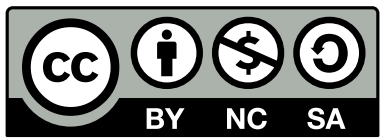
1010 Flottants

INF2171
Organisation des ordinateurs et assembleur

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur
v241



Rappels

Représentation des entiers

- Notation positionnelle (binaire)
- Non signés vs. signés (complément à deux)
- Nombre de bits fixes: attention aux débordements

Plan

Représentation des nombres flottants

Nombre décimaux

Nombres

- Entiers $\mathbb{N} \subset$ relatifs $\mathbb{Z} \subset$ décimaux $\mathbb{D} \subset$ fractions $\mathbb{Q} \subset$ réels $\mathbb{R}$

Notation décimale et nombres décimaux

- Un signe
- Une partie entière
- Un séparateur décimal (virgule ou point)
- Une partie décimale
- Exemple : 12.9, 0.0000000419, -17478000000
- Note: $\frac{1}{3}$ ou π non représentable ainsi

Notation positionnelle à virgule

Puissances de la base

- Avant la virgule : des puissances positives de la base
- Après la virgule : des puissances négatives (rappel: $a^{-b} = \frac{1}{a^b}$)
- Base 10 : ..., 1000, 100, 10, 1, 0.1, 0.01, 0.001, ...
- Base 2 : ..., 8, 4, 2, 1, 0.5, 0.25, 0.125, ...

Exemples

- $10.25_{(10)} = 1 \times 10^1 + 0 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$
- $1010.01_{(2)} = 2^3 + 2^1 + 2^{-2} = 8 + 2 + 0.25 = 10.25_{(10)}$

Puissances de deux

- ...
- $10000_{(2)} = 2^4 = 16_{(10)}$
- $1000_{(2)} = 2^3 = 8_{(10)}$
- $100_{(2)} = 2^2 = 4_{(10)}$
- $10_{(2)} = 2^1 = 2_{(10)}$
- $1_{(2)} = 2^0 = 1_{(10)}$
- $0.1_{(2)} = 2^{-1} = \frac{1}{2} = 0.5_{(10)}$
- $0.01_{(2)} = 2^{-2} = \frac{1}{4} = 0.25_{(10)}$
- $0.001_{(2)} = 2^{-3} = \frac{1}{8} = 0.125_{(10)}$
- $0.0001_{(2)} = 2^{-4} = \frac{1}{16} = 0.0625_{(10)}$
- ...

Représentation binaire à virgule fixe

Virgule fixe

On a un nombre fini de bits (16, 32, 64, etc.)

- Fixer un nombre bits pour la partie entière
- Fixer un nombre de bits pour la partie fractionnaire

En pratique: rarement utilisé

- Problème d'ordre de grandeur
- C'est généralement plus simple d'utiliser des entiers et de changer l'unité
  - Secondes $\rightarrow$ nanosecondes
  - Dollars $\rightarrow$ milliers de cents

Notation scientifique

- Un signe
- La partie entière: un seul chiffre non nul
- Une partie fractionnaire (finie)
- Une puissance de la base (signée et finie)
- Exemples : 1.29×10^{1}, 4.19×10^{-8}, -1.7198×10^{10}

Avantage

- L'ordre de grandeur est explicite
- Possiblement moins de chiffres (de 0) pour les nombres avec des grands/petits ordres de grandeur

Notation E (compacte)

- Variante compacte avec la lettre *e* (exposant)
- Habituel dans les affichages électroniques (calculatrice)
- Exemples : `1.29e1`, `4.19e-8`, `-1.7198e10`

Norme IEEE 754

IEEE

- Institute of Electrical and Electronics Engineers (1963)
- Institut des ingénieurs électriciens et électroniciens

Activités

- Publications et conférences scientifiques
- Établissement de normes

Exemples de normes IEEE

- IEEE 802 : réseaux locaux (LAN)
  - 802.11 : Wi-Fi
- IEEE 1003 : Posix
- IEEE 754 : Représentation en virgule flottante

Norme IEEE 754 (depuis 1985)

- IEEE Standard for Floating-Point Arithmetic, IEEE 754-2019 (84 pages)
- Norme ISO/IEC 60559:2020 (Même contenu)

Contenu

- Format des nombres: 3 de base pour le binaire
- Règles d'arrondis: 5 règles possibles
- Opérations arithmétiques: addition, soustraction, multiplication, division, racine carrée...
- Gestion des exceptions: débordement, division par zéro, nombre spécial 0...

