



## **Laboratório 2**

Alunas: Jéssica Regina dos Santos e Myllena da Conceicao Correa

Matrícula: 22100626 e 22104061

Disciplina: Organização de Computadores I

Professor: Marcelo Daniel Berejuck

Entrega: 27/04/25

**Objetivo:** Este relatório descreve a implementação de três programas em Assembly MIPS para realizar operações com matrizes, incluindo multiplicação, transposição e armazenamento de resultados em memória e arquivos. O programa foi desenvolvido utilizando o simulador MARS e atende aos requisitos das questões apresentadas em laboratório.

### Questão 1

**Enunciado:** Dadas as matrizes A e B, armazenadas em memória, elabore um programa em Assembly para o processador MIPS (usando o simulador MARS) que encontre a matriz resultante do produto  $A \cdot B^T$ . Armazene a matriz resultante na memória de dados.

$$A = \begin{vmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{vmatrix} \qquad B = \begin{vmatrix} 1 & -2 & 5 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{vmatrix}$$

#### **Resposta:**

Implementamos um programa para calcular o produto da matriz A pela transposta da matriz B ( $A \cdot B^T$ ), conforme solicitado na questão. Ao final, armazenamos a matriz resultante C na memória de dados do MIPS.

1º) As matrizes A e B foram armazenadas como arrays na memória, seguindo a ordem linha por linha.

```
A: .word 1, 2, 3, 0, 1, 4, 0, 0, 1    # Matriz A 3x3
B: .word 1, -2, 5, 0, 1, -4, 0, 0, 1 # Matriz B 3x3

# A: .word 1, 2, 3, 0, 1, 4, 0, 0, 1
# representa
# A[0][0] = 1  A[0][1] = 2  A[0][2] = 3
# A[1][0] = 0  A[1][1] = 1  A[1][2] = 4
# A[2][0] = 0  A[2][1] = 0  A[2][2] = 1
# logo,
# se a matriz tem 3 colunas, então a posição linear de A[i][k] é: posicao = (i * 3) + k
# Para obter o endereço de memória real, multiplicamos essa posição por 4 (porque cada inteiro ocupa 4 bytes).
```

2º) A matriz resultante C foi alocada em uma região de memória reservada, onde cada elemento ocupa 4 bytes (equivalente a 32 bits).

```
C: .space 36 # Espaço para matriz C (3x3 * 4 bytes) = 9 elementos * 4 (cada inteiro ocupa 4 bytes na arquitetura MIPS (32 bits)).
# Isso reserva 36 bytes de memória para armazenar os 9 inteiros da matriz C
```

3º) Para evitar calcular  $B^T$ , escolhemos acessar os elementos de B como  $B[j][k]$  em vez de  $B^T[k][j]$ , aproveitando a propriedade matemática  $B^T[k][j] = B[j][k]$ .

```
# Sendo:
# A (3x3) = 1 2 3
#           0 1 4
#           0 0 1
# B (3x3) = 1 -2 5
#           0 1 -4
#           0 0 1
# B^t (3x3) = 1 0 0
#            -2 1 0
#            5 -4 1
# Para calcular a transposta de uma matriz, basta trocar linhas por colunas
# O resultado C = A * B^t será uma matriz 3x3
# C[i][j] = A[i][k] * B^t[k][j]
#          = A[i][k] * B[j][k]      pois B^t[k][j] = B[j][k]
```

4º) Para realizar Multiplicação de Matrizes, nosso algoritmo utiliza três loops aninhados para percorrer:

- Loop externo (i): Percorre as linhas de A (0 a 2).
- Loop intermediário (j): Percorre as colunas de  $B^T$ , equivalentes às linhas de B.
- Loop interno (k): Realiza o produto escalar entre a linha i de A e a linha j de B. Índice comum é k para o produto escalar.

Para cada elemento  $C[i][j]$ , calculamos  $C[i][j] = \sum_{k=0}^2 A[i][k] \cdot B[j][k]$ .

