

## Présentation

a2i12 à l'IUT de Troyes représente 2 TDs d'une heure et demie au début du 2° semestre + 1TP d'une heure et demie sur tous le semestre.

<b>Semestre 2</b>	
Bimestre 3	Bimestre 4
TD a2i12 C	
TDa2i12 68HC11	
TP a2i12 68HC11 assembleur	TP a2i12 68HC11 langage C
	TP a2i12 langage C sous LINUX

Tous les modules ci-dessus sont d'une heure et demie par semaine.

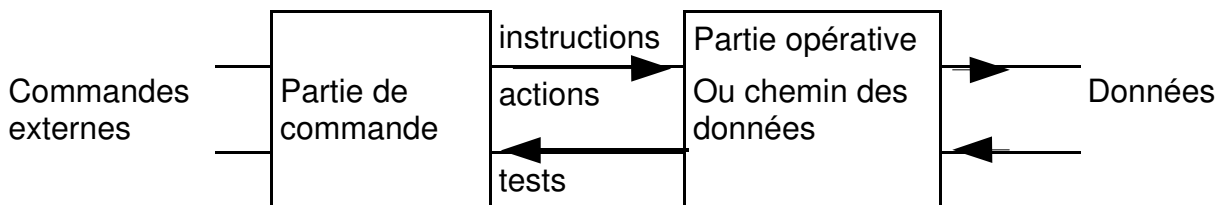
Un des deux TD est dédié à l'apprentissage de la programmation en C tandis que l'autre est consacré au 68HC11 et à son assembleur. Il est probable que dans les années à venir l'on remplace peu à peu la partie assembleur par du C. Ainsi à terme tout a2i12 sera consacré au C, un TD avec le 68HC11 et un TD avec LINUX.

Pour tout renseignement complémentaire envoyer un mél à [s.moutou@iut-troyes.univ-reims.fr](mailto:s.moutou@iut-troyes.univ-reims.fr)

## A2I12 TD n°1

### I) Algorithme

Nous définissons un algorithme en utilisant une division entre partie de commande et partie opérative (division pas toujours facile à faire).



La partie opérative (chemin des données) est caractérisée par un ensemble d' états (ensemble discret ou continu), par exemple un processeur et sa mémoire (ensemble discret d' états) ou un robot mobile (ensemble continu d' états).

On appelle action ou pour nous instruction toute fonction permettant de faire passer la partie opérative d' un état à l' autre. Un algorithme est une suite (en général finie) d' instructions. Il permet donc de faire passer la partie opérative d' un état initial vers un état final. Le déroulement de cet algorithme consomme des données en entrées et produit des sorties.

### II) Exemples pour le C

1°) On prend comme partie opérative un écran d' ordinateur. Il peut recevoir 25 lignes de 80 caractères. Si on n' utilise que les caractères de l' alphabet ' a' ,... ' z' , ' A' ,..., ' Z' ainsi que le caractère espace, combien d' états possède-t-il ?

Réponse :  $(2 \times 26 + 1) = 53$ ,  $25 \times 80 = 2\,000 \Rightarrow N = 53^{2000}$

C' est une caractéristique de l' informatique moderne de présenter un nombre d' états important.

Les instructions classiques pour passer d' un état à un autre sont putchar et printf. Si on veut les détailler il faut encore ajouter des états pour les  $25 \times 80$  états (à détailler un peu en TD : une chaîne peut dépasser l' écran...)

**Remarque** : en général la connaissance du nombre d' états n' a aucun intérêt. Ce nombre est calculé ici uniquement pour son ordre de grandeur.

2°) On prend comme partie opérative un ensemble de trois cases mémoires de 8 bits. On appelle cela des variables. Comme elles n' ont que huit bits, elles sont dites de type char.

#### Exercice 1

Combien d' états possède l' ensemble des trois variables ci-dessus ?

Les instructions d' un langage de programmation sont abstraites dans le sens où elles permettent de passer d' un ensemble d' états (et non pas d' un état) à un autre. Par exemple si vous décidez de faire une addition de deux cases mémoires pour mettre le résultat dans une troisième case en notant par exemple `c<-a+b` vous serez toujours capable de trouver l' état futur connaissant l' état présent.

### 1°) Syntaxe et présentation du C

Soit le petit programme :

```
#include <stdio.h>          /* directives ici... */
/* ceci est un commentaire */
main() {                   /* programme principal commence ici */
    printf("bonjour\n");
    printf("à vous ! ");
    return 0; // commentaire C++ (valable en C) dit fin de ligne
}
```

## 2°) Les variables

Une variable est caractérisée par quatre choses :

- un nom : commence forcément par un caractère puis caractère ou chiffre. C fait la distinction entre minuscule et majuscule : mavariabLe et MaVariable sont 2 variables distinctes.
- un type : il définit le genre de valeurs que contiendra la variable par exemple : int, char, unsigned char, float, double et des types construits qui seront vus plus tard.
- une valeur qui est forcément du type défini plus haut.
- une adresse à laquelle elle range sa valeur.



L' utilisation d' une variable nécessite sa déclaration sous la forme : type nom\_variable.  
 Cette déclaration doit être vue comme un contrat entre le programmeur et le compilateur. Comme tout contrat, il devra être respecté sinon le programmeur s' expose à une série de messages d' erreurs.

Exemple : int a,b,c; déclare 3 variables de type entières.



**Une déclaration est un contrat !!!**  
 Vous venez de dire promis juré que dans les variables nommées a,b,c vous ne mettrez que des entiers et que par conséquent jamais vous n' écrirez a=1.2; Ce sont les termes de votre contrat que vous devez respecter !!!



**&a désigne l'adresse de la variable a.** Cette adresse seul le compilateur a besoin de la connaître mais pas vous. Par contre vous aurez parfois besoin de la désigner avec donc l' opérateur &

Mettre une valeur dans une variable est ce que l' on appelle une affectation. Il existe plusieurs types d' affectation :

- directe avec le signe =, par exemple a=5; met 5 dans a,
- au clavier avec l' instruction scanf ou autres. L' utilisation de scanf nécessite quelques explications que nous allons donner maintenant. Elle s' écrit :  
scanf(format,adresse,...)
- format peut prendre les valeurs : "%d" entier décimal, "%u" entier décimal non signé, "%f" flottant, "%c" caractère, "%s" chaîne de caractères. Adresse est l' adresse de la variable obtenue par exemple avec &.
- à l' aide des opérateurs +=, -=, \*=, /=, %= . Ces opérateurs seront utilisés plus loin.

## Exemple d'entrées sorties

Les deux programmes suivants sont équivalents si l' on rentre 1 2 3RC ou 1RC2RC3RC (RC désigne retour chariot).

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int a,b,c;
    system("clear"); /*efface
ecran*/
    printf("Entrez trois
valeurs\n");
    scanf("%d%d%d",&a,&b,&c);
    printf("vos valeurs sont :
%d,%d,%d",a,b,c);
    return 0;
}

#include <stdio.h>
main() {
    int a,b,c;
    printf("Entrez trois
valeurs\n");
    scanf("%d",&a);
    scanf("%d",&b);
    scanf("%d",&c);
    printf("vos valeurs sont :
%d,%d,%d",a,b,c);
    return 0;
}
```

### **3°) Les expressions**

Les expressions arithmétiques sont construites à l' aide des opérateurs ' +' , ' -' , ' \*' , ' /' . On dispose aussi de l' opérateur modulo ' %' .

#### **Exercice 2**

On rappelle que le code ASCII de x est 120, celui de ' 0' est 48.

On a la déclaration suivante : int i,j;

Que trouvera-t-on dans i avec les affectations :

```
i = ' x' ;
i = ' 0' ;
i = 3.3;
i = (' x' -' 0' )/3;
i = (' x' -' 0' )%3;
```

Le symbole « / » désigne la division. Celle-ci sera entière (c' est à dire sans reste) si elle est entourée par deux entiers. L' associativité de « \* » et « / » est gauche vers la droite, ce qui signifie que l' on réalise ces opérations en commençant par la gauche. Que trouvera-t-on dans i avec les affectations:

```
i = 2 * j / 2; si la valeur dans j est 5
i = 2 * (j / 2); si la valeur dans j est 5
```

Il existe en C un opérateur d' incrémentation : ++ et un opérateur de décrémentation : --

#### **Exercice 3**

Given the program :

