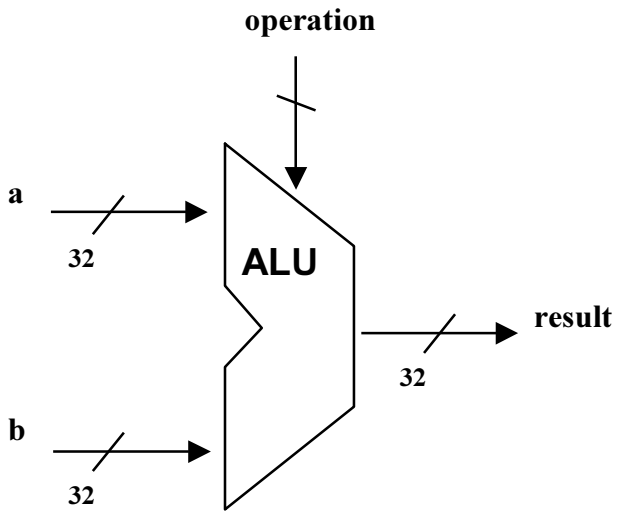
Chapter Four

Arithmetic for Computers

Arithmetic

- **Where we've been:**
  - **Performance (seconds, cycles, instructions)**
  - **Abstractions:**
    Instruction Set Architecture
    Assembly Language and Machine Language
- **What's up ahead:**
  - **Implementing the Architecture**



Numbers

- **Bits are just bits (no inherent meaning)**
  — conventions define relationship between bits and numbers
- **Binary numbers (base 2)**
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0...2^n-1$
- **Of course it gets more complicated:**
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number)
- **How do we represent negative numbers?**
  i.e., which bit patterns will represent which numbers?

Possible Representations

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- **Issues: balance, number of zeros, ease of operations**
- **Which one is best? Why?**

MIPS

- **32 bit signed numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Two's Complement Operations

- **Negating a two's complement number: invert all bits and add 1**
  - **remember: “negate” and “invert” are quite different!**
- **Converting n bit numbers into numbers with more than n bits:**
  - **MIPS 16 bit immediate gets converted to 32 bits for arithmetic**
  - **copy the most significant bit (the sign bit) into the other bits**

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - **"sign extension" (lbu vs. lb)**

Novas instruções

- **instruções “unsigned”: (exemplo de aplicação, cálculo de memória)**
- **sltu $t1, $t2, $t3 # diferença é “sem sinal”**
- **slti e sltiu # envolve imediato, com ou sem sinal**

- **Exemplo pag 215: supor $s0 = FF FF FF FF e $s1 = 00 00 00 01**

slt $t0, $s0, $s1
como $s0 < 0 e $s1 > 0 ⇒ $s0<$s1 ⇒ $t0 = 1

sltu $t0, $s0, $s1
como $s0 e $s1 não tem sinal ⇒ $s0>$s1 ⇒ $t0 = 0

Cuidados com extensão 16 bits

- **beq $s0, $s1, nnn # salta para PC + nnn se teste OK**
- **nnn tem 16 bits e PC tem 32 bits**
  - **estender de 16 para 32 bits antes daoperação aritmética**
- **se nnn > 0**
  - **preencher com zeros à esquerda**
- **se nnn < 0 CUIDADO**
  - **preencher com 1´s à esquerda**
  - **verificar**
- **por este motivo operação é chamada de**
  - **EXTENSÃO DE SINAL**

Addition & Subtraction

- **Just like in grade school (carry/borrow 1s)**

```
   0111          0111          0110
 + 0110        - 0110        - 0101
```

- **Two's complement operations easy**
  - **subtraction using addition of negative numbers**

```
   0111
 + 1010
```

