“Arquitetura e Organização de Computadores I – Aula_04 – Aritmética Computacional”

Prof. Dr. Emerson Carlos Pedrino

DC/UFSCar

São Carlos

Números com Sinal e Sem Sinal

- Sistema Sinal-Magnitude
  - Zero positivo e negativo.
  - *Hardware* mais complexo.
- Sistema de Complemento de 2
  - *Hardware* mais simples.
- *Exercício: mostrar como seriam implementadas as operações aritméticas de soma e subtração no sistema S-M.

Comparação entre números com sinal e sem sinal

- No MIPS:
  - slt: *set on less than* e slti: *set on less than immediate* -> para inteiros com sinal.
  - sltu: *set on less than unsigned* e sltiu: *set on less than immediate* -> para inteiros sem sinal.

Exercício*

- Dados:
- $s0=1111 1111 1111 1111 1111 1111 1111 1111
- $s1= 0000 0000 0000 0000 0000 0000 0000 0001
- $t0? e $t1? para:
  - slt $t0,$s0,$s1
  - sltu $t1,$s0,$s1
    - Sol: $s0=-1 ou 4294967295 e $s1=1 nos dois casos. Assim, $t0=1, pois -1<1 e $t1=0 pois 4294967295>1.

Extensão de Sinal

- Bit de sinal replicado à esquerda.
- No MIPS: *load byte* (lb), *load half* (lh) e *load byte unsigned* (lbu) e *load half unsigned* (lhu).
- *Loads* sem sinal simplesmente preenchem com 0s à esquerda dos dados.

Exercício* - Atalho para verificação de limites (reduz o custo para verificar 0<=x<y)

- Dado: (dica: usar stlu e beq)
  - Se $a1>=$t2 ou $a1<0 vá para IndiceForaDosLimites
    - Sol.:
    - sltu $t0,$a1,$t2 # $t0<-0 se k>=tamanho ou k<0
    - beq $t0,$zero,IndiceForaDosLimites # se fora dos limites, vá para Erro.
      - Obs.: números em C_2 se parecem com números grandes sem sinal.
      - Assim, uma comparação sem sinal de x<y também verifica se x é negativo.

Resumo até aqui

MIPS operands

Name	Example	Comments
32 registers	$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
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 assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load half unsigned	lhu $s1,100($s2)	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	sh $s1,100($s2)	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte unsigned	lbu $s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; two's complement
	set less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; unsigned numbers
	set less than immediate unsigned	sltiu $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; unsigned numbers
Uncondi-tional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Adição e Subtração

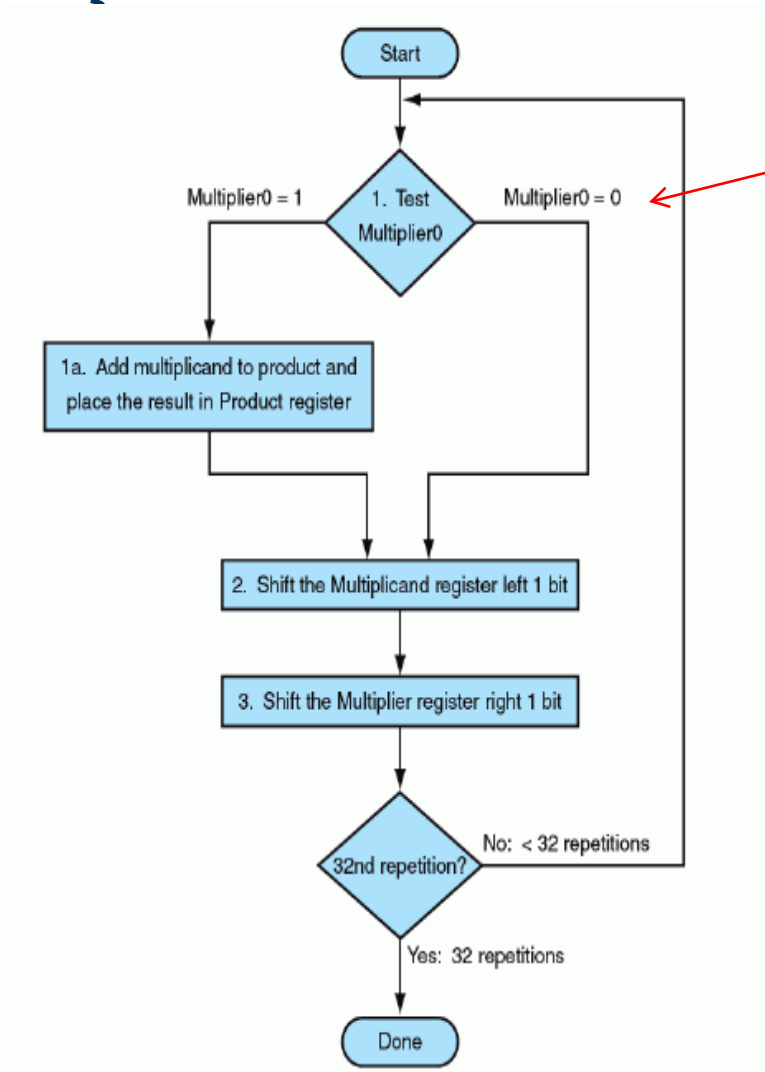
- Condições de *overflow (resultado de uma operação não pode ser representado com o hardware disponível)* para adição e subtração.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Exceções (Interrupção)

