Chapitre 5
Éléments de base de l'assembleur

5.1 Règles d'assemblage

5.1.1 Identificateurs

Les symboles définis par le programmeur en assembleur PEP 8 doivent avoir un nombre de caractères inférieur ou égal à 8. Les caractères alphanumériques utilisés pour l'écriture des symboles comprennent les 26 lettres de l'alphabet majuscule, les 26 lettres de l'alphabet minuscule et les dix chiffres décimaux. Notez bien que les lettres accentuées sont exclues. Un symbole doit nécessairement débuter par une lettre.
Exemples:

```
zone3
lireEnti
Somme
Q123
a
B3B8
```

5.1.2 Constantes

Les constantes apparaissant dans les instructions d'un programme Assembleur peuvent être spécifiées de différentes façons (nombre décimal, nombre hexadécimal et caractère), mais dans tous les cas l'assembleur convertira la valeur en son équivalent binaire avant de s'en servir ou de l'inclure dans le programme objet.

Les nombres décimaux sont les plus faciles à spécifier: il suffit de les écrire sous leur forme habituelle avec ou sans signe, comme 19 ou 43227.

Les valeurs hexadécimales ou caractères (identifiées par leur code) sont précédées des deux caractères 0x ou 0X indiquant le type de la valeur. Les valeurs hexadécimales apparaissent donc comme 0x03E ou 0x165.

Les constantes caractères comprennent un seul caractère, lequel est placé entre deux apostrophes comme '*', ou sont représentées par leur code numérique ASCII.

C'est au programmeur de choisir la forme la plus pratique pour une constante donnée. Ainsi '*', 0x2A et 42 représentent la même valeur, mais s'il s'agit du caractère astérisque on utilisera la première notation.

5.1.3 Instructions et directives

Quel que soit le langage de programmation utilisé, on trouvera toujours deux sortes d'instructions: d'une part les instructions dites exécutables qui traduisent l'algorithme et qui sont elles-mêmes traduites en langage machine, et d'autre part les énoncés destinés au programme effectuant la

traduction (compilateur, assembleur, etc…) qui ne sont pas traduits en instructions machine, par exemple les déclarations de variables en C++.

Ces énoncés sont parfois appelés instructions non exécutables; en langage d'assemblage on les appelle les *pseudo-instructions* ou les *directives*. Les directives en assembleur ont la même forme que les instructions proprement dites. Les directives ont un effet durant la phase d'assemblage, tandis que les instructions n'ont un effet qu'au cours de la phase d'exécution.

5.1.4 Format des instructions

Une instruction Assembleur occupe normalement une ligne. L'instruction Assembleur proprement dite comporte quatre champs:

```
Étiquette:      Opération   Opérandes   ; Commentaires
```

En assembleur, le format d'une instruction n'est pas fixe: les champs ne doivent pas nécessairement commencer dans une colonne donnée. Le format est libre et les champs sont identifiés par l'ordre dans lequel ils apparaissent. Les champs sont séparés par au moins une espace.

Le champ étiquette est facultatif et sert à donner un nom à l'instruction, nom qui pourra servir de repère (variable ou point de branchement). Le symbole doit être terminé par un signe deux-points. Chaque étiquette d'un programme doit être unique, sinon on crée une ambiguïté: deux adresses nécessairement différentes sont repérées par une même étiquette. L'assembleur fait la différence entre majuscule et minuscule, mais n'utilisez pas de symboles qui ne diffèrent que par la casse, par exemple `Boucle` et `boucle` qui sont deux symboles distincts, que le programmeur ne différencie pas nécessairement lors de la lecture.

