# Organização e Arquitetura de Processadores

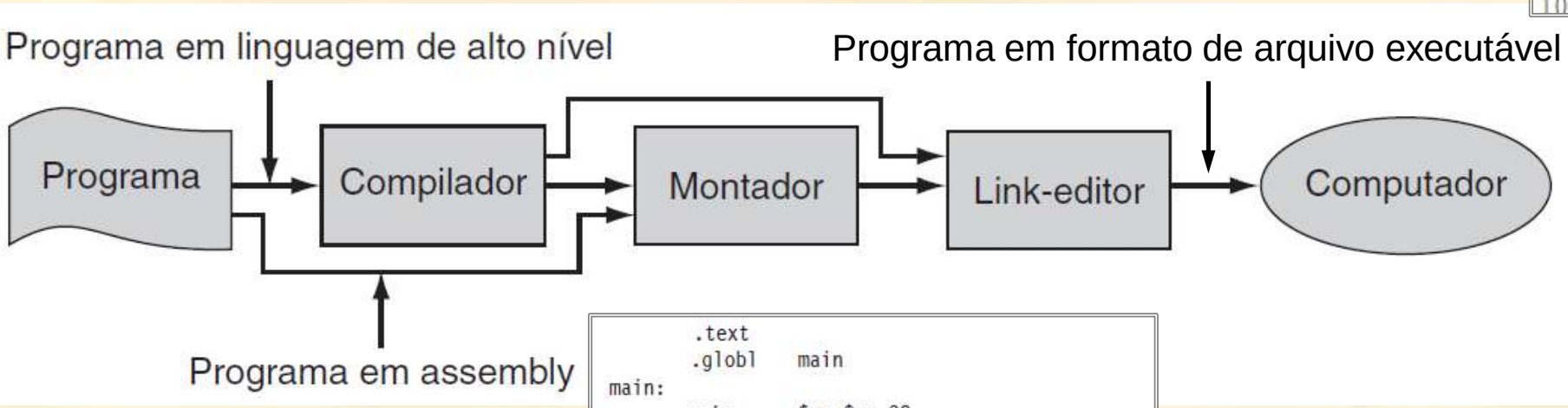
# Introdução à Arquitetura do MIPS

*Principais ferramentas:* ***MARS***
***Livro do Patterson e Hennessy***
***Apêndice A do livro do Patterson e Hennessy***

**Prof. Dr. César Marcon**

# Do Programa à Execução

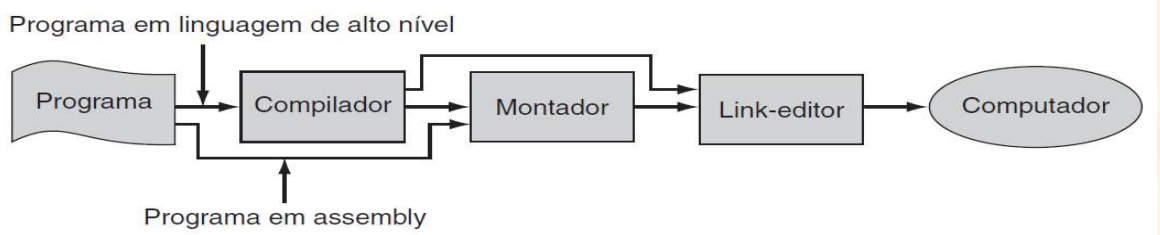
```
main (int argc,char *argv [ ])
{
     int i;
     int sum =0;
     for (i =0;i <=100;i =i +1)sum =sum +i *i;
     printf ("The sum from 0 ..100 is %d \n",sum);
}
```




```
        .text
        .globl   main
main:
        subu     $sp,$sp,32
        sw       $0,24($sp)
        sw       $0,28($sp)
loop:
        lw       $t6,28($sp)
        mul      $t7,$t6,$t6
        lw       $t8,24($sp)
        move     $v0,$0
        lw       $ra,20($sp)
        addu     $sp,$sp,32
        jr       $ra

        .data
        .align   0
str:
        .asciiz  "The sum from 0 ..100 is %d \n"
```

# O Montador - Edição




File Edit Run Settings Tools Help

Edit Execute

VetorPonteiro.asm

```
1   .text
2     .globl main
3   main:
4     la     $a0, String1      # | main() {
5     li     $v0, 4            # |
6     syscall                  # +-> printf("Digite val: ");
7     li     $v0, 5            # | $v0=5 lê inteiro e armazena em
8     syscall                  # | chamada ao SO
9     move   $t1, $v0          # +-> val = getchar();
10    la     $t6, vet          # + int *p = $t6 = &vet[0];
11    la     $t5, vet+40       # + int *pFim = $t5 = &vet[10];
12  loop:                      # |
13    bge    $t6, $t5, fim     # +-> while(p<&vet[10]); (p>=pFim)
14    sw     $t1, 0($t6)       # + *p = val;
15    add    $t6, $t6, 4       # + p++;
16    j      loop              # +-}
17  fim:                       # |
18    li     $v0, 10           # | $v0=10 encerra programa
19    syscall                  # +-> chamad ao SO }
20
21  .data
22  String1:   .asciiz "Digite val: "
23  .align 2 # necessário, caso dados não estejam alinhados
24  vet:       .space 40      # int vet[10]; Espaço para 10 words
25
```

