Arithmétique à virgule flottante Chapitre 14

Bien que les entiers constituent une représentation exacte des valeurs numériques, ils souffrent de deux inconvénients majeurs : l'impossibilité de représenter des valeurs fractionnaires et leur plage limitée. L'arithmétique à virgule flottante résout ces deux problèmes aux dépenses de la précision, et, sur certains processerus, de la vitesse. Beaucoup de programmeurs sont conscients de la perte en performances que l'arithmétique à virgule flottante comporte et cependant ils ne se rendent pas toujours compte de l'autre inconvénient, le manque de précision.

Pour beaucoup d'applications, les avantages de l'arithmetique flottante dépassent ces désavantages. Cependant, pour utiliser correctement ce genre d'arithmétique dans tout programme, vous devez en apprendre le fonctionnement. Intel, en comprenant l'importance de l'arithmétique flottante dans les programmes moderne, a fourni du support déjà à partir des premiers modèles 8086 : le FPU 80x87 (unité d'arithmétique flottante ou coproccesseur mathématique). Néanmoins, sur les processeurs antérieurs au 80486 (ou sur le 80486sx), le processeur d'arithmétique flottante est un dispositif optionnel ; s'il n'est pas présent, il faut le simuler de façon logicielle.

Ce chapitre contient quatre sections principales. La première traite l'arithmétique en virgule flottante d'un point de vue mathématique. La seconde section discute des formats binaires en virgule flottante communement utilisés dans les processeurs Intel. La troisième a pour objet la virgule flottante du point de vue logiciel, ainsi que des routines mathématiques de la bibliothèque standard UCR. Et, finalement, la quatrième discute des puces du FPT 80x87.

14.0 Vue d'ensemble du chapitre

Ce chapitre contient quatre importantes sections : une description des formats à virgule flottante et des opérations (deux sections), une discussion sur la virgule flottante sopportée par la bibliothèque UCR standard et une discussion sur le CPU 80x87 (unité à virgule flottante). Les sections ayant le préfixe "•" sont essentielles, alors que celles ayant un "o" concernent des sujets plus avancés que vous pouvez mettre de côté pour le moment.

- La mathématique de l'arithmétique à virgule flottante.
- Formats de virgule flottante IEEE
- Les routines de la blibliothèque standard UCR
- Les coprocesseurs à virgule flottante 80x87
- Instructions de mouvement de données FPU

o Conversions

- Instructions arithmétiques
- Instructions de comparaison

o Instructions constantes
o Instructions transcendantes
o Instructions diverses
o Opérations d'entiers
o Fonctions trigonométriques supplémentaires

14.1 La mathématique de l'arithmétique à virgule flottante

Un gros problème avec l'arithmétique flottante est qu'elle ne suit pas les mêmes règles que l'algèbre. Néanmoins les programmeurs appliquent ces règles, même lorsqu'ils utilisent l'arithmétique à virgule flottante. Ce qui est une source de bogues dans beaucoup de programmes. L'un des buts primaires de cette section est de décrire les limites de cette arithmétique, ainsi vous pourrez la comprendre correctement.

Les règles normales de l'algèbre s'appliquent seulement à une arithmétique de *précision infinie*. Considérez l'expression x:=x+1, où x est un entier. Sur un ordinateur moderne cette expression respecte les règles normales *jusqu'à ce qu'un dépassement de capacité ne se produit*. Donc, en informatique, l'équation est valide seulement poru certaines valeurs de x (minint <= x < maxint). Beaucoup de programmeurs n'ont pas de problèmes avec cela parce que ils sont bien conscients du fait que les entiers d'un programme ne suivent pas les règles standard de l'algèbre (par exemple, $5/2 \neq 2.5$).

Dans un ordinateur, les entiers ne suivent pas les mêmes règles que l'algèbre parce que celui-ci les représente avec un nombre fini de bits. Par exemple, vous ne pouvez pas représenter des valeurs moindres que minint ou supérieures à maxint, dans le domaine qu'on vient de considérer ci-dessus. Les valeurs à virgule flottante ont le même problème et c'est même pire. D'après tout, les entiers sont un sous-ensemble des réels. Par conséquent, les valeurs à virgule flottante doivent représenter aussi l'ensemble, infini, des nombres entiers. Cependant, il y a un nombre infini de valeurs entre deux entiers, donc, ce problème est infiniment pire que l'autre. Donc, à part de limiter son ensemble de valeurs représentables entre un minimum et un maximum, il faut aussi limiter toute plage de valeur entre un nombre et l'autre.

Pour représenter les nombres réels, la plupart des formats à virgule flottante emploient la notation scientifique et utilisent un certain nombre de bits pour représenter une *mantisse* et un encore plus petit nombre de bits pour représenter un *exposant.* Le résultat final est que les nombres à virgule flottante ne peuvent représenter des nombres qu'avec un nombre spécifique de *chiffres significatifs.* Ce qui a un grand impact sur les opérations d'arithmétique à virgule flottante. Pour saisir facilement cet impact, nous utiliserons un format simplifié dans nos exemples. Ce format simplifié utilisera une mantisse de trois chiffres significatifs et un exposant décimal de deux chiffres. Les deux sont des valeurs signées (voir Figure 14.1).

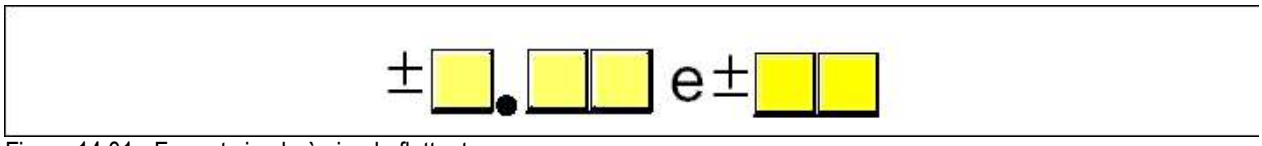

Figure 14.01 - Format simple à virgule flottante

En additionnant ou en soustrayant deux nombres en notations scientifique, il faut ajuster les deux valeurs pour que leurs exposants soient les mêmes. Par exemple, en additionnant 1.23e1 et 4.56e0, il faut ajuster les valeurs de sorte qu'elles aient les mêmes exposants. Une façon de le faire est de convertir 4.56e0 en 0.456e0 et puis additionner, ce qui produit 1.686e1. Malheureusement, le résultat ne rentre pas dans trois chiffres décimaux significatifs, il faut donc: soit *arrondir*, soit *tronquer* le résultat à trois chiffres décimaux significatifs. Arrondir produit généralement le résultat le plus précis, donc, arrondissons pour obtenir 1.69e1. Vous pouvez néanmoins remarquer que le manque de *précision* (le nombre de chiffres ou de bits qu'on maintient dans un calcul), affecte l'exactitude du résultat.

Dans l'exemple précédent, on était capable d'arrondir le résultat parce qu'on maintenait *quatre* chiffres décimaux significatifs *pendant* le calcul. Mais, si notre calcul à virgule flottante est limité à trois chiffres décimaux *pendant* le calcul, il faudrait tronquer le dernier chiffre du nombre le plus petit, en obtenant 1.68e1, qui est encore moins correct. Des chiffres supplémentaires disponbiles pendant un calcul sont connus comme *chiffres de garde* (ou *bits de garde* dans le cas du format binaire). Ils améliorent grandement l'exactitude pendant une longue chaîne de calculs.

La perte d'exactitude pendant un seul calcul n'est pas normalement très importante, sauf quand la précision est un facteur déterminant. Cependant, si le résultat est une valeur qui est le résultat d'une séquences d'opérations en virgule flottante, l'erreur s'*accumule* et affecte de plus en plus l'exactitude du résultat. Par exemple, supposons qu'on additionne 1.23e3 avec 1.00e0. Ajuster les nombres de sorte qu'ils aient les mêmes exposants avant l'addition produit 1.23e3 + 0.001e3. La somme de ces valeurs, même après arrondissement correspond à 1.23e3. Ce qui peut sembler parfaitement raisonnable, d'après tout on ne peut maintenir que trois chiffres significatifs, donc additionner une petite valeur ne devrait affecter pas du tout le résultat. Cependant, supposons maintenant qu'on doit additionner 1.00e0 et 1.23e3 *dix fois*. La première fois, le résultat est 1.23e3. A partir de la deuxième fois, il se trouve qu'on obtient le même résultat toujours, jusqu'à la dixième fois. Et cependant, si l'on additionnait 1.00e0 avec lui même dix fois et ensuite on additionnait le résultat avec 1.23e3, on aurait obtenu un résultat différent, 1.24e3. Voici la chose la plus importante à retenir à propos de la précision arithmétique limitée:

L'ordre de l'évaluation peut affecter l'exactitude du résultat.

On peut obtenir de meilleurs résultats si les grandeurs relatives (autrement dit, les exposants) sont proches l'un de l'autre. Si on est en train d'effectuer un calcul en chaîne avec des additions et des soustractions, il nous faudrait grouper les valeurs de façon appropriée.

Un autre problème avec l'addition et la soustraction à virgule flottante est la *fausse précision*. Considérons 1.23e0 - 1.22e0. Ceci produit 0.01e0. Bien que cela soit mathématiquement équivalent à 1.00e-2, cette dernière version suggère que les deux derniers chiffres sont exactement zéro. Malheureusement, on a obtenu seulement un seul chiffre significatif

correct cette fois. Certains FPU ou certains packages logiciels en virgule flottante peuvent insérer des chiffres (ou bits) aléatoires dans les positions les moins significatives. Ce qui amène à une seconde importante règle de l'arithmétique à précision limitée:

> *Quand on soustrait deux nombres ayant le même signe ou quand on additionne deux nombres de signe différent, l'exactitude du résultat peut être moindre que la précision disponible du format à virgule flottante.*

La multiplication et la division n'ont pas les mêmes problèmes que l'addition et la soustraction, car on n'a pas à ajuster les exposants avant l'opération. Tout ce qu'on a à faire est d'additionner les exposants et de multiplier les mantisses (ou soustraire les exposants et diviser les mantisses, dans le cas de la division). Par ce fait même, la multiplication et la division produisent des résultats meilleurs. Cependant, ces opérations tendent à multiplier toute erreur déjà existante dans la valeur. Par exemple, si l'on multiplie 1.23e0 par 2, alors qu'on devrait multiplier 1.24e0 par 2, le résultat est encore moins exact. Ce qui porte à une troisième importante règle de l'arithmétique à précision limitée :

> *Quand on effectue une chaîne de calculs concernant l'addition, la soustraction, la multiplication ou la division, essayons d'effectuer les opérations de multiplication et de division en premier.*

Souvent, en appliquant des transformations algébriques ordinaires on peut arranger une opération de sorte à faire apparaître les opérations de multiplication et de division en premier. Par exemple, supposez que vous voulez calculer x*(y+z). Normalement vous additionneriez y et z et vous multiplieriez leur somme par x. Cependant vous obtiendriez un résultat plus exact en transformant x*(y+z) pour obtenir x*y + x*z et calculer le résultat en effectuant les multiplications en premier.

La multiplication et la division ne sont pas exemptes de problèmes. Quand vous multipliez deux valeurs très grandes ou très petites, vous risquez toujours le dépassement de capacité, pas excès ou par défaut. La même situation peut survenir quand vous divisez entre un nombre grand et un nombre petit ou vice-versa. Ce qui amène à une quatrième règle que vous devriez tenir en compte quand vous appliquez ces opérations:

> *En multipliant ou en divisant des nombres, essayez d'arranger les multiplications de sorte à toujours faire multiplier les petits nombres et les grands nombres ensemble. De même essayez de arranger les divisions de sorte à grouper les valeurs de grandeur similaire.*

Effectuer des calculs avec les nombres réels est très dangereux. Étant donné la marge d'erreur présente en tout calcul (en incluant les conversions entre une chaîne d'entrée et une valeur à virgule flottante), vous ne devriez *jamais* comparer deux valeurs à virgule flottante pour voir si elles sont égales. Dans un format binaire à virgule flottante, des calculs différents produisant le même, mathématique, résultat peuvent différerer dans leurs bits moins significatifs. Par exemple, additionner 1.31e0+1.69e0 devrait produire 3.00e0. De même, additionner 2.50e0+1.50e0 devrait également produire 3.00e0. Cependant, quand vous comparez (1.31e0+1.69e0) avec (2.50e0+1.50e0), vous réaliseriez que ces sommes *ne sont pas* égales. Le test de l'égalité réussit si et seulement si tous les bits (ou *digits*) des deux opérandes sont exactement pareils. Puisque ceci n'est pas nécessairement vrai après deux calculs à virgule flottante produisant le même résultat, un test strict pour égalité pourrait échouer.

La manière standard de comparer l'égalité de deux nombres réels est de déterminer combien de marge d'erreur (ou tolérance) vous allez permettre dans une comparaison et vérifier si une valeur est dans la même marge d'erreur que l'autre. La façon simple de faire ceci est d'utiliser un test comme le suivant :

```
if Valeur1 >= (Valeur2-erreur) and Valeur1 <= (Valeur2+erreur) then ...
```

Un autre moyen commun de réaliser la même comparaison est d'utiliser un code comme ce qui suit :

```
if abs(Valeur1-Valeur2) <= erreur then ...
```

Beaucoup d'ouvrages, en parlant des comparaisons réelles se limitent à discuter de l'égalité et ne vont pas plus loin, comme à supposer que les autres formes de comparaison fonctionnent parfaitement en arithmétique flottante. Ce qui est faux ! Si l'on suppose que x=y si x est dans y±erreur, alors une simple comparaison de bits prouverait que x<y si y>x, mais y<y+erreur. Cependant, dans ces cas x devrait réellement être traité comme étant égal à y et non moindre que y. Par conséquent, on doit toujours comparer deux nombres à virgule flottante en utilisant des plages, peu importe la véritable comparaison qu'on voudrait effectuer. Essayer de comparer deux nombres réels directement pourrait produire des erreurs. Pour comparer deux nombres à virgule flottante, il faut utiliser l'une des deux formes suivantes :

```
=       if abs(x-y) <= erreur then ...
≠       if abs(x-y) > erreur then ...
<       if (x-y) < erreur then ...
≤       if (x-y) <= erreur then ...
>       if (x-y) > erreur then ...
≥       if (x-y) >= erreur then ...
```

Il faut faire attention quand on choisit une valeur pour *erreur*. Cela devrait être une valeur légèrement plus grande que la plus grande marge d'erreur qu'on peut obtenir dans les calculs. La valeur exacte dépendra du format à virgule flottante spécifique que vous utiliserez, mais nous parlerons mieux de cela un peu plus tard. La règle finale que l'on spécifiera dans cette section est :

> *En comparant deux valeurs à virgule flottante, vérifiez toujours qu'une des deux valeurs est dans la plage donnée par la seconde, plus (ou moins) une certaine marge d'erreur.*

D'autres petits problèmes peuvent arriver avec les valeurs à virgule flottante. Ce livre ne pourra discuter que de certains problèmes majeurs et vous rendre conscients du fait que vous ne pouvez pas traiter les nombres à virgule flottante de la même façon que les nombres réels dans l'algèbre traditionnelle. Les inexactitudes inévitables de la précision limitée de l'arithmétique numérique pourrait vous mettre en sérieux embarras si vous ne faites pas attention. Un bon livre sur l'analyse numérique ou même sur le calcul scientifique pourrait vous aider à comprendre les détails qui ne seront pas couverts ici. Si vous allez travailler en arithmétique à virgule flottante, vous devriez prendre le temps d'étudier les effets de la précision limitée dans vos calculs, cela reste vrai *dans tous les langages de programmation.*

14.2 Les formats à virgule flottante IEEE

Quand les experts d'Intel ont conçu le co-processeur à virgule flottante pour leur nouveau microprocesseur 8086, ils étaient assez intelligents pour réaliser que les ingénieurs électroniques et les physiques qui allaient projeter les pouces, n'étaient peut-être pas les meilleurs candidats pour faire l'analyse numérique nécessaire pour déterminer la meilleure représentation binaire possible pour un format à virgule flottante. Donc ils ont embauché les meilleurs analystes qu'ils pouvaient trouver pour projeter un format adécuat pour leur pouce 8087. Les personnes qui ont réalisé cette tâche étaient Kahn, Coonan et Stone. Leur travail a permis le standard KCS que l'organisation IEEE adopta aussi[1].

Pour satisfaire certaines exigences de précision et de performance, Intel a conçu trois formats à virgule flottante: précision simple, double précision et précision étendue. Les premier deux formats correspondent aux types de données float et double du langage C, ou bien aux types real et double du FORTRAN. Intel a conçu la précision étendue pour traiter les chaînes longues de calcul. La précision étendue permet 16 bits supplémentaires pouvant être utilisés pour accroître la précision et éviter les erreurs dus à l'arrondissement.

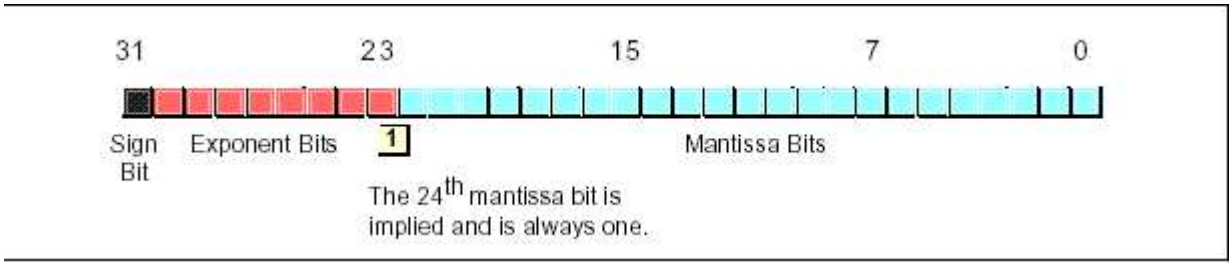

Figure 14.2 Le format de précision simple

Le format à précision simple utilise une mantisse de 24 bits et un excès exponentiel de 8 bits. La mantisse représente normalement une valeur entre 1.0 et 2.0. Son bit le plus significatif est supposé être toujours 1 et représente une valeur qui

[1] À part certains changements mineurs sur le traitement de certaines opérations dégénérées, la représentation IEEE était pour le reste inchangée.

se trouve juste à la gauche du *point binaire*[2]. Les 23 bits restants de la mantisse apparaissent à la droite du point binaire. Par conséquent, la mantisse représente la valeur:

```
1.mmmmmmmm mmmmmmmm mmmmmmmm
```

Le caractère "mmmm..." représente les 23 bits de la mantisse. Gardez à l'esprit qu'ici on est en train de travailler avec des nombres binaires. Par conséquent, chaque position à la droite du point binaire représente une valeur (zéro ou un) de valeurs successives à la puissance négative de deux. Le premier bit est toujours multiplié par 2^0, qui est un. C'est pourquoi la mantisse est toujours supérieure ou égale à 1. Même si tous les autres bits de la mantisse sont à 0, le premier bit donne toujours la valeur 1[3]. Certainement, même si on a un nombre infini de bits valant un après le point binaire, cette valeur n'arriverait pas à deux. C'est pourquoi la mantisse ne peut que représenter des valeurs comprises entre l'intervalle [1, 2[.

