Introdução Curta ao MIPS

Simão Melo de Sousa

RELEASE - RELiablE And SEcure Computation Group
Computer Science Department
University of Beira Interior, Portugal
desousa@di.ubi.pt
http://www.di.ubi.pt/~desousa/

Este documento é uma tradução adaptada do capítulo "Introduction à l'assembleur MIPS" da sebenta "Cours de Compilation" de Jean-Christophe Filliatre (http://www.lri.fr/~filliatr).

Plano

Arquitectura e assembly MIPS

- Máquina alvo para estas aulas.
- MIPS na wikipedia
- Um bom livro de Arquitectura de computador que introduz a arquitectura MIPs:

  Computer Organization & Design: The Hardware/Software Interface, Second Edition. By John Hennessy and David Patterson. Published by Morgan Kaufmann, 1997. ISBN 1-55860-428-6.
- uns acetatos completos e pedagógicos sobre o MIPS: (link1)

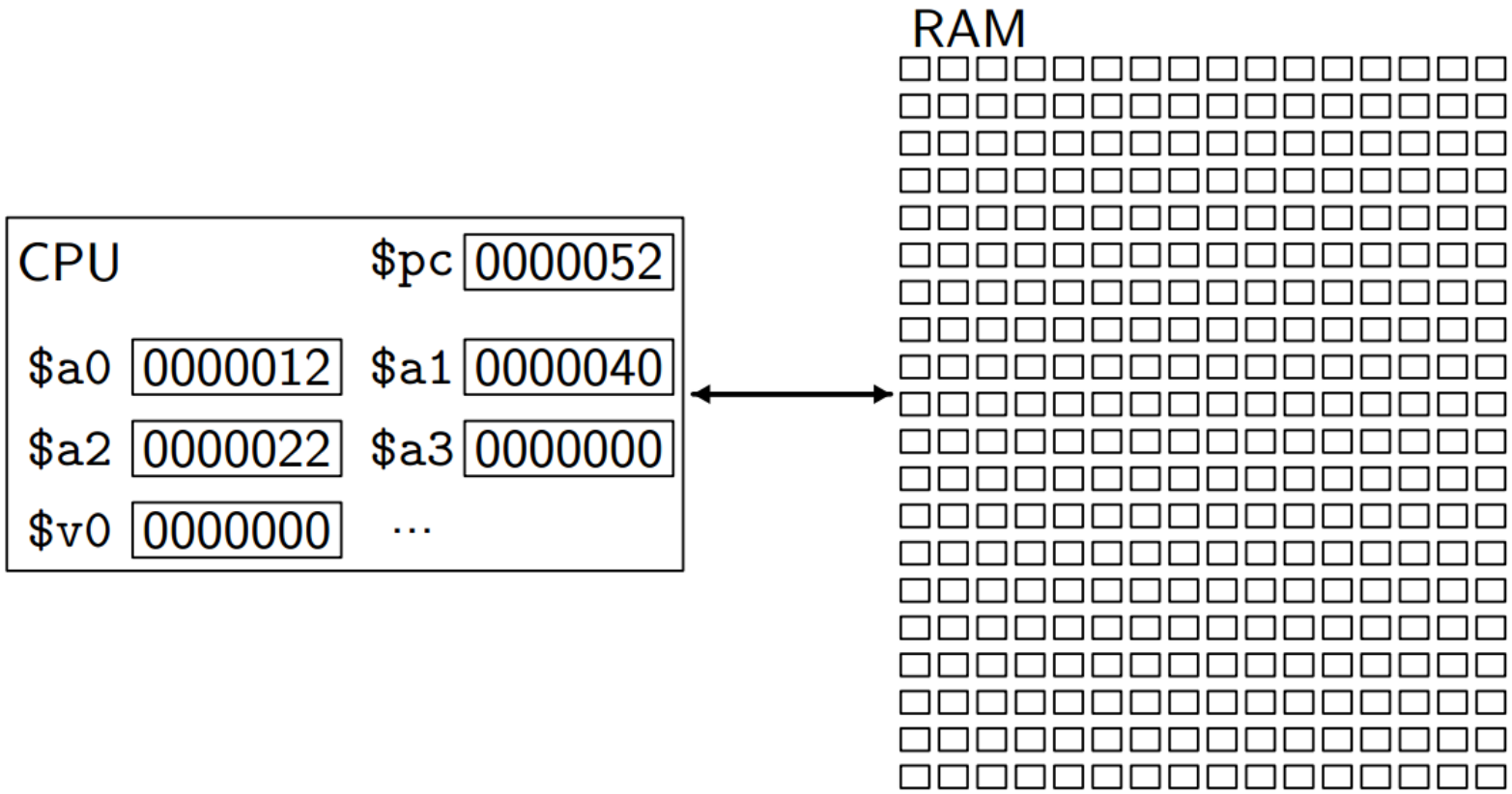
  (link2)
- SPIM: Simulador MIPS
  - (link1)
  - (link2)

