Chapitre 7
Instructions du processeur PEP 8

Afin de mieux comprendre le fonctionnement du processeur PEP 8, nous allons passer en revue les instructions et donner pour chacune une description détaillée de son fonctionnement.

7.1 Format des instructions

Chaque processeur possède un ensemble d'instructions qui lui est propre; mais il existe des familles de processeurs qui possèdent des ensembles d'instructions semblables. Le processeur PEP 8, quant à lui, a 39 instructions dans son ensemble d'instructions. Chaque instruction possède un code instruction d'un octet; certaines instructions ne comportent que cette partie, on les appelle des instructions *unaires* (pour un seul octet). Cependant, la majorité des instructions comprennent trois octets, c'est-à-dire le code instruction sur un octet, suivi immédiatement de la spécification de l'opérande sur deux octets.

Code instruction
L'octet comprenant le code instruction peut avoir plusieurs parties, la première étant le code opération (voir Annexe C). Dans PEP 8, ce code opération a une longueur variable et peut comporter de huit à quatre bits. Par exemple, le code opération de l'instruction `MOVFLGA` comprend les huit bits 00000011. À l'inverse, l'instruction `ADDr` possède un code opération de quatre bits 0111, le reste de l'instruction comprenant des bits permettant de spécifier le registre utilisé (1 bit repéré par r) et le mode d'adressage de l'opérande (3 bits repérés par aaa). D'autres instructions, comme l'instruction de retour d'un sous-programme (voir chapitre 9), ont en plus du code opération un champ de trois bits repéré par le code nnn. La figure 7.1 illustre la signification des champs aaa et a pour l'adressage et r pour le registre. Le champ nn ou nnn n'apparaît que dans deux instructions (`NOPn` et `RETn`) et représente un nombre.

aaa	Mode d'adressage aaa	a	Mode d'adressage a	r	Registre r
000	Immédiat	0	Immédiat	0	Accumulateur A
001	Direct	1	Indexé	1	Registre index X
010	Indirect				
011	Pile				
100	Pile indirect				
101	Indexé				
110	Indexé pile				
111	Indexé pile indirect				

Figure 7.1 Champs de spécification des instructions

Spécification de l'opérande
Pour les registres, 0 indique l'accumulateur et 1 le registre d'index. Pour le mode d'adressage repéré par un seul bit, que l'on retrouve dans les instructions de branchement conditionnel, il n'y a que deux possibilités. Pour le mode d'adressage repéré par trois bits, les huit modes d'adressage sont possibles et repérés par les valeurs indiquées dans la figure 7.1.

Selon le mode d'adressage, la spécification d'opérande peut être soit l'opérande lui-même en mode d'adressage immédiat, soit l'adresse de l'opérande en mode direct, soit l'adresse de l'adresse de l'opérande en mode indirect, soit l'adresse d'un tableau qui sera indexé par le registre X en mode indexé, soit un opérande repéré sur ou par la pile (voir chapitre 9).

7.2 Instructions unaires simples

Nous plaçons dans cette catégorie toutes les instructions dont le code opération occupe huit bits et qui n'ont pas d'opérande. Ces instructions n'affectent pas le code de condition. Il y en a quatre.

Instruction `STOP`
L'instruction `STOP` a un code opération nul. Lorsque cette instruction est exécutée, il y a arrêt de l'exécution du programme et retour au système qui l'avait lancée, dans le cas de PEP 8 au simulateur, mais dans le cas d'un processeur réel au système d'exploitation.

Instruction `RETTR`
Cette instruction indique un retour en fin de traitement d'une interruption (*return from trap*); le contexte du programme interrompu est rétabli et le programme interrompu reprend (voir chapitre 12).

Instruction `MOVSPA`
Cette instruction copie la valeur du pointeur de pile (*stack pointer SP*) dans le registre A (voir chapitre 9).

Instruction `MOVFLGA`
Cette instruction copie la valeur des codes de condition NZVC dans le registre A, les 4 bits étant précédés de 12 bits nuls.
Par exemple :

```
       LDA  0xC234,i ; A = C234
       MOVFLGA       ; A = 0008
```

7.3 Instructions de chargement et rangement

Les instructions de chargement et de rangement permettent de manipuler le contenu des registres.

Instruction `LDr`
Cette instruction charge le registre d'une valeur occupant 16 bits, soit située dans l'instruction, soit située en mémoire. Les codes de condition N et Z sont affectés par cette opération : N est mis à 1 si le résultat est négatif, sinon il est mis à zéro. Z est mis à 1 si les seize bits sont nuls, sinon il est mis à zéro.

Pour donner une meilleure idée de la façon dont les instructions sont exécutées, prenons l'instruction `LDA valeur,d` et suivons son exécution pas à pas en supposant que `valeur` soit une variable de deux octets située en mémoire à l'adresse 017A et qui contient 8AB2. Dans le cycle de von Neumann, le premier octet de l'instruction est placé dans le début du registre instruction et le décodage commence; on reconnaît une instruction de chargement de l'accumulateur avec un mode d'adressage direct; les deux octets suivants sont obtenus de la mémoire en plaçant l'adresse du premier octet dans le registre

adresse du bus d'adresse et, une fois l'accès mémoire effectué, en récupérant la valeur correspondante, 017A, dans le registre du bus de données. Comme il s'agit d'un adressage direct, on place la valeur obtenue dans le registre adresse du bus d'adresse et, après lecture de la mémoire, on trouve l'opérande du registre dans le bus de données, 8AB2, et on le place dans le registre A. Le code N est mis à 1 (valeur négative) et le code Z à zéro (valeur non nulle).

Instruction `STr`
L'instruction de rangement place deux octets en mémoire à partir du registre spécifié. Les codes de condition ne sont pas affectés par l'exécution de cette instruction.

Supposons que nous exécutions l'instruction `STX result,d`, que le registre X contienne 124C, et que l'adresse de `result` soit 3C12. Une fois l'instruction décodée, on place le contenu du registre dans le registre du bus de données, on place ensuite l'adresse 3C12 dans le registre du bus d'adresse et le contenu de la mémoire est alors modifié, de telle sorte que l'octet d'adresse 3C12 contienne la valeur 12 et que l'octet d'adresse 3C13 contienne la valeur 4C.

Instruction `LDBYTEr`
Cette instruction charge la moitié droite du registre d'une valeur de 8 bits, laquelle est soit située dans l'instruction, soit située en mémoire. Faites attention que la moitié gauche du registre demeure *inchangée*. Les codes de condition N et Z sont affectés par cette opération : N est mis à 1 si le contenu du registre est négatif, sinon il est mis à zéro. Z est mis à 1 si les seize bits sont nuls, sinon il est mis à zéro.
Par exemple, si le registre A contient 1234, l'exécution de l'instruction `LDBYTEA c'*',i` modifie le contenu en 122A et N et Z sont tous les deux mis à zéro.

Instruction `STBYTEr`
L'instruction de rangement place un octet en mémoire à partir de la moitié droite du registre spécifié. Les codes de condition ne sont pas affectés par l'exécution de cette instruction.

7.4 Instructions arithmétiques

Ces instructions modifient le contenu d'un registre ou du pointeur de pile par addition ou soustraction.

Instruction `ADDr`
Cette instruction obtient l'opérande et l'additionne au contenu du registre qui est le seul modifié. Les codes de condition N, Z, V et C sont affectés par cette opération : N est mis à 1 si le contenu du registre est négatif, sinon il est mis à zéro. Z est mis à 1 si les seize bits sont nuls, sinon il est mis à zéro. V est mis à 1 s'il y a débordement, sinon à zéro. C est mis à 1 s'il y a retenue, sinon à zéro.

