Chapitre 3
Codage de l'information

Les cellules de mémoire ne contiennent pas vraiment des valeurs décimales : elles contiennent en fait des valeurs binaires. Le programmeur en langage évolué n'a pas à s'en soucier, par contre le programmeur en langage d'assemblage doit connaître en particulier le système de numération binaire, et même le système de numération hexadécimal.

3.1 Systèmes de numération

Une valeur numérique représente une quantité qui donne généralement la mesure de quelque chose; comme le montre la figure 3.1, cette valeur numérique peut être représentée par différents symboles, selon le système de numération utilisé.

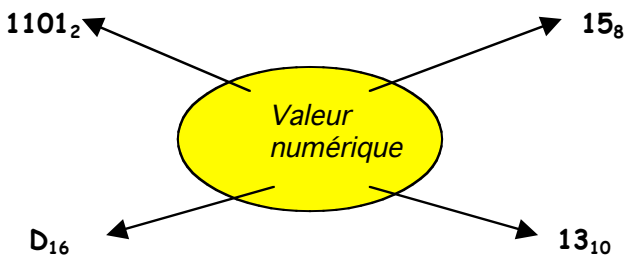

Figure 3.1 Une valeur numérique et certaines de ses représentations

Un nombre est écrit au moyen de *chiffres*[1] et représente une valeur; pour ce qui concerne le système décimal, la représentation d'un nombre et sa valeur sont étroitement associées, à cause de l'habitude que l'on a de ce système. On doit cependant se rappeler les règles permettant de calculer la valeur d'un nombre à partir de sa représentation. Ainsi, le nombre décimal représenté par 101 a pour valeur 1 centaine + 0 dizaine + 1 unité. La notation *positionnelle* met en jeu les puissances de la base du système de numération utilisé, ainsi :

$101_{10} = 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$ pour le système décimal. Dans le système binaire (base 2) le nombre 101 a pour valeur $101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ soit 5 dans le système décimal.

D'une façon générale, le symbole : $d_n d_{n-1}$ .....$d_2 d_1 d_0$

représente dans le système de numération en base b, la valeur obtenue par l'expansion polynomiale :

$$d_n b^n + d_{n-1} b^{n-1} + \ldots\ldots\ldots\ldots + d_2 b^2 + d_1 b^1 + d_0 b^0$$

où $0 \le d_i < b$ pour $i = 0, 1, 2, \ldots\ldots\ldots\ldots, n-1, n$.

Les **chiffres** utilisés pour représenter un **nombre** dans un système de numération donné doivent être compris entre zéro et la base-1.

base 10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
base 2	0, 1

[1] Notez la différence entre nombre et chiffre et ne tombez pas dans le travers des journalistes qui parlent savamment du "chiffre" 2 347 422! Dans quelle base ?

base 16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
base 8	0, 1, 2, 3, 4, 5, 6, 7

3.1.1 Système binaire

Dans le système binaire on ne dispose que de deux chiffres pour la représentation des nombres. Pour convertir un nombre binaire en son équivalent décimal, on applique l'expansion polynomiale vue plus haut.

$1101_2 = 2^3 + 2^2 + 2^0 = 13_{10}$

$101110_2 = 2^5 + 2^3 + 2^2 + 2^1 = 46_{10}$

$111111_2 = 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 63_{10}$

$1100011001010111000_2 = 2^{18} + 2^{17} + 2^{13} + 2^{12} + 2^9 + 2^7 + 2^5 + 2^4 + 2^3$

$= 262144 + 131072 + 8192 + 4096 + 512 + 128 + 32 + 16 + 8$

$= 406200_{10}$

Inversement pour convertir un nombre décimal en son équivalent binaire on doit procéder par une suite de divisions par deux en conservant le reste de chaque division, ces restes constituant les chiffres du nombre converti (pris en ordre inverse).

Exemple 1:

Nombre	Base	Reste	
19_{10}	2	1	^
9	2	1	|
4	2	0	|
2	2	0	|
1	2	1	|
0			donc $19_{10} = 10011_2$

Exemple 2 :

Nombre	Base	Reste	
139_{10}	2	1	^
69	2	1	|
34	2	0	|
17	2	1	|
8	2	0	|
4	2	0	|
2	2	0	|
1	2	1	|
0			donc $139_{10} = 10001011_2$

Exemple 3:

Nombre	Base	Reste	
52_{10}	2	0	^
26	2	0	|
13	2	1	|
6	2	0	|
3	2	1	|
1	2	1	|
0			donc $52_{10} = 110100_2$

3.1.2 Parties fractionnaires d'un nombre réel

Les exemples que nous venons de voir n'ont touché que les nombres entiers. Bien souvent, nous devons manipuler des valeurs possédant une partie fractionnaire non nulle, des valeurs dites réelles (voir chapitre 11). Dans ce cas, nous utilisons toujours le système de numération présenté en y ajoutant les puissances négatives de la base du système. Ainsi le nombre décimal 1517,1536 a pour valeur
$1 \times 10^3 + 5 \times 10^2 + 1 \times 10^1 + 7 \times 10^0 + 1 \times 10^{-1} + 5 \times 10^{-2} + 3 \times 10^{-3} + 6 \times 10^{-4}$
La partie fractionnaire est calculée en utilisant des puissances négatives de la base, c'est-à-dire dixièmes, centièmes, millièmes, etc.

