Lycée Polyvalent PIERRE EMILE MARTIN	**Langage machine et assembleur**	académie d'Orléans-Tours
TP2 **Approfondissement**	**Programmation en Assembleur 6800** Opérations arithmétiques, transferts en mémoire, tests et boucles	

🎯 **Objectifs du TP**
- 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

Date : _____ **Classe** : _____

Nom : _____

Prénom : _____

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

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.

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

On effectue l'opération **sum ← 200_{10} + 100_{10}** dans le programme ci-dessous,

Q1. a) Avant d'exécuter le programme, prédisez :
- Donnez le résultat de 200_{10} + 100_{10} 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 SDK-6800 et chargez le fichier ***depassement.asm*** situé dans **TP2_SDK68xx**.

```
; SDK6800 - depassement.asm
; 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
```

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] <- 200_{10}

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

b) Relevez le contenu de la mémoire (**code machine** ci-dessous et identifiez 100_{10} et 200_{10} 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 :

- 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

_____

_____

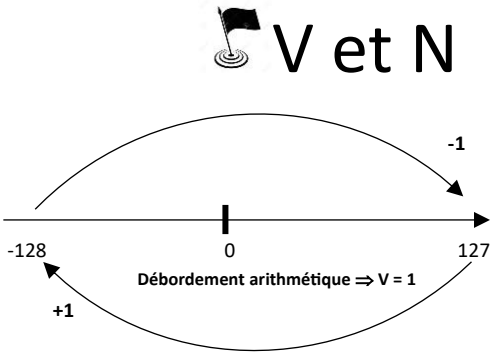
Remarque : Quand **C=1**, cela signifie qu'il y a eu un dépassement (retenue) voir l'**annexe 1**.

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)

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	_____	_____	_____	_____	_____

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

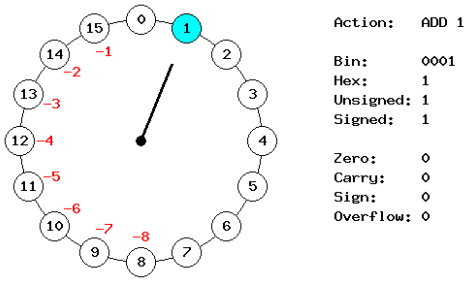


***Figure 1** : animation (Sign = N, Overflow = V)*

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]
          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
```

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)**

c) Testez avec v1=-128 (soit 80_{16}) et v2=-1 (soit FF_{16})

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 :

```
; [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
```

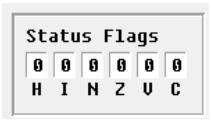


Figure 2 : registre d'état des microprocesseurs 6800/6811

Q3 a) Testez avec v1=-1 (soit FF_{16}) et v2=1

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 80_{16}) ou supérieur au maximum positif (ici $7F_{16}$).
- **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 00_{16}) ou supérieur au maximum (FF_{16}).

A.3. Multiplication en machine

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

Spécifications :

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

```
A       0110   (6)
B       0111   (7)
       01110
      01110
     0110
   0000
  00101010  (42)
```

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

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)

Après MUL : D = A × B (16 bits)
A contient la partie haute (v3H)
B contient la partie basse (v3L)

Remarque : On utilise le simulateur en mode 6811 [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

Exercice A.3.1 : Test avec dépassement 8 bits

Sélectionnez 6811 et chargez le fichier ***mul.asm*** situé dans **TP2_SDK68xx**..

```
; SDK6811 - mul.asm
; opération : v3 ← v1 x v2
; [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
```

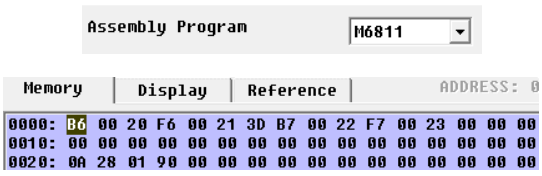


***Figure 4**: code machine de mul.asm (non exécuté)*

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

Instruction	A (hexa)	B (hexa)	D (hexa)	Commentaire
ldaa v1	0A	_____	_____	A = 10_{10}
ldab v2	_____	28	_____	B = 40_{10}
mul	_____	_____	_____	$10_{10} \times 40_{10}$ = _____
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 ? _____

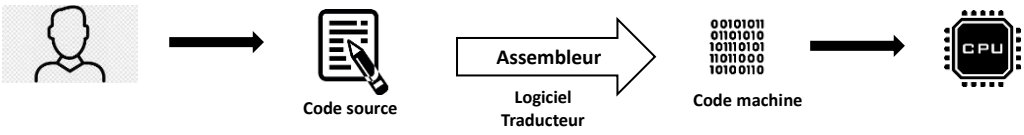
c) **Synthèse** - code source et code machine correspondant

- **Complétez** le code machine ci-dessous **<u>après</u>** 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.



- 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

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)
- **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)

Date : _____ **Classe** : _____

Nom : _____

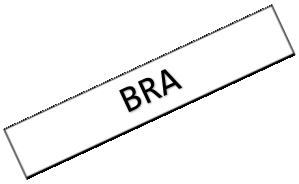
Prénom : _____

B.1. Introduction aux sauts

Exercice B.1.1 : Saut inconditionnel (code BRA)

Objectif : Comprendre comment sauter une instruction

BRA : BRanch **A**lways



Chargez le fichier ***saut.asm*** situé dans **TP2_SDK68xx**.