Instruction `SUBr`
Cette instruction obtient l'opérande et le soustrait du contenu du registre qui est le seul modifié. Les codes de condition N, Z, V et C sont affectés par cette opération : N est mis à 1 si le contenu du registre est négatif, sinon il est mis à zéro. Z est mis à 1 si les seize bits sont nuls, sinon il est mis à zéro. V est mis à 1 s'il y a débordement, sinon à zéro. C est mis à 1 s'il y a retenue, sinon à zéro.

Instruction NEGr

Cette instruction unaire (sans opérande) interprète le contenu du registre comme un entier en complément à deux et en fait la négation. Les codes de condition N, Z et V sont affectés. Le code de condition V n'est touché que si la valeur originale est –32768, puisqu'il n'existe pas de valeur positive 32768 dans l'arithmétique en complément à deux sur 16 bits.

Instruction ADDSP

Cette instruction ajoute une valeur au contenu du pointeur de pile; si la valeur est positive, le sommet de la pile est déplacé vers le bas, ce qui correspond à un désempilage. Si la valeur est négative il s'agit d'un empilage. Les codes de condition N, Z, V et C sont affectés par cette opération : N est mis à 1 si le contenu de SP est négatif, sinon il est mis à zéro. Z est mis à 1 si les seize bits sont nuls, sinon il est mis à zéro. V est mis à 1 s'il y a débordement, sinon à zéro. C est mis à 1 s'il y a retenue, sinon à zéro.

Instruction SUBSP

Cette instruction soustrait une valeur au contenu du pointeur de pile; si la valeur est positive, le sommet de la pile est déplacé vers le haut, ce qui correspond à un empilage. Si la valeur est négative il s'agit d'un désempilage. Les codes de condition N, Z, V et C sont affectés par cette opération : N est mis à 1 si le contenu de SP est négatif, sinon il est mis à zéro. Z est mis à 1 si les seize bits sont nuls, sinon il est mis à zéro. V est mis à 1 s'il y a débordement, sinon à zéro. C est mis à 1 s'il y a retenue, sinon à zéro.

Exemple

L'exemple suivant est un petit programme calculant le résultat de la multiplication de deux nombres entiers. La méthode employée est simple, puisqu'elle effectue la multiplication par additions successives. Le programme lit deux valeurs entières et affiche le début du message de sortie du résultat. Si le deuxième nombre lu (multiplicateur), placé dans le registre X, est négatif, les deux valeurs sont changées de signes, de sorte que le multiplicateur, utilisé pour compter les additions, soit positif, et que le résultat obtenu ait le bon signe. Les additions du premier nombre (multiplicande) sont répétées dans le registre A, tant que le multiplicateur n'atteint pas la valeur zéro. Le résultat est ensuite enregistré et affiché. Par exemple, avec les valeurs d'entrée suivantes : **-12 6**, la sortie indique : **-12*6=-72**

```
; Multiplication de deux nombres entiers par la méthode simple.
;       Lorne H. Bouchard  (adapté par Ph. Gabrini Mars 2006)
;
debut:   DECI    nb1,d     ; cin >> nb1;
         DECI    nb2,d     ; cin >> nb2;
         DECO    nb1, d    ; cout << nb1
         CHARO   '*',i     ;      << '*'
         DECO    nb2,d     ;      << nb2
         CHARO   '=',i     ;      << '=';
         LDX     nb2,d     ; X = nb2
         BRGE    commence  ; if(nb2 < 0){
         LDA     nb1,d     ;
         NEGA              ;
         STA     nb1,d     ;   A = nb1 = -nb1;
         LDX     nb2,d     ;
         NEGX              ;   X = nb2 = -nb2;
         STX     nb2,d     ; }
```



```
commence:LDA     0,i        ; A = 0;
addition:ADDA    nb1,d      ; do{ A += nb1;
         SUBX    1,i        ;     X--;
         BRNE    addition   ; } while(X != 0);
fini:    STA     res,d      ; res = A;
         DECO    res,d      ; cout << res;
         STOP               ;
nb1:     .WORD   0
nb2:     .WORD   0
res:     .WORD   0
         .END
```

7.5 Instructions d'entrée-sortie

Normalement au niveau de l'assembleur, il n'y a pas d'instructions d'entrée-sortie. Pour réaliser des opérations d'entrée-sortie, il faut alors généralement utiliser le système d'interruption du processeur en cause (voir chapitre 12). PEP 8 fournit cependant deux instructions d'entrée-sortie pour les caractères. Les trois autres instructions d'entrée-sortie de PEP 8 sont réalisées au moyen du système d'interruption : en effet elles correspondent à des codes instructions non réalisés; le chapitre 12 en dira plus à ce sujet.

Instruction `CHARI`
Cette instruction lit un caractère en entrée et le range dans un octet en mémoire. Les registres et les codes de condition ne sont pas affectés. Le mode d'adressage immédiat n'est, bien entendu, pas autorisé.

Instruction `CHARO`
Cette instruction affiche un caractère sur l'organe de sortie. Les registres et les codes de condition ne sont pas affectés. Tous les modes d'adressage sont permis.

Instruction `STRO`
Cette instruction (qui engendre une interruption au niveau de la machine, mais, pour le programmeur, se comporte comme toute autre instruction Assembleur) affiche une chaîne de caractères sur l'organe de sortie. Le dernier caractère de la séquence de caractères à sortir doit être un caractère dont le code est nul. Les registres et les codes de condition ne sont pas affectés. Seuls les modes d'adressage direct, indirect et indirect pile sont permis.

Instruction `DECI`
Cette instruction (qui engendre une interruption au niveau de la machine, mais , pour le programmeur, se comporte comme toute autre instruction Assembleur) lit une valeur entière sur l'organe d'entrée. Elle permet un nombre quelconque d'espaces ou de sauts de ligne avant le début de la valeur à lire qui doit débuter par un signe plus, un signe moins ou un chiffre décimal; le reste de la valeur est fait de chiffres décimaux. Les codes de condition Z et N sont affectés; si la valeur est trop grande le code de condition V prend la valeur 1. Le code de condition C n'est pas affecté. Le mode d'adressage immédiat n'est, bien entendu, pas autorisé.

Instruction DECO

Cette instruction (qui engendre une interruption au niveau de la machine, mais , pour le programmeur, se comporte comme toute autre instruction Assembleur) affiche une valeur décimale sur l'organe de sortie. La valeur est précédée d'un signe moins si elle est négative; elle n'est pas précédée d'un signe plus si elle est positive et n'est pas précédée de zéros de tête. Elle est affichée avec le nombre minimum de caractères. Les registres et les codes de condition ne sont pas affectés. Tous les modes d'adressage sont permis.

Exemple

```
Addr  Code    Symbol    Mnemon  Operand     Comment
              ; lecture et addition de deux nombres entiers
              ; et affichage du résultat
              FINLIGNE:.EQUATE 0x000A
0000  41002B            STRO    demande,d   ; cout << "Donnez un nombre: "
0003  310025            DECI    nb1,d       ; cin >> nb1
0006  50000A            CHARO   FINLIGNE,i  ; cout << endl;
0009  41002B            STRO    demande,d   ; cout << "Donnez un nombre: "
000C  310027            DECI    nb2,d       ; cin >> nb2
000F  50000A            CHARO   FINLIGNE,i  ; cout << endl;
0012  C10025            LDA     nb1,d       ; nb1
0015  710027            ADDA    nb2,d       ;    + nb2
0018  E10029            STA     res,d       ;         => res
001B  41003E            STRO    result,d    ; cout << "Résultat = "
001E  390029            DECO    res,d       ; cout << res
0021  50000A            CHARO   FINLIGNE,i  ;      << endl;
0024  00                STOP
0025  0000    nb1:     .WORD   0           ; int nb1;
0027  0000    nb2:     .WORD   0           ; int nb2;
0029  0000    res:     .WORD   0           ; int res;
002B  446F6E  demande: .ASCII  "Donnez un nombre: \x00"
      6E657A
      20756E
      206E6F
      6D6272
      653A20
      00
003E  52E973  result:  .ASCII  "Résultat = \x00"
      756C74
      617420
      3D2000
                       .END
```

Les résultats de l'exécution en mode interactif sont les suivants :

```
Donnez un nombre: 45
Donnez un nombre: 67
Résultat = 112
```

7.6 Instructions logiques

Les instructions logiques sont des instructions qui traitent les valeurs comme des ensembles de bits.

