**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS – CAMPUS I**

ENGENHARIA DE COMPUTAÇÃO

ARQUITETURA DE COMPUTADORES

Erick Matheus Lopes Pacheco – 18711630

Hiago Silva Fernandes – 18726455

Marcos Antônio Vasconcellos Junior – 18720920

Victor Reis – 18726471

**MIPS: GERENCIADOR DE DESPESAS PESSOAIS**

DOCUMENTAÇÃO, DETALHAMENTO E RESULTADOS DO PROJETO

CAMPINAS

Agosto 2019

SUMÁRIO

[1. INTRODUÇÃO 3](#_Toc19002845)

[2. ESPECIFICAÇÃO 4](#_Toc19002846)

[2.1. DETALHES DO PROJETO 4](#_Toc19002847)

[2.1.1. ESTRUTURAÇÃO DAS FUNÇÕES 4](#_Toc19002848)

[2.1.2. DEMAIS INFORMAÇÕES 6](#_Toc19002849)

[2.2. DETALHES DE IMPLEMENTAÇÃO 7](#_Toc19002850)

[2.2.1. VARIÁVEIS UTILIZADAS 8](#_Toc19002851)

[2.2.2. FUNÇÕES 11](#_Toc19002852)

[3. RESULTADOS 24](#_Toc19002853)

[3.1. TESTES 24](#_Toc19002854)

[3.2. RESULTADOS E DISCUSSÃO 28](#_Toc19002855)

[4. BIBLIOGRAFIA 29](#_Toc19002856)

# INTRODUÇÃO

Dá-se a proposta de projeto pelo professor: desenvolver um programa em linguagem Assembly utilizando a arquitetura MIPS cujas funções permitam o gerenciamento de gastos pessoais.

Citaremos algumas dificuldades inerentes a execução do trabalho nos capítulos posteriores e mostraremos todo o contexto da aplicação, bem como, seus modelos, especificações, bugs, e funcionalidades pertinentes e proveitosas no cotidiano do usuário.

O propósito do projeto consiste em aprimorar nossas habilidades individuais e de trabalho em grupo colocando em prática tudo aquilo quanto aprendemos nas aulas teóricas de Arquitetura de Computadores, visando também o desenvolvimento do aluno no que diz respeito ao autodidatismo, devido as pesquisas e estudos que tivemos que realizar para aprendermos a manipular e a entender a arquitetura trabalhada.

O grupo organizou-se de maneira a todos trabalharem juntos, sem que houvesse falhas de comunicação ou conflito de atividades.

Para isso, utilizamos duas ferramentas: a primeira delas, o “Discord”, software desktop e mobile que permite a comunicação VOIP de todos os integrantes do grupo, além do compartilhamento de imagens, de tela, de mensagens e arquivos. A segunda ferramenta utilizada foi o software de desenvolvimento “Atom”, que com o plugin nativo “Teletype” permite que venhamos programar juntos em tempo real no mesmo arquivo.

Foram traçadas as regras de negócios na proposta do professor a serem executadas, a fim de estabelecer um modelo de software condizente com a solicitação do cliente, no caso, o Sr. Edmar. Estabelecidas as regras, delimitamos o escopo de trabalho de cada componente da equipe e o escopo do projeto.

Pudemos observar a variação no desempenho do programa através das diferentes implementações dos integrantes da equipe.

Por fim, começamos o desenvolvimento do projeto utilizando a plataforma em Java “MARS”, nele conseguimos eficientizar os testes e depurações com o debug interno.

# ESPECIFICAÇÃO

Houve entre o grupo a discussão sobre a expressa necessidade de seguirmos coordenadamente a mesma linha de raciocínio durante todo o projeto.

Tendo isto firmado entre as partes, decidimos por utilizar majoritariamente as “I-format Instructions” ao discorrer da programação, evitando também o uso de pseudo-instruções, embora algumas tenham sido empregadas.

Definidas as variáveis principais que seriam utilizadas no projeto - quais especificaremos melhor nos detalhes de implementação - encontramos posteriormente a necessidade de criarmos variáveis auxiliares a fim de simplificar a lógica que implementaríamos.

