

# Dizia eu que a aritmética...

## Arquitetura de Computadores

---

Emilio Francesquini  
[e.francesquini@ufabc.edu.br](mailto:e.francesquini@ufabc.edu.br)

2020.Q1

Centro de Matemática, Computação e Cognição  
Universidade Federal do ABC



- Estes slides foram preparados para o curso de **Arquitetura de Computadores** na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- O conteúdo destes slides foi **baseado no conteúdo do livro** *Computer Organization And Design: The Hardware/Software Interface*, 5<sup>th</sup> Edition.

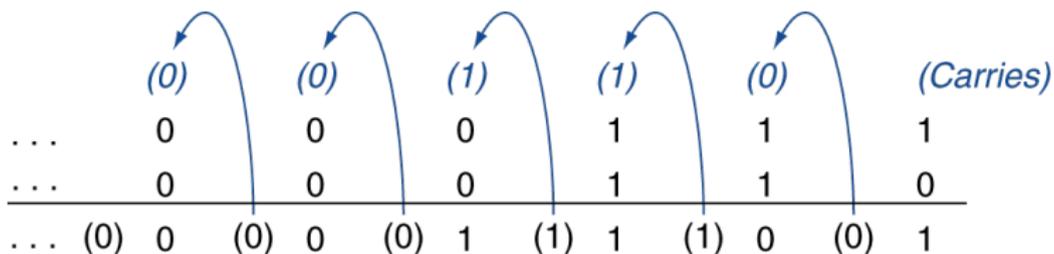


# Aritmética para computadores

---

- Operações em inteiros
  - ▶ Soma e subtração
  - ▶ Multiplicação e divisão
  - ▶ Lidando com overflow
- Números de ponto flutuante
  - ▶ Representação e operações

- Exemplo:  $7 + 6$



- Causa um *overflow* se o resultado estiver fora da faixa suportada
- A soma de operando +s e -s nunca causa overflow. Pq?
- Na adição de dois operandos positivos
  - ▶ há overflow se o sinal do resultado for negativo
- Na adição de dois operandos negativos
  - ▶ há overflow se o sinal do resultado for positivo

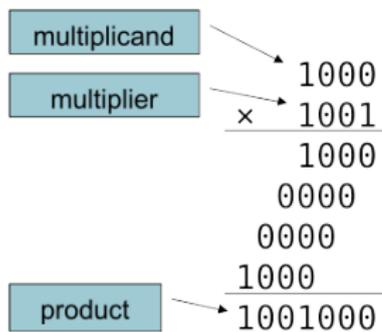
- Como usamos complemento de 2, o mais fácil é somar a negação do segundo operando!
- Exemplo:  $7 - 6 = 7 + (-6)$

$$\begin{array}{r} +7: 0000\ 0000\ \dots\ 0000\ 0111 \\ -6: 1111\ 1111\ \dots\ 1111\ 1010 \\ \hline +1: 0000\ 0000\ \dots\ 0000\ 0001 \end{array}$$

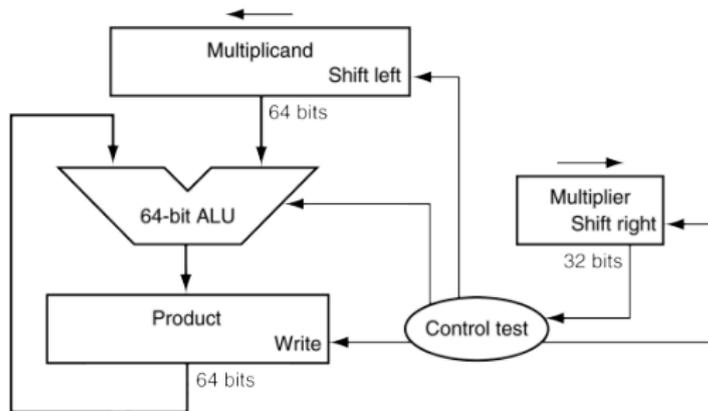
- Algumas linguagens simplesmente ignoram o overflow (ex. C).
  - ▶ Utilizam as instruções **addu** e **subu** do MIPS.
- Já outras linguagens como Ada e Fortran lançam exceções quando overflows ocorrem.
  - ▶ Utilizam as instruções **add**, **addi**, **sub** do MIPS.
  - ▶ Quando um overflow ocorre elas invocam o um tratador de exceções.
    - Salva o **PC** no registrador **EPC** (*exception program counter*).
    - Salta para o tratador de exceção pré-definido.
    - **mfc0** (*move from coprocessor reg*) pode ser usado para recuperar o valor de **EPC** e voltar ao ponto de erro logo ter corrigido o problema.

- Operações gráficas e de mídia normalmente trabalham com vetores de dados com 8 ou 16 bits
  - ▶ Pode-se utilizar um somador de 64 bits com a cadeia de "carry" particionada
    - Assim, pode-se operar em vetores de 8x8, 4x16 ou 2\*32 bits
  - ▶ **SIMD** - *Single Instruction Multiple Data*
- Operações com saturação
  - ▶ No caso de overflows, o resultado é sempre o maior valor representável
  - ▶ Exemplos: cortes de áudio, saturação em vídeo

Começamos com o algoritmo de papel e lápis...

