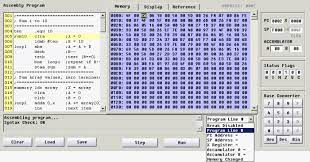
|  |  |  |
| --- | --- | --- |
| **Logo_Lycée** | **Langage machine et assembleur**  **(Assembleur 68xx)** | logo%20ac%20orl%E9ans%20toursDescription : Description : Description : pemDescription : Description : Description : pem |
| **TP**  **Approfondissement** | **Classe :** \_\_\_\_\_\_\_\_  **Nom :** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Prénom :** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |



**Figure 1** : SDK6800-6811

**Mots-clés :** adresse, donnée, instruction machine, mnémonique, opcode, opérande, directive d’assemblage, mode d’adressage, assembleur, variable, constante, opération symbolique.

**Capacité attendue**

Dérouler l’exécution d’une séquence d’instructions simples du type langage machine.

**Table des matières**

[1. Introduction 2](#_Toc175750417)

[2. Les différents niveaux de langage 2](#_Toc175750418)

[3. Langage d’assemblage et code machine 2](#_Toc175750419)

[3.1 Organisation du code source 6800 2](#_Toc175750420)

[3.2 Un premier programme 2](#_Toc175750421)

[3.3 Le simulateur SDK-6800 3](#_Toc175750422)

[3.4 Identification du code machine du premier programme 3](#_Toc175750423)

[3.5 Multiplication en machine 4](#_Toc175750424)

[3.6 Rôle du registre d’état (Status Register) 5](#_Toc175750425)

[3.7 Instructions de rupture de séquence 5](#_Toc175750426)

[3.8 Sous-programme 8](#_Toc175750427)

**MATÉRIELLES ET LOGICIEL**

Micro-ordinateur, simulateur SDK-6800 http://www.hvrsoftware.com/6800emu.htm

**CONDITION DE RÉALISATION**

1 élève par poste informatique

**PAGE DE PRÉSENTATION DU TP**

https://bit.ly/3EtqG6w

**RESSOURCES**

6800 Instruction Set https://webge.fr/6800.html

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

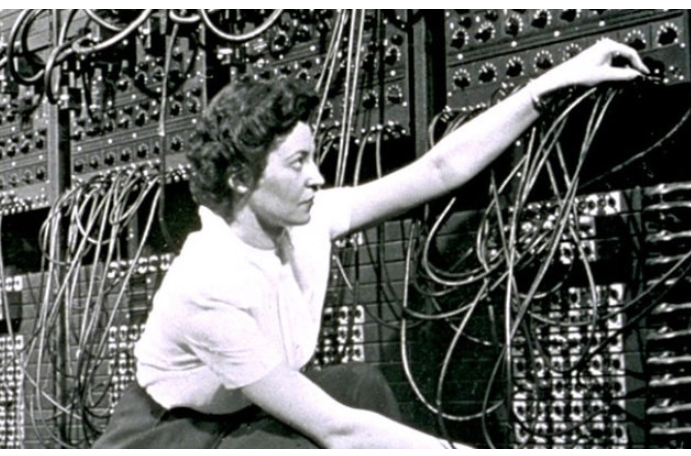
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

****



# 1. Introduction

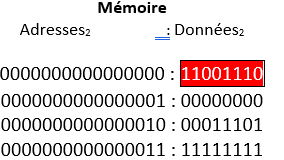


**Figure 2** : 1943 ENIAC premier calculateur

*« La programmation s’effectue de nos jours dans des environnements dédiés, voire avec des* ***interfaces graphiques****. L’introduction de la notion de* ***programme enregistré*** *au sein des premiers calculateurs dans les années 1940-50 a permis de passer de* ***l’automate câblé*** *à la machine informatique. Les premiers programmes étaient construits avec le* ***langage du processeur****, constitué* ***d’instructions-machine.*** *Aujourd’hui, les processeurs sont toujours construits selon une architecture et disposent d’un* ***jeu d’instructions****. »*

# 2. Les différents niveaux de langage

Plusieurs niveaux de langage de programmation coexistent sur un ordinateur. Au plus près de l’utilisateur, le **langage de haut niveau** (Python, C, Java, etc.) permet au programmeur d’exprimer simplement un algorithme. Ce langage de haut niveau est indépendant de la machine. Le programme est donc compatible avec les machines, quel que soit leur processeur.

