LAB02 - SPIM

O SPIM é um simulador que executa programas escritos para os processadores MIPS R2000 e R3000. Ele é capaz de ler e executar arquivos em linguagem de montagem do MIPS. Um manual detalhado deste simulador feito pelo próprio autor pode ser encontrado aqui.

A seguir serão apresentadas rapidamente algumas de suas principais funções e será demonstrada a sua utilização em exemplos práticos, para que este possa ser usado na elaboração do Projeto 2.

Para que você possa se familiarizar com o funcionamento do software, veja antes Familiarizando-se com a interface do SPIM (apenas o PCSPIM)6.

Para executar seus programas escritos em linguagem de montagem do MIPS, estes devem ser feitos em um editor de texto comum, como o notepad.exe. Estes arquivos deverão ser salvos com a extensão "*.s*" ou "*.asm*" para depois serem abertos e executados pelo SPIM.

A seguir são apresentados alguns exemplos de programas que podem auxiliar no desenvolvimento do projeto:

Exemplo 1 - Trabalhando com registradores

Exemplo 2 - Trabalhando com dados em memória

Exemplo 3 - Outra forma de trabalhar com a memória

Exemplo 4 - Chamada de procedimentos e armazenamento de dados na pilha.

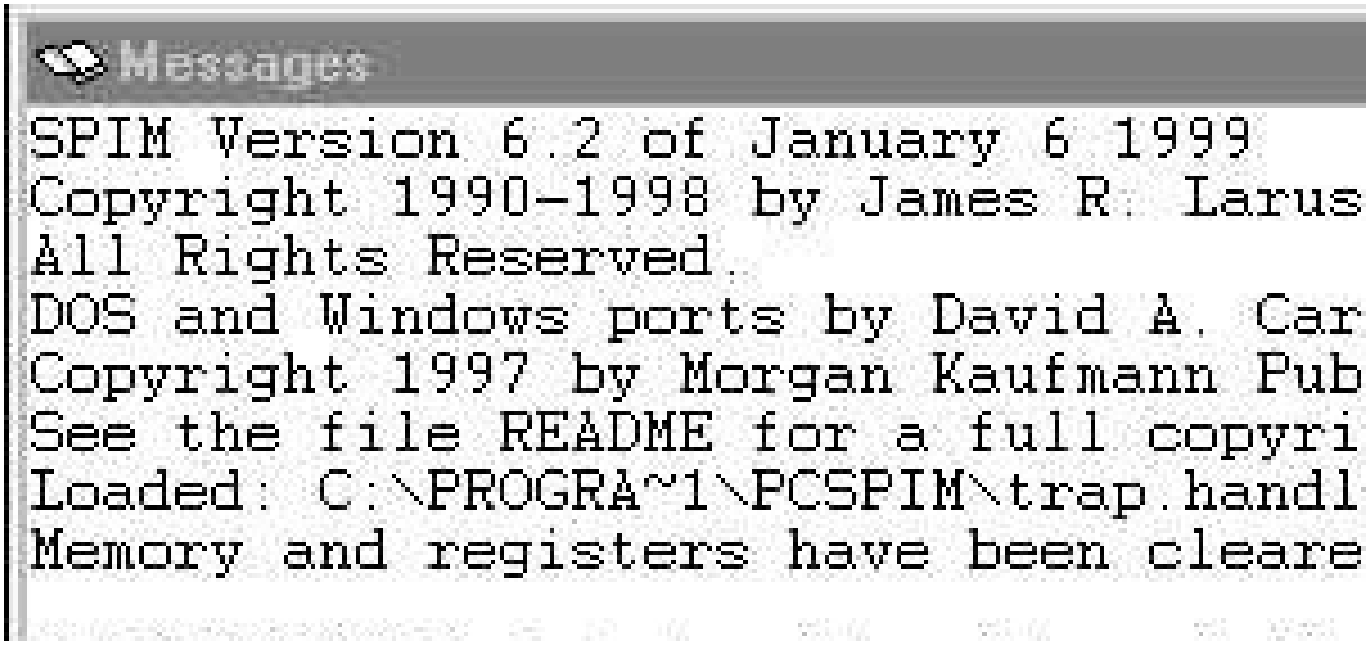
Exemplo 5 - Utilização do console do SPIM para entrada e saída de dados.

Familiarizando-se com o SPIM

O PCSpim é composto por cinco janelas principais, que são:

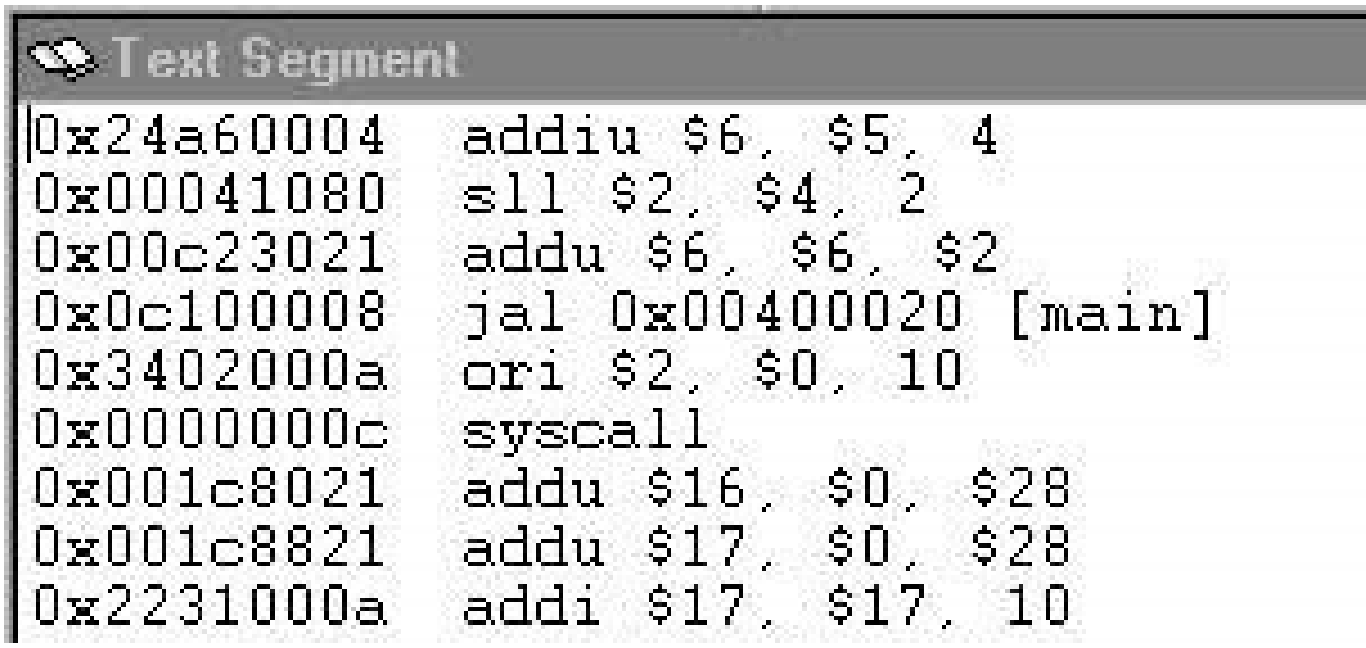
1. Messages

Contém as mensagens geradas pelo Spim para o usuário. Geralmente são apresentadas mensagens sobre o carregamento do programa ou a execução do mesmo e erros ocorridos se for o caso.



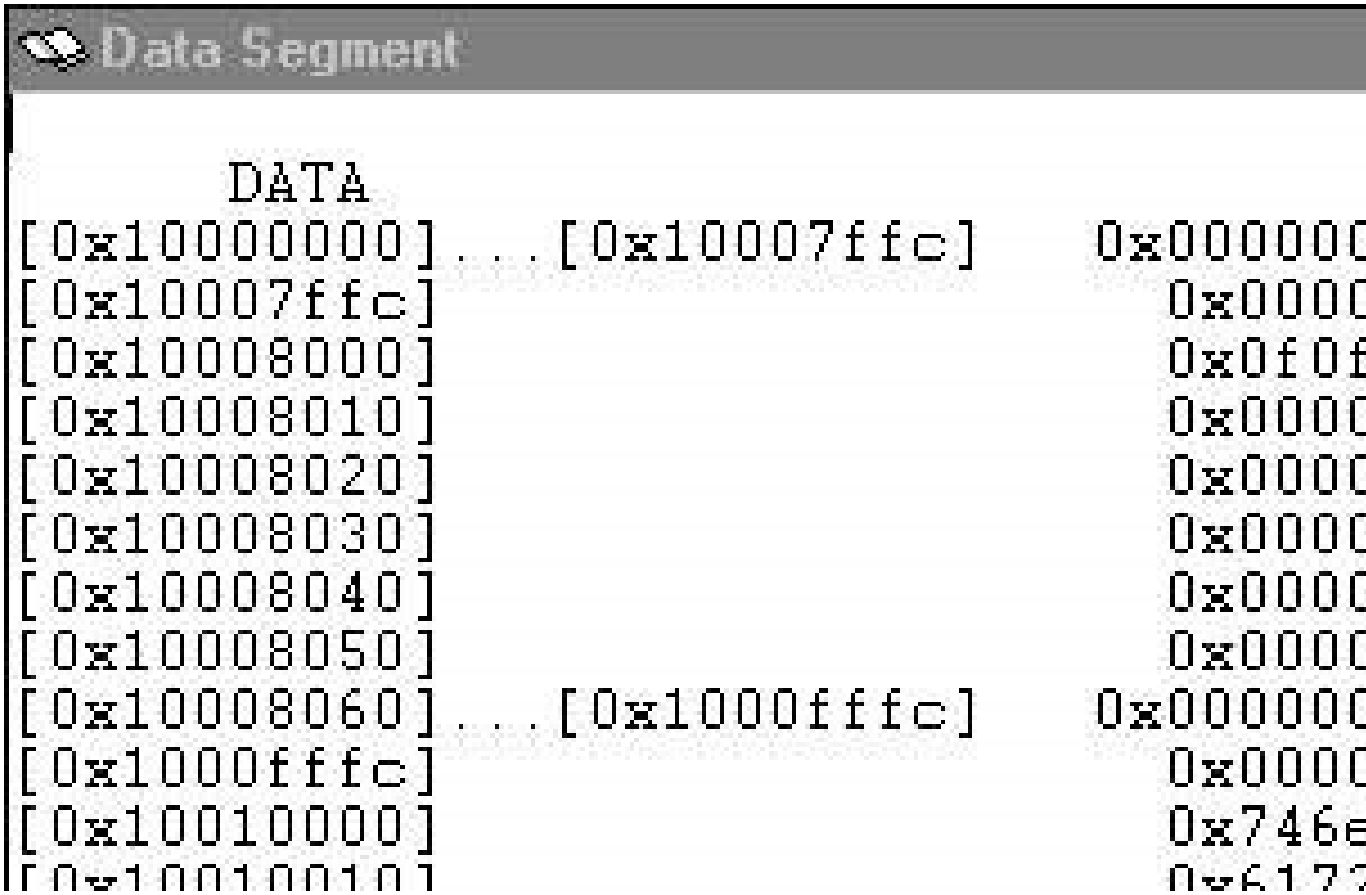
2. Text Segment

Nesta janela é mostrado as instruções que foram carregadas em memória. As instruções aparecem em duas colunas, a da direita para o código que foi carregado e a da esquerda para as instruções geradas pelo Spim.