Line: 14 Column: 38 Show Line Numbers

Mars Messages Run I/O

Clear

```
-- program is finished running --

Digite val: **** user input : 2

-- program is finished running --
```

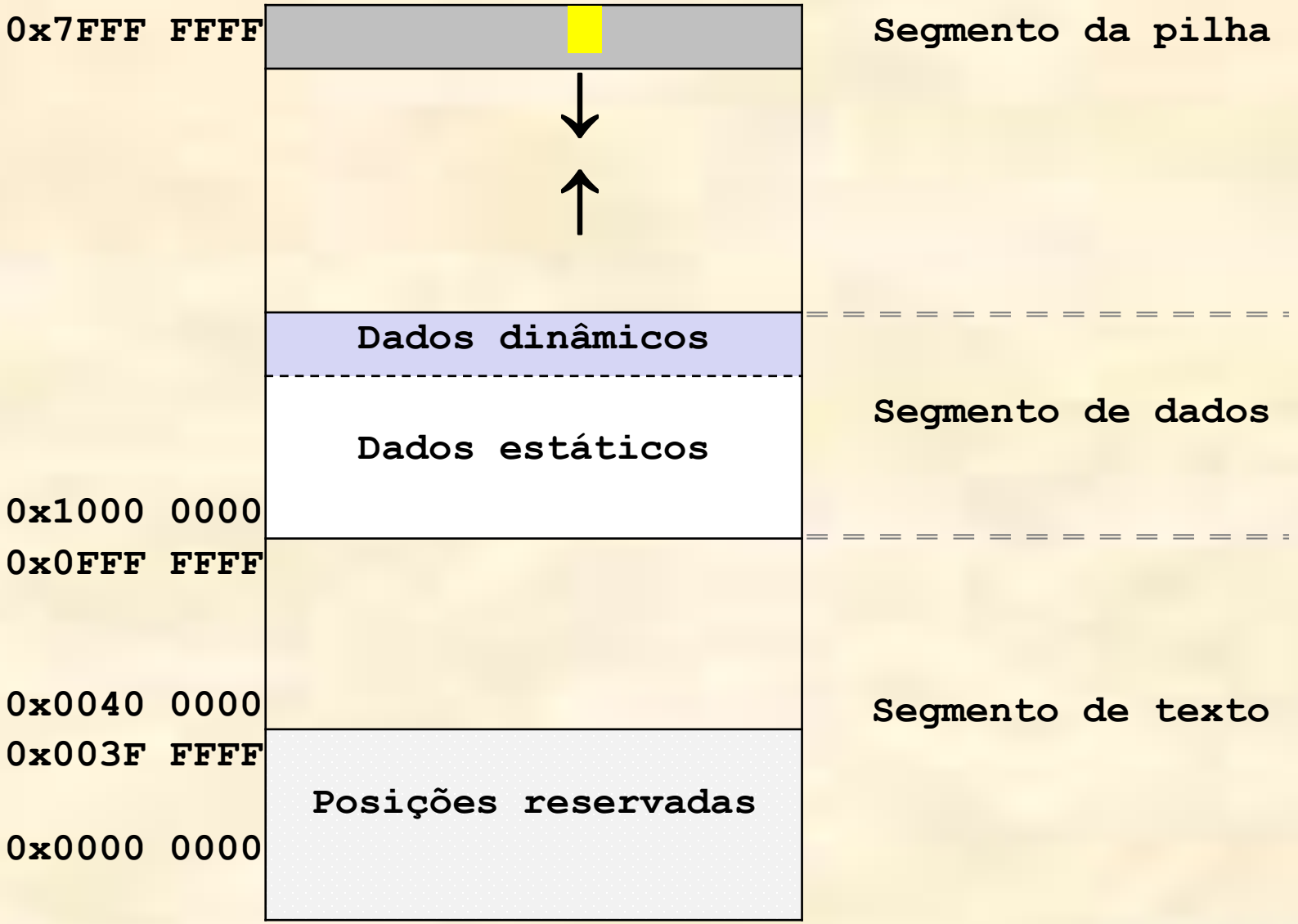
Coproc 0 Registers Coproc 1

| Name | Num... | Value |
|---|---|---|
| $zero | 0 | 0x00000000 |
| $at | 1 | 0x00000000 |
| $v0 | 2 | 0x0000000a |
| $v1 | 3 | 0x00000000 |
| $a0 | 4 | 0x10010000 |
| $a1 | 5 | 0x00000000 |
| $a2 | 6 | 0x00000000 |
| $a3 | 7 | 0x00000000 |
| $t0 | 8 | 0x00000000 |
| $t1 | 9 | 0x00000002 |
| $t2 | 10 | 0x00000000 |
| $t3 | 11 | 0x00000000 |
| $t4 | 12 | 0x00000000 |
| $t5 | 13 | 0x10010038 |
| $t6 | 14 | 0x10010038 |
| $t7 | 15 | 0x00000000 |
| $s0 | 16 | 0x00000000 |
| $s1 | 17 | 0x00000000 |
| $s2 | 18 | 0x00000000 |
| $s3 | 19 | 0x00000000 |
| $s4 | 20 | 0x00000000 |
| $s5 | 21 | 0x00000000 |
| $s6 | 22 | 0x00000000 |
| $s7 | 23 | 0x00000000 |
| $t8 | 24 | 0x00000000 |
| $t9 | 25 | 0x00000000 |
| $k0 | 26 | 0x00000000 |
| $k1 | 27 | 0x00000000 |
| $gp | 28 | 0x10008000 |
| $sp | 29 | 0x7fffeffc |
| $fp | 30 | 0x00000000 |
| $ra | 31 | 0x00000000 |
| pc | | 0x00400048 |
| hi | | 0x00000000 |
| lo | | 0x00000000 |

