4. ARITMÉTICA PARA COMPUTADORES

- **Representação de números sinalizados e não sinalizados**

$$N = \sum_{i=0}^{n-1} d_i \times base^i$$

Exemplo

$$1011_2 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 11_{10}$$

- **Números sem sinal**

Representação com 32 bits – palavra do MIPS

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 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	1 0 1 1

Maior número de 32 bits ➔ $2^{32} – 1 = 4.294.967.295_{10}$

- **Números sinalizados**

S	
0	0 0 0 0000 0000 0000 0000 0000 0000 00001011

Números sinalizados de 32 bits ➔ (-2^{31}) até $(2^{31} – 1)$
➔ -2.147.483.648 até 2.147.483.647

- **Complemento de dois**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$
.....
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2.147.483.645_{10}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2.147.483.646_{10}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2.147.483.647_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2.147.483.648_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2.147.483.647_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2.147.483.646_{10}$
.....
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$

- **Conversão binária (representação em complemento de dois ➔ decimal)**

$$N=(x_{31} \text{ X } -2^{31})+(x_{30} \text{ X } 2^{30})+(x_{29} \text{ X } 2^{29})+ ...+(x_1 \text{ X } 2^1) + + (x_0 \text{ X } 2^0)$$

onde x_i é o i-ésimo dígito do número.

Exemplo

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 = ?_{10}$

$$N=(1 \text{ X } -2^{31})+(1 \text{ X } 2^{30})+(1 \text{ X } 2^{29})+ 2^3 + 2^2+0+0 =$$
$$= -2.147.483.648 + 2.147.483.644 = -4_{10}$$

- **Comparação de números sinalizados e não sinalizados**

  - **Instruções:**

slt → set on less than
slti → set on less than immediate
} → **comparam inteiros sinalizados**

sltu → set on less than
sltiu → set on less than immediate
} → **comparam inteiros não sinalizados**

Exemplo

$s0 → 1111 1111 1111 1111 1111 1111 1111 1111_2
$s1 → 0000 0000 0000 0000 0000 0000 0000 0001_2

O que acontece com as instruções :

```
slt   $t0, $s0,$s1  # comparação sinalizada
sltu  $t1, $s0,$s1  # comparação não sinalizada
```

Solução:

O valor em $s0 representa –1 se for um inteiro sinalizado e 4.294.967.295 se for não sinalizado.
O valor em $s1 representa 1 em ambos os casos.
Então $t0 tem o valor 1, pois –1 < 1 e $t1 tem o valor 0 pois 4.294.967.295 > 1.

- **Complemento dois pela inversão e incremento**

2_{10} = 0000 0000 0000 0000 0000 0000 0010

1111 1111 1111 1111 1111 1111 1101
+ 1

-2_{10} ➔ 1111 1111 1111 1111 1111 1111 1110

-2_{10} = 1111 1111 1111 1111 1111 1111 1110

0000 0000 0000 0000 0000 0000 0001
+ 1

2_{10} ➔ 0000 0000 0000 0000 0000 0000 0010

- **Extensão do sinal**

2_{10} = 0000 0000 0000 0010 com 16 bits, estendendo para 32 bits ➔ 0000 0000 0000 0000 0000 0000 0010

-2_{10} = 1111 1111 1111 1110 com 16 bits, estendendo para 32 bits ➔ 1111 1111 1111 1111 1111 1111 1110

- **Conversão binária ←→ hexadecimal**

Figura 4.1 – Tabela de conversão hexadecimal

Hexa	Bin	Hexa	Bin	Hex	Bin	Hex	Bin
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

Exemplo1

e c a d 8 6 4 20_{16}

1110 1100 1010 1000 0110 0100 0010 0000_2

Exemplo2

0001 0011 0101 0111 1001 1011 1101 1111_2

1 3 5 7 9 b d f

- **Figura 4.2 Resumo – arquitetura MIPS**

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 words 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 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
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
Unconditional 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

FIGURE 4.2 MIPS architecture revealed thus far. Color indicates portions from this section added to the MIPS architecture revealed in Chapter 3 (Figure 3.20 on page 155). MIPS machine language is listed in the back endpapers of this book.

- **Adição e subtração**

  - **7 + 6**

```
      0000 0000 0000 0000 0000 0000 0000 0111
+     0000 0000 0000 0000 0000 0000 0000 0110
_____________________________________________
      0000 0000 0000 0000 0000 0000 0000 1101
```

  - **7 – 6**

```
      0000 0000 0000 0000 0000 0000 0000 0111
-     0000 0000 0000 0000 0000 0000 0000 0110
_____________________________________________
      0000 0000 0000 0000 0000 0000 0000 0001
```

  - **7 – 6 = 7 + (-6)**

```
      0000 0000 0000 0000 0000 0000 0000 0111
+     1111 1111 1111 1111 1111 1111 1111 1010
_____________________________________________
      0000 0000 0000 0000 0000 0000 0000 0001
```

- **Overflow**

Figura 4.4 – Condições de overflow

Operação	Operador A	Operador B	Resultado Indicando Overflow
A+B	= 0	= 0	< 0
A+B	< 0	< 0	= 0
A-B	= 0	< 0	< 0
A-B	< 0	= 0	= 0

- **Figura 4.5 – Arquitetura MIPS - RESUMO**

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, for example, 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 words 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; 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	Used to copy Exception PC plus other special registers
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 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
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
Unconditional 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

FIGURE 4.5 MIPS architecture revealed thus far. Color indicates the portions revealed since Figure 4.2 on page 219. MIPS machine language is also listed on the back endpapers of this book.

- **Operações Lógicas**

  - **Instrução *sll* ➔ shift left logical**

    sll $t2,$s0,8 # $t2 ← $s0 deslocado de 8 bits, para a esquerda.

op	rs	rt	rd	shamt	funct
0	0	16	10	8	0

$s0 ➔ 0000 0000 0000 0000 0000 0000 0000 1101
$t2 ➔ 0000 0000 0000 0000 0000 1101 0000 0000

  - **Instrução *and* e *or***

$t2 ➔ 0000 0000 0000 0000 0000 1101 0000 0000
$t1 ➔ 0000 0000 0000 0000 0011 1100 0000 0000

and $t0,$t1,$t2 # $t0 ← $t1 & $t2
$t0 ➔ 0000 0000 0000 0000 0000 1100 0000 0000

or $t0,$t1,$t2 # $t0 ← $t1 | $t2
$t0 ➔ 0000 0000 0000 0000 0011 1101 0000 0000

- **Figura 4.6 – operações Lógicas e correspondência com C**

Operador Lógico	Operador em C	Instrução MIPS
Shift left	<<	sll
Shift right	>>	srl
Bit by bit AND	&	and, andi
Bit by bit OR	|	or, ori

- **Figura 4.7 – Arquitetura MIPS - RESUMO**

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 words 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; 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	Used to copy Exception PC plus other special registers
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
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg 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
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 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
Condi-tional 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; natural numbers
	set less than immediate unsigned	sltiu $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural 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

FIGURE 4.7 MIPS architecture revealed thus far. Color indicates the portions introduced since Figure 4.5 on page 224. MIPS machine language is also listed on the back endpapers of this book.

- **Manipulação de bits**

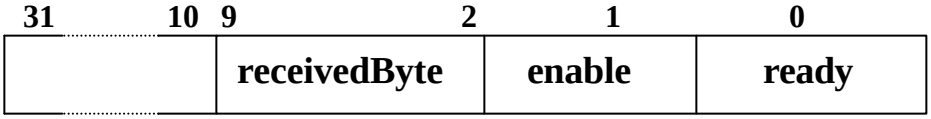
  Exemplo

  Seja o trecho de programa em C abaixo:

```
int data;
struct
{
    unsigned int ready:          1;
    unsigned int enable:         1;
    unsigned int receivedByte:8;
}receiver;
.....
    data = receiver.receivedByte;
    receiver.ready = 0;
    receiver.enable = 1;
```

  Este código aloca três campos para registro receiver, 1 bit para o campo chamado *ready*, 1bit para o cachamado *enable* e 8 bits chamado *receivedByte*. No corpo do programa, o código copia *receivedByte* para *data*, atribui a *ready* o valor 0 e a *eanble* o valor 1. Como fica o código compilado para o MIPS? Assumir que para *data* e *receiver* são alocados $s0 e $s1.

Solução



Primeiro isolaremos 8bits do *receivedByte.*

```
sll    $s0,$s1,22      # move os 8 bits para o fim da
                         palavra ( à esquerda )
srl    $s0,$s0,24      # move os 8 bits para o início da
                         palavra ( à direita)
```

A próxima instrução limpa o bit menos significativo e a última instrução seta o bit vizinho.

```
andi   $s1,$s1,FFFEh    # bit 0 = 0
ori    $s1,$s1,0002h    # bit 1  = 1
```

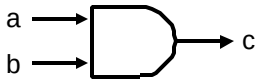
Outra solução:

```
srl    $s0,$s1,2
andi   $s0,$s0, 0x0ff
```

- **Unidade Lógica Aritmética**

  - **ALU de 1 bit - Figura 4.8 – blocos usados para construção de uma ALU**

1. AND gate (c = a . b)



a	b	c = a . b
0	0	0
0	1	0
1	0	0
1	1	1

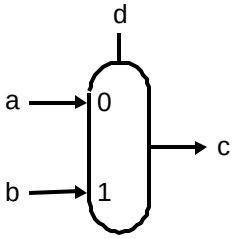
2. OR gate (c = a + b)

a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter (c = a)

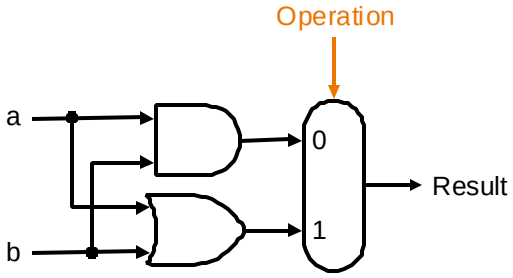
a	c = ā
0	1
1	0

4. Multiplexor
   (if d = = 0, c = a;
   else c = b)

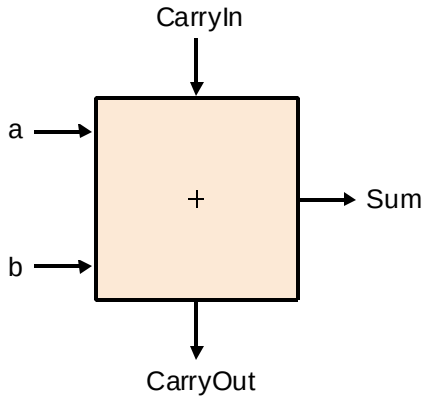


d	c
0	a
1	b

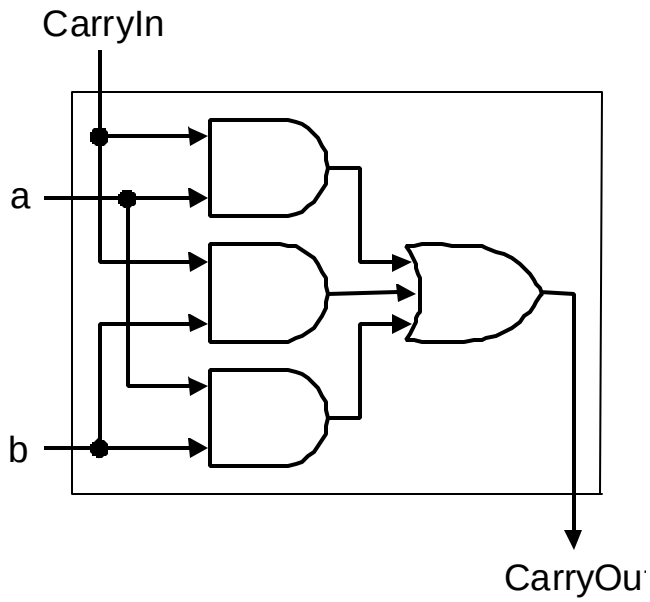
- **Figura 4.9 – Unidade and-or de 1 bit**



