Vírgula Flutuante

António de Brito Ferrari
ferrari@ua.pt

Representação de números reais

- A representação de um número real em vírgula flutuante **(*floating point*)** é o método de representar uma aproximação do valor do real de modo a poder suportar um compromisso entre a gama de valores passíveis de serem representados e a respetiva precisão

6 dígitos + sinal:

<u>Vírgula fixa</u>:

Ex. parte inteira e parte decimal 3 dígitos cada:

Maior valor representável: 999,999

Erro (**<u>absoluto</u>**) da representação: 5×10^{-4}

<u>Vírgula flutuante</u>:

Ex. Expoente e significando 3 dígitos cada:

Maior valor representável: 10^{999}

Erro (**<u>relativo</u>**) da representação: 5×10^{-4}

Representação em vírgula flutuante

- Necessário representar números muito pequenos e muito grandes
- Notação científica:

-2.34×10^{56} ← normalizada

$+0.002 \times 10^{-4}$

$+987.02 \times 10^{9}$ ← 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* (m = 1.<u>xx…xx</u>)
  - exp – *expoente*

    fração ou ***mantissa***
  - b – base (implícita – desnessário incluí-la na representação dos números)
- **Gama de representação** – determinada pelo nº de bits do expoente
- **Precisão** – determinada pelo nº de bits do significando

1. Formatos de representação

STANDARD IEEE PARA VÍRGULA FLUTUANTE

IEEE 754

O standard especifica:

1. Os formatos vírgula flutuante de precisão simples e de precisão dupla, normalisados
2. As exceções do formato: ± 0, $\pm\infty$, desnormalisado, não números (NaN)
3. As operações de adição, subtração, multiplicação, divisão, raiz quadrada, resto e comparação
4. As conversões entre inteiros e vírgula flutuante
5. As conversões entre formatos de vírgula flutuante
6. As conversões entre vírgula flutuante e representação decimal
7. Os modos de arredondamento (muito importante)
8. As exceções e o modo como são tratadas

Formato de representação

Single Precision: 8 bits 23 bits
Double Precision: 11 bits 52 bits

S	Exponent	Fraction

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

S: sign bit (0 $\Rightarrow$ número >= 0; 1 $\Rightarrow$ número < 0)

Significando Normalizado: $1.0 \leq$ significando < 2.0

Bit à esquerda da vírgula sempre 1- desnecessário explicitá-lo (***hidden bit***) Significando = 1, Fraction

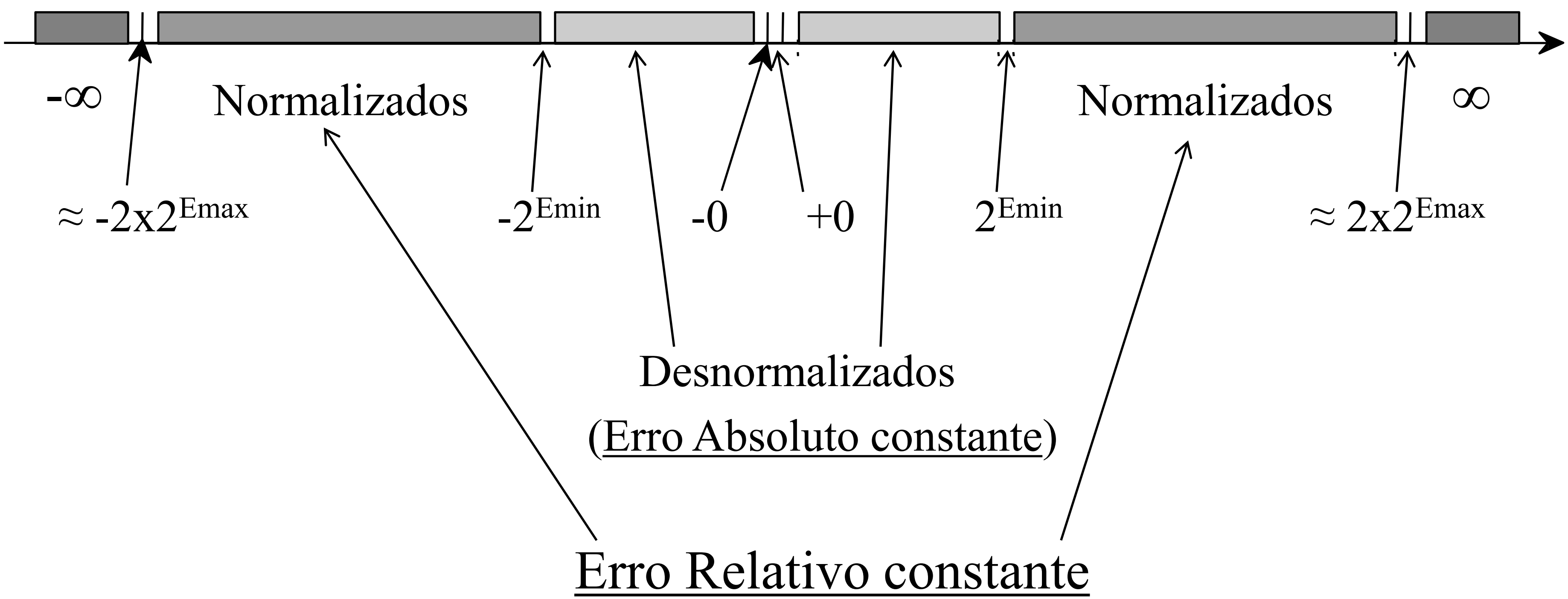
Expoente: representação em excesso:

