# Initiation à l'Assembleur (1)

*par André C. et tous ceux qui voudront bien y participer...*

Une nouvelle rubrique ? Pas vraiment, car il y a eu des précédents dans le Ceo-Mag : Une trentaine d'articles, écrits par Vagelis B. (n°151), Roger B. (n°35), Fabrice F. (n°122, 126, 127), Thomas G. (n°1, 5, 8, 11, 13), François L. (n°20, 22, 26, 34), Dominique P. (n°85, 86, 87, 88, 89, 99) et moi-même (n°51, 54, 71).

En fait il y a bien plus d'articles que ça qui ont été consacrés au langage machine, Mais où se situe la limite de l'initiation ? Je crois me souvenir que seuls François et Thomas ont intitulé leurs articles «Initiation à l'assembleur».

Toujours est-il que la demande pour de tels articles est plus pressante que jamais. Combien de fois n'ai-je pas entendu soupirer «Je suis nul en assembleur !». Cette «nouvelle» série est dédiée en particulier à Stéphane W., qui a adhéré au CEO entre autres pour apprendre le langage machine. Ne vous laissez pas impressionner par la longueur de cet article, ni par sa densité. Il y a forcément beaucoup de choses à dire au commencement. Mettez à profit les mois d'été pour jouer avec les exemples proposés. Vous allez voir c'est simple et satisfaisant à comprendre.

**Quels Outils faut-il ?**

Un Oric (ou un émulateur) et c'est à peu près tout. Il va sans dire que pour une initiation, nous resterons modestes et ne parlerons pas des compilateurs croisés, ni même des assembleurs sophistiqués, avec labels etc.

Comme pour la rentrée des classes, je peux encore vous recommander quelques fournitures optionnelles, qui vont alourdir un peu votre sacoche... Une calculette permettant d'effectuer des conversions décimal/hexadécimal/binaire vous rendra de grands services. Les modèles simples ont existé, mais ont disparu au profit d'usines à gaz inutilisables par toute personne normalement constituée, âgée de plus de 18 ans. A défaut, utilisez une table de conversion, par exemple celle parue dans le Ceo-Mag n°147, page 68, grâce à François S. Il existe aussi des programmes de conversion tournant sous windows.

Un Moniteur/Assembleur/Désassembleur pourra aussi vous simplifier la tâche (à moins que vous ne soyez un fervent du Basic et de ses POKEs). La liste des bons programmes sur Oric est longue. En voici quelques-uns (par ordre alphabétique) : AS des AS d'Isosoft, Assembleur Symbolique de Loriciels, Automon d'André Chénière, Editeur-Assembleur de Micrologic, Hadès de Ere Informatique, MonAsm de Vismo (retouché Roger B., Théoric n° 35 page 44 et Ceo-Mag 128 page 13), Moniteur 1.0 de Loriciels (Ceo-Mag de juin), Moniteur/Assembleur & Super Désassembleur de Jean-Paul Laurent (Théoric), sans compter les excellents programmes de Jean-Jacques Jung ou de Jim P. (articles à venir)...

Un programme simple sera plus facile à utiliser pour cette initiation, mais évidemment plus limité par la suite, si vous devenez un accro du langage machine. Un programme complexe (avec éditeur de fichiers sources, labels étendus, compilation multi-passes etc.) pourra vous causer des soucis, surtout si la notice n'est pas à la hauteur du programme. Prêtez bien attention à la notice avant de vous décider. Sans notice, il vaut mieux renoncer à utiliser un programme comme Hadès (excellent logiciel).

Les exercices auxquels nous allons nous livrer sont vraiment élémentaires et ne demandent pas un outil sophistiqué. Utilisez de préférence celui que vous connaissez déjà un peu ou à défaut un programme simple. Faites aussi attention à ce que les programmes les plus anciens utilisent uniquement la cassette pour les entrées/sorties. Il existe parfois une adaptation disquette, c'est le cas par exemple pour Hadès (adaptation Sedoric de Denis Henninot). Vous hésitez encore ? Bon, prenez Automon d'André Chénière. Il est simple, compact et... automobile (programme et notice sur simple demande).

