Chapitre 11
L'arithmétique réelle

Les instructions arithmétiques `ADD` et `SUB` de PEP 8 ne traitent que des valeurs entières, également dites "*valeurs en point fixe*". Comme on le sait, l'arithmétique entière présente certaines limites, en particulier celle de ne pas permettre de manipuler des nombres dont la valeur absolue est supérieure à 2^{15}. On sait déjà que les langages de programmation évolués permettent l'utilisation de valeurs numériques entières (généralement sur 32 bits), mais aussi de valeurs numériques réelles (`REAL` en Pascal ou en Modula-2, `float` en C ou en Ada 95); ceci est dû au fait que les ordinateurs peuvent en effet traiter plusieurs types de valeurs numériques. La plupart des ordinateurs actuels possèdent en fait trois types d'arithmétique: arithmétique entière, arithmétique réelle (seulement si équipés d'un co-processeur mathématique) et arithmétique décimale.

L'arithmétique réelle correspond à ce qu'on appelle souvent valeurs numériques "en *point flottant*"; dans ce vocable, le mot "point" identifie le point décimal (ou virgule) séparant la partie entière de la partie fractionnaire d'une valeur numérique réelle. En "point fixe", ce point décimal a une position fixe, généralement à droite de la valeur, qui ne comprend alors qu'une partie entière. En "point flottant", ce point décimal est situé n'importe où dans le nombre, comme dans 3.14159 ou 6.023×10^{-23} .[1]

11.1 Principes de la représentation des nombres réels

Les nombres réels auxquels nous sommes habitués possèdent une partie entière et une partie fractionnaire, et ont la forme des exemples suivants:

```
12.9        0.000419        -17.19        -0.0042
```

Ces quatre nombres peuvent également être exprimés sous la forme suivante :

0.129×10^{2} 0.419×10^{-3} -0.1719×10^{2} -0.42×10^{-2}

Cette forme est dite *normalisée*, dans la mesure où la partie entière des nombres est toujours nulle et où le premier chiffre de la partie fractionnaire est toujours différent de zéro. La valeur du nombre est ainsi indiquée par le signe, la partie fractionnaire et une puissance de dix. Dans cette représentation, un nombre réel peut donc être représenté par trois champs: le signe du nombre, la partie fractionnaire et la puissance de dix par laquelle multiplier la partie fractionnaire pour obtenir la valeur du nombre. En fait, c'est d'une façon semblable que les nombres réels sont représentés en mémoire, sous forme binaire cependant.

11.2 Représentation des nombres réels selon la norme IEEE 754

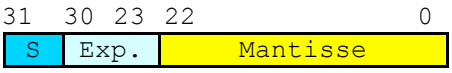
Lorsqu'on utilise une représentation des nombres réels basée sur le système binaire, le principe reste le même, mais au lieu de multiplier une partie fractionnaire décimale par une puissance de dix, on

[1] Si l'exposant était positif, on aurait la constante d'Avogadro représentant le nombre de molécules dans une mole. La formule est due au comte d'Avogadro (1776-1856), originaire de Turin et dont le nom est Amedeo di Quaregna e Ceretto.

multiplie une partie fractionnaire binaire par une puissance de deux. On rappelle que dans le système binaire, une multiplication par deux revient à un décalage d'un chiffre vers la gauche, tandis qu'une division par deux revient à un décalage d'un chiffre vers la droite. Ainsi, -17.25 est équivalent à -0.1725×10^2, soit en hexadécimal -11.4 qui est équivalent à -10001.01 ou -0.1000101×2^5 en binaire. Avec cette représentation, un nombre réel est alors toujours caractérisé par une partie fractionnaire (la partie entière étant nulle), un signe et une puissance de 2 aussi appelée *exposant*. La partie fractionnaire est aussi appelée *mantisse*. Cette représentation est *légèrement différente* de celle adoptée par la norme, mais le principe reste le même.

Selon la norme IEEE 754 (datant de 1985), les nombres réels ont **toujours** la forme binaire normalisée **$1.f \times 2^e$**, où f représente la partie significative de la partie fractionnaire. Notez bien ici que *le premier chiffre est toujours 1* et n'est donc pas conservé. Dans la représentation en simple précision (32 bits) l'exposant conservé, relatif aux puissances de 2, occupe 8 bits; la partie fractionnaire occupe les 23 bits restants, car le premier bit à gauche est utilisé comme *bit de signe*. Cette norme **n'utilise pas la représentation en complément à deux**.

Il existe, en fait, deux représentations possibles pour un nombre réel selon la norme IEEE. Elles sont basées sur les mêmes principes. Dans la première forme, comme nous venons de le voir, on divise les 32 bits d'un long mot mémoire en trois parties: le bit de gauche est utilisé pour le signe du nombre (0 pour plus et 1 pour moins), les 8 bits suivants comprennent l'exposant et les 23 bits restants reçoivent la mantisse.



La partie exposant peut prendre une valeur allant de 00 à FF. On remarque que ces valeurs sont considérées être toujours positives. Comme on veut pouvoir avoir des exposants négatifs, on divise ce domaine en deux: les valeurs inférieures allant de 00 à 7E seront considérées comme négatives tandis que celles supérieures à 7F seront considérées comme positives, et que 7F représentera la valeur zéro. La valeur hexadécimale 7F est appelée *pôle;* elle représente la valeur qu'il faut ajouter à l'exposant vrai avant de le ranger dans le champ exposant du nombre. La mantisse comprend les chiffres binaires de la partie fractionnaire. Le nombre -17.25, que nous avons exprimé plus tôt sous la forme -0.1000101×2^5 , ou mieux, la forme normalisée -1.000101×2^4, sera représentée par la valeur hexadécimale:

`C18A0000`

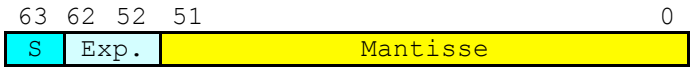
soit en binaire: `11000001100010100000000000000000`.
Le premier bit à gauche est le bit de signe: bit 1 (nombre négatif) ; les 8 bits suivants (83 en hexadécimal) représentent l'exposant 4 (+ le pôle 127_{10} ou $7F_{16}$). Les bits restants représentent la partie fractionnaire de la forme normalisée. Cette forme correspond à la représentation des *nombres réels courts*.

