Procedimentos

- Estruturação de programas.
- Mais fácil de entender.
- Reutilização.
- Dividir em partes menores.

Panorama

- Funções em C
- Instruções MIPS para Funções
- A pilha (Stack)
- Convenções de Registradores
- Exemplo


```
main ( )
{
  int x, y, z;
  x=1; y=2;
  z = soma (x,y);

}
```

De quais informações o compilador/programador deve manter o registro?

```
int soma (int parcela1, int  parcela2)
{
return (parcela1 + parcela2);
}
```

Como fazer essa parte em MIPS?


```
main ( )
{
  int x, y, z;
  x=1; y=2;
  z = soma (x,y);

}
```

```
int soma (int parc1, int  parc2)
{
return (parc1 + parc2);
}
```

MEMÓRIA

Endereço	instrução
1000	
1004	
1008	
1012	
1016	
1020	
1024	
1028	
1032	
1036	

Contabilidade de Chamada de Função

- Registradores tem papel fundamental para manter registro de informações nas chamadas de funções.
- Convenção de Registradores:
  - Endereço de retorno `$ra`
  - Argumentos `$a0, $a1, $a2, $a3`
  - Valor de Retorno `$v0, $v1`
  - Variáveis locais `$s0, $s1, • •, $s7`
- **Uso da instrução, jump register ou jr**

Endereço instrução

```
1000   addi  $s1, $zero, 1       #  i = 1
1004   addi  $s2, $zero, 2       #  j = 2
1008   add  $a0, $zero, $s1      #  $a0  = i
1012   add  $a1, $zero, $s2      #  $a1  = 2
1016   addi $ra, $zero,  1024    #  $ra = 1024, end. de retorno da funçao
1020   j soma                   #  salta para o endereço soma
1024   ……                       #  k = soma(i,j)

 ….


soma:  5000   add $vo, $a0, $a1      # $vo = $a0 + $a1
       5004   jr  $ra                #  salta para o endereço contido em $ra
```


Instruções de Suporte para Funções

- Instrução única para **pular e salvar o endereço de retorno**: jump and link (`jal`)
- Antes:

```
1016   addi $ra, $zero,  1024    #  $ra = 1024, end. de retorno da funçao
1020   j soma                    #  salta para o endereço soma
```

- Depois:

```
1016   jal soma                  #  salta para o endereço soma
```

- Por que ter uma `jal`?
  - Torne o **caso comum rápido**: funções são muito comuns.

Instruções de Suporte para Funções

- Sintaxe de `jal` (jump and link) é a mesma de `j` (jump):
  - `jal label`
- `jal` deveria na verdade ser chamada `laj` de link and jump•!
  - Passo 1 (link): **Salva o endereço da próxima instrução** em $ra (Por que a próxima instrução? Por que não a corrente?)
  - Passo 2 (jump): **Pule para o label** dado

Instruções de Suporte para Funções

- Sintaxe de **`jr`** (jump register):
  - `jr  register`
- Ao invés de prover um label para pular para, a instrução `jr` provê um **registrador** que **contém** um **endereço** para onde pular.
- Útil somente se nós sabemos o **endereço exato** para onde pular: raramente aplicável.
- Muito útil para **chamadas de funções**:
  - **`jal` guarda** o **endereço de retorno** no registrador ($ra) (**chamada de uma função**)
  - **`jr` pula** de volta **para** aquele **endereço (retorno da função)**


```
main ( )
{
...
   x = 1;
   y = 2;
   z = soma ( x , y );

}

int soma (int r, int s)
{
   return ( r + s )
}
```

Mapeamento:
x → $s1
y → $s2
z → $s3


```
main ( )
{
...
   x = 1;
   y = 2;
   z = soma ( x , y );
}

int soma (int r, int s)
{
   return ( r + s )
}
```

