Laboratório de Arquitetura e Organização de Computadores 1

Semana(s) 04 e 05

Introdução ao simulador MARS [1] [2] [3]

1 Instruções Gerais

- Ler atentamente todo o procedimento desta experiência antes de realizá-la;

2 Objetivos da Prática

- Apresentar as funcionalidades do simulador MARS;
- Apresentar a linguagem *Assembly MIPS*.

3 Materiais e Equipamentos

- Simulador MARS, disponível em http://courses.missouristate.edu/kenvollmar/mars/
- simulador online Java, disponível em https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html

4 Fundamentos teóricos

4.1 Introdução

Para controlar o *hardware* de um computador é preciso falar de sua linguagem. As palavras da linguagem de um computador são chamadas **instruções** e seu vocabulário é denominado **conjunto de instruções**. Você pode imaginar que as linguagens dos computadores são tão diversificadas quanto a dos humanos, mas na realidade, as linguagens de computadores são muito semelhantes entre si, mais parecidas com dialetos regionais do que linguagens independentes. Logo, quando você aprender uma, ficará mais fácil aprender as outras.

[1] Documento adaptado das Práticas de Laboratório dos professores: Ricardo Menotti, Maurício Figueiredo, Edilson Kato, Celso Furukawa e Luciano Neris.
[2] Elaborado em 21/09/2020: Prof. Artino Quintino.
[3] Revisado em 18/02/2021: Prof. Maurício Figueiredo.

A semelhança entre os conjuntos de instruções ocorre porque todos os computadores são construídos a partir de tecnologias de *hardware* baseadas em princípios básicos semelhantes e porque existem operações básicas que todos os computadores precisam oferecer. Além disto, todo projeto de computador tem um objetivo em comum: encontrar uma linguagem que facilite o projeto do *hardware* e do compilador enquanto maximiza o desempenho e minimiza o custo.

4.2 MIPS - *Microprocessor without Interlocked Pipeline Stages*

O processador MIPS (*Microprocessor without Interlocked Pipeline Stages*) é adotado para efeitos do desenvolvimento dos conceitos pertinentes à disciplina . É utilizado em pequenos produtos para entretenimento ao consumidor, dispositivos de redes domésticas, equipamentos de infraestrutura, *modems* LTE e aplicativos incorporados, dispositivos voltados para a Internet das Coisas (*IoT*). O conjunto de instruções correspondente compõe uma linguagem relativamente bem simples, desenvolvida pela *MIPS Technologies* na década de 1980 e adquirida pela empresa *Wave Computing* em 2018.

4.3 *Assembly*

Os primeiros programadores se comunicavam com os computadores em números binários, mas isto era tão cansativo e ineficaz que eles inventaram novas notações minimamente próximas da linguagem natural. Usando a própria máquina para ajudar a programar a si mesma, os programadores desenvolveram também programas que traduzem a notação simbólica para binário. O primeiro destes programas foi chamado de **montador (*assembler*)**. Esse programa traduz uma versão simbólica de uma instrução para uma versão binária. Por exemplo, o programador escreveria:

add A, B

e o montador traduziria essa notação como

1000110010100000

Essa instrução diz ao computador para somar dois números, A e B. O nome criado para esta linguagem simbólica é **linguagem assembly**. Já a linguagem binária que a máquina entende é a chamada **linguagem de máquina**.

4.4 Registradores

Dentro da hierarquia de memória de um computador, os registradores são as memórias contidas dentro da CPU e servem para armazenar os dados, em sua maioria, resultados das operações realizadas pela ULA – unidade lógica e aritmética.

O processador MIPS tem 32 registradores e cada um possui um tamanho de 32 bits. Na hierarquia de memória de um computador, os registradores são altamente velozes, chegando a executar uma instrução entre 1 a 2 nano segundos. Na arquitetura MIPS, podemos encontrar os registradores de uso geral (os de dados) e registradores de uso específico (de controle e de estado). Por conta de sua função prescípua, armazenamento (memória), os registradores estão envolvidos essencialmente (mesmo que implicitamente) em duas classes de operações básicas: GRAVAR (Write) e LER (Read).

