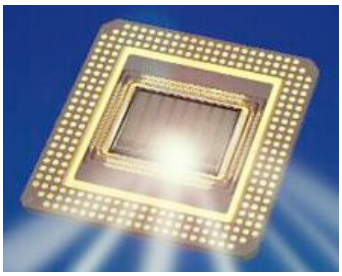


Arquitectura de Computadores
2004 / 2005

Introdução ao Assembly usando o Simulador SPIM

Guia dos Laboratórios

Pedro F. Campos
Departamento de Matemática e Engenharias
Universidade da Madeira

Planeamento dos Laboratórios:

1ª Aula: Introdução à Linguagem Assembly do MIPS R2000

Tópicos: Introdução ao ambiente de laboratório. Definição dos grupos. Treino na utilização do simulador SPIM. Treino nos métodos de teste e depuração dos programas. Estrutura de um programa em Assembly: directivas, etiquetas e pseudo-instruções. Arquitectura do MIPS: Coprocessador 0 e 1. Utilização do coprocessador 1 (Unidade de Vírgula Flutuante).

2ª Aula: Gestão dos Dados em Memória

Tópicos: Declaração de palavras e *bytes* em memória. Declaração de cadeias de caracteres. Reserva de espaço em memória. Alinhamento dos dados na memória. Modos de endereçamento. Carregamento de constantes. Carregamento de palavras/bytes da memória para registos. Armazenamento de palavras/bytes de registos para a memória. **1º Trabalho de avaliação**.

3ª Aula: Operações Aritméticas e Lógicas e Avaliação de condições

Tópicos: Operações Aritméticas com constantes e com dados em memória. Multiplicação, divisão e operações lógicas. Operadores de rotação. Avaliação de condições simples e compostas por operadores lógicos. Definição de uma notação uniforme para a escrita de fluxogramas. Criação de fluxogramas a partir de exemplos de programas.

4ª Aula: Estruturas de controlo condicional e Chamadas ao Sistema

Tópicos: Estruturas de controlo do tipo "se...senão...então". Estruturas de controlo repetitivas do tipo "enquanto", "repetir até" e "para". Treino na concepção e depuração de pequenos troços de programas. Leitura/Escrita de Inteiros a partir da Consola. Leitura/Escrita de Cadeias de caracteres a partir da Consola. **2º Trabalho de Avaliação**.

5ª Aula: Estruturas de controlo condicional e Chamadas ao Sistema

Tópicos: Escrita de pequenos programas utilizando as estruturas ensinadas na aula anterior. Comparação de programas escritos em linguagens de alto nível (e.g. C) com os equivalentes em Assembly. Treino na escrita de programas mais complexos.

6ª Aula: Gestão de subrotinas

Tópicos: Noção de rotinas em Assembly e de métodos de passagem de parâmetros. Gestão da pilha. Visualização da pilha em várias situações. Treino na concepção de pequenos programas com rotinas. **3º Trabalho de Avaliação**.

7ª Aula: Gestão de subrotinas

Tópicos: Treino na concepção de programas mais complexos com rotinas. Programas recursivos em Assembly. Escrita de rotinas destinadas a serem utilizadas no projecto.

8ª Aula: Gestão das Entradas/Saídas por consulta do estado.

Tópicos: Leitura/Escrita no Porto de dados. Consulta ao Porto de Controlo. Apoio à realização do projecto. **4º Trabalho de Avaliação**.

9ª Aula: Gestão das Entradas/Saídas por Interrupções.

Tópicos: Processamento das excepções no simulador SPIM. Descrição do controlador do teclado simulado pelo SPIM. Controlo da entrada de dados mediante interrupções. Apoio à realização do projecto.

10ª Aula: Gestão das Entradas/Saídas por Interrupções.

Tópicos: Continuação da aula anterior e Apoio à realização do projecto. **5º Trabalho de Avaliação**.

11ª Aula: Introdução à Micro-programação.

Tópicos: Codificação e formatos das micro-instruções. Exemplificação para o caso do processador MIPS R2000. Apoio à realização do projecto.

12ª/13ª Aula: Discussões dos Projectos.

Avaliação nos Laboratórios

Cada um dos trabalhos de laboratório possui um objectivo específico. A nota obtida depende directamente do cumprimento desse objectivo, segundo a escala seguinte:

0 - Não compareceu / não atingiu os objectivos mínimos

5 - Atingiu uma pequena parte dos objectivos

10 - Quase atingiu todos os objectivos

15 - Atingiu todos os objectivos

20 - Atingiu plenamente todos os objectivos e superou as expectativas

É de notar que para algumas as aulas de laboratório pode ser necessária uma preparação com antecedência, sob pena de não se conseguirem atingir os objectivos de forma satisfatória.

Motivação + Ambição + Espírito de Equipa = Sucesso

1
Introdução ao Assembly e ao Simulador SPIM

There are three reasons to program in Assembly: speed, speed and speed.

In "The Art of Assembly"

O objectivo desta aula é a familiarização com a ferramenta que será utilizada ao longo de todo o semestre: o simulador SPIM, que simula um processador MIPS R2000. Iremos aprender também a estrutura básica de um programa em linguagem Assembly, e tentaremos compreendê-lo usando as ferramentas de depuração do simulador.

Introdução ao Assembly

A linguagem Assembly não é mais do que uma representação simbólica da codificação binária de um computador: a linguagem máquina. A linguagem máquina é composta por micro-instruções que indicam que operação digital deve o computador fazer. Cada instrução máquina é composta por um conjunto ordenado de zeros e uns, estruturado em campos. Cada campo contém a informação que se complementa para indicar ao processador que acção realizar.

A linguagem Assembly oferece uma representação mais próxima do programador, o que simplifica a leitura e escrita dos programas. Cada instrução em linguagem Assembly corresponde a uma instrução de linguagem máquina, mas, em vez de ser especificada em termos de zeros e uns, é especificada utilizando mnemónicas e nomes simbólicos. Por exemplo, a instrução que soma dois números guardados nos registos `R0` e `R1` e colocar o resultado em `R0` poderá ser codificada como `ADD R0,R1`. Para nós, humanos, é muito mais fácil memorizar esta instrução do que o seu equivalente em linguagem máquina.

Mas... quando usar Assembly? E para quê?

Tipicamente, quando um programador utiliza Assembly, é porque a velocidade ou a dimensão do programa que está a desenvolver são críticas. Isto acontece muitas vezes na vida real, sobretudo quando os computadores são embebidos noutras máquinas (e.g. carros, aviões, unidades de controlo de produção industrial...). Computadores deste tipo[1] devem responder rapidamente a eventos vindos do exterior. As linguagens de alto nível introduzem incerteza quanto ao custo de execução temporal das operações, ao contrário do Assembly, onde existe um controlo apertado sobre que instruções são executadas.

Além deste motivo, existe outro que também está relacionado com a execução temporal dos programas: muitas vezes é possível retirar grandes benefícios da optimização de programas. Por exemplo, alguns jogos que recorrem a elaborados motores 3D são

[1] Este tipo de computadores são designados por computadores embebidos (do inglês *embedded computers*.

parcialmente programados em Assembly (nas zonas de código onde a optimização é mais benéfica que são normalmente as zonas de código mais frequentemente utilizado).

A ideia geral

A linguagem Assembly é uma linguagem de programação. A principal diferença entre esta linguagem e as linguagens de alto nível (como C, C++ ou Java) está no facto de só disponibilizar ao programador poucas e simples instruções e tipos de dados. Os programas em Assembly não especificam o tipo de dados de uma variável (e.g. *float* ou *int*) e tem também de ser o programador a implementar tudo o que tenha a ver com controlo de fluxo, isto é, ciclos, saltos etc...

Estes factores fazem com que a programação em Assembly seja mais propícia a erros e mais difícil de entender do que o habitual, daí a necessidade de um forte rigor e disciplina ao desenvolver programas nesta linguagem.

Estrutura dos programas em Assembly

Descobriremos ao longo do tempo toda a sintaxe da linguagem Assembly, mas torna-se necessário introduzir já a estrutura principal de um programa escrito nesta linguagem. Alguns conceitos básicos são:

Comentários. Estes são especialmente importantes quando se trabalha com linguagens de baixo nível, pois ajudam ao desenvolvimento dos programas e são utilizados exaustivamente. Os comentários começam com o caracter "`#`".

Identificadores. Definem-se como sendo sequências de caracteres alfanuméricos, *underscores* (_) ou pontos (.) que não começam por um número. Os códigos de operações são palavras reservadas da linguagem e não podem ser usadas como identificadores (e.g. `addu`).

Etiquetas. Identificadores que se situam no princípio de uma linha e que são sempre seguidos de dois pontos. Servem para dar um nome ao elemento definido num endereço de memória. Pode-se controlar o fluxo de execução do programa criando saltos para as etiquetas.

Pseudo-instruções. Instruções que o Assembly interpreta e traduz em uma ou mais micro-instruções (em linguagem máquina).

Directivas. Instruções que o Assembly interpreta a fim de informar ao processador a forma de traduzir o programa. Por exemplo, a directiva `.text` informa que se trata de uma zona de código; a directiva `.data` indica que se segue uma zona de dados. São identificadores reservados, e iniciam-se sempre por um ponto.

Q1.1. Dado o seguinte programa:

```
          .data
dados:    .byte 3       # inicializo uma posição de memória a 3
          .text
          .globl main   # deve ser global
main:     lw $t0,dados($0)
```

Indique as etiquetas, directivas e comentários que surgem no mesmo.

O simulador SPIM

O SPIM S20 é um simulador que corre programas para as arquitecturas MIPS R2000 e R3000. O simulador pode carregar e executar programas em linguagem Assembly destas arquitecturas. O processo através do qual um ficheiro fonte em linguagem Assembly é traduzido num ficheiro executável compreende duas etapas:

- *assembling*, implementada pelo assemblador
- *linking*, implementada pelo linker

O assemblador realiza a tradução de um módulo de linguagem Assembly em código máquina. Um programa pode conter diversos módulos, cada um deles parte do programa. Isto acontece usualmente, quando se constrói uma aplicação a partir de vários ficheiros.

A saída do assemblador é um módulo objecto para cada módulo fonte. Os módulos objecto contém código máquina. A tradução de um módulo não fica completa caso o módulo utilize um símbolo (um *label*) que é definido num módulo diferente ou é parte de uma biblioteca.

É aqui que entra o linker. O seu objectivo principal é resolver referências externas. Por outras palavras, o linker irá emparelhar um símbolo utilizado no módulo fonte com a sua definição encontrada num outro módulo ou numa biblioteca. A saída do linker é um ficheiro executável.

O SPIM simula o funcionamento do processador MIPS R2000. A vantagem de utilizar um simulador provém do facto de este fornecer ferramentas de visualização e depuração dos programas que facilitam a tarefa sem prejuízo da aprendizagem desta linguagem. Além disso, o uso de um simulador torna-o independente da máquina (ou seja, não é necessário comprar um processador MIPS e podemos utilizar o simulador numa variedade de plataformas, e.g. Windows, Mac, UNIX...).

A janela do SPIM encontra-se dividida em 4 painéis:

Painel dos Registos. Actualizado sempre que o programa pára de correr. Mostra o conteúdo de todos os registos do MIPS (CPU e FPU).

Painel de Mensagens. Mostra as mensagens de erro, sucesso, etc.

Segmento de Dados. Mostra os endereços e conteúdos das palavras em memória.

Segmento de Texto. Mostra as instruções do nosso programa e também as instruções do núcleo (*kernel*) do MIPS. Cada instrução é apresentada numa linha:

```
[0x00400000] 0x8fa40000 lw $4,0($29)    ; 89: lw $a0, 0($sp)
```

O primeiro número, entre parêntesis rectos, é o endereço de memória (em hexadecimal) onde a instrução reside. O segundo número é a codificação numérica da instrução (iremos estudar este tema no final dos laboratórios). O terceiro item é a descrição mnemónica da instrução, e tudo o que segue o ponto e vírgula constitui a linha do ficheiro Assembly que produziu a instrução. O número 89 é o número da linha nesse ficheiro.

Utilização das Ferramentas de Depuração

O objectivo desta secção é utilizar o SPIM para visualizar o estado da memória e dos registos ao longo da execução de um programa simples.

Q1.2. Usando um editor à escolha, crie o programa listado de seguida.

```
        .data
msg1:   .asciiz "Digite um numero inteiro: "
```

```
        .text
.globl main
# No interior do main existem algumas chamadas (syscalls) que
# irão alterar o valor do registo $ra o qual contém inicialmente o
# endereço de retorno do main. Este necessita de ser guardado.
main:   addu $s0, $ra, $0 # guardar o registo $31 em $16
        li $v0, 4         # chamada sistema print_str
        la $a0, msg1      # endereço da string a imprimir
        syscall

# obter o inteiro do utilizador
        li $v0, 5         # chamada sistema read_int
        syscall           # coloca o inteiro em $v0

# realizar cálculos com o inteiro
        addu $t0, $v0, $0 # mover o número para $t0
        sll $t0, $t0, 2   #
# imprimir o resultado
        li $v0, 1         # chamada sistema print_int
        addu $a0, $t0, $0 # mover o número a imprimir para $a0
        syscall
# repôr o endereço de retorno para o $ra e retornar do main
        addu $ra, $0, $s0 # endereço de retorno de novo em $31
        jr $ra            # retornar do main
```

O conteúdo do registo `$ra` é guardado noutro registo. O registo `$ra` (utilizado para o mecanismo de chamada/retorno) tem de ser salvo à entrada do main apenas no caso de se utilizar rotinas de sistema (usando `syscall`) ou no caso de se chamar as nossas próprias rotinas. A gestão das rotinas será mais tarde explicada em pormenor.

Guardar o conteúdo de `$ra` em `$s0` (ou em qualquer outro registo com esse fim) apenas funciona se só houver um nível de chamada (ou seja, caso não haja chamadas recursivas da rotina) e se as rotinas não alterarem o registo usado para guardar o endereço de retorno.

Q1.3. Tente descobrir o que faz o programa anterior.

Q1.4. Crie um novo programa alterando a primeira linha que começa com a etiqueta `main` para:

```
label1: li $v0, 4 # chamada sistema print_int
```

Para cada símbolo, o simulador mostra o endereço de memória onde a instrução etiquetada está armazenada. Qual a dimensão de uma instrução (em bytes)?

Q1.5. Usando o `step` do simulador, preencha a seguinte tabela:

Etiqueta	Endereço (PC)	Instrução Nativa	Instrução Fonte

Agora iremos aprender a colocar *breakpoints* no programa. Vamos supor que queremos parar a execução do programa na instrução

```
[0x00400040]  sll $t0,$t0,2
```

Verifique o endereço onde esta instrução reside e adicione um *breakpoint* nesse endereço. Corra o programa.