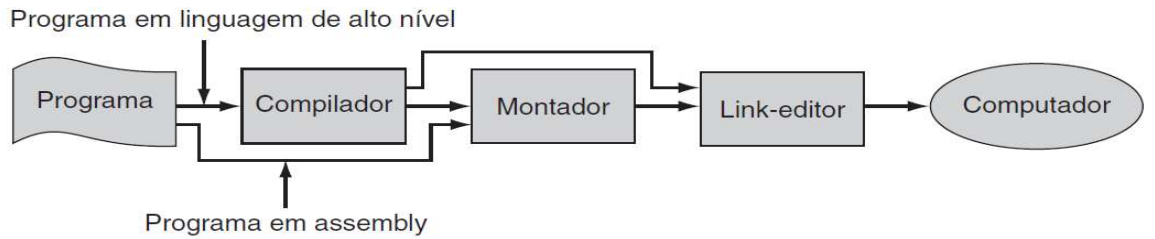
# Layout de Memória do MIPS (Convenção de software)

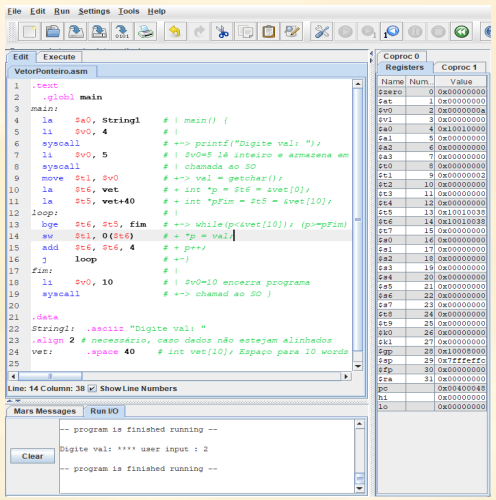
```
        .text
        .globl  main
main:
        subu    $sp,$sp,32
        sw      $0,24($sp)
        sw      $0,28($sp)
loop:
        lw      $t6,28($sp)
        mul     $t7,$t6,$t6
        lw      $t8,24($sp)
        move    $v0,$0
        lw      $ra,20($sp)
        addu    $sp,$sp,32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "The sum from 0 ..100 is %d \n"
```



# O Montador - Execução



File Edit Run Settings Tools Help

Edit | Execute

**Text Segment**

Program Arguments:

| Bkpt | Address | Code | Basic | | | | |
|---|---|---|---|---|---|---|---|
| | 0x00400000 | 0x3c011001 | lui $1,0x00001001 | 4: | la | $a0, String1 | # \| main() { |
| | 0x00400004 | 0x34240000 | ori $4,$1,0x00000000 | | | | |
| | 0x00400008 | 0x24020004 | addiu $2,$0,0x00000004 | 5: | li | $v0, 4 | # \| |
| | 0x0040000c | 0x0000000c | syscall | 6: | syscall | | # +-> printf("D |
| | 0x00400010 | 0x24020005 | addiu $2,$0,0x00000005 | 7: | li | $v0, 5 | # \| $v0=5 lê in |
| | 0x00400014 | 0x0000000c | syscall | 8: | syscall | | # \| chamada ao |
| | 0x00400018 | 0x00024821 | addu $9,$0,$2 | 9: | move | $t1, $v0 | # +-> val = get |
| | 0x0040001c | 0x3c011001 | lui $1,0x00001001 | 10: | la | $t6, vet | # + int *p = $t |
| | 0x00400020 | 0x342e0010 | ori $14,$1,0x00000010 | | | | |
| | 0x00400024 | 0x3c011001 | lui $1,0x00001001 | 11: | la | $t5, vet+40 | # + int *pFim = |
| | 0x00400028 | 0x342d0038 | ori $13,$1,0x00000038 | | | | |
| | 0x0040002c | 0x01cd082a | slt $1,$14,$13 | 13: | bge | $t6, $t5, fim | # +-> while(p<& |
| | 0x00400030 | 0x10200003 | beq $1,$0,0x00000003 | | | | |
| | 0x00400034 | 0xadc90000 | sw $9,0x00000000($14) | 14: | sw | $t1, 0($t6) | # + *p = val; |
| | 0x00400038 | 0x21ce0004 | addi $14,$14,0x0000... | 15: | add | $t6, $t6, 4 | # + p++; |

**Data Segment**

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) | Value (+18) |
|---|---|---|---|---|---|---|---|
| 0x10010000 | 0x69676944 | 0x76206574 | 0x203a6c61 | 0x00000000 | 0x00000007 | 0x00000007 | 0x00000007 |
| 0x10010020 | 0x00000007 | 0x00000007 | 0x00000007 | 0x00000007 | 0x00000007 | 0x00000007 | 0x00000000 |
| 0x10010040 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010060 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010080 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100a0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100c0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100e0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010100 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010120 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010140 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010160 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |

