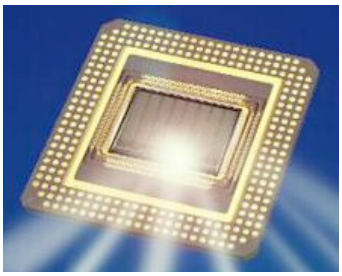

Arquitectura de Computadores
2004 / 2005

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`		