|  |  |  |
| --- | --- | --- |
| **Logo_Lycée** | **Langage machine et assembleur** | logo%20ac%20orl%E9ans%20toursDescription : Description : Description : pemDescription : Description : Description : pem |
| **TP2**  **Approfondissement** | **Programmation en Assembleur 6800**  Opérations arithmétiques, transferts en mémoire, tests et boucles |

**🎯 Objectifs du TP**

**Date** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Classe** : \_\_\_\_\_\_\_

**Nom** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Prénom** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* A. Détecter les dépassements, multiplier. 🔧 Notions : Flags C et V, instruction MUL.

Maîtriser les flags N, Z, V, C. 🔧 Notions : N, Z, V, C

* B. Maîtriser boucles et branchements. 🔧 Notions : BRA, BEQ, BNE

**Prérequis :** TP1 - Découverte Assembleur 6800

**Ressource :** <https://webge.fr/6800.html>

**A. Arithmétique avancée**

🎯 **Objectifs** : Détecter les dépassements, multiplier.

🔧 **Notions** : Flags C et V, instruction MUL.

**A.1. Dépassement de capacité et flags N, V, C**

**🎯 Objectif** : Comprendre le comportement des registres 8 bits lors d’une addition sur des **nombres signés** et **non signés**.

**Rappel** : organisation du programme source

Les instructions d'assemblage contiennent les champs suivants :

**[Label] Operation [operand] [comment]**

**📚 Remarque** : chaque champ doit être séparé par au moins un espace.

**📝 Rappel**

Un registre 8 bits peut stocker des valeurs comprises entre **0** et **255**.

**🧪 Exercice A.1.1 : Dépassement lors d'une addition (nombres non signés)**

On effectue l’opération **sum ← 200₁₀ + 100₁₀** dans le programme ci-dessous,

**Q1.** **a)** Avant d'exécuter le programme, prédisez :

* Donnez le résultat de 200₁₀ + 100₁₀ sur 9 bits : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (utilisez une calculatrice)
* On place ce résultat dans le registre A (8 bits) du 6800. Que contiendra A (en binaire) ? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Donnez ce résultat en hexadécimal \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_et en décimal : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lancez** le simulateur en cliquant sur  et chargez le fichier ***depassement.asm*** situé dans **TP2\_SDK68xx**.

**; SDK6800 - depassement.asm**

**RAPPELS**

**Champ operation** : contient un **opcode** ou une **directive d’assemblage**

**Directives d’assemblage**

**.org** : la directive ORG indique l'adresse de départ du code assemblé.

**.byte** : la directive BYTE indique que la valeur qui suit est un octet.

**Adressage immédiat**

**📚Dans l’adressage IMMÉDIAT l’opérande est une donnée. Cet adressage est identifié par dièse le symbole dièse (#).**

Ex : ldaa #200 ; [A] <- 20010

**; opération : Addition de deux entiers non signés**

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldaa #200 ; [A] ← 200 en décimal ou [A] ← $C8 en hexa

adda #100 **;** [A] ← 200 + 100 en décimal

staa sum ; [sum] ← [A]

.org $0020 ; Origine des données en mémoire (variables)

sum .byte 0

**Assemblez** le code en cliquant **une fois** sur 

**b)** **Relevez** le contenu de la mémoire (**code machine** ci-dessous et identifiez 10010 et 20010 dans ce code.

**0000**: \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

**c) Exécutez** le programme pas à pas et complétez :

| **Après l'instruction** | **A (hexa)** | **A (décimal)** | **Flag N** | **Flag V** | **Flag C** |
| --- | --- | --- | --- | --- | --- |
| ldaa #200 | C8 | 200 | 1 | 0 | \_\_\_\_\_ |
| adda #100 | \_\_\_\_\_ | \_\_\_\_\_ | 0 | 0 | \_\_\_\_\_ |

**d) Analysez** les résultats :

C

* Après l’opération adda #100, le registre A contient : \_\_\_\_\_\_\_\_\_\_\_\_ (hexa) = \_\_\_\_\_\_\_\_\_\_\_\_ (décimal)
* Le flag **C** (Carry) vaut : \_\_\_\_\_\_\_\_\_\_\_\_\_
* **Conclusion** sur le résultat de l’opération dans sum. Quelle opération utilisant C et sum permet de trouver le bon résultat \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**+1**

-128 0 127

**Débordement arithmétique ⇒ V = 1**

**-1**

**📚 Remarque** : Quand **C=1**, cela signifie qu'il y a eu un dépassement (retenue) voir l’**annexe 1**.

**🧪Exercice A.1.2 : Dépassement lors d'une addition (nombres signés, importance du bit V [oVerflow])**

**📚 Remarque : Le flag V concerne les nombres signés (complément à 2)**

V et N

**Modifiez** le programme comme ci-dessous :

**; SDK6800 - depassement.asm**

**; Opération : Addition de deux entiers signés (signe +)**

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldaa **#127 ;** [A] ← 127

adda **#1** ; [A] ← [A] + 1

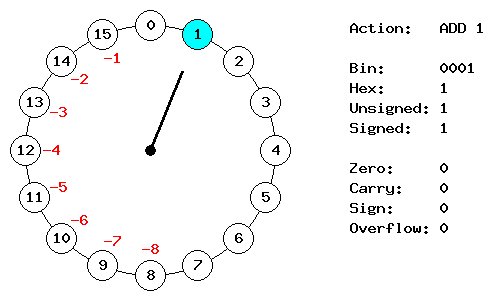
staa sum ; [sum] ← [A]