## DETALHES DO PROJETO

Aqui explicaremos como elaboramos as funções e a forma como elas executam. Em detalhes de implementação estaremos discorrendo toda a parte técnica em torno delas.

### ESTRUTURAÇÃO DAS FUNÇÕES

São as funções do programa:

#### REGISTRAR DESPESA

Guardamos em um registrador o contador do nosso vetor principal, e então utilizamos os comandos de leitura com seus respectivos códigos para ler o ID, a categoria, o valor da despesa e a data em que foi feita.

Salvamos cada uma dessas leituras - salvas em um registrador - em nosso vetor, sempre caminhando o respectivo número de bytes de cada campo após a leitura e incrementando o nosso contador que guarda o tamanho do nosso vetor. Por fim, é perguntado se o usuário deseja continuar registrando despesas, a resposta é guardada num registrador e é feito um jump com base no valor guardado.

#### LISTAR DESPESAS

Dentro de um looping, escrevemos cada word (4 bytes) de cada campo do nosso vetor em registradores e apresentamos na tela conforme caminhamos, até à medida que o nosso contador (lê "i") for igual ou maior que nosso tamanho do vetor (lê-se "n"), quando então paramos. É verificado também no início da função se o vetor está vazio, caso esteja, não listamos nada.

#### EXCLUIR DESPESA

Mostradas na tela as informações de inserção, é pedido o ID da despesa que será excluída. Este, por sua vez, é guardado num registrador. Entramos então num looping onde varremos nosso vetor em busca do ID, uma vez encontrado, toda a despesa é excluída do vetor e realizamos aquilo que chamamos de “tapa buraco”, isto é, preencher o espaço vazio deixado pela despesa apagada movendo a última despesa do vetor no lugar.

#### EXIBIR GASTO MENSAL

Utilizando um vetor auxiliar, inicializamos ele com a gravação do primeiro elemento de nosso vetor principal.

Após isso, entramos em um looping que nos permite comparar os elementos (lê-se dados da data das despesas) do nosso vetor principal com o que está presente no vetor auxiliar. Caso as datas (mês e ano) sejam iguais, fazemos a soma do valor da despesa e guardamos nesta mesma posição do vetor auxiliar, caso contrário nós colocamos essa despesa na próxima posição.

Sucessivamente nós repetimos isto até que nosso vetor principal seja todo percorrido.

#### EXIBIR GASTO POR CATEGORIA

A mesma lógica da função anterior se aplica aqui, com a diferença que compararemos se as categorias são iguais ao invés da data. Outro diferencial está na implementação de uma ordenação por ordem alfabética das categorias.

#### EXIBIR RANKING DE DESPESAS

Mesmo esboço da função de exibir gasto por categoria, com o pequeno diferencial que ao invés de ordenarmos por ordem alfabética, ordenamos em ordem decrescente de valor. As despesas mais custosas estarão no início do vetor, sendo apresentadas primeiro.

### DEMAIS INFORMAÇÕES

É válido ressaltar que iniciamos o projeto utilizando números inteiros ao invés de ponto flutuante como requisitado pelo professor. Visamos a implementação posterior do seu uso, porém até o então momento isso não é uma realidade no projeto.

Também devemos considerar que não estamos certo quanto ao funcionamento do projeto na plataforma Linux, visto que ele foi inicialmente projetado e testado em dispositivos Windows.

## DETALHES DE IMPLEMENTAÇÃO

Aqui detalharemos cada ponto importante da implementação, a começar pela declaração das variáveis utilizadas, bem como o uso de registradores em cada uma das funções.

Nosso vetor é constituído de maneira que cada despesa ocupa 36 bytes na posição de memória, dos quais estão na seguinte ordem e tamanho:

* 4 bytes (uma word), reservado para o ID da despesa;
* 16 bytes (quatro words), reservados para a string de categoria da despesa;
* 4 bytes (uma word), reservados para o valor da despesa;
* 4 bytes (uma word), reservados para o dia da despesa;
* 4 bytes (uma word), reservados para o mês da despesa;
* 4 bytes (uma word), reservados para o ano da despesa.

Totalizando, então, 36 bytes.