Dans cette représentation, l'exposant maximum semble donc être 128, ce qui donnerait une valeur maximum d'environ 2^{128}, soit approximativement 10^{38} ; en réalité, il n'est égal qu'à 127 (voir plus bas). L'exposant minimum semble être –127, ce qui donnerait une valeur dans le voisinage de 10^{-38} ; en réalité, il est égal à –126 (voir plus bas).

Exemples

Base 10	Base 2	Représentation interne
0,5	1.0×2^{-1}	3F000000
1,0	1.0×2^{0}	3F800000
0,0	0.0	00000000
-15,0	-1.111×2^{3}	C1700000
59,25	1.1101101×2^{5}	426D0000

Les *nombres réels longs* occupent 64 bits et les 32 bits additionnels prolongent la mantisse qui occupe alors 52 bits, car l'exposant est étendu à 11 bits.



Dans cette représentation en "double précision", le pôle vaut 1023_{10} ou $3FF_{16}$ et l'exposant maximum est 1023 ce qui donne une valeur maximum d'environ 2^{1023} soit approximativement 10^{304}.

Exemples

Base 10	Base 2	Représentation interne
6,5859375	1.101001011×2^{2}	401A580000000000
0,1	$1.10011001100 \times 2^{-4}$	3FB9999999999999

11.2.1 Exercices

1. Représenter les nombres hexadécimaux suivants en représentation réelle sur 32 bits.
   a) $AB1.234 \times 16^{3}$
   b) $0.12ABC34 \times 16^{15}$
   c) $64.532A \times 16^{-8}$
   d) $0.BCDEF \times 16^{10}$
   e) $A1.B2C3 \times 16^{-3}$
2. Représenter les nombres hexadécimaux suivants en représentation réelle sur 64 bits
   a) $123456789ABCDEF \times 16^{22}$
   b) $FEDCBA.9876543210 \times 16^{-12}$
   c) $0.123456789AB \times 16^{-20}$
   d) $0.00000AECDEF \times 16^{8}$
   e) $1234567.89ABC \times 16^{-7}$
3. Représenter les nombres réels suivants en notation scientifique usuelle (base 16).
   a) 0A123456
   b) C643210A
   c) 45AB12C0
   d) EFABCDEF
   e) FFFFFFFF
4. Représenter les nombres réels suivants en notation scientifique usuelle (base 16).
   a) B2123456 789ABCDE
   b) 12345678 9ABCDEF0
   c) 81234567 81234567

d) FEDCBA98 76543210
e) 6543210F EDCBA987

5. Convertir les nombres hexadécimaux suivants en nombres décimaux.
   a) 0.00A
   b) 0.1234
   c) 0.A
   d) 12.AB
   e) AEC.123
6. Convertir les nombres décimaux suivants en nombres hexadécimaux.
   a) 0.1250
   b) 0.22900390625
   c) 0.9375
   d) 9876.5
   e) 8192.0940185546875
7. Deux nombres X et Y ont la représentation réelle suivante:
   X: C7A40000 et Y: 402C0000
   Déterminer la représentation flottante de X+Y et de X*Y.

11.3 Valeurs spéciales

Avec la représentation choisie, il est nécessaire de se doter de quelques valeurs spéciales, afin d'être en mesure de vérifier et de traiter tous les cas.

La droite des nombres réels est continue et infinie en mathématiques, et, entre deux nombres réels, il existe une infinité de valeurs. Lorsqu'on opère en informatique, avec la représentation choisie pour les valeurs réelles, on sait que l'infini n'existe plus et que la droite des réels n'est ni infinie, ni continue.

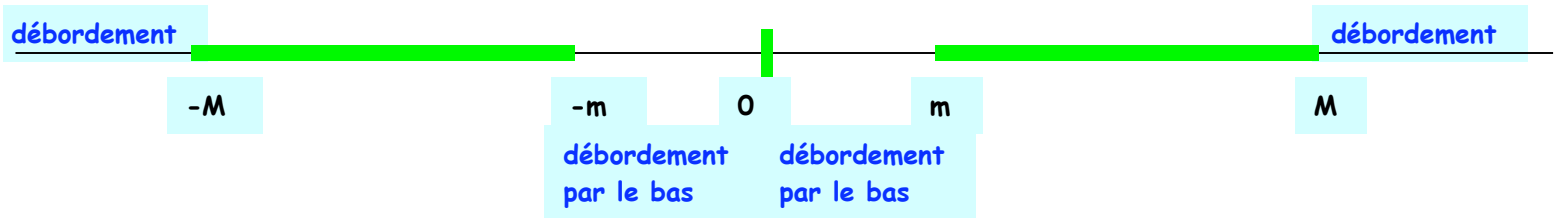


Figure 11.1 Droite des réels en informatique

On retrouve sur cette droite (figure 11.1) quatre zones de débordement : valeurs négatives ayant une valeur absolue trop grande (à gauche de la figure, *overflow*), des valeurs positives trop grandes (à droite de la figure, *overflow*), des valeurs négatives ayant une valeur absolue trop petite (à gauche du zéro, *underflow*) et des valeurs positives trop petites (à droite du zéro, *underflow*). Les valeurs limites M et m seront fixées ci-dessous.

Valeur zéro
Cette valeur ne peut pas avoir de représentation normalisée, puisqu'elle ne possède aucun bit 1 dans sa représentation binaire. Selon la norme, une représentation ne comprenant que des bits zéro serait normalement interprétée comme non nulle, puisque le bit 1 caché de la norme lui serait automatiquement ajouté, et la partie exposant serait considérée comme étant minimum, donc la valeur

00000000000000000000000000000000 (ou en hexadécimal 0X00000000) serait interprétée comme $1.00000000000000000000000 \times 2^{-127}$, soit environ 10^{-38}, ce qui serait la plus petite valeur positive.