****

**Figure 3** : codes machine

Au **niveau matériel**, le processeur a été défini avec un **jeu d’instructions-machine**. Ces instructions sont des **chaînes binaires** qui codent des actions élémentaires (addition, chargement de registres, manipulation de mots en mémoire) entre les registres et la mémoire et mettent en jeu les circuits de l’unité arithmétique et logique (**UAL**). Le **processeur** n’exécutant que les **instructions-machine**, un programme de haut niveau doit être traduit vers son équivalent en langage machine à l’aide d’un logiciel (**compilateur**, **interpréteur**, ou une combinaison des deux).

**ldx #$001D**

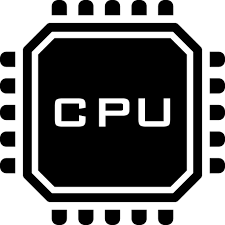
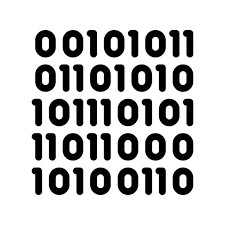
Instruction en

Langage d’assemblage 6800

En tant que **programmeur**, il est très courant de travailler avec un langage de haut niveau. Cependant, certaines parties d’applications tels que les **systèmes d’exploitation** ou les **gestionnaires de périphériques** ont desoin de s’interfacer directement avec le matériel ce qui nécessitent une programmation de **bas niveau**. Dans ce cas, le langage utilisé est appelé **langage d’assemblage**.



**Logiciel Assembleur**



**Code source en** **langage d’assemblage**

**Code machine en mémoire**

**Traducteur**

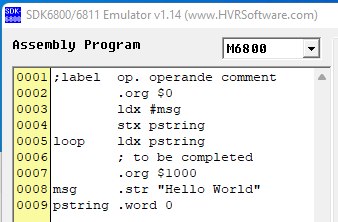
Le langage d’assemblage est un équivalent du langage machine pour lequel les chaînes binaires de l’instruction-machine sont remplacées par des **mnémoniques** alphanumériques plus aisément mémorisable et manipulable par un être humain. Un autre **traducteur** transforme ce langage vers le langage machine équivalent : c’est **l’assembleur**.

# 3. Langage d’assemblage et code machine

## 3.1 Organisation du code source 6800

Le code source se décompose en **champs** de texte :

**[*Étiquette*] Opération [*Opérande(s)*] [*Commentaire*]**



**Figure 4** : champs d'un code source en assembleur

* Le champ***Étiquette*** *ou* ***label*** est utilisé pour définir un symbole. Pour ignorer ce champ, on introduit au moins un espace ou une tabulation.
* Le champ **Opération** est occupé par un **opcode** ou une **directive d'assemblage**.
* Le champ **Opérande(s)** contient une adresse ou des données. Il est ignoré lorsque l’instruction utilise le mode d’adressage implicite (mis en œuvre page 4).
* Le champ **Commentaire** est utilisé pour la documentation du logiciel. Un commentaire commence par un point-virgule et peut se situer à la fin ou au début d’une déclaration.

## 3.2 Un premier programme

**Opération attendue** : **v3 ← v1 + v2** # Les variables v3, v2 et v1 sont des octets, 0 ≤ vx10 ≤ 255

**Étape 1 - Déclarer, initialiser et positionner les variables dans la mémoire**

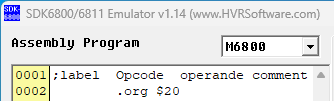
Pour cela, nous allons utiliser des **directives d’assemblage** et des **étiquettes**.

|  |  |
| --- | --- |
| **Informations** | Les **directives** sont des **pseudo-instructions** : elles ne correspondent à aucune instruction-machine ; ce sont des ordres destinés à l’assembleur. |

Exemple : pour placer des variables à partir de l’adresse **002016** dans la mémoire du simulateur, il suffit d’écrire la directive **.org $20** comme ci-dessous.

