

**UNIVERSIDADE DO VALE DO ITAJAÍ**

ESCOLA DO MAR, CIÊNCIA E TECNOLOGIA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CC4189 – ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Professores,

Cesar Albanes Zeferino, Dr.

Eduardo Alves da Silva, MSc.

**PROGRAMAÇÃO EM LINGUAGEM DE MONTAGEM DO MIPS**

Acadêmicos,

Israel Efraim de Oliveira,

José Carlos Zancanaro.

Itajaí (SC), 17 de maio de 2018.

**PROGRAMAÇÃO EM LINGUAGEM DE MONTAGEM DO MIPS**

Israel Efraim de Oliveira

José Carlos Zancanaro

Maio / 2018

Professores: Cesar Albanes Zeferino, Eduardo Alves da Silva.

Curso: Bacharelado em Ciência da Computação.

Palavras-chave: Linguagem de montagem, assembly, MIPS, processador tipo RISC, programação de procedimentos.

Número de páginas: 11.

*“I think it's a new feature. Don't tell anyone it was an accident.”*

*“Acredito que seja um novo recurso. Não conte a ninguém que foi um acidente.”*

*Larry Wall*

# Introdução

Este relatório aborda a resolução de um problema proposto na disciplina de Arquitetura e Organização de Computadores do curso de bacharelado em Ciência da Computação. O problema consiste na simulação de cadastramento e gerenciamento de séries, similar ao Netflix, que possibilita a ordenação de forma crescente ou decrescente das informações fornecidas pelo usuário. O trabalho foi desenvolvido utilizando a linguagem de montagem do processador MIPS (*Microprocessor without interlocked pipeline stages*), baseado na arquitetura RISC. Para a resolução do problema, utilizamos a interface de desenvolvimento integrado MARS (MIPS assembly and runtime simulator).

O objetivo central do problema foi exercitar o uso de procedimentos e pilha em linguagem de montagem. Além disto, outro desafio apresentado foi a manipulação de vetores correlacionados.

O relatório consistirá na construção e explicação da resolução abordada pelos autores, evidenciando o código fonte sempre que for necessário.

# Construção

Para o desenvolvimento da solução, procuramos distribuir as responsabilidades de cada etapa do processo em procedimentos que pudessem ser chamados através do método principal (main) no momento em que fosse mais conveniente.

Vale ressaltar que, quando tratamos com programação de procedimentos, devemos conservar os valores dos registradores salvos (*$s*), sendo esta medida estabelecida por convenção. Para tal resolução, utilizamos a pilha para armazenar os valores dos registradores salvos, eventualmente utilizando-a para passar parâmetros e conservar o valor do registrador de retorno de uma função específica (em instruções do tipo *jr $ra*).

O primeiro passo da solução é solicitar o tamanho real de elementos que nossos vetores de séries terão, respeitando a restrição de tamanho proposta (de 2 à 6 elemenos). Deste modo, abstraímos esta função para que fosse possível utiliza-la em outras partes do programa que também necessitam de validação de entrada. Para realizar este tipo de validação, precisamos apenas de um laço de repetição do tipo *do-while* que só será finalizado quando o usuário inserir um valor válido, como a seguir:

    # Conferir se o valor inserido é válido

    slt $s5, $s4, $s2               # $s5 = $s4 < $s2 ? 1 : 0

    sgt $s6, $s4, $s3               # $s6 = $s4 > $s3 ? 1 : 0

    or  $s5, $s5, $s6               # $s5 = $s5 or $s6

    # Pular para o retorno se o valor é válido

    beq $s5, $zero, solicitar\_valor\_exclusivo\_retornar

No exemplo acima, *$s4* armazena o valor digitado pelo usuário, *$s2* contém o tamanho mínimo e *$s3* possue o tamanho máximo. Existem outros registradores que são usados apenas para definir estados intermediários. A operação de validação deve apenas terminar se e somente se nenhum dos casos (instrução de *slt* e *sgt*) retornar o valor numérico um, pois, no caso deste valor, significa que o usuário excedeu alguma restrição estabelecida.

