



Arquitectura dos
Computadores

Aula 5 –
Funções

Professor: Dr. Christophe Soares

Suporte para procedimentos

Os registos são muito importantes para gerir os dados necessários nas chamadas a funções

Convenções:

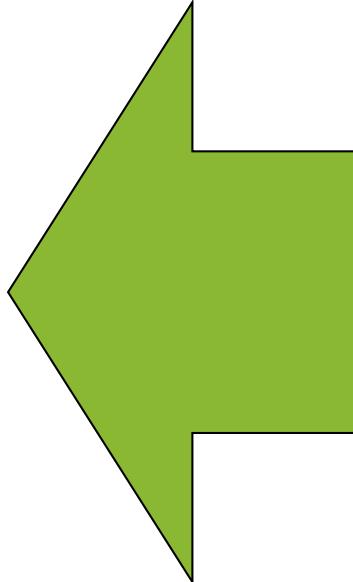
- | | |
|-----------------------|-----------------------|
| ✓ Endereço de retorno | \$ra |
| ✓ Argumentos | \$a0,\$a1,\$a2,\$a3 |
| ✓ Valores de retorno | \$v0, \$v1 |
| ✓ Variáveis locais | \$s0,\$s1, ... , \$s7 |

Instruções de suporte a procedimentos (1/6)

C sum(a,b);
 (...)

int sum(int x, int y) {
 return x+y;
}

	address
M	1000
I	1004
P	1008
S	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);
 (...)

int sum(int x, int y) {
 return x+y;
}

address			
M 1000	add	\$a0,\$s0,\$zero	# x = a
I 1004	add	\$a1,\$s1,\$zero	# y = b
P 1008	addi	\$ra,\$zero,1016	# \$ra=1016
S 1012	j	sum	# go to sum 1016
		(...)	
2000	sum:	add \$v0,\$a0,\$a1	
2004	jr	\$ra	# new instruction

Instruções de suporte a procedimentos (3/6)

C sum(a,b);
 (...)

int sum(int x, int y) {
 return x+y;
}

Porquê usar *jr* aqui? Porque não usar simplesmente *j*?

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”.

**2000
2004**

sum: add \$v0,\$a0,\$a1
 jr \$ra



```
# new instruction
```

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                # go to sum 1016
```

Agora:

```
1008 jal sum                  # $ra=1012, go to sum
```

Usando o *jal* não precisamos de conhecer os endereços de memória do código para especificar o *\$ra*

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*”:

- **1º Passo (*link*):** Guarda o endereço da *próxima* instrução no \$ra (Porquê a *próxima*?)
- **2º Passo (*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 onde saltar, a instrução *jr* indica um registo que contém o endereço de destino

Usado para chamadas a funções:

jal guarda o endereço de retorno no \$ra

jr \$ra salta de volta para esse endereço

Procedimentos Encadeados (1/2)

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

Existe um endereço no *\$ra* para o qual *sumSquare* irá ter que retornar, mas que será perdido quando *sumSquare* evocar *mult*!

Em procedimentos encadeados, será necessário guardar o endereço de retorno

Procedimentos Encadeados (2/2)

Em geral, é necessário guardar mais 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

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...

sumSquare:	addi \$sp,\$sp,-8 sw \$ra, 4(\$sp) sw \$a1, 0(\$sp)	# space on stack # save ret addr # save y
	add \$a1,\$a0,\$zero jal mult	# mult(x,x) # call mult
	lw \$a1, 0(\$sp) add \$v0,\$v0,\$a1	# restore y # mult() + y
“pop”	lw \$ra, 4(\$sp) addi \$sp,\$sp,8 jr \$ra	# get ret addr # restore stack
mult:	(...)	

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

Exercício 1 (1/2)

Como ficaria o *mult*?

Pensem nas convenções!

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

```
int mult(int a, int b){  
    return a*b;  
}
```

Exercício 1 (1/2)

MIPS:

```
mult:    mul $v0, $a0, $a1  
           jr $ra
```

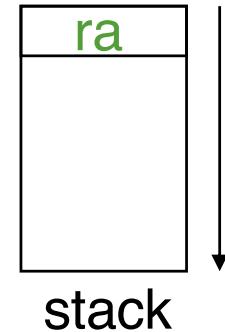
Código em C:

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

