Arquitectura dos Computadores

Aula 2 – Endereçamento e Decisões

Professor: Dr. Christophe Soares

Recordando

Memória		Registos	
Endereço	**Conteúdo**	**Nome**	**Conteúdo**
.....	.....	.....	.....
0000AAA0	00000003	S0	0000AAA4
0000AAA4	0000000A	S1	0000AAA8
0000AAA8	00000002	S2	00000002
0000AAAC	00000001	S3	00001234
.....	.....	.....	.....

Perante a memória e registos apresentados acima:

```
lw $t0, 4($s0)
```
⇢ `li $t0, 2`

```
sw $s3, 8($s0)
```
⇢ **colocar 00001234 na posição de memória 0000AAAC**

Recordando

Se escrever:

```
add $t2,$t1,$t0
```

então $t0 e $t1 devem conter NÚMEROS

```
lw $t2,0($t0)
```

então $t0 deve conter um ENDEREÇO

Não confundir!

Endereçamento: “Byte” vs. “Word”

Palavras de memória (“*words*”) têm o seu endereço

Computadores numeravam as palavras de memória como elementos de um vetor:

Memoria[0], Memoria[1], Memoria[2], …

Os computadores necessitam de aceder a *bytes* (8 bits) bem como a palavras (4 bytes/*word*)

Hoje em dia as máquinas endereçam a memória ao byte, (i.e.,“*Byte Addressed*”) sendo assim palavras de 32-bit (4 bytes) ocupam 4 posições de memória

Memoria[0], Memoria[1], Memoria[2], …

&Memoria+0*4, &Memoria+1*4, &Memoria+2*4, …

Como calcular endereços?

Qual o offset devo usar no lw para selecionar A[5] em C?

5ª posição inicia-se a 20 bytes do início do vector

Reescrevendo: **g = h + A[5]**

$s1 – g
$s2 – h
$s3 - &A[0] - A

```
lw  $t0, 20($s3)    # $t0 fica com A[5]
add $s1, $s2, $t0   # $s1 = h+A[5]
```

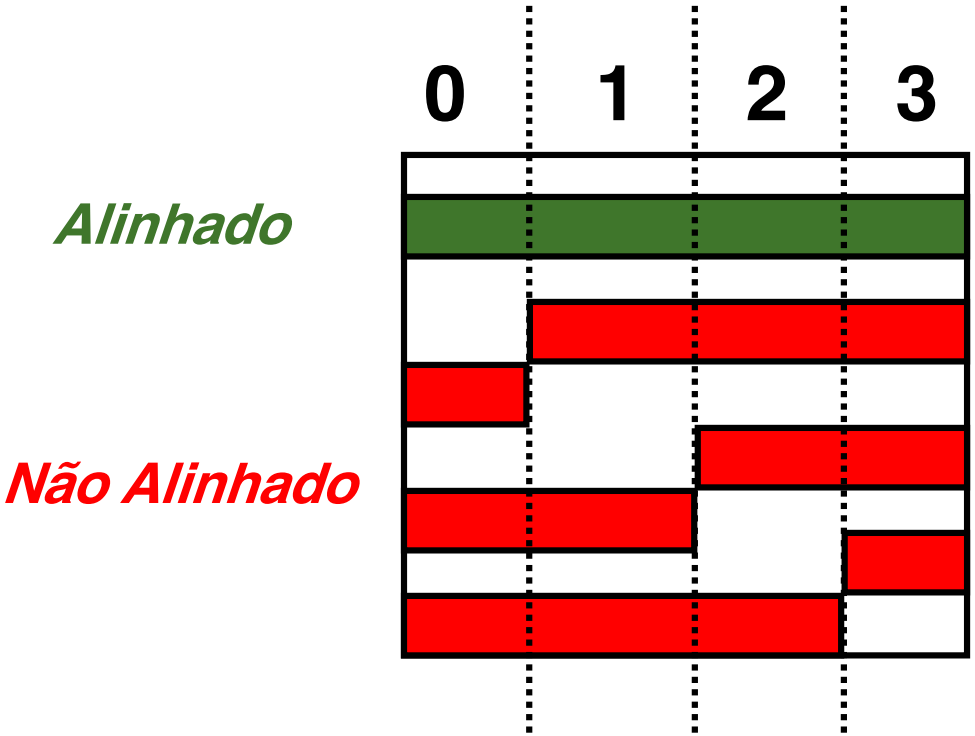
Notas sobre a memória: deslocamento

Falha comum:

- **Esquecer que os endereços de palavras consecutivas não diferem de 1 mas de 4!**
- **Portanto para o *lw* e *sw*, a soma do endereço base e o deslocamento tem que ser um múltiplo de 4!**
- **Manter o alinhamento à palavra!**

Notas sobre a memória: alinhamento

O MIPS requer que todas as palavras se iniciem em endereços múltiplos de 4 bytes…



Último dígito do endereço é:

0, 4, 8, ou C_{hex}

1, 5, 9, ou D_{hex}

2, 6, A, ou E_{hex}

3, 7, B, ou F_{hex}

Alinhamento: os objetos estão guardados em endereços múltiplos do seu tamanho

Movimentando bytes 1/2

Para além da transferência de palavras (*lw*, *sw*), no MIPS também há transferência de bytes:

load byte: *lb*

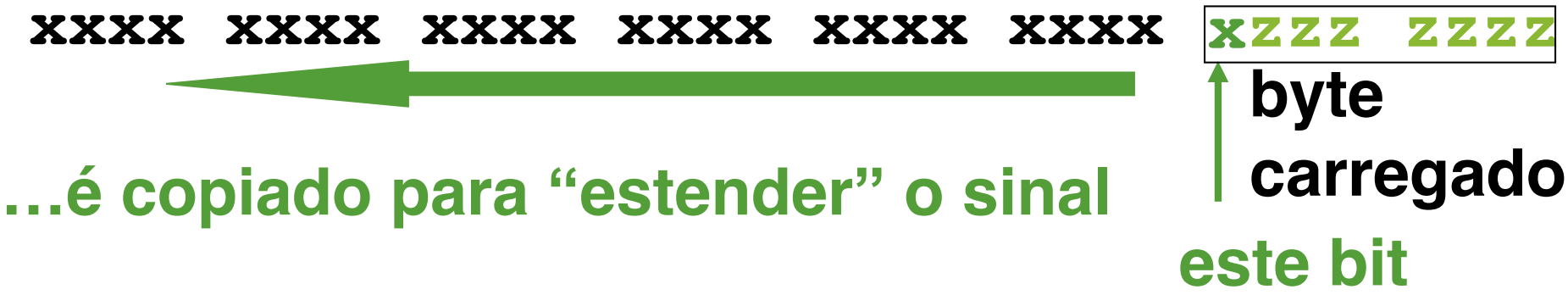
store byte: *sb*

Mesmo formato que: lw, sw

Movimentando bytes 2/2

O que fazer com os restantes 24 bits do registo de 32 bits?

- lb: o sinal estende-se para os 24 bits



Normalmente não queremos este comportamento com caracteres

Há uma instrução no MIPS que não estende o sinal ao carregar bytes: *lbu* (“*load byte unsigned*”)

Registos vs. Memória

O que fazer se há mais variáveis do que registos?

- **O compilador tenta ter as variáveis mais usadas nos registos**
- **E as menos usadas em memória: "spilling"**

Porque não manter todas em memória?

- **"*Smaller is faster*": Os registos são mais rápidos do que a memória**

Uma instrução aritmética no MIPS pode ler 2, operar com eles, e escrever o resultado num 3º

Uma instrução MIPS para transferir dados apenas lê ou escreve um operando sem executar qualquer operação!

"*Overflow*" em operações aritméticas (1/2)

"*Overflow*" ocorre quando há um erro em operações aritméticas devido à precisão limitada dos computadores.

Exemplo (números de 4-bit sem sinal):

15	1111
+3	+0011
18	1 0010

Como não temos espaço para a solução de 5-bit , ficamos apenas com 0010, o que é 2, e estaria errado!

"*Overflow*" em operações aritméticas (2/2)

Algumas linguagens detetam "*overflow*" (Ada), outras não (C)

A solução do MIPS foi ter 2 tipos de instruções aritméticas para tratar ambas as situações:

- **adição (*add*), adição com constante (*addi*) e subtração (*sub*) permitem detectar "overflow"**
- **Adição sem sinal (*addu*), adição com constante sem sinal (*addiu*) e subtração sem sinal (*subu*) não permitem detetar "*overflow*"**

O compilador seleciona a aritmética apropriada

- **Compiladores de C para MIPS produzem *addu*, *addiu*, *subu***

Duas instruções lógicas…

***Shift Left*:** `sll $s1, $s2, 2 #s1=s2<<2`

Guarda em $s1 o valor de $s2 deslocado 2 bits para a esquerda, inserindo 0's à direita; (<< em C)

$0000\ 0010_{bin}$	$0\ 2_{hex}$	2_{dec}
$0000\ 1000_{bin}$	$0\ 8_{hex}$	8_{dec}
$0010\ 0000_{bin}$	$2\ 0_{hex}$	32_{dec}

Que efeito aritmético tem o deslocamento à esquerda?

