Assembly para o Assemblador da GNU
Arquitectura Intel IA-32
Versão 0.4

Filipe Machado de Araújo

Outubro de 2005

Assembly para o Assemblador da GNU
Arquitectura Intel IA-32
Versão 0.4

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Filipe Machado de Araújo

Lisboa

Outubro de 2005

Agradecimentos

Agradeço ao Professor Vasco Vasconcelos pelas importantes indicações que tornaram o texto mais rigoroso.

Agradeço também ao Dr. Alexandre Pinto e à Dr.ª Teresa Chambel que ajudaram na revisão de alguns capítulos e contribuíram com algumas sugestões.

As gralhas e os erros que existirem são todos da minha responsabilidade e não da deles.

Lisboa, Outubro de 2005

Filipe Machado de Araújo

Índice

Lista de Figuras

Lista de Tabelas

1

Assembly

1.1 Introdução

De todos os elementos que compõem um computador o mais importante é, sem dúvida, a unidade central de processamento (CPU — *Central Processing Unit*), também designada simplesmente por processador.

O processador é um complexo circuito electrónico digital integrado, capaz de executar sequências de instruções. O conjunto destas instruções e respectivas regras de utilização compõem a linguagem que o processador compreende e é capaz de executar, que é designada por linguagem ou código máquina.

A programação directa em linguagem máquina é extremamente penosa e demorada para um ser humano, porque as instruções e os operandos são representados como números inteiros, o que significa que um programa nesta linguagem é uma lista, eventualmente longa, de números inteiros. Para resolver este problema existe uma linguagem alternativa com as mesmas instruções, mas representadas por mnemónicas em vez de números inteiros. O mesmo se passa com os operandos que, a menos que sejam precisamente números, são designados por mnemónicas. No caso das operações mais complexas as mnemónicas que representam operandos podem ser combinadas. No entanto, o resultado será sempre mais perceptível do que aquele que seria possível com uma simples representação numérica.

Note-se que, normalmente, haverá uma tradução directa de todas as instruções assembly em instruções do processador. Por esta proximidade que existe entre a linguagem máquina e a linguagem assembly esta última é categorizada como linguagem de baixo nível. Esta categoria está em oposição à das linguagens de alto nível, onde se incluem o Pascal, C, Java, etc. Nestas, não tem que existir qualquer correspondência directa entre as respectivas instruções e operandos e as instruções e operandos do processador. Nem seria, aliás, desejável que essa correspondência existisse.

É de realçar ainda, que não existe apenas uma única linguagem assembly possível, mas muitas diferentes. Se este facto é mais ou menos evidente no caso em que se consideram processadores diferentes [1], que têm instruções diferentes, é menos evidente quando o processador em causa é o mesmo. Mas, na verdade, para o mesmo processador podem existir várias linguagens assembly, por motivos que se relacionam com a existência de sistemas operativos e fabricantes de *software* diferentes que produzem assembladores incompatíveis entre si.

As linguagens assembly diferem entre si essencialmente na sintaxe das operações [2] porque, geralmente, as mesmas instruções usam mnemónicas idênticas ou, pelo menos, parecidas. Os operandos que são representados por uma certa ordem podem aparecer trocados noutro assemblador, por exemplo. Um caso concreto que ilustra este facto é o MASM — Microsoft Assembler — que usa a sintaxe Intel, em que os operandos aparecem pela ordem inversa relativamente à sintaxe AT&T usada pelo Gas.

[1] Ainda assim há assembladores, como sejam o Gas — Gnu Assembler, de que falaremos adiante — que produzem código para processadores diferentes e que concretizam alguma independência em relação à plataforma (microprocessador + sistema operativo).

[2] Outro aspecto em que não propriamente as linguagens, mas os assembladores que as interpretam diferem consideravelmente é nas directivas, que aparecem muitas vezes misturadas com o código na linguagem assembly.

De qualquer forma, para um mesmo processador todas as instruções duma linguagem assembly encontram correspondência com as instruções das outras linguagens assembly.

Neste texto vai ser considerada a utilização da linguagem assembly para o assemblador Gas, para a arquitectura de 32 *bits* da Intel, designada de IA-32 (*Intel Architecture*), que se iniciou com o processador 80386 [3]. O sistema operativo considerado, sempre que este aspecto for relevante é o Unix e mais particularmente o Linux.

1.2 Pressupostos

Quando um programador decide utilizar assembly tem que possuir um conjunto prévio de conhecimentos. Em primeiro lugar, tem que ter algum conhecimento da arquitectura da máquina. Este conhecimento inclui as instruções existentes, os registos e a organização de memória.

A organização da memória, por exemplo, abrange a questão de saber se a memória é linear ou segmentada e que tamanho têm os segmentos. Estes aspectos são relevantes porque, nalgumas plataformas, é necessário definir os segmentos dum programa de tal forma que o código seja arrumado num segmento, os dados noutro(s) e a pilha ainda noutro. Era, por exemplo, o caso da programação em assembly para o sistema operativo MS-DOS. No Unix a situação é ligeiramente diferente, como se verá no capítulo 8. Este é o tipo de problemas que normalmente passam completamente despercebidos

[3] O primeiro processador da Intel de 32 *bits* foi o 80386. No entanto, a Intel considera que a arquitectura IA-32 se iniciou antes, remontando ao 8086. Isto deve-se ao facto de haver uma linha de continuidade entre este processador e os mais recentes processadores desta arquitectura que mantêm sempre retrocompatibilidade. *Retrocompatibilidade* significa que o mais moderno dos Pentium consegue correr código máquina escrito para o 8086. Mesmo os processadores Itanium da próxima geração, os primeiros pertencentes à arquitectura IA-64, de 64 *bits*, vão manter esta capacidade.

a um programador duma linguagem de alto nível.

Outro problema relevante para quem programa em assembly tem a ver com os registos. Também aqui o programador precisa de saber quantos registos tem disponíveis e o que é possível fazer com cada um deles. O contraponto são mais uma vez as linguagens de alto nível onde esta questão nem sequer se põe — o programador limita-se a definir variáveis e a utilizá-las sem saber quando, onde e se há correspondência com os registos.

No que diz respeito às instruções é mais uma vez necessário saber o que é ou não possível fazer com uma instrução ou o que requer múltiplas instruções, estando muitas vezes em jogo, em caso de se tomar uma decisão errada, o desempenho do programa.

Outro tipo de situações em que é particularmente relevante o conhecimento profundo da arquitectura, quando se escreve um programa em assembly — e estes são precisamente os casos em que a utilização do assembly muitas vezes mais se justifica — põe-se para realizar o controlo de um periférico ou para sincronizar duas tarefas diferentes, por exemplo.

Outro aspecto que exige um grande domínio por parte do programador diz respeito à utilização das bases de numeração binária e hexadecimal por um lado e à representação dos diversos tipos de dados em memória. Este tipo de conhecimento mostra-se relevante muito mais vezes do que seria de calcular, especialmente quando é necessário fazer a depuração do programa com as respectivas inspecções ao conteúdo da memória, que essa actividade normalmente implica.

Finalmente, uma palavra para a organização do programador propriamente dita. Dada a pouca estruturação da linguagem é especialmente importante que o programa seja cuidadosamente organizado em termos de funções, módulos e definições de variáveis, sob pena de se tornar incompreensível e

impossível de depurar ou alterar. Este é muitas vezes o ponto que separa o programador experimentado do principiante.

1.3 Vantagens e Desvantagens do Assembly

Actualmente, quando um programador decide escrever um programa opta, normalmente, por uma linguagem de alto nível. Nem sempre assim foi, mas o conjunto dos prós e dos contras quando pesado tende a favorecer uma linguagem como o C em detrimento do assembly.

Uma das maiores vantagens do assembly era a velocidade, mas esse factor tende a tornar-se irrelevante a cada nova geração de microprocessadores, cada vez mais rápidos. Outra das vantagens do assembly prende-se com a capacidade que a linguagem oferece para aceder directamente aos recursos de *hardware* da máquina, o que nem sempre sucede nas linguagens de alto nível. Igualmente, num programa escrito em assembly, é possível ao programador ter um controlo rigoroso do tempo de execução, pois é ele/a) que escolhe as instruções de linguagem máquina e tem possibilidade de saber quanto tempo leva cada uma delas a executar-se. O acesso directo ao *hardware* e o controlo do tempo de execução fazem do assembly uma opção a considerar (ou simplesmente a única opção) para escrever determinados fragmentos críticos de código, especialmente, código do próprio sistema operativo [4].

Ainda outra vantagem da linguagem assembly consiste na possibilidade de se escrever código especializado na interface entre módulos escritos em linguagens de alto nível diferentes e que utilizam interfaces incompatíveis.

O último argumento favorável ao assembly tem a ver com a natureza humana: muitos programadores consideram muito mais compensador escrever

[4]Note-se que os sistemas operativos actuais são escritos em linguagens de alto nível. Por exemplo, o Linux é quase todo escrito em C.

código em assembly, onde a proximidade com a máquina é muito maior, do que numa linguagem de alto nível.

As desvantagens, por outro lado, são mais numerosas e significativas. Talvez por estes motivos o assembly não é praticamente utilizado nos programas actuais, senão em pequenas secções de código, que se revelem fundamentais para o desempenho global.

Em primeiro lugar, o assembly é uma linguagem pouco estruturada. Isto significa que se exige do programador um esforço de organização adicional, sob pena de se tornar impossível manter um programa (i.e., corrigi-lo ou adicioná-lo).

Outro dos problemas do assembly, que está relacionado com este, prende-se com o elevado número de instruções que são necessárias para fazer qualquer tarefa. Isto significa que a programação é trabalhosa e muito propensa a erros, ao ponto de se perder mais tempo com detalhes inerentes à linguagem do que com os problemas propriamente ditos. Isto, porque também é muito complicado fazer a depuração dos erros dum programa escrito em assembly — mais do que numa linguagem de alto nível, onde à partida ocorrem muito menos erros, porque alguns deles são imediatamente detectados pelo compilador. Estes são, sem dúvida, defeitos de peso.

Um outro problema do assembly prende-se com a portabilidade, que é pouca ou nenhuma. Um programa escrito em assembly para uma dada plataforma dificilmente pode ser transportado para uma plataforma diferente. Este facto está em tudo relacionado com a necessidade de conhecer pormenorizadamente a arquitectura em que se está a programar. Note-se que esta situação é de certa forma contrária à abstracção que normalmente se deseja na resolução de um problema qualquer.

1.4 Assemblador

Como não se afigura fácil ao programador fixar ou consultar numa tabela todos os códigos das instruções de linguagem máquina de um microprocessador, bem como os respectivos operandos e comprimentos (em número de *bytes*), existem programas — chamados assembladores — para resolver o problema. Um assemblador permite designar as instruções e respectivos operandos por mnemónicas em vez de números inteiros. O programa é então escrito como uma lista de mnemónicas que são depois traduzidas pelo assemblador para linguagem máquina. Note-se que se trata duma tradução directa, i.e., ao assemblador não compete — pelo menos em teoria — nenhuma tarefa de compilação ou sequer de optimização (veja-se a secção 1.5).

A tabela 1.1 pretende ilustrar a diferença entre linguagem assembly e linguagem máquina. Para isso é apresentado um pequeno programa em assembly e a respectiva codificação em linguagem máquina. Na primeira coluna da tabela está representado o endereço de memória onde se encontra a instrução, na segunda a dimensão da instrução, depois a sua mnemónica e finalmente a instrução correspondente e respectivos operandos em linguagem máquina (em hexadecimal).

Endereço	Dimensão	Mnemónica	Linguagem Máquina
0x8048440	3	`movl 0x8(%ebp), %ebx`	0x8b5d08
0x8048443	3	`movl 0xc(%ebp), %eax`	0x8b450c
0x8048446	2	`addl %ebx, %eax`	0x01d8
0x8048448	1	`ret`	0xc3

Tabela 1.1: Codificação da linguagem assembly em linguagem máquina

A assemblagem não é, no entanto, o único processo pelo qual um programa em assembly tem que passar antes de se tornar num executável. Desde o seu início são necessários os seguintes passos:

- escrever o programa em assembly com recurso a um editor de texto. Guardá-lo num ficheiro de texto (com extensão '`s`' no caso do assemblador Gas);
- assemblar o programa assim escrito. O resultado será um ficheiro objecto (usa-se o programa `as` — Gas — para este fim);
- o(s) ficheiro(s) objecto assim produzido(s) são editados pelo editor de ligações (*linker*, em inglês — programa `ld` no Unix), que liga as várias referências e recoloca os objectos, construindo no fim desse processo um ficheiro executável.

A figura 1.1 resume este processo.

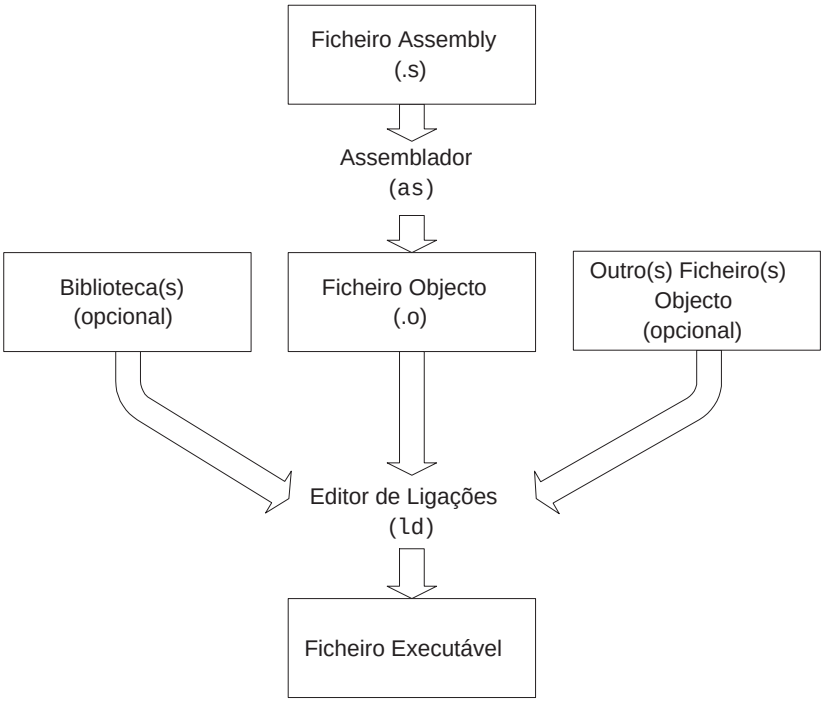

Figura 1.1: Construção de um executável a partir do *assembly*

Para simplificar a tarefa do programador pode-se utilizar o compilador

de *C*, `gcc`. Este compilador permite fazer a assemblagem e/ou a ligação. Para isso, invoca os programas `as` e `ld`, fornecendo-lhes todos os parâmetros necessários — que no caso do `ld` não são triviais. Sendo assim, para criar um executável a partir dum ficheiro de código fonte assembly chamado `ficheiro.s` basta fazer

`gcc ficheiro.s`.

Se o ficheiro contiver tudo o que for necessário para constituir um programa, isto será suficiente para criar um ficheiro executável de nome `a.out`.

1.5 Compilação *vs.* Assemblagem

Pese embora o facto de não ser objectivo deste texto apresentar uma linguagem de alto nível, convém perceber o papel de um compilador na concepção dum programa e a diferença entre compilação e assemblagem.

Como já se disse, um assemblador faz a tradução directa de mnemónicas para códigos de linguagem máquina. Pelo contrário, um compilador tem uma tarefa muito mais complexa, que consiste em construir um programa assembly, a partir do programa escrito numa linguagem de alto nível [5], com respectivas estruturas, instruções e tipos de dados, que poderão ser arbitrariamente complicados.

A complexidade de construir um programa é em parte passada do programador para o compilador embora, naturalmente, não seja este último a resolver os problemas propriamente ditos.

[5] Isto não significa que todos os compiladores de todas as linguagens façam uma tradução da linguagem de alto nível para assembly, antes de ser feita a tradução para linguagem máquina. Um bom exemplo duma situação diferente é o compilador de Java, que passa o programa para uma linguagem de baixo nível universal — *bytecode*, que pode não ter e provavelmente não terá correspondência directa com qualquer linguagem assembly da plataforma em questão.

A figura 1.2 ilustra o papel do compilador na concepção dum programa. Compare-se esta figura com a figura 1.1 que foi apresentada atrás.

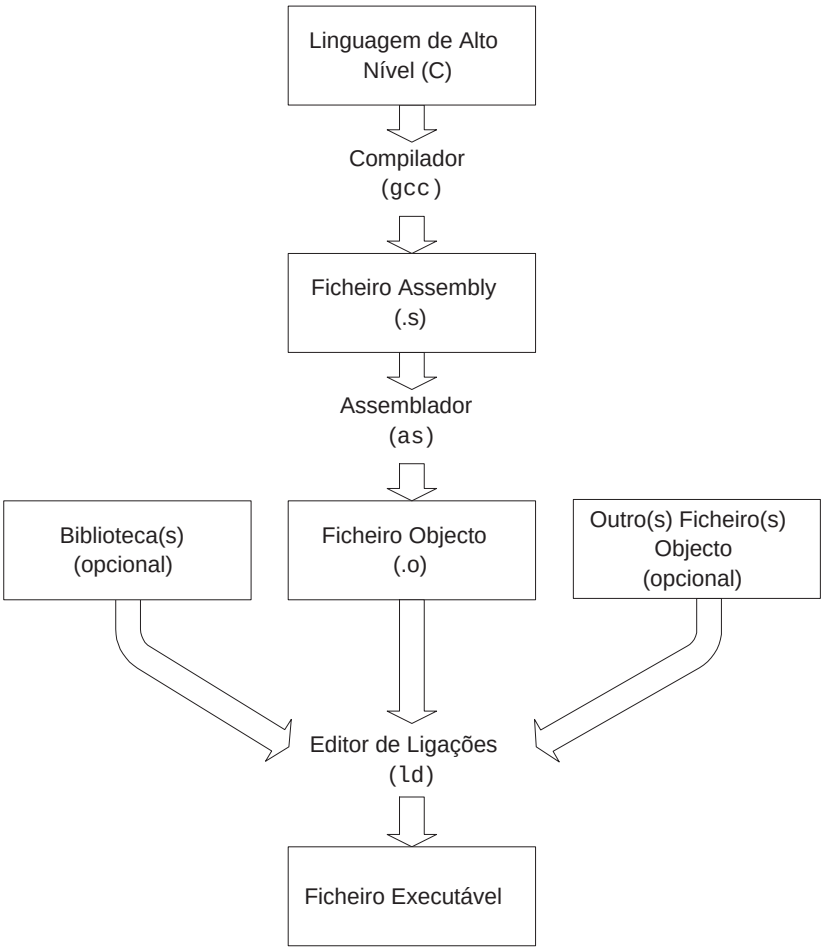

Figura 1.2: Construção de um executável a partir de uma linguagem de alto nível

2

Arquitectura

2.1 Introdução

Para um programa que utilize linguagens de alto nível parecem existir essencialmente dois níveis hierárquicos distintos de memória: primária e secundária ou, respectivamente, memória central (RAM — *Random Access Memory*) e disco. Na realidade existe um outro nível de memória, mais próxima do processador do que as outras, utilizável pelo programador de linguagem assembly: os registos [1].

Em termos breves, um registo é uma célula de memória, identificada por um nome, capaz de guardar um único valor de cada vez. Consoante a arquitectura do microprocessador pode haver mais ou menos registos, sendo que no 80386 há pouco mais do que 10, embora o número de registos susceptíveis de serem manipulados livremente seja inferior a este. A vantagem de utilizar registos, em vez de usar directamente a memória esta na muito maior velocidade de acesso. Esta aliás é uma questão relevante e que se tem vindo a agravar ao longo dos anos: o desempenho do processador tem evoluído muito mais rapidamente que o da memória; por este motivo os acessos à memória são, em termos relativos, cada vez mais dispendiosos.

[1] Em C também é possível aconselhar o compilador a utilizar registos, mas não existe garantia de que este o faça.

O 80386 e os seus sucessores até à data (Pentium 4) são microprocessadores de 32 *bits*. Entre outras coisas, isto significa que, geralmente, as instruções que estes microprocessadores executam lidam com operandos de 32 *bits* no máximo e que a dimensão dos registos corresponde também a este valor. Operandos de 32 *bits* significam caminhos de dados de acesso à memória (*bus* de dados) de 32 *bits*. Por razões históricas todos os registos de 32 *bits* podem em determinadas condições ser usados como registos de 16 *bits*. Alguns deles podem ainda ser divididos num par de registos de 8 *bits*. Como se verá, também continuam, apesar disto, a existir registos com apenas 16 *bits*, que servem de selectores de segmentos de memória (o conceito de segmento de memória fica adiado para o capítulo 3).

É de referir que, apesar da arquitectura ser de 32 *bits*, permite ainda utilizar operações e operandos de 16 *bits*, mesmo no modo protegido (ver secção 3.3 para uma descrição sobre os modos existentes nesta arquitectura). Isto é possível graças a uma *flag*, designada por *flag D*. Esta *flag* armazena um atributo presente em todos os descritores de segmentos de código, que determina o tamanho por omissão das operações e dos operandos ($D = 0$ para operações e operandos de 16 *bits*, $D = 1$, para 32 *bits*). No entanto, o valor desta *flag* pode ser invertido separadamente para as operações ou para os operandos de um única instrução, através de prefixos que se aplicam apenas a essa instrução. Veja-se (IA-32 developer's manual-I, 2001). Neste texto, sempre que necessário, assumiremos que operações e operandos de 32 *bits*.

2.2 Registos

2.2.1 Registos de Utilização Geral

Há oito registos de 32 *bits* que são considerados de utilização geral. Os seus nomes são `eax`, `ebx`, `ecx`, `edx`, `esp`, `ebp`, `esi` e `edi`. Antes do 80386 estes registos tinham apenas 16 *bits*, sendo os seus nomes iguais a menos da letra 'e' no início (`ax`, `bx`, ..., `di`). Ainda assim, continua a ser possível utilizar apenas os 16 *bits* menos significativos dos novos registos, bastando para isso aplicar as designações antigas.

Os 16 *bits* menos significativos dos registos `ax`, `bx`, `cx` e `dx` podem ainda ser divididos em dois registos de 8 *bits*, de acesso separado. Os oito registos de utilização geral e respectivas divisões estão representados na figura 2.1.

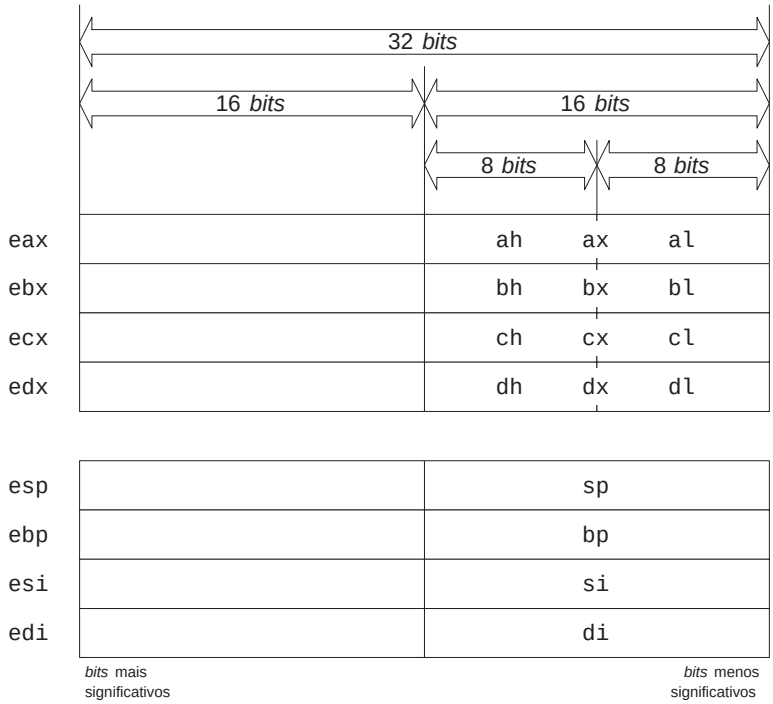

Figura 2.1: Registos de utilização geral

O nome de "registos de utilização geral"é aqui um pouco enganador, porque nem todos estes registos podem ser manipulados livremente e alguns deles apresentam funções específicas. No 80386 as diferenças entre os registos são menores do que nos seus antecessores mas, no entanto, continuam a existir.

Os registos `esp` e `ebp` são utilizados na manipulação da pilha (veremos adiante o que é uma pilha). O registo `esp` — *Extended Stack Pointer* — indica o topo da pilha. Alterar este registo significa mexer na localização do topo da pilha, pelo que estas alterações não podem ser arbitrárias. O registo `ebp` — *Extended Base Pointer* — é um registo auxiliar que permite manipular valores armazenados na pilha. É especialmente útil quando se utiliza a pilha para armazenar variáveis locais e para passar parâmetros a funções.

Os registos `esi` e `edi` chamam-se registos de índice, porque são muitas vezes usados para aceder a vectores ou tabelas (i.e., indexar vectores ou tabelas). A sigla `esi` significa *Extended Source Index* enquanto `edi` significa *Extended Destination Index*.

Os outros quatro registos têm, de facto, uma utilização mais geral, sendo `eax` conhecido por acumulador, `ebx` por base, `ecx` por contador e `edx` por registo de dados. Estes nomes reflectem algumas diferenças que se prendem com determinadas instruções que lhe estão individualmente associadas, como se verá ao longo do texto.

2.2.2 Registo de Segmento ou Selectores

No capítulo 3 será feita uma abordagem à organização da memória na arquitectura do 80386. Nessa altura será abordada com algum detalhe a questão da divisão de um programa em partes designadas de segmentos. Durante a

execução do programa a indicação dos segmentos é feita por registos destinados exclusivamente para esse fim. São eles os registos `cs`, `ss`, `ds`, `es`, `fs` e `gs`, todos de 16 *bits* e indivisíveis. Na figura 2.2 estão representados estes registos. No capítulo 3 será apresentada a tabela 3.1 onde estão resumidas as respectivas funções.

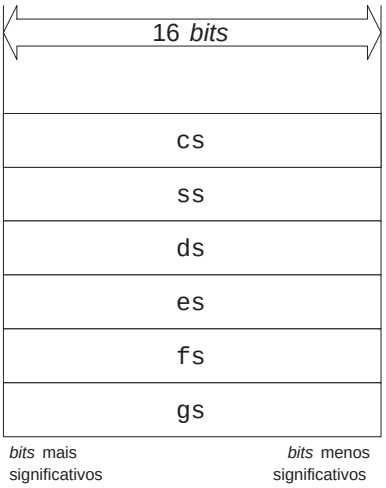

Figura 2.2: Registos de segmentos ou selectores

2.2.3 Registo Contador de Programa

Um registo que nenhum microprocessador pode dispensar é o contador de programa, que indica qual a próxima instrução a ser executada. Isto porque, depois de se carregar o programa para memória primária, é necessário saber, em cada momento, o endereço de memória onde pode ser encontrada a próxima instrução e respectivos operandos — guardar este endereço é a função do Contador de Programa.

"Contador de Programa"é um nome genérico, sendo o verdadeiro nome deste registo, no 80386, *Extended Instruction Pointer* — `eip`, ou Ponteiro de Instruções Estendido. A sua representação está feita na figura 2.3. Antes

do 80386 este registo tinha apenas 16 *bits* e chamava-se `ip`. Na figura 2.4 encontra-se um exemplo onde se pode ver que o `eip` armazena o endereço da próxima instrução a ser executada (e não da que está a ser executada no instante actual). Confrontando esta figura com a tabela 1.1, facilmente se verifica que a próxima instrução a executar será `movl 0xc(%ebp), %eax`.

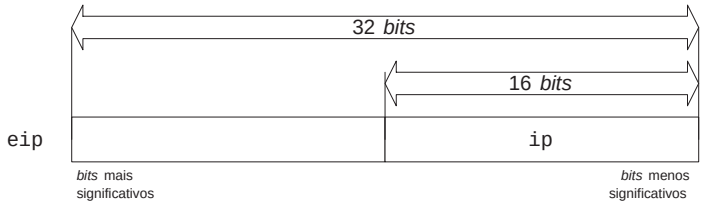

Figura 2.3: Representação do registo `eip`

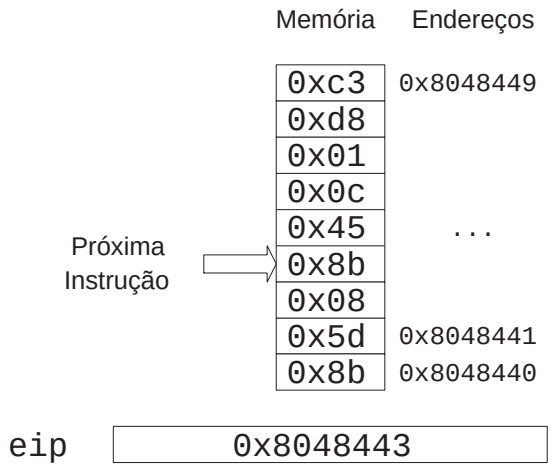

Figura 2.4: Exemplo de utilização do registo `eip`

2.2.4 *Flags*

Existe também um registo — registo `eflags`, *Extended Flags* — destinado a assinalar a ocorrência de determinadas condições. Estas condições dizem

respeito a certos eventos no código em execução, que podem ser, por exemplo, uma operação aritmética cujo resultado ultrapassou a capacidade do registo onde iria ser armazenado; pode ser também um resultado igual a 0 na última operação ou um resultado negativo, entre outros, que serão enumerados mais adiante.

O registo `eflags` encontra-se representado na figura 2.5 e a lista das *flags* mais importantes encontra-se descrita na tabela 2.1.

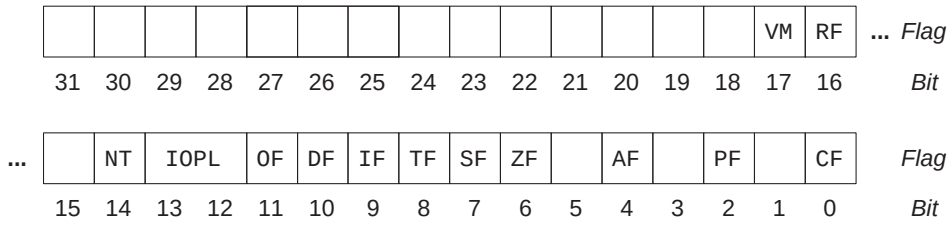

Figura 2.5: Representação do registo `eflags`

Estas *flags* são essenciais ao funcionamento dum programa. Com efeito, algumas das instruções utilizadas em estruturas de decisão (ver instruções `cmp` e `j`) seguem um procedimento que primeiro altera as *flags* em função de dois operandos e depois, em função dessas alterações, efectuam, ou não, um salto na sequência de instruções do programa. Sem esta capacidade escrever um programa de computador seria muito mais difícil se não impossível.

Os acrónimos das restantes *flags* significam:

- *TF*: *Trap Flag*;
- *IF*: *Interrupt Enable Flag* (*flag* de habilitação das interrupções);
- *IOPL*: *Input/Output Privilege Level Flag* (*flag* de privilégio de entradas/saídas);
- *NT*: *Nested Task Flag*;

- *VM*: *Virtual Memory Flag* (*flag* de memória virtual);
- *RF*: *Resume Flag* (*flag* de continuação).

A utilidade destas *flags* pode ser encontrada em (Murray & Pappas, 1986).

2.3 Conjunto de Instruções Abstractas

Ao longo deste texto serão apresentadas muitas das instruções da arquitectura de 32 *bits* dos processadores Intel. Para auxiliar nos comentários a efectuar utilizaremos um conjunto de instruções abstractas. Pretende-se que estas instruções não tenham qualquer relação com esta arquitectura específica. Pelo contrário, estas instruções deverão ser independentes de qualquer arquitectura e idealmente deverão permitir a descrição das instruções de qualquer processador. Este conjunto de instruções está representada na tabela 2.2. Muitas das operações, sobretudo aritméticas e operações que envolvem *bits*, mas não só, são idênticas às da linguagem de programação *C*.

2.4 Simbologia e Nomenclatura

Antes de se entrar no estudo do assemblador Gas propriamente dito é necessário apresentar a convenção utilizada pelo assemblador e que será também adoptada neste texto, relativa à simbologia. Em particular, interessa saber o seguinte:

- como se acede ao conteúdo duma célula de memória;
- como se utiliza o endereço duma célula de memória;

- como se acede a um registo;
- como se definem constantes;
- como se efectuam comentários;
- qual a nomenclatura adoptada para a dimensão dos dados.

No que diz respeito ao acesso ao conteúdo duma célula de memória, este faz-se em dois passos, que são dados, normalmente, em momentos distintos dum programa. Primeiro, na definição das variáveis do programa é (são) especificada(s) a(s) célula(s) de memória (num total de 1, 2, 4, 8 ou mais células, sempre em grupos que sejam potências de 2) onde é armazenada essa variável e qual o nome que a(s) identifica, que é também o nome dessa variável. O segundo passo do acesso faz-se com referência directa ao seu nome e acontece, normalmente, no corpo do programa. Note-se que se a célula de memória inicial não estivesse identificada pelo nome poder-se-ia utilizar directamente o número que corresponde ao seu endereço se este fosse conhecido — na prática dificilmente será.

Assim, se houver uma instrução chamada `xpto`, que recebe um operando e se existir uma célula de memória que corresponde a uma variável designada por `numero` a instrução

`xpto numero` **numero** $\longrightarrow$ **M[numero]**

faz um acesso ao conteúdo da(s) célula(s) de memória que armazena(m) a variável `numero` (i.e., um acesso à memória).

