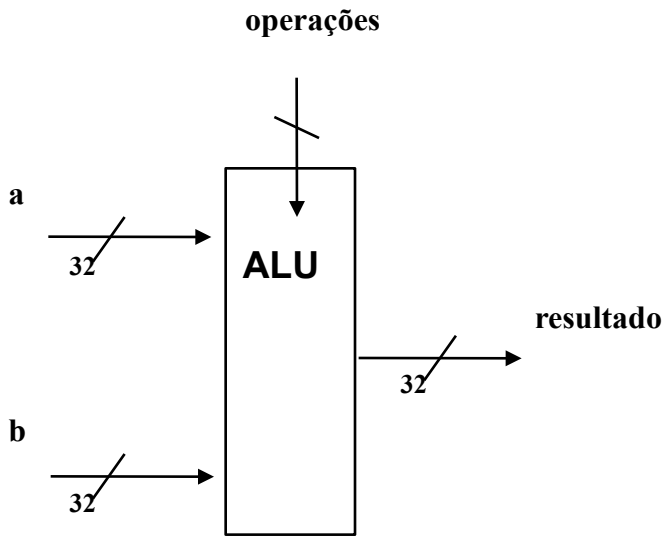
Aula 4 – Aritmética Computacional

Prof. Omar Paranaiba Vilela Neto

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

Aritmética

- **Onde nós estamos:**
  - **Desempenho (segundos, ciclos, instruções) e Instruções**
- **E agora:**
  - **Implementação da Arquitetura**



Números

- Bits são bits
  - convenções define relação entre bits e números
- Números Binários (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  - decimal: $0...2^n-1$

- mais complicado:
  - números são finitos (overflow)
  - frações e números reais
  - números negativos

    i.e., Arquiteturas não têm instrução subi; (addi pode somar um número negativo)

- Como nós representamos um número negativo?
  - i.e., qual padrão representará os números?

Representações Possíveis

- **Saídas**: balanço, números de zeros, facilidade das operações
- **Qual é a melhor? Porque?**

- Sinal Magnitude:
- 000 = +0
- 001 = +1
- 010 = +2
- 011 = +3
- 100 = -0
- 101 = -1
- 110 = -2

111 = -3

- Complemento de 1:
- 000 = +0
- 001 = +1
- 010 = +2
- 011 = +3
- 100 = -3
- 101 = -2
- 110 = -1

111 = -0

- Complemento de 2:
- 000 = +0
- 001 = +1
- 010 = +2
- 011 = +3
- 100 = -4
- 101 = -3
- 110 = -2

111 = -1

MIPS

- **32 bit complemento de dois:**

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

Operações em Complemento de dois

- Para negar um número em complemento de dois: inverter todos os bits e somar 1

Converter números de n bit em números com mais de n bits:

- 16 bit imediato convertido para 32 bits para aritmética
- copiar o mais significante bit (o bit de sinal) nos outros bits
- `0010 -> 0000 0010`
- `1010 -> 1111 1010`

Adição & Subtração

- **Como na escola primária (carry/borrow 1s)**
- `  0111      0111      0110`
- `+ 0110    - 0110    - 0101`

Exemplo: 7 – 6 = 7 + (–6)

+7: 0000 0000 … 0000 0111

–6: 1111 1111 … 1111 1010

+1: 0000 0000 … 0000 0001

- **Overflow (resultado maior que palavra do computador):**
  - **i.e., a soma de dois n-bit números não produz um número de n-bit**
  - `  0111`
  - `+ 0001` *note que o termo overflow é para o número,*
  - `1000` *ele não considera o carry "overflowed"*

Detectando Overflow

- Quando somamos um número positivo com um negativo não tem overflow
- Quando o sinal é o mesmo nas subtrações não tem overflow
- Overflow ocorre quando o valor afeta o sinal:
  - overflow quando somamos dois números positivos produzindo um negativo
  - ou, somando dois negativos dando um positivo
  - ou, subtraindo um negativo de um positivo dando um negativo
  - ou, subtraindo um positivo de um negativo dando um positivo

- Ocorre uma exceção (interrupção)
  - Controle salta para um endereço predefinido para a exceção
  - endereço Interrompido é salvo para possível retorno
- Detalhes baseado em software / linguagens

Revisão: Álgebra Booleana & Portas

- **Problema: Considere uma função lógica com três entradas: A, B e C**
  - **Saída D é verdade se pelo menos uma entrada for verdade**
  - **Saída E é verdade se duas entradas forem verdade**
  - **Saída F é verdade se todas as três forem verdade**

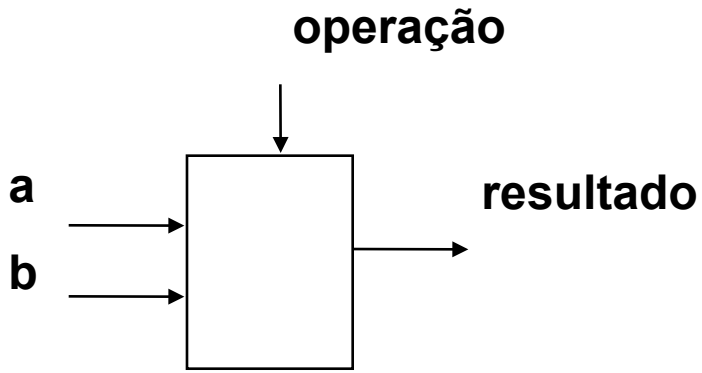
- **Mostre a tabela verdade para estas três funções.**

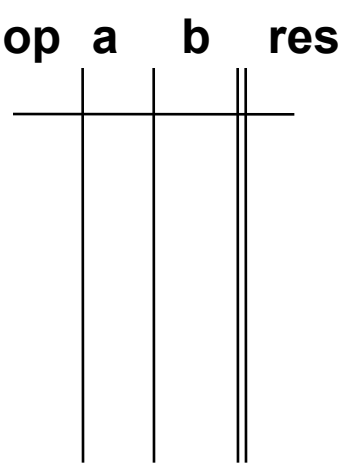
- **Mostre as equações Booleanas para as três funções.**

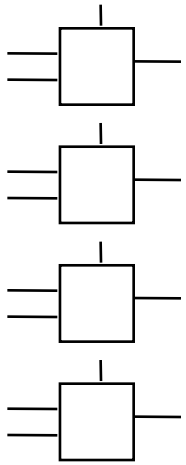
- **Mostre uma implementação com portas inversoras, AND e OR.**

