

## EP 07

# Modelo de programação no MIPS

### Modelo de memória

Sistemas de computadores modernos quase sempre usam memória cache e memória virtual. Na nossa visão abstrata de memória não os inclui. O propósito de memória virtual é fazer parecer como se um programa tivesse todo o espaço de endereço disponível. Assim, nosso modelo de programação tem o espaço de endereço cheio.

O propósito da cache é, de forma transparente, acelerar o acesso à memória. Assim nosso modelo de programação não incluirá cache. A Memória no nosso modelo de programação é como segue:

A d r e s s e s	0xFFFFFFFF	1000 0000
		.....
		.....
	0x00000008	0100 1001
	0x00000007	1100 1100
	0x00000006	0110 1110
	0x00000005	0110 1110
	0x00000004	0000 0000
	0x00000003	0110 1011
	0x00000002	0101 0001
	0x00000001	1100 1001
	0x00000000	0100 1111
Main Memory		

### DADOS:

A Memória do MIPS é um vetor de  $2^{32}$  bytes. Cada byte tem um 32 bit de endereço. Cada byte com 8 bits, e 256 possíveis padrões. Os endereços de MIPS de memória principal estão na faixa de 0x00000000 até 0xFFFFFFFF.

Porém, os programas e dados do usuário estão restritos aos primeiros  $2^{31}$  bytes. A última metade do espaço de endereço é usado pelo sistema operacional e para propósitos especializados.

### OPERAÇÕES:

O processador contém registradores que são componentes eletrônicos capazes de armazenar padrões de bits.

O processador interage com memória movendo bits entre a memória e seus registradores.

**Load:** um padrão de bits em um endereço da memória é carregado para um registrador.

**Store:** um padrão de bits em um registrador é carregado para um determinado endereço da memória.

Os padrões de bits são copiados entre a memória e o processador em grupos de um, dois, quatro, ou oito bytes contíguos. Quando vários bytes de memória forem usados em uma operação, só o endereço do primeiro byte do grupo é especificado.

### Perguntas:

Uma operação de **load** altera o padrão de bits na memória?

NÃO: uma cópia da memória é realizada em algum registrador.

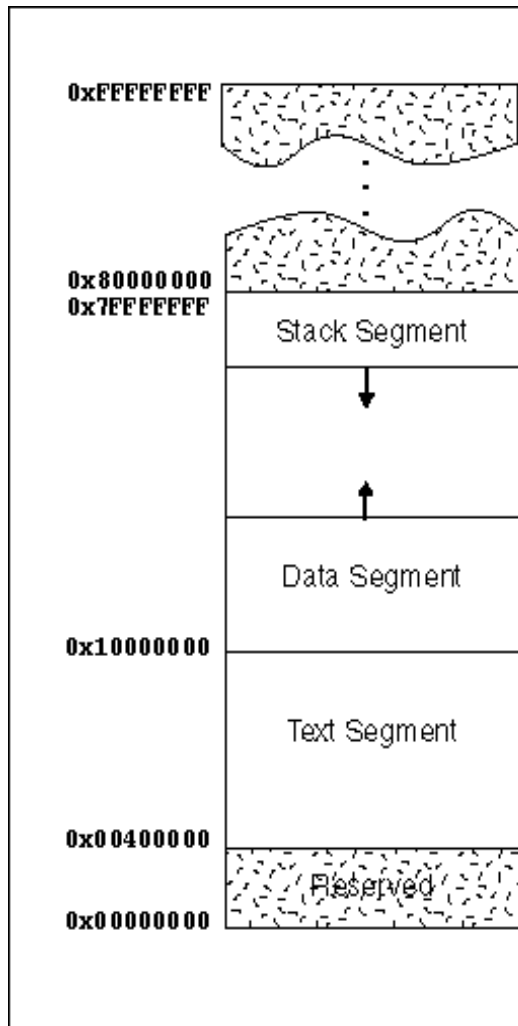
Uma operação de **store** altera o padrão de bits na memória?

