Introdução à Arquitetura de Computadores

Aula19

Assembly 4: Funções (cont.), *Stack* e *Addressing*

Funções (cont)

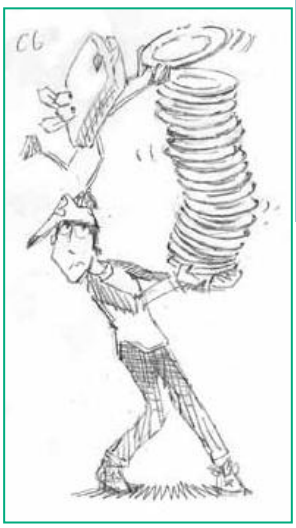
- Uso do *Stack*
- Salvaguarda de Registos
- Recursividade*

Modos de Endereçamento (*Addressing*)

- Tipo-R: Só-Registos (add, sub, and)
- Tipo-I: Imediato (addi, xori),
  Endereço-Base (lw, sw),
  PC-Relativo (beq, bne)
- Tipo-J: Pseudo-Direto (j, jal)

3 - Funções (10) - O Stack (A Pilha)

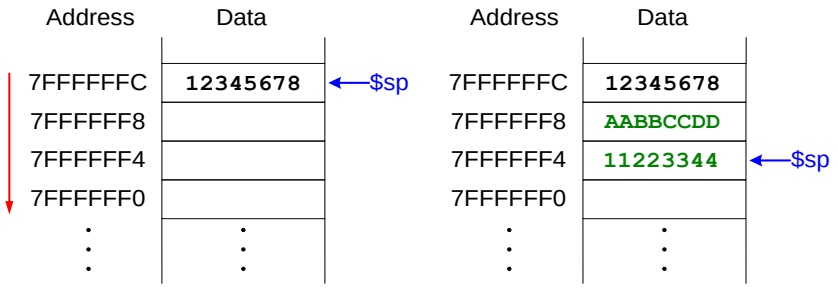
- Memória para guardar variáveis temporárias.
- Como uma *Pilha* de pratos, o último a ser colocado é o primeiro a ser retirado (*LIFO**)
- ***Expande***: Usa mais memória quando necessário.
- ***Contrai***: Liberta a memória quando o espaço deixa de ser necessário.



*LIFO - *Last-In First-Out* - O último a 'entrar' é o primeiro a 'sair'.

3 - Funções (11) - O Stack e o Stack Pointer ($sp=29)

- Expande para baixo (dos endereços maiores para os menores) e contrai para cima.
- *Stack Pointer*: **$sp** aponta para o topo do *stack*



Stack com uma só *word*: **0x12345678**

Stack com mais duas *words*: **0xAABBCCDD** e **0x11223344**

3 - Funções (12) - Salvaguarda de Registos (1)

- As funções invocadas não devem ter efeitos colaterais indesejados.
- Mas **diffofsums** altera o conteúdo de 3 registos: **$t0, $t1** e **$s0**

```
# $s0 = result
diffofsums:
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result =(f + g)-(h + i)
   add $v0, $s0, $0    # return value in $v0
   jr  $ra             # return to caller
```

3 - Funções (13) - Salvaguarda de Registos (2) - Soluções

Problema

- As funções invocadas não devem provocar efeitos colaterais.
- Mas diffofsums altera o conteúdo de 3 registos: $t0, $t1 e $s0

Soluções possíveis:

1. A *caller* guarda no *stack* todos os registos cujo conteúdo necessita de preservar antes de invocar a função.
ex: *main* guarda no *stack* $t0, $t1 e $s0

2. A *callee* guarda no *stack* todos os registos cujo conteúdo altera.
ex: diffofsums guarda no stack $t0, $t1 e $s0

3. A salvaguarda de registos no *stack* é repartida entre *caller* e *callee* (opção usada no MIPS).

3 - Funções (14) - Salvaguarda de Registos (3) - MIPS

A salvaguarda de registos no stack é repartida entre a *caller* e a *callee*. Como?