De même, en binaire le nombre 1101,1011 a pour valeur :
$2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-3} + 2^{-4}$ soit : 13,6875

La conversion d'un nombre réel binaire en nombre réel décimal n'est pas difficile à faire si on dispose d'une table des puissances de deux (il y en a une à l'annexe A). Par contre, la conversion de valeurs décimales en valeurs binaires demande un peu plus d'effort. Cependant, il existe une méthode simple de conversion des parties fractionnaires de nombres décimaux. La conversion de la partie entière du nombre décimal se fait de la façon indiquée plus haut. La conversion de la partie fractionnaire utilise une série de multiplications par deux, tel que l'illustre l'exemple de conversion de décimal à binaire suivant. Soit à convertir le nombre 4,8765. La partie entière donne 100. La partie fractionnaire est placée dans la colonne de droite de deux colonnes, puis successivement multipliée par deux, en conservant la partie entière de chaque résultat de multiplication.

4	,8765
1	,7530
1	,5060
1	,0120
0	,0240
0	,0480
0	,0960
0	,1920
0	,3840
0	,7680
1	,5360
1	,0720
0	,1440
0	,2880
0	,5760
1	,1520
0	,3040
0	,6080
1	,2160
0	,4320

En nous arrêtant arbitrairement à cette étape du calcul, la partie fractionnaire binaire est alors : ,1110000001100010010. Il est bien entendu, possible de continuer le calcul tant que la partie

fractionnaire calculée dans la colonne de droite n'est pas nulle. En notant que la conversion n'est pas terminée et que le résultat est, par conséquent, tronqué, le nombre décimal converti est donc :

$$4,8765_{10} = 100,1110000001100010010_2.$$

Lorsqu'on parle de conversion de nombres réels d'un système de numération à un autre il est bon de retenir une chose : des nombres qui possèdent une représentation finie dans un système de numération donné peuvent avoir une représentation infinie (ou cyclique) dans un autre système de numération. Prenons l'exemple simple de la valeur un dixième dans le système décimal. Si nous la convertissons au système binaire, nous obtenons :

0	,1
0	,2
0	,4
0	,8
1	,6
1	,2
0	,4
0	,8
1	,6
1	,2
0	,4
0	,8
1	,6
1	,2
0	,4
0	,8
1	,6
1	,2
0	,4
0	,8

La valeur binaire correspondant à la valeur un dixième est donc cyclique et infinie.

$$0,1_{10} = 0,00011001100110011001100110011\ldots_2$$

3.1.3 Système hexadécimal

Au lieu d'utiliser directement le système binaire, les programmeurs ont utilisé soit le système octal, soit le système hexadécimal qui sont plus pratiques pour eux et qui leur permettent de passer au système binaire de façon très simple. Ils ont utilisé le système octal lorsque leur ordinateur possèdait des cellules de mémoire dont la taille était un multiple de trois, et utilisent le système hexadécimal lorsque la taille des cellules de mémoire est un multiple de quatre, comme de nos jours.

Dans le cas des processeurs modernes, la taille des cellules de mémoire est un multiple de quatre, on utilise donc le système hexadécimal. On définit d'abord une table d'équivalence binaire-hexadécimal pour les 16 **chiffres** hexadécimaux (table 3.1).

Binaire	Hexadécimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Table 3.1 Équivalence binaire-hexadécimal

Pour convertir un nombre binaire en un nombre hexadécimal équivalent il suffit de grouper les chiffres binaires (bits) par quatre en partant de la droite et de remplacer chaque groupe par son équivalent hexadécimal tel que défini dans la table 3.1.

Par exemple 100010_2 => 0010 0010 = 22_{16}

10101011_2 => 1010 1011 => AB_{16}

100011010010111_2 => 0100 0110 1001 0111 => 4697_{16}.

Inversement, pour convertir un nombre hexadécimal en son équivalent binaire, il suffit de remplacer chaque chiffre hexadécimal par son équivalent binaire sur quatre bits.