En fait, on réserve cette représentation de 32 bits nuls pour la valeur zéro positif. Comme le même raisonnement s'applique à un bit non nul suivi de 31 bits nuls, on réserve cette seconde représentation pour –0. Ces deux choix pour représenter zéro, en hexadécimal 0X00000000 ou 00000000000000000000000000000000, et 10000000000000000000000000000000 (ou 0X80000000), forcent la plus petite valeur positive à 00000000000000000000000000000001, soit $1.00000000000000000001 \times 2^{-127}$, et, de la même façon, la plus grande valeur négative à $-1.00000000000000000001 \times 2^{-127}$.

Infini

On utilise la valeur infinie pour les valeurs qui se trouvent dans les zones de débordement. Si le résultat d'une opération produit une valeur dans la zone de débordement positif, on utilise la valeur infinie positive, soit : 01111111100000000000000000000000 (ou 0X7F800000). S'il s'agit d'une valeur négative trop grande en valeur absolue, on utilise la valeur infinie négative, soit : 11111111100000000000000000000000 (ou 0XFF800000).

NaN

Un ensemble particulier de bits a été défini pour représenter des valeurs qui ne sont pas des nombres. Ces valeurs, appelé NaN pour *not a number*, sont utilisées pour indiquer des opérations arithmétiques interdites. Par exemple, la racine carrée d'un nombre négatif ou une division par zéro produiraient cette valeur : 011111111 suivi d'un reste non nul, ou celle-ci : 111111111 suivi d'un reste non nul.

Les deux représentations pour l'infini et les valeurs qui ne sont pas des nombres utilisent la plus grande valeur de l'exposant, ce qui réduit l'intervalle des valeurs qu'on peut ranger. Sans ces restrictions, la plus grande valeur positive serait 01111111111111111111111111111111, soit $1.11111111111111111111 \times 2^{128}$, soit environ 2^{129}, ou un peu plus de 10^{38}. Avec ces restrictions, la plus grande valeur positive possible sera : 01111111011111111111111111111111 (ou 0X7F7FFFFF), soit $1.11111111111111111111 \times 2^{127}$, soit environ 2^{128}, ou environ 10^{38}.

Nombres dénormalisés

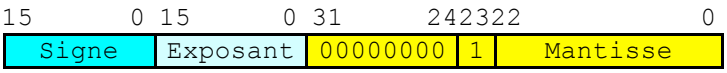
Les valeurs minima, indiquées plus haut dans la section sur la valeur zéro, peuvent être encore diminuées si on permet aux valeurs de ne pas respecter la norme et d'avoir un bit caché *nul*. Ceci permet d'obtenir des valeurs encore bien plus petites. Comme pour la valeur zéro, les valeurs dénormalisées auront un exposant nul; le plus petit nombre normalisé sera donc 00000000100000000000000000000000 soit $1.00000000000000000000 \times 2^{-126}$, soit un peu inférieur à 10^{-38}. Les valeurs dénormalisées iront de 0X00000001 ou 00000000000000000000000000000001 à 0X007FFFFF ou 00000000011111111111111111111111, ce qui correspond à $0.00000000000000000001 \times 2^{-126}$, et $0.11111111111111111111 \times 2^{-126}$. La plus petite valeur est aux environs de 10^{-45}, ce qui est bien mieux que ce que la valeur normalisée d'exposant nul aurait donné (10^{-38}); la seconde valeur est voisine de la plus petite valeur normalisée sans exposant nul que nous avons vu plus haut. Bien entendu, on retrouve les mêmes valeurs, au signe près, dans le domaine négatif.

Toutes ces valeurs se retrouvent, ajustées selon les nombres de bits de leurs parties exposant et mantisse, dans la représentation des réels sur 64 bits.

11.4 Format externe des nombres réels

Les opérations d'arithmétique réelle n'existant pas dans l'ordinateur PEP 8, il est nécessaire de les simuler par programme. Pour cette raison, on peut écrire un certain nombre de sous-programmes qui permettent de manipuler ces quantités. Le format compact de la représentation des nombres réels ne permet pas l'utilisation efficace des instructions de PEP 8 pour réaliser les instructions d'arithmétique réelle. Dans les formats simple et double, le premier bit 1 de la mantisse caché, la représentation du signe, et la compression de la mantisse sont autant de problèmes à contourner. Pour améliorer l'efficacité des opérations logicielles, on utilise une représentation décompressée des réels, que l'on appelle *format externe* ou également représentation étendue

Ce format externe utilise 4 mots pour représenter un nombre réel: le premier mot pour représenter le signe (0 ou 1), un second mot pour représenter l'exposant (de -127 à 128) et un double mot pour représenter la mantisse de 23 bits que l'on fait précéder du bit caché.



Exemples:

Nombre	Format interne	Format externe
3.0	40400000	0000 0001 00C00000
-0.1875	BE400000	0001 FFFD 00C00000
12.5	41480000	0000 0003 00C80000

Les premiers sous-programmes de manipulation de valeurs réelles doivent donc permettre de transformer les valeurs réelles du format interne au format externe, et inversement. Nous examinerons ici les deux exemples de ces sous-programmes de conversion.