.org $0020 ; Origine des données en mémoire

sum .byte 0 ; (variables)

**Q2 a) Exécutez** le programme pas à pas et complétez :

| **Après l’instruction** | **A (hexa)** | **A (décimal signé)** | **N** | **V** | **C** |
| --- | --- | --- | --- | --- | --- |
| adda #1 | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ |



**Figure 1** : animation (Sign = N, Overflow = V)

**b) Interprétation :**

* Après adda #1 le registre A contient \_\_\_\_\_\_\_\_\_₁₆ = \_\_\_\_\_\_\_\_\_\_(décimal)
* Flag V = \_\_\_\_\_\_\_\_\_\_ indique un dépassement en arithmétique signée
* Flag N = \_\_\_\_\_\_\_\_\_\_ indique un nombre négatif

**Modifiez** le programme comme ci-dessous :

**; SDK6800 - depassement.asm**

**; Opération : Addition de deux entiers signés (signe -)**

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldaa **v1 ;** [A] ← [v1]

adda **v2** ; [A] ← [A] + [v2]

**Rappel : adressage étendu (TD1)**

**Dans l’adressage ÉTENDU, l’opérande est une adresse**.

Ex : ldaa v1 ; vi ⬄ $0020

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

staa sum ; [sum] ← [A]

.org $0020 ; Origine des données en mémoire (variables)

**v1 .byte -128 ; (80 en hexa)**

**v2 .byte -1 ; (FF en hexa)**

sum .byte 0

**c) Testez** avec v1=-128 (soit 80₁₆) et v2=-1 (soit FF₁₆)

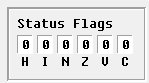
| **Après l’instruction** | **A (hexa)** | **A (signé)** | **N** | **V** | **C** |
| --- | --- | --- | --- | --- | --- |
| adda v2 | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | 1 |

**d) Interprétation :**

* Après adda v2 le registre A contient \_\_\_\_\_\_\_\_\_\_\_₁₆ = \_\_\_\_\_\_\_\_\_\_\_ en décimal signé
* Flag V = \_\_\_\_\_\_\_\_\_\_ indique un dépassement en arithmétique signée
* Flag N = \_\_\_\_\_\_\_\_\_\_ indique un nombre \_\_\_\_\_\_\_\_\_\_\_\_\_ **Flag C = 1 car -128 – 1 = -129 n’est pas représentable dans A**.

**A.2. Le résultat de l’opération est nul et flag Z**

**Modifiez** le programme comme ci-dessous :



**Figure 2** : registre d’état des microprocesseurs

6800/6811

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldaa **v1 ;** [A] ← [v1]

adda **v2** ; [A] ← [A] + [v2]

staa sum ; [sum] ← [A]

.org $0020  ; Origine des données en mémoire (variables)

**v1 .byte -1 ; (FF en Hexa)**

**v2 .byte 1**

sum .byte

**Q3 a) Testez** avec v1=-1 (soit FF₁₆) et v2=1

Z

| **Après l’instruction** | **A (hexa)** | **A (décimal signé)** | **N** | **Z** | **V** | **C** |
| --- | --- | --- | --- | --- | --- | --- |
| staa sum | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ |

**b) Interprétation :**

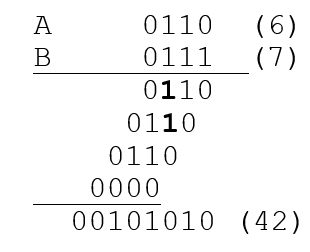
* Après staa sum la case mémoire contient \_\_\_\_\_\_\_\_\_\_\_
* Flag Z = \_\_\_\_\_\_\_\_\_\_ indique \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**💡 Points clés à retenir**

* **Z=1** : le résultat est exactement zéro (**très utile pour les boucles !)**
* **N=1** : le bit 7 (poids fort) est à 1, **nombre négatif en complément à 2**
* **V=1** : V est consulté pour vérifier si un **dépassement de capacité** s'est produit lors d'une **opération signée**. Cet indicateur est positionné à 1 lorsque le résultat est inférieur au minimum négatif (ici 8016) ou supérieur au maximum positif (ici 7F16).
* **C=1** : C est utilisé pour vérifier si un **dépassement de capacité** s'est produit lors d'une **opération non signée**. Cet indicateur est mis à 1 lorsque le résultat est inférieur au minimum (ici 0016) ou supérieur au maximum (FF16).

**A.3. Multiplication en machine**

**🎯 Objectif :** Effectuer l'opération**v3 ← v1 × v2**



**Figure 3** : multiplication de 2 nombres codés sur 4 bits

Exemple : multiplication

**📝 Spécifications :**

* Les variables v1 et v2 sont des octets codés en binaire naturel (non signés).
* La variable v3 est un mot sur 16 bits, 0 ≤ v3 ≤ 2¹⁶ - 1
* v3 est constituée de 2 octets v3H (octet de poids fort) et v3L (octet de poids faible)
* H pour High et L pour Low

**ℹ️ Information**

Le processeur **MC6811** dispose d'une instruction **MUL** qui multiplie les contenus de A et B.  
Le résultat 16 bits est stocké dans le registre double **D** constitué de A et B (D ⬄ A|B).