Quoique vous puissiez faire les exercices de cette rubrique quasiment sans matériel annexe, un bouquin vous permettra de donner libre cours à votre curiosité et de vérifier certains points. Là encore, il en existe des quantités. Je vous renvoie à «La Librairie Oric». Si vous avez l'occasion de mettre la main sur «L'Oric à Nu» de Fabrice Broche, n'hésitez pas, c'est la bible de tout Oricien éclairé. Autres célébrités : «6502, Programmation en langage assembleur» de L. A. Leventhal et «Programmation du

6502» de R. Zaks. Notez que votre Manuel Atmos contient toutes les informations de base. Les annexes, à la fin du bouquin, sont notamment bien faites.

**Un peu de théorie**

Ah ! Bon, c'est justement ce que vous craigniez ? Rassurez-vous, cette rubrique ne va marcher qu'avec des exemples concrets et si possible ludiques. Le Langage Machine, vous n'y comprenez rien ? Comment est-ce possible, puisque c'est le plus simple de tous les langages ? Il suffit de dire directement au microprocesseur ce qu'il doit faire et ceci avec des commandes qui par définition sont élémentaires. Par exemple «Prends la valeur #42 et met la dans la mémoire M». Si l'adresse M se trouve dans la zone de l'écran, alors la lettre «B» sera visible à l'écran (car #42 est le code Ascii de «B»). Peut-on imaginer quelque chose de plus simple ? Avec des pièces de Lego simples, on peut construire de jolis édifices...

Ah oui, il y a ces longs listings abscons ! Ne vous laissez pas impressionner. Ces listings ne sont qu'une suite d'opérations simples et se lisent de façon linéaire. Comme dans n'importe quel programme, il y a des tests et des branchements, voire des sous programmes. Et alors ?

Donc, je résume, pas de théorie sur les registres du 6502, les modes d'adressage etc. Si besoin, de temps en temps, je rappellerai sous forme de mémento comment fonctionnent quelques instructions qui ne sont pas très intuitives (BCS, ASL, ROR etc.).

Ah j'oubliais ! Je dois vous dire quand même qu'une ligne d'instruction se présente sous la forme «9815  A0 00  LDY #00» où le premier groupe (ici 9815) est l'adresse mémoire où est situé le code, le second groupe (ici A0 00) est le code en question. Ce code peut avoir de 1 à 3 octets de long, le premier étant l'instruction et les éventuels suivants étant les infos dont l'instruction a besoin. Enfin, le troisième groupe (ici LDY #00) est la traduction de cette instruction en (presque) clair, autrement dit en langage assembleur. Ici LDY #00 signifie «Load Y avec la valeur #00» soit Y=0. Toutes les instructions sont symbolisées par 3 lettres et sont appelées «mnémoniques».

**Bon, on passe aux travaux pratiques**

Que diriez-vous d'afficher «Salut les gars !» sur la ligne service ? Au moins, c'est du concret immédiat ! Et c'est simple : Il faut stocker ce message quelque part, lancer une boucle qui en lit un à un les octets et les copie sur la ligne service, qui n'est qu'une zone mémoire comme n'importe qu'elle autre, située de BB80 à BBA7.

Allez, il faut vous y faire : Les programmeurs utilisent l'hexadécimal et le binaire plutôt que le décimal. Tout simplement parce que ces deux systèmes sont plus près de la structure et du fonctionnement de l'ordinateur. Donc, il est plus facile d'imaginer comment l'ordinateur va opérer et de lui proposer une solution compacte et rapide.

