**Aritmética computacional:**

* Registradores: Armazenam 32 bits.
* Operações sobre inteiros de 32 bits.
* Números com sinal:
* Complemento a dois
* X é um número inteiro positivo

-x = , onde n é tamanho, em bits, da representação.

Ex: Num sistema de 4 bits:

obs: utilizaremos a base 2

0101 = 5

1010

+1

1011 = -5

= 11 = -5

* Adição e subtração de binários:

+

* Overflow:

ex: num sistema de 4 bits:

* Lidando com overflow na adição:

| Operação | Sinal A | Sinal B | Sinal A + B |
| --- | --- | --- | --- |
| A+B | >= | >= | < |
| A+B | < | < | >= |
| A - B | >= | < | < |
| A -B | < | >= | >= |

**Estes são os casos em que ocorre o overflow**

**Obs:** Nunca ocorre overflow na soma de inteiros de sinal diferentes (equivalente na subtração de inteiros com o mesmo sinal)

* As instruções add,addi e sub lançam exceção no caso de overflow, enquanto que addu,addiu e subu **não** lançam exceção.

**Obs:** Exceção é a interrupção de um programa.

* Para fazer o tratamento (sem interrupção) de um overflow em assembly mips:

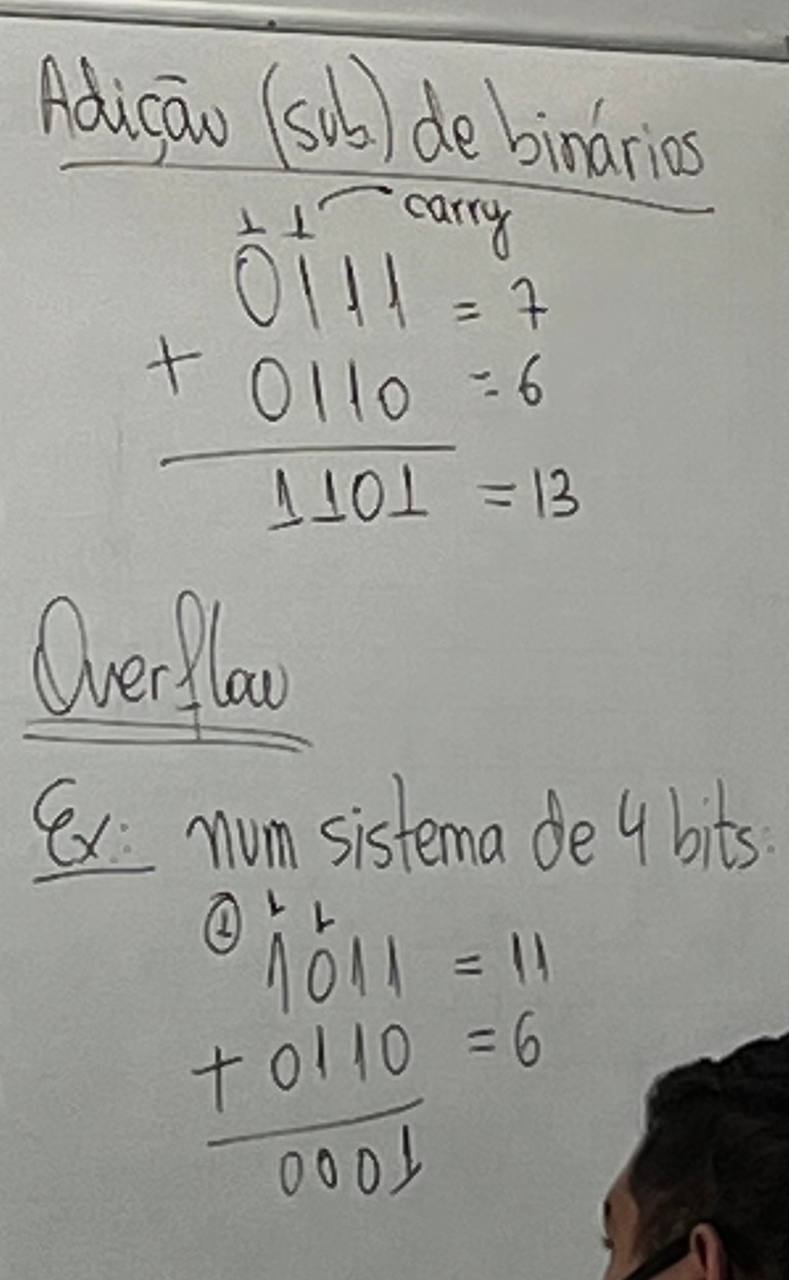
1. Calcule a soma usando addu
2. Se os sinais dos operandos forem iguais: Se o sinal do resultado for igual ao sinal do primeiro operando, não há overflow, senão, há overflow.

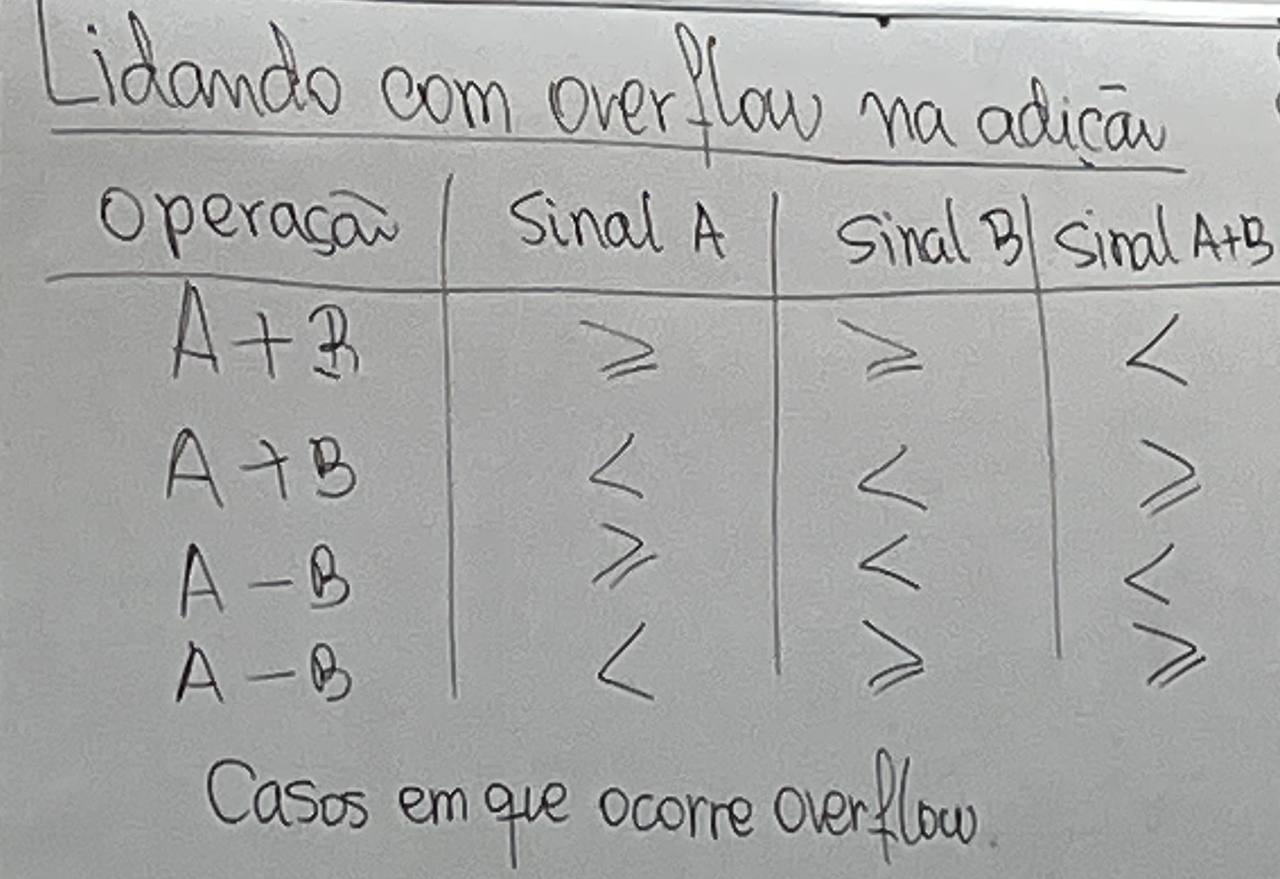
ex: Tratar overflow na soma de $t0 e $t1.