SIM: o padrão de bits substitui o conteúdo da memória no endereço designado.

## Layout da Memória

Operações de Load e Store copiam o padrão de bits da fonte no destino. A fonte (registrador ou memória) não muda. Claro que, o padrão no destino é substituído pelo padrão da fonte.

A Memória é construída para armazenar esses padrões. Tanto Instruções quanto dados são esses padrões, e qualquer um destes pode ser armazenado em qualquer lugar da memória (pelo menos, tão longe quanto o hardware esteja preparado para tal.) Porém, é conveniente para os programadores e sistemas de software organizarem a memória de forma que instruções e dados estejam separados. A figura ilustra o modo que freqüentemente os sistemas operacionais do MIPS dispõem a memória.



Embora o espaço de endereçamento seja de 32 bits, o endereço de 0x80000000 a 0xFFFFFFFF não está completamente disponível aos programas do usuário. Eles são usados para o sistema operacional e para ROM. Quando um chip do MIPS é usado em um controlador embutido, o programa de controle existe em uma ROM, dentro desta metade superior do espaço de endereçamento.

As partes do espaço de endereçamento acessíveis a um programa do usuário são divididas como se segue:

**Segmento de texto (Text Segment):** Significa o programa do usuário em linguagem de máquina (ou text).

**Segmento de dados (Data Segment):** Os dados nos quais o programa opera. Parte dos dados é estática. Estes são os dados que são alocados pelo montador os quais não mudam o tamanho enquanto um programa é executado. Os valores contidos nesses dados mudam; "estático" significa que o tamanho em bytes não muda durante execução. Na parte superior dos dados estáticos estão os dados dinâmicos.

Estes são dados alocados e desalocados enquanto o programa é executado. Em "C", a alocação e desalocação são realizadas pelos comandos malloc () e free ().

**Segmento de pilha (Stack segment):** No topo do espaço de endereçamento do usuário está a pilha. Em linguagens de níveis mais altos, variáveis locais e parâmetros são empilhados e desempilhados na pilha quando procedimentos são ativados ou desativados.

### Pergunta:

Quando o programa roda, cresce o segmento de dados para cima (quando variáveis dinâmicas são alocadas) e a pilha cresce descendentemente (quando procedimentos são chamados). Isso é sensato? (Sugestão: quanto de memória resta ao usuário quando os dois segmentos se encontram?)

Resp.:

Sim. Em lugar de se alocar uma quantia fixa de memória para cada, este arranjo significa que cada parte pode crescer dentro da memória disponível. Quando os dois se encontrarem, não resta nenhuma memória disponível.

## Registradores

Freqüentemente dados consistem em vários bytes contíguos. Cada fabricante de computador tem sua própria idéia do que chamar agrupamento maior que um byte. O seguinte é usado para o MIPS:

byte - oito bit.

word - quatro bytes, 32 bit.

Double word - oito bytes, 64 bit.

Um registrador é uma parte do registrador que armazena um padrão de bits. No MIPS, um registrador armazena 32 bits.

Existem diversos registradores em um processador, mas apenas alguns podem ser utilizados para a programação em assembly, os outros são utilizados por outras atividades do processador.

Uma operação de **load** copia um padrão de bits da memória em um registrador.

Uma operação de **store** copia um padrão de bits de um registrador para a memória.

Os registradores utilizáveis para a programação em assembly são chamados registradores de uso geral (**general purpose registers** e **floating point registers**).

Existem 32 **general purpose registers**. Cada um possui 32 bit. Para a programação eles serão nomeados como \$0, \$1, \$2, ... , \$31. Existem 32 floating point registers. Esses serão discutidos mais tarde.

Um dos registradores de uso geral possui um valor fixo com o valor 0x00000000 (todos os bits em zero).

