GUIDE DE PROGRAMMATION EN ASSEMBLEUR DES TI89 ET TI92 PLUS

I. Introduction

I.1. Présentation de l'ASM

I.1.a. Avantages/Inconvénients

Le langage assembleur est un langage destiné à des personnes déjà expérimentées dans la programmation et qui connaissent déjà relativement bien leur machine. C'est un langage difficile d'accès, qui risque de rebuter le débutant. Mais, s'il est vrai que ce langage est particulièrement complexe, il permet de générer des programmes d'une rapidité sans précédents, et d'une taille extraordinairement petite, avantages qui justifient à eux seuls l'apprentissage d'un tel langage. De plus, l'assembleur vous permet d'accéder a toutes les ressources de votre machine, à l'inverse du Basic qui rend certaines fonctions inaccessibles. Toutefois, ce tutorial s'adresse à des personnes n'ayant absolument aucune notion en assembleur. Il ne se veut en aucun cas exhaustif, mais devrait vous permettre de comprendre la plupart des autres tutoriaux.. Vous pourrez vous reporter, en fin d'ouvrage, à une liste de tutoriaux et de documents qu'il sera utile de lire, une fois ce document lu.

I.1.a. Mise en garde : Données

Malgré les avantages présentés ci-dessus, il est vivement déconseillé de commencer à apprendre l'assembleur avant d'avoir de bonnes bases en Basic. En effet, outre sa complexité, l'assembleur ne possède aucune gestion d'erreur : si votre programme est bogué, il bloquera totalement votre machine, et vous serez obligés d'effectuer un reset (qui entraînera la perte de vos données, tout au moins de celles sauvées dans la RAM). Il est donc très vivement conseillé de faire de fréquentes sauvegardes des données contenues dans votre machine lorsque vous développez en assembleur.

I.1. Notions fréquemment utilisées

I.1.a. Notions d'ordre général

RAM : Random Access Memory; ou mémoire vive. La RAM est une mémoire qui perd son contenu dès qu'elle cesse d'être alimentée. Sur les Ti 89 ou 92 Plus, il existe une petite pile au lithium de 3V qui a pour but de prendre le relais sur les piles principales pour assurer la conservation des données contenues dans la RAM lors d'une défaillance des piles : ainsi, la RAM des Ti est toujours alimentée, et conserve donc toujours son contenu (à moins que vous n'enleviez simultanément les 5 piles !). Du point de vue de l'utilisation, la RAM est une mémoire dont on peut modifier le contenu aussi souvent qu'on le désire. La Ti 92 Plus et la Ti 89 sont équipées de 256 Ko de RAM, qui peuvent servir aussi bien au stockage des fichiers qu'aux besoins du système d'exploitation (le stack, par exemple, est stocké dans la RAM).

ROM : Read Only Memory, ou mémoire morte. Ce type de mémoire, comme son nom l'indique, ne peut pas être modifié. Cette mémoire sert par conséquent à stocker des informations telles que le système d'exploitation (polices de caractères, contenu des menus, etc.) qui ne changeront jamais. De plus, ce type de mémoire n'a pas besoin d'être alimentée pour garder son contenu. Sur les Ti 89 et 92 Plus, la ROM contient uniquement le système d'exploitation. C'est pour cela que l'on assimile parfois la ROM au système d'exploitation : Il n'est pas rare sur un forum de voir "recherche ROM 2.03", ce qui est un abus de langage fréquemment utilisé. Ici, la personne désirait en fait le système d'exploitation v 2.03, contenu dans la ROM d'une Ti.

Flash ROM : Une Flash ROM de 2 Mo équipe les Ti 89 et 92 Plus. En fait, cette mémoire porte assez mal son nom : en effet, contrairement à ce que pourrait laisser croire le "ROM" du terme "Flash ROM", cette mémoire est réinscriptible, mais elle se comporte différemment d'une RAM classique : la Flash ROM n'a pas besoin d'être alimentée pour conserver son contenu, mais il est beaucoup plus lent de lire et d'écrire sur ce type de mémoire

que sur une RAM conventionnelle. La Flash ROM permet donc entre autres la mise à jour du système d'exploitation. Toutefois, bien que ces mises à jours apportent quelques améliorations (les ROM 2.0x permettent en effet de garder le contenu de la mémoire archivée après un plantage), ces mises à jours sont ennuyeuses pour les programmeurs en assembleur, puisqu'elles modifient certains paramètres (notamment les ROM CALLS).

Macro : En assembleur 68000, comme dans la plupart des autres langages de programmation, une macro est un petit morceau de code auquel vous donnez un nom, et que vous pourrez exécuter par la suite en donnant juste son nom. Ainsi, si vous créez une macro "pause", qui a pour rôle d'attendre la pression d'une touche, dès que vous écrirez "pause" dans le programme, celui-ci attendra la pression d'une touche. Il est particulièrement utile d'utiliser une macro pour les actions que vous répéterez souvent dans votre programme, afin de rendre le code source plus lisible. Attention toutefois : l'utilisation de Macros en assembleur n'optimise en rien votre programme, que ce soit du point de vue de la taille ou de la vitesse d'exécution; en effet, le compilateur recopie en interne toute la macro à chaque fois que vous l'invoquez. Si vous y faites souvent appel, préférez alors à la macro une subroutine (voir paragraphe correspondant).

Incrémentation et **Décrémentation** : Incrémenter une variable signifie augmenter cette variable de 1, décrémenter signifie diminuer de 1. Par exemple, si la variable D contient la valeur 3 et que vous la décrémentez, alors D contiendra la valeur 2.

Compilation : Il existe deux types de langages de programmation : les langages interprétés (Basic, Maple, ainsi que tous les langages de script en général), et les langages compilés (C/C++, Pascal, Forth, Assembleur). La différence entre les deux est la suivante : un programme interprété ne pourra pas être exécuté sans l'interpréteur (impossible de faire tourner un programme BASIC sur un ordinateur ne possédant pas le langage BASIC en ROM), alors qu'un langage compilé pourra être exécuté sur n'importe quelle machine. Ainsi, à titre d'exemple, la ROM des Ti 89 et 92 Plus a été programmée en C. Cependant, il n'y a pas de programme spécifique permettant de faire tourner un programme écrit en C intégré dans la ROM de la TI : le C est un programme compilé. En réalité, un programme écrit en langage interprété est stocké sous la forme d'un fichier texte, lu par l'interpréteur, qui va exécuter les commandes contenues dans ce fichier texte. A l'inverse, un fichier compilé est certes stocké d'un côté sous la forme d'un fichier texte (que l'on appelle dans ce cas "source"), mais également sous la forme d'un véritable programme autonome (du code machine, en fait), que l'on nomme "fichier binaire" ou "exécutable". On se retrouve donc avec deux fichiers dans le cas des programmes compilés : le "code source", et le fichier "exécutable". Le rôle du compilateur est de créer l'exécutable à partir de la source : sans compilateur, il est possible de lancer l'exécutable, mais on ne peut rien faire avec la source. De plus, l'exécutable ne peut pas être modifié, il ne peut qu'être exécuté. Pour pouvoir modifier un programme compilé (un programme assembleur, par exemple), il faut donc posséder la source, la modifier, et la recompiler. Lorsque vous programmez en assembleur, vous pouvez ainsi distribuer vos programmes tout en gardant un secret total sur les rouages internes de vos programmes. Il suffit pour cela de ne distribuer que l'exécutable, et de garder la source. Une telle attitude est cependant déconseillée : en effet, distribuer le code source permet de faire avancer la communauté de développeurs, en évitant à chacun de refaire ce qui à déjà été fait par d'autres.

I.1.a. Notions spécifiques à l'ASM Ti

AMS : AMS signifie Advanced Mathematics Software. C'est le nom du système d'exploitation des Ti 89 et 92 Plus. Pour connaître la version d'AMS que votre machine utilise, depuis l'écran Home, tapez [F1], puis [A]. Attention : comme mentionné précédemment, l'exécution d'un programme assembleur bogué entraîne un plantage de la Ti, qui nécessitera de procéder à un reset de la machine. cela implique une perte de tous les fichiers que la machine contient. Toutefois, pour les version d'AMS supérieures ou égales à la version 2.01, vous pourrez conserver, après un tel plantage, vos données archivées. Il est donc recommandé de mettre à jour votre machine si vous comptez développer en assembleur.