Exemple: $12F_{16}$ => 0001 0010 1111=> 100101111_2

$2A03_{16}$ => 0010 1010 0000 0011 => 10101000000011_2

Un nombre de 16 bits sera représenté par un nombre hexadécimal de 4 chiffres; la notation hexadécimale est plus proche des notations qui nous sont habituelles que la notation binaire, et, de ce fait, les erreurs se font moins facilement en hexadécimal qu'en binaire.

Pour convertir des nombres du système hexadécimal au système décimal, on utilise l'expansion polynomiale vue plus haut.

$27A_{16} = 2 \times 16^2 + 7 \times 16^1 + 10 \times 16^0 = 2 \times 256 + 7 \times 16 + 10 = 634_{10}$

$43227_{16} = 4 \times 16^4 + 3 \times 16^3 + 2 \times 16^2 + 2 \times 16^1 + 7 \times 16^0 = 262144 + 12288 + 512 + 32 + 7$

$= 274983_{10}$

Cette méthode est utilisée lors des conversions de binaire à décimal où l'on passe par la représentation hexadécimale de façon intermédiaire

$10111110_2 = BE_{16} = 11 \times 16^1 + 14 \times 16^0 = 190_{10}$

Inversement pour convertir un nombre décimal en un nombre hexadécimal on utilise une méthode semblable à celle utilisée pour passer du décimal au binaire. On fait une série de divisions par **16** en conservant le reste de chaque division.

Exemples:

	Base	Reste	
412_{10}	16	12	^
25	16	9	|
1	16	1	|
0			$19C_{16}$ => 110011100_2

	Base	Reste	
1492_{10}	16	4	^
93	16	13	|
5	16	5	|
0			$5D4_{16}$ =>10111010100_2

	Base	Reste	
53290_{10}	16	10	^
3330	16	2	|
208	16	0	|
13	16	13	|
0			$D02A_{16}$ =>1101000000101010_2

Cette conversion est souvent une étape intermédiaire au cours de la conversion de décimal à binaire.
La notation positionnelle s'applique de la même façon aux parties fractionnaires des nombres où chaque chiffre doit être multiplié par une puissance *négative* de la base, comme on l'a vu plus haut.

Par exemple: $0{,}875_{10} = 8 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$

$0{,}1A2_{16} = 1 \times 16^{-1} + 10 \times 16^{-2} + 2 \times 16^{-3} = 0{,}10205078125$

3.1.4 Exercices

3.1.4.1 Convertir les nombres décimaux suivants en leurs équivalents binaires:

152 127 256 90 33 65

3.1.4.2 Convertir les nombres binaires ci-dessous en leurs équivalents décimaux.

1100110101 11101110111 100010001000 111111111111 1010101010

3.1.4.3 Effectuer les additions binaires ci-dessous:

10111	110011	101010	10111001
+ 01011	+ 011100	+ 011111	+ 00011001

3.1.4.4 Effectuer les soustractions binaires ci-dessous:

100110	111111	101110	110000
- 010110	- 101010	- 011111	- 010111

3.1.4.5 Effectuer les additions hexadécimales ci-dessous:

ABCD	lEFO	AAAA	CDEF
+ 4321	+ 9042	+ OAAA	+ 1221

3.1.4.6 Effectuer les soustractions hexadécimales ci-dessous:

34A2	F975	54EA	FFFF
- 0191	- 94AE	- lEFA	- EDCB

3.2 Représentation de l'information

Les processeurs actuels sont des ordinateurs binaires: le bit est la plus petite unité d'information. Les bits sont groupés par huit en *octets* (« bytes »). Dans ces ordinateurs, l'octet (aussi appelé *caractère*) est la plus petite unité d'information adressable. Un octet comprenant 8 bits, peut comprendre 2^8 ou 256 combinaisons de bits différentes allant de 00000000 à 11111111. Un octet peut donc contenir deux chiffres hexadécimaux. Les bits d'un octet sont numérotés de droite à gauche de la position 0 à la position 7.

Les données utilisées par les programmeurs sont de natures diverses et occupent en mémoire soit des zones de longueur fixe soit des zones de longueur variable (de 1 à n octets). Les diverses instructions machine peuvent traiter des données de l'un ou l'autre type; dans le cas des zones de longueur variable il faudra cependant préciser dans l'instruction la longueur de la zone utilisée.

L'unité de base des zones de longueur fixe est le mot mémoire: pour PEP 8, un mot regroupe 2 octets, c'est-à-dire 16 bits, mais d'autres processeurs auront des mots de 32 bits, voire même de 64 bits. Les positions des bits d'un mot sont numérotées de 0 à 15 de droite à gauche (ou de 0 à 31, ou de 0 à 63). L'adresse du mot est toujours l'adresse du premier octet de la zone considérée (figure 3.2).