Exponent = Valor do expoente + Bias - Exponent unsigned

Single: Bias = 127; Double: Bias = 1023

(Expoente em ***excesso*** permite a mesma lógica de comparação para inteiros e vírgula flutuante)

A Reta Real e a representação IEEE vírgula flutuante



IEEE Floating-Point

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Representação de valores "especiais"

Expoente	Fração	Valor representado
$e = E_{min} - 1 = -Bias$	$f = 0$	± 0
$e = E_{min} - 1 = -Bias$	$f \neq 0$	$0.f \times 2^{E_{min}}$
$E_{min} <= e <= E_{max}$	-	$1.f \times 2^{e}$
$e = E_{max} + 1$	$f = 0$	$\pm\infty$
$e = E_{max} + 1$	$f \neq 0$	NaN

Precisão simples: $E_{min} = -126$; $E_{max} = +127$

Precisão dupla: $E_{min} = -1022$; $E_{max} = +1023$

Desnormalizados: **e = -126** mas representado como **-127**

Precisão simples: Gama de representação

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

hidden-bit

- Representar **–0.75**

$0.75*2 = 1.5 \quad 0.5*2 = 1 \longrightarrow 0.75_{10} = 1.1_2$

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

S = 1 Fraction = $1000\ldots00_2$

Expoente = –1 + Bias

  - Single: –1 + 127 = 126 = 01111110_2
  - Double: –1 + 1023 = 1022 = 01111111110_2

- Single: 101111110 1000…00
- Double: 101111111110 1000…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: $0110\ 1000_2 = \mathbf{104}_{\mathit{10}}$
  - Ajuste do Bias: 104 - 127 = -13 representa $\mathbf{2^{-13}}$
- Fração:
  - Expoente ≠ 0000 0000 ➡ *hidden bit*

$\underline{1}.101\ 0101\ 0100\ 0011\ 0100\ 0010$

Conversão Fl. Pt. para Decimal (2)

- Que numero é representado, em precisão simples, por

  110000001010 00…00

  – S = 1

  – Fraction = 01000…00_2

  – Expoente = 10000001_2 = 129

- $x = (-1) \times (1 + 0.01_2) \times 2^{(129 - 127)}$

  $= (-1) \times (1 + 2^{-2}) \times 2^2$

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

  $= -5.0$

Conversão Fl. Pt. para Decimal (3)

0	0110 1000	101 0101 0100 0011 0100 0010

- Significando: $1 + (s_1 x 2^{-1}) + (s_2 x 2^{-2}) + \ldots$

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

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

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

  $= 1.0 + (2793889)/4194304$

  $= 1.0 + 0.66612$

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

Álgebra de valores “especiais”

$x > 0$
$y > 0$
$z > 0$

a	b	a + b	a * b	a ÷ b
0	0	0	0	NaN
0	y	z	0	0
0	∞	∞	NaN	0
x	0	z	0	∞
x	y	0, z ou ∞	0, z ou ∞	0, z ou ∞
x	∞	∞	∞	0
∞	0	∞	NaN	∞
∞	y	∞	∞	∞
∞	∞	∞	∞	NaN
∞	$-\infty$	NaN	$-\infty$	NaN
NaN	0	NaN	NaN	NaN
NaN	y	NaN	NaN	NaN
NaN	∞	NaN	NaN	NaN

Standard IEEE 754-1985
Incoerências

MaxN – maior valor representável

Operação	Valor teórico	Valor obtido
$b = a + a$	$b = \text{MaxN}+\text{MaxN} = \mathbf{2\ MaxN}$	$b = \boldsymbol{\infty}$
$c = b \div a$	$c = 2\ (\text{MaxN} \div \text{MaxN}) = \mathbf{2}$	$c = \boldsymbol{\infty}$
$d = 1 \div c$	$d = 1 \div 2 = \mathbf{0{,}5}$	$d = 1 \div \infty = \mathbf{0}$
$e = 1 \div (d - 0{,}5)$	$e = 1 \div (0{,}5 - 0{,}5) = \boldsymbol{\infty}$	$e = 1 \div (0 - 0{,}5) = \mathbf{-2}$

