Arquitetura de Computadores I
A arquitetura MIPS – invocação de funções e procedimentos

António de Brito Ferrari
ferrari@ua.pt

11. Invocação de funções/procedures (sub-rotinas)

Invocação de procedimentos

Passos necessários:

1. Colocar os parametros em registos
2. Transferir o controlo para o procedimento
3. Adquirir espaço de memória para o procedimento
4. Executar as instruções do procedimento
5. Colocar o resultado num registo para o passar ao invocador
6. Regressar ao ponto do programa onde foi feita a chamada do procedimento

Utilização dos registos

- $a0 – $a3: argumentos inteiros de uma função (r4 – r7)
- $v0, $v1: result values (r2 e r3)
- $t0 – $t9: temporaries
  - O seu conteúdo pode ser destruído pelo ***callee***
- $s0 – $s7: saved
  - Têm de ser preservados (saved/restored) pelo ***callee***
- $gp: global pointer for static data (r28)
- $sp: stack pointer (r29)
- $fp: frame pointer (r30)
- $ra: return address (r31)

Instruções para invocação de procedimentos

- Procedure call: jump and link
  `jal ProcedureLabel`
  - Coloca endereço da instrução seguinte em $ra
  - Salta para o endereço alvo (`ProcedureLabel`)
- Procedure return: jump register
  `jr $ra`
  - Copia $ra para o program counter **(PC) = ($ra)**
  - Pode tambem ser usado para "computed jumps"
    - e.g., case/switch statements

Procedimento que não invoca outro

- Código C :

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Argumentos **g, …, j** em $a0, …, $a3
  - **f** em $s0 (necessário preservar $s0 no stack)
  - Resultado em $v0

`leaf_example`: código MIPS

`addi $sp, $sp, -4` `sw $s0, 0($sp)`	Guardar $s0 no stack

`add $t0, $a0, $a1`
`add $t1, $a2, $a3`
`sub $s0, $t0, $t1`
	Corpo do procedimento

`add $v0, $s0, $zero`	Resultado
`lw $s0, 0($sp)` `addi $sp, $sp, 4`	Restaurar $s0
`jr $ra`	Regresso ao ***caller***

Procedimentos que invocam outros procedimentos

- Para invocações em cadeia o ***caller*** precisa de guardar no stack:
  - O seu endereço de retorno
  - Valores de argumentos e variáveis temporárias de que necessite depois da invocação
- Restaurar o stack quando o procedimento que invocou retorna

Exemplo 2 – código C

- C:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

  - Argumento **n** em $a0
  - Resultado em $v0

Exemplo 2 – código MIPS

```
fact:
    addi $sp, $sp, -8     # ajusta o stack para 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # teste se n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, resultado = 1
    addi $sp, $sp, 8      #   pop 2 items do stack
    jr   $ra              #   e return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore valor original de n
    lw   $ra, 4($sp)      #   e return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiplicar to get result
    jr   $ra              # return
```


Tipos de funções / procedimentos e respetivas *stack frames*

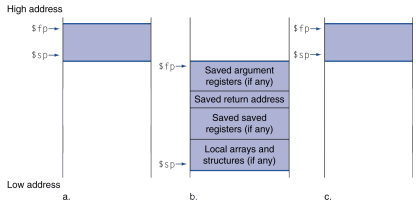
- Os compiladores classificam as rotinas numa das seguintes categorias:

1. Funções que invocam outras funções ou a si próprias (recursivas) – *non-leaf routines*
2. Funções que não invocam outras – *leaf routines*
   a) Requerem espaço no *stack* para variáveis locais
   b) Não requerem espaço no *stack* para variáveis locais

O que é e não é preservado na invocação de uma sub-rotina

Registos preservados	Registos não preservados
$s0 - $s7	$t0 - $t9
Stack pointer $sp	Registos dos argumentos $a0 - $a3
Registo com o endereço de retorno $ra	Registos que retornam valores $v0 - $v1
Conteúdo do stack acima de $sp	Conteúdo do stack abaixo de $sp

Dados locais no stack



Local data allocated by callee
- e.g., C automatic variables

Procedure frame (activation record)
- Usado por alguns compiladores para administrar o stack