Norme IEEE 754

Formes normalisées « scientifiques » en base 2

- Forme normalisée selon $\pm m \times 2^e$
- m est la mantisse
  - Représentation binaire, partie entière positive d'un seul chiffre
    $\rightarrow$ C'est forcément « 1. »
- e est l'exposant de la puissance de 2
  - Positif ou négatif

Codage normalisé

- 1 bit S code le signe
  $\rightarrow$ Pas de représentation en complément
- Des bits M codent la mantisse m
  $\rightarrow$ On code seulement les chiffres binaires qui suivent la virgule
- Des bits E codent l'exposant e
  $\rightarrow$ Codage binaire décalé (cf. 02-arithmétique)

Simple-précision (32 bits): binary32

- $-1^S \times 1.M \times 2^{E-127_{(10)}}$

S. Signe (1 bit)

- 0 = positif ; 1 = négatif

E. Exposant (8 bits)

- `0x00` et `0xFF` : spécial (plus tard)
- `0x01` à `0xFE` : 2^{-126} à 2^{127}
- `0x7F` (127) : $2^0 = 1$ (on appelle ça le pôle)
- Maximum aux alentours de 3.4×10^{38}

M. Mantisse (23 bits)

- Représentation traditionnelle positionelle
- Ne pas oublier le « 1. » implicite devant $\rightarrow$ 24 bits de précision
- La norme utilise le terme *significand* (au lieu de *mantissa*)
- Environ 7 chiffres décimaux de précision

Décimal vers simple-précision

Exercice: convertir -17.25

Décimal vers simple-précision

Exercice: convertir -17.25

- $-17.25_{(10)} = -10001.01_{(2)} = -1.000101_{(2)} \times 2^4$
- S: 1 (négatif)
- E: 127+4 = 131 $\rightarrow$ 10000011
- M: 00010100000000000000000 (1. implicite, combler avec 0)
- Soit 1100 0001 1000 1010 0000 0000 0000 0000
- Soit C1 8A 00 00 (représentation compacte hexadécimale)

Exercice: convertir 0.125

Décimal vers simple-précision

Exercice: convertir -17.25

- $-17.25_{(10)} = -10001.01_{(2)} = -1.000101_{(2)} \times 2^4$
- S: 1 (négatif)
- E: 127+4 = 131 $\rightarrow$ 10000011
- M: 00010100000000000000000 (1. implicite, combler avec 0)
- Soit 1100 0001 1000 1010 0000 0000 0000 0000
- Soit C1 8A 00 00 (représentation compacte hexadécimale)

Exercice: convertir 0.125

- $0.125 = \frac{1}{8} = 1 \times 2^{-3}$
- S: 0 (positif)
- E: 127-3 = 124 $\rightarrow$ 01111100
- M: 00000000000000000000000
- $\rightarrow$ 0011 1110 0000 0000 0000 0000 0000 0000
- $\rightarrow$ 3E 00 00 00

Décimal vers simple-précision
Exercice: convertir 0.2

Décimal vers simple-précision

Exercice: convertir 0.2

- Problème:
  $0.2 = 0.001100110011..._{(2)} = 1.100110011..._{(2)} \times 2^{-3}$
  - S: 0 (positif)
  - E: 127-3 = 124 $\rightarrow$ 01111100
  - M: 10011001100110011001100... on a pas assez de bits!
- Solution: arrondir vers le nombre représentable le plus proche
  - 0011 1110 0100 1100 1100 1100 1100 1100
  - $\rightarrow 0.00110011001100110011001100_{(2)}$
    $= 0.199999988079071044921875$
  - 0011 1110 0100 1100 1100 1100 1100 1101
  - $\rightarrow 0.00110011001100110011001101_{(2)}$
    $= 0.200000002980232238769531250 \leftarrow$ lui est plus proche