```
#include <stdio.h> // pour
printf()
main() {
    int i=1;
    printf("i = %d\n", i);
    printf("i = %d\n", ++i);
    printf("i = %d\n", i++);
    printf("i = %d\n", i);
    printf("i = %d\n", i--);
    printf("i = %d\n", i);
    printf("i = %d\n", --i);
    printf("i = %d\n", i);
    return 0;
}
```

What can we see on the screen when we launch this program ?

### Exercice 4

Écrire un programme qui demande deux nombres à l' utilisateur et affiche le résultat de l' addition de ces deux nombres.

### III) Exemples pour le 68HC11

#### 1°) La représentation des nombres

La donnée de base de 68HC11 est l'octet. l'octet à 8 bits. Le bit 0 est le bit de poids faible et le bit 7

est le bit de poids fort.

De plus, le 68HC11 connaît des mots de 16 bits. Le bit 0 est aussi le bit de poids faible et le bit 15 est

le bit de poids fort. Dans la mémoire l'octet de poids fort se trouve devant l'octet de poids faible.

L'unité centrale connaît trois présentations des nombres:

**Nombres entiers non signés** (Unsigned Integer): Un octet peut avoir des valeurs entre 0 et 255 (\$FF), un mot entre 0 et 65535 (\$FFFF).

**Nombres entiers signés complément à 2** (Two's complement). Un octet peut avoir des valeurs entre -128 (\$80) et +127 (\$7F), un mot entre -32768 (\$8000) et 32767 (\$7FFF). Le bit de poids fort indique toujours le signe.

**Décimal codé binaire** (BCD Binary Coded Decimal): Un octet contient deux chiffres décimaux. Les bits 7,6,5,4 contiennent le chiffre de poids fort, les bits 3,2,1,0 contiennent le chiffre de poids faible. Un octet peut donc avoir des valeurs entre 0 (\$00) et 99 (\$99).

Cette présentation est peu utilisée. Des **adresses** ont 16 bits et adressent des octets.

L'espace d'adressage de l'unité centrale comprend donc  $2^{16} = 65536$  octets

(\$0000..\$FFFF). On trouve toutes les mémoires comme la RAM, la ROM, l'EEPROM, et les registres d'E/S dans l'espace d'adressage.

### Exercice 1

Convert the following decimal numbers to binary : +11, -11, -23 (two' s complement representation with 8 bits). Then convert these numbers to hexadecimal.

Convert the following binary numbers to decimal : 111011, 11010101.

Perform the following binary subtraction using (i) the « ordinary » subtraction technique ; (ii) the two' s complement method.

11011 - 10101

11100 - 1001

**2°) Le modèle de programmation du 68HC11**

Il peut être présenté de la manière suivante :

7	Accumulateur A	0	7	Accumulateur B	0
15	Accumulateur D			0	

15	Index X	0
----	---------	---

15	Index Y	0
----	---------	---

15	Pointeur de pile SP	0
----	------------------------	---

15	Compteur ordinal PC	0
----	------------------------	---

S	X	H	I	N	Z	V	C
---	---	---	---	---	---	---	---

S : interdiction du mode STOP    N : négatif  
 X : masque d' interruptions /XIRQ    Z : zéro  
 H : demie-retendue (Half carry)    V : débordement (overflow)  
 I : masque d' interruption    C : retenue (Carry)

**2°) Quelques instructions**

Expliquer assemblage et desassemblage et parler au moins de l' adressage immédiat et inhérent.

**Les modes d' adressage**

Le 68HC11 connaît cinq modes d' adressage. Nous n' en verrons qu' un dans ce TD.

**Immédiat (Immediate) :** C' est l' adressage le plus facile. L' opérande se trouve directement dans le programme derrière le code de l' instruction. Au niveau assemblage on exprime ce mode d' adressage avec le symbole dièse (#).

```

86 7F      ldaa #127      charger 127 dans ACCA
8B 10      adda #$10     additionner 16
CE 10 00   ldx #$1000    charger la valeur $1000 dans le
registre IX
    
```

Mnemoni c	Descriptio n	Adressin g Mode	Instructions			Conditions Code							
			Opcode	Operand	cycles	S	X	H	I	N	Z	V	C
ABA	A+B->A	INH	1B	---	2	-	-	X	-	X	X	X	X
ABX	IX+B->IX	INH	3A	---	3	-	-	-	-	-	-	-	-

<b>Mnemoni c</b>	<b>Descriptio n</b>	<b>Adressin g</b>	<b>Instructions</b>			<b>Conditions Code</b>								
ABY	IY+B->IY	INH	183A	---	4	-	-	-	-	-	-	-	-	-
ADCA(opr )	A+M+C->A	IMM	89	ii	2	-	-	X	-	X	X	X	X	
		DIR	99	dd	3									
		EXT	B9	hh ll	4									
		IND,X	A9	ff	4									
		IND,Y	18A9	ff	5	-	-	X	-	X	X	X	X	
ADCB(opr )	A+M+C->B	IMM	C9	ii	2	-	-	X	-	X	X	X	X	
		DIR	D9	dd	3									
		EXT	F9	hh ll	4									
		IND,X	E9	hf	4									
		IND,Y	18E9	ff	5	-	-	X	-	X	X	X	X	
ADDA(opr )	A+M->A	IMM	8B	ii	2	-	-	X	-	X	X	X	X	
		DIR	9B	dd	3									
		EXT	BB	hh ll	4									
		IND,X	AB	ff	4									
		IND,Y	18AB	ff	5	-	-	X	-	X	X	X	X	
ADDDB(opr )	A+M->B	IMM	CB	ii	2	-	-	X	-	X	X	X	X	
		DIR	DB	dd	3									
		EXT	FB	Hh ll	4									
		IND,X	EB	Ff	4									
		IND,Y	18EB	Ff	5	-	-	X	-	X	X	X	X	
LDAA(opr)	M->A	IMM	86	ii	2	-	-	-	-	X	X	0	-	
		DIR	96	dd	3									
		EXT	B6	Hh ll	4									
		IND,X	A6	Ff	4									
		IND,Y	18A6	Ff	5	-	-	-	-	X	X	0	-	
LDAB(opr)	M->B	IMM	C6	ii	2	-	-	-	-	X	X	0	-	
		DIR	D6	dd	3									
		EXT	F6	Hh ll	4									
		IND,X	E6	Ff	4									
		IND,Y	18E6	Ff	5	-	-	-	-	X	X	0	-	

### 3°) Bien comprendre la partie opérative

Il est important de bien comprendre le fonctionnement général d' un micro-contrôleur pour bien se persuader qu' une connaissance parfaite d' un état permet de trouver les états suivants. Pour un micro-contrôleur on appelle état la connaissance des valeurs des registres et de celles des mémoires.

**Exercice 2** (données en hexadécimal)

On donne les états (connaissance partielle : on ne connaît pas tout, mais c' est suffisant) suivants :

<b>PC=2000</b>			<b>PC=4000</b>	<b>A=FF</b>	<b>C=0</b>	<b>V=0</b>
adresses	mémoire					
1FFF	C6	3FFF	FF			
	86		C6			
	24		FE			
	C6		CB			
	28		18			
2004	1B					

1°) D' après le schéma donné ci-dessus, désassembler la partie utile du code.

2°) Exécuter ensuite les instructions que vous pouvez, et donner les états successifs.

## A2I12 : TD n°2

### I) Cours

#### 1°) Le cahier des charges

Un problème, qu'est-ce que c' est ? Une question qui contient explicitement ou implicitement les éléments permettant d' aboutir à une réponse.

Résoudre un problème c' est répondre à la question qu' il pose ou bien le décomposer en sous-problèmes qu' il faut résoudre à leur tour...

Le cahier des charges est l' énoncé du problème dans lequel on s' est efforcé de minimiser les risques d' ambiguïté en réduisant l' information implicite à des "évidences".

Un produit quel qu' il soit, est dit de qualité s' il est conforme aux spécifications du cahier des charges.

Dans le cahier des charges, on définit les données et les résultats recherchés, sans préjuger de la solution qui sera retenue pour les relier. Autrement dit, on définit le "quoi" sans préjuger du "comment".

Pour reprendre la terminologie du TD 1, résoudre un problème est trouver une suite d' instructions qui permet de passer d' un état de départ (ou plutôt ensemble d' états de départ) à un état final (ou ensemble d' états finaux).

#### 2°) Expressions booléennes (E.B.)

Nous allons maintenant introduire les opérateurs relationnels :

< inférieur à

<= inférieur ou égal à

> supérieur à

>= supérieur ou égal à

== égal à

!= différent

&& et, || ou, sont des connecteurs logiques.

! est le complément logique.

Les valeurs prises par des expressions booléennes sont numériques : 0 (faux) ou 1 vrai.

#### Exercice 1

Soient i,j,k des variables entières contenant les valeurs 1, 2 et 3. Calculer la valeur des expressions :

i < j

(i+j) >= k

(i+j) > (i+4)

k != 3

j == 2

(j==3) || (k!=4)

(i<j) && (k<=4)

c != ' p' si c est de type caractère de valeur ' w'

Quelles valeurs seront dans i pour les différentes affectations :

i = i>j;

i = i+5;

i = i+ (k!=3);

$i = i*(j==2);$

Remarque : pour les deux dernières expressions le retrait des parenthèses ne change pas le résultat.

### 3°) Structures de contrôle

Un programme peut être défini comme un enchaînement d'actions qui modifient en fonction de certaines conditions l'état de l'ensemble V des variables.

L'action est exécutée par un processeur, homme ou machine, qui comprend un certain langage ; elle doit par conséquent être décrite dans ce langage.

La séquence, le choix et l'itération sont à la base de toute l'algorithmique.

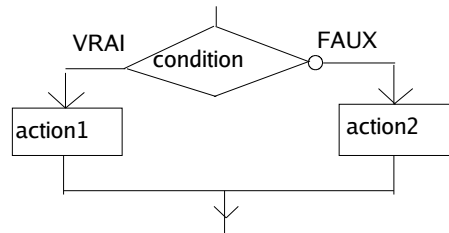
#### a) Enchaînement inconditionnel : la séquence



#### b) Enchaînement conditionnel : le choix

SI condition ALORS :  
  action1  
SINON :  
  action2  
FINSI

SI condition ALORS :  
  action1  
FINSI



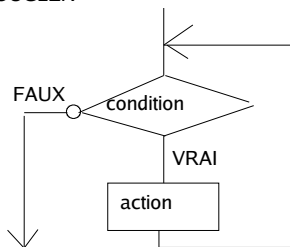
En C ces structures de contrôles seront codées :