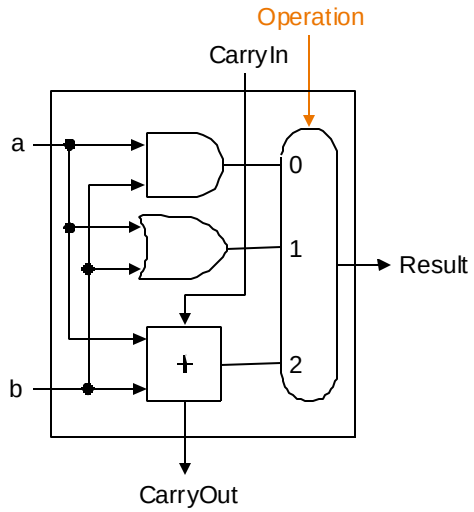
- **Figura 4.10 – Somador de 1 bit**



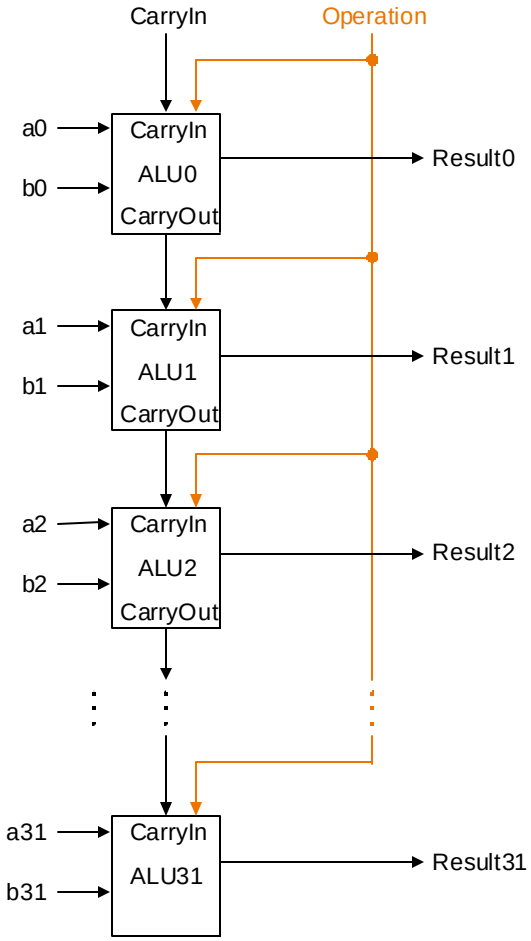
- **Figura 4.13 – Gerador de Carry**



- **ALU de 32 bits - Figura 4.14 – ALU de 1 bit**

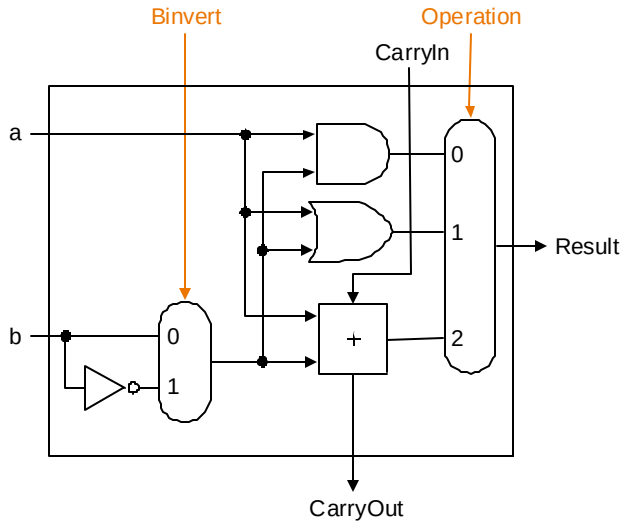


- **ALU de 32 bits - Ripple Carry**



- **Subtrator ➔ Complemento de 2**

  $a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$



- **Instruções *set on less than* ➔ *slt***

  Esta instrução gera 1 se rs < rt e 0, caso contrário.

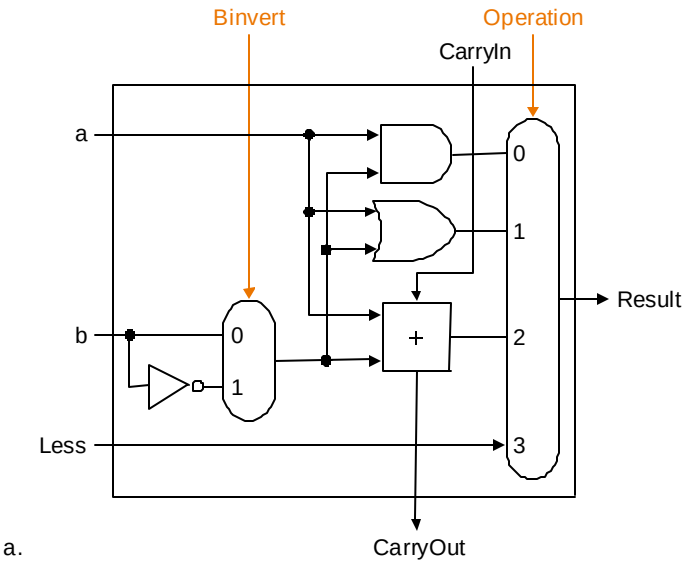
  Introduz-se uma nova entrada no multiplex de saída ➔ Less.

  Para todos os 31 bits mais significativos seu valor será zero.

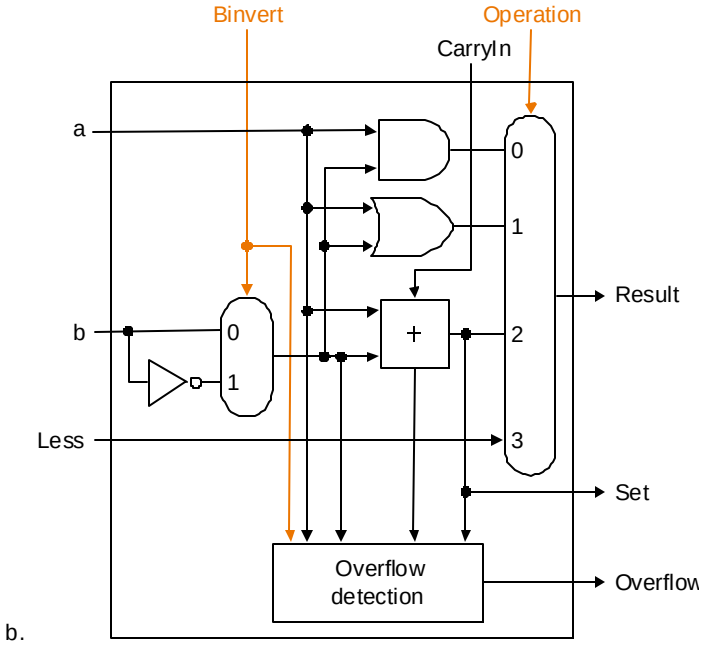
  O último bit será setado ou resetado, decorrende do valor da comparação de a com b. Se a < b então a-b < 0 (negativo) então e o bit será 1, caso contrário a - b > 0 (positivo)e o bit será 0.

  Para isto precisamos de um bit (o mais significativo da ALU) para setar ou resetar o bit menos significativo

- **Detecção de overflow.**

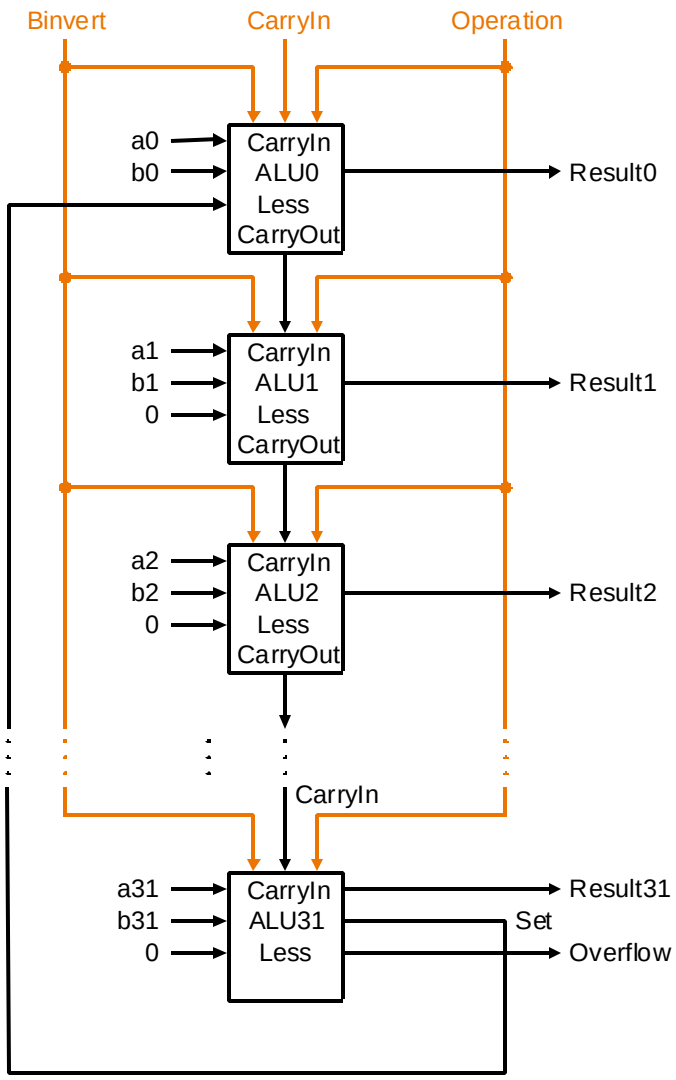


a.

