Ateliers

Inf 2170

Organisation des ordinateurs et assembleur

Copyright©Mario Latendresse

Dernière révision, août 1999

Table des matières

1 Histoire de bits **1**

1.1 Les entiers en binaire 1
1.2 Généralité sur la représentation binaire 2
1.3 La mémoire centrale 3
1.4 Le texte en mémoire centrale 4

2 Assemblage et exécution d'un programme **7**

2.1 Codification des instructions 7
2.2 Assemblage 8
2.3 Assemblage et chargement 9
2.4 Édition des liens 9

3 Expression arithmétique entière **11**

3.1 Addition binaire et bits NZVC 11
3.2 Évaluation d'expression arithmétique entière 12
3.3 Opérations arithmétiques particulières 13

4 Instructions conditionnelles **15**

4.1 Instructions de base 15
4.2 Instruction SI 16
4.3 Instruction Tantque 17
4.4 Programmes complets 18

5 Sous-routine **19**

5.1 Sous-routine avec passage des paramètres par les registres . . 19
5.2 Retourner un résultat par les bits du CCR 20
5.3 Passage d'une sous-routine à une sous-routine 21

6 Les vecteurs **23**
6.1 Chaîne de caractères 23
6.2 Vecteur de nombres entiers 23

7 Matrices **27**
7.1 Opérations de base 27
7.2 Traitement sur des matrices 27

8 Passage des paramètres sur la pile et espace local **29**

9 Les conversions pour les entrées et les affichages d'entiers. **33**
9.1 Sous-routines classiques de conversion 33
9.2 Un petit problème de comptage de bits 33

10 Manipulation de bits **35**
10.1 Opérations de base 35
10.2 Chaîne de bits 36
10.3 Graphique, matrice de bits 36
10.4 Ensembles représentés par des chaînes de bits 38

11 Représentation de nombres fractionnaires en virgule-fixe **39**
11.1 Éléments 39
11.2 Approximation de fonctions 41
11.3 L'utilisation de fractions comme valeurs non-entières 42

12 La codification en virgule-flottante IEEE-754 **43**
12.1 Éléments 43
12.2 Traitement en IEEE 45

13 Récursivité au niveau machine **47**
13.1 Évaluations récursives 47

A Corrigé partiel **49**

Préface

Qui pourrait affirmer que le savoir-faire d'un domaine peut s'acquérir sans pratique?

Rien ne remplace la pratique et la réflexion personnelles pour mieux consolider un apprentissage récent. Ce petit manuel d'exercices a été composé dans le but d'offrir des réflexions sur différents aspects abordés dans le cours Inf2170, *organisation des ordinateurs et assembleur*.

Un corrigé partiel est fourni à la fin de ce manuel. Toutefois, je vous invite à le consulter qu'après avoir cogité suffisament un exercice pour comparer votre solution. Autrement, l'exercice peut devenir futile et vous ne retireriez qu'une partie des bénéfices.

Certains des exercices ont été tirés d'examens. Aucun doute, cela en augmente leur pertinence! D'autres proviennent de travaux pratiques.

L'ordre des chapitres ne correspond pas nécessairement à l'ordre des thèmes abordés durant la session. En fait, les exercices ont été regroupés sous différentes bannières pour mieux les repérer. Les exercices à faire à chaque semaine vous seront communiqués en classe. Probablement que d'autres exercices viendront s'ajouter pour mieux travailler des sujets précis.

Bon travail!

Mario Latendresse
Août 1999
email: `latendresse@iro.umontreal.ca`
`http://www.iro.umontreal.ca/~latendre/inf2170`

Et le langage machine,
qui décide de notre destin en sous-main
et pour n'importe quel environnement?
Eh bien, cela relève de l'ancien Testament,
du Talmud et de la Cabale.

Umberto Eco
Comment voyager avec un saumon

Many readers are no doubt thinking, "Why does Knuth replace MIX by another machine instead of just sticking to a high-level programming language? Hardly anybody uses assemblers these days."

If I did use a high-level language, what language should it be? In the 1960s I would probably have chosen Algol W; in the 1970s, I would then have had to rewrite my books using Pascal; in the 1980s, I would surely have changed everything to C; in the 1990s, I would have had to switch to C++ and then probably to Java. In the 2000s, yet another language will no doubt be de rigueur. I cannot afford the time to rewrite my books as languages go in and out of fashion; languages aren't the point of my books, the point is rather what you can do in your favorite language. My books focus on timeless truths.

Donald E. Knuth
Extrait de sa page web personnelle, Mars 1999

Chapitre 1

Histoire de bits

Le bit est omniprésent au niveau du langage machine de l'ordinateur. Les nombres entiers, nombres non-entiers, instructions, textes, images, sont représentés sous la forme de séquence de bits. Pour bien comprendre le fonctionnement de l'ordinateur, au niveau du langage de l'UCT, il est essentiel de bien manipuler les représentations de base des entiers et du texte.

1.1 Les entiers en binaire

EXERCICE 1.1.1

Convertissez en hexadécimal les nombres décimaux suivants : a) 435 b) 234 c) 10 d) 8 e) 100 f) 255 g) 1 023 h) 566 i) 543 j) 9 k) 99 l) 200

EXERCICE 1.1.2

Convertissez en binaire complément à deux, sur 8 bits et 16 bits, les nombres décimaux suivants, et dites s'il y a débordement arithmétique : a) 1 234 b) 2 034 c) -56 d) 345 e) 36 000 f) -432 g) 1 009 h) 64 i) -5 j) 1 256 k) -1 024 l) 2 376

EXERCICE 1.1.3

Convertissez en binaire pur les nombres décimaux suivants.

a) 255 b) 16 c) 512 d) 2 654 e) 765

EXERCICE 1.1.4

Donnez les valeurs suivantes en décimal et en hexadécimal en supposant que

les chaînes de bits représentent des nombres entiers codés en complément à deux sur 16 bits.

a) 0000 1100 1001 0101

b) 1001 1101 1110 1010

c) 0001 1001 1001 0100

d) 0111 0000 1011 1100

EXERCICE 1.1.5

Soit une chaîne de n bits x_i représentant un nombre entier codé en complément à deux. Qu'arrive-t-il à la valeur de ce nombre si a) un bit quelconque est mis à 0, b) un bit quelconque est mis à 1?

EXERCICE 1.1.6

Supposons que le processeur d'un ordinateur augmente de 1, à chaque nanoseconde, une variable de 64 bits. Cette variable peut contenir des nombres de 0 à $2^{64} - 1$. Si le processeur démarre la variable à zéro, combien de temps cela prendra avant d'atteindre la valeur $2^{64} - 1$? Même question mais pour une variable de 32 bits.

1.2 Généralité sur la représentation binaire

EXERCICE 1.2.1

Combien de bits sont-ils nécessaires pour représenter les nombres entiers de 0 à 1 000 en binaire pur? ; des nombres entiers de 0 à 100 000? ; des nombres entiers de 0 à 1 000 000? ; des nombres entiers de 0 à 1 000 000 000? Donnez une expression arithmétique générale exprimant le nombre de bits nécessaire pour représenter les nombres entiers de 0 à n (en binaire pur).

EXERCICE 1.2.2

Combien de chaînes de bits différentes peut-on avoir dans un mot mémoire de quatre octets?

EXERCICE 1.2.3

Quelle est la caractéristique des nombres pairs écrits en binaire? Des nombres impairs?

EXERCICE 1.2.4

La céréale *Post Sugar Crisp* offrait à ses habitués du petit déjeuner, une surprise sous la forme d'un petit jeu de magicien. Ce jeu est composé de sept cartes différentes sur lesquelles sont inscrits plusieurs nombres entiers compris entre 1 et 127 inclusivement.

Le jeu se déroule de la façon suivante. Une personne, dont vous ne connaissez pas l'âge, consulte les sept cartes et celle-ci vous remet seulement les cartes sur lesquelles son âge est inscrit. À l'aide de ces cartes, par un coup d'oeil aux nombres des coins supérieurs gauches, suivi d'une addition rapide de ces nombres, vous déterminez l'âge exact de la personne. Élaborez la conception de ces cartes, ou en d'autres mots, déterminez les sous-ensembles de nombres à inscrire sur chacune des cartes.

1.3 La mémoire centrale

EXERCICE 1.3.1

Est-ce qu'un mot mémoire peut ne « rien » contenir?

EXERCICE 1.3.2

Une suite de 100 mots de 32 bits occupe combien d'octets?

EXERCICE 1.3.3

Un vecteur de 100 mots de 16 bits débute à l'adresse 1000. Quelle est l'adresse du dernier mot de ce vecteur?

EXERCICE 1.3.4

Que signifie, dans le contexte des UCT, le terme « big-endian »? ; le terme « little-endian »?

EXERCICE 1.3.5

Soit un processeur « big-endian» comme les MC68k. Si l'octet 3450 contient 1 et l'octet 3451 contient 2, quelle est la valeur du nombre entier contenue dans le mot de 16 bits de l'adresse 3450, en considérant une codification en complément à deux? Quelle est la valeur si ce mot est lu par un processeur « little-endian »?

EXERCICE 1.3.6

Donnez le contenu de la mémoire centrale, octet par octet et en décimal,

à partir de l'étiquette `x` étant donnée la déclaration suivante et le fait que la disposition mémoire suit la méthode « big-endian ». Même question mais pour une machine « little-endian ».

```
x    dc.b  0,1,'a','A','aA'
     dc.w  1,256,'a','A'
     dc.l  1,876,'a','A'
     dc.b  'Fini',10,13
     dc.w  -1,-2
```

EXERCICE 1.3.7

Indiquez le contenu de la mémoire centrale, octet par octet et en décimal, à partir de l'étiquette `x`. (Machine big-endian)

```
x      dc.l    -500,200
       dc.b    '!33!'
       dc.l    1222,-1,512
```

EXERCICE 1.3.8

Soient les définitions suivantes.

```
          data
somme:    dc.l   12
car:      ds.b   1
carre:    dc.w   10
y:        dc.w   8
          end
```

a) Donnez les valeurs, en décimal et octet par octet, du contenu de la mémoire centrale pour la section `data` à partir de l'étiquette `somme`. (La disposition suit la méthode *big-endian*).

b) Donnez le contenu du registre `d0.l`, en décimal, après l'exécution de chacune des instructions suivantes : (si le contenu est inconnu indiquez-le par le symbole —)

```
move.l   #1,d0
move.b   #'a',d0
move.b   car,d0
move.w   carre,d0
move.l   y,d0
```

1.4 Le texte en mémoire centrale

EXERCICE 1.4.1

Les chiffres hexadécimaux 3139383720402031393936 représentent une chaîne de caractères codés ASCII. Quelle est cette chaîne?

EXERCICE 1.4.2

Quelle est la différence entre les codes ASCII des lettres majuscules et les codes ASCII des lettres minuscules?

EXERCICE 1.4.3

En donnant le contenu d'un mot de 16 bits sous forme de chiffres binaires, est-il possible de dire si le contenu du mot est un nombre ou deux caractères codés ASCII?

EXERCICE 1.4.4

Le dictionnaire français contient environ 70000 mots. Si on suppose que la moyenne de la longueur des mots est de 8 caractères, est-il concevable de pouvoir représenter le dictionnaire français dans moins de 70000×8 octets?

EXERCICE 1.4.5

La codification des caractères sur 8 bits (e.g. ASCII) n'est pas suffisant pour les nombreux langages naturels existants à travers le monde. Ainsi, plusieurs autres codifications ont été créées pour remédier à ce problème. L'une de ces codifications est l'Unicode. Celle-ci utilise, pour un caractère, 16 bits plutôt que 8 bits. Pour avoir plus d'information, consultez le site Web suivant : `www.unicode.org`. L'Unicode semble s'imposer car certains langages de programmation, e.g. Java, l'utilisent. Quel est le code du caractère α (grec alpha) en Unicode?

Chapitre 2

Assemblage et exécution d'un programme

La codification des instructions se fait durant l'assemblage du programme. C'est le travail de l'assembleur. Cette codification suit un ensemble de règles relativement complexes.

Clairement il ne s'agit pas d'apprendre toutes ces règles. Ce serait un travail fort laborieux. Il s'agit plutôt de connaître le processus global de la codification d'une instruction. Étant donné toutes les informations pertinentes, vous devriez être en mesure de faire cette codification.

2.1 Codification des instructions

Le format de l'instruction `move` est:

```
bits         15  14  13  12  11  10  9   8  7  6  5  4  3  2  1   0
            ------------------------------------------------------
contenu     | 0 | 0 | size | registre  | mode   |  mode  | registre |
            ------------------------------------------------------
                                Destination          Source
      size = 01 pour .b
             11 pour .w
             10 pour .l
      mode = 000 pour dn
           = 010 pour (an)
           = 011 pour (an)+
           = 100 pour -(an)
           = 101 pour (depl,an)
```

Dans le cas où le mode d'adressage est `(depl,an)`, le déplacement est codé à la suite de l'instruction sur 16 bits.

EXERCICE 2.1.1

Donnez la codification, en hexadécimal, des instructions suivantes: (on suppose que le registre `a5` pointe la section `data`)

```
        move.w  d2,y+2
        move.w  d3,z
        move.w  x,x
        data
y       ds.l    1
z       ds.w    1
x       ds.w    1
```

EXERCICE 2.1.2

Donnez la codification binaire des instructions suivantes en supposant la section `data` pointée par le registre `a5`.

```
            move.w   d0,d1
            move.b   d2,d3
            move.w   0(a0),4(a2)
            move.l   d2,4(a2)
            move.w   x,d0
            move.w   x+2,d0
            move.w   (2*4,a0),d0
            ...
            data
M:          ds.w     20
x:          ds.w     1
```

2.2 Assemblage

EXERCICE 2.2.1

Quelle est la différence essentielle entre une directive et une instruction assembleur?

EXERCICE 2.2.2

Expliquez pourquoi un assemblage à deux passes dans le code source simplifie la tâche de l'assembleur dans la gestion des étiquettes.

EXERCICE 2.2.3

Montrez que pour les instructions de branchement, deux passes pour l'assembleur ne sont généralement pas suffisantes pour générer des branchements utilisant le format le plus compacte.

EXERCICE 2.2.4

Expliquez en quelques lignes ce que contient le code objet d'un programme assemblé.

2.3 Assemblage et chargement

EXERCICE 2.3.1

Un programme qui est chargé en mémoire pour être exécuté peut se retrouver à une adresse quelconque. Expliquez comment les instructions peuvent référencer les étiquettes de la partie `data` si les adresses réelles de ces étiquettes sont inconnues à l'assemblage du programme.

2.4 Édition des liens

EXERCICE 2.4.1

Quel est le but principal de l'édition des liens?

EXERCICE 2.4.2

Qu'est-ce que l'édition des liens statique? ; dynamique?

Chapitre 3

Expression arithmétique entière

3.1 Addition binaire et bits NZVC

EXERCICE 3.1.1

L'opération d'addition sur un certain nombre de bits peut générer un débordement arithmétique. Par exemple, la valeur 127 plus 1 sur huit bits en complément à deux génère un débordement puisque le résultat 128 ne peut être représenté sur huit bits en complément à deux.

Supposons deux nombres sous formes de chaînes de bits codés en complément à deux. Comment peut-on savoir, après une opération d'addition, s'il y a débordement arithmétique? Pour deux nombres codés en binaire pur?

EXERCICE 3.1.2

Trouvez des paires de nombres entiers de 16 bits, tels que leur addition génère : a) C = 1 et V = 0, b) C = 0 et, V = 0, c) C = 1 et V = 1, d) C = 0 et V = 1. Ainsi, il n'existe aucune relation entre les bits états C et V.

EXERCICE 3.1.3

Montrez que pour l'addition, les cas a) N=0 et C=0 et V=1, b) N=1 et C=1 et V=1 ne sont pas possibles. Montrez que les six autres cas sont possibles.

EXERCICE 3.1.4

Dans la suite d'instructions suivante, il y a deux instructions de comparaisons

inutiles : pourquoi le sont-elles?

```
...
sub.l   d0,d1
cmp.l   #0,d1   ; est-ce utile ?
beq     Ok
move.l  d2,d3
cmp.l   #0,d3   ; est-ce utile ?
bmi     Bien
...
```

3.2 Évaluation d'expression arithmétique entière

EXERCICE 3.2.1

Écrire une séquence d'instructions pour évaluer l'expression $x * (y + z) + 100$ en supposant que les valeurs x, y et z sont stockés dans les registres `d0.w`, `d1.w` et `d2.w` respectivement. Le résultat doit se retrouver dans le registre `d3.w`. Écrire quatre versions différentes : sans détection de débordement, avec détection de débordement et pour les cas complément à deux et binaire pur.

EXERCICE 3.2.2

Il y a deux instructions de multiplication, l'une pour les valeurs signées et une autre pour les valeurs non-signées. Montrez sur un exemple précis que cela est nécessaire pour obtenir des résulats corrects. (Remarquez qu'il n'y a pas deux instructions d'addition.)

EXERCICE 3.2.3

Écrire une séquence d'instructions pour multiplier deux nombres de 32 bits codés en complément à deux, donnant un résultat de 32 bits et fonctionant sur le M68000. Produire deux versions : avec détection de débordement et sans détection de débordement. Vous ne devez pas employer les instructions de multiplication de 32 bits disponibles sur les processeurs 68020, 68030 et 68040. Supposez que les registres `d0` et `d1` contiennent les opérandes ; le résultat se retrouve dans le registre `d2`.

EXERCICE 3.2.4

Écrire un programme en assembleur MC68k qui lit trois nombres entiers x, y, z et qui affiche les valeurs des expressions $x(y+z)$ et $xy+xz$. Les entrées et les sorties se font sur 16 bits en complément à deux. Vérifiez votre programme en utilisant, entre autres, les valeurs $x = 32767$, $y = 3$ et $z = -1$.

EXERCICE 3.2.5

Écrire un programme qui affiche la partie entière de la valeur de l'expression $x + x * 15\%$ pour une valeur entière x donnée en entrée par l'utilisateur du programme.

EXERCICE 3.2.6

Écrire des instructions assembleurs pour implanter les assignations suivantes a) $x = x + y$ b) $y = x + y$ c) $x = y + x * y$ d) $x = x + y + z$ e) $z = x + y + z$ étant données les définitions suivantes et en supposant que les valeurs déjà contenues en mémoire sont codées en complément à deux. (Sans se préoccuper de détection de débordement arithmétique.)

```
      data
x:    ds.l  1
y:    ds.w  1
z:    ds.b  1
```

3.3 Opérations arithmétiques particulières

Les instructions de décalages `asr`, `asl`, `lsl` et `lsr` permettent d'appliquer des opérations arithmétiques de divisions et de multiplications particulières ; et ceci de façon très rapide. La rapidité de leur exécution, en comparaison aux instructions habituelles `div` et `mul`, est l'intérêt majeur de leur utilisation. En effet, l'instruction `divu` est approximativement 60 fois plus lente qu'une opération `asr` ; l'instruction `mulu` est 30 fois plus lente que `asl`. Ceci n'est qu'une approximation et varie selon les processeurs ; en général, les opérations de division et de multiplication sont beaucoup plus lentes que les opérations d'addition, de soustraction et de décalage.

Ces quelques exercices présentent les techniques principales pour faire des opérations de multiplication et division rapides.

EXERCICE 3.3.1

Donnez une instruction pour calculer le quotient d'une division par huit du contenu du registre `d0.l` en supposant son contenu codé en binaire pur. De même pour une division par 32.

EXERCICE 3.3.2

Donnez une instruction pour mutiplier par 8 le contenu du registre `d0`. De même pour une multiplication par 32.

EXERCICE 3.3.3

Est-ce que les instructions `lsl` et `asl` produisent le même résultat?

EXERCICE 3.3.4

Est-ce que l'instruction `asr` produit le quotient d'une division par une puissance de 2 sur des nombres négatifs?

EXERCICE 3.3.5

Comment multiplier par la valeur 10, le contenu du registre `d0.l`, sans faire 10 additions mais en utilisant l'instruction `asl.l` et une addition? (On peut supposer que le contenu de `d0.l` est codé en binaire pur, mais cela est sans importance.)

EXERCICE 3.3.6

Pour calculer le quotient de la division de $x \geq 0$, de 16 bits, par une **constante** $c > 0$, il est possible de le faire en employant la multiplication et un décalage à droite (ce qui est une division par une puissance de deux, disons 2^p). En d'autres mots, étant donné une constante c, il y a deux constantes k et p telles que le quotient de $(xk)/2^p$ est identique au quotient de x/c. L'objectif est d'éviter les instructions de division dans le cas d'un diviseur constant. Montrez comment trouver les constantes k et p, étant donné c, pour calculer le quotient de x/c, à l'aide d'une multiplication et d'un décalage. (Suggestion : pour débuter essayez une constante précise, par exemple, $c = 5$. Remarque : ce genre de transformation est effectuée par les compilateurs pour améliorer le temps d'exécution, car la division est plus lente que la multiplication.)

Chapitre 4

Instructions conditionnelles

4.1 Instructions de base

EXERCICE 4.1.1

Quelle est la différence majeure entre une instruction de branchement conditionnelle et une instruction de branchement inconditionnelle?

EXERCICE 4.1.2

L'instruction `bne` utilise quel(s) bit(s) du registre d'état dans sa décision d'appliquer un branchement?

EXERCICE 4.1.3

Est-ce que la séquence d'instructions suivantes est utile?

```
      cmp.w  d0,d1
      beq    egal
egal: ....
```

EXERCICE 4.1.4

Est-ce que, dans la séquence qui suit, l'instruction `beq` est utile?

```
      add.l  d0,d1
      beq    plusloin
```

EXERCICE 4.1.5

Est-ce qu'une instruction de branchement conditionnelle doit nécessairement suivre une instruction `cmp` pour avoir une signification bien précise?

EXERCICE 4.1.6

Modifiez la séquence d'instructions suivante pour éliminer une instruction de branchement.

```
y:      ....
        cmp.w   d2,d4
        bne     x
        bra     y
x:
```

EXERCICE 4.1.7

Quelle est la différence entre `bhi` et `bgt`?

EXERCICE 4.1.8

Dans la séquence suivante, est-ce qu'il y a branchement quand `d0` est plus grand que `d1` ou quand `d1` est plus grand que `d0`?

```
        cmp.l   d0,d1
        bgt     ailleurs
        ...
ailleurs:
```

4.2 Instruction SI

EXERCICE 4.2.1

Implantez les instructions suivantes en assembleur MC68k. Les identificateurs `x`, `y` et `z` sont des étiquettes définies dans la section `data` à l'aide de la directive `ds.w`. On suppose que des nombres entiers codés en complément à deux sont déjà présents en mémoire à ces étiquettes.

```
a) Si (x > 0) Alors x = 1
              Sinon x = 2

b) Si (x < 10) et (y > 100) Alors z = x + y
                            Sinon z = x - y
```

```
c) Si (x < 9) ou (y < -45) Alors z = 0
                           Sinon z = x

d) Si (x < -1) Alors Si (y > 77) Alors y = 1
                                 Sinon z = 0
               Sinon y = 0
```

4.3 Instruction Tantque

EXERCICE 4.3.1

Écrire un programme qui lit deux nombres entiers et affiche le plus grand commun diviseur, i.e. le pgcd, de ces deux nombres.

Voici l'algorithme classique d'Euclide pour le calcul du pgcd de deux nombres positifs. (Il aurait été préférable d'écrire cet algorithme sous la forme d'une fonction.)

```
Programme pour afficher le pgcd de deux nombres entiers positifs
a = LireEntier()
Si a < 0 Alors Arret
b = LireEntier()
Si b < 0 Alors Arret
Tantque a<>0 et b<>0 Faire
   Début
   Si a > b Alors a = reste de a / b
            Sinon b = reste de b / a
   Fin
Si a <> 0 Alors afficher a
Sinon Si b <> 0 Alors afficher b
Sinon afficher 1
Fin
```

EXERCICE 4.3.2

Écrire un programme qui lit un nombre entier positif et affiche la racine carrée entière de ce nombre. Si le nombre lu est négatif, le programme s'arrête avec un message d'erreur approprié. Vous pouvez utiliser l'algorithme simple suivant. (Ce n'est pas un algorithme très rapide!)