Os registradores na arquitetura MIPS são precedidos por um '$' (cifrão - também citado como dólar), sendo representados por seu nome ou número. A tabela apresenta a relação de registradores da arquitetura MIPS.

Nome	Nº	Utilização
$0 ou $zero	0	Registrador com valor constante igual a zero
$at	1	Assemblador temporário – reservado ao montador
$v0-$v1	2-3	Registradores que recebem resultados de funções
$a0-$a3	4-7	Registradores para passagem de argumentos
$t0-$t7	8-15	Registradores temporários (não preservam os valores)
$s0-$s7	16-23	Registradores que preservam (salvam) os valores
$t8-$t9	24-25	Registradores temporários (não preservam os valores)
$k0-$k1	26-27	Registradores para kernel do sistema operacional
$gp	28	Registrador para ponteiro global
$sp	29	Registrador apontador para pilha
$fp	30	Registrador para apontador de frame
$ra	31	Registrador para guardar o endereço de retorno
$pc		Registrador especial usado para contar as execuções dos programas (program counter)
$lo		Registrador especial, guardam resultados da multiplicação e divisão. Não é acessado diretamente, precisa usar o comando mflo
$hi		Registrador especial, guardam resultados da multiplicação e divisão. Não é acessado diretamente, precisa usar o comando mfhi

Os registradores $v0 e $v1 são registradores que recebem valores das expressões e resultados de funções. Servem como retorno de chamadas do sistema – *Syscall*. O registrador $v0 é uma espécie de chaveador. Alguns números recebidos nele, faz com que o processador MIPS execute funções específicas. Os serviços mais usados estão listados na tabela abaixo:

Serviço	$v0	Argumento	Resposta
Imprime um inteiro	1	$a0 deve conter o número inteiro a imprimir	
Imprime um *float*	2	$f12 deve conter o número *float* a imprimir	
Imprime um *double*	3	$f12 deve conter o número *double* a imprimir	
Imprime uma *string*	4	$a0 deve conter a *string* a imprimir	
Lê um número inteiro	5		$v0 armazena o número inteiro digitado pelo usuário.
Lê um número *float*	6		$f0 armazena o número *float* digitado pelo usuário.
Lê um número *double*	7		$f0 armazena o número *double* digitado pelo usuário.
Lê uma *string*	8		$a0 armazena a *string* digitada pelo usuário.
Sair	10		

5 A estrutura do MIPS Assembly

5.1 Componentes do *MIPS Assembly*

No simulador MARS, o código adquire cores específicas, dependendo da função do trecho do código. A tabela abaixo exemplifica cada um dos casos possíveis.

Componente	Função
# Sim	Comentário
.data	Diretiva
li, la, add, beq, j	Instrução
$v0, $a0, $t0, $t1, $s0	Registradores
Principal: main:	Etiquetas
12.56, 3, 0xAFB	Dado
“Olá Mundo!”, “Marcos”, “Oi”	String
‘S’, ‘,N’	Caractere

5.2 Template do *MIPS Assembly*

```
# Autor: digite seu nome aqui
# Data: ___/____/ _____
# Descricao: Informe aqui qual o proposito do programa.

.data    # Diretiva de dados
         # Área de segmentação de dados:
         # Aqui são definidas as constantes e as variáveis do programa

.text    # Diretiva de texto
         # Área de segmentação de texto:
         # Aqui ficam as instruções MIPS do código do programa

.globl programa # Diretiva global programa:
                # O programa principal começa aqui
```

Diretiva .data

A diretiva *.data* (diretiva de dados) é o segmento de código separado para a declaração das constantes e das variáveis.

Para que sejam **declaradas constantes ou variáveis** no código MIPS assembly, é necessário primeiramente, definir um nome à variável. Depois é necessário definir o tipo de dados armazenado e por último defini o conteúdo da constante ou da variável.

