UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES (EACH)

SISTEMAS DE INFORMAÇÃO

ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES I

Bruno Friedrich Raquel

Matheus Silva Lopes da Costa

Organização e Arquitetura de Computadores: Relatório sobre a alocação, população e impressão de uma matriz seguindo a arquitetura MIPS

Computer Organization and Architecture: Report on allocation, population and printing of a matrix following the MIPS architecture

São Paulo

2023

Organização e Arquitetura de Computadores: Relatório sobre a alocação, população e impressão de uma matriz seguindo a arquitetura MIPS

Trabalho da disciplina Organização e Arquitetura de Computadores. O relatório consiste em sintetizar o que é a arquitetura MIPS e explicar um exercício-problema que será desenvolvido em C e Assembly MIPS.

Área de Concentração: Organização e Arquitetura de Computadores.

Professora: Gisele Craveiro

São Paulo

2023

RESUMO

Bruno Friedrich Raquel e Matheus Silva Lopes da Costa. **Organização e Arquitetura de Computadores:** Relatório sobre a alocação, população e impressão de uma matriz seguindo a arquitetura MIPS. Universidade de São Paulo, São Paulo, 2023.

Esse trabalho inicia-se com uma introdução à organização e arquitetura MIPS, arquitetura de conjunto de instruções comumente utilizada em processadores de computadores e dispositivos embutidos e serão discutidos conceitos fundamentais dessa arquitetura. A seguir ocorre a descrição do problema proposto e apresenta o código de alto nível da solução. Nessa seção, é explicado o contexto e os requisitos do problema, juntamente com a estratégia de solução adotada pela equipe. O código de alto nível fornecido é uma representação em C, facilitando a compreensão da lógica da solução.

Esse trabalho segue com a apresentação do código desenvolvido em Assembly. Aqui, a equipe traduz o código de alto nível anteriormente descrito para a linguagem Assembly MIPS. O código em linguagem de montagem é mais próximo do nível de linguagem de máquina, ele usa mnemônicos que são convertidos pelo Assembler para o código de máquina da arquitetura MIPS. Sobre o código em linguagem de montagem são discutidas as estruturas de controle, como laços e condicionais, bem como as instruções específicas utilizadas para implementar a solução proposta. Por fim, é apresentada uma explicação detalhada das instruções utilizadas no código desenvolvido pela equipe. Cada instrução é descrita em termos de sua função, sintaxe e comportamento. Também são abordados aspectos relacionados aos modos de endereçamento, uso de registradores e manipulação de memória.

Temas-chave: Organização e Arquitetura de Computadores, Matriz, Assembly, MIPS

SUMÁRIO

Seção 1 - Organização e Arquitetura MIPS

MIPS (Microprocessor without Interlocked Pipelined Stages) é uma arquitetura de processador desenvolvida pela MIPS Technologies em 1991. A arquitetura é do tipo RISC (Reduced Instruction Set Computer), cada instrução ocupa o espaço de 32 bits e as operações convencionais ocorrem apenas entre os registradores, de modo que para operar um dado contido na memória é necessário primeiro armazená-lo em um registrador, o que faz com que o conjunto de instruções do tipo registrador-registrador ou load-store. Existem aspectos fundamentais do conjunto de instruções, cuja compreensão é essencial para o uso eficaz do MIPS, são eles a sintaxe e função dos componentes principais da linguagem e o funcionamento dos recursos de armazenamento.

1. **Sintaxe básica:** um programa funcional em MIPS, ou seja um programa que passa pelo montador sem apresentar erros, exige que o programador use adequadamente as **diretivas,** os **tipos de endereçamento** e obedeça os **formatos de instrução** relacionados a cada operação.
    1.1. **Diretivas:** São trechos do código que servem para organizar o armazenamento dos dados e instruções estaticamente, ou seja durante a fase de montagem, elas formam um conjunto definido e todas são iniciadas por um ponto, uma das principais diretiva é a .text ela indica ao montador que as linhas seguintes são instruções e que portanto devem ser armazenadas no seguimento da memória principal dedicado a instruções, onde o program counter irá se deslocar. Outra diretiva importante é a .data que indica que as linhas seguintes são dados rotulados e devem ser armazenados no segmento de dados, sobre os dados rotulados há ainda diretivas para especificar o tipo de dados para garantir o armazenamento correto.
    1.2. **Tipos de endereçamento:** De acordo com a classificação de tipos de endereçamento de Stallings, estão disponíveis as formas de endereçamento do tipo por registrador, indireto por registrador, imediato, por indexação com offset e por pilha.
        1.2.1. **Endereçamento imediato:** o valor de um dos operandos é especificado diretamente na instrução.
        Ex: addi $s1,$s0, 5

O valor 5 é adicionado ao valor de $s0 e o resultado dessa operação é armazenado em $s1.