```
if (condition) {
    action1;
} else {
    action2;
}
```

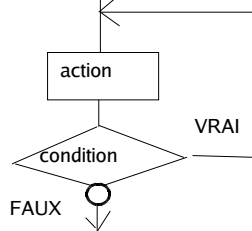
où condition est en général une expression qui donne un résultat différent ou égal à 0. On reconnaît une **expression booléenne** (E.B.)

#### c) Enchaînement répété : l'itération

TANT QUE condition, FAIRE :  
  action  
BOUCLER



REPETER  
  action  
TANT QUE condition



En C ces structures de contrôles seront codées :

```
while (condition) {
    actions;
}
```

```
do {
    actions;
} while (condition);
```

Il existe aussi la structure :

```
for (debut ; condition de non fin ; passage à la suite) {
    actions;
}
```

et aussi :

```
while (1) {
    action1s
    if (condition) break;
    action2s;
}
```

## **II) Exercices C**

1°) Quelles erreurs ont été commises dans chacun des groupes d' instructions suivants :

- a) `if (a<b) printf("ascendant")`  
`else printf("non ascendant");`
- b) `while a<24`  
`a++;`
- c) `do c=getchar() while(c!='\n');`
- d) `do while((c=getchar())!='\n');`
- e) `do {} while (1);`

2°) Écrire plus lisiblement :

```
do {} while(printf("donnez un nombre >0"), scanf("%d",&n), n<=0);
```


3°) Soit le petit programme suivant :

```
#include <stdio.h>
main()
{ int i,n,som;
  som = 0;
  for(i=0;i<4;i++) {
    printf("donnez un entier \n");
    scanf("%d",&n);
    som +=n;
  }
  printf("Somme : %d\n",som);
  return 0;
}
```

A) Écrire un programme réalisant exactement la même chose, en employant à la place du for :

- i) une instruction while
- ii) une instruction do ... while.

B) Modifier ce programme pour qu' il fasse la somme de nombres positifs (on ne connaît pas a priori le nombre de fois qu' il faut exécuter cette boucle). Une entrée nulle sera considérée comme la dernière à rentrer. Pour l' affichage de la somme, on affichera « Grand nombre : valeur de la somme » si la somme dépasse 100 et « Petit nombre : valeur de la somme » dans le cas contraire



**INFO Je retiens**

```

/* On commence par les directives
ici... */
#include <stdio.h>

/** PROGRAMME PRINCIPAL **/
main() {
/** DECLARATIONS VARIABLES ICI**/

    /** INSTRUCTIONS ET STRUCTURES DE
        CONTROLES
        *****/

    return 0;
}

```

Structure d' un programme en C

### III) Cours 68HC11

#### 1°) Les modes d' adressage (suite)

**Direct 16 bits (ou étendu) :** l' adresse de l' opérande se trouve dans les deux octets suivants le code de l' instruction. L' opérande peut être un octet ou un mot de 16 bits.

B6 10 33	ldaa \$1033	charger un octet situé à l' adresse \$1033
FF 11 00	stx \$1100	enregistrer le registre IX à l' adresse \$1100:1101
BD F8 77	jsr \$F877	sauter au sous-programme à l' adresse \$F877

**Direct 8 bits:** l' adresse de l' opérande se trouve dans l' octet suivant le code de l' instruction. L' opérande peut être un octet ou un mot de 16 bits. L' adressage permet d' adresser les octets aux adresses \$0000..\$00FF, dans la première page de la mémoire vive. Au niveau d' assemblage on ne voit pas la différence entre cet adressage et l' adressage de 16 bits. C' est l' assembleur qui prend automatiquement cet adressage lorsque c' est possible.

96 33	ldaa \$0033	charger un octet de l' adresse \$0033
DF 80	stx \$0080	enregistrer le registre IX à l' adresse \$0080:0081
9D B0	jsr \$00B0	sauter au sous-programme à l' adresse \$00B0

**Relatif PC:** Cet adressage est réservé aux instructions de branchement. L' adresse du branchement est calculée par l' adresse de l' instruction diminué jusqu' à -128 ou augmenté jusqu' à +127 par l' octet qui se trouve derrière le code de l' instruction. Au niveau de l' assembleur on indique l' adresse absolue et c' est l' assembleur qui calcule l' adressage.

1F 08 20 FC	brclr \$08,x \$20 *	attendre que le bit \$20 atteigne la valeur 1
-------------	---------------------	---

Comme cet adressage est réservé à un espace très étroit, l' assembleur remplace automatiquement une instruction qui se rend à une adresse trop loin comme 2C ?? bge trop loin brancher si plus grand ou égal par deux instructions 2D 03 blt L brancher si plus petit

7E FA 00	jmp trop loin	sauter si plus grand ou égal
L equ *		

#### Exercice 1 (données en hexadécimal)

On donne les états (connaissance partielle : on ne connaît pas tout, mais c' est suffisant) suivants :

<b>PC=1FFF</b>		<b>PC=3000</b>	<b>B=3F</b>	<b>A=FD</b>	<b>C=0</b>	<b>V=0</b>
adresses	mémoire					
1FFF	B6	2FFF	FE			
	20		B6			
	\$04		\$2F			
	C6		FF			
	28		1B			
2004	1B					

1°) D' après le schéma donné ci-dessus, désassembler la partie utile du code à l' aide du tableau d' instructions du TD1.

2°) Exécuter ensuite les instructions que vous pouvez et donner les états successifs.

## 2°) Mon premier programme en assembleur

```
m equ      $010 ;commentaire
org $8000
    ldaa #m
    jmp suite
suite      ldab ...
```

On distingue dans ce programme une étiquette, une définition symbolique, un commentaire et la définition de l' origine du programme.

## Exercice 2 (adressage relatif PC)

Calculer la valeur du branchement XX dans le programme donné ci-dessous :

<b>Adresse du code</b>	<b>Code</b>	<b>Instruction</b>	<b>Remarques</b>
B600	86 03	LDA A #03	Quand le processeur exécute le BNE, le PC =B605
B602	4A	lci DECA	
B603	26 XX	BNE lci	
B605			

## 3°) Les structures de contrôle

On présente maintenant les structures de contrôle du I-3°) dans le cas du 68HC11.

### 3-1 Structure si-alors-sinon

```
; si accumulateur A nul alors action 1 sinon action 2
    BEQ ACTION1 ; beq branche si 0 c'est a dire Z=1
    BRA ACTION2

; si accumulateur A = 5 alors action 1 sinon action 2
    SUBA #5 ; ou mieux encore CMPA #5
    BEQ ACTION1
```

```
BRA ACTION2
```

```
; si accumulateur A < 5 alors action 1 sinon action 2
SUBA #5 ; ou mieux encore CMPA #5
BLT ACTION1
BRA ACTION2
```

Pour les autres branchements conditionnels voir table 6-9 ( Branch Instructions) et table 6-10 (Jump Instruction) en fin de polycopié.

### 3-2 Structure répéter ... tant que ... et structure tant que...

```
; tant que A>0 faire action
tque CMPA #00 ; on peut aussi utiliser B CMPB
BLE suite ;A<=0 on va a suite
;action qui ne modifie pas A
BRA tque
suite      ....
```

```
; faire action tant que B>0
faire
;action qui ne modifie pas B
CMPB #00 ; on peut aussi utiliser B CMPB
BLE suite ;B<=0 on va a suite
BRA faire
suite      ....
```

### 3-3 Structure for ...

```
; pour i variant de 0 à 15
;(compris) faire action
LDY #$0000 ;ou avec X
for
;action qui ne modifie pas Y
INY ;Y<-Y+1
CMPY #$000F
BLT for
; suite
```

#### ***Technique utilisée par la suite***

```
; pour i variant de 15 à 0
;(compris) faire action
LDY #$000F ;ou avec X
for
;action qui ne modifie pas Y
DEY ;Y<-Y-1
BNE for
; on gagne une instruction
; suite
```

### **Exercice 3**

On suppose qu' en adresse \$1000 se trouvent des valeurs positives (octets) provenant de l' extérieur du 68HC11. Les détails sur la façon dont cela peut marcher ne nous intéressent pas ici. Faire un programme qui fait sans arrêt l' acquisition de 4 de ces valeurs, fait la somme et divise par 4 le résultat pour le ranger le tout à l' adresse \$1004.

#### **Indications**

La difficulté de ce problème réside dans le fait que l' on a à additionner un ensemble de valeur 8 bits qui peut donner un résultat sur plus de 8 bits. Les deux seules instructions du 68HC11 qui permettent d' additionner un nombre sur 8 bit à un nombre sur 16 bits sont ABX (X<-X+B) et ABY (Y<-Y+B). Si l' on prend Y pour le compteur de boucle il nous reste donc ABX.

La division sera réalisée par l' instruction LSRD (table 6.6), il faudra donc transférer X dans D. Cela peut se faire simplement avec XGDY (X<->D) La division par 2 devrait être réalisé par ASRD mais cette instruction n' existe pas, mais nous n' en avons pas besoin car notre résultat est forcément positif.

Voir l' instruction LDD (table 6.1) et les instructions INY et LDY (table 6.7)

## A2I12 TD n°3

### I) Cours

En C, un tableau à une dimension est déclaré ainsi :

```
type_des_éléments nom_du_tableau[nombre_éléments];
```

Les éléments du tableau sont numérotés de 0 à nombre\_éléments-1

Par exemple :

