pour déposer mon image, en augmentant à chaque fois mon adresse.

Imaginons maintenant que je veuille noter cette adresse. Je vais par exemple la noter à l'adresse $6F00.

Donc si je me promène le long du tube jusqu'à l'adresse $6F00 et que je regarde à ce niveau là dans le tube, j'y vois le chiffre $12C52 codé sur un long mot (les adresses sont codées sur des longs mots). Ce chiffre occupe donc 4 emplacements de tube correspondant à $6F00, $6F01, $6F02 ,$6F03. Or, que représente ce chiffre: Une adresse, celle de mon image!!!! J'espère que c'est bien clair...

Un programme, c'est donc pour le 68000 une suite de lectures du contenu du tube. Il va y trouver des chiffres qu'il va interpréter comme des ordres (revoir le cours 2). Grâce à ces ordres, nous allons lui dire par exemple de continuer la lecture à un autre en-droit de ce tube, de revenir en arrière, de prélever le contenu du tube et d'aller le déposer autre part (toujours dans ce même tube bien

sûr) etc... Pour savoir à quel endroit le 68000 est en train de lire les ordres qu'il exécute, il y a un compteur. Comme ce compteur sert pour le programme, il est appelé Program Counter, en abrégé PC.

Le 68000 a un PC sur 24 bits, c'est-à-dire qu'il peut prendre des valeurs comprises entre 0 et 16777215. Comme chaque valeur du PC correspond à une adresse et qu'en face de cette adresse (donc dans le tube) on ne peut mettre qu'un octet, une machine équipée d'un 68000 peut donc travailler avec 16777215 octets, ce qui fait 16 Méga. A titre indicatif, le 80286 de chez Intel qui équipe les 'gros' compatibles PC, ne comporte qu'un PC sur 20 bits ce qui restreint son espace à1 méga.

A noter que la mémoire est destinée à recevoir des octets mais que ce que représente ces octets (texte, programme, image...) n'a strictement aucune importance.

PREMIER PROGRAMME

Nous allons tout de suite illustrer notre propos. Nous lançons donc GENST. Ceux qui ont un écran couleur devront le lancer en moyenne r,solution, c'est préférable pour un meilleur confort de travail.

Même si vous avez un 520, choisissez dans les 'préférences' de GENST (dans le menu 'Options') un chargement automatique de MONST (Load MONST 'YES') mettez un Tab Setting de 11 et auto-indent sur YES.

Si MONST est déjà chargé son option dans le menu 'program' doit être disponible, sinon elle est en gris. Si c'est le cas, après avoir sauv, les préférences, quitter GENST et relancez le.

Maintenant, nous allons réaliser le programme suivant:

Met le chiffre $12345678 dans le registre D0
Met le chiffre $00001012 dans le registre D1
Additionne le registre D0 avec le registre D1

Tout d'abord il faut savoir que ces ordres seront mis dans le tube, et qu'il nous faudra parfois repérer ces endroits. Pour cela nous utiliserons des étiquettes, que nous poserons à côté du tube.

Ces ,tiquettes (ou Label en Anglais) sont à inscrire tout à gauche dans notre listing alors que les instructions (ce qui est à mettre DANS le tube) seront inscrites après un espace ou mieux pour la lisibilité, après une tabulation.

Ainsi notre programme devient:

```
MOVE.L    #$12345678,D0
MOVE.L    #$00001012,D1
ADD.L     D0,D1
```

REMARQUER LE SIGNE # AVANT LES CHIFFRES. LE SIGNE $ INDIQUE QUE CES CHIFFRES SONT INSCRITS EN HEXADÉCIMAL. LE SIGNE # INDIQUE QUE C'EST LA VALEUR $12345678 QUE NOUS VOULONS METTRE DANS D0.

Si nous avions fait MOVE.L $12345678,D0, c'est la valeur se trouvant à l'adresse $12345678 que nous aurions mis en D0.

Pourquoi y a t-il .L après les MOVE et le ADD ? Nous verrons cela dans quelques minutes.

Pour le moment assemblons en maintenant appuyé [ALTERNATE] puis en appuyant sur A.

Normalement, tout s'est bien passé, ou alors c'est que vous n'avez pas scrupuleusement recopié ce 'programme'. Maintenant, débuggons notre programme, en maintenant appuyé [ALTERNATE] et en appuyant sur D.

Hop, nous nous retrouvons dans MONST qui, é,tant appelé à partir de GENST, a automatiquement chargé notre programme.

Jetons tout d'abord un coup d'oeil à ce ramassis de chiffre... En haut nous retrouvons nos registres de données D0 à D7 ainsi que nos registres d'adresses A0 à A7 avec en prime A7'. Sous les registres de données, nous voyons SR et en dessous PC. Nous pouvons remarquer que PC nous montre une adresse et la première ligne de notre programme. Le PC indique donc ce qui va être exécuté.

La fenêtre du dessous (numéro 2) montre notre programme. Sur la gauche de

cette fenêtre nous voyons les adresses. Symboliquement nous pouvons dire que la partie droite de cette fenêtre montre nos instructions dans le tube et que les chiffres de gauche nous indique l'endroit, l'adresse par rapport au début du tube.

La fenêtre de droite (la 3) donne en fait la même chose que la 2, mais avec la vision du 68000. Nous avions vu dans le cours 2 que pour la machine notre suite d'ordres n'était qu'une suite de chiffres.

Lorsque nous avons assemblé, l'assembleur a simplement converti ligne par ligne notre programme en chiffres.

Normalement dans la fenêtre 2 vous devez voir notre programme avec en face de la première instruction, une petite flèche. Regardez l'adresse de cette instruction (c'est-à-dire le chiffre de gauche, qui indique a quel endroit dans le tube se trouve cet ordre). Avec un 1040 sous TOS 1.4, cela tourne autour de $61BF0.

NOTE: LE 68000 PERMET À UN PROGRAMME DE SE PLACER N'IMPORTE OÙ. SUR CERTAINS MICRO-PROCESSEURS LES PROGRAMMES DOIVENT IMPÉRATIVEMENT TOUS SE PLACER AU MÊME ENDROIT. POUR NOUS CE N'EST PAS LE CAS, CE QUI EXPLIQUE QUE SI MON PROGRAMME EST EN $61BF0 IL N'EN EST PAS FORCEMENT DE MÊME POUR VOUS: C'EST NORMAL.

Regardez maintenant la fenêtre 3 et cherchez-y la même adresse que celle que vous avez lue dans la fenêtre 2 en face de notre première ligne de programme. Normalement si vous n'avez touché à rien cette adresse doit normalement être la première.

Vous devez y voir 203C12345678. C'est ainsi que le micro-processeur reçoit MOVE.L #$12345678,D0!!!

Retournons sur la fenêtre 2. Notons l'adresse de la seconde ligne de notre programme et soustrayons ce chiffre à l'adresse de la première ligne. Nous obtenons 6. Nous en déduisons donc que :
MOVE.L #$12345678,D0 occupe 6 octets en mémoire.

Faisons maintenant avancer notre programme. Pour cela maintenez enfoncé [CONTROL] et appuyez une fois sur Z. La petite flèche a sauté sur la seconde ligne, cette même ligne est maintenant indiquée par le PC et notre registre D0 contient maintenant la valeur $12345678. MONST indique tous les chiffres en hexadécimal, vous commencez à comprendre l'intérêt de la calculatrice...

Continuons en refaisant Control+Z. C'est maintenant la ligne 3 de notre programme qui est indiquée par le PC tandis que D1 s'est trouvé rempli par $00001012.

Continuons avec Control+Z. L'addition entre D0 et D1 s'est effectuée. Comme nous l'avions vu dans le cours 2, les possibilités sont minimes car le résultat a écrasé l'ancienne valeur de D1.

Pour réaliser D0+D1=D2 il aurait d'abord fallu transférer D1 dans D2 puis faire ADD.L D0,D2.

Dans notre cas, D1 contient maintenant la valeur $1234668A. Notre programme n'ayant pas véritablement de fin, quittons le artificiellement en tapant Control+C.

SECOND PROGRAMME

Effacer le premier programme (alternate C) et tapez le suivant:

```
MOVE.L      #$12345678,D0
MOVE.W      D0,D1
MOVE.B      D1,D2
```

Nous avons vu dans Monst que D0-D7 étaient des registres assez grands. Nous avons réussi à mettre $12345678 dans D0, ce qui donne quand même 305419896 en décimal! En effet le 68000 est un micro-processeur 16/32 bits ce qui fait que ces registres ne sont pas codés sur 16 bits mais sur 32. 32 bits, cela fait un long mot (Long Word). Dans notre premier programme, nous voulions que l'instruction MOVE agisse sur tout le registre donc sur un long mot, c'est pour cela que nous avions précisé .L après le move.

NOTE: Le vocabulaire est très important et demande un petit effort au début. Ainsi MOVE.L ne veut rien dire. Il convient de

lire ce mnémonique (c'est ainsi que sont appelé les instructions assem- bleurs) MOVE LONG. D'ailleurs l'appellation mnémonique (qui a rapport avec la mémoire, qui sert à aider la mémoire) est à rapprocher de mné-motechnique (capable d'aider la mémoire par des moyens d'association mentale qui facilitent l'acquisition et la restitution des souvenirs /CF dictionnaire Le Robert). Autant donc lire les instructions en Anglais ce qui facilitera grandement la compréhension.

Puisque notre registre D0 (comme les autres d'ailleurs) et codé sur un long mot, il contient donc 2 words c"te-à-c"te. Pour les distinguer nous appellerons celui de gauche word de poids fort et celui de droite word de poids faible. Chacun de ces words est lui même composé de 2 bytes, celui de gauche étant de poids fort et celui de droite de poids faible. De poids faible car les change- ment qu'il peut apporter à la totalité du nombre sont faible alors que les données de gauche (donc de poids fort) y apportent des variations importantes.

Assemblons notre programme et débuggons. Exécutons la première ligne. Le résultat est le même que pour le premier programme: le PC indique la seconde ligne, tandis que D0 à reçu la valeur $12345678. Maintenant exécutons la seconde ligne. Que dit-elle ?

MOVE.W D0,D1

C'est-à-dire déplacer le contenu de D0 pour le mettre dans D1. Mais attention, le déplacement doit se faire sur un word (précisé par .W après le move. Cela se lit MOVE WORD). Or les opérations se font toujours sur le poids faible. Le MOVE va donc prélever le word de poids faible de D0 pour le mettre dans le word de poids faible de D1. Celui-ci va donc recevoir $5678. Continuons en exécutant la troisième ligne. Celle-ci demande:

MOVE.B D1,D2 (move byte d1 d2)

Donc transfert du byte de poids faible de D1 vers le byte de poids faible de D2. Regarder bien les registres et les valeurs qu'ils reçoivent!

Quittez maintenant le programme avec CONTROL+C.

TROISIÈME PROGRAMME

```
MOVE.L    #$12345678,D0
MOVE.L    #$AAAAAAAA,D1
MOVE.W    D0,D1
SWAP      D0
MOVE.W    D0,D2
```

On efface le programme précédent, on tape celui-ci, on assemble puis on débugue. L'exécution de la première et de la seconde ligne ne doivent plus poser de problème.

Nous devons obtenir D0=12345678 et D1=AAAAAAAA.

Exécutons maintenant la troisième ligne. Il y a bien transfert du word de poids faible de D0 vers le word de poids faible de D1. Nous constatons que le word de poids fort de D1 N'EST PAS AFFECTE par ce transfert, et qu'il reste tout à fait indépendant du word de poids faible.4ème ligne. Ce mnémonique 'SWAP' (To swap= échanger) va échanger les 16 bits de poids faible avec les 16 bits de poids fort. D0 va donc devenir 56781234.

Dernière ligne. Transfert du word de poids faible de D0 (qui maintenant est 1234 et plus 5678) vers le word de poids faible de D2.

Nous avons vu que D0 contenait en fait 2 données et que ces données étaient totalement indépendantes. Ceci permet une grande souplesse de travail mais demande aussi une grande rigueur car si au lieu de faire MOVE.W D0,D1 j'avais juste commis une faute de frappe en tapant MOVE.L D0,D1 j'écrasais le word de poids fort de D1 et après je me serais étonné de trouver 1234 dans D1 à l'endroit où je devrais encore trouver AAAA.

Nous voyons tout de suite les énormes avantages de ce système. Nous n'avons à notre disposition que 8 'poches' de données (D0 à D7) mais si nous ne voulons garder que des words, nous pouvons en mettre 2 par poche, c'est-à-dire 16 en tout. De même si notre codage ne se fait que sur des bytes, c'est 32 bytes que nous pouvons garder (4 par poche). Cela peut paraŒtre assez évident mais par exemple sur l'Archimède, ce

n'est pas possible. Sur cette machine, un registre contient un long word ou rien!

RÉSUMÉ DE VOCABULAIRE

MOVE.L = move long

MOVE.W = move word

MOVE.B = move byte

COURS NUMÉRO 4
LES REGISTRES D'ADRESSE

Nous allons aborder maintenant les registres d'adresse. Tout comme les registres de données, ces registres sont codés sur 32 bits (un long mot). Donc à priori aucune différence, puisque le micro-processeur ne connait que des chiffres, que ceux-ci représentent des données ou des adresses, peu lui importe. C'est vrai en grande partie et d'ailleurs sur certains micro-processeurs, il n'y a qu'un ou deux registres, qui peuvent contenir indifféremment adresse ou données.