Saviez-vous que dans beaucoup de civilisations (et notamment chez les gaulois) on utilisait, non pas le système décimal, mais un système à base vingt, tout simplement parce que pour compter, on peut utiliser non seulement les 10 doigts de ses mains, mais aussi ceux de ses pieds ? Il en reste des traces dans la langue française : La vingtaine et ses dérivés (comme vingt et un, quatre-vingts, quatre-vingt-douze...). Il y a aussi l'hôpital des Quinze-vingts (créé pour 300 aveugles revenus des croisades), etc.

Dommage que l'ordinateur ne soit pas structuré par vingtaines, nous aurions pu utiliser nos doigts au lieu d'avoir besoin d'une calculette. Quoique je préfère utiliser une calculette, plutôt que d'enlever mes chaussures !

Pour qu'il soit protégé du Basic par HIMEM, nous allons implanter notre programme au dessus de 9800 (c'est de l'hexadécimal, oubliez le décimal). Notez que les assembleurs brillent par l'hétérogénéité de leurs conventions : On voit souvent des choses comme #9800 et #$00, mais aussi le contraire $9800 et $#00. Il y a un point commun : Lorsqu'on liste un programme, on trouve des «#» ou des «$» à toutes les lignes ! De même, lorsqu'on tape le programme, il faut y mettre des paquets de «#» ou de «$». Petite convention d'un paresseux : Dans mes exemples, les adresses seront sans «#» ou «$» et les données avec un simple «#» (mais toujours en hexadécimal, évidemment). Coup de chance, le programme Automon d'André Chénière que j'ai utilisé pour vérifier mes exemples, accepte cette syntaxe simplifiée.

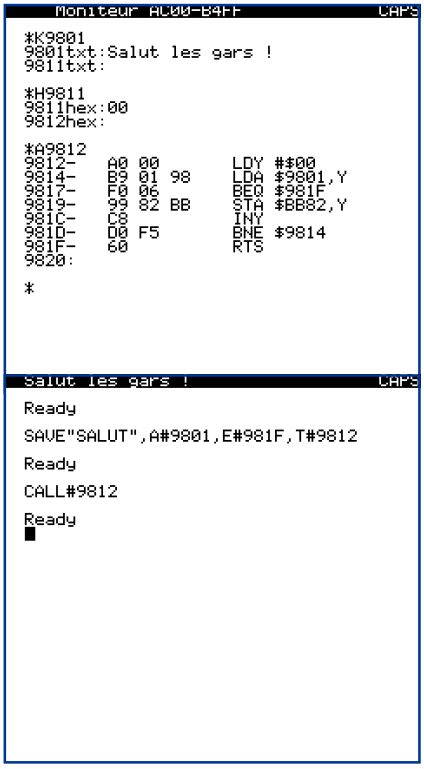
Bref, je reviens à mes moutons. Voici donc 3 variantes d'un même petit programme pour afficher une chaîne de caractères terminée par zéro.

Pour entrer les exemples ci-dessus, vous pouvez POKEr les octets en gras à partir de 9801 ou fabriquer un loader en Basic qui fera la même chose. Il en existe de nombreux exemples dans les revues et je ne reviendrai pas là-dessus. Vous pouvez aussi, **et c'est bien plus agréable,** utiliser un assembleur. Il y a

deux possibilités, soit cet assembleur travaille en mode immédiat (cas par exemple de Automon), soit il faut éditer un listing source que le programme va ensuite compiler (cas de la plupart des assembleurs). Consultez soigneusement la notice du programme que nous voulez utiliser afin de comprendre comment opérer.