- *A caller* guarda no stack os registos **$tx**, cujo conteúdo necessita preservar, antes de invocar a *callee*, e restaura-os após o retorno da jal.
- A *callee* começa por guardar no stack os registos **$sx** cujo conteúdo altera e restaura-os antes de retornar.

3 - Funções (15) - Salvaguarda de Registos (4) - MIPS

- *A caller* guarda no stack o conteúdo dos registos **$tx** (i.e., só se voltar a precisar deles)
- A *callee* guarda no stack os registos **$sx** que vai usar.

Não-Preservados *Caller-Saved*	Preservados *Callee-Saved*
$t0-$t9	$s0-$s7
$a0-$a3	$ra
$v0-$v1	$sp

diffofsums - não é necessário guardar no stack **$t0** e **$t1**, porque essa tarefa é da responsabilidade da *caller*. Pouco claro no livro...

3 - Funções (16) - Salvag. Registos (5) - *main* (caller)

```
# O main precisa* salvaguardar os registo $t0 e $t1
main:                   # $s0 = y
  ...
  addiu $sp, $sp,-8     # make space on stack
  sw    $t0, 4($sp)     # save $t0
  sw    $t1, 0($sp)     # save $t1
  addi  $a0, $0, 2      # arg0 = 2
  addi  $a1, $0, 3      # arg1 = 3
  addi  $a2, $0, 4      # arg2 = 4
  addi  $a3, $0, 5      # arg3 = 5
  jal   diffofsums      # call Function
  lw    $t1, 0($sp)     # restore $t1
  lw    $t0, 4($sp)     # restore $t0
  addiu $sp, $sp,8      # deallocate stack space
  add   $s0, $v0,$0     # y = returned value
  ...
```

main decrementa o $sp de 2 words; Guarda $t1 em 0($sp) e $t0 em 4($sp).

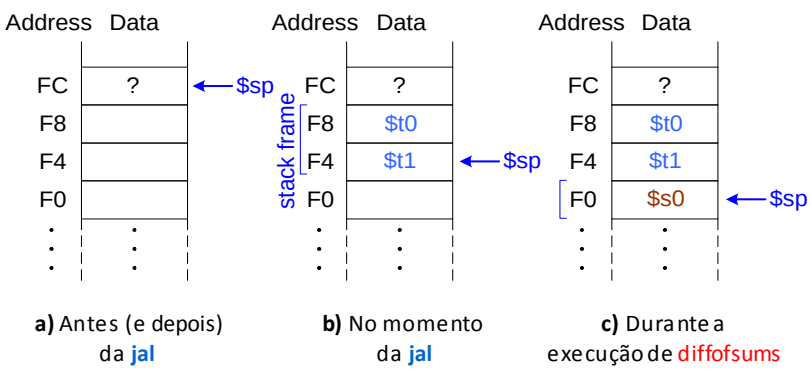
*Isto só é necessário caso o *main* não disponha de registos $sx disponíveis.

3 - Funções (17) - Salvag. Registos (6) - diffofsums (callee)

```
# $s0 = result
# A convenção MIPS obriga a callee a preservar $s0
diffofsums:
    addiu $sp, $sp, -4      # make space on stack
    sw    $s0, 0($sp)       # save $s0
                            # no need to save $t0 or $t1
    add   $t0, $a0, $a1     # $t0 = f + g
    add   $t1, $a2, $a3     # $t1 = h + i
    sub   $s0, $t0, $t1     # result =(f + g)-(h + i)
    add   $v0, $s0, $0      # put return value in $v0
    lw    $s0, 0($sp)       # restore $s0
    addiu $sp, $sp, 4       # deallocate stack space
    jr    $ra               # return to caller
```

diffofsums decrementa o $sp de 1 word; Guarda $s0 em 0($sp).

