



Aula 9

Aritmética Computacional

Aritmética Inteira





Principais Arquiteturas Aritméticas

■ Pilha:

- As operações são sempre realizadas com os argumentos na pilha e o resultado é também armazenado na pilha.

Ex.: calculadoras HP, Linguagem Forth, x87(ST0-ST7, 80 bits)

■ Acumulador:

- As operações são feitas sobre registradores (incluindo A) e o resultado armazenado em um registrador especial chamado Acumulador (A)

Ex.: Z80, 8051

■ Registrador-Registrador:

- As operações são feitas sobre registradores e o resultado é armazenado em qualquer registrador.

Ex.: RISC-V, MIPS, ARM

■ Registrador-Memória:

- As operações aritméticas buscam um dos argumentos na memória e armazenam o resultado em um registrador.

Ex.: x86, x64



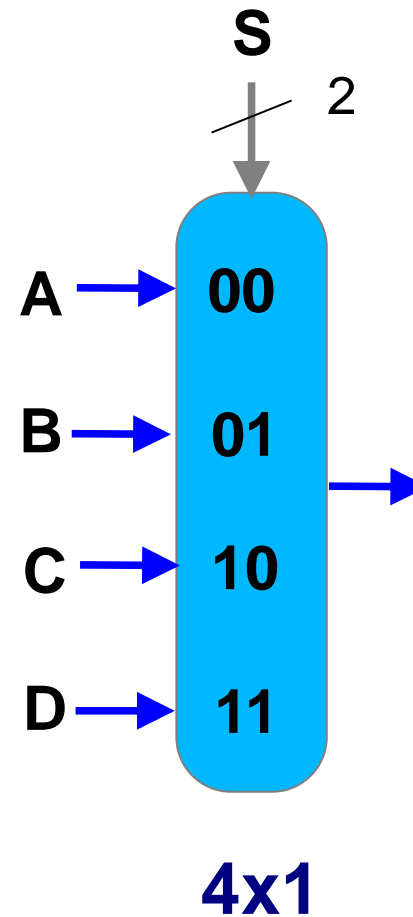
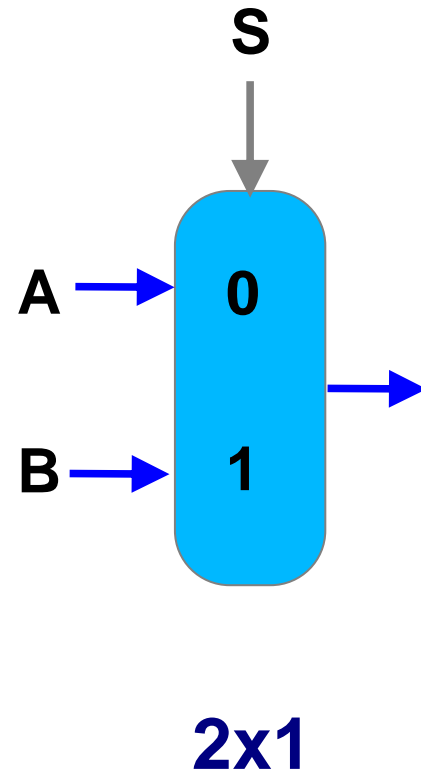
Unidade Lógica e Aritmética

- A arquitetura RV32I necessita uma ULA de 32 bits
- Para exemplificar, construiremos uma ULA com as seguintes operações:
 - Soma e subtração em complemento de 2
 - Operações lógicas **and**, **or** e **nor** (só para exemplificar)
 - Suporte à instrução **slt**
 - detecção de overflow (no RISC-V não é implementada)
 - detecção de igualdade



Elementos Básicos

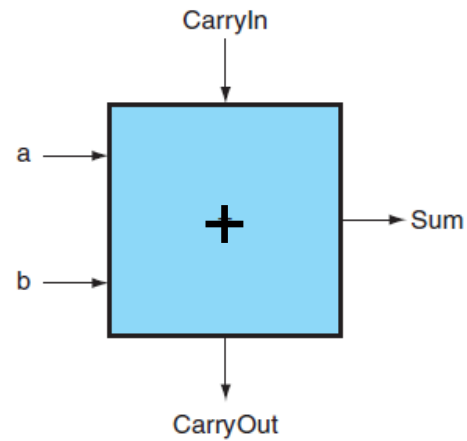
■ Multiplexador





Elementos Básicos

■ Somador



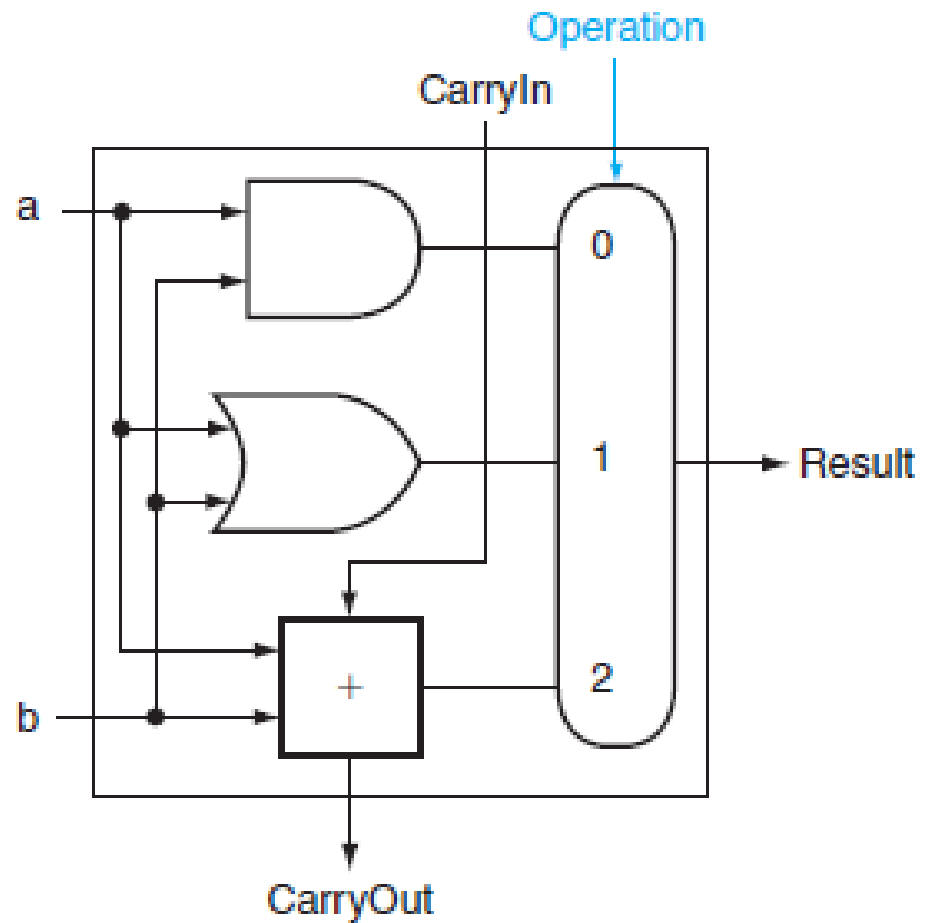
Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{\text{two}}$
0	0	1	0	1	$0 + 0 + 1 = 01_{\text{two}}$
0	1	0	0	1	$0 + 1 + 0 = 01_{\text{two}}$
0	1	1	1	0	$0 + 1 + 1 = 10_{\text{two}}$
1	0	0	0	1	$1 + 0 + 0 = 01_{\text{two}}$
1	0	1	1	0	$1 + 0 + 1 = 10_{\text{two}}$
1	1	0	1	0	$1 + 1 + 0 = 10_{\text{two}}$
1	1	1	1	1	$1 + 1 + 1 = 11_{\text{two}}$