- **Overflow (result too large for finite computer word):**
  - **e.g., adding two n-bit numbers does not yield an n-bit number**

```
   0111
 + 0001
   1000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- CONDIÇÕES DE OVERFLOW

op	A	B	resultado
A+B	+	+	-
A+B	-	-	+
A-B	+	-	-
A-B	-	+	+

Em hardware, comparar o “vai-um” e o “vem-um” com relação ao bit de sinal

Effects of Overflow

- **An exception (interrupt) occurs**
  - **Control jumps to predefined address for exception (EPC — EXCEPTION PROGRAM COUNTER)**
  - **Interrupted address is saved for possible resumption**
    - mfc0 (move from system control): copia endereço do EPC para qualquer registrador
- **Don't always want to detect overflow**
  — new MIPS instructions: `addu, addiu, subu`

  note: **`addiu`** ***still sign-extends!***
  note: **`sltu, sltiu`** ***for unsigned comparisons***

Instruções (fig 4.52 - pag 309)

add	add	R
add immediate	addi	I
add unsigned	addu	R
add immediate unsigned	addiu	I
subtract	sub	R
subtract unsigned	subu	R
and	and	R
and immediate	andi	I
or	or	R
or immediate	ori	I
shift left logical	sll	R
shift right logical	srl	R
load upper immediate	lui	I
load word	lw	I
store word	sw	I
load byte unsigned	lbu	I
store byte	sb	I
branch on equal	beq	I
branch on not equal	bne	I
jump	j	J
jump and link	jal	J
jump register	jr	R
set less than	slt	R
set less than immediate	slti	I
set less than unsigned	sltu	R
set less than immediate unsigned	sltiu	I

multiply	mult	R
multiply unsigned	multu	R
divide	div	R
divide unsigned	divu	R
move from Hi	mfhi	R
move from Lo	mflo	R