Voyons, grâce à un exemple, les différences en ce qui concerne le 68000 MOTOROLA.

Tapons donc le programme suivant, après avoir, bien sûr, effacé l'ancien, et assemblons.

```
MOVE.L  #$12345678,D0
MOVE.B  #$AA,D0
MOVE.L  #$12345678,A0
MOVE.B  #$AA,A0
MOVE.L  #$12345678,A1
MOVE.B  A1,D1
```

L'assembleur note 2 erreurs et nous les annonce par 'invalid size at line 4' et la même chose pour 'line 6'. Puisque c'est la taille et non l'opération elle-même qui semble poser problème, nous en déduisons que le MOVE vers ou à partir d'un registre d'adresse, n'est pas possible sur un byte. Rectifions donc la ligne 4 et la ligne 6 en rempla‡ant les MOVE.B par des MOVE.W et ré-assemblons.

NOTE: LORSQUE L'ASSEMBLEUR NOTE UNE ERREUR, IL DONNE LA LIGNE OÙ SE SITUE CELLE-CI. DANS CETTE NUMÉROTATION LES LIGNES VIDES SONT COMPTÉES.

Ainsi si vous aviez passé une ligne après MOVE.L #$12345678,D0 les erreurs auraient été annoncées ligne 5 et 7.

Cela fait déjà une différence puisque si vous regardez bien le programme, nous voulions réaliser une opération avec D0: Le remplir au maximum de sa taille, puis vérifier que le MOVE de la ligne 2, n'affecterait que le byte de poids faible, puis réaliser la même opération sur A0.

Impossible à priori. Tant pis, suite à notre modification, l'opération se déroulera donc sur un word au lieu d'un byte.

Débuggons notre programme. Première constatation: l'assembleur, voyant que les opérations ont lieu avec des registres d'adresse et non pas des registres de données, a automatiquement modifié les MOVE vers A0 et A1, pour les transformer en MOVEA, ce qui se lit MOVE ADDRESS.

Exécutons le programme pas-à-pas. D0 prend la valeur $12345678, puis seul son byte de poids faible est modifié, D0 prenant alors la valeur $123456AA. Ensuite A0 prend la valeur $12345678. Après la ligne suivante, l'opération affectant le word, nous devrions avoir $123400AA. Et bien pas du tout! Nous obtenons $000000AA.

Nous venons donc de voir qu'un registre d'adresse est totalement influencé (donc sur un long mot) lorsqu'il est la destination de l'opération. Qu'en est-il donc lorsqu'il en est la source ?

Continuons donc notre programme, avec le remplissage de A1 et de D1. Nous constatons par la suite que seul le word de poids faible de A1 vient écraser celui de D1.

NOTE: $AA est bien en chiffre en hexadécimal. Si vous pensiez qu'il s'agissait de simples lettres de l'alphabet, dormez 1 ou 2 jours, et reprenez le cours à la première leçon!

De tout ceci nous déduisons 2 définitions:

REGISTRE DE DONNÉES:

Chaque registre de données a une longueur de 32 bits. Les opérandes sous forme d'octet occupent les 8 bits de poids faible, les opérandes sous forme de mot, les 16 bits de poids faible et les opérandes longs, la totalité des 32 bits.

Le bit de poids le plus faible est adressé comme bit 0, le bit de poids le plus fort est adressé comme bit 31.

Lorsqu'un registre de données est utilisé soit comme opérande source, soit comme

opérande destination, seule la partie appropriée de poids faible est changée. La partie restante de poids fort n'est ni utilisée, ni modifiée.

REGISTRE D'ADRESSE:

Chaque registre a une longueur de 32 bits, et contient une adresse sur 32 bits. Les registres d'adresse n'acceptent pas une opérande dont la taille est l'octet. Par conséquent lorsqu'un registre d'adresse est utilisé comme opérande source, soit le mot de poids faible, soit l'opérande long dans sa totalité est utilisé, en fonction de la taille de l'opération.

Lorsqu'un registre d'adresse est utilisé comme destination d'opérande le registre entier est concerné, indépendamment de la taille de l'opération. Si l'opération porte sur un mot, tous les autres opérandes subissent une extension de signe sur 32 bits, avant que l'opération ne soit effectuée.

Définitions extraites du document réf EF68000 (circuit intégrés MOS THOMSON EFCIS), 45 avenue de l'Europe 78140 Velizy.

Dans ces définitions, nous remarquons un nouveau terme: opérande.

C'est le terme qui désigne la valeur utilisée dans l'opération.

Ainsi dans MOVE.W D0,D1 l'opérande source, c'est le word de poids faible de D0 alors que l'opérande destination, c'est le word de poids faible de D1. Nous savons maintenant ce qu'est le PC, un registre de données, un registre d'adresse, nous avons un peu idée de ce que nous montre les fenêtre de MONST, continuons donc à décortiquer ce fabuleux outil !

Pour observer la fenêtre de MONST, si vous n'avez pas assemblé de programme, impossible d'utiliser Alternate+D. Il vous sera répondu qu'il n'y a pas de programme en mémoire. Tapez donc Alternate+M, vous voyez MONST apparaltre, mais vous demandant quel fichier charger. Tapez ESC et nous voici tranquille pour une observation.

Nous voyons bien dans la fenêtre du haut nos registres de données et à droite nos registres d'adresse. Sous les registres de données SR puis PC. Le PC (program counter), nous savons ce que c'est, mais le SR ?

LE STATUS REGISTER

Le SR (prononcer Status Register, ce qui veut dire en Français registre d'état), est un registre codé sur un word (16 bits) et qui, comme son nom l'indique, nous renseigne sur l'état du micro-processeur.

Il est l'exemple frappant de ce que nous avons vu dans l'introduction du cours 3, à savoir qu'il est bien dangereux de traiter un ensemble de bits comme un simple chiffre, plus ou moins grand. Voyons la décomposition du Status Register.

numéro des bits 15----------0

appellation T . S . . . I2 I1 I0 . . . X N Z V C

Tout d'abord il faut savoir que certains bits du SR ne sont pas utilisés. Ils sont ici symbolisés par un point chacun.

Commençons par la description des bits de droite, en commençant par le 0.

Le bit C (C signifie Carry donc retenue en Français). Ce bit est mis à 1 lorsqu'il y a une retenue dans le bit le plus élevé (donc de poids le plus fort) de l'opérande objet, dans une opération arithmétique.

Le bit V (V signifie oVerflow donc dépassement en Français). Imaginons une addition de 2 nombres positifs, lorsque le résultat va déborder les limites du registres, on obtiendra en fait un nombre négatif à complémente à 2. En effet le fait de mettre le bit de poids le plus fort à 1 indique que le nombre est négatif. Comme ce n'est pas,dans le cas présent, le résultat recherché, on est prévenu du dépassement par le fait que le bit V est mis à 1. Il indique également, lors de divisions, que le quotient est plus grand qu'un word ou bien que nous avons un dividende trop grand.

Le bit Z (Z signifie Zéro). Il n'indique pas que le résultat est égal à 0, mais plutôt que le résultat est passé de l'autre coté de 0. En effet, ce bit est à 1 lorsqu'après une opération le bit de poids le plus fort du résultat est mis à 1, ce qui signifie que nous sommes en présence d'un nombre négatif en complément à 2. Le bit N (N signifie Negate) signifie que nous sommes en présence d'un nombre négatif.

Le bit X (X signifie eXtend donc extension). C'est un bit bien spécial qui se comporte un peu comme une retenue. Les instructions qui utilisent ce bit le précisent dans leur nom. Par exemple ADDX qui se lit add with extend est une opération d'addition prenant en compte ce bit X. Ce bit X est généralement le reflet du bit C, mais, contrairement, à celui-ci, certaines instructions ne le mo- difient pas.

Lorsque nous étudierons de plus prés les instructions du 68000, le fait que l'instruction affecte ou non tel ou tel bit sera parfois très important.

Le bit T (T signifie Trace donc suivre en Français). Lorsque ce bit est à 1, le 68000 se trouve en mode Trace. Alors là, soyez bien attentif, ce qui va suivre est primordial pour la suite des cours!!!

Le mode Trace est un mode de mise au point pour les programmes. Et oui, c'est carrément DANS le microprocesseur qu'une telle commande est insérée. A chaque fois que le 68000 exécute une instruction, il va voir dans quel état se trouve le bit T. S'il trouve ce bit à 0, il passe à la prochaine instruction. Par contre, si ce bit est à 1, le 68000 laisse de côté (temporairement) le programme principal pour se détourner vers une routine (un 'bout' de programme) qui affichera par exemple la valeur de tous les registres (D0 à D7 et A0 à A7). Imaginons qu'il faille appuyer sur une touche pour sortir de cette routine: Nous avons donc tout le temps de consulter ces valeurs. Nous appuyons sur une touche: fin de notre routine, le 68000 retourne donc au programme principal, exécute l'instruction suivante, teste le bit T, le trouve à nouveau à 1, se branche donc sur notre routine, etc... Nous avons donc un mode pas-à-pas. Or, vous avez déjà utilisé cette particularité en visualisant le déroulement des instructions avec MONST!

Tapez le programme suivant:

```
MOVE.W      #$23,D0
MOVE.W      #$15,D1
```

Assemblez et faites Alternate+D pour passer sous MONST. Appuyez une fois sur Control+Z et observez le Status Register. MONST a affiché T, indiquant ainsi que ce bit est à 1. Nous sommes donc bien en mode Trace. Quittez le programme par Control+C.

Nous arrivons maintenant à nous poser une question: Le 68000 a trouvé le bit T à 1. D'accord, il sait où est son Status register et il sait que le bit T c'est le 15ème. Mais après ? Le 68000 s'est détourné vers une routine qui dans le cas présent se trouve être une partie de MONST.

Mais comment a-t-il trouvé cette routine ? MONST est en effet un programme tout à fait ordinaire, qui a été chargé en mémoire à partir de la disquette, et qui peut être placé n'importe où dans cette mémoire.

Une première solution consisterait à toujours placer ce programme au même endroit. MOTOROLA aurait ainsi pu concevoir le 68000 en précisant: Les programmes de mise au point qui seront appelés grâce à la mise à 1 du bit T, devront commencer à l'adresse $5000. Simple, mais très gênant car il devient pratiquement impossible de faire résider plusieurs programmes en mémoire simultanément, sans courir le risque qu'ils se marchent sur les pieds!!!

Il y a pourtant une autre solution, un peu plus tordue mais en revanche beaucoup plus souple, qui consiste à charger le programme de mise au point n'importe où en mémoire, de noter l'adresse à la-quelle il se trouve, et de noter cette adresse à un endroit pré cis. Lorsque le 68000 trouvera le bit T à 1, il foncera à cet endroit prévu à l'avance par MOTOROLA, il y trouvera non pas la routine mais un long mot, adresse de cette routine, à laquelle il n'aura plus qu'à se rendre.

Cet endroit précis, où sera stocké l'adresse de la routine à exécuter lorsque le bit T sera trouvé à 1, c'est un endroit qui se situe

dans le premier kilo de mémoire (donc dans les 1024 premiers bytes). En l'occurrence pour le mode trace il s'agit de l'adresse $24.

Résumons: MONST se charge en mémoire. C'est un programme complet dont certaines routines permettent l'affichage des registres. MONST regarde l'adresse à laquelle commencent ces routines, note cette adresse puis va la mettre à l'adresse $24. Ce long mot est donc placé à l'adresse $24, $25, $26 et $27 puisque nous savons que le 'diamètre' du 'tube' mémoire n'est que d'un octet (byte). Lorsque le microprocesseur trouve le bit T à 1, il va à l'adresse $24, il y prélève un long mot qui se trouve être l'adresse des routines de MONST, et il fonce à cette adresse. ok?

Nous allons maintenant réaliser un petit programme et nous allons 'planter' votre ATARI! Tapez ce qui suit:

```
MOVE.W  #$1234,D1
MOVE.W  #$6789,D2
MOVE.W  #$1122,D3
```

Assemblez puis taper Alternate+D pour passer sous MONST. Faites une fois Control+Z. Le bit T du Status register est mis à 1, indiquant que nous sommes en mode Trace. Comme nous avons exécuté une instruction, D1 se trouve rempli avec $1234. Appuyons maintenant sur Alternate + 3.

Nous venons d'activer la fenêtre de droite (la numéro 3). Appuyons sur Alternate+A. Une demande s'affiche: nous devons indiquer quelle adresse sera la première visible dans la fenêtre. Il faut taper cette adresse en hexadécimal. Nous tapons donc...24. (pas de $ avant, MONST sait de lui-même que nous parlons en hexa) Nous voyons s'afficher l'adresse 24 en haut de la fenêtre et en face un chiffre qui est l'adresse de notre routine de MONST!

Pour moi c'est 00027086 mais comme je l'ai dit précédemment cela dépend des machines. Dans mon cas lorsque le 68000 trouve le bit T à 1, il fonce donc exécuter la routine qui se trouve en $00027086. Je vais donc modifier cette adresse! Appuyons sur Alternate+E pour passer en mode édition. Le curseur est placé sur le premier nibble de l'adresse. Tapez par exemple 11112222 ou n'importe quel autre chiffre. Repassez maintenant dans la fenêtre 1 en tapant Alternate+1.

Maintenant réfléchissons: Nous allons refaire Control+Z. Le 68000 va foncer en $24, va maintenant y trouver $11112222, et va foncer à cette adresse pour y exécuter ce qu'il va y trouver c'est-à-dire n'importe quoi! Il y a très peu de chance pour qu'il réussisse à y lire des choses cohérentes et vous indiquera une erreur.

