

# Aula 3: Aritmética Computacional - Parte I

Professor(a): Virgínia Fernandes Mota  
<http://www.dcc.ufmg.br/~virginiaferm>

OCS (TEORIA) - SETOR DE INFORMÁTICA



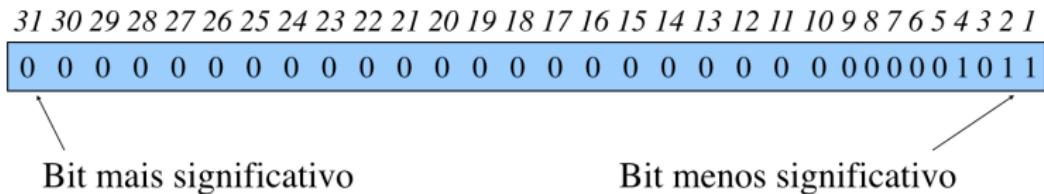
- Introdução
- Números com sinal e sem sinal
- Adição e Subtração
- Multiplicação
- Divisão

- Representação dos números
  - Como números negativos são representados?
  - Qual o maior número que pode ser representado em uma word?
  - O que ocorre se o resultado de uma operação for um número que não pode ser representado?
  - Como frações / números reais são representados?
- Algoritmos aritméticos
  - E o Hardware que acompanha esses algoritmos
  - Implicações para o conjunto de instruções

## Números com Sinal e sem Sinal

# Números com Sinal e sem Sinal

- Em qualquer base numérica, o valor do  $i$ -ésimo dígito  $d$  é
  - $d \times \text{base}^i$
  - $i$  começa em zero e aumenta da direita para a esquerda
  - Ex:  $1011_2$   
 $(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 8 + 0 + 2 + 1 = 11_{10}$



- Word MIPS com 32 bits de largura
  - Podemos representar  $2^{32}$  padrões diferentes de 32 bits
- Hardware projetado para somar, subtrair, multiplicar e dividir esses padrões de bits
  - Se número que é o resultado correto dessas operações não puder ser representado com 32 bits, dizemos que houve um **overflow**
  - SO e programa decidem o que fazer quando isso ocorre

# Números com Sinal e sem Sinal

- Representação precisa fazer distinção entre positivos e negativos. Representações possíveis:

Sinal e magnitude	Complemento a um	Complemento a dois
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

- Qual é o melhor? Por quê?

- Como representar um número negativo em complemento a dois?

- Representar o número em binário
- Negar o número (inverter os bits e somar 1)
- Exemplo: -2 usando 32 bits?

2 = 0000 0000 0000 0000 0000 0000 0010

Invertendo: 1111 1111 1111 1111 1111 1111 1111 1101

Somando 1: 1111 1111 1111 1111 1111 1111 1111 1110 = -2

# Números com Sinal e sem Sinal

- Representação por complemento a dois adotada

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{bin} = 0_{dec}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} = 1_{dec}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{bin} = 2_{dec}$$

...

...

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{bin} = 2.147.483.645_{dec}$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{bin} = 2.147.483.646_{dec}$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{bin} = 2.147.483.647_{dec}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{bin} = -2.147.483.648_{dec}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{bin} = -2.147.483.647_{dec}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{bin} = -2.147.483.646_{dec}$$

...

...

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{bin} = -3_{dec}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{bin} = -2_{dec}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{bin} = -1_{dec}$$

*maxint*

*minint*

- Complemento a dois tem vantagem de que todos os números negativos possuem 1 no bit mais significativo
  - Hardware só precisa testar esse bit para ver se um número é positivo ou negativo: Bit de sinal
- Com ou sem sinal também se aplica aos loads
  - Load byte (lb) trata byte como número com sinal: Copia o sinal para preencher os 24 bits mais à esquerda do registrador (extensão de sinal)
  - Load byte unsigned (lbu) trabalha com inteiros sem sinal: Preenche com 0s os bits à esquerda dos dados
  - O mesmo ocorre com load half (com sinal) e load half unsigned (sem sinal)

- Programas podem querer lidar com números que só podem ser positivos: Declarados como `unsigned` em C.
  - endereços de memória, contadores (não existem esses dados negativos)
- Instrução de comparação precisa lidar com essas duas classes de números
  - Em números com sinal, padrão de bits com 1 no bit mais significativo representa um número negativo.
  - É menor do que qualquer número positivo.
  - Mas se número sem sinal, 1 no seu bit mais significativo faz com que ele seja maior do que qualquer outro número que tenha 0 neste bit.

