Arquitectura de Computadores

Prof. Doutora Ana Isabel Leiria

Ano Lectivo 2004/05

Prof. Doutora Margarida Madeira e Moura

Eng. António Rosado

Ano lectivo 2005/06

1. INTRODUÇÃO

1.1 MEIOS

Para as aulas práticas de Arquitectura de Computadores é necessário:

- Conhecimento da linguagem C. Não se pretende programar em C, pelo que não será necessário o uso do compilador de C, mas pretende-se que os alunos compreendam e saibam interpretar programas ou excertos de programas escritos em C.
- Uso do simulador de MIPS R2000: SPIM. O simulador está instalado nos computadores da sala de aula. O SPIM poderá ser acedido através do seguinte endereço: http://www.cs.wisc.edu/~laurus/spim.html
- Guias das aulas práticas. Os alunos deverão trazer para cada aula o guia correspondente, que poderá ser acedido da página pessoal do docente das aulas práticas.

Livro recomendado:

- *Computer Organization and Design: The Hardware/Software Interface*; D. Patterson, J. Hennessy; Morgan Kauffmann Publishers, 2ª Ed., 1998.

1.2 OBJECTIVOS

São objectivos das aulas práticas de Arquitectura de Computadores:

- Introduzir os conceitos associados às linguagens Assembly.
- Aprender a linguagem Assembly do processador MIPS R2000.
- Comparar construções expressas em Assembly com construções em linguagens de alto nível.
- Executar programas em Assembly e ver os resultados.

1.3 ESTRUTURA DE UM COMPUTADOR

Os principais componentes funcionais de um computador são:

- A unidade de controlo.
- Os registos.
- A unidade lógica e aritmética (ULA).
- O "Program Counter" (PC)
- A memória
- O "Instruction Register" (IR)

A unidade de controlo emite uma sequência de sinais adequados para que sejam acedidas as instruções guardadas na memória e para que essas instruções possam ser executadas.

Os registos de uso geral armazenam temporariamente a informação que vem da memória ou valores de variáveis.

Na arquitectura MIPS, existem 32 registos, cada um com capacidade para guardar um número binário de 32 bits. Assim, em notação decimal podem ser representados números de $-2\,147\,483\,648$ (-2^{31}) até $2\,147\,483\,648$ ($2^{31}-1$).

Adicionalmente, o MIPS contém ainda 2 registos para poder operar com operandos de 64 bits, como acontece no caso da multiplicação e da divisão. Estes registos são designados por *hi* (de "high") e *lo* (de "low").

A unidade lógica e aritmética executa operações aritméticas binárias e inteiras, bem como operações lógicas binárias.

O "Program Counter" é um registo inicializado pelo sistema operativo com o endereço da primeira instrução do programa em memória.

Após cada instrução ter sido acedida da memória (e carregada no IR), o PC é incrementado de modo a que o CPU tenha acesso ao endereço da próxima instrução a ser executada.

A memória corresponde ao conjunto de localizações de onde a informação binária pode ser acedida e onde a informação binária pode ser guardada.

Na arquitectura MIPS, o termo "word" refere-se a uma quantidade de 32 bits, e cada localização de memória tem um endereço de 32 bits.

Todas as instruções MIPS têm 32 bits.

O "Instruction Register" é um registo de 32 bits que contém a cópia da última instrução que foi acedida.

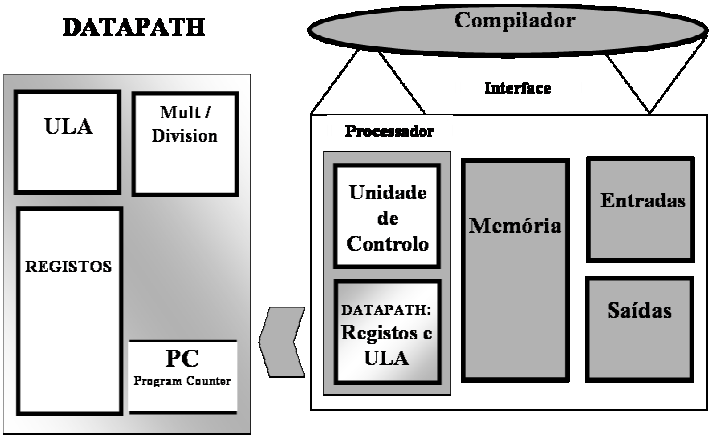


Figura 1.1 - Estrutura de um computador.

Nota: A arquitectura MIPS prevê ainda 32 registos de 32 bits para vírgula flutuante que não serão objecto de estudo nesta disciplina.

1.4 LINGUAGEM ASSEMBLY

A linguagem Assembly, que corresponde à representação simbólica da linguagem máquina, é normalmente preferida quando:

- a velocidade de execução ou tamanho da aplicação são críticos;
- é conveniente usar instruções especializadas ou algumas características do hardware;
- não há uma linguagem de alto nível disponível para o processador em uso;
- se pretende conhecer a arquitectura e organização do computador. O seu conhecimento permite que se concebam melhorias nos microprocessadores.

1.5 INTRODUÇÃO À PROGRAMAÇÃO EM ASSEMBLY

Quase todas as instruções em Assembly requerem que o programador identifique os registos que devem ser acedidos para obter os operandos ou para guardar resultados. Nesse sentido foi adoptada uma convenção que especifica que registos são apropriados para usar em cada circunstância específica.

A tabela seguinte mostra os nomes que foram dados aos registos de modo a ajudar a recordar essa convenção.

Nome	Número	Utilização
zero	0	Constante 0
at	1	Reservado para o Assemblador ("Assembler temporary")
v0 v1	2 3	Usado para retornar valores de funções
a0 a1 a2 a3	4 5 6 7	Usado para transferir os primeiros 4 argumentos para funções (os restantes usam a "stack")
t0 t1 t2 t3 t4 t5 t6 t7	8 9 10 11 12 13 14 15	Temporários (não é necessário que sejam preservados pelas funções)
s0 s1 s2 s3 s4 s5 s6 s7	16 17 18 19 20 21 22 23	"Saved temporary" (as funções devem salvaguardar e restaurar os valores nestes registos)
t8 t9	24 25	Temporários (não é necessário que sejam preservados pelas funções)
k0 k1	26 27	Reservados para o sistema operativo
gp	28	Ponteiro para a área global
sp	29	Ponteiro para "Stack" ("Stack Pointer")
fp	30	Ponteiro para "frame"
ra	31	Endereço de retorno ("Return Adress")

Tabela 1.1 – Registos de uso geral

Em código Assembly, os nomes de todos os registos começam com o símbolo $, por exemplo: $zero ($0), ou $t0 ($8).

Na arquitectura MIPS há três formatos diferentes de instruções que estão representados na seguinte figura.

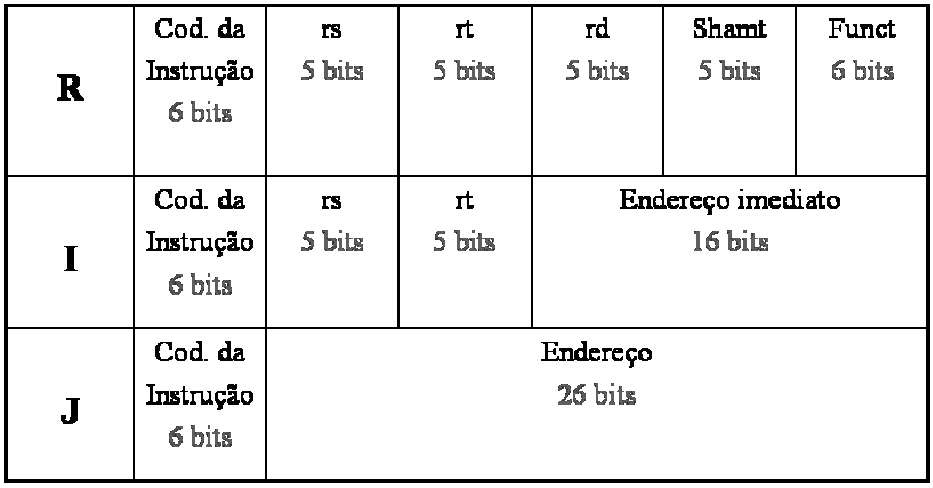


Figura 1.2 – Formatos de instruções na arquitectura MIPS

- Formato R: Formato aritmético ou formato registo.

Estas instruções começam com um campo de 6 bits (código da instrução) que deve ser preenchido com zeros, a identificação destas instruções é feita nos últimos 6 bits da "word". A localização dos operandos é identificada nos campos *rs* ("register source") e *rt* ("register target"), e a localização do registo onde vai ser guardado o resultado é definida no campo *rd* ("register destination").

O campo Shmt ("shift amount") é utilizado nas instruções de deslocamento ("shift"), que serão estudadas mais tarde, e indica o número de bits a deslocar.

Ex.:

Instrução *add $t2, $t0, $t1* (add rd, rs, rt) adiciona o conteúdo do registo *$t0* (rs) ao registo *$t1* (rt) e guarda o resultado no registo *$t2* (rd).

O código de função para esta instrução é $32_{10}=100000_2$

Os valores para cada campo da instrução em notação decimal são:

cod. inst.	rs	rt	rd	shmnt	funct
0	8[1]	9[1]	10[1]	0	32

E em notação binária (código máquina) são:

cod. inst.	rs	rt	rd	shmnt	funct
000000	01000	01001	01010	00000	100000

Ou seja: $00000001000010010101000000100000_2$

[1] Ver Tabela 1.1

E em hexadecimal: 0x01095020

- Formato I: Formato para instruções de transferência de dados ("Immediate format").

Estas instruções começam com o código da instrução que indica que operação deve ser executada. Os últimos 16 bits correspondem a um valor binário constante que é usado como operando ou como endereço a aceder. Existem ainda o campo *rs* que indica o registo que contém o outro operando e *rt* que indica em que registo deve ser guardado o resultado.

Ex.:

Instrução *addi $t2, $t1, 2* (addi rt, rs, imed.) adiciona o conteúdo do registo *$t1* (rs) ao valor constante 2 (imed.) e guarda o resultado no registo *$t2* (rt).

O código desta instrução é $8_{10}=1000_2$

Os valores para cada campo da instrução em notação decimal são:

cod. inst.	rs	rt	imed.
8	9[2]	10[2]	2

E em notação binária (código máquina) são:

cod. inst.	rs	rt	imed.
001000	01001	01010	0000000000000010

Ou seja: $00100001001010100000000000000010_2$

E em hexadecimal: 0x212A0002

- Formato J: Formato para instruções de salto incondicional ("Jump instructions").

Nestas instruções os últimos 26 bits indicam a localização em memória onde se encontra a próxima instrução a ser executada. Estas instruções serão objecto de estudo mais tarde.

1.6 INTRODUÇÃO AO SPIM

1. Para inicializar o simulador:

[2] Ver Tabela 1.1

a) Abra o simulador *PCSPIM*

b) Seleccione a opção *Simulator* do menu

c) Seleccione a opção *Settings*

d) Carregue o ficheiro *exceptions.s*

2. Para criar um programa:

a) Abra o *notepad*

b) Escreva o seu programa

c) Salve o ficheiro com extensão "*asm*" ou "*s*"

3. Para executar um programa:

a) Abra o *PCSPIM*

b) Seleccione a opção *File* do menu

c) Seleccione a opção *Open*

d) Escolha o ficheiro com o código que quer executar

e) Verifique na janela *Messages* se o programa foi carregado com sucesso

f) Seleccione a opção *Simulator*

g) Seleccione a opção *Go*

4. Para executar um programa passo a passo:

a) Repita os passos de 3a) a 3f)

b) Seleccione a opção *Single Step*

5. Janelas do simulador

Para além da consola, o *PCSPIM* apresenta quatro janelas (ver Figura 1.3).

A janela superior (*Registers*) mostra os valores nos registos. Para além dos valores dos registos de uso geral (identificados quer pelo nome quer pelo número), são ainda mostrados outros registos como o PC, *hi* e *lo* de que já falámos.

A segunda janela (*Text Segment*) mostra o segmento de texto. Os endereços onde estão carregadas as instruções são seguidos pelos códigos dessas instruções em linguagem máquina (em hexadecimal) e em linguagem Assembly.

A terceira janela (*Data Segment*) mostra o segmento de dados. São mostrados os valores guardados em endereços de memória, para além do conteúdo da Stack.

Finalmente na janela inferior (*Messages*) são mostradas mensagens que incluem mensagens de erro ou excepções.

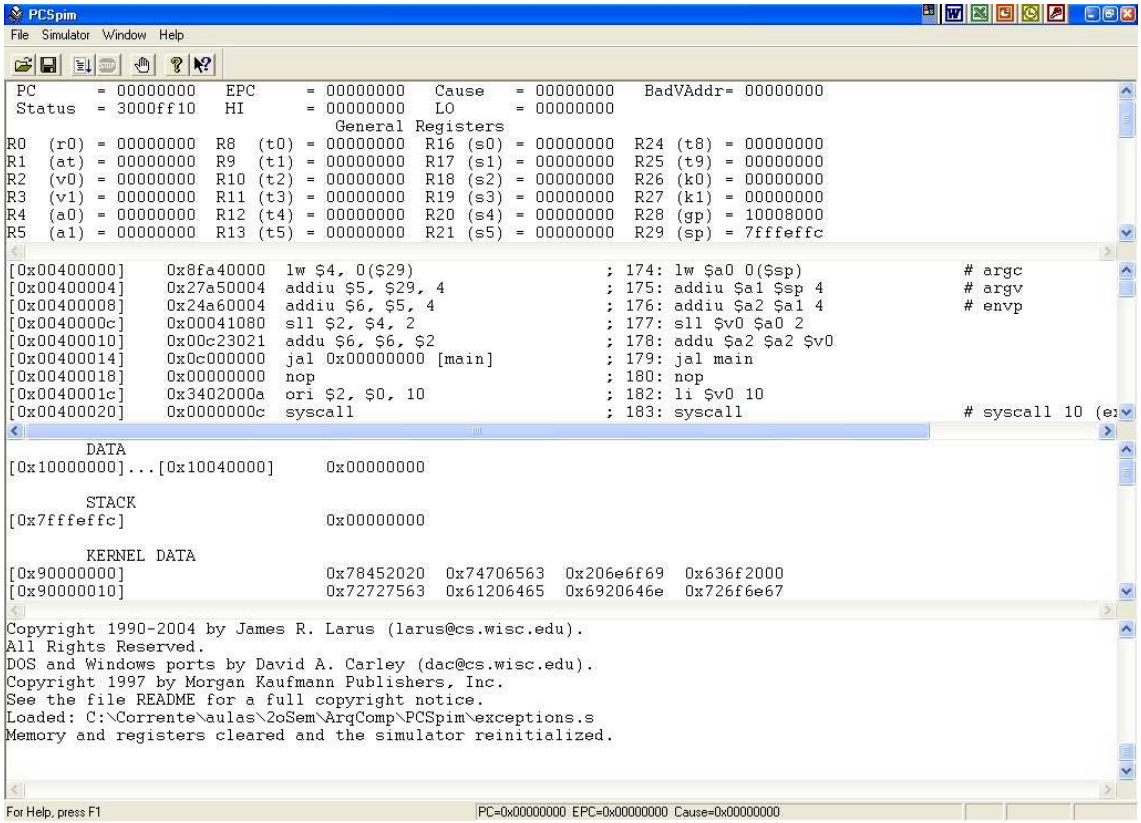


Figura 1.3 – Ambiente do *PCSPIM*

6. Requisitos em qualquer programa Assembly
   a) A etiqueta "*main:*" indica ao *assembler* o início do código a ser carregado no segmento de texto, pelo que deve constar do programa.
   b) O programa deve terminar com a instrução *jr $ra* (ou *jr $31*)

1.7 EXERCÍCIOS

1. Considere o seguinte programa

```
main:
      addi  $t0, $0, 3
      addi  $t1, $0, 1
      add   $t2, $t0, $t1
      jr    $ra
```

a) Corra o programa passo a passo enquanto vai preenchendo uma tabela com os valores encontrados nos registos.

passo	$t0	$t1	$t2
1			
2			
…			

b) O que faz este programa?

