CIRC2 : Assembleur, Numération, et Circuits

Licence informatique – semestre 4

Section 1

Représentation des nombres flottants

Représentation des nombres flottants - Norme IEEE 754

Rappels :

- Les nombres réels sont approximés par une représentation de nombres à virgule.
- Représentation : $mantisse \times 10^{exposant}$
- Exemple : $1,23475.10^3$
- Dans le cas général, on écrit :

$$X = (-1)^s M.b^E$$

où

- s donne le signe du nombre
- M est la mantisse sur p chiffres en base b : $x_0, x_1 x_2 x_3 ... x_{p-1}$ (et $x_0 \neq 0$)
- E est l'exposant

Le nombre flottant X est de précision p.

Conversion en binaire d'un nombre à virgule

Conversion de `-1234,75` - Codage de la mantisse

- la partie entière : conversion habituelle, divisions par 2 successives et on note le reste des divisions
  résultat `10011010010`
- la partie décimale : on passe dans les puissances négatives de 2 - multiplications par 2 successives et on note la partie entière, et on recommence sur la partie décimale
  $0,75 \times 2 = 1,5 \Rightarrow 1$
  $0,5 \times 2 = 1,0 \Rightarrow 1$
  et arrêt
- la mantisse est `1001101001011`.
- Le premier bit devant être $\neq 0$ (norme IEEE754), il ne sera pas gardé (1 implicite) $\Rightarrow$ mantisse `001101001011`

Simple précision, double précision, précision étendue

Format	Nombre de bits	Signe	Exposant	Mantisse
Simple précision	32	1	8	23 (+ 1 bit implicite)
Double précision	64	1	11	52 (+ 1 bit implicite)
Double précision étendue	80	1	15	64 (+ 1 bit implicite)

Approximation d'un nombre à virgule

La représentation en nombre à virgule flottante ne permet pas de représenter :

- les nombres trop grand ou trop petits, i.e. au-delà de la plus grande valeur de l'exposant (en simple précision, supérieurs à 2^{127} ou inférieurs à -2^{127})
- les nombres trop proches de zéro (en simple précision, les nombres compris dans $[-2^{-126-22}, 2^{-126-22}]$)
- les nombres très grands avec une précision très petite, du fait de la précision limitée au nombre de bits de la mantisse. Si 2^{60} est converti en flottant, $2^{60} + 1$ sera approximé à 2^{60} (double précision, mantisse à 53 bits).

Conversion en binaire d'un nombre à virgule (2)

Codage de l'exposant

- L'exposant peut être négatif (pour les valeurs inférieures à 0)
- Le codage utilisé n'est pas le complément à 2, mais la forme biaisée : si l'exposant est codé sur N bits, et si E est la valeur codée dans la place réservée à l'exposant, alors la valeur de l'exposant est $E - (2^{N-1} - 1)$
- Si l'exposant est codé sur N bits :
  - les valeurs $2^N - 1$ et 0 sont réservées
  - les valeurs possibles sont donc dans l'intervalle $[1; 2^N - 2]$
  - qui est à interpréter comme l'intervalle $[2 - 2^{N-1}; 2^{N-1} - 1]$

Les exposants prennent leur valeur

- En simple précision, dans $[-126; 127]$
- En double précision, dans $[-1022; 1023]$

Conversion en binaire d'un nombre à virgule (3)

Conversion de `-1234,75` en simple précision

- Signe : `1`
- Exposant : la mantisse est à lire `1,001101001011` alors que les 10 premiers bits derrrière la virgule codent la partie entière. L'exposant doit donc être égal à `10`, soit $10 + 127 = 137$ soit `10001001`
- Mantisse : `00110100101100000000000`

Le résultat est donc

`1 10001001 00110100101100000000000`

Valeurs possibles

Tableau des valeurs pour une précision simple :

Nombre	**Signe** (1 bit)	**Exposant** (8 bits)	**Mantisse** (23 bits)
+/-0	0/1	0	0
dénormalisé	0/1	0	0b0,xxx...xxx
normalisé>0	0	[1;254]	0b1,xxx...xxx
normalisé<0	1	[1;254]	0b1,xxx...xxx
+/-∞	0/1	255	0
+/-NaN (not a number)	0/1	255	≠ 0

Section 2

Calculer avec des nombres flottants

Directives pour les flottants en assembleur

- `.float`, `.single` : directives pour réserver de l'espace pour des flottants en simple précision (32 bits)
- `.double` : directive pour réserver de l'espace pour des flottants en double précision (64 bits)

```
1         .data
2 petit_pi:
3         .float  3.1416
4 notes:  .double 15.5, 16.8, 13.2
5 nb_notes:
6         .int    .-notes
```

Co-processeur FPU

- Le FPU (Floating Point Unit) est le co-processeur mathématique dédié aux calculs sur les flottants ;
- Le FPU possède 8 registres de 80 bits ;
- Ces registres sont organisés comme une pile de registre `%st(0), %st(1), ...%st(7)` ;
- `%st(0)` est le sommet de la pile.

