

Introdução à Arquitetura de Computadores

Aula19

Assembly 4: Funções (cont.) e Addressing

Funções (continuação)

- Stack: Definição
Salvaguarda de Registos
- Função terminal e não-terminal
- Recursividade; Ex: Factorial

Addressing (Modos de Endereçamento)

- Tipo-R: Só-Registros (`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) - A Pilha (Stack)

Definição

- Zona de memória **reutilizável**, onde são guardadas variáveis **temporárias**.
- Analogia com uma **pilha** (de pratos); o último a ser colocado é o primeiro a ser retirado (**LIFO***)

3.2 Funções - Stack

Funcionamento

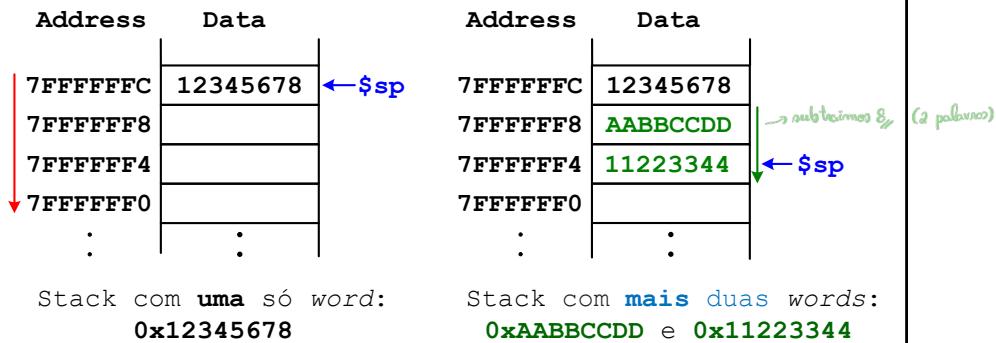
- **Expande:** Usa mais espaço de memória quando necessário.
- **Contrai:** Liberta o espaço de memória quando deixa de ser necessário.



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

3 - Funções (11) - 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 da stack.



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

- As funções que são **invocadas** não devem gerar efeitos colaterais adversos.
- 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
```

3.3 Funções - Salvaguarda de Registos

- Ora, **esta alteração** pode, potencial/, afectar a **main!**

main:

\$t0 = 3
\$t1 = 4
li \$a0, 42
li \$a1, 7
jr \$ra

sum: add \$v0, \$a0, \$a1
Empilha \$ra
Salva aux
Restaura \$ra
in \$ra

aux:

...
in \$ra

Terminar de ter cuidado com o \$ra

Resolver Problema!

II

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

Problema

- As funções invocadas não devem prejudicar o bom funcionamento da função *caller*.
- Mas *diffofsums* altera o conteúdo de 3 registos: $\$t0$, $\$t1$ e $\$s0$

Três soluções possíveis:

Problema:

Sempre que há um chomamento de uma função é sempre preciso utilizar a stack, mesmo quando não é necessário.

- A *caller (main)* guarda na *stack* todos os registos, cujo conteúdo necessita de preservar, antes de invocar a função, i.e., salvaguarda $\$t0$, $\$t1$ e $\$s0$;
- A *callee (diffofsums)* guarda na *stack* todos os registos, cujo conteúdo altera, i.e., salvaguarda $\$t0$, $\$t1$ e $\$s0$;
- A salvaguarda de registos na *stack* é repartida entre a *caller* e a *callee* (opção usada no MIPS).

Antes de chamar guarda na stack os valores dos registos importantes. Quando volta restaura...

© A. Nunes da Cruz

IAC - ASM4: Funções II e Addressing

*Registers: \$s's não salvaguardados
\$t's são temporários*

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

A salvaguarda de registos na stack é repartida entre a *caller* e a *callee*. Como?

- A *caller* guarda na stack os registos $\$tx$, cujo conteúdo necessita preservar, antes de invocar a *callee*, e restaura-os após o retorno da *jal*.

*Função chamada: não utiliza \$s's
Função chamada: não utiliza \$t's*

e...

- A *callee* guarda na stack os registos $\$sx$, cujo conteúdo altera, e restaura-os antes de retornar.

*↳ se for uma função intermédia temos um problema!
Não pode utilizar os \$s's nem os \$t's -> utiliza a stack*

© A. Nunes da Cruz

IAC - ASM4: Funções II e Addressing

5/29

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

- A *caller* guarda na stack o conteúdo dos registos **\$tx** (i.e., só se voltar a precisar deles)



Não-Preservados <i>Caller-Saved</i>	Preservados <i>Callee-Saved</i>
\$t0-\$t9	\$s0-\$s7
\$a0-\$a3	\$ra
\$v0-\$v1	\$sp



- A *callee* guarda na stack os registos **\$sx** que vai usar.

3 - Funções (16) - Salvag. Registros (5) - main (caller)