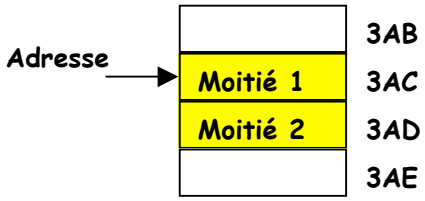

Figure 3.2 Repérage d'un mot mémoire par son premier octet

3.2.1 Représentation des nombres entiers

Un nombre entier est représenté par son équivalent binaire dans un mot de 16 bits. Le bit 15 ou bit de gauche du nombre peut être utilisé pour indiquer le signe du nombre: 0 pour plus et 1 pour moins. La figure 3.3 donne la représentation du nombre +190. On verra plus loin sur quelle base repose la représentation des nombres négatifs.

Bits	0000000010111110
Position	15 ... 43210

Figure 3.3 Représentation du nombre + 190

Les entiers pouvant être représentés dans des mots de 16 bits sont relativement petits; la valeur positive maximum qu'il est possible de représenter de cette façon est 32767; la plus grande valeur sans signe serait en fait 65535 (2^{16}-1, car 2^{16} est représenté en binaire par un chiffre 1 suivi de 16 zéros, et ne tient pas dans 16 bits). Avec 32 bits la valeur positive maximum serait 2 147 483 647.

3.2.2 Représentation des caractères

Comme nous l'avons vu l'information conservée en mémoire l'est sous forme numérique. Cependant on peut coder en mémoire deux types de données: les données numériques (telles les nombres entiers vus plus haut) et les données de type caractère. Un caractère est représenté par un code de huit bits occupant un octet. Ceci permet 256 codes différents, soit un « alphabet » de 256 caractères comprenant les 26 lettres majuscules et minuscules, les dix chiffres décimaux et des caractères spéciaux (point, virgule, signe égal, signe dollar, etc...). Le code interne de représentation des caractères est le code ASCII (« American Standard Code for Information Interchange ») que l'on trouvera à l'annexe A; ce code, qui n'utilise que 7 bits, a été étendu à 8 bits pour inclure les lettres accentuées des langages autres que l'anglais, et normalisé (ISO 8859-1).

Dans ce code, la lettre A est représentée par 01000001 (41 en hexadécimal), B par 01000010 (42_{16}), etc. Le caractère espace est représenté par la valeur hexadécimale 20, les chiffres décimaux par les valeurs hexadécimales allant de 30_{16} à 39_{16}. Avec ce code, on constate que le caractère espace précède les chiffres décimaux qui précèdent les lettres majuscules, lesquelles précèdent les lettres minuscules. La figure 3.4 donne l'aspect d'une zone de mémoire contenant la chaîne de caractères « MC68000 ».

	260	261	262	263	264	265	266	
	01001101	01000011	00110110	00111000	00110000	00110000	00110000	

Figure 3.4 Chaîne de caractères

On remarquera encore que rien ne différencie le contenu alphanumérique des octets du contenu numérique des mots: *tout dépend de l'interprétation qui en est faite*.

3.3 Langage machine

Nous avons vu que les instructions étaient codées numériquement et placées en mémoire centrale pour l'exécution du programme. Chaque instruction comprend un code opération qui indique l'opération que l'ordinateur doit exécuter. Une instruction comprend aussi des indications permettant à l'ordinateur de déterminer où se trouve le ou les opérandes de l'instruction; ces indications sont habituellement utilisées au cours du décodage de l'instruction pour calculer l'adresse effective de l'opérande. Certaines instructions comprennent également d'autres informations permettant de mieux définir l'opération à effectuer.

On peut diviser les instructions d'un ordinateur en diverses classes ou catégories: instructions arithmétiques pour les calculs, instructions logiques, instructions de transfert pour le déplacement d'informations, instructions de contrôle du déroulement de l'exécution du programme, instructions d'entrée-sortie. Le reste de ce cours sera de fait consacré à l'étude des diverses instructions disponibles et de leurs utilisations.

Pour donner une meilleure idée du langage machine nous allons prendre un exemple de programme très simple, qui nous permettra d'illustrer les instructions machine et la façon dont elles sont exécutées. La figure 3.5 représente une partie de programme en langage machine pour le processeur PEP 8.

On remarque sur cette figure que les instructions machine sont numériques (et données en hexadécimal) et qu'elles occupent trois octets de mémoire. Elles sont exécutées de façon séquentielle, c'est-à-dire les unes après les autres, sauf si une instruction cause un transfert de contrôle dans une autre partie du programme.

Supposons qu'avant l'exécution du programme de la figure 3.5, le contenu des registres utilisés et de certaines cellules de mémoire soit celui indiqué par la figure 3.6 (toujours en hexadécimal).