Bien qu'il y ait un nombre infini de valeurs entre 1 et 2, on ne peut que représenter huit millions d'elles, car la mantisse est composée de 23 bits (et le 24^{me} bit est toujours 1). Cela est précisément la cause de l'imprécision due aux opérations en virgule flottante, dans tous les calculs concernant la précision simple, on est limité à une précision de 23 bits.

La mantisse utilise le format de *complement à un* au lieu que le format de complément à deux. Cela veut dire que las valeur de 24 bits de la mantisse est simplement un nombre binaire non signé et le bit du signe détermine quand la valeur est positive ou négative. Les nombres complémenté à un ont la propriété inusuelle d'avoir deux représentations possibles pour la valeur zéro (avec le signe de bit à 1 ou 0). Généralement, cela a d'importance seulement pour les gens qui conçoivent des systèmes logiciels ou matériels à virgule flottante. Mais on supposera que la valeur zéro a toujours le bit de signe à zéro.

Pour réprésenter des valeurs hors de la plage [1.0, 2.0[, la portion exponentielle du format à virgule flottante nous vient en aide. Le format à virgule flottante élève deux à la puissance spécifiée par l'exposant et multiplie la mantisse pour cette valeur. L'exposant est de huit bits et il est gardé dans un format *excès-127*. Dans ce format, l'exposant 2^0 est représenté par la valeur 127 (7fh). Par conséquent, pour convertir un exposant dans ce format, il suffit d'additionner la valeur 127 à la valeur de l'exposant. L'usage de ce format rend plus facile la comparaison des valeurs à virgule flottante. La figure 14.2 illustre la représentation du format à virgule flottante en précision simple.

Avec une mantisse de 24 bits, on peut obtenir approximativement 6 - 1/2 chiffres de précision (le 1/2 signifie que les premiers six chiffres peuvent se trouver tous dans la plage 0..9, mais le septième chiffre peut seulement se trouver dans la plage 0..x où x<9 et il est généralment proche de 5). Con un excès exponentiel de huit bits, la plage dynamique de la précision simple est parfaitement acceptable pour beaucoup d'application. Cependant cette plage est parfois insuffisante pour certaines applications scientifiques et cette précision est vraiment très limitée pour beaucoup d'applications financières, scientifiques et ainsi de suite. De plus, dans des longues chaînes de calculs, la précision limitée de la précision simple peut faire encourir à de serieuses erreurs.

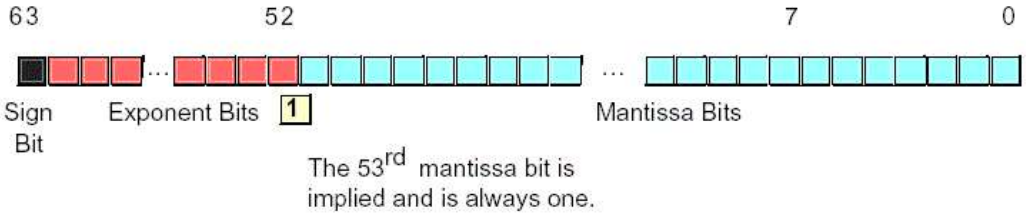

Figure 14.3 Format à double précision de 64 bits

Le format à précision double permet de remédier à cet inconvénient.. Utilisant le double de l'espace, le format à précision double possède un excès de 11 bits, un exposant de 1023 bits et une mantisse de 53 bits (avec un bit fort implicite valant 1), plus un bit de signe. Ce qui permet une plage dynamique d'environ $10^{\pm 308}$ et 14-½ chiffres de précision, ce qui est

[2] Le point binaire est la même chose que le point décimal, sauf qu'on le voit dans des nombres binaires au lieu que dans des nombres décimaux.

[3] À vrai dire, ceci n'est pas nécessairement vrai. Le format à virgule flottante IEEE supporte des valeurs dénormalisées où le bit le plus significatif est juste une valeur différente de zéro. Cependant, nous ignorerons ici les valeurs dénormalisées.

parfaitement suffisant pour la plupart des applications. Les valeurs à précision double prennent la forme montrée à la figure 14.3.

Afin d'aider à assurer la précision avec les longues chaînes de calculs concernant la précision double, Intel a conçu également un format de précision étendue. Ce format se sert de 80 bits. Douze des 16 bits supplémentaires servent de prolongement à la mantisse et les quatre restant prolongent l'exposant. Contrairement aux valeurs à précision simple et double, le format étendu ne possède pas un bit fort implicite qui est toujours à un. Par conséquent, le format étendu fournit une mantisse de 64 bits, un excès de 15 bits et un exposant de 16383, plus un bit de signe. Le format pour la précision étendue est montré à la figure 14.4.

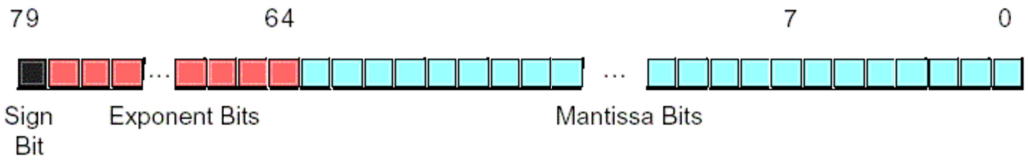

Figure 14.4 Le format à précision étendue

Dans les FPU 80x87 et les CPU 80486, tous les calculs sont faits à l'aide du format étendu. Même si vous chargez une valeur à précision simple ou double, le FPU la convertit automatiquement en valeur étendue. De même, quand vous gardez une valeur de précision simple ou double en mémoire, le FPU l'arrondit automatiquement en la taille appropriée avant de la stocker. En travaillant toujours avec le format à précision étendue Intel garantit un grand nombre de bits disponible pour assurer la précision des calculs. Certains livres affirment par erreur que vous ne devriez jamais utiliser le format à précision étendue dans vos programmes parce qu'Intel garantit la précision seulement à condition d'utiliser les formats à précision simple ou double. Ce qui est tout à fait faux. Au contraire, en effctuant tous les calculs avec 80 bits, Intel aide à assurer (tout en ne garantissant pas) que vous obtendrez toujours des précisions de 32 ou de 64 bits dans vos calculs. Puisque les FPU 80x87 et les CPU 80486 ne fournissent pas un grand nombre de bits de garde dans les calculs utilisant 80 bits, certaines erreurs pourraient apparaître dans les bits faibles. Cependant, si vos calculs sont corrects à 64 bits, les calculs à 80 bits pourront toujours fournir *au moins* 64 bits corrects. Et la plupart du temps vous obtiendrez même plus de bits corrects. Et quand on ne peut pas assurer d'obtenir un résultat correct en utilisant tous les 80 bits, on peut normalement obtenir mieux que 64 en utilisant le format à précision étendue.

Pour assurer une précision maximale pendant des calculs, on utilise la plupart des fois des valeurs *normalisées*. Le bit le plus significatif de la mantisse d'une telle valeur est toujours à 1. Presque toutes les valeurs non normalisées peuvent être normalisées en décalant la mantisse à la gauche et en décrémentant l'exposant d'une unité jusqu'à l'apparition d'un un dans le bit le plus significatif de la mantisse. N'oubliez pas que l'exposant est un exposant binaire. Chaque fois que vous l'incrémentez, vous multipliez la valeur à virgule flottante par deux et chaque fois que vous le décrémentez, vous divisez cette valeur par deux. En suivant le même raisonnement, si vous décalez la mantisse vers la gauche, vous multipliez en fait la valeur à virgule flottante par deux. De même, si vous décalez la mantisse vers la droite, vous divisez la valeur par deux. Par conséquent, décaler la mantisse vers la gauche d'une position *quand* vous décrémentez en même temps l'exposant, vous ne changez pas la valeur du tout.

Utiliser des nombres à virgule flottante normalisés est une bonne idée parce que cela maintient le nombre maximum de bits en précision pendant des opérations. Si tous les bits les plus significatifs de la mantisse sont à zéro, la mantisse se trouve avec moins de bits disponibles pour le calcul. Par conséquent, un calcul à virgule flottante sera plus précis s'il implique seulement des valeurs normalisées.

Il y a deux cas importants où un nombre à virgule flottante ne peut pas être normalisé. La valeur 0.0 est un cas spécial. Évidemment, elle ne peut pas être normalisée parce que la représentation à virgule flottante du nombre zéro n'a aucun bit à un dans la mantisse. Cela n'est pas un problème cependant, car on peut représenter la valeur zéro même avec un seul bit.

Le second cas est quand on a certains bits forts de la mantisse à zéro, mais les exposants partiels sont également à zéro (l'exposant le plus négatif que possible), le standard IEEE permet des valeurs *dénormalisées* spéciales pour

représenter ces valeurs très petites[4]. Bien que l'usage des valeurs dénormalisées selon la norme IEEE produit des meilleurs résultats que s'il y avait un simple dépassement de capacité, gardez à l'esprit que ces valeurs offrent définitivement moins de bits de précision et elles sont moins précises.

Puisque les FPU 80x87 et les CPU 80486 convertissent toujours les valeurs à précision simple et double en des valeurs en précision étendue, l'arithmétique directe en précision étendue est en définitive *plus rapide* que la précision simple ou double. Ce qui fait imaginer que la vitesse attendue des calculs à précision simple ou double est imaginaire, car ce type de format n'est plus implémenté dans ces pouces. Cependant, en projectant le Pentium/586, Intel a reconçu l'unité à virgule flottante pour mieux rivaliser avec les chips RISC. Donc, le format à double précision résulte finalement le plus rapide dans les processeurs Pentium et ultérieurs.

14.3 Les routines à virgule flottante de la bibliothèque UCR standard

Dans la plupart des ouvrages d'assembleur il y a des sections décrivant comment concevoir vos propres routines à virgule flottante pour l'addition, la soustraction, la multiplication et la division. Cet ouvrage ne le fera pas et cela pour diversese raisons : avant tout, pour concevoir une *bonne* bibliothèque de routines à virgule flottante il faut avoir des connaissances solides en analyse numérique, une condition que pas tous les lecteurs de ce livre sont supposés d'avoir. En second lieu, la bibliothèque standard UCR fournit déjà un ensemble raisonnable de routines à virgule flottante sous forme de code source. Donc, pourquoi gaspiller de l'espace ici si des sources sont disponibles ailleurs ? Troisièmement, les pouces à virgule flottante des CPU et des cartes mères sont désormais une norme, donc il n'y a plus de nécessité de décrire comment effectuer manuellement un calcul à virgule flottante, tout comme ce n'est pas nécessaire de décrire comment effectuer manuellement un calcul d'entiers. Par conséquent, cette section décrira comment utiliser la bibliothèque standard UCR au cas où vous ne disposiez pas de FPU disponible ; alors qu'une section ultérieure décrira comment se servir de l'unité à virgule flottante (FPU).

La biblithèque standard UCR fournit un vaste nombre de routines pour les calculs et les E/S avec des nombres à virgule flottante. Cette bibliothèque se sert du même format de mémoire pour les nombres à 32, 64 et 80 bits, tout comme les FPU 80x87. Ces routines ne respectent pas nécessairement les normes IEEE par rapport aux conditions d'erreur et aux cas de dégénération et peut produire des résultats légèrement différents par rapport à un FPU 80x87. Mais les résultats sont assez fidèles[5]. Par le fait que les routines de la bibliothèque standard utilise le même format de mémoire pour les nombres de 32, 64 et 80 bits (tout comme les FPU 80x87), vous pouvez interchanger librement des calculs utilisant le FPU ou la bibliothèque standard.

La bibliothèque standard fournit beaucoup de routines pour manipuler des nombres à virgule flottante. Cette section en décrit chacune par catégorie d'appartenance.

14.3.1 Routines de chargement et de stockage

Pusqu'un CPU sans FPU ne fournit pas des registres de 80 bits, la bibliothèque standard doit utiliser des variables en mémoire pour garder des valeurs à virgule flottante pendant un calcul. Celle-ci utilise deux *pseudo registres*, un accumulateur et une opérande. Par exemple, l'addition à virgule flottante additionne le registre d'opérande au registre accumulateur, en laissant le résultat dans ce dernier. Les routines de chargement et de stockage vous permettent de charger des valeurs à virgule flottante dans l'accumulateur comme dans l'opérande et de restituer ces données à la mémoire. Les routines appartenant à cette catégorie incluent accop, xaccop, lsfpa, ssfpa, ldfpa, sdfpa, lefpa, sefpa, lefpal, lsfpo, ldfpo, lefpo et lefpol.

La routine accop copie la valeur du pseudo-registre accumulateur au pseude-registre opérande. Cette routine est utilie quand on veut utiliser le résultat d'un calcul comme seconde opérande pour une opération ultérieure.

La routine xaccop permute les valeurs entre le pseudo-registre accumulateur et le pseudo-registre opérande. Notez cependant que certaines opérations à virgule flottante écrasent la valeur dans le pseudo-registre opérande, donc vous ne

[4] L'alternative serait le dépassement de capacité vers zéro.

[5] Noter que les résultats ne sont jamais les mêmes, pas plus entre les FPU eux-mêmes, en incluant ceux de la famille Intel. Donc, ce n'est pas important si la bibliothèque standard UCR ne produit pas les mêmes résultats qu'un FPU spécifique.

pouvez pas compter sur le fait que le contenu de ce registre soit préservé. Par conséquent, appeler cette routine n'a de sens que quand on effectue une opération dont on sait qu'elle n'affecte pas le registre opérande.

lspa, ldfpa et lefpa chargent l'accumulateur avec une valeur à précision simple, double ou étendue respectivement. La bibliothèque standard UCR utilise son propre format interne pour les calculs. Ces routines convertissent les valeurs spécifiées en leur format interne pendant le chargement. En utilisant chacune d'elles, es:di doit contenir l'adresse de la variable que vous voulez charger dans l'accumulateur à virgule flottante. Le code suivant illustre comment appeler ces routines :

```
rVar real4 1.0
drVar real8 2.0
xrVar real10 3.0
.
.
.
lesi rVar
lsfpa
.
.
.
lesi drVar
ldfpa
.
.
.
lesi xrVar
lefpa
```

Les routines lsfpo, ldfpo et lefpo sont semblables à lsfpa, ldfpa et lefpa, mais celles-ci chargent le pseudo-registre opérande au lieu du pseudo-registre accumulateur, toujours avec la valeur à l'adresse es:di.

lefpal et lefpol chargent l'accumulateur à virgule flottante ou l'opérande à virgule flottante avec une constante littérale de 80 bits apparaissant dans le flux de code. Pour utiliser ces deux routines, faites simplement l'appel avec une directive real10 et la constante appropriée, par exemple :

```
lefpal
real10 1.0
lefpol
real10 2.0e5
```

Les routines ssfpa, sdfpa et sefpa chargent la valeur de l'accumulateur dans la mémoire, en fonction de la variable à virgule flottante se trouvant dals es:di. Cependant il n'existe pas des routines ssfpo, sdfpo ou sefpo parce qu'un résultat qu'on voudrait garder ne devrait jamais apparaître dans le pseudo-registre opérande. S'il vous arrive de vouloir garder en mémoire une valeur se trouvant dans le pseudo-registre opérande, utilisez simplement la routine xaccop pour permuter entre accumulateur et opérande et gardez ensuite le résultat. Le code suivant illustre l'utilisation de ces routines :

```
rVar real4 1.0
drVar real8 2.0
xrVar real10 3.0
.
.
.
lesi rVar
ssfpa
.
.
.
lesi drVar
sdfpa
.
.
.
lesi xrVar
```

14.3.2 Conversions entier/virgule flottante

La bibliothèque standard UCR fournit diverses routines pour convertir entre des entiers et des nombres à virgule flottante. Ces routines sont itof, utof, ltof, ultof, ftoi, ftou, ftol et ftoul. Les premières quatre convertissent des entiers signés et non signés en format à virgule flottante, les dernière quatre tranchent un nombre à virgule flottante et le convertissent en entier.

itof convertit la valeur signée de 16 bits se trouvant dans ax en une valeur à virgule flottante et laisse le résultat dans l'*accumulateur* à virgule flottante. Cette routine n'affecte pas le *registre* opérande à virgule flottante. utof convertit la valeur non signée se trouvant dans ax en une valeur à virgule flottante de façon similaire. ltof et ultof convertissent l'entier de32 bits signé (ltof) ou non signé (ultof) se trouvant dans dx:ax en une valeur à virgule flottante, en laissant le résultat dans l'accumulateur à virgule flottante. Ces routines n'échouent jamais.

ftoi convertit la valeur se trouvant dans l'accumulateur à virgule flottante en une valeur entière signée, en laissant le résultat dans ax. La conversion est par tranchement, pas par arrondissement : cette routine ne conserve que la partie entière et ignore la partie fractionnaire. Si un dépassement de capacité se produit parce que l'entier résultant est trop grand pour 16 bits, alors ftoi retourne le drapeau de retenue mis à 1. Si la conversion se produit sans erreur, ftoi retourne le drapeau mis à 0. ftoi fonctionne de façon similaire, mais la valeur qu'elle convertit et qu'elle garde dans ax est une valeur non signée. Et elle retourne le drapeau mis à 1 seulement si la valeur à virgule flottante était négative.

ftol et ftoul convertissent une valeur se trouvant dans l'accumulateur à virgule flottante en un entier de 32 bits, en laissant le résultat dans dx:ax. ftol fonctionne sur les valeurs signées, alours que ftouls fonctionne sur celles non signées. Tout comme ftoi et ftou, ces routines retournent le drapeau de retenue mis à un si une erreur de conversion se produit.