Um pouco de Arquitectura de Computadores

De forma muitio resumida, um computador é composto de:

- Uma unidade de cálculo (CPU), contendo
  - un pequeno numero de registos inteiros ou flutuantes
  - mecanismos de cálculo
- de uma memória (RAM)
  - composta de um grande número de bytes (8 bits)
    por exemplo, 1Gb = 2^{30} bytes = 2^{33} bits, ou seja, $2^{2^{33}}$ configurações possíveis da memória
  - contém dados e programas (instruções)

Um pouco de Arquitectura de Computadores



O acesso à memória custa carro (tendo uma arquitectura capaz de realizar um bilião de instrução por segundo, a luz só consegue percorrer 30 centímetros entre cada instrução)

Um pouco de Arquitectura de Computadores

A realidade é um pouco mais complexa...

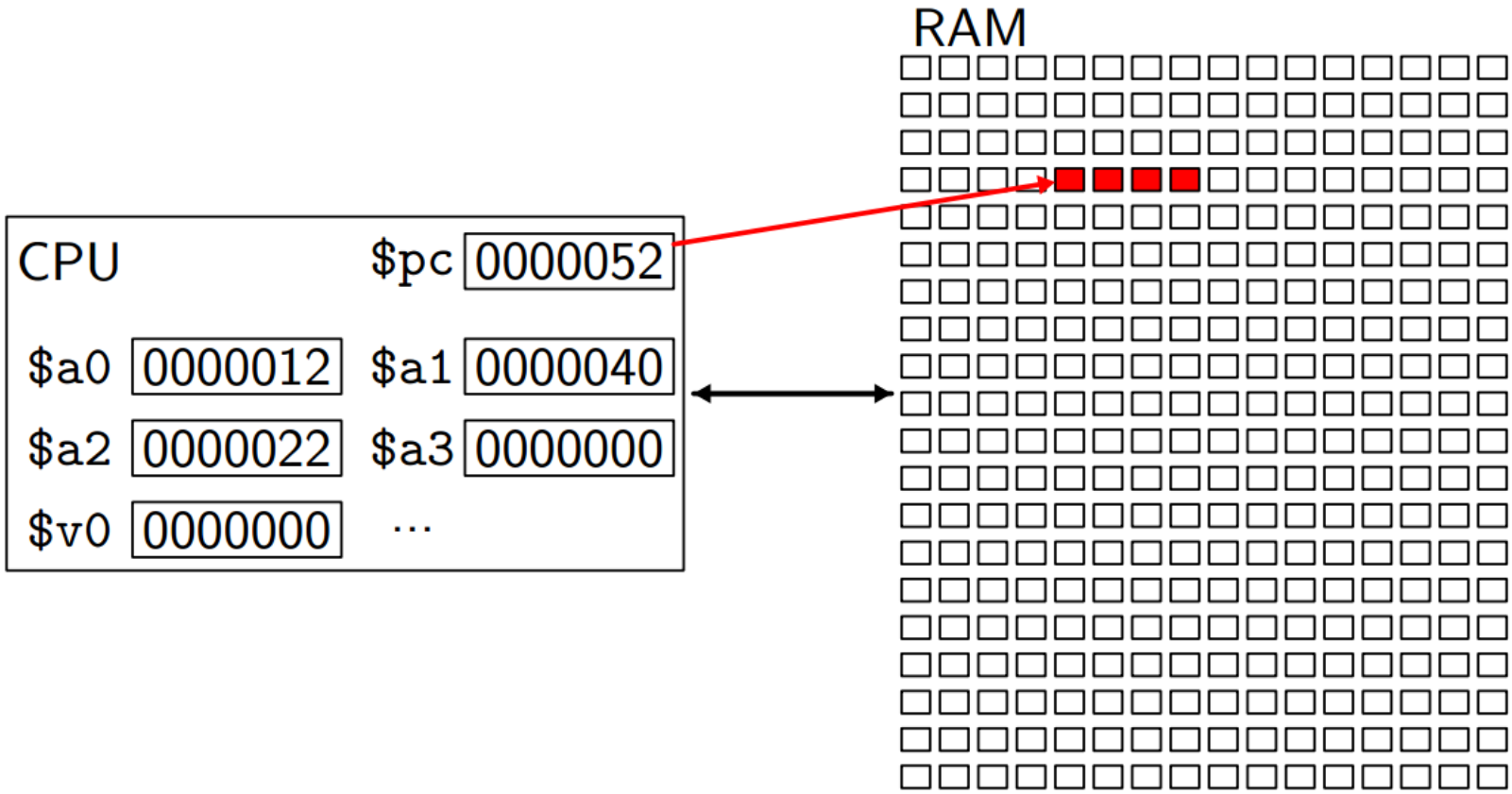
- presença de vários (co)processadores, alguns deles dedicados a operações sobre flutuantes
- uma ou várias caches (políticas LRU, random etc..)
- virtualização da memória (MMU)
- etc...

Princípio de execução

Esquematicamente, a execução decorre da seguinte forma:

- um registo ($pc) contém o endereço da instrução por executar
- lê-se os 4 (ou 8) bytes endereçados por $pc (fase designada de **fetch**)
- interpretar estes bits como sendo instruções (fase designada de **decode**)
- executar a dita instrução (fase designada de **execute**)
- modifica-se o registo $pc por forma a que este endereça a próxima instrução (a instrução seguinte, em geral, caso a instrução actual não obriga a um salto)