2. Faça um programa que execute o seguinte conjunto de instruções usando para isso apenas registo temporários.

```
f = 5;
g = 3;
t = f + g;
```

Execute o programa passo a passo e verifique os valores dos registos.

1.8 EXERCICIOS PROPOSTOS

1. Considere agora o programa

```
main:
      addi  $t0, $0, 3
      addi  $t1, $0, 1
      sub   $t2, $t0, $t1
      jr    $ra
```

1) Qual a função da instrução *sub*?
2) Sabendo que a instrução *sub* é uma instrução aritmética, recorrendo à janela do segmento de texto, identifique o código de função para esta instrução.

2. Faça um programa que execute o seguinte conjunto de instruções

```
g = 3;
h = 5;
i = 2;
j = 1;
f = (g+h) - (i+j);
```

3. Faça um programa que calcule:

```
f = 3*2;
```

4. Faça um programa que calcule:

$$\sum_{i=1}^{3} i$$

1.9 LEITURA RECOMENDADA

Computer Organization and Design: The Hardware/Software Interface, páginas 106 a 110.

1.10 EXERCÍCIOS ADICIONAIS

2. MEMÓRIA E TRANSFERÊNCIA DE DADOS

2.1 MEMÓRIA

Na arquitectura MIPS a memória encontra-se normalmente dividida em três partes.
A primeira parte corresponde ao segmento de texto (a partir do endereço 0x00400000), onde são guardadas as instruções do utilizador.
A segunda parte, acima do segmento de texto (ou seja a partir de 0x10000000) corresponde ao segmento de dados. Este está por sua vez dividido em duas partes, a parte dos dados estáticos e a parte dos dados dinâmicos. A memória de dados estáticos é usada para guardar dados cuja dimensão é conhecida pelo compilador e que podem ser acedidos durante toda a execução do programa. Os dados dinâmicos são guardados na segunda parte do segmento de dados. Este tipo de dados vai sendo alocado pelo programa durante a sua execução. Este segmento de memória vai sendo expandido à medida que vai sendo necessário.

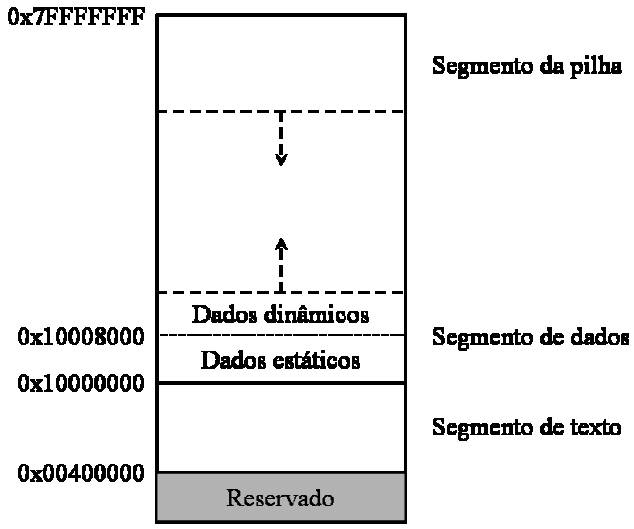


Figura 2.4 – Estrutura da memória principal

A terceira parte da memória corresponde ao segmento da pilha (começa no endereço 0x7fffffff). Tal como acontece na memória dinâmica, não é conhecida a dimensão deste segmento. À medida que o programa vai colocando valores na pilha o segmento vai sendo aumentado em direcção ao segmento de dados.

2.2 TRANSFERÊNCIA DE DADOS

A utilização de registos permite maiores velocidades de armazenamento de dados do que seria possível se se usasse apenas a memória. Para manter esta característica é necessário que eles existam em número limitado (32 no caso da arquitectura MIPS).
A memória, embora menos rápida, tem muito maior capacidade de armazenamento de dados.
Muitos programas, além de elementos de dados simples, podem conter estruturas de dados complexas (*arrays*) com mais dados do que a capacidade dos registos permite. Estas estruturas têm que ser armazenadas em memória.
Uma vez que as instruções aritméticas em MIPS operam sobre valores guardados nos registos, é necessário que no conjunto de instruções disponíveis estejam incluídas instruções de transferência de dados entre os registos e a memória.

2.3 MODOS DE ENDEREÇAMENTO

A memória pode ser interpretada como uma grande matriz unidimensional, e o endereço como um índice dessa matriz.
Assim, o endereço do terceiro elemento da memória na figura seria 2, e o valor contido nessa posição seria 10, ou seja, Memória[2]=10.

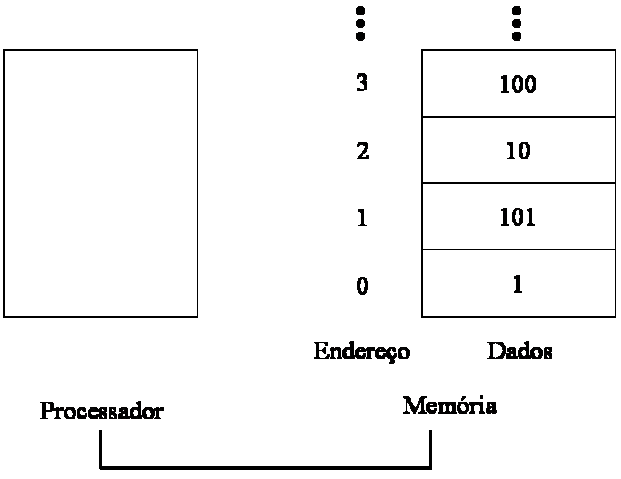


Figura 2.5 – Endereços de bytes em memória

Na realidade, cada byte (8 bits) em memória tem o seu endereço, e o MIPS é capaz de endereçar cada byte individualmente, tal como mostra a Figura 2.5.

Deste modo, o endereço de cada *word* corresponderá ao endereço de um dos bytes contido nessa *word*, ou seja os endereços de *words* sequenciais em memória diferem entre si de 4 unidades como mostra a próxima figura.

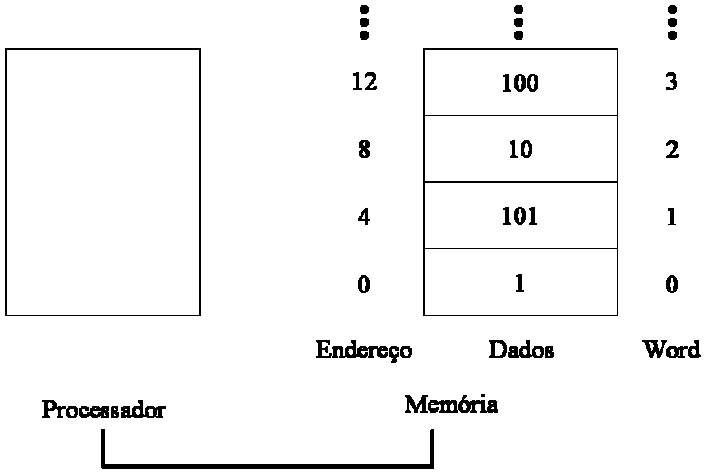


Figura 2.6 – Endereços de *words* em memória

O MIPS recorre aos dois tipos de endereçamento, por byte e por *word*.

2.4 INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

As instruções de transferência de dados entre a memória e os registos são instruções do formato I.

A instrução que permite transferir uma *word* da memória para um registo designa-se por *load* e a operação inversa, que permite transferir uma *word* de um registo para a memória designa-se por *store*.

Instrução	Descrição		Formato
lb rt,S(rs)	Carrega byte	rt = (Memória [rs+S])	I
lbu rt,S(rs)	Carrega byte sem sinal	rt = (Memória [rs+S])	I
lh rt,S(rs)	Carrega *half-word*	rt = (Memória [rs+S])	I
lhu rt,S(rs)	Carrega *half-word* sem sinal	rt = (Memória [rs+S])	I
lw rt,S(rs)	Carrega *word*	rt = (Memória [rs+S])	I
sb rt,S(rs)	Guarda byte	(Memória [rs+S])= rt	I
sh rt,S(rs)	Guarda *half-word*	(Memória [rs+S])= rt	I
sw rt,S(rs)	Guarda *word*	(Memória [rs+S])= rt	I

Nota: *S* representa o deslocamento e é uma constante.

Tabela 2.2 – Principais instruções para transferência de dados

As instruções de transferência de dados operam com endereçamento por byte.

Exemplo:

Supondo que se pretende carregar para memória o terceiro elemento de um vector (em que cada elemento tem a dimensão de uma *word*) com o valor 4.

- Em linguagem *C*:

```
A[2] = 4;
```

- Em linguagem *assembly*:

Suponha-se que o endereço do elemento A[0] está guardado em $s0 (ver Figura 2.7).

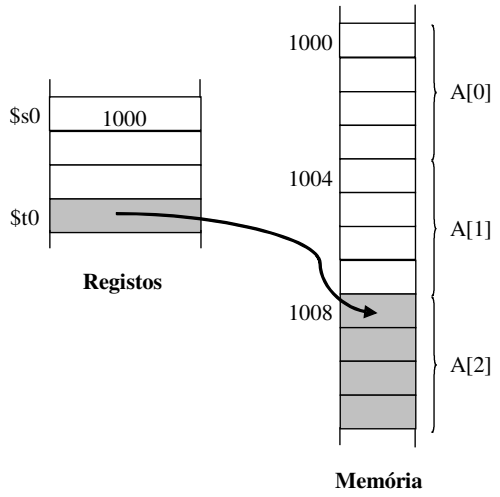


Figura 2.7 – A instrução sw (*store word*)

```
Então:
            addi $t0, $0, 4   # Carrega o valor 4 no registo $t0
            sw $t0, 8($s0)    # Guarda o valor em $t0 no
                              # endereço de memória $s0+8
```

Note-se que uma vez que estas instruções operam com endereçamento por byte, o terceiro elemento do vector deverá estar guardado 2 *words* = 2×4 bytes = 8 bytes depois do primeiro elemento (ver Figura 2.6), isto significa que o *offset* deverá ser 8.

Do mesmo modo, para transferir para um registo o conteúdo do décimo terceiro elemento do mesmo vector.

- Em linguagem *C*:

```
b=A[12];
```

- Em linguagem *assembly*:

Supondo que o endereço do elemento A[0] está guardado em $s0, então

```
lw $t0, 48($s0)   # Guarda em $t0 o valor que se
```

```
# encontra no endereço de memória
# $s0+48
```

2.5 EXERCÍCIOS

1) Faça um programa que carregue o vector $v=\{1,3,2,1,4,5\}$ para memória, isto é, que contenha a sequência de instruções MIPS correspondentes às instruções C seguintes:

```
v[0] = 1;
v[1] = 3;
v[2] = 2;
v[3] = 1;
v[4] = 4;
v[5] = 5;
```

   Assuma que o registo $s0 contém o endereço do início do vector, isto é, 0x10000100.

2) Modifique o programa anterior fazendo o acesso à memória de forma indexada. Isto é, assumindo que o registo $s0 contém o valor 0x10000100, faça um programa que contenha a sequência de instruções MIPS correspondentes às instruções C seguintes:

```
i = 0;
v[i] = 1;
i = i + 1;
v[i] = 3;
i = i + 1;
v[i] = 2;
i = i + 1;
v[i] = 1;
i = i + 1;
v[i] = 4;
i = i + 1;
v[i] = 5;
```

   Assuma que a variável *i* corresponde ao registo $s1.

3) Mostre a sequência de instruções MIPS para a seguinte instrução C:

```
v[4] = v[2] + c;
```

   Sugestão: Mostre as instruções MIPS para as seguintes instruções C:

```
b = v[2];
a = b + c;
v[4] = a;
```

2.6 INSTRUÇÕES COM OPERANDOS IMEDIATOS

As instruções com operandos imediatos, permitem ao programador utilizar constantes que não estão guardadas em registos. Estas operações são sempre do formato I.

Deste tipo de instruções as mais usadas estão representadas na tabela seguinte.

Nome	Instrução	Descrição
Add immediate	addi rt, rs, imed	Soma o valor em rs à constante indicada no campo imed e coloca o resultado em rt
Load upper immediate	lui rt, imed	Coloca a constante indicada no campo imed nos 16 bits mais significativos do registo rt
Set on less than immediate	slti rt, rs, imed	Se o valor em rs é menor do que a constante indicada no campo imed, rt toma o valor 1. Caso contrário rt toma o valor 0

Tabela 2.3 – Principais instruções com operandos imediatos

Exemplo:

Qual é o código *assembly* MIPS para carregar a constante 0x003C0900 para um registo de 32 bits?

O valor 0x003C0900 tem a seguinte representação em notação binária:

0000 0000 0011 1100 0000 1001 0000 0000

A instrução *addi*, só permite passar para um registo valores de 16 bits, que ficam colocados nos 16 bits menos significativos do registo.

Para se carregar um número de mais de 16 bits, é necessário fazê-lo em duas operações[3].

Os 16 bits mais significativos do valor 0x003C0900 são:

0000 0000 0011 1100 (correspondendo a 0x003C)

e os 16 bits menos significativos são:

0000 1001 0000 0000 (correspondendo a 0x0900)

Os 16 bits mais significativos podem ser carregados para o registo através da instrução *lui*:

```
lui $t0, 0x003C
```

ficando o registo $t0 com o seguinte conteúdo:

0000 0000 0011 1100 0000 0000 0000 0000 (correspondendo a 0x003C0000)

[3] As pseudo-instruções *load address* (la rd, imed) e *load immediate* (li imed) permitem fazer uma operação deste tipo de uma só vez.

Somando agora os 16 bits menos significativos ao conteúdo do registo $t0, ou seja fazendo

0000 0000 0011 1100 0000 0000 0000 0000 + 0000 0000 0000 0000 0000 1001 0000 0000

o que corresponde a 0x003C0000 + 0x00000900 = 0x003C0900

ou seja:

```
addi $t0, $t0, 0x0900
```

2.7 EXERCÍCIOS

1) Modifique o exercício 1 em 2.5 fazendo com que o primeiro elemento do vector seja colocado no endereço 0x10000100.
2) Experimente o código na aula.

2.8 EXERCÍCIOS PROPOSTOS

1) Faça um programa que:
   a) Carregue o vector v = {1,2,3,4,5} para memória. O primeiro elemento do vector deve ser colocado no endereço 0x10000100.
   b) Substitua todos os elementos pelo seu dobro.
   c) Corra o programa passo a passo e observe o segmento de dados, registando no relatório essa execução.
2) Faça um programa que:
   a) Carregue o vector v = {2,5,4,1} para memória. O primeiro elemento do vector deve ser colocado no endereço 0x10000100.
   b) Copie os elementos do vector *v* para um outro vector, *u*, cujo primeiro elemento tem o endereço 0x10000110.
3) Utilizando o programa MIPS no exercício 2 b), determine, para cada instrução, o formato da instrução e os valores decimais de cada um dos seus campos.

