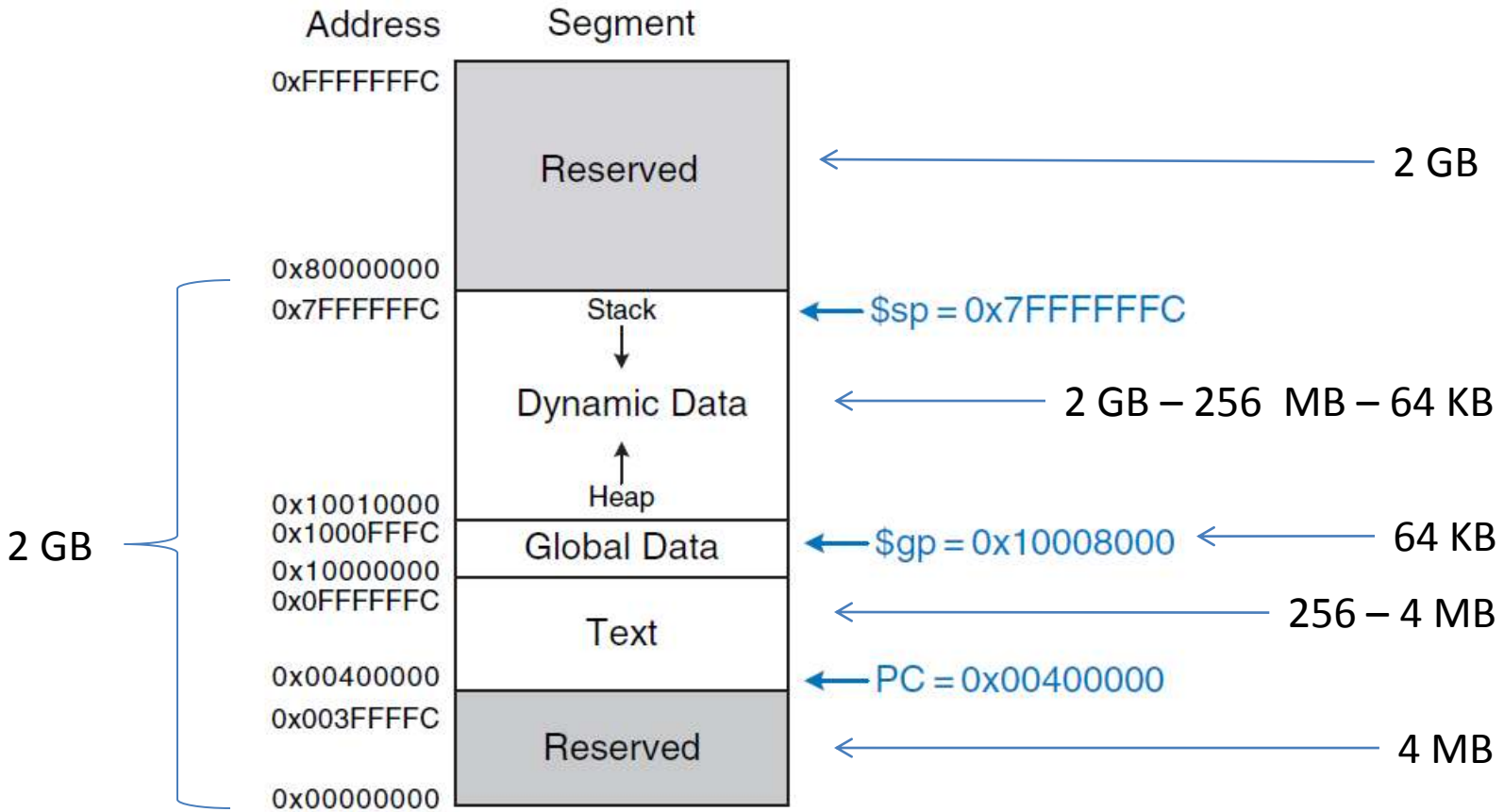
Registradores na Arquitetura MIPS

Table 6.1 MIPS register set

Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0–$v1	2–3	function return value
$a0–$a3	4–7	function arguments
$t0–$t7	8–15	temporary variables
$s0–$s7	16–23	saved variables
$t8–$t9	24–25	temporary variables
$k0–$k1	26–27	operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

