Conjunto de Instruções
MIPS

Conjunto de Instruções

- **Instrução é uma palavra da linguagem de máquina**
- **ISA (Instruction Set Architecture)**
  - **Conjunto de instruções de uma máquina**
- **ISA MIPS**
  - **3 formatos de instruções**
  - **instruções de 3 operandos**

Programa em C	Assembly MIPS
a = b + c; d = a – c;	add a,b,c sub d,a,c

f = (g + h) – (i + j);	add t0,g,h
	add t1,i,j
	sub f,t0,t_1
	o compilador cria t0 e t1 .

- **Operandos**
  - **No MIPs os operandos das instruções são registradores**
    - **32 registradores de 32 bits**

Programa em C	Assembly MIPS

f = (g + h) – (i + j);	add $t0,$s1,$s2
	add $t1,$s3,$s4
	sub $s0,$t0,$t1

Instruções de Movimentação de Dados

- **Load e Store**
  - **lw : instrução de movimentação de dados da memória para registrador (load word)**
  - **sw: instrução de movimentação de dados do registrador para a memória (store word)**
  - **Exemplo:**

    Seja A um array de 100 palavras. O compilador associou à variável g o registrador $s1 e a h $s2, além de colocar em $s3 o endereço base do vetor. Traduza o comando em C abaixo.

    g = h + A[8];

Solução

Primeiro devemos carregar um registrador temporário com A[8]:

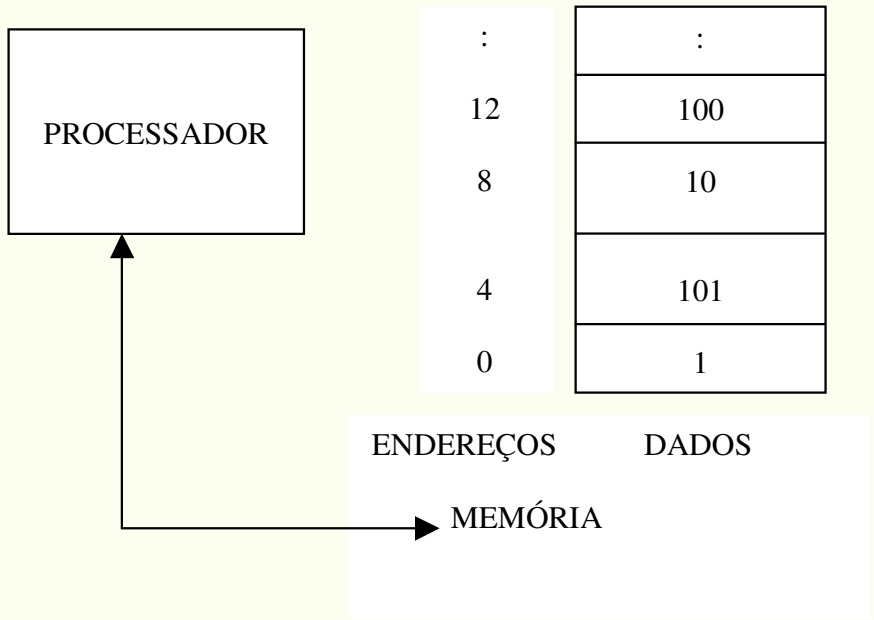
```
lw $t0, 8($s3)    #   registrador temporário $t0 recebe A[8]
```

Agora basta executar a operação:

```
add $s1,$s2,$t0    #  g = h + A[8]
```

MIPS - Organização da Memória

- No MIPS a memória é organizada em bytes, embora o endereçamento seja em palavras de 4 bytes (32 bits)



- **Exemplo: Suponha que h seja associado com o registrador $s2 e o endereço base do array A armazenado em $s3. Qual o código MIPS para o comando A[12] = h + A[8];?**

Solução:

```
lw    $t0,32($s3)    #  $t0 recebe A[8]
add   $t0,$s2,$t0    #  $t0 recebe h + A[8]
sw    $t0,48($s3)    # armazena o resultado em A[12]
```

Exemplo: Supor que o índice seja uma variável:

g = h + A[i];

onde: i é associado a $s4, g a $s1, h a $s2 e o endereço base de A a $s3.

Solução

```
add   $t1,$s4,$s4
add   $t1,$t1,$t1    #   $t1 recebe 4*i    ( porque ??? )

add  $t1,$t1,$s3     #   $t1 recebe o endereço de A[i]

lw    $t0,0($t1)      # $t0 recebe a[i]
add  $s1,$s2,$t0
```

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $t0, $t1, . . .	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

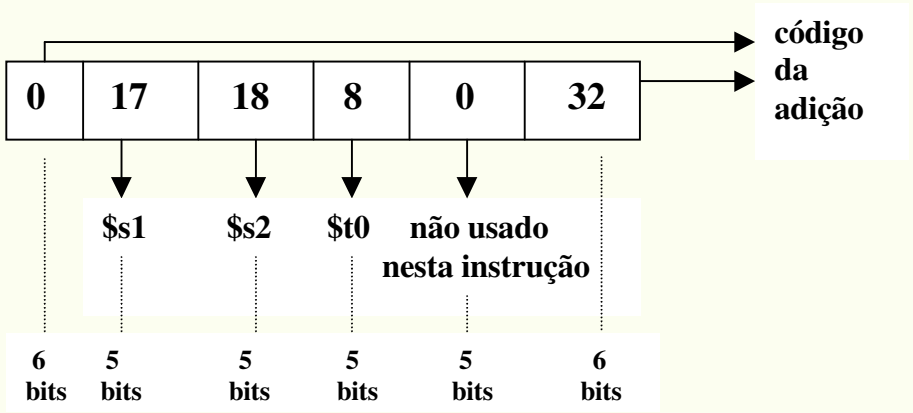
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

FIGURE 3.4 MIPS architecture revealed through section 3.3. Highlighted portions show MIPS assembly langua structures introduced in section 3.3.

