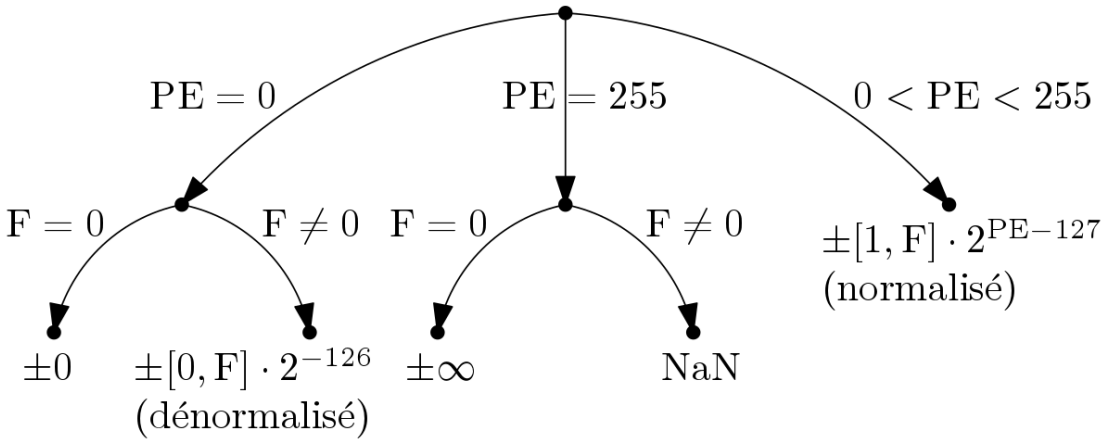
Nombres Flottants & Pipelines & Opérations Multicycles Correction

1. Nombres Flottants - Représentation

Schéma d'aide pour décoder un flottant en binaire.



1. Hexa → binaire → Décimal

- 0x41300000 : 0|100 0001 0|011 0000 0000 0000 0000 0000 → S = 1, Exposant = 130, Mantisse : $2^{-2}+2^{-3}$ → 1 * 1.375 * 2^3 =11
- 0x41E00000 : 0|100 0001 1|110 0000 0000 0000 0000 0000 → S = 1, Exposant = 131, Mantisse : $2^{-1}+2^{-2}$ = ½ + ¼ → 1 * 1,75 * 2^4 = 28
- 0x00000000 : +0
- 0xFFC00000 : 1|111 1111 1|100 0000 0000 0000 0000 0000 → S = -1, Exposant = 255 et Mantisse = 2^{-1} → NaN

2. 1/-100 en binaire/Hexa

- 1 : 0|011 1111 1|000 0000 0000 0000 0000 0000 = 0x3F800000
  Exposant doit être égal à 127 pour que l'on multiplie par $2^{127-127} = 2^0 = 1$
  La mantisse égale à 0 pour l'on multiplie seulement 1
  → 1 * 1 * 1 = 1
- -100 : 1|100 0100 0|111 1010 0000 0000 0000 0000 = 0xC47A0000
  Convertir 100 en binaire, sans signe, normalement : 11 1110 1000
  Décaler la virgule vers la gauche, de sorte à laisser 1,… : 1,1 1110 1000
  On a décalé la virgule de 9 positions.
  La mantisse est donc ce qui est derrière la virgule : 1 1110 1000 avec extension de 0 à droite si nécessaire.
  L'exposant doit être calculé de sorte que quand on lui soustrait 127, on obtienne 9 → 136.