Em seguida, faz-se necessário realizar a leitura de informações para agregar aos vetores, tais como, o código das séries, a quantidade de pessoas que assistiram apenas um episódio, a quantidade de pessoas que assistiram todos os episódios e a avaliação de cada série dada por uma quantidade de estrelas.

Todas estas operações são semelhantes na questão de exibição de textos, leitura e interação com o usuário, tendo mudanças apenas na lógica de que valor é restringido.

Para o código das séries, restringimos valores negativos e repetidos, isto é, cada série possui um código natural único de identificação. Para realizar tais validações, utilizamos as seguintes estratégias:

    # Chamar função para verificar se valores inseridos são exclusivos

    add $a0, $zero, $s0             # $a0 = $so (endereço-base do vetor)

    add $a1, $zero, $s2             # $a1 = $s2 (i)

    add $a2, $zero, $s4             # $s4 = $v0 (alvo)

    jal existe\_em\_vetor             # existe\_em\_vetor($a0, $a1, $a2)

    add $s5, $zero, $v0             # $s5 = $v0 (resultado de existe\_em\_vetor)

    slt $s6, $s4, $zero             # $s6 = $s4 < 0 ? 1 : 0

    or  $s7, $s5, $s6               # $s7 = $s5 or $s6

    beq $s7, $zero, preencher\_codigo\_series\_do\_while\_fim

A função *existe\_em\_vetor* itera um vetor passado por parâmetro através do registrador *$a0* e retorna o valor um na primeira ocorrência do valor encontrada; se nenhuma é encontrada, ela retornará o valor 0 e sabemos que este código será único, mas só poderá ser inserido se passar a validação de sinal. Ao final do preenchimento deste vetor, o conteúdo é copiado para um vetor auxiliar que assistirá na sincronização de vetores.

No vetor de quantidades de pessoas que assistiram um episódio (referenciado como *AS1* futuramente), a única restrição necessária é que o valor precisa ser zero ou qualquer outro positivo, mas já tratamos deste tipo de validação com o vetor anterior, portamento, não é necessário evidenciar o processo novamente.

Para o vetor de quantidade de pessoas que assistiram todos os episódios (tratado como *AST* futuramente), o valor de cada elemento será restringido pela quantidade inserida no vetor AS1 e pela naturalidade do número. Desta forma, precisamos comparar as entradas de mesmo índice de AS1 com o tamanho proposto para AST, como a seguir:

    lw  $s6, 0($s5)              # $s6 = $s5 (VETOR\_SERIES\_AS1[i])

    sgt $s6, $v0, $s6               # $s6 = $v0 > $s6 ? 1 : 0

    slt $s7, $v0, $zero             # $s6 = $v0 < 0 ? 1 : 0

    or  $s7, $s6, $s7               # $s6 = $s6 or $s7

    beq $s7, $zero, preencher\_assistiram\_todos\_do\_while\_fim

Ressaltamos que $s5 contém o endereço-base do vetor AS1 somado ao deslocamento do índice i por 2 (isto é, i \* 22).

No último vetor de informações, de avaliação, precisamos validar valores entre 1 e 10, que pode ser também resolvida através de implementação de testes lógicos utilizando instruções do tipo *slt* e *sgt*, conforme o código abaixo:

    slti    $s5, $s4, 1             # $s5 = $s4 < 0 ? 1 : 0

    addi    $s6, $zero, 10          # $s6 = 10 (máximo de estrelas)

    sgt     $s6, $s4, $s6           # $s6 = $s4 > 10 ? 1 : 0

    or  $s7, $s5, $s6               # $s7 = $s5 or $s6

    beq $s7, $zero, preencher\_estrelas\_do\_while\_fim

Posteriormente, foi necessário solicitar qual será o vetor a ser ordenado (AS1 / AST / EST), que nada mais é do que validar uma entrada novamente e, desta vez, podemos utilizar a função previamente definida como *solicitar\_valor\_exclusivo*:

    # Solicitar valor entre 1 e 3

    la  $a0, TXT\_LER\_SERIE\_ORDENACAO\_OPCAO

    la  $a1, TXT\_LER\_SERIE\_VALOR\_INVALIDO

    la  $a2, OPCAO\_ORDENACAO\_MINIMA         # $a2 = &OPCAO\_ORDENACAO\_MINIMA

    lw  $a2, 0 ($a2)                        # $a2 = \*OPCAO\_ORDENACAO\_MINIMA (1)

    la  $a3, OPCAO\_ORDENACAO\_MAXIMA         # $a3 = &OPCAO\_ORDENACAO\_MAXIMA

    lw  $a3, 0 ($a3)                        # $a3 = \*OPCAO\_ORDENACAO\_MAXIMA (3)

    jal solicitar\_valor\_exclusivo

O tipo de ordenação (crescente ou decrescente) também utiliza da função de valor exclusivo para validar entradas entre 0 (crescente) e 1 (decrescente).

    # Solicitar tipo de ordenação (Crescente / Decrescente)

    la  $a0, TXT\_MSG\_ORDEM\_BUBBLESORT       # $a0 = TXT\_MSG\_ORDEM\_BUBBLESORT

    la  $a1, TXT\_LER\_TAMANHO\_INVALIDO       # $a1 = TXT\_LER\_TAMANHO\_INVALIDO

    la  $a2, ORDEM\_CRESCENTE                # $a2 = &ORDEM\_CRESCENTE

    lw  $a2, 0 ($a2)                        # $a2 = \*ORDEM\_CRESCENTE

    la  $a3, ORDEM\_DECRESCENTE              # $a3 = &ORDEM\_CRESCENTE

    lw  $a3, 0 ($a3)                        # $a3 = \*ORDEM\_CRESCENTE

    jal solicitar\_valor\_exclusivo

Antes de realizar o procedimento de bubblesort e a sincronização dos vetores, foi necessário preparar alguns registradores para que fosse possível identificar qual é o vetor ordenado e quais devem ser sincronizados, pois, dado que existem três possibilidades de escolha, temos que lidar com três casos onde o vetor ordenado e os sincronizados são diferentes. Para resolver tal problema, realizamos um *switch case* que verifica o caso correspondente à escolha do usuário e prepara os registradores para serem utilizados pelos procedimentos.

Considerando que a escolha do usuário sobre qual vetor será ordenado foi salva em *$s3*, teremos três possíveis escolhas dentro do *switch case*: ordenar o vetor AS1 e sincronizar AST e EST; ordenar o vetor AST e sincronizar o AS1 e EST; ordenar o vetor EST e ordenar o AS1 e AST.

main\_escolha\_caso\_1:

    addi    $s3, $s1, -1

    bne $s3, $zero, main\_escolha\_caso\_2

    la  $s3, TXT\_MSG\_CLASSIFICACAO\_AS1     # $s3 = TXT\_MSG\_CLASSIFICACAO\_AS1

    la  $s5, VETOR\_SERIES\_AS1            # $s5 = &VETOR\_SERIES\_AS1

    la  $s6, VETOR\_SERIES\_AST            # $s6 = &VETOR\_SERIES\_AST

    la  $s7, VETOR\_SERIES\_EST            # $s7 = &VETOR\_SERIES\_EST

    j   main\_escolha\_sair                  # break

main\_escolha\_caso\_2:

    addi    $s3, $s1, -2

    bne $s3, $zero, main\_escolha\_caso\_3

    la  $s3, TXT\_MSG\_CLASSIFICACAO\_AST     # $s3 = TXT\_MSG\_CLASSIFICACAO\_AST

    la  $s5, VETOR\_SERIES\_AST              # $s5 = &VETOR\_SERIES\_AST

    la  $s6, VETOR\_SERIES\_AS1              # $s6 = &VETOR\_SERIES\_AS1

    la  $s7, VETOR\_SERIES\_EST              # $s7 = &VETOR\_SERIES\_EST

    j   main\_escolha\_sair            # break

main\_escolha\_caso\_3:

    addi    $s3, $s1, -3

    bne $s3, $zero, main\_escolha\_sair

    la  $s3, TXT\_MSG\_CLASSIFICACAO\_EST     # $s3 = TXT\_MSG\_CLASSIFICACAO\_EST

    la  $s5, VETOR\_SERIES\_EST            # $s5 = &VETOR\_SERIES\_EST

    la  $s6, VETOR\_SERIES\_AS1           # $s6 = &VETOR\_SERIES\_AS1

    la  $s7, VETOR\_SERIES\_AST           # $s7 = &VETOR\_SERIES\_AST

main\_escolha\_sair:

Desta forma, independentemente da escolha do usuário, só precisaremos realizar uma chamada, pois os registradores já foram configurados de maneira geral e uniforme.

Após ter configurado, podemos finalmente chamar a função de ordenação, passando o vetor de códigos, o vetor para ordenação escolhido, o tamanho do vetor e método de ordenação:

    # Chamar bubblesort

    add $a0, $s4, $zero             # $a0 = $s4 (vetor de códigos)

    add $a1, $s5, $zero             # $a1 = $s5 (vetor para ser ordenado)

    add $a2, $s0, $zero             # $a2 = $s0 (tamanho real do vetor)

    add $a3, $s2, $zero             # $a3 = $s2 (tipo de ordenação)

    jal bubblesort\_series           # bubblesort\_series ($a0, $a1, $a2, $a3)

Para o procedimento bubblesort, configuramos dois laços de repetição aninhados: um externo limitado pelo tamanho dos vetores; e um interno, limitado pela quantidade de elementos iteráveis (tamanho – 1) e pela quantidade de vezes que o primeiro laço já executou. O ponto interessante deste procedimento, neste relatório, é a possibilidade da ordenação nos dois sentidos (crescente, decrescente), implementada pela lógica abaixo:

Seja O o valor do tipo de ordenação (0 ou 1),

Seja Z o resultado de A > B e Y o resultado de A < B,

Z \* !O + Y \* O

    # Conferir se a ordenação é crescente e o par precisa ser ordenado

    sgt $t5, $t3, $t4               # $t5 = $t3 > $t4 ? 1 : 0

    nor $t6, $s3, $zero             # $t6 = !$s3

    and $t5, $t6, $t5               # $t5 = $t6 and $t5

    # Conferir se a ordenação é decrescente e o par precisa ser ordenado

    slt $t6, $t3, $t4               # $t6 = $t3 < $t4 ? 1 : 0

    and $t6, $t6, $s3               # $t6 = $t6 and $s3

    or  $t5, $t5, $t6               # $t5 = $t5 or $t6

    beq $t5, $zero, bubblesort\_series\_n\_ordena  # Se 0, não ordenar

A lógica para verificar se um par precisa ser ordenado é dada pela relação de maioridade do par em J e J+1 junto ao tipo de ordenação que está sendo realizado. Se o valor numérico do tipo de ordenação for 0 (a ordenação é crescente), a operação NOR com o tipo retornará 1 e o par só será trocado se estiver em ordem decrescente (A > B). Caso contrário e o valor numérico do tipo for 0 (a ordenação é decrescente), o par é trocado somente quando estiver em ordem crescente (A < B).