Pergunta:

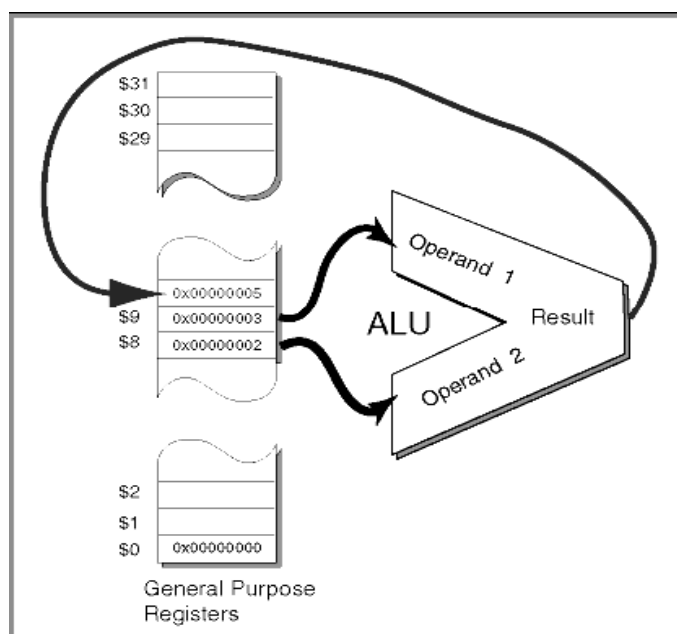
Qual dos registradores \$0, \$1, \$2, ... , \$31 você acha que possui zeros?

**Resp.:** O registrador \$0, isso acontece porque o padrão 0x00000000 ocorre com muita freqüência durante a programação. Ele também representa o **null**, usado na representação de algumas estruturas de dados.

## Registradores e a ALU

A unidade lógica e aritmética (ALU) de um processador executa a aritmética inteira e as operações lógicas.

Por exemplo, uma de suas operações é somar dois inteiros de 32-bit. Um inteiro usado como entrada em uma operação é chamado um operando. Um dos operandos para a ALU sempre é contido em um registrador. O outro operando podem estar em um registrador ou pode fazer parte da própria instrução de máquina. O resultado da operação é armazenado em um registrador de uso geral.



Instruções de máquina que usam a ALU especificam quatro coisas:

1. a operação para executar.
2. um primeiro operando (frequentemente em um registrador).
3. um segundo operando (frequentemente em um registrador).
4. um registrador que recebe o resultado.

### Convenções da nomenclatura de registradores

Os registradores de propósito geral são numerados \$0 a \$31. Porém, através de convenção (e às vezes através de hardware) registradores diferentes são usados para propósitos diferentes.

Além de um número \$0 - \$31, registros têm um nome mnemônico (um nome que faz lembrar de seu uso), por exemplo o registrador \$0 tem o zero de nome mnemônico. A tabela a seguir mostra os 32 registradores e o uso convencional de cada um.

Registros \$0 e \$31 são os únicos com comportamentos diferentes dos outros. O Registrador \$0 é permanentemente ajustado para conter zero e o registrador \$31 é automaticamente usado por algumas instruções de acoplamento de subrotinas para armazenar o endereço de retorno.

Número do Registrador	Nome Mnemonico	Uso Convencional
\$0	zero	Permanently 0
\$1	\$at	Assembler Temporary (reserved)
\$2, \$3	\$v0, \$v1	Value returned by a subroutine
\$4-\$7	\$a0-\$a3	Arguments to a subroutine
\$8-\$15	\$t0-\$t7	Temporary (not preserved across a function call)
\$16-\$23	\$s0-\$s7	Saved registers (preserved across a function call)
\$24, \$25	\$t8, \$t9	Temporary
\$26, \$27	\$k0, \$k1	Kernel (reserved for OS)
\$28	\$gp	Global Pointer
\$29	\$sp	Stack Pointer
\$30	\$fp	Frame Pointer
\$31	\$ra	Return Address (Automatically used in some instructions)

## Transferência de Dados: Memória para Registradores (lw)

Para transferir uma palavra de dados, nós devemos especificar duas coisas:

Registrador: especifique este pelo número (0 - 31)

Endereço da memória: mais difícil

Pense a memória como um array único uni-dimensional, de modo que nós podemos endereçá-la simplesmente fornecendo um ponteiro para um endereço da memória.