3. Plus grand/plus petit/etc

- Plus grand positif normalisé : MAX_FLOAT
  0|111 1111 0|111 1111 1111 1111 1111 1111 = 0x7F7FFFFF
  = $(2-2^{-23}) * 2^{127} = 3{,}40282346 * 10^{38}$
- Prédécesseur du plus grand positif normalisé :
  0|111 1111 0|111 1111 1111 1111 1111 1110 = 0x7F7FFFFE
  = $(2-2^{-22}) * 2^{127} = 3{,}40282326 * 10^{38}$
  Ecart : $2^{-23} * 2^{127} = 2 * 10^{31}$
- Plus petit positif normalisé :
  0|000 0000 1|000 0000 0000 0000 0000 0000 = 0x00800000
  = $2^{-126} = 1{,}1754921 * 10^{-38}$
- Plus petit positif dénormalisé :
  0|000 0000 0|000 0000 0000 0000 0000 0001 = 0x00000001
  = $2^{-149} = 1{,}17549 * 10^{-45}$
- Plus grand négatif (dénormalisé), *i.e.*, le plus proche de 0 (prendre le plus petit dénormalisé et mettre un signe négatif) :
  1|000 0000 0|000 0000 0000 0000 0000 0001 = 0x80000001
- Plus petit négatif (normalisé) (prendre le plus grand et mettre un signe négatif) :
  1|111 1111 0|111 1111 1111 1111 1111 1111 = 0xFF7FFFFF

2. Nombres Flottants – Conversions

Rappels :

- NaN : x/-+0, division/multiplication/addition/soustraction/addition
- N'importe quelle comparaison avec NaN est fausse sauf x != NaN qui est toujours vrai
- 1/-Inf = -0, -1/+Inf = +0
- 1/+0 = +Inf, 1/-0 = -Inf

1. VRAI/FAUX ?

- x == (int) (float) x FAUX
  Les 32 bits du int x ne tiennent pas dans la mantisse du float
- x == (int) (double) x VRAI
  Les 32 bits tiennent cette fois dans la mantisse du double.
- f == (float) (double) f VRAI
  On ne perd pas d'info dans la conversion (float plus petit que double)
- d == (float) d FAUX
  Taille mantisse
- 2/3 == 2/3.0 FAUX
  Division entière vs division flottante. On compare 0 avec 0,66 ici.

- d < 0.0 → 2 * d < 0.0 VRAI
  d n'est pas NaN et +Inf mais peut être -Inf mais 2 * -Inf est bien inférieur à 0.0
- d > f → -f < -d FAUX
  Ne peuvent être NaN.
  Fonctionne avec +/-inf
  MAIS le sens de la comparaison n'a pas changé !!
- d * d >= 0.0 VRAI
  Sauf si d = NaN.
- (d + f) – d == f FAUX
  Principal problème : si jamais f est trop petit par rapport à d, d+f peut être égal à d (si f inférieur à l'écart entre d et le nombre supérieur directement à d)

3. Pipelines

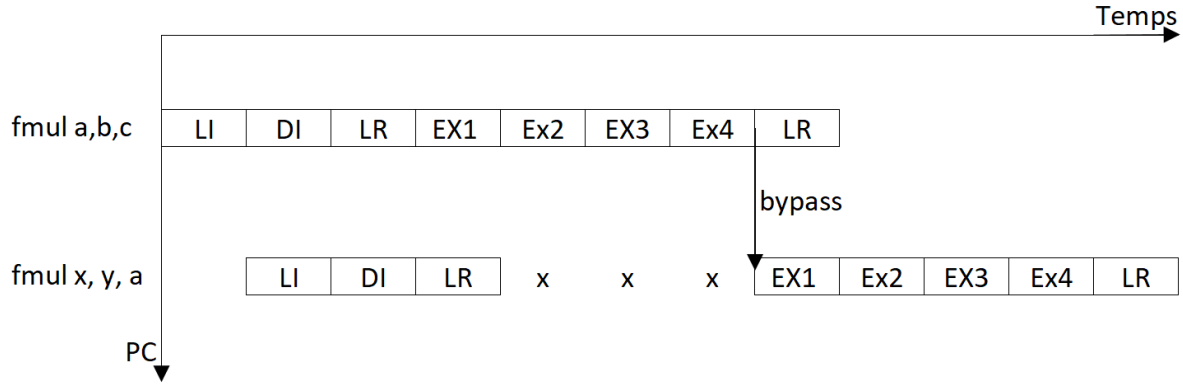
1. Latences

La latence s'obtient en regardant combien de cycles sont perdus si on veut enchaîner deux mêmes opérations, avec la deuxième qui utilise le résultat de la première.