Sintaxe: <nome da variável ou constante <:><tipo><valor>

```
.data                # Diretiva de dados
    idade: word 20   # nome da variável; tipo; valor
```

Diretivas de tipagem de dados

A tipagem de dados no assembly é uma diretiva. Tal como todas as diretivas, nesse caso cada qual deve preceder por um ponto. Em seguida estão algumas das principais tipagens:

Diretiva de Tipagem	Tipo
.asciiz	Usado para declarar uma constante ou variável do tipo texto (*String*)
.word	Usado para declarar uma constante ou variável do tipo inteiro (*Integer*)
.float	Usado para declarar uma constante ou variável do tipo decimal (*Float*)
.double	Usado para declarar uma constante ou variável do tipo decimal (*Double*)
.byte	Usado para declarar uma constante ou variável do tipo caractere (*Char*)
.space	<qtde>Usado para declarar uma constante ou variável do tipo String com espaços em branco.

Diretiva *.macro* e *end_macro*

As macros são usadas para geração de sub-rotinas. As sub-rotinas têm como propósito executar procedimentos específicos (por exemplo, funções) dentro do código. A sub-rotina modulariza o programa, deixando-o mais refinado. Aos ser carregada na memória, apenas o módulo principal é inicializado. Somente se o programa fizer uma chamada de sub-rotina é que, então, será carregada na memória para a execução. Esta estratégia deixa a execução do programa mais rápida.

Sub-rotinas

No exemplo em seguida, a sub-rotina tem como objetivo finalizar o programa através de uma macro, denominada "fim":

```
.macro fim            # Inicio da macro chamada fim
    li $v0, 10        # Configura para fim do programa
    syscall           # Executa a instrução
.end_macro            # Fim da macro
```

No código acima, a instrução *li* irá carregar o número inteiro 10 no registrador $v0, e configurar o serviço de fim do programa (*Exit*). Isto é equivalente a dizer que o registrador $v0 recebe o número inteiro 10 ou simplesmente $v0 = 10.

Se necessário, na criação da função, a entrada é definida na própria macro. Essa entrada é processada pela macro e ao final do processo, um valor de retorno é apresentado ao programa executa a chamada respectiva.

Nas linguagens de programação em alto nível, geralmente as funções são acompanhadas de parênteses para representar a entrada de dados. Por exemplo, na linguagem C, a

sentença *printf*("Olá") é usada imprimir uma string na tela. Já em *MIPS Assembly* é necessário criar uma macro que receba um valor no registrador e criar um mini código para tratar o valor do registrador.

Diretivas *.globl* e *.text*

A diretiva global para o MIPS assembly é semelhante a função principal main() da linguagem C. A linguagem precisa saber qual o módulo que deve ser inicializado primeiro, sendo assim reconhecido como a função principal do sistema. No assembly essa diretiva é conhecida como *.globl*.

Tal como anteriormente mencionado, um código *MIPS Assembly* possui segmentações, também chamamos de diretivas e que são muito usadas nos códigos: *.data* e a *.text*, sendo a diretiva *.data* um segmento (uma área) destinada a declaração das constantes e variáveis e a diretiva *.text* um segmento destinada propriamente dito ao código do programa. Em seguida são apresentados exemplos de aplicação das diretivas *.data*, *.text* e *.globl*.

Exemplo 01:

```
.data             # Diretiva de dados
.text             # Diretiva de texto
    li $v0, 10    # Configura o fim do programa
    syscall       # Executa a instrução
```

Neste exemplo, têm-se uma diretiva segmentando os dados (.data) e uma diretiva segmentando a área de construção do código (.text). No simulador MARS a diretiva .globl pode ser abstraída em algumas condições, principalmente se não houver fatores que causem conflitos como por exemplo, vários blocos de execução.

Exemplo 02:

```
.data                 # Diretiva de dados
    pi: .float 3.14   # Variável tipo float
.text                 # Diretiva de texto

.globl principal      # Definimos executar principal
principal:            # Bloco denominado de principal
    li $v0, 10        # Configura o fim do programa
    syscall           # Executa a instrução
```

Este código está dividido em diretiva de dados (segmento de dados), com um exemplo de criação de uma variável do tipo *.float*. Na diretiva *.text* encontra-se o código separando a execução por uma função principal. A configuração da função principal é definida pela diretiva *.globl*, e dentro da função principal têm-se dois comandos que configuram o processador para apenas terminar o programa.