- Duas versões para comparação!
- Set on less than (slt) e set on less than immediate (slti) trabalham com inteiros com sinal
- Set on less than unsigned (sltu) e set on less than immediate unsigned (sltiu) trabalham com inteiros sem sinal

- Registrador \$s0 contém  
1111 1111 1111 1111 1111 1111 1111 1111
- Registrador \$s1 contém  
0000 0000 0000 0000 0000 0000 0000 0001
- Qual o resultado de:  
`slt $t0, $s0, $s1 # registrador $t0: $s0 < $s1?`  
`sltu $t1, $s0, $s1 # registrador $t0: $s0 < $s1?`

- Registrador \$s0 contém  
1111 1111 1111 1111 1111 1111 1111 1111
- Registrador \$s1 contém  
0000 0000 0000 0000 0000 0000 0000 0001
- Qual o resultado de:  
`slt $t0, $s0, $s1 # registrador $t0: $s0 < $s1? 1`  
`sltu $t1, $s0, $s1 # registrador $t0: $s0 < $s1? 0`

- Podemos utilizar um atalho para verificar se  $0 \leq x < y$ 
  - Por exemplo, se o índice de um vetor está dentro de seus limites
- Idéia chave: inteiros negativos na notação em complemento a dois se parecem com números grandes na notação sem sinal
  - Comparação sem sinal de  $x < y$  também verifica se  $x$  é negativo

# Adição e Subtração

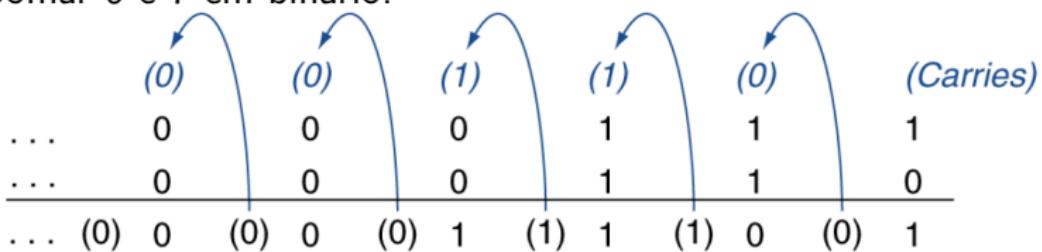
Adição e Subtração

# Adição e Subtração

- **Adição:** Dígitos somados bit a bit, da direita para a esquerda, com carries ("vai-um") sendo passados para o próximo dígito à esquerda.
- **Subtração:** Utiliza a adição - operando apropriado é negado antes de ser somado.

# Adição e Subtração

- Somar 6 e 7 em binário.



- Subtrair 6 de 7 em binário

$$+7: \quad 0000 \ 0000 \dots 0000 \ 0111$$

$$-6: \quad \underline{1111 \ 1111 \dots 1111 \ 1010}$$

$$+1: \quad 0000 \ 0000 \dots 0000 \ 0001$$

- Quando ocorre **overflow** na adição?
- Quando se somam dois números positivos e a soma é negativa
- Quando se somam dois números negativos e a soma é positiva
- Motivo: resultado precisa de 33 bits para ser expresso
  - Bit de sinal está sendo definido com o valor do resultado no lugar do sinal apropriado do resultado
  - Ocorreu um carry no bit de sinal

- Quando ocorre **overflow** na subtração?
  - Quando se subtrai um número negativo de um número positivo e o resultado é negativo
  - $A - B$ , quando  $B$  negativo, equivale a  $A + B$
  - Quando se subtrai um número positivo de um número negativo e o resultado é positivo
- E como detectar overflow para inteiros sem sinal?
  - Inteiros sem sinal normalmente usados para endereços de memória
  - Overflow ignorado
  - Implica que projetista precisa oferecer modo de overflow ser ou não ignorado

- Duas classes de instruções aritméticas
  - add, addi e sub: Causam exceções no overflow
  - addu, addiu, subu: Não causam exceções no overflow
- Mas o que é uma exceção?
  - Também chamada interrupção em muitos computadores
  - Evento não planejado que interrompe a execução do programa
    - Salto para rotina de tratamento da exceção