Le champ opération est celui qui indique l'opération à réaliser; il peut contenir une instruction ou une directive. Dans ce champ on doit utiliser les symboles instructions prédéfinis, comme par exemple `LDA`, `ADDX` , `SUBA`, `STA`, `LDX`. On peut également y utiliser des directives comme nous en avons déjà vues: `.BLOCK` pour réserver de la place en mémoire pour les variables, `.WORD` pour définir des constantes ou des variables possédant une valeur initiale, `.END` pour indiquer la fin physique du programme. Le champ opération est obligatoire et est terminé par une espace. En l'absence du champ étiquette, le champ opération peut commencer à la première colonne (chose à éviter pour des raisons de présentation).

Le champ opérande se compose d'un ou de deux sous-champs séparés par des virgules. Ce champ doit être séparé du champ opération par au moins une espace. Le nombre et la nature des sous-champs dépend du champ opération; dans les exemples déjà vus nous avons eu diverses formes pour ce champ:

```
Somme,d
Boucle
500,i
```

Le champ commentaire est facultatif et permet d'inclure des commentaires explicatifs dans chaque instruction. Il commence toujours par un point-virgule et se termine avec la ligne. Ce champ n'a aucun effet sur l'exécution de l'instruction mais est néanmoins une partie importante de l'instruction: l'utilisation de commentaires est cruciale pour la compréhension facile du programme par une autre personne que l'auteur du programme. Même l'auteur trouvera les commentaires utiles lorsqu'il travaillera sur un gros programme ou sur un ancien programme qu'il n'a pas revu depuis longtemps. Les

commentaires servent à indiquer l'objectif d'une instruction; les commentaires qui ne sont qu'une répétition en français de l'instruction sont inutiles, comme par exemple:

```
LDA     Somme,d     ; placer Somme dans A
```

Dépendant du *contexte de l'application traitée*, un commentaire explicatif pourrait être:

```
LDA     Somme,d     ; utiliser la somme cumulative
```

Les bons programmeurs choisissent bien leurs commentaires; en moyenne, on devrait au moins trouver un commentaire toutes les 2 ou 3 lignes de code. Dans certaines installations on exige des programmeurs un commentaire par instruction. Si le premier caractère d'une ligne est un point-virgule toute la ligne est alors un commentaire; ceci est très utile pour inclure en tête de programme une description complète de la fonction du programme. On peut également, pour aérer la présentation d'un programme, utiliser le point-virgule en début de ligne pour placer une ou plusieurs lignes vides et séparer les diverses parties logiques d'un programme. On reviendra sur l'aspect documentation des programmes dans un chapitre ultérieur étant donné l'importance considérable du sujet.

Bien que le format des instructions soit libre, on utilise habituellement un format fixe qui permet d'aligner les divers champs et de rendre la lecture des programmes plus facile. Le champ étiquette commence toujours en première colonne; le champ opération commence alors en colonne 10; le champ opérandes commence en colonne 20; le champ commentaires peut commencer à différents endroits après la colonne 32. L'utilisation du caractère de tabulation permet des alignements faciles, mais la liste d'assemblage est habituellement formatée par PEP 8.

5.2 Liste d'assemblage

L'assembleur produit une liste de la traduction qu'il effectue, accompagnée de la liste des instructions du programme source. La traduction est numérique et occupe la partie gauche de la liste. Au fur et à mesure de leur traduction les instructions du programme doivent être rangées en mémoire en vue de leur exécution ultérieure. Pour permettre le rangement des instructions numériques en mémoire l'assembleur utilise un compteur ordinal dont le contenu est augmenté de la longueur en octets de l'instruction chaque fois qu'une instruction est traduite.