| addu $t2,$t0,$t1  xor $t3,$t0,$t1 #$t3 < 0 se os sinais forem distintos.  slt $t3,$t3,$zero  bne $t3,$zero, sem\_overflow #se os sinais são diferentes não  há overflow  xor $t3,$t0,$t2  slt $t3,$t3,$zero  bne $t3,$zero,overflow |
| --- |

**obs:** Para verificar se o sinal de dois operandos são diferentes:

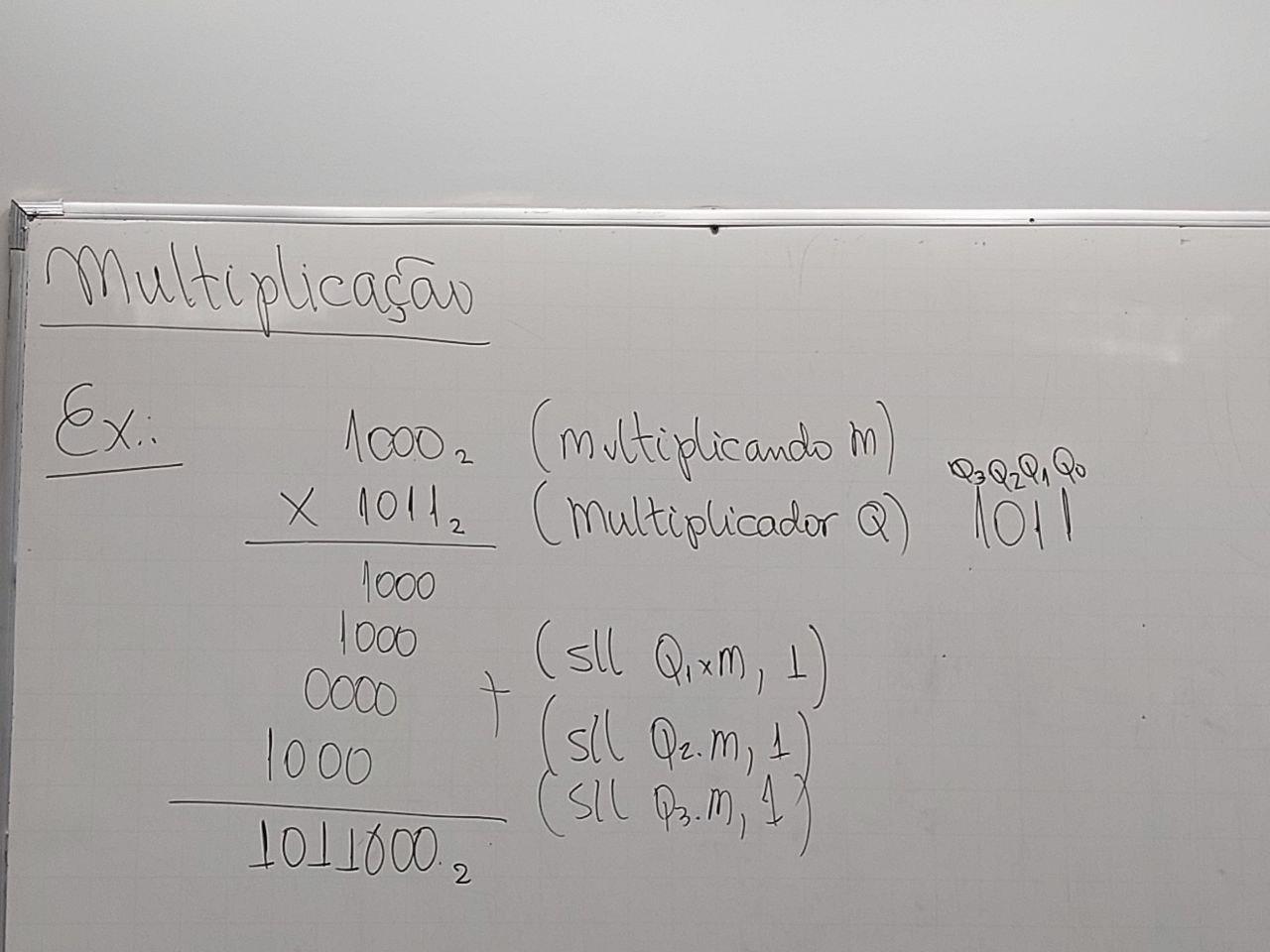
$t0 =1… ou $t1=0…: xor = 1





# ✖ Multiplicação ✖

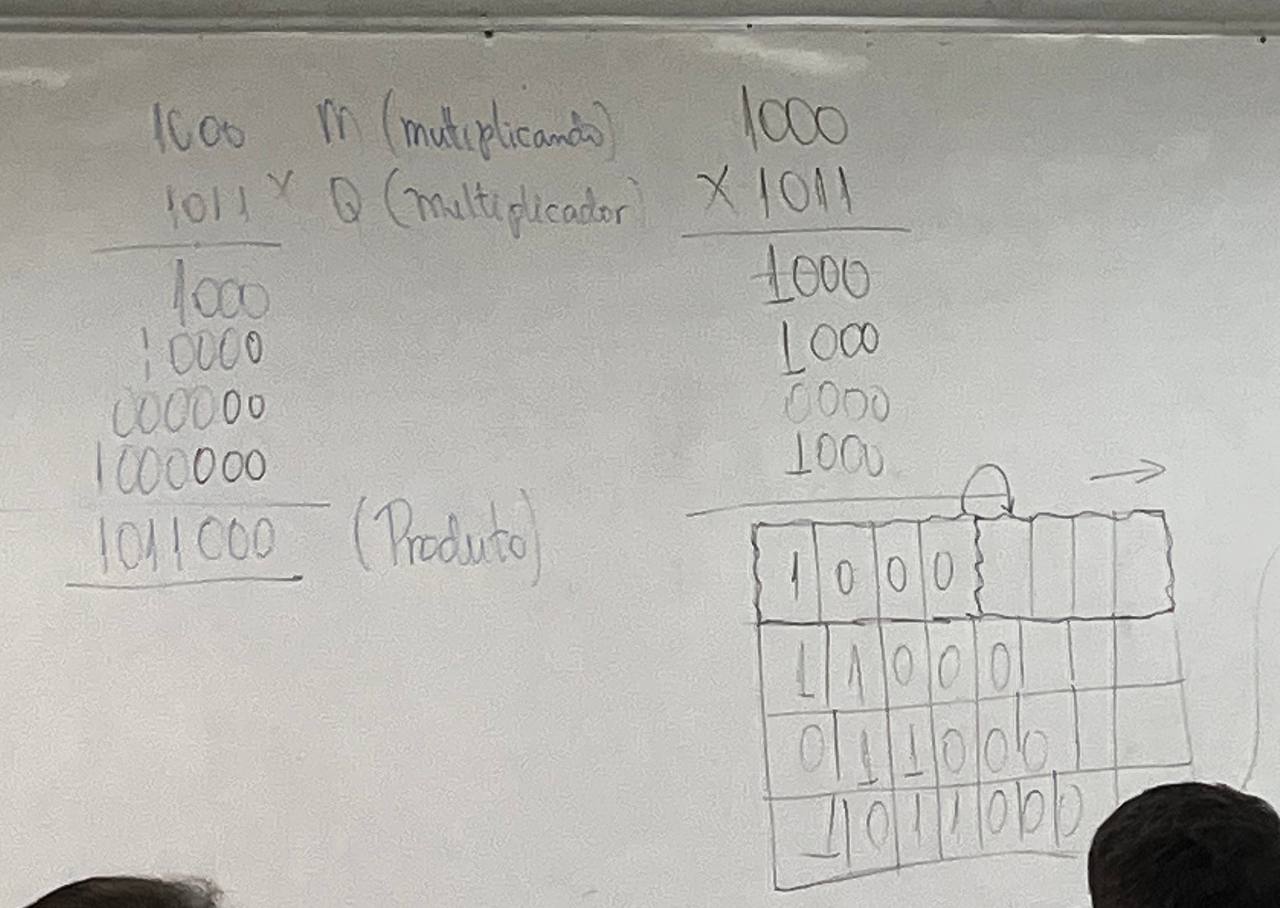
Ex:



* Produto: Registrador de 64 bits
  + O produto tem que ter 64 bits pois ele terá 2x -1 a quantidade de bits de Q, como Q é um reg de 32 bits, o Produto da multiplicação deve ter 64 bits!
* Multiplicando: reg. de 64 bits
  + O multiplicando precisa armazenar os valores resultantes de cada multiplicação
* Multiplicador: reg de 32 bits

**Algoritmo**

1. Inicialize P = 0 e contador = 1
2. P = → Condicional
3. Desloque o à esquerda.
4. Desloque Q à direita
5. Se contador = 32, pare. Senão, contador++ e volte para o passo 2.



**Algoritmo de multiplicação otimizado (Dados M e Q. saída: P[63…0])**

**Obs: GOSTA DE COBRAR NA PROVA ESSA SIMULAÇÃO POR UMA TABELA**

X

1. Inicialize P[63 ... 32] = 0 e P[31 ... 0] = M.
2. Se P[0] = 1 , P[63 ... 32]+=M.
3. Desloque P à direita.
4. Se não for a repetição, volte ao passo dois.