## Multiplicação

# Multiplicação

- Vamos lembrar o algoritmo da multiplicação!
- Pegar dígitos do multiplicador um a um, da direita para a esquerda, calculando a multiplicação deste pelo multiplicando
- Produto intermediário deslocado um dígito para esquerda dos produtos intermediários anteriores
- Resultado final é a soma dos produtos intermediários

Exemplo (método tradicional):

$$\begin{array}{r} 1000 \\ \times 1001 \\ \hline 1000 \\ 0000 \\ 0000 \\ \hline 1001000 \end{array}$$

# Multiplicação

- Como multiplicação de binários envolve somente 0s e 1s, cada etapa da multiplicação é simplificada!
- Basta colocar uma cópia do multiplicando se o dígito do multiplicador for 1 ou Colocar 0s se o dígito for zero
- Número de bits no produto maior que o número no multiplicando ou no multiplicador
  - Ignorando o bit de sinal, tamanho da multiplicação de  $n$  bits por  $m$  bits é  $n+m$  bits
  - Precisamos lidar com overflow

- Vamos por enquanto supor que estamos multiplicando apenas números positivos!
- Versão seqüencial do algoritmo imita algoritmo tradicional.
- Registrador multiplicando e registrador produto tem 64 bits
  - Produto iniciado com zero
  - Por que 64 bits? Durante 32 etapas, um multiplicando de 32 bits moveria 32 bits para a esquerda

# Multiplicação

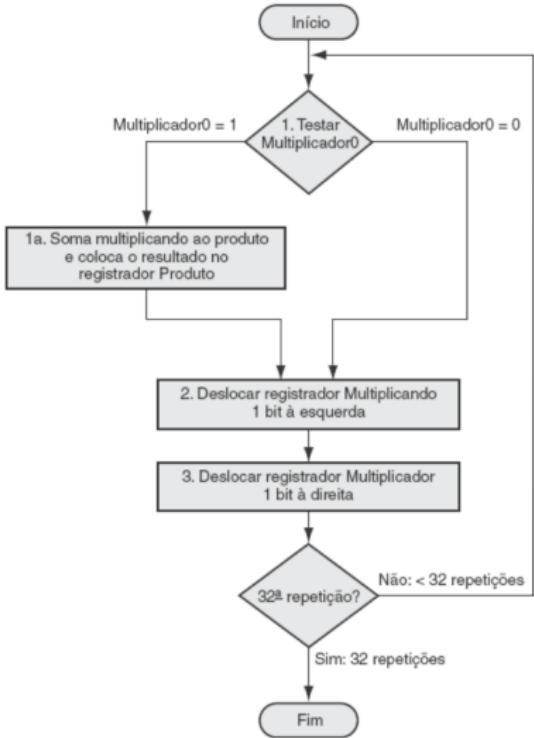
- Exemplo (1000 X 1001)

- 1<sup>º</sup> repetição:

- Multiplicador0 = 1? Sim
      - Produto += 1000
    - Multiplicando: 10000
    - Multiplicador: 100
    - 4<sup>º</sup> repetição? Não

- 2<sup>º</sup> repetição:

- Multiplicador0 = 1? Não
      - Multiplicando: 100000
      - Multiplicador: 10
      - 4<sup>º</sup> repetição? Não



# Multiplicação

- Exemplo (1000 X 1001)

- 3<sup>º</sup> repetição:

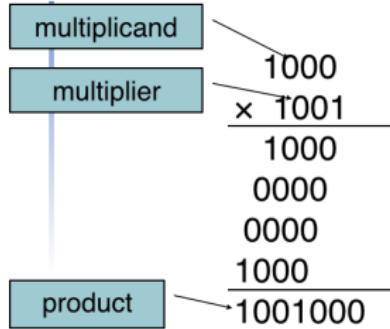
- Multiplicador0 = 1? Não
    - Multiplicando: 1000000
    - Multiplicador: 1
    - 4<sup>º</sup> repetição? Não

- 4<sup>º</sup> repetição:

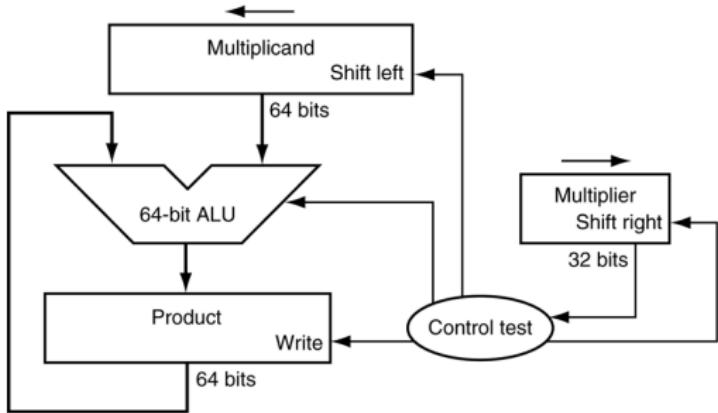
- Multiplicador0 = 1? Sim
      - Produto += 1000000
    - Multiplicando: 10000000
    - Multiplicador: 0
    - 4<sup>º</sup> repetição? Sim. Fim



# Multiplicação



Length of product is  
the sum of operand  
lengths



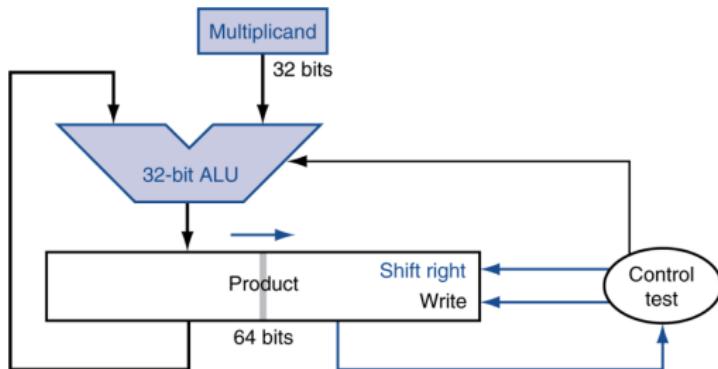
# Multiplicação

- **Problema:** Multiplicação tem três etapas por bit
  - Determinar se multiplicando é somado ao registrador produto
  - Deslocamento à esquerda do multiplicando
  - Deslocamento à direita para multiplicador
- Repetidas 32 vezes
- Se cada etapa demora 1 ciclo de clock, algoritmo exigiria quase **96 ciclos de clock!**



# Multiplicação

- Como melhorar desempenho?
- Realizar operações em paralelo!
- Multiplicando E multiplicador deslocados ENQUANTO multiplicando somado ao produto se bit do multiplicador for 1



# Multiplicação

- Exemplo:  $1000 \times 1001$ 
  - Produto: 0000 1001
  - Produto0 = 1? Sim
    - Soma 0000 (4 bits mais significativos de produto com 1000) = 1000
    - Novo produto: 1000 1001
    - Desloca produto à direita: 0100 0100
  - Produto0 = 1? Não
    - Desloca produto à direita: 0010 0010
- Produto0 = 1? Não
  - Desloca produto à direita: 0001 0001
- Produto0 = 1? Sim
  - Soma 0001 (4 bits mais significativos de produto com 1000) = 1001
  - Novo produto: 1001 0001
  - Desloca produto à direita: 0100 1000
- Fim (quarto deslocamento)

- Outra alternativa de multiplicação mais rápida: uso de um somador para cada bit do multiplicador
- Multiplicação com sinal
  - Converter multiplicando e multiplicador para números positivos
  - Executar o algoritmo por 31 iterações: Sinal fora do cálculo
  - Verificar sinais originais
    - Produto negativo se sinais forem diferentes: Converte resultado

# Multiplicação

- No MIPS, par de registradores (Hi e Lo) armazena produto
- Instruções mult e multu para operações de multiplicação com e sem sinal
- Instrução mflo (move from lo) e mfhi (move from hi) usados para acessar resultado

# Divisão

Divisão

- Algoritmo tradicional:

- 1001010 por 1000

1001010

-1000

---

1 conseguimos dividir? Sim quociente: 1

10 conseguimos dividir? Não quociente: 10

101 conseguimos dividir? Não quociente: 100

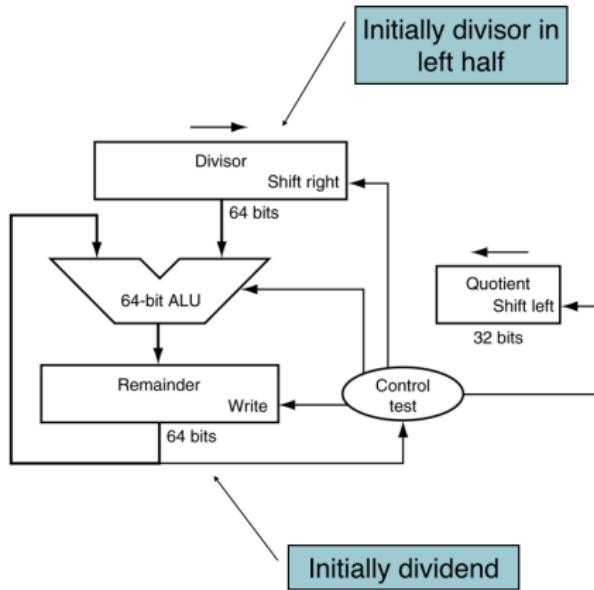
1010

- 1000 conseguimos dividir? Sim quociente: 1001

---

10 resto

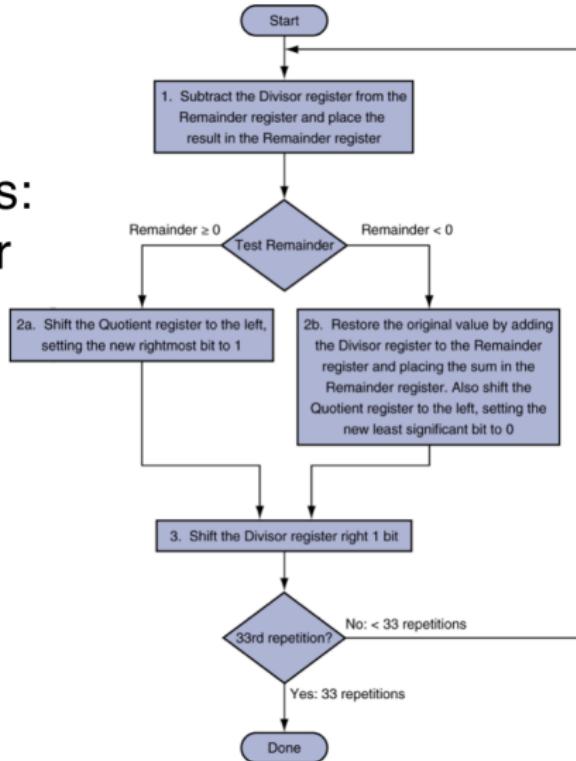
- Vamos supor que dividendo e divisor sejam positivos, portanto quociente e resto são não-negativos



- Exemplo com 4 bits:  
divisão de 0111 por  
0010

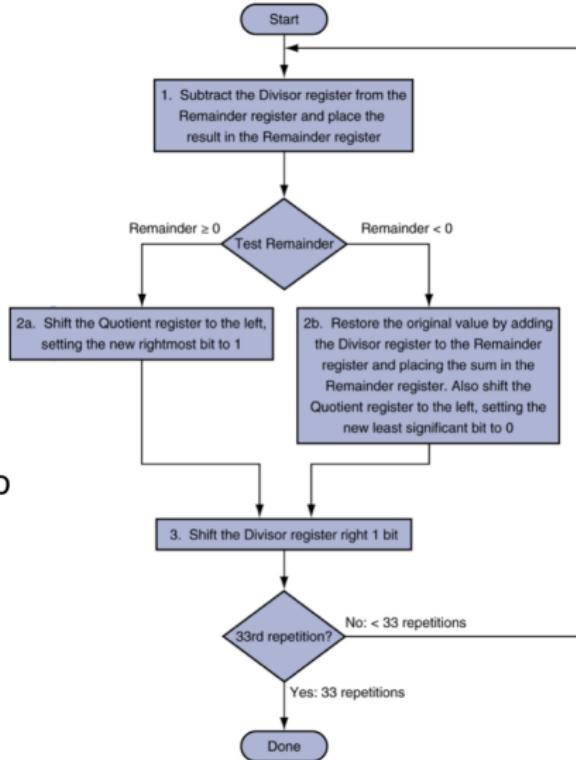
- Registradores:

- Quociente: 0000
    - Divisor: 0010 0000
    - Resto: 0000 0111



# Divisão

- Quociente: 0000
- Divisor: 0010 0000
- Resto: 0000 0111
- Iteração 1:
  - Resto -= Div:
    - Resto: 11100111
  - Resto < 0: restaura resto e desloca quociente
    - Resto: 0000 0111
    - Quociente: 0000
  - Desloca div direita:
    - Divisor: 0001 0000