Variante 1 :

```
9801 53 61 6C 75 74 20 6C 65 73 20 67 61 72 73 20 21 00  chaîne «Salut les gars !»
9812 A0 00     LDY #00    mettre la valeur zéro dans le registre Y
9814 B9 01 98  LDA 9801,Y lire l'octet présent dans la mémoire d'adresse 9801+Y
9817 F0 06     BEQ 981F   si la valeur lue est zéro, c'est fini on termine en 981F
9819 99 82 BB  STA BB82,Y sinon on copie l'octet en BB82+Y (dans la ligne service)
981C C8        INY        Y=Y+1 pour indexer l'octet suivant
981D D0 F5     BNE 9814   on reboucle en 9814 tant que Y ne repasse pas à zéro
981F 60        RTS        fin du programme on retourne au point d'appel
```




Commentaires complémentaires :

1) Remarquez le #00 placé en 9811 pour marquer la fin du message. On aurait aussi pu utiliser un compteur, puisqu'il y a 16 caractères dans ce message. Nous verrons dans un autre article comment ça marche. Mais sachez qu'un compteur prend plus de place que les 3 octets utilisés ici (le #00 ajouté à la fin et les deux octets du test «BEQ 981F». En outre, notre code est universel, il marchera quel que soit la longueur du message à afficher (enfin presque, voir plus loin). Dans l'autre solution, à chaque utilisation de la routine d'affichage, il faudra prévoir une mise à jour du compteur, en fonction du message.

2) L'instruction «BEQ» est une des plus courante. Elle signifie «Branche à l'adresse indiquée si EQual». Egal à quoi ? En fait, elle signifie «Branche si le drapeau Z est à un», ce qui se passe dans plusieurs conditions : Soit, une soustraction ou une comparaison à donné un résultat nul (Zéro), soit on a chargé une valeur nulle dans un registre, etc. Dans votre bouquin sur le langage machine 6502 ou à défaut dans le manuel de l'Atmos, pages 292 à 295, vous pouvez voir que certaines instructions seulement peuvent modifier le drapeau Z. En ce qui nous concerne aujourd'hui, c'est le cas de LDA et INY, mais pas de STA (store A, c'est à dire met le contenu du registre A à l'adresse qui suit).

3) Il semble logique de lire un octet (LDA 9801,Y) puis de regarder s'il est nul (drapeau Z mis à 1), auquel cas on sort du programme (BEQ 981F). Puisque le STA n'affecte pas le drapeau Z, nous aurions pu aussi placer le test BEQ après le STA. Mais dans ce cas le #00, marqueur de fin de message, serait aussi écrit dans l'écran. La valeur #00 correspond à l'attribut d'encre noire et cela pourrait perturber ce qui suit dans cette ligne de l'écran. Conclusion, ce n'est pas compliqué, mais rien n'est innocent, il faut faire attention car le microprocesseur exécute bêtement tout ce qu'on lui demande ! L'ordre dans lequel les instructions sont exécutées est très important, plus que lorsque vous mettez dans votre bouche le camembert puis le pain au lieu du pain puis du camembert !

4) En 9814 (LDA 9801,Y), pour lire les 16 caractères du message, la valeur de Y ira, à chaque passage dans la boucle, de la valeur 0 à la valeur 15. Mais en 981C avec l'instruction INY (INcrease Y, soit Y=Y+1), la valeur de Y passera de la valeur 1 à la valeur 16. Donc le BNE (branche si pas nul, c'est à dire dans ce cas particulier, branche si Y n'est pas nul) est un branchement forcé puisque Y sera toujours compris entre 1 et 16. **Ce truc du branchement forcé est courant.** Notre code marchera impeccablement, sauf si nous tentons d'afficher un message de longueur supérieur ou égale à #FF. En effet, après les incrémentations successives, lorsqu'on arrive à Y=#FF+1, on repasse à #00 et le branchement forcé ne marche plus. Le programme passera alors à l'instruction suivante (RTS, retour à l'appelant, c'est à dire fin de la routine) et terminera donc avec un affichage tronqué.

