



Introdução ao MIPS

- Operadores Aritméticos (continuação) -

Arquitetura de Computadores 2024/2025

Registos no MIPS (Revisão)

- Como os registos são construídos em hardware, existe um número pré-determinado que não pode ser aumentado.
 - Solução: O código do MIPS tem que ser feito com cuidado de forma a usar eficientemente os recursos disponíveis.
- O MIPS tem 32 registos de 32 bits cada (**word**). Os registos estão numerados de 0 a 31
- Os registos tanto podem ser referenciados por um número como por um nome:
 - Referência por número :
\$0, \$1, \$2, ... \$30, \$31
 - Referência por nome :
 - Semelhante às variáveis em C
\$16 - \$23 ➔ \$s0 - \$s7
 - Variáveis temporárias
\$8 - \$15 ➔ \$t0 - \$t7

Operações Aritméticas no MIPS (Revisão)

- Sintaxe:

1 2, 3, 4

Onde :

1) nome da operação

2) operando que recebe o resultado (“destination”)

3) 1º operando (“source1”)

4) 2º operando (“source2”)

- Adição e subtracção em assembly

–add \$s0,\$s1,\$s2 # \$s0=\$s1+\$s2

–sub \$s3,\$s4,\$s5 # \$s3=\$s4-\$s5

–addi \$s0,\$s1,10 # \$s0=\$s1+10

–add \$zero,\$zero,\$s0 # O que acontece?

Overflow Aritmético (1/2)

- O *overflow* acontece quando existe um erro numa operação aritmética devido à precisão limitada dos computadores (número fixo de *bits* por registo)

- Exemplo (números de 4-bits sem sinal):

+15	1111
<u>+3</u>	<u>+ 0011</u>
+18	1 0010

- Não há espaço para o 5º bit da soma, assim a solução seria 0010, que é +2 em decimal, e portanto está errada.

Overflow Aritmético (2/2)

- Algumas linguagens detectam o *overflow* (Ada), enquanto outras não (C)
- No MIPS existem 2 tipos de instruções:
 - add (add), add immediate (addi) e subtract (sub) em que o *overflow* é detectado
 - add unsigned (addu), add immediate unsigned (addiu) e subtract unsigned (subu) que não fazem detecção de *overflow* (no caso de ocorrer é ignorado)
- O compilador utiliza a aritmética conveniente
 - O compilador de C para o MIPS utiliza addu, addiu, subu

Multiplicação no MIPS (1/4)

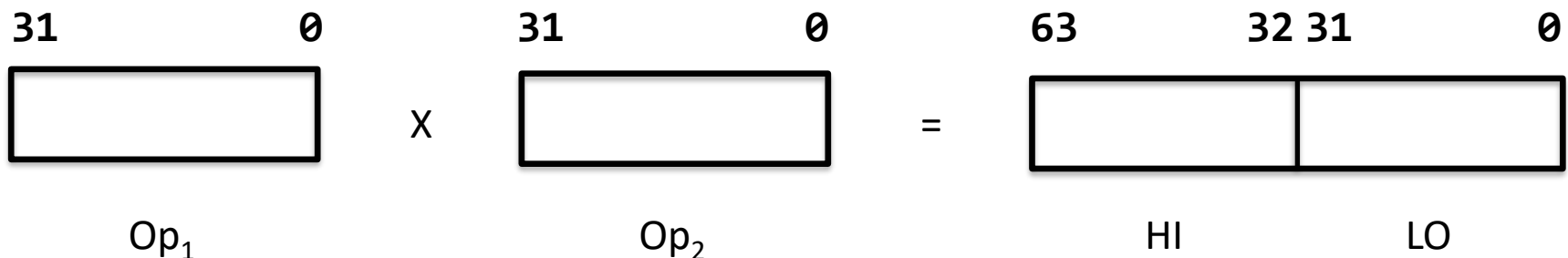
- O produto de dois números de N dígitos decimais pode resultar num número com 2N dígitos.
- Isto é válido para números expressos em qualquer outra base.
- Em particular, o produto de dois inteiros representados como números binários de N dígitos também pode originar um resultado até 2N bits.