P_1 : Fmul : 4 cycles
Fadd : 2 cycles

P_2 : Fmul : 6 cycles
Fadd : 4 cycles

Exemple avec Fmul sur P_1



Il faut se demander quel est le résultat dont on a besoin, et quand ?
Ici, on a besoin du résultat qui va être enregistré dans a pour commencer à faire le premier étage EX1. Grâce au bypass, on n'a pas besoin d'attendre que ce soit enregistré.

4. Latences

1. En assembleur

Cycles	P_1	P_2
1	loop : lf F0, 0(R1)	loop : lf F0, 0(R1)
2	lf F1, 400(R1)	lf F1, 400(R1)
3	addi R1, R1, 4	addi R1, R1, 4
4	fmul F0, F0, F1	fmul F0, F0, F1
5		
6		
7		
8	sf F0, 796(R1)	
9	bne R1, r3, loop	
10		sf F0, 796(R1)
11		bne R1, r3, loop

C'est de la forme while car le compilateur a vérifié qu'on faisait au moins une itération. Sinon, pas d'assembleur du tout.

2. Déroulage ordre 2

```
for(size_t i =0 ; i<100 ; i+=2){
   z[i+0] = x[i+0] * y[i+0];
   z[i+1] = x[i+1] * y[i+1];
}
```

3. Déroulage ordre 4

```
for(size_t i =0 ; i<100 ; i+=4){
   z[i+0] = x[i+0] * y[i+0];
   z[i+1] = x[i+1] * y[i+1];
   z[i+2] = x[i+2] * y[i+2];
   z[i+3] = x[i+3] * y[i+3];
}
```

4. Assembleur déroulage ordre 2

Cycles	P_1	P_2
1	loop : lf F0, 0(R1)	loop : lf F0, 0(R1)
2	lf F2, 4(R1)	lf F2, 4(R1)
3	lf F1, 400(R1)	lf F1, 400(R1)
4	lf F3, 404(R1)	lf F3, 404(R1)
5	fmul F0, F0, F1	fmul F0, F0, F1
6	fmul F2, F2, F3	fmul F2, F2, F3
7	addi R1, R1, 8	addi R1, R1, 8
8		
9	sf F0, 792(R1)	
10	sf F2, 796(R1)	
11	bne R1, R3, loop	sf F0, 792(R1)
12		sf F2, 796(R1)
13		bne R1, R3, loop

5. Assembleur déroulage ordre 4

Cycles	P_1	P_2
1	loop : lf F0, 0(R1)	loop : lf F0, 0(R1)
2	lf F2, 4(R1)	lf F2, 4(R1)
3	lf F4, 8(R1)	lf F4, 8(R1)
4	lf F6, 12(R1)	lf F6, 12(R1)
5	lf F1, 400(R1)	lf F1, 400(R1)
6	lf F3, 404(R1)	lf F3, 404(R1)
7	lf F5, 408(R1)	lf F5, 408(R1)
8	lf F7, 412(R1)	lf F7, 412(R1)
9	fmul F0, F0, F1	fmul F0, F0, F1
10	fmul F2, F2, F3	fmul F2, F2, F3
11	fmul F4, F4, F5	fmul F4, F4, F5
12	fmul F6, F6, F7	fmul F6, F6, F7
13	addi R1, R1, 16	addi R1, R1, 16
14	sf F0, 784(R1)	
15	sf F2, 788(R1)	sf F0, 784(R1)
16	sf F4, 792(R1)	sf F2, 788(R1)
17	sf F6, 796(R1)	sf F4, 792(R1)
18	bne R1, R3, loop	sf F6, 796(R1)
19		bne R1, R3, loop

6. Performances

	P_1	P_2
Code Original	8	10
Déroulage de 2	11/2 = 5,5	13/2 = 6,5
Déroulage de 4	18/4 = 4,5	19/4 = 4,75

A noter : plus on déroule, plus on gagne de cycles par itération, mais ce n'est pas proportionnel au nombre de déroulage.