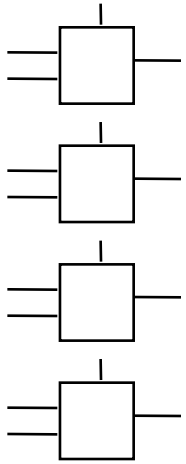
ALU (arithmetic logic unit)

- **Integrando uma ALU para supportar instruções `andi` e `ori`**
  - **1 bit ALU e usa-lo 32 vezes**





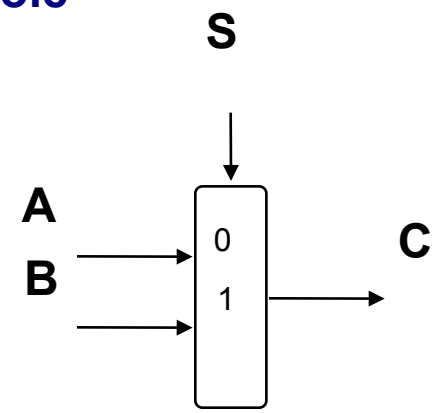




- **Implementação possível (soma de produtos):**

Revisão: Multiplexador

- Seleciona uma das entradas como saída, baseado em uma entrada de controle



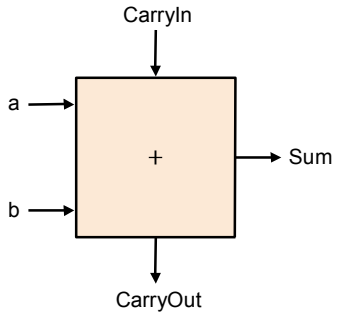
nota: Nós chamamos como mux 2-entrada mesmo tendo 3 entradas!

- Integrando nossa ALU usando um MUX:

Implementações Diferentes

- Não é fácil decidir o melhor caminho para a integração
  - Não desejamos várias entradas para uma porta
  - Não desejamos percorrer várias portas
  - Para nosso propósito, a facilidade de compreensão é importante

- Vermos 1-bit ALU para soma:

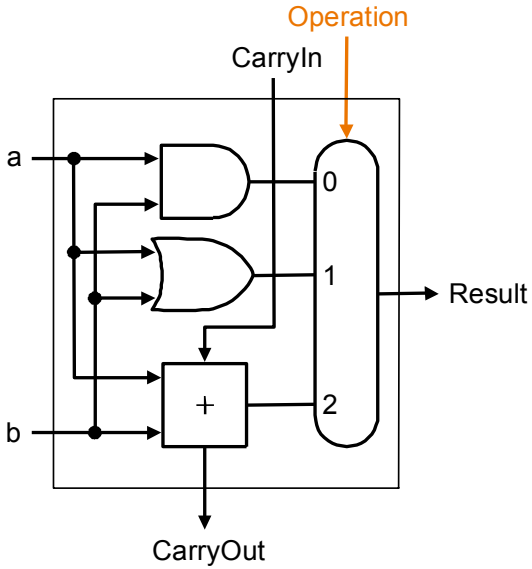


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

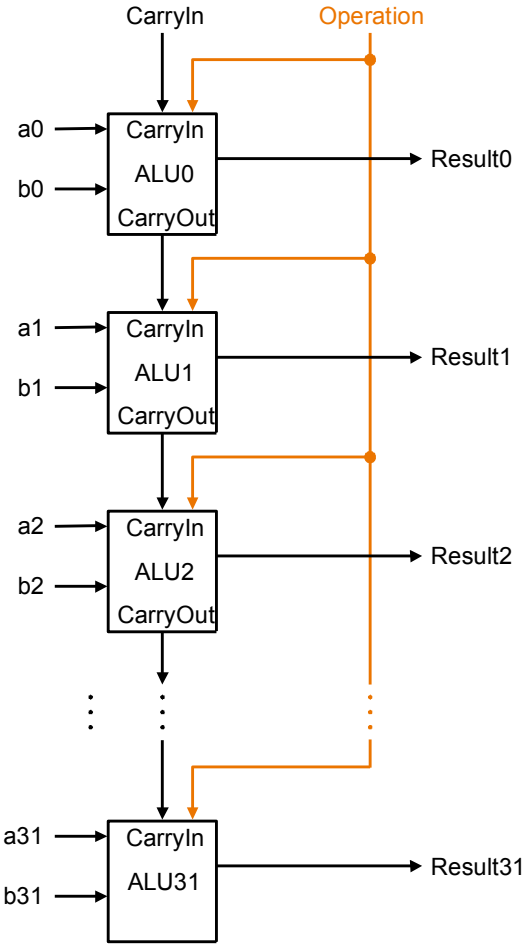
- 1º - Como poderíamos integrar 1-bit ALU para add, and e or?
- 2º - Como poderíamos integrar 32-bit ALU?

Integrando 32 bit ALU

1º



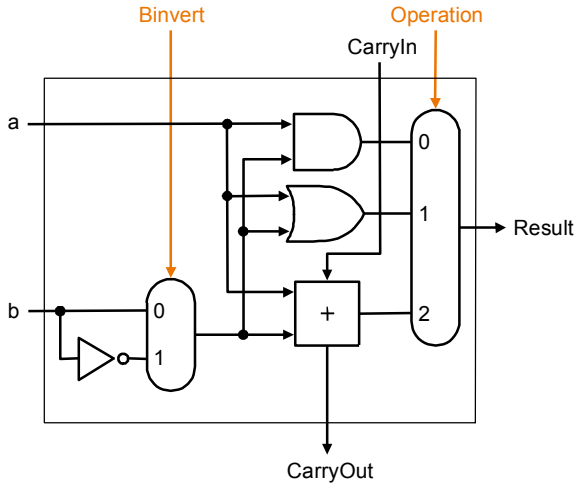
2º



Como subtrair (a – b) ?