```
int mult(int a, int b){  
    return a*b;  
}
```

Estrutura de uma função

```
entry_label:    addi $sp,$sp, -framesize
                sw $ra, framesize-4($sp) # store $ra
                # store other reg (if required)
                (...) # remaining code
                # call other functions (if required)
                # restore other reg (if required)
                lw $ra, framesize-4($sp) # restore $ra
                addi $sp,$sp, framesize # restore SP
                jr $ra
```



Regras para os procedimentos

Invocados com o *jal*, retornam com *jr \$ra*

**Aceitam-se até 4 argumentos \$a0, \$a1, \$a2 e
\$a3**

Retornam valores sempre em \$v0

Existem regras de utilização de registos! Quais?

Funções & Registos (1/4)

“Função chamadora”: A função que chama outra

“Função chamada”: A função que é chamada

Convenções: regras indicam quais os registos que não são alterados depois da invocação de um procedimento (jal) e quais os que podem ter sido alterados

Funções & Registros (2/4)

\$0: Não muda. Sempre 0.

\$s0-\$s7: Restaurar se alterar. 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 repor valores da pilha!

\$t0-\$t9: Podem mudar. É por isso que são chamados temporários: qualquer função pode alterá-los em qualquer momento

Funções & Registos (3/4)

`$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.

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 essenciais 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 alterar o seu conteúdo
- Atenção: Função chamadora/chamada necessita apenas de guardar os registos temporários/ “saved” que esteja a usar, e não todos os registos!

Analogia: Sozinhos em casa (1/5)

Pais (main) vão passar fora o fim-de-semana

Eles (chamador) entregam a chave de casa ao filho (chamado) e impõe regras (convenções):

- Podes desarrumar os divisões temporárias, como a sala de jogos e a cave (registos) se quiseres,
- MAS é bom que deixes os quartos (registos) que queremos guardar para as visitas intocados. “É bom que estejam iguais quando regressarmos!”

Analogia: Sozinhos em casa (2/5)

O filho agora “controla” todas as divisões da casa
(registos)

E pretende usar os quartos das **visitas** para uma festa
(computação)

O que deve ele (**chamado**) fazer?

- Pega em tudo o que está nestes quartos e coloca na garagem (**memória**)
- Inicia a festa, **desarruma tudo** (exceto a garagem...)
- Restaura os quartos que os pais querem **intocados** depois da festa recolocando as coisas guardadas na garagem
(memória)

Analogia: Sozinhos em casa (3/5)

Repetimos este cenário, exceto que ANTES do regresso dos pais e da reposição dos quartos **das visitas**...

Filho (**chamado**) deixou valores (**dados**) em toda a casa

- Um amigo (**outro chamado**) pretende usar a casa para outra festa quando o filho está ausente
- O filho sabe que o amigo poderá **desarrumar** toda a casa destruindo as suas coisas
- O filho (**agora chamador**) lembra-se das regras (**convenções**) impostas pelos pais e vai fazer o mesmo ao amigo

Analogia: Sozinhos em casa (4/5)

Se o filho tem coisas nas divisões temporárias (que vão ser desarrumados), tem 3 opções:

- Movê-las para a garagem (**memória**)
- Movê-las para os quartos das visitas cujo conteúdo original já se se encontra na garagem (**memória**)
- Melhorar o seu estilo de vida (**código**) de forma a que a quantidade de coisas que tem que movimentar de e para a garagem (**memória**) seja minimizado

Analogia: Sozinhos em casa (5/5)

Amigo agora “controla” as divisões (**registos**)

E pretende usar os quartos das visitas para uma grande festa (**computação**)

O que deve ele (**chamado**) fazer?

- O amigo pega em tudo o que está nestes quartos e coloca na garagem (**memória**)
- Inicia a festa, **desarruma tudo** (exceto a garagem)
- Restaura os quartos das visitas recolocando as coisas inicialmente presentes e que estavam guardadas na garagem (**memória**)

Exercício 2 (1/2)

Escrever o código MIPS:

```
main(){
    int i,s=0;
    for(i=0;i<10;i++) {
        s+=soma(i,i+1);
    }
}

int soma(int i, int j){
    return (i+j);
}
```

\$s0 - s
\$s1 - i

Exercício 2 (2/2)

Código do MIPS:

```
.text
    li $s0,0
    li $s1,0
Loop: slti $t0,$s1,10
      beq $t0,$0,End
      move $a0,$s1
      addi $a1,$a0,1
      jal Sum
      add $s0,$s0,$v0
      addi $s1,$s1,1
      j Loop
End:  li $v0,10
      syscall
Sum:  add $v0,$a0,$a1
      jr $ra
```

```
main(){
    int i,s=0;
    for(i=0;i<10;i++){
        s+=soma(i,i+1);
    }
}

int soma(int i, int j){
    return (i+j);
}
```

\$s0 – s
\$s1 – i

\$s0 for s
\$s1 for i
set \$t0 to 1 if \$s1 < 10
if \$t0=0 goto End
set the first parameter to \$a0
set the second parameter to \$a1
link and jump to Sum
use the returned value and add it to \$s0
increment i
goto Loop
Exit
easy function do not have to store in stack
return to the caller

Conclusão...

- Funções chamam-se com jal, retornam com jr \$ra.
- A pilha é uma ajuda! Pode usá-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!