De igual forma, se em vez de um nome for utilizado um número, esse número vai ser interpretado como um endereço absoluto (na prática o endereço será virtual). Isto significa também que um número nunca é interpretado literalmente, mas sempre como um endereço de memória. Por este

motivo, sempre que for necessário usar um valor numérico literal tem de ser usado um cifrão (‘`$`’) antes do número. Assim, a instrução

```
xpto $4                    $4 ⟶ 4
```

não resultará em nenhum acesso ao endereço 4, sendo em vez disso usado o valor literal 4. De igual forma, também a expressão `$numero` seria interpretada como um valor literal, sendo este valor igual ao endereço atribuído à variável número.

Para diferenciar o acesso aos registos da máquina dos acessos a variáveis, os nomes dos registos aparecem sempre precedidos do sinal de percentagem (‘`%`’). Por exemplo: `%eax`, `%ebx`, `%ax`, `%al`, etc.

No que toca à definição de constantes (como é o “4”do exemplo anterior) vão ser referidos dois tipos de constantes: numéricas e cadeias de caracteres. As constantes numéricas podem ser definidas de várias formas diferentes e usando, inclusivamente, quatro bases de numeração diferentes, o que não deixa de ser útil quando se está a programar em assembly. A forma mais simples de definir uma constante numérica é escrevendo-a directamente na base 10, como se fez no exemplo anterior com o “4”. Para escrever um número na base 2 terá de ser usado o prefixo `0b`, enquanto que na base 8 e na base 16 os prefixos são, respectivamente `0` e `0x`. Assim o número 16 seria escrito como `0b10000`, `020` e `0x10` em binário, octal e hexadecimal, respectivamente.

Uma constante em vírgula flutuante, por sua vez, recebe o prefixo `0f`, sendo admitida a notação científica com o expoente a surgir depois de um ‘`e`’ (maiúsculo ou minúsculo). Por exemplo, `0f1.3` ou `0f1.3e2`.

É ainda possível tomar directamente o valor ASCII [2] de um caracter escrevendo o caracter entre plicas (`'a'`, por exemplo) ou depois de uma plica (`'a`).

[2] American Standard Code for Information Interchange.

Em qualquer destes casos sempre que se quiser tomar o valor literal da constante é obrigatória a utilização do cifrão (‘`$`’) (quando o que estiver em causa for uma instrução e não uma directiva do assemblador).

As constantes que representam cadeias de caracteres, por sua vez, são definidas entre aspas.

Finalmente, o cardinal (‘`#`’) é utilizado para iniciar comentários. Numa linha de um programa em assembly do Gas tudo o que está depois do (‘`#`’) é ignorado.

No que se refere a dimensões, o Gas utiliza as seguintes designações: *byte* — 8 *bits*; *word* (palavra) — 16 *bits*; *long* — 32 *bits*. Na secção 3.2 este assunto voltará a ser abordado e serão apresentadas designações alternativas. A cada uma destas dimensões corresponde um sufixo, que deverá ser adicionado às instruções conforme o tamanho dos dados em que estas deverão operar: respectivamente ‘`b`’, ‘`w`’ ou ‘`l`’. Assim, se a instrução `xpto` operar sobre as células de memória que referenciam a variável de 32 *bits* `numero`, a instrução deverá ser rescrita para

```
xptol numero
```

Muitas vezes, a utilização deste sufixo não é obrigatória, mas é aconselhada por uma questão de clareza na leitura do código.

A tabela 2.3 resume a definição dos símbolos e da nomenclatura que tem vindo a ser feita.

Sigla	Descrição	Significado
`CF`	*Carry Flag* (*Flag* de Transporte)	Posta a 1 quando é gerado um transporte ou um empréstimo numa operação aritmética. Caso contrário é posta a 0. Note-se que a ocorrência de transporte ou empréstimo não implica transbordo.
`PF`	*Parity Flag* (*Flag* de Paridade)	Indica se nos 8 *bits* menos significativos de um resultado há um número ímpar de *bits* a 1 (caso em que é posta a 0) ou se há um número par de *bits* a 1 (caso em que é posta a 1).
AF	*Auxiliary Carry Flag* (*Flag* de Transporte Auxiliar)	Usada em aritmética BCD para indicar transporte ou empréstimo.
`ZF`	*Zero Flag* (*Flag* de Zero)	É posta a 1 quando um resultado der 0, posta a 0, caso contrário.
`SF`	*Signal Flag* (*Flag* de Sinal)	Posta a 1 quando o sinal do resultado é negativo, limpa, no caso contrário.
`OF`	*Overflow Flag* (*Flag* de Transbordo)	Indica a ocorrência de transbordo no resultado duma operação em complemento para dois, caso em que é posta a 1. É posta a 0, no caso contrário. O transbordo dá-se quando o número de *bits* utilizados para armazenar o resultado não é suficiente. Em adições em complemento para dois esta situação ocorre quando o transporte que entra no *bit* mais significativo é *diferente* do transporte que sai. Esta situação não deve ser confundida com o transbordo em operações sem sinal, que ocorre quando há transporte que sai do *bit* mais significativo (ver *flag* de transporte).
`DF`	*Direction Flag* (*Flag* de Direcção)	Indica a direcção das operações com *strings*. Quando a *flag* de direcção está a 0, os registos de índice, `esi` e `edi`, são automaticamente incrementados no fim de certas operações utilizadas em cópias de dados (nomeadamente de *strings*) em memória. Caso a *flag* de direcção esteja a 1, esses registos são automaticamente decrementados.

Tabela 2.1: Descrição das *flags* mais importantes

Símbolo	Significado
`eax`	Representação dum registo. Note-se a ausência do sinal de percentagem ('`%`') antes do nome do registo. As *flags* representam-se de forma semelhante. Por exemplo, a *flag* de sinal representa-se como `SF`.
25	Representação dum literal. Neste caso do 25.
$\longleftarrow$	Operação *toma o valor de*. Ao operando (que pode resultar duma expressão) que se encontra do lado esquerdo é atribuído o valor da expressão que se encontra do lado direito. Exemplo: `eip` $\longleftarrow$ `eip + 2`. Neste caso o registo `eip` é incrementado de duas unidades.
$\underset{x}{\longleftarrow}$	Operação *toma um valor convertido para* x bits . Idêntica à operação anterior, mas antes da atribuição a expressão da diretia é convertida para `x` *bits*, se for de tamanho diferente. Exemplo: `eax` $\underset{32}{\longleftarrow}$ `4`. É feita a extensão do número 4 para 32 *bits* e o respectivo valor é atribuído ao registo `eax`.
`M[x]`	Refere-se ao conteúdo duma ou mais células de memória. `x` representa o endereço de memória a que se faz o acesso. O número de células referenciadas depende do contexto, mas poderá ser 1, 2, 4 ou 8. Normalmente este operando surge numa expressão em que aprece $\underset{t}{\longleftarrow}$. Sempre que isso acontece o número de células referenciadas será de `t`/8, uma vez que `t` é dado em *bits*. Exemplo: `M[eax]` $\underset{32}{\longleftarrow}$ `4`. Neste caso, o valor decimal 4 é convertido para um valor binário de 32 *bits* a armazenar nas quatro células de memória com início no endereço `eax`.
$\longrightarrow$	Operação *representa*. A expressão que está à esquerda, escrita em assembly representa a expressão que está à direita. A expressão da direita obedece às convenções desta tabela. Exemplo: `20(%ebx, %eax, 4)` $\longrightarrow$ `M[ebx + 4 * eax + 20]`.
$\longleftrightarrow$	Operação *troca de valor com*. O valor esquerdo e o direito trocam de valores. Exemplo: `eax` $\longleftrightarrow$ `ebx`. Se `eax` valesse 6 e `ebx` 14, no fim da instrução `eax` valeria 14 e `ebx` 6.
$\underset{x}{\longleftrightarrow}$	Operação *troca de valor de* $\boldsymbol{x}$ bits *com*. Idêntico ao anterior, mas aqui é indicado o tamanho dos operandos. Exemplo: `M[eax]` $\underset{32}{\longleftrightarrow}$ `ebx`. O conteúdo das 4 células de memória cujo endereço se inicia em `eax` trocam de valor com o registo `ebx`.
$\leftarrow\!\!-\!\!-$	Operação *altera as* flags *sem atribuir o valor de*. Semelhante à operação $\longleftarrow$, mas sendo o operando esquerdo o registo das *flags*, que é alterado como na instrução $\longleftarrow$. O resultado do cálculo da expressão que fica à direita é descartado.
$\underset{F}{\longleftarrow}$	Operação *toma o valor de, sem alterar as* flags.. Semelhante à operação $\longleftarrow$, mas não altera as *flags*.
$\left\vert\longleftarrow\right\vert$, $\left\vert\underset{x}{\longleftarrow}\right\vert$	Operações *atribuição sem sinal*. Semelhantes à operação $\longleftarrow$, mas a expressão da direita é avaliada considerando os números como sendo sempre positivos. Exemplo: `edx:eax` $\left\vert\underset{64}{\longleftarrow}\right\vert$ `ecx` $\times$ `eax`. Os dois factores da multiplicação `ecx` e `eax` são, neste caso, considerados como números positivos.

Símbolo	Significado
$\circlearrowleft$ `x`, $\circlearrowright$ `x`	Operação *rotação para a esquerda/direita de* **x** bits . Estas operações serão definidas adiante. Exemplo: `eax` $\circlearrowleft$ `4`. Neste caso, o registo `eax` sofre uma rotação de 4 *bits* para a esquerda.
$\underset{C}{\circlearrowleft}$ `x`, $\underset{C}{\circlearrowright}$ `x`	Operação *rotação para a esquerda/direita de* **x** bits *pela* flag *de transporte*. Estas operações serão igualmente definidas adiante. Exemplo: `eax` $\underset{C}{\circlearrowleft}$ `4`. Neste caso, o registo `eax` sofre uma rotação de 4 *bits* para a esquerda com passagem pela *flag* de transporte.
`<<` `>>`	Operação *deslocamento para esquerda/direita de* **x** bits . A definir adiante, também. Exemplo: `eax >> 4`. Neste caso o registo `eax` sofre um deslocamento de 4 *bits* para a direita.
`edx:eax`	Representação da justaposição de dois registos como se de um só se tratasse.
`×`,`/`, `%`, `+`, `-`	Operações aritméticas. Respectivamente: multiplicação, divisão, módulo, adição e subtracção.
`==`,`!=`	Operações de comparação. Respectivamente: igual e diferente. A operação igual devolve verdade se a avaliação da expressão à esquerda resultar idêntica à avaliação da expressão à direita. Exemplo: `4 + 3 == 7` avalia como verdade, mas `eax == 8` avalia como falso, se `eax` armazenar o valor 20. A operação diferente devolve sempre o resultado complementar da operação igual.
`&`, `|`, `^`	Operações binárias *bit* a *bit*. Respectivamente: *e*, *ou* e *ou-exclusivo*. Exemplo: `eax & ebx`. Estas operações serão apresentadas mais adiante.
`~`	Operação unária de negação. Complementa todos os *bits* do operando indicado. Exemplo: `~ebx`. Esta operação será apresentada mais adiante.
`&&`, `||`	Operações aritméticas lógicas. Respectivamente: *e lógico* e *ou lógico*. Exemplo: `ZF == 0 && SF == OF`. Esta expressão é avaliada como verdade se a *flag* `ZF` for igual a 0 e as *flags* `SF` e `OF` forem iguais.
`?`	Operação de controlo de fluxo *se-então*. Se a expressão à esquerda do operador `?` for avaliada como verdade é executada a instrução que se encontrar à direita. Exemplo: `ecx != 0 ?` `eip` $\underset{32}{\longleftarrow}$ `eip - 10`. Neste caso se o registo `ecx` for diferente de 0 o registo `eip` é decrementado em 10 unidades, caso contrário passa-se à instrução seguinte.
`?:`	Operação de controlo de fluxo *se-então-senão*. Se a expressão à esquerda do operador `?` for avaliada como verdade é executada a instrução que se encontrar entre os sinais `?` e `:`. Caso contrário é executada a instrução à direita de `:`. Exemplo: `ecx != 0 ?` `eip` $\underset{32}{\longleftarrow}$ `eip - 10 :` `eax` $\underset{32}{\longleftarrow}$ `eax + 2`. A diferença relativamente ao caso anterior, é que agora o registo `eax` é incrementado de duas unidades caso `ecx` seja igual a 0.

Tabela 2.2: Lista das instruções abstractas (cont.)

Tipo de símbolo	Representação
Variáveis	Designadas por um nome que equivale a um endereço duma célula de memória.
Valores literais	Obtidos por um cifrão (‘`$`’)
Constantes	Números em binário: `0b01001110`. Números em octal: `07431`. Números em hexadecimal: `0x1f00`. Números de vírgula flutuante: `0f1.3e2`. Caracteres: `'a'`. Strings: `"Entre aspas"`.
Comentários	Precedidos por um cardinal (‘`#`’).
Registos	Precedidos por um sinal de percentagem (‘`%`’).
Nomenclatura	*byte*: 8 *bits*. Palavra: 16 *bits*. Inteiro/Longo: 32 *bits*. *Double* (depende do contexto): 32 ou 64 *bits* ou número de vírgula flutuante.
Sufixos	*byte*: ‘`b`’. Palavra: ‘`w`’. Longo: ‘`l`’.

Tabela 2.3: Definição de símbolos e nomes

3

Endereçamento

3.1 Introdução

Neste capítulo vai ser estudada a organização da memória dum computador. Em particular, veremos algumas características que são específicas à família Intel 80x86. Veremos também algumas das diferenças que separam o 80386 dos seus antecessores e em que é que essas diferenças se reflectem em termos de programação.

O capítulo termina com um estudo das diferentes formas de aceder à memória na linguagem assembly do Gas.

3.2 Conceitos Básicos

Em termos lógicos, a memória dum computador pode ser encarada como um vector de *bytes*, indexado por um número inteiro, desde 0 até um limite máximo, que é igual à sua dimensão menos uma unidade.

Neste modelo estão representados vários aspectos relativos ao funcionamento duma memória que convém compreender:

- uma memória tem capacidade finita de armazenar informação;
- a informação tem que ser armazenada em múltiplos de 8 *bits* (1 *byte*);

- a cada *byte* armazenado na memória está associado um índice — a que chamaremos endereço — que indica a sua localização exacta;
- estruturas de dados com dimensão superior a 1 *byte* terão de ser armazenadas em várias posições de memória diferentes. Geralmente estas posições são contíguas, em particular no caso das estruturas de dados mais simples, como é o caso dos inteiros com vários *bytes*, por exmeplo.

A figura 3.1 procura ilustrar este modelo para uma memória com dimensão de 16 Kbytes (16.384 *bytes*).

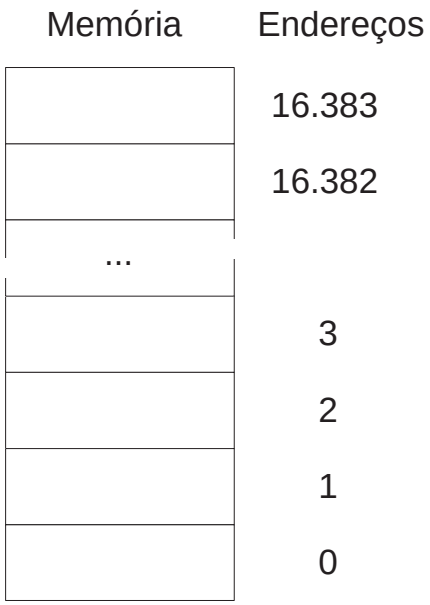

Figura 3.1: Representação de uma memória com 16 K

As memórias modernas são também designadas de memórias de acesso aleatório (*Random Access Memories* — RAM) por oposição ao que sucedia no passado em certo tipo de memórias.

É de notar, e isso resulta do que já dissemos, que a memória, em termos físicos é acessível duma forma totalmente linear: quaisquer duas posições de

memória consecutivas são referenciadas por dois números inteiros também consecutivos. Este facto que ao nível do endereçamento físico parece óbvio não vai encontrar paralelo ao nível de um programa, onde dois endereços consecutivos podem não referenciar células de memória consecutivas. Este facto deve-se a um mecanismo, designado de memória virtual, que ultrapassa o âmbito deste texto. Veja-se (Silbertschatz & Galvin, 1998). De qualquer forma o programador não tem que ter consciência desse facto, quando escreve um programa.

Se é verdade que a memória está dividida em *bytes*, a realidade é que muitas vezes, e como já dissemos, esta unidade se revela inadequada ou mesmo insuficiente em certas situações. O caso mais vulgar será aquele em que é necessário armazenar um número que não pode ser representado com apenas 8 *bits*. No caso do 80386 estão previstas outras dimensões básicas para estruturas de dados, que podem ser acedidas com uma única instrução: a *palavra* (*word*), de 16 *bits* e o *inteiro* ou *longo* (*integer* e *long*, respectivamente), de 32 *bits*. Estes três tipos de dados estão representados na figura 3.2.

Os processadores da família 80x86, a partir do 80386, permitem aceder directamente a tipos de dados de 32 *bits* numa única instrução. Esse máximo era, até ao 80286, inclusive, de 16 *bits* [1].

Quanto ao armazenamento em memória de dados de 32 *bits* este é sempre feito em quatro posições de memória consecutivas. Assim, se um inteiro de 32 *bits* está armazenado no endereço 400, por exemplo, o inteiro seguinte só poderá ser armazenado da posição 404 em diante.

[1] Com raras excepções, como as instruções `div` e `mul`.

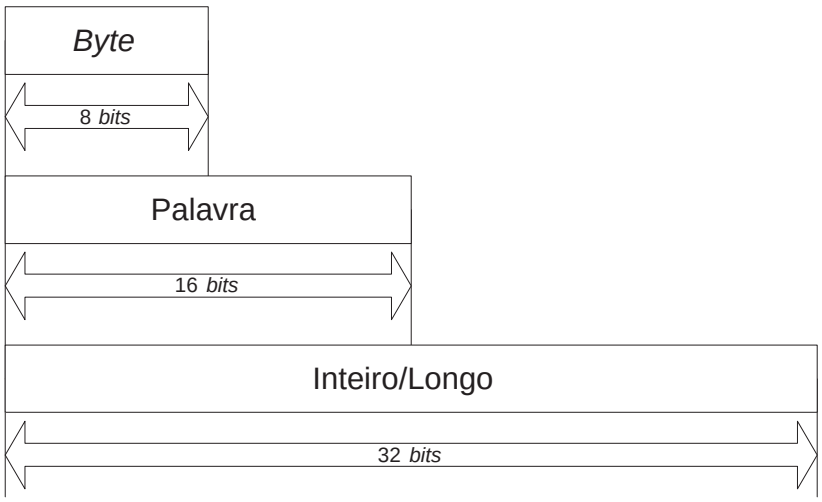

Figura 3.2: *byte*, palavra e longo

3.3 Divisão em segmentos

Se do ponto de vista físico a memória é perfeitamente linear, por outro lado, para um programa em execução faz sentido que a memória se divida em várias zonas, uma vez que ele próprio é composto por múltiplas partes: o código, a zona de dados inicializados, a zona de dados não inicializados, a pilha (onde são muitas vezes guardados dados locais) e nos sistemas operativos modernos incluem-se ainda um conjunto de funções, que não pertencem ao programa propriamente dito, mas que são por ele invocadas — estas funções pertencem a bibliotecas partilhadas [2].

A existência de todas estas partes justifica a divisão da memória em zonas chamadas segmentos. Cada segmento tem um tamanho máximo de 2^{32} *bytes* (4 Gbytes) e é endereçado linearmente — este tamanho deve-se ao facto de

[2]No Linux, talvez a biblioteca partilha mais conhecida seja a libc, que é utilizada por virtualmente todos os programas escritos em C. Isto significa que há um determinado número de funções (`printf`, por exemplo — que permite escrever na saída padrão), cujo código não está incluído nos programas, mas que todos eles invocam e que se encontra numa biblioteca dinâmica a que todos têm acesso.

os registos usados para indicar o deslocamento dentro do segmento, `esp`, `ebp`, `eip`, etc. terem 32 *bits*. Para designar os segmentos são utilizados registos de segmento ou selectores (`cs`, `ss`, `ds`, `es`, `fs` e `gs`) que têm 16 *bits*, dos quais apenas os 14 primeiros são usados para endereçamento (os outros dois são usados para indicar a protecção que se aplica ao segmento). Por este motivo um programa pode aceder a um limite teórico de 2^{46} *bytes* (64 Tbytes), sendo que metade deste espaço tem que ser partilhado entre todos, porque o último dos 14 *bits* de endereçamento dos selectores está reservado para separar os segmentos individuais dos comuns.

Cabe ao *hardware* e ao sistema operativo garantir que a distribuição e o acesso à memória física existente (que é inferior em várias ordens de grandeza à dimensão de memória endereçável pelos processos) se faz de forma coerente, em função do endereço, que será dado por um par {`segmento, endereço`}.

Os endereços de 48 *bits* dados pelos registos de segmento mais os registos de deslocamento acabam por ser traduzidos num endereço de 32 *bits* que é o máximo que um 80386 consegue, de facto, endereçar. O microprocessador utiliza, para isso, tabelas preparadas pelo sistema operativo. Acresce ainda a isto que a memória é paginada, mas como este facto é totalmente transparente para os programadores não vai ser abordado. Por este motivo, o endereço de 32 *bits* não é ainda um endereço físico, mas um endereço virtual que terá de ser traduzido num endereço físico. Veja-se (Silbertschatz & Galvin, 1998). A figura 3.3 procura ilustrar a conversão de um endereço da forma {`segmento, endereço`} de 48 *bits* para um endereço virtual de 32 *bits*. Note-se que um programa só lida com endereços virtuais, nunca lhe interessando saber qual é a correspondência que existe entre estes e os endereços físicos. A correspondência, também aqui, é garantida por uma cooperação entre o *hardware* e o sistema operativo.

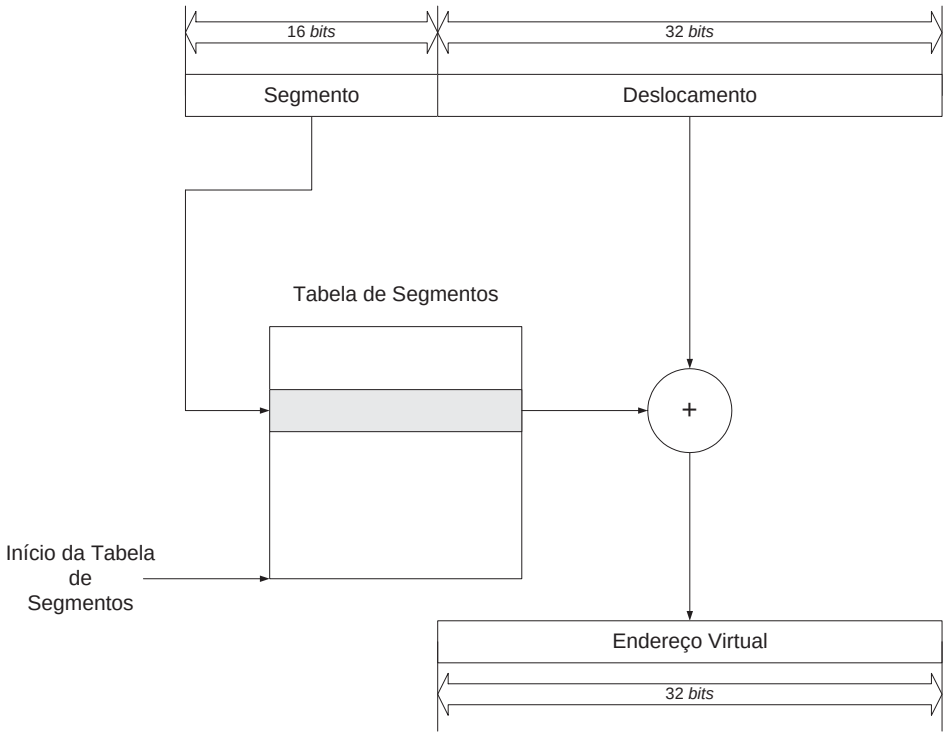

Figura 3.3: Tradução de um endereço de 48 *bits* num endereço virtual

É importante referir dois aspectos: a existência do endereçamento virtual justifica-se porque permite que cada programa utilize a gama de endereços que mais lhe convém, sem haver problemas quanto à coincidência de gamas entre programas diferentes. Por outro lado, um programa não pode tentar aceder a um endereço que ultrapasse os limites especificados para o segmento em causa, nem tão pouco tentar aceder a um segmento que não lhe pertence. Ao tentar fazê-lo o microprocessador gera uma excepção que poderá ser tratada pelo sistema operativo de diferentes formas. O Unix em geral e o Linux em particular, enviam um sinal `SIGSEGV` ao processo que tentou o acesso ilegal [3]. A resposta do programa a este evento geralmente não existe, embora possa ser especificada, pelo que o programa é terminado, sendo

[3]Para consultar uma lista completa de sinais veja-se (Stevens, 1990).

gerado um ficheiro com o estado em que este se encontrava no momento da terminação (`coredump`).

Apesar da divisão que é feita da memória em segmentos, na prática, e tradicionalmente, o Unix não suporta a divisão da memória em segmentos, razão pela qual um programa é todo construído no mesmo segmento e o endereçamento é linear em vez de ser segmentado. De qualquer forma, isto não invalida que o mecanismo dos segmentos inerente ao próprio microprocessador tenha que ser preparado de forma totalmente coerente.

O 80386 suporta o modo de funcionamento que acabámos de descrever em que o endereçamento é virtual e em que todos os acessos ilegais são impossíveis, designado de protegido. Além deste, suporta o modo real, que permite executar programas escritos para os microprocessadores 8086/8088 e para os pouco conhecidos 80186/80188 [4]. Neste modo de endereçamento o processador comporta-se como se fosse um processador 8086 de grande velocidade. Existe ainda um terceiro modo de endereçamento, conhecido por modo virtual 8086. Este modo foi criado para suportar no modo protegido programas escritos para o 8086. Isto permite que estes programas possam executar-se concorrentemente com outros programas. Na verdade o modo virtual 8086 não é um verdadeiro modo, sendo apenas uma tarefa que corre em modo protegido com um determinado atributo seleccionado. Qualquer tarefa pode seleccionar esse atributo. Quando o sistema operativo comuta de contexto para uma destas tarefas o processador passa a emular um processador 8086. Para mais detalhes veja-se (IA-32 developer's manual-I, 2001; IA-32 developer's manual-II, 2001; IA-32 developer's manual-III, 2001).

[4] Além destes também podemos considerar programas escritos explicitamente para o modo real dos processadores subsequentes a começar no 80286.

3.4 Modos de endereçamento

Como se viu, para aceder às várias posições de memória podem ser utilizados registos, sendo sempre necessário indicar o segmento e o deslocamento [5].

O segmento será então indicado num dos registos de 16 *bits* concebidos para o efeito, `cs`, `ss`, `ds`, `es`, `fs` ou `gs`. Alguns destes têm funções específicas que não podem ser alteradas: é o caso dos registos `cs`, `ss` e `ds`. As funções destes e dos restantes registos selectores está resumida na tabela 3.1.

Registo	Nome	Descrição
CS	*Code Segment*	Segmento de código
SS	*Stack Segment*	Segmento da pilha
DS	*Data Segment*	Segmento de dados
ES	*Extra Segment*	Segmento de dados extra
FS	(não tem nome)	Outro segmento de dados extra
GS	(não tem nome)	Outro segmento de dados extra

Tabela 3.1: Registos de segmento ou selectores

A determinação do deslocamento dentro de um segmento normalmente será feita com um registo, desta vez de 32 *bits*, embora existam outras formas de o fazer.

Antes disso tem de ser alargado o conceito de endereçamento, de forma a incluir também a noção de acesso a um valor armazenado num registo (note-se que um registo também é memória) ou, simplesmente, um valor numérico literal.

O 80386 tem sete formas diferentes de endereçamento, que são apresentadas a seguir. Note-se que algumas não passam de variantes de outras. A convenção adoptada neste texto é a mesma que em (Murray & Pappas, 1986).

[5] Por vezes o segmento não é indicado, mas nesse caso ele é assumido implicitamente pela situação. Por exemplo, nos acessos à pilha quando não é indicado o segmento é assumida implicitamente a utilização do registo de segmento da pilha.

3.4.1 Endereçamento literal ou imediato

Este é o tipo de endereçamento mais simples e que permite atribuir directamente um valor numérico literal a um registo ou a uma célula de memória, como nos seguintes exemplos (a instrução `movl` será abordada na secção 4.2 e não é relevante para a compreensão do modo de endereçamento):

`movl $84, %eax`	**eax** $\underset{32}{\longleftarrow}$ **84**
`movb $-10, %al`	**al** $\underset{8}{\longleftarrow}$ **-10**
`movw $0x42af, numero`	**M[numero]** $\underset{16}{\longleftarrow}$ **0x42af**

Note-se a necessidade do cifrão (‘`$`’) antes do valor numérico. Na sua ausência o assemblador interpretaria os números como endereços de memória e o resultado de qualquer uma destas instruções quando executadas seria um mais que provável acesso ilegal à memória.

3.4.2 Endereçamento por Registo

Este modo de endereçamento que é também muito simples permite aceder ao valor que tenha sido armazenado previamente num registo, como nos seguintes exemplos, em que são lidos os registo `ebx`, `bx` e `bl`, por esta ordem:

`movl %ebx, %eax`	**eax** $\underset{32}{\longleftarrow}$ **ebx**
`movw %bx, %cx`	**cx** $\underset{16}{\longleftarrow}$ **bx**
`movb %bl, %al`	**al** $\underset{8}{\longleftarrow}$ **bl**

Este tipo de endereçamento também pode ser considerado directo, que será o próximo tipo de endereçamento a abordar.

3.4.3 Endereçamento Directo

Neste modo de endereçamento são usados explicitamente os endereços das células de memória. O endereço indicado contém apenas o deslocamento dentro do segmento indicado ou, em caso de omissão, pelo registo que estiver implícito na operação que será geralmente `ds`. Felizmente para o programador não é necessário definir os endereços manualmente porque o assemblador permite utilizar rótulos que depois se encarrega de converter para endereços concretos. As próprias variáveis não são mais do que rótulos com espaço reservado.

O código 3.1 e 3.2 são exemplos de endereçamento directo relativo às células de memória `meu_valor` e `meu_byte`, respectivamente.

```
movl meu_valor, %eax
```
$$\mathbf{eax} \xleftarrow[32]{} \mathbf{M[meu\_valor]} \tag{3.1}$$

```
movb meu_byte, %al
```
$$\mathbf{al} \xleftarrow[8]{} \mathbf{M[meu\_byte]} \tag{3.2}$$

Há alguns aspectos a referir. Em primeiro lugar, note-se que não é usado qualquer cifrão (‘`$`’) para aceder ao valor das variáveis; note-se também que não é indicado o segmento, mas apenas o deslocamento — já dissemos que há um segmento, o `ds` neste caso, que está implícito. Finalmente, convém que os dados tenham um tamanho adequado às instruções em que são usados: neste caso, `meu_valor` deverá ter 4 *bytes*, enquanto `meu_byte` necessita de apenas 1 *byte*.

3.4.4 Endereçamento Indirecto por Registo

Este modo de endereçamento difere do anterior, no facto de agora o endereço da célula de memória não ser indicado por um rótulo, mas por um registo. Reutilizando os exemplos do modo de endereçamento anterior e supondo que

o registo `ebx` tem armazenado o endereço da célula de memória `meu_valor`, a instrução

$$\texttt{movl (\%ebx), \%eax} \qquad \mathbf{eax} \xleftarrow[32]{} \mathbf{M[ebx]} \tag{3.3}$$

tem um resultado idêntico ao do exemplo 3.1. A forma de atribuir esse valor ao registo `ebx` será introduzida mais adiante, quando for estudada a instrução `lea`.

Antes do 80386 nem todos os registos podiam desempenhar o papel de armazenar o endereço para fazer endereçamento indirecto — apenas os registos `bx` e nalgumas circunstâncias o registo `bp`.

3.4.5 Endereçamento Relativo à Base

Este modo de endereçamento apresenta um grau de complexidade adicional, relativamente ao anterior: aqui é possível, a partir dum registo base, aceder às células e memória imediatas, quer para a frente, quer para trás. No código 3.4 o registo `eax` vai tomar o valor do inteiro armazenado quatro *bytes* à frente do endereço presente em `ebx`.

$$\texttt{movl 4(\%ebx), \%eax} \qquad \mathbf{eax} \xleftarrow[32]{} \mathbf{M[ebx + 4]} \tag{3.4}$$

Este exemplo difere do anterior porque, neste caso, `eax` iria receber o inteiro seguinte ao que está na posição indicada, de facto, por `ebx` (note-se que cada inteiro ocupa 4 *bytes*).

3.4.6 Endereçamento Indexado Directo

Neste modo de endereçamento é suposto existir um vector ou uma tabela (veja-se a subsecção 8.3.5) que tem início numa célula de memória cujo endereço é referenciado por um rótulo. A ideia é conseguir aceder a qualquer

elemento dessa tabela, utilizando o rótulo para indicar o início da tabela e um registo para indicar a posição do elemento a ler. Admitindo que se queria ler o quinto elemento da tabela de inteiros `minha_tab` para o registo `ecx` executar-se-ia, depois de guardar o valor 16 no registo `eax`, o código 3.5.

```
movl minha_tab(, %eax), %ecx
```

$$\mathbf{ecx} \xleftarrow[32]{} \mathbf{M[minha\_tab + eax]} \tag{3.5}$$

A razão de ser da vírgula antes do registo de índice será evidente na subsecção 3.4.8.

