

# ARQUITECTURA DE COMPUTADORES

4<sup>a</sup> Edição Actualizada

José Delgado  
Carlos Ribeiro



FCA - Editora de Informática  
R.D. Estrela, 183-1º Esqº - 1000-154 Lisboa  
Tel: 21 353 27 35 (Departamento Editorial)  
E-mail: fca@fca.pt url: www.fca.pt

<b>APRENDE E DESCOBRE COM O TEU MAGALHÃES</b> Maria José Sousa/ Nuno Cardinhalo	<b>JAVA 6 E PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS</b> F. Mário Martins
<b>ASP.NET 4.0 Curso Completo</b> Luis Abeu	<b>MAYA</b> Depressa & Bem
<b>AUTOCAD 2010</b> Depressa & Bem	<b>MOODLE - CRIAÇÃO E GESTÃO DE CURSOS</b> João Reina
<b>AUTOCAD 2011 &amp; AUTOCAD LT 2011</b> CURSO Completo	<b>ONLINE</b> Álvaro Figueira/ Carmen Figueira/ Hugo Santos
<b>AUTODESK INVENTOR 2010</b> Curso Completo	<b>PHOTOSHOP CS4</b> Curso Completo Fernando Tavares Ferreira
<b>ANTÉRCIO COSTA</b>	<b>PROGRAMAÇÃO COM PHP 5.3</b> Curso Completo Carlos Serrão/ Joaquim Marques
<b>DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO</b> (2 <sup>a</sup> ED. ACT.)	<b>REDES DE COMPUTADORES</b> (7 <sup>a</sup> ED. REV. E ACT.) José Gouveia/ Alberto Magalhães
<b>Filomena C. Lopes/ M. Paula Morais/ Armando J. Carvalho</b>	<b>SEGURANÇA EM REDES INFORMÁTICAS</b> (3 <sup>a</sup> ED. ACT. E AUM.) André Zuquete
<b>DREAMWEAVER CS4</b> Depressa & Bem	<b>SISTEMAS DIGITAIS - PRINCÍPIOS E PRÁTICA</b> Morgado Dias
<b>Hélder Oliveira</b>	<b>SQL SERVER 2008</b> Curso Completo Alberto Magalhães
<b>EXERCÍCIOS DE UML</b> Henrique O'Neill/ Mauro Nunes/ Pedro Ramos	<b>TECNOLOGIAS DE COMPRESSÃO MULTIMÉDIA</b> Nuno Ribeiro/ José Torres
<b>EXERCÍCIOS DE GESTÃO DE PROJETOS RESOLVIDOS COM O MS PROJECT</b>	<b>TECNOLOGIAS DE INFORMAÇÃO - O que são?</b> Sérgio Sousa
<b>Adelaide Carvalho</b>	<b>UTILIZAÇÃO DO EXCEL 2007 PARA ECONOMIA E GESTÃO</b> (2 <sup>a</sup> ED. REV.) Luis Silva Rodrigues
<b>FLASH CS4</b> Depressa & Bem	<b>UTILIZAR O COMPUTADOR</b> Depressa & Bem (7 <sup>a</sup> ED. ACT.) Jorge Neves
<b>Helder Oliveira</b>	<b>VISUAL BASIC 2008 - DESENVOLVIMENTO DE APLICAÇÕES</b> Henrique Loureiro
<b>FOTOGRAFIA DIGITAL - TÉCNICAS COM PHOTOSHOP</b> (2 <sup>a</sup> ED. ACT.)	<b>VISUAL BASIC 2008</b> Curso Completo (2 <sup>a</sup> ED. ACT.) Henrique Loureiro
<b>Magno Urbano</b>	<b>VISUAL BASIC 2010</b> Curso Completo Henrique Loureiro
<b>GESTÃO E LIDERANÇA PARA PROFISSIONAIS DE TI</b>	<b>WINDOWS 7</b> Fundamental Carla Jesus/ Paulo Capela Marques
<b>Pedro Tavares Silva/ Catarina Botelho Torres</b>	
<b>GESTÃO DE PROJECTOS DE SOFTWARE</b> (4 <sup>a</sup> ED. ACT.)	
<b>António Miguel</b>	
<b>GESTÃO DE SISTEMAS E REDES EM LINUX</b> Jorge Granja	
<b>ILLUSTRATOR CS3 &amp; CS4</b> Curso Completo	
<b>Catarina Láis</b>	
<b>INTERNET SEGURA PARA CRIANÇAS - GUIA PARA PAIS E EDUCADORES</b> Paulo Santos/ José Manteladas	
<b>INTRODUÇÃO À PROGRAMAÇÃO EM JAVA</b> António Adrego da Rocha/ Osvaldo Rocha Pacheco	
<b>INTRODUÇÃO À PROGRAMAÇÃO EM VISUAL BASIC 2010</b> António Gameiro Lopes	

DIRJA-SE AO SEU FORNECEDOR HABITUAL OU

CONSULTE-NOS POR EMAIL: [livraria@lidel.pt](mailto:livraria@lidel.pt)

DISTRIBUIÇÃO



IDE - Edições Técnicas, Lda.

SEDE: R. D. Bettaini, 189, N/C Dto., 1049-057 LISBOA  
Internet: 21 354 14 18 – livraria@idel.pt Revenda: 21 351 14 43 – revenda@idel.pt

FormaçãoMarketing: 21 351 14 49 – formacao@idel.pt marketing@idel.pt  
Eis. LínguaEspectro: 21 351 14 42 – depinternacional@idel.pt  
Fax: 21 357 78 27 – 21 352 26 84

LIVRARIA: LISBOA: Av. Praia da Vitória, 14 – 1000-247 LISBOA – Tel.: 21 354 14 18, e-mail: livraria@idel.pt  
PORTO: R. Damão de Góis, 452 – 4050-224 Porto – Tel.: 22 557 35 10, e-mail: depporto@idel.pt

**PARA A VIRGÍNIA, A DIANA E O MIGUEL**  
José Delgado  
**PARA A CÉLIA, A SOFIA E O GUILHERME**  
Carlos Ribeiro

Copyright © Setembro 2010

FCA – Editora de Informática, Lda.

ISBN: 978-72-722-666-5

Capa: Miguel Monteiro  
Ilustração da capa: Miguel Monteiro

Impressão e acabamento: Rolo & Filhos II, S.A.

Depósito Legal N.º 31542/010

**FCA®** – Marca Registrada de FCA – Editora de Informática, Lda.

**FUNDAMENTAL "Depressão & Bem"** – Marcas Registradas de FCA – Editora de Informática, Lda.



Este pictograma merece uma explication. O seu propósito é alertar o leitor para a ameaça que representa para o futuro da escrita, nomeadamente na área da edição técnica e universitária, o desenvolvimento massivo da fotocópia. O Código do Direito de Autor estabelece que é crime punido por lei, a fotocópia sem autorização dos proprietários do copyright. No entanto, esta prática generalizou-se sobretudo no ensino superior, provocando uma queda substancial na compra de livros técnicos. Assim, num país em que a literatura técnica é tão escassa, os autores não sentem motivação para criar obras inéditas e fazê-las publicar, ficando os melhores impossibilitados de ter bibliografia em português. Lembramos portanto, que é expressamente proibida a reprodução, no todo ou em parte, da presente obra sem autorização da editora.

## AGRADECIMENTOS

Nenhuma obra é exclusivamente elaborada pelos seus autores. É mais o resultado do esforço abnegado de muitas pessoas, nem sempre perceptível ou mensurável e muitas vezes esquecido e incompreendido. Não é possível estabelecer estes créditos de forma justa e abrangente. Mas esta obra não poderia ficar completa sem uma referência às pessoas e entidades que, de uma forma mais intensa e incisiva contribuíram, directa ou indirectamente, para que se tornasse numa realidade.

Em primeiro lugar, os nossos agradecimentos a todos os alunos e pessoas interessadas nesta área que, ao longo dos anos, nos inspiraram com todas as suas dívidas e comentários. Eles constituem a principal razão que nos motivou a realizar este esforço e é a eles, e a todos os que ainda pretendem obter formação na área de arquitectura de computadores, que este livro se destina.

O Instituto Superior Técnico e o seu Departamento de Engenharia Informática, em que os autores leccionam a cadeira de Arquitectura de Computadores, proporcionaram as condições materiais que tornaram possível a obtenção da experiência lectiva necessária e o esforço de elaboração de todos os materiais pedagógicos. Em particular, o Programa de Melhoria de Qualidade de Ensino tem financiado o desenvolvimento de algumas das ferramentas usadas, como por exemplo o simulador usado extensivamente neste livro.

O nosso obrigado à FCA e à sua equipa, que pelo seu interesse, confiança e constante apoio nos deram a oportunidade e condições de publicar a nossa experiência pedagógica, visão tecnológica e filosofia de base sobre o que realmente deve ser a área de Arquitectura de Computadores, a um nível introdutório.

Inúmeros colegas, do Instituto Superior Técnico e não só, têm contribuído ao longo dos anos para o enriquecimento da nossa experiência nesta área, através de discussões e trocas de ideias. Sem elas, a nossa visão e perspectivas seriam muito mais limitadas. Na impossibilidade de os nomear a todos, mas sem esquecer a sua importância, gostaríamos de destacar os professores Rui Rocha, Guilherme Arroz, Rui Neves e João Cardoso, com quem temos partilhado a docência da cadeira de Arquitectura de Computadores, no Instituto Superior Técnico (Taguspark). Em particular, o professor Rui Rocha não é um dos autores deste livro apenas por manifesta falta de tempo, mas nem por isso deixa realmente de o ser, por todas as discussões e trocas de ideias que temos manido.

Para além do texto do livro em si, existe um conjunto de ferramentas pedagógicas que têm vindo a ser desenvolvidas ao longo dos anos e que fazem parte integrante de todo o projecto pedagógico em que o livro se insere, contribuindo de forma prática para muitos exemplos e sua comprovação experimental.

O simulador de arquitecturas de computadores é a ferramenta pedagógica de base e a sua implementação tem estado a cargo de um dos co-autores (Carlos Ribeiro). No entanto,

## PREFÁCIO

tem havido diversas contribuições para o seu desenvolvimento e utilização, em particular dos alunos Nuno Afonso, Alexandre Martins, Jorge Valadas, Bruno Lopes, João Trindade e Luís Pedroso. O professor Pedro Reis Santos desenvolveu um compilador de C que gera instruções *assembly* do PEPE, o processador usado como base do livro.

Finalmente, os nossos mais profundos agradecimentos aos co-autores invisíveis que, através de uma infinita compreensão, paciência e até mesmo encorajamento, suportaram estoicamente o roubo de inúmeras horas ao convívio familiar, transformadas à força em infindáveis horas de trabalho. Não sendo possível restituí-las, resta-nos colocar os seus nomes na Dedicatória e afirmar-lhes que o seu sacrifício é indubiativamente uma parte crucial do esforço de concretização desta obra.

JOSÉ DELGADO  
CARLOS RIBEIRO

Os computadores são um dos aspectos mais importantes das nossas vidas, constituindo um dos factores mais importantes de impacte e transformação da sociedade moderna. Cada vez mais pessoas lidam com computadores enquanto instrumento fundamental de trabalho.

Embora a esmagadora maioria das pessoas lide com os computadores apenas na perspectiva de utilizador de aplicações informáticas, o facto é que uma compreensão dos mecanismos básicos do seu funcionamento ajuda a perceber todo o contexto, com as suas limitações e aspectos que condicionam todo o desenvolvimento, funcionalidades, capacidades e desempenho dessas mesmas aplicações.

Não há uma separação radical entre *software* e *hardware*. Ambos têm de cooperar porque é dessa interacção que resulta a funcionalidade e desempenho finais que o utilizador experimenta. O *software* tem vários níveis, desde o aplicacional e de interface com o utilizador até ao mais baixo nível de gestão do computador, incluído no sistema operativo (normalmente Windows, Linux ou outro baseado em Unix). O *hardware* pode também ser visto a vários níveis, desde o nível de integrador, com grandes blocos que se integram numa caixa, passando pelo nível de arquitetura dos processadores e dos restantes blocos do sistema, até ao nível de implementação em circuitos digitais binários (para já não falar na electrónica dos circuitos).

Este livro situa-se essencialmente no nível da arquitetura dos vários blocos de *hardware* do computador, preocupando-se não apenas como eles se devem integrar mas também como é que devem interagir com o *software* e como devem ser implementados internamente.

Os seus destinatários são essencialmente estudantes universitários de nível introdutório, mas foi pensado também para quem quira perceber melhor o funcionamento dos computadores. A aprendizagem destas matérias justifica-se não apenas para quem pretenda conhecer melhor o *hardware* mas também, e sobretudo, para quem quira perceber o impacte, em termos de funcionalidade e desempenho, que o *hardware* tem no *software*.

No Instituto Superior Técnico, este livro é usado como bibliografia de base da cadeira de Arquitetura de Computadores, no 1.º ano, 2.º semestre, das licenciaturas nas áreas de Engenharia Informática, Engenharia de Redes de Comunicações e de Engenharia Electrónica. Portanto, num largo espectro de áreas de actuação.

Para entender o livro não são obrigatorias competências especiais prévias, quer em termos de programação quer em termos de sistemas digitais, embora seja conveniente possuir conhecimentos básicos de uma linguagem de programação de alto nível, em particular C, e alguma experiência de circuitos digitais tome mais fácil a compreensão de algumas

matérias. O livro segue uma via incremental. Em vez de apresentar soluções feitas sem um racional que as suporte, identifica primeiro os problemas e vai desenvolvendo as soluções pouco a pouco, em conjunto com o leitor. A estrutura básica é a seguinte:

UNIDADE	TEMAS
Capítulo 1	Enquadramento do tema e introdução histórica
Capítulo 2	Introdução aos sistemas digitais binários
Capítulo 3	Projeto de um computador básico de 8 bits, praticamente o mais simples possível, mas capaz de implementar aplicações completas como um controlador de semáforos
Capítulo 4	Racional para o projecto de um processador de 16 bits (PEPE) e descrição básica das suas instruções
Capítulo 5	Mecanismos de programação de um computador, indicando como é que um programa numa linguagem de alto nível (C, por exemplo) se consegue mapear nas instruções do processador
Capítulo 6	Técnicas de construção de um computador completo, incluindo periféricos, e avaliação do desempenho de um computador. Inclui ainda uma descrição da arquitetura do PC.
Capítulo 7	Descrição da arquitetura interna de um processador e de forma como os seus blocos constituintes estão organizados. Suporte do hardware para sistemas operativos, incluindo memória virtual e processos, constituindo mais um exemplo da interacção software-hardware que caracteriza a filosofia do livro: o hardware existe para suportar o software
Apêndice A	Resumo das características do processador do livro (PEPE), em formato de manual de referência
Apêndice B	Características de um microcontrolador (CRFEP), um processador com memória e periféricos num só circuito integrado
Apêndice C	Resumo das características fundamentais do simulador usado neste livro (SIMAC)
Apêndice D	Introdução à representação de números reais e computação em vírgula flutuante
Apêndice E	Tabela de codificação de caracteres segundo a norma ASCII, apenas para referência

O objectivo primordial deste livro é contribuir para a aquisição por parte do leitor das seguintes competências fundamentais:

- Identificar os componentes fundamentais de um computador e a função e impacte de cada um deles no desenvolvimento e execução de aplicações informáticas;
- Projectar um processador básico com um conjunto de instruções mínimo, com base em blocos constituintes de funcionalidade bem identificada;
- Implementar os blocos básicos constituintes de um computador usando sistemas digitais binários;
- Programar um computador em baixo nível, gerindo directamente os seus recursos e dominando a interacção com os dispositivos do mundo exterior;
- Estabelecer a forma de conversão das construções mais frequentes de uma linguagem de alto nível em instruções típicas de um processador;
- Avaliar o desempenho de um computador, identificando os estrangulamentos e optimizando-os de forma a maximizar o desempenho global;

- Conceber e implementar a camada de mais baixo nível de suporte de um computador a um sistema operativo.

Um dos problemas cruciais de qualquer livro sobre arquitetura de computadores é saber qual (ou quais) processador(es) se há-de usar para suportar a explicação dos vários conceitos e técnicas, os exemplos, os exercícios, as simulações, etc. Aqui, há essencialmente duas escolas:

- Usar apenas um processador como base para todo o livro, dando exemplos de outros apenas quando relevante, como [Patterson 2008] (com o MIPS) e [Scragn 1992] (com um processador hipotético chamado GEM);
- Usar dois ou mais processadores comerciais (normalmente de grande divulgação) e ir exemplificando para cada um deles as várias ideias que vão sendo expostas ao longo do livro, como por exemplo [Stallings 2006] (com o Pentium e o Power-PC), [Hamacher 2002] (com a arquitetura IA-32, o 680x0 e o ARM) e [Tannenbaum 1999] (com o Pentium, o UltraSPARC e o picoJava).

O grande problema desta segunda escola é o facto de o leitor ter de aprender vários processadores simultaneamente, com as respectivas soluções, compromissos e idiosincrasias, o que tende a enfatizar as soluções específicas em detrimento dos conceitos que constituem o objectivo inicial, para além da confusão resultante. Já é difícil aprender um processador, quanto mais vários, ao mesmo tempo e de forma entretecedal!

A utilização de um só processador permite uma visão mais coerente e uniforme de toda a área de arquitetura de computadores, sem impedir exemplos mais pontuais de implementações específicas noutras processadores quando trouxerem valor acrescentado real. Esta é a solução adoptada neste livro.

Mas qual processador escolher? A grande opção é entre um processador comercial e um projectado com fins pedagógicos. A vantagem óbvia de usar o primeiro é o facto de este já estar totalmente definido e implementado, pelo que tem de estar correcto e a funcionar. A grande desvantagem é ser demasiado complexo e optimizado para funcionalidade e desempenho, uma vez que o seu objectivo é ser um sucesso comercial e não constituir um instrumento pedagógico.

Para fins didácticos, um processador deve suportar a generalidade das técnicas ensinadas no livro, mesmo as mais avançadas, mas ser suficientemente simples para poder ser gerido e programado manualmente (e não numa linguagem de alto nível que esconde todos os detalhes da arquitetura). Esta combinação não existe comercialmente. Os processadores simples são muito limitados, não permitindo exercitar os conceitos e técnicas mais elaborados. Os processadores mais poderosos são demasiado complexos para programação manual e incluem apenas as características que maximizam a sua funcionalidade ou desempenho, não permitindo exemplificar outras características que existem noutras processadores.

O livro de referência nesta área a nível mundial é [Patterson 2008], que usa o MIPS como processador de base e inclui até um simulador que permite correr programas. No entanto, reflecte essencialmente apenas o modelo RISC, explicado na secção 6.6.7, na página 552,

que dificulta a exemplificação de técnicas muito importantes e usadas nouros processadores (como a pilha, explicada na secção 5.7.2.3, na página 321). Além disso é uma arquitectura de 32 bits, ao passo que 16 bits tende a ser o limite para programação manual, essencial do ponto de vista pedagógico (secção 4.3, na página 194).

Assim, este livro introduz um processador projectado totalmente com fins pedagógicos (PEPE – Processador Especial Para Ensino), de 16 bits, suportando a generalidade das técnicas de processadores mais elaborados (sem se limitar às mais adequadas a um dado modelo), com uma arquitectura interna suficientemente simples para ser totalmente explicable e com um conjunto de ferramentas adequadas, nomeadamente um simulador que permite não apenas programar e executar programas mas inclusivamente simular sistemas computacionais completos, com hardware à volta do processador.

A filosofia deste livro assenta no princípio básico de que é preciso fazer para aprender, pelo que compreender um computador implica programá-lo e geri-lo ao seu mais baixo nível (designado linguagem assembly). A componente prática é fundamental em qualquer aprendizagem. Ao longo do livro, a explicação das várias técnicas das arquitecturas de computadores é seguida de pequenos guões laboratoriais, usando o simulador como base para verificação experimental dos programas e dos circuitos previamente descritos.

Os guões laboratoriais em si estão no site de apoio ao livro ([www.fc.ul.pt](http://www.fc.ul.pt)). No entanto, o livro refere os vários guões existentes com a palavra **SIMULADOR** e descreve os principais objectivos de cada uma delas. Estes guões estão orientados ao auto-estudo, permitindo uma aprendizagem auto-suficiente, mas também podem ser usados em aulas de laboratório em cursos com professor ou como complemento das aulas nesta área. Esta componente prática é considerada uma componente fundamental deste livro. O leitor deve executar os vários programas de exemplo aqui descritos (todos contidos no site de apoio ao livro), passo a passo, de acordo com os guões, e perceber realmente o que se está a passar e os vários aspectos que estão envolvidos.

O simulador está implementado em linguagem Java e portanto funciona na generalidade dos ambientes (Windows, Linux, Mac OS, etc.), o que tem a particularidade fundamental de permitir aos alunos desenvolver os seus sistemas e programas em casa, ao seu ritmo, sem necessitar de equipamento específico em laboratório.

Para limitar o número de páginas do livro, os índices de figuras, de tabelas e de programas, bem como o glossário, estão no site de apoio. Incluem-se aqui apenas o índice geral e o índice de simulações, dada a sua importância para o auto-estudo.

Para não limitar a experimentação ao simulador, há também software para instalar numa placa baseada num microcontrolador comercial (Fig. 5.16, na página 398), de modo a fazer experiências e permitir ao PEPE interagir com periféricos reais. Os professores poderão usar ou adaptar este software para montar os seus laboratórios.

O site de apoio inclui ainda outros materiais, como slides para professores e exercícios resolvidos, que complementam todo este ambiente. Mais do que um mero texto didáctico, este livro é o culminar de todo um projecto pedagógico, nas suas diversas componentes.

## ÍNDICE GERAL

### AGRADECIMENTOS IX

### PREFÁCIO XI

### ÍNDICE DAS SIMULAÇÕES XXV

<b>1 - INTRODUÇÃO AO MUNDO DOS COMPUTADORES</b>	<b>1</b>
1.1 O computador como ferramenta.....	2
1.2 A importância dos computadores .....	4
1.3 Processamento da informação.....	5
1.4 Estrutura básica de um computador .....	6
1.5 O mundo com apenas dois símbolos .....	9
1.6 Interacção pessoa-computador .....	11
1.7 A gestão de um computador .....	14
1.8 A evolução dos computadores .....	16
1.9 Perspectivas de evolução futura.....	25
1.10 Conclusões.....	28
<b>2 - O MUNDO BINÁRIO 29</b>	
2.1 Circuitos electrónicos analógicos .....	30
2.2 Circuitos electrónicos digitais.....	31
2.2.1 Funcionamento básico .....	31
2.2.2 Diagramas temporais .....	33
2.2.3 Portas lógicas.....	34
2.3 Álgebra de Boole.....	38
2.4 Funções lógicas .....	39
2.5 Circuitos combinatórios.....	46
2.5.1 Síntese de circuitos combinatórios .....	46
2.5.2 Multiplexers .....	48
2.5.3 Descodificadores .....	51
2.5.4 ROMs .....	54

<b>2.6 Circuitos Sequenciais</b>	57	<b>3.3.2.5 Saltos no programa</b>	140
<b>2.6.1 Elementos bi-estáveis</b>	57	<b>3.3.2.6 Funcionamento detalhado do programa</b>	142
<b>2.6.1.1 Tríaco SR</b>	57	<b>3.3.3 O processador (PEPE-8) e as memórias</b>	145
<b>2.6.1.2 Tríaco D</b>	57	<b>3.3.3.1 Processador (PEPE-8)</b>	145
<b>2.6.1.3 Báscula D</b>	58	<b>3.3.3.2 Memória de dados</b>	146
<b>2.6.2 Registos</b>	59	<b>3.3.3.3 Memória de instruções</b>	147
<b>2.6.3 Portas lógicas de três estados (<i>tristate</i>)</b>	63	<b>3.4 Programação em baixo nível de um computador</b>	149
<b>2.6.4 Banco de registos</b>	65	<b>3.4.1 Instruções em vez de sinais de controlo</b>	149
<b>2.6.5 Contadores</b>	66	<b>3.4.2 Linguagem assembly</b>	152
<b>2.6.6 Registos de deslocamento</b>	68	<b>3.4.3 Implementação das instruções</b>	155
<b>2.6.7 Máquinas de estados</b>	73	<b>3.4.4 Programação em linguagem assembly</b>	160
<b>2.6.7.1 Modelo das máquinas de estados</b>	74	<b>3.4.5 Programação do PEPE-8 em assembly: contagem de bits</b>	163
<b>2.6.7.2 Diagramas de estados</b>	74	<b>3.5 Periféricos</b>	165
<b>2.6.7.3 Máquinas de estados sintetizadas</b>	76	<b>3.5.1 Estrutura do hardware</b>	165
<b>2.6.7.4 Máquinas de estados microprogramadas</b>	83	<b>3.5.2 Programação com periféricos</b>	173
<b>2.7 Representação de números</b>	83	<b>3.5.2.1 Uso de periféricos de saída</b>	174
<b>2.7.1 Números em base 10 (decimais) e 2 (binários)</b>	87	<b>3.5.2.2 Uso de periféricos de entrada</b>	176
<b>2.7.2 Números em base 16 (hexadecimais)</b>	87	<b>3.6 Soluções específicas ou genéricas?</b>	178
<b>2.7.3 Potências de 2</b>	89	<b>3.7 Conclusões</b>	179
<b>2.7.4 Quantos bits para representar um número?</b>	91	<b>3.8 Exercícios</b>	180
<b>2.7.5 Representação de números negativos</b>	93		
<b>2.7.6 Representação de números em complemento para 2</b>	94		
<b>2.7.7 Extenção do número de bits de um número</b>	95		
	97		
<b>2.8 Operações aritméticas</b>	99	<b>4. ARQUITECTURA BÁSICA DE UM PROCESSADOR</b>	183
<b>2.8.1 Soma de dois números binários</b>	99	<b>4.1 Banco de registos</b>	184
<b>2.8.2 Subtração de dois números binários</b>	100	<b>4.2 Endereços de dados e de instruções</b>	190
<b>2.8.3 Excesso</b>	101	<b>4.2.1 Memórias de dados e de instruções separadas; caches</b>	190
<b>2.8.4 Multiplicação de dois números binários</b>	103	<b>4.2.2 Espaço de endereçamento e mapa de endereços</b>	192
<b>2.8.5 Divisão de dois números binários</b>	105	<b>4.3 Impacte da largura das instruções</b>	194
<b>2.9 Conclusões</b>	109	<b>4.4 Endereçamento de <i>byte</i> e de palavra</b>	196
<b>2.10 Exercícios</b>	110	<b>4.5 Codificação das instruções</b>	200
		<b>4.6 Registos</b>	203
		<b>4.7 Bits de estado</b>	205
<b>3. O MEU PRIMEIRO COMPUTADOR</b>	113	<b>4.8 Conjunto de instruções</b>	210
<b>3.1 Componentes básicos de um computador</b>	114	<b>4.9 Instruções de salto</b>	213
<b>3.2 RAM – a memória para guardar informação</b>	115	<b>4.10 Instruções de transferência de dados</b>	216
<b>3.3 O processador (PEPE-8)</b>	119	<b>4.10.1 Combinacões de operandos</b>	216
<b>3.3.1 Unidade de dados</b>	120	<b>4.10.2 Transferências entre registos</b>	218
<b>3.3.1.1 Registo na unidade de dados</b>	120	<b>4.10.3 Transferência de uma constante para um registo</b>	219
<b>3.3.1.2 Unidade aritmética e lógica (ALU)</b>	120	<b>4.10.4 Transferências entre um registo e a memória</b>	223
<b>3.3.1.3 Funcionamento da unidade de dados</b>	123	<b>4.10.4.1 Endereços constantes e em registos</b>	223
<b>3.3.2 Unidade de controlo</b>	128	<b>4.10.4.2 Modos de acesso à memória em 16 bits</b>	224
<b>3.3.2.1 Sinais de controlo</b>	130		
<b>3.3.2.2 Contador de Programa (PC)</b>	130		
<b>3.3.2.3 Um programa simples</b>	131		
<b>3.3.2.4 Constantes no programa</b>	133		
	135		

4.10.4.3	Acesso à memória em 16 bits com índice variável .....	225
4.10.4.4	Acesso à memória em 16 bits sem índice.....	226
4.10.4.5	Acesso à memória em 16 bits com índice constante .....	227
4.10.4.6	Instruções de acesso à memória em 16 bits .....	229
4.10.4.7	Acesso à memória em 8 bits .....	230
4.10.4.8	Acesso à memória em 8 bits e 16 bits .....	235
4.10.5	Transferências para memória de uma constante ou memória .....	237
4.11	Instruções aritméticas .....	237
4.11.1	Instruções aritméticas mais simples .....	238
4.11.1.1	Soma e excesso: série de Fibonacci .....	239
4.11.1.2	Soma e transporte: números grandes .....	240
4.11.2	Multiplicação e divisão .....	242
4.12	Instruções Lógicas .....	246
4.12.1	Funcionalidade das instruções lógicas .....	246
4.12.2	Expressões booleanas .....	249
4.12.3	Instruções de manipulação de um só bit .....	250
4.12.4	Operações lógicas com máscaras .....	254
4.12.4.1	Funcionamento das máscaras .....	254
4.12.4.2	Máscaras AND .....	255
4.12.4.3	Máscaras OR .....	257
4.12.4.4	Máscaras XOR .....	259
4.13	Instruções de deslocamento .....	261
4.13.1	Instruções de deslocamento linear .....	262
4.13.2	Instruções de deslocamento circular (rotações) .....	266
4.14	Modos de endereçamento .....	268
4.15	Conclusões .....	271
4.16	Exercícios .....	272
<b>5 - PROGRAMAÇÃO DE UM COMPUTADOR 275</b>		
5.1	Um problema simples .....	
5.1.1	Modo de actuação de um ser humano .....	276
5.1.2	Modo de actuação de um computador .....	276
5.2	Modelação do problema com fluxogramas .....	276
5.3	Programação em alto nível .....	277
5.4	Mapeamento da programação de alto nível em linguagem assembly .....	280
5.5	Dados, declarações e directivas .....	283
5.5.1	Constantes simbólicas e a directiva EQU .....	285
5.5.2	Variáveis .....	286
5.5.2.1	Tipos das variáveis .....	286
5.5.2.2	Acesso a variáveis de tipos de dados estruturados .....	287
5.5.3	Directivas WORD, TABLE e STRING .....	288
5.5.4	Apontadores .....	291
		294

<b>6 - O COMPUTADOR COMPLETO 409</b>	
6.1 Interligação dos componentes de um computador .....	410
6.1.1 Barramentos.....	410
6.1.2 Operações de leitura e escrita.....	413
6.1.3 Descodificação de endereços (de palavra).....	416
6.1.3.1 Seleção de dispositivo a aceder.....	416
6.1.3.2 Implementação do mapa de endereços.....	417
6.1.3.3 Descodificação parcial dos endereços.....	422
6.1.3.4 Descodificação de mapas de endereços irregulares.....	425
6.1.3.5 Descodificação de endereços programável.....	427
6.1.4 Descodificação de endereços (de byte).....	430
6.1.5 Impacte do endereçamento de byte.....	435
6.1.5.1 Organização da memória em bytes .....	435
6.1.5.2 Endereçamento little-endian e big-endian .....	438
6.1.5.3 Alinhamento dos acessos .....	443
6.1.6 Ciclos de acesso à memória/periféricos.....	446
6.1.6.1 Ligação a barramento de dados.....	446
6.1.6.2 Ciclos de leitura e escrita .....	448
6.1.6.3 Temporizações no acesso aos dispositivos.....	451
6.1.6.4 Acesso a dispositivos lentos.....	456
6.2 Excepções .....	459
6.2.1 Princípios básicos .....	459
6.2.2 Interrupções.....	462
6.2.2.1 Pins de interrupção.....	462
6.2.2.2 Controlo do atendimento de interrupções .....	464
6.2.2.3 Comportamento das interrupções.....	466
6.2.2.4 Mecanismo básico de atendimento de interrupções .....	468
6.2.2.5 Programação com interrupções .....	469
6.2.2.6 Controlador de interrupções.....	476
6.2.3 Outras excepções .....	478
6.2.3.1 Invocação explícita e retorno de uma excepção .....	478
6.2.3.2 Excepções predefinidas .....	480
6.3 Tipos de periféricos .....	482
6.3.1 O que é um periférico? .....	482
6.3.2 Periféricos de memória de massa .....	482
6.3.3 Periféricos gráficos .....	483
6.3.4 Periféricos de comunicação .....	486
6.3.4.1 Princípios básicos .....	487
6.3.4.2 Comunicação paralela .....	487
6.3.4.3 Comunicação série.....	490
6.4 Arquitetura do sistema de periféricos .....	492
6.4.1 Barramentos hierárquicos .....	506
6.4.2 Modos de transferência de dados .....	507
6.4.2.1 Transferência por teste (polling) .....	507
6.4.2.2 Transferência por interrupções .....	509
<b>6.5 Exemplos de computadores completos .....</b>	<b>516</b>
6.5.1 Classes de computadores .....	516
6.5.2 O PC .....	518
6.5.2.1 Arquitetura original .....	518
6.5.2.2 Evolução nos processadores .....	520
6.5.2.3 Evolução nas memórias .....	527
6.5.2.4 Evolução nos periféricos .....	530
6.5.2.5 Evolução nos barramentos .....	531
6.5.3 O microcontrolador .....	534
6.5.3.1 Características básicas .....	534
6.5.3.2 CRePE: um microcontrolador baseado no PEPE .....	538
6.6 Avaliação de desempenho dos computadores .....	541
6.6.1 O que é o desempenho .....	541
6.6.2 Programas de avaliação (benchmarks) .....	543
6.6.3 A lei de Amdahl .....	545
6.6.4 Avaliação do desempenho do processador .....	547
6.6.5 Avaliação do desempenho da memória .....	548
6.6.6 O impacte do compilador .....	551
6.6.7 A filosofia RISC .....	552
6.6.8 Avaliação do desempenho dos periféricos .....	555
6.7 Conclusões .....	560
6.8 Exercícios .....	560
<b>7 - O PROCESSADOR EM DETALHE 565</b>	
7.1 Diagrama de blocos geral .....	566
7.2 Núcleo do processador .....	568
7.2.1 Caminho de dados .....	568
7.2.1.1 Funcionamento geral .....	572
7.2.1.2 Banco de registos .....	572
7.2.1.3 Gerador de constantes .....	574
7.2.1.4 Unidade aritmética e lógica (ALU) .....	575
7.2.2 Unidade de excepções .....	579
7.2.2.1 Unidade de controlo .....	580
7.2.2.2 Microprogramação .....	585
7.2.2.3 Circuito simples microprogramado .....	585
7.2.2.4 Microprogramação no PEPE .....	588
7.3 Processamento em estágios .....	595
7.3.1 Princípios de funcionamento .....	595
7.3.2 Cadeias de estágios .....	599
7.3.3 Implementação das cadeias de estágios .....	603
7.3.3.1 Cadeia de estágios de instruções .....	603
7.3.3.2 Cadeia de estágios de microinstruções .....	608
7.3.3.3 Excepções com processamento em estágios .....	610

**ARQUITECTURA DE COMPUTADORES**

ÍNDICE GERAL

7.3.5	Dependências de dados.....	613
7.3.6	Dependências de controlo.....	618
<b>7.4</b>	<b>Interface de memória .....</b>	<b>621</b>
<b>7.5</b>	<b>Caches.....</b>	<b>622</b>
7.5.1	Princípios de funcionamento das caches .....	622
7.5.2	Organização das caches .....	625
7.5.2.1	Princípios da organização .....	625
7.5.2.2	Mapeamento direto.....	625
7.5.2.3	Mapeamento associativo .....	630
7.5.2.4	Mapeamento associativo por conjuntos .....	633
7.5.3	Políticas de substituição de blocos .....	633
7.5.4	Políticas de escrita nas caches .....	635
7.5.5	Evolução do subsistema de caches .....	636
7.5.6	Casos em que não se quer cache .....	638
7.5.7	Caches no PEPE .....	640
<b>7.6</b>	<b>Memória virtual.....</b>	<b>642</b>
7.6.1	Hierarquia de memórias .....	643
7.6.2	Princípios de funcionamento da memória virtual .....	643
7.6.3	Tradução de endereços virtuais para físicos .....	644
7.6.4	Gestão das páginas .....	647
7.6.5	A TLB e o seu papel na tradução de endereços .....	651
7.6.6	Integração da memória virtual e das caches .....	654
7.6.7	Memória virtual no PEPE .....	660
<b>7.7</b>	<b>Suporte para Processos.....</b>	<b>664</b>
7.7.1	Modelos de programação e de execução .....	664
7.7.2	Multiprogramação .....	664
7.7.3	Interacção entre processos .....	665
7.7.3.1	Sincronização de baixo nível .....	671
7.7.3.2	Sincronização com semáforos .....	671
7.7.3.3	Comunicação .....	675
7.7.4	Programação cooperativa .....	678
7.7.5	Programação.....	679
7.7.6	Proteção .....	682
7.7.7	Gestores de periféricos .....	685
<b>7.8</b>	<b>Conclusões .....</b>	<b>687</b>
<b>7.9</b>	<b>Exercícios .....</b>	<b>689</b>
<b>APÊNDICE A - MANUAL DE PROGRAMADOR DO PEPE 695</b>		
<b>A.1</b>	<b>Pinos do módulo PEPE .....</b>	<b>695</b>
<b>A.2</b>	<b>Registos.....</b>	<b>695</b>
A.2.1	Registos principais .....	696
A.2.2	Registos auxiliares .....	696
A.2.2.1	Configuração do núcleo .....	697
A.2.2.2	Configuração das caches .....	698
A.2.2.3	Configuração da memória virtual .....	699
		700
<b>APÊNDICE B - MANUAL DE PROGRAMADOR DO CREPE 711</b>		
<b>A.3</b>	<b>Excepções .....</b>	<b>701</b>
<b>A.4</b>	<b>Conjunto de instruções .....</b>	<b>701</b>
<b>A.5</b>	<b>Programação do PEPE .....</b>	<b>708</b>
<b>APÊNDICE C - INTRODUÇÃO AO SIMULADOR (SIMAC) 725</b>		
<b>C.1</b>	<b>Desenho de circuitos .....</b>	<b>725</b>
<b>C.2</b>	<b>Simulação de circuitos .....</b>	<b>728</b>
<b>APÊNDICE D - COMPUTAÇÃO EM VÍRGULA FLUTUANTE 735</b>		
<b>D.1</b>	<b>Representação em vírgula flutuante .....</b>	<b>735</b>
<b>D.2</b>	<b>A norma IEEE 754 .....</b>	<b>737</b>
<b>D.3</b>	<b>Operações aritméticas em vírgula flutuante .....</b>	<b>740</b>
<b>APÊNDICE E - CODIFICAÇÃO DE CARACTERES EM ASCII 743</b>		
<b>BIBLIOGRAFIA 745</b>		
<b>ÍNDICE REMISSIVO 747</b>		

APÉNDICE A - MANUAL DE PROGRAMADOR DO PEPF 605

## ÍNDICE DAS SIMULAÇÕES

Simulação 2.1 – Portas 16gicas .....	37
Simulação 2.2 – Circuitos combinatórios .....	47
Simulação 2.3 – Multiplexers .....	50
Simulação 2.4 – Descodificadores .....	54
Simulação 2.5 – PROMs ( <i>Programmable Read Only Memory</i> ) .....	56
Simulação 2.6 – Trinco SR .....	58
Simulação 2.7 – Trincos D .....	59
Simulação 2.8 – Básicas D .....	62
Simulação 2.9 – Registos .....	64
Simulação 2.10 – Lógica de três estados .....	66
Simulação 2.11 – Contadores .....	72
Simulação 2.12 – Registros de deslocamento .....	74
Simulação 2.13 – Máquina de estados simples .....	78
Simulação 2.14 – Máquinas de estados microprogramadas .....	85
Simulação 2.15 – Soma e subtração .....	102
Simulação 3.1 – Utilização de RAMs .....	119
Simulação 3.2 – Unidade de dados .....	129
Simulação 3.3 – Funcionamento do PEPE-8 .....	149
Simulação 3.4 – PEPE-8: programação em assembly .....	162
Simulação 3.5 – PEPE-8: contagem de bits .....	165
Simulação 3.6 – Funcionamento dos periféricos .....	172
Simulação 3.7 – Semáforo simples .....	175
Simulação 3.8 – Semáforo de peões .....	177
Simulação 4.1 – Bits de estado .....	209
Simulação 4.2 – Transferência de dados entre registos .....	219
Simulação 4.3 – Inicialização um registo com uma constante .....	222
Simulação 4.4 – Acesso à memória em 16 bits com índice variável .....	226

## ARQUITECTURA DE COMPUTADORES

Simulação 4.5 – Acesso à memória em 16 bits sem índice .....	227
Simulação 4.6 – Acesso à memória em 16 bits com índice constante .....	229
Simulação 4.7 – Acesso à memória em 8 bits .....	235
Simulação 4.8 – Acesso à memória em 8 e 16 bits .....	237
Simulação 4.9 – Soma e excesso: série de Fibonacci .....	240
Simulação 4.10 – Multiplicação e excesso: factorial .....	243
Simulação 4.11 – Divisão: números primos .....	244
Simulação 4.12 – Divisão: factorização de um número .....	245
Simulação 4.13 – Expressões booleanas .....	250
Simulação 4.14 – Instruções SET, CLR e CPL .....	252
Simulação 4.15 – Instrução BIT .....	254
Simulação 4.16 – Máscara AND .....	256
Simulação 4.17 – Máscara OR .....	259
Simulação 4.18 – Máscara XOR .....	260
Simulação 4.19 – Deslocamentos lineares lógicos .....	265
Simulação 4.20 – Deslocamentos e máscaras .....	265
Simulação 5.1 – Ordenação por bolha .....	265
Simulação 5.2 – Directiva PLACE .....	284
Simulação 5.3 – Ordenação por bolha com apontadores .....	294
Simulação 5.4 – Chamada de rotinas com RL .....	297
Simulação 5.5 – Chamada de rotinas com a pilha .....	321
Simulação 5.6 – Guarda de registos nas rotinas .....	331
Simulação 5.7 – Ordenação por bolha com guarda de registos .....	340
Simulação 5.8 – Contextos de rotinas .....	342
Simulação 5.9 – Recursividade .....	356
Simulação 5.10 – Tabelas de uma só dimensão .....	358
Simulação 5.11 – Tabelas multidimensionais .....	368
Simulação 5.12 – Tabela de apontadores para dados .....	376
Simulação 5.13 – Tabela de apontadores para rotinas .....	378
Simulação 5.14 – Listas ligadas .....	389
Simulação 6.1 – Descodificação de endereços .....	434
Simulação 6.2 – Endereçamento de byte .....	446

## ÍNDICE DAS SIMULAÇÕES

Simulação 6.3 – Ciclos de acesso à memória e periféricos .....	451
Simulação 6.4 – Prolongamento dos acessos .....	459
Simulação 6.5 – Funcionamento básico das interrupções .....	475
Simulação 6.6 – Espera activa .....	509
Simulação 6.7 – Transferência de dados por DMA .....	510
Simulação 6.8 – Processamento em estágios .....	514
Simulação 7.1 – Circuito simples microprogramado .....	587
Simulação 7.2 – Microprogramação .....	594
Simulação 7.3 – Processamento em estágios .....	621
Simulação 7.4 – Caches .....	642
Simulação 7.5 – Memória virtual .....	663
Simulação 7.6 – Multiprogramação .....	671
Simulação 7.7 – Exclusão mútua .....	675
Simulação 7.8 – Programação cooperativa .....	682
Simulação 7.9 – Protecção .....	685
Simulação B.1 – Programação com um microcontrolador (CREPE) .....	724

# I - INTRODUÇÃO AO MUNDO DOS COMPUTADORES

O computador é sem dúvida a maior invenção humana já realizada. Na sua curta história de pouco mais de 50 anos, já revolucionou completamente a nossa forma de existir, nas suas mais variadas vertentes. Tomou completamente conta de nós, e nem por sombras se pense que o impacte nas nossas vidas se limita ao computador pessoal que cada vez mais é um parceiro imprescindível de um crescente número de pessoas, tanto na sua vida profissional como no lazer. Nem sempre os vemos, mas eles estão por toda a parte, desde simples eletrónicos, passando pelos automóveis, até aos grandes sistemas de informação que gerem todos os serviços na sociedade moderna.

Perceber como funcionam é essencial para melhor entendermos o papel que desempenham e o seu verdadeiro impacte nos programas que eles implementam, que cada vez fazem mais coisas e são mais complexos e elaborados. A procura de computadores com mais capacidades, em particular a rapidez de execução dos programas, é uma necessidade constante. A arquitectura (organização interna) de um computador é um factor essencial para o seu desempenho e constitui o objectivo fundamental de atenção neste livro. Neste primeiro capítulo introduzem-se os aspectos fundamentais do mundo dos computadores e estabelecem-se as bases para os capítulos seguintes, que aprofundam as várias vertentes da arquitectura dos computadores.

Mostra-se que um computador não é inteligente, mas limita-se a seguir cegamente as instruções do seu programador. O grande trunfo de um computador é processar informação muito mais rapidamente do que um ser humano e ter uma memória muito grande e precisa, para além de poder trabalhar continuamente sem se cansar.

No entanto, há algum paralelismo entre o modelo de funcionamento de um computador, incluindo os seus blocos constituintes, e o modelo macroscópico e organizacional do comportamento de um ser humano, na sua vida do dia-a-dia. Tantos uns como outros processam e memorizam informação e interagem com o mundo exterior. A grande diferença é que os computadores atacam os problemas de forma sistemática e repetitiva, sem se cansarem, enquanto que os seres humanos preferem o raciocínio, a dedução, a intuição e a criatividade.

Os seres humanos funcionam essencialmente por associações, com suporte em reacções químicas e actividades bioeléctricas, num conjunto harmonioso ainda muito pouco compreendido. Os computadores funcionam em base binária, só com dois símbolos (0 e 1), e apesar desta simplicidade conseguem fazer coisas extraordinárias. E fundamentalmente a arquitectura de um computador que determina a sua funcionalidade e capacidades.

Este capítulo termina com uma breve resenha histórica da evolução dos computadores, incluindo a competição fervilhante do mercado actual nesta área.

## 1.1 O COMPUTADOR COMO FERRAMENTA

Há muitos animais que utilizam ferramentas, incluindo aves e até mesmo peixes, embora apenas com a finalidade de facilitar a sua alimentação (quebrar ovos ou cascas de frutos, por exemplo). Os primatas têm mais facilidade do que os outros animais, não apenas pela sua inteligência adicional mas porque desenvolveram um dedo opponível aos restantes.

Desde há muito que o Homem se habituou a recorrer a ferramentas. Usou paus e pedras, não apenas no seu estrado natural mas transformadas (pedras lascadas para melhor cortar, paus afiados para melhor perfurar). Dominou o fogo e os metais, melhorou o seu conhecimento das tecnologias e dos materiais, inventou os materiais sintéticos, como o plástico, com tecnologia fez mais ferramentas e com ferramentas fez mais tecnologia. O desenvolvimento da humanidade tem sido marcado por inventos, descobertas e criações históricas, como a invenção da roda, a descoberta da electricidade e a invenção do motor, quer o de combustão, quer o eléctrico. Mas numa história de acontecimentos importantes, que se têm sucedido a uma cadência cada vez mais rápida, nenhum teve impacte mais bombástico, tanto em termos de abrangência como de rapidez de mudança face à escala humana, do que o aparecimento do computador, e em particular o do computador pessoal.

O computador é o rei das ferramentas! Não se trata de um equipamento mecânico, como uma alfaia agrícola ou uma simples alavanca. Mas o seu poder é tal que mesmo sem interferir directamente de forma física no nosso mundo (embora o possa fazer indirectamente, controlando equipamento electromecânico) mudou completamente a nossa forma de viver, comunicar e trabalhar, sendo responsável pela obsolescência de inúmeras áreas de trabalho e pela criação de muitas outras. É cada vez mais importante a sociedade saber lidar com computadores, mesmo que numa mera óptica de utilizador.

Mais do que a própria televisão, o computador é "a caixa que mudou o mundo". A televisão pode ser mais mediática, mas o computador bate-a aos pontos no que toca ao impacte nos mais variados aspectos da sociedade. Com o advento da televisão de alta definição e da melhoria das comunicações, a televisão e o computador parecem estar em rota de "colisão", juntando-se a capacidade de distribuição de informação multimédia, de interactividade global e de processamento local de informação.

No entanto, um computador em si não tem nada de mágico. Ao contrários dos seres humanos, não tem inteligência própria, no sentido de poder criar autonomamente novo conhecimento, através de raciocínio com base no que já tem memorizado. Trata-se simplesmente de um sistema electrónico, que consegue processar informação muito mais rapidamente do que um ser humano, de uma forma muito mais fiável e determinística, quer em termos de funcionalidade quer em termos de memória. É óptimo para executar tarefas repetitivas, mas tem de ser programado exaustivamente, prevendo-se todas as situações que podem ocorrer.

A chamada Inteligência Artificial, um conjunto de técnicas que permitem a um programa de computador aprender conhecimento e inferir díaz novas conclusões, consegue dotar o computador de uma inteligência aparente, em particular quando conjugada com a

Robótica, de forma a permitir construir sistemas autónomos que decidam sozinhos perante situações não programadas previamente. No entanto, ainda estamos longe da visão do computador (comum nos filmes de ficção científica) que consegue manter um diálogo e um raciocínio como se fosse uma pessoa.

**NOTA** Existem já pequenos computadores (chamados redes neuronais) que conseguem aprender e tomar decisões de uma forma semelhante à do cérebro humano (simulando os neurónios). São ainda pequenos protótipos experimentais mas já com algumas aplicações interessantes, podendo um dia evoluir para sistemas mais inteligentes e, de alguma forma, substituir os humanos em tarefas que impliquem decisões, tal como as máquinas mecânicas vieram substituir, com vantagens, o músculo humano em massa (de que o maior expoente talvez tenha sido a construção das pirâmides no antigo Egito).

Excluindo estas áreas, o comportamento dos computadores é feito por programação e não por aprendizagem. O computador obedece a um padrão de comportamento predefinido por um programador através de um programa. Quando um computador reage de forma "esperta" perante uma dada situação, seja num jogo, seja num programa de trabalho (na área da contabilidade, por exemplo), o que sobressai é apenas a inteligência de quem fez o programa, que anteviu a situação e programou o computador para reagir de determinada forma. Não é o computador que, tendo apenas por base o conhecimento geral da matéria em causa, analisa a situação, raciocina, toma decisões na altura e aprende com os erros e os sucessos passados, como se fosse um ser humano. Pelo contrário, tal foi feito pelo programador antecipadamente, quando fez o tal jogo ou o tal programa de contabilidade. Tudo tem de estar já previsto.

Um computador tem como blocos constituintes fundamentais um "cérebro" e uma memória. O dito cérebro não é mais de que uma unidade de processamento, capaz de operações muito, muito básicas. A memória não faz mais do que armazenar dados e as instruções dos programas. Um computador em funcionamento pode ser comparado a um ratinho num enorme labirinto, seguindo cegamente setas que indicam a direcção a tomar (as instruções do programa). É o conjunto dessas instruções que o programador desenvolveu e o seu comportamento macroscópico que dão a aparente inteligência ao computador. Esse, em si, não faz a mínima ideia do que está a fazer!

### ESSENCIAL

- O computador é verdadeira "caixa que mudou o mundo", mas não por mérito próprio. O computador executa cegamente as instruções que lhe dão, sem saber o que é que a fazer. A inteligência aparente de alguns programas é apenas à do programador, que soube antever as várias situações possíveis.
- O computador é a ferramenta mais elaborada que o Homem já desenvolveu. Uma das suas características fundamentais é servir para desenvolver outras ferramentas, actuando como meta-ferramenta e aumentando de forma exponencial o ritmo de desenvolvimento da sociedade.

O objectivo deste livro é mostrar como é que esse "cérebro" e essa memória podem estar organizados de forma a conseguir implementar programas arbitrariamente complexos e funcionalidades de alto nível que nos são verdadeiramente úteis na nossa vida quotidiana e sem as quais já não conseguimos conceber a sociedade moderna.

## 1.2 A IMPORTÂNCIA DOS COMPUTADORES

Antes do advento dos computadores tudo era manual, com mão-de-obra intensiva. Os departamentos de contabilidade, as repartições públicas, etc., estavam cheias de pessoas que laboriosamente gastavam metade do tempo a anotar a informação em livros de registo e a outra metade a consultá-los. As pessoas passavam dezenas de anos no mesmo emprego, a executar as mesmas tarefas repetidamente e da mesma forma. O computador veio mudar tudo isto, com características imbatíveis por qualquer ser humano:

- **Automatização** – O computador executa tarefas repetitivas indefinidamente e de forma autónoma sem se cansar, mesmo trabalhando 24 horas por dia;
  - **Capacidade de memória e de processamento** – Permite memorizar encyclopédias inteiras com a maior das facilidades e efectuar cálculos muito complexos;
  - **Rapidez** – Um computador consegue executar operações simples milhares de milhões de vezes mais rápido que um ser humano;
  - **Fiabilidade** – A probabilidade de um computador cometer um erro fortuito é na ordem de biliões de vezes mais baixa do que a de um ser humano (não contando com os erros de programação que os programadores sempre introduzem!...);
  - **Custo** – Um computador pode trabalhar continuamente sem contestar as ordens, sem receber qualquer ordenado, sem pedir aumentos e sem fazer greves. É o empregado ideal. Paga-se a ele próprio em pouco tempo!...
- Pelo seu lado, as pessoas têm custos de ordenado cada vez mais elevados mas (ainda) conseguem apresentar algumas vantagens:
- **Inteligência** – As capacidades de raciocínio de um computador são ainda demasiado primitivas em comparação com as dos seres humanos;
  - **Criatividade** – O expoente máximo da natureza humana e em termos práticos totalmente fora do alcance das capacidades de um computador actual;
  - **Trabalho físico especializado** – Apesar de todos os avanços tecnológicos na área da robótica, em termos de sensores artificiais (visão, tacto, etc.), de actuadores electromecânicos de precisão (mãos artificiais) e de programas de controlo e decisão, ainda nada consegue substituir os trabalhadores humanos nas operações que envolvam intervenção física não repetitiva.
- Estas características têm sido responsáveis pela mudança estrutural das organizações e dos seus métodos de trabalho. Os empregos de tarefas repetitivas e de baixa especialização têm sido sistematicamente substituídos por outros, de maior nível tecnológico e suportados por um número crescente de computadores, que "ocupam" agora os empregos que antes estavam atribuídos a pessoas.

## 1.3 PROCESSAMENTO DA INFORMAÇÃO

Mas afinal o que é que um computador é capaz de fazer, isto é, que tipo de problemas consegue resolver? Resposta: todos os que envolvem processamento de informação.

Deste ponto de vista, um computador não se comporta de forma muito diferente do que uma pessoa a processar a informação. Consegue é fazê-lo de forma mais rápida, mais fiável e com maiores capacidades de processamento e de memória. Na prática, o que um computador faz é simular o comportamento de uma ou mais pessoas. Trata-se de uma simulação simplificada, uma vez que as pessoas necessitam de truques organizacionais e de coordenação que num computador são totalmente supérfluos.

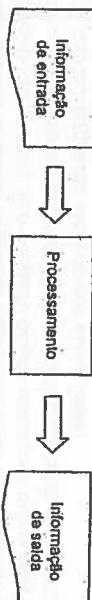


Fig. 1.1 - O computador como sistema de processamento de informação

Um computador consegue facilmente simular (com grandes vantagens) muitas das tarefas que antigamente eram executadas por um batalhão de pessoas num departamento de contabilidade (por exemplo) de uma empresa. O mesmo não se pode dizer de simular o trabalho de uma só pessoa a criar, seja um escritor, um filósofo, um artista plástico, um cientista, etc. Onde é que está a diferença? O primeiro caso corresponde a uma área de trabalho com procedimentos repetitivos e metodologias bem definidas, enquanto que no segundo o que está em causa é a inteligência e a criatividade, o desbravar de novas fronteiras, o estabelecer de novas regras.

Portanto, o processamento de informação pressupõe a existência prévia de um algoritmo (conjunto de passos elementares para se atingir um dado objectivo), que num computador é implementado por um programa e no caso de uma pessoa se traduz por um manual de procedimentos (escrito ou simplesmente explicado oralmente ou ainda assumido de forma implícita).

Aliás, o mesmo processamento pode ser implementado por algoritmos diferentes, alternativos. Por exemplo, o cálculo do perímetro de um rectângulo pode ser implementado de duas formas distintas:

- **Algoritmo 1 – Somar os lados todos (três somas):**
- **Algoritmo 2 – Sumar um lado com outro diferente e multiplicar a soma por 2 (uma soma e uma multiplicação).**

A escolha do melhor algoritmo depende das condições que se tem para os implementar. Se a soma e a multiplicação forem da mesma ordem de grandeza em termos de complexidade e de tempo de execução, então o segundo algoritmo parece ser o melhor. Mas se a multiplicação for muito mais complicada e morosa do que a soma num dado computador, então o primeiro algoritmo tornar-se-á mais simples e rápido. Terá mais passos, mas mais simples.

Em cada caso, tem de se avaliar a situação e escolher o algoritmo mais adequado. Um dado algoritmo pode ser bom para ser implementado num computador e mau para ser executado por uma pessoa e vice-versa, uma vez que pessoas e computadores têm capacidades distintas.

Os componentes de informação fundamentais que estão em causa são os seguintes:

- Programa – Conjunto de instruções (operações básicas) que quando executadas por uma dada ordem implementam o algoritmo;
- Dados de entrada – Conjunto de valores que o programa consome;
- Dados de saída – Conjunto de valores que o programa produz. Em termos intermédios, os dados de saída de uma instrução podem ser os dados de entrada de instruções seguintes;
- Regras de comunicação com o mundo exterior, em termos de:
  - Representação dos dados – Notação que todos entendam;
  - Protocolo de comunicação – Regras conhecidas e aceites por todos.

#### 1.4 ESTRUTURA BÁSICA DE UM COMPUTADOR

Nestas condições, poder-se-á dizer que um computador deverá ser constituído pelos seguintes blocos fundamentais:

- Processador – Executa as instruções;
- Memória de instruções – Onde as instruções que compõem o programa estão armazenadas;
- Memória de dados – Usada para armazenar os dados de entrada, os dados intermédios e os dados de saída do programa;
- Interface com o mundo exterior – Para interacção com este.

**NOTA** Assume-se normalmente que, do ponto de vista da execução do programa, a memória de instruções suporta apenas leitura. A escrita, necessária para memorizar o programa, ocorre – apenas – «às» da execução «tornear». eliminando-se a possibilidade de um programa se autodenunciar. É uma questão de segurança. (contra erros do próprio programador) e de simplicidade.

Este sistema funciona em ciclo, em que o processador é o motor, como se de um coração se tratasse, e executa repetidamente os seguintes passos:

1. Lê uma instrução da memória de instruções;
2. Interpreta a instrução e vê que operação é necessário fazer;
3. Lê da memória de dados a informação necessária para executar essa operação;
4. Executa a operação;
5. Armazena o resultado dessa operação na memória de dados;
6. Volta ao passo 1 onde irá ler a instrução seguinte na memória de instruções.

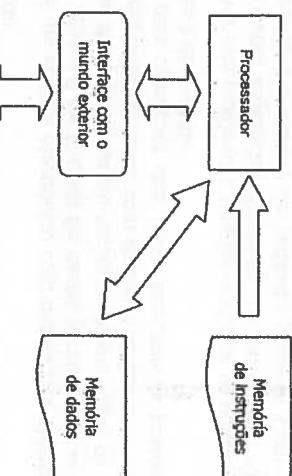


Fig. 1.2 - Estrutura básica de um computador

**NOTA** Algumas instruções usam a interface com o mundo exterior em vez da memória de dados nos passos 3 ou 5 ou em ambos. É esta troca de informação com o exterior que permite ao computador interagir com o mundo que o rodeia.

Se o elemento de processamento for uma pessoa, o processador será naturalmente o cérebro e a interface com o mundo exterior será composta pelos cinco sentidos, complementada pelos dispositivos motores (músculos e membros). O cérebro inclui ainda memória interna, mas podemos admitir para efeitos de sistematização (é até porque a memória humana não é de forma geral particularmente fiável nem de grande capacidade...) que a memória é apenas externa, em que a memória de instruções é constituída por um manual de procedimentos escrito e que a memória de dados consiste num simples bloco de apontamentos.<sup>1</sup>

Um computador é um elemento de processamento incomparavelmente mais simples e determinístico, mas ainda assim o princípio básico é o mesmo. A Fig. 1.3 representa o modelo básico de uma arquitetura de computador.

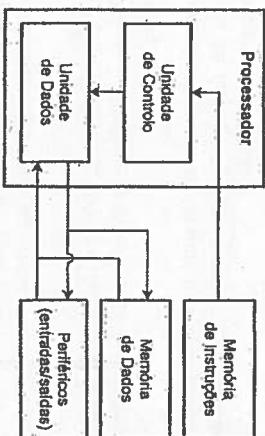


Fig. 1.3 - Arquitetura básica de um computador

<sup>1</sup> Claro que, desde o barro ou a pedra, passando pelo papiro, papel, fitas e discos magnéticos, CDs, DVDs, etc., o suporte físico de registo de informação tem evoluído bastante...

**NOTA**

Quem conheça mais em detalhe o interior de um PC poderá interrogar-se onde é que estão as duas memórias separadas, de instruções e de dados. Não há apenas um conjunto de módulos de memória? É verdade, mas os PCs têm vários níveis de memória. No nível mais perto do processador, há duas memórias separadas, conhecidas por caches. A secção 4.2.1, na página 190, e a secção 7.5, na página 622, apresentam mais detalhes.

Originalmente, a arquitectura clássica é conhecida como o modelo (ou arquitectura) de von Neumann, nome da pessoa a quem é geralmente atribuída a sua autoria (em 1945), e tem apenas uma só memória. A arquitectura da Fig. 1.3 é conhecida como arquitectura de Harvard e constitui uma variante ao modelo original. A sua grande vantagem é permitir ao processador aceder ao mesmo tempo à memória de dados e à memória de instruções. Logo, permite que o computador execute programas mais depressa.

O processador está dividido em duas partes fundamentais:

- **Unidade de Dados** – Executa as operações que o processador necessita de efectuar, quer a nível aritmético (soma, subtração, multiplicação, etc.), quer a nível lógico (operações com FALSO e VERDADEIRO);
- **Unidade de Controlo** – Responsável por ler e interpretar as instruções lidas da memória de instruções, dar ordens à Unidade de Dados para executar operações, coordenar as leituras e escritas na memória de dados, coordenar a comunicação com o mundo exterior através dos periféricos, etc.

**NOTA**

Existem outros modelos de computação, entre os quais se incluem as redes neurais referidas anteriormente. A descrição destes modelos está para além do âmbito deste livro, que se concentra no modelo mais clássico por ser o único usado actualmente em larga escala a nível mundial.

Os periféricos são dispositivos especializados que permitem ao processador:

- Comunicar com o mundo exterior, quer sejam:
- Outros computadores (interface de rede informática);
- Pessoas (interfaces para monitor, teclado, rato, etc.);
- Dispositivos físicos (sensores e actuadores), que permitem controlar sistemas complexos, como por exemplo um edifício.
- Armazenar grandes quantidades de informação (interfaces para discos magnéticos, CDs, DVDs, etc.) que não caberiam na memória de dados do computador. Além disso, as memórias (instruções e dados) de um computador real são normalmente voláteis (ou seja, a sua informação perde-se quando se desliga o computador) e de capacidade bastante limitada. Estes dispositivos (discos magnéticos, CDs, DVDs, etc.) permitem guardar dados e instruções de forma permanente e com capacidades de armazenamento muito superiores. Isto implica que haja transferência de informação entre as memórias e os discos, que estão ligados ao processador como periféricos.

**ESSENCIAL**

- Um computador é constituído basicamente por um processador, memória e periféricos. Quando o processador quer armazenar dados (para posteriormente os ler), usa a memória. Quando quer trocar dados com o mundo exterior, usa os periféricos.
- O processador é o elemento ativo (que faz a computação), mas não consegue programar. Têm de o ir buscar à memória de instruções, instrução a instrução, sequencialmente.
- Também não consegue os dados. Se uma instrução precisar de dados, tem de os ler da memória de dados. Se uma instrução produzir um resultado, tem de o armazenar na memória de dados.
- A vida de um processador resume-se a: lê instrução, lê dados, executa instrução, armazena dados, c. volta ao mesmo na instrução seguinte. E consegue fazer este ciclo vários milhares de milhões de vezes por segundo!!!

## 1.5 O MUNDO COM APENAS DOIS SÍMBOLOS

Os seres humanos são seres vivos que funcionam por processos químicos e bioeléctricos, com valores contínuos. Os efeitos são proporcionais às variações dos valores das concentrações de certos elementos químicos e das pequenas correntes eléctricas, de uma forma gradual e contínua.

Os computadores são sistemas electrónicos, com base em tensões eléctricas mas com valores discretos. Usam apenas dois valores básicos, numa base binária (normalmente, a ausência de tensão eléctrica designa-se 0 e a presença dessa tensão designa-se 1).

Macroscopicamente, as pessoas utilizam a base decimal para as operações aritméticas (com 10 valores diferentes, conhecidos por dígitos, uma vez que têm normalmente 10 dedos nas mãos) e um alfabeto composto por 26 símbolos, conhecidos por letras, para as operações com texto.

Um computador é um sistema electrónico com muitos fios eléctricos, cada um podendo apenas tomar os valores 0 ou 1. Cada fio destes permite representar um *bít* (termo que vem da contracção das palavras anglo-saxónicas *binary digit*, ou dígito binário). Assim, um computador tem de se contentar com apenas 2 símbolos (0 e 1), quer para operações aritméticas quer para operações com texto, enquanto as pessoas têm pelo menos 36 símbolos diferentes à disposição.

A forma de resolver este problema é simples: codificar os números em base decimal e as letras do alfabeto com sequências diferentes de vários *bits*. Quantos mais *bits* uma sequência tiver, maior é o número de variantes possível. Com 4 *bits*, por exemplo, é possível representar 16 sequências diferentes, simplesmente variando as combinações dos

valores dos vários *bits*. A Tabela 1.1 mostra essas sequências e a sua correspondência com valores em base decimal.