Mars Messages | Run I/O

Clear

```
-- program is finished running --

Digite val: **** user input : 7

-- program is finished running --
```

Coproc 1 | Coproc 0

**Registers**

| Name | Number | Value |
|---|---|---|
| $zero | 0 | 0x00000000 |
| $at | 1 | 0x00000000 |
| $v0 | 2 | 0x0000000a |
| $v1 | 3 | 0x00000000 |
| $a0 | 4 | 0x10010000 |
| $a1 | 5 | 0x00000000 |
| $a2 | 6 | 0x00000000 |
| $a3 | 7 | 0x00000000 |
| $t0 | 8 | 0x00000000 |
| $t1 | 9 | 0x00000007 |
| $t2 | 10 | 0x00000000 |
| $t3 | 11 | 0x00000000 |
| $t4 | 12 | 0x00000000 |
| $t5 | 13 | 0x10010038 |
| $t6 | 14 | 0x10010038 |
| $t7 | 15 | 0x00000000 |
| $s0 | 16 | 0x00000000 |
| $s1 | 17 | 0x00000000 |
| $s2 | 18 | 0x00000000 |
| $s3 | 19 | 0x00000000 |
| $s4 | 20 | 0x00000000 |
| $s5 | 21 | 0x00000000 |
| $s6 | 22 | 0x00000000 |
| $s7 | 23 | 0x00000000 |
| $t8 | 24 | 0x00000000 |
| $t9 | 25 | 0x00000000 |
| $k0 | 26 | 0x00000000 |
| $k1 | 27 | 0x00000000 |
| $gp | 28 | 0x10008000 |
| $sp | 29 | 0x7fffeffc |
| $fp | 30 | 0x00000000 |
| $ra | 31 | 0x00000000 |
| pc | | 0x00400048 |
| hi | | 0x00000000 |
| lo | | 0x00000000 |

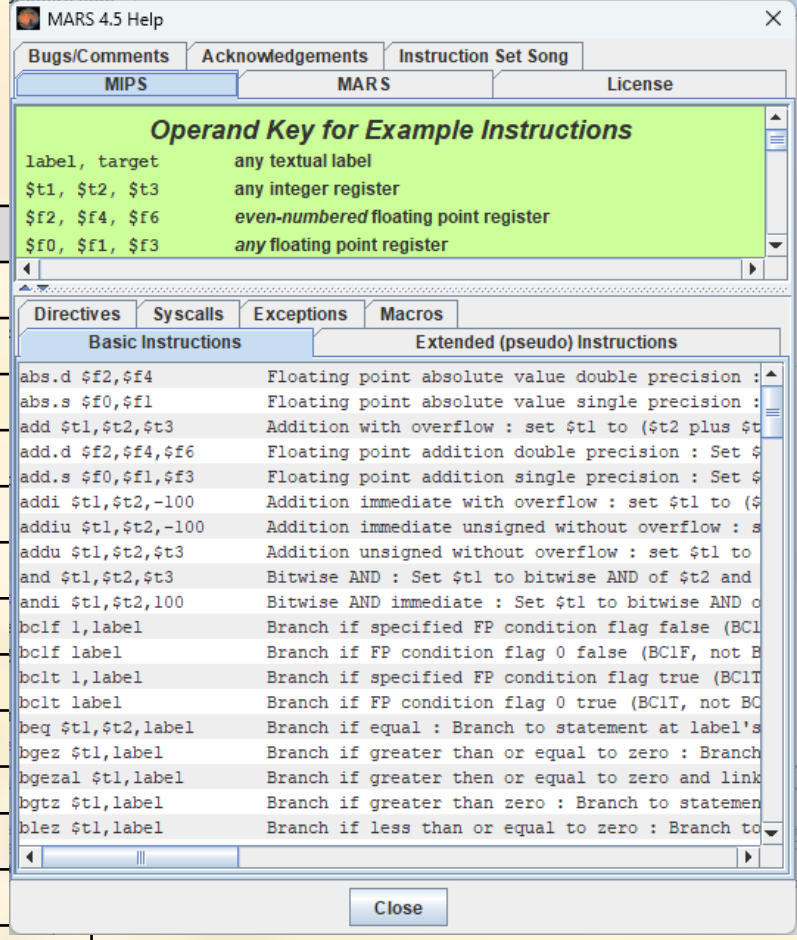
# Banco de Registradores

- **O MIPS tem um banco de 32 registradores de uso geral que podem ser endereçados em cada um dos campos de 5 bits da instrução (Patterson Apêndice A-17)**
  - Registradores são representados por um $ seguido do seu nome

| Número | Nome | Significado |
|---|---|---|
| 0 | $zero | constante 0 |
| 1 | $at | reservado para o montador |
| 2, 3 | $v0, $v1 | resultado de função |
| 4 - 7 | $a0 - $a3 | argumento para função |
| 8 - 15 | $t0 - $t7 | temporário |
| 16 - 23 | $s0 - $s7 | temporário (salvo nas chamadas de função) |
| 24, 25 | $t8, $t9 | temporário |
| 26, 27 | $k0, $k1 | reservado para o SO |
| 28 | $gp | apontador de área global (global pointer) |
| 29 | $sp | stack pointer |
| 30 | $fp | frame pointer |
| 31 | $ra | registrador de endereço de retorno |