Programme pour lire un entier n et afficher la racine carré entière de n.

```
n = LireEntier()
```

```
Si n < 0 Alors Afficher("Ne peut calculer la racine carree d'un negatif");
               Arreter
x = 0
y = 0
Tantque x + 2y + 1 <= n Faire
  Debut x = x + 2y +1
        y = y + 1
  Fin
Afficher y
Fin
```

EXERCICE 4.3.3

Écrire un programme qui lit quatre nombres entiers a, b, c et d, et qui affiche la somme des deux fractions a/b et c/d sous la forme d'une fraction simplifiée. Vous pouriez réutiliser la séquence d'instructions, déjà construite, pour calculer le pgcd.

4.4 Programmes complets

EXERCICE 4.4.1

Écrire un programme dont l'entrée est un nombre entier n, $0 \leq n \leq 32767$, suivi de n nombres entiers, de 16 bits, x_i. Votre programme lit séquentiellement ces nombres et affiche le nombre de *sommets* rencontrés. Un *sommet* est un nombre x_i tel que $x_{i-1} < x_i \geq x_{i+1}$. (Remarque : la première valeur ne peut être un sommet. Si $n \leq 2$, il n'y a pas de sommet, mais il faut quand même lire les nombres en entrée.)

Chapitre 5

Sous-routine

5.1 Sous-routine avec passage des paramètres par les registres

EXERCICE 5.1.1

Construire une sous-routine retournant la plus petite valeur de deux nombres entiers, codés en complément à deux, de 32 bits.

EXERCICE 5.1.2

Construire une sous-routine qui retourne la valeur absolue d'un nombre entier de 32 bits.

EXERCICE 5.1.3

Construire une sous-routine qui retourne le pgcd de deux nombres entiers codé en binaire pur.

EXERCICE 5.1.4

Traduire la fonction suivante en assembleur MC68k. Indiquez les registres pour passer les paramètres et retourner le résultat.

```
Fonction Inconnue(x : entier) : entier;
Var w, i : entier;
Debut
  w = 0;
  Pour i de 1 a x Faire w = w + i*i + 2*i;
```

```
    Retourner(w)
  Fin Inconnue;
```

EXERCICE 5.1.5

Construire une sous-routine qui affiche n fois le message "Pas de panique". La valeur n est passée en paramètre.

EXERCICE 5.1.6

Quelle est l'utilité pratique de sauvegarder tous les registres modifiés au début de la sous-routine?

EXERCICE 5.1.7

À la lumière de la méthode de passage des paramètres par registres, qu'est-ce qu'un paramètre formel et un paramètre actuel?

EXERCICE 5.1.8

Est-ce qu'une sous-routine peut appeller une autre sous-routine?

EXERCICE 5.1.9

Est-ce qu'une sous-routine interne peut accéder les étiquettes définies dans la partie `data` d'un même programme source?

EXERCICE 5.1.10

Est-ce qu'une sous-routine peut contenir plusieurs instructions `rts`?

5.2 Retourner un résultat par les bits du CCR

Les bits X, N, Z, Vet Cpeuvent être utilisés pour retourner, d'une sous-routine, un résultat booléen. L'essentiel est dans l'usage des instructions `andi.b #n,ccr` et `ori.b #n,ccr` où la constante `n` doit être choisie adéquatement pour mettre à zéro ou à 1 le bit voulu. Remarque : les bits X, N, Z, Vet Csont dans cet ordre du bit 4 au bit 0 dans le CCR. Ainsi pour mettre le bit V à 1, il suffit de faire `ori.b #2,ccr` ; le bit Z à 0, il suffit de faire `and.i #256-4,ccr` ; le bit N à 0, il suffit de faire `and.i #256-8,ccr`, etc.

EXERCICE 5.2.1

Écrire une sous-routine qui a deux paramètres x et y. Cette sous-routine retourne 1, par le bit Z, si et seulement si x est un diviseur de y. (Précision: si x est zéro, Z sera indéfini ; si y est zéro, et x n'est pas zéro, Z sera à 0.)

EXERCICE 5.2.2

Écrire une **sous-routine** pour générer des nombres aléatoires selon l'algorithme écrit suivant ; écrire une version utilisant la division dont le diviseur a 32 bits (non disponible sur le M68000) et une version utilisant la division dont le diviseur a 16 bits (disponible sur le M68000, mais plus difficile à réaliser car il y a dans l'algorithme suivant un diviseur ayant plus de 16 bits, il faut donc le briser en morceaux pour le faire avec l'opération de division dont le diviseur a 16 bits.) . Écrire une partie principale d'un programme pour afficher les 100 premiers nombres aléatoires. Dans le code suivant, la variable `seed` a 32 bits, le résultat retourné a donc aussi 32 bits. L'opération de division `/` donne le quotient de la division. L'opération $x\%y$ calcule le reste de la division de x par y.

```
int seed = 1;
int random()
{
  int hi = seed / 127773;
  int lo = seed % 127773;
  int x = lo*16807 - hi*2836;
  if (x<0) seed = x + 2147483648;
   else seed = x;
  return seed;
}
```

5.3 Passage d'une sous-routine à une sous-routine

Dans certains cas, il est utile de passer une sous-routine comme argument à une sous-routine. Par exemple, supposons une sous-routine `lireEntier` qui doit lire un entier satisfaisant une contrainte décrite par une fonction quelconque f ; celle-ci retourne par le bit C une indication d'acceptation ou de refus de l'entier. Cette sous-routine doit signaler une erreur si l'entier ne satisfait pas f. Elle doit refaire cette lecture jusqu'à ce que l'utilisateur entre un entier satisfaisant f. Ainsi, `lireEntier` a un paramètre qui est l'adresse d'une sous-routine; i.e. la fonction f.

EXERCICE 5.3.1

Écrire une sous-routine qui a deux paramètres : l'adresse d'une sous-routine f et l'adresse d'une chaîne S de caractères. Cette sous-routine lit des nombres entiers jusqu'à ce qu'un entier lu satisfasse la fonction f. La fonction f a un paramètre, le registre `d0.w` et sa sortie est le bit C (le bit C est à 1 si et seulement si le nombre dans `d0.w` est satisfaisant) . La sous-routine retourne l'entier satisfaisant par le registre `d0.w`. Si un nombre lu ne satisfait pas f, la chaîne S est affichée.

Chapitre 6

Les vecteurs

6.1 Chaîne de caractères

EXERCICE 6.1.1

Écrire une sous-routine qui accepte en entrée une chaîne de caractères se terminant par le caractère `<nul>` et qui retourne le bit Z à 1 si et seulement s'il y a au moins un caractère de la chaîne ayant le bit 7 à 1. Votre sous-routine a au plus 15 instructions MC68k.

EXERCICE 6.1.2

Construire une fonction pour trouver la première occurence d'une sous-chaîne de caractères dans une chaîne. La fonction retourne l'adresse du premier caractère se trouvant dans la chaîne ou 0 (nil) si la sous-chaîne n'est pas présente. Une chaîne de caractères se termine par un octet `<nul>`.

6.2 Vecteur de nombres entiers

EXERCICE 6.2.1

Traduire le pseudo-code suivant en assembleur MC68k. Le vecteur V est formé d'entiers de 32 bits, `x` et `y` sont des étiquettes.

```
x   ds.l  1
y   ds.l  1
V   ds.l  1000
```

```
Pour i de 0 a 999 Faire
  Début
  Si V[i] < x Alors V[i] = 1
  Si V[i+1] > y Alors V[i+1] = 2
  Fin
```

EXERCICE 6.2.2

Écrire une procédure qui copie un vecteur d'une longueur quelconque dans un espace mémoire donnée. Les deux espaces peuvent se chevaucher et il faut traiter adéquatement ce problème.

EXERCICE 6.2.3

Écrire une sous-routine pour trouver la valeur maximum dans un vecteur de 200 entiers de 16 bits.

EXERCICE 6.2.4

Supposons que le registre `a0` pointe un vecteur de 200 entiers de 32 bits. Écrire une séquence d'au plus 15 instructions pour placer dans le registre `d0.l` la plus grande différence `V[i]-V[i+1]`.

EXERCICE 6.2.5

Traduire le code suivant en assembleur. Le vecteur `V` est formé d'entiers de 16 bits et l'étiquette `x` réfère à un entier de 16 bits. Les identificateurs `i` et `j` ne sont que des symboles pour exprimer le code.

```
x = 0;
Pour i de 0 a 99 Faire
  Debut
  Pour j de 0 a 99 Faire
    Début
    Si V[i] > V[j] Alors x = x + V[j] FinSi
    Fin
  Fin
```

EXERCICE 6.2.6

Donnez une séquence d'au plus 15 instructions pour placer les puissances x^i, $0 \leq i \leq 14$, dans le vecteur `xp`.

```
x:     ds.w  1
xp:    ds.w  15
```

EXERCICE 6.2.7

Soit le vecteur `V` de 500 entiers de 32 bits. Écrire un bout code pour trier ce vecteur en ordre croissant selon le pseudo-code suivant.

```
Pour i de 0 a 498 Faire
  Debut
  min = v[i];
  Pour j de i+1 a 499 Faire
    Si min > v[j] Alors min = V[j];
  v[i] = min;
  Fin
```

Chapitre 7

Matrices

7.1 Opérations de base

EXERCICE 7.1.1

Soit la déclaration `M ds.l 200` dans le segment `data`. Est-ce que cet espace mémoire est suffisant pour contenir une matrice de 20 lignes et 20 colonnes d'entiers de 32 bits? Une matrice de 10 lignes et 20 colonnes d'entiers de 32 bits? Une matrice de 20 lignes et 10 colonnes d'entiers de 32 bits?

EXERCICE 7.1.2

Supposons que le registre `a0` pointe une matrice M de 5 lignes et 10 colonnes d'entiers de 16 bits. Donnez une instruction pour placer en $M[4,3]$ la valeur -1.

7.2 Traitement sur des matrices

EXERCICE 7.2.1

Soit une matrice `M` de 50 lignes et 20 colonnes de nombres entiers de 32 bits définie dans la partie `data`. (Le premier élément est M[0,0] et la matrice est stockée ligne par ligne en mémoire.). Donnez une séquence d'au plus 15 instructions assembleur MC68k pour sommer tous les nombres des colonnes 12, 16, 18 et 19. Le résultat se retrouve dans le registre `d0.l`. (Il n'est pas nécessaire de se préoccuper de débordement.)

EXERCICE 7.2.2

Écrire une sous-routine qui accepte en entrée une matrice de 25 lignes et 35 colonnes d'entiers de 16 bits et un pointeur vers une zone de (25*35*4) octets. La sous-routine retourne une copie de la matrice, dans la zone, mais dont les entiers sont étendus sur 32 bits. Votre solution a au plus 20 instructions.

EXERCICE 7.2.3

Un vecteur `V` est composé de 100 adresses de 32 bits, chacune pointant vers un vecteur de 50 entiers de 32 bits. Écrire une séquence d'au plus 20 instructions MC68k pour copier ces 5000 entiers sous la forme d'une matrice de 100 lignes et 50 colonnes à l'étiquette `M` telle que la première ligne de la matrice contient les entiers du premier vecteur, la deuxième ligne les entiers du deuxième vecteur, etc. (Les étiquettes `V` et `M` sont déja définies dans la partie `data`.)

EXERCICE 7.2.4

Écrire une fonction pour trouver dans une matrice d'entiers M de 5 lignes et 7 colonnes une entrée $[i, j]$ telle que $M[i, j]$ est la plus petite valeur sur la colonne j et la plus grande valeur sur la ligne i. Par exemple, dans la matrice suivante la position $[4, 5]$, ayant la valeur 62, a cette propriété. (Les numéros de lignes et de colonnes débutent par 1.) Dans le cas ou aucune entrée de la matrice ne satisfait la propriété, retourner l'adresse NIL (0) sinon retourner l'adresse de l'élément $M[i, j]$ trouvé. (Remarques : un maximum doit être unique sur une même ligne. De même, un minimum doit être unique sur une même colonne. Il ne peut y avoir qu'un seul élément ayant la propriété demandé.)

```
10  100  80   -3  98   0   0
 0   43  12   12  82   0   1
-2    0  11   66  72  65  43
-1   12  16   17 >62< 55  12
 0    0  99   65  71  33  10
```

EXERCICE 7.2.5

La transposé M^t d'une matrice M est définie par l'équation $M[i, j] = M^t[j, i]$. Écrire une procédure qui applique la transposition sur une matrice de dimension quelconque. L'espace mémoire occupée par la matrice doit être réutilisé pour recevoir le résultat et il ne faut pas employer un second espace mémoire pour transposer. La forme de la matrice change, mais l'espace occupé demeure le même.

Chapitre 8

Passage des paramètres sur la pile et espace local

EXERCICE 8.0.6

Décrire le contenu de la pile après l'exécution de l'instruction `movem` de la sous-routine f pour le code suivant. Indiquez le contenu de la pile en spécifiant les cas où on ne connaît pas ce contenu.

```
            move.w  #4,-(sp)
            move.l  #-1,-(sp)
            move.l  x,-(sp)
            jsr
            ...

f:          link    a6,#-10
            movem.w d0-d2,-(sp)

            ...

            unlk    a6
            rts

            data
x           dc.l    100
            end
```

EXERCICE 8.0.7

a) Traduisez en assembleur le pseudo-code suivant en passant tous les paramètres sur la pile et en allouant les variables locales sur la pile. Utilisez des EQU pour définir clairement les positions des paramètres et des variables locales et rendre plus claire les accès à ces variables. La fonction retourne son résultat par le registre zéro. N'optimisez pas le code, et implantez les accès v[i] en utilisant un registre d'index dn. Le vecteur v est passé par adresse et non par valeur. b) Supposons que les paramètres x, y et z sont passés par adresse et non par valeur; modifiez votre code assembleur conséquemment. Il faut modifier les déplacements pour accéder aux paramètres sur la pile car les adresses ont 32 bits.

a)

```
Fonction F(x, y, z : entier de 16 bits;
           v : vecteur[0..99] entiers 32 bits ) : entier 32 bits
Var  s : entier de 32 bits
     i : entier de 16 bits
Debut
  s = 0;
  Pour i de 0 a 99 Faire
    Si x < y Alors v[i] = v[i] - z;
             Sinon s = s + v[i]
  retourner s;
Fin
```

EXERCICE 8.0.8

Soient les fonctions

```
Fonction f(x : entier 32 bits) : entier 32 bits
Debut retourner(x+x+x) Fin

Fonction g(x, y, z : entier 32 bits) : entier 32 bits
Debut retourner(x+y+z) Fin

Fonction h(x : entier 16 bits) : entier 32 bits
Debut retourner(x+x) Fin
```

Implantez ces fonctions en passant les paramètres sur la pile et en retournant leur résultat sur la pile: le résultat doit se retrouver sur la pile après le rts, il faut donc allouer de l'espace avant d'empiler le premier argument avant l'appel. Les fonctions doivent enlever les paramètres à l'aide de l'instruction rtd.

Soit le pseudo-code suivant.

```
f( g(h(3), h(4), h(5)) + g(h(6), h(7), h(8)) )
```

où f, g et h réfèrent aux fonctions précédentes.

Implantez cette expression en assembleur. Observez la facilité d'appel dû aux résultats placés sur la pile.

Chapitre 9

Les conversions pour les entrées et les affichages d'entiers.

9.1 Sous-routines classiques de conversion

EXERCICE 9.1.1

Écrire une sous-routine pour convertir un nombre entier de 16 bits, codé en complément à deux, en une chaîne de caractères le représentant en décimal.

EXERCICE 9.1.2

Écrire une sous-routine pour convertir une nombre décimal, représenté sous la forme d'une chaîne de caractères, en un nombre binaire de 16 bits codé en complément à deux.

9.2 Un petit problème de comptage de bits

EXERCICE 9.2.1

Écrire une fonction qui détermine le nombre de bits nécessaire et suffisant pour convertir en complément à deux un nombre entier x exprimé en décimal sous forme de caractères. La détermination peut être approximative avec une erreur d'au plus 1 bit.

Par exemple, pour représenter le nombre 1234 en complément à deux, 12 bits sont suffisants et nécessaires. Pour le nombre 123456789, au moins 28

bits doivent être utilisés. Pour -12345678912345, c'est 45 bits.

Chapitre 10

Manipulation de bits

10.1 Opérations de base

EXERCICE 10.1.1

Donnez le contenu du registre `d0.w` et des bits `C` et `V`, en hexadécimal après chacune des instructions suivantes. Le registre `d0.w` contient `1000 1111 0101 1010` avant l'exécution de chaque instruction.

```
a) lsr.w #1,d0
b) lsl.w #1,d0
c) asr.w #1,d0
d) asl.w #1,d0
e) rol.w #1,d0
f) ror.w #1,d0
g) lsr.w #4,d0
h) asr.w #4,d0
i) lsl.w #4,d0
j) asl.w #4,d0
```

EXERCICE 10.1.2

Le registre `d0.l` contient deux nombres entiers, codés en complément à deux, dont le premier x s'étend du bit 0 au bit 19 et le second y du bit 20 au bit 31. Donnez une séquence d'au plus 10 instructions MC68k pour additionner y à x. Le résultat $x + y$ tient dans les bits 0 à 19 de `d0.l` avec y dans les bits 20 à 31. (On ne se préoccupe pas de débordement. Portez attention aux nombres négatifs.)

10.2 Chaîne de bits

EXERCICE 10.2.1

Le registre `d0.l` contient une chaîne de 32 bits. Écrire une séquence d'au plus 15 instructions MC68k pour placer à partir du bit 0 de `d1.l` tous les bits à 1 de `d0.l`. En d'autres mots, `d1.l` devra contenir en base unaire le nombre de bits à 1 dans `d0.l`. Exemple : si `d0.l` contient

0000 0111 0011 0101 0111 0001 0001 0111

`d1.l` devra contenir

0000 0000 0000 0000 0111 1111 1111 1111

EXERCICE 10.2.2

Le registre `d0.l` contient une chaîne de 32 bits qui dirige une voiture. En lisant la chaîne du bit 0 à 31, un bit à 1 en position paire indique d'avancer d'un centimètre à droite et un bit à 1 en position impaire indique d'avancer d'un centimètre à gauche. Si le bit est à 0, la voiture avance d'un centimètre vers l'avant. Écrire une séquence d'au plus 15 instructions pour calculer le nombre de centimètres, **vers la gauche**, où la voiture se trouve par rapport au point de départ, après son déplacement dirigé. Le résultat se retrouve dans le registre `d1.w`. (Remarque : le résultat est négatif si la voiture se retrouve à droite par rapport au point de départ.)

10.3 Graphique, matrice de bits

EXERCICE 10.3.1

Écrire une sous-routine qui accepte une matrice M de 100 lignes par 800 colonnes d'entiers de 16 bits et qui retourne une matrice M' de 100 lignes par 800 colonnes de bits, telle que $M'[i,j] = 1$ si et seulement si $M[i,j] \neq 0$. La matrice retournée M' est compactée en mémoire : huit éléments occupent 8 bits, i.e. un octet. L'élément $M'[0,0]$ occupe le bit 0 du premier octet, etc. Votre sous-routine a au plus 20 instructions MC68k.

EXERCICE 10.3.2

Écrire une procédure pour afficher un curseur, à la position (x, y), sur un écran graphique de 1400 pixels de large par 1000 pixels de haut. L'écran est une matrice de bits dont le coin supérieur gauche est $(0, 0)$. La forme de la matrice est présentée à la figure 10.1.

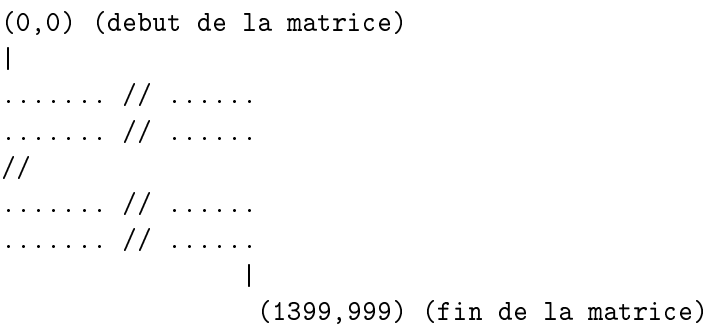
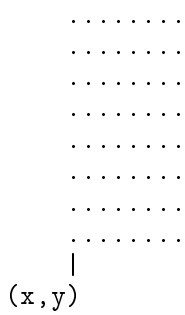


FIG. 10.1 – **Forme et coordonnées d'une matrice de pixels.**

Le curseur a 8 pixels de haut par 8 pixels de large. L'affichage du curseur doit être tel que le contenu actuel de l'écran n'est pas perdu et visible en négatif : un pixel qui était allumé est maintenant éteint, un pixel éteint est maintenant allumé. Cela permet de déplacer le curseur rapidement, car pour afficher le curseur à un autre point il suffit de réappliquer la procédure sur l'emplacement actuel avant de l'afficher au nouvel emplacement. Le point (x, y) spécifie le point inférieur gauche du curseur.



Un octet de la matrice contient 8 pixels. Le bit 0 d'un octet est le pixel de gauche, tandis que le pixel 7 est le pixel de droite. Cela apparaît donc inversé. Il y a 1400 pixels de large, donc 175 octets sur une ligne de la matrice pour représenter une ligne de 1400 pixels de l'écran.

Par exemple, le pixel (9,0) est dans le deuxième octet au bit 1, le pixel (34,0) est dans le cinquième octet au bit 2, le pixel (34,1) dans le 180ième octet au bit 2, etc.

EXERCICE 10.3.3

Un écran graphique est formé de pixels à 1024 couleurs. Chaque pixel utilise donc 10 bits. L'écran a 512 colonnes et 400 lignes et les pixels sont stockés

de façon compacte, c'est-à-dire avec aucune perte de bits. Le premier pixel occupe les bits 0 à 7 du premier octet et les bits 0 à 1 du deuxième octet, le deuxième pixel les bits 2 à 7 du deuxième octet et les bits 0 à 3 du troisième octet, etc. Supposons que le registre `d0.w` contient le numéro de ligne et le registre `d1.w` le numéro de colonne du pixel à modifier. Écrire une séquence d'instructions MC68k pour placer ce pixel à la couleur '102'. L'écran débute à l'adresse stockée dans `a0`. Votre solution a au plus 20 instructions.

10.4 Ensembles représentés par des chaînes de bits

EXERCICE 10.4.1

Écrire une procédure pour afficher un ensemble, représenté par une chaîne de 256 bits, à l'écran. Les entiers séparés par des virgules apparaissent entre accolades.

EXERCICE 10.4.2

Écrire une sous-routine pour compter le nombre d'éléments dans un ensemble.

EXERCICE 10.4.3

Écrire une sous-routine pour déterminer si un élément est dans un ensemble.

Chapitre 11

Représentation de nombres fractionnaires en virgule-fixe

La représentation des nombres fractionnaires la plus populaire dans les ordinateurs commerciaux est le standard IEEE-754. Cette représentation utilise le concept de virgule-flottante et est assez différente de la représentation des entiers en complément à deux.

Par contre, la représentation en *virgule-fixe* élimine la complexité de maintenir la position de la virgule qui délimite la partie entière de la partie fractionnaire en fixant sa position. Ceci simplifie les opérations de base en permettant un usage des opérations entières. Cette représentation est donc utile si on ne dispose pas d'instructions capable de traiter des nombres en virgule-flottante. De plus, dans plusieurs situations les performances en rapidité d'exécution sont meilleurs avec ce type de repésentation.

