PROGRAMMATION EN ASSEMBLEUR.

I. Introduction :

Au cœur de tout système microprogrammé, se trouve un microprocesseur. C'est lui qui gère toutes les opérations à effectuer. Pour réaliser ces opérations, chaque microprocesseur possède un jeu d'instruction codées en logique binaire. C'est un ensemble de codes qui forme le **langage machine.** Pour définir l'enchaînement des opérations à effectuer, il suffit d'écrire un **programme**. Seulement, l'écriture d'un programme à l'aide des instructions codées en binaire est pratiquement impossible et totalement inefficace. Aussi, pour faciliter l'écriture des programme, on utilise des langages évolués. On peut distinguer deux types de langage évolué :

- **Les langages de bas niveau :**
  Ce sont des langages proches du jeu d'instruction du microprocesseur. Ils traduisent chaque code du langage machine à l'aide de **mnémoniques** associés aux différents modes d'adressages supportés par le microprocesseur. Chaque microprocesseur ayant un jeu d'instruction particulier, il y aura donc un langage par microprocesseur. On les nomme des **langages Assembleurs.** Pour interpréter et traduire en langage machine un programme écrit en assembleur, on utilisera un **Assembleur** qui sera donc spécifique à chaque microprocesseur.

- **Les langages de haut niveau :**
  Ce sont des langages indépendants du type de microprocesseur. De plus, à chaque instruction de ces langages ne correspond pas une seule instruction en langage machine, mais plusieurs. Ils permettent de créer des programmes compliqués beaucoup plus facilement et rapidement. De ce fait, ce sont les langages les plus utilisés de nos jours. Pour interpréter et traduire en langage machine un programme écrit en langage de haut niveau, on utilise un **Compilateur** qui sera spécifique au microprocesseur utilisé, mais le **programme source** pourra être le même pour différents microprocesseur. Il existe de nombreux langages de haut niveau. On retiendra notamment : **PASCAL, ADA, FORTRAN, BASIC, COBOL, C, C++, JAVA, ...**

Ce cours ne s'occupe que de la programmation en langage **Assembleur**. Contrairement aux langages de haut niveau, l'assembleur n'est pas un langage structuré. Cela constitue sont principal inconvénient mais aussi un de ses avantages : en effet, cela permet une grande liberté dans l'organisation des différentes instructions constitutives d'un programme. Mais cela rend du même coup, difficile la lecture et la compréhension du programme.

Aussi, pour faire en sorte qu'un programme soit lisible, compréhensible et facile à maintenir, il est nécessaire qu'il soit **structuré**. Pour structurer un programme, il faut l'écrire avec méthode. Dans la suite de ce cours, nous allons étudier une méthode qui nous permettra de résoudre les problèmes de façons structurée et de telle sorte que l'écriture du programme dans un langage donné, soit la dernière étape de la résolution.

II. Programmation en langage Assembleur :

II.1. Méthodologie :

C'est au philosophe Descartes que l'on doit les début de la démarche scientifique avec la parution du « Discours de la méthode » dans lequel il résume sa méthode d'analyse des problèmes en quatre principes : faire apparaître les problèmes à l'aide du doute, diviser les problèmes en problèmes plus simples pour diviser la difficulté, ordonner la résolution des problèmes en commençant par les plus simples pour remonter au plus compliqué, faire des dénombrements si entiers et des revues si générales afin d'être assuré de ne rien oublier. Si, contrairement à Descartes, nous n'allons pas essayer de démontrer l'existence de Dieu, nous pouvons formaliser la méthode de sorte qu'elle nous aide à établir l'algorithme de résolution d'un problème.

De la même façon que pour les systèmes techniques, l'étude d'un programme pourra se faire à l'aide de l'analyse fonctionnelle. Chaque partie du programme étant associée à une fonction particulière, il suffira que l'agencement de celles-ci réalise correctement les fonctions principales définies dans le cahier des charges. Le programme pourra donc se décomposer en fonctions qui devront être aussi indépendantes que possible afin qu'elles puissent être utilisé, tel quel, dans un autre programme.