O programa inseriu um *break* antes de executar a referida instrução. Tome nota do valor do registo `$t0` antes da execução da instrução `sll`. Agora faça `step`. Qual o novo valor do registo `$t0`?

Mais acerca do SPIM

Nesta secção iremos utilizar os registos de vírgula flutuante (*floating point registers*) do SPIM. Por razıes práticas, a definição original da arquitectura do R2000 definiu um processador MIPS como sendo composto por:

- Unidade de Inteiros (o CPU propriamente dito)
- Co-processadores

Esta ideia tinha a ver com o facto de a tecnologia simplesmente não permitir integrar tudo numa única bolacha de silício. Assim, os co-processadores podiam ser circuitos integrados separados ou podiam ser emuladores de software (isto é, as operações de vírgula flutuante eram emuladas por software) no caso de se pretender obter um computador mais barato.

O SPIM simula dois co-processadores:

- Co-processador 0: lida com as interrupções, excepções e o sistema de memória virtual
- Co-processador 1: Unidade de Vírgula Flutuante (FPU)

A FPU (*Floating Point Unit*) realiza operações em:

- números de vírgula flutuante de precisão simples (representação em 32 bits); uma declaração como `float x = 2.7;` reservaria espaço para uma variável chamada `x`, que é um número em vírgula flutuante com precisão simples inicializada a 2.7;
- números de vírgula flutuante de precisão dupla (representação em 64 bits); uma declaração como `double x = 2.7;` reservaria espaço para uma variável chamada `x`, que é um número em vírgula flutuante com precisão dupla inicializada a 2.7;

O co-processador tem 32 registos, numerados de 0 a 31 (e nomeados de `$f0` a `$f31`). Cada registo tem 32 bits de dimensão. Para acomodar números com precisão dupla, os registos são agrupados (o registo `$f0` com o `$f1`, 2 com 3, ... , 30 com 31).

Q1.6. Crie um programa, usando o anterior como base, que lê um *float* a partir do teclado e que o escreve em seguida na consola. Será necessário consultar o conjunto de instruções do SPIM para descobrir que instrução utilizar para mover o conteúdo de um registo de vírgula flutuante. Preencha a coluna "Precisão Simples" na tabela . Como entrada, forneça os quatro últimos dígitos do seu BI, seguidos por um ponto (.) e os quatro dígitos do ano actual.

Q1.7. Corra o programa anterior e preencha a coluna "Precisão Dupla" na mesma tabela.

Registo	Precisão Simples	Precisão Dupla
`$f0`		
`$f2`		
`$f4`		
`$f6`		
`$f8`		
`$f10`		
`$f12`		
`$f14`		
`$f16`		
`$f18`		
`$f20`		
`$f22`		
`$f24`		
`$f26`		
`$f28`		
`$f30`		

2
Gestão dos Dados em Memória

"The struggle of man against power is the struggle of memory against forgetting."

Milan Kundera

O primeiro passo para o desenvolvimento de programas em Assembly consiste em saber gerir os dados em memória. Em Assembly, é da responsabilidade do programador toda a gestão de memória, o que pode ser feito através das directivas que irão ser estudadas.

Antes de prosseguir, convém relembrar que a unidade base de endereçamento é o byte. Uma palavra ou word tem 4 bytes (32 bits), a mesma dimensão que o barramento de dados do MIPS. Desta forma, qualquer acesso à memória supõe a leitura de 4 bytes (1 palavra): o byte com o endereço especificado e os 3 bytes que se seguem. Os endereços devem estar alinhados em posições múltiplas de quatro. Também é possível transferir meia-palavra (*half-word*).

Declaração de palavras em memória

Começaremos por utilizar as directivas `.data` e `.word`. Carregue o ficheiro `aula2-1.asm`:

```
          .data        # segmento de dados
palavra1: .word 13     # decimal
palavra2: .word 0x15   # hexadecimal
```

A directiva `.data [end]` indica o início da zona designada por segmento de dados. Se não especificarmos o endereço `[end]` então o SPIM utilizará, por omissão, o endereço `0x10010000`.

Q2.1. Procure os dados armazenados na memória pelo programa anterior. Localize-os na janela de dados e indique os seus valores em hexadecimal. Em que endereços se encontram esses dados? Porquê?

Q2.2. Quais os valores das etiquetas `palavra1` e `palavra2`?

Q2.3. Crie um segundo ficheiro com a seguinte alteração:

```
               .data          # segmento de dados
palavra1:      .word 13,0x13  # decimal,hexadecimal
```

Que diferenças existem em relação ao programa anterior?

Q2.4. Crie um programa que defina um vector de 5 palavras associado à etiqueta `vector` que comece no endereço `0x10000000` e que contenha os valores 10,11,12,13,14. Certifique-se que o vector é armazenado correctamente na memória.

Q2.5. O que acontece se quisermos que o vector comece no endereço `0x10000002`? Em que endereço começa realmente? Porquê?

Declaração de bytes em memória

A directiva `.byte valor` inicializa uma posição de memória da dimensão de um byte, contendo o `valor`. Crie um ficheiro com o seguinte código:

```
        .data
octeto: .byte 0x10
```

Apague todos os registos e a memória e carregue o novo programa.

Q2.6. Que endereço de memória foi inicializado com o conteúdo especificado?

Q2.7. Que valor foi armazenado na palavra que contém o byte?

Crie outro ficheiro com o seguinte código:

```
        .data
pal1:   .byte 0x10,0x20,0x30,0x40
pal2:   .word 0x10203040
```

Q2.8. Quais os valores armazenados em memória? Que tipo de alinhamento ou organização dos dados usa o simulador (Big-endian ou Little-endian)? Porquê?

Declaração de cadeias de caracteres

A directiva `.ascii "cadeia"` permite carregar, em posições de memória consecutivas (cada uma com dimensão de 1 byte), o código ASCII de cada um dos caracteres da cadeia. Crie o seguinte código:

```
        .data
cadeia: .ascii  "abcde"
octeto: .byte   0xff
```

Q2.9. Localize a cadeia na memória.

Q2.10. Altere a directiva `.ascii` para `.asciiz` e diga o que acontece. O que faz esta directiva?

Q2.11. Escreva um programa que carregue a mesma cadeia de caracteres mas utilizando a directiva `.byte`.

Reserva de espaço em memória

A directiva `.space n` serve para reservar espaço para uma variável em memória, inicializando-a com o valor 0. Crie o seguinte código:

```
                .data
palavra1:       .word 0x20
espaco:         .space 8
palavra2:       .word 0x30
```

Q2.12. Que intervalo de posições foram reservadas para a variável `espaco`? Quantos bytes foram reservados no total? E quantas palavras?

Alinhamento dos dados na memória

A directiva `.align n` alinha os dados seguintes a um endereço múltiplo de 2^n. Crie o seguinte código:

```
        .data
byte1:  .byte 0x10
espaco: .space 4
```

```
byte2:  .byte 0x20
pal:    .word 10
```

Q2.13. Que intervalo de posições de memória foram reservadas para a variável `espaco`? A partir de que endereços se inicializaram `byte1` e `byte2`? E a partir de que endereço se inicializou `pal`? Porquê?

Faça agora a seguinte alteração:

```
        .data
byte1:  .byte 0x10
        .align 2
espaco: .space 4
byte2:  .byte 0x20
pal:    .word 10
```

Q2.14. Responda novamente à Questão 2.13. O que fez a directiva `.align`?

Carregamento e Armazenamento dos Dados

Agora iremos estudar o carregamento e o armazenamento dos dados. O carregamento consiste no transporte dos dados da memória para os registos enquanto que o armazenamento consiste no transporte dos dados dos registos para a memória.

No MIPS R2000, as instruções de carregamento começam com a letra "l" (do inglês *load*) e as de armazenamento com a letra "s" (do inglês *store*), seguidas pela letra inicial correspondente ao tamanho do dado que se vai mover: "b" para byte, "h" para meia-palavra (*half-word*) e "w" para uma palavra (*word*).

Carregamento de dados Imediatos (Constantes)

Crie o seguinte programa:

```
        .text
main:   lui $s0,0x4532
```

Q2.15. Consulte o manual para descobrir o que faz a instrução `lui`. Carregue o programa e anote o conteúdo do registo `$s0`. Qual a dimensão total (em bits) dos registos?

Agora faça a seguinte alteração:

```
        .text
main:   lui $s0,0x98765432
```

Q2.16. Apague os valores da memória e carregue este programa. Verifique novamente o estado do registo `$s0`.

Carregamento de Palavras (Memória → Registo)

Nesta secção iremos aprender a transferir uma palavra (32 bits) da memória para um registo (`$s0`). A instrução `lw` carrega para um registo (primeiro argumento desta instrução) a palavra contida numa posição de memória cujo endereço é especificado no segundo argumento desta instrução. Este endereço é obtido somando o conteúdo do registo com o de um identificador, como se vê neste código que deverá carregar para o simulador:

```
        .data
pal:    .word 0x10203040
        .text
main:   lw $s0,pal($0)
```

Q2.17. Reinicialize a memória e os registos do simulador e carregue este programa. Verifique o conteúdo do registo `$s0`. Localize a instrução `lw $s0,pal($0)` na memória e repare como o simulador traduziu esta instrução.

Q2.18. Faça a seguinte modificação no código: em vez de transferir a palavra contida no endereço de memória referenciado pela etiqueta `pal`, transfira a palavra contida no endereço de memória referenciado por `pal + 1`. Explique o que acontece e porquê.

Q2.19. Modifique o programa anterior para passar a guardar no registo `$s0` os dois bytes de maior peso de `pal`. Sugestão: utilize a instrução `lh` que lhe permite carregar meias-palavras (16 bits) da memória para um registo (nos 16 bits de menor peso do mesmo).

Carregamento de Bytes (Memória → Registo)

Crie o seguinte programa:

```
        .data
oct:    .byte 0xf3
seg:    .byte 0x20
        .text
main:   lb $s0,oct($0)
```

Q2.20. A instrução `lb` carrega um byte a partir de um endereço de memória para um registo. O endereço do byte obtém-se somando o conteúdo do registo `$0` (sempre igual a zero) e o identificador `oct`. Localize a instrução na zona de memória das instruções e indique como é que o simulador transforma esta instrução.

Q2.21. Substitua a instrução `lb` por `lbu`. Que acontece ao executarmos o programa?

Armazenamento de Palavras (Registo → Memória)

Crie o seguinte programa:

```
        .data
pal1:   .word 0x10203040
pal2:   .space 4
pal3:   .word 0xffffffff
        .text
main:   lw $s0,pal1($0)
        sw $s0,pal2($0)
        sw $s0,pal3($0)
```

A instrução `sw` armazena a palavra contida num registo para uma posição de memória. O endereço dessa posição é, tal como anteriormente, obtido somando o conteúdo de um registo com o deslocamento especificado na instrução (identificador).

Q2.22. Verifique o efeito deste programa. De seguida, altere o programa para:

```
        .data
pal1:   .word 0x10203040
pal2:   .space 4
pal3:   .word 0xffffffff
        .text
main:   lw $s0,pal1
        sw $s0,pal2
        sw $s0,pal3+0
```

Que diferenças existem entre as duas versões?

Armazenamento de Bytes (Registo → Memória)

Crie o seguinte programa

```
        .data
pal:    .word 0x10203040
oct:    .space 2
        .text
main:   lw $s0,pal($0)
        sb $s0,oct($0)
```

A instrução `sb` armazena o byte de menor peso de um registo numa posição de memória. Apague os registos e a memória e carregue o código. Comprove também o efeito deste programa.

Q2.23. (1º Trab. Lab. 2003-2004) Sabendo que um inteiro se armazena numa palavra, escreva um programa em Assembly que defina na zona de dados uma matriz *A* de inteiros definida como:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

a partir do endereço `0x10000004` supondo que a matriz *A* se armazena por linhas. O seu programa deverá então imprimir no écran a primeira linha da matriz, sem recorrer a instruções de salto e utilizando a chamada sistema `syscall` (Tabela A.17 do Manual).

1º Trabalho de Laboratório

3
Operações Aritméticas e Lógicas e Avaliação de Condições

"Arithmetic is where the answer is right and everything is nice and you can look out of the window and see the blue sky - or the answer is wrong and you have to start over and try again and see how it comes out this time."

Carl Sandburg

Nesta secção iremos estudar as operações aritméticas, lógicas e de rotação de dados. As operações aritméticas são constituídas por operações de soma e subtracção (`add`, `addu`, `addi`, `addiu`, `sub` e `subu`) e por operações de multiplicação (`mult`, `multu`, `div`, `divu`). As instruções que terminam por "u" causam uma excepção de *overflow* no caso de o resultado da operação não for de dimensão acomodável em 32 bits (dimensão dos registos).

As instruções `or` (ou `ori`), `and` (ou `andi`) e `xor` (`xori`) permitem realizar as conhecidas operações lógicas com o mesmo nome. As instruções de rotação (aritméticas/lógicas), também conhecidas por shift são: `sra`, `sll` e `srl`.

Operações Aritméticas

Crie um ficheiro com o seguinte programa:

```
        .data
numero: .word 2147483647
        .text
main:   lw $t0,numero($0)
        addiu $t1,$t0,1
```

Q3.1. Carregue o programa no SPIM e comprove o seu efeito.

Q3.2. Altere a instrução `addiu` para `addi`. O que acontece?

Operações Aritméticas com Dados na Memória

Crie um ficheiro com o seguinte programa:

```
        .data
num1:   .word 0x80000000
num2:   .word 1
num3:   .word 1
        .text
main:
        lw $t0,num1($0)
```

```
        lw $t1,num2($0)
        subu $t0,$t0,$t1
        lw $t1,num3($0)
        subu $t0,$t0,$t1
        sw $t0,num1($0)
```

Q3.3. O que faz este programa? O resultado que fica armazenado no endereço `num3` é o resultado correcto? Porquê?

Q3.4. O que acontece se alterarmos as instruções `subu` por `sub`?

Q3.5. Escreva um programa em Assembly do MIPS que armazene um vector de inteiros V=[1 2 3] e armazene o resultado da soma dos seus três elementos na posição de memória cujo endereço é o primeiro endereço após o fim do vector.

Q3.6. Escreva um programa em Assembly que realize a seguinte operação:

```
f=(g-h)+(i-j)
```

Q3.7. Agore crie o seguinte programa:

```
        .data
num1:   .word 0x7FFFFFFF
num2:   .word 16
        .space 8
        .text
main:   lw $t0,num1($0)
        lw $t1,num2($0)
        mult $t0,$t1
        mfhi $t0
        mflo $t1
        sw $t0,num2+4($0)
        sw $t1,num2+8($0)
```

Este programa declara duas variáveis `num1` e `num2`, carrega-as para os registos `$t0` e `$t1` respectivamente e armazena o resultado da multiplicação nas palavras que vêm depois dessas variáveis (a directiva `.space` reserva espaço para essas duas palavras).