La Figure 11.2 donne l'exemple du sous-programme PackFloat, qui accepte un nombre réel en format externe (dont les quatre parties sont empilées) et retourne un nombre réel simple dans le nombre dont l'adresse est empilée comme premier paramètre. L'exposant est isolé et on lui ajoute le pôle 127. Le signe est remplacé par le bit correspondant, qui est placé en avant de l'exposant. La mantisse, à laquelle on a enlevé le bit 1 de tête, est rangée dans les 23 bits les moins significatifs du registre.

```
;------- PackFl -------
;PackFloat: convertir de la représentation étendue (4 mots) située
;sur la pile à la représentation IEEE 754 sur 32 bits dans le
;nombre réel repéré par son adresse sur la pile (1er paramètre).
;Pile inchangée.  Le nombre maximum positif est 7F7FFFFF; pour un nombre
;supérieur on retourne l'infini 7F800000. Si la valeur du nombre est
;inférieure à la valeur minimum positive 00800000 on retourne zéro.
pkVieuxX:.EQUATE   0        ; Sauvegarde registre X
pkVieuxA:.EQUATE   2        ; Sauvegarde registre A
pkAdRet: .EQUATE   4        ; Adresse de retour
pkExpo:  .EQUATE   6        ; Exposant
pkFract1:.EQUATE   8        ; Début fraction
pkFract2:.EQUATE   10       ; Fin fraction
pkSigne: .EQUATE   12       ; Signe
pkRes:   .EQUATE   14       ; Adresse nombre résultat
;              void PackFloat(float &B, int s1, int F12, int F11, int E1) {
PackFl:   SUBSP  4,i        ;
          STA    pkVieuxA,s ;  sauvegarder registre
          STX    pkVieuxX,s ;  sauvegarder registre
          LDA    0,i        ;
          LDX    0,i        ;
          STA    pkRes,sxf  ;  résultat1 = 0;
```



```
          ADDX   2,i        ;
          STA    pkRes,sxf  ;  résultat2 = 0;
          LDA    pkExpo,s   ;  if(exposant != 0) ||
          BRNE   Pack0      ;
          LDA    pkFract1,s ;     ((mantisse1 != 0) ||
          BRNE   Pack0      ;
          LDA    pkFract2,s ;     (mantisse2 != 0)) {
          BREQ   Pack2      ;
Pack0:    LDA    pkExpo,s   ;
          ADDA   127,i      ;    exposant += 127 ;
          BRLE   Ppetit     ;    vérifier si exposant inférieur au minimum (1)
          CPA    0xFF,i     ;    ou
          BRGE   Pgrand     ;      supérieur au maximum (254)
          LDX    pkSigne,s  ;    if(signe == '-')
          BREQ   Pack1      ;      ajoute signe a l'exposant
          ORA    0x100,i    ;
Pack1:    ASLA              ;    aligne signe et exposant
          ASLA              ;    faire place pour début de fraction
          ASLA              ;       (7 bits)
          ASLA              ;
          ASLA              ;
          ASLA              ;
          ASLA              ;    décale dans partie haute de A
          LDX    pkFract1,s ;    mantisse
          ANDX   0x7F,i     ;    réduite à 23 bits
          STX    pkFract1,s ;    mantisse
          ORA    pkFract1,s ;    combinée à l'exposant
          LDX    0,i        ;
          STA    pkRes,sxf  ;    ranger première moitié
          ADDX   2,i        ;
          LDA    pkFract2,s ;    et deuxième moitié
          STA    pkRes,sxf  ;  }// if
Pack2:    LDA    pkVieuxA,s ;  restaure A
          LDX    pkVieuxX,s ;  restaure X
          RET4              ;}// PackFloat;
Pgrand:   LDA    0x7F80,i   ;  trop grand: infini
          LDX    pkSigne,s  ;  if(signe == '-')
          BREQ   Prange     ;    ajoute signe à l'exposant
          ORA    0x8000,i   ;
          BR     Prange     ;
Ppetit:   LDA    0,i        ;  trop petit: zéro
Prange:   LDX    0,i        ;
          STA    pkRes,sxf  ;
          ADDX   2,i        ;  seconde partie
          LDA    0,i        ;
          STA    pkRes,sxf  ;
          BR     Pack2      ;
```

Figure 11.2 Sous-programme PackFloat

La Figure 11.3 illustre le sous-programme UnpackFloat, qui accepte un nombre réel simple dont l'adresse est transmise sur la pile, le convertit au format externe et retourne les quatre mots du format externe en mémoire sur la pile.

```
;------- UnpackFl -------
;UnpackFloat: étant donnée l'adresse d'un nombre réel
;sur la pile, le décomposer en ses 4 parties
;situées sur la pile.
ufVieuxX:.EQUATE   0        ; Sauvegarde registre X
ufVieuxA:.EQUATE   2        ; Sauvegarde registre A
ufAdRet: .EQUATE   4        ; Adresse de retour
ufExpo:  .EQUATE   6        ; Exposant
ufFract1:.EQUATE   8        ; Début fraction
ufFract2:.EQUATE   10       ; Fin fraction
ufSigne: .EQUATE   12       ; Signe
ufAdReel:.EQUATE   14       ; Adresse nombre réel à convertir
;              void UnpackFloat(float B, int &E1, int &F1, int &F2, int &s1) {
UnpackFl: SUBSP  4,i        ;
```

```
            STA    ufVieuxA,s ;  sauvegarder registre
            STX    ufVieuxX,s ;  sauvegarder registre
            LDA    0,i        ;
            STA    ufSigne,s  ;  signe = +;
            LDX    0,i        ;
            LDA    ufAdReel,sxf;  if(nombre == 0){
            ANDA   0x7FFF,I   ;     attention au -0
            CPA    0,i        ;
            BRNE   Unpack0    ;    {partie 1 du réel}
            ADDX   2,i        ;
            LDA    ufAdReel,sxf;
            BRNE   Unpack0    ;    {partie 2 du réel}
            LDA    0,i        ;
            STA    ufExpo,s   ;    exposant = 0;
            STA    ufFract1,s ;    fraction1 = 0;
            STA    ufFract2,s ;    fraction2 = 0;
            BR     Unpack2    ;  }
Unpack0:    LDX    0,i        ;  else if(nombre < 0) {
            LDA    ufAdReel,sxf;
            BRGE   Unpack1    ;
            LDA    1,i        ;    signe = -;
            STA    ufSigne,s  ;  }
Unpack1:    LDA    ufAdReel,sxf;  {première partie}
            ANDA   0x7f,i     ;   mantisse originale =
            ORA    0x80,i     ;              (1,mantisse);
            STA    ufFract1,s ;   huit premiers bits des 24 bits de mantisse
            LDA    ufAdReel,sxf;  {première partie}
            ASRA              ;  {décalée à droite de 7 bits}
            ASRA              ;
            ASRA              ;
            ASRA              ;
            ASRA              ;
            ASRA              ;
            ANDA   0xFF,i     ;  exposant original (enlève signe)
            SUBA   127,i      ;  exposant vrai;
            STA    ufExpo,s   ;  exposant de 8 bits
            ADDX   2,i        ;
            LDA    ufAdReel,sxf;  deuxième partie telle quelle
            STA    ufFract2,s ;  copie fraction2;
Unpack2:    LDA    ufVieuxA,s ;  restaure A
            LDX    ufVieuxX,s ;  restaure X
            RET4              ;} nettoyer pile et retourner
```

Figure 11.3 Sous-programme UnpackFloat

Le bit de signe est vérifié et la valeur correspondante est rangée dans le mot de signe. La mantisse de 23 bits est copiée et le bit 1 de tête lui est ajouté, avant rangement dans le double mot correspondant. L'exposant est réduit de 127 avant d'être placé dans le mot de l'exposant.