### VARIÁVEIS UTILIZADAS

*Trecho de código:*



Seguiremos com a explicação e justificativa de cada variável, ponderando suas utilizações e mostrando trechos de códigos onde estão presentes.

#### STRING1 E STRING2

Variáveis auxiliares do tamanho reservado (16 bytes) para a string de categoria do nosso vetor, utilizadas para fazer as trocas entre o vetor principal e o vetor auxiliar. Implementadas nas funções 5 e 6, para realizar a gravação da categoria byte a byte (utilizando os comandos **lb** e **sb**) e a comparação (**strcmp**) delas, de modo a averiguar se as categorias são iguais para, portanto, somar as despesas de mesma categoria. Exemplo de implementação:



Mais a frente, na explicação das funções, daremos o contexto em que está implementado e qual o uso dos registradores.

#### VETOR

Nosso vetor principal, utilizado em todo programa, responsável por guardar todas as despesas registradas. Com espaço de memória reservado de 400 bytes.

#### VETOR\_F4 E VET\_F6

Vetores auxiliares que conterão os dados de despesas ordenados acordo com as funções onde são implementados, na função 4 (gastos por mês e ano) e na função 5 e 6 respectivamente. Utilizou-se dois vetores auxiliares ao invés de um só para evitar possíveis conflitos, como também para melhor distinção dos membros do grupo.

Ambos tiveram 400 bytes de memória reservados, correspondendo ao vetor principal.

#### CONTADOR

Utilizada para realizar a contagem de bytes preenchidos em nosso vetor principal, a cada nova despesa adicionada, espera-se que essa variável contenha 36 bytes a mais.

Inicializamos esta variável contadora contendo o valor “-4” referente a primeira casa nosso vetor. Se considerarmos que o primeiro dado cadastrado tem 4 bytes (o ID) e deve estar no índice de **vetor[0]**, precisamos inicializar o nosso contador não com 0, mas com -4, de modo que caminhar corretamente com nosso vetor considerando o looping que temos.

Para isso, salvamos nosso contador no registrador **$s0**, e quando realizamos a primeira leitura, salvamos em outro registrador e somamos 4 ao **$s0**, passando ele de -4 para 0, permitindo que quando fizermos outra inserção de dados e dermos o jump para a leitura, nosso vetor esteja corretamente no byte 32, onde somará 4 no contador (guardado para manipulação em **$s0**) e gravará o ID da próxima despesa onde, de fato, ela deve começar, **36 posições após a anterior**.

Trecho de código que menciono:



Onde **$s1** é o registrador que contém o dado lido, sendo salvo na posição “**vet[$s0]**”, logo acima desse trecho de código, temos a label responsável pela repetição do looping. Para fins de orientação, este trecho trata-se da função de registro de despesas.

#### CONT\_VETF4 E CONT\_VETF6

Variáveis contadoras para os vetores **vetor\_f4** e **vet\_f6**, respectivamente. Válido ressaltar que, diferentemente do contador do vetor principal, estes contadores não contam bytes, mas sim, número de despesas registradas.

#### DEMAIS VARIÁVEIS

As demais variáveis, **table**, **new\_line**, **barra**, **opcaodedia**, **opcaodecategoria**, **opcaodevalor**, **opcaodeid**, **continuardigitando**, **opcaodeidexclusao**, **mês\_ano**, **menu** e **digitar** são utilizadas apenas para fim de impressão de instruções para o usuário na tela.

### FUNÇÕES

Detalharemos a seguir as partes mais relevantes das funções.

#### BUBBLE SORT

Como método de ordenação para as funções 5 e 6 (ordenar por ordem alfabética e por valor, respectivamente) decidimos utilizar o mais popular algoritmo de ordenação de ordem de complexidade , tanto pela implementação mais simplificada quanto pela eficácia em cumprir a função que desejávamos.

Utilizamos, porém, dois bubblesorts. Ambos funcionam essencialmente da mesma forma, com o diferencial de que um foi projetado para trabalhar com strings (ide função 5, ordenação por ordem alfabética) e o outro foi projetado para trabalhar com inteiros, cujas diferenças consistem também no tamanho de espaço em bytes reservado.