Finalement, leur étude nous permet de mieux voir les problèmes reliés au manque de précision des opérations fractionnaires utilisant une représentation binaire positionnelle et l'utilité de plusieurs opérations de base comme les décalages et les opérations logiques $\vee$ et $\wedge$.

11.1 Éléments

La codification en virgule-fixe [16,16] utilise les bits 16 à 31 pour la partie entière et les bits 0 à 15 pour la partie fractionnaire pour représenter un nombre potentiellement non entier. La virgule[1] séparant la partie entière

1. Nous ferons usage du point '.' plutôt que de la virgule ',' pour séparer la partie entière de la partie fractionnaire car son utilisation est très répandue en informatique. C'est ainsi

de la partie fractionnaire se trouve donc entre les bits 15 et 16. La partie entière est codée en complément à deux bien que la partie fractionnaire est en base deux. Ainsi le bit 15 vaut 1/2, le bit 14 vaut 1/4, le bit 13 vaut 1/8 etc. La valeur d'un nombre est la somme de la partie entière et de la partie fractionnaire.

EXERCICE 11.1.1

Donnez la représentation binaire des nombres suivants dans la codification en virgule-fixe [16,16]. Remarquez que la codification d'un nombre négatif peut être perçue comme l'addition d'une partie entière négative et d'une partie fractionnaire positive. Par exemple -12.75 est -13 plus 0.25.

a) 1 b) -1 c) 0.5 d) 0.25 e) 0.75 f) -0.75 g) -5 h) -4.5 i) 0.125 j) 0.875 k) -9 l) -8.125

EXERCICE 11.1.2

Montrez que la codification de -4.5 en virgule-fixe [16,16] est identique à la codification de $-4.5 \times 65536 = -294912$ sur 32 bits. De façon générale, pour coder un nombre x en virgule-fixe [16,16] il suffit de coder en complément à deux sur 32 bits la valeur arrondie de $x \times 65536$.

EXERCICE 11.1.3

Supposons que le registre `d0.l` contient un nombre codé en virgule-fixe [16,16] donnez une instruction `add`, de la famille MC68k, qui permet d'additionner la valeur 0.5 à ce nombre.

EXERCICE 11.1.4

Calculez la plus grande valeur représentable en virgule-fixe [16,16]. De même pour la plus petite valeur positive différente de zéro.

EXERCICE 11.1.5

Soient les deux méthodes suivantes pour multiplier deux nombres codés en virgule-fixe [16,16].

a) multiplier les deux nombres de 32 bits avec résultat sur 64 bits et décaler ce résultat de 16 bits vers la droite.

b) décaler les deux nombres de 8 bits vers la droite, multiplier avec résultat sur 32 bits.

Montrez avec des valeurs précises que la méthode `a` produit moins de perte de précision que la méthode `b`.

---

que dans la litérature informatique l'expression 'codification en virgule-fixe' est souvent traduite par 'codification en point-fixe'.

EXERCICE 11.1.6

Soient le registre `d0.w` contenant un nombre entier a et le registre `d1.w` contenant un entier $b > 0$ (les deux nombres sont codés en complément à deux). Écrire une séquence d'au plus 10 instructions MC68k pour placer dans le registre `d2.l` la représentation en virgule fixe [16,16] de a/b. (Il peut y avoir une perte de précision.)

EXERCICE 11.1.7

Écrire une sous-routine en MC68k qui retourne la valeur arrondie d'un nombre y codé en virgule-fixe [16,16]. La valeur retournée, i.e. l'arrondi, doit être aussi codée en virgule-fixe [16,16]. L'arrondi de y est la valeur $\lfloor y+1/2 \rfloor$ pour $y \geq 0$ et $-\lfloor |y| + 1/2 \rfloor$ si $y < 0$. (Par exemple, 2.5 donne 3, 12.2 donne 12, 4.9 donne 5, -2.4 donne -2, -4.9 donne -5). Votre sous-routine doit avoir un maximum de 10 instructions.

EXERCICE 11.1.8

Soit une codification à virgule-fixe avec 20 bits pour la partie entière et 12 bits pour la partie fractionnaire ([20,12]).

a) Donnez la plus grande valeur et la plus petite valeur représentable dans cette codification.

b) Indiquez comment faire une multiplication de deux nombres [20,12] en assembleur MC68k qui donne un résultat en virgule-fixe [20,12] avec au plus 5 instructions.

En se référant à votre code, est-ce qu'il y a un débordement possible? Pourquoi?

En se référant à votre code, est-ce qu'il y a une perte de précision possible? Pourquoi?

11.2 Approximation de fonctions

EXERCICE 11.2.1

Implanter une fonction pour calculer le sinus d'un nombre codé en virgule-fixe [16,16].

La fonction sinus peut être approximée par une série de MacLaurin

$$\sin x = x - x^3/3! + x^5/5! - x^7/7! + \cdots$$

(x est en radian)

Bien que cela semble prendre un temps important, c'est une méthode plus rapide que plusieurs implantations commerciales en virgule-flottante (IEEE-754). Des tests sur les processeurs Sparc et Alpha démontrent que c'est plus rapide par au moins un ordre de grandeur. (Trente cinq fois plus rapide sur le Sparc et quarante fois plus rapide sur l'Alpha.) Bien entendu, il y a moins de précision en virgule fixe [16,16] qu'en IEEE 64 bits.

11.3 L'utilisation de fractions comme valeurs non-entières

EXERCICE 11.3.1

Modifier le pseudocode suivant pour éviter les opérations de division et multiplication. Traduire en assembleur votre pseudocode pour le tester sur la machine.

```
Procedure AfficherMultiple9sur5(n : cardinal)
      { n >= 0 }
Var i : cardinal;
Debut
Pour i de 1 a n Faire Afficher(partie entiere de i*9/5);
Fin AfficherMultiple9sur5
```

Chapitre 12

La codification en virgule-flottante IEEE-754

La codification IEEE-754 est commercialement la plus répandue pour représenter les nombres rationnels. C'est la représentation employée pour les variables de type réel dans les langages de haut niveau. (`float` dans le langage C, `REAL` en Pascal, etc.). Une étude de cette codification nous fait découvrir un premier fait majeur : non seulement la majorité des calculs en IEEE-754 sont entachés d'approximation mais la transformation de décimal à IEEE-754 est généralement faite de façon approximative. Il faut donc éviter d'employer ce type de représentation pour des calculs ne tolérant aucune approximation.

12.1 Éléments

EXERCICE 12.1.1

Convertir les nombres suivants en IEEE-754 format 32 bits.

a) -832.0
b) 0.0390625
c) $-2.099689E-18$
d) -0.000751
e) $4.8765E-16$

EXERCICE 12.1.2

Convertir les nombres suivants, codés en IEEE-754 format 32 bits, en décimal.

```
    S Exposant  Mantisse
a)  1 1000 0000 1000 0000 0000 0000 0000 000
b)  0 0111 1111 0000 0000 0000 0000 0000 000
c)  1 1000 0000 1100 0000 0000 0000 0000 000
d)  0 0111 1110 1010 0000 0000 0000 0000 000
e)  1 0000 0000 0000 0000 0000 0000 0000 000
f)  0 0000 0000 0000 0000 0000 0000 0000 000
g)  0 0000 0001 1000 0000 0000 0000 0000 000
h)  0 0111 1011 0000 0000 0000 0000 0000 000
```

EXERCICE 12.1.3

Donnez un exemple d'un nombre non-entier qui ne peut être stocké sans perte de précision sous la forme IEEE, quelque soit la précision.

EXERCICE 12.1.4

Quelle est la valeur, en décimal, du plus grand nombre fini représentable en IEEE-754 sur 32 bits? La plus petite valeur finie, positive, différente de zéro?

EXERCICE 12.1.5

Quelle est la plus petite valeur **entière** x représentable en IEEE-754 de 32 bits telle que $x + 1$ n'est pas représentable exactement? (Cette valeur x nous donne la plage $[-x, x]$ des valeurs entières représentables sans perte de précision en IEEE-754 sur 32 bits.)

EXERCICE 12.1.6

Est-ce qu'on peut espérer, en utilisant la représentation IEEE-754 avec un grand nombres de bits, obtenir des calculs sans perte de précision dans tous les cas?

EXERCICE 12.1.7

Pourquoi les processeurs numériques commerciaux utilisent le point-flottant plutôt que le point-fixe?

EXERCICE 12.1.8

La représentation de IEEE-754 de 32 bits est mieux connue sous le vocable de point-flottant simple précision. Le format double précision est formé de 64 bits : il y a un bit de signe, 11 bits pour l'exposant et 52 bits pour la

mantisse. Dans le cas de la mantisse normalisée, comme pour le format de 32 bits, le bit extrême gauche à 1 n'est pas stocké. Ce qui donne une mantisse réelle de 53 bits. L'exposant est biaisé 1023. Pour ce format de 64 bits : a) Donnez la plus grande valeur représentable; b) la précision en décimale.

12.2 Traitement en IEEE

EXERCICE 12.2.1

Écrire une **sous-routine** pour multiplier par 4 un nombre réel IEEE-754 32 bits, sans utiliser aucune instruction à virgule-flottante. L'entrée et la sortie de votre sous-routine sont le registre `d0.l`. Votre sous-routine doit avoir au plus 15 instructions. (On ne se préoccupe pas de débordement, des nombres non normalisés, des NaN et des infinis ; portez attention au zéro.)

EXERCICE 12.2.2

Écrire une fonction pour convertir un nombre entier de 32 bits, codé en complément à deux, en IEEE 32 bits. S'il y a perte de précision retourner le bit C à 1, sinon retourner le bit C à 0.

EXERCICE 12.2.3

Décrire un algorithme pour additionner deux nombres IEEE de 32 bits en utilisant seulement des opérations entières disponibles sur les MC68k.

Chapitre 13

Récursivité au niveau machine

Le coût de l'utilisation de la récursivité est minime. Ce chapitre veut démontrer ce fait. De plus, la récursivité est un outil fort utile pour exprimer plusieurs algorithmes que ce soit en assembleur ou dans un langage de haut niveau. On voit aussi par l'étude, au niveau machine de sous-routines récursives, que la méthode de sauvegarder l'adresse de retour sur la pile offre une solution simple à l'implantation de la récursivité. Ainsi, la pile joue un rôle essentiel pour l'implantation de la récursivité.

13.1 Évaluations récursives

EXERCICE 13.1.1

Implantez la fonction factoriel en assembleur selon le pseudocode suivant.

```
Fonction factoriel(n : Cardinal): Cardinal;
Debut
Si n <= 1 Alors retourner(1)
      Sinon retourner(n * factoriel(n-1));
Fin factoriel;
```

EXERCICE 13.1.2

Le calcul du pgcd, ie le plus grand diviseur commun, de deux nombres positifs peut se faire par la fonction récursive suivante. Traduire le pseudocode suivant en assembleur.

```
Fonction Pgcd(a, b : Cardinal): Cardinal;
Debut
Si a = 0 et b = 0 Alors retourner(1)
   Sinon Si a = 0 Alors retourner(b)
   Sinon Si b = 0 Alors retourner(a)
   Sinon Si a > b Alors retourner(Pgcd(a mod b, b))
   Sinon retourner(Pgcd(a, b mod a));
Fin Pgcd;
```

EXERCICE 13.1.3

Le registre `a0` pointe une zone mémoire contenant trois champs. Le premier champ est un octet, le deuxième un nombre entier de 32 bits codé en complément à deux et le troisième une adresse de 32 bits pointant vers une autre zone de trois champs ayant le même format (cette description est donc récursive). L'octet indique l'opération à effectuer: 1 indique de faire une addition, 2 une opération de multiplication, 3 une opération de soustraction et 0 de ne pas faire d'opération mais de simplement retourner l'entier du deuxième champ. Cette opération est effectuée entre l'entier (premier argument) et la valeur provenant du traitement semblable appliqué sur la zone pointée par le troisième champ. Écrire une sous-routine d'au plus 20 instructions qui fait l'évaluation demandée et place le résultat dans le registre `d0.l`.

EXERCICE 13.1.4

Écrire une sous-routine récursive en assembleur MC68k pour évaluer une expression booléenne formée des constantes 1 et 0, des opérateurs + (ou-logique), $*$ (et-logique) et des parenthèses '()'. L'opérateur $*$ a précédence sur l'opérateur +. Il n'est pas nécessaire d'indiquer les erreurs de syntaxe. Le résultat doit être retourné par le bit Z.

Annexe A

Corrigé partiel

1.1.1

	Décimal	Hexadécimal
a)	435	= 01B3
b)	234	= 00EA
c)	10	= 000A
d)	8	= 0008
e)	100	= 0064
f)	255	= 00FF
g)	1023	= 03FF
h)	566	= 0236
i)	543	= 021F
j)	9	= 0009
k)	99	= 0063
l)	200	= 00C8

1.1.2

a) débordement sur 8 bits, 1234 = 0000 0100 1101 0010

b) débordement sur 8 bits, 2034 = 0000 0111 1111 0010

c) -56 = 1100 1000, -56 = 1111 1111 1100 1000

d) débordement sur 8 bits, 345 = 0000 0001 0101 1001

e) débordement sur 8 bits et 16 bits

f) débordement sur 8 bits, -432 =1111 1110 0101 0000

g) débordement sur 8 bits, 1009 = 0000 0011 1111 0001

h) 64 = 0100 0000, 64 = 0000 0000 0100 0000

i) -5 = 1111 1011, -5 = 1111 1111 1111 1011

j) débordement sur 8 bits, 1256 = 0000 0100 1110 1000

k) débordement sur 8 bits, -1024 = 1111 1100 0000 0000

l) débordement sur 8 bits, 2376 = 0000 1001 0100 1000

1.1.3 Voici en binaire pur les nombres demandés :

a) 1111 1111 b) 0001 0000 c) 0010 0000 0000 d) 1010 0101 1110

e) 0010 1111 1101

1.1.4

	Binaire	Décimal	Hexadécimal
a)	0000 1100 1001 0101	3221	0C95
b)	1001 1101 1110 1010	−25110	−6216 ou 9DEA
c)	0001 1001 1001 0100	6548	1994
d)	0111 0000 1011 1100	28860	70BC

1.1.5 a) Pour mieux voir ce qui se produit il faut distinguer deux cas, les nombres positifs et les nombres négatifs. Dans le cas positif le bit i mis à zéro diminue la valeur du nombre de 2^i si ce bit était à 1. Dans le cas négatif il faut distinguer le cas du bit extrême gauche des autres bits. Pour le bit extrême gauche, la valeur augmente de 2^{n-1}. Pour les autres bits, i.e. les bits i de 0 à $n-2$, la valeur diminue de 2^i si le bit était à 1. En résumé, la valeur diminue de 2^i si le bit i était à 1, sauf dans le cas du bit extrême gauche qui était à 1 qui fait augmenter de 2^{n-1}.

b) La valeur augmente de 2^i si le bit i était à 0, sauf dans le cas du bit extrême gauche qui fait diminuer de 2^{n-1}.

1.1.6 Une nanoseconde est 10^{-9} seconde. Pour 64 bits cela prendra $(2^{64}-1)\times 10^{-9}$ secondes, c'est-à-dire approximativement $1.8446744073709553e10$ secondes ; ce qui correspond à plus de 584 ans. Pour 32 bits cela prendra $(2^{32}-1)\times 10^{-9}$ secondes ; c'est-à-dire approximativement 4.294967295 secondes. Conclusion : un compteur du nombre de cycles d'une horloge d'un processeur devrait avoir plus de 32 bits, et 64 bits apparaît suffisant.

1.2.1 Pour 0 à 1 000, il faut au moins 10 bits, soit $\lceil \log_2 1001 \rceil$ bits ou $\lceil (\log_{10} 1001)/(\log_{10} 2) \rceil$ bits. Pour 0 à 100 000 il faut au moins 17 bits, soit $\lceil \log_2 100001 \rceil$ bits. Pour 0 à 1 000 000 il faut au moins 20 bits, soit $\log_2 1000001$ bits. Pour 0 à 1 000 000 000 il faut au moins 30 bits, soit $\lceil \log_2 1000000001 \rceil$ bits.

Pour représenter les nombres entiers de 0 à 1 000 il faut 1 001 chaînes de bits différentes, c'est pourquoi '1 001' apparaît dans l'expression '$\lceil \log_2 1\ 001 \rceil$'. De façon générale, pour représenter les nombres entiers de 0 à n, en binaire pur, il faut $\lceil \log_2 n \rceil$ bits.

1.2.2 On peut avoir 2^{32} chaînes, c'est-à-dire 4 294 967 296.

1.2.3 Le bit extrême droit est 0 pour pair et 1 pour impair. Cela est aussi vrai en complément à deux.

1.2.4 Numérotons les cartes de 0 à 6. La carte i contient tous les nombres dont la représentation binaire a le bit i à 1. Par exemple, la carte 0 a tous les nombres impairs, la carte 1 a les nombres 2, 3, 6, 7, 10, 11, 14, 15, etc. Les nombres sont placés en ordre croissant sur les cartes, de la gauche vers la droite et de haut en bas. Ainsi, les coins supérieurs gauches ont les nombres 1, 2, 4, 8, 16, 32 et 64. Pour le magicien, il suffit de faire la somme de ces valeurs, pour les cartes qui lui sont remises, pour obtenir l'âge de la personne. C'est une forme de conversion binaire à décimale. Il faut avoir l'oeil rapide et l'addition facile!

1.3.1 Un mot mémoire contient toujours une chaîne de bits, car chacun des bits de la mémoire centrale doit être dans un état stable. De plus, on ne peut supposer que cette valeur est zéro au début de l'exécution d'un programme.

1.3.2 Un espace de 400 octets, car 32 bits utilisent quatre octets.

1.3.3 L'adresse du dernier mot est $1000 + 99 \times 2 = 1198$. (Remarquez que l'adresse du dernier **octet** est 1199.)

1.3.4 Les termes « big-endian » et « little-endian » s'appliquent à la méthode de stockage des mots et des longs mots en mémoire centrale pour un processeur donné. (Notez que cela ne s'applique pas à la méthode de stockage dans les registres, qui est toujours simple: les bits sont numérotés de façon continue soit de la gauche vers la droite (e.g. IBM) ou de la droite vers la gauche (e.g. Motorola, AMD).) Soit un mot de 16 bits à stocker en mémoire centrale aux adresses 1000 et 1001. Dans le cas big-endian, la partie la plus haute (i.e. MSB, bits 8 à 15) du mot est stocké à l'adresse 1000 et la partie la plus basse (i.e. LSB, bits 0 à 7) à l'adresse 1001. Dans le cas « little-endian », le LSB est stocké à l'adresse 1000 et le MSB à l'adresse 1001. Dans le cas d'un long mot (32 bits), c'est une situation similaire : pour big-endian c'est l'octet MSB à l'adresse inférieure et le LSB à l'adresse supérieure. Les termes big-endian et little-endian sont empruntés aux aventures de Gulliver : les big-endians sont les lilliputiens qui mangent leurs oeufs par le gros bout et les little-endians par le petit bout. Les processeurs Intel (x86 et Pentium) sont des little-endians, les processeurs Motorola (MC68k et PowerPc) sont des big-endians. Le Hitachi SH3/SH4 (un processeur RISC) peut fonctionner dans les deux méthodes de stockage. Il y a des avantages et désavantages à chacune de ces méthodes. (Les processeurs Intel 486 et Pentium ont une instruction (i.e. `BSWAP` pour « byte swap ») pour interchanger l'ordre des octets d'un long mot de big-endian à little-endian et vice versa.)

1.3.5 La valeur est $1 \times 256 + 2 = 258$. Pour une machine « little-endian » la valeur lue sera plutôt $2 \times 256 + 1 = 513$, car le premier octet contenant la valeur 1 est la partie-basse et le 2 est la partie-haute. Cela peut causer certains

problèmes de portabilité de programmes écrits en C. (Disons-le, mal écrit en C, car le programmeur devrait se préoccuper de portabilité!)

1.3.6 Les valeurs, en décimal, contenues dans les octets à partir de l'étiquette `x` sont : 0, 1, 97, 65, 97, 65, 0, 1, 1, 0, 0, 97, 0, 65, 0, 0, 0, 1, 0, 0, 3, 108, 0, 0, 0, 97, 0, 0, 0, 65, 70, 105, 110, 105, 10, 13, 255, 255, 255, 254. Voici une image plus claire. (Remarque : 3*256+108 = 876.)

```
x    dc.b  0,1,'a','A','aA'    ; 0,1,97,65,97,65
     dc.w  1,256,'a','A'       ; 0,1,1,0,0,97,0,65
     dc.l  1,876,'a','A'       ; 0,0,0,1,0,0,3,108,0,0,0,97,0,0,0,65
     dc.b  'Fini',10,13        ; 70,105,110,105,10,13
     dc.w  -1,-2               ; 255,255,255,254.
```

Maintenant pour une machine « little-endian » cela donne : 0, 1, 97, 65, 97, 65, 1, 0, 0, 1, 97, 0, 65, 0, 1, 0, 0, 0, 108, 3, 0, 0, 97, 0, 0, 0, 65, 0, 0, 0, 70, 105, 110, 105, 10, 13, 255, 255, 254, 255. Dans ce cas nous avons interchangé les octets partie-basse et partie-haute pour les nombres 1, 256, -1 et -2 et les caractères 'a' et 'A' sur 16 bits(.w) et inversé la séquence de quatre octets pour les valeurs de 32 bits(.l).

1.3.7

```
x       dc.l    -500,200          255,255,254,12,0,0,0,200
        dc.b    '!33!'            33,51,51,33
        dc.l    1222,-1,512       0,0,4,198,255,255,255,255,0,0,2,0
```

1.4.1 La chaîne est "1987 @ 1996".

1.4.2 En ASCII, il y a un bit de différence. Le code ASCII de la lettre 'a' est $61_{16} = 0110\ 0001_2$ et le code ASCII de la lettre 'A' est $41_{16} = 0100\ 000_{12}$. Ils diffèrent donc par le bit 5. En ASCII, les lettres minuscules sont numériquement plus élevées que les lettres majuscules.

1.4.3 Dans plusieurs des cas il ne sera pas possible de distinguer deux caractères d'un nombre, mais dans d'autres cas il sera possible de trancher en faveur d'un nombre plutôt que deux caractères, puisque certains codes ASCII ne correspondent pas à aucun caractère.

1.4.4 Oui cela est concevable pour plusieurs raisons : il est possible de codifier les syllabes plutôt que les caractères, ou encore, plus simple, codifier chacun des caractères sur 6 bits plutôt que 8 bits. (Le dictionnaire français n'utilise pas plus de 64 caractères différents.)

1.4.5 C'est $03B1_{16}$.

2.1.1

```
Instructions     Codification hexadécimal

  move.w d2,y+2  3B42 0002
  move.w d3,z    3B43 0004
  move.w x,x     3B6D 0006 0006
  data
y ds.l 1
z ds.w 1
x ds.w 1
```

2.1.2

```
Instruction             Codification en hexadécimal (mot par mot)
move.w  d0,d1           3200
move.b  d2,d3           1602
move.w  0(a0),4(a2)     3568 0000 0004
move.l  d2,4(a2)        2542 0004
move.w  x,d0            302D 0028 (correspond à move.w (40,a5),d0)
move.w  x+2,d0          302D 002A (correspond à move.w (42,a5),d0)
move.w  (2*4,a0),d0     3028 0008
```

2.2.1 Une directive est une commande pour l'assembleur, mais une instruction est une commande pour l'UCT. La directive est interprétée lors de l'assemblage du programme tandis que l'instruction est interprétée lors de l'exécution du programme. La directive ne sera pas interprétée lors de l'exécution du programme ; elle n'a servi qu'à aider à l'assemblage du programme.