Mapa de Memória da Arq. MIPS



Intruções MIPS em Assembly

HIGH LEVEL LANGUAGE	ASSEMBLY CODE	ASSEMBLY WITH REGISTERS
a = b + c	add a, b, c	# $s0 = a, $s1 = b, $s2 = c add $s1, $s2, $s3
a = b - c	sub a, b, c	# $s0 = a, $s1 = b, $s2 = c sub $s1, $s2, $s3

a=b + c - d	sub t, c, d
	add a, b, t
		# $s0 = a, $s1 = b,$s2 = c, $s3 = c
		sub $t0, $s2, $s3
		add $s0, $s1, $t0

Transferência de Dados Mem ↔ Reg

lw $s0, off($s1)

Carrega o registrador $s0 com o conteúdo do endereço formado por conteúdo de $s1 + off

sw $s0, off($s1)

Armazena o conteúdo do registrador $s0 no endereço formado por conteúdo de $s1 + off

MIPS opera com words (= 4 bytes) alinhados na memória. Por isso off($nn) deve ser múltiplo de 4

Little x Big Endian

Conteúdo da Memória	
Endereço	Dados
7	F0 h
6	DE h
5	BC h
4	9A h
3	78 h
2	56 h
1	34 h
0	12 h

MIPS pode ser configurada para operar com endereçamento Big ou Little Endian.

Após a execução de `lw $s0, 4($0)`

Se o endereçamento for Big Endian:

`$s0 = 0x9ABCDEF0`

Se o endereçamento for Little Endian:

`$s0 = 0xF0DEBC9A`

Acesso a Bytes na Memória

As instruções lb $nn, off($mm) e sb $nn, off($mm) operam de forma análoga a lw e sw, porém acessando um único byte.

O byte menos significativo de $nn é transferido da ou para a memória. O deslocamento off não precisa ser múltiplo de 4.

Supondo que o conteúdo de $s0 seja 0x23456789, qual será seu conteúdo após a execução das duas instruções abaixo no caso de endereçamento Little Endian e no caso de Big Endian?

```
sw $s0, 1($0)
lb  $s0, 1($0)
```

BE: $s0 = 0x00000045

LE: $s0 = 0x00000067

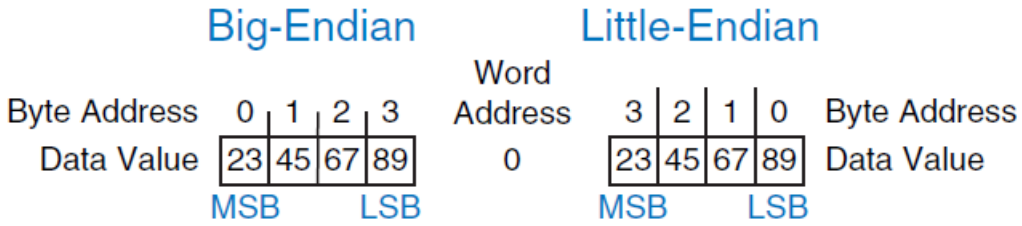
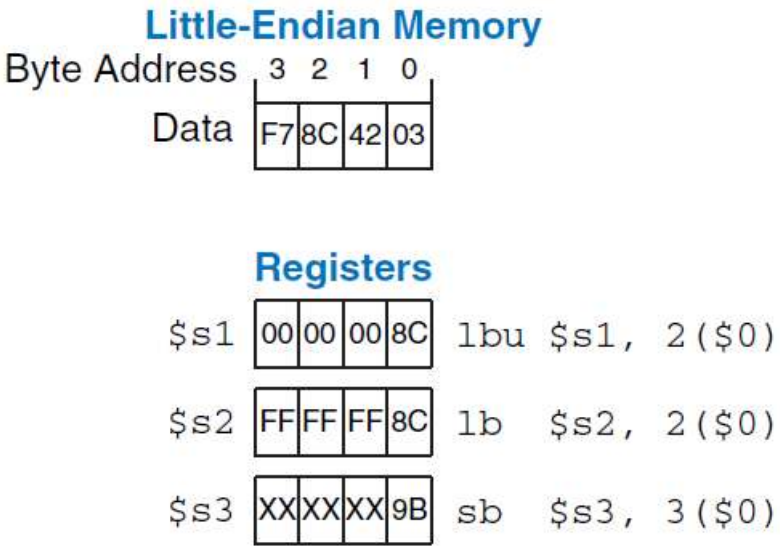


Figure 6.4 Big-endian and little-endian data storage

Acesso a Bytes – signed/unsigned

A instrução lb $nn, off($mm) considera o byte acessado como estando no formato com sinal. Para acesso a bytes sem sinal deve-se utilizar a instrução lbu $nnn, off($mm).