Les suffixes habituels dans les instructions assembleur :

- `s` : pour un flottant 32 bits
- `l` : pour un flottant 64 bits
- `t` : pour un flottant 80 bits

Charger un flottant dans la pile

Pour transférer une valeur flottante de la mémoire vers la pile des registres, ou entre les registres

- `FLD src` : *float load* - empile `src` dans `%st(0)`.
- `src` peut être une adresse mémoire ou un registre.
- Le contenu de `src` est converti en précision étendue (80 bits).
- `FILD src` : si `src` est un entier. Le convertit en flottant 80 bits avant de l'empiler.

Charger un flottant dans la pile (2)

```
1  FLDL   notes   # empile notes (converti en 80 bits)
2                 # dans %st(0)
3
4  FLDT   %st(3)  # la valeur dans %st(3) est copiee
5                 # dans %st(0) - la meme valeur est
6                 # maintenant dans %st(0) et %st(4)
7
8  FLDT   %st(0)  # duplique la valeur au sommet
9                 # qui est maintenant dans  %st(0) et %st(1)
10
11 FLDT   %st(7)  # erreur ! %st(7) doit toujours etre vide
12                # avant d'empiler
13
14 FILDL  nb_notes
15                # empile l'entier nb_notes apres
16                # l'avoir converti en flottant
```

Empiler une valeur prédéfinie

Sept valeurs sont codées dans le hard et peuvent être chargées dans la pile :

- `FLDZ` : empile 0
- `FLD1` : empile 1
- `FLDPI` : empile Pi
- `FLDL2E` : empile $\log_2(e)$
- `FLDL2T` : empile $\log_2(10)$
- `FLDLG2` : empile $\log_{10}(2)$
- `FLDLN2` : empile $\ln(2)$

Stocker un flottant

Pour transférer une valeur flottante de la pile des registres vers la mémoire ou vers un autre registre

- `FST dest` : *float store* - copie `%st(0)` dans `dest`.
- `dest` peut être l'adresse mémoire d'un `double` ou d'un `float` ou un registre (qui est alors dupliqué dans `%st(0)`).
- en aucun cas `%st(0)` ne peut être vide au moment de l'appel à `FST`.
- `FSTP dest` : *float store and pop*. Comme `FST` mais dépile la valeur.

Stocker un flottant (2)

```
1 FSTPL notes  # mets la valeur contenue dans %st(0)
2              # dans notes, puis depile;
3
4 FSTPL  %st(3) # la valeur de %st(0) est copiee
5              # dans %st(3) puis depilee -
6              # elle se trouve maintenant dans %st(2)
```

Instructions de calcul

Les instructions `FADD, FMUL, FSUB, FDIV src, dest`

- pour l'addition, la multiplication, la soustraction, la division
- avec deux opérandes, dont au moins un registre de la pile (et une adresse mémoire ou un autre registre de la pile)
- si une seule opérande, le sommet de la pile est implicitement le deuxième opérande
- `FIADD, FIMUL, FISUB, FIDIV` si la première opérande est un entier
- on peut ajouter la lettre `r` pour inverser le sens des opérateurs dans l'opération pour `FSUB` et `FDIV`

Exemples

La division se fait toujours entre `%st(0)` et un autre registre ou une adresse mémoire. Par défaut (sauf suffixe `r`), c'est `%st(0)` qui est le dividende.

```
1  FDIV %st(3), %st(0)   # divise ST(0) par ST(3),
2                        # resultat dans ST(0)
3  FDIV %st(3)           # divise ST(0) par ST(3),
4                        # resultat dans ST(0)
5  FDIVR %st(3)          # divise ST(3) par ST(0),
6                        # resultat dans ST(0)
7  FDIV %st(0), %st(3)   # divise ST(0) par ST(3),
8                        # resultat dans ST(3)
9  FDIVR %st(0), %st(3)  # divise ST(3) par ST(0),
10                       # resultat dans ST(3)
```

Instructions de contrôle de la pile des registres

- `FINIT` : pour ré-initialiser la pile des registres
- `FINCSTP` : dépile un élément
- `FDECSTP` : empile 0 (avec un état *vide)*

Encore quelques instructions

- `FABS` : remplace `%st(0)` par sa valeur absolue
- `FCHS` : change le signe de `%st(0)`
- `FSQRT` : remplace `%st(0)` par sa racine carrée
- `FIST dest` : stocke la valeur contenue dans `%st(0)` en l'arrondissant à l'entier le plus proche - `dest` doit être une adresse mémoire (16 ou 32 bits).
- `FISTP dest` : même chose, mais `%st(0)` est dépilé ensuite - `dest` doit être une adresse mémoire (16 ou 32 ou 64 bits).