Avant MUL : A = v1 (8 bits)

B = v2 (8 bits)

**📚 Remarque :** On utilise le simulateur en mode [6811](https://www.cs.uaf.edu/2007/fall/cs441/proj1notes/sawyer/inst.html) [https://**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

Après MUL : D = A × B (16 bits)

A contient la partie haute (v3H)

B contient la partie basse (v3L)

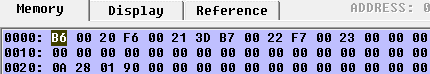
**🧪 Exercice A.3.1 : Test avec dépassement 8 bits**

**Sélectionnez 6811** etchargez le fichier ***mul.asm*** situé dans **TP2\_SDK68xx**..

**; SDK6811 - mul.asm**

**; opération : v3 ← v1 x v2**





**Figure 4**: code machine de mul.asm (non exécuté)

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldaa v1 ; [A] ← [v1]

ldab v2 ; [B] ← [v2]

mul ; [D] ← [A] × [B] (D ⬄ A|B sur 16 bits)

staa v3H ; [v3H] ← [A]

stab v3L ; [v3L] ← [B]

.org $0020 ; Origine des données en mémoire (variables)

v1 .byte 10 ; Premier facteur

v2 .byte 40 ; Deuxième facteur

v3H .byte 0 ; Résultat poids fort

v3L .byte 0 ; Résultat poids faible

**Q4. a**) **Testez** le programme pas à pas et complétez le tableau :

| **Instruction** | **A (hexa)** | **B (hexa)** | **D (hexa)** | **Commentaire** |
| --- | --- | --- | --- | --- |
| ldaa v1 | 0A | \_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ | A = 10₁₀ |
| ldab v2 | \_\_\_\_\_\_ | 28 | \_\_\_\_\_\_\_\_\_\_ | B = 40₁₀ |
| mul | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ | 10₁₀ × 40₁₀ = \_\_\_\_\_ |
| staa v3H | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ | V3H = \_\_\_\_\_\_\_\_ |
| stab v3L | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ | V3L = \_\_\_\_\_\_\_\_ |

**b)** Après exécution :

* v3H contient : \_\_\_\_\_\_\_\_\_\_\_\_ (hexa)
* v3L contient : \_\_\_\_\_\_\_\_\_\_\_\_ (hexa)
* Le résultat complet v3 (16 bits) est : v3H|v3L = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (hexa)
* Convertissez le résultat en décimal : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* **Conclusion** : Le résultat est-il correct ? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**c**) **Synthèse** - code source et code machine correspondant

* **Complétez** le code machine ci-dessous **après** l’exécution du programme.

**Code machine Code source assembleur**

***Address opcode operand* ;label operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

\_\_0000\_\_ \_\_\_\_\_\_ \_\_\_\_\_\_\_\_ ldaa v1 ; [A] ← [v1]

\_\_\_\_\_\_\_\_ \_\_\_\_\_\_ \_\_\_\_\_\_\_\_ ldab v2 ; [B] ← [v2]

\_\_\_\_\_\_\_\_ \_\_\_\_\_\_ \_\_\_\_\_\_\_\_ mul ; [D] ← [A] × [B] (D ⬄ A|B sur 16 bits)

\_\_0007\_\_ \_\_\_\_\_\_ \_\_\_\_\_\_\_\_ staa v3H ; [v3H] ← [A]

\_\_\_\_\_\_\_\_ \_\_\_\_\_\_ \_\_\_\_\_\_\_\_ stab v3L ; [v3L] ← [B]

.org $0020 ; Origine des données en mémoire (variables)

0020 \_\_\_\_\_\_ v1 .byte 10 ; Premier facteur

\_\_\_\_\_\_\_\_ \_\_\_\_\_\_ v2 .byte 40 ; Deuxième facteur

\_\_\_\_\_\_\_\_ \_\_\_\_\_\_ v3H .byte 0 ; Résultat poids fort

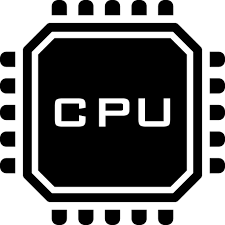
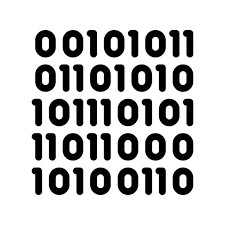
\_\_\_\_\_\_\_\_ \_\_\_\_\_\_ v3L .byte 0 ; Résultat poids faible

**💡 Points clés à retenir**

* Le **code source** (écrit par le programmeur) n’étant pas destiné à être exécuté par le processeur, un programme de **traduction** automatique (l’**assembleur**) est nécessaire.



**Assembleur**



**Code source**

**Code machine**

**Logiciel**

**Traducteur**

* Bien que plus facile à manipuler que les "codes machines", l'assembleur est **fastidieux à écrire,** car comme on le voit ci-dessus il faut aligner un grand nombre d'instructions pour obtenir un résultat, même simple. De plus, il **ne s'adresse qu'à un seul modèle de processeur**. Tout changement de machine nécessite une **réécriture** plus ou moins complète du **code**.
* Pour pallier ces défauts, des **langages évolués** comme le **C**, le **PHP** ou le **Python** ont été développés. Ils permettent au programmeur de se concentrer sur l'algorithmique des applications. Comme les instructions ne sont plus compréhensibles par l'ordinateur, une phase de traduction est nécessaire. C'est le rôle des **interpréteurs** et des **compilateurs**.

Aujourd'hui, l'assembleur reste utilisé pour écrire des parties des **systèmes d'exploitation**, **gestionnaires de périphériques,** etc.