```
# A main precisa salvaguardar os registos $t0 e $t1,
main:                                # $s0 = y
...                                     ↗ 2 palavras
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
...

```

Só salvo a função isto porque ocupou todos os \$s's

Decrementa o **\$sp** de 2 words (8 bytes);
Guarda **\$t0** em 4(**\$sp**) e **\$t1** em 0(**\$sp**). Assume-se que a **main** vai necessitar de usar os registos **\$t0** e **\$t1**, após a **jal**!

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

```

# $s0 = result
# A convenção MIPS obriga a callee a preservar $s0
diffofsums:

Empilha $ra
addiu $sp, $sp, -4      # make space on stack
sw    $s0, 0($sp)        # save $s0

Restaura $ra
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $0
lw    $s0, 0($sp)
addiu $sp, $sp, 4
jr    $ra                 # restore $s0
                                # deallocate stack space
                                # return to caller

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

```

*Não é necessário guardar na stack \$t0 e \$t1 porque essa tarefa é da responsabilidade da caller!

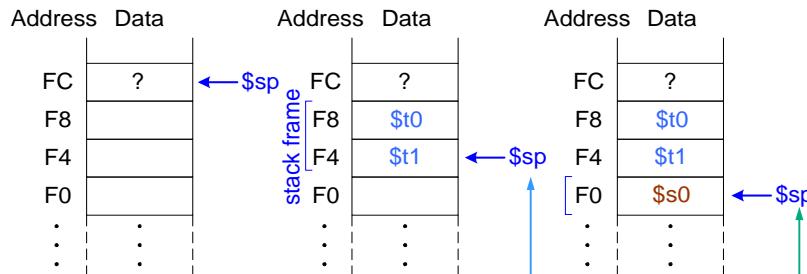
Registers \$t's são temporários por definição!

© A. Nunes da Cruz

IAC - ASM4: Funções II e Addressing

8/29

3 - Funções (18) - Stack: durante a 'jal diffofsums'



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

```

addiu $sp, $sp, -8
sw    $t0, 4($sp)
sw    $t1, 0($sp)

```

c) diffofsums: decrementa o \$sp de 4 bytes;
guarda \$s0 em 0(\$sp).

```

addiu $sp, $sp, -4
sw    $s0, 0($sp)

```

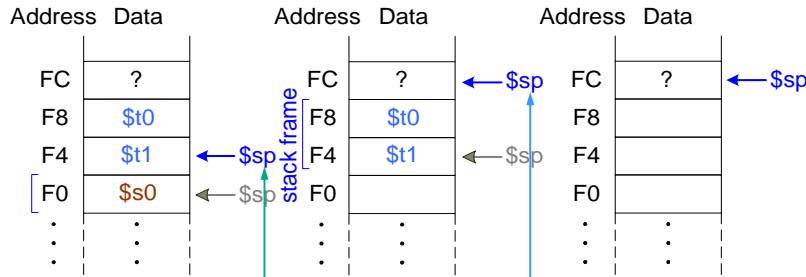
Corretamente não era necessária a pilha (\$t's eram passados para \$S's)
Isso é só um ex.

© A. Nunes da Cruz

IAC - ASM4: Funções II e Addressing

9/29

3 - Funções (19) - Stack: após o restauro dos registos



a) diffofsums:
antes de `jr $ra`

b) main: após a `jal` e o
restauro dos registos

c) main: regresso ao
estado anterior à `jal`

a) diffofsums: restaura `$s0` de `0($sp)` da stack;
incrementa o `$sp` de 4 bytes.

