Relatório Laboratório 02

Maria Eduarda Teixeira Costa e Sofia Gazolla da Costa Silva

Departamento de Informática e Estatística - Universidade Federal de Santa Catarina

Organização de Computadores I

Professor Marcelo Daniel Berejuck

Setembro 2025

1 Introdução

Neste segundo laboratório a atividade proposta consistiu na elaboração de um código Assembly para o processador MIPS, fazendo o uso do simulador MARS.

O exercício envolveu a elaboração de um código que fizesse o cálculo para encontrar a matriz resultante do produto de uma matriz A, por uma matriz B transposta e armazenar a matriz resultante em um arquivo .txt.

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

2 Objetivo

Para que se possa entender melhor como o código que foi desenvolvido, é importante compreender como funciona a transposição e a multiplicação de matrizes.

A transposta de uma matriz é obtida transformando as suas linhas em colunas e as suas colunas em linhas, da seguinte maneira:

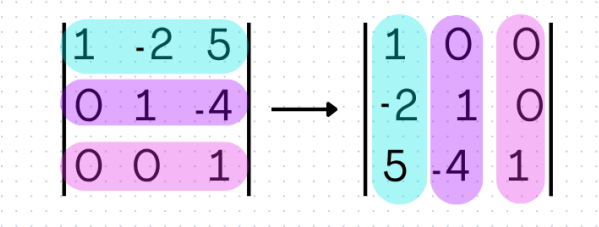

Figure 1: Transposição de Matrizes

Já para realizar a multiplicação de duas matrizes, multiplicamos cada elemento de cada linha da primeira matriz por cada elemento de cada coluna da segunda matriz (nesse caso, $A \times B^T$). Na figura a seguir, é demonstrado como se acha o primeiro elemento da matriz resultante.

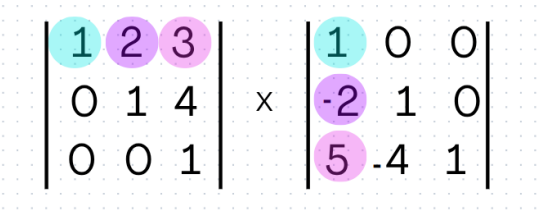

Figure 2: Multiplicação de Matrizes

Após isso, é necessário somar o resultado de todas as multiplicações:

$$\text{resultado}_{11} = 1 \times 1 + 2 \times (-2) + 3 \times 5 = 1 - 4 + 15 = 12$$

Esse resultado representa o primeiro elemento da matriz resultante.

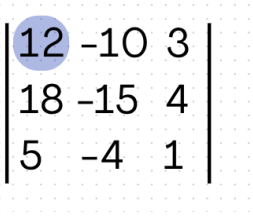

Figure 3: Matriz Resultante

Para achar os outros, devemos repetir esse processo. O segundo elemento da primeira linha será encontrado multiplicando a primeira linha pela segunda coluna e assim respectivamente.

3 Desenvolvimento

(a) Primeira parte da implementação: Armazenamento de dados na memória

Na seção .data os elementos das matrizes A e B já estão previamente definidos na memória. Para as matrizes que vão ser calculadas durante a atividade, foi reservado um espaço na memória para elas com .space. Como as matrizes possuem 9 elementos e cada .word ocupa 4 bytes foi armazenado um espaço de 36 bytes. Além das matrizes também existem as variáveis que vão pedir e armazenar o nome do arquivo .txt, que vai conter o resultado de ($A.B^T$).

(b) Segunda parte da implementação: A função main

Na função main ocorre a execução principal do programa. Inicialmente, são realizadas as configurações necessárias para os cálculos: o registrador `$t0` é carregado com o valor 3, representando a ordem das matrizes. Em seguida, os registradores de argumento `$a0`, `$a1`, `$a2` e `$a3` são utilizados como ponteiros para as variáveis em memória, utilizando a instrução la (load address). Após isso, o programa realiza as chamadas para as funções implementadas por meio da instrução jal (jump and link).

Antes de iniciarmos o desenvolvimento das funções, o programa abre um arquivo para a escrita, salvando seu identificador (file descriptor) no registrador `$s7`. Em seguida, chama uma função responsável por escrever o conteúdo no arquivo, e após a escrita, fecha o arquivo. Por fim, o programa é encerrado.

(c) Terceira parte da implementação: Implementação das funções dos cálculos

- `PROC_NOME` – função para pedir o nome do arquivo.