Endereçamento por registrador: a operação ocorre com os valores armazenados nos registradores passados na instrução.

Ex**:** add $t0,$t0,$t1

O valor de $t1 é somado com o de $t0 e o resultado é armazenado em $t0.

1.2.2. **Endereçamento Indireto por registrador:** O endereço de uma posição da memória pode ser salvo em um registrador, que armazenará o endereço de memória que será usado pelas instruções de load/store para obter o endereço onde devem escrever ou ler o operando.

Ex: sw $t0,($t1)

Assumindo que $t1 armazene um endereço de memória, essa operação atribui a $t0 o valor armazenado na memória no endereço contido em $t1.

1.2.3. **Endereçamento por indexação com offset:** Nesse tipo de endereçamento um valor em número de bytes é somado a um endereço de memória base que já está armazenado em um registrador e o valor armazenado nesse novo endereço é usado na operação.

Ex: lw $t0,16($t1)

Atribui a $t0 o inteiro contido no endereço $t1 + 16.

1.2.4. **Endereçamento por pilha:** O endereçamento por pilha utiliza a indexação com offset para armazenar ou ler dados em um segmento especial da memória designado com pilha de execução, cujo fim é armazenado em um registrador especial o $sp, a pilha pode ser usada em diversos contextos, mas seu emprego ocorre principalmente quando é necessário armazenar um número de dados maior que o número de registradores disponíveis.

1.2.5. Ex: sub $sp,$sp,4

lw $t0,$sp

Um espaço de armazenamento é aberto decrementando o apontador e em seguida o valor de $t0 é armazenado na posição da memória apontada por $sp.

1.3. **Formatos de instrução:** Como já exposto, cada instrução ocupa o tamanho fixo de 32 bits, no entanto no nível médio, identifica-se instruções com diferentes formatos devido a especificidades das operações, que por vezes dispensam a presença explícita de algum operando. Existem três tipos de formatos de instrução na linguagem de montagem: R-R,R-I,R-J.

1.3.1. **R-R:** Esse tipo de instrução usa três registradores, dois como operandos e um de destino onde o resultado da operação será armazenado.

1.3.2. **R-I:** Uma instrução desse tipo usa com um dos operandos um valor escrito diretamente na instrução, e armazena o resultado em um registrador de destino, o montador permite criar identificadores para representar endereços, que serão convertidos em operandos imediatos.

1.3.3. **R-J:** Formato usado por algumas instruções de salto que de maneira geral carregam ou adicionam valores no registrador usado para busca de instrução o $pc.

Algumas instruções de branch condicional podem usar uma combinação do formato R-I com o formato R-J, há um operador imediato na condição e um identificador para o salto.

Outro destaque importante é que algumas instruções não exigem a passagem de um ou outro operando explicitamente porque já possuem registradores predeterminados para aquele tipo de operação.

2. Operações principais: As instruções principais são as que executam as operações aritméticas, de acesso à memória e de saltos.

2.1. Operações aritméticas: São as operações aritméticas básicas, usam três operandos, sendo que um deles pode ou não ser imediato, as operações de divisão e multiplicação utilizam registradores específicos para auxiliar com questões de overflow e resto.

2.2. **Operações de acesso a memória:** Como já exposto, o MIPS realiza apenas operações de transferência na memória principal, para isso existem instruções que acessam um dado endereço e copiam em um registrador uma certa quantidade de bytes que caracterizam um tipo especificado.

2.3. **Operações de salto:** Operações usadas para controlar o fluxo de execução do programa, ocorrem por meio de desvios que na prática são modificações no valor de $pc, e permitem implementar condicionais, laços e funções.

3. **Usos da memória:**

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Imagem A.6.1 do livro Organização e Design de Computadores - MIPS Edition - 6ª Edição

Seção 2 - O problema e código em alto nível da solução

Agora que analisamos o que é a arquitetura MIPS vamos iniciar a compreensão do problema que devemos resolver. A dupla, de número 5, recebeu o problema 13 da lista de exercícios passada pela professora. O exercício em questão está disponibilizado a seguir:

13.

(a) Escreva uma função que lê, linha a linha, uma matriz real Amxn

(b) Escreva uma função que imprime uma matriz real Amxn

Dessa forma, como evidenciado o dever da dupla é alocar memória suficiente para suportar uma matriz de valores reais, chamada de A. Essa matriz tem dimensão mxn, ou seja, ‘m’ linhas e ‘n’ colunas. Além disso, após ter a matriz criada é necessário que ela seja preenchida, com a inserção de valores em cada posição e a impressão deles.