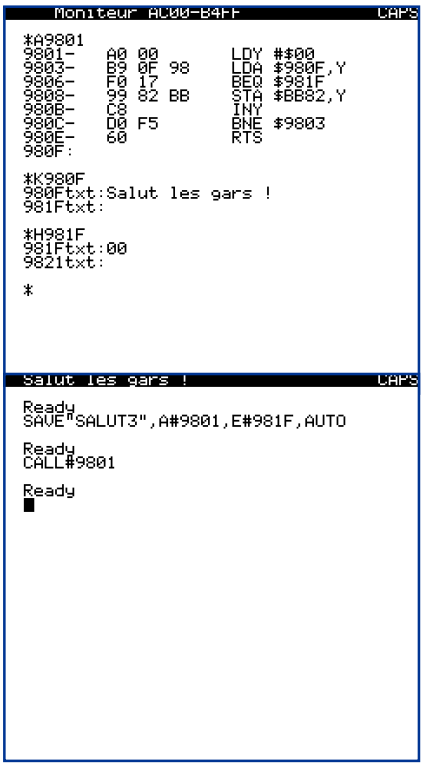
Variante 2

```
9801 4C 15 98  JMP 9815
9804 53 61 6C 75 74 20 6C 65 73 20 67 61 72 73 20 21 00 Salut les gars !
9815 A0 00     LDY #00
9817 B9 04 98  LDA 9804,Y
981A F0 06     BEQ 9822
```

```
981C 99 82 BB   STA BB82,Y
981F C8         INY
9820 D0 F5      BNE 9817
9822 60         RTS
```

Pour pouvoir utiliser un lecteur de cassette, qui ne connaît pas d'adresse d'exécution autre que celle du début du programme, nous avons ajouté un JMP 9815 au début du fichier sauvé en automatique. Si la variante 1 pouvait parfaitement être sauvée avec un SAVE«SALUT»,A#9801,E#981F,T#9812, par contre, un CSAVE«SALUT»,A#9801,E#981F,AUTO aurait conduit à un plantage, car les octets en 9801 ne correspondent pas à du code exécutable. Notez qu'il aurait été possible de faire un CSAVE«SALUT»,A#9801,E#981F et plus tard un CLOAD«» suivit d'un CALL#9812, mais cette dernière valeur n'est inscrite nulle part et risque d'être oubliée. Avec la variante 2, un CSAVE«SALUT2»,A#9801,E#9822,AUTO marche impeccablement. Le programme est seulement plus long de 3 octets, dus au JMP 9815 initial.

Toutefois, ce n'est pas une excellente pratique que de placer les données au milieu du code. Une autre solution est celle de la variante 3. Les données sont à la fin, après le code et un CSAVE«SALUT3»,A#9801,E#981F,AUTO sera alors bien adapté. De plus le programme retrouve sa longueur initiale. Vous allez me dire que pour 3 octets, il n'y a pas lieu de chipoter. Mais c'est pourtant l'obsession permanente du programmeur en langage machine : Faire le plus optimisé possible pour des questions de place et de vitesse d'exécution. Certes, ici cela ne joue pas beaucoup. Mais si vous voulez écrire un programme façon Jonathan B. ou Mickaël P., il faudra être très économe sur les ressources du système.




Variante 3

```
9801 A0 00     LDY #00
9803 B9 0F 98  LDA 980F,Y
9806 F0 06     BEQ 980E
9808 99 82 BB  STA BB82,Y
980B C8        INY
980C D0 F5     BNE 9803
980E 60        RTS
980F 53 61 6C 75 74 20 6C 65 73 20 67 61 72 73 20 21 00 Salut les gars !
```

**Il est temps de faire un petit point**

Je me souviens qu'à la question «Comment faire pour bien programmer», Fabrice Broche avait répondu un jour «Je ne sais pas, j'essaie de bien organiser mes données» (ou quelque chose comme ça). Et c'est profondément vrai !