  Para solicitar que o usuário digite o nome do arquivo, a função utiliza syscall 4, que imprime uma string, carregando o endereço da mensagem em `$a0`. Em seguida, syscall 8 é usada para ler a string digitada, armazenando o resultado no espaço reservado 'nome' e definindo o tamanho máximo através `$a1`. Ao final, jr `$ra` retorna à função chamadora (main) para garantir que o fluxo do programa continue corretamente.

- `PROC_TRANS` – função folha (não chama outra função).

Essa função realiza a transposição da matriz B, armazenando o resultado em outra matriz. Para preservar o estado do programa, ela inicialmente aloca espaço na pilha e salva os registradores `$ra` e `$s0-$s6`. Os registradores `$s0` e `$s1` são usados como contadores de linhas e colunas, enquanto `$t1-$t3` e `$t8-$t9` calculam os deslocamentos em memória para acessar e armazenar cada elemento corretamente. O registrador `$a1` aponta para a matriz original (B) e `$a2` para a matriz transposta. A função utiliza loops aninhados para percorrer todas as posições da matriz, a matriz B sendo percorrida por linhas e a transposta por colunas, carregando elementos de B com lw e os armazenando na transposta com sw. Ao final, os registradores salvos são restaurados da pilha e jr `$ra` retorna à função chamadora PROC_MUL.

```
#------------------------------ TRANSPOSTA ------------------------------------

PROC_TRANS:

        addi    $sp, $sp, -32           # aloca espaço para 8 registradores na pilha
        sw      $ra, 28($sp)            # salva o endereço de retorno na pilha
        sw      $s0, 24($sp)            # salva o registrador $s0 na pilha
        sw      $s1, 20($sp)            # salva o registrador $s1 na pilha
        sw      $s2, 16($sp)            # salva o registrador $s2 na pilha
        sw      $s3, 12($sp)            # salva o registrador $s3 na pilha
        sw      $s4, 8($sp)             # salva o registrador $s4 na pilha
        sw      $s5, 4($sp)             # salva o registrador $s5 na pilha
        sw      $s6, 0($sp)             # salva o registrador $s6 na pilha
```

Figure 4: Implementação da pilha

- `PROC_MUL` – função não folha (chama `PROC_TRANS`).

  Essa função reliza a multiplicação da matriz A pela transposta da matriz B, armazenando o resultado em uma matriz resultado. Inicialmente, aloca espaço na pilha e salva os registradores `$ra` e `$s0-$s7` para preservar o estado do programa. O registrador `$s0` conta as linhas e `$s1` as colunas da matriz resultante, enquanto `$s2` é usado como contador do loop interno para percorrer os elementos de cada linha e coluna. Os registradores `$t1-$t3` e `$t8-$t9` calculam os deslocamentos em memória para acessar os elementos de A, da transposta de B e armazenar na matriz resultado. `$t4` e `$t5` carregam os elementos a serem multiplicados, `$t7` armazena a multiplicação e `$t6` acumula os resultados para cada posição da matriz final. Ao final, os registradores salvos são restaurados da pilha e jr `$ra` retorna à função chamadora.

(d) Quarta parte da implementação: Implementação da rotina escreve

Essa função é responsável por percorrer a matriz resultado e gravar seus elementos em um arquivo texto. Para isso, para cada elemento ela chama a sub-rotina converte_numero,

que transforma o valor numérico em string e escreve no arquivo. Entre os elementos de uma mesma linha e é adicionado um espaço e ao final de cada linha (menos a última), é escrito um caractere de nova linha. Para isso, a função utiliza loops aninhados de linhas e colunas, manipulando os endereços em memória para acessar os valores. A pilha é usada no início para salvar os registradores importantes e restaurá-los no fim.

Quantidade de linhas

Assim como no último laboratório, verificamos a quantidade de linhas em basic e source, recebendo os seguintes valores:

Exercício	**Linhas Basic**	**Linhas Source**
01	215	196

Table 1: Quantidade de Linhas por Programa

Percebemos que ainda há mais linhas em basic do que em source, o que indica que pseudo-instruções ainda estão sendo utilizadas, mas, também é possível perceber que, proporcionalmente, a diferença entre a quantidade de linhas em basic e em source é muito menor que no primeiro laboratório, indicando uma redução significativa no uso de pseudo-instruções.

4 Materiais e métodos

Para o desenvolvimento dessa atividade foi utilizada a linguagem `Assembly` e a IDE `MARS`, na qual foi realizada toda a escrita, execução e depuração de todos os programas implementados. Além disso, foram utilizados editores de texto para conferir se a matriz resultante estava sendo armazenada corretamente.