Ex: M = 0110, Q = 1101, P = 01001110

| **Iterações** | **Descrição** | **P** |
| --- | --- | --- |
| 0 | Inicialização | 0000 1101 |
| 1 | P[7…4] +=M  Srl P | 0110 1101  0011 011**0** |
| 2 | mantém P  Srl P | 0011 0110  0001 101**1** |
| 3 | P[7…4] +=M  Srl P | 0111 1011  0011 110**1**  **+0110** |
| 4 | P[7…4] += M  Srl P | 1001 1101  0100 1110 |

**Obs:** Em vermelho é o produto.

**Obs:** O algoritmo funciona apenas para operandos sem sinal. Para o caso com sinal.

1. Armazene os sinais dos operandos e transforme-os em positivos.
2. Rode o algoritmo,
3. Se os sinais dos operandos eram diferentes, transforme o produto em negativo.

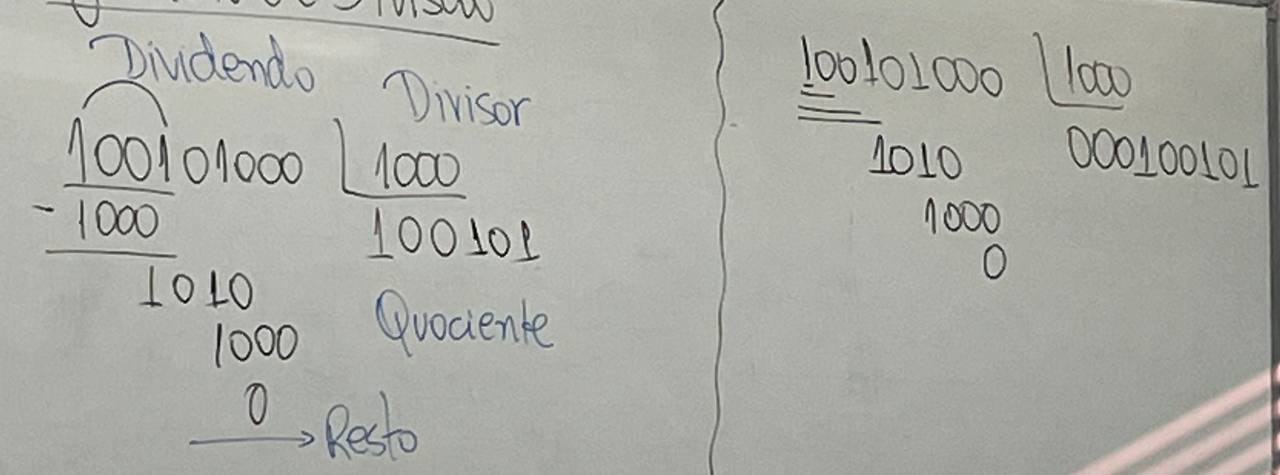
**obs:**Algoritmo de Booth lida diretamente com negativos.

**Instruções MIPS:**

mult reg1,reg2

* Calcula reg1\*reg2
* Salva o produto num par de registradores especiais chamados: **Hi**(mais significativo) e **Lo** (menos significativo).
* **mflo reg e mfhi reg**: Copiam o conteúdo de lo e hi para reg .
* **mtlo reg e mthi reg**: Escreve o conteúdo de reg em lo e hi.
* 12/08 → Somativa 2
* 19/09 → Somativa 3
* Vai contar só uma das duas listas que ele soltar

Algoritmo de divisão



| Divisor - 64 bits |
| --- |
| Resto - 64 bits |
| Quociente - 32 bits |

Passo 1: R = k, contador = 1.