   Nota: Considere apenas as seis primeiras instruções.

2.9 LEITURA RECOMENDADA

Computer Organization and Design: The Hardware/Software Interface, páginas 110 a 122.

2.10 EXERCÍCIOS ADICIONAIS

1) Mostre a sequência de instruções MIPS para a seguinte instrução C:

```
v[i] = h + v[i];
```

Sugestão: Mostre as instruções MIPS para as seguintes instruções C:

```
b = v[i];
a = h + b;
v[i] = a;
```

2) Prepare uma tabela com o seguinte formato para as instruções que conhece:

Categoria	**Instrução**	**Exemplo**	**Significado**	**Formato**
Aritmética	add	add $1,$2,$3	$1 = $2 + $3	R

0	2	3	1	0	32

Nota: A instrução *add* é apresentada como exemplo. Na 3 ª linha é apresentado o código máquina correspondente ao exemplo apresentado na 2ª linha.

3. SALTOS CONDICIONAIS E INCONDICIONAIS

3.1 INSTRUÇÕES DE SALTOS CONDICIONAIS

As instruções de saltos condicionais permitem tomar decisões.

No MIPS as duas principais instruções de tomada de decisão são designadas por *branch if equal* e por *branch if not equal*. São ambas do formato I.

A primeira, *branch if equal*, é da forma:

beq rs, rt, label

e no caso do valor em *rs* ser igual ao valor em *rt*, então a próxima instrução a ser executada está no endereço associado à *label* indicada. No caso dos valores em *rs* e *rt* serem diferentes o programa continua a sua execução normal.

A instrução *branch if not equal* é da forma:

bne rs, rt, label

e no caso de o valor em *rs* ser diferente do valor em *rt*, a próxima instrução a ser executada está no endereço associado à *label* indicada. No caso dos valores em *rs* e *rt* serem iguais o programa continua a sua execução normal.

Neste caso usa-se um endereçamento por byte relativo ao *Program Counter* (PC), isto é, o próximo valor a constar no PC é dado pelo valor actual em PC somado ao quádruplo do valor indicado em *label*.

Exemplo:

O extracto de programa em *C*

```
if (i==j)
      {h=i+j;
      g=i+1;}
else
      h=i+j;
```

pode ser traduzido para *assembly* como (assumindo que as variáveis `g`, `h`, `i`, `j` correspondem aos registos `$t3`, `$t4`, `$t1`, `$t2`):

```
      bne $t1, $t2,  L1
      addi $t3, $t1, 1
L1:   add $t4, $t1, $t2
```

A tradução deste extracto de programa para linguagem máquina permite exemplificar o endereçamento relativo ao PC.

Em notação decimal:

Endereço	cod. inst.	rs	rt	rd	shmnt	funct	instrução
0x00400000	5	9	10	2^4			bne $t1, $t2, L1
0x00400004	8	9	11	1			addi $t3, $t1, 1
0x00400008	0	9	10	12	0	32	add $t4, $t1, $t2

Em notação binária:

Endereço	cod. inst.	rs	rt	rd	shmnt	funct	instrução
0x00400000	000101	01001	01010	0000000000000010			bne $t1, $t2, L1
0x00400004	001000	01001	01011	0000000000000001			addi $t3, $t1, 1
0x00400008	000000	01001	01010	01100	00000	100000	add $t4, $t1, $t2

[4] Note-se que L1 está a 2 *words* deste instrução. Nesta altura o valor em PC é 0x00400000. O próximo valor em PC será 0x00400000 + (4×2) = 0x00400008.

Outra instrução de que faz sentido referir nesta altura é *set on less than*, uma instrução do formato R, que é da forma

slt rd, rs, rt

em que se o valor em rs for menor do que o valor em *rt*, *rd* toma o valor 1, caso contrário *rd* toma o valor 0.

A instrução `slt` permite suportar todas as condições das linguagens de alto nível em que apareçam os operadores > e <.

Exemplo:

O extracto de programa em *C*

```
if (i<j)
      h=i+j;
```

pode ser traduzido para *assembly* como (assumindo que as variáveis `h`, `i`, `j` correspondem aos registos `$t4`, `$t1`, `$t2`):

```
      slt $t3, $t1,  $t2      # se (i<j) então $t3 <- 1
                              # se ~(i<j) então $t3 <- 0
      beq $t3, $0, FimIf
      add $t4, $t1, $t2
FimIf:
```

Exemplo:

O extracto de programa em *C*

```
if (i≤j)
      h=i+j;
```

pode ser traduzido para *assembly* como (assumindo que as variáveis `h`, `i`, `j` correspondem aos registos `$t4`, `$t1`, `$t2`):

```
      slt $t3, $t2,  $t1      # se (i>j) então $t3 <- 1
                              # se ~(i>j) então $t3 <- 0
      bne $t3, $0, FimIf
      add $t4, $t1, $t2
FimIf:
```

3.2 EXERCÍCIOS

1) Reproduza cada um dos seguintes programas para *assembly*, atribuindo às variáveis valores que lhe permitam testar o seu programa.

```
a) if (a>b)
         a=a+1;
b) if (a≥b)
         b=b+1;

c) if (a≤b)
```

```
          a=a+1;
d) if (a==b)
      b=a;
```

3.3 INSTRUÇÕES DE SALTOS INCONDICIONAIS

Existem quatro instruções de saltos incondicionais como mostra a tabela seguinte.

Nome	Instrução	Descrição	Formato
Jump	j *label*	Salta para *label*	J
Jump and link	jal *label*	Salta para *label* e guarda o endereço da próxima instrução em $ra	J
Jump and link register	jalr rs, rd	Salta para endereço em rs, e guarda o endereço da próxima instrução em rd	R
Jump register	jr rs	Salta para o endereço em rs	R

Tabela 3.4 – Instruções de saltos incondicionais

As instruções *j* e *jal*, são do formato J, e portanto dispõem de 26 bits para indicar o endereço (ver Guia da 1ª aula). Estas instruções operam com endereçamento por *word*.

Se este endereço fosse expresso em bytes, com 26 bits apenas se poderiam considerar endereços até 64 *Mbytes*. Recorrendo ao endereçamento por *word* já é possível considerar endereços até 256 *Mbytes*.

Exemplo:

O endereçamento por *word* pode ser demonstrado traduzindo o seguinte segmento de linguagem *assembly* para linguagem máquina

```
          j exit
exit: jr $ra
```

Em linguagem máquina (notação decimal):

Endereço	cod. inst.	rs	rt	rd	shmnt	funct	instrução
0x0040000c	2	400010 / 4 = 100004					j exit
0x00400010	0	31	0	0	0	8	jr $ra

Em notação binária:

Endereço	cod. inst.	rs	rt	rd	shmnt	funct	instrução
0x0040000c	000010	00000100000000000000000100					j exit

0x00400010	000000	11111	00000	00000	00000	001000	jr $ra

A instrução `jump` permite suportar as instruções `if ... then ... else` nas linguagens de alto nível.

Exemplo:

O extracto de programa em *C*

```
if (i==j)
      h=i-j;
else
      h=i+j;
```

pode ser traduzido para *assembly* como (assumindo que as variáveis `h`, `i`, `j` correspondem aos registos `$t4`, `$t1`, `$t2`):

```
      bne $t1, $t2,  Else
      sub $t4, $t1, $t2
      j FimIf
Else:
      add $t4, $t1, $t2
FimIf:
```

Finalmente, a instrução `jump` também permite suportar os ciclos nas linguagens de alto nível.

Exemplo:

O extracto de programa em *C*

```
while (i==j)
      i=i+j;
```

pode ser traduzido para *assembly* como (assumindo que as variáveis `i`, `j` correspondem aos registos `$t1`, `$t2`):

```
While:
      bne $t1, $t2,  FimWhile
      add $t1, $t1, $t2
      j While
FimWhile:
```

A instrução `jump register`, `jr`, permite suportar de forma eficiente os comandos `case` nas linguagens de alto nível.

Exemplo:

O extracto de programa em *C*

```
switch (k) {
      case 0: i=i+j; break;
      case 1: i=i-j; break;
      case 2: i=i+h; break;
      case 3: i=i-h; break;
}
```

pode ser traduzido para *assembly* como (assumindo que as variáveis `h`, `i`, `j`, `k` correspondem aos registos `$t0`, `$t1`, `$t2`, `$t3`):

```
        add $t5, $t3, $t3       # $t5 = 4 × k
        add $t5, $t5, $t5
        add $t4, $s0, $t5
        lw $t3, 0($t4)          # $t3 = Endereçok
        jr $t3
Case0:
        add $t1, $t1, $t2
        j FimSwitch
Case1:
        sub $t1, $t1, $t2
        j FimSwitch
Case2:
        add $t1, $t1, $t0
        j FimSwitch
Case3:
        sub $t1, $t1, $t0
FimSwitch:
```

Nota: Assuma que o registo `$s0` (ver Figura 3.8) contém o endereço de início de um vector `t`, cujos elementos `t[0]` a `t[3]` são os endereços onde se encontram, respectivamente, as instruções correspondentes a cada uma das clausulas `case`.

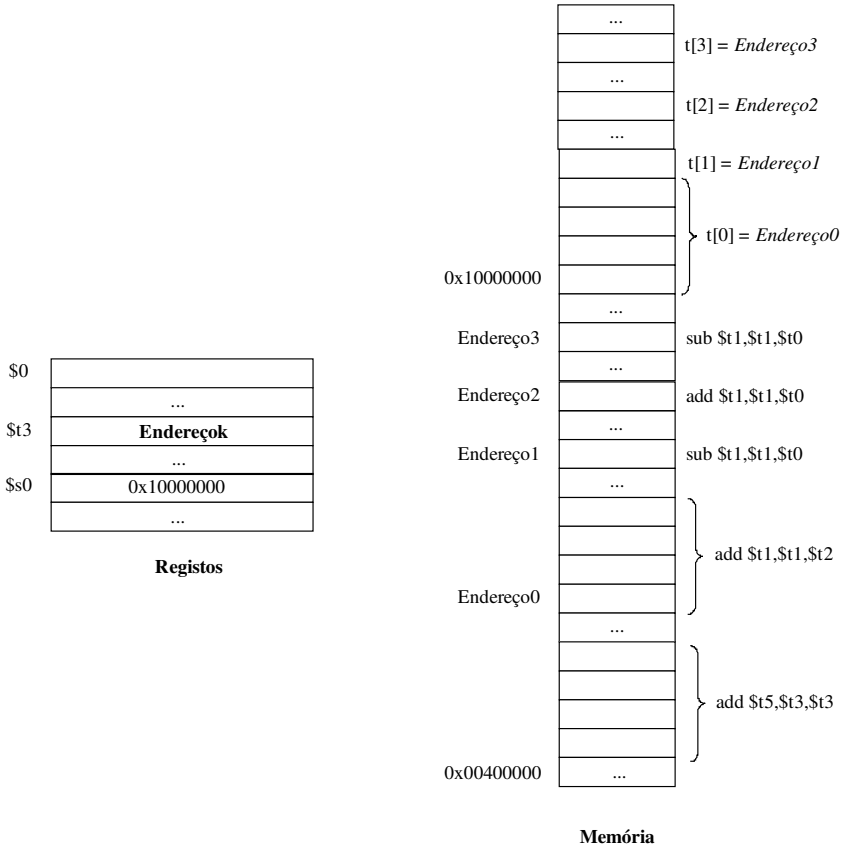


Figura 3.8 – O comando *switch*

3.4 EXERCÍCIOS

1. Reproduza cada um dos seguintes programas para *assembly*, atribuindo às variáveis valores que lhe permitam testar o seu programa.

```
1. if (a==b)
         a=a+1;
   else
         b=b+1;

2. if (a<b)
         a=a+1;
   else
         b=b+1;

3. a=0;b=0;c=5
   while (a<c)
         {a=a+1;
          b=b+c;}

4. a=0;b=0;
   for (i=0; i<5 ;i++)
         {a=a+1;
          b=b+2;}
```

3.5 EXERCÍCIOS PROPOSTOS

1. Experimente os programas dos exercícios referidos em 3.2 e 3.4.
2. Reproduza cada um dos seguintes programas para *assembly*, atribuindo às variáveis valores que lhe permitam testar o seu programa.

```
a) while (v[i]==k)
         i=i+j;
```

Nota: O código anterior é equivalente a:

```
a = v[i];
while(a==k) {
      i=i+j;
      a = v[i];
}
```

0. Faça um programa que, considerando o vector v = {1,2,3,4,5} no endereço 0x10000100:
   a) Calcule a soma dos elementos do vector;
   b) Adicione 5 aos elementos do vector com valor 1.

   Sugestão:

```
for(i=0;i<5;i++)
```

```
if (v[i]==1)
        v[i] = v[i] + 5;
```

3.6 EXERCÍCIOS ADICIONAIS

1. Faça um programa que, considerando o vector v = {2,5,4,1} no endereço 0x10000100:
   a) Copie os elementos do vector *v* para um outro vector, *u*, cujo primeiro elemento tem o endereço 0x10000120.

      Sugestão:

      ```
      for(i=0;i<4;i++)
            u[i] = v[i];
      ```

2. Considere o último exemplo da secção 3.3. Introduza a resolução no SPIM.
   a) Qual o valor dos endereços *Endereço0* ... *Endereço3* na Figura 3.8?
   b) Inicialize a variável k com o valor 2. O que faz a instrução `jr $t3`?
3. Assuma que as variáveis `a`, `b`, `c` correspondem aos registos `$t0`, `$t1` e `$t2`. Qual é o correspondente código MIPS?

   ```
   switch(a){
      case 1: b=c+1; break;
      case 2: b=c+2; break;
      default: b=c; break;
   }
   ```

4. Actualize a tabela do exercício 2.10.2 com as instruções seguintes: `beq`, `bne`, `slt`, `j`, `jr`.

4. PROCEDIMENTOS

A utilização de procedimentos permite:

- Estruturar os programas de modo a facilitar a sua leitura;
- Tornar o código reutilizável;
- Concentrar a atenção em uma tarefa de cada vez.

Considerando o programa principal e o procedimento seguintes:

```
int calculaExp(int g,int h,
                  int i,int j)
{
int f;
   f = (g + h)-(i + j);
   return f;
}
```

```
void main( ) {
int a;
    a = 0;
      …
      a       =       a       +
calculaExp(1,2,3,4);
      …
}
```

pretende-se, nas secções seguintes, obter os correspondentes códigos assembly (assumindo que as variáveis *a*, *g*, `h`, `i`, `j`, `f` correspondem aos registos `$s7`, `$a0`, `$a1`, `$a2`, `$a3`, `$v0`).

4.1 O SEGMENTO DA PILHA

Já foi referido em aulas anteriores que o segmento da pilha (*stack segment*) é um dos três segmentos de memória que são atribuídos pelo sistema operativo ao programa quando este é carregado para memória.

O segmento da pilha é a área para a qual os parâmetros podem ser passados, onde pode ser alocado espaço para as variáveis usadas pelos procedimentos e onde são guardados endereços de retorno para chamadas a procedimentos, procedimentos aninhados e a funções recursivas.

Através do uso desta área de memória é possível escrever programas sem preocupação com o número de parâmetros a passar.

O sistema operativo inicializa o registo $29 (*$sp*) com o endereço base do segmento da pilha.