Por fim, realiza-se a sincronização dos vetores através de outros vetores auxiliares que guardam as informações originais. Define-se um loop externo (i) e outro interno (j) e procura-se a entrada i do vetor de código originais no vetor de códigos ordenados em j, quando o código é encontrado, copia-se os valores dos vetores auxiliares em i para os vetores sendo sincronizados em j. Como a seguir:

    sub    $t1, $s5, $s6               # $t1 = $s5 - $s6

    beq $t1, $zero, sincronizar\_series\_nao\_troca

    # Criar ponteiro para vetor\_CS modificado em j

    sll $t1, $s6, 2             # $t1 = $s6 \* 4

    add $t2, $t1, $s1               # $t2 = $t1 + $s1 (&vetor\_cs\_modificado)

    # Conferir se os códigos do original em i e do modificado em j são iguais

    lw  $t2, 0 ($t2)                # $t2 = \*$t2

    lw  $t3, 0 ($t0)                # $t3 = \*$t0

    sub $t2, $t2, $t3               # $t2 = $t2 - $t3

    bne $t2, $zero, sincronizar\_series\_nao\_troca

Dada a equivalência dos códigos, a troca é realizada e o loop interior é finalizado:

    # Sincronizar primeiro vetor

    la  $t2, VETOR\_SERIES\_COPIA\_UM  # $t2 = &VETOR\_SERIES\_COPIA\_UM

    add $t2, $t2, $s7               # $t2 = $t2 + $s7

    lw  $t2, 0 ($t2)                # $t2 = \*$

    add $t3, $s3, $t1               # $t3 = $s3 + $t1

    sw  $t2, 0 ($t3)                # \*$t3 = $t2

    # Sincronizar segundo vetor

    la  $t2, VETOR\_SERIES\_COPIA\_DOIS# $t2 = &VETOR\_SERIES\_COPIA\_DOIS

    add $t2, $t2, $s7               # $t2 = $t2 + $t1

    lw  $t2, 0 ($t2)                # $t2 = \*$t2

    add $t3, $s4, $t1               # $t3 = $s3 + $t1

    sw  $t2, 0 ($t3)                # \*$t3 = $t2

    j   sincronizar\_series\_for\_int\_fim      # Break (sair do loop interior)

Após a sincronização dos vetores, o novo vetor ordenado é mostrado na tela para o usuário informando a categoria de ordenação escolhida.

# Experimento e análise de instruções

## Ordenação crescente

Insira a quantidade de séries (min. = 2, máx. = 6): 6

Insira o código das séries:

CS[0] : 17

CS[1] : 22

CS[2] : 13

CS[3] : 53

CS[4] : 1

CS[5] : 110

Insira a quantidade de pessoas que assistiram apenas um episódio:

AS1[0] : 52

AS1[1] : 329

AS1[2] : 471

AS1[3] : 628

AS1[4] : 1450

AS1[5] : 2131

Insira a quantidade de pessoas que assistiram todos os episódios:

AST[0] : 27

AST[1] : 220

AST[2] : 302

AST[3] : 467

AST[4] : 1108

AST[5] : 1401

Insira a avaliação das séries (min = 1, máx = 10):

EST[0] : 7

EST[1] : 8

EST[2] : 9

EST[3] : 6

EST[4] : 10

EST[5] : 7

Selecione o indicador a ser utilizado para a classificação das séries:

1. Assistiram apenas um episódio.

2. Assistiram todos os episódios.

3. Número de estrelas.

Digite a opção desejada para ordenação (1..3) : 1

Digite 0 para ordenação crescente ou 1 para decrescente: 1

Ordenando posições de forma decrescente.