Princípio de execução



instrução:	000000	00001	00010	0000000000001010
descodificação:	add	$a1	$a2	10

i.e. juntar 10 ao registo $a1 e arquivar o resultado no registo $a2

Princípio de execução

Aqui também, a realidade é bem mais complexa do que este último, muito simples, exemplo.

- Pipelines: várias instruções são executadas em paralelo (por exemplo, numa sequência de 3 instruções, enquanto que a primeira é executada, a segunda é já descodificada e a terceira é carregada, isso tudo num só ciclo CPU - i.e. em simultâneo)
- Branch prediction. Para optimizar o pipelining, existe a possibilidade de adivinhar (com base em determinadas heurísticas) o resultado de instruções de salto (que necessitam fazer um *flush* do pipeline caso haja salto)

Arquitectura MIPS

- 32 registos, $r_0 \cdots r_{31}$
  - r_0 contém sempre 0
  - Estes registos podem ser referenciados por outros nomes (correspondentes a uma convenção de nomes admitida nos simuladores que vamos usar):
    `zero, at, v0-v1, a0-a3, t0-t9, s0-s7, k0-k1, gp, sp, fp, ra`
  - três tipos de instruções:
    - ★ instruções de transferências, entre registos e/ou memória
    - ★ instruções de cálculo
    - ★ instruções de salto

SPIM

Na prática , utilizaremos um simulador MIPS designado por **SPIM**
Em linha de comando: `spim [-file] file.s`
Em modo gráfico: `xspim -file file.s`
(execução passo a passo, visualização dos registos, da memória, etc.)

