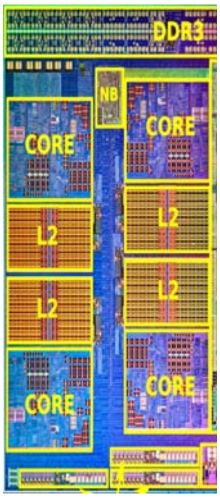
0273359 – Arquitetura e Organização de Computadores 1



Fonte: http://www.techspot.com/article/904-history-of-the-personal-computer-part-5/

ULA / Multiplicação - Divisão

Luciano de Oliveira Neris
luciano@dc.ufscar.br

Adaptado de slides do prof. Marcio Merino Fernandes

Departamento de Computação
Universidade Federal de São Carlos

2

Multiplicação

Multiplicação

- Mais complicado que soma
  - Realizado usando deslocamentos e somas sucessivas
- Operação menos frequente do que adição e subtração.
- Mais lenta e ocupa maior área de implementação em hardware.

- Esquema básico: 3 versões baseados no algoritmo que utilizamos desde o ensino fundamental.
- Ex: Efetuar a seguinte multiplicação de números decimais:

```
     1000   (multiplicando)
___x_1001   (multiplicador)
```

3

Exemplo

```
Multiplicando          1000ten
Multiplicador        x 1001ten
                   --------------
                         1000
                        0000
                       0000
                      1000
                   --------------
Produto               1001000ten
```

4

Multiplicação

Multiplicando 1000_{ten}

Multiplicador $\times\ 1001_{ten}$

```
   --------------
        1000
       0000
      0000
     1000
   --------------
```

Produto 1001000_{ten}

Em cada passo:

- multiplicando é deslocado p/ esquerda (shift-left)
- o próximo bit do multiplicador é verificado
- se for igual a 1, o multiplicando deslocado é adicionado ao produto.

5

Multiplicação

Multiplicando 1000_{two}

Multiplicador $\times\ 1001_{two}$

```
   --------------
        1000
       0000
      0000
     1000
   --------------
```

Produto 1001000_{two}

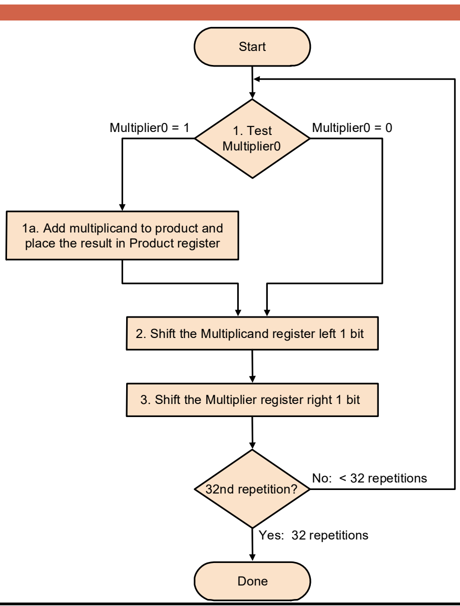
O mesmo procedimento é válido para números em base 2 !

6

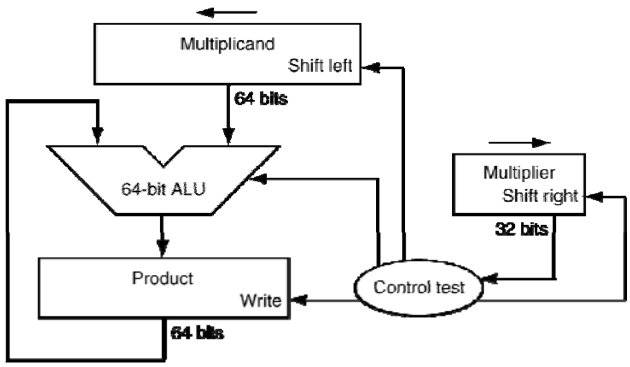
Multiplicação – Algoritmo 1

Baseado na versão "escolar" exemplificada anteriormente.