```
; 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
```

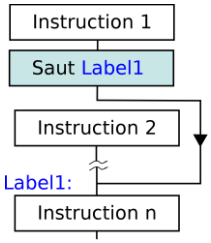


***Figure 5** : branchement inconditionnel*

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.

PC 0000

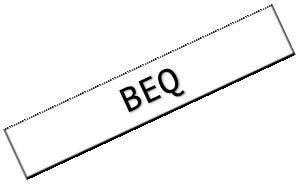
Le registre Compteur Programme (PC) ou compteur ordinal est un **registre** de **processeur** qui indique la position en mémoire de l'instruction en cours d'exécution dans le code binaire d'un **programme**.

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

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



Chargez le fichier **testzero.asm** situé dans **TP2_SDK68xx.**

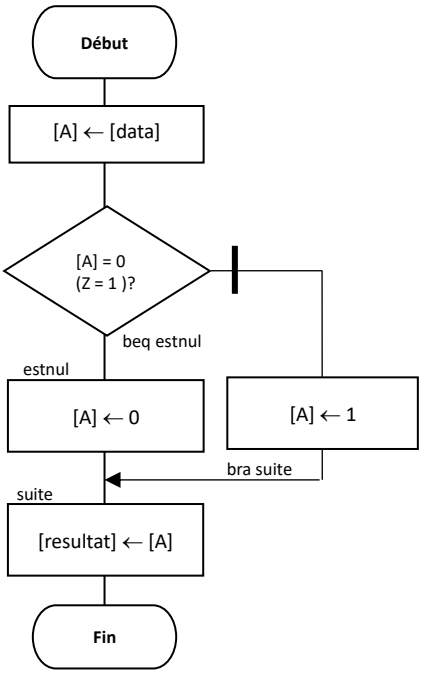
```
; SDK6811 - testzero.asm
; Branchement conditionnel : test zéro
; [Label] Operation [operand]   [comment]
        .org $0000      ; Origine du programme en mémoire
        ldaa data       ; [A] ← [data]
        beq estnul      ; si [A]=0 (Z=1) alors brancher à estnul
        ldaa #1         ; sinon [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

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

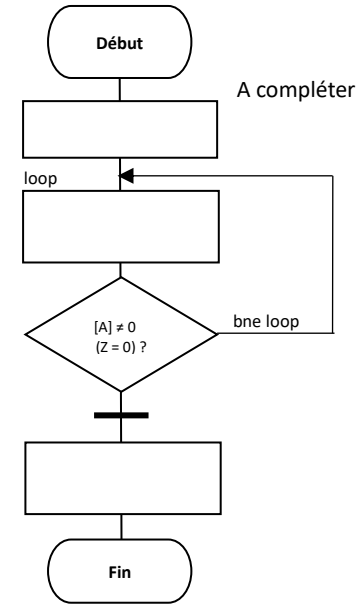
Valeur finale dans resultat : _____

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



Exercice B.2.1 : Compte à rebours

Objectif : Répéter des instructions avec un compteur

BNE = **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)

Date : _____ **Classe** : _____

Nom : _____

Prénom : _____

B.3. Boucles avec adressage indexé

🎯 Objectif : Parcourir des données en mémoire

🔧 Nouveauté : Registre X, mode d'adressage indexé

B.3.1. Découverte du registre X

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

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)

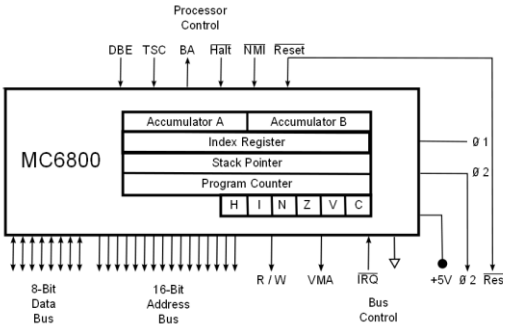


Figure 6 : registres du MC6800

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 ; *[A] ← [0020] : Affecter le contenu de la mémoire située à l'adresse fixe 0020_{16} au registre A*

Mode indexé (nouveau) :

ldx #$0020 ; [X] ← 0020_{16} : *Affecter la valeur 0020_{16} au registre X*
ldaa 0,X ; [A] ← [0+[X]] *: Affecter le contenu de la mémoire située à l'adresse 0 + [X] = 0020_{16} au registre A*
inx ; [X] ← [X] + 1
ldaa 0,X ; *[A] ← [0+[X]] : Affecter le contenu de la mémoire située à l'adresse 0 + [X] = 0021_{16} au 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

i 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'
$001E	65	'e'
$001F	6C	'l'
$0020	6C	'l'
$0021	6F	'o'
$0022	00	NULL (caractère de fin de chaîne)