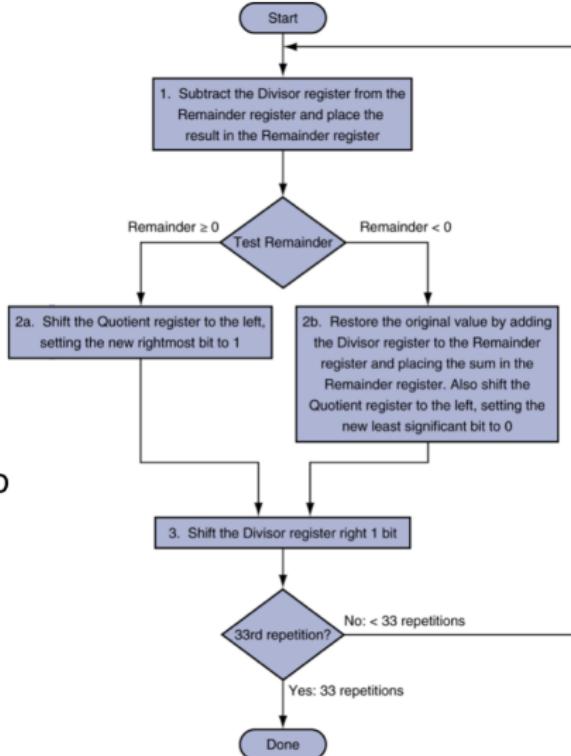


# Divisão

- Quociente: 0000
- Divisor: 0001 0000
- Resto: 0000 0111

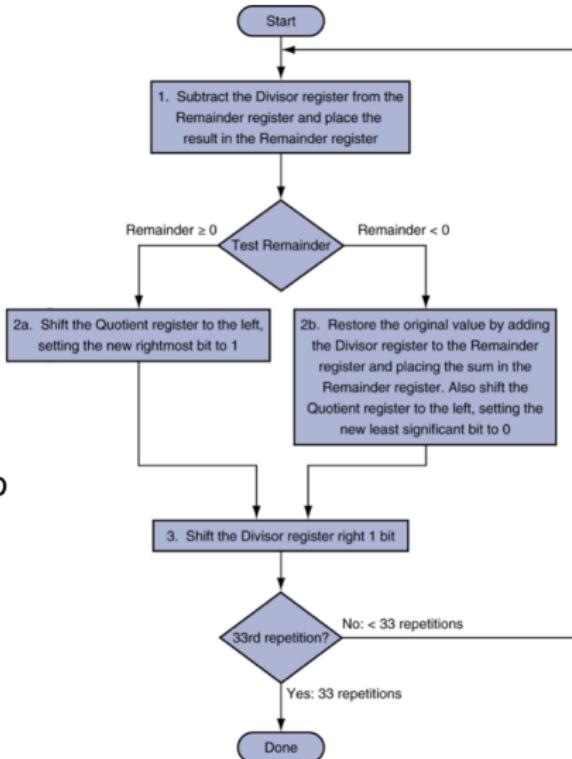
## ▪ Iteração 2:

- Resto -= Div:
  - Resto: 1111 0111
- Resto < 0: restaura resto e desloca quociente
  - Resto: 0000 0111
  - Quociente: 0000
- Desloca div direita:
  - Divisor: 0000 1000