La première tâche que nous avons donc eu à faire, était de stocker «Salut les gars !» quelque part en mémoire. Deux options étaient possibles : Avec le code et on sauvegarde le tout ensemble (c'est celle que nous avons choisie) ou ailleurs en mémoire (par exemple le programme en 9801 et le message en BB00) et il faudra sauver un fichier programme et un fichier de données. Ça ne change rien à la programmation proprement dite, a part quelques adresses qu'il faut modifier.

Dans l'option retenue (avec le code), il restait à définir où placer les données : Devant le code (variante 1), au milieu du code (variante 2) ou après le code (variante 3). Nous avons vu que devant le code, ça pose un problème avec CSAVE, qui sait seulement lancer l'exécution en début de fichier. Mais, si comme avec Sedoric, vous êtes capable de dire au Dos «Sauve le programme en Auto, de telle à telle adresse, avec exécution à telle autre adresse», alors vous pouvez faire comme vous voulez.

La deuxième tâche que nous ayons eu à faire, était de renseigner le programme sur la longueur du message à afficher. Nous avons vu que pour éviter d'avoir à mettre en place un compteur, nous avons marqué la fin du message à afficher avec un zéro. Cette solution est abondamment utilisée dans la Rom de l'Oric.

Il est possible bien entendu d'utiliser d'autres marqueurs. Sedoric, par exemple, ajoute #80 au dernier caractère du message, ce qui force le bit 7 à un (les 8 bis d'un octet se lisent de droite à gauche et sont donc notés b7 b6 b5 b4 b3 b2 b1 b0). Lors du chargement de ce dernier caractère, le drapeau N

(Négatif) passe à un et cela peut être détecté avec un BMI (Branche si Minus, c'est à dire négatif) au lieu du BEQ. Pour essayer, dans la variante 1, vous pouvez remplacer le #00 par #EA (NOP ou No OPeration, instruction bouche-trou qui ne fait rien), remplacer le #21 final par #A1 (correspondant à «!» + #80) et remplacer les lignes BEQ 981F et STA BB82,Y par STA BB82,Y et BMI 981F. En effet, il faut afficher le «!» avant de tester si c'est fini, sinon on termine sans l'afficher. Notez que grâce au NOP, vous n'aurez pas d'autre souci de mise à jour, le programme ayant la même longueur. Sauvez et lancez.

Horreur le programme affiche le «!» en vidéo inverse ! Dans Sedoric, Fabrice Broche a contourné cette difficulté en ajoutant avant l'affichage les trois instructions PHP (pour sauver la drapeau N), AND #7F (pour forcer le «!» à reprendre sa valeur #21, nous verrons cela dans un autre article) et PLP (pour restaurer le drapeau N). Résultat, pour raccourcir le message de 1 octet (suppression du #00 final), on a rallongé le code de 4 octets ! Quel est le sens de cela ?

Et bien c'est parce que le code de la routine d'affichage est rallongé une fois pour toute et sera utilisé dans Sedoric pour afficher des dizaines de messages, ce qui se traduira par des dizaines d'octets épargnés (au dépend d'une vitesse d'exécution imperceptiblement plus longue). Donc, sauf si vous devez afficher des dizaines de messages, laissez donc tomber cette solution.

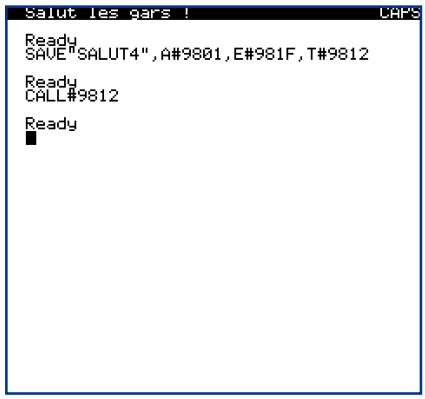
**Encore une variante**

Par contre, au lieu du #00, il est possible d'utiliser d'autres marqueurs de fin de message. On pourrait en fait choisir n'importe quel caractère, par exemple «*» de code Ascii #2A (42 en décimal, c.f. l'annexe 1 du manuel de l'Atmos, page 262). Mais pour détecter cette valeur, après le LDA, il faudra insérer une instruction CMP #2A (compare le contenu du registre A avec la valeur #2A et met le drapeau Z à un si égal) qui prendra 2 octets supplémentaires et donc on est perdant.