No exemplo acima, lbu preenche os bits 31:8 de $s1 com zeros. Já lb preenche os bits 31:8 de $s2 com sinal, que é negativo. A instrução sb desconsidera o valor dos bits 31:8 de $s3.

Constantes e Operandos Imediatos

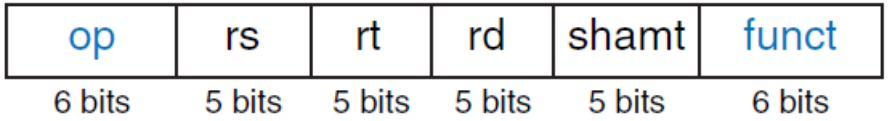
HIGH LEVEL LANGUAGE	ASSEMBLY CODE	ASSEMBLY WITH REGISTERS AND IMMEDIATES
a = a + 4	addi a, a, 4	# $s0 = a addi $s0, $s0, 4
a = a - 12	addi a, a, -12	# $s0 = a addi $s0, $s0, -12

Linguagem de Máquina

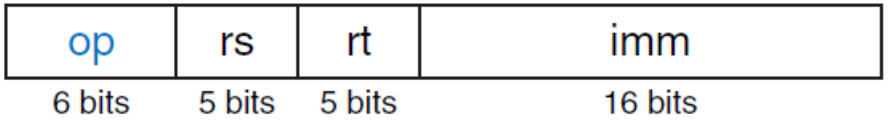
A Arquitetura MIPS codifica todas as instruções em 32 bits.

Há apenas 3 padrões de codificação de instruções:

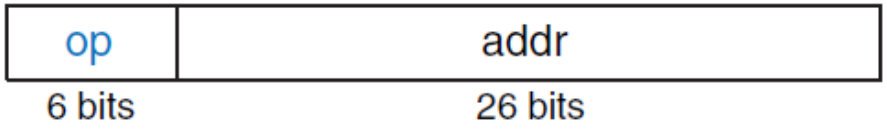
R – para instruções com operandos tipo registradores



I – para instruções com operandos imediatos



J – para instruções de saltos incondicionais (jump)



Instruções tipo “R”

Instruções tipo “R” seguem o padrão: funct rd, rs, rt

rd = registrador de destino
rs = primeiro registrador de origem (“source register”)
rt = segundo registrador de origem (letra “t” segue a letra “s”).

funct determina o tipo de operação – add, sub, etc.

Em linguagem de máquina, a sequência de registradores é diferente – veja os exemplos abaixo:

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
`add $s0, $s1, $s2`	0	17	18	16	0	32	000000	10001	10010	10000	00000	100000	(0x02328020)
`sub $t0, $t3, $t5`	0	11	13	8	0	34	000000	01011	01101	01000	00000	100010	(0x016D4022)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Figure 6.6 Machine code for R-type instructions

O campo “shamt” – shift amount – é usado apenas para “shifts”

Instruções tipo “I”

Instruções tipo “I” seguem o padrão: op rt, rs, imm

rt = registrador de destino (ou 2º de origem)

rs = registrador de origem (“source register”)

imm = constante imediata (16 bits)

op determina o tipo de operação – addi, lw, etc.

veja alguns exemplos abaixo:

Assembly Code	Field Values				Machine Code				
	op	rs	rt	imm	op	rs	rt	imm	
addi $s0, $s1, 5	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
addi $t0, $s3, -12	8	19	8	–12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
lw $t2, 32($0)	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
sw $s1, 4($t1)	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	

Figure 6.9 Machine code for I-type instructions

Instruções tipo “J”

Instruções tipo “J” seguem o padrão: op imm

Imm = especifica bits 27:2 do endereço de destino do salto (JTA)

op determina o tipo de salto incondicional – j, jal.

veja o exemplo para jal (= jump and link) abaixo:

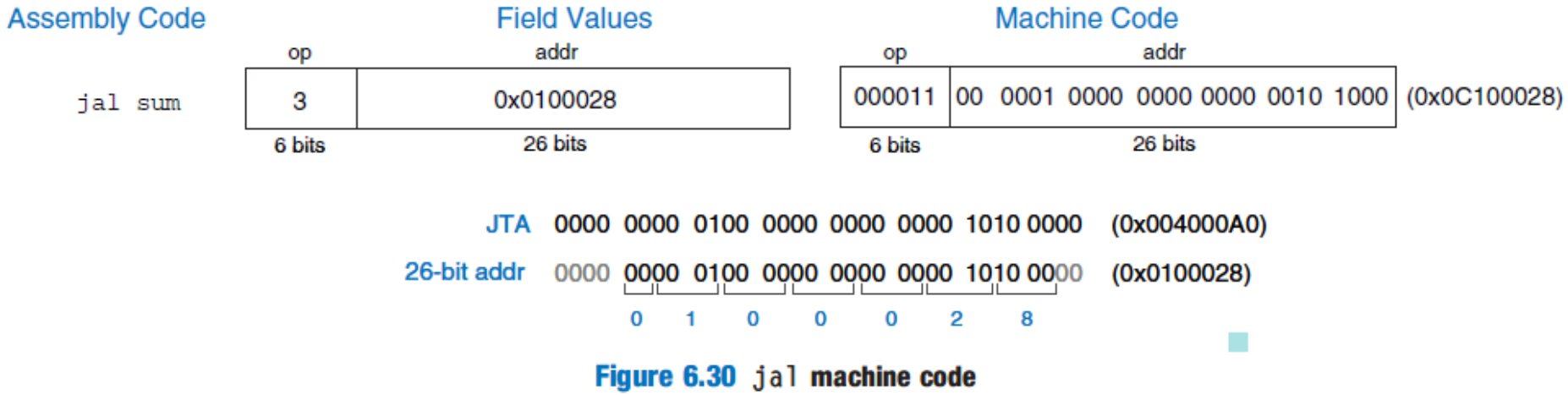


Figure 6.30 `jal` **machine code**

JTA é o Jump Target Address, formado pela seguinte regra:
bits 31:28 = bits 31:28 do PC+4, bits 27:2 = imm, bits 1:0 = 00

Intruções de desvio condicional

MIPS suporta duas instruções de desvio ("*branch*") condicional:

beq – "*branch if equal*"

bne – "*branch if not equal*"

Ambas são instruções do tipo "I" ("*immediate*")

beq rs, rt, imm

Se rs == rt salta para endereço especificado por imm (BTA).

bne rs, rt, imm

Se rs != rt salta para endereço especificado por imm (BTA).

O endereço de destino BTA ("*Branch Target Address*") é obtido a partir de imm e de PC + 4 da seguinte forma:

BTA = PC + 4 + SE(imm) x 4 (SE = "*Sign Extended*")

Esse endereçamento é chamado de *PC-Relative Addressing*

Intruções de desvio condicional

Code Example 6.12 CONDITIONAL BRANCHING USING `beq`

MIPS Assembly Code

```
  addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2         # $s1 = 1 << 2 = 4
  beq   $s0, $s1, target    # $s0 == $s1, so branch is taken   <- PC
  addi  $s1, $s1, 1         # not executed                     <- PC + 4
  sub   $s1, $s1, $s0       # not executed                     <- PC + 8

target:
  add   $s1, $s1, $s0       # $s1 = 4 + 4 = 8                  <- PC + 12
```

No exemplo acima, BTA = PC + 12, portanto imm = 2

BTA = PC + 4 + SE(imm) x 4 = PC + 4 + 2 x 4 = PC + 12

Código de máquina:

beq	$s0	$s1	imm
4	16	17	2
6 bits	5 bits	5 bits	16 bits

Desvio incondicional com jr

A instrução jr (“*jump register*”) não é do tipo “J”.

jr é do tipo “R” e utiliza um registrador de origem rs.

Code Example 6.15 UNCONDITIONAL BRANCHING USING `jr`

MIPS Assembly Code

```
0x00002000  addi  $s0, $0, 0x2010  # $s0 = 0x2010
0x00002004  jr    $s0              # jump to 0x00002010
0x00002008  addi  $s1, $0, 1       # not executed
0x0000200c  sra   $s1, $s1, 2      # not executed
0x00002010  lw    $s3, 44($s1)     # executed after jr instruction
```

Código de máquina:

op	rs	rt	rd	shamt	funct
0	16	x	x	x	8
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Chamada de Funções (MIPS)

Code Example 6.24 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code

```
int main()
{
  int y;

  . . .

  y = diffofsums(2, 3, 4, 5);

  . . .
}

int diffofsums(int f, int g, int h, int i)
{
  int result;

  result = (f + g) - (h + i);
  return result;
}
```