Mapeamento:
x → $s1
y → $s2
z → $s3

```
.text
.globl main

main:
addi $s1, $zero, 1
addi $s2, $zero, 2
add $a0, $zero, $s1
add $a1, $zero, $s2
jal soma
nop
add $s3, $zero, $v0

soma:
add $v0, $a0, $a1
jr $ra
```

Definindo a função e retornando


```
main ( )
{
...
   x = 1;
   y = 2;
   z = soma ( x , y );
   k = soma (x , x );
   m = soma (y , y );
   x = soma ( k , m)
}

int soma (int r, int s)
{
   return ( r + s )
}
```

Mapeamento:
x → $s1
y → $s2
z → $s3
k → $s4
m → $s5

FAZER !

```
main ( )
{
...
   x = 1;
   y = 2;
   z = soma ( x , y );
   k = soma (x , x );
  m = soma (y , y );
   x = soma ( k , m)
}

int soma (int r, int s)
{
   return ( r + s )
}
```

Mapeamento:
x → $s1
y → $s2
z → $s3
k → $s4
m → $s5

```
main:

addi $s1, $zero, 1
addi $s2, $zero, 2

add $a0, $zero, $s1
add $a1, $zero, $s2
jal soma
nop

add $s3, $zero, $v0
add $a0, $zero, $s1
add $a1, $zero, $s1
jal soma
nop

add $s4, $zero, $v0
add $a0, $zero, $s3
add $a1, $zero, $s4
jal soma
nop

add $s5, $zero, $v0
add $a0, $zero, $s4
add $a1, $zero, $s5
jal soma
nop

add $s1, $zero, $v0


soma:
add $v0, $a0, $a1
jr $ra
```



```
main ( )
{
  int i, j, k;
  i = mult ( j , k );
...
}

int mult (int mando, int mdor)
{




}
```

Exercício:

Fazer a função *mult*

Obs:

- só sabemos somar!

Mapeamento: i em $s0, j em $s1 e k em $s2

```
start:
Transfere argumentos e chama função
      add $a0,$s1,$0      # arg0 = j
      add $a1,$s2,$0      # arg1 = k
      jal mult            # call mult
```

```
main ( )
{
  int i, j, k;
  i = mult ( j , k );
...
}
```


Mapeamento: i em $s0, j em $s1 e k em $s2

```
start:
Transfere argumentos e chama função
      add $a0,$s1,$0      # arg0 = j
      add $a1,$s2,$0      # arg1 = k
      jal mult            # call mult
Salva valor de retorno na variável estática $s0
      add $s0,$v0,$0      # i = mult()
```

```
main ( )
{
  int i, j, k;
  i = mult ( j , k );
...
}
```

```
mult:
     add  $t0,$0,$0      # prod ($t0)=0

Loop:
     slt  $t1,$0,$a1     # mlr($a1) > 0?
     beq  $t1,$0,Fim     # não => vá para Fim
     add  $t0,$t0,$a0    # sim: prod += mc($a0)
     addi $a1,$a1,-1     # mlr -= 1
     j    Loop           # goto Loop

Fim:
     add  $v0,$t0,$0     # $v0 = prod
     jr   $ra            # retorna
```

```
int mult (int mc, int mlr)
{
  int prod;
  prod = 0;
  while ( mlr > 0 )
  {
    prod = produto + mc;
    mdor = mdor -1;
  }
return prod;
}
```


Funções Aninhadas (1/2)