Outras vezes, nós queremos ser capazes de deslocar a partir deste ponteiro.

Para especificar um endereço de memória e copiar dele, especifique duas coisas:

- Um registrador que contém um ponteiro para a memória.

- Um deslocamento numérico (em bytes)

O endereço de memória desejado é a soma destes dois valores.

Exemplo: 8(\$t0)

Especifica o endereço de memória apontado pelo valor em \$t0, mais 8 bytes

Sintaxe da instrução de carga (load): 1 2,3(4)

1) nome da operação (instrução)

2) registrador que receberá o valor

3) deslocamento numérico em bytes.

4) registrador contendo o ponteiro para a memória

Nome da Instrução:

lw (significa Load Word, logo 32 bits ou uma palavra é carregada por vez)

Exemplo: lw \$t0,12(\$s0)

Esta instrução pegará o ponteiro em \$s0, soma 12 bytes a ele, e então carrega o valor da memória apontado por esta soma calculada no registrador \$t0

Notas:

\$s0 é chamado **registrador base**

12 é chamado **deslocamento (offset)**

Deslocamento é geralmente utilizado no acesso de elementos de array ou estruturas: reg base aponta para o início do array ou estrutura.

## Transferência de Dados: Registrador para Memória (sw)

Também queremos armazenar um valor do registrador na memória.

Sintaxe da instrução store é idêntica à da instrução load.

Nome da Instrução:

sw (significa Store Word, logo 32 bits ou uma palavra será carregada por vez)

Exemplo: sw \$t0,12(\$s0)

Esta instrução tomará o ponteiro em \$s0, somará 12 bytes a ele, e então armazenará o valor do registrador \$t0 no endereço de memória apontado pela soma calculada.

# Programas

## // programa 9

Considere a memória inicial da seguinte forma:

.text

.data

x1: .word 15

x2: .word 25

x3: .word 13

x4: .word 17

soma: .word -1

Escrever um programa que leia todos os números, calcule e substitua o valor da variável soma por este valor.

## // programa 10

Considere o seguinte programa:  $y = 127x - 65z + 1$

Faça um programa que calcule o valor de y conhecendo os valores de x e z. Os valores de x e z estão armazenados na memória e, na posição imediatamente a seguir, o valor de y deverá ser escrito, ou seja:

.data

x: .word 5

z: .word 7

y: .word 0 # esse valor deverá ser sobrescrito após a execução do programa.

## // programa 11

Considere o seguinte programa:  $y = x - z + 300000$

Faça um programa que calcule o valor de y conhecendo os valores de x e z. Os valores de x e z estão armazenados na memória e, na posição imediatamente a seguir, o valor de y deverá ser escrito, ou seja:

.data

x: .word 100000

z: .word 200000

y: .word 0 # esse valor deverá ser sobrescrito após a execução do programa.

## // programa 12

Considere a seguinte situação:

int \*\*\*x;

onde x contem um ponteiro para um ponteiro para um ponteiro para um inteiro.

Nessa situação, considere que a posição inicial de memória contenha o inteiro em questão.

Coloque todos os outros valores em registradores, use os endereços de memória que quiser dentro do espaço de endereçamento do Mips.

Resumo do problema:

$k = \text{MEM} [ \text{MEM} [ \text{MEM} [ x ] ] ]$ .

Crie um programa que crie a estrutura de dados acima, leia o valor de K, o multiplique por 2 e o reescreva conhecendo-se apenas o valor de x.

## Novas instruções bne, beq, j, slt, slti, mult, div mfhi e mflo.

Consulte os slides/aula para as instruções:

Jump ( j )

Branch if equal (beq)

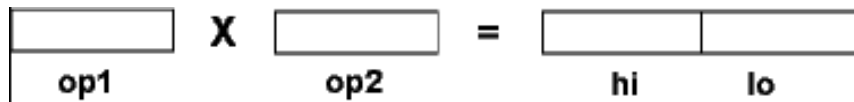
Branch if not equal (bne)

Set on less then (slt)

Set on less then com imediato (slti)

## 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**.

### mult e multu

mult s,t                      # hilo <-- \$s \* \$t (operandos em com. de dois)

multu s,t                    # hilo <-- \$s \* \$t (operandos unsigned )

A multiplicação inteira nunca causa um trap.