Problème des flottants

- Les nombres et les résultats des calculs sont pas toujours exacts
- $\rightarrow$ On y reviendra

Simple-précision vers décimal

Exercice: convertir `0x41240000`

Simple-précision vers décimal

Exercice: convertir 0x41240000

- 01000001001001000000000000000000
- S: 0 $\rightarrow$ positif
- E: $10000010_{(2)} - 127 = 130 - 127 = 3$
- M: 01001000000000000000000 $\rightarrow 1.01001_{(2)}$
- Résultat: $+1.01001_{(2)} \times 2^3 = 1010.01_{(2)} = 10.25_{(10)}$

Exercice: convertir 0x44800000

Simple-précision vers décimal

Exercice: convertir 0x41240000

- 01000001001001000000000000000000
- S: 0 $\rightarrow$ positif
- E: $10000010_{(2)} - 127 = 130 - 127 = 3$
- M: 01001000000000000000000 $\rightarrow 1.01001_{(2)}$
- Résultat: $+1.01001_{(2)} \times 2^3 = 1010.01_{(2)} = 10.25_{(10)}$

Exercice: convertir 0x44800000

- 01000100100000000000000000000000
- S: 0 $\rightarrow$ positif
- E: $10001001_{(2)} - 127 = 137 - 127 = 10$
- M: 00000000000000000000000 $\rightarrow 1.0_{(2)}$
- Résultat: $+1 \times 2^{10} = 1024_{(10)}$

Formats IEEE 754

Précision:	Simple	Double	Quadruple
Taille:	32 bits	64 bits	128 bits
Nom IEEE:	binary32	binary64	binary128
Type Java:	`float`	`double`	
Signe:	1 bits	1 bit	1 bit
Exposant:	8 bits	11 bits	14 bits
Pôle:	127 (`0x7F`)	1023 (`0x3FF`)	8191 (`0x1FFF`)
Mantisse:	23 bits	52 bits	113
Maximum:	$\approx 3.4 \times 10^{38}$	$\approx 1.8 \times 10^{308}$	$\approx 1.2 \times 10^{4932}$
Précision*:	$\approx$ 7 chiffres	$\approx$ 16 chiffres	$\approx$ 34 chiffres

- Demi-précision: 16 bits, *half-precision*, *binary16*
- Octuple-précision: 256 bits, *binary256*

* précision en chiffres décimaux

Littéraux flottants hexadécimaux

Notation p

Flottant littéral avec une représentation exacte IEEE 754

- Exemple `0xA.Bp5` = $\mathrm{A.B}_{(16)} \times 2^5$
- Disponible en Java, RARS, gnu as, C, C++, etc.

Syntaxe

- Optionnel: un signe
- `0x` (préfixe des littéraux hexadécimaux)
- Une partie hexadécimale entière
- Optionnel: un point suivi d'une partie hexadécimale fractionnaire
- La lettre `p`
- Une puissance de deux, **décimale**, possiblement négative

Exercice notation p

Représentation IEEE 754 simple-précision de $n =$ `0xA.Bp5` ?

Exercice notation p

Représentation IEEE 754 simple-précision de $n =$ `0xA.Bp5` ?