HW1 et **HW2** : Il existe deux versions différentes de Ti 89 et de Ti 92 Plus: les versions matérielles initiales (HW 1, pour HardWare 1), et les versions matérielles 2 (HW 2). Les différences qui existent entre ces deux machines sont relativement faibles, mais il est impossible de lancer des programmes en assembleur sur les HW2 (en fait c'est un peu plus complexes, mais ne vous fatiguez pas avec ça!). Heureusement, un programmeur a créé HW2Patch□ce programme modifie votre ROM, et permet aux HW2 d'exécuter les programmes en assembleur aussi bien que les HW1.
Attention : Pour des raisons complexes, n'envoyez surtout pas votre AMS à une autre machine si vous vous êtes servi d'HW2Patch. En effet, les programmeurs de chez Ti ont implémenté dans leurs calculatrices un système rejetant toute ROM non produite par Ti (Les ROMs officielles sont ainsi "signées"). Une ROM modifiée par HW2Patch serait alors automatiquement rejetée.

Truc : Pour savoir si la ROM de votre machine est bien signée, effectuez la manipulation suivante depuis l'écran home : Pour une Ti89, [F5], [Diamant], [CLEAR], [alpha], puis [3]. Pour une Ti 92 Plus, [F5], [Diamant], [(], puis [s]. Vous êtes alors dans le menu de test de votre Ti. Pressez la touche 1 : Si, après avoir affiché "Checking BASE CODE" pendant quelques secondes, vous revenez au menu, alors, votre ROM est correctement signée. Si un message d'erreur apparaît, la signature de votre ROM est défectueuse. NB : Vous pouvez effectuer les autres tests si vous voulez, mais faites attention au RAM test, qui effectue un reset de votre calculatrice)

I. Généralités sur le bas niveau

I.1. Base décimale, binaire, et hexadécimale

La base de numération binaire est un système qui permet d'écrire tout nombre sous la forme d'une succession de 0 et de 1 ; chaque 0 ou 1 de cette notation s'appelle bit. Ainsi, 86 se note 1010110 en binaire. Toutefois, pour ne pas confondre nombres en binaire et nombres en décimal, on adopte une notation :un `%` devant le nombre signifie binaire, rien signifie décimal. De même, un `$` signifie hexadécimal. On a donc %1010110 = 86 = $56. (Si vous n'êtes pas très à l'aise avec les changements de base, lisez le paragraphe intitulé "Systèmes de numération" du manuel de votre calculatrice). Dans un programme en assembleur, toutes les données sont stockées, en interne, en binaire, mais il est possible d'utiliser des valeurs décimales et hexadécimales lorsque l'on écrit son programme (elles seront converties par le compilateur).

Un point important de la programmation en assembleur est la taille des données manipulées : la commande utilisée pour additionner 12 et 43 sera différente de celle utilisée pour additionner 355266 et 655. En effet, le microprocesseur distingue trois types de données : les bytes (à ne pas confondre avec les bits), formées de 8 bits , les words, de 16 bits, et les longwords, de 32 bits (NB. Octet est la traduction française de byte). Pour savoir la taille de données à utiliser, on regarde le nombre de bits utilisés par notre nombre en notation binaire, et on choisi la classe supérieure ou égale. Ainsi, pour les nombres de 0 jusqu'à 255, on utilise un byte, pour les nombres de 256 jusqu'à 65535 un word, et pour les nombres de 65536 jusqu'à 4294967296 un longword (le processeur ne peut pas gérer 'à sec' de nombres plus grands). Enfin, on caractérise chaque structure (byte, word, et longword) par un ‘raccourci’, que l’on nomme mnémonique⊡ le mnémonique correspondant à byte est ‘.b’, celui correspondant à word est ‘.x’, et celui correspondant à longword est ‘.l’. Ce qui donne, en résumé⊡

Nombre à manipuler	Taille maximale en bits	Nom et mnémonique à utiliser
0 < n < 255	8 bits	Byte (.b)
256 < n < 65535	16 bits	Word (.w)
65535 < n < 4294967296	32 bits	Longword (.l)

I.1. Structure du microprocesseur

Les Ti 89 et 92 Plus sont architecturées autour de processeurs 68000 à 32 bits, fabriqués par Motorola, à une fréquence moyenne de 12 Mhz, la fréquence variant un peu selon le modèle de calculatrice (On abrège souvent le nom 68000 par 68k). Le processeur des Ti 89 et 92 Plus contient une petite quantité de mémoire cache, découpée en 16 zones de 32 bits (4 octets) chacune. Ces zones sont nommées registres, et peuvent être librement utilisées par le programmeur. On distingue deux types de registres : les registres d'adresse, ou *Adress register*, au nombre de 8, nommés a0 à A7, et les registres de données, ou *Data register*, au nombre de 8 également, et nommés d0 à d7. Du point de vue de leur structure microscopique, ces deux types de registres sont identiques. Toutefois, l'habitude veut que l'on utilise l'un ou l'autre type de registre dans des cas bien distincts ; les différences entre ces deux types de registres feront l'objet d'un paragraphe ultérieur.

Ces registres jouent en quelque sorte un rôle de variables locales, puisque leur contenu est détruit à la fin du programme. Attention toutefois au registre A7, nommé *Stack pointer*, qui joue un rôle assez particulier, que l'on détaillera plus tard. Enfin, il existe deux derniers registres, au rôle fondamentalement différent des 16 précédents : ces registres sont le *Program Counter* (PC) et le *Status Register* (SR). Ces deux registres sont plus complexes et feront eux aussi l'objet d'un paragraphe ultérieur.

Enfin, en plus des registres, il est possible d'utiliser des variables en assembleur, comme dans n'importe quel autre langage. Chaque variable peut être soit un octet (byte), soit 2 octets (word), soit 4 octets (longword). Ainsi, selon sa taille, on déclare une variable par `variable dc.b 0`, ou `variable dc.w 1200`. La valeur de droite correspond à la valeur initiale de la variable. Les chaînes de caractères sont déclarées par `chaîne dc.b "blabla bla bla",0`.

I.1. Adressage

L'adressage est à la base de la programmation en assembleur. Définissons donc tout d'abord ce qu'est une Adresse mémoire.

Adresse mémoire : La mémoire de la Ti peut être vue comme une succession de millions d'octets, numérotés 1,2,3,...,536623,... jusqu'à la fin de la mémoire. Pour désigner un octet dans la mémoire, on donne son adresse, c'est à dire son rang. Toutefois, puisque les nombres dont il s'agit sont très grands, on a l'habitude de donner les adresses en base hexadécimale pour réduire l'écriture.

L'adressage est un des concepts clés de la programmation en assembleur. Pour simplifier, on peut considérer que l'adressage correspond à la manière de donner des arguments aux instructions du 68000. Afin de faciliter la compréhension des différentes méthodes d'adressage, nous allons voir tout d'abord une première instruction, très simple, mais très utile, qui est l'instruction `move`. Le rôle de cette commande est de déplacer (copier, en fait) une certaine portion de la mémoire dans une autre. Par exemple, `move d0,a0` copiera le contenu du registre d0 dans le registre a0. La puissance du langage assembleur réside alors dans la variété des possibilités d'utilisation d'une même commande : ce sont les différentes méthodes d'adressage.

I.1.a. Adressage direct avec registres

Cet adressage manipule directement les registres. Il correspond a l'exemple donné précédemment, `move D0,A0`. On profitera de cet exemple pour remarquer que les paramètres sont passés dans un ordre bien précis : la source d'abord, la destination ensuite. Et cela est valable pour la plupart des instructions. Ainsi, `move D0,A0` copie D0 dans A0.

I.1.a. Adressage par adresse effective

Cet adressage est tout aussi simple que le précédent : il consiste à donner explicitement une adresse mémoire comme l'un des paramètres : `move 325,D0`. Attention ! Cela ne copie en AUCUN cas le nombre 325 dans D0; cela copie ce qui est contenu dans la mémoire a l'adresse 325 dans le registre D0.

I.1.a. Adressage immédiat

Cet adressage est sans doute le plus simple de tous : il consiste en effet à prendre comme premier argument une valeur fixée, par exemple 3, ou 25, ou 14, etc. Toutefois, si l'on se contentait de mettre `move 12,D0`, il serait impossible de faire la distinction entre l'adressage immédiat et l'adressage par adresse effective. On a alors recours a la distinction suivante : un `#` devant le nombre indique qu'il s'agit d'un adressage immédiat. Par exemple, `move #3,D0` copiera le nombre 3 dans le registre D0, et `move 3,D0` copiera le 3ème octet de la mémoire dans D0. Il est très important de bien saisir la différence qu'il y a entre ces deux méthodes d'adressage. En effet, une confusion entre ces deux méthodes peut être une source de bugs très compliqués à résoudre.