6 Simulador MARS

O simulador adotadp é o MARS (*MIPS Assembler and Runtime Simulator*). Trata-se de um editor de linguagem assembly, montador, simulador e depurador para o processador MIPS, desenvolvido pela Universidade do Estado do Missouri, nos EUA. O simulador é escrito em Java e não requer a instalação. Por outro lado, requer que a Máquina Virtual Java (JVM) esteja instalada para funcionar.

Instalando o MARS 4.5

- Para instalar no Windows

  1) Instale o Java em seu computador.
  Acesse https://www.oracle.com/technetwork/java/javase/downloads/index.html.

  2) Faça o download do simulador MARS:
  Acesse http://courses.missouristate.edu/KenVollmar/mars/MARS_4_5_Aug2014/Mars4_5.jar

  3) Execução do simulador pelo ambiente gráfico: Localize o arquivo Mars4_5.jar e dê dois cliques para executá-lo.

- No Linux

  1) Instale o Java
  sudo apt-get install default-jre -y
  sudo apt-get install default-jdk –y

  2) Faça o download do simulador MARS:
  ttp://courses.missouristate.edu/KenVollmar/mars/MARS_4_5_Aug2014/Mars4_5.jar

  3) Execução do simulador pelo ambiente texto:
  Digite **java –jar Mars4_5.jar** e para executar seu código diretamente no shell digite: **java –jar Mars4_5.jar mips1.asm**

  4) Execução do simulador pelo ambiente gráfico: Localize o arquivo Mars4_5.jar e dê dois cliques para executá-lo.

Limitações do MARS 4.5

A versão MARS 4.5 apresenta algumas limitaçõs e alguns BUGS:

- O trabalho com memória no simulador é limitada a 4MB.

- Não há execuções em pipelines.
- Não abre arquivos se eles forem links ou ícones para outros arquivos.
- Possui poucas configurações em que o usuário possa alterar.

- BUGS:

  - Apenas informa que o código contém erro.

  - Alguns usuários relatam que a IDE fica lenta quando se codifica por muito tempo, abrindo vários arquivos. O problema desaparece se o MARS for reiniciado.

Inicializando o MARS 4.5

Após descompactar o conteúdo do arquivo compactado em seu computador, basta executar o arquivo **Mars4_5.jar**. Então uma tela inicial se apresenta, Figura 1, e após criar/abrir um arquivo o ambiente muda levemente, Figura 2. O ambiente do simulador pode ser dividido em três segmentos: o editor no canto superior esquerdo onde todo o código é escrito, o console de mensagens abaixo do editor e a lista de registradores que representam a "CPU"para o programa.