**Directive d’assemblage**

**.org**



Le symbole **$** signifie que la valeur qui suit est en **hexadécimal** (base 16)

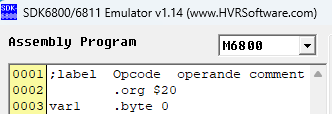
La déclaration d’une variable de type octet nécessite une étiquette et la directive d’assemblage **: .byte**

**Étiquette**

**v1**

|  |  |
| --- | --- |
| **Informations** | Une **étiquette** ou **label** est une chaîne de caractères permettant de **nommer** une **instruction** ou une **variable**. Une étiquette correspond à une **adresse** dans le programme. |

Exemple : on déclare une variable **v1** à la position **$20** et on l’initialise à **0** comme ci-dessous.



|  |  |
| --- | --- |
| *Adresse* | *Donnée* |
| **$0020** | **00** |

L’étiquette **v1** est une adresse en mémoire.

Numéros de ligne

**Étape 2 – Positionner en mémoire et écrire le traitement à réaliser sur les variables**

Rappel : on souhaite effectuer l’opération **v3 ← v1 + v2**

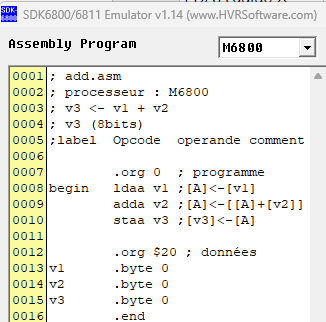


**Figure 5** : accumulateurs du processeur 68xx

Les registres du processeur sont mis à contribution dans cette deuxième étape. Le MC6800 possède seulement deux **accumulateurs** (**A** et **B**) pour effectuer des calculs. La manière dont les registres accèdent aux données est appelée : **mode d’adressage**.

Le programme est réalisé avec **3 instructions** :

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction n°** | **Description** | **Instruction 6800** | **Opération symbolique** |
| **1** | **Charger** la valeur (**opérande**) située à la position mémoire **v1** dans l’accumulateur **A.** | **ldaa** *v1* | **[A] ← [***v1***]** |
| **2** | **Ajouter** le contenu de l’accumulateur **A** à la valeur située à la position mémoire **v2** (le résultat de l’opération est automatiquement placé dans A) | **adda** *v2* | **[A] ← [A] + [***v2***]** |
| **3** | **Stocker** le contenu de l’accumulateur **A** à la position mémoire **v3.** | **staa** *v3* | **[***v3***]** **← [A]** |



Lecture d’une **opération symbolique**

**[]** ⬄ contenu de

Ex : [v1] : contenu de la mémoire à l’adresse v1

**←**: placer dans

Ex : [A] ←[v1] : placer le contenu de la mémoire à l’adresse v1 dans le registre A

**Figure 6** : code source v3 <- v1 + v2

Remarque : dans cet exemple, les instructions **ldaa** *v1***,**  **adda** *v2* et **staa** *v3* mettent en œuvre le **mode d’adressage étendu**.

|  |  |
| --- | --- |
| **Informations** | **Dans le mode d’adressage étendu**, l'adresse contenue dans le deuxième octet de l'instruction est utilisée comme octet supérieur de l'adresse de l'opérande. Le troisième octet de l'instruction est utilisé comme octet inférieur de l'adresse de l'opérande. Il s'agit d'une **adresse absolue** dans la mémoire.  ldaa, adda et staa sont codées sur **3 octets**.  **Exemple : ldaa $1000** charge le contenu de la mémoire située à l’adresse 100016 dans l'accumulateur A |



Mémoire du simulateur

(extrait)

## 3.3 Le simulateur SDK-6800

L’assembleur place les instructions-machine en mémoire conformément aux directives d’assemblage. La **mémoire du simulateur** est présentée sous forme **matricielle** comme ci-dessous :

|  |  |  |
| --- | --- | --- |
| ***Mémoire*** | |  |
| ***Adresse16*** | ***Donnée16*** | ***Commentaire*** |
| 0022 |  | v3 |
| 0021 |  | v2 |
| 0020 |  | v1 |
|  |  |  |
| 0008 |  |  |
| 0007 |  |
| 0006 |  |  |
| 0005 |  |  |
| 0004 |  |
| 0003 |  |  |
| 0002 | 20 | v1 |
| 0001 | 00 |
| 0000 | B6 | ldaa |

***Adresses Données***

0000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0010 ↓ ↓

Adresse = 000716 Adresse = 000F16

Les adresses et les données sont exprimées en **hexadécimal**.