11.5 Normalisation

En utilisant les puissances de la base, on sait qu'on peut représenter un même nombre de plusieurs façons différentes: ainsi 0.4325×10^3 et 0.0004325×10^6 représentent le même nombre décimal. De même, $0.0011001000101001 \times 2^5$ et $1.1001000101001 \times 2^2$ représentent le même nombre. La seule différence est la position du premier chiffre significatif.

Avec la norme IEEE 754, on dit qu'un nombre réel est *normalisé* lorsqu'il est tel que sa partie entière ne soit formée que d'un seul bit égal à 1. Par conséquent, $1.1001000101001 \times 2^2$ est normalisé, tandis que $0.0011001000101001 \times 2^5$ ne l'est pas.

Le sous-programme `Normaliz`, illustré par la Figure 11.4 ci -dessous, prend une mantisse de 24 bits rangée dans un double mot et décale ces bits de façon à ce que le bit 23 soit un bit 1. L'exposant est ajusté selon la direction du décalage. Le sous-programme reçoit la représentation externe d'un nombre réel sur la pile. Il fait appel au sous-programme `Shift` qui effectue le décalage et qui est illustré par la Figure 11.5. L'espace local de `Normaliz` sera directement partagé par le sous-programme `Shift`.

```
;------- Normaliz -------
;Normalisation d'un nombre réel; le bit de gauche de la
;mantisse doit se trouver en position 23; si ce n'est
;pas le cas la mantisse doit être décalée et l'exposant ajusté.
;Les 4 parties du nombre sont sur la pile et y demeurent.
nDecal:  .EQUATE   0         ; Décalage             (zone partagée avec Shift)
nFracD1: .EQUATE   2         ; Début fraction de travail (partagé avec Shift)
nFracD2: .EQUATE   4         ; Fin fraction de travail   (partagé avec Shift)
nVieuxX: .EQUATE   6         ; Sauvegarde registre X
nVieuxA: .EQUATE   8         ; Sauvegarde registre A
nAdRet:  .EQUATE   10        ; Adresse de retour
nExpo:   .EQUATE   12        ; Exposant
nFract1: .EQUATE   14        ; Début fraction
nFract2: .EQUATE   16        ; Fin fraction
nSigne:  .EQUATE   18        ; Signe
;                            ;void Normaliz(int expo, int f1, int f2,int signe){
Normaliz: SUBSP   10,i       ;  espace local sauvegarde
          STA     nVieuxA,s  ;  sauvegarde A
          STX     nVieuxX,s  ;  sauvegarde X
          LDX     1,i        ;  décalage = 1;
          LDA     nFract2,s  ;  copie de travail
          STA     nFracD2,s  ;
          LDA     nFract1,s  ;  if(mantisse1 == 0 ||
          STA     nFracD1,s  ;
          BRNE    Normal0    ;
          LDA     nFracD2,s  ;      mantisse2 == 0)
          BRNE    Normal0    ;
          LDX     0,i        ;    décalage = 0;
          BR      Normal1    ;  else
Normal0:  LDA     nFracD1,s  ;    while(true){
          SUBX    1,i        ;      décalage--;
          ASLA               ;      décale première partie
          BRC     Normal2    ;      if(bit == 1) break;
          STA     nFracD1,s  ;
          LDA     nFracD2,s  ;        décale deuxième partie
          ASLA               ;
          STA     nFracD2,s  ;
          BRC     Ajoute1    ;      if(retenue)
          BR      Normal0    ;
Ajoute1:  LDA     nFracD1,s  ;        récupère bit 1 dans première partie
          ADDA    1,i        ;
          STA     nFracD1,s  ;
          BR      Normal0    ;    }// while
Normal2:  LDA     nFracD2,s  ;    décale deuxième partie
          ASLA               ;
          STA     nFracD2,s  ;
          BRC     Ajoute2    ;    if(retenue){
          BR      Normal3    ;
Ajoute2:  LDA     nFracD1,s  ;
          ADDA    1,i        ;      récupère bit 1 dans première partie
          STA     nFracD1,s  ;
Normal3:  ADDX    8,i        ;      décalage += 8;{octet de tête}
;                            ;  }
Normal1:  STX     nDecal,s   ;  empiler temporaire pour appel Shift
          ADDX    nExpo,s    ;
          STX     nExpo,s    ;  exposant += décalage;
          LDA     nFract1,s  ;
          STA     nFracD1,s  ;  empiler f1 pour appel
          LDA     nFract2,s  ;  empiler f2 pour appel
          STA     nFracD2,s  ;
          CALL    Shift      ;  -->Décale mantisse
          LDA     nFracD1,s  ;  Recopie résultat 1
```

```
STA     nFract1,s  ;
LDA     nFracD2,s  ;  Recopie résultat 2
STA     nFract2,s  ;
LDA     nVieuxA,s  ;  restaure A
LDX     nVieuxX,s  ;  restaure X
ADDSP   10,i       ;  nettoyer pile
RET0               ;}// Normaliz;
```

Figure 11.4 Sous-programme Normalize

Étant donné un nombre réel, représenté en format externe par (0, 4, B00000), nous l'additionnons à lui-même. En notation binaire, ceci nous donne :

$$\begin{array}{rl} & 1.01100000 \times 2^4 \\ + & \underline{1.01100000 \times 2^4} \\ & 10.11000000 \times 2^4 \end{array} \quad \text{soit (0, 5, 1600000)}$$

Notez que la partie fractionnaire possède maintenant 25 bits, ce qui ne correspond pas à la représentation normalisée. Si nous lui appliquons le sous-programme `Normaliz`, nous obtenons la trace suivante. La boucle étiquetée `Normal0` produit les valeurs suivantes pour les deux moitiés de la partie fractionnaire et le registre X :