**B. Instructions de rupture de séquence**

**ℹ️ Qu'est-ce qu'une rupture de séquence ?**

Par défaut, le processeur exécute les instructions **séquentiellement** (l'une après l'autre).  
Les **ruptures de séquence** permettent de :

* **Brancher** à une autre partie du programme (**saut**)
* **Répéter** des instructions (boucles)

**Date** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Classe** : \_\_\_\_\_\_\_

**Nom** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Prénom** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* **Choisir** entre plusieurs chemins (conditions)

**Types de ruptures :**

* **Branchement inconditionnel** : BRA (Branch Always)
* **Branchements conditionnels** : BEQ (si Z=1), BNE (si Z=0), etc.
* **Appels de sous-programmes** : JSR / RTS (partie D)

**B.1. Introduction aux sauts**

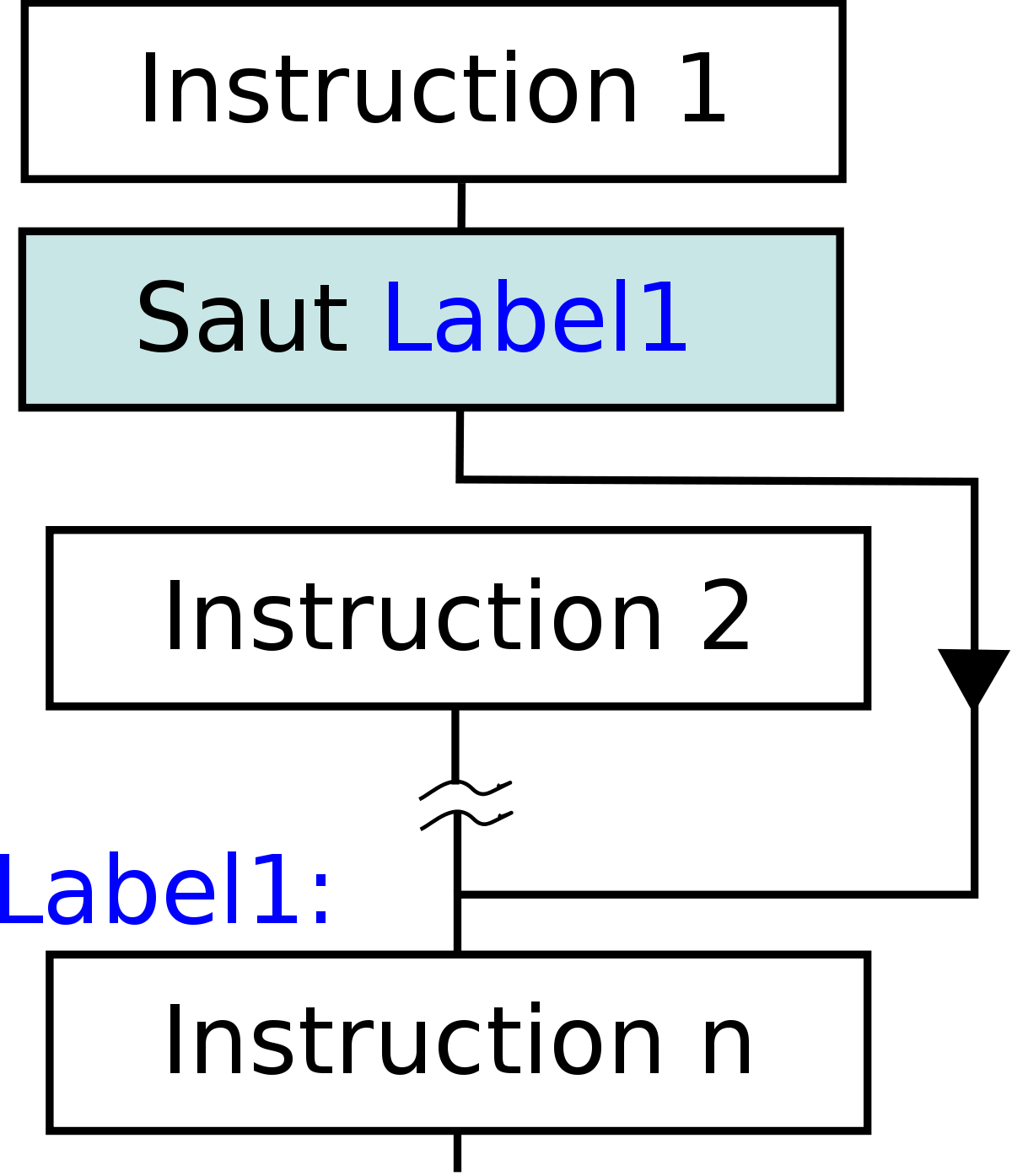
**🧪Exercice B.1.1 : Saut inconditionnel (code BRA)**

BRA

**🎯 Objectif** : Comprendre comment sauter une instruction

**ℹ️ BRA : BR**anch **A**lways

**Chargez** le fichier ***saut.asm*** situé dans **TP2\_SDK68xx**.



**Figure 5** : branchement inconditionnel

**; SDK6811 - saut.asm**

**; Branchement inconditionnel**

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldaa #5 ; [A] ← 5

bra *suite* ; Brancher à l'étiquette "suite"

ldaa #10 ; [A] ← 10 ⚠️ Cette ligne sera IGNORÉE

*suite* staa resultat ; [resultat] ← [A]

.org $0020 ; Origine des données en mémoire (variables)

resultat .byte 0



**Le registre Compteur Programme (PC)** ou compteur ordinal est un **registre** de **processeur** qui indique la position en mémoire de l'instruction en cours d'exécution dans le code binaire d'un **programme**.