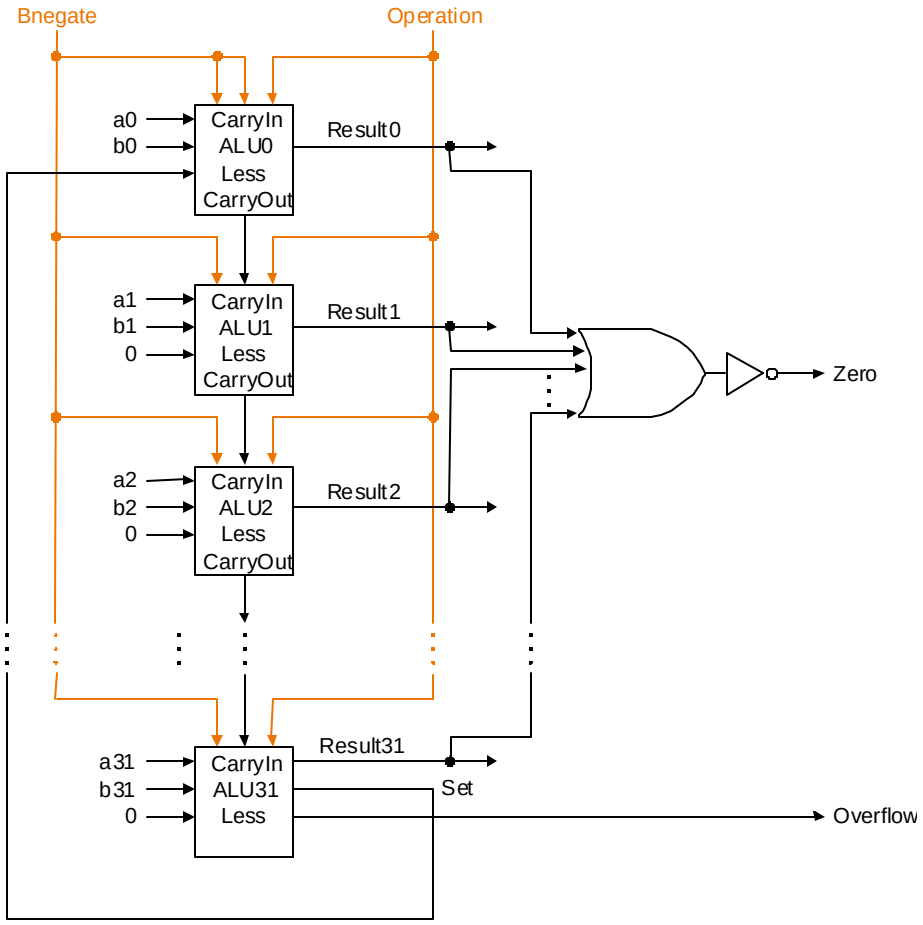


b.

- **ALU de 32 bits**



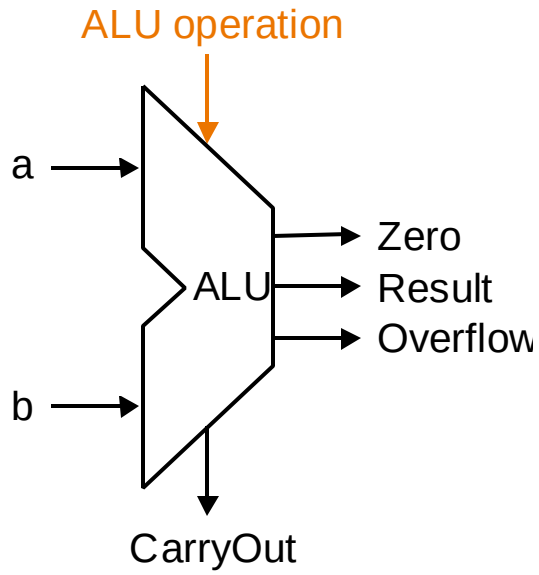
- **Detector de Zero**



- **Figura 4-20 – Sinal de Controle de operações da ALU**

Linha de Controle da ALU	Função
000	**and**
001	**or**
010	**add**
110	**substract**
111	**set on less than**

- **Símbolo de uma ALU**



- **Somador Carry Lookahead ➔ Calcular os carry's em função apenas das entradas.**

  Supondo um somador de 4 bits, temos (1° nível):
  $c_{i+1} = (b_i . c_i) + (a_i . c_i) + (a_i . b_i)$

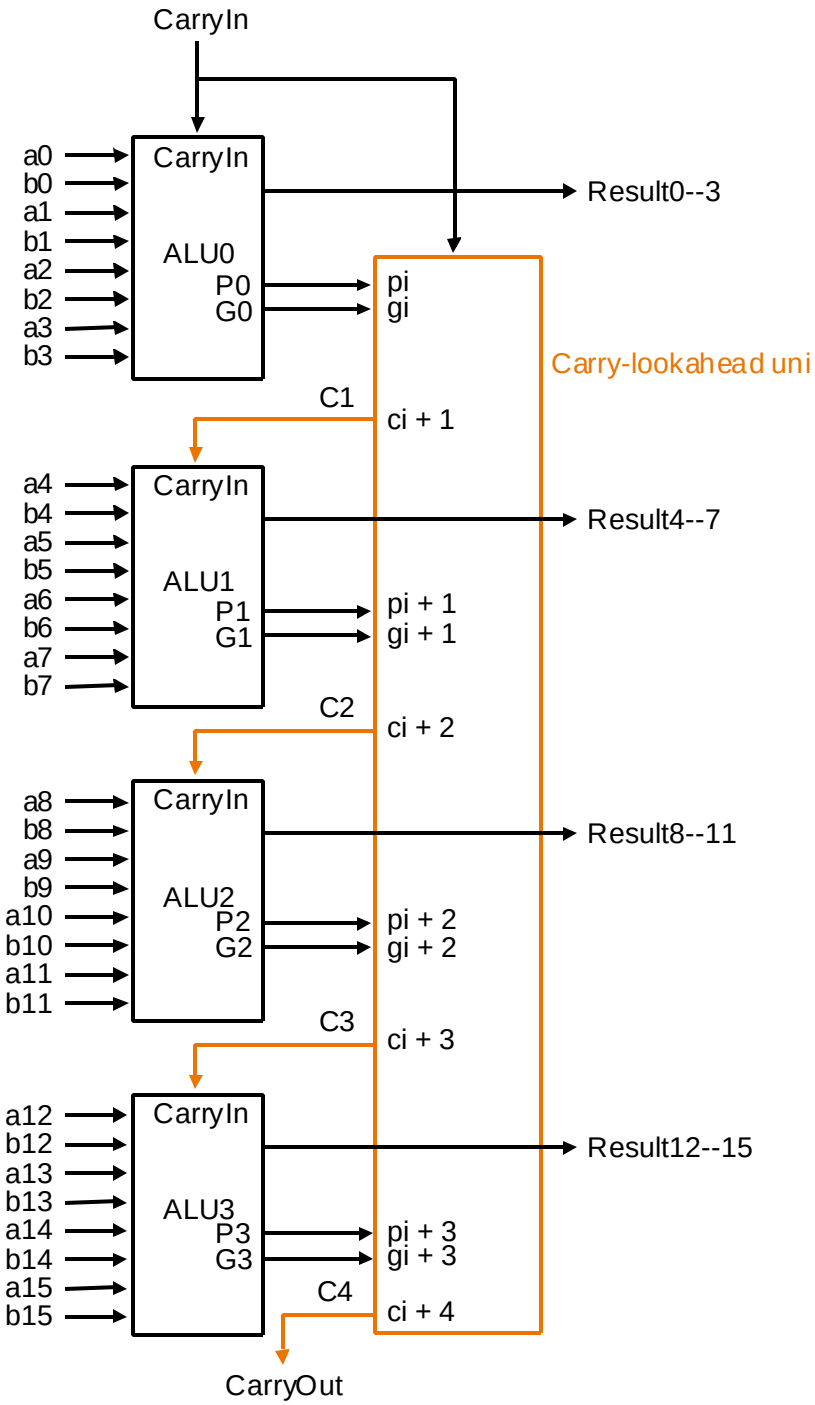
  $c_1 = (b_0 . c_0) + (a_0 . c_0) + (a_0 . b_0) =$
  $c_1 = (a_0 + b_0) . c_0 + (a_0 . b_0) = g_0 + (p_0 . c_0)$
  $c_2 = g_1 + (p_1 . c_1) = g_1 + p_1 (g_0 + (p_0 . c_0))$
  $c_2 = g_1 + (p_1 . g_0) + (p_1 . p_0 . c_0)$
  $c_3 = g_2 + (p_2 . g_1) + (p_2 . p_1 . g_0) + (p_2 . p_1 . p_0 . c_0)$
  $c_4 = g_3 + (p_3 . g_2) + (p_3 . p_2 . g_1) + (p_3 . p_2 . p_1 . g_0) +$
  $+ (p_3 . p_2 . p_1 . p_0 . c_0)$

  Supondo um somador de 16 bits usando 4 módulos de 4 bits visto acima , temos (2° nível):

  $C_1 = G_0 + (P_0 . c_0)$
  $C_2 = G_1 + (P_1 . G_0) + (P_1 . P_0 . c_0)$
  $C_3 = G_2 + (P_2 . G_1) + (P_2 . P_1 . G_0) + (P_2 . P_1 . P_0 . c_0)$
  $C_4 = G_3 + (P_3 . G_2) + (P_3 . P_2 . G_1) + (P_3 . P_2 . P_1 . G_0) +$
  $+ (P_3 . P_2 . P_1 . P_0 . c_0)$

  onde:
  $P_0 = p_3 . p_2 . p_1 . p_0$
  $P_1 = p_7 . p_6 . p_5 . p_4$
  $P_2 = p_{11} . p_{10} . p_9 . p_8$
  $P_3 = p_{15} . p_{14} . p_{13} . p_{12}$
  $G_0 = g_3 + (p_3 . g_2) + (p_3 . p_2 . g_1) + (p_3 . p_2 . p_1 . g_0)$
  $G_1 = g_7 + (p_7 . g_6) + (p_7 . p_6 . g_5) + (p_7 . p_6 . p_5 . g_4)$
  $G_2 = g_{11} + (p_{11} . g_{10}) + (p_{11} . p_{10} . g_9) + (p_{11} . p_{10} . p_9 . g_8)$
  $G_3 = g_{15} + (p_{15} . g_{14}) + (p_{15} . p_{14} . g_{13}) + (p_{15} . p_{14} . p_{13} . g_{12})$

- **ALU de 16 bits formado com ALU de 4 bits, utilizando carry lookahead**



Exemplo
Determine g*i*, p*i*, P*i* e G*i* para os valores de 16 bits abaixo:
a: 0001 1010 0011 0011
b: 1110 0101 1110 1011