Allez y, n'ayez pas peur, vous ne risquez pas de casser votre machine!

Hop Control+Z et, suivant les cas, vous obtenez divers messages (Illegal exception, Bus Error etc...). Quittez en faisant Control+C ou bien en dernier ressort faites un RESET.

J'espère que ce principe est TRES TRES BIEN COMPRIS. Si cela vous semble à peu près clair, relisez tout car la suite va très souvent faire référence à ce principe d'adresse dans le premier kilo, contenant l'adresse d'une routine.

La prochaine fois, nous finirons d'étudier le Status Register, en attendant je vais me prendre une petite vodka bien fralche. A la vôtre!

COURS NUMÉRO 5: SUITE DE L'ÉTUDE DU STATUS REGISTER, LES INTERRUPTIONS

Etant donné que nous avons parfaitement compris ce qui se passait dans le cas où le 68000 trouvait le bit T du Status Register à 1, c'est-à-dire tout le système d'adresse fixe à laquelle on trouve l'adresse de la routine, nous allons pouvoir continuer et en fait finir la description des autres bits de ce Status Register.

LE BIT S / SUPERVISEUR

Le 68000 peut évoluer dans 2 modes: le mode Superviseur et le mode Utilisateur. Dans le mode superviseur, nous avons accès à TOUTES les instructions du 68000 et à TOUTE la mémoire, alors qu'en mode utilisateur certaines instructions ne peuvent être employées, et l'accès à certaines parties de la mémoire est interdit.

Effectivement cela peut sembler au premier abord surprenant: Vous avez acheté une machine, c'est quand même pour pouvoir l'uti-iser dans sa totalité! Là encore, nous tombons dans le piège qui consiste à mélanger ATARI ST et 68000 MOTOROLA. Grâce à l'énorme puissance de ce micro-processeur, il est tout à fait possible d'envisager un travail multi-utilisateur.

Gonflons notre ST à 8 Mega octets, équipons le d'un énorme disque dur, et connectons le à plusieurs terminaux. Nous avons donc plusieurs claviers, plusieurs écrans, mais en revanche un seul micro-processeur, celui de l'unité centrale (dont le nom prend ici toute sa valeur) et une seule mémoire, dans laquelle tout le monde pioche à tours de bras. Là, la différenciation Superviseur / Utilisateur prend son sens. Le Superviseur, c'est le 'propriétaire' de l'unité centrale, les personnes utilisant les terminaux n'étant que des 'utilisateurs'. Le fait de ne leur autoriser qu'une partie des instructions et de la mémoire, a pour but d'éviter les plantages car si dans le cas d'une mono-utilisation, un plantage total de la machine est toujours gênant, dans le cas d'une multi-utilisation, cela relève de la catastrophe, car on ne plante plus le travail d'une seule personnes mais de plusieurs!

Le bit S du Status Register, s'il est à 0, indique que nous sommes en mode Utilisateur. A 1, il indique que nous sommes en Superviseur.

Tout comme MONST indiquait l'état Trace en indiquant T à côté du SR, il indique U ou S suivant le mode dans lequel nous nous trouvons.

Jetons un coup d'oeil en arrière sur le chapitre décrivant le brochage du 68000 (cours supplémentaire A). Nous retrouvons cette distinction au niveau des broches FC0, FC1, FC2.

Avant d'étudier les 3 bits restant du SR (I2, I1, I0), il faut savoir que le Status Register est en fait séparé en 2 octets. L'octet de poids fort (bit 8 à 15) est appelé octet superviseur, alors que l'octet de poids faible est l'octet utilisateur.

En mode utilisateur on ne peut écrire que dans l'octet utilisateur alors qu'en mode superviseur nous avons accès au word dans sa totalité.

L'octet utilisateur contenant les bits de conditions (bits X N Z V C), on l'appelle également registre des codes conditions (Condition Code Register), ou CCR.

LES BITS I2, I1 ET I0 (INTERRUPT MASK)

Ces 3 bits servent à représenter les masques d'interruption. Mais voyons tout d'abord ce qu'est une interruption. Nous avons étudié précédemment le fonctionnement lié au bit T (trace). Lorsque ce bit est positionné, le programme principal est interrompu, au profit d'une routine spéciale. C'est en quelque sorte le principe de l'interruption.

Une routine en interruption, c'est un bout de programme différent du programme principal. A intervalles réguliers ou à cause d'un élément extérieur, le 68000 va interrompre (c'est bien le mot!) le programme principal, pour aller exécuter cette routine. Lorsque celle-ci sera terminée, il y aura retour au programme principal.

L'exemple le plus simple est celui du

téléphone: Je travaille à mon bureau (c'est le programme principal) lorsque le téléphone sonne. Je détecte l'interruption, j'arrête mon travail et je décroche (exécution de l'interruption). La conversation terminée, je raccroche et je retourne à mon occupation première.

Maintenant, plus compliqué: Interruption de mon travail principal. Je décroche, mais en cours de conversation, on sonne à la porte. Là intervient le principe de la priorité d'interruption. Si la porte d'entrée à une priorité supérieure à celle du téléphone, j'interrompt la conversation téléphonique pour aller ouvrir: Il y a interruption de l'interruption. Une fois claqué la porte au 124ème marchand de balayettes de la journée je reprends le téléphone, je finis la conversation, je raccroche puis je me remets à ma tâche principale.

Par contre, si l'interruption 'porte d'entrée' a une priorité inférieure à celle du téléphone, j'attendrai d'avoir fini avec ce- lui-ci avant d'aller ouvrir.

Les 3 bits I2, I1 et I0 (Interrupt mask) permettent de définir le niveau mini d'interruption qui sera pris en cours. Comme on ne possède que 3 bits, on ne peut définir que 7 niveau, de 1 à 7 (on ne parle pas ici du niveau 0, car c'est le niveau de travail 'normal' de la machine. Si le niveau est à 0, c'est qu'il n'y a pas

d'interruption.). Ainsi, si nous avons 011 pour ces 3 bits, nous

obtenons 3 comme niveau mini. Les interruptions de niveaux 1 et 2

ne seront donc pas prises en compte. Puisque le niveau indiqué par

les 3 bits sera accepté comme niveau d'interruption, nous en dé-

duisons que si les bits sont à 111, seuls les interruptions de ni-

veau 7 seront prises en compte. Or nous voyons bien également

qu'il n'est pas possible de définir un niveau minimum de 8 par

exemple, et donc qu'il sera impossible d'empêcher une interruption de niveau 7. Ce niveau est donc dit 'non-masquable'.

Les interruptions de niveau 7 sont donc appelées NMI c'est à dire non-maskable-interrupt. A noter qu'il n'est pas possible d'opérer une sélection précise et par exemple d'autoriser les interruptions de niveaux 4, 5 et 7 et pas celles de niveau 6. Si les bits sont à 100, les interruptions de niveau 4, 5, 6 et 7 seront autorisées. Vous pouvez jeter à nou veau un coup d'oeil sur le cours annexe A. Vous retrouverez bien sur le 68000 les broches I2, I1 et I0. Une remarque cependant, ces broches sont actives à l'état bas, c'est-à-dire qu'elle indique quelque chose lorsqu'il n'y a pas de courant, à l'inverse des autres broches.

Par contre leur représentation au sein du Status Register se fait dans le bon 'sens'. Nous sommes maintenant amenés à nous poser une question similaire à celle que nous nous sommes posée lors de l'étude du mode Trace. Le 68000 re‡oit une demande d'interruption. Il compare le niveau de celle-ci à la limite fixée par les bits I du Status Register.

Si l'interruption est acceptable, il sauve le Status Register et met en place dans les bits I le niveau de l'interruption qu'il va exécuter afin de ne pas être gêné par une autre demande plus faible. Il stoppe alors l'exécution de son programme principal pour se détourner vers la routine. Une fois celle-ci terminée, il revient au programme principal. C'est bien joli, mais où a-t-il trouvé la routine en interruption ?

Et bien simplement en utilisant le même principe que pour le mode Trace. Nous avons vu que lorsque le bit T était en place, le 68000 allait voir à l'adresse $24 et qu'il y trouvait un long mot, ce long mot étant l'adresse de la routine. Pour les interruptions, le principe est le même: si c'est une interruption de niveau 4, c'est à l'adresse $70 que le 68000 trouvera un long mot, ce long mot, comme dans le cas du mode Trace étant l'adresse de la routine à exécuter. Si l'interruption est de niveau 1, c'est le long mot situé à l'adresse $64 etc... Il est bien évident que c'est au programmeur de placer ces long mots à ces adresses: On prépare une routine, on cherche son adresse de départ,

puis on note celle ci à l'endroit précis où l'on sait que le 68000 viendra la chercher.

Toutes ces adresses étant situées dans le premier kilo de mémoire de notre machine, étudions de plus près ces 1024 octets. (Vous trouverez un tableau représentant ce kilo en annexe) Pour le moment nous n'allons faire qu'y repérer les quelques éléments que nous avons déjà étudiés. Toutes ces adresses ont des numéros d'ordres, et à cause de leur fonction propre (ne faire que communiquer l'adresse d'une routine), on les appelle 'vecteurs'.

Nous retrouvons bien en $24 le vecteur 9, correspondant au mode Trace, de $64 à $7C les vecteurs correspondants aux interruptions de niveau 1 à 7. Le niveau 0, étant le niveau 'normal' de travail, n'a pas de vecteur.

Nous pouvons déjà expliquer d'autres vecteurs: Ainsi le numéro 5 (adresse $14) c'est le vecteur de division par 0. Le 68000 ne peut pas faire de division par 0. Lorsque le programme essaye, il se produit la même chose que pour le mode Trace: Ayant détecté une division par 0, le 68000 fonce à l'adresse $14, y trouve une adresse de routine et va exécuter celle-ci. Dans la plupart des cas cette routine va afficher quelques bombes à l'écran et tout bloquer. Rien ne vous empêche cependant de préparer votre propre routine et de mettre son adresse en $14. Ainsi dans un programme de math (beurkk!) cette routine peut afficher ôdivision par 0 impossibleô. Si l'utilisateur tente une telle division, inutile de faire des tests pour le prévenir de cette impossibilité, le 68000 s'en chargera tout seul.

LES AUTRES VECTEURS

Erreur bus. Nous avons vu précédemment que le 68000 utilise ce que nous appelons un bus pour recevoir ou transmettre des données. Si une erreur survient sur celui ci, il y a saut a l'adresse $8 pour trouver l'adresse de la routine qui sera alors exécutée.

Erreur d'adresse. Le 68000 ne peut accéder qu'à des adresses paires. S'il tente d'accéder à une adresse impaire, il se produit une erreur d'adresse (même principe de traitement que l'erreur bus, ou le mode Trace, vecteur, adresse etc…). Nous verrons plus tard qu'il nous sera possible d'accéder à des adresse impaires, mais avec des précautions.

Instructions illégales. Nous avons vu que le travail de l'as- sembleur consistait simplement à transformer en chiffres, ligne par ligne, notre programme. Cependant, si nous mettons en mémoire une image, celle-ci sera également placée dans le 'tube mémoire' sous forme de chiffres. La différence c'est que ces chiffres là ne veulent rien dire pour le 68000 en tant qu'instruction. Si nous ordonnons au 68000 d'aller à cette adresse (celle de l'image) il essayera de décrypter ces chiffres comme des instruc- tions, ce qui déclenchera une erreur 'instruction illégale'.

Violation de privilège. Nous avons vu que le 68000 pouvait évoluer en mode utilisateur ou en mode superviseur. On dit que l'état superviseur est l'état privilégié (ou état de plus haut privilège). Tenter d'accéder en mode utilisateur à une zone mémoire réservée au mode superviseur ou bien tenter d'exécuter une instruction privilégiée (donc utilisable uniquement en superviseur) provoquera une erreur 'violation de privilège'.

Connaltre ces différents types d'erreurs est très important. En effet la phase de mise au point est généralement longue en assembleur, surtout au début. De très nombreuses erreurs peuvent survenir, dont la cause est parfois juste sous notre nez. Le type même de l'erreur, si celle-ci est bien comprise, peut souvent suffire à orienter les recherches plus précisément et ainsi raccourcir le temps (pénible) de recherche du grain de sable qui bloque tout!

Tous les vecteurs constituant le premier kilo de mémoire ayant pour but de dérouter le programme principal vers une routine exceptionnelle, sont appelés 'vecteurs d'exceptions'.

Les vecteurs restants seront étudiés dans les séries suivantes, au fur et à mesure des besoins. Chaque chose en son temps!

Pour aujourd'hui nous nous arrêterons là. Ce fut court mais le prochain chapitre sera consacré à la pile et sera bien gros!

COURS NUMÉRO 6
LA PILE

Nous avons déjà utilisé la notion de 'tube' en ce qui concerne la mémoire. Nous pouvons y stocker différentes choses, et si nous nous rappelons l'adresse, nous pouvons revenir plus tard à cet endroit pour y récupérer ce que nous y avions déposé.