***Shift Right*:** `srl $s1, $s2, 2 #s1=s2>>2`

Até agora...

Todas as instruções que vimos apenas manipulam dados… temos uma calculadora.

Para construir um computador necessitamos de poder tomar decisões…

O C (e o MIPS) possuem etiquetas (“*labels*“) que suportam saltos (“*goto*”) para zonas do código.

- **C não são recomendados, mas no MIPS são essenciais!**

Decisões em C : if

2 tipos de condições em C:

if (condição) {código}

if (condição) {código1} else {código2}

Sendo que o *if...else* pode ser reescrito para:

```
if  (condição) goto L1;
      {código2}
      goto L2;
L1: {código1}
L2:
```

Decisões no MIPS

Instrução para decisões no MIPS:

```
beq   $s1, $s2, Label1
```

Registos ilustrativos: $s1, $s2, $s3, $s4

beq significa "branch if equal"
em C: if (s1==s2) goto Label1

Instrução de decisão complementar no MIPS:

```
bne   $s3, $s4, Label2
```

bne significa "branch if not equal"
em C: if (s3!=s4) goto Label2

Chamados Saltos Condicionais

Instrução “goto” no MIPS

Para além dos saltos condicionais, no MIPS há o salto incondicional:

```
j end    # salto para a label 'end'
```

É a instrução de salto (“jump”) salta diretamente para a label “*end*” sem verificar qualquer condição.

Em C: *goto end*

Tecnicamente é o mesmo que:

```
beq $0, $0, end
```

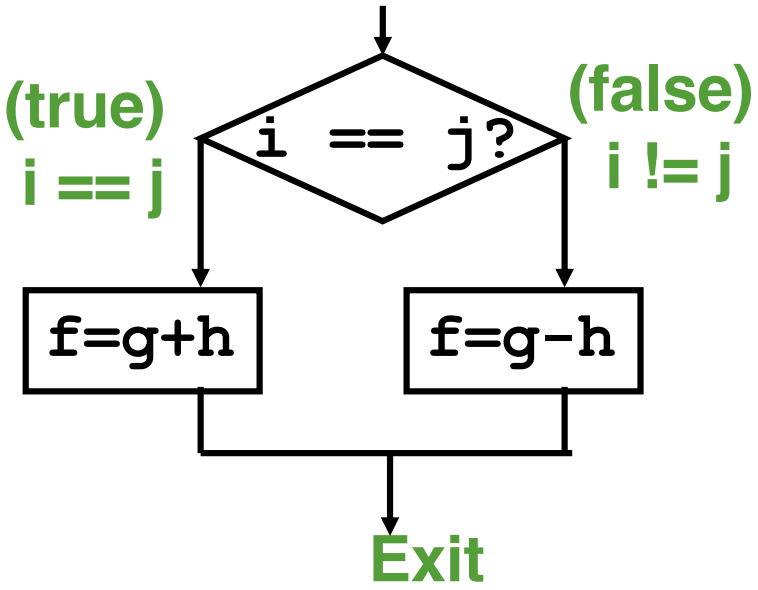
Uma vez que a condição é sempre verdadeira.

if em MIPS (1/2)

Compile à mão o seguinte código:

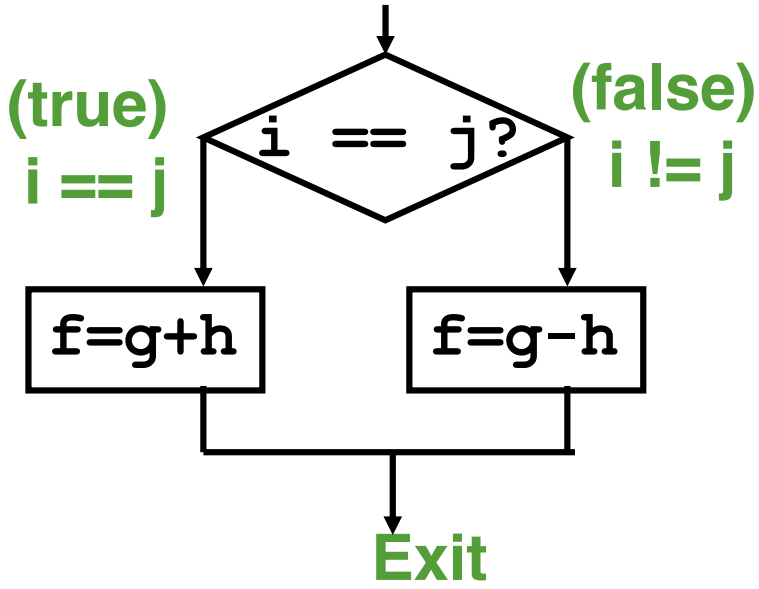
if (i == j) f=g+h;
else f=g-h;