3 - Funções (18) - Stack durante a call a diffofsums



a) Antes (e depois) da **jal**

b) No momento da **jal**

c) Durante a execução de diffofsums

b) main decrementa o $sp de 8 bytes (2 words); Guarda $t1 em 0($sp) e $t0 em 4($sp).

c) diffofsums decrementa o $sp de 4 bytes (1 word); Guarda $s0 em 0($sp).

3 - Funções (19) - Salvag. Registos (7) - main + diffofsums

Embora diffofsums altere o valor dos registos $t0, $t1 e $s0, usando a convenção anterior, isso não interfere com o bom funcionamento da main. Porquê?

1. main: garante que $t0 e $t1 preservam o valor após diffofsums ter sido invocada, guardando $t0 e $t1 no stack antes e restaurando-os após a *call* (jal <>) .

2. diffofsums: garante que o valor de $s0 é preservado, guardando-o no stack à entrada e restaurando-o à saída.

3 - Funções (20) - Invocação em Cadeia: proc1->proc2

```
proc1:
   addiu $sp, $sp, -4 # make space on stack
   sw    $ra, 0($sp)  # save $ra
   jal   proc2        # recall:jal changes $ra!
   ...
   lw    $ra, 0($sp)  # restore $ra
   addiu $sp, $sp, 4  # deallocate stack space
   jr    $ra          # return to caller
```

Quando uma função proc1, invoca outra proc2, tem de guardar no stack o registo **$ra** para que proc1 possa regressar ao código que a invocou (visto que a instrução **jal** usa (implicita/) o registo **$ra**!).

*proc1 **não** é uma rotina-terminal porque invoca outra, o que altera o registo $ra (jal).

3 - Funções (21) - Recursivas* (1) - Factorial C

High-level code

```
int factorial(int n) {
  if (n <= 1) return 1;
//else
return (n * factorial(n-1));
}
```

Recursiva

As implementações recursivas são em geral mais compactas (elegantes), embora sejam frequentemente mais lentas do que as implementações iterativas! Vão ser abordadas por serem interessantes, mas sobretudo para ilustrar o funcionamento do stack.

```
int factorial( int n ){
  int i, fact = 1;
  for ( i = n ; i>1; i-- )
    fact = i*fact;
  return fact;
}
```

Iterativa (não-recursiva)

*Recursiva - a função chama-se a si própria.

3 - Funções (22) - Recursivas (2) - Terminal vs Não-Terminal

Função Terminal e Não-terminal (*leaf* e *nonleaf*)

Uma função que não invoca outras é designada por *terminal*, diffosums é um exemplo.
Uma função que invoca outras é designada por *não-terminal*, main é um exemplo.

Regras de Salvaguarda de Registos no Stack (de novo)

1. *caller* salvaguarda os registos, que não são preservados pela convenção ($t0-$t9 e $a0-$a3), caso sejam necessários após a *call*.

2. *callee* salvaguarda os registos, que são preservados pela convenção ($ra e $s0-$s7) caso modifique o respetivo valor.

A função recursiva é uma função *não-terminal* que se chama a si própria, e rege-se pelas mesmas regras.