Solução
a: 0001 1010 0011 0011
b: 1110 0101 1110 1011
g*i*: 0000 0000 0010 0011 → a*i* . b*i*
p*i*: 1111 1111 1111 1011 → a*i* + b*i*

P*3* = 1 . 1 . 1 . 1 = 1
P*2* = 1 . 1 . 1 . 1 = 1
P*1* = 1 . 1 . 1 . 1 = 1
P*0* = 1 . 0 . 1 . 1 = 0
G*0* = g*3* + (p*3*.g*2*)+ (p*3*. p*2* .g*1*) + (p*3*. p*2* .p*1* .g*0*)
G*0* = 0 + (1 . 0) + (1 . 0 . 1) + (1 . 0 . 1 . 1) = 0
G*1* = g*7* + (p*7*.g*6*)+ (p*7*. p*6* .g*5*) + (p*7*. p*6* .p*5* .g*4*)
G*1* = 0 + (1 . 0) + (1 . 1 . 1) + (1 . 1 . 1 . 0) = 1
G*2* = g*11* + (p*11*.g*10*)+ (p*11*. p*10* .g*9*) + (p*11*. p*10*. .p*9*.g*8*)
G*2* = 0 + (1 . 0) + (1 . 1 . 0) + (1 . 1 . 1 . 0) = 0
G*3* = g*15* + (p*15*.g*14*)+ (p*15*. p*14* .g*13*) + (p*15*. p*14*. .p*13* .g*12*)
G*3* = 0 + (1 . 0) + (1 . 1 . 0) + (1 . 1 . 1 . 0) = 0

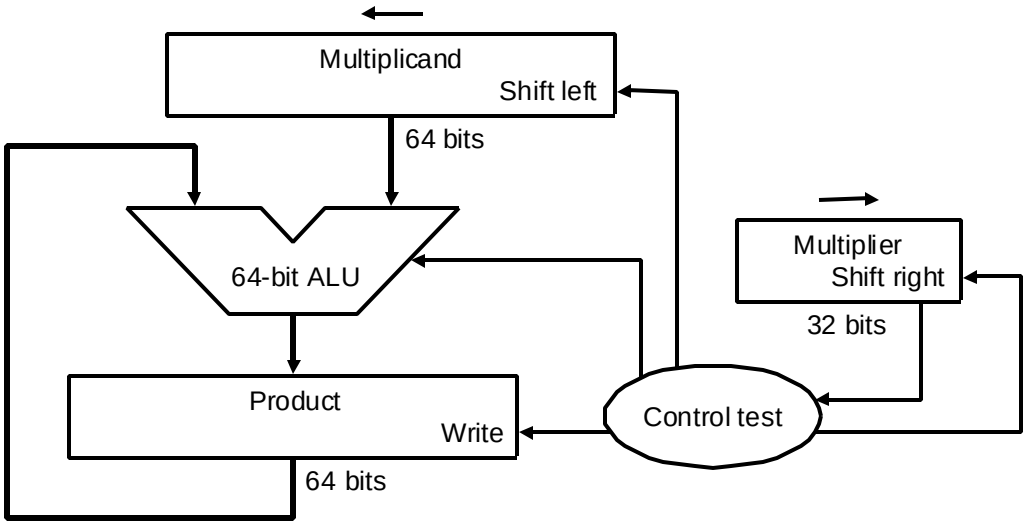
O carry out é :
C*4* = G*3* + (P*3* . G*2*) + (P*3* . P*2* . G*1*) + (P*3* . P*2* . P*1* . . G*0*) + (P*3* . P*2* . P*1* . P*0* . c*0*)
C*4* = 0 + (1 . 0) + (1 . 1 . 1) + (1 . 1 . 1 . 0) + (1 . 1 . 1 . 0 . . 0) = 1

OBS.: Comparar a velocidade do ripple carry e do carry lookahead

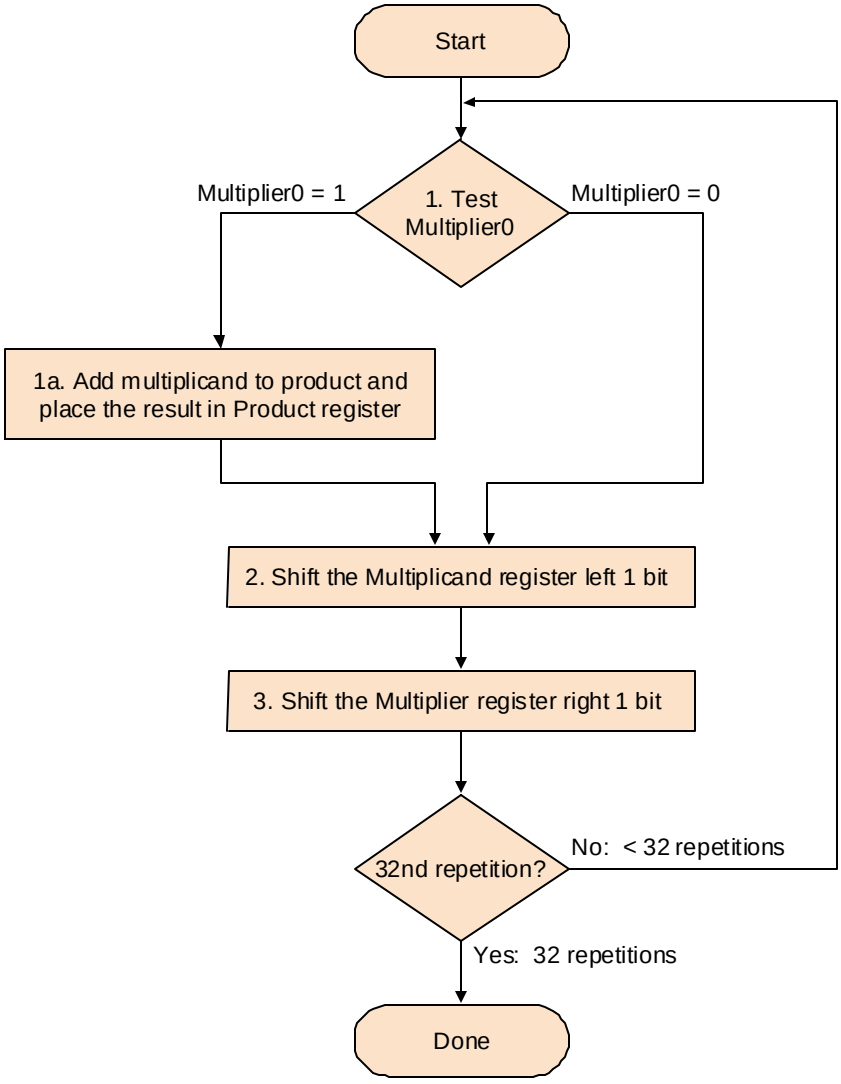
- **Multiplicação**

```
multiplicando      1000
multiplicador   x  1001
                   -------
                   1000
                  0000
                 0000
                1000
               -----------
produto        1001000
```

- **Hardware Multiplicador – Primeira versão**



- **Algoritmo de Multiplicação – Primeira versão**



Exemplo

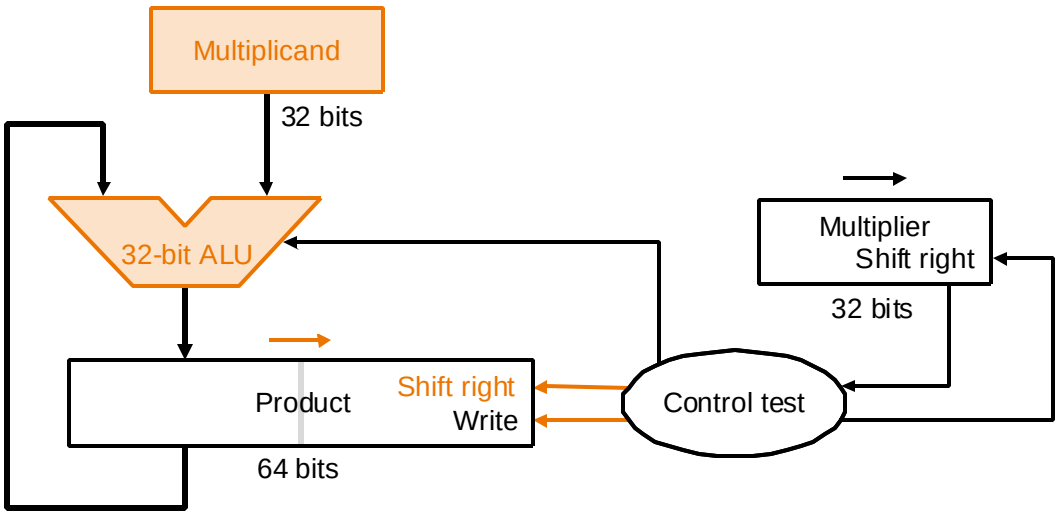
0010_2 x 0011_2 = 0000 0110_2

Figura 4.27 – Resolução do exemplo

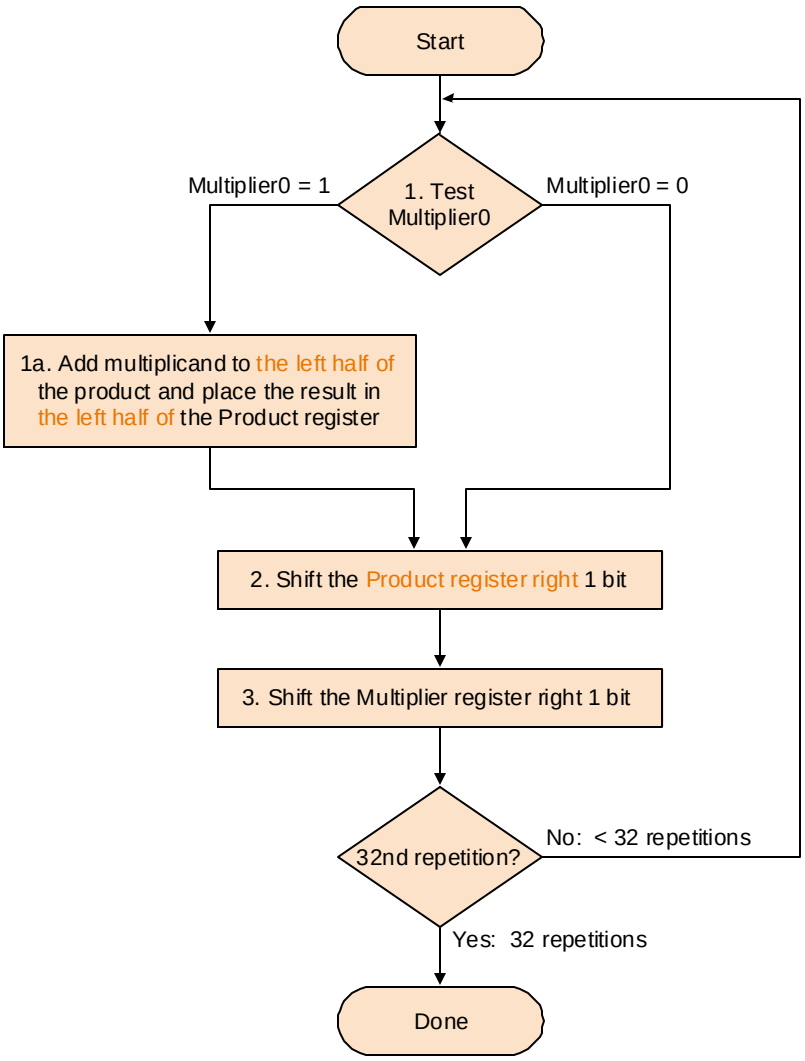
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	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

FIGURE 4.27 Multiply example using first algorithm in Figure 4.26. The bit examined to determine the next step is circled in color.