Agora que temos nosso problema em mente, antes de solucioná-lo a partir do desenvolvimento em Assembly, vamos primeiro ver a solução em uma linguagem de alto nível. Utilizaremos, como linguagem de alto nível, a linguagem C.

O motivo vem a seguir: a linguagem Assembly MIPS permite uma programação mais próxima do nível de linguagem de máquina, utilizando instruções específicas para manipular registradores, memória e controlar o fluxo do programa. Assim sendo, por ser uma linguagem que se aproxima da linguagem de máquina, possui um nível maior de complexidade, quando comparado às linguagens de alto nível, por exemplo. Como curiosidade, existem diversas utilidades incluindo alta eficiência para programação de sistemas embarcados e um maior controle sobre os recursos do processador MIPS, resultando em um desempenho otimizado.

A linguagem C é uma linguagem de programação de alto nível amplamente utilizada, conhecida por sua eficiência e portabilidade. Com sua sintaxe baseada em blocos de código, tipos de dados variados e recursos de controle de fluxo, como estruturas condicionais e de repetição, ela permite a criação de programas poderosos e flexíveis. Além disso, sua capacidade de manipulação direta de memória e o uso de ponteiros a tornam uma opção popular para o desenvolvimento de sistemas de software de baixo nível.

```c
#include <stdio.h>
#include <stdlib.h>

void imprimeLinha(float* Matriz, int colunas){
  for (int i = 0; i<colunas; i++){
    printf(" %.1lf", Matriz[i]);
  }
  printf("\n");
}

int main(void) {
  int m, n;

  printf("Digite quantas linhas a matriz deve ter\n");
  scanf (" %i", &m);
  printf("Digite quantas colunas a matriz deve ter\n");
  scanf (" %i", &n);
  // aloca espaço dinamicamente para a matriz
  float** Matriz = (float**)malloc(m * sizeof(float*));
  for (int i = 0; i < m; i++) {
    Matriz[i] = (float*)malloc(n * sizeof(float));
  }
  // recebe os valores da matriz
  for (int i = 0; i<m; i++){
    for (int j = 0; j<n; j++){
      scanf(" %lf", &Matriz[i][j]);
    }
  }
  // chama a impressão linha a linha da matriz
  for (int i = 0; i<m; i++){
    imprimeLinha (Matriz[i], n);
  }
```

```
  return 0;
}
```

Código em C que resolve o exercício proposto

Agora que temos o nosso código em alto nível, vamos entender como o algoritmo soluciona o problema e como podemos pensar nessa solução em uma linguagem em mais baixo nível. Para isso, iremos destrinchar parte por parte nosso código.

1. As bibliotecas stdio.h e stdlib.h são incluídas para permitir operações de entrada/saída e alocação dinâmica de memória, respectivamente.
2. A função imprimeLinha é definida. Essa função recebe um ponteiro para uma linha da matriz e o número de colunas da matriz. Em seguida, ela itera sobre os elementos da linha e os imprime usando printf. Cada elemento é formatado com uma casa decimal usando o especificador de formato %.1lf. Por fim, a função imprime uma nova linha.
3. A função main é definida como a função principal do programa.
4. As variáveis m e n são declaradas para armazenar o número de linhas e colunas da matriz, respectivamente.
5. O programa solicita ao usuário que digite o número de linhas da matriz e armazena o valor em ‘m’ usando scanf.
6. O programa solicita ao usuário que digite o número de colunas da matriz e armazena o valor em n usando scanf.
7. O programa aloca espaço dinamicamente para a matriz usando malloc. Primeiro, ele aloca espaço para m ponteiros para float usando malloc(m * sizeof(float*)). Em seguida, para cada ponteiro, ele aloca espaço para n elementos float usando malloc(n * sizeof(float)). Isso cria uma matriz mxn na memória.
8. O programa entra em um loop duplo para ler os valores da matriz. Ele itera sobre cada linha e coluna da matriz usando os loops for. Dentro do loop, ele usa scanf para ler um valor float e armazená-lo na posição apropriada da matriz Matriz[i][j].
9. Após ler todos os valores da matriz, o programa chama a função imprimeLinha para imprimir a matriz linha por linha. Ele itera sobre cada linha da matriz usando

um loop for e passa o endereço da linha Matriz[i] e o número de colunas n para a função imprimeLinha.