Multiplicação – Algoritmo 1



Multiplicação – Algoritmo 1



Multiplicação – Algoritmo 1

Exercício:

Efetue passo-a-passo a multiplicação de

$2_{ten} \times 3_{ten}$, ou $0010_{two} \times 0011_{two}$,

utilizando números de 4 bits.

Multiplicação – Algoritmo 1

Interação	Passo	Multiplicador	Multiplicando	Produto
0	Valores Iniciais	0011	0000 0010	0000 0000
1	1: prod= prod + multiplicando	0011	0000 0010	0000 0010
	shift left multiplicando	0011	0000 0100	0000 0010
	shift right multiplicador	0001	0000 0100	0000 0010
2	1: prod= prod+multiplicando	0001	0000 0100	0000 0110
	shift left multiplicando	0001	0000 1000	0000 0110
	shift right multiplicador	0000	0000 1000	0000 0110
3	0: nada a somar	0000	0000 1000	0000 0110
	shift left multiplicando	0000	0001 0000	0000 0110
	shift right multiplicador	0000	0001 0000	0000 0110
4	0: nada a somar	0000	0001 0000	0000 0110
	shift left multiplicando	0000	0010 0000	0000 0110
	shift right multiplicador	0000	0010 0000	0000 0110

Resultado final p/ 2x3: 6

11

Multiplicação – Algoritmo 1

Desempenho: Quantos ciclos de clock este algoritmo necessita para ser executado?

12

Multiplicação – Algoritmo 1

Desempenho: Quantos ciclos de clock este algoritmo necessita para ser executado?

Depende do que pode ser feito em paralelo. Assumindo que temos a seguinte dependência sequencial, e correspondente número de ciclos para cada uma:

teste (t) -> soma (s) -> shifts (sh)

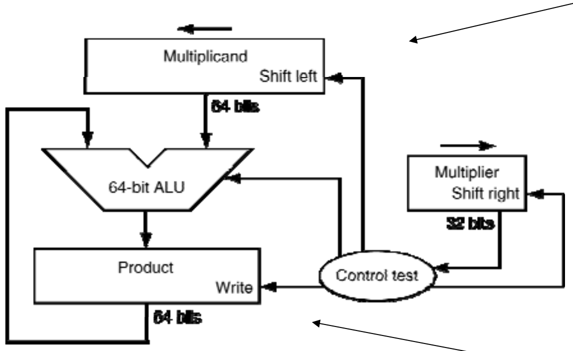
necessitaria de n* (t + s + sh) ciclos, onde n é o número de bits. Para inteiros de 32 bits, precisaria de quase 100 ciclos!

Porém, este é uma questão totalmente dependente da microarquitetura da CPU. O ponto crucial é que, e um computador, a multiplicação é uma operação lenta, comparada com adição e subtração.

Porém.... Add / Sub tipicamente são muito mais frequentes em programas do que Mul: 5 a 100 vezes mais !

Hardware p/ Algoritmo 1

64 bits p/ multiplicando, de modo a acomodar os "shift-left"



64 bits p/ produto, de modo a acomodar o resultado final

Em cada passo:
- multiplicando é deslocado p/ esquerda (shift-left)
- o próximo bit do multiplicador é verificado
- se for igual a 1, o multiplicando deslocado é adicionado ao produto.

Hardware p/ Algoritmo 1

- Outros algoritmos para multiplicação e divisão são baseados em princípios e estratégias similares:
  - Adições e Deslocamentos (shifts) sucessivos

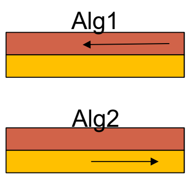
15

Multiplicação – Algoritmo 2

Otimização do Algoritmo 1, baseado nos seguintes fatos:

•Metade dos bits do multiplicando de 64 bits eram sempre iguais a zero, não acrescentando nada ao resultado final.

•Isso porque os novos bits inseridos a direita eram sempre iguais a zero.