14.3.3 Arithmétique à virgule flottante

L'arithméthque à virgule flottante est traitée par les routines fpadd, fpsub, fpcmp, fpmul et fpdiv. fpadd additionne la valeur à virgule flottante du pseudo-accumulateur par la valeur du pseudo-registre opérande[6]. fsub soustrait la valeur se trouvant dans le pseudo-registre opérande de la valeur se trouvant dans le pseudo-registre accumulateur. pfmul multiplie la valeur du pseudo-accumulateur avec celle du pseudo-registre opérande. fpdiv divise la valeur de l'accumulateur à virgule flottante avec la valeur du reguistre opérande. fpcmp compare la valeur de l'accumulateur à virgule flottante avec la valeur de l'opérande à virgule flottante.

14.3.4 Conversions virgule flottante / texte et printff

La bibliothèque standard fournit trois routines, ftoa, etoa et atof permettant de convertir des nombres à virgule flottante en chaînes ASCII et vice-versa. Elle fournit également une version spéciale de printf, printff permettant d'afficher des valeurs à virgule flottante tout comme d'autres types de données.

ftoa convertit un nombre à virgule flottante en une chaîne ASCII, qui est une représentation décimale de ce nombre à virgule flottante. Le nombre à convertir en chaîne se trouve dans l'accumulateur à virgule flottante, alors que es:di pointe sur sur le tampon en mémoire où ftoa placera la chaîne convertie. Le registre al contiendra la largeur du champ (nombre de positions d'affichage). Le registre ah contiendra le nombre de positions à afficher à la droite du point décimal. Si ftoa ne pourra pas afficher le nombre en fonction des données fournies par al et ah, alors la routine créera une chaîne de *n* caractères dont la longueur sera indiquée par le contenu de ah. es:di doit pointer sur un tableau d'octets contenant au moins al+1 caractères et doit contenir au moins cinq. Les valeurs de largeur de champ et longueur décimale des registres al et ah sont semblables aux valeurs apparaissant après le point décimal de cette instruction d'écriture en Pascal :

```
write(floatval:al:ah);
```

[6] L'auteur mentionne ici deux fois l'accumulateur, mais il s'agit sans doute d'une confusion, n.d.t.

etoa produit une sortie du nombre à virgule flottante en format exponentiel. Tout comme avec ftoa, es:di pointe sur le tampon où etoa gardera le résultat. Le registre al doit contenir au moins huit et c'est la largeur de champ pour le nombre. Si al contient moins de huit, etoa sortira une chaîne de *n* caractères. La chaîne pointée par es:di doit avoir au moins al+1 caractères. Cette routine de conversion est semblable à la procédure write du langage pascal, en sortant des valeurs réelles avec une seule spécification de largeur de champ :

```
write(realvar:al);
```

La routine printff de la bibliothèque standard fournit toutes les fonctionnalités de la routine standard printf, plus l'habileté de produire des sorties à virgule flottante. Elle inclut plusieurs nouvelles spécifications de format selon qu'on veuille afficher les nombres en forme décimale ou en notation scientifique. Ces spécifications sont :

- x.yF Affiche un nombre réel de 32 bits en format décimal.
- %x.yGF Affiche un nombre réel de 64 bits en format décimal.
- %x.yLF Affiche un nombre réel de 80 bits en format décimal.
- %zE Affiche un nombre réel de 32 bits en notation scientifique.
- %zGE Affiche un nombre réel de 64 bits en notation scientifique.
- %zLE Affiche un nombre réel de 80 bits en notation scientifique.

Dans le format spécifié ci-dessus, x et z sont des constantes entières qui indiquent la largeur de champ des nombres à afficher. L'élément y est également une constante entière spécifiant le nombre de positions à afficher après le point décimal. Les valeurs x.y sont comparables avec les valeurs passées à ftoa par al et ah. La valeur z est comparable à la valeur que la routine etoa s'attend depuis le registre al.

A part l'ajout de ces nouveaux formats, la routine printff est identique à printf. Si vous utilisez printff dans vos programmes, vous *n'avez pas* besoin d'utiliser printf aussi. printff possède toutes les fonctionnalités que printf et utiliser les deux gaspillerait seulement de la mémoire pour rien.

14.4 Le coprocesseur 80x87 à virgule flottante

Quand le premier CPU 8086 apparut dans le marché vers la fin des années 70, la technologie des semi-conducteurs n'était pas aussi développée pour permettre l'implementation des instructions à virgule flottante directement dans le CPU. Par conséquent, Intel mit en place un schéma où on pouvait utiliser une seconde puce entièrement dédiée à ce type d'opérations : l'unité à virgule flottante (ou FPU)[7]. Intel réalisa ainsi sa première puce à virgule flottante en l'année 1980. Ce FPU spécifique fonctionna avec les processeurs 8086, 8088, 80186 et 80188. Mais quand on mit le 80286 dans le marché, Intel conçut également un nuoveau FPU, le 80287 accompagnant ce nouveau processeur. Bien que le 80287 fut compatible avec le CPU 80386, Intel conçut un CPU encore meilleur, le 80387, à utiliser avec les systèmes 80386. Alors que le CPU 80486 fut le premier à inclure une puce intégrée à virgule flottante. Peu après, Intel présenta un autre CPU, le 80486sx qui ne fut autre chose qu'un autre 80486 sans le FPU intégré. Naturellement, pour effectuer des opérations à virgule flottante avec ce CPU, il fallait ajouter une puce 80487. Les puces /586 ont toutes un FPU à autes performances intégré. Il n'y a pas de coprocesseur mathématique disponible pour le Pentium.

Pour commodité on fera référence à toutes ces puces avec la terminologie FPU 80x87. Etant donné la vieillesse des puces 8086, 80286, 8087 et 80287, ce manuel se concentrera uniquement sur les puces 80387 et ultérieures. Il y a quelques différences entre les unités à virgule flottante 80387/80486/Pentium et les puces antérieures. Si vous avez besoin d'écrire du code qui s'exécutera sur ces vieilles machines, il vous faudra consulter la documentation appropriée d'Intel pour ces dispositifs.

14.4.1 Les registres du FPU

Le FPU 80x87 ajoute 13 registres aux processeurs 80386 et ultérieurs. Huit registres de données à virgule flottante, un registre de contrôle, un registre d'état, un registre de balise, un pointeur d'instruction et un pointeur de données. Ce dernier est similaire au jeu de registres 80x86, donc tous les calculs à virgule flottante prennent place dans ces registres. Le registre de contrôle contient des bits permettant de décider comment le 80x87 traite certains cas d'erreur comment

[7] Intel nomma également ce dispositif *Numeric Data Processor* (NDP), *Numeric Processor Extension* (NPX) et *Coprocesseur Mathématique*.

l'arrondissement de calculs imprécis, le contrôle de la précision et ainsi de suite. Ce registre est semblable au registre flags des processeurs 80x86 ; il contient les bits d'état de code et plusieurs autres drapeaux à virgule flottante décrivant l'état de la puce 80x87. Le registre balise contient divers groupes de bits déterminant l'état de la valeur dans chacun des huit registres généraux. Les pointeurs d'instruction et de données contiennent certaines informations d'état sur les dernières instructions à virgule flottante exécutées. Dans cet ouvrage nous ne considérons pas les derniers trois registres, consultez la documentation Intel pour plus de détails.

14.4.1.1 Les registres de données du FPU

Le FPU 80x87 fournit huit registres de 80 bits organisés comme une pile. C'est un écart significatif par rapport à l'organisation des registres généraux du 80x86. Intel se réfère à ces registres avec les termes ST(0), ST(1), ..., ST(7). Beaucoup d'assembleurs acceptent ST comme abbréviation de ST(0).

La plus grosse différence entre le jeu des registres du FPU et les registres 80x86 est l'organisation de la pile. Dans un CPU 80x86 le registre ax est toujours ax, peu importe ce qui arrive. Cependant, sur un 80x07, ax est une pile de huit éléments de valeurs à virgule flottante de 80 bits (voir la figure 14.5). ST(0) se réfère à l'élément en tête de la pile, ST(1) fait référence à l'élément suivant et aiusi de suite. Certaines instructions à virgule flottante empilent et désempilent des éléments ; par conséquent, ST(1) fera référence à l'élément précédent de ST(0) après qu'on aura empilé quelque chose dans la pile. Il faudra une certaine réflexion et de la pratique pour s'habituer que ces registres changent selon votre code, mais c'est un problème simple à dépassser.

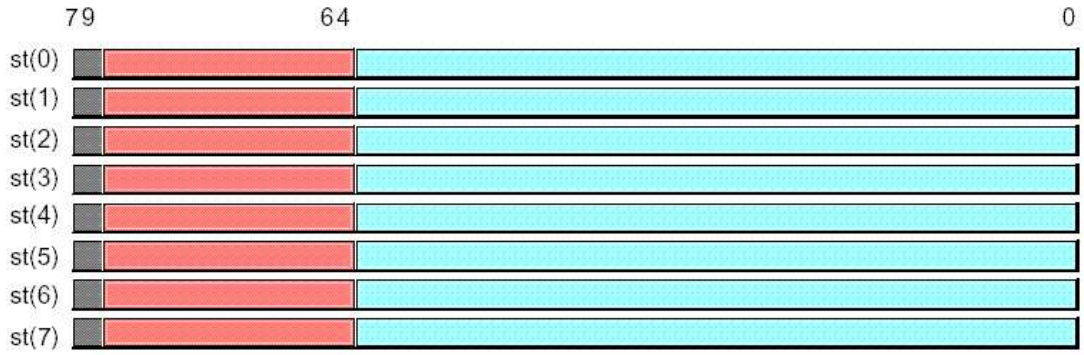

Figure 14.5 Pile de registres à virgule flottante 80x87

14.4.1.2 Le registre de contrôle du FPU

Quand Intel a conçu le 80x87 (et, pratiquement, le standard IEEE), il n'existait pas de standard dans le matériel à virgule flottante. Chaque constructeur d'ordinateur avait des formats différents et incompatibles. Malheureusement, beaucoup de logiciels avaient été écrits tenant en compte la nature de ces différents formats. Intel voulut donc concevoir un FPU pouvant fonctionner avec la plupart de ces logiciels (gardez à l'esprit que l'IBM PC existait déjà depuis un certain nombre d'années avant la construction du 8087, devant se conformer avec le grand nombre de logiciels disponibles). Malheureusement, beaucoup des caractériestiques de ces vieux formats à virgule flottante étaient mutuellement exclusives. Par exemple, dans certains systèmes, un arrondissement avait lieu lors d'une précision insuffisante ; dans d'autres, la partie décimale était coupée au net. Certaines application allaient fonctionner correctement avec certains systèmes et pas d'autres. Et Intel voulait le bon fonctionnement d'autant applications possibles avec les moindres changements possibles dans leur FPU, donc ils ont ajouté un registre spécial, le *registre de contrôle* du FPU qui laissait à l'utilisateur le choix entre différents modes d'opérations.

Ce registre contient donc 16 bits organisés comme montré à la Figure 14.6.

Figure 14.6 Registre de contrôle 80x87

Le bit 12 du registre de contrôle est présent seulement dans les puces 8087 et 80287. Il contrôle comment le 80x87 réactionne à l'infini. Les puces 80387 et ultérieures ont commencé à utiliser une forme d'infini nommée *affine closure* parce que c'est la seule forme admise par le standard IEEE 754/854. Donc, on ignorera toute utilisation de ce bit et on supposera qu'il contient toujours la valeur 1.

Les bits 10 et 11 fournissent un contrôle d'arrondissement selon les valeurs suivantes:

Table 58 : Contrôle d'arrondissement

Bits 10 et 11	Fonction
00	Au plus proche
01	Arrondissement vers le bas
02	Arrondissement vers le haut
03	Troncature

Le réglage "00" est le réglage par défaut. Le 80x87 arrondit les valeurs supérieures à la moitié du bit le moins significant. bit significant vers le haut. Il arrondit les valeurs inférieures à la moitié du bit le moins significant vers le bas. Si la valeur située sous le bit le moins significant est exactement la moitié de celui-ci, le 80x87 arrondit la valeur vers la valeur dont le bit le moins significant est zéro. Pour les longues chaînes de calculs, cela fournit un moyen raisonnable et automatique de maintenir une précision maximale.

Les options d'arrondissement par excès et par défaut sont présentes pour ces calculs où il est important de garder trace de la précision pendant l'opération. En utilisant un arrondissement par défaut, en effectuant l'opération et en faisant un arrondissement par excès ensuite, on peut déterminer les plages minimum et maximum entre lesquelles le vrai résultat correspondra.

L'option de cassure impose à tous les calculs de casser tout bit excedant pendant une opération. Vous utiliserez rarement cette option si la précision est importante pour vous. Cependant, pour des raisons de portabilité avec des vieux programmes qui ont été écrits sans le 80x87, cette option pourrait être utile.

Les bits huit et neuf du registre de contrôle, contrôlent la précision pendant un calcul. Cette possibilité est fournie principalement pour permettre encore la compatibilité avec des vieux programmes, comme requis par le standard IEEE 954. Les bits de contrôle de précisions utilisent les valeurs suivantes:

Tableau 59: Les bits de contrôle de précision de mantisse

Bits 8 & 9	Contrôle de précision
00	24 bits
01	Réservé
10	53 bits
11	64 bits

Pour les applications modernes, les bits de contrôle de précision devraient toujours être mis à "11", de sorte à obtenir 64 bits de précision. Ceci produira le plus précis résultat pendant une opération numérique.

Les bits 0 à 5 sont des masques d'exception (*exception masks*). Ils sont similaires au bit d'activation d'interruption dals le registre flag 80x86. Si ces bits contiennent un 1, la condition correspondante est ignorée par le 80x87. Cependant, si n'importe quel de ces bits contient 0 et la condition correspondante se produit, le FPU générera immédiatement une interruption et le programme pourra traiter une condition de dégénération.

Le bit 0 correspond à une error d'opération incorrecte. Ceci se produit généralement à l'occurrence d'une erreur de programmation. Des problèmes semblables incluent l'empilage de plus de huit éléments dans la pile, le désempilage d'un élément d'une pile vide, ou bien le calcul de la racine carrée d'un nombre négatif, ou encore le chargement d'un registre non vide.

Le bit 1 masque une interruption *non normalisée* qui se produit toutes les fois qu'on essaie de manipuler des valeurs non normalisées. Ces valeurs se produisent quand on charge des valeurs à précision arbitrairement étendue dans le FPU, ou

bien quand on travaille avec des nombres vraiment petits au-delà des capacités du FPU. Normalement on *n'active pas* cette exception.

Le bit 2 masque l'exception de division par zéro. Si ce bit contient zéro, le FPU générera une interruption si on essaie de diviser une valeur par zéro. Si on n'active pas cette exception, le FPU produira un résultat de type NaN (not a number) lors d'une telle division impossible.

Le bit 3 traite l'exception de dépassement de capacité (overflow). Le FPU détectera une telle exception si un calcul de dépassement de capacité se produit ou si vous essayez de stocker une valeur qui est trop grande pour une opérande de destination (par exemple, placer une valeur de précision très étendue dans une variable de précision simple).

Le bit 4, si mis à 1, masque l'exception de *underflow* (dépassement par le bas). Comme dans le cas de overflow, cette exception se produit quand vous stockerz une valeur de précision étendue dans une variable plus petite (à précision simple ou double) ou bien quand le résultat d'une opération est trop petit pour une précision étendue.

Le bit 5 contrôle le cas qui se produit quand une exception de *précision* se produit. Une telle erreur est générée quand le FPU produit un résultat imprécis, généralement le résultat d'une opération d'arrondissement interne. Bien que beaucoup d'opérations produisent un résultat exact, beaucoup d'autres ne le font pas. Par conséquent, ce bit est généralement activé, car les résultats inéxacts sont très communs.

Les bits 6 et les bits 13 à 15 sont actuellement indéfinis et réservés pour un usage futur. Le bit 7 est la masque d'activation d'interruption, mais n'est activé que dans le 8087 ; un zéro dans ce bit permet les interruptions du 8087 et un 1 les désactive.

Le 80x87 fournit deux instructions, FLDCW (load control word) et FSTCW (store control word), vous permettant de charger et de stocker les contenus du registre de contrôle. La seule opérande de cette instruction doit être un emplacement de mémoire de 16 bits. L'instruction FLDCW charge le registre de contrôle depuis un emplacement de mémoire spécifié et l'instruction FSTCW garde le registre de contrôle dans un emplacement de mémoire spécifié.

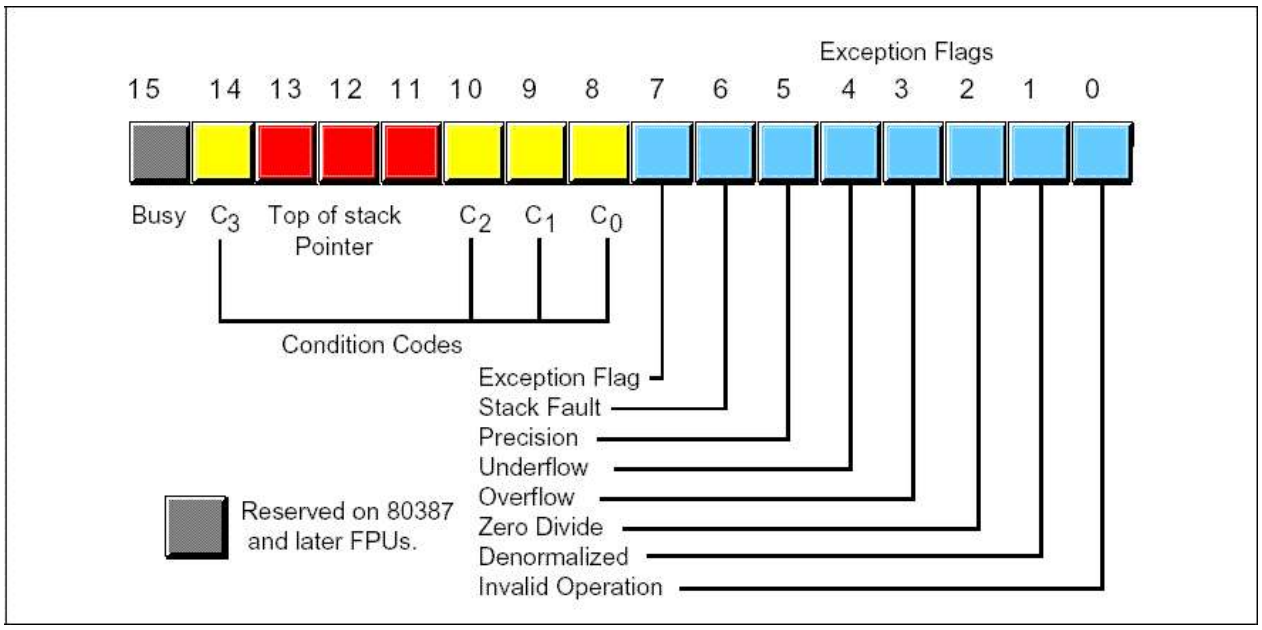

Figure 14.7 Le registre d'état du FPU

14.4.1.3 Le registre d'état du CPU

Le registre d'état du CPU fournit l'état du co-processeur au même instant qu'on le consulte. L'instruction FSTSW stocke le registre d'état dans l'opérande mod/reg/rm. Ce registre est un registre de 16 bits, sa disposition apparaît à la figure 14.7.