Adresse	**Instruction**
2100	C1212C
2103	B001F4
2106	0E211E
2109	C1212A
210C	E12128
210F	C1212C
2112	E1212A
2115	712128
2118	E1212C
211B	042103

Figure 3.5 Langage machine

L'exécution de notre partie de programme commencera par l'instruction située dans le mot d'adresse 2100; le code opération (repéré dans ce cas par les quatre premiers bits [1100]) indique une copie d'information, le bit suivant indique l'accumulateur [0] et les trois bits suivants [001] indiquent un mode d'adressage direct. C'est une instruction avec 2 opérandes, l'opérande destination est le registre accumulateur [0] et l'opérande source est repéré par les 16 bits suivants [0010000100101100] qui indiquent une adresse mémoire (hexadécimal 212C).

Registre	**Contenu**	**Adresse**	**Contenu**
A	02F8	212C	0001

Figure 3.6 Contenus avant exécution

L'exécution de l'instruction provoque la copie de la valeur 0001 de la variable située à l'adresse hexadécimale 212C en mémoire dans l'accumulateur. Cette instruction et ses opérandes occupent trois octets; l'instruction suivante est donc prise à l'adresse 2103. Il s'agit d'une instruction de comparaison de l'accumulateur avec la valeur immédiate 500 ($1F4_{16}$).

L'instruction suivante est prise à l'adresse 2106 puisque l'instruction de comparaison occupait trois octets. Le code opération de 7 bits (0000111) indique qu'il s'agit d'une instruction de branchement conditionnel. La condition est GE et le branchement se fait à un octet situé à l'adresse 211E (adresse qui suit le dernier octet de notre figure 3.5).

Nous avons ensuite une série de quatre instructions de déplacement d'information semblables à la première instruction.
2109 copie du contenu de l'adresse mémoire 212A dans le registre A
210C copie du contenu du registre A dans le mot mémoire d'adresse 2128

210F copie du contenu de l'adresse mémoire 212C dans le registre A
2112 copie du contenu du registre A dans le mot mémoire d'adresse 212A.

À l'adresse 2115 nous trouvons une instruction d'addition (code 0111) du mot mémoire d'adresse 2128 et du registre A. L'instruction suivante à l'adresse 2118 est une autre instruction de rangement d'information qui copie le contenu du registre A dans le mot mémoire d'adresse 212C.

Après exécution de ces 5 instructions les contenus des registres et de la mémoire utilisés sont tels que le montre la figure 3.7

Registre	Contenu	Adresses	Contenus
A	00000002	2128	0001
		212A	0001
		212C	0002

Figure 3.7 Contenus après exécution

L'instruction qui suit à l'adresse 211B est une instruction de branchement inconditionnel (code 0000010) à une adresse égale à 2103, c'est-à-dire l'adresse de la 2ème instruction de la figure 3.5.

L'exemple que nous venons de voir nous permet de constater que le langage machine, comme son nom l'indique, est fait pour l'ordinateur plutôt que pour nous. L'ordinateur est en effet une machine construite pour manipuler des quantités numériques. Si le langage machine présenté semble difficile à assimiler, encore faut-il se rappeler que le langage machine véritable utilise le système de numération binaire! Nous allons voir qu'il est possible d'utiliser une forme symbolique du langage machine qui nous sera bien plus pratique. On peut cependant retenir de l'exemple présenté, la façon répétitive dont les instructions machine sont exécutées.

3.4 Langage d'assemblage

L'exemple précédent a montré que pour écrire des programmes en langage machine on devait savoir exactement quelles étaient les adresses utilisées pour les instructions et les données, utiliser des tables donnant les divers codes opération et exprimer les constantes en binaire (ou en leur équivalent hexadécimal). Cette façon de programmer, si elle est adaptée à l'ordinateur, n'est pas adaptée au fonctionnement normal des programmeurs humains. Même si elle fut utilisée, faute de mieux, dans les premières années de l'informatique jusqu'au début des années cinquante du siècle dernier, elle fut abandonnée parce qu'elle était trop inefficace. Elle fut remplacée par la programmation en langage d'assemblage.

Un langage d'assemblage est une forme symbolique du langage machine; alors que le langage machine est numérique, le langage d'assemblage permet l'utilisation de noms alphabétiques pour les codes opérations et les opérandes en mémoire. Un programme appelé *assembleur* traduit un programme écrit en langage d'assemblage en son équivalent en langage machine qui peut alors être exécuté par l'ordinateur.