Essayez avec ce petit programme:

```
MOVE.L    #$12345678,D0
MOVE.L    D0,$493E0
MOVE.L    #0,D0
MOVE.L    $493E0,D0
```

Assemblez puis passez sous MONST. Avancez en pas à pas. D0 est d'abord rempli avec $12345678, puis le contenu de D0 est transféré à l'adresse $493E0. Notez bien qu'il n'y a pas de # devant $493E0, afin d'indiquer qu'il s'agit bien d'une adresse. Cette ligne étant exécutée, activez la fenêtre 3 ([Alternate+3]) et placez le début de celle-ci sur l'adresse $493E0 ([Alternate+A] puis tapez 493E0) Vous voyez bien 12345678 à cet endroit dans le 'tube'. Si j'ai choisit cette adresse c'est parce qu'elle se situe à 300 Kilo du début de la mémoire. Elle est donc accessible même sur un 520, et elle est suffisamment éloignée pour ne pas se trouver dans GENST ou MONST. En effet il n'y a qu'un 'tube' mémoire! Nous sommes donc en train d'écrire dans la mémoire alors qu'une partie de celle-ci est occupée par GENST et MONST! Ecrire à l'intérieur des zones occupées par ces programmes est possible, ce qui entralnera très certainement quelques plantages de ceux-ci!

Continuons en pas à pas, nous mettons D0 à 0 puis le contenu de l'adresse $493E0 (sans #) est remis dans D0.

La pile, c'est une partie de ce tube, mais que nous allons gérer d'une manière un peu différente. En effet, au lieu de placer les données dans le tube et de noter leurs adresses, nous allons cette fois-ci les empiler et pour les récupérer, les dépiler. L'avantage c'est le gain de temps (pas la peine de se demander à quelle adresse on a stocké les données) et un gain de place (si c'est pour stocker temporairement des données, inutile de conserver une portion de 'tube' juste pour ça).

Par contre l'inconvénient c'est que la gestion doit être rigoureuse. Imaginons que j'empile un premier chiffre puis 10 autres par dessus. Ensuite je dépile, mais erreur de ma part, je ne dépile que 9 chiffres! Quand je dépilerai une fois de plus, croyant retrouver le premier chiffre empilé, je récupérerai en fait le premier de la série de 10.

Nous en concluons 2 choses: d'abord que la pile est un moyen simple pour sauvegarder des données, mais ensuite que c'est une source de désagrément potentiel, tel que certains programmeurs hésite à s'en servir. C'est généralement à cause d'un manque de rigueur ce qui, je l'espère ne vous arrivera pas.

Une autre remarque: le dernier élément placé sur la pile sera toujours le premier à en sortir. C'est bien le même principe que ce-lui d'une pile d'assiettes: Regardez chez vous, il y a sûrement une énorme pile d'assiettes, mais par le simple fait que le range- ment après le lavage se fait par empilage et que mettre la table se fait par dépilage, vous mangez en fait toujours dans les mêmes assiettes... (d'où l'intérêt de bien faire la vaisselle!)

Cette structure de pile est appelée structure LIFO, c'est- à-dire Last In First Out, en Français: 'dernier entré premier sorti'. Cette structure est différente d'une autre structure fréquemment rencontrée en informatique, celle de la file, appelée aussi structure FIFO (First In First Out), la file étant similaire à une file d'attente devant un guichet: le premier dans la file sera le premier parti.

Mais concrètement, à quoi sert la pile? Nous allons le voir avec un exemple. Tapez le programme suivant:

```
        MOVE.L #$12345678,D0
        MOVE.L #$BD88,D1
        MOVE.L #$BD88,A0
        BSR AJOUTE
        MOVE.L #0,D0
        MOVE.L D2,D0

AJOUTE  MOVE.L #$11112222,D2
        ADD.L D1,D2
        RTS
```

Première remarque: ce programme diffère des précédents par le fait que nous utilisons une étiquette, un label qui se nomme AJOUTE.

Ce mot, 'AJOUTE', doit se trouver tout à gauche, contre le bord de la fenêtre de l'éditeur. Ce n'est pas quelque chose à placer DANS le tube mais bien une marque A COTE du tube.

Autre remarque, les listings en assembleur, contrairement aux listings dans les autres langages sont assez libres au niveau présentation. Il est tout a fait possible de passer des lignes, ce qui est fait ici pour séparer les 2 parties. Les sources assembleur sont bien souvent très longs, et même si cela fait perdre quelques lignes, espacer les modules permet de s'y retrouver plus facilement.

Assemblons puis débuggons. Avançons pas à pas avec Control+Z. Les 3 premières lignes nous sont familières mais pas la quatrième. Celle-ci se lit BRANCH SUB ROUTINE AJOUTE, c'est- à-dire branche- ment à une subroutine nommée AJOUTE. Pour préciser vers quelle subroutine on désire se diriger, son étiquette est précisée. Ici en l'occurrence c'est AJOUTE mais le nom importe peu. Il est tout a fait possible de mettre des noms assez longs et je ne peux que vous conseiller d'éviter dans vos listings les noms du genre X Y, Z ou encore AX1 etc... qui sont quand même moins explicites que DEBUT_IMAGE, NEW_PALETTE ou bien END_GAME.

Maintenant soyez très attentifs: à la lecture de cette instruction de nombreuses choses vont se passer. L'ordre demande donc au 68000 de poursuivre la lecture de ses instructions dans un sous programme dont le début se situe dans le tube, en face de l'étiquette AJOUTE. Cependant il s'agit bien ici d'un sous-programme. Ceci suppose qu'une fois terminé, le 68000 remontera pour exécuter la ligne qui suit BSR AJOUTE, en l'occurrence MOVE.L #0,D0.

Question: comment le 68000 saura-t-il où remonter? En effet le propre d'une subroutine est de pouvoir être appelée plusieurs fois et de plusieurs endroits différents et de pouvoir à chaque fois revenir à l'endroit même qui l'a appelé.

Eh bien le 68000 va justement utiliser la pile pour noter ce lieu de retour. Cette pile a bien sur une adresse, où se trouve-t-elle notée? En A7. Et oui, ce registre un peu spécial correspond à la pile.

Mais A7' alors? Et bien c'est aussi une pile, mais réservée au mode Superviseur. Ainsi si nous faisions tourner conjointement 2 programmes, l'un en mode utilisateur et l'autre en superviseur, chacun aurait sa pile.

Avant d'exécuter la ligne BSR AJOUTE, observons attentivement les registres d'adresses et les registres de données. Nous avons vu que les registres, qu'ils soient de données ou d'adresse, peuvent contenir des nombres codés sur 32 bits. Nous avons vu aussi qu'il existait 2 sortes de nombres pour la machine: ceux se trouvant à l'intérieur du 'tube' et ceux ce trouvant à l'extérieur, CONTRE ce tube, et indiquant une sorte de distance par rapport au début de celui-ci.

Ce second type de nombre est appelé adresse. Or il est tout à fait possible de stocker un nombre représentant une adresse dans un registre de données (D0-D7). Imaginons maintenant que nous devions stocker le score d'un joueur dans le jeu que nous programmons. Ce score va par exemple être placé dans la mémoire (dans le 'tube') à l'adresse $80792.

Mais que se passera-t-il si nous transférons cette adresse pour l'utiliser grâce à A1 par exemple? et bien A1 va prendre la valeur $80792. C'est bien joli, mais ce qui nous intéresse, ce n'est pas ça! Ce que nous voulons modifier, vérifier etc.. c'est ce qu'il y a DANS le tube à cette adresse.

Et bien notre débugger anticipe un peu cette demande. En effet, partant du principe que les nombres stockés en D0-D7 ou A0-A6 peuvent représenter des valeurs d'adresses, il indique à c“té des re- gistres, ce qu'il y a dans le tube, à l'adresse indiquée dans le registre.

En ce qui concerne les registres de données, MONST affiche à leur droite la valeur de 8 octets se trouvant dans le tube à l'adresse indiquée dans le registre. Pour les registres d'adresse, ce sont 10 octets qui sont indiqués. Vous remarquez certainement qu'en face du registre D0 (qui doit contenir $12345678 si vous avez fait correctement avancer le programme), MONST n'a

affiché que des étoiles. C'est normal car le nombre $12345678 correspond à un emplacement mémoire qui se serait accessible qu'avec 305 méga de mémoire!!! MONST indique donc qu'il ne peut pas atteindre cette zone mémoire en affichant des étoiles.

Regardons maintenant D1 et A0. Les nombres situés à leur droite montrent la même chose, ce qui est normal puisque les 2 registres D1 et A0 sont remplis avec le même nombre. On dit qu'ils pointent sur l'adresse $BD88. Allons voir en mémoire histoire de vérifier l'affichage. Activez la fenêtre 3 avec Alternate+3. Celle-ci nous affiche le contenu de la mémoire, mais nous sommes loin de $BD88!

Demandons donc que cette adresse soit celle du haut de la fenêtre 3, avec Alternate+A. Tapons cette adresse (BD88). La fenêtre 3 se ré affiche avec en haut l'adresse $BD88. Dans la colonne de droite nous voyons le contenu de la mémoire, dont nous avions déj à un aperçu avec l'affichage à droite de D1 et de A0. C'est clair?

Réactivons la fenêtre 1 (alternate+1). Normalement la petite flèche doit toujours se trouvez en face du BSR AJOUTE. Noter le chiffre se trouvant dans le registre A7 (donc l'adresse de la pile) et observer bien les chiffres à droite de ce registre, tout en faisant Control+Z.

Les chiffres ont changé! D'abord le registre A7 ne contient plus le même nombre. Celui qui s'y trouve actuellement est en effet plus petit que le précédent. Notons que cette différence est de 4. L'adresse de la pile a donc été décrémentée de 4. De plus des chiffres ont été placés dans la pile (on les voit à droite du registre A7). Or, regardez bien le nombre qui est à gauche de l'instruction MOVE.L #0,D0 de notre programme, c'est- à-dire l'adresse à laquelle devra revenir le 68000 une fois la subroutine terminée: c'est bien ce nombre qui a été placé dans la pile. Il y a donc empilage de l'adresse de retour, ce qui explique également le changement d'adresse de la pile de 4. En effet une adresse est codée sur 4 octets !

NOTE: ÉTANT DONNÉ QUE NOUS PARLONS DE PILE, ON DIT PLUS SOUVENT QUE LES DONNÉES SONT MISES SUR LA PILE ET MOINS SOUVENT DANS LA PILE.

Continuons notre programme avec Control+Z. Nous sommes maintenant dans la sous-routine. Arrêtez juste avant RTS. C'est cette instruction qui va nous faire "remonter". Elle se lit RETURN FROM SUBROUTINE.

Observons A7 (sa valeur mais aussi le contenu du 'tube' à cette adresse) et faisons un pas (Control+Z). L'adresse de retour a été dépilée, A7 a repris son ancienne adresse et nous pointons mainte- nant sur :

```
MOVE.L #0,D0.
```

Quittez ce programme avec Control+C, effacez le et tapez celui-ci:

```
MOVE.L #$12345678,D0
MOVE.L #$AAAAAAAA,D1
BSR AJOUTE
MOVE.W D2,D3

AJOUTE MOVE.W #$EEEE,D1
MOVE.W #$1111,D2
ADD.W D1,D2
RTS
```

Assemblez puis débuggez. Avancez pas à pas: D0 prend la valeur $12345678 D1 la valeur AAAAAAAA, puis nous partons vers la subroutine AJOUTE.

Malheureusement celle-ci utilise D1 et au retour nous constatons que celui-ci ne contient plus AAAAAAAA. En effet le branchement à une subroutine ne sauve rien d'autre que l'adresse de retour, et en assembleur les variables locales et autres bidouilles de langages évolués n'existent pas! C'est donc à nous de sauver les registres, et c'est ce que nous allons faire maintenant.

NOTE: LE REGISTRE A7 CONTENANT L'ADRESSE DU SOMMET DE LA PILE (CETTE ADRESSE VARIANT BIEN SÛR AVEC L'EMPILAGE ET LE DÉPILAGE), ON PEUT CONSIDÉRER CETTE ADRESSE COMME UN DOIGT INDIQUANT PERPÉTUELLEMENT LE SOMMET DE LA PILE.

Pour cette raison le registre A7 est aussi appelé pointeur de pile. Comme toujours nous utiliserons le vocabulaire anglo-

saxon, et nous dirons Stack Pointer, en abrégé SP.

Pour cette raison et parce que l'usage en est ainsi, nous remplacerons désormais A7 par SP (qui ne se lit pas "èss-pé" mais bien STACK POINTER!!!).

Imaginons que nous voulions sauvegarder D0 à l'entrée de la subroutine: Il ne faudra pas oublier de le récupérer à la sortie! Déplaçons donc le contenu de D0 vers la pile. Essayons MOVE.L D0,SP et réfléchissons: Ceci va mettre le contenu de D0 dans A7, malheureusement ce n'est pas ce que nous voulons faire. En effet nous désirons mettre le contenu de D0 DANS le tube, à l'endroit indiqué par A7 (donc SP).

Ceci va se faire avec MOVE.L D0,(SP) ,les parenthèses indiquant que la source de l'opération c'est l'intérieur du tube.

Effacez le programme actuel et tapez le suivant:

```
MOVE.L #$12345678,D0
MOVE.L D0,(A0)
MOVE.W D0,(A1)
```

Assemblez puis comme d'habitude débuggez. D0 prend la valeur $12345678, puis D0 est transféré dans sa totalité (à cause du .L qui indique que l'opération se passe sur un mot long) à l'adresse qui est notée dans A0, ensuite le poids faible de D0 est transféré dans le tube à l'adresse notée en A1. Pour le vérifier, vous pouvez activer la fenêtre 3 et demander à placer l'adresse notée dans A0 en haut de cette fenêtre, et vous constaterez qu'effectivement la valeur de D0 se trouve bien dans le 'tube'.