Que resultado se obtém após executar este programa? Porque fica o resultado armazenado em duas palavras de memória?

Q3.8. Modifique o código anterior para que `num1` e `num2` sejam respectivamente 10 e 3. Escreva código para dividir `num1` por `num2` e coloque o quociente e resto nas posições que vêm a seguir a `num1` e `num2`.

Operações Lógicas

As operações lógicas de deslocamento("shifts") permitem deslocar bits dentro duma palavra, "shift left" para a esquerda e "shift right"para a direita.

Outra classe de operações lógicas são o AND, OR e XOR, que operam entre duas palavras, numa correspondência unívoca, e operam bit a bit.

Por exemplo, para colocar o bit 2 de uma palavra de 32 bits a 1:

```
addi $t0,$0,0x000055aa        # t0 = 0000 0000 0000 0000 0101 0101 1010 1010
```

faz-se um OR com uma máscara 0x00000004:

```
move $t1, 0x00000004          # t1 = 0000 0000 0000 0000 0000 0000 0000 0100
or $t2,$t0,$t1                # t2 = 0000 0000 0000 0000 0101 0101 1010 1110
```

Por outro lado, para colocar o bit 5 de uma palavra de 32 bits a 0:

```
addi $t0,$0,0x000055aa        # t0 = 0000 0000 0000 0000 0101 0101 1010 1010
```

faz-se um AND com uma máscara 0xffffffdf:

```
move $t1, 0xffffffdf          # t1 = 1111 1111 1111 1111 1111 1111 1101 1111
and $t2,$t0,$t1
```

Q3.9. Crie o seguinte programa:

```
        .data
num:    .word 0x3ff41
        .space 4
        .text
main:   lw $t0,num($0)
        andi $t1,$t0,0xfffe
        # 0xfffe em binário é 0...0 1111 1111 1111 1110
        sw $t1,num+4($0)
```

Carregue-o e diga o que faz este programa.

Q.3.10. Modifique o código a fim de obter, na palavra seguinte, o mesmo número num com todos os 16 bits mais significativos tal e qual como estavam, e os 16 bits menos significativos a zero, excepto o bit 0, que deve manter o seu valor original.

Operações de Rotação

Q3.11. Escreva um programa em Assembly que receba da consola um número inteiro, divida esse número por 4 (utilizando um operador de rotação) e escreva o número outra vez na consola.

Q3.12. Escreva um programa que inverta os bits de um número.

Avaliação de Condições Simples

Como já vimos, a linguagem Assembly não possui qualquer estrutura de controlo do fluxo do programa que permita decidir sobre dois ou mais caminhos de execução distintos. Para implementar uma estrutura deste tipo é necessário avaliar previamente uma condição, simples ou composta. O caminho de execução que o programa segue dependerá então desta avaliação.

Nesta secção iremos aprender a implementar a avaliação de condições para o atingirmos, no capítulo seguinte, o objectivo pretendido.

O seguinte programa compara as variáveis `var1` e `var2` e deixa o resultado na variável (booleana) `res`:

```
        .data
var1:   .word 30
var2:   .word 40
res:    .space 1
        .text
main:   lw $t0,var1($0)       # carrega var1 em t0
        lw $t1,var2($0)       # carrega var2 em t1
        slt $t2,$t0,$t1       # coloca t2 a 1 se t0<t1
        sb $t2,res($0)        # armazena t2 em res
```

Q3.13. Verifique que valor se armazena na posição de memória `res`.

Q3.14. Modifique o código anterior para avaliar a condição: `res` fica a 1 no caso de `var1=var2`.

Avaliação de Condições Compostas por Operadores Lógicos

Crie o seguinte programa:

```
        .data
var1:   .word 40
var2:   .word -50
res:    .space 1
        .text
main:   lw $t8,var1($0)
        lw $t9,var2($0)
        and $t0,$t0,$0
        and $t1,$t1,$0
        beq $t8,$0,igual
        ori $t0,$0,1
igual:  beq $t9,$0,fim
        ori $t1,$0,1
fim:    and $t0,$t0,$t1
        sb $t0,res($0)
```

Apague a memória e os registos e carregue e execute este programa.

Q3.15. Qual o valor que fica armazenado na posição de memória `res`? Desenhe um fluxograma descrevendo este programa.

Q3.16. Responda à questão anterior inicializando `var1` e `var2` com os valores 0 e 20, respectivamente.

Q3.17. Responda à questão 3.15 inicializando `var1` e `var2` com os valores 20 e 0, respectivamente.

Q3.18. Responda à questão 3.15 inicializando `var1` e `var2` com os valores 0 e 0, respectivamente.

Q3.19. Que comparação composta se realizou entre `var1` e `var2`?

Vamos a mais um exemplo:

```
        .data
var1:   .word 30
var2:   .word -50
res:    .space 1
        .text
main:   lw $t8,var1($0)
        lw $t9,var2($0)
        and $t1,$t1,$0
        and $t0,$t0,$0
        beq $t8,$0,igual
        ori $t0,$0,1
igual:  slt $t1,$t9,$t8
fim:    and $t0,$t0,$t1
        sb $t0,res($0)
```

Q3.20. Qual o valor armazenado na posição de memória `res`? Desenhe o fluxograma para ajudar a perceber o programa.

Q3.21. Qual a comparação composta que se realizou?

4
Estruturas de controlo condicional

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to." said the Cat.
"I don't much care where" — said Alice.
"Then it doesn't matter which way you go." said the Cat.
"—so long as I get somewhere." Alice added as an explanation.

Lewis Carroll, *Alice's Adventures in Wonderland*

No capítulo anterior vimos como realizar comparações entre variáveis. Estas comparações servem para introduzirmos a implementação de estruturas de controlo do fluxo do programa.

O seguinte programa implementa uma estrutura de controlo condicional do tipo "se... então":

```
        .data
var1:   .word 40
var2:   .word 30
res:    .space 4
        .text
main:   lw $t0,var1($0)       # carrega var1 em t0
        lw $t1,var2($0)       # carrega var2 em t1
        and $t2,$t2,$0        # t2 = 0
se:     beq $t1,$0,fimse      # se t1 = 0, salta para fim do se
entao:  div $t0,$t1           # t0/t1
        mflo $t2              # armazenar LO em $t2
fimsem: add $t3,$t0,$t1       # t3 = t0 + t1
        add $t2,$t3,$t2       # t2 = t3 + t2
        sw $t2,res($0)        # guardar t2 em res
```

O mesmo programa escrito em pseudo-código:

```
variáveis:
        int var1=40; var2=30; res;
início: {
        t0=var1; t1=var2;
        t2=0;
        se t1 != 0 então {
                t2 = t0/t1;
        }
        t3 = t0 + t1;
        t2 = t2 + t3;
        res = t2;
}
```

Esta descrição quase directa do algoritmo em Assembly resulta de estarmos a lidar com um processador cuja arquitectura é baseada em load/store que necessita de usar os registos para armazenar temporariamente os valores da memória. Podemos abstrair mais e trabalhar directamente sobre as variáveis armazenadas na memória:

```
variáveis:
        int var1=40; var2=30; res;
início: {
        se var1 != 0 então {
                res = var1/var2;
        }
        res = var1 + var2;
}
```

Q4.1. Crie um programa que implemente o seguinte troço de código em C:

```
if (var1 == var2) {
        var1 = var3; /* trocar os valores de var1 e var2 para o valor de var3 */
        var2 = var3;
} else {
        tmp = var1;  /* executar quando var1 != var2 */
        var1 = var2; /* trocar os valores de var1 e var2 */
        var2 = tmp;
}
```

O programa deverá reservar espaço em memória para três variáveis designadas por `var1`, `var2` e `var3` (do tamanho de uma palavra) e os valores iniciais dessas variáveis serão, respectivamente, o primeiro e último dígitos do seu B.I. `Var3` será inicializada com –2003.

Note que `tmp` é uma variável temporária, pelo que deverá utilizar qualquer um dos registos `$t0` a `$t9`.

Estrutura de Controlo Se…Então com uma Condição Composta

Crie o seguinte programa, que implementa uma estrutura de controlo do tipo "se…então" mas desta vez avaliando uma condição composta:

```
        .data
var1:   .word 40
var2:   .word 30
res:    .space 4
        .text
main:   lw $t0,var1    # t0 = var1
        lw $t1,var2    # t1 = var2
        and $t2,$t2,$0 # t2 = 0
se:     beq $t1,$0,fimse      # se t1=0 saltar para fimse
        beq $t0,$0,fimse
entao:  div $t0,$t1    # t0/t1
        mflo $t2       # armazenar o registo LO em t2
fimse:  add $t3,$t0,$t1
        add $t2,$t3,$t2
        sw $t2,res
```

A descrição alto nível correspondente a este programa Assembly é a seguinte:

```
variáveis:
```

```
        int var1=40; var2=30; res;
início: {
        t0 = var1;
        t1 = var2;
        se ((t0 != 0) e (t1 != 0)) então {
                t2 = t0/t1;
        }
        t2 = t2 + t0 + t1;
        res = t2;
}
```

Q4.2. Desenhe o fluxograma correspondente a este programa Assembly.

Q4.3. Ao executar o programa, que valor se armazena na variável `res`?

Q4.4. Implemente o seguinte programa (descrito em pseudo-código) em Assembly:

```
variáveis:
        int var1=40; var2=30; res;
início: {
        se (var1 > 0) e (var2 >= 0) então {
                res = var1/var2;
        }
        res = res + var1 + var2;
}
```

Q4.5. (2º Trab. Lab. 2003-2004) Escreva um programa em Assembly do MIPS R2000 que comece por escrever a seguinte *prompt* ao utilizador:

```
Introduza um inteiro: >
```

E em seguida leia o inteiro para uma posição de memória etiquetada por `inteiro`. O programa deverá então verificar se o valor introduzido é menor do que 1000. Se for, o programa imprime o resultado de multiplicar o inteiro lido por 200, seguido da mensagem "fim". Se não for, o programa exibe a seguinte mensagem de erro e termina:

```
Introduziu um inteiro maior que 1000.
```

Aqui fica um exemplo de execução do programa:

```
Introduza um inteiro: > 12
2400
Fim.
```

No caso de o utilizador introduzir um número maior que 1000:

```
Introduza um inteiro: > 1200
Introduziu um inteiro maior que 1000.
Fim.
```

Teste o seu programa com uma gama de valores de teste para verificar a sua correção.

2º Trabalho de Laboratório

Estruturas de Controlo do Tipo "Enquanto…"

Até agora temos estudado estruturas de controlo do fluxo do tipo "se… senão… então". Nesta secção iremos implementar estruturas de controlo repetitivas (ciclos) do tipo "enquanto <condição> …" ou do tipo "para i de 0 até n…".

Vamos começar por estudar este código Assembly que implementa um ciclo do tipo "enquanto…" (equivalente ao `While` do C):

```
        .data
cadeia: .asciiz "AC-Uma"
        .align 2
n:      .space 4
        .text
main:   la $t0,cadeia  # carrega endereço da cadeia em t0
        andi $t2,$t2,0 # t2 <- 0
enquanto:
        lb $t1,0($t0)  # carrega o byte apontado por t0 em t1
        beq $t1,$0,fim # se t1 = 0, saltar para fim

        addi $t2,$t2,1 # t2 <- t2 + 1
        addi $t0,$t0,1 # t0 <- t0 + 1
        j enquanto
fim:    sw $t2,n
```

Este código corresponde ao seguinte programa em pseudo-código:

```
variáveis:
        string cadeia = "AC-Uma";
        int n;
início: {
        i = 0;
        n = 0;
        enquanto (cadeia[i] != 0) fazer {
                n = n + 1;
                i = i + 1;
        }
}
```

Q4.5. Comprove o efeito deste programa carregando-o no SPIM. Que valor fica armazenado na variável `n`?

Q4.6. Implemente o seguinte programa escrito em pseudo-código:

```
variáveis:
        string cadeia1 = "AC-Uma";
        string cadeia2 = "Labs";
        int n;
início: {
        i = 0;
        n = 0;
        enquanto (cadeia1[i] != 0) e (cadeia2[i] != 0) fazer {
                n = n + 1;
                i = i + 1;
        }
}
```

Estruturas de Controlo do Tipo "Para…"

Por vezes torna-se útil implementar ciclos do "Para…". Estes ciclos constituem ciclos "Enquanto…" onde se repete o código do ciclo um número de vezes conhecido *a priori*.

```
        .data
vector: .word 1,2,3,4,5,6
res:    .space 4
        .text
main:   la $t2,vector  # t2 <- endereço do vector
        and $t3,$0,$t3 # t3 <- 0
        li $t0,0       # t0 <- 0 # equivalente ao i
        li $t1,6       # t1 <- 6
para:
        bgt $t0,$t1,fimpara   # se t0 > t1 saltar para fimpara
        lw $t4,0($t2)         # carrega elemento do vector em t4
        add $t3,$t4, $t3      # soma os elementos do vector
        addi $t2,$t2, 4       # t2 <- t2 + 4
        addi $t0,$t0, 1       # t0 <- t0 + 1
        j para                # continua o ciclo
fimpara:
        sw $t3, res($0)       # armazenar t3 em res
```

A descrição em pseudo-código deste programa Assembly pode ser vista como uma estrutura do tipo "Enquanto…":

```
variáveis:
        int vector[] = {1,2,3,4,5,6};
        int res;
início: {
        res = 0;
        i = 0;
        enquanto (i <= 6) fazer {
                res = res + vector[i];
                i = i + 1;
        }
}
```

Mas também pode ser vista como uma estrutura do tipo "Para…" (semelhante ao `for` do C, disponível também em muitas outras linguagens de alto nível como Java, Pascal, etc.):

```
 variáveis:
        int vector[] = {1,2,3,4,5,6};
        int res;
início: {
        res = 0;
        i = 0;
        para i de 0 até 5 fazer {
                res = res + vector[i];
                i = i + 1;
        }
}
```

Um Exemplo

Como exemplo, sigamos o seguinte programa, que implementa o programa em C equivalente (em comentário):

```
        .data                   # declara segmento de dados
str:    .asciiz "AC-Uma"        # uma string arbitrária terminada por nulo
#
# Princípio do Segmento de Texto
#
        .text   # declara segmento de código
main:
        add $t1,$0,$0       # inicializa apontador (index)
# Este código SPIM implementa o seguinte programa em C:
#
#   int t1;
#   t1=0;
#   while (str[t1] != '\0')
#      t1++;
#   return t1;
#
# Neste loop do SPIM, o valor de $t1 assume o papel da variável i (em C)
ciclo:  lb $t5,str($t1)         # carrega byte (ler um caracter)
        beq $t5,$0,fim          # termina se for caracter sentinela ('\0')
        addi $t1,$t1,1          # incrementa offset (por 1, não 4; porquê?)
        j ciclo                 # repete

fim:    addi $v0,$0,1           # $v0=1 (para imprimir inteiro no $a0)
        add $a0,$t1,$0          # a0 <- t1
        syscall                 # chamada sistema print_int
        addi $v0,$0,10          # $v0=10 (para retornar o controlo ao SPIM)
        syscall
```

Q4.7. Escreva um programa em Assembly do MIPS que declare um vector V = [2, 3, 4, 5, 6] e uma variável `n = 5`. O programa deverá produzir o número de elementos maiores que 6.