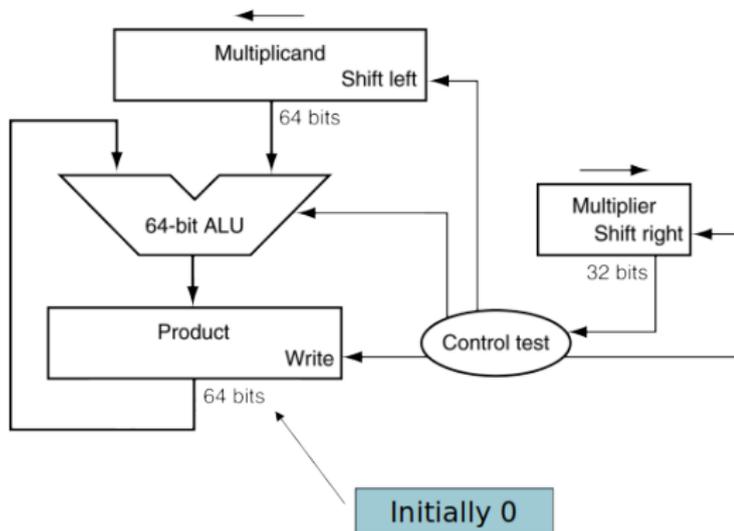
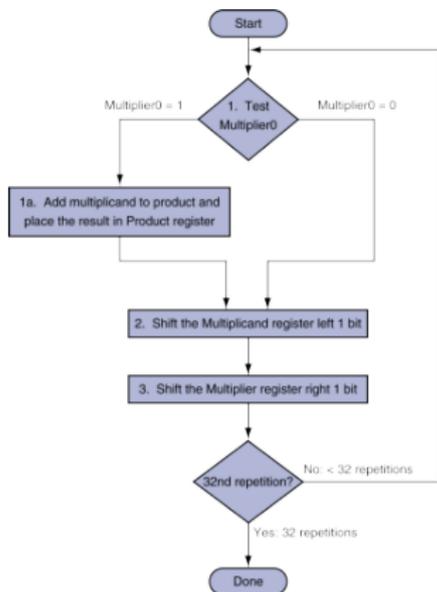