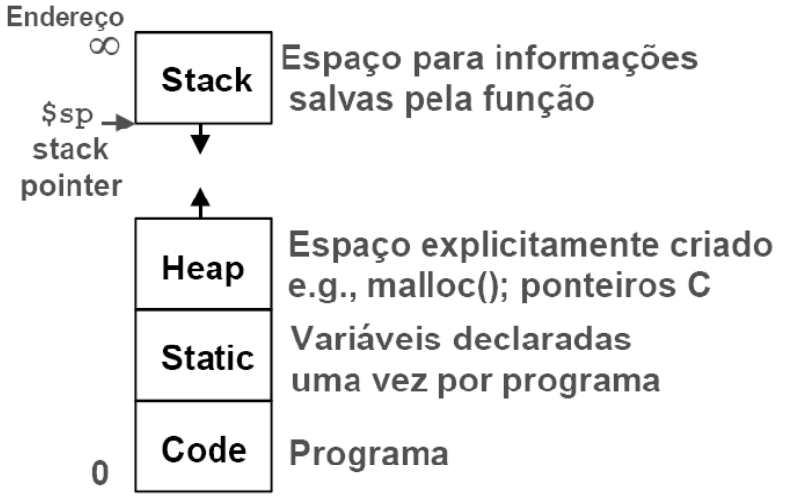
```
int sumSquare(int x, int y) {
     return mult(x,x)+ y;
  }
```

- Alguma coisa chamada **`sumSquare`**, que **chama `mult`**.
- Há um valor em **$ra** que `sumSquare` quer pular de volta, mas **será sobrescrito** pela chamada a `mult`.
- Precisa **salvar o endereço de retorno** de `sumSquare` antes de chamar `mult`.

Funções Aninhadas (2/2)

- Em geral, pode ser necessário **salvar** algum outro **registrador** além de $ra.
- Quando um programa C está rodando, existem 3 **importantes áreas de memória** que são alocadas:
  - **Static** (alocação estática): Variáveis declaradas uma vez por programa, deixam de existir somente quando a execução termina.
  - **Heap** (alocação dinâmica): Variáveis declaradas dinamicamente.
  - **Stack** (pilha): Espaço a ser utilizado pela função durante sua execução; é aqui que podemos **salvar** os valores dos **registradores**.

Alocação de Memória em C



Usando a Pilha (1/2)

- Nós temos um registrador **$sp** que **sempre aponta** para o **último espaço utilizado na pilha**.
- Para utilizar a pilha:
  - 1º **decrementamos** o ponteiro **$sp** de 4 bytes
  - 2º **preenchemos** os 4 bytes da pilha com a informação.
- Então, como compilamos isto?

```
int sumSquare(int x, int y) {
  return mult(x,x)+ y;}
```

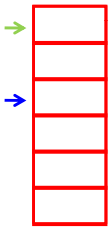

Usando a Pilha (2/2)

Função SumSquare

```
int sumSquare(int x, int y)
{
return mult(x,x)+ y;}
```

- Compile manualmente

```
sumSquare:
(salva end. retorno e argumento de SumSquare)
        addi    $sp,$sp,-8        # reserva espaço na pilha
```



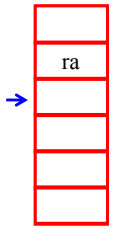
Usando a Pilha (2/2)

Função SumSquare

```
int sumSquare(int x, int y)
{
return mult(x,x)+ y;}
```

- Compile manualmente

```
sumSquare:
(salva end. retorno e argumento de SumSquare)
        addi    $sp,$sp,-8        # reserva espaço na pilha
        sw      $ra, 4($sp)       # salva reg. end. retorno
```



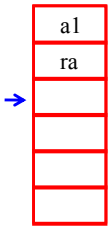
Usando a Pilha (2/2)

Função SumSquare

```
int sumSquare(int x, int y)
{
return mult(x,x)+ y;}
```

- Compile manualmente

```
sumSquare:
(salva end. retorno e argumento de SumSquare)
        addi    $sp,$sp,-8        # reserva espaço na pilha
        sw      $ra, 4($sp)       # salva reg. end. retorno
        sw      $a1, 8($sp)       # salvar argumento y
```



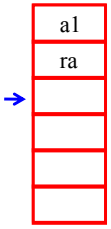
Usando a Pilha (2/2)

- Compile manualmente

Função SumSquare

```
int sumSquare(int x, int y)
{
return mult(x,x)+ y;}
```

```
sumSquare:
(salva end. retorno e argumento de SumSquare)
        addi    $sp,$sp,-8          # reserva espaço na pilha
        sw      $ra, 4($sp)         # salva reg. end. retorno
        sw      $a1, 8($sp)         # salvar argumento y
(Transfere arg. de mult e chama função mult)
        add     $a1,$a0,$zero       # transfere arg.x de mult(x,x)
        jal     mult                # chama mult
```