Instruções MIPS

- Carregamento de uma constante (16 bits com signo) num registo

```
li  $a0, 42              # a0 <- 42
lui $a0, 42              # a0 <- 42 * 2^16
```

- Cópia de um registo para outro

```
move $a0, $a1            # cópia de *a1* para *a0*
```

Instruções MIPS - aritmética

- Soma de dois registos

```
add $a0, $a1, $a2       # a0 <- a1 + a2
add $a2, $a2, $t5       # a2 <- a2 + t5
```

  idem para `sub`, `mul`, `div`

- soma do valor de um registo com uma constante

```
addi $a0, $a1, 42       # a0 <- a1 + 42
```

  (aqui não há o equivalente com `subi`, `muli`, `divi`)

- negação

```
neg $a0, $a1            # a0 <- -a1
```

- valor absoluto

```
abs $a0, $a1            # a0 <- |a1|
```

Instruções MIPS - bit-wise

- Negação lógica (not $100111_2 = 011000_2$)

```
not $a0, $a1              # a0 <- not(a1)
```

- Conjunção lógica (and $(100111_2, 101001_2) = 100001_2$)

```
and  $a0, $a1, $a2        # a0 <- and(a1,a2)
andi $a0, $a1, 0x3f       # a0 <- and(a1,0...0111111)
```

- Disjunção lógica (or $(100111_2, 101001_2) = 101111_2$)

```
and  $a0, $a1, $a2        # a0 <- or(a1,a2)
andi $a0, $a1, 42         # a0 <- or(a1,0...0101010)
```

Instruções MIPS - shift

- left shift (inserção de zeros)

```
sll  $a0, $a1, 2        # a0 <- a1 * 4
sllv $a1, $a2, $a3      # a1 <- a2 * 2^a3
```

- right shift aritmético (cópia do bit do sinal)

```
sra  $a0, $a1, 2        # a0 <- a1 / 4
```

- right shift lógico (inserção de zeros)

```
srl  $a0, $a1, 2
```

- Rotação

```
rol  $a0, $a1, 2
ror  $a0, $a1, 2
```

Instruções MIPS - transferências/leituras

- ler uma palavra (32 bits) em memória

```
lw  $a0, 42($a1)        # a0 <- mem[a1 + 42]
```

  O endereço é dado por um registo e um offset dado no formato 16 bits com sinal

- variantes para ler 8 ou 16 com sinal ou não (`lb`, `lh`, `lbu`, `lhu`).

Instruções MIPS - transferências/escritas

- escrever uma palavra (32 bits) em memória

```
sw   $a0, 42($a1)        # mem[a1 + 42] <- a0
                         #(cuidado com a ordem
                         # dos parâmetros)
```

  O endereço é dado por um registo e um offset dado no formato 16 bits com sinal

- variantes para ler 8 ou 16 (`sb, sh`).

Instruções MIPS - saltos

- salto condicional

```
beq  $a0, $a1, label  # se a0 = a1 salto para label
                      # senão... nada
```

- variantes: `bne`, `blt`, `ble`, `bgt`, `bge` (assim como comparações sem sinal).
- variantes: (comparações com zero) `beqz`, `bnez`, `bltz`, `blez`, `bgtz`, `bgez`.