$s0 – f
$s1 – g
$s2 – h
$s3 – i
$s4 – j



if em MIPS (2/2)

```
if (i == j) f=g+h;
else f=g-h;
```



Solução:

```
        beq $s3, $s4, True      # salta i==j
        sub $s0, $s1, $s2       # f=g-h(falso)
        j   End                 # goto End
True:   add $s0, $s1, $s2       # f=g+h (verdade)
End:
```

$s0 – f
$s1 – g
$s2 – h
$s3 – i
$s4 – j

Ciclos em C/Assembler (1/3)

Um ciclo simples em C; A[] é um vetor de inteiros

```
do {
    g = g + A[i];
    i = i + j;
} while (i ≠ h);
```

$s1 – g
$s2 – h
$s3 – i
$s4 – j
$s5 – &A[0]

Reescreve-se como:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i ≠ h) goto Loop;
```

Ciclos em C/Assembler (2/3)

Código do MIPS:

```
Loop:
        sll $t1, $s3, 2          # $t1= 4*I
        add $t1, $t1, $s5        # $t1=addr A[i]
        lw  $t1, 0($t1)          # $t1=A[i]
        add $s1, $s1, $t1        # g=g+A[i]
        add $s3, $s3, $s4        # i=i+j
        bne $s3, $s2, Loop       # if i≠h goto Loop
```

$s1 – g
$s2 – h
$s3 – i
$s4 – j
$s5 – &A[0]

Código original:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i ≠ h) goto Loop;
```

Ciclos em C/Assembler (3/3)

Há 3 tipos de ciclos em C:

while

do… while

for

Cada um pode ser escrito com qualquer dos outros dois, portanto o que fizemos atrás serve também para o *while* e o *for*

Embora existam múltiplas formas de escrever ciclos no MIPS, a chave para a decisão é o salto condicional

Exercício 1 (1/2)

Um ciclo simples em C; A[] é um vetor de inteiros

```
for(i=1 ;i≠j; i++) g = g + A[i];
```

Reescreve-se como:

```
        i = 1;
Loop:   if (i == j) goto End
        g = g + A[i];
        i = i + 1;
        goto Loop;
End:
```

$s1 – g
$s3 – i
$s4 – j
$s5 – &A[0]

Exercício 1 (2/2)

Código do MIPS:

```
        addi $s3,$0,1          #$s3= 1
Loop:   beq  $s3,$s4,End       #goto End if i=j
        sll  $t1,$s3,2         #$t1= 4*i
        add  $t1,$t1,$s5       #$t1=addr A[i]
        lw   $t1,0($t1)        #$t1=A[i]
        add  $s1,$s1,$t1       #g=g+A[i]
        addi $s3,$s3,1         #i=i+1
        j    Loop              #goto Loop
End:
```

$s1 – g
$s3 – i
$s4 – j
$s5 – &A[0]

Código original:

```
        i = 1;
Loop:   if (i == j) goto End
        g = g + A[i];
        i = i + 1;
        goto Loop;
End:
```

Exercício 2 (1/2)

Outro ciclo simples em C

```
g = i;
while(g ≠ 0) {
    j = j + g;
    g = g - 1;
}
```

$s1 – g
$s3 – i
$s4 – j

Primeiro reescrever com goto em C

```
        g = i;
Loop:   if (g == 0) goto End
        j = j + g;
        g = g - 1;
        goto Loop:
End:
```

Exercício 2 (2/2)

Código do MIPS:

```
        add  $s1,$0,$s3      #g=i
Loop:   beq  $s1,$0,End      #goto End if g==0
        add  $s4,$s4,$s1     #j=j+g
        addi $s1,$s1,-1      #g=g-1
        j    Loop            #goto Loop
End:
```

$s1 – g
$s3 – i
$s4 – j

Código original:

```
        g = i;
Loop:   if (g == 0) goto End
        j = j + g;
        g = g - 1;
        goto Loop:
End:
```

Conclusão...

- **Alinhamento da Memória**
- **LW e SW**
- **SB e LB**
- **Beq, Bne**
- **Jumps**
- **Ciclos com Saltos**