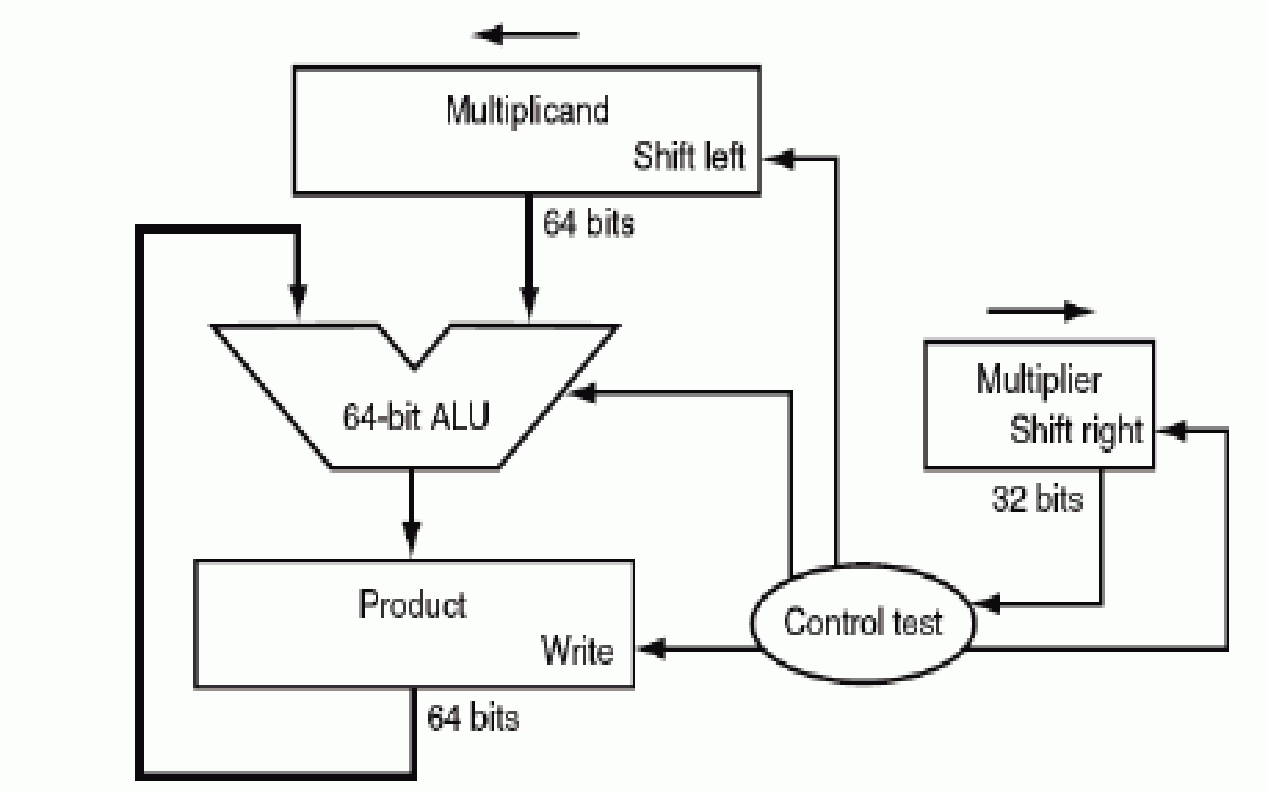
- add, addi, e sub causam exceções no *overflow*.
- addu, addiu e subu não causam.
- EPC (*exception program counter*) contém o endereço da instrução que causou a exceção.
- mfc0 (*move from system control*) -> usada para copiar o EPC para um registrador de uso geral para que o *software* possa retornar à instrução através da instrução jr.
- Registradores reservados para o SO: $k0 e $k1. As rotinas de exceção colocam o endereço de retorno em um desses registradores (*Ver detalhamento na pg. 132).