10. O programa retorna 0 para indicar que a execução foi concluída com sucesso

Seção 3 - Código em Assembly desenvolvido

A partir do entendimento do problema e da comparação com o código em alto nível, arquitetamos nossa solução em Assembly. Para essa solução utilizamos pelo menos 2 duas rotinas, passagem de parâmetro via pilha e loops, como os requisitos indicam.

```
#programa que recebe o número de linhas, o número de colunas e uma
matriz de float em seguida imprime a matriz

.data
    initialProgramText: .asciiz "Programa que lê e imprime uma
matriz real"
    getRowsNumberText: .asciiz "Insira o número de linhas: "
    getColsNumberText: .asciiz "Insira o número de colunas: "
    insertMatrixText: .asciiz "Insira a matriz:"
    newLineChar: .asciiz  "\n"
    spaceChar: .byte ' '
.text
    main:
      la $a0,initialProgramText #carrega o endereço de
initialProgramText em $a0
      li $v0,4 #carrega o valor 4 em $v0
      syscall # chamada do sistema para imprimir initialProgramText

     jal newLine # chama subrotina para imprimir quebra de linha

     la $a0,getRowsNumberText #carrega o endereço de
getRowsNumberText em $a0
      li $v0,4 #carrega o valor 4 em $v0 código de impressão de
string
      syscall # chamada do sistema para imprimir getRowsNumberText

     jal newLine
```

```
      li $v0,5 #carrega o valor 5 em $v0 código de leitura inteiro
      syscall # chamada do  sistema para ler um int e armazenar em
$v0

     add $s0,$v0,$zero #salvando número de linhas

      la $a0,getColsNumberText #carrega o endereço de
getRowsNumberText em $a0
      li $v0,4 #carrega o valor 4 em $v0 código de impressão de
string
      syscall # chamada do sistema para imprimir getRowsNumberText

      jal newLine # chama subrotina para imprimir quebra de linha

      li $v0,5 #carrega o valor 5 em $v0 código de leitura inteiro
      syscall # chamada do  sistema para ler um int e armazenar em
$v0

      add $s1,$v0,$zero #salvando o número de colunas

      mul $t0,$s0,$s1 #calculando numero de floats que serão
recebidos

      mul $t1,$t0,4 #calculando numero de bytes para armazenamento
e salvando em $t3

      add $a0,$zero,$t1 #carregando em $a0 o número de bytes
necessários
      li $v0,9 # código para solicitar reserva de memória heap
      syscall  # chamada do sistema para alocar memória
```

```
      div $t1,$s0 #calculando tamanho em bytes de uma linha
      mflo $s3 #salvando tamanho em bytes de uma linha

      add $s2,$zero,$v0 # salvando o endereço inicial da sequência
de bytes

      add $t1,$zero,$t0 # iniciando contador de elementos

      add $t0,$zero,$v0 # iniciando o iterador para armazenagem na
pilha

      la $a0,insertMatrixText #carrega o endereço de
getRowsNumberText em $a0
      li $v0,4 #carrega o valor 4 em $v0 código de impressão de
string
      syscall # chamada do sistema para imprimir getRowsNumberText

      jal newLine

    readLoop:
      li $v0,6 # carregando em $v0 código de leitura de float
      syscall # chamada do sistema para ler um float
      s.s $f0,($t0) # armazenando o valor lido na "matriz"
      addiu $t0,$t0,4 # apontando para o espaço de armazenamento
seguinte
      subi $t1,$t1,1 # decrementa o contador de elementos
      bne $t1,$zero,readLoop # carrega o endereço de readLoop em
$pc se $t1 for diferente de $zero

    jal newLine

    add $t2,$zero,$s2 # iniciando iterador de impressão
```

```
    printLineLoop:
      beq $t2,$t0,exit_sucess
      add $sp,$sp,-4
      sw $t2,($sp)
      jal imprimeLinha
      jal newLine
      add $t2,$t2,$s3
      add $sp,$sp,4
      j printLineLoop

    imprimeLinha:
      lw $t1,($sp) # armazena o valor de ($sp) em $t1 para imprimir
o valor
     add $t3,$t1,$s3
      printColLoop:
       beq $t3,$t1,return
       l.s $f12,($t1)
       li $v0,2 # armazena em $v0 código para impressão de float
       syscall # chamada do sistema para impressão de float
        la $a0,spaceChar # carrega o endereço de spaceChar é
armazenado em $a0
        li $v0,4 # armazena o código de impressão de char em $v0
        syscall # chamada do sistema para impressão de espaço
        addiu $t1,$t1,4
       j printColLoop
      return:
       jr $ra # carrega em $pc o valor de $ra

    newLine:
     la $a0, newLineChar #Carrega o endereço de newLine em $a0
     li $v0, 4 # carrega o código de impressão de caractere em $v0
```

```
    syscall # chamada do sistema para imprimir \n
    jr $ra

exit_sucess:
    li $v0,10
    syscall
```