Passo 2: R = R - D.

Passo 3: Desloque Q à esquerda.

* Se R >= 0, Q[0] = 1.
* Se R < 0, restaure o valor original de R. → (R=R+D)

Passo 4: Desloque D à direita.

Passo 5: Se o contador < 33, contador ++ e volte ao passo 2.

ex: 0111 / 0010

Divisor: 0010 0000 (8 bits)

Resto: 0000 0111 (8 bits)

### Exemplo para uma arquitetura de 4 bits:

| Interação | Descrição | Quociente | Divisor | Resto |
| --- | --- | --- | --- | --- |
| 0 | Inicialização | 0000 | 0010 0000 | 0000 0111 |
| 1 | R = R - D  srl D | 0000  0000 | 0010 0000  0001 0000 | <0  0000 0111 |
| 2 | R = R - D  srl D | 0000  0000 | 0001 0000  0000 1000 | <0  0000 0111 |
| 3 | R = R - D  srl D | 0000  0000 | 0000 1000  0000 0100 | <0  0000 0111 |
| 4 | R = R - D  Q[0] =1, srl D | 0000  0001 | 0000 0100  0000 0010 | 0000 0011  0000 0011 |
| 5 | sll Q, R -= D  Q[0] = 1, srl D | 0010  **0011** | 0000 0010  0000 0001 | 0000 0001  **0000 0001** |

**Instruções no Mips:**

* div (com sinal)
* divu (sem sinal)

div reg1,reg2

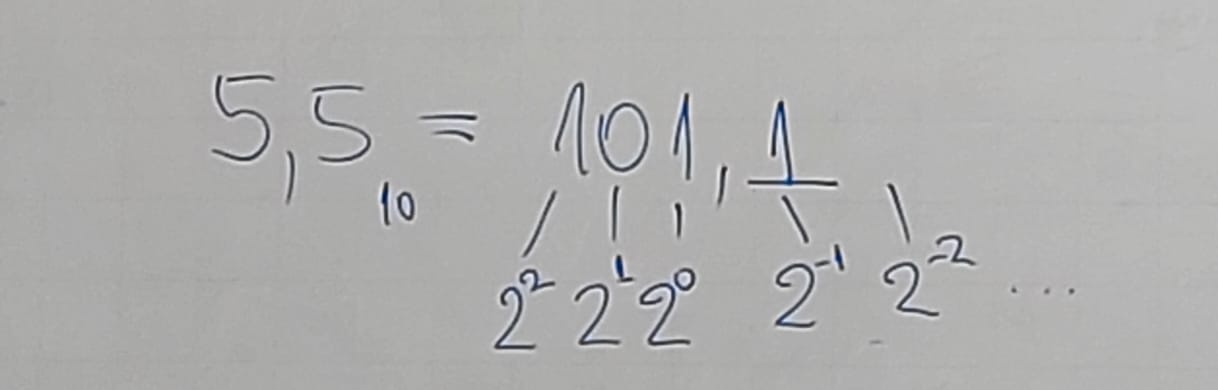
| resto | Quociente |
| --- | --- |
| hi | lo |

* mfhi reg → reg = hi
* mflo reg → reg = lo
* Obs: O resto sempre possui o mesmo sinal do dividendo.

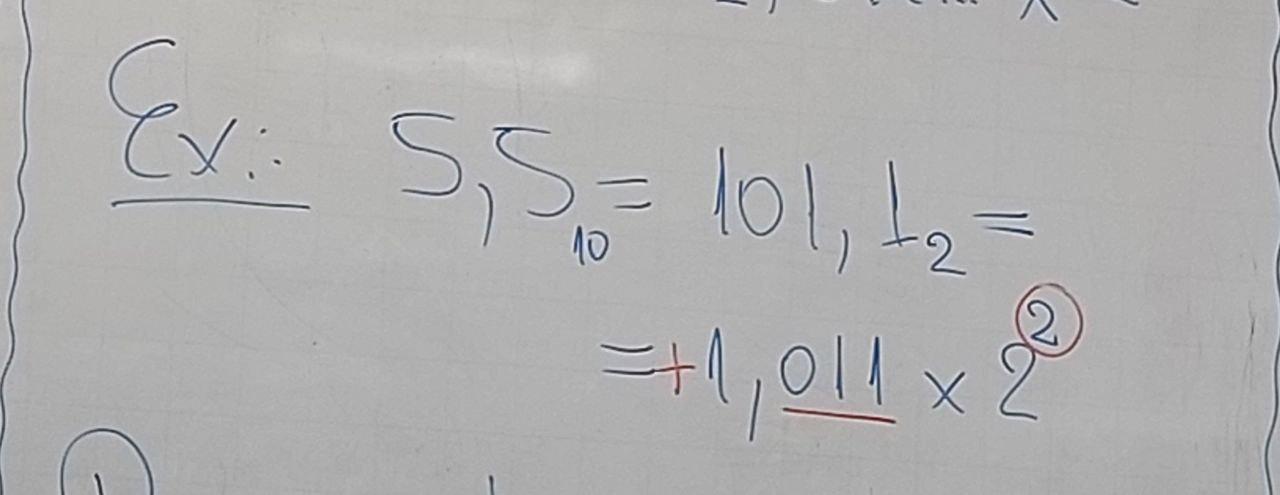
ex: 7/-3 = -2, resto = 1 ou -7/3 = -2, resto = -1

**Ponto flutuante:**

* Em binário:



* Notação científica : +- f
* Na versão normalizada: 1 <= f <= 9 (fração mantissa significando)
* Em binário: 1,zzzz… x



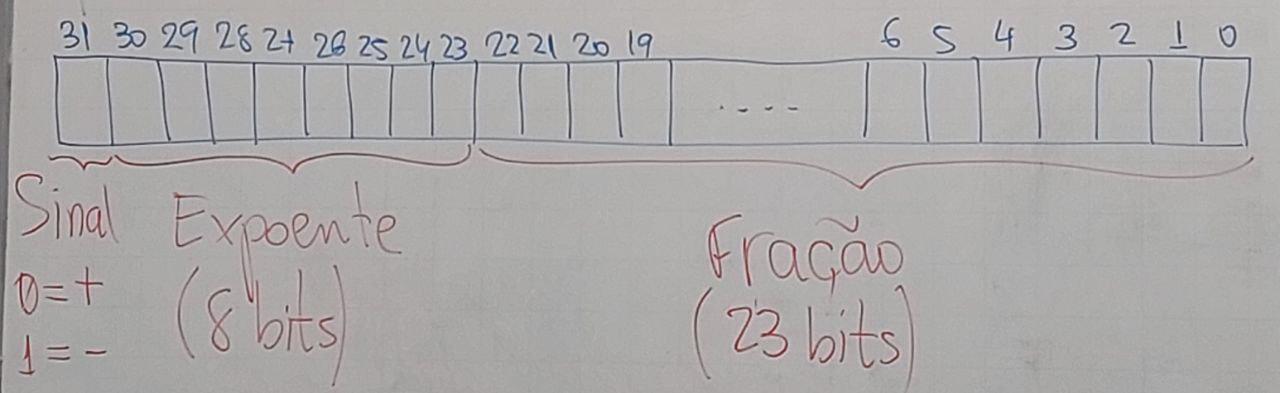
* Representar um número de ponto flutuante consiste em representar
* **Fração**
* **Expoente**
* **Sinal**
* Padrão IEEE 754
* Um número de ponto flutuante é representado por uma palavra (word = 32 bits) da seguinte forma:(float)