FracD1-FracD2	Registre X
0160-0000	1
02C0-0000	0
0580-0000	-1
0B00-0000	-2
1600-0000	-3
2C00-0000	-4
5800-0000	-5
B000-0000	-6
6000-0000	-7

Arrivé à l'étiquette `Normal3`, on ajoute 8 au registre X, ce qui donne un décalage égal à 1. Ce décalage est ajouté à l'exposant qui devient 5, et est passé au sous-programme `Shift`. Ensuite on copie la partie fractionnaire originale et on appelle `Shift`.

```
;------- Shift -------
;Décalage effectif de la partie fractionnaire située sur la pile
;avec le décalage pour que le premier bit 1 soit en position 23.
;La pile demeure telle quelle (3 éléments).
sVieuxX: .EQUATE   0         ; Sauvegarde registre X
sVieuxA: .EQUATE   2         ; Sauvegarde registre A
sAdRet:  .EQUATE   4         ; Adresse de retour
sDeca:   .EQUATE   6         ; Décalage
sFracD1: .EQUATE   8         ; Début fraction
sFracD2: .EQUATE   10        ; Fin fraction
;                            void Shift(int dec, int &frac1, int &frac2) {
Shift:    SUBSP   4,i        ;  espace local sauvegarde
          STA     sVieuxA,s  ;  sauvegarde A
          STX     sVieuxX,s  ;  sauvegarde X
          LDA     sFracD1,s  ;  début mantisse
          LDX     sDeca,s    ;  switch(correction exposant){
          BRLT    Shift1     ;       <0: décalage à gauche
          BREQ    Shift2     ;       =0: pas de décalage
Shift0:   LDA     sFracD2,s  ;  case >0: while(true) { //décalage à droite
          ASRA                 ;             deuxième moitié
          ANDA    0x7FFF,i   ;             enlever premier bit au cas où 1
          STA     sFracD2,s  ;
          LDA     sFracD1,s  ;             et première moitié
          ASRA               ;
```



```
          ANDA    0x7FFF,i   ;              enlever premier bit au cas où 1
          STA     sFracD1,s  ;
          BRC     Sajout1    ;              si bit 1 ajouter deuxième moitié
Scont1:   SUBX    1,i        ;              compte--;
          BRNE    Shift0     ;            if(compte == 0) break;
          LDA     sFracD1,s  ;            }// while
          ANDA    0xFF,i     ;            premier octet = 0;
          STA     sFracD1,s  ;
          BR      Shift2     ;
Sajout1:  LDA     sFracD2,s  ;             deuxième moitié
          ORA     0x8000,i   ;             ajouter bit décalé
          STA     sFracD2,s  ;
          BR      Scont1     ;
Shift1:   LDA     sFracD1,s  ;  case <0: while(true) { //décalage à gauche
          ASLA               ;              décale à gauche première moitié
          STA     sFracD1,s  ;
          LDA     sFracD2,s  ;               puis deuxième moitié
          ASLA               ;
          STA     sFracD2,s  ;
          BRC     Sajout2    ;               si bit décalé ajouter autre moitié
Scont2:   ADDX    1,i        ;               compte++ 1;
          BRNE    Shift1     ;             if(compte == 0) break;
          BR      Shift2     ;             }// while;
Sajout2:  LDA     sFracD1,s  ;             première moitié
          ADDA    1,i        ;             ajouter bit décalé
          STA     sFracD1,s  ;
          BR      Scont2     ;
Shift2:   LDA     sVieuxA,s  ;  case =0 : restaure A
          LDX     sVieuxX,s  ;            restaure X
          RET4               ;}// Shift;
```

Figure 11.5 Sous-programme Shift

Arrivés dans le sous-programme `Shift` avec les valeurs indiquées plus haut (décalage et partie fractionnaire originale), ce dernier détermine que le décalage est positif et on effectue la boucle d'étiquette `Shift0` une seule fois (décalage égal à 1). La partie fractionnaire, originellement égale à 01600000, a donc été décalée d'une position vers la droite, donnant 00B00000. Au retour dans le sous-programme `Normaliz`, l'exposant a déjà la bonne valeur et il suffit de recopier la nouvelle valeur de la partie fractionnaire, avant de retourner à l'appelant.

11.6 Conversion d'un nombre entier de 32 bits en un nombre réel de 32 bits

La figure 11.6 présente un petit sous-programme interne qui convertit un nombre entier de 32 bits en un nombre réel de 32 bits. La méthode utilisée est très simple. Elle consiste à tester la valeur absolue de la valeur entière transmise par la pile. Si la valeur est nulle on la conserve telle quelle. Le résultat allant dans l'adresse passée sur la pile, on n'en conserve d'abord que le signe. On recherche ensuite le premier bit 1 du nombre entier positif en faisant des décalages à gauche et en comptant ces décalages. Dès qu'un bit 1 déborde à gauche du registre, on arrête les décalages car on possède la mantisse: le 1 de la partie entière vient d'être éliminé. Cette mantisse est alors placée à sa position finale par un décalage à droite de 9 positions (on doit en effet laisser la place pour 9 bits à la gauche du résultat: le bit de signe et les 8 bits de l'exposant). Cette valeur est ajoutée au résultat.

On calcule ensuite l'exposant, en remarquant que pour un entier, le point décimal est situé après le dernier chiffre du nombre entier, alors que dans la représentation réelle il précède le premier chiffre de la mantisse. La valeur de cet exposant est donc égale à 31 moins le nombre de zéros qui précédaient le premier bit non nul de l'entier. Cet exposant vrai doit ensuite être modifié par l'addition du pôle

127. Une fois calculé, cet exposant est décalé à sa position finale et ajouté au résultat. On notera comment le résultat est composé par utilisation des fonctions logiques. Le signe du nombre entier est conservé dans la représentation réelle par un ET logique entre le nombre et un masque ne comportant qu'un seul bit un. La mantisse est ensuite ajoutée à ce résultat par un OU logique qui ne modifie que les 23 bits de la mantisse. De même l'exposant est ajouté au résultat par un OU logique qui ne modifie que les 8 bits de la position de l'exposant dans le résultat.