Length of product is the sum of operand lengths

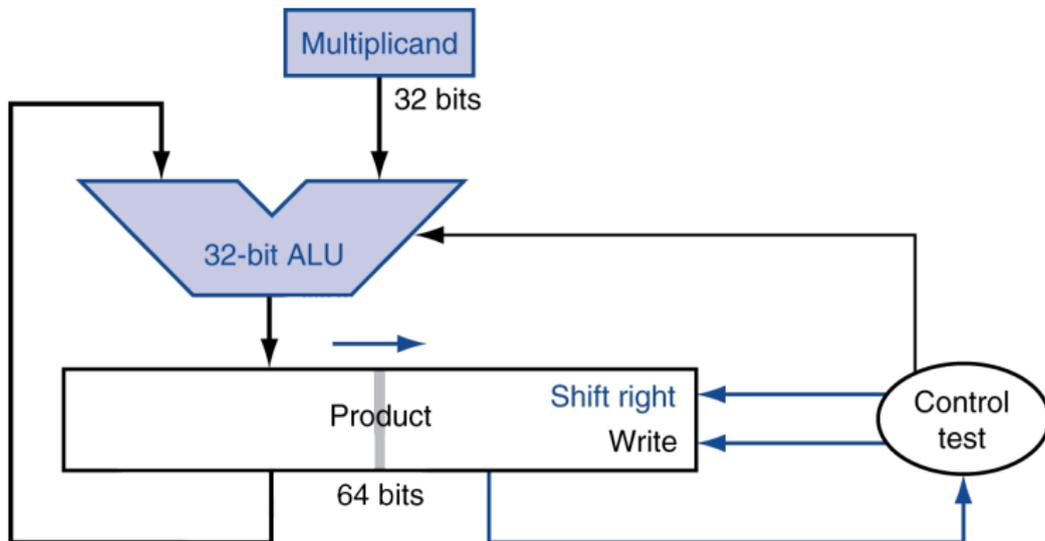


- $2 \times 3 = 6$
- 4 bits

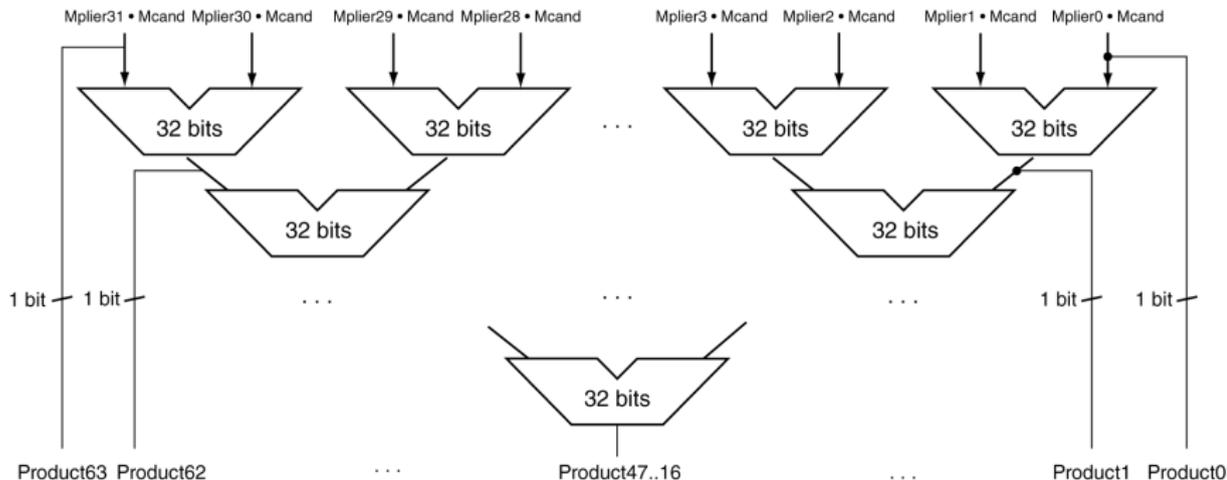
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mcand}$	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mcand}$	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: $0 \Rightarrow \text{No operation}$	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: $0 \Rightarrow \text{No operation}$	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110



- Desempenha vários passos em paralelo: soma/deslocamento
- Gasta um ciclo por adição do produto parcial
  - ▶ Não é tão grave se a frequência da ocorrência de multiplicações é baixa



- Utiliza diversos circuitos somadores
  - ▶ Compromisso (*tradeoff*) entre custo e desempenho



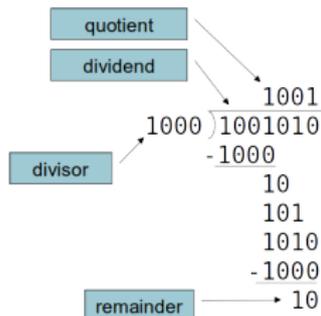
- Pode ser usado como pipeline para melhorar o desempenho através do paralelismo

- Utiliza dois registradores especiais de 32 bits para o produto
  - ▶ HI: 32 bits mais significativos
  - ▶ LO: 32 bits menos significativos
- Instruções disponíveis

---

```
1 # coloca o resultado do produto (64 bits) no HI/LO
2 mult rs, rt
3 multu rs, rt
4 # move resultados de HI/LO para rd
5 mfhi rd
6 mflo rd
7 # Multiplica rs por rt e guarda os 32 bits
8 # menos significativos em rd
9 mul rd, rs, rt
```

---

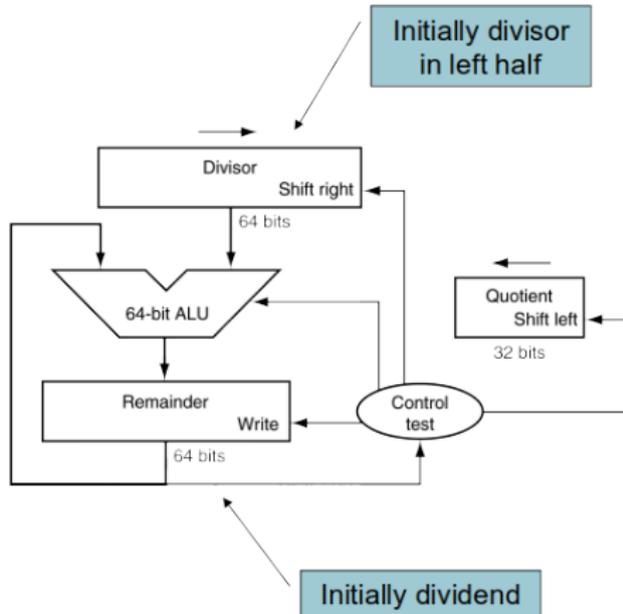
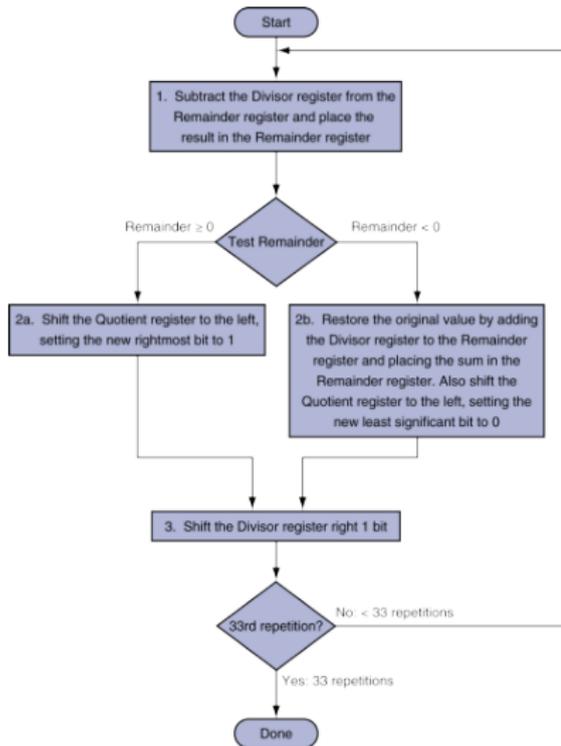


