Arquitectura de Computadores

MIPS : Desigualdades e Funções

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

Recordando…

- A memória é endereçada ao byte , mas o `lw` e o `sw` acedem à memória uma word de cada vez.
- O apontador (usado pelo `lw` e o `sw`) é apenas um endereço de memória, portanto podemos usa-lo em somas e subtracções (com o deslocamento, por exemplo).
- Uma instrução de decisão permite decidir o que executar em tempo real e não durante a escrita do programa.
- As decisões em C são efectuadas usando `if`, `while`, `do while`, `for`.
- No MIPS as instrução para decisões são os saltos condicionais `beq` e `bne` e o salto incondicional `j`

Desigualdades no MIPS (1/5)

- Até agora só testamos igualdades (== e != em C). Nos programas também temos que testar < e > .
- Instrução MIPS para desigualdade:
  - "Set on Less Than"
  - Sintaxe: `slt reg1,reg2,reg3`
  - Quer dizer:

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

  - Em assembler, "set" quer dizer "set to 1", "reset" quer dizer "set to 0".

Desigualdades no MIPS (2/5)

- Como se usa? Compilando à mão:

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

- Resposta: assembler do MIPS…

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

- Salta se `$t0 != 0` ➔ (`g < h`)
- O registo `$0` contém sempre o valor `0`, portanto o `bne` e o `beq` usam-no frequentemente depois de uma instrução `slt`.
- Um par `slt` ➔ `bne` representa `if(… < …)goto…`

Desigualdades no MIPS (3/5)

- Agora, podemos implementar <, mas como implementar >, ≤ and ≥ ?
- Podemos adicionar mais 3 instruções, mas:
  - Objectivo do MIPS : "Simpler is Better"
- Será que podemos implementar ≤ usando slt e os saltos condicionais?
- Que tal o >?
- E o ≥?

Desigualdades no MIPS (4/5)

- Talvez… Compilando à mão:

```
if (g <= h) goto LorEQ;
#g:$s0,h:$s1
```

- Resposta: assembler do MIPS…

```
    slt $t0,$s1,$s0    #$t0=1 if $s1<$s0 (h<g)
                       #$t0=0 if h>=g (= g<=h)
    beq $t0,$0,LoEQ    #goto LorEQ if $t0==0
LorEQ:
```

Desigualdades no MIPS (5/5)

```
(a) if (g > h) goto GTER;
(b) if (g >= h) goto GorEQ;
#g:$s0, h:$s1
```

- (a) Resposta: assembler do MIPS…

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

- (b) Resposta: assembler do MIPS…

```
slt $t0,$s0,$s1    #$t0=1 if $s0<$s1 (g<h)
                   #$t0=0 if g>=h
beq $t0,$0,GorEQ   #goto GorEQ if $t0==0
```

Constantes em Desigualdades

- Também existe uma versão do `slt` para fazer testes com constantes: `slti`
  - Ajuda nos ciclos `for`

C

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

MIPS

```
Loop:  . . .
slti $t0,$s0,1     # $t0 = 1 if
                   # $s0<1 (g<1)
beq  $t0,$0,Loop   # goto Loop
                   # if $t0==0
                   # (if (g>=1))
```

Um par `slt` ➔ `beq` representa `if(… ≥ …)goto…`

E para números sem sinal?

° Também existem instruções de desigualdade sem sinal :

`sltu, sltiu`

...Que colocam o resultado a 0 ou a 1 com base em comparações sem sinal

° Qual o valor de `$t0`, `$t1`?

(`$s0 = FFFF FFFA`$_{hex}$, `$s1 = 0000 FFFA`$_{hex}$)

```
slt $t0, $s0, $s1
sltu $t1, $s0, $s1
```

MIPS: Com sinal vs. Sem sinal

° Atenção! O significado do "u" depende da instrução em causa....