- $n = \text{A.B}_{(16)} \times 2^5$
- Passer en binaire (chaque chiffre hexa donne 4 bits)
  - $n = 1010.1011_{(2)} \times 2^5$
- Décaler la virgule au premier 1
  - Ici on décale de 3, et on met à jour la puissance de 2
  - $n = 1.0101011_{(2)} \times 2^8$
- On code le flottant IEEE simple-précision
  - S (1 bit): `0` (positif)
  - E (8 bits): 8+127 = 135 = `10000111`
  - M (23 bits): `01010110000000000000000`
  - $\rightarrow$ `0100 0011 1010 1011 0000 0000 0000 0000`
  - $\rightarrow$ `43 AB 00 00`

Flottants décimaux

- Codage des nombre à virgule décimaux (base10)
- $0.2 \rightarrow$ Une représentation exacte est possible
- Utilisations spécialisées (secteur financiers et fiscaux)
- Norme relativement récente: IEEE 754-2008
- Trois formats: decimal32, decimal64 et decimal128
- Représentation spéciale des chiffres décimaux de la mantisse

Exception et perte de précision

Exception et perte de précision

Problème

- On ne peut pas représenter tous les nombres réels
- Il y a donc des résultats de calculs non représentables
- $\rightarrow$ On approxime

Contrairement aux entiers

- Ces valeurs ne sont pas qu'aux extrémités
- Un flottant représente une infinité de nombres proches
- L'approximation d'un flottant n'est pas uniforme
  - Dépend de la magnitude
  - ULP: *Unit in the last place* = l'écart avec le flottant suivant

Travailler avec des flottants

- Savoir que les flottants ne sont pas précis
- Savoir comment minimiser l'impact de cette imprécision
- $\rightarrow$ On y reviendra

5 exceptions IEEE 754

5 situations problématiques identifiées:

- Opération invalide. Exemple: $\sqrt{-1}$
  - $\rightarrow$ Valeur spéciale « pas un nombre » (NaN, *not a number*)
- Division par zéro. Exemple: $\frac{1}{0}$
  - $\rightarrow$ Valeur spéciale « infini »
- Débordement, dépassement (*overflow*). Ex.: `grand * grand`
  - Exposant trop grand (nombre trop grand)
  - $\rightarrow$ Valeur spéciale « infini »
- Débordement, soupassement (*underflow*).
  - Exposant trop petit (nombre trop proche de zéro)
  - $\rightarrow$ Valeur spéciale « dénormalisée »
  - $\rightarrow$ Ou « zéro » si vraiment trop proche de zéro
- Inexactitude.
  - Pas assez de bits dans la mantisse
  - $\rightarrow$ Arrondi (5 règles possibles)

Valeur zéro

Pas de représentation normalisée possible de 0

- On ne peut écrire 0 avec un « 1, » implicite dans la mantisse

Représentation dénormalisée (convention)

- Si la mantisse et l'exposant sont tous à zéro
- Alors, le nombre vaut zéro

Mais le bit de signe reste $\rightarrow$ deux zéros

- $+0$ et -0 ont des représentations différentes
- Idée: distinguer « **très très** proche de zéro mais négatif » et « **très très** proche de zéro mais positif »
- Mais piégeux: `+0.0 == -0.0` $\rightarrow$ `true`

Infini

Codage

- Mantisse à 0
- Exposant à `0xFF`
- Le signe distingue $+\infty$ ou $-\infty$

Utilisation

- Débordement sur valeurs extrêmes: grand + grand
- Division par zéro: `1.0/0.0` $\rightarrow +\infty$; `-1.0/0.0` $\rightarrow -\infty$
  - Piégeux: `1.0/-0.0` $\rightarrow -\infty$
  - le == de zéro nous a menti!

