Vírgula Flutuante

António de Brito Ferrari
ferrari@ua.pt

Representação de números reais

- Necessário representar números muito pequenos e muito grandes
- Notação científica:
  - ❖-2.34×10^{56} ← normalizada
  - ❖$+0.002 \times 10^{-4}$ ← não normalizada
  - ❖$+987.02 \times 10^{9}$ ← não normalizada
- Em binário
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Tipos `float` e `double` em C

Representação em vírgula flutuante

- X = m * b^{exp} X = (m, exp)
  - m – *significando* (tambem designado *mantissa*)
  - exp – *expoente*
  - b – base (implícita – desnessário incçuí-la na representação dos números)
- Objetivo: máxima precisão com o nº de bits disponível
  - Significando normalizado
  - Base 2

1. O standard IEEE para Vírgula Flutuante

Formato de representação

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
Significando Normalizado: 1.0 ≤ significando < 2.0
Bit à esquerda da vírgula sempre 1- desnecessário explicitá-lo (***hidden bit***)
Significando = 1, Fraction
Expoente: representação em excesso:
Valor do expoente = Exponent + Bias - Exponent unsigned
Single: Bias = 127; Double: Bias = 1023

Precisão simples: Gama de representação

- Expoentes 00000000 e 11111111 reservados
- Menor valor representável
  - Exponent: 00000001
    ⇒ valor efetivo = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significando = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Maior valor representável
  - Exponent: 11111110
    ⇒ valor efetivo = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significando ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Precisão dupla: Gama de representação

- Expoentes 0000…00 and 1111…11 reservados
- Menor valor representável
  - Expoente: 00000000001
    $\Rightarrow$ valor efetivo = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significando = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Maior valor representável
  - Expoente: 11111111110
    $\Rightarrow$ valor efetivo = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significando ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Precisão da representação

- Erro Relativo
  - Todos os bits do significando são significativos
  - Single: approx 2^{-23}
    - Equivalente a $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ dígitos decimais de precisão
  - Double: approx 2^{-52}
    - Equivalente a $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ dígitos decimais de precisão

Conversão Decimal para FP

- Representar **–0.75**

  $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$

  S = 1

  Fraction = $1000\ldots00_2$

  Expoente = –1 + Bias
  - Single: $-1 + 127 = 126 = 01111110_2$
  - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Conversão Fl. Pt. para Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- $(-1)^S$ x (1+Significand) x $2^{(\text{Exponent-Bias})}$
- Sinal: 0 => $(-1)^0$ = 1 => positivo
- Expoente: $0110\ 1000_2$ = $\mathbf{104_{10}}$
  - Ajuste do Bias: 104 - 127 = -13 representa $\mathbf{2^{-13}}$
- Fração:
  - Expoente ≠ 0000 0000 - *hidden bit*

1.101 0101 0100 0011 0100 0010

ABF AC I - FP 10

Conversão Fl. Pt. para Decimal (2)

- Que numero é representado, em precisão simples, por

  1100000010100…00
  - S = 1
  - Fraction = $01000…00_2$
  - Expoente = 10000001_2 = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

ABF AC I - FP 11

Conversão Fl. Pt. para Decimal (3)

0	0110 1000	101 0101 0100 0011 0100 0010

- Significando: $1 + (s1x2^{-1}) + (s2x2^{-2}) + ...$

  $1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$

  = 1+ 1/2 + 1/8 + 1/32 +1/128 + 1/512 + 1/16384 + 1/32768 + 1/131072 + 1/4194304

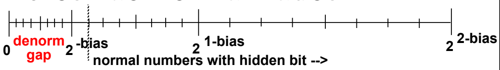
  = 1.0 + (2097152 + 524288 + 131072 + 32768 + 8192 + 256 + 128 + 32 + 1)/4194304

  = 1.0 + (2793889)/4194304

  = 1.0 + 0.66612
- Valor = $+1.66612_{10}*2^{-13}$ ($\sim +2.034*10^{-4}$)

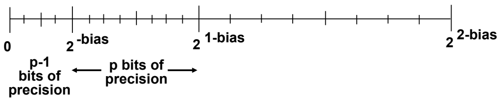
ABF AC I - FP 12

Números não-normalizados



Intervalo entre *0* e o número seguinte representavel >> Intervalos entre os outros numeros vizinhos.

standard IEEE usa ***denormalized numbers*** para preencher esse intervalo



Números não-normalizados

- **Expoente = 000...0** ⇒ hidden bit é 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Números mais pequenos que os normalizados
  permite *underflow* gradual (com precisão reduzida)
- Denormal com fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Duas representações de 0.0!