Formato de Instruções

- **Formato da instrução add $t0,$s1,$s2**



- **Formato das instruções tipo R (R-type) e seus campos**

op	rs	rt	rd	shamt	funct

op ➔ **operação básica da instrução (opcode)**
rs ➔ **o primeiro registrador fonte**
rt ➔ **o segundo registrador fonte**
rd ➔ **o registrador destino**
shamt ➔ **shift amount, para instruções de deslocamento**
funct ➔ **function. Seleciona variações das operação especificada pelo opcode**

- **Formato das Instruções tipo I (I-type)**

- **Exemplo de instruções I-type**
  - **lw $t0, 32($s3)**

- **Codificação de Instruções MIPS**

Instrução	Formato	Op	rs	rt	rd	Sham t	func	end.
Add	R	0	reg	reg	reg	0	32	n.d
Sub	R	0	reg	reg	reg	0	34	n.d
Lw	I	35	reg	reg	n.d.	n.d	n.d	end.
Sw	I	43	reg	reg	n.d	n.d	n.d	end.

Exemplo:

Dê o código assembly do MIPS e o código de máquina para o seguinte comando em C: "A[300] = h + A[300];" , onde $t1 tem o endereço base do vetor A e $s2 corresponde a h.

```
lw    $t0,1200($t1)   #  $t0 recebe A[300]
add   $t0,$s2,$t0     #  $t0 recebe h + A[300]
sw    $t0,1200($t1)   #  A[300] recebe h + A[300]
```

- **Linguagem de máquina**

Op	rs	rt	rd	end/shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $s7 $t0, $t1, . . . , $t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0–$s7 map to 16–23 and $t0–$t7 map to 8–15.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 3.6 MIPS architecture revealed through section 3.4. Highlighted portions show MIPS machine language structures introduced in section 3.4. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; *shamt* field, which is unused in Chapter 3 and hence always is 0; and the *funct* field, which specifies the specific operation of R-format instructions. I-format keeps the last 16 bits as a single *address* field.

Instruções de desvio condicional

- **beq registrador1, registrador2, L1**
  - **se o valor do registrador1 for igual ao do registrador2 o programa será desviado para o label L1 (beq = branch if equal).**

- **bne registrador1, registrador2, L1**
  - **se o valor do registrador1 não for igual ao do registrador2 o programa será desviado para o label L1 (beq = branch if not equal).**

Exemplo - Compilando um comando IF.

Seja o comando abaixo:

```
      if ( i == j ) go to L1;
      f = g + h;
L1: f = f - i;
```

Supondo que as 5 variáveis correspondam aos registradores $s0..$s4, respectivamente, como fica o código MIPS para o comando?

Solução

```
      beq   $s3,$s4,L1     #   vá para L1 se i = j
      add   $s0,$s1,$s2    #   f = g + h, executado se i != j
L1:   sub   $s0,$s0,$s3    #   f = f – i, executado se i = j
```

Instrução de desvio incondicional

- **J L1**
  - quando executado faz com que o programa seja desviado para L1

Exemplo – Compilando um comando if-then-else

Seja o comando abaixo:

if (i == j) f = g + h; else f = g – h;

Solução

```
        bne $s3,$s4,Else    #  vá para Else se i != j
        add $s0,$s1,$s2     #  f = g + h, se i != j
        j Exit              #  vá para Exit
Else:   sub $s0,$s1,$s2     #  f = g – h, se i = j
Exit:
```

Loops

- **Usando IF**

Exemplo

```
Loop:   g = g + A[i];
        i = i + j;
        if ( i != h ) go to Loop
```

Solução

```
Loop:  add   $t1,$s3,$s3     #  $t1 = 2 * i
       add   $t1,$t1,$t1     #  $t1 = 4 * i
       add   $t1,$t1,$s5     #  $t1 recebe endereço de A[i]
       lw    $t0,0($t1)      # $t0 recebe A[i]
       add   $s1,$s1,$t0     # g = g + A[i]
       add   $s3,$s3,$s4     # i = i + j
       bne   $s3,$s2,Loop    # se i != h vá para Loo
```

- **Usando while**

Exemplo

```
while (save[i] == k)
      i = i + j;
```

Solução

Para i,j e k correspondendo a $s3,$s4 e $s5, respectivamente, e o endereço base do array em $s6, temos:

```
Loop:   add   $t1,$s3,$s3      #   $t1 = 2 * i
        add   $t1,$t1,$t1      #   $t1 = 4 * i
        add   $t1,$t1,$s6      #  $t1 = endereço de save[i]
        lw    $t0,0($t1)       # $t0 recebe save[i]
        bne   $t0,$s5,Exit     # va para Exit se save[i] != k
        add    $s3,$s3,$s4     # i = i + j
        j      Loop
Exit:
```

Instruções para teste de maior ou menor

- **slt reg_temp, reg1, reg2**
  - **se reg1 é menor que reg2, reg_temp é setado, caso contrário é resetado.**
  - **Nos processadores** MIPS o registrador $0 possui o valor zero ($zero).