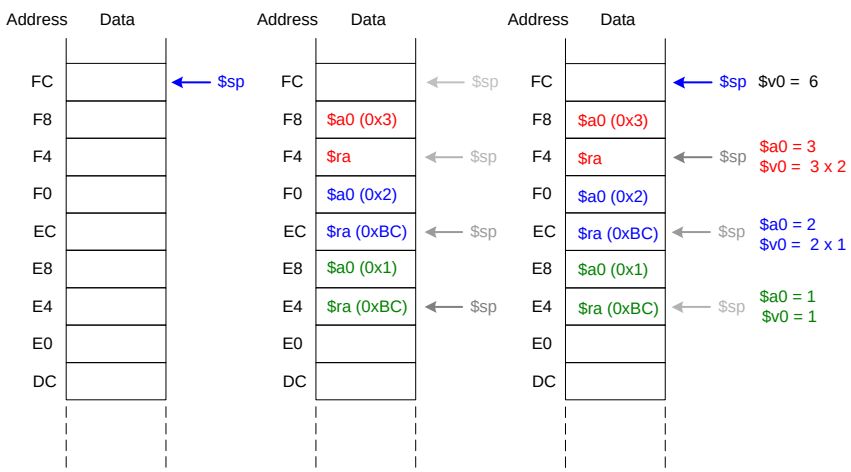
3 - Funções (23) - Recursivas (3) - Factorial ASM

```
int factorial(int n) {
  if (n <= 1) return 1;
//else
return (n * factorial(n-1));
}
```

A função factorial modifica os valores de $ra e de $a0, por isso salvaguarda o respetivo valor no *stack*.

```
0x90 factorial: addi $sp, $sp, -8    # make room
0x94            sw   $a0, 4($sp)     # store $a0
0x98            sw   $ra, 0($sp)     # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0   # n <= 1 ?
0xA4            beq  $t0, $0, else   # no: go to else
0xA8            addi $v0, $0, 1      # yes: return 1
0xAC            addi $sp, $sp, 8     # restore $sp
0xB0            jr   $ra             # return
#
0xB4 else:      addi $a0, $a0, -1    # n = n - 1
0xB8            jal  factorial       # recursive call
0xBC            lw   $ra, 0($sp)     # restore $ra
0xC0            lw   $a0, 4($sp)     # restore $a0
0xC4            addi $sp, $sp, 8     # restore $sp
0xC8            mul  $v0, $a0, $v0   # n * factorial(n-1)
0xCC            jr   $ra             # return
```


3 - Funções (24) - Recursivas (3) - Stack durante a call



Stack durante a execução da "jal factorial(3)":

A 1ª vez cria um *stack-frame* a vermelho, com $a0 (=3) e $ra (=main address?)
A 2ª vez cria um *stack-frame* a azul, com $a0 (=2) e $ra (0xBC)
A 3ª vez cria um *stack-frame* a verde, com $a0 (=1) e $ra (0xBC)

3 - Funções (25) - Resumo da Convenção de Uso de Regs.

- *Caller*
  - Guarda no stack os registos a preservar ($a0-$a3, $t0-$t9)
  - Coloca os argumentos em $a0-$a3
  - 'Salta' para o código da função *invocada*; jal <*callee*>
  - Utiliza o resultado devolvido em $v0
  - Restaura o conteúdo dos registos
- *Callee*
  - Guarda no stack os registos cujo conteúdo modifica ($ra, $s0-$s7)
  - Executa a função
  - Coloca o resultado em $v0
  - Restaura o conteúdo dos registos
  - Regressa ao código da *caller*; jr $ra

4 - Modos de Endereçamento* (1)

Onde estão os operandos da instrução?

- Todos operandos estão em registos
- Registos e uma constante ($Imediato_{16}$)
- Endereço-Base (registo) + $SignExt\text{-}Imediato_{16}$
- Relativo-ao-PC: ($PC + 4*SignExt\text{-}Imediato_{16}$)
- Pseudo-Direto: ($PC_{31..28}$ | $4*Imediato_{26}$)

*'Addressing Modes': Vão ser usados aquando da implementação do Datapath do CPU.