Instruções MIPS - saltos

salto incondicional

- para um endereço

```
j label
```

- para um endereço, com arquivo do endereço da instrução seguinte no registo `$ra`.

```
jal label                # jump and link
```

- para um endereço arquivado num registo.

```
jr $a0
```

- para um endereço arquivado num registo, com arquivo do endereço da instrução seguinte num registo.

```
jalr $a0, $a1     # salto para $a0, e arquivo da próxima
                  # instrução  em $a1
```

Instruções MIPS - chamadas ao sistema

É possível invocar alguns serviços disponibilizados pela arquitectura de suporte (num simulador, são serviços do sistema operativo subjacente). A instrução MIPS para esta chamada é **syscall**.
O código da instrução por chamar deve estar no registo **$v0**, os argumentos em **$a0 - $a3**. O resultado será colocado no registo **$v0**.
Por exemplo, para usufruir dos serviços da função `print_int` basta:

```
li  $v0, 1          #código de print_int
li  $a0,42          #o valor (do universo) por mostrar
syscall
```

Assembly MIPS

Como já sabem..... não é costume programar em linguagem máquina, mas sim com recurso ao assembly.
O assembly fornece certas facilidades:

- *labels* simbólicos
- alocações de dados globais
- pseudo-instruções

Assembly MIPS

A directiva

```
.text
```

indica que instruções seguem. A directiva

```
.data
```

indica que dados seguem
O código ficará arquivado a partir do endereço `0x400000` e os dados ficarão arquivados a partir do endereço `0x10000000`.
Uma etiqueta simbólica (label) é introduzida por

```
label:
```

e o endereço que ela representa pode ser carregado num registo

```
la  $a0, label
```

Hello world!

SPIM invoca o programa endereçado por `main` e entrega-lhe o endereço onde *recomeçar* no fim da sua execução no registo **$ra**

```
        .text
main:   li      $v0, 4      # código de print_int
        la      $a0, hw     # endereço da string
        syscall             # chamada ao sistema
        jr      $ra         # fim do programa
        .data
hw:     .asciiz "hello world\n"
```

(`.asciiz` é uma facilidade para `.byte 104, 101, ... 0`)

Compilação e MIPS

O desafio da compilação para o assembly (MIPS, em particular) é conseguir traduzir um linguagem de alto nível para este conjunto de instruções. Em particular, é preciso:

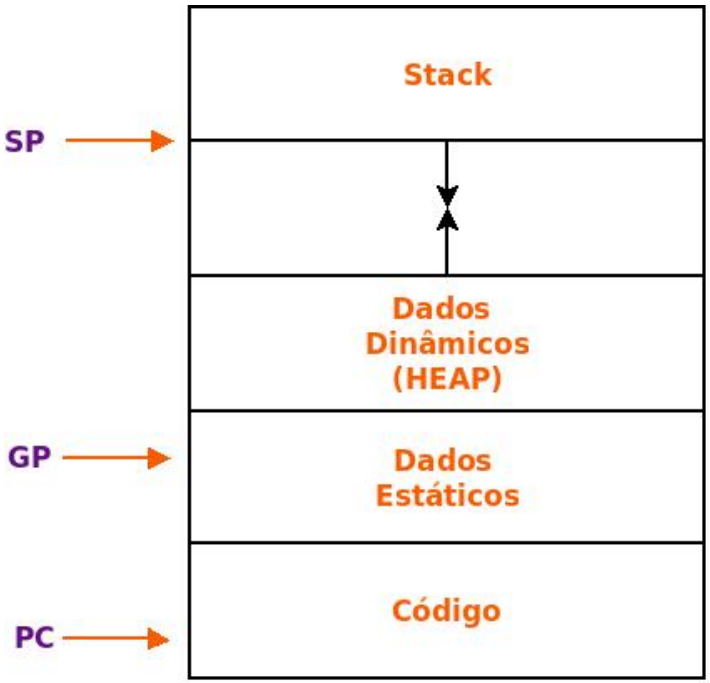
- traduzir as estruturas de controlo (testes, ciclos, excepções, etc.)
- traduzir as chamadas às funções
- traduzir as estruturas de dados complexas (vectores, registos, objectos, fechos, etc.)
- alocar memória de forma dinâmica

Chamadas a funções

Facto: as chamadas a funções podem estar arbitrariamente aninhadas.
$\Rightarrow$ Os registos podem não chegar para arquivar todos os parâmetros, variáveis locais e outros dados locais a cada chamada
$\Rightarrow$ É necessário alocar memória para tal

As funções procedem na base de uma política *last-in first-out*, i.e. uma **pilha**.