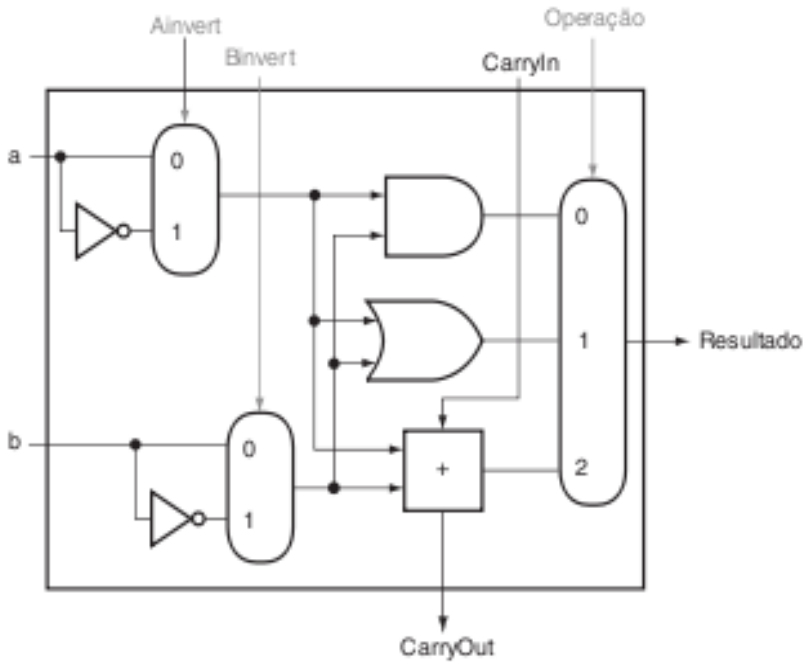
- **Complemento de dois: negar b e somar.**
- **Como fazer para negar?**

- **Uma solução:**



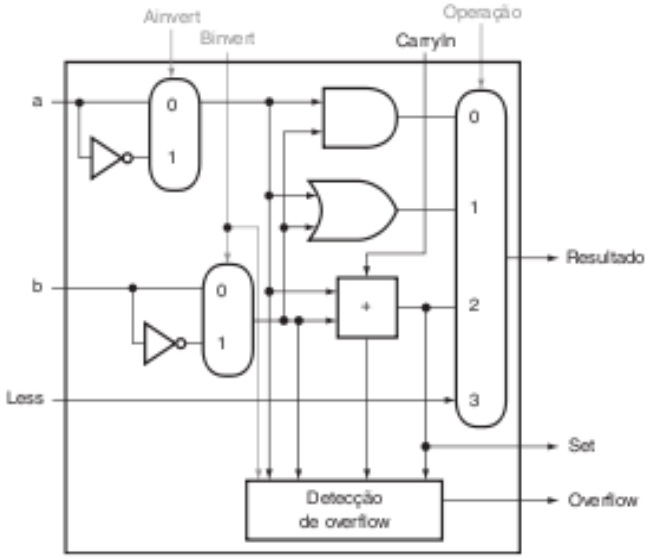
Acrescentando uma função NOR

Também podemos escolher inverter a. Como obtemos um "a NOR b"?

