L2 groupe C - Promo 2019

Systèmes numériques

Rapport de projet

CHAZAL Baptiste - MOUSSEAUX Thomas

16

I. Introduction

Ce projet est une initiation à la manipulation d'un microcontrôleur, ainsi que de son langage machine, dans le cadre des cours de Systèmes Numériques. Ici, nous nous pencherons sur la carte de développement CME11E9-EVBU, et plus précisément sur le microcontrôleur 68HC11 qui le constitue. Ce dernier, développé par Motorola, s'anime autour du langage Assembleur.

Ainsi, dans le cadre de ce projet, nous nous initierons à ce type de langage machine, tout en utilisant la carte de développement. Pour cela, nous travaillerons sur le logiciel AxIDE, qui sert d'intermédiaire entre la carte et l'utilisateur (soit nous). La carte de développement que nous avons eu l'occasion d'utiliser se rapprochait de celle-ci :

On note un port d'alimentation, un port DB9 qui servira à relier la carte avec l'ordinateur sur lequel nous utiliserons AxIDE, un bouton RESET au milieu de la carte pour la réinitialiser, et une "plaquette électronique" sur la droite, sur laquelle nous allons pouvoir brancher des éléments annexes à l'aide de câbles.

II. Prise en main du 68HC11

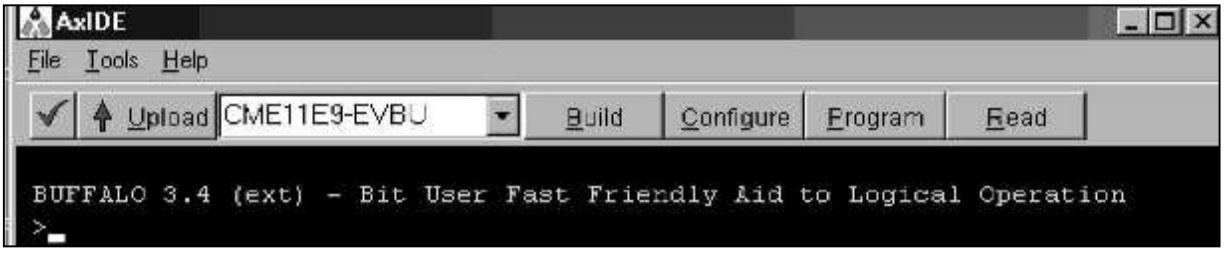
1. Communication avec la carte

Dans un premier temps, il nous faut alimenter la carte (bloc d'alimentation sur 6V), la connecter à l'ordinateur via le câble DB9, ouvrir AxIDE, puis paramétrer la liaison entre les deux via les paramètres suivants :

- Port: COM1
- Vitesse : 9600 Bauds
- Format : 8 bits de donnée + 1 bit de stop
- Contrôle de parité : Pas de parité

La communication avec la carte (soit les instructions que nous utiliserons dans la console) est gérée par le moniteur BUFFALO3.4, qui se trouve directement dans la mémoire interne du processeur.

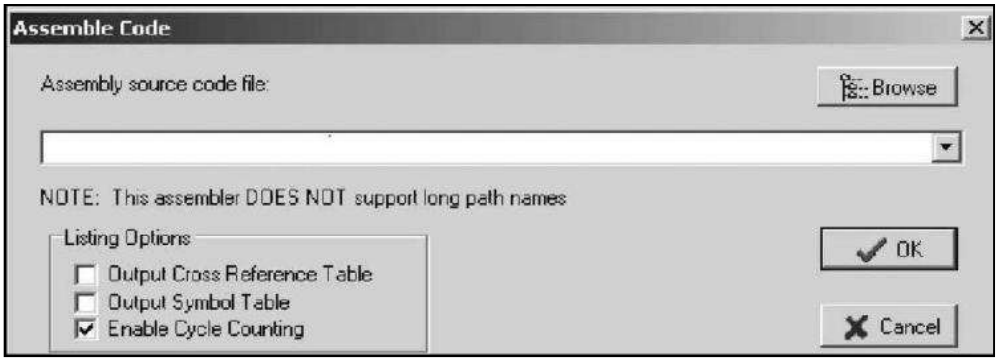
Lorsque tout est paramétré et que la carte est connectée, un appel de commande ("prompt", ici ">") apparaît. Cela signifie que la liaison est bien établie, et que nous pouvons taper une commande.