7. Déroulage Max

On dispose de 32 registres flottants. 2 sont nécessaires par itération.
Le déroule maximal est donc de 16.

Il y aura 32 chargements, 16 multiplications, 16 stockages, une addition et un branchement = 66 instructions pour 16 itérations.

66/16 = 4,125 cycles par itérations. (Ce qui ne prend pas en compte que 100 n'est pas un multiple de 16, donc en réalité, il y aura une dernière boucle avec seulement un déroulage de 4, ie 4,5 cycles par itérations).

On peut observer que le déroulage par 16 permet d'obtenir une amélioration minime par rapport à celui par 4.
De plus, la différence entre P1 et P2 s'estompe complétement et ils ont la même performance.

8. Cas général

```
size_t i ;

for(i =0 ; i+2<N; i+=2){
   z[i+0] = x[i+0] * y[i+0];
   z[i+1] = x[i+1] * y[i+1];
}

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

5. Ex bonus

6. Multiplication vs Division

1. Assembleur pour les boucles sans déroulage

Cycles	div	mul
1	loop : lf F0, 0(R1)	loop : lf F0, 0(R1)
2	lf F1, 400(R1)	lf F1, 400(R1)
3	fdiv F0, F0, F31	fmul F0, F0, F1
4	addi R1, R1, 4	addi R1, R1, 4
5		
6		
7		
8		fadd F0, F0, F1
9		
10		
11		sf F0, 796(R1)
12		bne R1, R3, loop
13		
14		
15		
16		
17		
18	fadd F0, F0, F1	
19		
20		
21	sf F0, 796(R1)	
22	bne R1, R3, loop	

1. Boucle avec division, déroulage d'ordre 2

```
for (size_t i = 0; i < 100; i+=2)
{
        z[i] = x[i] / a + y[i];
        z[i+1] = x[i+1] / a + y[i+1];
}
```

2. Boucle avec multiplication, déroulage d'ordre 2

```
for (size_t i = 0; i < 100; i+=2)
{
        z[i] = x[i] * a_inv + y[i];
        z[i+1] = x[i+1] * a_inv + y[i+1];
}
```

3. Assembleur pour les boucles avec déroulage d'ordre 2

Cycles	div	mul
1	loop : lf F0, 0(R1)	loop : lf F0, 0(R1)
2	lf F1, 400(R1)	lf F2, 4(R1)
3	fdiv F0, F0, F31	fmul F0, F0, F31
4	lf F2, 4(R1)	fmul F2, F2, F31
5	lf F3, 404(R1)	lf F1, 400(R1)
6	addi R1, R1, 8	lf F3, 404(R1)
7		addi R1, R1, 8
8		fadd F0, F0, F1
9		fadd F2, F2, F3
10		
11		sf F0, 792(R1)
12		sf F2, 796(R1)
13		bne R1, R3, loop
14		
15		
16		
17		
18	fdiv F2, F2, F31	
19	fadd F0, F0, F1	
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		
33	fadd F2, F2, F3	
34	sf F0, 792(R1)	
35		
36	sf F2, 796(R1)	
37	bne R1, R3, loop	

Calcul de performances :

Divisions : 37 cycles/2 itérations = 18.5/itération.
Multiplications : 13 cycles/2 itérations = 6.5/itération.

Donc la multiplication est plus efficace en terme de cycles par itération : on a perdu 15 cycles pour la division une seule.

➔ We love multiplications !