**Q5**. **Exécutez** pas à pas et observez le registre **C**ompteur **P**rogramme (**PC**) :

| **Instruction** | **PC avant** | **PC après** | **A** | **Commentaire** |
| --- | --- | --- | --- | --- |
| ldaa #5 | 0000 | \_\_\_\_\_\_\_ | 05 | A = 5 |
| bra suite | \_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_ | 05 | PC saute ! |
| ldaa #10 | ////// | ////// | /// | Non exécutée |
| staa resultat | \_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_ | Suite du prog. |

**Conclusion** : L'instruction BRA provoque un branchement \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (conditionnel/inconditionnel).

**🧪 Exercice B.1.2 : Premier branchement conditionnel (BEQ)**

🎯 **Objectif** : Brancher seulement si une condition est vraie

**ℹ️BEQ** = **B**ranch if **EQ**ual to zero → Brancher si Z=1

**Chargez** le fichier **testzero.asm** situé dans **TP2\_SDK68xx.**

BEQ

**; SDK6811 - testzero.asm**

**; Branchement conditionnel : test zéro**

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldaa data ; [A] ← [data]

beq *estnul* ; **si** [A]=0 (Z=1) **alors** brancher à *estnul*

ldaa #1 ; **sinon** [A] ← 1 (data ≠ 0)

bra *suite* ; brancher à *suite*

*estnul* ldaa #0 ; [A] ← 0 (data = 0)

*suite* staa resultat ; [resultat] ← [A]

.org $0020 ; Origine des données en mémoire (variables)

data .byte 0 ; ⚠️ Exécutez le programme avec 0 puis avec 5

resultat .byte 0

**Test 1** : valeur = 0

**Q6**. **Complétez** le tableau

**Fin**

[A] ← [data]

[A] ← [A] + v2

[v3\_\_\_] ← [A]

[A] = 0 (Z = 1 )?

[A] ← 0

[resultat] ← [A]

[A] ← 1

**Début**

estnul

bra suite

suite

beq estnul

| **Instruction** | **A** | **Flag Z** | **Branchement ?** | **Chemin pris** |
| --- | --- | --- | --- | --- |
| ldaa data | 00 | 1 | ////////// | ////////// |
| beq estnul | \_\_\_\_\_ | \_\_\_\_\_ | OUI / NON | \_\_\_\_\_\_\_\_\_\_\_ |

Valeur finale dans resultat : \_\_\_\_\_\_\_\_\_\_

**Test 2** : Modifiez valeur = 5, réassemblez

**Q7. Complétez** le tableau

| **Instruction** | **A** | **Flag Z** | **Branchement ?** |
| --- | --- | --- | --- |
| ldaa data | 05 | \_\_\_\_\_ | ////////// |
| beq estnul | \_\_\_\_ | \_\_\_\_\_ | OUI / NON |

Valeur finale dans resultat : \_\_\_\_\_\_\_\_\_\_

**Conclusion** :

* BEQ branche si et seulement si \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ = 1
* Cela correspond à un résultat égal à \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**💡 Transition vers B.2**

Vous savez maintenant faire des sauts et des tests. Combinons-les pour créer des BOUCLES ! │

**B.2. Premières boucles simples**

**Fin**

[A] ← [A] + v2

[v3\_\_\_] ← [A]

[A] ≠ 0

(Z = 0) ?

**Début**

loop

bne loop

**🧪 Exercice B.2.1 : Compte à rebours**

**🎯 Objectif** : Répéter des instructions avec un compteur A compléter

**ℹ️BNE** = **B**ranch if **N**ot **E**qual to zero → Brancher si Z=0

**Chargez** le fichier ***compteur.asm*** situé dans **TP2\_SDK68xx**.

**; SDK6811 - compteur.asm**

**; Décompter**

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldaa #5 ; [A] ← 5

*loop* deca ; [A] ← [A] - 1

bne *loop* ; **si** [A]≠0 (Z=0) **alors** branchement à loop

staa cmpt ; **sinon** [cmpt] ← [A]

.org $0020 ; Origine des données en mémoire (variables)

cmpt .byte 0

Avant d'exécuter le programme, répondez :

* Combien de fois la boucle s'exécute-t-elle ? \_\_\_\_\_\_\_\_\_\_
* Quelle sera la valeur finale de cmpt ? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Q8.** **Exécutez** pour compléter le tableau d'exécution de la boucle et vérifiez vos prédictions.

| **Passage** | **A avant DECA** | **A après DECA** | **Flag Z** | **Branchement ?** | **Commentaire** |
| --- | --- | --- | --- | --- | --- |
| 1 | 05 | 04 | 0 | OUI | Continue |
| 2 | 04 | 03 | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_ |
| 3 | 03 | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_ |
| 4 | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_ |
| 5 | \_\_\_\_\_\_ | 00 | 1 | NON | Sort de la boucle |