VALOR EM BINÁRIO	VALOR EM DECIMAL
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Tabela 1.1 - Representação em binário e em decimal dos números de 0 a 15

De igual forma, é possível codificar letras e sinais de pontuação em diversas sequências de *bits*. Durante muitos anos usou-se a chamada codificação ASCII, em que as 52 letras (26 maiúsculas, 26 minúsculas) mais uma série de sinais de pontuação (vírgula, ponto final, etc.) eram codificadas em 8 *bits*, o que permite 256 combinações diferentes. Há alguns anos foi adoptada uma nova codificação, designada Unicode, que utiliza 16 *bits* e permite 65.536 combinações diferentes, suportando assim melhor os alfabetos com muitos símbolos (es orientais, nomeadamente).

Em computadores, o conjunto de 8 *bits* é muito importante. Por esse motivo, tem um nome próprio: byte. Mas não constitui nada mais do que um conjunto de 8 *bits*. Normalmente, as sequências de *bits* usadas têm um número de *bits* que é múltiplo de 8 (e, por conseguinte, de bytes). A vantagem de usar bytes como unidade mínima de manipulação em vez de *bits* individuais é facilitar a referência e a manipulação dos números binários (porque estes têm normalmente muitos *bits*).

Também na comunicação entre elementos de processamento se notam as diferenças:

- As pessoas usam língua natural, falada ou escrita (recorrendo às letras e dígitos), obedecendo a regras de conversação que vão aprendendo ao longo da sua vida;

- Os computadores ligam-se por fios eléctricos, mais uma vez na base binária, em que para além dos dados são também usados *bits* de controlo que estabelecem protocolos de comunicação extremamente simples e bem definidos, na sua essência equivalentes a semáforos (de todos os computadores envolvidos numa comunicação, só um computador de cada vez recebe luz verde para "falar").

COMPONENTE DE INFORMAÇÃO	PESSOA	COMPUTADOR
Programa	Manual de procedimentos	Memória de instruções
Dados	Bloco de notas	Memória de dados
Elemento de processamento	Cérebro	Processador
Representação de dados	Língua natural, letras, dígitos decimais	Bits, bytes
Regras de comunicação	Regras de conversação	Protocolos com sinal binário

Tabela 1.2 - Comparação entre características das pessoas e dos computadores

A Tabela 1.2 sumariza as características das pessoas e dos computadores em termos dos componentes de informação enunciados na secção 1.3. Esta comparação destina-se apenas a ilustrar a nível "macroscópico" que quer as pessoas quer os computadores enfrentam o mesmo tipo de problemas ao processar informação e ao comunicar. A analogia não pode ser levada demasiado longe. Os processadores dos computadores têm um modelo de funcionamento e capacidades (rapidez de processamento, memória, fiabilidade, etc.) completamente diferentes dos do cérebro. E por essa razão que os computadores não conseguem substituir as pessoas nem vice-versa. Precisamos de ambos!

## 1.6 INTERACÇÃO PESSOA-COMPUTADOR

A Tabela 1.2 sistematiza as características fundamentais das pessoas e dos computadores, do ponto de vista do processamento da informação, mas de forma separada. Naturalmente, logo surge a pergunta seguinte: "Como é que pessoas e computadores comunicam entre si?".

A comunicação "computador → pessoa" é relativamente fácil. Os computadores têm actualmente uma boa interface gráfica e apresentam sem problemas texto, gráficos, diagramas, som (música ou voz), vídeo, etc., de modo a melhor transmitir a mensagem adequada ao seu utilizador.

No entanto, convém não esquecer que toda esta informação tem de ser primeiro introduzida no computador por um programador ou operador e o computador tem de ser programado para apresentar a informação no formato adequado e pela ordem correcta. É aqui que surgem os problemas, na comunicação "pessoa → computador".

Introduzir dados é relativamente fácil (embora potencialmente moroso e trabalhoso). O teclado permite introduzir texto e já há máquinas fotográficas digitais, digitalizadores de imagem (scanners), equipamento para digitalizar vídeo, etc. Mas depois como é que o

computador vai "saber" o que fazer com tudo isto? Que processamento dos dados é preciso fazer? Deve apresentar o quê, como e quando?

Programar computadores correctamente, de forma a cumprir determinadas especificações, não é nada fácil. Os seres humanos são conhecidos por, apesar de toda a sua inteligência, terem dificuldade em comunicar com os seus pares e exprimir correctamente as suas ideias. Muitas vezes dizemos uma coisa quando na realidade querímos dizer algo distinto, e o nosso interlocutor percebe ainda outra coisa diferente.

Por isso, porque é que "dizer" a um computador exactamente o que queremos havia de ser fácil? A este respeito, existe uma frase típica:

*O meu computador é estúpido que nem uma porta! Não faz o que eu quero, só faz o que eu lhe mando fazer!*

Isto quer dizer que cometemos facilmente erros na programação do computador, que não antecipamos todas as situações e que às vezes o programa, ao ser executado no computador, faz coisas que não queríamos que fizesse. Mas a culpa é nossa, pois o computador segue fiel e cegamente as nossas instruções. A frase seguinte é por demais conhecida:

*A máquina tem sempre razão!*

Dado que os computadores são máquinas precisas e determinísticas, não é fácil entender de forma directa a linguagem dos seres humanos, que são "máquinas" difusas e probabilísticas, de pensamento abstracto e comportamento muito aleatório. Por isso, ainda não se consegue falar directamente com os computadores em língua natural de forma suficientemente prática e segura para ser este o meio normal de comunicar com os computadores.<sup>2</sup> Assim, a transformação de uma ideia de uma pessoa num programa de computador que a implemente tem de seguir um caminho longo e árduo, com vários passos, que pode ser descrito pela Fig. 1.4.

Podem distinguir-se os seguintes passos fundamentais, desde o nível das pessoas até ao nível do computador:

1. Uma dada pessoa (chamemos-lhe autor) elabora uma ideia, e a respectiva especificação em língua natural (em português, por exemplo), podendo incluir diagramas, tabelas, imagens, etc. (tudo o que ajude a explicitar a ideia). No entanto, mesmo que toda a documentação seja feita num computador, este não entende o conteúdo semântico. Para o computador, um texto não passa de uma sequência linear de caracteres, uma imagem é apenas uma matriz de pontos (pixels), etc. Nesta figura, a frase "... depositar 100 euros..." exemplifica a língua natural;

<sup>2</sup> Já existem alguns programas que foram desenvolvidos para admitirem texto do utilizador em língua natural, sem formatos pré-definidos. No entanto, são tipicamente limitados à pesquisa de informação pré-memorizada e num contexto de conhecimento normalmente específico.

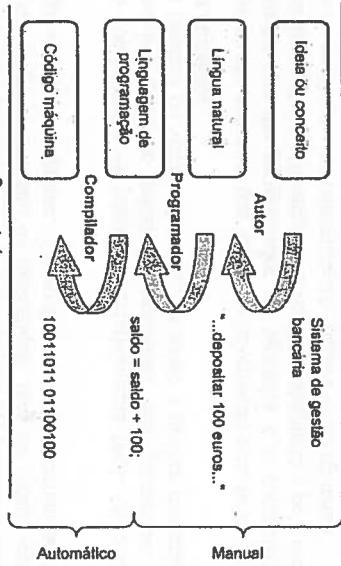


Fig. 1.4 - Processo de transformar um conceito num programa que um computador saiba executar. Parte do percurso é automático, mas a parte mais difícil ainda é manual

2. Outra pessoa (ou a mesma do passo anterior), o programador, transforma a especificação do sistema num programa, constituído por um algoritmo expresso numa linguagem de programação, como C ou Java. A este nível, o programa é apenas texto que o computador ainda não entende directamente, mas (i) já está mais sistematizado e rigoroso do que a especificação original e (ii) pode ser transformado automaticamente na linguagem que o computador entende directamente (código-máquina). O texto do programa é também designado código-fonte. Neste exemplo, a acção de depositar 100 euros é programada como somando 100 ao conteúdo de uma célula de memória, com o nome sugestivo de saldo (de uma hipotética conta bancária), e guardando o resultado da soma nova, mente na célula de memória saldo;
3. O programa do passo anterior é de seguida transformado por outro programa (designado compilador), que faz a conversão para código-máquina, que consiste numa sequência de números binários representando as instruções básicas e que o computador já entende e pode executar directamente. Este já não é um nível que as pessoas entendam, mas não tem importância. O compilador é o responsável por fazer a conversão correctamente entre código-fonte e código-máquina.

**NOTA**

O código-máquina consiste numa sequência de instruções básicas que o computador sabe executar directamente e que reflectem directamente os recursos internos de que o processador dispõe. Há casos em que um programador especializado tem interesse em verificar em detalhe essas instruções. Para que tal seja possível, cada computador tem uma representação dessas instruções em texto, com nomes (e não simples números binários), designada linguagem assembly. Existe uma correspondência de um para um entre cada instrução em linguagem assembly e em código-máquina.

De alguma forma, este é um processo semelhante à digestão dos seres vivos, em que os alimento (que não são assimiláveis directamente pelo organismo) vão sendo sucessivamente mastigados e digeridos, passando por várias transformações até chegar a um estágio em que o organismo já consegue assimilar os nutrientes directamente. O paralelismo vai até um pouco mais longe, pois os compiladores conseguem normalmente detectar e eliminar código-fonte inútil (funções que nunca são chamadas, por exemplo), antes de gerar o código-máquina.

Os primeiros computadores eram programados directamente em código-máquina e em conversão até ao código-máquina. Os programadores não têm que fazer mais do que transformar o enunciado do problema numa descrição equivalente usando uma linguagem de programação de alto nível (C, Java, etc.). "Só" isto já é bastante complicado!

**Nota** A passagem da especificação para o programa é um aspecto bastante complexo, em particular para grandes programas. Normalmente faz-se uma análise mais rigorosa dos requisitos expressos na especificação, estabelece-se um modelo, tomam-se decisões sobre a implementação. As empresas que se dedicam a produzir programas com base em especificações dos clientes têm um conjunto elaborado de regras e pessoal especializado em cada uma das fases de desenvolvimento do programa. A tarefa não termina aqui, pois é preciso produzir as instruções na linguagem de alto nível, testar o programa, corrigir os erros e fazer manutenção após o programa ser entregue ao cliente. É toda uma área gigantesca (Engenharia da Programação) que sai completamente fora do âmbito deste livro.

## 1.7 A GESTÃO DE UM COMPUTADOR

Os primeiros computadores eram programados directamente em binário, usando interruptores e mudando as ligações físicas com cabos. Os resultados eram visualizados num conjunto de lâmpadas (uma lâmpada acesa significava 1, apagada significava 0).

A utilização de um computador era uma tarefa moncha e fastidiosa, reservada para especialistas. O computador estava parado enquanto era programado. Corria o programa e parava de novo. Só podia correr um programa de cada vez. Era um sistema totalmente dedicado a um só programa.

Hoje em dia já não é assim. O computador tem de executar muitos programas, inclusive vários ao mesmo tempo, nomeadamente:

- Programas que o programador especifica;
- Programas que suportam todo o ciclo de desenvolvimento de um programa do utilizador, tais como:
  - Introdução no computador do texto do programa de alto nível (editor);
  - Conversão do texto do programa em código-máquina (compilador);
  - Carregamento na memória de instruções do programa em código-máquina para ser executado.

- Programas que gerem a interface com o utilizador (cerca do monitor, teclado, rato);
- Programas que gerem o dispositivo de memória não volátil usado para guardar os programas (tipicamente, um disco magnético).
- Tudo isto requer uma gestão adequada de forma que os vários programas cooperem de um forma harmoniosa. Assim, um computador típico está sempre a executar um programa de base, conhecido por sistema operativo, que é responsável por:
  - Executar todos os programas necessários;
  - Gerir todos os recursos do computador;
  - Fornecer uma interface ao utilizador que lhe permita dar comandos ao sistema e visualizar os resultados dos programas.

Quando um computador se liga, começa logo a executar o sistema operativo, que depois se encarrega de executar os programas que forem necessários. Quando um computador se desliga, não deve simplesmente desligar-se o botão. Deve fazer-se primeiro o shutdown (parar o sistema operativo), que deixa os vários recursos do sistema de forma arrumada e consistente.

O sistema operativo é provavelmente o programa mais complexo que um computador executa, e tem de estar sempre em execução enquanto o computador está ligado. Se por erro ou outro motivo qualquer o sistema operativo deixar de funcionar bem, todo o computador pára, e diz-se que "o computador foi abaixo". Se tal acontecer, é preciso reiniciar todo o computador (felizmente, com a evolução da tecnologia dos sistemas operativos isto é cada vez menos frequente).

### ESSENCIAL

- Os computadores funcionam em base binária, com apenas dois símbolos: 0 e 1.
- Não entendem língua natural directamente.
- É preciso converter as nossas ideias (expressas em língua natural) para os 0 e 1 (código-máquina).
- Parte da conversão tem de ser feita manualmente, por quem programa o computador, que traduz o algoritmo para instruções numa linguagem de alto nível, como C ou Java, que constituem o programa ou código-fonte.
- O resto já é feito de forma automática, pelo próprio computador, que produz secuências de 0s e 1s executáveis directamente pelo computador (código-máquina).
- Não é só de um computador essa sempre a executar um programa fundamental, o sistema operativo, que é responsável por executar todos os outros programas, gerir os recursos do computador e implementar a interface com o utilizador.

Os sistemas operativos mais conhecidos são:

- **MS-DOS** – Usado pelos primeiros PCs e já só com interesse histórico;
- **Windows** – Usado actualmente nos PCs;
- **Unix** – Pode ser usado em praticamente qualquer computador (PC ou não), e as suas variantes, como por exemplo:
  - **Linux** – Uma versão simplificada do Unix para PCs;
  - **Mac OS X** – Sistema operativo dos computadores Apple.

## 1.8 A EVOLUÇÃO DOS COMPUTADORES

O desenvolvimento da ciência e da sociedade tem acelerado a um ritmo exponencial, e o computador não só é um dos principais responsáveis por essa evolução como também tem ele próprio beneficiado das suas capacidades, suportando todas as ferramentas de desenvolvimento de novos computadores.

Os computadores evoluíram rapidamente, em particular após a 2.ª Guerra Mundial. Paradoxalmente, foi a própria guerra que mais estimulou o seu desenvolvimento inicial, antes de se perceber o seu potencial em aplicações para benefício da sociedade civil.

A tecnologia dos computadores começou pelos grandes sistemas, suportáveis apenas por grandes entidades estatais, mas o aparecimento do PC no início da década de 80 é que realmente mudou a sociedade, porque afectou directamente a vida de cada um de nós.

Como em muitas áreas do conhecimento, a história dos computadores está recheada de desenvolvimentos tecnológicos, do domínio do Homem sobre a matéria. Mas o que é verdadeiramente importante, o que realmente faz a diferença em alturas cruciais é a visão de algumas pessoas que muitas vezes parecem lunáticos a reinar contra a maré, mas que depois se verifica serem os grandes visionários que mudam o rumo da história. Mas, como sempre, ninguém tem razão o tempo todo.

Em 1977, o fundador da DEC (Digital Equipment Corporation, um dos mais importantes fabricantes de computadores), afirmava:

*Não há razão nenhuma que leve alguém a querer um computador em sua casa*

Entretanto, Bill Gates (um dos fundadores da Microsoft) aparecia com a sua visão de um computador em cima da secretaria (e mais tarde em casa) de toda a gente e isso, juntamente com o aparecimento dos microprocessadores num só circuito integrado durante a década de 70, mudou o mundo. Em particular, o PC foi um dos grandes responsáveis pelo declínio de alguns dos fabricantes de grandes computadores, incluindo a DEC.

No entanto, em 1981 Bill Gates afirmava, referindo-se à limitação do DOS (sistema operativo anterior ao Windows) em termos de capacidade de memória:

*640 KBytes [de memória] deverão ser suficientes para qualquer pessoa*

Nessa altura, os programas para computadores eram escassos, rudimentares e pequenos. Uma capacidade de memória de 640.000 bytes (aproximadamente) parecia uma imensidão inegociável. Hoje em dia, os computadores pessoais têm cerca de 10.000 vezes esta capacidade de memória e irá continuar a não chegar. Os servidores têm ainda mais.

Por outro lado, Bill Gates tentou dominar o mercado da Internet, quando este explodiu no início da década de 90, introduzindo a Microsoft Network™ e tentando levar as pessoas a aceder à informação através do seu sistema. Não conseguiu, pois a Internet rapidamente adquiriu um caráiz próprio descentralizado, mas em compensação levou os seus vastos recursos humanos a mudarem todo o conjunto de ferramentas de software produzidas pela Microsoft (o Office™, nomeadamente) para estarem integradas com as tecnologias da Internet e continuou à frente do mercado.

Esta capacidade de se adaptar ao que não consegue vencer tem sido uma das características mais marcantes da personalidade de Bill Gates e uma das grandes razões para o sucesso da Microsoft. É algo de que nem todas as empresas se podem orgulhar, em particular as empresas dos grandes computadores que sofreram com o aparecimento do PC e com a democratização no acesso aos computadores.

Há pouco mais de 20 anos, o cenário mundial da computação era encabeçado pelos grandes dinossauros, os supercomputadores que custavam uma fortuna e se vendiam às poucas dezenas de unidades, sendo usados exclusivamente por entidades estatais ou grandes empresas privadas.

Hoje em dia vendem-se anualmente cerca de 300 milhões de PCs no mundo inteiro, cada um mais poderoso do que um supercomputador de há uns anos atrás e com um custo na ordem dos 500 a 1000 euros, sendo um produto de consumo que se vende em hipermercados, ao lado dos televisores e de outros electrodomésticos.

Os PCs já deixaram o terreno do escritório para invadir os nossos lares e escolas. É já comum haver mais de um PC em cada casa. Um factor de crucial importância é a exposição das crianças desde tenra idade ao computador e à Internet. Com todos os seus benefícios e perigos, é algo que está a mudar muito a sociedade, e de forma muito rápida.

Com alguns aspectos de ironia, muitos dos supercomputadores de hoje em dia não passam de redes de PCs interligados, cooperando na execução de uma mesma aplicação complexa. Sai drasticamente mais barato do que desenvolver um supercomputador à medida<sup>3</sup> e o potencial de poder de computação é igualmente avassalador.

Segundo o TOP 500<sup>4</sup>, 88,8% dos computadores mais potentes do mundo, em Novembro de 2009, eram baseados em processadores da Intel e da AMD, cuja arquitectura evoluiu do

<sup>3</sup> Além de que é uma tarefa morosa e um esforço brutal. Já houve projectos de supercomputadores que foram cancelados porque demoraram tanto tempo que entretanto a tecnologia evoluiu e tornou o próprio projecto obsoleto ainda antes de entrar no mercado.

<sup>4</sup> [www.top500.org](http://www.top500.org)

8086, contra 77,8% em 2005 e apenas 1,2% em 2000! Em Novembro de 2009, o supercomputador mais potente do mundo usava quase 40.000 processadores Opteron da AMD, de 6 núcleos cada, para conseguir uma capacidade de cálculo máxima de 1,76 peta ( $10^{15}$ ) operações por segundo! Já em 2010, foram lançados Opterons de 12 núcleos. Mas para aqui chegar foi necessário o esforço de muitos génios e visionários, para além de muito esforço. O fim da 2.ª Guerra Mundial marcou o início da verdadeira história dos computadores. O período anterior pode ser considerado como pré-história. A Tabela 1.3 apresenta uma fita cronológica com os acontecimentos mais relevantes da história dos computadores e dos processadores. Informação mais detalhada sobre os desenvolvimentos mais antigos pode ser encontrada em [Tannenbaum 1999], [Buchanan 2001], [Patterson 2008], [Wurster 2002] e [IEEE].

ANO	FACTOS MAIS RELEVANTES
Desde 3000 a.C.	<ul style="list-style-type: none"> <li>O abacô foi o primeiro objecto de cálculo e ainda hoje é usado, em particular na Ásia rural;</li> <li>Também foram usadas pedras para fazer contas no antigo Egito e noutras regiões. O termo Cálculo deriva do termo latim <i>cálculus</i> (pedra), e por esta razão estandardizou a designação Cálculo Matemático. Em medicina usa-se o termo cálculo renal para designar uma "pedra" nos rins;</li> <li>Até ao fim da Renascença pouco ou nada se passou em termos de dispositivos para calcular. Os principais desenvolvimentos deram-se ao nível da matemática e da lógica, tendo os árabes, os gregos e depois os europeus, como protagonistas. Instrumentos como o astrolábio e o nôtron eram instrumentos de medição, não de cálculo;</li> <li>Um dos factores fundamentais para o aparecimento das primeiras máquinas de cálculo mecânicos foi o desenvolvimento durante os séculos XVI e XVII da mecânica de precisão usada na relojoaria.</li> </ul>
1622	<ul style="list-style-type: none"> <li>William Oughtred inventou uma régua de cálculo (circular) para calcular logaritmos neperianos.</li> </ul>
1642	<ul style="list-style-type: none"> <li>Blaise Pascal, filho de um cobrador de impostos, desenvolveu o primeiro somador mecânico, a "Pascaline", que já incluía um mecanismo de "e vai um" automático.</li> </ul>
1801	<ul style="list-style-type: none"> <li>Joseph-Marie Jacquard desenvolveu um tear mecânico cujas operações eram comandadas por meio de cartões perfurados.</li> </ul>
1822	<ul style="list-style-type: none"> <li>Charles Babbage construiu a <i>Difference Engine</i>, o primeiro computador mecânico usado para calcular funções matemáticas e produzir tabelas dos valores resultantes.</li> </ul>
1834	<ul style="list-style-type: none"> <li>Charles Babbage começou a desenhar a <i>Analytical Engine</i> que, ao contrário da máquina anterior, podia executar vários algoritmos, usando cartões perfurados para entrada de dados e uma placa de cobre que era gravada com os dados de saída. À frente do seu tempo, esta máquina integralmente mecânica tinha já os componentes fundamentais dos computadores de hoje. O problema é que a tecnologia mecânica não era fiável para uma máquina tão complexa. A parte de cálculo só foi acabada pelo seu filho em 1906.</li> </ul>
1889	<ul style="list-style-type: none"> <li>Herman Hollerith desenvolveu uma máquina electromecânica para auxiliar o trabalho de processamento de dados nos censos da população dos Estados Unidos. Usava cartões perfurados, em que um estribo permitia a passagem de corrente eléctrica quando o cartão apresentava um buraco. Foi usada com sucesso nos censos de 1890 e 1900, a primeira grande aplicação de processamento de dados. Hollerith fundou uma empresa que mais tarde deu origem à IBM.</li> </ul>

ANO	FACTOS MAIS RELEVANTES
1906	<ul style="list-style-type: none"> <li>Lee de Forest inventou a válvula electrónica com controlo de corrente, que permitiu mais tarde implementar os primeiros computadores electrónicos. Uma válvula electrónica tem um electrodo (o cátodo) aquecido ao rubro e que emite electrões para outro electrodo (ânodo). Um terceiro electrodo colocado entre os dois (grélha) permite controlar a corrente de electrões. Com isto é possível construir circuitos electrónicos, mas uma válvula gasta muita potência para o conserto.</li> </ul>
1934	<ul style="list-style-type: none"> <li>Konrad Zuse desenvolveu um computador com relés electromagnéticos (operacional em 1941).</li> </ul>
1943	<ul style="list-style-type: none"> <li>Sob pressão da 2.ª Guerra Mundial, o Colossus foi desenvolvido em Inglaterra para decifrar o sistema de cifra das mensagens alemãs (que usavam um dispositivo chamado ENIGMA). Foi o primeiro computador electrónico, usando válvulas electrónicas.</li> </ul>
1944	<ul style="list-style-type: none"> <li>O Mark I foi desenvolvido por Howard Aiken na universidade de Harvard, usando relés com base na <i>Analytical Engine</i> que Babbage não conseguiu construir com componentes mecânicos. Tinha cerca de 750.000 componentes electromecânicos e era programado estabelecendo ligações eléctricas dentro do próprio computador.</li> </ul>
1945	<ul style="list-style-type: none"> <li>John von Neumann publicou um artigo em que descreve a arquitetura básica de um computador em que o programa estava guardado em memória, como números, eliminando a necessidade de fazer alterações físicas nas ligações do computador;</li> <li>Grace Hopper descobriu o primeiro <i>bug</i> no Mark II, sucessor do Mark I: uma traça (insecto) tinha ficado presa no contacto de um relé, impedindo-o de funcionar!</li> </ul>
1946	<ul style="list-style-type: none"> <li>O ENIAC ficou operacional, embora tarde demais para as aplicações militares que suscitaram o seu desenvolvimento (a 2.ª Guerra Mundial acabou). Tinha 18.000 válvulas electrónicas e pesava 30 toneladas. Era programado com uma multidão de cabos num painel. As válvulas eram pouco fiáveis, e o computador fundava poucas horas até alguma das 18.000 válvulas se avistar.</li> </ul>
1947	<ul style="list-style-type: none"> <li>Foi desenvolvido o primeiro transistor. Tal como a válvula electrónica, permite implementar circuitos de computador, mas sem a energia de aquecimento do catodo e com tensões mais baixas, gastando muito menos energia e de forma muito mais fiável. Por este desenvolvimento, Bardeen, Brattain e Schotley ganharam um prémio Nobel em 1956. Em poucos anos, os transistores tornaram-se válvulas obsoletas.</li> </ul>
1948	<ul style="list-style-type: none"> <li>Apareceu o primeiro disco magnético como memória de massa não volátil.</li> </ul>
1949	<ul style="list-style-type: none"> <li>O EDSAC ficou operacional. Foi o primeiro computador a usar a memória para armazenar o programa, segundo o modelo de von Neumann.</li> </ul>
1954	<ul style="list-style-type: none"> <li>No MIT foi desenvolvido o primeiro computador, apenas com transistores, o TX-0. Entretanto, o estado da arte comercial era estabelecido pelo IBM 650, que usava válvulas.</li> </ul>
1957	<ul style="list-style-type: none"> <li>John Backus da IBM desenvolveu o primeiro compilador de FORTRAN (FORmula TRANslator), uma linguagem de programação de alto nível adequada ao cálculo científico. Até então, os computadores eram programados directamente em código-máquina.</li> </ul>
1958	<ul style="list-style-type: none"> <li>Foi produzido o primeiro circuito integrado, com cinco transistores.</li> </ul>
1959	<ul style="list-style-type: none"> <li>A IBM desenvolveu o primeiro computador comercial com transistores, o IBM 7090; processamento de dados;</li> <li>John McCarthy desenvolveu a linguagem LISP para aplicações de inteligência artificial.</li> </ul>

ANO	FACTOS MAIS RELEVANTES
1960	<ul style="list-style-type: none"> <li>Foi definida a linguagem ALGOL, precursora da maior parte das linguagens de programação, tais como Pascal e C;</li> <li>A DEC desenvolveu o PDP-1, o primeiro computador comercial com um teclado e um monitor (até aqui, a interface com o utilizador resumia-se a lmeios painéis de interruptores e luzes, para além de unidades de leitura e escrita de fitas de papel perfuradas);</li> </ul>
1961	<ul style="list-style-type: none"> <li>A Fairchild Semiconductors produziu o primeiro circuito integrado comercial.</li> </ul>
1963	<ul style="list-style-type: none"> <li>Foi definido o código ASCII (<i>American Standard Code for Information Interchange</i>), que colocou ordem na diversidade de codificação dos caracteres usada pelos vários fabricantes. Isto possibilitou a troca de informação entre computadores.</li> </ul>
1964	<ul style="list-style-type: none"> <li>A primeira versão de BASIC (<i>Beginner's All-purpose Symbolic Instruction Code</i>) foi desenvolvida;</li> <li>O "trato" foi inventado por Doug Engelbart.</li> </ul>
1965	<ul style="list-style-type: none"> <li>Gordon Moore notou que o número de transistores que se conseguiam colocar num só circuito integrado aumentava para o dobro em cada 24 meses, o que ficou conhecido como a Lei de Moore. Surpreendentemente, tam vindo a manter-se aproximadamente válida desde então;</li> <li>A IBM desenvolveu o System 360, o primeiro computador a usar circuitos integrados em vez de transistores individuais. Foi também o primeiro a usar a mesma arquitetura ao longo de uma família de computadores, desde um pequeno até um com maiores capacidades. Foi o primeiro passo no sentido de separar a especificação da arquitetura da sua implementação em hardware, dando uma interface comum aos vários modelos para o software;</li> <li>A DEC produziu o PDP-8, o primeiro minicomputador, muito mais barato que os grandes IBMs e abriu o leque de empresas que podiam ter computador próprio.</li> </ul>
1968	<ul style="list-style-type: none"> <li>É criada a Intel, empresa fabricante de circuitos integrados;</li> <li>Na sequência do desenvolvimento do rato, Douglas Engelbart demonstrou pela primeira vez que os computadores podiam ser controlados por manipulação de objectos no ecrã em vez de linhas de texto.</li> </ul>
1969	<ul style="list-style-type: none"> <li>A Intel produziu o primeiro microprocessador, o 4004, capaz de processar apenas 4 bits de cada vez e incorporando 2300 transistores.</li> </ul>
1971	<ul style="list-style-type: none"> <li>A primeira versão do Unix (um sistema operativo, ou programa de gestão de um computador) foi desenvolvida nos Bell Laboratoty. Uma das suas grandes vantagens e razão do sucesso é a facilidade de adaptação às arquiteturas nas quais se pretende instalar;</li> <li>Niklaus Wirth concebeu a primeira linguagem de programação estruturada, Pascal;</li> <li>A IBM desenvolveu a primeira disquete, de 8 polegadas.</li> </ul>
1972	<ul style="list-style-type: none"> <li>A Intel produziu o primeiro microprocessador de 8 bits, o 8008.</li> </ul>
1974	<ul style="list-style-type: none"> <li>Bob Metcalfe da Xerox demonstrou pela primeira vez a ethernet, a tecnologia que está na base das redes locais de computadores;</li> <li>A Intel desenvolveu o 8080, um microprocessador de 8 bits melhorado face ao 8008 e usado nos primeiros computadores pessoais. Um aspecto interessante é que esses primeiros computadores eram vendidos em kit, e os utilizadores tinham de os montar antes de os utilizar;</li> <li>A linguagem C apareceu, bastante integrada com o sistema operativo Unix, e desde logo foi um sucesso, em particular, no meio universitário. Inicialmente, uma das razões para o seu sucesso foi o seu relativo baixo nível, permitindo com facilidade truques de implementação que por exemplo Pascal, com as suas regras de programação estruturada, simplesmente não permitia.</li> </ul>
1975	<ul style="list-style-type: none"> <li>A Intel produziu o 8085 como resposta ao Z80, um microprocessador da Zilog, com mais funcionalidade, mais rápido e mais barato do que o 8080 da Intel. O 8085 continuou a ter 8 bits;</li> </ul>
1976	<ul style="list-style-type: none"> <li>Bill Gates e Paul Allen fundaram a Microsoft;</li> <li>Seymour Cray desenvolveu o Cray-1, o primeiro supercomputador (no outro extremo da escala dos computadores).</li> </ul>
1978	<ul style="list-style-type: none"> <li>A Intel desenvolveu o primeiro microprocessador de 16 bits, o 8086.</li> </ul>
1979	<ul style="list-style-type: none"> <li>Dan Bricklin desenvolveu a primeira folha de cálculo, o VisiCalc, o precursor do Lotus 123 e do Microsoft Excel. Corria no Apple II e teve um grande papel no sucesso deste computador;</li> <li>A Intel lançou o 8088, um 8086 com uma ligação de dados à memória de apenas 8 bits, para ter um custo menor.</li> </ul>
1981	<ul style="list-style-type: none"> <li>A IBM lançou o primeiro computador pessoal (IBM-PC) orientado ao mercado empresarial (por oposição aos anteriores computadores pessoais, mais dedicados ao entusiasta e aos jogos). Tinha um 8088 como microprocessador e a primeira versão do MS-DOS como sistema operativo. A IBM fez algo muito incharacterístico: lançou uma arquitetura que não era sua, com um sistema operativo que não era seu e ainda por cima publicou as especificações do computador. Este facto permitiu a inúmeras empresas fabricarem imitações (clones) a mais baixo custo e produzirem placas de hardware para estender a funcionalidade do PC. Resultado: um enorme sucesso;</li> <li>David Patterson desenvolveu um pequeno microprocessador a que chamou RISC I (<i>Reduced Instruction Set Computer</i>), com uma arquitetura muito simples, por oposição às arquiteturas RISC I foi desenvolvido em estreita colaboração com o compilador e os excelentes resultados surpreendentes a indústria. Hoje em dia, todos os microprocessadores incorporam as ideias básicas da filosofia RISC (embora não tenham necessariamente um número reduzido de instruções);</li> </ul>
1982	<ul style="list-style-type: none"> <li>A Intel lançou o 80186, que essencialmente era um 8086 com muitos dos periféricos encontrados no PC Incorporados no próprio circuito integrado do microprocessador;</li> <li>A Intel lançou ainda o 80286, um microprocessador de 16 bits mas com importantes melhorias na funcionalidade e nas capacidades. Um factor extremamente importante é que tem sido ponto de honra desde então nos microprocessadores da Intel é a compatibilidade. Qualquer novo processador tem sido compatível com os anteriores (os que não foram redundaram em fracasso);</li> <li>Apareceu o Cray X-MP, que consistia em dois Cray 1 ligados em paralelo;</li> <li>O Osborne 1 foi o primeiro computador transportável (e não propriamente portátil, pois pesava 12 Kg).</li> </ul>
1983	<ul style="list-style-type: none"> <li>A Microsoft desenvolveu uma nova versão do sistema operativo do PC, MS-DOS 2.0;</li> <li>A revista Time elegeu o IBM-PC como o "homem" do ano!</li> </ul>
1984	<ul style="list-style-type: none"> <li>A Apple lança o primeiro Macintosh.</li> </ul>

## ANO

## FACTOS MAIS RELEVANTES

1985	<ul style="list-style-type: none"> <li>A Microsoft lançou o primeiro versão do Microsoft Windows, baseando-se em boa parte as ideias da interface de utilizador do Macintosh (que por sua vez se tinha baseado nas ideias do Alto, um computador pessoal muito inovador desenvolvido pela Xerox);</li> <li>A Intel desenvolveu o 80836, o primeiro microprocessador de 32 bits;</li> <li>Os CD-ROMs apareceram também neste ano;</li> <li>O Cray 2 foi desenvolvido, com uma capacidade de cálculo de cerca de 1000 milhões (<math>10^9</math>) de operações por segundo. Também a Connection Machine, composta por milhares de processadores elementares, apareceu neste ano para explorar o paralelismo massivo nos programas.</li> </ul>
1986	<ul style="list-style-type: none"> <li>A Microsoft lançou o MS-DOS 3.0;</li> <li>Foi desenvolvido o primeiro microprocessador RISC comercial, o MIPS R2000.</li> </ul>
1987	<ul style="list-style-type: none"> <li>A IBM e a Microsoft lançaram a primeira versão do OS/2, um sistema operativo alternativo, que acabou por não "pegar";</li> <li>A Sun Microsystems desenvolveu a arquitetura SPARC (Scalable Processor ARChitecture), com base nas ideias do RISC II, sucessor do RISC I. No entanto, não o fabricou. Licenciou-a a várias fabricantes de circuitos integrados. Esta estratégia contrastou com a postura de outras empresas, como a DEC, que acabou por não conseguir competir com as sucessivas evoluções do PC e acabou por ser comprada pela Compaq em 1998.</li> </ul>
1988	<ul style="list-style-type: none"> <li>A Microsoft lançou o MS-DOS 4.0;</li> <li>O primeiro vírus apareceu na Internet.</li> </ul>
1989	<ul style="list-style-type: none"> <li>Tim Berners-Lee concebeu a tecnologia de base da WWW (<i>World Wide Web</i>). O objectivo inicial era apenas desenvolver um sistema para visualização de gráficos para os cientistas do CERN, mas acabou-se tornar numa das tecnologias com maior impacte, ao permitir às pessoas e empresas disponibilizar informação na Internet de forma simples e universal;</li> <li>A Intel lançou o 80486, que incorporou pela primeira vez no próprio microprocessador a unidade de computação em vírgula flutuante.</li> </ul>
1990	<ul style="list-style-type: none"> <li>A Microsoft lançou o Windows 3.0;</li> <li>A Microsoft lançou o MS-DOS 5.0;</li> </ul>
1991	<ul style="list-style-type: none"> <li>A Thinking Machines introduziu o CM-5, um supercomputador massivamente paralelo com uma capacidade máxima de cerca de 64.000 milhões de operações por segundo;</li> <li>Apareceu o MIPS R4000, o primeiro microprocessador comercial de 64 bits;</li> </ul>
1992	<ul style="list-style-type: none"> <li>Apareceu o DEC Alpha, a segunda arquitetura de 64 bits, dirigido ao mercado profissional de estações de trabalho e servidores. Com um excelente desempenho (muito superior aos microprocessadores da Intel de 32 bits e muito mais baixa frequência de trabalho), nunca teve software adequado e foi o último grande desenvolvimento da DEC.</li> </ul>
1993	<ul style="list-style-type: none"> <li>A Intel lançou o primeiro Pentium, sucessor do 80486 mas ainda de 32 bits;</li> <li>A IBM, Motorola e Apple lançaram o primeiro microprocessador da família do PowerPC, o 601;</li> <li>A Microsoft lançou o Windows NT, o MS-DOS 6.0 e um conjunto de ferramentas (Office 4.0);</li> </ul>
1994	<ul style="list-style-type: none"> <li>A Intel descobriu uma falha no algoritmo de divisão do Pentium, mas com baixa probabilidade de erro. Em vez de corrigir, o que teria custos e atrasaria a produção, decidiu avançar assim mesmo, corrigindo o erro na próxima versão, enquanto produzia cerca de 5 milhões de Pentiums com o erro. Pouco depois, um matemático descobriu o erro e publicou-o na Internet. A Intel tentou desvalorizar o problema, dizendo que a probabilidade de acontecer era muito pequena e que afectava apenas números muito precisos; mas a polémica que se seguiu a nível mundial forçou a Intel a recolher e substituir todos os Pentiums defeituosos, com um custo estimado em 500 milhões de dólares, talvez cerca de 1000 vezes o custo de reparar o erro antes de produzir os Pentiums. Para já não falar nos custos de imagem da Intel...;</li> </ul>

## ANO

## FACTOS MAIS RELEVANTES

1995	<ul style="list-style-type: none"> <li>A Sun Microsystems desenvolveu a linguagem de programação Java, a primeira a ter em conta explicitamente a natureza distribuída da Internet. Pela primeira vez, é assumido que um programa não tem de estar todo carregado num computador para este o executar. Isto levou mais tarde ao princípio "A rede é o computador" e ao desenvolvimento de terminais gráficos sem processamento local ligados em rede a um servidor;</li> <li>A Sun Microsystems desenvolveu a arquitetura UltraSPARC, de 64 bits;</li> <li>A Microsoft lançou o Windows 95, que melhorou bastante a interface com o utilizador;</li> <li>A Intel melhorou o Pentium, lançando o Pentium Pro.</li> </ul>
1996	<ul style="list-style-type: none"> <li>A Intel acrescentou um conjunto de instruções (MMX) ao Pentium para melhor suportar as operações multimédia;</li> <li>A Microsoft estendeu a funcionalidade e a integração do Office com o Office 97;</li> <li>Foi descoberto novo erro no Pentium. Desta vez mostraram que tinham aprendido a ligá-lo. Recontracaram o erro e arranjaram uma solução de correção em Software.</li> </ul>
1997	<ul style="list-style-type: none"> <li>A Microsoft lançou o Windows 98 e enfrentou problemas legais, em especial devido à incorporação do Internet Explorer no próprio sistema operativo.</li> </ul>
1998	<ul style="list-style-type: none"> <li>O reino do Unix foi estendido aos PCs através de uma versão simplificada, o Linux, que tem ganho sucessivamente importância e é na prática a única alternativa ao Windows nos PCs;</li> <li>A Intel lançou o Pentium III.</li> </ul>
1999	<ul style="list-style-type: none"> <li>A Intel lançou o Pentium IV.</li> </ul>
2000	<ul style="list-style-type: none"> <li>A Intel lançou o seu primeiro processador de 64 bits (Itanium);</li> <li>A IBM lançou o primeiro processador comercial de dois núcleos (Power4).</li> </ul>
2001	<ul style="list-style-type: none"> <li>A Intel lançou um Pentium 4 com capacidade de correr duas tarefas simultaneamente (hyperfluxo, ou hyperthreading);</li> <li>A Intel lançou o Itanium 2, com 220 milhões de transistores.</li> </ul>
2002	<ul style="list-style-type: none"> <li>A AMD lançou os seus primeiros processadores de 64 bits (Opteron e Athlon 64).</li> </ul>
2003	<ul style="list-style-type: none"> <li>A Intel lançou o Itanium 2 Madison, com quase 600 milhões de transistores num só circuito integrado.</li> </ul>
2004	<ul style="list-style-type: none"> <li>A Intel lançou o Itanium 2 Madison, com quase 600 milhões de transistores num só circuito integrado.</li> </ul>
2005	<ul style="list-style-type: none"> <li>A Intel lançou os seus processadores de dois núcleos (o Xeon e os Pentium Extreme e D);</li> <li>A AMD lançou os seus processadores com dois núcleos (Dual-core Opteron e Athlon 64 X2);</li> <li>A Apple anunciou que ia deixar de usar PowerPCs e passar a usar processadores da Intel;</li> <li>O número de computadores portátiles vendidos foi superior ao dos computadores de secretária, nos EUA.</li> </ul>
2006	<ul style="list-style-type: none"> <li>A Intel lançou as arquiteturas Core (32 bits) e Core 2 (64 bits), com mais ênfase no consumo do que na frequência do relógio, e fez chegar os dois núcleos aos PCs portátiles (Core 2 Duo);</li> <li>A Intel lançou o Itanium Montecito, um processador com dois núcleos Itanium com 12 MB/s de cache interna cada, num circuito com cerca de 1720 milhões de transistores;</li> <li>A SUN lançou o Niagara, um processador UltraSPARC II com oito núcleos;</li> <li>A AMD anunciou o primeiro processador (Barcelona) com quatro núcleos nativos Opteron;</li> <li>A Intel lançou um Core 2 Quad (com quatro núcleos), embora sejam na prática dois processadores de dois núcleos cada no mesmo circuito integrado, ligados por um barramento.</li> </ul>

ANO	FACTOS MAIS RELEVANTES
2007	<ul style="list-style-type: none"> <li>A Microsoft lançou oficialmente em Janeiro o seu novo sistema operativo, designado Vista;</li> <li>Em Maio, a IBM lançou o Power5, sucessor do Power4, com 4,7 GHz de frequência de relógio, 790 milhões de transistores e tecnologia de 65 nm;</li> <li>Em Agosto, a SUN lançou o UltraSPARC T2 (também conhecido por Niagara II), com alto vias de hiperfluxo em cada um dos oito processadores, ou 64 processadores lógicos, com tecnologia de 65 nm e uma frequência de 1,4 GHz;</li> <li>Em Setembro, a AMD lançou o Opteron Barcelona, o primeiro processador (dos compatíveis com os PCs) com quatro núcleos nativos, 463 milhões de transistores, tecnologia de 65 nm e embora ainda com dois processadores de dois núcleos no mesmo circuito.</li> </ul>
2008	<ul style="list-style-type: none"> <li>A AMD lançou a sua linha de processadores de quatro núcleos para os computadores pessoais, o Phenom X4;</li> <li>A Intel lançou um novo processador de baixa gama, o Atom, grande responsável pelo sucesso de uma nova classe de computadores de baixo custo, os netbooks;</li> <li>A Intel desenvolveu uma nova arquitetura, Nehalem, como sucessora da arquitetura Core;</li> <li>O número de computadores portáteis vendidos foi superior ao dos computadores de secretaria, a nível mundial.</li> </ul>
2009	<ul style="list-style-type: none"> <li>A AMD lançou o Phenom II, em tecnologia de 45 nm e com suporte para memória DDR3 e com 6 núcleos;</li> <li>Em Outubro, a Microsoft lançou o Windows 7;</li> <li>Em Novembro, AMD e Intel chegaram a um acordo histórico, que resolveu as suas inúmeras disputas legais até então;</li> <li>Os netbooks tiveram um aumento de vendas de 80% face a 2008, chegando a 10% das vendas dos computadores pessoais.</li> </ul>
2010 (até Abril)	<ul style="list-style-type: none"> <li>Em Janeiro, a Intel chegou à tecnologia de 32 nm, com o processador Clarkdale, em cuja embalagem integrava ainda um processador gráfico;</li> <li>Em Fevereiro, a Intel apresentou a nova versão do Itanium, o Tukwila, com 4 núcleos e mais de 2000 milhões de transístores, enquanto a IBM apresentou o Power7, com 8 núcleos, e a SUN o UltraSPARC T3, com 16 núcleos;</li> <li>Em Março, a Intel lançou o Gulftown, em tecnologia 32 nm e 6 núcleos;</li> <li>Em Abril, a AMD lançou o Phenom II X6, com 6 núcleos, e o Opteron 6100, com 12 núcleos e 1800 milhões de transístores, enquanto a Intel lançou o Nehalem-EX, com 8 núcleos e 2300 milhões de transístores.</li> </ul>

Tabela 1.3 - Fita cronológica dos factos mais relevantes na história dos computadores e dos processadores até Abril de 2010

Uma das previsões mais espantosas da vida dos computadores ocorreu em 1965, quando Gordon Moore (um dos fundadores da Intel) observou a evolução dos circuitos integrados até então e previu uma duplicação em cada 24 meses do número de transístores que era possível colocar (com o menor custo) em cada circuito integrado, pelos próximos anos. Apesar de outras previsões, muito posteriores, indicarem que os limites da electrónica fariam reduzir esse ritmo de evolução, o facto é que esta previsão se manteve há quase 50 anos e ainda não há limitações concretas no horizonte.

A Fig. 1.5 demonstra este comportamento, desde há muito conhecido pela Lei de Moore, no campo dos microprocessadores, neste caso os da Intel (que é a única empresa que os fabrica desde o início). O processador com maior número de transístores produzido até

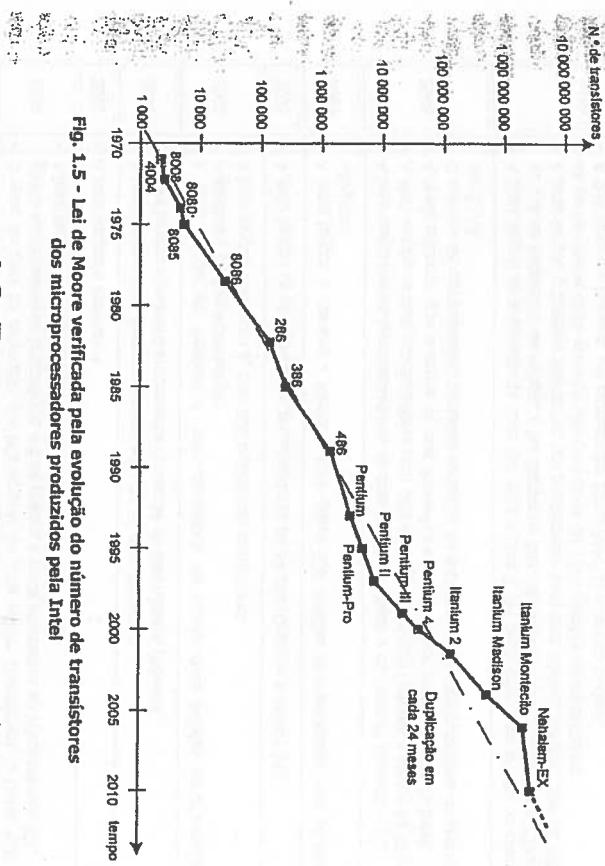


Fig. 1.5 - Lei de Moore verificada pela evolução do número de transístores dos microprocessadores produzidos pela Intel

## 1.9 PERSPECTIVAS DE EVOLUÇÃO FUTURA

Até o leitor menos atento poderá notar na Tabela 1.3 que a história dos últimos anos tem sido dominada pela evolução do PC, quer a nível do seu sistema operativo (históricamente desenvolvido pela Intel), quer a nível do seu microprocessador (historicamente produzidos de forma dominante pela Microsoft). Esta plataforma (designada muitas vezes por Wintel, de Windows e Intel) tem dominado o mercado porque ganhou a guerra no mercado de massas (o dos computadores pessoais) e o poder de cálculo tem evoluído de tal forma que tem invadido o território dos computadores mais profissionais, com um custo reduzido face a processadores específicos, dado o enorme volume de produção. A AMD é a única concorrente da Intel e tem conseguido ganhar-lhe algum terreno no desenvolvimento de processadores, em particular na área dos servidores. A secção 6.5.2.2, na página 520, contém uma descrição mais detalhada dos desenvolvimentos neste campo nos últimos anos.

No lado do software, a Microsoft não tem rival à vista, apesar de a tecnologia Java da Sun Microsystems (adquirida em 2010 pela Oracle) ter ganho grande aceitação no mercado e o software open-source, baseado no mundo Unix, ter obtido grande divulgação. Mesmo

sem concorrência real nos seus mercados mais estabelecidos, já tem sido levada a fazer algumas evoluções na estratégia devido a empresas que conseguem um grande impacto em certas áreas de mercado, como a Google na área da pesquisa de informação na Web.

Ao longo da história tem havido inúmeros fabricantes a tentar a sua sorte nos mais variados domínios, desde os processadores, passando pelos sistemas operativos até às aplicações. No passado, cada fabricante de computador fazia tudo, desde o hardware até ao sistema operativo e ao software aplicacional. Como em muitas outras áreas, poucos são os que conseguem sobreviver nestas condições. Além disso, os grandes inovadores acabam muitas vezes por perder face às tecnologias mais evolucionárias, pois o mercado é bastante conservador e não gosta de grandes mudanças que envolvem riscos e grande esforço. Mas se não se inovar não se consegue entrar no mercado, de modo que o truque é ter a ideia certa, na altura adequada, com o esforço necessário e com a flexibilidade suficiente para acompanhar a evolução do mercado. O problema é que não é nada fácil conseguir conjugar todos estes factores.

Ao nível dos sistemas operativos, sobraram na prática dois: Unix (e as suas variantes, em particular o Linux) e o Windows. Nenhum parece poder eliminar o outro no futuro próximo. Tem características e mercados diferentes. A Microsoft continua forte, com a sua panóplia de produtos, incluindo a plataforma .Net, providenciando a habitual filosofia de integração de aplicações, e embora tenha sido forçada a arranjar uma alternativa à linguagem Java (desenvolvendo a linguagem C#) devido a batalhas legais, não parece ameaçada no imediato pela perspectiva, cunhada pela Sun Microsystems, de que o computador é a rede. As aplicações exigem cada vez mais informação e parece não haver rede que consiga oferecer a capacidade e segurança necessárias. O processamento distribuído terá de continuar a coexistir com o processamento local.

No entanto, as tecnologias de rede também estão a evoluir rapidamente e sopram novos ventos no reino dos sistemas distribuídos, numa nova organização designada por computação em nuvem (*cloud computing*), em que se usam recursos computacionais na Internet, sem se saber exactamente onde estão, em vez de servidores locais. Este nome deriva da história já clássico de representar a Internet e os seus recursos como uma nuvem, cuja composição e localização não são conhecidas em detalhe. Com a sociedade cada vez mais global e interligada, quer a nível das redes sociais, quer a nível das redes empresariais, o cenário global da informática poderá realmente mudar o seu centro de gravidade para a nuvem (Internet) e uma nova ordem, com uma maior relevância de novos intervenientes (Google, nomeadamente), poderá surgir.

Em termos de arquitectura de computadores, a diversidade é maior mas também se está a reduzir. A tecnologia actual permite desacopiar o software do hardware de forma razoavelmente simples, pelo que só as arquitecturas com maiores capacidades e desempenho sobreviverão. O PC é naturalmente dominante no mercado dos computadores pessoais. Apesar de a Apple ter tido algum ressurgimento nos últimos anos, continua com uma fatia de mercado pequena. Por enquanto, continua a haver mercado para outras arquitecturas, nomeadamente nos níveis acima dos PCs (estações de trabalho e servidores)

res) e abaixo (sistemas enbebidos, em que a necessidade de baixo custo e consumo tem permitido a outras arquitecturas, como o ARM, por exemplo, marcar pontos).

Em particular, é de destacar a evolução não apenas dos computadores pessoais como também da própria atitude das pessoas e das empresas face à sua utilização. O mundo está cada vez mais sem fios, com computação e comunicação em qualquer lugar. Pela primeira vez, em 2005 o número de computadores portáteis (*laptops*) vendidos nos E.U.A. ultrapassou o de computadores de secretaria (*desktops*) de forma consistente. A nível mundial, tal ocorreu em 2008. Há lojas que já não vendem computadores de secretaria, devido aos baixos ganhos, perda de quota de mercado e aumento das capacidades e da diversidade de oferta no reino dos computadores portáteis. As pessoas tendem a levar o seu portátil consigo, usando-o como computador de secretaria transportável.

Por outro lado, há uma forte explosão no mundo dos PDAs (*Personal Digital Assistants*), pequenos computadores de bolso que de simples agendas electrónicas já evoluíram para verdadeiros canivetes suíços computacionais, com telefone, redes locais sem fios, câmara fotográfica e de vídeo, localização geográfica por GPS e rádio, receptor de TV digital, leitor de RFIDs, autenticação por impressão digital, etc. É o mundo na mão!

Mesmo os verdadeiros computadores pessoais tiveram uma revolução significativa a partir de 2008, com uma nova classe de computadores, os *netbooks*. Com uma funcionalidade igual a dos restantes computadores pessoais (*laptops* e *desktops*), suportando as últimas versões dos sistemas operativos (nomeadamente, Windows 7), possuem um desempenho sensivelmente inferior e um ecrã mais pequeno, de que resulta um custo significativamente menor (na ordem de 200 a 400 euros, em 2010). Tendo capacidades mais que suficientes para pesquisa na Internet (note-se o prefixo "net" na designação *netbook*) e edição de documentos, a que alia uma grande autonomia sem ligação à tomada (10 horas ou mais), o seu sucesso torna-se inevitável. De 2008 para 2009 o aumento das vendas foi na ordem de 80%, para cerca de 30 milhões de unidades, o que representa aproximadamente 10% do total de vendas de computadores pessoais. Esta percentagem deverá subir sensivelmente nos próximos anos, levando os computadores pessoais para um nível de penetração na sociedade que tenderá a aproximar-se da que existe nos telemóveis. Em Portugal, é de realçar o impacte da iniciativa e-Escolinha, que o com o computador Magalhães (um *netbook*) contribuiu de forma muito significativa para a divulgação destes computadores.

Ao nível dos grandes computadores, a tendência é usar cada vez mais redes de PCs (ou servidores baseados na arquitectura do PC) em vez de grandes sistemas específicos, cujo desenvolvimento sai demasiado caro para ser rentabilizado. Um facto significativo foi o anúncio em 2005 por parte da Apple de que ia trocar os processadores PowerPC, que tinha vindo a usar até então nos seus Macs, por processadores da Intel nos seus novos modelos. Mesmo os fabricantes das arquitecturas não-PC (nomeadamente, IBM e SUN – esta adquirida em 2010 pela Oracle) têm a sua oferta de servidores de alto desempenho também baseada nos processadores usados nos PCs, da Intel ou da AMD. É a concentração cada vez maior em torno dos processadores destes dois fabricantes.

Ninguém sabe exactamente como a tecnologia irá evoluir e onde estará daqui a 10 anos, por exemplo. No entanto, ninguém parece ter dúvidas de que a capacidade das redes de comunicação e o poder computacional dos computadores vão ainda evoluir muito, a ponto de poder tornar realidade aplicações ainda hoje nem sonhadas. O poder de comunicação e de computação irá com certeza chegar de forma intensa a dispositivos de acesso à informação sem fios e literalmente colocados na nossa mão. A nossa vida continuará a mudar muito, cada vez mais rapidamente, em boa parte devido à existência de uma peça fundamental chamada computador.

## 1.10 CONCLUSÕES

O computador é um dos desenvolvimentos humanos mais significativos, pelo impacte que já teve nos últimos 50 anos da sua história. Muitas tarefas já foram automatizadas, em particular ao nível do processamento de informação, permitindo às pessoas desempenhar tarefas de maior nível.

O desenvolvimento tem sido exponencial, em boa parte devido à tecnologia que tem permitido produzir circuitos integrados cada vez mais rápidos e com maiores capacidades. Isto tem permitido fazer programas mais interactivos, mais agradáveis e fáceis de utilizar e com mais funcionalidades.

Os computadores pessoais têm sido os maiores responsáveis pela massificação do uso dos computadores, por terem um impacte directo no dia-a-dia de cada um de nós, e aqueles em que o esforço de produção de ferramentas de trabalho e de lazer tem sido mais intenso, dado o volume de vendas de muitos milhões por ano.

No entanto, apesar de todos estes desenvolvimentos, os computadores continuam a usar o mesmo modelo de programação, executando laboriosamente uma instrução a seguir à outra, sem terem noção macroscópica do que estão a fazer. Cabe ao programador ter a visão de conjunto, tentando prever todas as situações possíveis, e programar o computador de forma precisa com uma sequência de instruções simples, que este seja capaz de executar.

Apesar de tudo, são imbatíveis na sua rapidez de cálculo, capacidade de memória, fiabilidade, disponibilidade e custo. Os computadores estão para ficar, e o futuro dirá se poderão ficar intrinsecamente inteligentes a ponto de falarmos directamente com eles na nossa língua natural. Para já, o futuro mais imediato irá permitir novas aplicações através da integração do computador com dispositivos de comunicação, de lazer e de controlo (telemóveis, televisões, electrodomésticos, equipamento audiovisual, etc.).

# 2. O MUNDO BINÁRIO

Os circuitos electrónicos começaram realmente a sua história em 1906, quando Lee de Forest inventou a válvula electrónica com controlo de corrente, com um eléctrodo (o catodo) aquecido ao rubor e que emite eléctrons para outro eléctrodo (anodo), passando por um terceiro eléctrodo colocado entre os dois (grelha) que permite controlar a corrente de eléctrons de forma a amplificar o sinal aplicado à grelha. Em 1947 apareceu o primeiro transistor com iguais capacidades mas maisável, com menores tensões e muito menor consumo. Quer a válvula quer o transistor permitem amplificar sinais analógicos (com uma gama contínua de valores de tensão e corrente), como por exemplo áudio e vídeo.

Os computadores são implementados com circuitos electrónicos digitais, chamados assim por oposição aos circuitos analógicos, indicando que se usam apenas alguns valores possíveis e não uma gama contínua. Para ser mais simples de implementar, usam-se apenas dois valores, o mínimo (0) e o máximo permitido pelo circuito (1), o que permite tratar os circuitos como binários perfeitos e aplicar as regras da Álgebra de Boole, cujos princípios foram estabelecidos pelo matemático inglês George Boole em 1854.

Nesta álgebra existem variáveis, que podem tomar um de dois valores, 0 e 1, e três operações básicas: conjunção (AND), disjunção (OR) e negação (NOT). Com esta base, é possível especificar funções binárias arbitrariamente complexas. Introduzindo a noção de estado, que pode evoluir ao longo do tempo, é ainda possível construir sistemas binários complexos, desde simples contadores até aos microprocessadores mais recentes.

Do ponto de vista dos computadores, o importante são os dados, que representam informação sob a forma de números binários, e o processamento que se pode fazer sobre esses dados. Assim, este capítulo tem por objectivo fazer uma introdução aos sistemas digitais binários, com dois aspectos fundamentais:

- Funcionamento dos blocos binários básicos usados para construir microprocessadores;
  - Representação de números em base 2 (com apenas dois símbolos, 0 e 1) e conversão entre números binários e em base 10 (a que nós, humanos, estamos mais habituados).
- Este capítulo faz apenas uma breve introdução a este tema, como base para a compreensão dos blocos constituintes dos microprocessadores nos capítulos que se seguem. Outros livros tratam este assunto com mais detalhe, como por exemplo [Baptista 2002], [Mano 2003] e [Fathal 2003].

Os circuitos digitais mais simples são as portas lógicas básicas, que implementam directamente as operações atrás enunciadas (AND, OR e NOT). Com base neles é possível construir outros blocos mais complexos, como por exemplo somadores (que permitem aos computadores efectuar cálculos complexos) e básculas (que permitem construir células de memória). Este capítulo descreve apenas os blocos mais relevantes. Só no capítulo 3 se verá como se pode construir um computador com base nesses blocos.

## 2.1 CIRCUITOS ELECTRÓNICOS ANALÓGICOS

Os circuitos electrónicos são basicamente amplificadores. Um receptor de rádio (parte receptora de um telemóvel, por exemplo) amplifica os débeis sinais recebidos na antena até o áudio poder ser recuperado, e este é depois amplificado até um nível suficiente para ser reproduzido no altifalante.

Os sinais em causa são analógicos, isto é, podem tomar um valor de uma gama contínua entre dois valores extremos. Um sinal de áudio, por exemplo, é uma forma de onda complexa com várias frequências sobrepostas, entre cerca de 20 Hz e 20 KHz (gama de frequências a que o ouvido humano é sensível, embora a gama efectiva dependa da pessoa e da idade). A frequência de 1 Hz corresponde a um ciclo completo (até voltar a repetir o padrão) por cada segundo.

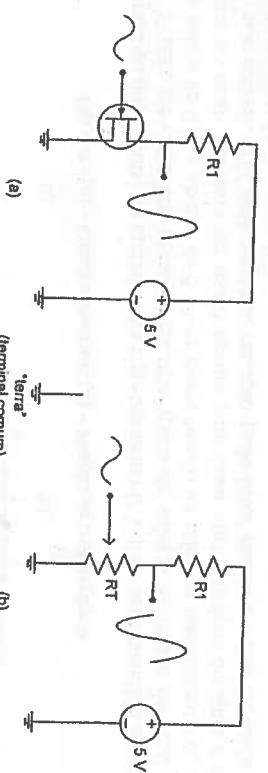


Fig. 2.1 - Amplificador de sinais analógicos. (a) – Com um transistor e uma resistência variável;

(b) – O transistor comporta-se como uma resistência variável,

controlada pelo sinal de entrada.

Esta situação pode ser comparada a um depósito de água elevado (correspondente à fonte de alimentação de 5 V) do qual sai um tubo de descarga (que equivale a R<sub>1</sub> e cujo diâmetro faz oferecer alguma resistência à passagem da água) com uma torneira (que desce o seu valor, respectivamente, deixando passar mais ou menos água (que equivale à corrente no circuito)). A torneira é assim uma resistência variável ao fluxo de água.

A tensão de saída (entre R<sub>1</sub> e RT e que corresponde à pressão da água no exemplo do depósito) pode ser obtida por uma simples proporção dos 5 V da fonte.

<sup>5</sup> Volt, unidade de medida de tensão eléctrica.

$$\text{Tensão saída} = \frac{RT}{R_1 + RT} \times 5\text{V}$$

Este circuito amplifica porque uma variação no sinal de entrada origina uma variação em RT proporcionalmente muito maior.<sup>6</sup> Note-se que as variações do sinal de saída (Fig. 2.1):

- São de maior amplitude do que as do sinal de entrada (devido à amplificação);
- Estão invertidas em relação ao sinal de entrada, uma vez que, quando o sinal de entrada sobe, RT (e a tensão de saída) desce.

## 2.2 CIRCUITOS ELECTRÓNICOS DIGITAIS

### 2.2.1 FUNCIONAMENTO BÁSICO

Os circuitos electrónicos analógicos não são adequados para construir computadores, pois são apenas capazes de efectuar transformações aos sinais (como a amplificação) e não se consegue lidar facilmente com valores muito precisos de forma fiável (armazena-los numa memória, por exemplo).

Nem os seres humanos lidam facilmente com grandezas contínuas. Todos os seus registos se baseiam em alfabetos de símbolos em número finito, como os números (10 dígitos) e os textos (26 letras e mais alguns símbolos de pontuação).

Poder-se-ia conceber um computador que funcionasse apenas com tantos níveis de tensão quantos os símbolos usados pelos seres humanos, mas tal seria muito difícil de implementar em termos electrónicos. Assim, adopta-se uma concepção mais radical e muito mais fácil, com apenas dois valores: o máximo (5 V, no caso da Fig. 2.1), também conhecido por "1", e o mínimo (0 V), também conhecido por "0". Nas secções seguintes ver-se-á como é que com apenas estes dois símbolos (0 e 1) se podem construir sistemas arbitrariamente complexos.

Havendo apenas dois símbolos, estes circuitos deviam designar-se por binários. No entanto, é vulgar designá-los por digitais (de "dígito", ou "dado"), indicando que, tal como na numeração decimal, se constroem números arbitrariamente grandes com apenas um conjunto reduzido (neste caso, 2) de símbolos bem determinados. A Tabela 1.1 já mostrou como se podem representar números superiores a 1, juxtapondo vários destes símbolos. Este termo tem raízes históricas e não quer dizer que estes circuitos lidem com 10 símbolos diferentes!

Note-se ainda que os circuitos não deixaram de ser analógicos. A restrição que se faz foi apenas limitar a gama de valores possíveis a 2, em regime estacionário (Fig. 2.2a). Mas na altura da transição (regime transitório), a saída passa pelos valores todos entre os dois extremos (Fig. 2.2b).

<sup>6</sup> Dependendo dos transistores, na ordem das dezenas ou centenas de vezes.

Dada a variação invertida de  $RT$  face ao sinal de entrada, este circuito funciona como inversor. Um 0 à entrada produz um 1 à saída e vice-versa, tal como pode ser visto na função de transferência do circuito (sinal de saída em função do sinal de entrada) representada na Fig. 2.2b. Na equação anterior, que dá a tensão de saída,  $RT$  varia entre 0 e infinito.