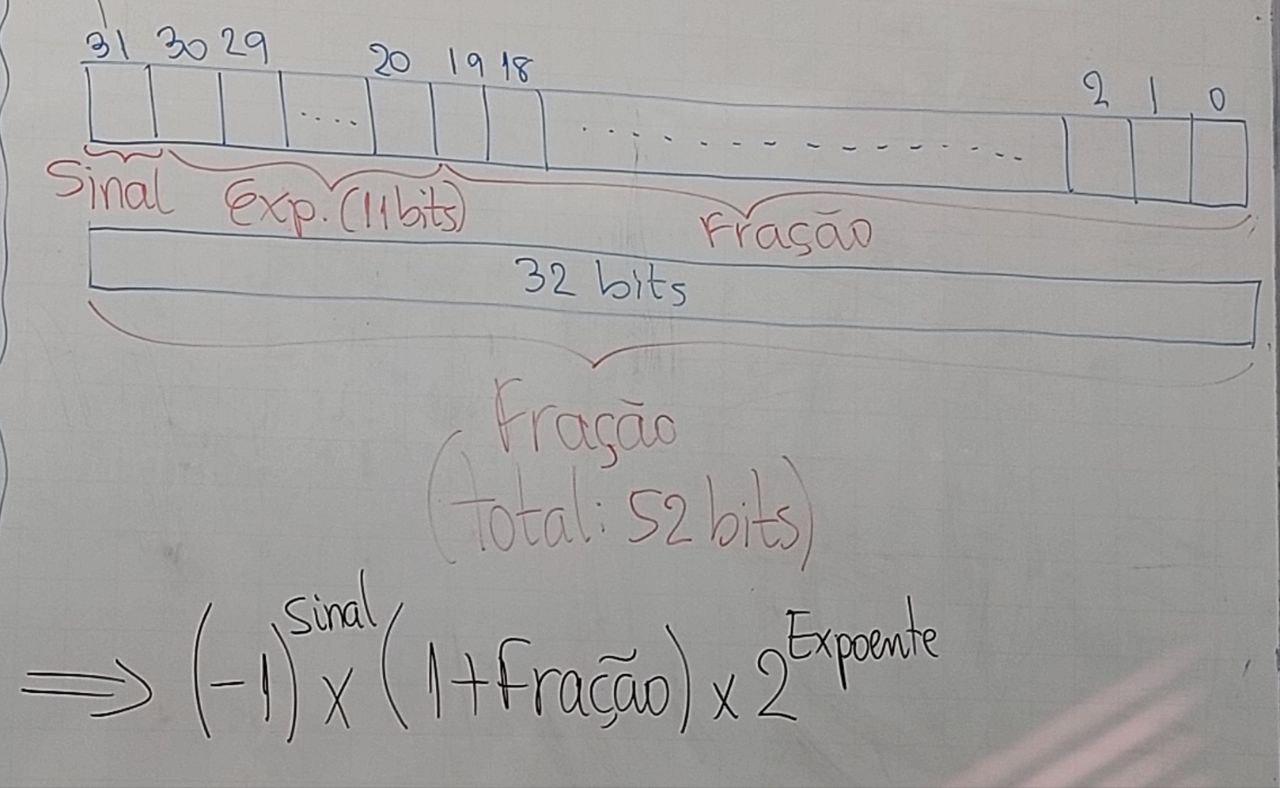
31 30 29 28 27 26 25 24 23 22 21 20 19 … 6 5 4 3 2 1 0

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

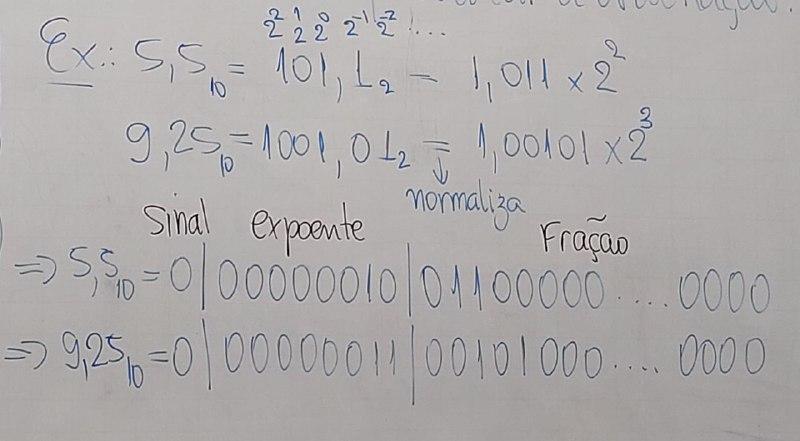
— —------------------------ —----------------------------------------------------  
⤷Sinal ⤷Expoente ⤷Fração

0 = + (8 bits) (23 bits)