Q4.8. Escreva um programa em Assembly do MIPS que copie uma *string* `x` para uma *string* `y`.

5
Procedimentos e Gestão de Subrotinas

Wait a minute, Doc. Are you telling me you built a time machine out of a DeLorean?

Marty McFly para o Dr. Brown no filme de 1985, "Back to the future"

O desenho de um programa pode ser simplificado subdividindo o problema a resolver em um ou mais sub-problemas, seguindo a abordagem de dividir para conquistar. Neste capítulo, iremos aprender o mecanismo de chamada de procedimentos (em Assembly por vezes designadas por subrotinas) do MIPS. Estas subrotinas permitem escrever um programa complexo de uma forma modular, permitindo assim uma melhor depuração de erros e também a reutilização de subrotinas ao longo de todo o programa.

Quando um procedimento (o *caller*) chama outro procedimento (o *callee*), o controlo do programa é transferido do *caller* para o *callee*, isto é, o controlo é transferido do procedimento que chama para o procedimento chamado. As instruções que tornam possível o correcto funcionamento deste mecanismo são as instruções `jal` e `jr`, que analisaremos em detalhe.

Os quatro passos necessários para executar um procedimento são:

1. Guardar o endereço de retorno. O endereço de retorno é o endereço da instrução que surge imediatamente após o ponto da chamada.
2. Chamar o procedimento.
3. Executar o procedimento.
4. Recuperar o endereço de retorno e retornar ao ponto inicial (que é o ponto da chamada).

A arquitectura do MIPS fornece duas instruções que são utilizadas em conjunção para realizar a chamada e retornar:

```
jal nome_do_procedimento
```

Esta instrução guarda no registo `$ra` o endereço da instrução que surge imediatamente após o jal (endereço de retorno) e salta para o procedimento cuja etiqueta se especifica.

Esta instrução implementa os passos 1. e 2. Para o retorno (passo 4), utiliza-se a instrução

```
jr $ra
```

a qual salta para o endereço armazenado no registo `$ra`. Durante a execução o conteúdo de `$ra` substitui o conteúdo do registo `PC`. A instrução `jr` também pode ser utilizada conjuntamente com outros registos e não é necessário que seja usada apenas para a implementação do procedimento de retorno, por exemplo,

```
lui $t0, 0x50  # t0 <- 0x00500000
jr $t0         # o fecth da próxima instrução será a partir do endereço
               # 0x00500000
```

É obrigatório salvaguardar o endereço de retorno (`$ra`) quando se entra num procedimento que chama outros procedimentos. Se escolhermos guardar o `$ra` num registo, então é necessário que o registo escolhido seja um daqueles que são preservados ao longo de chamadas de procedimentos (consultar a Tabela A.10, "Registers Usage Conventions", do Manual). Melhor ainda é guardar o registo `$ra` na pilha.

Gestão da Pilha

A instrução `jal` guarda o endereço de retorno no registo `$ra`. Vamos assumir agora que um procedimento pretende chamar um procedimento. Então, o programador tem de guardar o registo `$ra` porque senão o novo `jal` irá escrever por cima do seu valor. Uma vez que o número de chamadas embebidas não tem limite (procedimentos podem chamar outros procedimentos que também chamam procedimentos etc.) não podemos utilizar uma localização fixa (seja ela um registo ou uma posição de memória) para guardar o registo.

Em vez disso, necessitamos de uma estrutura dinâmica, qualquer coisa que nos permita armazenar dados mas sem escrever por cima de dados previamente guardados.

Como não podemos utilizar o DeLorean do Dr. Brown, e como sabemos que o último dado guardado será o primeiro que iremos querer utilizar (o último valor de `$ra` guardado será o primeiro a ser utilizado aquando do retorno), utilizamos uma estrutura que designamos por pilha (*stack*).

A arquitectura do MIPS descreve uma pilha cujo topo começa por ter um valor elevado (0x7fff ffff) na memória principal e que cresce para baixo, isto é, sempre que se coloca alguma coisa na pilha – operação designada por *push* – o endereço do topo do *stack* decresce. O registo `$sp` (*stack pointer*) aponta sempre – se o programa estiver correcto – para o topo da pilha, isto é, para a primeira posição vazia na pilha.

Uma vez que a pilha é apenas uma secção da memória principal, iremos acedê-la usando as mesmas intruções de *load*/*store* que usamos para aceder ao segmento de dados. Por exemplo, para guardar o valor de um registo na pilha, faz-se

```
add $sp,$sp,-4 # primeiro actualizar o stack pointer
sw $ra, 4($sp) # push do $ra para o stack
```

Começamos por subtrair a quantidade correcta ao endereço do *stack pointer*, uma vez que a pilha vai aumentando para baixo (na memória principal). Neste exemplo subtrai-se 4 porque queremos guardar uma palavra (4 bytes) que está no registo $ra. Depois fazemos o armazenamento (store) usando um deslocamento positivo: 4($sp) significa o endereço contido em $sp + 4, a fim de apontarmos para a posição de memória vazia para onde o $sp apontava antes de o modificarmos.

Esta operação designa-se por push. Então e como fazer para recuperar um valor previamente salvaguardado na pilha de volta para o seu registo? Esta operação é a inversa do push, e designa-se por pop:

```
lw $ra,4($sp)  # recupera-se o dado da pilha
add $sp,$sp,4  # actualiza-se o $sp: encolhe-se o stack uma palavra
```

No caso de se pretender guardar vários registos, repete-se a operação:

```
add $sp,$sp,-8 # queremos guardar 4 palavras (8 bytes)
sw $s0,8($sp)
sw $s1,4($sp)
```

Passagem de Parâmetros

Existem duas formas de passar parâmetros a um procedimento: pelos registos e pela pilha. A convenção de utilização dos registos no MIPS especifica que os 4 primeiros argumentos de um procedimento são passados por registos (`$a0-$a3`). Os restantes são passados pela pilha.

Passar parâmetros pelos registos é eficiente porque evita acessos à memória. Se o procedimento chamar outros procedimentos, então os registos que contiverem parâmetros têm de ser salvaguardados (`$a0-$a3` não são preservados ao longo das chamadas). Essa salvaguarda pode ser realizada

- Nos registos que são preservados ao longo das chamadas (`$s0-$s7`);
- Na pilha.

Chamando um Procedimento

O *caller* (procedimento que chama, ou procedimento chamante) realiza o seguinte antes de chamar outro procedimento:

1. Guarda quaisquer registos que não são preservados ao longo de chamadas (i.é, `$a0-$a3` e `$t0-$t9`) e que se espera que sejam usados após a realização da chamada.
2. Passa os parâmetros: os 4 primeiros são passados em `$a0-$a3`, os restantes são passados pela pilha.
3. Salta para o procedimento usando a instrução `jal`.
4. Após a chamada, ajusta o `$sp` para apontar acima dos argumentos passados na pilha (isto é equivalente a fazer um *pop*, só que nada é lido da pilha).

O *callee* (procedimento chamado) faz o seguinte:

1. Guarda o `$ra` se o procedimento for chamar outro procedimento.
2. Guarda quaisquer registos `$s0-$s7` que o procedimento modifique.
3. Realiza o que tem que realizar.
4. No caso de retornar um valor, coloca esse valor no registo `$v0`.
5. Restaura todos os registos previamente guardados (*pop* da pilha).
6. Retorna, executando `jr $ra`.

Exemplo

O seguinte programa constitui um procedimento que copia uma string x para uma string y:

```
# Arquitectura de Computadores
# Programa que copia uma string y para uma string x
                .data
str1:   .asciiz "origem "
str2:   .asciiz "copia "
                .text
main:           addi $sp, $sp, -4       # reserva uma posição no stack
                sw $ra, 0($sp)          # guarda o endereço de retorno
                jal strcpy
                lw $ra, 0($sp)          # recupera endereço de retorno
```

```
                addi $sp, $sp, 4        # pop do stack pointer
end:            jr $ra
strcpy:
                addi $sp, $sp, -4
                sw $s0, 0($sp)          # guarda $s0 no stack
                add $s0, $0, $0         # $s0 = i = 0
                la $t5, str1            # guarda endereço de str1 em $t5
                la $t6, str2            # guarda endereço de str2 em $t6
L1:
                add $t1, $s0, $t5 # carrega em $t1 endereço de str1 que é o y(i)
                lb $t2, 0($t1)          # carrega o byte y(i) em $t2
                add $t3, $s0, $t6 # carrega em $t3 endereço de str2 que é o x(i)
                sb $t2, 0($t3) # armazena em x(i) o byte y(i) que está em $t2

                addi $s0, $s0, 1        # incremento de 1 (byte seguinte)
                bne $t2, $0, L1         # se y(i) != 0 então continua a copiar

                lw $s0, 0($sp)          # senão, recupera antigo valor de $s0
                addi $sp, $sp, 4
                jr $ra
```

Q5.1. (3º Trab. Lab. 2003-2004) Escreva um programa com um procedimento em Assembly do MIPS R2000 que dada uma cadeia de caracteres na memória, conte o número de palavras dessa cadeia e imprima o resultado no écran.

O argumento do procedimento (o endereço da cadeia de caracteres) deverá ser passado no registo `$a0`. O procedimento deverá usar o registo `$s0` para contar o número de palavras, o registo `$s1` como contador e o registo `$s2` para armazenar o argumento. Por isso, o procedimento deve guardar estes registos na pilha, repôndo os seus valores antes de retornar.

Não se esqueça também que os procedimentos em Assembly do MIPS devolvem sempre o(s) resultado(s) no(s) registo(s) `$v0-$v1` (ver Tabela A.10 do Manual).

O código do caracter espaço (‘ ‘) é o 32 (0x20), pelo que deverá colocá-lo num registo a fim de efectuar comparações:

```
addi $t0, $0, 32        # coloca caracter espaço em $t0= 0x20
```

3º Trabalho de Laboratório

Procedimentos Recursivos

Um procedimento recursivo é um procedimento que se chama a si mesmo. Contudo um procedimento recursivo tem eventualmente que terminar. Logo, um ingrediente essencial de um procedimento recursivo é a condição de paragem, a qual especifica quando deve um procedimento recursivo parar de se chamar a si mesmo.

Um outro ponto a ter em conta é o facto de o procedimento chamar-se a si mesmo de cada vez com argumentos (parâmetros) diferentes, caso contrário a computação não acaba.

Vamos começar por utilizar o clássico exemplo da função factorial. A sua definição indutiva é a seguinte:

- Base: $0! = 1$
- Indução: $N! = (N - 1)! \cdot N$, para $N \geq 1$

Esta definição traduz-se simplesmente no programa recursivo que se segue:

```
int factorial(int num) {
        int fact;
        if (num == 0) return 1;       /* condição de paragem */
        fact = factorial(num-1);      /* chamada recursiva */
        return fact;
}
```

Em Assembly do MIPS, o factorial codifica-se neste exemplo:

```
# Este procedimento calcula o factorial de um inteiro positivo
# O argumento é recebido em $a0
# O resultado (um inteiro de 32 bits) é devolvido em $v0
# Não usa nenhum registo $s0-$s7 por isso não é preciso guardá-los
factorial:
        subu $sp, $sp, 4
        sw $ra, 4($sp)          # guarda endereço de retorno no stack
        beqz $a0, termina       # testa base da recursão
        subu $sp, $sp, 4        # ainda não termina
        sw $a0, 4($sp)          # guarda o argumento
        sub $a0, $a0, 1         # será chamado com n-1
        jal factorial  # após a base da recursão ser executada, estas linhas
                       # serão executadas
        lw $t0, 4($sp)          # argumento que guardei no stack
        mul $v0, $v0, $t0       # n*(n-1)
        lw $ra, 8($sp)          # preparar o retorno
        addu $sp, $sp, 8        # retirei do stack dois elementos (um endereço e
                                # um argumento)
        jr $ra
termina:
        li $v0, 1               # 0! = 1 é o valor de retorno
        lw $ra, 4($sp)          # obtém endereço de retorno
        addu $sp, $sp, 4        # ajusta o stack pointer
        jr $ra          # retorna
```

Q5.2. Baseando-se no exemplo anterior, crie um novo programa com as seguintes alterações:

- No `main` pede ao utilizador que introduza um inteiro; armazene-o em `$t0`;

- Verifica se o número é negativo. Se for, imprime uma mensagem de erro e volta a pedir outro número ao utilizador;
- Chama o procedimento factorial (com o inteiro introduzido) e retorna o factorial desse número;
- Passa o parâmetro num registo;
- Imprime uma mensagem e o valor do factorial.

Q5.4. Quantas chamadas recursivas são realizadas para n=5?

Q5.5. Cada vez que uma chamada recursiva é feita, a pilha aumenta. Por quantos bytes por chamada?

Q5.6. Uma vez que a pilha cresce cada vez que uma chamada recursiva é feita, a memória pode acabar. Assumindo que a dimensão da pilha é 2 000 000 bytes, que valor de entrada iria gerar chamadas suficientes para "rebentar" com a pilha?

Q5.7. Escreva um programa que calcule o número de Fibonacci.

6
Gestão das Entradas/Saídas por Consulta de Estado

"Nunca há tempo para fazer todo o nada que se quer."

Calvin, em "Calvin&Hobbes" de Bill Watterson

Os computadores não funcionam isoladamente: têm de relacionar-se com outros periféricos: teclado, impressora, monitor, etc. Os periféricos comunicam com o computador por intermédio de um sistema, chamado sistema de entradas/saídas (E/S, ou I/O, de *Input/Output*).

O elemento do computador que permite essa comunicação é designado por interface. A interface é constituída por um circuito, chamado controlador ou dispositivo de E/S.

Estas interfaces e controladores são específicas de cada periférico e o computador comunica com eles a fim de programá-los e/o comunicar com os periféricos, lendo e escrevendo em elementos endereçáveis chamados portos.