```
lw    $s0, 0($sp)
addiu $sp, $sp, 4
```

b) main: restaura `$t0` de `4($sp)` e `$t1` de `0($sp)`;
incrementa o `$sp` de 8 bytes (2 words).

```
lw    $t1, 0($sp)
lw    $t0, 4($sp)
addiu $sp, $sp, 8
```

3 - Funções (20) - Salvaguarda de Registos (7) - Solução

`diffofsums` altera o valor dos registos `$t0`, `$t1` e `$s0`, mas isso não interfere com o bom funcionamento da `main`, se usarmos a convenção anterior. Porquê?

1. `main`: garante que `$t0` e `$t1` preservam o valor após `diffofsums` ter sido invocada, guardando `$t0` e `$t1` na stack antes da `call` (`jal <>`) e restaurando-os após.

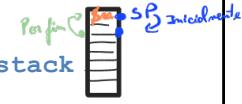
2. `diffofsums`: garante que o valor de `$s0` é preservado, guardando-o na stack à entrada e restaurando-o à saída.

3 - Funções (21) - Invocação em Cadeia: proc1->proc2

```

proc1:
Empilha $ra [addiu $sp, $sp, -4 # make space on stack
              sw    $ra, 0($sp)   # save $ra
              jal   proc2        # recall: jal changes $ra!
              ...
Restaura $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 na *stack* o registo **\$ra** para que **proc1** possa regressar ao código que a invocou (visto que a instrução **jal** usa **implicitamente** o registo **\$ra**).

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

3 - Funções (22) - Função Terminal vs Não-Terminal

Função terminal vs Função 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**, → Tenha cuidado!
main é um exemplo.

Regras de Salvaguarda de Registros na Stack (de novo)

1. **caller** salvaguarda os registos que não são preservados pela convenção (\$t0-\$t9, \$a0-\$a3 e \$v0-\$t1), caso sejam necessários após a *call*.
2. **callee** salvaguarda os registos que são preservados pela convenção (\$s0-\$s7, \$ra e \$sp), caso modifique o respetivo valor.

text

main: li \$50, 2 # uma_variavel = 2
li \$a0, 2
li \$a1, 3
li \$a2, 4
li \$a3, 5

\$ra [jal diff_of_sums

move \$t1, \$v0 # \$t1 = y

move \$a0, \$50

li \$v0, 1

syscall # print_int(uma_variavel)

move \$a0, \$t1

li \$v0, 1

syscall # print_int(y)

li \$v0, 10

syscall # exit()

sum: add \$v0, \$a0, \$a1 # \$v0 = \$a0 + \$a1
jr \$ra

Como é uma função intermédia

tem de guardar o \$ra que recebeu!

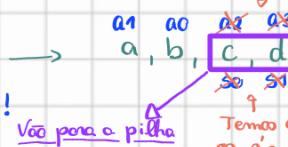
(para depois voltar)

\$sp →

↑ \$ra

→ valor de retorno

diff_of_sums: addi \$sp, \$sp, -4 - 12 - 16
sw \$ra, 0(\$sp) → stackpointer
sw \$a0, 4(\$sp)
sw \$a1, 8(\$sp) → cada vez que mudo a
pilha temos de mudar aqui!

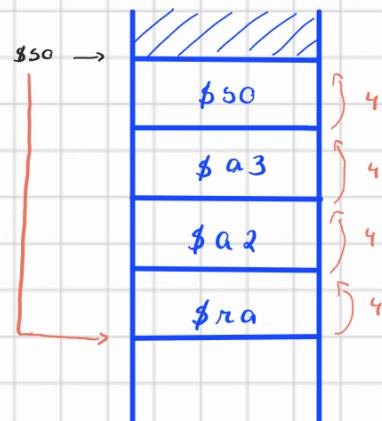


sw \$a2, 4(\$sp)

sw \$a3, 8(\$sp)

sw \$50, 12(\$sp)