3.4.7 Endereçamento Indexado à Base

Este modo de endereçamento é semelhante ao anterior, na medida em que também existe um registo que indica a posição do elemento que se quer ler dentro dum vector ou duma tabela. A diferença é que aqui a base não é uma variável designada por um rótulo, mas um registo.

Assim, se o registo `ebx` tivesse o valor do endereço de `minha_tab`, o código 3.6 seguinte teria o mesmo efeito do código 3.5

```
movl (%ebx, %eax), %ecx
```

$$\mathbf{ecx} \xleftarrow[32]{} \mathbf{M[ebx + eax]} \tag{3.6}$$

Este tipo de endereçamento, bem como o anterior, são casos particulares do endereçamento indexado, que será visto já de seguida. Em ambos os casos é possível adicionar ao registo de índice um inteiro que indique o tamanho de cada elemento do vector (ou tabela). Isto permite que o índice represente a posição a aceder independentemente da dimensão do tipo em questão. No exemplo, que tem vindo a ser usado essa dimensão é 4, pelo que `eax` seria também 4, uma vez que se deseja aceder ao quinto elemento (i.e., à posição

4, porque o primeiro elemento está na posição 0). A forma de fazer o que está a ser dito está representada no código 3.7. Note-se que o número que indica a dimensão dos elementos do vector não é precedido de um cifrão.

```
movl $4, %eax              eax <-32- 4
movl (%ebx, %eax, 4), %ecx                    (3.7)
                           ecx <-32- M[ebx + eax × 4]
```

3.4.8 Endereçamento Indexado

Este é o modo de endereçamento mais complexo e engloba praticamente todos os outros. Permite indicar um rótulo que referencie uma tabela, um registo de base e um registo de índice, além do tamanho dos elementos da tabela, sendo o seguinte endereço calculado pela fórmula que lhe sucede:

```
rotulo + const(base, índice, tamanho)
```

```
Endereço = rotulo + const + base + índice * tamanho
```

A presença de todos estes parâmetros é opcional embora, naturalmente, não possam ser omitidos todos ao mesmo tempo. Se o parâmetro a omitir for a base utiliza-se a seguinte sintaxe, à semelhança do que já se tinha dito:

```
rotulo + const(, índice, tamanho)
```

Os exemplos 3.8, 3.9 e 3.10 pretendem ilustrar duas situações em que são utilizadas formas de endereçamento indexadas.

```
movl minha_tab(, %eax, 4), %ecx
                           ecx <-32- M[minha_tab + 4 × eax]
                                                  (3.8)
```

```
movl minha_tab(%ebx, %eax, 4), %ecx
```

$$\mathbf{ecx} \underset{32}{\longleftarrow} \mathbf{M[minha\_tab + ebx + 4 \times eax]} \tag{3.9}$$

```
movl (%ebx, %eax, 4), %ecx
```

$$\mathbf{ecx} \underset{32}{\longleftarrow} \mathbf{M[ebx + 4 \times eax]} \tag{3.10}$$

Os valores armazenados em `ecx` nos exemplos 3.8 e 3.9 estão ilustrados na figura 3.4, para exemplos de valores concretos armazenados em `eax` e `ebx`.

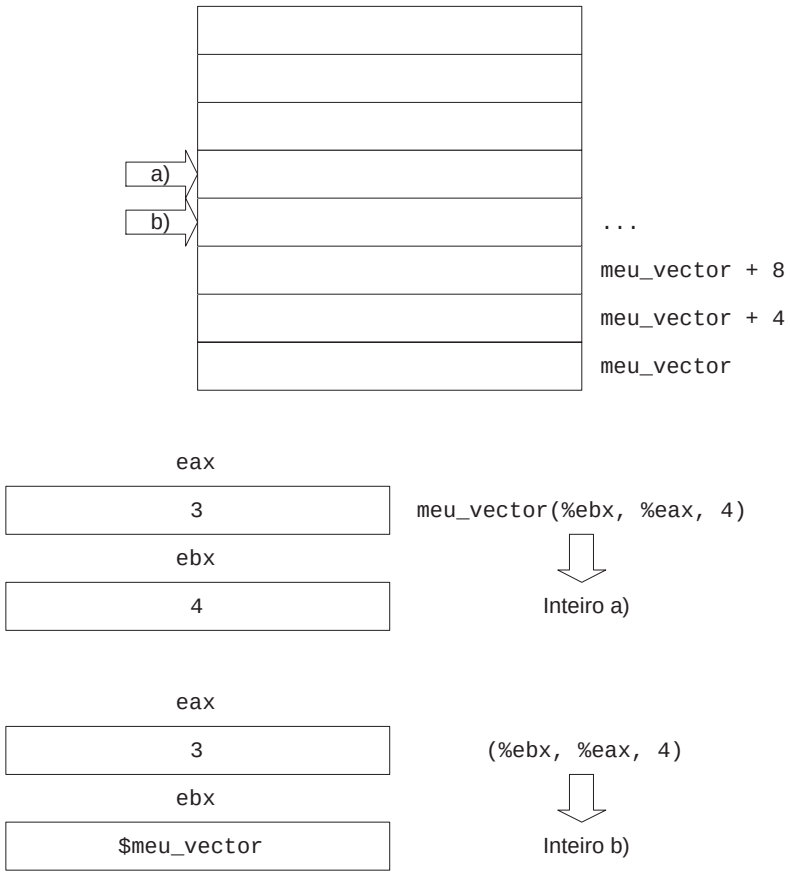

Figura 3.4: Exemplo de endereçamento indexado

3.5 Pilha

Um dos segmentos de memória que o 80386 tem capacidade suportar é o segmento da pilha. A pilha pode ser encarada como uma zona de dimensão variável, capaz de fazer o armazenamento temporário de dados.

O nome de pilha advém do facto de o acesso ser normalmente feito a partir do topo, de tal forma que novos dados deixam-se no topo e são os dados que estão no topo que saem primeiro. A esta disciplina de entradas e saídas chama-se *Last In First Out* (LIFO —- último a entrar, primeiro a sair).

As instruções básicas de manipulação da pilha chamam-se `push` e `pop` e servem, respectivamente, para pôr e tirar dados da pilha. Serão apresentadas com detalhe no capítulo 4. Estas instruções alteram o valor do registo `esp` que armazena o deslocamento do topo da pilha. Quanto ao segmento da pilha, este é armazenado em SS, como já vimos. Uma instrução `push` diminui o valor de `esp`, enquanto que a instrução `pop` aumenta o mesmo registo. Por este motivo a pilha cresce para baixo (i.e., para endereços mais baixos). É também possível descartar dados da pilha ou reservar espaço, adicionando ou subtraindo de `esp` o número desejado de *bytes*, de acordo com a figura 3.5. Note-se que o registo `esp` aponta para o último item ocupado na pilha.

Também são possíveis acessos mais complexos à pilha, utilizando para o efeito o registo `ebp`. Neste caso, o acesso à pilha faz-se praticamente como se a pilha fosse uma tabela, sendo guardado o início dessa tabela em `ebp`. Esta capacidade é especialmente aproveitada nas chamadas a funções para passar parâmetros e para guardar variáveis locais a uma função.

Consideremos como exemplo uma função que recebe três parâmetros e que utiliza duas variáveis locais (`a` e `b`) — todos com 4 *bytes*. O estado da pilha dentro desta função está representado na figura 3.6.

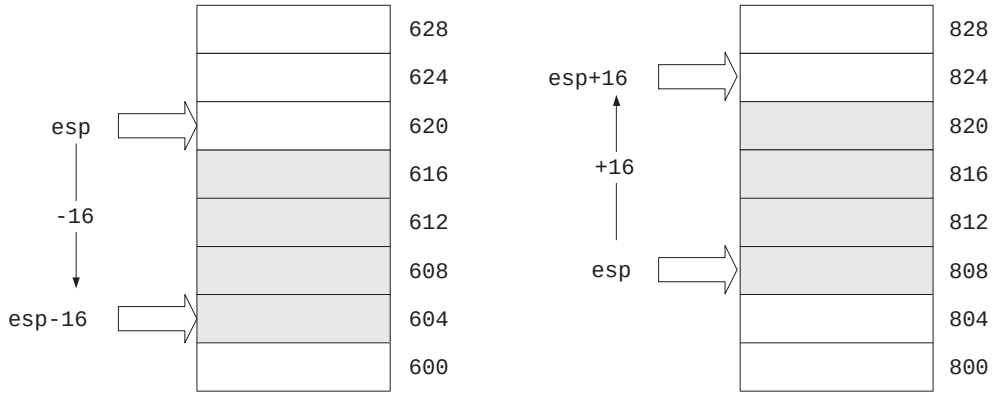

Figura 3.5: Descartar e reservar espaço na pilha

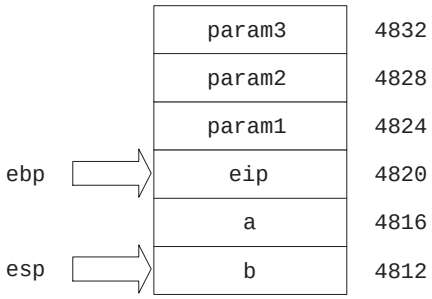

Figura 3.6: Acesso a parâmetros e variáveis locais dentro duma função

Uma forma de fazer o acesso aos parâmetros e aos dados será usando o registo `ebp` como base. Para aceder aos parâmetros utiliza-se um deslocamento ou índice positivo, enquanto que para aceder aos dados se utiliza um índice negativo.

No código 3.11 está exemplificada a forma de armazenar o parâmetro `param3` no registo `eax`, estando no código 3.12 uma forma alternativa de o fazer. No código 3.13 está exemplificado o acesso à variável `b`. É de notar que a instrução `mov` só será apresentada na secção 4.2. Funções, respectivos

parâmetros e variáveis locais serão abordados mais abaixo [6].

```
movl 12(%ebp), %eax
```
$$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{M[ebp + 12]} \tag{3.11}$$

```
movl $3, %ecx
```
$$\mathbf{ecx} \underset{32}{\longleftarrow} \mathbf{3}$$

```
movl (%ebp, %ecx, 4), %eax
```
$$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{M[ebp + 4 \times ecx]} \tag{3.12}$$

```
movl -8(%ebp), %eax
```
$$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{M[ebp - 8]} \tag{3.13}$$

[6] No entanto ainda não estão disponíveis nesta versão do texto.

4

Instruções Gerais

4.1 Introdução

Antes de mais vai ser definido o conceito de instrução. Neste contexto, instrução é uma frase, escrita na sintaxe própria do assemblador e que este traduz para código máquina. Cada instrução inclui uma mnemónica e respectivos operandos (se os houver). A cada mnemónica corresponde um número inteiro (código) segundo uma dada correspondência [1]. Uma instrução pode ter 0, 1 ou mais operandos, de acordo com o seguinte modelo (em que são apresentados dois operandos):

```
nome+sufixo operando1, operando2
```

Naturalmente que se a instrução só tiver um operando não se utiliza o segundo operando nem a vírgula, assim como se a instrução não tiver operandos indica-se apenas o nome. Podem-se adicionar mais operandos, se for caso disso, mas sempre separados por vírgulas. Note-se que algumas instruções não utilizam sufixo.

Uma questão a esclarecer é a de que, normalmente, quando é referida uma instrução em particular — `mov`, `push`, `xchg`, etc. — está a ser cometido

[1] A uma mnemónica só corresponde um código, mas o contrário não é necessariamente verdade, porque há várias mnemónicas que têm o mesmo código, i.e., correspondem todas à mesma instrução de linguagem máquina.

um pequeno abuso de linguagem. Com efeito, seria mais correcto falar em conjunto ou família de instruções, porque a cada combinação diferente de operandos pode corresponder uma instrução de máquina diferente, identificada por um código único.

Tome-se como exemplo a instrução `mov`. Cada uma das quatro instruções (sem sufixo), ilustradas no código 4.1 apresenta uma combinação diferente de operandos, que podem ser registos de vários tamanhos, uma célula de memória ou um valor literal.

`mov %eax, %ebx`	$\mathbf{ebx} \xleftarrow[32]{} \mathbf{eax}$
`mov $1500, %eax`	$\mathbf{eax} \xleftarrow[32]{} \mathbf{1500}$
`mov $25, %ax`	$\mathbf{ax} \xleftarrow[16]{} \mathbf{25}$
`mov $25, %ah`	$\mathbf{ah} \xleftarrow[8]{} \mathbf{25}$

(4.1)

Apesar de, em termos lógicos, e para o programador estas quatro instruções formarem uma única unidade, a realidade é que para o microprocessador elas são distintas e, como tal, cada uma delas necessita do seu próprio identificador. Elas podem inclusivamente nem sequer ter o mesmo tamanho. Por exemplo, para estes quatro exemplo os códigos das instruções `mov` são, pela ordem em que eles aparecem: `0x89`, `0xc7`, `0xb8` e `0xb0`.

De qualquer forma, ao longo deste texto, é referida sempre a "instrução `mov`" e não o "conjunto de instruções `mov`", mas chama-se a atenção para o facto de que isto se trata de um abuso de linguagem, do ponto de vista do processador.

É de referir que a sintaxe do assemblador que se descreve (Gas) ajuda a fazer esta distinção entre instruções pertencentes ao mesmo conjunto, já que permite alterar o nome da instrução de forma a identificar a dimensão dos operandos envolvidos, através de um sufixo '`b`', '`w`' ou '`l`' (*byte* - 8 *bits*, *word* - 16 *bits* ou *longo* - 32 *bits*) como se poderá ver pelas instruções atrás apre-

sentadas. Esta distinção não é obrigatória, podendo ser omitida sempre que for possível ao assemblador deduzir implicitamente a dimensão da operação. Na instrução `mov $150, %eax`, por exemplo, é possível saber que se trata de uma instrução de 32 *bits*, pelo que não é obrigatório escrever `movl`. Para o programador, também aqui, a regra é clara: tornar o código o mais legível possível. Por este motivo é aconselhável a utilização do sufixo que indica a dimensão da operação.

Quanto aos operandos das instruções a apresentar, estes receberão nomes diferentes conforme a situação em que venham a ser utilizados e, em particular, conforme as restrições que se aplicam à instrução em causa. Assim, a tabela 4.1 resume todos os tipos de operandos existentes.

Mnemónica	Significado
`dst`, `reg-mem`	Operando que aparece frequentemente à esquerda do sinal de atribuição '⟵'. Não poderá ser um literal. Designado por valor esquerdo ou destino. Poderá ser um registo ou uma célula de memória. Por este motivo tem a designação alternativa de `reg-mem`. Conforme o contexto será utilizada uma das designações.
`org`, `op`	Operando que aparece à direita do sinal de atribuição '⟵'. Designado por valor direito ou origem. Poderá ser um operando qualquer, inclusive um literal. Por este motivo, também poderá ter a designação alternativa de `op`. Conforme o contexto será utilizada uma das designações.
`reg`	Operando tem que forçosamente ser um registo.
`mem`	Operando tem forçosamente que referir-se a um endereço duma célula de memória.
`literal`	Operando só poderá ser um literal.

Tabela 4.1: Tipos de operandos

Além dos tipos de operandos que aparecem na tabela é ainda necessário ter em conta o conjunto de restrições que se aplicam a cada um deles, que

serão necessariamente enumeradas à medida que forem sendo apresentadas as instruções. Sobre os operandos `dst` e `org` há que referir que estes operandos são intrinsecamente diferentes, como se poderá constatar na seguinte expressão:

`dst` ⟵ `org`

É evidente que o operando `dst` nunca poderá ser um literal, enquanto o operando `org` pode perfeitamente ser um literal (o número '5' por exemplo). Nem todas as instruções têm obrigatoriamente um operando origem e um destino: é o caso da instrução `xchg`, que será apresentada na secção 4.3, que não usa um operando origem, sendo ambos operandos destino.

Outra restrição que se aplica sistematicamente diz respeito ao registo `eip`. A utilização directa deste registo é pura e simplesmente impossível. As formas de o alterar serão sempre indirectas como veremos mais adiante.

4.2 Instrução `mov`

`mov org, dst` `dst` ⟵ `org`

Instrução de atribuição.

A instrução `mov` é possivelmente, e de entre todas as que existem na família de processadores 80x86, a mais utilizada.

Esta instrução inclui sempre dois operandos e destina-se a atribuir o valor do operando `org` ao operando `dst`. O operando `org` será, assim, a origem que não é afectada pela operação e `dst` o destino, que no fim da instrução toma o valor do operando `org`, destruindo irremediavelmente qualquer valor que lá estivesse armazenado previamente. Trata-se, portanto, duma instrução de atribuição.

Os operandos `org` e `dst` podem ser diversos, embora com algumas limitações. Na tabela 4.1 estão ilustradas as várias formas possíveis para os dois operandos e na tabela 4.2 as respectivas restrições. É de notar que a posição em que aparecem os operandos na tabela é relevante. Assim, quando numa determinada linha aparece um operando sozinho isto significa que este operando não pode ser utilizado. É o caso do registo `eip` — quer como operando destino, quer como operando origem. Quando aparecem dois operandos na mesma linha, isso significa que essa combinação não existe. Por exemplo, o operando destino e o operando origem não podem ser endereços de células de memória em simultâneo.

Note-se também que conforme a dimensão dos operandos haverá diferentes instruções `mov`: `movb` (8 *bits*), `movw` (16 *bits*) e `movl` (32 *bits*). Como é natural, só é possível fazer atribuições entre operandos da mesma dimensão e é em função desta dimensão que se aplica uma das várias instruções `mov` existentes.

Valor direito (`org`)	Valor esquerdo (`dst`)
célula de memória	célula de memória
registo `eip`/`ip`	
	registo `eip`/`ip`
literal	registo de segmento
registo de segmento	registo de segmento
	registo `cs`

Tabela 4.2: Restrições aos operandos na instrução `mov`

De seguida são apresentados alguns exemplos ilustrados no código 4.2. Os dois primeiros exemplos são óbvios e estão comentados.

`movl $25, %eax`	**eax** $\underset{32}{\longleftarrow}$ **25**
`movb %ah, %al`	**al** $\underset{8}{\longleftarrow}$ **ah**
`movl $numero, %eax`	**eax** $\underset{32}{\longleftarrow}$ **numero**
`movl numero, %eax`	**eax** $\underset{32}{\longleftarrow}$ **M[numero]**

(4.2)

A diferença entre os dois últimos exemplos é subtil, mas importante. Considere-se a situação da figura 4.1, onde existe uma célula de memória com 32 *bits* (na realidade são quatro posições de memória diferentes) designada por `numero`. O endereço do primeiro *byte* desta célula é 5000 e o seu conteúdo é 20.

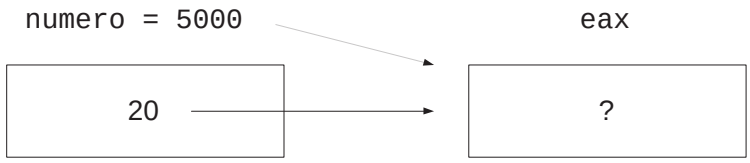

Figura 4.1: Valor armazenado em memória, no endereço designado por `numero`

Após a instrução `movl numero, %eax`, o registo `eax` vai valer 20. Se, porém, se fizer `movl $numero, %eax`, o registo `eax` ficará com o valor 5000.

4.3 Instrução `xchg`

`xchg dst1, dst2`	`dst2` $\longleftrightarrow$ `dst1`

Troca o conteúdo de dois operandos.

A instrução `xchg` (*exchange* — que em inglês significa "trocar") recebe dois operandos e permite trocar o seu conteúdo.

Ao contrário do que sucede na instrução `mov`, aqui não existe o conceito de operando origem nem destino, uma vez que ambos são, em simultâneo,

origem e destino. Seguem-se no código 4.3 alguns exemplos de utilização desta instrução.

```
xchgb valor, %al        al <--8--> M[valor]
xchgw %ax, %si          si <--16--> ax          (4.3)
xchgl %eax, %esi        esi <--32--> eax
```

As duas primeiras restrição são óbvias, estando a primeira já implícita na convenção adoptada neste texto para os operandos destino: nesta instrução nenhum dos operandos pode ser um valor literal. A outra restrição tem a ver com os tamanhos dos operandos que têm de ser iguais. Também não é possível trocar directamente duas células de memória, nem trocar um registo de segmento com qualquer outro registo. Aplicam-se, além destas, todas as outras restrições da tabela 4.3.

Operandos da instrução `xchg`
literal
registo `eip/ip`
registo de segmento

Tabela 4.3: Restrições aos operandos na instrução `xchg`

4.4 Instrução `nop`

`nop` **(nenhuma operação)**

Instrução *No operation*.

Esta instrução não faz nada e não tem qualquer operando. Porém, pode apresentar alguma utilidade em determinadas circunstâncias relativas a problemas de alinhamento e à depuração de erros. Veja-se (Hahn, 1992).

4.5 Instrução `lea`

`lea mem, reg` $\qquad\qquad$ $\mathbf{reg} \underset{32}{\longleftarrow} \mathbf{mem}$

Carrega o endereço efectivo para registo de 32 *bits*.

A instrução `lea` (*Load Effective Address Offset* — que em inglês significa "Carrega Deslocamento do Endereço Efectivo", numa tradução à letra) permite atribuir a um registo indicado em `reg` o deslocamento de memória representado pelo operando `mem` (ver figura 3.3). Este operando tanto pode ser o nome duma variável, como aparece no exemplo de código 4.4, como poderá ser uma expressão mais complexa que indexa uma tabela em memória, como nos exemplos 4.5 e 4.6, respectivamente.

```
leaw valor, %ax #Só para 80286
```
$$\mathbf{ax} \underset{16}{\longleftarrow} \mathbf{valor} \tag{4.4}$$

```
movl $16, %eax leal tabela(,%eax), %ecx
```
$$\mathbf{ecx} \underset{32}{\longleftarrow} \mathbf{tabela} + \mathbf{eax} \tag{4.5}$$

```
leal tabela, %ebx
```
$\mathbf{ebx} \underset{32}{\longleftarrow} \mathbf{tabela}$

```
movl $4, %eax
```
$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{4}$

```
leal (%ebx, %eax, 4), %ecx
```
$$\mathbf{ecx} \underset{32}{\longleftarrow} \mathbf{ebx} + \mathbf{eax} * \mathbf{4} \tag{4.6}$$

Como acontece em muitas instruções pode ser adicionado um sufixo para indicar a dimensão da operação, que neste caso será sempre de 16 ou 32 *bits* ('`w`' ou '`l`', respectivamente), uma vez que o valor a armazenar em `op2` será sempre um endereço [2].

[2] No 80286 a dimensão era de 16 *bits*, enquanto que no 80386 essa dimensão é de 32.

No exemplo 4.4 o registo `ax` toma o valor do deslocamento presente na variável `valor` (80286).

Os exemplo 4.5 e 4.6 são mais complexos e iguais entre si, sendo que o deslocamento efectivamente carregado no registo `ecx` é o endereço onde começa a quinta posição do vector de inteiros `tabela`, tal como está representado na figura 4.2.

Note-se que, enquanto o primeiro exemplo é equivalente à instrução `movl $valor, %ax`, não é possível só com instruções `mov` substituir as instruções dos exemplos subsequentes, embora tal fosse possível e relativamente trivial de fazer com uma combinação das instruções `mov` e `add`, a apresentar mais adiante, embora com claro prejuízo de desempenho.

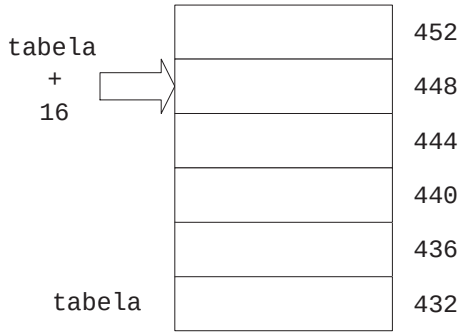

Figura 4.2: Resultado da instrução `lea`

As restrições a aplicar à instrução `lea` são as que estão implícitas nos tipos de operandos, além das representadas na tabela 4.4.

`reg`
registo `eip/ip`
registo de segmento
registo de 8 *bits*

Tabela 4.4: Restrições aos operandos na instrução `lea`

Como se verá na secção 6.2 a instrução `lea` pode ser usada para efectuar adições.

4.6 Instruções `lds`, `les`, `lfs`, `lgs` e `lss`

`lds mem, reg`

$$\mathbf{reg} \underset{32}{\longleftarrow} \mathbf{M[mem]}$$
$$\mathbf{ds} \underset{16}{\longleftarrow} \mathbf{M[mem + 4]}$$

Carrega endereço efectivo para registo de segmento `ds` e para registo de 32 *bits*.

Nesta secção serão apresentadas várias instruções, que apesar de algumas diferenças, são muito semelhantes entre si. Por este motivo, sempre que for necessário concretizar alguma explicação, vai ser utilizada a instrução `lds`, não sendo, por regra, difícil generalizar essas mesmas explicações às restantes instruções.

O operando `mem` deverá ser o endereço duma célula de memória; `reg` um registo. As restrições que afectam os operandos são as mesmas que afectavam a instrução `lea`. Note-se que não há restrições ao operando que referencia uma célula de memória (além da restrição que este facto constitui). As restrições estão, então, representadas na tabela 4.4.

`lds`, que significa "Load Data Segment"(Carrega Segmento de Dados) tem um duplo efeito: altera o registo especificado em `reg` e altera o registo do segmento de dados `ds`. Para isso a partir da célula de memória especificada em `mem` deverá estar um endereço válido de 48 *bits* (segmento + deslocamento), sendo atribuídos a `reg` os primeiros 32 *bits* relativos ao deslocamento e a `ds` os últimos 16 *bits* relativos ao segmento. A figura 4.3 procura ilustrar o funcionamento desta instrução, em particular como é que se representa um ponteiro completo em memória.

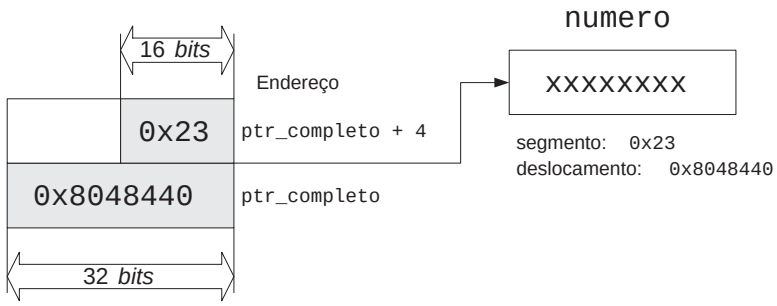

Figura 4.3: Ponteiro de 48 *bits* para a instrução `lds`

No fim da instrução

```
lds ptr_completo, %ebx
```

$$\mathbf{ebx} \xleftarrow[32]{} \mathbf{M[ptr\_completo]}$$

$$\mathbf{ds} \xleftarrow[16]{} \mathbf{M[ptr\_completo + 4]}$$

o registo `ebx` toma o valor do endereço da variável `numero` (0x8048440) e o registo de segmento `ds` o valor do segmento 0x23 (onde se assume estar a variável `numero`).

As instruções `les` ("Load Extended Data Segment-– Carrega Segmento de Dados Estendido), `lfs`, `lgs` e `lss` são em tudo equivalentes, com a diferença de que o registo de segmento afectado é, respectivamente, e como o nome indica `es`, `fs`, `gs`, e `ss`.

Estas instruções são pouco úteis no Unix, porque um programa utiliza apenas um único segmento. Por este motivo, não há necessidade de alterar os registos de segmento.

É importante referir que uma tentativa de carregar um valor inválido num registo de segmento resulta na geração duma excepção que termina com um envio do sinal `SIGSEGV` ao processo que faz essa tentativa (a que se segue, em geral, um *coredump*). Isto significa que o operando `mem` tem que se referir a uma zona de memória correctamente preparada para a instrução `lds` ou

para uma das suas congéneres. O código 4.7 mostra como poderia ser feita a inicialização conducente à situação da figura 4.3, seguido da instrução `lds` propriamente dita.

```
movl $numero, ptr_completo
                          M[ptr_completo] <--32-- numero
movw %ds, %ax             ax <--16-- ds
movw %ax, ptr_completo + 4
                          M[ptr_completo + 4] <--16-- ax
lds ptr_completo, %ebx    ebx <--32-- M[ptr_completo]
                          ds <--16-- M[ptr_completo + 4]
```

(4.7)

Usando também directivas de definição de dados (a apresentar no capítulo 8), teríamos a situação ilustrada no código 4.8.

```
numero:  .int 1225 #Um valor qualquer
ptr_completo:  .long $numero
                          M[ptr_completo] <--32-- numero
  movw %ds, %ax           ax <--16-- ds
movw %ax, ptr_completo + 4
                          M[ptr_completo + 4] <--32-- ax
lds ptr_completo, %ebx    ebx <--32-- M[ptr_completo]
                          ds <--16-- M[ptr_completo + 4]
```

(4.8)

4.7 Instruções push e pop

`push op`	**esp** $\underset{32}{\longleftarrow}$ **esp - 4**
	M[esp] $\underset{32}{\longleftarrow}$ **op**
`pop reg-mem`	**op** $\underset{32}{\longleftarrow}$ **M[esp]**
	esp $\underset{32}{\longleftarrow}$ **esp + 4**

Instrução `push`: carrega dados de operando para a pilha. Instrução `pop`: descarrega dados da pilha para operando.

As instruções `push` e `pop` admitem um único operando. A instrução `push` armazena o valor do operando `op` na pilha, enquanto que a operação `pop` faz precisamente o contrário, i.e., retira um valor da pilha e armazena-o no operando `reg-mem`.

O armazenamento dum valor qualquer na pilha, através da instrução `push`, é feito em dois passos:

- em primeiro lugar, o registo que indica onde está o topo da pilha (`esp`) é actualizado (decrementado) de forma a reservar espaço para o novo valor;
- depois, o valor passado no operando `op` é copiado para a zona de memória apontada pelo deslocamento `esp` (relativamente ao início do segmento `ss` — *stack segment*).

Em termos de instruções, estes dois passos correspondem ao código 4.9. No código assume-se que o operando `op` tem a dimensão de quatro *bytes*. Caso fosse apenas de dois a primeira instrução decrementaria apenas duas unidades ao registo `esp` e a segunda seria relativa a uma palavra (`movw`).

`subl $4, %esp`	**esp** $\underset{32}{\longleftarrow}$ **esp - 4**	(4.9)
`movl op, (%esp)`	**M[esp]** $\underset{32}{\longleftarrow}$ **op**	

A instrução `pop` faz precisamente o contrário, i.e., retira primeiro o valor da pilha e, depois, incrementa o registo `esp` no número de unidades necessárias. O funcionamento das instruções `push` e `pop` com 32 *bits* encontra-se representado nas figuras 4.4 e 4.5. As setas cinzentas indicam a posição final do topo da pilha (registo `esp`).

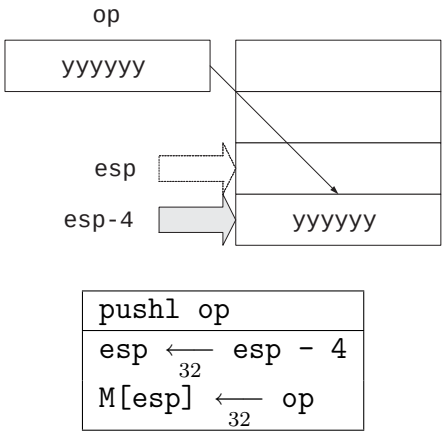

Figura 4.4: Instrução `push`

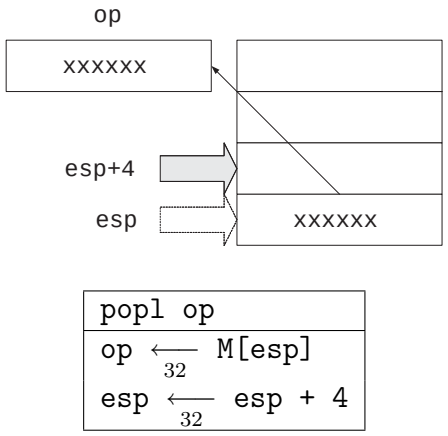

Figura 4.5: Instrução `pop`

Como se poderá inferir da descrição feita, a pilha cresce dos endereços mais altos para os mais baixos, sendo normal dizer-se, por este motivo, que a pilha cresce para baixo. Note-se que isto não se trata de nenhuma convenção, mas de um facto concreto inerente ao funcionamento dos microprocessadores da família 80x86, onde o registo `esp` é decrementado pela instrução `push` e incrementado pela instrução `pop`.

Embora à primeira vista possa não ser muito evidente a utilidade de ter uma pilha e de usar estas instruções, à medida que forem apresentados alguns exemplos de programas reais, essa utilidade vai tornar-se cada vez mais evidente. Para já fica registado que a pilha serve para os seguintes fins (o segundo e o terceiro itens desta lista não são evidentes para já):

- armazenar valores temporários sem ser necessário recorrer a variáveis

ou a registos; esta capacidade é tanto mais útil quanto menos registos houver na máquina;

- armazenar o endereço de retorno duma função;
- passar parâmetros ou receber resultados duma função.

Os operandos admitidos pelas duas instruções podem ter 16 ou 32 *bits*; podem ser registos (qualquer um) ou células de memória. Como sempre, quando os operandos têm 16 *bits* usa-se o sufixo '`w`', quando têm 32 *bits*, usa-se o sufixo '`l`'.