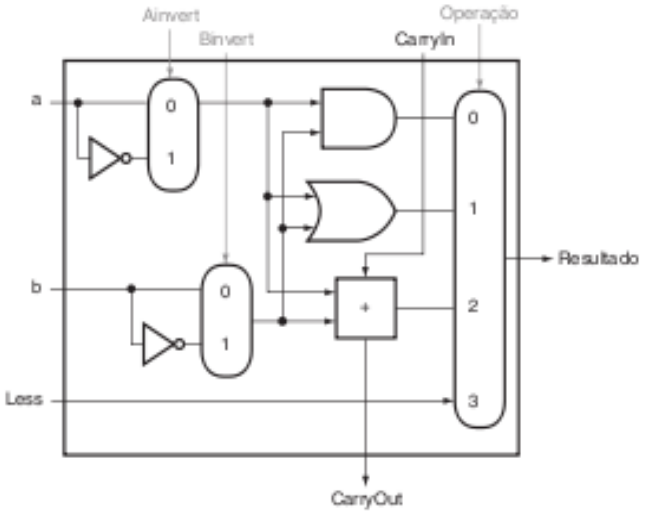


Suportando comparação

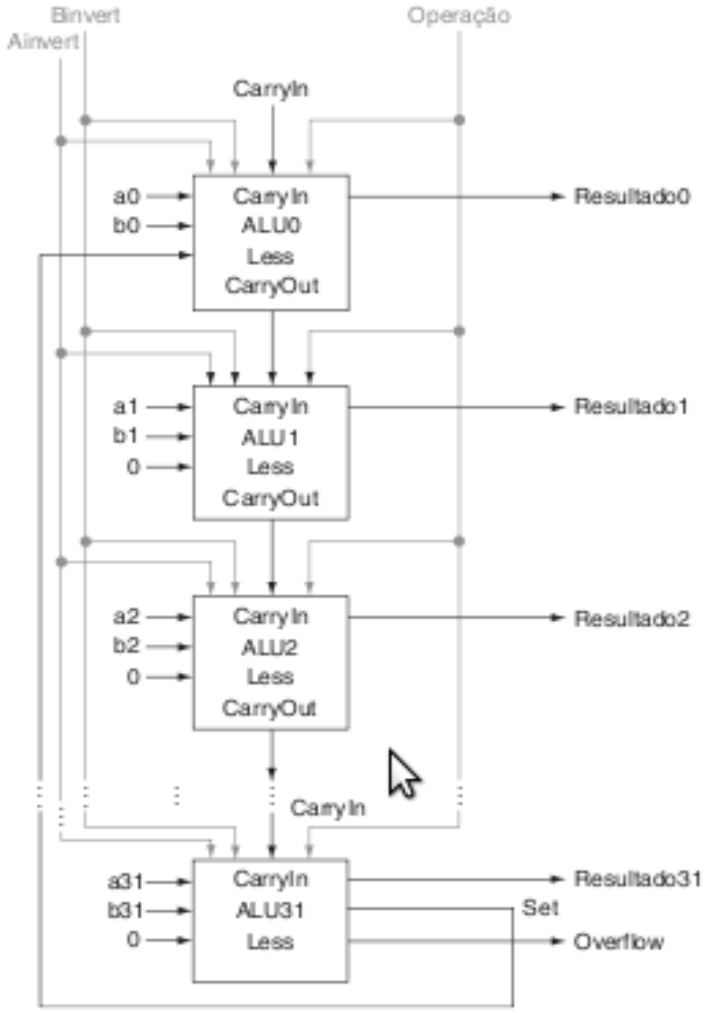
- **Podemos imaginar a idéia?**



Use esta ALU para o bit mais significativo



Todos os outros bits

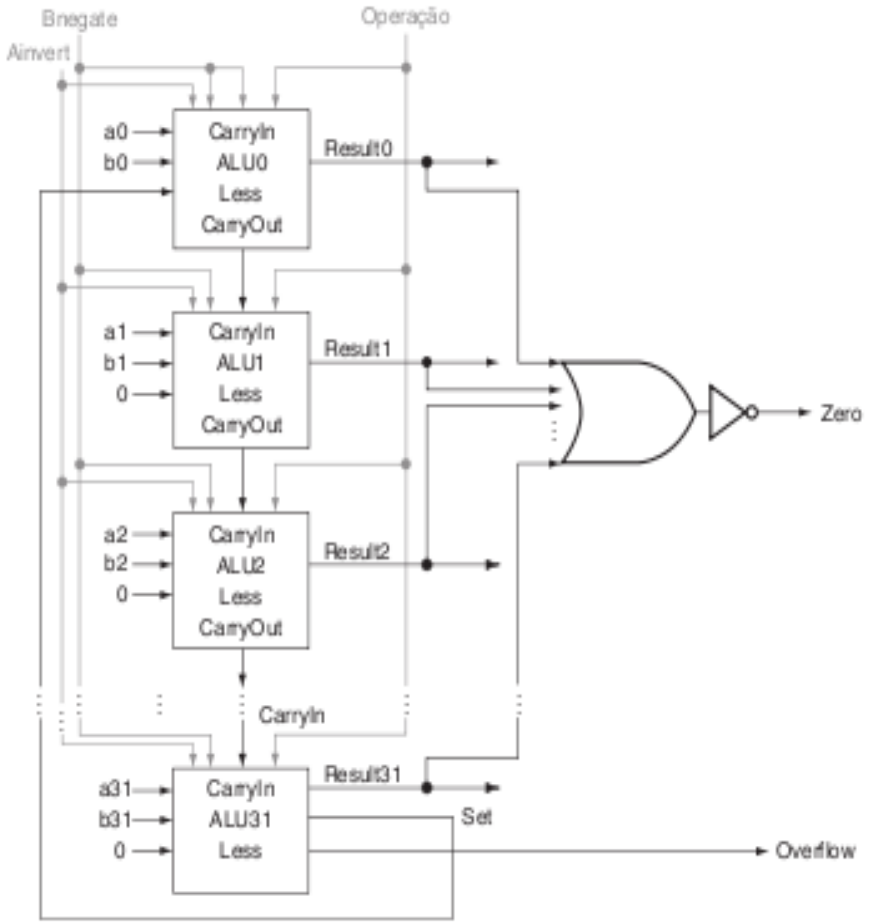

Binvert
Ainvert
Operação
CarryIn
a0
b0
CarryIn
ALU0
Less
CarryOut
Resultado0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Resultado1
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Resultado2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Resultado31
Set
Overflow

Teste de igualdade

- **Linhas de controle:**

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

- *Note: zero é um 1 quando o resultado é zero!*



Conclusão

- Podemos integrar uma ALU para suportar o conjunto de instruções
  - idéia chave: usar multiplexador para selecionar a saída
  - podemos fazer subtrações usando complemento de dois
  - podemos replicar 1-bit ALU para produzir 32-bit ALU
- Pontos importantes sobre hardware
  - todas as portas estão sempre trabalhando
  - a velocidade da porta é afetada pelo número de entradas da porta
  - a velocidade de um circuito é afetada pelo número de portas em série

- Nosso primeiro foco: compreensão; entretanto,
  - Mudanças na organização podem melhorar o desempenho
  - (similar a usar um algoritmo melhor em software)
  - vejamos dois exemplos para soma e multiplicação

Problema: somador "ripple carry" é lento

- Uma 32-bit ALU é tão rápida quanto uma 1-bit ALU?
- Existe mais de uma forma para somar?
  - dois extremos: ripple carry e soma de produtos (sum-of-products)

Você pode ver a propagação?

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

Somador “Carry-lookahead”

- Uma solução entre nossos dois extremos
- Motivação:
  - Se nós não conhecemos o valor do carry-in, como podemos fazer?
  - Poderíamos gerar o “carry”? $g_i = a_i b_i$
  - Poderíamos propagar o “carry”? $p_i = a_i + b_i$
- Did we get rid of the ripple?