```
char chaine[100];
```

déclare une chaîne de 100 caractères, numérotés de 0 à 99.



La déclaration d' un tableau réalise en fait deux choses :

- réservation de place en mémoire pour les cases du tableau,
- le nom du tableau désigne désormais son adresse.



Par exemple avec `char chaine[100];`  
 Vous venez de dire promis juré que désormais `chaine` désigne l' adresse du tableau qui comporte 100 cases numérotées de 0 à 99. Il ne faut donc pas utiliser le `&` pour désigner l' adresse. Ce sont les termes de votre contrat que vous devez respecter !!!

Un tableau à plusieurs dimensions est déclaré ainsi :

```
type_des_éléments nom_du_tableau[nombre_éléments1][nombre_éléments2];
```

Par exemple, `float t[3][4];` déclare un tableau de 3x4 réels.

Il est possible de déclarer d' abord un type tableau à l' aide de l' instruction `typedef`. Pour la déclaration du nombre d' éléments il est conseillé d' utiliser la directive `#define`.

Par exemple :

```
#define MAX 100
char chaine[MAX];
```

est une déclaration correcte.



Les expressions `chaine[3]` et `*(chaine+3)` désignent la même chose : la 4<sup>o</sup> case du tableau `chaine`.



Il est rarement possible de faire une affectation globale à un tableau ! même si l' on utilise un autre tableaux :

Si je déclare : `int tab1[10], tab2[10];`  
 L' affectation `tab1=tab2;` n' est pas correcte puisque l' on tente d' affecter l' adresse `tab2` à l' adresse `tab1`. Si cette opération était autorisée par un compilateur on gagnerait en souplesse mais on perdrait en fiabilité de programme.

### II) Exercices

1°) Quels résultats fournira le programme :

```
#include <stdio.h>
main() {
    int t[3]={10,20,30};
    int i,j;
    for (i = 0; i < 3;i++) printf ("%d ",t[i]);
    for (i = 0,j = 0;i < 3;i++) t[i] = j++ + 1;
    for (i = 0; i < 3;i++) printf ("%d ",t[i]);
    printf ("\n");
    return 0;
}
```

2°) Soit le tableau t déclaré ainsi : float t[3];

Écrire les (seules) instructions permettant de calculer dans une variable nommée som, la somme des éléments de t. Reprendre le même travail pour une déclaration float t[3][4];

3°) Réaliser un programme qui affiche le résultat de la conversion d' un nombre entier positif, exprimé en base 10, dans une base quelconque (comprise entre 2 et 10). Le nombre à convertir ainsi que la valeur de la base seront fournis en données.

i) Écrire l' algorithme réalisant ce programme.

ii) Coder cet algorithme en C, en utilisant un ou plusieurs tableaux.

### **III)Le 68HC11, les tableaux et l' adressage indexé**

#### **1°) L' adressage indexé**

**Indexé (IX, IY) :** L' adresse est calculée par le contenu du registre IX ou IY en ajoutant une valeur entre \$00 et \$FF qui se trouve dans l' octet suivant le code de l' instruction. Le résultat est sur 16 bits et adresse donc la totalité de l' espace d' adressage. Le programme suivant efface la mémoire vive de l' adresse \$0040 à \$004F

CE 00 40	ldx #\$0040	Commencer à l' adresse \$0040
86 10	ldaa #16	ACCA compteur: 16 octets
6F 00	L: clr 0,x	Effacer l' octet à l' adresse dans IX
08	inx	Augmenter de 1 l' adresse
4A	deca	Diminuer de 1 le compteur
26 FA	bne L	Continuer si c' est pas encore fini

Il peut donc être réalisé avec le registre X ou Y. Par exemple :

```
debuttab EQU $2000
```

```
LDX #debuttab
LDAA 0,X ;case 0 dans A : A<- mem(X+0)
STAA 1,X ;A dans case 1 mem(X+1) <- A
```

#### **2°) Les tableaux**

Les tableaux se gèrent avec cet adressage indexé naturellement. Contrairement à ce qui se passe en langage C, il faut gérer la mémoire des tableaux vous-même.

#### **3°) Exercices**

1) Déterminer les valeurs de A et X suite aux instructions suivantes :

<b>Valeur de A ?</b>		<b>Valeur de X ?</b>	
LDX	#01	LDAB	#\$55
STX	\$100	STAB	\$103
LDAA	\$100	LDX	\$103

2) On suppose qu' en adresse \$1000 se trouvent des valeurs (octets) provenant de l' extérieur du 68HC11. Les détails sur la façon dont cela peut marcher ne nous intéressent pas ici. Faire un programme qui fait l' acquisition de 4 de ces valeurs et les stocke dans un tableau. Ensuite on calculera la somme de ces 4 cases divisée par 4 et le résultat sera rangé à l' adresse \$1004.

3) Modifier ce programme pour qu' il calcule le maximum des 4 valeurs du tableau et le range en mémoire à la suite du tableau.

4) Modifier ce programme pour que le passage d' un tableau de 4 cases à un tableau de 10 (ou autre) cases se fasse seulement par la changement d' une seule valeur dans votre programme.

## A2I12 TD n°4

### I) Cours

Lorsque plusieurs alternatives sont emboîtées, il s'agit d'un choix multiple, ce qui dans un algorithme peut s'écrire :

**AU CAS OU**

condition 1, FAIRE : action 1  
condition 2, FAIRE : action 2  
condition 3, FAIRE : action 3  
condition 4, FAIRE : action 4

**DANS LES AUTRES CAS, FAIRE : action 0**



Attention : L'ordre des conditions, dans un tel choix multiple, est essentiel.

**En C :**

```
switch (variable) { /* variable est de type int ou
char */
case valeur_1 : action_1; break;
case valeur_2 : action_2; break;
...
case valeur_n : action_n; break;
default : action_0;
}
```

Les "break" imposent des conditions exclusives sur la variable. S'ils ne sont pas présents (car non obligatoires) l'algorithme exécuté peut se traduire :

**AU CAS OU**

condition 1, FAIRE : action\_1 et toutes les suivantes  
condition 2, FAIRE : action\_2 et toutes les suivantes  
condition 3, FAIRE : action\_3 et toutes les suivantes  
condition 4, FAIRE : action\_4 et la suivante

**DANS LES AUTRES CAS, FAIRE : action\_0**

Une action peut être une procédure (c'est un sous programme). Celle-ci a l'une des deux structures :

<pre>void NomProcédure(void) {   Code; }</pre>	<pre>void NomProcédure() {   Code; }</pre>
--	--

Elle s'appelle par "NomProcédure();" ou "NomProcédure(void);"

La structure de nos programmes est maintenant du type :

```
#include < .h> /* tous les includes ici */
void proc(void); /* déclaration des prototypes des procédures
ici */
main()
{ int i,j /* déclaration des variables locales au main ici */
```

```

    /* code du main */
    proc(); /* par exemple appel de proc */
    return 0;
} /* fin du main */
void proc(void)
{ /*code de proc */}

```

L' utilisation d' un sous-programme ou procédure mérite quelques explications :



<b>Je déclare</b>	<b>J'appelle</b>	<b>J'écris le code</b>
<code>void truc();</code>	<code>truc();</code>	<code>void truc() {</code> <code>....</code> <code>}</code>



**Attention** : L' utilisation d' un sous-programme nécessite une déclaration qui peut encore être considérée comme un contrat. Vous venez de dire promis juré que désormais votre sous-programme s' appelle truc qu' il n' a pas de paramètre (parenthèse vide) et qu' il ne retourne rien (void)

→ Introduction des variables locales.

## II) Exercices

1°) Soit le programme suivant :

```

#include <stdio.h>
main()
{ int n;
  printf("Entrez une valeur : ");
  scanf("%d", &n);
  switch (n)
  { case 0 : printf("Nul\n");
    case 1 :
    case 2 :
    printf("Petit\n");break;
    case 3 :
    case 4 :
    case 5 : printf("Moyen\n");
    default : printf("Grand\n");
  } // fin du switch
  return 0;
} // fin du main

```

Quels résultats affiche-t-il lorsqu' on lui donne comme entrée :  
a) 0, b) 1, c) 4, d) 10, e) -5.

2°) Pour un nombre fourni donner à l' aide d' un menu le choix à l' utilisateur d' afficher :  $x^2$ ,  $x^3$ ,  $1/x$ ,  $x^{1/2}$ .

Suivant le choix traiter l' entrée :  $1/x$  n' existe pas pour  $x=0$ ,  $x^{1/2}$  n' existe pas pour  $x$  négatif. Écrire des procédures pour le traitement. Revenir au menu initial si l' on veut recommencer.

`sqrt(d)` renvoie la racine carrée de `d` de type double : prototype dans `math.h`

`pow(d1,d2)` double (prototype dans `math.h`) renvoie `d1` élevé à la puissance `d2`.

## III) Les sous-programmes le switch et le 68HC11

Les instructions BSR et JSR RTS (table 6-11) pour l' appel de sous-programme. Le rôle de la pile pour les appels imbriqués.

Il est courant d' initialiser la pile en haut de la première partie mémoire basse entre \$0000 et \$00FF, c' est à dire à \$FF. Cela se fait avec l' instruction LDS \$00FF

La programmation de suivant le cas faire... peut se faire de différente manière :

```

;si A=0 faire le sous programme spgm0 si A=1 faire spgm1
    ; A contient ici une valeur <=N
    LDY #tabchoix ;Y<- addr tabchoix
    LSLA ; A <- A*2
    TAB ; B <- A
    ABY ; Y <- Y+B
    JMP 0,Y
tabchoix BRA spgm0
        BRA spgm1
        ...
        BRA spgmN
suite    ....

spgm0   ;code ici
        BRA suite

```

**Remarque** : les sous-programmes ne sont pas ici de vrais sous-programmes (appelés avec BSR finissant par RTS). D' autre part, cette technique ne vérifie pas le dépassement du tableau...