## 3.4 Identification du code machine du premier programme

***Objectif :*** *identifier le code machine correspondant à un programme écrit en assembleur.*

1. **Préparez** votre dossier **home** sur le NAS\_SINen suivant la fiche **"Organisation du dossier de travail".**
2. **Ouvrez** le fichier ***add.asm*** dans le simulateur**SDK-6800/6811.**
3. **Exécutez** le programme en mode **pas à pas, observez** l’évolution du contenu des registres et de la mémoire.
4. **Complétez** le tableau ci-contre avec les résultats de la simulation.
5. **Donnez** la valeur de v1, v2 et v1 + v2 en décimal, en hexadécimal et en binaire.

\_\_v1\_=\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_ v2\_=\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_ v1 + v2 \_=\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Remplacez** les étiquettes v1, v2 et v3 par les adresses correspondantes dans le code et ci-dessous. Testez cette nouvelle version du programme.

[label] [opcode][operand][comment]

**ldaa** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**adda** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**staa** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Remplacez** la valeur de v1 par $FF et celle de v2 par 1 dans le code source. Testez le programme. Pourquoi obtient-on ce résultat dans v3 ?

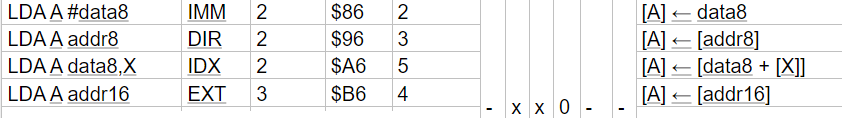
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| Avertissement avec un remplissage uni | Les instructions du MC6800 sont décrites ici : <https://webge.fr/6800.html>. **Cliquez** sur le mnémonique **LDA** (**L**oa**D** **A**cumulator) pour avoir sa description et ses **modes d’adressage**. Un extrait est donné ci-dessous. |

Exemple : quelques modes d’adressage de l’accumulateur A





**IMM** : Adressage immédiat **DIR** : adressage direct **IDX** : adressage indexé sur X **EXT** : adressage étendu

1. D’après le code généré par l’assembleur pour l’instruction ***ldaa*** *v1*, quel **mode d’adressage** a été utilisé parmi les 4 proposés dans le tableau ci-dessus ?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 3.5 Multiplication en machine

**Objectif** : multiplier deux octets et sauvegarder le résultat en mémoire soit : **v3 ← v1 \* v2**

# Les variables v1 et v2 sont des octets.

# La variable v3 est un mot sur 16bits, 0 ≤ v3 ≤ ?\_\_\_\_\_\_\_\_\_\_\_\_ À chercher

# Elle est constituée de 2 octets **v3H** et **v3L** (H pour High et L pour Low)

1. **Ouvrez** le fichier ***mul.asm*** dans le simulateur **SDK-6800/6811. Placez** le simulateur en **mode M6811.**
2. **Testez** le programme avec v1 = 1010 et v2 = 4010**.**

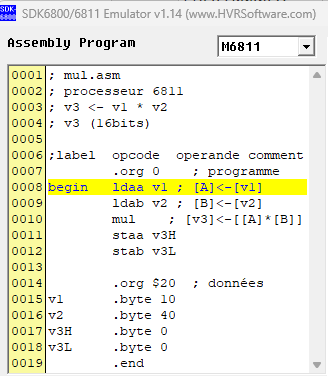
On utilise le simulateur en mode [6811](https://www.cs.uaf.edu/2007/fall/cs441/proj1notes/sawyer/inst.html) [**bit.ly/3ditlC2**] pour effectuer la **multiplication**. Le 6800 n’en avait pas !