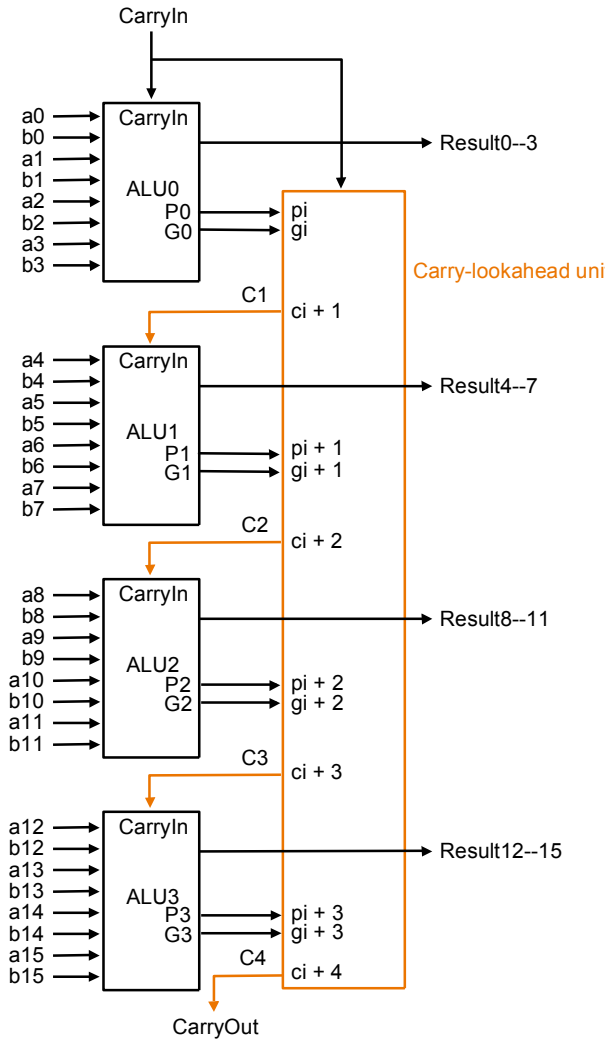
$c_1 = g_0 + p_0 c_0$

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

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

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

Princípio para integrar grandes somadores



Multiplicação

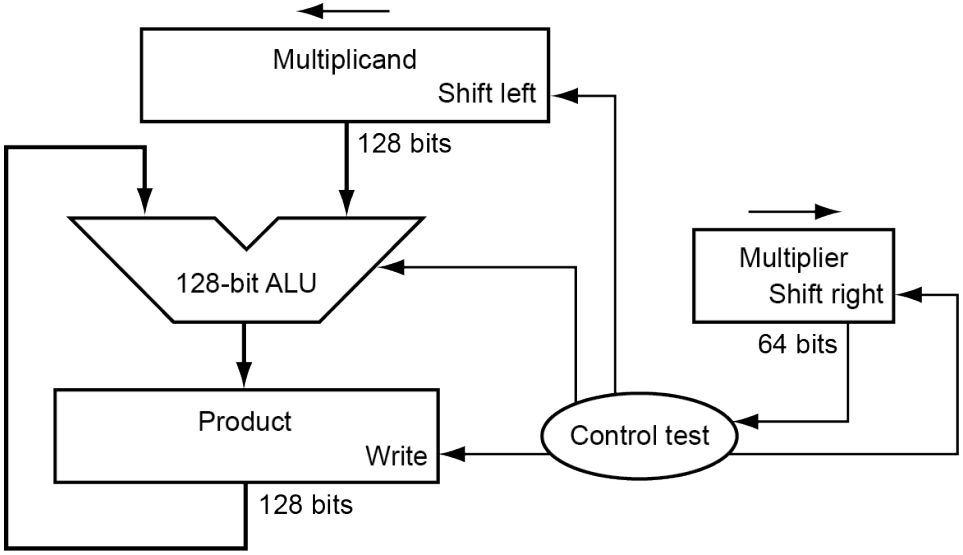
- **Mais complicado que soma**
  - **Resolvida através de somas e deslocamentos**
- **Mais tempo e mais área de chip**
- **Veremos 3 versões baseadas em algoritmos que aprendemos na escola**

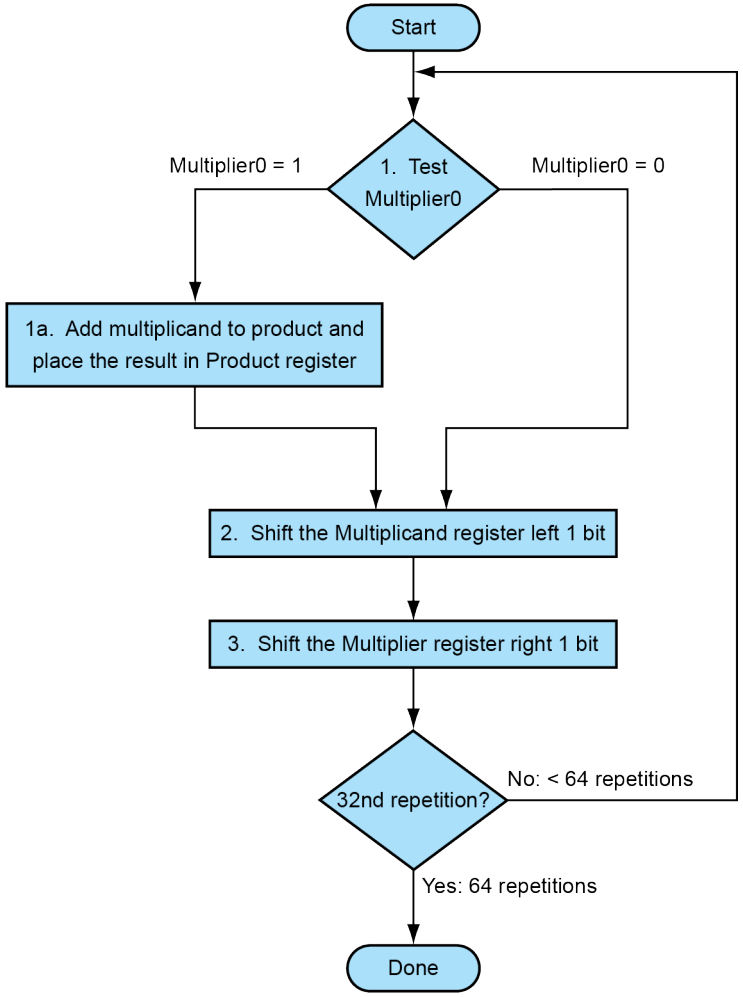
- `0010` **(multiplicando)**
- `x 1011` **(multiplicador)**