```

;si B=1 on fait spgm1, si B=2 on fait spgm2
    CMPB #1
    BNE suite
spgm1   ....
        ....
        BRA fin
suite   CMPB #2
        BNE suitel1
spgm2   ....
        ....
        BRA fin
suitel1 CMPB #3
        ...
        CMPB #N
        BNE fin
spgmN   ...
        ...
        ; le BRA fin est inutile pour le dernier
fin     ;c'est fini ici

```

**Remarque** : toutes les étiquettes spgm1 ... spgmN sont absolument inutiles, elles sont utilisées ici pour simplifier la lecture du programme.

## **IV) Exercices**

### **Exercice 1**

Modifier le premier exemple du « suivant le cas faire ... » pour que ce que l' on doit faire se trouve dans des sous-programmes.

### **Exercice 2**

Écrire un programme qui lit sans arrêt la mémoire \$1000 pour la mettre dans A et qui suivant la valeur décimale (1, 2, 4, 8, 16, 32, 64, 128) appelle un des sous-programme do, re, mi, fa, sol, la, si et do2. On ne vous demande pas de détailler les sous-programmes par contre on sait que les appels nécessiteront un branchement à une adresse 16 bits. Pourquoi ne peut-on pas utiliser directement la méthode du premier exemple du « suivant le cas faire ... » ? On utilisera donc la deuxième technique mais on vous impose d' utiliser des sous-programmes dont le contenu sera NOP suivi de RTS (c' est le principe qui nous intéresse).

Modifier ensuite le programme en éliminant les BRA fin (ou JMP fin) mais en gardant des sous-programmes qui auront cette fois un prologue qui vient changer la valeur de la pile pour que le RTS final nous ramène à fin.



**Attention** : ce genre de manipulation de la pile est à mettre au point avec le plus grand sérieux, autrement un plantage est assuré.

### **Exercice 3**

Écrire un programme qui, suivant la valeur présente en adresse \$1000 soit fait la somme des cases d' un tableau de 4 cases, soit calcule la moyenne de ces cases, soit cherche le maximum du tableau.

## A2I12 - TD n°5

### I) Cours

Si un tableau permet de ranger de l' information de même type dans chaque "case", il est possible de définir une structure pour ranger des types différents : c' est la structure ou l' enregistrement.

En C cela se déclare :

```
struct Fiche
{ char Nom[24];
  int Note;
};
```

Nom et Note sont alors appelés des membres ou des champs.

Pour déclarer une variable de ce type on pourra le faire de la façon suivante :

```
struct Fiche demo;
```

ou encore :

```
struct Fiche fiches[23]; qui déclare un tableau de fiches.
```

L' affectation directe se réalisera alors de la façon suivante :

```
strcpy(demo.Nom , "Denis"); /* affectation directe impossible
*/
fiches [6].Note = 18;      /* prototype de strcpy dans
string.h */
```

### II) Exercices

1°) Given the following program :

```
#include <stdio.h>
struct date { char nom[80];
             int jour;
             int mois;
             int annee; };

main(){
  struct date anniv[3] = {"Nathalie",12,9,76,
                        "Agnes",4,3,77,
                        "Florence",16,8,80};

  printf("%s\n",anniv[0].nom);
  printf ("%d\n",++anniv[0].jour);
  printf ("%d\n",anniv[1].mois++);
  printf ("%d %d
%d\n",anniv[2].jour,anniv[2].mois,anniv[2].annee);
  return 0;
}
```

What can we see on the screen when we execute this program ?

2°) On désire gérer une classe de 20 élèves. Le type classe sera défini comme un tableau de type élève. Ce dernier type est défini comme un enregistrement de Nom, Prénom, Date de naissance, moyenne. Le type date de naissance sera une chaîne de caractères.

i) Définir les types correspondants au problème.

ii) Écrire un programme qui permet de remplir ce tableau par demandes successives à l' utilisateur et qui affiche ensuite la classe en la sortant en colonnes : nom, prénom,.....

**III) 68HC11**

La notion d' enregistrement n' étant pas primordiale en assembleur 6811, nous allons poursuivre par un autre thème : les fonctions logiques booléennes (voir Table 6-4).

**Exercice**

Given a truth table, find different methods to perform this boolean function (input address \$1000, output address \$1004).

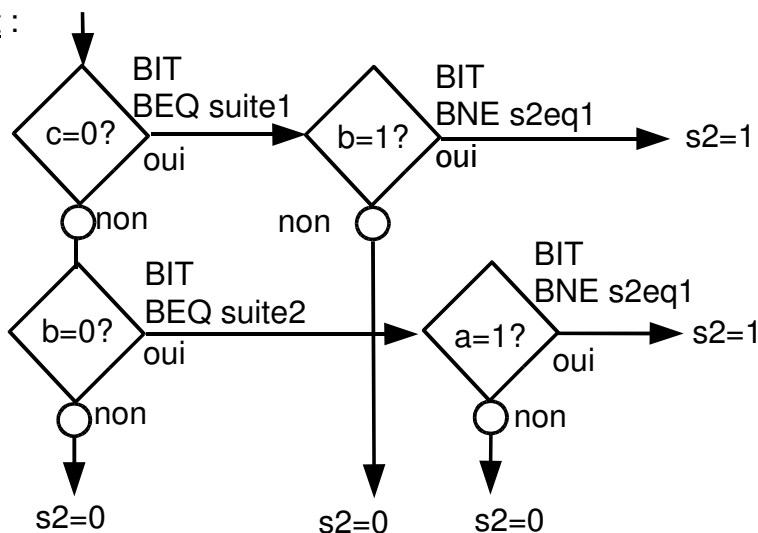
c	b	a	s2	s1	s0
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	0	1	1

- 1°) Show that we can perform this boolean function with an array. Write the complete program which have to initilize this array.
- 2°) Show that we can perform this boolean function with use of logical operation instructions (Table 6-4) or Branch Instructions (Table 6-9).

Write a program for s2 with branch instructions.

Write a program for s1 with logical operation instructions.

Hint :



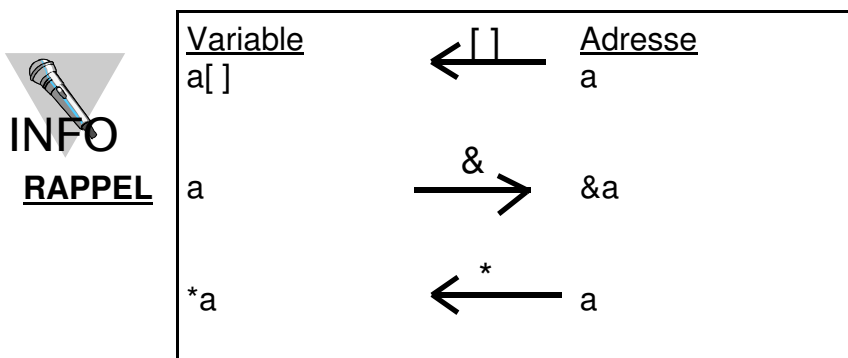
## A2I12 TD n°6

### I) Cours

En C, la portée des variables, c'est à dire les endroits où elles sont connues, dépend de l'endroit où elles ont été déclarées. Il existe cependant des moyens de déroger à cette règle, mais nous ne les examinerons pas ici.

Jusqu'à présent, nous n'avons utilisé que des variables locales au main : elles sont déclarées en tout début de chaque programme après le mot clef "main()". Il existe aussi les variables locales aux procédures ou fonctions, qui sont déclarées en début de procédures et qui ne sont visibles qu'à l'intérieur de ces procédures.

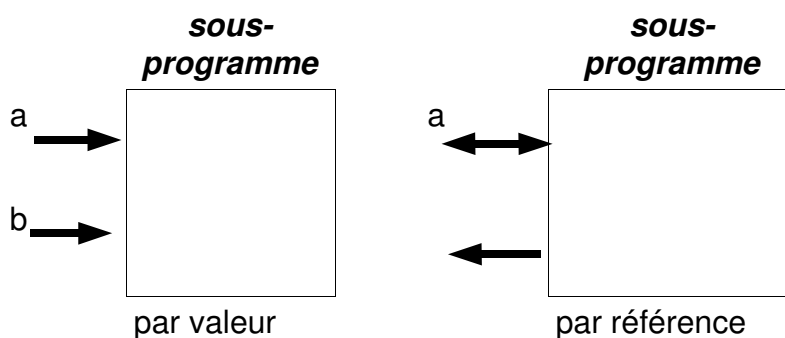
Il existe enfin des variables globales, c'est à dire visibles de tout le programme. Elles sont alors déclarées avant le "main()".



Les procédures permettent le passage de paramètres :


On rappelle qu'il en existe de deux sortes : par valeur et par adresse (ou par référence). Le passage par référence sert entre autre à "retourner" des valeurs au programme appelant.

#### 1°) Quand?



#### 2°) Comment?

##### Passage par valeur



**INFO**

<b>Je déclare</b>	<b>J' appelle</b>	<b>J' écris le code</b>
<code>void truc(int a);</code>	<code>truc(5);</code> <code>truc(c);</code>	<code>void truc(int a) {</code> <code>....</code> <code>}</code>



**Attention** : L' utilisation d' un sous-programme nécessite une déclaration qui peut encore être considérée comme un contrat. Vous venez de dire promis juré que désormais votre sous-programme s' appelle truc qu' il a un seul paramètre par valeur (entier) et qu' il ne retourne rien (void)

**Passage par référence**

Ce qui caractérise le passage par référence est la présence de l' étoile dans la déclaration des paramètres et du et commercial au moment de l' appel. Ceci est vrai sauf pour les tableaux qui eux se passent de toute façon toujours par référence.

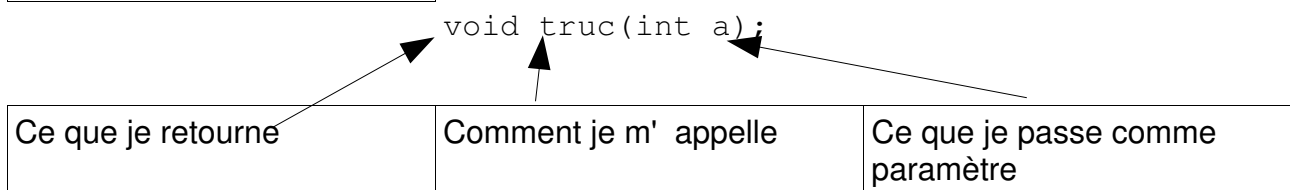


<b>Je déclare</b>	<b>J' appelle</b>	<b>J' écris le code</b>
<code>void truc(int *a);</code>	<code>truc(5);NON!!!! truc(&amp;c);</code>	<code>void truc(int *a) { .... }</code>



**Attention** : L' utilisation d' un sous-programme nécessite une déclaration qui peut encore être considérée comme un contrat. Vous venez de dire promis juré que désormais votre sous-programme s' appelle truc qu' il a un seul paramètre par référence (c' est la petite étoile) et qu' il ne retourne rien (void)

**Déclaration = Contrat**



Respecter le contrat de la déclaration oblige le programmeur à respecter un certain nombre de règles, un peu comme un contrat de mariage.

- Quand j' écris le code la première ligne est identique à ma déclaration sauf que je remplace le « ; » par une accolade ouvrante : « { »
- Quand j' appelle, même si je le fais avec une variable je ne rappelle pas son type. La raison est qu' alors le compilateur prendrait cela comme une nouvelle déclaration.

En résumé nous avons maintenant une structure de programme possible :