Nous allons donc utiliser ce type de transfert pour sauvegarder D0

Mais réfléchissons encore un peu. MOVE.L D0,(SP) va bien placer le contenu du long mot D0 dans le tube, mais si nous voulons placer une autre valeur sur la pile, celle-ci va écraser notre première valeur car avec MOVE.L D0,(SP) l'adresse indiqué par SP (donc A7) ne va pas être modifiée, ce qui devrait être le cas.

Nous allons donc réaliser le transfert différemment (en fait nous allons encore améliorer notre vocabulaire, puisque nous allons parler maintenant de type ou de mode d'adressage).

Nous allons faire:

```
MOVE.L D0,-(SP)
```

C'est le mode d'adressage avec pré-décrémentation. Derrière ce vocabulaire pompeux se cache toute une suite d'événements. En une seul instruction, nous diminuons l'adresse du pointeur de pile de 4 (puisque dans notre exemple nous voulions transférer un long mot donc 4 octets), et nous plaçons en mémoire à cette adresse le long mot D0.

Pour récupérer D0, c'est- à-dire dépiler, il faudra faire:

```
MOVE.L D0,(SP)+
```

Comme nous avions décrémenté le pointeur de pile pour ensuite déposer D0 à cette adresse, nous récupérons donc D0 sans oublier ensuite de modifier le pointeur de pile dans l'autre sens, pour qu'il retrouve son ancienne position. Notons que dans le cas présent, et si nous nous contentons de réfléchir très sommairement, il aurait été possible de sauver D0 par MOVE.L D0,(SP) et de le récupérer par MOVE.L (SP),D0. C'est compter sans le fait que la pile est un réservoir commun à beaucoup de choses. Il faut donc de préférence jouer à chaque fois le jeu d'un empilage correct et réfléchi mais aussi d'un dépilage 'collant' parfaitement avec l'empilage précédent.

Vérifions tout cela avec l'exemple suivant:

```
MOVE.L #$12345678,D0 valeur dans D0
MOVE.W #$AAAA,D1     valeur dans D1
MOVE.L D0,-(SP)      sauve D0.L sur la pile
MOVE.W D1,-(SP)  idem D1 mais en word
MOVE.L #0,D0              remet D0 à 0
MOVE.W #0,D1                et D1 aussi
MOVE.W D1,(SP)+      récupère D1 (word)
MOVE.L D0,-(SP)                puis D0
```

Assemblez puis faites défiler ce programme pas à pas sous MONST. Notez plusieurs choses: tout d'abord des commentaires ont été ajoutés au source. Il suffit que ceux-ci soient séparés des opérandes pour que l'assembleur sache qu'il s'agit de commentaires.

Si vous désirez taper une ligne de commentaires (c'est- à-dire que sur celle-ci il n'y aura rien d'autre que ce commentaire), vous devez le faire précéder du caractère étoile ou d'un point virgule.

Seconde chose, nous avions empilé D0 puis D1, ensuite nous avons dépilé D1 puis D0. Il faut en effet bien faire attention à l'ordre et aux tailles de ce que nous empilons, afin de pouvoir dépiler les mêmes tailles, dans l'ordre inverse de l'empilage.

Voici un dernier exemple.

```
        MOVE.L #$12345678,D0
        BSR AJOUTE saut vers subroutine
        MOVE.L D0,D1 transfert

AJOUTE MOVE.L D0,-(SP) sauve d0.l sur la
pile
        MOVE.W #8,D0
        MOVE.W #4,D1
        ADD.W D0,D1
        MOVE.L (SP)+,D0
        RTS
```

Assemblez puis suivez le déroulement sous MONST en étudiant bien le déroulement. Vous voyez bien que le BSR sauve l'adresse de retour sur la pile, puis que D0 et mis par dessus pour être ensuite récupéré. Ensuite c'est l'adresse de retour qui est reprise et le programme remonte.

Maintenant, provoquons une erreur, une toute petite erreur mais qui sera fatale à notre programme. Au lieu de récupérer D0 par un MOVE.L (SP)+,D0, commettons une faute de frappe et tapons à la place MOVE.W (SP)+,D0.

Assemblez et suivez pas à pas. Au moment de la sauvegarde de D0, ce sont bien 4 octets qui vont être placés sur la pile, modifiant celle-ci d'autant. Malheureusement la récupération ne va re-modifier cette pile que de 2 octets. Au moment où l'instruction RTS va essayer de récupérer l'adresse de retour, le pointeur de pile sera faux de 2 octets par rapport à l'endroit où se trouve réellement cette adresse de retour, et celui-ci va se faire à une adresse fausse. En conclusion: prudence et rigueur!!!!!!

Nous venons donc de voir que la pile était utilisée par le 68000 pour certaines instructions, et qu'elle était bien commode comme sauvegarde.

Il est aussi possible de l'utiliser pour transmettre des données, c'est ce que nous allons voir pour conclure ce chapitre.

Problème: Notre programme principal utilise les registres A0 à A6 et D0 à D6. Il va appeler une subroutine destinée à additionner 2 nombres et à retourner le résultat dans D7. Il faudra donc utiliser 2 registres par exemple D0 et D1 pour travailler dans notre routine, et donc les sauvegarder à l'entrée de celle-ci.

Voici le début du programme.

```
        MOVE.L #$11111111,D0
        MOVE.L #$22222222,D1
        MOVE.L #$33333333,D2
        MOVE.L #$44444444,D3
        MOVE.L #$55555555,D4
        MOVE.L #$66666666,D5
        MOVE.L #$77777777,D6
```

Les 7 premiers registres sont remplis avec des valeurs bidons, juste pour nous permettre de vérifier leurs éventuelles modifications.

Maintenant il faut placer les 2 nombres que nous désirions additionner, dans un endroit tel qu'ils pourront être récupérés par la routine d'addition. Plaçons donc ces 2 nombres sur la pile.

```
        MOVE.L #$12345678,-(SP)
        MOVE.L #$00023456,-(SP)
        BSR AJOUTE  et en route !
```

Rédigeons maintenant notre subroutine, afin de suivre l'ordre de travail du 68000.

De quoi aurons nous besoin dans cette routine ? De D0 et de D1 qui vont recevoir les nombres empilés et qui vont nous servir au calcul. Il va nous falloir également un registre d'adresse. En effet, lorsque nous allons dépiler nous allons modifier le pointeur de pile, or nous venons d'effectuer un BSR le 68000 a donc empilé l'adresse de retour sur la pile, et modifier celle-ci va compromettre le retour! Nous allons donc copier l'adresse de la pile dans A0, et utiliser cette copie.

NOTE: J'AI DÉCIDÉ D'UTILISER D0, D1 ET A0 MAIS N'IMPORTE QUEL AUTRE REGISTRE AURAIT TOUT AUSSI BIEN CONVENU.

Commençons donc par sauver nos 3 registres. Cela pourrait se faire par:

```
MOVE.L D0,-(SP)
MOVE.L D1,-(SP)
MOVE.L A0,-(SP)
```

NOTE: JE RAPPELLE QUE CELA SE LIT MOVE LONG!

Mais le 68000 possède une instruction très utile dans un pareil cas, qui permet de transférer plusieurs registres d'un coup.

Nous allons donc faire:

```
MOVEM.L D0-D1/A0,-(SP)
```

Ce qui se lit: move multiple registers.

Si nous devions transférer de D0 à D5 nous aurions fait :

```
MOVEM.L D0-D5,-(SP)
```

et, pour transférer tous les registres d'un seul coup:

```
MOVEM.L D0-D7/A0-A6,-(SP)
```

Compris?

Sauvons maintenant l'adresse de la pile dans A0. Comme c'est l'adresse qu'il faut sauver et non pas le contenu, cela se fait par:

```
MOVE.L A7,A0
```

transfert du registre A7 vers A0

Maintenant nous allons récupérer les 2 nombres que nous avions empilé avant l'instruction BSR.

Imaginons ce qui s'est passé. (A ce propos je vous conseille TRES fortement de vous aider d'un papier et d'un crayon. N'hésitez pas à écrire sur ces cours. Ce sont les vôtres et je ne les réclamerai pas!

Faire un petit dessin ou de placer des pinces sur votre bureau pour vous aider à comprendre est une excellente chose. Bien souvent les manipulations de mémoire ont tendance à devenir abstraites et un petit dessin arrange bien des choses!)

Nous avons décalé de 4 octets le STACK POINTER, puis nous y avons déposé $12345678. Mais dans quel sens avons nous décalé ce SP ?

Vers le début de la mémoire, vers l'adresse 0 de notre tube puisque nous avons fait -(SP). Le pointeur de pile remonte donc le long du tube. Nous avons ensuite recommencé la même chose pour y déposer $23456. Ensuite BSR, donc même chose mais réalisé automatiquement par le 68000 afin d'y déposer l'adresse de retour (4 octets).

Est-ce tout? Non car une fois rendu dans la subroutine nous avons déposé sur la pile les registres D0, D1 et A0. Le transfert ayant été effectué sur le format long mot (MOVEM.L) nous avons transféré 3 fois 4 octets soit 12 octets (bytes).

Notre copie de A7 qui est en A0 ne pointe donc pas sur nos 2 nombres mais beaucoup plus loin. Le nombre que nous avons placé en second sur la pile est donc à 16 vers le début du tube (faites le calcul: 1BSR, + 12 bytes de sauvegarde cela fait bien 16 bytes) et le nombre placé en premier sur la pile suit son copain et se trouve donc à 20 bytes d'ici, en vertu toujours du principe de la pile: le dernier entré, c'est le premier sorti.

Nous pouvons donc dire que $23456 est à A0 décalé de 16 et que $12345678 est à A0 décalé de 20.

Pour récupérer ces 2 nombres plusieurs actions sont possibles:

1) ajouter 16 à l'adresse de A0 puis récupérer.

Une addition d'adresse se fait par ADDA (add adress). Nous faisons donc

```
ADDA.L #16,A0
```

A0 pointe donc maintenant sur $23456, récupérons donc ce nombre et profitons du mode d'adressage pour avancer l'adresse indiquée dans A0 et ainsi tout de suite être prêt pour récupérer l'autre nombre.

```
MOVE.L (A0)+,D0
```

L'adresse ayant été augmentée nous pouvons donc récupérer la suite:

```
MOVE.L (A0)+,D1
```

2) Autre méthode, utilisant un autre mode d'adressage:

La méthode précédente présente un inconvénient: après le ADDA, A0 est modifié et si nous voulions garder cette adresse, il aurait fallu le sauvegarder.

Ou bien nous aurions pu ajouter le décalage à A0, récupérer les données et ensuite retirer le décalage à A0 pour qu'il retrouve sont état de départ.

Autre méthode donc, indiquer dans l'adressage le décalage à appliquer. Cela se fait par:

```
        MOVE.L 16(A0),D0
        MOVE.L 20(A0),D1
```

Cela permet de pointer sur le 16ème octet à partir de l'adresse donnée par A0 et ensuite de pointer sur le 20ème par rapport à A0. Dans les 2 cas, A0 n'est pas modifié.

Voilà le listing complet de cet exemple.

```
        MOVE.L #$11111111,D0  initialisation de D0
        MOVE.L #$22222222,D1  idem
        MOVE.L #$33333333,D2  idem
        MOVE.L #$44444444,D3  idem
        MOVE.L #$55555555,D4  idem
        MOVE.L #$66666666,D5  idem
        MOVE.L #$77777777,D6  idem
        MOVE.L #$12345678,-(SP) passage nombre 1 dans la pile
        MOVE.L #$00023456,-(SP) passage nombre 2 dans la pile
        BSR AJOUTE  et en route !
        MOVE.L D7,D0   transfert du résultat pour voir..
```

notre subroutine:

```
        AJOUTE MOVEM.L D0-D1/A0,-(SP) sauvegarde
        MOVE.L A7,A0   copie de SP en A0
        MOVE.L 16(A0),D0  récupère 23456 et le met en D0
        MOVE.L 20(A0),D1  récupère 12345678 en D1
        ADD.L D0,D1   addition
        MOVE.L D1,D7   transfert du résultat
        MOVEM.L (SP)+,D0-D1/A0 récupération
        RTS    et retour
```

NOTE: CE PROGRAMME N'AYANT PAS DE FIN 'NORMALE', LORSQUE VOUS* SEREZ RENDU AU RETOUR DE LA SUBROUTINE C'EST- À-DIRE APRÈS LA LIGNE" MOVE.L D7,D0 ", QUITTEZLE AVEC CONTROL+C, ASSEMBLEZ ET SUIVEZ BIEN TOUT LE DÉROULEMENT.

Bien sûr, il aurait été possible de faire cela tout différemment. Par exemple nous aurions pu éviter de travailler avec A0. En effet 16(A0) et 20(A0) ne modifiant pas A0, il aurait été plus simple de faire 16(A7) et 20(A7) au lieu de recopier d'abord A7 en A0. De même il aurait été possible de transférer $23456 directement en D7 et $12345678 en D1 puis de faire ADD.L D1,D7 afin d'éviter la sauvegarde de D0 (qui aurait été inutilisée), et le transfert D1 vers D7 qui n'aurait alors pas eu lieu d'être. De même nous au- rions pu retourner le résultat par la pile au lieu de le faire par D7.

Beaucoup de variantes possibles n'est ce pas ?