Instruction `NOTr`

Cette instruction unaire effectue une opération de négation logique sur le contenu du registre. Tous les bits sont inversés. L'instruction affecte les codes de condition N et Z. Par exemple :

```
LDA     0x3F4A,i    ; A = 3F4A  NZVC=0000
NOTA                ; A = C0B5  NZVC=1000
```

Instruction `ANDr`

Cette instruction effectue une opération ET logique bit à bit sur les contenus du registre et de l'opérande; elle est souvent utilisée pour masquer des portions d'une valeur de 16 bits. Elle affecte les codes de condition N et Z et ne modifie pas les codes de condition C et V. Par exemple :

```
LDA     0xC0B5,i    ; A = C0B5  NZVC=1000
ANDA    0x3F4A,i    ; A = 0000  NZVC=0100
```

Instruction `ORr`

Cette instruction effectue une opération OU logique bit à bit sur les contenu du registre et de l'opérande; elle est souvent utilisée pour insérer des bits 1 dans une valeur de 16 bits. Elle affecte les codes de condition N et Z et ne modifie pas les codes de condition C et V. Par exemple :

```
LDA     0xC0B5,i    ; A = C0B5  NZVC=1000
ORA     0x3F4A,i    ; A = FFFF  NZVC=1000
```

Instruction `ASLr`

L'instruction `ASLr` effectue un décalage à gauche (L pour *left*) d'une position. Le bit de gauche qui sort du registre est placé dans le code de condition C. Un bit zéro entre par la droite pour prendre la place du dernier bit, comme le montre la figure 7.2. Les codes de condition N, Z, V et C sont affectés. Le code de condition V est mis à 1 dans le cas où la valeur du premier bit (à gauche) change.

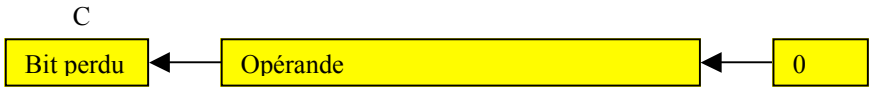

Figure 7.2 Décalage arithmétique à gauche ASLr

Exemple :

```
LDX     0x789F,i    ; X = 789F  NZVC=0000
ASLX                ; X = F13E  NZVC=1010
ASLX                ; X = E27C  NZVC=1001
```

Instruction `ASRr`

L'instruction `ASRr` effectue un décalage à droite (R pour *right*) d'une position. Le bit de droite qui sort du registre est placé dans le code de condition C. Le bit de signe est reproduit et prend la place du premier bit, comme le montre la figure 7.3. Les codes de condition N, Z, et C sont affectés. Le code de condition V n'est pas touché.

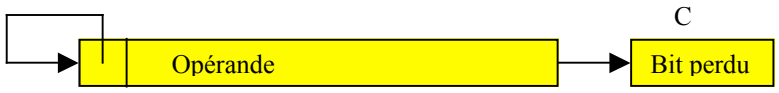

Figure 7.3 Décalage arithmétique à droite ASRr

Exemple :

```
LDA     0x9876,i    ; A = 9876  NZVC=1001
```

```
ASRA                    ; A = CC3B  NZVC=1000
ASRA                    ; A = E61D  NZVC=1001
```

Instruction `ROLr`

L'instruction `ROLr` effectue une rotation cyclique à gauche (L pour *left*) d'une position du contenu du registre associé au code de condition C. Le bit de droite du registre reçoit le code de condition C, ce dernier reçoit comme nouvelle valeur le bit sorti du registre, comme le montre la figure 7.4. Les codes de condition N, Z et V ne sont pas affectés.

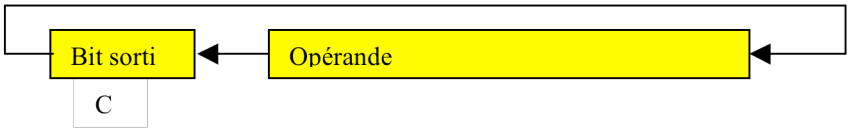

Figure 7.4 Décalage à gauche ROLr

Par exemple, les instructions suivantes produisent les résultats indiqués en commentaires.

```
LDA 0x385C,i ; NZVC = 0000 A = 385C
ROLA         ; NZVC = 0000 A = 70B8
ROLA         ; NZVC = 0000 A = E170
ROLA         ; NZVC = 0001 A = C2E0
ROLA         ; NZVC = 0001 A = 85C1
```

Instruction `RORr`

L'instruction `RORr` effectue une rotation cyclique à droite (R pour *right*) d'une position du registre associé au code de condition C. Le bit de gauche du registre reçoit la valeur du code de condition C et ce dernier reçoit le bit de droite du registre comme nouvelle valeur, tel que le montre la figure 7.5. Les codes de condition N, Z et V ne sont pas affectés.

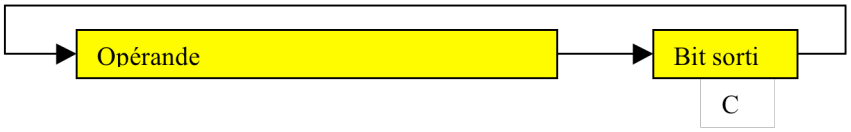

Figure 7.5 Décalage à gauche RORr

Par exemple, les instructions suivantes produisent les résultats indiqués en commentaires.

```
LDA 0x1C2E,i ; NZVC = 0000 A = 1C2E
RORA         ; NZVC = 0000 A = 0E17
RORA         ; NZVC = 0001 A = 070B
RORA         ; NZVC = 0001 A = 8385
RORA         ; NZVC = 0001 A = C1C2
```

Instruction `CPr`

L'instruction de comparaison `CPr` soustrait l'opérande de la valeur contenue dans le registre et positionne les codes de condition N, Z, V et C selon le résultat obtenu. Le contenu du registre ne change pas.

Exemple

Le programme ci-dessous compte et affiche le nombre de bits égaux à 1 dans un mot mémoire donné. Pour mieux « voir » la place des bits 1 dans le mot, on utilise le sous-programme HEXO que l'on