Les bits 0 à 5 représentent les drapeaux d'exception. Ces bits apparaissent dans le même ordre que les masques d'exception dans le registre de contrôle. Si la condition correspondante existe, alors le bit est activé. Mais ces bits sont indépendants des bits du registre de contrôle. Le 80x87 active ou désactive ces bits indépendamment de l'état des masques.

Le bit six (actif seulement dans les 80386 et ultérieurs), indique une *défaillance de la pile* (*stack fault*). Ceci arrive quand on dépasse la capacité de la pile soit en haut qu'en bas (overflow / underflow). Quand ce bit est activé, le code de condition C_1 détermine s'il s'est produit un overflow ($C_1 = 1$) ou un underflow ($C_1 = 0$).

Le bit sept est mis à 1 si n'*importe quel* bit d'erreur de condition est activé. C'est un OU logique des bits 0 à 5. Un programme peut interroger ce bit pour déterminer rapidement l'existence d'une erreur de condition.

Les bits 8, 9, 10 et 14 sont les bits des codes de condition du co-processeur. Plusieurs instructions affectent ces bits, selon ce que montre le tableau suivant :

Table 60: Bits de condition de code du FPU

Instruction	Bits de code de condition				Condition
	C3	C2	C1	C0	
fcom, fcomp,	0	0	X	0	ST > source
fcompp,	0	0	X	1	ST < source
ficom,	1	0	X	0	ST ou source indéfini
	X = Pas important				
ftst	0	0	X	0	ST positif
	0	0	X	1	ST négatif
	1	0	X	0	ST nul (+ ou -)
	1	1	X	1	ST incomparable
fxam	0	0	0	0	+ non autorisé
	0	0	1	0	- non autorisé
	0	1	0	0	+ normalisé
	0	1	1	0	- normalisé
	1	0	0	0	+ 0
	1	0	1	0	- 0
	1	1	0	0	+ dénormalisé
	1	1	1	0	- dénormalisé
	0	0	0	1	+ NaN
	0	0	1	1	- NaN
	0	1	0	1	+ infinity
	0	1	1	1	- infinity
	1	X	X	1	registre vide
fucom,	0	0	X	0	ST > source
fucomp	0	0	X	1	ST < source
	1	0	X	0	ST = source
	1	1	X	1	Unorder
	X= pas important				

Table 61: Code d'interprétation de condition

Instruction(s)	C_0	C_3	C_2	C_1
fcom, fcomp, fcmpp, ftst, fucom, fucomp, fucompp, ficom, ficomp	Résultat d'une comparaison. Voir la table ci-dessus.	Résultat de comparaison. Voir la table ci-dessus	L'opérande n'est pas compatible.	Résultat de comparaison (voir la table ci-dessus) ou stack overflow / underflow (si bit d'exception de pile activé).
fxam	Voir la table précédente	Voir la table précédente	Voir la table précédente	Signe du résultat ou stack overflow / underflow (si bit d'exception de pile activé).
fprem, fprem1	Le bit 2 du reste	Le bit 0 du reste	0 réduction faite. 1 réduction incomplète	Bit 1 du reste ou stack overflow / underflow (si bit d'exception de pile activé).
fist, fbstp, frndint, fst, fstp, fadd, fmul, fdiv, fdivr, fsub, fsubr, fscale, fsqrt, fpatan, f2xm1, yl2x, fyl2xp1	Non défini	Non défini	Non défini	Arrondissement par exès ou stack overflow / underflow (si bit d'exception de pile activé).
fptan, fsin, fcos, fsincos	Non défini	Non défini	0 réduction faite 1 réduction incomplète	Arrondissement par exès ou stack overflow / underflow (si bit d'exception de pile activé).
fchs, fabs, fxch, fincstp, fdecstp, *constant loads,* fxtract, fld, fild, fbld, fstp (80 bit)	Non défini	Non défini	Non défini	Résultat nul ou stack overflow / underflow (si bit d'exception de pile activé).
fldenv, fstor	Récupéré des opérandes de mémoire	Récupéré des opérandes de mémoire	Récupéré des opérandes de mémoire	Récupéré des opérandes de mémoire
fldcw, fstenv, fstcw, fstsw,fclex	Non défini	Non défini	Non défini	Non défini
finit, fsave	Mis à zéro	Mis à zéro	Mis à zéro	Mis à zéro

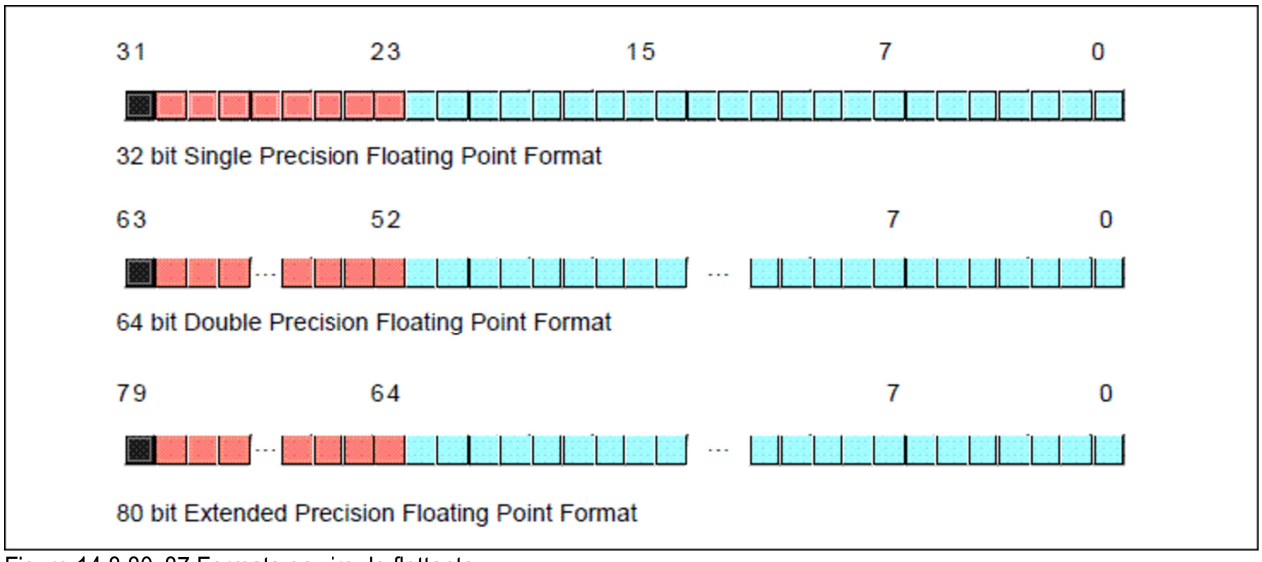

Figure 14.8 80x87 Formats en virgule flottante

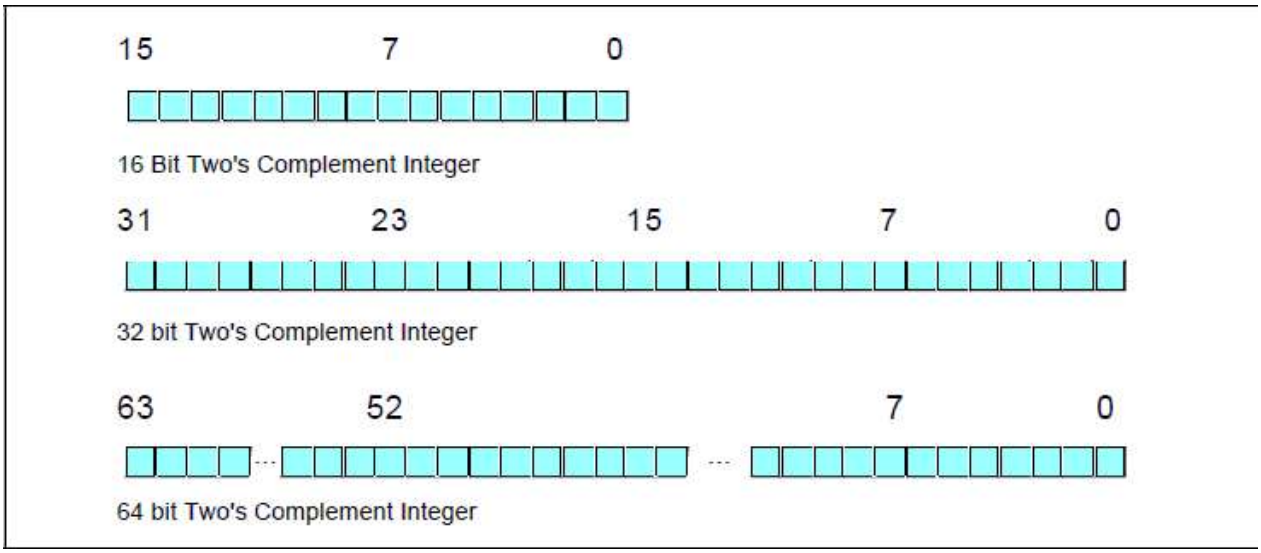

Figure 14.9 80x87 Formats entiers

Les bits 11 à 13 du registre d'état du FPU donnent le numéro de registre du sommet de la pile. Pendant les calculs, le 80x87 ajoute (modulo 8) les nombres de registre *logiques* que le programmeur a fourni à ces trois registres pour déterminer le numéro registre *physique* pendant l'exécution.

Le bit 15 du registre d'état est le bit d'*occupation.* Il est activé quand le FPU est occupé. La plupart des programmes n'auront pas besoin d'avoir accès à ce bit.

14.4.2 Types de données du FPU

Le FPU du 80x87 supporte sept types de données différents : trois types entiers, un type décimal préfabriqué (packed) et trois types à virgule flottante. Pusque le CPU 80x86 supporte déjà les types de données entiers, vous n'avez pas vraiment de raison d'utiliser les types fournis par le 80x87. Le type décimal préfabriqué fournit un entier signé de 17 chiffres (BCD). Tout de même, ce livre ne couvre pas ce type d'arithmétique, donc on va l'ignorer. Les trois types qui restent sont des types à virgule flottante de 32, 64 et 80 bits qu'on a vus précédemment. Les types de données 80x87 apparaissent aux figures 14.8, 14.9 et 14.10.

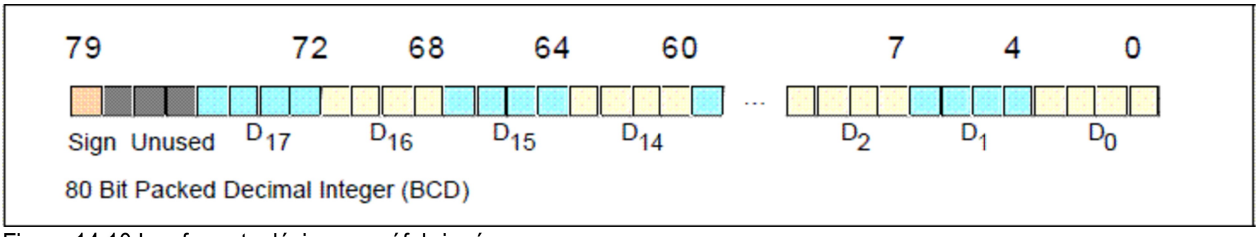

Figure 14.10 Les formats décimaux préfabriqués

Le FPU 80x87 stocke généralement les valeurs dans un format *normalisé*. Quand un nombre à virgule flottante est normalisé, le bit le plus signifiant est toujours 1. Dans les formats à 32 et 64 bits, le 80x87 ne stocke pas ce bit, il prend pour acquis qu'il est à 1. Par conséquent, les nombres à virgule flottante de 32 et de 64 bits sont toujours normalisés. Dans le format à précision étendue de 80 bits, le 80x87 *ne prend pas* pour acquis que le bit fort de la mantisse est un, le bit le plus signifiant du nombre apparaît comme faisant partie de la chaîne de bits.

Les valeurs normalisées permettent la plus grande précision pour un nombre donné de bits. Cependant, il y a un grand nombre de valeurs non normalisées qu'on *peut* repréenter à l'aide du format de 80 bits. Ces valeurs sont très proches de zéro et représentent l'ensemble de valeurs où le bit le plus signifiant de la mantisse est différent de zéro. Le FPU 80x87 supporte une forme spéciale de valeur 80 bits qu'on connaît avec le terme de *valeurs dénormalisées*. Ces valeurs

permettent au FPU d'encoder de très petites valeurs qui ne peuvent pas être encodées en utilisant des valeurs normalisées, mais à un prix : les valeurs dénormalisée offrent moins de bits de précision par rapport aux valeurs normalisées. Par conséquent, utiliser les valeurs dénormalisées peut amener de petites pertes de précision dans un calcul. Sans doute, c'est toujours mieux qu'arrondir les valeurs dénormalisées à zéro (ce qui peut rendre le calcul encore moins précis), mais il faut absolument garder à l'esprit que si vous travaillez avec de très petites valeurs, vous pouvez encourir à une certaine perte de précision. Notez que les registres d'état du 80x87 contiennent un bit que vous pouvez utiliser pour détecter quand le FPU utiliser une valeur dénormalisée.

14.4.3 Le jeu d'instructions du FPU

Le FPU 80387 (et ultérieurs) ajoutent plus de 80 nouvelles instructions dans le jeu 80x86. On peut classifier ces instructions dans les catégories suivantes : *instructions de mouvement de données, conversions, instructions arithmétiques, instructions de comparaison, instructions constantes, instructions transcendantes* et *instructions diverses.* Les paragraphes suivants décrivent chacune de ces instructions dans ces catégories.

14.4.4 Instructions de mouvement de données

Les instructions de mouvement de données transfèrent les données entre les registres internes du FPU et la mémoire. Ces instructions sont fld, fst, fstp et fxch. L'instruction fld pousse toujours son opérande dans une pile à virgule flottante. L'instruction fstp extrait toujours son opérande de cette pile après avoir stocké le sommet dans son espace opérationnel. Les instructions restantes n'affectent pas le nombre d'objets dans la pile.

14.4.4.1 L'instruction FLD

L'instruction fld charge une valeur de 32, 64 ou 80 bits dans la pile. Elle convertit les opérandes de 32 ou 64 bits en une valeur de 80 bits de précision étendue avant de la placer dans la pile.

L'instruction fld décrémente d'abord le pointeur du sommet de pile (bits 11-13 du registre d'état) et puis stocke la valeur de 80 bits dans le registre physique spécifié par le nouveau pointeur de sommet de pile. Si l'opérande source de l'instruction fld est un registre de données à virgule flottante ST(*i*), alors le registre que le 80x87 utilisera pour l'opération de chargement sera le registre correspondant au nombre de registre spécifié *avant* de décrémenter le pointeur du sommet de la pile.

L'instruction fld met à un le bit d'échec de la pile si un erreur de dépassement de pile survient (stack overflow). Et il active le bit d'exception dénormalisée dans l'éventualité qu'une valeur de 80 dénormalisée est chargée. Il active aussi le bit d'opération invalide si vous essayez de charger un registre à virgule flottante vide dans le sommet de la pile (ou si vous effectuez n'importe quelle autre opération non valide).

Exemples:

```
fld     st(1)
fld     mem_32
fld     MaValeurReelle
fld     mem_64[bx]
```

14.4.4.2 Les instructions FST et FSTP

Les instructions fst et fstp copient la valeur qui se trouve au sommet du registre de la pile à virgule flottante dans un autre registre à virgule flottante de 32, 64 ou 80 bits ou bien dans une variable de mémoire ayant les mêmes tailles. En copiant des données dans une variable de 32 ou de 64 bits, la valeur de précision étendue de 80 bits au sommet de la pile est arrondie au plus petit format tel que spécifié dans le bit de contrôle d'arrondissement du registre de contrôle du FPU.

L'instruction fstp extrait la valeur du sommet de la pile lorsqu'on est en train de la transférer dans l'emplacement de destination. Elle fait cela en incrémentant le pointeur du sommet de la pile dans le registre d'état après avoir eu accès aux

données dans st(0). Si l'opérande de destination est un registre à virgule flottante, le FPU stocke la valeur correspondant au nombre de registre spécifié *avant* d'extraire la valeur de la pile.

Exécuter une instruction fstp st(0) extrait la valeur qui se trouve au sommet de la pile sans qu'il y ait un transfert de données. Exemples:

```
fst     mem_32
fstp    mem_64
fstp    mem_64[ebx*8]
fst     mem_80
fst     st(2)
fstp    st(1)
```

Le dernier exemple énuméré extrait st(1) en laissant st(0) dans le sommet de la pile.