**Perg.:** Dois inteiros pequenos são multiplicados, onde está o resultado?

**Resp:** Se o resultado for inferior a 32 bits, em lo, hi possuirá zeros.

## 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 0100 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 1011 1100 1010 1001 0010 0010

O número possui 23 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:  
 $01001010 \times 00010101$

**Resp:** Aproximadamente 12 bits.

### **mfhi e mflo**

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

mfhi d            # d <-- hi. Mover de Hi  
mflo d            # d <-- lo. Mover de Lo

**Perg:** Deve-se mover os resultados de lo e hi antes de uma multiplicação seguinte?

**Resp:** Sim.

**Perg.:** Como computar  $5 \times X - 74$  ?

**Resp.:**

```
## novoMult.asm
##
## Programa para calcular  $5 \times x - 74$ 
##
## $8 x
## $9 resultado
.text
.globl main
main:
ori $8, $0, 12            # x em $8
ori $9, $0, 5            # 5 em $9
mult $9, $8            # lo <-- 5x
mflo $9            # $9 = 5x
addi $9, $9, -74        # $9 = 5x - 74
## End of file
```

**Perg.:** O que significa o "u" nas seguintes instruções:

addu \_\_\_\_\_  
multu \_\_\_\_\_

**Resp:**

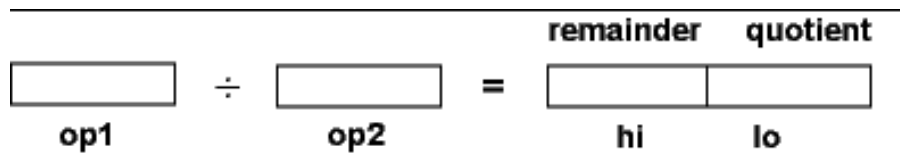
addu, não acontece um trap no overflow.

multu, os operandos são unsigned.



**div e divu**

De maneira análoga, temos em  $hi$  e  $lo$  o resultado e o quociente, respectivamente.



```
div s,t      # lo <-- s div t
              # hi <-- s mod t
              # operandos em comp. de dois
divu s,t     # lo <-- s div t
              # hi <-- s mod t
              # operandos em comp. De dois
```

**Perg.:** Calcular  $(y + x) / (y - x)$

**Resp.:**

```
## exemplodiv.asm
##
## Programa para calcular  $(y + x) / (y - x)$ 
##
## $8 x
## $9 y
## $10 x/y
## $11 x%y
.text
.globl main
main:
ori $8, $0, 8           # x em $8
ori $9, $0, 36          # y em $9
add $10, $9, $8          # $10 <-- (y+x)
sub $11, $9, $8          # $11 <-- (y-x)
div $10, $11             # hilo <-- (y+x)/(y-x)
mflo $10                 # $10 <-- quociente
mfhi $11                 # $11 <-- resultado
## End of file
```