[Checar direto no código faz mais sentido, para visualização dos loops aninhados]

5º) Em seguida, o endereço de cada elemento é calculado conforme a posição linear. Para uma matriz 3x3, o elemento  $[i][k]$  está na posição  $(i * 3 + k) * 4$  (considerando 4 bytes por inteiro).

```
# calcular A[i][k]
li $t4, 3                # número de colunas de A
mul $t5, $t0, $t4        # i * 3
add $t5, $t5, $t2        # (i*3) + k
sll $t5, $t5, 2          # * 4 (bytes)
la $t6, A
add $t6, $t6, $t5
lw $t7, 0($t6)          # $t7 = A[i][k]
```

6º) Ao final, os valores calculado para  $C[i][j]$  são armazenados nas posições correspondentes em C:

```
# armazenar C[i][j]
li $t4, 3
mul $t5, $t0, $t4      # i * 3
add $t5, $t5, $t1      # (i*3) + j
sll $t5, $t5, 2
la $t6, C
add $t6, $t6, $t5
sw $t3, 0($t6)
```

7º) O programa encerra com a syscall 10.

### Resultados Esperados

A matriz resultante  $C = A \cdot B^T$  é:

Produto de matrizes: linhas da primeira matriz são multiplicados por colunas da matriz segundo.

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 5 & -4 & 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + 2 \cdot (-2) + 3 \cdot 5 & 1 \cdot 0 + 2 \cdot 1 + 3 \cdot (-4) & 1 \cdot 0 + 2 \cdot 0 + 3 \cdot 1 \\ 0 \cdot 1 + 1 \cdot (-2) + 4 \cdot 5 & 0 \cdot 0 + 1 \cdot 1 + 4 \cdot (-4) & 0 \cdot 0 + 1 \cdot 0 + 4 \cdot 1 \\ 0 \cdot 1 + 0 \cdot (-2) + 1 \cdot 5 & 0 \cdot 0 + 0 \cdot 1 + 1 \cdot (-4) & 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} 12 & -10 & 3 \\ 18 & -15 & 4 \\ 5 & -4 & 1 \end{pmatrix}$$

### [Calculadora de Matrizes](#)

Após a execução, os valores da matriz C podem ser verificados nos endereços reservados. A comparação confirma a precisão da resposta final.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	1	2	3	0	1	4	0	0
268501024	1	1	-2	5	0	1	-4	0
268501056	0	1	12	-10	3	18	-15	4
268501088	5	-4	1	0	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0
268501184	0	0	0	0	0	0	0	0

### Questão 3

**Enunciado:** Considere o problema apresentado na questão 1. Altere o seu programa para que ele “chame” dois procedimentos: um para multiplicar matrizes (PROC\_MUL) e outro para gerar uma matriz transposta (PROC\_TRANS)

#### **Resposta:**

Conforme solicitado, implementamos em Assembly MIPS um programa que calcula o produto da matriz A pela transposta da matriz B ( $A \cdot B^T$ ), agora, estruturando o código em dois procedimentos principais: PROC\_TRANS para transposição e PROC\_MUL para multiplicação das matrizes.

Como já havíamos implementado a questão 1, reutilizamos o código da mesma, com algumas alterações:

1º) Dividimos a lógica em dois procedimentos autônomos PROC\_TRANS para transposição matricial PROC\_MUL para multiplicação matricial. Isso nos proporcionou maior organização e reutilização de código.

2º) Adicionamos alocação explícita para a matriz transposta e mantivemos os espaços originais para A, B e C.

```
.data
A:  .word 1, 2, 3, 0, 1, 4, 0, 0, 1 # matriz A (3x3)
B:  .word 1, -2, 5, 0, 1, -4, 0, 0, 1 # matriz B 3x3
BT: .space 36 # transposta de B (3x3)
C:  .space 36 # resultado (3x3)
```

3º) Substituímos partes “monolíticas” por chamadas de procedimentos. Cada procedimento preserva o contexto anterior usando registradores temporários t0 à t9 .

```
.text
main:
    # chama PROC_TRANS para calcular BT = B^t
    jal PROC_TRANS

    # chama PROC_MUL para calcular C = A * BT
    jal PROC_MUL
```

## Benefícios das mudanças

### - Organização

Na versão anterior, todo o código estava implementado de forma linear no programa principal, o que dificultava a identificação das partes lógicas do algoritmo.

Já na versão modularizada, organizamos os códigos em procedimentos especializados – PROC\_TRANS para transposição matricial e PROC\_MUL para multiplicação – criando uma estrutura mais lógica e intuitiva, refletindo os diferentes “estágios” do processamento matricial.

### - Reutilização

O código original não permitia reutilização, pois toda a lógica estava embutida no fluxo principal. A nova versão, ao modularizar as operações, permite que os procedimentos sejam chamados múltiplas vezes quando necessário, inclusive poderiam ser utilizados em outros programas sem modificações. Isso seria muito útil em aplicações que precisam realizar várias operações matriciais, por exemplo.

