Nombre Flottants & Pipelines & Opérations Multicycles

1 Nombre Flottants - Représentation "virgule flottante" (standard IEEE 754)

Soit :
- S est le bit de signe
- PE est la partie exposant
- F représente la partie fractionnaire (mantisse) après le 1 implicite

Pour les flottants simples précision (32 bits), on a le codage suivant :

S	PE : Partie Exposant	F : Partie Fractionnaire	
31	30 23	22 0	(numéro bit)

Avec les valeurs suivantes :

0 ou 1	0	0	représente 0
0 ou 1	0	$\neq 0$	représente $S \times 0, F \times 2^{-126}$ (nombre *dénormalisé*)
0 ou 1	$0 < PE < 255$	quelconque	représente $S \times 1, F \times 2^{PE-127}$ (nombre *normalisé*)
0 ou 1	255	0	représente $\pm\infty$
0 ou 1	255	$\neq 0$	représente NaN (Not a Number)

1.1 Quels nombres simple précision correspondent aux mots 32 bits suivants ?

`0x41300000`

`0x41E00000`

`0x00000000`

`0xFFC00000`

1.2 Écrire 1 et -1000 de façon normalisée.

1.3 Donner le plus grand positif et son prédécesseur. Indiquer leur écart.

1.4 Donner le plus petit positif normalisé et dénormalisé (non nul).

1.5 Donner le plus grand et le plus petit négatif (non nul).

2 Nombre Flottants - Conversions

Soient les déclarations C suivantes :

```
int x;    // Entier 32 bits
float f;  // Flottant 32 bits simple précision
double d; // Flottant 64 bits double précision
          // (1 bit de signe, 11 bits d'exposant, 52 bits de mantisse)
          // (pour les nombres normalisés, l'exposant est décalé de 1023)
          // (pour les nombres dénormalisés, l'exposant est -1022)
```

2.1 Indiquer si les assertions suivantes sont vraies ou fausses en justifiant.

`x == (int)(float)x`

`x == (int)(double)x`

`f == (float)(double)f`

`d == (float)d`

`2 / 3 == 2 / 3.0`

`d < 0.0` $\rightarrow$ `2 * d < 0.0`

`d > f` $\rightarrow$ `-f < -d`

`d * d >= 0.0`

`(d + f) - d == f`

3 Pipelines

Un processeur *P1* dispose du pipeline à 5 étages pour les opérations entières et les chargements / rangements flottants et des pipelines suivants pour les instructions flottantes :

Multiplication : `fmul`	LI	DI	LR	EX1	EX2	EX3	EX4	ER
Addition : `fadd`	LI	DI	LR	EX1	EX2	ER		

Les instructions flottantes `lf` (Load Float) et `sf` (Store Float) ont les mêmes pipelines que les instructions entières LW (Load Word) et SW (Store Word).

Une instruction i a une latence de n si l'instruction suivante (qui utilise le résultat de la première) peut commencer au cycle $i + n$. Une latence de 1 signifie que l'instruction suivante peut commencer au cycle suivant.

3.1 Donner les latences des instructions flottantes `fmul` et `fadd`.

3.2 Donner les latences des instructions flottantes `fmul` et `fadd` pour le processeur *P2* suivant.

Multiplication : `fmul`	LI	DI	LR	EX1	EX2	EX3	EX4	EX5	EX6	ER
Addition : `fadd`	LI	DI	LR	EX1	EX2	EX3	EX4	ER		

4 Latences

Soit le jeu d'instructions suivant : (on dispose de 32 registres entiers R0 à R31 et de 32 registres flottants F0 à F31)