Pour terminer, un petit exercice. Relancer ce petit programme et analysez PARFAITEMENT TOUT ce qui s'y passe. Quelque chose ne va pas ! Je vous aide en disant qu'il s'agit bien sûr de la pile. Cherchez et essayez de trouver comment faire pour arranger ça.

La réponse sera au début du prochain cours mais essayez d'imaginer que c'est votre programme et qu'il ne marche pas et cherchez!!!

Bon, le cours sur la pile se termine ici. Ce fut un peu long mais je pense, pas trop compliqué. Relisez le, car la pile est un truc délicat dont nous allons nous servir TRES abondamment dans le prochain cours. Si vous avez à peu près tout compris jusqu'ici il est encore temps de rattraper le temps perdu et de tout reprendre au début, car il faut avoir PARFAITEMENT tout compris et pas à peu prés!

Afin de vous remonter le moral, je vous signal que vous êtes presque à la moitié du cours...

COURS NUMERO 7

Nous abordons maintenant le septième cours de la série. La totalité du cours étant en 2 séries (enfin à l'heure où je tape ces lignes c'est ce qui est prévu!), celui-ci est le dernier de la première!

A la fin de celui-ci et si vous avez très attentivement et très scrupuleusement suivi les 6 cours précédents, vous devriez être capable d'afficher des images, sauver des fichiers etc...

Mais tout d'abord revenons à notre pile et à la question du cours précédent. Avez vous trouvé l'erreur ?

Eh bien regardez la valeur de A7 avant d'y empiler $12345678 et $23456, et comparez à la valeur à la sortie du programme. Malheur! ce n'est pas la même! Normal, si nous comptons les empilages et les dépilages, nous nous rendons compte que nous avons empilé 8 octets de plus que nous n'avons dépilé. En effet, comme nous avons récupéré nos 2 nombres en sauvegardant au préalable A7 dans A0, nous n'avons pas touché A7 au moment de la récupération. Heureusement d'ailleurs car le retour de la routine aurait été modifié!

Partant du principe de dépilage dans l'ordre inverse, il nous faut donc corriger la pile une fois revenu de la subroutine. Comme nous avons empilé en faisant -(SP) il faut ajouter pour que la pile redevienne comme avant. Ayant empilé 2 nombres de 4 octets chacuns, nous devons ajouter 8 octets à l'adresse de la pile pour la corriger comme il faut. Nous avons déj à vu comment augmenter une adresse, avec ADDA.

Il convient donc de rajouter juste après la ligne BSR AJOUTE une addition sur SP, en faisant ADDA.L #8,SP (qui se lit ADD ADRESS LONG 8 STACK POINTER)

Un appel à une subroutine en lui passant des paramètres sur la pile sera donc typiquement du genre:

```
MOVE.W #$1452,-(SP)
MOVE.L #$54854,-(SP)
MOVE.L #TRUC,-(SP)
BSR BIDOUILLE
ADDA.L #10,SP
```

Nous passons le word de valeur $1452 dans la pile (modifiée donc de 2 octets), le long mot de valeur $54854 dans la pile (modifiée de 4 octets), l'adresse repérée par le label TRUC dans la pile (modifiée de 4 octets) puis nous partons vers notre subroutine. Au retour correction de 2+4+4=10 octets du stack pointer pour revenir à l'état d'origine.

La pile possède une petite particularité. Nous avons vu dans les cours précédents que le 68000 était un micro-processeur 16/32 bits. Il lui est très difficile d'accéder à des adresses impaires. Or si nous commençons à empiler des octets et non plus uniquement des words ou des long words, le Stack Pointer peut très facilement pointer sur une adresse impaire, ce qui risque de planter notre machine.

Taper le programme suivante:

```
MOVE.L #$12345678,D0
MOVE.L D0,-(SP)
MOVE.B D0,-(SP)
MOVE.L #$AAAAAAAA,D1
```

Assemblez puis passez sous MOnst et avancez pas à pas en observant bien l'adresse du SP (donc celle visible en A7).

Nous constatons que le pointeur de pile se modifie bien de 4 lorsque nous faisons MOVE.L D0,-(SP) mais qu'il se modifie de 2 lorsque nous faisons MOVE.B D0,-(SP) alors que nous pouvions nous attendre à une modification de 1 ! Les erreurs provoqués par des adresses impaires sont donc écartées avec la pile . Merci Monsieur MOTOROLA!

NOTE: CECI EST UNE PARTICULARITÉ DES REGISTRES A7 ET A7'. SI NOUS AVIONS TRAVAILLÉ AVEC A3 PAR EXEMPLE AU LIEU DE SP, CELUI-CI AU- RAIT EU UNE ADRESSE IMPAIRE. C'EST LE TYPE D'USAGE QUI EST FAIT DE LA PILE QUI A CONDUIT LES GENS DE MOTOROLA À CRÉER CETTE DIFFÉ- RENCE.

Abordons maintenant l'ultime chapitre de cette première série:

LES 'TRAP'

Une instruction TRAP est comparable à une instruction BSR. Elle agit comme un branchement vers une routine. Cependant, contrairement à l'instruction BSR qui demande à être complétée par l'adresse, c'est- à-dire le label permettant de trouver la routine, l'instruction TRAP se contente d'un numéro. Ce numéro peut varier de 0 à 15. Lorsque le 68000 rencontre une instruction TRAP il regarde son numéro et agit en conséquence. Vous vous rappeler des tout premiers cours, dans lesquels nous avions parlé du principe utilisé par le 68000 lorsqu'il trouvait la bit T (mode trace) du SR (status register) à 1 ? Saut dans le premier kilo de mémoire (table des vecteurs d'exceptions), recherche de l'adresse $24, on regarde dans le tube à cette adresse, on y trouve un long mot, ce long mot c'est l'adresse de la routine et on fonce à cette adresse exécuter cette routine.

Et bien regardez la feuille qui donne la liste des vecteurs d'exceptions, et jetez un coup d'oeil aux vecteurs 32 à 47. Les voil à nos vecteurs TRAP !!! Lorsque le 68000 rencontre par exemple l'instruction TRAP #8, il fonce à l'adresse $0A0 pour y trouver l'adresse de la routine qu'il doit exécuter.

A priori cela semble bien compliqué pour pas grand chose! En effet il faut prévoir sa routine, la mettre en mémoire, puis placer son adresse dans le vecteur. Plus compliqué qu'un BSR, surtout que BSR REGLAGE_CLAVIER et plus parlant qu'un TRAP #5 ou un TRAP #12 !!!

Là, nous retournons encore en arrière (je vous avais bien dit que TOUT était important dans ces cours!!!!!) pour nous souvenir de la notion de mode Utilisateur et de mode Superviseur. Le Superviseur accède à toute la mémoire et à toutes les instructions, pas l'Utilisateur.

S'il s'agit d'interdire à l'Utilisateur des instructions assembleur telles que RESET, notre Utilisateur ne sera pas trop gêné par contre c'est en ce qui concerne la mémoire que tout va très sérieusement se compliquer. Voulez vous connaltre la résolution dans laquelle se trouve votre machine ? C'est facile, c'est noté à l'adresse $FF8260.

Vous voulez changer la palette de couleur ? Rien de plus simple, elle est notée en $FF8240. Imprimer un petit texte ? A l'aise, il suffit d'employer les registres de communications vers l'extérieur du chip son (étonnant n'est ce pas!). C'est situé en $FF8800 et $FF8802.

Pardon ??? Quoi ??? Vous êtes Utilisateur ??? Ah bon.... Parce que c'est gênant... Toutes ces adresses sont situées dans la zone mé- moire uniquement accessible au Superviseur.....

L'Utilisateur se trouve bien coincé et les possibilités s'en trouvent dr"lement réduites. Heureusement, les TRAP sont là !!! Grâce à ce système l'utilisateur va avoir accès à des zones qui lui sont normalement interdites. Pas directement, bien sûr, mais grâce au superviseur. Le superviseur a, en effet, fabriqué des routines qu'il a placé en mémoire et dont les adresses sont dans les vecteurs TRAP. Ces routines sont exécutés en mode superviseur et tapent à tour de bras dans les zones mémoires protégées. Lorsque l'Utilisateur veut les utiliser il les appelle par les TRAP. La protection est donc bien assurée car l'Utilisateur ne fait que déclencher une routine dont généralement il ne connalt que les paramètres à lui passer et le type de message qu'il aura en réponse. C'est de cette manière que nous pouvons accéder au système d'exploitation de notre Atari !!!

QU'EST CE QU'UN SYSTÈME D'EXPLOITATION ?

Le premier qui répond c'est GEM se prend une paire de claques. GEM, c'est l'interface utilisateur et pas le système d'exploitation.

Le système d'exploitation (ou Operating System) dans notre cas c'est TOS. La confusion entre interface Utilisateur et système d'exploitation vient du fait que certains systèmes d'exploitation intègrent également un interface utilisateur: c'est par exemple le cas sur PC avec MS DOS.

Le système d'exploitation c'est un ensemble de routine permettant d'exploiter la machine. Ces multiples routines permettent par

exemple d'afficher un caractère à l'écran d'ouvrir un fichier, de formater une piste de disquette, d'envoyer un octet sur la prise MIDI etc... En fait tous les 'trucs' de base, mais jamais de choses compliquées. Une routine du système d'exploitation ne permettra pas, par exemple, de lire le contenu d'un fichier se trouvant sur la disquette. En effet ceci demande plusieurs opérations avec à chaque fois des tests:

Ouverture du fichier: existe t-il, la disquette n'est elle pas ablmée etc... positionnement du pointeur dans le fichier: le positionnement s'est il bien passé?

Lecture: N'as t-on pas essayé de lire trop d'octets etc, etc....

Il faudra donc bien souvent plusieurs appels à des routines différentes pour réaliser ce que l'on veut.

Il est toujours possible de se passer du système d'exploitation, spécialement lorsque l'on programme en assembleur. En effet l'ensemble des routines de l'OS (abréviation de Operating System) est destiné à un usage commun, tout comme d'ailleurs les routines de l'interface Utilisateur.

Ceci explique bien souvent la ré-écriture de toutes petites parties du système afin de n'utiliser que le strict nécessaire. La routine de gestion souris du GEM par exemple doit s'occuper de la souris mais aussi du clavier, du MIDI et du joystick. Pour un jeu il peut être intéressant de ré-écrire cette routine afin de gérer uniquement le joystick et donc d'avoir une routine qui 'colle' plus au besoin.

Nous verrons beaucoup plus tard comment regarder dans le système d'exploitation afin de pouvoir par la suite réaliser soi-même ses routines. Avant cela, utilisons simplement ce système!

Nous allons donc l'appeler grâce aux TRAPs. 4 traps sont accessibles 'normalement' dans le ST:

TRAP #1 routines du GEMDOS
TRAP #2 routines du GEM
TRAP #13 routines du BIOS
TRAP #14 routines du BIOS étendu (eXtended Bios donc XBIOS)

GEMDOS =Graphic environment manager disk operating system
GEM =Graphic environment manager (se découpe par la suite enAES, VDI etc.. Un chapitre de la seconde série y sera consacrée)
BIOS =Basic Input Output System
XBIOS =Extended Basic Input Output System

Les autres vecteurs TRAP (0, 3 à 12 et 15) sont, bien entendu, actifs mais aucune routine n'y est affectée. Nous pouvons les utiliser pour peu que nous y mettions avant nos routines, ce qui sera l'objet du premier cours de la seconde série.

Nous constatons que le TRAP #1 permet d'appeler le GEMDOS. Or il n'y a pas qu'une routine GEMDOS mais une bonne quantité. De plus ces routines demandent parfois des paramètres. Comment faire pour les transmettre ? Et bien tout simplement par la pile !!!

Taper le programme suivant:

```
MOVE.W #65,-(SP)
MOVE.W #2,-(SP)
TRAP #1
ADDQ.L #4,SP
MOVE.W #7,-(SP)
TRAP #1
ADDQ.L #2,SP
MOVE.W #0,-(SP)
TRAP #1
ADDQ.L #2,SP
```

Assemblez ce programme mais ne le débuggez pas, lancez le par Alternate+ X. Vous voyez apparaltre un A sur l'écran de votre ST.

Appuyer sur une touche et hop vous revenez dans GENST! Analysons ce que nous avons fait car l à de très très nombreuses choses se sont passées, et avouons le, nous n'avons rien vu !!!!!

Tout d'abord nous avons appelé la fonction Cconout() du Gemdos. Nous avons appelé le Gemdos avec le TRAP #1, mais cette instruc- tion nous a envoyé vers un ensemble de routine, toutes appartenant au Gemdos. Pour indiquer à cette routine principale

vers quelle subroutine du Gemdos nous désirons aller, nous avons passé le numéro de cette subroutine dans la pile. Partant toujours du principe du dernier entré premier sorti, il est bien évident que ce numéro doit se trouver empilé en dernier afin de pouvoir être dépilé en premier par la routine principale de Gemdos, afin qu'elle puisse s'orienter vers la sous-routine qui nous intéresse. La fonction Cconout ayant le numéro 2, nous avons donc fait MOVE.W #2,-(SP) .(voir plus haut pour se rappeler que 2 peut très bien être codé sur un octet mais, comme nous travaillons vers la pile, il sera pris comme un word de toutes façons).