Como exemplo, vamos pensar num caso em que fosse necessário trocar o conteúdo do registo `ax` com o `es`. Infelizmente não é possível efectuar a troca directamente com a instrução `xchg`, pelo que poderíamos, recorrendo à pilha, usar a solução apresentada no código 4.10.

```
pushw %ax
pushw %es
popw %ax
popw %es
```
(4.10)

Este exemplo ilustra também o facto de os elementos serem retirados da pilha pela ordem inversa em que aí são colocados (neste caso como esse requisito não é cumprido os registos acabam trocados) — daí o nome de pilha que foi dado a esta estrutura de dados.

Não pode ser omitida aqui uma informação de importância essencial: para que o programa funcione e, em particular, termine correctamente é necessário que se retirem todos os objectos colocados na pilha. Também não podem ser retirados objectos a mais. Isto significa que o registo `esp` antes da última instrução do programa (`ret`, como teremos ocasião de ver) deve ser precisamente igual ao que era no início.

Outro aspecto importante tem a ver com a localização da pilha: quando um programa arranca tem já forçosamente que incluir uma zona de memória reservada para esse efeito.

4.8 Instruções `pusha` e `popa`

`pusha`	$\mathbf{M[esp - 4]} \underset{32}{\longleftarrow} \mathbf{eax}$
	$\mathbf{M[esp - 8]} \underset{32}{\longleftarrow} \mathbf{ecx}$
	$\mathbf{M[esp - 12]} \underset{32}{\longleftarrow} \mathbf{edx}$
	...
	$\mathbf{M[esp - 32]} \underset{32}{\longleftarrow} \mathbf{edi}$
	$\mathbf{esp} \underset{32}{\longleftarrow} \mathbf{esp - 32}$
`popa`	$\mathbf{edi} \underset{32}{\longleftarrow} \mathbf{M[esp]}$
	$\mathbf{esi} \underset{32}{\longleftarrow} \mathbf{M[esp + 4]}$
	$\mathbf{ebp} \underset{32}{\longleftarrow} \mathbf{M[esp + 8]}$
	$\mathbf{ebx} \underset{32}{\longleftarrow} \mathbf{M[esp + 16]}$
	...
	$\mathbf{ebp} \underset{32}{\longleftarrow} \mathbf{M[esp + 28]}$
	$\mathbf{esp} \underset{32}{\longleftarrow} \mathbf{esp + 32}$

Instrução `pusha`: carrega todos os registos para a pilha. Instrução `popa`: descarrega todos os registos da pilha.

As instruções `pusha` e `popa` são invocadas sem qualquer parâmetro. Respectivamente, guardam e retiram os seguintes registos da pilha, pela ordem em que são aqui enumerados: `eax`, `ecx`, `edx`, `ebx`, `esp` original, `ebp`, `esi` e `edi`. Da instrução `pusha` resulta um decremento de 32 *bytes* no registo `esp`, enquanto que da instrução recíproca, `popa`, resulta um incremento de

32 *bytes* no mesmo registo. A instrução `popa` não altera o registo `esp`, sendo o valor armazenado na pilha descartado.

Estas instruções podem ser úteis numa situação em que se pretenda salvaguardar o valor de todos os registos, para os recuperar mais tarde. Será o caso duma função, por exemplo, que utilize os registos no seu código, mas que lhes reponha o valor original antes de terminar.

4.9 Instruções `pushf` e `popf`

`pushf`	$\mathbf{esp} \xleftarrow[32]{} \mathbf{esp - 4}$
	$\mathbf{M[esp]} \xleftarrow[32]{} \mathbf{eflags}$
`popf`	$\mathbf{eflags} \xleftarrow[32]{} \mathbf{M[esp]}$
	$\mathbf{esp} \xleftarrow[32]{} \mathbf{esp + 4}$

Instrução `pushf`: carrega o registo `eflags` na pilha. Instrução `popf`: descarrega o registo `eflags` da pilha.

`pushf` e `popf` não têm operandos. Respectivamente, guardam e retiram o registo das *flags*, `eflags`, da pilha.

5

Instruções de Controlo de Fluxo

5.1 Introdução

No capítulo 4 foi apresentado um conjunto de instruções que serão utilizadas com muita frequência. No entanto, não é possível escrever um programa dispondo apenas deste conjunto. Com efeito, nenhuma destas instruções permite alterar o fluxo de execução do programa. Quer isto dizer que a sequência de instruções executadas permanece inalterada, independentemente de quaisquer entradas externas ao programa. Naturalmente que esta situação não é razoável e qualquer microprocessador tem que incluir um conjunto adicional de instruções, chamadas de "controlo de fluxo". Grosso modo, estas instruções permitem que em determinados pontos dum programa seja possível optar por uma sequência de instruções ou por outra.

5.2 Instrução `jmp`

`jmp mem` $\qquad\qquad$ $\mathbf{eip} \underset{32}{\longleftarrow} \mathbf{mem}$

Salto incondicional.

De entre as instruções de controlo de fluxo, a instrução `jmp` (*jump* — salto) é a mais simples de todas. Posto de uma forma simples, o resultado da

instrução `jmp` equivale a alterar o valor do registo `eip` com uma instrução `mov` (embora isso não seja possível de fazer), tal como se encontra neste exemplo de código:

$$\begin{array}{ll} \texttt{movl \$0x08400000, \%eip \#impossível} & \\ & \textbf{eip} \underset{32}{\longleftarrow} \textbf{0x08400000} \\ \texttt{jmp \$0x08400000 \#certo} & \textbf{eip} \underset{32}{\longleftarrow} \textbf{0x08400000} \end{array} \tag{5.1}$$

Como o registo `eip` guarda o endereço da próxima instrução a executar, o resultado prático de alterar o valor deste registo traduz-se num salto na ordem pela qual são executadas as instruções.

Antes de se passar a um exemplo concreto, tem que ser introduzido o conceito de *rótulo*, que se apresentará de seguida. A esta instrução não se aplica qualquer restrição para além do facto de ser necessário que o operando tenha 32 *bits*. Desta forma, o operando poderá ser um literal, um registo ou um deslocamento de memória.

5.3 Rótulos

A realização de código com o aspecto apresentado no exemplo 5.1 não é muito prática. Não é razoável exigir ao programador que saiba em que endereço é que se encontra cada uma das instruções do código para onde pretende fazer um salto no fluxo de execução (com a instrução `jmp`) [1].

Para resolver esse problema, existem os rótulos, que permitem identificar, através dum nome inteligível para o programador, endereços concretos

[1] Aliás, o caso não é exactamente este, porque só numa fase que se sucede à assemblagem — essa fase é a edição de ligações, como se verá — é que fica determinado o endereço onde ficarão localizadas as partes dum programa, incluindo o código executável. Portanto, quando muito seria possível fazer referências relativas ao início da secção de código.

de partes do programa. No caso presente interessa-nos que os rótulos identifiquem o início duma instrução para onde se deseja saltar.

Um rótulo é um símbolo seguido por dois pontos (‘:’). O nome de um símbolo pode começar por qualquer letra, pelo ponto ou pela barra inferior (‘.’ e ‘_’, respectivamente). Os restantes caracteres, que podem ser em número arbitrário, podem ser também algarismos. O cifrão (‘$’) pode ser incluído no nome de um rótulo. Nos nomes assim definidos a distinção entre maiúsculas e minúsculas é relevante, sendo o rótulo `Lixo` diferente de `lixo`, por exemplo.

É essencial compreender que o conceito de rótulo existe num nível acima do microprocessador e que um rótulo não tem (como deverá ser evidente) qualquer tradução em termos de instruções de código máquina. Quando o microprocessador executa o código a que no programa assembly corresponde `jmp nome_do_rotulo`, o que encontra de facto no lugar do rótulo é um endereço, i.e., `jmp 0x08000000`.

Podem-se então apresentar dois exemplos de utilização da instrução `jmp`. No primeiro, representado no código 5.2 a instrução que se segue ao `jmp` nunca é executada; no segundo, ilustrado no código 5.3, os saltos são feitos a partir de endereçamentos indexados. Neste caso, o endereço de destino do salto vai ser encontrado algures numa célula de memória. Este é um caso interessante em que o destino do salto não é conhecido à partida.

```
    movl $20, %eax          eax <-32- 20
    jmp ignora_ebx          eip <-32- ignora_ebx
    movl %eax, %ebx         ebx <-32- eax                    (5.2)
ignora_ebx:
    movl %eax, %ecx         ecx <-32- eax       ...
```

$$\begin{array}{ll} \texttt{jmp tabela(, \%esi)} & \mathbf{eip} \underset{32}{\longleftarrow} \mathbf{M[tabela + esi]} \\ \ldots & \\ \texttt{jmp (\%ebx, \%ecx, 1)} & \mathbf{eip} \underset{32}{\longleftarrow} \mathbf{M[ebx + 1 \times ecx]} \end{array} \tag{5.3}$$

5.4 Saltos Condicionais — Instrução `jcc`

`jcc mem` $\qquad\qquad$ **cc ? eip** $\underset{32}{\longleftarrow}$ **eip + mem**

Salto condicional.

Mesmo depois de apresentada a instrução `jmp` muitas das limitações que descrevemos no início deste capítulo continuam a aplicar-se, uma vez que a instrução `jmp` permite fazer, apenas, saltos incondicionais [2].

Para realizar saltos condicionais, existem as instruções `jcc`. Ao '`j`' é acrescentado um sufixo, que varia consoante o teste que deverá ser efectuado pela instrução (representado por `cc`). Ao `j` e ao sufixo, que compõem o nome da instrução, acresce um operando, de acordo com a forma atrás apresentada. O operando da instrução `jcc` deverá ser um número inteiro (positivo ou negativo), que representa a distância relativamente à posição actual do registo `eip`. Na prática, e para simplificar a programação, é possível utilizar um deslocamento de memória representado por um rótulo, em vez de ser necessário indicar a distância, cabendo ao assemblador fazer as contas que forem necessárias.

No código 5.4, é utilizado o rótulo `ptoA`, sendo sempre o assemblador responsável por determinar a distância em termos de *bytes*, desde o ponto da instrução `jcc`, que faz a referência ao rótulo `ptoA` até esse mesmo rótulo.

[2] Isto não significa, no entanto, que o destino do salto seja sempre o mesmo. Veja-se por exemplo o código 5.3.

```
    jz ptoA              ZF == 1 ? eip <--(32)-- eip + 2
    movl %eax, %ebx      ebx <--(32)-- eax
ptoA:
    movl %eax, %ecx      ecx <--(32)-- eax
```
(5.4)

Ao contrário do que sucedia com o `JMP`, aqui o operando não indica um endereço absoluto, mas um endereço relativo à instrução actual. Isto significa que se o teste, que a própria instrução efectua, for verdadeiro vai ser adicionado o valor do operando ao registo `eip`. Caso contrário, o fluxo de execução prossegue na instrução imediatamente a seguir.

Também aqui, e felizmente para o programador, não é necessário saber a localização absoluta, nem tão pouco relativa, das instruções. Continua a ser possível usar rótulos, cabendo ao assemblador fazer o cálculo das distâncias a usar na linguagem máquina.

Quase todas as instruções `jcc` tem duas representações diferentes em código máquina (uma excepção é a instrução `jcxz`). Isto tem a ver com uma optimização que é possível fazer caso o deslocamento relativo esteja contido no intervalo $[-128, 127]$. Neste caso bastam apenas 8 *bits* para representar o deslocamento sendo, então, possível representar toda a instrução com apenas 2 *bytes* (um *byte* para o código da instrução outro para o deslocamento). Quando o deslocamento relativo sai fora desta gama, passam a ser usados deslocamentos de 32 *bits*, que acrescentados aos 2 *bytes* para o código da instrução, totalizam 6 *bytes* por instrução. Isto é o triplo da situação anterior. Por este motivo, o assemblador deve ter o cuidado de verificar se o deslocamento pode ser representado com apenas 1 *byte*.

Na tabela 5.1 é apresentada uma lista completa das instruções de salto condicional. Nesta tabela encontram-se representadas também as *flags* que são verificadas e o valor que devem assumir para que cada um dos saltos

condicionais se verifique.

Naturalmente que antes de o código atingir uma instrução de salto condicional `jcc` o programador tem que saber exactamente o valor que as *flags* assumem em cada situação possível e que *flags* são verificadas pelo salto condicional, sob pena de o programa se comportar duma forma inesperada. Para facilitar esta tarefa existe uma operação que permite manipular as *flags* e que será apresentada de seguida. A utilização desta instrução vai permitir utilizar os saltos condicionais `jcc` duma forma muito mais intuitiva, ao ponto de não ser necessário ao programador conhecer exactamente quais as *flags* avaliadas por cada instrução de salto condicional.

5.5 Instrução `cmp`

`cmp org, dst` **eflags** ←-- `dst − org`

Compara dois operandos.

Esta operação efectua a subtracção `dst- org`, afectando as *flags* em função do resultado, mas sem afectar qualquer dos operandos. A ideia é que, depois de um `cmp`, venha um salto condicional, cujo resultado como se viu depende de uma ou mais *flags*.

As restrições que se aplicam aos operandos da instrução `cmp` são as mesmas que se aplicavam à instrução `mov` e estão descritas na tabela 4.2. Além destas restrições, aplica-se uma outra que diz que nenhum dos operandos poderá ser um registo de segmento.

Para que se compreenda melhor a utilização desta instrução, é apresentado de seguida um exemplo, que corresponde à estrutura de decisão `if-else` habitualmente existente nas linguagens de alto nível. No exemplo apresentado verifica-se se o valor presente no registo `eax` é positivo. Se for, é efec-

tuada uma adição, caso contrário uma subtracção:

```
    cmpl $0, %eax          eflags <-- eax - 0
    jg L1                  se (ZF == 0 && SF == OF) então eip <-(32)- eip + x1
    jmp L2                 eip <-(32)- eip + x2
L1:
    #efectua a adição
    jmp L3                 eip <-(32)- eip + x3
L2:
    #efectua a subtracção
L3:
```

(5.5)

5.6 Instruções `loop` e `loopcc`

`loop rotulo`	**ecx** $\overset{}{\underset{\not F}{\longleftarrow}}$ **ecx - 1**
	ecx != 0 ? eip $\longleftarrow$ **eip + yy**
`loopz rotulo`	**ecx** $\overset{}{\underset{\not F}{\longleftarrow}}$ **ecx - 1**
	ecx != 0 && ZF == 1 ? eip $\longleftarrow$ **eip + yy**
`loope rotulo`	**ecx** $\overset{}{\underset{\not F}{\longleftarrow}}$ **ecx - 1**
	ecx != 0 && ZF == 1 ? eip $\longleftarrow$ **eip + yy**
`loopnz rotulo`	**ecx** $\overset{}{\underset{\not F}{\longleftarrow}}$ **ecx - 1**
	ecx != 0 && ZF == 0 ? eip $\longleftarrow$ **eip + yy**
`loopne rotulo`	**ecx** $\overset{}{\underset{\not F}{\longleftarrow}}$ **ecx - 1**
	ecx != 0 && ZF == 0 ? eip $\longleftarrow$ **eip + yy**

Instrução `loop`: usa o registo `ecx` para controlar um ciclo. Instruções `loopcc`: usam o registo `ecx` e a flag `ZF` para controlar um ciclo.

A instrução `loop` admite apenas um operando com significado idêntico ao da instrução `jcc`: distância do salto a efectuar, sendo que esta distância é relativa à instrução actual e é representada apenas por um *byte*. No caso da instrução `loop` não existem deslocamentos relativos de 32 *bits*. Por este motivo, a distância do salto está limitada aos 128 *bytes* anteriores e aos 127 *bytes* posteriores à instrução seguinte ao salto. O seu modo de operação é o seguinte: decrementa o registo `ecx` em uma unidade (sem afectar qualquer *flag*); se `ecx` $\neq 0$ salta para o ponto indicado pelo operando (a partir do valor corrente do registo `eip`), senão continua normalmente na instrução seguinte.

Considere-se como exemplo a seguinte situação: existe uma tabela com dez números inteiros em memória e o objectivo é adicionar esses dez números. Uma forma de resolver o problema seria criando um ciclo controlado por um par de instruções `cmp` e `jcc`. Em vez disso, poderemos utilizar a instrução `loop`, tal como aparece no exemplo de código 5.6.

```
movl $10, %ecx          ecx <--(32) 10
movl $0, %eax           eax <--(32) 0
ciclo:
addl tabela - 4(, %ecx, 4), %eax
                        eax <--(32) M[tabela - 4 + 4 × ecx]
loop ciclo              ecx <--(F) ecx - 1
                        ecx != 0 ? eip <-- eip + yy
```

(5.6)

A instrução `loop` apresenta duas variantes, que além de decrementarem e testarem o valor de `ecx`, verificam também o valor da *flag* `ZF`:

- `loope/loopz`: salta se `ecx != 0` e `ZF == 1`;
- `loopne/loopnz`: salta se `ecx != 0` e `ZF == 0`.

Estas versões do `loop` permitem utilizar a instrução `cmp` antes (ou qualquer outra que afecte a *flag* `ZF`), o que poderá constituir uma vantagem, sempre que se pretender uma conjugação de condições.

5.7 Instruções `std`, `cld`, `sti`, `cli`, `stc`, `clc`, `cmc`

`std`	$\mathbf{DF} \underset{1}{\longleftarrow} \mathbf{1}$
`cld`	$\mathbf{DF} \underset{1}{\longleftarrow} \mathbf{0}$
`stc`	$\mathbf{CF} \underset{1}{\longleftarrow} \mathbf{1}$
`clc`	$\mathbf{CF} \underset{1}{\longleftarrow} \mathbf{0}$
`cmc`	$\mathbf{CF} \underset{1}{\longleftarrow} \mathbf{\tilde{}CF}$

Afectação das *flags*.

Nenhuma das instruções de afectação das *flags* aceita operandos. O nome e a função destas instruções está resumida na tabela 5.2. Além destas instruções para limpar e activar *flags* existem ainda as instruções `sti` e `cli`, que serão apresentadas nas Secções 11.4 e 11.5.

Instrução	Descrição	*Flags* testadas
`ja`	Jumps if Above	`CF` = 0 e `ZF` = 0
`jnbe`	Jumps if Not Below or Equal	`CF` = 0 e `ZF` = 0
`jae`	Jumps if Above or Equal	`CF` = 0
`jnb`	Jumps if Not Below	`CF` = 0
`jb`	Jumps if Below	`CF` = 1
`jnae`	Jumps if Not Above or Equal	`CF` = 1
`jbe`	Jumps if Below or Equal	`CF` = 1 ou `ZF` = 1
`jna`	Jumps if Not Above	`CF` = 1 ou `ZF` = 1
`je`	Jumps if Equal	`ZF` = 1
`jz`	Jumps if Zero	`ZF` = 1
`jne`	Jumps if Not Equal	`ZF` = 0
`jnz`	Jumps if Not Zero	`ZF` = 0
`jg`	Jumps if Greater	`ZF` = 0 e `SF` = `OF`
`jnle`	Jumps if Not Less or Equal	`ZF` = 0 e `SF` = `OF`
`jge`	Jumps if Greater or Equal	`SF` = `OF`
`jnl`	Jumps if Not Less	`SF` = `OF`
`jnge`	Jumps if Not Greater or Equal	`SF` $\neq$ `OF`
`jl`	Jumps if Less	`SF` $\neq$ `OF`
`jle`	Jumps if Less or Equal	`ZF` = 1 ou `SF` $\neq$ `OF`
`jng`	Jumps if Not Greater	`ZF` = 1 ou `SF` $\neq$ `OF`
`jb`	Jumps if Borrow	`CF` = 1
`jc`	Jumps if Carry	`CF` = 1
`jnc`	Jumps if Not Carry	`CF` = 0
`jno`	Jumps if Not Overflow	`OF` = 0
`jo`	Jumps if Overflow	`OF` = 1
`jnp`	Jumps if Not Parity	`PF` = 0
`jpo`	Jumps if Parity Odd	`PF` = 0
`jp`	Jumps if Parity	`PF` = 1
`jpe`	Jumps if Parity Even	`PF` = 1
`jns`	Jumps if Not Signal	`SF` = 0
`js`	Jumps if Signal	`SF` = 1
`jcxz`	Jumps if CX zero	`cx` = 0

Tabela 5.1: Instruções de salto condicional

Instrução	Acrónimo	Descrição
`std`	Set Direction Flag	Assinala *flag* `DF`
`cld`	Clear Direction Flag	Limpa *flag* `DF`
`sti`	Set Interrupt Flag	Assinala *flag* `IF`
`cli`	Clear Interrupt Flag	Limpa *flag* `IF`
`stc`	Set Carry Flag	Assinala *flag* `CF`
`clc`	Clear Carry Flag	Limpa *flag* `CF`
`cmc`	Complement Carry Flag	Troca valor de *flag* `CF`

Tabela 5.2: Instruções para alteração das *flags*

6

Instruções Aritméticas

6.1 Introdução

Neste capítulo são apresentadas as instruções do 80386 que permitem efectuar operações aritméticas e as mnemónicas que lhes correspondem na linguagem assembly. Naturalmente que este é um conjunto de instruções que um microprocessador não poderia dispensar.

Antes de se introduzirem as instruções, chama-se a atenção para dois aspectos importantes. Em primeiro lugar, todas as operações aritméticas que estas instruções realizam são inteiras. Para realizar operações com vírgula flutuante terão de ser usadas instruções do co-processador aritmético, não abordado neste texto. Veja-se sobre este assunto (Murray & Pappas, 1986).

O outro aspecto importante a ter em conta prende-se com a dimensão das operações. Como no 80386 os registos têm 32 *bits*, as operações estão limitadas a esta dimensão, embora com duas importantes excepções: nas divisões o dividendo pode ter até 64 *bits*, nas multiplicações o produto também pode ser armazenado em 64 *bits*.

No fim deste capítulo, na secção 6.9 encontram-se resumidas em forma de tabela as restrições que se aplicam aos operandos das operações aritméticas.

6.2 Adição

`add org, dst`	`dst` ⟵ `dst + org`
`adc org, dst`	`dst` ⟵ `dst + org + CF`

Instrução `add`: adição. Instrução `adc`: adição com transporte.

Existem duas operações de adição diferentes: `add` e `adc`. Ambas recebem dois operandos e adicionam-nos, deixando a soma no segundo operando (`dst`) que é, portanto, alterado por esta instrução. A diferença entre `add` e `adc` é que `adc` considera na adição o valor actual da *flag* de transporte (`CF`). Isto significa que a soma dependerá não só das parcelas, mas também do resultado da última instrução que tenha alterado a `CF` antes da adição.

As restrições que se aplicam aos operandos são análogas às que se aplicavam à instrução `mov`, por exemplo, e estão representadas na tabela 4.2. Ver também 6.5.

A título de regra poderá dizer-se que quase sempre será utilizada a instrução `add`, exceptuando alguns casos excepcionais em que a `CF` seja relevante. Entre estes casos poderão ser dados dois exemplos:

- para calcular adições em complementos para um, a `CF` tem de ser reintroduzida na soma [1];
- quando se pretende alargar a gama de representação dos números para além do limite do processador (de 32 para 64 *bits*, por exemplo) será necessário efectuar a adição em duas partes: primeiro os *bits* menos significativos, depois os *bits* mais significativos, sendo que nesta segunda adição tem que ser incluído o transporte resultante da primeira (o mesmo se aplicaria na subtracção).

[1] Um caso concreto em que é necessário calcular adições em complementos para um põe-se no cálculo de somas de verificação dos segmentos TCP (*Transmission Control Protocol*).

Como exemplo de utilização destas instruções, vamos considerar o código 6.1 onde se adicionam os números de 64 *bits* contidos nos registos `edx:eax` e `ecx:ebx`, sendo a soma armazenada no primeiro par de registos.

```
add %ebx, %eax #parte baixa
```
$$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{eax} + \mathbf{ebx}$$
```
adc %ecx, %edx #parte alta
```
$$\mathbf{edx} \underset{32}{\longleftarrow} \mathbf{edx} + \mathbf{ecx} + \mathbf{CF} \tag{6.1}$$

Seguem-se outros exemplos mais simples. De referir que no terceiro exemplo o endereço da célula de memória que armazena a variável `end_literal` (e não o seu conteúdo) é adicionado ao registo `ecx`, enquanto que no último é adicionado ao registo `eax` o elemento de `vector` — no caso uma palavra — que se inicia na posição dada pelo registo `esi` relativamente ao início do vector:

`addb $8, %al`	$\mathbf{al} \underset{8}{\longleftarrow} \mathbf{al} + \mathbf{8}$
`addw %bx, memoria`	$\mathbf{M[memoria]} \underset{16}{\longleftarrow} \mathbf{M[memoria]} + \mathbf{bx}$
`addl $end_literal, %ecx`	$\mathbf{ecx} \underset{32}{\longleftarrow} \mathbf{ecx} + \mathbf{end\_literal}$
`addw vector(, %esi), %eax`	
	$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{eax} + \mathbf{M[vector + esi]}$

A operação adição pode ser realizada duma forma alternativa, em algo que se poderá classificar como uma característica curiosa dos processadores da família x86. Neste caso, a instrução `lea` (veja-se secção 4.5) pode assumir um papel para o qual possivelmente não teria sido pensada. Veja-se, nesta secção, o exemplo de código 4.6, por exemplo. Em particular, para adicionar os registos `eax`, `ebx` e o número 18 e guardar o resultado em `ecx`, poder-se-ia fazer como se encontra no código 6.2. Esta alternativa tem uma vantagem

clara em termos de codificação, uma vez que com a instrução `add` não seria possível usar uma única instrução. Em termos de desempenho a vantagem pende também para a instrução `lea`, que segundo (Murray & Pappas, 1986) precisa apenas de 2 ciclos de relógio, contra 2 a 7 ciclos por cada instrução `add`.

```
leal 18(%eax, %ebx), %ecx
```
(6.2)

6.3 Subtracção

`sub org, dst`	`dst` $\longleftarrow$ `dst` $-$ `org`
`sbb org, dst`	`dst` $\longleftarrow$ `dst` $-$ `org` $-$ **CF**

Instrução `sub`: subtracção. Instrução `sbb`: subtracção com empréstimo.

À semelhança do que sucede na adição, na subtracção existem também duas variantes: são elas `sub` e `sbb` — na primeira não é considerada a `CF`, ao contrário do que sucede na segunda. A mnemónica `sbb` significa "subtract with borrow" (subtrair com empréstimo). A situação mais normal será utilizar-se a instrução `sub`, ficando a instrução `sbb` remetida para casos mais invulgares, semelhantes aos que serviram de exemplo quando foi apresentada a adição.

A diferença será armazenada no operando `dst`, permanecendo `org` inalterado. As restrições que se aplicam aos dois operandos são as mesmas que vigoram para as instruções `add` e `adc`.

Um aspecto muito relevante a ter em conta tem a ver com o formato em que são representados os números negativos. De forma a efectuar as subtracções directamente, ignorando o sinal dos operandos é utilizada a representação em complementos para dois, representação esta não abordada neste texto. Confirma-se apenas o que se está a dizer através da apresentação dos

exemplos representados na figura 6.1. É de referir que o aparecimento dum empréstimo para o *bit* mais significativo resulta no assinalar da `CF`, mas que esse facto não é sinónimo de transbordo (que será assinalado na `OF`).

	1 | 00000000	00000001	= +1
-	11111111	11111111	= -1
	0 | 11111111	11111110	= +2
	11111111	11111100	= -4
-	00000000	00001000	= +8
	11111111	11110100	= -12

Figura 6.1: Subtracção em complementos para 2

Sendo a subtracção efectuada desta forma é evidente que os números também podem ser considerados como não tendo sinal mas, neste caso, é preciso tomar atenção ao tratamento das subtracções de que resultariam números negativos: esta situação, que corresponde à ocorrência de transbordo é, desta vez, detectável pela *flag* de transporte e não pela *flag* de transbordo propriamente dita (i.e., pela `CF` e não pela `OF`).

6.4 Multiplicação

A multiplicação é outra das operações aritméticas básicas que aqui vão ser apresentadas. Ao contrário do que sucede na adição e na subtracção, onde o facto de se considerarem os números como tendo ou não sinal era irrelevante, graças à representação em complementos para dois, aqui já não se passa o mesmo. Para perceber este facto considere-se o número binário 1111, de apenas quatro *bits*. Importa saber se este número tem sinal, caso em que representará o valor -1 ou se não tem, caso em que valerá 15. Isto porque, se este número for o multiplicador e se o multiplicando fosse igual a 2, por exemplo, o resultado seria 11111110 (-2), se se considerasse sinal ou 00011110

(30), caso contrário e considerando que há 8 *bits* disponíveis para o resultado. Posto de outra forma, isto quer dizer que tem de haver duas multiplicações diferentes.

6.4.1 Multiplicação Sem Sinal

`mul reg-mem` **edx:eax** $\left|\underset{64}{\longleftarrow}\right|$ **reg-mem** $\times$ **eax**

Multiplicação sem sinal.

A mais simples das multiplicações é a que não considera sinal. A instrução que efectua esta operação chama-se `mul`.

Recebe um único operando que só pode ser uma célula de memória ou um registo. Não poderá ser um literal. A instrução a efectuar, de facto, depende da dimensão do operando que poderá ter 8, 16 ou 32 *bits*, de acordo com a tabela 6.1.

Tamanho 1º Factor	2º Factor	Produto
byte	`al`	`ax`
palavra	`ax`	`dx:ax`
longo	`eax`	`edx:eax`

Tabela 6.1: Dimensão das operações `mul` e `imul`

Sempre que o operando for uma célula de memória é obrigatório indicar a sua dimensão com um dos sufixos habituais — '`b`', '`w`' ou '`l`' — que se adiciona como sempre ao nome da instrução. Caso contrário não será possível ao assemblador saber a dimensão da operação em causa.

Não serão apresentados exemplos desta instrução. Estes são, em tudo, idênticos aos da instrução `imul` a apresentar já de seguida.

No caso da multiplicação sem sinal (e também no caso da multiplicação com sinal — instrução `imul` — quando esta só recebe um operando) nunca

pode ocorrer transbordo, uma vez que o espaço para armazenar o produto tem sempre o dobro dos *bits* dos factores. No entanto, se o produto exceder a dimensão dos factores iniciais (i.e., não couber em `al`, `ax` ou `eax`, conforme o caso) esse facto é registado nas *flags* `CF` e `OF` que são postas a 1. Caso contrário estas *flags* são limpas.

6.4.2 Multiplicação Com Sinal

`imul reg-mem`	**edx:eax** $\underset{32}{\longleftarrow}$ **reg-mem** $\times$ **eax**
`imul op, reg`	**reg** $\longleftarrow$ **reg** $\times$ **op**
`imul literal, reg-mem, reg`	**reg** $\longleftarrow$ **reg-mem** $\times$ **literal**

Multiplicação com sinal.

A instrução que efectua a multiplicação com sinal é a `imul` (multiplicação inteira). Esta instrução tem três variantes que diferem entre si no número e no significado dos seus operandos. É de referir que a descrição apresentada da primeira versão desta instrução pressupõe a utilização de registos de 32 *bits*. Em termos de linguagem assembly utilizar-se-ia o sufixo `l` para a instrução, nesse caso, ou seja, `imull`. Importa referir que também são possíveis operandos com 8 e com 16 *bits*.

A primeira das três é, em tudo, semelhante à multiplicação sem sinal — `mul`. Também aqui, `reg-mem` poderá ser um registo ou uma célula de memória, mas não um literal. O(s) registo(s) de armazenamento do resultado dependem do tamanho da operação em causa (8, 16 ou 32 *bits*), de acordo com a tabela 6.1. A afectação das *flags* também é idêntica.

Na segunda versão, que recebe dois operandos é efectuado o produto do operando `reg` pelo valor do operando `op`, sendo o resultado armazenado em `reg`. Este operando tem de ser um registo com, pelo menos, 16 *bits*. O

operando `op` terá de ter o mesmo tamanho de `reg`, a menos que aquele se trate de um literal. Nesse caso o literal poderá ter menos *bits*, cabendo ao assemblador em alguns dos casos fazer a conversão para números com o número de *bits* adequado. Neste caso a conversão terá de ter forçosamente em conta o sinal desse número (ver subsecção 6.6).

A terceira versão da instrução `imul` recebe três parâmetros, sendo calculado o produto de `reg-mem` pelo valor literal com o resultado a ficar armazenado em `reg`. Aqui, tanto `reg` como `reg-mem` têm de ser registos com a mesma dimensão (16 ou 32 *bits*). O literal poderá ser menor.

Nas múltiplas versões desta instrução, caso haja uma ultrapassagem da capacidade do registo ou célula(s) de memória que armazena(m) o resultado (o que não é de todo impossível, porque se dois operandos têm n *bits*, poderão ser necessários até $2 \times n$ *bits* para armazenar o seu produto) esse facto é assinalado pelas *flags* `CF` e `OF`, que são postas a 1. Caso não haja transbordo `CF` e `OF` são limpas. As *flags* `SF`, `ZF`, `AF` e `PF` ficam indefinidas no fim da operação.

De forma a exemplificar o funcionamento desta instrução, são apresentados na figura 6.2 três exemplos de código e para cada um deles são apresentados os registos `eax` e `edx` no seu estado final. Há alguns aspectos a considerar, que facilitam a compreensão dos exemplos:

- a extensão do número de *bits* de um número em complementos para dois faz-se introduzindo nas posições mais significativas *bits* com o valor do sinal (ver subsecção 6.6);
- `eax` e `edx` são iniciados a 0 para simplificar a análise do resultado final;
- em todos os casos é sempre feita a multiplicação de -1 por 2, sendo usado o registo `ecx` como multiplicador; só muda o espaço disponível

para armazenar o resultado.

```
movl $0, %edx
movl $0x2, %eax
movb $-1, %cl
imulb %cl
```

eax			
00	00	ff	fe
edx			
00	00	00	00

```
movl $0, %edx
movl $0x2, %eax
movw $-1, %cx
imulw %cx
```

eax			
00	00	ff	fe
edx			
00	00	ff	ff

```
movl $0, %edx
movl $0x2, %eax
movl $-1, %ecx
imull %ecx
```

eax			
ff	ff	ff	fe
edx			
ff	ff	ff	ff

Figura 6.2: Exemplos de utilização da instrução `imul`

6.5 Divisão

`div reg-mem`	**eax** $\left\vert \underset{32}{\longleftarrow} \right\vert$ **edx:eax / reg-mem**
`idiv reg-mem`	**eax** $\underset{32}{\longleftarrow}$ **edx:eax / reg-mem**

Instrução `div`: divisão sem sinal. Instrução `idiv`: divisão com sinal.