move from system control (EPC)	mfc0	R
fp add single	add.s	R
fp add doublr	add.d	R
fp subtract single	sub.s	R
fp subtract double	sub.d	R
fp multiply single	mul.s	R
fp multiply double	mul.d	R
fp divide single	div.s	R
fp divide double	div.d	R
load word to fp single	lwc1	I
store word to fp single	swc1	I
branch on fp true	bclt	I
branch on fp false	bclf	I
fp compare single (x= eq,neq,lt,le,gt,ge)	c.x.s	R
fp compare double (x= eq,neq,lt,le,gt,ge)	c.x.d	R

Review: Boolean Algebra & Gates

- **Problem: Consider a logic function with three inputs: A, B, and C.**

  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true

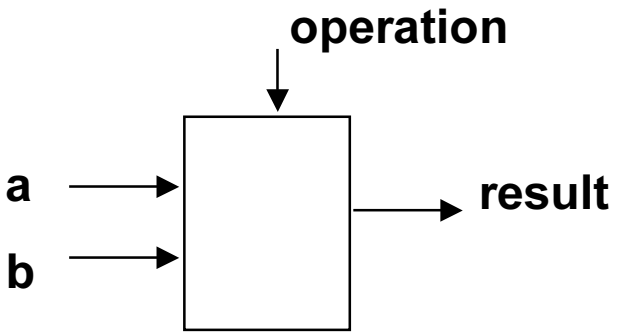
- **Show the truth table for these three functions.**

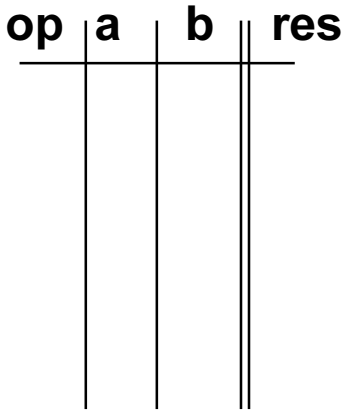
- **Show the Boolean equations for these three functions.**

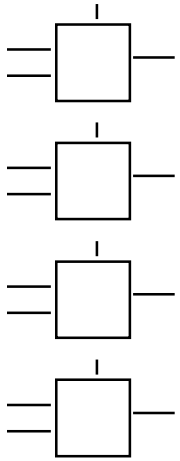
- **Show an implementation consisting of inverters, AND, and OR gates.**

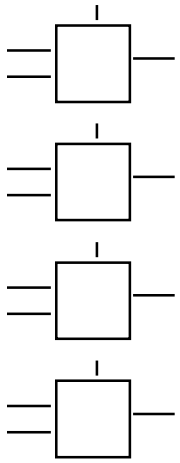
An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them





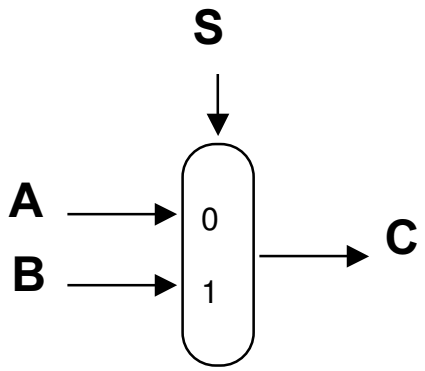




- Possible Implementation (sum-of-products):

Review: The Multiplexor

- **Selects one of the inputs to be the output, based on a control input**

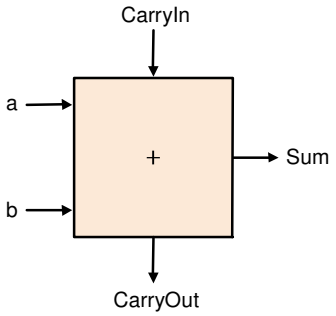


note: we call this a 2-input mux even though it has 3 inputs!

- **Lets build our ALU using a MUX:**

Different Implementations

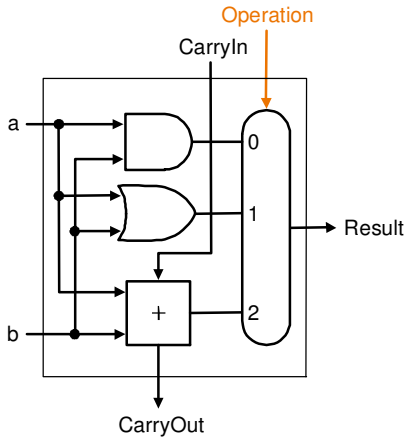
- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Dont want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