Les instructions fst et fstp activeront le bit d'exception de pile si un underflow de pile se produit (tentative de stockage d'une valeur à partir d'une pile de registres vide). Elles activent aussi le bit de précision s'il se produit une perte de précision durant l'opération de stockage (cela peut arriver si, par exemple, on tente de stocker une valeur de précision étendue de 80 bits dans une variable de memoire de 32 ou de 64 bits et des pertes de précision se produisent pendant la conversion). Elles activent le bit d'exception de capacité inverse (underflow exception) quand on tente de stocker une valeur de 80 bits dans une variable de mémoire et que la valeur est trop petite pour entrer dans l'opérande de destination. De même, ces instructions activeront l'exception de dépassement de capacité (overflow exception) si la valeur au sommet de la pile et trop grande pour entrer dans une variable de 32 ou de 64 bits. Les instructions fst et fstp activent le drapeau de dénormalisation quand vous tentez de stocker une valeur dénormalisée dans un registre ou une variable de 80 bits[8]. Elles activent le flag d'opération invalide si une opération non valide (par exemple empiler un registre vide) se produit. Et, finalement, elles mettent le bit de condition C_1 si un arrondissement se produit pendant l'opération de stockage (ceci se produit seulement quand on essaie de garder une valeur dans une variable de 32 ou de 64 bits et on doit arrondir la mantisse pour que la valeur puisse entrer dans la destination).

14.4.4.3 L'instruction FXCH

L'instruction fxch permute la valeur au sommet de la pile avec un des autres registres du FPU. Cette instruction prend deux formes : une avec un seul registre du FPU comme opérande et l'autres sans opérandes. La seconde forme de fxch permute le sommet de la pile avec st(1).

Plusieurs instructions du FPU, par exemple fsqrt, fonctionnent seulement sur le sommet de la pile de registre. Si vous voulez effectuer une telle opération sur une valeur qui ne se trouve pas au sommet de la pile, vous pouvez utiliser l'instruction fxch pour permuter ce registre avec ce sommet, effectuer l'opération désirée et donc utiliser encore fxch pour permuter celui-ci avec le registre originel. L'exemple suivant prend la racine carrée de st(2) :

```
fxch    st(2)
fsqrt
fxch    st(2)
```

L'instruction fxch active le bit d'exception de pile si celle-ci est vide. Elle active le bit d'opération invalide si vous spécifiez un registre vide comme opérande. Cette instruction met toujours à zéro le bit de condition de code C_1.

14.4.5 Conversions

La puce 80x87 effectue toutes les opérations arithmétiques sur des quantités réelles de 80 bits. Dans un certains sens, les instructions fld et fst/fstp sont des instructions de conversion en même titre qu'elles sont des instructions de mouvement de données, puisque elles convertissent automatiquement entre le véritable format interne de 80 bits et les formats de mémoire de 32 et de 64 bits. Néanmoins, nous les classerons simplement comme des opérations de déplacement de données plutôt que de conversions, car elles déplacent des valeurs réelles vers et depuis la mémoire. Le

[8] Stocker une valeur dénormalisée dans une variable de mémoire de 32 ou de 64 bits activera toujours le bit d'exception de dépassement de capacité négative (underflow exception bit).

FPU 80x87 fournit cinq routines convertissant entre entiers et binaires codés décimaux (BCD) pendant le mouvement de données. Ces instructions sont fild, fist, fistp, fbld, et fbstp.

14.4.5.1 L'instruction FILD

Cette instruction (integer load) convertit des entiers complémentés 2 de 16, 32 ou 64 bits en format à 80 bits de précision étendue et place le résultat sur la pile. Cette instruction s'attend toujours une opérande. Cette opérande doit être l'adresse d'une variable entière de type word, double word ou quad word. Bien que le format de l'instruction utilise le mod/rm familier, l'opérande doit être une variable, même pour des entiers de 16 et de 32 bits. Vous ne pouvez pas spécifier des registres de 16 ou de 32 bits. Si vous voulez pousser un registre dans la pile FPU, vous devez d'abord le stocker dans une variable de mémoire et puis utiliser fild pour placer cette valeur dans la pile.

L'instruction fild active le bit d'exception de pile et C_1 (respectivement) si un dépassement de capacité de la pile a lieu. Exemples:

```
fild     mem_16
fild     mem_32[ecx*4]
fild     mem_64[ebx+ecx*8]
```

14.4.5.2 Les instructions FIST et FISTP

Les instructions fist et fistp convertissent la variable de précision étendue de 80 bits au sommet de la pile en un entier de 16, 32 ou 64 bits et stocke le résultat dans la variable de mémoire spécifiée par l'opérande. Ces instructions effectuent la conversion en accord avec les réglages d'arrondissement du registre de contrôle du FPU (bits 10 et 11). Comme dans le cas de l'instruction fild, ces instructions ne vous permettent pas de spécifier un registre comme opérande de destination. L'instruction fist convertit la valeur au sommet de la pile en un entier et puis stocke le résultat ; elle n'affecte pas le registre de pile à virgule flottante d'une autre façon. L'instruction fistp enlève la valeur du registre de pile à virgule flottante après avoir stocké la valeur convertie.

Ces instructions affectent le bit d'exception de pile si le registre de pile est vide (ceci nettoiera aussi C_1). Et elles affectent les bits de précision (opération imprécise) et C_1 si un arrondissement se produit (c'est-à-dire s'il y a une partie fractionnelle dans la valeur de st(0)). Ces instructions activent l'exception underflow si le résultat est trop petit (c'est-à-dire plus petit que 1 et plus grand que 0 ou plus petit que 0 et plus grand que -1). Exemples:

```
fist    mem_16[bx]
fist    mem_64
fistp   mem_32
```

N'oubliez pas que ces instruction utilisent les règlages d'arrondissement pour déterminer comment elles convertissent les valeurs à virgule flottante en entiers pendant l'opération de stockage. Par défaut, le contrôle d'arrondissement est mis en mode "round" ; néanmoins, la plupart des programmeurs s'attendent une cassure et pas un arrondissement. Si vous voulez tronquer les valeurs à virgule flottante, vous devrez ajuster les bits de contrôle de façon appropriée dans le registre de contrôle à virgule flottante.

14.4.5.3 Les instructions FBLD et FBSTP

Les instructions fbld et fbstp chargent et stockent des valeurs de 80 bits BCD. L'instruction fbld convertit une valeur BCD en son équivalent nombre à virgule flottante de 80 bits en précision étendue et stocke le résultat dans la pile. L'instruction fbstp enlève la valeur à virgule flottante du sommet de la pile, elle la convertit en un nombre BDC de 80 bits (avec arrondissement en conformité aux règlages du registre de contrôle à virgule flottante), et stocke la valeur convertie à l'adresse spécifiée par l'opérande de mémoire de destination. Notez qu'il n'existe pas une instruction fbst pouvant stocker la valeur dans le sommet de la pile sans la désempiler.

L'instruction fbld active le bit d'exception de la pile et le bit C_1 s'il se produit un stack overflow. Elle active le bit d'opération invalide si vous cherchez de charger une valeur BCD invalide. L'instruction fbstp active le bit d'exception de

pile et nettoie le bit C_1 s'il se produit un dépassement de pile vers le bas (stack underflow), c'est-à-dire si la pile est vide. Elle active le drapeau underflow sous le mêmes conditions que fist et fistp. Exemples:

```
; En supposant qu'il y a moins de 8 éléments dans la pile
; la séquence de code suivante est équivalente à
; l'instruction fbst:
        fld     st(0)   ; valeur dupliquée au sommet de la pile
        fbstp   mem_80

; L'exemple suivant convertit facilement une valeur de 80 bits
; BCD en un entier de 64 bits:
        fbld    bcd_80          ; Obtenir la valeur BCD à convertir
        fist    mem_64          ; Stocker cette valeur comme entier
```

14.4.6 Instructions arithmétiques

Les instructions aritmétiques constituent un sous-ensemble petit, mais important du jeu d'instructions 80x87. Ces instructions tombent dans deux catégories générales : celles qui travaillent sur des valeurs réelles et celles qui travaillent sur les valeurs aussi bien réelles et entières.

14.4.6.1 Les instructions FADD et FADDP

Ces deux instructions prennent les formes suivantes :

```
fadd
faddp
fadd    st(i), st(0)
fadd    st(0), st(i)
faddp   st(i), st(0)
fadd    meme
```

Les premières deux formes sont équivalentes. Elles elèvent les deux valeurs du sommet de la pile, les additionnent et placent le résultat dans la pile.

Les deux formes suivantes de l'instruction fadd, celles qui prennent les registres du FPU comment opérandes, se comportent comme l'instruction add du 80x86. Elles additionnent la valeur du second registre à celle du premier. Notez qu'une des opérandes registre doit être st(0)[9].

L'instruction faddp avec deux opérandes additionne une variables de 32 ou de 64 bits à la valeur dans st(0). Cette instruction convertit la valeur de 32 ou de 64 bits en une valeur de 80 bits à précision étendue avant d'effectuer l'addition. Notez que cette instruction ne permet pas directement une opérande de mémoire de 80 bits.

Ces instructions peuvent causer les exceptions: *stack*, *precision*, *underflow*, *overflow*, *denormalized* et *illegal operation*. Si une exception *stack* a lieu, alors le registre C_1 dénotera un dépassement de capacité soit de type *overflow,* soit de type *underflow*.

14.4.6.2 Les instructions FSUB, FSUBP, FSUBR, et SFUBRP

Ces deux instructions prennent les formes suivantes :

```
fsub
fsubp
fsubr
```

[9] Puisqu'en travaillant avec le 80x87 on utilise st(0) souvent, MASM permet l'abbréviation st. Cependant, ici nous utiliseront toujours la notation st(0) pour ne pas créer de confusion.

```
fsubrp
fsub    st(i). st(0)
fsub    st(0), st(i)
fsubp   st(i), st(0)
fsub    mem
fsubr   st(i). st(0)
fsubr   st(0), st(i)
fsubrp  st(i), st(0)
fsubr   mem
```

Sans opérandes, les instructions fsub et fsubp fonctionnent de façon identique. Elles désempilent st(0) et st(1) du registre de pile, calculent st(0) - st(1) et placent le résultat de retour dans la pile. Les instructions fsubr et fsubrp (soustraction inverse) fonctionnent de façon très semblable, sauf qu'elles calculent st(1) - st(0) et placent la différence dans la pile.

Avec deux registres comme opérandes (destination, source) l'instruction fsub calcule *destination := destination - source.* Un des deux registres doit être st(0). Avec deux registres comme opérandes, l'instruction fsubp calcule aussi *destination := destination - source,* mais elle enlève st(0) de la pile après avoir calculé la différence. Pour l'instruction fsubp aussi, l'opérande de source doit également être st(0).

La version à deux opérandes de fsubr et fsubrp, fonctionne de façon très similaire, sauf qu'elle calcule l'opération suivante *destination := source - destination*.

Les instructions fsub mem et fsubr mem acceptent une opérande de mémoire de 32 ou de 64 bits. Elles convertissent cette opérande en une valeur de 80 bits à précision étendue et soustraient cette valeur de st(0) (fsub) ou bien soustraient st(0) de cette valeur (fsubr) et gardent le résultat de retour dans st(0).

Ces instructions peuvent déclencher les exceptions stack, precision, underflow, overflow, denormalized et illegal, selon le cas. Si une exception de pile a lieu, C_1 indiquera un dépassement de pile (stack overflow ou underflow).

14.4.6.3 Les instructions FMUL et FMULP

Les instructions fmul et fmulp multiplient deux valeurs à virgule flottante. Ces instructions permettent les formes suivantes:

```
fmul
fmulp
fmul    st(0), st(i)
fmul    st(i), st(0)
fmul    mem
fmulp   st(i), st(0)
```

Sans opérandes, fmul et fmulp font la même chose: elles désempilent st(0) et st(1), elles multiplient ces valeurs et replacent le résultat dans la pile. L'instruction fmul avec deux registres comme opérandes effectue *destination := destination * source*. L'un des deux registres (source ou destination) doit être st(0). L'instruction fmul mem requiert une opérande de mémoire de 32 ou de 64 bits. Elle convertit cette variable en une valeur de 80 bits à précision étendue, puis elle multiplie st(0) par cette valeur.

Ces instruction peuvent déclencher les exceptions suivantes : stack, precision, underflow, overflow, denormalized et illegal operation. Si un arrondissement a lieu pendant le calcul, C_1 indiquera stack overflow ou underflow.

14.4.6.4 Les instructions FDIV, FDIVP, FDIVR et FDIVRP

Ces quatre instructions prennent les formes suivantes :

```
fdiv
fdivp
```

```
fdivr
fdivrp
fdiv    st(0), st(i)
fdiv    st(i), st(0)
fdivp   st(i), st(0)
fdivr   st(0), st(i)
fdivr   st(i), st(0)
fdivrp  st(i), st(0)
fdiv    mem
fdivr   mem
```

Avec zéro opérandes, les instructions fdiv et fdivp désempilent st(0) et st(1), calculent st(0)/st(1) et placent le résultat de nouveau dans la pile. Les instructions fdivr et fdivrp font la même chose, mais elles effectuent le calcul avant de placer le quotient dans la pile.

Avec deux opérandes de registre, ces instructions calculent les quotient suivants:

```
fdiv    st(0), st(i)    ;st(0) := st(0)/st(i)
fdiv    st(i), st(0)    ;st(i) := st(i)/st(0)
fdivp   st(i), st(0)    ;st(i) := st(i)/st(0)
fdivr   st(i), st(i)    ;st(0) := st(0)/st(i)
fdivrp  st(i), st(0)    ;st(i) := st(0)/st(i)
```

Les instructions fdivp et fdivrp enlèvent aussi st(0) de la pile après avoir effectué la division. La valeur pour *i* dans ces deux instructions est calculée avant d'enlever st(0).

Ces instructions peuvent déclencher les exceptions suivantes: stack, precision, underflow, overflow, denormalized, zero divide, et illegal operation. Si un arrondissement a lieu pendant le calcul, C_1 indiquera un dépassement de pile vers le bas ou vers le haut.

14.4.6.5 L'instruction FSQRT

La routine FSQRT n'accepte pas d'opérandes. Elle calcule la racine carrée de la valeur dans tos et remplace st(0) avec le résultat. La valeur dans tos doit être plus grande ou égale à zéro, sinon fsqrt générera une exception d'opération invalide. Cette instruction peut déclencher les exceptions d'opération suivantes: stack, précision, denormalized ou invalid, selon le cas. Si un arrondissement a lieu durant l'opération, C_1 indiquera un stack overflow ou underflow.

Exemple:

```
; Calcule Z := sqrt(x**2 + y**2);

                fld     x               ; Charge x.
                fld     st(0)           ; Fait une copie de x dans TOS.
                fmul                    ; Calcule x**2.

                fld     x               ; Charge y.
                fld     st(0)           ; Fait une copie de y dans TOS.
                fmul                    ; Calcule y**2.

                fadd                    ; Calcule x**2 + y**2.
                fsqrt                   ; Calcule sqrt(x**2 + y**2).
                fst     z               ; Stocke le résultat dans z
```

14.4.6.6 L'instruction FSCALE

L'instruction fscale enlève deux valeurs de la pile. Elle multiplie st(0) par la deuxième valeur et place le résultat dans la pile. Si la valeur dans st(1) n'est pas un entier, fscale la tranche tranche par défaut avant d'effectuer l'opération.

Cette instruction active l'exception de pile s'il n'y a pas deux valeurs dans la pile. Elle nettoie aussi le drapeau C_1 chaque fois qu'un stack underflow a lieu). Et elle active l'exception de précision s'il y a une perte de précision par suite de l'opération (cela arrive quand st(1) contient un grande valeur négative). De même, l'instruction affecte les exceptions

underflow et overflow si vous multipliez st(0) par une puissance de deux très grande, positive ou négative. Si le résultat de l'opération est très petit, fscale pourrait affecter le *denormalized bit*. Aussi cette instruction pourrait produire une erreur d'opération invalide si vous tentez d'affecter à fscale des valeurs invalides. fscale active C_1 si un arrondissement a lieu dans un calcul qui serait autrement correct. Exemple:

```
                fild Sixteen    ; Place sixteen dans la pile.
                fld x           ; Calcule x * (2**16).
                fscale
                .
                .
                .
                Sixteen word 16
```

14.4.6.7 Les instructions FPREM et FPREM1

Les instructions FPREM et PFREM1 calculent un *reste partiel*. Intel a conçu l'instruction fprem avant que IEEE finalisait son standard à virgule flottante. Lors de cette finalisation la définition de fprem était un peu différent par rapport à la définition originale de Intel. Qui plus est, Intel avait besoin de maintenir la compatibilité avec les logiciels existant qui utilisaient cette instruction, par conséquent on implémenta une nouvelle version de l'instruction, fprem1, pouvant calculer l'opération de reste partiel conformément au standard IEEE. En définitive vous devrez toujours utiliser fprem1 dans tout nouveau programme que vous écrivez. Pour cette même raison nous ne décrirons ici que fprem1, bien que vous utilisez fprem de façon identique.

fprem1 calcule le reste *partiel* de st(0)/st(1). Si la différence entre les exposants de st(0) et st(1) est moindre de 64, fprem1 peut calculer le reste exact en une seule opération. Sinon, il faudra exécuter fprem1 deux ou plus fois pour obtenir le reste correct. Le bit de condition C_2 détermine quand le calcul est complet. Notez que fprem1 n'enlève pas les deux opérandes de la pile ; elle laisse le reste partiel dans st(1) au cas où vous avez besoin de trouver un autre produit partiel pour compléter le résultat.

L'instruction fprem1 active le drapeau d'exception de pile s'il n'y a pas deux valeurs au sommet de la pile. Elle active les exceptions underflow et denormal si le résultat est trop petit. Et elle met à un le bit d'opération invalide si les valeurs dans le sommet de la pile sont inappropriées pour cette opération. Elle affacte le bit de condition C_2 si l'opération de reste partiel est incomplète. Et finalement, elle charge C_3, C_1 et C_0 avec les bits 0, 1 et 2 du quotient respectivement. Par exemple :

```
                ; Calculer Z := X mod Y
                fld     y
                fld     x
PartialLp:      fprem1
                fstsw   ax
                test    ah, 100b        ; Affecter les bits de condition dans AX
                jnz     PartialLp       ; Répéter si ce n'est pas complet
                fstp    Z               ; Stocker le reste quelquepart
                fstp    st(0)           ; Désempiler l'ancienne valeur de y
```

14.4.6.8 L'instruction FRNDINT

L'instruction frndint arrondit la valeur dans le sommet de la pile à l'entier le plus proche en utilisant l'algorithme d'arrondissement spécifié dans le registre de contrôle.