Maintenant le Gemdos ayant trouvé 2 comme paramètre, s'oriente vers cette routine au nom barbare, qui a pour fonction d'afficher un caractère sur l'écran. Une fois rendu vers cette routine, le Gemdos va chercher à savoir quel caractère afficher. C'est pour cela que nous avons placé le code ASCII de ce caractère sur la pile avec MOVE.W #65,-(SP).

NOTE: POUR L'ASSEMBLEUR, LE CODE ASCII PEUT ÊTRE REMPLACÉ PAR LA LETTRE ELLE-MÊME. NOUS AURIONS DONC PU ÉCRIRE MOVE.W #"A",-(SP) SANS OUBLIER TOUTEFOIS LES GUILLEMETS!

De retour du TRAP nous devons corriger la pile, afin d'éviter le problème qui a fait l'objet du début de ce cours. Nous avions empilé un word donc 2 octets et ensuite un autre word soit au total 4 octets. Nous allons donc ajouter 4 au SP. Nous profitons ici d'une opération d'addition plus rapide que ADDA, ADDQ qui se lit add quick. Cette addition est autorisée jusqu' à 8 inclus. Il n'est pas possible par exemple de faire

ADDQ.L #12,D1.

Ensuite nous recommençons le même genre de chose, avec la fonction 7 du GEMDOS (nommée Crawcin)qui elle n'attend aucun paramètre,c'est pourquoi nous passons juste son numéro sur la pile. Cette fonction attend un appui sur une touche. Ayant passé un paramètre sur un word, nous corrigeons au retour du TRAP la pile de 2.

Le programme se termine avec la fonction 0 du GEMDOS (Ptermo) qui libère la mémoire occupée par notre programme et le termine pour de bon. Cette routine n'attend pas de paramètre, nous ne passons dans la pile que son numéro donc correction de 2. Note: la correction de pile pour la fonction Ptermo n'est l à que par souci pédagogique. Cette fonction terminant le programme, notre dernière instruction ADDQ.L #2,SP ne sera jamais atteinte!

Plusieurs choses maintenant. D'abord ne soyez pas étonnés des noms bizarres des fonctions du GEMDOS, du Bios ou du Xbios. Ce sont les véritables noms de ces fonctions. En assembleur nous ne les utiliserons pas directement puisque l'appel se fait pas un numéro, mais en C par exemple c'est ainsi que sont appelées ces fonctions. Dans les cours d'assembleur de ST MAG (dont les vertus pédagogiques sont plus que douteuses), nous pouvons lire que les noms de ces fonctions ont été choisis au hasard et que la fonction Malloc() par exemple aurait pu s'appeler Mstroumph(). C'est ridicule!

Chacun des noms est, comme toujours en informatique, l'abréviation d'un expression anglo-saxonne qui indique concrètement le but ou la fonction. Ainsi Malloc signifie Memory Allocation, cette fonction du GEMDOS permet donc de réserver une partie de mémoire!!!

Malheureusement de nombreux ouvrages passe sur ce 'détail' et ne fournissent que l'abréviation.

Ceci n'empêche qu'il vous faut impérativement une liste de toutes les fonctions du GEMDOS, du BIOS et du XBIOS. Ces fonctions sont décrites dans le Livre du Développeur, dans la Bible mais égale- ment dans les dernières pages de la doc du GFA 3.

NOTE: DANS LA DOC DU GFA, IL MANQUE LA FONCTION GEMDOS 32 QUI PERMET DE PASSER EN SUPERVISEUR. CE MODE N'ÉTANT POUR LE MOMENT QUE D'UN INTÉRÊT LIMITÉ POUR VOUS, PAS DE PANIQUE, NOUS DÉCRIRONS TOUT CELA DANS LA SECONDE SÉRIE.

Continuons pour le moment avec des petits exemples. Affichons une phrase sur l'écran à la place d'un lettre.

Ceci va se faire avec la programme suivant:

```
MOVE.L #MESSAGE,-(SP) adresse du texte
MOVE.W #9,-(SP) numéro de la fonction
TRAP #1  appel gemdos
ADDQ.L #6,SP correction pile

* attente d'un appui sur une touche
MOVE.W #7,-(SP) numéro de la fonction
TRAP #1  appel GEMDOS
ADDQ.L #2,SP correction pile
* fin du programme

MOVE.W #0,-(SP)
TRAP #1
SECTION DATA
MESSAGE DC.B "SALUT",0
```

Une nouveauté, le passage d'une adresse. En effet la fonction 9 du gemdos demande comme paramètre l'adresse de la chalne de caractère à afficher. Nous avons donc donné MESSAGE, qui est le label, l'étiquette servant à repérer l'emplacement dans le tube où se trouve notre phrase, tout comme nous avions mis une étiquette AJOUTE pour repérer notre subroutine, dans le cours précédent.

Ce message est une suite de lettres, toutes codées sur un octets. Pour cette raison nous disons que cette chalne est une constante constituée d'octet. Nous définissons donc une constante en octets: Define Constant Byte, en abrégé DC.B Attention ceci n'est pas une instruction 68000 ! C'est simplement une notation pour l'assembleur afin de lui dire: n'essaye pas d'assembler ça comme du code normal, ce n'est qu'une constante. De même nous définissons une zone.

La fonction 9 du GEMDOS demande à ce que la phrase se termine par 0, ce qui explique sa présence à la fin.

Réalisons maintenant un programme suivant le schéma suivant:

affichage d'un texte de présentation en inverse vidéo;

ce texte demande si on veut quitter ou voir un message. Si on choisit quitter, bye bye. Sinon on affiche 'coucou' et on redemande etc…

Détaillons un peu plus, en traduisant ce programme en pseudo-code.

C'est ainsi que l'on nomme la façon de présenter un déroulement d'opération en langage clair mais dont l'organisation se rapproche déjà de la programmation.

```
AFFICHE "QUITTER (Q) OU VOIR LE MESSAGE (V) ?"
        SI REPONSE=Q VA A QUITTER
        SI REPONSE=V
                AFFICHE "COUCOU"
                RETOURNE A AFFICHE "QUITTER...."
        SI REPONSE DIFFERENTE RETOURNE A AFFICHE "QUITTER..."
```

Par commodité, ce listing se trouve sur une feuille séparée (listing numéro 1 / Cours numéro 7).

Tout d'abord affichage de la phrase qui servira de menu, avec la fonction Gemdos 9. Cette phrase se trouve à l'étiquette MENU, al-ons la voir pour la détailler. Nous remarquons tout d'abord qu'- elle commence par 27. Après avoir regardé dans une table de code ASCII, nous notons qu'il s'agit du code ASCII de la touche Escape. Nous cherchons donc d'abord à afficher Escape. Mais, comme vous le savez sûrement, ce caractère n'est pas imprimable!

Impossible de l'afficher à l'écran!

C'est tout à fait normal! en fait il n'est pas question ici d'afficher réellement un caractère, mais plut"t de faire appel à un ensemble de routines, répondant au nom de VT52. Pour appeler ces routines, il faut afficher Escape. Voyant cela le système se dit: "Tiens, on cherche à afficher Escape, c'est donc en fait que l'on cherche à appeler le VT52".

L'émulateur VT52 réagit donc, mais que doit-il faire ? et bien pour le savoir il va regarder la lettre qui suit Escape. En l'occurrence il s'agit ici de E majuscule. Regardez dans les feuilles annexes à cette série de cours, il y en a une consacrée au VT52. Nous voyons que Escape suivi de E efface l'écran, c'est donc ce qui va se passer ici.

Ensuite il était dit dans le 'cahier des charges' de notre programme, que le MENU devait être affiché en inverse vidéo.

Consultons donc la feuille sur le VT52. Nous y trouvons: Escape et 'p' minuscule = passe en écriture inverse vidéo. Juste ce qu'il nous faut! Nous remettons donc 27,"p" dans notre phrase.

TROIS REMARQUES:

Tout d'abord il faut remettre à chaque fois Escape. Faire 27,"E","p" aurait effacé l'écran puis aurait affiché p.

Seconde remarque, il faut bien faire la différence entre les let- tres majuscules et les lettres minuscules. Escape+E efface l'écran mais Escape+e active le curseur!!!

Troisième remarque, on peut représenter dans le listing une lettre par son 'caractère' ou bien par son code ASCII.

Ainsi si on veut afficher Salut, on peut écrire le listing comme ceci:TXT DC.B Salut",0 ou bien comme cela:

```
TXT DC.B 83,97,108,117,116,0
```

Il est de même possible de mélanger les données en décimal , en binaire, en hexadécimal et les codes ASCII. Par exemple ceci:
TXT DC.B 65,$42,%1000011,"D",0
affichera ABCD si on utilise cette "phrase" avec Gemdos 9.

Ceci vous sera bien utile lorsque vous chercherez à afficher des lettres difficiles à trouver sur le clavier. Pour le 'o' tréma, il est possible de faire:

```
TXT DC.B "A bient",147,"t les amis.",0
```

Note: J'espère que depuis le début, il n'y en a pas un seul à avoir lu DC.B "décébé"!!!! Je vous rappelle que cela se lit
Define Constant Byte.

Continuons l'exploration de notre programme. Notre phrase efface donc l'écran puis passe en inverse vidéo. Viens ensuite le texte lui-même:

```
QUITTER (Q) OU VOIR LE MESSAGE (V) ?
```

Ensuite une nouvelle commande VT52 pour repasser en vidéo normale, puis 2 codes ASCII qui, eux non plus, ne sont pas imprimables. Ce sont les codes de retour chariot. Le curseur va donc se retrouver tout à gauche de l'écran, une ligne plus bas. Enfin le 0 indiquant la fin de la phrase.

Une fois le 'menu' affiché, nous attendons un appui sur une touche avec la fonction Gemdos numéro 7. Cette fonction renvoi dans D0 un résultat. Ce résultat est codé sur un long mot, comme ceci:

Bits 0 à 7 code ASCII de la touche
Bits 8 à 15 mis à zéro
Bits 16 à 23 code clavier
Bits 24 à 31 Indication des touches de commutation du clavier (shifts..)

Dans notre cas nous ne nous intéresserons qu'au code ASCII de la touche enfoncée. Nous allons donc comparer le word de D0 avec chacun des codes ASCII que nous attendons, c'est à dire Q, q, V et v. Cette comparaison va se faire avec une nouvelle instruction:

Compare (CMP). Comme nous comparons un word nous notons CMP.W, que nous lisons COMPARE WORD. Nous comparons Q avec D0 (nous aurions pu marquer CMP.W #81,D0 puisque 81 est le code ASCII de Q).

Cette comparaison effectuée, il faut la tester. Nous abordons ici les possibilités de branchement dépendant d'une condition, c'est- à-dire les branchements conditionnels.

LES BRANCHEMENTS CONDITIONNELS

Chacune de ces instructions commence par la lettre B, signifiant BRANCH. En clair, ces instructions peuvent être lues comme:

Va à tel endroit si...
Mais si quoi ???

Eh bien plusieurs conditions sont disponibles, que l'on peut regrouper en 3 catégories:

D'abord une catégorie qui réagit à l'état d'un des bits du Status Register:

BCC Branch if carry clear (bit de retenue à 0)

BCS Branch if carry set (bit de retenue à 1)

BNE Branch if not equal (bit de zéro à 0)

BEQ Branch if equal (bit de zéro à 1)

BVC Branch if overflow clear (bit de dépassement à 0)

BVS Branch if overflow set (bit de dépassement à 1)

BPL Branch if plus (bit négatif à 0)

BMI Branch if minus (bit négatif à 1)

Une seconde catégorie, réagissant à la comparaison de nombres sans signe.

BHI Branch if higher (branche si supérieur à)

BLS Branch if lower or same (inférieur ou égal)

(on peut aussi remettre BEQ et BNE dans cette catégorie)

Une troisième catégorie, réagissant à la comparaison de nombres avec signe.

BGT Branch if greater than (si supérieur à)

BGE Branch if greater or equal (si supérieur ou égal à)

BLT Branch if lower than (si plus petit que)

BLE Branch if lower or equal (si plus petit ou égal)

(on peut encore remettre BEQ et BNE!!!)

Je suis profondément désolé pour les gens de MICRO-APPLICATION (Le Langage Machine sur ST, la Bible, le Livre du GEM etc...) ainsi que pour le journaliste qui écrit les cours d'assembleur dans STMAG, mais les branchements BHS et BLO, malgré le fait qu'ils soient acceptés par de nombreux assembleurs, N'EXISTENT PAS!!!!!

Il est donc impossible de les trouver dans un listing assemblé, l'assembleur les convertissant ou bien les rejetant.

Cet ensemble de branchement conditionnel constitue un ensemble de commande du type Bcc (branch conditionnaly)

Poursuivons notre lente progression dans le listing... La comparaison est effectuée, testons la:

```
CMP.W #"Q",D0 est-ce la lettre 'Q' ?
BEQ QUITTER branch if equal 'quitter'
```

C'est à dire, si c'est égal, sauter à l'étiquette QUITTER. Si ce n'est pas égal, le programme continue comme si de rien n'était, et tombe sur un nouveau test:

```
CMP.W #"q",D0 est-ce q minuscule ?
BEQ QUITTER branch if equal quitter
```

Nous comparons ensuite à 'V' majuscule et en cas d'égalité, nous sautons à AFFICHAGE. Viens ensuite le test avec 'v' minuscule. Là, c'est l'inverse: Si ce n'est pas égal, retour au début puisque toutes les possibilités ont été vues. Par contre, si c'est 'v' qui a été appuyé, le programme continuera sans remonter à DEBUT, et tombera de lui même sur AFFICHAGE.