→ metemos o \$50 na pilha para podermos utilizar \$50 = aux1



\$ra [jal sum # sum(a,b)

move \$50, \$v0 # \$50 = aux1 = sum(a,b)

lw \$a0, 4(\$sp) # a0 = c

lw \$a1, 8(\$sp) # a1 = d

\$ra [jal sum # \$50 = aux2 = sum(a,b)

sub \$v0, \$50, \$v0 # v0 = aux1 - aux2

valor de retorno

lw \$s0, 12(\$sp)
lw \$ra, 0(\$sp)
add \$sp, \$sp, 16

]

Retornar os valores que temos de (\$s0 e \$ra)
mantendo iguais fora da função
→ Retorna o stack pointer

jr \$ra

3 - Funções (23) - Recursivas (1) - Factorial C

Código C

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

Recursiva

3.4 Funções Recursivas

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

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

Iterativa
(não-recursiva)

Recursiva - função que se chama a si própria, sendo implícita/ uma função não-terminal!

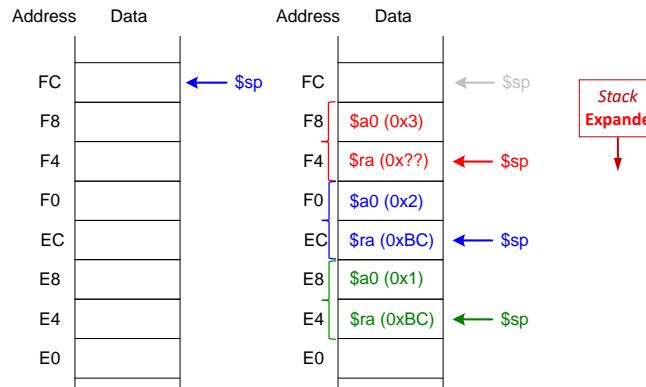
3 - Funções (24) - Recursivas (2) - Factorial ASM

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

A função factorial modifica os valores de **\$ra** e de **\$a0**, por isso salvaguarda os respetivos valores na *stack*.

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

3 - Funções (25) - Recursivas (3): Stack 'jal factorial' - Inv.



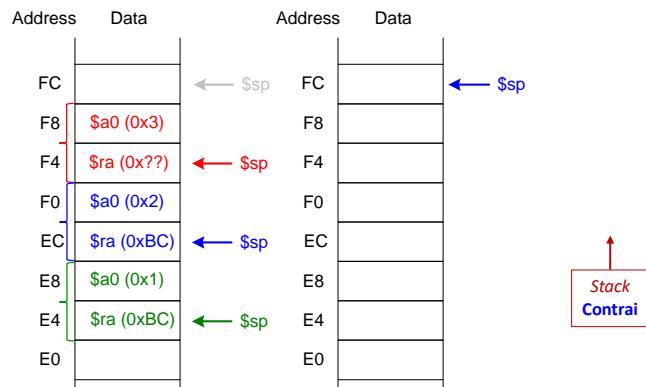
Stack durante a **Invocação** da 'jal factorial' (c/ \$a0=3):

A **1^a** vez cria uma **stack-frame** a **vermelho**, \$a0 = 3 e \$ra = 0x?? (**main**)

A **2^a** vez cria uma **stack-frame** a **azul**, \$a0 = 2 e \$ra = 0xBC

A **3^a** vez cria uma **stack-frame** a **verde**, \$a0 = 1 e \$ra = 0xBC

3 - Funções (26) - Recursivas (4): Stack 'jal factorial' - Ret.