- **Números negativos: converter e multiplicar**
  - **Há técnicas melhores**

Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}

Multiplicação: Implementação





multiplicand

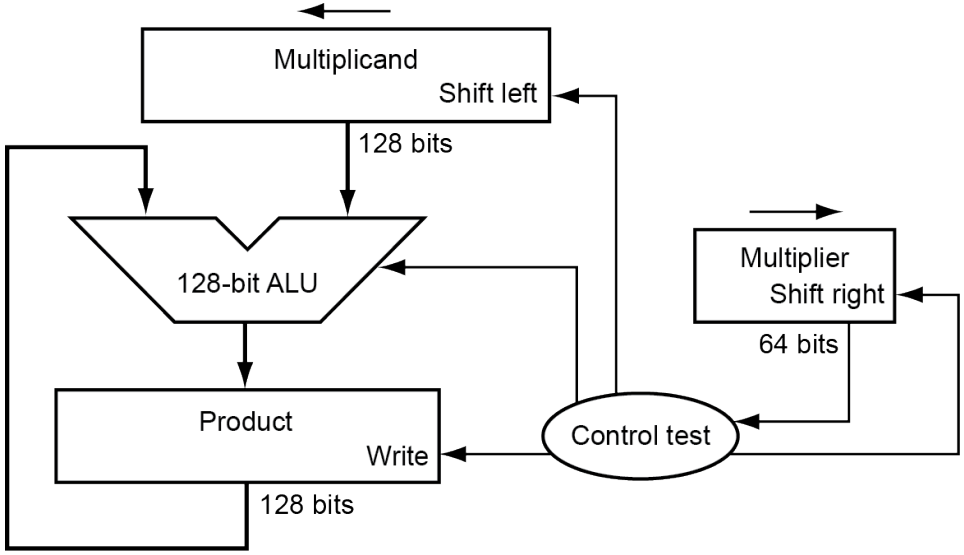
multiplier

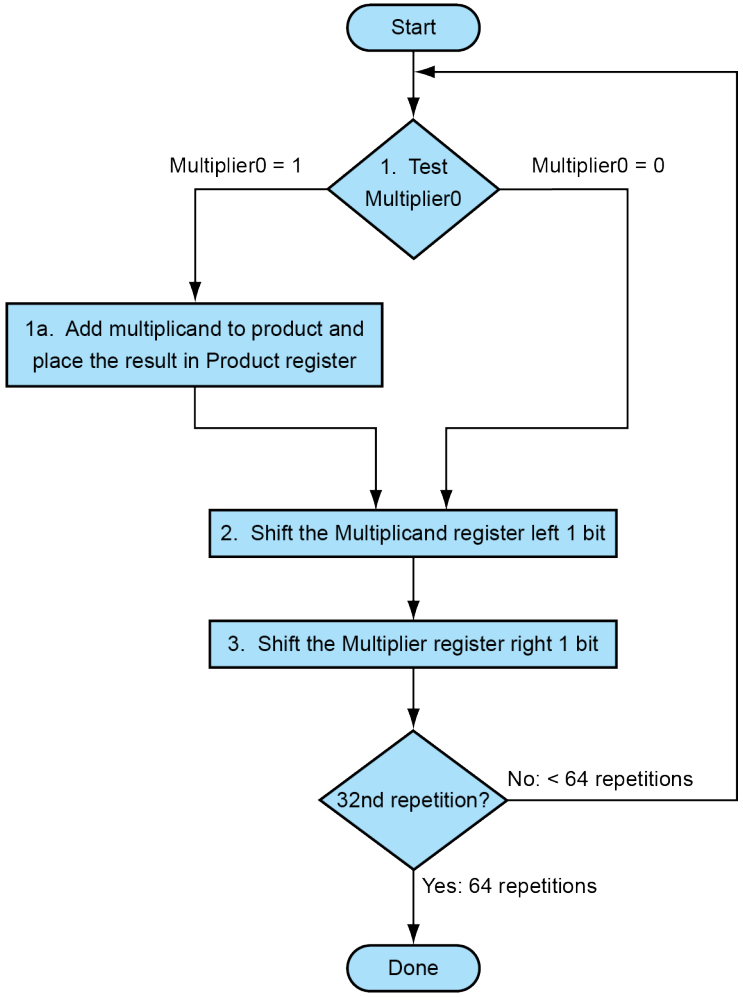
1000

× 1001

product

Multiplicação: Implementação





multiplicand → 1000

multiplier → × 1001

1000

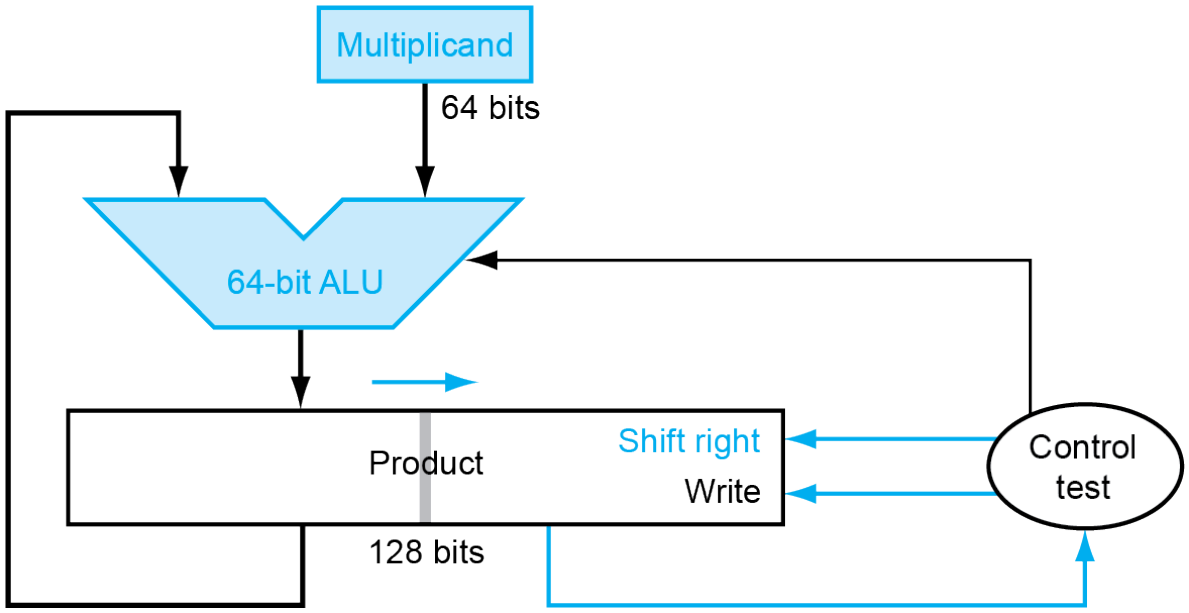
0000

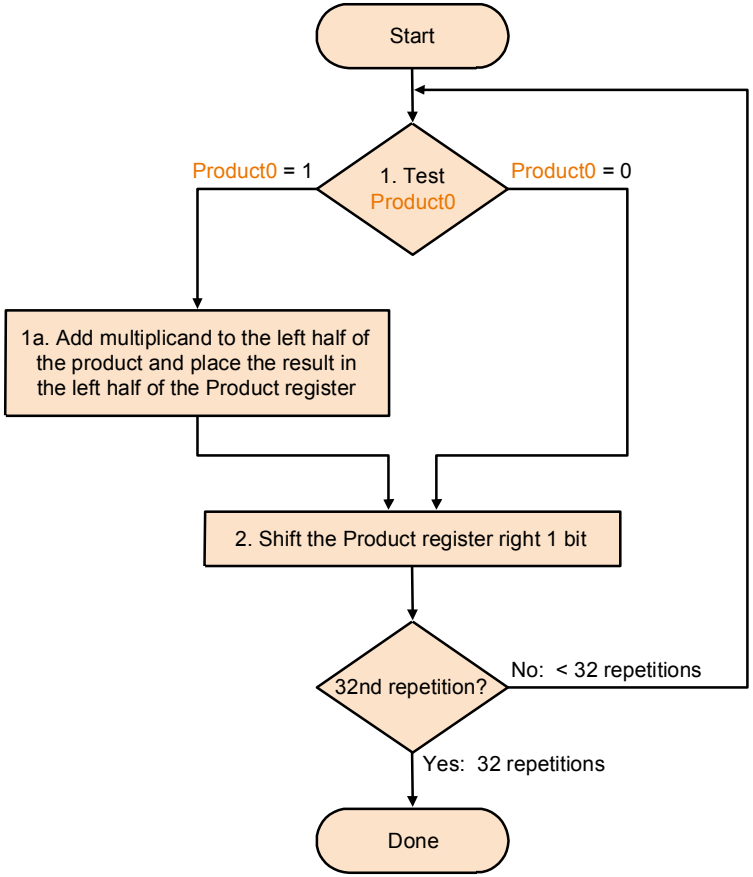
0000

1000

product → 1001000

Versão otimizada

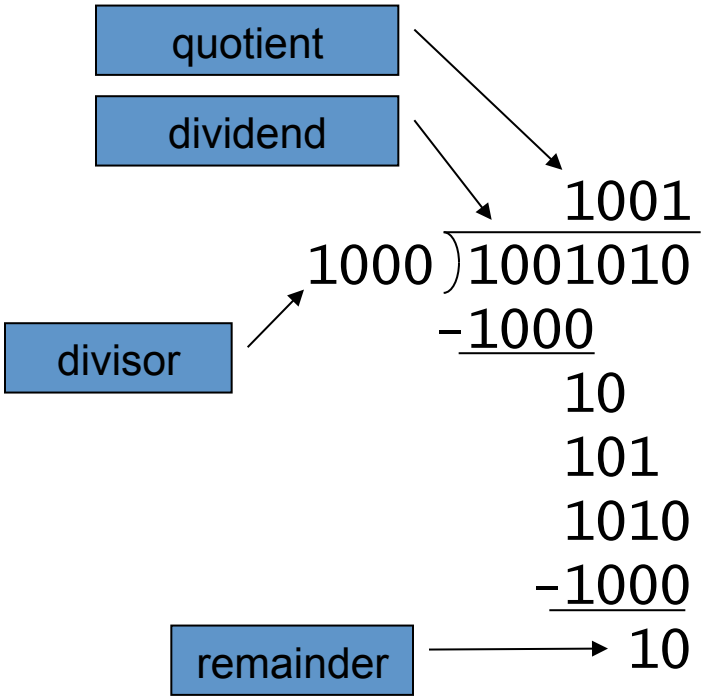




Multiplicação no RISC-V

- Quatro instruções de multiplicação:
  - mul: multiply
    - Retorna os 64 bits do produto
  - mulh: multiply high
    - Retorna os 64 bits “mais altos” do produto, assumindo que os operandos têm sinal
  - mulhu: multiply high unsigned
    - Retorna os 64 bits “mais altos” do produto, assumindo que os operandos não têm sinal
  - mulhsu: multiply high signed/unsigned
    - Retorna os 64 bits “mais altos” do produto, assumindo que um operando tem sinal e o outro não
  - Use o resultado de **mulh** para verificar 64-bit overflow