SYNTAX Mode BYTES CODE CYCLES SYMBOLIC OPERATION

mul **INH** 1 $3D 10 [A] ← ([A] \* [B])/256

[B] ← ([A] \* [B])\*256

**3.** Le résultat sur 16bits de l’opération est constitué avec les registres A et B. Valeur du mot AB ? **AB16 = \_\_\_\_\_\_\_\_\_\_\_\_\_**



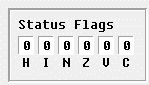
|  |  |
| --- | --- |
| **Informations** | **INH** signifie Implicite (inhérent)  Dans le **mode d'adressage implicite**, l'instruction donne l'adresse de manière inhérente (c'est-à-dire un pointeur de pile, un registre d'index, etc.). Les instructions inhérentes sont utilisées lorsqu'aucun opérande ne doit être récupéré. Ce sont des instructions sur 1 octet. |

4. **Convertissez** 19016 en décimal.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**5.** Conclusion : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 3.6 Rôle du registre d’état (Status Register)



**Figure 7** : registre d'état

Les bits du registre d’état servent d’**indicateurs** (également appelés « drapeaux » ou ***flags***) sur l’état du processeur.

**1. Illustration du rôle des bits du registre d’état**

**a.** Reprenez le code ***add.asm*** (**v3 ← v1 + v2**). Les calculs se font sur **8 bits**. Utilisez *add.asm* pour complétez le tableau ci-dessous.

-128 0 127

**Débordement arithmétique ⇒ V = 1**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **v110** | **v210** | **v310**  attendu | **v316**  **(obtenu en $22)** | **N** | **Z** | **V** | **C** | **v310** obtenu  **non signé** | **v310** obtenu  **complément à 2** |
| 1 | 0 | 0 | 0 | 00 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 255 | 1 | 256 |  |  |  |  |  |  |  |
| 3 | 255 | 0 | 255 |  |  |  |  |  |  | -1 |
| 4 | 255 | 255 | 510 |  |  |  |  |  | 254 |  |
| 5 | 127 | 1 | 128 |  |  |  |  |  |  | -128 |

Les résultats dans **N**, **Z**, **V** et **C** doivent correspondre à la description ci-dessous.

|  |  |
| --- | --- |
| **Informations** | Le **bit N** (***N****egative*) est mis à **1** si le résultat de l’instruction est **négatif (complément à 2)**, sinon il est mis à 0. C’est la recopie du bit de poids fort du résultat.  Le **bit Z** (***Z****ero*) est mis à **1** si le résultat de l’instruction est **nul**, sinon il est mis à 0.  Le **bit V** (o**V**erflow) est mis à **1** si l’instruction génère un **débordement** [arithmétique](https://fr.wikipedia.org/wiki/Indicateur_de_d%C3%A9bordement) (https://bit.ly/3mjzBio), sinon il est mis à 0.  Le **bit C** (***C****arry*) est mis à **1** si l’instruction génère une **retenue**, sinon il est mis à 0. |

Illustration du débordement arithmétique sur un mot de 8bits

|  |  |
| --- | --- |
| **Informations** | **Les *flags* sont utilisés** par les **instructions de rupture de séquence** telles que**: jmp, bra, beq, bne, bcc, bcs,** etc. |

**b.** À quelle condition le résultat des lignes 3 et 5 est-il conforme à l’attendu ?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 3.7 Instructions de rupture de séquence

|  |  |
| --- | --- |
| **Informations** | Les **ruptures de séquence** sont les **sauts**, les **branchements** et les **appels et retours de procédure**. <https://webge.fr/6800.html>. |

Début

Fin

[A] ← [v1]

[A] ← [[A] + [v2]]

**carry=1**

[v3L] ← [A]

[A] ← 1

[A] ← 0

?

ldaa v1

?

?

?

bcs setv3H

bra fol

[v3H] ← [A]

?