No caso da divisão, as instruções que consideram a existência, ou não, de sinal são perfeitamente idênticas, pelo que é preferível serem apresentadas em conjunto. À semelhança do que acontecia na multiplicação, também aqui o prefixo '`i`' designa a operação com sinal. Assim, a instrução `div` designa uma divisão sem sinal, enquanto `idiv` se refere a uma divisão com sinal.

O operando recebido pela instruções `div` e `idiv`, que pode ser uma célula de memória ou um registo é o divisor da operação. O dividendo a utilizar está dependente do tamanho do divisor, mas não é passível de ser seleccionado pelo programador — poderá ser um registo ou uma composição de registos desde 16 até 64 *bits*: `ax`, `dx`:`ax`, `edx`:`eax`. O quociente e o resto são armazenados também em registos; quais, depende da dimensão da operação. A correspondência exacta é apresentada na tabela 6.2.

Divisor (`op`)	Dividendo	Quociente	Resto
byte	`ax`	`al`	`ah`
Palavra	`dx:ax`	`ax`	`dx`
Longo	`edx:eax`	`eax`	`edx`

Tabela 6.2: Operações `div` e `idiv`

Como exemplo considere-se o caso em que se pretende dividir um valor armazenado em memória de 32 *bits*, pelo número 7. A solução está ilustrada no código 6.3. Como não há garantia de o quociente caber em 16 *bits* convém efectuar uma divisão longa. Admite-se que número é positivo.

`movl numero, %eax`	$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{M[numero]}$
`movl $0, %edx`	$\mathbf{edx} \underset{32}{\longleftarrow} \mathbf{0}$
`movl $7, %ecx`	$\mathbf{ecx} \underset{32}{\longleftarrow} \mathbf{7}$
`idivl %ecx`	$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{edx{:}eax \ / \ ecx}$
	$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{edx{:}eax \ \% \ ecx}$

(6.3)

Se `numero` fosse negativo (i.e., tivesse 1 no *bit* mais significativo) o código 6.3 estaria errado. A instrução `movl $0, %edx` teria de ser substituída por outra(s) que fizessem a extensão do número com sinal armazenado em `eax` de forma correcta. Esse será o tema da próxima secção.

6.6 Extensão de Números Com Sinal

`cbtw`	$\mathbf{ax} \underset{16}{\longleftarrow} \mathbf{al}$
`cwtl`	$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{ax}$
`cwtd`	$\mathbf{dx{:}ax} \underset{32}{\longleftarrow} \mathbf{ax}$
`cltd`	$\mathbf{edx{:}eax} \underset{64}{\longleftarrow} \mathbf{eax}$

Instrução `cbtw`: Converte *byte* para palavra. Instrução `cwtl`: Converte palavra para longo. Instrução `cwtd`: Converte palavra para duplo. Instrução `cltd`: Converte longo para duplo.

Em certas ocasiões, nomeadamente para efectuar operações aritméticas, torna-se necessário fazer o aumento do número de *bits* da representação dum número. Se esse número não tiver sinal esse aumento é feito de forma trivial, bastando para o efeito introduzir 0 em todos os *bits* que serão sempre adicionados à esquerda do número inicial. Na tabela 6.3 está ilustrada a extensão do número 15 (1111) de 4 para 8 *bits*.

Se fosse usada a mesma técnica para estender número com sinal o mesmo padrão binário (1111), que representaria neste caso o número -1, seria estendido para 15 o que estaria manifestamente incorrecto. Se em vez do número anterior, fosse estendido o 7 (01111) introduzindo zeros, desta vez, o resultado estaria novamente correcto. Postos estes exemplos, pode-se avançar para uma regra: um número com sinal estende-se introduzindo à esquerda do número, em todos os *bits* que correspondem à extensão, o valor do sinal. ou seja, a extensão correcta de 1111 seria 11111111, uma vez que o sinal (i.e. o *bit* mais significativo é igual a 1). Todos os exemplos que têm vindo a ser apresentados estão representados na tabela 6.3.

Com sinal?	Número	Decimal	Extensão	Decimal	Correcto?
Não	1111	15	00001111	15	Sim
Sim	1111	-1	00001111	15	Não
Sim	0111	7	00000111	7	Sim
Sim	1111	-1	11111111	-1	Sim

Tabela 6.3: Exemplos de extensões realizadas correcta e incorrectamente

Para facilitar a vida do programador existem várias instruções capazes de efectuar extensões de números com sinal para as dimensões mais utilizadas

no 80386. São elas `cbtw`, `cwtl`, `cwtd` e `cltd`, que convertem, respectivamente, de 8 para 16, de 16 para 32, de 16 para 32 e de 32 para 64 *bits*. O '`c`' inicial e o '`t`' na terceira posição significam, respectivamente, "convert"e "to"; o '`b`', o '`w`', o '`l`' e o '`d`' referem-se ao comprimento dos dados e têm o significado habitual. Respectivamente, *byte*, *word*, *long* e *double*. Note-se que a palavra *double* tem aqui um significado duplo porque tanto se refere a números de 32 como de 64 *bits*.

Todas estas instruções operam sobre o registo `eax` (ou subregistos). O modo de operar destas instruções está representado na tabela 6.4.

Instrução	Reg. Inicial	Reg. Final	Dim. Inicial	Dim. Final
`cbtw`	`al`	`ax`	8	16
`cwtl`	`ax`	`eax`	16	32
`cwtd`	`ax`	`dx:ax`	16	32
`cltd`	`eax`	`edx:eax`	32	64

Tabela 6.4: Instruções para extensão de números com sinal

Note-se que nenhuma destas instruções recebe operandos, uma vez que estes estão implícitos. O exemplo de código 6.3 atrás apresentado e que só funciona para números positivos pode agora ser rescrito com uma solução que funciona em todos os casos. Esta solução está representada no código 6.4.

```
movl numero, %eax       eax <-32- M[numero]
cltd                    edx:eax <-64- eax
movl $7, %ecx           ecx <-32- 7                    (6.4)
idivl %ecx              eax <-32- edx:eax / ecx
                        edx <-32- edx:eax % ecx
```

6.7 Incrementação e Decrementação

`inc reg-mem`	**reg-mem** ⟵ **reg-mem + 1**
`dec reg-mem`	**reg-mem** ⟵ **reg-mem − 1**

Instrução `inc`: incrementação. Instrução `dec`: decrementação.

Estas instruções são extremamente simples: `inc` incrementa o operando `reg-mem`; `dec` decrementa-o. O operando terá de ser um registo ou uma célula de memória, sendo que neste segundo caso terá de ser forçosamente indicada a dimensão da operação.

Nenhuma destas instruções afecta a *flag* de transporte (`CF`).

6.8 Negação

`neg reg-mem`	**reg-mem** ⟵ **~reg-mem**

Negação *bit* a *bit*.

Calcula o simétrico do operando `reg-mem` (registo ou endereço de memória). Em termos de representações em complementos para 2 esta operação corresponde a calcular o complemento para 2 do operando `reg-mem` (trocar todos os *bits* e adicionar uma unidade).

6.9 Restrições às operações aritméticas

As restrições que se aplicam às operações aritméticas estão representadas por operação na tabela 6.5, além das limitações próprias do tipo de operando, representadas na tabela 4.1 e das restrições que se aplicam a cada uma das operações quando apresentadas na respectiva secção.

Registo ou memória (`reg-mem`), registo (`reg`), valor direito (`org`), valor esquerdo (`dst`)
Registo `eip/ip` Registo de segmento

Tabela 6.5: Restrições aos operandos nas instruções aritméticas multiplicação, divisão, adição e subtracção

7

Operações com bits

7.1 Introdução

Neste capítulo vai ser apresentado um conjunto de instruções que permitem efectuar manipulações aritméticas sobre os seus operandos. Como estas instruções têm a capacidade de considerar os *bits* individualmente, ou então, conjuntos de *bits*, são designadas por operações com *bits*.

7.2 Instruções `and`, `or` e `xor`

`and org, dst`	`dst` ⟵ `dst` & `org`
`or org, dst`	`dst` ⟵ `dst` | `org`
`xor org, dst`	`dst` ⟵ `dst` ^ `org`

Instrução `and`: *e bit* a *bit*. Instrução `or`: *ou bit* a *bit*. Instrução `xor`: *ou-exclusivo bit* a *bit*.

Como habitualmente, os operandos `org` e `org` podem ser um registo, um valor literal ou uma célula de memória. Aplicam-se entre eles as mesmas restrições que foram apresentadas na tabela 4.2.

Todas as três operações podem ser efectuadas com *bytes*, palavras ou longos. O resultado da operação em causa é armazenado no operando `dst`.

Na tabela 7.1 está definida a função `and`. As funções `or` e `xor` estão definidas nas tabelas 7.2 e 7.3, respectivamente.

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 7.1: Operação lógica *and*

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 7.2: Operação lógica *or*

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 7.3: Operação lógica *xor*

As operações lógicas estão definidas apenas no domínio $\{0, 1\}$. Por este motivo, quando efectuadas sobre um inteiro, estas operações lógicas consideram os *bits* desse inteiro individualmente, pelo que os operandos numa operação deste tipo deverão ter sempre o mesmo tamanho. Assim sendo, quando é executada uma destas instruções — o `and` digamos —, na verdade, não é calculado apenas um *E* lógico, mas sim um *E* lógico para cada *bit* do número inteiro. O mesmo se passa com as outras duas instruções.

Vamos considerar, a título de exemplo, dois números de oito *bits*: 185 e 233. A sua representação em binário sem sinal é a seguinte:

185 = 10111001

233 = 11101001

Os resultados de efectuar cada uma das três operações sobre estes dois números seria o que está representado na figura 7.1.

De seguida apresentamos o exemplo de código 7.1 que permitiria efectuar estas três operações armazenando os resultados em `ah` (`and`), `bl` (`or`) e `cl`

	10111001	= 185		10111001	= 185		10111001	= 185
and	11101001	= 233	*or*	11101001	= 233	*xor*	11101001	= 233
	10101001	= 169		11111001	= 249		01010000	= 80

Figura 7.1: Exemplo de cada uma das operações lógicas

(`xor`).

`movb $185, %al`	$\mathbf{al} \underset{8}{\longleftarrow} \mathbf{185}$
`movb $233, %ah`	$\mathbf{ah} \underset{8}{\longleftarrow} \mathbf{233}$
`andb %al, %ah`	$\mathbf{ah} \underset{8}{\longleftarrow} \mathbf{ah}\ \&\ \mathbf{al}$
`movb $233, %bl`	$\mathbf{bl} \underset{8}{\longleftarrow} \mathbf{233}$
`orb %al, %bl`	$\mathbf{bl} \underset{8}{\longleftarrow} \mathbf{bl}\ \&\ \mathbf{al}$
`movb $233, %cl`	$\mathbf{cl} \underset{8}{\longleftarrow} \mathbf{233}$
`xorb %al, %cl`	$\mathbf{cl} \underset{8}{\longleftarrow} \mathbf{cl}\ \hat{\mathbf{a}}\mathbf{l}$

(7.1)

É ainda de referir que é vulgar utilizar-se a instrução `xor` para zerar um número, em vez de fazer uma atribuição do valor literal zero:

`xorl %eax, %eax`	$\mathbf{eax} \underset{32}{\longleftarrow} \mathbf{0}$

7.3 Instruções `sar`, `shr`, `sal` e `shl`

`sal 1, dst`	`dst` $\longleftarrow$ `dst` $<<$ **1**
`sal %cl, dst`	`dst` $\longleftarrow$ `dst` $<<$ **cl**
`sal literal, dst`	`dst` $\longleftarrow$ `dst` $<<$ **literal**
`sar 1, dst`	`dst` $\longleftarrow$ `dst` $>>$ **1**
`sar %cl, dst`	`dst` $\longleftarrow$ `dst` $>>$ **cl**
`sar literal, dst`	`dst` $\longleftarrow$ `dst` $>>$ **literal**
`shl 1, dst`	`dst` $\vert\longleftarrow\vert$ `dst` $<<$ **1**
`shl %cl, dst`	`dst` $\vert\longleftarrow\vert$ `dst` $<<$ **cl**
`shl literal, dst`	`dst` $\vert\longleftarrow\vert$ `dst` $<<$ **literal**
`shr 1, dst`	`dst` $\vert\longleftarrow\vert$ `dst` $>>$ **1**
`shr %cl, dst`	`dst` $\vert\longleftarrow\vert$ `dst` $>>$ **cl**
`shr literal, dst`	`dst` $\vert\longleftarrow\vert$ `dst` $>>$ **literal**

Instrução `sal`: deslocamento para a esquerda com sinal. Instrução `sar`: deslocamento para a direita com sinal. Instrução `shl`: deslocamento para a esquerda sem sinal. Instrução `shr`: deslocamento para a direita sem sinal.

O operando `org`, pode ser o registo `cl`, um valor literal diferente de 1 ou pode ser 1 (a razão desta distinção prende-se com um comportamento ligeiramente diferente que estas instruções apresentam neste último caso). O operando `dst` será o registo ou endereço duma célula de memória afectado pela instrução.

As instruções `sal` e `shl` são a mesma instrução, mas com mnemónicas diferentes, e deslocam os *bits* de `dst` para a esquerda um determinado número de unidades. Se o deslocamento for de n unidades os n *bits* mais significativos saem de `dst` e entram por sua vez n zeros para o lugar dos n *bits* menos significativos. Este deslocamento de n *bits* corresponde a fazer uma

multiplicação por 2^n.

No que diz respeito a deslocamentos para a direita, a instrução `shr` faz um deslocamento de n *bits* para a direita, introduzindo zeros (0) nas posições mais significativas. Isto corresponde a fazer uma divisão sem sinal por 2. A instrução `sar` difere apenas no facto de fazer uma divisão com sinal. Para isso, os *bits* introduzidos nas posições mais significativas são idênticos ao *bit* mais significativo antes da operação.

É de referir que, no deslocamento à esquerda, a multiplicação é sempre feita com sinal, quando os números negativos são representados em códigos de complementos para dois. Com efeito, se houver uma mudança de sinal ou, por outras palavras, do *bit* mais significativo, isto significará que houve transbordo. Note-se que quando os dois *bits* mais significativos têm sinais diferentes não é possível efectuar uma multiplicação por dois sem que ocorra transbordo.

Em qualquer uma das operações que foram introduzidas, a `CF` (*carry flag*) é sempre envolvida:

- quando o deslocamento é de apenas 1 *bit* a `CF` guarda o valor do *bit* que saiu do operando (*bit* mais significativo se o deslocamento foi para esquerda, *bit* menos significativo se o deslocamento foi para a direita);

- os deslocamentos superiores a 1 *bit* podem ser pensados como múltiplos deslocamentos de apenas 1 *bit*, pelo que a `CF` armazena o valor do último *bit* que saiu do operando.

A *flag* de transbordo (`OF`) também é afectada, embora apenas nos deslocamentos de 1 *bit*. A ideia é que, como fazer um deslocamento equivale a fazer uma multiplicação ou divisão (conforme o deslocamento seja para a esquerda ou para a direita, respectivamente), então a `OF` deve reflectir a existência ou

ausência de transbordo na instrução. A regra será então a seguinte: para deslocamentos à esquerda a `OF` é reposta a 0 se o *bit* mais significativo do resultado for igual à `CF` (isto significa que não houve mudança de sinal na multiplicação, ou seja, que não houve transbordo); para a operação `sar` a `OF` é posta a 0 e para a instrução `shr` (divisão sem sinal) a *flag* de transbordo armazena o sinal (i.e., o *bit* mais significativo) do operando inicial

Consideremos como exemplos destas operações os seguintes casos:

- dividir o registo `eax` por 16;
- multiplicar um número por 2. Se o número inicial for negativo verificar se houve transbordo.

O primeiro caso está representado no código 7.2 e é bastante simples de resolver, aliás, basta uma instrução. O segundo caso, representado no código 7.3 é ligeiramente mais complexo.

```
sarl $4, %eax                    eax <-32- eax / 16          (7.2)
```

```
    movl numero, %eax            eax <-32- M[numero]
    shll $1, %eax                eax <-32- eax × 2
    jc negativo                  CF ? eip <-32- eip + x
    jmp ok                       eip <-32- ok
negativo:
    jo overflow                  OF ? eip <-32- eip + x      (7.3)
    jmp ok                       eip <-32- ok
overflow:
    #Número negativo e overflow - faz qq coisa
ok:
    #Número positivo ou não overflow - continua
```

7.4 Instruções `rcl`, `rcr`, `rol` e `ror`

`rcl 1, reg-mem`	**reg-mem** $\overset{\circlearrowleft}{C}$ **1**
`rcl %cl, reg-mem`	**reg-mem** $\overset{\circlearrowleft}{C}$ **cl**
`rcl literal, reg-mem`	**reg-mem** $\overset{\circlearrowleft}{C}$ **literal**
`rcr 1, reg-mem`	**reg-mem** $\overset{\circlearrowright}{C}$ **1**
`rcr %cl, reg-mem`	**reg-mem** $\overset{\circlearrowright}{C}$ **cl**
`rcr literal, reg-mem`	**reg-mem** $\overset{\circlearrowright}{C}$ **literal**
`rol 1, reg-mem`	**reg-mem** $\circlearrowleft$ **1**
`rol %cl, reg-mem`	**reg-mem** $\circlearrowleft$ **cl**
`rol literal, reg-mem`	**reg-mem** $\circlearrowleft$ **literal**
`ror 1, reg-mem`	**reg-mem** $\circlearrowright$ **1**
`ror %cl, reg-mem`	**reg-mem** $\circlearrowright$ **cl**
`ror literal, reg-mem`	**reg-mem** $\circlearrowright$ **literal**

Instrução `rcl`: rotação para a esquerda pela *flag* de transporte. Instrução `rcr`: rotação para a direita pela *flag* de transporte. Instrução `rol`: rotação para a esquerda. Instrução `ror`: rotação para a direita.

A forma de utilização destas instruções é idêntica à das instruções anteriores de deslocamento. Também aqui, `dst` é o operando que sofrerá a rotação, enquanto `org` pode ser um valor literal igual a 1, diferente de 1 ou poderá, ainda, ser o registo `cl`. As restrições que se aplicam ao operando `reg-mem` são as de sempre (ver tabela 4.1). Quanto ao primeiro operando, este poderá ser uma unidade, o registo `cl` ou um literal de apenas 8 *bits*.

No caso das instruções `rol` e `ror`, uma rotação consiste num deslocamento, respectivamente, para a esquerda ou para a direita de todos os *bits*, mas onde o *bit* que sai, que será o *bit* mais significativo, no caso da `rol`,

reentra na posição menos significativa (exactamente o oposto na instrução `ror`). No caso da instrução `rol` o valor inicial do *bit* mais significativo é passado também para a `CF`. Na instrução `ror` é ao contrário, sendo passado para a `CF` o valor do *bit* menos significativo.

As instruções `rcl` e `rcr` diferem das anteriores porque aqui a `CF` também participa na rotação. Para isso o *bit* mais significativo (no caso da `rcl`) ou o *bit* menos significativo (no caso da `rcr`) saem para a `CF` e o conteúdo da `CF` passa para a posição deixada livre pelo deslocamento dos *bits* do operando.

A figura 7.2 ilustra o funcionamento destas quatro operações.

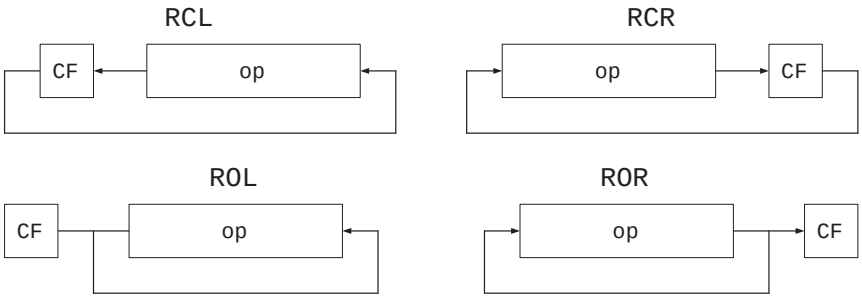

Figura 7.2: Operações `rol`, `ror`, `rcl` e `rcr`

Naturalmente que a `CF` é afectada nesta operação. Além desta, também a `OF` é afectada, apenas em rotações de 1 *bit*, segundo a seguinte regra: se a `CF` for igual ao *bit* mais significativo, a `OF` é posta a 0, caso contrário é posta a 1. Para as operações `rcl`, `rol` e `ror`, o teste à `OF` é feito no fim da operação. Para a instrução `rcr` o teste à `OF` é feito antes da rotação.

7.5 Instrução `test`

`test org, dst` **eflags** `<-- dst & org`

Efectua em *e* lógico, alterando apenas as *flags* (i.e., sem alterar os operandos).

Os operandos `org` e `dst` podem ser quaisquer, com as restrições presentes na tabela 4.2. Esta operação efectua um *E* lógico dos dois operandos sem os alterar. Afecta apenas as seguintes *flags*: `OF` $= 0$ e `CF` $= 0$, sendo as *flags* `SF`, `ZF` e `PF` afectadas de acordo com o resultado da operação.

A situação em que é expectável encontrar esta instrução será antes de um salto condicional `jcc`.

8

Definição de Dados

8.1 Introdução

A memória dum computador é organizada como uma sequência endereçada de *bytes*, capaz de armazenar qualquer tipo de informação, desde que traduzida para os habituais 0 e 1 agrupados em conjuntos de 8 *bits* ou em múltiplos. A memória e as questões que lhe estão relacionadas foram, aliás, o tema do capítulo 3. Isto significa que, num endereço qualquer, poderá à partida ser armazenado um dado de um tipo arbitrário (caracter, palavra, inteiro, número de vírgula flutuante, etc.), o que é o mesmo que dizer que a memória não é tipada.

Este facto reflecte-se também na linguagem assembly, que apresenta apenas um suporte mínimo para tipos, mas sem que haja por isso qualquer espécie de verificações. Apesar disto, a utilização de tipos justifica-se essencialmente por dois motivos:

- é mais fácil e directo reservar espaço em memória para as variáveis necessárias;
- é mais fácil iniciar essas variáveis.

A definição de um tipo para uma variável não impede, porém, que se atribua à variável em questão um valor representado noutro tipo diferente,

mesmo quando este tenha um tamanho diferente. A consequência de o fazer é a possibilidade de introduzir erros no programa, mas a decisão fica sempre nas mãos do programador, pois o assemblador Gas não efectua verificação dos tipos.

8.2 Símbolo

No contexto deste capítulo, poderemos definir símbolo como sendo o conjunto de um ou mais caracteres, que representam alguma coisa. Um símbolo poderá ser então um rótulo, um nome duma instrução ou o nome duma variável, por exemplo.

8.3 Tipos e Tamanhos

O assemblador Gas prevê a existência de três tipos de variáveis:

- números inteiros;
- cadeias de caracteres (*strings*);
- números de vírgula flutuante.

Cada um destes grandes tipos divide-se posteriormente em subtipos, que diferem entre si nalguns aspectos, nomeadamente o tamanho. De entre estes começamos com os números inteiros.

8.3.1 Números inteiros

Os números inteiros podem ter diversos tamanhos, que se encontram representados na tabela 8.1.

Nome	Dimensão	Gama sem sinal	Gama com sinal
Byte	8 *bits*	$[0, 255]$	$[-128, 127]$
Palavra	16 *bits*	$[0, 65535]$	$[-32768, 32767]$
Longo	32 *bits*	$[0, 4.294.967.295]$	$[-2.147.483.648, 2.147.483.647]$
Quadword	64 *bits*	$[0; 1,844 \times 10^{19}]$	$[-9,223 \times 10^{18}, 9,223 \times 10^{18}]$
Octaword	128 *bits*	$[0; 3,403 \times 10^{38}]$	$[-1,701 \times 10^{38}; 1,701 \times 10^{38}]$

Tabela 8.1: Tipos de números inteiros

A cada um destes tipos está associada uma directiva que permite definir variáveis do tipo respectivo. Como facilmente se compreenderá, se for definida uma variável do tipo longo, por exemplo, será reservado em memória espaço para armazenar 32 *bits* (4 *bytes*) consecutivos. Chama-se novamente a atenção para o facto de que nestes 4 *bytes* se pode armazenar informação de qualquer espécie e não apenas um conjunto de 32 *bits* que representem um número inteiro. A tabela 8.2 lista as directivas a utilizar para cada tipo.

Nome	Dimensão	Directiva
Byte	1 *byte*	`.byte`
Palavra	2 *bytes*	`.word`, `.short`, `.hword`
Longo	4 *bytes*	`.long`, `.int`
Quadword	8 *bytes*	`.quad`
Octaword	16 *bytes*	`.octa`

Tabela 8.2: Tipos de números inteiros

A sintaxe da definição de uma variável de qualquer tipo e em particular de variáveis inteiras é apresentada de seguida:

```
nome: .directiva valor_inicial {, valor_inicial}
```

A definição começa obrigatoriamente por um nome, que pode ser escolhido segundo as mesmas regras que se aplicam aos restantes símbolos, nomeadamente rótulos. A directiva pode ser uma das que se encontram

na tabela 8.2 ou qualquer outra relativa a um tipo de dados diferente. Na definição de uma variável é sempre indicado o seu valor inicial, que a variável garantidamente assume quando o programa se inicia. Entre {} encontra-se uma parte opcional e que pode ser repetida um número arbitrário de vezes. Para cada valor inicial introduzido é reservado espaço para mais uma variável do mesmo tipo, num endereço contíguo de memória, de forma que na mesma definição podem-se especificar vectores ou tabelas. Uma forma alternativa de definir um vector consiste em por noutra linha, ou na mesma linha, mas separada por um ponto e vírgula (';'), a mesma directiva com outra inicialização.

No código 8.1 apresentam-se alguns exemplos, sendo a variável `var_em_hexa` e a variável `filtro` iniciadas com valores em hexadecimal e em binário, respectivamente.

```
letra:       .byte 'a'
pontos:      .int 0
mto_gde:     .quad 8895321
var_em_hexa: .word 0x844f
filtro:      .int 0b01001011
```
(8.1)

8.3.2 Ponteiros

Um tipo especial de números inteiros são os ponteiros, que guardam valores que representam endereços de outras variáveis. Não é objectivo deste texto aprofundar muito este tema, mas interessa de qualquer forma saber como se define um ponteiro.

O espaço necessário para um ponteiro pode ser de 48 *bits* ou de 32 *bits*. Depende do facto de se estar a considerar o endereço completo (segmento + deslocamento) ou apenas o deslocamento. Normalmente, este segundo

elemento é suficiente porque o programa reside todo no mesmo segmento. Note-se que isto não se verifica em todas as plataformas.

O exemplo de código 8.2 ilustra a forma de definir e iniciar um ponteiro (`ptr_numero`) capaz de armazenar um deslocamento. A figura 8.1 representa a situação que se obtém depois desta definição. Em termos conceptuais, `ptr_numero` aponta para `numero`; daí a seta da figura e a utilização do nome ponteiro.

Para armazenar um endereço completo seria necessário reservar espaço para o segmento (depois do deslocamento) e fazer a inicialização do segmento à parte — no código. Isto está ilustrado no código 4.7, apresentado na secção 4.6, relativa às instruções `lxs`.

```
numero: .int 15
ptr_numero: .int numero
```
(8.2)

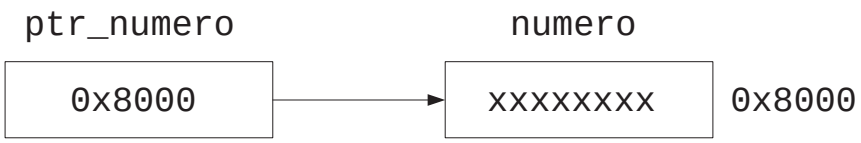

Figura 8.1: Definição de um ponteiro

É de referir que a utilização de ponteiros já tinha sido feita anteriormente, embora esse facto não tivesse sido referido antes. Por exemplo, no endereçamento indirecto, quando se escreve `(%ebx)` (entre parêntesis) o registo `ebx` tem de conter o endereço válido duma célula de memória onde se encontre um determinado dado, o que é o mesmo que dizer que `ebx` é um ponteiro para essa zona de memória.

8.3.3 Cadeias de Caracteres (*Strings*)

É suposto neste texto que o leitor esteja familiarizado com o conceito de cadeia de caracteres. Por isso, aqui, não vai ser feita mais do que uma enumeração a que é adicionada uma curta descrição dos tipos de cadeias suportados pelo assemblador Gas, que se encontram ilustradas na tabela 8.3, juntamente com a directiva a utilizar.

Directiva	Descrição
`.ascii`	Cadeia de caracteres não terminada
`.asciz`	Cadeia de caracteres terminada com '\0'
`.string`	Cadeia de caracteres terminada com '\0' (igual a `.asciz` no 80386)

Tabela 8.3: Tipos de cadeias de caracteres

É de referir que a utilização de cadeias do tipo `asciz` e `string` se revela especialmente útil quando se pretende utilizar algumas bibliotecas, em particular a biblioteca do C (`libc`). Isto porque muitas funções destas bibliotecas assumem que a representação de cadeias de caracteres é feita segundo a regra normalmente utilizada pelos programadores de C, onde todas as cadeias de caracteres devem terminar com o caracter '\0' (código 0), que não representa nenhum caracter válido na tabela ASCII.

A sintaxe para definição de cadeias de caracteres é a mesma que se usa para definição de variáveis inteiras. As cadeias de caracteres constantes usadas para atribuir o valor inicial são escritas entre aspas ('"'). Seguem-se alguns exemplos no código 8.3. A representação em memória destas cadeias de caracteres assim definidas encontra-se na figura 8.2. Por uma questão de simplicidade assumiu-se que todas as cadeias se iniciam na posição de memória 0.

```
nome:    .string "Eusébio da Silva Ferreira"
posicao: .ascii "Avançado"
pais:    .asciz "Portugal"
```
(8.3)

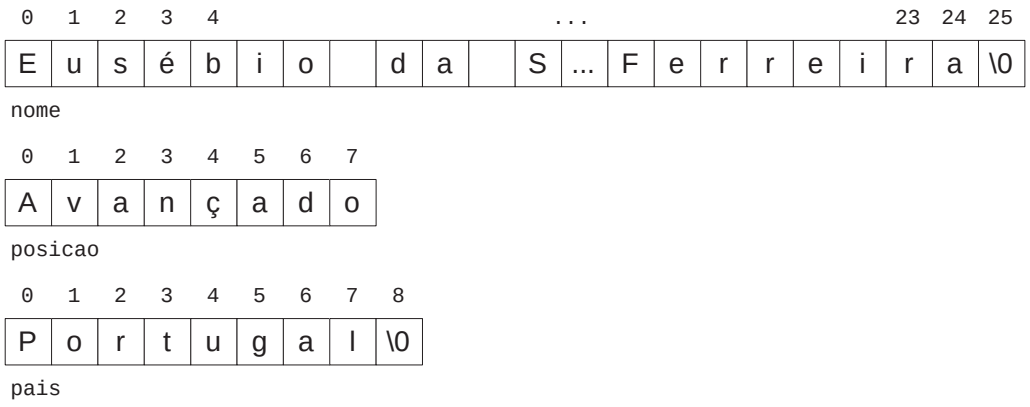

Figura 8.2: Representação em memória das cadeias de caracteres

8.3.4 Números de Vírgula Flutuante

Este tópico extravasa o âmbito deste documento, pelo que apenas são enumeradas na tabela 8.4 as directivas que permitem definir variáveis de vírgula flutuante, que podem ser de dois tipos diferentes: *float* e *double*.

Nome	Dimensão	Directiva
Float	4 *bytes*	`.float`
Double	8 *bytes*	`.double`

Tabela 8.4: Tipos de números de vírgula flutuante

A definição de constantes em vírgula flutuante é feita com o prefixo `0f`, sendo válida a utilização do símbolo `E` (maiúsculo ou minúsculo) para designar o expoente, mas tanto a mantissa como o expoente têm de ser escritos em decimal. Alguns exemplos:

```
numero:     .float 0f2.178
peso:       .float 0f1E2
distancia:  .double 0f3.84e8
```

8.3.5 Vectores e Tabelas

Muitas vezes faz sentido armazenar em espaços contíguos de memória um conjunto de dados do mesmo tipo. Uma estrutura de dados deste tipo designa-se por vector ou tabela, conforme tenha uma ou mais dimensões, respectivamente.

Um exemplo do quotidiano de utilização de um vector poderá ser uma pauta com as notas dos alunos, sendo que o vector neste caso é a coluna com as notas. Um exemplo de uma tabela poderá ser um horário, sendo nos elementos da tabela armazenadas as salas onde decorrerão as aulas. É importante compreender a diferença entre os dois exemplos: no primeiro caso a indexação é feita apenas por uma variável (nome do aluno); no segundo caso, a indexação dos valores armazenados é feita a partir de duas variáveis (hora e dia da semana). Por este motivo um vector é unidimensional e uma tabela bidimensional (eventualmente poderá ter uma dimensão superior). A figura 8.3 ilustra a diferença.