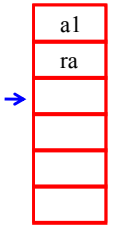
Usando a Pilha (2/2)

- Compile manualmente

Função SumSquare

```
int sumSquare(int x, int y)
{
return mult(x,x)+ y;}
```

```
sumSquare:
(salva end. retorno e argumento de SumSquare)
        addi    $sp,$sp,-8          # reserva espaço na pilha
        sw      $ra, 4($sp)         # salva reg. end. retorno
        sw      $a1, 8($sp)         # salvar argumento y
(Transfere arg. de mult e chama função mult)
        add     $a1,$a0,$zero       # transfere arg.x de mult(x,x)
        jal     mult                # chama mult
(restaura arg. De SumSquare e executa operação)
        lw      $a1, 8($sp)         # restaura arg. y
        add     $v0,$v0,$a1         # mult()+y
```



Usando a Pilha (2/2)

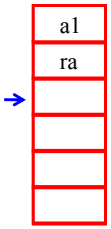
- Compile manualmente

Função SumSquare

```
int sumSquare(int x, int y)
{
return mult(x,x)+ y;}
```

```
sumSquare:
(salva end. retorno e argumento de SumSquare)
        addi    $sp,$sp,-8          # reserva espaço na pilha
        sw      $ra, 4($sp)         # salva reg. end. retorno
        sw      $a1, 8($sp)         # salvar argumento y
(Transfere arg. de mult e chama função mult)
        add     $a1,$a0,$zero       # transfere arg.x de mult(x,x)
        jal     mult                # chama mult
(restaura arg. De SumSquare e executa operação)
        lw      $a1, 8($sp)         # restaura arg. y
        add     $v0,$v0,$a1         # mult()+y

(restaura end. retorno de SumSquare e a pilha)
        lw      $ra, 4($sp)         # restaura end. retorno
        addi    $sp,$sp,8           # restaura pilha
        jr      $ra                 # retorna para prog. principal
```



Passos para fazer uma chamada de função

1) **Salvar** os valores necessários na pilha.
2) **Atribuir** argumento(s), se existir(em).
3) **Chamar** a função `jal`
4) **Restaurar** os valores da pilha.

Regras para Funções

- **Chamada** com uma instrução `jal` **retorna** com uma `jr $ra`
- Aceita **até 4 argumentos** em $a0, $a1, $a2 e $a3
- **Valor** de **retorno** sempre está **em $v0** (e se necessário em $v1)
- Deve seguir as convenções de registradores (mesmo em funções que somente você vai chamar)! Então, quais são elas?

Registradores MIPS

A constante 0	$0	$zero
Valores de Retorno	$2-$3	$v0-$v1
Argumentos	$4-$7	$a0-$a3
Temporários	$8-$15	$t0-$t7
Salvos	$16-$23	$s0-$s7
Mais Temporários	$24-$25	$t8-$t9
Stack Pointer	$29	$sp
Return Address	$31	$ra

- Em geral, você pode utilizar ou o nome ou o número. Os nomes deixam o código mais fácil de se ler.

Convenções de Registradores (1/5)

- Caller (*chamador*): a função que faz a chamada
- Callee (*função*): a função sendo chamada
- Quando a *função* retorna da execução, o *chamador* precisa saber quais **registradores podem ter mudado e quais não mudaram**.
- **Convenções** de Registradores: Um conjunto geralmente aceito de regras de quais registradores não mudam após uma chamada de função (`jal`) e quais podem ter sido mudados.

Convenções de Registradores (2/5)

- $0: Nunca muda. Sempre 0.
- $v0-$v1: Muda. Estes são esperados conter novos valores.
- $a0-$a3: Muda. Estes são registradores de argumentos voláteis.
- $t0-$t9: Muda. Por isso eles são chamados temporários: qualquer função pode mudá-los a qualquer momento.