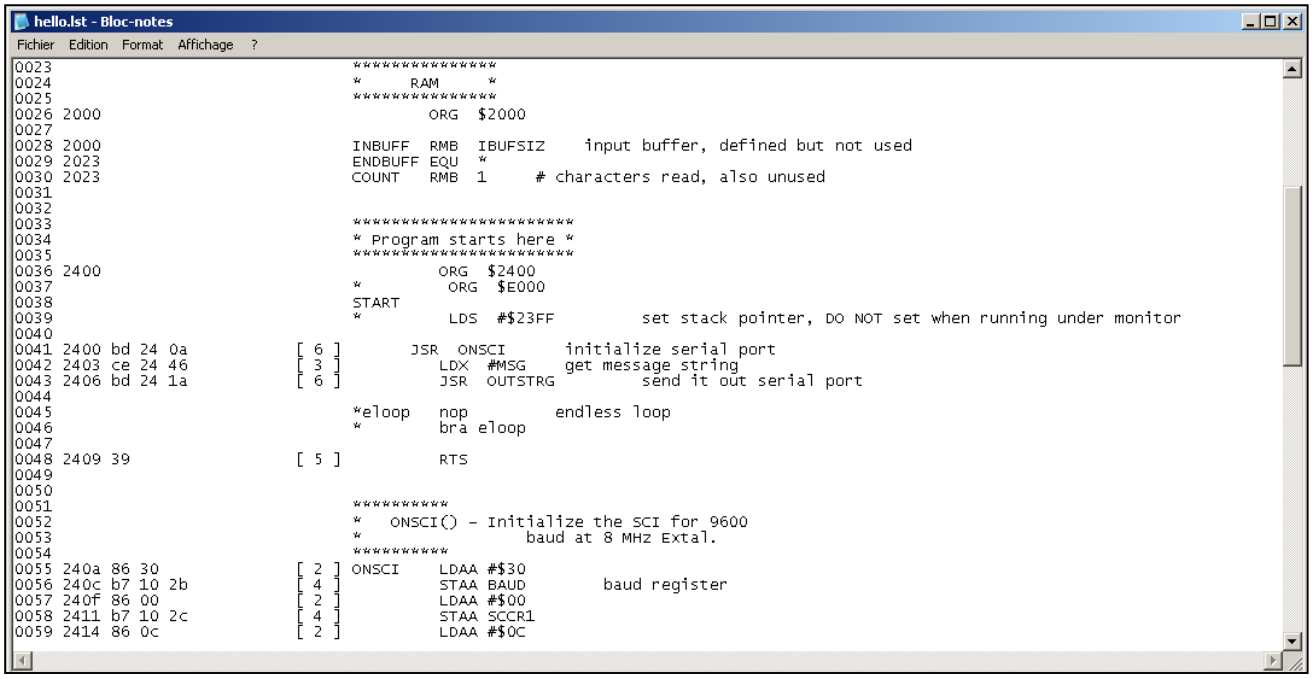
2. Compilation

A présent, il nous faut savoir comment compiler des lignes de codes pour en faire un programme que notre carte pourra suivre. Dans un premier temps, nous allons prendre le fichier "hello.asm", fourni par le constructeur, qui nous servira d'exemple. Il faut différencier ici les fichiers .asm, qui ne contiennent que du texte, des fichiers .s19 qui sont les exécutables, soit les programmes. Notre fichier "hello.asm" contient de nombreuses lignes, des commandes (en assembleur), des directives, des commentaires de code, etc...

Nous allons alors créer un dossier "sn_prj" dans la racine de notre espace disque personnel (sur le réseau), et y placer le fichier "hello.asm". Puis, dans AxIDE, nous allons utiliser la fonction "Build".



Nous allons alors sélectionner "hello.asm" via le bouton "Browse", cocher "Enable Cycle Counting" afin d'identifier le nombre de cycles machine utilisés pour l'exécution, puis valider via le bouton "OK". AxIDE va alors compiler le fichier .asm pour créer un fichier "hello.s19" (soit l'exécutable). Un fichier "hello.lst" sera également créé, contenant nos lignes de code (ce qui était dans "hello.asm"), ainsi que la numérotation des lignes, et diverses informations concernant les adresses et les données, mais également les éventuelles erreurs (ici, il n'y en a pas) :



```
hello.lst - Bloc-notes
Fichier  Edition  Format  Affichage  ?
0023                                  ****************
0024                                  *      RAM     *
0025                                  ****************
0026 2000                                     ORG  $2000
0027
0028 2000                             INBUFF  RMB  IBUFSIZ    input buffer, defined but not used
0029 2023                             ENDBUFF EQU  *
0030 2023                             COUNT   RMB  1     # characters read, also unused
0031
0032
0033                                  ************************
0034                                  * Program starts here *
0035                                  ************************
0036 2400                                     ORG  $2400
0037                                  *        ORG  $E000
0038                                  START
0039                                  *         LDS  #$23FF         set stack pointer, DO NOT set when running under monitor
0040
0041 2400 bd 24 0a            [ 6 ]       JSR  ONSCI      initialize serial port
0042 2403 ce 24 46            [ 3 ]          LDX  #MSG    get message string
0043 2406 bd 24 1a            [ 6 ]          JSR  OUTSTRG         send it out serial port
0044
0045                                  *eloop   nop         endless loop
0046                                  *        bra eloop
0047
0048 2409 39                  [ 5 ]          RTS
0049
0050
0051                                  ***********
0052                                  *   ONSCI() - Initialize the SCI for 9600
0053                                  *                 baud at 8 MHz Extal.
0054                                  ***********
0055 240a 86 30               [ 2 ] ONSCI    LDAA #$30
0056 240c b7 10 2b            [ 4 ]          STAA BAUD        baud register
0057 240f 86 00               [ 2 ]          LDAA #$00
0058 2411 b7 10 2c            [ 4 ]          STAA SCCR1
0059 2414 86 0c               [ 2 ]          LDAA #$0C
```