retrouvera au chapitre 8 et dont l'explication complète sera donnée au chapitre 9. Ce sous-programme affiche le contenu d'un mot mémoire au moyen de quatre chiffres hexadécimaux. Le programme commence par lire la valeur dont il va compter les bits 1 ; il l'affiche ensuite en décimal et en hexadécimal. Il élimine ensuite le bit de signe, qu'il compte s'il vaut 1. Le traitement est alors fait de façon répétitive jusqu'à ce que la valeur traitée soit nulle (plus aucun bit 1). Lorsque la valeur n'est pas nulle, on ne conserve que le bit de droite du mot au moyen d'une instruction ET logique ; si ce bit n'est pas nul, on le compte. On effectue alors un décalage à droite de la valeur traitée que l'on conserve, avant de recommencer. En fin de traitement on affiche la valeur du compte de bits 1. Par exemple, si la valeur donnée est `32767`, le programme affiche : **`#32767=x7FFF=15`**.

```
; Compter et afficher le nombre de bits égaux à "1" dans un mot mémoire.
;    Lorne H. Bouchard (adapté par Ph. Gabrini mars 2006)
;
debut:    DECI     nbLu,d    ;
          CHARO    '#',i     ; cout << '#'
          DECO     nbLu,d    ;      << nbLu décimal
          CHARO    '=',i     ;      << '='
          CHARO    'x',i     ;      << 'x'
          LDA      nbLu,d    ;
          STA      -2,s      ;
          SUBSP    2,i       ;
          CALL     HEXO      ;      << nbLu hexadécimal
          CHARO    '=',i     ;      << '=';
          LDA      nbLu,d    ;
          BRGE     boucle,i  ;
          ANDA     0x7FFF,i  ; on se débarrasse du bit de signe
          STA      nbLu,d    ;
          LDX      1,i       ;
          STX      nombre1,d ; mais on le compte
boucle:   LDA      nbLu,d    ;
          BREQ     fini      ; il n'y a plus de "1"
          ANDA     1,i       ;
          BREQ     zero      ; si le bit est "1"
          LDA      1,i       ;
          ADDA     nombre1,d ;
          STA      nombre1,d ;   on compte le bit
zero:     LDA      nbLu,d    ;
          ASRA               ; on passe au bit suivant
          STA      nbLu,d    ;
          BR       boucle    ;
fini:     DECO     nombre1,d ; cout << nombre de bits 1
          STOP               ;
nbLu:     .WORD    0         ; nombre lu
nombre1:  .WORD    0         ; nombre de "1" dans nbLu
       .........
         .END
```

7.7 Instructions de branchement

Les instructions de branchement permettent de rompre l'exécution séquentielle des instructions du programme en faisant passer de l'instruction de branchement à une autre instruction non consécutive à cette instruction. Une fois les codes de condition définis par une instruction quelconque ou par une

instruction de comparaison, on peut utiliser des instructions de branchement conditionnel dont le comportement est dicté par ces codes de condition : soit un branchement, soit une continuation en séquence. Ces instructions ne modifient pas les codes de condition et leur mode d'adressage est soit immédiat, comme l'instruction « `BRGE  Boucle,i` » de l'exemple en fin de section 7.6, soit indexé. Par défaut, l'assembleur vous permet d'omettre le « `,i` » et d'écrire « `BRGE  Boucle` », qu'il considère équivalente à l'instruction ci-dessus.

Instruction `BR`

L'instruction de branchement inconditionnel `BR` place l'adresse qui se trouve dans sa partie opérande dans le compteur ordinal, forçant ainsi la prochaine instruction a être celle dont l'adresse se trouve dans l'instruction de branchement. L'exécution en séquence est automatique, ne faites donc pas :

```
                        STA     Var,d
                        BR      LaSuite
                LaSuite: LDX    Var,d
```

où l'instruction BR est totalement inutile.

Exemple 1

Ce premier exemple illustre une possibilité offerte par la programmation de bas niveau exploitée, il y a bien longtemps, alors que les machines disposaient de très peu de mémoire (par exemple un gros « mainframe » de plusieurs millions de dollars avec une mémoire centrale de 64 Kmots de 36 bits!). Comme la place manquait, on modifiait le code au fur et à mesure de la progression dans une boucle, par exemple. Cette sorte de programmation est à proscrire, car lorsqu'on lit le code sur papier ou à l'écran, après la première passe dans la boucle les instructions affichées ne sont pas celles qui sont exécutées! À l'époque on se servait de cette technique, car on avait trop peu de mémoire pour avoir le choix, et non parce que les programmeurs étaient nuls.

```
Addr  Code    Symbol   Mnemon  Operand    Comment
              ;Un programme impur (qui modifie ses instructions au cours de
              ;l'exécution) à ne pas émuler, mais à comprendre.
0000  C00000          LDA     0,i        ; vider registre A
0003  D1001C Encore:  LDBYTEA texte,d    ;while(true){
0006  0A001B          BREQ    Arret      ; if(caractère == 0) break;
0009  51001C          CHARO   texte,d    ; else{  afficher caractère
000C  C10004          LDA     0x0004,d   ;   copier opérande LDBYTEA
000F  700001          ADDA    1,i        ;   et l'augmenter
0012  E10004          STA     0x0004,d   ;   modifier opérande LDBYTEA
0015  E1000A          STA     0x000A,d   ;   modifier opérande CHARO
0018  040003          BR      Encore     ; }//if
001B  00      Arret:  STOP               ;}//while
001C  556E20 texte:   .ASCII  "Un texte assez long....\x00"
      746578
      746520
      617373
      657A20
      6C6F6E
      672E2E
      2E2E00
                    .END
```

Les opérandes des instructions aux adresses 3 et 9 sont modifiés à chaque passage dans la boucle : les adresses sont augmentées de 1 à chaque fois, ce qui permet de passer au prochain caractère dans la

chaîne de caractères `texte`. Ainsi l'instruction à l'adresse 3 qui est à l'origine `D1001C` devient `D1001D`, puis `D1001E`, puis encore `D1001F` et `D10020`, etc. L'instruction à l'adresse 9 subit des changements semblables. De cette façon la chaîne de caractères est sortie entièrement.

Exemple 2

Ce programme est une seconde version de l'exemple précédent; cette version peut être dite « pure », car le code n'est pas modifié au cours de l'instruction. Cette notion de pureté est importante dans les systèmes où un grand nombre d'utilisateurs peuvent partager des logiciels semblables. Pour un tel partage, si le code est « pur », le système n'a besoin que d'une seule copie du code, qui est alors exécutée simultanément par plusieurs utilisateurs, ayant chacun leur ensemble de registres et leur compteur ordinal.

```
Addr  Code    Symbol   Mnemon  Operand    Comment
            ; Programme pur d'affichage d'une chaîne de caractères.
0000  C80000          LDX     0,i        ; X = 0;
0003  C00000          LDA     0,i        ; A = 0;
0006  D50016 Encore:  LDBYTEA texte,x    ; while(true) {
0009  0A0015          BREQ    Arret      ;   if(texte[X] == 0) break;
000C  550016          CHARO   texte,x    ;   else{ cout << texte[X];
000F  780001          ADDX    1,i        ;   X++;}
0012  040006          BR      Encore     ; }
0015  00      Arret:  STOP
0016  556E20 texte:   .ASCII  "Un texte assez long.....\x00"
      746578
      746520
      617373
      657A20
      6C6F6E
      672E2E
      2E2E2E
      00
                      .END
```

Dans cet exemple la chaîne `texte` est indicée par le registre X, lequel débute à zéro et avance d'un octet à la fois, repérant ainsi les caractères de la chaîne un par un. Là encore, la chaîne de caractères est sortie complètement.

Instruction `BRLE`

L'instruction de branchement conditionnel `BRLE` vérifie les codes de condition et si Z ou N valent 1 produit un branchement à l'adresse comprise dans l'instruction. Sinon le traitement se poursuit par l'instruction suivant le `BRLE`.

Instruction `BRLT`

L'instruction de branchement conditionnel `BRLT` vérifie les codes de condition et si N vaut 1 produit un branchement à l'adresse comprise dans l'instruction. Sinon le traitement se poursuit par l'instruction suivant le `BRLT`.

Instruction BREQ
L'instruction de branchement conditionnel `BREQ` vérifie les codes de condition et si Z vaut 1 produit un branchement à l'adresse comprise dans l'instruction. Sinon le traitement se poursuit par l'instruction suivant le `BREQ`.

Instruction BRNE
L'instruction de branchement conditionnel `BRNE` vérifie les codes de condition et si Z vaut 0 produit un branchement à l'adresse comprise dans l'instruction. Sinon le traitement se poursuit par l'instruction suivant le `BRNE`.