MIPS Assembly Code

```
# $s0 = y
main:
  . . .
  addi $a0, $0, 2    # argument 0 = 2
  addi $a1, $0, 3    # argument 1 = 3
  addi $a2, $0, 4    # argument 2 = 4
  addi $a3, $0, 5    # argument 3 = 5
  jal  diffofsums    # call function
  add  $s0, $v0, $0  # y = returned value
  . . .

# $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   # put return value in $v0
  jr  $ra            # return to caller
```

Argumentos em $a0 - $a3 Valores retornados em $v0 - $v1

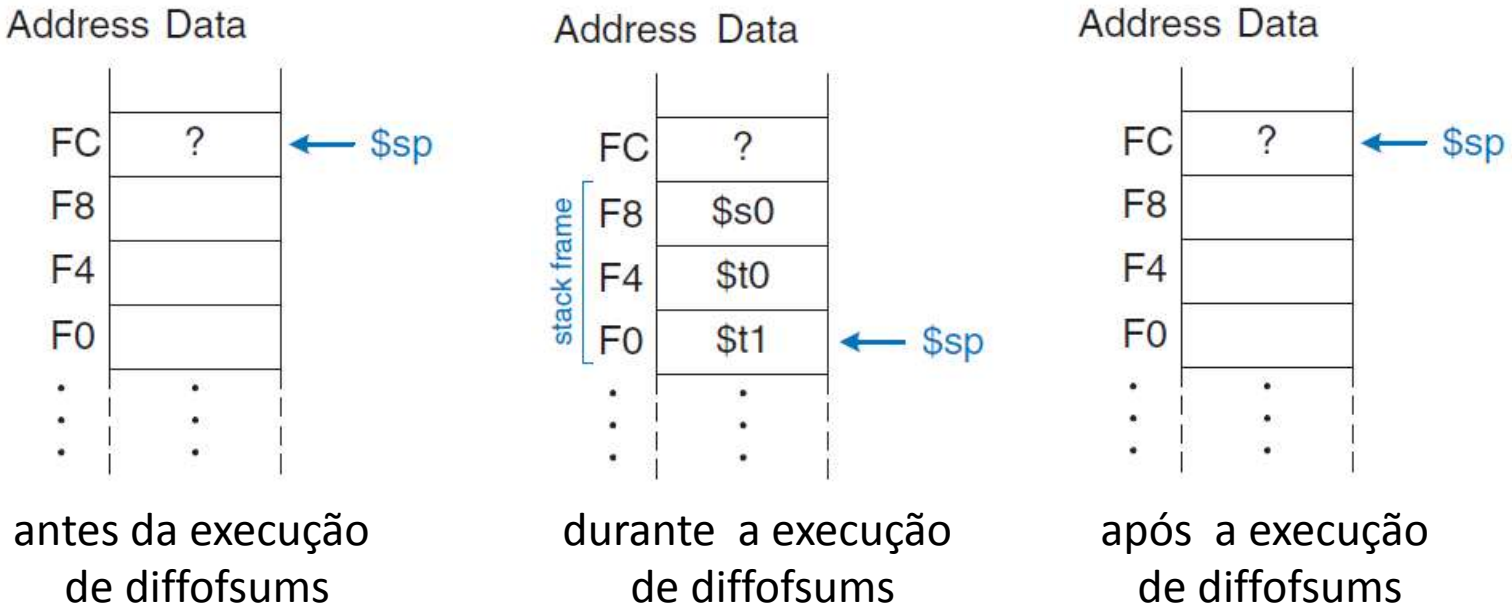
jal coloca endereço de retorno em $ra jr $ra retorna da função

Salvamento de registradores usados

No exemplo anterior *diffofsums* fez uso de $s0, $t0 e $t1

Caso a função chamadora (*main*) fizesse uso desses registradores, seu conteúdo seria perdido.