```
#include < .h> /* tous les includes ici */
int gi,gj; /* déclaration de variables globales */
void proc(void); /* déclaration des prototypes ici */
void procl(int a,int b,char c);
/* passage de paramètre par valeur */

void proc2(int *a,int *b,char c[]);
/* passage tout par référence */

void proc3(int *a,int b); /* passage de paramètre mixte */
main()
{ int i,j /* déclaration des variables locales au main
ici */
char car;
int *ptr; /* ptr est un pointeur sur un entier
*/
/***** code du main *****/
```

```

    proc();                /* par exemple appel de proc */
    procl(i,j,car);        /* par exemple appel de procl */
    proc2(ptr,&i,&car);      /* par exemple appel de proc2 */
    proc3(ptr,i);          /* par exemple appel de proc3 */
}                          /* fin du main */
void proc(void)
{                          /*code de proc */
} // fin de proc
void procl(int a,int b,char c)
{                          /* code de procl */
} // fin de procl
void proc2(int *a,int *b,char c[])
{ int la;                  /* déclaration de variable locale */
                          /* code de proc2 */
} // fin de proc2
void proc3(int *a,int b)
{ int la;                  /* déclaration de variable
locale */
                          /* code de proc3 */
} // fin de proc3

```

## II) Exercices

1°) Soit le programme C suivant :

```

#include <stdio.h>
void Procl(float x, float y);
void Proc2(float *x,float *y);
void Proc3(float x,float y,float *z);
float a,b,c;

/***** PROGRAMME PRINCIPAL *****/
main() {
    scanf("%f%f",&a,&b);
    printf("%f %f %f\n",a,b,c);
    Procl(a,b);
    printf("%f %f\n",a,b);
    Proc2(&a,&b);
    printf("%f %f\n",a,b);
    Proc3(a,b,&c);
    printf("%f %f %f %f\n",a,b,c,i);
    return 0;
}
/***** FIN PRGM PRINCIPAL *****/

void Procl(float x,float y) {
    printf("%f %f\n",x,y);
    x = x+10;
    y = y*4;
    printf("%f %f\n",x,y);
}

void Proc2(float *x,float *y) {
    printf("%f %f\n",*x,*y);
    *x = *x+10;
    *y = *y * 4;
    printf("%f %f\n",*x,*y);
}

void Proc3(float x,float y,float *z)
{ float i;
    printf("%f %f %f\n",x,y,i);
}

```

On demande :

de nommer les variables locales et les variables globales, de trouver une erreur de portée de variable, erreur détectée à la compilation, d' écrire les valeurs qui seront affichées sur l' écran au fur et à mesure de son déroulement, si l' on suppose que l' utilisateur entre respectivement 2 et 4 pour a et b lors du "scanf". Comment éviter les trois variables globales ?

```

x = x+10;
i = x;
y = y*4;
*z = i+y;
printf("%f %f %f %f\n", x, y, *z, i);
}

```

2°) Écrire un sous programme qui calcule une factorielle de façon itérative. Il y aura donc 2 paramètres à passer un pour le calcul et un pour le résultat.

3°) Reprendre le TD précédent et écrire une procédure qui prend la classe et ressort la moyenne. (hors TD)

### **III)Le passage de paramètres en assembleur 68HC11**

#### **1°) les variables globales**

Chaque fois que l' on se réserve une case mémoire il s' agit naturellement d' une variable globale puisqu' elle est visible partout dans le programme.

#### **2°) les paramètres**

Un document de Steven L. Barnicki (1998) trouvé sur Internet explique le passage de paramètre pour le compilateur C du GNU : GCC, Passing Parameters to Assembly Subroutines. Il a été légèrement simplifié pour nos besoins.

The basic types to pass are 8 bit quantities (bytes, chars) and 16 bit quantities (ints, pointers). All parameters are passed on the stack, starting with an offset of 4 bytes.

The typical prologue and epilogue of the assembly subroutine is:

```

pshy      ;save the stack frame, could use x
tsy       ;set current stack frame Y <- SP
.         ;your code here
.
.
puly      ;epilogue, restore stack frame
rts       ;return from the function

```

The first through N<sup>th</sup> parameter is on the stack, starting with an offset of 4 bytes. If a subroutine has four integer parameters, for example, the prologue, epilogue, and access to the parameters would be:

```

pshy
tsy       ;Y <- SP
ldd 4,y   ;get the first 16 bit parameter
.
.
.
ldd 6,y   ;get the second 16 bit parameter
.
.
.
ldd 8,y   ;get the third 16 bit parameter
.
.
.
ldd 10,y  ;get the fourth 16 bit parameter

```

```

.
.
.
puly
rts

```

L' appel de ces sous-programmes nécessite une attention particulière : il faut supprimer les paramètres de la pile.

#### **IV)Exercices 6811**

- 1) Écrire un sous-programme qui prend un paramètre en entrée et le convertit pour afficheur 7 segments pour l' adresse \$1004 (a poids faible et g poids fort). Le compléter par le programme principal qui lit l' adresse \$1000 en boucle et appelle ce sous-programme.

On donne pour information les valeurs décimales et hexadécimales correspondant aux différents affichages de 0 à F :

- décimales : 63, 6, 91, 79, 102, 109, 125, 7, 127, 111, 110, 124, 57, 94, 121, 113
- hexadécimales : 3F, 06, 5B, 4F, 66, 6D, 7D, 07, 7F, 6F, 77, 7C, 39, 5E, 70, 71

- 2) Écrire un programme qui passe deux paramètres à un sous-programme : l' adresse du début d' un tableau d' octets et son nombre de cases. Ce sous programme calculera la somme de toutes les cases (forcément sur 16 bits) et n' en fera rien ?!!  
Pour simplifier, le programme principal sera chargé uniquement d' initialiser et d' appeler correctement le sous programme.

## A2I12 TD n°7

### I) Cours

Notion de fonction. L' instruction return.

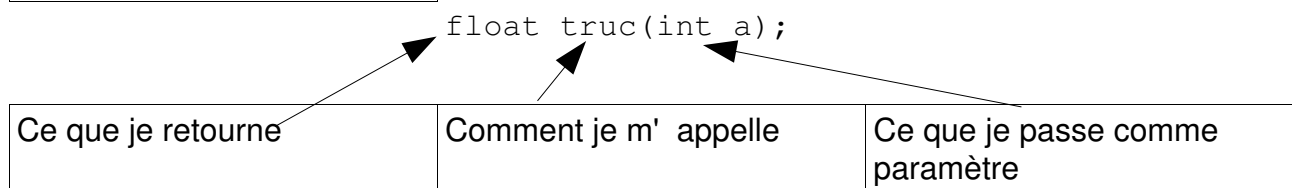


<b>Je déclare</b>	<b>J' appelle</b>	<b>J' écris le code</b>
<code>float truc(int a);</code>	<code>d=truc(5); d=truc(c);</code>	<code>float truc(int a) { .... return x; }</code>



**Attention** : L' utilisation d' un sous-programme nécessite une déclaration qui peut encore être considérée comme un contrat. Vous venez de dire promis juré que désormais votre sous-programme s' appelle truc qu' il a un seul paramètre par valeur (de type int) et qu' il retourne un flottant (float)

**Déclaration = Contrat**



Respecter le contrat de la déclaration oblige le programmeur à respecter un certain nombre de règles, un peu comme un contrat de mariage.

- Quand j' écris le code de la fonction la première ligne est identique à ma déclaration sauf que je remplace le « ; » par une accolade ouvrante : « { »
- Quand j' appelle, même si je le fais avec une variable je ne rappelle pas son type. La raison est qu' alors le compilateur prendrait cela comme une nouvelle déclaration.
- Quand j' appelle une fonction je dois mettre la valeur retournée quelquepart, en général dans une variable prévue à cet effet.

### II) Exercices

1°) Soit le programme C suivant :