Algoritmo Sequencial de Multiplicação



Bit menos significativo

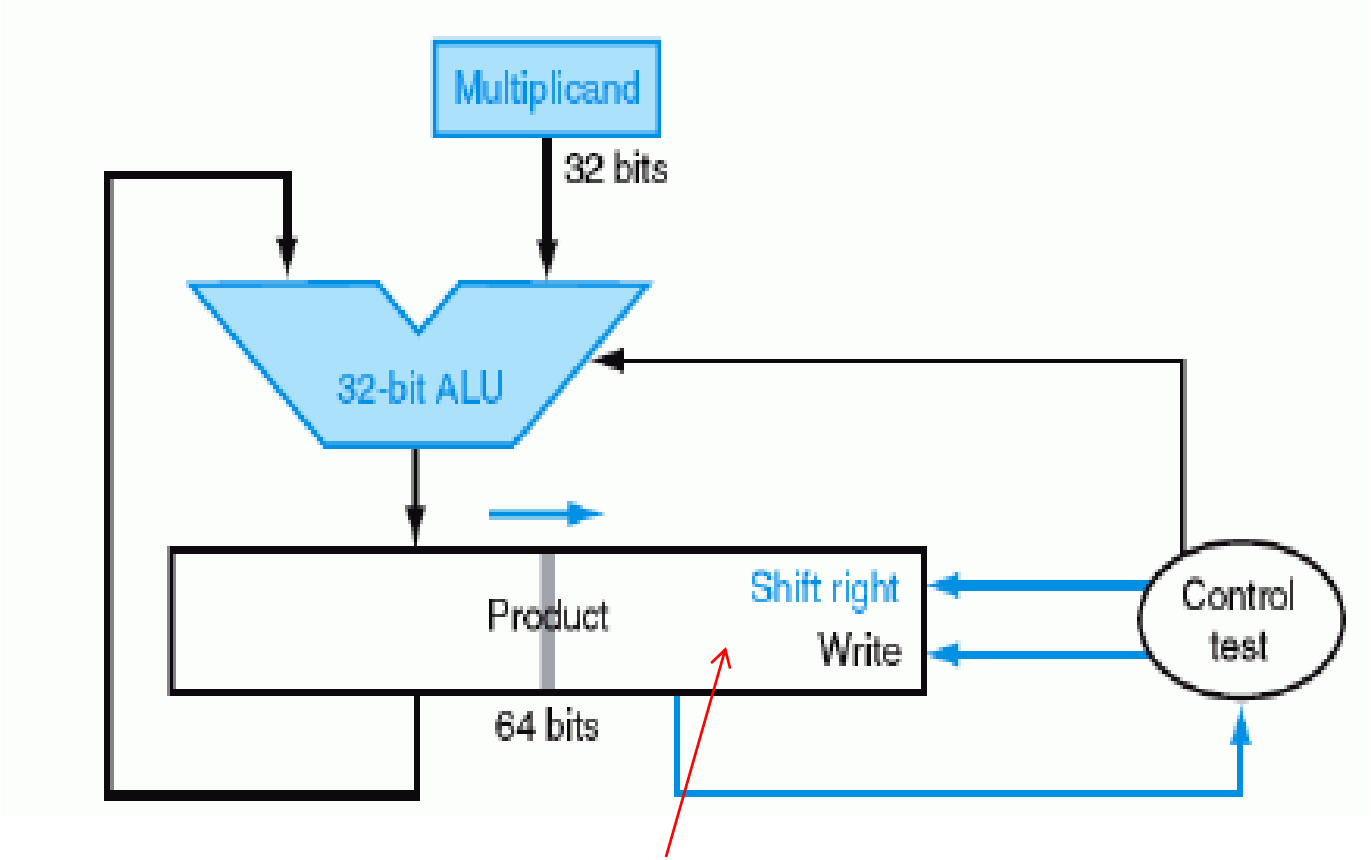
Hardware de Multiplicação



Observações

- As três etapas anteriores são repetidas 32 vezes para obter o produto. Se cada etapa usar um ciclo de *clk*, o algoritmo exigirá aproximadamente 100 ciclos de *clk* para multiplicar dois números de 32 bits.

*Pesquisar: Versão Refinada do *Hardware* de Multiplicação

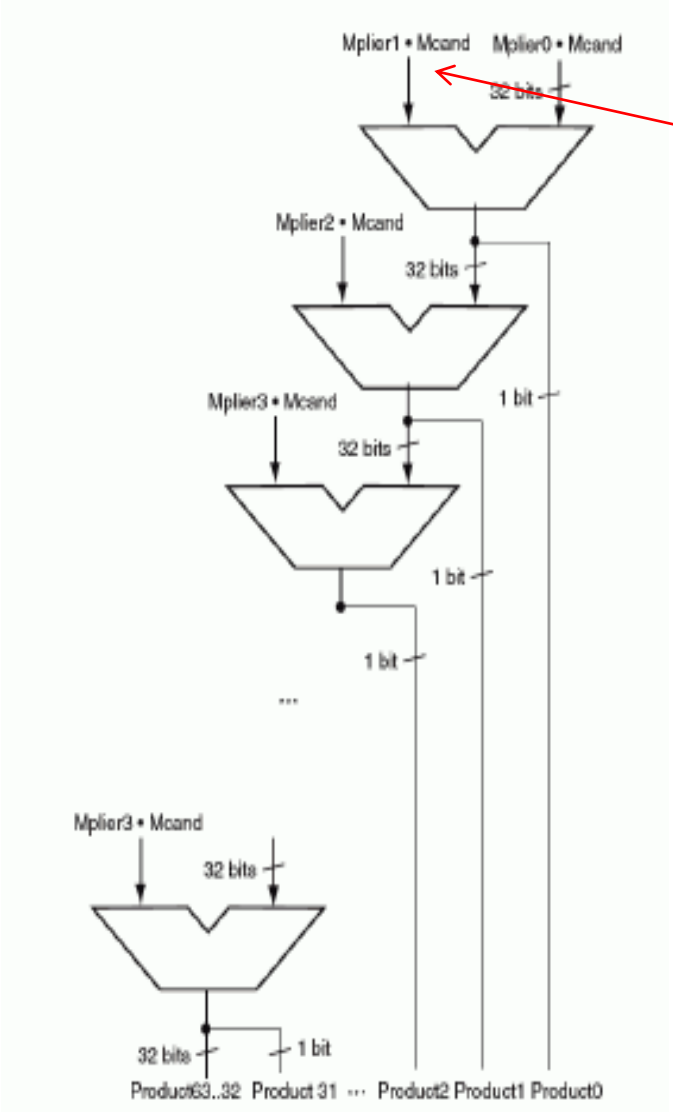


multiplicador

Exemplo de Multiplicação de 4 bits usando o Algoritmo: 2_{10} x 3_{10}

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + 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 ⇒ Prod = Prod + 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 ⇒ 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 ⇒ 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

Hardware de Multiplicação Rápida



0?

O *hardware* "desenrola o *loop*" para usar 32 somadores. Cada somador produz uma soma de 32 bits e um bit de *carry*. O bit menos significativo é o bit de produto, e o *carry* com os 31 bits mais significativos da soma são passados adiante para o próximo somador.