Quando os endereços destes portos correspondem a endereços do espaço de memória do computador, para ler ou escrever nestes portos utiliza-se as mesmas instruções que lêem/escrevem da memória. Neste caso diz-se que o sistema de E/S se encontra mapeado na memória. É este o sistema utilizado no caso do processador MIPS.

Se, por outro lado, se utilizam instruções especiais, diz-se que o computador possui um mapa de endereços de E/S independente (caso da Intel).

Neste capítulo abordamos a forma mais simples de interagir com os periféricos (i. é, gerir as entradas/saídas). Essa forma designa-se por consulta de estado. Os periféricos que vamos usar são o teclado (entrada) e a consola (saída).

A técnica de consulta de estado pressupõe que o controlador do dispositivo possua um porto de leitura/escrita de dados e pelo menos um porto de controlo. A leitura, por parte do processador, deste porto de controlo, e em particular de um bit chamado bit de estado, indica se o controlador está pronto para ler ou escrever dados através do periférico mediante o porto de leitura/escrita de dados.

E… no SPIM?

No SPIM, para a entrada de dados a partir do teclado, utiliza-se o porto de dados cujo endereço é o `0xffff0004`. O porto de controlo situa-se no endereço `0xffff0000` (sendo o bit 0 o bit de estado, o qual indica que o controlador possui um caracter digitado no teclado.

Para a saída de dados para a consola, o porto de dados mora no endereço `0xffff000c` e o porto de controlo no endereço `0xffff0008` (sendo o bit 0 o bit de estado que indica que se pode escrever um caracter para a consola).

Entrada de Dados (Teclado)

Nesta secção vamos estudar o controlo da entrada de dados a partir do teclado do computador. Vamos estudar o seguinte código:

```
        .data 0x10010000
long:   .word 7         # dimensão do buffer
buffer: .space 7        # buffer onde se armazenam os caracteres teclados
        .data 0xffff0000
cin:    .space 1        # porto de controlo do teclado
        .data 0xffff0004
in:     .space 1        # porto de leitura do teclado
        .text
        .globl main
main:   la $a0, buffer          # carrega endereço do buffer
        lw $v0,long($0)         # controlo da longitude do buffer
        addi $v0,$v0,-1
        li $v1,0x0a     # caracter return, muda de linha (\n)
        la $a1, in      # carrega endereço in
        la $a2, cin     # carrega endereço de controlo do teclado
ctr:
        # por agora não colocamos nada

        lb $s0, 0($a1) # lê do porto do teclado
        sb $s0,0($a0)  # armazena o dado no buffer
        addi $a0,$a0,1 # incrementa apontador do buffer
        addi $v0,$v0,-1         # decr. tamanho restante buffer
        bne $v0,$0, ctr         # controlo de fim de string

fim:    li $s0, 0x00
        sb $s0,0($a0)  # armazena fim de string no buffer
        jr $ra          # retorna ao main
```

Este código constitui um programa que lê os caracteres introduzidos a partir do teclado, sem contudo efectuar um correcto controlo da entrada de dados (eliminado propositadamente do código, ver etiqueta `ctr`:).

Com o simulador executado com a opção "Mapped I/O", carregue e execute este programa.

Q6.1. Que caracter é armazenado no espaço reservado em buffer?

Vamos colocar agora o código necessário para que se leia do porto de dados do teclado quando um caracter é teclado. Inserimos o seguinte código na etiqueta `ctr`:

```
############## consulta de estado #############
ctr:    lb $t0, 0($a2)
        andi $t0,$t0,1
        beq $t0,$0, ctr
################################################
```

Q6.2. O que faz este código? Verifique o novo comportamento do programa.

Q6.3. Modifique o programa para que, quando se carregue na tecla "Return" o programa armazene o caracter fim de linha (0) e termine.

Saída de Dados (Consola)

À semelhança do exercício anterior, iremos agora testar o controlo da saída de dados pela consola do simulador. Crie um ficheiro com o seguinte código:

```
        .data 0x10010000
cadeia: .asciiz “Agora sim, sai bem!”
        .data 0xffff0008
cout:   .space 1 # Porto de controlo da consola
        .data 0xffff000c
out:    .space 1 # Porto de escrita da consola
        .text
        .globl main
main:   la $a0,cadeia  # carrega endereço da cadeia
        la $a1,out     # carrega endereço de saída
        lb $s0,0($a0)  # lê o caracter (byte) da cadeia
        la $a2,cout    # carrega endereço de controlo da saída
############ ciclo de consulta de estado ##########
ctr1:
##################################################
        sb $s0, 0($a1) # escreve na consola
        addi $a0,$a0,1 # incrementa apontador para a cadeia
        lb $s0, 0($a0) # lê o caracter (byte) seguinte
        bne $s0,$0,ctr1 # controlo de fim de cadeia (0)
############ ciclo de consulta de estado ##########
ctr2:
##################################################
        sb $0, 0($a1)  # escreve na consola fim de cadeia (0)
        jr $ra         # regressa à rotina principal
```

Q6.4. Este código lê os caracteres armazenados na memória a partir do endereço `cadeia` e escreve-os no porto de saída de dados para a consola do simulador, `out`. Contudo, não realiza correctamente o controlo da saída de dados por consulta do estado (propositadamente deixado de fora, tal como no caso anterior).

Executemos o programa. Como se pode verificar, a frase armazenada na memória não aparece.

Q6.5. Introduzimos então o código seguinte nas etiquetas `ctr1` e `ctr2`:

```
############### ciclo de atraso ##############
        li $t0,n # n deve ser substituído por um valor
cont:   addi $t0,$t0,-1
        bnez $t0,cont
##############################################
```

Vamos substituir `n` por 10 e executar o programa.

Q6.6. Incremente progressivamente o valor de `n`. A partir de que valor começa a surgir a cadeia inteira na consola? O que estamos a fazer, ao aumentar o valor de `n`?

Q6.7. Esse valor de `n` obtido teria que ser o mesmo para outro processador? Porquê?

Q6.8. Seguindo o exemplo do programa da questão 6.1, realize o controlo da saída de dados por consulta de estado.

Entrada/Saída

Vamos agora estudar um exemplo: este programa permite ler os caracteres teclados (até um total de 10) e mostrá-los na consola:

```
        .data
long:   .word 10       # tamanho do buffer
        .data 0xffff0000
```

```
cin:    .space 1
        .data 0xffff0004
in:     .space 1
        .data 0xffff0008
cout:   .space 1
        .data 0xffff000c
out:    .space 1
        .text
main:   addi $sp,$sp,-4
        sw $ra,0($sp)
        lw $s0,long
        addi $s0,$s0,-1
        li $t6,0x0a    # return
        la $t1, in     # carrega endereço in
        la $t2, cin    # carrega endereço de controlo do teclado
        la $t3, out    # carrega endereço out
        la $t4, cout   # carrega endereço de controlo da saída
ctri:   jal wi
        lb $t7, 0($t1) # leio do teclado
        jal wo
        sb $t7, 0($t3) # escrevo na consola
        addi $s0,$s0,-1
        beq $t7,$t6, fim       # controlo fim de linha
        bne $s0,$0, ctri       # controlo de fim de cadeia
        j zero
fim:    li $t7, 0x0a
        jal wo
        sb $t7, 0($t3) # escrevo na consola
zero:   andi $t7,$t7,0
        jal wo
        sb $t7, 0($t3) # escrevo na consola
        lw $ra, 0($sp)
        addi $sp,$sp,4
        jr $ra
############ Consulta de estado da entrada #########
wi:
##################################################
############ Consulta de estado da saída ##########
wo:
##################################################
```

As rotinas `wi` e `wo` (omitidas) realizam o controlo de consulta de estado dos periféricos.

Q6.9. Complete o programa anterior com as rotinas de controlo dos periféricos (`wi` e `wo`) e comprove o funcionamento correcto do programa. Baseie-se nos exemplos anteriores.

Q6.10. (4º Trab. Lab. 2003-2004) Escreva um programa em Assembly do MIPS R2000 que peça um inteiro do teclado e guarde-o em memória. O programa não pode utilizar a chamada de sistema `syscall` (isto é, deve gerir as E/S por consulta de estado).

4º Trabalho de Laboratório

7
Gestão das E/S por Interrupções

Never interrupt your enemy when he is making a mistake.

Napoleão Bonaparte

Neste capítulo iremos conhecer o mecanismo de excepções do processador MIPS e no fim deste capítulo seremos capazes de escrever código de tratamento de excepções (*exception handlers*) para o MIPS.

Os saltos condicionais (*branches*) e incondicionais (*jumps*) constituem uma forma de alterar o controlo de fluxo de um programa. As excepções também são uma maneira de alterar o controlo de fluxo de um programa em Assembly.

A convenção do MIPS considera uma excepção toda e qualquer alteração inesperada no controlo do fluxo de um programa, independentemente da sua origem (i. é, não há distinção entre uma fonte externa ou uma fonte do próprio processador).

Uma excepção é síncrona se ocorrer no mesmo sítio de cada vez que um programa é executado com os mesmos dados e com a mesma alocação de memória. Os *overflows* aritméticos, instruções não definidas, *page faults*, são exemplos de excepções síncronas.

As excepções assíncronas, por outro lado, ocorrem sem qualquer relação temporal com o programa que é executado. Pedidos de E/S, erros de memória, falhas de fornecimento de energia etc. são exemplos de excepções assíncronas.

Uma interrupção é uma excepção assíncrona. As excepções síncronas, que resultam directamente da execução do programa, são designadas por *traps*.

Quando uma excepção ocorre, o controlo é transferido para um programa diferente, chamado *exception handler* (rotina de tratamento da excepção), escrito explicitamente para lidar com as excepções. Após a excepção, o controlo é devolvido ao programa que estava a ser executado na altura em que ocorreu a excepção: esse programa continua como se nada tivesse acontecido. Uma excepção faz parecer como se uma rotina (sem parâmetros e sem valor de retorno) tivesse sido inserida no programa.

Uma vez que o código de tratamento da excepção pode ser executado a qualquer altura, não pode haver passagem de parâmetros para esse código: tal passagem requer uma preparação prévia. Pela mesma razão não pode haver retorno de valores. É importante ter em conta que o exception handler tem de preservar o estado do programa que foi interrompido de modo a que a sua execução possa prosseguir mais tarde.

Tal como acontece com qualquer outro procedimento, o *exception handler* tem de guardar quaisquer registos que possa modificar, restaurando o seu valor antes de devolver o controlo ao programa interrompido. Guardar os registos na memória é um problema no MIPS: endereçar a memória requer um registo (o registo base) no qual o endereço está formado. Isto significa que um registo tem de ser modificado antes que qualquer registo possa ser guardado! A convenção de utilização dos registos no MIPS (ver Tabela do Manual) reserva os registos `$26` e `$27` (`$k0` e `$k1`) para utilização por parte do *interrupt handler*.

Isto significa que o *interrupt handler* pode usar estes registos sem primeiro ter de os salvar. Um programa escrito para o MIPS que use estes registos pode encontrá-los subitamente alterados (daí serem considerados como registos reservados).

O mecanismo de excepções do MIPS

O mecanismo de excepções é implementado no coprocessador 0, que está sempre presente no MIPS (ao contrário do coprocessador 1, que pode ou não estar presente). O sistema de memória virtual também é implementado no coprocessador 0. Note, contudo, que o SPIM não simula esta parte do processador.

O CPU opera em dois modos possíveis: user e kernel. Os programas do utilizador correm no modo user. O CPU entra no modo kernel quando ocorre uma excepção. O Coprocessador 0 só pode ser utilizado no modo kernel.

Toda a metade superior do espaço de memória é reservado para o modo kernel: não pode, por isso, ser acedido no modo utilizador. Ao correr no modo kernel, os registos do coprocessador 0 podem ser acedidos usando as seguintes instruções:

Instrução	Comentário
`mfc0 Rdest, C0src`	Move o conteúdo do registo do Coprocessador 0 `C0src` para o registo destino `Rdest`.
`mtc0 Rsrc, C0dest`	O registo de inteiros `Rsrc` é movido para o registo `C0dest` do Coprocessador 0.
`lwc0 C0dest, address`	Carrega palavra a partir do endereço `address` para o registo `C0dest`.
`swc0 C0src, address`	Armazena o conteúdo do registo `C0src` no endereço `address` na memória.

Os registos do coprocessador 0 relevantes para o tratamento das excepções no MIPS são:

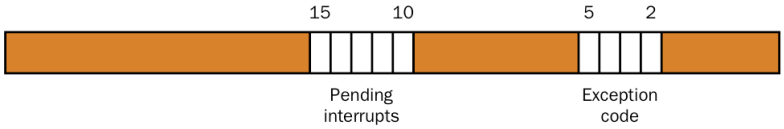
Nº do Registo	Nome	Utilização
8	`BadVAddr`	Endereço de memória onde ocorreu a excepção.
12	`Status`	Máscara da interrupção, *bits enable* e *status* na altura da interrupção.
13	`Cause`	Tipo de excepção e *bits* de interrupções pendentes.
14	`EPC`	Endereço da instrução que causou a interrupção.

O registo BadVAddr

Este registo, cujo nome vem de Bad Virtual Address, contém o endereço de memória onde ocorreu a excepção. Um acesso desalinhado à memória, por exemplo, irá gerar uma excepção e o endereço onde o acesso foi tentado será guardado em BadVAddr.

O registo Cause

O registo Cause fornece informação sobre quais as interrupções pendentes (bits 10 a 15) e a causa da excepção. O código da excepção é armazenado como um inteiro sem sinal usando os bits 5-2 no registo Cause. A figura seguinte representa este registo:



O bit *i* do grupo de bits *Pending Interrupts* fica a 1 se uma interrupção ocorre no nível *i* e está pendente (diz-se que ainda não foi servida).

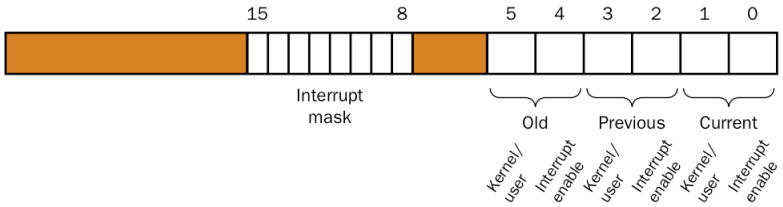
O código da excepção indica o que causou a excepção:

Código	Nome	Descrição
`0`	`INT`	**Interrupção**.
`4`	`ADDRL`	*Load* a partir de um endereço ilegal.
`5`	`ADDRS`	*Store* para um endereço ilegal.
`6`	`IBUS`	Erro do *bus* ao efectuar *fetch* de uma instrução.
`7`	`DBUS`	Erro do *bus* numa referência a dados.
`8`	`SYSCALL`	Instrução `syscall` executada.
`9`	`BKPT`	Instrução `break` executada.
`10`	`RI`	Instrução reservada.
`12`	`OVF`	*Overflow* aritmético.