A pilha



- A **pilha** é posicionada na parte superior, e cresce no sentido dos endereços decrescentes. O registo **$sp** aponta para o topo da pilha.
- Os dados dinâmicos (que sobrevivem às chamadas a funções) são alocadas na **heap** (eventualmente por um GC), na parte inferior da zona de dados, logo a seguir aos dados estáticos.
- Assim temos casa arrumada...

Chamadas a funções

Quando uma função f (quem invoca, ou **caller**) pretende invocar uma função g (que é chamado ou **callee**), f executa:

```
jal g
```

quando a função invocada termina, esta devolve o controlo para o caller, com:

```
jr $ra
```

Mas então:

- se g ela própria invoca uma função, o registo $ra será actualizado (e perderá o valor de que chamou g)
- da mesma forma, qualquer registo utilizado por g ficará inutilizável por f posteriormente.

Existe várias formas de resolver esta problemática, mas de forma geral costuma-se respeitar uma **convenção para chamadas**

Convenção para chamadas

Uso dos registos:

- **$at, $k0, $k1** estão reservados para o Sistema Operativos
- **$a0, ... , $a3** usados para passar os 4 primeiros argumentos (ou outros são passados via pilha)
- **$v0, ... , $v1** usados para passar devolver o resultado de uma chamada
- **$t0, ... , $t9** são registos **caller-saved**, i.e. o caller deve salvaguarda-los, se necessário. São usados tipicamente para dados que não necessitam sobreviver às chamadas
- **$s0, ... , $s7** são registos **callee-saved**, i.e. o callee deve salvaguarda-los, se necessário. São usados tipicamente para dados de duração longa, que necessitam assim de sobreviver às chamadas
- **$sp, $fp, $ra, $gp** são, respectivamente, o apontador para o topo da pilha, o apontador para a **frame**, o *return address* e o registo que aponta para o meio da zona de dados estáticos (10008000_{16})

Chamada, em 4 etapas

Há 4 fases numa chamada a uma função:

1. Para o caller, mesmo antes da chamada
2. Para o callee, logo a seguir à chamada
3. Para o callee, mesmo antes do fim da chamada
4. Para o caller, logo a seguir ao fim da chamada

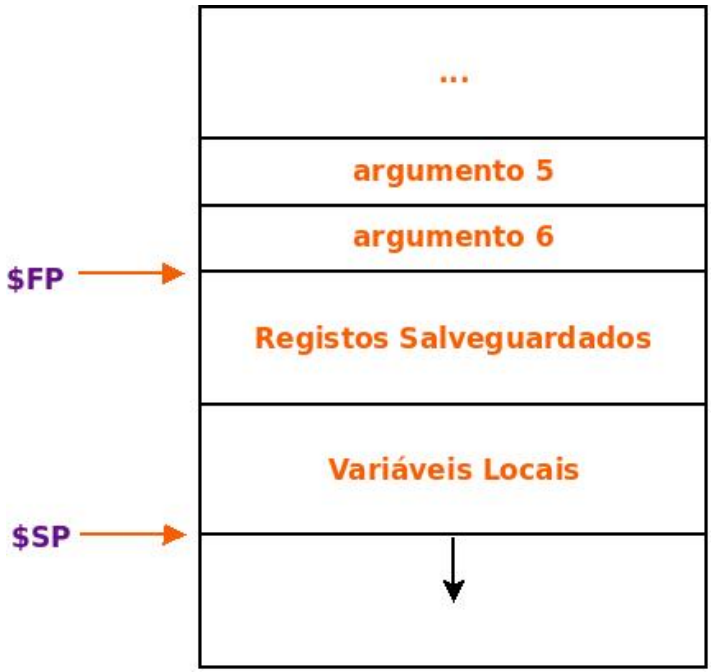
Estas organizam-se na base de um segmente de memória localizado no topo da pilha, designada de **tabela de activação** ou (em inglês) de **stack frame**, estando este situado entre $fp e $sp.