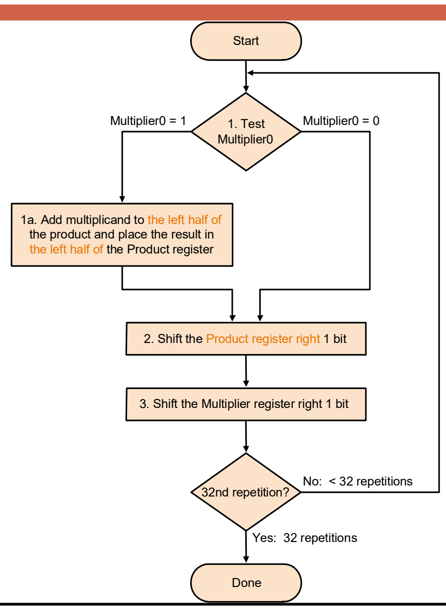


•Assim, ao invés de efetuar "shift-left" no multiplicando, esta nova versão efetua um "shift-right" no produto.

•Desse modo, necessita-se de uma ALU de 32 bits, ao invés de 64 bits, que é mais rápida para efetuar Adições.

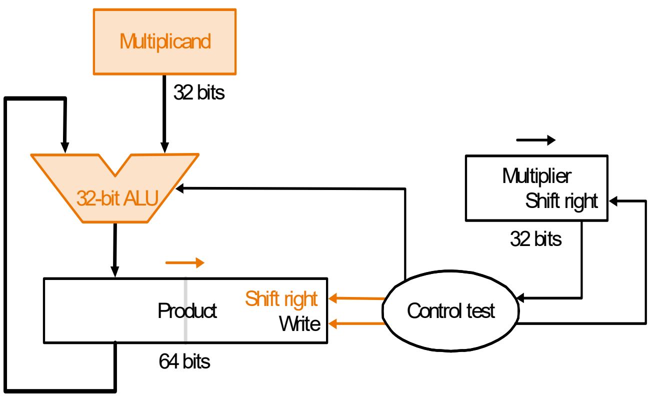
16

Multiplicação – Algoritmo 2



17

Multiplicação – Algoritmo 2



18

Multiplicação – Algoritmo 2

Exercício:

Utilizando o Algoritmo 2, efetue passo-a-passo a multiplicação de

$2_{ten} \times 3_{ten}$, ou $0010_{two} \times 0011_{two}$,

utilizando números de 4 bits.

19

Multiplicação – Algoritmo 2

Interação	Passo	Multiplicador	Multiplicando	Produto
0	Valores Iniciais	0011	0010	0000 0000
1	1: prodH= prodH + multiplicando	0011	0010	0010 0000
	shift right produto	0011	0010	0001 0000
	shift right multiplicador	0001	0010	0001 0000
2	1: prodH= prodH + multiplicando	0001	0010	0011 0000
	shift right produto	0001	0010	0001 1000
	shift right multiplicador	0000	0010	0001 1000
3	0: nada a somar	0000	0010	0001 1000
	shift right produto	0000	0010	0000 1100
	shift right multiplicador	0000	0010	0000 1100
4	0: nada a somar	0000	0010	0000 1100
	shift right produto	0000	0010	0000 0110
	shift right multiplicador	0000	0010	0000 0110

*prodH: somar à metade esquerda do produto

20

Resultado final p/ 2x3: 6

Multiplicação – Algoritmo 2

Desempenho: Quantos ciclos de clock a versão 2 do algoritmo necessita para ser executado?

Multiplicação – Algoritmo 2

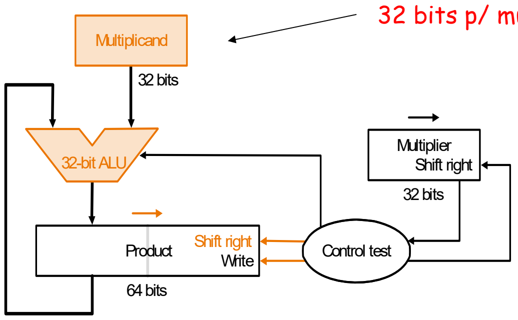
Desempenho: Quantos ciclos de clock este algoritmo necessita para ser executado?