Ainsi l'entier hexadécimal 12345678 produit dans le résultat d'abord la valeur 00000000, puis dans le registre la valeur 23456780, qui devient 0011A2B3 (la valeur précédente décalée de 9 bits), et le résultat devient 0011A2B3. L'exposant calculé est (31-3+127) soit 155 ou 9B hexadécimal et le résultat est finalement 4D91A2B3. On remarquera que cette conversion ne peut donner de valeur comprise entre zéro et un: l'exposant sera toujours positif.

```
;------- EntierR -------
;Conversion d'un nombre entier de 32 bits dont
;la valeur se trouve sur la pile en un nombre réel
;dont l'adresse se trouve sur la pile.
;La pile est nettoyée des deux paramètres.
eDecal: .EQUATE 0         ; Décalages
eVieuxX:.EQUATE 2         ; Sauvegarde registre X
eVieuxA:.EQUATE 4         ; Sauvegarde registre A
eAdRet: .EQUATE 6         ; Adresse de retour
ePart1: .EQUATE 8         ; Début entier
ePart2: .EQUATE 10        ; Fin entier
eNomb:  .EQUATE 12        ; Adresse nombre réel
;
EntierR:  SUBSP  6,i        ;
          STA    eVieuxA,s  ;  sauvegarder registre
          STX    eVieuxX,s  ;  sauvegarder registre
          LDX    0,i        ;
          LDA    ePart1,s   ;
          ANDA   0x8000,i   ;  placer signe dans le résultat
          STA    eNomb,sxf  ;
          LDA    ePart1,s   ;  d1 = N;
          BRGT   Epositif   ;
          BRLT   Enega      ;
          LDA    ePart2,s   ;  si nul vérifier 2e partie
          BREQ   Ezero      ;  zéro
          BR     Epositif   ;  première partie nulle alors positif
Enega:    LDA    ePart1,s   ;  if(N < 0) {
          NOTA              ;    partie fraction A = -partie fraction A;
          STA    ePart1,s   ;
          LDA    ePart2,s   ;
          NEGA              ;
          STA    ePart2,s   ;  }
          BRNE   Epositif   ;
          LDA    ePart1,s   ;  if(seconde partie nulle]
          ADDA   1,i        ;    ajuster complément à 2 partie 1
          STA    ePart1,s   ;
Epositif: LDX    1,i        ;  décalage = 1;
                            ;  while(true) {
Echerbit: SUBX   1,i        ;    décalage--;
          LDA    ePart1,s   ;
          ASLA              ;    décale N1 de un bit à gauche
          STA    ePart1,s   ;
          BRC    Eapres     ;  if( bit non nul) break;
          LDA    ePart2,s   ;
          ASLA              ;    décale N2 de un bit à gauche
          STA    ePart2,s   ;
          BRC    Eajout     ;
          BR     Echerbit   ;  }// while
Eapres:   LDA    ePart2,s   ;
          ASLA              ;    décale N2 de un bit à gauche
          STA    ePart2,s   ;
```



```
          BRC    Egauche    ;
Econt:    STX    eDecal,s   ;    décalages pour exposant
          LDX    9,i        ;  for(int i = 9; i != 0; i--) {
Eboucle:  LDA    ePart2,s   ;    redécale à droite
          ASRA              ;
          ANDA   0x7FFF,i   ;    enlever premier bit
          STA    ePart2,s   ;    partie 2
          LDA    ePart1,s   ;    et partie 1
          ASRA              ;
          STA    ePart1,s   ;
          BRC    Edroite    ;    transférer bit
Eredecal: SUBX   1,i        ;
          CPX    0,i        ;
          BRNE   Eboucle    ;  }// for
          LDA    ePart1,s   ;
          ANDA   0x7F,i     ;  garder les 7 bits de la fraction
          STA    ePart1,s   ;
          LDA    eDecal,s   ;  décalages
          ADDA   158,i      ;  exposant = 31 - décalage + 127 ;
          ASLA              ;  décale exposant à sa position finale (7 positions)
          ASLA              ;
          ASLA              ;
          ASLA              ;
          ASLA              ;
          ASLA              ;
          ASLA              ;
          ORA    ePart1,s   ;  ajoute début fraction
          LDX    0,i        ;
          ORA    eNomb,sxf  ;  avec le signe
          STA    eNomb,sxf  ;
          LDX    2,i        ;
          LDA    ePart2,s   ;  deuxième moitié
          STA    eNomb,sxf  ;
Ezero:    LDA    eAdRet,s   ;  effacer paramètres
          STA    eNomb,s    ;
          LDA    eVieuxA,s  ;  restaure A
          LDX    eVieuxX,s  ;  restaure X
          ADDSP  12,i       ;  nettoyer pile
          RET0              ;}// EntierReel;
Eajout:   LDA    ePart1,s   ;  faire glisser le bit de partie 2
          ADDA   1,i        ;      à partie 1
          STA    ePart1,s   ;
          BR     Echerbit   ;
Egauche:  LDA    ePart1,s   ;  faire glisser le bit de partie 2
          ADDA   1,i        ;      à partie 1
          STA    ePart1,s   ;
          BR     Econt      ;
Edroite:  LDA    ePart2,s   ;  faire glisser le bit de partie 1
          ORA    0x8000,i   ;      à partie 2
          STA    ePart2,s   ;
          BR     Eredecal   ;
```

Figure 11.6 Conversion d'entier à réel

11.7 Conversion d'un nombre réel de 32 bits en un nombre entier de 32 bits

La figure 11.7 présente un petit sous-programme interne qui convertit un nombre en représentation réelle sur 32 bits, en un nombre entier. La méthode utilisée est là encore assez simple, bien qu'un peu longue. L'adresse du nombre à convertir se trouve sur la pile. On vérifie alors que sa valeur absolue est bien inférieure à 2^{31}, sinon la conversion ne peut être faite puisque le résultat entier exigerait plus de 32 bits. Si le nombre est trop grand, le sous-programme affiche un message, mais pourrait tout aussi bien retourner un code d'erreur. Si le nombre vaut zéro, on retourne un résultat nul, sans faire de calculs savants.