**Conclusion** : Pour sortir de la boucle, il faut que Z = \_\_\_\_\_ (condition de BNE)

**B.3. Boucles avec adressage indexé**

**Date** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Classe** : \_\_\_\_\_\_\_

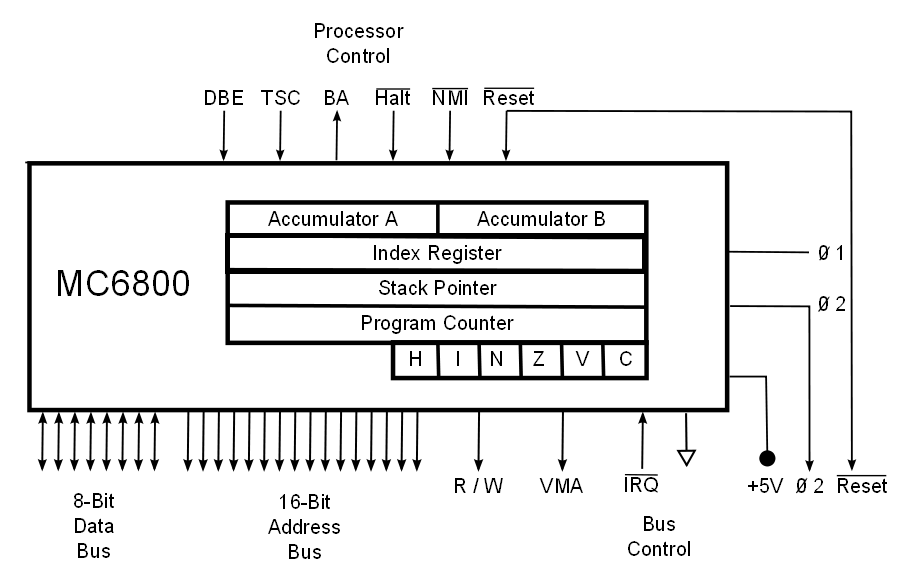
**Nom** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Prénom** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

🎯 Objectif : Parcourir des données en mémoire

🔧 Nouveauté : Registre X, mode d’adressage indexé

**B.3.1. Découverte du registre X**



**Figure 6** : registres du MC6800

🎯 **Objectif** : Comprendre le registre d'index et l'adressage indexé

**ℹ️ Le registre X (Index)**

Le registre **X** est un registre 16 bits qui sert de **pointeur** vers la mémoire. Il permet de parcourir des tableaux ou des chaînes de caractères.

**Instructions clés :**

* LDX #adresse : Charger une adresse dans X
* LDAA 0,X : Charger dans A la donnée pointée par X
* INX : Incrémenter X (pointer sur l'octet suivant)

**🧪 Exercice B.3.1.1 : Lire trois octets consécutifs**

**Chargez** le fichier ***parcours.asm*** situé dans **TP2\_SDK68xx.**

**; SDK6811 - parcours.asm**

**; Introduction aux pointeurs**

**; [Label] Operation [operand] [comment]**

.org $0000 ; Origine du programme en mémoire

ldx #data ; [X] ← data (pointe sur le premier octet)

ldaa 0,X ; [A] ← [0 + [X]] (Lire le premier octet)

inx ; [X] ← [X] + 1 (pointer sur l’octet suivant)

ldab 0,X ; Lire le deuxième octet

inx

ldaa 0,X ; Lire le troisième octet

.org $0020 ; Origine des données en mémoire (variables)

data .byte 10,20,30

**Q9.** Avant d'exécuter le programme, complétez le tableau :

| **Instruction** | **X (hexa)** | **A (hexa)** | **B (hexa)** | **Donnée pointée**  **en hexa (décimal)** | **Adresse de**  **la donnée pointée (hexa)** |
| --- | --- | --- | --- | --- | --- |
| ldx #data | 0020 | // | // | 0A (10) | 0020 |
| ldaa 0,X | 0020 | 0A | // | 0A (10) | 0020 |
| inx | 0021 | 0A | // | 14 (20) | 0021 |
| ldab 0,X | \_\_\_\_\_\_ | \_\_\_\_ | \_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ |
| inx | \_\_\_\_\_\_ | \_\_\_\_ | \_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ |
| ldaa 0,X | \_\_\_\_\_\_ | \_\_\_\_ | \_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_ |

**Exécutez** et vérifiez.

**Conclusion** :

* Le registre X contient une \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (donnée/adresse)
* L'instruction LDAA 0,X signifie : lire la donnée à l'adresse \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* INX permet de passer à l'octet \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (précédent/suivant)

🔍 **Comprendre l'adressage indexé**

**Mode étendu** (vu dans TP1 Découverte) :

ldaa $0020 ; Lire la valeur située à l'**adresse fixe** $0020 et la placer dans le registre A

**Mode indexé (nouveau)** :

ldx #$0020 ; X ← $0020

ldaa 0,X ; Lire la valeur située à l'adresse 0 + [X] soit $0020 et la placer dans le registre A

inx ; [X] ← [X] + 1

ldaa 0,X ; Lire la valeur située à l'adresse 0 + [X] soit $0021 et la placer dans le registre A

**Avantage** : L'**adresse** change **dynamiquement** → parfait pour les boucles !

**B.3.2. Chaîne de caractères**

**Date** : \_\_\_\_\_\_\_\_\_\_\_\_\_ **Classe** : \_\_\_\_\_\_

**Nom** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Prénom** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

🎯 Objectif : Détecter la fin d'une chaîne de caractères

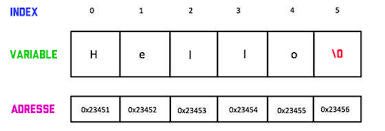
**ℹ️ Qu'est-ce qu'une chaîne de caractères ?**

En informatique, une **chaîne de caractères** est une suite ordonnée de caractères.

En assembleur 6800, la fin d'une chaîne est identifiée par le caractère **ASCII NULL (0x00)**.