Relembre-se ainda que a pilha (ver Figura 3.8) cresce em direcção aos endereços mais baixos. À medida que se inserem valores ou retiram valores da pilha, o registo

`$sp` vai sendo modificado de forma a conter o endereço do *topo da pilha*, isto é, a última posição de memória ocupada.

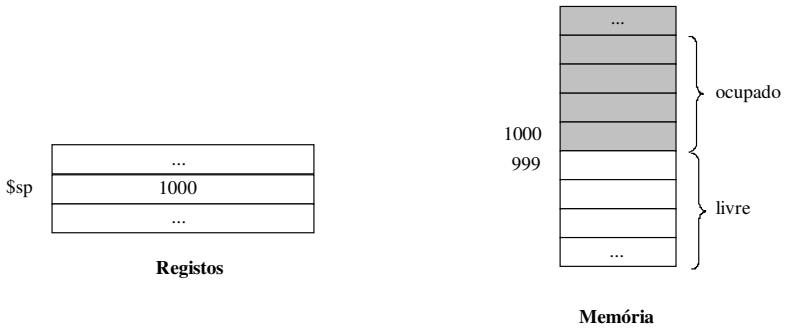


Figura 4.9 – A pilha

4.2 INVOCAÇÃO E RETORNO

4.2.1 INVOCAÇÃO DE UM PROCEDIMENTO

A chamada a um procedimento é feita através da instrução `jump and link`, `jal`, seguida pelo nome do procedimento, o qual corresponde ao endereço da primeira instrução do procedimento.

Exemplo:

```
main:                    calculaExp:
            ...                          add $t0,
            jal
   calculaExp               $a0, $a1
                                         add $t1,
            ...
            jr $ra           $a2, $a3
                                         sub $v0,
                             $t0, $t1
                                    jr $ra
```

A instrução `jal` realiza duas tarefas:

1. Armazena o endereço da instrução seguinte à que produz a chamada (*endereço de retorno*) no registo `$ra`;
2. Salta para a primeira instrução do procedimento.

4.2.2 RETORNO DE UM PROCEDIMENTO

A última instrução de um procedimento (a instrução `jr $ra`) corresponde a um salto para o endereço de retorno. jr significa `Jump register`, isto é, salta para o endereço contido no registo em causa.

Exemplo:

O programa principal e o procedimento `calculaExp` têm o seguinte código:

```
main:                    calculaExp:
            ...                                  add $t0,
            jal              $a0, $a1
   calculaExp                                    add $t1,
            ...              $a2, $a3
            jr $ra                               sub $v0,
                             $t0, $t1
                                        jr $ra
```

4.3 PASSAGEM DE PARÂMETROS

4.3.1 PARÂMETROS DE ENTRADA

Os parâmetros de entrada são armazenados em registos $a.

Exemplo:

Neste exemplo os parâmetros de entrada são armazenados nos registos `$a0` a `$a3`:

```
main:
            ...
            addi $a0, $0, 1          # $a0 = 1
            addi $a1, $0, 2          # $a1 = 2
            addi $a2, $0, 3          # $a2 = 3
            addi $a3, $0, 4          # $a3 = 4
            jal calculaExp
            ...
            jr $ra
```

4.3.2 PARÂMETROS DE SAÍDA

Os parâmetros de saída também são armazenados em registos, mas $v.

Exemplo:

Neste exemplo, o parâmetro de saída é armazenado no registo `$v0`. Isto é, o procedimento coloca o valor em `$v0` antes de retornar e o programa principal vai buscar o valor que foi colocado em `$v0`.

```
calculaExp:                     main:
            add $t0, $a0, $a1               ...
            add $t1, $a2, $a3               addi $a0, $0, 1
            sub $v0, $t0, $t1               addi $a1, $0, 2
        jr $ra                              addi $a2, $0, 3
                                            addi $a3, $0, 4
                                            jal calculaExp
                                            add $s7, $s7, $v0    # a = a +
                                                                 #
                                    calculaExp(...)
                                            ...
                                            jr $ra
```

4.4 COMENTÁRIOS

Além dos comentários habituais, um procedimento deve ter um cabeçalho com a informação necessária para a utilização do procedimento. Assim, o cabeçalho deve conter a seguinte informação do procedimento:

1. Nome;
2. Breve descrição;
3. Breve descrição dos parâmetros de entrada e saída.

Exemplo: Assim, temos para o procedimento `calculaExp`:

```
            # procedimento calculaExp
            # descrição: calcula o valor de uma expressão numérica
            # parâmetros de entrada: $a0, $a1, $a2, $a3
            # parâmetro de saída: $v0
        calculaExp:
                        add $t0, $a0, $a1
                        add $t1, $a2, $a3
                        sub $v0, $t0, $t1
                jr $ra
```

4.5 PRESERVAÇÃO DOS REGISTOS

Um procedimento comporta-se como se fosse um *espião*, isto é, quando termina de realizar a sua tarefa não deve deixar qualquer rasto. Assim, os valores dos registos e da memória devem ser iguais antes do procedimento se iniciar e depois do procedimento ter terminado.

Para se comportar desta forma, o procedimento, antes de modificar o valor de um registo que pudesse estar a ser utilizado pelo programa principal e não devesse ser alterado, deve guardar o valor desse registo e, antes de retornar para o programa principal, deve voltar a colocar o valor original no registo.

Os valores dos registos são guardados na pilha (ver Figura 4.10 b)). A explicação para o facto de se utilizar a pilha pode ser compreendida facilmente mais à frente quando se examinar os procedimentos recursivos.

Exemplo:

A preservação dos registos `$t0` e `$t1` adiciona seis novas instruções ao procedimento:

```
calculaExp:
     addi $sp, $sp, -8       # modificar topo da pilha
     sw $s0, 4($sp)                # guardar $s0
     sw $s1, 0($sp)                # guardar $s1
     add $s0, $a0, $a1
     add $s1, $a2, $a3
     sub $v0, $s0, $s1
     lw $s1, 0($sp)                # restaurar $s1
     lw $s0, 4($sp)                # restaurar $s0

     addi $sp, $sp, 8        # actualizar topo da pilha
     jr $ra
```

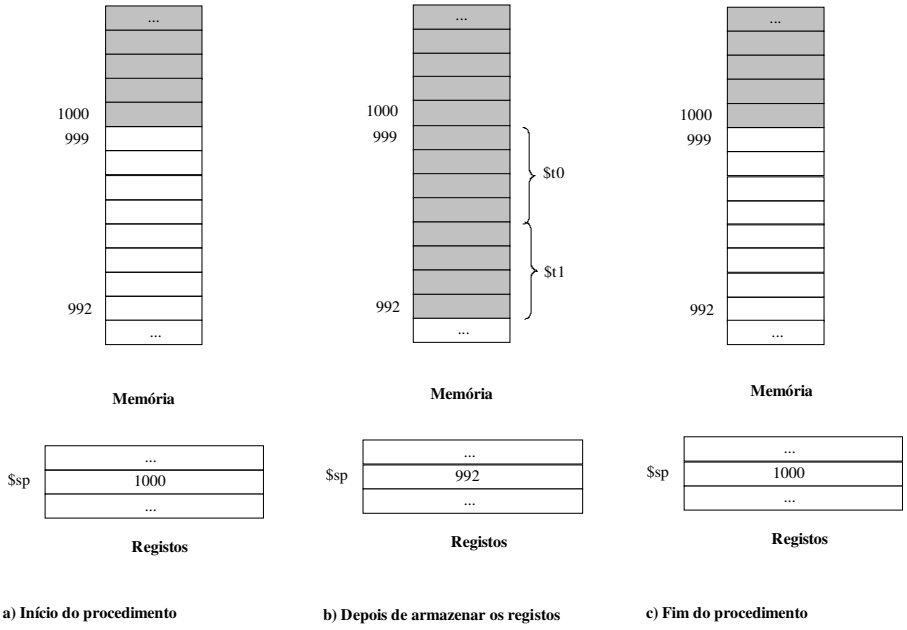


Figura 4.10 – Evolução da pilha durante o procedimento *calculaExp*

4.6 EXERCÍCIOS

1. O SPIM tem na sua memória um programa, *programa base*, o qual invoca o procedimento *main*.
    a) Crie um programa que seja constituído pelo procedimentos `main` e `calculaExp` atrás.
    b) Execute o programa.
    c) Observe que o programa entrou em ciclo infinito, o que o impede de terminar. De facto, a execução da instrução `jr $ra` (ver *main*) deveria fazer com que o programa terminasse, visto que o registo `$ra` guarda o endereço de retorno para o programa base. Acontece que a instrução `jal calculaExp` guarda no registo `$ra` o endereço de retorno do procedimento, perdendo-se assim o anterior valor de `$ra`, isto é, o endereço de retorno para o programa base.

2. A solução para o problema descrito em 1c) é o procedimento *main* preservar o registo `$ra`, da mesma forma que o procedimento `calculaExp` preserva os registos.
    a) Termine o procedimento *main* adicionando-lhe as instruções correspondentes aos dois comentários.

```
main:
            # guardar $ra na stack
      addi $a0, $0, 1
      addi $a1, $0, 2
      addi $a2, $0, 3
      addi $a3, $0, 4
      jal calculaExp
      add $s7, $s7, $v0
            # restaurar $ra
      jr $ra
```

b) Execute o programa, verificando que o problema anterior foi resolvido.

4.7 USO DE CONVENÇÕES

Pode acontecer que os diferentes procedimentos que compõem um programa sejam efectuados por um só programador. No entanto, geralmente, os procedimentos são efectuados por diferentes entidades. Por exemplo, um procedimento pode ser efectuado por um programador e outro procedimento ter sido gerado por um compilador. Neste caso, as diferentes entidades têm de usar as mesmas convenções de forma a que os respectivos procedimentos possam comunicar entre eles.

Assim, convencionou-se que:

- Os registos *$a0* a *$a3* são utilizados para passar parâmetros para os procedimentos;
- Os registos *$v0* e *$v1* são utilizados para passar resultados para o procedimento que chamou o procedimento;
- O registo *$sp* é utilizado como apontador para o topo da pilha;
- Os registos *$s0* a *$s7* são preservados por um procedimento;
- Os registos *$t0* a *$t9* não necessitam de ser preservados por um procedimento.

Exemplo:

Utilizando as convenções, as instruções do procedimento `calculaExp` necessárias para preservar os registos `$s0` e `$s1` podem ser retiradas, usando os registos `$t0` e `$t1` para os valores temporários. Assim, temos:

```
      # procedimento calculaExp
```

```
     # descrição: calcula o valor de uma expressão numérica
     # parâmetros de entrada: $a0, $a1, $a2, $a3
     # parâmetro de saída: $v0
calculaExp:
                add $t0, $a0, $a1
                add $t1, $a2, $a3
                sub $v0, $t0, $t1
       jr $ra
```

4.8 EXERCÍCIO

1. Considere o seguinte procedimento C:

```
swap(int v[], int k) {
        int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

   a) Qual o código MIPS para o procedimento (assuma que o registo `$t0` é associado à variável `temp`)?

      Nota: Considere que o registo `$a0` contém o endereço do elemento `v`[0] e que o registo `$a1` contém o valor da variável `k`.

   b) Reescreva o procedimento, utilizando as convenções. Qual a diferença relativamente ao número de instruções?

   c) Experimente o código na aula.

4.9 EXERCÍCIOS PROPOSTOS

1. Elabore:

   a) Um procedimento que calcule a soma dos elementos de um vector `v`. O procedimento tem como parâmetros de entrada o endereço do elemento `v`[0] e o número de elementos do vector. O parâmetro de saída é a soma dos elementos do vector.

   b) O programa principal de forma a testar o procedimento efectuado, inicializando o vector v = {1,2,3,4,5} no endereço 0x10000100.

2. Construa:

a) Um procedimento que copie os elementos de um vector *v* para um outro vector *u*. O procedimento tem como parâmetros de entrada os endereços dos elementos `v`[0] e `u`[0], bem como o número de elementos a copiar.

b) O programa principal de forma a testar o procedimento efectuado, inicializando o vector `v` = {1,2,3,4,5} no endereço 0x10000100 e considerando que `u`[0] tem o endereço 0x10000120.

4.10 EXERCÍCIOS ADICIONAIS

1. Considere o seguinte procedimento C, o qual calcula o maior elemento de um vector `v` com `n` elementos.

```
int máximo(int v[], int n) {
        int max = v[0];
        for(i = 1; i < n; i++) {
              if (v[i] > max)   max = v[i];
}
```

   a) Qual o código MIPS correspondente?

   b) Teste o procedimento efectuado, considerando um programa principal que inicializa o vector v = {1,2,3,4,5} no endereço 0x10000100.

2. Actualize a tabela do exercício 2.10.2 com as instruções `jal` e `jr`.

4.11 QUESTÕES

2) Qual é a diferença entre Assembly e código de máquina? Justifique a sua afirmação, exemplificando se necessário.
3) Explique por palavras suas a estrutura de memória do MIPS.
4) O que entende por endereçamento por *byte* ou por *word*? Exemplifique.

5. AVALIAÇÃO

Considerando que a quinta semana poderá ser de avaliação, não se avança na apresentação de conceitos relacionados com a linguagem Assembly. Esta aula será para todos, uma aula de exercícios.

Aqueles que optarem pela avaliação em frequência devem ter os relatórios completos (incluindo os EXERCÍCIOS), sendo avaliada a resolução dos EXERCÍCIOS PROPOSTOS.

5.1 EXERCÍCIOS

1. Usando um procedimento, converta o segmento de código *C* para linguagem A*ssembly*:

```
if (a == 0)
      for (i = 0; i <= 3; i=i+1) B[i]=A[i];
else
      for (i = 0; i <= 3; i=i+1) B[i]=A[3-i];
```

   Teste o programa supondo que A[0] corresponde ao endereço 0x10000000 e que B[0] está 400 *words* depois de A[0]. Suponha que $A = \{20, 21, 22, 23\}$.

   Dê valores de 0 ou 1 à variável a para testar o seu programa.

5.2 EXERCÍCIOS PROPOSTOS

1. Considere o seguinte código em C para uma primeira aproximação a uma função que calcula iterativamente a soma dos números inteiros até um número dado.

```
int integ (int n){
int i, res;
      res=1;
      for (i=2; i<=n; i++)
            res = res + i;
return (res);
}
```

a) Codifique em *assembly* essa função.

b) Faça um programa que chame a função para calcular a soma até 5 (cinco). Corra o programa passo a passo enquanto vai preenchendo uma tabela semelhante à usada no exercício 1.7.1 com os valores encontrados nos registos usados no procedimento.

2. Considere o seguinte procedimento em C, que calcula o menor elemento de um vector `v` com `n` elementos.