I.1.a. Adressage indirect

Cette méthode d'adressage est la plus complexe, mais d'un autre coté la plus puissante. Elle ne marche qu'avec les registres d'adresse, et elle est caractérisée par des parenthèses : par exemple, `move D0,(A0)`. Notre exemple a une signification totalement différente de `move D0,A0`. Supposons, pour bien saisir la différence, que, avant d'exécuter cette commande, D0 contienne la valeur 3, et A0 la valeur 5. Si l'on fait `move D0,A0`, alors D0 contient toujours la valeur 3, mais A0 a changé de valeur : il vaut désormais 3 aussi. Maintenant, si l'on fait `move D0,(A0)` (avec les parenthèses, donc adressage indirect), le processeur copie le contenu de D0 non pas dans A0, mais à l'adresse pointée par A0. Cela signifie que, puisque A0 vaut 5, le processeur va copier la valeur de D0, ici 3, dans l'adresse mémoire 5. Ainsi, une fois cette instruction effectuée, D0 et A0 conservent leur valeur initiale, mais le 5ème octet de la mémoire contient le nombre 3.

On vient de voir là les 4 méthodes d'adressage les plus importantes. Il existe de plus quelques variantes :

I.1.a. Adressage indirect avec post-incrémentation

Cet adressage est marqué par un `+` après la parenthèse de l'adressage indirect. Par exemple, `move (A0)+,D0`. Le plus signifie que, une fois l'instruction (sans le plus) effectuée, on augmente le registre entouré des parenthèses de 1. Cette opération revient donc à copier ce qui est contenu dans l'adresse mémoire pointée par A0 dans D0, PUIS à ajouter 1 a A0. Cette fonctionnalité est très utile dans le cas de boucles.

I.1.a. Adressage indirect avec pré-décrémentation

Cet adressage est marqué par un **–** avant la parenthèse de l'adressage indirect. Par exemple, `move -(A0),D0`. Le moins signifie que, avant que l'instruction soit effectuée, on diminue le registre de 1. Cette opération revient donc à soustraire 1 à A0, PUIS à copier ce qui est contenu dans l'adresse mémoire pointée par A0 dans D0. Cette fonctionnalité est, elle aussi, très utile dans le cas de boucles.

I.1.a. Adressage indirect avec décalage

Cet adressage ne concerne que les registres d'adresse. Il se note par exemple `move x(A1,y),D0`. La valeur x doit être un nombre, par exemple 4. y doit être un registre, par exemple A2. Alors, dans ce cas, l'opération `move 4(A1,A2),D0` copiera l'octet enregistré à l'adresse 4+A1+A2 dans D0. En fait, cet adressage est assez proche des deux précédents. Il correspond en effet à une extension de l'adressage indirect. Ainsi, lorsque l'on écrit `move x(A1,y),D0`, tout ce passe comme si on ajoutait à A1 tout d'abord x, puis y, et si l'on faisait ensuite `move (A1),D0`. Toutefois, l'intérêt de notre nouvelle méthode est de ne pas modifier A1. Par exemple, si A1 vaut 5, et A2 vaut 3, et que l'on fait `move 4(A1,A2),D0`, cela revient à ajouter 3 puis 4 dans A1 (donc A1 vaut maintenant non plus 5 mais 5+3+4=12), puis à faire `move (A1),D0`. Tout se déroule identiquement, sauf que si on avait fait `move 4(A1,A2),D0`, A1 vaudrait toujours 5, alors que notre autre méthode modifie A1 qui, à la fin, vaut 12.

I.1. Instructions du 68000

Les processeur 68000 compte près de 80 instructions distinctes. Mais le programmeur moyen (pas seulement le débutant) en utilise régulièrement le quart. Et il y a plusieurs raisons à cela : tout d'abord, certaines instructions sont très proches, avec des nuances très subtiles. Si vous mettez une forme générale de l'instruction, le compilateur choisira automatiquement la forme la mieux adaptée, et donc cela restreint considérablement le nombre d'instructions à apprendre. De plus, il existe un certain nombre d'instructions "exotiques", dont on a très rarement besoin.

De plus, il est très important, maintenant que nous allons voir de nombreuses instructions, d'introduire la notion de mnémonique. Le mnémonique est une invention assez géniale. Celui-ci permet en effet de limiter la portée d'une instruction, dans la mesure où il limite la taille des données manipulées par l'instruction. Ainsi, la plupart des instructions acceptent un suffixe (un mnémonique), parmi : `.l` pour longword (4 octets), `.w` pour word (2 octets), et `.b` pour byte (1 octet), comme nous l'avons déjà vu. Ainsi, `move D0,A0` ne s'écrit pas exactement comme ça dans un programme. On doit mettre `move.l D0,A0` si l'on désire copier la totalité des 32 bits de D0 vers A0, mais l'on peut mettre aussi `move.b D0,A0`, si seul les 8 derniers bits nous intéressent. (NB : Oui, il s'agit bien des 8 DERNIERS bits, et non des 8 premiers, cela est dû à la façon dont le 68k stocke les nombres. Mais lorsque vous débutez, ne vous embêtez pas avec ça, utilisez simplement toujours des longwords). Ainsi, on peut utiliser juste la quantité de mémoire dont on a besoin, et ainsi optimiser à la fois vitesse d'exécution et mémoire utilisée. (NB : Pour le débutant, il n'est pas la peine de se compliquer la vie à optimiser vos programmes de la sorte. Au début, utilisez simplement des longword. Une fois
que vous aurez acquis de bonnes bases, alors là oui, il devient intéressant, voire nécessaire, de bien maîtriser les mnémoniques.) Par ailleurs, on remarquera que la plupart des instructions du 68k obéissent à l'ordre source-cible : comme pour "move", le premier argument est la source, le second la cible. Retenez bien cela, car c'est vrai pour la quasi-totalité des instructions. Enfin, dans ce guide, un exemple de syntaxe possible est donné avec chaque instruction. Remarquez que ce n'est pas la seule syntaxe possible : d'autres peuvent être essayées. Pour une référence complète, consultez un guide destiné à un public plus averti. Remarquez aussi que seuls certains modes d'adressage sont possibles pour chaque instruction⊡ par exemple, l'instruction `eor` ne peut accepter comme premier argument un registre d'adresse. Si votre compilateur vous renvoie une erreur, lisez donc attentivement ce qu'il vous indique, et reportez vous en cas de doute à une référence plus complète des instructions du 68000.

I.1.a. Manipulation de données

EXG (EXchanGe) : Cette instruction échange le contenu de deux registres, que ce soit des registres d'adresse ou de donnée.

Syntaxe : `exg Dn,An`

LEA (Load Effective Address) : Cette instruction met une adresse effective dans l'un des registres d'adresse. Elle est principalement utilisée pour manipuler des variables. Par exemple, `lea num(PC),a0` chargera l'adresse de la variable num dans a0. (NB : Le (PC) final correspond, comme vous pouvez le deviner, à une structure

d'adressage indirect plus complexe. Mais pour l'instant, contentez-vous de le mettre, de toute façon il le faut toujours quand on utilise un nom de variable avec Lea).

Syntaxe : `lea <ae>,An` (<ae> = Adresse effective)

MOVE : Comme nous l'avons vu, l'instruction move copie un certain endroit de la mémoire dans un autre.

Syntaxe :
```
move.l Dn,An
move.b <ae>,Dn
move.w #<data>,Dn
```

SWAP : Cette instruction échange les 16 premiers bits et les 16 derniers bits d'un registre de donnée.

Syntaxe : `swap Dn`

I.1.a. Arithmétique

ADD : Cette instruction ajoute le premier argument au second. Par exemple, si D0 vaut 4 et A2 vaut 5, alors après `add.l A2,D0`, A2 vaut toujours 5, mais D0 vaut 9.

Syntaxe :
```
add.l Dn,<ae>
add.w (An)+,Dn
```

SUB (SUBstract) : Cette instruction soustrait le premier argument au second. Par exemple, si D0 vaut 7 et A1 vaut 2, alors après `sub.l A1,D0`, A1 vaut toujours 2, mais D0 vaut 5.

Syntaxe :
```
sub.l Dn,<ae>
sub.w (An),Dn
```

CLR (CLeaR) : Cette instruction efface le contenu du paramètre fourni.

Syntaxe :
```
clr.l D0
clr.b A0
```

MULU (MULtiply Unsigned) : Cette instruction multiplie le second argument par le premier. Par exemple, si D0 vaut 4 et A2 vaut 5, alors après `mulu.l A2,D0`, A2 vaut toujours 5, mais D0 vaut 20.