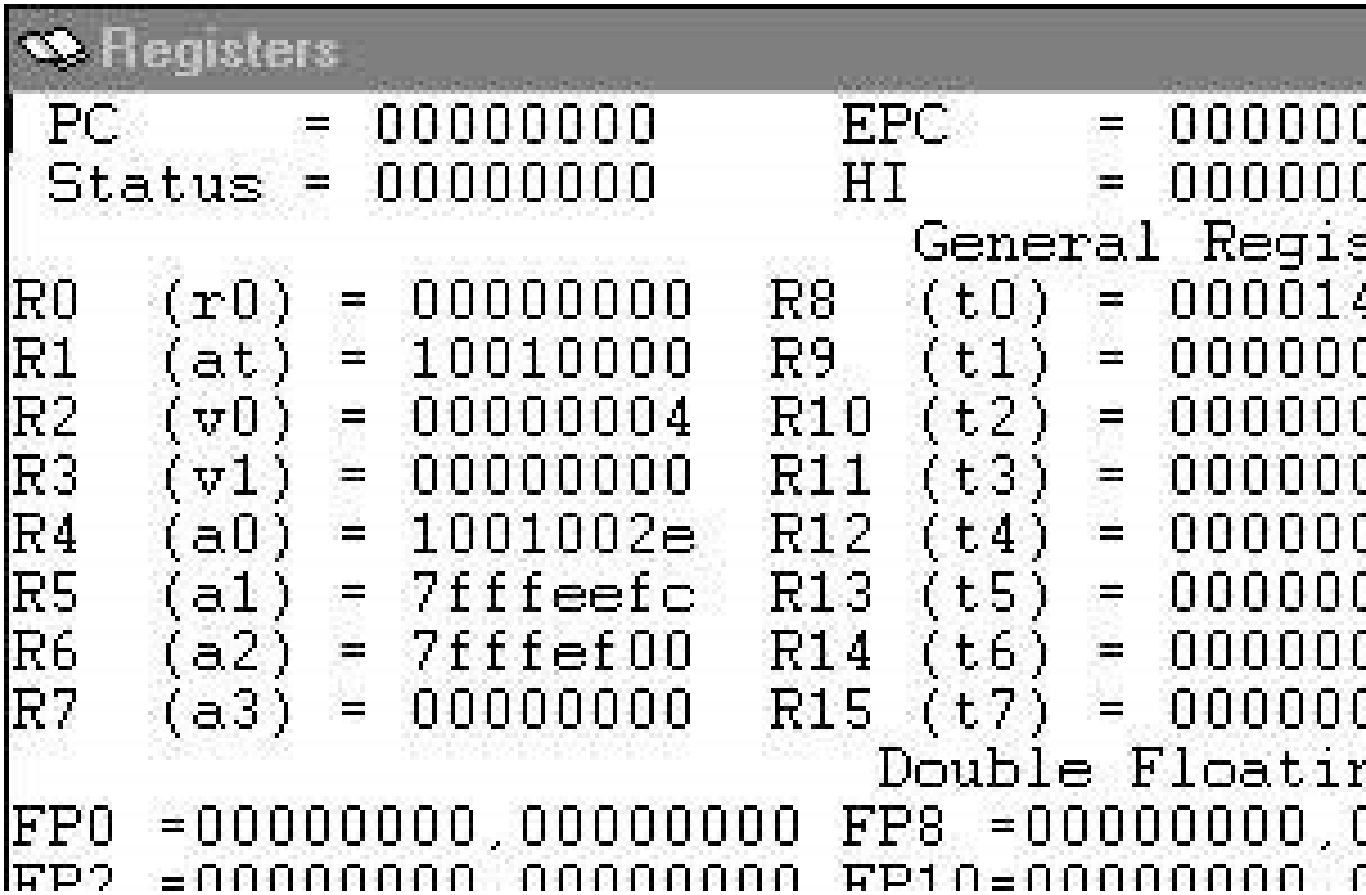
3. Data Segment

Mostra os dados carregados em memória e os dados da pilha.