5 Resultados

Os resultados obtidos mostraram o funcionamento correto de todas as etapas propostas. A transposição da matriz foi realizada com sucesso, assim como a multiplicação, garantindo a consistência dos cálculos. As funções foram criadas e chamadas de forma adequada, com correto uso da pilha para salvar e restaurar registradores, preservando o

estado do programa. A manipulação da memória permitiu acessar e armazenar os elementos necessários em cada operação, e ao final a escrita no arquivo .txt foi concluída, registrando a matriz resultante.

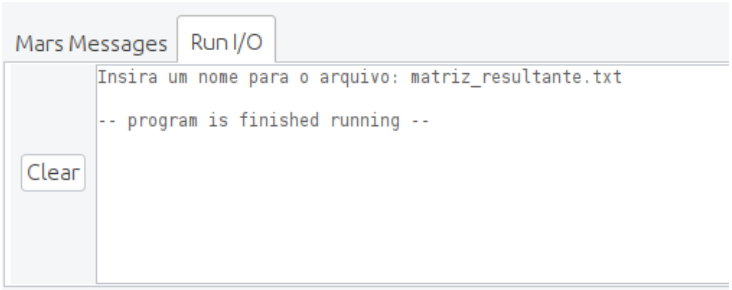

Figure 5: Terminal do Mars

Figure 6: Arquivo nos arquivos

Figure 7: Arquivo aberto

6 Dificuldades

Na transposta e na multiplicação de matrizes, o maior problema encontrado foi calcular o deslocamento de endereço para acessar cada elemento na memória. Para algumas partes, o offset deveria ser por linhas e, em algumas, por colunas. Nosso primeiro problema foi que o deslocamento por colunas estava errado na multiplicação, pois estavamos usando o contador errado, então a multiplicação estava sendo feita na ordem errada. Após identificar esse problema e corrigí-lo, não tivemos mais nenhum problema com a lógica da transposta e da multiplicação.

O que realmente foi a nossa maior dificuldade foi a parte da escrita da matriz resultante no arquivo.

Na nossa primeira tentativa, os arquivos estavam saindo em binário, pois a conversão para string não estava sendo realizada nem minimamente corretamente. Após refazer a função que convertia os números, ela ainda não estava montando as strings corretamente, o que levava a geração de arquivos em branco. Então, refizemos a conversão do zero, a testando em um arquivo diferente e, finalmente, conseguimos que essa parte funcionasse perfeitamente.

Após isso, notamos outros dois problemas: não estávamos conseguindo chamar uma função dentro da outra (todas as funções eram folha, pois foi mais simples desenvolver o programa assim inicialmente) e o programa não estava se encerrando sozinho. Após muita análise e pesquisa, descobrimos que funções em MIPS precisam preservar registradores e, como isso não estava sendo feito diretamente, eles estavam sendo sobrescritos, fazendo com que o código não funcionasse corretamente. Nossa solução para fazer isso foi criar pilhas em cada função, que salvassem o endereço de retorno ($s0) e outros registradores importantes. Essas pilhas salvavam os valores no começo da função e os desempilhavam no final das funções, garantindo que os valores fossem preservados e não se perdessem.

7 Conclusão

Apesar de o laboratório 1 ter sido excelente para a compreensão de diversos conceitos, essa atividade foi muito boa para que pudessemos nos aprofundar em outras partes de Assembly. Alguns procedimentos serem folha e outros não serem foi uma ótima maneira

de entender melhor como funciona o fluxo de programa quando se trata de Assembly. Mexer com pilhas também mostrou a importância de ter cuidado com a preservação e restauração do conteúdo dos registradores, nos permitindo conhecer um uso excelente de pilhas nessa linguagem.

Além disso, ele permite uma reflexão muito interessante sobre reusabilidade de código. Há diversas partes similares no código, como os offset ou até mesmo as pilhas que, talvez, poderiam ser transformados em procedimentos, em projetos futuros, onde já tivermos mais familiarização com a linguagem.

Outra lição muito importante que pode ser tirada dessa atividade é a importância de documentar e comentar muito bem códigos em Assembly. Como muitos registradores são utilizados, muitas vezes o mesmo para coisas diferentes, é essencial que o código esteja muito bem explicado, para que outras pessoas possam compreender o que está sendo feito e para que nós não nos perdessemos na lógica e cometessemos algum erro de sobrescrição. Os comentários são especialmente importantes nesse caso pois como estávamos trabalhando em duas no código, era muito importante que o que já havia sido feito fosse muito bem explicado, para que uma de nós pudesse continuar o trabalho de onde a outra havia parado.