Arquitetura de Computadores I
Aritmética binária

António de Brito Ferrari
ferrari@ua.pt

1. Soma e subtração

Representação de Inteiros

- No. de bits = comprimento de palavra da arquitetura
  - MIPS: 32-bits
- Representação de valores negativos - 2's complement
  - Bit de sinal: bit mais significativo (com peso -2^{n-1})
  - Gama de representação: -2^{n-1} … $(2^{n-1} - 1)$

Soma de inteiros

- Unsigned: se valor da soma > ($2^n - 1$) carry-out da posição mais significativa ($C_n = 1$) então valor da soma excede a gama de representação - não representável em n bits: A+B > ($2^n - 1$)
- 2's complement: se $C_n \neq C_{n-1}$ (C_n **xor** $C_{n-1} = 1$) **overflow** - valor da soma excede a gama de representação - não representável em n bits: A+B > ($2^{n-1} - 1$) ou A+B < (-2^{n-1})
  - MIPS: quando ocorre overflow em *add, addi, sub* é gerada uma exceção

Subtração de inteiros

- Em 2's complement: subtrair é somar ao subtraendo o complemento para 2 do subtrator:

  $A - B = A + (2^n - B)$
- Obtenção do 2's complement:
- **$(2^n - B) = (2^n - 1 - B) + 1$**

somar 1

negar todos os bits de B

Somadores

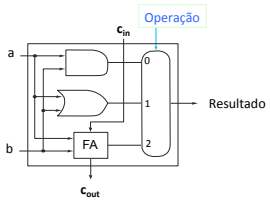
- Somador de 1 bit: Full-Adder (FA)

  Entradas: **a**, **b**, $\mathbf{c_{in}}$

  Saídas: **s**, $\mathbf{c_{out}}$ **$s = a$ xor b xor c_{in}**

  $c_{out} = (a.b)$ or $(a.\ c_{in})$ or $(b.\ c_{in})$
- Somadores de n-bits:
  - Ripple-Carry Adder (RCA) – ligação em série de n FAs
    - Lento: $t_{delay} = nt_{FA}$
  - Carry-Lookahead Adder (CLA)
    - Mais rápido, mais caro

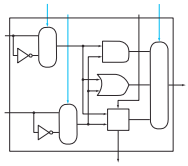
1-bit ALU



Código de Operação
00 – AND
01 – OR
11 - ADD

Sinais de Controle – a azul, verticais
Dados (***Datapath***)- a preto, na horizontal

1-bit ALU: AND, OR, ADD, SUB



2. Multiplicação

Multiplicação

Unsigned

multiplicando

multiplicador

produto

```
  1000   8
x 1001   9
  1000
 0000
0000
 1000
01001000  72
```

Operandos: $0..(2^n-1)$ Produto: $0..(2^{2n}-2^{n+1}+1)$
No. de bits do produto = 2n-bits (soma do no. de bits dos operandos)

Signed

$X*Y = X*y_0 + X*2y_1 + \ldots + X*2^{(n-2)}y_{n-2} - X*2^{(n-1)}y_{n-1}$

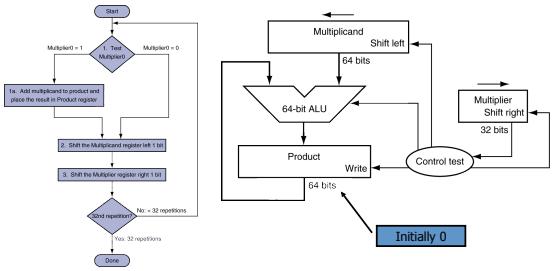
```
  1000   -8 = X
x 1001   -7 = Y
11111000  <- sign-extension
0000000
000000      2's complement
01000   <-  do multiplicando
00111000  +56
```

Operandos: $-2^{n-1}..+(2^{n-1}-1)$
Produto: $(-2^{2n-2}+2^{n+1}).. +2^{2n-2}$

2n-bits

Multiplicador série

Algoritmo Add-shift

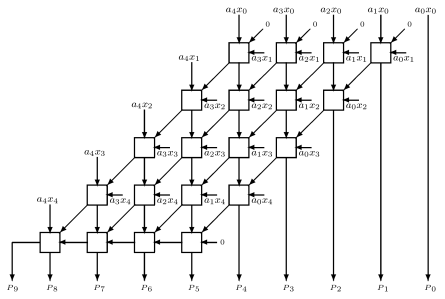


Multiplicador série: otimização

1. Modifique o algoritmo add-shift de modo a utilizar uma ALU de 32-bits e um registo de 32-bits para armazenar o multiplicando. Desenhe o respetivo esquema do multiplicador.
2. Usando o desenho otimizado em 1 melhore-o assumindo que inicialmente o multiplicador está armazenado na metade direita do registo produto. Escreva o algoritmo modificado e desenhe o respetivo multiplicador.
3. Que modificações teria de introduzir para multiplicar em 2's complement?