Instruction BRGE
L'instruction de branchement conditionnel `BRGE` vérifie les codes de condition et si N vaut 0 produit un branchement à l'adresse comprise dans l'instruction. Sinon le traitement se poursuit par l'instruction suivant le `BRGE`.

Instruction BRGT
L'instruction de branchement conditionnel `BRGT` vérifie les codes de condition et si Z et N valent 0 produit un branchement à l'adresse comprise dans l'instruction. Sinon le traitement se poursuit par l'instruction suivant le `BRGT`.

Instruction BRV
L'instruction de branchement conditionnel `BRV` vérifie le code de condition V et, s'il vaut 1, produit un branchement à l'adresse comprise dans l'instruction, sinon le traitement se poursuit par l'instruction suivant le `BRV`.

Exemple 3
Le programme qui suit vérifie le résultat d'une opération arithmétique. La première opération est la soustraction du nombre négatif de plus petite valeur absolue (0x8000) de la valeur zéro, ce qui revient à simplement en changer le signe. Comme la valeur positive la plus grande est 0x7FFF, il y aura débordement, vérifié par l'instruction BRV, laquelle affichera le message **`r=8000 V=1`** avant de revenir à l'étiquette `continue`. La seconde opération en est une d'addition de la plus petite valeur négative à elle-même, ce qui produit un débordement, à nouveau détecté par l'instruction BRV, laquelle affiche le message : **`r=0000 V=1`**. Ce programme utilise également le sous-programme HEXO qui permet d'afficher des valeurs hexadécimales.

```
;  Test de débordement
;    Lorne H. Bouchard  (adapté par Ph. Gabrini mars 2006)
;
debut:   LDA    0,i       ; 0
         SUBA   0x8000,i  ; - nombre (négatif) le plus petit
         STA    res,d     ;
         BRV    deborde   ;
continue:LDA    1,i       ;
         STA    indic,d   ;
         LDA    0x8000,i  ; nombre (négatif) le plus petit
         ADDA   0x8000,i  ; + nombre (négatif) le plus petit
         STA    res,d     ;
         BRV    deborde   ;
         STOP             ;
```

```
deborde: CHARO  'r',i      ;  cout << "r="
         CHARO  '=',i      ;
         LDA    res,d      ;
         STA    -2,s       ;
         SUBSP  2,i        ;
         CALL   HEXO       ;       << valeur hexadécimale
         CHARO  ' ',i      ;
         CHARO  'V',i      ;       << " V=1"
         CHARO  '=',i      ;
         CHARO  '1',i      ;
         CHARO  0xA,i      ;       << endl;
         LDA    indic,d    ;
         BREQ   continue   ;  if(indic == 0) continue
         STOP              ;
res:     .WORD  0
indic:   .WORD  0
     ........
         .END
```

Instruction `BRC`

L'instruction de branchement conditionnel `BRC` vérifie le code de condition C et, s'il vaut 1, produit un branchement à l'adresse comprise dans l'instruction, sinon le traitement se poursuit par l'instruction suivant le `BRC`.

Exemple 4

Le programme ci-dessous effectue la même tâche que le programme vu à la section 7.6, lequel compte et affiche le nombre de bits égaux à 1 dans un mot mémoire donné. Ce programme commence de la même façon que le précédent, en lisant la valeur dont il va compter les bits 1 ; il l'affiche ensuite en décimal et en hexadécimal (par appel du sous-programme HEXO). Il élimine ensuite le bit de signe, qu'il compte s'il vaut 1. Le traitement est alors fait de façon répétitive jusqu'à ce que la valeur traitée soit nulle (plus aucun bit 1). Lorsque la valeur n'est pas nulle, on effectue un décalage à droite de la valeur traitée, que l'on conserve, avant de vérifier le contenu de C et de recommencer. Si le contenu de C vaut 1, l'instruction BRC effectue un saut à l'étiquette `un`, où on compte un bit 1, avant de continuer. En fin de traitement, on affiche la valeur du compte de bits 1. Par exemple, si la valeur donnée est `32767`, le programme affiche : **`#32767=x7FFF=15`**.

```
; Compter et afficher le nombre de bits égaux à "1" dans un mot mémoire.
; Utilisation du code de condition C, touché par le décalage.
; Lorne H. Bouchard (adapté par Ph. Gabrini mars 2006)
;
debut:    DECI     nbLu,d    ;
          CHARO    '#',i     ;
          DECO     nbLu,d    ;
          CHARO    '=',i     ;
          CHARO    'x',i     ;
          LDA      nbLu,d    ;
          STA      -2,s      ;
          SUBSP    2,i       ;
          CALL     HEXO      ;
          CHARO    '=',i     ;
          LDA      nbLu,d    ;
          BRGE     boucle    ;  if(nbLu < 0)
          ANDA     0x7FFF,i  ;    on se débarrasse du bit de signe
```

```
          STA     nbLu,d    ;
          LDX     1,i       ;
          STX     nombre1,d ;      et on le compte
boucle:   LDA     nbLu,d    ;  while(encore des bits 1)
          BREQ    fini      ;
          ASRA                ;    décaler A dans C
          STA     nbLu,d    ;    conserver
          BRC     un        ;    if(C == 1){
          BR      boucle    ;
un:       LDA     1,i       ;
          ADDA    nombre1,d ;      nombre1++;
          STA     nombre1,d ;      compter le bit
          BR      boucle    ;  }
fini:     DECO    nombre1,d ;      cout << nombre1;
          STOP                ;
nbLu:     .WORD   0         ; nombre lu
nombre1:  .WORD   0         ; nombre de "1" dans n
     ........
          .END
```

7.8 Instructions liées aux sous-programmes

Deux instructions sont liées aux sous-programmes et permettent l'appel et le retour; elles seront décrites plus en détail au chapitre 9.

Instruction `CALL`

L'instruction `CALL` empile la valeur du compteur ordinal et place l'adresse donnée dans l'instruction dans le compteur ordinal, transférant ainsi le contrôle à la première instruction du sous-programme. Son mode d'adressage est soit direct, soit indexé. Les codes de condition ne sont pas affectés.

Instruction `RETn`

L'instruction unaire `RETn` désempile n octets de la pile (n occupe trois bits et doit être compris entre 0 et 7), donne au compteur ordinal la valeur du nouveau sommet de la pile qui est alors désempilée. Les codes de condition ne sont pas affectés.

7.9 Instructions non réalisées

En plus des trois instructions d'entrée-sortie réalisées par traitement d'interruptions (STRO, DECI, DECO) il existe deux autres instructions basées sur des codes instruction non réalisés et qui sont traitées par le système d'interruption de PEP 8.

Instruction `NOPn`

L'instruction `NOPn` est une instruction unaire, qui n'a donc pas d'opérande autre que le nombre n. Ce dernier occupe 2 bits et doit donc avoir une valeur entre 0 et trois. L'exécution de cette instruction engendre une interruption au niveau de la machine, mais elle peut être utilisée comme toute autre instruction Assembleur. Cette instruction ne fait rien (`NOP` = *no operation*), mais elle pourrait être remplacée par toute autre instruction utile, comme par exemple des instructions de traitement des valeurs réelles (voir chapitre 11). Les codes de condition ne sont pas affectés.

Instruction NOP

L'instruction NOP est une instruction qui nécessite un opérande en mode d'adressage immédiat. Cet opérande peut être utilisé dans le traitement de l'instruction qui est basée sur un code d'instruction non réalisé. L'exécution de cette instruction engendre donc une interruption au niveau de la machine, mais elle peut être utilisée comme toute autre instruction Assembleur. Cette instruction ne fait rien (NOP = *no operation*), mais il est possible de modifier son traitement dans le système d'exploitation pour mettre en place d'autres instructions utiles (voir chapitre 11), comme par exemple des instructions de multiplication et de division entière. Les codes de condition ne sont pas affectés.