" *It makes me nervous to fly an airplane since I know they are designed using floating-point arithmetic* "
Anton Householder (1904-1993), matemático, analista numérico

Números não-normalizados

- **Expoente = 000...0** (= -Bias) ⇒ hidden bit é 0

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

- 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 Aritméticas

STANDARD IEEE PARA 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

   $$1.0015 \times 10^{2}$$

4. Arredondar e renormalizar se necessário

   $$1.002 \times 10^{2}$$

Arredondamento

- Representação com 4 dígitos:

$4{,}584 * 10^0 + 5{,}753*10^3$

$= (5{,}753 + 0{,}004584)*10^3$ (full-precision)

- Suponhamos a adição com 4 dígitos significativos:

$= (5{,}753 + 0{,}004)*10^3 = 5{,}757*10^3$

- Com "guard digit":

$= (5{,}753 + 0{,}0045)*10^3 = 5{,}7575*10^3 = 5{,}758*10^3$

➢ se parte remanescente > 5 somar 1, se < 5 truncar

Dígito de guarda e dígito de arredondamento

- Representação com 3 dígitos:

$$1.04_{10} * 10^2 - 9.52_{10} * 10^0$$

$$\begin{array}{r} 1.0400 \\ \underline{-\ 0.0952} \end{array} \quad \text{operação com 5 dígitos}$$

$$0.9448 * 10^2 = 9.448 * 10^1 \xrightarrow{\text{sem round digit}} 9.44 * 10^1$$

com round digit

$$9.45 * 10^1$$

Guard digit **Round digit**

➢ **Necessário 2 dígitos (guard e round) para manter precisão**

$Z = X + Y$

Algoritmo de Adição em Vírgula Flutuante

(1) calcular diferença dos expoentes Ye – Xe (para Ye > Xe**)**

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

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

(4) ***If*** **(**Xm $2^{(Ye - Xe)}$ + Ym = 0) **Resultado = 0** *End*

If **(**Xm $2^{(Ye - Xe)}$ + Ym) **não normalizado**

If **(**Carry out = 1) **shift à direita de Zm, incrementar Ze** *End*

repetir

shift à esquerda de Zm, decrementar Ze

até MSB do resultado = 1

(5) Se *overflow* **ou** *underflow* *gerar excepção*

senão arredondar resultado

6) Se resultado não normalizado *goto 4*

Adição em Vírgula Flutuante - Exemplo

Representação binária com 4-dígitos

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

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

   $1.000_2 \times 2^{-1} + (-0.111_2 \times 2^{-1})$

2. Somar significandos

   $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = (1.000_2 + 1.001_2) \times 2^{-1}$