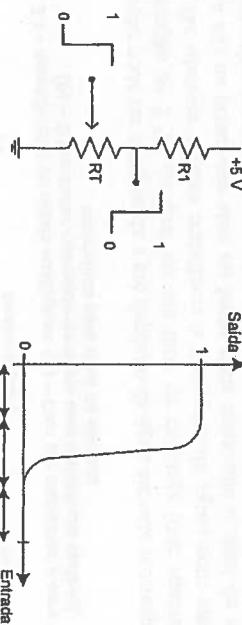


Fig. 2.2 - (a) – Circuito Inversor; (b) – Função de transferência

Não é preciso que o sinal de entrada seja rigorosamente 0 V para ser entendido como 0 nem 5 V para ser entendido como 1. Há uma gama de valores da tensão do sinal de entrada acima de 0 V e abaixo de 5 V em que a tensão de saída se mantém estável. Há uma zona no meio que deve ser usada apenas para fazer uma transição do sinal e não como valor estável, pois uma pequena variação pode fazer mudar o valor da saída de forma não controlada.

Esta folga de valores de tensão de entrada (quer a 0, quer a 1) em que a tensão de saída se mantém, mesmo que a tensão de entrada varie um pouco é designada margem de ruído e é extremamente importante para a estabilidade de comportamento dos sistemas digitais. O ambiente está cheio de ruído electromagnético, produzido por pequenas faiscas que ocorrem nos motores e nos interruptores. Essas faiscas produzem ondas electromagnéticas (como as de rádio e televisão) que se induzem nos fios de ligação dos circuitos (como se fossem antenas) e provocam pequenas variações dos sinais. Enquanto estas não forem superiores à margem de ruído, não haverá qualquer consequência. Caso contrário, um 0 poderá ser entendido como um 1 ou vice-versa, e isso é o suficiente para um computador se baralhar completamente. Felizmente, devido à margem de ruído, esta situação é muito rara.

Assim, o inversor é encarado como digital (ou binário) puro, com apenas dois estados possíveis (0 e 1) e em que se assume que o transistor se comporta como um interruptor comandado pelo sinal de entrada, uma vez que a sua resistência varia entre um valor nulo (Fig. 2.3a – entrada com valor 1, interruptor fechado, saída com valor 0 porque fica ligada directamente à “terra”) e infinito (Fig. 2.3b – entrada com valor 0, interruptor aberto, saída com valor 1).

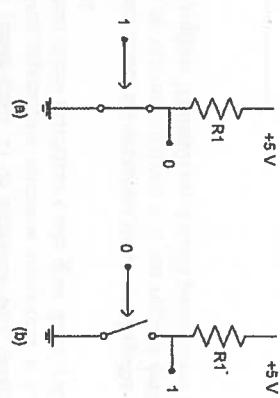
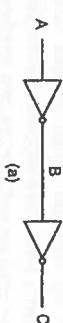


Fig. 2.3 - Circuito Inversor binário, (a) – Com 0 à entrada e 1 à saída; (b) – Situação inversa

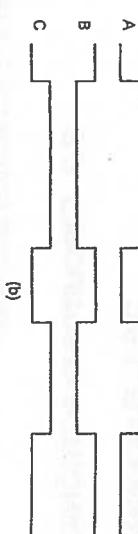
## 2.2.2 DIAGRAMAS TEMPORAIS

Para evitar estar sempre a representar todos os detalhes internos de um inversor (que na realidade até é mais complexo do que o representado na Fig. 2.2a), usa-se uma notação mais compacta, ilustrada pela Fig. 2.4a, que representa dois inversores seguidos. O sinal B está invertido em relação ao sinal A, o que significa que o sinal C é igual ao A.

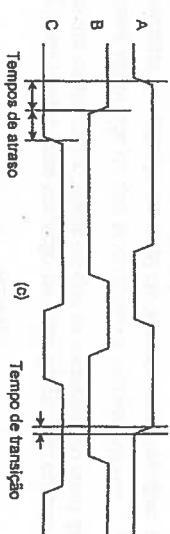
Isto mesmo pode ser visto na Fig. 2.4b, que representa um diagrama temporal dos sinais A, B e C. O eixo horizontal é o tempo e o vertical é o valor, 0 ou 1 (para cada um dos sinais). Os eixos não são representados por simplicidade. Pode verificar-se que os sinais B e C acompanham as variações do sinal de entrada (A) e de acordo com o funcionamento do inversor. Os diagramas temporais são óptimos para ilustrar o comportamento do circuito em várias situações, através da variação dos seus sinais de entrada.



(a)



(b)



(c)

Fig. 2.4 - Comportamento de dois inversores, (a) – Circuito; (b) – Diagrama temporal simplificado; (c) – Diagrama temporal detalhado

Este é um diagrama ideal, mas a prática é diferente, tal como ilustrado pela Fig. 2.4c, que reflecte melhor o funcionamento real do circuito:

- O inverter demora sempre algum tempo a reagir a uma transição do seu sinal de entrada. Por isso, o seu sinal de saída tem sempre um atraso (designado tempo de atraso) em relação ao sinal de entrada. Circuitos em série somam os seus atrasos, como se pode ver pelo sinal C em relação a A.

Dado que os circuitos trabalham com sinais que variam muito rapidamente, os tempos de atraso não são desprezáveis. São efeitos que limitam a frequência de trabalho dos computadores. Se por exemplo o sinal de entrada variar de 0 para 1 e para 0 outra vez de forma sensivelmente mais rápida do que o tempo de atraso do inverter, este pode nem dar pela transição, pois nem tem tempo de reagir;

- Os tempos de transição são mais pequenos que os de atraso, mas também não são zero. São importantes apenas para quem projecta estes circuitos electrónicos.

Estas considerações são intrínsecas à tecnologia da electrónica e por conseguinte aplicam-se a qualquer circuito electrónico digital.

Para explicar o comportamento dos circuitos em termos de funcionalidade, sem preocupações temporais, usam-se diagramas como os da Fig. 2.4b, que são mais simples. Para ver bem os detalhes, nomeadamente os de Indute temporal (para confirmar se o circuito tem o tempo necessário para reagir, por exemplo) usam-se diagramas como os da Fig. 2.4c. A Fig. 6.21, na página 453, apresenta um exemplo mais completo.

### 2.2.3 PORTAS LÓGICAS

O inverter é o circuito digital mais básico, mas não permite combinar mais do que um sinal de modo a implementar funcionalidades mais complexas do que uma simples inversão, razão pela qual se usam outros circuitos básicos, designados portas lógicas. "Lógicas" porque com dois símbolos se pode encarar o 0 como FALSO e 1 como VERDADEIRO<sup>7</sup>, e usar as operações usuais da lógica binária (AND, OR e NOT – E, OU e NEGAÇÃO). "Portas" porque uns sinais podem bloquear outros (por exemplo, um sinal de entrada de um AND com o valor 0, ou FALSO, força a saída desse AND a 0 independentemente do valor da(s) outra(s) entrada(s).

A Tabela 2.1 mostra as características básicas de algumas portas lógicas, em que:

- Os sinais também são designados variáveis (termo útil para a secção 2.3), por poderem variar o seu valor ao longo do tempo, ou por *bits* (*binary digit*, ou dígito binário) por poderem tomar um de apenas dois valores;
- A porta NOT (negação) inverte (nega) o valor da entrada x. Se x=0 (FALSO), a saída z fica com o valor 1 (VERDADEIRO), e vice-versa;

<sup>7</sup> A associação ao contrário (0 ser VERDADEIRO e 1 ser FALSO) também é perfeitamente legítima, mas menos intuitiva.

PORTA	SÍMBOLOS	FUNÇÃO	TABELA DE VERDADE
NOT	X → Z	X → Z	Z = X̄
AND	X → Y → Z	X → Y → Z	Z = X · Y
OR	X → Y → Z	X → Y → Z	Z = X + Y
NAND	X → Y → Z	X → Y → Z	Z = X · Y
NOR	X → Y → Z	X → Y → Z	Z = X + Y
XOR	X → Y → Z	X → Y → Z	Z = X ⊕ Y

Tabela 2.1 - Características de algumas portas lógicas

Na coluna "Função", a barra horizontal por cima de uma variável ou expressão significa a sua negação, tal como exemplificado pela porta NOT ( $z = \bar{x}$ ); A coluna "Tabela de verdade" apresenta uma descrição tabular extensiva de cada porta, indicando qual o valor da saída z para cada uma das combinações possíveis dos valores dos sinais de entrada (x e y);

A coluna "Símbolos" apresenta o símbolo gráfico para representar cada uma das portas. Existem duas alternativas: uma mais clássica, com um símbolo diferente para cada porta, e outra, mais recente, com uma simbologia mais uniforme (retângulo), em que a função é indicada pela pequena expressão dentro do retângulo. Ambas são reconhecidas pela norma de símbolos para funções lógicas [IEEE 1984]. Este livro segue a primeira alternativa por ser a mais expressiva em termos gráficos (é a geralmente preferida pelos projectistas e desenhistas em termos de documentação);

A porta AND só tem a sua saída a 1 quando ambas as entradas forem 1 ( $x=1$  e  $y=1$ ), tal como se pode ver na tabela de verdade. Esta função também é conhecida por conjunção, o que quer dizer que só quando se der a conjunção das duas entradas serem 1 é que a saída z fica a 1. A operação correspondente, na função, é representada pelo símbolo “.”, que não deve ser confundido com a multiplicação, pois a operação que está aqui em causa é lógica e não aritmética;

A porta OR tem a sua saída a 1 quando qualquer das entradas for 1 ( $x=1$  ou  $y=1$ ), tal como se pode ver na tabela de verdade. Esta função também é conhecida por disjunção, o que quer dizer que não precisa de haver conjunção das duas entradas a 1 para a saída z ficar a 1. A operação correspondente, na função, é representada pelo símbolo “+”, mas não deve ser confundida com a soma vulgar, pois  $1+1=1$  (em termos lógicos, não aritméticos);

A porta NAND é equivalente a uma porta AND seguida de um NOT. Graficamente, isso é indicado pela pequena bolha na saída da porta. Na função, a barra horizontal ( $z = \overline{x \cdot y}$ ) indica negação da expressão AND. Na tabela de verdade, os valores de z estão negados face aos da porta AND;

*Idem* para a porta NOR, mas em relação à porta OR;

A porta XOR (Exclusive OR, ou “OU Exclusivo”) é semelhante a uma porta OR mas com a diferença de que se elimina a possibilidade de ambas entradas terem o valor 1. Isto é, são mutuamente exclusivas. Outra forma de ver a questão é reparar que a saída z só é 1 quando as entradas tiverem valores diferentes, tal como se pode verificar na tabela de verdade. O símbolo para esta função é “ $\oplus$ ”;

A porta NOT só pode ter uma entrada, mas as restantes podem ter qualquer número de entradas, mantendo a semântica. As portas só têm uma saída.

Estas portas podem ser combinadas em circuitos arbitrariamente complexos, em que a saída de uma porta liga à entrada de outra, tal como se vê nas secções seguintes.

A Fig. 2.5 mostra a implementação (simplificada) de duas portas lógicas, NAND e NOR, tornando como base o inverter (porta NOT) da Fig. 2.3, e a evolução temporal das respectivas saídas quando as suas entradas passam pelas várias combinações possíveis. A saída do NAND (Fig. 2.5a) só é 0 quando ambas as entradas estão a 1, pois só nessa altura ambos os “interruptores” estão fechados, ligando a saída à “terra”. Pelo contrário, a saída do NOR é 0 desde que qualquer das entradas esteja a 1, pois qualquer delas consegue fechar o seu “interruptor” e ligar a saída à “terra” independentemente da outra. Dada a natureza inversora dos transistores, estes são os circuitos mais simples com duas entradas, o que quer dizer que as portas lógicas AND e OR se implementam colocando um NOT a seguir a um NAND e a um NOR, respectivamente. Exactamente ao contrário do que poderia parecer à primeira vista (um NAND seria um AND ligado a um NOT, e um NOR seria um OR ligado a um NOT), mas ambas as alternativas são funcionalmente equivalentes.

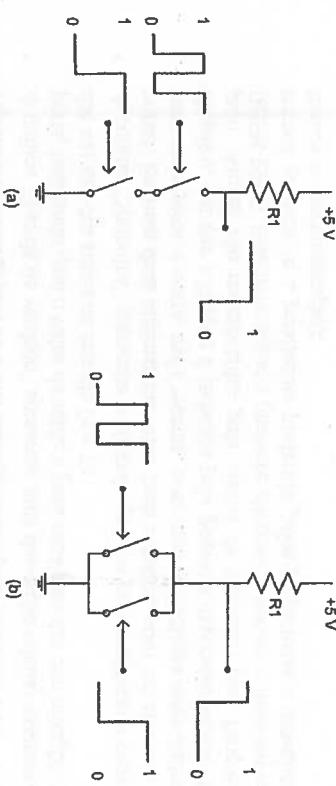


Fig. 2.5 - Implementação simplificada de duas portas lógicas. (a) - NAND; (b) - NOR

### SIMULAÇÃO 2.1 – PORTAS LÓGICAS

A melhor forma de aprender o funcionamento de qualquer sistema é fazer experiências com ele. Existem implementações físicas das portas lógicas, em circuitos integrados, mas nem sempre existem as condições apropriadas para usar uma implementação física.

Felizmente, existem simuladores (em computador) que permitem reproduzir de forma muito realista o comportamento destes sistemas. Todo este livro está baseado nesta perspectiva. Em vez de se limitar a descrever as diversas técnicas, o livro está recheado de referências a guiaços práticos (como esta Simulação 2.1) que permitem ao leitor experimentar, na prática, os sistemas e técnicas que acabou de descobrir. Estes guiaços envolvem detalhes que ocupariam demasiado espaço no texto do livro, razão pela qual se encontram disponíveis no site de apoio ao livro ([www.fca.pt](http://www.fca.pt)). O Apêndice C, na página 725, contém uma breve introdução às capacidades do simulador.

Esta simulação inclui os seguintes aspectos:

- Introdução ao simulador, com construção (no simulador) dos pequenos circuitos de teste das portas lógicas;
- Utilização de interruptores como dispositivos de entrada (produzindo 0 ou 1, bastando um clique com o rato para mudar o valor) e LEDs<sup>8</sup> para visualizar o estado das saídas;
- Verificação da tabela de verdade das várias portas lógicas, com várias entradas;
- Verificação de algumas equivalências (por exemplo, uma porta NAND e o seu equivalente, uma porta AND ligada a uma porta NOT);
- Verificação do tempo de atraso das portas lógicas.

<sup>8</sup> LED (Light Emitting Diode) – Dispositivo que produz luz quando percorrido por uma corrente eléctrica, na ordem de 3 a 30 mA.

**ESSENCIAL**

- Os circuitos electrónicos digitais são circuitos analógicos que usam apenas duas gamas de valores de tensão situadas nos extremos da gama total de valores possíveis. Qualquer valor na gama binária é um 0 e qualquer valor na gama alta (perto do valor da tensão de alimentação) é um 1.
- A transição de um 0 para um 1 e vice-versa **gasta tempo e energia**. Isto limita a frequência máxima de operações de um circuito digital e por conseguinte de um computador.
- Uma porta lógica é um circuito destes cujo valor de saída depende de uma ou mais entradas. A porta NOT troca a entrada ( $s_1$ ) de 0 e vice-versa. A porta AND só é uma porta AND se todas as entradas forem 1. A porta OR tem 1 na saída desde que pelo menos uma das entradas tenha um 1.
- Uma tabela de verdade é uma tabela que indica o valor da(s) saída(s) para cada combinação dos valores das entradas. Um diagrama temporal permite ver as variações dos sinais de saída ao longo do tempo em função das variações das entradas de entrada.

## 2.3 ÁLGEBRA DE BOOLE

Em 1854, Boole<sup>9</sup> publicou um tratado em que descrevia uma álgebra com dois valores (0 e 1) e três operações (AND, OR e NOT), com a representação e significado já introduzido pelas primeiras três portas lógicas da Tabela 2.1. Em 1938, Shannon<sup>10</sup> propôs usar esta álgebra no projecto de sistemas electrónicos digitais.

**NOTA** Na realidade, na altura já havia válvulas electrónicas, mas eram usadas essencialmente para amplificação de sinais analógicos. Pouco depois, durante a 2.ª Guerra Mundial, começaram a ser usadas para construir o ENIAC, mas os relés (interruptores comandados de forma electromecânica) foram realmente os primeiros "utilizadores" digitais da álgebra de Boole. O primeiro transistor só apareceu em 1947.

Nestas expressões, a operação com maior precedência (a que tem de ser efectuada primeiro) é a negação (NOT, barra horizontal), seguindo-se o AND ("·") e finalmente o OR ("+").

Podem usar-se parênteses curvos para alterar as regras de precedência, pois uma operação só pode ser efectuada depois de calcular os seus operandos. Se um operando de uma operação for uma expressão entre parênteses, esta tem de ser calculada primeiro. Assim, a equação anterior é equivalente a

$$Z = X + (Y \cdot \bar{V})$$

mas não a, por exemplo,

$$Z = (X + Y) \cdot \bar{V}$$

em que  $Z$  será 1 apenas quando pelo menos uma das variáveis  $X$  e  $Y$  for 1 e simultaneamente a variável  $V$  for 0. Para simplificar a notação, e sempre que não haja ambiguidades de nomes de variáveis, o operador ":" pode omitir-se, simplificando as expressões. Exemplos:

$$Z = X + Y \bar{V}$$

$$Z = (X + Y) \bar{V}$$

$$Z = XY\bar{V} + \bar{X}\bar{Y}V$$

$$Z = (XY) \oplus \bar{V}$$

Para além dos conjuntos dos valores e das operações, uma álgebra inclui ainda um conjunto de axiomas que definem as propriedades das operações sobre os valores. A Tabela 2.2 indica os axiomas da álgebra de Boole.

## 2.4 FUNÇÕES LÓGICAS

A álgebra de Boole é importante para dar uma base matemática ao funcionamento dos circuitos electrónicos numa base binária, como ilustrado pela Fig. 2.3. Existe um mapeamento directo das operações desta álgebra nas portas lógicas, tal como descritas na secção 2.2.3. Por outro lado, a notação de representação das funções constitui uma alternativa à especificação de sistemas digitais que pode ser mais compacta do que as tabelas de verdade introduzidas na Tabela 2.1.

Imaginemos, por exemplo, um sistema digital com 4 entradas e 2 saídas, descrito pela tabela de verdade especificada na Tabela 2.3. Apesar de ser um sistema extremamente simples quando comparado com um computador, a tabela já é um pequeno mar de 0s e 1s que não deixa antever minimamente qual a sua funcionalidade. Note-se que  $N$  entradas permitem  $2^N$  combinações diferentes (neste caso, 4 entradas originam 16 combinações).

$$Z = X + Y \cdot \bar{V}$$

<sup>9</sup> George Boole, matemático inglês do século XIX (1815-1864).

<sup>10</sup> Claude Shannon, engenheiro electrotécnico e matemático norte-americano (1916-2001).

Nome	Símbolização	Comentário
IDENTIDADE	$X + 0 = X$ $X \cdot 1 = X$	Elementos neutros das operações
INVERSO	$X \cdot \bar{X} = 0$ $X + \bar{X} = 1$	As operações sobre valores complementares dão sempre o mesmo resultado
INVOLUÇÃO	$\bar{\bar{X}} = X$	A negação alterna entre os dois valores possíveis. Duas negações dão o mesmo valor
IDEMPOTÊNCIA	$X + X = X$ $X \cdot X = X$	As operações não são antinéticas
COMUTATIVIDADE	$X + Y = Y + X$ $X \cdot Y = Y \cdot X$	
ASSOCIATIVIDADE	$X + (Y + V) = (X + Y) + V$ $X \cdot (Y \cdot V) = (X \cdot Y) \cdot V$	Estas propriedades já existem na álgebra dos números inteiros
DISTRIBUTIVIDADE	$X \cdot (Y + V) = X \cdot Y + X \cdot V$ $X + (Y \cdot V) = (X + Y) \cdot (X + V)$	
DUALIDADE (TEOREMA OU LEIS DE MORGAN <sup>11</sup> )	$\overline{X + Y} = \bar{X} \cdot \bar{Y}$ $\overline{X \cdot Y} = \bar{X} + \bar{Y}$	A negação da soma é o produto das negações, e a negação do produto é a soma das negações

Tabela 2.2 - Axiomas da álgebra de Boole

Este sistema deve ser implementado por um conjunto de portas lógicas que cumpra esta tabela de verdade. A forma mais usual é derivar de forma independente duas expressões (uma para  $Z_1$  e outra para  $Z_2$ ), cada uma delas como uma soma de produtos (isto é, um OR de termos, cada um constituído por um AND de várias variáveis). Também é possível usar um produto de somas (dada a dualidade da álgebra de Boole, mas é menos intuitiva porque os produtos (ANDs) têm precedência sobre as somas (ORs) e por isso esta última forma obriga a parênteses).

Na soma (OR) de produtos (ANDs), cada produto corresponde a uma das combinações de variáveis de entrada ( $A, B, C$  e  $D$ ), tal como indicado na Tabela 2.4, e designa-se termo mínimo. Em cada termo mínimo, cada variável de entrada aparece negada se o valor dessa variável na combinação correspondente estiver a 0. A função que produz os valores de uma dada saída ( $Z_1$  ou  $Z_2$ , por exemplo) é constituída pela soma (OR) de todos os termos mínimos em que essa saída deva estar a 1. Os termos mínimos correspondentes às combinações das variáveis de entrada em que a saída esteja a 0 não devem ser incluídos. Para uma dada combinação das variáveis de entrada, apenas um termo mínimo pode ter o valor 1. Os restantes são obrigatoriamente 0.

<sup>11</sup> Augustus De Morgan, matemático inglês do século XIX (1806-1871).

Tabela 2.3 - Exemplo de uma tabela de verdade de um sistema digital com 4 entradas (16 combinações de valores diferentes) e 2 saídas

Nestas condições, a saída  $Z_1$  pode ser descrita pela seguinte função com 11 termos mínimos (aqueelas combinações das variáveis de entrada em que a função  $Z_1$  vale 1):

$$\begin{aligned} Z_1 = & \bar{A} \bar{B} \bar{C} \bar{D} + \bar{A} \bar{B} \bar{C} D + \bar{A} \bar{B} C \bar{D} + \bar{A} B \bar{C} \bar{D} + A \bar{B} \bar{C} \bar{D} + A \bar{B} C \bar{D} + \\ & + A \bar{B} \bar{C} D + A \bar{B} C D + A B \bar{C} \bar{D} + A B C \bar{D} \end{aligned}$$

Esta função poderia ser implementada simplesmente por um OR de 11 entradas, 11 ANDs de 4 entradas cada um e 4 NOTs (para negar as variáveis de entrada). No entanto, as funções complexas podem normalmente ser simplificadas, usando os axiomas da álgebra de Boole. Por exemplo, os dois primeiros termos podem ser reduzidos:

$$\bar{A} \bar{B} \bar{C} \bar{D} + \bar{A} \bar{B} \bar{C} D = \bar{A} \bar{B} \bar{C} (\bar{D} + D) = \bar{A} \bar{B} \bar{C}$$

uma vez que  $\bar{D} + D$  tem sempre o valor 1. Outras simplificações são possíveis, mas esta forma de efectuar simplificação de funções (simplificação algébrica) é tediosa e muito sujeita à erros. Felizmente existem já programas que simplificam automaticamente as funções, mas para simplificações manuais o melhor processo consiste em elaborar mapas de Karnaugh, um para cada uma das variáveis de saída.

ENTRADAS				SAÍDAS	
A	B	C	D	Z1	Z2
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	1	1
1	1	1	1	1	1

TERMOS MÍNIMOS	ENTRADAS				SAÍDAS	
	A	B	C	D	Z1	Z2
A B C D	0	0	0	0	1	1
A B C D̄	0	0	0	1	1	1
A B̄ C D	0	0	1	0	1	1
A B̄ C D̄	0	0	1	1	1	1
A B C̄ D	0	1	0	0	1	0
A B C̄ D̄	0	1	0	1	0	0
A B̄ C̄ D	0	1	1	0	1	0
A B̄ C̄ D̄	0	1	1	1	1	1
A B C D̄	1	0	1	0	1	1
A B C̄ D̄	1	0	1	1	1	1
A B̄ C D̄	1	0	0	1	1	0
A B̄ C̄ D̄	1	0	1	1	1	1
A B C̄ D	1	1	0	0	0	0
A B C̄ D̄	1	1	0	1	0	0
A B C D̄	1	1	1	0	1	1
A B C̄ D̄	1	1	1	1	1	1

Tabela 2.4 - Exemplo da Tabela 2.3, explicitando todos os termos mínimos possíveis.

Para a função de uma saída só contam aqueles em que a saída vale 1.

Um mapa de Karnaugh não é mais do que uma representação matricial, para uma dada variável de saída, dos valores que essa variável toma para cada uma das combinações das variáveis de entrada. No seu conjunto, estes valores definem a função que transforma as variáveis de entrada na variável de saída.

A Fig. 2.6 ilustra estes mapas para várias funções com 4 variáveis de entrada (duas variáveis em cada eixo). Cada quadrado corresponde a um dos 16 termos mínimos possíveis, em que para maior clareza os valores 0 da função foram omitidos e apenas os termos mínimos em que o valor da função é 1 estão explicitamente referidos.

Um dos aspectos primordiais de um mapa de Karnaugh é a sequência de combinações dos valores das variáveis em cada eixo. Em vez de 00, 01, 10 e 11 (contagem crescente), usase 00, 01, 11 e 10 A troca de ordem de 10 com 11 permite que entre quaisquer combinações adjacentes (consequentes) em cada eixo apenas uma variável de cada vez mude o seu valor, mesmo considerando os extremos de cada eixo como adjacentes (isto é, dando “volta”).

Este pormenor permite reconhecer visualmente grupos de 1s adjacentes em torno de eixos de simetria (em que apenas uma variável mude – ver descrição a seguir) e simplificar os termos correspondentes.

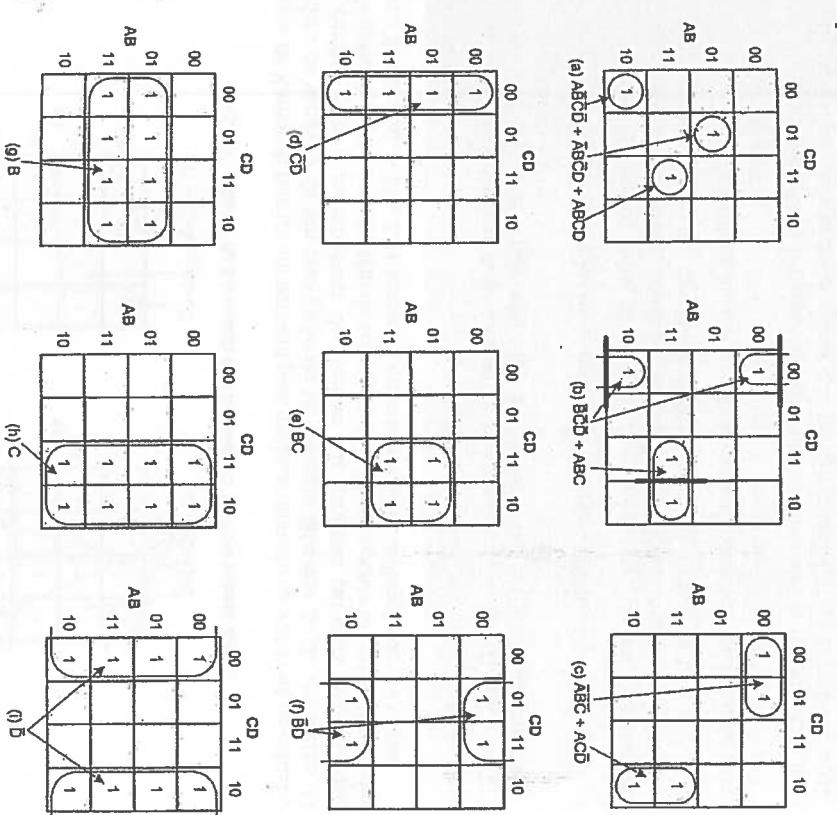


Fig. 2.6 - Exemplos de mapas de Karnaugh de 4 entradas

Analisemos cada um dos casos da Fig. 2.6 (as alíneas identificam cada um dos mapas na figura):

- Trata-se de uma função com apenas três termos mínimos e a sua expressão é referida por baixo da tabela. Não é possível qualquer simplificação porque entre qualquer dois termos mínimos não há nenhum caso de eixo de simetria (dois 1s correspondentes a combinações de entrada adjacentes, em que dum para outra mude apenas uma das variáveis de entrada);

b) Aqui a situação já é diferente. Esta função tem quatro termos mínimos, mas podem ser reunidos em dois grupos de dois termos cada um, em torno de eixos de simetria (representados por traços mais grossos). Os dois termos mais à esquerda são  $\bar{A} \bar{B} C D$  (em cima) e  $A \bar{B} \bar{C} D$  (em baixo), que dão apenas a variável  $A$ . Pela propriedade associativa, a sua soma é equivalente a  $(A + A) B \bar{C} D$ , ou  $B \bar{C} D$ , uma vez que  $A + A$  é sempre 1. Estes dois termos mínimos são adjacentes (diferem apenas numa variável de entrada), em torno do eixo de simetria que se situa nos extremos do eixo  $AB$ , que se tocam (o eixo é o mesmo). De igual modo, o segundo termo já simplificado resulta da reunião de dois termos adjacentes, em que a variável  $D$  muda e como tal é eliminada, ficando apenas  $A B C$ ;

c) Por simplicidade, os eixos de simetria deixam de ser representados, mas de novo há dois grupos de termos mínimos que podem ser simplificados. No primeiro é a variável  $D$  que varia, sendo eliminada, enquanto no segundo é a variável  $B$  que varia e é eliminada;

d) Neste caso, há um grupo de quatro 1s adjacentes. Apenas as variáveis  $C$  e  $D$  mantêm o seu valor em todos, pelo que são as únicas que ficam;

e) Idem, mas agora apenas as variáveis  $B$  e  $C$  mantêm os seus valores. As outras são eliminadas;

f) Idem, com simetria através dos extremos do eixo  $AB$ . Apenas as variáveis  $B$  e  $D$  mantêm o seu valor;

g) Agora são só os termos mínimos adjacentes. Apenas a variável  $B$  se mantém. Todas as outras são eliminadas da expressão da função;

h) Idem, a variável que se mantém é a  $C$ ;

i) Idem, com simetria pelo extremo do eixo  $CD$ . Apenas a variável  $D$  se mantém.

Os mapas de Karnaugh são uma forma visualmente expedita de reconhecer termos mínimos adjacentes e simplificá-los, eliminando variáveis em que tanto um valor como o seu negado estão presentes (cujo OR dá 1 e portanto pode ser eliminado porque este é o valor neutro do AND).

Voltando ao exemplo da Tabela 2.4, a Fig. 2.7 representa os mapas de Karnaugh das funções  $Z_1$  e  $Z_2$  (tendo em atenção que nos eixos 10 e 11 estão trocados em relação à simples contagem binária). Pode logo notar-se a grande simplicidade da função  $Z_1$  face aos 11 termos de 4 variáveis cada um que se teria se a função não fosse simplificada.

O método de simplificação é o mesmo que nos exemplos da Fig. 2.6, em que os vários grupos de 1s geram termos que se somam para dar a expressão completa da função. Note-se ainda que:

- Não há qualquer problema em alguns 1s serem partilhados entre termos diferentes, pois a mesma variável pode participar várias vezes numa operação (propriedade da idempotência, referida na Tabela 2.2);
- Deve-se sempre procurar que os grupos de 1s sejam o maior possível, dentro das regras da simetria. Por exemplo, o segundo termo da função  $Z_1$  poderia resultar

apenas do grupo de dois 1s no canto inferior esquerdo. Não estaria errado, mas no termo apareceria mais a variável  $A$ , desnecessariamente (a função não estaria tão simplificada quanto possível);

O primeiro termo de  $Z_2$  é curioso, pois usa os extremos de ambos os eixos como simetria. No entanto, apesar de a disposição física na tabela ser diferente, trata-se de um caso semelhante ao do primeiro termo de  $Z_1$ .

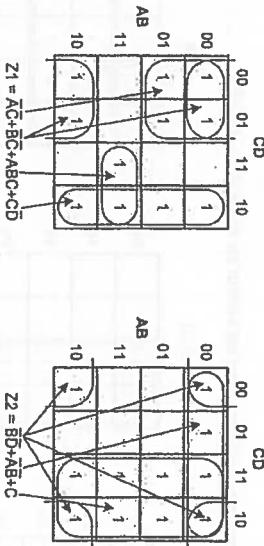


Fig. 2.7 - Mapas de Karnaugh das funções  $Z_1$  e  $Z_2$  da Tabela 2.4

Os mapas de Karnaugh funcionam também para qualquer número de variáveis de entrada. O número de variáveis de um eixo pode até ser bastante diferente do do outro eixo. O único cuidado a ter é na sequência dos valores das variáveis em cada eixo, em que combinações contíguas no eixo têm de ser adjacentes (diferem apenas no valor de uma das variáveis de entrada). A Fig. 2.31 apresenta outro exemplo de utilização destes mapas.

### ESSENCIAL

gebrade Boole tem dois valores (0 e 1) e três operações (AND, OR e NOT) que permitem descrever o funcionamento dos circuitos digitais que funcionam em binário (0 e 1).

Os logicos implementam as operações da álgebra de Boole.

Por vel [presentar sinais de saída em funão dos sinais de entrada por meio de expressões booleanas, como por exemplo  $Z = X + Y$ , em que as variáveis representam os sinais dos circuitos e podem tomar os valores 0 ou 1].

O termos que constituem uma expressão booleana podem ser derivados de uma bôl de verdade;

o gema de Boole inclui ainda axiomas que establecem as propriedades das operações, estabelecendo as regras que permitem transformar expressões noutras equivalentemente mais simples;

- mapa de Karnaugh são um meio expediente para simplificar manualmente expressões booleanas;
- mapa de Karnaugh são um meio expediente para simplificar manualmente expressões booleanas;

## 2.5 CIRCUITOS COMBINATÓRIOS

### 2.5.1 SÍNTSEDE CIRCUITOS COMBINATÓRIOS

Designa-se circuito combinatório todo o circuito digital sem realimentações, isto é, em que nenhuma entrada de uma porta lógica dependa, directa ou indirectamente, da sua própria saída.

Nestas condições, a(s) saída(s) do circuito são independentes umas das outras, no sentido de que cada saída depende exclusivamente das entradas. Para a mesma combinação de valores das variáveis de entrada, os valores das saídas serão sempre os mesmos, independentemente da evolução passada dos sinais de entrada. O circuito não tem estado interno e é funcional puro (cada saída é uma função exclusivamente das entradas).

A secção 2.6 discute circuitos com realimentações internas e em que as saídas dependem não apenas das variáveis de entrada mas também da forma como as saídas evoluíram no passado (estado interno do circuito). Devido a este último aspecto, esses circuitos são designados sequenciais.

Em termos abstratos, é possível implementar qualquer circuito combinatório, com qualquer número de entradas e de saídas, desde que se faça a sua tabela de verdade e se simplifiquem as funções das saídas. Depois é só converter as funções em portas lógicas, o que é directo a partir das negações, produtos e somas. A Fig. 2.8 ilustra o diagrama lógico (ou logigrana) do circuito cuja tabela de verdade consta da Tabela 2.4, usando as funções simplificadas derivadas pela Fig. 2.7 (senão, seria mais complexo).

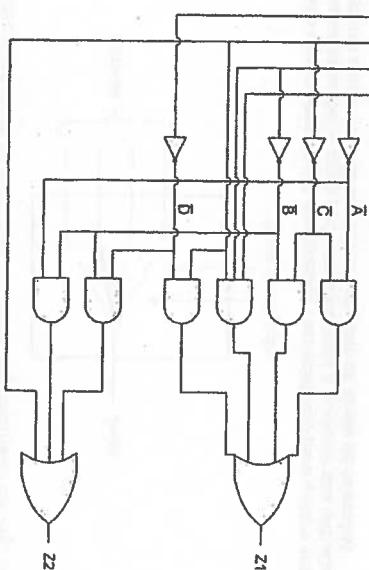


Fig. 2.8 - Logigrana do circuito correspondente às saídas Z1 e Z2 na Fig. 2.7

Este processo de projecto destes circuitos designa-se síntese de circuitos combinatórios e permite desenvolver circuitos optimizados em termos de portas lógicas. No entanto, o

esforço de desenvolvimento é apreciável, pois obriga a especificar completamente as possibilidades todas numa tabela de verdade, o que só é realmente prático para sistemas muito pequenos.

Em casos mais complexos, o mais usual é recorrer a módulos já feitos, com uma dada funcionalidade que é frequentemente necessária. Geralmente, é depois preciso recorrer a algumas portas à medida, para negar um sinal ou fazer um AND de dois sinais, por exemplo, mas com soluções relativamente simples. O circuito final pode ter um número de portas lógicas superior ao mínimo possível para a funcionalidade a implementar, mas a complexidade do desenvolvimento e a dificuldade de compreensão do circuito são mais reduzidas.

**NOTA** Existem também linguagens de descrição de hardware, como VHDL e Verilog [Bothros 2005], que permitem descrever os circuitos não extensivamente, pela sua tabela de verdade, mas pela sua estrutura e funcionalidade, como se de programas em hardware autónomica, em que um programa analisa as descrições dos módulos expressas em VHDL ou Verilog, gera as respectivas funções lógicas e optimiza-as, sem intervenção do projectista, como se fosse um compilador (só que gera circuitos, não instruções). Isto é particularmente utilizado no desenvolvimento de circuitos integrados, pelo que este programa recebe normalmente a designação de compilador de silício.

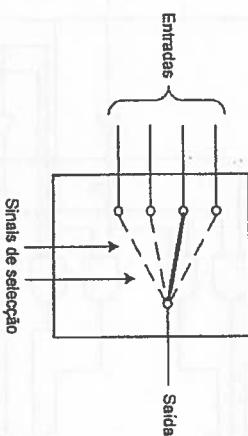
### SIMULAÇÃO 2.2 - CIRCUITOS COMBINATÓRIOS

Esta simulação ilustra o funcionamento dos circuitos combinatórios, usando o circuito da Fig. 2.8 como base. Os aspectos cobertos incluem os seguintes:

- Verificação do cumprimento da tabela de verdade da Tabela 2.4;
- Inspecção de valores das ligações intermédias;
- Observação dos picos causados pelo comportamento transitório dos valores das ligações enquanto decorrem as interacções das mudanças de sinais;
- Verificação do tempo de atraso total do circuito.

## 2.5.2 MULTIPLEXERS

Um *multiplexer* permite escolher entre uma de várias entradas e transportar o seu valor para uma saída, sob controlo de um ou mais sinais de selecção (tanto quantos os necessários para referenciar todas as entradas), tal como representado na Fig. 2.9.



**Fig. 2.9 - Circuito equivalente de um *multiplexer*, exemplificado para quatro entradas e dois sinais de seleção. A saída pode ser feita igual a qualquer das entradas, mas apenas a uma de cada vez (de acordo com os sinais de seleção)**

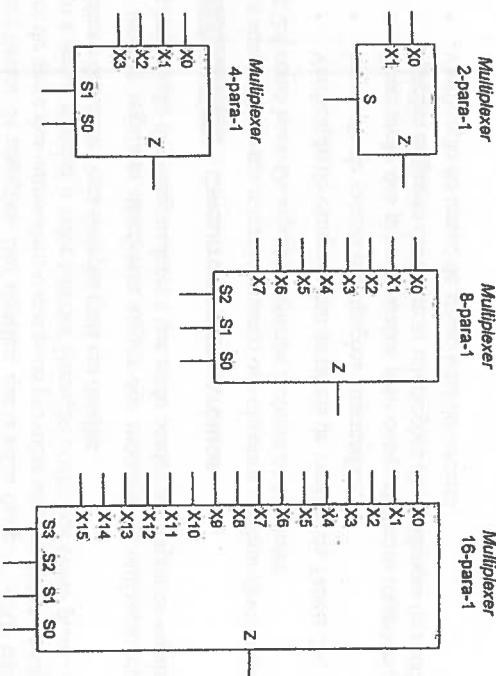
O número de entradas é normalmente uma potência de 2 (valores típicos: 2, 4, 8 e 16), pois N sinais de seleção permitem seleccionar  $2^N$  hipóteses diferentes, normalmente numeradas entre 0 e  $2^N-1$ , tal como indicado pela Tabela 2.5. Por exemplo, a Tabela 2.3 mostra que com 4 variáveis se podem ter 16 hipóteses diferentes ( $16=2^4$ ).

NÚMERO DE SÍMBOLOS DE SELEÇÃO	NÚMERO DE COMBINAÇÕES POSSÍVEIS	GAMA DE NÚMERAÇÃO DAS COMBINAÇÕES	
		MÍNIMO	MÁXIMO
1	$2^1$	2	0
2	$2^2$	4	0
3	$2^3$	8	0
4	$2^4$	16	0

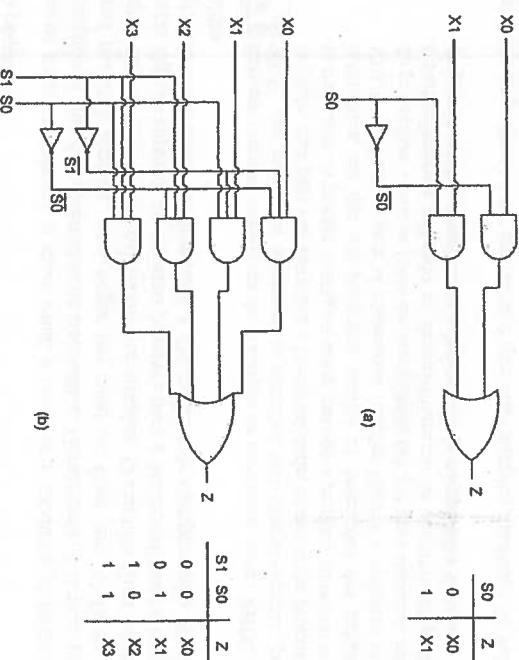
Tabela 2.5 - Número de entradas possível para um multiplexor para vários números

A Fig. 2.10 ilustra o símbolo que pode ser usado para representar *multiplexers* de 2 a 16 entradas ( $X$ ), com 1 a 4 sinais de selecção ( $S$ ). Quando há vários sinais relacionados, como as várias entradas e os vários sinais de selecção, é usual dar-lhes o mesmo nome, seguido de um número, que começa em 0. A saída ( $Z$ ) é apenas uma, e a função destes circuitos é fazer o valor da saída igual ao valor de uma das entradas, aquela cujo número seja igual à ordem da combinação de entrada.

A Fig. 2.11 exemplifica os detalhes de dois destes *multiplexers*, os de duas e de quatro entradas, mostrando as suas tabelas de verdade e os respectivos logogramas. As saídas  $z$  destes dois multiplexers podem ser descritas, respectivamente, por  $z = s x_0 + s x_1$  e  $z = s_1 s_0 x_0 + s_1 s_0 x_1 + s_1 s_0 x_2 + s_1 s_0 x_3$ .



**Fig. 2.10 - Multiplexers de 2, 4, 8 e 16 entradas, com 1, 2, 3 e 4 sinal de seleção, respectivamente**



**Fig. 2.11 - Logograma e tabela de verdade dos multiplexers da Fig. 2.10.**

A Fig. 2.33 na página 72 e a Fig. 2.44 na página 86 ilustram a utilização de um *multiplexer*. A Fig. 3.5 na página 122 revela a sua primeira utilização na implementação de um computador.

Os *multiplexers* constituem também uma forma expedita de implementar funções. Por exemplo, as funções  $z_1$  e  $z_2$  da Tabela 2.3, em vez de serem simplificadas pelos mapas de Karnaugh e implementadas de acordo com o circuito da Fig. 2.8, podem ser implementadas com dois *multiplexers* de 16 entradas, tal como indicado pela Fig. 2.12. Os valores destas entradas são obtidos directamente a partir da tabela de verdade da Tabela 2.3. As entradas do circuito descrito por esta tabela constituem as entradas de seleção dos *multiplexers*, que assim seleccionam um dos termos mínimos.

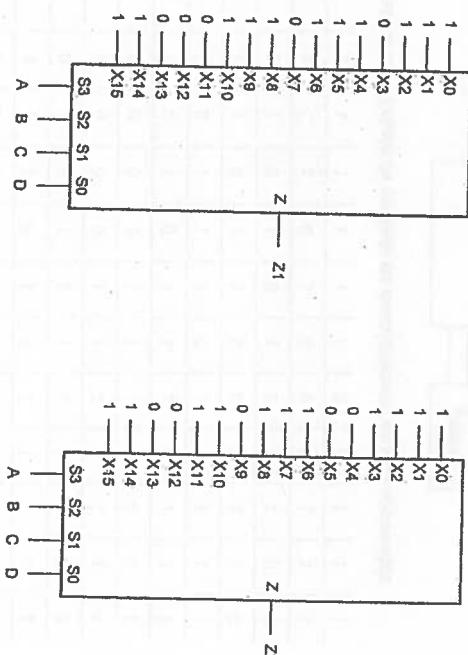


Fig. 2.12 - Utilização de *multiplexers* para implementar as funções  $z_1$  e  $z_2$  da Tabela 2.3

Esta implementação gasta mais portas lógicas do que a da Fig. 2.8, mas os *multiplexers* são blocos que já estão feitos, pelo que o esforço de desenvolvimento é menor.

### SIMULADOR - MULTIPLEXERS

Esta simulação ilustra o funcionamento dos *multiplexers*, usando as figuras desta secção como circuitos de base. Os aspectos cobertos incluem os seguintes:

- Verificação do funcionamento dos *multiplexers* de várias entradas;
- Verificação do funcionamento do circuito da Fig. 2.11;
- Utilização de um *multiplexer* para implementar uma função.

Um exemplo clássico de descodificador é o que produz os sinais necessários a um mostrador (*display*) de sete segmentos a partir de 4 bits. A Fig. 2.13 ilustra um destes mostradores, em formato de 8, com os segmentos identificados. Cada um destes segmentos é luminoso e acende ou apaga consoante o valor do bit ligado ao respetivo segmento, permitindo gerar todos os 10 dígitos e as letras A a F (o B e o D usam letras minúsculas, pois as maiúsculas confundem-se com 8 e 0, respectivamente).

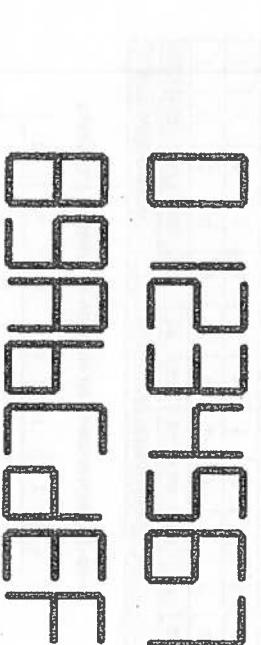


Fig. 2.13 - Mostrador (*display*) de sete segmentos e algumas das combinações possíveis

A Tabela 2.6 mostra a tabela de verdade que o descodificador tem de implementar, assumindo que cada segmento acende com o valor 1.

Com base nesta tabela, é possível simplificar as funções usando os mapas de Karnaugh, tal como foi feito na Fig. 2.7, e derivar o logograma do descodificador de sete segmentos. Esta tarefa é deixada como exercício para o leitor (Exercício 2.22).

A Fig. 2.14 mostra que se pode afixar um determinado símbolo no mostrador especificando apenas 4 bits ( $x_0 \dots x_3$ ) em vez de 7 bits. O descodificador faz a conversão. Note-se que 7 bits permitem  $128=2^7$  combinações possíveis, das quais é apenas possível, com este circuito, aproveitar 16. As restantes correspondem a outros símbolos, que poderão ou não

fazer sentido numa dada aplicação. Se for importante poupar bits, pode usar-se o circuito da Fig. 2.14. Se o importante for a flexibilidade de afixar o símbolo que se quiser (dentro das combinações de segmentos que o mostrador permite), então deve usarse o mostrador directamente, sem o descodificador.

SÍMBOLO	BITS DE ENTRADA				SEGMENTOS						
	X3	X2	X1	X0	A	B	C	D	E	F	G
0	0	0	0	0	1	1	1	1	1	0	
1	0	0	0	1	0	1	1	0	0	0	
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	
4	0	1	0	0	0	1	1	0	0	1	
5	0	1	0	1	0	1	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	0	0	0	0	0	
8	1	0	0	1	1	1	1	1	1	1	
9	1	0	0	1	1	1	1	0	1	1	
A	1	0	1	0	1	1	1	0	1	1	
b	1	0	1	1	0	0	1	1	1	1	
C	1	1	0	0	1	0	0	1	1	0	
d	1	1	0	1	0	1	1	1	0	1	
E	1	1	1	0	1	0	0	1	1	1	
F	1	1	1	1	1	0	0	0	1	1	

Tabela 2.6 - Tabela de verdade do descodificador de sete segmentos



Fig. 2.14 - Ligação do descodificador ao mostrador de sete segmentos

Outro exemplo típico dos descodificadores é a desmultiplicação de  $n$  bits de entrada em  $2^n$  bits de saída, dos quais apenas um está a 0 (aquele que corresponde à combinação dos bits de entrada) e todos os restantes a 1 (também poderia ser um bit a 1 e todos os outros a 0, mas a primeira hipótese é mais usada).

Estes descodificadores designam-se descodificadores 1-de- $P$  (porque dos  $P$  bits de saída apenas um está activo em cada instante). As tabelas de verdade dos descodificadores de

1-de-4 e 1-de-8 estão representadas na Tabela 2.7 e na Tabela 2.8, respectivamente. Estas são as dimensões mais usadas. A generalização para outras dimensões é imediata. Os descodificadores incluem uma entrada de controlo (E - Enable), que quando está a 0 desativa todas as saídas (ficam todas a 1), independentemente das variáveis de selecção.

BITS DE ENTRADA	BITS DE SAÍDA			
	Z0	Z1	Z2	Z3
0000	1	1	1	1
0001	0	1	1	1
0010	1	0	1	1
0011	1	1	0	1
0100	1	1	1	0
0101	0	0	1	1
0110	1	0	0	1
0111	1	1	0	0
1000	1	1	0	0
1001	0	1	0	0
1010	1	0	0	0
1011	1	1	0	0
1100	1	0	0	0
1101	0	0	1	0
1110	1	0	0	0
1111	0	0	0	1

Tabela 2.7 - Tabela de verdade do descodificador 1-de-4

BITS DE ENTRADA	BITS DE SAÍDA			
	Z0	Z1	Z2	Z3
0000	1	1	1	1
0001	0	1	1	1
0010	1	0	1	1
0011	1	1	0	1
0100	1	1	1	0
0101	0	0	1	1
0110	1	0	0	1
0111	1	1	0	0
1000	1	1	0	0
1001	0	1	0	0
1010	1	0	0	0
1011	1	1	0	0
1100	1	0	0	0
1101	0	0	1	0
1110	1	0	0	0
1111	0	0	0	1

Tabela 2.8 - Tabela de verdade do descodificador 1-de-8

A Fig. 2.15 mostra a representação esquemática destes descodificadores.

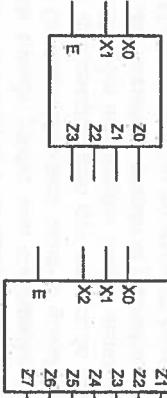


Fig. 2.15 - Exemplos de representação. (a) - Descodificador 1-de-4; (b) - Descodificador 1-de-8

A grande vantagem destes descodificadores é terem uma só saída activa por cada combinação dos valores das variáveis de entrada, o que permite depois tratar cada uma dessas combinações de forma independente das restantes. A Fig. 2.29 e a Fig. 6.3 (na página 418) e as seguintes mostram exemplos de utilização de descodificadores.

#### SIMULADOR – DESCODIFICADORES

Esta simulação ilustra o funcionamento dos descodificadores. Os aspectos cobertos incluem os seguintes:

- Verificação do funcionamento do descodificador de sete segmentos (Fig. 2.14);
- Verificação do funcionamento dos descodificadores 1-de-4 e 1-de-8 (Fig. 2.15).

#### 2.5.4 ROMS

Tal como um descodificador, uma ROM (*Read Only Memory*) é um circuito combinatório que, para cada uma das  $2^N$  combinações das N variáveis de entrada, produz uma combinação das P variáveis de saída, mas sem a restrição de  $N < P$  e em que a implementação dos valores de saída é feita de forma tabular e não por meio de portas lógicas.

Uma combinação das variáveis de saída designa-se palavra. Cada valor de endereço faz aparecer um valor de palavra na saída. Nada impede que dois endereços façam aparecer o mesmo valor de palavra, mas podem ser todos diferentes.

A Fig. 2.16 apresenta um exemplo de uma ROM que tem 16 palavras (4 bits de endereço, A<sub>3..A0</sub>) com 8 bits cada (D<sub>7..D0</sub>). Para os bits de endereço é costume usar a letra A (de Address, ou endereço) e para os bits da palavra costuma usar-se a letra D (de Data, ou dados).

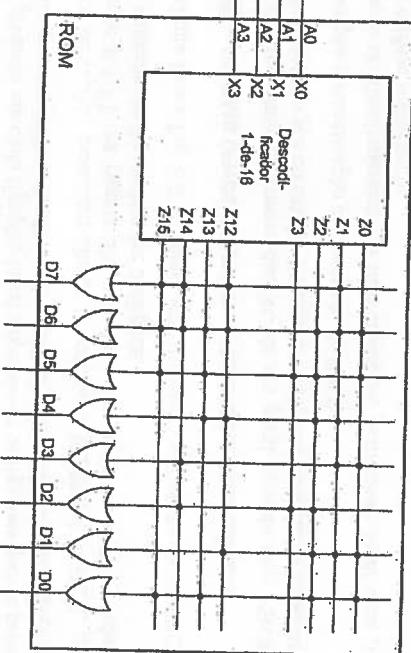


Fig. 2.16 - Estrutura interna de uma ROM (*Read Only Memory*). Cada OR tem tantas entradas quantas as saídas do descodificador

O descodificador selecciona apenas uma das suas 16 saídas (Z15..Z0). Neste caso, todas as saídas estão a 0 excepto uma, que fica a 1 (ao contrário dos descodificadores da secção anterior, mas tanto faz). Cada OR tem na realidade 16 entradas, embora por simplicidade na Fig. 2.16 apareça apenas uma. Para um dado endereço (correspondente a uma só linha horizontal a 1), cada bit de saída da ROM terá 1 apenas se existir uma ligação entre a linha vertical e essa linha horizontal. Caso contrário, será 0.

A Tabela 2.9 mostra a correspondência entre cada endereço e a palavra respectiva. Em vez de sintetizar esta tabela de verdade em portas lógicas, como fizemos em exemplos anteriores, as ligações entre linhas verticais e horizontais permitem transportar esta matriz directamente para o hardware. Este tem sempre a mesma estrutura, e o que varia são as ligações, de acordo com a tabela.

BITS DO ENDEREÇO				BITS DA PALAVRA							
X3	X2	X1	X0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	1	1	0	1	0	1	1
0	0	0	1	1	1	0	1	1	0	1	0
0	0	1	0	0	1	1	1	0	1	1	1
0	0	1	1	0	0	1	0	0	1	0	0
0	1	0	0	1	0	1	0	0	1	0	0
0	1	0	1	0	1	1	1	0	0	1	1
0	1	1	0	0	1	0	1	1	0	0	1
0	1	1	1	0	1	1	1	0	0	1	1
1	0	0	0	1	0	1	0	0	1	0	0
1	0	0	1	0	1	1	1	0	0	0	1
1	0	1	0	0	1	0	1	1	0	1	0
1	0	1	1	0	1	1	1	0	0	1	1
1	1	0	0	1	0	0	1	1	1	0	0
1	1	0	1	0	1	1	1	0	0	1	1
1	1	1	0	1	0	0	1	1	1	0	0
1	1	1	1	0	1	1	1	0	0	1	1

Tabela 2.9 - Tabela de verdade da ROM

Uma ROM é uma memória. Especifica-se um endereço e a ROM devolve a palavra que lhe corresponde. A sua capacidade (número de palavras que a memória pode armazenar) é  $2^N$ , em que N é o número de bits de endereço. Neste exemplo, N=4 e a capacidade da ROM é de 16 palavras.

É importante perceber que:

- O número N de bits do endereço determina a capacidade da memória, que é sempre  $2^N$  (por causa do número de saídas do descodificador);
- O número P de bits da palavra é totalmente independente de N, pois depende exclusivamente do número de ORs utilizados.

As ROMs propriamente ditas só se podem ler, sendo fabricadas já com as ligações adequadas. As PROMs (*Programmable Read Only Memories*) podem ser programadas, vindo de fábrica com todas as ligações efectuadas, o que significa que todos os bits de

todas as palavras estão a 1. Com um circuito adequado é depois possível fundir um pequeno fusível presente em cada ligação, interrompendo-se as ligações que se pretender. Os bits respectivos passam então a 0, não sendo possível voltar a fazer essas ligações.

Outros dois tipos de ROMs permitem alterar o conteúdo: EEPROM e Flash, que são descritas na secção 6.5.3.1, na página 534. Nenhum dos tipos de ROMs perde a sua informação se a alimentação do circuito for desligada.

### SIMULAÇÃO 2.5 – PROMS (PROGRAMMABLE READ ONLY MEMORIES)

Esta simulação ilustra o funcionamento das ROMs em geral, usando uma PROM que permite alterar facilmente o seu conteúdo. Os aspectos cobertos incluem os seguintes:

- Especificação e visualização do conteúdo da PROM;
- Verificação do funcionamento de uma memória, permitindo obter uma palavra após especificar o endereço;
- Implementação com uma PROM do descodificador de sete segmentos da Fig. 2.14, com a tabela de verdade indicada pela Tabela 2.6.

#### ESSENCIAL

Um circuito combinatório é aquele em que nenhuma entrada de uma porta lógica dependa directa ou indirectamente da sua saída. Assim, as entradas do circuito dependem apenas das suas entradas, sem realimentações.

Na sinse de, circuitos que se baseiam na tabela de verdade valores das saídas em função dos valores das entradas) simplifica-se com as tabelas de Karnaugh determinando os termos da função e depois implementa-se a função usando portas lógicas ou outros dispositivos, como por exemplo uma ROM. Também é possível usar circuitos já feitos, como os multiplexers e os descodificadores, como se fossem portas lógicas mais elaboradas.

Os multiplexers permitem escolher uma das entradas para reproduzir na sua saída. Sob controlo de  $k$  sinais de seleção, em que  $N=2^k$ . Os multiplexers mais frequentes têm 2, 4, 8 e 16 entradas, com 1, 2, 3 e 4 sinais de seleção.

O decodificadores têm nsáidas todas a 1 e cepo uma a jnd cada rom, mas de seleção, em que  $N=2^k$ . Os valores de  $N$  mais comuns são 4, 8 e 16.

As ROMs (Read Only Memories) são memórias só de leitura, constituídas por  $N$  células com  $R=1$ , cada podendo especificar a sua saída com o valor de cada bit. Tem de ser especificado  $N$ , em que  $N=2^k$ , para indicar qual célula se querer. As ROMs já vêm programadas de fábrica. As PROMs (Programmable ROMs) permitem programação (unívoca) pelo utilizador.

## 2.6 CIRCUITOS SEQUENCIAIS

Ao contrário dos circuitos combinatórios, em que as saídas dependem apenas dos valores das entradas, nos circuitos sequenciais o valor das saídas pode depender não apenas das entradas mas também dos valores anteriores das saídas.

Uma combinação dos valores das saídas do circuito designa-se estado. Quando as entradas do circuito mudam o seu valor, os valores das saídas podem mudar em consequência, dizendo-se nessa altura que houve uma transição de estado.

A designação "sequenciais" deriva do facto de o estado para o qual um circuito transita poder depender não apenas da variação ocorrida nas entradas como também do estado anterior e até mesmo da sequência de estados (ou seja, do historial dos estados) pela qual o circuito transitou.

### 2.6.1 ELEMENTOS BI-ESTÁVEIS

Estes são elementos com apenas uma saída (um bit, dois estados possíveis) e com capacidade de memória, isto é, manter a saída mesmo que a entrada mude.

#### 2.6.1.1 TRINCO SR

Um trinco (*latch*) de uma porta tem a propriedade de permitir à porta fechar (encolhendo a lingueta) quando se empurra mas depois prender e já não deixar abrir quando se tenta fazer o movimento inverso (puxar). É preciso actuar nouro local (bolão ou chave da fechadura) para permitir abrir a porta novamente. O circuito da Fig. 2.17 ilustra este princípio aplicado aos circuitos digitais. Note-se a dupla realimentação que faz com que as saídas dependam não exclusivamente das entradas S (Set, pôr a 1) e R (Reset, repor a 0) mas também do valor das próprias saídas Q e  $\bar{Q}$ .

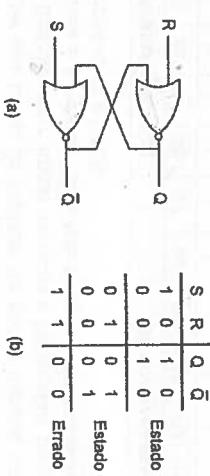


Fig. 2.17 - Trinco SR com NORs. (a) - Circuito; (b) - Tabela de estados

O funcionamento pode ser analisado seguindo os sinais e aplicando as funcionalidades das portas lógicas. Por exemplo, na situação indicada pela primeira linha da tabela da Fig. 2.17, a entrada S=1 força a saída Q a 0, o que com R=0 força a saída  $\bar{Q}$  a 1. Esta é uma situação estável, que se mantém enquanto as entradas não mudarem. Mesmo que S passe para 0 (segunda linha da Fig. 2.17), as saídas mantêm o estado pois a saída Q a 1 força o estado do NOR de baixo.

No entanto, se agora a entrada R mudar para 1, as saídas mudam, trocando o seu valor (basta seguir os sinais). Se R voltar a 0, as saídas voltam a não sofrer alteração (mas agora o estado está trocado em relação ao anterior).

As entradas S e R podem estar numa de três situações:

- Diferentes. Neste caso, a saída Q está a 1 ou 0, consoante a entrada que esteja a 1 (S ou R, respectivamente);
- Iguais com o valor 0. O estado anterior de Q mantém-se, seja ele 0 ou 1, numa semântica de memorização;
- Iguais com o valor 1. Esta é uma situação que não deve ser usada, pois ambas as entradas a 1 forçam ambas as saídas a 0, quebrando a semântica de negação entre as saídas.

Também é possível um comportamento semelhante com portas lógicas NAND, tal como ilustrado pela Fig. 2.18, em que se devem trocar os 1s pelos 0s e vice-versa.

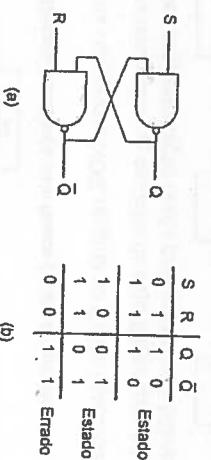


Fig. 2.18 - Trinco SR com NANDs. (a) – Circuito; (b) – Tabela de estados

### SIMULAÇÃO 2.5 – TRINCO SR

Esta simulação permite verificar o funcionamento dos trincos SR, tornando a Fig. 2.17 e a Fig. 2.18 como base. Os aspectos cobertos incluem os seguintes:

- Sequenciamento dos estados do trinco SR com NORs;
- Verificação dos estados do trinco SR com NANDs;
- Verificação do estado com S e R activos.

### 2.6.1.2 TRINCO D

O trinco SR permite memorizar 0 ou 1 na saída Q, mas precisa de dois sinais separados para poder mudar de estado e não tem controlo sobre mudar ou não a saída quando as entradas mudam. O trinco D permite resolver estes dois problemas, usando o circuito da Fig. 2.19a.

A tabela de estados mostra que quando C=1 a saída Q fica igual à entrada D. Se esta variar, a saída varia também (e a saída  $\overline{Q}$  é a negação). Nesta situação, diz-se que o trinco está transparente. Quando C=0, o trinco não altera o seu estado de saída, nem mesmo que D varie, pelo que memorizou o último valor da entrada D na altura em que C passou para 0.

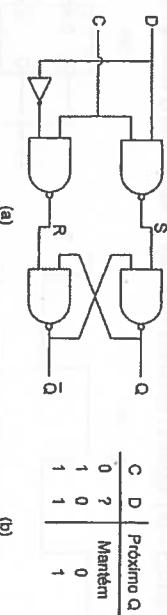


Fig. 2.19 - Trinco D. (a) – Circuito; (b) – Tabela de estados

**SIMULAÇÃO 2.7 – TRINCOS D**  
Esta simulação ilustra o funcionamento dos trincos D, usando a Fig. 2.19 como base. Os aspectos cobertos incluem os seguintes:

- Sequenciamento das variáveis de entrada e verificação da tabela de estados;
- Verificação dos efeitos de transparência e de memorização.

### 2.6.1.3 BÁSCULA D

Um dos defeitos do funcionamento dos trincos é o facto de durante a fase de transparência serem, na prática, simples circuitos combinatórios, em que a saída depende apenas das entradas e o histórico dos estados deixa de contar.

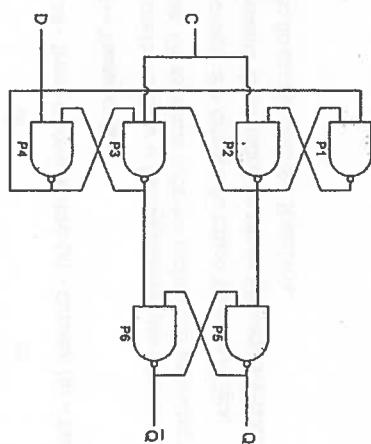
Uma báscula (*flip-flop*) é um circuito, constituído por dois trincos<sup>12</sup> em que, quando um está transparente, outro está a memorizar, de forma a que nunca se verifique uma situação em que a saída dependa exclusivamente da entrada, sendo os instantes possíveis de transição da saída comandadas por um sinal de controlo.

Há vários tipos de báscula, mas a mais usada é a báscula D activada no flanco (*edge-triggered*), o que significa que a saída só pode mudar quando o sinal C transita de valor 0 para 1, por exemplo. Tal como no trinco D, a saída Q memoriza o valor que a entrada D tem, mas a diferença é que a báscula tira uma espécie de fotografia à entrada D na altura da transição de C, e mantém esse valor mesmo que depois a entrada D varie, ao passo que o trinco acompanha as variações da entrada D enquanto C estiver activo (e não apenas na transição). Uma báscula ou reage no flanco ascendente de C (transição de 0 → 1) ou descendente (transição de 1 → 0), mas não ambos (é o desenho interno do circuito que o determina).