O código 0 indica que uma interrupção ocorreu. Olhando para os bits *Pending Interrupts* individuais, o processador pode saber que interrupção específica é que ocorreu.

O registo Status

O registo Status contém uma máscara de interrupções nos bits 5-10 e informação de estado nos bits 5-0. O esquema do registo é o seguinte:



Se o bit *Interrupt mask i* estiver a 1, então as interrupções do nível *i* estão activas (*enabled*). Caso contrário estão inactivas (*disabled*).

O bit 1 no registo indica se o programa está a correr no modo utilizador (=1) ou kernel (=0). O bit 3, indica se o processador estava a correr no modo kernel (=0) ou utilizador na altura em que ocorreu a última excepção. Esta informação é importante porque ao retornar do *exception handler*, o processador tem de estar no mesmo estado em que estava antes de ocorrer a excepção.

Os bits 5-0 no registo Status implementam então uma pilha de três níveis com informação acerca de excepções anteriores. Quando uma excepção ocorre, o estado *previous* (bits 3 e 2) é guardado como o estado old e o estado *current* é guardado como estado previous. O estado *old* é perdido. Os bits do estado *current* são ambos postos a zero (modo kernel com interrupções inactivas). Ao retornar do *exception handler* (executando uma instrução que já veremos, designada por `rfe`) o estado *previous* torna-se o estado *current* e o estado *old* torna-se em *previous*. O estado *old* não é alterado.

Os bits de activação de interrupções (Interrupt Enable), indicam se as interrupções estão activas (ou habilitadas, ou permitidas), caso em que estão a 1, ou inactivas, caso em que estão a 0, no respectivo estado. Por exemplo, se o bit 0 é zero, então o processador está a correr actualmente com as interrupções inactivas.

O registo EPC

Quando um procedimento é chamado usando `jal`, duas coisas acontecem:

- O controlo é transferido para o endereço fornecido pela instrução.
- O endereço de retorno é guardado no registo `$ra`.

No caso de uma excepção, não há uma chamada explícita. No MIPS, o controlo é transferido para uma localização fixa, 0x80000080, quando uma excepção ocorre. O *exception handler* tem de ficar nesse endereço.

O endereço de retorno não pode ser salvo em `$ra`, uma vez que pode escrever por cima de um endereço de retorno que tenha eventualmente sido colocado nesse registo antes da excepção. O registo EPC (*Exception Program Counter*) é usado para armazenar o endereço da instrução que estava sendo executada quando a excepção foi gerada.

Retornar de uma excepção

A sequência de retorno é *standard*:

```
#
# the return sequence from the exception handler for the case of an
# external exception (interrupt)
#
        mfc0 $k0, $14 # get EPC in $k0
        rfe # return from exception
        jr $k0 # replace PC with the return address
```

A arquitectura estabelece uma distinção clara entre interrupções (excepções externas) e *traps* (incluindo invocações explícitas por software, como o `syscall`). No caso de uma interrupção, o Program Counter é avançado para apontar para a próxima instrução no momento em que o controlo foi transferido para o *exception handler*. Por outras palavras, o EPC contém o endereço da instrução a executar após o retorno do exception handler. No caso de um *trap* ou `syscall`, o EPC contém o endereço da instrução que gerou o *trap*.

Para evitar executar a instrução outra vez ao retornar do *exception handler*, o endereço de retorno tem de ser incrementado por 4. Assim, assegura-se que a instrução que se segue no fluxo do programa será executada.

```
# the return sequence from the exception handler for the case of a
# trap (including a syscall).
#
        mfc0 $k0, $14  # get EPC in $k0
        addiu $k0, 4   # make sure it points to next instruction
```

```
        rfe             # return from exception
        jr $k0          # replace PC with the return address
```

Compreendendo um *Exception Handler*

Resumindo, quando se gera uma excepção, o processador leva a cabo as seguintes acções:

1. Salvaguarda o endereço da instrução que gerou a excepção no registo EPC.
2. Armazena o código da excepção produzida no registo Cause. Quando se trata de uma interrupção, activa o bit que indica o nível do pedido.
3. Salvaguarda o registo Status no próprio registo de Status, deslocando a informação dos 6 primeiros bits 2 posições para a esquerda, e coloca os dois primeiros bits a 0, indicando que se passa ao modo *kernel* e que as interrupções estão mascaradas.
4. Carrega no Program Counter um endereço de memória fixo (0x80000080 no espaço de endereçamento do kernel). Este é o endereço a partir do qual está armazenada a rotina de serviço geral de tratamento de excepções (*interrupt handler*, que se encontra no ficheiro `trap.handler`). Esta rotina determina a causa da excepção e salta para o local adequado do Sistema Operativo a fim de tratar a excepção. A rotina de tratamento da excepção responde a uma excepção fazendo terminar o programa que a causou ou realizando uma determinada sequência de acções.

Descrição do controlador do teclado simulado pelo SPIM

O controlador do teclado que é simulado pelo SPIM dispõe de dois registos de controlo e de dados.

O registo de controlo está no endereço 0xffff0000 e só se usam dois dos seus bits. O bit 0 é o bit de estado e já vimos o seu funcionamento no capítulo anterior. O bit 1 é o bit de permissão de interrupções. Quando está a 1 significa que a interrupção deste dispositivo é permitida (está activa, ou habilitada). A zero não está permitida. Este bit pode ser modificado por um programa.

Se este bit estiver a 1, o dispositivo solicitará uma interrupção de nível 0 ao processador quando o bit de estado também estiver a 1.

De todos os modos, para que o processador atenda esta interrupção, todas as interrupções devem estar activas também no registo de estado (Status) do processador (bits 0 e bits de 8 a 1).

O registo de dados localiza-se no endereço 0xffff0004.

Exemplo: Controlo da entrada de Dados perante Interrupções

Este programa é adaptado do ficheiro `trap.handler` do núcleo do SPIM:

```
                .data 0x10010000
perm:           .word 0x00000101
caracter:       .asciiz "----------------------\n"
lido:           .word 0

                .data 0xffff0000
cin:            .space 1        # controlo de leitura
                .data 0xffff0004
```

```
in:             .space 1        # porto de leitura
                .data 0xffff0008
cout:           .space 1        # controlo de escrita
                .data 0xffff000c
out:            .space 1        # porto de escrita

# Agora vem a zona de dados do Sistema Operativo do MIPS:

        .kdata
inter:  .asciiz " int "         # mensagem que aparece quando o processador
                                # atende a interrupção

__m1_:  .asciiz "  Exception "
__m2_:  .asciiz " occurred and ignored\n"
__e0_:  .asciiz "  [Interrupt] "
__e1_:  .asciiz ""
__e2_:  .asciiz ""
__e3_:  .asciiz ""
__e4_:  .asciiz "  [Unaligned address in inst/data fetch] "
__e5_:  .asciiz "  [Unaligned address in store] "
__e6_:  .asciiz "  [Bad address in text read] "
__e7_:  .asciiz "  [Bad address in data/stack read] "
__e8_:  .asciiz "  [Error in syscall] "
__e9_:  .asciiz "  [Breakpoint] "
__e10_: .asciiz "  [Reserved instruction] "
__e11_: .asciiz ""
__e12_: .asciiz "  [Arithmetic overflow] "
__e13_: .asciiz "  [Inexact floating point result] "
__e14_: .asciiz "  [Invalid floating point result] "
__e15_: .asciiz "  [Divide by 0] "
__e16_: .asciiz "  [Floating point overflow] "
__e17_: .asciiz "  [Floating point underflow] "
__excp: .word __e0_,__e1_,__e2_,__e3_,__e4_,__e5_,__e6_,__e7_,__e8_,__e9_
        .word __e10_,__e11_,__e12_,__e13_,__e14_,__e15_,__e16_,__e17_
s1:     .word 0
s2:     .word 0

# Rotina do serviço geral de excepções armazenada no núcleo do S.O.
        .ktext 0x80000080
        .set noat
        # Because we are running in the kernel, we can use $k0/$k1 without
        # saving their old values.
        move $k1 $at   # Save $at
        .set at
        sw $v0 s1      # Not re-entrent and we can't trust $sp
        sw $a0 s2
        mfc0 $k0 $13   # Ler Registo Cause e guardá-lo em $k0

        sgt $v0 $k0 0x44       # Se for uma interrupção externa, salta para a
                               # rotina de tratamento!
        bgtz $v0 ext

        addu $0 $0 0
        li $v0 4       # syscall 4 (print_str)
        la $a0 __m1_
        syscall
```

```
        li $v0 1       # syscall 1 (print_int)
        srl $a0 $k0 2  # shift Cause reg
        syscall
        li $v0 4       # syscall 4 (print_str)
        lw $a0 __excp($k0)
        syscall
        bne $k0 0x18 ok_pc # Bad PC requires special checks
        mfc0 $a0, $14  # EPC
        and $a0, $a0, 0x3 # Is EPC word-aligned?
        beq $a0, 0, ok_pc
        li $v0 10      # Exit on really bad PC (out of text)
        syscall

ok_pc:
        li $v0 4       # syscall 4 (print_str)
        la $a0 __m2_
        syscall
        mtc0 $0, $13   # Clear Cause register
        bnez $v0, ret

#########################################################################
# Rotina (nossa) de serviço à interrupção:
#########################################################################
ext:
        li $v0 4
        la $a0 inter
        syscall

        li $v0 1
        lb $a0 in
        syscall

        sb $a0 lido($0)

        li $v0 1
        lb $a0 lido($0)
        syscall

        li $v0,105 # tecla i

        beq $a0,$v0,ok
        bne $a0,$v0,continua
ok:
        li $v0 4
        la $a0 caracter
        syscall
continua:
        mtc0 $0, $13   # Limpar o registo Cause

# Retorno do tratamento de uma excepção para a instrução interrompida:
ret:    lw $v0 s1
        lw $a0 s2
        mfc0 $k0 $14   # EPC
        .set noat
        move $at $k1   # Restore $at
```

```
        .set at
        rfe             # Return from exception handler
        addiu $k0 $k0 4 # Return to next instruction
        jr $k0

# Standard startup code.  Invoke the routine main with no arguments.
        .text
        .globl __start
__start:
        lw $a0, 0($sp) # argc
        addiu $a1, $sp, 4 # argv
        addiu $a2, $a1, 4 # envp
        sll $v0, $a0, 2
        addu $a2, $a2, $v0
        jal main
        li $v0 10
        syscall         # syscall 10 (exit)

# código de inicialização necessário para que a entrada de dados
# seja gerida por interrupções:
        .globl main
main:   addi $sp,$sp,-4
        # activação das interrupções do teclado
        lw $t0,cin($0)
        ori $t0,$t0,2
        sw $t0,cin($0)
        # activação das interrupções do nível zero
        mfc0 $t0,$12
        lw $t1,perm($0)
        or $t0,$t0,$t1
        mtc0 $t0,$12

haz:
        addi $t1,$0,10000
wat:
        addi $t1,$t1,-1
        bnez $t1,wat
        addi $s0,$s0,1
        bnez $s0,haz

        lw $ra,0($sp)
        addi $sp,$sp,4
        jr $ra
```

O programa anterior inclui, por um lado, o código necessário para que a entrada de dados se possa gerir perante interrupções e, por outro lado, as acções que se devem executar para cada uma das excepções que se produzem no sistema.

Compare este ficheiro com o ficheiro `trap.handler` original.

Escreva o ficheiro anterior a partir do ficheiro que contém o código standard para tratamento das excepções (trap.handler). Carregue-o e execute-o no simulador SPIM com as opções "Mapped I/O" e sem a opção "Load Trap File".

Q7.1. O que faz este programa? O que ocorre quando se carrega numa tecla?

Q7.2. Modifique o programa anterior para imprimir o caracter teclado.

Q7.3. Um *exception handler* necessita de salvar todos os registos que possa modificar. Isto poderá ser feito no *stack*, usando o `$sp`? Explique.

Q7.4. O *trap handler* imprime um mensagem de erro para cada excepção usando o código da excepção (que se encontra no registo Cause) como índice de uma tabela que contém os endereços das mensagens.

Identifique essa tabela e mostre algumas das suas entradas, preenchendo a tabela que se segue. Na coluna "Etiqueta", a primeira linha será a etiqueta onde a tabela começa. Na coluna "Comentário" indique qual o conteúdo armazenado em cada um dos endereços da coluna "Endereços".

Etiqueta	Endereço	Conteúdo	Comentário

Q.7.5 (5º Trab. Lab. 2003-2004) Escreva um programa em Assembly do MIPS R2000 que imprime "cima", "baixo", "esquerda" e "direita" quando se carrega nas teclas `i`,`k`,`j` e `l`, respectivamente. O programa tem de gerir as E/S por interrupções, pelo que deverá substituir o ficheiro `trap.handler` original por um modificado (contendo a rotina de tratamento de uma interrupção vinda do teclado). A impressão das mensagens "cima", etc… pode, contudo, ser realizada recorrendo a um `syscall`.

Este programa deverá servir como ponto de partida para a implementação do modo manual do projecto.

5º Trabalho de Laboratório

8
Introdução à Micro-Programação

Calvin: Queres ver uma coisa estranha? Repara, colocas pão nesta ranhura e carregas no botão para baixo. Esperas alguns minutos, e zás! Sai uma tosta!
Hobbes: Uau! Para onde foi o pão?
Calvin: Não faço a mínima ideia. Não é esquisito?

Calvin, em "Calvin & Hobbes" de Bill Watterson

Tal como o Calvin, temos estado a programar (até agora) em Assembly, sem saber o que se passa no interior da nossa "torradeira" que é o MIPS. Sabemos que o Assembly é uma linguagem de programação de baixo nível e que constitui uma representação simbólica da codificação binária de um computador: a linguagem máquina (ver Capítulo 1).

Ora, a linguagem máquina é composta por micro-instruções que indicam que operação digital deve o computador fazer. Cada instrução máquina é composta por um conjunto ordenado de zeros e uns, estruturado em campos. Cada campo contém a informação que se complementa para indicar ao processador que acção realizar.

Vamos então neste capítulo observar alguns exemplos dos programas em linguagem máquina correspondentes a alguns dos troços de programas Assembly que temos vindo a estudar. E ficaremos a perceber o que se passa "dentro" do processador.

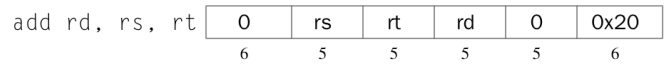
Formatos das instruções MIPS em linguagem máquina

Qualquer instrução MIPS codifica-se em linguagem máquina, isto é, zeros e uns. Uma instrução MIPS ocupa, como sabemos, 4 palavras. Isto significa que qualquer instrução MIPS será codificada usando 32 bits (mesmo que não necessite de todos esses bits).