Comme tout langage de programmation, le langage d'assemblage possède une syntaxe qui doit être respectée lors de l'écriture des instructions. En général une instruction d'un programme en langage

d'assemblage correspond à une instruction du langage machine; la traduction est par conséquent simple et conduit la plupart du temps à remplacer une instruction en langage d'assemblage par l'instruction équivalente en langage machine. Une instruction en langage d'assemblage possède habituellement quatre champs: étiquette, opération, opérandes et commentaires. La figure 3.8 donne un exemple simple permettant d'illustrer le concept de langage d'assemblage en PEP 8. Pour en réduire la nouveauté nous avons repris l'exemple vu à la section 3.3.

```
Fibo:   LDA     somme,d    ;
Boucle: CPA     500,i      ; while(somme < 500)
        BRGE    Affiche    ; {
        LDA     dernier,d  ;
        STA     avant,d    ;   avant = dernier;
        LDA     somme,d    ;
        STA     dernier,d  ;   dernier = somme;
        ADDA    avant,d    ;
        STA     somme,d    ;   somme = avant + dernier;
        BR      Boucle     ; }//while
```

Figure 3.8 Instructions en langage d'assemblage PEP 8

Ces instructions effectuent les opérations qui ont été décrites plus haut et qui correspondent à une partie de l'algorithme de la figure 1.1. On y distingue en particulier l'utilisation du registre A et des variables `avant`, `dernier` et `somme` situées en mémoire.

3.5 Traitement d'un programme en langage d'assemblage

Un programme écrit en langage d'assemblage doit subir un certain nombre de transformations avant de pouvoir être exécuté. La figure 3.9 présente les différentes étapes du traitement d'un programme en langage d'assemblage.

La première phase de traitement est l'assemblage que nous allons voir ci-dessous. Le programme source est traduit en langage machine et le programme objet ainsi obtenu est rangé dans un fichier; une liste d'assemblage peut également être produite. La seconde phase de traitement est la phase de chargement et d'édition de liens; le programme objet est placé en mémoire centrale, les divers modules nécessaires à son exécution sont également chargés en mémoire et les liaisons entre les divers modules sont établies. La troisième phase du traitement est l'exécution du programme proprement dite qui peut utiliser des données et qui produit des résultats.

3.6 Assemblage

Pour illustrer la phase d'assemblage, reprenons ici le morceau de programme de la figure 3.8 afin de montrer ce que l'assembleur en fait. Il nous faut faire quelques hypothèses sur le reste du programme. L'assembleur assigne aux instructions qu'il traduit des adresses dites *relatives* car elles sont attribuées de façon séquentielle à partir du début du programme. Si la première instruction de notre exemple

```
LDA     somme,d
```

n'était pas la première instruction du programme, l'assembleur lui attribuerait une adresse suivant en séquence les adresses des instructions qui précèdent.