3. Chargement et exécution

Nous avons désormais notre exécutable "hello.s19", il nous faut désormais l'envoyer vers la carte. Pour cela, nous allons utiliser la fonction "upload". Mais il faut d'abord prévenir la carte que nous allons lui envoyer cet exécutable. Ainsi, dans la console de AxIDE, nous allons entrer la commande "load t", ou simplement "l t", qui est une des directives BUFFALO 3.4 dont nous parlions précédemment. On sélectionne donc le fichier "hello.s19" via la fonction "upload" et on valide. Le logiciel nous annonce alors que le transfert a été effectué correctement en affichant "done" :

On exécute alors le programme via la commande "call 2400" :

```
>l t

done
>call 2400
Hello World
P-2400 Y-FFFF X-2451 A-0D B-90 C-D4 S-0041
>
```

Il s'agit donc d'un programme affichant "Hello World". La ligne qui suit représente des informations concernant certaines adresses typiques. Par exemple, ici, S est le pointeur de pile.

Nous pouvons alors nous demander pourquoi est-ce que l'on écrit "call 2400" pour débuter le programme. Il s'agit tout simplement de l'adresse de début de programme. Si l'on regarde à nouveau le fichier "hello.lst" :

hello.lst - Bloc-notes

Fichier Edition Format Affichage ?

```
0023                                    ****************
0024                                    *      RAM     *
0025                                    ****************
0026 2000                                       ORG  $2000
0027
0028 2000                               INBUFF  RMB  IBUFSIZ    input buffer, defined but not used
0029 2023                               ENDBUFF EQU  *
0030 2023                               COUNT   RMB  1     # characters read, also unused
0031
0032
0033                                    ************************
0034                                    * Program starts here *
0035                                    ************************
0036 2400                                       ORG  $2400
0037                                    *        ORG  $E000
0038                                    START
0039                                    *        LDS  #$23FF         set stack pointer, DO NOT set when running under monitor
0040
0041 2400 bd 24 0a                [ 6 ]       JSR  ONSCI      initialize serial port
0042 2403 ce 24 46                [ 3 ]          LDX  #MSG    get message string
0043 2406 bd 24 1a                [ 6 ]          JSR  OUTSTRG        send it out serial port
0044
0045                                    *eloop   nop         endless loop
0046                                    *        bra eloop
0047
0048 2409 39                      [ 5 ]          RTS
0049
0050
0051                                    ***********
0052                                    *   ONSCI() - Initialize the SCI for 9600
0053                                    *                 baud at 8 MHz Extal.
0054                                    ***********
0055 240a 86 30                   [ 2 ] ONSCI    LDAA #$30
0056 240c b7 10 2b                [ 4 ]          STAA BAUD        baud register
0057 240f 86 00                   [ 2 ]          LDAA #$00
0058 2411 b7 10 2c                [ 4 ]          STAA SCCR1
0059 2414 86 0c                   [ 2 ]          LDAA #$0C
```

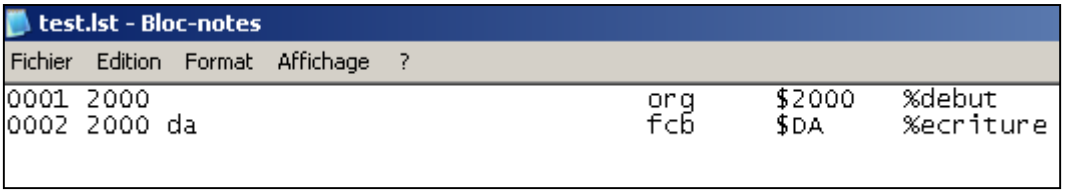
On remarque la ligne "ORG $2400" au début du programme, qui signifie "on se place sur l'adresse 2400". Toutes les lignes se trouvant avant cette instruction ne sont que des directives, c'est donc bien ici que commence le programme. Notons que les lignes entre étoiles "*" correspondent à des commentaires et ne sont pas prises en compte dans le programme. Un peu plus bas nous avons l'instruction "LDS" avec une adresse en hexadécimal, cela signifie "charger cette adresse dans S", avec "LD" pour "LOAD". Il s'agit de l'initialisation du pointeur de pile dont nous avons parlé précédemment. Ensuite, nous avons une instruction "JSR ONSCI" pour initialiser le port série. Puis, juste en dessous, on a "LDX #MSG" qui signifie que le programme va charger "#MSG" dans X ; si l'on descend en bas du fichier, on peut y lire que ce "#MSG" correspond au message "Hello World" : on a donc chargé le message que l'on souhaite afficher dans la console :

```
0101                                    *** TEXT TABLES ***
0102
0103 2446 48 65 6c 6c 6f 20             MSG     FCC   'Hello World'
     57 6f 72 6c 64
0104 2451 04                                    FCB   EOT
0105
0106
0107                                    *        org     $FFFE        set the reset vector
0108                                    *        fdb     START
0109
0110
0111                                             END     START
```

Puis, on a l'instruction "JSR OUTSTRG" qui sert à envoyer le message que l'on a chargé sur le port série, soit à l'afficher dans la console de AxIDE. Finalement, après les deux lignes de commentaires avec des étoiles, on trouve l'instruction "RTS" qui indique que nous sommes à la fin du programme.