- Estender ou não o sinal

  (`lb`, `lbu`)

- Com/Sem overflow

  (`add`, `addi`, `sub`, `mult`, `div`)
  (`addu`, `addiu`, `subu`, `multu`, `divu`)

- Comparar números com/sem Sinal
  (`slt`, `slti`/`sltu`, `sltiu`)

Questão...

```
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)

Qual o código C que preenche correctamente o espaço?

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

1:	j	<	2	&&	j	<	i
2:	j	<=	2	&&	j	<	i
3:	j	<	2	&&	j	<=	i
4:	j	<=	2	&&	j	<=	i
5:	j	>	2	&&	j	<	i
6:	j	<=	2	||	j	<	i
7:	j	<=	2	||	j	<=	i
8:	j	>=	2	||	j	<=	i
9:	j	>	2	||	j	<	i
0:	j	>	2	||	j	<	i

Pseudoinstruções...

Branch on less than

`blt rsrc1, rsrc2, label` *pseudoinstruction*

Branch on less than unsigned

`bltu rsrc1, rsrc2, label` *pseudoinstruction*

Conditionally branch to the instruction at the label if register `rsrc1` is less than `rsrc2`.

Branch on greater than

`bgt rsrc1, src2, label` *pseudoinstruction*

Branch on greater than unsigned

`bgtu rsrc1, src2, label` *pseudoinstruction*

Conditionally branch to the instruction at the label if register `rsrc1` is greater than `src2`.

Pseudoinstruções...

Branch on less than equal

```
ble rsrc1, src2, label        pseudoinstruction
```

Branch on less than equal unsigned

```
bleu rsrc1, src2, label       pseudoinstruction
```

Conditionally branch to the instruction at the label if register `rsrc1` is less than or equal to `src2`.

Branch on greater than equal

```
bge rsrc1, rsrc2, label       pseudoinstruction
```

Branch on greater than equal unsigned

```
bgeu rsrc1, rsrc2, label      pseudoinstruction
```

Conditionally branch to the instruction at the label if register `rsrc1` is greater than or equal to `rsrc2`.

Exercício

- **Escreva o código MIPS do seguinte programa em C:**

```
main()
{
   int i=0, j=20, s=0;
   while(i!=j) {
        s=s+i;
        i++;
   }
    for(i=0;i<j;i++) s=s*2;
}
```

Use o mapeamento: i=$s0, j=$s1, s=$s2

Resposta...

```
            .data
            .text
            .globl  main
main:       add  $s0,$0,$0        # i=0
            addi $s1,$0,20        # j=20
            addi $s2,$0,0         # s=0
loop1:      beq  $s0,$s1,end1     # if(i=j)goto end
            add  $s2,$s2,$s0      # s=s+i
            addi $s0,$s0,1        # i=i+1
            j loop1
end1:       add $s0,$0,$0          # i=0
loop2:      slt  $t0,$s0,$s1      # $t0=1 if i<j
            beq  $t0,$0,end2      # if $t0=0 goto END2
            sll  $s2,$s2,1        # s=s*2
            addi $s0,$s0,1        # i=i+1
            j loop2
end2:       li $v0, 10
            syscall
```

Suporte para procedimentos

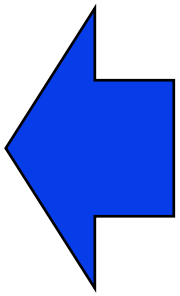
- Os registos são muito importantes para gerir os dados necessários nas chamadas a funções.
- **Convenções**:
  - Endereço de retorno `$ra`
  - Argumentos `$a0,$a1,$a2,$a3`
  - Valores de retorno `$v0, $v1`
  - Variáveis locais `$s0,$s1, … , $s7`
- A pilha também é usada como veremos..

Instruções de suporte a procedimentos (1/6)

C
```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
   return x+y;
}
```

MIPS
```
address
1000
1004
1008
1012
1016

2000
2004
```



No MIPS, todas as instruções ocupam 4 bytes, e são guardadas em memória tal como os dados. Portanto aqui mostramos os endereços em que as instruções vão ser guardadas.

Instruções de suporte a procedimentos (2/6)

C
```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
   return x+y;
}
```

MIPS
```
address
1000 add  $a0,$s0,$zero  # x = a
1004 add  $a1,$s1,$zero  # y = b
1008 addi $ra,$zero,1016 #$ra=1016
1012 j    sum      #salta para sum
1016 ...