# Conjunto de Instruções

| Instrução | Operação |
|---|---|
| LA rDest, endereço | rDest ← endereço |
| LI rDest, imm | rDest ← imm |
| LB rt, endereço (byte) e LW rt, endereço (word) | rt ← M[endereço] |
| SB rt, endereço (byte) e SW rt, endereço (word) | M[endereço] ← rt |
| MOVE rDest, Rsrc | rDest ← Rsrc |
| ADD rd, rs, rt | rd ← rs + rt |
| ADDI rd, rs, CTE | rd ← rs + CTE (CTE é uma constante inteira) |
| OR rd, rs, rt | rd ← rs OR rt |
| SLT rd, rs, rt | Se rs menor que rt rd ← 1, senão rd ← 0 |
| J endereço | $pc ← endereço (endereço de 26 bits+2 desloc. = $2^{28}$) |
| JAL endereço (chamada de função) | $ra ← $pc + 4 e $pc ← endereço |
| J $ra (retorno de função) | $pc ← $ra (Salto a registrador = $2^{32}$) |
| BEQ rs, rt, label (label de 16 bits, salto de +/- $2^{15}$) | Se igual então $pc ← label, senão $pc ← $pc + 4 |
| BGT rs, rt, label | Se maior então $pc ← label, senão $pc ← $pc + 4 |
| BGEZ rs, label | Se maior ou igual a zero $pc ← label, senão $pc ← $pc + 4 |
| BGTZ rs, label | Se maior que zero $pc ← label, senão $pc ← $pc + 4 |




```
MOVE    $a1, $v0
SUB     $t1, $t2, $t3
LA      $a0, $TxtFact
LI      $v0, 4
```

# Exercício

- **Mostre qual a instrução do MIPS ou conjunto de instruções do MIPS necessário para implementar os comandos em C que seguem**

```
// Supor a e b nos registradores $t0 e $t1
      a = b + 100;
```

```
// Supor a e b valor inteiros em memória
      a = b + 100;
```

| Instrução |
|---|
| LA rDest, endereço |
| LI rDest, imm |
| LB rt, endereço (byte) e LW rt, endereço (word) |
| SB rt, endereço (byte) e SW rt, endereço (word) |
| MOVE rDest, Rsrc |
| ADD rd, rs, rt |
| ADDI rd, rs, CTE |
| OR rd, rs, rt |
| SLT rd, rs, rt |
| J endereço |
| JAL endereço (chamada de função) |
| J $ra (retorno de função) |
| BEQ rs, rt, label (label de 16 bits, salto de +/- $2^{15}$) |
| BGT rs, rt, label |
| BGEZ rs, label |
| BGTZ rs, label |

# Resposta

- **Mostre qual a instrução do MIPS ou o menor conjunto de instruções do MIPS necessário para implementar os comandos em C que seguem**

```
// Supor a e b nos registradores $t0 e $t1
      a = b + 100;

ADDI  $t0, $t1, 100      # $t0 ← $t1 + 100
```

```
// Supor a e b valor inteiros em memória
      a = b + 100;
```

| Instrução |
|---|
| LA rDest, endereço |
| LI rDest, imm |
| LB rt, endereço (byte) e LW rt, endereço (word) |
| SB rt, endereço (byte) e SW rt, endereço (word) |
| MOVE rDest, Rsrc |
| ADD rd, rs, rt |
| ADDI rd, rs, CTE |
| OR rd, rs, rt |
| SLT rd, rs, rt |
| J endereço |
| JAL endereço (chamada de função) |
| J $ra (retorno de função) |
| BEQ rs, rt, label (label de 16 bits, salto de +/- $2^{15}$) |
| BGT rs, rt, label |
| BGEZ rs, label |
| BGTZ rs, label |

# Resposta

- **Mostre qual a instrução do MIPS ou o menor conjunto de instruções do MIPS necessário para implementar os comandos em C que seguem**

```
// Supor a e b nos registradores $t0 e $t1
      a = b + 100;

ADDI  $t0, $t1, 100      # $t0 ← $t1 + 100
```

```
// Supor a e b valor inteiros em memória
      a = b + 100;

LW    $t2, b             # $t2 ← b
ADDI  $t2, $t2, 100      # $t2 ← $t2 + 100
SW    $t2, a             # a   ← $t2
```

| Instrução |
|---|
| LA rDest, endereço |
| LI rDest, imm |
| LB rt, endereço (byte) e LW rt, endereço (word) |
| SB rt, endereço (byte) e SW rt, endereço (word) |
| MOVE rDest, Rsrc |
| ADD rd, rs, rt |
| ADDI rd, rs, CTE |
| OR rd, rs, rt |
| SLT rd, rs, rt |
| J endereço |
| JAL endereço (chamada de função) |
| J $ra (retorno de função) |
| BEQ rs, rt, label (label de 16 bits, salto de +/- $2^{15}$) |
| BGT rs, rt, label |
| BGEZ rs, label |
| BGTZ rs, label |