Nombres dénormalisés

Codage

- Si l'exposant est à `0x00` (vaut 2^{-126})
- Alors, la mantisse n'a plus un « 1, » implicite mais un « 0, »
- $\rightarrow$ Permet de représenter des nombres **très proches** de zéro
- Contrainte: la précision diminue plus on s'approche de zéro

Exercice

- Décoder le flottant simple-précision `0x00000001`

Nombres dénormalisés

Codage

- Si l'exposant est à `0x00` (vaut 2^{-126})
- Alors, la mantisse n'a plus un « 1, » implicite mais un « 0, »
- $\rightarrow$ Permet de représenter des nombres **très proches** de zéro
- Contrainte: la précision diminue plus on s'approche de zéro

Exercice

- Décoder le flottant simple-précision `0x00000001`
- 0000 0000 0000 0000 0000 0000 0000 0001
- Vaut $+0.00000000000000000000001_{(2)} \times 2^{-126}$
- Soit 2^{-149}, qui est de l'ordre de 10^{-45}
- $\rightarrow$ Le plus petit nombre positif représentable en simple-précision

Pas des nombres

NaN (*not a number*)

- Représente des résultats non définis
- Exemples: $\sqrt{-1}$, $\frac{0}{0}$, $+\infty - +\infty$, etc.
- Parfois représente l'absence de donnée
- Mais piégeux: `NaN == NaN` $\rightarrow$ `false` ; `NaN != NaN` $\rightarrow$ `true`
- Et repiégeux: `NaN >= 1` $\rightarrow$ `false` ; `NaN <= 1` $\rightarrow$ `false`

Codage

- Signe quelconque
- Exposant à `0xFF`
- Mantisse $\neq 0$

NaN silencieux et NaN avertisseur

NaN silencieux (*quiet NaN*)

- Par défaut, NaN se propage silencieusement
- `1+NaN` $\rightarrow$ `NaN`

NaN avertisseur (*signaling NaN*)

- L'instruction lève une erreur
- `1+NaN` $\rightarrow$ instruction invalide

Codage spécifique

- x86, ARM, RISC-V: on regarde le premier bit de la mantisse
  - 1 $\rightarrow$ NaN silencieux
  - 0 $\rightarrow$ NaN avertisseur

Table récapitulative

Type	Exposant	Mantisse
Zéros	`0x00`	0
Dénormalisés	`0x00`	$\neq 0$
Normalisés	`0x01` à `0xFE`	libre
Infinis	`0xFF`	0
NaN	`0xFF`	$\neq 0$

5 arrondis IEEE 754

Exemples: arrondir $1.0011_{(2)}$ à 2 bits après la virgule

- Vers 0: on tronque la mantisse (`trunc`)
  - $1.0011_{(2)} \to 1.00_{(2)}$ car ce qui suit les 2 bits disparait
- À l'excès: vers $+\infty$ (`ceil`)
  - $1.0011_{(2)} \to 1.01_{(2)}$ car $1.01_{(2)} \geq 1.0011_{(2)}$
- À défaut: vers $-\infty$ (`floor`)
  - $1.0011_{(2)} \to 1.00_{(2)}$ car $1.00_{(2)} \leq 1.0011_{(2)}$
- Au plus proche, l'infini (loin de 0) si équidistant
  - $1.0011_{(2)} \to 1.01_{(2)}$ car plus proche que $1.00_{(2)}$
  - $1.001_{(2)} \to 1.01_{(2)}$ car équidistant mais arrondi vers ∞
  - Celui souvent utilisé à l'école
- « Au chiffre pair » le plus proche: le plus commun
  - $1.0011_{(2)} \to 1.01_{(2)}$ car plus proche que $1.00_{(2)}$
  - $1.001_{(2)} \to 1.00_{(2)}$ car équidistant mais dernier bit forcé à 0
  - Avantage: les équidistances se résolvent équitablement

Arrondi

Exemple : rationnels infinis

- Certains rationnels ont une représentation finie en décimal mais infinie en binaire
- Exemple: $0.2_{(10)}$

Arrondi

- On arrondi vers un nombre binaire représentable
- Exemple: 0.2 sera représenté par 0.200000002980232238769531250

Cette perte d'information se combine

- $0.37 + 0.2 = ?$
- La somme des arrondis n'est pas l'arrondi de la somme !