- **Hardware Multiplicador – Segunda versão**



- **Algoritmo de Multiplicação – Segunda versão**



Exemplo

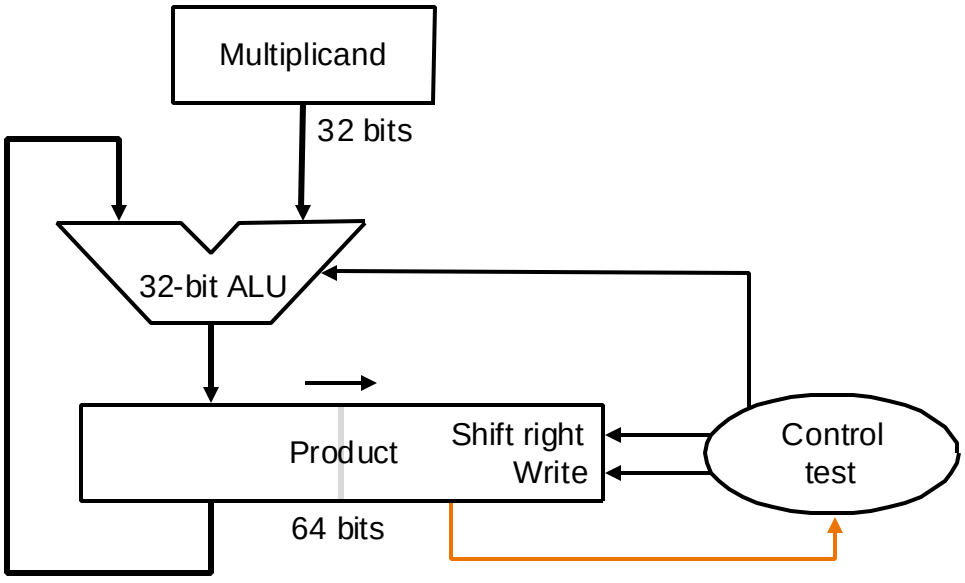
$$0010_2 \times 0011_2 = 0000\ 0110_2$$

- **Figura 4.30 – Resolução do exemplo**

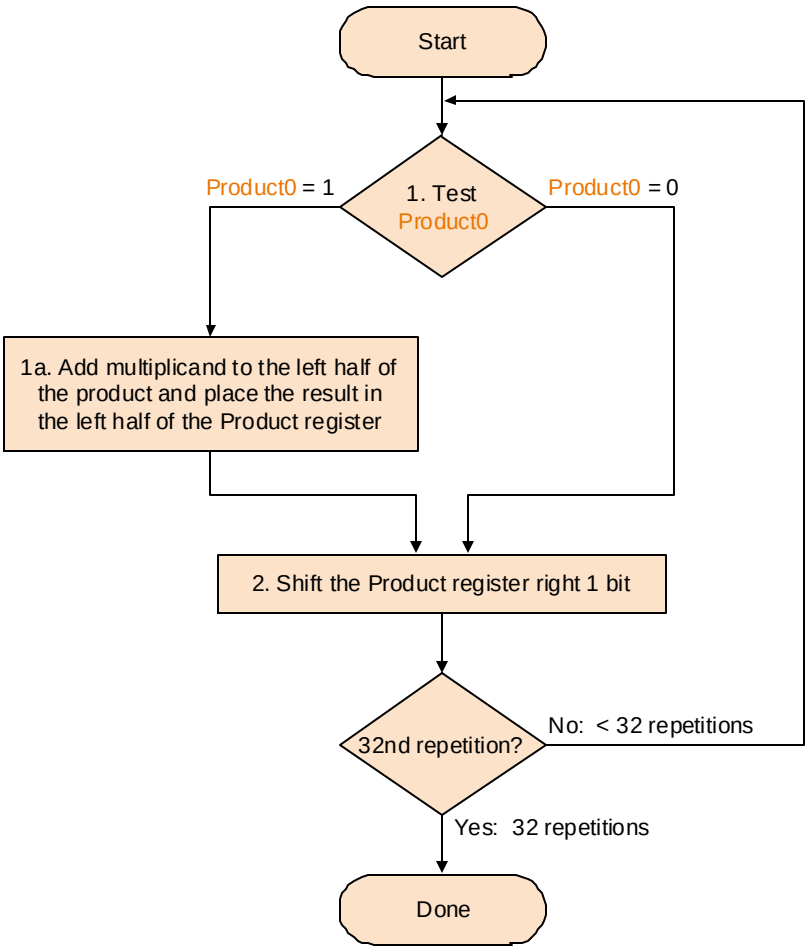
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0010	0000 0000
1	1a: 1 => Prod = Prod + Mcand	0011	0010	0010 0000
	2: Shift right Product	0011	0010	0001 0000
	3: Shift right Multiplier	0001	0010	0001 0000
2	1a: 1 => Prod = Prod + Mcand	0001	0010	0011 0000
	2: Shift right Product	0001	0010	0001 1000
	3: Shift right Multiplier	0000	0010	0001 1000
3	1: 0 => no operation	0000	0010	0001 1000
	2: Shift right Product	0000	0010	0000 1100
	3: Shift right Multiplier	0000	0010	0000 1100
4	1: 0 => no operation	0000	0010	0000 1100
	2: Shift right Product	0000	0010	0000 0110
	3: Shift right Multiplier	0000	0010	0000 0110

FIGURE 4.30 Multiply example using second algorithm in Figure 4.29. The bit examined to determine the next step is circled in color.

- **Hardware Multiplicador – Terceira Versão**



- **Algoritmo de Multiplicação – Terceira Versão**



Exemplo

$$0010_2 \times 0011_2 = 0000\ 0110_2$$

- **Figura 4.33 – Resolução do exemplo**

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	1a: 1 => Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 => Prod = Prod + Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: 0 => no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: 0 => no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

FIGURE 4.33 Multiply example using third algorithm in Figure 4.32. The bit examined to determine the next step is circled in color.

- **Multiplicação sinalizada**

  Booth's algoritm

0010_2 → 2_{10}
x 0110_2 → 6_{10}

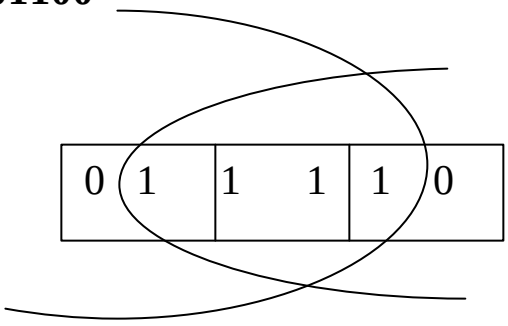
+ 0000 shift
+ 0010 add
+ 0010 add
+ 0000 shift

00001100

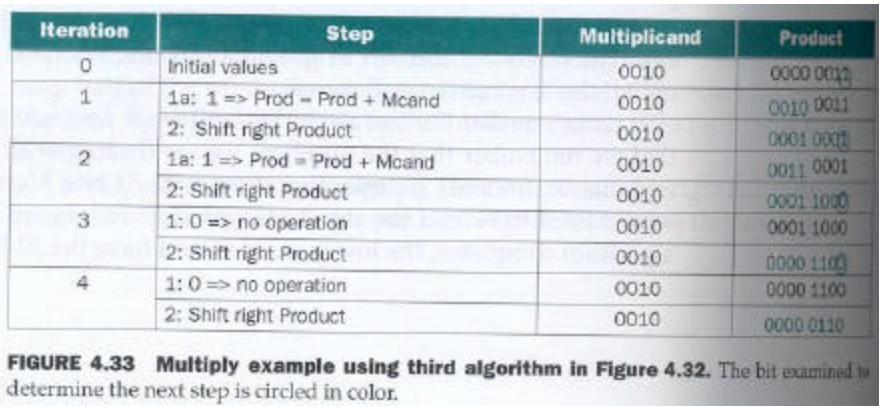
0010_2 → 2_{10}
x 0110_2 → 6_{10}

+ 0000 shift
- 0010 sub (primeiro 1 no multiplicador)
+ 0000 shift (meio da cadeia de 1s)
+ 0010 add (passo anterior – último 1)

00001100



- **Figura 4.33 – Exemplo algoritmo Booth**



Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0010
1	1a: 1 => Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 => Prod = Prod + Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: 0 => no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: 0 => no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

FIGURE 4.33 Multiply example using third algorithm in Figure 4.32. The bit examined to determine the next step is circled in color.

- **Passos do algoritmo de Booth**
  - **Dependendo do bit corrente e anterior, faça:**
    - **00 : meio de uma cadeia de 0s ➔ nenhuma operação**
    - **01 : fim de uma cadeia de 1s ➔ some o multiplicando à metade esquerda do produto**
    - **10 : começo de uma cadeia de 1s ➔ subtraia o multiplicando da metade esquerda do produto**
    - **11 : meio de uma cadeia de 1s ➔ nenhuma operação**
  - **Como no terceiro algoritmo de multiplicação, desloque o produto 1 bit para a direita.**

- **Figura 4.34 – Comparação do Algorimto de booth e da terceira versão do Algoritmo de Multiplicação, para números positivos**

Iteration	Multiplicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 0 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	1a: 1 ⇒ Prod = Prod + Mcand	0010 0011	1c: 10 ⇒ Prod = Prod – Mcand	1110 0011 0
	0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1 ⇒ Prod = Prod + Mcand	0011 0001	1d: 11 ⇒ no operation	1111 0001 1
	0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1: 0 ⇒ no operation	0001 1000	1b: 01 ⇒ Prod = Prod + Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

FIGURE 4.34 Comparing algorithm in Figure 4.32 and Booth's algorithm for positive numbers. The bit(s) examined to determine the next step is circled in color.