L'affichage se fait classiquement avec Gemdos 9. Cet affichage terminé, il faut remonter au début. Ici, pas besoin de test car il faut absolument remonter. Nous utilisons donc un ordre de branche- ment sans condition (inconditionnel) qui se lit BRANCH ALWAYS (branchement toujours)et qui s'écrit BRA.

En cas de choix 'Q' ou 'q', il y a saut à QUITTER et donc à la fonction Gemdos 0 qui termine le programme.

N'hésitez pas à modifier ce programme, à essayer d'autres tests, à jouer avec le VT52, avant de passer au suivant.

("Quelques heures passent..." In ('Le manoir de Mortevielle')

acte 2 scène III)

Prenons maintenant le listing numéro 3. Nous étudierons le numéro 2 en dernier à cause de sa longueur un peu supérieure.

Le but de ce listing est de réaliser un affichage un peu comparable à celui des horaires dans les gares ou les aéroports: chaque lettre n'est pas affichée d'un coup mais 'cherchée' dans l'alphabet.

D'abord effacement de l'écran en affichant Escape et 'E' avec Gemdos 9: rien que du classique pour vous maintenant!

Ensuite cela se complique. Nous plaçons l'adresse de TXT_FINAL dans A6. Regardons ce qu'il y a à cette étiquette 'TXT_FINAL': nous y trouvons la phrase à afficher.

Observons maintenant TRES attentivement ce qui se trouve à l'adresse TXT. Nous y voyons 27,"Y",42 . En regardant notre feuille du VT52 nous voyons que cela correspond à une fonction plaçant le curseur à un endroit précis de l'écran. Nous constatons aussi 2 choses:

1) La commande est incomplète

2) Une phrase affichée par exemple avec gemdos 9, doit se terminer par 0, ce qui ici n'est pas le cas ! En effet, la phrase est incomplète si on se contente de lire cette ligne. Jetons un coup d'oeil sur la ligne suivante. Nous y trouvons 42, qui est peut être la suite de la commande (nous avons donc escape+Y+42+42), et une ligne encore plus bas nous trouvons deux zéros. Nous pouvons remarquer également que si la phrase commence à l'étiquette TXT, la seconde ligne possède également une étiquette ('COLONE') ainsi que la troisième ligne ('LETTRE').

Imaginons maintenant que nous ayons une lettre à la place du premier zéro en face de l'étiquette LETTRE. Si nous affichons cette phrase nous verrons s'afficher cette lettre sur la 10ème colonne de la 10ème ligne (révisez la commande Escape+Y sur la feuille du VT52).

Imaginons ensuite que nous ajoutions 1 au chiffre se trouvant à l'étiquette COLONNE et que nous recommencions l'affichage. Nous verrions notre lettre toujours 10ème ligne, mais maintenant 11ème colonne! C'est ce que nous allons faire, en compliquant d'avantage. Plaçons le code ASCII 255 (c'est le code maximale autorisé puisque les codes ASCII sont codés sur un byte) à la place du premier zéro de l'étiquette LETTRE. Nous faisons cela par MOVE.B #255,LETTRE. Ajoutons 1 ensuite au chiffre des colonnes avec ADD.B #1,COLONNE ensuite posons nous la question suivante: la lettre que je vais afficher (actuellement de code ASCII 255), est-ce la même que celle de la phrase finale ? Pour le savoir il faut prélever cette lettre de cette phrase. Comme nous avons placé l'adresse de cette phrase dans A6, nous prélevons tout en faisant avancer A6 pour pointer sur la seconde lettre. MOVE.B (A6)+,D6

Et si la lettre que nous venons de prélever était le code ASCII 0? Cela voudrais donc dire que nous sommes à la fin de la phrase et donc qu'il faut s'en aller!!! Nous comparons donc D6 qui contient le code ASCII de la lettre, avec 0.

```
CMP.B #0,D6
BEQ FIN si c'est égal, bye bye!
```

Ouf! Ce n'est pas la dernière lettre; nous pouvons donc afficher notre phrase. Cela se fait avec Gemdos 9, en lui passant l'adresse du début de la phrase dans la pile. Cette adresse c'est TXT et le Gemdos affichera jusqu' à ce qu'il rencontre 0. Il affichera donc 27,"Y",42,43,255,0. Ceci étant fait, comparons la lettre que nous venons d'afficher, et qui se trouve en face de l'étiquette LETTRE avec celle qui se trouve dans D6 et qui a été prélevée dans la phrase modèle.

Si c'est la même, nous remontons jusqu' à l'étiquette PROCHAINE, nous changeons de colonne, nous prélevons la lettre suivante dans la phrase modèle et nous recommençons. Mais si ce n'est pas la même lettre?

Et bien nous diminuons de 1 le code ASCII de 'LETTRE' (SUB.B #1,LETTRE) et nous ré-affichons notre phrase qui est maintenant 27,"Y",42,43,254,0

C'est compris ?

La aussi c'est une bonne étude qui vous permettra de vous en sortir.

N'abandonner pas ce listing en disant "oh ça va j'ai à peu près compris" il faut PARFAITEMENT COMPRENDRE. N'hésitez pas à vous servir de MONST pour aller voir à l'adresse de LETTRE ce qui s'y passe. Pour avoir les adresses des étiquettes, taper L quand vous êtes sous MONST. Il est tout à fait possible de demander à ce que la fenêtre mémoire (la 3) pointe sur une partie vous montrant LETTRE et COLONE, puis de revenir sur la fenêtre 2 pour faire avancer pas à pas le programme. Ceci vous permettra de voir le contenu de la mémoire se modifier tout en regardant les instructions s'exécuter.

Il reste un petit point à éclaircir, concernant le mot EVEN qui est situé dans la section data. Nous avons déj à compris (du moins j'espère) que l'assembleur ne faisait que traduire en chiffres des instructions, afin que ces ordres soient compris par la machine. Nous avons vu également que le 68000 n'aimait pas les adresses impaires (du moins nous ne l'avons pas encore vu, et ce n'est pas plus mal...). Lorsque l'assembleur traduit en chiffre les mnémoniques, il n'y a pas de souci à se faire, celles-ci sont toujours traduites en un nombre pair d'octets.

Malheureusement ce n'est pas forcément le cas avec les datas. En l'occurrence ici, le label CLS commence à une adresse paire (car avant lui il n'y a que des mnémoniques) mais à l'adresse CLS on ne trouve que 3 octets. Nous en déduisons que le label TXT va se trouver à une adresse impaire. Pour éviter cela, l'assembleur met à notre disposition une instruction qui permet d'imposer une adresse paire pour le label suivant, EVEN signifiant pair en Anglais.

NOTE: TOUT COMME SECTION DATA, DC.B, DC.W OU DC.L, EVEN N'EST PAS UNE INSTRUCTION DU 68000. C'EST UN ORDRE QUI SERA COMPRIS PAR L'ASSEMBLEUR.

Généralement ces ordres sont compris par beaucoup d'assembleurs mais il existe parfois des variantes. Ainsi certains assembleurs demandent à avoir .DATA ou bien DATA et non pas SECTION DATA. De même pour certains assembleurs, les labels (étiquettes) doivent être impérativement suivis de 2 points. Il faut chercher dans la doc de son assembleur et faire avec, c'est la seule solution! Notez cependant que ceci ne change en rien les mnémoniques!

Passons maintenant au dernier listing de ce cours, le numéro 2.

Ce listing affiche une image Degas dont le nom est inscrit en section data, à l'étiquette NOM_FICHIER. Il est bien évident que ce nom ne doit pas contenir de c cédille mais plut"t une barre oblique inversée, que mon imprimante a refusée d'imprimer!

Seules 2 ou 3 petites choses vous sont inconnues. Tout d'abord l'instruction TST.W (juste après l'ouverture du fichier image) Cette instruction se lit Test et donc ici on lit:

Test word D0.

Cela revient tout simplement à faire *CMP.W #0,D0.*

Seconde chose qui vous est encore inconnue, la SECTION BSS.

Nous avons vu dans les précédents que les variables initialisées étaient mises dans une SECTION DATA. Et bien les variables non initialisées sont mises dans une section nommée SECTION BSS. Cette section possède une particularité intéressante: les données y figurant ne prennent pas de place sur disque !

Ainsi si vous avez un programme de 3 kilooctets mais que dans ce programme vous désirez réserver 30 kilo pour pouvoir par la suite y charger différentes choses, si vous réservez en faisant *TRUC DC.B 30000* votre programme, une fois sur disquette fera 33000 octets. Par contre si vous réservez par TRUC DS.B 30000, votre programme n'occupera que 3 Ko sur le disque.

Ces directives placées en section BSS sont assez différentes de celles placés en section data.

TRUC DC.W 16 réserve de la place pour 1 word qui est initialisé avec la valeur 16.TRUC DS.W 16 réserve de la place pour 16 words.

Il faut bien faire attention à cela, car c'est une faute d'étourderie peu fréquente mais ça arrive! Si on note en section BSS:

```
TRUC DS.W 0
MACHIN DS.W 3
```

Lorsque l'on cherchera le label TRUC et que l'on écrira des données dedans, ces données ne pourront pas aller DANS truc puisque cette étiquette ne correspond à rien (0 word de réservé) et donc nous écrirons dans MACHIN, en écrasant par exemple ce que nous y avions placé auparavant.

Bon, normalement vous devez en savoir assez long pour utiliser le Gemdos, le Bios et le Xbios (je vous rappelle que le Bios s'appelle par le Trap #13, exactement de la même manière que le Gemdos ou le Xbios).

Vous devez donc être capable de réaliser les programmes suivants:

Demande du nom d'une image. On tape le nom au clavier, puis le programme lit l'image sur la disquette et l'affiche. Prévient et redemande un autre nom si l'image n'est pas trouvée. Si on tape X, c'est la fin et on quitte le programme.

Lecture du premier secteur de la première piste de la disquette. Si le premier octet de ce secteur est égale à $61 (c'est le code de l'instruction BRA), faire cling cling cling en affichant le code ASCII 7 (clochette), afficher "disquette infectée", attendre un appui sur une touche et bye bye. Si disquette non infectée, afficher "je remercie le Féroce Lapin pour ses excellents cours d'assembleur, super bien faits à que d'abord c'est lui le meilleur" et quitter.

Vous pouvez aussi tenter la vaccination, en effaçant carrément le premier octet (mettre à 0 par exemple).

Autre exemple assez intéressant à programmer. Vous avez vu dans le listing 3 comment prélever des données situées les unes après les autres dans une chalne: D6 contient bien d'abord F puis E puis R etc... Imaginez que vous ayez 3 chalnes: la première contient des chiffres correspondant à la colonne d'affichage, la seconde des chiffres correspondant à la ligne et la troisième des chiffres correspondant à la couleurs, ces 3 données au format VT52. (regardez Escape+'Y' et Escape+'b' ou Escape+'c'). On met un registre d'adresse pour chacune de ces listes, on lit un chiffre de chaque, on place ce chiffre dans une phrase:

```
(27,"Y",X1,X2,27,"b",X3,"*",0)
```

X1 étant le chiffre prélevé dans la liste 1
X2 étant le chiffre prélevé dans la liste 2
X3 étant le chiffre prélevé dans la liste 3

On affiche donc à différentes positions une étoile, de couleur différente suivant les affichages.

Conseil: Essayez de faire le maximum de petits programmes, afin de bien comprendre l'utilisation du VT52, du Gemdos, du Bios et du Xbios. Cela vous permettra également de vous habituer à commenter vos programmes, à les ordonner, à chasser l'erreur sournoise.

Scrutez attentivement vos programmes à l'aide de MONST. Pour le moment les erreurs seront encore très faciles à trouver, il est donc impératif de très très bien vous entralner!!!

Si un de vos programmes ne tourne pas, prenez votre temps et réfléchissez. C'est souvent une erreur ENORME qui est juste devant vous: notez sur papier les valeurs des registres, faites avancer pas à pas le programme sous MONST, repensez bien au principe de la pile avec ses avantages mais aussi ses inconvénients. Utilisez le principe des subroutines en y passant des paramètres afin de très bien maltriser ce principe.

Vous recevrez la seconde série de cours dans un mois environ. Cela vous laisse le temps de bosser. Surtout approfondissez, et résistez à la tentation de désassembler des programmes pour essayez d'y comprendre quelque chose, ou à la tentation de prendre de gros sources en croyant y trouver des choses fantastiques. Ce n'est pas du tout la bonne solution, au contraire!!!

Si vraiment vous voulez faire tout de suite un gros trucs, alors faite un traitement de texte. Avec le VT52, le Gemdos et le Bios, c'est tout à fait possible. Bien sûr, il n'y aura pas la souris et il faudra taper le nom du fichier au lieu de cliquer dans le sélecteur, mais imaginez la tête de votre voisin qui frime

avec son scrolling en comprenant 1 instruction sur 50 quand vous lui annoncerez "Le scrolling c'est pour les petits... moi je fais un traitement de texte!! "

De tout coeur, bon courage Le Féroce Lapin (from 44E)

Sommaire provisoire de la série 2

Reprogrammer les Traps,

Désassemblage et commentaire d'un programme dont nous ne sommes

pas les auteurs,

la mémoire écran

les animations (scrolling verticaux, horizontaux, sprites, ...),

la musique (avec et sans digits,

les sound trackers...),

création de routines n'utilisant pas le système d'exploitation,

le GEM et les ressources etc....