**Perg.:**  $(36+8) / (36-8) =$

**Resp:**  $(36+8) / (36-8) = 1 \text{ R } 16$ , or  $0x1 \text{ R } 0x10$

# Exercícios

## Responda

1. Se tivermos 2 inteiros, cada um com 32 bits, quantos bits podemos esperar para o produto?  
**A.** 16  
**B.** 32  
**C.** 64  
**D.** 128
2. Quais os registradores que armazenam os resultados na multiplicação?  
**A.** high e low  
**B.** hi e lo  
**C.** R0 e R1  
**D.** \$0 e \$1
3. Qual a operação usada para multiplicar inteiros em comp. de dois?  
**A.** mult  
**B.** multu  
**C.** multi  
**D.** mutt
4. Qual instrução move os bits menos significativos da multiplicação para o reg. 8?  
**A.** move \$8,lo  
**B.** mvlo \$8,lo  
**C.** mflo \$8  
**D.** addu \$8,\$0,lo
5. Se tivermos dois inteiros, cada um com 32 bits, quantos bits deveremos estar preparados para receber no **quociente**?  
**A.** 16  
**B.** 32  
**C.** 64  
**D.** 128
6. Após a instrução div, qual registrador possui o quociente?  
**A.** lo  
**B.** hi  
**C.** high  
**D.** \$2
7. Qual a inst. Usada para dividir dois inteiros em comp. de dois?  
**A.** dv  
**B.** divide  
**C.** divu  
**D.** div

8. Faça um arithmetic shift right de dois no seguinte padrão de bits: 1001 1011

A. 1110 0110

B. 0010 0110

C. 1100 1101

D. 0011 0111

9. Qual o efeito de um **arithmetic shift right** de uma posição?

A. Se o inteiro for unsigned, o shift o divide por 2. Se o inteiro for signed, o shift o divide por 2.

B. Se o inteiro for unsigned, o shift o divide por 2. Se o inteiro for signed, o shift pode resultar em um valor errado.

C. Se o inteiro for unsigned, o shift pode ocasionar um valor errado. Se o inteiro for signed, o shift o divide por 2.

D. O shift multiplica o número por dois.

10. Qual sequencia de instruções avalia  $3x+7$ , onde  $x$  é iniciado no reg. \$8 e o resultado armazenado em \$9?

A.

ori \$3,\$0,3

mult \$8,\$3

mflo \$9

addi \$9,\$9,7

B.

ori \$3,\$0,3

mult \$8,\$3

addi \$9,\$8,7

C.

ori \$3,\$0,3

mult \$8,\$3

mfhi \$9

addi \$9,\$9,7

D.

mult \$8,3

mflo \$9

addi \$9,\$9,7

## PROGRAMAS

### // programa 13:

Escreva um programa que leia um valor A da memória, identifique se o número é negativo ou não e encontre o seu módulo. O valor deverá ser reescrito sobre A.

### // programa 14:

Escreva um programa que leia da memória um valor de Temperatura TEMP. Se  $TEMP \geq 30$  e  $TEMP \leq 50$  uma variável FLAG, também na memória, deverá receber o valor 1, caso contrário, FLAG deverá ser zero.

### // programa 15:

Escrever um programa que crie um vetor de 100 elementos na memória onde  $vetor[i] = 2*i + 1$ . Após a última posição do vetor criado, escrever a soma de todos os valores armazenados do vetor.

### // programa 16

$$1) \ y = \begin{cases} x^4 + x^3 - 2x^2 & \text{se } x \text{ for par} \\ x^5 - x^3 + 1 & \text{se } x \text{ for impar} \end{cases}$$

Os valores de x devem ser lidos da primeira posição livre da memória e o valor de y deverá ser escrito na segunda posição livre.

### // programa 17:

Considere que a partir da primeira posição livre da memória temos um vetor com 100 elementos. Escrever um programa que ordene esse vetor de acordo com o algoritmo da bolha. Faça o teste colocando um vetor totalmente desordenado e verifique se o algoritmo funciona.

### // programa 18

Para a expressão a seguir, escreva um programa que calcule o valor de k:

$$k = x * y$$

O valor de x deve ser lido da primeira posição livre da memória e o valor de y deverá lido da segunda posição livre. O valor de k, após calculado, deverá ainda ser escrito na terceira posição livre da memória.

### // programa 19

Para a expressão a seguir, escreva um programa que calcule o valor de k:

$$k = x^y$$

O valor de x deve ser lido da primeira posição livre da memória e o valor de y deverá lido da segunda posição livre. O valor de k, após calculado, deverá ainda ser escrito na terceira posição livre da memória. Obs. Procure executar este exercício considerando que a instrução e multiplicação não poderá ser utilizada (apenas add, addi, etc).