En plus de l'analyse fonctionnelle, la description du fonctionnement des programmes se fera à l'aide d'algorithmes et de leur représentation graphique, les algorigrammes. Ils permettent de connaître ce que fera un programme et comment il le fera de manière totalement indépendante du langage de programmation utilisé.

Pour la conception des programmes, on pourra donc résumer la méthode comme suit :

- **Énoncé du problème :**

A l'aide du cahier des charges, on se pose des questions qui font apparaître les problèmes à résoudre.

- **Analyse :**

On analyse les différents problèmes afin de faire apparaître ce dont on aura besoin pour les résoudre. C'est à ce moment que l'on fait l'analyse fonctionnelle.

- **Algorithme :**

On réalise chacune des fonctions précédemment définies à l'aide des algorithmes. On commence par les fonctions principales pour terminer avec les fonctions secondaires. A la fin de cette étape, les problèmes doivent être résolus.

- **Programmation :**

Il ne reste plus qu'à traduire les algorithmes dans le langage de programmation choisit.

- **Exécution :**

On vérifie le fonctionnement du programme en l'exécutant sur une machine. On fait un maximum d'essais afin de trouver les éventuels problèmes non prévus et que le programme ne résout pas.

- **FIN**

Si tous les résultats sont corrects, alors le programme est fini. Sinon, il faut recommencer l'étude depuis l'analyse.

II.2. Algorithmes et algorigrammes :

L'algorithme permet de décrire, en français, le déroulement d'un processus, d'un programme de façon indépendante du langage de programmation utilisé. Bien qu'il y ait au moins autant de solutions que de problèmes, on montre que tout algorithme peut s'écrire à l'aide d'un nombre restreint de structures de bases.
L'algorithme représente les actions qui sont réalisées dans l'ordre de leur exécution.
L'algorigramme constitue la traduction graphique de l'algorithme.

II.2.1. Structure séquentielle :

```
DEBUT
|     LIRE (a)
|     ECRIRE ( racine_carré(a) )
FIN
```

Il y a passage d'une action à la suivante à la seule condition que la précédente soit exécutée.

II.2.1.1. Affectations :

A un instant donné, une variable ne peut avoir qu'une seule valeur. Cette valeur reste mémorisée jusqu'à ce qu'une nouvelle affectation efface la précédente.
Dans les algorithmes, on représente une affectation par :

Var1 ← 25 est équivalent à **Var1 = 25**
Une autre notation : **Var1 ppv 25** soit **Var1 p**rend **p**our **v**aleur **25**

II.2.1.2. Opérateurs arithmétiques :

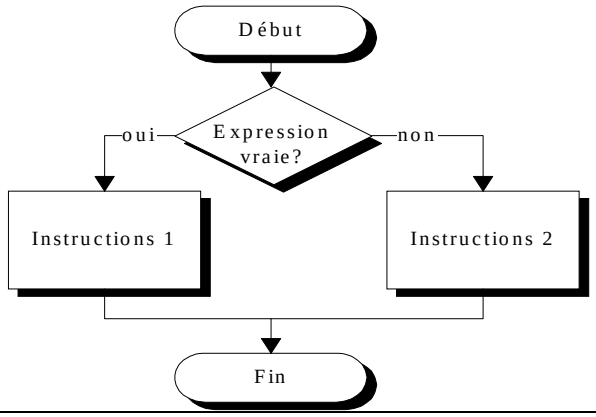
La majorité des traitements envisageables sur un ordinateur exige l'exécution de calculs. Les opérateurs qui permettent ces calculs dépendent du type des variables concernées. Pour les variables numériques, les opérateurs arithmétiques sont :

+ (Addition) ; **-** (Soustraction) ; ***** (multiplication) ; **/** (Division) ; **%** (Reste de la division entière) ;

II.2.2. Structure alternative :

Une expression logique est testée. Si l 'expression logique est vrai alors un bloc d'instruction associé est exécuté. Sinon, donc si elle est fausse, un autre bloc d'instruction est exécuté.