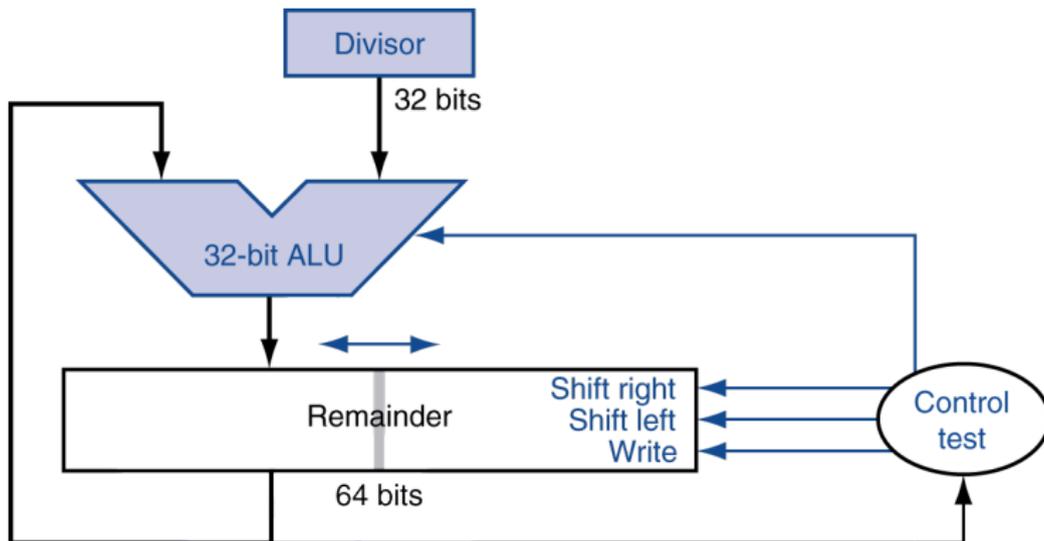
Operandos de  $n$  bits resultam em um quociente e resto de  $n$  bits.

- Verifica divisão por 0
- Abordagem de divisão papel e lápis
  - ▶ Se o número de bits do divisor  $\leq$  dividendo
    - 1 no quociente, subtrai
  - ▶ Caso contrário
    - 0 no quociente, "desce" o próximo bit
- Divisão com restauração
  - ▶ Faz a subtração e se o resto for menor que 0, adiciona o divisor de volta.
- Divisão com sinal
  - ▶ Divide-se utilizando os valores absolutos
  - ▶ Ajusta o sinal do quociente e do resto conforme necessário

- $7 / 2 = 3$  resto 1
- 4 bits

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	①110 0111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	①111 0111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	①111 1111
	2b: Rem < 0 $\Rightarrow$ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	①000 0011
	2a: Rem $\geq$ 0 $\Rightarrow$ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	①000 0001
	2a: Rem $\geq$ 0 $\Rightarrow$ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001





- Um ciclo a cada subtração para resto-parcial
- Parece muito com um multiplicador
  - ▶ De fato, o mesmo hardware pode ser utilizado por ambos!

- Infelizmente não é possível fazer um hardware paralelo como o da multiplicação
  - ▶ A subtração é condicionada ao sinal do resto
- Circuitos para divisão mais rápidos (por ex. SRT) geram múltiplos bits do quociente a cada passo.
  - ▶ Mas ainda assim necessitam de múltiplos passos.

- Utiliza os registradores **HI** e **LO** para os resultados
  - ▶ **HI**: Guarda o resto (32 bits)
  - ▶ **LO**: Guarda o quociente (32 bits)
- Instruções

---

1 **div** *rs*, *rt*  
2 **divu** *rs*, *rt*

---

- Não há verificação para os casos de divisão por 0 ou por overflow
  - ▶ É trabalho do software cuidar disto caso necessário
- Utiliza as instruções **mfhi** e **mflo** para acessar os resultados.

- Representam números não inteiros
  - ▶ Incluindo aqueles muito grandes ou muito pequenos.
- É muito semelhante à notação científica
  - $-2,34 \times 10^{56}$     normalizado
  - $+0,002 \times 10^{-4}$     não normalizado
  - $+987,02 \times 10^9$     não normalizado
- Em binário é a mesma coisa
  - ▶  $\pm 1,xxxx_2 \times 2^{yyy}$
- Em C, por exemplo, são representados por **float** e **double**.