4. Ecriture d'une donnée en mémoire vive

A présent, nous allons étudier l'état de la mémoire avant et après extinction de la carte de développement. On va écrire notre propre programme afin de se placer à l'adresse 2000 et d'y écrire une valeur. On utilise donc bloc-notes (ou Notepad++, qu'importe) pour créer un fichier "test.asm". Dans ce fichier, on écrira "org $2000" afin de se placer à l'adresse 2000, puis "fcb $DA" afin d'y inscrire la valeur "DA". "DA" est bien un nombre, il est simplement écrit en hexadécimal (en base 16), comme l'indique le symbole "$". Ici, l'indentation est importante : une directive ou une commande doit être précédée d'un tabulation, sinon il s'agit d'une étiquette. Ensuite, on compile notre programme pour créer "test.s19" et, par conséquent "test.lst". Pour cela, on procède de la même manière que précédemment avec "hello.asm".



Les symboles "%" indiquent que ce qui suit sur la même ligne sont des commentaires, donc ignorés dans le programme, de la même manière qu'avec les étoiles "*". Ensuite, on envoie le programme sur la carte, sans oublier de la prévenir avec "l t". On regarde alors l'état de la mémoire avec la directive "md 2000" qui signifie "affiche l'état de la mémoire à partir de l'adresse 2000". Puis, on éteint l'alimentation de la carte, on la rallume, et on redemande l'état de la mémoire :

```
>l t

done
>md 2000

2000 DA EE 55 5C DD 1A EF C7 76 BB 2E 75 8D 53 E1 71   U\    v .u S q
2010 FF FC 69 19 5D 1F 18 1E E6 C6 1F FF FE A2 FF EF   i ]
2020 5F 93 92 DD 8D AC FB B3 FF 1F 7B FF 4B 1D F6 AF _           K
2030 F4 0B 5E F3 FF 10 EC 6D 0E 3C 73 5A C0 EF 7F EA   ^    m <sZ
2040 FE B9 EF 72 8B FC 35 D8 D4 C8 9D 36 43 50 5B B6    r  5    6CP[
2050 2B DB 51 D8 5B A0 9C D5 47 27 BB F8 7B DF 65 F0 + Q [   G'    e
2060 FB B7 7E 5F 6A 6D AB 7B A1 BF 56 EE AF 5E D7 FA    _jm    V  ^
2070 77 2F 54 6B FD 8F 5F E9 9E F1 D6 ED 7F 7B DF 7F w/Tk  _
2080 F7 F3 F6 ED CD 3E D9 28 8D BE AF C3 7B 7F E1 D3      > (
>
BUFFALO 3.4 (ext) - Bit User Fast Friendly Aid to Logical Operation
>md 2000

2000 C3 EE 55 5C DD 1A EF C3 76 BA 6E 65 8D 53 E5 F1   U\    v ne S
2010 FF FC 69 19 5D 1F 18 1E E6 C6 5F FD FE A2 FF EF   i ]     _
2020 5F 93 92 DF AD AC FB BB FB 1F 7B FF 4B 19 F6 AF _           K
2030 D4 0B 5E F3 FF 10 EC 6D 0E 2C 73 7A C2 ED 7F EA   ^    m ,sz
2040 BE F9 EF 72 8B FC 35 D8 D0 4A 95 76 43 50 5B 96    r  5  J vCP[
2050 2B DB 71 D8 5B A0 9C D5 57 27 BB F8 7B D7 65 F1 + q [   W'    e
2060 FB B7 7F 5F 6A 6D AB 7B A1 BF 56 EE AF 5E D7 FA    _jm    V  ^
2070 77 2F 54 6B F9 87 5D F9 9E F1 D6 ED 7F 7B 5F 7F w/Tk  ]       _
2080 F5 D2 F6 6D CD 3E D8 28 8D BE AF C3 7B 7F E1 D3    m > (
>
```

Sur chaque ligne se trouvent 16 adresses mémoire, l'indice de la première de la ligne étant indiqué au début de la ligne. Par exemple, sur la première ligne, on a les adresses de 2000 à 200F (on compte ici en hexadécimal). Ainsi, sur le premier état de la mémoire, on note que l'on a bien "DA" à l'adresse 2000, comme nous l'avions demandé. Cependant, après extinction de la carte, l'adresse 2000 ne contient plus la valeur "DA". On note tout de même certaines ressemblances entre les deux états de la mémoire (certaines valeurs n'ont pas changé), mais les deux états de la mémoire ne sont pas identiques. On remarque ainsi que la mémoire vive ne se conserve pas après extinction de la carte.