Algorithme	Algorigramme

DEBUT
  SI (*expression logique)* vraie
  ALORS
  Bloc d'instructions1
  SINON
  Bloc d'instructions 2
  FIN SI
FIN
	

- **Les expressions logiques :**

L'expression logique est une expression booléenne. Elle ne peut donc prendre que 2 valeur différente : Vrai ou Faux. Elle peut utiliser tous les opérateur relationnels ainsi que les opérateurs booléens.

Opérateur relationnel	Signification	Exemple
<	Inférieur	Delta < 0
<=	Inférieur ou égal	C<= 2
=	Égal	X = y
>	Supérieur	C > 2
>=	Supérieur ou égal	A >= B
≠	Différent	Y ≠ X

On peut réaliser des opérations booléennes avec les expressions contenant des opérateurs relationnels :

SI ((a<=0) OU (a>=100)) ALORS .....

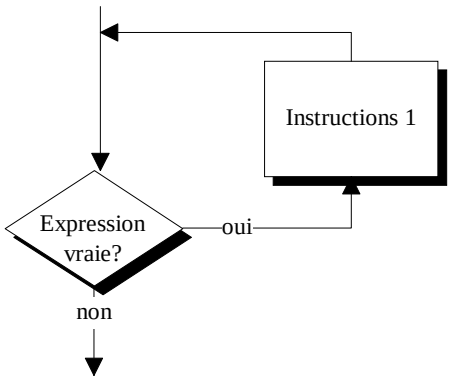
Les opérateurs booléens sont : **ET OU NON** et **OU EXCLUSIF**

II.2.3. Structures de contrôle :

II.2.3.1. Structures répétitives :

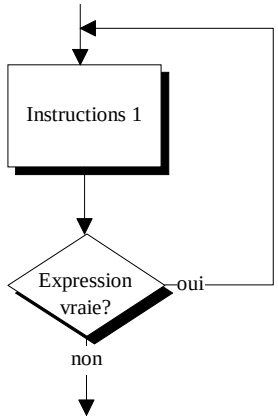
A- Première forme :

Algorithme	Algorigramme

TANT QUE (*Expression logique)* vraie **FAIRE**
  DEBUT
  Bloc d'instruction(s)
  FIN
FIN TANT QUE
	

Le bloc d'instruction est exécuté TANT QUE l'expression logique est vraie. Si la condition n'est pas remplie lors de la première évaluation, le bloc d'instruction ne sera jamais exécuté.

B- Deuxième forme :

Algorithme	Algorigramme
FAIRE **DEBUT** Bloc d'instruction(s) **FIN** **TANT QUE** (*Expression logique)* vraie	

Dans cette forme, le bloc d'instruction est exécuté au moins une fois.
Une variante existe pour chacune des formes précédentes. Elle consiste à vérifier si l'expression logique est fausse. Dans ces cas, le bloc d'instruction n'est exécuté que si la condition n'est pas réalisée. Mais ces variantes sont totalement équivalentes, et il suffit de complémenter l'expression logique pour retrouver ces deux formes :

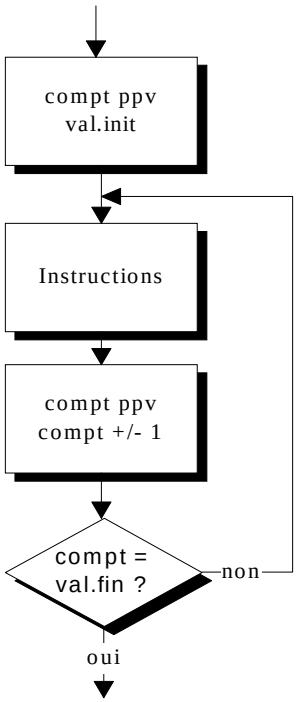
FAIRE
DEBUT
Bloc d'instruction(s)
FIN
TANT QUE (*Expression logique)* fausse

Équivalant à

FAIRE
DEBUT
Bloc d'instruction(s)
FIN
TANT QUE (***NON(****Expression logique))* vraie

C- Troisième forme : forme itérative :

Dans cette forme, ce n'est plus une expression logique qui permet de sortir de la boucle, mais un compteur. A l'entrée de la boucle on initialise une variable entière. On indique sa valeur finale, si celle-ci est supérieure à la valeur initiale, on augmente la variable compteur, sinon on la diminue.