7.10 Exemples d'application

Exemple 1

```
; lire deux entiers et imprimer le plus petit suivi du plus grand
FINLIGNE:.EQUATE    0xA    ; caractère de fin de ligne
         STRO   demande,d  ; cout << "Donnez un nombre: ";
         DECI   nb1,d      ; cin >> nb1
         CHARO  FINLIGNE,i ; cout << endl;
         STRO   demande,d  ; cout << "Donnez un nombre: "
         DECI   nb2,d      ; cin >> nb2
         CHARO  FINLIGNE,i ; cout << endl;
         STRO   titre,d    ; cout << "En ordre = "
         LDX    nb1,d      ; if(nb1
         CPX    nb2,d      ;     > nb2)
         BRLE   EnOrdre    ; {
         DECO   nb2,d      ;   cout << nb2
         CHARO  ' ',i      ;        << ' '
         DECO   nb1,d      ;        << nb1
         CHARO  FINLIGNE,I ;        << endl;
         STOP              ; }else{
EnOrdre: DECO   nb1,d      ;   cout << nb1
         CHARO  ' ',i      ;        << ' '
         DECO   nb2,d      ;        << nb2
         CHARO  FINLIGNE,I ;        << endl;
         STOP              ; }
nb1:     .WORD   0         ; int nb1;
nb2:     .WORD   0         ; int nb2;
demande: .ASCII  "Donnez un nombre: \x00"
titre:   .ASCII  "En ordre = \x00"
         .END
```

Résultat en mode interactif :

```
Donnez un nombre: 987
Donnez un nombre: 125
En ordre = 125 987
```

Exemple 2

```
;affichage des nombres de 1 à limite
LIMITE:  .EQUATE    12   ; nombre de valeurs à afficher
         STRO   titre,d  ; cout << "Affichage des nombres: " >> endl;
         LDX    1,i      ; compte = 1;
Repete:  STX    nb,d     ; do {
         DECO   nb,d     ;   cout << nb
         CHARO  ' ',i    ;        << ' '
```

```
         ADDX   1,i      ;   compte++;
         CPX    LIMITE,i ; }
         BRLE   Repete   ; while(compte <= LIMITE);
         CHARO  0xA,i    ; cout << endl;
         STOP            ;
nb:      .WORD   0       ; int nb;
titre:   .ASCII  "Affichage des nombres: \x00"
         .END
```

Résultat de l'exécution :

```
Affichage des nombres: 1 2 3 4 5 6 7 8 9 10 11 12
```

Exemple 3

```
;lecture, calcul et affichage de la somme de nombres entiers,
;-9999 est la sentinelle qui marque la fin des données.
                        ; while(true){
Boucle:  DECI   n,d     ;   cin >> n;
         LDA    n,d     ;
         CPA    -9999,i ;   if(n != -9999)
         BREQ   Affiche ;
         ADDA   somme,d ;     somme += n;
         STA    somme,d ;
         BR     Boucle  ;   else break;}
Affiche: DECO   somme,d ; cout << somme
         CHARO  0xA,i    ;      << endl;
         STOP            ;
n:       .WORD  0        ; int n = 0;
somme:   .WORD  0        ; int somme = 0;
         .END
```

Résultat de l'exécution avec les données 1 3 5 7 9 11 13 17 19 -9999 : `85.`

Exemple 4

Cet exemple est simple et illustre les branchements conditionnels. Le programme demande à l'utilisateur de deviner un nombre compris entre 0 et 100. Si la valeur soumise est en dehors de l'intervalle permis un message explicatif est donné. Sinon on indique si la valeur soumise est trop petite, trop grande ou la bonne valeur. Le programme se termine sur la bonne valeur.

```
Addr  Code   Symbol   Mnemon  Operand     Comment
                                          ;Programme devinette: deviner un nombre
                                          ;while(true){
0000  41003B Devine:  STRO    demande,d   ;  cout << "Donnez un nombre 0..100: "
0003  310037 Suivant: DECI    essai,d     ;  cin >> essai;
0006  C10037          LDA     essai,d     ;
0009  08001E          BRLT    Cas0        ;  if(essai < 0)Cas0
000C  B00064          CPA     100,i       ;
000F  10001E          BRGT    Cas0        ;  if(essai > 100)Cas0
0012  B10039          CPA     nb,d        ;
0015  080024          BRLT    Cas1        ;  if(essai < nb)Cas1
0018  0A002A          BREQ    Cas2        ;  elsif(essai == nb)Cas2
001B  100030          BRGT    Cas3        ;  elsif(essai > nb)Cas3
001E  410055 Cas0:    STRO    msg0,d      ;  cout << "En dehors, autre nombre: "
0021  040033          BR      Continue    ;  continue;
0024  41006F Cas1:    STRO    msg1,d      ;  cout << "Trop petit, autre nombre:
0027  040033          BR      Continue    ;  continue;
002A  41008A Cas2:    STRO    msg2,d      ;  cout << "Vous avez gagné; "
```



```
002D  040036          BR      Fin         ;  continue;
0030  41009A Cas3:    STRO    msg3,d      ;  cout << "Trop grand, autre nombre: "
0033  040003 Continue:BR      Suivant     ;}
0036  00     Fin:     STOP                ;
0037  0000   essai:   .BLOCK  2           ;int essai
0039  003D   nb:      .WORD   61          ;const int nb = 61;
003B  446F6E demande: .ASCII  "Donnez un nombre 0..100: \x00"
      6E657A
      20756E
      206E6F
      6D6272
      652030
      2E2E31
      30303A
      2000
0055  456E20 msg0:    .ASCII  "En dehors, autre nombre: \x00"
      646568
      6F7273
      2C2061
      757472
      65206E
      6F6D62
      72653A
      2000
006F  54726F msg1:    .ASCII  "Trop petit, autre nombre: \x00"
      702070
      657469
      742C20
      617574
      726520
      6E6F6D
      627265
      3A2000
008A  566F75 msg2:    .ASCII  "Vous avez gagné\x00"
      732061
      76657A
      206761
      676EE9
      00
009A  54726F msg3:    .ASCII  "Trop grand, autre nombre: \x00"
      702067
      72616E
      642C20
      617574
      726520
      6E6F6D
      627265
      3A2000
                      .END
```

Le résultat d'une exécution interactive suit.

```
Donnez un nombre 0..100: 45
Trop petit, autre nombre: 75
Trop grand, autre nombre: 110
En dehors, autre nombre: -12
```

```
En dehors, autre nombre: 55
Trop petit, autre nombre: 65
Trop grand, autre nombre: 60
Trop petit, autre nombre: 62
Trop grand, autre nombre: 61
Vous avez gagné
```