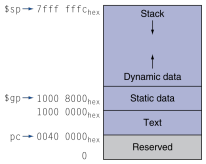
Mapa de memória

Text: program code

Static data: global variables
- e.g., static variables in C, constant arrays and strings
- $gp initialized to address allowing ±offsets into this segment

Dynamic data: heap
- E.g., malloc in C, new in Java

Stack: automatic storage



12. Representação de outros tipos de dados

Carateres

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java (16-bit characters), C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

O Código ASCII

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Operações com ***bytes*** e ***halfwords***

- Podem usar operações bitwise
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

Strings

- 3 alternativas para representar strings:
  1. A primeira posição da string é reservada para armazenar o seu comprimento
  2. Uma variável ligada à string indica o comprimento da string (como numa *structure*) – strings em Java
  3. A útima posição da string é indicada por um carater usado para marcar o fim da string – em C uma string é terminada por um byte cujo valor é zero (o carater Null em ASCII)

Exemplo: String copy

- C:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Endereços de x, y em $a0, $a1
  - i em $s0

String copy: código MIPS

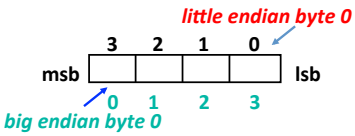
```
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw   $s0, 0($sp)       # save $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
    lbu  $t2, 0($t1)       # $t2 = y[i]
    add  $t3, $s0, $a0     # addr of x[i] in $t3
    sb   $t2, 0($t3)       # x[i] = y[i]
    beq  $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j    L1                # next iteration of loop
L2: lw   $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # and return
```

13. Modos de endereçamento de constantes de 32-bits, *branch* e *jump*

Imediatos de 32-bits

- ***lui*** – load upper immediate
  - Carrega na metade esquerda do registo (16-bits mais significativos) a constante indicada no código de instrução e preenche com zeros os 16-bits da metade direita
- Organização da memória:
  - Low-endian – o byte menos significativo é armazenado no menor endereço (i86)
  - High-endian - o byte menos significativo é armazenado no maior endereço (IBM, Motorola)
  - Bi-endian – a arquitetura permite definir qualquer uma das duas organizações (MIPS, ARM)

Little Endian vs. Big Endian



Little Endian: Intel 80x86, DEC Alpha
Big Endian: HP PA, IBM/Motorola PowerPC, SGI, Sparc

Cohen, D. "On holy wars and a plea for peace (data transmission)." *Computer*, vol.14, (no.10), Oct. 1981. p.48-54

The machine language version of `lui $t0, 255 # $t0 is register 8`:

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255`:

0000 0000 1111 1111	0000 0000 0000 0000

Endereçagem de ***branch***

- As instruções de *branch* especificam
  - Opcode, dois registos, target address
- A maioria dos branch saltam para instruções próximas
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

  - Endereçagem PC-relative
    - Target address = PC + offset × 4
    - PC já a apontar para a instrução seguinte (previamente incrementado de 4)

Endereçagem de ***jump***

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- (Pseudo)Direct jump addressing
  - Target address = $PC_{31...28}$: (address × 4)

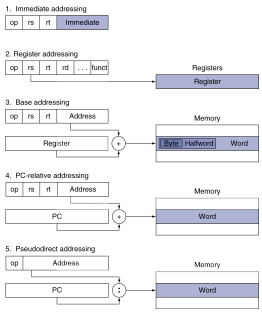
Target Addressing: exemplo

– Assume-se que Loop em 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	4	0	
80004	0	9	22	9	0	32	
80008	35	9	8	0			
80012	5	8	21	2			
80016	8	19	19	1			
80020	2	20000					
80024							

Modos de Endereçamento: sumário



Instruções MIPS já vistas

MIPS instructions	Name	Format
add	add	R
subtract	sub	R
add immediate	addi	I
load word	lw	I
store word	sw	I
load half	lh	I
load half unsigned	lhu	I
store half	sh	I
load byte	lb	I
load byte unsigned	lbu	I
store byte	sb	I
load linked	ll	I
store conditional	sc	I
load upper immediate	lui	I
and	and	R
or	or	R
nor	nor	R
and immediate	andi	I
or immediate	ori	I
shift left logical	sll	R
shift right logical	srl	R
branch on equal	beq	I
branch on not equal	bne	I
set less than	slt	R
set less than immediate	slti	I
set less than immediate unsigned	sltu	I
jump	j	J
jump register	jr	R
jump and link	jal	J