Infinitos e NaNs (Not a Number)

- **Exponent = 111...1, Fraction = 000...0**
  - ±Infinity
  - Pode ser usado em cálculos subsequentes dispensando "overflow check"
- **Exponent = 111...1, Fraction ≠ 000...0**
  - Not-a-Number (NaN)
  - Indica resultado ilegal ou indefinido (p.ex. 0.0 / 0.0)
  - Pode ser usado em cálculos subsequentes

2. Operações em Vírgula Flutuante

Adição em Vírgula Flutuante

Exemplo: representação decimal com 4-dígitos

$9.999 \times 10^{1} + 1.610 \times 10^{-1}$

1. Alinhar os operandos
   - Shift à direita numero com o menor expoente

   $9.999 \times 10^{1} + 0.016 \times 10^{1}$
2. Somar os significandos

   $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
3. Normalizar o resultado & check for over/underflow

   1.0015×10^{2}
4. Arredondar e renormalizar se necessário

   1.002×10^{2}

Dígito de guarda e dígito de arredondamento

- Representação com 3 dígitos:

$2.56_{10} * 10^{0} + 2.34_{10} * 10^{2}$

2.3400

+ 0.0256 operação com 5 dígitos

2.3656 necessário para manter precisão

Guard digit **Round digit**

Algoritmo de Adição em Vírgula Flutuante

(1) calcular Ye - Xe

(2) shift à direita de Xm de (Ye - Xe) posições para obter Xm $2^{(Ye - Xe)}$

(3) Cálculo de Xm $2^{(Ye - Xe)}$ + Ym

(4) Se Xm $2^{(Ye - Xe)}$ + Ym = 0 Expoente Resultado = 0 *End*
Se Xm $2^{(Ye - Xe)}$ + Ym não normalizado, então:
repetir
shift à esquerda do resultado, decrementando o expoente (e.g., 0.001xx…)
shift à direita do resultado, incrementando o expoente (e.g., 10.1xx…)
até MSB do resultado = 1 **(NOTA: Hidden bit no Standard IEEE)**

(5) Se *overflow* **ou** *underflow* gerar excepção
senão arredondar resultado

6) Se resultado não normalizado *goto 4*

ABF AC 1 - FP 19

Adição em Vírgula Flutuante Exemplo

Representação binária com 4-dígitos

$\mathbf{1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}}$ (0.5 + –0.4375)

1. Alinhar os operandos
   - Shift à direita numero com o menor expoente

   $\mathbf{1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}}$
2. Somar significandos

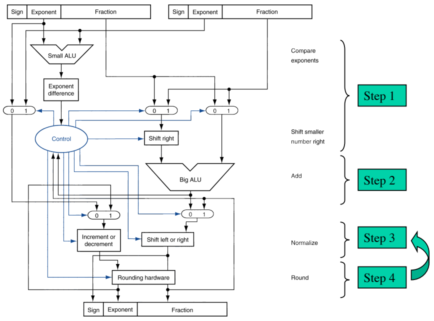
   $\mathbf{1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}}$
3. Normalizar resultado & check for over/underflow

   $\mathbf{1.000_2 \times 2^{-4}}$
4. Arredondar e renormalizar se necessário

   $\mathbf{1.000_2 \times 2^{-4}}$ = 0.0625

ABF AC 1 - FP 20

Floating-Point Adder



ABF AC 1 - FP 21

Floating-Point Adder

- Muito mais complexo que o somador para inteiros
- Execução da adição em vários ciclos de relógio
  - Muito mais lento que operações com inteiros
  - Pode ser pipelined

ABF AC 1 - FP 22

Multiplicação em Vírgula Flutuante

- Exemplo: operandos em binário, 4-bits
  $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

1. Somar os expoentes
   - Unbiased: $-1 + -2 = -3$
   - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
2. Multiplicar significandos
   $1.000_2 \times 1.110_2$ $= 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
3. Normalizar o resultado & check for over/underflow
   $1.110_2 \times 2^{-3}$
4. Arredondar e renormalizar se necessario
   $1.110_2 \times 2^{-3}$
5. Determinar o sinal: $+ve \times -ve \Rightarrow -ve$
   $-1.110_2 \times 2^{-3}$ $= -0.21875$

ABF AC 1 - FP 23

Algoritmo de Multiplicação

(1) calcular Ye + Xe – 127 (Nota: Ye + Xe = exp(X) + exp(Y) + 127 + 127)

Se *overflow* ou *underflow* *gerar excepção*

(2) Multiplicar significandos Xm * Ym

(3) Se o resultado requer normalização, então:

shift à direita do resultado, incrementando o expoente (e.g., 10.1xx…)