Cette instruction active le drapeau d'exception de pile s'il n'y trouve pas de valeur (et nettoie aussi C_1 dans ce cas). Elle met à 1 les bits d'exception *precision* et *denormalized* s'il arrive une perte de précision et elle active également le drapeau d'opération invalide si la valeur dans le sommet de la pile n'est pas un nombre valide.

14.4.6.9 L'instruction FXTRACT

L'instruction fxtract est le complément de l'instruction fscale. Elle enlève une valeur du sommet de la pile et y empile une valeur qui correspond à l'équivalent entier de son exposant (en forme réelle 80 bits), puis elle empile la mantissa avec un exposant de zéro (3fffh en forme biasée). Ceci est vrai surtout lorsqu'il y a un grand nombre d'entrées dans la table.

Cette instruction affecte l'exception de pile s'il se produit un *stack underflow* en désempilant la valeur originale ou s'il se produit un *stack overflow* quand on empile les deux résultats (C_1 détermine de quel type de dépassement de capacité il s'agit). Si le sommet de la pile était zéro, fxtract active le drapeau d'exception de division par zéro. Le drapeau *denormalized* est mis à 1 si le résultat le garantit ; et le drapeau d'opération invalide est activé s'il y a des valeurs d'entrée incorrectes en exécutant fxtract. Exemple:

```
; L'exemple suivant extrait l'exposant binaire de X et
; le stocke dans une variable entière de 16 bits Xponent.
        fld     x
        fxtract
        fstp    st(0)
        fistp   Xponent
```

14.4.6.10 L'instruction FABS

fabs calcule la valeur absolue de st(0) en nettoiant le bit de signe de st(0). Elle active l'exception de pile et les bits d'opérations invalides si la pile est vide.

Par exemple:

```
; Calcule X:= sqrt(abs(x));
        fld     x
        fabs
        fsqrt
        fstp    x
```

14.4.6.11 L'instruction FCHS

fchs change le signe de la valeur de st(0) en inversant le bit de signe. Elle active le bit l'exception de pile et celui d'opération invalide si la pile est vide. Par exemple:

```
; Calcule -X:= X si X est positif, X := X si X est négatif
        fld     x
        fabs
        fchs
        fstp    x
```

14.4.7 Instructions de comparaison

Le jeu 80x87 fournit diverses instructions pour comparer des valeurs à virgule flottante. Les instructions fcom, fcomp, fcompp, fucom, fucomp et fucompp comparent les deux valeurs au sommet de la pile et ajustent les codes de condition en conséquence. L'instruction ftst comprare la valeur au sommet de la pile avec zéro. L'instruction fxam vérifie la valeur du sommet de la pile et indique le signe, la normalisation et l'information de balise.

Généralement, la plupart des programmes testent le code de condition immédiatement après une comparaison. Malheureusement il n'y a pas d'instructions de saut conditionnel pour les codes de conditions du FPU. En revanche, vous pouvez utiliser l'instruction fstsw pour copier le registre d'état à virgule flottante (voir le paragraphe sur le registre d'état du FPU, paragraphe 14.4.1.3) dans le registre ax ; ensuite vous pouvez utiliser l'instruction sahf pour copier le registre ax dans le bits de code de condition du 80x86. Après avoir fait ceci, vous pouvez utiliser les sauts conditionnels pour tester certaines conditions. Cette technique copie C_0 dans le *carry flag*, C_2 dans le *parity flag* et C_3 dans le *zero flag*. L'instruction sahf ne copie C_1 dans aucun des bits drapeaux du 80x86.

Puisque l'instruction sahf ne copie aucun des bits d'état du 80x87 dans les drapeaux *sign* ou *overflow*, vous ne pouvez pas utiliser les instructions jg, jl ou jle. Utilisez plutôt les instructions ja, jae, jb, jbe, je et jz quand vous avez besoin de vérifier les résultats d'une comparaison à virgule flottante. *Oui, ces sauts conditionnels testent normalement des valeurs non signées, alors que les nombres à virgule flottante sont des valeurs signées*. Cependant, vous devriez utiliser les

instructions non signées de toute façon ; les instructions fstsw et sahf activent le registre flags du 80x86 pour utiliser les branchements non signés.

14.4.7.1 Les instructions FCOM, FCOMP et FCOMPP

Les instructions fcom, fcomp et fcompp comparent st(0) avec l'opérande spécifiée et activent le bit de condition 80x87 correspondant en fonction du résultat de l'opération. Les formes correctes de ces instructions sont :

```
fcom
fcomp

fcom    st(i)
fcomp   st(i)

fcom    mem
fcomp   mem
```

Sans opérandes, fcom, fcomp et fcompp comparent st(0) avec st(1) et activent les drapeaux du processeur de façon correspondante. En plus, fcomp désempile st(0) de la pile et fcompp désempile les deux st(0) et st(1).

Avec une opérande de registre, fcom et fcomp comparent st(0) avec le registre spécifié. fcomp désempile aussi st(0) après la comparaison.

Avec une opérande de mémoire de 32 ou de 64 bits, les instructions fcom et fcomp convertissent la variable de mémoire en une valeur de 80 bits à précision étendue et puis elles comparent st(0) avec cette valeur en activant les bits de conditions appropriés. fcomp désempile également st(0) après la comparaison.

Ces instructions affectent C_2 (et aussi le drapeau de parité) si les deux opérandes ne sont pas comparables (par exemple si elles ne sont pas des nombres). S'il est possible, pour une valeur à virgule flottante incorrecte, d'être comparée, alors il faudra vérifier le drapeau de parité en cas d'erreur avant de vérifier la condition désirée.

Ces instructions activent le bit de pile s'il n'y a pas deux objets dans le registre de pile. Elles affectent l'exception de dénormalisation si une opérande, ou les deux, ne sont pas normalisées. Et finalement, elles affectent le drapeau d'opération invalide si une ou les deux opérandes ne sont pas des nombres. Ces instructions nettoient toujours le code de condition C_1.

14.4.7.2 Les instructions FUCOM, FUCOMP et FUCOMPP

Ces instructions sont semblables à fcom, fcomp et fcompp, mais elles ne permettent que les formes suivantes:

```
fucom
fucomp
fucompp
fucom st(i)
fucomp st(i)
```

La différence entre fcom/fcomp/fcompp et fucom/fucomp/fucompp est relativement mineure. Les instructions fcom/fcomp/fcompp activent l'exception d'opération invalide si vous comparez deux NaN (not a number). Alors que fucom/fucomp/fucompp ne le font pas. Dans tous les autres cas, ces deux ensembles d'instructions sont identiques.

14.4.7.3 L'instruction FTST

Cette instruction compare la valeur dans st(0) avec 0.0. Elle se comporte exactement comme le ferait l'instruction fcom en comparant st(1) avec 0.0. Si la valeur dans st(0) correspond à une de ces valeurs, ftst activerait C_3 pour indiquer l'égalité. Si vous avez besoin de soustraire -0.0 de +0.0, utilisez plutôt l'instruction fxam. Notez que ftst ne désempile pas st(0).

14.4.7.4 L'instruction FXAM

Cette instructione examine la valeur dans st(0) et rapporte le résultat si les bits de condition de code (voir le paragraphe 14.4.1.3 sur le registre d'état du FPU pour avoir des détails sur comment fxam active ces bits). Cette instruction ne désempile pas st(0).

14.4.8 Instructions de constantes

Le FPU 80x87 fournit diverses instructions vous permettant de charger des constantes fréquemment utilisées dans la pile registre du FPU. Ces instructions activent les bits de stack fault, invalid operation et C1 s'il arrive un dépassement de capacité (overflow); hors de ce cas, elles n'affectent pas les drapeaux du FPU. Les instructions spécifiques de cette catégorie incluent:

```
fldz                    ; Empile +0.0
fld1                    ; Empile +1.0
fldpi                   ; Empile le nombre π
fldl2t                  ; Empile log2(10)
fldl2e                  ; Empile log2 (e)
fldlg2                  ; Empile log10 2
fldln2                  ; Empile ln(2)
```

14.4.9 Instructions transcendantales

Les 80387 et les FPU ultérieurs fournissent huit instructions transcendentales (log et trigonometriques) pour calculer une tangente partielle, une arc-tangente partielle, 2^x-1, y*$\log_2(x)$ et y*$\log_2(x+1)$. En utilisant plusieurs identités algebriques il est facile de calculer la plupart des autres fonctions transcendentales en utilisant ces instructions.

14.4.9.1 L'instruction F2XM1

Cette instruction calcule $2^{st(0)}$-1. La valeur dans st(0) doit être dans la plage -1.0 ≤ st(0) ≤ +1.0. Si st(0) est hors de cette plage, alors f2xm1 génère un résultat indéfini, mais elle n'affecte aucune exception. La valeur calculée remplace la valeur dans st(0). Par exemple:

```
; Calcule 10^x en utilisant l'identité: 10^x = 2^(x*lg(10))  (lg = log2).
fld     x
fldl2t
fmul
f2xml
fld1
fadd
```

Notez que f2xm1 calcule 2^x - 1, c'est pourquoi le code ci-dessus additionne 1.0 au résultat à la fin du calcul.

14.4.9.2 Les instructions FSIN, FCOS et FSINCOS

Ces instructions ôtent la valeur du sommet de la pile de registre et calculent le sinus, le cosinus ou les deux. Enfin, elles placent le(s) résultat(s) dans la pile. fsincos place dans la pile le sinus suivi par le cosinus de l'opérande originelle, donc, elle laisse cos(st(0)) dans st(0) et sin(st(0)) dans st(1).

Ces instructions présupposent que st(0) spécifie un angle en radians et cet angle doit être dans la plage -2^{63} < st(0) + 2^{63}. Si l'opérande originale est hors cette plage, alors ces instructions affectent le drapeau C_2 et laissent st(0) inchangé. Vous pouvez utiliser fprem1 avec un diviseur de 2π, pour réduire l'opérande à une plage raisonnable.

Ces instructions affectent le *stack fault* et C_1, les drapeaux de précision, de underflow, denormalized et opération invalide, selon le résultat du calcul.

14.4.9.3 L'instruction FPTAN

Cette instruction calcule la tangente dans st(0) et elle empile cette valeur, puis elle empile 1.0 dans la pile. Tout comme les instructions fsin et fcos, la valeur dans st(0) est censée être en radians et doit être dans la plage $-2^{63} < st(0) < +2^{63}$. Si la valeur est hors de cette range, alors fptan active C_2 pour indiquer que la conversion n'a pas eu lieu. Tout comme fsin, fcos et fsincos, vous pouvez utiliser fprem1 pour réduire l'opérande à une plage raisonnable à l'aide d'un diviseur 2π.

Si l'argument n'est pas valide (par exemple, 0 ou π radians qui causent une division par zéro), le résultat est indéfini et l'instruction n'affecte pas d'exception. fptan activera le drapeau stack fault, precision, underflow, denormal, invalid operation, C_2 et C_1, selon le résultat de l'opération.

14.4.9.4 L'instruction FPATAN

Cette instruction attend deux valeurs dans le sommet de la pile. Elle les désempile et calcule ce qui suit::

$$st(0) = \tan^{-1}(st(1)/st(0))$$

La valeur résultante est l'arctangente du rapport dans la pile exprimé en radians. Si vous avez une valeur dont vous voulez calculer la tangente, utilisez fld1 pour créer le rapport approprié, puis exécutez l'instruction fpatan.

Cette instruction affecte les drapeaux: stack fault/C_1, precision, underflow, denormal et invalid operation s'il se produit un problème pendant l'opértion. Elle affecte le code de condition C_1 si elle doit arrondir le résultat.

14.4.9.5 Les instructions FYL2X et FYL2XP1

Les instructions fyl2x et fyl2xp1 calculent st(1) * $\log_2$(st(0)) et st(1) * $\log_2$(st(0)+1), respectivement. fyl2x exige que st(0) soit plus grand que zéro, et fyl2xp1 requiert que st(0) soit dans la plage:

$$\left(-1-\left(\frac{\sqrt{2}}{2}\right)\right) < st(0) < \left(1-\left(\frac{\sqrt{2}}{2}\right)\right)$$

fyl2x est utile pour calculer des logarithmes en bases autres que deux, alors que fyl2xp1 est utile pour calculer des intérêts composés, en maintenant une précision maximale pendant le calcul.

fyl2x peut affecter *tous* les drapeaux d'exception. C_1 indique un arrondissement s'il n'y a pas d'autres erreurs, alors qu'il se produit un dépassement de capacité de pile si le bit de stack fault est activé.

L'instruction fyl2xp1 n'affecte pas les drapeaux overflow ou division par zéro. Ces exceptions se produisent quand st(0) est très petit ou proche de zéro. Puisque fly2xp1 additionne un à st(0) avant de calculer la fonction, cette condition n'est pratiquement jamais atteinte. fyl2xp1 affecte les autres drapeaux de façon identique que fyl2x.

14.4.10 Instructions variées

Le FPU 80x87 inclut plusieurs autres instructions qui contrôlent le FPU, qui synchronisent les opérations et qui vous permettent de tester divers bits d'état. Ces instructions comprennent finit/fninit, fdisi/fndisi, feni/fneni, fldcw, fstcw/fnstcw, fclex/fnclex, fsave/fnsave, frstor, frstpm, fstsw/fnstsw, fstenv/fnstenv, fldenv, fincstp, fdecstp, fwait, fnop et ffree. Les instructions fdis/fndis, feni/fneni et frstpm sont actives seulement dans les FPU antérieurs au 80387, donc nous n'allons pas les considérer ici.

Beaucoup de ces instructions ont deux formes. La première es fxxxx et la deuxième est fnxxxx. La version sans "n" produit une instruction fwait avant le opcode (ce qui est du standard pour la plupart des instructions du co-processeur). La version avec "n" ne produit pas d'instruction fwait (étant donné qu'"n" signifie *no wait*).

14.4.10.1 Les instructions FINIT et FNINIT

L'instruction finit initialize le FPU pour un fonctionnement correct. Vos applications devraient exécuter cette instruction avant d'exécuter toute autre instruction du FPU. Cette instruction initialize le registre de contrôle à 37Fh (voir "Le registre de contrôle du FPU" du paragraphe 14.4.1.2), le registre d'état à zéro (voir "Le registre d'état du FPU" du paragraphe 14.4.1.3) and le *tag word* à 0FFFFh. Les autres registres restent inaffectés.

14.4.10.2 L'instruction FWAIT

Cette instruction met le système en pause jusqu'à ce que n'importe quelle instruction couramment en exécution ne termine pas sa tâche. Ceci est requis parce que le FPU dans le 80486SX et dans les anciennes combinaisons CPU/FPU peuvent exécuter ces instructions en parallèle avec le CPU. Par conséquent, toute instruction FPU qui lit ou écrit la mémoire peut encourir en un conflit de données si le CPU est en train de lire le même emplacement de mémoire avant que le FPU lit ou écrit sur cet emplacement. L'instruction fwait vous permet de synchroniser l'opération du FPU en attendant jusqu'au accomplissement de l'instruction FPU courante. Ceci résout le conflit de donnée, en insérant un "stall" explicite dans le flux d'exécution.

14.4.10.3 Les instructions FLDCW et FSTCW

Les instructions fldcw et fstcw demandent une opérande de 16 bits:

```
fldcw   mem_16
fstcw   mem_16
```

Les deux instructions chargent le registre de contrôle (voir "Le registre de contrôle du FPU" du paragraphe 14.4.1.2) depuis un emplacement de mémoire (fldcw) ou elles stockent le mot de contrôle dans un emplacement de mémoire de 16 bits (fstcw).

Quand on utiliser l'instruction fldcw pour activer l'une des exceptions, si le drapeau de condition est activé quand vous habilitez l'exception, le FPU générera une interruption immédiate avant que le CPU exécuter la prochaine instruction. Par conséquent, vous devrez utiliser l'instruction fclex pour nettoyer toute interruption courante avant de changer les bits d'exception du FPU.

14.4.10.4 Les instructions FCLEX et FNCLEX

Les instructions fclex et fnclex nettoient tous les bits d'exceptions dans le bit de fault de la pile. Elles nettoient aussi le drapeau *busy* dans le régistre d'état du FPU (voir "Le registre d'état du FPU du paragraphe 14.4.1.2).

14.4.10.5 Les instructions FLDENV, FSTENV et FNSTENV

```
fstenv  mem_14b
fnstenv mem_14b
fldenv  mem_14b
```

Les instructions fstenv/fnstenv stockent un record d'environnement du CPU dans un emplacement de mémoire spécifié. En mode réel (le seul que ce livre considère), le recod prend la forme apparaissant à la figure 14.11.

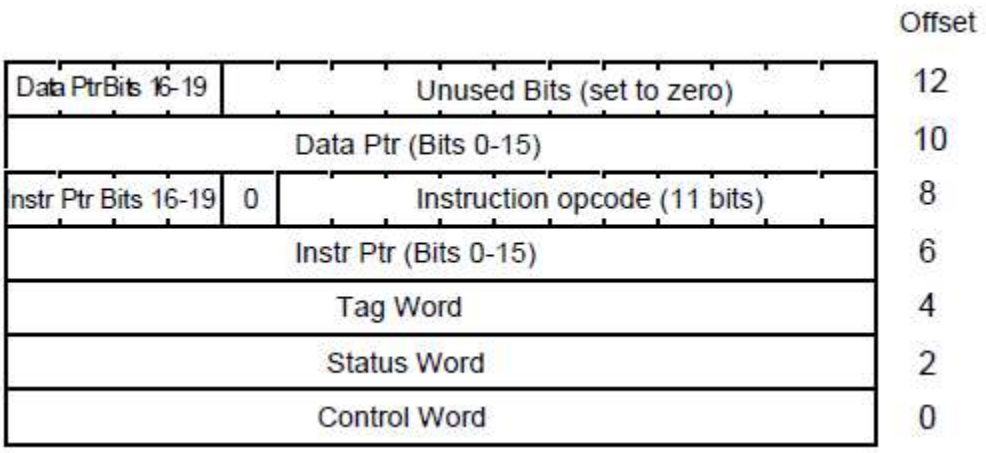

Figure 14.11 - Record d'environnement (en mode réel 16 bits)

Il faut exécuter les instructions fstenv et fnstenv avec les interrupteurs du CPU désactivés. De plus, vous devriez toujours vous assurer que le FPU n'est pas occupé avant d'exécuter cette instruction. Ce qui peut se faire facilement en utilisant le code suivant:

```
pushf                   ; Préserve le drapeau I
cli                     ; Désactive les interrupteurs
fstenv  mem_14b         ; Attente implicite pour not busy
fwait                   ; Attendre l'opération pour terminer
popf                    ; Restaurer le drapeau I
```

L'instruction fldenv charge l'environnement du FPU depuis l'opérande de mémoire spécifiée. Notez que cette instruction vous permet de charger le mot d'état. Il n'y a pas d'instruction explicite comme fldcw pour achever cela.