A Fig. 2.20 mostra uma báscula D activada no flanco ascendente, quando C transita de 0 para 1. Quando C=0, tanto P2 como P3 têm o valor 1, o que faz o trinco SR P5-P6 manter o estado (Fig. 2.18). P4 e P1 reagem às variações de D, mas C=0 impede essas variações de afetarem o trinco P5-P6. Quando C transita de 0 → 1, D aparece à entrada de P5 e D à entrada de P6, o que faz com que Q fique com o valor de D e  $\overline{Q}$  com o valor de  $\overline{D}$ . Ou seja,

<sup>12</sup> Para implementar algumas propriedades, como no caso da Fig. 2.20, as básculas podem ter mais de dois trincos.

a báscula memoriza o valor que D tinha imediatamente antes da transição de C de 0 → 1. Logo depois, mesmo que D varie, já não afeta a saída da báscula. A entrada de P5 ou P6 que estiver a 0 força o trinco SR para trás (P1-P2 ou P3-P4) a esse estado, independentemente de D, e a outra entrada de P5 ou P6 pode variar entre 0 e 1 sem alterar o estado do trinco P5-P6 (Fig. 2.18b). Se C voltar a 0 continua a não afectar a saída da báscula, pelo que o seu estado se pode alterar apenas na altura em que o flanco ascendente (0 → 1) de C ocorre.

Fig. 2.20 - Báscola D activada pelo flanco (*edge-triggered*) ascendente

Se esta báscula fosse feita com portas NOR reagiria no flanco descendente do sinal C.

Existe uma notação para representar as entradas de cujo flanco uma báscula depende para reagir. A Fig. 2.21 mostra a representação usada para os trincos e básculas D.

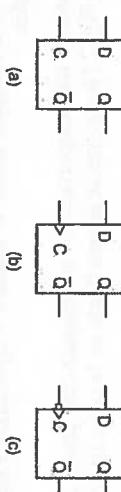


Fig. 2.21 - Representação dos trincos e básculas D. (a) – Trinco; (b) – Báscola D activada no flanco ascendente de C; (c) – Báscola D activada no flanco descendente de C

A. Fig. 2.22 caracteriza o comportamento dos trincos e das básculas, exprimindo a semântica de controlo de flanco no sinal C, quer no flanco ascendente, quer no flanco descendente.

Para que a báscula D funcione bem, a entrada D tem de manter o seu valor fixo desde um pequeno tempo antes da transição de C (designado tempo de preparação, ou *setup time*) até um pequeno tempo após essa transição (designado tempo de manutenção, ou *hold time*). Estes tempos são geralmente muito curtos face ao tempo de atraso da báscula (tempo de propagação da transição da entrada para a saída, ou *delay time*) e ao tempo

entre transições do sinal C, pelo que em termos lógicos, de funcionalidade, são ignorados (são importantes quando se pretende saber a rapidez máxima de variação dos sinais).

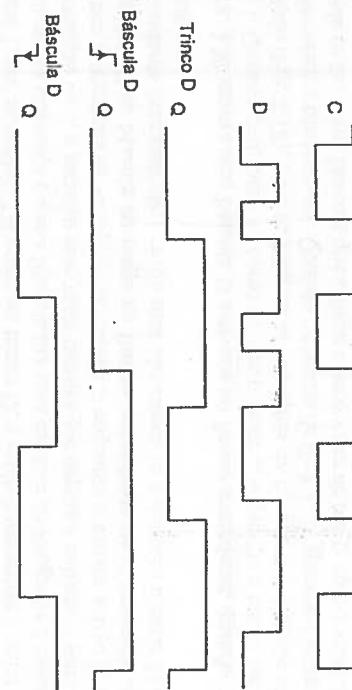
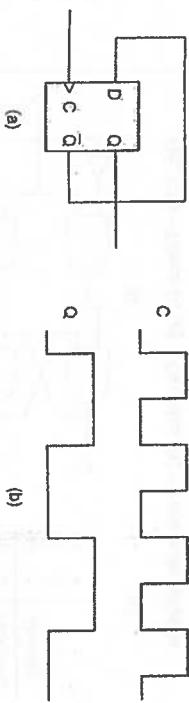


Fig. 2.22 - Exemplo de comportamento dos trincos e básculas D

Esta transição controlada de estado é fundamental para construir circuitos síncronos, em que saídas de básculas constituem entradas de outras. O circuito síncrono mais simples é provavelmente aquele em que a saída Q liga à entrada D da própria báscula, tal como descrito pela Fig. 2.23a, em que a saída alterna (*toggle*) de valor em cada transição ascendente do sinal C. Se este sinal variar periodicamente, como na Fig. 2.23b, o sinal de saída varia com metade da frequência (ou o dobro do período), uma vez que para um ciclo completo na saída são precisas duas transições ascendentes no sinal C.

Se este circuito usasse um trinco D, em vez de uma báscula D, o funcionamento seria instável, pois enquanto o sinal C estivesse a 1 o trinco estaria transparente, provavelmente oscilando entre 0 e 1, uma vez que a negação da entrada seria realimentada para essa mesma entrada, sem atingir um ponto estável. A báscula D também faz esta realimentação, mas quando a saída muda, após o tempo de atraso do circuito, já a báscula deixou de estar sensível a variações no sinal de entrada, o que permite atingir imediatamente um ponto estável. Só no próximo flanco ascendente do sinal C é que a saída poderá mudar novamente. Assim, a oscilação também se dá, mas ao ritmo controlado do sinal C e não ao do tempo de atraso do circuito.

Fig. 2.23 - Báscola D em circuito de transição alternada (*toggle*). (a) – Circuito; (b) – Diagrama temporal

Este sinal periódico é extremamente importante em circuitos digitais, pois permite condicionar o ritmo a que as operações evoluem, e designa-se relógio (*clock*), uma vez que marca os instantes de tempo em que as coisas acontecem.

Também é usual as básculas terem duas entradas adicionais para poder forçar um determinado estado, 0 ou 1, em qualquer altura e independentemente (de forma assíncrona) das transições do relógio, o que pode ser importante em termos de inicialização de um circuito. A Fig. 2.24 mostra uma possível implementação, usando o circuito da Fig. 2.20 como base. Essencialmente, a técnica consiste em acrescentar entradas em cada trinco SR que permitam forçar um determinado estado. Os sinais PR (*Present*), que fazem Q=1, e CL (*Clear*), que fazem Q=0, são activos a 0. Só um destes sinais pode ser activado de cada vez. Não basta forçar o trinco SR de saída. Os anteriores têm também de serem inicializados, para garantir que depois de desactivar PR ou CL o estado destes é compatível com o estado em que o trinco de saída foi colocado.

### SIMULAÇÃO 2.18 – BÁSCULAS D

Esta simulação ilustra o funcionamento das básculas D, não apenas de forma intrínseca, mas também contrastando-o com o dos trincos, usando as figuras anteriores como base. Os aspectos cobertos incluem os seguintes:

- Verificação do comportamento da báscula, usando as portas lógicas internas (Fig. 2.20);
- Comparação do comportamento de uma báscula activada num flanco ou noutra, com o de um trinco (Fig. 2.22);
- Verificação do comportamento do divisor de frequência da Fig. 2.23;
- Inicialização a 0 ou 1 da báscula da Fig. 2.24.

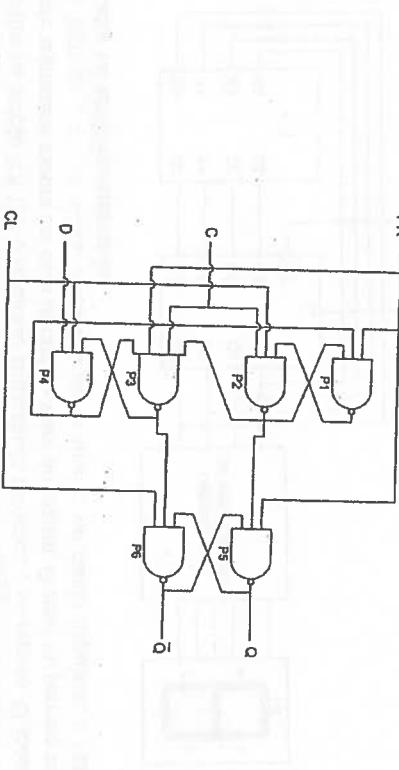


Fig. 2.24 - Báscola D com entradas para forçar estados de 0 (CL - Clear) e 1 (PR - Present)

Um registo é um conjunto de elementos de memória, sejam trincos ou básculas, permitindo armazenar vários bits simultaneamente. Em termos de linguagem corrente, é comum usar os termos "trinco (latch)" (de  $N$  bits) para designar um registo de  $N$  trincos D e "registo com básculas" (de  $N$  bits), ou simplesmente "registo" quando não houver confusão, para designar um registo de  $N$  básculas D.

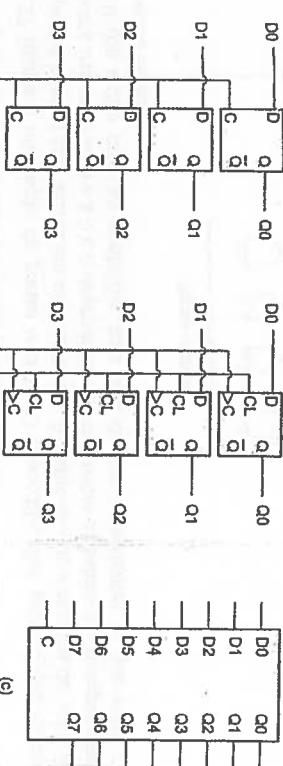
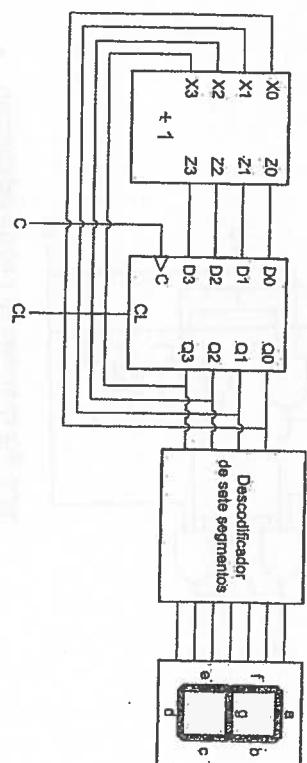


Fig. 2.25 - Registros: (a) – Trinco (latch) de 3 bits; (b) – Registro com básculas de 3 bits, activado pelo flanco ascendente do relógio e com sinal de inicialização a 0; (c) – Representação do trinco de 8 bits; (d) – Representação do registo de 8 bits

A Fig. 2.25 ilustra a estrutura destes registos, em que todos os trincos ou básculas são activados pelo mesmo sinal C. No caso do trinco de 8 bits (Fig. 2.25a), cada saída Q<sub>i</sub> acompanha a entrada respectiva D<sub>i</sub>, enquanto o sinal C estiver a 1. Quando este sinal passa para 0, as saídas memorizam os valores que tinham nessa altura. No caso do registo com básculas de 8 bits (Fig. 2.25b), as saídas só mudam quando o sinal C passa de 0 para 1, memorizando os valores que as respectivas entradas tinham nessa altura. O comportamento é o mesmo que o trinco e a báscula individuais representado na Fig. 2.22, só que simultâneo para vários trincos e básculas. Os registos do tipo trinco usam-se quando se pretende apenas memorizar um valor. Os registos com básculas usam-se quando se pretende efectuar operações em que a saída do registo sofra algum processamento e o

valor resultante seja de novo memorizado no mesmo registo, tal como ilustrado pela Fig. 2.26, em que o valor do registo é incrementado de uma unidade (os somadores são descritos na secção 2.8.1) e o resultado armazenado novamente no registo. O mostrador de sete segmentos mostra em cada instante o valor do registo. O sinal  $C1$  permite colocá-lo a zero em qualquer altura. À medida que o sinal  $C$  vai dando impulsos, o valor do contador vai sendo incrementado.



... com base nas quais as entradas do registo dependem

Este circuito funciona bem porque o registo usa básculas activadas no flanco, que memorizam as entradas como se de uma fotografia se tratasse, em que se as entradas variarem logo a seguir já não afectam as saídas (apenas na próxima transição  $0 \rightarrow 1$  do sinal C). Se em vez de um registo com 4 básculas se usasse um triunco de 4 bits, sempre que é enquanto o sinal C estivesse a 1, o valor resultante da soma seria introduzido de novo no triunco, indo de novo afectar as saídas mal a alteração se propagasse pelo triunco e tornando o sistema instável.

Os registos com básculas são também usados nas máquinas de estado (secção 2.6.7), em que o estado da máquina, armazenado no registo, tem influência no novo estado para o qual se deve transitar na próxima transição do relógio, novo estado esse que será também armazenado no registo.

SINUÁCÃO 2.9 — REGISTOS

Os aspectos cobertos incluem os seguintes:

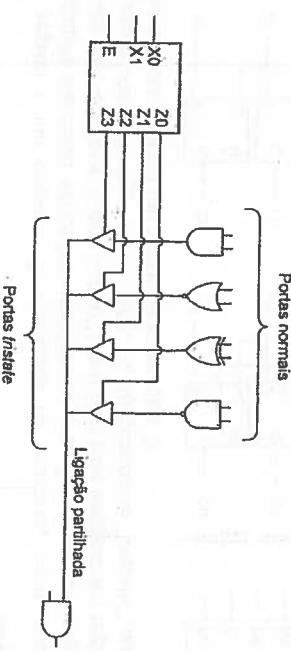
- Verificação do comportamento do circuito da Fig. 2.26, incluindo o efeito do sinal CL;  
Constatação do funcionamento incorrecto caso o registo da Fig. 2.26 seja substituído por um registo com trineos.

## 2.6.3 PORTAS LÓGICAS DE TRÊS ESTADOS (*TRISTATE*)

## 26.3 PORTAS LÓGICAS DE TRÊS ESTADOS (TRISTATE)

Isto implica que cada saída destas tenha um sinal de controlo, que permite ligar ou desligar a saída. Tipicamente, usa-se uma porta lógica específica, entre uma saída normal só com dois estados, 0 ou 1) e a ligação partilhada entre várias saídas, e cuja única função é acoplar a saída normal à ligação (deixando passar 0 ou 1) ou simplesmente desacoplar-a da ligação. Esta porta de três estados (porta tristate) é um interruptor eletrónico, na prática, sob comando de um sinal de controlo.

A Fig. 2.27 ilustra a utilização de portas *tristate*. O descodificador garante que apenas uma das portas *tristate* está activa em cada instante. Assume-se que a entrada de controlo de cada porta *tristate* é activa a 0 e o descodificador tem todas as saídas a 1 excepto uma, que terá 0 (ou então tudo ao contrário, mas tem de ser consistente), o que evita os conflitos entre saídas.



• 2.2.7 - Utilização de saídas com capacidade de três estados (*tristate*)

O conjunto descodificador com as portas *tristate* é equivalente, em termos práticos, a um *multiplexer*. Apenas um dos sinais (o indicado pelos sinais  $x_1$  e  $x_0$ ) vindos das portas lógicas é colocado na ligação partilhada.

Pode pensar-se o que sucederia se nenhuma porta *tristate* estivesse activa. O descodificador tem uma entrada de controlo  $E$  que se estiver a 0 permite desactivar todas as saídas (que ficam todas a 1 – Tabela 2.7), caso em que todas as portas *tristate* ficam desligadas e nenhuma porta força um valor na ligação partilhada. Neste caso, diz-se que esta ligação está “no ar” ou em estado de alta impedância.

O problema desta situação é que a porta AND cuja entrada depende da ligação partilhada fica num estado indefinido e o comportamento do circuito pode ficar aleatório, pelo que a utilização de portas *tristate* se tem de rodear de alguns cuidados. Se a uma ligação a que ligam portas *tristate* ligarem também entradas de portas lógicas, tem de se garantir que há sempre uma porta *tristate* ligada, para garantir um valor válido na ligação, excepto quando o valor dessa entrada for irrelevante para o circuito a que ela liga (por exemplo, se o sinal presente na outra entrada do AND de saída na Fig. 2.27 for 0).

### SIMULACRADOZING – LÓGICA DE TRÊS ESTADOS

Esta simulação demonstra o funcionamento das portas *tristate*, tornando a Fig. 2.27 como base. Os aspectos cobertos incluem os seguintes:

- Leitura de uma porta *tristate* em estado de alta impedância (valor aleatório);
- Conflitos numa ligação em caso de activação simultânea de duas portas *tristate*.

### 2.6.4 BANCO DE REGISTOS

Num exemplo tão simples como o anterior o mais fácil é usar mesmo um *multiplexer*. As portas *tristate* são mais interessantes em situações mais complexas, como por exemplo um banco de registos, em que um conjunto de registos partilha a mesma entrada e saída (sendo possível seleccionar qual o registo que se quer escrever ou ler).

Para que tal seja possível, cada registo tem de ter saídas *tristate*, tal como representado na Fig. 2.28. O sinal de controlo  $OE$  (*Output Enable*, ou activador da saída) é activo a 0 para ser compatível com as saídas dos descodificadores (Tabela 2.8). Se  $OE=0$ , as portas *tristate* desse registo estão ligadas e as saídas Q7..Q0 são transpostas para as saídas Z7..Z0. Se  $OE=1$ , as portas *tristate* estão desligadas e, embora as saídas do registo Q7..Q0 tenham um dado valor, esse valor não é imposto às saídas Z7..Z0.

Ligar as saídas de vários registos com saída *tristate* entre si é fácil, bastando ligar os sinais de controlo  $OE$  dos vários registos a um descodificador, para garantir que só um deles está activo de cada vez, tal como representado na Fig. 2.29, que ilustra um banco de quatro registos. Os bits  $x_1$  e  $x_0$  escolhem com qual dos registos se está a trabalhar. De lado esquerdo, as negações compatibilizam os flancos dos relégos. Quando  $C=1$  todos os sinais C dos registos estão a 0 e que quando  $C=1$  só um registo terá o seu sinal C a 1.

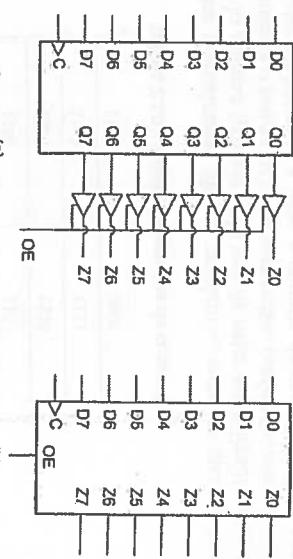


Fig. 2.28 - Registo de 8 bits com saídas *tristate*. (a) – Versão mais detalhada; (b) – Representação do conjunto a usar em diagramas mais complexos

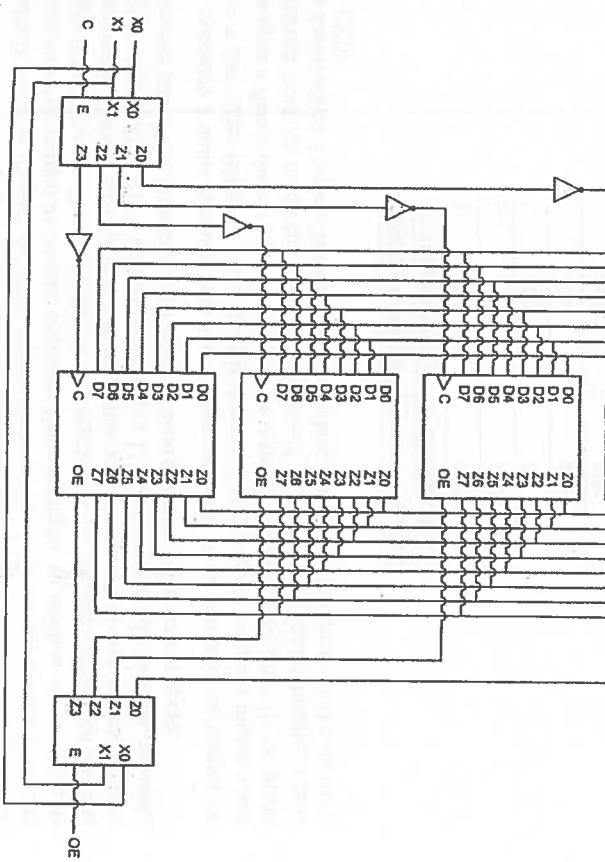


Fig. 2.29 - Banco de registos

Qualquer computador actual tem um banco de registos, podendo variar entre menos de 10 (8, por exemplo) até largas dezenas ou mesmo centenas, dependendo da complexidade do processador. A secção 4.6 na página 203 e a secção 7.2.1.2 na página 572 apresentam informação adicional sobre este tópico.

## 2.6.5 CONTADORES

Uma das operações mais básicas dos circuitos digitais é contar (impulsos de relógio). Veja-se por exemplo a Tabela 2.6, em que as quatro variáveis de entrada ( $x_3..x_0$ ) vão sucessivamente variando entre 0000 e 1111, passando por todas as combinações intermédias, tal como já tinha sido indicado pela Tabela 1.1 na página 10.

Note-se que  $x_0$  é a variável que muda mais frequentemente, tal como o dígito da direita num número decimal (o dígito das unidades) varia mais frequentemente quando se conta entre 0000 e 9999. Aqui é a mesma coisa, com a diferença de que os números são binários e só há dois símbolos, 0 e 1.

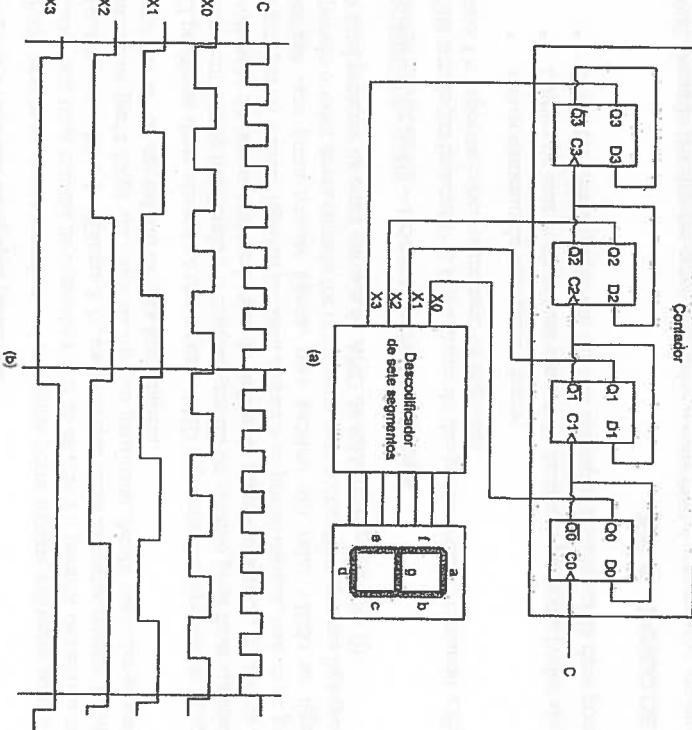


Fig. 2.30 - Contador com báscula D em sequência (ripple), ligado a um descodificador de sete segmentos. (a) - Circuito; (b) - Diagrama temporal, ilustrando os sucessivos atrasos

Uma forma de conseguir implementar um contador com 4 bits é usar 4 básculas D em sequência, ou onda (*ripple*), tal como representado na Fig. 2.30, em que a saída Q de uma báscula serve de relógio da báscula seguinte. Neste exemplo, ao efectuar vários ciclos de relógio (sinal C, de *clock*) o mostrador de sete segmentos vai passando pelas suas diversas combinações (Tabela 2.6), entre 0 e F, enquanto as variáveis  $x_3..x_0$  vão sucessivamente variando entre 0000 e 1111. Quando chegarem a 1111, as variáveis dão a volta, passando novamente para 0000, tal como se pode ver no lado direito do diagrama temporal.

Este contador é muito simples, mas tem o inconveniente de ir somando os tempos de atraso  $T$  das várias básculas. Uma báscula demora sempre algum tempo a mudar a sua saída após o flanco do seu sinal de relógio. Se o relógio mudar no instante  $TC$ , Q0 muda no instante  $TC+T$ , Q1 muda em  $TC+2T$ , Q2 em  $TC+3T$  e Q3 em  $TC+4T$  (naturalmente, o caso mais desfavorável é aquele em que os bits mudam todos, de 1111 para 0000 e de 0111 para 1000).

Tabela 2.10 - Tabela de estados do contador

ESTADO CORRENTE	ESTADO SEGUINTE
$x_3\bar{x}_2\bar{x}_1\bar{x}_0$	$\bar{x}_3\bar{x}_2\bar{x}_1\bar{x}_0$
0000	0001
0001	0010
0010	0011
0011	0100
0100	0101
0101	0110
0110	0111
0111	1000
1000	1001
1001	1010
1010	1011
1011	1100
1100	1101
1101	1110
1110	1111
1111	0000

Tabela 2.10 - Tabela de estados do contador

Estes atrasos ocasionam normalmente alguns problemas, entre os quais a limitação da frequência do relógio do sistema (porque tudo tem de estar estabilizado antes de o sinal de relógio mudar novamente o seu valor). Por esta razão, prefere-se normalmente aplicar o sinal de relógio a todos as básculas e ligar a entrada de cada báscula a um pequeno circuito combinatório que diz a essa báscula se deve mudar o seu estado ou não. Tal pode

ser feito com recurso a tabelas de estados (indicando para cada estado do contador qual novo estado, após novo impulso de relógio) e simplificando as funções usando mapas de Karnaugh, tal como indicado na Tabela 2.10 e na Fig. 2.31.

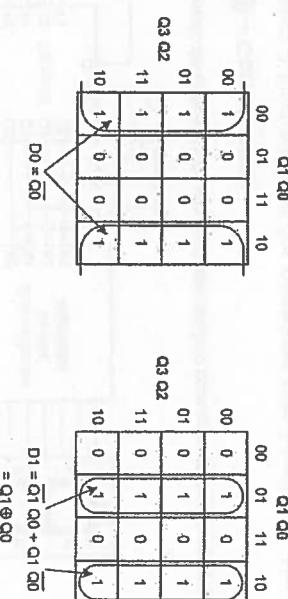


Fig. 2.31 - Mapas de Karnaugh para simplificar as expressões das entradas das básculas do contador

A utilização de OU-exclusivos permite simplificar o aspecto das expressões. O circuito correspondente está representado na Fig. 2.32. No fundo, é um registo com basculas, em que cada entrada é uma função diferente das saídas.

Uma possibilidade muito usada em contadores é o carregamento em paralelo (em todas as básculas) de um dado valor, a partir do qual a contagem se desenvolve nos ciclos de relógio seguintes. Para este efeito, um contador tem de ter como entradas adicionais os bits para ligar as várias básculas e uma entrada de controlo (PI - Parallel Load) que indica se num dado ciclo de relógio o contador conta (passando ao número binário seguinte) ou carrega nas básculas os valores dos bits do carregamento paralelo (independentemente do valor actual de contagem). A Fig. 2.33 mostra como este contador pode ser implementado. Essencialmente, insere-se um multiplexor na entrada D de cada báscula, permitindo seleccionar os bits do valor de contagem seguinte ou os de carregamento paralelo. O multiplexor de duas entradas é o da Fig. 2.11a.

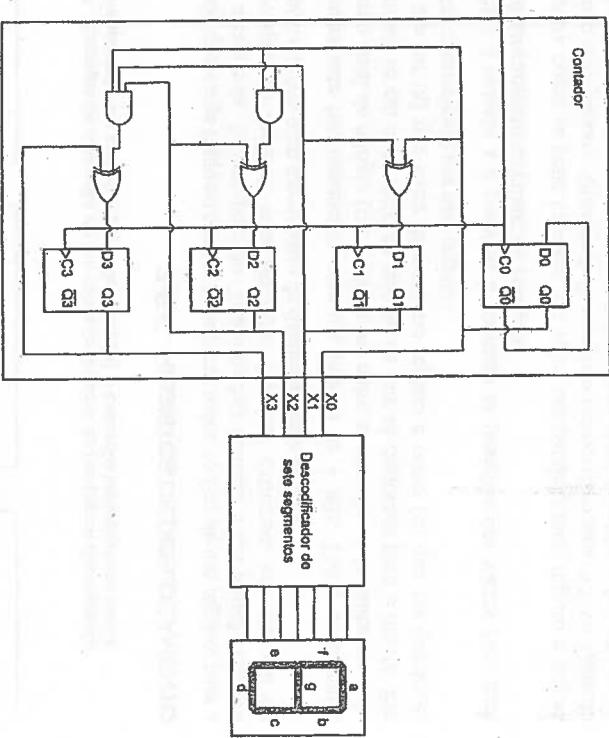


Fig. 2.32 - Contador binário, ligado a um descodificador de sete segmentos

A Fig. 2.34 ilustra o funcionamento deste contador binário com carregamento em paralelo. Quando o contador chega a 9 (1001 em binário), a saída do AND fica a 1 e activa o carregamento paralelo (PL). No próximo flanco ascendente do sinal de relógio (C), o contador carrega o valor 0000, reconhecendo a contagem normal no ciclo de relógio seguinte. O efeito é o mostrador de sete segmentos contar de 0 a 9 de forma cíclica, em vez de circular pelos 16 valores possíveis (0 a F).

Também é vulgar outra variante, a contagem decrescente (o contador contar "para trás"). Os circuitos combinatórios que ligam às entradas D das básculas são diferentes, bastando refazer a Tabela 2.10 e Fig. 2.31 (o que é deixado para o Exercício 2.20). Combinado com o carregamento paralelo, permite carregar o contador com um valor e contar esse número de impulsos até o contador chegar a 0, altura em que um simples NOR permite recarregar o valor no. contado. Isto permite gerar uma forma de onda de período programável, com um impulso de N+1 em N+1 impulsos de relógio, em que N é o valor carregado no contador.

A Fig. 2.35 ilustra este funcionamento. Quando o contador chega a 0000, o NOR fica com a saída a 1 e activa o carregamento paralelo (PL), o que faz com que na próxima transição do sinal de relógio (C) seja carregado o valor 0110 (6, em decimal) em vez de o contador dar a volta e passar para 1111 (este contador conta para trás). Nos ciclos de relógio seguintes o contador (e também o mostrador de sete segmentos) irá passar pelos

valores 5, 4, 3, 2, 1 até voltar a 0 novamente e o ciclo se repetir. Variando o valor na entrada de carregamento paralelo do contador pode variar-se facilmente o período (tempo ao fim do qual o sinal se repete) do sinal de saída Z.

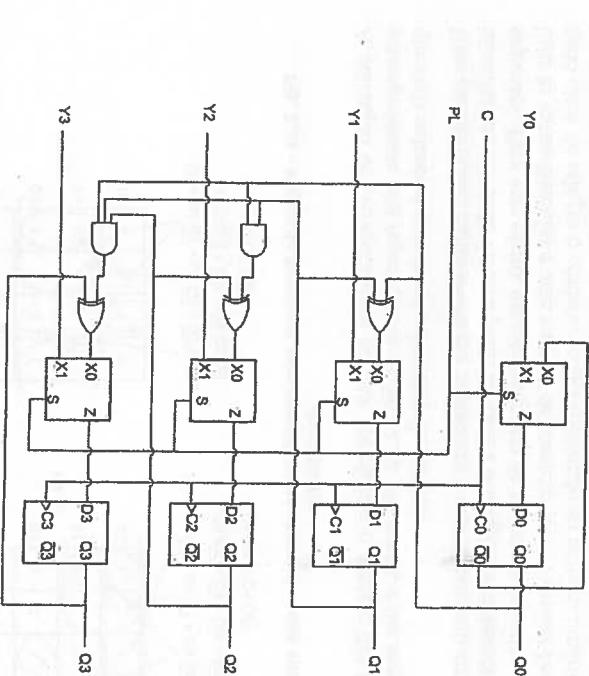


Fig. 2.33 - Contador binário com possibilidade de carregamento de um valor em paralelo

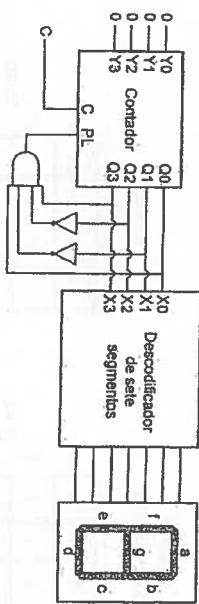


Fig. 2.34 - Exemplo de aplicação de um contador binário com carregamento em paralelo

### SIMULAÇÃO 2.1 — CONTADORES

Esta simulação ilustra o funcionamento dos contadores com básculas D, com base nas figuras e exemplos desta secção. Os aspectos cobertos incluem os seguintes:

- Verificação do comportamento do contador em sequência (*ripple*) e da soma dos tempos de atraso (Fig. 2.30);

Verificação do comportamento do contador da Fig. 2.32 e do seu tempo de atraso face ao relógio;

Contagem crescente com carregamento paralelo (Fig. 2.34);

Contagem decrescente com carregamento paralelo (Fig. 2.35) e programabilidade do período de contagem.

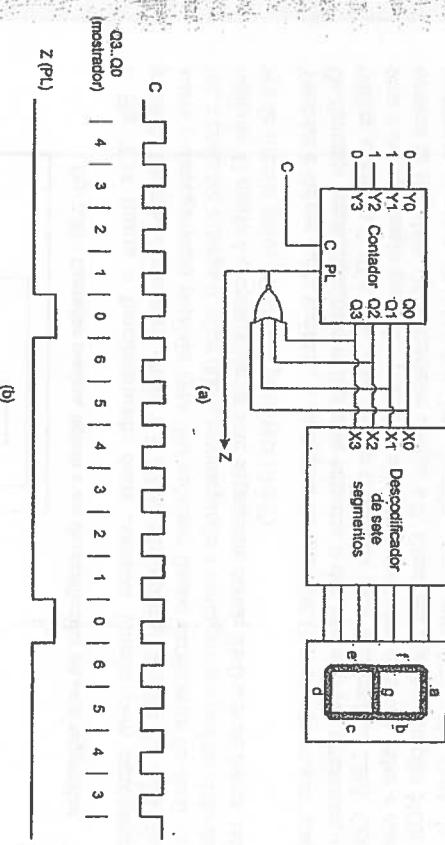


Fig. 2.35 - Exemplo de aplicação de um contador binário de contagem decrescente com carregamento em paralelo. (a) – Circuito; (b) – Evolução temporal dos sinais

## 2.6.6 REGISTOS DE DESLOCAMENTO

Uma necessidade que surge frequentemente é deslocar todos os bits de um registo, para a direita ou para a esquerda, de uma posição. Todos os bits mantêm a sua posição relativa (mas noutra posição do registo), com exceção dos bits extremos, em que um bit desaparece e outro é novo, num movimento de deslocamento linear.

A Fig. 2.36 ilustra este funcionamento com um registo de 4 bits. Todas as básculas recebem o mesmo sinal de relógio (C). As ligações entre as entradas e as saídas de cada báscula determinam se em cada ciclo de relógio os bits se deslocam para a direita. Em cada ciclo, há um bit (X) que entra de novo no registo e outro (Z) que sai (perde-se, deixando de estar memorizado por este registo).

Ligando a saída Z à entrada X é possível ir rodando as posições dos vários bits, num movimento de deslocamento circular, ou rotação.

A Fig. 2.37 mostra como se pode carregar um valor em paralelo num registo e depois deslocá-lo, usando *multiplexers*. Quando D=0, o registo memoriza o sinal Y3-Y0. Quando D=1, o registo faz um deslocamento para a direita em cada ciclo de relógio. Se se ligar Z a X, faz uma rotação.

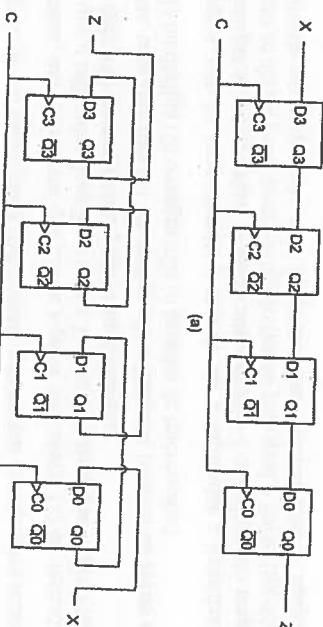


Fig. 2.36 - Registros de deslocamento. (a) - Para a direita; (b) - Para a esquerda

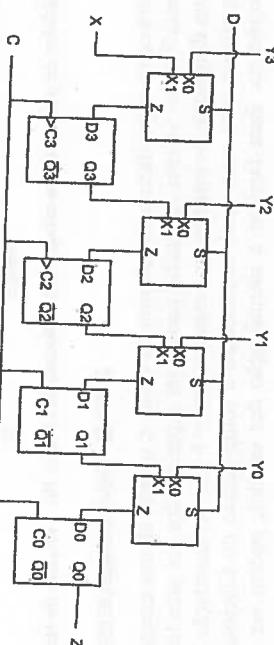


Fig. 2.37 - Registro de deslocamento para a direita com carregamento em paralelo

### SIMULAÇÃO - REGISTOS DE DESLOCAMENTO

Esta simulação ilustra o funcionamento dos registos de deslocamento. Os aspectos cobertos incluem os seguintes:

- Deslocamento linear do registo de deslocamento (Fig. 2.36), para a direita e para a esquerda, incluindo os bits que entram e que se perdem;
- Registo de deslocamento com carregamento paralelo (Fig. 2.37);
- Funcionamento da rotação.

## 2.6.7 MÁQUINAS DE ESTADOS

### 2.6.7.1 MODELO DAS MÁQUINAS DE ESTADOS

Um estado é uma situação estável de um circuito, que não muda enquanto o sinal de relógio não tiver uma transição à qual as básculas sejam sensíveis. Uma máquina de

estados é um circuito que pode transitar entre vários estados, quando o relógio tem uma transição e de acordo com uma tabela de estados, que indica, para cada estado em que a máquina pode estar e para cada combinação dos valores das entradas, qual o valor das saídas e qual o estado seguinte para que a máquina transitará no próximo ciclo do relógio. O contador binário da Fig. 2.33 é uma máquina de estados. A Tabela 2.10 é uma tabela de estados que indica qual o próximo estado, dado o estado actual (ou corrente). Neste caso, a sequência de transição é fixa e não depende de nenhuma variável de entrada. Também não há nenhuma saída para além do estado, pelo que se trata de um exemplo simples. O contador binário da Fig. 2.33, com carregamento paralelo, já é mais interessante deste ponto de vista, pois a sequência de transição entre estados já pode ser alterada através das entradas PL e Y3..Y0. Não tem saídas para além do próprio estado (saídas das básculas, Q3..Q0), mas é fácil conceber, por exemplo, uma saída obtida pelo NAND de todas as saídas das básculas e que só vale 1 no estado 0000. Esta saída poderá ser útil para saber quando o contador dá "a volta" e passa de 1111 para 0000. A Fig. 2.35 ilustra outro exemplo de saída obtida a partir do estado (saídas das básculas).

**Nota** O relógio é um sinal fundamental e intrínseco de qualquer máquina de estados síncrona, em que todas as básculas transitam ao mesmo tempo, quando o relógio tem uma transição à qual sejam sensíveis. Por este motivo, o relógio nunca é considerado uma variável de entrada.

Também existem máquinas de estado assíncronas, sem relógio e em que são as próprias variáveis de entrada que, ao variarem os seus valores, fazem a máquina de estados transitar entre estados. A diferença face a um circuito combinatório é o facto de possuirem memória da sequência de sinais ocorrida, podendo a mesma combinação de variáveis de entrada produzir combinações diferentes dos sinais de saída. São no entanto circuitos mais complexos de projectar, com problemas específicos, estando fora do âmbito deste capítulo introdutório.

No caso geral, há dois modelos de máquinas de estados, tal como indicado na Fig. 2.38:

- Modelo de Moore (Fig. 2.38a), em que as saídas dependem apenas do estado corrente;
  - Modelo de Mealy (Fig. 2.38b), em que as saídas dependem também das entradas.
- O modelo de Mealy conduz geralmente a menos estados do que o de Moore, pois algumas combinações das saídas usam apenas um estado (aproveitando variações nas entradas), mas também tem um funcionamento menos claro porque pode haver mudanças da saída mesmo sem o relógio mudar, o que obriga a maiores cuidados.
- O modelo de Moore acaba por ser mais explícito, garantindo-se que durante um estado as saídas não mudam. Em máquinas de estados projectadas manualmente privilegia-se a clareza e a facilidade de compreensão, razão pela qual neste contexto se adopta normalmente este modelo, tal como sucede nos exemplos seguintes.

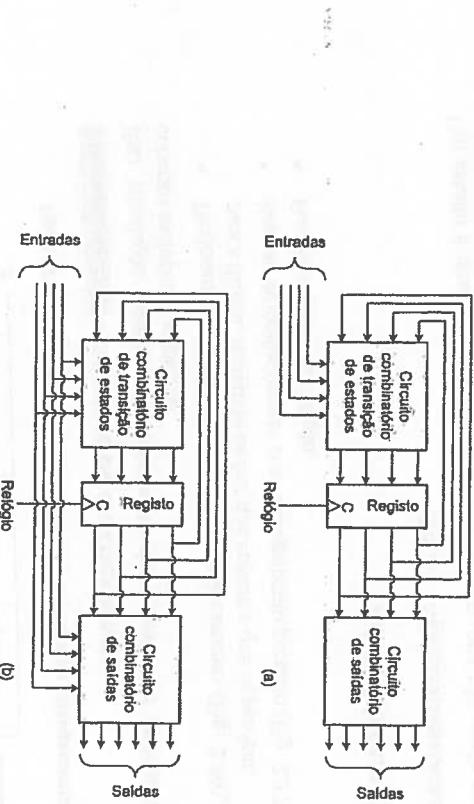


Fig. 2.38 - Modelos de máquinas de estados. (a) - Modelo de Moore; (b) - Modelo de Mealy

### 2.6.7.2 DIAGRAMAS DE ESTADOS

Esta secção ilustra como especificar uma máquina de estados, usando alguns exemplos. A transição entre estados e os valores das saídas podem ser especificados por uma tabela de estados ou por um diagrama de estados, que essencialmente é uma representação gráfica da primeira, permitindo uma mais fácil visualização e compreensão do funcionamento por parte do projectista. Para facilitar a identificação dos estados, podem ser usados nomes.

#### MICROONDAS SIMPLES

Imagine-se o controlo de um microondas rudimentar, com apenas os seguintes recursos:

- Duas entradas, um interruptor que diz se a porta está aberta (1) ou fechada (0) e um botão que o utilizador deve ligar (pôr a 1) quando quiser que o forno funcione e desligar (pôr a 0) quando quiser parar o aquecimento;
- Duas saídas, uma que liga (1) ou desliga (0) a lâmpada interior do forno e outra que liga (1) ou desliga (0) o magnetron (o gerador de microondas).

Para ser mais simples, não há temporizador e tem de ser o utilizador a ligar/desligar o botão para ligar/desligar as microondas. O único automatismo é o desligar do magnetron (por segurança) caso se abra a porta com ele ligado (razão pela qual o botão ligar/desligar não actua directamente no magnetron mas é apenas uma entrada para a máquina de estados).

A Fig. 2.39 descreve este funcionamento por meio de um diagrama de estados (neste caso apenas três), a que corresponde a tabela de estados indicada na Tabela 2.11.

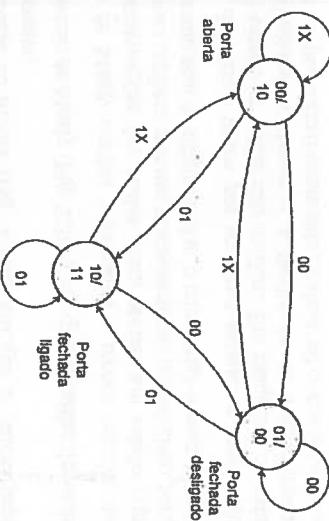


Fig. 2.39 - Diagrama de estados do microondas simples

Cada bola representa um estado, com um número único (como são três estados bastam 2 bits) seguido de uma “/” e do valor das variáveis de saída (pela ordem indicada na Tabela 2.11, lâmpada e magnetron) que vigoram nesse estado.

ESTADO ACTUAL	ESTADO SEGUINTE (P/ DOIS)	SALIDAS
ESTADO ACTUAL	NOTA	ESTADO ACTUAL
Q1 Q0	00 00	01 / 11 LÂMPADA Magnetron
Porta aberta	00	01 / 10 00 00 1 0
Porta fechada desligado	01	01 / 10 00 00 0 0
Porta fechada ligado	10	01 / 10 00 00 1 1

Tabela 2.11 - Tabela de estados do microondas simples

Cada seta representa uma transição entre estados, que se dá quando o relégio faz as basculas transitar de estado, e tem também um número associado, que é o valor que as entradas têm de ter (pela ordem indicada na Tabela 2.11, interruptor da porta e botão) para a transição para o estado seguinte se dar através dessa seta. O conjunto das setas que saem de um estado deve contemplar todas as combinações possíveis das variáveis de estado. Se o valor de uma variável for irrelevante numa dada transição, esse valor pode ser substituído por um “X” (tal como exemplificado pela Fig. 2.39, querendo dizer que essa transição se dá, seja esse valor 0 ou 1).

A tabela de estados (Tabela 2.11) mostra como é possível a partir do estado actual (saídas das basculas Q1 e Q0) e das entradas (P e B, de porta e botão) derivar as equações que definem o estado seguinte (entradas das basculas D1 e D0) e as saídas (I. e M, de lâmpada e magnetron). O circuito correspondente está representado na Fig. 2.40.

Um aspecto fundamental é compreender que o valor de uma variável de entrada que conta para a transição de estados é o valor que a variável tem na altura em que as basculas transitam de estado. Neste exemplo, se o utilizador carregar brevemente no botão, de tal

forma que quando o relógio mude o valor da variável de entrada já esteja outra vez a zero. A máquina nem sequer dá por esse breve impulso. Para garantir que este exemplo funcione bem, o utilizador tem de carregar no botão por mais de 1 segundo, que é o período do relógio. Existem técnicas para resolver este problema com recurso a um circuito extra, mas que se encontram fora do âmbito introdutório deste capítulo. No entanto, a solução mais comum é usar uma frequência de relógio mais elevada (de tal modo que se garanta que o acto de carregar no botão demora muito mais do que um ciclo de relógio), com contagem dos tempos por outros meios (com técnicas explicadas ao longo deste livro).

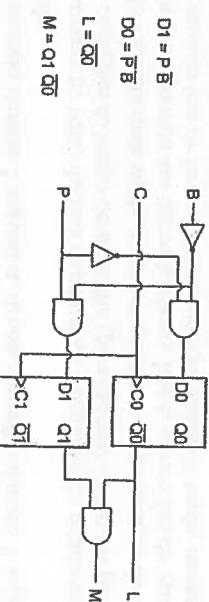


Fig. 2.40 - Circuito do controlo do microondas simples

**NOTA**

Não são os sinais  $L$  e  $M$  que fornecem directamente a energia para a lâmpada do interior do forno e para o magnetron funcionarem. Estes sinais têm 5 V ou menos, enquanto aqueles dispositivos funcionam a 220 V. É necessário usar circuitos electrónicos adicionais que façam a conversão, mas sob controlo do circuito da Fig. 2.40. Estes circuitos estão fora do âmbito deste livro.

**SIMULAÇÃO - MÁQUINA DE ESTADOS SIMPLES**

Esta simulação ilustra o funcionamento da máquina de estados do controlo do microondas, com o circuito da Fig. 2.40. Os aspectos cobertos incluem os seguintes:

- Verificação do comportamento da máquina de estados, nomeadamente em relação à Fig. 2.39 e à Tabela 2.11;
- Demonstração de que uma actuação momentânea de uma variável de entrada, que não apanhe uma transição do relógio, não é detectada pela máquina de estados.

**SEMAFORO SIMPLES**

Outro exemplo é o bem conhecido semáforo de trânsito, com rotação temporizada entre os estados verde, amarelo e vermelho, em ciclo infinito. Assumem-se os seguintes tempos:

- Verde - 5 segundos;
- Amarelo - 2 segundos;
- Vermelho - 7 segundos.

**SEMAFORO SIMPLES**

Se pretendessemos acrescentar um semáforo de peões teríamos de ter mais uma saída (para o verde do semáforo de peões, sendo o vermelho a negação do verde), em que o verde de peões estaria activo durante os 5 segundos do meio do estado vermelho do semáforo das viaturas, dando assim alguma folga (1 segundo no início e outro no fim), tal como sucede num semáforo real (exercício 2.23).

**SEMAFORO COM BOTÃO PARA PEÕES**

Ao contrário do semáforo simples, que está constantemente a circular pelas três cores, este está sempre em verde e só quando um peão carrega no botão é que a máquina de estados faz um ciclo completo, voltando depois ao verde. O funcionamento pode ser descrito desta forma:

- O estado normal é verde, podendo manter-se indefinidamente se o botão não for carregado;

Como o máximo divisor comum destes valores é 1 segundo, a forma mais fácil de implementar este sistema é utilizar um relógio de frequência de 1 Hz (período de 1 segundo) e com 14 estados ( $5 + 2 + 7$ ), com as saídas para controlar as lâmpadas verde, amarela e vermelha activas durante o tempo respectivo. Esta máquina terá assim quatro básculas (três não chegam porque só suportariam oito estados), três saídas (para as três lâmpadas) e nenhuma entrada (uma vez que o ciclo é fixo). A Tabela 2.12 exprime a evolução dos estados e das saídas, tal como a Fig. 2.41.

Nº	Nome	ACTUAL	SEGUINTE	SAÍDAS (NO ESTADO ACTUAL)		
				VERDE	AMARELO	VERMELHO
0	Verde1	0000	0001	1	0	0
1	Verde2	0001	0010	1	0	0
2	Verde3	0010	0011	1	0	0
3	Verde4	0011	0100	1	0	0
4	Verde5	0100	0101	1	0	0
5	Amarelo1	0101	0110	0	1	0
6	Amarelo2	0110	0111	0	1	0
7	Vermelho1	0111	1000	0	0	1
8	Vermelho2	1000	1001	0	0	1
9	Vermelho3	1001	1010	0	0	1
10	Vermelho4	1010	1011	0	0	1
11	Vermelhos	1011	1100	0	0	1
12	Vermelhos	1100	1101	0	0	1
13	Vermelho7	1101	0000	0	0	1

Tabela 2.12 - Tabela de estados do semáforo simples

seguinte depende do valor da entrada. Cada um dos dois arcos que sai de um estado com dois estados seguintes tem o valor que a entrada deve ter para o estado transitar para o próximo estado por esse arco;

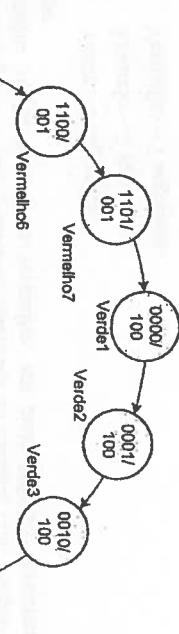


Fig. 2.41 - Diagrama de estados do semáforo simples

Quando o peão carrega no botão, mantém o verde durante 2 segundos, depois passa para amarelo durante 2 segundos, depois vermelho durante 7 segundos e volta ao verde;

Se o peão carregar no botão durante os primeiros 4 segundos do estado vermelho, recomeça a contagem do tempo de vermelho (para dar mais tempo aos peões);

Ao passar de vermelho para verde, o sistema garante o tempo mínimo de 5 segundos de verde (para permitir aos carros arrancar). Isto é, entre o fim do vermelho e o início do amarelo tem de haver pelo menos 5 segundos de verde, mesmo que o peão carregue no botão no primeiro segundo de verde;

Se o peão carregar no botão durante o período mínimo de verde, o pedido deve ficar registado até poder ser atendido.

O diagrama de estados que implementa este funcionamento está descrito na Fig. 2.42. Este diagrama é semelhante ao da Fig. 2.41, mas com o seguinte funcionamento:

- Este circuito tem uma entrada, o valor do botão, que se assume valer 1 quando o botão está carregado e 0 em caso contrário;

- Alguns estados têm dois estados seguintes (dois arcos a sair desse estado), como por exemplo os estados Verde2 e Vermelho3. Isto sucede quando o estado

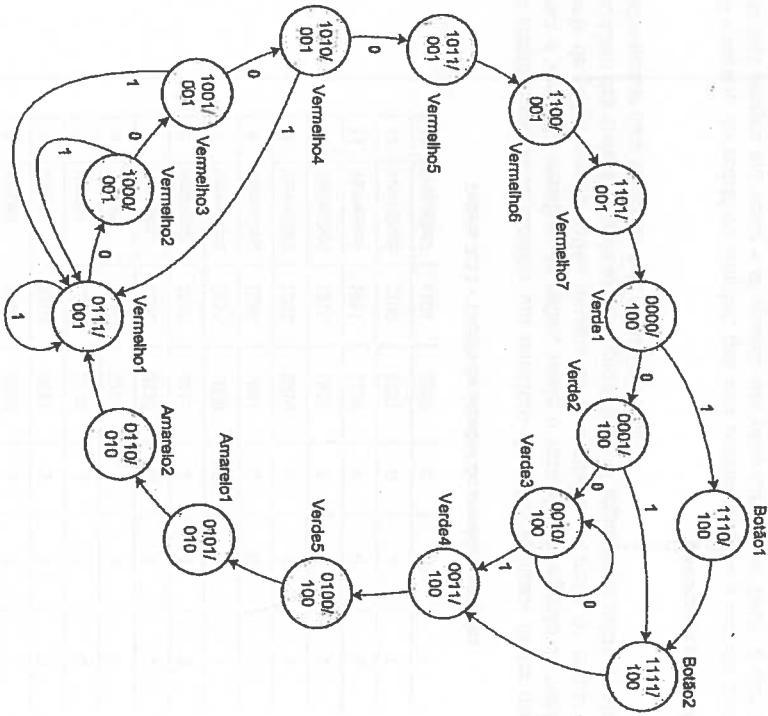


Fig. 2.42 - Diagrama de estados do semáforo com botão para peões

Outros estados têm apenas um estado seguinte. A transição é independente do valor da variável de entrada, o botão, razão pela qual o seu valor não se especifica;

O estado Verde3 transita para ele próprio enquanto o botão estiver a 0, ou seja, enquanto não for carregado. Isto corresponde ao verde por tempo indeterminado, até um peão carregar no botão. Quando o botão for 1, transita para o estado Verde4 e depois Verde5, que correspondem aos 2 segundos a verde após carregar no botão;

Os estados Verde1 e Verde2 garantem, juntamente com os estados Verde3, Verde4 e Verde5, os 5 segundos de verde no mínimo. Se o ciclo começasse pelo

Verde3 e se o peão carregasse logo no botão, poderia haver apenas 3 segundos de verde. No entanto, para que no caso de o peão carregar no botão durante os estados Verde1 e Verde2 essa informação não se perca mesmo que o botão deixe de ser carregado, foram criados os estados Botão1 e Botão2, que desempenham igualmente a função de espera, mas transitando depois diretamente para o estado Verde4, sem se bloquear no estado Verde3. É uma via alternativa, que no total espeta os 5 segundos mas que já não pára à espera de carregar de novo no botão.

Depois do verde, o sistema passa sem escolha pelos estados da luz amarela, e chega aos estados da luz vermelha,

Tal como indicado nas especificações do sistema, durante os primeiros quatro estados da luz vermelha o estado seguinte volta a ser o estado Vermelho1 caso o botão seja de novo premido. Mas o mesmo já não se passa com os estados Vermelhos, Vermelhoso e Vermelho7, em que mesmo que o peão carregue no botão a sequência já não é interrompida.

Pode assim verificar-se que é possível introduzir as especificações do sistema directamente na máquina de estados, e que o diagrama de estados é uma forma bastante clara de verificar o seu funcionamento.

Nº	Nome	ESTADOS			SAÍDAS (NO ESTADO ACTUAL)			
		ACTUAL	SEGUINTE	BOTÃO1	BOTÃO2	VERDE	AMARELO	VERMELHO
0	Verde1	0000	0001	1110	1	0	0	
1	Verde2	0001	0010	1111	1	0	0	
2	Verde3	0010	0010	0011	1	0	0	
3	Verde4	0011	0100	0100	1	0	0	
4	Verde5	0100	0101	0101	1	0	0	
5	Amarelo1	0101	0110	0110	0	1	0	
6	Amarelo2	0110	0111	0111	0	1	0	
7	Vermelho1	0111	1000	0111	0	0	1	
8	Vermelhos2	1000	1001	0111	0	0	1	
9	Vermelhos3	1001	1010	0111	0	0	1	
10	Vermelhos4	1010	1011	0111	0	0	1	
11	Vermelhos5	1011	1100	1100	0	0	1	
12	Vermelhos6	1100	1101	1101	0	0	1	
13	Vermelhos7	1101	0000	0000	0	0	1	
14	Botão1	1110	1111	1111	1	0	0	
15	Botão2	1111	0011	0011	1	0	0	

Tabela 2.13 - Tabela de estados do semáforo com botão para peões

A Tabela 2.13 contém a mesma informação que a Fig. 2.42, mas não consegue exprimir com tanta clareza o funcionamento do sistema. Em matéria de máquinas de estados, um diagrama vale mais que uma tabela. Os estados Botão1 e Botão2 foram acrescentados no final, para não alterar a numeração de estados já feita para o exemplo do semáforo simples. Notem-se as duas colunas de estados seguinte. Se houvesse duas variáveis de saída, teria de haver quatro colunas de estados seguintes (como já sucedeu na Tabela 2.11), e assim sucessivamente. Cada estado no diagrama de estados teria até quatro saídas de saída. Com muitas variáveis de entrada, os diagramas de estados e as tabelas de estados ficam pouco manejáveis.

### 2.6.7.3 MÁQUINAS DE ESTADOS SINTETIZADAS

Com as tabelas de estados dos dois exemplos dos semáforos, Tabela 2.12 e Tabela 2.13, é possível sintetizar os respectivos circuitos sequenciais, tal como já foi feito com a Fig. 2.40 para o exemplo do microondas simples.

O problema é que o circuito se complica para qualquer máquina de estados que ultrapasse o trivial. O circuito do semáforo de peões simples já é mais complicado, e o do semáforo com botão de peões ainda mais. Existem programas que permitem derivar circuitos automaticamente a partir de tabelas de verdade, mas o facto é que cada vez que o diagrama de estados/tabela de estados é mudado o circuito muda e obriga a substituir o hardware. É uma solução possível, mas pouco flexível.

### 2.6.7.4 MÁQUINAS DE ESTADOS MICROPROGRAMADAS

Existe uma alternativa mais simples e flexível para programar máquinas de estados, mesmo complexas, usando uma ROM (seção 2.5.4), representada na Fig. 2.43. Os dois elementos fundamentais são um registo e uma ROM.

O registo mantém o estado actual do sistema (mudado em cada ciclo do relógio) e é constituído por  $N$  basculas, tantas quantas as necessárias para poder representar todos os estados. Por exemplo, para suportar os 14 estados do semáforo simples (Fig. 2.41) são precisas 4 basculas (porque  $16=2^4$  é a menor potência de 2 maior ou igual a 14).

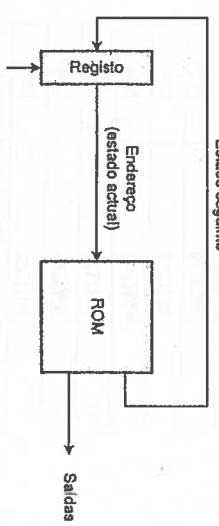


Fig. 2.43 - Esquema básico de uma máquina de estados microprogramada, com uma ROM (sem variáveis de entrada)

**Nota**

O relógio é aqui representado apenas para o seu papel ficar mais explícito. Normalmente nem se representa, pois todas as basculas o recebem de forma sincronizada e portanto tem sempre de existir.

A ROM é obtida directamente da tabela de estados (Tabela 2.12 no exemplo do semáforo simples). O seu endereço é o valor do registo (estado actual). A sua palavra, para cada estado actual, inclui o número do estado seguinte (4 bits), e as saídas no estado actual (3 bits).

A Tabela 2.14 ilustra o conteúdo desta ROM neste exemplo (7 bits de saída). Note-se que os dois últimos endereços podem conter qualquer valor (representados por "X"), pois nunca serão usados pela máquina de estados (só são usados 14 e 4 bits permitem 16). Esta tabela deve ser comparada com a Tabela 2.12, constatando-se um mapeamento directo entre as duas.

As grandes vantagens deste esquema são:

- O conteúdo da ROM é derivado directamente da tabela de estados, sem necessidade de simplificação de expressões nem de construir circuitos com portas 16x16 (o circuito base é comum para todas as máquinas de estados);
- Para alterar a máquina de estados, desde que não se ultrapasse os 16 estados nem se use mais de três variáveis de saída, basta alterar o conteúdo da ROM (que nesse caso teria de ser reprogramável) ou substituí-la, mas sem alterar o circuito.

Esta programabilidade de um mesmo circuito base vale a estas máquinas de estados a designação de *microprogramadas*. O prefixo "micro" designa-se a distinguir esta programação em muito baixo nível da programação de computadores, que mesmo em linguagem *Assembly* (secção 3.4, na página 149) se faz a um nível bem superior. Cada palavra da ROM é designada *microinstrução*. Se quiséssemos acrescentar uma saída para um semáforo de peões (verde/vermelho), bastava acrescentar mais um bit na palavra da ROM e preencher o valor desse bit em cada um dos endereços da ROM.

No entanto, esta máquina de estados não suporta variáveis de entrada, o que se resolve na Fig. 2.44, com um *multiplexer* que selecciona um dos possíveis estados seguintes com base no valor da variável de entrada. Há aqui duas hipóteses alternativas:

- a ROM especifica os dois estados seguintes possíveis (Fig. 2.44a);
- assume-se que por omissoão que uma das hipóteses é o estado com número a seguir, caso em que se pode usar um simples somador e poupar alguns bits na ROM (Fig. 2.44b).

Esta última não pode ser usada caso haja algum estado que tenha dois estados seguintes diferentes do seu estado actual acrescido de uma unidade (o que é o caso do semáforo com botão de peões, cuja ROM tem de incluir os dois estados seguintes e pode ser retirada directamente da Tabela 2.13).

ENDERECO	PALAVRA DA ROM (4 BITS DO ESTADO SEGUINTE + 3 BITS DAS SAÍDAS)
0000	0001100
0001	0010100
0010	0011100
0011	0100100
0100	0101100
0101	0110010
0110	0111010
0111	1000001
1000	1001001
1001	1010001
1010	1011001
1011	1100001
1100	1101001
1101	0000001
1110	XXXXXXX
1111	XXXXXXX

Tabela 2.14 - Conteúdo da ROM no exemplo do semáforo simples

Suporar mais variáveis de entrada é fácil, sendo necessário a ROM especificar, em cada palavra, todos os estados seguintes possíveis. Isto pode levar a algumas repetições, quando para algumas combinações das variáveis de entrada a máquina transitar para o mesmo estado, mas as ROMs actuais têm já grandes capacidades e portanto este facto não é problema.

A secção 7.2.4, na página 585, apresenta mais detalhes sobre a microprogramação.

### **SIMULAÇÃO 2.14 - MÁQUINAS DE ESTADOS MICROPROGRAMADAS**

Esta simulação ilustra o funcionamento das duas máquinas de estados do controlo do semáforo, sem e com botão de peões, usando o circuito da Fig. 2.44a. Os aspectos cobertos incluem os seguintes:

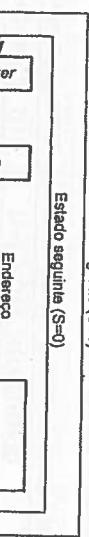
- Preenchimento da ROM;
- Verificação do comportamento das máquinas de estados, nomeadamente em relação aos respectivos diagramas de estados;
- Inclusão da saída adicional para um semáforo de peões e reprogramação da ROM para esse efeito.

Estado seguinte ( $S=1$ )

Estado seguinte ( $S=0$ )

(Estado actual)

Endereço



S (variável de entrada)  
Relógio  
Estado seguinte ( $S=1$ )  
Estado seguinte ( $S=0$ )

(Estado actual)

Endereço

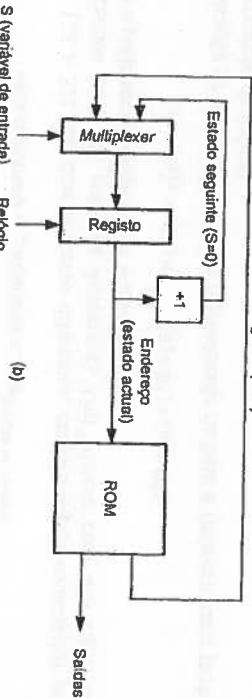
Registo

Multiplexor

Multiplexor

ROM

Saídas



S (variável de entrada)

Relógio

Estado seguinte ( $S=1$ )

(Estado actual)

Endereço

Registo

Multiplexor

Multiplexor

ROM

Saídas

Fig. 2.44 - Esquema básico de uma máquina de estados microprogramada, com uma ROM (com uma variável de entrada). (a) - versão mais genérica; (b) - com incremento de estado por omissão

### ESSENCIA

- Os circuitos sequenciais possuem realimentações a partir das saídas que lhes permitem memorizar estados. Os trunços (*jatches*) são sensíveis a um nível do relógio. As básculas (*flip-flops*) são sensíveis a um dos flancos do relógio.
- Os registos podem ser feitos com vários trunços ou básculas e tipicamente têm saída triticeira para poderem ligar directamente a outros registos sem conflitos.
- Os contadores são registos em que a entrada das básculas (ao podem ser trunço) é calculada para o valor seguinte do registo ser igual ao anterior mas/menos um.
- No registo de deslocamento, a entrada de uma báscula liga à saída da do lado.
- As máquinas de estado permitem implementar uma funcionalidade arbitrária complexa usando um registo para manter o estado e usando tipicamente uma ROM para dar saídas em cada estado e indicar qual o estado seguinte. As unidades pode querer qual o estado seguinte, geralmente usando um multiplexor que escolhe um dos vários estados seguintes possíveis.