Avec PEP 8 la valeur de ce compteur ordinal correspondra aux adresses réellement utilisées au moment de l'exécution. Il faut cependant garder en tête que la valeur du compteur ordinal de l'assembleur indique une *adresse relative* apparaissant dans la seconde colonne de gauche de la liste du programme et qui, dans les processeurs actuels, sera toujours différente de l'adresse occupée en mémoire à l'exécution. Ensuite apparaît l'instruction traduite en code machine hexadécimal. La colonne suivante représente les instructions du programme source. La figure 5.1 présente la liste d'assemblage d'un court programme exemple, lequel ne devrait pas s'exécuter correctement à cause d'un appel à un sous-programme inexistant (cependant l'exécution semble bien se passer, car la première instruction du sous-programme, à l'adresse 1234 ou 0x4D2, est située dans la mémoire inutilisée, laquelle est mise à zéro, zéro étant le code de l'instruction `STOP`! On y remarquera que les directives ont un effet différent que les instructions: le code objet peut être absent (`.EQUATE`, `.WORD`, `.END`) ou correspondre à des constantes qui ne sont pas découpées en tranches de un ou trois octets comme les instructions.

```
Addr  Code   Symbol   Mnemon  Operand    Comment
             Saut:    .EQUATE 1234
0000  C00006 Exemple: LDA     6,i        ; Entier1 = 6
0003  E10010           STA     Entier1,d
0006  C00005           LDA     5,i        ; Entier2 = 5
0009  E10012           STA     Entier2,d
000C  1604D2           CALL    Saut
000F  00               STOP
0010  0000   Entier1:  .WORD   0
0012  0000   Entier2:  .WORD   0
                       .END

Symbol    Value
Saut       04D2
Exemple    0000
Entier1    0010
Entier2    0012

No errors. Successful assembly.
```

Figure 5.1 Liste d'assemblage

5.3 Exercices

5.3.1 Quels caractères peut-on utiliser pour délimiter les champs d'une instruction?

5.3.2 Donner deux façons de définir la constante décimale 1940.

5.3.3 Écrire les instructions Assembleur nécessaires pour calculer `X = (X+4) - (Y-12) + 3;`

5.4 Classes d'instructions

Nous avons déjà vu (dans la section 3.3) que les instructions machine de l'ordinateur PEP 8 avaient des longueurs variables de 1 octet ou de 3 octets. Le programme de la figure 5.1 a également illustré ce fait (comparez les instructions d'adresses relatives 6 et F).

Les instructions machine correspondent en fait à plusieurs formats d'instruction différents. Ces formats diffèrent essentiellement par les caractéristiques d'adressage des opérandes: opérandes situés dans les registres, opérandes situés dans l'instruction elle-même, opérandes situés en mémoire centrale adressés directement, opérandes situés en mémoire centrale adressés par registre d'index, opérandes situés sur la pile. La longueur des instructions correspondant aux divers formats peut varier à cause de la place requise pour la spécification des opérandes. Dans toutes les instructions, le premier octet contient le code opération ainsi que les numéros des registres utilisés et le mode d'adressage utilisé pour les opérandes. Ce premier octet peut être suivi d'octets d'extension correspondant à ses opérandes.

5.5 Directives

On a déjà vu que certaines instructions Assembleur n'étaient pas traduites en langage machine et avaient pour but de fournir des informations à l'assembleur: ce sont les directives. Elles n'engendrent pas d'instructions machine mais fournissent des indications d'assemblage. On peut diviser les directives en deux grandes catégories: celles qui servent à réserver de la place en mémoire et celles qui n'engendrent aucune réservation de mémoire. Les directives appartenant à la première catégorie

peuvent être comparées aux déclarations que l'on retrouve généralement au début d'un programme C++ pour les variables et les constantes. Les directives de la seconde catégorie ne servent qu'à donner des indications à l'assembleur, comme par exemple la définition de symboles externes, l'indication de la fin du programme, etc…

5.5.1 .BLOCK

La directive .BLOCK permet de réserver de la place en mémoire en initialisant le contenu à zéro. Le compte indique le nombre d'octets que l'on réserve en mémoire. La forme générale de la directive est la suivante:

```
.BLOCK      nombre d'octets
```

La valeur maximum permise est de 255; si on a besoin de plus de mémoire, il faut utiliser plusieurs directives .BLOCK.

Par exemple:
```
.BLOCK      5
```
réserve 5 octets consécutifs.