14.4.10.6 Les instructions FSAVE, FNSAVE et FRSTOR

```
fsave   mem_94b
fnsave  mem_94b
frstor  mem_94b
```

Ces instructions enregistrent et restaurent l'état du FPU. Ceci inclut enregistrer tous les registres internes de contrôle, état et données. L'emplacement de destination pour fsave/fnsave (qui sera l'emplacement source pour frstor) doit être longue 94 octets. Les premiers 14 octets correspondent aux enregistrements d'environnement que fldenv et fstenv utilisent. Les 80 octets restants maintiennent les données depuis la pile de registre du FPU écrite sous la forme st(0) jusqu'à st(7). frstor charge de nouveau l'enregistrement d'environnement et les registres de virgule flottante depuis l'opérande de mémoire spécifiée.

Les instructions fsave/fnsave et frstor servent principalement à permettre le passage d'une tâche à une autre (task switching). Vous pouvez également utiliser ces instructions en tant que séquence "push all" et "pop all" pour préserver l'état du FPU.

Tout comme avec les instructions fstenv et fldenv, les interruptions devraient être désactivés pendant l'enregistrement ou la remise en place de l'état du FPU. Sinon, une autre routine de service d'interruption pourrait manipuler les registres FPU et rendre invalides les opérations des instructions fsave/fnsave et frestore. Le code suivant protège d'une façon correcte les données d'environnement pendant la sauvegarde et la restauration de l'état du FPU:

```
; préserver l'état du FPU, en supposant que di pointe sur
; l'enregistrement d'environnement en mémoire
pushf
cli
fsave   [si]
```

```
fwait
popf
.
.
.
pushf
cli
frstor  [si]
fwait
popf
```

14.4.10.7 Les instructions FSTSW et FNSTSW

```
fstsw   ax
fnstsw  ax
fstsw   mem_16
fnstsw  mem_16
```

Ces instructions enregistrent le registre d'état du FPU (voir "Le registre d'état du FPU" au paragraphe 14.4.1.2) dans un emplacement de mémoire de 16 bits ou dans le registre ax. Ces instructions sont inusuelles en ce sens qu'elle peuvent copier une valeur du FPU dans un des registres généraux du 80x86. Bien sûr, tout le véritable but derrière le fait de permettre le transport du registre d'état dans ax est de permettre au CPU de tester facilement le registre de code de condition avec l'instruction sahf.

14.4.10.8 Les instructions FINCSTP et FDECSTP

Les instructions fincstp et fdecstp ne prennent pas d'opérandes. Elles incrémentent et décrementent simplement les bits de pointeur de pile (mod 8) dans le registre d'état du FPU. Ces deux instructions nettoient le drapeau C_1, mais elle n'affectent pas d'une autre manière les bits de condition de code dans le registre d'état du FPU.

14.4.10.9 L'instruction FNOP

Cette instruction est seulement un alias de fst st, st(0). Elle n'effectue pas d'autres opérations dans le FPU.

14.4.10.10 L'instruction FFREE

```
ffree   st(i)
```

Cette instruction modifie les tag bits pour le registre *i* dans le registre de tag pour marquer le registre spécifié comme vide. La valeur n'est pas modifiée par l'instruction, mais le FPU ne sera plus capable d'accéder cette donnée (sans réinitialiser correctement les tag bits).

14.4.11 Opérations d'entiers

Le FPU 80x87 fournit des instructions spéciales pour combiner les entiers avec opérations de conversion à précision étandue et également d'autres opérations arithmétiques et de comparaison. Ces instructions sont les suivantes:

```
fiadd int
fisub int
fisubr int
fimul int
fidiv int
fidivr int
ficom int
ficomp int
```

Elles convertissent leur opérandes de 16 ou de 32 bits dans une valeur de 80 bits en précision étendue et utilisent cette valeur en tant qu'opérande source de l'opération spécifiée. Ces instructions utilisent st(0) comme opérande de destination.

14.5 Programme d'exemple : fonctions trigonométriques supplémentaires

Cette section fournit plusieurs exemples de la programmation 80x87. Ce groupe de routines que vous allez voir fournissent diverses fonctions trigonométriques, trigonométriques inverses, logarithmiques et exponentielles à l'aide de diverses identités algébriques. Toutes ces fonctions s'attendent que la valeur d'entrée soit dans la pile et qu'elle soit aussi dans une plage valide pour une fonction donnée. Les routines trigonométriques s'attendent à des angles exprimés en radians, trandis que les routines trigonométriques inverses produisent des angles mesurés en radians.

Ce programme (transcnd.asm) est également disponible dans le répertoire de ce chapitre.

```
                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

                .386
                .387
                option segment:use16

dseg    segment para public 'data'
result  real8   ?

; Certaines variables qu'on va utiliser dans les routines de ce
; package:

cotvar          real8   3.0
cotRes          real8   ?
acotRes         real8   ?

cscvar          real8   1.5
cscRes          real8   ?
acscRes         real8   ?

secvar          real8   0.5
secRes          real8   ?
asecRes         real8   ?

sinvar          real8   0.75
sinRes          real8   ?
asinRes         real8   ?

cosvar          real8   0.25
cosRes          real8   ?
acosRes         real8   ?

Two2xvar        real8   -2.5
Two2xRes        real8   ?
lgxRes          real8   ?

Ten2xVar        real8   3.75
Ten2xRes        real8   ?
logRes          real8   ?

expVar          real8   3.25
expRes          real8   ?
lnRes           real8   ?

Y2Xx            real8   3.0
```

```
Y2Xy            real8   3.0
Y2XRes          real8   ?

dseg            ends

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

; COT(x) - Calcule la cotangente de st(0) en laisse le résultat
; dans st(0).
; st(0) contient x (en radians) et doit être dans la plage
; -2**63 et +2**63
;
; Il faut au moins un registre libre dans la pile pour que cette
; routine fonctionne correctement.
;
; cot(x) = 1/tan(x)
cot             proc    near
fsincos
fdivr
ret
cot             endp

; CSC(x) - calcule la cosécante de st(0) et laisse le résultat
; dans st(0).
; st(0) contient x (en radians) et doit être dans la plage
; -2**63 et +2**63.
; La cosécante de x est indéfinie pour toute valeur de sin(x) qui
; produit zéro (par exemple, zéro radians ou π radians).
;
; Il faut au moins un registre libre dans la pile pour que cette
; routine fonctionne correctement.
;
; csc(x) = 1/sin(x)

csc             proc    near
fsin
fld1
fdivr
ret
csc             endp

; SEC(x) - calcule la sécante de st(0) et laisse le résultat dans
; st(0), qui contient x (en radians) et doit être dans la plage
; -2**63 et +2**63.
;
; La sécante de x est indéfinie pour toute valeur de cos(x) qui
; produit zéro (par exemple, π/2 radians).
;
; Il faut au moins un registre libre dans la pile pour que cette
; routine fonctionne correctement.
;
; sec(x) = 1/cos(x)

sec             proc    near
fcos
fld1
fdivr
ret
sec             endp

; ASIN(x)- Calcule le arcsinus de st(0) en laissant le résultat dans
; st(0).
```

```
; Plage disponible: -1≤x≤+1
; Il faut au moins deux registres libres pour que cette
; routine fonctionne correctement.
;
; asin(x) = atan(sqrt(x*x/(1-x*x)))

asin            proc    near
fld             st(0)           ; fait une copie de x au sommet
                                ; de la pile.
fmul                            ; calcule X².
fld             st(0)           ; fait une copie de x² au sommet
                                ; de la pile.
fld1                            ; calcule 1-X².
fsubr
fdiv                            ; calcule X²/(1-X²).
fsqrt                           ; calcule sqrt(x²/(1-X²)).
fld1                            ; Obtient l'arctan complète.
fpatan                          ; Fait l'arctan de la valeur
                                ; ci-dessus.
ret
asin            endp

; ACOS(x)- Calcule le arcosinus de st(0) en laissant le résultat dans
; st(0).
; Plage disponible: -1<=x<=+1
; Il faut au moins deux registres libres pour que la
; routine fonctionne correctement.
;
; acos(x) = atan(sqrt((1-x*x)/(x*x)))

acos            proc    near
fld             st(0)           ; fait une copie de x
                                ; au sommet de la pile.
fmul                            ; Compute X².
fld             st(0)           ; fait une copie de x²
                                ; au sommet de la pile.
fld1                            ; calcule 1-X².
fsubr
fdiv                            ; calcule (1-x²)/X².
fsqrt                           ; calcule sqrt((1-X²)/X²).
fld1                            ; calcule l'arctan complète.
fpatan                          ; calcule l'arctan de la valeur
                                ; ci-dessus
ret
acos            endp

; ACOT(x) - Calcule le arccotangente de st(0) en laissant le résultat
; dans st(0).
; x ne peut pas valoir zéro.
; Il faut au moins deux registres libres
; pour que cette routine fonctionne correctement.
;
; acot(x) = atan(1/x)

acot            proc    near
fld1                    ; calcule fpatan
fxch                    ; atan(st(1)/st(0)).
fpatan                  ; on veut atan(st(0)/st(1)).
ret
acot            endp

; ACSC(x)- Calcule le arccosécante de st(0) en laissant le
; résultat dans st(0).
```

```
; abs(X) doit être plus grand que un.
; Il faut au moins deux registres libres pour que cette routine
; fonctionne correctement.
;
; acsc(x) = atan(sqrt(1/(x*x-1)))

acsc            proc    near
fld             st(0)           ; calcule x*x
fmul
fld1                            ; calcule x*x-1
fsub
fld1                            ; calcule 1/(x*x-1)
fdivr
fsqrt                           ; calcule sqrt(1/(x*x-1))
fld1
fpatan                          ; fait le arctan de la valeur
                                ; ci-dessus.
ret
acsc            endp

; ASEC(x)- Calcule le arcsécante de st(0) en laissant le
; résultat dans st(0).
; abs(X) doit être plus grand que un.
; Il faut au moins deux registres libres pour que cette
; routine fonctionne correctement.
;
; asec(x) = atan(sqrt(x*x-1))

asec            proc    near
fld             st(0)           ; calcule x*x
fmul

fld1                            ; calcule x*x-1
fsub
fsqrt                           ; calcule sqrt(x*x-1)
fld1
fpatan                          ; calcule le arctan de la valeur
                                ; ci-dessus.
ret
asec            endp

; TwoToX(x)- Calcule 2^x.
; Il fait cela par l'identité algébrique:
;
; 2^x = 2^int(x) * 2^frac(x).
; On peut facilement calculer 2^int(x) avec fscale et
; 2^frac(x) avec f2xm1.
;
; Cette routine a besoin de trois registres libres.

SaveCW          word    ?
MaskedCW        word    ?

TwoToX          proc    near
fstcw           cseg:SaveCW

; Modifier le mot de contrôle pour troncher lors de l'arrondissement.

fstcw           cseg:MaskedCW
or              byte ptr cseg:MaskedCW+1, 1100b
fldcw           cseg:MaskedCW

fld             st(0)           ; copie le sommet de la pile.
```

```
fld             st(0)
frndint                         ; calcule la portion entière.
fxch                            ; permute les valeurs.
fsub            st(0), st(1)    ; calcule la partie fractionnelle.
f2xm1                           ; calcule 2^frac(x)-1.
fld1
fadd                            ; calcule 2^frac(x).
fxch                            ; obtient la portion entière.
fld1                            ; calcule 1*2^int(x).
fscale
fstp            st(1)           ; supprime st(1) (qui vaut 1).
fmul                            ; calcule 2^int(x) * 2^frac(x).
fldcw           cseg:SaveCW     ; réétablit le mode arrondissement.
ret
TwoToX          endp

; TenToX(x)- Calcule 10^x.
;
; Cette routine demande trois registres libres.
;
; TenToX(x) = 2^(x * lg(10))

TenToX          proc    near
fldl2t                          ; place lg(10) dans la pile
fmul                            ; calcule x*lg(10)
call            TwoToX          ; calcule 2^(x * lg(10))
ret
TenToX          endp

; exp(x) - Calcule e^x.
;
; Cette routine demande trois registres libres.
;
; exp(x) = 2^(x * lg(e))

exp             proc    near
fldl2e                          ; place lg(e) dans la pile
fmul                            ; calcule x*lg(e)
call            TwoToX          ; calcule 2^(x * lg(e))
ret
exp             endp

; YtoX(y,x) - Calcule y^x (y=st(1), x=st(0))
;
; Cette routine demande trois registres libres.
;
; y doit être plus grand que zéro.
;
; YtoX(y,x) = 2^(x * lg(y))

YtoX            proc    near
fxch                            ; calcule lg(y)
fld1
fxch
fyl2x
fmul                            ; calcule x*lg(y)
call TwoToX                     ; calcule 2^(x*lg(y))
ret
YtoX            endp

; LOG(x) - Calcule la base 10 du logarithme de x.
;
; La plage usuelle pour x (>0).
```

```
;
; LOG(x) = lg(x)/lg(10).

log             proc    near
fld1
fxch
fyl2x                           ; calcule 1*lg(x)
fldl2t                          ; charge lg(10).
fdiv                            ; calcule lg(x)/lg(10)
ret
log             endp

; LN(x) - Calcule la base e du logarithme de x.
;
; x doit être plus grand que zéro.
;
; ln(x) = lg(x)/lg(e).

ln              proc    near
fld1
fxch
fyl2x                           ; calcule 1*lg(x)
fldl2e                          ; charge lg(e)
fdiv                            ; calcule lg(x)/lg(10)
ret
ln              endp

; Le programme principal teste les diverses fonctions de ce package.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                meminit
                finit

; Vérifier cot et acot.

                fld     cotVar
                call    cot
                fst     cotRes
                call    acot
                fstp    acotRes
                printff

byte    "x=%8.5gf, cot(x)=%8.5gf, acot(cot(x)) = %8.5gf\n",0
dword   cotVar, cotRes, acotRes

; Vérifier csc et acsc.
                fld     cscVar
                call    csc
                fst     cscRes
                call    acsc
                fstp    acscRes
                printff

byte    "x=%8.5gf, csc(x)=%8.5gf, acsc(csc(x)) = %8.5gf\n",0
dword   cscVar, cscRes, acscRes

; Vérifier sec et asec.
                fld     secVar
                call    sec
```

```
                fst     secRes
                call    asec
                fstp    asecRes
                printff

byte    "x=%8.5gf, sec(x)=%8.5gf, asec(sec(x)) = %8.5gf\n",0
dword   secVar, secRes, asecRes

; Vérifier sin et asin.
                fld     sinVar
                fsin
                fst     sinRes
                call    asin
                fstp    asinRes
                printff

byte    "x=%8.5gf, sin(x)=%8.5gf, asin(sin(x)) = %8.5gf\n",0
dword   sinVar, sinRes, asinRes

; Vérifier cos et acos.
                fld     cosVar
                fcos
                fst     cosRes
                call    acos
                fstp    acosRes
                printff

byte    "x=%8.5gf, cos(x)=%8.5gf, acos(cos(x)) = %8.5gf\n",0
dword   cosVar, cosRes, acosRes

; Vérifier 2^x et lg(x).
                fld     Two2xVar
                call    TwoToX
                fst     Two2xRes
                fld1
                fxch
                fyl2x
                fstp    lgxRes
                printff

byte    "x=%8.5gf, 2**x =%8.5gf, lg(2**x) = %8.5gf\n",0
dword   Two2xVar, Two2xRes, lgxRes

; Vérifier 10^x et l0g(x).
                fld     Ten2xVar
                call    TenToX
                fst     Ten2xRes
                call    LOG
                fstp    logRes
                printff

byte    "x=%8.5gf, 10**x =%8.2gf, log(10**x) = %8.5gf\n",0
dword   Ten2xVar, Ten2xRes, logRes

; Vérifier exp(x) et ln(x).
                fld     expVar
                call    exp
                fst     expRes
                call    ln
                fstp    lnRes
                printff

byte    "x=%8.5gf, e**x =%8.2gf, ln(e**x) = %8.5gf\n",0
```

```
dword   expVar, expRes, lnRes

; Vérifier y^x.
                fld     Y2Xy
                fld     Y2Xx
                call    YtoX
                fstp    Y2XRes
                printff

byte    "x=%8.5gf, y =%8.5gf, y**x = %8.4gf\n",0
dword   Y2Xx, Y2Xy, Y2XRes

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends
end             Main

La sortie devraît être:
x= 3.00000, cot(x)=-7.01525, acot(cot(x)) = 3.00000
x= 1.50000, csc(x)= 1.00251, acsc(csc(x)) = 1.50000
x= 0.50000, sec(x)= 1.13949, asec(sec(x)) = 0.50000
x= 0.75000, sin(x)= 0.68163, asin(sin(x)) = 0.75000
x= 0.25000, cos(x)= 0.96891, acos(cos(x)) = 0.25000
x=-2.50000, 2^x = 0.17677, lg(2^x) = -2.50000
x= 3.75000, 10^x = 5623.41, log(10^x) = 3.75000
x= 3.25000, e^x = 25.79, ln(e^x) = 3.25000
x= 3.00000, y = 3.00000, y^x = 27.0000
```

14.6 Exercices de laboratoire

14.6.1 Précision entre FPU et StdLib

Dans cet exercice de laboratoire vous allez exécuter deux programmes qui effectuent 20 000 000 additions à virgule flottante. Ces programmes effectueront les premières 10 000 000 additions en utilisant le jeu 80x87 du FPU, et les restantes en utilisant les routines de la bibliothèque standard. Cet exercice démontre la précision relative des deux mécanismes.