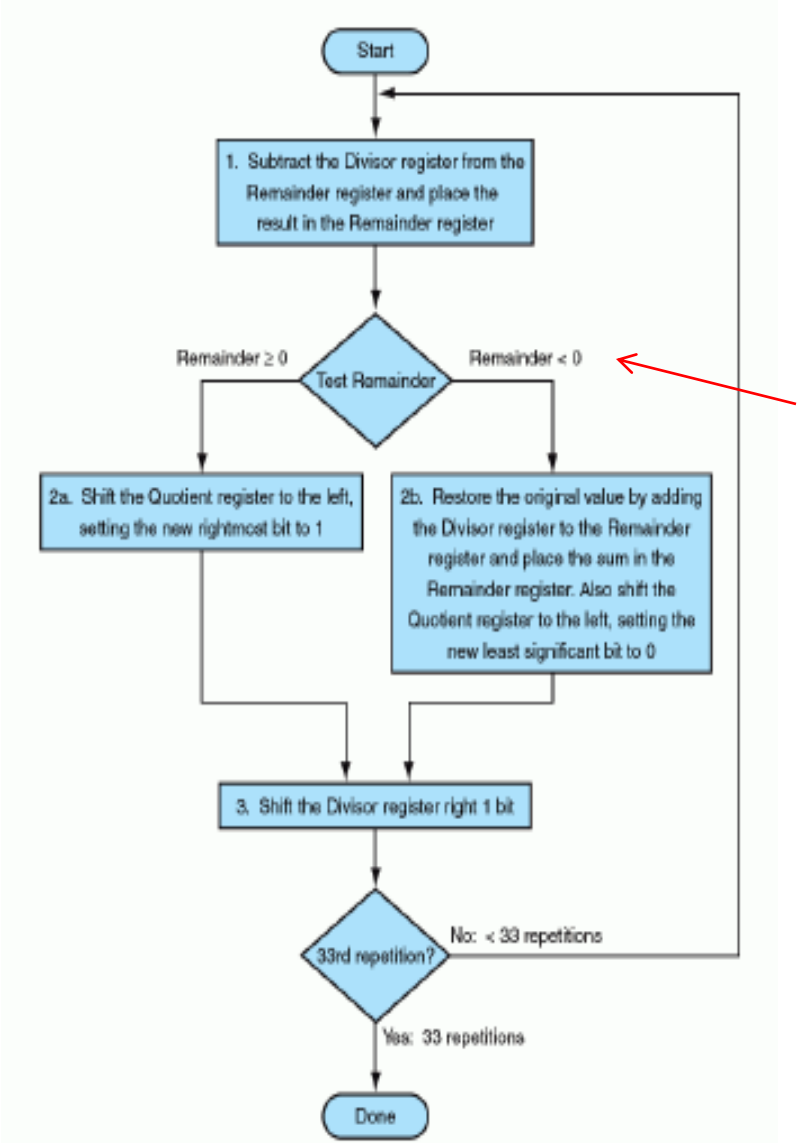
*Pesquisar

- Somador com vai um antecipado!

Multiplicação no MIPS

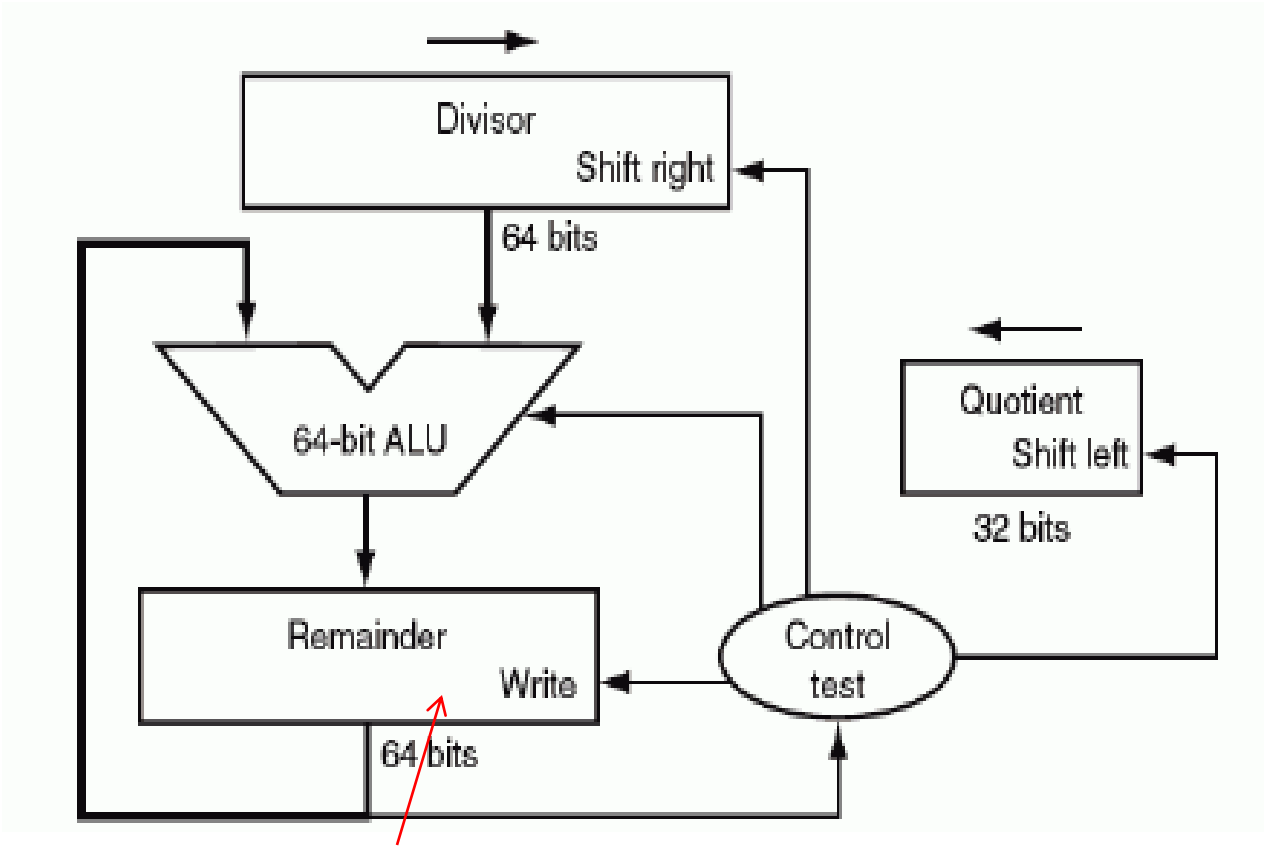
- Hi e Lo: par separado de registradores de 32 bits para armazenar o produto de 64 bits.
- mult (*multiply*).
- multu (*multiply unsigned*).
- mflo (*move from lo*).
- mfhi (*move from hi*).
- Obs.: as duas instruções de multiplicação ignoram o *overflow*. O teste deve ser realizado por *software*.