Attention : L'instruction MULU ne fonctionne qu'avec des words.

Remarque : L'instruction MULS est similaire, mais elle tient compte du signe des nombres (Le signe d'un nombres en binaire n'ayant pas été abordé, nous ne discuterons pas de cette instruction).

Syntaxe :
```
mulu.w #3,Dn
mulu.w (A1),Dn
```

DIVU (DIVide Unsigned) : Cette instruction divise (division euclidienne sans reste) le second argument par le premier. Par exemple, si D0 vaut 4 et A2 vaut 3, alors après `divu.l A2,D0`, A2 vaut toujours 5, mais D0 vaut 1 (dans 4 il va UNE fois trois et reste 1).

Attention : Comme pour MULU, l'instruction DIVU ne fonctionne qu'avec des words.

Remarque : L'instruction DIVS est similaire, mais elle tient compte du signe des nombres.

Syntaxe :
```
divu.w #3,Dn
divu.w (A1),Dn
```

I.1.a. Logique

AND : Cette instruction effectue un ET logique entre le premier et le second argument, le résultat étant stocké dans le second argument.

Syntaxe :
```
and.w <ae>,Dn
and.l Dn,<ae>
```

EOR (Exclusive OR) : Cette instruction effectue un OU Exclusif entre le premier et le second argument, le résultat étant stocké dans le second argument.

Syntaxe : `eor.w Dn,<ae>`

OR : Cette instruction effectue un OU logique entre le premier et le second argument, le résultat étant stocké dans le second argument.

Syntaxe :
```
or.b <ae>,Dn
or.l Dn,<ae>
```

NOT : Cette instruction effectue un NON logique du paramètre fourni.
Syntaxe : `not.l <ae>`

I.1.a. Décalages et rotations

LSL (Logical Shift Left) : Décale n fois vers la gauche les bits du second paramètre, n étant la valeur du premier paramètre. Par exemple, si D0 contient %0110, et que l'on fait `lsl.b #1,D0`, alors D0 vaut %1100.
Remarque : Cette instruction permet d'optimiser de façon importante vos programmes : en effet, si vous avez besoin de multipilier un nombre par 2, 4, 8, ou toute puissance de 2, plutot que de faire `mulu.l #8,DO`, faites `lsl.l #3,D0`, qui revient au meme, mais va beaucoup plus vite.
Syntaxe : `lsl.l #3,D0`
`lsl.b Dn,Dn`

LSR (Logical Shift Right) : Décale n fois vers la droite les bits du second paramètre, n étant la valeur du premier paramètre. Par exemple, si D0 contient %0110, et que l'on fait `lsr.b #1,D0`, alors D0 vaut %011.
Remarque : Cette instruction permet elle aussi d'optimiser de façon importante vos programmes : en effet, si vous avez besoin de diviser un nombre par 2, 4, 8, ou toute puissance de 2, plutôt que de faire `divu.l #8,DO`, faites `lsr.l #3,D0`, qui revient au même, mais va beaucoup plus vite.
Syntaxe : `lsr.l #3,Dn`
`lsr.b Dn,Dn`

ROL (ROtate Left) : Décale n fois vers la gauche les bits du second paramètre, n étant la valeur du premier paramètre. Les n bits rajoutés à droites sont ceux qui ont été enlevés à gauche lors de la rotation. Par exemple, si D0 contient %11100110, et que l'on fait `rol.b #1,D0`, alors D0 vaut %11001101.
Syntaxe : `rol.l #3,D0`
`rol.w Dn,Dn`
`rol.w <ae> (1 rotation)`

ROR (ROtate Right) : Décale n fois vers la droite les bits du second paramètre, n étant la valeur du premier paramètre. Les n bits rajoutés à gauche sont ceux qui ont été enlevés à droite lors de la rotation. Par exemple, si D0 contient %01100111, et que l'on fait `ror.b #1,D0`, alors D0 vaut %10110011.
Syntaxe : `ror.l #3,D0`
`ror.l Dn,Dn`
`ror.w <ae> (1 rotation)`

I.1.a. Manipulation de bits

BCHG (Bit CHanGe) : Cette instruction change la valeur du n-ième bit du second paramètre, n étant la valeur du premier paramètre. Par exemple, si D0 vaut %1001, alors après `bchg.b #2,D0`, D0 vaut %1011.
Syntaxe : `bchg.b Dn,<ae>`
`bchg.l #n,<ae>`

BCLR (Bit CLeaR) : Cette instruction efface (met à 0) le n-ième bit du second paramètre, n étant la valeur du premier paramètre. Par exemple, si D0 vaut %1011, alors après `bclr.b #2,D0`, D0 vaut %1001.
Syntaxe : `bclr.b Dn,<ae>`
`bclr.l #n,<ae>`

BSET (Bit SET) : Cette instruction met à 1 le n-ième bit du second paramètre, n étant la valeur du premier paramètre. Par exemple, si D0 vaut %1001, alors après `bset.b #2,D0`, D0 vaut %1011.
Syntaxe : `bset.b Dn,<ae>`
`bset.l #n,<ae>`

I.1.a. Comparaisons, sauts, et étiquettes

Avant d'aller plus loin, il est nécessaire de préciser que, comme la plupart des autres langages, l'assembleur possède une structure de saut conditionnel. Cependant, celle-ci se résume à comparer deux valeurs, puis à sauter vers une étiquette ou une vers une autre en fonction du résultat de la comparaison. Une étiquette en assembleur se note alors `nom_de_l'etiquette:` (n'oubliez pas le : final). On a alors :

BRA (BRanch Always) : Cette instruction est la plus simple : c'est le saut inconditionnel. En effet, suite à cette instruction, le programme saute vers l'étiquette passée en second paramètre. Par exemple, après `bra boucle`, le programme effectuera ce qui est après `boucle:` dans le programme. On remarquera que, après `bra`, l'on ne met pas les `:` à la fin du nom de l'étiquette, contrairement à lorsque l'on déclare cette dernière.

CMP (CoMPare) : Cette instruction compare le premier paramètre et le second paramètre, et stocke le résultat dans une zone de la mémoire du microprocesseur. Toute seule elle ne sert à rien, mais elle est un préliminaire indispensable à tout saut conditionnel.

Syntaxe : `cmp.b Dn,<ae>`
`cmp.l #n,<ae>`

BEQ (Branch if EQual) : Cette instruction saute vers le second paramètre si le résultat de la dernière comparaison était une égalité. Par exemple, si l'on fait `cmp.l #2,#3` puis `beq boucle`, alors rien ne se passe car 2 n'est pas égal à 3. Mais si l'on fait `cmp.l #2,#2` puis `beq boucle`, alors le programme saute vers `boucle:`.

Syntaxe : `beq etiquette`

BGE (Branch if Greater or Equal) : Cette instruction saute vers le second paramètre si le résultat de la dernière comparaison était une inégalité large telle que paramètre 2 soit plus grand que paramètre 1. Par exemple, si l'on fait `cmp.l #3,#2` puis `bge boucle`, alors rien ne se passe car 2 n'est pas supérieur ou égal à 3. Mais si l'on fait `cmp.l #2,#2` puis `bge boucle`, alors le programme saute vers `boucle:`.

Syntaxe : `bge etiquette`

BLE (Branch if Lower or Equal) : Cette instruction saute vers le second paramètre si le résultat de la dernière comparaison était une inégalité large telle que paramètre 2 soit plus petit que paramètre 1. Par exemple, si l'on fait `cmp.l #2,#3` puis `ble boucle`, alors rien ne se passe car 3 n'est pas inférieur ou égal à 2. Mais si l'on fait `cmp.l #2,#2` puis `ble boucle`, alors le programme saute vers `boucle:`.

Syntaxe : `ble etiquette`

BGT (Branch if Greater Than) : Cette instruction saute vers le second paramètre si le résultat de la dernière comparaison était une inégalité stricte telle que paramètre 1 soit plus grand que paramètre2. Par exemple, si l'on fait `cmp.l #2,#2` puis `bgt boucle`, alors rien ne se passe car 2 n'est pas strictement supérieur à 2. Mais si l'on fait `cmp.l #2,#3` puis `bgt boucle`, alors le programme saute vers `boucle:`.

Syntaxe : `bgt etiquette`

BLT (Branch if Lower Than) : Cette instruction saute vers le second paramètre si le résultat de la dernière comparaison était une inégalité stricte telle que paramètre 2 soit plus petit que paramètre 1. Par exemple, si l'on fait `cmp.l #2,#3` puis `blt boucle`, alors rien ne se passe car 3 n'est pas strictement inférieur à 2. Mais si l'on fait `cmp.l #3,#2` puis `blt boucle`, alors le programme saute vers `boucle:`.