2.2.2 En assembleur les étiquettes référencées peuvent apparaître avant leur déclaration. L'assembleur à deux passes peut dédier la première passe à l'accumulation des déclarations des étiquettes pour permettre à la deuxième passe de générer entièrement les instructions.

2.2.3 Cela se produit car le processus du choix des instructions de branchement influence la longueur des instructions. Ainsi, si dans un premier temps on découvre qu'une instruction de branchement peut être racourcie, ce gain en espace peut alors influencer d'autres instructions de branchement. Le choix optimal entre branchements court et long ne peut se faire qu'après plusieurs passes. Ce qui n'est pas toujours fait. Ainsi, pour l'assembleur du M68000, une instruction de branchement peut avoir une indication de distance ; e.g.

`bge.s etq` indique que l'étiquette `etq` n'est «pas loin» («s» pour «short») i.e. que le déplacement est codifiable sur 8 bits.

2.2.4 Le code objet contient principalement les instructions du code source codées en binaire. De plus, il contient une table des références externes et des identificateurs publiques, i.e. les définitions d'exportations d'identificateurs. Les références externes vont être utiles pour leur résolution durant l'édition des liens. En plus, il y a des informations sur la taille de la partie contenant les instructions et les données statiques (`data`).

2.3.1 L'assembleur suppose que le registre `a5` pointe le début de la partie `data` et pour toute instruction qui référence une étiquette déclarée dans le segment `data`, génère une instruction qui utilise ce registre comme registre d'adresse de base. Pour chaque étiquette il y a un déplacement par rapport au début de la partie `data` qui est calculé par l'assembleur et utilisé dans l'instruction comme déplacement. On appelle ce déplacement l'adresse relative. Le contenu du registre `a5` doit être initialisé correctement lors du chargement du programme en mémoire ; c'est une tâche du chargeur.

2.4.1 Un programme source compilé en un code object fait souvent référence à des procédures ou fonctions qui sont prédéfinies et existantes dans des librairies systèmes ou construites par l'utilisateur. La phase de l'édition des liens collige ces différents codes externes pour produire un code exécutable.

2.4.2 L'édition des liens statique collige tous les codes objets externes en un seul code exécutable. L'édition des liens dynamiques prépare la partie principale du programme pour charger, selon les besoins, d'autres codes objets durant l'exécution (ce sont les librairies dynamiques, Une librairie dynamique sous Windows a l'extension `.dll`.). L'édition dynamique a l'avantage d'éviter de produire des codes exécutables utilisant beaucoup de mémoire. Toutefois, le désavantage majeur est que le code exécutable dépend de librairies que l'utilisateur pourrait ne pas avoir lors de l'exécution.

3.1.1 Pour deux nombres en complément à deux, il faut regarder le signe des nombres de départ et le signe du résultat. Par exemple, si les deux nombres sont positifs, bit extrême gauche à 0, et que le résultat est négatif, bit extrême gauche à 1, alors il y a débordement arithmétique. Il ne faut pas croire qu'un bit report 1 sortant indique un débordement. (Cela s'applique pour les nombres codés en binaire pur.) Le même raisonnement s'applique pour deux nombres négatifs. Si les deux nombres de départ sont de signes opposés, il ne peut se produire un débordement.

Pour deux nombres en binaire pur, c'est le bit report sortant qui détermine si il y a débordement. Un bit report sortant de 1 indique un débordement, un bit report de zéro indique un non débordement.

3.1.2 a) $-1+1$ b) $0+0$ c) $-32768+-1$ d) $32767+1$.

3.1.3 cas a) Pour produire la situation N=0 et V=1 il faut nécessairement additionner deux nombres négatifs qui donne un résultat positif (N=0) ce qui bien entendu est un débordement (V=1). Mais dans ce cas le bit sortant C doit être nécessairement à 1 car les bits extrêmes gauches des deux nombres sont à 1 (nombres négatifs). Ainsi le cas N=0 et C=0 et V=1 est impossible.

cas b) Raisonnement semblable pour le cas N=1 et C=1 et V=1, car il faudrait additionner deux nombres positifs qui donnerait un résultat négatif (N=1 et V=1), mais dans ce cas il ne pourrait se produire un bit sortant à 1 (C=1) car les deux nombres positifs ont leur bit extrême gauche à 0. Voici des valeurs x et y, sur 16 bits, qui additionnées donnent les bits d'états à gauche.

```
N C V       x          y
0 0 0       0          0
0 0 1   impossible
0 1 0      -1          1
0 1 1  -32768         -1
1 0 0       0         -1
1 0 1   32767          1
1 1 0      -1         -1
1 1 1   impossible
```

3.1.4 L'instruction `cmp.l #0,d1` est inutile car l'instruction suivante `beq` branche dans le cas où `d1` contient zéro, mais la soustraction `sub` positionne le Zexactement de la même façon. De même pour la deuxième comparaison : le `move.l d2,d3` positionne le bit `N`, il indique si `d3` est négatif ou non. La comparaison suivante `cmp.l #0,d3` devient inutile.

3.2.1 Une version sans détection de débordement en complément à deux.

```
move.w  d1,d3        ; y
add.w   d2,d3        ; y+z
muls.w  d0,d3        ; x*(y+z), cela modifie tout le registre d3
add.w   #100,d3      ; x*(y+z)+100, on considère par la suite d3.w
```

Une version avec détection de débordement en complément à deux.

```
move.w  d1,d3        ; y
add.w   d2,d3        ; y+z
bvs     debordement  ; le bit V indique si un débordement en C2
```

```
       muls.w  d0,d3        ; x*(y+z) (cela modifie tout le registre d3)
       move.w  d3,d7        ; début détection de débordement
       ext.l   d7           ; si la partie basse de d3 est identique
       cmp.l   d7,d3        ; à la valeur de 32 bits de d3
       bne     debordement  ; alors il n'y a pas de débordement
       add.w   #100,d3      ; x*(y+z)+100
       bvs     debordement
       ...                  ; ici le résultat est dans d3.w
debordement:
       jsr     stop         ; simple arrêt du programme
```

Une version sans détection de débordement en binaire pur.

```
       move.w  d1,d3        ; y
       add.w   d2,d3        ; y+z
       mulu.w  d0,d3        ; x*(y+z) ; cela modifie tout le registre d3
       add.w   #100,d3      ; x*(y+z)+100 ; on considère par la suite d3.w
```

Une version avec détection de débordement en binaire pur.

```
       move.w  d1,d3        ; y
       add.w   d2,d3        ; y+z
       bcs     debordement  ; bit C indique si un débordement en binaire pur
       mulu.w  d0,d3        ; x*(y+z) (cela modifie tout le registre d3)
       cmpi.l  #65535,d3    ; Il y a débordement, sur 16 bits,
       bhi     debordement  ; si la valeur est supérieure à 65535
       add.w   #100,d3      ; x*(y+z)+100
       bcs     debordement
       ...                  ; ici le résultat est dans d3.w
debordement:
       jsr     stop         ; simple arrêt du programme
```

3.2.2 Il faut effectivement deux instructions de multiplication pour diriger le processeur vers le bon algorithme selon la codification utilisée. L'instruction d'addition n'a pas deux versions, mais un exemple simple permet de voir que l'algorithme de multiplication doit être différent selon la codification supposée. À titre d'exemple, la valeur -1 sur 16 bits, qui est 65535 en binaire pur, multipliée par -1, doit donner 1, mais devrait donner 4294836225 sur 32 bits pour le cas binaire pur. On a donc ici deux résultats dont la codification ne concorde pas.

Mais fait remarquable, $4294836225 = FFFE0001_{16}$ — et là nous sortons du cadre de la question — ce qui veut dire que le résultat de la partie basse est la même en binaire pur qu'en complément à deux dans le cas de -1×-1.

Est-ce le cas pour tout nombre de 16 bits? C'est le cas. Ainsi, si nous gardons seulement la partie faible de 16 bits du résultat, il n'y a pas de différence entre `MULS` et `MULU`.

3.2.3 Voici une version avec détection de débordement. Soient les nombres x et y de 32 bits. Supposons que ces deux nombres sont positifs. Notons par $x.h$ et $y.h$ les parties hautes de x et y ; de même par $x.l$ et $y.l$ les parties basses de x et y. Ainsi, $x = x.h \times 2^{16} + x.l$; de même $y = y.h \times 2^{16} + y.l$. Donc on a l'identité suivante $x \times y = x.l \times y.l + (x.l \times y.h + y.l \times x.h)2^{16} + (x.h \times y.h)2^{32}$. Cette identité nous permet d'implanter le code pour multiplier deux nombres de 32 bits par morceaux de 16 bits. L'exposant 2^{16} ne pose pas de problème car il s'agit de manipuler les parties hautes des registres. L'exposant 2^{32} est encore moins problématique car il « disparaît » : le facteur $(x.h \times y.h)$ se retrouve dans un registre de 32 bits, il ne sert qu'à déterminer si un débordement s'est produit. Le signe du résultat peut être calculé séparément ; ainsi nous prenons les valeurs absolues de x et y avant de faire tous les calculs et appliquons une négation au résultat final si les signes de x et y sont différents.

```
            move.w  #1,d6         ; d6 contiendra le signe du résultat
            cmpi.l  #0,d0         ; x positif?
            bgt     posd0         ; oui, c'est bien
            neg.l   d0            ; non, x devient positif
            neg.w   d6            ; le signe du résultat vient de changer
posd0:      cmpi.l  #0,d1         ; y positif?
            bgt     posd1         ; oui, c'est bien
            neg.l   d1            ; non y devient positif
            neg.w   d6            ; le signe du résultat vient de changer
posd1:      move.w  d0,d2         ; ici x et y sont positifs
            mulu.w  d1,d2         ; y.l*x.l sur 32 bits dans d2
            swap    d0            ; rend plus facilement accessible x.h
            move.w  d0,d3         ; une copie de x.h
            mulu.w  d1,d3         ; y.l*x.h sur 32 bits dans d3
            swap    d1            ; rendre accessible y.h
            move.w  d0,d4         ; une copie de y.h
            mulu.w  d0,d4         ; x.h*y.h
            bne     debordement   ; Si x.h*y.h <>0 Alors debordement
            swap    d0            ; x.l
            mulu.w  d1,d0         ; y.h*x.l
            add.l   d0,d3         ; x.l*y.h + y.l*x.h
            cmpi.l  #65535,d3     ; dépasse les 16 bits?
            bhi     debordement   ; oui, débordement
            swap    d2            ; partie haute de y.l*x.l
            add.w   d3,d2         ; additionne partie haute ensemble
            bcs     debordement   ;
            swap    d2            ; replace la partie haute à sa place
            tst.w   d6            ; est-ce que le signe final est négatif?
            bgt     positif
            neg.l   d2            ; introduire le signe dans le résultat
```

```
          bra     fin
positif:
          tst.l   d2            ; cas particulier de débordement
          bmi     debordement   ; où le résultat final est 2^32
fin:                            ; ici d2 contient le résultat final
          ....                  ; reste du programme
debordement:
          jsr     stop          ; simple arrêt du programme
```

Voici une version sans détection de débordement. Cette solution plus courte consiste à utiliser une propriété de la codification du complément à deux : le résultat de la multiplication signée est identique à la multiplication non-signée pour la partie basse. Ainsi, les signes non pas à être considérés si on garde seulement la partie basse du résultat, c'est-à-dire les 32 bits inférieurs. Ce qui est notre cas, car du résultat de 64 bits, seulement la partie inférieure de 32 bits est conservée. C'est beaucoup plus simple.

```
            move.w  d0,d2
            mulu.w  d1,d2         ; y.l*x.l sur 32 bits dans d2
            swap    d0            ; rend accessible x.h
            move.w  d0,d3         ; copie de x.h
            mulu.w  d1,d3         ; y.l*x.h sur 32 bits dans d3
            swap    d1            ; y.h
            swap    d0            ; rend accessible x.l
            move    d0,d4         ; copie de x.l
            mulu.w  d1,d4         ; y.h*x.l
            add.l   d3,d4         ; x.l*y.h + y.l*x.h
            swap    d2            ; addition de partie haute de y.l*x.l
            add.w   d4,d2         ; avec partie basse de x.l*y.h + y.l*x.h
            swap    d2            ; le résultat est dans d2
            ...
positif:    ...                   ; ici le reste du programme
debordement:
            jsr     stop          ; simple arrêt du programme
```

3.2.5

```
            xref    decin,decout,strout,stop
            move.w  #msgfin-msg,d0    ; longueur du message
            lea     msg,a0            ; adresse du message
            jsr     strout            ; afficher un message de demande d'entier
            jsr     decin             ; lire un nombre entier de 16 bits
            move.w  d0,d1             ; sauvegarder la valeur de x
            muls.w  #15,d1            ; 15*x (peut déborder!)
            divs.w  #100,d1           ; quotient dans d1.w = partie entière 15%*x
            add.w   d0,d1             ; partie entière de x + x*15/100 = x + x*15%
```

```
            move.w  #msgRfin-msgR,d0  ; longueur du message
            lea     msgR,a0           ; adresse du message
            jsr     strout            ; afficher un message annoncant le résultat
            move.w  d1,d0             ; pour afficher la valeur de x + x*15%
            jsr     decout            ; afficher cette valeur
            jsr     stop              ; arrêt du programme
            data
msg:        dc.b    'Entrez un nombre entier (appelons-le x) '
msgfin:
msgR:       dc.b    'La partie entière de x+x*15% est '
msgRfin:
            end
```

3.2.6

```
a)      move.w  y,d0          ; y sur 16 bits
        ext.l   d0            ; de 16 bits à 32 bits
        add.l   d0,x          ; addition de deux nombres de 32 bits
b)      move.l  x,d0          ; prend les 32 bits de x
        add.w   d0,y          ; mais ne traite que la partie basse de x
b')     add.w   x+2,y         ; ça fonctionne aussi !
c)      move.w  y,d0          ; prendre y (16 bits)
        ext.l   d0            ; de 16 bits à 32 bits
        move.l  d0,d1         ; sauvegarde y (32 bits)
        muls.l  x,d0          ; x*y
        add.l   d1,d0         ; x+x*y sur 32 bits
        move.l  d0,x          ; assignation à x
d)      move.b  z,d0
        extb.l  d0
        move.w  y,d1
        ext.l   d1
        add.l   d1,d0         ; x+y sur 32 bits
        add.l   d0,x          ; x = x+y+z
e)      add.b   x+3,z         ; 8 bits les plus bas de x additionner à z
        add.b   y+1,z         ; 8 bits les plus bas de y additionner à z
```

3.3.1 L'instruction `lsr.l #3,d0` produit le quotient d'une division par 8 du contenu du registre d0. L'instruction `lsr.l #5,d0` produit le quotient d'une division par 32 du contenu du registre `d0`. En général, pour produire le quotient d'une division par 2^i, il suffit d'appliquer un décalage vers la droite de i bits. Remarquez que cette instruction produit toujours un résultat entier. Ainsi, il s'agit ici du calcul de $\lfloor x/2^i \rfloor$.

3.3.2 L'instruction `lsl.l #3,d0` multiplie par 8 le contenu du registre `d0`. L'instruction `lsl.l #5,d0` multiplie par 32 le contenu du registre `d0`. En général,

pour multiplier par 2^i, il suffit d'appliquer un décalage vers la gauche de i bits.

3.3.3 Les instructions `lsl` et `asl` produisent le même résultat final, mais l'instruction `lsl` positionne toujours le bit V à zéro, tandis que `asl` indique s'il y a débordement arithmétique. Dans le cas de nombres codés en complément à deux, une multiplication par une puissance de 2, i.e. 2^i, peut se faire par l'instruction `asl`. En fait, cela peut aussi se faire par `lsl`, mais `asl` indiquera, par le bit V, si un débordement arithmétique s'est produit.

3.3.4 Il faut porter attention à ce que signifie «le quotient d'une division». Si on définit ce quotient comme étant $\lfloor x/2^i \rfloor$, c'est bien ce que fait l'instruction `asr`. Toutefois, remarquez que cela n'est pas la même définition pour `divs`. Par exemple, si le dividende est -3 et le diviseur 2, le résultat d'un décalage d'un bit à l'aide de `asr`, donnera -2. (Ce résultat s'obtient simplement en prenant la représentation de -3 et en le décalant d'un bit : $FFFFFFFC = -3$ décalé d'un bit à droite, en gardant le signe, on obtient $FFFFFFFE = -2$.) C'est bien la signification de $\lfloor x/2^i \rfloor$: le plus grand entier plus petit que $-3/2 = -1.5$, ce qui est -2. Mais l'instruction `divs` donnera plutôt le quotient -1 (avec un reste -1).

3.3.5 Il suffit de faire

```
asl.l   #1,d0        ; deux fois le contenu de d0
move.l  d0,d1
asl.l   #2,d1        ; huit fois le contenu de d0
add.l   d1,d0        ; dix fois le contenu de d0
```

Et voici une solution avec détection de débordement arithmétique dans le cas du complément à deux

```
asl.l   #1,d0        ; deux fois le contenu de d0
bvs     debordement
move.l  d0,d1
asl.l   #2,d1        ; huit fois le contenu de d0
bvs     debordement
add.l   d1,d0        ; dix fois le contenu de d0
bvs     debordement
```

Mais la solution suivante ne donne pas une détection correct de débordement en binaire pur, car pour le deuxième décalage, le bit C n'indique pas correctement s'il y a eu un débordement en binaire pur (le bit C peut être à 0, mais un bit à 1 est sorti en premier).

```
lsl.l   #1,d0        ; deux fois le contenu de d0
bcs     debordement
```

```
move.l  d0,d1
lsl.l   #2,d1        ; huit fois le contenu de d0
bcs     debordement
add.l   d1,d0        ; dix fois le contenu de d0
bcs     debordement
```

3.3.6 Toute la méthode provient de la possibilité d'approximer $1/c$ par une fraction de la forme $k/2^p$.

Soit x un nombre positif de 16 bits. Pour calculer le quotient de la division de x par c, une constante positive, il suffit de multiplier par $k = \lceil 2^{16}/c \rceil$ et de calculer le quotient de la division par 2^{16}.

Par exemple, pour $c = 5$, on multiplie x par 13108 et on décale de 16 bits à droite (ceci est une division entière par 2^{16}). Par exemple, pour les dividendes 5 et 65535 : $5 \times 13108/2^{16} = 1$ et $65535 \times 13108/2^{16} = 13107$.

Voici deux autres exemples : pour $c = 10$ on a $k = 6554$, et pour $c = 25$ on a $k = 2621$. Toutes les opérations proposées peuvent se faire à l'aide de l'opération `MULU` et `ASR`.

La solution donnée n'est pas unique ; car pour un c donné, il est possible qu'une puissance de deux inférieure à 2^{16} soit suffisante.

4.1.1 L'instruction de branchement inconditionnelle, le `bra` pour les MC68k, effectue toujours un branchement, bien qu'une instruction de branchement conditionnelle utilise un ou plusieurs bits du registre d'état pour décider si un branchement devrait être appliqué.

4.1.2 Le bit Z. L'instruction `bne` effectue un branchement si et seulement si Z=0.

4.1.3 Le branchement n'est d'aucune utilité puisque quelque soit le résultat de la comparaison, la prochaine instruction à exécuter sera à l'étiquette `egal`.

4.1.4 Oui, car l'addition modifie le bit Z, donc l'instruction `beq plusloin` va brancher à l'étiquette `plusloin` si et seulement si l'addition donne le résultat 0.

4.1.5 Une instruction de branchement conditionnelle peut suivre une instruction quelconque et avoir une signification précise. Par exemple, l'instruction `bvs` peut être placée après une instruction d'addition pour dérouter l'exécution en cas de débordement arithmétique. La plupart des instructions modifient les bits du registre d'état. L'instruction `cmp` a la particularité de ne modifier que les bits du registre d'état.

4.1.6 Il serait plus simple d'écrire les branchements de la façon suivante. C'est typiquement le genre d'amélioration à appliquer pour simplifier le code. (Probablement que l'étiquette `x` devient inutile, mais il faudrait connaître le reste du code pour l'affirmer.)

```
y:      ....
        cmp.w d2,d4
        beq   y
x:
```

4.1.7 L'instruction `bhi` est principalement employée pour brancher dans le cas de nombres binaires purs, tandis que `bgt` pour le branchement dans le cas de nombres codés en complément à deux. Dans les deux cas il s'agit d'un branchement si « plus grand que ».

4.1.8 Il y a branchement quand `d1` est plus grand que `d0` (en considérant les opérandes codés en complément à deux).

4.2.1 Remarquez que les solutions données ne sont que des bouts de programme, en d'autres mots, une séquence d'instructions qui fait strictement ce que le pseudo-code signifie.

a)

```
        cmpi.w   #0,x      ; x > 0
        ble      sinon
        move.w   #1,x      ; x = 1
        bra      cont
sinon:  move.w    #2,x     ; x = 2
cont:
```

b)

```
        cmpi.w   #10,x     ; x < 10 ?
        bge      sinon
        cmpi.w   #100,y    ; y > 100
        ble      sinon
        move.w   x,z
        move.w   y,d0
        add.w    d0,z      ; z = x + y
        bra      cont
sinon:  move.w   x,z
        move.w   y,d0
        sub.w    d0,z      ; z = x - y
cont:
```

c)

```
        cmpi.w   #9,x       ; x < 9 ?
        blt      alors
        cmpi.w   #-45,y     ; y < -45
        bge      sinon
alors:  move.w   #0,z       ; z = 0
        bra      cont
sinon:  move.w   x,z        ; z = x
cont:
```

d)

```
       cmpi.w    #-1,x      ; x < -1 ?
       bge       sinon2
       cmpi.w    #77,y      ; y > 77 ?
       ble       sinon1
       move.w    #1,y       ; y = 1
       bra       cont
sinon1:move.w    #0,z       ; z = 0
       bra       cont
sinon2:move.w    #0,y       ; y = 0
cont:
```

4.3.1 Voici un programme qui lit deux nombres entiers et affiche le pgcd de ces deux nombres. L'attribution des registres est `d1.w` pour a et `d2.w` pour b. On n'a fait aucune transformation pour améliorer le pseudo-code. En effet il est possible d'améliorer ce code assembleur en éliminant quelques instructions.

```
        xref    stop,decout,decin
        jsr     decin
        move.w  d0,d1         ; la valeur pour a
        bmi     fin           ; arrêt si négatif
        jsr     decin
        move.w  d0,d2         ; la valeur pour b
        bmi     fin           ; arrêt si négatif
tantque:
        cmp.w   #0,d1         ; a ≠ 0?
        beq     finTantque
        cmp.w   #0,d2         ; b ≠ 0?
        beq     finTantque
        cmp.w   d2,d1         ; a > b?
        bls     sinon
        and.l   #$0000ffff,d1; a sur 32 bits pour la division qui suit
        divu.w  d2,d1         ; d1.h = reste de a / b
        swap    d1            ; apporter le reste dans d1.w
        bra     tantque
sinon:  and.l   #$0000ffff,d2; b sur 32 bits pour la division qui suit
```

```
        divu.w  d1,d2       ; d2.h = reste de b /a
        swap    d2          ; apporter le reste dans d2.w
        bra     tantque
finTantque:
        cmp.w   #0,d1       ; a ≠ 0 ?
        beq     sinon2
        move.w  d1,d0
        jsr     decout      ; afficher a
        bra     fin
sinon2: cmp.w   #0,d2       ; b ≠ 0 ?
        beq     sinon3
        move.w  d2,d0
        jsr     decout      ; afficher b
        bra     fin
sinon3: move.w  #1,d0
        jsr     decout      ; afficher 1
fin:    jsr     stop        ; arrêt du programme
        end
```