10
16
14
9
12
11
18

Vector

3.1.6		3.2.14		3.1.7
3.1.9	3.1.10		3.2.14	3.1.7
	3.1.10			8.2.39
8.2.30			8.2.10	
		...		

Tabela

Figura 8.3: Diferença entre vector e tabela

Vectores e Tabelas inicializados

A definição de um vector é extremamente simples e passa apenas por definir uma sequência de variáveis do mesmo tipo, mas sob o mesmo nome. Como vimos, isto é muito simples de fazer, basta especificar uma lista de valores separados por vírgulas, depois da directiva que indica o tipo dos elementos em questão. A sintaxe exacta pode ser encontrada na subsecção 8.3.1.

Uma pauta como a da figura 8.3 seria então definida da seguinte forma:

```
pauta: .int 18, 11, 12, 9, 14, 16, 10
```

No momento da definição de dados, as dimensões máximas dos vectores ou das tabelas terão de ser conhecidas, pois não será possível *a posteriori* adicionar-lhes ou retirar-lhes espaço.

Uma das muitas vantagem de se utilizar um vector em vez de se utilizarem variáveis distintas consiste na possibilidade de utilização de formas de endereçamento mais complexas e que permitem escrever programas duma forma mais flexível.

O acesso à nota do quarto aluno poderia ser então armazenado no registo `eax` da forma que se apresenta de seguida (`ecx` regista a posição do aluno em questão, sendo que a primeira de todas é a posição 0, motivo pelo qual o quarto aluno está na posição 3):

```
movl $3, %ecx
movl pauta(, %ecx, 4), %eax
```

O endereço do elemento genérico `i` seria calculado de acordo com a fórmula seguinte:

```
Endereço = vector + i * tamanho
```

Uma tabela, pelo contrário, é uma estrutura de dados com que é mais difícil de lidar porque, na prática, com um tabela é feita uma correspondência entre a memória que é linear (uma dimensão) e uma unidade de dados abstracta que tem mais do que uma dimensão. Isto significa que é necessário aplicar alguma forma de correspondência entre as células duma tabela e as células duma memória. Uma forma de o fazer está expressa na fórmula seguinte, onde `i` é a linha e `j` a coluna. Tanto as linhas como as colunas começam na posição 0:

```
Endereço = tabela + tamanho * (colunas * i + j)
```

Na figura 8.4 está ilustrado o endereço em que seriam armazenadas as células da tabela da figura 8.3, se fossem necessários 4 *bytes* para guardar o número da sala. Estes endereços são relativos ao endereço de origem da tabela.

0	4	8	12	16
20	24	28	32	36
40	44	48	52	56
60	64	68	72	76
		...		

Figura 8.4: Endereços relativos das células do horário

Seria naturalmente possível trocar o papel das linhas e das colunas na equação utilizada para calcular o endereço duma célula da tabela. No caso que apresentámos, diz-se que a ordem é "row-major", enquanto que esta última hipótese seria "column-major".

Em termos de definição, uma tabela não é diferente dum vector. Se a tabela tiver x células, é necessário iniciar os seus x elementos exactamente

da mesma forma que se fazia para um vector. Aliás, o assemblador não distingue uma tabela de um vector, sendo essa diferença assumida apenas pelo programador, que tem sempre que respeitar a ordem pela qual as células são armazenadas em memória, tal como ilustrado na figura 8.4.

8.3.6 Declaração de Dados Não Inicializados

Neste momento deve ser trivial calcular o espaço necessário para armazenar uma tabela ou um vector. Este espaço será igual ao produto do número de elementos da tabela ou do vector pela dimensão de cada uma desses elementos.

A necessidade deste cálculo põe-se numa situação em que, por qualquer motivo, não se pretenda inicializar o vector ou a tabela (por só ser possível conhecer os valores iniciais no momento de execução do programa, por exemplo). Neste caso, é possível reservar espaço, usando a directiva `.lcomm`, de acordo com a seguinte sintaxe (em parêntesis rectos está um parâmetro opcional):

```
.lcomm nome_da_var, espaço [alinhamento]
.comm nome_da_var, espaço [alinhamento]
```

A diferença entre as directivas `.lcomm` e `.comm` será abordada na secção 8.4.

É de referir que esta directiva permite reservar espaço para outras variáveis não inicializadas que não apenas tabelas ou vecores — é só questão de se calcular a dimensão dessas variáveis, eventualmente escalares.

8.4 Âmbito dos símbolos (Locais *vs.* Globais)

Sempre que é definido um rótulo ou uma variável, o nome em questão é visível apenas no módulo (i.e., no ficheiro) onde se encontra a sua definição.

Como facilmente se compreende tem de haver uma forma de tornar os símbolos existentes visíveis noutros módulos diferentes. Quanto mais não seja, o símbolo (rótulo) que marca o início do programa tem que ser visível para que se possa fazer a entrada nesse programa.

Para tornar um símbolo visível utiliza-se a directiva `.global` (ou apenas `.globl`). A sintaxe é a seguinte:

```
.global símbolo
```

Mais concretamente, após a utilização da directiva `.global` sobre um símbolo qualquer, este passa a ser público para o editor de ligações (`ld`, por exemplo) e ficará, desta forma, acessível para outros programas parciais que sejam ligados em conjunto.

No que diz respeito às variáveis não iniciadas, a diferença entre as variáveis declaradas com as directivas `.lcomm` e `.comm`, é que as primeiras têm um âmbito local ao ficheiro onde estão declaradas, enquanto que as segundas são globais a todos os programas parciais que componham o programa total.

8.5 Secções de Dados

Um programa é composto por múltiplas secções [1]. Três delas assumem particular relevância. Uma é a secção de texto, que é a zona de memória onde pode ser encontrado o código do programa (eventualmente também constantes). Esta secção difere das outras porque normalmente o seu conteúdo não é alterável e porque pode ser partilhada entre diferentes processos.

Outra secção importante é a de dados, chamada `data`, que é utilizada, normalmente, para armazenar variáveis do programa. A secção de texto e a

[1] Mas depois, como no Unix um programa é composto por um único segmento todas elas são agrupadas nesse segmento.

secção de dados pertencem a uma mesma categoria de secções. Juntam-se a esta categoria todas as secções criadas pelo programador e que recebam um nome, qualquer que ele seja.

Uma secção diferente e que não encaixa nesta categoria é a secção `bss`, que contém espaço para variáveis não iniciadas. Entre estas incluem-se todas as que são definidas com as directivas `.lcomm` e `.comm`. Quando é reservado o espaço relativo a esta secção, todas as variáveis são iniciadas a 0.

Acrescem a estes dois tipos de secções a secção absoluta, que contém endereços preestabelecidos e não relativos a uma determinada posição de memória (veja a página de `info` do `as` [2] e a secção indefinida. Todos os símbolos que sejam utilizados mas não definidos são, no momento da assemblagem, categorizados nesta secção. Isto só pode acontecer quando um programa é composto por vários módulos. Neste caso, se o módulo *A* define a variável `A`, que é utilizada pelo módulo *B*, então, para o assemblador, no momento da assemblagem do módulo *B*, a variável `A` pertencerá à secção indefinida. Este problema da indefinição só será resolvido quando os dois módulos já em formato objecto forem ligados.

Na figura 8.5 está representada a forma como são agrupadas as diversas secções existentes nos diferentes módulos que compõem um programa. É de referir que as secções correspondentes são agrupadas entre si e que depois as secções unificadas ocupam posições relativas fixas no programa final. O espaço de todas estas secções é depois reservado num único segmento do programa — isto no caso do sistema operativo Unix.

Num ficheiro de código assembly é possível comutar entre secções usando a directiva `.section`, de acordo com a seguinte sintaxe:

```
.section NOME
```

[2] `info as`, na linha de comando duma *shell* Unix.

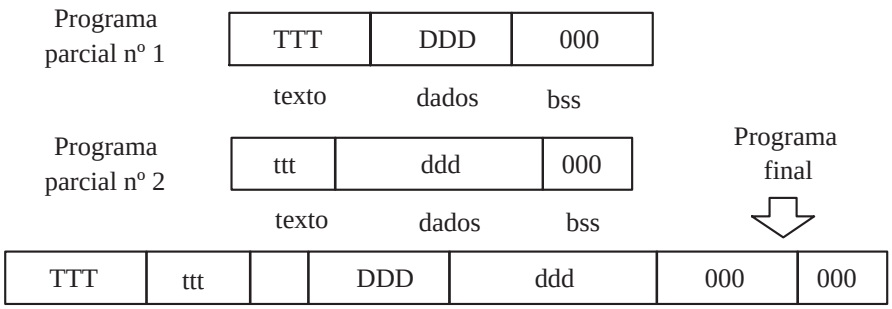

Figura 8.5: Agrupamento das várias secções dum programa

Todo o código ou definição de variáveis inicializadas que se faça imediatamente depois desta directiva (e antes de outra que comute novamente a secção) será incluída na secção com o nome `NOME` indicado.

Para simplificar a tarefa, no caso das secções de dados e de texto, podem-se usar as directivas `.data` e `.text`, respectivamente.

Note-se que a utilização desta directiva `.section` só faz sentido para a primeira categoria de secções que foi referida — as que têm nome.

Além das secções que foram referidas, existe também a secção da pilha que não tem de ser explicitamente definida no programa.

O programa executável final é então composto por múltiplas secções distintas (todas no mesmo segmento). Cada secção ocupa uma posição preestabelecida relativamente às outras secções, mas a sua dimensão pode variar. A figura 8.6 ilustra a posição de cada secção no espaço de endereçamento do segmento onde o programa será executado (i.e., do espaço de endereçamento do processo que executa o programa).

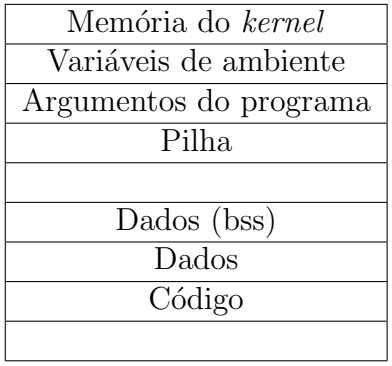

Figura 8.6: Secções de um programa

9

Funções

9.1 Introdução

Neste capítulo será apresentado um conceito de importância fundamental para a realização dum programa: as funções. Para isso, procura-se inicialmente compreender as razões que justificam a existência de funções. De seguida, são apresentadas as instruções do assembly que permitem implementar funções, passando-se depois para as convenções utilizadas na sua programação. O capítulo termina com a apresentação de funções exemplo.

9.2 Divisão de um Programa em Funções

Uma característica que tem sido comum a todos os fragmentos de código apresentados até aqui é a sua curta extensão. Nem sempre é assim. Muitos programas são, na verdade, extremamente longos. É o caso de praticamente todos os programas populares: editores e processadores de texto, folhas de cálculo, *browsers* de Internet, servidores de bases de dados, etc.. Também os sistemas operativos são, por norma, bastante extensos. Todos estes programas têm também em comum que em determinados pontos efectuam acções que se repetem ou que se assemelham consideravelmente, sendo exemplos prosaicos de situações destas, a impressão de informação no monitor ou a

leitura de dados do disco. Estas são acções repetitivas e que tendem a ocorrer múltiplas vezes.

Por exemplo, considere-se um programa em que se pretendem ler os primeiros 20 *bytes* dum ficheiro qualquer num dado momento e que mais tarde se desejam ler outros 30 *bytes*. Repetir praticamente o mesmo código nas duas situações não seria apropriado — melhor será escrever código genérico, que sirva para ambas. O comportamento deste código poderá ser ditado por um conjunto de parâmetros: uma referência para o ficheiro, o número de *bytes* a ler e o local a partir do qual será armazenada a informação que interessa. Um outro exemplo ainda mais simples ocorre com funções matemáticas — será o caso da raiz quadrada ou da potenciação isto para nomear apenas duas. Aqui, haverá um "mecanismo" cujo funcionamento não é relevante (excepto, claro, para quem o concebeu) capaz de produzir o resultado correcto dependendo do(s) parâmetro(s) de entrada da função.

O código que resolve um subproblema é então agrupado numa unidade que constitui um subprograma, a que se chama normalmente *função* ou *procedimento*. Em muitas linguagens de programação pode haver uma diferença objectiva entre estes dois conceitos. No *Pascal*, por exemplo, uma função devolve sempre um valor, sendo invocada na posição de um valor direito. Se existisse uma função chamada `raiz_quadrada()`, esta seria invocada da seguinte forma: `y = raiz_quadrada(x)`, onde `x` seria o argumento e `y` o resultado. Um procedimento, pelo contrário, não devolve qualquer valor e não é nem um valor esquerdo nem um valor direito. Por exemplo, em *Pascal* existe um procedimento chamado `writeln()` que escreve no monitor. Este procedimento deve ser invocado sem receber nem devolve qualquer valor. Será simplesmente `writeln('Olá')` para imprimir a palavra *Olá* no monitor. Noutras linguagens, como o *C* não existe diferenciação entre estas

situações, sendo todos os subprogramas chamados de funções. Neste texto será adoptada esta convenção, de forma que os termos *função* e *procedimento* serão usados de forma comutável.

9.3 Necessidade e Utilidade das Funções

A utilização de funções nos casos exemplificados permite reduzir a redundância dum programa, mas este não é o único argumento a favor da sua utilização. Um outro argumento tem a ver com a modularização do programa. Por modularização entende-se a divisão do problema inicial em subproblemas. Quando escrito de forma descuidada e monolítica, um programa, mesmo que não seja muito grande, facilmente se torna ilegível, inclusivamente para o seu programador. Isto torna o programa difícil de manter e de alterar.

Para evitar que isso aconteça o programador deve sempre preocupar-se em modularizar devidamente o seu programa. A cada módulo caberá uma ou mais tarefas e a alteração dum destes módulos não implicará a alteração dos restantes. Esta definição poderá ser recursiva, sendo possível dividir novamente os módulos iniciais em módulos mais pequenos e mais simples. Cada módulo elementar assim obtido será depois implementado por um conjunto de linhas de código isoladas no interior duma função. Cada função agrupa as linhas de código que dizem respeito à operação de um determinado módulo e deve ser tão independente das restantes quanto possível.

No entanto, não é objectivo deste texto aprofundar as condições que devem presidir à definição duma função. Aqui interessa perceber o seguinte:

- como se define uma função;
- como se invoca uma função;

- como se passam parâmetros para uma função;

- como se recebem resultados duma função.

9.4 Instrução `call`

`call reg-mem`	**esp** $\underset{32}{\longleftarrow}$ **esp - 4**
	M[esp] $\underset{32}{\longleftarrow}$ **eip**
	eip $\underset{32}{\longleftarrow}$ `reg-mem`

A instrução `call` serve para invocar uma função. A instrução `call` recebe um operando, que pode ser um endereço de memória ou um registo que represente esse endereço. Este será o endereço da primeira instrução da função que se está a invocar. O efeito mais óbvio da sua invocação é a alteração do registo `eip` e consequentemente a alteração da sequência de execução do programa. Neste aspecto a instrução `call` assemelha-se à instrução `jmp`. A grande diferença é que na instrução `call` o valor inicial do registo `eip` é salvaguardado na pilha. Isto não acontece na instrução `jmp`, onde o ponteiro de instruções `eip` é pura e simplesmente obliterado. A razão para armazenar o valor original de `eip` na pilha prende-se com a necessidade de retornar ao ponto seguinte à chamada da instrução `call` [1]. Este assunto será, posteriormente, abordado com mais detalhe.

[1] Neste ponto merece a pena fazer uma chamada de atenção para o seguinte: o valor do registo `eip` armazenado na pilha é o da instrução que se segue ao `call` uma vez que o microprocessador avança automaticamente o ponteiro de instruções `eip` quando faz a leitura de cada instrução.

9.5 Instrução `ret`

`ret`	$\mathbf{eip} \underset{32}{\longleftarrow} \mathbf{M[esp]}$
	$\mathbf{esp} \underset{32}{\longleftarrow} \mathbf{esp + 4}$
`ret literal`	$\mathbf{eip} \underset{32}{\longleftarrow} \mathbf{M[esp]}$
	$\mathbf{esp} \underset{32}{\longleftarrow} \mathbf{esp + literal + 4}$

A instrução `ret` pode ser considerada como a instrução recíproca da instrução `call`. Se o microprocessador encontrar uma instrução `ret` imediatamente depois duma instrução `call`, o efeito de ambas anula-se e o microprocessador vai encontrar-se na instrução imediatamente após o `call` inicial. Este seria o caso duma função vazia, i.e., que não contivesse qualquer instrução. Não será esta, evidentemente, a situação normal, uma vez que habitualmente existem outras instruções entre o `call` e o `ret`. No entanto, independentemente de quantas instruções existirem entre o `call` e o `ret`, estas deverão aparecer sempre conjugadas — para cada `call` deverá existir sempre um `ret` correspondente, ficando entre elas as instruções que compõem a função *chamada* [2]. Resumindo, o `call` é usado por quem chama, mais exactamente, pela função *chamante*. O `ret` é usado pela função chamada para terminar.

O funcionamento deste par de instruções está ilustrado na figura 9.1. Assume-se a existência duma função chamada `soma()`, cujo código é omitido, por não ser relevante.

A sequência temporal onde são considerados cinco estados está assinalada na figura por números dentro dum círculo. Pela ordem em que ocorrem as acções representadas são as seguintes:

[2] Note-se que isto não implica que o número de instruções `call` e `ret` num programa tenha que ser precisamente igual. É perfeitamente possível que uma função inclua muitos pontos diferentes por onde seja possível sair, i.e., pode ter muitos `ret`, mas apenas um deles será executado para terminar a função.

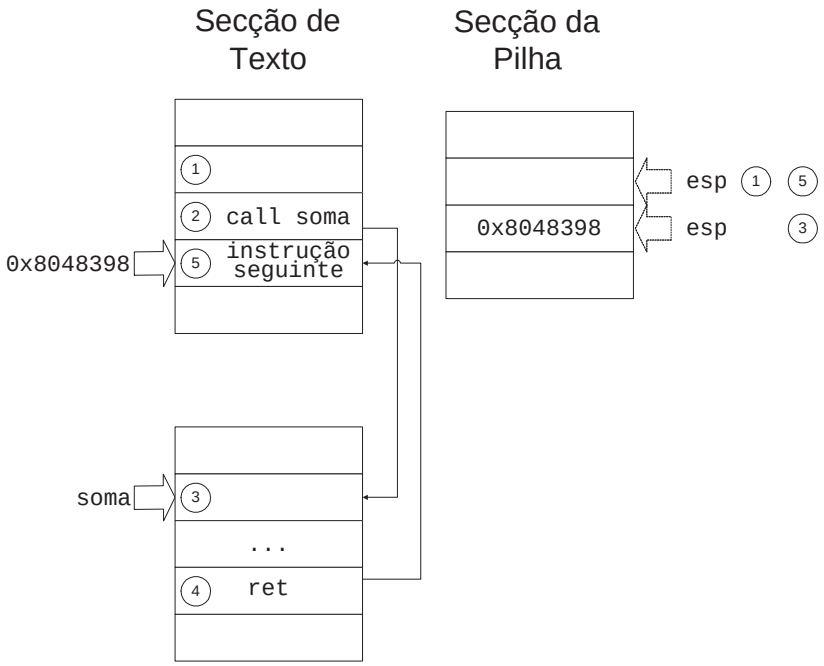

Figura 9.1: Funcionamento das instruções `call` e `ret`

- Na instrução anterior ao `call` o ponteiro `esp` encontra-se na posição indicada na figura com o número 1;
- a instrução `call soma` introduz na pilha o endereço correspondente à instrução que se lhe segue no código;
- a instrução que o microprocessador executa de facto é a que corresponde ao endereço onde se encontra a função `soma()`. O ponteiro `esp` mudou de posição;
- a função `soma` decorre normalmente até ser atingida a instrução `ret`. Aqui, o microprocessador encontra a pilha exactamente no mesmo estado em que esta se encontrava quando a função teve o seu início. Isto não significa, porém, que a pilha não tenha sido utilizada, mas apenas que tem de ser deixada exactamente no mesmo estado. A instrução

`ret` altera mais uma vez a sequência normal de execução do código. A próxima instrução a executar não é a que se lhe segue na secção de texto, mas a instrução seguinte ao `call soma`. Esta instrução corresponde ao endereço retirado da pilha;

- terminada a chamada à função a pilha encontra-se no estado em que se encontrava inicialmente, assinalada com o número 5.

A sequência descrita mostra, finalmente, porque razão é que todos os objectos que entram na pilha têm de ser retirados, de acordo com o que foi afirmado na secção 4.7.

9.6 Código das Funções

Viu-se atrás que a execução duma função vai desde a instrução seguinte ao `call` da função chamante até ao último `ret` da função chamada [3]. Isto pressupõe que haja um rótulo associado à primeira instrução da função, de forma a ser possível fazer a chamada à função através da instrução `call rotulo`. No código 9.1 está representada uma função que efectua a adição dos registos `ebx`, `ecx` e `edx` e devolve o resultado em `eax`.

```
soma_regs:
    movl %ebx, %eax       eax <-32- ebx
    addl %ecx, %eax       eax <-32- eax + ecx          (9.1)
    addl %edx, %eax       eax <-32- eax + edx
    ret
```

Interessa referir que os rótulos que identificam funções são processados pelo assemblador de forma idêntica à dos rótulos que identificam dados. Com

[3] Pode ainda acontecer que dentro da função chamada seja invocada outra função e assim sucessivamente.

efeito, durante o processo de assemblagem, o assemblador mantém um contador relativo ao início da secção corrente. Suponhamos que o assemblador se encontra na secção de texto. À medida que vai processando instruções tem que as converter de assembly para código máquina e calcular o espaço que estas ocupam. Isto permite-lhe saber exactamente quantos *bytes* já passaram desde o início da secção. Quando encontra um rótulo, o valor atribuído a esse rótulo depende da distância relativa ao início dessa mesma secção. Se a secção não for de texto, mas de dados, a única diferença é que o espaço será normalmente ocupado por dados e não por instruções. Evidentemente que o assemblador sabe o tamanho que os dados definidos pelo programador ocupam, sendo novamente possível atribuir valores aos rótulos. Uma diferença importante reside no facto de a maior parte dos dados terem rótulos, não sendo estes tão necessários para as instruções. A razão para isto é que nem todas as instruções são destinos de saltos mas, quanto aos dados, será normal que todos sejam individualmente acedidos.

Outro aspecto que importa referir diz respeito às funções que compõem um programa. Um programa pode incluir muitas funções que se invocam umas às outras. Poderá acontecer, por exemplo, haver uma função `A`, que invoca uma função `B`, que, por sua vez, invoca uma função `C` e depois uma `D`, etc., num processo que tem algo de recursivo [4].

Entre as várias questões que se levantam uma delas é em que ponto é que o programa começa, mais precisamente, qual é a primeira função do programa a ser chamada e como é que essa chamada acontece. A resposta à segunda pergunta está no sistema operativo, que inclui ele próprio uma função [5] capaz

[4] Existe também uma classe de funções em que estas se invocam a elas próprias de forma controlada. Estas funções são conhecidas por *funções recursivas*. A sua utilização justifica-se essencialmente para manipular estruturas, elas próprias, eminentemente recursivas como sejam árvores, por exemplo.

[5] Mais exactamente, uma chamada ao sistema.

de fazer uma troca de programas — trata-se da função `exec()` [6]. A invocação da chamada ao sistema `exec()` permite comutar de programa e passar a executar um novo programa, que poderá ser um editor de texto, por exemplo. A resposta à outra pergunta está numa função chamada `main()`. A chamada ao sistema `exec()` recebe o nome do ficheiro executável onde se encontra o novo programa. O programa é então carregado total ou parcialmente para memória e a função `exec()` procura uma função especial do programa com o nome `main()` [7].

Importante também é a noção de processo. A norma POSIX define processo como um espaço de endereçamento com um ou mais fluxos de execução. Espaço de endereçamento são as zonas de memória reservadas para a secção de texto, dados, pilha, etc.. Fluxo de execução refere-se à efectiva utilização dum microprocessador para executar as instruções do programa. Para uma definição mais detalhada de processo veja-se (Silbertschatz & Galvin, 1998).

É precisamente na função `main()` que o novo programa se inicia. Note-se que parece haver aqui um problema de recorrência, uma vez que um programa é sempre iniciado por outro programa. A chave da resposta reside nos processos, que se podem multiplicar. Na verdade há um processo — conhecido por *init* — que é origem ancestral de todos os processos que executam programas na máquina. Digamos que é possível construir uma espécie de árvore genealógica dos processos, sendo o processo *init* a raiz de todos eles. Cada novo processo que é criado, por norma (embora não seja obrigatório) invoca a chamada ao sistema `exec()` para correr um programa diferente. Este programa tem, por sua vez, o início na função `main()`. A partir da função `main()` invoca depois outro número de funções até que a função `main()` en-

[6] Para informação sobre a família de funções `exec()` execute-se `man exec` na linha de comandos duma *shell* Unix.

[7] Na verdade isto é uma simplificação. A função `exec()` procura uma função chamada `_start()` mas, por norma, esta não precisa de ser definida pelo programador.

contra o seu fim numa instrução `ret` [8]. Então, por norma, novos programas executam-se após aparecimento de novos processos, quando estes invocam a chamada ao sistema `exec()`.

Na realização de funções, há ainda aspectos a considerar. Como se viu atrás, a pilha deve ser deixada intacta pela função. Quanto aos registos do microprocessador, a situação é ligeiramente diferente. O programador da função pode optar, ou não, por manter os registos inalterados à saída da função. Por exemplo, o compilador `gcc` adopta a seguinte convenção: os registos `ebp`, `esi`, `edi` e `ebx` são salvaguardados pela função chamada; o registo `eax` é utilizado para devolver o resultado (se o resultado necessitar de 64 *bits* são usados os registos `edx:eax`); os restantes registos deverão, se necessário, ser salvaguardados pela função chamante. Fundamental sempre será guardar o registo `ebp`, pelas razões que se verão adiante.

Caso se pretenda fazer a salvaguarda de registos no interior duma função será normal utilizar-se a pilha para o efeito. No código 9.2 é apresentado o exemplo duma função que salvaguarda os registos `ebp`, `esi`, `edi` e `ebx`.

[8] Note-se que aqui há uma nova simplificação: o programa não acaba num `ret`, mas numa função chamada `_exit()`, que é uma fachada para a primeira de todas as chamadas ao sistema. Depois do `ret` da função `main()` o programa volta à função `_start()`, onde é feito o `_exit()`.

```
minha_func:
    pushl %ebp              Salvaguarda registos
    pushl %esi
    pushl %edi
    pushl %ebx
    # código da função aqui                          (9.2)
    popl %ebx               Reposição dos registos
    popl %edi               por ordem inversa
    popl %esi
    popl %ebp
    ret
```

Neste caso, são usadas as instruções recíprocas `push` e depois `pop` pela ordem inversa, naturalmente, uma vez que a pilha é uma estrutura LIFO. Como se viu na secção 4.8 também é possível guardar e repor todos os registos com um único par de instruções `pusha`/`popa`.

9.7 Variáveis Locais

Para armazenarem os seus próprios dados locais as funções necessitam também das suas próprias variáveis, designadas habitualmente de *variáveis locais*. O armazenamento de variáveis locais pode ser feito de três formas diferentes:

- nos registos;
- na memória;
- na pilha.

A utilização de registos como variáveis locais a uma função tem como vantagem a simplicidade e a rapidez. A desvantagem evidente é que na

família 80x86 até ao Pentium o número de registos de utilização geral é muito limitado, pelo que todos os registos fazem diferença. No código 9.1 em que a soma dos registos `ebx`, `ecx` e `edx` é armazenado no registo `eax` este último desempenha o papel de variável local.

A utilização de memória não é tão imediata como a utilização de registos, mas é, ainda assim, bastante simples, também. Além disso não conta com a desvantagem de ser um recurso extremamente limitado. Não é difícil a uma função guardar, por exemplo, vários inteiros em memória. Porém, esta solução também apresenta desvantagens. Em particular, uma desvantagem será a falta de isolamento das variáveis que, em geral, serão visíveis a partir do exterior da função. Esta situação pode propiciar duas situações desvantajosas: conflitos de nomes entre variáveis supostamente locais de funções diferentes; incentivo de práticas de programação desajustadas. Com efeito, pode haver a tentação de usar as variáveis para passar parâmetros para o interior da função — estaríamos, neste caso, perante variáveis globais. Posto de outra forma, a fronteira entre variáveis locais a uma função e globais a todo o programa pode passar a ser uma linha demasiado ténue com prejuízo para a clareza da programação. A programação com variáveis globais é vivamente desaconselhada porque limita a independência entre as partes constituintes dum programa. Isto é o mesmo que dizer que o programa será mais difícil de ler, corrigir e modificar. A outra desvantagem de usar variáveis locais em memória é mais subtil. Ocorre nos casos em que uma função possa ser chamada uma segunda vez antes de ter terminado a execução anterior. Isto ocorre nas funções recursivas, por exemplo [9]. Com a segunda execução, o conteúdo das variáveis necessário à primeira execução estará corrompido. A

[9] Mas não só. Caso um processo tenha múltiplos fios de execução, i.e., seja multitarefa, pode acontecer que dois desses fios de execução se encontrem na mesma função em simultâneo.

salvaguarda dos valores das variáveis para a pilha e posterior recuperação no fim da execução não resolve todos os casos em que há problemas de concorrência [10].

Voltando ao exemplo da adição dos registos `ebx`, `ecx` e `edx`, o código 9.3 representa uma situação em que a variável `soma` armazenada em memória serve para guardar as somas parciais da operação. O resultado é novamente devolvido no registo `eax`. Não se pode falar propriamente de variável local, uma vez que, em geral, a variável `soma` será visível também do exterior da função.

```
.lcomm soma, 4
...
soma_regs:
    movl %ebx, soma        M[soma] <-32- ebx
    addl %ecx, soma        M[soma] <-32- M[soma] + ecx
    addl %edx, soma        M[soma] <-32- M[soma] + edx
    movl soma, %eax        eax <-32- M[soma]
    ret
```

(9.3)

O terceiro método faz recurso à pilha. Este é o método mais elegante para situações em que o número de variáveis locais não permita a utilização de registos. O grande inconveniente é a sua relativa complexidade. A programação segundo este método é difícil, sobretudo para programadores pouco experientes. No entanto, uma vez compreendido o mecanismo, a sua utilização é sempre igual.

Atente-se novamente no exemplo da adição, ilustrado no código 9.1. Agora, em vez de se usar o registo `eax` como variável local, vai ser usada a pilha. O

[10] Precisamente nos casos em que há múltiplos fios de execução.

espaço para a variável local inteira é criado pela instrução `subl $4, %esp` (que deixa quatro *bytes* livres na pilha). O acesso a essa variável é feito com endereçamento relativo à base `ebp`, o que pressupõe a inicialização adequada deste registo.

A reserva do espaço e inicialização do registo `ebp` será então feita da forma representada no código 9.4. Note-se que é suposto que a função não altere o registo `ebp`, razão pela qual o valor desse registo é salvaguardado na pilha.

```
soma_regs:
    pushl %ebp          Salvaguarda registos
    movl %esp, %ebp     ebp <--(32)-- esp
    subl $4, %esp       esp <--(32)-- esp - 4
```
(9.4)

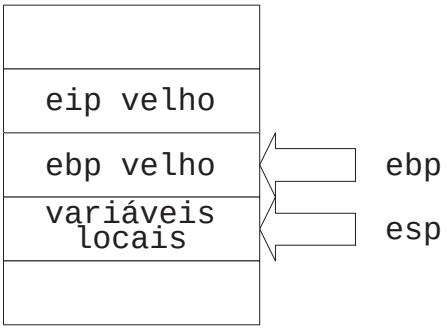

Figura 9.2: Utilização de variáveis locais na pilha

A situação resultante representada na figura 9.2 mostra que a variável local está armazenada 4 *bytes* abaixo do ponteiro base `ebp`, pelo que o acesso a essa variável se faz com a expressão `-4(%ebp)`. Esta variável vai desempenhar o papel que o registo `eax` tinha no código 9.1. A conclusão da função soma está representada no código 9.5.

```
soma_regs:
    pushl %ebp              Salvaguarda registos
    movl %esp, %ebp         ebp <--(32)-- esp
    subl $4, %esp           esp <--(32)-- esp - 4
    movl %ebx, -4(%ebp)     M[ebp - 4] <--(32)-- ebx
    addl %ecx, -4(%ebp)     M[ebp - 4] <--(32)-- M[ebp - 4] + ecx
    addl %edx, -4(%ebp)     M[ebp - 4] <--(32)-- M[ebp - 4] + edx
    movl -4(%ebp), %eax     eax <--(32)-- M[ebp - 4]
    ret
```

(9.5)

Neste exemplo, que é muito simples, a vantagem de usar a pilha para armazenar uma variável local não é muito evidente. Na verdade, esta solução será muito mais lenta do que a apresentada no código 9.1, porque inclui vários acessos à memória. No código 9.1 não existe qualquer acesso, visto que o valor da soma é armazenado em `eax`.

A vantagem de ter variáveis locais na pilha só será evidente quando o número de variáveis locais necessário e/ou a complexidade do código for maior. Facilmente se percebe que se forem necessárias 10 variáveis locais não será possível armazená-las nos registos.