**Exemple : Adresse Valeur Caractère**

$001D 48 'H'



**Figure 7** : chaîne dans la mémoire

$001E 65 'e'

$001F 6C 'l'

$0020 6C 'l'

$0021 6F 'o'

$0022 00 NULL (caractère de fin de chaîne)

La directive **.str** ajoute automatiquement le caractère NULL :

Exemple : msg .str "Hello" ; Assemblé en 48 65 6C 6C 6F 00

**🎯 Stratégie : Parcours avec test de fin (version avec compteur fixe)**

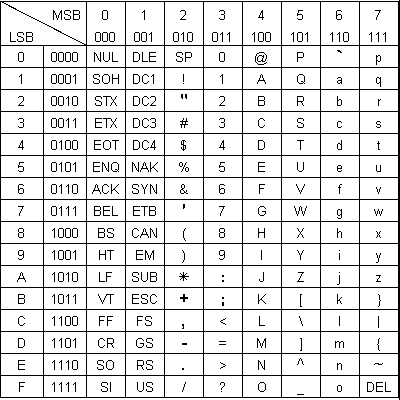
**Étape 1 : Parcours de 5 caractères exactement**

Chargez le fichier **parcoursfixe.asm** situé dans **TP2\_SDK68xx.**

.org $0000 ; origine du programme en mémoire

ldx #msg ; X pointe sur le premier caractère

ldab #5 ; Compteur fixe : 5 caractères



**Figure 8** : Table des codes de caractères ASCII

*loop* ldaa 0,X ; Lire le caractère

; (Ici on pourrait l'afficher)

inx ; Passer au caractère suivant

decb ; Décrémenter compteur

bne *loop* ; Continuer si B≠0

.org $0020 ; origine de données en mémoire (variables)

msg .str "Hello"

**Q10.** Testez et complétez le tableau :

| **Passage** | **X (hexa)** | **A (hexa)** | **B (hexa)** | **Caractère** |
| --- | --- | --- | --- | --- |
| 1 | 0020 | 48 | 5 | 'H' |
| 2 | 0021 | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ |
| 3 | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ |
| 4 | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ |
| 5 | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ | \_\_\_\_\_ |

**Problème** : Cette méthode nécessite de connaître la longueur de la chaîne de caractères.  
**Solution** : Détecter le caractère NUL présent dans la chaîne !

**🎯 Stratégie finale : Détection du NUL**

**b) Complétez les instructions assembleur :**

\_\_\_\_\_\_\_\_\_\_  ; X ← adresse de msg

loop \_\_\_\_\_\_\_\_\_\_  ; A ← [0+[X]]

cmpa #0 ; (A = 0 (Z=1)?)

\_\_\_\_\_\_\_\_\_\_\_;si oui alors branchement à suite

\_\_\_\_\_\_\_\_\_\_\_; sinon X ← X + 1

\_\_\_\_\_\_\_\_\_\_\_; branchement à loop

suite ; suite du programme

**c)** **Chargez et complétez** le fichier ***finchaine.asm*** situé dans **TP2\_SDK68xx.**

**d) Modifier** le programme pour que "Hello" s’affiche dans l’onglet  du simulateur.



**Étape 2 : Amélioration du code précédent, parcours jusqu'à NUL**

**Q11. a) Terminez** la **boucle** dans l'algorigramme ci-dessous*.*

Appel prof

Suite prog

Début Finchaîne

[A] ← [0 + [X]]

[A]=0

(Z=1)?

[X] ← [X] + 1

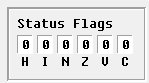
cmpa #0

loop

bra loop

beq suite

[X] ← msg



Registre d’état des microprocesseurs

6800/6811

**Annexe 1 - Registre d’état**

**Source : Wikipédia** [urlr.me/XzhK2T]

Le **registre d'état**, ou **registre de drapeaux** est un ensemble de [bits](https://fr.wikipedia.org/wiki/Bit) représentant des [drapeaux](https://fr.wikipedia.org/wiki/Drapeau_(informatique)) au sein d'un [processeur](https://fr.wikipedia.org/wiki/Processeur). Le registre [RFLAGS](https://fr.wikipedia.org/wiki/RFLAGS) est un exemple de registre d'état propre à l'architecture de processeurs [x64](https://fr.wikipedia.org/wiki/X64).

Les **bits** composant le registre d'état sont **indépendants les uns des autres**, et la valeur de chacun **apporte une information** supplémentaire quant **au résultat d'une opération antérieure**. En effet, au cours d'un calcul, le processeur va automatiquement mettre à jour le registre d'état, en plus de fournir le résultat de l'opération. Le registre d'état comporte en général un **minimum de quatre drapeaux**, que sont les indicateurs de nullité (**résultat égal à zéro**), de retenue (**l'opération a produit une retenue**), de dépassement de capacité (**le signe du résultat diffère du signe des opérandes**), ainsi que de négativité (**le résultat est inférieur à zéro**).

Ces drapeaux peuvent ensuite être **utilisés**, notamment pour déterminer si une **opération conditionnelle** doit être exécutée ou non. Une utilisation fréquente de ce registre consiste à déterminer si un [branchement](https://fr.wikipedia.org/wiki/Branchement) (saut vers une portion spécifique du code) doit être effectué. Pour cela, on effectue tout d'abord une comparaison entre deux valeurs, qui consiste dans les faits à réaliser une soustraction entre les deux valeurs, opération qui met à jour le registre d'état. Ensuite, il suffit par exemple de tester la valeur du registre indiquant un résultat négatif pour savoir laquelle des deux valeurs était la plus grande, et en fonction de cette valeur, réaliser ou non le branchement.