Il vaut mieux que le test soit fait directement par le LDA. Or, l'instruction LDA ne peut affecter que les drapeaux Z et N. Donc si on ne veut pas utiliser le marqueur #00 (drapeau Z), il faudra utiliser un marqueur dont le chargement mettra le drapeau N à un. C'est à dire un peu comme dans Sedoric, mais ici ce sera un octet supplémentaire non affiché (pas de problème de vidéo inverse) qui sera utilisé. Faisons simplement un essai avec #FF pour voir, mais n'importe quelle valeur comprise entre #80 et #FF donnerait le même résultat.

Variante 4

```
9801 53 61 6C 75 74 20 6C 65 73 20 67 61 72 73 20 21 FF Salut les gars !
9812 A0 00     LDY #00
9814 B9 01 98  LDA 9801,Y
9817 30 06     BMI 981F
9819 99 82 BB  STA BB82,Y
981C C8        INY
981D D0 F5     BNE 9814
981F 60        RTS
```




Lorsque les 16 caractères du message seront successivement chargés par LDA 9801,Y le drapeau N restera obstinément à zéro car aucun des octets correspondant n'est «négatif» (leur bit de poids le plus fort ou b7 est à zéro dans tous les cas or c'est ce bit qui est reporté dans le drapeau N). Mais au 17e tour, lorsque #FF sera lu (soit **1**111 1111 en binaire le b7 en gras est à un), alors N passera à un et le BMI qui suit détournera l'exécution du programme vers l'adresse indiquée soit 981F où se trouve le RTS marquant la fin de notre petit exemple.

**A propos des instructions de branchement**

Vous avez peut-être observé quelque chose de curieux. Dans tous les exemples ci-dessus, il y a un mystère lorsqu'une instruction de branchement est mise en jeu. Par exemple la ligne «9817  30 06 BMI 981F» devrait vous laisser perplexe. L'octet 30 est bien le code du mnémonique (les 3 lettres qui résument le nom de l'instruction) BMI, mais d'où vient l'octet 06 qui suit et comment fournit-on l'adresse de branchement 981F ?

Réponse n°1 : C'est pour gagner un octet (ici réduction de 33% de la longueur du code, puisqu'au lieu de 3 octets (30 1F 98) on en a seulement deux (30 06).

Réponse n°2 :  06 est une adresse relative (un offset positif ou négatif) que le microprocesseur va ajouter au registre PC où il stocke l'adresse de la prochaine instruction à exécuter (ici 9819, l'adresse

de la ligne suivante). Donc il calcule 9819 + 06 = 981F, qui est bien l'adresse du RTS. Cet offset peut être soit positif (comme dans l'exemple de ce BMI), soit négatif (comme dans le BNE situé en 981D). En effet dans ce dernier cas, l'offset indiqué est #F5. Il s'agit dans ce contexte d'un entier signé de valeur -0B (-11 en décimal). Si l'adresse de l'instruction qui suit le BNE est 981F on a 981F - B = 9814, qui est bien l'adresse où l'on désire reboucler. Vous êtes sans doute perplexe sur ce qu'est un entier signé. Je ne vais pas encore vous noyer dans les explications (il s'agit d'une sombre histoire appelée complément à 2). Sachez que lorsqu'on compte de #00 à #7F, l'équivalent décimal est de 0 à 127, puis lorsqu'on compte de **#80 à #FF, l'équivalent décimal passe de -128 à -1 pour un entier signé**. Vous comprendrez facilement que s'il fallait mettre manuellement un branchement au point, il ne serait pas toujours évident de calculer cet offset. Heureusement les assembleurs font ça pour nous : On indique l'adresse de branchement et l'assembleur calcule l'offset.