4.3.2

```
        xref    stop,decin,strout,decout
        jsr     decin           ; n
        cmp.w   #0,d0           ; n < 0 ?
        bge     cont
        move.w  #msgfin-msg,d0
        lea     msg,a0
        jsr     strout
        bra     fin
cont:   move.w  #0,d1           ; x = 0
        move.w  #0,d2           ; y = 0
tantque: add.w  d2,d1
        add.w   d2,d1
        add.w   #1,d1           ; x = x + 2y + 1
        cmp.w   d1,d0           ; x + 2y + 1 <= n ?
        blt     finTantque
        add.w   #1,d2           ; y = y + 1
        bra     tantque
finTantque:
        move.w  d2,d0
        jsr     decout          ; afficher y
fin:    jsr     stop
        data
msg     dc.b    'Ne peut calculer la racine carree d'un negatif',10,13
msgfin
        end
```

4.4.1 Voici une solution qui fontionne pour le système x68k (car les entrées/sorties utilisent la méthode des trap). Il n'y a aucun message affiché, ni pour l'entrée ni pour la sortie.

```
;
; Lire n nombres x_i et afficher le nombre de sommets.
; Un nombre x_i est un sommet ssi x_{i-1} < x_i <= x_{i+1}
; (quand x_{i-1} et x_{i+1} existe).
; On utilise les registres de la façon suivante:
; d1 : x_{i+1}
; d2 : x_i
; d3 : x_{i-1}
          org     $400
debut:
          clr.l   d4            ; zéro sommet au début de l'exécution
          move.w  #32767,d3     ; pour démarrer l'itération
          move.w  d3,d2         ; on force une position élevée
          move.b  #4,d0         ; code pour lire
          trap    #15           ; lire nombre de nombres (n)
          move.w  d1,d5         ; pour le garder dans d5
          sub.w   #1,d5         ; pour dbra
          bmi     finIteration  ; aucun nombre, on arrête
prchNombre:
          move.b  #4,d0         ; code pour lire un nombre
          trap    #15           ; lire x_{i+1} (d1)
          cmp.w   d2,d3         ; (d3)x_{i-1} < (d2)x_i ?
          bge     passe         ; non, on passe
          cmp.w   d1,d2         ; (d2)x_i ≥ (d1)x_{i+1} ?
          blt     passe         ; non, on passe
          add.l   #1,d4         ; oui, un sommet de plus
passe:
          move.w  d2,d3         ; x_i devient x_{i-1}
          move.w  d1,d2         ; x_{i+1} devient x_i
          dbra    d5,prchNombre; un nombre en moins
finIteration:     ; ici, tous les nombres sont lus
          move.b  #3,d0         ; code pour afficher
          move.l  d4,d1         ; pour afficher le nombre de sommets
          trap    #15           ; affiche nombre de sommets
          move.b  #9,d0         ; code pour terminer
          trap    #15           ; termine
          end     debut
```

5.1.1

```
; Entrée :    d0.l, d1.l deux nombres entiers de 32 bits.
; Sortie :    d0.l la plus petite valeur de d0.l et d1.l
```

```
;
min:          cmp.l   d0,d1        ; quel paramètre est le plus petit?
              bge     d0Petit
              move.l  d1,d0        ; d1 contient une plus petite valeur.
d0Petit:
              rts     ; retour à l'appeleur
```

5.1.2

```
; Entrée :d0.l contient un nombre entier. (complément à deux)
; Sortie :d0.l contient la valeur absolue de d0.l. (binaire pur)
;
abs:  tst.l   d0           ; est-ce positif?
      bpl     okPositif
      neg.l   d0           ; le rendre positif.
okPositif:
      rts                  ; retour à l'appeleur
```

5.1.3

```
; Entrée :d0.w, d1.w, deux nombres entiers x et y, codés en binaire pur.
; Sortie :d0.w contient le pgcd de x et y.
;        ( Note pgcd(0,0) = 1. )
;
pgcd:     movem.l d1-d2,-(sp)  ; sauvegarde les registres modifiés
          and.l   #$ffff,d0    ; partie haute de d0 à zéro. (pour divu.w)
          beq     retourPgcd1  ; si x = 0 Alors aucun calcul à faire
          and.l   #$ffff,d1    ; partie haute de d0 à zéro. (pour divu.w)
          beq     retourPgcd2  ; si y = 0 Alors d0 contient le pgcd
          cmp.w   d0,d1        ; interchange x et y?
          bls     okOrdre
interchange:
          exg.l   d0,d1        ; interchange pour avoir le plus petit dans d1
okOrdre:
          divu.w  d1,d0        ; calcule le reste(le quotient n'est pas utile)
          clr.w   d0           ; élimine le quotient pour le prochain divu.w
          swap    d0           ; place le reste dans la partie basse de d0
          bne     interchange  ; d0 contient la plus petite valeur
retourPgcd1:
          tst.w   d1           ; si x=0 et y=0 Alors pgcd = 1
          bne     okD1NonZero
          move.w  #1,d1        ; cas particulier où d1 et d0 = 0
okD1NonZero:
          move.w  d1,d0        ; d1 est différent de 0, c'est le pgcd
retourPgcd2:
```

```
        movem.l (sp)+,d1-d2  ; rétablir les registres
        rts
```

5.1.5

```
; Entrée :    d0.w contient la valeur n.
; Sortie :    le message "Pas de panique" est affiché n fois.
              xref    strout
AffichePasdePanique:
              movem.l a0/d0-d1,-(sp)    ; sauvegarder les registres modifiés
              lea     msgPasdePanique,a0; charger adresse du message
              move.w  d0,d1             ; copier n
              beq     finAffiche        ; si n=0 retourner immédiatement
              move.w  #16,d0            ; longueur du message
UnautreFois:
              jsr     strout            ; afficher message "Pas de panique"
              sub.w   #1,d1             ; compter le nombre d'itérations
              bne     UnAutreFois       ; si d1≠ 0 afficher encore
finAffiche:
              movem.l (sp)+,a0/d0-d1    ; rétablir les registres modifiés
              rts                       ; retourner à l'appeleur
              ...
              data
              ...
msgPasdePanique: dc.b 'Pas de panique',10,13
              ...
              end
```

5.1.6 Cela permet de simplifier la tâche de l'appel à une sous-routine. Autrement, il faudrait sauvegarder, avant l'appel d'une sous-routine, le contenu de tous les registres en utilisation.

5.1.7 Un paramètre formel est un registre. Le paramètre actuel est la valeur qui sera placée par l'appelleur dans ce registre.

5.1.8 Oui. Une sous-routine peut donc retourner, par l'instruction `rts`, à une autre sous-routine et non nécessairement dans la partie principale du programme.

5.1.9 Oui. Pour un même programme source, les instructions des sous-routines internes ne sont pas différentes des instructions de la partie principale. Dans le cas où la partie `data` est accessible par l'entremise du registre `a5` (e.g. MAS 2.0), les étiquettes définies dans cette partie sont accessibles par toutes les instructions d'un même programme source. Et dans le cas où les instructions

accèdent à la partie `data` par des adresses absolues (e.g. l'émulateur X68k), ces mêmes adresses absolues peuvent être utilisées par une sous-routine interne.

5.1.10 Oui, car on peut concevoir une sous-routine à plusieurs points de sortie. L'instruction `rts` ne marque pas nécessairement « la fin » d'une sous-routine, elle est plutôt utilisée pour exécuter un retour à l'appeleur.

5.2.1 La méthode consiste à diviser y par x ; si le reste de cette division est zéro, x est un diviseur de y, sinon x n'est pas un diviseur de y. Les deux opérandes sont des nombres signés, nous devrions donc utiliser `divs`, mais lors d'une division de deux nombres signés de 16 bits, il existe un cas de débordement : si $x = -1$ et $y = -2^{16}$ alors l'opération de division déborde car le quotient 2^{16} ne peut être codé sur 16 bits; cela produit un reste non défini. En fait, l'opération `divu` fait mieux le travail, car même si les valeurs sont signées, cette opération donne un reste à zéro si et seulement x divise y. Ainsi nous utilisons `divu` et évitons ce débordement rare.

```
; Entrée :    d0.w, nombre entier signé x
;             d1.w, nombre entier signé y
;
; Sortie :    Z = 1 si et seulement si x est un diviseur de y
;             si x = 0 alors Z est quelconque (indéfini)
;             si x <>0 et y = 0 alors Z = 0
DiviseurQ:   movem.l d1,-(sp)     ; sauvegarde d1
             tst.w   d0           ; pour le cas x=0, éviter la division
             beq     retour       ; (retourne Z=1 mais c'est sans importance)
             ext.l   d1           ; y sur 32 bits
             divu.w  d0,d1        ; c'est le reste de y/x qui est utile
             swap    d1           ; apporte le reste dans la partie basse
             tst.w   d1           ; est-ce un reste à zéro?
;
; Ici le bit Z est correctement positionné, car si d1.w, le reste, est zéro
; le bit Z est à 1, ce qui indique que x est diviseur de y
;
retour:      movem.l (sp)+,d1     ; rétablir d1
             rts     ; retour à l'appeleur
```

5.2.2 Rappelons le pseudo-code (c'est en langage C).

```
int seed = 1;
int random()
{
  int hi = seed / 127773;
  int lo = seed % 127773;
```

```
    int x = lo*16807 - hi*2836;
    if (x<0) seed = x + 2147483648;
    else seed = x;
    return seed;
  }
```

La variable `seed` a 32 bits. En fait, une analyse du code nous montre que ses valeurs possibles sont dans l'intervalle $[0, 2^{31} - 1]$. Une preuve par induction le démontre. C'est vrai pour le premier appel : `seed` vaut 1. Supposons que `seed` est positif et vaut moins de 2^{31} à l'entrée de `random`. Alors, `hi` vaut au maximum 16807. La variable `lo` a une valeur inférieure à 127773 indépendamment de ce fait. Ainsi la valeur de l'expression `lo*16807 - hi*2836` assignée à `x` est comprise dans l'intervalle [-47 664 652, 2 147 480 811]. Dans le cas où la valeur de `x` est négative, elle est remise positive par l'addition de 2^{31}. Ainsi, la valeur de `x` ne peut être négative ou dépassée $2^{31} - 1$. Pour la suite de la présentation on peut donc affirmer que le dividende est positif, ce qui simplifie le problème pour le M68000 (voir plus loin) et permet l'usage d'instructions à opérandes codés en binaire pur (e.g. `divu`, `mulu`).

Voici une première solution utilisant des instructions `mulu` et `divu` de 64 bits (pour les processeurs 68020 et plus). C'est une implantation directe du pseudo-code C. Remarquez que nous avons implanté la variable globale `seed` comme le pseudo-code l'indique, i.e. comme espace statique dans la section `data`.

```
;
; Partie principale pour afficher 100 nombres aléatoires.
;
            xref    decout_long,newline,stop
            move.w  #99,d1        ; 100 nombres aléatoires à afficher
prchNombre:
            jsr     random        ; prochain nombre aléatoire
            jsr     decout_long   ; affiche ce nombre de 32 bits
            jsr     newline       ;
            dbra    d1,prchNombre; itération ...
            jsr     stop          ; arrêt du programme
;
; Entrée :  aucun paramètre, mais accès à la variable globale seed.
; Sortie :  d0.l, la nouvelle valeur de seed (seed a été modifiée)
;           calculée selon le pseudo-code C.
random:     movem.l d1,-(sp)      ; sauvegarde le registre modifié
            move.l  seed,d1
            divul.l #127773,d0:d1; hi(d1) et lo(d0) (seed est positif)
            mulu.l  #16807,d0     ; lo*16807
            mulu.w  #2836,d1      ; hi*2836
            sub.l   d1,d0         ; x = lo*16807 - hi*2836
            bpl     retour        ; si (x<0) ...
```

```
            add.l   #2147483648,d0 x + 2147483648
retour:     move.l  d0,seed       ; seed = x
            movem.l (sp)+,d1      ; rétablir le registre modifié
            rts     ; retour à l'appeleur
;                   data
seed:       dc.l    1             ; valeur de départ pour seed
            end
```

Regardons le problème de l'implantation de cet algorithme pour le M68000, en particulier la difficulté de diviser avec un diviseur de plus de 16 bits. Rappelons que le M68000 n'a qu'un format pour l'opération de division : un diviseur de 16 bits et un dividende de 32 bits résultant en un quotient et un reste, chacun de 16 bits.

La division par 127773 présente une difficulté : il a plus de 16 bits. Toutefois, il se décompose en deux facteurs de moins de 16 bits : 42591 et 3. Pour effectuer cette division où $d = d_1 d_2$ on peut faire deux divisions $n = q_1 d_1 + r_1$ et $q_1 = q_2 d_2 + r_2$ [1]. Le quotient q_2 est le quotient q recherché, car par substitution : $n = (q_2 d_2 + r_2) d_1 + r_1 = q_2 d_2 d_1 + r_2 d_1 + r_1 = q_2 d + (r_2 d_1 + r_1)$ où $r_2 d_1 + r_1 < d$. (Ce dernier fait prouve que q_2 est le quotient recherché car un reste inférieur à d implique que q_2 est le quotient). Nous avons donc une méthode de décomposition où les diviseurs sont représentables sur 16 bits en binaire pur. Remarquez que la valeur `seed` ne dépasse pas 2^{31}, ainsi le premier quotient ne peut dépasser $2^{31}/42591 = 50421$; c'est représentable sur 16 bits. Le reste a besoin, en général, d'un espace de 32 bits.

Voici le code assembleur pour la fonction `random` seulement.

```
; VERSION M68000
; Entrée : aucun paramètre, mais accès à la variable globale seed.
; Sortie : d0.l, la nouvelle valeur de seed (seed a été modifiée)
;          calculée selon le pseudo-code C.
;
random:
           movem.l d1-d2,-(sp)  ; sauvegarde registres modifiés
           move.l  seed,d0
           divu.w  #42591,d0    ; division par le grand facteur de 127773
           move.l  d0,d1        ; pour conserver le reste intermédiaire
           and.l   #$ffff,d0    ; élimine reste et prépare division suivante
           divu.w  #3,d0        ; voici le quotient final de seed/127773
           move.w  d0,d2        ; conserve quotient final
           swap    d0           ; deuxième reste
           mulu.w  #42591,d0    ; c'est une partie du reste final
           clr.w   d1           ; prépare l'addition sur 32 bits
```

1. Rappelons que si n est le dividende et d le diviseur, nous avons l'identité $n = qd + r$, où q est le quotient et $r < d$, le reste.

```
          swap    d1           ; du premier reste au deuxième reste
          add.l   d1,d0        ; reste final de seed/127773
;
; Le reste r dans d0 est maintenant traité en deux parties :
; r = r.h * 2^16 + r.l
; pour effectuer une multiplication en deux parties
; r*16807 = (r.h * 2^16 + r.l) * 16807
; r*16807 = (décaler à gauche de 16 bits r.h*16807) + r.l * 16807
;
          move.w  d0,d1        ; partie basse du reste
          mulu.w  #16807,d1    ;
          swap    d0           ; partie haute du reste
          mulu.w  #16807,d0    ;
          lsl.l   #8,d0        ;
          lsl.l   #8,d0        ;
          add.l   d1,d0        ; résultat de lo*16807
          mulu.w  #2836,d2     ; hi*2836
          sub.l   d2,d0        ; x = lo*16807 - hi*2836
          bpl     retour
          add.l   #$80000000,d0 ; force seed à être positif
retour:
          move.l  d0,seed      ; tel que le demande le pseudo-code
          movem.l (sp)+,d1-d2  ; rétablir registres pour l'appeleur
          rts     ; retour à l'appeleur
```

Finalement, les 100 premiers nombres aléatoires générés sont :

16807 282475249 1622650073 984943658 1144108930 470211272 101027544 1457850878
1458777923 2007237709 823564440 1115438165 1784484492 74243042 114807987 1137522503
1441282327 16531729 823378840 143542612 896544303 1474833169 1264817709 1998097157
1817129560 1131570933 197493099 1404280278 893351816 1505795335 1954899097 1636807826
563613512 101929267 1580723810 704877633 1358580979 1624379149 2128236580 784575628
812987216 1585177098 389361404 614444619 1869336757 257119289 666792459 1211188367
445394256 1760750797 613989519 662921998 584859750 705165931 1909038171 1798853817
1061319853 603597389 2096052142 1026605206 1270077244 200788728 959342059 354763937
1102885087 1258299952 1975821255 1088199224 1393619916 2113274030 564584477 1388552493
696957902 1395648176 1842501498 208487146 1497634565 88307468 272414599 37029989
1740251140 1821121487 1651894965 730088339 2020638262 560875776 1333440549 2143450599
936038968 1689220901 1001869767 5897842 340782732 196490175 1728005786 24403274
2123933188 1471910282 1531979781 1802735384

6.1.1

```
; Entrée :     a0 pointe une chaîne de caractères se terminant par <nul>.
; Sortie :     bit Z est à 1 si et seulement s'il y a un caractère >127.
;
```

```
DetecteCarNonStandard:
              movem.l a0,-(sp)
prchCar:      tst.b   (a0)+        ; caractère <nul>?
              beq     finDetecte
              bpl     prchCar      ; si bit 7 à 0 regarder prochain car
              ori.w   #4,ccr       ; mettre le bit Z à 1
              bra     retour
finDetecte:   andi.w  #$fffb,ccr   ; mettre le bit Z à 0
retour:       movem.l (sp)+,a0
              rts     ; retourner à l'appeleur
```

6.1.2 Voici un pseudocode possible.

```
   Fonction AdrSousChaine(t, s : chaine de caracteres):adresse memoire.
   Var i, j : entier;
   Debut
1  i = 0;
2  Faire iterativement
     Debut
3    j = 0;
4    Tantque s[j] <> 0 et s[j] = t[i+j] Faire j = j+1;
5    Si s[j]   = 0 Alors retourner(adresse de t[i]);
6    Si t[i+j] = 0 Alors retourner(0)
7    i = i+1;
     Fin
   Fin
```

Un premier exercice intéressant est de convertir l'algorithme en utilisant des indices `i` et `j`, et en traduisant litéralement les accès indicés à `s` et `t`. Cela n'est pas très performant mais démontre la simplicité de la méthode. Notez que la condition de l'instruction 5 n'est pas traduite puisqu'elle est incluse dans l'instruction 4.

```
; Entrée :    a0 pointe chaîne t.
;             a1 pointe chaîne s.
; Sortie :    a0 contient adresse début, dans t, de la sous-chaîne s ou
;             contient 0 si la sous-chaîne s n'est pas dans t.
;
; Var i :     d0.w
; j :         d1.w
;
AdrSousChaine:
              movem.l a2/d0-d3,-(sp)
              move.w  #0,d0        ; i = 0;
iteratif:     move.w  #0,d1        ; j = 0;
tantque:      movea.l a1,a2
```

```
            adda.w  d1,a2        ; a2 = adresse de s[j]
            move.b  0(a2),d3     ; s[j]
            beq     retAdr       ; s[j] <>0?
            movea.l a0,a2        ; prépare le calul de l'adr t[i+j]
            move.w  d0,d2        ; calculer i+j
            add.w   d1,d2
            adda.w  d2,a2        ; a2 =adresse de t[i+j]
            cmp.b   0(a2),d3     ; s[j] = t[i+j]?
            bne     finTantque
            add.w   #1,d1        ; j = j+1
            bra     tantque
finTantque: cmpi.b  #0,0(a2)     ; Si t[i+j] = 0 Alors ...
            beq     retNil
            add.w   #1,d0        ; i = i+1
            bra     iteratif
retNil:     movea.w #0,a0        ; retourner(0)
            bra     finAdr
retAdr:     adda.w  d0,a0        ; retourner(adresse t[i])
finAdr:     movem.l (sp)+,a2/d0-d3
            rts
```

Pour améliorer le code on utilise des adresses réelles au lieu des variables indices `i` et `j`. En fait, l'indice `i` sera le pointeur passé en paramètre. En plus, le test de l'instruction 5 n'est pas traduit puisqu'il est fait à l'instruction 4. Le tout donne quelque chose comme suit.

```
; Entrée :   a0 pointe chaîne t.
;            a1 pointe chaîne s.
; Sortie :   a0 contient adresse début, dans t, de la sous-chaîne s ou
;            contient 0 si la sous-chaîne s n'est pas dans t.
;
; Var i, j :
;            Adresse de t[i] dans a0.
;            Adresse de s[0] dans a1. (ne change pas)
;            Adresse de s[j] dans a2.
;            Adresse de t[i+j] dans a3.
AdrSousChaine:
            movem.l a2-a3/d0,-(sp)
iteratif:   movea.l a1,a2        ; a2 = adresse de s[j]
            movea.l a0,a3        ; a3 = adresse de t[i+j]
tantque:    move.b  0(a2),d0
            beq     retAdr       ; s[j] <>0?
            cmp.b   0(a3),d0     ; s[j] = t[i+j]?
            bne     finTantque   ;
            lea     1(a3),a3     ; j = j+1
            lea     1(a2),a2
```

```
            bra     tantque
finTantque:
            cmpi.b  #0,0(a3)     ; Si t[i+j] = 0 Alors
            beq     retNil       ; ...
            lea     1(a0),a0     ; i = i+1
            bra     iteratif
retNil:     movea.w #0,a0        ; ... retourner(0)
retAdr:     movem.l (sp)+,a2-a3/d0
            rts
```

Finalement, il est encore possible d'améliorer ce code en éliminant les instructions `lea` et en utilisant le mode post-incrément `(an)+`. Voici cette dernière version.

```
; Entrée :   a0 pointe chaîne t.
;            a1 pointe chaîne s.
; Sortie :   a0 contient adresse début, dans t, de la sous-chaîne s ou
;            contient 0 si la sous-chaîne s n'est pas dans t.
;
; Var i, j : (ces deux variables n'existent plus, mais sont remplacées par...)
;            Adresse de t[i] dans a0.
;            Adresse de s[0] dans a1. (ne change pas)
;            Adresse de s[j] dans a2.
;            Adresse de t[i+j] dans a3.
AdrSousChaine:
            movem.l a2-a3/d0,-(sp)  ; sauvegarder les registres
iteratif:   movea.l a1,a2           ; a2 = adresse de s[j]
            movea.l a0,a3           ; a3 = adresse de t[i+j]
tantque:    move.b  (a2)+,d0
            beq     retAdr          ; Si s[j] = 0 Alors retourner(adr t[i])
            cmp.b   (a3)+,d0        ; s[j] = t[i+j]?
            beq     tantque         ;
finTantque:
            lea     1(a0),a0        ; nouvelle adresse de t[i]. (i = i+1)
            tst.b   0(a3)           ; Si t[i+j] = 0 Alors
            bne     iteratif        ;
            movea.w #0,a0           ; ... retourner(0)
retAdr:     movem.l (sp)+,a2-a3/d0  ; rétablir les registres
            rts                     ; retourner à l'appeleur
```

6.2.1 La solution suivante utilise des registres d'adresses qui pointent directement le vecteur V.

```
      lea     V,a0          ; a0 pointe V[0]
      move.w  #999,d0       ; 1000 itérations
```

```
      move.l  x,d1
      move.l  y,d2
pouri:cmp.l   (a0)+,d1     ; V[i] <x?
      ble     cont
      move.l  #1,-4(a0)    ; V[i] = 1
cont: cmp.l   (a0),d2      ; V[i+1] >y?
      bge     cont2
      move.l  #2,(a0)      ; V[i+1] = 2
cont: dbra    d0,pouri
```