### - Legibilidade

Enquanto a versão linear exigia um esforço significativo para entender as diferentes partes do algoritmo, a versão modular apresenta uma estrutura mais clara. Cada procedimento tem uma responsabilidade definida (indicada até por seus nomes), e a lógica interna de cada operação fica encapsulada, tornando o código mais fácil de ler e entender.

### - Manutenção

A manutenção do código linear era difícil, pois qualquer modificação exigia que navegássemos por todo o programa. Na versão modular, como cada funcionalidade está isolada em seu próprio procedimento, agora, atualizações podem ser feitas de forma localizada. Por exemplo, se o método de multiplicação precisar ser alterado, basta modificar o PROC\_MUL sem afetar o resto do sistema.

### - Consistência

Na implementação original, todos os cálculos estavam embutidos no fluxo principal, o que poderia levar a inconsistências. A versão modular encapsula a lógica de cada operação, garantindo que cada procedimento processe seus dados de forma independente. Isso também reduz a chance de efeitos colaterais indesejados durante modificações.

## Nova organização do código

```
1 .data
2 A: .word 1, 2, 3, 0, 1, 4, 0, 0, 1 # matriz A (3x3)
3 B: .word 1, -2, 5, 0, 1, -4, 0, 0, 1 # matriz B 3x3
4 BT: .space 36 # transposta de B (3x3)
5 C: .space 36 # resultado (3x3)
6
7 .text
8 main:
9     # chama PROC_TRANS para calcular BT = B^t
10    jal PROC_TRANS
11
12    # chama PROC_MUL para calcular C = A * BT
13    jal PROC_MUL
14
15    # finaliza o programa
16    li $v0, 10
17    syscall
18
19 #####
20 # PROC_TRANS: calcula a transposta de B e armazena em BT
21 # entrada: B
22 # saída: BT (transposta de B)
23 #####
24
25 PROC_TRANS:
26     li $t0, 0 # i (linha de B)
27 trans_i:
28     li $t1, 0 # j (coluna de B)
29 trans_j:
30     # posição em B: (i*3) + j
31     mul $t2, $t0, 3
32     add $t2, $t2, $t1
33     sll $t2, $t2, 2
34     la $t3, B
35     add $t3, $t3, $t2
36     lw $t4, 0($t3)
37
38     # posição em BT: (j*3) + i
39     mul $t2, $t1, 3
40     add $t2, $t2, $t0
41     sll $t2, $t2, 2
42     la $t3, BT
43     add $t3, $t3, $t2
44     sw $t4, 0($t3)
45
46     addi $t1, $t1, 1
47     li $t5, 3
48     blt $t1, $t5, trans_j
49
50     addi $t0, $t0, 1
51     blt $t0, $t5, trans_i
52
53     jr $ra
54
```

```

55 #####
56 # PROC_MUL: multiplica A por BT e armazena em C
57 # Entrada: A, BT (transposta de B)
58 # saída: C
59 #####
60
61 PROC_MUL:
62     li $t0, 0 # i (linha de A)
63 mul_i:
64     li $t1, 0 # j (coluna de BT)
65 mul_j:
66     li $t2, 0 # k (índice comum)
67     li $t3, 0 # soma acumulada
68
69 mul_k:
70     # A[i][k]
71     mul $t4, $t0, 3
72     add $t4, $t4, $t2
73     sll $t4, $t4, 2
74     la $t5, A
75     add $t5, $t5, $t4
76     lw $t6, 0($t5)
77
78     # BT[k][j]
79     mul $t4, $t2, 3
80     add $t4, $t4, $t1
81     sll $t4, $t4, 2
82     la $t5, BT
83     add $t5, $t5, $t4
84     lw $t7, 0($t5)
85
86     # soma parcial
87     mul $t8, $t6, $t7
88     add $t3, $t3, $t8
89
90     addi $t2, $t2, 1
91     li $t9, 3
92     blt $t2, $t9, mul_k
93
94     # armazena C[i][j]
95     mul $t4, $t0, 3
96     add $t4, $t4, $t1
97     sll $t4, $t4, 2
98     la $t5, C
99     add $t5, $t5, $t4
100    sw $t3, 0($t5)
101
102    addi $t1, $t1, 1
103    blt $t1, $t9, mul_j
104
105    addi $t0, $t0, 1
106    blt $t0, $t9, mul_i
107
108    jr $ra
109

```



## Questão 2

**Enunciado:** Considere o problema apresentado na questão 1, porém armazene a matriz resultante em um arquivo com extensão .txt.