Syntaxe : `blt etiquette`

Remarque : Il existe une variante de toutes ces instructions : `Dbxx Dn,etiquette`. A chaque fois qu'une de ces instructions est rencontrée, on branche vers étiquette en fonction du dernier test, si Dn est non nul, puis Dn est décrémenté. Ces instructions sont très utiles pour construire des boucles, un peu à la manière d'une boucle for.

I.1.b. Contrôle du programme

JSR (Jump to SubRoutine) : Cette instruction saute vers une étiquette, mais en enregistrant au préalable l'endroit d'ou elle est partie dans une zone du microprocesseur.

Syntaxe : `JSR etiquette`

RTS (ReTurn from Subroutine) : Cette instruction va de pair avec JSR. En effet, elle revient juste après le dernier JSR exécuté. Cela permet, après un saut, de reprendre le programme exactement où on l'avait laissé. Si aucun JSR n'a été rencontré précédemment, RTS quitte le programme (Il faut d'ailleurs *toujours* finir ses programmes par un RTS).

Syntaxe : `RTS`

TRAP : Cette instruction permet de faire appel à des fonctions du système d'exploitation. Par exemple `Trap #4` éteint la calculatrice. Vous pourrez vous reporter aux documentations indiquées en annexe pour plus de détails sur les différents traps possibles.

Syntaxe : `Trap #nombre`

NOP (No OPeration) : Cette instruction ne fait rien ! Elle peut être utile pour mettre des toutes petites pauses dans vos programmes, imperceptibles "à l'œil nu", mais qui permettront de laisser le temps au matériel de réagir (au clavier, par exemple).

Syntaxe : `NOP`

Ce document s'adresse à un public débutant dans la programmation en assembleur. Certaines instructions ont donc été volontairement omises, et d'autres n'ont été décrites que partiellement. Si vous désirez une documentation plus exhaustive, consultez des ouvrages destinés à un public expérimenté. Vous pourrez par exemple vous reporter au guide de Jimmy Mardell (voir référence en annexe).

I.2. Le Stack

Nous avions déjà parlé succinctement du rôle particulier du registre d'adresse A7, encore connu sous le nom de stack pointer. Le stack est une zone de la mémoire qu'utilise le système d'exploitation AMS pour stocker toutes les informations temporaires dont il a besoin (par exemple, paramètres des programmes en Basic). Il est alors intéressant d'utiliser cet espace mémoire assez important (environ 15 Ko) dans vos programmes en assembleur. Tout d'abord, il faut savoir que le stack possède une structure de pile. Une pile mémoire, fonctionne comme une pile de feuilles, d'où son nom. Pour accéder à une feuille d'une pile, il faut obligatoirement désempiler toutes celles qui sont dessus. Le stack marche de la même façon. Par exemple, vous allez mettre une donnée A dans le stack, puis, une donnée B, puis une donnée C. Si vous désirez à nouveau lire A, il faut d'abord enlever B, et C, du stack. Mais, la seule chose que l'on sait, c'est que, a tout instant, la valeur contenue dans A7 est l'adresse mémoire du haut de la pile. Ainsi, si vous mettez quelque chose sur le stack, il vous faudra mettre à jour A7. Rien de plus facile. Supposons que l'on veuille mettre la valeur 39 dans le stack. On penserait donc à faire `move.b #39,(A7)`. Mais A7 désigne le haut de la pile. L'emplacement pointé par A7 est donc déjà occupé. Or, A7 compte la pile négativement : plus elle est haute, plus il est négatif. Ainsi, si vous voulez mettre un octet dans le stack, il faut décaler A7 de –1, pour que A7 désigne un emplacement libre. Il faut donc faire : `sub.l #1,A7` puis `move.b #39,(A7)`. Les deux opérations peuvent alors être résumées en `move.b #39,-(A7)`. Vous pouvez continuer à empiler des valeurs sur le stack : `move.b #12,-(A7)`, `move.b #161,-(A7)`. Pour les désempiler dans D0, il faut maintenant faire `move.b (A7)+,D0` trois fois de suite. Cet exemple de programme est bien sûr stupide est inutile, car au final les informations sont perdues, chaque donnée dépilée venant écraser l'autre dans D0. Mais il illustre le fonctionnement du stack.

Remarque : Quand vous empilez des données sur le stack, il faut penser à décrémenter au préalable A7, et ce d'autant d'unités que vous pensez stocker d'octet. Ainsi, si vous prévoyez stocker un longword, il faudra préalablement soustraire 4 à A7. Cela peut paraître assez fastidieux. Heureusement, l'adressage indirect avec pré-décrémentation automatise la tache : il soustrait automatiquement à A7 la quantité adéquate en fonction du mnémonique utilisé. Ainsi, lorsque vous faite `move.l #123,-(A7)`, l'opération soustrait préalablement 4 à A7 (puisque un longword correspond à 32bits, soit 4 octets), tandis que si vous faites `move.w #123,-(A7),` elle ne soustraira préalablement que 2 à A7.

Attention : Pensez à toujours laisser le registre A7 dans l'état dans lequel vous l'avez trouvé, sinon le système d'exploitation ne trouverait plus les données qu'il avait lui-même empilé avant que votre programme ne soit lancé.

Enfin, une des fonctionalités intéressantes du stack est la possibilité de sauvegarder les registres : en effet, si le stack est un espace mémoire pouvant être librement modifié tant que vous n'abîmez pas les données empilées par le système d'exploitation, il est plus risqué de modifier les registres. En effet, modifier un registre et ne pas le restaurer à la fin du programme peut, même si c'est globalement peu fréquent, faire planter votre machine. Une astuce consiste donc à empiler tous les registres sur le stack au début du programme, et à les dépiler tout à la fin. cela s'effectue avec les commandes : `movem.l d0-d7/a0-a6,-(A7)`, puis à la fin du programme `movem.l (A7)+,d0-d7/a0-a6`. L'instruction movem (MOVE Multiple), n'avait pas été présentée car c'est en effet presque exclusivement avec le stack qu'on l'utilise. Elle est équivalente a une succession de move. Ici, d0-d7/a0-a6 signifie "de d0 à d7" puis de "a0 à a6". Vous pouvez bien sur modifier cette formule un peu générale pour l'adapter à vos besoins.

I.3. Heap, Handles, et fichiers

Le stack est un espace de stockage très utile. Toutefois, il est fort probable que vous ayez besoin à un moment ou à un autre, d'un espace de stockage assez étendu (plusieurs Ko) dont vous pourriez disposer librement (plus de système de pile par exemple). Une manière de faire cela serait, par exemple, d'écrire dans une zone donnée de la RAM. Mais la mémoire des Ti 89 et 92 Plus est utilisée par le système d'exploitation AMS.
Celui-ci s'en sert en effet pour enregistrer tous vos fichiers, vos réglages, l'historique, etc ... Ainsi, écrire n'importe où dans la RAM va, d'une façon presque inévitable, entraîner un plantage de la machine puisque cela risque très probablement de corrompre les données utilisées par AMS. Il vous faut donc "cohabiter" avec le système d'exploitation, en lui demandant de vous allouer un espace mémoire. Un tel espace est appelé Heap. Pour créer un heap, il suffit de mettre dans le stack un longword indiquant la taille (en octets) du Heap que vous désirez obtenir. On effectue ensuite un appel à la ROM par la commande `jsr tios::HeapAlloc`. En pratique, cela donne :

```
move.l taille(PC),-(A7)
jsr tios::HeapAlloc
add.l #4,A7
```

On obtient ensuite dans le registre d0 l'adresse de l'espace ainsi alloué par AMS. Toutefois, l'adresse n'est pas retournée directement : le registre d0 contiendra en réalité ce que l'on appelle le Handle de votre Heap. Les handles sont une sorte d'index, qu'AMS utilise pour simplifier l'écriture. Ainsi, Heap et Handle vont de paire : chaque Heap attribué par la calculatrice possède son handle. Toutefois, si vous désirez utiliser cet espace mémoire ainsi alloué, il vous faudra connaître l'adresse du Heap, et non son Handle. Heureusement, la macro `Deref d0,a0` (contenue dans le fichier d'en-tête "tios.h") vous permet d'obtenir dans a0 l'addresse du heap correspondant au handle contenu dans d0. Vous voilà fin prêt à utiliser votre Heap. Enfin, à la fin de votre programme, n'oubliez pas de détruire votre Heap, sinon vous allez utiliser pour rien de la RAM : en effet, le système d'exploitation vous a alloué cet espace, et n'y touchera donc plus, jusqu'à ce que vous lui demandiez explicitement de le détruire. Si vous ne le faites pas, cet espace restera inutilisé, et ne sera pas détruit. Pour se faire, après avoir mis le handle du Heap à détruire sur le stack (le handle doit être un word), faites `jsr tios::HeapFree`; cela donne en pratique :

```
move.w handle(PC),-(A7)
jsr tios::HeapFree
add.l #2,A7
```

Enfin, il faut savoir que la structure que l'on vient de mettre en œuvre est très proche de la structure qui permet à la Ti d'enregistrer vos fichiers : en effet, chaque fichier de votre machine est un Heap. Les handles de tous les fichiers sont stockés dans un endroit bien précis de la mémoire (la table d'allocation de fichiers), qui permet à la machine de s'y retrouver. Vous pourrez trouver de nombreux documents quant à traitant de la manipulation des fichiers à partir de programmes en assembleur sur l'Internet.