Código em Assembly-MIPS que resolve o exercício proposto

Agora vamos ao entendimento de cada trecho desse código para, a seguir, podermos compreender as microinstruções de cada operando:

1) A seção .data, em um código em Assembly, é onde as constantes e dados estáticos são definidos. Nessa seção, podemos declarar e inicializar variáveis, strings, constantes numéricas e outras informações que serão acessadas durante a execução do programa. Nesse caso em específico, .data possui algumas strings que serão utilizadas ao longo do programa. Para facilitar seu uso, definimos essas strings nesta seção.
    a) **initialProgramText:** Essa é uma string que armazena o texto "Programa que lê e imprime uma matriz real". Essa string é usada para imprimir uma mensagem inicial ao usuário.
    b) **getRowsNumberText:** Essa é uma string que armazena o texto "Insira o número de linhas: ". Essa string é usada para solicitar ao usuário que digite o número de linhas da matriz.
    c) **getColsNumberText:** Essa é uma string que armazena o texto "Insira o número de colunas: ". Essa string é usada para solicitar ao usuário que digite o número de colunas da matriz.
    d) **insertMatrixText:** Essa é uma string que armazena o texto "Insira a matriz:". Essa string é usada para solicitar ao usuário que digite os valores da matriz.
    e) **newLineChar:** Essa é uma string que armazena o caractere de nova linha "\n". Ela é usada para imprimir uma quebra de linha no console.
    f) **spaceChar:** Essa é uma declaração de byte que armazena o caractere de espaço " ". É usado para imprimir um espaço entre os elementos da matriz.

2) A seção .text é onde o código principal do programa é escrito.
   - O rótulo main é onde a execução do programa começa.

   a) As 3 linhas a seguir possuem como objetivo imprimir a mensagem inicial do programa. Dessa forma, o endereço da String initialProgramText é carregada em $a0, o valor 4 é guardado em $v0 (código de operação para impressão de String) e, por fim, ocorre uma chamada do sistema para imprimir initialProgramText

```
la $a0,initialProgramText
li $v0,4
syscall
```

   b) No código, haverá uma série de chamadas a uma sub-rotina utilizada para imprimir uma quebra de linha. O nome dessa rotina é newLine e é chamada da seguinte forma: **“jal newLine”.** Tudo que essa função faz é carregar o endereço da String ‘\n“ em $a0 e em seguida realizar a operação de impressão. Por fim, a função realiza um jr (jump-register) para o endereço guardado em $ra (return-address)

```
newLine:
    la $a0, newLineChar
    li $v0, 4
    syscall
    jr $ra
```

   c) Após a mensagem inicial, de formas gerais, é impresso uma mensagem que solicita ao usuário a impressão do número de linhas e recebe o valor digitado por ele.

      i) A primeira parte é responsável pela impressão padrão de uma string: carrega o endereço de getRows em $a0, carrega o valor 4 em $v0 (código de impressão de string) e realiza a chamada do sistema para impressão da string. Após essa operação chama-se a sub-rotina newLine.

```
la $a0,getRowsNumberText
li $v0,4
syscall
```

ii) Para receber o valor, carrega-se 5 no registrador $v0 (código para a leitura de inteiro) e realiza uma chamada no sistema para ler o inteiro que fica armazenado em $v0. Para salvar o número de linhas, realiza-se a operação add em $s0 e a soma será do valor inteiro recebido e o número zero (guardado no registrador $zero).

```
li $v0,5
syscall
add $s0,$v0,$zero
```

d) O trecho de código carrega o endereço da string chamada getColsNumberText em $a0. Essa string solicita ao usuário que digite o número de colunas que sua matriz deve possuir. Em seguida, chama-se newLine. Depois disso, lê um número inteiro fornecido pelo usuário usando outra chamada do sistema e armazena-o em $v0. Após isso, o valor de $v0 é copiado para o registrador $s1 usando a instrução add, com $zero como segundo operando para manter o valor original. Isso serve para salvar o número de coluna em $s1.

```
la $a0,getColsNumberText
li $v0,4
syscall

jal newLine

li $v0,5
syscall

add $s1,$v0,$zero
```

e) Os próximos comandos são usados para calcular o número total de elementos na matriz (quantos floats serão recebidos), o número total de bytes necessários para armazenar a matriz e alocar memória suficiente na heap para armazenar a matriz.