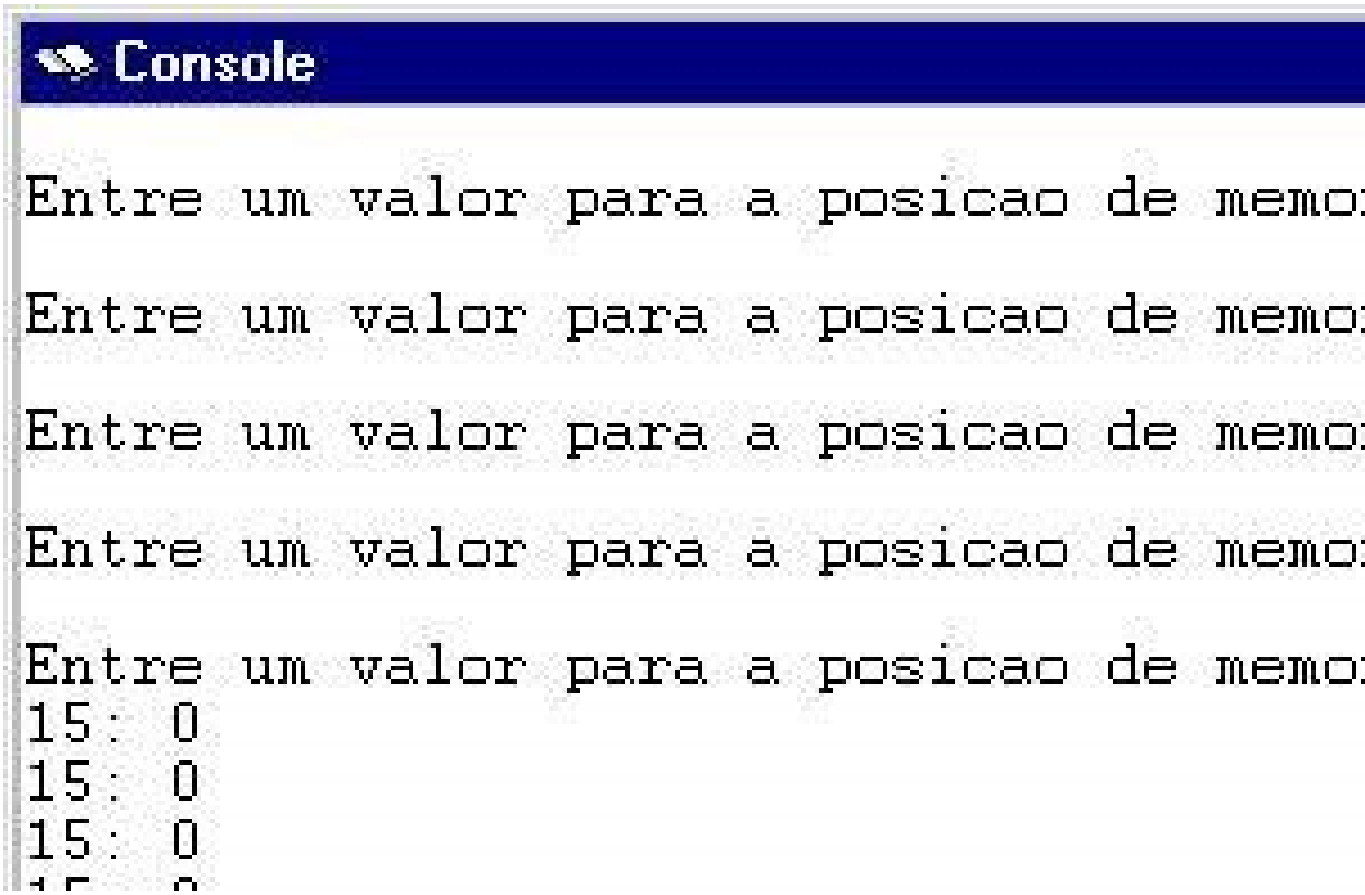
4. Registers

Esta janela mostra os valores armazenados em todos os registradores do MIPS, incluindo os da unidade de ponto flutuante (FPU).



5. Console

No Spim é possível usar uma espécie de "console" para exibir mensagens e receber entrada de dados.


Console
Entre um valor para a posicao de memo
Entre um valor para a posicao de memo
Entre um valor para a posicao de memo
Entre um valor para a posicao de memo
Entre um valor para a posicao de memo
15: 0
15: 0
15: 0

Serviços do Sistema

O Spim possui uma lista de serviços implementados para auxiliar principalmente na interface com o usuário por meio de um console. Para utilizar um desses serviços, basta colocar em $v0 o código do serviço, definir os parâmetros (se houverem) e em seguida utilizar a instrução syscall.

Serviço	**Código**	**Parâmetros**	**Resultados**	**Descrição**
print_int	1	$a0 = integer		Escreve um valor inteiro no console.
print_float	2	$f12 = float		
print_double	3	$f12 = double		
print_string	4	$a0 = string		Escreve uma string no console.
read_int	5		integer (em $v0)	Lê um valor inteiro entrado no console.
read_float	6		float (em $f0)	
read_double	7		double (em $f0	
read_string	8	$a0 = buffer, $a1 = length		Lê uma string do console (n caracteres)
sbrk	9	$a0 = quantidade	endereço (em $v0)	Retorna um ponteiro para um bloco de memória contendo n bytes)
exit	10			Encerra a execução do programa.

Exemplo 1 - Trabalhando com registradores

Abra o notepad.exe (ou outro editor se preferir). Executaremos o exemplo apresentado a seguir, cujo objetivo é realizar a operação *f = (g + h) - (i + j)*.

Detalhe importante: Para ques os programas possam ser executados no SPIM, deverá **sempre** ser definido onde o programa deverá ser iniciado. Isto é feito através da declaração do símbolo *main,* da seguinte maneira:

```
.text                   # indica que as linhas seguintes contém instruções
.globl main             # define o símbolo main como sendo global
main:                   # indica o início do programa
```

Agora podemos entrar com as instruções do programa. De início, precisamos atribuir valores para as variáveis *g, h, i* e *j* (registradores $s1, $s2, $s3 e $s4 respectivamente). Para isso entraremos com o seguinte código:

```
li $s1,15               # registrador $s1 contém o valor imediato 15
li $s2,36               # registrador $s2 contém o valor imediato 36
addi $s3, $zero, 12     # registrador $s3 contém o valor imediato 12
addi $s4, $zero, 19     # registrador $s4 contém o valor imediato 19
```

O uso das instruções *li* e *addi* foi proposital para demonstrar essas duas formas de carga de valores imediatos em registradores. Agora vamos executar a operação colocando o resultado em $s0.

```
add $t0,$s1,$s2         # registrador $t0 contém g + h
add $t1,$s3,$s4         # registrador $t1 contém i + j
sub $s0,$t0,$t1         # registrador $s0 contém (g + h) - (i + j)
```