Existem, como vimos, duas grandes classes de instruções Assembly MIPS: instruções implementadas em hardware (dizem-se *hard-wired*) e as pseudo-instruções implementadas pelo Assembler. Uma pseudo-instrução expande-se em mais de uma instrução *hard-wired*.

Uma instrução é codificada em vários campos (grupos de bits contíguos), cada um deles com um significado próprio. Por exemplo, a seguinte figura (retirada do manual do SPIM), ilustra a codificação da instrução add (com *overflow*):

Addition (with overflow)



Na adição com overflow, a instrução é codificada em seis campos. Cada campo tem uma largura (em termos de número de bits) indicada sob o respectivo campo. Esta instrução começa com um grupo a zeros (6 bits a zero, à esquerda). Este grupo de bits é o *opcode* (código da instrução).

Os registos são especificados por rs, rt e rd. Portanto o campo seguinte é especifica um registo chamado rs (de *register source*). Este registo é o segundo operando da instrução. Outro campo comum é designado imm16, que especifica um número imediato de 16

bits. De um modo geral, as instruções seguem o formato especificado na tabela seguinte:

Nome do campo	Cód. Instrução	2º Operando	3º Operando	1º Operando	shamt	Funct
Largura do Campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R	Cód. da Inst. **6 bits**	rs **5 bits**	rt **5 bits**	rd **5 bits**	Shamt **5 bits**	Funct **6 bits**
I	Cód. da Inst. **6 bits**	rs **5 bits**	rt **5 bits**	Endereço Imediato **16 bits**		
J	Cód. da Inst. **6 bits**	Endereço **26 bits**				

A segunda tabela mostra que existem 3 tipos de formatos para as instruções: R, I e J, consoante o primeiro operando seja um registo (formato R), um imediato (formato I) ou a instrução seja uma instrução de salto (formato J), a qual só contém como argumento um endereço.

Q8.1. Partindo do seguinte programa simbólico escrito em Assembly do MIPS R2000, pretendemos obter o programa respectivo em linguagem máquina:

```
add $1, $2, $3          # Equivale a $1 # =$2+$3
sub $1, $2, $3          # Equivale a $1 # =$2-$3
lw $1, 100($2)          # Equivale a $1= Memória ($2+100)
sw $1, 100($2)          # Memória ($2+100) = $1
```

A primeira tabela apresenta o formato da instrução em causa, o seu código, os operandos e shamt/Funct. A segunda tabela é equivalente à anterior, mas mostra a codificação binária da instrução, ou seja, a instrução tal e qual como o computador a vê. E mostra também a instrução em hexadecimal (a qual também surge no SPIM).

	Cód. Instrução	1ºoperando	2ºoperando	3ºoperando	shamt	funct	Instrução MIPS
Formato R	0	2	3	1	0	32	add $1,$2,$3
Formato R	0	2	3	1	0	34	sub $1,$2,$3
Formato I	35	2	1	100			lw $1,100($2)
Formato I	43	2	1	100			sw $1,100($2)

	Cód. Instrução	2º Operando	3º Operando	1º Operando	shamt	Funct	
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
				ou 16 bits			em Hexadecimal:
em Binário:	0000 00	00 010	0 0011	0000 1	000 00	10 0000	0x00430820
	0000 00	00 010	0 0011	0000 1	000 00	10 0010	0x00430822
	1000 11	00 010	0 0001	0000 0000 0011 0100			0x8c410034
	1010 11	00 010	0 0001	0000 0000 0011 0100			0xac410034

Q8.2. Considere o seguinte código escrito em linguagem Assembly do processador MIPS:

```
la $t0,palavra # palavra corresponde ao endereço 0x10010005 do seg. dados
lw $a0,0($t0)
jal randnum    # randnum reside no endereço 0x00400060 do seg. código
add $t2,$v0,$0
```

Sabendo que a primeira instrução tem como endereço `0x400028`, traduza este troço de código para linguagem máquina MIPS e escreva o conteúdo de cada instrução em binário e hexadecimal numa tabela com o seguinte aspecto:

Endereços	Cód. Instrução	2º Operando	3º Operando	1º Operando	shamt	Funct	
0x00400028							lui $8, 0x1001

Endereços	Cód. Instrução	2º Operando	3º Operando	1º Operando	shamt	Funct	
Largura do Campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	em Hexadecimal:
0x00400028							0x ...

Resolução:

(há que ter em atenção que a instrução `la` é uma instrução que se expande em duas – neste caso, uma vez que o endereço de `palavra` tem de ser calculado).

Endereços	Cód. Instrução	rs	rt	rd	shamt	Funct	
0x00400028	15	0	8	0x1001			lui $8,0x1001
0x0040002c	8	8	8	5			addi $8,$0,5
0x00400030	35	8	4	0			lw $4,0($8)
0x00400034	3	0x400060					jal 0x400060
0x00400038	0	2	0	10	0	32	add $10,$2,$0

Endereços	Cód. Instrução	2º Operando	3º Operando	1º Operando	shamt	Funct	
Largura do Campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	em Hexadecimal:
0x00400028	0011 11	00 000	0 1000	0001 0000 0000 0001			0x3c081001
0x0040002c	0010 00	01 000	0 1000	0000 0000 0000 0101			0x21080005
0x00400030	1000 11	01 000	0 0100	0000 0000 0000 0000			0x8d040000
0x00400034	0000 11	00 0001 0000 0000 0000 0001 1000					0x0c100018
0x00400038	0000 00	00 010	0 0000	0101 0	000 00	10 0000	0x00405020

Q8.3. Pretende-se o programa em Assembly do MIPS para a seguinte instrução em pseudo-código: `A[12] = h + A[8]`

Assumindo que A é uma tabela e associando a variável h ao registo `$s1`, e sabendo ainda que o endereço base de `A` encontra-se em `$s3`:

```
main:   lw $t0,32($s3)        # carrega A[8] em $t0
        add $t0,$s1,$t0       # soma $t0=h+A[8]
        sw $t0,48($s3)        # armazena $t0 em A[12]
```

A tabela seguinte mostra este programa em linguagem máquina:

Endereços	Cód. Instrução	2º Operando	3º Operando	1º Operando	shamt	Funct	
0x00400000	35	19	8	8*4 = 32			lw $t0,32($s3)
0x00400004	0	17	8	8	0	32	add $t0,$s1,$t0
0x00400008	43	19	8	12*4 = 48			sw $t0,48($s3)

E esta tabela mostra o programa em hexadecimal:

Endereços	Cód. Instrução	2º Operando	3º Operando	1º Operando	shamt	Funct	
Largura do Campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	em Hexadecimal:
0x00400000	1000 11	10 011	0 1000	0000 0000 0010 0000			0x8e680020
0x00400004	0000 00	10 001	0 1000	0100 0	000 00	10 0000	0x02284020
0x00400008	1010 11	10 011	0 1000	0000 0000 0011 0000			0xae680030

Q8.4. Considere o seguinte segmento de programa em linguagem C:

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

Escreva o respectivo troço de programa em linguagem Assembly do MIPS. Em seguida, preencha as tabelas com os formatos apresentados (programa em linguagem máquina e em hexadecimal). Considere que tem as variáveis h, i e j nos registos $16, $17 e $18, em que se $17=$18 somam-se; se diferentes, subtraem-se os conteúdos dos registos $17 e $18.

Resolução:

O programa Assembly pedido é muito simples:

```
main:   bne $17, $18, L1
        add $16, $17, $18
        j Exit
L1:     sub $16, $17, $18
Exit:   jr $ra
```

O programa em linguagem máquina é o seguinte:

Endereços	Cód. Instrução	2º op.	3º op.	1º op.	shamt	Funct	
0x00400000	5	17	18	3*4=12			bne $17,$18,L1
0x00400004	0	17	18	16	0	32	add $16,$17,$18
0x00400008	3	0x400010					j Exit
0x0040000c	0	17	18	16	0	34	sub $16,$17,$18
0x00400010	0	31	0	0	0	8	Exit: jr $ra

Questão: qual o formato da instrução `bne`? Porque é que o último campo tem o valor 12 (= 3*4)?

Exercícios Resolvidos

Exame de Época Normal, 2003/2004, Parte Prática:

1.(0,5 val.) Considere o seguinte troço de programa em linguagem Assembly do MIPS:

```
                .data 0x10010000        # segmento de dados
palavra1:       .word 13
palavra2:       .word 0x15
```

Indique, em hexadecimal, quais os valores das seguintes etiquetas:

`palavra1:` `0x10010000`

`palavra2:` `0x10010004`

Nota: também foram aceites as respostas que se referiram ao valor armazenado nas posições de memória (13 e 0x15), ou os seus equivalentes em hexadecimal/decimal, embora a resposta correcta seja a que consta desta resolução.

2.(0,5 val.) Considere o seguinte programa em Assembly do MIPS:

```
        .data
var1:   .word 30
var2:   .word -50
res:    .space 1
        .text
main:   lw $t8,var1($0)
        lw $t9,var2($0)
        and $t1,$t1,$0
        and $t0,$t0,$0
        beq $t8,$0,haz
        ori $t0,$0,1
haz:    slt $t1,$t9,$t8
wat:    and $t0,$t0,$t1
        sb $t0,res($0)
```

Qual o valor armazenado na posição de memória `res`?

1.

3. (0,2+0,3+0,5 val.) Considere o seguinte troço de programa em linguagem Assembly do MIPS:

```
        .data 0x10010000
byte1:  .byte 0x10
espaco: .space 4
byte2:  .byte 0x20
pal:    .word 10
```

3.(a) Quantos bytes foram reservados para a variável espaco?

2 bytes.

3.(b) A partir de que endereços se inicializaram:

`byte1:` `0x10010000`

`byte2: 0x10010003`

3.(c) O que acontece se o processador executar a instrução `lw $t0,pal`? Como resolver essa situação?

Dá-se um acesso desalinhado à memória, visto que se pretende carregar uma palavra (4 bytes) a partir de um endereço que não é múltiplo de 4. Uma forma possível é introduzir a directiva .align 2 no início da zona de dados.

Nota: todos os alunos tiveram a cotação total (0,5 val.) nesta pergunta, devido à gralha .space 2 do exame (deveria estar .space 4, como nesta resolução).

4. (1 val.) Complete o seguinte código: escreva um procedimento em Assembly do MIPS que percorra o vector de bytes `sala` e conte o número de bytes cujo valor é igual a 1. Armazene o resultado da contagem na variável `lixos`. (Pode assumir que a dimensão do vector `sala` é sempre 64 bytes).

```
        .data           # declara segmento de dados
sala:   .space 64       # variável que representa o conteúdo de uma sala
lixos:  .word 0         # variável que conta o número de lixos
#
# Princípio do Segmento de Texto
#
        .text   # declara segmento de código
main:
        li $t0,0        # auxiliar (lê byte i)
        li $t1,0        # conta número de lixos
        li $t2,0        # variável i do ciclo
ciclo:
        lb $t0,sala($t2)
        bne $t0,1,sala($t2)
        addi $t1,$t1,1
continua:
        addi $t2,$t2,1
        beq $t2,64,fim
        j ciclo
fim:    sb $t1,lixos
```

5. (1 val.) Considere que já se encontra implementado o procedimento `randnum`, o qual devolve um inteiro aleatório entre 0 e um dado número limite:

- argumentos: número máximo aleatório pretendido (em `$a0`)
- resultados: inteiro aleatório entre 0 e argumento `$a0` menos 1 !

Escreva um procedimento `Joga`, em Assembly, que recebe como argumento um inteiro `Aposta`. O procedimento chama o `randnum` com o argumento `10`. Caso o resultado seja igual ao argumento `Aposta`, o procedimento imprime na consola a string "Certo!". Caso contrário, imprime "Errado!".

```
joga:
        addi $sp,$sp,-4
        sw $ra,4($sp)
        move $t0,$a0   # t0 contém o argumento aposta
        li $a0,10
        addi $sp,$sp,-4
        sw $ra,4($sp)
        jal randnum
```

```
        lw $ra,4($sp)
        addi $sp,$sp,4
        beq $v0,$t0,certo      # resultado em v0
errado:
        la $a0,string_errado
        li $v0,4
        syscall
        j fim
certo:
        la $a0,string_certo
        li $v0,4
        syscall
fim:
        lw $ra,4($sp)
        addi $sp,$sp,4
        jr $ra
```

6. (1+1 val.) Pretende-se codificar um procedimento Substitui(string,x,y), em Assembly do MIPS, que dada uma string e dois caracteres, x e y, substitui nessa string todas as ocorrências do caracter x pelo caracter y.

O procedimento terá como argumentos:

- o endereço da string
- o caracter a procurar
- o caracter para a substituição
- o número de caracteres da string

Por exemplo, para a string "Sobstitoi", se o caracter a procurar for o caracter 'o', e o caracter para a substituição for o caracter 'u', a string deve ser alterada para "Substitui".

6.(a) Desenhe um fluxograma para este procedimento.

6.(b) Escreva o programa em Assembly do MIPS.

```
substitui:
        addi $sp,$sp,-4
        sw $ra,4($sp)
        li $t1,0
ciclo:
        lb $t0,($a0)
        bne $t0,$a1,continua
sub:
        sb $a2,($a0)
continua:
        addi $a0,$a0,1
        addi $t0,$t0,1
        bne $t1,$a3,ciclo
        # senão, termina
        lw $ra,4($sp)
        addi $sp,$sp,4
        jr $ra
```

7. (1 val.) Descreva o que acontece quando um procedimento é chamado usando jal.

Quando um procedimento é chamado usando `jal`, duas coisas acontecem:

- O controlo é transferido para o endereço fornecido pela instrução.
- O endereço de retorno é guardado no registo `$ra`.

8. (1 val.) Em que consistem as excepções assíncronas? Será um overflow aritmético uma excepção assíncrona?

As excepções assíncronas são as que ocorrem sem qualquer relação temporal com o programa que é executado. Pedidos de E/S, erros de memória, falhas de fornecimento de energia etc. são exemplos de excepções assíncronas. Um overflow aritmético não é uma excepção assíncrona, visto que ocorre no mesmo sítio de cada vez que um programa é executado com os mesmos dados e com a mesma alocação de memória.

Exame de Recurso, 2003/2004, Parte Prática:

1.(1 + 0,5 val.)

1.(a) Crie um programa que defina um vector de 5 palavras associado à etiqueta `vector` que comece no endereço `0x10000000` e que contenha os valores 10,11,12,13,14.

```
        .data 0x10000000
vector: .word 10,11,12,13,14
```

1.(b) O que acontece se quisermos que o vector comece no endereço `0x10000002`? Em que endereço começa realmente? Porquê?