- Definiu um padrão para a representação de números de ponto flutuante
- Surgiu em resposta às divergentes maneiras de representar números
  - ▶ Havia sérios problemas de portabilidade para códigos científicos
- Agora é adotado quase que universalmente
- Estabeleceu duas representações padrões
  - ▶ Precisão simples (32 bits)
  - ▶ Precisão dupla (64 bits)

	Sinal	Expoente	Mantissa
Precisão simples	1 bit	8 bits	23 bits
Precisão dupla	1 bit	11 bits	52 bits

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

- **Bit de sinal:** 0 positivo, 1 negativo
- O significando, ou **mantissa** é mantido normalizado
  - ▶  $1.0 \leq \text{mantissa} < 2.0$
  - ▶ Logo possuo, sempre, um bit 1 no início
  - ▶ Por isto este bit não é guardado explicitamente e acrescentando 1 bit à representação da mantissa
- **Expoente:** guardado como um expoente + deslocamento (bias)
  - ▶ Expoente é sempre guardado como um número sem sinal
    - Bias para precisão simples: 127
    - Bias para precisão dupla: 1023

- Os expoentes **00000000** e **11111111** são reservados
- Os "menores" valores representáveis são:
  - ▶ Expoente: **00000001**
    - Expoente real:  $1 - 127 = -126$
  - ▶ Mantissa: **0000...00**, que com o bit implícito 1 se torna 1.0
  - ▶ Valores:  $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Os "maiores" valores representáveis são:
  - ▶ Expoente: **11111110**
    - Expoente real:  $254 - 127 = +127$
  - ▶ Mantissa: **1111...11**, que com o bit implícito 1 é  $\approx 2.0$
  - ▶ Valores:  $\pm 2.0 \times 2^{+127} \approx 3.4 \times 10^{+38}$

- Os expoentes **000000000000** e **111111111111** são reservados
- Os "menores" valores representáveis são:
  - ▶ Expoente: **000000000001**
    - Expoente real:  $1 - 1023 = -1022$
  - ▶ Mantissa: **0000...00**, que com o bit implícito 1 se torna 1.0
  - ▶ Valores:  $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Os "maiores" valores representáveis são:
  - ▶ Expoente: **111111111110**
    - Expoente real:  $2046 - 1023 = +1023$
  - ▶ Mantissa: **1111...11**, que com o bit implícito 1 é  $\approx 2.0$
  - ▶ Valores:  $\pm 2.0 \times 2^{+1023} \approx 1.8 \times 10^{+308}$

- A precisão é relativa
  - ▶ Todos os bits da mantissa são levados em consideração
  - ▶ Precisão simples:  $\approx 2^{-23}$ 
    - Equivalente a  $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$  casas decimais
  - ▶ Precisão dupla:  $\approx 2^{-52}$ 
    - Equivalente a  $52 \times \log_{10} 2 = 52 \times 0.3 \approx 16$  casas decimais

- Como representar 0.75
  - ▶  $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - ▶ Sinal: 1
  - ▶ Expoente =  $-1 + \text{bias}$ 
    - Simples:  $-1 + 127 = 126 = 01111110_2$
    - Dupla:  $-1 + 1023 = 1022 = 011111111110_2$
  - ▶ Mantissa: (sem o 1 antes da vírgula):  $1000\dots00_2$
- Precisão simples: 1 01111110 1000...00
- Precisão dupla: 1 011111111110 1000...00

- Que número é representado pelo float de precisão simples: 1 10000001 01000...00

- ▶ Sinal: 1

- ▶ Expoente:  $10000001_2 - 127 = 129 - 127 = 2$

- ▶ Mantissa:

$$1 + .01_2 = 1 + (0 \times 2^{-1} + 1 \times 2^{-2}) = 1 + 1 \times \frac{1}{4} = 1 + 0.25 = 1.25$$

- ▶  $x = (-1)^1 \times 1.25 \times 2^2 = -1 \times 1.25 \times 2^2 = -5.0$

## ■ $\pm$ Infinito

- ▶ Expoente: 1111...11
- ▶ Mantissa: 0000...00
- ▶ Pode ser utilizado pelos cálculos seguintes sem a necessidade de verificações de overflow
  - Qualquer operação com **Inf** resulta em **Inf**, exceto com **NaN** que resulta em **NaN**.

## ■ NaN - *Not a Number*

- ▶ Expoente: 1111...11
- ▶ Mantissa:  $\neq$  0000...00
- ▶ Indica uma operação ilegal ou resultado não definido ( $\sqrt{-1}$  ou divisão por 0, por exemplo).
- ▶ Pode ser utilizado pelos cálculos seguintes sem a necessidade de verificações de overflow.
  - Qualquer operação com **NaN** resulta em **NaN**.