```

/***** DIRECTIVES *****/
#include <stdio.h>

/***** DECLARATIONS *****/
float a,b,c;
void Procl(float x,float y);
float Proc2(float x);
float Proc3(float x,float y,float *z);

/**** PROGRAMME PRINCIPAL *****/

main () {
    scanf ("%f %f",&a,&b);

```

```

printf("%f %f %f\n", a, b, c);
Proc1(a, b);
printf("%f %f\n", a, b);
b=Proc2(a);
printf("%f %f\n", a, b);
a = Proc3(a, Proc2(b), &c);
printf("%f %f %f %f\n", a, b, c, i);
return 0;
}
/ *** DEBUT DES FONCTIONS *** /

void Proc1(float x, float y)
{ printf("%f %f\n", x, y);
  x = x+10;
  y= y*4;
  printf("%f %f\n", x, y);
}

float Proc2(float x)
{ float y;
  printf("%f\n", x);
  x = x+10;
  y = x*4;
  printf("%f %f\n", x, y);
  return(y*2);
}

float Proc3(float x, float y, float *z)
{ float i;
  printf("%f %f %f \n", x, y, i);
  x = x+10;
  i = x;
  y = y*4;
  *z=i+y;
  printf("%f %f %f %f\n", x, y, *z, i);
  return(*z+x+y);
}

```

On demande :

- de nommer les variables locales et les variables globales,
- de trouver une erreur de portée de variable, erreur détectée à la compilation,
- d' écrire les valeurs qui seront affichées sur l' écran au fur et à mesure de son déroulement, si l' utilisateur entre respectivement 3 et 5 pour a et b lors du "scanf".

Peut-on éviter les trois variables globales ?

2) Écrire un programme qui utilise une fonction factorielle (avec calcul itératif).

3) Écrire une fonction qui reçoit en arguments 2 nombres flottants et un caractère et qui fournit un résultat correspondant à l' une des 4 opérations appliquées à ses deux premiers arguments, en fonction de la valeur du dernier, à savoir :

- addition pour le caractère +
- soustraction pour le caractère -
- division pour le caractère /
- multiplication pour le caractère \*

Écrire un petit programme main utilisant cette fonction pour effectuer les 4 opérations sur 2 nombres fournis en données.



### Structure d'un programme en C