i) Primeiramente, executa-se a operação de multiplicação entre os registradores $s0 e $s1 e armazena o resultado no registrador $t0. Essa operação é usada para calcular o número de floats

que serão recebidos, ou seja, o produto dos valores contidos nos registradores $s0 e $s1 será armazenado em $t0.

ii) Ocorre uma multiplicação entre o valor contido no registrador $t0 e o valor imediato 4. O resultado dessa multiplicação é armazenado no registrador $t1.Essa operação é utilizada para calcular o número de bytes necessários para armazenar os floats calculados anteriormente, considerando que cada float ocupa 4 bytes de espaço na memória, segundo especificação do MIPS.

iii) Após o cálculo do número de bytes necessários para armazenamento, esse valor é carregado em $a0. Utilizando a chamada do sistema com código 9, é solicitada a reserva de memória na heap. A quantidade de bytes especificada em $a0 é alocada na memória e o endereço base dessa região alocada é armazenado em $v0.

```
mul $t0,$s0,$s1

mul $t1,$t0,4

add $a0,$zero,$t1
li $v0,9
syscall
```

iv) Por fim, salva o endereço inicial da sequência de bytes no registrador $s2 a partir da soma de $zero com o valor armazenado em $v0.

```
add $s2,$zero,$v0
```

f) A seguir, para auxiliar na manutenção da matriz, calcula-se o tamanho, em bytes de uma linha. Essa operação é realizada a partir da divisão da quantidade de bytes (armazenada em $t1) pela quantidade de linhas ($s0). O valor é salvo em $s3 a partir da operação que pega a parte inteira.

```
div $t1,$s0
mflo $s3
```

g) No seguinte trecho de código, a primeira linha adiciona o valor do registrador $t0 ao registrador $zero e armazena o resultado em $t1, iniciando assim um contador de elementos. A seguir adiciona-se o valor do registrador $v0 ao registrador $zero e armazena o resultado em $t0, iniciando um iterador que será utilizado para armazenar elementos na pilha.

```
add $t1,$zero,$t0

add $t0,$zero,$v0
```

- Agora que alocamos memória, vamos receber os valores da matriz.

h) Para imprimir que desejamos receber floats para a matriz, colocaremos o endereço da string insertMatrixText em $a0 e carrega-se o valor 4 em $v0 que indica o código de impressão de string. Após isso chama-se o sistema para realizar a impressão e chama a rotina newLine para imprimir o '\n'.

```
la $a0,insertMatrixText
li $v0,4
syscall
```

i) Agora vamos fazer nosso loop de leitura. O nome dele é readLoop e possui a seguinte sequência lógica:
   i) Realiza a leitura de um float a partir do carregamento do valor 6 em $v0 e chama o sistema
   ii) A instrução s.s armazena o valor lido (contido em $f0) na "matriz" (ou na posição de memória) apontada pelo valor presente em $t0. Isso permite armazenar o float lido em um local específico de memória.
   iii) A seguir, a instrução 'addiu' incrementa o valor contido em $t0 em 4. Isso faz com que o registrador $t0 aponte para o próximo espaço de armazenamento na memória, avançando em intervalos de 4 bytes, considerando que cada float ocupa 4 bytes.

iv) O próximo passo é decrementar o valor do contador de elementos, representado por $t1. A operação utilizada é o 'subi' que diminui o valor imediato do conteúdo de seu registrador.

v) Por fim, a instrução bne verifica se o valor contido em $t1 é diferente de zero. Se for verdadeiro, ou seja, se $t1 não for igual a zero, o programa é redirecionado para o rótulo readLoop, reiniciando o loop de leitura. Caso contrário, se $t1 for igual a zero, o loop é encerrado e o programa continua a partir do próximo comando após o loop.

```
readLoop:
  li $v0,6
  syscall
  s.s $f0,($t0)
  addiu $t0,$t0,4
  subi $t1,$t1,1
  bne $t1,$zero,readLoop
```

- Agora que os valores foram recebidos, precisamos imprimi-los. Para isso, contaremos com a utilização de um loop que lê os valores armazenados na memória e os imprime, linha a linha.

j) Para iniciar essa parte do código, vamos definir $t2 como iterador de impressão. Para essa função, vamos copiar o valor armazenado em $s2 (endereço inicial da sequência de bytes).

```
add $t2,$zero,$s2
```

k) Agora, para iniciarmos a impressão, vamos utilizar uma rotina chamada de "printLineLoop". A responsabilidade dela é imprimir linha a linha uma matriz.