```
int menor(int v[], int n) {
int min = v[0];
for(i = 1; i < n; i++) {
                  if (v[i] < min)   min = v[i];
}
```

a) Codifique em assembly essa função.

b) Teste o procedimento efectuado, considerando um programa principal que inicializa o vector v = {1,2,3,4,5} no endereço 0x10000100.

5.3 EXERCÍCIOS ADICIONAIS

1. Actualize a tabela com as novas instruções que conheceu, mantendo o formato proposto em 2.10.
2. Num diagrama, ilustre o esquema de utilização da memória no exercício 5.2.1 (factorial iterativo):

a. Antes da chamada da função
b. Durante a execução da função
c. Após a chamada da função

3. Faça uma função que calcule a soma dos elementos de um vector como os usados em 5.2.2.

6. OPERAÇÕES ARITMÉTICAS E LÓGICAS

Para algumas aplicações ou para simplificar e tornar mais rápidas algumas operações é muitas vezes conveniente aceder ou operar sobre apenas alguns dos bits numa *word*. Existem algumas instruções em *assembly* que permitem actuar sobre apenas alguns bits.

6.1 MULTIPLICAÇÃO E DIVISÃO

Os registos `hi` e `lo` armazenam o resultado das operações de multiplicação e de divisão.

A instrução `mult rs, rt` multiplica os conteúdos dos registos `rs` e `rt` e armazena:

1. Os 32 bits menos significativos do produto no registo `lo`;
2. Os 32 bits mais significativos do produto no registo `hi`.

A instrução `div rs, rt` divide o conteúdo do registo `rs` pelo conteúdo do registo `rt` e armazena:

1. O quociente no registo `lo`;
2. O resto no registo `hi`.

Também, existem instruções que permitem armazenar num registo os valores contidos em `hi` ou `lo`. Assim, a instrução `mfhi rd` (move from high) copia o conteúdo de `hi` para o registo `rd`, e a instrução `mflo rd` (move from low) copia o conteúdo de `lo` para `rd`.

Todas as instruções acima mencionadas têm o formato R.

Exemplo:

Assumindo que a variável `a` corresponde ao registo `$t2`, o valor da expressão

```
a = 3 * 2
```

pode ser calculado utilizando o seguinte código MIPS:

```
addi $t0,$0,2
addi $t1,$0,3
mult $t0,$t1
mflo $t2
```

6.2 EXERCICIOS

1. Assumindo que as variáveis `f` e `g` correspondem aos registos `$t0` e `$t1`:
   a) Escreva código MIPS para calcular os valores das seguintes expressões:
   ```
   f = 11 / 2;
   g = 11 % 2;
   ```
   b) O número 11 é divisível por 2? Justifique.
2. Escreva um procedimento que multiplique dois números positivos e verifique se o resultado cabe em 32 bits. Os números são passados nos registos `$a0` e `$a1`. O resultado é retornado em `$v0` e o registo `$v1` deverá retornar o valor 1 se o resultado colocado em `$v0` estiver correcto.

6.3 DESLOCAMENTOS

As operações lógicas de deslocamento permitem deslocar bits dentro duma word.

A instrução *shift left logical*, `sll`, permite deslocar bits para a esquerda, e é da forma *sll rd, rt, shmnt*. Os bits no registo *rt* sofrem um deslocamento de `shmnt` para a esquerda e o resultado é guardado no registo *rd*.

Exemplo:

Se no registo $t1 estiver o valor 53, ao fazer

```
sll $t2, $t1, 8
```

o registo $t2 ficará com o valor 53×2^8, isto é, 13568. Isto corresponde à operação:

00000000000000000000000000110101 << 8 = 00000000000000000011010100000000.

Do mesmo modo, a instrução *shift right logical*, `srl`, permite deslocar bits para a direita, e é da forma *srl rd, rt, shmnt*. O valor que está no registo *rt* sofre um deslocamento de shmnt para a direita e o resultado é guardado no registo *rd*.

Exemplo:

Se no registo $t1 estiver o valor 53, ao fazer

```
srl $t2, $t1, 4
```

o registo $t2 ficará com o valor $53/2^4$, isto é, 3. Isto corresponde à operação:

00000000000000000000000000110101 >> 4 = 00000000000000000000000000000011

As instruções `sll` e `srl` têm o formato R e o número de bits a deslocar é colocado no campo *shmnt*.

6.4 EXERCÍCIOS

1. Inicialize o registo `$t0` com um valor à sua escolha.
   a) Qual o valor do registo $t0 após a instrução `sll $t0,$t0,2`?
   b) Que operação aritmética foi realizada pela instrução anterior?
   c) Resolva o exercício 6.2.1 não utilizando a instrução `mult`.
2. Relativamente à instrução `sll $t0,$t0,2`:
   a) Qual o seu formato de instrução?
   b) Partindo da correspondente instrução em linguagem máquina, determine o valor do campo *shmnt*?

6.5 OPERAÇÕES LÓGICAS

Os operadores lógicos AND, OR e XOR também têm representação em *assembly* MIPS R2000. As tabelas da figura mostram como funcionam estes operadores.

AND

	0	1
0	0	0
1	0	1

OR

	0	1
0	0	1
1	1	1

XOR

	0	1
0	0	1
1	1	0

Figura 6.11 – Operadores lógicos.

A instrução *`and rd, rs, rt`* executa uma operação AND entre os bits de *rs* e *rt*, e coloca o resultado no registo *rd*; a instrução *`or rd, rs, rt`* executa uma operação OR entre os bits de *rs* e *rt*, e coloca o resultado no registo *rd*; e finalmente, a instrução *`xor rd, rs, rt`* executa uma operação XOR entre os bits de *rs* e *rt*, e coloca o resultado no registo *rd*. As 3 versões das operações lógicas têm a correspondente versão no formato I (*`andi`*, *`ori`*, *`xori`*).

Exemplo1:

Colocar a zero (*clear*) os bits 4 e 8 do número 0x2A55, colocado no registo $t0.

```
lui $t1, 0xFFFF         # $t1 <- 11111111111111110000000000000000
```

```
addi $t1, $t1, 0xFEEF   # $t1 <- 11111111111111111111011101111
addi $t0, $0, 0x2A55    # $t0 <- 00000000000000000010101001010101
and $t2, $t0, $t1       # $t2 <- 00000000000000000010101001000101
```

Exemplo2:

Colocar a 1 (*set*) os bits 4 e 8 do número 0x2A55, colocado no registo $t0.

```
addi $t1, $0, 0x0110    # $t1 <- 00000000000000000000000100010000
addi $t0, $0, 0x2A55    # $t0 <- 00000000000000000010101001010101
or $t2, $t0, $t1        # $t2 <- 00000000000000000010101101010101
```

Exemplo3:

Inverter os bits 4 e 8 do número 0x2A55, colocado no registo $t0.

```
addi $t1, $0, 0x0110    # $t1 <- 00000000000000000000000100010000
addi $t0, $0, 0x2A55    # $t0 <- 00000000000000000010101001010101
xor $t2, $t0, $t1       # $t2 <- 00000000000000000010101101000101
```

Finalmente, o seguinte exemplo ilustra como se pode ler o valor de um determinado bit.

Exemplo4:

Assumindo que as variáveis `a` e `b` correspondem aos registos `$t0` e `$t2`, o seguinte pseudo-código:

```
if [(bit 20 da variável a) == 1]
      b = 0;
```

pode ser efectuado com as instruções:

```
      lui $t1, 0x0010
      addi $t1, $t1, 0x0000   # $t1 <- 00000000000100000000000000000000
                              # $t0 <- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
      and $t3, $t0, $t1       # $t3 <- 00000000000x00000000000000000000
      beq $t3, $0, fimIf
      add $t2, $0, $0
fimIf:
```

6.6 EXERCÍCIOS

1. Faça um programa que teste o valor do bit 3 de um número, armazenado no registo `$t0`. Se o bit for zero então deve ser colocado a 1; se o bit for um então todos os bits devem ser deslocados 2 bits para a direita.

6.7 EXERCÍCIOS PROPOSTOS

1. O sistema de ficheiros do sistema operativo Unix prevê que o acesso a um ficheiro para fins de leitura, escrita e execução seja codificado em três bits consecutivos. Por exemplo:
    - 100 significa permissão de leitura;
    - 010 significa permissão de escrita;
    - 001 significa permissão de execução.

   Existem três classes de utilização: *user* (*u*), *group* (*g*) e *others* (*o*). Assim, um ficheiro com a máscara 100100100 pode ser lido por todos os utilizadores do sistema (ver Figura 6.12).

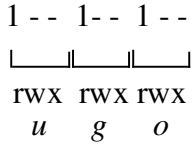


Figura 6.12 – Máscara para permissões de acesso a um ficheiro.

   A máscara de um ficheiro pode ser modificada através do comando *chmod*. Assim, por exemplo:
    - *chmod* u+w teste, adiciona (+) um privilégio para a classe *user* (*u*), a qual passa a poder escrever (*w*) no ficheiro *teste*;
    - *chmod* o-rwx teste, retira (-) privilégios para a classe *others* (*o*), a qual deixa de ter qualquer tipo de acesso (leitura, escrita ou execução) ao ficheiro *teste*.

   Supondo que no comando *chmod* quer a sequência dos caracteres *r*, *w* e *x* quer os caracteres *u*, *g* e *o* poderiam ser substituídos por um número, os dois exemplos acima poderiam ser expressos da seguinte forma:
    - *chmod* 2+2
    - *chmod* 0-7

   considerando que os caracteres *r*, *w* e *x* têm, respectivamente, os pesos 4, 2 e 1 (ver Figura 6.12) e que as classes *u*, *g* e *o* são representadas, respectivamente, por *2*, *1* e *0*.

   Escreva em código *assembly*:
    a) O procedimento *adiciona*;
    b) O procedimento *retira*,

os quais recebem em `$a0` a máscara de um ficheiro, em `$a1` o número correspondente aos acessos a adicionar/retirar e em `$a2` o número da classe a modificar, retornando em `$v0` a nova máscara do ficheiro.

2. Escreva e teste um programa em *assembly* que calcule os primeiros 100 números primos. Deve implementar dois procedimentos:
   - primo(`n`), o qual retorna 1 se `n` é primo e 0 se `n` não é primo;
   - main( ).

6.8 EXERCÍCIOS ADICIONAIS

1. Considere o seguinte código em C para uma primeira aproximação a uma função que calcula iterativamente o factorial de um número.

```
int fact (int n){
int i, res;
      res = 1;
      for (i = 2; i <= n; i++)
            res = res * i;
      return (res);
}
```

   a) Codifique em *assembly* essa função.
   b) Faça um programa que chame a função para calcular o factorial de 5 (cinco). Corra o programa passo a passo enquanto vai preenchendo uma tabela com os valores encontrados nos registos.
2. Escreva um procedimento em código *assembly* correspondente ao seguinte procedimento *C*, o qual conta o número de ocorrências de `1`s numa palavra de 32 bits.

```
int contaUns(int número) {
      int cont;

      cont = 0;
      for (n = 0; n < 32; n++) {
            if [bit n de número == 1]     cont = cont + 1;
      }
```

3. Actualize a tabela do guia 4 com as instruções seguintes: `mult`, `div`, `mfhi`, `mflo`, `sll`, `srl`, `and`, `or` e `xor`.

7. CONCLUSÃO DAS INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS: *STRINGS*. DIRECTIVAS.

7.1 STRINGS

Uma *string* é um vector de caracteres em que o último elemento do vector é o *caracter nulo*, isto é, o caracter '\0'. Uma constante deste tipo é denotada através de uma sequência de caracteres entre aspas (" "). Os caracteres especiais em *strings* seguem as convenções usadas em linguagem C, como por exemplo, " \n ", " \t " ou " \" ".

Cada caracter é representado através do seu código ASCII. O código ASCII de um caracter é um código de 7 bits, os quais são armazenados num byte.

Vimos no guia 2 que para efectuar a transferência de palavras entre a memória e o processador se utiliza as instruções `lw` e `sw`. De forma semelhante, a transferência de bytes é efectuada com as instruções `lb`/`lbu` e `sb`, as quais são, por isso, utilizadas na manipulação de strings (da mesma forma que `lw` e `sw` o eram para a manipulação de vectores de inteiros).

A instrução *load byte* destina-se a ler um byte da memória interpretando a informação nele contida como *tendo sinal*.

Exemplo 1 (ver Figura 7.13):

Ler o inteiro -1 da posição 0x10000100 para o registo `$t0`.

```
lui $t1, 0x1000
addi $t1, $t1, 0x0100
lb $t0, 0($t1)
```

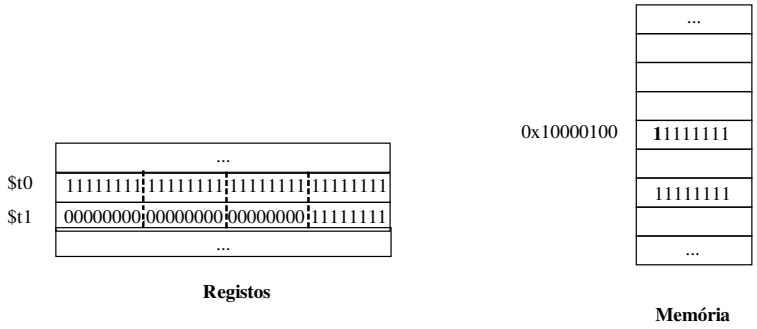


Figura 7.13 – As instruções *lb* e *lbu*

A instrução `lb` interpreta o valor na posição 0x10000100 como uma quantidade com sinal, isto é, o inteiro -1. Por isso, faz a *extensão do sinal* no registo `$t0` para que o respectivo valor seja -1.
A instrução *load byte unsigned* destina-se a ler um byte da memória interpretando a informação nele contida como *não tendo sinal*.

Exemplo 2 (ver Figura 7.13):
Ler o caracter na posição 0x10000101 para o registo `$t1`.

```
lui $t1, 0x1000
addi $t1, $t1, 0x0101
lbu $t1, 0($t1)
```

A instrução `lbu` interpreta o valor na posição 0x10000101 como uma quantidade sem sinal, isto é, o código de um caracter. Por isso, não faz a *extensão do sinal* no registo `$t1`. Neste caso, o valor no registo é 255 (o mesmo que na posição 0x10000101).

7.2 EXERCÍCIOS

1. Inicialize as posições de memória a partir do endereço 0x10000100 com a string "Sol" (Sugestão: reveja a definição de *string* com que se inicia a secção 7.1).
2. Considerando que `$t0` contém o endereço de um caracter, qual a instrução correcta para ler o caracter para o registo `$t1`: `lb $t1, 0($t0)` ou `lbu $t1, 0($t0)`?
3. No exercício 2 na secção 4.9 pedia-se o programa principal e um procedimento para copiar os elementos de um vector para outro. A partir desse exercício, crie agora outro que efectue a cópia de uma *string*. Note que o procedimento apenas tem dois parâmetros: o endereço da string a copiar e o endereço para onde se deve copiar a string. No programa principal, considere os mesmos endereços para `u` e `v`, sendo este último inicializado com a string "Sol".

7.3 DIRECTIVAS

Já vimos em guias anteriores que um assemblador, além de instruções, permite a inclusão num programa em linguagem assembly de comentários e *labels*. Os

comentários em *assembly* são precedidos do símbolo cardinal (#). A partir deste símbolo tudo até ao fim da linha é ignorado pelo assemblador.

As *labels* são identificadores declarados no início de uma linha e seguido por dois pontos (:). Os identificadores que podem ser utilizados são uma sequência de caracteres alfanuméricos incluindo o traço (_) e o ponto (.) e não podem começar por um número. Os códigos de instruções não podem ser usados como identificadores.

Vamos agora ver que num programa em linguagem assembly também podemos incluir directivas. As directivas não são instruções: as instruções são executadas pelo processador; as directivas não fazem parte do programa em linguagem máquina. Assim, as directivas são utilizadas pelo assemblador para várias finalidades como, por exemplo, alocação de espaço no segmento de dados e respectiva inicialização (isto é, *declaração de variáveis*).

O SPIM aceita um subconjunto das directivas suportadas pelo assemblador MIPS. As principais directivas do SPIM estão representadas na tabela seguinte (são precedidas de um ponto (.) e pelo identificador da directiva).

.

Sintaxe	Efeito
.ascii *str*	Armazena *string str* em memória, mas não coloca o caracter nulo no fim da *string*.
.asciiz *str*	Armazena *string str* em memória, colocando o caracter nulo no fim da *string*.
.byte *b1,..., bn*	Armazena as *n* quantidades de 8 bits em bytes sucessivos na memória
.data <*addr*>	Itens subsequentes são armazenados no segmento de dados. Se o argumento opcional *addr* está presente, os itens são armazenados começando no endereço *addr*.
.global *sym*	Declara que o símbolo *sym* é global e pode ser referenciado noutros ficheiros.
.half *hl, ..., hn*	Armazena *n* quantidades de 16 bits em sucessivas posições de memória.
.word *w1, ...wn*	Armazena *n* quantidades de 32 bits em *words* sucessivas de memória.
.space *n*	Reserva n bytes de espaço no segmento de dados.
.text <*addr*>	Itens subsequentes são colocados no segmento de código. Estando presente o argumento opcional *addr*, os itens são armazenados a partir do endereço *addr*.

Tabela 5 – Principais directivas do SPIM

Exemplo 3 (ver Figura 3.8 a)):

Alocar 3 bytes no endereço *vector1* e inicializá-los com os valores *1*, *2* e *3*.