Algoritmo para Divisão



Testa se o divisor cabe no dividendo.

Hardware

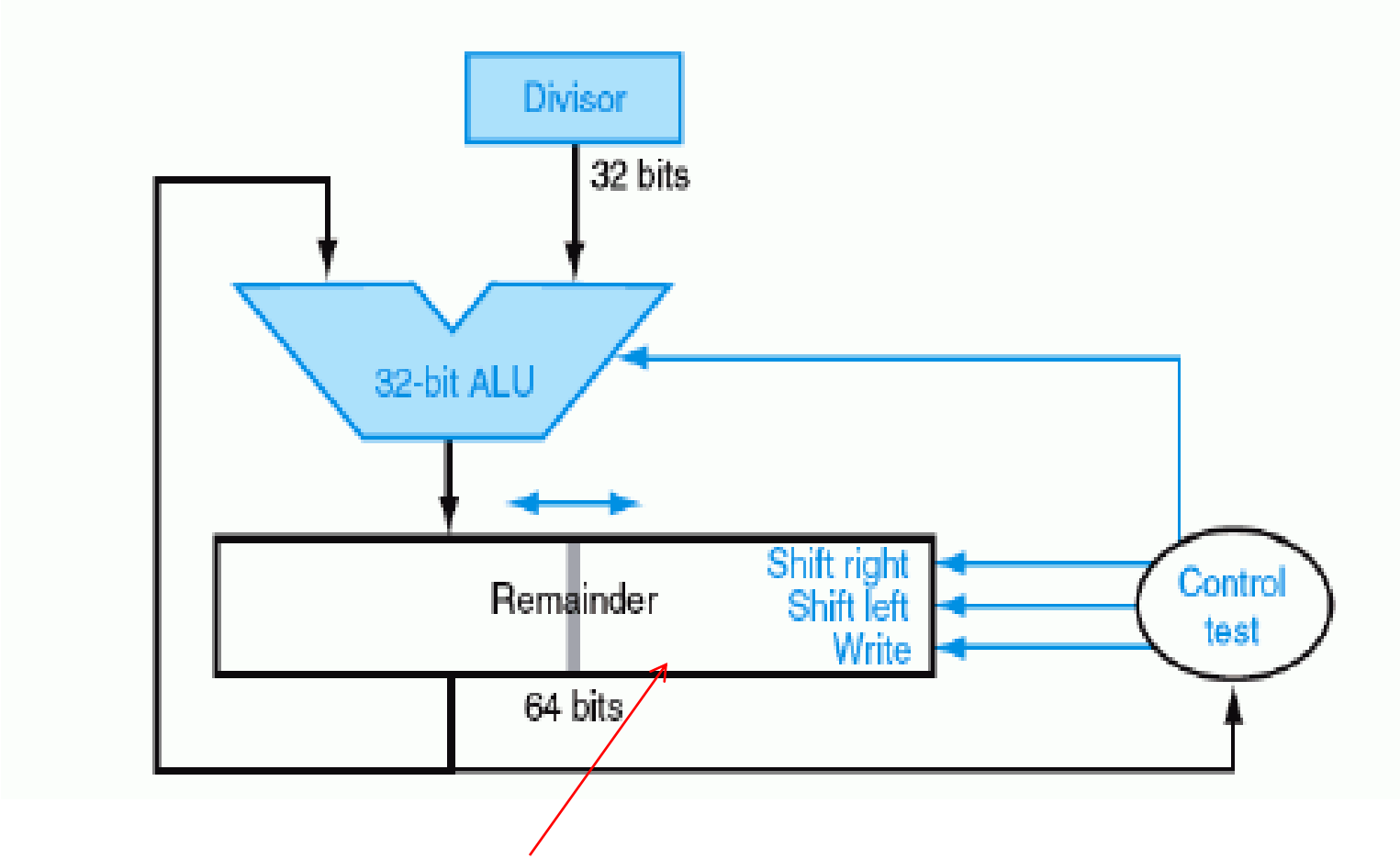


Resto: inicializado
com o dividendo

Exemplo: 7/2

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	(1)110 0111
	2b: Rem < 0 ⇒ +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	(1)111 0111
	2b: Rem < 0 ⇒ +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	(1)111 1111
	2b: Rem < 0 ⇒ +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	(0)000 0011
	2a: Rem ≥ 0 ⇒ 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	(0)000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

*Pesquisar: Versão Melhorada do *Hardware* para Divisão e Multiplicação



Divisão no MIPS

- Hi: contém o resto.
- Lo: contém o quociente.
- div (*divide*).
- divu (divide unsigned).
- Uso de mflo e mfhi para colocar o resultado desejado em um registrador de uso geral.
- Instruções de divisão no MIPS ignoram o *overflow*.