i) O loop começa com uma comparação que verifica se o registrado $t2 é igual ao valor do registrador $t0. Isso significa que $t2 percorreu todos os valores e agora aponta para a última memória da sequência de bytes (que é representada por $t0). Se o valor for igual, o programa é encerrado com a chamada de 'exit_sucess'' que, por conseguinte, carrega 10 em $v0 e chama o sistema para a encerrar a execução do programa.

```
exit_sucess:
    li $v0,10
    syscall
```

ii) Se o valor de $t2 for diferente de $t0, então inicia-se o processo de impressão de fato. Para isso, subtrai-se 4 da pilha para armazenar um valor. Nesse caso, o valor armazenado corresponde ao apontado por $t2 e é carregado a partir da instrução sw que armazena o valor na posição de memória apontada por $sp (que foi decrementado anteriormente).

iii) Após isso, realiza-se a chamada da rotina imprimeLinha e newLine que, como o próprio nome já indica, imprime uma linha inteira e imprime o '\n'. A rotina printLineLoop será explicada no próximo tópico.

iv) Depois da impressão dessa linha, o registrador $t2 é incrementado do valor do registrador $s3 que representa o tamanho em bytes de uma linha. Dessa forma, o contador da posição da memória apontada passa a verificar a próxima linha.

v) Por fim, o ponteiro de pilha é somado a 4 para desfazer o espaço reservado anteriormente. Além disso, salta incondicionalmente para o rótulo printLineLoop, reiniciando assim o loop.

```
printLineLoop:
beq $t2,$t0,exit_sucess
add $sp,$sp,-4
sw $t2,($sp)
jal imprimeLinha
jal newLine
add $t2,$t2,$s3
add $sp,$sp,4
j printLineLoop
```

l) Finalizando o passo a passo de nosso programa, vamos decompor o procedimento que 'imprimeLinha' realiza.

i) Primeiramente, o valor armazenado em $sp é carregado em $t1 para imprimir seu valor. Além disso, o valor de $s3, tamanho de bytes de uma linha, é somado a $t1 e salvo em $t3.

ii) A seguir inicia-se 'printColLoop' responsável por imprimir cada uma das colunas da linha. A condição de parada é se o valor da coluna for igual ao valor de elementos dela. Assim, se a condição de parada for verdadeira ocorre um desvio nomeado return que volta para o endereço da instrução do jump do método printLineLoop.

iii) A instrução "l.s $f12, ($t1)" carrega o valor em ponto flutuante (float) da posição de memória apontada por $t1 em $f12, que será usado para impressão. Após o carregamento do valor para impressão, chama-se o sistema a partir do código 2, salvo em $v0, interpretado como código para impressão de float.

iv) Ademais, chama-se o sistema para a impressão de uma string, com o código 4 carregado em $v0. A String a ser impressa está salva no .data e corresponde ao 'spaceChar' que é simplesmente para colocar um espaço entre os números.

v) Por fim, Incrementa o valor de $t1 em 4 para passar para a próxima coluna e realiza um jump incondicional para o marcador 'printColLoop', continuando o loop de impressão de colunas

```
imprimeLinha:
    lw $t1,($sp)
    add $t3,$t1,$s3
    printColLoop:
            beq $t3,$t1,return
            l.s $f12,($t1)
            li $v0,2
            syscall

            la $a0,spaceChar
            li $v0,4
            syscall
            addiu $t1,$t1,4
            j printColLoop
    return:
    jr $ra
```

Seção 4 - Instruções utilizadas no código:

Agora vamos a interpretação das instruções que foram utilizadas no código e suas microoperações. Para isso, vamos analisar uma tabela que possui o comando à esquerda, seus parâmetros e, em sequência, o que é executado a partir da interpretação da instrução.

Load Immediate li $registrador, constante	O comando "li" é usado para carregar um valor imediato (constante) em um registrador. O comando "li" carrega o valor imediato especificado no registrador fornecido.
Load Address la $registrador, rótulo/variável	O comando "la" traduz o rótulo ou o nome da variável em um endereço de memória e o armazena no registrador especificado.
syscall - System Call syscall Antes deve-se li $v0, código_do_sistema	O comando "syscall" é usado para fazer uma chamada ao sistema (system call). Essas chamadas permitem que o programa interaja com o ambiente do sistema operacional. O código específico da chamada do sistema é armazenado no registrador $v0.
jal - Jump and Link jal subrotina	O comando "jal" é usado para realizar um salto para uma sub-rotina (função). Ele armazena o endereço de retorno no registrador $ra (link de retorno) e salta para o rótulo especificado.
j - Jump j rótulo	O comando "j" é usado para fazer um salto incondicional para um rótulo específico. Ele simplesmente altera o valor do registrador de programa ($pc)