**1. Addition de 2 mots de 8 bits (résultat sur 16bits)**

**Objectif**: coder le résultat de l’opération **v3 ← v1 + v2** sur **16bits.**

**v1** et **v2** sont des octets. **v3** est un mot sur **16bits** constitué de **v3H** et **v3L**.

Le code suivant est disponible dans ***addv2.asm***

*;label opcode operande comment*

.org 0

begin ldaa v1 ; *début du programme*

adda v2

staa v3L

**bcs** setv3H

clra

**bra** fol

setv3H ldaa #1

fol staa v3

.org $20 *; données*

v1 .byte 255

v2 .byte 0

v3H .byte 0 ; v3H, octet de poids fort de v3

v3L .byte 0 ; v3L, octet de poids faible de v3

.end

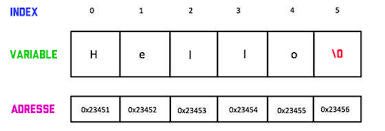
|  |  |
| --- | --- |
| **Informations** | **#** devan**t 1** dans l’instruction **ldaa #1** identifie une **valeur. L’adressage** est **immédiat**: [A] ←1**.** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **v110** | **v210** | **v310** | **v316 = v3Hv3L** | **Commentaires** |
| 255 | 0 | **255** | **00FF** |  |
| 255 | 255 | **510** | **01FE** |  |

**a. Testez** le code en mode pas à pas et comparez v316 à v310 dans les commentaires du tableau.

**b. Complétez** l’algorigramme ci-dessus avec les **instructions** **assembleur** du programme.

**2. Chaîne de caractères**



***Objectif****:* *détecter la fin d'une chaîne de caractères.*

|  |  |
| --- | --- |
| **Informations** | En informatique, une **chaîne de caractères** est conceptuellement une suite ordonnée de **caractères** et physiquement une suite ordonnée d'unités de code. |

En assembleur, la fin d'une chaîne de caractères est identifiée par le caractère **ASCII NULL** (**0x00**). Il est automatiquement ajouté à la fin de la chaîne par l'assembleur lorsqu’elle est déclarée avec la directive **.str**.

Exemple : 001D 48 65 6C 6C 6F 20 57 6F 72 6C 64 00 msg .str "Hello World"

|  |  |
| --- | --- |
| **Informations** | Le caractère de fin de chaîne facilite la manipulation d’une chaîne par exemple lors du transfert d’une chaîne de la mémoire centrale dans la mémoire de l’écran comme dans le code ci-dessous. Celui-ci balaye" la chaîne "*Hello*" tant qu’un caractère **NULL** n'est pas détecté. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithme** FinChaine  **Variables**  msg <- "Hello" : chaîne de caractères  **Registres**  X : reg 16bits, adresse d'un caractère de la chaîne msg  A : reg 8bits, valeur pointée par X  **début**  X <- adresse du premier caractère de msg  **faire**  A <- \*X // \*X est la valeur pointée par X  X <- X+1  **tant que** (A ≠ 0)  **fin** | ;label opcode operand comment  start ldx #msg ; on charge l'adresse du premier caractère  ; de msg dans le registre x  loop ldaa 0,X ; [A] ← [0 + [X]]  cmpa #0 ; on compare [A] avec 0  **beq** fol ; si Z = 1 alors branchement à fol  inx ; sinon [X] ← [[X] + 1]  **bra** loop ; branchement inconditionnel à l’étiquette loop  fol nop ; suite du programme  .org $20 ; adresse de début de la chaîne  msg .str "Hello" ; une chaîne de caractère est entourée de "  .end ; fin du code source | |  |  |  | | --- | --- | --- | | Adresse | ASCII |  | | 0020 | 48 | 'H' | | 0021 | 65 | 'e' | | 0022 | 6C | 'l' | | 0023 | 6C | 'l' | | 0024 | 6F | 'o' | | 0025 | 00 | NULL | |

1. **Complétez l’algorigramme** correspondant au programme *FinChaine* avec les **instructions** en **assembleur**.

bra loop

Début

[X] ← #msg

[A] ← [0 + [X]]

[A] = 0

[X] ← [X] + 1

ldx #msg

Suite prog

ldaa 0,x

|  |  |
| --- | --- |
| **Informations** | **Adressage indexé**. Dans l’instruction **ldx #msg**, **x** est un **registre d'index**. Un registre d'index participe au calcul de l'adresse d'un opérande durant l'exécution d'un programme, par exemple pour faire des **opérations répétitives sur les éléments d'un tableau**. |

1. **Test 1 en mode 6800** : analyse **du premier passage dans la boucle** du programme