$$\begin{array}{r}
 1011 \\
 1101 \\
 \hline
 1011 \\
 0000. \\
 1011.. \\
 1011... \\
 \hline
 10001111
 \end{array}$$

$$\begin{array}{r}
 11_{10} \\
 13_{10} \\
 \hline
 143_{10}
 \end{array}$$

Multiplicação no MIPS (2/4)

- A unidade multiplicadora do MIPS contém dois registos de 32 bits chamados HI e LO.
- Estes registos não são de uso geral.
- Quando dois registos de 32 bits são multiplicados, o resultado que poderá conter até 64 bits é colocado nos registos HI e LO.
- Os bits 32 a 63 do resultado são guardados em HI e os bits 0 a 31 no registo LO.



Multiplicação no MIPS (3/4)

- Existe uma instrução de multiplicação sem sinal para operandos positivos - *multu*
- E uma multiplicação para números com sinal (operados representados em complementos para 2) - *mult*.

```
mult  $s,$t      # HILO ← $s * $t operandos com sinal
multu $s,$t      # HILO ← $s * $t operandos sem sinal
```

- Existe ainda uma outra pseudo-operação de multiplicação especial que guarda a *word* menos significativa num dos registos de uso geral - *mul*

```
mul    $d,$s,$t      # HILO ← $s * $t operandos com sinal
                     # $d = LO
```


Multiplicação no MIPS (4/4)

- Existem duas instruções que permitem depois copiar o valor do resultado da multiplicação para registos de uso geral: *mfhi* e *mflo*

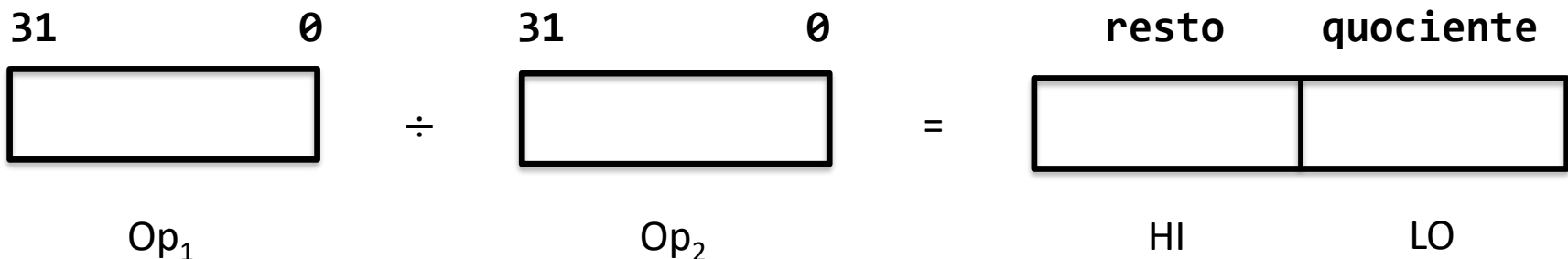
```
mfhi $d      # $d ← HI
```

```
mflo $d      # $d ← LO
```

- Exemplo:

```
# Programa calcular 5 × x - 74 ($t0 ← x e $t1 ← resultado)
```

```
.text
main: addi    $t0,$zero,12    # coloca x no registo $t0
      addi    $t1,$zero,5     # coloca 5 no registo $t1
      mult    $t1,$t0         # lo <-- 5x
      mflo    $t1             # $t1 = 5x
      addi    $t1,$t1,-74     # $t1 = 5x - 74
```



Divisão no MIPS (2/2)