# Exercício

- **Coloque comentários e faça um pseudocódigo do trecho assembly abaixo, presumindo que $a0 é usado para entrada e $v0 é usado para saída**

```
begin:
        ADDI   $t0, $zero, 0
        ADDI   $t1, $zero, 0
loop:
        SLT    $t2, $a0, $t1
        BNE    $t2, $zero, finish
        ADD    $t0, $t0, $t1
        ADDI   $t1, $t1, 2
        J      loop
finish:
        ADD    $v0, $t0, $zero
```

| Instrução | Operação |
|---|---|
| ADD rd, rs, rt | rd ← rs + rt |
| ADDI rd, rs, CTE | rd ← rs + CTE (CTE é uma constante inteira) |
| OR rd, rs, rt | rd ← rs OR rt |
| SLT rd, rs, rt | Se rs menor que rt, então rd ← 1, senão rd ← 0 |
| J endereço | $pc ← endereço |
| BEQ rs, rt, label | Se igual então $pc ← label, senão $pc ← $pc + 4 |
| BNE rs, rt, label | Se diferente então $pc ← label, senão $pc ← $pc + 4 |

# Resposta

- **Coloque comentários e faça um pseudocódigo do trecho assembly abaixo, presumindo que $a0 é usado para entrada e $v0 é usado para saída**

```
begin:
      ADDI   $t0, $zero, 0        # t0 ← 0
      ADDI   $t1, $zero, 0        # t1 ← 0
loop:
      SLT    $t2, $a0, $t1        # Se(a0 < t1) t2 ← 1
                                  # Senão t2 ← 0
      BNE    $t2, $zero, finish   # Se(t2 != 0) PC ← finish
                                  # Senão PC ← PC + 4
      ADD    $t0, $t0, $t1        # t0 ← t0 + t1
      ADDI   $t1, $t1, 2          # t1 ← t1 + 2
      J      loop                 # PC ← loop
finish:
      ADD    $v0, $t0, $zero      # v0 ← t0 + 0
```

```
t0 = 0;
t1 = 0;
while(t1 >= entrada) {


  t0 = t0 + t1;
  t1 = t1 + 2;
}

saída = t0;
```

# Estrutura de Um Programa Assembly

```
int a = 4
int b = 7;
int maior;

main()
{
     if(a > b)
          maior = a;
     else
          maior = b;
};
```

```
.data
a:  .word 4
b:  .word 7
maior:  .space 4
.text
        .globl main
main:                                   # main() {
        lw      $t0, a                  # t0 ← a
        lw      $t1, b                  # t1 ← b
        bgt     $t0, $t1, a_Maior       # if(a > b)
b_Maior:
        sw      $t1, maior              #      maior = b;
        j       fim
a_Maior:
        sw      $t0, maior              #      maior = a;
fim:                                    # }
        j       fim
```

*Diretivas do montador*

*Rótulos (variáveis, funç..*

*Instruções do assembly*

*Comentários a partir de #*

# Chamada ao Sistema Operacional

- **Operações de entrada e saída de dados com recursos do sistema operacional!**
  - Comunicação entre programa e SO através de registradores!

```
.text
.globl main
                    ...
.main:
        MOVE    $a1, $v0
                    ...
        LA      $a0, $TxtFact
        LI      $v0, 4
        SYSCALL
                    ...

.data
$TxtFact:
        .ascii “O fatorial de 10 é:”
                    ...
```

| Serviço | Código (registrador $v0) | Argumentos | Resultado |
|---|---|---|---|
| print_int | 1 | $a0 = valor inteiro | |
| print_float | 2 | $f12 = valor em ponto flutuante | |
| print_double | 3 | $f12 = valor em double | |
| print_string | 4 | $a0 = endereço da string | |
| read_int | 5 | | $v0 (valor inteiro) |
| read_float | 6 | | $f0 (valor float) |
| read_double | 7 | | $f0 (valor double) |
| read_string | 8 | $a0 = endereço da string<br>$a1 = tamanho da string | |
| exit | 10 | | |

# Estrutura de Um Programa Assembly – Serviço de Exit

```
int a = 4
int b = 7;
int maior;

main()
{
     if(a > b)
          maior = a;
     else
          maior = b;
}
```

```
.data
a:  .word 4
b:  .word 7
maior:  .space 4
.text
        .globl main
main:                                   # main() {
        lw      $t0, a                  # t0 ← a
        lw      $t1, b                  # t1 ← b
        bgt     $t0, $t1, a_Maior       # if(a > b)
b_Maior:
        sw      $t1, maior              #      maior = b;
        j       fim
a_Maior:
        sw      $t0, maior              #      maior = a;
fim:    li      $v0, 10
        syscall                          # }
```