Neste caso, o acesso a essas variáveis seria feito segundo a correspondência apresentada na figura 9.1, para as variáveis identificadas na figura 9.3.

var1	$\longleftrightarrow$	`-4(%ebp)`
var2	$\longleftrightarrow$	`-8(%ebp)`
	...	
var10	$\longleftrightarrow$	`-40(%ebp)`

Tabela 9.1: Tipos de números inteiros

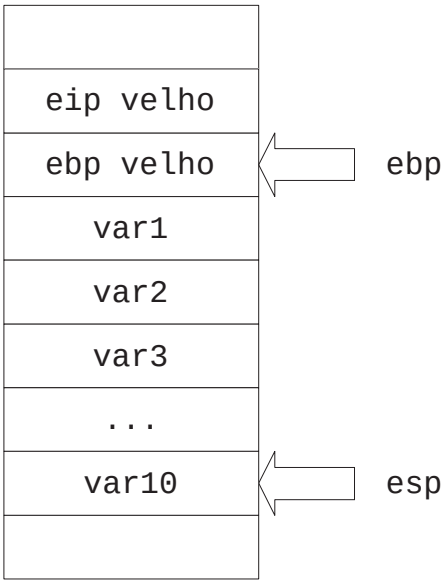

Figura 9.3: Função com muitas variáveis locais

Note-se que quando a programação é feita numa linguagem de alto nível, digamos em *Pascal* ou em *C*, no interior das funções são usados nomes para as variáveis locais, tais como `i`, `j` ou `var1`, por exemplo. Caberá depois ao compilador substituir esses nomes segundo uma correspondência idêntica a esta. Se a função for escrita directamente em assembly caberá ao programador manter, para sua própria orientação, uma tabela como esta, o que representa um trabalho bastante fastidioso.

9.8 Passagem de Parâmetros para uma Função

Uma função que produzisse sempre o mesmo resultado seria talvez pouco interessante. Por exemplo, a utilidade duma função que imprima a frase "Olá Mundo!" no monitor será, em princípio, muito reduzida. Tem que ser possível a uma função alterar o seu comportamento de forma controlada a partir do exterior. Uma forma de controlar o comportamento duma função

será a partir dos seus parâmetros [11]. Por exemplo, uma função matemática pode receber um ou mais parâmetros. Para calcular o seno é necessário um parâmetro, que é o ângulo.

Muitas vezes (mas nem sempre, veja-se a função `getchar` apresentada na secção 10.4) uma função necessita de receber parâmetros. À semelhança do que sucede com o armazenamento de variáveis locais, os parâmetros podem ser recebidos de três formas diferentes: por registo, por memória ou pela pilha. Os argumentos a favor duma ou de outra opção assemelham-se muito aos que foram usados aquando da mesma explicação para as variáveis locais. Justifica-se a utilização de registos em casos onde há poucos parâmetros a passar e os registos não fazem falta para construir a função. A utilização de memória, por sua vez, ultrapassa o problema de haver poucos registos, mas traz novos potenciais problemas que já foram descritos. Finalmente, a utilização de pilha é, em simultâneo, a solução mais flexível e mais complexa.

Este último mecanismo merece um olhar mais atento. Na figura 9.4 está representado o estado duma pilha no interior duma função que recebe dois parâmetros `nome1` e `nome2` e usa duas variáveis locais chamadas `vogais` e `consoantes` onde armazenará, respectivamente, o número de vogais e de consoantes existentes nos dois nomes. Esta pilha não é mais que uma extensão ao tipo de pilha que foi apresentada na figura 9.3, mas agora são considerados parâmetros, para além de variáveis locais. É de referir o papel fundamental que o ponteiro base `ebp` desempenha nesta situação, servindo de início de contagem para aceder quer às variáveis locais, quer aos parâmetros. O acesso a estes elementos é feito segundo a correspondência da tabela 9.2.

São ainda de notar dois aspectos: em primeiro lugar, a posição do ponteiro base `ebp` não tem que ser exactamente a posição que está representada

[11] Outra forma de alterar o comportamento duma função seria através de operações de E/S, como sejam leitura duma tecla ou do disco, etc.

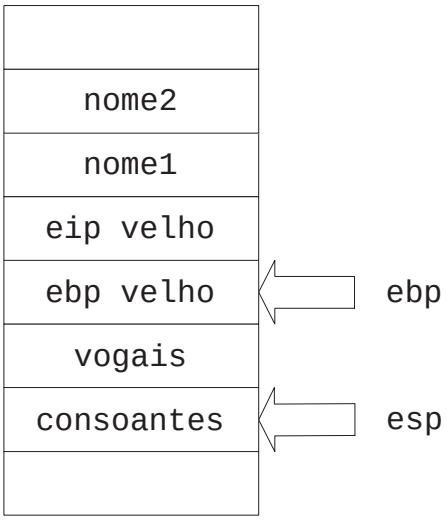

Figura 9.4: Passagem de parâmetros pela pilha

nome2	$\longleftrightarrow$	`12(%ebp)`
nome1	$\longleftrightarrow$	`8(%ebp)`
vogais	$\longleftrightarrow$	`-4(%ebp)`
consoantes	$\longleftrightarrow$	`-8(%ebp)`

Tabela 9.2: Acesso às variáveis locais e aos parâmetros

na figura 9.4. No entanto, se o ponteiro base `ebp` estiver num sítio ligeiramente diferente (quatro *bytes* mais abaixo, por exemplo), a forma de aceder às variáveis e aos parâmetros tem que variar de forma correspondente. Em segundo lugar, o registo `ebp` é usado para estas funções por razões histórias, anteriores ao aparecimento do 80386. A partir deste microprocessador, qualquer registo de utilização geral poderia desempenhar o mesmo papel.

Para melhor comparar os três mecanismos de passagem de parâmetros apresentados — registo, memória e pilha — considere-se como exemplo uma função que deverá determinar o maior de dois números e armazenar o resultado no registo `eax`. As soluções para o problema estão representadas nos fragmentos de código 9.6, 9.7 e 9.8, respectivamente — sobre este último

caso veja-se a figura 9.5, que ilustra o estado da pilha durante a execução da função.

Note-se que nos fragmentos de código 9.7 e 9.8 é utilizada uma variável local armazenada no registo `ebx`. No caso da passagem por registos, os números são enviados nos registos `ebx` e `ecx`; quando a passagem é por memória, em `num1` e `num2`; na passagem por pilha primeiro põe-se na pilha o último número.

```
maior_regs:
    cmpl %ecx, %ebx        ecx >= ebx ?
    jge maior1
    jmp maior2
maior1:
    movl %ebx, %eax
    jmp fim
maior2:
    movl %ecx, %eax
fim:
    ret
```
(9.6)

```
maior_mem:
    movl num1, %ebx
    cmpl num2, %ebx       num2 >= ebx ?
    jge maior1
    jmp maior2
maior1:
    movl %ebx, %eax
    jmp fim
maior2:
    movl num2, %eax
fim:
    ret
```

(9.7)

```
maior_pilha:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %ebx
    cmpl 8(%ebp), %ebx    num2 >= ebx ?
    jge maior1
    jmp maior2
maior1:
    movl %ebx, %eax
    jmp fim
maior2:
    movl 8(%ebp), %eax
fim:
    popl %ebp
    ret
```

(9.8)

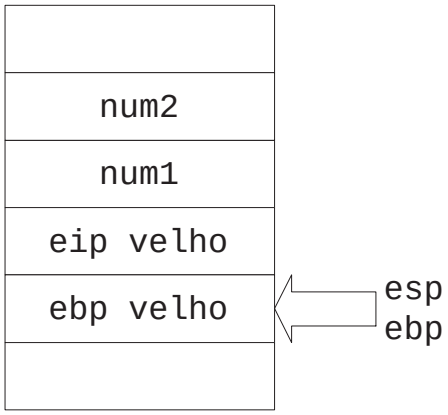

Figura 9.5: Parâmetros passados pela pilha

É bastante evidente, que o código 9.8 é mais complexo que o código 9.7 que, por sua vez, é mais complexo que o código 9.6.

No tocante à passagem de parâmetros, resta ainda ver como é que as funções deverão ser invocadas para cada um dos três mecanismos. Para se codificar a chamada às funções é necessário saber onde é que inicialmente estão armazenados os valores a passar nos parâmetros. No exemplo que tem vindo a ser tratado consideremos que o que está a ser comparado são duas idades armazenadas nas variáveis `idade1` e `idade2`. A chamada para os três casos está representada nos fragmentos de código 9.9, 9.10 e 9.11.

```
movl idade1, %ebx
movl idade2, %ecx                                    (9.9)
call maior_regs          resultado volta em eax
```

```
movl idade1, %eax
movl %eax, num1
movl idade2, %eax                                    (9.10)
movl %eax, num2
call maior_mem           resultado volta em eax
```

```
pushl idade2
pushl idade1
call maior_pilha          resultado volta em eax
addl $8, %esp             retirar parâmetros da pilha
```
(9.11)

Uma hipótese alternativa ao que se encontra no código 9.8 seria ter `ret 8` no fim do procedimento `maior_pilha` em vez de apenas `ret`. Nesse caso, não seria mais necessária a instrução `addl $8, %esp` no final da chamada ao procedimento, tal como se encontra em 9.11.

9.9 Retorno de Valores duma Função

No que concerne ao retorno de resultados do interior para o exterior duma função, há também múltiplas alternativas, entre elas as mesmas que temos vindo a ver: registos, memória e pilha. A utilização de registos até já foi exemplificada atrás, quando o registo `eax` foi usado para devolver o maior dos números.

A tabela 9.3, apresentada em (Hyde, 2003) representa a convenção que determina quais os registos que devem ser usados primeiro para devolver resultados.

Usar	Antes	Depois
Bytes:	`al`, `ah`, `dl`, `dh`, `cl`, `ch`, `bl`, `bh`	
Palavras:	`ax`, `dx`, `cx`, `si`, `di`, `bx`	
Inteiros:	`eax`, `edx`, `ecx`, `esi`, `edi`, `ebx`	

Tabela 9.3: Registos a utilizar na devolução de resultados

A utilização de memória é também muito simples, não merecendo mais explicações.

A utilização da pilha é, como sempre, mais complexa, mas uma forma de o fazer é deixando os resultados no topo da pilha após o término da função, como ilustrado na figura 9.6. A função chamante deverá retirar os resultados da pilha com uma ou mais instruções `pop` (duas neste caso). No entanto, para que esta técnica seja possível é necessário que a função chamante reserve o espaço necessário para a devolução dos resultados, antes de efectuar a chamada à função. Em certas ocasiões é possível também reutilizar o espaço dos parâmetros para esse efeito.

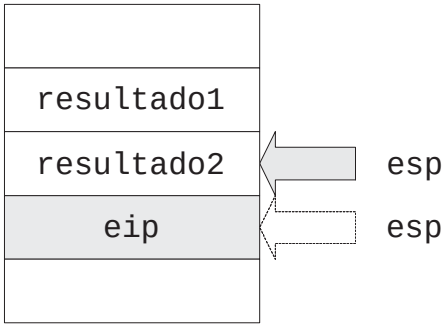

Figura 9.6: Devolução de resultados pela pilha

Caso também houvesse lugar à passagem de parâmetros pela pilha a situação seria ligeiramente mais complexa. Considere-se a existência duma função capaz de calcular o resultado da expressão $i \times (num1 + num2)$. No código 9.12 encontra-se o código desta função. No código 9.13 encontra-se o código a utilizar para efectuar a chamada à função e imprimir o resultado retornado. Na figura 9.7 está representado o estado da pilha. A cinzento estão todas as palavras que deverão desaparecer para aceder aos resultados. Entre parêntesis estão os nomes dos parâmetros a passar à função e, no caso dos valores devolvidos, os registos onde o resultado final foi armazenado inicialmente.

```
calcula:
    pushl %ebp              Prepara enquadramento
    movl %esp, %ebp         da pilha
    movl 12(%ebp), %eax     eax ⟵ num1
    addl 16(%ebp), %eax     eax ⟵ eax + num2
    imull 8(%ebp)           edx:eax ⟵ eax × i
    movl %edx, 24(%ebp)     Guarda resultado na pilha
    movl %eax, 20(%ebp)
    popl %ebp
    ret $12                 Retorna e retira parâmetros
                            da pilha
```

(9.12)

```
.section .rodata          Define variáveis
num1:  .int 20
num2:  .int 25
i:  .int 4
formato:  .string "edx = %d, eax = %d\n"

.text

.globl main

main:
    pushl %ebp            Salvaguarda ebp
    subl $8, %esp         Guarda espaço para os resultados
    pushl num2            Passa parâmetros para calcula
    pushl num1
    pushl i
    call calcula          Chama função calcula
    popl %eax             Obtém resultados
    popl %edx
    pushl %eax            Passa parâmetros para printf
    pushl %edx
    pushl $formato
    call printf           Chama função printf
    addl $12, %esp
    popl %ebp
    xorl %eax, %eax       Retorna o código de sucesso 0
    ret                   Termina programa
```

(9.13)

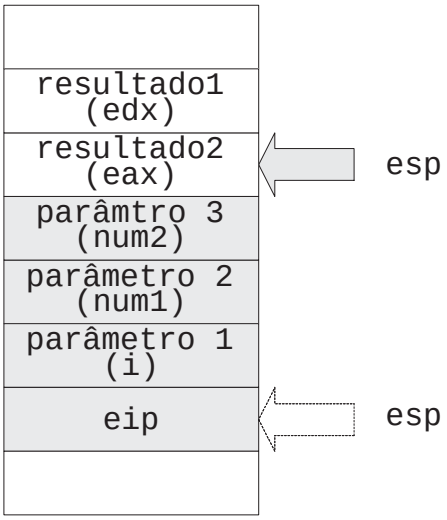

Figura 9.7: Devolução de resultados pela pilha com existência de parâmetros

Uma diferença relevante no problema do retorno de resultados, relativamente aos problemas apresentados anteriormente, de utilização de variáveis locais e de passagem de parâmetros é que na devolução de resultados será mais frequente a utilização de registos. Isto deve-se ao facto de haver muitas funções que devolvem resultados muito simples, do tipo 0 em caso de sucesso, -1 em caso de erro. A estas funções basta um registo, nomeadamente o `eax`, para o efeito. No capítulo 10 podem encontrar-se um conjunto de funções que o fazem.

Há, porém, muitas funções que devolvem resultados usando um mecanismo diferente de qualquer um dos que se descreveram (por exemplo a função `scanf` — ver secção 10.4.2). Neste caso, a função chamante tem que reservar um espaço em memória, que se destina a armazenar o resultado produzido pela função chamada. Para que esta saiba onde se encontra esse espaço, aquela deverá indicar a sua localização. Para isso utiliza um dos parâmetros,

que funciona como um ponteiro para a zona de memória em causa. Não raras vezes é também necessário acrescentar um outro parâmetro, que indica o espaço disponível nessa zona de memória. Este parâmetro adicional é dispensável, caso o resultado pretendido seja de tamanho fixo (como será o caso de um inteiro). Será no entanto necessário se o tamanho não tiver um limite implícito ou conhecido, como acontece com as cadeia de caracteres, por exemplo. No código 9.14 apresenta-se um exemplo do código a usar pela função chamante, estando ilustrada na figura 9.8 a situação vista pela função chamada. Após o retorno da função chamada o conteúdo da pilha pode ser destruído sem problemas, visto que o resultado terá sido convenientemente armazenado numa zona segura de memória. Apesar do exemplo utilizar a pilha para a passagem por referência, também poderia ter sido usado outro mecanismo de passagem de parâmetros.

```
.EQU tamanho, 20          tamanho ⟵ 20
pushl tamanho             passa tamanho
pushl $seunome            passa ponteiro
call perguntanome
addl $8, %esp             retira parâmetros da pilha
```
(9.14)

9.10 Instrução `enter`

```
enter literal1, literal2
```

$$\mathbf{esp} \underset{32}{\longleftarrow} \mathbf{esp - 4}$$

$$\mathbf{M[esp]} \underset{32}{\longleftarrow} \mathbf{ebp}$$

$$\mathbf{ebp} \underset{32}{\longleftarrow} \mathbf{esp}$$

$$\mathbf{esp} \underset{32}{\longleftarrow} \mathbf{esp - literal1}$$

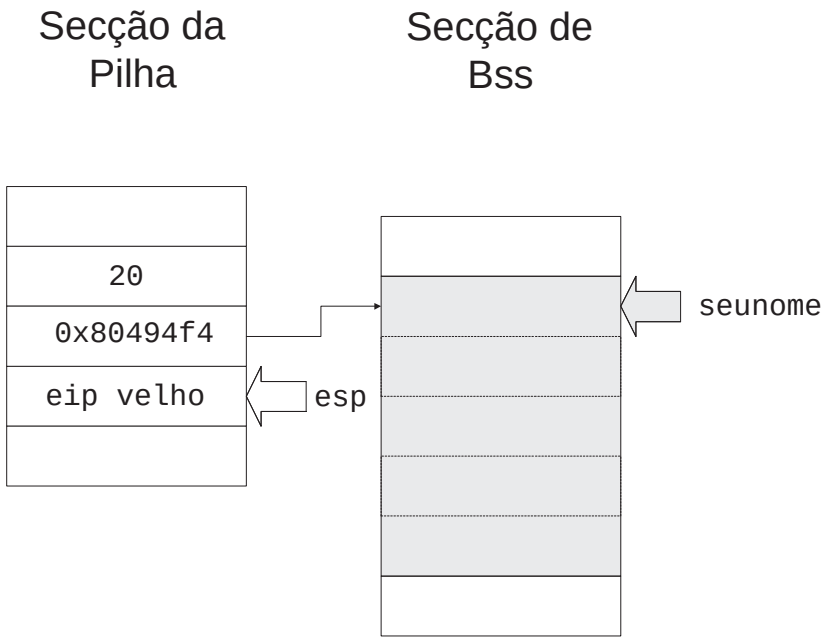

Figura 9.8: Passagem de parâmetros por referência

Existem três variantes da função `enter`. Numa delas o valor `literal2` assume o valor 0, noutra assume o valor 1 e na última o valor `literal2` pode ser qualquer número inteiro com 8 *bits*. O operando `literal1` tem sempre 16 *bits*.

A instrução `enter` prepara a pilha para a execução duma função. A preparação da pilha consiste em salvaguardar o ponteiro de base `ebp`, atribuir ao ponteiro `ebp` o valor de `esp` e subtrair o `literal1` a `esp`. O valor `literal2` é usado para indicar o nível de *nesting* da função. Para simplificar vamos sempre considerar que este nível é igual a 0. Para mais detalhes sobre esta instrução poderá consultar-se (IA-32 developer's manual-II, 2001).

Após a realização da instrução `enter $4, $0`, por exemplo (que por norma será invocada no interior duma nova função), dentro duma função serão reservados 4 *bytes* para variáveis locais enquanto os ponteiros `esp` e

ebp são preparados para aceder a essas variáveis e a eventuais parâmetros. A situação que se obtêm após esta instrução é a mesma que foi representada na figura 9.2.

9.11 Instrução leave

leave	$\mathbf{esp} \underset{32}{\longleftarrow} \mathbf{ebp}$
	$\mathbf{ebp} \underset{32}{\longleftarrow} \mathbf{M[esp]}$
	$\mathbf{esp} \underset{32}{\longleftarrow} \mathbf{esp + 4}$

A instrução leave é a instrução complementar da instrução enter. A instrução leave deverá ser executada no fim da função, imediatamente antes da instrução ret. Independentemente do estado em que se encontra a pilha, a instrução leave atribui o valor do ponteiro ebp ao ponteiro esp, o que liberta todo o espaço ocupado na pilha, incluindo as variáveis locais. Em seguida, a instrução leave faz o popl do registo ebp antigo, o que efectua imediatamente a mudança de enquadramento de volta para a função chamante. Finalmente, após a instrução leave, deverá existir uma função ret, opcionalmente ret literal, se se quiser libertar o espaço reservado para os parâmetros da função que termina.

10

Bibliotecas de funções

10.1 Introdução

As instruções mais importantes dos processadores x86 foram apresentadas nos capítulos 4, 5, 6 e 7. No capítulo 8 o tema foi a definição e declaração de variáveis para utilização nos programas; finalmente, no capítulo anterior foram postas em conjunto as partes que devem constituir um programa. Se é verdade que com esta informação (e com a adição de alguma experiência, talvez) é já possível escrever um programa completo, com princípio, meio e fim, a verdade é que continua a faltar um elo essencial: as entradas e saídas (E/S) do programa.

Para efectuar operações de E/S existem as instruções dedicadas `in` e `out`. Estas instruções permitem ler (`in`) e escrever (`out`) de e para portos do sistema. Para simplificar, pode pensar-se nos portos como sendo a visão que o microprocessador tem dos periféricos do sistema. Cada porto tem um endereço próprio [1] e a cada periférico é atribuído um conjunto de portos. Por exemplo, à primeira porta série (COM 1), usada muitas vezes pelo rato ou por modems, poderão estar atribuídos os portos com endereços 0x03f8 a 0x3ff, sendo que, no entanto, estes endereços são configuráveis no caso das portas série. Isto significa que é lendo desses portos que o processador pode

[1] Normalmente é usado o sistema de numeração hexadecimal para referir esses endereços.

saber qual a direcção do movimento do rato, no caso de ser um rato que está ligado na primeira porta série [2] — que deverá ser programado pelos gestores destes periféricos (*device drivers*).

Nalgumas plataformas rudimentares é possível a programas não privilegiados (em rigor, nesses sistemas nem sequer havia hierarquias de privilégios, como no caso do MS-DOS, da Microsoft) aceder directamente aos portos do sistema. No entanto, nos sistemas operativos modernos multitarefa e multiutilizador isso já não é possível, pois o privilégio requerido para ler ou escrever de portos está para além do privilégio possuído por um programa de um utilizador normal. Esta decisão justifica-se em pleno, porque não é admissível que um programa qualquer possa controlar os periféricos, fazendo-se assim substituir ao sistema operativo. Não é difícil imaginar que consequências traria o facto de ser possível a um programa de utilizador manipular directamente o controlador do disco.

Há ainda outra razão para que não seja desejável efectuar operações de E/S desta forma. Esta razão tem a ver com a falta de transparência destes processos, que requerem um conhecimento muito detalhado do funcionamento do *hardware* e uma complexidade considerável na sua programação, que não poderia ser evitada.

Dadas estas questões, a solução para o problema do privilégio resolve-se colocando todas as funcionalidades relativas às E/S no sistema operativo. Competirá depois ao sistema operativo em nome dos processos, efectuar essas operações. Cabe-lhe também verificar quando é que um determinado

[2] É de referir que as operações de E/S não se processam sempre desta forma. Alguns periféricos mais inteligentes e sobretudo mais rápidos, que requerem larguras de banda elevadas na comunicação com a memória central, conseguem retirar o microprocessador do caminho dos dados, deixando que este último se dedique a outras tarefas. Isto é possível graças a um outro microprocessador, chamado controlador de DMA — Acesso Directo à Memória, *Direct Memory Access*, em inglês

processo deve ou não poder efectuar a operação em questão. Esta verificação é fundamental, uma vez que o sistema operativo tem sempre o privilégio suficiente para isso. O facto de ser o sistema operativo a encarregar-se das E/S resolve também a questão da simplicidade que se deseja para estas operações. Com efeito, as funções para realizar operações de E/S que o sistema operativo disponibiliza — chamadas ao sistema — têm já uma interface e utilização relativamente simples, quando comparadas com a programação directa de portos. No entanto, além das chamadas ao sistema, existem muitas vezes funções (além de constantes e variáveis) com capacidades ainda mais elaboradas e que são postas à disposição do programador.

Estas funções estão reunidas em conjuntos designados de "bibliotecas". Note-se que estas bibliotecas podem também incluir muitas funções que não realizam operações de E/S. Uma biblioteca, tem algumas semelhanças com um programa executável, nomeadamente o facto de também conter código, variáveis, e alguns símbolos públicos, que permitem fazer o acesso a essas funções e variáveis. A principal diferença reside no facto de a biblioteca não ter uma função `main()`, que lhe permita ser executada autonomamente. Ao contrário, as funções existentes numa biblioteca têm de ser ligadas (pelo editor de ligações — veja-se figura 1.1) com outras funções. Entre o conjunto das funções que constituem o programa, apenas uma delas poderá ser a função `main()`. Isto implica que entre todos os ficheiros objecto existe apenas uma função `main()`.

Note-se que, muitas vezes, a biblioteca já tem uma existência anterior e não é sequer programada pela(s) mesma(s) pessoa(s) que escreve(m) a função `main()`. É o caso da biblioteca da linguagem de programação *C*, que oferece uma miríade de capacidades para controlar o acesso às E/S, para manipular cadeias de caracteres, etc.; da biblioteca `ncurses`, que oferece

funções, constantes e variáveis, que facilitam a programação de aplicações visuais em modo texto; da biblioteca *pthreads*, que permite fazer programação multitarefa, etc. — não há limites. Pode-se admitir sem hesitações, que o esforço de escrever um programa é muito significativamente reduzido pela existência de bibliotecas.

10.2 Bibliotecas Dinâmicas

A situação descrita na secção anterior, em que funções de biblioteca(s) são ligadas com o resto do programa para construir o executável acontece quando são utilizadas bibliotecas estáticas. Admita-se, por exemplo, a existência duma função `seno()` numa destas bibliotecas. Se forem escritos e armazenados no disco dum computador vinte programas que façam o cálculo de senos e todos forem ligados com a biblioteca que contém os senos, isso significa que todos os *bytes* do código da função `seno()` se encontram repetidos no disco vinte vezes, em cada um desses programas.

Uma alternativa melhor seria não incluir toda a função `seno()` no código dos programas, mas incluir apenas o código necessário à sua invocação, ficando a função `seno()` armazenada num único local. Depois, de alguma forma, o código da função `seno()` teria de ser trazido para o espaço de endereçamento do processo que corre o programa — a forma exacta de o fazer extravasa o âmbito deste texto. Este mesmo raciocínio estende-se naturalmente a qualquer outra função armazenada numa biblioteca.

Este tipo de bibliotecas, onde as funções não são ligadas com o programa, sendo carregadas apenas quando o programa começa são designadas de bibliotecas dinâmicas.

As figura 10.1 e 10.2 procuram ilustrar a relação entre ficheiros objecto,

executável, bibliotecas estáticas e dinâmicas.

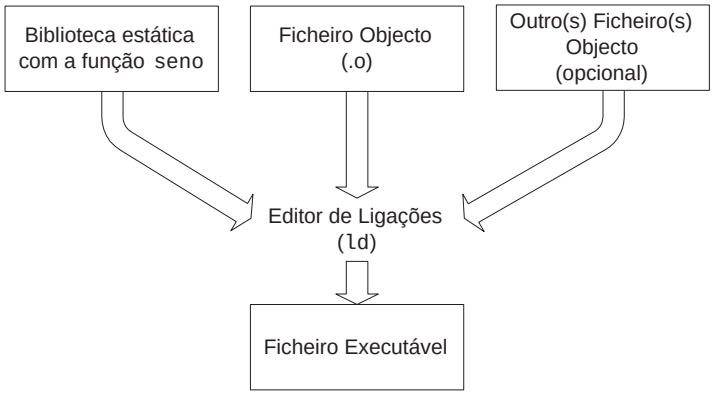

Figura 10.1: Construção e execução dum programa que utiliza uma biblioteca estática

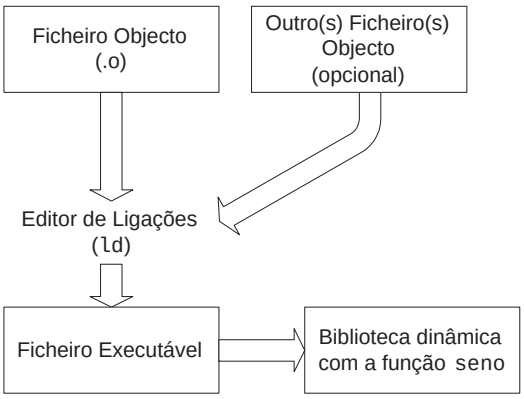

Figura 10.2: Construção e execução dum programa que utiliza uma biblioteca dinâmica

Há ainda duas vantagens adicionais a apontar às bibliotecas dinâmicas. A primeira, e talvez mais importante, é que permitem salvaguardar a memória física do sistema, uma vez que são carregadas para uma zona de memória passível de ser partilhada por vários programas. A segunda prende-se com o facto de ser possível actualizar as bibliotecas sem ser necessário recompilar os programas que as utilizam.

10.3 Ferramentas de Manipulação de Bibliotecas

No Linux os ficheiros das bibliotecas são armazenados no formato *elf* — *Executable and Linking Format* também conhecido por *Extended Linker Format* —, que é também o formato dos ficheiros objecto e dos ficheiros executáveis (o formato utilizado antes da existência do *elf* era conhecido por *a.out*). Uma especificação detalhada do formato *elf* também está fora do âmbito deste texto, embora seja interessante perceber como se faz a distinção entre um ficheiro executável e uma biblioteca. Essa distinção faz-se no cabeçalho ELF do ficheiro onde, entre outras coisas, se encontra o tipo de ficheiro em questão. Para uma abordagem mais extensiva do assunto veja-se (Card *et al.*, 1998).

No que diz respeito a ferramentas para construir e manipular bibliotecas referenciam-se aqui as seguintes:

- `ar` — constrói arquivos. Mais exactamente, o `ar` permite juntar ficheiros objecto a uma biblioteca previamente existente ou construir uma biblioteca de raiz;
- `objdump` — mostra informação de ficheiros objecto;
- `nm` — inspecciona símbolos de ficheiros objecto (incluindo executáveis e bibliotecas). O `nm` indica o nome e a localização ou tamanho de todos os símbolos existentes no ficheiro. Indica também se os símbolos são locais, i.e., apenas visíveis dentro do ficheiro objecto, ou se são globais, i.e., se a biblioteca exporta os símbolos para outros programas, por exemplo. Note-se que no caso das bibliotecas (tal como nos ficheiros objecto, naturalmente) a localização dos símbolos não é final, sendo

portanto relativa ao início da biblioteca (posição 0);

- `ldd` — mostra todas as bibliotecas partilhadas de que o programa depende. Esta funcionalidade pode ser particularmente relevante numa situação em que seja necessário saber de que bibliotecas partilhadas uma máquina precisa para se poder executar um programa;

- `strace` — assinala todas as chamadas ao sistema [3] efectuadas por um programa em execução.

A compreensão de alguns detalhes relativos às bibliotecas pode ser melhorada, experimentando o comando `nm`. Para ilustrar esse facto uma pequena amostra do resultado de executar o comando `nm /lib/libc.so.6` está representado na tabela 10.1. Para descobrir qual é e onde está a biblioteca do *C* basta executar o comando `ldd` sobre um executável vulgar escrito em *C*, para que esta biblioteca apareça na lista das dependências. Na verdade, o

Endereço	Tipo	Nome
000eefe0	B	errno
00052ef0	T	getchar
0004cf4c	T	printf
000506f4	T	scanf

Tabela 10.1: Execução do comando `nm /lib/libc.so.6`

resultado da execução do comando ultrapassou as 40.000 linhas de texto, das quais só são apresentadas quatro, que dizem respeito a três funções muito conhecidas e a uma variável (`errno`). Os tipo dos símbolos são, neste caso, representados por letras maiúsculas porque todos eles são globais, i.e., utilizados fora do contexto do ficheiro da biblioteca. Caso fossem locais a este ficheiro seriam utilizadas letras minúsculas. O `B` no caso da variável `errno`

[3] Na verdade, também assinala a recepção de sinais.

indica que esta variável se encontra na secção de símbolos não incializados (bss). O `T` no caso das três funções indica que as funções se encontram definidas na secção de texto da biblioteca. Na primeira coluna da tabela encontram-se os endereços de cada um dos símbolos, relativamente ao início da biblioteca.

10.4 Biblioteca do *C*

Entre as bibliotecas referidas neste texto interessa-nos em particular a biblioteca do *C*. Será importante o conhecimento das seguintes funções, a descrever de seguida, pois permitem efectuar um conjunto de operações de E/S de forma relativamente simples:

- `printf()`
- `scanf()`
- `getchar()`

10.4.1 Função `printf()`

Esta função permite imprimir texto na saída padrão (que normalmente será o monitor). A sua utilização faz-se através de, pelo menos, um parâmetro sendo todos os outros opcionais, segundo o seguinte protótipo:

```
int printf(formato, valor1, valor2, ...);
```

Esta representação, que foi simplificada relativamente à verdadeira representação em *C*, indica que a função devolve um inteiro e recebe um número arbitrário de parâmetros. O primeiro dos parâmetros chama-se *formato* e os que se lhe seguem são valores. No parâmetro `formato` é indicado um endereço de memória com uma *string*, que indica não só o conteúdo dos

restantes parâmetros, como também o aspecto com que os dados deverão ser impressos. De seguida dão-se alguns exemplos de utilização do formato, mas para uma referência completa veja-se a página de manual do `printf()` (`info printf`) ou o livro (Kernighan & Ritchie, 1988).

- uma frase, terminada com um `0` (caracter `'\0'` = 0, que é diferente de `'0'` = 48 no código ASCII). Por exemplo, `printf("Olá")` [4];
- `"%s"` (caracter `'%'`, caracter `'s'`, caracter `'\0'`) para indicar que nos argumentos seguintes é dado um endereço de memória onde se encontra uma frase que tem de terminar com `'\0'`. Por exemplo, `printf("%s", ptrfrase)`;
- `"%d"` para indicar que se segue um número inteiro de 32 bits. Por exemplo, `printf("%d", idade)`;
- O caracter `'\n'` permite efectuar mudanças de linha. Por exemplo, `printf("Olá!\nTenho %d anos", idade)`;

No caso destas funções os parâmetros são colocados na pilha pela ordem inversa em que aparecem no protótipo, i.e., o último parâmetro é colocado na pilha primeiro e o formato é o último parâmetro a entrar.