Portanto diffosums deveria preservar $s0, $t0 e $t1 armazenando-os na pilha e restaurando-os depois:



antes da execução de diffofsums

durante a execução de diffofsums

após a execução de diffofsums

Salvamento de registradores usados

Code Example 6.25 FUNCTION SAVING REGISTERS ON THE STACK

MIPS Assembly Code

```
# $s0 = result
diffofsums:
   addi $sp, $sp, -12 #  make space on stack to store three registers
   sw   $s0, 8($sp)   #  save $s0 on stack
   sw   $t0, 4($sp)   #  save $t0 on stack
   sw   $t1, 0($sp)   #  save $t1 on stack
   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   $t1, 0($sp)   #  restore $t1 from stack
   lw   $t0, 4($sp)   #  restore $t0 from stack
   lw   $s0, 8($sp)   #  restore $s0 from stack
   addi $sp, $sp, 12  #  deallocate stack space
   jr   $ra           #  return to caller
```

Salvamento de registradores usados

A versão anterior de diffofsums pode ser ineficiente, ao salvar registradores que a função chamadora não utiliza.

Para minimizar esse problema, a arquitetura MIPS divide os registradores em duas classes: *preservados* e *não preservados*.

Por ex. $s0 – $s7 são da classe preservados – daí seu nome "*saved*" já $t0 – $t9 são não preservados – daí seu nome "*temporaries*".

Registradores *preservados* devem ser obrigatoriamente salvos pela função chamada. Já os *não preservados* não. Caso a função chamadora faça uso de registradores não preservados, deverá salvá-los antes de chamar outra função.

Veja na próxima transparência qual a classe de cada registrador.

Salvamento de registradores usados

A tabela abaixo indica quais os registradores *preservados* e quais os *não preservados*:

Table 6.3 Preserved and nonpreserved registers

Preserved	Nonpreserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Return address: `$ra`	Argument registers: `$a0–$a3`
Stack pointer: `$sp`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

Utilizando essa convenção, a função *diffofsums* não precisa mais preservar $t0 e $t1, apenas $s0 (veja próxima transparência).

Caso *main* faça uso de $t0 e $t1, deverá salvá-los antes de chamar *diffofsums*.

Salvamento de registradores usados

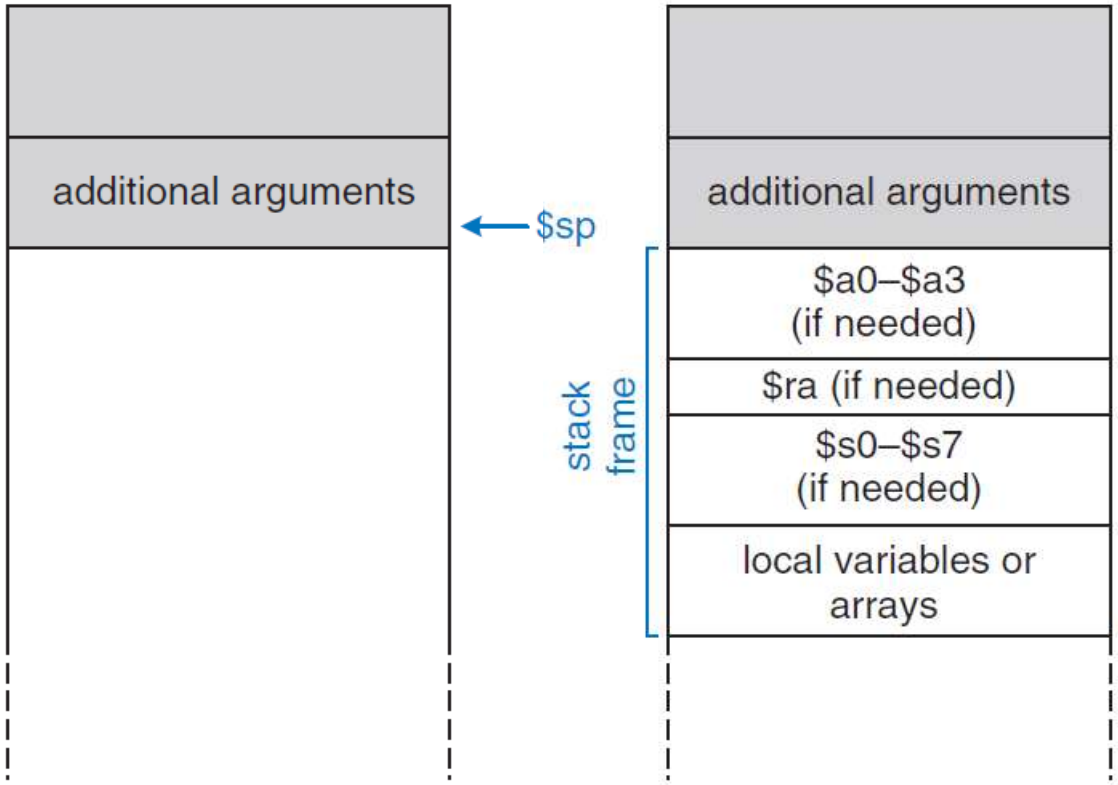
Code Example 6.26 FUNCTION SAVING PRESERVED REGISTERS ON THE STACK

