Initiation à l'Assembleur (7)

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

RAPPELS SUR LES COMPARAISONS (CMP, CPX ET CPY)

La comparaison des registres A, X ou Y avec une valeur quelconque M consiste en fait en une soustraction fictive : Le résultat n'est pas écrit, mais les drapeaux Z, N et C sont affectés. Toutefois, faites attention, avec les instructions "Compare" car :

(1) Il n'y a jamais de retenue et la soustraction effectue tout simplement "A - M".

(2) Le drapeau V n'est jamais affecté.

Ces instructions sont presque toujours suivies d'un Branch. Pour simplifier, on peut dire que:

BEQ branchera si valeur absolue de A = valeur absolue de M.

BNE branchera si valeur absolue de A <> valeur absolue de M.

BPL branchera si valeur absolue de A - valeur absolue de M < #80, soit de #00 à #7F.

BMI branchera si valeur absolue de A - valeur absolue de M >= #80, soit de #80 à #FF.

BCC branchera si valeur absolue de A < valeur absolue de M.

BCS branchera si valeur absolue de A >= valeur absolue de M.

En cas de doute (et dans tous les cas pour BVC et BVS, heureusement peu utilisés), il faudra effectuer la pseudo soustraction "A - M" en binaire pour arriver au bon résultat. Voir les exemples 3 et 4 ci-dessus car on revient à une soustraction sans retenue. En pratique un "Compare" se fait de la manière suivante:

-Ecrire M en binaire, inverser les 8 bits, ajouter 0000 0001.

-Puis, toujours en partant de l'hexadécimal, écrire A en binaire.

-Enfin additionner le tout et déterminer le ou les drapeaux utiles.

RAPPELS SUR LES INSTRUCTIONS BIT (BIt Test)

UTILISATION 'NORMALE' DES INSTRUCTIONS BIT

Elles effectuent un ET logique fictif (le résultat n'est pas écrit, seuls les indicateurs d'état sont affectés) entre A et une cellule mémoire M dont l'adresse suit (BIT LL ou BIT LLHH).

Remarque : L'adresse de la cellule mémoire indiquée pouvant être en page zéro (ce qui nécessite un seul octet) ou absolue (ce qui nécessite deux octets), les commandes BIT LL et BIT LLHH ne diffèrent que par leur code (respectivement #24 et #2C).

Si le résultat de ce ET logique est nul, Z = 1. **Mais, attention N et V reprennent les b7 et b6 de la cellule mémoire M et non du résultat du ET.**

Exemple : A contient #A6 et la cellule mémoire M d'adresse ABCD contient #E0. BIT ABCD effectue un ET logique entre le contenu de A et le contenu de la cellule ABCD:

```
  Hexa :        Binaire :      Décimal :  Observation :
   #A6       1010 0110        166        (ou -90 si nombre signé)
 ET#E0       1110 0000        224        (ou -32 si nombre signé)
  -------------------------
  =#A0      =1010 0000       =160        (ou -96 si nombre signé)
```

Résultat : A et M inchangés, Z = 0, N = 1 et V = 1.

UTILISATION 'SPECIALE' DE L'INSTRUCTION BIT

Les instructions BIT et surtout celle de code #2C (adressage absolu, dont sur deux octets) est cependant utilisée la plupart du temps dans un autre but. Il s'agit d'un "truc" de programmation qui permet de "cacher" **le** (cas de BIT LL) ou **les** (cas de BIT LLHH) octets qui suivent l'instruction BIT.

Si lors de son déroulement, le programme rencontre une instruction BIT (+Adresse) ou même une suite de BIT, l'exécution de ce ou ces BIT reste sans incidence sur le déroulement du programme. Seuls les drapeaux Z, N et V ont été affectés. Tant qu'on ne les utilise pas, cela reste neutre. Des instructions NOP (**N**o **OP**eration) auraient aussi bien fait l'affaire.

Cependant, dans cette zone 'neutre', les adresses indiquées peuvent aussi être utilées à autre chose. Un "Jump" ou un "Branch" adressé sur la cellule mémoire située juste après l'un de ces BIT, entraîne l'exécution du ou des octets "cachés". Ceci ne marcherait pas avec des NOP.

Voici un exemple tiré de la Ram Overlay du Sedoric, situé à partir de l'adresse DD4A et qui présente plusieurs entrées (en DD4A, DD4D, DD50 ou DD53) :

En DD4A on a A9 40 2C A9 C0 2C A9 80 2C A9 00 20 28 DE etc. ce qui peut se désassembler en :

```
DD4A    LDA #40
DD4C    BIT C0A9
DD4F    BIT 80A9
DD52    BIT 00A9
DD55    JSR DE28 etc.
```

Mais il y a aussi une entrée 'cachée' en DD4D où on a A9 C0 2C A9 80 2C A9 00 20 28 DE etc. ce qui peut se désassembler en :

```
DD4D    LDA #C0
DD4F    BIT 80A9
DD52    BIT 00A9
DD55    JSR DE28 etc.
```

De même, il y a une autre entrée cachée en DD50 où on a A9 80 2C A9 00 20 28 DE etc. ce qui peut se désassembler en :

```
DD50    LDA #80
DD52    BIT 00A9
DD55    JSR DE28 etc.
```

Enfin en DD53 on a une dernière entrée 'cachée' avec A9 00 20 28 DE etc. ce qui peut se désassembler en :

```
DD53    LDA #00
DD55    JSR DE28 etc.
```

Ce système permet d'exécuter le sous-programme JSR DE28 avec 4 valeurs initiales dans A, valeurs qui différencient SAVEM, SAVEU, SAVE et SAVEO :

Entrée pour l'exécution de la commande Sedoric SAVEM :

```
DD4A-   A9 40      LDA #40    initialise l'accumulateur avec #40
DD4C-   2C A9 C0   BIT  C0A9  'ne fait rien' et continue en DD4F
```

Entrée pour l'exécution de la commande Sedoric SAVEU :

```
DD4D-   A9 C0      LDA $C0    initialise l'accumulateur avec #C0
DD4F-   2C A9 80   BIT  80A9  'ne fait rien' et continue en DD52
```

Entrée pour l'exécution de la commande Sedoric SAVE :

```
DD50    A9 80      LDA $80    initialise l'accumulateur avec #80
DD52-   2C A9 00   BIT  00A9  'ne fait rien' et continue en DD55
```

Entrée pour l'exécution de la commande Sedoric SAVEO :

```
DD53-   A9 00      LDA $00    initialise l'accumulateur avec #00
DD55-   20 28 DE   JSR  DE28  saut au sous-programme DE28 etc.
```