- Exemplo:

```
## Programa para calcular (y + x) / (y - x)
#### Registos:
## $t0  x, $t1  y
## Devolver em $t2 o quociente e em $t3 o resto
```

```
.text
.globl main      # main(8,36)
```

```
main: addi  $t0,$zero,8    # coloca x em $t0
      addi  $t1,$zero,36   # coloca y em $t1
      addu  $t2,$t1,$t0    # $t2 <-- (y+x)
      subu  $t3,$t1,$t0    # $t3 <-- (y-x)
      div   $t2,$t3        # hilo <-- (y+x)/(y-x)
      mflo  $t2            # $t2 <-- quociente
      mfhi  $t3            # $t3 <-- resto
```



Introdução ao MIPS

- Instruções de Decisão -

Arquitetura de Computadores 2024/2025

O que vimos até agora ...

- As instruções que vimos até agora só manipulam informação (operações aritméticas e transferência de dados) ...
- Para construir um computador precisamos de tomar decisões e alterar a sequência de execução durante a execução do programa
- Imagine como seria difícil criar um programa se não existissem instruções “*if*”, “*while*”, “*for*”, etc.!
- O MIPS (e o C!) permitem usar [labels](#) como suporte ao comando “goto”.
 - No C: o uso de “breaks” e “goto” é deselegante e altamente desaconselhado;
 - No MIPS: A utilização de “goto” é a única forma de modificar o fluxo sequencial de execução!



Decisões em C: o comando `if`

- Existem 2 tipos de “`if` statements” em C

```
if (condition) clause
```

```
if (condition) clause1 else clause2
```

- Rearranje o 2º `if` da seguinte forma:

```
if (condition) goto L1;
```

```
    clause2;
```

```
    goto L2;
```

```
L1: clause1;
```

```
L2:
```

- Não é tão elegante como `if-else`, mas faz mesma coisa

Instruções de decisão no MIPS

- Instrução de decisão no MIPS:

```
beq    register1, register2, L1
```

beq significa “Branch if (registers are) equal”

A tradução em C seria:

```
if (register1==register2) goto L1
```

- Instrução de decisão complementar

```
bne    register1, register2, L1
```

bne significa “Branch if (registers are) NOT equal”

A tradução em C seria :

```
if (register1!=register2) goto L1
```

- Estas instruções designam-se por “conditional branches”
(saltos condicionais)

Instrução “goto” no MIPS

- Para além dos saltos condicionais, o MIPS tem ainda o salto incondicional (unconditional branch):

```
j label
```

–O salto na execução é feito directamente para o sítio referenciado por “label” sem ser necessário satisfazer uma condição

- Equivalente em C a:

```
goto label
```

- Tecnicamente tem o mesmo efeito que :

```
beq $0,$0,label
```

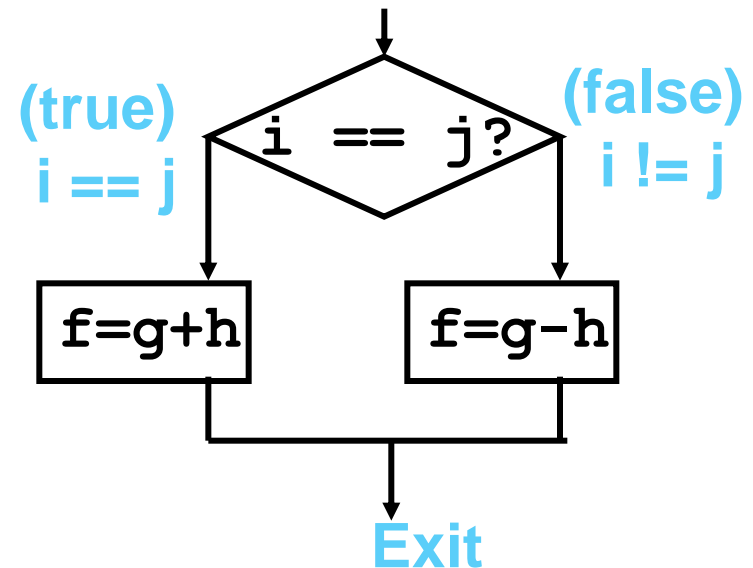

Compilação de um `if` em C (1/2)