Convenções de Registradores (3/5)

- $s0-$s7: Sem mudança. Muito importante, por isso eles são chamados registradores salvos. Se **a função chamada** (*calle*) muda estes registradores de algum modo, ela **deve restaurar os valores originais antes de retornar**.
- $sp: Sem mudança. O ponteiro da pilha deve apontar para o mesmo lugar antes e depois de uma chamada de jal ou então a função chamadora (*caller*) não será capaz de restaurar os valores da pilha. **A função chamada deve restaurar os valores originais antes de retornar**
- $ra: Muda. A chamada a `jal` vai mudar este registrador por si mesma.

Convenções de Registradores (4/5)

- O que estas convenções significam?
  - Se a função A chama B
    - **a função A deve salvar qualquer registrador temporário** que esteja utilizando na pilha antes de fazer uma chamada `jal`.
    - **A função B deve salvar qualquer registrador S (salvos - sp, $s0-$s7)** que ela pretende utilizar antes de modificar seus valores.
  - Lembre-se: *Caller/callee* precisam salvar somente os registradores temporários que eles estejam utilizando, não todos os registradores.

Convenções de Registradores (5/5)

- Note que, se ***callee*** vai utilizar algum registrador S, ela **deve**:
  - **Salvar** aqueles **registradores S** na pilha.
  - Utilizar os registradores
  - Restaurar os registradores S da pilha.
  - `jr $ra`
- Com os **registradores temporários**, a *calle* **não precisa salvar** na pilha.
- Portanto, a ***caller*** deve **salvar** aqueles **registradores temporários que quer preservar** através da chamada.

Exemplo: Compile isto (1/5)

```
main() {
 int i,j,k,m; /* i-m:$s0-$s3 */
 i = mult(j,k);...;
 m = mult(i,i);...
}
int mult (int mcand, int mlier){
 int product;
 product = 0;
 while (mlier > 0)  {
     product += mcand;
     mlier -= 1; }
 return product;
}
```

Exemplo: Compile isto (2/5)

Programa Principal

```
main() {
int i,j,k,m; /* i-m:$s0-$s3 */
i = mult(j,k);...;
m = mult(i,i);...}
```

```
start:
Transfere argumentos e chama função
      add $a0,$s1,$0      # arg0 = j
      add $a1,$s2,$0      # arg1 = k
      jal mult            # call mult
Salva valor de retorno na variável estática $s0
      add $s0,$v0,$0      # i = mult()
Transfere argumentos e chama função
      add $a0,$s0,$0      # arg0 = i
      add $a1,$s0,$0      # arg1 = i
      jal mult            # call mult
Salva valor de retorno na variável estática $s3
      add $s3,$v0,$0      # m = mult()
Done:
...
```


Exemplo: Compile isto (3/5)

- Notas:
  - **função `main`** (principal) termina com `done`, não com `jr $ra`, assim **não é necessário salvar $ra na pilha**.
  - todas as **variáveis** utilizadas **na função `main`** são **registradores salvos** (**estáticos**), assim não é necessário salvá-las na pilha.

Exemplo: Compile isto (4/5)

Função Multiplicação

```
mult:
      add  $t0,$0,$0      # prod ($t0)=0

Loop:
      slt  $t1,$0,$a1     # mlr($a1) > 0?
      beq  $t1,$0,Fim     # não => vá para Fim
      add  $t0,$t0,$a0    # sim: prod += mc($a0)
      addi $a1,$a1,-1     # mlr -= 1
      j    Loop           # goto Loop

Fim:
      add  $v0,$t0,$0     # $v0 = prod
      jr   $ra            # retorna
```

Mlr:multiplicador
Mc:multiplicando

Exemplo: Compile isto (5/5)

- Notas:
  - **Função não aninhada**: nenhuma chamada `jal` é feita de `mult`
    - não utilizamos qualquer registrador salvo, logo não precisamos salvar nada na pilha.
    - poderíamos ter utilizados **registradores S**, mas **teríamos que salvar** na pilha.
  - **registradores temporários** são utilizados para cálculos intermediários (**não precisam ser salvos**)
  - **$a1** é modificado diretamente (ao invés de copiá-lo em um registrador temporário) já que nós somos **livres para mudá-lo**.
  - o **resultado** é posto **em $v0** antes de retornar.