# Exercício

- **Com os serviços do SO, faça um pseudocódigo que descreve o trecho de programa abaixo**

```
        LI      $v0, 5
        SYSCALL
        MOVE    $t0, $v0
        LI      $v0, 5
        SYSCALL
        MOVE    $t1, $v0
        SUB     $t0, $t0, $t1
        BGT     $t0, $t1, Label1
        LI      $v0, 1
        MOVE    $a0, $t0
        SYSCALL
        J       Label2
Label1:
        MOVE    $a0, $t1
        LI      $v0, 1
        SYSCALL
Label2:
```

| Serviço | Código (registrador $v0) | Argumentos | Resultado |
|---|---|---|---|
| print_int | 1 | $a0 = valor inteiro | |
| print_float | 2 | $f12 = valor em ponto flutuante | |
| read_int | 5 | | $v0 (valor inteiro) |
| read_float | 6 | | $f0 (valor float) |
| read_double | 7 | | $f0 (valor double) |
| exit | 10 | | |

| Instrução | Operação |
|---|---|
| ADD rd, rs, rt | rd ← rs + rt |
| ADDI rd, rs, CTE | rd ← rs + CTE (CTE é uma constante inteira) |
| LI rDest, imm | rDest ← imm |
| MOVE rDest, Rsrc | rDest ← Rsrc |
| SUB rd, rs, rt | rd ← rs - rt |
| J endereço | $pc ← endereço |
| BEQ rs, rt, label | Se rs igual a rt, então $pc ← label, senão $pc ← $pc + 4 |
| BGT rs, rt, label | Se rs maior que rt, então $pc ← label, senão $pc ← $pc + 4 |

# Resposta

- Com os serviços do SO, faça um pseudocódigo que descreve o trecho de programa abaixo

```
        LI      $v0, 5
        SYSCALL
        MOVE    $t0, $v0
        LI      $v0, 5
        SYSCALL
        MOVE    $t1, $v0
        SUB     $t0, $t0, $t1
        BGT     $t0, $t1, Label1
        LI      $v0, 1
        MOVE    $a0, $t1
        SYSCALL
        J       Label2
Label1:
        MOVE    $a0, $t0
        LI      $v0, 1
        SYSCALL
Label2:
```

```
        LE(A)
        LE(B)
        A ← A - B
        SE(A > B)
                ESCREVE(A)
        SENÃO
                ESCREVE(B)
```

# Exercício

- **Descreva um código assembly completo do MIPS que implementa o programa em C**

```
int i=4;
int j;

main() {
  j = getchar();
  if(i==j)
    i = i + 2;
  else
    j = j - 1;
  printf("%d", i);
  printf("%d", j);
}
```

| Serviço | Código (registrador $v0) | Argumentos | Resultado |
|---|---|---|---|
| print_int | 1 | $a0 = valor inteiro | |
| print_float | 2 | $f12 = valor em ponto flutuante | |
| print_double | 3 | $f12 = valor em double | |
| print_string | 4 | $a0 = endereço da string | |
| read_int | 5 | | $v0 (valor inteiro) |
| exit | 10 | | |

| Instrução | Operação |
|---|---|
| ADD rd, rs, rt | rd ← rs + rt |
| ADDI rd, rs, CTE | rd ← rs + CTE (CTE é uma constante inteira) |
| LI rDest, imm | rDest ← imm |
| MOVE rDest, Rsrc | rDest ← Rsrc |
| SUB rd, rs, rt | rd ← rs – rt |
| SUBI rd, rs, CTE | rd ← rs - CTE (CTE é uma constante inteira) |
| J endereço | $pc ← endereço |
| BEQ rs, rt, label | Se rs igual a rt, então $pc ← label, senão $pc ← $pc + 4 |
| BGT rs, rt, label | Se rs maior que rt, então $pc ← label, senão $pc ← $pc + 4 |

# Resposta

- **Descreva um código assembly completo do MIPS que implementa o programa em C**

```
int i=4;
int j;

main() {
  j = getchar();
  if(i==j)
    i = i + 2;
  else
    j = j - 1;
  printf("%d", i);
  printf("%d", j);
}
```

```
.data
i: .word 4
j: .space 4

.text
        .globl main
main:                                   # main() {
        li      $v0, 5                  # |
        syscall                         # |
        move    $t1, $v0                # +-> j = getchar();
        lw      $t0, i                  # t0 ← i
        beq     $t0, $t1, eh_Igual      # if(a == b)
eh_Diferente:
        subi    $t1, $t1, 1             #     j = j - 1;
        j       fim                     # else
eh_Igual:
        addi    $t0, $t0, 2             #     i = i + 2;
fim:
        li      $v0, 1                  # |
        move    $a0, $t0                # |
        syscall                         # -> printf("%d", i);
        move    $a0, $t1                # |
        syscall                         # -> printf("%d", j);
        li      $v0, 10
        syscall                         # }
```

# Exercícios Básicos

***Patterson e Hennessy***

# Exercício

- **Descreva um conjunto de instruções do MIPS necessário para implementar o trecho de código C**

```
int x[30];
int C = 8;

x[10] = x[11] + C;
```

# Resposta

- **Descreva um conjunto de instruções do MIPS necessário para implementar o trecho de código C**

```
int x[30];
int C = 8;

x[10] = x[11] + C;
```

```
LA    $t0, $x              # t0 ← &x
LW    $t1, 44($t0)         # t1 ← x[11]
LW    $t2, $C              # t2 ← C
ADD   $t1, $t1, $t2        # t1 ← x[11] + C
SW    $t1, 40($t0)         # x[10] ← x[11] + C
```

# Exercício

- **Corrija o trecho de código abaixo para permitir copiar palavras do endereço armazenado no registrador $a0 para o endereço armazenado em $a1, escrevendo o número de palavras copiadas em $v0. O programa para de copiar quando encontrar uma palavra igual a 0. A palavra final deve ser copiada, mas não deve ser contada**

```
loop:
      LW    $v1, 0($a0)
      ADDI  $v0, $v0, 1
      SW    $v1, 0($a1)
      ADDI  $a0, $a0, 1
      ADDI  $a1, $a1, 1
      BNE   $v1, $zero, loop
```