Pseudo MIPS	Name	Format
move	move	R
multiply	mult	R
multiply immediate	multi	I
load immediate	li	I
branch less than	blt	I
branch less than or equal	ble	I
branch greater than	bgt	I
branch greater than or equal	bge	I

13. Arrays e Ponteiros

C Pointers

- Pointer: uma variável que contem o endereço de outra variável
  - Versão em linguagem de alto nível do endereço em linguagem máquina
- Porquê usar Pointers?
  - Por vezes são a única possibilidade de expressar uma computação
  - Código mais compacto e eficiente

Ponteiros em C

- Variável ***c*** tem o valor 100, localisado no endereço de memória 0x10000000
- O operador ***&*** dá o endereço:
  `p = &c;` atribui a ***p*** o endereço de ***c***
  - p "points to" `c: p == 0x10000000`
- O operador ***** dá o valor para que o ponteiro aponta:
  if **`p = &c;`** then **`* p == 100`** - "Dereferencing p"

Ponteiros em Assembly

... = *p; ⇒ ***load***

(obter o valor armazenado na posição apontada por p)

p = ...;** ⇒ ***store

(armazenar o valor na posição apontada por p)

Ponteiros em assembly

int c, tem o valor **100**, no endereço **0x10000000**,

p em **$a0**, **x** em **$s0**

```
# p = &c;   /* p = 0x10000000 */
lui $a0,0x1000 # p = 0x10000000
# x = *p;   /* x = 100 */
lw $s0,0($a0) # $s0 = 100
# *p = 200; /* c = 200 */
addi $t0,$0,200
sw  $t0,0($a0) # c = 200
```


Arrays vs. Ponteiros

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

Ciclo: 6 instruções

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       move $t0,$a0    # p = & array[0]
       sll $t1,$a1,2   # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                       #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4  # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                       #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

Ciclo: 4 instruções

Exemplo: **Bubble Sort**

- Non-leaf (invoca swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j+1];j -= 1) {
      swap(v,j);
    }
  }
}
```

v em $a0, k em $a1, i em $s0, j em $s1

swap

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```


swap em assembly

- **$t0** usado para armazenar **temp**

```
swap:  sll   $t1, $a1, 2
       add   $t1, $a0, $t1  # $t1 = endereço de v[k]
       lw    $t0, 0($t1)   # $t0 (temp) = v[k]
       lw    $t2, 4($t1)   # $t2 = v[k+1]
       sw    $t2, 0($t1)
       sw    $t0, 4($t1)
       jr    $ra
```

Bubble Sort em Assembly

```
sort: addi $sp,$sp, -20      # espaço no stack para 5 registos
      sw $ra, 16($sp)        # save $ra on stack
      sw $s3,12($sp)         # save $s3 on stack
      sw $s2, 8($sp)         # save $s2 on stack
      sw $s1, 4($sp)         # save $s1 on stack
      sw $s0, 0($sp)         # save $s0 on stack
        …                       # procedure body
        …
      exit1: lw $s0, 0($sp)  # restore $s0 from stack
      lw $s1, 4($sp)         # restore $s1 from stack
      lw $s2, 8($sp)         # restore $s2 from stack
      lw $s3,12($sp)         # restore $s3 from stack
      lw $ra,16($sp)         # restore $ra from stack
      addi $sp,$sp, 20       # restore stack pointer
      jr $ra                 # return to calling routine
```


Procedure body

```
        move $s2, $a0          # save $a0 into $s2
        move $s3, $a1          # save $a1 into $s3
        move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3    # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, –1      # j = i – 1
for2tst: slti $t0, $s1, 0      # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2       # $t1 = j * 4
        add  $t2, $s2, $t1     # $t2 = v + (j * 4)
        lw   $t3, 0($t2)       # $t3 = v[j]
        lw   $t4, 4($t2)       # $t4 = v[j + 1]
        slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
        move $a0, $s2          # 1st param of swap is v (old $a0)
        move $a1, $s1          # 2nd param of swap is j
        jal  swap              # call swap procedure
        addi $s1, $s1, –1      # j –= 1
        j    for2tst           # jump to test of inner loop
exit2:  addi $s0, $s0, 1       # i += 1
        j for1tst              # jump to test of outer loop
```