4 - Endereçamento (2) - Register Only & Immediate

Só Registos (Tipo-R)

- Todos os operandos contidos em registos
  - Exemplo: `add $s0, $t2, $t3`
  - Exemplo: `sub $t8, $s1, $0`

Valor Imediato (Tipo-I)

- Valor imediato de 16-bits usado como operando
  - Exemplo: `addi $s4, $t5, -73`
  - Exemplo: `ori $t3, $t7, 0xFF`

4 - Endereçamento (3) - Base Addressing*

Endereço-Base (Tipo-I)

- O Endereço-efetivo do operando é:

  Endereço-Base + *Offset* de 16-bits (sign-extended)

  - **Exemplo:** `lw $s4, 72($0)`

    Endereço = `$0 + 72`

  - **Exemplo:** `sw $t2, -25($t1)`

    Endereço = `$t1 - 25`

*Por vezes também designado por Endereçamento-Indexado

4 - Endereçamento (4) - PC-Relative (Branches)

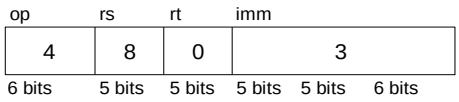
Relativo-ao-PC (Tipo-I)

```
0x10              beq   $t0, $0, else
0x14              addi  $v0, $0, 1
0x18              addi  $sp, $sp, i
0x1C              jr    $ra
0x20      else:   addi  $a0, $a0, -1
0x24              jal   factorial
```

Assembly Code

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

Field Values



BTA = (PC + 4) + imm<<2; ex: 0x14 + 3*4 = 0x20

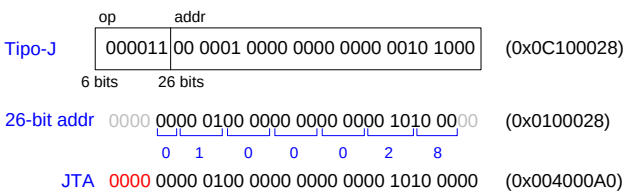
BTA - *Branch Target Address*; No exemplo, 3 representa o número de instruções.

4 - Endereçamento (5) - Pseudo-Direto (Jumps)

Pseudo-Direto (j e jal) - (Tipo-J)

```
0x0040005C          jal   sum
...
0x004000A0  sum:    add   $v0, $a0, $a1
```

Endereço (quase) completo codificado na instrução:

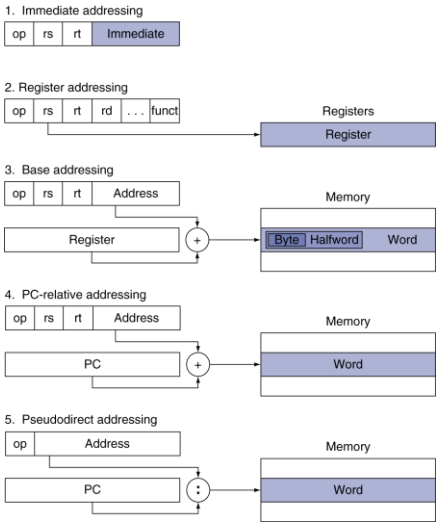


JTA: $(PC+4)_{31..28}$: (Imm26<<2) (':' -> concatenação)

ex: 0x0 : (0x010 0028)*4 = 0x0040 00A0

JTA - *Jump Target Address*. O JTA pode estar em qualquer posição do segmento *text*.

4 - Endereçamento (6) - Resumo



Exemplo

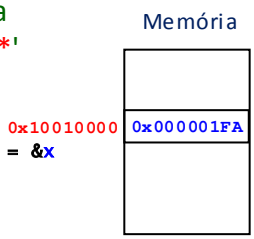
```
addi $s4, $t5, -73
add $s0, $t2, $t3
sw  $t2, -25($t1)
beq   $t0, $t1, label
jal   sum
```


XX - NEXT: Ponteiros

Uma variável x do tipo inteiro, tem o valor 0x01FA e está localizada no endereço de memória 0x10010000.

1. Declarar um ponteiro, em C?

```
// Declaração dum ponteiro, p_int, para uma
// variável do tipo int, através do operador '*'
int*  p_int;
```



2. Inicializar o ponteiro?

```
// Inicialização é feita através do operador '&'
int   x;
int*  p_int  = &x; // '&' atribui a p_int o endereço de x
```