Salve o arquivo com o nome de "*lab02_1.s*" e abra-o no SPIM através do menu File, Open. Abra a janela dos Registradores no menu Window, Registers. Observe que os registradores estão todos zerados. Agora clique no menu Simulator, Go e em seguida em OK. Isto irá executar o código. Observe agora a mudança no estado dos registradores.

```
General Registers
(t0) = 00000033  R16 (s0) = 00000014
(t1) = 0000001f  R17 (s1) = 0000000f
(t2) = 00000000  R18 (s2) = 00000024
(t3) = 00000000  R19 (s3) = 0000000c
(t4) = 00000000  R20 (s4) = 00000013
(t5) = 00000000  R21 (s5) = 00000000
(t6) = 00000000  R22 (s6) = 00000000
(t7) = 00000000  R23 (s7) = 00000000
```

O $s0 contém o resultado da nossa operação: 14h (os valores dos registradores estão todos em hexadecimal). Conferindo: (15+36) = 51; (12+19) = 31; (51-31) = 20 = (14h).

Exemplo 2 - Trabalhando com dados em memória

Detalhe importante: O acesso à memória do SPIM em nossos programas deve ser feito com valores acima da posição inicial do global pointer (10008000h). Esta é a parte da memória do SPIM que iremos utilizar para armazenar nossos dados. Mais a frente veremos também como armazenar dados "constantes" de uma forma mais prática, sem a necessidade de utilizar as instruções de store.

Vamos fazer um exercício simples de acesso a memória:

Tendo-se um array de 100 elementos (words) que inicia no endereço de memória 5000 (em direção aos endereços crescentes) transfira este array para o endereço 6000.

Primeiro precisamos carregar na memória este array de elementos que foi considerado no enunciado. Como as posições de memória 5000 e 6000 estão fora da área que temos acesso na memória, utilizaremos o $gp (global pointer) + 5000 como endereço inicial do array fonte. Já para o array destino, utilizaremos o valor de $gp + 6000.

```
.text
.globl main
main:
move $s0,$gp
addi $s0,$s0,5000          # Ponteiro para os dados do array fonte ($gp) + 5000
move $s2,$s0
addi $s2,$s2,400           # Marcador para indicar o final do array $gp + (100posições de 4bytes)
```

Desta forma temos os ponteiros necessários para trabalhar com o primeiro array. O $s0 será o ponteiro e será incrementado sempre em 4 posições para apontar para a próxima word (próximo elemento). Vamos armazenar nele um dado qualquer, como por exemplo um valor incrementado sempre em 9 (9, 18, 27, 36...)

Detalhe importante: Para marcar pontos importantes no programa, utilizamos "labels" (rótulos). É através deles que executamos funções como jump e branch. Para definir um label, coloque um identificador seguido do sinal de dois pontos, ex: "repetir:" e escreva o código. Neste exercício faremos um loop para preencher os 100 elementos do array, por isso vamos precisar de um label para chamar a cada iteração. Quando o label é chamado (através de um bne por exemplo) a próxima instrução a ser executada é a da linha seguinte ao label.

```
li $t0,9                   # Carrega no reg. temporário $t0 um valor para ser armazenado no array
dados:
sw $t0,0($s0)              # Armazena o valor na posição do
```

```
                                    array apontada por $s0
addi $s0,$s0,4                      # Aponta para a próxima posição no
                                    array (incrementa em 4 o ponteiro)
addi $t0,$t0,9                      # Altera o valor a ser armazenado
                                    no array (incrementa em 9)
bne $s0,$s2,dados                   # Enquanto não chegar ao fim do
                                    array, repete o laço
```

Vamos executar esta primeira parte do programa para testarmos o armazenamento dos valores do array. Salve o arquivo com o nome de "*lab02_2a.s*" e abra-o no SPIM. Execute o código (F5 e depois OK). Vamos verificar se ocorreu tudo bem. Na janela dos registradores o $s0 deverá estar em 10009518h que é a marca do final do array (o array inicia em 10009388h e o último elemento está em 10009514Ch). O registrador $t0 contém o valor 38Dh (909 em decimal), ou seja, o valor que seria armazenado na posição seguinte a última. Até aqui tudo Ok.

Agora verificaremos os valores em memória. Abra a janela Data Segment (Window, Data Segment). Ela deverá estar assim:

```
      DATA
[0x10000000]...[0x10009384]   0x00000000
[0x10009384]                    0x00000000  0x00000009  0x00000012
[0x10009390]                    0x0000001b  0x00000024  0x0000002d  0x00000036
[0x100093a0]                    0x0000003f  0x00000048  0x00000051  0x0000005a
[0x100093b0]                    0x00000063  0x0000006c  0x00000075  0x0000007e
[0x100093c0]                    0x00000087  0x00000090  0x00000099  0x000000a2
[0x100093d0]                    0x000000ab  0x000000b4  0x000000bd  0x000000c6
[0x100093e0]                    0x000000cf  0x000000d8  0x000000e1  0x000000ea
[0x100093f0]                    0x000000f3  0x000000fc  0x00000105  0x0000010e
[0x10009400]                    0x00000117  0x00000120  0x00000129  0x00000132
[0x10009410]                    0x0000013b  0x00000144  0x0000014d  0x00000156
[0x10009420]                    0x0000015f  0x00000168  0x00000171  0x0000017a
[0x10009430]                    0x00000183  0x0000018c  0x00000195  0x0000019e
[0x10009440]                    0x000001a7  0x000001b0  0x000001b9  0x000001c2
[0x10009450]                    0x000001cb  0x000001d4  0x000001dd  0x000001e6
[0x10009460]                    0x000001ef  0x000001f8  0x00000201  0x0000020a
[0x10009470]                    0x00000213  0x0000021c  0x00000225  0x0000022e
[0x10009480]                    0x00000237  0x00000240  0x00000249  0x00000252
[0x10009490]                    0x0000025b  0x00000264  0x0000026d  0x00000276
[0x100094a0]                    0x0000027f  0x00000288  0x00000291  0x0000029a
[0x100094b0]                    0x000002a3  0x000002ac  0x000002b5  0x000002be
[0x100094c0]                    0x000002c7  0x000002d0  0x000002d9  0x000002e2
[0x100094d0]                    0x000002eb  0x000002f4  0x000002fd  0x00000306
[0x100094e0]                    0x0000030f  0x00000318  0x00000321  0x0000032a
[0x100094f0]                    0x00000333  0x0000033c  0x00000345  0x0000034e
[0x10009500]                    0x00000357  0x00000360  0x00000369  0x00000372
[0x10009510]                    0x0000037b  0x00000384  0x00000000  0x00000000
[0x10009520]...[0x10040000]   0x00000000
```