Em princípio o mesmo que a versão 1 do algoritmo:

$n^* (t + s + sh)$ ciclos, onde n é o número de bits.

Porém, o valor de s é diretamente dependente do desempenho da adição, que certamente será melhor para a ALU de 32 bits do algoritmo 2.

Hardware p/ Algoritmo 2



32 bits p/ multiplicando e ALU

64 bits p/ produto, de modo a acomodar o resultado final

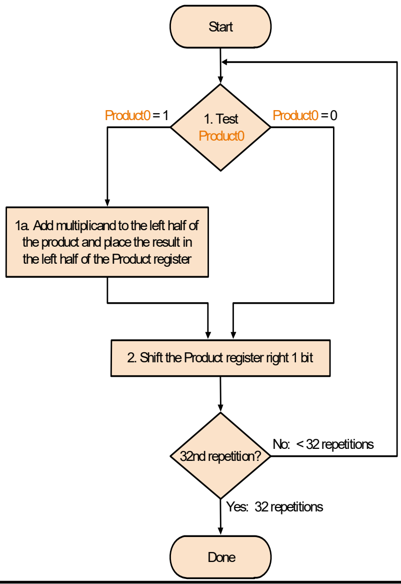
Multiplicação – Algoritmo 3

Otimização do Algoritmo 2, baseado nos seguintes fatos:

- O registrador para armazenar o produto desperdiça espaço de armazenamento igual ao tamanho do multiplicador.

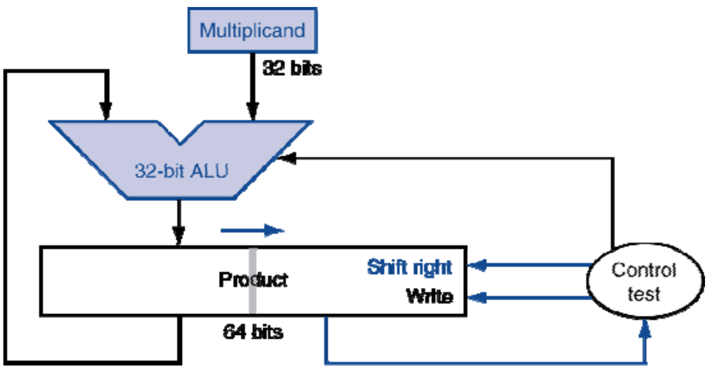
- Assim, a versão 3 do algoritmo utiliza a metade direita do registrador do produto para armazenar o multiplicador. Na medida em que ocorrem "shifts-right", abre-se espaço para os dígitos significativos do produto.

Multiplicação – Algoritmo 3



25

Multiplicação – Algoritmo 3



26

Multiplicação – Algoritmo 3

Exercício:

Utilizando o Algoritmo 3, efetue passo-a-passo a multiplicação de

$2_{ten} \times 3_{ten}$, ou $0010_{two} \times 0011_{two}$,

utilizando números de 4 bits.

27

Multiplicação – Algoritmo 3

Interação	Passo	Multiplicando	Produto
0	Valores Iniciais	0010	0000 0011
1	1: prodH= prodH + multiplicando	0010	0010 0011
	shift right produto	0010	0001 0001
2	1: prodH= prodH + multiplicando	0010	0011 0001
	shift right produto	0010	0001 1000
3	0: nada a somar	0010	0001 1000
	shift right produto	0010	0000 1100
4	0: nada a somar	0010	0000 1100
	shift right produto	0010	0000 0110

Multiplicador inicialmente armazenado na metade inferior do Produto

*prodH: somar à metade esquerda do produto

Resultado final p/ 2x3: 6

28

Multiplicação – Algoritmo 3

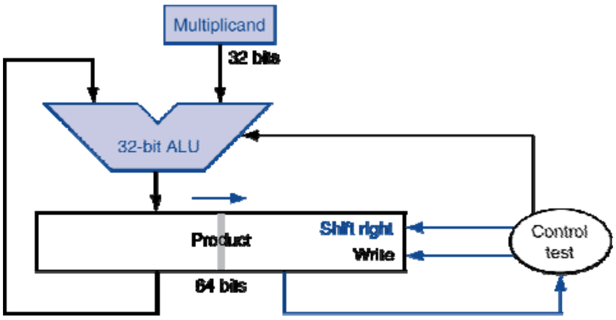
Desempenho: Quantos ciclos de clock a versão 3 do algoritmo necessita para ser executado?