1 = - 

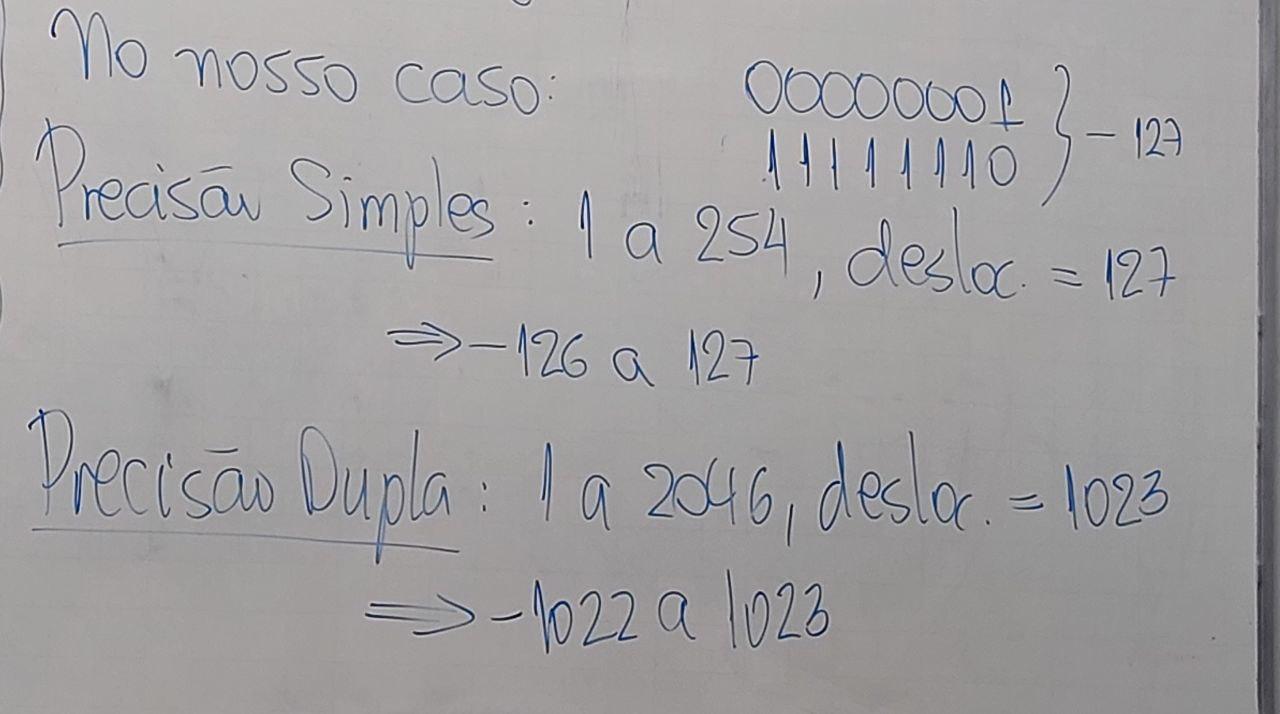
* Chamamos essa representação de ponto flutuante de **precisão simples.**
* Há também a representação em **precisão dupla**
  + É chamado de “dupla”, pois utiliza **dois registradores** para representar o ponto flutuante (não confundir com o dobro do valor, já que será bem mais que o dobro)
* O padrão IEEE 754 estabelece os seguintes casos particulares:

| precisão simples | | precisão dupla | | **representação** |
| --- | --- | --- | --- | --- |
| **expoente** | **fração** | **expoente** | **fração** |
| 0 | 0 | 0 | 0 | zero |
| 0 | qualquer coisa diferente de 0 | 0 | qualquer coisa diferente de 0 | Número desnormalizado |
| 1-254 | qualquer coisa | 1-2046 | qualquer coisa | ponto flutuante normalizado |
| 255 | 0 | 2047 | 0 | +-infinito |
| 255 | diferente de zero | 2047 | diferente de zero | NaN ( Not a Number) |

* E por que os elementos de um ponto flutuante são representados nessa ordem em binário ? (**boa questão de prova**)
* **Para facilitar a ordenação.**
* Por esse motivo que não podemos utilizar complemento de dois para representar números negativos pois afetaria na ordenação.