caller, mesmo antes da chamada

1. Passa os argumentos em $a0 - $a3, e os restantes na pilha (se houver mais do que 4 argumentos)
2. salvaguarda os registos $t0-$t9 que entende utilizar após a chamada (na sua própria tabela de activação)
3. executa:

```
jal callee
```

callee, no início da chamada



1. Alocar a sua tabela de activação, por exemplo:

```
addi $sp, $sp, -28
```

2. Salvaguardar $fp e, a seguir, posicioná-lo, por exemplo

```
sw   $fp, 24($sp)
addi $fp, $sp, 24
```

3. salvaguardar $0 - $s7 e $ra caso seja necessário

$fp permite alcançar facilmente os argumentos e as variáveis locais, com base num offset fixo, qualquer que seja o estado da pilha.

o callee, no fim da chamada

1. coloca o resultado em $v0 (e em $v1 se for necessário)
2. restabelece os registos salvaguardados
3. *pop* da sua tabela de activação, por exemplo

```
addi $sp, $sp, 28
```

4. executa

```
jr $ra
```

caller, logo a seguir ao fim da chamada

1. *pop* dos eventuais argumentos 5, 6, etc.... (os que não couberam nos reguistos $a0 - $a3)
2. restabelece os registos *caller-saved*.

Plano

MIPS em resumo

```
li   $r0, C                 $r0 <- C
lui  $r0, C                 $r0 <- 2^16*C
move $r0, $r1               $r0 <- $r1

add  $r0, $r1, $r           $r0 <- $r1 + $r2
addi $r0, $r1, C            $r0 <- $r1 + C
sub  $r0, $r1, $r2          $r0 <- $r1 - $r2
div  $r0, $r1, $r2          $r0 <- $r1 / $r2
div  $r1, $r2               $lo <- $r1 / $r2, $hi <- $r1 mod $r2
mul  $r0, $r1, $r2          $r0 <- $r1 * $r2 (sem overflow)
neg  $r0, $r1               $r0 <- -$r1

slt  $r0, $r1, $r2          $r0 <- 1 se $r1 <  $r2, $r0 <- 0 senão
slti $r0, $r1, C            $r0 <- 1 se $r1 <  C,   $r0 <- 0 senão
sle  $r0, $r1, $r2          $r0 <- 1 se $r1 <= $r2, $r0 <- 0 senão
seq  $r0, $r1, $r2          $r0 <- 1 se $r1 =  $r2, $r0 <- 0 senão
sne  $r0, $r1, $r2          $r0 <- 1 se $r1 <> $r2, $r0 <- 0 senão
```

MIPS em resumo

```
la      $r0, adr                    $r0 <- adr

lw      $r0, adr                    $r0 <- mem[adr]

sw      $r0, adr                    mem[adr] <- $r0

beq     $r0, $r1, label             salto se $r0 = $r1
beqz    $r0, label                  salto se $r0 = 0
bgt     $r0, $r1, label             salto se $r0 > $r1
bgtz    $r0, label                  salto se $r0 > 0


beqzal $r0, label                   salto se $r0 = 0, $ra <- $pc + 1
bgtzal $r0, label                   salto se $r0 > 0, $ra <- $pc + 1

j       label                       salto para label
jal     label                       salto para label, $ra <- $pc + 1
jr      $r0                         salto para $r0
jalr    $r0                         salto para $r0, $ra <- $pc + 1
```