## 2.7 REPRESENTAÇÃO DE NÚMEROS

Até aqui, vimos como as portas lógicas podiam ser combinadas para implementar funções que a partir de entradas produziam saídas, com ou sem memória (estados), de forma a implementar funcionalidades interessantes, como o exemplo do controlador de semáforos. No entanto, os computadores destinam-se a muito mais do que simplesmente controlar sinais que comandam dispositivos do mundo real. Essencialmente, a grande área de aplicação dos computadores é o processamento de informação, sempre representada por números (binários), uma vez que os computadores só sabem lidar com dois valores, 0 e 1). Mesmo os textos (caracteres – letras, dígitos e sinais de pontuação) e as imagens (pixels – pontos individuais da imagem) são números. O trunço é reconhecer um conjunto de entidades e atribuir um código (um número) a cada elemento desse conjunto, de maneira a poder identificá-lo de forma única em relação aos restantes elementos.

Por exemplo, os caracteres são muitas vezes representados pelo código ASCII (American Standard Code for Information Interchange), descrito no Apêndice E, na página 743. Este código é apenas uma convenção, em que se definem 128 símbolos (letras, dígitos e sinais de pontuação) e se atribui um código, um número entre 0 e 127, a cada um deles. O que um computador faz não é mais do que armazenar e processar esses números, sem saber realmente o que está a fazer. É a inteligência do programador, que desenvolveu o programa, que faz com que pareça que um computador está a fazer processamento de texto, por exemplo. Mas não está. Do ponto de vista do computador, está apenas a processar números entre 0 e 127. No ecrã é feita a correspondência entre cada código de carácter e a sua representação gráfica (desenho), o que completa a ilusão.

Nesta perspectiva, é fundamental perceber como é que um computador consegue representar e processar números, quando apenas conhece dois símbolos, 0 e 1.

### 2.7.1 NÚMEROS EM BASE 10 (DECIMAIS) E 2 (BINÁRIOS)

A Tabela 2.15 compara as pessoas e os computadores no que diz respeito à representação dos números. As pessoas usam base 10 (decimal, com 10 símbolos, 0 a 9) porque têm 10 dedos nas mãos, enquanto a restrição fundamental dos computadores é que só podem usar dois símbolos, razão porque representam os números em base 2 (binário, com dois símbolos, 0 e 1). O número de símbolos usado define a base de numeração.

Não há qualquer diferença conceptual entre usar base 10, 2 ou outra qualquer. A sequência de contagem, referida nesta tabela, indica que os números se formam fazendo todas as combinações de símbolos, primeiro com um algarismo (um símbolo ocupando uma dada posição num número), depois com dois, depois com três, e assim sucessivamente. Os algarismos da base decimal designam-se dígitos, enquanto os da base binária se designam bits (*binary digit*, ou dígito binário).

Outro aspecto muito importante é o facto de, tanto em binário como em decimal, se representar o algarismo (*bit* ou *dígito*) de maior peso do lado esquerdo e o de menor peso do lado direito.

Computador	Pessoas	Computadores
Fundamental	10 dedos nas mãos	2 estados básicos: ligado e desligado
Base de numeração	10 (decimal)	2 (binária)
Número de símbolos	10 (0 a 9)	2 (0 e 1)
Contagem	0, 1, 2, 3, 4, 5, 6, 7, 8, 9 10, 11, 12, 13, 14, 15, ..., 99 100, 101, ..., 999	0, 1 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, ..., 1101, 1110, 1111

Tabela 2.15 - Comparação da forma como as pessoas e os computadores representam os números. Quando se esquecem as combinações dos símbolos, volta-se ao princípio com mais um símbolo

A diferença mais notória é o facto de um dado número precisar de muitos mais algarismos em binário do que em decimal para ser representado, o que no entanto não é de estranhar pois os números binários dispõem de apenas dois símbolos. O número decimal 15, por exemplo, necessita apenas de 2 dígitos (Tabela 2.15), enquanto em binário precisa de 4 bits (1111). A simplicidade dos computadores, com apenas dois símbolos, paga-se depois em número de bits.

O princípio básico da construção de qualquer número em qualquer base é usar uma sequência de algarismos, em que a contribuição de cada algarismo para o valor do número é pesada pela sua posição na sequência, a contar da direita para a esquerda:

$$\text{valor do número} = \sum \text{algarismo} \times \text{base}^{\text{posição no número}}$$

Por exemplo, 123 e 321 são dois números diferentes, embora sejam construídos pelos mesmos algarismos (mas em posições diferentes). Um número com três algarismos,  $a_2, a_1, a_0$ , na base B pode ser decomposto da seguinte forma:

$$(a_2, a_1, a_0)_B = (a_2 \times B^2) + (a_1 \times B^1) + (a_0 \times B^0)$$

Note-se a indicação de que o número está representado na base B (número entre parênteses, com B na posição de índice).

A Tabela 2.16 exemplifica esta decomposição em decimal e em binário. As mesmas regras aplicam-se no caso de representação noutras bases.

DECIMAL		BINÁRIO	
NÚMERO	FACTORES	NÚMERO	FACTORES
5	$5 \times 10^0$	101	$(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$
15	$(1 \times 10^1) + (5 \times 10^0)$	1111	$(1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
29	$(2 \times 10^1) + (9 \times 10^0)$	11101	$(1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

Tabela 2.16 - Decomposição em factores de números decimais e binários

Dado um número, é possível mudar a base em que ele está representado, usando a decomposição indicada na Tabela 2.16. Por exemplo, para converter um número em decimal para binário deve seguir-se o seguinte algoritmo, determinando os bits da direita (menor peso) para o esquerda (maior peso):

- Dividir o número por 2, passando o quociente inteiro a ser o novo número. O resto da divisão (0 ou 1) é o bit a usar;
- Se o novo número é maior ou igual a 2, voltar ao passo 1;
- O último bit (o mais à esquerda) é o novo número (quociente da última divisão) que restar (inferior a 2).

Com exemplo, considere-se o número 29 já usado na Tabela 2.16 e a divisão sucessiva de 29 por 2, representada na Fig. 2.45. Os restos das divisões e o último quociente são o número binário indicado na Tabela 2.16 (11101), mas ao contrário (o resto da primeira divisão é o bit mais à direita).

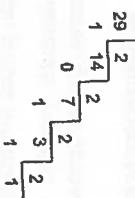


Fig. 2.45 - Conversão de um número decimal em binário

Para converter um número binário em decimal basta somar os vários factores (que são potências de 2) correspondentes a cada bit (só se for 1) no número binário, tal como feito na coluna da direita da Tabela 2.16.

## 2.7.2 NÚMEROS EM BASE 16 (HEXADECIMAIS)

A grande vantagem de usar a base hexadecimal é representar números grandes com quatro vezes menos algarismos do que em binário. Existe uma correspondência directa entre um número em binário e em hexadecimal, em que cada grupo de 4 bits (designado nibble) do número em binário corresponde a um dígito hexadecimal, de acordo com a Tabela 2.17.

Tal como a tabuada, que se sabe de cor, é de toda a conveniência saber de cor a correspondência entre os números decimais 0 a 15 e os correspondentes números binários 0000 a 1111, tal como indicado na Tabela 2.17. Mais do que isso é difícil decorar, e 4 bits contêm metade de um byte (8 bits), uma unidade muito usada em computadores. De facto, os grupos de 4 bits são tão comuns que acaba por ser mais fácil utilizar a base 16 (base hexadecimal).

Como só temos 10 símbolos para os dígitos, usam-se as 6 primeiras letras do alfabeto (A, B, C, D, E e F) para os restantes símbolos, com os valores 10 a 15, respectivamente. É comum usar o termo dígito hexadecimal para designar um algarismo hexadecimal, que pode tomar um de 16 valores (0..9 ou A..F), tal como indicado na Tabela 2.17.

DECIMAL	BINARIO	HEXADECIMAL
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Tabela 2.17 - Representação dos números 0 a 15 em decimal, binário e hexadecimal

A Fig. 2.46 ilustra com um exemplo. A separação entre grupos de 4 bits destina-se apenas a melhor visualizar o agrupamento (num número binário, como em qualquer outro, os algarismos estão todos seguidos).

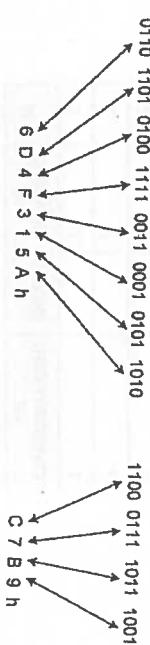


Fig. 2.46 - Correspondência entre as representações de um número em binário e em hexadecimal

O "h" após os números em hexadecimal destina-se a indicar que os números estão em hexadecimal. Nestes exemplos até é óbvio (por causa das letras) que não estão em decimal, mas 1234, por exemplo, pode representar o número  $(1234)_10$  ou  $(1234)_6$ , que são números diferentes. Ou ainda 100, que pode ser interpretado como estando em decimal, binário ou hexadecimal (correspondendo aos valores decimais 100, 4 e 256, respectivamente). Quando não houver confusão (depende do contexto), os números podem representar-se apenas pelos algarismos. Para tornar claro a base em que estão representados, usa-se normalmente uma letra do lado direito: 100d, 100b ou 100h, respectivamente. A letra pode ser maiúscula ou minúscula.

Note-se também que o número binário maior na Fig. 2.46 tem um zero à esquerda, para que o número total de bits (32, neste caso) seja múltiplo de 4 e permitir a conversão para hexadecimal. Um zero à esquerda não afeta o valor de um número.

A conversão entre decimal e hexadecimal faz-se exactamente como entre decimal e binário, em que 2 é substituído por 16. Assim, para converter o número hexadecimal 3CAH para decimal faz-se:

$$(3 \times 16^3) + (C \times 16^2) + (A \times 16^1) + (0 \times 16^0) = (3 \times 256) + (12 \times 16) + (10 \times 1) = 970$$

Para converter 970 em hexadecimal, divide-se sucessivamente por 16, obtendo-se de novo 3CAH.

$$\begin{array}{r} 970 \\ \hline 16 \\ 60 \\ \hline 3 \end{array}$$

Fig. 2.47 - Conversão entre as representações de um número em decimal e em hexadecimal

### 2.7.3 POTÊNCIAS DE 2

Em computadores é normal usar grandes números, não apenas em cálculo numérico mas também em processamento de dados, pois há grandes quantidades de informação para tratar e gerir. O elemento mais básico de informação é o bit mas em termos de dados, na prática, usa-se o byte (8 bits) como unidade. Por exemplo, é comum exprimir-se a capacidade dos discos de um computador em gigabytes (ou GBs, ou GB) ou mesmo em terabytes (ou TBs, ou TB), no caso de grandes computadores. O tamanho de um ficheiro exprime-se normalmente em bytes, KBytes (ou KB), megabytes (ou MBs, ou MB) ou até mesmo GBytes. Em vez das designações por extenso (10 megabytes, por exemplo), é mais frequente usar as designações mais abreviadas (10 MBytes ou 10 MB).

Como os computadores funcionam em base 2, estas designações para os prefixos de byte referem-se a números que são potências de 2, tal como indicado pela Tabela 2.18, e não 10, como é usual por exemplo na física. Assim, o factor multiplicativo entre designações consecutivas não é 1000 mas sim 1024, pois este é o valor mais próximo de 1000 que é potência de 2. Isto faz com que, por exemplo, 1 MByte (um megabyte) não seja 1.000.000 mas sim 1.048.576 bytes (1024x1024).

SÍMBOLO	LIGA	ÉQUIVALENTE	POTÊNCIA DE 2	VALOR DECIMAL	POTÊNCIA DE 10 (APROXIMADA)
K	kapa	1024	2 <sup>10</sup>	1.024	10 <sup>3</sup>
M	mega	1024 K	2 <sup>20</sup>	1.048.576	10 <sup>6</sup>
G	giga	1024 M	2 <sup>30</sup>	1.073.741.824	10 <sup>9</sup>
T	tera	1024 G	2 <sup>40</sup>	1.099.511.627.776	10 <sup>12</sup>

Tabela 2.18 - Principais factores multiplicativos das potências de 2

**NOTA**

Estes factores multiplicativos também se usam para *bits* ou para palavras. Por exemplo, 1 *MByte* (1 MB) corresponde a 8 *Mbits* (8 Mb) e 1 M palavras de 16 bits cada são equivalentes a 2 *MBytes*. Na versão mais abreviada da unidade, o B (minúsculo) designa *byte* e b (minúsculo) representa *bit*. Para evitar confusões, no texto deste livro usam-se as versões com *byte* e *bit* por extenso, abreviando apenas o factor multiplicativo. A versão mais abreviada é apenas usada em fórmulas ou tabelas em que o espaço seja limitado.

No entanto, nem todos os valores são tão certos, embora normalmente tudo gire em torno das potências de 2, pelo que é conveniente saber de cor os valores das potências mais usadas (tipicamente até  $2^{16}$  e as da Tabela 2.18).

N.	$2^N$ (DECIMAL)	K (HEXADECIMAL)	$2^N$ (HEXADECIMAL)
0	1	1	1
1	2	2	2
2	4	4	4
3	8	8	8
4	16	10H	10H
5	32	20H	20H
6	64	40H	40H
7	128	80H	80H
8	256	100H	100H
9	512	200H	200H
10	1024	1K	400H
11	2048	2K	800H
12	4096	4K	1000H
13	8192	8K	2000H
14	16384	16K	4000H
15	32768	32K	8000H
16	65536	64K	10000H

Tabela 2.19 - Primeiras potências de 2

A Tabela 2.20 mostra como calcular o valor de uma outra qualquer potência de 2, usando alguns truques simples, com base na decomposição do expoente dessa potência em expoentes cuja potência já tenha valor conhecido (sabendo que a multiplicação de potências da mesma base é obtida pela soma dos expoentes, e a divisão pela subtração).

Tabela 2.20 - Exemplos de cálculo de potências de 2 com base em potências já conhecidas

**2.7.4 QUANTOS BITS PARA REPRESENTAR UM NÚMERO?**

Ab contrário do que a Tabela 2.19 poderá eventualmente sugerir, olhando para a coluna de direita, os números num computador não se representam apenas com a quantidade de bits estritamente necessária. Como as unidades que processam números (um somador, por exemplo) têm de funcionar com qualquer valor<sup>13</sup>, têm de se definir qual a gama de valores que se pretende suportar, em que os extremos dessa gama definem quantos bits usar.

A partir daí, todos os números têm de ser representados com igual número de bits, até mesmo o zero. Os números mais pequenos têm zeros à esquerda para perfazer o número de bits necessário. Por exemplo, a Tabela 2.17 mostra todos os números binários de 0 a 15 representados com 4 bits, em que os primeiros têm zeros à esquerda.

O número de bits usado, mas uma vez, é tipicamente uma potência de 2. A Tabela 2.21 ilustra os casos mais comuns e a designação mais frequente que lhe corresponde, embora nem todos os sistemas atribuam as mesmas designações aos números representados com mais de 8 bits.

N.º BITS	NOME	GAMA	N.º BYTES
1	bit	0...1	--
4	nibble	0..15	--
8	byte	0..255	1
16	Palavra de 16 bits (short)	0..64 K-1	2
32	Palavra de 32 bits (word)	0..4 G-1	4
64	Palavra de 64 bits (double word)	0..2 <sup>64</sup> -1	8

Tabela 2.21 - Designações usuais para o tamanho (em bits) das representações de números e respectiva gama de valores representáveis

<sup>13</sup> O hardware é fixo e não se pode alterar consoante se quer processar números de maior ou menor valor.

Esta representação, que com  $N$  bits permite representar números entre 0 e  $2^N - 1$ , designa-se por representação sem sinal (*unsigned*), porque só contempla números positivos.

### 2.7.5 REPRESENTAÇÃO DE NÚMEROS NEGATIVOS

Os números referidos até aqui são todos não negativos. Mas os números negativos existem e precisam de ser representados. A forma mais óbvia de o fazer é reservar um bit (o de maior peso, o da esquerda) para o sinal, com os restantes a indicar o módulo ('valor absoluto do número'). É a chamada representação em módulo e sinal. Se tiver 1 no bit de sinal é negativo, caso contrário é positivo. Esta representação tem o problema de ter duas representações para o número zero, uma "positiva" e outra "negativa", o que pode causar problemas.

Hoje em dia, a representação universalmente utilizada é a representação em complemento para 2. A parte não negativa (zero e números positivos) é igual nas duas representações, mas a parte negativa está invertida e o zero "negativo" não existe em complemento para 2, tal como se pode constatar na Tabela 2.22, que ilustra a escala de 4 bits para números positivos e negativos que é possível representar com 4 bits. Em qualquer dos dois casos, um 1 no bit de maior peso indica que o número é negativo.

NÚMERO	MÓDULO E SINAL	COMPLEMENTO PARA 2
+7	0111	0111
+5	0110	0110
+3	0101	0101
+4	0100	0100
+2	0011	0011
+1	0010	0010
0	0001	0001
-1	0000	0000
-2	1001	1111
-3	1010	1110
-4	1011	1101
-5	1100	1100
-6	1101	1011
-7	1110	1010
-8	1111	1001

Tabela 2.22 - Representações de números positivos e negativos, em módulo e sinal e em complemento para 2

É importante notar que:

- Em módulo, o valor do maior inteiro que se consegue representar reduz-se a cerca de metade em relação ao caso dos inteiros sem sinal, pois é preciso contemplar positivos e negativos;
- É obrigatório sabermos com quantos bits estamos a representar os números. Por exemplo, o número 1111 é negativo com 4 bits, mas se o considerássemos com 8 bits e zeros à esquerda, 0000 1111, já seria um número positivo;
- Um conjunto de  $N$  bits não tem, a partida, significado próprio específico, nem nada que diga intrinsecamente qual a sua representação. O seu significado é-lhe atribuído pela operação que o usa como operando. Umas operações podem considerá-lo um número sem sinal, outras como um número em complemento para 2, por exemplo.

As principais vantagens da representação em complemento para 2 são as seguintes:

- Só há uma representação para o zero;
- Os números negativos conseguem representar mais um valor no extremo negativo do que na representação de módulo e sinal;

A soma de um número com o seu simétrico é zero, tal como estamos habituados na aritmética decimal. Por exemplo, a soma de 0001 (1) com 1111 (-1) dá 10000, mas como estamos no reino dos 4 bits o 5.º (o de maior peso, o da esquerda) é descartado e ficam apenas os 4 bits a 0. Não é este o caso da notação em módulo e sinal, como se pode verificar por exemplo somando 0001 (1) com 1001 (-1), o que dá 1010 (-2);

Devido ao ponto anterior, subtrair um número  $B$  de outro  $A$  é igual a somar  $A$  com o complemento para 2 (simétrico) de  $B$ . Basta ver que:

$$A - B = A + (-B) + B - B = A + (-B)$$

Esta última propriedade é importante porque permite usar sempre o mesmo circuito para somar ou subtrair, bastando ter um somador e complementando para 2 o segundo operando se se quiser efectuar uma subtração.

### 2.7.6 REPRESENTAÇÃO DE NÚMEROS EM COMPLEMENTO PARA 2

A representação em complemento para 2 com qualquer número  $N$  de bits segue as seguintes regras básicas (exemplificadas pela Tabela 2.22 para  $N=4$ ):

- Os números não negativos seguem uma numeração normal binária, entre 00...00 (zero, com todos os bits a 0) e 01...11 ( $2^{N-1}$ , valor máximo representável, com todos os bits a 1 excepto o de maior peso, o da esquerda);
- Os números negativos, em módulo crescente, a partir de -1, representam-se desde 11...11 (-1, todos os bits a 1) até 10...00 ( $-2^{N-1}$ , valor mínimo representável, com todos os bits a 0 excepto o de maior peso, o da esquerda);

- Todos os números representáveis com  $n$  bits têm o seu simétrico representável em  $n$  bits também, com exceção do número mais negativo  $10\dots 00$  ( $-2^{n-1}$ ). O simétrico de 0 é 0;
- Dado um número, o seu simétrico é obtido negando os seus bits todos e somando 1 ao resultado.

A soma em binário faz-se exactamente da mesma forma que em decimal, somando os algarismos da direita para a esquerda e contando com os "é vai um". A tabuada da soma é no entanto mais simples, pois  $0+0=0$ ,  $0+1=1$  e  $1+1=10$  (ou 0 e vai um).

A Fig. 2.48 ilustra algumas situações, usando números de 8 bits (mas seria semelhante para qualquer número de bits). A Fig. 2.48a mostra como obter o simétrico de um número, negando todos os seus bits e somando 1. O número era positivo, pelo que o simétrico é negativo. A Fig. 2.48b faz a prova, somando o número com o seu simétrico, que dá zero (descartando o 1 adicional que já não cabe nos 8 bits). Fig. 2.48c mostra que o simétrico de 0 é zero à mesma, o que permite tratar o zero como qualquer outro número e mesmo assim ter uma só representação (em vez de duas, como na representação em módulo e sinal). Finalmente, a Fig. 2.48d obtém o simétrico de um exemplo conhecido (o número -1), cujo resultado é 1. O número era negativo, pelo que o seu simétrico é positivo.

$$\begin{array}{r} 0100\ 0010 \xrightarrow{\text{nega bits}} 1011\ 1101 \\ + 1011\ 1101 \\ \hline 10000\ 0000 \end{array} \quad \begin{array}{r} 0100\ 0010 \\ + 1011\ 1101 \\ \hline 10000\ 0000 \end{array}$$
  

$$\begin{array}{r} 0000\ 0000 \xrightarrow{\text{nega bits}} 1111\ 1111 \\ + 1111\ 1111 \\ \hline 0000\ 0000 \end{array} \quad \begin{array}{r} 1111\ 1111 \xrightarrow{\text{nega bits}} 0000\ 0000 \\ + 0000\ 0000 \\ \hline 0000\ 0001 \end{array}$$
  

$$\begin{array}{r} (a) \qquad \qquad \qquad (b) \\ (c) \qquad \qquad \qquad (d) \end{array}$$

Fig. 2.48 - Complemento para 2. (a) - Obtenção do simétrico de 42H; (b) - Soma de 42H com o seu simétrico; (c) - Simétrico de zero; (d) - Simétrico de -1.

Há uma confusão comum que é importante esclarecer. Uma coisa é a representação em complemento para 2, que obedece às regras atrás enunciadas. Outra coisa é a determinação do simétrico de um número, que infelizmente também se denomina obtenção do complemento para 2 do número original. A primeira é uma notação, a segunda uma operação.

Quando se pede para representar um dado número em complemento para 2, muitas pessoas acham logo o simétrico, seja o número positivo ou negativo. Ou seja, pedir-se uma notação, e a resposta (incorrecta) foi uma operação.

Assim, importa distinguir dois tipos de pedidos diferentes:

- Representar um dado número (geralmente dado em decimal, que é a notação que nós é mais familiar) em complemento para 2. Se for positivo, basta simplesmente converter para binário, tal como indicado na secção 2.7.1. Se for negativo, coi-

verte-se o seu módulo (valor positivo) para binário e depois acha-se o seu simétrico;

Determinar o simétrico (ou o complemento para 2) de um dado número. Neste caso, procede-se como indicado pela Fig. 2.48, seja o número positivo ou negativo. Se o número for dado em decimal, converte-se o seu módulo para binário e, caso o número original fosse positivo, ainda tem de se obter o simétrico, como indicado pela Fig. 2.48a.

Para evitar esta confusão clássica, o melhor é evitar a designação "obter o complemento para 2 de um número" e usar em vez disso "obter o simétrico de um número".

Outro aspecto a reforçar é que, olhando para um dado número em binário (sequência de bits), não é possível responder a alguém que pergunta se esse número está ou não representado em complemento para 2. O que essa sequência de bits representa depende da forma como a operação que sobre ela actua a interpreta. Nos computadores, há operações que assumem que os operandos estão em complemento para 2, outras que estão representados sem sinal, e assim por diante.

## 2.7.7 EXTENSÃO DO NÚMERO DE BITS DE UM NÚMERO

As vezes, há a necessidade de efectuar uma operação sobre dois números representados com um número diferente de bits. Por exemplo, somar um número de 4 bits com outro de 8 bits.

Generalmente, as operações exigem que os dois operandos tenham a mesma gama de valores, até porque algumas devem ser comutativas (soma, por exemplo). Como não se pode cortar o número de bits de um número (pois o valor representado pode ser elevado e precisar dos bits todos), a única solução é estender o número com menos bits até ter o número de bits do outro.

As regras para o fazer em representação de complemento para 2 são muito simples. A extensão é efectuada:

À esquerda (para o lado dos bits de maior peso);

Replicando o bit de sinal (o de maior peso). Se o número original é positivo (ou zero), estende-se com 0s. Se for negativo, estende-se com 1s.

A Tabela 2.23 exemplifica com os números +2 e -2. Os espaços entre grupos de 4 bits destinam-se exclusivamente a facilitar a legibilidade do número. As regras mantêm-se para extensões com outros números de bits (64, por exemplo).

N.º DE BITS	NÚMERO POSITIVO (+2)	NÚMERO NEGATIVO (-2)
4	0010	1110
8	0000 0010	1111 1110
16	0000 0000 0010	1111 1111 1110

Tabela 2.23 - Extensão de um número binário com sinal (em complemento para 2)

Note-se que a manutenção do bit de sinal é fundamental. Se, por exemplo, se estender o número 1110 (-2) para 0000 1110, o número já é outro, positivo e com o valor 14. A extensão implica apenas um aumento do número de bits usados para representar número, mas nunca uma alteração do seu valor.

### ESSENCIAL

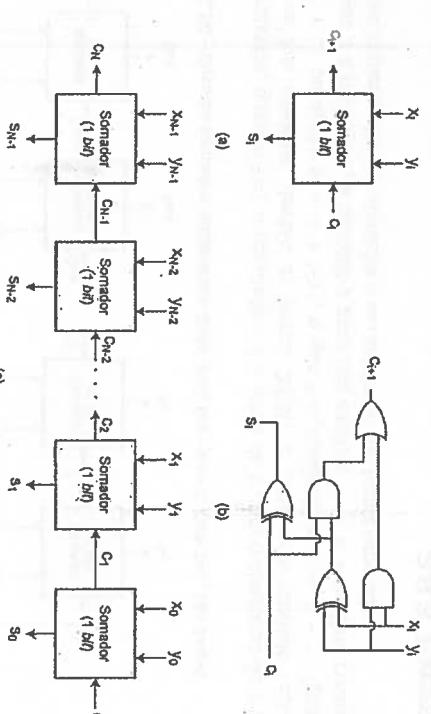
- As pessoas representam os números em base 10, com dez símbolos diferentes (0 a 9). Os computadores representam-nos em base 2, com dois símbolos (0 e 1).
- O mesmo número precisa de mais algarismos em base 2 (*n* bits) para ser representado.
- Cada algarismo em base hexadecimal (16) tem uma correspondência directa com um grupo de 4 bits, permitindo representações mais compactas do que em binário.
- Em qualquer base, um número é obtido pelo somatório dos produtos de cada algarismo pela base elevada à sua posição no número (começando em 0 do lado direito do número, que tem o algarismo de menor peso). Com esta regra, é possível converter um número de uma base para outra.
- N* bits permitem representar números na gama  $0 \dots 2^N - 1$ . Esse representação é designada sem sinal e só permite representar números positivos.
- Para contemplar os números negativos, os números são representados em complemento para 2, em que o bit mais à esquerda indica se o número é positivo (0) ou negativo (1). A metade superior da representação sem sinal é invertida para representar os números negativos, com a representação com todos os bits 1 usada para designar o valor -1.
- Para representar os números em binário num computador usa-se um número de bits, o necessário para representar o número maior (em módulo) da gama de números que o computador é capaz de processar. Os números mais pequenos têm de ser preenchidos à esquerda com bits neutros (0s para números positivos) e para números negativos.
- Representar um número em complemento para 2 é diferente de obter o seu complemento para 2 (simétrico), operação que se faz negando todos os bits de um número e somando-lhe 1.
- A maior vantagem da representação em complemento para 2 é o facto de permitir uma grande uniformidade de tratamento entre os números positivos e negativos. Noteadamente, (i) um somador para números sem sinal funciona igualmente bem com números negativos; (ii) a soma de um número com o seu simétrico dá zero; (iii) o simétrico de zero é zero; e (iv), a subtração de dois números é igual à soma do primeiro com o simétrico do segundo.

Havendo números, há necessidade de efectuar operações sobre eles. Esta secção aborda apenas os números inteiros. O Apêndice D, na página 735, aborda o tema dos números com parte fraccionária.

## 2.8 OPERAÇÕES ARITMÉTICAS

A soma de dois números binários faz-se bit a bit, começando da direita (menor peso) para a esquerda (maior peso), passando o transporte ("é vai um"), se existir, para o bit seguinte. Os dois números devem ter o mesmo número de bits, fazendo a extensão de um úteis, se necessário, de acordo com as regras da secção 2.7.7. A Fig. 2.48 apresenta alguns exemplos.

Uma forma simples de implementar um somador de *n* bits é implementar *N* somadores elementares de 1 bit, com saída de transporte (*carry*) de um somador elementar para o próximo, de forma muito semelhante ao algoritmo que implementamos nas contas feitas manualmente. A Fig. 2.49 ilustra esta organização.



Apesar de o resultado poder ser maior do que qualquer dos dois operandos, o resultado usa geralmente tantos bits como os operandos, até porque esse resultado pode ser, por sua vez, um operando numa soma posterior. O eventual transporte devido à soma dos bits de maior peso (*c<sub>N</sub>*) é geralmente ignorado, como acontece na Fig. 2.48b e Fig. 2.48c, pois já não é representável nos *N* bits que constituem cada um dos operandos. Este último bit de transporte pode ainda assim ter alguma utilidade (secção 2.8.3).

Uma das vantagens da representação em complemento para 2 é o facto de a soma funcionar da mesma forma com números positivos e negativos e não ser preciso tratamento especial para o bit de sinal, que é somado como qualquer um dos outros. O circuito da Fig. 2.49b é derivado da tabela de verdade do somador elementar, descrita pela Tabela 2.24 e que reflecte a tabuada de somar em binário, com transporte.

ENTRADAS		SAÍDAS	
X <sub>0</sub>	X <sub>1</sub>	y <sub>0</sub>	y <sub>1</sub>
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

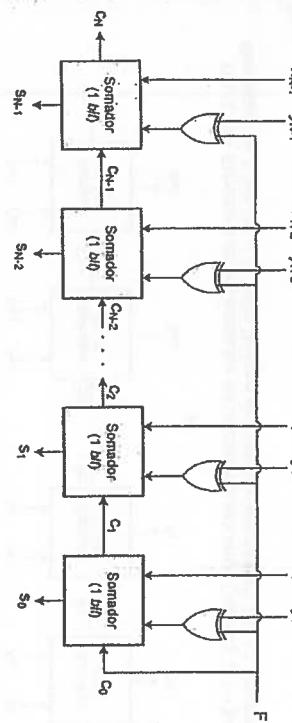
Tabela 2.24 - Tabela de verdade do somador elementar de 1 bit com transporte

Neste tipo de somador as somas individuais dos pares de bits dos dois operandos são feitas todas ao mesmo tempo, mas os bits de transporte vão-se propagando ao longo dos vários somadores elementares, provocando variações nos bits do resultado que potencialmente só estabilizam ao fim de N atrasos de propagação do transporte. Por esta razão, este somador é conhecido por somador de transporte em onda (*ripple-carry adder*).

Para diminuir este tempo total de atraso, os somadores usados comercialmente expandem a dependência de cada transporte ( $c_{i-1}$ ) directamente em função do transporte de entrada ( $c_0$ ) e dos vários resultados de soma de cada par de bits. O circuito fica mais complexo, mas o tempo total de atraso fica menor. Os detalhes desta técnica, conhecida por *carry-look-ahead* (antecipação do transporte), estão fora do âmbito deste livro.

## 2.8.2 SUBTRAÇÃO DE DOIS NÚMEROS BINÁRIOS

É perfeitamente possível projectar um subtrator constituído por subtractores elementares de 1 bit, com uma tabela de verdade de onde se deriva o circuito respectivo. No entanto, neste momento já sabemos que subtrair B de A é equivalente a somar A com o simétrico de B, sejam estes números positivos ou negativos. Usando este truque, é possível projectar um circuito pouco mais complexo que um somador e que consegue fazer tanto somas como subtrações. Este circuito está representado na Fig. 2.50.

Fig. 2.50 - Circuito somador/subtrator. Com  $F=0$ , faz  $S=X+Y$ . Com  $F=1$ , faz  $S=X-Y$ 

Por outras palavras, quando  $F=1$  o resultado  $S$  é a soma de  $X$  com o simétrico de  $Y$  (que tem todos os bits negados, devido às portas XOR) e mais uma unidade ( $c_0$ ), ou seja,  $S = X + (\bar{Y} + 1)$  ou ainda  $S = X + (-Y)$ , o que é equivalente a  $S = X - Y$ . Quando  $F=0$ , o número  $Y$  é inalterado, a unidade a mais não existe e  $S = X + Y$ . Este circuito é melhor do que implementar um somador e um subtrator de forma separada.

## 2.8.3 EXCESSO

Pode perfeitamente suceder que o resultado de uma operação de soma ou subtração não seja representável na gama dos  $N$  bits com que os números estão representados. Reparemos, por exemplo, na Tabela 2.22, e imagine-se a soma de  $+6$  (0110) com  $+3$  (0011). Estes números estão representados com 4 bits e qualquer delles é um número perfeitamente válido. A soma deles devia valer  $+9$ , mas este número já não é representável com 4 bits (tiveria que ser 1001, mas com 4 bits esse número é negativo, -7, como a Tabela 2.22 indica).

O problema é que o resultado da soma é demasiado grande para caber na escala disponível (número máximo +7, com 4 bits). O mesmo raciocínio poderia ser estabelecido para uma soma de dois números negativos, ou uma subtração de dois números de sinais diferentes. Esta situação designa-se excesso (*overflow*).

O somador/subtrator não deixa de dar um resultado, mas está incorrecto. No caso deste exemplo, o resultado seria simplesmente 1001, ou -7, o que nitidamente não está correcto. Se o número é negativo, o que não pode ser atendendo a que se estavam a somar dois números positivos.

A solução trivial é aumentar o número de bits com que se representam os números. Mas, se é qual for o número de bits usados, o problema mantém-se sempre. Pode ocorrer em

apenas lógica XOR pode ser considerada como uma negação de uma entrada controlada pela outra entrada. Só nega se a entrada usada para controlo ( $F$ ) for 1, pois se esta for 0 reproduz na saída a primeira entrada. Nestas condições, os bits  $y_i$  são todos negados se  $F=0$ .

O bit de transporte de entrada do somador ( $c_0$ ) também vale 1 neste caso, o que significa que ao resultado da soma se está a somar mais uma unidade.

números de maior grandeza. Por isso, as unidades somadoras/subtractoras limitam-se a assimilar que houve uma situação de erro motivada por excesso, podendo haver outras unidades que tornem as acções mais adequadas (como por exemplo terminar o programa de um computador em que uma operação de soma ou subtração originou este erro).

A detecção da situação de excesso na adição pode ser feita usando as seguintes regras:

- Se os operandos têm sinal igual, não pode haver excesso;
- Se os operandos têm sinal diferente, ocorre excesso quando o sinal do resultado for diferente do do primeiro operando.

Na prática, pode mostrar-se que ocorre excesso quando o último bit de transporte ( $c_n$ ) é diferente do penúltimo ( $c_{n-1}$ ), tanto para a soma como para a subtração, pelo que o circuito de detecção de excesso pode ser acrescentado ao somador/subtractor da Fig. 2.50 na forma indicada pela Fig. 2.51.

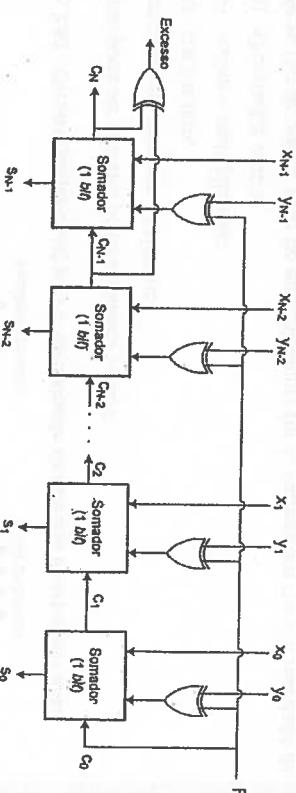


Fig. 2.51 - Circuito somador/subtractor com detecção de excesso (*overflow*)

Note-se que o facto de o último bit de transporte ( $c_n$ ) ser 1 não implica uma situação de excesso, como aliás é exemplificado pela Fig. 2.48b e Fig. 2.48c. Para haver excesso nesta situação, o penúltimo bit de transporte ( $c_{n-1}$ ) teria de ser diferente, ou 0, o que não acontece nestes dois casos. O facto de  $c_n=0$  também não quer dizer que não há excesso.

### SIMULAÇÕES — SOMA E SUBTRAÇÃO

Esta simulação ilustra o funcionamento do somador/subtractor da Fig. 2.51, com 4 bits, tomando como base um somador elementar de 1 bit (Fig. 2.49a). Os aspectos cobertos incluem os seguintes:

- Verificação da tabela de verdade do somador elementar (Tabela 2.24);

No caso da subtração:

- Se os operandos têm sinal igual, não pode haver excesso;
- Se os operandos têm sinal diferente, ocorre excesso quando o sinal do resultado for diferente do do primeiro operando.

O resultado precisa de 8 bits para ser representado, para permitir toda a gama de valores dos operandos. Dado que só há dois valores possíveis em cada bit, pode interpretar-se a multiplicação como a soma de uma série de termos, constituídos pelo multiplicando sucessivamente deslocado de uma posição para a esquerda, soma essa que só ocorre nas posições em que o bit do multiplicador é 1 (mas o deslocamento ocorre sempre).

$$\begin{array}{r}
 & \begin{array}{r} 1110 \\ \times 1011 \\ \hline \end{array} \\
 & \begin{array}{r} 1110 \\ 1110 \\ 0000 \\ 1110 \\ \hline 10011010 \end{array} \\
 & \text{multiplicador} \\
 & \text{produo}
 \end{array}$$

Fig. 2.52 - Multiplicação de números binários sem sinal

Uma alternativa (mais eficiente do ponto de vista de implementação com um circuito digital) é ver o resultado da multiplicação como uma série de somas parciais do resultado, sucessivamente deslocado para a direita. Ou seja, em vez de ir somando o multiplicando ao resultado em posições cada vez mais à esquerda, soma-se o multiplicando sempre na posição mais à esquerda do resultado e em cada soma desloca-se o resultado obtido para a direita. No fim da operação, o resultado ficará correcto. A Fig. 2.53 mostra um circuito adequado para este efeito, incluindo dois registos de deslocamento com carregamento paralelo, P e M, semelhantes ao registo da Fig. 2.37, na página 74. Cada um destes registos deve ter N bits (número de bits de cada operando, neste caso 4).

Este circuito inclui ainda um somador, como o da Fig. 2.49c, uma báscula D, como a da Fig. 2.21b, na página 60, para memorizar o transporte gerado pelo somador, um contador de contagem decrescente com detecção de passagem por zero, como o da Fig. 2.35a, na página 73, e um circuito de controlo, uma máquina de estados que poderá ser sintetizada (secção 2.6.7.3) ou microprogramada (secção 2.6.7.4) e que é responsável pelo sequenciamento das várias operações a executar para calcular o produto, gerando os diversos sinais

- Verificação do comportamento do somador em complemento para 2 (com números positivos e negativos);
- Utilização do somador como subtractor, em complemento para 2 (com números positivos e negativos);
- Detecção da condição de excesso;
- Simulação passo a passo para verificação da operação em sequência (*ripple*) e da soma dos tempos de atraso (secção 2.8.1).

### 2.8.4 MULTIPLICAÇÃO DE DOIS NÚMEROS BINÁRIOS

Estas operações são mais complexas do que a soma e subtração, em particular quando se pretende contemplar números negativos. A Fig. 2.52 ilustra um exemplo simples de multiplicação de dois números binários sem sinal de 4 bits, usando o clássico método papel-e-lápis que todos aprendemos na escola.

O resultado precisa de 8 bits para ser representado, para permitir toda a gama de valores dos operandos. Dado que só há dois valores possíveis em cada bit, pode interpretar-se a multiplicação como a soma de uma série de termos, constituídos pelo multiplicando sucessivamente deslocado de uma posição para a esquerda, soma essa que só ocorre nas posições em que o bit do multiplicador é 1 (mas o deslocamento ocorre sempre).

necessários para controlar os recursos do multiplicador. Por simplicidade e clareza, este circuito não inclui todos os detalhes (como por exemplo a inicialização do registo P a zero).

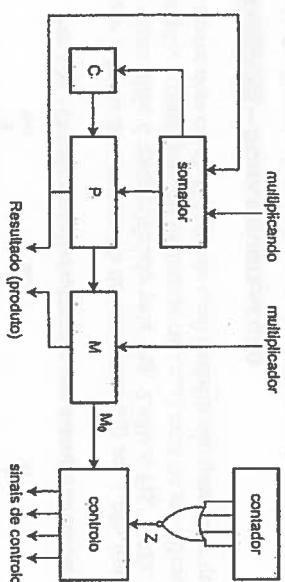


Fig. 2.53 - Circuito (simplificado) para a multiplicação de números binários sem sinal

O algoritmo pode ser descrito pelos seguintes passos:

1. Inicializa os recursos do sistema:
  - a) C e P a zero;
  - b) M com o multiplicador;
  - c) O contador com N.
2. Se  $M_0$  (bit de menor peso do multiplicador) for 1, carrega em C e P o resultado da soma do valor anterior de P com o multiplicando. Se  $M_0=0$ , não faz nada neste passo;
3. Desloca todo o conjunto C, P e M de um bit para a direita (em C entra 0 e o  $M_0$  antes de deslocar perde-se);
4. Decrementa o contador. Se o seu valor já for zero, termina o algoritmo. Se ainda for maior que zero, volta para o passo 2;
5. O algoritmo terminou. O resultado final (produto) está disponível no conjunto dos dois registos P e M.

A Tabela 2.25 mostra a evolução dos registos após cada um dos passos indicados, ao longo do algoritmo e das várias iterações no caso do exemplo da Fig. 2.52 (multiplicando=110, multiplicador=101, N=4). Note-se que na iteração 3 a soma não é feita, pois  $M_0=0$  (mas o deslocamento a direita é sempre feito). O produto fica nos registos P e M e é igual ao produto da Fig. 2.52, como não podia deixar de ser.

Este algoritmo não funciona correctamente se quisermos considerar os operandos como números em complemento para 2 (potencialmente negativos). Consultando a Tabela 2.22, verifica-se que o multiplicando na Fig. 2.52 (110) em complemento para 2 (com 4 bits) corresponde ao número -2, enquanto o multiplicador (101) é o número -5 (com 4 bits). O seu produto deveria ser +10 (com 8 bits), mas em vez disso na Fig. 2.52 obteve-se o valor de -102 (1001 1010, com 8 bits). Nitidamente, está errado.

O problema básico está nas somas parciais, em que implicitamente se assumiu que os termos estavam a ser estendidos para 8 bits com 0s à esquerda (ou seja, que eram positivos). Mas o facto é que, com 4 bits, o multiplicando é negativo. Note-se que o resultado tem 8 bits e os operandos da soma têm de ser estendidos para 8 bits também. Mas simplesmente estender os bits do multiplicando com o seu bit de sinal também só funciona se o multiplicador for positivo.

ALGORITMO		REGISTOS DO CONTROLO			ENTRADAS DO CONTROLO
ITERAÇÃO	PASSO	C	P	M	
—	1	0 0000	1011	0100	1 0
1	2	Soma	0 1110	1011	0100 1 0
1	3	Desloca	0 0111	0101	0100 1 0
1	4	Decrementa	0 0111	0101	0011 1 0
2	2	Soma	1 0101	0101	0011 1 0
2	3	Desloca	0 1010	1010	0011 0 0
2	4	Decrementa	0 1010	1010	0010 0 0
3	2	NÃO soma	0 1010	1010	0010 0 0
3	3	Desloca	0 0101	0101	0010 1 0
3	4	Decrementa	0 0101	0101	0001 1 0
4	2	Soma	1 0011	0101	0001 1 0
4	3	Desloca	0 1001	1010	0001 0 0
4	4	Decrementa	0 1001	1010	0000 1 1
5	5	Termina	0 1001	1010	0000 0 1

Tabela 2.25 - Evolução dos registos do circuito de multiplicação no exemplo da Fig. 2.52, em que o multiplicador é 110. O resultado final (produto) está nos registos P e M.

Existem algoritmos que permitem resolver os vários problemas que se levantam, nomeadamente o algoritmo de Booth, mas dado o carácter introdutório deste livro e por limitações de espaço este tema não é mais detalhado aqui. Recomenda-se [Hamacher 2002], [Farhat 2003] ou [Stallings 2006] para um tratamento mais desenvolvido deste tópico.

## 2.8.5 DIVISÃO DE DOIS NÚMEROS BINÁRIOS

Tal como na multiplicação, o algoritmo básico de divisão binária contempla apenas números sem sinal. As referências indicadas no fim da secção anterior apresentam soluções para números com sinal, em complemento para 2. Este algoritmo baseia-se no método papel-e-lápis que usamos para efectuar divisões manualmente, com a simplificação de que, em cada passo da iteração, o algoritmo do divisor pode apenas ser 1 ou 0. Ou seja, o divisor ou cabe ou não cabe (respectivamente) na parte do dividendo que estamos

a considerar nessa iteração. Se couber, é só fazer a subtração para achar o resto parcial dessa iteração, e baixar o bit seguinte do dividendo, para uma nova iteração.

A Fig. 2.54 ilustra este método, usando os valores da Fig. 2.52, para prova da multiplicação pela operação inversa (divisão), em que se acrescentou apenas 0100 (4) ao produto (que desempenha agora aqui o papel de dividendo) para que o resto da divisão não seja zero. Tal como seria de esperar, o quociente obtido é o multiplicando do exemplo da Fig. 2.52, com o resto igual ao valor adicional introduzido (0100), uma vez que a divisão como operação inversa da multiplicação dá resto zero.

dividendo	1001110	divisor	1011
	-1011		
	1110	quociente	
	-1011		
	01101		
	-1011		
	00100	resto	

Fig. 2.54 - Divisão de números binários sem sinal

A Fig. 2.55 mostra o circuito que se pode usar para implementar este algoritmo num sistema digital. É semelhante ao usado para a multiplicação, mas agora usa-se um subtrator (de números sem sinal) e o deslocamento é feito para a esquerda. O objectivo é começar com o dividendo no registo Q e em cada iteração ir deslocando os seus bits para o registo R e retirando-lhe o valor do divisor. O quociente é construído bit a bit, a partir do lado direito do registo Q, no espaço deixado livre pelo dividendo (evitando usar um terceiro registo). No final, o registo R conterá o resto da divisão e o registo Q o quociente. A saída do subtrator para o circuito de controlo é o bit de sinal da subtração, permitindo indicar se o divisor cabe ou não dentro da parte do dividendo contida no registo R.

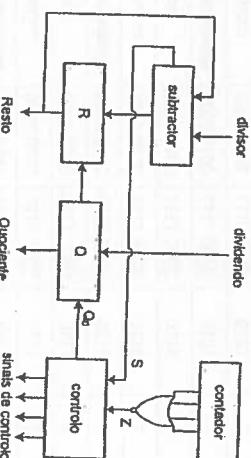


Fig. 2.55 - Circuito (simplificado) para a divisão de números binários sem sinal

O algoritmo pode ser descrito pelos seguintes passos:

1. Inicializa os recursos do sistema:

- a) R a zero;
- b) Q com o dividendo;
- c) O contador com N (número de bits do dividendo).

A Tabela 2.26 mostra a evolução dos registos após cada um dos passos indicados, ao longo do algoritmo e das várias iterações no caso do exemplo da Fig. 2.54 (dividendo=1001110, divisor=1011, N=8).

Neste exemplo o registo R não usa os 8 bits, mas se o divisor fosse por exemplo igual ao dividendo menos 1 tal já sucederia. O quociente e resto obtidos são os indicados na Fig. 2.54, embora nesta tenham sido omitidos os 0s à esquerda.

#### ESSENCIAL

A forma mais simples de efectuar uma soma de dois números de  $N$  bits consiste em interligar  $N$  somadores elementares de 1 bit, passando o transporte de somador em somador (embora os tempos de atraso dos vários somadores se acumulem);

Usando representação em complemento para 2, um subtrator é simplesmente um somador com o simétrico do número.

Uma operação pode dar excesso (*overflow*) se o resultado não puder ser expresso correctamente com o número de bits disponível. Tipicamente, o efeito é o resultado 'dar a volta' e ficar com o sinal errado, algo que deve ser detectado pelos computadores.

A multiplicação é complexa, em particular se envolver números negativos. Uma forma simples de a implementar para números sem sinal (só positivos) é usar um circuito que vá deslocando e acumulando o multiplicando se o bit do multiplicador for 1 exactamente como se faz manualmente.

E preciso cuidado com o número de bits necessário para guardar o resultado de uma multiplicação, pois potencialmente precisa de tantos como o multiplicando e o multiplicador juntos.

A divisão é igualmente complexa, podendo no entanto ser feita de acordo com o algoritmo manual que as pessoas usam para o caso dos números sem sinal, o que permite calcular o quociente e o resto.

## 2.9 CONCLUSÕES

ALGORITMO	O QUE FAZ	R	Q	COT	S	Z	ENTRADAS DO CONTROLO
-	-	0000 0000	1001 1110	1000	1	0	
1	Initia	0000 0001	0011 1100	1000	1	0	
2	Desloca	0000 0001	0011 1100	1000	1	0	
3	Nada	0000 0001	0011 1100	1000	1	0	
4	Decrementa	0000 0001	0011 1100	0111	1	0	
2	Desloca	0000 0010	0111 1000	0111	1	0	
3	Nada	0000 0010	0111 1000	0111	1	0	
4	Decrementa	0000 0010	0111 1000	0110	1	0	
2	Desloca	0000 0100	1111 0000	0110	1	0	
3	Nada	0000 0100	1111 0000	0110	1	0	
4	Decrementa	0000 0100	1111 0000	0101	1	0	
2	Desloca	0000 1000	1110 0000	0101	1	0	
3	Nada	0000 1000	1110 0000	0101	1	0	
4	Decrementa	0000 1000	1110 0000	0100	1	0	
2	Desloca	0001 0011	1000 0000	0100	0	0	
3	Subtra	0000 1000	1100 0000	0100	1	0	
4	Decrementa	0000 1000	1100 0000	0101	1	0	
2	Desloca	0001 0011	1000 0000	0111	0	0	
6	Subtra	0000 0110	1000 0000	0111	1	0	
4	Decrementa	0000 0110	1000 0000	0110	1	0	
2	Desloca	0001 0010	0000 0000	0010	0	0	
7	Subtra	0000 0010	0000 0000	0010	1	0	
4	Decrementa	0000 0010	0000 0000	0010	1	0	
2	Desloca	0000 1101	0000 0000	0010	0	0	
7	Subtra	0000 0010	0000 0000	0010	1	0	
4	Decrementa	0000 0010	0000 0000	0001	1	0	
2	Desloca	0000 0100	0000 0000	0001	1	0	
3	Nada	0000 0100	0000 0000	0001	1	0	
8	Decrementa	0000 0100	0000 0000	0000	1	1	
5	Termina	0000 0100	0000 0000	0000	1	1	

Tabela 2.26 - Evolução dos registos do circuito de divisão no exemplo da Fig. 2.54, em que o divisor é 1011. O resultado final (resto e quociente) fica nos registos R e Q. O círculo mostra a entrada sucessiva do dividendo no registo Q.

Este algoritmo é extremamente inefficiente face ao uso em computadores, destinando-se apenas a ilustrar a operação. Os computadores comerciais têm unidades aritméticas que além de executarem as operações de forma muito mais rápida ainda contêm números negativos e com parte fraccionária, de forma a implementarem os números reais.

É impressionante como com apenas circuitos electrónicos simples se conseguem implementar funcionalidades interessantes, como por exemplo o semáforo com botão. Al deve-se essencialmente ao facto de se usarem circuitos digitais, com dois valores bem determinados, álgebra de Boole, que permite estabelecer regras de combinação de sinais, circuitos síncronos, que reagem de forma sincronizada com um relégio e permitem memorizar o estado do sistema e implementar máquinas de estados.

Estas funcionalidades seriam extremamente difíceis de implementar com circuitos analógicos e impossíveis com circuitos digitais combinatórios. Apenas os circuitos sequenciais conseguem esta capacidade. O que não quer dizer que não se recorra a circuitos combinatórios de grande utilização, como multiplexers ou descodificadores. Mas estes, só por si, não conseguem implementar uma máquina de estados.

Há circuitos sequenciais que também são muito utilizados, como os contadores. Mas a sua funcionalidade é fixa e portanto limitada. As máquinas de estado sintetizadas, com base num registo e num circuito combinatório projectado à medida, são muito mais flexíveis, mas o circuito resultante é complexo e difícil de alterar (apesar de haver programas que automatizam o processo).

As máquinas de estado microprogramadas recorrem a blocos já existentes (registos ou controladores, multiplexers, somadores, ROMs, etc.) para implementarem um circuito de hardware cuja funcionalidade pode ser largamente alterada em função do programa na ROM, sem modificações ao circuito. Estas máquinas de estado estão na base da unidade de controlo dos computadores.

Por outro lado, os computadores não foram feitos apenas para controlo de sinais físicos, mas também para processamento da informação, o que envolve essencialmente números, nem que seja para representar símbolos não numéricos (como por exemplo o alfabeto). Isto implica capacidade para representar números em forma binária, o que se consegue limitando N bits, em que nos computadores tipicamente N vale 8, 16, 32 ou 64. Quantos bits, maior a gama de números que é possível representar. Como um computador deve estar preparado para lidar com qualquer número dentro de uma dada gama de valores, todos os números representados têm de ter o mesmo número de bits que os maiores valores a representar, o que implica algum desperdício mas facilita a implementação. Se necessário, um número tem bits neutros à esquerda (0s se for positivo ou 1s se for negativo).

A melhor representação para os números binários inteiros é o complemento para 2, em que um número positivo é representado tal e qual como em representação sem sinal (só para números positivos), mas sempre com o bit de maior peso a 0, e os números negativos são representados com o bit de maior peso a 1 e pela ordem inversa dos positivos (isto é o valor numérico em representação sem sinal, todos os bits a 1, é em complemento para 2 directamente o de menor módulo, ou -1). A grande vantagem deste esquema é que permite tratar em larga medida os números positivos e negativos da mesma forma, sem

tratamento especial para cada um deles. Nomeadamente, a subtração de dois números igual à soma do primeiro com o simétrico do segundo.

Os números binários não inteiros (que representam os números reais no formato de vírgula flutuante) têm um tratamento mais complexo, que é referido no Apêndice D, na página 735.

## 2.10 EXERCÍCIOS

2.1 Indique o significado da sequência de bits 10111011 quando interpretada como:

- Carácter ASCII (considerando apenas os sete bits de menor peso);
- Número binário sem sinal;
- Número binário em complemento para 2.

2.2 Suponha que esta sequência é armazenada num registo de 16 bits, para o que precisa de ser estendida. Para cada uma das alíneas anteriores, qual o valor (em hexadecimal) que esse registo deve ter para que o valor interpretado não se altere?

2.3 Faça a tabela de verdade de cada uma das expressões seguintes:

- $A B + \bar{A} \bar{B} + \bar{A} B$
- $A + \bar{B} C + \bar{A} B$
- $\bar{A} B \bar{C} D + \bar{A} \bar{B} \bar{C} D + A \bar{B} \bar{C}$
- $\bar{A} B + A C + B C$
- $A B C \bar{D} + A B \bar{C} \bar{D} + A \bar{B} \bar{C} \bar{D}$
- $\bar{A} \bar{B} \bar{C} \bar{D} + \bar{A} B \bar{C} \bar{D} + A \bar{B} C \bar{D}$

2.4 Use mapas de Karnaugh para simplificar as expressões do exercício anterior.

2.5 Elabore a tabela de verdade de um *multiplexer* de 1-de-8 e sintetize o respectivo circuito, tal como foi feito na Fig. 2.11, na página 49. Simplifique com mapas de Karnaugh.

2.6 O código Gray permite contar apenas com 0s e 1s, mas mudando apenas um bit de cada vez, ao contrário do que sucede no código binário. Com este esquema conseguem-se evitar problemas decorrentes dos tempos de propagação dos sinais ao longo do circuito (porque só um muda). A Tabela 2.27 ilustra este código para os primeiros oito números. Com base nessa tabela sintetize o circuito de um conversor, que receba 3 bits em código binário e produza o valor correspondente em código Gray (use mapas de Karnaugh para simplificar as expressões).

Código binário	Código Gray
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Tabela 2.27 - Contagem em binário e em código Gray

2.7 Desenhe um circuito constituído por um AND de duas entradas, com um NOT inserido numa delas. Suponha que estas duas portas lógicas têm um tempo de atraso de atraso de 10 (a unidade de tempo não é relevante). Faça um diagrama temporal da saída do circuito, assumindo que as duas entradas do circuito (no AND e no NOT) passam ao mesmo tempo de 0 para 1, 50 unidades de tempo a entrada do AND volta a 0. À luz deste exercício explique a vantagem do código Gray.

2.8 Com base na tabela de verdade de um descodificador de 1-de-8, na página 53, sintetize o respectivo circuito, usando os mapas de Karnaugh para simplificar as expressões.

2.9 Represente cada um dos números decimais -2, +130, -100, +12, -128, +5, -1024, +255, -15 em hexadecimal e complemento para 2, com 4, 8 e 16 bits (indique os casos em que não for possível).

2.10 Indique qual o maior e o menor número inteiro representável com 4, 8 e 16 bits, em representação:

- Sen sinal;
- De módulo e sinal;
- Em complemento para 2.

2.11 Calcule o valor, em decimal, dos seguintes números representados em hexadecimal, complemento para 2 e com o mínimo de dígitos necessário: FFH, 8FH, 1C0H, FFFFH, 1000H, FEH, FFFFFFFFH, 8000H, 8H, 7FH.

2.12 Quantos bits precisa, no mínimo, para representar o número decimal 3.456.789? Mostre como é que pode responder sem converter o número para binário.

2.13 Mostre que a soma de um número binário com N bits com o seu complemento para 2 dá sempre zero, considerando a soma também com N bits. Pista: Some em binário um número qualquer de 4 bits com as parcelas necessárias para o converter para complemento para 2.

2.14 Converte os números decimais para representação binária em complemento para 2 com 8 bits e efectue as operações em binário (converte subtrações em somas), indicando quais produzem uma situação de excesso.

- 32 + 16
- 64 + 70
- 53 - 75
- 50 - 128
- 34 - 15
- 14 \* 7
- 16 \* (-8)
- 8 \* (23 - 40)
- 3 \* 20 + 100

2.15 Fazendo aritmética de 16 bits, qual o número com maior módulo que é possível multiplicar por FFFFH sem dar excesso? Porque?

2.16 Calcule, em decimal, quantos bits têm 12 KBytes.

2.17 Use o Apêndice E, na página 743, para conseguir ler o seguinte texto: 01000010 011011101101010010000001100100 01101001011000010010001.

- 2.18** Utilize *multiplexers* de 2 e 4 entradas para implementar um *multiplexer* de 8 entradas. Simule e teste o circuito resultante.

**2.19** Refaca o circuito da Fig. 2.51, mantendo a mesma funcionalidade mas assumindo que não tem portas XOR. Pode recorrer a outros blocos combinatórios. Simule e teste o circuito resultante, com 4 bits.

- 2.20** Modifique o contador da Fig. 2.32 para contar para trás, de acordo com o referido na página 71, simule o circuito e verifique o seu funcionamento.
- 2.21** Usando uma ROM, um contador a contar para trás (exercício 2.20) com um relógio de tempo real de 1 Hz e um mostrador de 7 segmentos, implemente um circuito que vá mostrando (de segundo em segundo) cada letra do seu apelido (use a combinação de segmentos que melhor reproduz cada letra, maiúscula ou minúscula), em ciclo infinito. Simule este circuito e verifique o seu funcionamento.

- 2.22** Simplifique a tabela de verdade do descodificador de sete segmentos (Tabela 2.6 na página 52), implemente o respectivo circuito no simulador e verifique o seu funcionamento.
- 2.23** Implemente a variante ao semáforo simples referida na página 79, por inclusão de símbolos que o mostrador mostra são os da Fig. 2.13.

- 2.24** Projecte, sintetize (com basculas e portas lógicas) e simule uma máquina de estados que reconheça a sequência 0011 numa variável de entrada e produza um 1 num semáforo de peões. Simule o circuito e verifique o seu funcionamento.

- 2.25** Idem, mas usando uma máquina de estados microprogramada.

- 2.26** Projecte, sintetize (com basculas e portas lógicas) e simule uma máquina de estados que, dependendo do valor (0 ou 1) de uma entrada A reconheça a sequência 0011 ou 0010 numa variável de entrada B e produza um 1 numa variável de saída durante o último bit da sequência.

- 2.27** Idem, mas usando uma máquina de estados microprogramada.

- 2.28** Projecte, sintetize (com basculas e outros módulos que entende) e simule um circuito microprogramado que conte o número de bits a 1 de um valor memorizado num registo.

- 2.29** Projecte e implemente um alarme residencial, usando um interruptor, dois sensores e uma sirene, todos activos a 1. Um dos sensores é imediato (a sirene toca imediatamente se o sensor for activado), enquanto o outro é temporizado (espera x segundos antes de tocar a sirene) para permitir desactivar o alarme quando se entra em casa. O interruptor (em local escondido) é de efeito imediato para desactivar o alarme e temporizado (y segundos) para o activar (para permitir sair de casa). O toque de sirene dura z segundos, após o que o alarme volta a ser armado (estado normal após ser activado). Use microprogramação e teste o sistema no simulador (com um relógio de tempo real de 1 Hz), usando um LED para simular a sirene. Use outro LED para saber quando é que o alarme está activado.

### 3 ■ O MEU PRIMEIRO COMPUTADOR

Com este título intimista pretende-se dar uma primeira noção dos vários aspectos que envolvem a estrutura interna e o uso de um computador, de modo a constituir uma base para nos capítulos seguintes se aprofundar os conhecimentos nesta área.

No capítulo anterior vimos como era possível construir pequenos sistemas de controlo usando simplesmente circuitos específicos, sem programação, ou programáveis, mas controlando directamente todos os sinais internos (microprogramação) e com pouca flexibilidade para alterações. Um computador tem de ser muito mais geral que isto. Tem de possuir uma interface (monitor e teclado, tipicamente) a partir da qual possa ser programável de forma facilmente alterável e sem necessidade de modificação dos seus circuitos físicos, para além de ser capaz de executar programas muito complexos.

Este capítulo descreve a estrutura básica de um computador, usando os circuitos básicos descritos no capítulo anterior, e dá uma primeira noção de como ele pode ser programado. Porém, em vez de simplesmente apresentar uma dada arquitetura, este capítulo segue uma aproximação racional e evolutiva, em que se parte dos requisitos e do modelo básico de computação para, passo a passo, derivar a arquitetura de um computador que cumpre esses requisitos, nas três vertentes fundamentais: processador, memória e periféricos. Estes módulos serão aprofundados em capítulos subsequentes. O objectivo aqui é fazer uma introdução ao funcionamento do conjunto que proporciona um mapa-guia que suporte a aprendizagem sectorial sem perder a visão global do que é um computador.

Este capítulo introduz a arquitetura usada ao longo deste livro, mas de forma simplificada e apenas no reino dos números de 8 bits. O microprocessador resultante das várias secções que o constroem, passo a passo, designa-se PEPE-8 (Processador Especial Para Ensino com 8 bits). Embora só no capítulo 4 seja introduzido o microprocessador definitivo, o PEPE (com 16 bits), o PEPE-8 é já um microprocessador completo e mesmo realizável fisicamente, o que permite introduzir exemplos concretos de programação em linguagem assembly, a forma de mais baixo nível de programar um computador.

Aliás, muitos dos microprocessadores do mercado são de 8 bits e são capazes de executar muitas funções, pelo que a estratégia incremental deste livro de apresentar primeiro uma arquitetura mais simples permite ilustrar a sua funcionalidade mas também as suas limitações e melhor justificar a introdução posterior de um microprocessador de 16 bits.

O capítulo termina com uma breve comparação entre as várias soluções de implementação de sistemas digitais, na perspectiva de que soluções específicas já só se justificam em casos muito particulares e que na esmagadora maioria dos casos se deve adoptar a solução mais genérica que se conhece: um computador com um programa adequado.

### 3.1 COMPONENTES BÁSICOS DE UM COMPUTADOR

A secção 2.6.7.3 mostrou que é possível construir pequenos sistemas de controlo desenvolvendo circuitos específicos, que variam de sistema para sistema. Isto é, sempre que algo muda, tem de se voltar a desenvolver o circuito.

A secção 2.6.7.4 ilustrou que com uma técnica razavelmente simples (microprogramação) era possível mudar o comportamento do sistema através da reprogramação da memória (ROM). O hardware deixa assim de ser específico para passar a ser genérico. Mudar a aplicação não implica alterar as ligações do circuito, mas apenas o conteúdo da ROM, o que se pode fazer trocando a ROM (por outra com o novo conteúdo) ou reprogramando a já existente com um programador adequado.<sup>14</sup>

A microprogramação em si já é uma forma de programação, mas cujo objectivo fundamental é produzir sinais de controlo para cada um dos estados possíveis do sistema. Não inclui nenhum mecanismo para fazer processamento de informação, como algo de tão básico como efectuar uma simples soma de dois números. A microprogramação é essencialmente uma forma de controlo, não de processamento.

Um computador é já uma máquina completa, pois inclui estas duas perspectivas. A Figura 3.1 recorda os três tipos de componentes básicos de um computador:

- Processador – Coordena tudo e executa as operações definidas pelo programa e inclui duas unidades fundamentais:
    - Unidade de Dados – Faz o processamento, executando as operações (soma, subtração, etc.) sobre os dados;
    - Unidade de Controlo – Interpreta as instruções do programa e controla tudo (não apenas a unidade de dados mas também os acessos às memórias e aos periféricos). Sucede frequentemente que a unidade de controlo utiliza microprogramação para definir as suas operações básicas. A secção 7.2.4, na página 585, elabora este assunto.
  - Memória:
    - De Instruções – Armazena as instruções do programa a executar;
    - De Dados – Armazena não apenas os dados de entrada do programa mas também os resultados de saída e mesmo os resultados parciais, intermédios.
    - Periféricos – Permitem efectuar a entrada e saída de informação no computador.
- O objectivo deste capítulo é fazer uma introdução a cada um destes componentes.