Classificação das séries após ordenação: Assistiram apenas um episódio

1o lugar - CS = 110, Assistiram apenas um episódio = 2131

2o lugar - CS = 1, Assistiram apenas um episódio = 1450

3o lugar - CS = 53, Assistiram apenas um episódio = 628

4o lugar - CS = 13, Assistiram apenas um episódio = 471

5o lugar - CS = 22, Assistiram apenas um episódio = 329

6o lugar - CS = 17, Assistiram apenas um episódio = 52

-- program is finished running --

O quadro estatístico resultante foi:

|  |  |  |
| --- | --- | --- |
| **Classe** | **Nº de execuções** | **Percentual** |
| Aritmética e Lógica (ALU) | 1333 | 52% |
| Desvio incondicional (Jump) | 153 | 06% |
| Desvio condicional (Branch) | 231 | 09% |
| Acesso à memória (Memory) | 602 | 23% |
| Outras | 253 | 10% |
| **Total** | 2572 | 100% |

## Ordenação decrescente

Insira a quantidade de séries (min. = 2, máx. = 6): 6

Insira o código das séries:

CS[0] : 17

CS[1] : 22

CS[2] : 13

CS[3] : 53

CS[4] : 1

CS[5] : 110

Insira a quantidade de pessoas que assistiram apenas um episódio:

AS1[0] : 2131

AS1[1] : 1450

AS1[2] : 628

AS1[3] : 471

AS1[4] : 329

AS1[5] : 52

Insira a quantidade de pessoas que assistiram todos os episódios:

AST[0] : 1401

AST[1] : 1108

AST[2] : 467

AST[3] : 302

AST[4] : 220

AST[5] : 27

Insira a avaliação das séries (min = 1, máx = 10):

EST[0] : 7

EST[1] : 10

EST[2] : 6

EST[3] : 9

EST[4] : 8

EST[5] : 7

Selecione o indicador a ser utilizado para a classificação das séries:

1. Assistiram apenas um episódio.

2. Assistiram todos os episódios.

3. Número de estrelas.

Digite a opção desejada para ordenação (1..3) : 2

Digite 0 para ordenação crescente ou 1 para decrescente: 0

Ordenando posições de forma crescente.

Classificação das séries após ordenação: Assistiram todos os episódios

1o lugar - CS = 110, Assistiram todos os episódios = 27

2o lugar - CS = 1, Assistiram todos os episódios = 220

3o lugar - CS = 53, Assistiram todos os episódios = 302

4o lugar - CS = 13, Assistiram todos os episódios = 467

5o lugar - CS = 22, Assistiram todos os episódios = 1108

6o lugar - CS = 17, Assistiram todos os episódios = 1401

-- program is finished running –

O quadro estatístico resultante foi:

|  |  |  |
| --- | --- | --- |
| **Classe** | **Nº de execuções** | **Percentual** |
| Aritmética e Lógica (ALU) | 1334 | 51% |
| Desvio incondicional (Jump) | 154 | 06% |
| Desvio condicional (Branch) | 232 | 10% |
| Acesso à memória (Memory) | 602 | 23% |
| Outras | 253 | 10% |
| **Total** | 2575 | 100% |

# Conclusão

Através de todo o escopo do programa, utilizamos diversos procedimentos, principalmente na parte de preenchimento de vetores e exibição, que eram semelhantes, isto é, grande parte do corpo era composta por características similares, mas diferiam apenas em algum teste lógico. Para resolução disto, poderíamos ter implementado um tipo de callback, ou seja, passar uma função por parâmetro que seria responsável por realizar aquele teste lógico ou chamar alguma outra função que assimilaria o trabalho e, portanto, teríamos apenas um método mais efetivo e flexível em relação à reusabilidade de código e não teríamos apenas algumas repetições de trabalho.

Em relação aos testes e quadros estatísticos, já esperava-se o grande uso de instruções de acesso à memória e aritmética/lógica, devido a quantidade de iterações e restrições que existem dentro do problema (validação de códigos de séries, tamanhos, bubblesort, sincronizar as séries após ordenação, etc.).

De modo geral, a construção desta resolução foi satisfatória, pois utilizamos diversos conceitos ensinados durante o decorrer da disciplina e tivemos a oportunidade de implementar diversos tipos de fluxo de informação que não implementamos no trabalho anterior (switch-case, por exemplo), além da prática do uso da pilha e da programação de procedimentos.