Algorithme	Algorigramme
POUR (compt = val.init) **JUSQU'À** (compt = val.fin) **FAIRE** **DEBUT** Bloc d'instruction(s) Compt ← compt +/- 1 **FIN** **FIN POUR**	

D- Quatrième forme : structure à choix multiples :

Lorsque des actions différentes peuvent être exécutées en fonction de la valeur d'une variable, on peut utiliser la structure alternative pour comparer la valeur de la variable aux valeurs autorisant l'exécution d'un bloc d'instruction. Mais, la structure alternative ne permet de comparer une variable qu'à un seul cas à la fois. Donc, si on a plusieurs cas possibles, il faudra autant de structures alternatives que de cas possibles. Cela risque d'alourdir l'algorithme. La structure à choix multiples, permet donc de résoudre ce problème. Mais il ne faut pas oublier que ce n'est qu'une mise en forme plus simple de plusieurs structures alternatives.

Algorithme	Algorigramme

SELON (variable) **FAIRE**

DEBUT

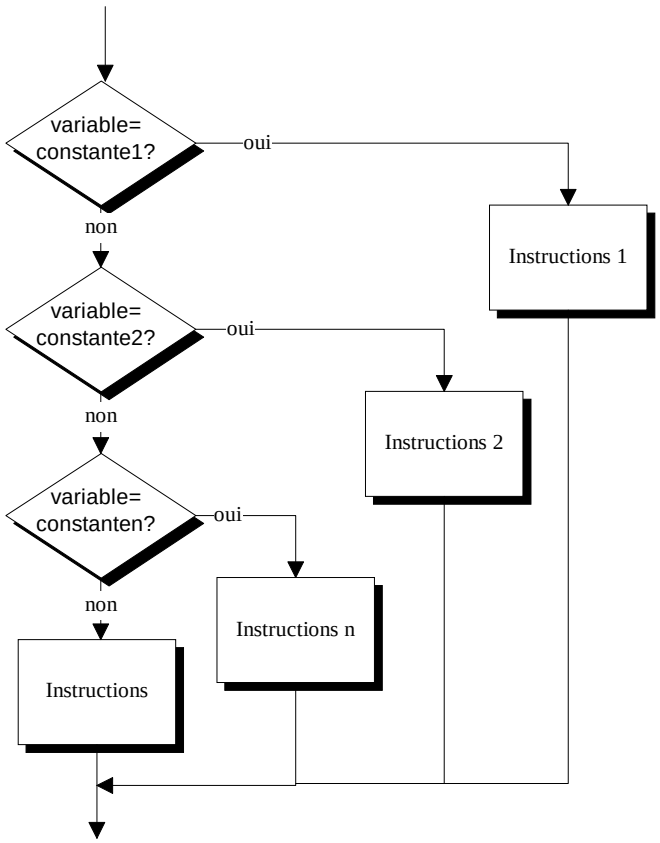
Constante1 : Bloc d'instruction(s) 1
QUITTER SELON

Constante2 : Bloc d'instruction(s) 2
QUITTER SELON

Constante n : Bloc d'instruction(s) n
QUITTER SELON

Autres cas : Bloc d'instruction(s)
QUITTER SELON

FIN
FIN SELON



II.3. L'ordinogramme :

L'***ordinogramme*** assure également, sous forme graphique, la description du déroulement du programme, mais contrairement à l'algorigramme, ***il fait explicitement référence au µC ou au µP chargé d'exécuter le programme***.
Par conséquent, les différents symboles graphiques pourront faire référence aux registres internes du µC utilisé.

Afin de respecter au mieux les indications portées par MOTOROLA dans le document ***CPU12 Reference Guide***, on utilisera, pour chacun des symboles, les règles syntaxiques conformes aux exemples suivants :