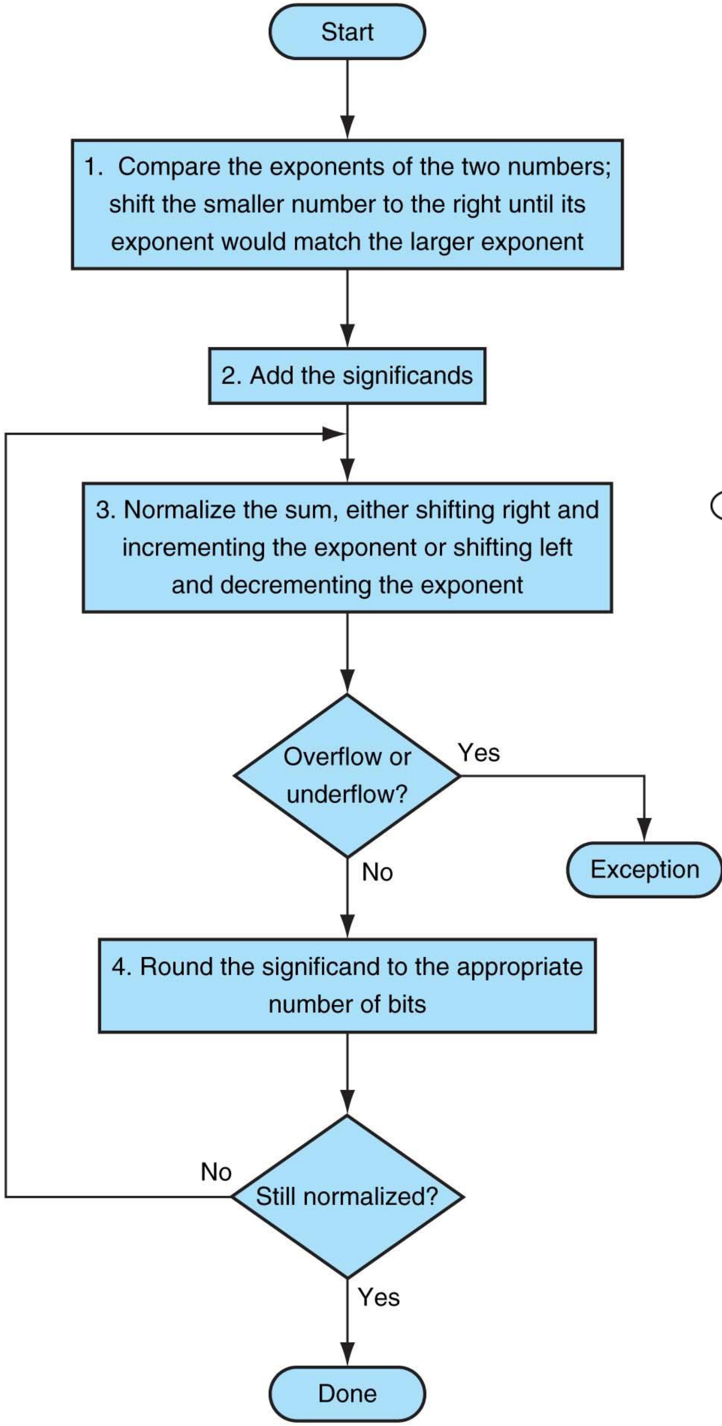
   $= 0.001_2 \times 2^{-1}$

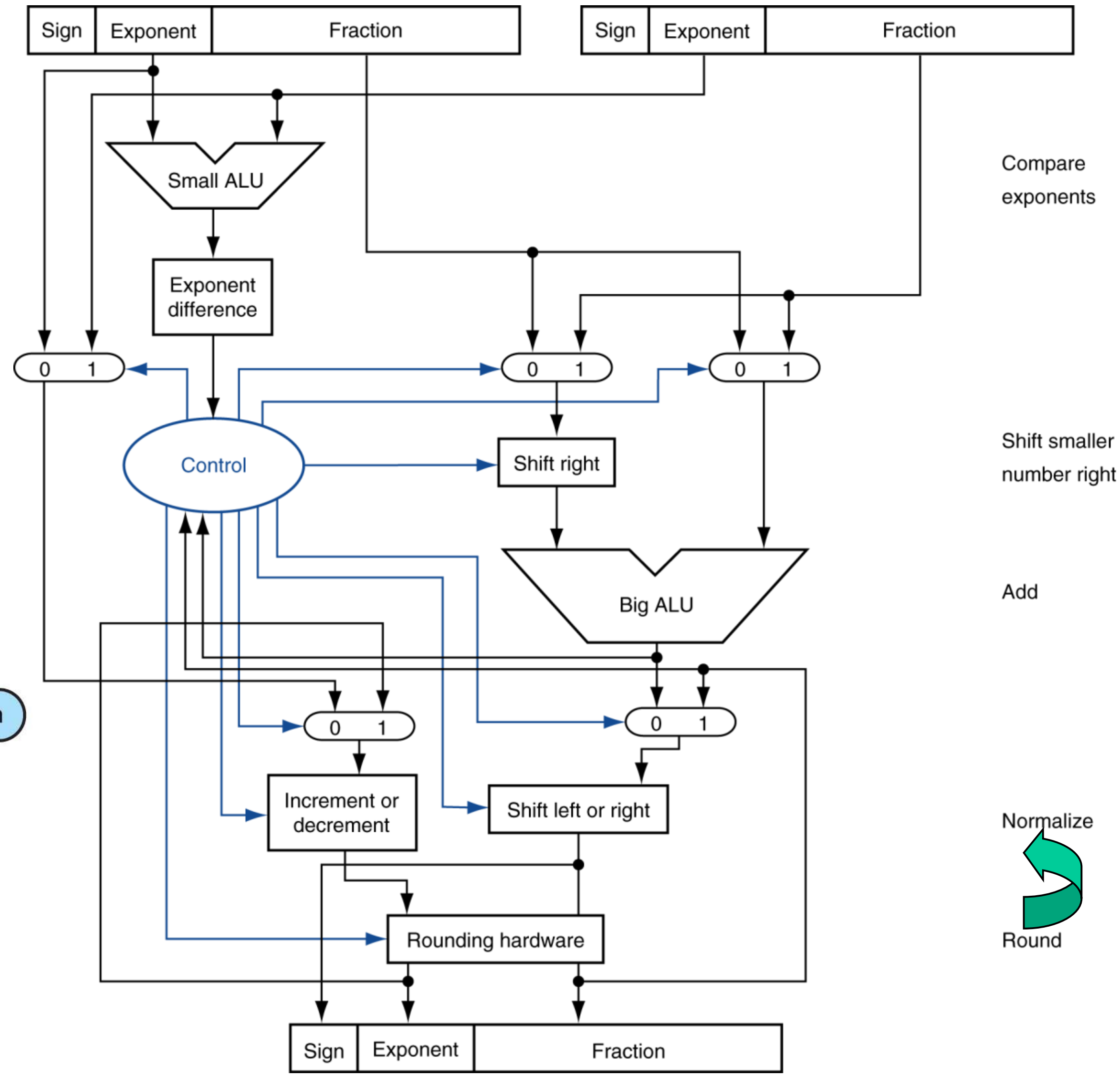
3. Normalizar resultado & check for over/underflow