Exemplo: Compilando o teste less than

Solução:

```
slt   $t0,$so,$s1        #  $t0 é setado se $s0 < $s1
bne $t0,$zero,Less   # vá para Less, se $t0 != 0 , ou seja a<b
```

Exemplo – Compilando o case/switch

Seja o comando abaixo:

```
switch (k)  {
      case 0:  f = f + j; break;
      case 1:  f = g + h; break;
}
```

Solução: supor que $t2 tenha 2 e f..k = $s0..$s5, respectivamente.

```
        slt     $t3,$s5,$zero       # teste se k < 0
        bne     $t3,$zero,Exit      # se k < 0 vá para Exit

        slt     $t3,$s5,$t2         # teste se k < 2
        beq     $t3,$zero,Exit      # se k>=2 vá para Exit

        add     $t1,$s5,$s5         # $t1 = 2 * k
        add     $t1,$t1,$t1         #  $t1 = 4 * k

        # assumindo que 4 palavras na memória, começando
        no endereço contido em $t4, tem endereçamento
        correspondente a L0, L1, L2

        add     $t1,$t1,$t4         #  $t1 = endereço de tabela[k]
        lw      $t0,0($t1)          #  $t0 = tabela[k]
        jr      $t0         # salto para endereço carregado em $t0

L0:     add     $s0,$s3,$s4         # k = 0 ➔ f = i + j
        j       Exit

L1:     add     $s0,$s1,$s2         #  k = 1 ➔ f = g + h

Exit:
```

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $s7 $t0, $t1, . . . , $t7, $zero	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0–$s7 map to 16–23 and $t0–$t7 map to 8–15. MIPS register $zero always equals 0.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $t1	go to $t1	For *switch* statements

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see section 3.8)
jr	R	0	9	0	0	0	8	jr $t1
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 3.9 MIPS architecture revealed through section 3.5. Highlighted portions show MIPS structures introduced in section 3.5. The J-format, used for jump instructions, is explained in section 3.8. Section 3.8 also explains the proper values in address fields of branch instructions.

Suporte a Procedimentos

- **Para a execução de um procedimento deve-se:**
  - **Colocar os parâmetros em um local onde o procedimento possa acessá-los**
  - **Transferir o controle ao procedimento**
  - **Adquirir os recursos necessários ao procedimento**
  - **Executar a tarefa**
  - **Colocar o resultado em um local onde o programa possa acessá-lo**
  - **Retornar o controle ao ponto onde o procedimento foi chamado**

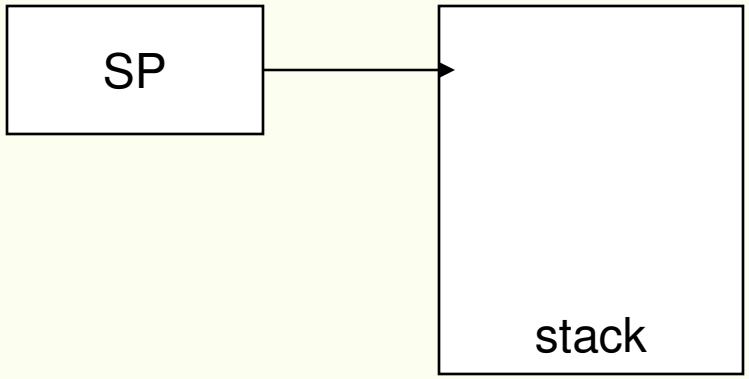
- **Para este mecanismo, o MIPS aloca seus registradores, para chamada de procedimentos, da seguinte maneira:**
  - **$a0 .. $ a3 ➔ 4 registradores para passagem de argumentos**
  - **$v0 .. $v1 ➔ para retornar valores**
  - **$ra ➔ para guardar o endereço de retorno**

- **Instrução para chamada de procedimento**
  - **jal End_proc - (jump-and-link) ➔ desvia para o procedimento e salva o endereço de retorno (PC+4) em $ra (return address - $31)**

- **Instrução para retorno de chamada de procedimento**
  - **jr $ra ➔ desvia para o ponto de onde foi chamado o procedimento**

- **Qual o problema para chamadas aninhadas ==. $ra é destruído.**

- **Qual a solução ➔ utilizar uma pilha (LIFO)**



  - **Registrador utilizado para o stack pointer ➔ $sp ($29)**

- **Exemplo:**
  - **Os parâmetros g, h, i e j correspondem a $a0 .. $a3, respectivamente e f a $s0. Antes precisaremos salvar $s0, $t0 e $t1 na pilha, pois serão usados no procedimento**

Exemplo

Seja o procedimento abaixo:

```
int exemplo (int g, int h, int i, int j)
{
        int f;

        f = (g + h) – (i + j);
        return f;
}
```

```
sub $sp,$sp,12   # ajuste do sp para empilhar 3 palavras
sw  $t1,8($sp)   # salva $t1 na pilha
sw  $t0,4($sp)   # salva $t0 na pilha
sw  $s0,0($sp)   # salva $s0 na pilha
```

No procedimento

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

Para retornar o valor f

```
add $v0,$s0,$zero
```

Antes do retorno é necessário restaurar os valores dos registradores salvos na pilha

```
lw $s0, 0($sp)
lw $t0, 4($sp)
lw $s1, 8($sp)
add $sp,$sp,12
```