Importante: você deve converter os elementos da matriz resultante em representações ASCII para poderem ser visualizados por uma pessoa no arquivo .txt.

### **Resposta:**

Para essa questão, implementamos a funcionalidade de salvar a matriz resultante em um arquivo de texto (.txt), atendendo à exigência de tornar os resultados acessíveis em formato legível.

Para isso, introduzimos ao código da questão um novos syscalls específicos para manipulação de arquivos, incluindo:

#### - Abertura do arquivo (syscall 13)

Syscall 13 é responsável por abrir um arquivo existente ou criar um novo, caso ele não exista.

```
# Agora vamos salvar a matriz C no arquivo
# Primeiro, criar/abrir o arquivo
li $v0, 13          # syscall para abrir arquivo
la $a0, nome_arquivo # nome do arquivo
li $a1, 1           # flags (1 = escrita)
li $a2, 0           # modo (ignorado)
syscall
move $s6, $v0       # salvar o descritor de arquivo
```

Antes de realizar a chamada de sistema, o código prepara os argumentos.

\$a0 recebeu o endereço da string que contém o nome do arquivo ("matriz\_resultado.txt"),

\$a1 define o modo de acesso (1 para escrita) e \$a2 foi ignorado.

Ao executar a syscall 13, o sistema retorna um descritor de arquivo em \$v0.

Esse descritor é armazenado em \$s6 para ser utilizado depois.

#### - Escrita no Arquivo (syscall 15)

Syscall 15 foi utilizada para escrever os dados no arquivo.

```
# Escrever buffer no arquivo
li $v0, 15          # syscall para escrever no arquivo
move $a0, $s6       # descritor de arquivo
la $a1, buffer       # buffer
subu $a2, $s1, $s0   # tamanho = ponteiro atual - início do buffer
syscall
```

Antes de realizar a chamada de sistema, o código prepara os argumentos.

\$a0 recebe o descritor de arquivo salvo anteriormente.

\$a1 contém o endereço do buffer onde os dados a serem escritos estão armazenados.

\$a2 especifica o número de bytes a escrever (calculado pela diferença entre o ponteiro atual e o início do buffer).

Durante a execução, o sistema operacional transfere o conteúdo do buffer para o arquivo.

Se a escrita for bem-sucedida, \$v0 conterá o número de bytes escritos. Caso contrário, retorna um valor negativo.

É importante destacar que o buffer foi previamente preenchido com a [representação ASCII da matriz](#), incluindo espaços e quebras de linha para formatação.