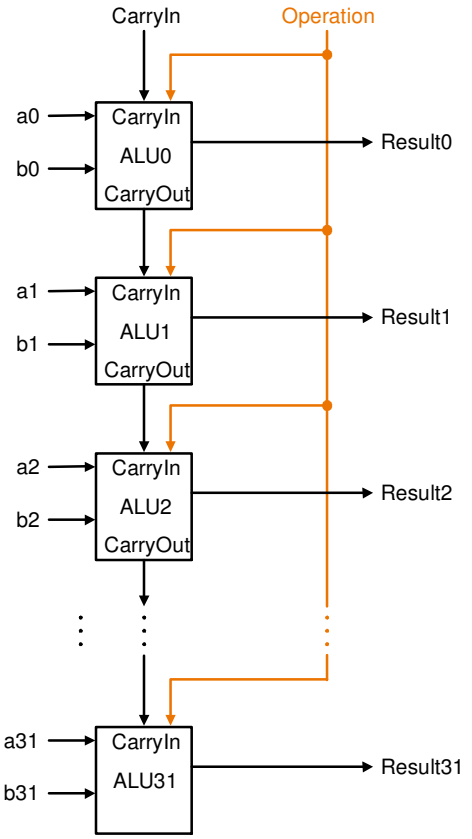


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU





What about subtraction (a – b) ?

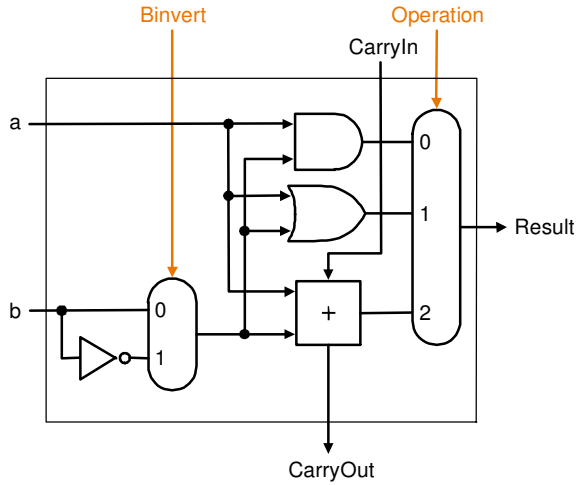
- **Two's complement approch: just negate b and add.**

  a - b = a + (- b)

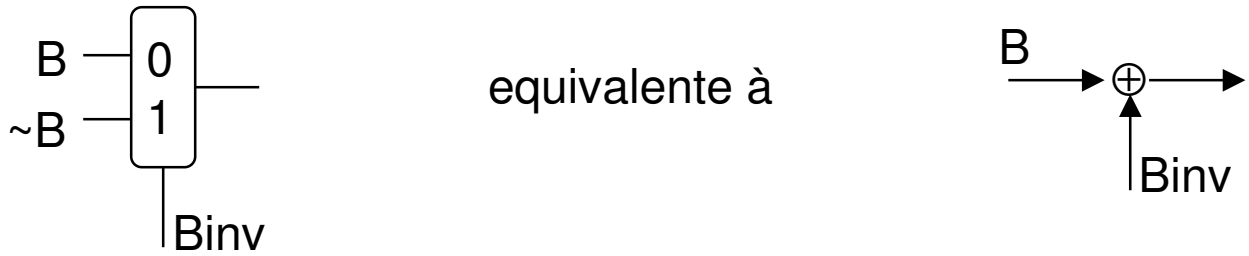
- **How do we negate?**

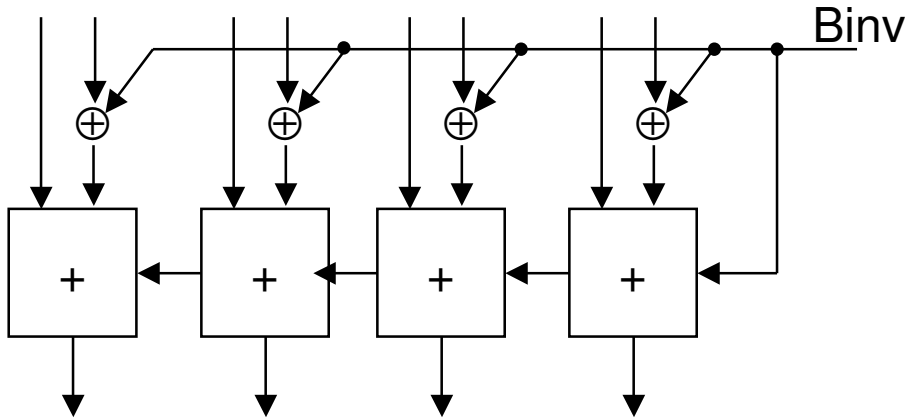
  (- a) = $comp_2(a) = comp_1(a) + 1$

- **A very clever solution:**



Subtrator



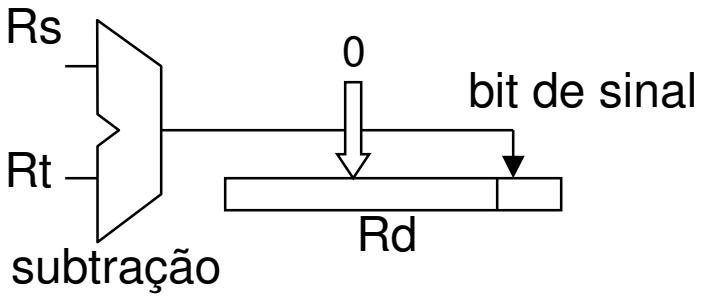


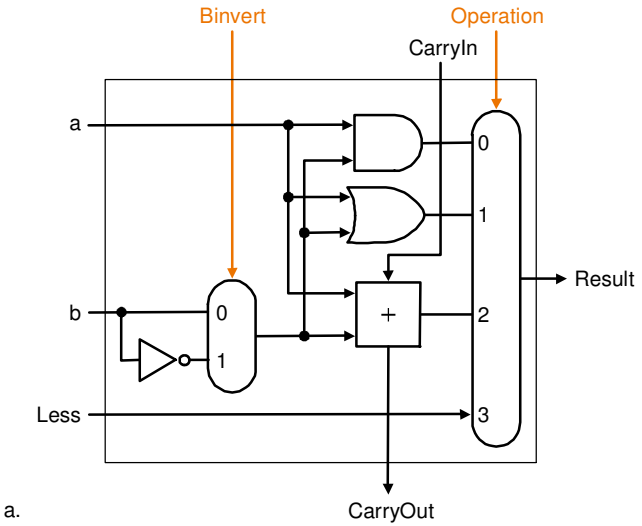
Tailoring the ALU to the MIPS

- **Need to support the set-on-less-than instruction (slt)**
  - **remember: slt is an arithmetic instruction**
  - **produces a 1 if rs < rt and 0 otherwise**
  - **use subtraction: (a-b) < 0 implies a < b**
- **Need to support test for equality (beq $t5, $t6, $t7)**
  - **use subtraction: (a-b) = 0 implies a = b**

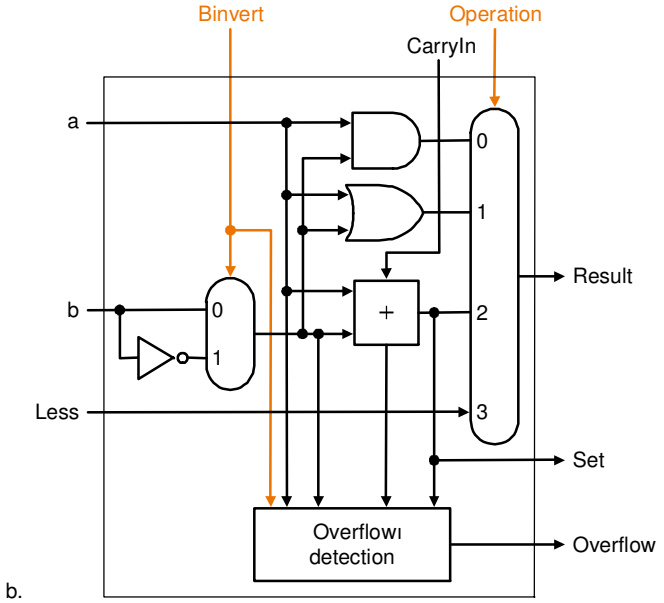
Supporting slt

- **Can we figure out the idea?**

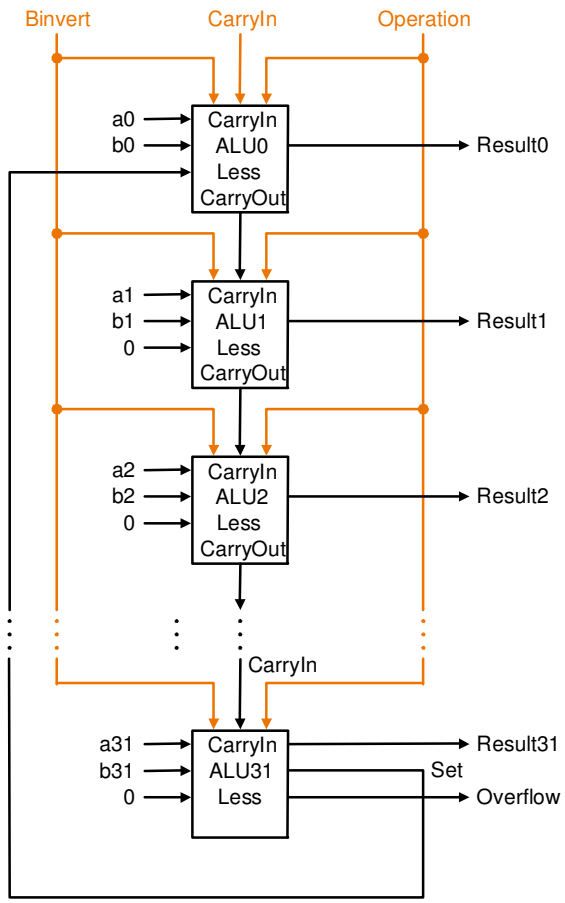




a.



b.

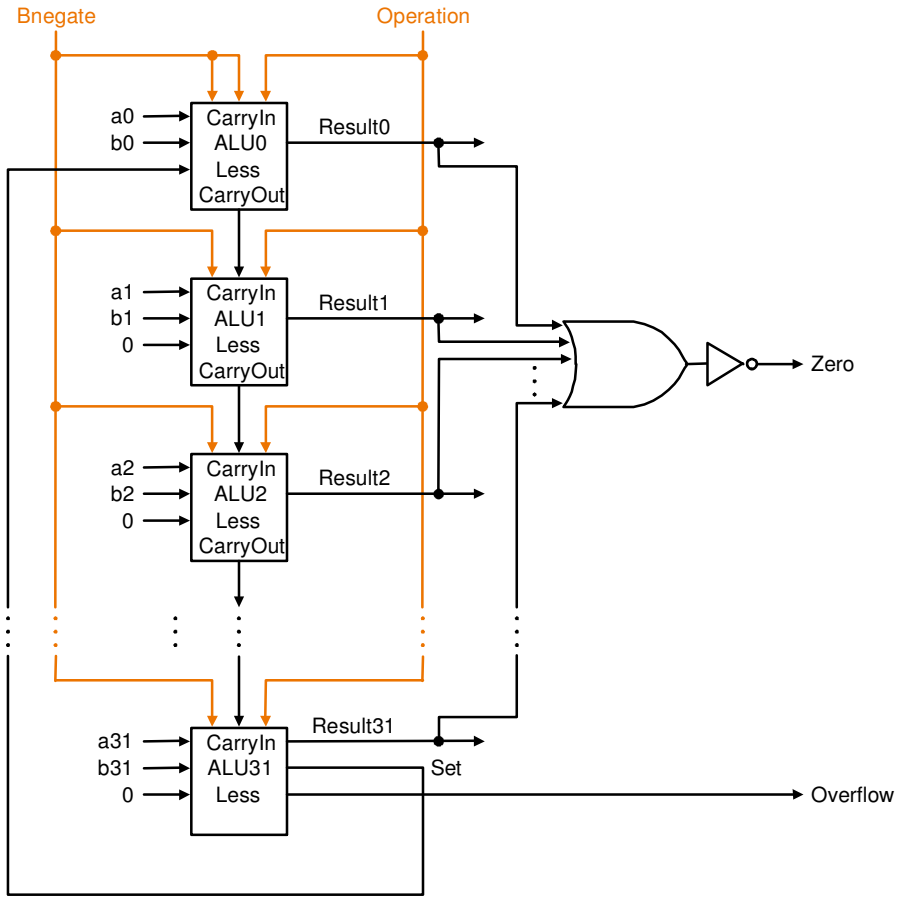


Test for equality

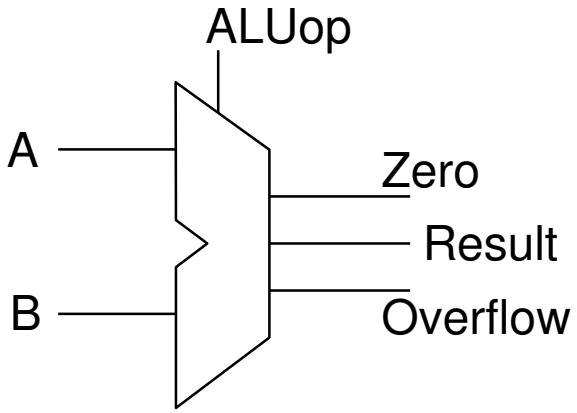
- **Notice control lines:**

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

•Note: zero is a 1 when the result is zero!



ALU



32 bits: A, B, result

1 bit: Zero, Overflow

3 bits: ALUop

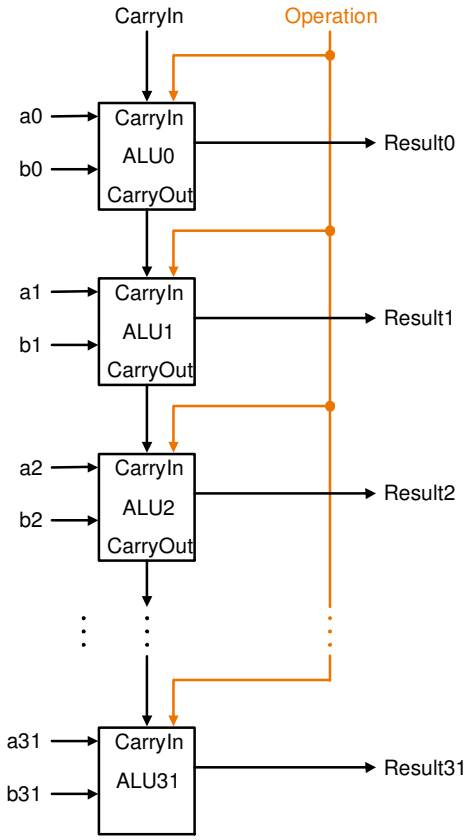
ALUop Binv-OP	Instrução
0 00	and
0 01	or
0 10	add
1 10	sub
1 11	slt
1 10	beq

Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
  atraso (ent ⇒ soma ou carry = 2G)
  n estágios ⇒ 2nG