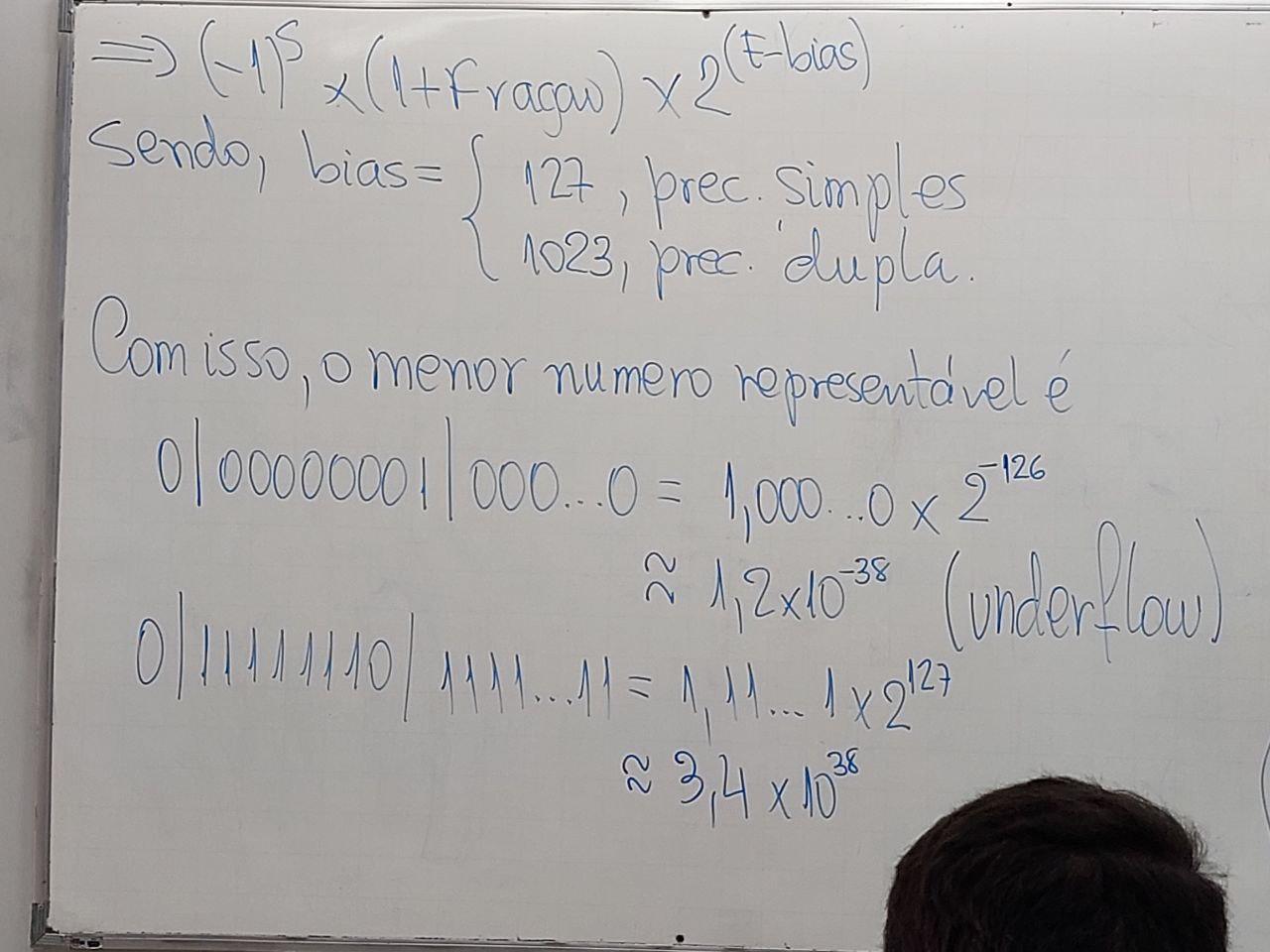
Ex: 0.5 =

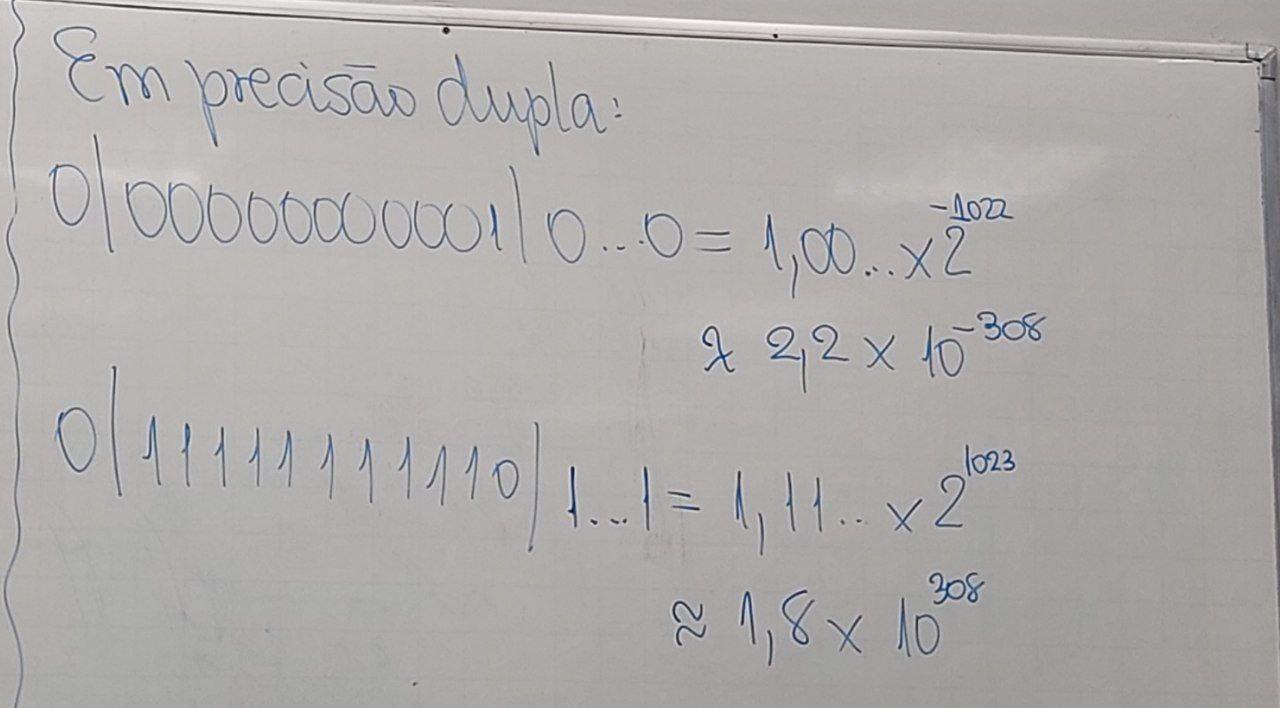
→ = 0 | 11111111 | 0 0 0 0 … 0 0

* logo, se usarmos complemento a 2 para representar expoentes negativos, perderíamos a propriedade de que binários ordenados implicam pontos flutuantes ordenados. Por isso, o expoente é representado por **excesso.**|
* **Representação por excesso:** define-se um deslocamento (bias) que somado ao número mais negativo resulte 0.

→ ( \* , sendo, bias 127, prec. simples

1023, prec. dupla

Com isso, o menor número representável é: 1,2 \*



Extras para prova:

* **Algoritmo de multiplicação:**

multfac:

#pre-passos:

addi $t4, $zero, 32

slt $t8, $a0, $zero

slt $t9, $a1, $zero

beq $t8, $zero, verificando\_a1

nor $a0, $a0, $zero

addi $a0, $a0, 1

verificando\_a1:

beq $t9, $zero, passo1

nor $a1, $a1, $zero

addi $a1, $a1, 1

#passo 1:

passo1:

move $t0, $zero

#passo 2:

move $t1, $a1

#passo 3:

passo3:

andi $t2, $t1, 1

beq $t2, $zero, passo4

add $t0, $t0, $a0

#passo 4:

passo4:

andi $t3, $t0, 1

srl $t0, $t0, 1

srl $t1, $t1, 1

sll $t3, $t3, 31

add $t1, $t1, $t3

#passo 5:

addi $t4, $t4, -1

beq $t4, $zero, sinal

j passo3

#retorno:

sinal:

beq $t8, $t9, fim

nor $t1, $t1, $zero

nor $t0, $t0, $zero

addi $t1, $t1, 1

fim:

mthi $t0

mtlo $t1

jr $ra

* **Algoritmo de divisão**

.data

msg1: .asciiz "Digite o dividendo (M): "

msg2: .asciiz "Digite o divisor (Q): "

res1: .asciiz "hi = "

res2: .asciiz "lo = "

nl: .asciiz "\n"

.text

main:

# $a0 : dividendo (M)

# $a1 : divisor (Q)

li $v0, 4

la $a0, msg1

syscall

li $v0, 5

syscall

move $a0, $v0

li $v0, 4

la $a0, msg2

syscall

li $v0, 5

syscall

move $a1, $v0

jal divfac

li $v0, 4

la $a0, res1

syscall

li $v0, 1

mfhi $a0

syscall

li $v0, 4

la $a0, nl

syscall

li $v0, 4

la $a0, res2

syscall

li $v0, 1

mflo $a0

syscall

li $v0, 4

la $a0, nl

syscall

li $v0, 10

syscall

divfac:

#Passo 1

# $t0 : Divisor (D)

add $t0, $a1 ,$zero

# $t1 : Quociente (R[63..32])

move $t1, $a0

# $t2 : Resto (R[32.. 0])

add $t2, $a1, $zero

# $t9 : contador

addi $t4, $zero, 32

#Passo 2

slt $t3, $t2, $zero

sll $t1, $t1, 1

sll $t2, $t2, 1

beq $t3, $zero, passo3

addi $t1, $t1, 1

#Passo 3

passo3:

sub $t1, $t1, $t0

#Passo 4

slt $t3, $t1, $zero

beq $t3, $zero, ehmaior

slt $t3, $t2, $zero

add $t1, $t1, $t0

sll $t2, $t2, 1

beq $t3, $zero, passo5

addi $t1, $t1, 1

j passo5

ehmaior:

sll $t2, $t2, 1

ori $t2, $t2, 1

beq $t3, $zero, passo5

addi $t1, $t1, 1

#Passo 5

passo5:

addi $t4, $t4, -1

beq $t4, $zero, passo6

j passo3

#Passo 6

passo6:

srl $t1, $t1, 1

#Retorno

mthi $t1

mtlo $t2

jr $ra

* **Algoritmo overflow:**

.data #atribuicao de valor para as variaveis

overflow: .asciiz "overflow" #atribuicao de valor para a string overflow

pulaLinha: .asciiz "\n" #atribuicao de valor para a string pulaLinha

.text #instrucoes do programa

main: #inicio para a execucao do programa

li $v0, 5 #chamada de sistema para que leia um inteiro

syscall #chamada

move $t4, $v0 #$t4 recebe o valor lido do usuário

li $v0, 5 #chamada de sistema para que leia um inteiro

syscall #chamada

move $t0, $v0 #$t0 recebe o valor lido do usuário

li $v0, 5 #chamada de sistema para que leia um inteiro

syscall #chamada

move $t1, $v0 #$t1 recebe o valor lido do usuário

addu $t2, $t0, $t1 #faz a soma sem sinal de $t0 com $t1 e armazena em $t2

bne $t4, $zero, comSinal #verifica a opção escolhida pelo usuário

nor $t3, $t0, $zero #nega o primeiro termo

sltu $t3, $t3, $t1 #se menor que o resultado da soma sem sinal, quer dizer que houve oveflow