Flottants en RISC-V

Extensions F, D et Q

- F: flottants simple-précision (32 bits)
- D: flottants double-précision (64 bits)
- Q: flottants quadruple-précision (128 bits)

Caractéristiques

- Compatible avec IEEE 754-2008
- `f0` à `f31`: 32 registres additionnels pour les flottants
  - Avec noms d'ABI: `ft0-ft11`, `fs0-fs11`, `fa0-fa7`
- `fcsr`: un registre CRS (*control ans status register*)
  - Mode d'arrondi implicite (un des 5 modes)
  - Exceptions flottantes (chacune des 5 exceptions)
- Une trentaine de nouvelles instructions
- $\rightarrow$ Ça peut doubler la taille du processeur !
- $\rightarrow$ On fait juste un survol (très) rapide

Arrondi et exceptions

Registre spécial `fcsr`

- Lu et modifié via instructions spéciales `csr*`
- Et souvent pseudoinstructions dédiées

Exceptions: `fflags` (5 premiers bits de `fcsr`)

- Un bit pour chacun des cas d'exception IEEE 754
  - `nv` (invalide), `dz` (division par 0), `of` (*overflow*), `uf` (*underflow*), `nx` (inexact)
- Automatiquement (et silencieusement) mis à 1 par le CPU
- $\rightarrow$ Permet de vérifier globalement la qualité du résultat à la fin d'une séquence d'instrustions

Arrondi et exceptions

Arrondis: `frn` (3 bits suivants de `fcsr`)

- Pour configurer le mode d'arrondi par défaut (`dyn`, dynamique)
  - `rne` (*Round to Nearest, ties to Even*), le défaut du défaut
  - `rtz` (*Round Towards Zero*)
  - `rdn` (*Round Down*), vers -∞
  - `rup` (*Round Up*), vers $+\infty$
  - `rmm` (*Round to nearest, ties to Max Magnitude*)

Load et Store flottants

- Lectures `flw` (32 bits), `fld` (64 bits), `flq` (128 bits)
- Écritures: `fsw` (32 bits), `fsd` (64 bits), `fsq` (128 bits)
- Pseudo-instructions habituelles

Exemples

- `fld fs0, 8(sp)`
- `fsw fs0, label, t0` $\leftarrow$ pseudoinstruction

Directives de taille fixe

- `.float num` $\leftarrow$ alloue 4 octets (flottant simple-précision)
- `.double num` $\leftarrow$ alloue 8 octets (flottant double-précision)

Instructions arithmétiques

- `fadd.s rd, rs1, rs2, dyn`
- `add`: l'opération. `sub`, `mul` ou `div` aussi possibles
- `s`: la taille du flottant. `d` ou `q` aussi possibles
- `dyn`: le mode d'arrondi est celui configuré dans `fcsr`.
  - Un arrondi explicite est possible (`rne`, `rtz`, `rdn`, `rup`, `rmm`)

Remarques

- Pas de version immédiate
- Pas non plus de *load immediate*
- L'arrondi est configurable par instruction
- Pseudoinstruction: `dyn` est implicite (ouf!)

Exemple

- `fsub.d fs0, fs0, fs1` $\leftarrow$ pseudoinstruction, `dyn` est implicite

Autres opérations arithmétiques

Minimum et maximum

- `fmin.s rd, rs1, rs2`
- `fmin` ou `fmax` ; `s`, `d`, ou `q`

Racine carrée

- `fsqrt.s rd, rs1, dyn`
- `s` ou `d` ; `dyn` (implicite) ou un autre arrondi

Opération fusionnée: multiplication et addition

- `fmadd.s rd, rs1, rs2, rs3, dyn`
- Calcule `rs1*rs2+rs3` d'un coup (avec un seul arrondi final)
- Autres opérations possibles
  - `fnmadd` $\rightarrow$ `-rs1*rs2+rs3`
  - `fnsub` $\rightarrow$ `rs1*rs2-rs3`
  - `fnmsub` $\rightarrow$ `-rs1*rs2-rs3`

Conversions

Plein d'instructions `fcvt`

`fcvt.s.w rd, rs1, dyn`

Combinatoire de paramètres:

- Flottants $\rightarrow$ entiers (`s.w`), ou entiers $\rightarrow$ flottants (`w.s`)
- Taille de l'entier (`w` ou `l`), et taille du flottant (`s`, `d`, ou `q`)
  - Ici `l` (*long*) signifie double-mot (64 bits)
  - Oui, c'est incohérent avec le reste...
- Entier signé ou non (`w` ou `wu`, `l` ou `lu`)
- Mode d'arrondi (`dyn` ou les 5 explicites)

Exemple

`fcvt.d.l fs0, s0`

- Convertit le double-mot `s0` vers le flottant double-précision `fs0`
- Utilisé pour initialiser des flottants avec des valeurs entières

Copies

Injection de signe: `fsgnj`

`fsgnj.s rd, rs1, rs2`

- Copie `rs1` dans `rd` mais avec le signe de `rs2` (WTF!)
- ... ou l'inverse du signe de `rs2` : `fsgnjn` (WTF!!)
- ... ou le xor des signes de `rs1` et `rs2` : `fsgnjx` (WTF!!!)

Pseudoinstructions dérivées pertinentes

- `fmv.s rd, rs1` $\leftarrow$ copie un registre flottant
- `fneg.s rd, rs1` $\leftarrow$ calcul de l'opposé
- `fabs.s rs, rs1` $\leftarrow$ calcul de la valeur absolue

Copies de bits

- `fmv.x.w` (ou `fmv.w.x`)
- Copie un registre flottant vers/de un registre général
  - Copie les bits tels quels
  - Ne décode et n'interprète pas les nombres

Comparaison de flottants

`flt.s rd, rs1, rs2`

- `flt`: opération de comparaison.
  - `feq` et `fle` aussi disponibles en instructions
  - `fge` et `fgt` disponibles en pseudoinstruction
  - pas de `fne` malheureusement
- `s`: taille du flottant (ou `d`, ou `q`)
- `rd`: registre général: 0 si faux et 1 si vrai
  - Pour brancher, on teste `rd` ensuite avec un *branch* classique
- `rs1` et `rs2`: les deux registres flottants à comparer

Classification des flottants

- `fclass.s rd, rs1`
- Range dans le registre général `rd` un nombre correspondant à la nature du registre flottant `rs1`

rd	rs1
0	$-\infty$
1	nombre normalisé négatif
2	nombre dénormalisé négatif
3	-0
4	$+0$
5	nombre dénormalisé positif
6	nombre normalisé positif
7	$+\infty$
8	NaN avertisseur
9	NaN silencieux

Appels systèmes RARS

Permet de faire des entrées-sorties avec des flottants

- `PrintFloat`: `a7`=2
  - `fa0`: nombre à afficher
- `PrintDouble`; `a7`=3
  - `fa0`: nombre à afficher
- `ReadFloat`: `a7`=6
  - Nombre lu stocké dans `fa0`
- `ReadDouble`: `a7`=7
  - Nombre lu stocké dans `fa0`

Exercice

Écrire un programme `c2f.s`

- Qui convertit un température en degré Celsius vers des degrés Fahrenheit

Rappel: $f = (c \times 1.8) + 32$

- Afficher « Trop froid! » si la température est inférieure à 82°F
  - En lab

Autres architectures

- Avoir des registres flottants distincts des généraux est classique
- Des instructions mathématiques supplémentaire sont souvent disponibles: logarithmes, trigonométrie, etc. (IEEE 754 en spécifie une quarantaine optionnelles)

Programmer avec les flottants

Programmer avec IEEE 754

IEEE 754: Beaucoup de puissance et de détails

- Règles d'arrondis configurables
- NaN silencieux et avertisseurs
- Flags d'exceptions

Support des architectures, langages et bibliothèques

- Raisonnablement supporté par les processeurs (dont RISC-V)
  - Avec beaucoup de détails
- Habituellement supportés par les langages
  - Mais accès parfois limité aux facilité avancés (arrondis...)
  - $\rightarrow$ Extension de compilateurs, bibliothèques spécialisés, etc.