Ex.: Rever a implementação com portas lógicas



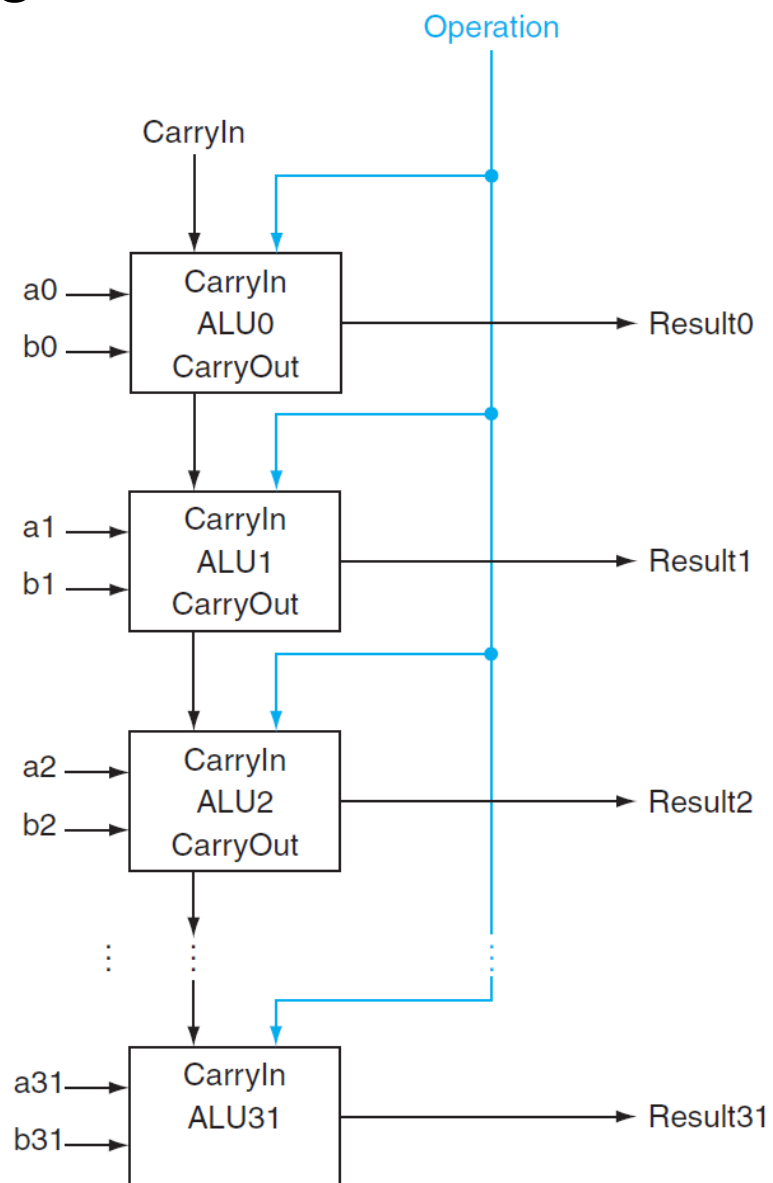
ULA de 1-bit

Operações de soma, and, or :

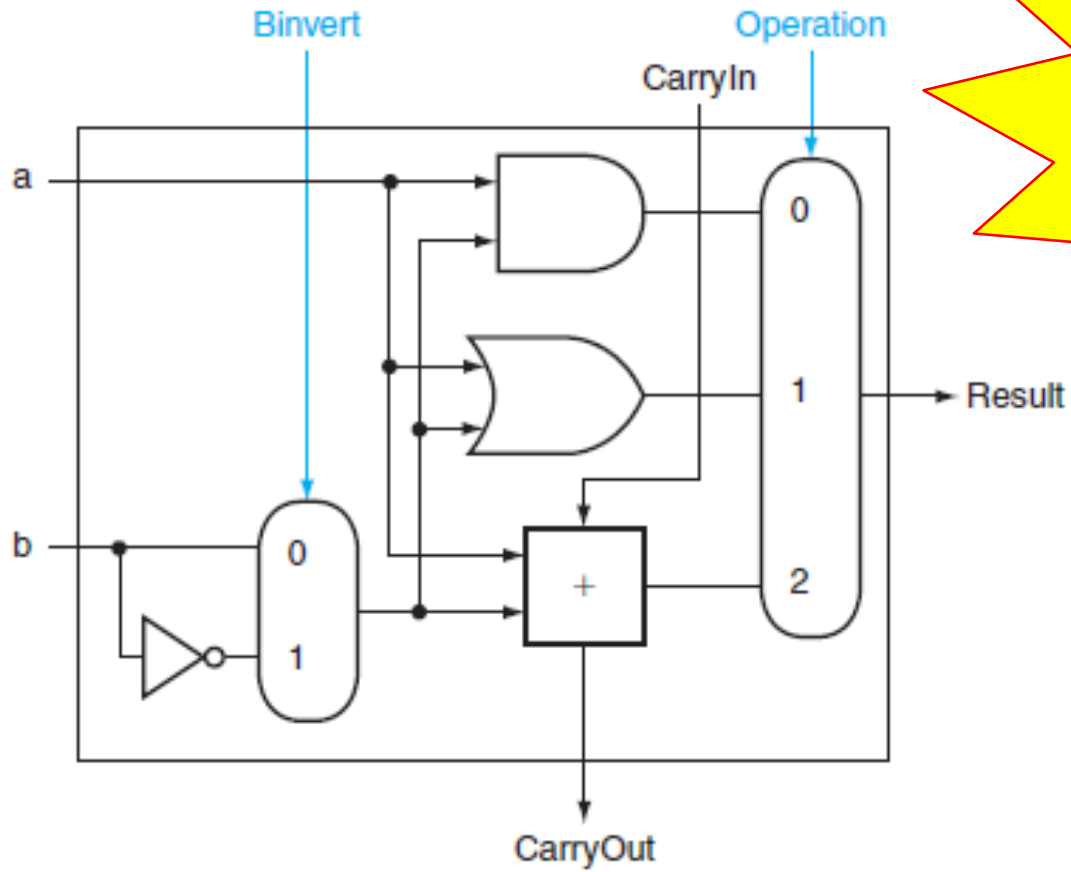




ULA de 32 bits



Incluindo Subtração:



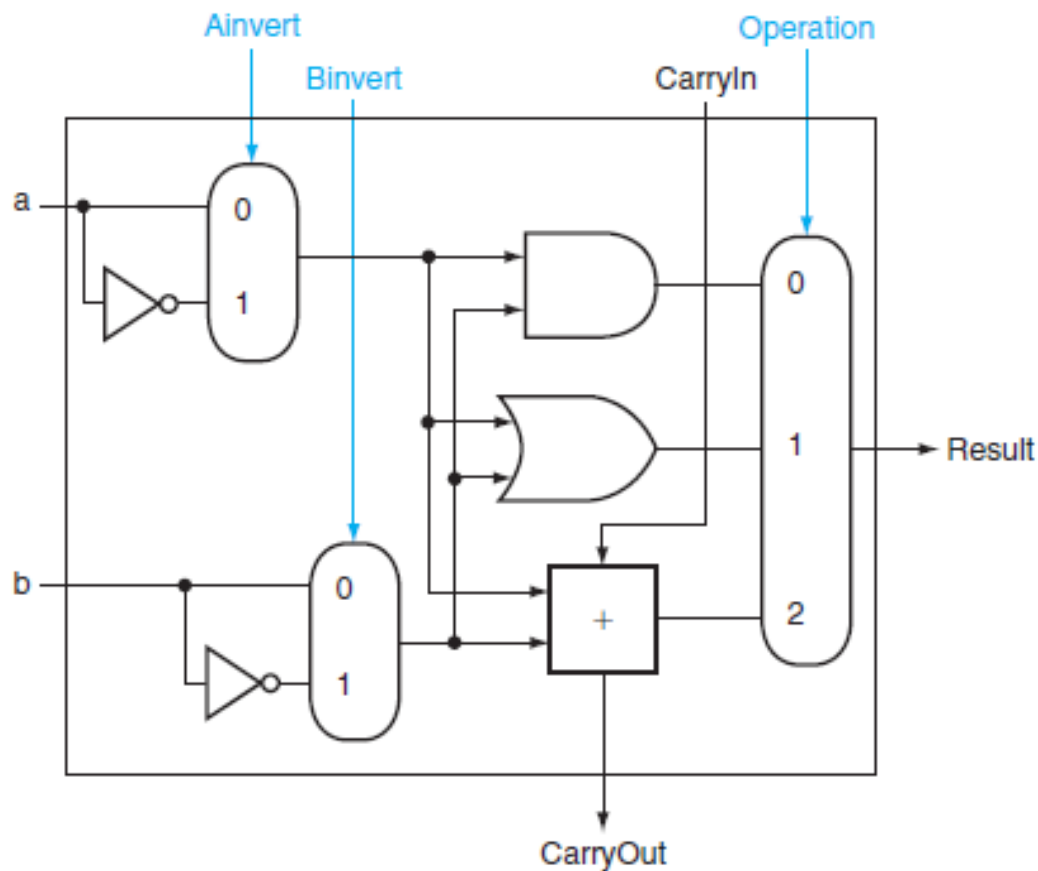
Eis o motivo
do uso do
complemento
de 2!

Obs.: Em Complemento de 2: $\mathbf{a - b = a + (\overline{b} + 1)}$



Operação nor:

Pelo Teorema de DeMorgan: $\overline{a + b} = \bar{a} \cdot \bar{b}$



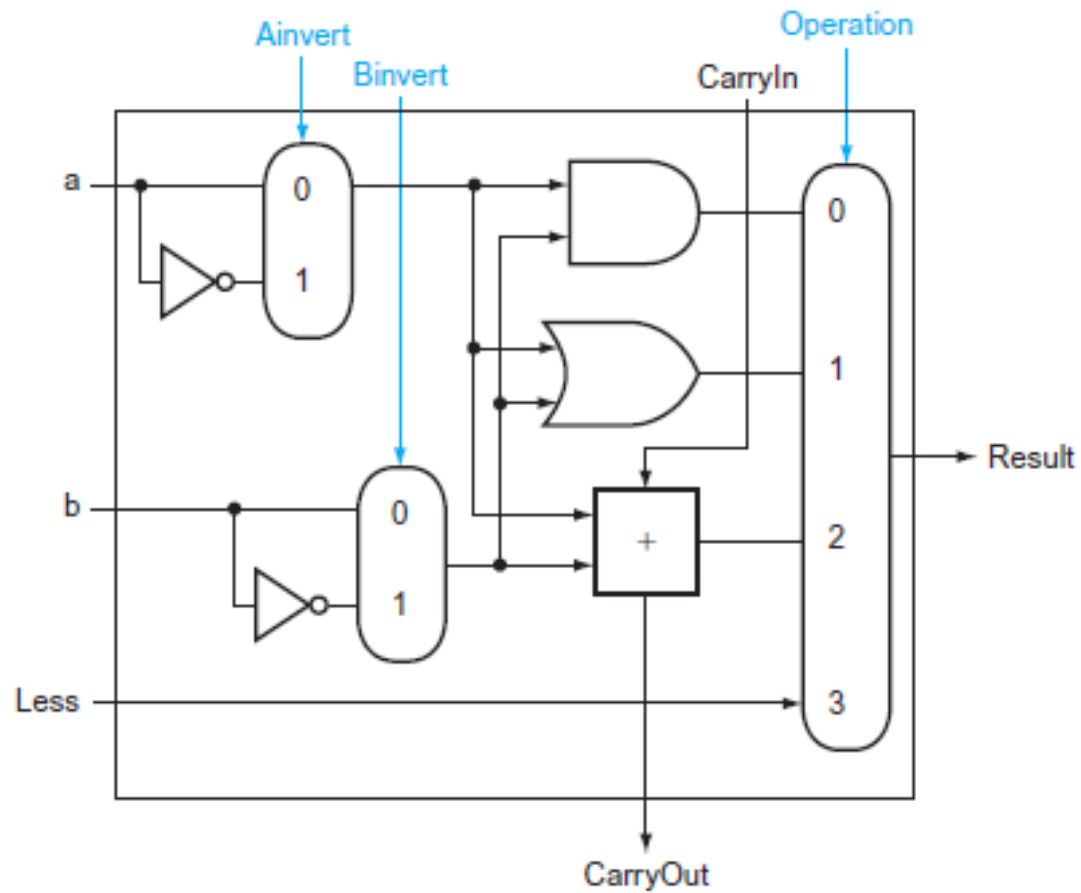


Instrução slt

- A instrução **slt** gera saída 1 se $R[rs1] < R[rs2]$, e 0 caso contrário. Assim, todos os bits, com exceção do bit menos significativo, são fixados em zero.
- O bit menos significativo depende do resultado da comparação.
- Inclui-se uma entrada chamada **Less**