- Compile à mão

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

- Assumindo o seguinte mapeamento variável-registo:

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



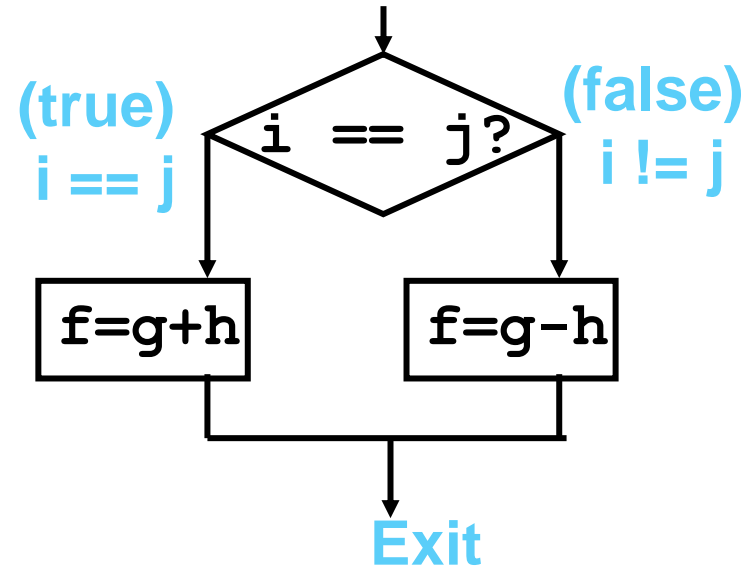
Compilação de um `if` em C (1/2)

- Compile à mão

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

- Código em assembly para MIPS:

	<code>beq \$s3,\$s4,True</code>	<code># branch i==j</code>
	<code>sub \$s0,\$s1,\$s2</code>	<code># f=g-h (false)</code>
	<code>j Fim</code>	<code># goto Fim</code>
<code>True:</code>	<code>add \$s0,\$s1,\$s2</code>	<code># f=g+h (true)</code>
<code>Fim:</code>		



Nota: O compilador em C cria automaticamente labels quando aparecem instruções de decisão (branches).

Ciclos (Loops) em C/Assembly (1/4)

- Em C um ciclo for tem a seguinte forma:

```
for (i=0 ; i<10 ; i++) {  
    ...  
}
```

- Este ciclo é equivalente a um ciclo while:

```
i=0 ;  
while (i<10) {  
    ...  
    i++  
}
```

Em Assembly do MIPS:

```
li $t0,0;      # i=0  
li $t1,10;     # 10  
loop:  
    beq $t0,$t1,fim  
    ...  
    addi $t0,$t0,1 # i++  
    j loop  
fim:  
    ...
```

Ciclos (Loops) em C/Assembly (2/4)

- Exemplo em C: considerando que `A[]` é uma tabela do tipo `int`

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

- Reescrevendo de uma forma deselegante:

Loop:

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

- Assumindo o seguinte mapeamento variável-registo:

```
g,      h,      i,      j,  base da tabela A
$S1,    $S2,    $S3,    $S4,  $S5
```

Ciclos (Loops) em C/Assembly (3/4)

- Código compilado para MIPS:

```

Loop:    mul  $t1, $s3, 4           #$t1= 4*i
         add  $t1, $t1, $s5        #$t1=addr A
         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 (guia):

```

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

Ciclos (Loops) em C/Assembly (4/4)

- Existem 3 tipos diferentes de ciclos em C:
 - `while`
 - `do... while`
 - `For`
- Cada um destes ciclos pode ser re-escrito usando um dos outros dois. Assim o método utilizado para o `do... while` pode ser também usado para implementar o `while` e `for`.
- **Ideia Chave:** Apesar de existirem diferentes formas de construir um ciclo em MIPS, todos eles passam por tomar uma decisão com um **conditional branch**

Desigualdades no MIPS (1/4)

- Até agora só trabalhámos com igualdades (`==` e `!=` no C). No entanto um programa também trabalha com desigualdades (`<` e `>` em C).
- Instruções de desigualdade no MIPS :
 - “Set on Less Than”
 - Sintaxe: `slt reg1, reg2, reg3`
 - Significado:

```
if (reg2 < reg3)
    reg1 = 1;
else
    reg1 = 0;
```

“set” significa “set to 1”,
“reset” significa “set to 0”.

