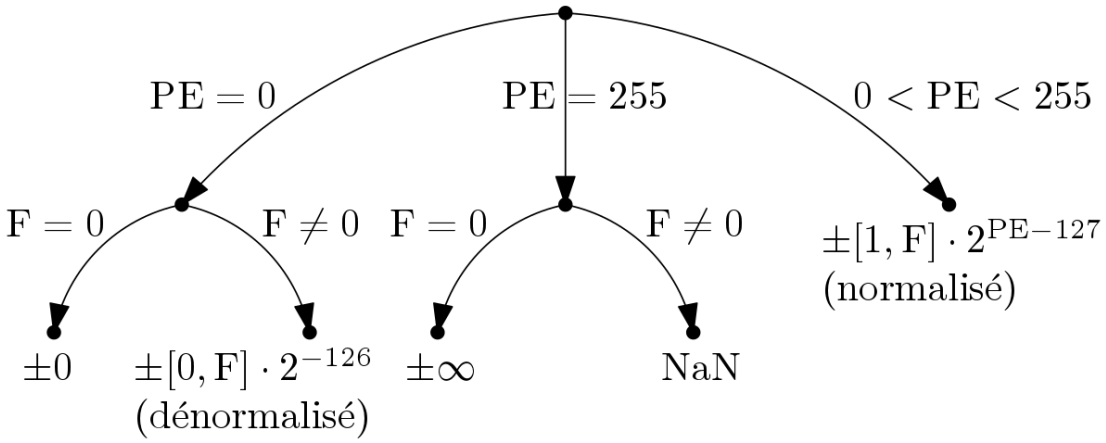
Nombres Flottants & Pipelines & Opérations Multicycles Correction

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

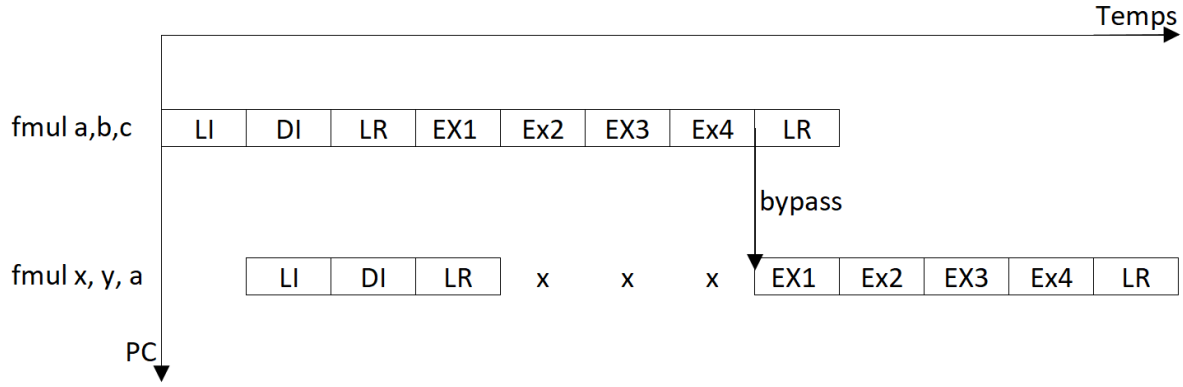
1. Latences

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.