Multiplicação – Algoritmo 3

Desempenho: Quantos ciclos de clock este algoritmo necessita para ser executado?

Em princípio o mesmo que a versão 2 do algoritmo, porém com uma lógica mais simples, o que tende a melhorar (diminuir) o tempo de um ciclo de clock.

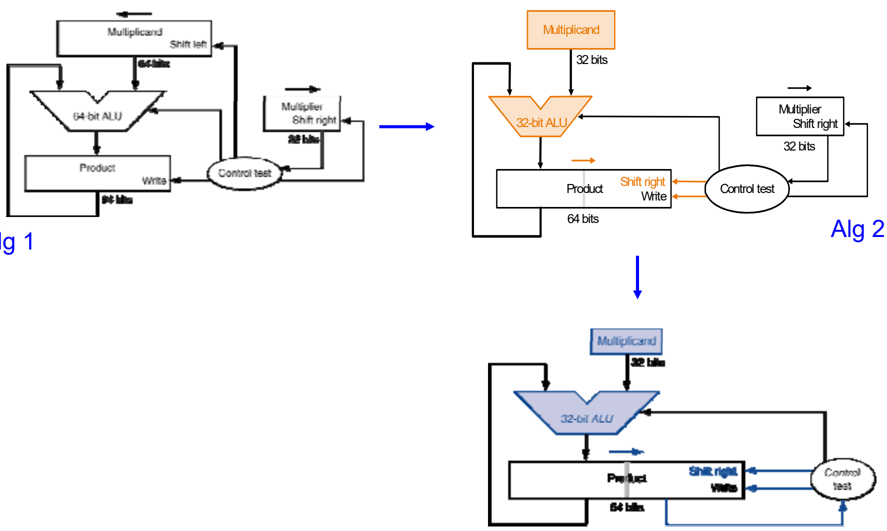
Hardware p/ Algoritmo 3



- A ALU de 32-bits e o registrador do multiplicando não são modificados
- A soma é sucessivamente deslocada para a direita
- Em cada passo, o número de bits em (produto+multiplicador) =64 permitindo assim compartilhar esse registrador.

31

Hardware Algortimos Mul: 1,2,3



Alg 1

Alg 2

32

Alg 3

Exercício

Exercício:

Utilizando o Algoritmo 3, efetue passo-a-passo a multiplicação de

$3_{ten} \times 5_{ten}$, ou $0011_{two} \times 0101_{two}$,

utilizando números de 4 bits.

33

Multiplicação com Sinal

- O algoritmo anterior também funciona para números com sinal (em Complemento de 2).
- Outra alternativa é converter os números negativos para positivo, efetuar a multiplicação, e aplicar o sinal no resultado de acordo com os sinais originais (ex: + * - = -)
- O produto de dois números de 32 bits é um número de 64 bits. Assim, em MIPS o produto é armazenado em dois registradores de 32-bits.