La conversion proprement dite commence par isoler la mantisse et par lui ajouter un bit 1 de tête. L'exposant est ensuite isolé et on calcule l'exposant vrai. Si cet exposant est négatif, le résultat est inférieur à 1 en valeur absolue et est donc zéro. Le décalage à appliquer à la mantisse est calculé à partir de l'exposant vrai et ce calcul tient compte du fait que le point décimal est situé entre les bits 22 et 23. La mantisse est décalée à droite ou à gauche en fonction de la valeur de l'exposant réduite de 23, pour donner la valeur absolue du résultat. Ce résultat est alors retourné tel quel, si le nombre à convertir était positif, sinon, on retourne son négatif (représentation en complément à deux).

```
;------- ReelEnt -------
;Conversion d'un nombre réel sur 32 bits dont
;l'adresse se trouve sur la pile en un nombre entier
;dont l'adresse se trouve également sur la pile.
;La pile est nettoyée des deux paramètres.
rExpo:  .EQUATE   0         ; exposant
rVieuxX:.EQUATE   2         ; Sauvegarde registre X
rVieuxA:.EQUATE   4         ; Sauvegarde registre A
rAdRet: .EQUATE   6         ; Adresse de retour
rNomb:  .EQUATE   8         ; Adresse nombre réel
rPart1: .EQUATE   10        ; Début entier
rPart2: .EQUATE   12        ; Fin entier
;
ReelEnt:  SUBSP  6,i        ;
          STA    rVieuxA,s  ;  sauvegarder registre
          STX    rVieuxX,s  ;  sauvegarder registre
          LDX    0,i        ;
          LDA    rNomb,sxf  ;  nombre réel
          ANDA   0x7FFF,i   ;    au cas où -0
          BRNE   Rnonzero   ;  if(N != 0) {
          LDX    2,i        ;
          LDA    rNomb,sxf  ;    deuxième moitié
          BREQ   Rzero      ;
          LDX    0,i        ;
          LDA    rNomb,sxf  ;    première moitié
Rnonzero: ANDA   0x7FFF,i   ;    if(abs(N) < 2**31) {
          CPA    0x4F00,i   ;      4F0 -> 9E -> 7F+1F -> 1F = 31 décimal
          BRGE   Rgrand     ;
          LDA    rNomb,sxf  ;      nombre réel
          ANDA   0x007F,i   ;      début mantisse
          ORA    0x0080,i   ;      ajouter bit 1 de la partie entière
          STA    rPart1,s   ;
          ADDX   2,i        ;
          LDA    rNomb,sxf  ;
          STA    rPart2,s   ;      deuxième partie mantisse
          LDX    0,i        ;
          LDA    rNomb,sxf  ;
          ANDA   0x7F80,i   ;      partie exposant;
          ASRA              ;      poussée à droite (7 positions)
          ASRA              ;
          ASRA              ;
          ASRA              ;
          ASRA              ;
          ASRA              ;
          ASRA              ;
          ANDA   0xFF,i     ;
          SUBA   127,i      ;      exposant vrai;
          STA    rExpo,s    ;
          BRLT   Rzero      ;   si négatif résultat nul
          LDX    23,i       ;   X = 23 - exposant vrai ;
          SUBX   rExpo,s    ;   if(X > 0) {
          BRLT   Rneg       ;
Rboucd:   CPX    0,i        ;     while(X != 0) {
          BREQ   Rcompose   ;
          LDA    rPart2,s   ;       décalage droite du nombre
          ASRA              ;       partie 2
          ANDA   0x7FFF,i   ;       éliminer bit gauche
          STA    rPart2,s   ;
          LDA    rPart1,s   ;       partie 1
```



```
          ASRA             ;
          ANDA   0x7FFF,i  ;       éliminer bit gauche
          STA    rPart1,s  ;
          BRC    Rajoutd   ;       si bit 1 perdu
Rcontd:   SUBX   1,i       ;
          BR     Rboucd    ;     }// while
Rajoutd:  LDA    rPart2,s  ;
          ORA    0x8000,i  ;     ajouter bit de tête
          STA    rPart2,s  ;
          BR     Rcontd    ;   }
Rneg:     NOP0             ;   else {
          NEGX             ;     X = -X;
Rboucg:   CPX    0,i       ;     while(X != 0) {
          BREQ   Rcompose  ;
          LDA    rPart1,s  ;       décalage gauche du nombre
          ASLA             ;       partie 1
          STA    rPart1,s  ;
          LDA    rPart2,s  ;
          ASLA             ;       partie 2
          STA    rPart2,s  ;
          BRC    Rajoutg   ;       si bit 1 perdu
Rcontg:   SUBX   1,i       ;
          BR     Rboucg    ;     }// while
Rajoutg:  LDA    rPart1,s  ;
          ADDA   1,i       ;     ajouter bit de queue
          STA    rPart1,s  ;
          BR     Rcontg    ;
Rcompose: LDX    0,i       ;  }
          LDA    rNomb,sxf ;  nombre réel
          BRGT   Rfini     ;  if(N < 0) {
          LDA    rPart1,s  ;
          NOTA             ;     entier = -(entier);
          STA    rPart1,s  ;
          LDA    rPart2,s  ;
          NEGA             ;     complément à 2
          STA    rPart2,s  ;
          BRNE   Rfini     ;    if(seconde partie nulle) {
          LDA    rPart1,s  ;
          ADDA   1,i       ;      ajuster complément à 2 partie 1
          STA    rPart1,s  ;
          BR     Rfini     ;  }
Rgrand:   STRO   Averti,d  ;  cout >> "Nombre réel trop grand ne peut être converti."
          CHARO  LF,i      ;       >> endl;
Rzero:    LDA    0,i       ;
          STA    rPart1,s  ;  entier = 0;
          STA    rPart2,s  ;  entier = 0;
Rfini:    LDA    rAdRet,s  ;  effacer paramètre
          STA    rNomb,s   ;
          LDA    rVieuxA,s ;  restaure A
          LDX    rVieuxX,s ;  restaure X
          ADDSP  8,i       ;  nettoyer pile
          RET0             ;}// ReelEntier;
```

Figure 11.7 Conversion de réel à entier