   $1.000_2 \times 2^{-4}$

4. Arredondar e renormalizar se necessário

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


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round

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

Multiplicação em Vírgula Flutuante

- Exemplo: operandos em binário, 4-bits

  $\mathbf{1.001_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}}$ (0.5×-0.4375)

1. Somar os expoentes

   necessário subtrair *bias*

   - Unbiased: $-1 + -2 = -3$
   - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254$ **$-$ 127** $= -3 + 127$

2. Multiplicar significandos

   $\mathbf{1.001_2 \times 1.110_2} = 1.111110_2$

3. Normalizar o resultado & check for over/underflow

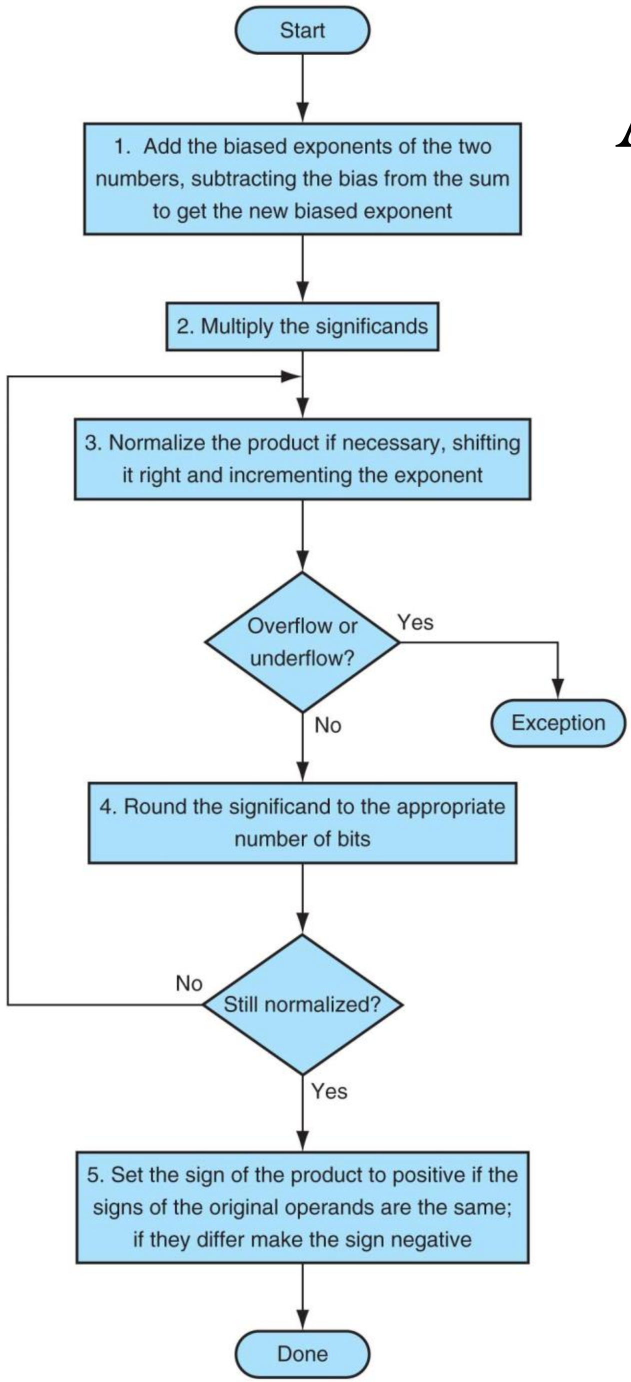
   $\mathbf{1.111110_2}$ (resultado normalizado)

4. Arredondar e renormalizar se necessário

   $1.111110_2 \Rightarrow \mathbf{10.000_2 \times 2^{-3}} \Rightarrow \mathbf{1.000_2 \times 2^{-2}}$

5. Determinar o sinal: **0** XOR **1** ⇒ **1**

   $\mathbf{-1.000_2 \times 2^{-2}} = -0.25$ 11111110100000000000000000000000



Algoritmo de Multiplicação

(1) calcular Ye + Xe – 127

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

(2) Multiplicar significandos Xm * Ym

(3) Se o resultado requer normalização

(4) 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**

Divisão Vírgula Flutuante

$$Z = s_1 x 2^{e_1} / s_2 x 2^{e_2} = (s_1 / s_2) x 2^{(e_1 - e_2)}$$

Subtrair expoente do divisor ao expoente do dividendo:

$Exp_1 - Exp_2 = (e_1 + 127) – (e_2 + 127) = e_1 – e_2$

$Exp_Z = (e_1 – e_2)$ **+ 127** – necessário somar o valor do *Bias*

$Significand_Z = (s_1 / s_2)$

$Sign_Z = sign_1 \text{ XOR } sign_2$

❖ Divisão menos frequente que Mult e Add/Sub – ser mais lenta afeta menos a velocidade de execução dos programas