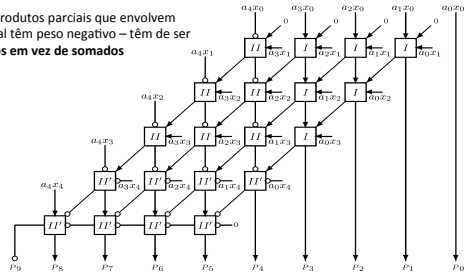
Multiplicador Paralelo
Array Multiplier (unsigned)

Carry-Save array

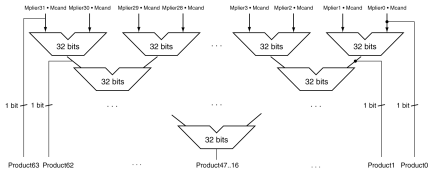


Multiplicador Paralelo
Array Multiplier (signed)

Bits dos produtos parciais que envolvem bit de sinal têm peso negativo – têm de ser **subtraídos em vez de somados**



Multiplicador Paralelo em árvore:
Wallace tree



Algoritmo de Booth (signed multiplication)

Recodificação do multiplicador:

y_i	y_{i-1}	Operação	
0	0	Shift	
0	1	Add - Shift	← Fim de sequência de 1s
1	0	Add 2's complement - Shift	← Início de sequência de 1s
1	1	Shift	

- Algoritmo baseia-se na decomposição de sequências de 1s:

$\sum_{i=0}^{k-1} y_i = 2^k - 2^0$ Exemplo: 1111 = 10000 – 0001

- Algoritmo para multiplicador de n-bits: i = 0 … n; $i_{-1} = 0$

Algoritmo de Booth "2 bits at a time"

y_{i+1}	y_i	y_{i-1}	Operação	
0	0	0	Shift 2 bits	Sequência de 0s
0	0	1	Add X – shift 2 bits	Fim de sequência de 1s na posição (**i-1**)
0	1	0	Add X - shift 2 bits	1 isolado na posição **i**
0	1	1	Add 2*X - shift 2 bits	Fim de sequência de 1s na posição **i**
1	0	0	Sub 2*X - shift 2 bits	Início de sequência de 1s na posição (**i+1**)
1	0	1	Sub X - shift 2 bits	Fim de sequência de 1s na posição (**i-1**) e Início de sequência de 1s na posição (**i+1**)
1	1	0	Sub X - shift 2 bits	Início de sequência de 1s na posição **i**
1	1	1	Shift 2 bits	Sequência de 1s

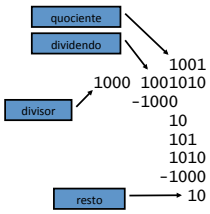
Metade do número de Produtos Parciais gerados – **n/2 ciclos**

Multiplicação no MIPS

- Dois registos de 32-bit para o produto
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instruções
  - `mult rs, rt / multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd / mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

Divisão

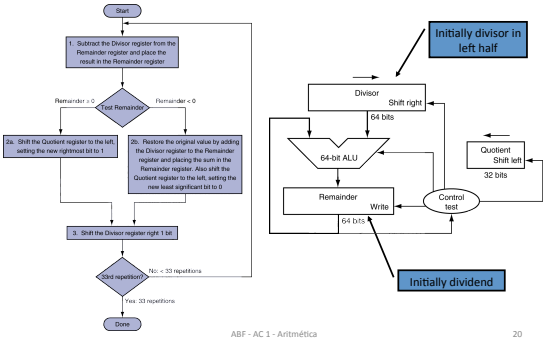
Dividendo = Quociente * Divisor + Resto



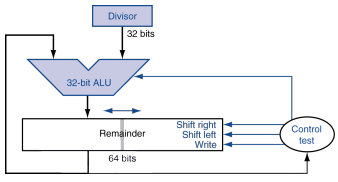
Operandos de n-bits: *n*-bit quociente e resto: *n*-bit

- Check for 0 divisor
- Divisor ≠ 0
  - ***If*** divisor ≤ bits do dividendo
    - novo bit do quociente = 1, subtrair
  - ***else***
    - novo bit do quociente = 0
  - Juntar ao resto parcial bit seguinte do dividendo
- Restoring division
  - Subtrair sempre o divisor; se resto < 0 somar de novo o divisor
- Signed division
  - Dividir usando os valores absolutos
  - Ajustar o sinal do quociente e do resto

Divisor



Divisor otimizado



- Um ciclo por cada subtração do resto parcial
- Semelhante ao multiplicador!
  - O mesmo hardware pode ser usado para ambos

Exercício

- Reescrever o fluxograma da divisão para a versão otimizada do divisor

Divisão no MIPS

- Registos HI/LO usados para o resultado
  - HI: resto da divisão
  - LO: quociente
- Instruções
  - `div rs, rt` / `divu rs, rt`
  - **No divide-by-0 checking**
    - Cabe ao software fazer o teste, se necessário
  - `mfhi`, `mflo` para aceder ao resultado