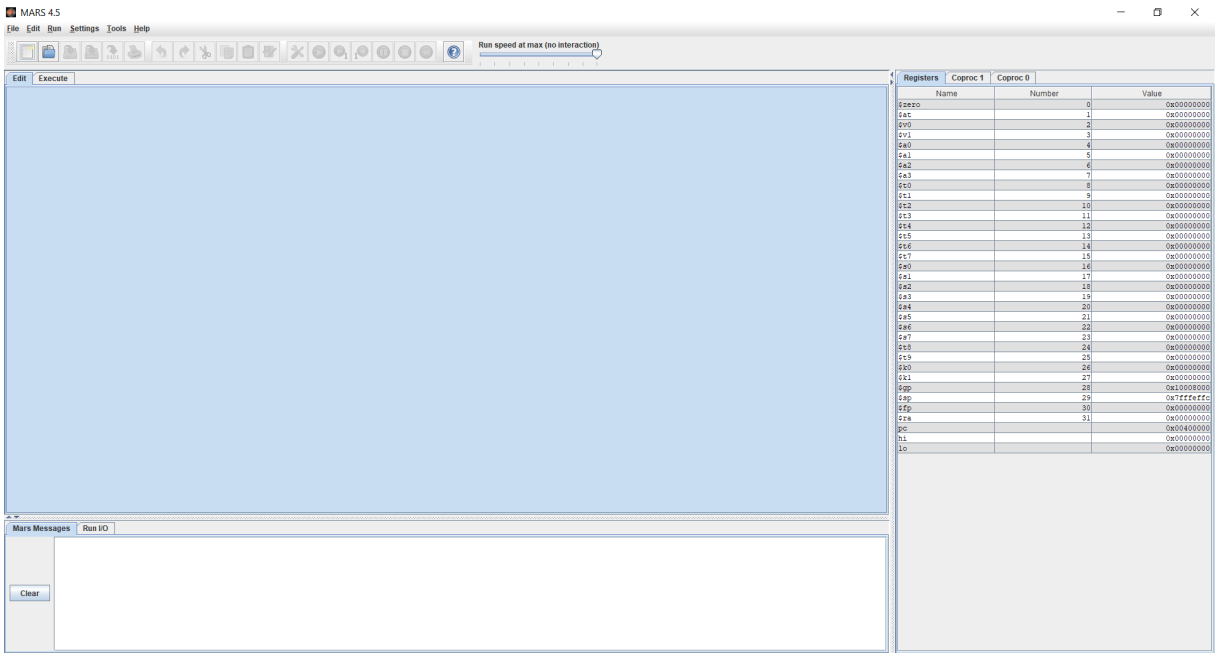

Figura 1: Tela inicial do programa MARS em execução

-

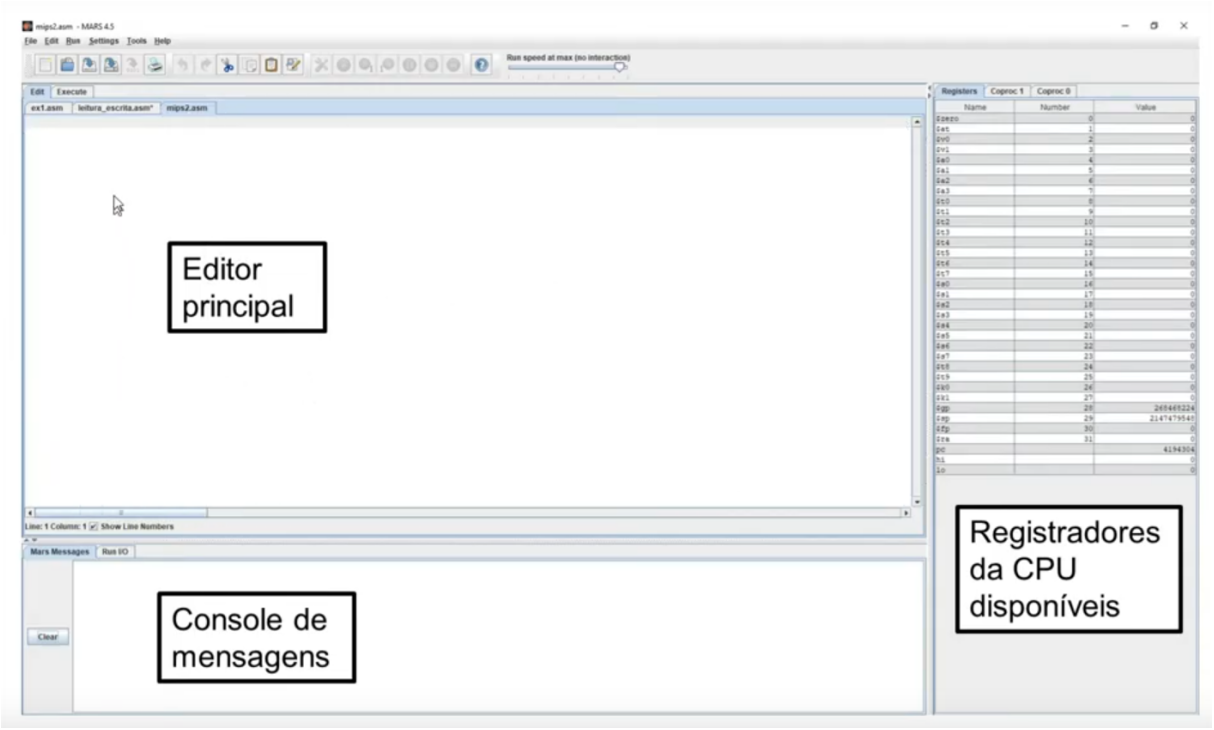

Figura 2: Tela de edição do MARS

Para fazer a montagem, Figura 3, pressione F3. O ambiente muda, com dois novos segmentos obtendo a posição do editor: o segmento de texto onde cada linha de código de montagem fica livre de "pseudoinstruções"na coluna "básica"e o