Tentons autre chose à présent. Nous écrivons désormais ceci dans un fichier "test2.asm" :

```
      org     $2000
val   rmb     1
      org     $200A
      ldaa    #$DA
      staa    val
      rts     val
```

Puis, comme précédemment, on le compile en un programme et on obtient "test2.s19" que l'on envoie sur la carte, et "test2.lst" que voici :

```
test2.lst - Bloc-notes
Fichier  Edition  Format  Affichage  ?
0001 2000                                  org     $2000
0002 2000                            val   rmb     1
0003 200a                                  org     $200A
0004 200a 86 da                [ 2 ]       ldaa    #$DA
0005 200c b7 20 00             [ 4 ]       staa    val
0006 200f 39                   [ 5 ]       rts     val
```

Analysons ce programme. Celui-ci commence à l'adresse 2000 comme on peut le voir sur la première instruction "org $2000" qui signifie que l'on se place sur l'adresse 2000. Ensuite, on a "rmb 1" qui signifie que l'on réserve un espace de 1 octet. La première colonne étant destinée aux étiquettes comme nous l'avons évoqué plus tôt, on remarque que l'on a étiqueté cet espace avec le nom "val". Puis, on se place sur l'adresse "200A". L'instruction "ldaa #$DA" signifie que l'on charge dans l'accumulateur 'a' la valeur "DA" qui est en hexadécimal. Ensuite, l'instruction "staa val" signifie que l'on range le contenu de l'accumulateur 'a' dans "val". Enfin, "RTS" indique la fin du programme.

Ainsi, sur la capture d'écran ci-dessus, la première colonne correspond à la numérotation des lignes, la seconde correspond aux adresses, la colonne avec "val" est la colonne des étiquettes, la suivante correspond aux instructions et la dernière correspond aux données. La colonne avec les valeurs entre crochets indique le nombre de fois où l'instruction est utilisée. Quant aux valeurs après les adresses (86, da, b7, etc...), il s'agit de nombres qui sont attribués par l'ordinateur aux instructions. On a donc ici "86" pour "ldaa", ou "b7" pour "staa", ou encore "39" pour "rts". D'ailleurs, si l'on ouvre le fichier exécutable "test2.s19" avec bloc-notes, on remarque qu'il est constitué des instructions, sous la forme de ces nombres :

```
TEST2.S19 - Bloc-notes
Fichier  Edition  Format  Affichage  ?
S109200A86DAB72000395C
S9030000FC
```

Si l'on observe l'état de la mémoire à partir de l'adresse 2000, on note la présence de "DA" à l'adresse 2000. Rappelons que l'on utilise pour cela la directive "md 2000". D'ailleurs, si l'on tape dans la console "md 200F", on obtient le même résultat, car il s'agit de la même ligne :

```
BUFFALO 3.4 (ext) - Bit User Fast Friendly Aid to Logical Operation
>l t

done
>call 200A

P-200A Y-FFFF X-FFFF A-DA B-FF C-D8 S-0041
>md 2000

2000 DA EE 55 5C DD 3A EF C3 76 BA 86 DA B7 20 00 39   U\ :  v      9
2010 FF FE 69 19 5D 1F 1A 1E E6 C6 5F FF FE A2 FF EF   i ]      _
2020 5F 93 92 DD AD A4 FB B3 FF 1F 7B FD 4B 19 F6 AF _           K
2030 F4 0B 5E F3 7F 10 EC 6D 0E 2C 73 5B D2 EF 7F AA   ^    m ,s[
2040 BE B9 FF 72 8B FD B5 D8 D4 CA 95 32 43 50 5B B6    r       2CP[
2050 2B DB 75 D8 5B A0 9C D5 47 27 BB F8 7B D7 65 F0 + u [   G'    e
2060 FA B7 7C 5F 6A 6D AB 7B A1 BF 56 FE AF 5E D3 FA    _jm    V  ^
2070 77 2F 55 6B F9 8F 5D EB BE F1 D2 ED 7F 7B 5F 7F w/Uk  ]       _
2080 F7 F3 F6 6D CD 3E D9 28 8D B6 BF C3 7B 7F E1 D3    m > (
>md 200F

2000 DA EE 55 5C DD 3A EF C3 76 BA 86 DA B7 20 00 39   U\ :  v      9
2010 FF FE 69 19 5D 1F 1A 1E E6 C6 5F FF FE A2 FF EF   i ]      _
2020 5F 93 92 DD AD A4 FB B3 FF 1F 7B FD 4B 19 F6 AF _           K
2030 F4 0B 5E F3 7F 10 EC 6D 0E 2C 73 5B D2 EF 7F AA   ^    m ,s[
2040 BE B9 FF 72 8B FD B5 D8 D4 CA 95 32 43 50 5B B6    r       2CP[
2050 2B DB 75 D8 5B A0 9C D5 47 27 BB F8 7B D7 65 F0 + u [   G'    e
2060 FA B7 7C 5F 6A 6D AB 7B A1 BF 56 FE AF 5E D3 FA    _jm    V  ^
2070 77 2F 55 6B F9 8F 5D EB BE F1 D2 ED 7F 7B 5F 7F w/Uk  ]       _
2080 F7 F3 F6 6D CD 3E D9 28 8D B6 BF C3 7B 7F E1 D3    m > (
>
```