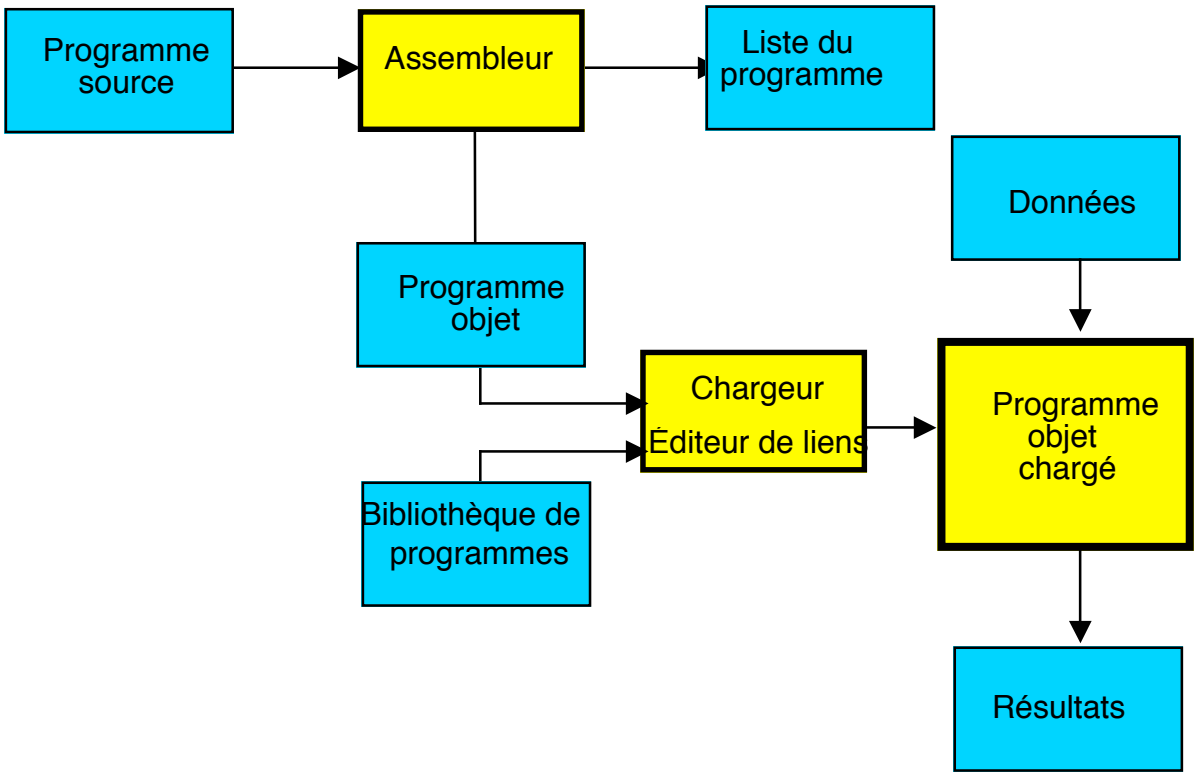

Figure 3.9 Traitement d'un programme

La figure 3.10 donne la liste d'assemblage des instructions de la figure 3.8 produite par le système PEP 8 (ftp://ftp.pepperdine.edu/pub/compsci/pep8): à la gauche des instructions symboliques se trouvent les instructions équivalentes en langage machine. La première instruction de notre exemple est en fait la première instruction du programme et se voit donc attribuer l'adresse relative zéro.

```
0000  C1002C Fibo:    LDA     somme,d    ;
0003  B001F4 Boucle:  CPA     500,i      ; while(somme < 500)
0006  0E001E          BRGE    Affiche    ; {
0009  C1002A          LDA     dernier,d  ;
000C  E10028          STA     avant,d    ;   avant = dernier;
000F  C1002C          LDA     somme,d    ;
0012  E1002A          STA     dernier,d  ;   dernier = somme;
0015  710028          ADDA    avant,d    ;
0018  E1002C          STA     somme,d    ;   somme = avant + dernier;
001B  040003          BR      Boucle     ; }//while
```

Figure 3.10 Liste d'assemblage PEP 8

La colonne d'extrême gauche donne les adresses relatives des instructions en hexadécimal. Celles-ci sont suivies de la représentation hexadécimale des instructions machine, découpée en groupes de trois octets. Les colonnes suivantes reproduisent les instructions symboliques. Comme il s'agit du même exemple que celui de la figure 3.5 on remarquera que les instructions machine produites sont les mêmes, à part les adresses mémoire des opérandes.

3.7 Chargement et exécution

On ne peut exécuter un programme que s'il se trouve en mémoire centrale. Comme on l'a vu, l'assembleur range le programme objet en mémoire secondaire; avant de pouvoir exécuter ce programme objet il faut le charger en mémoire centrale. Chaque instruction du programme objet possède une adresse relative qui lui a été attribuée par l'assembleur; il serait simple de placer le programme objet en mémoire de façon à ce que les adresses relatives correspondent aux adresses réelles des instructions (appelées *adresses absolues*). Dans la réalité, ceci n'est pas possible, car une partie de la mémoire centrale est occupée en permanence par le système d'exploitation; le programme objet ne pourra alors être placé en mémoire à partir de l'adresse zéro. Si la zone de mémoire disponible pour le programme commence à l'adresse physique 1320, la première instruction du programme (d'adresse relative zéro) sera placée à l'adresse 1320 et les instructions suivantes aux adresses qui suivent. La figure 3.11 illustre ce processus de chargement du programme objet. On remarque que toutes les adresses relatives sont modifiées par addition de l'adresse de chargement 1320: on dit que le chargeur a *translaté* le programme objet.

De la même façon les variables avant, dernier et somme occuperont des adresses absolues différentes de 28, 2A ou 2C. On remarquera que les instructions machine sont restées inchangées par la translation à l'exception de la partie identifiant les opérandes qui se trouvent en mémoire centrale. Notez que la régularité de la figure n'est due qu'au fait que les instructions considérées dans notre exemple occupent *toutes* trois octets; il y aurait pu y avoir des instructions n'occupant qu'un seul octet, ce qui aurait modifié la progression des adresses. Si l'on compare la figure 3.11 à la figure 3.5 on remarque une différence au niveau des adresses physiques des instructions chargées et des opérandes: la figure 3.5 supposait un chargement à partir de l'adresse absolue 2100 et non 1320.

Une fois le programme ainsi chargé en mémoire centrale, il peut être exécuté. Son exécution sera faite de la même façon que ce qui a été vu à la section 3.3, la seule différence étant que les instructions et les données occupent des adresses absolues différentes; les résultats seront identiques, à la position en mémoire près.

À l'uri http://www.info.uqam.ca/Members/gabrini_p/PDP11 se trouvent quatre petits films illustrant le fonctionnement du mini ordinateur DEC PDP 11, montrant très clairement les exigences matérielles des trois phases décrites ci-dessus.