Réponse n°3 : L'adressage relatif est 100% relogeable. Ceci signifie que si vous déplacez votre programme ailleurs dans la mémoire, il marchera toujours, puisque le branchement se fera de manière relative. Dans nos deux exemples ci-dessus, le branchement se fera 6 cases en avant de l'instruction suivant le BMI et 11 cases en arrière de l'instruction qui suit le BNE.

Le fait qu'un programme soit relogeable est très important. Imaginez que vous ayez oublié une lettre dans la chaîne de caractères placée devant le code : Les adresses 981F et 9813 devraient alors être corrigées. Dans un programme plus long, cela deviendrait vite une grosse galère.

Un peu plus haut dans cet article, j'ai signalé incidemment que les branchements forcés étaient d'un usage courant. En effet au lieu du BNE 981F, nous aurions pu mettre un JMP 981F (avec toutefois 3 octets au lieu de deux, soit un code 33% plus long). Mais en cas de déplacement dans la mémoire, ce JMP 981F ne serait pas mis à jour...

Allez, pour être complet, il faut ajouter que le branchement relatif peut aller jusqu'à 2 + 127 = 129 en avant et 2 -128 = 126 en arrière. Le 2 provient du fait que le registre PC vise la prochaine instruction située 2 octets plus loin que celle de branchement. Résultat, si vous voulez brancher hors de cette fourchette, il faudra trouver une astuce (style branchements en cascade) ou tomber dans le piège du JMP.

**Les labels**

Je vous avais promis de ne pas me lancer dans les labels, ceci afin de rester dans le cadre d'une initiation. C'est donc seulement pour information que je déroge à ma promesse. Vous avez sûrement compris que le langage machine tire sa simplicité du fait que les instructions sont élémentaires et en nombre limité (une cinquantaine, à comparer aux 127 du Basic). Mais le revers de la médaille est que pour faire la moindre tâche, il faut aligner plusieurs instructions.

Les exemples proposés illustrent bien ce fait, si on les compare avec PRINT«Salut les gars !». Noter toutefois que la commande PRINT ne marche pas pour la ligne service. Voilà une démonstration montrant qu'avec le langage machine, on peut faire plus de choses qu'avec le Basic. Et surtout avec une vitesse d'exécution bien plus grande, puisque dernière la commande PRINT, il y a quelques 120 lignes d'instructions en assembleur (sans compter les méandres de l'interpréteur), au lieu de 7 dans notre exemple. La commande PRINT est plus complexe car elle doit faire face à tous les cas, alors que notre routine est taillée sur mesure.

Quand on pense que Jonathan B. a développé manuellement la plupart de ses merveilleux programmes ! Bref, si vous voulez faire comme lui, vous aurez intérêt à ne commettre aucune erreur dans votre code, sinon, bonjour le problème des corrections, surtout s'il y a des insertions à faire !

Je vous rassure tout de suite, il existe des assembleurs symboliques. A l'aide d'un éditeur, on tape un programme source en assembleur, sauf que les adresses sont remplacées par des labels. Par exemple au lieu de 9801 on mettra SALUT pour indiquer que c'est l'adresse du message «Salut les gars !». Au lieu de 981F on mettra FIN et au lieu de 9814 on mettra REBOUCL, etc. Un gros programme sera découpé en petites unités indépendantes (par exemple affichage, calcul, entrée clavier etc.) auxquelles on accède par des JSR (JSR AFFIC, JSR CALCUL ou JSR CLAVIER). Donc la mise au point en est grandement facilitée, puisque d'une part elle se fait sous-programme par sous-programme et que d'autre part l'implantation en mémoire n'est plus un souci : C'est l'assembleur qui calcule les adresses à partir des labels.

Potassez-bien votre leçon pour la rentrée de septembre…