Instructions qui travaillent sur des flottants :	
`lf Fdest, address`	Charge un flottant depuis l'adresse (l'adresse peut être de la forme `N(R)` avec `N` un immédiat et `R` un registre entier)
`sf Fsrc, address`	Stocke un flottant dans l'adresse
`fadd Fdest, Fsrc1, Fsrc2`	`Fdest` ← `Fsrc1` + `Fsrc2` (addition flottante simple précision)
`fsub Fdest, Fsrc1, Fsrc2`	`Fdest` ← `Fsrc1` − `Fsrc2` (soustraction flottante simple précision)
`fmul Fdest, Fsrc1, Fsrc2`	`Fdest` ← `Fsrc1` × `Fsrc2` (multiplication flottante simple précision)
`fdiv Fdest, Fsrc1, Fsrc2`	`Fdest` ← `Fsrc1` / `Fsrc2` (division flottante simple précision)
Instructions qui travaillent sur des entiers :	
`lw Rdest, address`	Charge un entier depuis l'adresse (l'adresse peut être de la forme `N(R)` avec `N` un immédiat et `R` un registre entier)
`sf Rsrc, address`	Stocke un entier dans l'adresse
`add Rdest, Rsrc1, Rsrc2`	`Rdest` ← `Rsrc1` + `Rsrc2` (addition sur des entiers)
`addi Rdest, Rsrc1, Rsrc2`	`Rdest` ← `Rsrc1` + `Rsrc2` (addition sur des entiers non signés)
Branchements :	
`bne Ra, Rb, Label`	Si `Ra` ≠ `Rb` : `goto Label`
`bneq R, Label`	Si `R` ≠ 0 : `goto Label`

Les additions, soustractions et multiplications flottantes sont pipelinées. Une nouvelle instruction peut démarrer à chaque cycle. Les latences sont de 2 cycles pour `lf` et fournies par les résultats de l'exercice 1 pour les multiplications et additions flottantes.
La division a une latence de 15 cycles. Elle n'est pas pipelinée : une nouvelle division ne peut commencer que lorsque la division précédente est terminée.

```
float x[100];   float y[100];   float z[100];

for (size_t i = 0; i < 100; ++i)
{
    z[i] = x[i] * y[i];
}
```

4.1 Traduire ce code C en assembleur en énumerant les numéros de cycles à côté des instructions pour les deux processeurs de l'exercice 1. Les tableaux `x`, `y` et `z` sont rangés à la suite en mémoire. `R1` contient déjà l'adresse de `x[0]` et `R3` l'adresse de `x[100]` (l'adresse de fin de `x`).

4.2 Donner le code C de la boucle après déroulage d'ordre 2.

4.3 Donner le code C de la boucle après déroulage d'ordre 4.

4.4 Traduire le code C déroulé avec un facteur de 2 en assembleur en énumerant les numéros de cycles à côté des instructions pour les deux processeurs.

4.5 Traduire le code C déroulé avec un facteur de 4 en assembleur en énumerant les numéros de cycles à côté des instructions pour les deux processeurs.

4.6 Pour chaque code et chaque processeur : quel est le nombre de cycles par itération ?

4.7 Quel est le facteur de déroulage maximal qui permettrait d'améliorer le nombre de cycle par itération ? Quel serait le nombre des différentes instructions (chargements, multiplications, ...) ? le nombre de cycles ? Le nombre de cycles par itération ?

4.8 Donner le code C de la boucle après déroulage d'ordre 2 dans le cas général, c'est-à-dire en remplaçant 100 par `N`.

5 Somme Des Carrés (Bonus)

Soit le code C suivant :

```
float x[100];   float s = 0.f;

for (size_t i = 0; i < 100; ++i)
{
    s += x[i] * x[i];
}
```

5.1 Mêmes questions que pour l'exercice 2.

6 Multiplication Vs Division

Soit les code C suivants équivalents mathématiquement :

```
float x[100];
float y[100];
float z[100];
float a;

// Calcul de a


for (size_t i = 0; i < 100; ++i)
{
    z[i] = x[i] / a + y[i];
}
```

```
float x[100];
float y[100];
float z[100];
float a;

// Calcul de a

float a_inv = 1.f / a;

for (size_t i = 0; i < 100; ++i)
{
    z[i] = x[i] * a_inv + y[i];
}
```

Dans cet exercice, on utilise les latences suivantes :

Instructions	Latence	Pipelinée ?
`lf`	2	oui
`fadd`, `fsub`	3	oui
`fmul`	5	oui
`fdiv`	15	non

Les tableaux `x`, `y` et `z` sont rangés à la suite en mémoire.
`R1` contient déjà l'adresse de `x[0]` et `R3` l'adresse de `x[100]` (l'adresse de fin de `x`).
`a` est déjà calculé dans `F31` pour le premier code et `a_inv` est déjà calculé dans `F31` pour le deuxième.

6.1 Pour chaque code C sans déroulage et avec un déroulage de 2, écrire le code assembleur correspondant en énumerant les numéros de cycle.