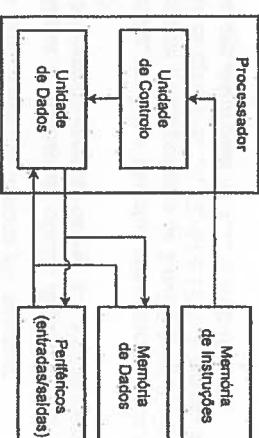


Fig. 3.1 - Arquitectura básica de um computador

### 3.2 RAM – A MEMÓRIA PARA GUARDAR INFORMAÇÃO

A secção 2.5.4 descreveu o que é uma ROM (*Read Only Memory*). A sua grande vantagem é ser não volátil, isto é, o seu conteúdo não se perde se o sistema for desligado. Quando se ligar de novo o sistema, o seu conteúdo estará lá preservado, célula a célula.

A desvantagem das ROMs é não poderem ser escritas<sup>15</sup>, o que as torna inadequadas para:

- Memória de instruções – Uma ROM até pode ser adequada para conter o programa, uma vez que para executar um programa o processador só precisa de ler as instruções, e com a vantagem adicional de o programa ser não volátil. Mas isto apenas nos computadores que têm o programa fixo. Naqueles em que se pode correr vários programas (um PC, por exemplo), a memória de programa tem de poder ser escrita frequentemente, de cada vez que se muda o programa;
- Memória de dados – Em qualquer programa, a escrita de valores na memória é muito frequente, seja para armazenar resultados de um programa, seja para guardar valores intermédios ao calcular uma expressão.

Surge assim a necessidade de outro tipo de memórias, as RAMs (*Random Access Memory*), que podem ser escritas tantas vezes quantas as necessárias.

**NOTA** O termo "Random," (aleatório) indica apenas que em qualquer altura se pode aceder a qualquer das células da memória, especificando apenas o seu endereço (ao contrário de uma fita magnética, por exemplo, em que o acesso é sequencial). Desse ponto de vista, até as ROMs são RAMs. No entanto, por motivos históricos, chama-se RAM exclusivamente às memórias que podem ser escritas com uma grande frequência (para além de poderem também ser lidas, naturalmente).

Tal como as ROMs (secção 2.5.4, na página 54), as RAMs são constituídas por um conjunto linear de células de memória, seleccionadas por um endereço, que está ilustrado

<sup>14</sup> Algumas ROMs podem programar-se apenas uma vez e designam-se PROMs (*Programmable Read Only Memories*). Outras permitem apagar o seu conteúdo e ser reprogramadas, como EEPROMs (*Electrically Erasable PROMs*) e as PROMs Flash.

<sup>15</sup> Há ROMs programáveis, as PROMs (*Programmable Read Only Memories*), mas que só podem ser escritas uma única vez.

na Fig. 3.2 para 16 células. A grande diferença reside na existência de um sinal de escrita, WR, que quando é activado ('valor 0') permite escrever um valor numa das células de memória (a especificada por Endereço).

**Nota** O nome WR significa "Write", indicando que se trata de um sinal de escrita. Tipicamente, este sinal é activo a 0. Ou seja, se for 0 está activo e escreve, enquanto se for 1 está inactivo e le. Esta designação é tão comum que já é universalmente reconhecida, pelo que se usa aqui o termo em inglês e a mesma convenção (sinal activo, a escrever, quando tem o valor 0).

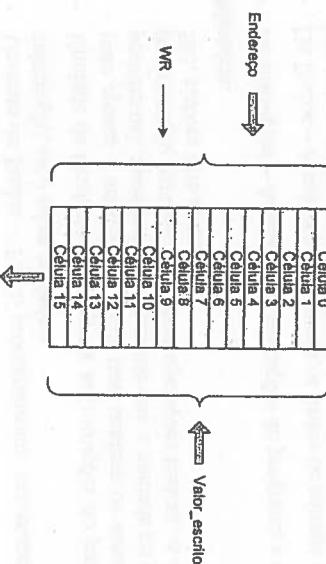


Fig. 3.2 - Esquema básico de uma RAM (*Random Access Memory*) com 16 células

**Nota**

Quando se representa graficamente uma tabela de endereços, como é o caso de uma memória, há duas possibilidades:

- Endereço 0 no topo e os restantes endereços de cima para baixo (como na Fig. 3.2). A grande vantagem deste esquema é ficar compatível com o facto de nós termos um texto de cima para baixo, e o mesmo sucede a um programa de computador (ver, por exemplo, o Programa 3.2 na página 137). As primeiras instruções (as que surgem primeiro numa página) são as que ocupam os primeiros endereços (os mais baixos). Em termos gráficos, tem a desvantagem de os endereços de valor mais alto aparecerem por baixo;
- Endereço 0 em baixo e os restantes endereços de baixo para cima. Graficamente parece menos contra-natura (os endereços mais baixos estão por baixo, os endereços mais altos estão mais alto), mas tem a desvantagem que fica ao contrário de qualquer tabela em que apareça um programa e os respectivos endereços.

Na realidade, tanto faz. O funcionamento de uma memória não é afectado pela forma como a representarmos. Por razões de consistência (evitando representar umas vezes duma forma, outras vezes de outra), este livro representa todos os conjuntos de endereços com os endereços crescentes de cima para baixo, para que fique compatível com o que sucede nas listagens dos programas.

OPERACAO	SINAIS DE ENTRADA DA RAM		SINAL DE SAIDA DA RAM	
	ENDEREÇO	WR	VALOR_ESCRITO	VALOR_LIDO
Leitura	Inativo		Especifica qual a célula acedida	Conteúdo da célula lida
Escrita	Activo			Irrelevante

Tabela 3.1 - Operações básicas suportadas por uma RAM

**Nota** A operação de escrita numa célula não afecta o valor das outras células. Só uma célula pode ser escrita ou lida de cada vez. Quando a operação de escrita acaba (quando WR passa de activo para inactivo), a RAM passa para modo de leitura e o Valor\_lido passa a exhibir o valor que acabou de ser escrito, desde que não se altere o valor de Endereço.

Ao contrário das ROMs, a RAM é um dispositivo de memória volátil. Se a alimentação do sistema for desligada, o conteúdo de todas as células perde-se. Quando a alimentação for de novo ligada, cada célula terá um valor aleatório ("lixo") e não o último a ser escrito nessa célula. Por esta razão, qualquer programa deve sempre inicializar (escrever um valor conhecido) todas as células de memória com que pretenda trabalhar.

Há RAMs não voláteis, mas tal não é devido à tecnologia da RAM em si. Simplesmente, usam uma pequena bateria de longa duração (de litio, tipicamente) que mantém a RAM alimentada mesmo com sistema desligado. Um circuito especial consegue fazer a gestão entre as duas alimentações (bateria e sistema) de modo a evitar conflitos e a manter a RAM sempre alimentada.

Hi alguns conceitos básicos numa RAM que importa não esquecer:

- Capacidade em células – Número de células disponíveis na RAM. É sempre uma potência de 2. No caso do exemplo da Fig. 3.2, a capacidade é de 16 células, numeradas de 0 a 15;
- Largura de cada célula, em bits – Quantos bits consegue cada célula armazenar de cada vez. Tipicamente é uma potência de 2 múltipla de 8, ou seja, um número inteiro de bytes (valores típicos: 8, 16, 32, 64 e 128). A largura da célula também é designada palavra, que assim pode ter 1, 2, 4, 8 ou 16 bytes;

- Capacidade em bytes – Produto da capacidade em células pela largura da célula em bytes. Se na Fig. 3.2 cada célula tiver 16 bits (2 bytes), a capacidade da RAM será de 32 bytes;

**Endereço** – Número da célula a que se pretende aceder, começando em 0 e terminando em  $N-1$ , em que  $N$  é a capacidade (em células) da RAM. O número de bits necessário para o endereço depende de  $N$ . Na Fig. 3.2 são necessários 4 bits para poder representar os 16 endereços diferentes (0 a 15). Se a capacidade fosse de 256 células, seriam necessários 8 bits (endereços 0 a 255). No caso geral, um endereço necessita de um número de bits igual ao logaritmo de base 2 da capacidade (em células) da RAM;

**NOTA** Para um melhor aproveitamento da memória, os computadores permitem normalmente aceder a cada byte de uma RAM individualmente (endereçamento por byte) e não apenas célula a célula (endereçamento por palavra). Ou seja, é possível escrever um determinado byte numa determinada célula sem alterar o valor dos restantes bytes dessa mesma célula. Como cada palavra pode ter vários bytes, no endereçamento por bytes são necessários mais bits (mais endereços possíveis). A secção 4.4 detalha este aspecto.

Se imaginarmos uma RAM com 32 Mega<sup>16</sup> células com 64 bits cada uma, por exemplo,

sua capacidade em bytes será 256 Megabytes. Como cada célula tem 8 bytes, então  $256 \text{ Megabytes} = 32 \text{ Mega} * 8 \text{ bytes}$ .

**NOTA** A largura de uma RAM é feita de modo a ser igual ao número de bits de memória que esse valor é 64. Outros valores típicos são 128, 32, 16 e 8.

A capacidade de uma RAM em células e a largura em bits de cada célula são factores independentes, podendo conceber-se casos extremos como uma RAM só com uma célula com muitos bits de largura ou uma RAM com uma capacidade de muitas células com apenas 1 bit cada. Na prática, as RAMs tendem a ter uma largura múltipla de 8 bits (8, 16, 32, etc.) e uma capacidade que é uma potência de 2, normalmente bastante grande.

**NOTA** Normalmente, cada circuito integrado de RAM tem 8 bits de largura. RAMs de maior largura constroem-se colocando vários circuitos integrados de RAM ao lado uns dos outros, razão pela qual os módulos de memória de um PC (e de outros computadores) têm vários circuitos integrados.

Voltando à Fig. 3.1, a utilização de RAMs implica que, para além das ligações de informacão indicadas nesta figura, o processador disponibiliza o endereço da célula a que pretende aceder, quer para a memória de instruções quer para a memória de dados. Este aspecto ficará mais claro ao longo das secções seguintes.

<sup>16</sup>  $\text{Mega} = 2^{20} = 1024 * 1024$

### SIMULAÇÃO 3.1 – UTILIZAÇÃO DE RAMS

Esta simulação ilustra a utilização de RAMs. Os aspectos cobertos incluem os seguintes:

- Visualização do conteúdo inicial (aleatório) de uma RAM;
- Leitura de células específicas dando o seu endereço;
- Verificação da capacidade de uma RAM e relação com o endereço;
- Escrita de células específicas dando o seu endereço, com actuação do sinal WR;
- Leitura de uma célula após a escrita (por desactivação de WR);
- Verificação da volatilidade da RAM (simulação do desligar da alimentação).

#### ESSENCIAL

▪ Uma RAM é um conjunto de células de memória independentes, acessíveis individual e aleatoriamente (sem ordem predeterminada) através do seu índice (conhecido por endereço) no conjunto;

▪ A capacidade de uma RAM mede-se em células ou, mais vulgarmente, em bytes;

▪ A largura de uma RAM (palavra) é o número de bits de cada célula (tipicamente, uma potência de 2);

▪ A célula é a unidade de leitura ou escrita numa RAM (todos os bits de uma célula são lidos ou escritos ao mesmo tempo);

▪ Sozinha, uma célula pode ser acedida de cada vez. Na escrita, apenas uma das restantes mantém o seu valor;

▪ Ao contrário das ROMs, as RAMs são voláteis. O conteúdo das suas células perde-se se a alimentação for desligada;

▪ Não é possível endereçar uma célula com o seu conteúdo. Por exemplo, uma célula pode ter o endereço 3 e conter o valor 34;

### 3.3 O PROCESSADOR (PEPE-8)

Esta secção parte do modelo básico da Fig. 3.1 e dos requisitos para as operações básicas de um computador para determinar qual a arquitetura de um processador didáctico, muito simples mas que consiga executar essas operações. Este processador foi baptizado de PEPE-8 (Processador Especial Para Ensino com 8 bits) porque os dados que consegue manipular directamente são números binários de 8 bits (ver página 124).

Na Fig. 3.1, a unidade de dados é a responsável por efectuar as operações sobre os dados, que são lidos da memória de dados, processá-los e produzir os resultados que são de novo armazenados na memória de dados.

A unidade de controlo coordena os vários recursos do sistema e impõe o sequenciamento das várias operações necessárias.

Uma das operações mais básicas que um computador deve poder fazer é uma soma de dois operandos, produzindo um resultado. Afinal, os pioneiros da computação foram simples dispositivos de cálculo, como o ábaco e o primeiro somador mecânico, a Pascaline de Blaise Pascal (secção 1.8).

Olhando para a Fig. 3.1, os passos envolvidos numa simples soma deverão ser os indicados no seguinte algoritmo (assumindo que os operandos, que são dados, estão já armazenados na memória de dados):

#### Algoritmo de soma (versão 1):

1. O processador lê os operandos da memória de dados;
2. A unidade de dados do processador inclui um somador (secção 2.8.1), que faz a soma e produz um resultado;
3. O resultado é armazenado na memória de dados (podendo servir de operando numa eventual soma a fazer a seguir).

Até nem parece complicado, mas há uma série de problemas que é preciso resolver e que conduzem à arquitectura básica de um processador. Estes problemas e as respectivas soluções são descritos nas secções seguintes.

### 3.3.1 UNIDADE DE DADOS

#### 3.3.1.1 REGISTRO NA UNIDADE DE DADOS

**PROBLEMA** O somador da unidade de dados da Fig. 3.1 é um circuito combinatório, pelo que precisa dos dois operandos simultaneamente. Como é que se podem ler dois valores ao mesmo tempo da memória de dados, que só consegue aceder a uma célula de cada vez?

**SOLUÇÃO** Decompor a soma em duas operações:

1. Lê um operando para um registo (secção 2.6.2) interno da unidade de dados;
2. Faz a soma entre o primeiro operando (que é lido do registo) e o segundo operando (que é lido da memória).

A Fig. 3.3 ilustra esta solução, em que o sinal ESCR\_A é o que permite escrever no registo A. Este sinal deve ser activado apenas durante a primeira operação. Naturalmente, o sinal WR, que permite escrever na memória, deve estar inactivo em ambas as operações.

Por outro lado, há a questão de onde guardar o resultado do somador. Se não for memorizado, perder-se-á. Estava previsto no algoritmo da soma guardá-lo na memória de dados (na Fig. 3.3, seria só ligar a saída do somador à entrada da memória de dados), mas há outro problema:

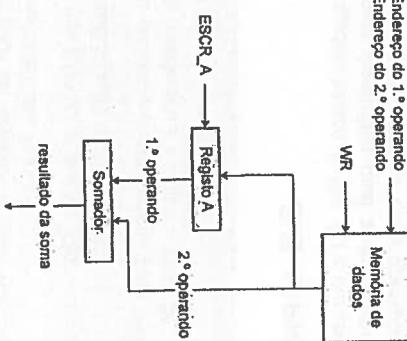


Fig. 3.3 - O sinal ESCR\_A memoriza no registo A o primeiro operando lido da memória

**SOLUÇÃO** Guardar o resultado da soma no próprio registo A, tal como ilustrado pela Fig. 3.4. Este registo contém apenas uma cópia do 1.º operando e armazenar lá o resultado da soma não destrói o 1.º operando da soma. O registo A permite memorizar um valor produzido com base no próprio valor da sua saída (se não for um trinco – ver secção 2.6.2), activando o sinal ESCR\_A (ativo a 1). Embora não representado por simplicidade, assume-se que o registo liga também a um sinal de relógio e só quando este muda, com ESCR\_A activo, é que a escrita é efectuada.

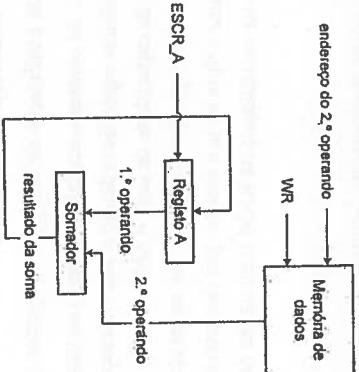


Fig. 3.4 - Escrita no registo A do resultado de uma soma, activando o sinal ESCR\_A

**PROBLEMA** O resultado da soma não pode ser guardado na memória de dados, pois esta está ocupada a ler o valor do 2.º operando. Não é possível escrever o resultado num endereço diferente (se o endereço mudasse, também o 2.º operando mudaria). Destruir o valor do 2.º operando (com o resultado) pode não ser aceitável.



3-O MEU PRIMEIRO COMPUTADOR

**NOTA**

a 0. O facto de um sinal ser activo a 0 ou a 1 tem às vezes razões físicas (tecnologia electrónica, que não é objecto de estudo neste livro), mas mais frequentemente é uma questão de convenção.

O sinal WR é um sinal exterior do PEPE-8 e tem de ligar às memórias e outros dispositivos, o que significa que tem de respeitar as convenções de outros circuitos já existentes. Mas ESCR\_A é um sinal de controlo interno, o que por um lado se traduz em circuitos electrónicos diferentes e por outro significa que pode usar as suas próprias convenções. Assim, por razões de simplicidade convenciona-se que todos os sinais internos do PEPE-8 são activos a 1.

**PROBLEMA** É necessário conjugar as soluções 1 e 2, mas a entrada do registo A não pode ligar ao mesmo tempo à saída da memória e à saída do somador.

**SOLUÇÃO** Ligar um *multiplexer* à entrada do registo A, como ilustrado na Fig. 3.5. Desta forma, é possível seleccionar qual a fonte onde obter o valor a escrever no registo A (ou a saída do somador ou a memória de dados).

Por outro lado, um programa não terá com certeza uma única operação de soma, pelo que é expectável que um programa lide com várias operações, operandos e resultados.

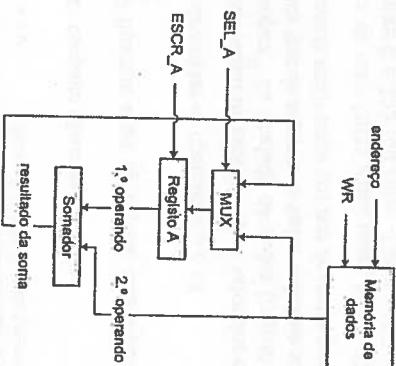


Fig. 3.5 - A utilização de um *multiplexer* permite escolher, de acordo com o sinal SEL\_A, a fonte do valor a escrever no registo A: ou o resultado de uma soma ou um valor lido da memória de dados.

**PROBLEMA** O registo A só pode conter um resultado de uma operação de cada vez, pelo que tem que ser possível, mais tarde ou mais cedo, guardar o valor do registo A na memória de dados. É concebível que em alguns casos um resultado sirva de operando na operação seguinte, mas não em todos os casos.

**SOLUÇÃO** Ligar a saída do registo A à entrada da memória de dados, como ilustrado na Fig. 3.6. Desta forma, é possível copiar o conteúdo do registo A para uma célula de memória, especificando qual o seu endereço e activando o sinal WR.

O circuito da Fig. 3.6 já permite fazer algumas operações básicas e elaborar uma nova versão do algoritmo de soma, desta vez já de uma forma implementável por este circuito.

#### Algoritmo de soma (versão 2):

1. O processador lê um operando da memória de dados para o registo A;
2. O processador lê o outro operando da memória de dados, soma-o com o registo A e armazena o resultado no registo A (podendo servir de operando numa eventual soma a fazer a seguir, caso em que o passo 1 dessa soma seguinte já estaria feito);
3. Se o resultado não for operando da operação seguinte, guarda o valor do registo A na memória de dados.

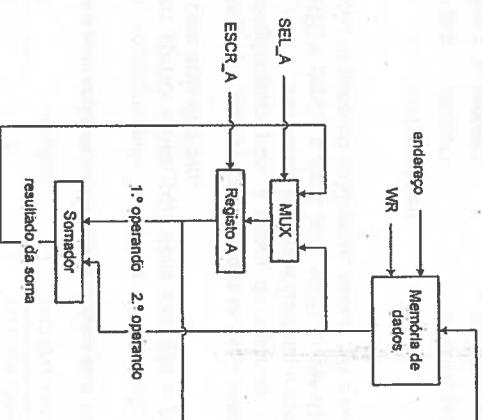


Fig. 3.6 - A ligação da saída do registo A à entrada da memória de dados permite guardar o valor deste registo na memória de dados (só quando se activa WR).

#### 3.3.1.2 UNIDADE ARITMÉTICA ELÓGICA (ALU)<sup>17</sup>

No entanto, um computador não pode só fazer somas. E multiplicações, por exemplo? Normalmente, os processadores são capazes de fazer um conjunto de operações variado, assunto que será desenvolvido no capítulo 4.

<sup>17</sup> A sigla em português é UAL (Unidade Aritmética e Lógica), mas a sigla anglo-saxónica ALU (Arithmetic and Logic Unit) já entrou em uso corrente, pelo que é aqui mantida.

Para simplificar, porque estamos ainda a um nível muito básico, vamos considerar apenas quatro operações: cada uma com dois operandos, a e b, tal como indicado na Tabela 3.2.

OPERAÇÃO	REPRESENTAÇÃO	TIPO DE OPERAÇÃO
SOMA	a + b	Aritmética (assume-se que a e b estão em complemento para 2)
SUBTRAÇÃO	a - b	
CONJUNÇÃO LÓGICA (AND)	a $\wedge$ b	Lógica, aplicada bit a bit de forma independente dos restantes (a e b são encarados como valores binários sem sinal)
DISJUNÇÃO LÓGICA (OR)	a $\vee$ b	

Tabela 3.2 - Conjunto básico de operações

A Fig. 3.7, a Fig. 3.8 e a Fig. 3.9 ilustram estas operações, mas antes há uma questão básica que precisa de ser resolvida:

**PROBLEMA 3.2** Quantos bits devem ter os operandos a e b?

**SOLUÇÃO** O hardware não pode variar de acordo com o valor dos dados que estão a ser processados. Por isso, a largura<sup>18</sup> da unidade de dados (registo A, multiplexer e somador) e da memória de dados tem de suportar o valor máximo admissível para os operandos a e b (os valores menores terão zeros nos bits de maior peso).

E agora depende. Os processadores de mais baixa gama usam 8 bits. São mais simples, mas ficam limitados aos valores entre 0 e 255 (ou entre -128 e +127, em complemento para 2). Depois há os processadores de 16, 32 e 64 bits. Uma das vantagens de ter mais bits é permitir manipular números maiores. No entanto, o hardware fica mais complicado de implementar (e mais dispendioso).

Assim, em sistemas muito pequenos (por exemplo, os pequenos processadores que muitos electrodomésticos já têm), usam-se 8 e 16 bits, pois nestes casos o custo é um factor muito importante e não estão envolvidos números muito grandes.

Os processadores de 32 bits usam-se sobretudo nos computadores de uso pessoal, seja de secretaria, seja portátil, seja até mesmo de mão, incluindo as agendas electrónicas e os telemóveis. Nestes casos o custo já não é tão crítico e já é preciso alguma capacidade de cálculo.

Os processadores que conseguem manipular operandos de 64 bits pertencem ao reino dos grandes sistemas (servidores), pois a sua capacidade de cálculo é grande mas o alto custo não favorece a sua utilização em larga escala a nível pessoal.

Neste capítulo introdutório, e por razões de simplicidade, assumimos que os operandos e o processador são de 8 bits (razão da inclusão do número 8 no nome do processador, PEPE-8, Processador Especial Para Ensino com 8 bits).

<sup>18</sup> Número máximo de bits que podem ser processados simultaneamente.



Fig. 3.7 - Soma e subtração de dois números binários de 8 bits

O nono dígito do resultado, correspondente ao último "e vai um", é descartado. O resultado deve continuar a ter apenas 8 bits, pois neste exemplo o processador não é capaz de manipular números com mais de 8 bits.

A subtração pode ser substituída por uma soma desde que se use o simétrico do segundo operando, ou seja, o seu complemento para 2 (troca do valor de todos os bits, ou complemento para 1, e soma com 1). As contas seguintes ilustram o complemento para 2 do número -04H (que dá +04H) e depois a soma com +66H, o que dá +6AH, tal como anteriormente com a subtração (o nono bit é diferente, mas como é descartado não tem influência).

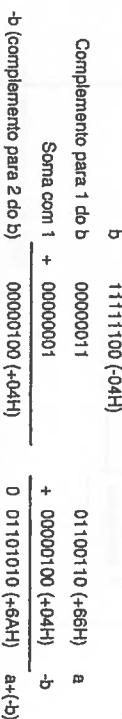


Fig. 3.8 - Equivalência entre subtração e soma com o complemento para 2

As secções 2.8.1 e 2.8.2 explicam estas operações aritméticas em mais detalhe. Nomeadamente, a secção 2.8.2 explica como se pode implementar uma unidade que suporte subtração à custa de um somador (fazendo o complemento para 2 do operando b antes de o fazer entrar no somador, que assim faz a operação  $a + (-b)$  em vez de  $a - b$ , mas produzindo o mesmo resultado).

As operações lógicas fazem-se igualmente bit a bit, mas agora a operação em cada bit é independente dos restantes. Os operandos são encarados como simples sequências de bits e não como números positivos ou negativos. Nestas operações não há "e vai um".

A Fig. 3.9 exemplifica a conjunção (AND) e disjunção (OR) lógicas. São usados os mesmos valores para os operandos que na Fig. 3.7, mas poderiam ser quaisquer outros valores.

As operações aritméticas fazem-se bit a bit, com um eventual "e vai um" para o bit seguinte, tal como fazemos nas contas à mão em decimal mas agora em binário (secção 2.8, na página 99). Assume-se que os operandos estão representados em complemento para 2 (secção 2.7.5), o que permite representar números positivos e negativos da mesma forma. Para cada número, que indica se esse número é positivo (se for 0) ou negativo (se for 1), resulta os números +62H e +6AH, respectivamente. Note-se o bit de maior peso (sinal) de cada número, que indica se esse número é positivo (se for 0) ou negativo (se for 1).

A operação AND dá 1 num bit do resultado apenas quando os dois bits correspondentes nos operandos são também 1. Caso contrário (pelo menos um deles a 0), o resultado nesse bit é 0.

A operação OR dá 0 num bit do resultado apenas quando os dois bits correspondentes nos operandos são também 0. Caso contrário (pelo menos um deles a 1), o resultado nesse bit é 1.

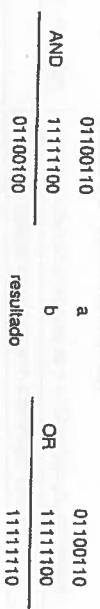


Fig. 3.9 - Conjunção (AND) e disjunção (OR) de dois números binários de 8 bits

Nestas operações, o operando b pode ser encarado como uma máscara, que permite afectar apenas alguns bits deixando os restantes intactos.

A operação AND da Fig. 3.9 copia o operando a para o resultado, mas apenas nos bits em que o operando b é 1. Os outros bits são forçados a 0 no resultado, mesmo que os bits correspondentes no operando a sejam 1.

A operação OR da Fig. 3.9 copia o operando a para o resultado, mas apenas nos bits em que o operando b é 0. Os outros bits são forçados a 1 no resultado, mesmo que os bits correspondentes no operando a sejam 0.

É possível assim forçar certos bits a 0 ou a 1 num valor guardado numa célula de memória.

Para forçar bits a 0, basta fazer o AND do valor na célula de memória com uma máscara adequada (número que tenha 0 nos bits que se quer forçar a 0 e 1 nos restantes bits), armazenando depois o resultado nessa mesma célula de memória.

Para forçar bits a 1, o processo é o mesmo mas deve usar-se a operação OR e uma máscara adequada (com 1 nos bits que se quer forçar a 1 e 0 nos restantes bits).

A secção 4.12.4, na página 254, apresenta mais detalhes.

#### PROBLEMA 5 Como fazer para que o processador suporte várias operações?

**SOLUÇÃO** Em vez de um simples somador, usar uma unidade que permita implementar várias operações, nomeadamente aritméticas e lógicas.

A Fig. 3.10 ilustra este conceito, em que em vez de um somador usa uma ALU (Unidade Aritmética e Lógica), capaz de executar uma das quatro operações indicadas na Tabela 3.2 sob selecção do sinal SEL\_ALU, que tem 2 bits (para permitir as quatro operações).

A Fig. 3.11 descreve uma possível implementação para esta ALU, constituída em torno de um multiplexor que selecciona para a saída o resultado correcto. Note-se que esta implementação não é única e nem sequer a mais optimizada, mas será certamente uma das mais simples.

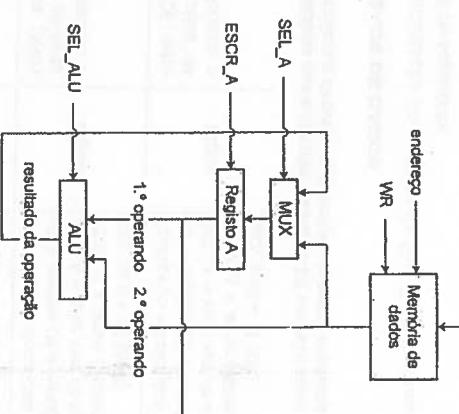


Fig. 3.10 - Unidade de dados com uma unidade aritmética e lógica (ALU) em vez de um simples somador. O sinal SEL\_ALU especifica a operação pretendida

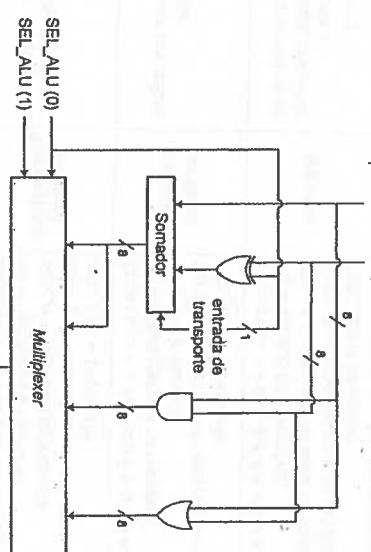


Fig. 3.11 - Possível implementação da unidade aritmética e lógica (ALU) da Fig. 3.10

A Tabela 3.3 descreve o funcionamento desta ALU em termos dos dois bits do sinal de selecção SEL\_ALU. Todas as unidades nesta ALU têm de ter o mesmo número de bits que os operandos, 8. Cada "porta lógica" é um conjunto de 8 portas lógicas de 2 entradas.

Quem usa uma RTL tem de apresentar uma tabela com as convenções usadas, tal como é feito na Tabela 3.4.

Para descrever um circuito de forma completa, incluindo os seus blocos constituintes e o seu comportamento, existem linguagens de descrição de hardware com regras universalmente definidas, como por exemplo VHDL e Verilog [Botros 2005].

SEGURO ALU	OPERAÇÃO	MULTIPLEXER	QUESTIONNAÇÕES
0	0	A + B	Saída do somador OU-exclusivo não altera o operando B
0	1	A - B OU A + (-B)	Saída do somador (que agora é um subtraçor, pois o segundo operando é o simétrico de B) B é complementado para 2: • OU-exclusivo troca todos os bits do operando B, pois SEL_ALU(0) = 1. • A entrada de transporte do somador fica a 1, pelo que soma mais 1
1	0	A $\wedge$ B	Saída do AND Conjunção lógica (AND) bit a bit dos dois operandos
1	1	A $\vee$ B	Saída do OR Disjunção lógica bit a bit(OR) dos dois operandos

Tabela 3.3 - Funcionamento da ALU da Fig. 3.11. O bit de SEL\_ALU é o da direita (0 de menor peso)

### 3.3.1.3 FUNCIONAMENTO DA UNIDADE DE DADOS

O circuito da Fig. 3.10 constitui uma unidade de dados básica, mas já é capaz de implementar um conjunto interessante de operações, que a Tabela 3.5 summariza, indicando o tipo de transformação operada sobre os dados e o valor que cada sinal de controlo deve ter em cada operação para a sua execução correcta.

A descrição de cada operação é feita não apenas em texto mas também em RTL (*Register Transfer Language*, ou Lingagem de Transferência de Registos), uma forma simbólica e compacta de descrever as operações básicas à custa de algumas convenções simples, descritas na Tabela 3.4.

RTL	SINIFICAÇO
A	Registo A
M[endereço]	Célula de memória com o endereço especificado
A $\leftarrow$ M[endereço]	Escreve no registo A uma cópia da célula de memória com o endereço indicado
M[endereço] $\leftarrow$ A	Escreve uma cópia do registo A na célula de memória com o endereço indicado
+,-, $\wedge$ , $\vee$	Operações básicas da ALU (soma, subtração, AND e OR)
(expressão) : operação	Executa a operação apenas se a expressão for verdadeira

Tabela 3.4 - Convenções da RTL

**NOTA** A RTL (*Register Transfer Language*) não é uma linguagem de programação, nem sequer é propriamente uma linguagem que permita descrever completamente a estrutura dos computadores. O seu objectivo é ajudar a descrever o comportamento do hardware, em muito baixo nível, e não passa na realidade de uma notação, razão pela qual também às vezes se usa a designação RTN (*Register Transfer Notation*) em vez de RTL.

Também é importante saber que não há só uma RTL, nem sequer uma notação universalmente aceite. Apenas o objectivo e os conceitos subjacentes são comuns.

OPERAÇÃO EM TEXTO E EM RTL	TIPO	SINALS DE CONTROLO
Lê uma célula da memória de dados e copila-a para o registo A	Transferência de dados	WR = inactivo (lê a memória) SEL_A = valor vem da memória ESCR_A = inativo (escreve no registo) SEL_ALU = irrelevante
A $\leftarrow$ M[endereço]		Endereço = indica qual é a célula a aceder WR = activo (escreve na memória) SEL_A = inativo (não escreve no registo) SEL_ALU = irrelevante
• Copia o registo A para uma célula da memória de dados	Transferência de dados	Endereço = indica qual é a célula a aceder WR = activo (escreve na memória) SEL_A = inativo (não escreve no registo) SEL_ALU = irrelevante
• M[endereço] $\leftarrow$ A		WR = inativo (lê a memória) SEL_A = valor vem da saída do somador ESCR_A = activo (escreve no registo) SEL_ALU = 00 (soma)
A $\leftarrow$ A + M[endereço]	Aritmética	Endereço = indica qual é a célula a aceder WR = inativo (lê a memória) SEL_A = valor vem da saída do somador ESCR_A = activo (escreve no registo) SEL_ALU = 01 (subtração)
• Subtra o registo A o valor de uma célula da memória de dados, colocando o resultado no registo A	Aritmética	Endereço = indica qual é a célula a aceder WR = inativo (lê a memória) SEL_A = valor vem da saída do somador ESCR_A = activo (escreve no registo) SEL_ALU = 10 (AND)
• A $\leftarrow$ A - M[endereço]	Aritmética	Endereço = indica qual é a célula a aceder WR = inativo (lê a memória) SEL_A = valor vem da saída do somador ESCR_A = activo (escreve no registo) SEL_ALU = 11 (OR)
Faz a conjunção lógica (AND) entre o registo A e uma célula da memória de dados, colocando o resultado no registo A	Lógica	Endereço = indica qual é a célula a aceder WR = inativo (lê a memória) SEL_A = valor vem da saída do somador ESCR_A = activo (escreve no registo) SEL_ALU = 10 (AND)
A $\leftarrow$ A $\wedge$ M[endereço]	Lógica	Endereço = indica qual é a célula a aceder WR = inativo (lê a memória) SEL_A = valor vem da saída do somador ESCR_A = activo (escreve no registo) SEL_ALU = 11 (OR)
Faz a disjunção lógica (OR) entre o registo A e uma célula da memória de dados, colocando o resultado no registo A	Lógica	Endereço = indica qual é a célula a aceder WR = inativo (lê a memória) SEL_A = valor vem da saída do somador ESCR_A = activo (escreve no registo) SEL_ALU = 11 (OR)
A $\leftarrow$ A $\vee$ M[endereço]	Lógica	Endereço = indica qual é a célula a aceder WR = inativo (lê a memória) SEL_A = valor vem da saída do somador ESCR_A = activo (escreve no registo) SEL_ALU = 11 (OR)

Tabela 3.5 - Operações básicas que o circuito da Fig. 3.10 permite fazer, com os valores dos sinais que controlam o circuito. A descrição aparece em texto e em RTL.

### SIMULAÇÃO 3.2 – UNIDADE DE DADOS

Esta simulação ilustra a utilização do circuito da Fig. 3.10, incluindo o da Fig. 3.11. Os aspectos cobertos incluem os seguintes:

- Inicialização da memória de dados (valores dos operandos) pelo utilizador;

- Especificação pelo utilizador de cada um dos sinais da Fig. 3.10;
- Verificação do funcionamento de cada operação básica por actuação desses sinais, com visualização dos dados presentes em cada um dos pontos do circuito, incluindo o interior da ALU;
- Detalhes do funcionamento dos sinais WR e ESCR\_A (flancos em que são activos);
- Implementação das restantes operações suportadas pela ALU;
- Verificação da possibilidade de utilização do resultado de uma operação no registo A como operando da operação seguinte.

**ESSENCIAL**

- A existência de um registo dentro da unidade de dados do processador é fundamental. Sem ele, as operações com dois operandos só seriam possíveis se as memórias permitissem ler duas células e escrever outra ao mesmo tempo!
- A ALU (Unidade Aritmética e Lógica) permite implementar diversas operações com dois operandos;
- Para além das operações da ALU, a unidade de dados tem de saber transferir dados entre o registo e a memória de dados;
- A largura em bits da unidade de dados (é de todos os seus recursos internos, incluindo o registo) deve ser igual à da memória e define a gama de números com que o processador consegue lidar (com  $N$  bits, o processador consegue lidar com numeros em complemento para 2 entre  $-2^{N-1}$  e  $2^{N-1}$  (secção 2.7.5)).
- Os processadores com uma largura de 8 e 16 bits usam-se para aplicações específicas de baixo custo e sem necessidade de grande capacidade de cálculo. Os processadores de 32 e 64 bits usam-se em computadores de utilização geral (a média é alto desempenho). (computadores de secretaria e servidores, respectivamente);
- A RTL (Register Transfer Language) é uma notação simbólica que permite descrever de forma compacta as operações básicas que envolvem transferência de dados entre elementos de memorização (células de memória e registos), bem como operações sobre esses dados.

### 3.3.2 UNIDADE DE controLO

#### 3.3.2.1 SINAIS DE controLO

A Tabela 3.5 indica que é preciso especificar cada um dos sinais que controlam o circuito pelo que é óbvio que ainda não resolvemos um problema fundamental:

**PROBLEMA 3.7** Para cada operação básica, onde e como se especifica que valor deve ter cada um dos sinais que controlam o circuito?

**SOLUÇÃO** Cada operação básica corresponde na realidade a uma instrução do processador. Logo, cada instrução (conteúdo de uma célula na memória de instruções) deve especificar todos os sinais necessários para se executar.

A Fig. 3.12 ilustra esta solução. Note-se que:

- endereço de dados deve ter o número de bits necessário para poder endereçar qualquer das células da memória de dados (depende do número de células dessa memória);
- WR, SEL\_A e ESCR\_A correspondem a 1 bit cada um;
- SEL\_ALU corresponde a 2 bits, para poder especificar uma das quatro operações que a ALU suporta.

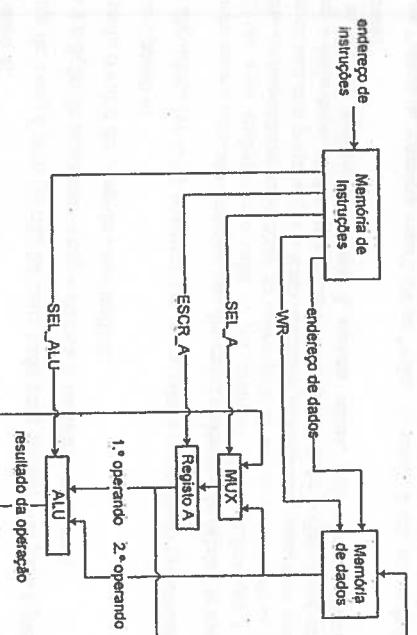


Fig. 3.12 - Especificação dos sinais de controlo pelas instruções na memória de instruções

#### 3.3.2.2 CONTADOR DE PROGRAMA (PC)<sup>19</sup>

Assim, cada célula da memória de instruções conterá uma instrução, com os valores de todos os sinais de controlo necessários para comandar os vários recursos do processador. Tal como a memória de dados, a memória de instruções precisa que se especifique um endereço, para indicar qual das suas células (qual instrução) se pretende executar. Na realidade, um programa não é normalmente constituído por uma única instrução, mas sim por uma sequência de instruções. Desta forma, não basta indicar apenas uma das

<sup>19</sup> A sigla em português será CP, mas ninguém a usa. A sigla anglo-saxónica PC (Program Counter) já entrou em uso corrente, pelo que é a usada neste livro.

instruções armazenadas, mas sim um conjunto de instruções, uma a uma, com uma dada sequência.

**PROBLEMA:** Como indicar quais as instruções (e o seu sequenciamento) que se pretendem executar num dado programa?

**SOLUÇÃO:** Utilizar um registo que indique qual das instruções da memória de instruções está em execução e um mecanismo que indique qual o valor seguinte desse registo, ou seja, qual a próxima instrução a ser executada.

O papel deste registo é especificar em cada instante o endereço (na memória de instruções) da instrução que se está a executar, tal como indicado na Fig. 3.13. Com esse endereço, a memória de instruções é lida e os sinais de controlo necessários aparecem na sua saída. Variando o valor desse registo, o endereço da memória de instruções muda e os sinais de controlo mudarão, executando uma nova instrução.

O modelo clássico de computação assume que as instruções são normalmente executadas em sequência, isto é, uma após outra. Se instruções consecutivas estiverem armazenadas na memória de instruções em endereços consecutivos, então para executar a sequência de instruções basta usar um simples contador e depois ir incrementando-o.

A Fig. 3.13 ilustra este conceito, usando um contador conhecido pelo nome óbvio de Contador de Programa (ou PC, *Program Counter*).<sup>20</sup> Em cada ciclo do sinal de relógio, o valor do PC é incrementado de uma unidade, passando a executar-se uma nova instrução.

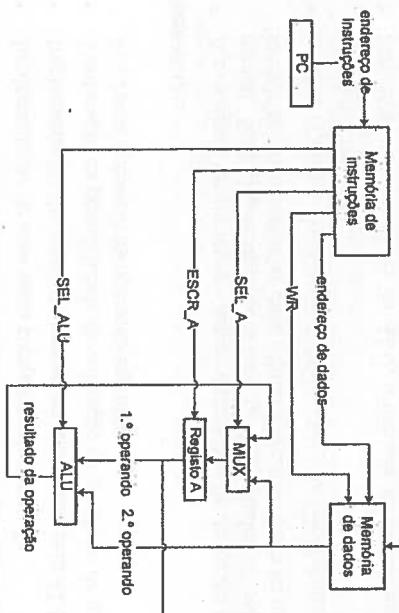


Fig. 3.13 - Especificação da instrução a executar pelo contador de programa (PC - *Program Counter*). Embora rudimentar, este circuito já é o embrião do PEP-E8

<sup>20</sup> Alguns processadores comerciais utilizam a designação IP (*Instruction Pointer*, ou Apontador de Instrução), mas a funcionalidade é a mesma.

### 3.3.2.3 UM PROGRAMA SIMPLES

Vamos ilustrar o funcionamento do PC e da unidade de controlo com um programa simples. Consideremos, por exemplo, que pretendemos calcular a soma de um dado número positivo N com todos os números positivos menores que ele, ou seja:

$$\text{soma} = N + (N-1) + (N-2) + \dots + 2 + 1$$

Para implementar este cálculo, vamos utilizar duas células da memória de dados, a que chamarímos *soma* e *temp*<sup>21</sup>, com um algoritmo simples:

1. Inicializa o valor de *soma* com zero;
2. Inicializa o valor de *temp* com N;
3. Se o valor de *temp* é negativo (errado, só valores positivos são admitidos), vai para o passo 8;
4. Se o valor de *temp* é zero (já não há mais nada para somar), vai para o passo 8;
5. Efectua a adição de *soma* com *temp* e coloca o resultado em *soma*;
6. Decrementa o valor de *temp* de uma unidade;
7. Vai para o passo 4;
8. Fim do algoritmo (se N for positivo, em *soma* estará o resultado pretendido).

**NOTA:** Os nomes *soma* e *temp* são símbolos que designam endereços de células de memória. Se ao símbolo *soma* atribuirmos o valor 3, por exemplo, isso significa que a célula de memória correspondente se localiza no endereço 3. O uso de nomes para os endereços em vez de números permite que o texto descritivo se mantenha, quaisquer que sejam os endereços onde estas células de memória se localizam (não podem corresponder as duas ao mesmo endereço, ou seja, à mesma célula, pois correspondem a dados diferentes).

Falar na “célula de memória *soma*”, ou no “valor de *soma*”, é um abuso de linguagem que se tolera apenas para facilitar a referência. Na realidade, dever-se-ia dizer “Valor da célula de memória localizada no endereço indicado pelo valor do símbolo *soma*”.

Este algoritmo:

- Testa se o utilizador se entrou e inicializou o valor de *temp* com um número N negativo, caso em que termina logo;
- Também funciona se o utilizador especificar o valor zero para N;
- Executa os passos 1, 2, 3 e 8 apenas uma vez (1, 2 e 3 no início e 8 no fim);
- Executa os passos 4 a 7 um número de vezes igual ao valor de N (na realidade, o passo 4 é executado mais uma vez, quando o algoritmo termina);

<sup>21</sup> O nome indica que se trata de uma variável cujo valor é usado apenas temporariamente.

- Em cada execução dos passos 4 a 7, adiciona à célula soma mais um número e decrementa o valor de temp, que fica preparado para ser adicionado à célula soma na passagem seguinte.
- O algoritmo está expresso em texto num formato razoavelmente abstrato, pelo que em RTL (Tabela 3.4) ficará transformado num programa mais conciso e compacto. Por simplicidade, o número do passo do algoritmo é usado como endereço da memória de instruções (isto é, cada passo corresponde a uma instrução que ocupa uma célula de memória de instruções, começando a partir do endereço 1). Entre parênteses aparece uma breve descrição de cada instrução do programa (designada comentário).

```

1. M[soma] ← 0           (inicializa soma com zero)
2. M[temp] ← N          (inicializa temp com N)
3. (M[temp] < 0) : PC ← 8   (se temp for negativo, salta para o fim)
4. M[temp] ← 0 ; PC ← 8   (se temp for zero, salta para o fim)
5. M[soma] ← M[soma] + M[temp]  (adiciona temp a soma)
6. M[temp] ← M[temp] - 1    (decrementa temp)
7. PC ← 4                 (salta para o endereço 4)
8. PC ← 8                 (fim do programa)

```

#### Programa 3.1 - Programa que implementa o algoritmo de soma (versão 1)

Note-se que:

- soma, temp e N são constantes simbólicas, isto é, nomes que representam números constantes que terão de ser concretizados com valores numéricos efectivos antes de o programa ser executado;
- As várias instruções do programa são executadas sequencialmente, uma de cada vez e pela ordem com que aparecem especificadas (começando na 1);
- O registo PC (*Program Counter*) informa em que instrução a execução está, o que quer dizer que:

- No início, o PC deve ser inicializado com 1 (endereço da primeira instrução);
- Avançar para a instrução seguinte corresponde a incrementar o PC;
- Nas instruções 3, 4 e 7, em que a ordem natural de execução é interrompida (diz-se que há um salto), o PC deve ser carregado com o novo endereço onde deve recomeçar a execução, em vez de ser incrementado;
- Os saltos na execução podem ser:
  - Condicionais, se saltar ou não depende de uma condição (passos 3 e 4). Se a condição for falsa e o salto não for efectuado, então o PC deve simplesmente ser incrementado;
  - Incondicionais, se o salto é sempre efectuado, independentemente de qualquer condição (passo 7).
- O "fim do programa" é implementado por um ciclo infinito: a instrução 8 salta para ela própria. Isto reflecte o facto de que os processadores normalmente não

param (isto é, enquanto o sinal de relógio estiver presente, a unidade de controlo vai executando instruções sem parar). É como um automóvel, que quando para (num semáforo, por exemplo) continua com o motor a trabalhar. Desligar o motor (quando se retira a chave) corresponde nos processadores a desligar a fonte de alimentação.

A tarefa seguinte será tentar mapear este programa expresso em RTL no circuito da Fig. 3.13, para se verificar como é que as operações realmente funcionam. Uma simples inspecção ao Programa 3.1 e ao circuito da Fig. 3.13 levanta logo alguns problemas que é preciso resolver.

#### 3.3.2.4 CONSTANTES NO PROGRAMA

Em vários endereços do programa são referidas constantes, que podem ser de dois tipos:

- Litterais (números propriamente ditos, como 8 ou 4);
- Simbólicas (nomes que representam números, como soma, temp e N), que têm de ser substituídas pelo seu valor numérico efectivo antes de o programa ser executado.

Uma constante é um valor fixo, que não pode mudar, e que por conseguinte não faz sentido corresponder ao conteúdo de uma célula da memória de dados (cujo valor pode ser mudado ao longo do tempo, efectuando escritas nessa célula). Em qualquer dos casos, as constantes têm um valor numérico a que é preciso aceder ao longo da execução do programa. A Tabela 3.6 indica as constantes que são necessárias em cada um dos endereços do programa.

ENDEREÇO DA INSTRUÇÃO	CONSTANTE 1	CONSTANTE 2	CONSTANTE 3
1	soma	0	
2	temp	N	
3		0	8
4	temp	0	8
5	soma	temp	
6		1	
7	4		
8	8		

Tabela 3.6 - Constantes usadas pelo Programa 3.1

**PROBLEMA 5** Como especificar as constantes no programa?

**SOLUÇÃO (RESPOSTA)** Especificar as constantes nas próprias instruções que as referenciam, tal como indicado na Fig. 3.14. É apenas mais um campo de cada célula da memória de instruções, tal como o endereço de dados que já existia.

O multiplexer colocado na entrada da memória de dados (Fig. 3.14) permite mesmo escrever directamente o valor de uma constante na memória de dados (sem perder a hipótese de copiar o conteúdo do registo A para uma célula da memória de dados).

No entanto, tal como indicado pela Tabela 3.6, se algumas instruções precisam de especificar apenas uma constante, outras instruções precisam de mais constantes. Como a largura das células da memória de instruções não pode variar de acordo com a instrução que contém, os bits necessários para o número máximo de constantes nas várias instruções teriam de existir em todas as células da memória de instruções, o que implicaria um grande desperdício de memória.

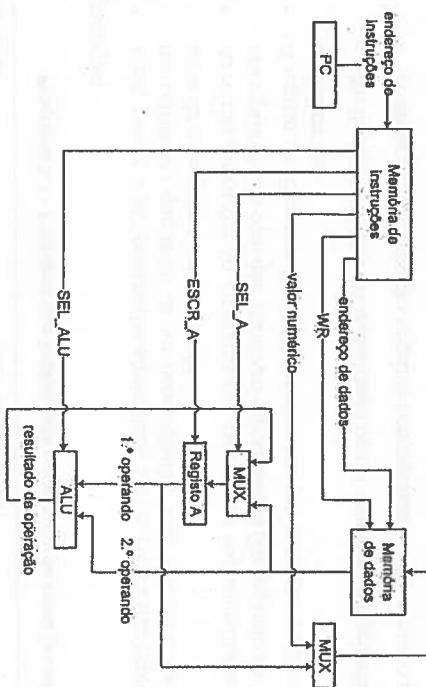


Fig. 3.14 - Hipótese de especificação de constantes nas instruções. O problema é que nem todas as instruções têm apenas uma constante. Esta não é uma boa solução

Por conseguinte, é necessária uma nova solução:

**SOLUÇÃO INICIAL:** Especificar as constantes nas próprias instruções, mas apenas uma por cada instrução. O truque é transformar instruções que usem mais do que uma constante numa sequência de instruções mais simples, em que cada uma use apenas uma constante.

Assim, os campos endereço de dados e valor numérico na Fig. 3.14 podem ser o mesmo, partilhado. Numas instruções servirá como endereço de uma célula da memória de dados. Noutras servirá para poder especificar um valor numérico.

Isto implica algumas mudanças no Programa 3.1, nomeadamente para transformar as instruções que usem duas constantes em duas instruções que usem apenas uma constante cada uma. O segredo é usar o registo A.

Por exemplo, a instrução 1 do Programa 3.1,

$M[soma] \leftarrow 0$  (inicializa a zero o conteúdo da célula de memória soma)

é transformada em

$A \leftarrow 0$  (inicializa o registo A a zero)  
 $M[soma] \leftarrow A$  (cópia o registo A para a célula de memória soma)

Também as referências em leitura a células de memória podem ser substituídas por referências em leitura ao registo A quando o seu valor seja idêntico (por exemplo, nas instruções 3 e 4 do Programa 3.1 pode substituir-se  $M[temp]$  por A, pois este registo nesta altura contém o mesmo valor que a célula de memória  $M[temp]$ ).

Finalmente, é preciso ter em atenção que os resultados das operações feitas na ALU (como por exemplo a soma da instrução 5 e a subtração da instrução 6 no Programa 3.1) são sempre guardados no registo A (Fig. 3.13). Assim, a forma correcta de implementar estas operações consiste em efectuá-las com o resultado a ser guardado no registo A e depois acrescentar uma instrução para copiar o resultado do registo A para a célula de memória correcta.

Com estas alterações, o Programa 3.1 fica com o aspecto do Programa 3.2. Note-se que:

- soma, temp e N são constantes simbólicas que terão de ser concretizados com valores numéricos efectivos antes de o algoritmo ser executado;
- Já não há referências a mais do que uma constante na mesma instrução, com exceção das instruções 5 e 6 que ainda referem duas constantes, 0 e 13. É um problema que será resolvido na página 141.

**NOTA** No Programa 3.2, a instrução 4 e 11 são idênticas e têm a mesma função (actualizar temp na memória a partir do registo A). Se observarmos o programa, facilmente verificaremos que para executar a instrução 6 tem sempre de se executar antes ou a 4 ou a 11 (devido ao salto incondicional da instrução 12). Isto permite fazer uma optimização no programa, eliminando a instrução 11 e fazendo o salto para a instrução 4 em vez de 6 (dado que a instrução 5 não altera o valor do registo A).

Na prática, isto significa que a instrução 4 passa a ser partilhada entre a inicialização da célula temp com N (valor obtido na instrução 3) e as actualizações subsequentes com os valores N-1, N-2, etc., obtidos na instrução 10 em passagens sucessivas.

#### Programa 3.2 - Programa que implementa o algoritmo de soma (versão 2)

O programa otimizado ficaria nesta forma

1.	$A \leftarrow 0$	(inicializa A com zero)
2.	$M[\text{sona}] \leftarrow A$	(inicializa sôma com zero)
3.	$A \leftarrow N$	(inicializa A com o valor de N)
4.	$M[\text{temp}] \leftarrow A$	(actualiza temp na memória)
5.	$(A < 0) : PC \leftarrow 12$	(se temp for negativo, salta para o fim)
6.	$(A = 0) : PC \leftarrow 12$	(se temp for zero, salta para o fim)
7.	$A \leftarrow A + M[\text{sona}]$	(adiciona sôma a A, que é igual a temp)
8.	$M[\text{sona}] \leftarrow A$	(actualiza sôma na memória)
9.	$A \leftarrow M[\text{temp}]$	(vai buscar temp de novo)
10.	$A \leftarrow A - 1$	(decrementa A. A célula temp é atualizada em 4)
11.	$PC \leftarrow 4$	(fim para o endereço 4)
12.	$PC \leftarrow 12$	(fim do programa)

### **Programa 3.3 - Programa 3.2 optimizado**

Esta optimização não tem qualquer impacte no comportamento do programa. A sua vantagem é apenas a de o programa ficar mais curto. No entanto, é importante verificar que as optimizações fazem o programa ficar mais obscuro e conduzem frequentemente a erros (nem sempre a optimização se pode fazer, ao alterar um programa o programador já não se lembra em que condições fez a optimização e depois faz alterações não compatíveis, etc.). A decisão sobre optimizar ou não deve pesar os benefícios (são mesmo importantes ou pouco adiantam?) contra os potenciais malefícios (normalmente sempre muito perigosos).

Em caso de dúvida, é melhor não optimizar. Em nome da simplicidade e clareza, essa optimização não é feita na evolução deste exemplo ao longo deste capítulo.

utilizações:

- Ser usadas como endereço num acesso à memória de dados (instruções 2, 4, 7, 8 e 11);
  - Ser guardadas no registo BC (instruções 5, 6 e 12), o que corresponde a saltos no programa;
  - Ser usadas como um dos operandos de uma operação aritmética (instrução 10). Estas diferentes utilizações necessitarão com certeza de diferentes soluções para a sua implementação.

**Solução** Especificar caminhos que permitam às constantes fluir das instruções para os recursos em que vão ser usadas.

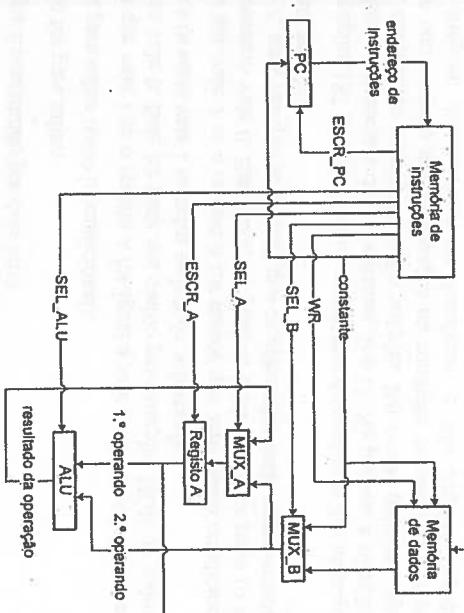


Fig. 3.15 - Circuito com suporte para constantes

O sinal endereço de dados vindo da memória de instruções é agora chamado constante, pois serve para todas as constantes;

- Entrada de endereços da memória de dados (já Ligava) – Permite a algumas instruções especificar um endereço de dados;
  - Novo *multiplexer* (MUX\_B) – Suporta as instruções que guardam constantes no registo A e instruções em que o 2.º operando é uma constante;
  - Entrada do PC – Suporta os saltos (em que a constante que especifica o novo endereço é guardada no PC).

Há mais dois sinais de controlo (ambos de 1 bit cada um):

  - SEL\_B – Controla a selecção da entrada do *multiplexer* MUX\_B;
  - ESCR\_PC – Indica em cada instrução se o próximo valor de PC é obtido por contagem simples (incremento de uma unidade, passa à instrução seguinte) ou por carregamento do endereço especificado na instrução (caso de um salto).

OPERAÇÃO	SEL_A	SEL_B	ESCR_A	ESCR_PC
$A \leftarrow \text{constante}$	Lado do MUX_B	Lado da constante	Inactivo	Inactivo
$A \leftarrow M[\text{constante}]$	Lado do MUX_B	Lado da memória	Activo	Inactivo
$PC \leftarrow \text{constante}$	Irrelevante	Irrelevante	Inactivo	Activo
$A \leftarrow A + \text{constante}$	Lado da ALU	Lado da constante	Activo	Inactivo

Tabela 3.7 - Casos de utilização de constantes e valores dos sinal relevantes que permitem que as constantes sejam levadas até aos recursos adequados

### 3.3.25 SALTOS NO PROGRAMA

Onde há uma sequência de instruções surgirá, mais tarde ou mais cedo, a necessidade de interromper essa sequência, passando o controlo para outro ponto do programa (o que vulgarmente se denomina salto).

O salto é implementado simplesmente alterando o valor do PC e pode ser de um de dois tipos:

- **Condicional** – Se a efectivação do salto depende de uma condição (caso das instruções 5 e 6 do Programa 3.2). Se a condição for verdadeira, o PC é carregado com a constante indicada. Se for falsa, o PC é simplesmente incrementado e a constante é ignorada;
- **Incondicional** – Se o salto for sempre realizado, independentemente de qualquer condição (caso da instrução 12 do Programa 3.2).

O circuito da Fig. 3.15 já suporta saltos incondicionais, simplesmente activando ESCR<sub>PC</sub> quando a instrução na memória de instruções inclui a constante adequada (novo valor do PC). No entanto, não prevê nada para suportar saltos condicionais.

**PROBLEMA** Como implementar os saltos condicionais?

**SOLUÇÃO** Definir quais as condições possíveis e seleccionar um dos bits que as representam por meio de um *multiplexer*, tal como indicado na Fig. 3.16. Cada bit seleccionado pode ser 0 ou 1, dependendo do valor da condição.

O objectivo é activar ESCR<sub>PC</sub> (e o salto é realizado) ou não (e passa à instrução seguinte), de acordo com uma dada condição. Dado que estamos ao nível do hardware, não é praticável admitir condições muito complexas para os saltos condicionais. Na realidade, consideram-se apenas quatro condições extremamente simples, de forma a englobar todas as situações:

- Nunca salta (condição sempre falsa, passa sempre à instrução seguinte) – É o que acontece em todas as instruções que não são de salto;
- Salta sempre (condição sempre verdadeira, salta para o endereço indicado pela constante) – É o caso das instruções de salto incondicional, como a instrução 12 do Programa 3.2;

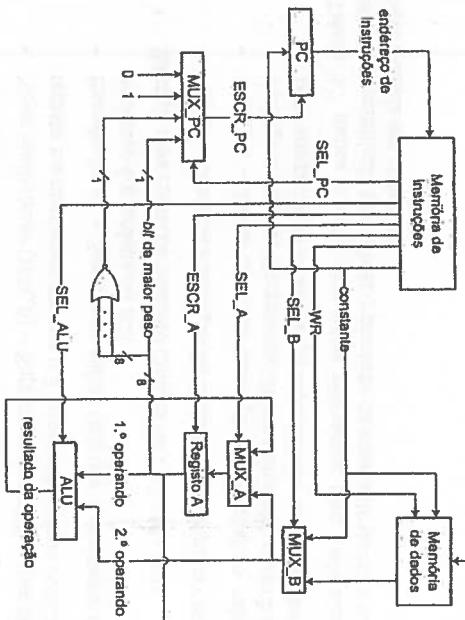


Fig. 3.16 - Circuito com suporte para saltos (condicionais e incondicionais)

Salta se o valor do registo A for igual a zero (condição depende do valor de A) – É o caso da instrução de salto condicional 6 no Programa 3.2;

Salta se o valor do registo A for menor que zero (condição depende do valor de A);

– É o caso da instrução de salto condicional 5 no Programa 3.2.

Arbitrando que ESCR<sub>PC</sub> é activo a 1, o MUX<sub>PC</sub> na Fig. 3.16 selecciona (por indicação do sinal SEL<sub>PC</sub>, que é constituído por dois bits):

- 0 se não for para saltar;
- 1 se for para saltar (salto incondicional);

Um bit que vale 1 se o registo A for igual a zero (salto condicional se  $A=0$ ), caso contrário vale 0. Este bit pode ser obtido pelo simples NOR de todos os bits do registo A (a saída vale 1 se todos os bits de A forem 0);

Um bit que vale 1 se o registo A for menor que zero (salto condicional se  $A<0$ ), caso contrário vale 0. Este bit é, na prática, o bit de maior peso (o de sinal) do registo A, uma vez que se assume que os operandos estão representados em complemento para 2.

Entretanto, na página 137 foi identificado o problema de cada uma das instruções 5 e 6 do Programa 3.2 fazer referência a duas constantes, 0 e 13. Na prática, a solução 11 resolve este problema, escolhendo condições muito simples. Por outras palavras, apenas o zero pode ser usado como termo de comparação na condição do salto, o que permite uma solução mais simples, directamente em hardware, e não implica especificar outra constante. Portanto, apenas uma constante, o 13, precisa de ser incluída nas instruções 5 e 6 do Programa 3.2.

### 3.3.2.6 FUNCIONAMENTO DETALHADO DO PROGRAMA

Usando o circuito da Fig. 3.16 como base, o Programa 3.2 pode representar-se de forma mais detalhada, por especificação do valor dos vários sinais para cada instrução, tal como indicado na Tabela 3.8. Esta tabela representa ainda a forma como a memória de dados é usada em cada instrução (nas que a usam).

Esta tabela é estática, isto é, refere-se apenas às instruções que existem no programa mas não à sua execução. Isto é feito na Tabela 3.9, que representa a evolução do registo A e das duas células da memória de dados (soma e temp) ao longo das várias iterações assumindo  $n=4$  (o valor é propostadamente baixo, para limitar o tamanho da tabela).

Se compararmos a Fig. 2.44, na página 86, com a Fig. 3.16, verifica-se que, na realidade, esta última não passa de uma máquina de estados microprogramada (secção 2.6.7.4), em que:

- A memória de instruções desempenha o papel de ROM com microinstruções;
  - Cada estado equivale a uma instrução do PEPE-8. A célula da memória de instruções correspondente inclui o valor de cada um dos sinais que comandam o circuito, de acordo com a Tabela 3.8;
  - O PC indica qual o estado (instrução) em que a máquina de estados se encontra.
- O PEPE-8 é um processador muito simples e cada uma das suas instruções mapeia-se directamente numa só microinstrução, executada num só ciclo de relógio. O PEPE (processador de 16 bits – ver capítulo 4) é mais complexo e cada instrução é o resultado da execução de várias microinstruções (secção 7.2.4, na página 585).

#### ESSENCIAL

- A unidade de controlo inclui um registo (PC – *Program Counter*) que contém o endereço (na memória de instruções) da instrução a ser executada. Cada instrução inclui directamente os sinais de controlo e uma constante (seja um dado, seja um endereço); pelo que é semelhante a uma microinstrução numa unidade de controlo microprogramada;
- O PC é um contador com carregamento paralelo. Normalmente conta sequencialmente, avançando em cada ciclo de relógio. No entanto, se a instrução assim o especificar, pode saltar para um endereço diferente, carregando a constante da instrução no PC. Os saltos podem ser condicionais (dependem de uma condição ou incondicionais (fazem sempre). As condições são normalmente muito simples ( $A=0$  ou  $A < 0$ ));
- Em cada ciclo de relógio é executada uma nova instrução. Quando o relógio inicia um novo ciclo, o PC muda e as acções preparadas durante a instrução anterior (escrita no registo A, por exemplo) ocorrem nessa altura. Assumem-se registos que mudam o estado num dos flancos do relógio.