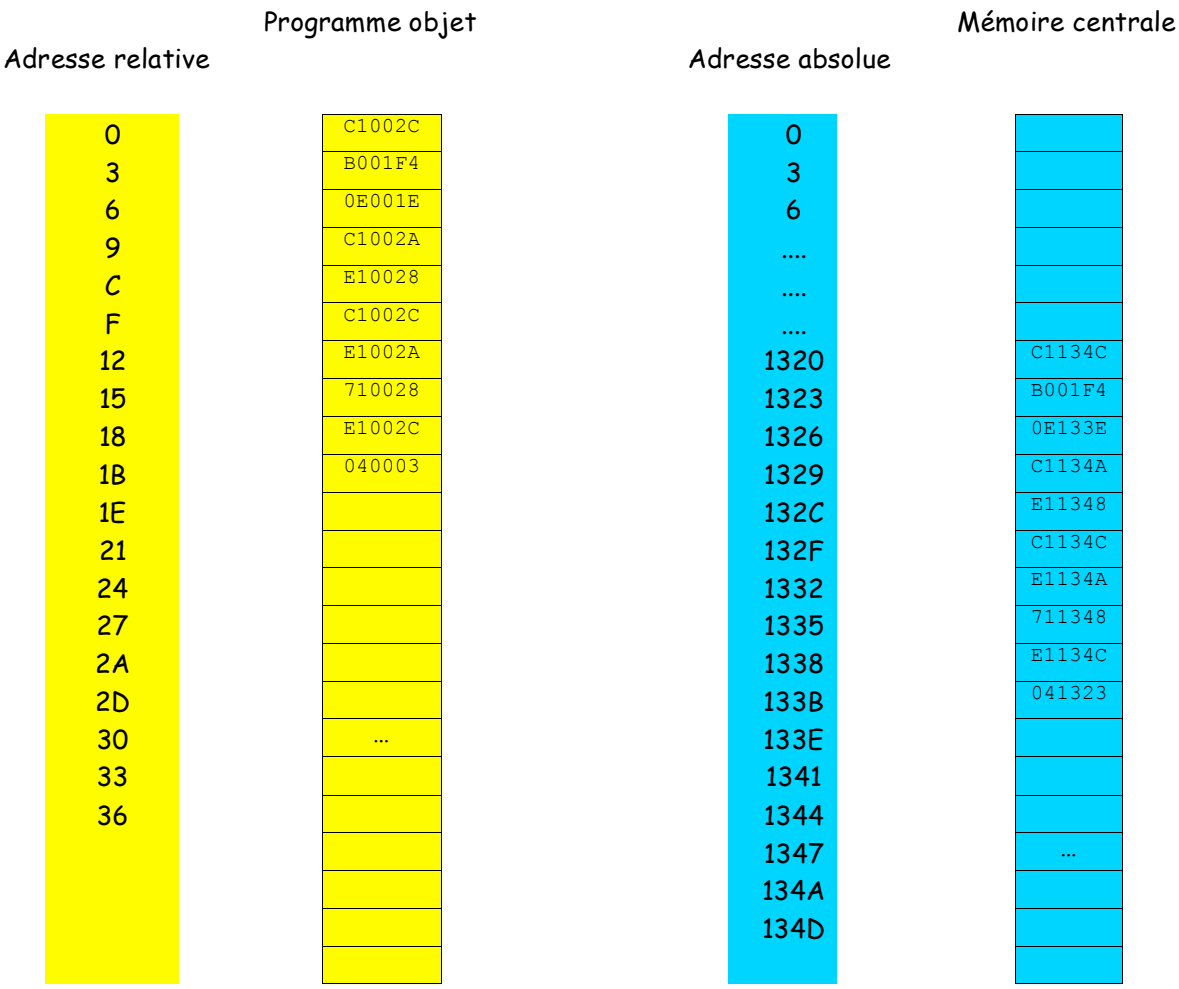

Figure 3.11 Chargement du programme

3.8 Exercices

3.8.1 Soit trois octets de mémoire dont le contenu binaire est le suivant:

001110010000000000101100

Ce contenu représente-t-il un nombre entier; si oui, lequel?
Ce contenu représente-t-il un ensemble de caractères; si oui lesquels?
Ce contenu représente-t-il une instruction; si oui laquelle?
(Utilisez les annexes: tables des puissances de 2, des codes caractères ASCII ou des codes opérations).

3.8.2 Toutes les instructions de l'exemple de la figure 3.5 ont des opérandes repérés par des adresses mémoire à l'exception d'un seul. Identifiez l'instruction qui le comprend et essayez d'expliquer la différence.

3.8.3 Une phase du chargement d'un programme en mémoire est appelée « édition de liens ». Pouvez-vous expliquer ce terme de façon intuitive?