Aqui o SPIM apresenta os dados carregados em memória. Perceba que ele mostra somente as posições ocupadas. Os dados no array devem ser os seguintes: o valor 9 na primeira posição, o valor 18 na segunda e assim sucessivamente até o valor 900 na última posição. Podemos conferir os valores armazenados no nosso array: 9h, 12h, 1Bh... = 9, 18, 27... O último valor é 384h = 900. Tudo Ok até aqui!

Agora podemos continuar com o exercício. Vamos fazer a cópia dos dados para o array destino.

```
move $s0,$gp
addi $s0,$s0,5000                   # Definimos novamente o ponteiro para os
                                    dados do array fonte ($gp + 5000)
move $s1,$gp
```

```
addi $s1,$s1,6000          # Ponteiro para os dados do array destino ($gp + 6000)
transfere:
lw $t0,0($s0)              # Armazena em t0 o conteúdo da posição apontada por $s0 (array fonte)
sw $t0,0($s1)              # Armazena no array destino (apontado por $s1) o valor carregado
addi $s0,$s0,4             # Incrementa s0 em 4 (para chegar-se ao próximo elemento no array fonte)
addi $s1,$s1,4             # Incrementa s1 em 4 (para chegar-se ao próximo elemento no array destino)
bne $s0,$s2,transfere      # Enquanto s0 não chegar em 400 (100 elementos), repete o laço
```

Salve novamente o arquivo ("*lab02*_2b.s") e execute-o.

Agora a janela de dados vai apresentar os dois arrays, sendo que o segundo foi armazenado da posição 10009770h para cima. Observe que esta posição é o $gp (10008000h) + 6000.

```
[0x10009520]...[0x1000976c]   0x00000000
[0x1000976c]                    0x00000000
[0x10009770]                    0x00000009  0x00000012  0x0000001b  0x00000024
[0x10009780]                    0x0000002d  0x00000036  0x0000003f  0x00000048
[0x10009790]                    0x00000051  0x0000005a  0x00000063  0x0000006c
[0x100097a0]                    0x00000075  0x0000007e  0x00000087  0x00000090
[0x100097b0]                    0x00000099  0x000000a2  0x000000ab  0x000000b4
[0x100097c0]                    0x000000bd  0x000000c6  0x000000cf  0x000000d8
[0x100097d0]                    0x000000e1  0x000000ea  0x000000f3  0x000000fc
[0x100097e0]                    0x00000105  0x0000010e  0x00000117  0x00000120
[0x100097f0]                    0x00000129  0x00000132  0x0000013b  0x00000144
[0x10009800]                    0x0000014d  0x00000156  0x0000015f  0x00000168
[0x10009810]                    0x00000171  0x0000017a  0x00000183  0x0000018c
[0x10009820]                    0x00000195  0x0000019e  0x000001a7  0x000001b0
[0x10009830]                    0x000001b9  0x000001c2  0x000001cb  0x000001d4
[0x10009840]                    0x000001dd  0x000001e6  0x000001ef  0x000001f8
[0x10009850]                    0x00000201  0x0000020a  0x00000213  0x0000021c
[0x10009860]                    0x00000225  0x0000022e  0x00000237  0x00000240
[0x10009870]                    0x00000249  0x00000252  0x0000025b  0x00000264
[0x10009880]                    0x0000026d  0x00000276  0x0000027f  0x00000288
[0x10009890]                    0x00000291  0x0000029a  0x000002a3  0x000002ac
[0x100098a0]                    0x000002b5  0x000002be  0x000002c7  0x000002d0
[0x100098b0]                    0x000002d9  0x000002e2  0x000002eb  0x000002f4
[0x100098c0]                    0x000002fd  0x00000306  0x0000030f  0x00000318
[0x100098d0]                    0x00000321  0x0000032a  0x00000333  0x0000033c
[0x100098e0]                    0x00000345  0x0000034e  0x00000357  0x00000360
[0x100098f0]                    0x00000369  0x00000372  0x0000037b  0x00000384
[0x10009900]...[0x10040000]   0x00000000
```

Pronto! Os dados foram transferidos para o array destino.

Exemplo 3 - Outra forma de trabalhar com a memória

Existe uma outra maneira de se carregar dados em memória sem a utilização de instruções store. Ela é muito útil principalmente para a definição de valores ditos constantes (na verdade eles podem ser alterados futuramente já que temos acesso a sua posição de memória) e em inicialização de dados.