6.2.2

```
; Entrée :     a0 pointe un vecteur V; a1 pointe un vecteur W;
;              d0.l contient le nombre d'octets de V (i.e. sa longueur)
;              (W a la même longueur)
; Sortie :     V a été copié à l'adresse de W
;
CopieVW:       movem.l a0-a1/d0,-(sp)
               tst.l   d0             ; longueur = 0?
               beq     fin            ; retour immédiat si vecteurs vides
               cmpa.l  a0,a1          ; comparer les adresses des vecteurs
               beq     fin            ; même adresse donc rien à copier
               bhi     copierARec     ; a1>a0 =>V est plus bas que W en mémoire
;
; Si le vecteur V est plus haut que W en mémoire il suffit
; de copier le vecteur V vers W en avançant. Cela résout le
; problème potentiel de chevauchement.
;
copierAv:      move.b  (a0)+,(a1)+    ; copie un octet de V à W (en avançant)
               sub.l   #1,d0          ; un octet de moins
               bne     copierAv       ; copie tous les octets de V
               bra     fin
;
; Si le vecteur V est plus bas que W en mémoire il suffit
; de copier le vecteur V vers W en reculant. Cela résout le
; problème potentiel de chevauchement.
;
copierArec:    adda.l  d0,a0          ; pointer passé V
               adda.l  d0,a1          ; pointer passé W
copieRec:      move.b  -(a0),-(a1)    ; copie un octet de V à W (en reculant)
               sub.l   #1,d0          ; un octet de moins
               bne     copieRec       ; copie tous les octets de V
fin:           movem.l (sp)+,a0-a1/d0 ; rétablir les registres modifiés
               rts                    ; rerourner à l'appeleur
```

6.2.3

```
; Entrée :   a0 pointe un vecteur v de 200 entiers de 16 bits.
; Sortie :   d0.w contient la valeur maximum de ce vecteur
;
Maximum:    movem.l a0/d1,-(sp)  ; sauvegarder les registres modifiés
            move.w  (a0)+,d0     ; prend le premier élément
            move.w  #198,d1      ; itération sur 199 éléments
rechMax:    cmp.w   (a0)+,d0     ; max >v[i]?
            bge     passe
            move.w  -2(a0),d0    ; non, max = v[i]
passe:      dbra    d1,rechMax   ; itère sur tout le vecteur v
            movem.l (sp)+,a0/d1  ; rétablir les registres modifiés
            rts     ; retourner à l'appeleur
```

6.2.4

```
          lea     v,a0         ; v est l'étiquette du début du vecteur
          move.w  #199,d1      ; boucle de 200 itérations
          move.l  #$80000000,d0; la plus petite valeur sur 32 bits
prchi:    move.l  (a0)+,d2     ; v[i] ; en plus avance a0 sur suivant
          sub.l   (a0),d2      ; v[i]-v[i+1]
          cmp.l   d2,d0
          bge     d0plusgrandegal
          move.l  d2,d0        ; d2 est plus grand, on le garde
d0plusgrandegal:
          dbra    d1,prchi     ; itération
          ; ici d0 contient le résultat
```

6.2.5 La solution suivante utilise des pointeurs plutôt que des indices `i` et `j`. Ces deux variables sont toujours utiles, mais des registres d'adresses contiennent les adresses aux éléments `v[i]` et `v[j]`.

```
         move.w  #0,x         ; x = 0
         lea     V,a0         ; a0 = adresse de V[i], au départ adresse de V[0]
         move.w  #99,d0       ; 100 itérations sur i
pouri:   lea     V,a1         ; a1 = adresse de V[j], au départ adresse de V[0]
         move.w  #99,d1       ; 100 itérations sur j
pourj:   move.w  (a1)+,d2     ; prend V[j]
         cmp.w   (a0),d2      ; V[i] >V[j]?
         bge     passe
         add.w   d2,x         ; x = x + V[j]
passe:   dbra    d1,pourj     ; itération sur j
         lea     2(a0),a0     ; passe au prochain élément dans V
```

```
        dbra    d0,pouri      ; itération sur i
```

6.2.6

```
        move.w  #1,d0         ; x^0 = 1
        move.w  #14,d1        ; 15 itérations
        lea     xp,a0         ; pointe le début du vecteur xp
prchP:  move.w  d0,(a0)+      ; xp[i] = x^i; i = i + 1
        muls.w  x,d0          ; x^i * x donne x^(i+1)
        dbra    d1,prchP
```

6.2.7 Cette solution utilise amplement les pointeurs et évite les indices explicites. Remarquez l'utilisation du mode d'adressage post-incrément pour éviter les instructions explicites d'augmentation des adresses. Cela provoque l'usage de déplacement négatif pour certaines instructions faisant des références ultérieures aux mêmes éléments. (Remarque: rappelons que la syntaxe `-4(a0)` indique une référence mémoire sans modifier le registre a0, mais que la syntaxe post-incrémente `(a0)+` fait une référence mémoire et ensuite modifie le registre a0.)

```
          lea     v,a0          ; adresse de v[0] pour v[i]
          lea     v+(499*4),a2  ; adresse de v[499] pour teste d'arrêt
prchi:    move.l  (a0)+,d0      ; min = v[i]; avance a0 prochain élément
          movea.l a0,a1         ; a1 adresse de v[j] (a0 pointe déjà plus loin)
prchj:    cmp.l   (a1)+,d0      ; Si min >v[j] Alors ...
          ble     min0k
          move.l  -4(a1),d0     ; min = v[j] ( -4(a1) car a1 pointe plus loin )
min0k:    cmpa.l  a2,a1         ; la fin du vecteur v est dépassée pour j?
          bls     prchj
          move.l  d0,-4(a0)     ; v[i] = min ( -4(a0) car a0 pointe déjà plus loin )
          cmpa.l  a2,a0         ; la fin du vecteur v est dépassée pour i?
          bne     prchi         ; il faut arrêter sur v[498] et non v[499]
                                ; ... ici le vecteur v est trié en ordre croissant ...
```

7.1.1 L'espace mémoire occupe 200 mots de 32 bits qui est suffisant pour contenir une matrice de 10 lignes et 20 colonnes ou une matrice de 20 lignes et 10 colonnes d'entiers de 32 bits. Toutefois, une matrice de 20 lignes et 20 colonnes d'entiers de 32 bits a besoin de 400 mots de 32 bits.

7.1.2 Il suffit d'ajouter la valeur $2 \times (4 \times 10 + 3) = 86$ à l'adresse contenue dans `a0` pour obtenir l'adresse de l'élément $M[4,3]$. (Rappelons que les numéros des indices des lignes et colonnes débutent à 0.) Dans l'expression $2 \times (4 \times 10 + 3)$,

le 10 provient du nombre de colonnes, le 4 est le numéro de la ligne, le 3 le numéro de colonne et le 2 est le nombre d'octets occupé par chaque élément de la matrice (un mot de 16 bits). Il n'est pas nécessaire d'explicitement additionner la valeur 86 à `a0`. Il suffit décrire `move.w #-1,86(a0)` pour assigner -1 à $M[4,3]$ sans modifier `a0`. De façon équivalente, on peut écrire `move.w #-1,(86,a0)`.

7.2.1

```
          lea     M+12*4,a0    ; a0 pointe l'entier M[0,12]
          clr.l   d0           ; somme à zéro
          move.w  #49,d1       ; 50 lignes
lignes:   add.l   (a0),d0      ; colonne 12
          add.l   (4*4,a0),d0  ; colonne 16
          add.l   (6*4,a0),d0  ; colonne 18
          add.l   (7*4,a0),d0  ; colonne 19
          lea     (20*4,a0),a0 ; passe à la ligne suivante
          dbra    d1,lignes
```

7.2.2

```
; Entrée :      a0 pointe une matrice M de 25 lignes et 35 colonnes d'entiers
;               de 16 bits
;               a1 pointe une zone de 25*35*4 = 3500 octets
; Sortie :      a1 pointe une copie dans la zone de M dont les entiers
;               sont étendus sur 32 bits
;
CopieEtendue:  movem.l a0-a1/d0-d1,-(sp)
               move.w  #25*35-1,d0  ; nombre d'éléments à traiter
prchEntier:    move.w  (a0)+,d1
               ext.l   d1           ; de 16 bits à 32 bits signé
               move.l  d1,(a1)+
               dbra    d0,prchEntier
               movem.l (sp)+,a0-a1/d0-d1
               rts
```

7.2.3

```
              lea     M,a1         ; adresse de la matrice
              lea     V,a0         ; adresse des adresses
              move.w  #99,d0       ; 100 vecteurs à copier
prchVecteur:  movea.l (a0)+,a2     ; charger l'adresse d'un vecteur
              move.w  #49,d1       ; 50 entiers à copier pour ce vecteur
```

```
prchNombre:   move.l  (a2)+,(a1)+  ; déplacer un entier dans la matrice
              dbra    d1,prchNombre; prochain nombre
              dbra    d0,prchVecteur; prochain vecteur
```

7.2.4 Voici une solution en pseudocode.

```
Fonction PointMinMax(M : matrice d'entiers, 5 par 7): adresse;
Var e, jmax, i, j, k : entier;
Debut
Pour i de 1 a 5 Faire
  Debut
  (* trouver l'element maximum dans la ligne i *)
  (* e compte le nombre d'occurences du maximum. Le maximum devrait
     apparaitre une seule fois pour etre accepte. *)
  jmax = 1; e = 0;
  Pour j de 2 a 7 Faire Si M[i,j] > M[i,jmax] Alors jmax = j; e = 0;
                        Sinon Si M[i,j] = M[i,jmax] Alors e = e+1;
  Si e = 0 Alors
    Debut
    (* Regarder si l'element M[i,jmax] est minimum dans la colonne jmax *)
    k = 1;
    Tantque (k <= 5 et M[i,jmax] < M[k,jmax]) ou k=i Faire k = k+1
    Si k = 6 Alors retourner adresse M[i,jmax]
    Fin
  Fin
retourner(NIL);
Fin PointMinMax.
```

Voici une traduction en assembleur où on suppose que les entiers ont 16 bits. Le code a été traduit en faisant quelques transformations. La plus importante est l'utilisation d'adresses plutôt que d'indices. Ainsi, les variables `j` et `jmax` ne sont pas directement représentés dans le code assembleur. Il s'agit plutôt des adresses des éléments indicés gardées dans les registres d'adresses.

```
; Entrée :      a0 pointe une matrice d'entiers de 16 bits de
;               5 lignes et 7 colonnes.
; Sortie :      a0 adresse de M[i,j] où M[i,j] est un MinMax.
;
; Var
;               adresse de M[i,jmax] : a0
;               adresse de M[i,j] : a1
;               adresse de M[k,jmax] : a2
;               adresse de M[i+1,1] : a3
;               registre de travail : d0
```

```
;               e ou k : d1
;               i : d2
;
PointMinMax:    movem.l a1-a3/d0-d2,-(sp)
                move.w  #1,d2         ; i = 1;
                lea     14(a0),a3     ; adresse de M[2,1]
pouri:
                move.w  #0,d1         ; e = 0
                lea     2(a0),a1      ; adresse de M[i,j] (j = 2)
pourj:          move.w  0(a1),d0      ; charge M[i,j]
                cmp.w   0(a0),d0      ; M[i,j] >M[i,jmax]?
                ble     sinon         ;
                lea     0(a1),a0      ; modifie adr pour M[i,jmax] (jmax = j)
                move.w  #0,d1         ; e = 0
                bra     prchj
sinon:          bne     prchj         ; M[i,j] = M[i,jmax]?
                add.w   #1,d1         ; e = e + 1
prchj:          lea     2(a1),a1      ; adr pour M[i,j] (j = j+1)
                cmpa.l  a3,a1         ; Si j <= 7 alors pourj
                blo     pourj
                cmp.w   #0,d1         ; e = 0?
                bne     prchi
;
; Regarder si l'élément M[i,jmax] est minimum dans la colonne jmax
; a0 pointe M[i,jmax]
;
                move.w  d2,d0         ; calculer adresse de M[1,jmax]
                sub.w   #1,d0         ; dans a2
                mulu.w  #14,d0        ;
                movea.l a0,a2         ;
                suba.w  d0,a2         ;
                move.w  #1,d1         ; k = 1
tantque:        cmp.w   d1,d2         ; k = i?
                beq     cont
                cmpi.w  #6,d1         ; k <= 5?
                beq     retPointMM
                move.w  0(a2),d0      ; charge M[k,jmax]
                cmp.w   0(a0),d0      ; M[i,jmax] <M[k,jmax]?
                ble     prchi
cont:           add.w   #1,d1         ; k = k+1
                lea     14(a2),a2     ; adresse de M[k,jmax]
                bra     tantque
prchi:          addi.w  #1,d2         ; i = i+1
                movea.l a3,a0         ; adresse de M[i,jmax] (jmax = 1)
                lea     14(a3),a3     ; adresse de M[i,1]
                cmpi.w  #5,d2         ; Si i <= 5 alors pouri
                bls     pouri
                movea.w #0,a0         ; retourner(0)
retPointMM:     movem.l (sp)+,a1-a3/d0-d2
```

```
        rts                     ; retourner à l'appeleur
```

7.2.5 Voici un pseudocode possible. Les paramètres n et m spécifient le nombre de lignes et de colonnes respectivement. Le problème principal est la réutilisation de l'espace mémoire de la matrice sans faire usage d'un autre espace mémoire lors des copies. (Pour le reste de la discussion on suppose $n > 1$ et $m > 1$, car sinon il n'y a rien à faire. On note par M^t la matrice transposée.)

Ce qui est intéressant dans ce problème est de regarder attentivement la disposition de la matrice en mémoire.

Par exemple, prenons la matrice de 5 lignes et 3 colonnes

```
 1  2  3
 4  5  6
 7  8  9
10 11 12
13 14 15
```

Sa disposition en mémoire est

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

Si on considère la transposé on a

```
 1  4  7 10 13
 2  5  8 11 14
 3  6  9 12 15
```

Sa disposition en mémoire est

```
1  4  7 10 13  2  5  8 11  14   3   6   9  12  15
```

En plaçant en parallèle ces deux dispositions on voit mieux les remplacements des locations.

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
1  4  7 10 13  2  5  8 11  14   3   6   9  12  15
```

L'algorithme proposé est de balayer l'espace de la matrice originelle mais en considérant déjà les indices de la matrice résultante M^t. Puisque les premier et dernier éléments ne bougent pas, on peut balayer l'espace de l'indice $M^t[1,2]$ à $M^t[m, n-1]$.

Dans le cas simple, pour l'élément $M^t[i,j]$ il suffit de copier l'élément $M[j,i]$. La valeur se trouvant en $M^t[i,j]$ sera copié en $M[j,i]$ pour être redéplacer plus tard si nécessaire. Toutefois, il y a une complication si l'emplacement $M[j,i]$ se trouve en mémoire avant l'emplacement $M^t[i,j]$. On reviendra sur ce point, regardons le tout en action pour mieux voir la méthode.

Voici un trace du contenu de la mémoire après les premiers échanges appliqués sur la matrice ci-haut.

```
etat initial  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
Mt[1,2]       1  4  3  2  5  6  7  8  9  10  11  12  13  14  15
Mt[1,3]       1  4  7  2  5  6  3  8  9  10  11  12  13  14  15
Mt[1,4]       1  4  7 10  5  6  3  8  9   2  11  12  13  14  15
Mt[1,5]       1  4  7 10 13  6  3  8  9   2  11  12   5  14  15
```

Pour $M^t[2,1]$, c'est un cas spécial car $M[1,2]$ se trouve en mémoire avant $M^t[2,1]$. Ainsi, $M[1,2]$ ne contient pas la valeur originelle, cette valeur devrait se trouver à l'emplacement de $M[2,1]$, mais encore une fois cet emplacement est avant $M^t[2,1]$, il faut donc considérer cet emplacement comme appartenant à M^t, c'est-à-dire comme étant $M^t[1,4]$. Ainsi, on considère $M[4,1]$, et dans ce cas cet emplacement est après $M^t[2,1]$ donc contenant la bonne valeur (2). Donc, on échange $M[4,1]$ avec $M^t[2,1]$.

```
              1  4  7 10 13  2  3  8  9   6  11  12   5  14  15
```

Pour $M^t[2,2]$ c'est la même situation, $M[2,2]$ se trouve avant l'emplacement de $M^t[2,2]$. On passe par le même raisonement : on considère $M[2,2]$ comme faisant partie de M^t, donc on trouve que c'est l'emplacement de $M^t[1,5]$, on calcule l'adresse de $M[5,1]$ qui se trouve après $M^t[2,2]$. C'est donc l'endroit où se trouve la bonne valeur (5). On échange $M[5,1]$ avec $M^t[2,2]$ $(3 \leftrightarrow 5)$.

```
              1  4  7 10 13  2  5  8  9   6  11  12   3  14  15
```

Voici le reste de la trace. (L'emplacement $M^t[2,3]$ contenant la valeur 8 ne bouge pas.)

```
8<->8         1  4  7 10 13  2  5  8  9   6  11  12   3  14  15
9<->11        1  4  7 10 13  2  5  8 11   6   9  12   3  14  15
6<->14        1  4  7 10 13  2  5  8 11  14   9  12   3   6  15
9<->3         1  4  7 10 13  2  5  8 11  14   3  12   9   6  15
12<->6        1  4  7 10 13  2  5  8 11  14   3   6   9  12  15
9<->9         1  4  7 10 13  2  5  8 11  14   3   6   9  12  15
12<->12       1  4  7 10 13  2  5  8 11  14   3   6   9  12  15
```

Les dernières étapes apparaissent inutiles, mais elles font partie de l'algorithme général.

Malheureusement l'algorithme n'est pas de complexité linéaire.

Passons à une description sous forme de pseudocode.

```
Procedure Transposer(Var M: matrice d'entiers de 32 bits,
                         n, m: cardinal)
Var aMt, aM, i, j, i1, j1 : cardinal;
Debut
Si n < 2 ou m < 2 Alors retourner (* rien a faire *);
(* i et j sont les indices de Mt *)
i = 1; j = 2;
Tantque i <= m Faire
  Debut
  aMt = adresse de Mt[i,j];
  aM  = adresse de M[j,i];
  Tantque aM < aMt Faire
    Debut
    calculer indices i1, j1 ou adresse Mt[i1,j1] = aM;
    aM = adresse M[j1,i1];
    Fin
  echanger element a aMt avec element a adresse aM;
  j = j+1; Si j = n+1 Alors i = i+1; j = 1;
  Fin
Fin Transposer
```

Voici une traduction en assembleur. Il y a quelques améliorations lors de la traduction pour diminuer les complications. Premièrement, il est préférable d'avoir les indices i et j à partir de zéro. De plus, l'adresse de $M^t[i,j]$ devrait être maintenue dans le registre `a1` et non recalculée à chaque itération.

```
;
; Entrée :      a0 pointe matrice M d'entiers de 32 bits.
;               d0.w nombre de lignes de M, n.
;               d1.w nombre de colonnes de M, m.
; Sortie :      l'espace occupé par M contient sa transposé.
;
; Var
;               aMt : a1
;               aM : a2
;               i : d2.w
;               j : d3.w
;               i1 : d4.w
;               j1 : d5.w
```

```
;
Transposer:     movem.l a1-a2/d2-d5,-(sp); Sauvegarde registres.
                cmp.w   #2,d0           ; n <2?
                blo     retourner
                cmp.w   #2,d1           ; m <2?
                blo     retourner
                move.w  #0,d2           ; i = 1 (recalibré à zéro)
                move.w  #1,d3           ; j = 2 (recalibré à zéro)
                lea     4(a0),a1        ; adresse de Mt[1,2]
tantque:        cmp.w   d2,d1           ; i <= m?
                blo     retourner
                move.w  d3,d5           ; j
                mulu.w  d1,d5           ; * m
                and.l   #$0000ffff,d2   ; d2 sur 32 bits
                add.l   d2,d5           ; + i
                asl.l   #2,d5           ; * 4
                lea     (a0,d5.l),a2    ; adresse de M[j,i]
tantque2:       cmpa.l  a2,a1           ; aM <aMt?
                bls     finTantque2
                asr.l   #2,d5           ; obtenir j*m+i
                divu.w  d0,d5           ; déplacement de M[j,i]/n
                move.w  d5,d4           ; quotient = i1
                swap    d5              ; reste = j1
                mulu.w  d1,d5           ; j1 * m
                and.l   #$0000ffff,d4   ; d4 sur 32 bits
                add.l   d4,d5           ; + i1
                asl.l   #2,d5           ; * 4
                lea     (a0,d5.l),a2    ; adresse de M[j1,i1]
                bra     tantque2
finTantque2:
                move.l  0(a1),-(sp)     ; début interchange
                move.l  0(a2),0(a1)
                move.l  (sp)+,0(a2)     ; (aMt) <->(aM)
                lea     4(a1),a1        ; maintenir adresse Mt[i,j]
                add.w   #1,d3           ; j = j+1
                cmp.w   d3,d0           ; j = n? (recalibré à zéro)
                bne     tantque         ;
                add.w   #1,d2           ; i = i+1
                move.w  #0,d3           ; j = 1 (recalibré à zéro)
                bra     tantque         ;
retourner:      movem.l (sp)+,a1-a2/d2-d5
                rts
```

8.0.6 La pile a la forme

```
registre d0 (on ne connaît pas ce contenu)
registre d1 (on ne connaît pas ce contenu)
```

```
registre d2 (on ne connaît pas ce contenu)
Espace local de 10 octets (on ne connaît pas ce contenu)
l'ancien contenu du registre a6 (pointe cadre précédent)
adresse de retour (c'est l'adresse de l'instruction après jsr)
100 (32 bits)
-1 (32 bits)
4 (16 bits)
```

8.0.7

```
; Entrée :   x, y, z entiers de 16 bits (valeur)
;            v vecteur de 11 entiers de 32 bits (adresse)
;            (passage par la pile)
; Sortie :   d0 contenant 0 ou la somme des éléments de v
;
; Var ;      s : 32 bits
;            i : 16 bits
;
x            EQU     16
y            EQU     14
z            EQU     12
v            EQU     8
s            EQU     -4
i            EQU     -6
F:           link    a6,#-6
             movem.l d0-d2/a0,-(sp)
             clr.l   (s,a6)        ; s = 0
             clr.w   (i,a6)        ; Pour i de 0 ...
pouri:
             cmp.w   #99,(i,a6)    ; à 99
             bgt     finPouri
             move.w  (y,a6),d0
             cmp.w   (x,a6),d0
             ble     sinon
             movea.l (v,a6),a0
             move.w  (i,a6),d0
             asl.w   #2,d0         ; i*4
             move.l  (a0,d0),d1    ; v[i]
             move.w  (z,a6),d2
             ext.l   d2
             sub.l   d2,d1         ; v[i] - z
             move.l  d1,(a0,d0)    ; v[i] = v[i] - z
             bra     continue
sinon:
             movea.l (v,a6),a0
             move.w  (i,a6),d0
             asl.w   #2,d0         ; i*4
```

```
            move.l  (a0,d0),d1   ; v[i]
            add.l   d1,(s,a6)    ; s = s + v[i]
continue:
                    addi.w       #1,(i,a6)
            bra     pouri
finpouri:
                    move.l       (s,a6),d0
            movem.l (sp)+,d0-d2/a0
            unlk    a6
            rts
```

b)

```
; Entrée :  x, y, z entiers de 16 bits (valeur)
;           v vecteur de 11 entiers de 32 bits (adresse)
;           (passage par la pile)
; Sortie :  d0 contenant 0 ou la somme des éléments de v
;
; Var
;           s : 32 bits
;           i : 16 bits
;
x           EQU     20           ; <-- Modifier pour tenir
y           EQU     16           ; <-- compte des adresses de 32 bits
z           EQU     12
v           EQU     8
s           EQU     -4
i           EQU     -6
F:          link    a6,#-6
            movem.l d0-d2/a0,-(sp)
            clr.l   (s,a6)       ; s = 0
            clr.w   (i,a6)       ; Pour i de 0 ...
pouri:
            cmp.w   #99,(i,a6)   ; à 99
            bgt     finPouri
            movea.l (y,a6),a0    ; <-- Modifier
            move.w  (a0),d0      ; <-- pour accéder
            move.l  (x,a6),a0    ; <-- à x et
            cmp.w   (a0),d0      ; <-- y
            ble     sinon
            movea.l (v,a6),a0
            move.w  (i,a6),d0
            asl.w   #2,d0        ; i*4
            move.l  (a0,d0),d1   ; v[i]
            move.w  (z,a6),d2
            ext.l   d2
            sub.l   d2,d1        ; v[i] - z
```