ULA de 1-bit com slt (entrada Less)



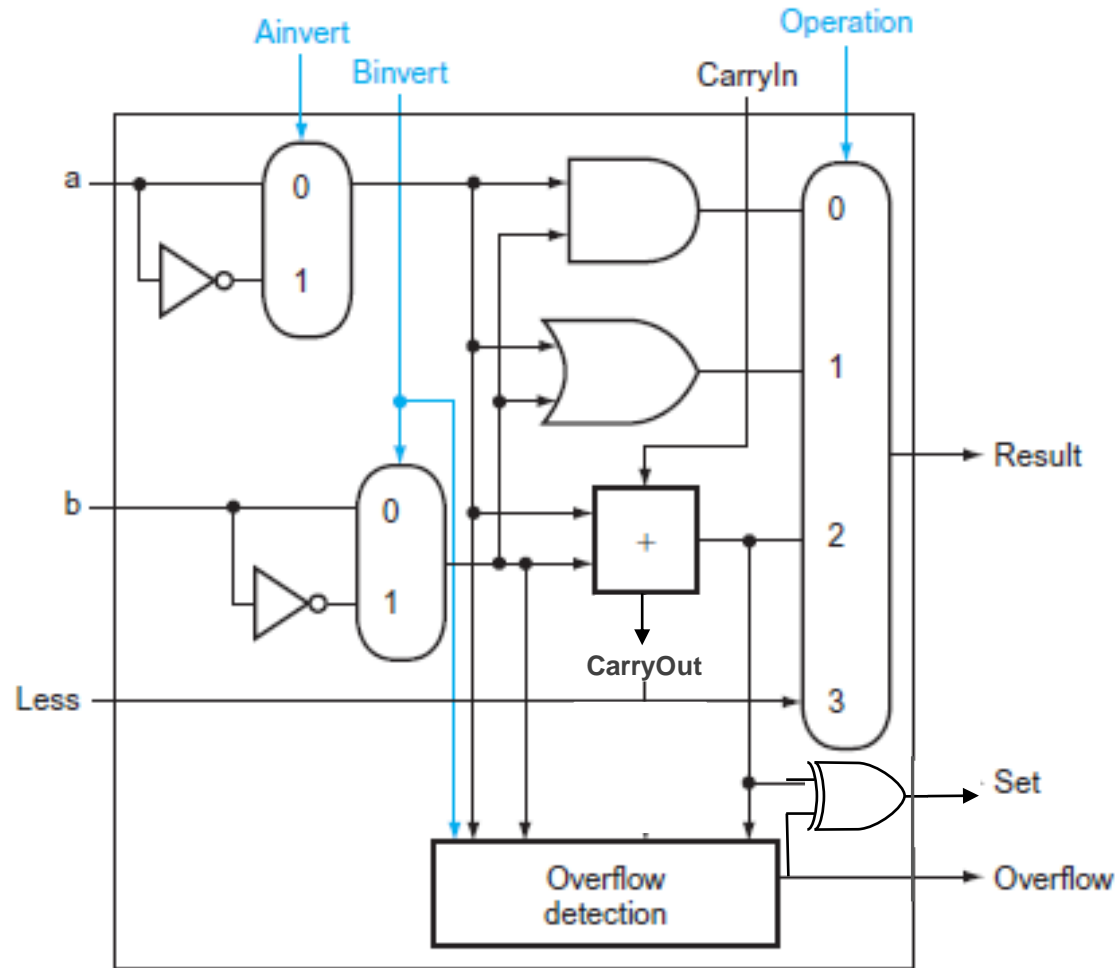


Calculando slt

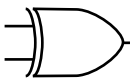
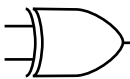
- O bit menos significativo da ULA deve ser 1 se $A < B$
- Calcula-se este valor subtraindo A de B e tomando-se o bit mais significativo (“Sinal”):
 - se $A - B < 0$ então $A < B$
- Utiliza-se o próprio subtrator da ULA para obter este valor, modificando-se o último estágio
- Cuidar que pode ocorrer overflow!!!