Unidade de Vírgula Flutuante

- Complexidade de FP mult semelhante à do FP adder
  - usa um multiplicador para os significandos em lugar de um somador
- Unidade de Processamento em Vírgula Flutuante:
  - Adição, Subtração, Multiplicação, Divisão, Recíproco, Raiz Quadrada
  - FP $\leftrightarrow$ Integer conversion
- As operações são executadas em mais do que um ciclo de relógio (pode ser *pipelined*)

Arredondamento

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

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

	0	2	1.69	$= 1.69\underline{00} * 10^{2-bias}$
-	0	0	7.85	$= - .07\underline{85} * 10^{2-bias}$
	0	2	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

Sticky Bit

Bit adicional à direita do "round digit" (permite decidir se parte remanescente = 50 (*sticky bit* = 0) ou > 50 (*sticky bit* = 1)

\+ d0 . d1 d2 d3 . . . dp-1 0 0 0
0 . 0 0 X . . . X X X S
X X S

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

\- d0 . d1 d2 d3 . . . dp-1 0 0 0
0 . 0 0 X . . . X X X 0
X X 0

\- d0 . d1 d2 d3 . . . dp-1 0 0 0
0 . 0 0 X . . . X X X 1

gera um borrow

Rounding Summary:

Raiz 2 minimiza perdas de precisão

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

Um *round bit* necessário para arredondamento correcto

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

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

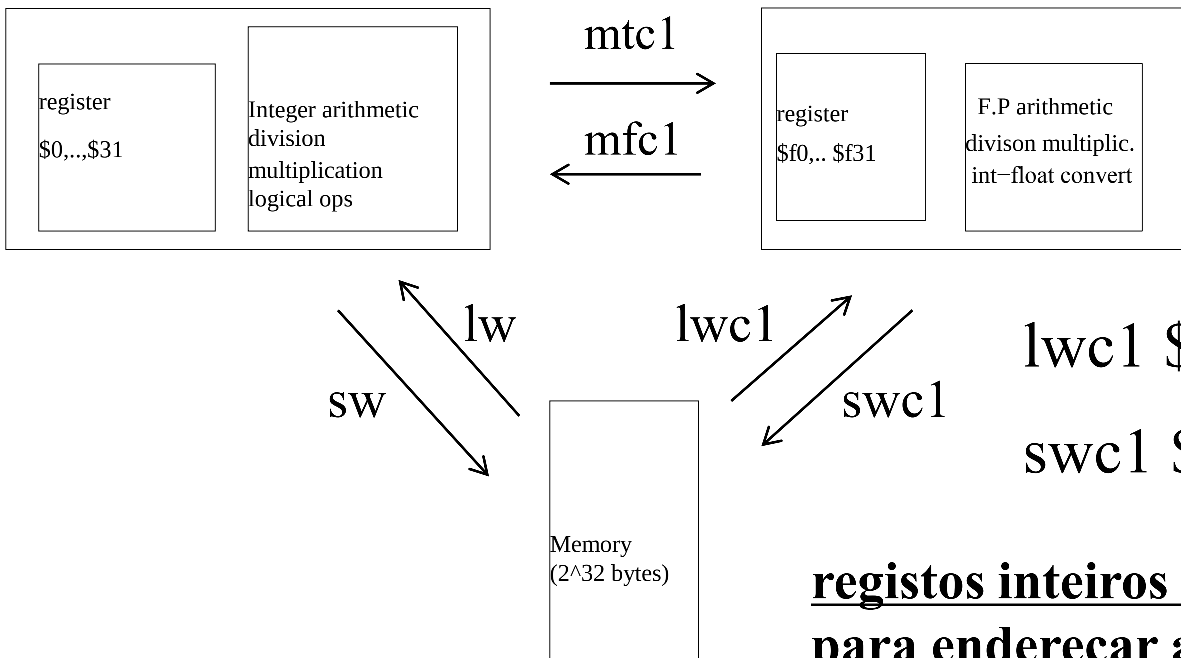
3. Suporte ao processamento em vírgula flutuante na arquitetura MIPS

Coprocessador de vírgula flutuante **C1**

- Coprocessador – processador usado para complementar as funções do processador central (CPU)
- MIPS: C0 – System Coprocessor
  C1 - Floating-Point Coprocessor
- C1
  - Originariamente num chip próprio (75000 transistores)
  - Atualmente integrado no mesmo chip do CPU (unidade aritmética de vírgula flutuante)

CPU (central processing unit)

FPU (floating point unit)
"coprocessor 1"



lwc1 $f_, x(**$r_**)

swc1 $f_, x(**$r_**)