Exemple 5

Le programme ci-dessous lit un ensemble de valeurs entières et les range dans un vecteur d'entiers. Ensuite, il affiche ces valeurs à partir de la fin du tableau à raison d'une par ligne, chaque valeur étant précédée de son indice.

```
Addr  Code   Symbol   Mnemon  Operand    Comment
             ;Programme qui lit des données, les place dans un vecteur et les
             ;affiche à l'envers une par ligne, précédées de leur indice.
             TAILLE:  .EQUATE 12         ;taille du vecteur
0000  C80000 LitVec:  LDX     0,i        ;int main() {
0003  E90060          STX     index,d    ;
0006  B8000C Boucle1: CPX     TAILLE,i   ;  for(i = 0; i < TAILLE; i++){
0009  0E001C          BRGE    FinBouc1   ;
000C  1D              ASLX               ;    //entier = 2 octets
000D  350048          DECI    vecteur,x  ;    cin >> vector[i];
0010  C90060          LDX     index,d    ;
0013  780001          ADDX    1,i        ;
0016  E90060          STX     index,d    ;
0019  040006          BR      Boucle1    ;  }
001C  C8000C FinBouc1:LDX     TAILLE,i   ;  for(i = TAILLE-1; i >= 0; i--){
001F  880001          SUBX    1,i        ;
0022  E90060          STX     index,d    ;
0025  50000A          CHARO   '\n',i     ;    cout << endl;
0028  B80000 Boucle2: CPX     0,i        ;
002B  080047          BRLT    FinBouc2   ;
002E  390060          DECO    index,d
0031  500020          CHARO   ' ',i      ;         << ' '
0034  1D              ASLX               ;    //entier = 2 octets
0035  3D0048          DECO    vecteur,x  ;         << vector[i]
0038  50000A          CHARO   '\n',i     ;         << endl;
003B  C90060          LDX     index,d    ;
003E  880001          SUBX    1,i        ;
0041  E90060          STX     index,d    ;
0044  040028          BR      Boucle2    ;  }
0047  00      FinBouc2:STOP              ;  return 0;}
0048  000000 vecteur: .BLOCK  24
      000000
      000000
      000000
      000000
      000000
      000000
      000000
0060  0000   index:   .BLOCK  2          ; int index;
                     .END
```

Avec les données `39 25 54 46 51 42 21 18 11 14 23 49`, l'exécution du programme donne les résultats suivants.

```
11 49
10 23
9 14
8 11
7 18
6 21
5 42
4 51
3 46
2 54
1 25
0 39
```

Exemple 6

Cet exemple est plus important en taille, mais n'est quand même pas très compliqué, Il est facile à expliquer et en suivant l'explication ainsi que les commentaires en pseudo C++, il ne devrait pas y avoir de difficulté de compréhension. Le programme traite des températures exprimées soit en degrés Fahrenheit[1], soit en degrés Celsius[2]. Il accepte des données dans l'un des deux systèmes et calcule la valeur équivalente dans l'autre système, avant d'afficher les deux valeurs.

Exprimé en C++ le programme serait :

```
int main(){
  cout << " Entrez température ";
  cin >> temperature;
  cin >> degres;
  while(temperature != SENTINELLE){
    if(degres != 'F' && degres != 'f'){
      celsius = temperature;
      fahrenheit = celsius * 9 / 5 + 32;
    }
    else{
      fahrenheit = temperature;
      celsius = (temperature-32) * 5 / 9;
    }
    cout << fahrenheit/10 << '.' << fahrenheit%10 << " Fahrenheit = ";
    cout << celsius/10 << '.' << celsius%10 << " Celsius" << endl;
    cout << " Entrez température ";
    cin >> temperature;
    cin >> degres;
  }//while
  return 0 ;
}
```

Comme on peut le constater, ce code est assez simple. Le programme correspondant en assembleur est cependant bien plus long. Il y a pour cela plusieurs raisons. D'abord, on le sait, un programme Assembleur comporte toujours bien plus d'instructions qu'un programme en langage évolué. Ensuite, le processeur PEP 8, qui est, disons le, quelque peu rudimentaire, ne comporte pas d'instructions de multiplication et de division entières ; il faut donc écrire des sous-programmes pour les émuler et faire des appels à ces sous-programmes (voir le chapitre 9 pour les détails), tout en ajoutant leur code à la solution. Enfin, ces sous-programmes de multiplication et de division ne traitent que des valeurs

[1] Daniel Fahrenheit, physicien allemand (1686-1736)
[2] Anders Celsius, astronome suédois (1701-1744)