Retornar

```
jr  $ra
```

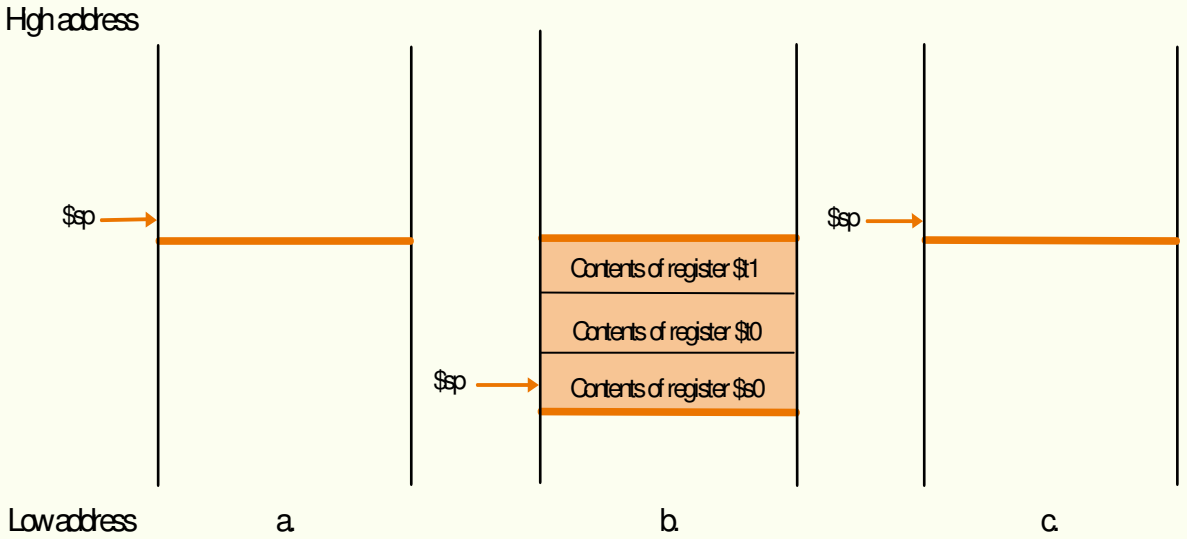


Figura 3.10 – Valores de sp antes, durante e depois da chamada do procedimento

- **Observações**
  - **$t0 .. $t9 ➔ 10 registradores temporários que não são preservados em uma chamada de procedimento**
  - **$s0 .. $s7 ➔ 8 registradores que devem ser preservados em uma chamada de procedimento**

Exemplo – procedimento recursivo

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

Supor n correspondente a $a0

```
fact:
   sub  $sp,$sp,8        # ajuste da pilha
   sw   $ra,4($sp)       # salva o endereço de retorno
   sw   $a0,0(sp)        #salva o argumento n

   slt  $t0,$a0,1        #teste para n<1
   beq  $t0,$zero,L1     #se n>=1, vá para L1

   add  $v0,$zero,1      #retorna 1 se n < 1
   add  $sp,$sp,8        #pop 2 itens da pilha
   jr   $ra
L1:
   sub  $a0,$a0,1        #n>=1, n-1
   jal  fact             #chamada com n-1

   lw   $a0,0($sp)       #retorno do jal; restaura n
   lw   $ra,4($sp)
   add  $sp,$sp,8

   mult $v0,$a0,$v0      #retorna n*fact(n-1)

   jr   $ra
```

- **Alocação de espaço para novos dados**
  - **O segmento de pilha que contém os registradores do procedimento salvos e as variáveis locais é chamado de procedure frame ou activcation record. O registrador $fp é usado para apontar para a primeira palavra deste segmento.**

- **Figura 3.11 – O que é preservado ou não numa chamada de procedimento.**

Registradores Preservados	Registradores Não Preservados
Salvos: $s0-$s7	Temporários: $t0-$t7
Apontador para pilha: $sp	Argumentos: $a0-$a3
Endereço de retorno: $ra	Valores de Retorno: $v0-$v1
Pilha acima do Apontador para pilha	Pilha abaixo do Apontador para pilha