```
vector1:    .byte 1, 2, 3
```

Exemplo 4 (ver Figura 3.8 b)):

Alocar 3 *words* no endereço *vector2* e inicializá-las com os valores *1*, *2* e *3*.

```
vector2:    .word 1, 2, 3
```

Exemplo 5 (ver Figura 3.8 c)):

Alocar 3 *half - words* no endereço *vector3* e inicializá-las com os valores *1*, *2* e *3*.

```
vector3:    .half 1, 2, 3
```

Exemplo 6 (ver Figura 3.8 d)):

Alocar 3 *bytes* nosegmento de dados no endereço *vector4*.

```
vector4:    .space 3
```

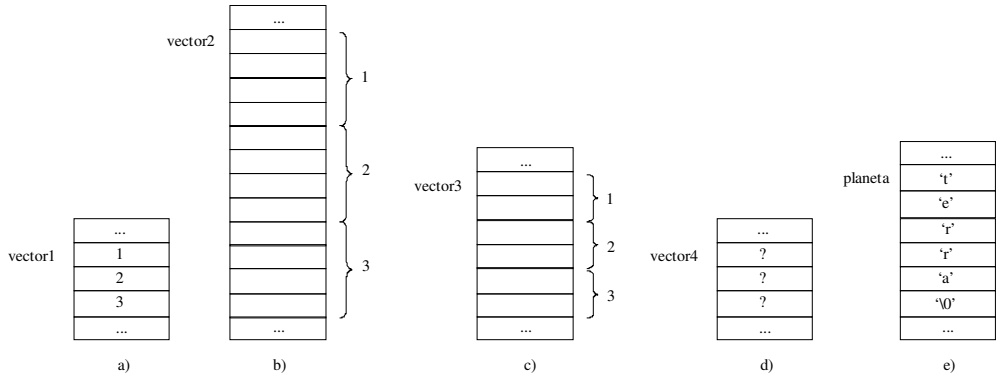


Figura 7.14 – Directivas

Notar que nos exemplos anteriores o programador em linguagem assembly utilizou um identificador para especificar o endereço de uma variável (endereço no segmento de dados). Em guias anteriores, o programador utilizou o próprio endereço. Assim, as directivas possibilitam uma programação com um grau de abstração maior (nível mais elevado).

7.4 EXERCÍCIOS

1. O seguinte programa em assembly:

```
                .data 0x10002000
vector:         .word 1, 2
```

   armazena a variável *vector* no segmento de dados no endereço 0x10002000 e inicializa-a com os valores *1* e *2*. Resolva o exercício 1 a) da secção 2.8 utilizando directivas.

2. Os programas assembly que construiu até ao momento, possuíam apenas instruções (segmento de código), não tendo parte para declaração de variáveis (segmento de dados). Quando um programa só tem segmento de código, não

precisa das directivas `.data` e `.text` para efectuar a separação entre código e dados.

a) O seguinte programa precisa da directiva `.text`?

```
            .text
main:       addi $t0, $0, 2
            jr $ra
```

b) O seguinte programa está correcto? Corrija-o.

```
            .text
            .word 1, 2
main:       addi $t0, $0, 2
            jr $ra
```

0. As directivas `.ascii` e `.asciiz` permitem inicializar variáveis do tipo string. O seguinte programa:

```
            .data
planeta:    .asciiz "terra"
```

inicializa a variável *planeta* com a string "terra" (ver Figura 3.8 e)). Resolva o exercício 1 da secção 7.2 utilizando directivas.

7.5 EXERCÍCIOS PROPOSTOS

1. Considere a instrução
   a) Qual o valor do registo `$t0` após executar a instrução `addi $t0,$0,'i'`?
   b) Qual o significado desse valor?
2. Construa:
   a) Um procedimento que conte as ocorrências de um caracter `c` numa frase. O procedimento está definido em código *C* da seguinte forma:

```
int conta_ocor (char *frase, char c) {
int i, cont;

      i = 0;
      cont = 0;
      while (frase[i] != '\0') {
            if (frase[i] == c)  cont++;
            i++;
      }
      return cont;
}
```

b) O programa principal de forma a testar o procedimento efectuado, inicializando o vector `frase` com o valor "A Terra gira à volta do Sol" no endereço 0x10000100, utilizando directivas.

7.6 EXERCÍCIOS ADICIONAIS

1. Resolva o exercício 1 da secção 7.4 para o seguinte procedimento:

```
int contaPal(char *frase) {
int i, n_pal;

      i = 0;
      n_pal = 0;
      while (frase[i] != "\0") {
            i++;
            if ((frase[i-1] != " ") && (frase[i] == " ")||
                  (frase[i] == "\0"))
                  n_pal ++;
      }
      return n_pal;
}
```

2. Actualize a tabela do guia 6 com as instruções seguintes: `sb`, `lb` e `lbu`.

8. CHAMADAS AO SISTEMA. A PSEUDO-INSTRUÇÃO *LOAD ADDRESS*

8.1 A PSEUDO-INSTRUÇÃO "LOAD ADDRESS"

Com a introdução de directivas, no guia 7, o acesso aos valores armazenados no segmento de dados tornou-se mais simples visto que o endereço de cada um dos valores passou a ser especificado através de um identificador (tal como acontece nas linguagens de alto nível).

Contudo, como sabemos, o acesso aos valores armazenados no segmento de dados requer o conhecimento do seu endereço. Assim, dado o nome de uma variável, precisamos de saber qual é o respectivo endereço. A pseudo-instrução `load address` (`la reg,label`) permite solucionar este problema: obtemos no registo `reg` o endereço que `label` representa.

8.2 EXERCÍCIOS

1. Considere o seguinte programa em que a variável *mens1* foi inicializada com a string "O Sol é uma estrela":

```
                .data
mens1: .asciiz "O Sol é uma estrela"
                .text
main:       la $a0, mens1
            jr $ra
```

   a) Qual o valor do registo `$a0` após a pseudo-instrução `la $a0, mens1`?

   b) Como poderia obter o endereço associado à *label* `main`?

8.3 CHAMADAS AO SISTEMA

O Sistema Operativo (SO) dispõe de vários serviços relacionados com a gestão de periféricos. Estes serviços podem ser acedidos em assembly através da instrução *syscall*. A instrução *syscall* transfere o controlo de execução do programa do utilizador para o programa SO, isto é, coloca no registo `PC`, um endereço de uma rotina do SO.

Alguns dos serviços que podem ser acedidos por *syscall* estão representados na tabela seguinte.

Serviço	**Código**	**Argumentos**	**Resultado**
Imprimir inteiro	$v0 = 1	$a0 = inteiro	
Imprimir *string*	$v0 = 4	$a0 = endereço da *string*	
Ler inteiro	$v0 = 5		$v0 = inteiro
Ler *string*	$v0 = 8	$a0 = *buffer*; a1 = comprimento	
Terminar programa	$v0 = 10		

Tabela 6 – Principais serviços do sistema acedidos por *syscall*

Para requisitar um serviço, o código do serviço deve ser colocado no registo $v0 e os argumentos nos registos $a0–$a3. As chamadas ao sistema que devolvem valores, fazem-no através do registo $v0.

Exemplo1: imprimir um inteiro no ecrã.

```
main:
            addi $v0, $0, 1   # Selecionar o procedimento/serviço
            addi $a0, $0, -5  # Parâmetro de entrada
            syscall           # Invocar o procedimento/serviço

            jr $ra
```

Exemplo2: imprimir uma string no ecrã.

```
                  .data
mens2: .asciiz "A Terra é um planeta"
                  .text
main:
            addi $v0, $0, 4   # Selecionar o procedimento/serviço
            la $a0, mens2     # Parâmetro de entrada
            syscall           # Invocar o procedimento/serviço

            jr $ra
```

8.4 EXERCÍCIOS

1. Escreva um programa que leia um inteiro do teclado.

É importante referir aqui algumas particularidades. O serviço que permite imprimir *strings*, imprime todos caracteres até encontrar um caracter nulo. O serviço que permite ler *strings*, lê até (comprimento–1) caracteres para o *buffer* e termina a *string* com um byte nulo. Se menos do que (comprimento–1) caracteres forem introduzidos, o caracter '\n' (nova linha) é introduzido.

Exemplo3: Ler uma string do teclado.

```
                  .data
planeta:    .space 40
                  .text
main:
            addi $v0, $0, 8   # Selecionar o procedimento/serviço
            la $a0, planeta   # Parâmetros de entrada
            addi $a1, $0, 40
            syscall           # Invocar o procedimento/serviço

            jr $ra
```

Nota: Para ler um caracter do teclado, utiliza-se o serviço *ler string*, colocando no registo `$a1` o valor 2.

8.5 EXERCÍCIOS PROPOSTOS

1. Efectue em assembly um programa para:
   a) Ler através do teclado uma linha de texto;
   b) Ler um caracter;
   c) Imprimir na consola o número de ocorrências do caracter na linha de texto.

   Nota: Na alínea c), considere o procedimento *conta_ocor* do exercício 2. a) na secção 7.5, o qual conta o número de ocorrências de um caracter numa string.

2.
   a) Escreva um procedimento, *atoi*, em linguagem assembly MIPS que converta uma string ASCII num inteiro. A string contém apenas caracteres '0' ... '9' (isto é, o procedimento não considera números negativos). O procedimento deve calcular o valor inteiro equivalente à string. Se estiver presente na string um caracter diferente de '0' ... '9', o procedimento deverá retornar o valor -1.

      Os parâmetros do procedimento são os seguintes: no registo `$a0` está armazenado o endereço da string a converter; no registo `$v0` é armazenado o inteiro calculado. Por exemplo (ver Figura 8.15), se o registo `$a0` tiver o valor 0x10000100, correspondente ao endereço da string "24", o registo `$v0` deverá ficar com o valor 24_{10}.

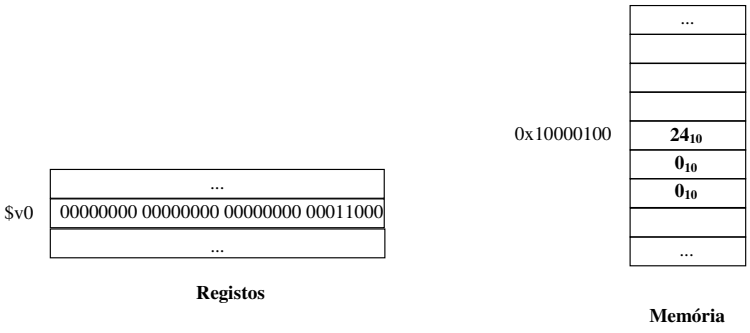


Figura 8.15 – Conversão string – inteiro

Sugestão: Para o exemplo considerado, o procedimento deverá calcular os dígitos $d_1 = 2$ e $d_0 = 4$ e, em seguida, efectuar o seguinte cálculo:

$$\text{inteiro} = d_1 \times 10^1 + d_0 \times 10^0.$$

Os dígitos d_i deverão ser calculados da seguinte forma:

d_i = (código ASCII do caracter correspondente) – (código ASCII do '0')

b) Escreva um programa que leia uma string do teclado e imprima no ecrã o inteiro correspondente.

Nota: O programa deverá utilizar o procedimento *atoi* elaborado na alínea a). Assim, a string a ler só deverá ser constituída pelos dígitos '0' a '9'.

8.6 EXERCÍCIOS ADICIONAIS

1. Escreva:

   a) Um procedimento, *itoa*, em linguagem assembly MIPS que converta um inteiro numa string ASCII. O procedimento deve calcular a string a equivalente ao inteiro e armazená-la no endereço pretendido (a string deverá conter o caracter '-' caso o inteiro seja negativo).

   Os parâmetros do procedimento são os seguintes: no registo `$a0` está armazenado o inteiro, no registo `$a1` está armazenado o endereço da string que se pretende calcular; no registo `$v0` deverá ser retornado o número de caracteres diferentes do caracter nulo que foram armazenados a partir do endereço em `$a1`.

   Por exemplo (ver Figura 8.16), se o registo `$a0` tiver o valor 24_{10} e o registo `$a1` tiver o valor 0x10000100, então a string "24" (caracteres 50 e 52 da tabela ASCII) deverá ser armazenada no endereço 0x10000100 e o registo `$v0` deverá ficar com o valor 2_{10}.

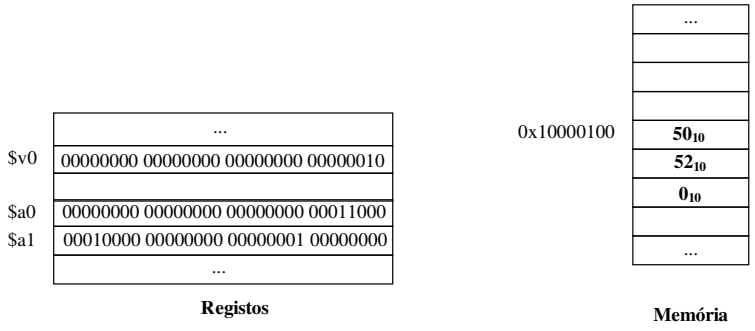


Figura 8.16 – Conversão inteiro - string

b) Um programa que leia um inteiro do teclado e imprima no ecrã a string correspondente.

Nota: O programa deverá utilizar o procedimento *itoa* elaborado na alínea a).

2. Actualize a tabela do guia 7 com a instrução seguinte: `syscall`.

9. PROCEDIMENTOS ENCADEADOS

(Nota prévia: este guia deverá ser resolvido em duas aulas práticas)

No guia número 4, foi estudado como um procedimento em linguagem C é suportado num computador com processador MIPS. Agora, vamos generalizar esse estudo, considerando procedimentos encadeados, isto é, procedimentos que, por sua vez, invocam outros procedimentos. Notar que um procedimento recursivo, isto é, um procedimento que se invoca a si próprio é um caso particular de um procedimento encadeado.

Quando estamos na presença de procedimentos encadeados, a preservação dos registos é ainda mais importante pois os registos `$ra` e os registos `$a0` - `$a3` são solicitados para armazenar mais do que um valor. Consequentemente, antes de atribuir um novo valor a esses registos é necessário armazenar na pilha o valor desses registos pois esses valores serão necessários no futuro.

9.1 EXERCÍCIOS

1. Escreva um procedimento, *localB*, que localize o primeiro caracter 'b' numa string. Prepare o procedimento de forma que:

   `$a0` é o endereço da string;

   `$v0` venha a conter o endereço do primeiro caracter 'b'.

   Quando não existir nenhum caracter 'b' na string, o procedimento deve retornar o endereço do caracter fim da string, isto é, o caracter '\0'.

   Por exemplo, se o valor em `$a0` for 0x10000100 (ver Figura 7.13), então o valor em `$v0` será 0x10000101.

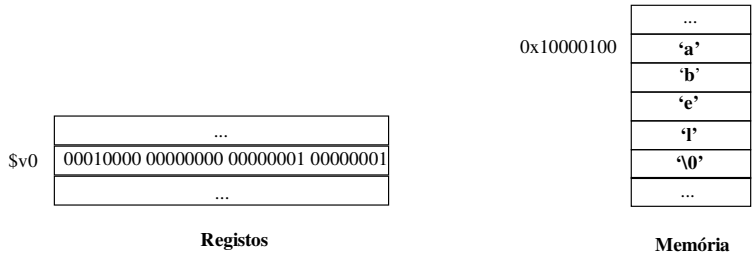


Figura 9.17 – O procedimento *localB*

2. Escreva um procedimento, *localCar*, que localize um determinado caracter numa string, considerando que $a0 contém o endereço da string, `$a1` o caracter a localizar e `$v0` conterá o endereço do caracter.

   Quando o caracter não pertencer à string, o procedimento deve retornar o endereço do caracter fim da string, isto é, o caracter '\0'.

   Nota: Para efectuar o procedimento deve utilizar o código efectuado em 1. e efectuar o menor número de modificações possível.

9.2 EXERCÍCIOS PROPOSTOS

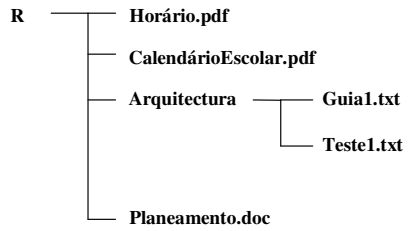
1. Modifique o código MIPS do procedimento *conta_ocor* (exercício 2 a) na secção 7.5) de forma a utilizar o procedimento *localCar* na respectiva implementação.
2. Escreva um programa que:

2.1 Conte o número de caracteres 'b' existentes na string "O bar está aberto no Sábado".

2.2 Conte o número de caracteres 'o' existentes na referida string.

Nota: Os programas deverão utilizar o procedimento *conta_ocor* efectuado em 1.

3. Pretende-se imprimir no ecrã uma determinada "árvore" de directórios (ver Figura 9.2 a)). A informação relativa a cada um dos elementos da árvore de directórios (ficheiro ou directório) é armazenada numa string, a qual tem (ver Figura 9.2 b)): 'F' ou 'D', como primeiro caracter, consoante se trate respectivamente de um ficheiro ou directório; ' ' (espaço) como segundo caracter; nome do ficheiro ou directório como resto da string.



a) Exemplo de uma "árvore" de directórios