positives ; ceci nous a obligé à vérifier le signe des valeurs calculées et à travailler avec des valeurs absolues en rajoutant le signe négatif à la sortie, si besoin est. Tout ceci a contribué à augmenter la quantité de code.

```
Addr  Code   Symbol   Mnemon  Operand    Comment
             ; Thermos effectue la conversion de températures de Fahrenheit
             ; en Celsius ou inversement. La température est donnée en dixième
             ; de degrés et suivie d'une espace et de la lettre F ou f ou C ou c.
             ; Le résultat est donné avec une décimale.
             ;         Philippe Gabrini    septembre 2002-revu en octobre 2005
             SENTINEL:.EQUATE 20000      ; limite des degrés
             CINQ:    .EQUATE 5          ; facteur multiplicateur ou diviseur
             NEUF:    .EQUATE 9          ; facteur diviseur ou multiplicateur
             DIX:     .EQUATE 10         ; facteur diviseur
             TRENTE2: .EQUATE 320        ; constante à soustraire
             NEWLINE: .EQUATE 0x000A     ;
             ESPACE:  .EQUATE 0x0020     ;

0000  410159 Thermos: STRO    entrez,d   ;  cout << " Entrez température ";
0003  310146          DECI    temper,d   ;  cin >> température;
0006  49014A          CHARI   degres,d   ;  cin >> degres;
0009  C10146 TestFin: LDA     temper,d   ;
000C  B04E20          CPA     SENTINEL,i ;  while(temper != sentinelle){
000F  0A013F          BREQ    Fin        ;
0012  C00000          LDA     0,i        ;
0015  D1014A          LDBYTEA degres,d   ;    if(degres != 'F'
0018  B00046          CPA     'F',i      ;
001B  0A006E          BREQ    Fahr       ;
001E  B00066          CPA     'f',i      ;      && degres != 'f'){
0021  0A006E          BREQ    Fahr       ;
0024  C10146          LDA     temper,d   ;
0027  E1014F          STA     celsius,d  ;      celsius = temper;
002A  0E0034          BRGE    Calcul     ;      if(celsius < 0) {
002D  E10155          STA     negaCels,d ;        negaCels = celsius;
0030  1A              NEGA               ;
0031  E1014F          STA     celsius,d  ;        celsius = -celsius;}
0034  E3FFFA Calcul:  STA     -6,s       ;      empiler
0037  C00009          LDA     NEUF,i     ;      empiler 9
003A  E3FFF8          STA     -8,s       ;
003D  680008          SUBSP   8,i        ;
0040  1601C4          CALL    Mulss      ;      celsius * 9
0043  C30002          LDA     2,s        ;      deuxième moitié seulement
0046  600004          ADDSP   4,i        ;      désempiler résultat
0049  E3FFF8          STA     -8,s       ;
004C  C00005          LDA     CINQ,i     ;      empiler 5
004F  E3FFFA          STA     -6,s       ;
0052  680008          SUBSP   8,i        ;
0055  160240          CALL    Divss      ;      celsius * 9 / 5
0058  C30002          LDA     2,s        ;
005B  600004          ADDSP   4,i        ;
005E  C90155          LDX     negaCels,d ;      if(celsius < 0)
0061  0E0065          BRGE    OKplus     ;
0064  1A              NEGA               ;        -(celsius *9 / 5)
0065  700140 OKplus:  ADDA    TRENTE2,i  ;      + 32
0068  E10148          STA     fahren,d   ;      fahren = celsius * 9 / 5 + 32;
006B  0400AB          BR      Sortie     ;    else {
```

```
006E  C10146 Fahr:    LDA     temper,d   ;
0071  E10148          STA     fahren,d   ;
0074  800140          SUBA    TRENTE2,i  ;      temper - 32
0077  0E007E          BRGE    Continue   ;      if(temper - 32 < 0)
007A  E10155          STA     negaCels,d ;        negaCels = temper - 32;
007D  1A              NEGA               ;        -(temper - 32);
007E  E3FFFA Continue:STA     -6,s       ;      empiler
0081  C00005          LDA     CINQ,i     ;      empiler 5
0084  E3FFF8          STA     -8,s       ;
0087  680008          SUBSP   8,i        ;
008A  1601C4          CALL    Mulss      ;      (temper-32) * 5
008D  C30002          LDA     2,s        ;      deuxième moitié seulement
0090  600004          ADDSP   4,i        ;      désempiler résultat
0093  E3FFF8          STA     -8,s       ;
0096  C00009          LDA     NEUF,i     ;      empiler 9
0099  E3FFFA          STA     -6,s       ;
009C  680008          SUBSP   8,i        ;
009F  160240          CALL    Divss      ;      (temper-32) * 5 / 9
00A2  C30002          LDA     2,s        ;
00A5  600004          ADDSP   4,i        ;      celsius = (temper-32) * 5 / 9;
00A8  E1014F          STA     celsius,d  ;    }//if
00AB  C1014F Sortie:  LDA     celsius,d  ;    empiler celsius
00AE  E3FFF8          STA     -8,s       ;
00B1  C0000A          LDA     DIX,i      ;    empiler 10
00B4  E3FFFA          STA     -6,s       ;
00B7  680008          SUBSP   8,i        ;
00BA  160240          CALL    Divss      ;    celsius / 10
00BD  C30000          LDA     0,s        ;
00C0  E10153          STA     cels2,d    ;    cels2 = celsius % 10;
00C3  C30002          LDA     2,s        ;
00C6  E10151          STA     cels1,d    ;    cels1 = celsius / 10;
00C9  600004          ADDSP   4,i        ;
00CC  C10148          LDA     fahren,d   ;    if(fahren < 0){
00CF  0E00D9          BRGE    Decoupe    ;
00D2  E10157          STA     negaFahr,d ;
00D5  1A              NEGA               ;      fahren = -fahren;
00D6  E10148          STA     fahren,d   ;    }
00D9  E3FFF8 Decoupe: STA     -8,s       ;    empiler fahren
00DC  C0000A          LDA     DIX,i      ;    empiler 10
00DF  E3FFFA          STA     -6,s       ;
00E2  680008          SUBSP   8,i        ;
00E5  160240          CALL    Divss      ;    fahren / 10
00E8  C30000          LDA     0,s        ;
00EB  E1014D          STA     far2,d     ;    far2 = fahren % 10;
00EE  C30002          LDA     2,s        ;
00F1  E1014B          STA     far1,d     ;    far1 = fahren / 10;
00F4  600004          ADDSP   4,i        ;    désempiler
00F7  50000A          CHARO   NEWLINE,i  ;
00FA  C10157          LDA     negaFahr,d ;    if(negaFahr != 0){
00FD  0A0103          BREQ    Fplus      ;      cout << '-';
0100  50002D          CHARO   '-',i      ;    }
0103  39014B Fplus:   DECO    far1,d     ;    cout << Fahrenheit1
0106  50002E          CHARO   '.',i      ;         << '.'
0109  39014D          DECO    far2,d     ;         << Fahrenheit2
010C  410197          STRO    fahrenhe,d ;         << " Fahrenheit = ";
010F  C10155          LDA     negaCels,d ;    if(negaCels == 0)
0112  0A0118          BREQ    Cplus      ;      cout << '-';
```

```
0115  50002D          CHARO   '-',i      ;
0118  390151 Cplus:   DECO    cels1,d    ;    cout << Celsius1
011B  50002E          CHARO   '.',i      ;         << '.'
011E  390153          DECO    cels2,d    ;         << Celsius2
0121  4101A6          STRO    celsiu,d   ;         << " Celsius";
0124  50000A          CHARO   NEWLINE,i  ;
0127  50000A          CHARO   NEWLINE,i  ;
012A  C00000          LDA     0,i        ;
012D  E10155          STA     negaCels,d ;    negaCels = 0;
0130  E10157          STA     negaFahr,d ;    negaFahr = 0;
0133  410159          STRO    entrez,d   ;    cout << " Entrez température ";
0136  310146          DECI    temper,d   ;    cin >> temper;
0139  49014A          CHARI   degres,d   ;    cin >> degres;
013C  040009          BR      TestFin    ;  }//while

013F  4101AF Fin:     STRO    fini,d     ;  cout << "Arrêt du...";
0142  50000A          CHARO   NEWLINE,i  ;
0145  00              STOP

0146  0000   temper:  .WORD   0
0148  0000   fahren:  .WORD   0
014A  00     degres:  .BYTE   0          ; F ou C
014B  0000   far1:    .WORD   0
014D  0000   far2:    .WORD   0
014F  0000   celsius: .WORD   0
0151  0000   cels1:   .WORD   0
0153  0000   cels2:   .WORD   0
0155  0000   negaCels:.WORD   0
0157  0000   negaFahr:.WORD   0
0159  456E74 entrez:  .ASCII  "Entrez une température degrés*10 suivie de F ou C
(ex:380 F) \x00"
      72657A
      20756E
      652074
      656D70
      E97261
      747572
      652064
      656772
      E9732A
      313020
      737569
      766965
      206465
      204620
      6F7520
      432028
      65783A
      333830
      204629
      2000
0197  204661 fahrenhe:.ASCII  " Fahrenheit = \x00"
      687265
      6E6865
      697420
      3D2000
01A6  204365 celsiu:  .ASCII  " Celsius\x00"
```



```
      6C7369
      757300
01AF  417272 fini:    .ASCII  "Arrêt du thermomètre\x00"
      EA7420
      647520
      746865
      726D6F
      6DE874
      726500
            ...................
                      .END
```

Les sous-programmes `Mulss`, `Divss` et `Granss` ne sont pas donnés, pour simplifier. Ils permettent respectivement de multiplier deux entiers positifs de 16 bits, de diviser deux entiers positifs de 16 bits et de comparer deux entiers de 16 bits sans tenir compte des signes.

Voici le résultat d'une exécution interactive du programme.

```
Entrez une température degrés*10 suivie de F ou C (ex:380 F) 2120 F
212.0 Fahrenheit = 100.0 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 380 F
38.0 Fahrenheit = 3.3 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 320 F
32.0 Fahrenheit = 0.0 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 200 f
20.0 Fahrenheit = -6.6 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) -120 f
-12.0 Fahrenheit = -24.4 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 0 f
0.0 Fahrenheit = -17.7 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) -400 c
-40.0 Fahrenheit = -40.0 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) -100 C
14.0 Fahrenheit = -10.0 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 0 C
32.0 Fahrenheit = 0.0 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 1000 C
212.0 Fahrenheit = 100.0 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 370 C
98.6 Fahrenheit = 37.0 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 1200 C
248.0 Fahrenheit = 120.0 Celsius

Entrez une température degrés*10 suivie de F ou C (ex:380 F) 20000 f
Arrêt du thermomètre
```

7.11 Exercices

7.11.1 Dans l'exemple 1 de la section 7.10, modifiez le programme pour qu'il sorte le plus grand nombre suivi du plus petit.

7.11.2 Dans l'exemple 2 de la section 7.10, modifiez le programme pour qu'il affiche une valeur par ligne.

7.11.3 Dans l'exemple 3 de la section 7.10, modifiez le programme pour qu'il calcule et affiche la somme des valeurs positives et la somme des valeurs négatives.

7.11.4 Dans l'exemple 4 de la section 7.10, modifiez le programme pour qu'il ne permette à l'utilisateur qu'un maximum de dix essais.

7.11.5 Dans l'exemple 5 de la section 7.10, modifiez le programme pour que l'affichage des valeurs se fasse de la première valeur du vecteur à la dernière.

7.11.6 Dans l'exemple 6 de la section 7.10, modifiez le programme pour vérifier que la température est bien suivie d'un 'c' ou d'un 'C' lorsque ce n'est pas un 'f' ou un 'F'.