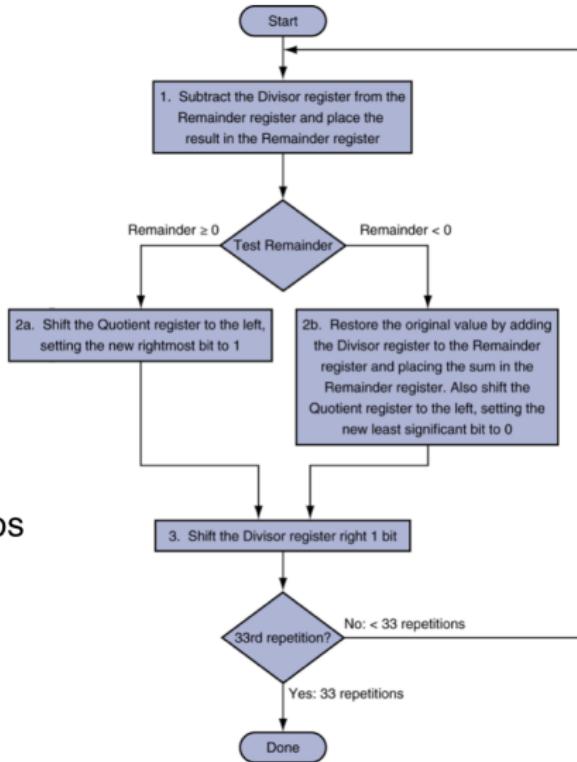
# Divisão

- Quociente: 0000
- Divisor: 0000 1000
- Resto: 0000 0111
- Iteração 3:
  - Resto -= Div:
    - Resto: 1111 1111
  - Resto < 0: restaura resto e desloca quociente
    - Resto: 0000 0111
    - Quociente: 0000
  - Desloca div direita:
    - Divisor: 0000 0100



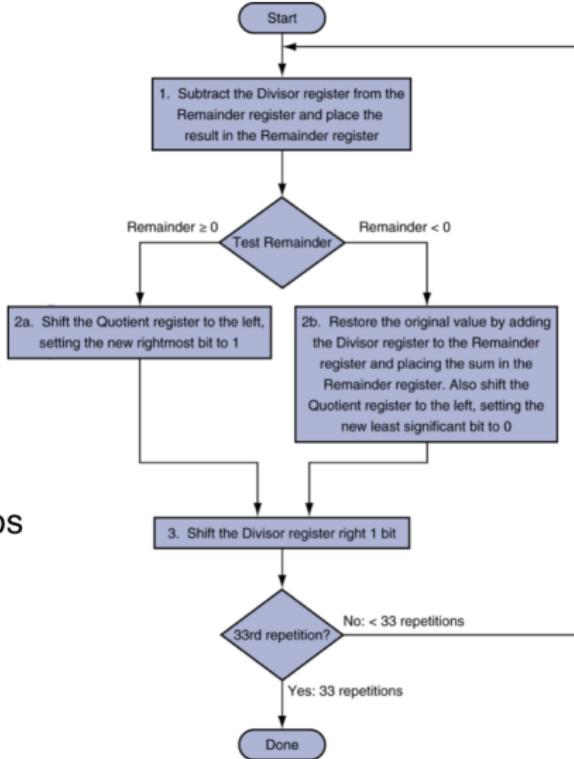
# Divisão

- Quociente: 0000
- Divisor: 0000 0100
- Resto: 0000 0111
- Iteração 4:
  - Resto -= Div:
    - Resto: 0000 0011
  - Resto > 0: desloca quociente, 1 no bit menos significativo
    - Quociente: 0001
  - Desloca div direita:
    - Divisor: 0000 0010

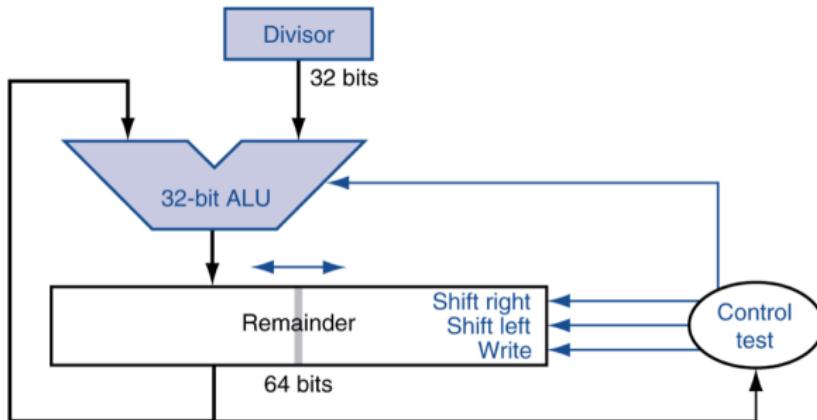


# Divisão

- Quociente: 0001
- Divisor: 0000 0010
- Resto: 0000 0011
- Iteração 5:
  - Resto -= Div:
    - Resto: 0000 0001
  - Resto > 0: desloca quociente, 1 no bit menos significativo
    - Quociente: 0011
  - Desloca div direita:
    - Divisor: 0000 0001



- Hardware mais rápido com deslocamento dos operandos e do quociente no mesmo tempo da subtração → Semelhante à multiplicação



- Divisão com sinal
  - Lembrar sinais do dividendo e divisor
  - Se diferentes, negar quociente
- Divisão no MIPS
  - Instruções div e divu para divisão com e sem sinal
  - Hi contém o resto
  - Lo o quociente

## Aritmética Computacional Parte II