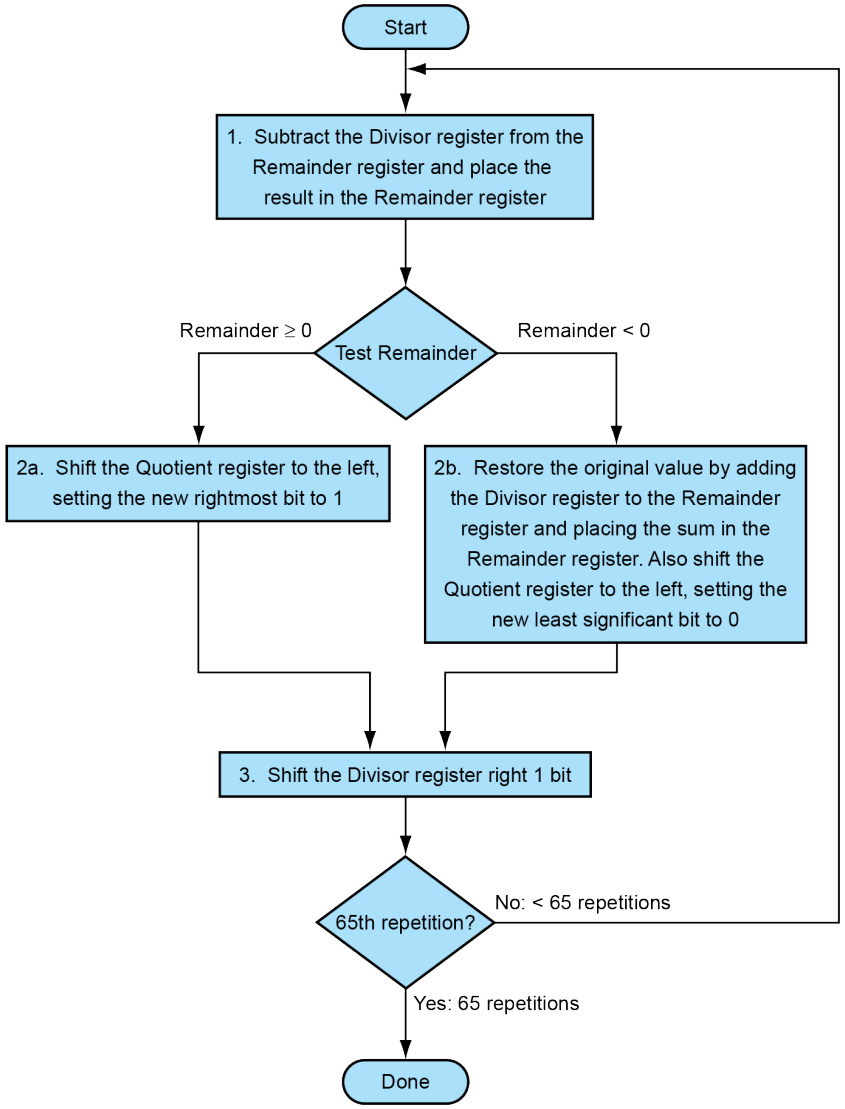
Divisão

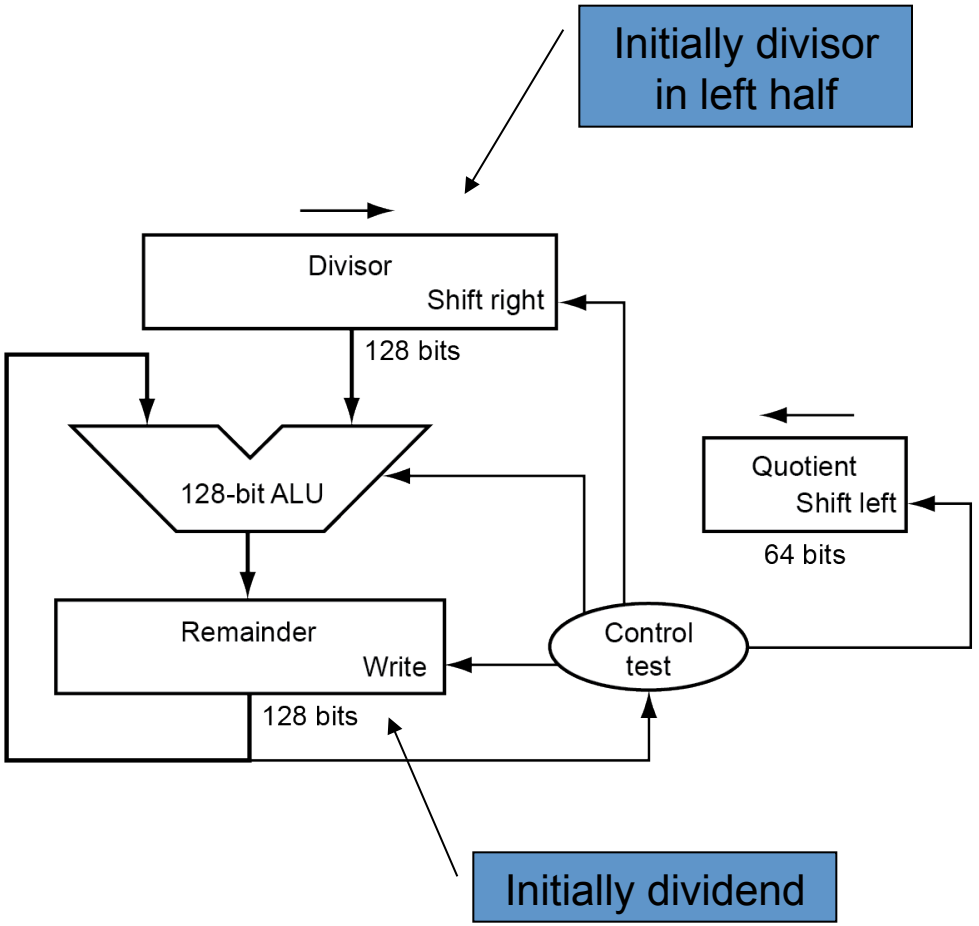


Exemplo: 74 / 9

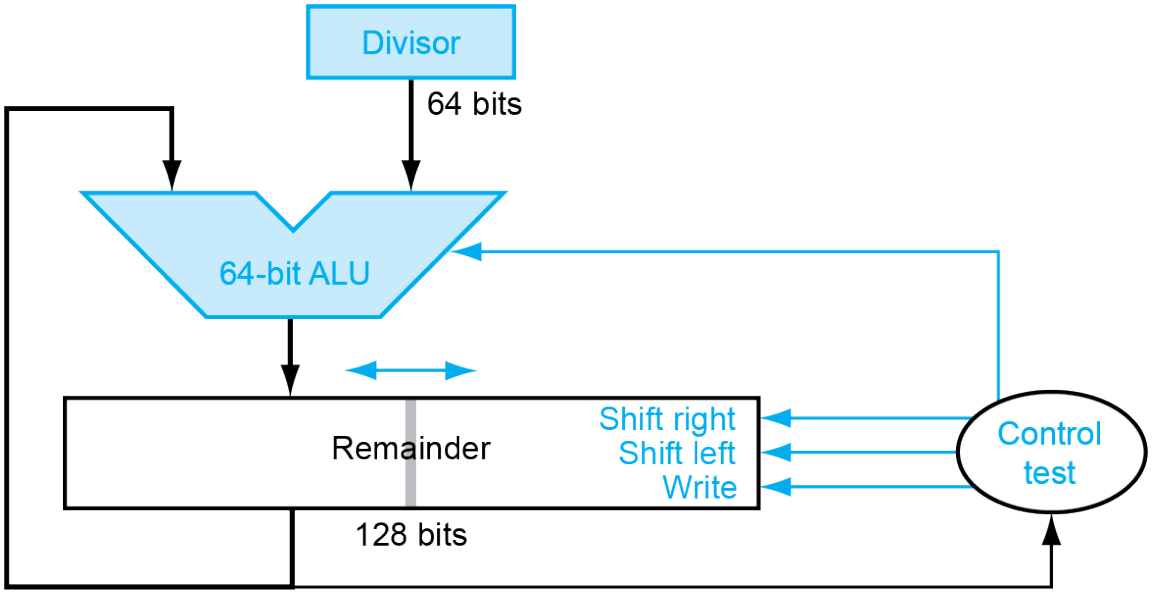
1001010 / 1001

Divisão


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
65th repetition?
No: < 65 repetitions
Yes: 65 repetitions
Done


Initially divisor in left half
Divisor
Shift right
128 bits
128-bit ALU
Quotient
Shift left
64 bits
Remainder
Write
Control test
128 bits
Initially dividend

Divisão Otimizada



Pode usar o mesmo Hardware da multiplicação

Divisão no RISC-V

- Quatro Instruções:
  - div, rem: signed divide, remainder
  - divu, remu: unsigned divide, remainder

- Overflow e divisão por zero não geram erros