Colocaremos aqui apenas o bubble de complexidade um pouco maior, utilizado na função 5 para comparação de caracteres. Neste primeiro trecho definimos os contadores em registradores:



O registrador **$t8** armazena o tamanho do nosso vetor auxiliar, o **$s5** será nosso **k**, o **$s0** nosso **j** e o **$s1** nosso **j+1**, enquanto nosso **$t5** será um contador responsável por contar até 16 (tamanho da nossa string categoria).

Aqui é onde começa nosso processo de percorrer nosso vetor auxiliar e comparar os caracteres byte a byte:



O processo de comparação ocorre no **for2**, nosso vetor **vet\_f6** é comparado de modo que se **vet\_f6[0] > vet\_f6[1]** (lê-se índice 0 como **$s0** e índice 1 como **$s1**), damos um jump para a label de **troca**. Caso contrário, incrementamos os contadores e verificamos se devemos continuar no looping de dentro (**for2**), senão saímos dele.

Processo de troca:



Pós troca, apenas realizamos a verificação de contadores, a fim de saber quando parar cada um dos nossos loopings.



Caso tenhamos chegado ao fim de nosso vetor auxiliar, simplesmente realizamos um **jr $ra** que retornará ao endereço da próxima linha de onde o comando **jal** foi executado.

#### DETALHES REGISTRAR DESPESA



Trecho de código se repete de maneira semelhante para a leitura dos demais itens da despesa. Printamos uma de nossas strings e guardamos o valor lido salvo em **$s1** em nosso **vetor[$s0]**.

#### DETALHES LISTAR DESPESAS



Define os contadores, printa o ID e a mesma lógica se repete para os demais itens da despesa, por fim, chegamos ao trecho que verifica se estamos no final do vetor e, caso esteja, finaliza com um jump para o início:



#### DETALHES EXCLUIR DESPESA

Código que realiza exclusão, procura pelo ID lido em **$a0** comparando com o salvo do vetor em **$t0**:



#### DETALHES EXIBIR GASTO MENSAL



A seguir definimos os contadores para o vetor auxiliar e principal, como também para os laços de repetição. Fazemos então a primeira gravação de dados no vetor auxiliar, para inicializá-lo, utilizamos os registradores **$t3** para o mês, **$t4** para o ano, puxando esses dados de **vetor[0].** Após isso, vamos para a próxima casa do nosso vetor principal incrementado 28 posições em seu contador através do comando **addi $s0, $zero, 28**.

Agora, iniciamos nosso laço de repetição:

****

Chegado ao looping, copiamos novamente o mês e o ano de nosso **vetor** e damos um jump para o looping interno, neste looping ajustamos contadores, copiamos a data armazenada em nosso vetor auxiliar e comparamos com o mês com o do vetor original, caso iguais damos um jump para comparar o ano, caso contrário pulamos para outra condição.



Aqui está um trecho importante, caso a data não seja igual nós caímos nesta label **continua\_and** e realizamos o processo de copiar os dados do nosso vetor principal para a próxima casa do nosso vetor auxiliar (**vetor\_f4[$s0]**), tendo em vista que a ideia é a seguinte:

* Se datas iguais = soma o valor da despesa **i** do vetor principal na casa atual do vetor auxiliar
* Se datas diferentes = não soma valor da despesa, vai para a próxima casa do vetor auxiliar e copia os dados da despesa.

Após copiarmos, voltamos para o topo do laço de repetição da nossa função para continuar percorrendo o vetor.



Por fim, quando temos ambas datas iguais verificadas pouco após o início da função, faz-se a soma da despesa no vetor auxiliar, quando não, insere o dado novo na próxima posição desse vetor.



Aqui fazemos o reset de registradores e evitamos possíveis erros de endereçamento, inclusive zerando byte a byte o nosso vetor auxiliar.

Prosseguimos então com o próximo trecho de código:



Aqui realizamos o print do nosso vetor auxiliar com as despesas todas reunidas por data.

#### DETALHES EXIBIR GASTO POR CATEGORIA E RANKING DE DESPESAS

Mesmas bases da função 4, apenas com sutis diferenças de implementação. A exemplo os registradores que são inicializados no início da função.

A despeito da função 5, outro diferencial presente é a forma como realizamos a contagem, de modo que venhamos armazenar a categoria inteira do vetor, e não somente a data e a despesa como na função anterior.