2000 sum: add $v0,$a0,$a1
2004 jr   $ra  # nova instrução
```

Instruções de suporte a procedimentos (3/6)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
   return x+y;
}
```

MIPS

- Pergunta: Porquê usar `jr` aqui? Porque não usar simplesmente `j`?
- Resposta: O procedimento `sum` pode ser chamado por muitas funções, logo não podemos retornar para um local fixo! A função que chama `sum` tem que poder dizer "retorna para aqui ".

```
...0 sum: add $v0,$a0,$a1
...4 jr   $ra  # nova instrução
```

Instruções de suporte a procedimentos (4/6)

- Uma única instrução para saltar e guardar o endereço de retorno: jump and link (`jal`)
- **Antes**:

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum               #goto sum
```

- **Agora**:

```
1008 jal sum  # $ra=1012,goto sum
```

- Porquê `jal`? Para aumentar o desempenho: as chamadas a funções são muito comuns. Para além disso, com o `jal` não precisamos de conhecer os endereços de memória do código…

Instruções de suporte a procedimentos (5/6)

- A sintaxe do `jal` (jump and link) é a mesma do `j` (jump):

```
jal label
```

- `jal` deveria chamar-se `laj` para "link and jump":
  - Passo 1 (link): Guardar o endereço da *próxima* instrução no `$ra` (Porquê a próxima instrução? Porque não a corrente?)
  - Passo 2 (jump): Salta para a "label" indicada

Instruções de suporte a procedimentos (6/6)

- Sintaxe do `jr` (jump register):

```
jr registo
```

- Em vez de indicar uma "label" para a qual saltar, a instrução `jr` indica um registo que contém o endereço para o qual saltar.
- Só é útil se conhecermos o endereço exacto para onde queremos saltar.
- Muito útil para chamadas a funções:
  - `jal` guarda o endereço de retorno no registo (`$ra`)
  - `jr $ra` salta de volta para esse endereço

Procedimentos Encadeados (1/2)

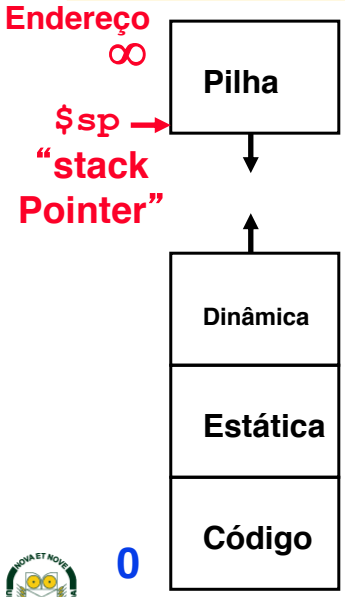
```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- procedimento `sumSquare` chama a função `mult`.
- Portanto há um endereço no `$ra` para o qual `sumSquare` vai ter que retornar, mas que será perdido quando `sumSquare` chamar `mult`!
- É necessário guardar o endereço de retorno de `sumSquare` antes de chamar `mult`.

Procedimentos Encadeados (2/2)

- Em geral é necessário guardar mais alguma informação para além do `$ra`.
- Quando um programa em C é executado há 3 zonas de memória alocadas:
  - **Estática**: variáveis declaradas uma vez no programa e que apenas desaparecem quando o programa termina (ex: globais)
  - **Dinâmica**: variáveis dinâmicas
  - **Pilha**: Espaço usado pelos procedimentos durante a execução; é aqui que podemos guardar os valores dos registos…

Alocação de memória em C



Espaço para guardar Informação dos procedimentos

Espaço alocado explicitamente, ex: malloc(); apontadores C

Variáveis declaradas uma vez no programa

Programa

Usando a Pilha (1/2)

- Portanto temos um registo, `$sp` que aponta sempre para o último espaço usado na pilha
- Para usar a pilha, decrementamos este apontador do espaço que pretendemos usar e depois guardamos a informação
- Portanto como traduzir isto?

```
int sumSquare(int x, int y) {
  return mult(x,x)+ y;
}
```

Usando a Pilha (2/2)

- Compilando...