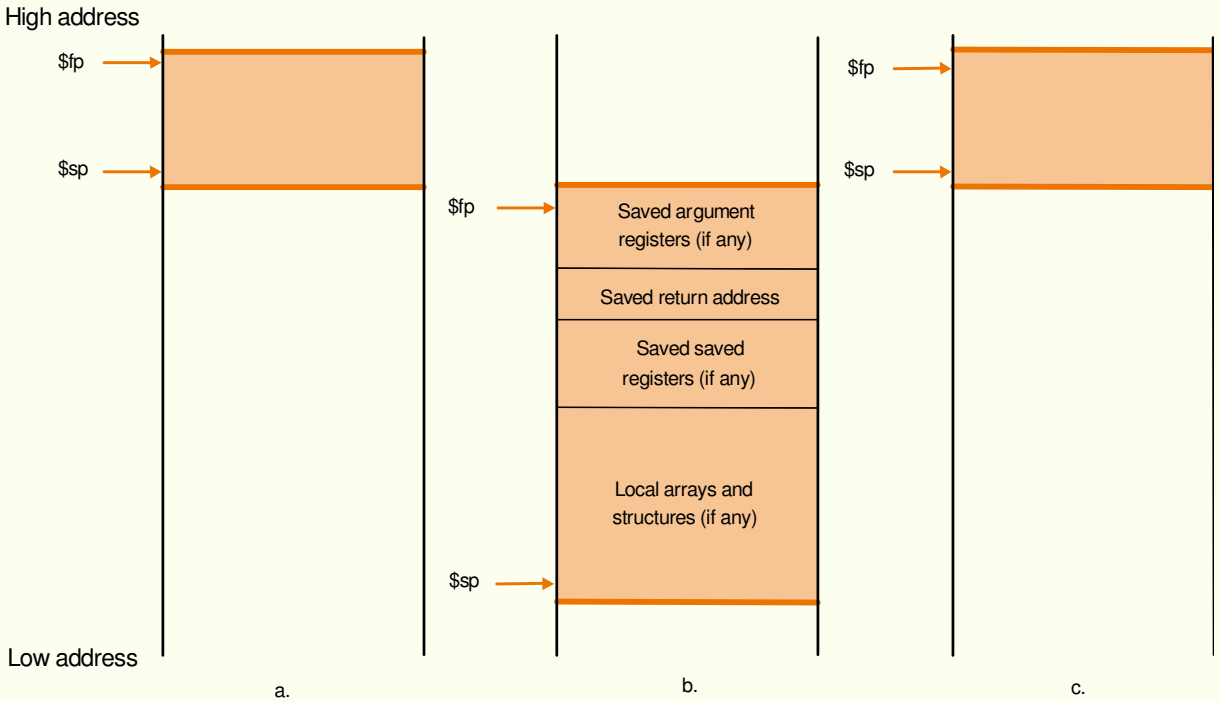


Figura 3.12 – Ilustração da pilha antes, durante e depois da chamada de procedimento.

- **Figura 3.13 – Convenção de registradores no MIPS**

Nome	Número	Uso	Preservado em chamadas?
$zero	0	Constante 0	n.d
$v0-$v1	2-3	Resultados e avaliações de expressões	Não
$a0-$a3	4-7	Argumentos	Sim
$t0-$t7	8-15	Temporários	Não
$s0-$v7	16-23	Salvos	Sim
$t8-$t9	24-25	Temporários	Não
$gp	28	Ponteiro global	Sim
$sp	29	Ponteiro para pilha	Sim
$fp	30	Ponteiro para frame	Sim
$ra	31	Endereço de retorno	Sim

MIPS operands

Name	Example	Comments
32 registers	$s0–$s7, $t0–$t9, $zero, $a0–$a3, $v0–$v1, $gp, $fp, $sp, $ra	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp (29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1=1; else $s1 = 0	Compare less than; for beq, bne
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see section 3.8)
jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					jal 10000 (see section 3.8)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 3.14 MIPS architecture revealed through section 3.6. Highlighted portions show MIPS assembly language structures introduced in section 3.6. The J-format, used for jump and jump-and-link instructions, is explained in section 3.8. This section also explains why putting 25 in the address field of beq and bne machine language instructions is equivalent to 100 in assembly language.

Endereçamento no MIPS

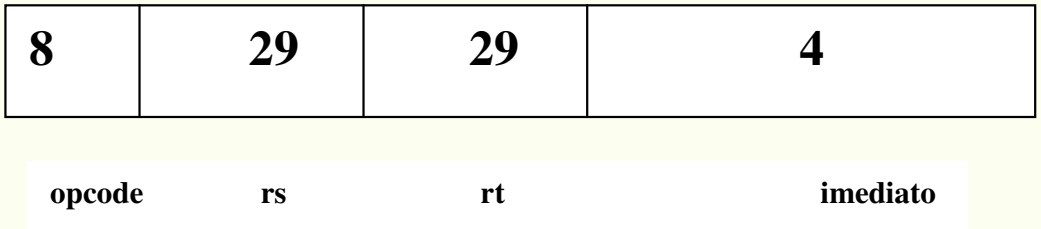
- **Operandos constantes ou imediatos**
  - **Para somar uma constante ou um imediato**

```
lw    $t0,end_constante($zero)  # end_constante = endereço
                                   da cosntante na memória
add  $sp,$sp,$t0
```

Observação: Outra forma é permitir instruções aritméticas do tipo I (constantes com 16 bits)

- **Exemplo**
  - A instrução add do tipo I é chamada addi (add immediate). Para somar 4 a $sp temos:

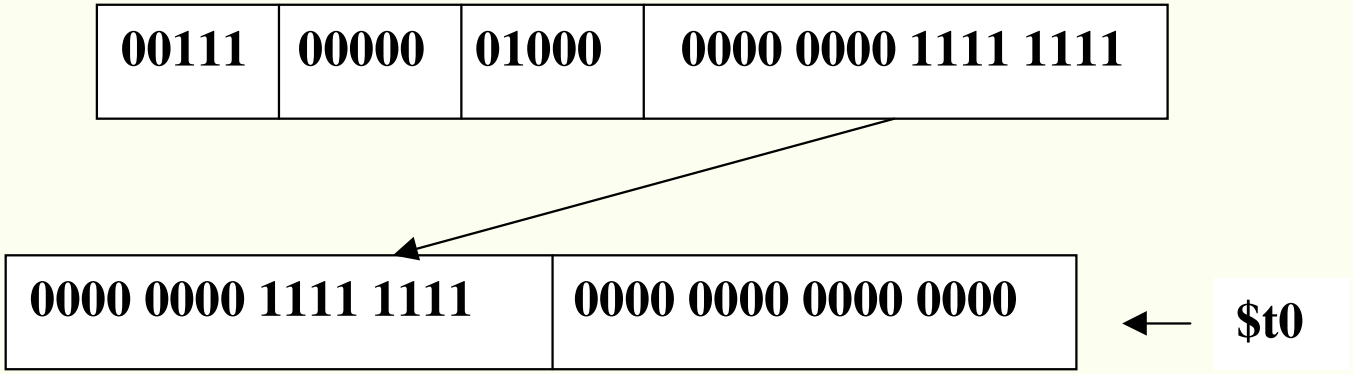
addi $sp,$sp,4



  - Em comparações
    - slti $t0,$s2,10 # $t0 =1 se $s2 < 10

Instruções de Carga

lui $t0,255 #load upper immediate



Exercício:

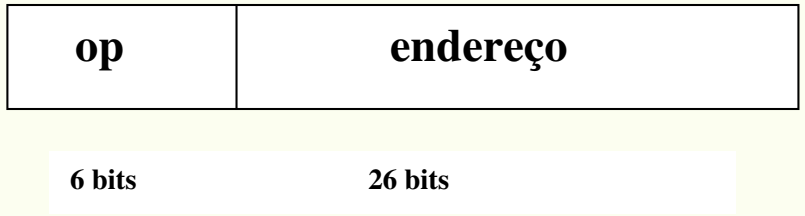
- **Qual o código MIPS para carregar uma constatnte de 32 bits no registrador $s0 ?**
  0000 0000 0011 1101 0000 1001 0000 0000

Solução

```
lui    $s0,61          # 61_10 = 0000 0000 0011 1101_2
addi   $s0,$s0,2304    # 2304_10 = 0000 1001 0000 0000_2
```

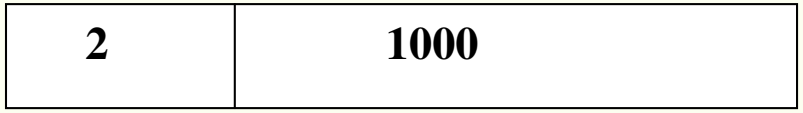
Endereçamento em branches e jumps

- **Instruções J-type**



Exemplo

j 1000 # vá para 1000



Endereçamenteo relativo ao PC

- **Branch (I-type)**

Exemplo

bne $s0,$s1,Exit

5	16	17	Exit

PC ← PC + Exit

Exemplo

```
Loop:
      add  $t1,$s3,$s3      # $t1 = 2 * i
      add  $t1,$t1,$t1      # $t1 = 4 * i
      add  $t1,$t1,$s6      # $t1 = endereço de save[i]
      lw   $t0,0($t1)       # $t0 recebe save[i]
      bne  $t0,$s5,Exit     #vá para Exit se save[i] != k
      add  $s3,$s3,$s4      #i = i+j
      j    Loop
Exit:
```

Assumindo que o loop está alocado inicialmente na posição 80000 na memória, teremos a seguinte seqüência de código em linguagem de máquina:

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	21	9	0	32
80012	35	9	8	0		
80016	5	8	21	8		
80020	0	19	20	19	0	32
80024	2	80000				
80028	.....					

Exemplo

Dado o branch abaixo, rescrevê-lo de tal maneira a oferecer um offset maior

```
beq  $s0,$s1,L1
```

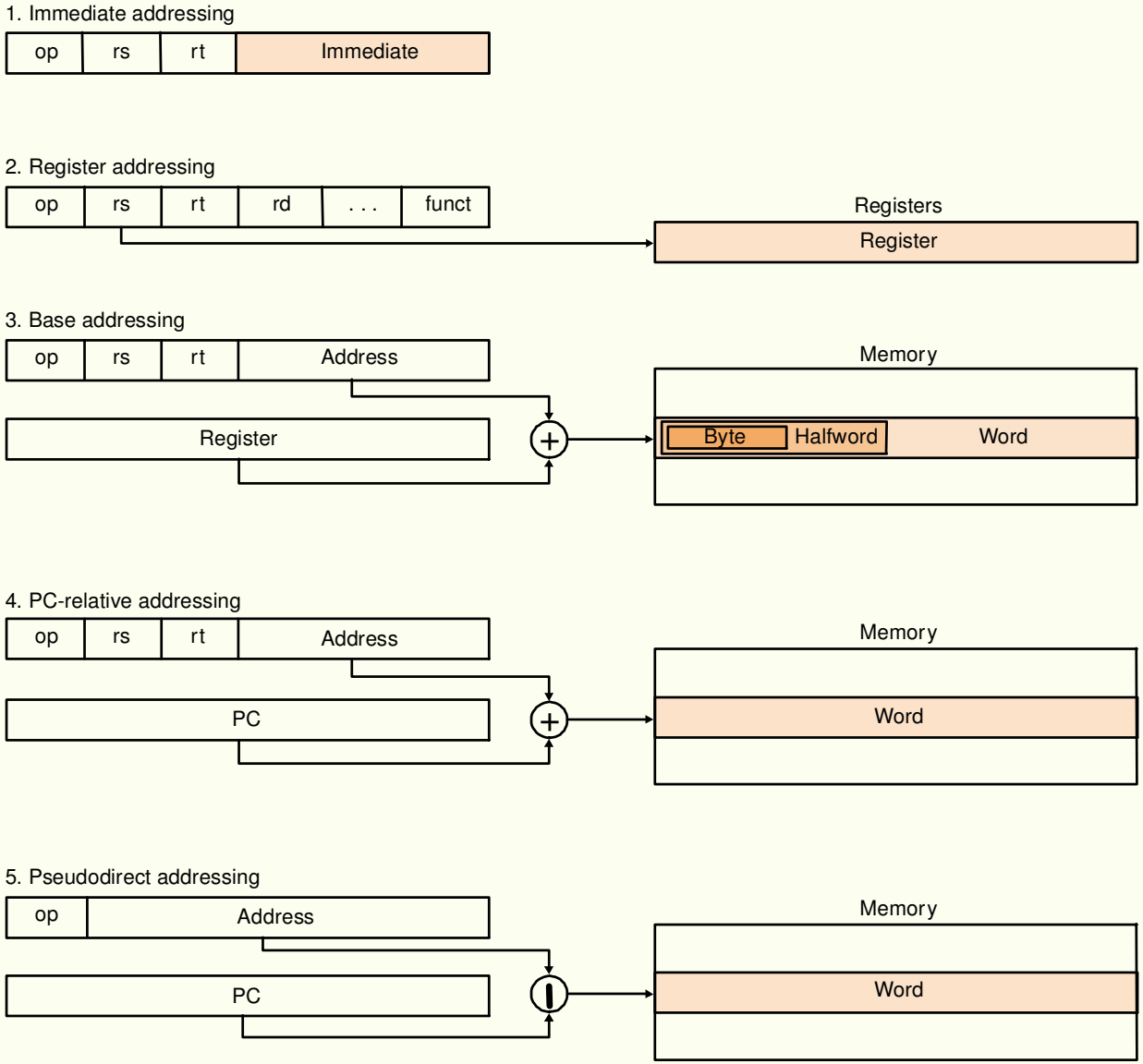
Solução

```
     bne $s0,$s1,L2
     j     L1
L2:
```