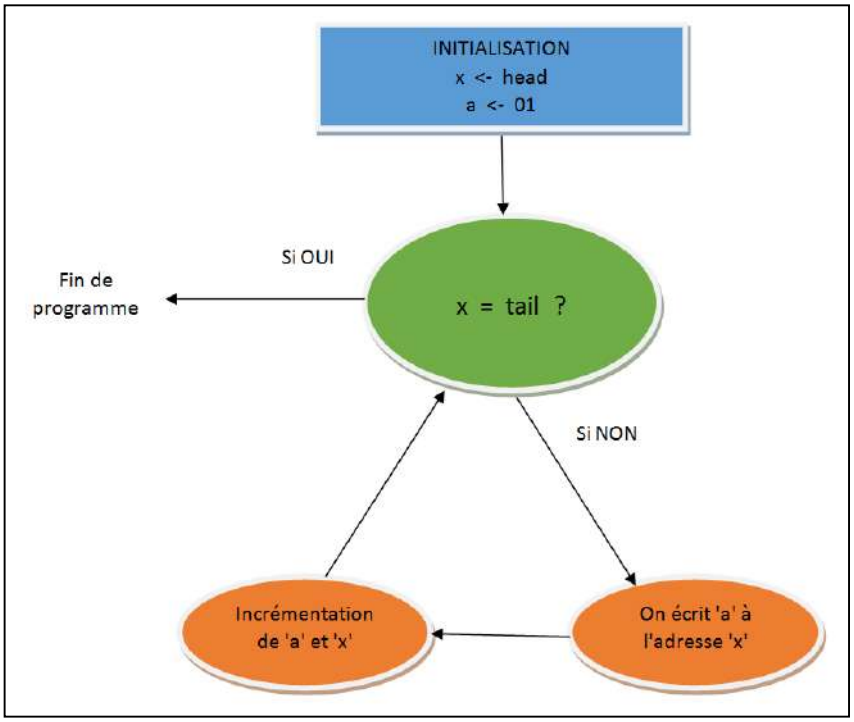
Intéressons-nous maintenant aux boucles. Nous créons le fichier "test3.asm" dans lequel on écrit le programme suivant :

```
head   equ    $2000
tail   equ    $2009
       org    $200A
       ldaa   #$01
       ldx    #head
loop   staa   0, x
       inca
       inx
       cpx    #tail
       bne    loop
       rts
       end
```

Puis on le compile et on l'envoie dans la carte de la même manière que précédemment.

Comme la ligne "org $200A" nous l'indique, l'adresse de début de programme est "200A". Ce programme sert à incrémenter à partir de 1 les adresses entre 2000 et 2009. Penchons nous sur son fonctionnement :

Les deux premières lignes servent à étiqueter l'adresse 2000 avec le nom "head", et l'adresse 2009 avec le nom "tail". Ensuite, on se place sur l'adresse 200A. On charge la valeur "01" dans l'accumulateur 'a' via l'instruction "ldaa #$01", et on charge "head" (donc 2000) dans 'x'. Ensuite, il s'agit d'une boucle. On identifie la première instruction de la boucle en lui étiquetant le nom "loop", cette instruction sert à écrire 'a' à l'adresse 'x'. Ensuite, on va incrémenter 'a' ainsi que 'x' avec les instructions "inca" et "inx" ; l'instruction n'est pas exactement la même, hormis la lettre désignant 'a' ou 'x', car l'un est une valeur et l'autre est une adresse. Ensuite, l'instruction "cpx #tail" sert à comparer la valeur actuelle de 'x' avec la valeur de "tail", soit 2009. Tant que ce n'est pas le cas, c'est à dire tant que la boucle n'a pas atteint l'adresse 2009 étiquetée par "tail", alors l'instruction "bne loop" nous renvoie à la ligne étiquetée "loop", soit le début de la boucle. Sinon, c'est la fin du programme ("rts").



Observons le fichier "test3.lst" :

```
6: Warning --- Indexed Addressing Assumed
0001 2000                          head  equ    $2000
0002 2009                          tail  equ    $2009
0003 200a                                org    $200A
0004 200a 86 01              [ 2 ]       ldaa   #$01
0005 200c ce 20 00           [ 3 ]       ldx    #head
6: Warning --- Indexed Addressing Assumed
0006 200f a7 00              [ 4 ] loop  staa   0, x
0007 2011 4c                 [ 2 ]       inca
0008 2012 08                 [ 3 ]       inx
0009 2013 8c 20 09           [ 4 ]       cpx    #tail
0010 2016 26 f7              [ 3 ]       bne    loop
0011 2018 39                 [ 5 ]       rts
0012                                     end
```

On retrouve une fois de plus la numérotation des lignes, les adresses, les étiquettes, les instructions, les données, les compteurs d'instructions (combien de fois cette instruction est effectuée), et les valeurs attribuées aux instructions. Notons néanmoins un avertissement de la part d'AxIDE, "Indexed Addressing Assumed", soit "adressage indexé supposé", certainement puisque nous utilisons dans nos instructions des étiquettes là où l'instruction attend normalement une adresse (mais ces étiquettes renvoie sur des adresses comme nous l'avons vu précédemment). Le fichier "test3.s19", quant à lui, contient bien les instructions sous leur forme de nombre (ici "86" pour "ldaa", "8c" pour "cpx", ou encore "26" pour "bne"), comme nous l'avons vu plus tôt.