34

Multiplicação Paralela

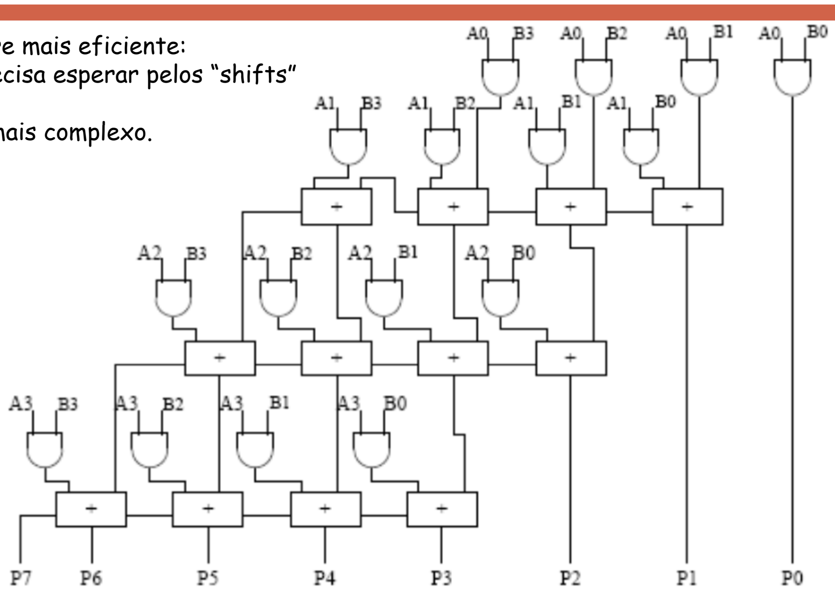
				b3	b2	b1	b0	= B
				a3	a2	a1	a0	= A
				a0b3	a0b2	a0b1	a0b0	= W1
			a1b3	a1b2	a1b1	a1b0		= W2
		a2b3	a2b2	a2b1	a2b0			= W3
	a3b3	a3b2	a3b1	a3b0				= W4
P7	P6	P5	P4	P3	P2	P1	P0	= AxB=P

35

Possível Implementação

Hardware mais eficiente:
•Não precisa esperar pelos "shifts"

Porém, mais complexo.



http://ppginf.ucpel.tche.br/TI-arquivos/2007/PPGINF-UCPel-TI-2007-1-01.pdf

Instruções MIPS para Multiplicação

Produto calculado (64 bits) é armazenado em 2 registradores de 32 bits cada:

Registrador $HI: 32 bits mais significativos

Registrador $LO: 32 bits menos significativos

37

Instruções MIPS para Multiplicação

Instruções:

```
mul $rd, $rs, $rt
```

32 bits menos signifivativos do produto: $rd

```
mult $rs, $rt  /  multu $rs, $rt  # Mult com/sem sinal
```

Resultado de 64-bits em HI/LO

```
mfhi rd  /  mflo rd
```

Move registradors $HI ou $LO para um registrador de uso geral

Exemplo de uso: Testar se $HI != 0 para verificar se ocorreu overflow em uma multiplicação de 32 bits.

38

39

Divisão

Divisão

- Similar à Multiplicação, porém com algumas complicações adicionais:
  - Divisão por zero
  - Calculo dos sinais do quociente e resto

- Feita por meio de subtrações e comparações sucessivas.

Divisão

```
          Dividendo          Divisor
           1001010ten      | 1000ten
          -1000            |_________
             1                 1001ten Quociente
             10
             101
             1010
            -1000
Resto     ----------------
               10
```

41

Divisão

```
                     Dividendo          Divisor
                      1001010ten      | 1000ten
                     -1000            |_________
                        1                 1001ten Quociente
comparação    --->      10
                        101
                        1010
                       -1000                   subtração
           Resto     ----------------
                          10
```

42

Divisão – Adaptação p/ HW Digital

```
 1001010     | 1000000
-1000000             1
----------------
 0001010
```

a)

```
 1001010     | 1000000
-1000000              10
----------------
 0001010
-0100000
----------------
 0001010
```

b)