Exemplo: Compile isto (1/2)

- Fatorial **Função recursiva**

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

- Colocar na **pilha**, **todos** os **registradores** que **necessitam ser preservados**:
  - $a0-$a3, $t0-$t9, $ra, $s0, ajustar $sp.
  - No retorno: restaurar regs, restaurar Ssp.

Exemplo: Compile isto (2/2)

```
fact:
        addi    $sp, $sp,-8      # reserva espaço na pilha  (2 palavras)
        sw      $ra, 4($sp)      # salva end. retorno na pilha
        sw      $a0, 0($sp)      # salva argumento (n)
        slt     $t0, $a0, 1      # n<1?
        beq     $t0, $zero, L1   # não: (n>=1) vá para L1

        addi    $v0, $zero, 1    # n=0: $v0=1
        addi    $sp, $sp, 8      # restaura a pilha
        jr      $ra              # retorna da função
L1:     addi    $a0, $a0,-1      # n=n-1 =>$a0=$a0-1
        jal     fact             # chama fact(n-1)=> $v0=fact($a0)

        lw      $a0, 0($sp)      # restaura argumento n ($a0)
        lw      $ra, 4($sp)      # restaura end. retorno
        addi    $sp, $sp, 8      # restaura pilha

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

        jr      $ra              # retorna da função
```


Coisas para Lembrar (1/2)

- Funções são chamadas com `jal`, e retornam com `jr $ra`.
- A **pilha** é sua amiga: use-a para **salvar** qualquer coisa que precise. Apenas assegure-se de **deixá-la como a achou**.
- Convenções de Registradores: Cada registrador tem um propósito e limites no seu uso. Aprenda-os e siga-os.

Coisas para Lembrar (2/2)

- Instruções que nós conhecemos até agora:

  Aritmética: `add, addi, sub`

  Memória: `lw, sw`

  Decisão: `beq, bne, slt`

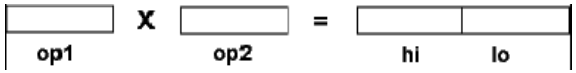
  Desvios incondicionais (pulos): `j, jal, jr`

- Registradores que nós conhecemos até agora:
  - $zero, $v0-$v1, $a0-$a3, $t0-$t7, $s0-$s7, $t8-$t9, $sp, $ra

Novas Instruções

Unidade de Multiplicação no MIPS

A unidade multiplicadora do Mips possui dois registradores de 32-bit denominados **hi** e **lo**. Não são registradores de propósito geral. Quando dois registradores com operandos de 32-bit são multiplicados, **hi** e **lo** possuem o resultado de 64 bits.



Bits 32 até 63 estão em **hi** e bits 0 até 31 em **lo**.

Novas Instruções

Uso da instrução

mult s,t # hilo $s * $t

mfhi e mflo

São as instruções utilizadas para mover os resultados da multiplicação para registradores de uso geral.

mfhi d # Mover de Hi para GPR d
mflo k # Mover de Lo para GPR k

Novas Instruções

Bits Significantes

Os bits significantes em um num. Positivo ou unsigned são todos os bits à direita do bit com valor 1 mais à esquerda do número:

0000 0000 0**100 0011 0101 0110 1101 1110**

O número possui 23 bits significantes.

Os bits significantes em um num. Negativo, são todos os bits à direita do bit com valor 0 mais à esquerda do número:

1111 1111 1**011 1100 1010 1001 0010 0010**

O número possui 23 bits significantes.