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

- Considere um exemplo, em decimal, de 4 dígitos

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

- 1 Alinhar os pontos decimais

- ▶ Desloca-se números com menor expoente

- ▶  $9.999 \times 10^1 + 0.016 \times 10^1$

- 2 Adicionar mantissa

- ▶  $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$

- 3 Normalizar resultado e verificar por overflow/underflow

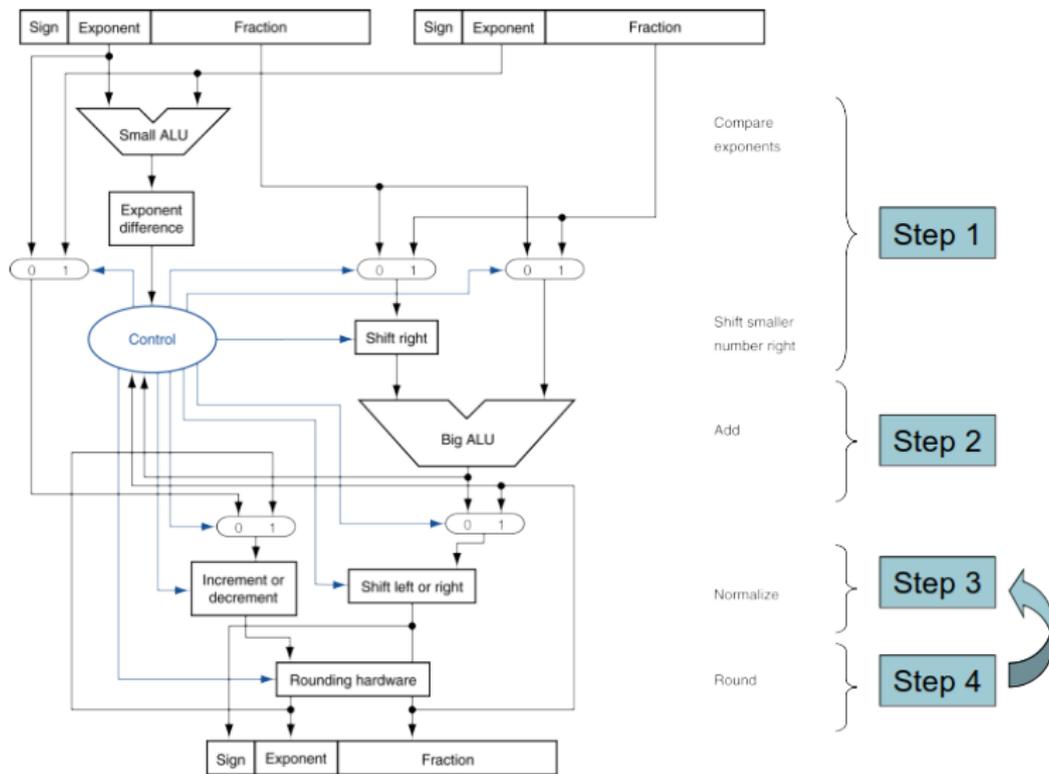
- ▶  $1.0015 \times 10^2$

- 4 Arredondar e renormalizar se necessário

- ▶  $1.002 \times 10^2$

- Agora considere um exemplo com um binário de 4 dígitos
  - ▶  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2} (0.5 + -0.4375)$
- ① Alinhar os pontos binários
  - ▶ Desloca-se números com menor expoente
  - ▶  $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- ② Adicionar mantissa
  - ▶  $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- ③ Normalizar resultado e verificar por overflow/underflow
  - ▶  $1.000_2 \times 2^{-4}$ , sem over/underflow
- ④ Arredondar e renormalizar se necessário
  - ▶  $1.000_2 \times 2^{-4}$  (sem mudanças) = 0.0625

- Muito mais complexo que um somador de inteiros.
- Para fazer a soma em um ciclo de clock faria com que o ciclo fosse muito longo
  - ▶ Muito mais longo do que operações sobre inteiros
  - ▶ Mas um clock mais lento penalizaria todas as instruções
- Consequência: o somador de FPs leva mais de um ciclo
  - ▶ Contudo pode ser colocado em pipeline!



- Considere um exemplo decimal com 4 dígitos

- ▶  $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$

- 1 Some expoentes

- ▶ Para expoentes com *bias* subtraia o *bias* da soma
- ▶ Novo expoente =  $10 + -5 = 5$

- 2 Multiplique as mantissas

- ▶  $1.110 \times 9.200 = 10.212 \rightarrow 10.212 \times 10^5$

- 3 Normaliza o resultado e verifica por over/underflow

- ▶  $1.0212 \times 10^6$

- 4 Arredonda e renormaliza se necessário

- ▶  $1.021 \times 10^6$

- 5 Determina o sinal do resultado pelo sinal dos operandos

- ▶  $+1.021 \times 10^6$

- Agora considere um exemplo binário de 4 dígitos
  - ▶  $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} (0.5 \times -0.4375)$
- ① Soma expoentes
  - ▶ Sem bias:  $-1 + -2 = -3$
  - ▶ Com bias:  $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- ② Multiplica mantissa
  - ▶  $1.000_2 \times 1.110_2 = 1.110_2 \rightarrow 1.110_2 \times 2^{-3}$
- ③ Normaliza resultado e verifica over/underflow
  - ▶  $1.110_2 \times 2^{-3}$  (sem mudanças) sem over/underflow
- ④ Arredonda e renormaliza (se necessário)
  - ▶  $1.110_2 \times 2^{-3}$  (sem mudanças)
- ⑤ Determina sinal:  $+ \times - \rightarrow -$ 
  - ▶  $-1.110_2 \times 2^{-3} = -0.21875$