Divisão – Adaptação p/ HW Digital

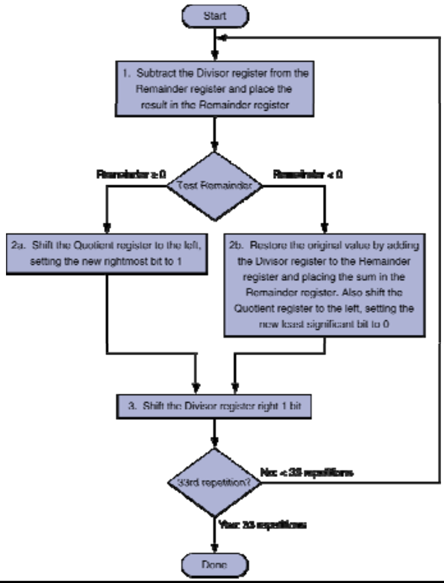
```
 1001010     | 1000000
-1000000              100
----------------
 0001010
-0100000
----------------
 0001010
-0010000
----------------
 0001010
```

c)

```
 1001010     | 1000000
-1000000              1001
----------------
 0001010
-0100000
----------------
 0001010
-0010000
----------------
 0001010
-0001000
----------------
 0000010
```

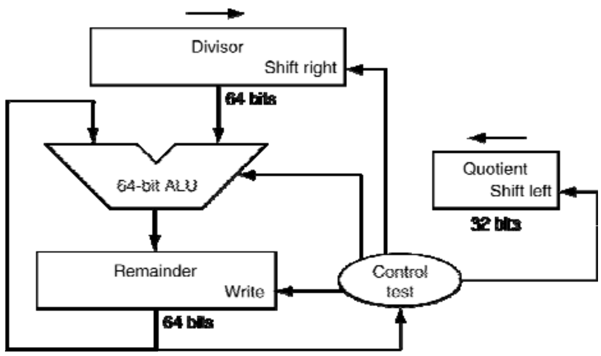
d)

Divisão - Algoritmo



45

Divisão - Algoritmo



46

Divisão

Os passos básicos para se efetuar a divisão de dois números binários são:

1. Inicialmente, quociente= 0, resto= dividendo
2. Subtrair o divisor do resto, armazenando o resultado no resto.
3. Se resto >= 0, Deslocar quociente p/ Esq ←,

   porém inserindo o valor 1 no bit mais à direita.

   Senão, restaure o valor original do resto (adicionando o divisor).

   Deslocar quociente p/ Esq ←, inserindo 0 como bit menos significativo.
4. Deslocar Divisor p/ Dir →
5. Repita passos 2-4 n vezes (n= nro de bits)

47

Divisão

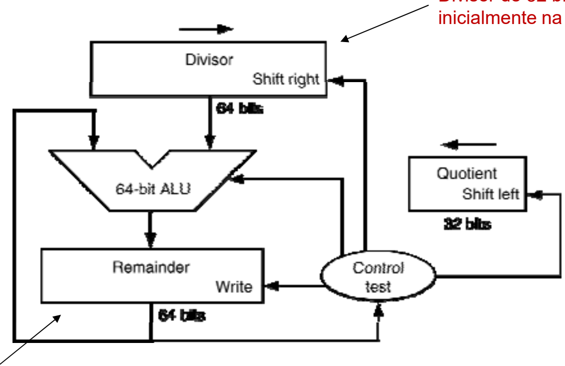
- Exemplo: Dividir 7_{ten} ($0000\ 0111_{two}$) por 2_{ten} (0010_{two})

Iter	Passo	Quociente	Divisor	Resto
0	Valores Iniciais	0000	0010 0000	0000 0111
1	Resto = Resto – Div Resto < 0 → Resto+Div, Shift Left Quociente, inserindo 0. Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Mesmo que anterior	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	Mesmo que anterior	0000	0000 0100	0000 0111
4	Resto = Resto – Div Resto >= 0 → Shift Left Quociente, inserindo 1.	0000 0001 0001	0000 0100 0000 0100 0000 0010	0000 0011 0000 0011 0000 0011
5	Mesmo que anterior	0011	0000 0001	0000 0001

48

Hardware para Divisão

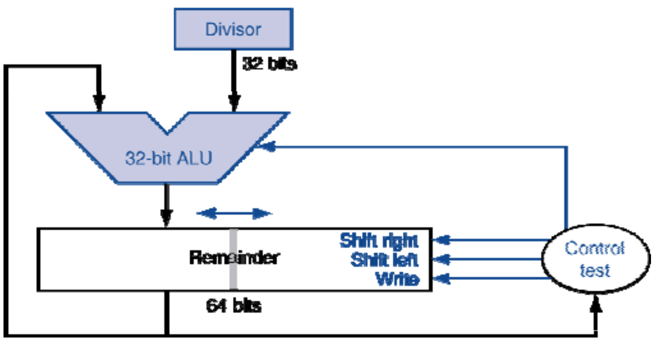
Divisor de 32 bits,
inicialmente na metade esquerda



Dividendo, inicialmente.