MIPS Assembly Code

```
# $s0 = result
diffofsums
   addi  $sp, $sp, -4      #  make space on stack to store one register
   sw    $s0, 0($sp)       #  save $s0 on stack
   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 from stack
   addi  $sp, $sp, 4       #  deallocate stack space
   jr    $ra               #  return to caller
```

Argumentos Adicionais e Var. Locais

Caso seja necessário passar mais do que 4 argumentos para uma função, os primeiros 4 vão via $a0-$a3, os demais na pilha.



Pilha antes da chamada da função.

Pilha depois da chamada da função.

Se a função chamada fizer uso de registradores preservados, ou chamar outra função passando parâmetros em $a0-$a3, estes deverão também ser salvos na pilha.

Variáveis locais são alocadas na pilha logo após os registradores salvos.

Modos de Endereçamento (MIPS)

Register-Only Addressing: endereço de todos os operandos de origem e destino contidos em registradores. Usado por todas as instruções tipo "R".

Immediate Addressing: utiliza o campo imm de 16 bits, em conjunto com registradores para o endereçamento. Usado por certas instruções tipo "I" como addi ou lui.

Base Addressing: usado por instruções de acesso à memória como lw e sw. Endereço efetivo = Conteúdo de rs + SE(imm).

PC-Relative Addressing: usado na determinação do destino de salto em instruções de desvio condicional (branches).

BTA ("*Branch Target Address*") = PC + 4 + SE(imm) x 4

Pseudo-Direct Addressing: usado na determinação do destino de salto em instruções tip "J" (j e jal).

JTA ("*Jump Target Address*") utiliza 4 bits de PC e addr x 4

Pseudo Instruções

Como MIPS é uma arquitetura RISC, muitas instruções que se poderia imaginar importantes não estão disponíveis, pois podem ser implementadas por uma sequência de instruções disponíveis:

Table 6.5 Pseudoinstructions

Pseudoinstruction	Corresponding MIPS Instructions
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`clear $t0`	`add $t0, $0, $0`
`move $s2, $s1`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Essas instruções são chamadas de ***pseudo-instruções***

Exemplo: pseudo inst beq $t0,imm,dst

A instrução compara o conteúdo de $t0 com o valor imm e salta para dst e ambos forem iguais. O Assembler utiliza o registrador $at (“*assembler temporary*”) para implementar imm:

Table 6.6 Pseudoinstruction using `$at`

Pseudoinstruction	Corresponding MIPS Instructions
`beq $t2, imm15:0, Loop`	`addi $at, $0, imm15:0` `beq $t2, $at, Loop`

Outro exemplo é a instrução la reg, addr (“*load address into reg*”). Veja, no simulador MARS, no programa Fibonacci, como o Assembler implementou a instrução la $s0, fibs.

Exceções

MIPS utiliza um único vetor para tratamento de exceções, localizado em um endereço em 0x80000180 (2º GB da memória).

A rotina de tratamento, nesse endereço, obtém a causa da exceção via um registrador especial, denominado “Cause”.

Table 6.7 Exception cause codes

Exception	Cause
hardware interrupt	0x00000000
system call	0x00000020
breakpoint/divide by 0	0x00000024
undefined instruction	0x00000028
arithmetic overflow	0x00000030

O endereço de retorno é armazenado em outro registrador especial, denominado EPC (*Exception Program Counter*).

Exceções

Os registradores especiais Cause e EPC não fazem parte do conjunto de registradores de uso geral do MIPS ("*register file*").

Esses registradores ficam em uma unidade chamada *Coprocessor 0* também conhecida como *MIPS processor control*, e são acessados por meio da instrução mfc0 rd, reg (*move from coprocessor 0*).

As instruções syscall e break, também causam uma exceção. Nesse caso a rotina de tratamento, após salvar os registradores, obtém o valor de EPC com mfc0 e examina o código para diferenciar entre um *breakpoint* e uma chamada de sistema.

Para retornar, a rotina de tratamento de exceções executa o código

```
mfc0 $k0, EPC
jr $k0
```