Exemplo

2_{10} x -3_{10} = -6_{10} ➔ 0010_2 x 1101_2 = $1111\ 1010_2$

- **Figura 4.35 – Resolução do exemplo**

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 1101 0
1	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b: 01 ⇒ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c: 10 ⇒ Prod = Prod – Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d: 11 ⇒ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

FIGURE 4.35 Booth's algorithm with negative multiplier example. The bits examined to determine the next step are circled in color.

- **Multiplicação por potência de 2, por deslocamento**

- **Multiplicação no MIPS**
  - **produto de números de 32 bits ➔ 64 bits ➔ *Hi* e *Lo***
  - **instruções *mult* (multiplicação) e *multu* (multiplicação sem sinal)**
  - **instruções para manipular *Hi* e *Lo* ➔ *mflo* e *mfhi* (move from *Lo* e move from *Hi*)**

- **Divisão**

dividendo ←1001010 | 1000 ➔ divisor

- 1000 1001 ➔ quociente

10

101

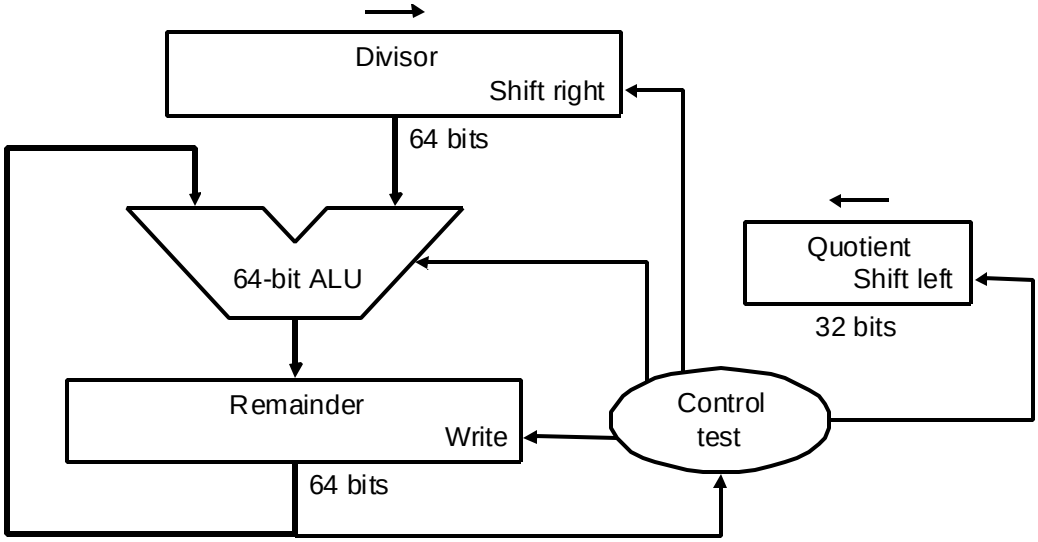
1010

- 1000

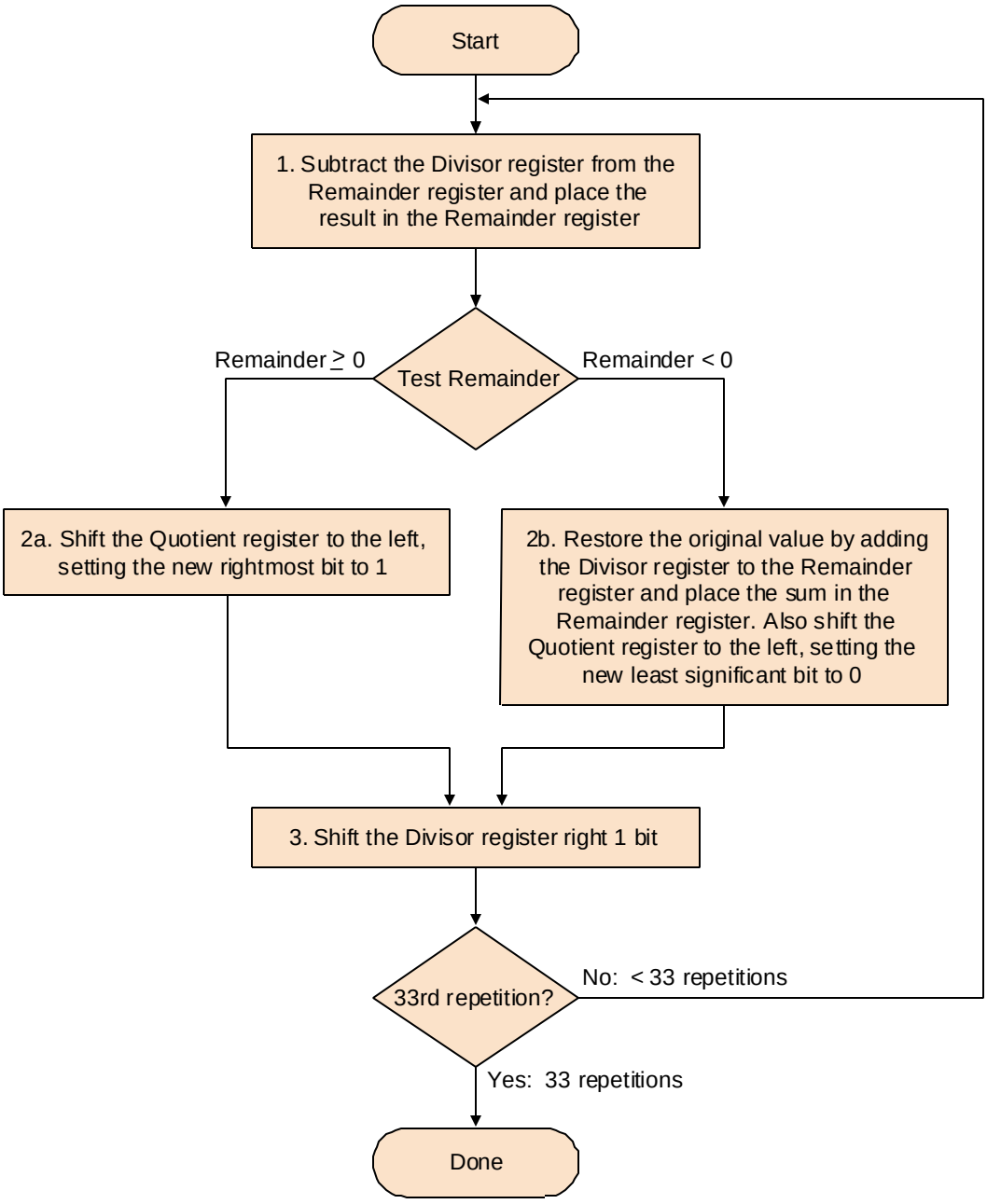
---

10 ➔ resto

- **Hardware Divisor - Primeira Versão**



- **Algoritmo de Divisão – Primeira Versão**



Exemplo

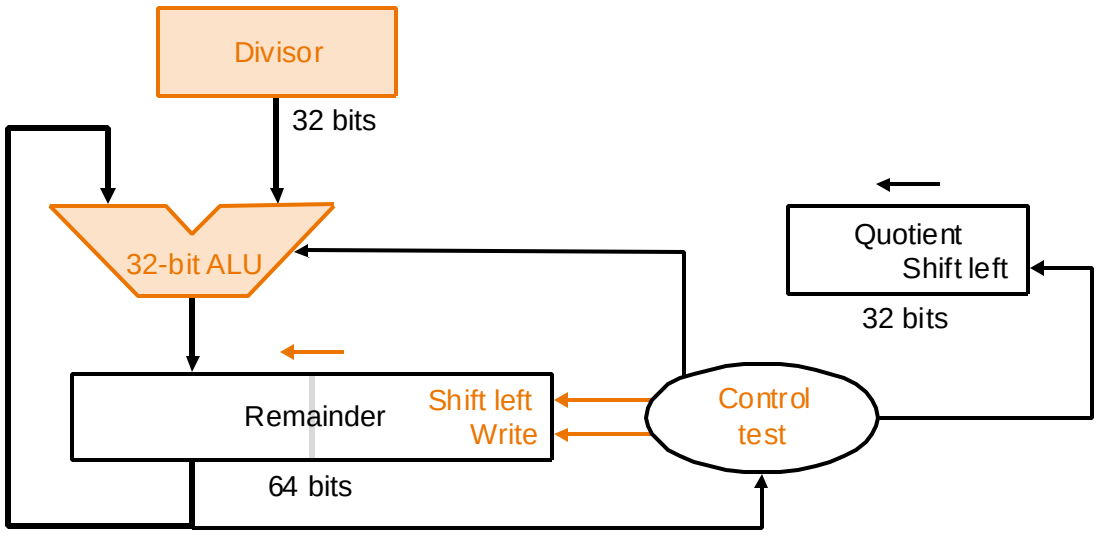
$0000\ 00111_2$ / 0010_2

- **Figura 4.38 – Resolução do exemplo**

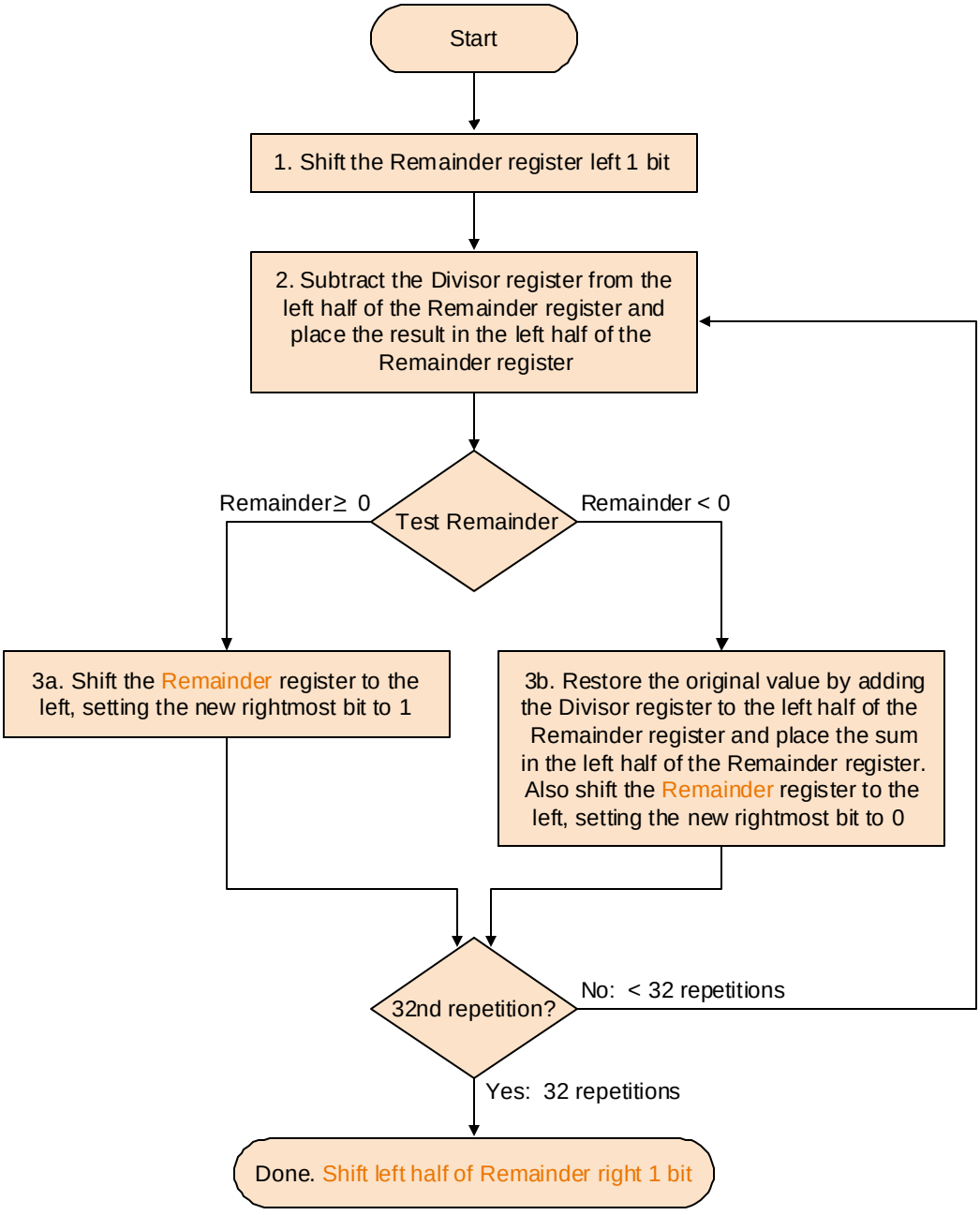
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, 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 ⇒ +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	0000 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	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 4.38 Division example using first algorithm in Figure 4.37. The bit examined to determine the next step is circled in color.

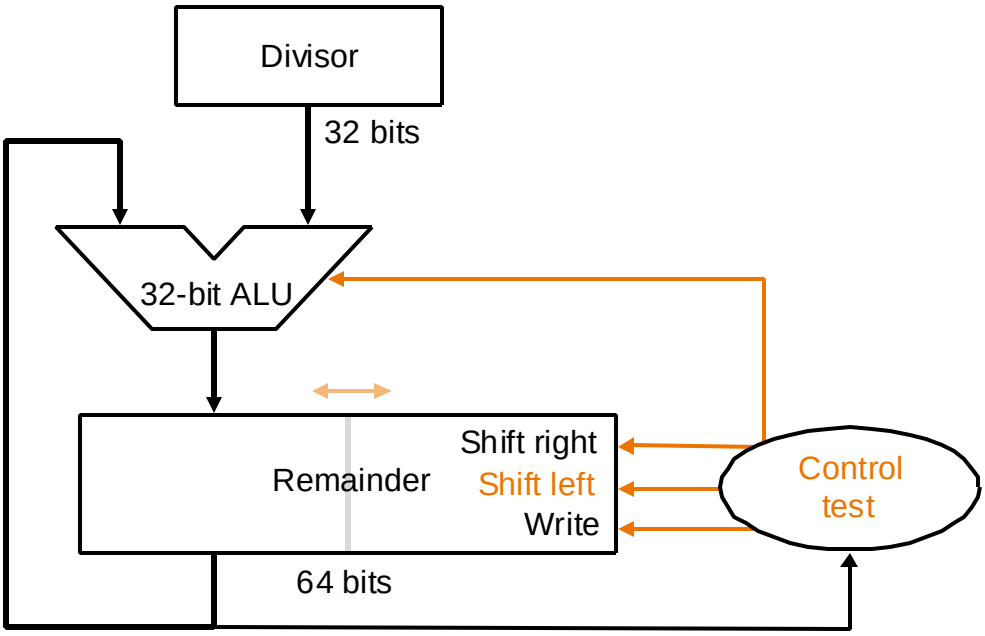
- **Hardware Divisor - Segunda Versão**



- **Versão final do algoritmo de divisão**



- **Hardware Divisor - Versão Final**



Exemplo

$0000\ 0111_2$ / 0010_2

- **Figura 4.42 – Resolução de Exemplo**

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem – Div	0010	1110 1110
	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem – Div	0010	1111 1100
	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