Novas Instruções

Bits Significantes

Para garantir que o produto não possuirá mais do que 32 bits, a soma dos bits significantes dos números deverá ser menor ou igual a 32.

Perg.: Aproximadamente quantos bits significantes você espera do produto:

0100 1010 × 0001 0101

Novas Instruções

Uso da instrução

```
mult s,t          # hilo  $s * $t
mfhi d        # Mover de Hi para GPR d
mflo k            # Mover de Lo para GPR k
```

Passar para MIPS usando mult a expressão:

y = 5x - 74

Novas Instruções

Uso da instrução div (equivalente a div e mod)

```
div s,t          # hilo  recebem resultado e resto
```

De maneira análoga à multiplicação:

```
mfhi d          # Move o resto de HI para GPR d
mflo k              # Move resultado de LO para GPR k
```

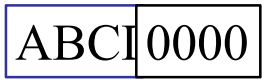

Novas Instruções

lui reg, valor # carrega parte mais alta de reg com valor

Exemplo:

lui $s1, 0xABCD

#s1



addi $s1, 0xABCD

#s1

Pseudo Instruções

Exemplos (move, li, la, lw e sw)

```
move $t1,$t0          #move de t0 para t1, traduzida por addu

li $t0, 0x12345678    # carrega o registrador com um imediato de até 32 bits,

li $t1, -12345678     # usa $at (assembler temporary), é  traduzida como lui e
                      # ori, pode-se usar um numero negativo
```

Pseudo Instruções

Exemplos (move, li, la, lw e sw)

```
la $t0, varx      # carrega o registrador com o endereço simbólico,
la $t1, vary      # usa os registradores at, e instr. lui e ori
                  # para determinarmos o endereço.

lw $t0, varx      # carrega o registrador com o valor do endereço simbólico,
sw $t1, vary      # usa os registradores at, e instr. lui e ori
                  # para determinarmos o endereço.
                  # ATENCÃO:  não confundir com o lw $reg,
imediato($reg)
```

```
.data
varx: .word 3
vary: .word 5
```


Serviços do Sistema Operacional (Handlers) do MIPS

Service	Code in $v0	Arguments	Returned Value
print integer	1	$a0 == integer	
print float	2	$f12 == float	
print double	3	($f12, $f13) == double	
print string	4	$a0 == address of string	
read integer	5		$v0 == integer
read float	6		$f0 == float
read double	7		($f0, $f1) == double
read string	8	$a0 == buffer address $a1 == buffer length	
allocate memory	9	$a0 == number of bytes	$v0 == address
exit	10		


```
li $v0,4            # código 4 == print string
la $a0,palavra      # $a0 == endereço da string
syscall             # Solicita o serviço ao SO

li $v0,5            # código 5 == read integer
syscall             # Solicita serviço ao SO
                    # Lê a linha em caracteres ascii
                    # Converte para um inteiro de 32-bit
                    # $v0 <-- recebe o inteiro

li $v0,1            # code 1 == print integer
lw $a0,numero       # $a0 = o que está no endereço numero
syscall             # Solicita serviço ao SO
                    # Converte inteiro de 32-bit em caracteres
                    # Mostra caracteres no monitor

li $v0,10           # código 10 == exit
syscall             # Retorna controle ao SO


.data
palavra: .asciiz "Alo MIPS!\n Enfim sei programar !!!\n"
numero:  .word 100
```



```
## soma.asm
## programa para adicionar dois inteiros armazenados na memória
##

la $t0,val2         # carrega um endereço de 32-bit em $t0
lw $t1,0($t0)       # carrega primeiro valor, 2
lw $t2,val3         # carrega Segundo valor, 3
sll $0,$0,0         # load delay slot
add $t3,$t1,$t2     # calcula a soma
sw $t3, val4        # escreve na memória

.data
val1: .word 1
val2: .word 2
val3: .word 3
val4: .word 4
texto1: . asciiz "Digite um numero\n"
```