- O hardware para multiplicação de FPs é semelhante ao de somas
  - ▶ No lugar de um somador para as mantissas usa um multiplicador
- Aritmética de FPs geralmente lida com
  - ▶ Soma, subtração, multiplicação, divisão, inverso e raiz quadrada
  - ▶ Conversão inteiro  $\Leftrightarrow$  FP
- Cada operação em geral leva vários ciclos
  - ▶ Contudo pode ser colocada em um pipeline

- O hardware para FP é o coprocessador 1
  - ▶ É um processador adjunto que complementa a ISA
- Por isto contém um conjunto extra de registradores
  - ▶ 32 registradores de precisão simples: `$f0`, `$f1`, ..., `$f31`
  - ▶ São usados pareados para precisão dupla: `$f0 / $f1`, `$f2 / $f3`, ..., `$f30 / $f31`
- Operações de FPs só funcionam em registradores de FPs
  - ▶ Programas em geral não fazem operações inteiras em dados de FPs ou vice-versa
  - ▶ Podemos então ter mais registradores com pouco impacto no tamanho do código
- Para carregar e armazenar FPs temos as instruções
  - ▶ `lwc1`, `ldc1`, `swc1`, `sdcl`

---

1 `ldc1 $f8, 32($sp)`

---

---

```
1 # precisão simples: add.s sub.s mul.s div.s
2 add.s $f0, $f1, $f6
3 # precisão dupla: add.d sub.d mul.d div.d
4 mul.d $f4, $f4, $f6
5
6 # Comparação
7 # P. Simples: c.eq.s c.lt.s c.le.s ...
8 # P. Dupla: c.eq.d c.lt.d c.le.d ...
9 # Ambos os casos setam o bit de condição para 0 ou 1
10 c.lt.s $f3, $f4
11
12 # E para usar o bit temos o branch condicional de FPs
13 # bc1t, bc1f - true ou false
14 bc1t TargetLabel
```

---

```
1 float f2c (float fahr) {  
2     return ((5.0 / 9.0) * (fahr - 32.0));  
3 }
```

- `fahr` em `$f12`, resultado em `$f0`, literais na memória global

```
1 f2c:  
2 lwc1 $f16, const5($gp) # Carrega 5.0 para $f16  
3 lwc2 $f18, const9($gp) # Carrega 9.0 para $f18  
4 div.s $f16, $f16, $f18 # Div. $f16 por $f18  
5 lwc1 $f18, const32($gp) # Carrega 32.0 para $f18  
6 sub.s $f18, $f12, $f18 # Subtrai $f18 de $f12  
7 mul.s $f0, $f16, $f18 # Multiplica $f16 $f18  
8 jr $ra # Retorna
```

- $X = X + Y \times Z$ 
  - ▶ Todos são matrizes de  $32 \times 32$  de FPs de precisão dupla
- Código em C

---

```
1 void mm (double x[][], double y[][], double z[][]) {
2     int i, j, k;
3     for (i = 0; i != 32; i = i + 1)
4         for (j = 0; j != 32; j = j + 1)
5             for (k = 0; k != 32; k = k + 1)
6                 x[i][j] = x[i][j] + y[i][k] * z[k][j];
7 }
```

---

- Endereços de  $x, y, z$  in  $\$a0, \$a1, \$a2$  e  $i, j, k$  em  $\$s0, \$s1, \$s2$

```
1      li    $t1, 32      # $t1 = 32 (tamanho linha/fim do laço)
2      li    $s0, 0      # i = 0; inicializa primeiro laço
3  L1:  li    $s1, 0      # j = 0; reinicia segundo laço
4  L2:  li    $s2, 0      # k = 0; reinicia terceiro laço
5      sll   $t2, $s0, 5  # $t2 = i * 32 (tam. da linha de x)
6      addu  $t2, $t2, $s1 # $t2 = i * tamanho(linha) + j
7      sll   $t2, $t2, 3  # $t2 = byte offset de [i][j]
8      addu  $t2, $a0, $t2 # $t2 = endereço de x[i][j]
9      l.d   $f4, 0($t2)  # $f4 = 8 bytes de x[i][j]
10 L3:  sll   $t0, $s2, 5  # $t0 = k * 32 (tam da linha de z)
11      addu  $t0, $t0, $s1 # $t0 = k * tamanho(linha) + j
12      sll   $t0, $t0, 3  # $t0 = byte offset de [k][j]
13      addu  $t0, $a2, $t0 # $t0 = endereço de z[k][j]
14      l.d   $f16, 0($t0) # $f16 = 8 bytes de z[k][j]
15      sll   $t0, $s0, 5  # $t0 = i*32 (tam. da linha de y)
16      addu  $t0, $t0, $s2 # $t0 = i*tamanho(linha) + k
17      sll   $t0, $t0, 3  # $t0 = byte offset de [i][k]
18      addu  $t0, $a1, $t0 # $t0 = endereço de y[i][k]
19      l.d   $f18, 0($t0) # $f18 = 8 bytes de y[i][k]
20      mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
21      add.d $f4, $f4, $f16 # f4=x[i][j] + y[i][k]*z[k][j]
22      addiu $s2, $s2, 1   # $k = k + 1
23      bne  $s2, $t1, L3  # if (k != 32) vai para L3
24      s.d   $f4, 0($t2)  # x[i][j] = $f4
25      addiu $s1, $s1, 1   # $j = j + 1
26      bne  $s1, $t1, L2  # if (j != 32) vai para L2
27      addiu $s0, $s0, 1   # $i = i + 1
28      bne  $s0, $t1, L1  # if (i != 32) vai para L1
```