- Is there more than one way to do addition?
  - two extremes:
    ripple carry (2nG)
    sum-of-products (2G)



Can you see the ripple? How could you get rid of it?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$
$$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \qquad c_2 =$$
$$c_3 = b_2c_2 + a_2c_2 + a_2b_2 \qquad c_3 =$$
$$c_4 = b_3c_3 + a_3c_3 + a_3b_3 \qquad c_4 =$$

Not feasible! Why?

Carry-lookahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple?

$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 c_1$ $\qquad c_2 =$

$c_3 = g_2 + p_2 c_2$ $\qquad c_3 =$

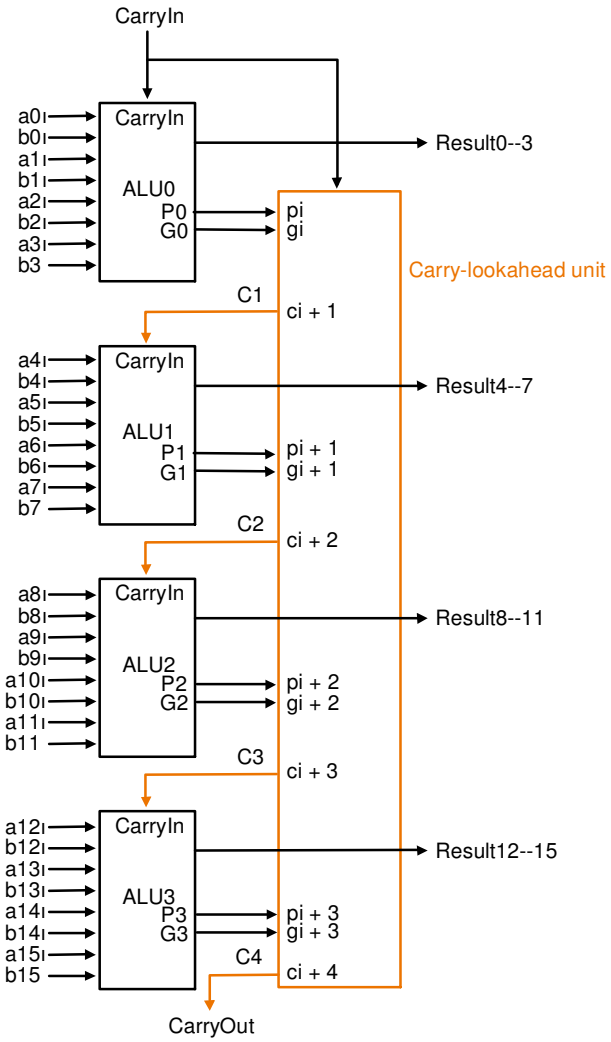
$c_4 = g_3 + p_3 c_3$ $\qquad c_4 =$

Feasible! Why?

- atraso: ent $\Rightarrow$ g_i p_i (1G)
  g_i p_i $\Rightarrow$ carry (2G)
  carry $\Rightarrow$ saídas (2G)

total: 5G independente de n

Use principle to build bigger adders



- **Can't build a 16 bit adder this way... (too big)**
- **Could use ripple carry of 4-bit CLA adders**
- **Better: use the CLA principle again!**
  - **super propagate (ver pag 243)**
  - **super generate (ver pag 245)**
  - **ver exercícios 4.44, 45 e 46 (não será cobrado)**

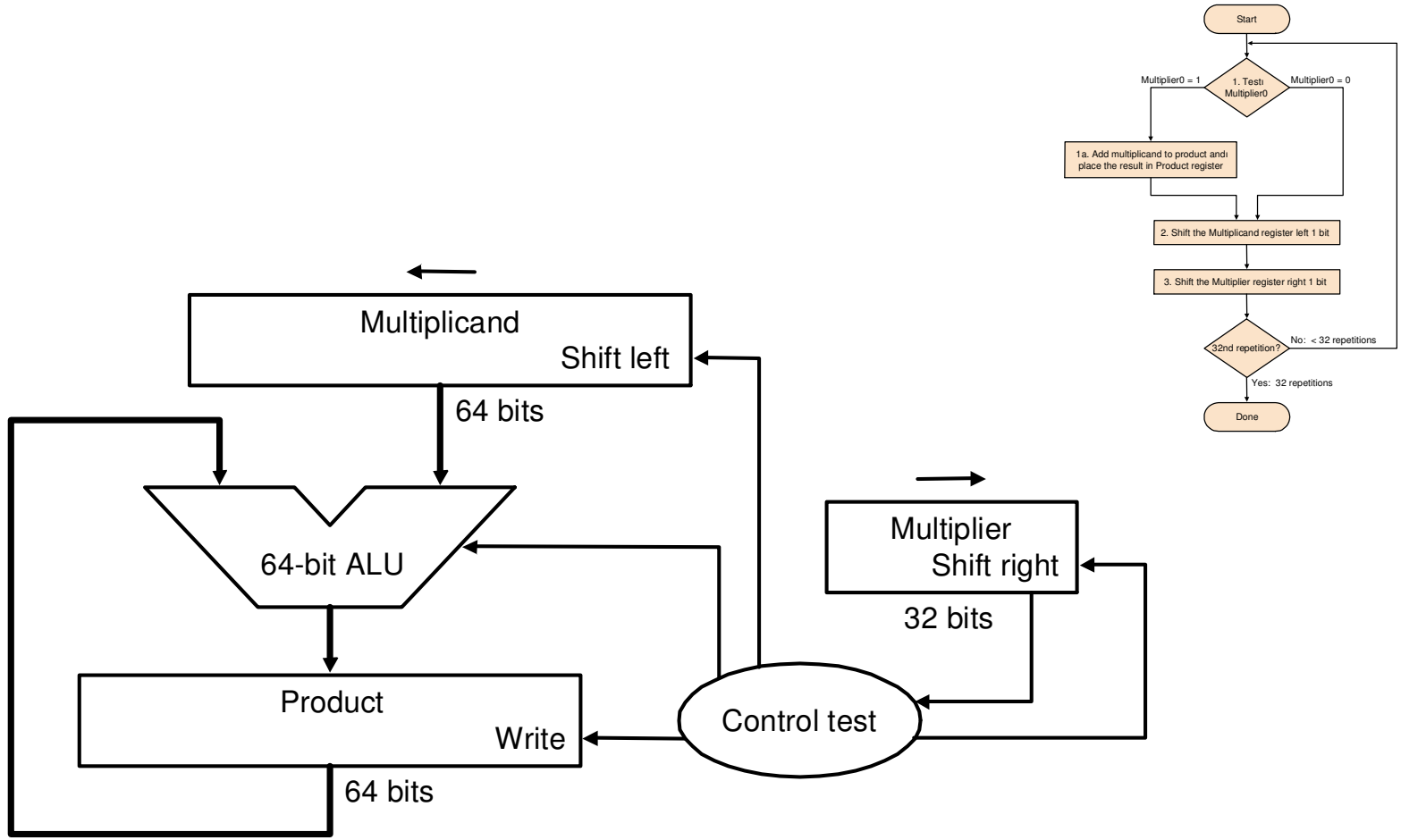
Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on gradeschool algorithm

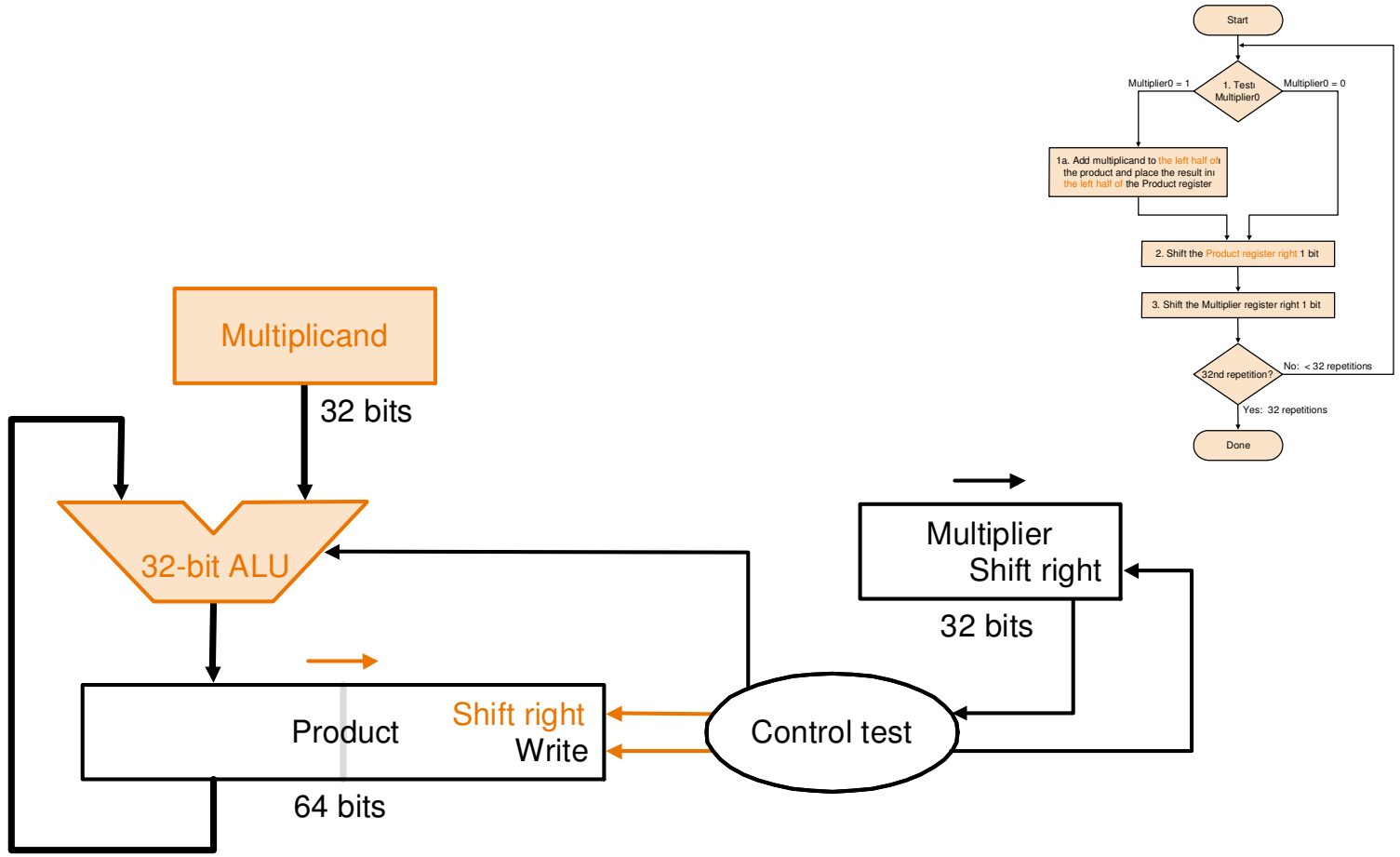
8_{10}	1000	multiplicando
9_{10}	1001	multiplicador
	1000	
	0000	produtos parciais
	0000	
	1000	
72_{10}	1001000	max= $(2^4-1)*(2^4-1) = 225$
		225 > 128 $\Rightarrow$ 8 bits
		32 * 32 bits $\Rightarrow$ 64 bits

- Negative numbers: convert and multiply
  - there are better techniques, we won't look at them