Também fazemos o uso da função **strcmp**, que podemos visualizar aqui:



Entramos na função, copiamos em **$t6** e **$t4** nossos caracteres gravados anteriormente na função nos registradores **$s2** e **$s3**, e nele gravamos nossa string inteira das categorias que queremos verificar se são iguais que foi salva nas variáveis **string1** e **string2**, entramos então no loop, copiamos byte a byte nossas strings para os registradores **$t2** e **$t3** e comparamos para verificar se são iguais. Se forem diferentes, damos um jump para **cmpne**, caso seja igual a zero (ou seja, nulo, chegou no **\0** da string), damos outro jump, e caso seja igual, continuamos o looping para verificar os próximos caracteres da sequência.

Vejamos a continuação a seguir:



Caso os valores armazenados (caracteres) nos bytes não sejam iguais, caímos na label **cmpne**, que passa de volta os bytes armazenados nos registradores temporários **$t4** e **$t6** para **$s2** e **$s3** que usamos ao longo da função e seta em **$t5** o valor zero para usarmos como flag, sinalizando que não são iguais, portanto, ordenando de acordo na função. Em caso de toda a string ser igual, fazemos a mesma cópia dos registradores temporários e setamos nossa flag em **$t5** como 1.

Quanto a função 6, idêntica a função 5, apenas com pequenas diferenças no bubble sort e de soma de contador.

# RESULTADOS

Desenvolvido o programa, vamos aos testes e resultados:

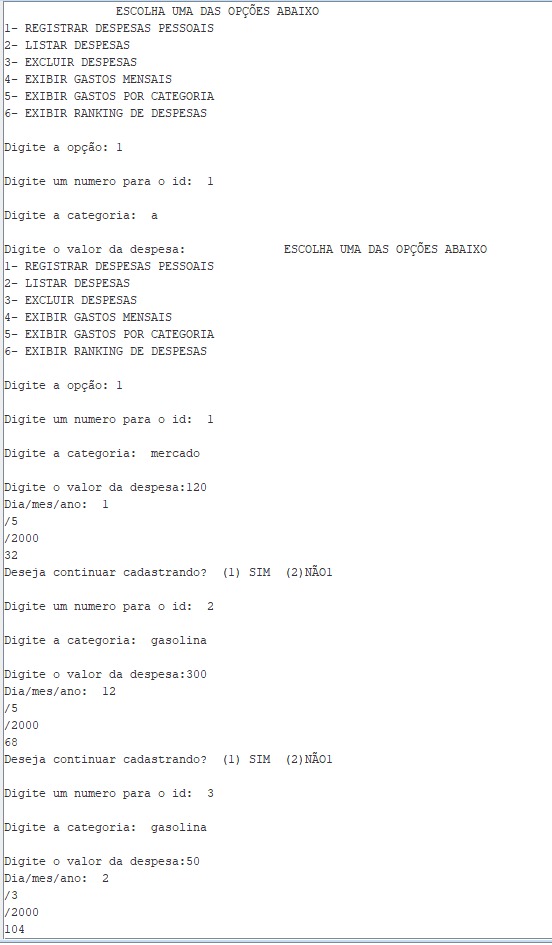
## TESTES

Foi utilizada uma variedade de testes diferentes a fim de abranger a maior possibilidade de erros possíveis. Para tais testes foi feita a inserção de despesas, sendo preenchido campos como o ID, a categoria, o valor e a data.

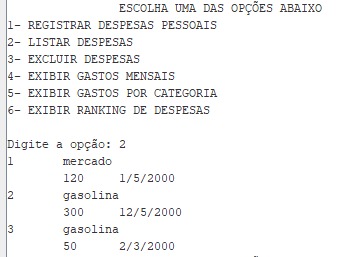
Feito essas inserções para testes, foi então dado um passo à frente para o teste do restante das funções, e para a comprovação de que a função de inserção estava funcionando, fez-se a impressão dos dados.

Para a comprovação dos dados obtidos pelo programa, foi feito um teste de mesa para comprovar a efetividade dos valores obtidos no programa. Após esse teste foi possível ver que o programa realmente estava com o funcionamento todo em ordem.