Último estágio da ULA, com Set e detector Overflow

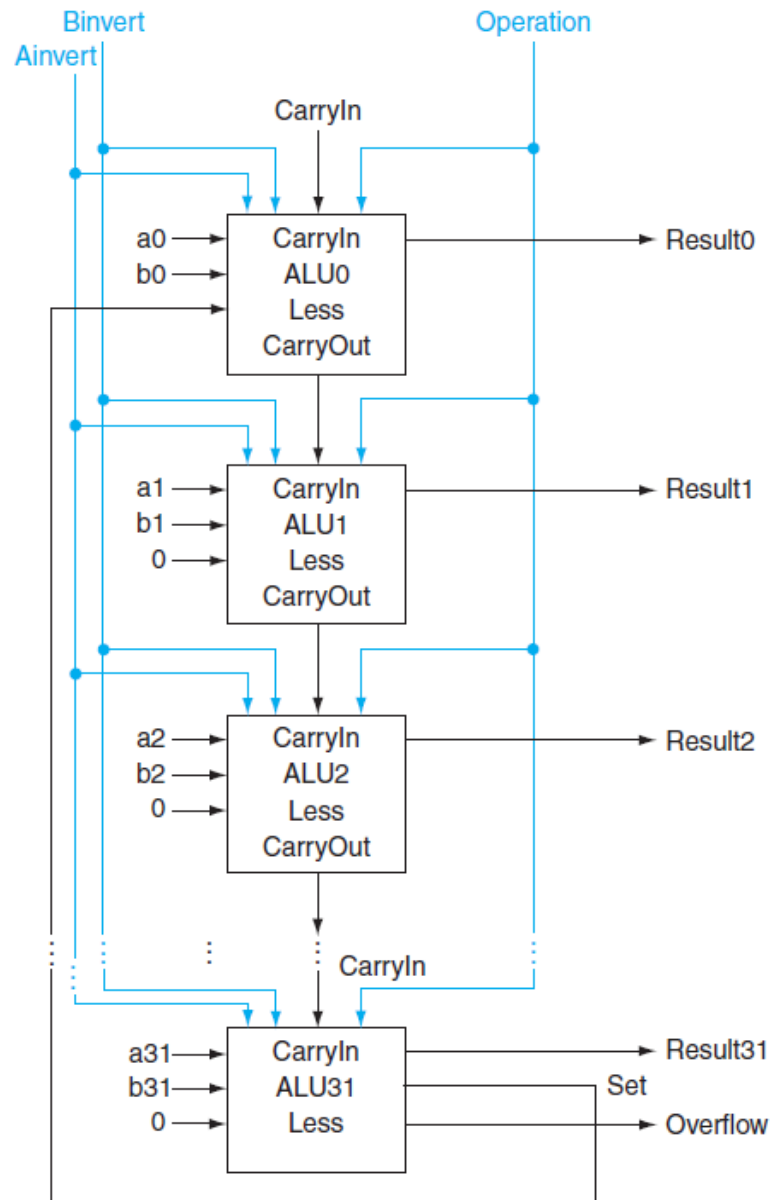


Forma alternativa:

CarryIn  CarryOut  Overflow



ULA de 32 bits



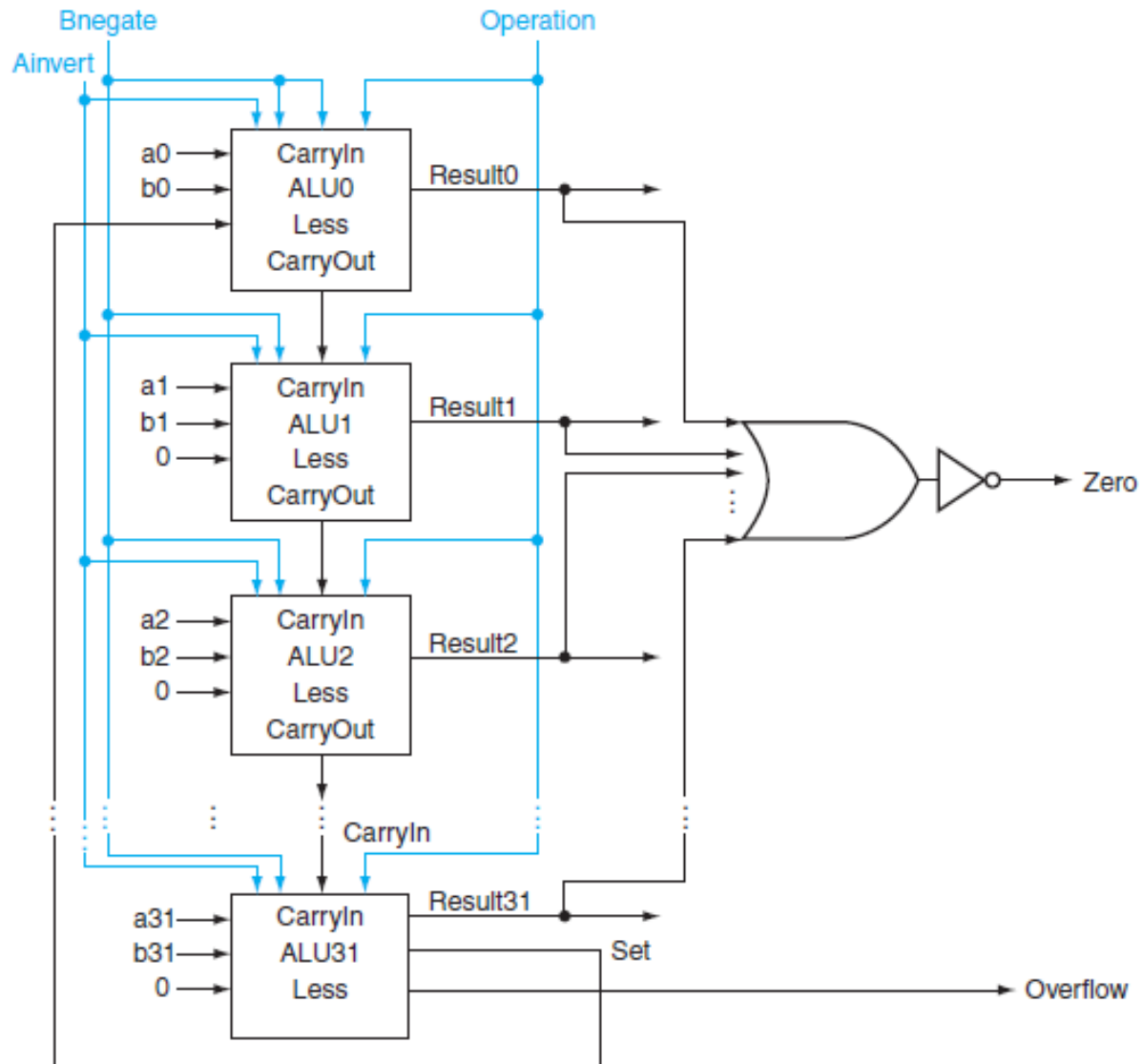


Testando igualdade

- As instruções bne e beq testam se dois valores são iguais ou não
- Para testar igualdade subtrai-se os dois operandos e testa-se se o resultado é zero:
 - $A = B$ se $A - B = 0 = \text{Result}$
 - Teste se o resultado é zero: operação NOR entre os bits do resultado
 - $\text{Zero} = \overline{(\text{Result}_0 + \text{Result}_1 + \text{Result}_2 + \cdots + \text{Result}_{31})}$



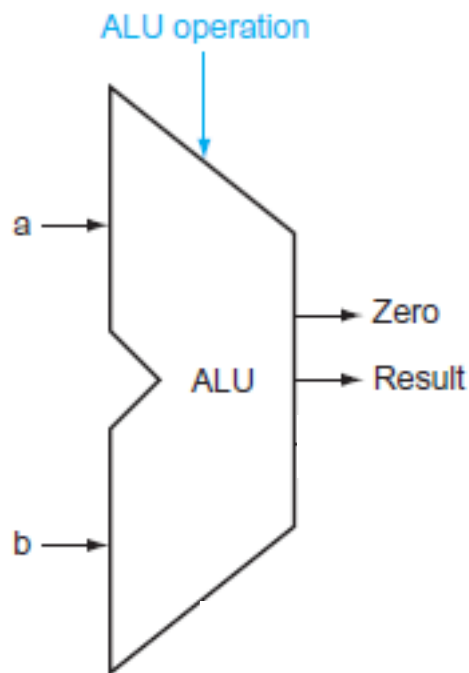
ULA 32 bits com Comparação





ULA: Linhas de Controle e Simbologia

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR



Obs.: Quais os tamanhos das linhas?