<u>registos inteiros usados</u>
<u>para endereçar a memória</u>

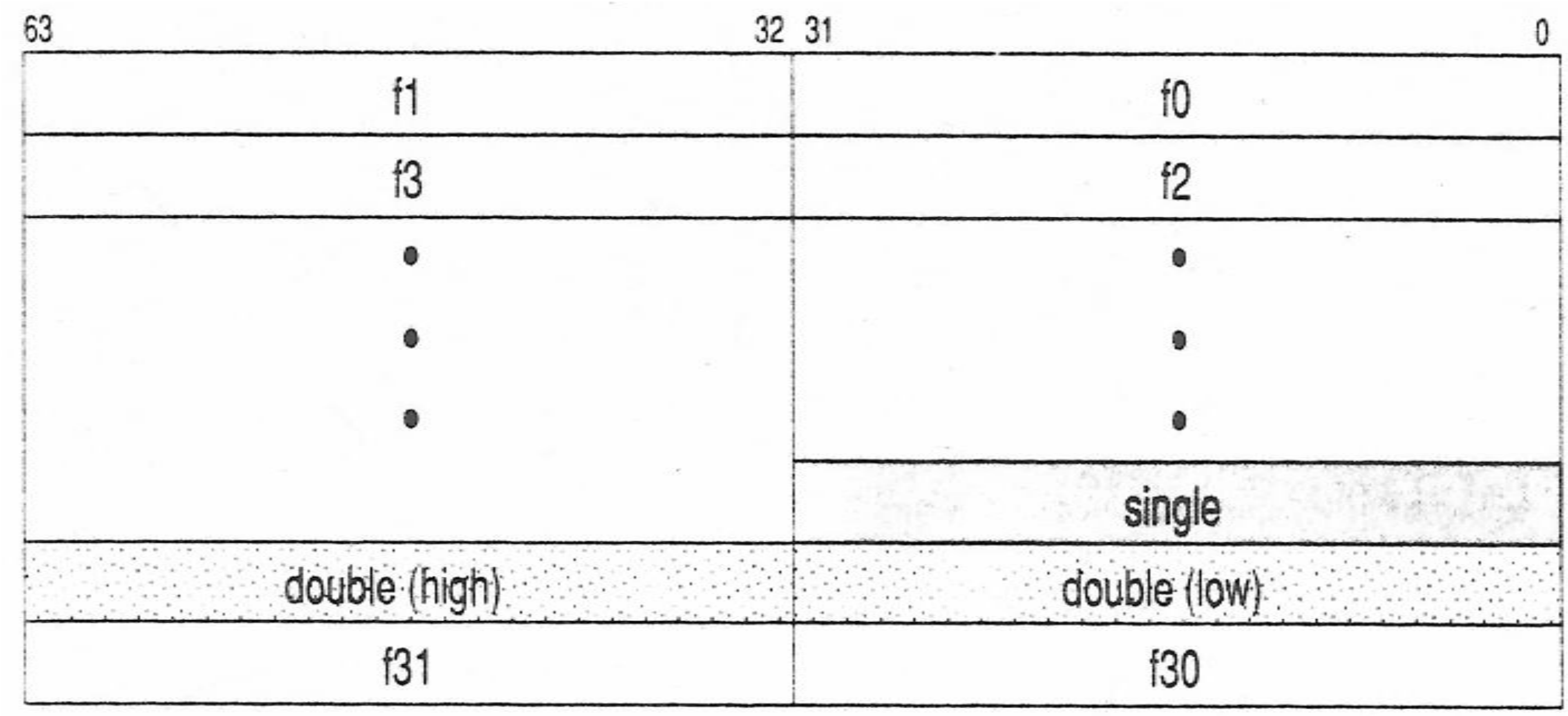
mtc1 $r_, $f_ # transfere do registo inteiro para o de vírgula flutuante

mfc1 $r_, $f_ # transfere do registo de vírgula flutuante para o reg. inteiro

MIPS Floating Point

- Unidade de Vírgula Flutuante: coprocessador ***C1***

Registos para operandos de vírgula flutuante:



Status Register

23		17	2	1 0
0	C	0	IEEE except.	rnd

condition bit

MIPS Floating Point

- 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`, …

Floating-Point Instruction Format

FR

op	cop	ft	fs	fd	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

`Opcode` = 17

source operands

Single-precision: `cop` = 16

destination operand

Double-precision: `cop` = 17

FI

op	rt	fs	immediate
6 bits	5 bits	5 bits	16 bits

Registo geral

Registo de C1

Floating-Point Registers (CP1)

Name	Register Number	Usage
`$fv0 - $fv1`	0, 2	return values
`$ft0 - $ft3`	4, 6, 8, 10	temporary variables
`$fa0 - $fa1`	12, 14	Function arguments
`$ft4 - $ft8`	16, 18	temporary variables
`$fs0 - $fs5`	20, 22, 24, 26, 28, 30	saved variables

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

X: eq, le, lt

MIPS: Instruções F.P. (2)