# Resposta

- **Corrija o trecho de código abaixo para permitir copiar palavras do endereço armazenado no registrador $a0 para o endereço armazenado em $a1, escrevendo o número de palavras copiadas em $v0. O programa para de copiar quando encontrar uma palavra igual a 0. A palavra final deve ser copiada, mas não deve ser contada**

```
loop:
      LW    $v1, 0($a0)
      ADDI  $v0, $v0, 1
      SW    $v1, 0($a1)
      ADDI  $a0, $a0, 1
      ADDI  $a1, $a1, 1
      BNE   $v1, $zero, loop
```

```
      LI    $v0, 0
loop: LW    $v1, 0($a0)
      SW    $v1, 0($a1)
      BEQ   $v1, $zero, fim
      ADDI  $v0, $v0, 1
      ADDI  $a0, $a0, 4
      ADDI  $a1, $a1, 4
      J     loop
fim:
```

# Exercício

- **Implemente a instrução BCP que copia $t3 words da memória do endereço apontado por $t1 para o endereço apontado por $t2**

```
BCP:
```

# Resposta

- **Implemente a instrução BCP que copia $t3 words da memória do endereço apontado por $t1 para o endereço apontado por $t2**

```
BCP:
      blez  $t3, fim
      lw    $t4, 0($t1)
      sw    $t4, 0($t2)
      subiu $t3, $t3, 1
      addiu $t1, $t1, 4
      addiu $t2, $t2, 4
      j     BCP
fim:
```

# Exercício

- Apresente um algoritmo que descreva o trecho de código abaixo

```
main() {
   int i = 0;
   int sum = 0;
   do {
      sum = sum + i * i;
      i = i + 1;
   } (i <= 100);
   printf("\nSum 0..100=%d", sum);
}
```

# Resposta

- Apresente um algoritmo que descreva o trecho de código abaixo

```
main() {
   int i = 0;
   int sum = 0;
   do {
      sum = sum + i * i;
      i = i + 1;
   } (i <= 100);
   printf("\nSum 0..100=%d", sum);
}
```

```
.text
.globl main
main:
   move  $t0, $zero        # int i=0;
   move  $t1, $zero        # int sum = 0;
loop:
   mul   $t2, $t0, $t0     # i * i
   addu  $t1, $t1, $t2     # sum = sum + i * i;
   addiu $t0, $t0, 1       # i=i+1
   ble   $t0, 100, loop    # i <= 100
   la    $a0, str          # $a0 ← endereço de str
   li    $v0, 4            # printf... str
syscall                    # Chamada ao SO
   li    $v0, 1            # printf... sum
   move  $a0, $t1
syscall
   j     $ra

.data
str: .asciiz "\nSum 0..100="    # Caracteres ASCII
```

# Exercício

- Escreva um programa que leia continuamente números inteiros e os some cumulativamente. Ao ler uma entrada em 0, o programa para e imprime a soma

```
main() {
   int valor, acc=0;

   do {
      valor = getchar();
      acc = acc + valor;
   } while(valor!=0);
   printf("%d", acc);
}
```

# Resposta

- Escreva um programa que leia continuamente números inteiros e os some cumulativamente. Ao ler uma entrada em 0, o programa para e imprime a soma

```
main() {
   int valor, acc=0;

  do {
     valor = getchar();
     acc = acc + valor;
  } while(valor!=0);
  printf("%d", acc);
}
```

```
  .text
   .globl main
main:
   li    $t0, 0                    # acc = 0
inicio:
   li    $v0, 5
   syscall                         # valor = getchar();
   beq   $v0, $zero, fim
   addu  $t0, $t0, $v0             # acc = acc + valor
   j inicio
fim:
   move  $a0, $t0
   li    $v0, 1
   syscall                         # printf("%d", acc);
   j     $ra
```

# Exercício

- Converter a função Potencia descrita no assembly do MIPS para uma linguagem mais abstrata. Considere $a0 e $a1 como os parâmetros valor e exp da função e que a mesma retorne um inteiro armazenado na variável total

```
Potencia:
  move  $t0, $a0
  move  $t1, $a1
  li    $t2, 1
inicio:
  blez  $t1, fim
  mul   $t2, $t2, $t0
  subu  $t1, $t1, 1
  j     inicio
fim:
  move  $v0, $t2
  jr    $ra
```

# Resposta

- Converter a função Potencia descrita no assembly do MIPS para uma linguagem mais abstrata. Considere $a0 e $a1 como os parâmetros valor e exp da função e que a mesma retorne um inteiro armazenado na variável total

```
Potencia:
  move   $t0, $a0
  move   $t1, $a1
  li     $t2, 1
inicio:
  blez   $t1, fim
  mul    $t2, $t2, $t0
  subu   $t1, $t1, 1
  j      inicio
fim:
  move   $v0, $t2
  jr     $ra
```

```
int Potencia(int valor, int exp)
{
   int total=1;

   while(exp > 0)
   {
      total = total * valor;
      exp--;
   }
   return total;
}
```