10.4.2 Função `scanf()`

Esta função permite fazer a leitura de dados da entrada padrão (teclado, normalmente). A sua utilização, bem como o seu protótipo são semelhantes à função `printf()`:

```
int scanf(formato, endereço do valor1, endereço do valor2, ...);
```

[4] Normalmente o valor de retorno do `printf()` é ignorado, porque a probabilidade de ocorrência de algum problema se assume como inexistente.

Há, no entanto, algumas diferenças a considerar. Em primeiro lugar, no parâmetro formato, não faz sentido indicar uma frase de texto para imprimir na saída padrão (este princípio inclui naturalmente o caracter de mudança de linha - `'\n'` - que não deve ser usado). Depois, quando se pretende ler um inteiro [5] (formato `"%d"`), tem de se passar no segundo parâmetro da função, o endereço de memória onde o inteiro deverá ser armazenado e não o inteiro propriamente dito, como será mais ou menos intuitivo, uma vez que esse é o valor que se deseja obter, não sendo, portanto, conhecido de antemão. Não seria portanto adequado efectuar uma passagem de parâmetro por valor. No caso de se efectuar a leitura de uma frase (*string* — `"%s"`) a partir da entrada padrão para uma zona de memória, a função `scanf()` garante que a frase termina sempre com um `'\0'`.

10.4.3 Função `getchar()`

O protótipo da função `getchar()`, em termos de linguagem *C*, é o seguinte:

```
int getchar(void);
```

Esta função lê o próximo caracter da entrada padrão e devolve-o como um inteiro ou devolve o código `EOF` (-1) em caso de erro ou de não existirem mais dados disponíveis a partir da entrada padrão.

10.4.4 Exemplo de Utilização das Funções da Biblioteca do C

```
.EQU TAMNOME, 1024
.EQU TAMINTEIRO, 4
```

[5] O mesmo se aplica a outro tipo de dados qualquer.

```
.lcomm nome, TAMNOME
.lcomm idade, TAMINTEIRO

.section .rodata

perguntaNome:     .string "Como se chama? "
perguntaIdade:     .string "Quantos anos tem? "
perguntaFinal:     .string "Quer continuar (s em caso afirmativo)? "

formatoNome:     .string " %s"
formatoIdade:     .string " %d"
resposta:     .string "%s tem %d anos de idade.\n"
limpabarran:     .string "%*[\n]"

.text

.globl main

main:
     pushl %ebp
inicio:
     pushl $perguntaNome
     call printf
     addl $4, %esp
     pushl $nome
     pushl $formatoNome
     call scanf
```

```
addl $8, %esp

pushl $perguntaIdade
call printf
addl $4, %esp
pushl $idade
pushl $formatoIdade
call scanf
addl $8, %esp

pushl idade
pushl $nome
pushl $resposta
call printf
addl $12, %esp

pushl $perguntaFinal
call printf
addl $4, %esp
pushl $limpabarran        #senão não permite resposta
call scanf
addl $4, %esp
call getchar
cmpl $'s', %eax
je inicio

popl %ebp
```

```
xorl %eax, %eax
ret
```

11

Interrupções e Chamadas ao Sistema

11.1 Interrupções

Suponhamos que estamos a desenvolver uma rotina para fazer a leitura de valores introduzidos pelo teclado. Uma solução ingénua para este problema consistiria em consultar o teclado periodicamente para saber se o utilizador tinha premido ou não uma tecla. Nesta abordagem, o processador teria periodicamente de pôr de lado outras tarefas para ir controlando o teclado. A esta acção de, repetidamente, se ir verificar se um dispositivo "está pronto" chama-se, à falta de melhor termo em português, "polling". Embora o conceito de *polling* seja simples, na prática este tipo de programação não é muito eficiente — será mais interessante ocupar o processador com tarefas melhores do que verificar periodicamente uma dúzia de periféricos, que em geral não terão dados novos.

Felizmente existe uma outra abordagem mais eficaz: em vez de ser o processador a verificar periodicamente o teclado, é este que avisa o processador de que há uma nova tecla disponível. Nesta segunda solução poderíamos ter uma rotina que permaneceria adormecida durante a maior parte do tempo, sendo depois acordada a pedido do "periférico" *apenas* quando necessário. Desta forma, logo que termine a rotina, o processador ficará disponível para

outras tarefas. Esta solução baseia-se no no conceito de "interrupção", cuja ideia fundamental consiste em permitir a um periférico indicar um determinado evento ao processador. Este evento poderá ser a pressão duma tecla, o movimento do rato ou a leitura de dados provenientes duma placa de rede. É de salientar, no entanto, que esta alternativa requer suporte especial por parte do *hardware*. Posto de um modo mais formal, e de acordo com (IA-32 developer's manual-III, 2001), uma interrupção é um evento que transfere a execução de um programa ou tarefa para uma rotina especial, chamada "handler" (que poderemos designar por "rotina de tratamento").

Um conceito que está relacionado com o de interrupção e que também é abrangido pela definição anterior é o de "excepção". Tal como uma interrupção, uma excepção também transfere o controlo para uma rotina de tratamento. A diferença entre interrupção e excepção encontra-se na natureza do evento que é gerado. Numa interrupção este evento é *assíncrono*, isto é, trata-se de um acontecimento não previsível, por não estar relacionado com o código que está a ser executado (por exemplo devido a um erro de paridade na memória). Pelo contrário, uma excepção é um acontecimento síncrono, que ocorre de forma previsível num determinado local do código. Há três tipos de excepções: faltas[1], "traps" e "aborts"[2].

Uma falta é uma excepção que pode ser corrigida e que, caso o seja, permite ao programa continuar a sua execução normal. Isto poderá suceder, por exemplo num erro com uma operação aritmética tal como uma divisão por zero. Uma *trap* é uma excepção que executa a rotina de tratamento no fim de apenas uma instrução do processador (chamada "trapping instruction").

[1] Outra tradução possível para a palavra inglesa "fault" seria "falha". Embora diferentes autores utilizem as palavras "falta" e "falha" como sinónimos, quando utilizadas no mesmo texto estas palavras têm geralmente um significado bem diferente.

[2] A opção de não traduzir nenhum destes termos é deliberada, uma vez que a palavra correspondente em português reduziria consideravelmente a legibilidade deste texto.

Este tipo de excepções permite fazer a depuração de um programa. Após a execução de cada instrução do programa o depurador retoma o controlo, permitindo ao utilizador analisar variáveis, ler zonas da memória do programa, alterar a próxima instrução, etc. Um *abort* é uma excepção que não permite o retorno ao programa e pode ser causado, por exemplo, por um erro de *hardware* (será o caso de um erro de paridade na memória). Estes três tipos de excepções (faltas, *traps* e *aborts*) são gerados pelo próprio processador IA-32. Há ainda um outro tipo de excepções que são gerada pelo próprio programa, através das intruções `int` ou `int 3`. Este último tipo de excepções também é designado por "interrupções de *software*" (ver (IA-32 developer's manual-III, 2001)).

Uma vez que há diferentes eventos que podem causar interrupções, cada tipo de evento é identificado por um número diferente, conhecido por "vector de interrupção". Desta forma, o sistema operativo pode manter em memória uma tabela que associa a cada número de vector de interrupção uma rotina de tratamento. Isto permite determinar qual a rotina de tratamento do rato, e qual a rotina de tratamento do teclado, por exemplo. Dada esta tabela, chamada "tabela de descritores de interrupções" (*Interrupt Descriptor Table*), a acção do processador em caso de interrupção é sempre a mesma: procurar a entrada que corresponda ao número do vector de interrupção em causa e invocar a rotina de tratamento respectiva. Nesta consulta, o processador recorre a um registo chamado "registo da tabela de descritores de interrupções" (*Interrupt Descriptor Table Register*). Este registo aponta para a tabela de descritores de interrupções, sendo que, em simultâneo, indica também o seu limite que poderá ir até aos 256 elementos. O registo usa 32 *bits* para a localização em memória da tabela mais 16 *bits* para o número de elementos da tabela. Cada elemento da tabela consiste num "porta"

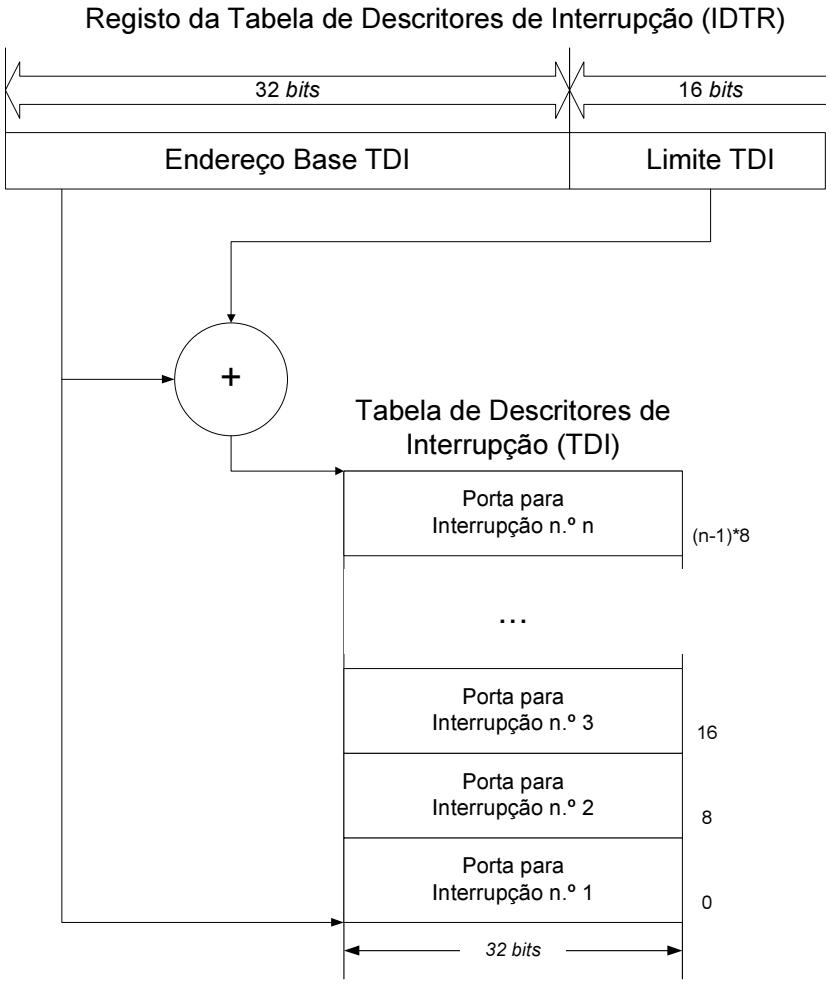

Figura 11.1: Acesso à tabela de descritores de interrupção

que invoca a interrupção respectiva com as permissões adequadas. Isto está representado na Figura 11.1. Por exemplo, quando o processador recebe a interrupção número 3, tem de procurar a "porta" que está na 3^a posição do tabela de descritores de interrupções.

No passado, noutros sistemas operativos mais rudimentares, como o MS-DOS, o processador não corria no modo protegido, mas sim no modo real (endereçamento real). Neste caso, a tabela de interrupções encontrava-se localizada no endereço `0` de memória (Hyde, 2003) e cada posição deste vector

continha directamente o endereço da rotina de tratamento. Na Tabela 11.1, encontra-se uma lista com as intrrupções da arquitectura IA-32. Esta tabela pode ser encontrada em (IA-32 developer's manual-III, 2001). INM significa "Interrupção Não Mascarável". Uma INM é uma interrupção, que pode interromper todas as interrupções de *software* e as de dispositivos de *hardware* não vitais. Ao contrário do que sucede com as interrupções que têm uma prioridade associada, uma INM nunca pode ser ignorada.

De uma forma geral, num PC as interrupções não são geridas pelo processador central, mas por um processador associado, o Intel 8259A, que é um Controlador de Interrupções Programável (CIP). O CIP pode receber até 8 interrupções provenientes de periféricos diferentes, através de entradas chamadas de "requisição de interrupção" (IRQ). Por sua vez, o CIP tem uma saída que é conectada ao pino INTR do processador. Neste pino, o CIP indica ao processador que há um interrupção pendente, usando o *bus* de sistema para indicar ao processador central qual é o número do vector de interrupção em causa. Uma característica interessante do CIP 8259A é que a um pino IRQ do 8259A pode ser ligado outro 8259A, sendo assim possível encadear até oito 8259A num nono 8259A que será ligado ao processador. Neste caso é possível disponibilizar até 64 requisições de interrupção diferentes (8 por cada 8259A, com excepção do que faz a conexão dos restantes). Na prática, os PCs usam dois 8259A, sendo o segundo ligado à IRQ 2 do primeiro. Assim, as IRQs 0-1 e 3-7 são da responsabilidade do primeiro 8259A, enquanto que as IRQs 8-15 são da responsabilidade do segundo 8259A (ver Figura 11.2). A IRQ 2 serve apenas para fazer o encadeamento. Como as interrupções nos pinos IRQ com número mais baixo têm prioridade mais elevada, a ordem real das prioridades é: 0-1-8-9-10-11-12-13-14-15-3-4-5-6-7. No Linux as IRQs são individualmente atribuídas a números de interrupção disponíveis entre o 32

Tabela 11.1: Tabela de interrupções e excepções em modo protegido

N.º vector	Descrição	Tipo	Causa
0	Erro de Divisão	falta	Instruções `div` e `idiv`
1	Depuração	falta/*trap*	Qualquer referência a código ao a dados ou a instrução `int 1`
2	Interrupção "não mascarável"	interrupção	Origem exterior "não mascarável"
3	*Breakpoint*	*trap*	Instrução `int 3`
4	Transbordo	*trap*	Instrução `into`
5	Violação de limites (*bound range exceeded*)	falta	Instrução `bound`
6	Código de instrução inválido (não definido)	falta	Instrução `UD2` ou código de instrução reservado
7	Co-processador matemático ausente	falta	Instrução de vírgula flutuante ou instruções `wait`/`fwait`
8	Dupla falta	*abort*	Qualquer instrução que gere uma excepção, uma INM ou uma INTR.
9	Ultrapassagem do segmento do co-processador	falta	Instrução de vírgula flutuante (os processadores IA-32 após o 386 não geram esta falta)
10	Selector de Segmento de Tarefa (TSS) inválido	falta	mudanção de tarefa ou acesso a um TSS
11	Segmento ausente	falta	Carregamento de registos de segmento ou acesso a segmentos do sistema
12	Falta no segmento da pilha	falta	Operações na pilha e carregamentos do registo SS
13	Protecção geral	falta	Qualquer referência a memória ou outras verificações
14	Falta de página	falta	Qualquer referência a memória
15	Reservado		
16	Erro na unidade de vírgula flutuante x87 (Erro matemático)	falta	Erro de vírgula flutuante no x87 ou instruções `wait`/`fwait`
17	Erro de alinhamento	falta	Qualquer referência a dados em memória (só a partir do processador 486)
18	Verificação da máquina (*machine check*)	*abort*	Depende do modelo
19	Excepção de vírgula flutuante numa instrução *Single Instruction Multiple Data*	falta	Instruções de vírgula flutuante `SSE` e `SSE2`
20-31	Reservadas pela Intel		
32-255	Interrupções definidas pelo utilizador	interrupção	Interrupção externa ou instrução `int n`

Tabela 11.2: Correspondência entre IRQs e números vector de interrupção

IRQ	Interrupção	Periférico	Fixa/Variável
0	32	Temporizador	Fixa
1	33	Teclado	Fixa
2	34	encadeamento do CIP	Fixa
3	35	Porta série n.^{o} dois	Variável
4	36	Porta série n.^{o} um	Variável
6	37	*Drive* de disquetes	Fixa
8	40	Relógio do Sistema	Fixa
11	43	Placa de rede	Variável
12	44	Rato PS/2	Variável
13	45	Co-processador matemático	Fixa
14	46	Primeiro controlador de disco EIDE	Variável
15	47	Segundo controlador de disco EIDE	Variável

e 47. Esta correspondência é mostrada na Tabela 11.2. Algumas das IRQs são fixas, outras podem ser alteradas.

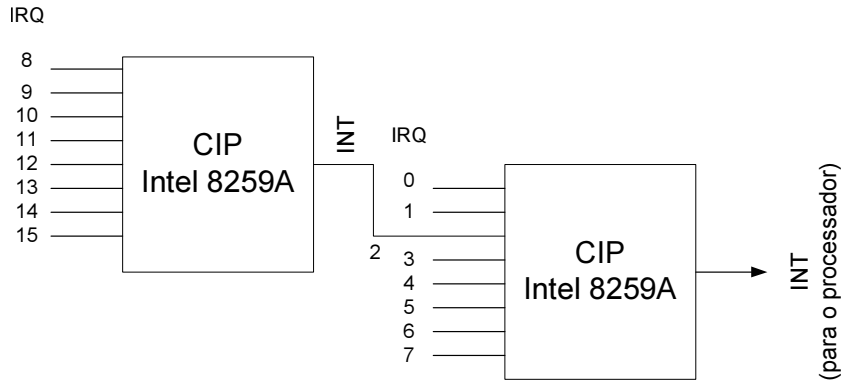

Figura 11.2: Encadeamento de controladores de interrupções

Para fazer a gestão das interrupções, incluindo a invocação de interrupções de *software*, existe um conjunto de instruções, das quais apresentamos de seguida as mais importantes.

11.2 Instrução `int`

```
int literal
```

Gera uma interrupção de *software*.

Esta instrução gera uma interrupção de *software*, que resulta numa chamada à rotina de tratamento do número de vector de interrupção indicado pelo literal (como se encontra descrito na Secção 11.1). A instrução `int` é muito semelhante à instrução `call`, visto que na verdade, ambas as instruções provocam uma alteração ao registo `eip`. Mais precisamente a semelhança com a instrução `call` dá-se no caso de chamadas a funções remotas, isto é que estão (ou poderão estar) num segmento diferente. Neste caso, o registo de segmento `cs` também é guardado na pilha e depois alterado, à semelhança do que sucede com o `eip`. Por outro lado, as instruções diferem em dois aspectos. Em primeiro lugar, no caso da instrução `int`, o programador não tem alternativa quanto ao endereço da função que vai executar: este está predeterminado na porta que corresponde ao número do vector de interrupção (ver Figura 11.1). Além disto, a instrução `int` salvaguarda o registo `eflags` na pilha antes de invocar a rotina de tratamento. A instrução `int` salvaguarda primeiro o registo `eflags`, só depois salvaguardando os registos `cs` e `eip`. Isto significa que a pilha fica num estado diferente daquele em que uma instrução `call` a deixa (mesmo que no caso de chamadas remotas). Por esta razão, uma rotina de tratamento de uma interrupção não pode terminar com a instrução `ret`, uma vez que esta deixaria ficar o registo `eflags` na pilha.

11.3 Instrução `iret`

```
iret
```

Retorna de uma rotina de tratamento.

Esta instrução termina uma rotina de tratamento e devolve o controlo ao programa que foi interrompido. Retira os valores para os registos `eip`, `cs` e `eflags` da pilha e devolve o controlo ao programa ou procedimento interrompido. A diferença relativamente à instrução `ret` reside no facto de o retorno duma interrupção retirar também o registo `eflags` da pilha.

11.4 Instrução `cli`

`cli` $\qquad\qquad$ **IF** $\underset{1}{\longleftarrow}$ **0**

Limpa a *flag* de interrupções IF.

Esta instrução faz com que o processador ignore interrupções externas mascaráveis. Esta instrução não afecta as interrupções não mascaráveis e as excepções geradas pela execução de *software*. O objectivo desta instrução é o de permitir que no interior duma rotina de tratamento não seja possível iniciar uma nova rotina de tratamento de uma outra interrupção. Assim, uma determinada interrupção pode ser tratada até ao seu final antes da próxima. Naturalmente que isto implica que uma boa rotina de tratamento de interrupções demore a menor quantidade de tempo possível, uma vez que todas as outras interrupções ficarão pendentes durante o espaço de tempo em que esta *flag* se encontra desactivada.

11.5 Instrução `sti`

`sti` $\qquad\qquad$ **IF** $\underset{1}{\longleftarrow}$ **1**

Activa a *flag* de interrupções IF.

Tipicamente, esta instrução é executada antes do `iret` que termina uma rotina de tratamento duma interrupção. Esta instrução não tem efeito imediato, sendo ainda executada mais uma instrução antes da *flag* ser activada. Isto permite que a rotina de tratamento termine após a instrução `iret`, antes de ser servida a próxima interrupção.

As instruções `sti` e `cli` só podem ser executadas em determinados níveis de privilégio [3], i.e., um programa a correr em modo de utilizador normal num sistema operativo como o Linux não as pode executar, uma vez que inibir ou permitir interrupções são decisões que podem comprometer toda a máquina e não apenas um processo.

11.6 Chamadas ao Sistema

De forma a utilizar os recursos da máquina, o sistema operativo disponibiliza um conjunto de funções, habitualmente designadas por "chamadas ao sistema". Sendo o sistema operativo que controla os recursos disponíveis na máquina e sendo, por norma, necessário utilizar o sistema operativo para aceder a esses recursos, facilmente se compreenderá que o conjunto de chamadas ao sistema existente tem que ser bastante completo. Por exemplo, o sistema operativo oferece uma chamada ao sistema que permite escrever num ficheiro, outra que permite ler, um outra que permite ler o valor do relógio, uma para obter informação sobre o processo que executa o programa, entre muitas outras. Por exemplo, na versão 2.2 do Kernel do Linux havia cerca de 190 funções destas.

Para acederem a estas funções, os programas têm de usar a interrupção `0x80`. Para o Linux a interrupção `0x80` está, como estava para o MS-DOS a

[3] Mais exactamente, quando o privilégio do processo em execução for igual ou superior ao privilégio indicado na *flag* `IOPL` - *Input/Output Privilege Level*.

famosa interrupção `0x21`. O sistema operativo guarda uma tabela com a lista completa das suas chamadas e quando o programa executa uma instrução `int $0x80` indica ao sistema operativo qual a chamada ao sistema que pretende invocar. Para isso utiliza o registo `eax`. Por exemplo, para invocar a 1^a chamda ao sistema (de nome `exit()`), o programa coloca o valor do registo `eax` a `1` e invoca a interrupção `80h`. Na prática, o registo `eax` é utilizado para indexar uma tabela onde se encontram todas as chamadas ao sistema. Os parâmetros da chamada ao sistema são passados nos registos `ebx`, `ecx`, `edx`, `esi` e `edi`, num máximo de cinco. A função número `117` (`sys_ipc`) necessita de seis parâmetros. Neste caso, o registo `ebx` aponta para um vector em memória com os seis parâmetros. Todas as chamadas ao sistema devolvem um resultado através do registo `eax`.

No fundo, a utilização da interrupção `0x80` resolve o problema de fornecer funções do sistema operativo aos programas do utilizador. O programa tem de invocar estas funções através do mecanismo de interrupções porque as portas existentes na tabela de descritores de interrupção (ver Figura 11.1) permitem alterar o nível de privilégio do programa utilizador, única e exclusivamente para o fim com que o utilizador invoca uma função do sistema, por exemplo, para aceder a dados que estão no disco. Por outras palavras, quando entra em modo privilegiado através da porta correspondente ao vector de interrupção `0x80`, o processo do utilizador apenas pode efectuar um conjunto de operações muito restrito, que estão previstas numa das mais de 190 funções oferecidas pelo sistema operativo. Repare-se que isto não poderia ser feito através de bibliotecas de funções como aquelas que foram apresentadas no Capítulo 10. As funções destas bibliotecas são executadas no mesmo nível de privilégio do programa do utilizador. Não permitiriam,

por exemplo, aceder ao disco[4].

Os seguinte programas exemplificam a utilização de chamadas ao sistema. O primeiro programa escreve uma frase na saída padrão do processo (em geral o monitor). Para isso, utiliza a chamada ao sistema número `4`, chamada `write()`, que recebe três parâmetros. O primeiro é o descritor do ficheiro onde o sistema operativo deve escrever. No caso deste programa esse descritor tem o número `1`, que corresponde à saída padrão (`stdout`, geralmente o monitor). O segundo parâmetro é um ponteiro para a zona de memória onde se encontram os dados a escrever, que neste caso é a frase `strteste`. Finalmente, o último parâmetro é o número de *bytes* que a função deverá escrever no ficheiro (e que, portanto, devem corresponder a algum conteúdo que se encontra presentemente na zona de memória indicada). Este valor é calculado como a diferença entre a posição actual do assemblador e a posição inicial da frase `strteste`. Note-se que esta frase não necessita de ser terminada (nem sequer deveria) com um '`\0`', graças a este terceiro parâmetro que indica inequivocamente o final da frase. Preenchidos estes parâmetros é invocada a chamada ao sistema através da instrução `int $0x80`. Para simplificar o programa ignora o valor retornado pela chamada ao sistema `write()`. Chama-se, no entanto, a atenção do leitor para o facto de ser um boa prática de programação fazer o tratamento do valor de retorno. Em particular, a função `write()` retorna `-1` em caso de erro, que poderá ocorrer por muitas razões diferentes. Por exemplo, se o descritor de ficheiro passado no primeiro parâmetro não corresponder, de facto, a qualquer descritor válido.

```
.section .rodata
```

[4]Naturalmente que o poderão fazer, mas apenas porque elas próprias utilizam internamente chamadas ao sistema.

```
# É irrelevante a string terminar ou não com um \0
strteste:   .ascii "Este programa usa a chamada ao sistema write!\n"
# Comprimento da string em bytes
comprstr:   .long . - strteste

.text

.globl main

main:
    movl $4, %eax          ;# função write
    movl $1, %ebx          ;# descritor stdout
    leal strteste, %ecx    ;# ecx aponta para o texto
    movl comprstr, %edx    ;# número de bytes a passar ao stdout
    int $0x80              ;# executa a chamda ao sist. write
                           ;# por simplicidade ignoramos o valor
                           ;# de retorno

    xorl %eax, %eax
    ret
```

É de referir que a chamada ao sistema `write()` se encontra em termos de funcionalidade abaixo da função `printf()` oferecida pela biblioteca do `C`. Em particular, seria relativamente complicado escrever o valor dum número, de forma inteligível, na saída padrão, a menos que este fosse incluído na própria frase `strteste`. Por exemplo, seria necessário algum esforço de codificação para imprimir o valor do registo `eax`. O problema advém do facto de a repre-

sentação de um inteiro em memória não corresponder à representação desse mesmo inteiro em caracteres ASCII no monitor (sendo esta a representação de interpretação mais simples para um ser humano). Assim, enquanto a função `printf()` permite fazer essa conversão de forma transparente para o programador, a função `write()` escreve os dados em bruto, tal e qual eles se encontram em memória. Para exemplificar, consideremos que se pretende escrever o conteúdo de uma posição de memória com quatro *bytes* que no momento estão todos a 0. Enquanto que a função `write()` tentaria escrever quatro zeros na saída padrão, a função `printf()` escreveria o caracter ASCII '0' (o valor 48). Para isso, a função `printf()` tem internamente de fazer as conversões necessárias, de forma a invocar a função `write()`, passando-lhe uma localização de memória onde se encontra o número 48 e indicando que apenas deve ser escrito um *byte* a partir daquela posição de memória.

O exemplo seguinte utiliza a primeira de todas as chamada ao sistema, chamada `exit()`, para terminar o programa. Note-se que, neste caso o programa não termina com um `ret` final, uma vez que o registo `eip` já não retorna a esse ponto. À semelhança do programa anterior, este programa escreve uma frase na saída padrão utilizando a chamada ao sistema `write()`, invocando depois a chamada `exit()`. O único parâmetro que esta função recebe é o código de erro com que o programa deve terminar, sendo `0` uma indicação de sucesso. Assim sendo, neste caso, o registo `ebx` deve ser posto a zero. Desempenha, portanto, um papel idêntico ao registo `eax` quando é, também, colocado a `0` antes da instrução `ret` final, como, por exemplo, no programa anterior. É interessante referir, que no caso da função `exit()` não é necessário verificar o valor de retorno, pela simples razão de que nunca há retorno.

```
.section .rodata
```

```
despedida:  .ascii "Saída pela syscall exit...\n"
comprfrase: .long . - despedida

.text

.globl main

main:
    movl $4, %eax
    movl $1, %ebx
    leal despedida, %ecx
    movl comprfrase, %edx
    int $0x80

    movl $1, %eax       ;# função exit
    xorl %ebx, %ebx     ;# código de retorno 0 = sucesso
    int $0x80

    #nunca se chega aqui...
```

Finalmente, o último deste exemplos utiliza a função `getpid()`, que tem o número 20 para obter o número do processo que está a executar o programa. Pelas razões apontadas anteriormente, utilizamos a função `printf()` para fazer a impressão do número do processo.

```
.section .rodata
```

```
formato: .string "O identificador do processo (pid) é: %d\n"


.text

.globl main

main:
    movl $20, %eax          ;# função sys_getpid
    int $0x80              ;# execute write system call

    #não tente escrever %eax com a syscall write
    #porque %eax tem a representão *binária* do
    #pid

    pushl %eax
    pushl $formato
    call printf
    popl %eax
    popl %eax

    xorl %eax, %eax
    ret
```

12

Exemplos de Programas

12.1 Programa *Olá Mundo*

```
.section .rodata

strOlaMundo: .string     "Ola Mundo!\n"


.text

.globl main

main:
     pushl %ebp
     pushl $strOlaMundo
     call printf
     addl $4, %esp
     pop %ebp
     xorl %eax, %eax
     ret
```

12.2 Programa *Soma 100 primeiros*

```
.section .rodata

strResult: .string "1 + 2 + 3 ... + 100 = %d\n"


.text


.globl main

main:

     pushl %ebp
     movl $100, %ecx
     xorl %eax, %eax
ciclo:
     addl %ecx, %eax
     loop ciclo          #100 + 99 + ... + 1
     pushl %eax
     pushl $strResult
     call printf
     addl $8, %esp
     popl %ebp
     xorl %eax, %eax
     ret
```

12.3 Programa *Opera dois números*

```
.lcomm num1, 4
.lcomm num2, 4
.lcomm result, 4
.lcomm operacao, 4


.section .rodata

pergunta:    .string "Introduza dois numeros inteiros:\n"
formatoPerg: .string " %d %d"
perguntaOp:  .string "Introduza a operacao (+, -, *, /): "
formatoPergOp: .string " %c"
fraseErro:   .string "A operacao que introduziu nao e valida\n"


.text

#Os parametros encontram-se na pilha por esta ordem:
#num1, op, num2, result
fraseResult: .string "%d %c %d = %d\n"
escreveResult:
     pushl %ebp
     movl %esp, %ebp
     pushl 20(%ebp)
     pushl 16(%ebp)
     pushl 12(%ebp)
```

```
     pushl 8(%ebp)
     pushl $fraseResult
     call printf
     addl $20, %esp
     popl %ebp
     ret

.globl main

main:
     pushl %ebp

     pushl $pergunta
     call printf
     addl $4, %esp
     pushl $num2
     pushl $num1
     pushl $formatoPerg
     call scanf
     addl $12, %esp
     pushl $perguntaOp
     call printf
     addl $4, %esp
     pushl $operacao
     pushl $formatoPergOp
     call scanf
     addl $8, %esp
```

```
    movl operacao, %eax

    cmpl $'+', %eax
    je mais
    cmpl $'-', %eax
    je menos
    cmpl $'*', %eax
    je vezes
    cmpl $'/', %eax
    je dividir
    jmp erro

mais:
    movl num1, %eax
    addl num2, %eax
    pushl %eax
    pushl num2
    pushl $'+'
    pushl num1
    call escreveResult
    addl $16, %esp
    jmp fimcase

menos:
    movl num1, %eax
    subl num2, %eax
    pushl %eax
```

```
    pushl num2
    pushl $'-'
    pushl num1
    call escreveResult
    addl $16, %esp
    jmp fimcase

vezes:
    movl num1, %eax
    imull num2
    pushl %eax          #facil haver transbordo...
    pushl num2
    pushl $'*'
    pushl num1
    call escreveResult
    addl $16, %esp
    jmp fimcase

dividir:
    movl num1, %eax
    cltd
    idivl num2
    pushl %eax
    pushl num2
    pushl $'/'
    pushl num1
    call escreveResult
```

```
     addl $16, %esp
     jmp fimcase

erro:
     pushl $fraseErro
     call printf
     addl $4, %esp

fimcase:
     popl %ebp
     xorl %eax, %eax
     ret
```

References

Card, Rémy, Dumas, Éric, & Mével, Franck. 1998. *the Linux Kernel book*. John Wiley & Sons.

Hahn, Harley. 1992. *Assembler Inside-Out*. Osborne McGraw-Hill.

Hyde, Randall. 2003. *The Art of Assembly Language*. No Starch Press.

IA-32 developer's manual-I. 2001. *The IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation, P.O. Box 7641, Mt. Prospect IL 60056-7641. http://developer.intel.com/design/Pentium4/manuals/245470.htm.

IA-32 developer's manual-II. 2001. *The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Corporation, P.O. Box 7641, Mt. Prospect IL 60056-7641. http://developer.intel.com/design/Pentium4/manuals/245471.htm.

IA-32 developer's manual-III. 2001. *The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, P.O. Box 7641, Mt. Prospect IL 60056-7641. http://developer.intel.com/design/Pentium4/manuals/245472.htm.

Kernighan, Brian W., & Ritchie, Dinnis M. 1988. *The C Programming Language*. Prentice Hall.

Murray, William H., & Pappas, Chris H. 1986. *80386/80286 Assembly Language Programming.* Berkeley, California: Osborne McGraw-Hill.

Silbertschatz, Abraham, & Galvin, Peter Baer. 1998. *Operating System Concepts.* Addison-Wesley Publishing Company.

Stevens, W. Richard. 1990. *Unix Network Programming.* Prentice Hall.