II. De la théorie à la pratique

II.1. Compilation du premier programme

II.1.a. Choix du compilateur

Nous allons enfin pouvoir créer nos premiers programmes. Pour ce faire, il faut bien sûr choisir un compilateur, mais heureusement votre choix se limitera à deux options : un compilateur *on-calc* (comprenez : compilateur tournant sur votre Ti), ou un compilateur sur PC. Chaque option a ses avantages; en voici la liste :

- Compilateur sur PC :
  - Rapidité de compilation : Bien évidemment, le processeur de votre Ti ne peut en aucun cas rivaliser avec celui d'un PC, aussi vieux soit-il. Vos programmes seront donc compilés beaucoup plus vite sur PC. Toutefois, le temps de compilation sur Ti reste raisonnable, tant que votre code source ne dépasse pas les 5 Ko.
  - Fiabilité : a68k, LE compilateur sur PC, est de bien meilleure facture que ses homologues sur Ti : il supporte bien plus d'options qu'eux, et surtout cible très précisément les erreurs de votre code source. Lorsqu'il n'arrive pas à compiler un morceau de votre programme, il vous fourni un rapport détaillé qui vous permettra rapidement de corriger votre erreur. Il en va tout

autrement sur Ti, où trouver l'erreur qui empêche votre code d'être compilé relève parfois du miracle.

- Compilateur sur Ti :
  - Mobilité : Est-il vraiment nécessaire de détailler cet énorme avantage ?
  - Pas de transferts : En effet, chaque fois que vous compilez un programme sur PC, vous devez par la suite le transférer sur votre Ti. Cette contrainte disparaît dès que l'on compile le programme sur la calculatrice.

Enfin, il faut savoir que lorsque vous programmerez en assembleur, votre Ti plantera très, très souvent. C'est tout à fait normal, et sans aucun danger pour la calculatrice. Mais, comme nous l'avons déjà vu, chaque plantage implique une perte des données en RAM. Si vous développez en assembleur, à fortiori *on-calc*, mettez votre ROM à jour vers une version 2 (la 2.03 est assez bien), et archivez toutes vos données⊡ainsi, celles-ci seront conservées après un reset.

II.1.b. Premier exemple de programme

Contrairement à la tradition, nous ne commencerons pas par un "Hello, world", car ce programme ne constitue pas un bon exemple dans notre situation. Voilà donc notre premier programme :

```
;Pour mettre un commentaire, mettez un point virgule en début de ligne
_main:
;La notation _main: est une notation est standard. Elle indique le début du programme
movem.l d0-d7/a0-a6,-(A7)
;On vient d'empiler tous les registres sur le stack. On peut donc les modifier librement : on n'aura qu'à les
; désempiler à la fin du programme pour les remettre dans ; l'étant dans lequel on les a trouvés.

trap #4
;Ici, on fait appel à une fonctionnalité du système d'exploitation AMS : l'extinction de la Ti. Je tiens à souligner
; que nous n'avons pas véritablement écrit ici un programme qui éteint la Ti. Notre programme se contente de
; faire appel à la fonction d'AMS idoine. Mais il serait absolument idiot de vouloir faire autrement (par exemple
; de recopier le code du système d'exploitation).

movem.l (A7)+,d0-d7/a0-a6
;Comme prévu, on restaure les registres
rts
;Comme nous l'avons vu, étant donné qu'il n'y a pas eu de jsr auparavant, ce rts signifie "fin du programme". On
; peut l'interpréter comme un "retour à AMS".
end
;Cette directive indique la fin du code source. Elle n'est utilisée que par le compilateur. Remarque : sous AS92
; ne la mettez pas.
```

Voila votre premier programme : un programme capable d'éteindre votre Ti ! Vous voyez donc ici un des intérêt de l'assembleur : vous pouvez accéder à l'intégralité des fonctionnalités de votre machine.

II.1.c. Les différents modes de compilation

Il existe deux options, des "directives de compilation", que vous pouvez mettre au début de votre code source. Ces directives sont les directives `_nostub` et `_strout`. Il faut savoir qu'il ne faut jamais utiliser `_strout` tout seul : on utilise soit aucune directive, soit `_nostub`, soit `_nostub` ET `_strout` (un programme avec `_strout` seulement serait compilé mais planterait votre machine).Voyons l'effet de ces directives sur notre premier programme⊡

- Si l'on met la directive `_nostub`, notre programme compilé pert quelque octets (cool!), mais fonctionne (presque) exactement comme avant.
- Si l'on rajoute la directive `_strout`, le compilateur ne sort plus un exécutable, mais un fichier texte, contenant une succession de chiffres et de lettres (A,B, C, D, E, et F, plus particulièrement). Je suis sûr que vous aurez immédiatement reconnu de l'hexadécimal. Maintenant, recopiez cette chaîne de caractères (en l'occurrence "4E444E750000") dans l'écran home, et faites la précéder de "Exec" (on a donc 'Exec "4E444E750000"'). Validez : miracle, votre programme s'exécute! En fait, la directive "_strout" (STRing OUTput) permet de sortir une chaîne de caractères exécutable par Exec.

La directive `_nostub` est plus subtile. En effet, en théorie, les Ti89 et 92 Plus sont supposées pouvoir lancer, d'origine, n'importe quel programme en assembleur. Dans la pratique, la plupart des programmes nécessitent ce que l'on appelle un shell, ou encore un kernel. Ces programmes se chargent en RAM une fois pour toutes (jusqu'au prochain reset), et modifient un peu le comportement du système d'exploitation (mais pas la ROM, rassurez vous). Parmi les modifications qu'apporte un shell digne de ce nom, on trouve de nombreuses extensions facilitant la vie au programmeur en assembleur. Par exemple, un shell permet l'utilisation de librairies. Pour pouvoir accéder à toutes ces fonctionnalités, il suffit de ne rien mettre dans votre code source. Si cependant vous n'avez pas besoin de ces fonctionnalités (comme c'est par exemple le cas dans notre programme), vous pouvez inclure la directive `_nostub`. Celle-ci diminuera légèrement la taille de votre programme, mais rendra surtout possible l'exécution de votre programme sans avoir recours à un shell. Un plus non négligeable, car souvent les utilisateurs non avertis ne savent pas qu'il faut un shell pour lancer la majorité des programmes en assembleur. Enfin, il n'est pas recommandé de débuter en utilisant la directive `_nostub`, le shell rendant quand même pas mal de services.