Implementação em Verilog

```
1  /*
2   * ALU
3   * Arithmetic Logic Unit with control signals as defined by the COD book.
4   */
5  module ALU (
6      input signed [31:0] iA, iB,
7      input [3:0] iControl,
8      output oZero,
9      output [31:0] oALUresult
10  );
11
12  assign oZero = (oALUresult == 32'b0);
13
14  always @(*)
15  begin
16      case (iControl)
17          4'b0000:
18              oALUresult = iA & iB;
19          4'b0001:
20              oALUresult = iA | iB;
21          4'b0010:
22              oALUresult = iA + iB;
23          4'b0110:
24              oALUresult = iA - iB;
25          4'b0111:
26              oALUresult = (iA < iB ? 32'd1 : 32'd0);
27          default:
28              oALUresult = 32'b0;
29      endcase
30  end
31  endmodule
```

Cuidar pois esta não é uma implementação do projeto feito anteriormente!!!!

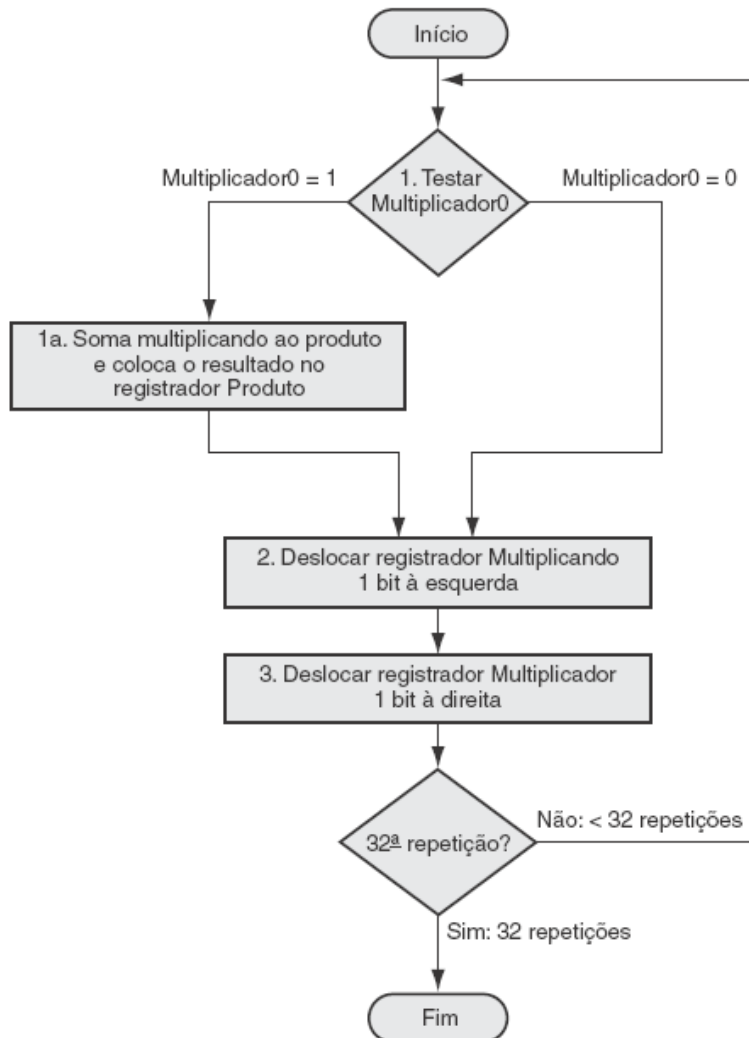
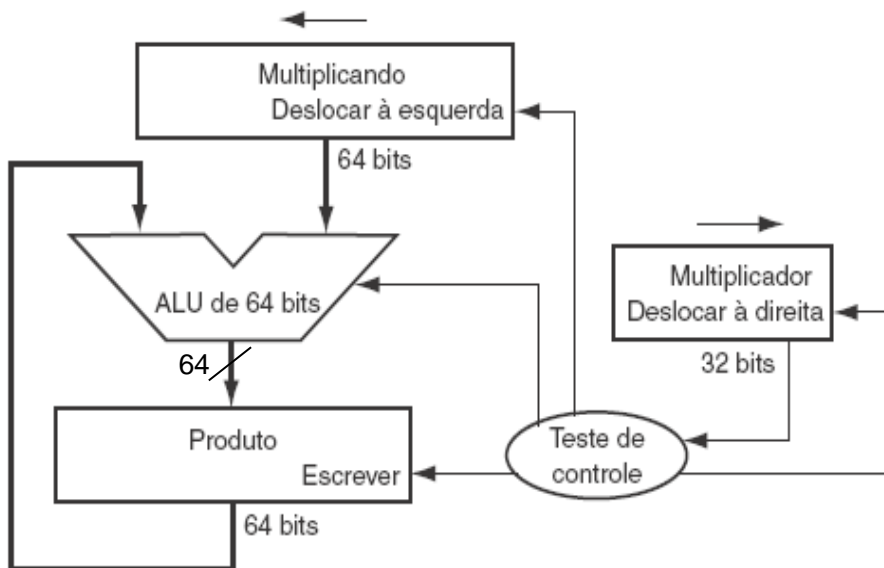


Multiplicação

- Mais complexa do que a adição
 - realizada através de deslocamento e adição
 - Requer mais tempo e mais área de chip
 - Veremos apenas três versões
 - Números negativos: converta para sem sinal (número natural), multiplique usando o algoritmo clássico e defina o sinal do resultado.
- existem técnicas mais eficientes, que não serão tratadas neste curso (vide Algoritmo de Booth)

Multiplicação: Versão Sequencial

0010 (multiplicando)
x 1011 (multiplicador)