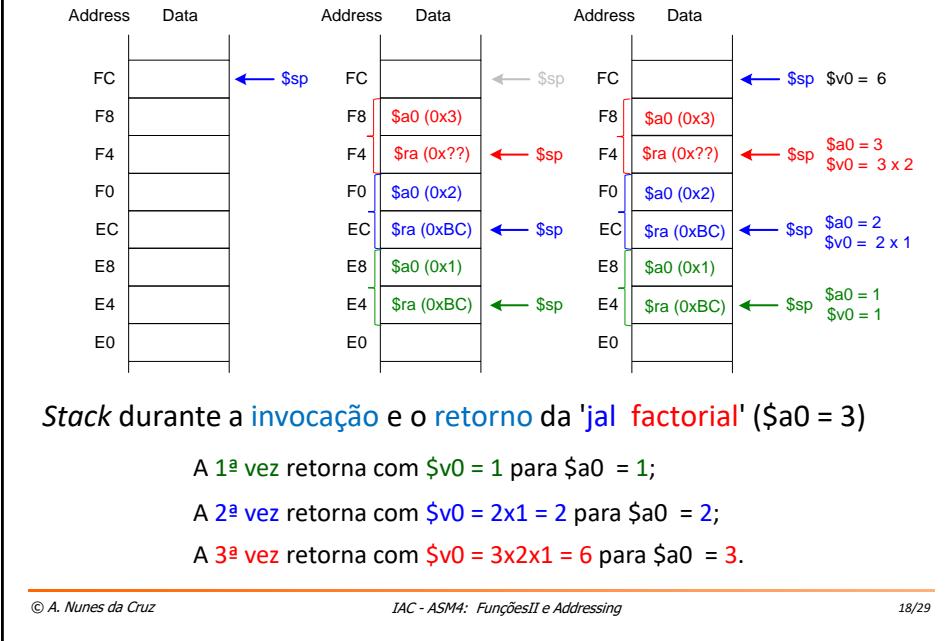
Stack antes do **retorno** através de 'jr \$ra':

A **1^a** vez liberta a **stack-frame** a **verde**, e retorna com \$ra = 0xBC

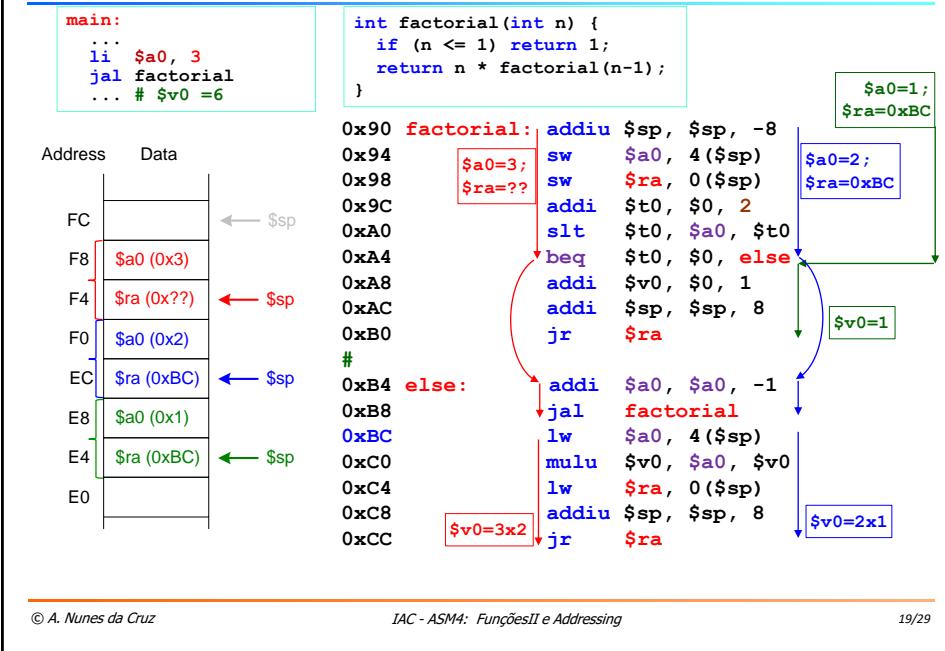
A **2^a** vez liberta a **stack-frame** a **azul**, e retorna com \$ra = 0xBC

A **3^a** vez liberta a **stack-frame** a **vermelho**, e retorna com \$ra = 0x??