**Drapeaux les plus communs**

Les drapeaux ci-dessous sont présents dans la plupart des processeurs actuels.

|  |  |  |
| --- | --- | --- |
| **Drapeau** | **Nom** | **Description** |
| **Z** | [**Z**éro](https://fr.wikipedia.org/wiki/Indicateur_de_z%C3%A9ro) | Indique que le résultat d'une opération est nul. |
| **C** | [Retenue](https://fr.wikipedia.org/wiki/Indicateur_de_retenue) (***C****arry*) | Le résultat de l'opération est incomplet, car une retenue a été produite. Ce bit peut être utilisé pour réaliser des calculs sur des opérandes plus grands que la taille du processeur, en séparant les valeurs. Par exemple, un processeur 32 bits pourra additionner des mots de 64 bits en les séparant en deux mots de 32 bits, additionnés indépendamment, et en utilisant la retenue pour faire le lien entre les deux. |
| **N** / **S** | [Signe](https://fr.wikipedia.org/wiki/Indicateur_de_signe) (**N**egative ou **S**ign) | Indique que le résultat de l'opération est inférieur à zéro. |
| **V** / **O** | [Dépassement de capacité](https://fr.wikipedia.org/wiki/Indicateur_de_d%C3%A9bordement) (***OV****erflow*) | Le signe du résultat diffère du signe des opérandes, ce qui indique que la valeur a débordé sur le bit de signe, et donc que la taille du processeur est trop petite pour stocker le résultat. |

## **Annexe 2 – Description des instructions utilisées dans les programmes**

Le tableau ci-dessous est extrait de la source : <https://webge.fr/6800.html> . Il permet de connaître l’organisation d’une instruction, son rôle et sa syntaxe en assembleur **MC6800 / 6811**.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Opcode**  (Base 16) | **Nombre d’octets (Opcode + Opérande(s))** | **Syntaxe assembleur de l’instruction** | **Opération symbolique** | **Description** | **Remarque** |
| **27** | 2 | **BEQ** disp | (Z == 1) ? {[[PC](http://www.8bit-era.cz/6800.html#PC-reg)] [←](http://www.8bit-era.cz/6800.html#transfer-desc) [[PC](http://www.8bit-era.cz/6800.html#PC-reg)] + [disp](http://www.8bit-era.cz/6800.html" \l "disp-desc) + 2} | **Si** l’instruction précédente a produit un résultat nul **alors** PC= PC + disp + 2 **sinon** PC=PC+2.  **Attention**, disp est codé en complément à 2. | **B**ranch if **E**qual to zero |
| **20** | 2 | **BRA** disp | [[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 | Saut inconditionnel à la position PC + disp + 2.  **Attention**, disp est codé en complément à 2. | **BR**anch **A**lways |
| **81** | 2 | **CMPA** #data8 | [A] – data8 | Compare le contenu de l’accumulateur avec data8 | **C**o**MP**are **A** with data8 |
| **08** | 1 | **INX** | [X] ← [X] + 1 | Incrémente le registre d’index X. | **IN**crement **X** |
| **B6** | 3 | **LDA**[**A**](http://www.8bit-era.cz/6800.html#A-reg) addr16 | [A] ← [addr16] | Charge l’accumulateur A avec la donnée sur 8 bits (opérande) située à la position addr16. | **L**oad **A**ccumulateur **A** from memory |
| **F6** | 3 | **LDA**[**B**](http://www.8bit-era.cz/6800.html#A-reg) addr16 | [B] ← [addr16] | Charge l’accumulateur B avec la donnée sur 8 bits (opérande) située à la position addr16. | **L**oad **A**ccumulateur B from memory |
| **CE** | 3 | **LDX** #addr16 | [[X](https://webge.fr/6800.html#X-reg)[(HI)](https://webge.fr/6800.html#HI-desc)] [←](https://webge.fr/6800.html#transfer-desc) data16[(HI)](https://webge.fr/6800.html#HI-desc), [[X](https://webge.fr/6800.html#X-reg)[(LO)](https://webge.fr/6800.html#LO-desc)] [←](https://webge.fr/6800.html#transfer-desc) data16[(LO)](https://webge.fr/6800.html#LO-desc) | Charge le registre d’index X avec la donnée sur 16bits (opérande) data16. | **L**oad**D** the index register **X** |
| **01** | 1 | **NOP** |  |  | **N**o **O**peration |
| **B7** | 3 | **STAA** addr16 | [addr16] [←](http://www.8bit-era.cz/6800.html#transfer-desc) [A] | Sauvegarde le contenu de A à l’adresse addr16. | **ST**ore **A**ccumulator **A** in Memory |
| **F7** | 3 | **STAB** addr16 | [addr16] [←](http://www.8bit-era.cz/6800.html#transfer-desc) [B] | Sauvegarde le contenu de B à l’adresse addr16. | **ST**ore **A**ccumulator **B** in Memory |

**Seulement MC6811 (**[6811 instructions set](https://www.cs.uaf.edu/2007/fall/cs441/proj1notes/sawyer/inst.html)**)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Opcode**  (Base 16) | **Nombre d’octets (Opcode + Opérande(s))** | **Syntaxe assembleur de l’instruction** | **Opération symbolique** | **Description** | **Remarque** |
| **3D** | 1 | **MUL** | [D] ← [A] \* [B] | D est constitué de A et B tel que  D(HI) = A et D(LO)=B. | **MUL**tiplication **A** with **B** |

**←** : **affectation** (la donnée ou l’adresse est transférée dans la direction de la flèche). **[...]** : contenu de ...

**$**: la valeur qui suit l’**opcode** est en hexadécimal. **disp** : déplacement d’adresse **signé sur 8 bits**.

**#** : la valeur qui suit l’**opcode** est une donnée (en l’absence de # c’est une adresse). **addr16** : adresse codée sur 16bits.

**data8** : donnée codée sur 8bits.