```
            move.l  d1,(a0,d0)   ; v[i] = v[i] - z
            bra     continue
sinon:
            movea.l (v,a6),a0
            move.w  (i,a6),d0
            asl.w   #2,d0        ; i*4
            move.l  (a0,d0),d1   ; v[i]
            add.l   d1,(s,a6)    ; s = s + v[i]
continue:
            addi.w  #1,(i,a6)
            bra     pouri
finpouri:
            move.l  (s,a6),d0
            movem.  (sp)+,d0-d2/a0
            unlk    a6
            rts
```

8.0.8

```
    resultatf  EQU 12
    xf         EQU 8
    f:    link    a6,#0
          movem.l d0,-(sp)
          move.l  (xf,a6),d0
          move.l  d0,(resultatf,a6)
          add.l   d0,(resultatf,a6)
          add.l   d0,(resultatf,a6)
          movem.l (sp)+,d0
          unlk    a6
          rtd     4
    resultatg  EQU 20
    xg         EQU 16
    yg         EQU 12
    zg         EQU 8
    g:    link    a6,#0
          movem.l d0,-(sp)
          move.l  (xg,a6),d0
          add.l   (yg,a6),d0
          add.l   (zg,a6),d0
          move.l  d0,(resultatg,a6)
          movem.l (sp)+,d0
          unlk    a6
          rtd     12
    resultath  EQU 10
    xh         EQU 8
    h:    link    a6,#0
```

```
        movem.l d0,-(sp)
        move.w  (xh,a6),d0
        ext.l   d0
        move.l  d0,(resultath,a6)
        add.l   d0,(resultath,a6)
        movem.l (sp)+,d0
        unlk    a6
        rtd     2
```

L'expression `f( g(h(3), h(4), h(5)) + g(h(6), h(7), h(8)) )`

```
        lea     (-4,sp),sp   ; espace pour résultat de f
        lea     (-4,sp),sp   ; espace pour résultat de g
        lea     (-4,sp),sp   ; espace pour résultat de h
        move.w  #3,-(sp)
        jsr     h            ; h(3)
        lea     (-4,sp),sp   ; espace pour résultat de h
        move.w  #4,-(sp)
        jsr     h            ; h(4)
        lea     (-4,sp),sp   ; espace pour résultat de h
        move.w  #5,-(sp)
        jsr     h            ; h(5)
        jsr     g            ; les trois arguments de g sont sur la pile
        lea     (-4,sp),sp   ; espace pour résultat de g
        lea     (-4,sp),sp   ; espace pour résultat de h
        move.w  #6,-(sp)
        jsr     h            ; h(6)
        lea     (-4,sp),sp   ; espace pour résultat de h
        move.w  #7,-(sp)
        jsr     h            ; h(7)
        lea     (-4,sp),sp   ; espace pour résultat de h
        move.w  #8,-(sp)
        jsr     h            ; h(8)
        jsr     g            ; les trois arguments de g sont déjà sur la pile
        move.l  (sp)+,d0     ; prend résultat du second appel de g
        add.l   d0,(sp)      ; additionne au premier résultat de g
        jsr     f            ; l'argument de f est déjà sur la pile
; ici le dessus de la pile contient le résultat de f
```

9.2.1 La détermination exacte du nombre de bits ne peut se faire sans explicitement le convertir en complément à deux. On s'attarde donc à une solution approximative.

Notons par n le nombre de bits nécessaire et suffisant pour représenter en complément à deux le nombre entier positif $|x|$. (On travail donc sur le nombre sans considérer son signe.) On a la relation $\lceil log_2(|x|+1) \rceil + 1 = n$.

Puisque nous voulons éviter de convertir x, on ne peut faire directement le calcul du logarithme. C'est pourquoi la solution proposé sera une approximation, mais celle-ci sera très près du résultat exact. Voici une solution qui donne soit n ou $n+1$. Le nombre de bits donné est donc potentiellement 1 de plus que nécessaire, mais ce nombre est toujours suffisant.

De plus, la solution complète en assembleur fonctionne que pour des nombres x avec au plus 12929 chiffres, à cause principalement de la simulation de la multiplication de nombres fractionnaires. (Car $12929 = \lfloor(2^{31}-1)/332192\rfloor$. Voir plus loin.) Notons par r ce résultat, c'est-à-dire que $r = n$ ou $r = n+1$.

Toute la méthode repose sur le fait que la valeur de $\log_2(10^{(l-1)})$, où l est le nombre de chiffres décimaux significatifs de $|x|$, est très près de $\log_2|x|$. En fait cette approximation est légèrement inférieure d'au plus 4 à $\log_2|x|$. On a $\log_2(10^{(l-1)}) = (l-1)\times\log_2 10$. Le calcul se réduit donc à une multiplication de $(l-1)$ à une constante fractionnaire. Cette multiplication sera approximée et simulée par un calcul avec des entiers.

La première étape consiste à balayer le nombre, qui est sous forme de caractères, et de compter le nombre de chiffres significatifs. Il faut donc sauter les espaces et le signe, ainsi que les chiffres '0' qui préfixerait le nombre. Ainsi pour " `-0000120003`" il y a 6 chiffres significatifs. Soit l ce nombre.

On fait ensuite le calcul de $\lceil(l-1)\times log_2 10\rceil$. En fait, on approxime en faisant le calcul de $\lceil(l-1)\times 3.32192\rceil$. (En effet, le logarithme en base 2 de 10 est plus précisément 3.3219280948873626 (c'est encore une approximation, mais plus précise), mais il ne sera pas possible de manipuler la forme entière de cette valeur sur 32 bits. C'est donc une approximation, d'où la limite de 12929 chiffres.)

Le calcul de $\lceil(l-1)\times 3.32192\rceil$ va être fait sous forme entière, car on veut l'entier de $\lceil(l-1)\times 3.32192\rceil = np$. Ce calcul se fait en multipliant $(l-1)$ par 332192 et en divisant par 100000. Le quotient est augmenté de 1 pour obtenir cette valeur np. (Il faut que le reste de la division soit non zéro pour augmenter de 1. Cela se produit toujours pour les valeurs que nous considérons.)

Ce np représente le nombre de bits nécessaire pour représenter en base deux un nombre quelconque de $(l-1)$ chiffres décimaux. On doit donc ajouter un bit de plus pour représenter ce même nombre en complément à deux.

On effectue un ajustement sur np fondé sur le premier chiffre significatif.

Voici la table pour cet ajustement.

Premier chiffre significatif	Valeur à ajouter à np pour obtenir r
1	1
2	2
3	2
4	3
5	3
6	3
7	3
8	4
9	4

Cette table a été conçue en considérant l'augmentation du nombre de bits lors d'une multiplication par une certaine valeur. Par exemple, si le nombre débute par le chiffre 2, ce nombre peut en fait être, dans la pire situation, $299\ldots999$. Ainsi, ce serait trois fois plus élevé que prévu par le calcul de np. En multipliant par trois, il faut deux bits de plus.

Dans le code assembleur, l'ajustement est combiné avec les deux augmentations de 1 précédentes : pour le calcul de $\lceil (l-1) \times 3.32192 \rceil$ et pour le passage de base deux à complément à deux.

Un point intéressant à retenir dans tout ce problème : il est possible d'utiliser des divisions entières pour approximer des multiplications avec des nombres fractionnaires. Voici enfin le code assembleur sous la forme d'une fonction.

```
; Entrée :       a0 pointe une chaîne de caractères représentant un
;                nombre décimal. Il peut y avoir des espaces avant
;                le nombre ou/et un signe '+' ou '-'.
;                (Il ne peut y avoir plus de 12929 chiffres significatifs.)
;
; Sortie :       d0.w contient le nombre de bits nécessaires et suffisants
;                pour représenter ce nombre en complément à deux.
;                Ce résultat est potentiellement un de plus que
;                nécessaire.
;
NbBits:         movem.l a0/d1,-(sp)
chercheDebut:   cmpi.b  #' ',(a0)+   ; passe les espaces
                beq     chercheDebut
                cmpi.b  #'-',(a0)    ; passe le signe '-'
                beq     passe_zeros
                cmpi.b  #'+',(a0)
```

```
                beq     passe_zeros  ; passe le signe '+'
                lea     -1(a0),a0    ; pointe premier chiffre
passe_zeros:    cmpi.b  #'0',(a0)+   ; passe les zéros non significatifs
                beq     passe_zeros
                move.b  (-1,a0),d1   ; le premier chiffre significatif
;
; Compte le nombre de chiffres significatifs. d0.l va contenir
; cette longueur moins 1, c'est-à-dire (l-1).
;
                move.l  #-1,d0
nombre:         add.l   #1,d0        ; un chiffre de plus
                cmpi.b  #'0',(a0)
                blo     fin_nombre
                cmpi.b  #'9',(a0)+
                bls     nombre
fin_nombre:     ; d0.l contient (l-1)
;
; Faire l'approximation de ⌈(l − 1) log2 10⌉ par le calcul
; de ⌈(l − 1) × 332192/100000⌉. L'addition de 1 va se faire plus
; loin.
;
                mulu.l  #332192,d0   ; résultat dans d0.l
                divu.l  #100000,d0   ; quotient dans d0.l
                                     ; on considère seulement d0.w
;
; appliquer l'ajustement basé sur le premier chiffre et l'addition de 2.
;
                cmp.b   #'1',d1
                beq     ajuste3
                cmp.b   #'3',d1
                bls     ajuste4
                cmp.b   #'7',d1
                bls     ajuste5
                add.w   #1,d0
ajuste5:        add.w   #1,d0
ajuste4:        add.w   #1,d0
ajuste3:        add.w   #3,d0        ; résultat final dans d0.w
                movem.l (sp)+,a0/d1
                rts
```

10.1.1 a) 0100 0111 1010 1101, C=0, V=0 b) 0001 1110 1011 0100, C=1, V=0
c) 1100 0111 1010 1101, C=0, V=0 d) 0001 1110 1011 0100, C=1, V=1
e) 0001 1110 1011 0101, C=1, V=0 f) 0100 0111 1010 1101, C=0, V=0
g) 0000 1000 1111 0101, C=1, V=0 h) 1111 1000 1111 0101, C=1, V=0
i) 1111 0101 1010 0000, C=1, V=0 j) 1111 0101 1010 0000, C=1, V=1

Remarque : dans le cas j, il y a débordement, même si le signe final est identique au signe de départ car le signe à changer durant l'opération de

décalage de 4 bits. En d'autres mots, la valeur arithmétique résultante est nettement incorrecte si on considère le décalage de 4 bits comme l'opération d'une multiplication par 2^4.

10.1.2

```
move.l  d0,d1        ; prendre une copie de y,x
move.w  #20,d2       ; pour le décalage qui suit
asr.l   d2,d1        ; enlever x en gardant le signe de y
add.l   d1,d0        ; additionner y à x
and.l   #$FFFFF,d0   ; mettre à zéro les bits 20 à 31
lsl.l   d2,d1        ; déplacer y dans les bits 20 à 31
or.l    d1,d0        ; coler y dans d0.l
```

10.2.1

```
          clr.l   d1          ; tous les bits à 0
prchBit:  lsr.l   #1,d0       ; regarder le bit suivant de d0
          bcc     bitZero     ; si bit à 0 ne pas transférer ds d1
          lsl.l   #1,d1       ; tasser les bits déjà transférés
          ori.l   #1,d1       ; ajouter un bit à 1 à l'extrême droite
bitZero:  bne     prchBit     ; continuer sauf si C=0 et d0.l=0
```

Une autre solution possible, probablement plus rapide.

```
           move.l  #-1,d1       ; tous les bits à 1
prchBit:   lsr.l   #1,d0        ; regarder le bit suivant de d0
           bcc     bitZero      ; si bit à 0 ne pas ...
           lsl.l   #1,d1        ; ajouter un bit à 0 dans d1
bitZero:   bne     prchBit      ; continuer sauf si C=0 et d0.l=0 ou d1.l=0
           not.l   d1           ; bits 0 <->bits 1
```

10.2.2

```
           clr.w   d1           ; 0 cm à gauche du point de départ
prchBit:   tst.l   d0           ; bits à 1 dans la chaîne direction?
           beq     fini         ; c'est fini si aucun bit à 1
           lsr.l   #1,d0        ; prochain bit pair
           bcc     pasAdroite   ; si bit sortant 0 =>va vers l'avant
           add.w   #-1,d1       ; sinon =>va vers la droite
pasAdroite:
           lsr.l   #1,d0        ; prochain bit impair
```

```
            bcc     prchBit      ; si bit sortant 0 =>va vers l'avant
            add.w   #1,d0        ; sinon =>va vers la gauche
            bra     prchBit
fini:                            ; d1.w contient le nombre de cms à gauche
```

10.3.1

```
; Entrée :    a0 pointe une matrice de 100 lignes par 800 colonnes
;             d'entiers de 16 bits. (matrice M)
;             a1 pointe une zone pour stocker une matrice de bits de
;             100 lignes et 100 colonnes. (matrice M')
; Sortie :    a1 pointe la matrice M'
;
Compacte:     movem.l a0-a1/d0-d2,-(sp)
              move.w  #100*100,d0  ; 10000 octet dans M'
prch8bits:    clr.b   d1           ; au départ les 8 bits sont à 0
              move.w  #7,d2        ; 8 bits à générer
unOctet:      lsl.b   #1,d1        ; tasser les bits déjà générés
              tst.w   (a0)+        ; est-ce un zéro dans M?
              beq     bitAzero     ; oui =>ne pas mettre le bit à 1
              ori.b   #1,d1        ; non =>mettre le bit à 1
bitAzero:     dbra    d2,unOctet   ;
              move.b  d1,(a1)+     ; placer les 8 bits dans M'
              dbra    d0,prch8bits
              movem.l (sp)+,a0-a1/d0-d2
              rts
```

10.3.2 Le curseur peut chevaucher deux octets de la même ligne si et seulement si la coordonnée y n'est pas multiple de 8. Dans le code en assembleur, on se préoccupe de ce fait en traitant deux octets adjacents. Un seul octet sera modifié dans le cas où il n'y a pas de chevauchement, mais cela est un cas particulier contenu dans le traitement général.

Le masque du curseur tient sur deux octets. C'est la valeur 255 décalée vers la gauche de 0 à 7 bits. Par exemple, voici le masque dans le cas où le reste de y divisé par 8 donne 2.

$$00000001111111100$$

Ce masque modifierait deux octets adjacents de la matrice. Les deux octets à charger de la matrice M, au départ, sont aux adresses

$$\text{adresse de } M + \text{y*175} + \text{quotient de y/8},$$
$$\text{adresse de } M + \text{y*175} + \text{quotient de y/8} + 1$$

Ces deux octets sont permutés dans un mot pour remettre les pixels dans le même sens que perçu sur l'écran. Le masque est appliqué à ce mot par un ou exclusif, ce qui inverse les bits, donc inverse les pixels. Les deux octets sont repermutés et replacés dans la matrice. Le même processus est répété sept autres fois sur les lignes précédentes.

```
; Entrée :   x: d0.w, y : d1.w, a0 pointe la matrice de bits M.
;           (La matrice a 175 colonnes (octet) de 8 pixels.)
;           ( 0 <= x <1393, 7 <= y <1000 )
;           ; Sortieum curseur à effet négatif est appliqué au carré
;           de 8 pixels par 8 pixels à la position (x,y) (coin
;           inférieur gauche) de la matrice M.
;
;
AfficheCurseur:
            movem.l d0-d6,-(sp)     ; sauvegarder les registres modifiés
;
; Calculer déplacement pour atteindre le mot contenant les huit pixels
; (x,y)...(x+7,y)
;
            move.w  d1,d2           ; d2 = y
            ext.l   d2              ; y sur 32 bits
            asl.l   #4,d2           ; d2 = y*16
            move.l  d2,d3           ;
            sub.l   d1,d3           ; d3 = y*15
            asl.l   #1,d2           ; d2 = y*32
            add.l   d2,d3           ; d3 = y*15 + y*32 = y*47
            asl.l   #2,d2           ; d2 = y*128
            add.l   d2,d3           ; d3 = y*128 + y*47 = y*175
            move.w  d0,d2           ; conserver d0.w = x
            ext.l   d0              ; x sur 32 bits
            asr.l   #3,d0           ; quotient de x/8
            add.l   d0,d3           ; d3 = y*175 + quotient x/8
;
; Construire le masque dans d5
;
            and.w   #7,d2           ; reste de x/8
            move.w  #255,d5
            lsl.w   d2,d5           ; d5 contient masque du curseur
            move.w  #7,d6           ; itérer 8 fois
prch_paire:
            move.w  (a0,d3.l),d4    ; un mot, 16 pixels adjacents
            rol.w   #8,d4           ; place les 2 octets dans le sens perçu
            eor.w   d5,d4           ; inverse les pixels
            rol.w   #8,d4           ; repermute les octets dans le sens originel
            move.w  d4,(a0,d3.l)    ; remet le mot dans la matrice
            sub.l   #175,d3         ; recule d'une ligne dans la matrice
            dbra    d6,prch_paire   ; deux autres octets du carré à inverser
```

```
movem.l (sp)+,d0-d6     ; rétablir les registres modifiés
rts                     ; retour à l'appeleur
```

10.3.3 On calcule $(512x + y)10$, où x est le numéro de ligne et y le numéro de colonne, qui est le numéro du premier bit du pixel à modifier. Soit n cette valeur. Pour accéder aux 10 bits de ce pixel il faut avoir l'adresse en octet, par rapport au début de l'écran, qui est la valeur $\lfloor n/8 \rfloor$. L'octet adjacent contient une partie des 10 bits, il faut donc charger le mot à cette adresse. Le reste de la division par 8 est le nombre de bits à sauter au début du mot. Il ne faut pas oublier d'inverser les octets pour voir les bits dans le bon sens.

```
mulu.w  #512,d0      ; résultat sur 32 bits
ext.l   d1           ; numéro de colonne sur 32 bits
add.l   d1,d0        ; d0.l contient indice du pixel
mulu.l  #10,d0       ; d0.l contient l'indice du premier bit, n
divul.l #8,d1:d0     ; d0.l contient l'indice de l'octet
move.w  (a0,d0.l),d2 ; charger les 2 octets contenant les 10 bits
addi.w  #8,d1        ; plus 8 à cause de l'inversion des octets
move.w  #$fc00,d4    ; préparer un masque pour mettre les 10 bits à 0
rol.w   d1,d4        ; aligner les 10 bits
and.w   d4,d2        ; mettre les 10 bits à 0
move.w  #102,d3      ;
rol.w   d1,d3        ; aligner la valeur 102 avec les 10 bits
or.w    d3,d2        ; mettre la valeur 102 dans le pixel
move.w  d2,(a0,d0.l) ; replacer le mot avec la nouvelle couleur
```

10.4.1

```
; Entrée :    a0 pointe un ensemble A.
;
; Sortie :    affichage de l'ensemble à l'écran entre accolades.
;
; Var
;             d1.w :   compte les bits dans un octet
;             d2.w :   compte les octets dans un ensemble
;             d3.b :   contient un octet de A
;             d4.b :   booléen, vrai si un élément est affiché
;             d5.w :   valeur de l'élément courant
;
Ens_afficher:
             movem.l a0/d0-d5,-(sp)
             move.w  #'{',d0           ; '{'
             jsr     putchar
             move.w  #31,d2            ; 32 octets
```

```
            move.w  #0,d5             ; débute à l'élément 0
            move.b  #0,d4             ; faux, pas d'élément affiché
afficher_octet:
            move.b  (a0)+,d3          ; 8 bits de A
            move.w  #7,d1             ; 8 bits à traiter
prch_element:
            lsr.b   #1,d3             ; place un bit de A dans le bit C
            bcc     pas_dans_A        ; Si 0 alors l'élément d5 n'est pas là
            tst.b   d4                ; un élément affiché?
            beq     le_premier        ; non, pas de virgule
            move.w  #',',d0           ; oui, affiche une virgule
            jsr     putchar           ;
le_premier:
            move.w  d5,d0             ; affiche la valeur de l'élément
            jsr     decout            ; présent dans A
            move.b  #1,d4             ; vrai, un élément a été affiché
pas_dans_A:
            add.w   #1,d5             ; valeur du prochain élément
            dbra    d1,prch_element   ; passer au prochain bit
            dbra    d2,afficher_octet ; passer au prochain octet
            move.w  #'}',d0
            jsr     putchar           ; '}'
            movem.l (sp)+,a0/d0-d5
            rts     ; retour à l'appeleur
```

10.4.2

```
; Entrée :    a0 pointe un ensemble A.
;
; Sortie :    d0.w contient le nombre d'éléments de A.
;
; Var
;            d1.w :   décompte du nombre de mots de 32 bits dans A
;            d2.w :   décompte du nombre de bits dans un mot de 32 bits
;            d3.l :   contient une partie de A
;
Ens_card:   movem.l a0/d1-d3,-(sp)   ; sauvegarder les registres modifiés
            move.w  #0,d0            ; aucun élément pour l'instant
            move.w  #7,d1            ; 8 long mots à traiter
prch_long_mot:
            move.l  (a0)+,d3         ; prend une partie de l'ensemble
            move.w  #31,d2           ; 32 bits à traiter dans cette partie
prch_bit:
            lsr.l   #1,d3            ; apporte un bit dans le bit C
            bcc     pas_d_element    ; si C=0, cet élément n'est pas dans A
            add.w   #1,d0            ; un élément de plus
pas_d_element:
```

```
            beq     passe_bits       ; plus rapide s'il y a peu d'éléments
            dbra    d2,prch_bit      ; décompte pour un long mot
passe_bits:
            dbra    d1,prch_long_mot ; décompte des longs mots de A
            movem.l (sp)+,a0/d1-d3   ; rétablir registre modifiés
            rts                      ; retourner à l'appeleur
```

10.4.3

```
; Entrée :    a0 pointe un ensemble A;
;             d0.w contient l'élément x à vérifier. ( 0 <= x <= 255 )
;
; Sortie :    Z = 1, si l'élément x est dans A;
;             Z = 0, si l'élément x n'est pas A.
;
Ens_eliminer:
            movem.w d0-d1,-(sp)  ; sauvegarder les registres modifiés
            move.w  d0,d1
            and.w   #7,d1        ; b = reste de x / 8
            lsr.w   #3,d0        ; i = quotient de x / 8
            move.b  (a0,d0.w),d0 ; octet i à vérifier
            not.b   d0           ; pour inverser le résultat suivant
            btst.b  d1,d0        ; si bit b à 0 =>Z = 1
                                 ; si bit b à 1 =>Z = 0
            movem.w (sp)+,d0-d1  ; rétablir les registres modifiés
            rts                  ; retourner à l'appeleur
```

11.1.1 Voici les nombres codés en virgule-fixe [16,16].