3 - Funções (27) - Recursivas (5): Stack 'jal factorial'- I e R



3 - Funções (28) - Recursivas (6): Stack 'jal factorial' - I e R



3 - Funções (29) - Recursivas (7) - factorial optimizada - I

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

A função **fact** modifica os valores de **\$ra** e de **\$a0**, por isso salvaguarda os respetivos valores na *stack*.

```
0x90 fact: addiu $sp, $sp, -8    # make room
0x94     sw   $a0, 4($sp)      # store $a0
0x98     sw   $ra, 0($sp)      # store $ra
0x9C     addi $v0, $0, 1       # f = 1
0xA0     addi $t0, $0, 2       #
0xA4     slt  $t0, $a0,$t0      #
0xA8     bne  $t0, $0, fex      # if(n < 2) return 1
#
0xAC     addi $a0, $a0, -1      # n = n - 1
0xB0     jal   fact            # fact(n-1)
0xB4     lw    $a0, 4($sp)      # restore $a0
0xB8     mulu $v0, $a0, $v0      # n * factorial(n-1)
0xBC     lw    $ra, 0($sp)      # restore $ra
0xC0 fex: addiu $sp, $sp, 8      # restore $sp
0xC4     jr   $ra              # return
```

Alternativa1: Mudando o **beq** em **bne**, obtemos uma redução de 16 para 14 instruções!

3 - Funções (30) - Recursivas (8) - factorial optimizada - II

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

A função **fact** modifica os valores de **\$ra** e de **\$a0**, por isso salvaguarda os respetivos valores na *stack*.

```
0x90 fact: addi $v0, $0, 1      # f = 1
0x94     slti $at, $a0,2        #
0x98     bne  $at, $0,fex       # if(n < 2) return 1
#
0x9C     addiu $sp, $sp, -8      # make room
0xA0     sw   $ra, 0($sp)      # save $ra
0xA4     sw   $a0, 4($sp)      # save $a0
0xA8     addi $a0, $a0,-1       # n = n - 1
0xAC     jal   fact            # fact(n-1)
0xB0     lw    $a0, 4($sp)      # restore $a0
0xB4     mulu $v0, $a0, $v0      # n* fact(n-1)
0xB8     lw    $ra, 0($sp)      # restore $ra
0xBC     addiu $sp, $sp, 8       # restore $sp
0xC0 fex: jr   $ra              # return
```

Alternativa2: 1. A utilização de **slti** em vez de **slt** poupa mais uma instrução;
2. Não há necessidade de criar uma *stack frame* para $n \leq 1$.

3 - Funções (31) - Convenção de Uso de Registos (Resumo)

- **Caller**

- Guarda na *stack*, decrementando o **\$sp**, os registos a preservar (**\$t0-\$t9**, **\$a0-\$a3** e **\$v0-\$v1**)
- 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** e o valor do **\$sp**

- **Callee**

- Guarda na *stack* os registos cujo conteúdo modifica (**\$ra**, **\$s0-\$s7**)
- Usa os argumentos em **\$a0-\$a3**, executa a função e coloca o resultado em **\$v0**
- Restaura o conteúdo dos **registos** e o valor do **\$sp**
- Regressa ao código da *caller* (**jr \$ra**)

Nota: Existem 3 "tipos" para a instrução do Tipo I
 addi RD, RS, RT
 lw RT, imm(RS) → RT = Mem[RS + imm]
 beq RS, RT, imm16 → BTA = (PC + 4) + 4 * imm

4 - Modos de Endereçamento* (1)

Onde estão os operandos da instrução?

4. Modos de Endereçamento

- | | |
|----------------------------|--|
| Tipo R
Tipo I
Tipo J | <ul style="list-style-type: none"> • Todos em Registos • Registros e Imediato₁₆ (constante) • Endereço-Base (Registro) e Imediato₁₆ • Relativo-ao-PC: (PC4 + 4*Imediato₁₆) • Pseudo-Direto: (PC4_{31..28} : 4*Imediato₂₆) |
|----------------------------|--|

⚠ Não precisamos saber os mesmos, mas sim saber usar

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

4 - Endereçamento (2) - Register Only & Immediate

1. Só Registros (tipo-R)

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

2. Imediato (tipo-I)

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

A extensão dos 16-bits para 32-bits, é *sign-extended* para `addi`
mas *zero-extended* para `ori`!