Programmer avec les flottants

Problème

- Les résultats sont arrondis (ou mis à une valeur spéciale)
- Les pertes de précisions se combinent
- $\rightarrow$ Le développeur doit en être conscient
- Et programmer en en prenant compte
- L'**analyse numérique** est un sous-domaine de l'informatique en soit

Ressources

- « What every computer scientist should know about floating-point arithmetic » — David Goldberg, 1991 (44 pages)
  - Version HTML
- « Handbook of Floating-Point Arithmetic » — Jean-Michel Muller et al., 2e édition 2018 (627 pages)

Non-associativité

Opérations flottantes

- L'arrondi a lieu après chaque opération flottante
- Conséquence: les opérations simples (comme l'addition) ne sont pas associative
  - $(a + b) + c \neq a + (b + c)$

Exemple: les gros mangent les petits

- (1e30 + -1e30) + 1 = ?
- 1e30 + (-1e30 + 1) = ?

Solution

- Traiter les nombres par groupe de magnitude
- Maintenir l'information de l'erreur

Garanties algorithmiques

Exemple: algorithme *2sum*

- Faire la somme de deux flottants a et b
- Récupérer le résultat (arrondi) $s = fl(a + b)$
- Calculer l'erreur (exacte) $t = a + b - s$

```
// 2sum en Java
double a, b;
// ...
double s = a + b;
double ax = s - b;
double bx = s - ax;
double da = a - ax;
double dx = b - bx;
double t = da + db;
```

- Fonctionnement prouvé (sous conditions)

Comparaison

Comparer c'est se tromper

- La comparer par l'égalité de deux flottants n'est pas robuste
- 0.37 + 0.2 - 0.57 == 0 → ?
- ... et c'est sans compter les surprises de NaN

Solution

- Comparer les distance
- `Math.abs(0.37 + 0.2 - 0.57) < epsilon`
- où `epsilon` est une constante (petite) sémantique au programme

Variations architecturales

Processeurs

- Des processeurs utilisent des représentation étendues
- Exemple : x86 stocke les flottants sur 80 bits (plus de précision)

Effet

- En fonction du processeur, un programme peut avoir des résultats différents
- $\rightarrow$ Problèmes de reproductibilité, bugs subtils, etc.

Utiliser la bonne représentation

Flottants

- Limiter l'utilisation des flottants aux bon domaines
  - Analyse numérique, physique, 3D, etc.
- Utiliser des doubles
  - Voire des grand flottants (`BigDecimal` de Java)
- Prudence vis-à-vis de la perte de précision
  - Ne faire confiance à personne

Entiers exacts

- Adapter l'unité si besoin
- Entiers processeur si les entiers sont petits
- Bibliothèques si les entiers sont très très grands (`BigInteger` de Java)

Autre représentations

Généralement fournies par bibliothèques, parfois par le langage

Fractions

- Deux entiers : un numérateur et un dénominateur
- Idéal pour l'arithmétique classique

Point fixe

- La magnitude est fixe
- Bonus: protection (parfois) des erreurs de précision

Arithmétique d'intervalle

- Deux flottants: une borne minimum et un borne maximum
- Alternative: une valeur et une erreur

Représentation symbolique

- On ne manipule pas des nombres mais des formules
- Systèmes algébriques: sympy, maple, mathematica, etc.

Conclusion

Résumé

Représentation des flottants

- Norme IEEE 754 utilisée quasiment partout
- Essaye d'être robuste et généraliste, mais nombreuses subtilités

RISC-V

- Support complet de la norme
- Jeu d'instructions nécessairement complexe

Programmation

- Faites attention aux surprises
- Exactitude et précision
  - Beaucoup de rigueur nécessaire
  - Sous-domaine spécifique

La prochaine fois

- Entrés-sorties
- Interruptions
- Appels systèmes