- Transferência Fl. Pt. reg. - Fl. Pt. reg. : ***mov.s, mov.d***
- Transferência Fl. Pt. reg. - memória : ***`lwc1`, `swc1`***
  sdc1 (Store double)

Pseudoinstruções: ***l.d*** ***s.d***

- Transferência Fl. Pt. reg. -> g.p.r.: ***`mfc1`*** \$r2, \$f1
- Transferência g.p.r. -> Fl. Pt. reg. : ***`mtc1`*** \$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 – compare: coloca o bit ***C*** (condition bit) do registo de status a 1 se a condição (**eq**, **ne, le**, **lt, ge, gt**) for verdadeira, a 0 se for falsa

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

Status Register

0	C	0	IEEE except.	rnd

FP: loads e stores

- Loads e stores:

FP register   Integer register

  - ***lwc1*** **$ft1, 42($s1)**
  - ***swc1*** **$fs2, 17($sp)**

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2, . . . , $f31	MIPS floating-point registers are used in pairs for double precision numbers.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[$s2 + 100] = $f1	32-bit data to memory
Conditional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

MIPS floating-point machine language

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

Exemplo: Multiplicação de Matrizes

$X = X + Y * Z$

```
void mm (double x[32][32], double y[32][32], double z[32][32])
{
  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];
}
```

Exemplo: Multiplicação de Matrizes $X = X + Y * Z$

- Os endereços base 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 as pseudoinstruções: **`l.d`** / **`s.d`**

  (load/store 64 bits – traduzidas pelo *assembler* em sequências de ***lwc1 / swc1***)
- **Nota**: em C Matrizes são armazenadas por linha (*row-major*)

Armazenamento na memória de matrizes quadradas *“row major”*

$$\&x[i][j] = \&x[0][0] + i*(32*8) + j*8$$
$$= \&x[0][0] + (i*32 + j)*8$$

Adress	Content
0	x[0][0]
8	x[0][1]
16	x[0][2]
…	…
31*8	x[0][31]
1*32*8	x[1][0]
	x[1][1]
	…
	x[1][31]
2*32*8	x[2][0]
	…
2*32*8 + 31*8	x[2][31]
	…
31*32*8 + 31*8	x[31][31]

1. Inicializar as variáveis de controle de ciclo

i = 0, j = 0, k = 0; $t0 = 32 (condição de terminação dos ciclos)

2. Multiplicar matrizes

```
do        # ciclo i
          do        # ciclo j – cálculo dos elementos da linha i
                    do        # ciclo k – cálculo do novo valor de x[i][j]
                       x[i][j] = x[i][j] + y[i][k] * z[k][j];
                       k = k +1
                    while k < 32
             j = j +1
          while j < 32
   i = i+1
while i < 32
return
```

1. Inicializar as variáveis de controle de ciclo

```
mm:     ...
        li      $t0, 32         # $t0 = 32
        li      $s0, 0          # i = 0; ciclo externo
L1:     li      $s1, 0          # j = 0; restart ciclo intermédio
L2:     li      $s2, 0          # k = 0; restart ciclo interno
```

2. Obter posição de x[i][j] - saltar i linhas (i*32), somar j

```
        sll     $t1, $s0, 5     # $t1 = i * 2^5
        addu    $t1, $t1, $s1   # $t1 = i*2^5 + j
        sll     $t1, $t1, 3     # $t1 = (i*2^5 + j)*2^3
```

3. Obter endereço e load x[i][j] (double – cada elemento representado em 8 bytes)

```
        addu    $t1, $a0, $t1           # Endereço de x[i][j]
        l.d     $f4, 0($t1)             # $f4 = x[i][j]
```

Ciclo K:

4. Load y[i][k] em $f6

```
L3:   sll   $t2, $s0, 5    # $t2 = i * 2^5
      addu  $t2, $t2, $s2  # $t2 = i*2^5 + k
      sll   $t2, $t2, 3    # $t2 = (i*2^5 + k)*2^3
      addu  $t2, $a1, $t2  # $t2 = Endereço de y[i][k]
      l.d   $f6, 0($t2)    # $f6 = y[i][k]
```

5. Load z[k][j] em $f8

```
      sll   $t3, $s2, 5    # $t3 = k * 2^5
      addu  $t3, $t3, $s1  # $t3 = k*2^5 + j
      sll   $t3, $t3, 3    # $t3 = (k*2^5 + j)*2^3
      addu  $t3, $a2, $t3  # $t3 = Endereço de z[k][j]
      l.d   $f8, 0($t0)    # $f16 = z[k][j]
```

$f4: x[i][j], $f6: y[i][k], $f8: z[k][j]

6. y*z + x

```
mul.d  $f6, $f8, $f6    # 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    $s2, $t0, 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, $t0, L2     # if (j!=32) goto L2
```

9. Incrementar i; se i = 32, return

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