A comparação exige uma subtração; se o sinal do resultado for negativo, o divisor deve ser adicionado de volta.

Otimizações de Hardware



Semelhante ao
Hardware da
da multiplicação, possibilitando
seu uso p/ ambas operações.

Divisão de Números Negativos

- Solução mais simples: converter os operandos para positivo, e ajustar os sinais do quociente e resto depois.
- Note que existem múltiplas soluções para a equação abaixo:

Dividendo = Quociente x Divisor + Resto

+7 div +2	Quo =	Resto =
-7 div +2	Quo =	Resto =
+7 div -2	Quo =	Resto =
-7 div -2	Quo =	Resto =

Divisão de Números Negativos

- Solução mais simples: converter os operandos para positivo, e ajustar os sinais do quociente e resto depois.
- Note que existem múltiplas soluções para a equação abaixo:

Dividendo = Quociente x Divisor + Resto

+7 div +2	Quo = +3	Resto = +1
-7 div +2	Quo = -3	Resto = -1
+7 div -2	Quo = -3	Resto = +1
-7 div -2	Quo = +3	Resto = -1

Divisão de Números Negativos

- Solução mais simples: converter os operandos para positivo, e ajustar os sinais do quociente e resto depois.
- Note que existem múltiplas soluções para a equação abaixo:

Dividendo = Quociente x Divisor + Resto

+7 div +2	Quo = +3	Resto = +1
-7 div +2	Quo = -3	Resto = -1
+7 div -2	Quo = -3	Resto = +1
-7 div -2	Quo = +3	Resto = -1

Convenção:

- Dividendo e resto tem o mesmo sinal
- Quociente é negativo se os sinais são diferentes

Instruções MIPS para Divisão

Registrador $HI: 32 bits armazenam o resto da divisão

Registrador $LO: 32 bits armazenam o quociente da divisão

Instruções MIPS para Multiplicação

Instruções:

```
div $rs, $rt  /  divu $rs, $rt   # Div com/sem sinal
```

Resulado de 64-bits em HI/LO

Obs: hardware não verifica overflow ou divisão por 0. Essas verificações devem ser feitas pelo software!

```
mfhi rd  /  mflo rd
```

Move registradors $HI ou $LO para um registrador de uso geral

Exemplo de uso: Testar se $HI != 0 para verificar se ocorreu overflow do quociente.

Conclusão

- Multiplicação e Divisão:
  - Operações relativamente pouco frequentes em programas de uso geral;
  - Exigem hardware mais complexo, e vários ciclos de processamento.
  - Podem retardar a execução de alguns programas.
  - Existem diversos algoritmos e implementações além daquelas apresentadas na aula. Alguns possuem vantagens de maneira geral, outros apenas para arquiteturas específicas.
  - Até aqui tratamos apenas de operações de números inteiros.
  - Números reais serão tratados na aula sobre Representação em Ponto Flutuante.

Lab – MIPS Assembly - GCD

Dados 2 números a e b, implementar um programa para calcular o maior divisor comum entre eles (GCD).

Lab – MIPS Assembly - GCD

Ex: GCD (30,18) = 6

a	b
30	18
18	12
12	6
6	0

Código C:

```
int gcd(int a, int b) {
   if (b == 0)
      return a;
   else
      return gcd(b, a % b);   // chamada recursiva
}

void main(void) {
   int a, b;

   printf("Digite os valores de a e b: ");
   scanf("%d%d", &a, &b);
   printf("gcd(%d, %d) = %d\n", a, b, gcd(a, b));
}
```