Se *overflow* *gerar excepção*

(5) Arredondar resultado

(6) Se resultado não normalizado *goto 3*

(7) Sinal do resultado = Ys *XOR* Xs

ABF AC 1 - FP 24

Unidade de Multiplicação em Vírgula Flutuante

- Complexidade semelhante à do FP adder
  - usa um multiplicador para os significandos em lugar de um somador
- Unidade de Processamento em Vírgula Flutuante:
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- As operações são executadas em mais do que um ciclo de relógio
  - Pode ser pipelined

ABF AC I - FP 25

Arredondamento

resultado normalizado, mas existem dígitos não nulos para a direita do significando --> o numero deve ser arredondado

E.g., B = 10, p = 3:

$$\boxed{0}\boxed{2}\boxed{1.69} = 1.69\underline{00} * 10^{2-bias}$$

$$- \boxed{0}\boxed{0}\boxed{7.85} = -.07\underline{85} * 10^{2-bias}$$

$$\boxed{0}\boxed{2}\boxed{1.61} = 1.61\underline{15} * 10^{2-bias}$$

Um "round digit" deve ser conservado à direita do "guard digit" para permitir que depois de um shift à esquerda para normalização o resultado possa ser correctamente arredondado

Standard IEEE: **round to nearest (default)**
round towards plus infinity
round towards minus infinity
round towards 0

round to nearest: ***minimiza erro médio do arredondamento***

round digit < B/2: truncate
> B/2: round up (add 1 to *ULP*: unit in last place)
= B/2: round to nearest even digit

ABF AC I - FP 26

Sticky Bit

Bit adicional à direita do "round digit"

```
+ d0 . d1 d2 d3 . . . dp-1  0  0  0   <-- Sticky bit: set to 1 if any 1 bits fall off
  0 .  0  0  X . . .  X    X  X  S        the end of the round digit
                           X  X  S

- d0 . d1 d2 d3 . . . dp-1  0  0  0      - d0 . d1 d2 d3 . . . dp-1  0  0  0
  0 .  0  0  X . . .  X    X  X  0          0 .  0  0  X . . .  X    X  X  1
                           X  X  0                    gera um borrow
```

Rounding Summary:

Raiz 2 minimiza perdas de precisão

+,-,*,/ requerem um carry/borrow bit + um *guard digit*

Um *round digit* necessário para arredondamento correcto

Sticky bit **necessário quando round digit = B/2**

Rounding to nearest **tem erro médio nulo assumindo uma distribuição uniforme dos digitos**

ABF AC I - FP 27

3. Suporte ao processamento em vírgula flutuante na arquitetura MIPS

MIPS Floating Point

- Unidade de Vírgula Flutuante: coprocessador ***C1***
- Versões Single Precision e Double Precision de add, subtract, multiply, divide, compare
  - Single **`add.s`**, **`sub.s`**, **`mul.s`**, **`div.s`**, **`c.lt.s`**
  - Double **`add.d`**, **`sub.d`**, **`mul.d`**, **`div.d`**, **`c.lt.d`**
- Registos? operações Int. e Fl.Pt. sobre diferentes dados - registos Fl.Pt. distintos para melhor performance:
  - 32-bit Fl. Pt. reg: `$f0`, `$f1`, `$f2` ..., `$f31`
  - Double Precision: pares de registos atuam como 64-bit reg. `$f0`, `$f2`, `$f4`, ...

MIPS: Instruções F.P.

- add.s $f0, $f1, $f2 — Fl. Pt. Add (single)
- add.d $f0, $f2, $f4 — Fl. Pt. Add (double)
- sub.s $f0, $f1, $f2 — Fl. Pt. Subtract (single)
- sub.d $f0, $f2, $f4 — Fl. Pt. Subtract (double)
- mul.s $f0, $f1, $f2 — Fl. Pt. Multiply (single)
- mul.d $f0, $f2, $f4 — Fl. Pt. Multiply (double)
- div.s $f0, $f1, $f2 — Fl. Pt. Divide (single)
- div.d $f0, $f2, $f4 — Fl. Pt. Divide (double)
- sqrt.s $f0, $f1, $f2 — Fl. Pt. Square Root (single)
- Sqrt.d $f0, $f4, $f2 — Fl. Pt. Square Root (double)
- abs.s, abs.d — Valor absoluto
- c.X.s $f0, $f1 — Fl. Pt.Compare (single)
- c.X.d $f0, $f2 — Fl. Pt.Compare (double)
- bc1t — Fl. Pt. Branch true
- bc1f — Fl. Pt. Branch false