Les instructions ***cmpa***, ***beq*** et ***nop*** sont dans la ressource [*https://webge.fr/6800.html*](https://webge.fr/6800.html)*.* **Chargez** le code de l'exercice (situé dans le fichier ***finchaine.asm****)* dans le simulateur, **exécutez-le** en mode pas-à-pas et complétez le tableau ci-dessous lorsque [A] = 4816.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Adresses16 | **opcode**16 and opérand16 | [label] [**opcode**] [operand] | X16 | A16 | Status Flag Z | PC16 |
| 0000 | 00 | 0 | 0000 |
| 0000 | **CE** 00 20 | start **ldx** #msg | 0020 | 00 | 0 |  |
| 0003 | **A6** 00 | loop **ldaa** 0,x | 0020 | 48 |  |  |
| 0005 | **81** 00 | **cmpa** #0 | 0020 | 48 |  |  |
| 0007 | **27** 03 | **beq** fol | 0020 | 48 |  |  |
| 0009 | **08** | **inx** | 0021 | 48 |  |  |
| 000A | **20** F7 | **bra** loop | 0021 | 48 |  |  |
| 000C | **01** | fol **nop** |  |  |  |  |
|  |  | .org $20 |  |  |  |  |
| 0020 | 48 65 6C 6C 6F 00 | msg .str "Hello" |  |  |  |  |

Remarque : L’assembleur produit l’opérande F716 dans l’instruction bra loop car, d’après la ressource l’instruction bra disp effectue

l’opération [[PC](https://webge.fr/6800.html#PC-reg)] [←](https://webge.fr/6800.html#transfer-desc) [[PC](https://webge.fr/6800.html#PC-reg)] + [disp](https://webge.fr/6800.html" \l "disp-desc) + 2.

Donc @arrivée = @départ + disp + 2 soit disp = @arrivée - @départ - 2 => disp = 000316 - 000A16 - 216 = 310 - 1010 – 210 = -910.

Or 910=000010012 donc -910 = 111101102 + 1 = 111101112 = **F716**

1. **Test 2 en mode 6800** : analyse du programme lorsqu’il a atteint le caractère de fin de la chaîne

**Remplacez** *ldx #msg* par *ldx #$25 pour que le programme pointe sur le caractère de fin de chaîne* ***00*** *à l’adresse* ***$0025***. **Complétez** le tableau ci-dessous.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Adresses16 | **opcode**16 opérand16 | [label] [**opcode**] [operand] | X16 | A16 | Status Flag Z | PC16 |
| 0000 | 00 | 0 | 0000 |
| 0000 | **CE** 00 20 | start **ldx** #msg | 0025 | 00 | 0 | 0003 |
| 0003 | **A6** 00 | loop **ldaa** 0,x | 0025 |  |  |  |
| 0005 | **81** 00 | **cmpa** #0 | 0025 |  |  |  |
| 0007 | **27** 03 | **beq** fol | 0025 |  |  |  |
| 0009 | **08** | **inx** |  |  |  |  |
| 000A | **20** F7 | **bra** loop |  |  |  |  |
| 000C | **01** | fol **nop** | 0025 |  |  |  |
|  |  | .org $20 |  |  |  |  |
| 0020 | 48 65 6C 6C 6F 00 | msg .str "Hello" |  |  |  |  |

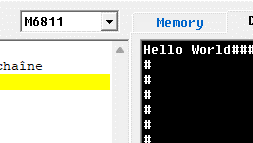
1. Que fait le programme lorsqu’il a détecté le caractère ASCII NULL ?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Le programme *FinChaine* peut être optimisé en supprimant l’instruction **cmpa #0** car Z=1 (condition de branchement) dès que [A]=0. Faites-le et vérifiez le fonctionnement.

**3. Transfert d’une zone mémoire vers une autre**

***Objectif****:* *afficher une chaîne de caractères sur l’écran de l’émulateur.*



**Présentation**

Le programme précédent peut être légèrement modifié pour afficher la chaîne de caractère "***Hello World***" sur l’écran de l’émulateur. Pour faciliter la programmation, nous allons le positionner en **M6811** et ainsi disposer de **Y**, un deuxième **registre d’index**.