bne $t3, $zero, comOverflow #se for overflow, pule para o rótulo comOverflow

j semOverflow #pule para o rótulo semOverflow

comSinal: #rótulo comSinal

xor $t3, $t0, $t1 #faz o xor entre os dois números

slt $t3, $t3, $zero #a partir do xor verifica se são de sinais diferentes

bne $t3, $zero, semOverflow #se são diferentes, pula para semOverflow já que contas entre números de sinais diferentes não geram overflow

xor $t3, $t2, $t0 #faz o xor entre o resultado e um dos números

slt $t3, $t3, $zero #a partir do xor verifica se são de sinais diferentes

bne $t3, $zero, comOverflow #se forem diferentes, houve overflow, e então pula para comOverflow

j semOverflow #pule para semOverflow

comOverflow: #rótulo comOverflow

la $a0, overflow #passa para $a0 o endereço de overflow

li $v0, 4 #chamada para imprimir uma string

syscall #chamada

j fim #pule para fim

semOverflow: #rótulo semOverflow

move $a0, $t2 #move para $a0 o conteúdo de $t2

li $v0, 1 #chamada para imprimir um inteiro

syscall #chamada

fim: #rótulo fim

la $a0, pulaLinha #passa o endereço de pulaLinha para o registrador $a0

li $v0, 4 #chamada pare imprimir uma string

syscall #chamada

li $v0, 10 #chamada de saida do programa

syscall #chamada

# Arredondamento

Há 4 tipos:

* Sempre pra cima : 2,11 -> 2,2

2,15 -> 2,2

* Sempre pra baixo : 2,11 -> 2,1

2,15 -> 2,1

* Truncamento, despreza bits menos significativos: ->

->

* Ao mais próximo (mais utilizado):

2,11 -> 2,1

2,19 -> 2,2

terminando em 5 vai ao par mais próximo

# Ponto Flutuante no MIPS

* Há 32 registradores separados em outro processador para lidar com números de ponto flutuante
* Os registradores são $f0, $f1, $f2,..., $f30 e $f31
* Não há registrador reservado para uso especial (os 32 registradores podem fazer qualquer coisa)
* Para **precisão dupla**: registradores de número par.
  + Exemplo: Usa-se $f2, instrução ocupa $f2 e $f3 (convenção Assembly)

## Novos tipos de dados:

* **float**: f1, f2, …, fn
* **double**: d1, d2, …, dn

(para usar em .data)

## Syscalls de Ponto Flutuante

| **Serviço** | **Código** | **Argumento** | **Resposta** |
| --- | --- | --- | --- |
| imprimir float | 2 | $f12 | - |
| imprimir double | 3 | $f12 | - |
| ler float | 6 | - | $f0 |
| ler double | 7 | - | $f0 |

## Instruções

### Padrão

op.prec

op -> mnemônico da operação

precisão -> s (single) ou d (double)

### Aritméticas

* 1. add.s $f2, $f4, $f6 (f2 = f4 + f6)
  2. add.d
  3. sub.s
  4. sub.d
  5. mul.s
  6. mul.d
  7. div.s
  8. div.d

### Acesso a memória

* 1. l.s $f0, 0 ($t2) *carrega*
  2. l.d
  3. s.s $f0, 100 ($s0) *escreve*
  4. s.d
  5. move.s $s0, 0
  6. move.d

OBS:

precisão simples: 4 bytes

**precisão dupla: 8 bytes**

**(Boa questão de prova)**

Exemplo: Suponha um vetor de ponto flutuante de **precisão dupla** com **endereço base em** $s0. Como carregar o quinto elemento em $f0 ?

| l.d $f0,40($s0) # 40 = 5\*8 (8 bytes para precisão dupla) |
| --- |

Obs: se fosse precisão simples usamos l.s

### 3. Desvios Condicionais

É feito em duas etapas:

1. Comparação entre registradores, salva o resultado num registrador especial
2. Efetua o desvio baseado no valor do registrador especial

#### (a) Comparação

* + c.eq.s $f2, $f4 # =
  + c.ne.s !=
  + c.le.s <=
  + c.lt.s <
  + c.ge.s >=
  + c.gt.s >

#### (b) Desvio

* + bc1f label
    - O primeiro desvia para label se o reg. especial é falso (== 0)
  + bc1t label
    - O segundo desvia se é verdadeiro (== 1)

### Exemplo: Fahrenheit - Celsius

Faça um programa para converter a escala de temperatura fahrenheit para celsius.

| float f2c(float F) {  return(5.0/9.0)\*(F-320); } |
| --- |



| .data  c5: .float 5.0  c9: .float 9.0  c32: .float 32.0  .text  li $v0,6 #lê um float retorno em $f0  syscall   l.s $f1, c5 #forma alternativa de usar o load word, usando  diretamente a constante  l.s $f2, c9  l.s $f3, c32  sub.s $f0, $f0, $f3 #F - 32  div.s $f4, $f1, $f2 # f4 = 5/9  mul.s $f12, $f4, $f0 # f12 -> °C  li $v0,2 # imprime $f12 na tela  syscall |
| --- |

Na prova ele pode pedir Média ou algo parecido/que tenha

* Média ponderada:

.data #atribuicao de valor para as variaveis

.text #instrucoes do programa

main: #inicio para a execucao do programa

li $v0, 5 #chamada de sistema para que leia um inteiro

syscall #chamada

li $t0, 0

move $t2, $v0

mtc1 $zero, $f10

mtc1 $zero, $f14

FOR:

slt $t1, $t0, $t2

beq $t1, $zero, Fim

li $v0, 7

syscall

add.d $f14, $f14, $f0

mov.d $f2, $f0

li $v0, 7

syscall

mul.d $f6, $f0, $f2

add.d $f10, $f10, $f6

addi $t0, $t0, 1

j FOR

Fim:

div.d $f10, $f10, $f14

mov.d $f12, $f10

li $v0, 3

syscall

li $v0, 10 #chamada de saida do programa

syscall #chamada

* Velocidade média

.data #atribuicao de valor para as variaveis

.text #instrucoes do programa

main: #inicio para a execucao do programa

li $v0, 7 #chamada de sistema para que leia um double

syscall #chamada

mov.d $f2, $f0 #move para $f1 o que está em $f0

li $v0, 7 #chamada de sistema para que leia um double

syscall #chamada

div.d $f12, $f2, $f0 #armazena em $f12 o resultado da divisão entre $f1 e $f0

li $v0, 3 #chamada de sistema para que imprima um float

syscall #chamada

li $v0, 10 #chamada de saida do programa

syscall #chamada