MIPS: Instruções F.P. (2)

- Transferência Fl. Pt. reg. - Fl. Pt. reg. : ***mov.s, mov.d***
- Transferência Fl. Pt. reg. - memória : ***`lwc1, swc1`***
  Pseudoinstruções: ***l.d*** ***s.d***
- Transferência Fl. Pt. reg. - g.p.r.: ***`mfc1`*** \$r2, \$f1
- Conversão
  Single - Double: ***`cvt.d.s`***
  Double - Single: ***`cvt.s.d`***
  Single - Integer: ***`cvt.w.s`***
  Double - Integer: ***`cvt.w.d`***
  Integer - Single: ***`cvt.s.w`***
  Integer - Double: ***`cvt.d.w`***

F.P. Compare e Branch

- Inteiros: set-on-less-than e branch

slt \$t0, \$t1, \$t2 – se \$t1 < \$t2 \$t3 = 1 senão \$t3 = 0

beq \$t3, label – efetua o salto se \$t3 = 0 (bne se \$t3 ≠ 0)

- Floating-Point:

c.X.s (.d) \$f0, \$f2 – coloca o bit ***CC*** (condition code) de um registo de controle a 1 se a condição (eq, le, lt) for verdadeira, a 0 se for falsa

bclt (bclf) - efetua o salto se ***CC*** = 1 (bclt), 0 (bclf)

Exemplo: Multiplicação de Matrizes X = Y * Z

- Os endereços iniciais das matrizes X, Y e Z são os parâmetros passados em \$a0, \$a1, e \$a2 (*passagem por referência*)
- Variáveis inteiras **i, j** e **k** em \$s0, \$s1, e \$s2. Arrays de 32 por 32
- Usar pseudoinstructions:
  - **`li`** (load immediate)
  - **`l.d`** / **`s.d`** (load/store 64 bits)

X = X + Y * Z

```
void mm (double x[][], double y[][], double z[][])
{
  int i, j, k;
  for (i=0; i!=32; i=i+1)
    for (j=0; j!=32; j=j+1)
      for (k=0; k!=32; k=k+1)
          x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```


1. Inicialisar as variáveis de controle de ciclo

```
mm:   ...
      li     $t1, 32        # $t1 = 32
      li     $s0, 0         # i = 0; 1st loop
L1:   li     $s1, 0         # j = 0; reset 2nd
L2:   li     $s2, 0         # k = 0; reset 3rd
```

2. To fetch x[i][j], saltar i linhas (i*32), somar j

```
      sll    $t2, $s0, 5    # $t2 = i * 2^5
      addu   $t2, $t2, $s1  # $t2 = i*2^5 + j
```

3. Get byte address (8 bytes), load x[i][j]

```
      sll    $t2, $t2, 3    # i, j byte addr.
      addu   $t2, $a0, $t2  # @ x[i][j]
      l.d    $f4, 0($t2)    # $f4 = x[i][j]
```


4. Load z[k][j] em $f16

```
L3:   sll    $t0, $s2, 5    # $t0 = k * 2^5
      addu   $t0, $t0, $t4  # $t0 = k*2^5 + j
      sll    $t0, $t0, 3    # k, j byte addr.
      addu   $t0, $a2, $t0  # @ z[k][j]
      l.d    $f16, 0($t0)   # $f16 = z[k][j]
```

5. Load y[i][k] em $f18

```
      sll    $t0, $s0, 5    # $t0 = i * 2^5
      addu   $t0,$t0,$t5    # $t0 = i*2^5 + k
      sll    $t0, $t0, 3    # i, k byte addr.
      addu   $t0, $a1, $t0  # @ y[i][k]
      l.d    $f18, 0($t0)   # $f18 = y[i][k]
```

$f4: x[i][j], $f16: z[k][j], $f18: y[i][k]

6. y*z + x

```
mul.d $f16, $f18, $f16      # y[][]*z[][]
add.d $f4, $f4, $f16        # x[][]+ y*z
```

7. Incrementar k; se fim do ciclo K, store x

```
addiu $s2, $s2, 1   # k = k + 1
bne   $t5, $t1, L3  # if (k!=32) goto L3
s.d   $f4, 0($t2)   # x[i][j] = $f4
```

8. Incrementar j; ciclo intermédio se j < 32

```
addiu $s1, $s1, 1   # j = j + 1
bne   $s1, $t1, L2  # if (j!=32) goto L2
```

9. Incrementar i; se i = 32, return

```
addiu $s0, $s0, 1   # i = i + 1
bne   $s0, $t1, L2  # if (i!=32) goto L1
jr    $ra
```

ABF AC I - FP 37