```
                .data
          .align 0
R:        .asciiz "F Horário.pdf\n"
          .asciiz "F CalendárioEscolar.pdf\n"
          .asciiz "D Arquitectura   "
          .word     0x10010052
          .asciiz "F Planeamento.doc"
          .asciiz "Z"
Arquitectura:
          .asciiz "F Guia1.txt "
          .asciiz "F Teste1.txt\n"
          .asciiz "Z"
```

b) Uma implementação para a "árvore" de directórios em a)

Figura 9.18 – Árvore de directórios

No caso de um directório, além da respectiva string também existe o *endereço do directório*, isto é, o endereço onde se encontram as strings correspondentes aos elementos que pertencem ao directório. Por exemplo (ver **Error! Reference source not found.** b)), o endereço de *Arquitectura* é `0x10010052`.

Escreva em assembly MIPS:

a) Um procedimento, *imprimeFicheiro* que imprima no ecrã o nome do ficheiro. Caso a string corresponda a um directório, o procedimento deverá invocar o procedimento *imprimeDirectório* para imprimir a respectiva árvore de directórios.

   O procedimento recebe em `$a0` o endereço da string correspondente a um determinado elemento da árvore de directórios. O procedimento retorna em `$v0` o código correspondente ao elemento imprimido (`$v0` = 0 significa "Z", i.e., o indicador de fim de directório; `$v0` = 1 significa "D", i.e., um directório) e em `$v1` é retornado o endereço do fim da string correspondente ao elemento imprimido.

   Nota: A leitura do endereço de um directório não poderá ser efectuada com a instrução `lw`. Em vez disso, utilize o procedimento *carregarPalavra*, cujo código se apresenta a seguir.

```
# procedimento "carregarPalavra"
# descrição: Lê uma palavra utilizando a instrução "load byte"
# parâmetro de entrada: $a0 = endereço da palavra
# parâmetro de saída: $v0 = palavra

carregarPalavra:
            add $v0, $0, $0
            add $t1, $0, $0
            addi $t2, $0, 4
ciclo1:
            lbu $t0, 0($a0)          # $t0 = byte lido
            sllv $t0, $t0, $t1
            or $v0, $v0, $t0
```

```
add $a0, $a0, 1          # $a0 = $a0 + 1
addi $t1, $t1, 8
addi $t2, $t2, -1
bne $t2, $0, ciclo1
jr $ra
```

b) Um procedimento, *imprimeDirectório*, o qual deve imprimir no ecrã cada um dos seus elementos constituintes, através da invocação do procedimento *imprimeFicheiro* para cada um desses elementos. O procedimento recebe em `$a0` o endereço do directório a imprimir.

Nota: O nome dos elementos constituintes do directório deverá sofrer uma identação relativamente ao nome do directório (ver Figura 9.2 a)).

c) Um programa que imprima o directório armazenado no início do segmento de dados, isto é, no endereço `0x10010000`. Assim, o programa deverá invocar o procedimento *imprimeDirectório*.

9.3 EXERCÍCIOS ADICIONAIS

1. Qual é o código assembly MIPS para o procedimento recursivo seguinte, o qual calcula o factorial de um número?

```
int fact (int n) {

        if (n < 1) return(1);
        else
                    return(n * fact(n-1));
}
```

Teste o procedimento *fact*, considerando `n` = 1, 2, 3, 4, 5.

2. Para o cálculo de `fact`(3), mostre o conteúdo da pilha no início de cada uma das chamadas ao procedimento.

10. TEMPO DE EXECUÇÃO

O MipsIt [1] é simulador animado que inclui um ambiente de desenvolvimento e a representação do CPU, de caches de instruções e dados, de parte da RAM, uma janela da consola e alguns dispositivos de entrada saída.

Nesta aula, introduzir-se-á este simulador, e são propostos alguns exercícios visando integração dos conhecimentos adquiridos na disciplina, nomeadamente uma outra visão de como as construções das linguagens de alto-nível são implementadas em linguagem assembly e como essas mesmas construções afectam o desempenho.

O MipsIt contribuí também para a compreensão da organização e estrutura de um computador, para além do que é possível com o SPIM e facilita o estudo autónomo.

O MipsIt está instalado nos computadores da sala de aula. Aos alunos que o queiram usar noutros computadores, sugere-se que acedam à máquina 10.10.22.171 na rede do DEEI, requisitem na biblioteca o CD que acompanha o livro [2] ou acedam a http://www.it.lth.se/dtlab/mipsit/MipsIt.zip.

10.1 O REGISTO $FP

No guia número 4, vimos que o segmento da pilha armazena os registos preservados num procedimento. Assim, no início de um procedimento são alocadas as posições de memória necessárias para armazenar esses registos. Por exemplo, a Figura 4.10 a) mostra os registos `$s0` e `$s1`.

Igualmente, vimos que os parâmetros de entrada de um procedimento são armazenados nos registos `$a0` - `$a3`. O que acontece se um procedimento tiver mais de 4 parâmetros de entrada? Os restantes parâmetros são armazenados na pilha [2].

Finalmente, onde é armazenado um vector declarado dentro do procedimento? As variáveis locais ao procedimento também são armazenadas na pilha. Por exemplo, a Figura 4.10 a) mostra um vector `v` de 10 elementos.

Assim, cada procedimento quando é iniciado aloca um determinado número de posições de memória na pilha para armazenar os registos a preservar, os parâmetros de entrada e as variáveis locais. Por exemplo, o procedimento da Figura 4.10 a) aloca 48 posições de memória.

Durante a execução do procedimento, este pode ter necessidade de aceder a algum item de informação armazenado na pilha. Para referenciar esses itens pode-se utilizar o registo `$sp`. Por exemplo, o elemento `v`[2] do vector `v` (ver Figura 4.10 a)) tem o endereço `$sp + 8`. Contudo, se o valor do registo `$sp` for modificado, para aceder ao mesmo item utilizando `$sp` é necessário modificar o valor do *offset*. Por exemplo, se fôr inserido na pilha um novo valor (ver Figura 4.10 b)), o elemento `v`[2] passa a ter o

endereço `$sp + 12`, isto é, durante a execução do procedimento tem dois endereços diferentes (`$sp + 8` e `$sp + 12`).

Para resolver aquele problema, os elementos na pilha são referenciados relativamente a um registo, `$fp` (*frame pointer*) cujo valor permanece constante durante a execução do procedimento. Assim, o conjunto das posições de memória alocadas é denominado *frame* e ao registo `$fp` é atribuído o valor do registo `$sp` no início do procedimento (ver Figura 4.10 c)).

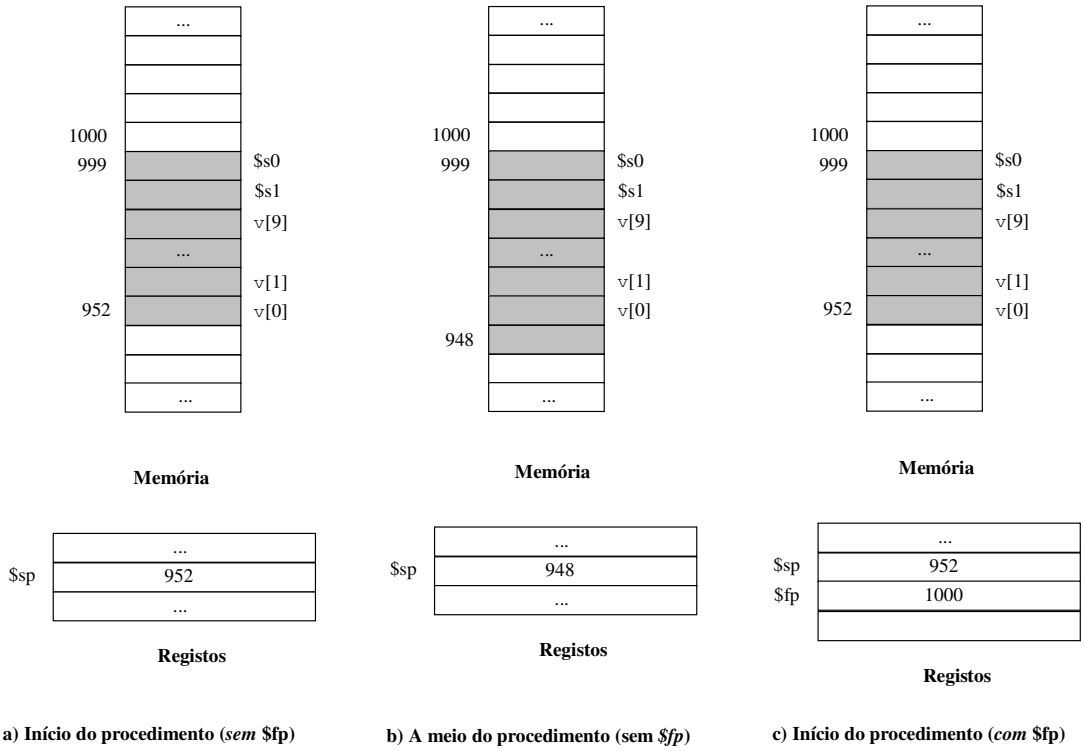


Figura 10.1 – Utilização do registo `$fp`

10.2 MIPSIT

O *MipsIt* é um ambiente de desenvolvimento integrado, o qual possui três janelas (ver Figura 1.3). A janela da esquerda (*project view*) mostra os ficheiros incluídos no projecto. A janela inferior (*output window*) mostra as mensagens resultantes da execução dos diferentes comandos. A terceira janela (*console*) permite comunicar com uma placa, *MIPS board*, e não será utilizada neste guia.

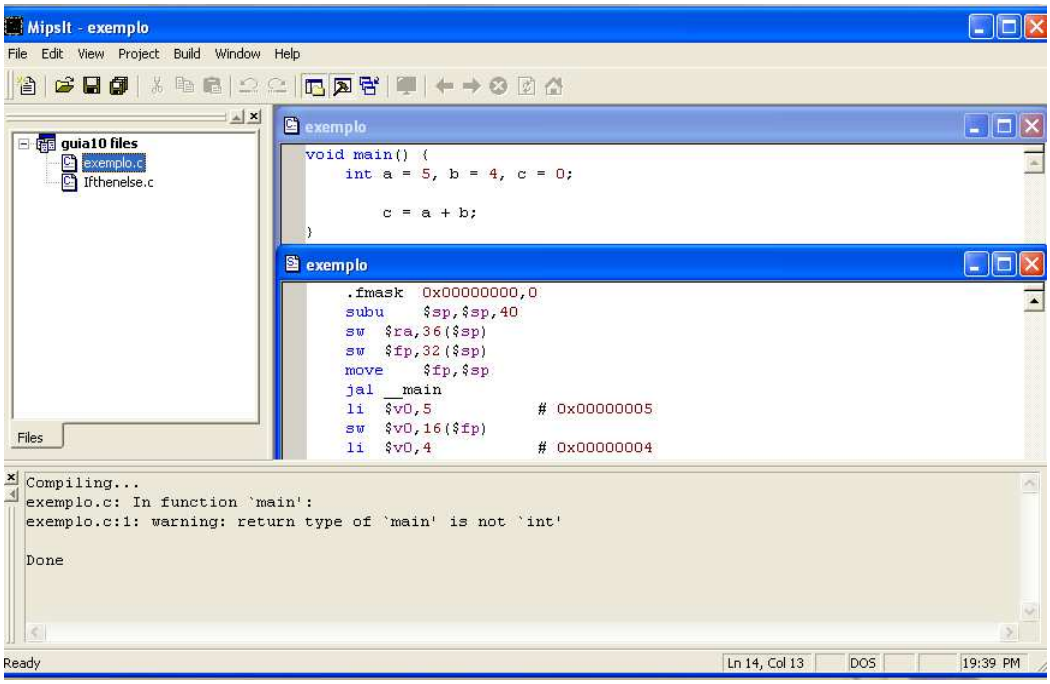


Figura 10.2 – Ambiente do *MipsIt*

10.3 COMPILANDO UM PROGRAMA C UTILIZANDO O MIPSIT

10.3.1 CRIAR UM PROJECTO

Utilizando o comando File > New, aparece uma caixa de diálogo que permite criar um novo projecto (ver Figura 7.13).

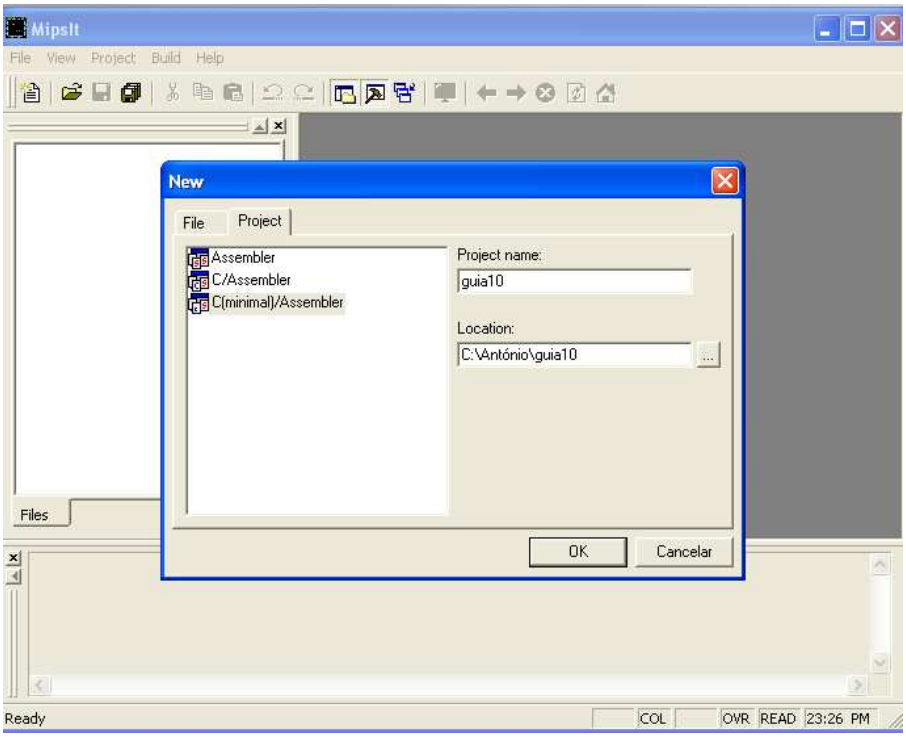


Figura 10.3 – Criar um projecto

O preenchimento da caixa de diálogo deve ser efectuado da seguinte forma. O tipo de projecto a escolher é *C(minimal)/Assembler*. Além disso, é preciso fornecer um nome para o projecto e a respectiva localização.

10.3.2 ADICIONAR UM FICHEIRO AO PROJECTO

Após a criação, é necessário adicionar ficheiros ao projecto, utilizando o comando File > New (ver Figura 10.4).

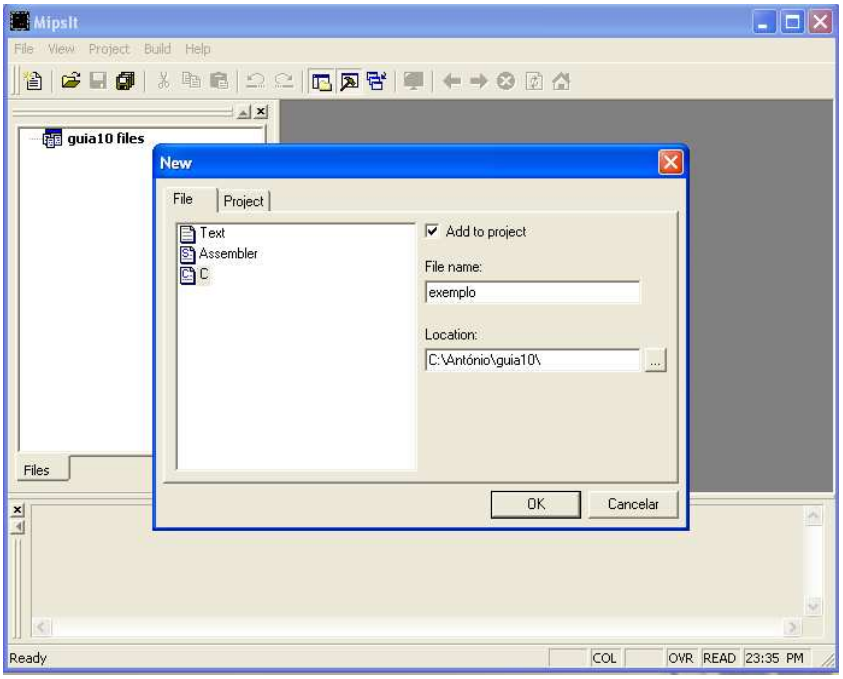


Figura 10.4 – Adicionar um ficheiro ao projecto

O preenchimento da caixa de diálogo deve ser efectuado da seguinte forma. O tipo de ficheiro a escolher é *C*. Além disso, é preciso selecionar a caixa *Add to project* e fornecer o nome e a localização do ficheiro.

Notar que depois de adicionado ao projecto, o ficheiro pode ser editado utilizando a janela correspondente.

Por exemplo, considere-se que foi adicionado ao projecto *guia10* o ficheiro *exemplo.c* seguinte:

```
void main() {
  int a = 5, b = 4, c = 0;

        c = a + b;
}
```

10.3.3 DEFINIR AS OPÇÕES DE COMPILAÇÃO

Antes de efectuar qualquer compilação, é necessário definir as opções de compilação, utilizando o comando Project > Settings (ver Figura 10.5).

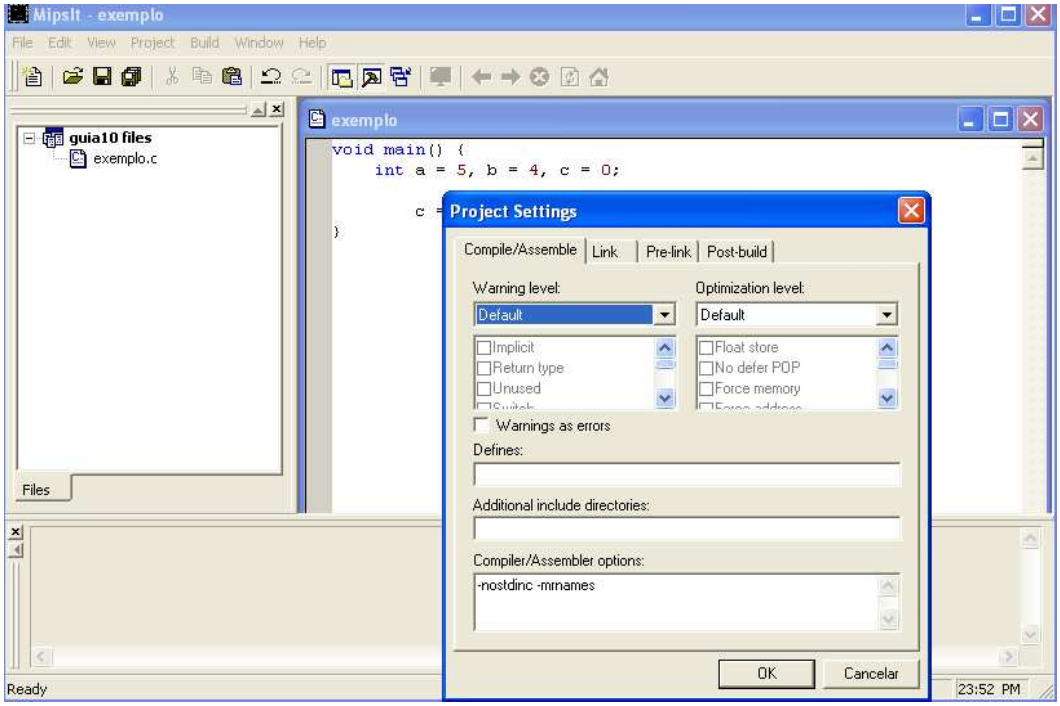


Figura 10.5 – Definir as opções de compilação

O preenchimento da caixa de diálogo deve ser efectuado da seguinte forma. As opções de compilação (*Compiler/Assembler options*) a escolher são *-nostdinc* e *–mrnames*. Além disso, é preciso selecionar os níveis de aviso (*warning level*) e optimização (*optimization level*), os quais devem ser *Default*.

10.3.4 OBTER O CÓDIGO ASSEMBLY

Uma vez criado o ficheiro *C*, podemos obter o correspondente código assembly, utilizando o comando *Build > View Assembler*. Se não houve erros de compilação (ver janela *Output*), o código assembly é mostrado numa janela.

Por exemplo, o código assembly que se obtém para *exemplo.c* é:

```
main:
        li      $v0, 5                  # 0x00000005
        sw      $v0, 16($fp)
        li      $v0, 4                  # 0x00000004
        sw      $v0, 20($fp)
        sw      $0, 24($fp)
        lw      $v0, 16($fp)
        lw      $v1, 20($fp)
        addu    $v0, $v0, $v1
        sw      $v0, 24($fp)
$L1:
        j       $ra
```

Nota: O código assembly obtido inclui também um conjunto de instruções no início

do programa (imediatamente a seguir a `main`):

```
subu  $sp,$sp,40
sw    $ra,36($sp)
sw    $fp,32($sp)
move  $fp,$sp
jal   __main
```

e um conjunto de instruções no fim do programa (imediatamente a seguir a `$L1`):

```
move  $sp,$fp
lw    $ra,36($sp)
lw    $fp,32($sp)
addu  $sp,$sp,40
j     $ra
```

Ambos os conjuntos de instruções, essencialmente, inicializam/restauram os registos `$sp` e `$fp`.

10.4 EXERCÍCIO

Adicione ao projecto os programas em linguagem *C* do exercício 3.4 a), 3.4 b) e 3.4 d). Obtenha os respectivos códigos assembly.

10.5 EXERCÍCIOS PROPOSTOS

1. Compare o código assembly obtido com o código assembly que desenvolveu na aula 3, preenchendo a seguinte tabela.
   Nota: Não considere as instruções que inicializam/restauram os registos `$sp` e `$fp`.

	Código obtido	Código desenvolvido
Categoria das instruções:		
# instruções		
# instruções executadas		
# registos utilizados		

2. Considerando a tabela acima:
   a) Indique quais as instruções (se alguma) que foram apenas utilizadas pelo compilador. Justifique.
   b) Indique quais as instruções (se alguma) que foram apenas consideradas pelo programador. Justifique.

c) Quais as simplificações efectuadas pelo compilador (se alguma) que não considerou.

d) Qual o tempo de execução de cada um dos programas num computador com uma frequência de relógio igual a 2.0 GHz?

Nota: A Tabela 10.1 mostra os valores CPI (*clock cycles per instruction*) para cada instrução.

Instrução	CPI (em média)
arithmetic	1.0 ciclos de relógio
data transfer	1.4 ciclos de relógio
conditional branch	1.7 ciclos de relógio
jump	1.2 ciclos de relógio

Tabela 10.1 – Valores CPI

e) Qual a frequência de relógio de um computador necessária para executar cada um dos programas em 2.0 segundos? (Utilize os valores CPI da Tabela 10.1)

10.6 EXERCÍCIO ADICIONAL

1. Considere o seguinte programa *C*:

```
void main() {
   int a[101], b[101], c;

      for(i = 0; i ≤ 100; i = i + 1) {
            a[i] = b[i] + c;
      }
}
```

a) Utilizando o *MipsIt*, obtenha o respectivo código assembly.

b) Quantas instruções são executadas durante a execução do código assembly?

c) Quantas referências à memória de dados são efectuadas durante a execução?

10.7 BIBLIOGRAFIA

[1] G. Fischer and T. Harms, "Mips lab environment reference," Department of information technology, Lund University, Sweden October 27 2003.
[2] D. Patterson and J. Hennessy, *Computer organization and design: the hardware/software interface*, third ed: Elsevier, 2005.

11. AVALIAÇÃO

Considerando que a última semana poderá ser de avaliação, os exercícios são de natureza geral que demonstrem a compreensão da matéria.

Os alunos que optaram pela avaliação em frequência devem ter os relatórios completos (incluindo os EXERCÍCIOS), sendo avaliada a resolução dos EXERCÍCIOS PROPOSTOS.

11.1 EXERCÍCIOS

1. Considere o código MIPS que a seguir se apresenta, assumindo que $a0 e $a1 são usados para *input* e inicialmente contêm os inteiros a e b, respectivamente, e que $v0 é usado para *output*. Adicione comentários ao código e descreva, numa frase ou numa expressão aritmética, o objectivo do segmento de código apresentado.

```
            add   $t0, $zero, $zero
loop:       beq   $a1, $zero, finish
            add   $t0, $t0, $a0
            sub   $a1, $a1, 1
            j     loop
finish:     addi  $t0, $t0, 100
            add   $v0, $t0, $zero
```

2. O programa que se segue tenta copiar palavras do endereço contido em $a0 para o endereço contido em $a1, contando em $v0 o número de palavras copiadas. O programa pára de copiar quando encontra uma palavra igual a 0 (zero). Não é necessário preservar o conteúdo dos registos $a0, $a1 e $v0. A palavra de terminação deve ser copiada mas não contada.

```
            addi  $v0, $zero, 0     #Inicia o contador
loop:       lw    $v1, 0($a0)       #Lê próxima palavra de origem
            sw    $v1, 0($a1)       #Escreve para destino
            addi  $a0, $a0, 4       #Avança p/a próxima origem
            addi  $a1, $a1, 4       #Avança p/a próximo destino
```

```
            beq   $v1, $zero, loop  #Loop se a palavra copiada
                                    #     for diferente de zero
```

Existem vários erros neste programa. Corrija-os, criando um programa correcto, usando, se necessário, um dos simuladores apresentados nas aulas práticas.

11.2 EXERCÍCIOS PROPOSTOS

1. Faça um programa em linguagem assembly MIPS que leia uma string contendo um número expresso em notação hexadecimal e imprima o número (em decimal) e a string do número correspondente em notação binária.
   Estruture o seu programa de forma a incluir
   a) um procedimento *hexdec* que converta a string no número decimal
   b) um procedimento *hexbin* que converta um caracter hexadecimal nos 4 bits correspondentes;
   c) o programa principal que leia uma string do teclado e imprima no ecrã o número decimal e string do número em binário, utilizando os procedimentos identificados nas alíneas anteriores.

11.3 EXERCÍCIOS ADICIONAIS

1. O fragmento de código que a seguir se apresenta produz um resultado em $v0 a partir de dois vectores. Assuma que cada vector consiste em 2500 palavras indexadas de 0 a 2499, que a base dos vectores está armazenada em $a0 e $a1, respectivamente e que a dimensão (2500) está armazenada em $a2 e $a3, respectivamente. Adicione comentários ao código e descreva, numa frase, o que este segmento faz e, especificamente, que significa o valor devolvido em $v0.

```
            sll   $a2, $a2, 2
            sll   $a3, $a3, 2
            add   $v0, $zero, $zero
            add   $t0, $zero, $zero
outer:      add   $t4, $a0, $t0
            lw    $t4, 0($t4)
            add   $t1, $zero, $zero
inner:      add   $t3, $a1, $t1
            lw    $t3, 0($t3)
            bne   $t3, $t4, skip
            addi  $v0, $v0, 1
skip:       addi  $t1, $t1, 4
```

```
bne   $t1, $a3, inner
addi  $t0, $t0, 4
bne   $t0, $a2, outer
```

2. Considere a correcção que fez no programa apresentado no ponto 2 de Exercícios. Elabore um segmento de código em C que pudesse ter dado origem a esse fragmento de código Assembly. Assuma que a variável source corresponde o registo $a0, a variável destination corresponde a $a1 e a variável count a $v0. Inclua a declaração de variáveis mas assuma que source e destination foram inicializadas com os valores adequados.

11.4 BIBLIOGRAFIA

[1] D. Patterson and J. Hennessy, *Computer organization and design: the hardware/software interface*, third ed: Elsevier, 2005.