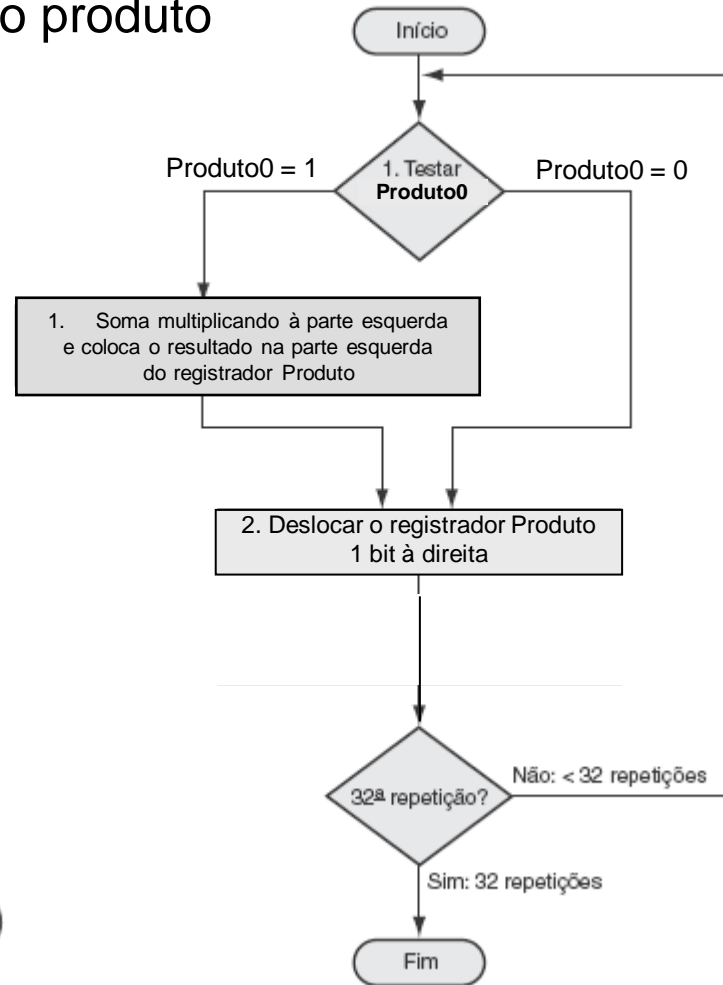
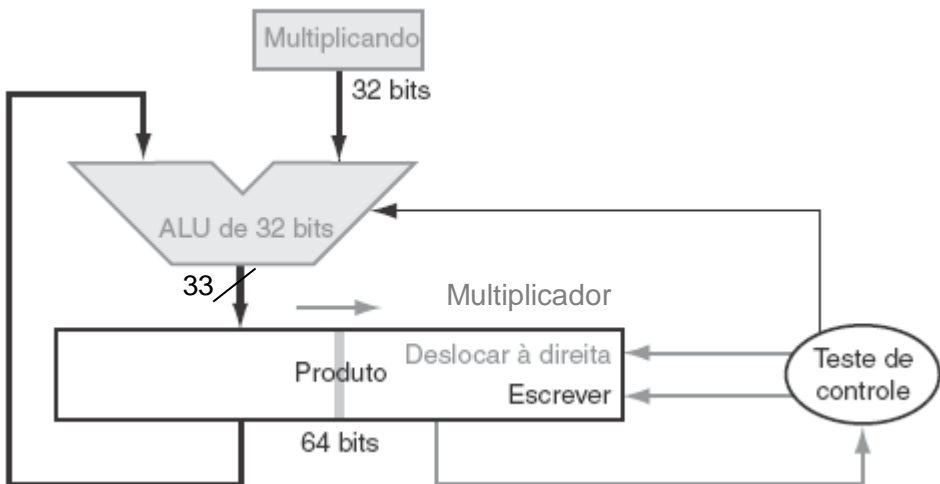




Versão Refinada

O multiplicador inicia na metade direita do produto

0010 (multiplicando)
x 1011 (multiplicador)



Tempo de atraso?



Multiplicação: Versão Combinacional

- Maior área em chip
- Maior custo
- Tempo de atraso?

-Versão com menor atraso

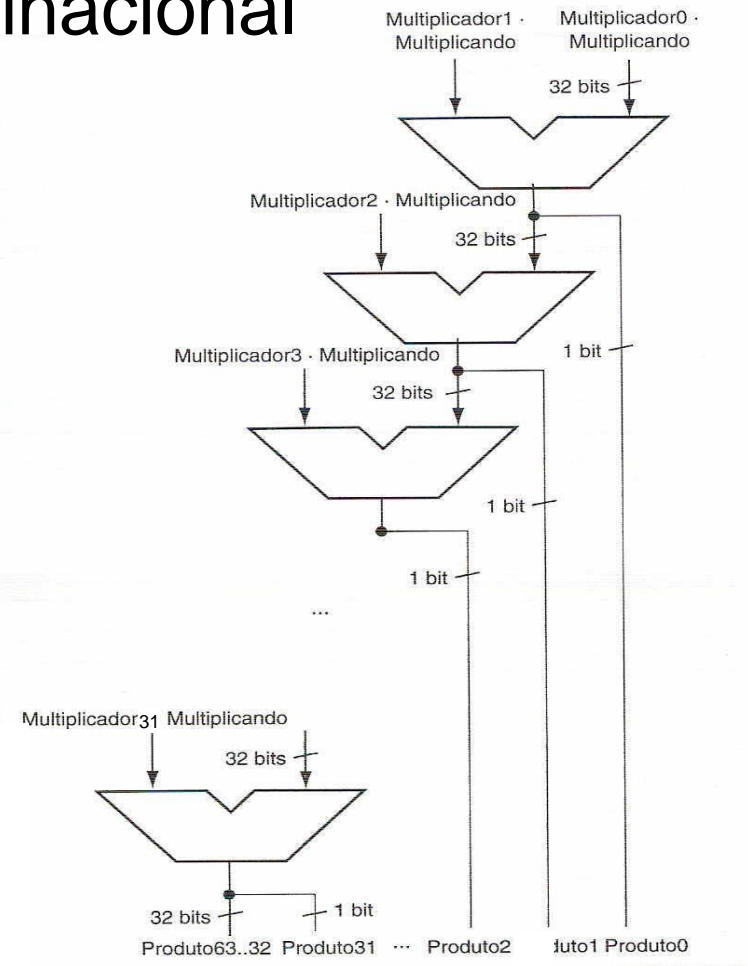
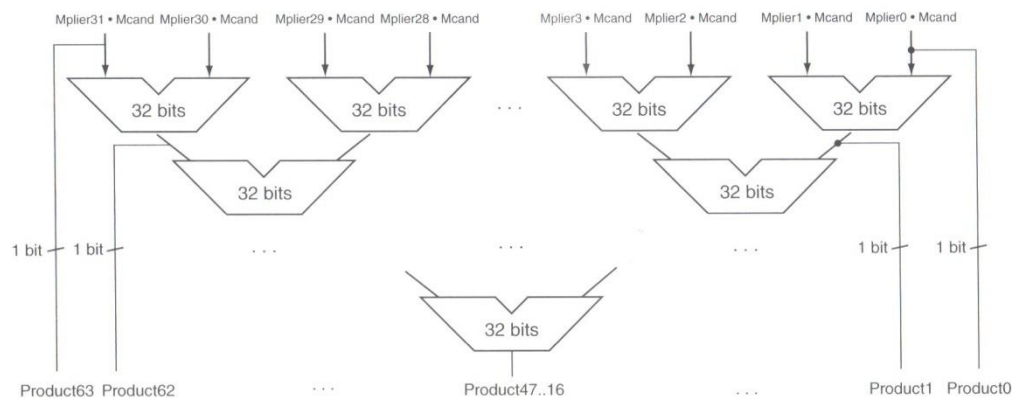


FIGURE 3.8 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware “unrolls the loop” to use 31 adders and then organizes them to minimize delay.