Endereçamento MIPS - Resumo

- **Endereçamento por registrador → o operando é um registrador**
- **Endereçamento por base ou deslocamento → o operando é uma localização de memória cujo endereço é a soma de um registrador e uma constante na instrução**
- **Endereçamento imediato => onde o operando é uma constante na própria instrução**
- **Endereçamento relativo ao PC → onde o endereço é a soma de PC e uma constante da instrução**
- **Endereçamento pseudodireto →onde o endereço de desvio (26 bits) é concatenado com os 4 bits mais significativos do PC**

Figura 3.17 – Modos de endereçamento do MIPS



- **Figura 3.18 – Codificação das instruções do MIPS**

op(31:26)								
28–26 / 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	lh	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	sh	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26)=010000 (TLB), rs(25:21)								
23–21 / 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)								
2–0 / 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	sll		srl	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	nor
5(101)			set l.t.	sltu				
6(110)								
7(111)								

FIGURE 3.18 MIPS instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure `load word` is found in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, `R-format` in row 0 and column 0 (op = 000000_{two}) is defined in the bottom part of the figure. Hence `subtract` in row 4 and column 2 of the bottom section means that the funct field (bits 5–0) of the instruction is 100010_{two} and the op field (bits 31–26) is 000000_{two}. The `FlPt` value in row 2, column 1 is defined in Figure 4.48 on page 292 in Chapter 4. `Bltz/gez` is the opcode for four instructions found in Appendix A: `bltz`, `bgez`, `bltzal`, and `bgezal`. Instructions given in full name using color are described in Chapter 3, while instructions given in mnemonics using color are described in Chapter 4. Appendix A covers all instructions.

Figura 3.19 – Formato de instruções do MIPS

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

FIGURE 3.19 MIPS instruction formats in Chapter 3. Highlighted portions show instruction formats introduced in this section.

Figura 3.20 – Linguagem assembly do MIPS

MIPS operands

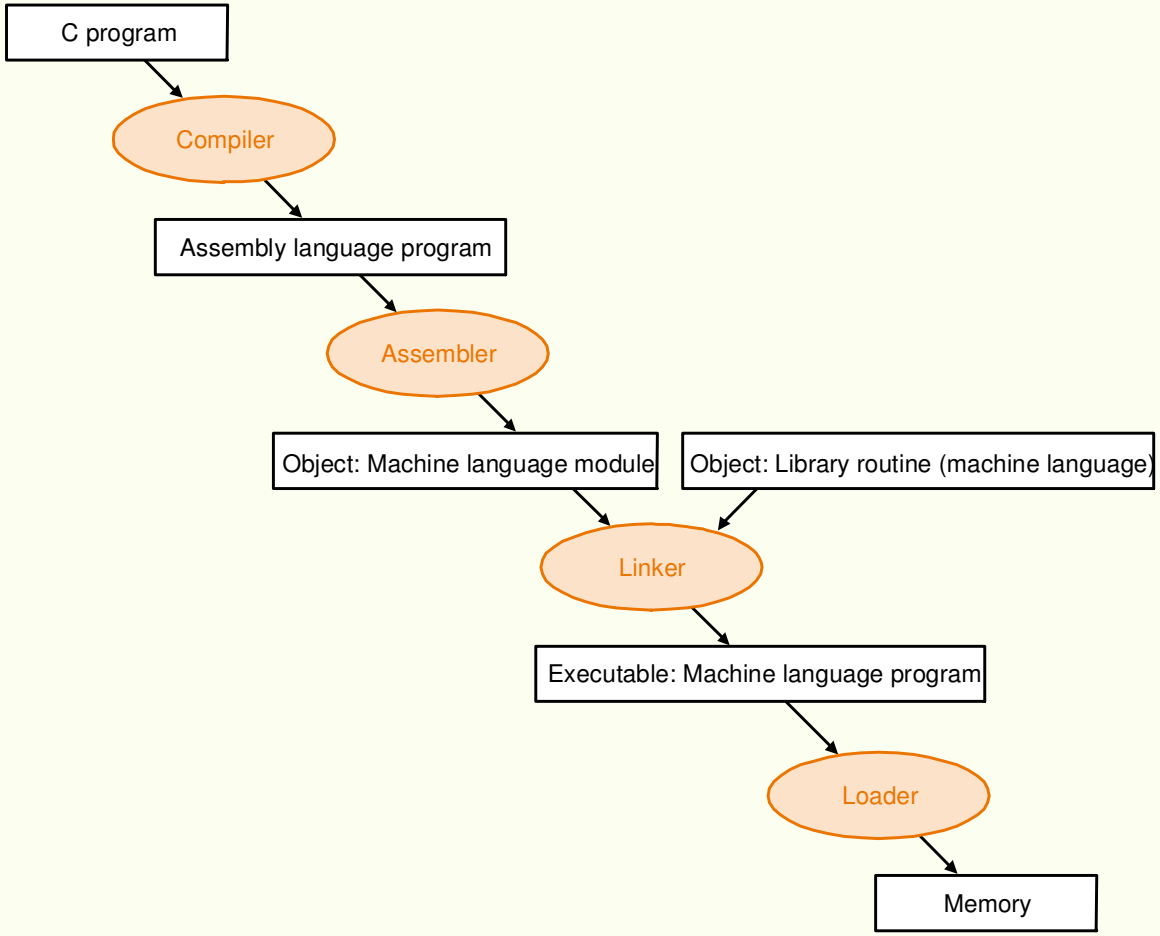
Name	Example	Comments
32 registers	$s0–$s7, $t0–$t9, $zero, $a0–$a3, $v0–$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