FIGURE 4.42 Division example using third algorithm in Figure 4.40. The bit examined to determine the next step is circled in color.

- **Divisão no MIPS**
  - **divisão de números de 64 bits por 32 bits ➔ *Hi* e *Lo***
  - **instruções *div* (divisão) e *divu* (divisão sem sinal)**
  - **instruções para manipular *Hi* e *Lo* ➔ *mflo* e *mfhi* (move from *Lo* e move from *Hi*)**

- **Figura 4.43 – Arquitetura do MIPS – Resumo**

MIPS operands

Name	Example	Comments
32 registers	$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at, Hi, Lo	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. Hi and Lo contain the results of multiply and divide.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

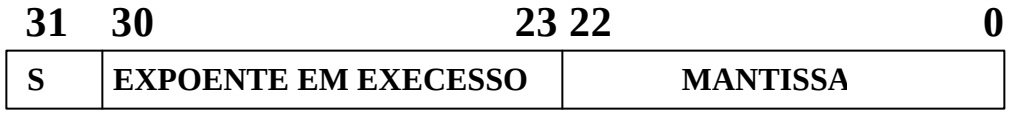
Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$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	Used to copy Exception PC plus other special registers
	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
Logical	and	and	$s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; logical AND
	or	or	$s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; logical OR
	and immediate	andi	$s1,$s2,100	$s1 = $s2 & 100	Logical AND reg, constant
	or immediate	ori	$s1,$s2,100	$s1 = $s2 | 100	Logical OR reg, 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
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 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
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; natural numbers
	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Unconditional 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

FIGURE 4.43 MIPS architecture revealed thus far. Color indicates the portions revealed since Figure 4.7 on page 228. MIPS machine language is listed on the back endpapers of this book. (page 274)

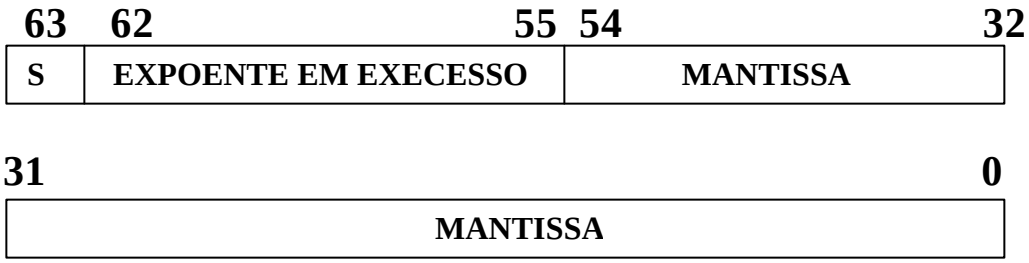
- **Representação em ponto flutuante**
  - **Padrão IEEE 754 – normalizado, expoente em excesso 127**

$$N = (-1)^S \times 1.M \times 2^E$$

    - **precisão simples**



    - **precisão dupla**



Exemplo

$-0{,}75_{10} = -0{,}11_2$

Normalizando ➔ $1{,}1 \times 2^{-1}$

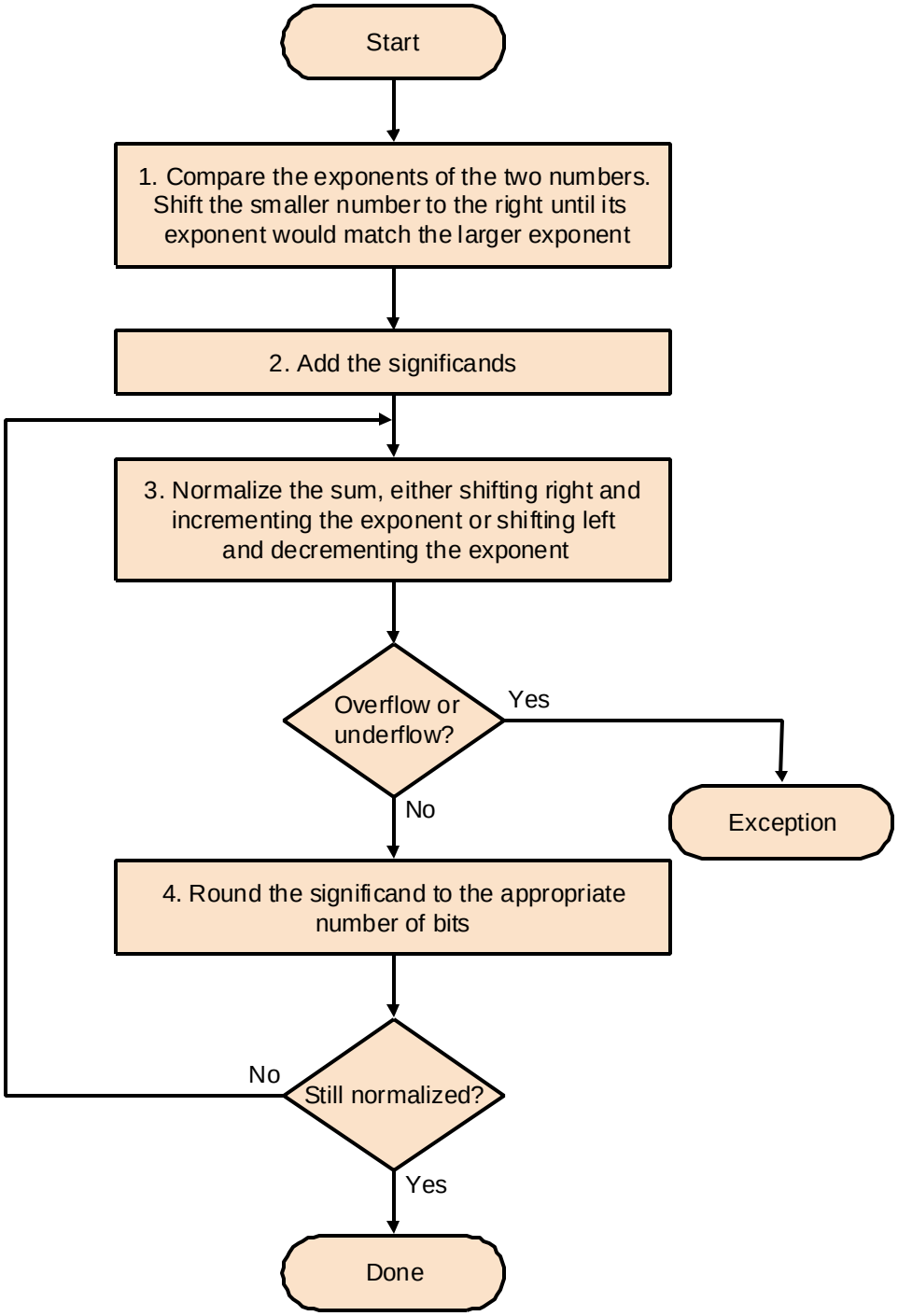
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Exemplo: Qual o decimal correspondente ?