MIPS em resumo

Registers Name	Number	Calling Convention Use	Callee must preserve?
$zero	$0	constant 0	N/A
$at	$1	assembler temporary	No
$v0-$v1	$2-$3	values for function returns and expression evaluation	No
$a0-$a3	$4-$7	function arguments	No
$t0-$t7	$8-$15	temporaries	No
$s0-$s7	$16-$23	saved temporaries	Yes
$t8-$t9	$24-$25	temporaries	No
$k0-$k1	$26-$27	reserved for OS kernel	N/A
$gp	$28	global pointer	Yes
$sp	$29	stack pointer	Yes
$fp	$30	frame pointer	Yes
$ra	$31	return address	N/A

MIPS - `syscall` - Chamadas ao sistema

Pode depender do simulador utilizado (consultar documentação)

Serviço	Código em $v0	Argumentos	Resultados
print_int	1	$a0 = o inteiro por imprimir	
print_float	2	$f12 = o float por imprimir	
print_double	3	$f12 = o double por imprimir	
print_string	4	$a0 = endereço da string por imprimir	
read_int	5		$v0 = o inteiro devolvido
read_float	6		$f0 = o float devolvido
read_double	7		$f0 = o double devolvido
read_string	8	$a0 = endereço da string por ler $a1 = comprimento da string	
sbrk/malloc	9	$a0 = quantidade de memória por alocar	endereço em $v0
exit	10	$v0 = o código devolvido	

MIPS - syscall - Chamadas ao sistema

```
#Print out integer value contained in register $t2
li $v0, 1           # load appropriate system call code into register $v0;
                    # code for printing integer is 1
move $a0, $t2       # move integer to be printed into $a0:  $a0 = $t2
syscall             # call operating system to perform operation

#Read integer value, store in RAM location with label int_value
#(presumably declared in data section)
li $v0, 5           # load appropriate system call code into register $v0;
                    # code for reading integer is 5
syscall             # call operating system to perform operation
sw $v0, int_value   # value read from keyboard returned in register $v0;
                    # store this in desired location

#Print out string (useful for prompts)
.data
string1 .asciiz "Print this.\n" # declaration for string variable,
                                # .asciiz directive makes string null terminated
.text
main: li $v0, 4         # load appropriate system call code into register $v0;
                        # code for printing string is 4
      la $a0, string1   # load address of string to be printed into $a0
      syscall           # call operating system to perform print operation
```

MIPS - um exemplo : Fib

```
#-----------------------------------------------
# fib - recursive Fibonacci function.
# http://www.cs.bilkent.edu.tr/~will/courses/
# CS224/MIPS%20Programs/fib_a.htm
#
#       a0 - holds parameter n
#       s0 - holds fib(n-1)
#       v0 - returns result
#-----------------------------------------------
# Code segment
        .text
fib:    sub $sp,$sp,12      # save registers on stack
        sw $a0,0($sp)
        sw $s0,4($sp)
        sw $ra,8($sp)

        bgt $a0,1,notOne
        move $v0,$a0        # fib(0)=0, fib(1)=1
        b fret              # if n<=1
```

MIPS - um exemplo : Fib

```
notOne: sub $a0,$a0,1      # param = n-1
        jal fib            # compute fib(n-1)
        move $s0,$v0       # save fib(n-1)

        sub $a0,$a0,1      # set param to n-2
        jal fib            # and make recursive call
        add $v0,$v0,$s0    # add fib(n-2)

fret:   lw $a0,0($sp)      # restore registers
        lw $s0,4($sp)
        lw $ra,8($sp)
        add $sp,$sp,12
        jr $ra

# data segment
        .data
endl:   .asciiz "\n"
```

Instruções macros para a máquina de pilhas

- Um registo $\$sp$ (stack pointer) que aponta para a primeira célula livre da pilha
- O endereçamento faz-se com base em bytes: uma palavra de 32 bits cabe em 4 bytes.

Funções de arquivo e de carregamento na pilha:

```
let pushr r = sub $sp, $sp, 4   |
              sw  r, 0($sp)

let popr r = lw   r, 0($sp)     |
             add $sp, $sp, 4
```