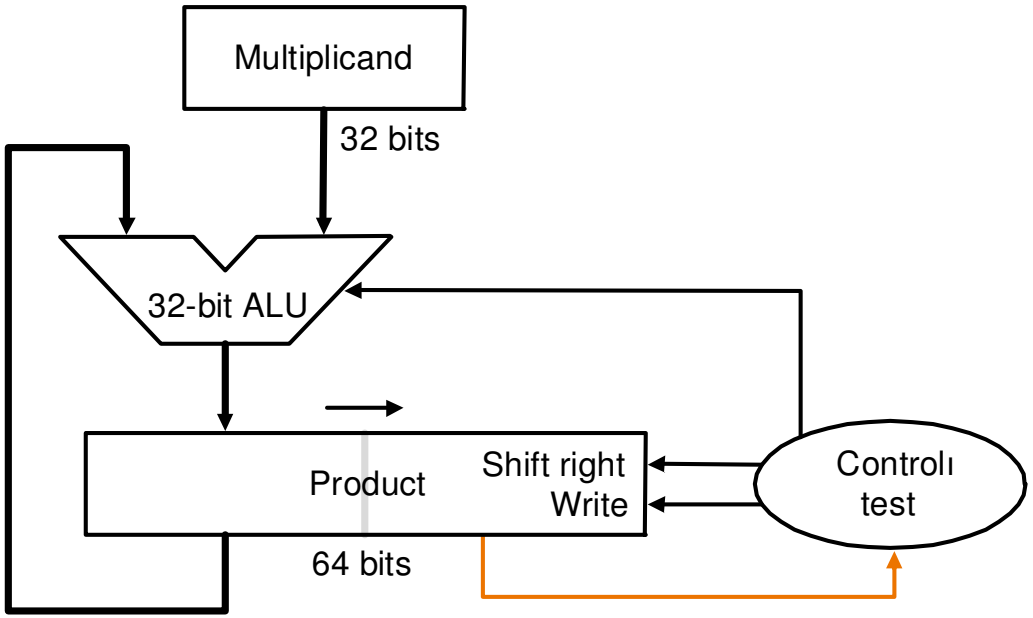
Multiplication: Implementation

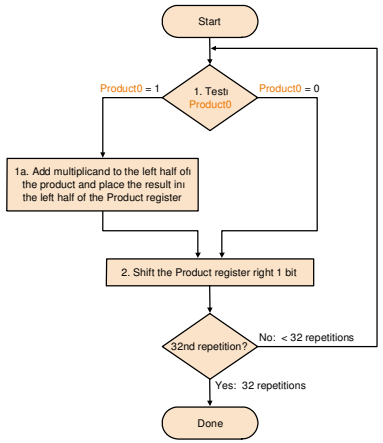


Second Version



Final Version





- **No MIPS:**
  - **dois novos registradores de uso dedicado para multiplicação: Hi e Lo (32 bits cada)**
  - **mult $t1, $t2 # Hi Lo ⇐ $t1 * $t2**
  - **mfhi $t1 # $t1 ⇐ Hi**
  - **mflo $t1 # $t1 ⇐ Lo**

Algoritmo de Booth (visão geral)

- Idéia: “acelerar” multiplicação no caso de cadeia de “1´s” no multiplicador:

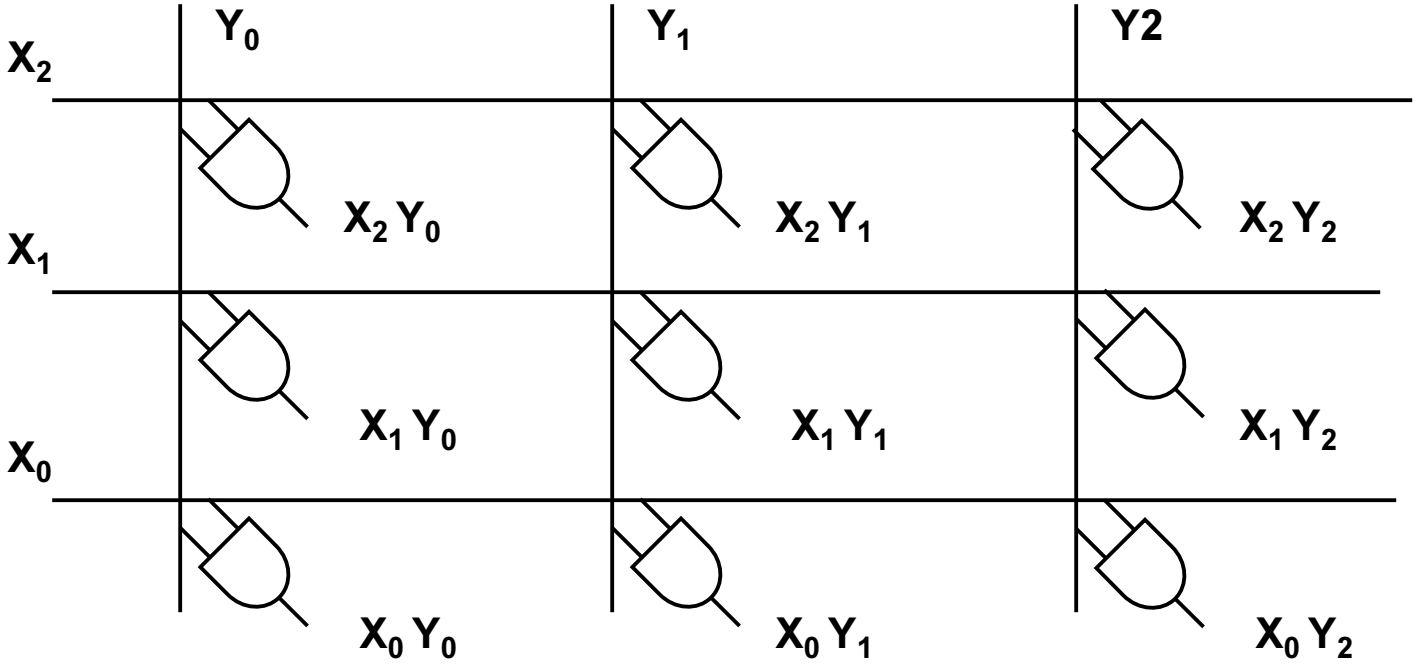
  0 1 1 1 0 * (multiplicando) =

  \+ 1 0 0 0 0 * (multiplicando)

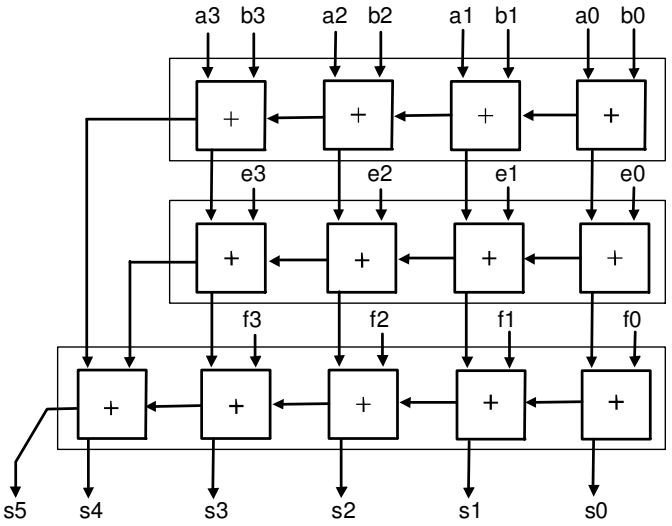
  \- 0 0 0 1 0 * (multiplicando)
- Olhando bits do multiplicador 2 a 2
  - 00 nada
  - 01 soma (final)
  - 10 subtrai (começo)
  - 11 nada (meio da cadeia de uns)
- Funciona também para números negativos
- Para o curso: só os conceitos básicos
- Algoritmo de Booth estendido
  - varre os bits do multiplicador de 2 em 2
- Vantagens:
  - (pensava-se: shift é mais rápido do que soma)
  - gera metade dos produtos parciais: metade dos ciclos

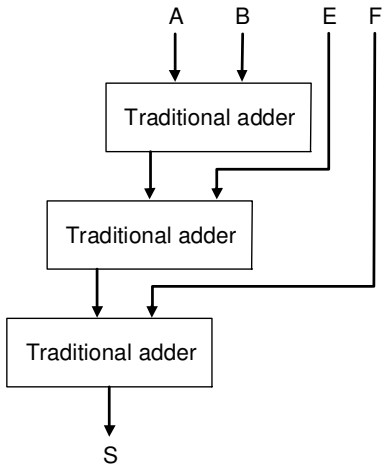
Geração rápida dos produtos parciais

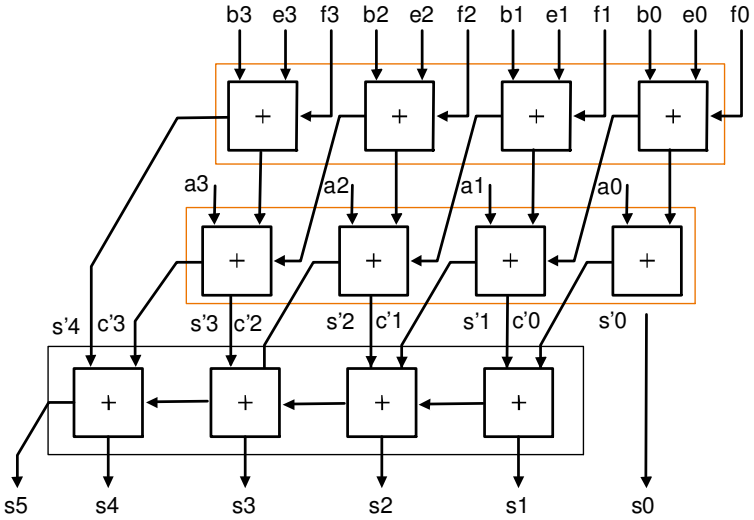
		Y_0	Y_1	Y_2
		X_0	X_1	X_2
		$X_2 Y_0$	$X_2 Y_1$	$X_2 Y_2$
	$X_1 Y_0$	$X_1 Y_1$	$X_1 Y_2$	
$X_0 Y_0$	$X_0 Y_1$	$X_0 Y_2$		

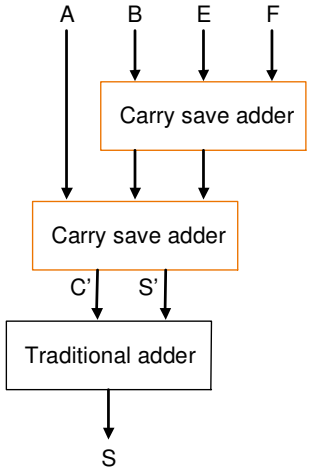


Carry Save Adders (soma de produtos parciais)









Divisão

$29 \div 3 \Rightarrow$ $29 = 3 * Q + R = 3 * 9 + 2$

dividendo, divisor, quociente, resto

$29_{10} = 011101$ $3_{10} = 11$