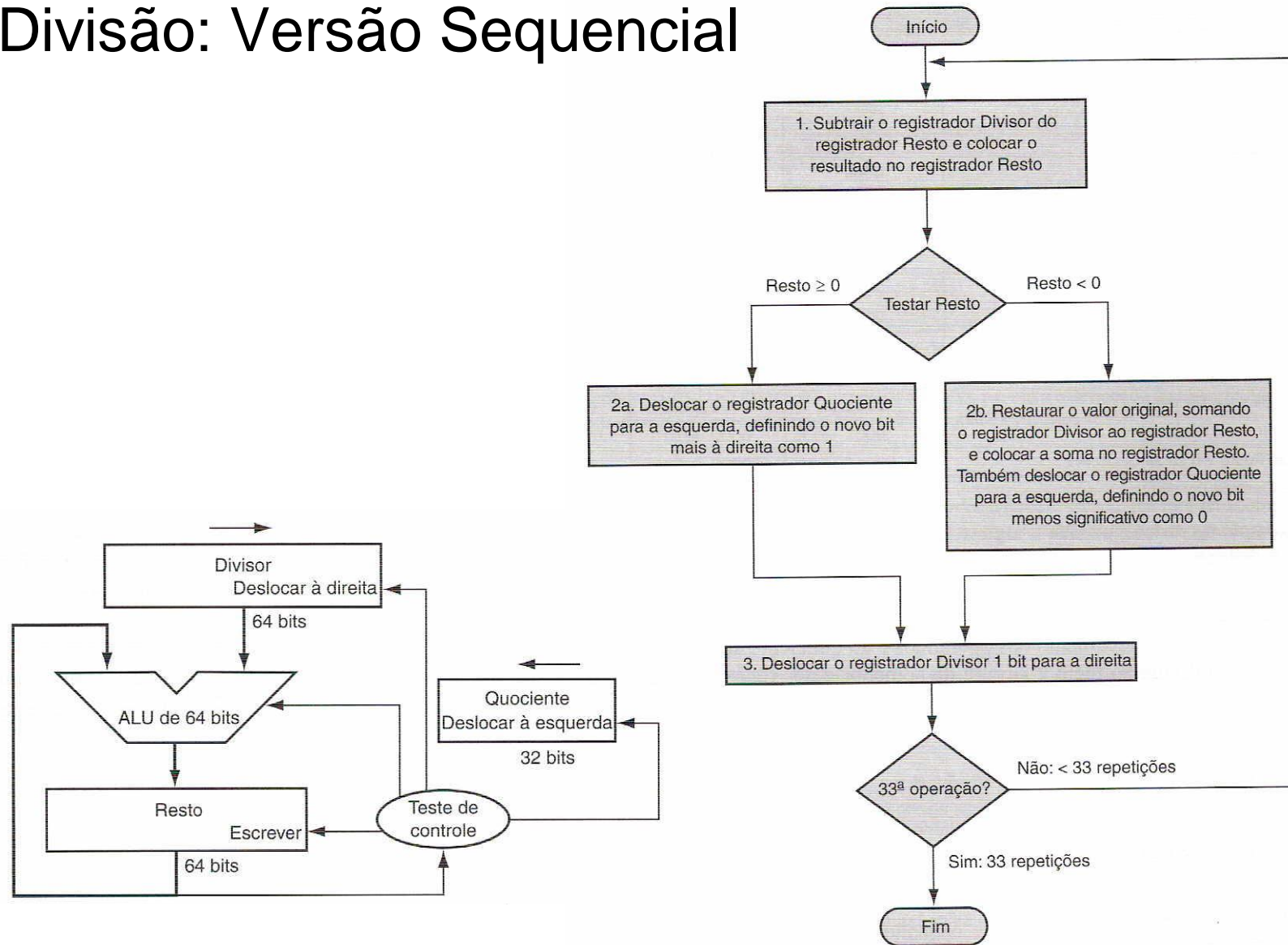
Divisão

- Operação recíproca da multiplicação
- Também será visto apenas o algoritmo clássico usado para números naturais
- Ainda menos frequente e mais peculiar

$$13270 \overline{) 13} \qquad 110100 \overline{) 101}$$

- $\text{Dividendo} = \text{Divisor} \times \text{Quociente} + \text{Resto}$

Divisão: Versão Sequencial



Obs.: Existem versões otimizadas



Multiplicação e Divisão inteira em complemento de 2

Os algoritmos clássicos apenas funcionam com números sinal e magnitude.
Devemos operar com os módulos dos operandos e ajustar o sinal do(s) resultado(s).

Ajuste do sinal do resultado da multiplicação:

Basta verificar se os sinais do multiplicando e do multiplicador são iguais ou diferentes, definindo assim o sinal do produto.

Ajuste dos resultados da divisão:

Precisamos definir o sinal do quociente e do resto.

Dividendo = Quociente x Divisor + Resto

- ☐ $7 \div 2 \rightarrow 7 = 3 \times 2 + 1$
- ☐ $(-7) \div 2 \rightarrow -7 = (-3) \times 2 + (-1)$
- ☐ $7 \div (-2) \rightarrow 7 = (-3) \times (-2) + 1$
- ☐ $(-7) \div (-2) \rightarrow -7 = 3 \times (-2) + (-1)$

Regra: Quociente: Mesma regra da multiplicação.

Resto: Mesmo sinal do Dividendo.



Aritmética inteira na ISA RV32IM - Multiplier

- Adição e subtração: não sinaliza overflow!
 - add, addi, sub # não há subi
 - Ex.: `addi t0, t1, -4` # `t0=t1-4` sempre com extensão de sinal
- Multiplicação: precisa de 64 bits para o resultado
 - mul # LSW da multiplicação
 - mulh # MSW da multiplicação
 - mulhu # MSW considerando os operandos sem sinal
 - mulhsu # MSW considerando um signed e outro unsigned
 - Ex.: `mul t0, t1, t2` # `t0=Lower{t1×t2}`
- Divisão: requer 2 resultados de 32 bits
 - div, divu Cálculo do quociente
 - Ex.: `divu t0, t1, t2` # `t0 = floor(t1/t2)` operandos unsigned
 - Não sinaliza erro em caso divisão por zero! (o RARS sinaliza em fcsr)
 - rem, remu Cálculo do resto
 - Ex.: `rem t0, t1, t2` # `t0 = t1%t2`