II.2. Les librairies

Il existe sur Ti de nombreuses librairies. Toutefois, les trois librairies suivantes sont les plus répandues : GraphLib, UserLib, et FileLib. Comme leur nom l'indique, GraphLib offre de nombreuses fonctionnalités graphiques, FileLib permet d'effectuer différentes manipulations sur les fichiers, et Userlib permet d'accomplir différentes taches souvent requises (gestion du clavier par exemple). Pour utiliser une des fonctions d'une librairie, il faut passer par deux étapes : tout d'abord, il faut inclure le fichier d'en-tête de la librairie. Il suffit d'ajouter dans votre code source `include "nom_librairie.h"`. Ensuite, pour exécuter une fonction de la librairie, il suffit de faire `jsr nomlib::fonction`. Par exemple, si vous voulez exécuter la fonction idle_loop de userlib, faites `jsr userlib::idle_loop`. Le passage d'éventuels paramètres s'effectue au moyen des registres, et c'est encore avec eux que l'on peut récupérer les éventuelles informations retournées par la librairie. Voici documentées la plupart des fonctions des trois librairies principales :

II.3. Les ROM CALLs

Les ROM CALLs (Appels à la ROM) fonctionnent d'une manière très similaire à celle des librairies. En fait, un ROM CALL est une fonction intégrée dans la ROM de votre machine, à laquelle vous pouvez faire très simplement appel dans votre programme. Il existe toutefois trois différences majeures entre les librairies et les ROM CALLs : tout d'abord, les ROM CALLs (comme la ROM en entier) ont été programmés en C. A fonctionnalité égale, ils sont donc beaucoup plus lents que les librairies, qui sont quant à elles programmées en assembleur (et en plus elles sont bien optimisées). De plus, la plupart des paramètres fournis lors d'un appel à un ROM CALL ne sont pas passés par les registres, mais par le stack : vous devez y empiler tous vos paramètres. Enfin, les ROM CALLs peuvent, contrairement aux librairies, être utilisés en mode `_nostub`. Voici une liste de quelques ROM CALL utiles. Il existe en tout près de 1000 ROM CALLs disponibles, mais une liste exhaustive des ROM CALLs existant sortirait de l'objectif de ce document. Vous pourrez vous reporter à l'excellente documentation de TiGCC, qui documente la quasi-totalité des ROM CALLs (voir annexe). En voici tout de même quelques-uns commentés et expliqués. Pour faire appel à l'un d'eux, après avoir mis sur le stack les paramètres nécessaires, dans l'ordre donné, deux possibilités s'offrent a vous⊡

- Si vous programmez en mode Kernel, il vous faudra avoir inclus le fichier "tios.h". Ensuite, faites "`jsr tios::NomDuRomCall`".
- Si vous programmez en mode _nostub, vous devrez inclure le fichier "OS.h", et faire appel a la macro ROM_CALL de la façon suivante⊡"`ROM_CALL NomDuRomCall`".

tios::ST_ShowHelp(Texte)⊡ Cette fonction est très utile : elle permet en effet de placer un texte quelconque dans la barre de statut, en bas de l'écran.
Entrée : 1-Longword : Adresse au texte
Sortie : Rien
Pour cette première fonction, nous allons donner un exemple d'utilisation. Ainsi, ici, il fallait comprendre :

```
lea texte(pc),-(A7)
;Ici, le paramètre donné est l'adresse du texte à mettre. Comme on faisait précédemment avec A0, on met
; l'adresse du texte sur le stack, et on oublie pas la pré-décrémentation, afin de ne pas écraser ce qui était déjà
; sur le stack. On a utilisé lea car il s'agissait d'une chaine de caractères. On aurait utilisé move si c'était un
; nombre, en utilisant le mnémonique adéquat (conformément à la syntaxe précisée).
```

```
jsr tios::ST_ShowHelp
;On fait appel au ROM CALL
add.l #4,A7
;On dépile les données. En fait, on ne les dépile pas vraiment, car elles sont toujours sur le stack, mais tout se
; passe comme si on les avait dépilé. En effet, on a changé l'adresse du haut de la pile, et nos données se
; trouvent donc maintenant au dessus du plus haut objet de la pile. Elles seront donc écrasées lorsque l'on
; rempilera quelque chose sur le stack.
rts
texte dc.b "Ce texte est génial !",0
```

Comme on vient de le voir, il ne faut jamais oublier de dépiler les paramètres que l'on a mit (ou au moins de modifier l'adresse du haut de la pile).

tios::FontSetSys(Police)⊡Cette fonction très pratique permet de régler la police utilisée par toutes les fonctions d'affichage. La police 0 est la police mini, celle utilisée en bas de l'écran, la police 1 est celle utilisée dans le menu APPS, et la police 2, utilisée par AMS seulement sur 92 Plus (mais existant aussi sur 89) est celle utilisée dans la barre d'entrée de la fenêtre home.
Entrée : 1-Word : Taille de la police
Sortie : Rien
Voici un dernier exemple pour être sûr que vous avez bien compris :

```
move.w #2,-(A7)
jsr tios::FontSetSys
add.l #2,A7
```

tios::DrawStr(Couleur,Texte,y,x)⊡Cette fonction permet d'afficher un texte à l'écran, dans la police choisie par FontSetSys.
Entrée : 1-Word : Couleur
2-Longword : Texte à afficher
3-Word : Ordonnée
4-Word : Abscisse
Remarque : Pour utiliser des ROM_CALLs dans vos programmes, n'oubliez pas d'inclure le fichier tios.h ou OS.h. Enfin, faites attention au fait que souvent les ROM CALL altèrent le contenu des différents registres. Mieux vaut donc les copier sur le stack avant de faire appel à un ROM CALL.

III. Entrées - Sorties

Nous y voila enfin ! Vous allez pouvoir vous y mettre sérieusement. En effet, pour l'instant, les seuls moyens que nous avions de communiquer avec l'utilisateur étaient les librairies, et les ROM CALLs. Ceux-ci, bien que souvent très utiles, ne nous laissent pas pourtant tirer partie de toute la puissance de notre machine. Nous allons ainsi apprendre désormais comment accéder directement à chacune des parts du matériel de nos machines.

III.1. L'écran

III.1.a. La mémoire vidéo

Il existe de nombreuses façons d'accéder à l'écran de la calculatrice. Nous en avons déjà vu deux : par la librairie GraphLib (qui est très performante), et par les ROM CALLs. Toutefois, il est possible, d'accéder directement à l'écran, puisque l'assembleur est un langage de bas niveau. C'est cette méthode que nous allons étudier maintenant. Mais il faut auparavant savoir certaines choses sur le fonctionnement de l'écran. La première chose à savoir est que la Ti 89 possède une mémoire vidéo égale à celle de la Ti 92 Plus. Cela signifie que la Ti 89 se comporte, au niveau de l'écran, comme une Ti 92 Plus dont on aurait masqué un morceau d'écran. Par ailleurs, il faut savoir aussi que l'écran n'est pas géré directement par le processeur principal (le 68000, celui que nous programmons), mais par un co-processeur graphique (ce type de processeur est habituellement nommé GPU, pour Graphical Processing Unit). Ce processeur graphique se charge de lire en permanence (plusieurs centaines de fois par seconde) une zone de la RAM, qu'il affiche à l'écran. En effet, il existe dans la RAM de la Ti un espace réservé, la mémoire vidéo, qui correspond à ce qui est affiché l'écran. Le contenu de l'écran est codé de façon très simple : on sait que l'écran de la Ti est monochrome. Chaque pixel sera donc soit noir, soit blanc. Il

suffit donc d'associer à chaque pixel un bit de la mémoire : si le bit est à 0, le pixel correspondant sera éteint, si le bit est à 1, le pixel allumé. Ainsi, le contenu de l'écran est enregistré dans la RAM à partir de l'adresse $4C00.
A partir de là, chaque pixel de l'écran correspond à un bit, sachant que le premier bit correspond au pixel en haut à gauche de l'écran, et que la numérotation se poursuit dans le sens de lecture (de gauche à droite, puis de bas en haut). Mais attention : l'adresse $4C01 ne correspond pas au second pixel, mais aux pixels numéro 9 à 17. En effet, une adresse désigne un octet, c'est à dire huit pixels. Le second pixel est donc le deuxième bit de l'adresse $4C00. Enfin, la Ti 92 Plus comporte 240*128=30720 pixels; la mémoire vidéo s'étend donc sur 30720/8=$F00 octets. La mémoire vidéo se termine donc à $4C00+$FC00=$5B00. L'avantage de cette méthode est d'être très rapide. On peut en effet modifier l'intégralité de l'écran en une fraction de seconde. Il est par contre fastidieux de l'utiliser pour des modifications de pixels.
Vous pouvez donc maintenant modifier l'affichage comme bon vous semble : il vous suffit de modifier la mémoire entre $4C00 et $5B00. Essayez par exemple `move.l #$FFFFFF,$4C00`, vous verrez !
Remarque : Les programmes en mode Kernel incluent par défaut (même si votre programme est vide !) une procédure de sauvegarde et de restauration de l'écran. Si vous exécutez ce programme en mode Kernel, vous ne verrez rien. En mode _nostub, par contre, tout se passe normalement.