Resumo das Instruções Vistas

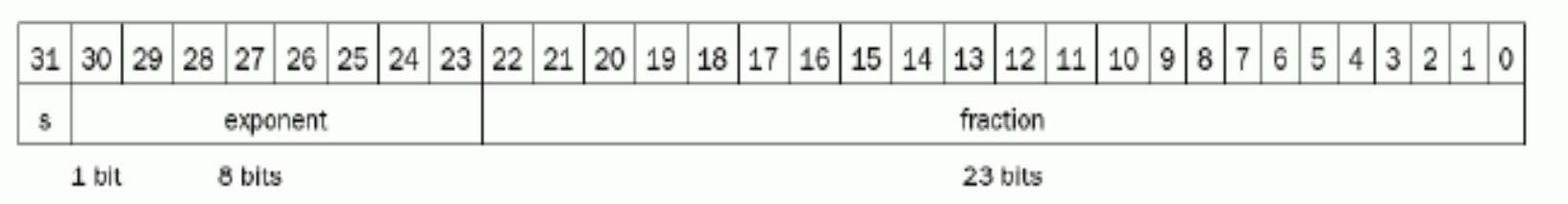
MIPS assembly language

Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo

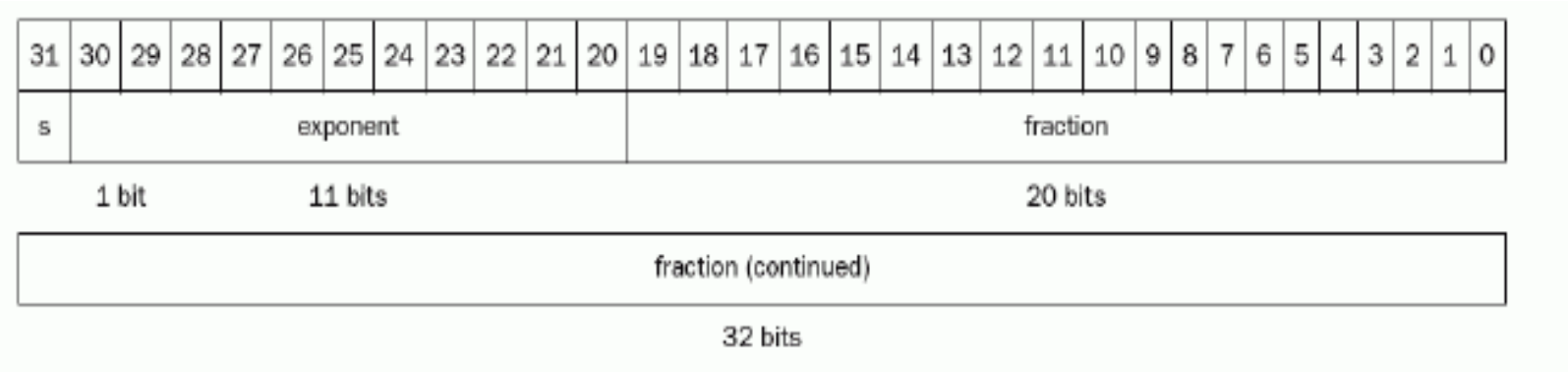
Ponto Flutuante (Precisão Simples)

*Pesquisar: *Overflow x Underflow*

- Representação sinal-magnitude:



Ponto Flutuante (Precisao Dupla)



Padrão IEEE 754

- Deixa implícito o *bit* 1 inicial dos números binários normalizados. Assim, o número tem na realidade 24 *bits* (significando) de largura na precisão simples (1 implícito e fração de 23 *bits*) e 53 *bits* de extensão na precisão dupla (1+52).
- Utiliza símbolos especiais para representar eventos incomuns (div/0, 0/0, inf-inf) -> ±inf, NaN. O maior expoente é reservado para esses símbolos.

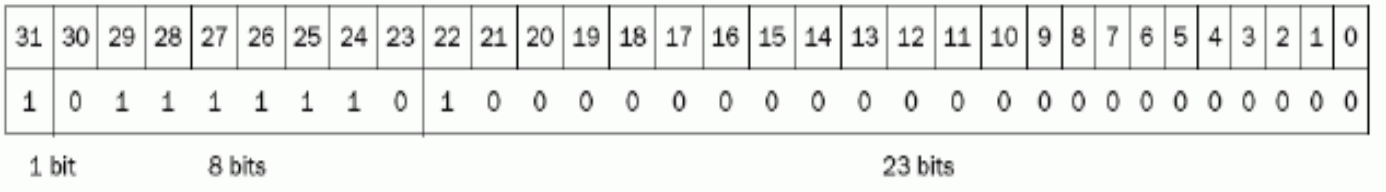
Representação em Ponto Flutuante

- Os projetistas do IEEE 754 quiseram uma representação de PF que pudesse ser facilmente processada por comparações de inteiros, especialmente para ordenação.
- A notação desejada precisa representar o expoente mais negativo como $00..00_2$ e o mais positivo como $11..11_2$. Logo:

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

Exemplo

- Mostre a representação binária IEEE 754 do número -0.75_{10} em precisão simples e dupla.
- Sol.:
  - Em notação científica: $-0.11x2^{0}$ e em notação normalizada: $-1.1x2^{-1}$.
  - Representação em precisão simples: $(-1)^{s}x(1+\text{Fração})x2^{(\exp-127)}$.
  - Logo: $-1^{(1)}x(1+.1000\ 0000\ 0000\ 0000\ 0000\ 000_{bin})x2^{(126-127)}$.