INSTRUÇÕES (RTL)	MEMÓRIA DE INSTRUÇÕES							MEMÓRIA DE DADOS		
	ENDEREÇO	SEL <sub>PC</sub> (2)	SEL <sub>ALU</sub> (2)	ESCR_A	SEL_A (1)	SEL_B (1)	WR (1)	CONST (6)	ENDEREÇO	ACESSO
$A \leftarrow 0$	1	Segue		Sim	MUX_B	Const	Não	0	soma	Escrita
$M[soma] \leftarrow A$	2	Segue		Não			Sim		N	
$A \leftarrow N$	3	Segue		Sim	MUX_B	Const	Não		temp	
$M[temp] \leftarrow A$	4	Segue		Não			Sim			
$(A < 0) : PC \leftarrow 13$	5	Salta_N		Não			Não	13		
$(A = 0) : PC \leftarrow 13$	6	Salta_Z		Não			Não	13		
$A \leftarrow A + M[soma]$	7	Segue	Soma	Sim	ALU	Mem	Não	soma	soma	Leitura
$M[soma] \leftarrow A$	8	Segue		Não			Sim	soma	soma	Escrita
$A \leftarrow M[temp]$	9	Segue		Sim	MUX_B	Mem	Não	temp	temp	Leitura
$A \leftarrow A - 1$	10	Segue	Subtra	Sim	ALU	Const	Não	1		Escrita
$M[temp] \leftarrow A$	11	Segue		Não			Sim	temp	temp	
$PC \leftarrow 6$	12	Salta		Não				Não	6	
$PC \leftarrow 13$	13	Salta		Não				Não	13	

Tabela 3.8 - Conteúdo da memória de instruções e uso da memória de dados no caso do Programa 3.2. Os valores entre parênteses são o número de bits que cada sinal ocupa, num total de 16 bits (largura da memória de instruções)

Convenções:

- Nas células em branco o valor do respectivo sinal é irrelevante (não afecta o funcionamento);
- SEL<sub>PC</sub> – Valores possíveis: 00-Segue (não salta), 01-Salta (salto incondicional), 10-Salta\_Z (salta se  $A=0$ ), 11-Salta\_N (salta se  $A < 0$ );
- SEL<sub>ALU</sub> – Valores possíveis: 00-Soma, 01-Subtrai, 10-AND, 11-OR;
- ESCR\_A, WR – Valores possíveis: 0-Não (inactivo, lê), 1-Sim (ativo, escreve);
- SEL\_A – Valores possíveis: 0-ALU (seleciona a entrada que liga à saída da ALU), 1-MUX\_B (seleciona a entrada do lado do MUX\_B);
- SEL\_B – Valores possíveis: 0-Const (seleciona a entrada do lado da constante), 1-Mem (seleciona a entrada do lado da memória de dados).

### 3.3.3 O PROCESSADOR (PEPE-8) E AS MEMÓRIAS

A Fig. 3.17 representa o circuito da Fig. 3.16 com o processador e as suas duas unidades internas (de dados e de controlo) identificadas. Note-se que:

- A unidade de controlo lida sobre tudo com a geração dos sinais de controlo, gestão do PC e da sua actualização, através dos saltos condicionais e incondicionais, e interacção com a memória de instruções;
  - A unidade de dados lida sobre tudo com o processamento dos dados, nomeadamente através da ALU e do registo A, e com a interacção com a memória de dados.
- No topo da simplicidade (mais simples que 16 bits, por exemplo), razão pela qual este processador foi baptizado de PEPE-8 (Processador Especial Para Ensino com 8 bits). As secções seguintes indicam quais as implicações (no processador e nas memórias) decorrentes dessa decisão.

INSTRUÇÕES (RTL)	ENDEREÇO	ITERAÇÃO 0	ITERAÇÃO 1	ITERAÇÃO 2	ITERAÇÃO 3	ITERAÇÃO 4	ITERAÇÃO 5						
		A	SOMA	TEMP	A	SOMA	TEMP	A	SOMA	TEMP	A	SOMA	TEMP
$A \leftarrow 0$	1	0	xx	xx									
$M[soma] \leftarrow A$	2	0	0	xx									
$A \leftarrow N$	3	4	0	xx									
$M[temp] \leftarrow A$	4	4	0	4									
$(A < 0) : PC \leftarrow 13$	5	4	0	4									
$(A = 0) : PC \leftarrow 13$	6	4	0	4	3	4	3	2	7	2	1	9	1
$A \leftarrow A + M[soma]$	7	4	0	4	7	4	3	9	7	2	10	9	1
$M[soma] \leftarrow A$	8	4	4	4	7	7	3	9	9	2	10	10	1
$A \leftarrow M[temp]$	9	4	4	4	3	7	3	2	9	2	1	10	1
$A \leftarrow A - 1$	10	3	4	4	2	7	3	1	9	2	0	10	1
$M[temp] \leftarrow A$	11	3	4	3	2	7	2	1	9	1	0	10	0
$PC \leftarrow 6$	12	3	4	3	2	7	2	1	9	1	0	10	0
$PC \leftarrow 13$	13										0	10	0

Tabela 3.9 - Evolução do registo A do processador e da memória de dados ao longo da execução do Programa 3.2 para N=4. Os rectângulos a cinzento indicam os endereços de instruções por onde passa o controlo em cada iteração

#### Notas:

- As células de memória soma e temp estão inicialmente com um valor qualquer, não previsível, indicado por "xx". Só depois da primeira escrita passam a ter um valor definido;
- O valor do PC é sempre igual ao endereço da instrução corrente;
- Em cada Instrução, a escrita no registo A ou na memória de dados só acontece realmente no ciclo de relógio seguinte, pois durante a instrução corrente apenas são preparados os sinais para que a escrita ocorra na próxima transição do relógio. É também nessa altura que o PC muda para a instrução seguinte. Assumem-se registos que mudam o seu estado num dos flancos do relógio. *Idem* para a memória de dados, relativamente a WR.

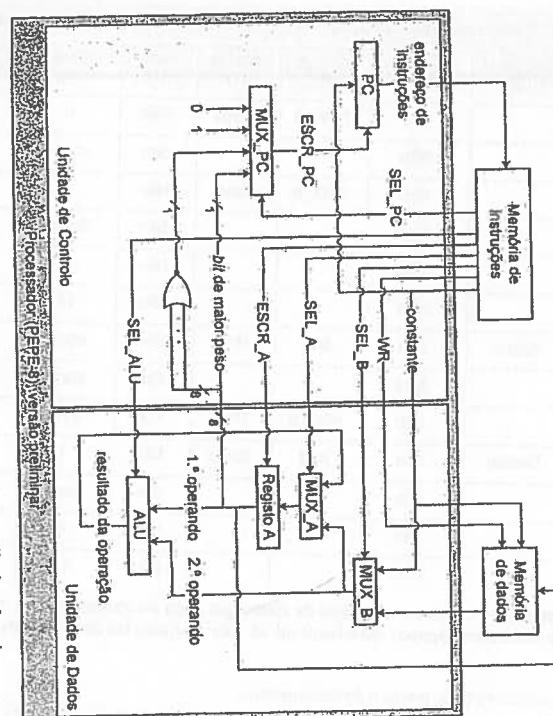


Fig. 3.17 - Circuito completo do processador (PEPE-8, versão preliminar) e das memórias de instruções e de dados

#### 3.3.3.1 PROCESSADOR (PEPE-8)

Todos os recursos da unidade de dados do PEPE-8 têm uma largura de 8 bits. Isto implica que o registo A tem 8 bits, e o mesmo acontece à ALU. Não é possível representar directamente números em complemento para 2 fora da gama -128 a +127.

Um dos truques usados pelos processadores de pequena largura é dividir o processamento de números maiores do que a largura permite em diversas operações. Por exemplo, para fazer uma soma entre dois números de 16 bits num processador de 8 bits fazem-se duas somas, 8 bits de cada vez:

- Cada operando (e o resultado) gasta duas células de memória (de 8 bits cada uma);
- Os 8 bits de menor peso do resultado são obtidos pela soma dos 8 bits de menor peso de cada um dos dois operandos;
- Os 8 bits de maior peso do resultado são obtidos pela soma dos 8 bits de maior peso de cada um dos dois operandos;
- Aos 8 bits de maior peso do resultado soma-se 1 se tiver havido transporte ("e vai um") da soma dos 8 bits de menor peso.

Idêntico raciocínio poderia ser feito para outras operações. Obviamente, seria mais rápido efectuar a operação de uma só vez num processador com largura de 16 bits, mas pelo menos consegue fazer-se a operação num processador de 8 bits.

O mesmo se pode dizer de um processador de 16 bits a tentar executar operações com operandos de 32 bits. A secção 4.11.2, na página 240, trata precisamente desse problema.

### 3.3.3.2 MEMÓRIA DE DADOS

A memória de dados deve também ter a mesma largura, 8 bits, pois o PEPE-8 escreve ou lê da memória 8 bits de uma só vez.

Poderá ter mais, mas só 8 bits poderão ligar ao PEPE-8 (os restantes serão desperdiçados), pelo que não tem interesse. Não pode ter menos, pois alguns dos bits do PEPE-8 não poderiam ser memorizados.

O número de bits necessários para endereçar a memória de dados é independente da largura da memória. Depende apenas do número de células (capacidade) que se pretende ter na memória de dados.

No entanto, os endereços das células da memória de dados (soma e temp no Programa 3.2, por exemplo) têm de ser especificados no campo da constante de cada instrução, precisamente o mesmo usado para especificar os valores das constantes de dados (como no Programa 3.2).

Ora um dos casos de utilização das constantes (Tabela 3.7) é serem memorizadas no registo A e outro é entrarem na ALU para ser feita uma operação com o registo A, o que obriga as constantes a terem 8 bits.

Logo, uma decisão que faz sentido é limitar o número de bits do endereço da memória de dados também a 8, tal como os dados. É importante notar que esta decisão se baseia apenas na análise da arquitectura do processador (Fig. 3.17) e não em qualquer limitação da memória de dados.

Assim, o PEPE-8 suporta apenas uma capacidade da memória de dados de 256 células (8 bits de endereço de dados), o que é manifestamente pouco pelos padrões actuais.

(qualquer computador pessoal tem centenas de milhões de células de memória), mas suficiente para pequenos programas pedagógicos.

### 3.3.3.3 MEMÓRIA DE INSTRUÇÕES

Na Fig. 3.17 pode verificar-se que cada célula da memória de instruções tem de incluir vários sinais, tal como indicado na Tabela 3.10, num total de 16 bits. O campo da constante tem 8 bits, tal como acabou de ser discutido.

Isto implica que a memória de instruções tenha 16 bits de largura, para cada instrução poder ser completamente lida de uma só vez. Não há qualquer incongruência com a largura de dados do PEPE-8. Nem todos os 16 bits de cada instrução vão para o mesmo ponto do circuito. Apenas os 8 bits da constante poderão entrar no circuito dos dados. Os restantes são apenas de controlo.

SINAL	FUNÇÃO	N.º DE BITS
SEL_PC	Escolhe forma de opter o próximo valor do PC	2
SEL_ALU	Seleciona uma das operações da ALU	2
ESCR_A	Quando activo, o registo A é escrito com o valor na sua entrada	1
SEL_A	Seleciona a fonte da entrada do registo A (saída da ALU ou saída do MUX_B)	1
SEL_B	Seleciona a fonte do 2.º operando (constante ou saída da memória de dados)	1
WR	Quando activo, a memória de dados é escrita com o valor na sua entrada (a constante indica qual o endereço da célula escrita)	1
Constante	Valor da constante especificada na instrução (endereço ou valor de dados)	8
	Total	16

Tabela 3.10 - Resumo dos sinais que cada célula da memória de instruções deve conter, juntamente com o número de bits necessário para cada sinal

**[NOTA]** O PEPE-8 é um processador muito básico, em que o principal objectivo é a simplicidade. Num processador mais elaborado, como o PEPE (processador de 16 bits), a introduzir já no capítulo 4, as instruções não estão tão separadas dos dados, o que significa que instruções e dados partilham alguns dos recursos de hardware e circuitos por algumas caminhos comuns. Por esta razão, adoptam-se algumas técnicas de modo a conseguir que o número de bits de cada instrução seja igual à largura de dados do processador. No caso do PEPE-8 tal não acontece, por uma razão extremamente pragmática: tal como se pode ver na Tabela 3.10; 8 bits não são suficientes para codificar uma instrução.

E quanto à capacidade da memória de instruções, e número de bits que o respectivo endereço deve ter? Tal como na memória de dados, o endereço da memória de instruções tem de ser especificado no campo "constante" das instruções, em caso de salto. Logo,

continuamos limitados a 8 bits, pelo que poderão existir no máximo 256 instruções de 16 bits cada.

A conclusão é que a largura de dados de um processador também condiciona a capacidade máxima de memória de instruções suportada, pois os endereços acabam por ter de circular no mesmo caminho (pelo menos parcialmente) do que os dados.

#### ESSENCIAL

- A definição da largura de um processador é fundamental. Não só define a gama de numeros inteiros com que pode trabalhar directamente como tem implicações no número de bits dos endereços das memórias.
- A memória de dados deve ter a mesma largura do processador. Embora a capacidade da memória de dados possa ser arbitrariamente grande, acaba por ter de ser limitada pelo facto de os endereços desta memória terem de circular pelo mesmo caminho que os dados. Logo, tipicamente o número de bits do endereço da memória de dados não é superior à largura do processador.
- A largura da memória de instruções pode ser diferente da largura do processador, como é o caso do PEPE-8.
- A unidade de controlo tem a seu cargo a gestão dos sinais de controlo, PC e da interacção com a memória de instruções.

A unidade de dados gera o processamento dos dados, nomeadamente através da ALU e do registo A e a interacção com a memória de dados.

- Cada instrução no PEPE-8 demora apenas um ciclo de relógio a ser executada. Durante a execução de uma instrução os sinais necessários para a operação em causa (escrita no registo A, por exemplo) são preparados. Quando o relógio inicia um novo ciclo o PC muda e as acções preparadas durante a instrução anterior ocorrem nessa altura. Assumem-se registos disparados pelos flancos do relógio (*edge-triggered*).

#### NOTA

- O desenvolvimento da tecnologia tem permitido que os computadores façam tarefas cada vez mais complexas, e os programas que as implementam exigem cada vez mais das arquiteturas. Esta evolução é uma consequência essencialmente dos seguintes factores:
- Cada vez mais dados para processar, o que exige maior capacidade da memória de dados e por conseguinte mais bits para o endereço de dados;
  - Cada vez os programas têm mais instruções, o que exige maior capacidade da memória de instruções e por conseguinte mais bits para o endereço de instruções;
  - Manipulação de dados com cada vez mais bits (números maiores e/ou com mais casas decimais).

Assim, os processadores têm vindo sucessivamente a aumentar a sua largura de dados (8, 16, 32 e mais recentemente 64 bits), tal como ilustrado na Tabela 1.3, que termina na página 24. A de 8 bits usa-se apenas nos processadores de mais baixo custo, enquanto que a de 64 bits se usa sobretudo nos processadores de mais alto desempenho (e custo).

#### SIMULAÇÃO – FUNCIONAMENTO DO PEPE-8

Esta simulação ilustra o funcionamento detalhado do PEPE-8 (círcuito da Fig. 3.17), usando o Programa 3.2 e a informação contida na Tabela 3.8 e na Tabela 3.9. Os aspectos cobertos incluem os seguintes:

- Funcionamento do PC, quer em termos do endereçamento da memória de instruções, quer da sua evolução, normal e por saltos;
- Comportamento do PEPE-8 para cada uma das instruções do programa, com os respectivos sinais de controlo;
- Endereçamento das memórias, de dados e de instruções;
- Caminho percorrido pelos dados;
- Funcionamento global do circuito, com uma instrução executada em cada ciclo de relógio.

### 3.4 PROGRAMAÇÃO EM BAIXO NÍVEL DE UM COMPUTADOR

#### 3.4.1 INSTRUÇÕES EM VEZ DE SINAIS DE CONTROLO

Por "programação de um computador" entende-se a especificação das várias instruções que compõem um programa com um dado fim. Programar um computador é o mesmo que programar o processador, pois o que está em causa são as instruções que o processador irá executar. Só que este não faz nada sem as memórias, razão pela qual só o computador completo pode funcionar.

A Tabela 3.8 ilustra como um computador pode ser programado, especificando laboriosamente em cada instrução os valores de todos os sinais de controlo. Nos primeiros computadores era exactamente assim que se fazia, com enormes painéis de interruptores e uma montanha de cabos que algumas vezes ligavam a uma ficha, outras vezes a outra.

O problema é que isto é muito complexo e de muito baixo nível. O PEPE-8 é um processador extremamente simples, apenas com seis sinais de controlo (um processador moderno pode ter várias centenas de sinais de controlo internos!), e mesmo assim já é preciso muita atenção e cuidado para não nos esquecermos de um sinal ou trocarmos o valor de algum deles.

Felizmente, nem todas as combinações possíveis dos vários sinais de controlo correspondem a instruções úteis. De facto, das 256 possíveis (são seis sinais de controlo mas gastam 8 bits no total), apenas 15 são suficientemente interessantes a ponto de constituir instruções. Por exemplo, não é particularmente útil activar os sinais ESCR\_A e WR ao

mesmo tempo. Por outro lado, se não se está a activar o sinal ESCR\_A os valores dos sinais

- SEL\_A, SEL\_B e SEL\_ALU são irrelevantes. Isto significa que:
  - Só há 15 instruções diferentes e não 256, o que torna a programação mais simples;
  - Para especificar uma instrução não é preciso indicar os valores dos vários sinais de controlo mas apenas o número da instrução, normalmente conhecido por código de operação ou *opcode*, que neste caso precisa de apenas 4 bits (para 15 instruções);
  - A memória de instruções pode ter uma largura menor, pois basta especificar o *opcode* e não os sinais de controlo todos.

Cada combinação de valores dos sinais de controlo que seja interessante é assim codificada num número (o *opcode*) e memorizada na memória de instruções. Ao executar o programa, o processador vai lendo cada instrução e o respectivo *opcode* tem de ser descodificado e convertido na combinação de sinais de controlo que lhe corresponde, o que é conseguido com uma ROM de descodificação, tal como ilustrado pela Fig. 3.18. Esta ROM tem 4 bits de endereço (o *opcode*) e uma largura de palavra de 8 bits (sinais de controlo). Este é o circuito final do PEPE-8.

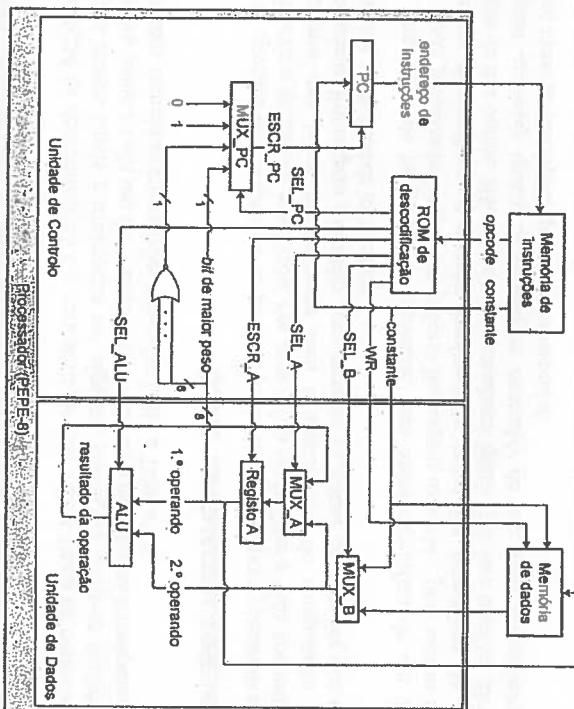


Fig. 3.18 - Circuito completo do processador (PEP-8, versão final), com descodificação dos opcodes das instruções por ROM

A Tabela 3.11 descreve as instruções interessantes e a ROM de descodificação.

**Tabela 3.11 - Conteúdo da ROM de descodificação de instruções.** "x" representa um bit cujo valor é indiferente

Note-se que só os sinais com valor fixo para uma dada instrução podem constituir o *opcode* e ser incluídos nesta ROM. A constante, cujo valor pode variar de instrução para instrução, é um parâmetro que se junta ao *opcode* para formar a instrução, mas não é usada para a descodificação, apenas como dado ou endereço.

A palavra da ROM de descodificação é a concatenação dos vários sinais de controlo. O valor de alguns desses sinais é irrelevante em algumas instruções. Note-se também a associação de um nome a cada um dos valores desses sinais, o que facilita a compreensão de todo o conjunto, usando as convenções indicadas junto à Tabela 3.8.

### 3.4.2 LINGUAGEM ASSEMBLY

No entanto, os *opcodes* são números e as pessoas preferem nomes, pois permitem mais facilmente traduzir a semântica envolvida. Foi com o fim de facilitar a vida aos programadores humanos que lidam directamente com a arquitetura do computador que apareceu a chamada linguagem *assembly*, que permite substituir os *opcodes* por uma notação simbólica, mais fácil de memorizar.

Não é uma linguagem de programação universal, mas antes específica de um dado processador. Cada processador tem a sua própria linguagem *assembly*, pois tem os seus próprios recursos de *hardware* e permite implementar operações específicas de uma forma diferente da dos outros. Não há dois processadores iguais, por isso também não há duas linguagens *assembly* iguais. A linguagem *assembly* de um dado processador é constituída por todas as instruções que ele sabe executar.

No caso do PEPE-8, cada instrução em linguagem *assembly* é representada por:

- Uma palavra-chave (conhecida por mnemónica) que identifica a instrução;
- Na maior parte das instruções, um parâmetro que indica o operando sobre o qual essa instrução actua, que pode ser interpretado, dependendo da instrução, como:
  - Um valor (operando imediato), ou
  - O endereço da célula de memória de dados onde o valor está (operando em memória), que se distingue do caso anterior por estar expresso entre parênteses rectos.

A Tabela 3.12 descreve a linguagem *assembly* do PEPE-8 e ilustra a sintaxe das instruções, ao mesmo tempo que mostra a correspondência entre cada mnemónica e o respectivo *opcode*. Note-se que algumas instruções têm a mesma mnemónica (para simplicidade de memorização), mas o tipo de operando é diferente (e os respectivos *opcodes* também).

Se o operando for imediato o seu valor é o especificado na instrução, enquanto que o endereço aparece entre parênteses rectos nas instruções que têm de ir à memória de dados buscar o operando propriamente dito.

OPCODE	INSTRUÇÃO ASSEMBLY	SIGNIFICADO EXTENSO	DESCRIÇÃO EM RTL	DESCRIÇÃO EM TEXTO
0	LD valor	Load (imediato)	$A \leftarrow \text{valor}$	Escreve no registo A a constante indicada por <i>valor</i> (especificada na instrução)
1	LD [endereço]	Load (memória)	$A \leftarrow M[\text{endereço}]$	Escreve no registo A uma cópia da célula da memória de dados indicada por <i>endereço</i>
2	ST [endereço]	Store (memória)	$M[\text{endereço}] \leftarrow A$	Escreve na célula da memória de dados indicada por <i>endereço</i> uma cópia do registo A
3	ADD valor	Add (imediato)	$A \leftarrow A + \text{valor}$	Soma o registo A com a constante indicada por <i>valor</i> e escreve o resultado no registo A
4	ADD [endereço]	Add (memória)	$A \leftarrow A + M[\text{endereço}]$	Soma o registo A com a célula da memória de dados indicada por <i>endereço</i> e escreve o resultado no registo A
5	SUB valor	Subtract (imediato)	$A \leftarrow A - \text{valor}$	Subtrai do registo A a constante indicada por <i>valor</i> e escreve o resultado no registo A
6	SUB [endereço]	Subtract (memória)	$A \leftarrow A - M[\text{endereço}]$	Subtrai do registo A a célula da memória de dados indicada por <i>endereço</i> e escreve o resultado no registo A
7	AND valor	AND (imediato)	$A \leftarrow A \wedge \text{valor}$	Efectua a conjunção lógica bit a bit do registo A com a constante indicada por <i>valor</i> e escreve o resultado no registo A
8	AND [endereço]	AND (memória)	$A \leftarrow A \wedge M[\text{endereço}]$	Efectua a conjunção lógica bit a bit do registo A com a célula da memória de dados indicada por <i>endereço</i> e escreve o resultado no registo A
9	OR valor	OR (imediato)	$A \leftarrow A \vee \text{valor}$	Efectua a disjunção lógica bit a bit do registo A com a constante indicada por <i>valor</i> e escreve o resultado no registo A
10	OR [endereço]	OR (memória)	$A \leftarrow A \vee M[\text{endereço}]$	Efectua a disjunção lógica bit a bit do registo A com a célula da memória de dados indicada por <i>endereço</i> e escreve o resultado no registo A
11	JMP endereço	Jump	$PC \leftarrow \text{endereço}$	Salta para a instrução indicada por <i>endereço</i>
12	JZ endereço	Jump if zero	$(A=0) : PC \leftarrow \text{endereço}$	Salta para a instrução indicada por <i>endereço</i> se o valor do registo A for zero
13	JN endereço	Jump if negative	$(A<0) : PC \leftarrow \text{endereço}$	Salta para a instrução indicada por <i>endereço</i> se o valor do registo A for negativo
14	NOP	No operation		Não faz nada. Apenas o PC é incrementado como normalmente

Tabela 3.12 - Conjunto de instruções suportada pelo PEPE-8 e que constituem a sua linguagem assembly

Esta tabela inclui ainda a seguinte informação:

- Significado por extenso (em inglês) das mnemónicas, que não são mais do que abreviaturas das operações, usando siglas fáceis de memorizar. Poderiam usar-se operações e mnemónicas em português, mas assim é mais fácil de reconhecer as mnemónicas de outros processadores que existam no mercado, que são dadas em inglês (as mnemónicas das operações mais comuns tendem a ser parecidas);
- Descrição em RTL das instruções, usando as convenções indicadas na Tabela 3.4, na página 128;
- Descrição em texto do significado das instruções, igual à da Tabela 3.11 e aqui reproduzida para melhor se poder fazer a correspondência.

A instrução NOP (*No operation*) é a única que não tem parâmetro e na realidade não faz nada a não ser avançar o PC. Usa-se quando se quer apenas gastar tempo (um ciclo de relógio), ou guardar espaço para uma futura instrução. Como todas as instruções têm de ter a mesma largura, a instrução NOP tem na realidade de ter lá os bits reservados para a constante, embora não sejam usados.

**NOTA**

Em geral, os processadores suportam outras instruções sem parâmetros, normalmente operações envolvendo apenas o valor do registo (como por exemplo negar todos os bits) ou de registo implicitamente associados, de forma fixa, a essas instruções, razão pela qual não é preciso especificar mais nada além do opcode.

Ao contrário do PEPE-8, que é muito simples, os processadores têm geralmente muitos registos do que simplesmente um (seção 4.1), o que significa que as suas instruções têm de ter pelo menos dois parâmetros (além da constante tem de se indicar qual é o registo envolvido).

**ESSENCIAL**

- A linguagem assembly é específica de cada processador e permite codificar uma longa lista de valores de sinais num só número, o opcode (código de operação). É uma notação mais compacta, mais simples e reduz os enganos de programação;
- Os valores que possam variar para uma mesma instrução (valor da constante, por exemplo), não podem ser incluídos no opcode e têm de constar, ao lado desse, em cada instrução;
- O assembler converte o programa assembly (texto) em números binários, instruções que o processador consegue executar directamente.

É importante reconhecer que a Tabela 3.11 e a Tabela 3.12 constituem duas perspectivas diferentes sobre a mesma entidade, o PEPE-8:

- A primeira reflecte a forma como as instruções estão implementadas, em termos dos sinais (é a perspectiva do hardware);
- A segunda expressa a funcionalidade das várias instruções e a forma como se especificam num programa, através da sintaxe (é a perspectiva do software).

Hoje em dia os computadores são tipicamente programados em alto nível, usando linguagens de programação como C, Java ou C#, que escondem os detalhes do hardware e tornam a programação independente da arquitetura do computador (o compilador trata de gerar as instruções correctas).

No entanto, há situações em que se pretende optimizar a funcionalidade ou o desempenho de um computador e há necessidade de programar em baixo nível, de forma específica e optimizada para uma dada arquitetura. Nestes casos usa-se a linguagem assembly, encarando-se o processador como uma caixa preta em que o importante não é os detalhes de implementação mas sim a funcionalidade, expressa pelo conjunto de instruções.

Esta visão do processador designa-se por arquitetura de conjunto de instruções, ou ISA (Instruction Set Architecture).

A perspectiva do hardware interno do processador interessa sobretudo em termos de implementação do circuito, mas quem programa deve conhecer os princípios básicos desta implementação, dado que tem um grande impacte no desempenho dos programas. As linguagens de programação de alto nível suportam a portabilidade funcional dos programas (isto é, a funcionalidade do programa não depende do computador em que é executado). No entanto, para maximizar o desempenho é necessário muitas vezes adaptar (em maior ou menor grau) a estrutura dos programas à arquitetura do processador.

Os compiladores das linguagens de alto nível geram código-máquina de uma forma específica para um dado processador e têm um conhecimento detalhado da forma como esta está implementada.

### 3.4.3 IMPLEMENTAÇÃO DAS INSTRUÇÕES

As figuras seguintes ilustram a arquitetura do PEPE-8 com os sinais activos (quer de controlo, quer de dados) em destaque no caso da execução dos vários tipos de instruções

referidos na Tabela 3.12, considerando o seu mapeamento nos sinais de controlo referidos na Tabela 3.11.

Para entender estas figuras deve-se ter em atenção o seguinte:

- São os sinais de controlo que comandam tudo o que acontece. Os diferentes comportamentos das várias instruções conseguem-se especificando combinações diferentes dos valores dos sinais de controlo;

- Os dados fluem (como se fossem água) para onde encontram uma passagem. Para os encaminhar basta abrir e fechar comportas (como por exemplo os sinais de selecção dos *multiplexers* e o sinal de escrita no registo A), tal como se faz nos canais de rega na agricultura;

Apenas os sinais relevantes estão destacados nas figuras. Por exemplo, ao escrever no registo A a sua saída muda de valor, o que provoca alterações em todos os sinais de dados que dela dependem (saída da ALU, por exemplo). Essas alterações não são destacadas por não terem influência no comportamento da instrução descrita na figura respetiva;

Num determinado ciclo de relógio, o PC selecciona uma dada instrução, o que faz preparar os sinais que alteram o estado do sistema (ESCR\_PC, WR e ESCR\_A). Se quando o relógio mudar é que o PC é alterado para o novo valor e, se for o caso, ou a memória ou o registo A são escritos.

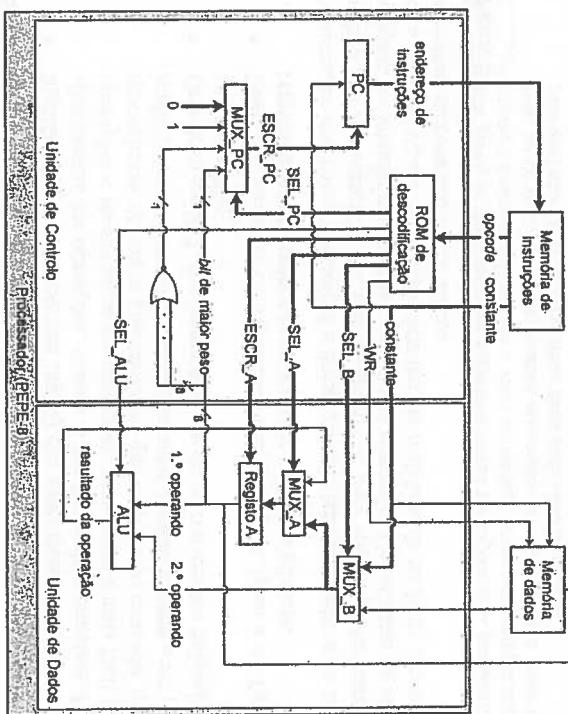


Fig. 3.19 - Sinais activos na instrução LD [valor]

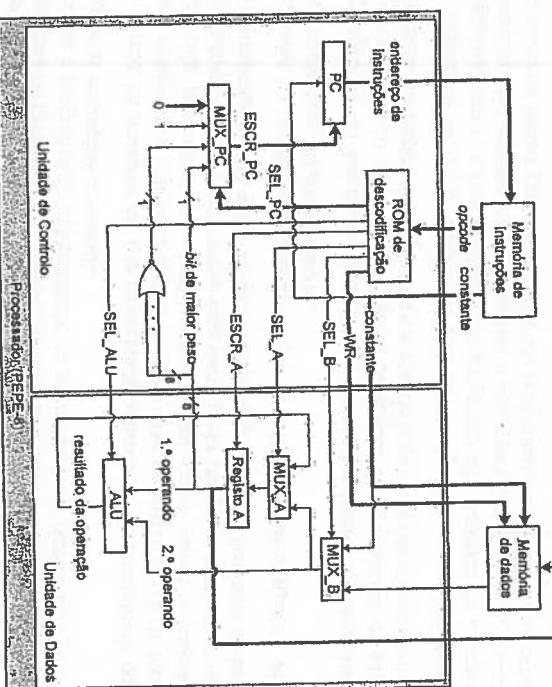


Fig. 3.21 - Sinais activos na instrução ST [endereço]

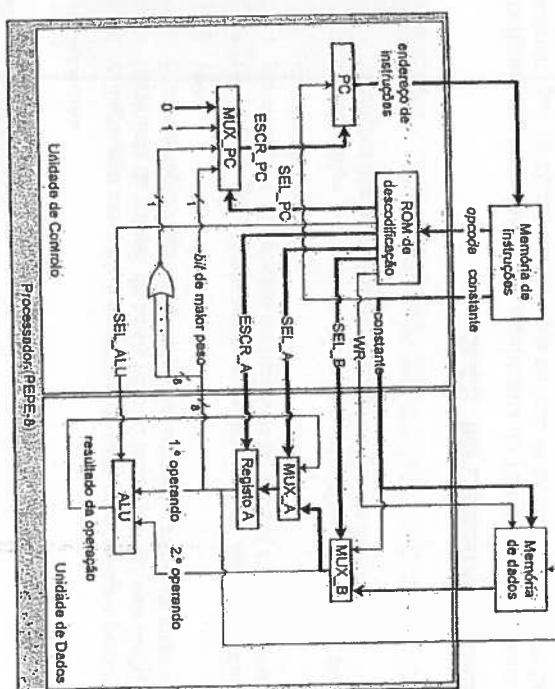


Fig. 3.20 - Sinais activos na instrução LD [endereço]

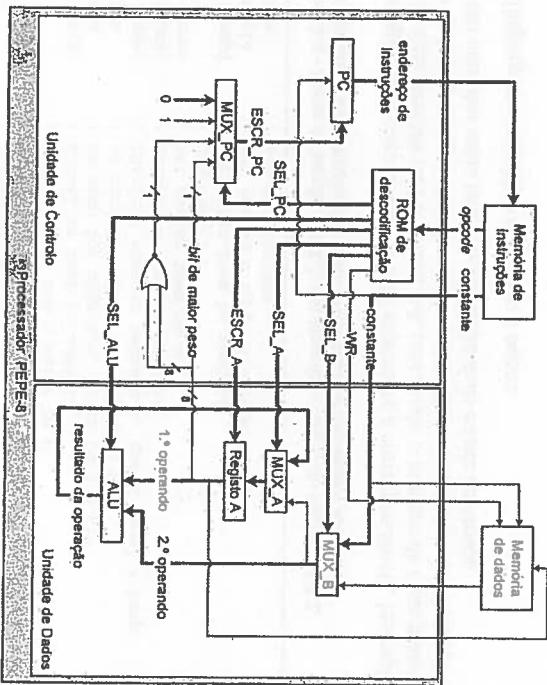


Fig. 3.22 - Sinais activos na instrução ADD valor

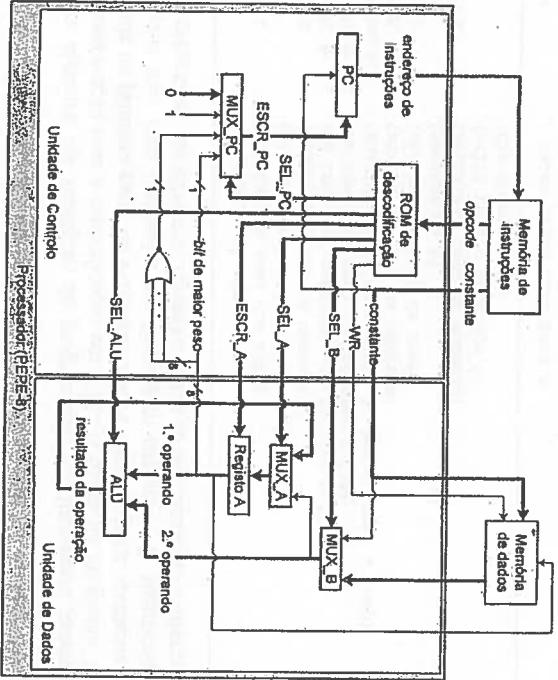


Fig. 3.23 - Sinais activos na instrução ADD endereço

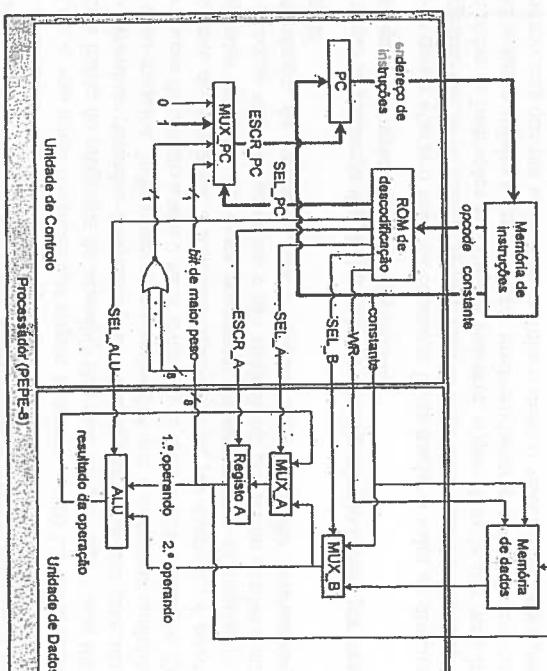


Fig. 3.24 - Sinais activos na instrução JZ endereço

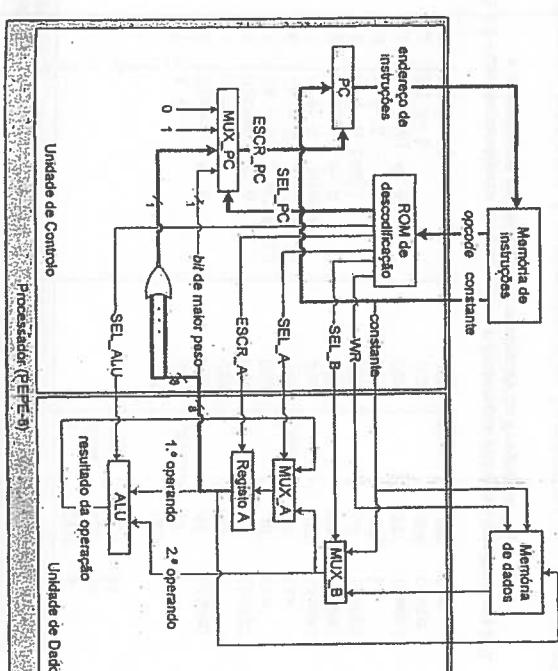


Fig. 3.25 - Sinais activos na instrução JZ endereço

### 3.4.4 PROGRAMAÇÃO EM LINGUAGEM ASSEMBLY

Esta secção ilustra algumas das vantagens da programação em linguagem assembly, efectuando uma comparação com a especificação em RTL, já usada no Programa 3.2 e que está reproduzida na primeira parte do Programa 3.4. A segunda parte apresenta um programa equivalente mas com instruções em linguagem assembly. O significado e resultado dos dois programas são idênticos. Apenas a forma de representação é diferente.

PASSO DO ALGORITMO	ALGORITMO (RTL)	ENDEREÇO NA MEMÓRIA	INSTRUÇÕES (ASSEMBLY)
1.	A ← 0	-00H	N EQU 0
2.	M[soma] ← A	01H	soma EQU 30
3.	A ← N	02H	temp EQU 31
4.	M[temp] ← A	03H	
(A < 0) : PC ← 13	04H		
5.	(se temp for negativo, salta para o fim)	05H	JN fim
6.	(A = 0) : PC ← 13	06H	JZ fim
7.	(se temp for zero, salta para o fim)	07H	ADD [soma]
8.	A ← A + M[soma]	08H	ST [soma]
M[soma] ← A	09H	LD [temp]	
A ← A + M[temp]	0AH	SUB 1	
(actualiza soma na memória)	0BH	ST [temp]	
(actualiza soma de novo)	0CH	JMP teste	
(decrementa A)	0DH	fim:	
9.	(vai buscar temp de novo)	11	JMP temp
10.	A ← A - 1	12	fim:
M[temp] ← A	13	JMP fim	
11.			
12.			
13.	PC ← 13		

Programa 3.4 - Soma da secção 3.3.2.3, em algoritmo (cópia do Programa 3.2, para facilidade de comparação) e em instruções de linguagem assembly

Para tornar o exemplo mais claro, a Tabela 3.13 estabelece a correspondência, instrução a instrução, entre as duas versões do Programa 3.4, mostrando o número de cada passo do algoritmo com o endereço em que cada instrução se localiza em memória (de instruções). Os comentários são omitidos nesta tabela por simplicidade e falta de espaço.

O programa em linguagem assembly é explicado a seguir.

Tabela 3.13 - Correspondência de instruções e endereços entre o algoritmo em RTL e o programa em linguagem assembly do Programa 3.4

Há algumas diferenças entre estas duas representações do programa que importa referir:

- A linguagem assembly do Programa 3.4 resolve um problema que até aqui já tinha sido referido mas não resolvido: a atribuição de valores numéricos às constantes simbólicas (seja numa instrução que requer um valor de dados, como LD constante, seja numa instrução que requer endereço, como LD [endereço]). As primeiras linhas do programa de assembly têm essa função. EQU é uma directiva para o assembler (também designada pseudo-instrução) que não gera instruções em código-máquina. Indica apenas ao assembler que a constante simbólica que aparece antes da directiva tem o valor indicado. É apenas uma definição. Durante todo o resto do programa, a constante simbólica (N, por exemplo) é equivalente ao seu valor numérico (4, neste programa). Normalmente, as constantes 0 e 1 usam-se como valores numéricos e não simbólicos, por serem usadas com extrema frequência. As vantagens das constantes simbólicas são essencialmente as seguintes:

- O nome da constante diz já algo sobre o seu significado (soma, por exemplo), algo que um simples número não permite;
- Se se quiser alterar o valor da constante basta mudar o valor na directiva EQU, enquanto que se se usarem apenas valores numéricos tem de se alterar o valor em todas as instruções em que ele aparecer, o que além de dar mais trabalho está bastante sujeito a erros (alterar inadvertidamente outro número não relacionado mas que por coincidência tinha o mesmo valor) e omissões (esquecer de alterar num ou mais sítios).

- Os *assemblers* suportam comentários, tipicamente desde o fim de uma instrução até ao fim da linha, começando com um carácter fixo. A sintaxe varia, mas o uso de „;” para iniciar o comentário é bastante frequente. Um comentário não gera instruções em código-máquina, sendo ignorado pelo *assembler*. Não passa de texto usado para explicar melhor o conteúdo e função da instrução. Normalmente, quase todas as instruções em *assembly* devem ter um comentário;
- O valor das constantes de endereço de programa, usadas nas instruções de salto, poderia ser especificado por uma directiva EQU. No entanto, tal implicaria que o programador andasse a contar instruções para saber qual o endereço para onde deveria saltar. Os *assemblers* pouparam essa tarefa através das etiquetas (*labels*), tais como inicio, teste e fim. À medida que vai convertendo as instruções *assembly* para instruções em código-máquina, o *assembler* vai contando os endereços e sabe em que endereço está cada instrução, o que permite atribuir automaticamente os valores correctos às etiquetas (não é o programador que faz isto). Depois é só usar as etiquetas como constantes simbólicas nas instruções de salto, tal como é feito na parte de *assembly* do Programa 3.4, e o *assembler* substitui-as pelo valor respectivo. Tem ainda a vantagem de que se forem introduzidas novas instruções, os valores das etiquetas são alterados automaticamente, sem necessidade de intervenção (é possibilidade de erro...) por parte do programador;
- Pelo contrário, no algoritmo em RTL tem de ser o programador a colocar explicitamente os números dos passos do algoritmo nas instruções de salto. Tem de contar instruções para determinar esses números e tem de ter muito cuidado quando inserir instruções novas ou apaga algumas, pois isso altera a numeração e algumas instruções de salto poderão ter de ser alteradas. Um esquema em que é muito fácil cometer erros!
- O *assembler* atribui o endereço 00H à primeira instrução do programa em *assembly*, pois os endereços da memória de instruções começam em 00H. Além disso, é neste endereço que o PEPE-8 executa a primeira instrução após ser inicializado (reset). No entanto, no algoritmo, os programadores numeram o primeiro passo com 1, pois os humanos estão habituados a que zero seja ausência e 1 seja a primeira instância de algo. Não haveria qualquer problema em o programa em *assembly* começar com os endereços em 00H, mas então inicio, teste e fim ficariam com os valores 0, 5 e 12, em vez de 1, 6 e 13, respectivamente, que são os valores correspondentes usados no algoritmo. Para evitar confusões e tornar os endereços iguais, tanto na parte de RTL como na parte de linguagem *assembly*, usou-se uma solução simples que consiste em colocar uma instrução que não faz nada (NOP) antes do programa em *assembly* propriamente dito, de forma a gastar o endereço 00H. Para além desta razão, não há qualquer outra justificação para o NOP neste exemplo.

#### SIMULAÇÃO – PEPE-8: PROGRAMAÇÃO EM ASSEMBLY

Esta simulação ilustra o funcionamento do PEPE-8 (círculo da Fig. 3.18) com programação em linguagem *assembly*, usando o Programa 3.4 e a informação da Tabela 3.11.

Os aspectos cobertos incluem os seguintes:

- Montagem do circuito com um módulo que simula o PEPE-8 ao nível das instruções;
- Funcionamento individual das várias instruções do PEPE-8, em termos dos recursos de hardware e sinais de controlo que envolvem;
- Execução do programa em ambiente de desenvolvimento:
  - Execução das instruções passo a passo (*single-step*), verificando a evolução dos registos A e PC;
  - Utilização de pontos de paragem (*breakpoints*);
  - Modificação manual do valor dos registos durante a execução do programa (após uma paragem motivada por execução passo a passo ou ponto de paragem).

#### 3.4.5 PROGRAMAÇÃO DO PEPE-8 EM ASSEMBLY: CONTAGEM DE BITS

Esta secção apresenta outro programa simples que ilustra o estilo de programação em linguagem *assembly*. A ênfase desta vez está nas operações lógicas.

O objectivo é contar o número de bits a 1 que um dado valor de 8 bits tem. O algoritmo a usar baseia-se numa máscara (página 126) que tem apenas um bit a 1, bit esse que vai mudando de posição desde a de menor peso (0000 0001) até à de maior peso (1000 0000). Em cada iteração, faz-se a conjunção lógica (E) da máscara com o valor a testar, o que permite isolar um dado bit, o da posição em que a máscara tem o bit a 1. Se o resultado der zero, o bit correspondente no valor a testar estava a 0. Se der diferente de zero, o bit estava a 1 e nesse caso incrementa-se um contador. No fim, o contador terá o número de bits que estavam a 1 no valor a testar.

A Tabela 3.14 ilustra a evolução do algoritmo através das suas grandezas principais.

POSIÇÃO DE TESTE	VALOR TESTE	VALOR MÁSCARA	BIT TESTADO	CONTADOR
0	0000 0001	0111 0110	0000 0000	Não 0
1	0000 0010	0111 0110	0000 0010	Sim 1
2	0000 0100	0111 0110	0000 0100	Sim 2
3	0000 1000	0111 0110	0000 0000	Não 2
4	0001 0000	0111 0110	0001 0000	Sim 3
5	0010 0000	0111 0110	0010 0000	Sim 4
6	0100 0000	0111 0110	0100 0000	Sim 5
7	1000 0000	0111 0110	0000 0000	Não 5

Tabela 3.14 - Evolução do algoritmo de contagem de bits a 1 em Valor. Os bits a negrito são os correspondentes à posição em teste em cada iteração

O número 76H (0111 0110) é usado como exemplo de valor a testar. O bit em teste varia desde a posição 0 até a 7, e a máscara vai tendo apenas um bit a 1 em cada uma destas posições. O truque para fazer evoluir a máscara é reconhecer que a máscara seguinte é o dobro da máscara anterior (o bit a 1 desloca-se uma posição para a esquerda, o que corresponde a multiplicar por 2), pelo que basta ir somando a máscara com ela própria para dobrar o seu valor.

O algoritmo é assim o seguinte:

1. contador ← 0
2. máscara ← 01H
3. Se !(máscara AND valor) = 0)
  - salta para 5
  4. contador ← contador + 1
  5. Se !(máscara = 80H) salta para 8
  6. máscara ← máscara + máscara
7. Salta para 3
8. Salta para 8

O programa correspondente em linguagem assembly pode ser o indicado no Programa 3.5. A primeira directiva EQU define qual o valor numérico a testar, que é um parâmetro deste programa. As restantes já são valores fixos, independentes do valor a testar.

```
; constantes de dados
valor EQU 76H ; valor cujo numero de bits a 1 é para ser contado
máscaraInicial EQU 01H ; 0000 0001 em binário (máscara inicial)
máscaraFinal EQU 80H ; 1000 0000 em binário (máscara final)

; constantes de endereços (na memória de dados)
contador EQU 00H ; endereço da célula de memória de dados que guarda
                    ; o valor correte do contador de bits a 1
máscara EQU 01H ; endereço da célula de memória de dados que guarda
                    ; o valor correte da máscara

; programa (memória de instruções)
início: LD 0 [contador] ; inicializa o registo A a zero
        ST [máscaraInicial] ; inicializa o contador de bits com zero
        LD [máscara] ; carrega valor da máscara inicial
        ST [máscara] ; actualiza na memória
teste: AND JZ [máscara] ; valor a bit que se quer ver se é 1
        valor ; se o bit for zero, tem de incrementar o contador
        ADD 1 [contador] ; vai buscar o valor actual do contador,
                        ; incrementa-o
        ST [máscara] ; actualiza o valor da máscara na memória
        ADD 1 [máscara] ; vai fazer mais um teste com a nova máscara
        SUB 1 [máscaraFinal]
        JZ fim ; se der zero, eram iguais e portanto já terminou
        ADD 1 [máscara] ; tam de carregar a máscara de novo
        ADD 1 [máscara] ; soma com ela própria para a multiplicar por 2
        ST [máscara] ; actualiza o valor da máscara na memória
        JMP teste ; vai fazer mais um teste com a nova máscara
fim:    JMP fim ; fim do programa
```

### Programa 3.5 - Implementação do algoritmo de contagem de bits a 1

Após este programa, o resultado (5, de acordo com a Tabela 3.14) estará disponível na célula de memória de dados com o endereço contador (ou 00H, em termos numéricos). Basta contar o número de bits a 0, em vez de contar o número de bits a 1, basta inverter o salto condicional a seguir à etiqueta teste, ficando desta forma:

teste: AND valor	; isola o bit que se quer ver se é 0
JZ máscara	; se o bit for zero, tem de incrementar o contador
JMP próxino	Passa à máscara seguinte
próximo: LD [contador]	vai buscar o valor actual do contador,
ADD 1	incrementa-o
ST [contador]	e actualiza de novo na memória
próximo: LD [máscara]	vai buscar de novo a máscara actual

Actualização da Tabela 3.14 para este caso, bem como o cálculo do novo resultado, é deixado como exercício para o leitor, fazendo parte integrante da simulação seguinte.

### SIMULAÇÃO - PEPE-8: CONTAGEM DE BITS

Esta simulação ilustra o funcionamento do PEPE-8 com o Programa 3.5 e segundo os passos da Tabela 3.14. Os aspectos cobertos incluem os seguintes:

- Montagem do circuito com um módulo que simula o PEPE-8 ao nível das instruções;
- Funcionamento individual das várias instruções do PEPE-8, em particular as operações lógicas;
- Execução do programa em ambiente de desenvolvimento:
  - Execução das instruções passo a passo (*single-step*), verificando a evolução dos registos A e PC;
  - Utilização de pontos de paragem (*breakpoints*);
  - Modificação manual do valor dos registos durante a execução do programa (após uma paragem motivada por execução passo a passo ou ponto de paragem);
  - Exploração da variante ao programa que conta o número de bits a 0 no valor a testar, em vez de contar o número de bits a 1.

## 3.5 PERIFÉRICOS

### 3.5.1 ESTRUTURA DO HARDWARE

Até aqui, cobrimos apenas os aspectos relacionados com as memórias (de dados e de instruções) e com o processador. No entanto, a Fig. 3.1 na página 115 indica que ainda falta referir os periféricos, a única forma de um computador interagir com o mundo que o rodeia.

Como exemplo orientador vamos usar o sistema de controlo de semáforo de peões já apresentado na secção 2.6.7.4, que utilizou a técnica da microprogramação para o implementar com um sistema programado ao nível dos sinais de *hardware*. O objectivo aqui é implementar exactamente o mesmo sistema, mas usando um processador (o PEPE 8) e programação em linguagem *assembly*, com periféricos que permitam controlar as luzes do semáforo e ler o estado do botão dos peões.

Do ponto de vista do processador, um periférico não passa de uma célula de memória de dados. Aliás, não há mesmo nada que indique ao processador se está a aceder a uma memória de dados ou a um periférico. Nem interessa. A operação (de leitura ou de escrita) é exactamente a mesma e a largura em *bites* também. A diferença reside no comportamento dos periféricos:

- Ao contrário das células de memória de dados, um periférico só suporta um tipo de operação: ou leitura ou escrita (mas não ambas<sup>25</sup>). Isso implica que haja dois tipos de periféricos:
    - De entrada, que só suportam leituras feitas pelo processador;
    - De saída, que só admitem escritas feitas pelo processador.
  - Ao contrário das células de memória, os *bits* de um periférico estão disponíveis para ligar a dispositivos exteriores (controláveis digitalmente com Os e 1s):
    - Um periférico de saída memoriza o valor escrito pelo processador, tal como uma célula de memória, mas, em vez de o valor ficar apenas memorizado internamente, os *bits* memorizados podem ser ligados a dispositivos externos.
    - Um periférico de entrada não tem memória. Ao contrário de uma célula de memória, que ao ser lida reproduz o valor memorizado, um periférico de entrada limita-se a ler o valor dos *bits* produzidos pelos dispositivos externos a que esse periférico liga.

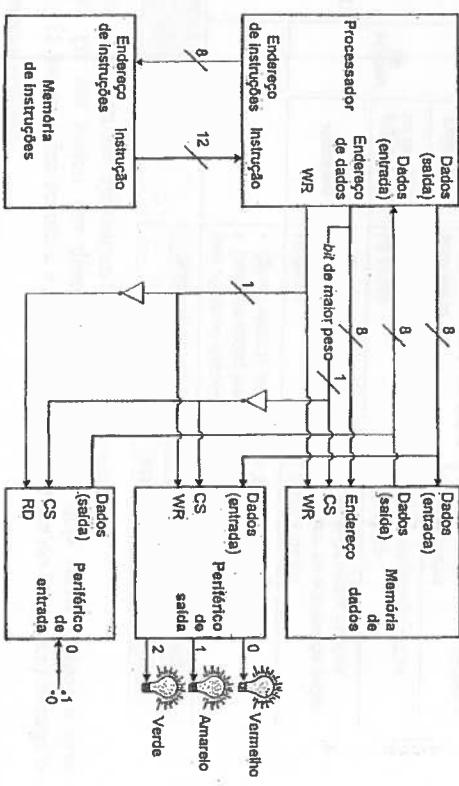
- Eléctricos, como uma resistência de um aquecedor ou de um termostato;
  - Elétrico-ópticos, como uma lâmpada de incandescência ou um LED;

Tudo o que seja controlável digitalmente por meio de 0s e 1s pode ser ligado a umas Electrò-acusticos, como um altifalante ou uma sirene.

de um periférico de saída que consegue controlar uma lampada de 60 W<sub>23</sub> ou um motor.

<sup>22</sup> Há dispositivos que suportam as duas operações mas que são constituídos por dois periféricos, um de entrada e outro de saída.

A Fig. 3.26 ilustra a ligação das memórias e dos periféricos ao processador. O processador não distingue a memória dos periféricos, pelo que estes têm de partilhar a mesma ligação ao processador. Isto levanta alguns problemas que é preciso resolver.



**Fig. 3-26 - PEPE-8 com ligação às memórias e aos periféricos. As ligações mais importantes têm a sua largura em bits especificada. O bit de maior peso do endereço de dados indica se o acesso é à memória ou aos periféricos**

Os aspectos mais relevantes das ligações na Fig. 3.26 são os seguintes:

- O periférico de saída liga apenas à saída de dados do processador. O periférico de entrada liga apenas à entrada de dados do processador. A memória de dados liga tanto à entrada como à saída de dados do processador;
  - Quer a memória de dados, quer os periféricos, têm 8 bits de largura. A memória de instruções tem 12 bits (4 bits de opcode e 8 bits de constante);
  - O periférico de saída liga às três lâmpadas dos semáforos, vermelha, amarela e verde, nos bits 0, 1 e 2, respectivamente. Os restantes 5 bits ficam com as saídas

de um elevador, por exemplo. Um *bit* de saída trabalha com 5 V (ou menos) de tensão continua e potências na ordem dos 50 mW (milésimos de Watt). Não se compara com os 220 V e 60 W de uma lâmpada de incandescência ou os 380 V (trifásicos) de um motor de 3000 W, por exemplo (e estas tensões são alternadas e não contínuas). Apenas os dispositivos electrónicos e de mais baixa potência (como um LED, por exemplo, que não precisa mais do que uns 30 mW para funcionar) podem ser controlados directamente por um *bit* de um periférico. Todos os outros, que precisam de mais potência, necessitam de uma interface específica, que liga ao *bit* por um lado e ao dispositivo por outro, fazendo o controlo em potência.

"no ar" (não ligadas). Por simplicidade, ignora-se aqui o circuito de controlo em potência das lâmpadas, que deverá existir entre cada bit e a lâmpada propriamente dita;

O periférico de entrada liga apenas ao interruptor do botão dos peões, que serve para pedir verde para os peões atravessarem. Este interruptor está normalmente a 0, indo a 1 apenas enquanto estiver a ser premido e voltando a 0 logo que for libertado. O interruptor liga ao bit 0 do periférico. Os outros 7 bits, como não são usados, devem ser ligados a 0, para que ao ler este periférico esses bits sejam lidos como 0. Não podem é ficar "no ar" (não ligados), pois o valor lido nesses bits seria aleatório (dependentes de ruído electromagnético, como o produzido por um motor eléctrico ou pelo interruptor da lâmpada de uma sala);

O periférico de saída é constituído essencialmente por um registo de 8 bits, em que os bits de saída do registo estão disponíveis para ligar as lâmpadas.

A escrita de dados por parte do processador, tanto na memória de dados como no periférico de saída, ocorre quando o processador activa WR. A leitura ocorre quando WR está inactivo. O periférico de saída não pode ser lido (portanto não faz nada com WR inactivo), mas por outro lado o periférico de entrada não pode ser escrito, pelo que funciona ao contrário: não faz nada quando WR está activo. Por isso, os dispositivos têm normalmente uma entrada de controlo, também activa a 0, para leitura, designada RD por contraste com WR. No circuito da Fig. 3.26, RD é simplesmente a negação de WR.

Assim:

- A memória suporta escritas e leituras, usando o WR para distinguir (activo, ou 0, é escrita; inactivo, ou 1, é leitura);
- O periférico de saída só suporta escritas (quando WR está inactivo não faz nada);
- O periférico de entrada só suporta leituras (quando RD está inativo, não faz nada).

O sinal WR (ou a sua negação, RD) liga a todos os periféricos. Se só usássemos este sinal, as consequências seriam:

- Num acesso de escrita do processador (WR activo), tanto a memória como o periférico de saída seriam escritos com o mesmo valor, o que não está correcto (os dispositivos têm de funcionar independentemente);
- Num acesso de leitura (WR inactivo, RD activo), tanto a memória como o periférico de entrada colocariam os seus próprios valores (serão iguais só por acaso) na entrada de dados do processador, o que corresponde a um conflito entre sinais que pode destruir os circuitos.

Nenhuma das situações é aceitável, pelo que a solução universalmente adoptada engloba os seguintes aspectos:

- Qualquer dispositivo desenvolvido para ligar a um processador tem sempre, para além do sinal de WR, outro sinal que indica se é a esse dispositivo que o processa-

dor quer aceder. Os nomes mais típicos para este último sinal são CS<sup>24</sup> ("Chip Select"), CE ("Chip Enable") ou simplesmente EN ("Enable"). Na prática, é um sinal de seleção que permite seleccionar apenas um de todos os dispositivos que ligam ao processador (só um dispositivo de cada vez poderá ter o seu sinal CS activo). Tipicamente, este sinal é activo a 0;

Os dispositivos que podem ser lidos (memória e periféricos de entrada) têm de inactivar a sua saída de dados (que liga à entrada de dados do processador) quando o seu sinal CS está inactivo. Cada bit desta saída deve poder ter 3 estados (saída tristate – ver secção 2.6.3): 0, 1 e desligado, caso em que não força nenhum valor. Desta forma, só um dispositivo colocará um valor na entrada de dados do processador (os outros terão a sua saída de dados desligada, ou inactiva), eliminando o conflito.

A Tabela 3.15 indica o que acontece a um dispositivo (memória ou periférico) quando o processador faz um acesso, nas quatro combinações dos dois sinal ligados a esse dispositivo, WR e CS (só um dispositivo poderá ter CS activo).

CS	SINAIS		WR
	INACTIVO (1)	ACTIVO (0)	
Activo (0)	MEMÓRIA	Lê dados	Sem efeito no dispositivo
		• Saída de dados activa; • Ignora entrada de dados.	• Saída de dados inactiva, se a tiver; • Ignora entrada de dados, se a tiver.
	PERIFÉRICO DE ENTRADA	Lê dados	Escreve dados
		• Saída de dados activa; • Ignora entrada de dados.	• Entram na entrada de dados; • Saída de dados inactiva.
Periférico de saída	WR	Sem efeito	Sem efeito
		• Ignora entrada de dados.	• Escreve dados • Entram na entrada de dados.

Tabela 3.15 - Efeito de um acesso do processador, em leitura e em escrita, num determinado dispositivo (memória ou periférico)

<sup>24</sup> Circuito tem de garantir de alguma forma que apenas um sinal de seleção de dispositivo é actuado de cada vez. O circuito da Fig. 3.26 é simples e tem apenas uma memória e um periférico de cada tipo (entrada e saída). Os periféricos não interferem um com o outro, uma vez que um só actua na leitura e outro na escrita.

Esta simplicidade permite usar apenas o bit de maior peso do endereço para distinguir se estamos a acceder à memória ou a um periférico (o sinal WR distingue os dois periféricos).

<sup>25</sup> Este é o nome escolhido para este sinal nos exemplos deste livro, por ser aquele que melhor traduz a sua semântica de seleção.

O espaço de endereçamento do PEPE-8 é de 256 endereços, de 0 a 255.<sup>25</sup> Se for feito um acesso a um endereço entre 0 e 127, o bit de maior peso do endereço está a 0, e o CS da memória é activado (mas o CS dos periféricos não) e a memória de dados é accedida. Se o endereço estiver entre 128 e 255, o bit de maior peso do endereço está a 1, o CS dos periféricos é activado (mas o CS da memória não) e é um periférico que é accedido dependendo do sinal WR. Em nenhum caso há uma escrita ou uma leitura em dois dispositivos ao mesmo tempo.

A Fig. 3.27 e a Fig. 3.28 ilustram os sinais envolvidos nos acessos em escrita, na memória e no periférico de saída.

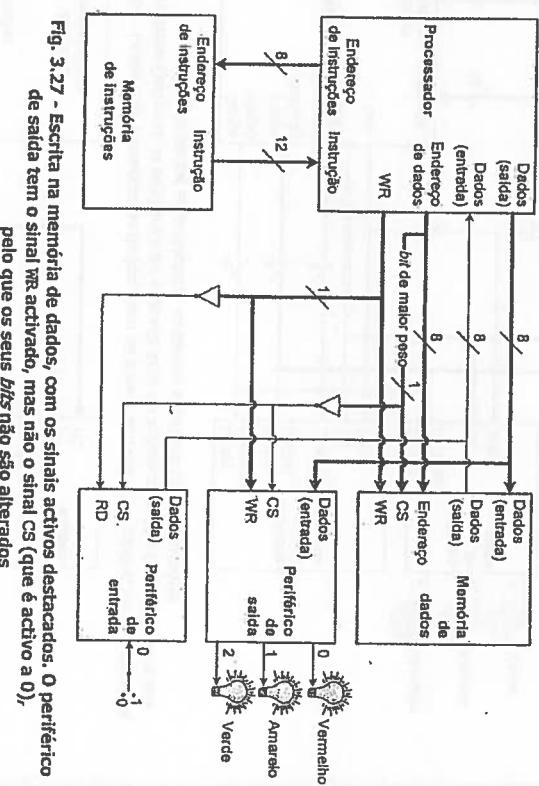


Fig. 3.27 - Escrita na memória de dados, com os sinais activos destacados. O periférico de saída tem o sinal WR activado, mas não o sinal CS (que é activo a 0), pelo que os seus bits não são alterados

O periférico de entrada não é um registo, porque não precisa de memorizar o estado dos bits a que liga. Quando for lido, basta apenas reproduzir o valor dos bits a que liga (neste caso, o interruptor).