```

/* On commence par les directives ici...
*/
#include <stdio.h>

/* DECLARATION DES VARIABLES GLOBALES*/
/**/ DECLARATION DES PROTOTYPES /**/

/**/ PROGRAMME PRINCIPAL /**/
main() {
/* DECLARATIONS VARIABLES LOCALE ICI */

    /**/ INSTRUCTIONS ET STRUCTURES DE
        CONTROLES ET APPELS PROCEDURES**/

    return 0;
}

/**/** PROCEDURES ET FONCTIONS /**/**/**/

/* TYPE */ Proc( /* PARAMETRES */ ) {
/* DECLARATIONS VARIABLES LOCALE ICI */

    /**/ INSTRUCTIONS ET STRUCTURES DE
        CONTROLES ET APPELS PROCEDURES**/
}

```

### **III) Le 68HC11 et ses périphériques**

L' importance du Chip Select en général actif à l' état bas (CS)

Le décodage d' adresses.

**V) Exercice 6811** (Ce TD sur le matériel est en fait réalisé en deuxième année)

**Exercice 1** (Décodage d'adresses)

On vous donne un schéma partiel d'une carte micro-contrôleur. Analyser ce schéma et répondre aux questions.

1°) Quelle est la capacité de l'EPROM ?

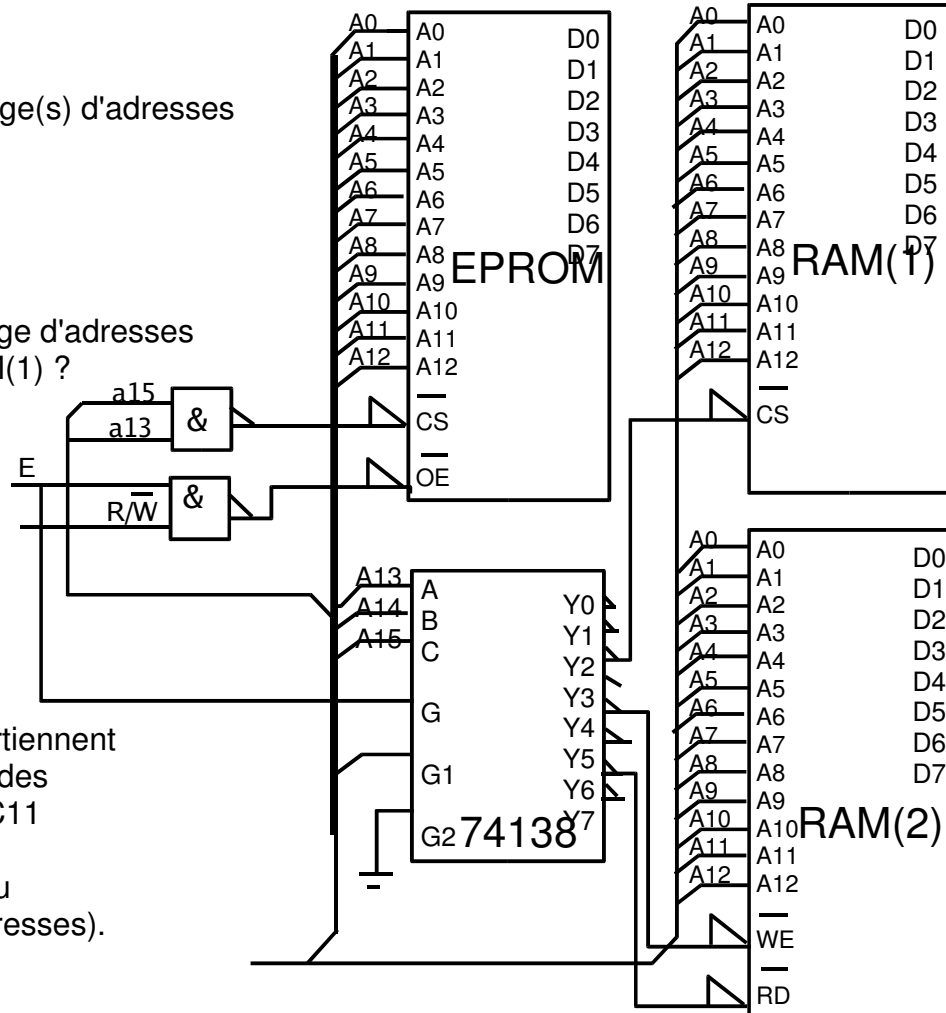
**Réponse :**

2°) Sur quelle(s) page(s) d'adresses apparaît l'EPROM ?

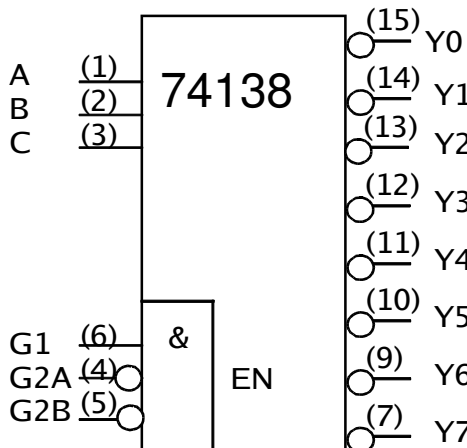
**Réponse :**

3°) Quelle est la plage d'adresses d'écriture de la RAM(1) ?

**Réponse :**



E et R/W appartiennent au diagramme des temps du 68HC11 (inutile pour le combinatoire du décodage d'adresses).



Entrées					Sorties							
G1	G2	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	H	H	H	L	H	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	H	L	H	H	H	H	H	L	H	H
H	L	H	H	H	H	H	H	H	H	H	L	L

## ANNEXE : les instructions du 68HC11

**Table 6-1. Load, Store, and Transfer Instructions**

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH	
Clear Memory Byte	CLR			7F hhll	6F ff	186F ff		
Clear Accumulator A	CLRA						\$4F	
Clear Accumulator B	CLRB						\$5F	
Load Accumulator A	LDAA	86 ii	96 dd	B6 hhll	A6 ff	18A6 ff		
Load Accumulator B	LDAB	C6 ii	D6 dd	F6 hhll	E6 ff	18E6 ff		
Load Double Accumulator D	LDD	CC jjkk	DC dd	FC hhll	EC ff	18EC ff		M->A, M+1->B
Pull A from Stack	PULA						\$32	SP<- SP+1 A<-stack
Pull B from Stack	PULB						\$33	
Push A onto Stack	PSHA						\$36	A->stack, SP<-SP-1
Push B onto Stack	PSHB						\$37	
Store Accumulator A	STAA		97 dd	B7 hh ll	A7 ff	18A7 ff		A=>M
Store Accumulator B	STAB		D7 dd	F7 hh ll	E7 ff	18E7 ff		B=>M
Store Double Accumulator D	STD		DD dd	FD hh ll	ED ff	18ED ff		A=>M, B=>M+1
Transfer A to B	TAB						\$16	
Transfer A to CCR	TAP						\$06	
Transfer B to A	TBA						\$17	
Transfer CCR to A	TPA						\$07	
Exchange D with X	XGDX						\$8F	
Exchange D with Y	XGDY						\$188 F	

**Table 6-2. Arithmetic Operation Instructions**

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Add Accumulators	ABA						X
Add Accumulator B to X	ABX						X

Function	Mnemonic	IMM	DIR	EXT	IND X	INDY	INH
Add Accumulator B to Y	ABY						X
Add with Carry to A	ADCA	X	X	X	X	X	
Add with Carry to B	ADCB	X	X	X	X	X	
Add Memory to A	ADDA	X	X	X	X	X	
Add Memory to B	ADDB	X	X	X	X	X	
Add Memory to D (16 Bit)	ADDD	X	X	X	X	X	
Compare A to B	CBA						X
Compare A to Memory	CMPA	X	X	X	X	X	
Compare B to Memory	CMPB	X	X	X	X	X	
Compare D to Memory (16 Bit)	CPD	X	X	X	X	X	
Decimal Adjust A (for BCD)	DAA						X
Decrement Memory Byte	DEC			X	X	X	
Decrement Accumulator A	DECA						X
Decrement Accumulator B	DECB						X
Increment Memory Byte	INC			X	X	X	
Increment Accumulator A	INCA						X
Increment Accumulator B	INCB						X
Twos Complement Memory Byte	NEG			X	X	X	
Twos Complement Accumulator A	NEGA						X
Twos Complement Accumulator B	NEGB						X
Subtract with Carry from A	SBCA	X	X	X	X	X	
Subtract with Carry from B	SBCB	X	X	X	X	X	
Subtract Memory from A	SUBA	X	X	X	X	X	
Subtract Memory from B	SUBB	X	X	X	X	X	
Subtract Memory from D (16 Bit)	SUBD	X	X	X	X	X	
Test for Zero or Minus	TST			X	X	X	
Test for Zero or Minus A	TSTA						X
Test for Zero or Minus B	TSTB						X

**Table 6-3. Multiply and Divide Instructions**

Function	Mnemonic	INH
Multiply (A × B .D)	MUL	X
Fractional Divide (D ÷ X .X; r .D)	FDIV	X

Function	Mnemonic	INH
Integer Divide ( $D \div X .X; r .D$ )	IDIV	X

**Table 6-4. Logical Operation Instructions**

Function	Mnemonic	IMM	DIR	EXT	IND X	IND Y	INH
AND A with Memory	ANDA	X	X	X	X	X	
AND B with Memory	ANDB	X	X	X	X	X	
Bit(s) Test A with Memory	BITA	X	X	X	X	X	
Bit(s) Test B with Memory	BITB	X	X	X	X	X	
One's Complement Memory Byte	COM			X	X	X	
One's Complement A	COMA						X
One's Complement B	COMB						X
OR A with Memory (Exclusive)	EORA	X	X	X	X	X	
OR B with Memory (Exclusive)	EORB	X	X	X	X	X	
OR A with Memory (Inclusive)	ORAA	X	X	X	X	X	
OR B with Memory (Inclusive)	ORAB	X	X	X	X	X	

**Table 6-5. Data Testing and Bit Manipulation Instructions**

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Bit(s) Test A with Memory	BITA	X	X	X	X	X	
Bit(s) Test B with Memory	BITB	X	X	X	X	X	
Clear Bit(s) in Memory	BCLR			X	X	X	
Set Bit(s) in Memory	BSET			X	X	X	
Branch if Bit(s) Clear	BRCLR			X	X	X	
Branch if Bit(s) Set	BRSET			X	X	X	

**Table 6-6. Shift and Rotate Instructions**

Function	Mnemonic	IMM	DM	EXT	INDX	INDY	INH
Arithmetic Shift Left Memory	ASL			X	X	X	
Arithmetic Shift Left A	ASLA						X
Arithmetic Shift Left B	ASLB						X
Arithmetic Shift Left Double	ASLD						X
Arithmetic Shift Right Memory	ASR			X	X	X	

Function	Mnemonic	IMM	DM	EXT	INDX	INDY	INH
Arithmetic Shift Right A	ASRA						X
Arithmetic Shift Right B	ASRB						X
(Logical Shift Left Memory)	(LSL)			X	X	X	
(Logical Shift Left A)	(LSLA)						X
(Logical Shift Left B)	(LSLB)						X
(Logical Shift Left Double)	(LSLD)						X
Logical Shift Right Memory	LSR			X	X	X	
Logical Shift Right A	LSRA						X
Logical Shift Right B	LSRB						X
Logical Shift Right D	LSRD						X
Rotate Left Memory	ROL			X	X	X	
Rotate Left A	ROLA						X
Rotate Left B	ROLB						X
Rotate Right Memory	ROR			X	X	X	
Rotate Right A	RORA						X
Rotate Right B	RORB						X

**Table 6-7. Stack and Index Register Instructions**

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Add Accumulator B to X	ABX						X
Add Accumulator B to Y	ABY						X
Compare X to Memory (16 Bit)	CPX	X	X	X	X	X	
Compare Y to Memory (16 Bit)	CPY	X	X	X	X	X	
Decrement Stack Pointer	DES						X
Decrement Index Register X	DEX						X
Decrement Index Register Y	DEY						X
Increment Stack Pointer	INS						X
Increment Index Register X	INX						X
Increment Index Register Y	INY						X
Load Index Register X	LDX	X	X	X	X	X	
Load Index Register Y	LDY	X	X	X	X	X	
Load Stack Pointer	LDS	X	X	X	X	X	
Pull X from Stack	PULX						X
Pull Y from Stack	PULY						X

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Push X onto Stack	PSHX						X
Push Y onto Stack	PSHY						X
Store Index Register X	STX	X	X	X	X	X	
Store Index Register Y	STY	X	X	X	X	X	
Store Stack Pointer	STS	X	X	X	X	X	
Transfer SP to X	TSX						X
Transfer SP to Y	TSY						X
Transfer X to SP	TXS						X
Transfer Y to SP	TYS						X
Exchange D with X	XGDX						X
Exchange D with Y	XGDY						X

**Table 6-8. Condition Code Register Instructions**

Function	Mnemonic	INH
Clear Carry Bit	CLC	X
Clear Interrupt Mask Bit	CLI	X
Clear Overflow Bit	CLV	X
Set Carry Bit	SEC	X
Set Interrupt Mask Bit	SEI	X
Set Overflow Bit	SEV	X
Transfer A to CCR	TAP	X
Transfer CCR to A	TPA	X

**Table 6-9. Branch Instructions**

Function	Mnemonic	REL	DIR	INDX	INDY	Comments
Branch if Carry Clear	BCC	X				C = 0 ?
Branch if Carry Set	BCS	X				C = 1 ?
Branch if Equal Zero	BEQ	X				Z = 1 ?
Branch if Greater Than or Equal	BGE	X				Signed =
Branch if Greater Than	BGT	X				Signed >
Branch if Higher	BHI	X				Unsigned >
Branch if Higher or Same (same as BCC)	BHS	X				Unsigned =
Branch if Less Than or Equal	BLE	X				Signed =
Branch if Lower (same as BCS)	BLO	X				Unsigned <
Branch if Lower or Same	BLS	X				Unsigned =
Branch if Less Than	BLT	X				Signed <
Branch if Minus	BMI	X				N = 1 ?

Function	Mnemonic	REL	DIR	INDX	INDY	Comments
Branch if Not Equal	BNE	X				Z = 0 ?
Branch if Plus	BPL	X				N = 0 ?
Branch if Bit(s) Clear in Memory Byte	BRCLR		X	X	X	Bit Manipulation
Branch Never	BRN	X				3-cycle NOP
Branch if Bit(s) Set in Memory Byte	BRSET	X				Bit Manipulation
Branch if Overflow Clear	BVC					V = 0 ?
Branch if Overflow Set	BVS					V = 1 ?

**Table 6-10. Jump Instruction**

Function	Mnemonic	DIR	EXT	INDX	INDY	INH
Jump	JMP	X	X	X	X	

**Table 6-11. Subroutine Call and Return Instructions**

Function	Mnemonic	REL	DIR	EXT	INDX	INDY	INH
Branch to Subroutine	BSR	X					
Jump to Subroutine	JSR		X	X	X	X	
Return from Subroutine	RTS						X

**Table 6-12. Interrupt Handling Instructions**

Function	Mnemonic	INH
Return from Interrupt	RTI	X
Software Interrupt	SWI	X
Wait for Interrupt	WAI	X

**Table 6-13. Miscellaneous Instructions**

Function	Mnemonic	INH
No Operation (2-cycle delay)	NOP	X
Stop Clocks	STOP	X
Test	TEST	X