En demandant l'état de la mémoire à partir de l'adresse 2000 (puisque c'est l'adresse étiquetée "head"), on voit que ce programme est totalement fonctionnel :

```
BUFFALO 3.4 (ext) - Bit User Fast Friendly Aid to Logical Operation
>l t

done
>call 200A

P-200A Y-FFFF X-2009 A-0A B-FF C-D4 S-0041
>md 2000

2000 01 02 03 04 05 06 07 08 09 BA 86 01 CE 20 00 A7
2010 00 4C 08 8C 20 09 26 F7 39 C6 5F FF FE A2 FF EF  L     & 9 _
2020 5F 93 92 DD AD A4 FB B3 FF 1F 7B FD 4B 19 F6 AF _           K
2030 F4 0B 5E F3 7F 10 EC 6D 0E 2C 73 5B D2 EF 7F AA  ^    m ,s[
2040 BE B9 FF 72 8B FD B5 D8 D4 CA 95 32 43 50 5B B6    r       2CP[
2050 2B DB 75 D8 5B A0 9C D5 47 27 BB F8 7B D7 65 F0 + u [   G'    e
2060 FA B7 7C 5F 6A 6D AB 7B A1 BF 56 FE AF 5E D3 FA    _jm    V  ^
2070 77 2F 55 6B F9 8F 5D EB BE F1 D2 ED 7F 7B 5F 7F w/Uk  ]       _
2080 F7 F3 F6 6D CD 3E D9 28 8D B6 BF C3 7B 7F E1 D3    m > (
>
```

On peut également demander la trace du programme, via la commande "trace" ou "t" dans la console d'AxIDE :

```
>md 2000

2000 01 02 03 04 05 06 07 08 09 BA 86 01 CE 20 00 A7
2010 00 4C 08 8C 20 09 26 F7 39 C6 5F FF FE A2 FF EF  L     & 9 _
2020 5F 93 92 DD AD A4 FB B3 FF 1F 7B FD 4B 19 F6 AF _            K
2030 F4 0B 5E F3 7F 10 EC 6D 0E 2C 73 5B D2 EF 7F AA   ^     m ,s[
2040 BE B9 FF 72 8B FD B5 D8 D4 CA 95 32 43 50 5B B6    r        2CP[
2050 2B DB 75 D8 5B A0 9C D5 47 27 BB F8 7B D7 65 F0 + u [   G'    e
2060 FA B7 7C 5F 6A 6D AB 7B A1 BF 56 FE AF 5E D3 FA    _jm    V  ^
2070 77 2F 55 6B F9 8F 5D EB BE F1 D2 ED 7F 7B 5F 7F w/Uk  ]        _
2080 F7 F3 F6 6D CD 3E D9 28 8D B6 BF C3 7B 7F E1 D3    m > (
>trace
LDAA #$01           P-200C Y-FFFF X-2009 A-01 B-FF C-90 S-0041
>
LDX  #$2000         P-200F Y-FFFF X-2000 A-01 B-FF C-90 S-0041
>
STAA $00,X          P-2011 Y-FFFF X-2000 A-01 B-FF C-90 S-0041
>
INCA                P-2012 Y-FFFF X-2000 A-02 B-FF C-90 S-0041
>
INX                 P-2013 Y-FFFF X-2001 A-02 B-FF C-90 S-0041
>
CPX  #$2009         P-2016 Y-FFFF X-2001 A-02 B-FF C-99 S-0041
>
BNE   $200F         P-200F Y-FFFF X-2001 A-02 B-FF C-99 S-0041
>
STAA $00,X          P-2011 Y-FFFF X-2001 A-02 B-FF C-91 S-0041
>
INCA                P-2012 Y-FFFF X-2001 A-03 B-FF C-91 S-0041
>
INX                 P-2013 Y-FFFF X-2002 A-03 B-FF C-91 S-0041
>
CPX  #$2009         P-2016 Y-FFFF X-2002 A-03 B-FF C-99 S-0041
>
BNE   $200F         P-200F Y-FFFF X-2002 A-03 B-FF C-99 S-0041
>
STAA $00,X          P-2011 Y-FFFF X-2002 A-03 B-FF C-91 S-0041
>
INCA                P-2012 Y-FFFF X-2002 A-04 B-FF C-91 S-0041
>
INX                 P-2013 Y-FFFF X-2003 A-04 B-FF C-91 S-0041
>_
```

Grâce à cette directive, on peut voir l'état de 'a' et de 'x' (ainsi que des pointeurs P et S) étape par étape, pour chaque instruction du programme.

III. Fréquencemètre logiciel

1. Méthode par scrutation

Nous allons à présent utiliser les cycles machines en adoptant une minuterie sur le programme suivant :

```
0001 100e                          TCNT equ     $100E   % CP
0002 2000                               org     $2000   % début
0003 2000 fe 10 0e            [ 5 ]     ldx     TCNT    % CP1
0004 2003 01                  [ 2 ]     nop             % rien
0005 2004 18 fe 10 0e         [ 6 ]     ldy     TCNT    % CP2
0006 2008 39                  [ 5 ]     rts             % retour  t
```

En utilisant la minuterie TNTC, on remarque que deux cycles séparent les deux chargements ldx et ldy.

```
BUFFALO 3.4 (ext) - Bit User Fast Friendly Aid to Logical Operation
>l t

done
>call 2000

P-2000 Y-ADDA X-ADD2 A-FF B-FF C-D8 S-0041
>
```

Pour cela, il faut s'appuyer sur le registre. On sait par analyse décimale que X = 2 et Y = 6. Si l'on effectue l'opération Y-X, on obtient donc 6 - 2 = 4. Cela nous permet de déduire que l'horloge du CPU et celle du TNTC sont les mêmes.