código de máquina para cada instrução na coluna "código", e o segmento de dados onde podemos dar uma olhada em uma representação da memória de um processador com ordem little-endian .

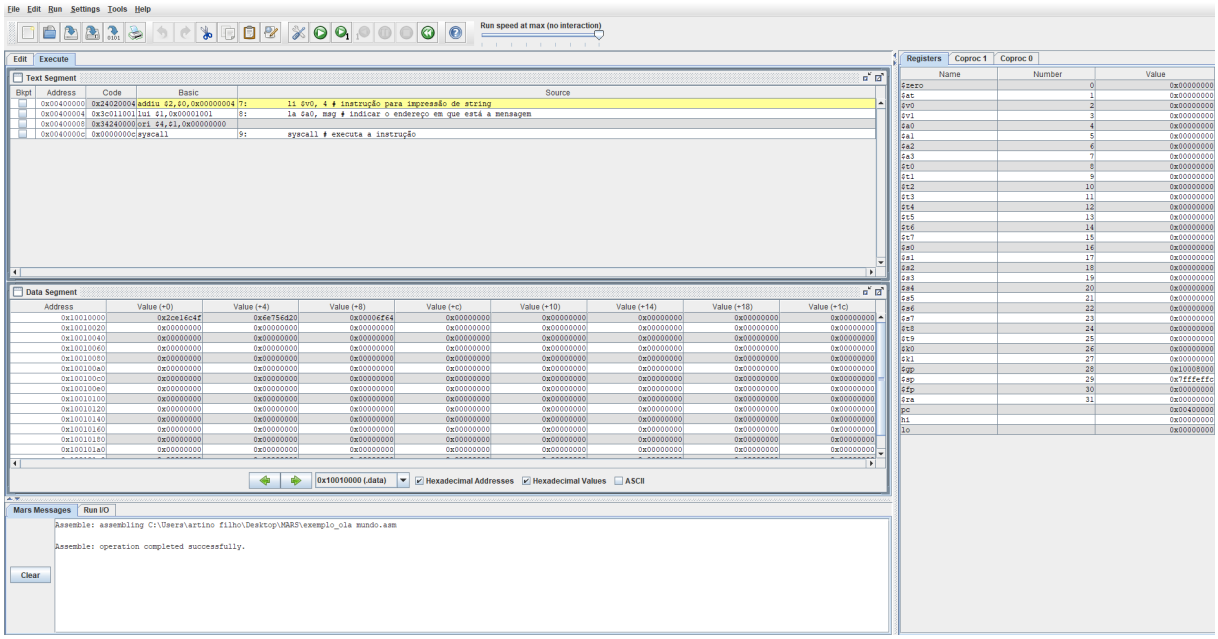

Figura 3: Montagem do código

Após a montagem, é possível executar o código de uma vez (F5) ou passo a passo (F7), bem como retroceder a execução vários passos para trás (F8).