III.1.b. Les écrans virtuels

Un gros problème se pose toutefois rapidement si vous avez besoin d'enchaîner de nombreuses images à l'écran (une animation par exemple). Ce problème est très facile à comprendre : considérons l'exemple d'un jeu, mettons un jeu comme Space Invaders. A chaque mouvement du décor, le programme devra remettre à jour l'affichage. Si le décor bouge lentement, pas de problèmes. Mais si l'animation s'accélère, de nombreux phénomènes parasites vont entrer en compte, pour la bonne et simple raison que le temps pris par la modification de l'image ne sera plus négligeable devant le temps passé à afficher l'image : si l'action devient endiablée, votre programme ne passera à la fin son temps qu'à rafraîchir l'écran. On arrive alors à une situation gênante : certains graphismes sont effacés alors qu'ils viennent juste d'être affichés, car la situation du jeu a changé, alors que d'autres, à l'inverse (ceux affichés en premier par la routine) restent affichés plus longtemps que prévu. Le fait que tous les graphismes ne soient pas affichés aussi longtemps se traduit par le fait que vous ne voyez plus rien (enfin, plus qu'une bouillie de pixels) à l'écran. Il existe heureusement une façon très élégante de remédier à ce problème : cette méthode consiste à utiliser ce que l'on appelle des écrans virtuels. Grâce à cette méthode, vous obtiendrez des graphismes d'une fluidité impressionnante. Ainsi, nous avons vu précédemment que les Ti 89 et 92 Plus possédaient un GPU. Ce processeur, qui se charge d'afficher à l'écran une certaine zone de la RAM, peut être également programmé, d'une manière indirecte : il a été dit précédemment que ce processeur allait chercher les données à afficher à l'adresse $4C00. cela n'est que partiellement vrai : en effet, le GPU va effectivement chercher ces données à cette adresse en temps normal, mais il est possible de modifier l'adresse à laquelle le GPU va chercher les informations à afficher à l'écran. On peut lui demander par exemple, si on le souhaite, d'afficher à l'écran les données contenues à l'adresse $10000. On peut ainsi entretenir deux écrans différents, et choisir de basculer de l'un à l'autre : il suffit pour cela de changer, à chaque fois que l'on désire passer d'un écran à l'autre, l'adresse à laquelle le GPU va chercher les informations à afficher à l'écran. Mais attention : si la zone mémoire $4C00-$5B00 est réservée à l'affichage, il n'en est rien pour la zone $10000-$10F00, par exemple. Si vous décidez ainsi d'utiliser la zone $10000-$10F00 pour l'affichage, vous risquez d'écrire sur une zone utilisée par le système d'exploitation, ce qui, tôt ou tard, fera planter la machine. Ainsi, si vous désirez utiliser un ou plusieurs écrans virtuels, il vous faut demander au système AMS de vous allouer un espace mémoire de $F00 octets par écran virtuel. Vous aurez donc compris que l'on a besoin, tout simplement, d'un Heap de $F00 octets. Ainsi, après avoir crée un (ou plusieurs) Heap (voir paragraphe II.6 pour plus de détails), vous n'aurez plus qu'à indiquer au GPU lequel utiliser. Cela n'est pas très compliqué. En effet, pour ce faire, il suffit d'obtenir l'adresse de votre Heap (par Deref; voir II.6), de diviser cette adresse par 8, et de mettre le résultat de cette division dans $600010. Par exemple, cela peut donner

```
;Je suppose que vous avez créé un heap, dont l'adresse est stockée dans A0.
lsr.l #3,A0
;cela revient, comme nous l'avons déjà vu, à diviser par deux à la puissance 3 (c'est à dire huit), sauf que ça va
;VRAIMENT beaucoup plus vite que de faire "divu.l #8,A0" (qui reviendrait au même).
move.l A0,$600010
```

Voilà, c'est tout : le GPU ira chercher ce qu'il doit afficher dans votre handle. Enfin, n'oubliez pas de rétablir l'affichage à la fin du programme, avant de détruire votre handle : remettez $4C00/8=$980 dans $600010 (`move.l #$980,$600010`). Vous voilà prêt à réaliser de superbes animations !
Remarque : L'utilisation d'écrans virtuels n'est absolument pas nécessaire. Cette partie a été rajoutée pour vous montrer la puissance et la flexibilité du langage assembleur, ainsi qu'une application possible d'un Heap.

III.2. Le clavier

Le clavier est, comme l'écran, géré par les ROM CALLs ou les librairies. Toutefois, vous aurez peut-être remarqué que ni les librairies, ni la ROM, ne vous permettez de gérer complètement celui-ci : en effet, il est impossible de tester rapidement si telle ou telle touche est pressée ou pas. Pour ce faire, nous allons, comme précédemment, profiter du fait que l'assembleur soit un langage de bas niveau pour accéder directement à l'état des touches. Le clavier des Ti 89 et des Ti 92 Plus est un clavier dit "matriciel" (comprenez "sous la forme de tableau"). Ainsi, pour lire l'état d'une touche, vous devrez d'abord indiquer quelle ligne du tableau vous désirez lire, et la calculatrice vous donnera la position des touches éventuellement pressées sur la ligne que vous avez sélectionnée. Toutefois, un petit problème se pose : la disposition des touches sur les Ti 89 et sur les Ti 92 Plus est différente. Il faudra donc que vous adaptiez votre programme en fonction de la machine à laquelle il sera déstiné. Voici le tableau (la matrice) représentant le clavier des Ti 89 :

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Bit 0	Alpha	Diamant	Shift	2nd	Droite	Bas	Gauche	Haut
Bit 1	F5	Clear	^	/	*	-	+	Enter
Bit 2	F4	□	T	,	9	6	3	(-)
Bit 3	F3	Catalog	Z	)	8	5	2	.
Bit 4	F2	Mode	Y	(	7	4	1	0
Bit 5	F1	Home	X	=	|	EE	Sto	Apps
Bit 6								Esc

Voici le tableau représentant le clavier des Ti 92 Plus :

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Bit 0	Bas	Droite	Haut	Gauche	Main	Shift	Diamant	2nd
Bit 1	3	2	1	F8	W	S	Z	
Bit 2	6	5	4	F3	E	D	X	
Bit 3	9	8	7	F7	R	F	C	Sto
Bit 4	,	)	(	F2	T	G	V	Espace
Bit 5	Tan	Cos	Sin	F6	Y	H	B	/
Bit 6	P	Enter	Ln	F1	U	J	N	^
Bit 7	*	Apps	Clear	F5	I	K	M	=
Bit 8		Esc	Mode	+	O	L	Teta	□
Bit 9	(-)	.	0	F4	Q	A	Enter	-

Pour lire l'état d'une touche, voilà la procédure à suivre : tout d'abord, vous devez indiquer à la machine la ligne que vous désirez lire. Sur une Ti 92 Plus, supposons que vous désirez lire l'état des touches directionnelles. Celles-ci se situent sur la première ligne. Ainsi, vous devez indiquer à la machine que seule cette zone vous intéresse : pour ce faire, vous devrez écrire dans l'adresse mémoire $ 600018. Dans notre exemple, la première ligne nous intéresse. Ainsi, nous devons mettre à un tous les bits de $600018, sauf le bit 1. Les bits se comptant de la droite vers la gauche, nous devons mettre %1111111111111110 dans $600018, soit condensé en hexadécimal, `move.w #$FFFE,$600018`. Ensuite, il faut laisser le temps au processeur et au clavier de réagir. Six `nop` conviendront. Nous pouvons désormais tester l'état de n'importe quelle touche de la première ligne. Pour une Ti 92 plus, supposons que l'on veuille savoir si la touche gauche est pressée. Il nous suffira de tester le bit numéro 4 de l'adresse idoine, à savoir $60001B. Ainsi, on effectuera `btest.b #4,$60001B`. Il ne restera plus qu'a sauter vers l'étiquette souhaitée au moyen d'une instruction `beq` ou d'une instruction similaire.

Remarque : La touche ON n'apparaît pas dans le tableau. Elle est accessible à part, et à tout moment, en testant le bit 1 de l'adresse $6001A (i.e. `btest.b #1,$60001A`).

IV. Contact

Adresse : Romain Goyet
5, traverse Regny
13009 Marseille
France

Email⊡ r_goyet@yahoo.fr

N'hésiter pas à m'adresser vos questions/commentaires⊡cela permettra d'améliorer les futures versions de ce document.

V. Annexe

http://members.chello.at/gerhard.kofler/kevin/ti89prog/asmnstbf.htm
Compléments d'information sur la programmation en mode _nostub.

http://www.ticalc.org/pub/text/68k/68kguide.txt
Référence très complète des instructions du 68000 (en anglais).

http://tigcc.ticalc.org/manual.html
Référence des ROM_CALLs (en anglais).

VI. Légal

Version⊡GAT v1.0 – 21 décembre 2002