Desigualdades no MIPS (2/4)

- Compile “à mão” o seguinte código

```
if (g < h) goto Less; # assume g:$s0, h:$s1
```

- O resultado em assembly para o MIPS é ...

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h  
bne $t0,$0,Less # goto Less  
# if $t0!=0  
# (if (g<h)) Less:
```

- O registo \$0 contém sempre o valor 0, e por isso é frequentemente utilizado com bne e beq depois de uma instrução slt.
- O par de instruções slt → bne significa if (... < ...) goto...

Desigualdades no MIPS (3/4)

- Com o `slt` podemos implementar “ $<$ ” ! Mas como será que podemos implementar o $>$, \leq e \geq ?
- Poderiam existir mais 3 instruções similares, mas a filosofia do MIPS: é *Simpler is Better, Smaller is faster*
- Será que podemos implementar o \geq usando unicamente o `slt` e “branches”?
- E quanto ao $>$?
- E o \leq ?

Desigualdades no MIPS (4/4)

```
                                # a:$s0, b:$s1
slt  $t0, $s0, $s1             # $t0 = 1 if a < b
beq  $t0, $0, skip             # skip if a >= b
    <stuff>                     # do if a < b
```

skip:

Existem sempre duas variações:

Usar `slt $t0, $s1, $s0` em vez de `slt $t0, $s0, $s1`

Usar `bne` em vez de `beq`

Exercício

Qual dos seguintes segmentos de código em C reproduz mais fielmente o ciclo em *assembly* indicado em baixo?

1. `while (($s3 >= $s4) || ($s5 < $s4)) { $s4--; }`

2. `while (($s3 >= $s4) && ($s5 < $s4)) { $s4--; }`

3. `while (($s3 < $s4) || ($s5 >= $s4)) { $s4--; }`

4. `while (($s3 < $s4) && ($s5 < $s4)) { $s4--; }`

```
loop:
    slt        $t0, $s3, $s4
    bne        $t0, $0, out
    slt        $t1, $s5, $s4
    beq        $t1, $0, out
    addi       $s4, $s4, -1
    j loop
out:
```



Desigualdades e Imediatos

- Existe também uma versão do `slt` para trabalhar com argumentos imediatos (constantes) : `slti`

– Útil em ciclos `for`

```
if (g >= 1) goto Loop
```

C `Loop: . . .`

MIPS	<code>slti \$t0,\$s0,1</code>	<code># \$t0 = 1 if</code>
		<code># \$s0<1 (g<1)</code>
	<code>beq \$t0,\$0,Loop</code>	<code># goto Loop</code>
		<code># if \$t0==0</code>
		<code># (if (g>=1)</code>

O par `slt` → `beq` significa em C `if (... ≥ ...) goto...`

E quanto aos números sem sinal?

- Existe ainda uma instrução de desigualdade para trabalhar com números sem sinal (**unsigned**) :

`sltu, sltiu`

...que coloca o registo de output a 1 (set) ou 0 (reset) em função de uma comparação sem sinal

- Qual é o valor de `$t0` e `$t1`?

(`$s0 = FFFF FFFAhex`, `$s1 = 0000 FFFAhex`)

`slt $t0, $s0, $s1`

`sltu $t1, $s0, $s1`

Signed/Unsigned tem diferentes significados!

- Os termos *Signed/Unsigned* estão “sobre utilizados”. É preciso ter cuidado com os seus múltiplos significados
 - Faz / Não faz extensão de sinal
(lb, lbu)
 - Não detecta overflow
(addu, addiu, subu, multu, divu)
 - Faz comparação com/sem sinal
(slt, slti/sltu, sltiu)

Exemplo: O Switch do C (1/3)

- Escolha entre quatro alternativas diferentes em função de k ter os valores 0, 1, 2 ou 3. Compile “à mão” o seguinte código em C:

```
switch (k) {  
    case 0: f=i+j; break; /* k=0 */  
    case 1: f=g+h; break; /* k=1 */  
    case 2: f=g-h; break; /* k=2 */  
    case 3: f=i-j; break; /* k=3 */  
}
```

Exemplo: O Switch do C (2/3)

- Isto representa uma sequência de instruções complicada, portanto o primeiro passo é simplificar.
- Escreva a sequência como uma cadeia de declarações if-else, as quais já sabemos compilar:

```
if (k==0) f=i+j;  
    else if (k==1) f=g+h;  
        else if (k==2) f=g-h;  
            else if (k==3) f=i-j;
```

- Assumindo o seguinte mapeamento:

```
f:$s0, g:$s1, h:$s2,  
i:$s3, j:$s4, k:$s5
```


Exemplo: O Switch do C (3/3)

- O código compilado é:

```

    bne $s5,$0,L1      # branch k!=0
    add $s0,$s3,$s4    #k==0 so f=i+j
    j    Exit          # end of case so Exit
L1: addi $t0,$s5,-1    # $t0=k-1
    bne $t0,$0,L2      # branch k!=1
    add $s0,$s1,$s2    #k==1 so f=g+h
    j    Exit          # end of case so Exit
L2: addi $t0,$s5,-2    # $t0=k-2
    bne $t0,$0,L3      # branch k!=2
    sub $s0,$s1,$s2    #k==2 so f=g-h
    j    Exit          # end of case so Exit
L3: addi $t0,$s5,-3    # $t0=k-3
    bne $t0,$0,Exit    # branch k!=3
    sub $s0,$s3,$s4    #k==3 so f=i-j
Exit:

```

QUIZ

```
Loop: addi $s0, $s0, -1    # i = i - 1
      slti $t0, $s1, 2    # $t0 = (j < 2)
      beq  $t0, $0, Loop  # goto Loop if $t0 == 0
      slt  $t0, $s1, $s0  # $t0 = (j < i)
      bne  $t0, $0, Loop  # goto Loop if $t0 != 0
```

(\$s0=i, \$s1=j)

Indique o que deveria estar na zona com os pontos de interrogação!

```
do {i--;} while(???) ;
```

0	:	:	<	2	&&&	:	<	:
1	:	U	<	2	&&&	U	<	:
2	:	U	<	2	&&&	U	<	:
3	:	U	<	2	&&&	U	<	:
4	:	U	<	2	&&&	U	<	:
5	:	U	<	2	—	U	<	:
6	:	U	<	2	—	U	<	:
7	:	U	<	2	—	U	<	:
8	:	U	<	2	—	U	<	:
9	:	U	<	2	—	U	<	:

Concluindo

- Os branches permitem tomar a decisão do que vai ser executado em “runtime” em vez de “compile time”.
- As decisões em C são feitas usando conditional statements como o `if`, `while`, `do while`, `for`.
- As decisões em MIPS são feitas usando conditional branches: `beq` e `bne`.
- Para complementar os conditional branches em decisões que envolvam desigualdades, vimos as instruções “Set on Less Than”: `slt`, `slti`, `sltu`, `sltiu`
- Novas instruções que vimos:
`beq`, `bne`, `j`, `slt`, `slti`, `sltu`, `sltiu`

Para saber mais ...

- P&H - Capítulos 2.1, 2.2, 2.3, 2.5 e 2.6
- P&H - Capítulo 3.3