*# Função para converter inteiro para ASCII*

*# Entrada: \$a0 = número*

*# Saída: dígitos armazenados no buffer (apontado por \$s1)*

*inteiro\_para\_ascii:*

*move \$t8, \$a0                   # salvar número original*

*li \$t9, 10                   # divisor*

*# Tratar caso especial de zero*

*bnez \$a0, nao\_zero*

*li \$t5, 48                   # '0'*

*sb \$t5, 0(\$s1)*

*addi \$s1, \$s1, 1*

*jr \$ra*

*nao\_zero:*

*# Tratar números negativos*

*bgez \$a0, positivo*

*li \$t5, 45                   # '-'*

*sb \$t5, 0(\$s1)*

*addi \$s1, \$s1, 1*

*neg \$a0, \$a0               # tornar positivo*

```

positivo:
    # Contar dígitos
    li $t4, 0                # contador de dígitos
    move $t5, $a0            # cópia do número

contar_digitos:
    divu $t5, $t9            # dividir por 10
    mflo $t5                # quociente
    addi $t4, $t4, 1        # incrementar contador
    bnez $t5, contar_digitos

    # Armazenar dígitos (começando pelo mais significativo)
    add $s1, $s1, $t4        # avançar para o final
    move $t5, $a0            # cópia do número

armazenar_digitos:
    addi $s1, $s1, -1        # retroceder uma posição
    divu $t5, $t9            # dividir por 10
    mfhi $t6                # resto
    mflo $t5                # quociente
    addi $t6, $t6, 48        # converter para ASCII
    sb $t6, 0($s1)          # armazenar dígito
    bnez $t5, armazenar_digitos

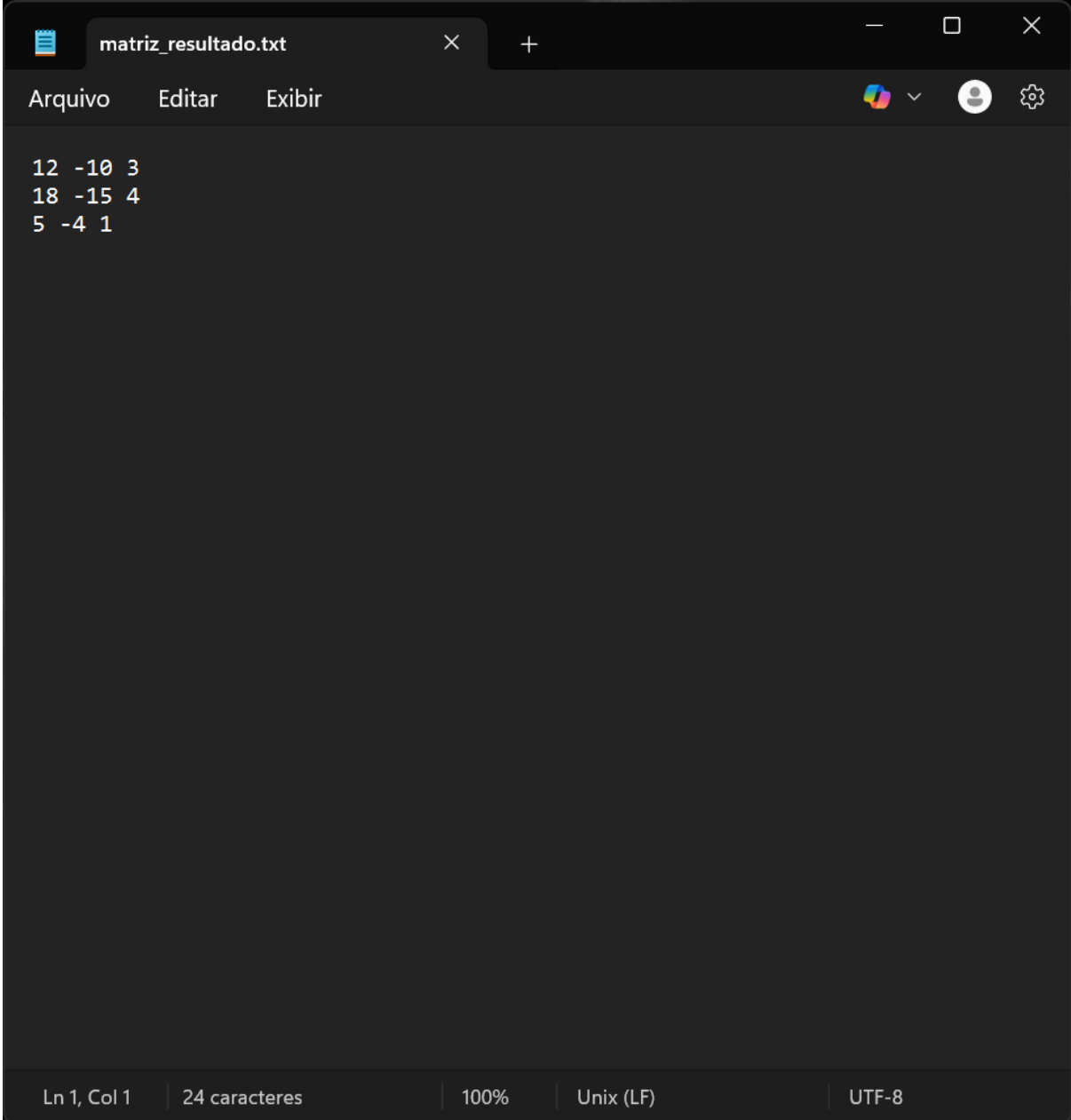
    add $s1, $s1, $t4        # avançar para após o último dígito
    jr $ra

```

- Fechamento do Arquivo (syscall 16)

Syscall 16 foi chamada para fechar o arquivo. O descritor de arquivo (armazenado em \$s6) é passado via \$a0.

## Arquivo gerado



The image shows a code editor window with a dark theme. The title bar at the top displays the file name 'matriz\_resultado.txt' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with three items: 'Arquivo', 'Editar', and 'Exibir'. On the right side of the menu bar, there are icons for a color palette, a dropdown arrow, a user profile, and a settings gear. The main editing area contains three lines of text, each representing a row of a 3x3 matrix. The status bar at the bottom provides details about the current cursor position and file encoding: 'Ln 1, Col 1', '24 caracteres', '100%', 'Unix (LF)', and 'UTF-8'.

```
12 -10 3
18 -15 4
5 -4 1
```