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	0	0	0

$N = -(1+0.25) \times 2^{(129-127)} = -1{,}25 \times 4 = -5{,}0$

- **Adição em ponto flutuante**



Exemplo: $0{,}5_{10} + (-0{,}4375_{10})$

$0{,}5_{10} = 0{,}1_2 = 1{,}000_2 \times 2^{-1}$

$-0{,}4375_{10} = -0{,}0111_2 = -1{,}110_2 \times 2^{-2}$

Etapas

1. **Deslocar à direita a mantissa do número com menor expoente**

   $-1{,}110_2 \times 2^{-2} = -0{,}111_2 \times 2^{-1}$

2. **Somar as mantissas**

   $1{,}000_2 \times 2^{-1} + (-0{,}111_2 \times 2^{-1}) = 0{,}001_2 \times 2^{-1}$

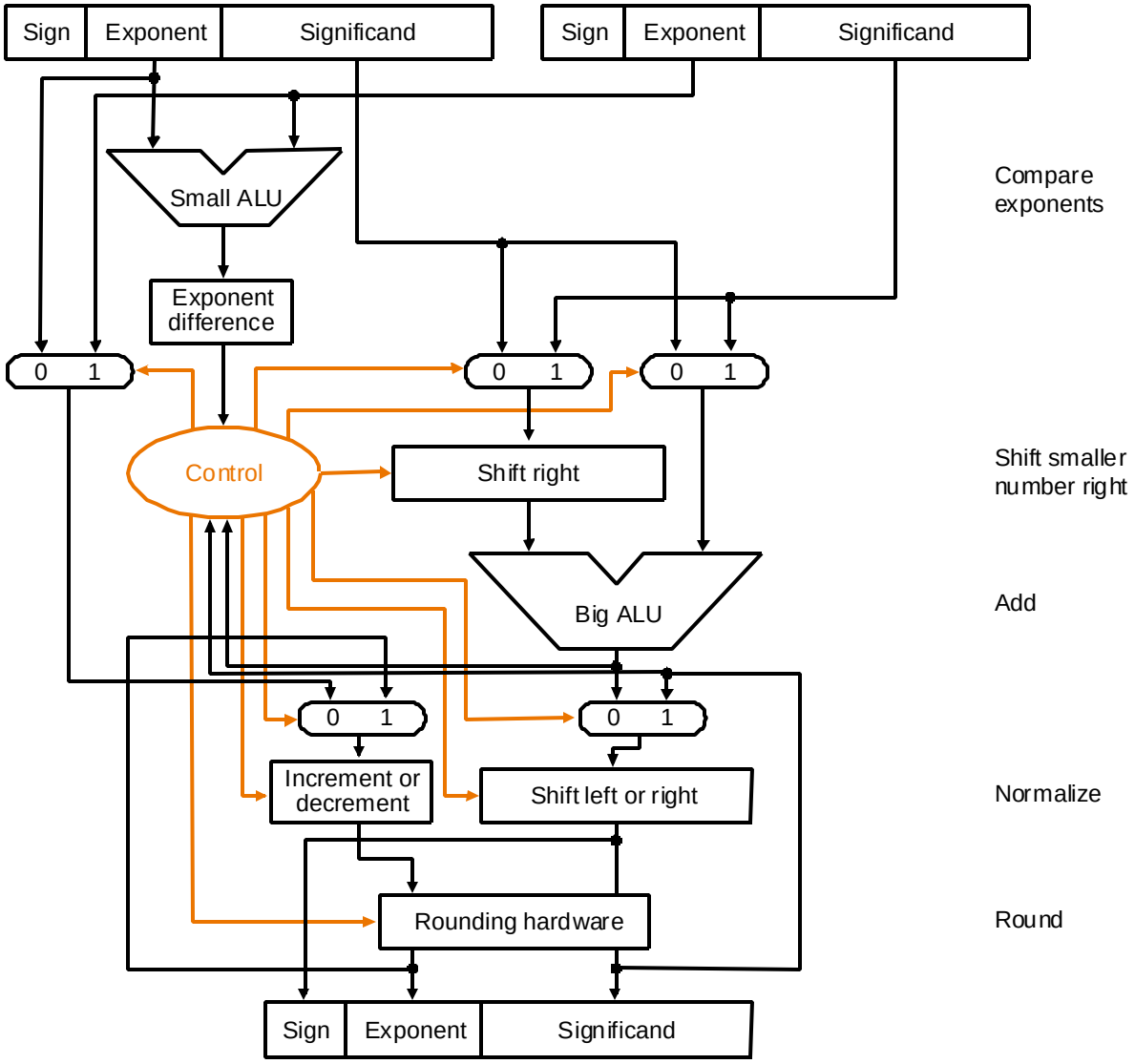
3. **Normalizar o resultado**

   $0{,}001_2 \times 2^{-1} = 1{,}000_2 \times 2^{-4}$

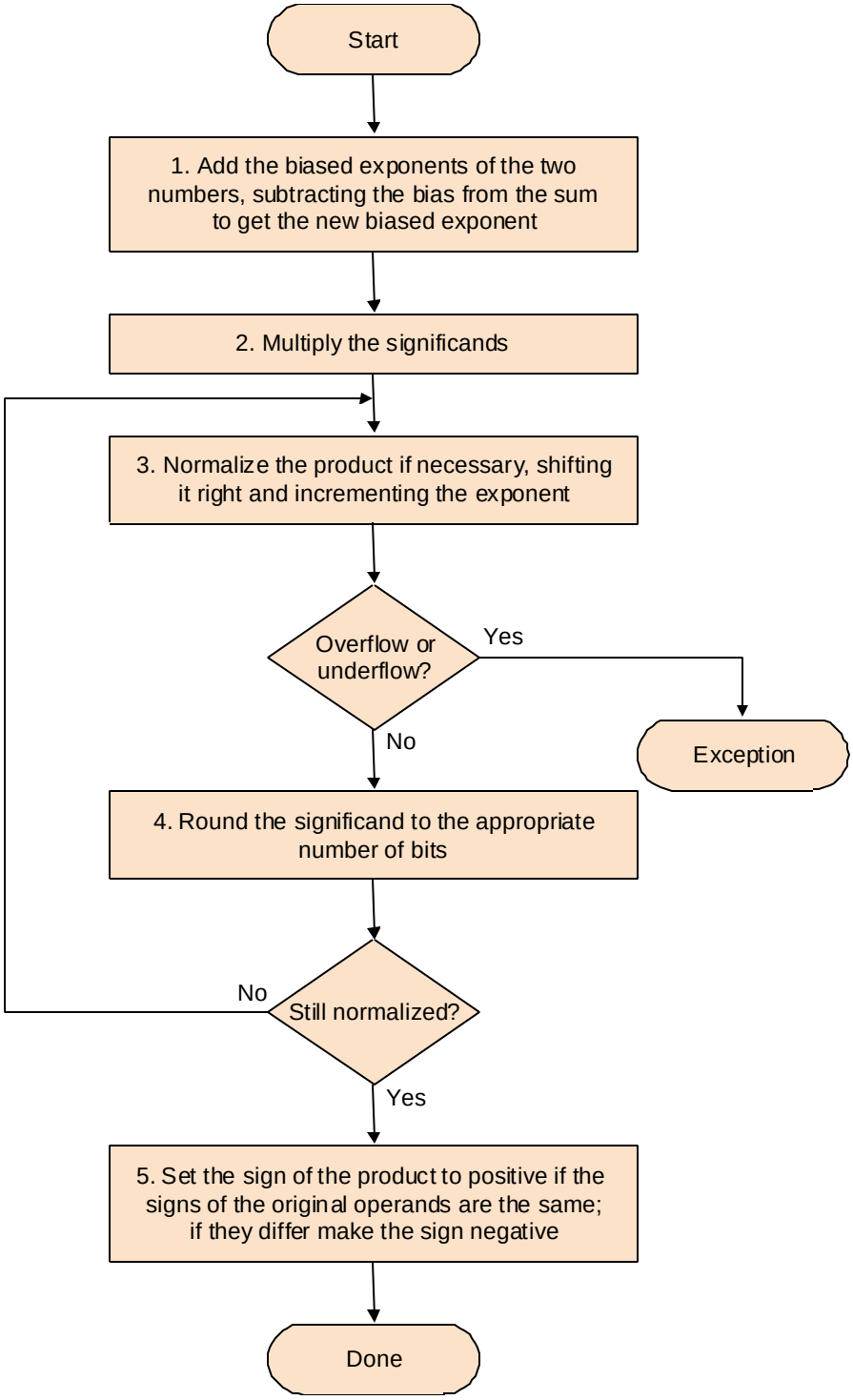
4. **Arredondar o resultado**

   $1{,}000_2 \times 2^{-4} \rightarrow 0{,}0625_{10}$

- **Figura 4.45 – Hardware de adição de ponto flutuante**



- **Multiplicação em ponto flutuante**



Exemplo : $0,5_{10} \times (-0,4375_{10})$

$0,5_{10} = 0,1_2 = 1,000_2 \times 2^{-1}$
$-0,4375_{10} = -0,0111_2 = -1,110_2 \times 2^{-2}$

Etapas

1. **Somar os expoentes**

   $-1 + (-2) = -3$

2. **Multiplicar as mantissas**

   $-1,110_2 \times 1,000_2 = -1,110_2$

3. **O resultado final**

   $-1,110_2 \times 2^{-3}$ ➔ **-0,21875**

- **Instruções em ponto flutuante no MIPS**

  - **adição, simples (*add.s*) – adição, double (*add.d*)**
  - **subtração, simples (*sub.s*) – subtração, double (*sub.d*)**
  - **multiplicação, simples (*mul.s*) – multiplicação, double (*mul.d*)**
  - **divisão, simples (*div.s*) – divisão, double (*div.d*)**
  - **comparação, simples (*c.x.s*) – comparação, double (*c.x.d*), onde x pode ser equal (*eq*), not equal (*neq*), less than (*lt*), less than or equal (*le*), greater than (*gt*) ou greater than or equal (*ge*)**
  - **branch, true (*bclt*) – branch, false (*bclf*)**
  - **registradores $f0,$f1,$f2,… (32 registradores)**

- **Figura 4.47 – Arquitetura MIPS - Resumo**

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 words 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

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

FIGURE 4.47 MIPS floating-point architecture revealed thus far. See Appendix A, section A.10, on page A-49, for more detail