Sur ce programme, nous avons choisi d'ajouter 4 occurrences de « nop ».

```
l tmd call0001 100e                                TCNT        equ      $10
0002 2000                                  org      $2000
0003 2000                          initial      rmb      2
0004 2002                          delta        rmb      2
0005 2050                                  org      $2050
0006 2050 fc 10 0e            [ 5 ]        ldd      TCNT
0007 2053 fd 20 00            [ 5 ]        std      initial
0008 2056 01                  [ 2 ]        nop
0009 2057 01                  [ 2 ]        nop
0010 2058 01                  [ 2 ]        nop
0011 2059 01                  [ 2 ]        nop
0012 205a fc 10 0e            [ 5 ]        ldd      TCNT
0013 205d b3 20 00            [ 6 ]        subd     initial
0014 2060 fd 20 02 md            [ 5 ]     std      delta
0015 2063 39                  [ 5 ]        rts
0016                                       end  elta
```

On obtient 8 cycles séparant les deux opérations de chargement.

```
P-2050 Y-604F X-6047 A-00 B-12 C-D0 S-0041
>md 2000

2000 5A 0C 00 12 18 FE 10 0E 39 DF 28 B7 76 7A 27 E8 Z       9 ( vz'
2010 FB FB 99 B9 F7 34 3C DD DC DD BD E7 B9 07 57 EE      4<       W
2020 B3 DB B2 AC 08 5B 55 3F A0 BD AB 79 3D DA FB 65      [U?   y=  e
2030 73 53 BD CF 72 7F F8 DF D9 32 43 47 7B 7D 6E 98 sS  r    2CG  n
2040 45 B5 0C 36 52 DB 76 F7 93 77 D6 B8 72 27 DC E6 E  6R v  w  r'
2050 FC 10 0E FD 20 00 01 01 01 01 FC 10 0E B3 20 00
2060 FD 20 02 39 5F 1D 3F FD 47 8C A7 89 F5 3B E6 32    9_ ? G    ; 2
2070 92 5A F7 94 6D 1F 9B F7 FE C3 EB 6F 5F 21 BC 26  Z  m      o_! &
2080 ED 17 E6 6B 7F B1 33 C4 79 A7 3E CD AA 24 3B B5    k  3 y >  $;
>md 2002

2000 5A 0C 00 12 18 FE 10 0E 39 DF 28 B7 76 7A 27 E8 Z       9 ( vz'
2010 FB FB 99 B9 F7 34 3C DD DC DD BD E7 B9 07 57 EE      4<       W
2020 B3 DB B2 AC 08 5B 55 3F A0 BD AB 79 3D DA FB 65      [U?   y=  e
2030 73 53 BD CF 72 7F F8 DF D9 32 43 47 7B 7D 6E 98 sS  r    2CG  n
2040 45 B5 0C 36 52 DB 76 F7 93 77 D6 B8 72 27 DC E6 E  6R v  w  r'
2050 FC 10 0E FD 20 00 01 01 01 01 FC 10 0E B3 20 00
2060 FD 20 02 39 5F 1D 3F FD 47 8C A7 89 F5 3B E6 32    9_ ? G    ; 2
2070 92 5A F7 94 6D 1F 9B F7 FE C3 EB 6F 5F 21 BC 26  Z  m      o_! &
2080 ED 17 E6 6B 7F B1 33 C4 79 A7 3E CD AA 24 3B B5    k  3 y >  $;
>
```

D'après le registre ci-dessus, on remarque qu'avant l'adresse $2050 le registre n'est pas modifié. Il en est de même à l'adresse $2002, le registre n'est modifié qu'à partir de l'adresse $2050.

Selon le schéma fonctionnel du registre, le registre acteur qui utilise le port PA2 utilise comme entrée du signal le registre TCI1. Ce registre TCI1 permet d'enregistrer la valeur TCNT au front montant sur l'entrée du port PA2.

Pour lire le programme suivant, on va utiliser deux registres : TCI1 et TFLG1 . Ils sont dits de captures d'entrées. Il nous faut également un registre de configuration qui est TCTL2. L'entrée du programme est $1021 et le front actif est le front descendant. On déduit que le registre scruté est TFLG1 et que le bit scruté est le bit à l'adresse $04. Tant que sa valeur = 0, on retourne sur la ligne lire 1 avec la commande Beq. Il s'agit de la même démarche dans la seconde partie du programme avec lire 2. Le registre de capture est « front » qui notifie le premier front descendant.

IV. Conclusion

Lors de ces deux séances, nous avons appris à utiliser, comprendre et modifier des registres. Il nous a fallu tout d'abord comprendre le langage assembleur des registres puis interagir avec celui-ci via le logiciel AxIDE dont la prise en main est assez délicate. Malgré quelques difficultés, ces deux séances et le sujet ont été très intéressants et enrichissants. Il est ainsi regrettable de ne pas avoir pu le terminer entièrement.