Para se armazenar os dados em memória, basta colocar a diretiva *.data*, que indica ao SPIM que os dados que a seguem devem ser carregados no segmento de dados.

```
.data                           # indica ao SPIM que as próximas linhas são dados
const1: .byte 1                 # const1 declarado como byte com valor 1
const2: .word 4                 # const2 declarado como word com valor 4
array1: .byte 9, 21, 16, 18, 38 # array1 declarado com 5 elementos (bytes)
tam1 : .byte 5                  # tam1 (tamanho do array1 em bytes)
array2: .word 206, 1543, 348, 709, 7000, 994 # array2 declarado com 6 elementos (words)
tam2 : .byte 24                 # tam2 (tamanho do array2 em bytes)
```

Realizaremos algumas operações simples com estes dados para ver como é feito o acesso a eles.

```
.text
.globl main
main:
lb $s0,const1                   # $s0 recebe o valor de const1 (1)
lw $s1,const2                   # $s1 recebe o valor de const2 (4)
```

Observe que o label do dado declarado funciona como um ponteiro de memória. Assim, para acessá-lo basta fazer a instrução de load, passando-o como parâmetro. Com os dois arrays vamos fazer algo para acessar todos os seus elementos, como por exemplo somar todos colocando o resultado em um registrador.

```
add $s2,$zero,$zero             # Zera o registrador $s2
add $t0,$zero,$zero             # Zera o registrador $t0
lb $t1,tam1                     # $t1 recebe o tamanho do array1 (5bytes)
soma1:
lb $t2,array1($t0)              # $t2 recebe o valor da posicao apontada por array1 + $t0
add $s2,$s2,$t2                 # $s2 é somado com o valor carregado do array1
add $t0,$t0,$s0                 # $t0 irá apontar para a próxima posição (atual +1)
bne $t0,$t1,soma1               # Repete até chegar ao final do array1
```

Para o segundo array basta trocar a instrução de carga de load byte para load word:

```
add $s3,$zero,$zero          # Zera o registrador $s3
add $t0,$zero,$zero          # Zera o registrador $t0
lb $t1,tam2                  # $t1 recebe o tamanho do array2 (24bytes)
soma2:
lw $t2,array2($t0)           # $t2 recebe o valor da posicao apontada
                             por array2 + $t0
add $s3,$s3,$t2              # $s3 é somado com o valor carregado do
                             array2
add $t0,$t0,$s1              # $t0 irá apontar para a próxima posição
                             do array (atual +4)
bne $t0,$t1,soma2            # Repete até chegar ao final do array2
```

Salve o arquivo com o nome de "*lab02_3.s*". Abra a Janela do segmento de dados. Observe que os dados definidos já estão carregados em memória. É interessante observar como esses dados são armazenados:

```
      DATA
[0x10000000]...[0x1000fffc]   0x00000000
[0x1000fffc]                    0x00000000
[0x10010000]                    0x00000001  0x00000004  0x04101509  0x00000526
[0x10010010]                    0x000000ce  0x00000607  0x0000015c  0x000002c5
[0x10010020]                    0x00001b58  0x000003e2  0x00000018  0x00000000
[0x10010030]...[0x10040000]   0x00000000
```

A primeira posição armazenada (10011000h) foi ocupada apenas por um byte (valor 1). O próximo valor era uma word (de valor 4). Em seguida vem o primeiro array (5 elementos - bytes). Veja que o SPIM opera a memória em *little-endian*, ou seja, a posição de memória vai sendo ocupada de trás para frente. O último valor do array1 é armazenado em outra posição de memória e após ele há um byte contendo o tamanho do array. Segue-se com o array2, mas desta vez utilizando toda a posição de memória (word) para cada elemento.

Execute o código. Na tela de registradores você vai ver os valores carregados em $s0, $s1, $s2, e $s3 - 1, 4, 88 (soma dos valores do array1) e 10800 (soma dos valores do array2).

Observação: Os dados carregados em memória através da diretiva .data são armazenados a partir da posição 10011000h. Tome cuidado quando estiver usando este tipo de carga em memória para não alcançar esta posição pelo programa, ou seja, não use valores maiores que $gp + 36864. Se necessitar de mais memória verifique até que posição os dados estáticos foram carregados e utilize um valor maior que esse.

Observação: Existem outros tipos além de .word e .byte. Entre eles podemos citar o tipo .double para armazenar valores de ponto flutuante com precisão dupla, o .float para precisão simples, o .half para armazenar meia-palavra (16bits) e outro que vamos fazer muito uso principalmente quando chegarmos à escrita de dados no console do SPIM: o .asciiz que armazena strings.

Curiosidade: O processador MIPS pode operar tanto em big-endian como em little-endian. O SPIM por sua vez, utiliza o padrão da máquina que o simulador está sendo executado. Assim, o SPIM é little-endian no Intel 80x86 e big-endian no Macintosh.