```
0 1 1 1 0 1 | 1 1
  1 1       -------
-------       0 1 0 0 1
  0 0 1 0 1
      1 1
-------
      1 0
```

Q = 9 R = 2

Como implementar em hardware?

Alternativa 1: divisão com restauração

- hardware não sabe se “vai caber ou não”
- registrador para guardar resto parcial
- verificação do sinal do resto parcial
- caso negativo ⇒ restauração

$29 - 3 * 2^4 = -19$	$q_4 = 1$	
$-19 + 3 * 2^4 = 29$	$q_4 = 0$	Restauração
$29 - 3 * 2^3 = 5$	$q_3 = 1$	
$5 - 3 * 2^2 = -7$	$q_2 = 1$	
$-7 + 3 * 2^2 = 5$	$q_2 = 0$	Restauração
$5 - 3 * 2^1 = -1$	$q_1 = 1$	
$-1 + 3 * 2^1 = 5$	$q_1 = 0$	Restauração
$5 - 3 * 2^0 = 2$	$q_0 = 1$	

$R = 11 = 2$ $\quad q_4q_3q_2q_1q_0 = 01001 = 9$

Alternativa 2: divisão sem restauração

Regras

se resto parcial	> 0	próxima operação	soma	objetivo
se resto parcial	< 0	próxima operação	subtração	$R \rightarrow 0$

se operação corrente	+	$q_i = \cancel{1}$
se operação corrente	-	$q_i = 1$

$29 - 3 * 2^4 = -19 < 0$	próx = SOMA	$q_4 = 1$
$-19 + 3 * 2^3 = 5 > 0$	próx = SUB	$q_3 = \cancel{1}$
$5 - 3 * 2^2 = -7 < 0$	próx = SOMA	$q_2 = 1$
$-7 + 3 * 2^1 = -1 < 0$	próx = SOMA	$q_1 = \cancel{1}$
$-1 + 3 * 2^0 = 2$		$q_0 = \cancel{1}$

Resto = 2 **Quociente =** $1\cancel{1}1\cancel{1}\cancel{1}$ **??**

Alternativa 2: conversão do resultado

$$1\ \bar{1}\ 1\ \bar{1}\ \bar{1} = (2^4 - 2^3 + 2^2 - 2^1 - 2^0)$$

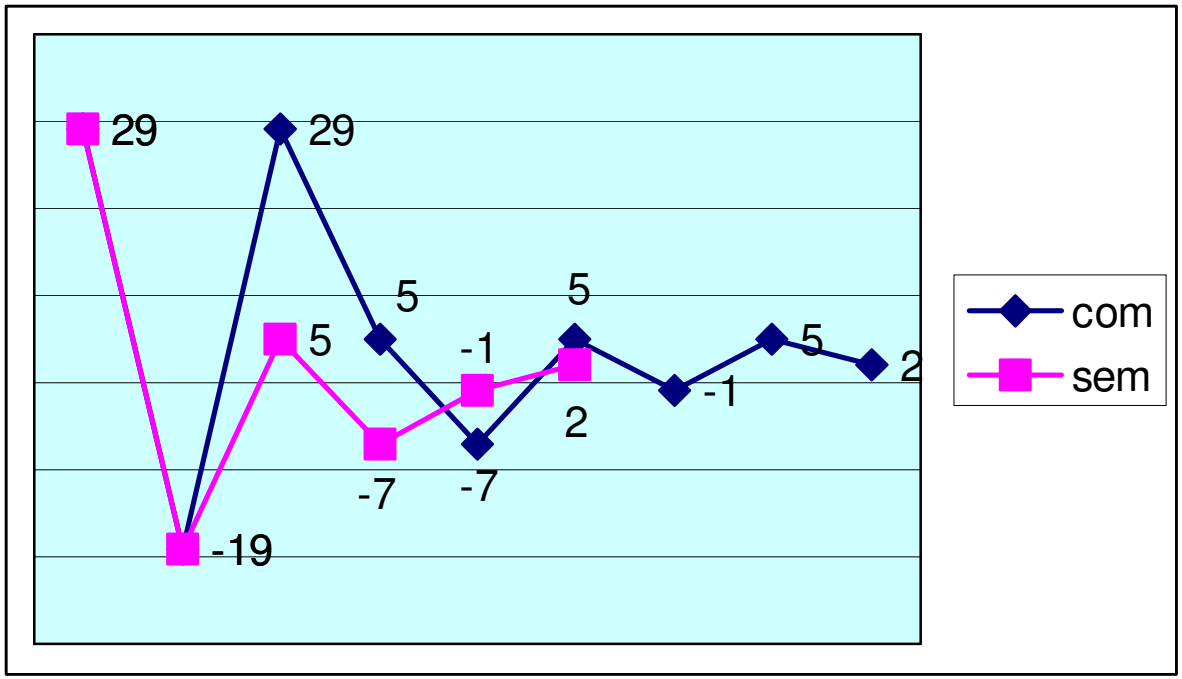
16 - 8 + 4 - 2 - 1

$$\ldots 1\ \bar{1} \ldots = 2^n - 2^{(n-1)} = 2^{(n-1)}(2-1) = 2^{(n-1)}$$

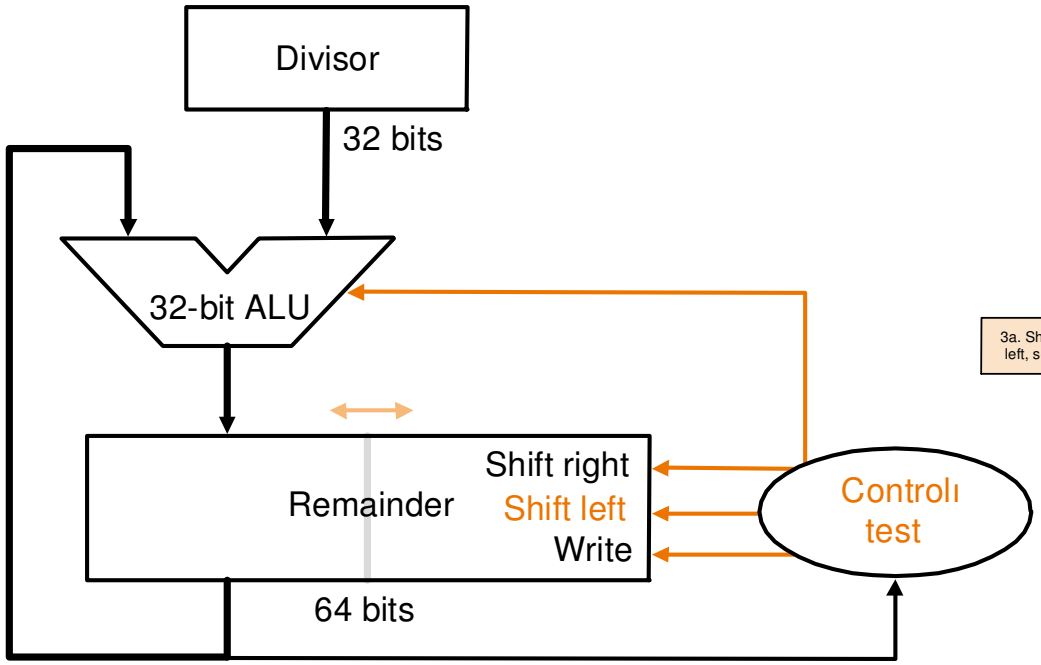
$$\begin{array}{r} 1\ \bar{1}\ 1\ \bar{1}\ \bar{1} \\ 0\ 1\ 0\ 1\ \bar{1} \\ 0\ 1 \end{array}$$

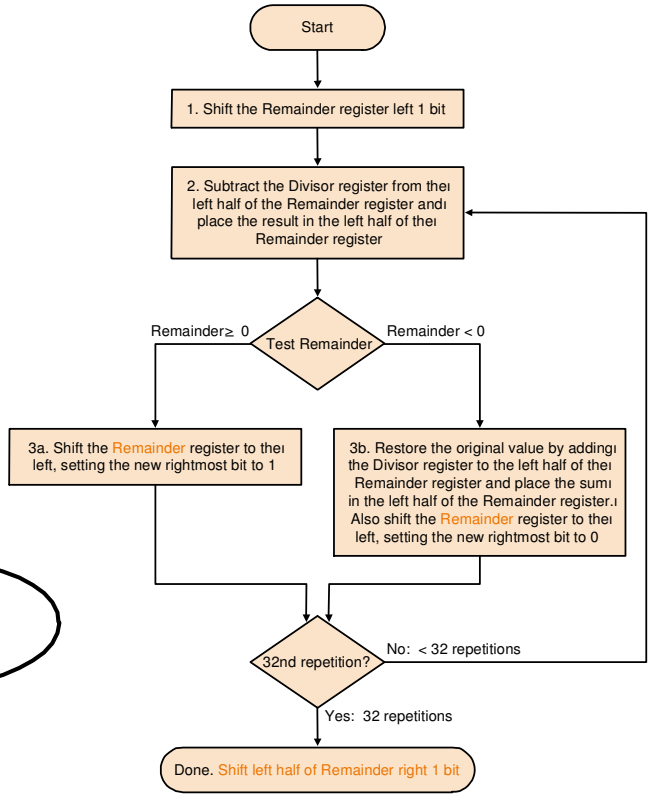
- **Nº de somas: 3**
- **Nº de subtrações:2**
- **Total: 5**
- **OBS: se resto < 0 deve haver correção de um divisor para que resto > 0**

Comparação das alternativas



Hardware para divisão: terceira alternativa





Instruções

- **No MIPS:**
  - **dois novos registradores de uso dedicado para multiplicação: Hi e Lo (32 bits cada)**
  - **mult $t1, $t2 # Hi Lo ⇐ $t1 * $t2**
  - **mfhi $t1 # $t1 ⇐ Hi**
  - **mflo $t1 # $t1 ⇐ Lo**

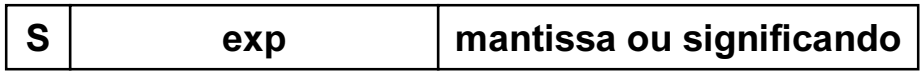
- **Para divisão:**
  - **div $s2, $s3 # Lo ⇐ $s3 / $s3**
    Hi ⇐ $s3 mod $s3
  - **divu $s2, $s3 # idem para "unsigned"**

Ponto Flutuante

- **Objetivos:**