4 - Endereçamento (3) - Base Addressing

3. Endereço-Base (tipo-I)

- O Endereço-efetivo do operando é dado por:

Endereço-Base + *Imediato16* (*sign-extended*)

- `lw $s4, 72($0)`
Endereço-efetivo = $(\$0) + 72$
- `sw $t2, -25($t1)`
Endereço-efetivo = $(\$t1) - 25$

4 - Endereçamento (4) - PC-Relativo (Branches) - BTA

4. Relativo-ao-PC (tipo-I)

Program Counter
temos sempre o endereço
da prox. instrução!

endereços →

<code>0x10</code> <code>0x14 (PC+4)</code> <code>0x18</code> <code>0x1C</code> 0x20 else: <code>0x24</code>	<code>beq \$t0, \$0, else</code> <code>addi \$v0, \$0, 1</code> <code>addi \$sp, \$sp, i</code> <code>jr \$ra</code> <code>addi \$a0, \$a0, -1</code> <code>jal factorial</code>
---	---

Assembly Code

Field Values

op	rs	rt	imm
4	8	0	3
6 bits	5 bits	5 bits	16 bits

Quantos períodos em salto para lá chegar

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

BTA - Branch Target Address; O valor **imm (3)** representa o número de instruções.

© A. Nunes da Cruz IAC - ASM4: Funções II e Addressing 26/29

4 - Endereçamento (5) - PC-Relativo (Branches) - Imm=-1

4. Relativo-ao-PC (tipo-I) - Caso particular

0x10 stuck: `beq $0, $0, stuck`
0x14 (PC+4) `addi $v0, $0, 1` # never reached!

Assembly Code

Field Values

op	rs	rt	imm
4	0	0	-1
6 bits	5 bits	5 bits	16 bits

1 posição para trás

BTA = (PC + 4) + imm<<2; ex: $0x14 - 1*4 = 0x10$

BTA - Branch Target Address; O valor **imm (-1)** representa o número de instruções.

© A. Nunes da Cruz IAC - ASM4: Funções II e Addressing 27/29

4 - Endereçamento (6) - Pseudo-Direto (Jumps)

5. Pseudo-Direto (je e jal) - (tipo-J)

0x0040005C

jal

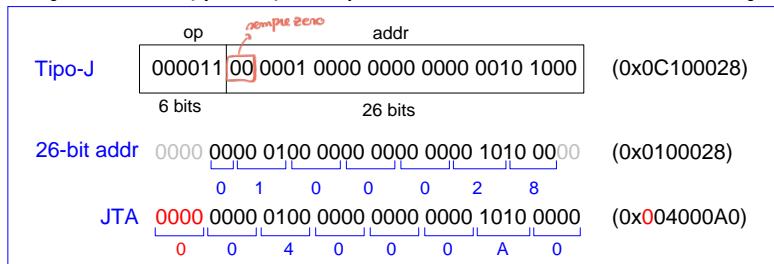
sum

...

0x004000A0 sum: add \$v0, \$a0, \$a1

endereço de 32 bits tem
de ser colocado em 26 bits

Endereço de **sum** (quase) completo, está codificado na instrução:



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

$$\text{Ex: } 0x0 : (0x010\ 0028)*4 = 0x0040\ 00A0$$

JTA - Jump Target Address. O endereço codificado na instrução está dividido por 4!

© A. Nunes da Cruz

IAC - ASM4: FunçõesII e Addressing

28/29

$\text{JTA} = (PC+4)_{31..28} : (Imm26 \ll 2)$

\downarrow concatenação

Faltam 4 bits,
estão aqui!

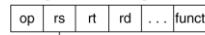
4 - Endereçamento (6) - Resumo

1. Immediate addressing



addi \$s4, \$t5, -73

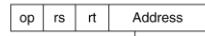
2. Register addressing



Registers

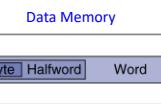
add \$s0, \$t2, \$t3

3. Base addressing



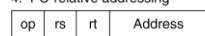
Data Memory

Register



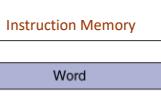
sw \$t2, -25(\$t1)

4. PC-relative addressing



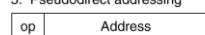
Instruction Memory

PC



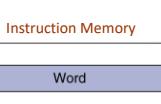
beq \$t0, \$t1, label

5. Pseudodirect addressing



Instruction Memory

PC



jal sum

© A. Nunes da Cruz

IAC - ASM4: FunçõesII e Addressing

29/29