```
a)  1     = 0000 0000 0000 0001 0000 0000 0000 0000
b) -1     = 1111 1111 1111 1111 0000 0000 0000 0000
c)  0.5   = 0000 0000 0000 0000 1000 0000 0000 0000
d)  0.25  = 0000 0000 0000 0000 0100 0000 0000 0000
e)  0.75  = 0000 0000 0000 0000 1100 0000 0000 0000
f) -0.75  = 1111 1111 1111 1111 0100 0000 0000 0000
g) -5     = 1111 1111 1111 1011 0000 0000 0000 0000
h) -4.5   = 1111 1111 1111 1011 1000 0000 0000 0000
i)  0.125 = 0000 0000 0000 0000 0010 0000 0000 0000
j)  0.875 = 0000 0000 0000 0000 1110 0000 0000 0000
k) -9     = 1111 1111 1111 0111 0000 0000 0000 0000
l) -8.125 = 1111 1111 1111 0111 1110 0000 0000 0000
```

11.1.2 La codification de l'entier 294912 est 00000000000001001000000000000000 une fois complémenté à deux on a 11111111111101111000000000000000 ce qui est bien la codification de -4.5 en virgule-fixe [16,16].

11.1.3 L'instruction `add.l #$00008000,d0` additionne 0.5 à `d0`. On pourrait aussi écrire `add.l #32768,d0` ce qui fait exactement la même chose.

11.1.4 La plus grande valeur en binaire est

`0111 1111 1111 1111 . 1111 1111 1111 1111`,

ce qui vaut en décimal $2^{15} - 1 + (1 - 1/2^{16}) = 2^{15} - 1/2^{16} \approx 32767.99998$

La plus petite valeur en binaire est

`0000 0000 0000 0000 . 0000 0000 0000 0001`

ce qui vaut en décimal $1/2^{16} \approx 0.00001$.

11.1.5 En utilisant la valeur $2^7 + 1/2^{16}$ mise au carré la méthode `a` donne $2^{14} + 1/2^8$ (il y a perte de précision de $1/2^{32}$) mais la méthode `b` donne 2^{14} ce qui est encore moins précis.

11.1.6 Il s'agit de calculer $\lfloor \frac{a2^{16}}{b} \rfloor$.

```
move.w  d0,d2        ;
lsl.l   #8,d2        ; décaler a
lsl.l   #8,d2        ; de 16 bits, i.e. a*2^16
ext.l   d1           ; b sur 32 bits
divs.l  d1,d2        ; obtenir (a 2^16) div b
```

11.1.7

```
; Entrée : d0.l nombre y
; Sortie : d0.l l'arrondi de y selon la définition donnée.
; (note : la définition de l'arrondi n'est pas "standard")
;
Arrondi: tst.l   d0           ; négatif?
         bpl     positif      ; oui =>traite cas positif
         neg.l   d0           ; mettre positif
         add.l   #32768,d0    ; ajouter 0.5 (codé en [16,16])
         and.l   #$ffff0000,d0; enlever la partie fractionnaire
         neg.l   d0           ; remettre négatif
         bra     fin
positif: add.l   #32768,d0    ; ajouter 0.5
         andi.l  #$ffff0000,d0; enlever la partie fractionnaire
fin:     rts
```

11.1.8 a) 524287.99975. Il s'agit du nombre binaire

$$01111111111111111111.111111111111$$

qui vaut $2^{19} - 1 + (1 - 2^{-12}) \approx 524287.99975$. La plus petite valeur positive différente de $0 : 0.00024$. Il s'agit du nombre binaire 0.000000000001, qui vaut $2^{-12} \approx 0.0002441$.

b)

```
;
; Le premier nombre [20,12] est dans d0.l et le deuxième nombre
; [20,12] est dans d1.l.
; Le résultat de la multiplication est dans d1.l
;
      muls.l  d0,d2:d1     ; résultat sur 64 bits
      divs.l  #4096,d2:d1  ; décalage de 12 bits (plutôt coûteux)
```

On peut aussi le faire en six instructions pour éviter la division.

```
;
; Le premier nombre [20,12] est dans d0.l et le deuxième nombre
; [20,12] est dans d1.l.
; Le résultat de la multiplication est dans d1.l
;
      muls.l  d0,d2:d1     ; résultat sur 64 bits
      move.w  #12,d3       ; fait un décalage de 12 bits
      lsr.l   d3,d1        ; effectuer
      ror.l   d3,d2        ; le décalage de
      andi.l  #$fff00000,d2; 12 bits de la chaîne de 64 bits
      or.l    d2,d1        ; et obtenir le résultat sur 32 bits
```

Il y a potentiellement un débordement, car le passage de 64 bits à 32 bits peut éliminer des bits significatifs de la partie haute. Il y a aussi potentiellement une perte de précision, car le décalage de 12 bits pour le passage de 64 bits à 32 bits peut faire sortir des bits significatifs de la partie basse.

11.2.1 Le calcul à effectuer peut se faire à l'aide de cinq termes seulement si on se limite à un argument x inférieur à $\pi/2 \approx 1.570796$. Une telle limitation de l'argument est justifiée car pour toute autre valeur, le résultat s'obtient par l'utilisation d'identités trigonométriques.

Ainsi, le terme $x^{11}/11!$ pour la plus grande valeur possible $x = \pi/2$ donne $\pi^{11}/(11!2^{11}) \approx 1.7572 \times 10^{-9}$, ce qui est plus petit que $1/2^{16}$ qui est la précision en virgule-fixe avec 16 bits dans la partie fractionnaire. D'autre

part, le terme $x^9/9!$ donne approximativement 1.604×10^{-4}, ce qui n'est pas inférieure à $1/2^{16} \approx 1.525 \times 10^{-5}$. Il est donc nécessaire de le comptabiliser.

Nous utilisons donc l'approximation suivante.

$$\sin x \approx x - x^3/3! + x^5/5! - x^7/7! + x^9/9!$$

Si on veut limiter les divisions on peut faire le calcul équivalent.

$$\sin x \approx x + (-9!/3!x^3 + 9!/5!x^5 - 9!/7!x^7 + x^9)/9!$$

Les constantes $9!/3! = 60480$, $9!/5! = 3024$ et $9!/7! = 72$ sont directement insérées dans le programme. Les constantes 60480 et 3024 posent quelques problèmes, car il faut garder les résultats des multiplications sur 64 bits. Les additions seront de 64 bits entre ces termes.

Dernier point à se préoccuper. Est-ce que tous les calculs peuvent se faire sans débordement? Un débordement pourrait surgir lors du calcul de x^9 ou de la somme des puissances. Toutefois, la valeur de $(\pi/2)^9$ est inférieur à 60. De plus, la somme des puissances avant la division par 9! ne dépasse pas -207130, ce qui tient sur 32 bits.

Le passage du terme x^i au terme $x^{(i+2)}$ peut se faire en multipliant par x^2. Il est donc utile de conserver la valeur de x^2. La figure A.1 présente le code assembleur.

Ce code source peut être compilé pour produire un code objet pouvant servir d'élément à une librairie de fonctions. La directive `xdef SinVf` définit le symbole `SinVf` comme identificateur publique, i.e. qu'il peut être référé par une directive `xref` dans un autre code source indépendant.

```
; Entrée :    d0.l contient un nombre x en virgule-fixe [16,16].
;            (On suppose 0 <= x <= pi/2)
; Sortie :    d0.l contient sin x.
;
; Var
;            d1.l : x^2 en [16,16]
;            d2.l : pour calculer x^2, x^3, x^5, x^7, x^9 sur 64 bits
;            d3.l : x^3, x^5, x^7, x^9
;            d5.l,d4.l : le numérateur (-9!/3!x^3 + 9!/5!x^5 - 9!/7!x^7 + x^9) en [48,16]
;            d6.l,d7.l : pour calculer les termes du numérateur
;
             xdef    SinVf
SinVf:       movem.l d1-d7,-(sp)     ; sauvegarde registres modifiés
             move.l  d0,d1           ; copie x
             muls.l  d1,d2:d1        ; x^2 en [32,32]
             move.w  d2,d1           ; d2:d1 est décalé à droite de 16 bits
             swap    d1              ; résultat de x^2 en [16,16]
             move.l  d1,d3           ; copie x^2
             muls.l  d0,d2:d3        ; x^3 en [32,32]
             move.w  d2,d3           ; x^3
             swap    d3              ; x^3 en [16,16] (pas de débordement)
             move.l  d3,d4
             muls.l  #-60480,d5:d4   ; -9!/3!*x^3 en [48,16], le numérateur
             muls.l  d1,d2:d3        ; x^5 en [32,32]
             move.w  d2,d3
             swap    d3              ; x^5 en [16,16] (pas de débordement)
             move.l  d3,d6
             muls.l  #3024,d7:d6     ; 9!/5!*x^5 en [48,16]
             add.l   d6,d4           ; addition sur 64 bits
             addx.l  d7,d5           ; du terme 9!/5!*x^5 au numérateur
             muls.l  d1,d2:d3        ; x^7 sur [32,32]
             move.w  d2,d3
             swap    d3              ; x^7 en [16,16] (pas de débordement)
             move.l  d3,d6
             muls.l  #-72,d7:d6      ; -9!/7!*x^7 en [48,16]
             add.l   d6,d4           ; addition sur 64 bits
             addx.l  d7,d5           ; du terme -9!/7!*x^7 au numérateur
             muls.l  d1,d2:d3        ; x^9 en [32,32]
             move.w  d2,d3
             swap    d3
             add.l   d3,d4           ; addition sur 64 bits
             addx.l  #0,d5           ; du terme x^9 au numérateur
             divs.l  #362880,d4:d5   ; numérateur/9!
             add.l   d5,d0           ; numérateur/9! + x, résultat final dans d0
             movem.l (sp)+,d1-d7
             rts
             end
```

FIG. A.1 – **Code assembleur pour** sin.

11.3.1

```
Procedure AfficherMultiple9sur5(n : cardinal)
      { n >= 0 }
Var i, ipq, ipr : cardinal;
Debut
ipq = 1; ipr = 4; { 9/5 donne un quotient de 1 reste 4 }
Pour i de 1 a n Faire
  Debut
  Afficher(ipq);
  ipr = ipr + 4; ipq = ipq + 1;
  Si ipr >= 5 Alors ipq = ipq + 1; ipr = ipr - 5;
  Fin
Fin AfficherMultiple9sur5
```

En assembleur :

```
; Entrée :  d0.w : n >= 0
;
; Sortie :  affiche les parties entières des multiples de 9/5,
;           de 9/5 jusqu'à n*9/5.
;
; Var
;           ipq : d0.w
;           i : d1.w
;           ipr : d2.w
;           n : d3.w
;
AfficherMultiple9sur5:
            movem.w d0-d3,-(sp)
            move.w  d0,d3         ; copie de n
            move.w  #1,d0         ; ipq = 1
            move.w  #4,d2         ; ipr = 4
            move.w  #1,d1         ; i = 1
pour:       cmp.w   d1,d3         ; i <= n?
            blo     fin_pour
            jsr     decout        ; Afficher(ipq)
            add.w   #4,d2         ; ipr = ipr + 4
            add.w   #1,d0         ; ipq = ipq + 1
            cmp.w   #5,d2         ; ipr >= 5?
            blt     prch_i        ;
            add.w   #1,d0         ; ipq = ipq + 1
            sub.w   #5,d2         ; ipr = ipr - 5
prch_i:     add.w   #1,d1         ; i = i + 1
            bra     pour
fin_pour:
            movem.w (sp)+,d0-d3
            rts
```

Voici un programme qui utilise la procédure AfficherMultiple9sur5.

```
            xref   strout,decout,stop
            move.w #20,d0
            jsr    AfficherMultiple9sur5
            jsr    stop
;
; Placer la sous-routine ici.
;
            end
```

12.1.1

S Exposant Mantisse

a) 1 10001000 1010 0000 0000 0000 0000 000
sans perte de précision.

b) 0 01111010 0100 0000 0000 0000 0000 000
sans perte de précision.

c) 1 01000100 0011 0101 1101 1100 0000 001
avec une perte de précision de moins de 2E-25.

d) 1 01110100 1000 1001 1011 1101 1000 001
avec une perte de précision.

e) 0 01001100 0001 1001 0001 1100 0111 000
avec une perte de précision de moins de 2.7E-23.

12.1.2

a) -3.0
b) 1.0
c) -3.5
d) 0.8125
e) -0.0
f) 0.0
g) $1.5/2^{126}$ approximativement 1.7632415262334313E-38
h) 0.0625

12.1.3 Le nombre 4.1 ne peut être stocké sans perte de précision quelque soit la précision du IEEE. La valeur 1/10 comme toutes les valeurs qui ne peuvent être représentées sous la forme $n/2^m$ ne peut être représenté exactement en binaire quelque soit le nombre de bits dans la partie fractionnaire.

12.1.4 La plus grande valeur est

$$2^{127}(2-1/2^{23}) = 2^{128} - 2^{104} = 340282346638528859811704183484516925440$$

La plus petite valeur est $2^{-126} \approx 1.1754943508222875e-38$. La plus grande valeur est un nombre entier mais la plus petite ne l'est pas.

12.1.5 Cette valeur est 2^{24} car elle est représentable par l'exposant 24 et la mantisse 1.0 mais la valeur $2^{24}+1$ n'est plus représentable exactement.

12.1.6 Non, pas dans tous les cas, car il existe des nombres qui ne sont pas au départ représentable exactement en IEEE-754, et cela quelque soit le nombre de bits. Par exemple, 1/10 n'est pas représentable exactement en IEEE-754, et cela quelque soit le nombre de bits de la mantisse.

12.1.7 Au départ, IEEE-754 est un standard international. S'éloigner de ce standard ne pourrait que provoquer des problèmes de compatibilités. De plus, la représentation en point-fixe ne peut facilement représenter à la fois des très grands nombres et des très petits nombres sans utiliser plus de bits que le IEEE-754. Cela est dû à l'utilisation d'un exposant en IEEE qui n'existe pas en point-fixe.

12.1.8 La plus grande valeur est $(2-1/2^{52}) * 2^{1024} \simeq 3.5953862697246462E308$. La précision est de $1/2^{52}$, soit approximativement $2.220446049250313e-16$, c'est-à-dire 15 chiffres décimaux significatifs.

12.2.1 Voici une sous-routine pour multiplier par quatre un nombre IEEE-754, 32 bits. Cette sous-routine additionne 2 à l'exposant si et seulement si la valeur n'est pas zéro.

```
; Entrée :   d0.l : un nombre fini x IEEE-754 32 bits.
; Sortie :   d0.l : 4*x en IEEE-754 32 bits.
;
MultVF4:    and.l   #$efffffff,d0; x = 0?
            beq     finMultVF4   ; c'est -0 ou 0, le résultat ne change pas
            add.l   #$01000000,d0; additionne 2 à l'exposant de x
finMultVF4:
            rts                  ; retour à l'appeleur
```

12.2.2 La conversion d'un nombre entier de 32 bits vers la codification IEEE-754 de 32 bits peut produire une perte de précision mais pas un débordement. Il peut se produire une perte de précision car la mantisse n'a que 24 bits réels bien que le nombre entier en a 31. Premièrement, si le nombre est zéro, on retourne immédiatement zéro. Si le nombre est négatif il est mis positif et un bit de signe négatif, i.e. 1, est placé dans le résultat au bit 31. Le nombre entier est décalé dans la direction pour obtenir un bit à 1 en position 23. Le nombre et la direction de décalage donne l'exposant du résultat. Un décalage vers la droite donne un exposant positif, un décalage vers la gauche un exposant négatif. Ce bit à 1 en position 23 est éliminé et la mantisse et l'exposant biaisé 127 sont introduits dans le résultat dans les bits 0 à 30. Si un décalage vers la droite a été effectué et que des bits à 1 sont sortis, alors le bit C est mis à 1 car une perte de précision s'est produite. Voici le code complet.

```
; Entrée :   d0.l nombre entier x de 32 bits codé en complément à deux.
; Sortie :   d0.l nombre x codé en IEEE-754 sur 32 bits
;            C = 1 si une perte de précision s'est produit
;            C = 0 si x est représenté exactement en IEEE-754
;
EntAIEEE:   movem.l d1-d2,-(sp)
            move.l  d0,d1           ; on travaille sur x dans d1.l
            beq     finEntAIEEE     ; si x = 0, rien à faire
            bpl     OkPositif
            neg.l   d1              ; rendre x positif
OkPositif:  and.l   #$80000000,d0   ; garde seulement le signe de x
            or.l    #$4f800000,d0   ; place exposant 127+32
cherche1:   sub.l   #$00800000,d0   ; soustrait 1 de l'exposant
            lsl.l   #1,d1           ; décale x vers la gauche jusqu'à
            bcc     cherche1        ; sortir le bit extrême gauche à 1
            move.l  d1,d2           ; garde une copie pour vérifier précision
            lsr.l   #8,d1           ; décale x pour l'insérer dans la
            lsr.l   #1,d1           ; mantisse de d1
            or.l    d1,d0           ; insère mantisse dans d1
            and.w   #$1ff,d2        ; perte de précision? (bit C mis à 0)
            beq     finEntAIEEE     ; si bits 0 à 8 = 0 =>pas de perte
            ori.w   #1,ccr          ; positionne le bit C à 1 si perte
finEntAIEEE:
            movem.l (sp)+,d1-d2
            rts
```

12.2.3 Si l'un des deux nombres est zéro, retourner l'autre nombre. Si l'un des deux nombres a l'exposant 128, retourner ce nombre. Dans les autres cas, pour additionner deux nombres IEEE il faut normaliser les mantisses sous l'exposant du plus grand nombre en valeur absolue. Soit k la valeur absolue de la différence entre les exposants. Si $k > 23$ le résultat de l'addition est le

plus grand nombre car le plus petit nombre ne peut modifier le grand nombre à cause de la mantisse réelle de 24 bits. Pour normaliser la mantisse du plus petit nombre, il faut ajouter le bit de valeur 1 à cette mantisse et la décaler vers la droite de k bits. Puis, le bit de valeur 1 manquant du plus grand nombre est ajouté, les exposants et les signes sont éliminés des deux chaînes de bits. Si les signes des deux nombres sont opposés, il faut soustraire le plus petit du plus grand pour obtenir une mantisse positive. Le signe du résultat est le signe du plus grand nombre. Les deux mantisses sont additionnées sur 32 bits comme des nombres entiers. Ce résultat est finalement renormalisé pour avoir un bit à 1 en position 23 et l'exposant est ajusté en fonction de ce décalage.

13.1.1 On applique le principe simple d'appel de sous-routines.

```
; Entrée :  d0.l contient n
; Sortie :  d0.l contient n!
;
factoriel: movem.l d1,-(sp)     ; sauvegarder d1
           cmp.l   #1,d0        ; si n <= 1
           bhi     calcule
           move.l  #1,d0        ; retourner(1)
           bra     fin
calcule:   move.l  d0,d1        ; garder n
           sub.l   #1,d0        ; n-1
           jsr     factoriel    ; appel récursif factoriel(n-1)
           mulu.l  d1,d0        ; n * factoriel(n-1) (résultat dans d0.l)
fin:       movem.l (sp)+,d1     ; rétablir d1
           rts     ; retour à l'appeleur
```

13.1.2

```
; Entrée :  a est dans d0.w; b est dans d1.w
; Sortie :  d0.w
;
Pgcd:
           tst.w   d0           ; Si a = 0 Alors
           bne     anonz        ; Sinon ...
           tst.w   d1           ; Si b = 0 Alors
           bne     azbnonz      ; Sinon ...
           move.w  #1,d0        ; retourner(1)
           rts
azbnonz:   move.w  d1,d0
           rts                  ; retourner(b)
anonz:     tst.w   d1
           bne     casgeneral
```

```
            rts                 ; retourner(a)
casgeneral:
            cmp.w   d0,d1       ; a >b?
            bhs     sinon
            swap    d0          ; échanger a avec la partie haute
            move.w  d0,-(sp)    ; sauvegarder la partie haute de d0
            swap    d0          ; replacer a dans la partie basse de d0
            and.l   #$0000ffff,d0; étendre a (cardinal) sur 32 bits
            divu.w  d1,d0
            swap    d0          ; a mod b (le reste de la division)
            jsr     Pgcd        ; appel récursif Pgcd(a mod b, b)
            swap    d0
            move.w  (sp)+,d0    ; reprendre la partie haute de d0
            swap    d0          ; replacer la partie haute et a dans d0.w
            rts
sinon:                          ; ici a <= b
            move.l  d1,-(sp)    ; garder b et la partie haute de d1
            and.l   #$0000ffff,d1; étendre un cardinal sur 32 bits
            divu.w  d0,d1
            swap    d1          ; b mod a (le reste de la division)
            jsr     Pgcd        ; appel récursif Pgcd(a, b mod a)
            move.l  (sp)+,d1    ; rétablir b et la partie haute de d1
            rts
```

13.1.3

```
; Entrée :  a0 pointe une zone de trois champs.
; Sortie :  d0 contient la valeur de l'évaluation de la zone.
;
Evaluer:    movem.l a0/d1,-(sp)
            move.l  1(a0),d0    ; prendre le premier opérande
            cmp.b   #0,(a0)     ; opérateur?
            beq     fin
            movea.l 5(a0),a0    ; prendre l'adresse de l'autre zone
            jsr     Evaluer     ; évaluer récursivement le second opérande
            movea.l 4(sp),a0    ; reprendre le a0
            move.l  d0,d1       ; garder le second opérande
            move.l  1(a0),d0    ; reprendre le premier opérande
            cmpi.b  #1,1(a0)    ; addition?
            beq     addition
            cmpi.b  #2,1(a0)    ; multiplication?
            beq     mult
            sub.l   d1,d0       ; sinon soustraction.
            bra     fin
addition:   add.l   d1,d0
            bra     fin
mult:       muls.l  d1,d0
```

```
fin:       movem.l (sp)+,a0/d1
           rts
```

13.1.4

```
; Entrée : a0 pointe une expression booléenne sous la forme d'une chaîne
;         de caractères.
;
; Sortie : bit Z = 1 si l'expression évalue à 1
;         bit Z = 0 si l'expression évalue à 0
;         a0 pointe le premier caractère non-espace après l'expression
;         booléenne.
;
ExprBool: movem.w d0-d1,-(sp)
          jsr     TermeBool
          move.w  ccr,d0       ; garder le résultat de TermeBool
tantque0u:
          cmp.b   #'+',(a0)
          bne     finExprBool
          lea     (1,a0),a0    ; passe l'opérateur +
          jsr     TermeBool
          move.w  ccr,d1       ; prendre le résultat de TermeBool
          or.w    d1,d0        ; opérateur ou-logique
          bra     tantque0u
finExprBool:
          move.w  d0,ccr       ; retourner le résultat
          movem.w (sp)+,d0-d1
          rts



; Entrée :  a0 pointe un terme booléen sous la forme d'une chaîne
;          de caractères.
;
; Sortie :  bit Z = 1 si le terme évalue à 1
;          bit Z = 0 si le terme évalue à 0
;          a0 pointe le premier caractère non-espace après le terme.
;
TermeBool: movem.w d0-d1,-(sp)
           jsr     FacteurBool
           move.w  ccr,d0       ; garder ce premier facteur
tantqueEt:
           cmp.b   #'*',(a0)
           bne     finTermeBool
           lea     (1,a0),a0    ; passe l'opérateur *
           jsr     FacteurBool
           move.w  ccr,d1
```

```
            and.w   d1,d0        ; opération ET entre bits Z
            bra     tantqueEt
finTermeBool:
            move.w  d0,ccr       ; retourner le résultat
            movem.w (sp)+,d0-d1
            rts



; Entrée :    a0 pointe un facteur booléen sous la forme d'une chaîne
;             de caractères.
;
; Sortie :    bit Z = 1 si le facteur évalue à 1
;             bit Z = 0 si le facteur évalue à 0
;             a0 pointe le premier caractère non-espace après le facteur.
;
FacteurBool: movem.w d0,-(sp)
             cmp.b   #' ',(a0)+   ; passe les espaces
             beq     FacteurBool
             cmp.b   #'(',(-1,a0) ; si une '(' il y a une sous-expression
             bne     valeurBin
             jsr     ExprBool     ; évaluer la sous-expression
             move.w  ccr,d0       ; garder le résultat pour la fin
             lea     (1,a0),a0    ; passe la parenthèse ')'
             bra     passeEspaces
valeurBin:   cmp.b   #'1',(-1,a0) ; bit Z = 1 si '1', Z = 0 si non '1'('0')
             move.w  ccr,d0       ; garder le résultat pour la fin
passeEspaces:cmp.b   #' ',(a0)+
             beq     passeEspaces
             lea     (-1,a0),a0   ; revient sur le caractère non-espace
             move.w  d0,ccr       ; retourne le résultat par le bit Z
finFacteur:  movem.w (sp)+,d0
             rts
```