Exemplo

- Em precisão dupla:

$$(-1)^1 \times (1 + .1000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000_{two}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

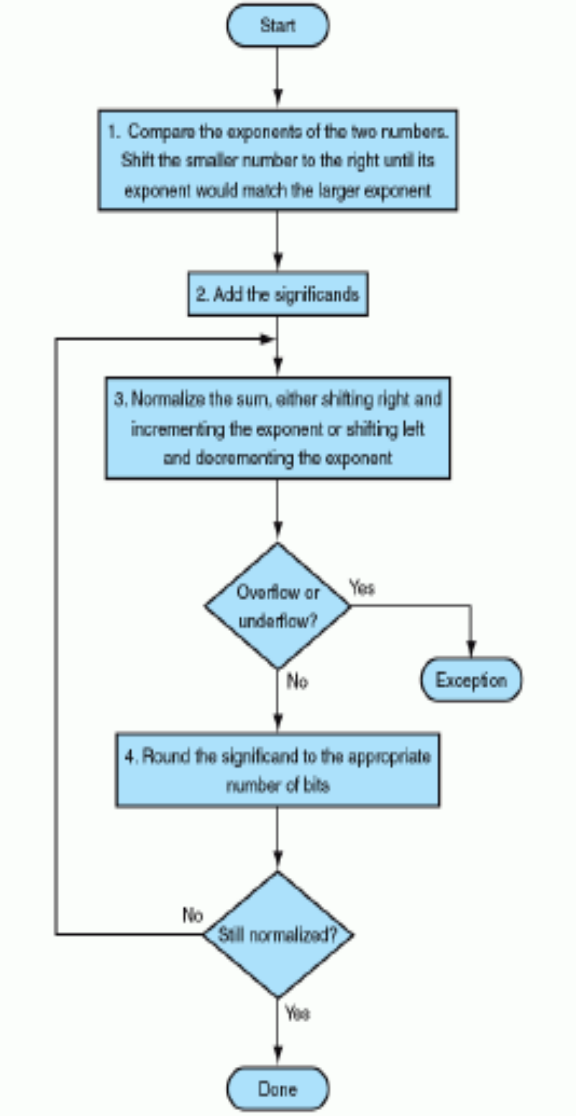
32 bits

Exercício* (Para Casa)

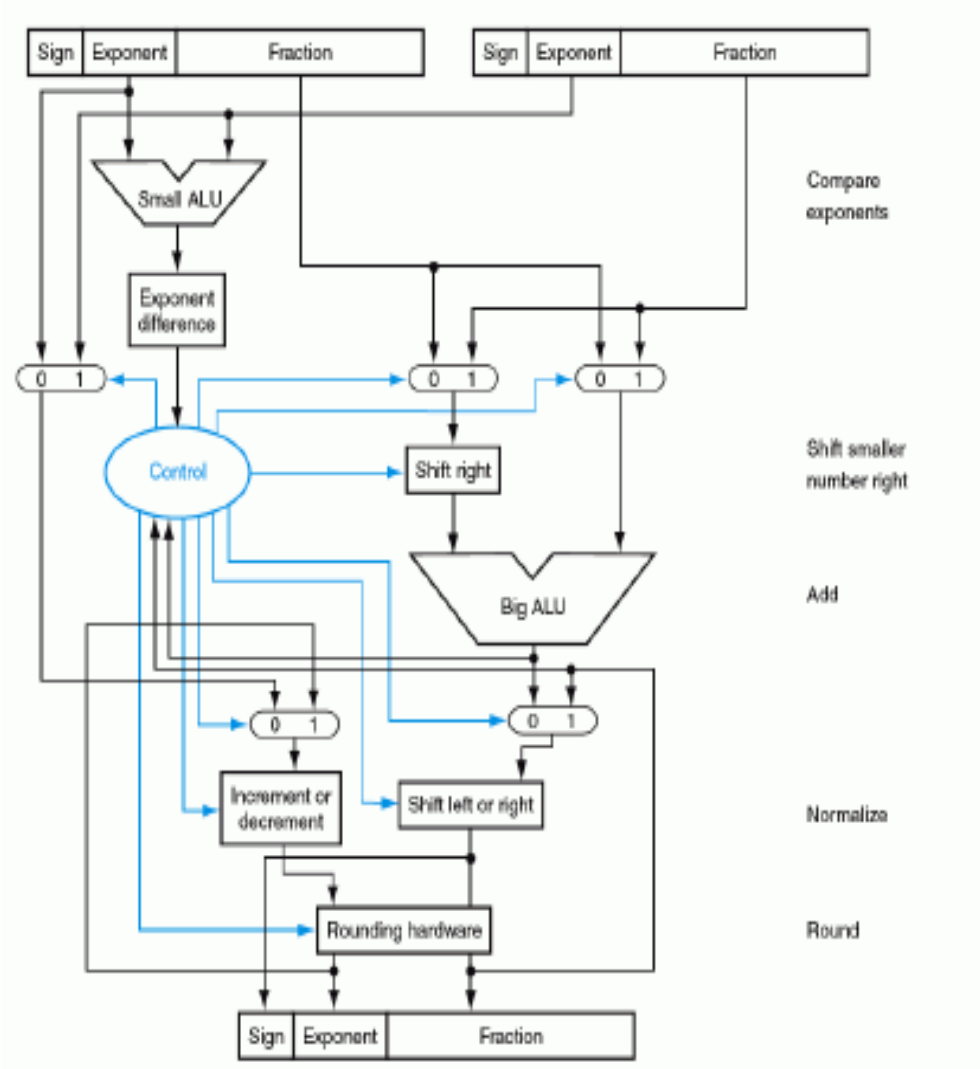
- Qual número decimal é representado pelo seguinte *float* de precisão simples?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