  - **representação de números não inteiros**
  - **aumentar a capacidade de representação (maiores ou menores)**
- **Formato padronizado**

  1.XXXXXXXXX ..... * 2^{yyy} (no caso geral B^{yyy})

- **No MIPS:**



↑ **exp** ↑ **faixa**

↑ **mantissa** ↑ **precisão**

sinal-magnitude $(-1)^S$ **F * 2^E**

Ponto Flutuante e padrão IEEE 754

expoente $\in$ [-128 , 127] **se $2^{10} \approx 10^3$**

128 = 8 + 10 * 12;

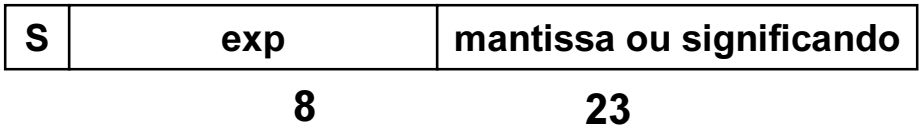
$2^{128} = 2^{(8 + 10 * 12)} = 2^8 * 2^{(10 * 12)} \approx 2 * 10^{38}$

overflow $\Rightarrow$ Nº > 10^{38}
underflow $\Rightarrow$ Nº < 10^{-38}

PADRÃO IEEE 754

um implícito

1.XXXXXXXXXXX

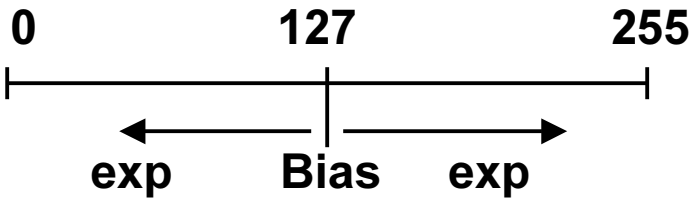


mantissa: precisão simples: 23 bits (+1)
precisão dupla: 52 bits (+1)

Padrão IEEE754: bias

Nº = $(-1)^S$ (1 + Mantissa) * 2^E

Para simplificar a ordenação (sorting): BIAS



No padrão: $2^{(nE - 1)} - 1 = 127$

EXP = $CAMPO_{EXP}$ - BIAS

Exemplo: representar - $0{,}75_{10}$ = - (1/2 + 1/4)

- $0{,}75_{10}$ = - $0{,}11_2$ = $-1{,}11 * 2^{-1}$

mantissa = 1000000 ..... (23 bits)

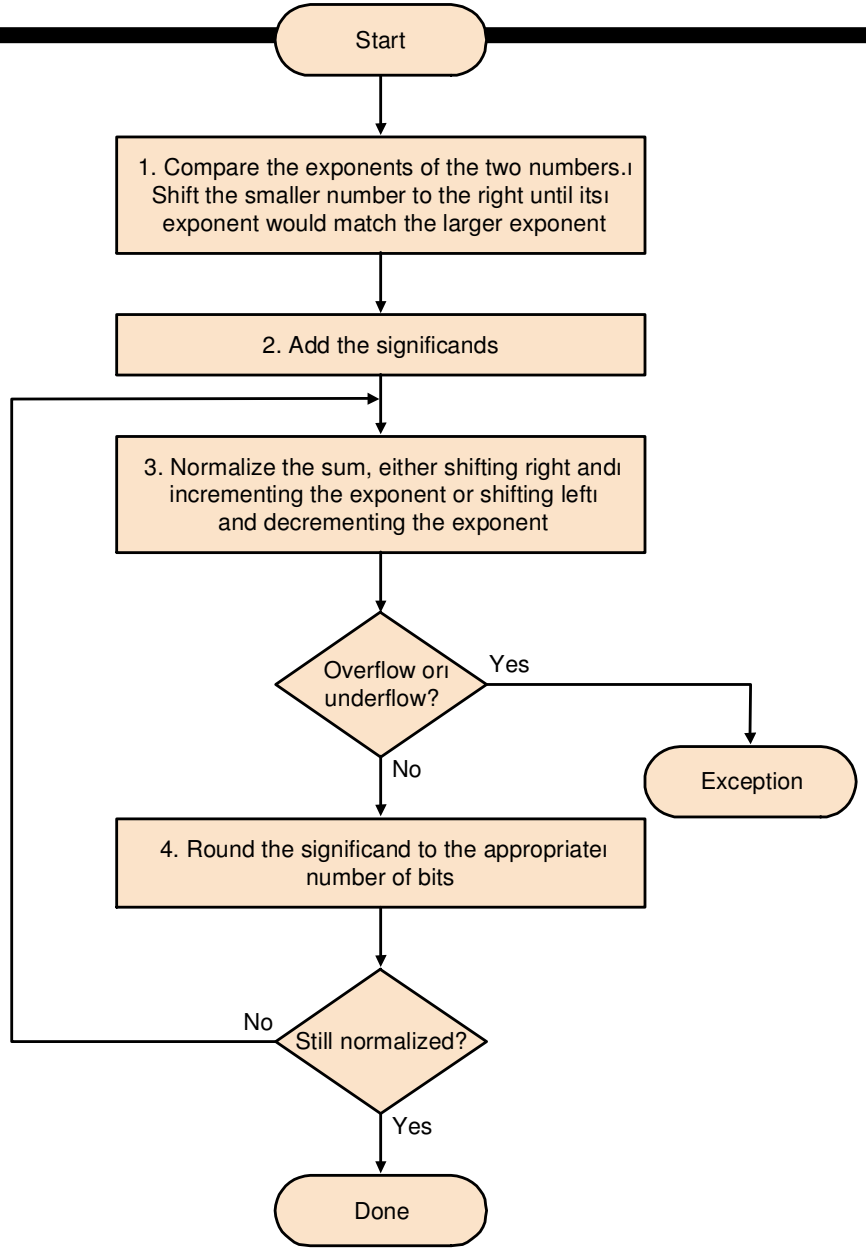
campo expoente: - 1 + 127 = 126_{10} = $0111\ 1110_2$

1	0111 1110	1000 0000 0000 0000 0000 000

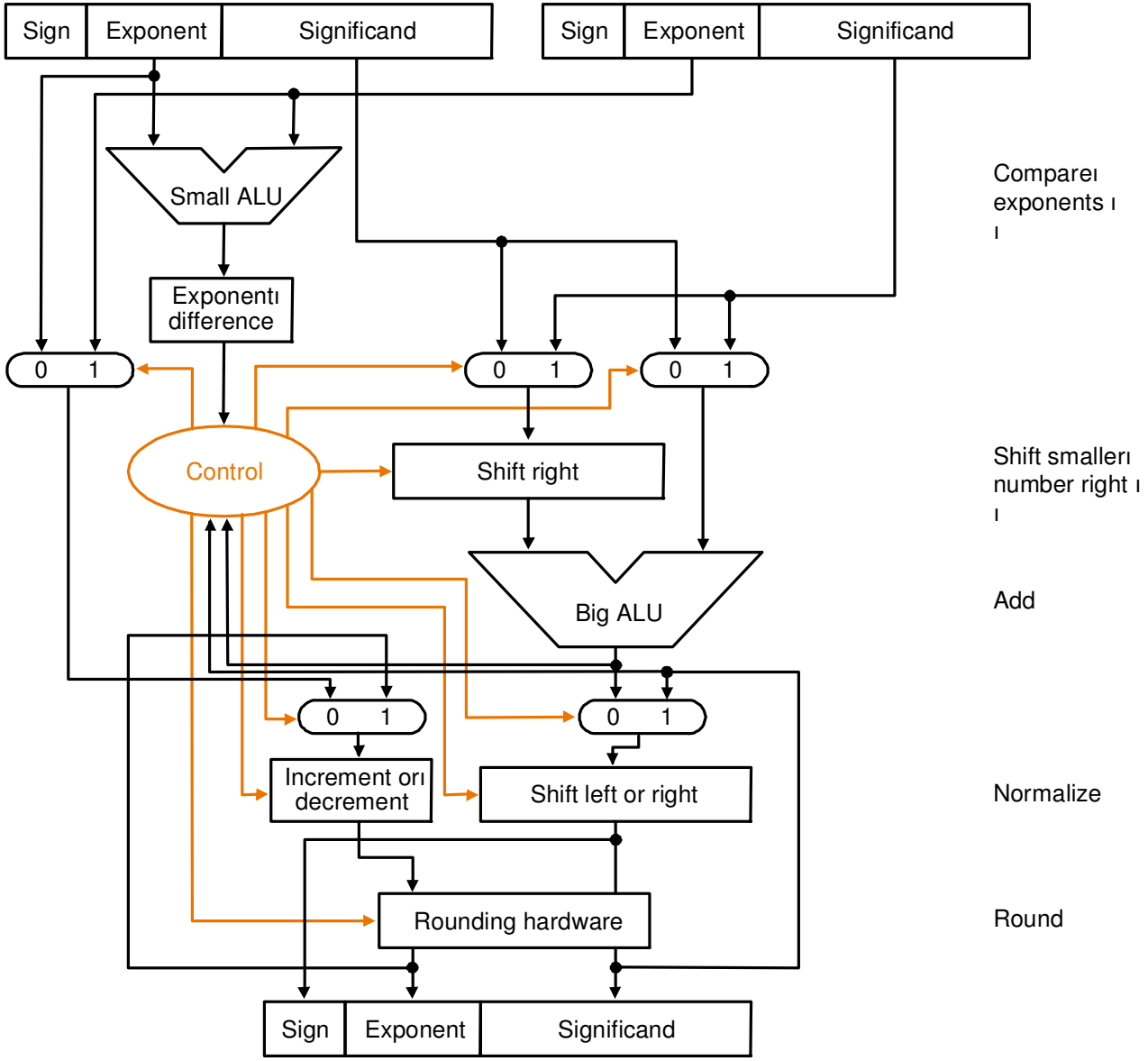
Tabela de faixas de representação do IEEE 754

Precisão simples		Precisão dupla		Significado
Expoente	Mantissa	Expoente	Mantissa	
0	0	0	0	0
0	$\neq 0$	0	$\neq 0$	Número não normalizado
1 – 254	qquer	1 – 2046	qquer	Número normalizado
255	0	2047	0	infinito
255	$\neq 0$	2047	$\neq 0$	NaN (not a number)
8bits	23(+1)	11	52(+1)	

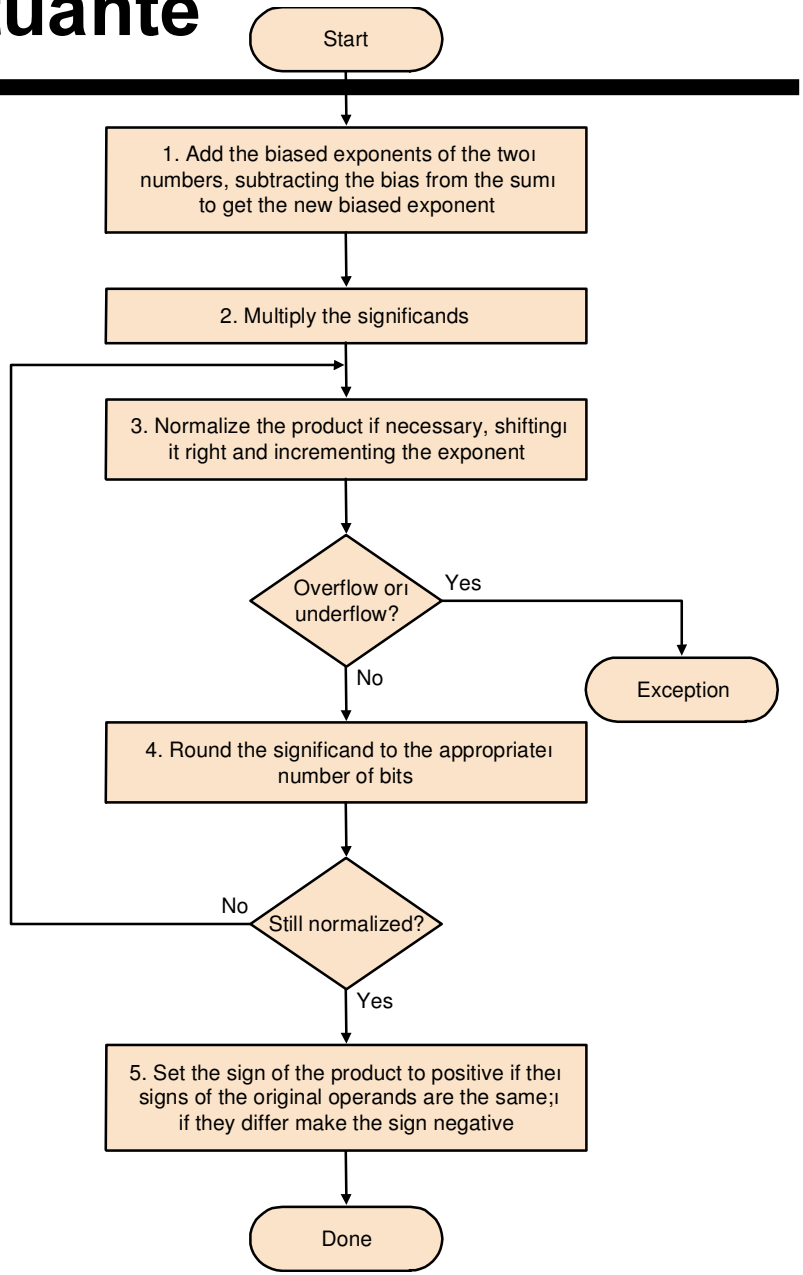
Soma em ponto flutuante



ULA para soma em ponto flutuante



Multiplicação em ponto flutuante



Conjunto de instruções do MIPS para fp

Fig 4.47 Pag 291

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Chapter Four Summary

- **Computer arithmetic is constrained by limited precision**
- **Bit patterns have no inherent meaning but standards do exist**
  - **two's complement**
  - **IEEE 754 floating point**
- **Computer instructions determine "meaning" of the bit patterns**
- **Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).**

- **We are ready to move on (and implement the processor)**

  you may want to look back (Section 4.12 is great reading!)