7 Exemplos

7.1 Olá Mundo

Olá Mundo! (simplificado)

Segue uma sugestão para criar o primeiro código: "Olá Mundo!". Consiste de duas instruções: *li* e *la*. A instrução *li* serve para ler imediatamente um valor num registrador e a instrução *la* serve para ler imediatamente um endereço de memória num registrador. É tal como segue:

```
.data    # area para especificar dados e posicoes da memória

         msg: .asciiz  "Ola, mundo!" # mensagem a ser exibida ao usuario

.text              # area para instrucoes do programa

         li $v0, 4         # instrucao para impressão de string
         la $a0, msg       # indicar o endereço em que esta a mensagem
         syscall           # executa a instrução
```

Como é um código muito pequeno e não está dividido em blocos ou usando diversas funções e procedimentos, o MARS entende que abaixo do texto tudo faz parte da função principal.

Olá Mundo! (com *.globl*)

Uma outra possibilidade seria:

```
.data                # Diretiva de dados
    msg: .asciiz  "Ola, mundo!"

.text                # Diretiva de texto
.globl bloco1        # Diretiva global
bloco1:              # Bloco de código bloco1
      li $v0, 4      # impressão de string
      la $a0, msg    # $a0 recebe msg
      syscall        # Executa o modulo bloco1
      li $v0, 10     # Fim do programa
      syscall        # Executa a instrucao acima
```

Olá Mundo! (com *.globl* e com *.macros*)

Neste código, cada função ou procedimento pode ter sua diretiva .data e .text em particular. Em seguida tem-se um exemplo:

```
.data                          # Diretiva de dados
.macro finalizarprograma       # Macro
    li $v0, 10                 # Sair do programa
    syscall                    # Executa
.end_macro                     # Fim da macro
.macro printf (%str)           # Macro printf
    .data                      # Diretiva de dados da macro
        msg:  .asciiz  %str    # Declaração da variável msg
    .text                      # Diretiva text da macro
        li $v0,4               # Imprimir string
        la $a0, msg            # Carrega msg no registrador
        syscall                # Executa
.end_macro                     # Fim da macro printf
.text                          # Diretiva .text
.globl principal               # Diretiva global
principal:                     # Bloco de codigo principal
    printf("Olá Mundo!\n")     # Executa a funcao printf
    finalizarprograma          # Executa o procedimento
```

7.2 Inicializar vetor com 10 posições

No exemplo a seguir, inicializa-se um vetor - inicialmente, sem nada em sua memória - com o dígito 1. Ou seja, ao final da execução do código, as 10 posições do vetor devem receber o valor 1.

```
.data   # area para especificar dados e posicoes da memoria
v: .word        # declaração da variavel v
```

```
.text   # area para instrucoes do programa

li $t1,1          # atribui o valor 1 ao registrador $t1
li $t2,10         # atribui o valor 10 ao registrador $t2
li $t3,0          # atribui o valor 0 ao registrador $t3
la $t4,v          # carrega o endereço na memória

loop:
        sw $t1,0($t4)    # atribui o conteúdo do registrador $t1 para a
                         # primeira posição do vetor (indicado por $t4)
        addi $t4,$t4,4       # adiciona 4 ao registrador $t3 (cada
                             # registrador tem 4 bytes)
        addi $t3,$t3,1       # adiciona 1 ao registrador $t3
        bne $t2,$t3,loop     # se t2 e $t3 não forem iguais, executa
                             # o desvio syscall
```

8 Leitura recomendada

Os endereços em seguida auxiliam no aprimoramento da técnica do uso da linguegem, do editor e do simulador:

- Documentação do MIPS
- Site Embarcados
- Introdução à Arquitetura de Computadores/Instruções do MIPS - WikiLivros

9 Atividades/Tarefas

- Para efeitos de estudo e capacitação no uso do simulador (não é necessário entrega de material) sugere-se:
  - Implementar o exemplo da seção 7.2 no simulador MARS;
  - Executar passo a passo o programa em Assembly, verificando as alterações na memória e registradores;
  - Alterar o programa para que a memória (”array”) seja preenchida com valores crescentes;
- Responder ao teste prático (Teste P03), no AVA (Semana 4)

10 Bibliografia

- Silva Junior, M.T.G. **”O guia prático da linguagem MIPS Assembly - Didático, prático e fácil.**

- Patterson, D.A. Hennessy, J.L. **"Organização e projeto de computadores - A interface hardware/software"**.