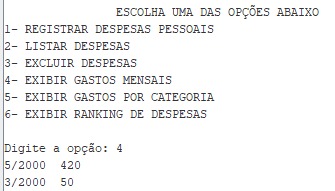
Imagens dos testes:



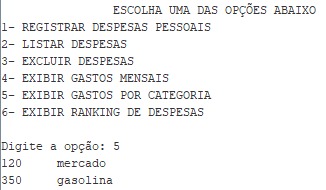
Função 1 – Registro de despesas



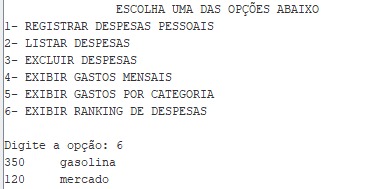
Função 2 - Listagem de despesas



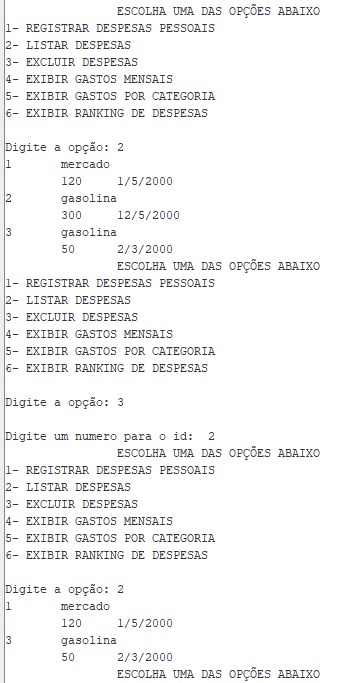
Função 4 - Gastos por data



Função 5 - Gastos por ordem alfabética



Função 6 - Ranking de despesas



Função 3 - Exclusão de despesa

## RESULTADOS E DISCUSSÃO

Ao longo do projeto foram obtidos erros por conta da falta de atenção do grupo em relação aos registradores, onde foi perdido muito tempo em coisas bobas como exemplo na falta de incrementar ou até mesmo em confusões causando troca de registradores em lugares indevidos.

Por conta destes pequenos problemas e erros de lógica, implementamos todo o programa conforme requisitado com apenas uma pequena exceção: não utilizamos ponto flutuante. O desafio encontrou-se em conciliar o tempo que restava entre terminar pequenos detalhes e ter que aprender a utilizar o coprocessador para float, como também modificar as funções para funciona dessa maneira. Sendo assim o que optamos por fazer foi a correção dos pequenos detalhes, deixando de lado tal implementação.

Ao fim do projeto foi obtido êxito em relação ao não funcionamento do programa e um grande salto de aprendizagem da linguagem aos integrantes do grupo, como também da lógica de funcionamento de baixo nível, manipulação de memória, compreensão do sistema de registradores e de como são formados os programas que conhecemos em mais alto nível.

# BIBLIOGRAFIA

Autor desconhecido. “**INTRODUÇÃO AO ALGORITMO DE ORDENAÇÃO BUBBLE SORT**”. Dev Fúria. Disponível em <http://www.devfuria.com.br/logica-de-programacao/introducao-ao-algoritmo-de-ordenacao-bubble-s–ort/>. Acesso em 29 de agosto de 2019 às 20:00.

VOLLMAR, KEN. “**HANDOUT: ARRAYS, LW AND SW INSTRUCTION**”. Missouri State University. Disponível em <https://courses.missouristate.edu/KenVollmar/s09-csc285/MIPS%20arrays%20lw%20sw.pdf/>. Acesso em 16 de agosto de 2019 às 18:15.

C. GATTO, ELAINE. “**ARMAZENANDO UM VALOR EM ARRAY NO MIPS**”. Embarcados. Disponível em <https://www.embarcados.com.br/armazenando-um-valor-em-array-no-mips/>. Acesso em 16 de agosto de 2019 às 18:20.

C. GATTO, ELAINE. “**EXECUTANDO UM ARRAY NO MARS PARA MIPS**”. Embarcados. Disponível em < https://www.embarcados.com.br/executando-um-array-no-mars-para-mips/>. Acesso em 20 de agosto de 2019 às 21:00.