	para o endereço do rótulo especificado.
jr - Jump to Address in Register jr $ra	O comando "jr" é usado para fazer um salto para o endereço especificado em um registrador. Ele altera o valor do registrador de programa ($pc) para o valor armazenado no registrador especificado.
add - Addition add $destino, $fonte1, $fonte2	O comando "add" é usado para realizar uma operação de adição entre dois registradores. Ele adiciona o valor de $fonte1 ao valor de $fonte2 e armazena o resultado em $destino.
mul - Multiplication mul $destino, $fonte1, $fonte2	O comando "mul" é usado para realizar uma operação de multiplicação entre dois registradores. Ele multiplica o valor de $fonte1 pelo valor de $fonte2 e armazena o resultado em $destino.
div - Division div $dividendo, $divisor	O comando "div" é usado para realizar uma operação de divisão entre dois registradores. Ele divide o valor do registrador divisor pelo valor do registrador dividendo. O resultado da divisão é armazenado nos registradores especiais HI (parte alta) e LO (parte baixa).

mflo - Move from Low Order Register mflo $registrador	O comando "mflo" é usado para mover o resultado da operação de divisão (quociente) do registrador especial LO para um registrador especificado. Ele copia o valor armazenado em LO para o registrador especificado.
lw - Load Word lw $registrador, endereço	O comando "lw" é usado para carregar uma palavra (4 bytes) da memória para um registrador. Ele carrega o valor armazenado no endereço de memória especificado para o registrador especificado.
l.s - Load Single Precision Floating Point	O comando "l.s" é usado para carregar um valor de ponto flutuante de precisão simples (float) da memória para um registrador. Ele carrega o valor armazenado no endereço de memória especificado para o registrador especificado.
s.s - Store Single Precision Float s.s $registrador_float, endereço	O comando "s.s" é usado para armazenar um valor de ponto flutuante de precisão simples (float) de um registrador na memória. Ele armazena o valor do registrador especificado no endereço de memória especificado.
addiu - Addition Unsigned Immediat addiu $destino, $valor1, 5 :	O comando "addiu" é usado para realizar uma operação de adição de uma constante a um registrador. Ele adiciona o valor imediato especificado ao valor do registrador e armazena o resultado no mesmo registrador.

subi - Subtraction Immediate subi $destino, $valor1, 2	O comando "subi" é usado para realizar uma operação de subtração de um valor imediato (constante). Ele subtrai o valor imediato especificado do valor do registrador e armazena o resultado no mesmo registrador.
bne - Branch if not equals bne $destino, $valor_condição, rótulo	O comando "bne" é usado para realizar um desvio condicional (branch) se dois registradores não forem iguais. Ele compara os valores dos dois registradores e desvia o fluxo de execução para o rótulo especificado se os valores forem diferentes.
beq - Branch if equals beq $destino, $valor_condição, rótulo	O comando "beq" é usado para realizar um desvio condicional (branch) se dois registradores forem iguais. Ele compara os valores dos dois registradores e desvia o fluxo de execução para o rótulo especificado se os valores forem iguais.
sw - Store Word: sw $registrador, endereço	O comando "sw" é usado para armazenar uma palavra (4 bytes) de um registrador na memória. Ele armazena o valor do registrador especificado no endereço de memória especificado.

Seção 5 - Referências Bibliográficas

Organização de Computadores - Aula 06 - Conjunto de Instruções do MIPS. Disponível em: <https://www.youtube.com/watch?v=VYSy21RwNIc&ab_channel=UNIVESP>

Guia Rápido MIPS Tipos de Dados e Formatações. [s.l: s.n.]. Disponível em: <https://www.inf.ufpr.br/wagner/ci243/GuiaMIPS.pdf>

Introdução à Arquitetura de Computadores/Instruções do MIPS - Wikilivros. Disponível em: <https://pt.wikibooks.org/wiki/Introdu%C3%A7%C3%A3o_%C3%A0_Arquitetura_de_Computadores/Instru%C3%A7%C3%B5es_do_MIPS>

GATTO, E. C. Arquitetura de Conjunto de Instruções MIPS. Disponível em: <https://embarcados.com.br/arquitetura-de-conjunto-de-instrucoes-mips/>

Arquitetura MIPS. Disponível em: <https://pt.wikipedia.org/wiki/Arquitetura_MIPS>

Linguagem assembly. Disponível em: <https://pt.wikipedia.org/wiki/Linguagem_assembly>. Acesso em: 16 jul. 2023

FREEDEV. Entenda o que é Assembly. Disponível em: <https://freedev.medium.com/entenda-o-que-%C3%A9-assembly-ed64526cab49>.