5.5.2 .WORD

La directive .WORD permet de réserver un mot en mémoire et d'en initialiser le contenu. Notez que le contenu de cette « constante » n'est pas protégé et peut être modifié. La valeur est un nombre décimal ou hexadécimal et définit le contenu initial du mot à réserver. La forme générale de la directive est la suivante:

```
.WORD       Valeur
```

Par exemple, les 2 directives:
```
.WORD       0x456A
.WORD       7
```
réservent 2 mots consécutifs (4 octets) contenant les valeurs hexadécimales 456A, et 0007.

5.5.3 .BYTE

La directive .BYTE permet de réserver un octet en mémoire et d'en initialiser le contenu. Notez que le contenu de cette « constante » n'est pas protégé et peut être modifié. Les valeurs sont des nombres ou des caractères et définissent le contenu initial de l'octet à réserver. La forme générale de la directive est la suivante:

```
.BYTE       Valeur
```

Par exemple, les 2 directives:
```
.BYTE       0x6A
.BYTE       '*'
```
réservent 2 octets consécutifs contenant les valeurs 6A, et 2A.

5.5.4 .ASCII

La directive .ASCII permet de réserver un nombre d'octets en mémoire et d'en initialiser le contenu à partir d'une chaîne de caractères. Notez que le contenu de cette chaîne n'est pas protégé et peut être modifié. La forme générale de la directive est la suivante:

```
.ASCII      "Chaîne"
```

Par exemple, les 2 directives:
```
.ASCII      "ceci est une chaîne"
```

```
                        .ASCII      " et demi\x00"
```

réservent 28 octets consécutifs contenant les deux chaînes à la suite l'une de l'autre.

Il est possible d'insérer des caractères spéciaux dans une chaîne en les préfixant de la barre oblique inversée (*backslash* `\`). Par exemple pour insérer un guillemet dans la chaîne, il faut le préfixer d'une barre oblique inversée `\"`, de même si vous voulez inclure une barre oblique inversée dans votre chaîne il faut la doubler `\\`. `\n` représente le caractère de saut de ligne. On peut inclure tout caractère en utilisant son code ASCII de deux chiffres précédé de `\x`, comme par exemple `\x0A` qui est l'équivalent de `\n`. Ainsi les chaînes :

```
                .ASCII    "Bonjour!\nComment ça va?"
```

et

```
                .ASCII    "Bonjour!\x0AComment ça va\x3F"
```

engendrent la même suite de 23 caractères.

5.5.5 .EQUATE

La directive `.EQUATE` permet de définir un symbole en le rendant équivalent à une valeur numérique. La forme générale de la directive est la suivante:

```
                        Symbole:    .EQUATE     Valeur
```

Exemple:

```
                        Limite:     .EQUATE     555
                        Bond:       .EQUATE     0x0E
```

La valeur entière, qui sera utilisée dans les instructions du programme, devrait être entre –32 768 et 32 767.

5.5.6 .ADDRSS

La directive `.ADDRSS` (notez l'orthographe!) réserve l'espace d'un mot mémoire et y place l'adresse du symbole opérande, comme par exemple :

```
                        .ADDRSS     table
```

qui place l'adresse correspondant au symbole `table` dans les deux octets réservés.

5.5.7 .BURN

La directive .BURN est particulière et ne doit être utilisée que dans les cas de modification du système d'exploitation pour initialiser une nouvelle définition de la mémoire morte (ROM). L'assembleur engendre du code machine pour les instructions qui suivent la directive, mais pas pour celles qui précèdent. Il fait l'hypothèse que les instructions sont destinées à la « mémoire morte » de PEP 8 et seront placées dans le haut de la mémoire (voir chapitre 12).

5.5.8 .END

Cette directive doit être la dernière d'un programme, car elle signale la fin de l'assemblage. Sa forme est la suivante:

```
                        .END
```