

Arquitectura de Computadores

MIPS: Endereçamento e Decisões



Docente: **Pedro Sobral**
<http://www.ufp.pt/~pmsobral>



Arquitectura de Computadores L04 MIPS: Endereçamento e Decisões (1)

Pedro Sobral © UFP

Recordando...

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

◦ **lw \$t0 4(\$s0)** é o mesmo que fazer
t0=2

◦ **sw \$s3 8(\$s0)** é o mesmo que colocar
00001234 na posição de memória
0000AAAC



Arquitectura de Computadores L04 MIPS: Endereçamento e Decisões (2)


Pedro Sobral © UFP

Recordando...

- Se escrever `add $t2,$t1,$t0`
então \$t0 e \$t1
devem conter **NÚMEROS**
- Se escrever `lw $t2,0($t0)`
Então \$t0 deve conter um **ENDEREÇO**
- Não confundir!



Endereçamento: “Byte” vs. “Word”

- Todas as palavras de memória (“words”) têm o seu endereço
- Os primeiros computadores numeravam as palavras de memória como elementos de um vector:
 - `Memoria[0], Memoria[1], Memoria[2], ...`

- Os computadores necessitam de aceder a **bytes** (8bits) 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[4], Memoria[8], ...`



Como calcular endereços?

- Qual o offset devo usar no lw para seleccionar A[5] em C?
- **4x5=20 logo a 5ª posição inicia-se a 20 bytes do início do vector....**
- Compilando à mão usando registos:
 $g = h + A[5];$
 - g: \$s1, h: \$s2, \$s3: endereço base de A
- 1º transferir da memória para um registo:
 lw \$t0, 20(\$s3) # \$t0 fica com A[5]
 (somar 20 a \$s3 para seleccionar A[5]...)
- De seguida executar a adição...
 add \$s1, \$s2, \$t0 # \$s1 = h+A[5]



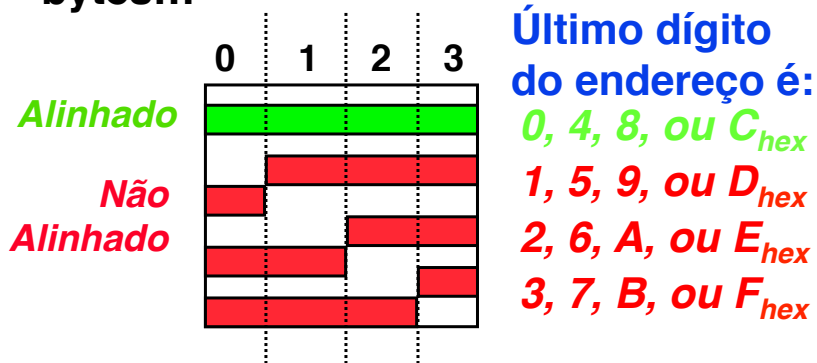
Notas sobre a memória: deslocamento

- 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 ! **(para termos alinhamento à palavra)**



Notas sobre a memória: alinhamento

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



- Tem o nome de **Alinhamento**: os objectos 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 1/2

◦ 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
- Os registos são mais versáteis:
 - 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)

- Nota: “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 10010
- Como não temos espaço para a solução de 5-bit, ficamos apenas com 0010, o que é +2, e está errado.



“Overflow” em operações aritméticas (2/2)

- Algumas linguagens detectam “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 subtracção (sub) **permitem detectar “overflow”**
 - Adição sem sinal (addu), adição com constante sem sinal (addiu) e subtracção sem sinal (subu) **não** permitem detectar “overflow”
- O compilador selecciona 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)

• Antes: `0000 0002hex`
`0000 0000 0000 0000 0000 0000 0000 0010two`

• Depois: `0000 0008hex`
`0000 0000 0000 0000 0000 0000 0000 1000two`

- Que efeito aritmético tem o deslocamento à esquerda?

◦ **Shift Right:** `srl` é o deslocamento oposto; >>



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: Estilo horrível! ; MIPS: **Essencial!**



Decisões em C : *if*

- 2 tipos de condições em C:
 - *if (condição) {código}*
 - *if (condição) {código1} else {código2}*
- Podemos alterar o 2º tipo para:

```
if (condição) goto L1;
código2;
goto L2;
L1: código1;
L2:
```
- Não tão elegante como o *if-else*, mas tem o mesmo significado



Decisões no MIPS

- Instrução para decisões no MIPS:
 - *beq registo1, registo2, L1*
 - *beq* significa “branch if equal”
O mesmo que (em C):
if (registo1==registo2) goto L1
- Instrução de decisão complementar no MIPS:
 - *bne registo1, registo2, L1*
 - *bne* significa “branch if not equal”
O mesmo que (em C):
if (registo1!=registo2) goto L1
- Chamados Saltos Condicionais



Instrução “goto” no MIPS

- Para além dos saltos condicionais, no MIPS há o salto incondicional:
 - j label
- É a instrução de salto (“jump”) salta directamente para a “label” indicada sem verificar qualquer condição.
- O mesmo que (em C): goto label
- Tecnicamente é o mesmo que:
 - beq \$0,\$0,label
 - Uma vez que a condição é sempre verdadeira.



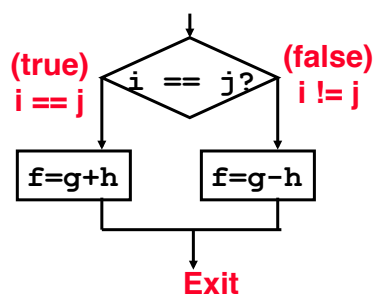
Compiling C `if` into MIPS (1/2)

- Compile à mão o seguinte código:

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

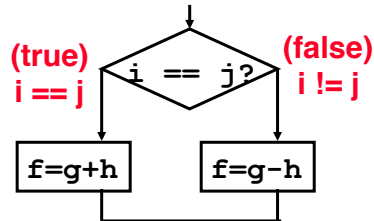
- Use o mapeamento:

```
f: $s0  
g: $s1  
h: $s2  
i: $s3  
j: $s4
```



Compilando o if (em C) para MIPS (1/2)

```
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:
  
```

Nota: O compilador gera automaticamente as “labels” para as decisões (saltos). Geralmente não estão presentes em código de alto nível.



Ciclos em C/Assembler (1/3)

- ° Um ciclo simples em C; A[] é um vector de ints

```

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

- ° Reescreve-se como:

```

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

- ° Use o mapeamento: g, h, i, j, base of A, \$s1, \$s2, \$s3, \$s4, \$s5



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 # goto Loop
                        # if i != h
```

◦ 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.

◦ **Importante:** Embora existam multiplas 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 vector de ints

```
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:
```

- ° Use o mapeamento: $g, i, j, \text{base of } A, \$s1, \$s3, \$s4, \$s5$



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:
```

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;  
}
```

◦ Use o mapeamento: $g, i, j, \$s1, \$s3, \$s4$



Exercício 2 (2/2)

Código do MIPS:

```
Loop: add    $s1,$0,$s3    #g=i  
      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:
```