- O padrão IEEE 754 também especifica alguns controles de arredondamento adicionais
  - ▶ Bits extras de precisão (*guard*, *round* e *sticky*)
  - ▶ Escolha do modo de arredondamento
  - ▶ Permite que o programador ajuste o comportamento desejado para sua aplicação
- Nem todas as unidades de ponto flutuante fornecem todos os modos
  - ▶ Boa parte das linguagens de programação e bibliotecas de FP usam os ajustes padrão
- Há novamente um trade-off entre a complexidade do hardware, o desempenho e requerimentos de marketing

- Originalmente baseada no coprocessador 8087
  - ▶ Registradores de precisão estendida de 80 bits
  - ▶ Utilizado como uma pilha
  - ▶ Registradores indexados a partir do TOS: ST(0), ST(1), ...
- Os valores em FP são armazenados em 64 bits na memória
  - ▶ Convertidos durante o load/store do operando
  - ▶ Operando inteiros também são convertidos neste momento
- É muito complicado gerar e otimizar o código para esta arquitetura
  - ▶ Resultado: baixo desempenho.

Transf. Dados	Aritmética	Comparação	Transcendental
F   LD mem/ST(i)	F   ADD P mem/ST(i)	F   COM P	FPATAN
F   ST P mem/ST(i)	F   SUB RP mem/ST(i)	F   UCOM P	F2XMI
FLDPI	F   MUL P mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F   DIV RP mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FPSIN
	FRNDINT		FYL2X

## ■ Variações

- ▶ Operando inteiro: I
- ▶ Pop do operando da pilha: P
- ▶ Ordem inversa dos operandos: R
- ▶ Mas nem todas as combinações são possíveis.

- Adicionou 4 registradores de 128 bits
  - ▶ Mais 4, totalizando 8, foram adicionados no AMD64/EM64T
- Podem ser usados para armazenar múltiplos operandos FP
  - ▶ 2 operandos de 64 bits (precisão dupla)
  - ▶ 4 operandos de 32 bits (precisão simples)
- As instruções operam nos valores simultaneamente
  - ▶ SIMD - Single Instruction Multiple Data

- Fazer o deslocamento para esquerda (*left shift*) em  $i$  posições multiplica um inteiro por  $2^i$
- Deslocar para a direita (*right shift*) em  $i$  posições divide por  $2^i$ ?
- Para números sem sinal, sem problema
- Para números com sinal
  - ▶ É preciso estender o bit de sinal
  - ▶ Exemplo:  $-5/4$ 
    - $11111011 \gg 2 = 11111110 = -2$
    - Arredonda para  $-\infty$
    - Confronte com:  $11111011 \gg 2 = 00111110 = +62$

- Programas paralelos podem intercalar operações em ordem inesperadas
- Suposições sobre associatividade podem falhar

		$(x + y) + z$	$x + (y + z)$
x	$-1.50E + 38$		$-1.50E + 38$
y	$+1.50E + 38$	$0.00E + 00$	
z	1.0	1.0	$+1.50E + 38$
		$1.00E + 00$	$0.00E + 00$

- É preciso fazer uma rodada a mais de testes em programas paralelos para garantir os resultados com mais níveis de paralelismo

- É importantíssimo para código científico
- E para consumidores do dia a dia?
  - ▶ Imagine o seu saldo bancário com uma diferença de 0.0002 centavos...
- O Bug do Pentium FDIV
  - ▶ O mercado espera precisão
  - ▶ Veja "The Pentium Chronicles", por Colwell

- Bits não tem nenhum significado inerente
  - ▶ A interpretação dos bits depende das instruções que forem aplicadas a eles
- Representação pelo computador de números
  - ▶ Tem limites finitos e de precisão
  - ▶ Programas precisam levar isto em conta

- As ISAs dão suporte a operações aritméticas
  - ▶ Inteiros com e sem sinal
  - ▶ Aproximação de números reais por FPs
- Limites de valores e precisão
  - ▶ Operações podem dar overflow ou underflow
- A ISA do MIPS
  - ▶ Instruções principais: 54
    - Representam 100% das instruções usadas pelo SPECINT
    - Representam 97% das instruções usadas pelo SPECFP
  - ▶ Outras instruções são bem menos frequentes.