Exemplo 4 - Chamada de procedimentos e armazenamento de dados na pilha

Objetivo: Implementar um procedimento para o cálculo do fatorial de n. Tal procedimento deverá ser feito de tal maneira a chamar um segundo procedimento.
$v0 = retorna o resultado ao programa principal
$v0 = 0 se n < 1
$v0 = 1 se n = 1
$v0 = n! se n > 1
n! = n.(n-1).(n-2)... .1

```
.text
.globl main
main:                       # Programa Principal
li $a0, 5                   # Parametro = 5
jal Fatorial                # Chama a função fatorial,
                            # armazenando o endereco da proxima instrucao em $ra
move $s0, $v0               # Registrador $s0 recebe o valor resultado
li $v0, 10                  # Serviço do sistema no. 10 : Exit
syscall                     # Chamada de serviço do sistema
                            # Obs. Isto é necessário para encerrar aqui o programa
Fatorial:                   # Função Fatorial. Retorna o fatorial de n ($a0)
sub $sp,$sp,4               # Abre espaço para armazenar 1 item na pilha
sw $ra, 0($sp)              # Armazena o conteúdo do registrador $ra
                            # Obs: Caso os registradores $s0 - $s9 fossem alterados, seria necessário guardá-los na pilha também, e restaurá-los ao fim do procedimento. Por convenção, os registradores temporários $t* não precisam ser guardados.
li $t1, 1                   # $t1 = 1
slti $t0, $a0, 2            # Seta $t0 se $a0 < 2
beq $t0, $zero, Calcula     # Se $t0 não setado, $a0 > 1, portanto Calcula
add $v0, $zero, $zero       # Se não, $v0 = 0
beq $a0, $zero, Sai         # Se $a0 = 0, Sai
add $v0, $t1, $zero         # Se não, $v0 = 1
Sai:
lw $ra, 0($sp)              # Carrega o registrador $ra da pilha
add $sp, $sp, 4             # Elimina 1 item da pilha
jr $ra                      # Retorna ao programa principal
Calcula:
```

```
add $a1, $a0, $zero          # $a1 = $a0
Loop:
sub $a1, $a1, $t1            # $a1 = $a1 - 1
jal Multiplica               # Multiplica $a0 por $a1
add $a0, $v0, $zero          # $a0 = resultado da multiplicação
bne $a1, $t1, Loop           # Enquanto $a1 for diferente de 1, Loop
j Sai
Multiplica:
mult $a0, $a1                # Multiplica $a0 por $a1
mflo $v0                     # resultado em $v0
                             # Obs:. O resultado da instrução mult fica
                             armazenado nos registradores # $hi e $lo.
                             Para buscar os seus valores, utiliza-se
                             mflo e mfhi. Para # podermos trabalhar com
                             valores maiores em nosso programa bastaria
                             retornar o $lo em $v0 e o $hi em $v1.
jr $ra                       # Retorna
```

Salve o arquivo com o nome de "*lab02_04.s*" e abra-o no SPIM. Antes de executá-lo, abra as janelas de Text Segment e a de Data Segment e redimensione-as de modo que as duas fiquem visíveis na tela. Vamos executar o programa passo a passo, através do Single Step (F10). As primeiras 8 linhas de código no Text Segment foram geradas automaticamente pelo SPIM para controlar o fluxo do programa. Continue executando linha por linha até chegar ao nosso código, que começa com `[ 5: li $a0, 5 ]`. A próxima linha muda o fluxo do programa, chamando a função fatorial. Continue executando. Observe que dentro da função fatorial armazenamos o conteúdo de $ra na pilha, e isto pode ser confirmado na janela do segmento de dados. Agora abra a janela de Registradores e continue fazendo o passo a passo, acompanhando a evolução do resultado do fatorial em $a0. Ao final do laço, retornamos ao programa principal que coloca o resultado em $s0.

Outra maneira interessante de se resolver este exercício é através de procedimentos recursivos, ou seja, uma função fatorial que chama a si mesma com um parâmetro (que seria o resultado intermediário) até chegar a um resultado final.

Exemplo 5 - Utilização do console do SPIM para entrada e saída de dados

Objetivo: Implementar um programa para o cálculo da média das notas. O usuário pode entrar com quantas notas quiser no console. Para facilitar o cáculo, utilizaremos a divisão inteira (div), ou seja, o resultado será apenas a parte inteira da média.

Para trabalharmos com o console do SPIM, utilizaremos os Serviços de Sistema (abaixo) que ele fornece.

```
.data
msg1: .asciiz "\nEntre o
numero de avaliações da
disciplina: "
msg2: .asciiz "\nEntre um
valor para a nota "
msg3: .asciiz ": "
msg4: .asciiz "\nA média das
notas é: "
.text
.globl main
main:
add $t0, $zero, $zero        # Limpa o conteúdo de $t0
add $t1, $zero, $zero        # Limpa o conteúdo de $t1
numnotas:
li $v0, 4                    # Codigo SysCall p/ escrever strings
la $a0, msg1                 # Parametro (string a ser escrita)
syscall
li $v0, 5                    # Codigo SysCall p/ ler inteiros
syscall                      # Inteiro lido vai ficar em $v0
add $s0, $v0, $zero          # Armazena em $s0 o número de notas
loopnotas:
addi $t0, $t0, 1             # Incrementa $t0 - contador de notas
li $v0, 4                    # Codigo SysCall p/ escrever strings
la $a0, msg2                 # Parametro (string a ser escrita)
syscall
li $v0, 1                    # Codigo SysCall p/ escrever inteiros
add $a0, $zero, $t0          # Parametro (inteiro a ser escrito)
syscall
li $v0, 4                    # Codigo SysCall p/ escrever strings
la $a0, msg3                 # Parametro (string a ser escrita)
syscall
li $v0, 5                    # Codigo SysCall p/ ler inteiros
syscall                      # Inteiro lido vai ficar em v0
add $t1, $t1, $v0            # Soma a nota ao total
```

```
bne $t0, $s0, loopnotas         # Enquanto não preencher todas as notas, loop
Calcula:
div $t1, $s0                    # Divide o total pelo numero de notas
mflo $t2                        # Move o resultado para $t2
li $v0, 4                       # Codigo SysCall p/ escrever strings
la $a0, msg4                    # Parametro (string a ser escrita)
syscall
li $v0, 1                       # Codigo SysCall p/ escrever inteiros
add $a0, $zero, $t2             # Parametro (inteiro a ser escrito)
syscall
li $v0, 5                       # Apenas para esperar um [ENTER]
syscall
```

Salve o arquivo com o nome de "*lab02_5.s*" e abra-o no SPIM. Execute-o (F5). O SPIM deverá abrir a janela do Console para a entrada de dados.