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

```
sumSquare:
"push"  addi $sp,$sp,-8  # space on stack
        sw $ra, 4($sp)   # save ret addr
        sw $a1, 0($sp)   # save y

        add $a1,$a0,$zero # mult(x,x)
        jal mult          # call mult

        lw $a1, 0($sp)   # restore y
        add $v0,$v0,$a1  # mult()+y
        lw $ra, 4($sp)   # get ret addr
"pop"   addi $sp,$sp,8   # restore stack
        jr $ra
mult:   ...
```

Passos para invocar um procedimento

1) Guardar os valores a preservar na Pilha.
2) Atribuir o(s) argumento(s), se for o caso.
3) Chamar `jal`
4) Restaurar os valores da pilha

Regras para os procedimentos

- Invocados com a instrução `jal`, retornam com `jr $ra`
- Aceitam até 4 argumentos `$a0`, `$a1`, `$a2` e `$a3`
- Retornam valores sempre em `$v0` (e se necessário em `$v1`)
- Necessário seguir regras de utilização de registos (mesmo em funções que apenas você vai invocar)! Quais regras?

Funções & Registos (1/4)

- "Função chamadora": A função que chama outra
- "Função chamada": A função que é chamada
- Quando a função chamada retorna, a chamadora necessita saber quais os registos que poderão ter sido alterados e quais os que garantidamente se mantêm inalterados.
- Convenções: Um conjunto de regras indicando quais os registos que não são alterados depois da invocação de um procedimento (`jal`) e quais os que podem ter sido alterados.

Funções & Registos (2/4) - saved

- `$0`: Não muda. Sempre 0.
- `$s0-$s7`: Restaurar se alterar. Muito importante! É por isso que tem o nome de "saved registers". Se a função chamada os alterar de alguma forma, tem que restaurar os valores originais antes de retornar.
- `$sp`: Restaurar se alterar. O "stack pointer" tem que apontar para a mesma posição antes e depois da instrução `jal`, caso contrário a função chamadora não poderá repôr valores da pilha!
- Sugestão – Todos estes registos começam por S!

Funções & Registos (3/4) - voláteis

- `$ra`: Pode mudar. A instrução `jal` vai mudar este registo. A função chamadora necessita de o guardar na pilha se vai invocar outras funções.
- `$v0-$v1`: Podem mudar. Estes registos vão conter os valores de retorno da função.
- `$a0-$a3`: Podem mudar. Estes são registos voláteis usados para passar argumentos a funções.
- `$t0-$t9`: Podem mudar. É por isso que são chamados temporários: qualquer função pode alterá-los em qualquer momento.

A função chamadora tem que guardar estes registos se precisar deles após a chamada.

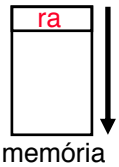
Funções & Registos (4/4)

- Qual o significado destas regras?
  - Se a função R chama a função A, então a função R tem que guardar na pilha os registos temporários que esteja a usar antes de executar a instrução `jal`.
  - A função A tem que guardar qualquer registo S ("saved") que necessite de usar antes de apagar o seu conteúdo.
  - Lembrar: Função chamadora/chamada necessita apenas de guardar os registos temporários/ "saved" que esteja a usar, e não todos os registos.

Estrutura básica de uma função

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)   # guardar $ra
guardar outros regs se necessário
```

Corpo ··· (chamar outras funções…)




```
Restaurar outros regs se necessário
lw $ra, framesize-4($sp)   # restaurar $ra
addi $sp,$sp, framesize
jr $ra
```

Exemplo: Números de Fibonacci 1/7

- Os números de fibonacci estão definidos como :

  F(n) = F(n – 1) + F(n – 2),
  F(0) = F(1) = 1

- Em C fica:

```
int fib (int n) {
 switch (n){
     case 0: return 1;
     case 1: return 1;
     default: return (fib(n-1)+fib(n-2));
 }
}
```

Exemplo: Números de Fibonacci 2/7

- Traduzindo para MIPS!
- Vamos necessitar de espaço para 3 "words" na Pilha
- A função usa o registo $s0
- O Prólogo:

```
fib:
addi $sp, $sp, -12      # Espaço para 3 "words"
sw $ra, 8($sp)          # guardar $ra
sw $s0, 4($sp)          # guardar $s0
```

Exemplo: Números de Fibonacci 3/7

° Agora o Epílogo:

```
fin:
   lw $s0, 4($sp)      # repôr o $s0
   lw $ra, 8($sp)      # repôr o  $ra
   addi $sp, $sp, 12   # repôr o $sp
   jr $ra              # Saltar para a chamadora
```

Exemplo: Números de Fibonacci 4/7

° Agora o corpo. Considere o código em C, começe pelas linhas com os comentários…

```
int fib(int n) {
 if(n == 0) { return 1; } /*Traduz-me!*/
 if(n == 1) { return 1; } /*Traduz-me!*/
 return (fib(n - 1) + fib(n - 2));

}
```

```
addi $v0, $zero, 1    # $v0 = 1
beq $a0, $zero, fin   # if (n == 0). . .
addi $t0, $zero, 1    # $t0 = 1
beq $a0, $t0 , fin    # if (n == 1). . .
```

Continua…

Exemplo: Números de Fibonacci 5/7

° Quase lá…mas esta parte é mais complicada!

```
int fib(int n) {
  . . .
 return (fib(n - 1) + fib(n - 2));
}
```

```
addi $a0, $a0, -1     #$a0 = n-1
sw $a0, 0($sp)        #guardar $a0
jal fib               #fib(n - 1)
lw $a0, 0($sp)        #Restaurar $a0
addi $a0, $a0, -1     #$a0 = n-2
```

Continua…

Exemplo: Números de Fibonacci 6/7

° Lembrar que o chamador é que preserva $v0 !

```
int fib(int n) {
  . . .
 return (fib(n - 1) + fib(n - 2));
}
```

```
add $s0, $v0, $zero   # guarda fib(n – 1)
                      # num registo que é
                      # preservado ($s0)
jal fib               #  fib(n – 2)
add $v0, $v0, $s0     #  $v0 = fib(n-1) + fib(n-2)
```

Para o epílogo…

Exemplo: Números de Fibonacci 7/7

° Código completo:

```
fib:
addi $sp, $sp, -12
sw $ra, 8($sp)
sw $s0, 4($sp)
addi $v0, $zero, 1
beq $a0, $zero, fin
addi $t0, $zero, 1
beq $a0, $t0, fin
addi $a0, $a0, -1
sw $a0, 0($sp)
jal fib
add $s0, $v0, $zero
lw $a0, 0($sp)
addi $a0, $a0, -1
jal fib
add $v0, $v0, $s0
fin:
lw $s0, 4($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra
```

Registos do MIPS

A constante 0	$0	$zero
Reserv. para o assemblador	$1	$at
Valores de retorno	$2-$3	$v0-$v1
Argumentos	$4-$7	$a0-$a3
Temporarios	$8-$15	$t0-$t7
Guardados (Saved)	$16-$23	$s0-$s7
Mais temporários	$24-$25	$t8-$t9
Usados pelo "Kernel"	$26-27	$k0-$k1
Apontador Global	$28	$gp
Apontador da pilha	$29	$sp
"Frame Pointer"	$30	$fp
Endereço de retorno	$31	$ra

Outros registos

- `$at`: pode ser usado pelo assemblador em qualquer altura; não deve ser usado
- `$k0-$k1`: podem ser usados pelo SO em qualquer altura; não devem ser usados.
- `$gp`, `$fp`: Não vamos usar…
- Nota: Podem ler sobre o `$gp` and `$fp` no Apêndice A, mas podem escrever bom código MIPS sem os usar…

"E em conclusão…"

- Funções chamam-se com `jal`, retornam com `jr $ra`.
- A pilha é uma ajuda! Pode usa-la para guardar qualquer coisa que precise. É necessário deixá-la como a encontrou!!!!!
- Instruções conhecidas…
  - Aritméticas: `add,addi,sub,addu,addiu,subu`
  - Memória: `lw, sw`
  - Decisão: `beq, bne, slt, slti, sltu, sltiu`
  - Saltos incondicionais ("Jumps"): `j, jal, jr`
- Registos:
  - Todos!