Começa no endereço 0x10000004, por ser uma palavra de memória (4 bytes).

2. (1 val.) Considere o seguinte programa:

```
        li $t1,0
ciclo:  lb $t5,string($t1)
        beq $t5,$0,fim
        addi $t1,$t1,1
        j ciclo

fim:    li $v0,1
        move $a0,$t1
        syscall                 # chamada sistema print_int
        addi $v0,$0,10          # $v0=10 (para retornar o controlo ao SPIM)
        syscall
```

Para a cadeia de caracteres "`AC-Uma`", declarada na memória como `.asciiz`, que número aparece na consola quando este programa é executado? O que faz este programa?

Conta o número de caracteres numa dada string. Para a string "AC-Uma", aparece na consola 6.

3. (1 val.) Suponha que num determinado protocolo de comunicação de dados, é necessário introduzir caracteres de controlo ('#','/','%').

Por exemplo, se o texto a transmitir é:

```
Calvin: A realidade continua a arruinar a minha vida.
```

no destino chega:

```
Ca%lvin:# A re/alid/ade continu/a a arru#inar a minh/a vi%da.
```

Pretende-se desenvolver uma rotina que no computador destino suprima estes caracteres do texto, de forma a obter de novo o texto original. Parta do seguinte cabeçalho:

```
        .data 0xffff8888       # declara zona de memória partilhada
mensagem:
        .space 128
        .data 0x10010000       # declara segmento de dados
resultado:
#
# Princípio do Segmento de Texto
#
        .text   # declara segmento de código
main:
```

e considere que a string `mensagem` (string recebida no computador destino) termina com o caracter 0x0, e que os valores ascii dos caracteres de controlo são 0x12,0xff e 0x63 (para efeitos de comparação).

Guarde a string resultante em `resultado`.

4. (1+1 val.) Considere um vector de números em posições consecutivas de memória e cujo fim é indicado pelo valor 0. Pretende-se desenvolver uma rotina que incremente em uma unidade cada um dos números no vector. Por exemplo:

vector inicial = [1 2 3 4 5 0]

resultado = [2 3 4 5 6 0]

Considere que o endereço do vector inicial é passado para a rotina pelo registo `$a0` e que o resultado deve ficar num endereço de memória que é passado para a rotina pelo registo `$a1`.

a) Desenhe um fluxograma que realize a função pretendida.

b) Escreva a rotina em linguagem assembly do processador MIPS.

5. (1 + 0,5 val.) Considere a seguinte rotina de tratamento a um interrupção:

```
#############################################################################
# Rotina (nossa) de serviço à interrupção:
#############################################################################
ext:    li $v0 4
        la $a0 inter
        syscall

        li $v0 1
        lb $a0 in
        syscall

        sb $a0 lido($0)
continua:
        mtc0 $0, $13   # Limpar o registo Cause
```

5. a) O que acontece quando ocorre uma interrupção vinda do teclado? Uma rotina de serviço à interrupção deve ser curta? Porquê?

5. b) Considere que a etiqueta in está associada ao porto de leitura do teclado. O que faz a instrução lb $a0 in?

6. (1 val.) Qual a diferença entre o código de tratamento de uma excepção e o código de uma rotina? Uma excepção pode retornar valores? Porquê?

Enunciados de Projectos

Ano Lectivo 2003/2004 – 1º Semestre

Enunciado do Projecto:
Um Controlador para o Robot Shakeaspira

Introdução

Você e o seu colega trabalham como consultores júniores na empresa de consultadoria B., Anha & D'Akobra[2], uma empresa especializada no desenvolvimento de sistemas embebidos em tempo real (e.g. controladores industriais) para os quais é frequente programar em Assembly.

Foram ambos destacados pelo gestor de projecto para o desenvolvimento de um controlador para um robot chamado Shakeaspira. O Shakeaspira é um robot aspirador automático para a casa: movimenta-se automaticamente num quarto, evitando objectos e quaisquer obstáculos (nomeadamente as paredes) ao mesmo tempo que limpa o pó do quarto.

Figura 1: Robot aspirador para a casa com movimentação automática (modelo experimental[3]). No topo esquerdo, o robot encontra-se na estação de recarga da bateria.

O Shakeaspira possui incorporada uma bateria, que torna o robot autónomo, sem precisar de manutenção. A bateria vai descarregando ao longo do tempo, e quando atinge um determinado valor limite, o Shakeaspira dirige-se para uma localização – pré-definida – onde existe o carregador da sua bateria.

O problema é que o prazo para o desenvolvimento é curto, uma vez que a concorrência – uma empresa japonesa – já está em vias de comercializar um robot semelhante, actualmente em fase experimental (ver Figura 1). Por isso você terá de ser eficaz, criativo e inovador.

---

[2] Nome fictício, obviamente.

[3] Para mais informações sobre o robot (real) da "concorrência", visite:
`http://www.i4u.com/japanreleases/hitachirobot.htm`

Descrição do Controlador a Implementar

Por motivos de custo, o controlador será simulado em software, usando o simulador SPIM, que simula um processador MIPS R2000, a fim de testar as suas funcionalidades sem custos adicionais de hardware.

A Figura 2 ilustra o quarto onde o robot aspira. O quarto é discretizado numa grelha. Em cada posição da grelha só pode existir ou um objecto ou lixo.

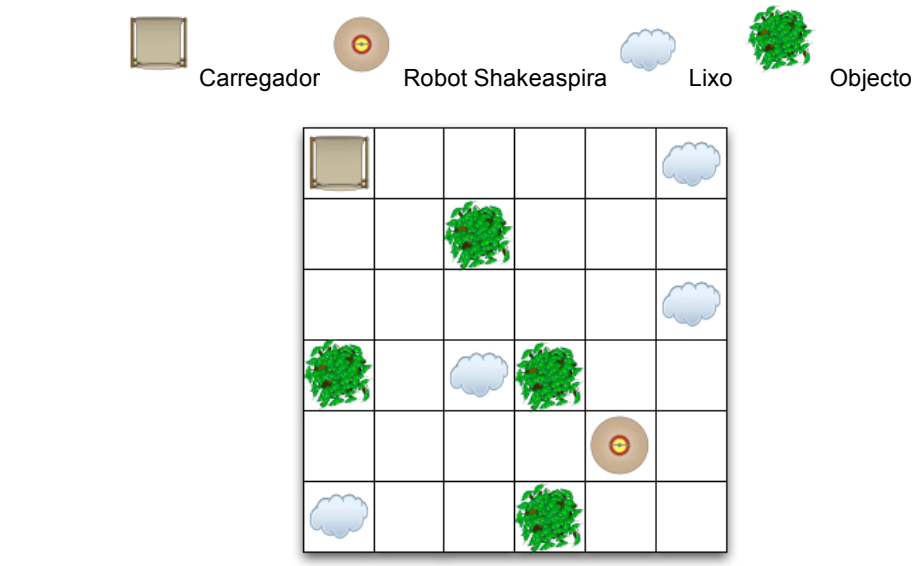


Figura 2: Ilustração de um exemplo de quarto do Shakeaspira (6×6).

No início da execução do programa, pede-se ao utilizador que introduza os seguintes parâmetros:

- Dimensão do quarto (4×4, …, 8×8)
- Quantidade de objectos (mínimo 2, máximo 6)
- Quantidade de lixo (mínimo 2, máximo 6)
- Nível inicial da bateria (mínimo 10, máximo 5000)

Não é necessário verificar se os valores introduzidos estão dentro dos limites estipulados. Os objectos e os lixos são colocados em posições aleatórias. A localização da bateria do robot é fixa: posição cuja coordenada é a origem da grelha: (0,0), pelo que nesta posição não pode ser colocado objecto ou lixo algum. A localização inicial do robot é a posição (1,0), à direita do carregador.

O robot tem dois modos de funcionamento: o **modo automático** e o **modo manual**. No **modo automático**, após a definição dos parâmetros, o robot avança à procura de lixo para recolher. O robot tem de percorrer todas as posições: se o conteúdo da posição onde se encontra possuir lixo, recolhe-o e incrementa uma variável inteira que conta o número de lixos recolhidos. Se, por outro lado, o conteúdo da posição onde se encontra possuir um objecto, o robot não o recolhe.

Sempre que o robot avança, a bateria (representada como uma variável inteira na memória) é decrementada uma unidade. Quando o valor da bateria atinge um determinado nível mínimo, o robot imprime a mensagem "Bateria Baixa!", e tem de retornar à posição base da bateria (0,0) onde recarrega a sua bateria para o valor do nível máximo (500). Cabe a si determinar um valor plausível para o nível mínimo. O robot termina a sua tarefa automática quando todo o lixo tiver sido recolhido.

O exemplo seguinte ilustra a forma como o programa deverá funcionar no **modo automático**:

```
Shakeaspira> Introduza o tamanho do quarto (4x4...8x8): 6
Shakeaspira> Introduza quantidade de objectos (2...6): 4
Shakeaspira> Introduza quantidade de lixo (2...6): 4
Shakeaspira> Introduza a bateria inicial (10...500): 100
```

Após a introdução dos parâmetros, o programa pede uma confirmação:

```
Shakeaspira> Tamanho=6, Objectos=4, Lixo=4, Bateria=100, confirma? (s=1/n=0): 1
```

No caso de o utilizador teclar `0`, o programa volta ao ciclo em que pergunta os parâmetros. Quando o utilizador teclar `2`, o programa termina. Caso o utilizador confirme, o programa mostra a sequência de acções do robot:

```
 0 1 2 3 4 5
0C O
1    R   L
2
3  O     L O
4      O
5    L   L
Shakeaspira> Lixo recolhido=0 Bateria=99
```

Respeite este formato de apresentação do conteúdo do quarto. `C` representa o carregador da bateria, `R` o robot, `O` um objecto e `L` um lixo. O programa mostra sempre quanto lixo já recolheu, assim como o nível da bateria. O programa fica à espera que o utilizador pressione a tecla `Enter`, para avançar e mostrar o estado resultante da acção seguinte[4].

```
 0 1 2 3 4 5
0C O
1      R L
2
3  O     L O
4      O
5    L   L
Shakeaspira> Lixo recolhido=0 Bateria=98

 0 1 2 3 4 5
0C O
1        R
2
3  O     L O
4      O
5    L   L
Shakeaspira> Lixo recolhido=1 Bateria=97
```

[4] Sempre que o utilizador pressionar a tecla `2`, seguida de `Enter`, o programa termina.

Neste exemplo, o robot recolheu 1 lixo na posição (4,1). O lixo (`L`) dessa posição desaparece. O programa termina quando o lixo recolhido for igual ao lixo inicialmente introduzido. O robot não pode executar saltos! Isto é, só avança entre posições contíguas, em cada uma das direcções norte,sul,este,oeste.

No **modo manual**, o funcionamento é semelhante, excepto no facto de o robot ficar parado à espera que o utilizador pressione uma das teclas direccionais já referidas.

Neste modo, o Shakeaspira é operado pelo teclado, usando as teclas `i`,`j`,`k`,`l` para especificar a direcção do seu movimento (`i`=norte, `k`=sul, `j`=este, `l`=oeste).

Para implementar este modo não pode usar a chamada de sistema `syscall`, já que esta pede sempre um `Enter` e mostra no écran a tecla digitada. Em vez disso, deve criar um novo ficheiro a partir do `trap.handler` onde associa uma rotina de tratamento das interrupções do teclado a uma rotina que move o robot numa dada direcção. Desta maneira, pode-se teclar à vontade que apenas surge na consola a "imagem" actualizada do quarto do robot.

Não é necessário, neste modo, introduzir um valor de bateria, visto que esta não é utilizada. Também não é necessário verificar se estamos a mover o robot contra uma das paredes (essa funcionalidade dá direito, contudo, a crédito extra).

Funcionalidades já implementadas

O seu gestor de projecto possui dois ficheiros que serão, provavelmente, do seu interesse, pelo que deve contactá-lo para obtê-los: um deles é um gerador de números aleatórios, que servirá para testar o Shakeaspira num quarto dispondo o lixo e os obstáculos aleatoriamente (a localização da bateria é fixa e pode assumir que o robot a conhece).

O outro é uma rotina de tratamento de interrupções adaptada da rotina original do núcleo do Sistema Operativo simulado pelo SPIM. Servirá como ponto de partida para codificar o modo de operação manual do Shakeaspira. Lembre-se que pode (e deve) reaproveitar o código do modo automático para implementar o modo manual.

Objectivos mínimos e Créditos Extra

Para obter aprovação no Projecto, é necessário que este funcione correctamente para a gama especificada de valores dos parâmetros, tanto no modo manual como no modo automático.

Existe um conjunto de funcionalidades que, a serem correctamente implementadas, conduzem a créditos extra na nota final:

- Shakeaspira que evita obstáculos: + 2 valores em relação à nota base;
- Shakeaspira em modo aleatório: + 1 valor em relação à nota base; o modo aleatório é um modo em que o robot escolhe em cada passo uma direcção aleatória, mantendo o restante comportamento já definido;
- Programa que permite colocar o Shakeaspira numa posição inicial aleatória, mantendo o restante comportamento: + 1 valor em relação à nota base.
- Alerta: no modo manual, imprime um aviso quando se move o Shakeaspira contra uma das paredes: + 1 valor.

Prazos e Critérios de Avaliação

O projecto seguirá uma entrega por fases. A nota só é dada após a discussão. No entanto é obrigatório apresentar os artefactos designados no prazo respectivo. Por cada dia de atraso na entrega, sofre uma penalização de 1 valor na nota final.

As fases e artefactos a entregar são os seguintes:

1. Fluxograma de alto nível
2. Programa a funcionar no modo automático com os objectivos mínimos
3. Versão Final do Programa + Relatório

Os **prazos** para cada fase são:

1. Dia 25 de Novembro de 2003.
2. Dia 19 de Dezembro de 2003.
3. Dia 16 de Janeiro de 2003[5].

O formato e local de entrega serão oportunamente divulgados.

Os seguintes aspectos serão avaliados:

- Fluxograma e Relatório;
- Cumprimento dos requisitos (correcto funcionamento do programa, em modo automático e em modo manual);
- Qualidade do código (e.g.: Os comentários são úteis? As etiquetas fazem sentido? Segue a convenção de utilização dos registos e procedimentos do MIPS? O código encontra-se bem estruturado?);
- Discussão do Projecto (esta componente da nota é individual e obrigatória);
- Serão valorizadas soluções que se diferenciem pela inovação, criatividade e desempenho em relação às restantes.

Projectos iguais, ou muito semelhantes, serão classificados com 0 (zero) valores.

Consulte regularmente a página Web dos laboratórios pois serão afixadas as respostas às perguntas mais frequentes dos alunos.

[5] Ainda por confirmar.