L’écran se compose de **20 lignes** de **54 caractères**. L’adresse du premier caractère situé en haut et à gauche de l’écran est **FB00**16. L’écran est visible en sélectionnant l’onglet *Display*.

**Cahier des charges**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mémoire pointée par : **X** |  |  | Mémoire pointée par : **Y** | Ecran |
| $0020 | H | A copier vers-> | $FB00 | H |
| $0021 | e | A copier vers-> | $FB01 | e |
| $0022 | l | A copier vers-> | $FB02 | L |
| $0023 | l | A copier vers-> | $FB03 | L |
| $0024 | o | A copier vers-> | $FB04 | 0 |

Le programme ***hello*** est construit en modifiant le programme *finchaine* fait précédemment. Pour cela, il suffit de lui ajouter judicieusement les instructions suivantes :

|  |  |
| --- | --- |
| Opérations symboliques | Instructions assembleur 6811 |
| [Y] ← [Y] + 1 | iny |
| [Y] ← #$fb00 | ldy #$fb00 |
| [0 + [Y]] ← [A] | staa 0,y |

1. **Chargez** le code de l'exercice (***hello.asm****)* dans le simulateur. **Complétez l’algorigramme** correspondant au programme ***hello*** avec les **instructions** en **assembleur**.

ldy #$fb00 ; adresse du 1er caractère de l’écran

Début

[X] ← #msg

[A] ← [0 + [X]]

[A] = 0

[0 + [Y]] ← [A]

ldx #msg

[Y] ← #bscreen

[X] ← [X] + 1

[Y] ← [Y] + 1

beq fol

Suite prog

ldaa 0,x

1. **Testez** le programme.

Appel prof

Pour faire vérifier le fonctionnement

**4. Remplir une zone mémoire avec un caractère ASCII**

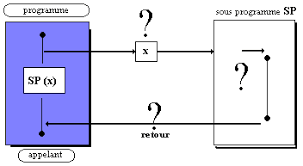
***Objectif****:* *effacer l’écran de l’émulateur.*

Le code ci-dessous peut effacer (partiellement ou totalement) l'écran de l’émulateur. Chargez le fichier ***cls.asm*** et testez-le.

|  |  |
| --- | --- |
| [label] [opcode][operand][comment  ; Constantes (@ ⬄ adresse)  bscreen .equ $FB00 ; @ de la première position de l'écran  escreen .equ $FB14 ; @ de la dernière position à effacer  space .equ $20 ; code ASCII du caractère espace  .org 0  cls ldaa #space  ldx #bscreen  loop staa 0,x  inx  cpx #escreen  bne loop  .end | **Algorithme** ClearScreen  **Constantes**  bscreen = $FB00, adresse de la première position de l'écran : mot 16bits  escreen = adresse de la dernière position à effacer : mot 16bits  space = $20 : code ASCII du caractère espace : octet 8bits  **Registres**  X <- bscreen : registre 16bits  A <- space : registre 8bits  **début**  A <- space  X <- bscreen  **faire**  \*X <- A // \*X est la valeur pointée par X  X <- X+1  **tant que** (X ≠ escreen)  **fin** |

**a.** **Modifiez** le programme pour qu'il efface la première ligne puis tout l'écran. Notez ci-dessous les modifications apportées.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



## 3.8 Sous-programme

***Objectif****:* *effacer l’écran de l’émulateur et afficher "Hello World".*

|  |  |
| --- | --- |
| **Informations** | **En informatique, un sous-programme est** un programme secondaire pouvant être utilisé en plusieurs points d'un programme principal. |

Le programme précédent peut être transformé en un **sous-programme** en le terminant avec l’instruction **RTS**. Ceci a été réalisé dans le fichier ***cls2.asm.*** *Chargez-le*dans l’émulateur. Vous constaterez que le code comprend deux parties le sous-programme identifié par l’étiquette ***cls*** et se terminant avec l’instruction **rts** et le programme principal identifié par l’étiquette ***main***.

**a.** **Testez** le programme en mode pas à pas puis **modifiez**-le pour qu’il efface la première ligne et la remplace par le texte : *Hello World*.

Pour faire vérifier le fonctionnement

Appel prof