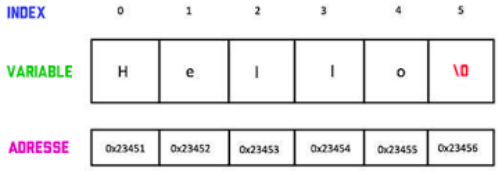


Figure 7 : chaîne dans la mémoire

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
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"
```

MSB / LSB	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 0000	NUL	DLE	SP	0	@	P	`	p
1 0001	SOH	DC1	!	1	A	Q	a	q
2 0010	STX	DC2	"	2	B	R	b	r
3 0011	ETX	DC3	#	3	C	S	c	s
4 0100	EOT	DC4	$	4	D	T	d	t
5 0101	ENQ	NAK	%	5	E	U	e	u
6 0110	ACK	SYN	&	6	F	V	f	v
7 0111	BEL	ETB	'	7	G	W	g	w
8 1000	BS	CAN	(	8	H	X	h	x
9 1001	HT	EM	)	9	I	Y	i	y
A 1010	LF	SUB	*	:	J	Z	j	z
B 1011	VT	ESC	+	;	K	[	k	{
C 1100	FF	FS	,	<	L	\	l	|
D 1101	CR	GS	-	=	M	]	m	}
E 1110	SO	RS	.	>	N	^	n	~
F 1111	SI	US	/	?	O	_	o	DEL

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

Q10. Testez et complétez le tableau :

Passage	X (hexa)	A (hexa)	B (hexa)	Caractère
1	0020	48	5	'H'
2	0021	_____	_____	_____
3	_____	_____	_____	_____
4	_____	_____	_____	_____
5	_____	_____	_____	_____

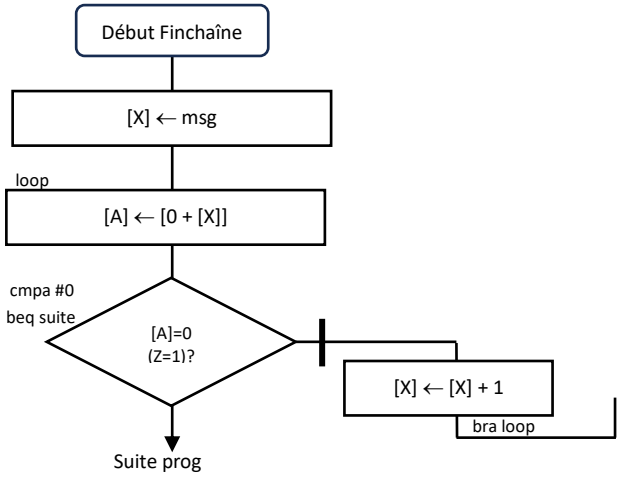
Problème : Cette méthode nécessite de connaître la longueur de la chaîne de caractères.

Solution : Détecter le caractère NUL présent dans la chaîne !

🎯 **Stratégie finale : Détection du NUL**

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

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



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

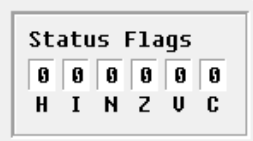
c) Chargez et complétez le fichier ***finchaine.asm*** situé dans **TP2_SDK68xx.**

d) Modifier le programme pour que "Hello" s'affiche dans l'onglet `Display` du simulateur.

```
Display  Buffer @ 0xFB00
comprises 54 x 20 chars
```

Appel prof

Annexe 1 - Registre d'état



Registre d'état des microprocesseurs 6800/6811

Source : Wikipédia [urlr.me/XzhK2T]

Le **registre d'état**, ou **registre de drapeaux** est un ensemble de bits représentant des drapeaux au sein d'un processeur. Le registre RFLAGS est un exemple de registre d'état propre à l'architecture de processeurs 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 (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	Zéro	Indique que le résultat d'une opération est nul.
C	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 (**N**egative ou **S**ign)	Indique que le résultat de l'opération est inférieur à zéro.
V / O	Dépassement de capacité (***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] ← [PC] + disp + 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] ← [PC] + disp + 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	**LDAA** 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	**LDAB** 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(HI)] ← data16(HI), [X(LO)] ← data16(LO)	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] ← [A]	Sauvegarde le contenu de A à l'adresse addr16.	**ST**ore **A**ccumulator **A** in Memory
F7	3	**STAB** addr16	[addr16] ← [B]	Sauvegarde le contenu de B à l'adresse addr16.	**ST**ore **A**ccumulator **B** in Memory

Seulement MC6811 (6811 instructions set**)**

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).
$: la valeur qui suit l'**opcode** est en hexadécimal.
: la valeur qui suit l'**opcode** est une donnée (en l'absence de # c'est une adresse).

[...] : contenu de ...
disp : déplacement d'adresse **signé sur 8 bits**.
addr16 : adresse codée sur 16bits.
data8 : donnée codée sur 8bits.