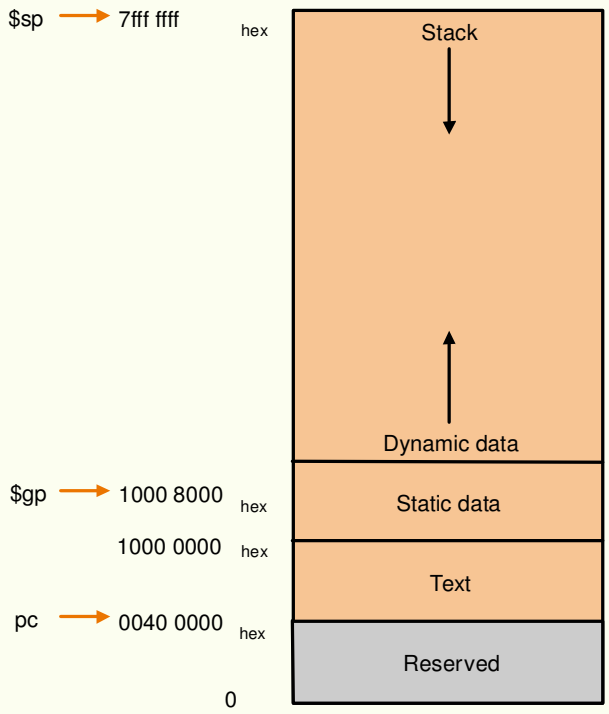
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	lb $s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 3.20 MIPS assembly language revealed in Chapter 3. Highlighted portions show portions from sections 3.7 and 3.8.

- **Traduzindo um Programa**





- **Quando da tradução de C para assembly deve-se fazer:**
  - **alocar registradores para as variáveis do programa**
  - **produzir código para o corpo do procedimento**
  - **preservar os registradores durante a chamada do procedimento**

PowerPC (Motorola, Apple, IBM)

- 32 registradores de 32 bits, instruções de 32 bits
- Indexed addressing
  - example: `lw $t1,$a0+$s3  #$t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?

- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: `lwu $t0,4($s3)`
    `#$t0=Memory[$s3+4];$s3=$s3+4`
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register "`bc Loop`"

decrement counter, if not 0 goto loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
  (mostly designed for higher performance)
- 1997: MMX is added

"This history illustrates the impact of the "golden handcuffs" of compatibility

"adding new features as someone might add clothing to a packed bag"

"an architecture that is difficult to explain and impossible to love"

A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

"what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective"

Conclusão

- **Erro: instruções mais poderosas aumentam desempenho**
- **VAX:**
  - **CALL: salva endereço de retorno, nº de parâmetros, quaisquer registros modificados e valor antigo do SP**
  - **instrução para apagar lista duplamente ligada**
- **IBM 360:**
  - **10 instruções mais freqüentes: 80% das ocorrências**
  - **16 instruções mais freqüentes: 90% das ocorrências**
  - **21 instruções mais freqüentes: 95% das ocorrências**
  - **30 instruções mais freqüentes: 99% das ocorrências**
- **MIPS**

classe	instr	gcc	spice
arit.	`add, sub, addi`	48%	50%
transf. dados	`lw, sw, lb, sb, lui`	33%	41%
desvio cond.	`beq, bne, slt, slti`	17%	8%
jump	`j, jr, jal`	2%	1%

Máquinas de 0, 1, 2 e 3 endereços

X = A * B + C * C onde X, A, B, C são endereços de posições de memória

Um endereço

```
LOAD A
MULTIPLY B
STORE T
LOAD C
MULTIPLY C
ADD T
STORE X
```

AC ⇐ AC op END

Dois endereços

```
MOVE T, A
MULTIPLY T, B
MOVE X, C
MULTIPLY X, C
ADD X, T
```

END1 ⇐ END1 op END2

Três endereços

```
MULTIPLY T, A, B
MULTIPLY X, C, C
ADD X, X, T
```

END1 ⇐ END2 op END3

Load-Store

```
load R1, A
load R2, B
load R3, C
mult R1, R1, R2
mult R3, R3, R3
add R1, R1, R3
store R1, X
```

R1 ⇐ R2 op R3

Zero endereços

```
push A
push B
mult
push C
push C
mult
add
pop X
```

tos ⇐ tos op tos

Máquinas de 0, 1, 2 e 3 endereços

- **Qual é o melhor?**
  - **tamanho do código fonte**
  - **tamanho do código objeto**
  - **tempo de execução**
  - **simplicidade e desempenho do hardware para suportar arquitetura**