Adição em Ponto Flutuante



*Hardware: Estudar seu funcionamento



Exemplo

- 0.5 - 0.4375? Usar 4 *bits* de precisão.
- Sol.:

  $1x2^{-1} – 1.11x2^{-2}$,

  - deslocar significando do número de menor expoente à direita:$-0.111x2^{-1}$,

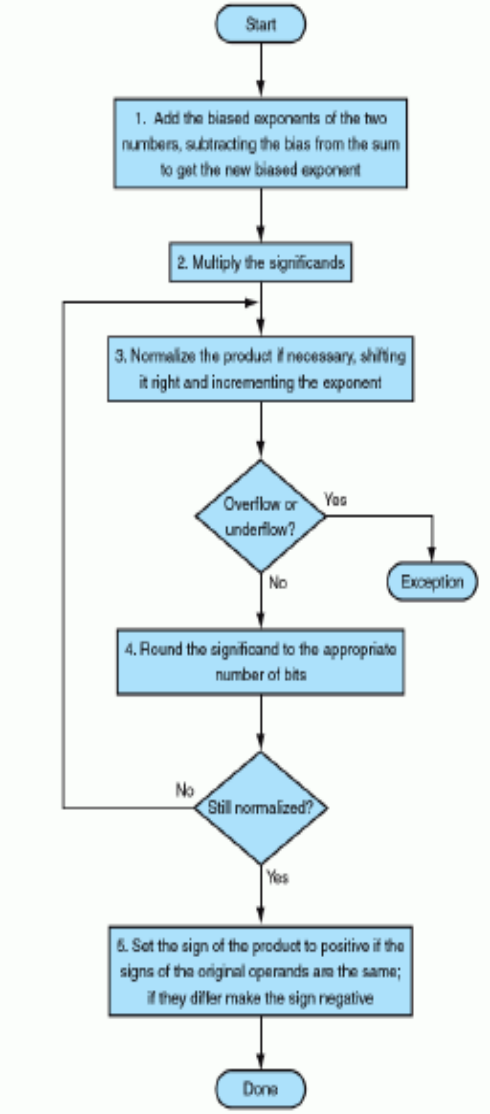
  - somar os significandos: $0.001x2^{-1}$,

  - normalizar soma e verificar se houve *overflow* ou *underflow*: $1.000x2^{-4}$,

  - 127>=-4>=-126, logo, não existe *ovf* nem *und*.

  - Arrendondar a soma: $1.000x2^{-4}$. Portanto, a soma já cabe em 4 *bits*.

Multiplicação em Ponto Flutuante



Exemplo

- $1.000x2^{-1}$ x $-1.110x2^{-2}$? Usar precisão de 4 bits.
- Sol.:
  - Somar expoentes sem *bias*: -1+-2=-3. Usando a representação deslocada: -1+127 + -2+ 127 -127 = 124.
  - Multiplicar os significandos: $1.000x1.110=1.110000x2^{-3}$. Em 4 *bits*: $1.110x2^{-3}$.
  - 127>=-3>=-126, logo, não existe *ovf* nem *und*. (ou: 254>=124>=1).
  - O arredondamento não causa mudança.
  - Como os sinais dos operandos originais diferem, torne o sinal do produto negativo. Logo: $-1.110x2^{-3}$.

Instruções de Ponto Flutuante no MIPS

- Adição simples (add.s) e dupla (add.d).
- Subtração simples (sub.s) e dupla (sub.d).
- Multiplicação simples (mul.s) e dupla (mul.d).
- Divisão simples (div.s) e dupla (div.d).
- Comparação simples (c.x.s) e dupla (c.x.d) onde x->eq, neq,lt,le,gt,ge.
- Desvio verdadeiro (belt) e falso (bclf).

Registradores

- $f0, $f1, $f2…$f31
- *Loads* e *stores*: lwc1 e swc1.
- Registrador de precisão dupla: um par de registradores (par e ímpar) de precisão simples.
- Exemplo:

```
lwc1    $f4,x($sp)  # Load 32-bit F.P. number into F4
lwc1    $f6,y($sp)  # Load 32-bit F.P. number into F6
add.s   $f2,$f4,$f6 # F2 = F4 + F6 single precision
swc1    $f2,z($sp)  # Store 32-bit F.P. number from F2
```

Resumo

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 x $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 x $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
Condi-tional 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

*Pesquisar!

- *Bits*: guarda, arredondamento e *sticky*.