Pour votre rapport de laboratoire : assemblez et exécutez le programme EX14_1.asm qui se trouve dans le répertoire qui accompagne ce chapitre. Ce programme additionne 10 000 000 valeurs à virgule flottante de 64 bits et affiche le résultat. Décrivez-le dans votre rapport. Egalement, mesurez le temps employé et notez la différence dans votre rapport. Notez que la somme *exacte* de ces opérations devrait produire $1{,}00000010000_e{+}0000$.

Après avoir exécuté le programme EX14_1.asm, exécutez le programme EX14_2.asm, qui fait la même chose mais en utilisant les routines de la bibliothèque standard qui utilise des opérandes de mémoire de 80 bits (le FPU ne peut pas utiliser des opérandes de mémoire de 80 bits). Mesurez encore le temps d'exécution des deux components de l'exercice EX14_2.asm, puis comparez-le avec les performances de l'exercice EX14_1.asm et expliquez les différences possibles.

Nous reportons ici les exercices EX14_1.asm et EX14_2.asm en guise d'exemple:

```
; EX14_1.asm
```

```
;
; Ce programme exécute certains tests pour déterminer comment
; l'arithmétique à virgule flottante de la bibliothèque standard
; fonctionne par rapport à celle du 80x87. Il fait ceci en effectuant
; diverses opérations en utilisant les deux méthodes et en comparant
; les résultats.
;
; Évidemment, il faut disposer d'un FPU 80x87 (ce qui correspond à un
; processeur 80486 ou ultérieur) afin d'exécuter ce code.

.386
option          segment:use16
include         stdlib.a
includelib      stdlib.lib
dseg            segment para public 'data'

; Puisque ceci est un test de précision, ce code utilise des valeurs
; REAL8 pour toutes les opérations

slValue1        real8   1.0
slSmallVal      real8   1.0e-14
Value1          real8   1.0
SmallVal        real8   1.0e-14
Buffer          byte    20 dup (0)

dseg            ends

cseg            segment para public 'code'
assume          cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit
                finit                   ; Initialiser le FPU

; Effectuer les 10.000.000 additions à virgule flottante:

                printff
                byte    "Additionner les 10.000.000 valeurs en utilisant"
                byte    "le FPU",cr,lf,0

                mov     ecx, 10000000
FPLoop:         fld     Value1
                fld     SmallVal
                fadd
                fstp    Value1
                dec     ecx
                jnz     FPLoop
                printff

byte            "Résultat = %20GE\n",cr,lf,0
                dword   Value1

; Effectuer les 10.000.000 additions à virgule flottante en utilisant
; la routine de la bibliothèque standard fpadd

                printff
                byte    cr,lf
                byte    "Additionner valeurs 10.000.000 avec "
                byte    "StdLib", cr,lf
                byte    "A noter: ceci peut tarder un peu... "
                byte    "veuillez patienter"
                byte    cr,lf,0

                mov     ecx, 10000000
```

```
SLLoop:         lesi    slValue1
                ldfpa
                lesi    slSmallVal
                ldfpo
                fpadd
                lesi    slValue1
                sdfpa
                dec     ecx
                jnz     SLLoop
                printff

                byte    "Résultat = %20GE\n",cr,lf,0
                dword   slValue1
Quit:           ExitPgm                 ; macro du DOS pour quitter.

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db 16 dup (?)
zzzzzzseg       ends

end Main

; EX14_2.asm
;
; Ce programme exécute certains tests pour déterminer comment
; l'arithmétique à virgule flottante de la bibliothèque standard
; fonctionne par rapport à celle du 80x87. Cette fois il le fait
; en utilisant le format 80 bits, lequel peut être utilisé
; seulement par la bibliothèque standard.
;
; Évidemment il faut utiliser un FPU 80x87 (ou un processeur 80486
; ou ultérieur) pour pouvoir exécuter ce code.

.386
option          segment:use16
include         stdlib.a
includelib      stdlib.lib

dseg            segment para public 'data'
slValue1        real10 1.0
slSmallVal      real10 1.0e-14
Value1          real8 1.0
SmallVal        real8 1.0e-14
Buffer          byte 20 dup (0)
dseg            ends

cseg            segment para public 'code'
assume          cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit
                finit                   ; initialiser le FPU

; Effectuer les 10.000.000 additions à virgule flottante:
                printff
                byte    "Additionner 10.000.000 valeurs avec le "
                byte    "FPU",cr,lf,0
```

```
                mov     ecx, 10000000
FPLoop:         fld     Value1
                fld     SmallVal
                fadd
                fstp    Value1
                dec     ecx
                jnz     FPLoop
                printff

                byte    "Résultat = %20GE\n",cr,lf,0
                dword   Value1

; Effectuer les 10.000.000 additions en utilisant la routine de la
; bibliothèque standard fpadd

printff
                byte    cr,lf
                byte    "Additionner 10,000,000 valeurs en utilisant "
                byte    "StdLib", cr,lf
                byte    "A noter: ceci peut tarder un peu... "
                byte    "veuillez patienter."
                byte    cr,lf,0
                mov     ecx, 10000000
SLLoop:         lesi    slValue1
                lefpa
                lesi    slSmallVal
                lefpo
                fpadd
                lesi    slValue1
                sefpa
                dec     ecx
                jnz     SLLoop
                printff
                byte    "Résultat = %20LE\n",cr,lf,0
                dword   slValue1
Quit:           ExitPgm                 ;Macro du DOS pour quitter.

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db 16 dup (?)
zzzzzzseg       ends

end Main
```

14.7 Projets de programmation

Pas de projets pour ce chapitre.

14.8 Résumé

Pour beaucoup d'applications l'arithmétique entière a deux inconvénients insormontables : il n'est pas possible d'y représenter des valeurs fractionnelles et les entièrs ont une plage dynamique limitée. L'arithmétique à virgule flottante donne une approximation de l'arithmétique réelle qui dépasse ces deux inconvénients.

Mais l'arithmétique à virgule flottante ne vient pas sans ses propres problèmes. En fait sa précision est *limitée* et, par ce fait, des imprécisions peuvent entrer en ligne de compte dans les calculs. Par conséquent l'arithmétique à virgole flottante ne suit pas complètement les règles algébriques normales. Il y a cinq règles très importantes à garder à l'esprit avec celle-ci : (1) l'ordre d'évaluation peut affecter la précision du résultat ; (2) en additionnant des nombres de signe différent ou en soustrayant des nombres de même signe l'accurance peut être mineure que la précision fournie par le format à virgule flottante ; (3) en effectuant une chaîne ce calculs comprenant l'addition, la soustraction, la multiplication et la division, essayez d'effectuer la multiplication et la division en premier ; (5) en multipliant et divisant des valeurs, essayer de multiplier les nombres grands et les nombres petits ensembles d'abord et diviser les nombres avec la même grandeur d'abord ; (5) en comparant deux nombres à virgule flottante, gardez toujours à l'esprit que des erreurs peuvent s'introduire dans les calculs, par conséquent il faut vérifier si une valeur se trouve dans une certaine plage de l'autre. Pour plus d'informations, voir

• "La mathématique de l'arithmétique à virgule flottante " (14.1)

Chez Intel, on a individué très tôt que pour les nombres à virgule flottante il fallait un processeur *ad hoc*. À l'aide de trois mathématiciens ils ont mis au point la famille 80x87 des FPUs. Ces formats, avec quelques modifications mineures, sont devenus les IEEE 754 et IEEE854 standars à virgule flottante. Ce standard comprend trois formats différents : un format de 32 bits de précision standard, un format 64 bits de précision double et un format étendu de 80 bits[10]. Le format à 32 bits utilise une mantisse de 24 bits (où le bit fort aussi bien qu'un un implicite ne fait pas partie de ces 32 bits), un biais 127 de 8 bits pour l'excès de exposant et un bit de signe. Le format de 64 bits requiert une mantisse de 53 bits (une fois encore le bit fort et l'un implicite ne sont pas compris), un excès 1023 de 11 bits pur l'excès de exposant et un bit de signe. Et, finalement, le format à 80 bits utilise un exposant de 64 bits, un excès 16363 de 15 bits d'exposant et un bit de signe. Pour plus d'informations, voir:

• "Les formtats à virgule flottante IEEE" (14.2)

Bien que les CPUs avec les co-processeurs mathématiques intégrés sont très communs, il est toujours possible d'exécuter du code qui utilise l'arithmétique à virgule flottante sur une machine qui n'a pas de FPU. Dans ces cas il faut fournir des routines logicielles pour pouvoir effectuer ces opérations. Heureusement, la bibliothèque UCR standars fournit un ensemble de routines d'arithmétique flottante qu'on peut appeler. Cette bibliothèque inclut des routines qui chargent et stockent des valeurs à virgule flottante, qui convertissent entre des formats entiers et des formats à virgule flottante et qui effectuent les quatre opérations fondamentales sur des nombres à virgule flottante. Même si vous avez un FPU installé, les routines de conversion et de sortie de la bibliothèque sont encore très utiles. Pour plus d'informations voir :

• "Les routines à virgule flottante de la bibliothèque UCR standard" (14.3)

Pour une arithmétique à virgule flottante rapide le logiciel ne peut pas se comparer avec le matériel. C'est pour cela que le FPU 80x87 fournit la possibilité d'effectuer des opérations à virgule flottante rapides et convénientes, en étendant le jeu 80x86 pour venir à l'encontre de l'arithmétique à virgule flottante. Mais à part les nouvelles instructions, le FPU 80x87 fournit également huit nouveaux registres de données, un registre de contrôle, un registre d'état et divers autres registres internes. À différence des registres 80x86, les registres du FPU sont organisés en tant que pile. Même s'il est possible de manipuler les registres de façon à fonctionner commes des registres ordinaires, la plupart des applications qui utilisent le FPU se servent du mécanisme de pile en calculant les résultats à virgule flottante. Le registre de contrôle du FPU vous permet d'initilaliser le FPU 80x87 en plusieurs modes différents. Le registre de contrôle permet de contrôler l'arrondissement, la précision disponible durant un calcul et de choisir quelles exceptions peuvent causer une interruption. Le registre d'état du 80x87 rapporte l'état courant du FPU. Ce registre fournit des bits qui déterminent si le FPU est occupé, si une instruction précédente a généré une exception, il détermine le numéro de registre physique au sommet de la pile et fournit les codes de condition du FPU. Pour plus d'informations sur le jeu de registres du 80x87 voir :

- "Le coprocesseur 80x87 à virgule flottante" (14.4)
- "Les registres du FPU" (14.4.1)
- "Les registres de données du FPU" (14.4.1.1)
- "Le registre de contrôle du FPU" (14.4.1.2)
- "Le registre d'état du FPU " (14.4.1.3)

[10] Le standars IEEE requiert que le format de précision étendue soit majeur que celui de double précision.

En plus des types de données simple, double et étendu, le FPU du 80x87 supporte aussi plusieurs types de données entiers et BCD. Le FPU convertit automatiquement ces types de données quand on charge et on stocke ces valeurs. Pour plus d'informations sur ces types de données, voir

• "Types de données du FPU" (14.4.2)

Le FPU du 80x87 fournit un vaste éventail d'opérations à virgule flottante en augmentant le jeu d'instruction s80x86. On peut classifier les instructions du FPU en huit catégories : instructions de mouvement des données, instructions de conversion, instructions arithmétiques, instructions de comparaison, instructions constantes, instructions transcendentales, instructions diverses et instructions d'entiers. Pour plus d'informations sur ces types d'instructions voir

• "Le jeu d'instructions du FPU" (14.4.3)
• "Instructions de mouvement de données" (14.4.4)
• "Conversions" (14.4.5)
• "Instructions arithmétiques" (14.4.6)
• "Instructions de comparaison" (14.4.7)
• "Instructions de constantes" (14.4.8)
• "Instructions transcendentales" (14.4.9)
• "Instructions variées" (14.4.10)
• "Opérations d'entiers" (14.4.11)

Bien que les CPU 80387 et ultérieurs fournissent un jeu riche de fonctions transcendentales, il y a diverses fonctions trigonométriques, trigonométriques inverses, exponentielles et logarithmiques qui ne sont pas fournies. Cependant ces fonctions peuvent parfaitement être réalisées à l'aide d'identités algébriques. Ce chapitre fournit le code source pour plusieurs de ces routines comme exemple de programmation FPU. Pour plus d'informations, voir

• "Programme d'exemple: fonctions trigonométriques supplémentaires" (14.5)

14.9 Questions

1. Pourquoi les règles normales de l'algèbre ne s'appliquent-elles pas dans l'arithmétique à virgule flottante ?
2. Donnez un exemple de séquence d'opérations dont l'ordre d'évaluation produit des résultats différents avec l'arithmétique à précision finie.
3. Expliquez pourquoi des opérations d'addition et de soustraction en précision limitée peuvent causer une perte de précision lors d'un calcul.
4. Pourquoi devriez-vous, autant que possible, effectuer les multiplications et les divisions en premier lors d'un calcul impliquant la multiplication ou la division, de même que l'addition ou la soustraction ?
5. Expliquez la différence entre des valeurs à virgule flottante normalisées, non-normalisées et dénormalisées.
6. A l'aide de la Bibliothèque Standard UCR, convertissez les expressions suivantes en code assembleur 80x86 (estimez que toutes les variables sont des valeurs en double précision 64 bits). Assurez-vous d'exécuter toutes les manipulations algébriques de façon à assurer la précision maximum. Partez du principe que toutes les variables se situent dans la plage $\pm 1^{E-10} ... \pm 1^{E+10}$.

   a) Z := X*X+Y*Y
   b) Z := (X-Y)*Z
   c) Z := X*Y-Y/Y
   d) Z := (X+Y)/(X-Y)
   e) Z := (X*X)/(Y*Y)
   f) Z := X*X+Y+1.0

7. Convertissez les instructions précédentes en code FPU 80x87.
8. Les problèmes suivants fournissent des définitions pour les fonctions *trigonométriques hyperboliques.* Codez chacune d'elles au moyen des instructions FPU 80x87 et des routines exp(x) et ln(x) fournies dans ce chapitre.

   a) $\sinh x = \frac{e^x - e^{-x}}{2}$
   b) $\cosh x = \frac{e^x + e^{-x}}{2}$
   c) $\tanh x = \frac{\sinh x}{\cosh x}$
   d) $\operatorname{csch} x = \frac{1}{\sinh x}$
   e) $\operatorname{sech} x = \frac{1}{\cosh x}$
   f) $\coth x = \frac{\cosh x}{\sinh x}$
   g) $\operatorname{asinh} x = ln(x + \sqrt{x^2 + 1})$
   h) $\operatorname{acosh} x = ln(x + \sqrt{x^2 - 1})$
   i) $\operatorname{atanh} x = \frac{ln(\frac{1+x}{1-x})}{2}$
   j) $\operatorname{acsch} x = ln(\frac{x \pm \sqrt{1+x^2}}{x})$
   k) $\operatorname{asech} x = ln(\frac{x \pm \sqrt{1-x^2}}{x})$
   l) $\operatorname{atanh} x = \frac{ln(\frac{x+1}{x-1})}{2}$

9. Créez une fonction log(x,y) qui calcule $\log_y x$. L'identité algébrique en est :

$$\log_y x = \frac{\log_2 x}{\log_2 y}$$

10. L'intervalle arithmétique implique un calcul dont chaque résultat est arrondi vers le bas, puis le même calcul en arrondissant chaque résultat vers le haut. À la sortie de ces deux calculs, vous savez que le résultat doit se

situer entre ces deux calculs. Les bits de contrôle d'arrondissement du registre de contrôle du FPU vous permettent de passer d'un mode d'arrondissement à l'autre. Refaitesz la question 6 en appliquant l'intervalle arithmétique et calculez les limites pour chacun des problèmes (a-f).

11. Les bits de contrôle de précision de mantisse du registre de contrôle du FPU contrôlent simplement l'endroit où le FPU arrondit le résultat. Choisir une précision moindre n'accélère pas le fonctionnement du FPU. C'est pourquoi vous devriez mettre ces deux bits à 1 pour obtenir une précision de 64 bits pour vos calculs. Pouvez-vous fournir une raison pour laquelle vous pourriez vouloir définir la précision sur autre chose que 64 bits ?

12. Supposez que vous avez deux variables, X et Y, que vous voulez comparer pour voir si elles sont égales. Comme vous l'avez appris, vous ne devez pas les comparer directement, mais plutôt, vous devriez comparer si elles différent de moins d'une certaine valeur. Supposez que ε, la constante d'erreur, est 1^{E-300}. Fournissez le code qui charge ax avec zéro si X=Y et avec un si X<>Y.

13. Réfaites le problème 12, en testant pour:

    a) X<=Y b) X<Y
    c) X>=Y d) X>Y e)X<>Y

14. Quelle instruction pouvez-vous utiliser pour voir si la valeur dans st(0) est dénormalisée ?

15. Dans l'hypothèse où il n'y a aucun dépassement, par défaut ou par excès, à quoi sert habituellement le bit de code de condition C_1 ?

16. De nombreux livres sur le FPU, suggèrent son usage aussi pour l'arithmétique entière. L'argument généralement avancé est que le FPU supporte des entiers de 64 bits au lieu de 16 ou 32 bits. Qu'est-ce qui cloche avec cet argument ? Pourquoi *refuseriez-vous* d'utiliser le FPU pour exécuter de l'arithmétique entière ? Pourquoi le FPU lui-même fournit-il des instructions pour entiers ?

17. Supposez que vous avez une valeur en virgule flottante double précision de 64 bits en mémoire. Décrivez comment vous pouvez obtenir la valeur absolue de cette variable sans utiliser le FPU (c.-à-d., en n'utilisant que les instructions 80x86).

18. Expliquez comment changer le signe de la variable de la question 17.

19. Pourquoi la fonction TwoToX (voir "Programme d'exemple : fonctions trigonométriques supplémentaires" (14.5)) doit-elle calculer le résultat en utilisant **fscale** et **fyl2x** ? Pourquoi ne peut-on pas utiliser que **fyl2x** ?

20. Expliquez un problème possible avec la séquence de code suivante :

```
stp     mem_64
xor     byte ptr mem_64+7, 80h      ;change bit de signe
```