O periférico de entrada não pode ser lido ao mesmo tempo que a memória de dados, senão há conflito de valores entre as saídas de dados da memória e do periférico (ambos ligam à entrada de dados do processador). Isto significa também que estas saídas de dados não podem ser normais, mas sim tristate (secção 2.6.3). O seu estado inactivo é de alta impedância (desligado), não provocando qualquer conflito.

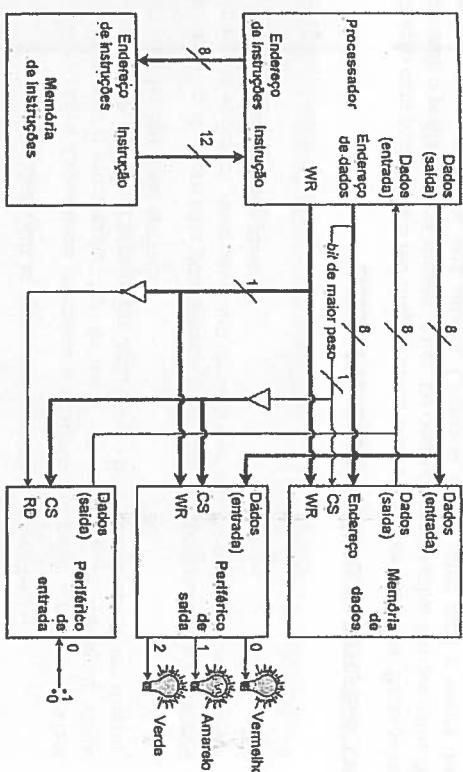


Fig. 3.28 - Escrita no periférico de saída, com os sinais activos destacados. A memória de dados tem WR activado, mas não CS (que é activo a 0), pelo que a célula endereçada não é alterada. O periférico de entrada tem CS activo mas não RD, pelo que não é accedido

Quando há uma leitura (WR no processador desactivado, RD no periférico de entrada activado), depende do valor do bit de maior peso do endereço de dados:

- Se for 0 (endereços de 0 a 127) – O CS da memória estará activo e o CS dos periféricos inactivo. Será uma leitura à memória (Fig. 3.29), que activará a sua saída de dados (a saída de dados do periférico de entrada estará inactiva);
- Se for 1 (endereços de 128 a 255) – O CS dos periféricos estará activo e o CS da memória inactivo. A saída de dados da memória ficará inactiva e o periférico de entrada activará a sua saída de dados (Fig. 3.30), que o processador irá ler.

Este esquema funciona bem, mas importa refletir os seguintes aspectos:

- A capacidade da memória de dados fica reduzida a metade (128 bytes, 0 a 127), pois a outra metade passa a estar reservada para os periféricos;
- Todos os 128 endereços de 128 a 255 accedem ao periférico de saída, pois este não liga nada aos restantes bits do endereço, apenas ao bit de maior peso. Isto significa que há um mau aproveitamento do espaço de endereçamento, pois o periférico apenas necessitava de um endereço e está a usar 128.

A secção 6.1.3, na página 416, apresenta soluções mais eficientes para este problema.

<sup>25</sup> Os endereços são sempre entendidos como números sem sinal, uma vez que endereços negativos não existem.

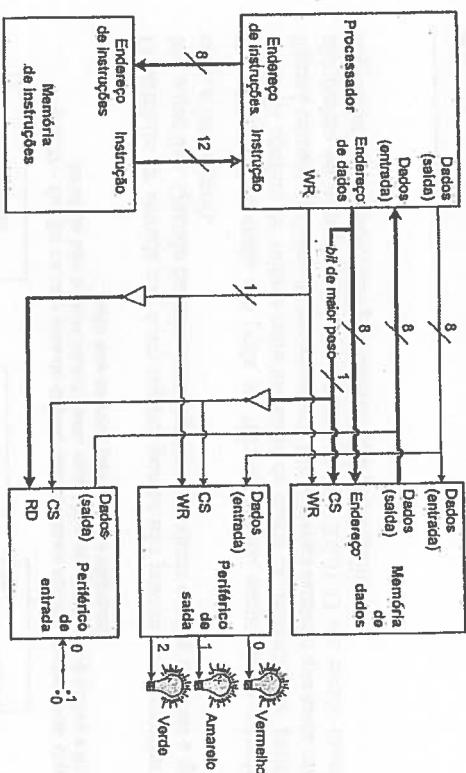


Fig. 3.29 - Leitura da memória de dados, com os sinais activos destacados. WR da memória está inactivo (leitura). O periférico de entrada tem RD activo mas não CS, pelo que a sua saída de dados está inactiva, evitando conflitos com a memória

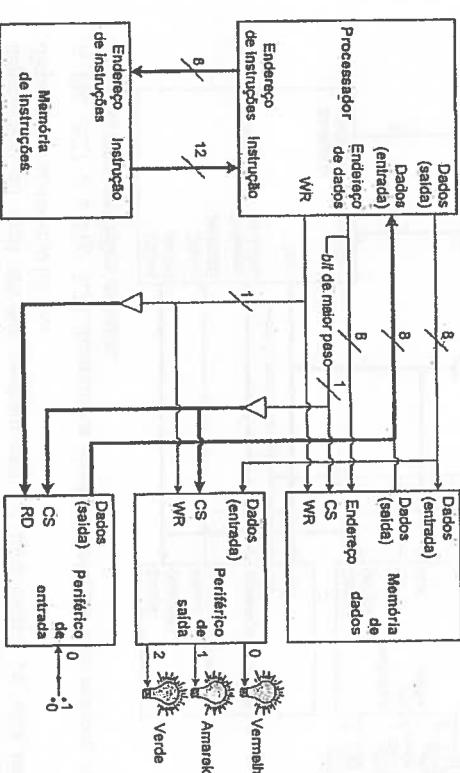


Fig. 3.30 - Leitura da memória de dados, com os sinais activos destacados. A memória tem a sua saída de dados inactiva, para evitar conflito com o periférico

### SIMULAÇÃO — FUNCIONAMENTO DOS PERIFÉRICOS

Esta simulação ilustra o funcionamento dos periféricos. O circuito a usar é basicamente o da Fig. 3.26, mas em que o processador é substituído por dispositivos que permitem controlar manualmente os sinais normalmente gerados pelo PEPE-8, para ilustrar o

funcionamento dos periféricos em detalhe e com controlo total. Os aspectos cobertos incluem os seguintes:

- Funcionamento básico do periférico de saída (com memorização do valor), e do periférico de entrada, com ênfase na sua interface tristate;
- Distinção entre acesso à memória e acesso aos periféricos, com ilustração do que acontece quando se activa simultaneamente a memória e um periférico, quer em leitura quer em escrita.

#### ESSENCIAL

- Os periféricos constituem o único meio de um computador comunicar com o meio exterior. Cada periférico aparece ao processador como uma memória especial, com muito poucas células (pode ser apenas uma), em que os bits estão disponíveis para ligação ao mundo exterior. Um periférico de saída é um registo que memoriza os bits nele escritos. Um periférico de entrada não tem memória, limitando-se a ler o estado dos bits do mundo exterior aos quais liga.
- Os periféricos lidam apenas com bits, de muito baixas tensões e potências, e não são capazes de controlar directamente dispositivos de altas tensões e potências (uma lâmpada de incandescência, por exemplo), sendo necessárias interfaces especiais para fazer a conversão.

- Os bits não usados de um periférico de entrada devem ser ligados a 0 (ou a 1). Não podem ficar "no ar".

- Os periféricos, de entrada e a memória de dados têm uma interface tristate com o processador para permitir ler ou da memória ou o periférico, sem conflitos.

- Original wr permite distinguir as operações de leitura das de escrita.

- Há vários dispositivos (memória e periféricos) ligados à mesma interface de dados com o processador, tem de haver um sinal (CS) de distinguir a qual dispositivo se dirige um acesso. Em cada acesso do processador só um dispositivo pode ter este sinal activado.
- Os sinais CS são derivados dos endereços. Quando o processador faz um acesso a um dado endereço, deve haver um circuito que active apenas o CS do dispositivo a que esse endereço pertence.

### 3.5.2 PROGRAMAÇÃO COM PERIFÉRICOS

A programação com periféricos é tão simples como com células de memória. Basta saber o endereço onde o periférico está situado. Aliás, tal como é preciso saber em que gama de endereços a memória de dados está situada. Chama-se a isto saber qual o mapa dos endereços.

No caso da Fig. 3.26, e atendendo à explicação anterior, o mapa de endereços de dados (não tem nada a ver com os endereços das instruções, que são separados) é o seguinte:

<b>DISPOSITIVO</b>	<b>GAMA DE ENDEREÇOS EM ESCRITA (WR ACTIVO)</b>	<b>GAMA DE ENDEREÇOS EM LEITURA (WR INACTIVO)</b>
Memória de dados	0 a 127	0 a 127
Periférico de saída	128 a 255	Nenhum (o sinal RD do periférico está inactivo)
Periférico de entrada	Nenhum (o sinal RD do periférico está inactivo)	128 a 255

Tabela 3.16 - Mapa de endereços de dados do circuito da Fig. 3.26e

Note-se que a gama de endereços do periférico de saída é a mesma que a do periférico de entrada, mas não interferem mutuamente porque um está activo só nas escritas e o outro só nas leituras.

Para qualquer um destes periféricos pode-se escolher um dos endereços da gama 128 a 255 (qualquer um serve pois o periférico é activado em qualquer endereço desta gama). Por isso, pode-se escolher o endereço 128, por exemplo, para ambos os periféricos.

### **3.5.2.1 USO DE FERIFÉRICOS DE SAÍDA**

O Programa 3.6 ilustra o uso de periféricos a partir de instruções em linguagem assembly. O circuito de base é o da Fig. 3.26, mas apenas é usado o periférico de saída, num ciclo de programação fixa.

O objectivo é implementar o semáforo simples já especificado na página 78, com uma máquina de estados que implementa uma sequência fixa verde-amarelo-vermelho, com tempos predefinidos.

Para evitar o problema de medir tempos (tratado apenas na secção 6.2.2, na página 469), usa-se uma frequência de relógio de 1 Hz (uma instrução por cada segundo) para o processador, o que permite derivar os tempos directamente das instruções executadas. Para alterar os tempos de cada cor no semáforo é preciso acrescentar ou retirar instruções, não sendo possível definir os tempos com constantes. É um esquema rudimentar, mas tem o condão de ser muito simples.

Assumem-se os seguintes tempos (curtos para serem mais facilmente testáveis):

- a) Amarelo – 2 segundos;

Há alguns aspectos neste programa que importa referir:

Só após a instrução `SI [semáforo]` é que os bits actualizados aparecem na saída do periférico. Só nessa altura o semáforo muda de cor. Na prática, uma cor inicia-se no princípio do ciclo de relógio da instrução com a etiqueta com o nome dessa saída.

cor. Por exemplo, note-se que o verde perdura mesmo quando o amarelo já está a

Os compassos de espera para cumprir os tempos indicados atrás são feitos com instruções `NOE` (que não fazem nada a não ser gastar um ciclo de relógio). Note-se que a execução de instruções `NOE` não consome tempo de relogio.

que a cor amarela não precisa de instruções `INC` porque o tempo de processamento é menor que o tempo de processamento da cor vermelha já esgota os 2 segundos (duas instruções) requeridos. A instrução `JMP` no início também gasta tempo, contabilizado na cor vermelha;

Quando o programa arranca, o periférico de saída ainda não foi inicializado. Isto quer dizer que durante as duas primeiras instruções o semáforo pode ter um valor aleatório, podendo mesmo estar totalmente apagado ou ter as três lâmpadas acesas.

```

; constantes de dados
vermelho EQU 01H ; valor do vermelho (lâmpada liga ao bit 0 - vermelho)
amarelo EQU 02H ; valor do amarelo (lâmpada liga ao bit 1)
verde EQU 04H ; valor do verde (lâmpada liga ao bit 2)

; constantes de endereços
semaforo DQU 80H ; endereço 128 (periférico de saída)

```

### Programa 3.6 - Implementação do semáforo simples

SINOPSE NO 37 - SEMÁFORO SIMPLESS

Esta simulação ilustra o funcionamento do Programa 3.6. O circuito a usar é o da Fig. 3.26. Os aspectos cobertos incluem os seguintes:

- Funcionamento do periférico de saída (com memorização do valor e actuação nas lâmpadas); Verificação dos tempos de cada cor e sua relação com as instruções do programa; Verificação do comportamento do periférico quando o programa arranca e antes de ser inicializado.

### 3.5.2.2 USO DE PERIFÉRICOS DE ENTRADA

Esta secção continua a usar os semáforos para ilustrar o uso dos periféricos, com temporizações dependentes de um botão a actuar pelo peão, e permite exemplificar os periféricos de entrada. O circuito de base continua a ser o da Fig. 3.26, usando agora os dois periféricos. O ciclo de relógio do processador de 1 Hz (uma instrução por segundo) mantém-se.

O objectivo é implementar um semáforo que normalmente está a verde para os carros e implementa um ciclo amarelo-vermelho-verde sempre que um peão carregue no botão (para poder atravessar a rua), com as seguintes temporizações:

- Verde – até o peão carregar no botão;
- Amarelo – 2 segundos;
- Vermelho – 7 segundos.

Esta especificação é uma simplificação do semáforo com botão de peões descrito na página 79. O programa correspondente em linguagem assembly pode ser o indicado no Programa 3.7, cujo funcionamento é explicado pelos comentários.

```
; constantes de dados
vermelho EQU 01H ; valor do vermelho (lâmpada liga ao bit 0 - ver
; a Fig. 3.26)
amarelo EQU 02H ; valor do amarelo (lâmpada liga ao bit 1)
verde EQU 04H ; valor do verde (lâmpada liga ao bit 2)
mascBotão EQU 01H ; máscara a usar para isolar o bit do botão no
; periférico de entrada (ligado no bit 0)

; constantes de endereços
semáforo EQU 80H ; endereço 128 (periférico de saída)
botão EQU 80H ; endereço 128 (periférico de entrada)

; programa
início: LD verde ; carrega o registo A com o valor para verde
ST [semáforo] ; actualiza o periférico de saída
semVerde: LD [botão] ; lê o valor do periférico de entrada
AND mascBotão ; isola bit do botão (força os outros a 0)
JZ semVerde ; se for zero, o botão não foi carregado
LD amarelo ; carrega o registo A com o valor para amarelo
ST [semáforo] ; actualiza o periférico de saída
semVermelho: LD [semáforo] ; carrega o registo A com o valor para vermelho
ST [semáforo] ; actualiza o periférico de saída
semVerm: NOP ; faz um compasso de espera
NOP ; faz um compasso de espera
NOP ; faz um compasso de espera
NOP ; vai fazer mais uma ronda
JMP início
```

Programa 3.7 - Implementação do semáforo com botão para peões

A Fig. 3.26 indica que:

- As lâmpadas vermelho, amarelo e verde ligam aos bits 0, 1 e 2 do periférico de saída, respectivamente;
- O botão liga ao bit 0 do periférico de entrada, em que 1 corresponde ao botão carregado.

À alguns aspectos neste programa que importa referir:

- Desde que o botão não seja usado muito frequentemente, este programa passa a maior parte do tempo no ciclo de leitura do periférico de entrada e teste ao valor do botão, ciclo esse (etiqueta semVerde) que demora 3 segundos (3 instruções);
- O periférico de entrada (a que o botão liga) só é lido pelo processador no fim da instrução LD [botão] (só nessa altura o registo A memoriza este valor). Assim, é possível que com este programa o peão carregue no botão e volte a largar sem que o programa dê por isso. Com este programa e com um relógio de 1 Hz, a solução é o peão carregar no botão até ver o semáforo passar para amarelo; Mesmo depois de o programa ler para o registo A o valor do periférico de entrada com o botão carregado, ainda têm de passar 4 segundos até o semáforo mudar para amarelo, devido às várias instruções (cada uma a demorar 1 segundo) até o amarelo aparecer nos bits do periférico de saída;

Outras soluções mais elegantes para este problema envolvem:

- Fazer leituras muito mais frequentes, ou seja, aumentar consideravelmente a frequência do relógio. No entanto, tal implica outro controlo do tempo que não pelo tempo de execução das instruções (secção 6.2.2, na página 469);
- Usar interrupções (secção 6.2.2, na página 462) para o processador detectar que o botão foi carregado, em vez de estar constantemente a testar o valor do bit correspondente no periférico de entrada.

A instrução AND mascBotão, que efectua a conjunção bit a bit do registo A e da constante mascBotão, destina-se a tornar o teste ao bit do botão independente dos outros 7 bits lidos pelo periférico de entrada. É dispensável se estes 7 bits estiverem fisicamente ligados a 0, mas a sua utilização é mais segura em caso de eventuais alterações ao programa, em que se passem a usar alguns desses bits do periférico de entrada para outros fins e deixem de estar sempre a zero.

#### EMULACÃO — SEMÁFORO DE PEÕES

Esta simulação ilustra o funcionamento do Programa 3.7. O circuito a usar é o da Fig.

- 3.26. Os aspectos cobertos incluem os seguintes:
  - Funcionamento do periférico de entrada com leitura do valor do botão;
  - Funcionamento ciclo de teste e papel da instrução AND mascBotão;
  - Verificação do tempo de resposta à actuação do botão;
  - Verificação dos tempos de cada cor e sua relação com as instruções do programa.

### 3.6 SOLUÇÕES ESPECÍFICAS OU GENÉRICAS?

Os sistemas de controlo de semáforos, simples e de peões, foram já implementados com três soluções diferentes:

- Máquina de estados sintetizada (secção 2.6.7.3, na página 83), com hardware específico para uma dada aplicação derivado a partir do diagrama de estados;
- Máquina de estados microprogramada (secção 2.6.7.4, na página 83), com uma estrutura de hardware básica e em que se controlam os diversos sinais de controlo explicitamente por meio de microinstruções numa ROM;
- Programa em linguagem *assembly*, com processador e periféricos (secção 3.5.2), em que a funcionalidade está expressa em termos de um número muito limitado de operações básicas (as instruções do microprocessador).

Uma questão legítima é saber quando é que se deve usar uma solução ou outra. A Tabela 3.17 sumariza as características fundamentais das três soluções. A diferença fundamental entre um sistema baseado num microprocessador e num circuito desenvolvido especificamente para uma dada aplicação é que o primeiro usa um circuito genérico, que pode implementar qualquer funcionalidade em software, enquanto que o segundo usa hardware específico, que precisa de ser alterado quando se quer fazer qualquer alteração à sua funcionalidade.

CARACTERÍSTICA	MÁQUINA DE ESTADOS		
	SINTETIZADA	MICROPROGRAMADA	MICROPROCESSADOR
Funcionalidade em	Hardware	ROM	Software
Potencial de funcionalidade	Pequeno	Grande	Muito grande
Trabalho de implementação	Grande	Médio	Pequeno
Facilidade de alteração	Pequena	Média	Grande

Tabela 3.17 - Comparação das características das várias soluções de implementação de sistemas de controlo

Ao contrário do hardware, o software é facilmente alterável e muito mais fácil de desenvolver para aplicações mais complexas. Isto faz com que hoje em dia já não faça sentido desenvolver hardware específico para uma dada aplicação<sup>26</sup>, preferindo-se o hardware genérico que depois é programado. A máquina de estados microprogramada inclui a flexibilidade do software mas é de muito baixo nível e difícil de programar em aplicações minimamente complexas.

Por estas razões, os microprocessadores são a escolha mais correcta, mesmo para os sistemas muito pequenos. O custo dos microprocessadores está tão baixo que aparecem a

incluídos em qualquer sistema com um mínimo de funcionalidade, mesmo brinquedos. A secção 5.9.4.2, na página 401, apresenta mais alguns detalhes sobre estes sistemas mais pequenos.

#### ESSENCIAL

- Para usar os periféricos é preciso saber qual o mapa dos endereços, que indica em que endereços os periféricos estão disponíveis.

O acesso aos periféricos faz-se da mesma forma e com as mesmas instruções que no acesso à memória. No entanto, a interacção com o mundo exterior obriga normalmente a respeitar certas temporizações, tipicamente bastante lentas (milissegundos, segundos ou mesmo mais lentas) face à escala de perfodos de relógio que a tecnologia hoje em dia permite (nanosegundos ou mesmo menos), pelo que este factor tem de ser tido em conta. Ou se reduz a frequência do relógio ou o processador tem de esperar que chegue o tempo de interagir com os periféricos.

- Independentemente da solução usada, e salvo aplicações muito específicas tipicamente, de circuitos de frequência de relógio mesmo muito alta, o microprocessador é a implementação por exceléncia de qualquer sistema digital, em que o hardware é genérico, muito semelhante para as várias aplicações, e em que a funcionalidade está essencialmente no software, mesmo nos sistemas mais simples. Isto facilita não apenas o desenvolvimento de uma dada aplicação mas também a sua manutenção (correcção de erros ou melhorias na funcionalidade).

### 3.7 CONCLUSÕES

Os sistemas baseados em processadores são mais flexíveis, mais fáceis de desenvolver e capazes de funcionalidades impensáveis em sistemas desenvolvidos à medida. Mesmo com processadores muito simples, como o PEPE-8, é possível desenvolver sistemas com aplicações úteis e bem interessantes.

Actualmente existe uma grande variedade de processadores comerciais, com arquiteturas mais ou menos complexas, sendo capazes de processar números binários não apenas de 8 mas também de 16, 32 e 64 bits. Cada uma tem um arsenal de técnicas e recursos para executar programas o mais rápido possível, dentro dos seus objectivos.

Os processadores de menor largura são mais simples e mais baratos mas também de menores capacidades e usam-se em aplicações com menores requisitos. Os de maior largura são mais potentes, mas também mais dispendiosos, e usam-se em aplicações que exigem maiores recursos e poder de cálculo. No entanto, todos usam a mesma arquitetura básica, segundo os princípios enunciados para o PEPE-8, com unidade de dados e de controlo, memória de dados e de instruções, contador de programa e um conjunto de instruções, a que corresponde uma linguagem assembly.

<sup>26</sup> Com exceção de casos muito particulares, como por exemplo um circuito digital de frequência elevada que exija uma implementação optimizada em hardware.

O PEPE-8 tem provavelmente o circuito mínimo que se pode ter para lhe podermos chamar processador. Com um espaço de endereçamento de 8 bits não é possível ir muito longe. Até o processador comercial mais pequeno é mais poderoso do que o PEPE-8, incluindo características de que ainda nem falámos. No entanto, mais do que apresentar uma arquitectura completa, este capítulo pretende demonstrar as razões pelas quais os processadores são como são, os blocos necessários tendo em conta o funcionamento das memórias e dos registos, os sinais de controlo típicos e quais as instruções básicas necessárias, que agrupam as combinações de sinais mais interessantes e evitam ter de se programar especificando sempre todos os sinais de controlo. Mostrou-se que basta apenas um registo, tal como especificado pela arquitectura original de von Neumann, embora isso obrigue a acessos constantes à memória.

Por outro lado, um processador pode funcionar apenas com memória, mas fica isolado do mundo. Os periféricos são fundamentais num computador, e é por isso que este capítulo apresentou exemplos concretos de um sistema completo com interacção com o mundo exterior, capaz de implementar aplicações reais e com aplicação prática.

Este capítulo fez assim uma introdução aos mecanismos básicos que permitem aos computadores controlar o mundo, dando uma panorâmica geral do papel de cada um dos seus componentes fundamentais. Para já, temos um sistema completo e funcional, o que foi demonstrado por simulação, que constitui a parte prática da estratégia pedagógica deste livro. Muito mais há a descobrir, em termos do reforço das capacidades, funcionalidade e desempenho do processador, o que será feito nos próximos capítulos.

### 3.8 EXERCÍCIOS

- 3.1** Indique quantos bits de endereço devem ter as RAMs com as seguintes características:
- 128 KBytes de capacidade e ligada a um processador de 8 bits;
  - 4 Mbits de capacidade e ligada a um processador de 16 bits.
- 3.2** Um computador tem uma RAM de 8 KBytes com 13 bits de endereço. Que largura de dados tem o processador?
- 3.3** Indique quais as limitações de um computador que, tanto para a memória de dados como para a de instruções, use:
- ROM;
  - RAM.
- 3.4** Explique a diferença entre capacidade de uma RAM em células e em bytes. Exemplifique, calculando essa capacidade para uma RAM de 2 Mbits ligada a um processador de 16 bits.
- 3.5** No circuito da Fig. 3.26, ligue o periférico de saída a um descodificador e mostrador de sete segmentos (Fig. 2.14, na página 52), substituindo as lâmpadas (LEDs). Use um relógio de 1 Hz para o PEPE-8.

- a) Simule um pequeno programa que circule o mostrador de 0 a F, tal como acontece na Fig. 2.32, na página 71;
- b) Explique porque é que o mostrador não evolui de segundo a segundo e implemente a solução para que tal aconteça (aproximadamente).

- 3.6** Simule um *multiplexer* de duas entradas com o PEPE-8. Use o circuito da Fig. 3.26, com três interruptores no periférico de entrada. Um serve de sinal de seleção para o *multiplexer* e os outros como entradas, cujas variações de valor o LED no bit 0 ou 1 (dependendo do sinal de selecção) do periférico de saída deve reproduzir. O programa deve estar em ciclo. Explique porque é que com um relógio de baixa frequência (1 Hz, por exemplo) o programa não funciona bem.
- 3.7** Simule um trinco D (Fig. 2.19, na página 59, e Fig. 2.22) com o PEPE-8. No circuito da Fig. 3.26, use dois interruptores (um é o relógio da báscula e outro a entrada D) e dois LEDs (saídas Q e  $\bar{Q}$ ). Para o PEPE-8, use um relógio com uma frequência elevada face à escala humana (50 Hz, por exemplo).

- 3.8** Idem, mas para uma báscula D (Fig. 2.22), activado no flanco:
- Ascendente;
  - Descendente.

- 3.9** Altere o Programa 3.6 para incluir mais dois LEDs, de forma a implementar um semaforo para os peões, tal como referido na página 79.

- 3.10** Na Fig. 3.26 destaque os sinais activos (à semelhança da Fig. 3.27, por exemplo) quando o PEPE-8 estiver a executar cada uma das instruções seguintes (sugestão: imprima a figura a partir do site de apoio ao livro):
- |              |             |
|--------------|-------------|
| a) ADD 23H   | f) LD [34H] |
| b) JMP 34H   | g) ST [84H] |
| c) SUB [23H] | h) LD [A7H] |
| d) LD 34H    | i) JZ 23H   |
| e) NOP       | j) ST [5AH] |

- 3.11** No circuito da Fig. 3.26, mediram-se alguns sinais, indicados nas situações seguintes (independentes entre si). Para cada uma delas indique uma das instruções (inâmérica e valor da constante, em hexadecimal) que o PEPE-8 poderá executar (originando esses sinais). A solução não é única.
- Periférico de entrada (RD=1, CS=0), Memória de Dados (entrada=6AH);
  - Periférico de saída (RD=0, CS=1), PEPE-8 (saída=FFH, endereço=7FH);
  - Periférico de entrada (RD=0, CS=1), Memória de Dados (saída=3DH);
  - Memória de dados (CS=1, WR=1, saída=ADF).

- 3.12** Converta para linguagem assembly as seguintes instruções em código-máquina do PEPE-8 (sem constantes simbólicas).

- |         |         |         |
|---------|---------|---------|
| a) 1C3H | d) EEEH | g) 843H |
| b) B3FH | e) COEH | h) 2F4H |
| c) 45AH | f) 0FFH | i) 6EAH |

**3.13** Explique o efeito prático (em termos de processamento e transferência dos dados das seguintes sequências de instruções do PEPE-8, quando executadas no circuito da Fig. 3.26.

- a) LD [83H]; ST [F0H]
- b) LD 10H; ADD [A0H]; ST [80H]
- c) LD 80H; ADD [80H]; SUB [80H]; ST [80H]

**3.14** O PEPE-8 tem apenas 15 instruções e uma codificação livre (Tabela 3.12). Uma instrução muito útil e imprescindível em qualquer processador é a que permite deslocar um registo de  $N$  bits para a direita ou esquerda. Explique todas as alterações que teria de fazer ao PEPE-8 para ele passar a suportar a instrução SHIFT valor, cujo efeito seria um deslocamento do registo A de valor bits para a direita. Indique as alterações necessárias no hardware e a combinação de sinais internos que permitiria executar esta funcionalidade (última linha da Tabela 3.12).

**3.15** Suponha agora que a instrução livre é MUL [endereço], com significado correspondente ao ADD, mas relativo à multiplicação. Escreva um programa para o PEPE-8 (com MUL) que consiga calcular a expressão  $A \times B + C \times D$ , em que os operandos são células de memória. Se precisar, pode usar outras células de memória.

**3.16** Usando o circuito da Fig. 3.26 como base, coloque LEDs em todos os bits do periférico de saída. Escreva um programa que faça sequencialmente várias células de RAM e as escreva no periférico de saída, voltando depois ao princípio. No simulador, inicialize manualmente estas células de RAM, com valores adequados para formar padrões de evolução dos LEDs ao longo do tempo (por exemplo, um só LED ligado que circula da direita para a esquerda e depois ao contrário). Controle o ritmo de evolução dos LEDs com o período do relógio do PEPE-8.

**3.17** Imagine que implementou um interruptor electrónico com o PEPE-8, que passa a fazer parte da sua máquina de lavar roupa. No circuito da Fig. 3.26, o interruptor é de pressão (só está a 1 enquanto estiver a ser premido). Quando carrega no interruptor, o LED no bit 0 do periférico de saída acende-se. Se carregar de novo, o LED apaga-se, e assim sucessivamente.

- a) Explique porque é que a memória de instruções não pode ser RAM;
- b) Faça o programa que implementa este comportamento;
- c) Simule este programa. Se o LED acender e apagar muito rapidamente enquanto carrega no botão, explique o que está a acontecer. Se não for o caso, explique o que fez para que tal não aconteça.

## 4 ARQUITECTURA BÁSICA DE UM PROCESSADOR

É tentador dizer que o processador é o módulo mais importante de um computador. O facto é que sem memória um computador pouco ou nada faz, e sem periféricos nem sequer se consegue comunicar com ele para o programar. O que é indiscutível é que o processador é o módulo mais complexo e o grande mestre que orquestra todo o computador.

O capítulo 3 introduziu a primeira versão de um processador, o PEPE-8, de 8 bits, cuja simplicidade esconde limitações que o capítulo 4 agora explica. Para as ultrapassar, apresenta-se aqui um novo processador, o PEPE (sem “-8”). O facto de agora ter 16 bits é a menor das diferenças. Trata-se de um processador que já suporta as características fundamentais que se podem encontrar nos processadores comerciais modernos, mas ao mesmo tempo refém a simplicidade que é fundamental para uma explicação didáctica.

Os aspectos mais profundos do PEPE, quer em termos de funcionalidades mais especializadas, quer em termos de implementação, serão objecto de estudo nos capítulos seguintes. Este capítulo descreve a estrutura e funcionamento básicos do PEPE, segundo a mesma aproximação já adoptada no capítulo 3: partir dos requisitos para as soluções, construindo a arquitectura em conjunto com o leitor e evitando apresentar soluções consumadas sem um racional que as suporte.

Ao desenhar um processador, um dos objectivos mais importantes é o desempenho. Um processador tem de executar os programas com a maior rapidez possível. A ciéncia da arquitectura de computadores resume-se em grande parte à utilização de inúmeras pequenas técnicas, em que cada uma constitui um pequeno melhoramento face à arquitectura de von Neumann (seção 1.4) mas que no seu conjunto conseguem executar um programa várias vezes mais depressa, para a mesma frequência do relógio. Por outro lado, os computadores actuais têm arquitecturas complexas que lidam com números binários de 32 ou mesmo 64 bits, programados em linguagens de alto nível (C, Java, C#, etc.) com recurso a compiladores que escondem muitos dos detalhes e problemas do processador.

A arquitectura do PEPE, ao invés, é pedagógica. O seu objectivo fundamental é ensinar as técnicas de arquitectura de computadores mais relevantes, com o desempenho em segundo lugar. Tem de ser fácil de programar directamente em linguagem assembly, ao nível dos recursos do hardware, o que na prática limita a sua largura a 16 bits (é difícil lidar manualmente com números de 32 bits). Outro requisito é incluir as várias técnicas existentes em vários processadores comerciais, de modo a poder exemplificá-las, e não apenas as que maximizam o desempenho. Finalmente, tem de ser realista e implementável realmente. O compromisso e o equilíbrio resultantes são descritos nas secções seguintes.



```

; constantes de dados
valor      EQU    76H ; valor cujo número de bits a 1 é para ser contado
máscaraInicial EQU    01H ; 0000 0001 em binário (máscara inicial)
máscaraFinal   EQU    80H ; 1000 0000 em binário (máscara final)

; utilização dos registos:
; A - valores intermédios
; B - valor actual do contador de bits a 1
; C - valor actual da máscara

; endereços
contador  EQU    00H ; endereço da célula de memória que guardará
                     ; o número de bits a 1 no fim da contagem

; programa
início: MOV    B, 0           ; inicializa o contador de bits com zero
        MOV    C, máscaraInicial ; coloca também a máscara inicial no registo A
        AND    A, valor         ; isola o bit que se quer ver se é 1
        JZ     próximo          ; se o bit for zero, passa à máscara seguinte
        ADD    B, 1             ; o bit é 1, incrementa o valor do contador
próximo: CMP   C, máscaraFinal ; compara com a máscara final
        JZ     acaba           ; se forem iguais, já terminou
        ADD    C, C             ; soma máscara com ela própria para a
        ADD    C, C             ; multiplicar por 2
        teste   JMP   [contador], B ; vai fazer mais um teste com a nova máscara
acaba:   MOV    fin            ; se for necessário, pode armazenar o numero
                     ; de bits a 1 numa célula de memória
fin:     JMP   fin            ; fim do programa

```

Programa 4.2 - Se o PEPE-8 tivesse três registos, A, B e C, o Programa 4.1 poderia

Há uma série de alterações importantes que devem ser explicitadas:

- Cada instrução tem agora dois operandos (com exceção dos saltos), separados por vírgulas, pois o 1.º operando já não é sempre o registo A;
  - Uma convenção importante é que o 1.º operando (o da esquerda, portanto) desempenha também o papel de resultado, tal como se pode ver em algumas instruções que podem servir de exemplo (cujo significado em RTL é apresentado como comentário):
- ```

MOV   B, 0           ; B ← 0
MOV   C, máscaraInicial ; C ← máscaraInicial
MOV   A, C           ; A ← C
AND   A, valor       ; A ← A ∧ valor
ADD   B, 1           ; B ← B + 1
ADD   C, C           ; C ← C + C
MOV   [contador], B  ; M[contador] ← B

```
- O 1.º operando nunca pode ser uma constante (porque esta não pode ser alterada), e portanto não pode servir para guardar o resultado;

Nas operações de leitura e escrita da memória já não basta indicar apenas o endereço da célula a aceder. Agora também tem de se indicar o registo envolvido, pois pode já não ser o registo A. Nestas condições, as instruções LD (*load*) e ST (*store*) podem ser substituídas com vantagem por uma única instrução de utilização mais geral, MOV *destino*, *origem* (de move, ou mover, em português), que designa uma transferência (cópia) de dados em que a origem e o destino se indicam sempre explicitamente. Já se vêem aqui alguns exemplos, mas a secção 4.10.4 dá mais detalhes sobre esta instrução;

Também os saltos condicionais sofrem uma mudança com esta alteração, embora seja pouco aparente no Programa 4.2. É que já não se pode usar o valor do registo A para tomar a decisão de saltar ou não, pois o resultado da operação que sirva de base a essa decisão pode ser armazenado noutra registo que não o A. A solução típica é definir um registo específico (o Registo de Estado, ou RE) que memorize, após cada operação, as condições usadas pelas instruções de salto condicionais (imediatamente zero e negativo) e que decorrem do resultado dessa operação. Há algumas instruções que não alteram esse registo (como, por exemplo, as instruções de salto condicionais). Mais detalhes serão dados nas secções 4.7 e 4.9;

A instrução CMP (*Compare*, ou comparar, em português) constitui apenas uma melhoria ao Programa 4.1, pois substitui a instrução SUB (subtração) com vantagens. Faz também a subtração, para comparar, mas não armazena o resultado da subtração, evitando destruir o valor do registo. Se o fizesse, tinha de se guardar o valor da máscara actual ainda noutra registo ou então na memória. Como resultado, a instrução CMP memoriza apenas a informação necessária para o salto condicional que vem a seguir, ou seja, altera o Registo de Estado. Isto está relacionado com o ponto anterior e será discutido na secção 4.7.

O que é necessário fazer no *hardware* para suportar vários registos é substituir o registo A pelo chamado banco de registos, tal como indicado na Fig. 4.2, que apresenta a estrutura interna do processador com um banco de registos. Já não é o PEPE-8, com a sua simplicidade, mas ainda tem alguns problemas, pelo que não passa de uma versão intermédia a caminho do PEPE. No entanto, é o suficiente para descrever as instruções básicas do PEPE, que constituem o tema deste capítulo. Nesta figura há algumas evoluções importantes face ao PEPE-8 que importa referir:

- O banco de registos é um conjunto de registos (no fundo, uma pequena memória) com a particularidade que consegue ler dois registos ao mesmo tempo (tem só uma entrada, para escrita, mas duas saídas, para leitura);
  - Os sinais REG\_1 e REG\_2 são os índices (dentro do banco de registos) dos registos que servem como 1.º e 2.º operandos, respectivamente. No fundo, são os endereços de cada registo dentro do banco de registos. Nada impede que REG\_1 e REG\_2
- <sup>17</sup> Note-se que o destino é sempre representado do lado esquerdo, para a ordem ficar coerente com a instrução RTL que lhe é equivalente: destino ← origem.

sejam iguais, caso em que o mesmo valor é usado para os dois operandos (tal como já aconteceu na instrução ADD C, C);

O resultado das operações da ALU é escrito no mesmo registo que continha o primeiro operando (indicado por REG\_1). O multiplexor MUX\_RES (de resultado) permite seleccionar também um valor lido da memória e escrever num registo (na instrução MOV registo, [endereço] ou, o seu equivalente em notação RTL, registo  $\leftarrow M[\text{endereço}]$ );

Ao mesmo tempo que o resultado de uma operação na ALU é escrito no registo indicado por REG\_1, o RE (registo de estado) é actualizado com a informação sobre esse resultado, memorizando os bits Z (1 se o resultado for zero) e N (1 se o resultado for negativo);

Alguns bits do RE têm uma saída directa do banco de registos, independentemente da selecção efectuada por REG\_1 e REG\_2 ( nomeadamente os bits Z e N), para estarem sempre disponíveis para a unidade de controlo (saltos condicionais);

O banco de registos obriga a que cada instrução indique quais os registos que quer ler, o que implica que cada célula da memória de instruções tenha mais dois campos, REG\_1 e REG\_2.

Um processador deve ter tantos registos quanto possível. Quantos mais registos houverem, menor é a probabilidade de ser necessário estar a guardar valores na memória de dados. No entanto, é preciso não esquecer os índices REG\_1 e REG2. Quanto maior o número de registos, mais bits estes índices precisam de ter, para poder seleccionar um dos registos. Em cada instrução é preciso ainda especificar dois índices, para além da constante e do opcode. Por isso, um dos primeiros problemas a resolver no desenho de um processador é definir a largura em bits das instruções, o que será feito na secção 4.3.

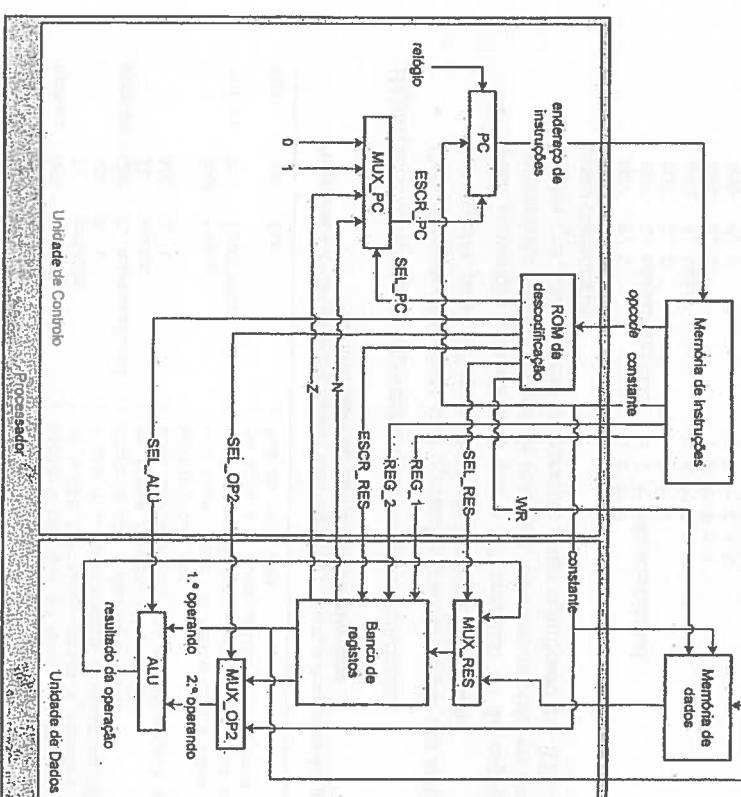


Fig. 4.2 - Processador com banco de registos

- Os operandos da ALU passam a estar exclusivamente dentro do processador (registos ou constantes), o que melhora o desempenho e reflecte a filosofia de usar os registos tanto quanto possível;

- O primeiro operando é sempre um registo. O segundo operando pode eventualmente ser uma constante, dependendo da selecção do multiplexer MUX\_OP2 (caso em que REG\_2 é irrelevante);

- Isto quer dizer que o acesso à memória passa a ser feito exclusivamente por instruções MOV (quer leitura, quer escrita da memória), deixando de ser possível os operandos de instruções como ADD, AND, etc., estarem em memória. Se for o caso, tem de se fazer primeiro o MOV do operando para um registo e só depois se pode fazer a operação. Isto simplifica a implementação das instruções;

#### ESSENCIAL

- Os dois operandos em memória apenas nas instruções de transferência de dados (MOV) e não nas restantes (aritméticas, lógicas, etc.) torna a arquitetura mais simples.

- A maioria das instruções tem dois operandos.

- O primeiro operando desempenha, também geralmente, o papel de resultado. O que significa que se pode ler dois registos e escrever um deles numa mesma operação da ALU.

- O registo de estado é um registo especial na medida em que alguns dos seus bits Z e N, nomeadamente, são actualizados automaticamente após cada operação como ALU e são usados para tomar decisões nos saltos condicionais.

- Cada instrução tem de especificar onde encontram os seus operandos, incluindo registos e constantes. Isto limita o número de registos do banco que podem aparecer na ordem de algumas unidades a poucas dezenas.

## 4.2 ENDEREÇOS DE DADOS E DE INSTRUÇÕES

### 4.2.1 MEMÓRIAS DE DADOS E DE INSTRUÇÕES SEPARADAS: CACHES

Uma das características fundamentais para o funcionamento do PEPE-8 é a existência de memórias separadas, de dados e de instruções, de forma a permitir o acesso simultâneo a uma instrução e aos seus operandos para uma execução num único ciclo de relógio. Tem no entanto a desvantagem óbvia que o uso da memória não pode ser adaptado a programas que usem poucas instruções e muitos dados ou vice-versa. Sendo separadas, a capacidade de cada uma delas constitui uma limitação também separada, podendo chegar-se a uma situação em que a memória de instruções se esgotar, estando a memória de dados quase vazia (ou vice-versa).

Claro que este não é um problema no caso do PEPE-8, que não passa de um processador académico de introdução à área de arquitetura de computadores. Mas o mesmo já não se pode dizer de processadores mais elaborados e que podem correr vários programas complexos simultaneamente, como é o caso do processador de um PC, por exemplo. Podem assim ser identificados dois objectivos conflituosos:

- A existência de uma única memória permite optimizar o uso da sua capacidade, mais para instruções ou mais para dados, de acordo com as necessidades dos programas;

- A existência de memórias separadas permite ler as instruções ao mesmo tempo que os dados, reduzindo o tempo de execução dos programas face à existência de uma única memória, pela qual a unidade de controlo (com as instruções) e a unidade de dados (com os dados) têm de competir nos acessos à memória.

Os processadores actuais permitem conciliar as duas vantagens e ligam apenas a uma memória, que serve tanto para dados como para instruções. Aliás, este já era o caso do modelo original de von Neumann, mas há uma diferença fundamental:

- No modelo original, o acesso a uma instrução tinha de ser feito primeiro e só depois se accedia aos dados necessários para essa instrução;
- Os processadores actuais incluem no seu interior pequenas memórias de instruções e de dados separadas, chamadas *caches* (de instruções e de dados), tal como indicado na Fig. 4.3, em que o núcleo do processador é essencialmente o módulo identificado como Processador na Fig. 4.2.

Esta separação é fundamental para o desempenho da execução, mas as limitações do espaço disponível limitam o tamanho destas memórias, que têm acesso rápido mas capacidade reduzida, face à memória externa, mais lenta mas de muito maior capacidade, designada memória principal. Esta é única para instruções e dados e permite assim um melhor aproveitamento por parte dos programas, usem mais instruções ou mais dados. As *caches* não são visíveis ao programador porque têm um funcionamento especial e automático, contendo apenas uma cópia das células da memória principal mais usadas.

(quer como instruções, quer como dados, na *cache* respectiva). Se o processador tentar ler uma instrução ou aceder a um dado que não tem uma cópia na *cache* relevante, esta lê automaticamente da memória a célula em falta. Aliás, há esta e mais algumas em endereços adjacentes, jogando na antecipação de acessos futuros.

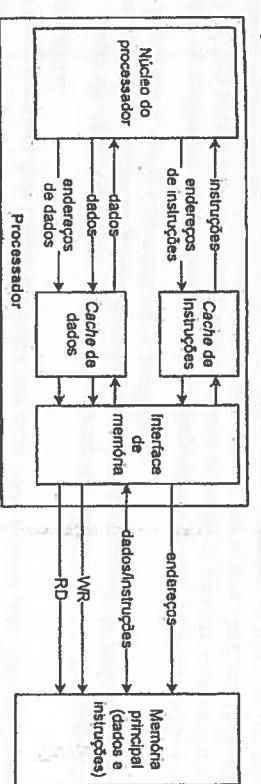


Fig. 4.3 - As *caches* permitem uma aparente separação das memórias de instruções e de dados. A interface de memória coordena as escritas e leituras à memória principal

Os programas não accedem à memória aleatoriamente e vão usando endereços perto uns dos outros (quer em instruções, quer em dados), pelo que esta antecipação estatística até funciona bastante bem. Em média, a probabilidade de o processador encontrar a célula a que pretende acceder na *cache* é tipicamente superior a 0,95 (ou seja, tem sucesso em mais de 95% dos acessos).

Os endereços de instruções e de dados que o núcleo do processador produz não são endereços das *caches*, mas sim endereços na memória principal. A *cache* usa-as apenas para saber se a célula pretendida tem ou não uma cópia na *cache* (dado que esta contém apenas as células mais usadas).

Desta forma, os efeitos práticos deste esquema são os seguintes:

- Tanto a unidade de controlo como a unidade de dados podem acceder ao mesmo tempo à respectiva *cache* (como se estivessem ligadas a memórias separadas), desde que a célula pretendida tenha uma cópia na *cache*;
- Se a *cache* não tiver uma cópia da célula pretendida, vai buscá-la automaticamente e sem que o programa se aperceba (pode demorar mais tempo no primeiro acesso a essa célula, mas nos acessos seguintes a célula já lá está);
- Ambas as *caches* ligam à interface de memória externa, que faz a coordenação e arbitrio dos acessos das *caches* se estas decidirem ao mesmo tempo que precisarem de ir buscar uma célula à memória principal.

Existem técnicas que permitem gerir as *caches* e a memória principal automaticamente, para que na prática o núcleo do processador possa "pensar" que tem disponíveis duas grandes memórias separadas. A grande consequência é que existe apenas um espaço de endereçamento de memória, e não dois separados. Por outras palavras, apesar da aparente separação das memórias de instruções e de dados, uma célula (e respetivo endereço) pertence na realidade a uma única memória, a memória principal. Uma dada célula tanto

pode ser usada como instrução ou como dado, mas não pode haver duas células diferentes no mesmo endereço.

A interface de memória da Fig. 4.3 tem ainda a missão de coordenar os acessos em escrita e em leitura, de modo a ser necessária apenas uma ligação de dados a cada dispositivo exterior ao processador. As memórias comerciais suportam normalmente apenas um tipo de operação de cada vez (ou leitura, ou de escrita) e têm apenas uma ligação de dados bidireccional, para poupar pinos (o custo é bastante dependente do número de pinos e cada ligação de dados corresponde a muitos bits).

As *cache*s e a interface de memória continuam a suportar a ilusão estatística do núcleo do processador, que "pensa" que tem quatro ligações separadas às memórias (de dados e de instruções, cada uma com entrada e saída), tal como indicado na Fig. 4.2.

O facto de a ligação de dados ser bidireccional implica que a mudança de direcção numa sequência de operações leitura/escreta ou vice-versa demore algum tempo (os circuitos electrónicos não conseguem mudar instantaneamente), em que nenhum acesso deve estar a ser feito, nem de leitura nem de escrita.

Para que tal seja possível, o PEPE tem não apenas o sinal WR (que inativo no PEPE-8 indicava leitura) mas também o sinal RD, que precisa de estar activo (a 0) explicitamente para indicar uma operação de leitura. O sinal de WR continua a existir. Se ambos estiverem inactivos, o processador não está a aceder à memória. Estes sinais estão representados na Fig. 4.3 e serão descritos em mais detalhe na secção 6.16.1, na página 446.

#### 4.2.2 ESPAÇO DE ENDEREÇAMENTO E MAPA DE ENDEREÇOS

Se um processador tiver  $n$  bits de endereço, consegue endereçar  $2^n$  células de memória diferentes, isto é, desde o endereço zero até ao endereço  $2^n - 1$ . Esta gama define o espaço de endereçamento do processador.

Tem de se decidir em que zonas deste espaço se devem colocar as instruções e os dados. Normalmente, tanto faz. As células de memória são todas iguais. Nem tem que haver só uma zona de instruções, nem só uma zona de dados. Com instruções de salto é possível passar de uma zona de instruções para outra, e em termos de dados basta o programa especificar os endereços que pretende. A restrição básica é que não haja sobreposições de endereços entre uma zona e outra. Pode haver bocados do espaço de endereçamento totalmente livres (não usados), em que não haja nem instruções nem dados.

Se o processador ligar também a periférico, estes também necessitam de endereços, um por periférico. Normalmente, juntam-se todos os periféricos numa dada zona, mas mais uma vez pode haver várias zonas reservadas para periféricos.

A secção 6.1.3, na página 416, apresenta o endereçamento com mais detalhe.

Se um processador tiver 16 bits de endereço, consegue endereçar  $2^{16}$  ( $65,536$ ) células de memória, isto é, desde o endereço zero até ao endereço 65.535 (esta gama define o seu espaço de endereçamento).

**NOTA**

Na realidade, é normal haver alguns endereços com significados especiais.

Um deles é o endereço da instrução em que o processador começa a execução quando arranca (quando se liga a alimentação, ou quando se reinicializa – reset). Este endereço tem de ser conhecido pelo hardware do processador e portanto tem de ser fixo. Isto significa que pelo menos neste endereço (e provavelmente nos seguintes) terá de ser colocada uma instrução, e não um dado qualquer.

Devem existir também em memória algumas células contendo endereços das instruções para que o processador deve saltar quando ocorrem certos acontecimentos, designados exceções (secção 6.2, na página 459). O hardware do processador tem de saber em que endereços essas células estão, pelo que normalmente esses endereços são predefinidos ou existe um registo que os identifica.

Para cada processador tem de se conhecer as restrições deste género para saber onde colocar instruções e dados.

Além disso, para colocar dados ou instruções numa dada gama de endereços é necessário que fisicamente exista memória nesse endereço (nem todos os endereços do espaço de endereçamento de um processador têm de estar usados).

Se a título de exemplo assumirmos que:

- O endereço da primeira instrução executada pelo processador quando este arranca é zero;
  - A memória tem uma capacidade de  $2^{12}$  (4096) células e está localizada a partir do endereço zero;
  - Há 100 periféricos diferentes e estão em endereços contíguos a partir do endereço 32.768 ( $2^{15}$ ), ou a primeira célula da 2.ª metade do espaço de endereçamento), inclusive;
  - O programa tem dois blocos de instruções, um com 800 e outro com 500 instruções. A primeira instrução do bloco de 800 instruções é aquela em que o programa deve começar a ser executado. Do primeiro bloco passa-se para o segundo por um simples salto (instrução JNE);
  - O programa usa dois blocos de dados, um com 1000 e outro com 700 células de memória;
  - Então o mapa de endereços deste sistema é o indicado na Tabela 4.1. Por mapa de endereços entende-se a lista de gamas de endereços em que no espaço de endereçamento existem dispositivos, como a memória ou os periféricos.
- Nem todos os endereços têm de corresponder a dispositivos (como aliás se verifica na Tabela 4.1). Um acesso em escrita num endereço sem dispositivo não tem consequências (mas o valor escrito não será memorizado) e um acesso em leitura dará um resultado imprevisível, pois não haverá nenhum dispositivo para fornecer o valor a ler pelo processador. Por outro lado, nem todos os endereços da memória têm de estar a ser usados num dado momento (como também sucede neste exemplo), podendo definir-se um mapa de utilização da memória, tal como indicado na Tabela 4.2.

| GAMA DE ENDEREÇOS | DISPOSITIVO FÍSICO            |
|-------------------|-------------------------------|
| 0 a 4095          | Memória (4096 de capacidade)  |
| 4096 a 32.767     | Livre (sem dispositivo)       |
| 32.768 a 32.887   | Periféricos (100 periféricos) |
| 32.888 a 65.535   | Livre (sem dispositivo)       |

Tabela 4.1 - Mapa de endereços

| GAMA DE ENDEREÇOS DA MEMÓRIA | UTILIZAÇÃO              |
|------------------------------|-------------------------|
| 0 a 799                      | 1.º bloco de instruções |
| 800 a 1000                   | Livre (não usado)       |
| 1000 a 1999                  | 1.º bloco de dados      |
| 2000 a 2499                  | 2.º bloco de instruções |
| 2500 a 2999                  | Livre (não usado)       |
| 3000 a 3699                  | 2.º bloco de dados      |
| 3700 a 4095                  | Livre (não usado)       |

Tabela 4.2 - Mapa de utilização da memória

Note-se que esta tabela apresenta apenas um dos cenários possíveis, pois as especificações deste exemplo não indicam por que ordem os diversos blocos de dados e instruções devem aparecer, nem os endereços em que devem estar localizados.

Há apenas duas restrições:

- O bloco de 800 instruções tem de estar localizado a partir do endereço zero, para que o endereço da instrução em que o processador arranca seja o da primeira instrução do programa;
- Não pode haver sobreposição, nem mesmo parcial, entre dois blocos quaisquer.

### 4.3 IMPACTE DA LARGURA DAS INSTRUÇÕES

Na secção 3.3.3 argumentou-se que a largura das instruções não tinha necessariamente que ser igual à largura dos dados, uma vez que o tipo de informação é diferente e vai para sitios diferentes do processador.

Embora isto esteja correcto, há outras considerações a ter em conta. A mais importante prende-se com o facto de tanto as instruções como os dados usarem a mesma memória principal, tal como discutido na secção 4.2.

Esta restrição impõe na prática que as instruções tenham a mesma largura (ou um múltiplo) que a largura dos dados. Pretende-se que as instruções sejam tão compactas

quanto possível e que sejam lidas num só acesso à memória, para além de se tentar minimizar o número de pinos do processador, para o exterior por motivos de custo. Por outro lado, é frequente ter de armazenar (em registos e na memória) não apenas dados mas também os próprios endereços das células de memória em que esses dados estão armazenados.

Estas e outras considerações (entre as quais a simplicidade) levam a que na prática se use apenas uma largura para tudo, incluindo instruções, endereços e dados.

Em termos de dados, o PEPE a utilizar neste livro está, à partida, limitado a 16 bits. A razão é simples e prende-se com o facto de 16 bits (4 dígitos hexadecimais) constituirem o limite de razoabilidade para a programação manual em linguagem assembly.

4 dígitos hexadecimais ainda se escrevem sem problemas, mas ao passar para 32 bits (8 dígitos hexadecimais) já não se consegue olhar para um número e ver instantaneamente quantos dígitos hexadecimais tem, pelo que se começa a ter de contar os dígitos e fica muito mais confuso.

A partir dos 32 bits já é o reino da programação automática, com compiladores que geram directamente o código-máquina a partir das linguagens de programação de alto nível. Para comprovar isto, basta comparar o aspecto visual de dois números, um com 16 bits e outro com 32:

7F43H      3AF20A4CH

Olhando para o número da esquerda, consegue-se ver instantaneamente quantos dígitos hexadecimais tem. No número da direita, tem de se contar e até é difícil conseguí-lo logo à primeira vez. Assim, em termos didáticos, mais vale suportar a limitação dos 16 bits:

- Em termos de dados, o PEPE consegue apenas lidar directamente com inteiros em complemento para 2 na gama -32.768 até +32.767 e números binários sem sinal (usados nas instruções lógicas, por exemplo) entre 0 e 65.535;
- Em termos de endereços, o PEPE consegue gerar 65.536 endereços diferentes, desde o endereço 0000H até FFFFH (em hexadecimal).

Tomada a decisão de usar 16 bits, fica o desafio de conseguir compactar em 16 bits todas as instruções que um processador deve suportar, juntamente com as constantes e os índices dos registos envolvidos nessas instruções, o que será feito na secção 4.5. Mas antes ainda é preciso analisar os endereços em maior detalhe.

**NOTA** Uma das vantagens de um processador ter 32 bits é haver mais bits para codificar os números dos registos envolvidos em cada instrução ou até mesmo poder especificar 3 operandos por instrução, usando um terceiro registo para o resultado.

No entanto, a principal razão que motivou a evolução para os 32 bits (com exceção dos sistemas mais simples – muito mais que um PC) é a limitação da gama de representação de números inteiros com 16 bits, demasiado pequena para os casos práticos em que os computadores se usam.

## ESSENCIAL

- Os computadores têm uma só memória que serve tanto para instruções como para dados;
- Mas intencionalmente o processamento é mais rápido se as memórias forem separadas e puderem ser acedidas ao mesmo tempo em vez de uma de cada vez, em sequência;
- Este problema resolve-se com duas *caches* (para dados e para instruções), que possuem uma cópia das células da memória principal mais usadas, conseguindo assim suportar os acessos simultâneos na maior parte dos casos.
- O espaço de endereçamento é único, tal como a memória principal. O que significa que se tem de colocar as instruções e os dados em endereços diferentes.
- Os dados e instruções têm de ser armazenados na mesma memória. Também é frequente ter de armazenar os próprios endereços em registos e na memória. Isto leva na prática a que dados, instruções e endereços tenham a mesma largura.

(a) A B C D E F

(b)

| LARGURA DO PROCESSADOR | ACESSOS POSSÍVEIS À MEMÓRIA, EM BYTES |   |    |     |         |   |    |      |
|------------------------|---------------------------------------|---|----|-----|---------|---|----|------|
|                        | 16 bits                               |   |    |     | 32 bits |   |    |      |
|                        | A                                     | B | AB | --- | C       | D | CD | CDEF |
| 16 bits                | A                                     | B | AB | --- | C       | D | CD | CDEF |
| 32 bits                | E                                     | F | EF |     |         |   |    |      |

Fig. 4.4 - Palavras com os bytes identificados por letras. (a) – De 16 bits; (b) – De 32 bits

Tabela 4.3 - Acessos possíveis à memória no caso de processadores de 16 bits (Fig. 4.4a) e 32 bits (Fig. 4.4b) com capacidade de endereçamento de byte

Isto tem uma consequência imediata e de grande impacte: cada byte tem memória item de ter o seu endereço individual. Se um processador suportar isto, diz-se que tem endereçamento de byte (por oposição ao endereçamento por palavra).

Porque é que isto é tão importante?<sup>23</sup>

- Suporrtar endereçamento de byte reduz a capacidade de memória que o processador é capaz de endereçar. Com endereçamento de palavra, há um endereço diferente para cada célula de memória de 16 bits. Ou seja, com 16 bits de endereço o processador consegue endereçar  $2^{16}$  (64 K, ou 65.536) células de 16 bits cada, ou 128 KBytes ( $2 \times 64$  K). Com endereçamento de byte, apenas 64 KBytes (metade) podem ser endereçados, pois os 16 bits de endereço mantêm-se e agora tem de ser feito um endereço diferente para cada byte individual;
- O endereçamento de byte interessa apenas para os dados, mas os endereços das instruções também são afectados, pois a memória principal é a mesma. Como as instruções são indivisíveis, os endereços das instruções (o valor do PC) têm de avançar de 2 em 2. Ao incrementar o PC para passar a endereçar a instrução seguinte tem de se somar 2 ao PC e não apenas 1;
- Nas instruções de acesso à memória (tem dados) tem de se indicar que quantidade de informação se pretende usar, 8 ou 16 bits, o que implica mais opcodes (na prática, equivale a duplicar o número de instruções dedicadas à transferência de dados de e para a memória);
- Cada célula de memória de 16 bits contém 2 bytes, com endereços par (x) e ímpar (x+1). Cada célula começa sempre num byte de endereço par;

<sup>23</sup> Aqui refere-se apenas o caso dos processadores de 16 bits, mas as conclusões são facilmente extrapoláveis para 32 bits, 64 bits, etc., com as necessárias adaptações.

A Fig. 4.4 representa duas palavras, uma de um processador de 16 bits e outra de um processador de 32 bits, com os bytes identificados por letras. A Tabela 4.3 indica os acessos possíveis para cada processador.

Fazer um acesso à memória (quer em dados, quer em instruções) em 16 bits especificando o endereço  $y$  implica aceder ao byte com o endereço  $y$  e ao byte com o endereço  $y+1$ . Se  $y$  for par (acesso alinhado), está-se a aceder apenas a uma célula. Se  $y$  for ímpar, está-se a aceder ao segundo byte de uma célula e o primeiro byte da célula seguinte (acesso desalinhado). Para suportar isto, o processador tem de fazer dois acessos diferentes à memória e depois combinar os bytes das duas células lidas, o que complica o hardware, razão pela qual muitos processadores, incluindo o PEPE, profíbrem isto (geram um erro se tal acontecer);

O endereçamento de byte não é transparente para o programador de linguagem assembly. Ao usar os endereços, é preciso ter em atenção que os endereços das palavras (células de 16 bits) evoluem de 2 em 2 e são pares, enquanto os endereços dos bytes evoluem de 1 em 1 e podem ser pares ou ímpares. A Tabela 4.14, na página 219, apresenta um exemplo com algumas instruções (de 16 bits) e os respectivos endereços na memória, pares e a evoluir de 2 em 2.

O PEPE seria mais simples se não suportasse endereçamento de byte, mas em termos didácticos tal seria uma lacuna uma vez que os processadores comerciais têm esta capacidade, com grande impacte ao nível da programação em linguagem assembly. A importância deste tópico justifica refazer o exemplo da secção 4.2.2 com endereçamento de byte. Os dois exemplos devem ser comparados, tendo em atenção que:

- O processador continua a ter os mesmos 16 bits de endereço. O espaço de endereçamento continua a ser desde 0 até 65.535, mas agora referindo-se a bytes. Isto significa que, em células (palavras), o espaço foi reduzido para metade, pois agora só suporta metade (32.768 palavras em vez de 65.360);
- O endereço da primeira instrução executada pelo processador quando este arranca continua a ser 0000H (entenda-se instrução contendo os bytes com endereços 0000H e 0001H);
- Mantemos em  $2^{12}$  (4096) a capacidade em células da memória do computador, uma vez que só estamos a mudar o endereçamento, não o programa. Mas agora precisamos do dobro dos endereços ( $2^{13}$ , ou 8192), para poder endereçar toda a memória (que continua a estar localizada a partir do endereço 0000H), uma vez que todos os bytes têm de ser endereçados individualmente;
- Continua a haver 100 periféricos diferentes, cada um com 16 bits, e continuam a estar localizados a partir do primeiro endereço da segunda metade do espaço de endereçamento, ou seja, 32.768, tal como anteriormente. Mas agora gastam 200 endereços e o segundo periférico está no endereço 32.770 e não no 32.769 (porque os periféricos têm 16 bits e gastam dois endereços);
- O programa continua a ter dois blocos de instruções, um com 800 e outro com 500 instruções. Mas cada instrução tem 2 bytes, o que quer dizer que agora gastam 1600 e 1000 endereços, respectivamente;
- O programa continua a usar dois blocos de dados, um com 1000 e outro com 700 células de memória. Mas cada célula (palavra) tem 2 bytes, o que quer dizer que estes dois blocos de dados gastam 2000 e 1400 endereços, respectivamente.

Nestas condições, o mapa de endereços com endereçamento de byte é o indicado na Tabela 4.4. Comparando com a Tabela 4.1, nota-se que o espaço (número de endereços) gasto pelos dispositivos (memória e periféricos) duplicou, à custa da redução do espaço livre. Há efectivamente uma redução no número total de palavras que o processador consegue endereçar, pois o número de bytes dos endereços manteve-se e a entidade mínima endereçável é agora mais pequena (byte).

| GAMA DE ENDEREÇOS | DISPOSITIVO FÍSICO                                   |
|-------------------|------------------------------------------------------|
| 0 a 8191          | Memória (4096 palavras, ou 8192 bytes de capacidade) |
| 8191 a 32.767     | Livre (sem dispositivo)                              |
| 32.768 a 32.967   | Periféricos (100 periféricos, 200 endereços)         |
| 32.968 a 65.535   | Livre (sem dispositivo)                              |

Tabela 4.4 - Mapa de endereços com endereçamento por byte

O novo mapa de utilização da memória consta da Tabela 4.5. Cada bloco (seja de dados, seja de instruções) ocupa agora o dobro do número de endereços, pois tem de endereçar cada um dos bytes de cada palavra de 16 bits. Note-se que nem o programa nem os dados mudaram de tamanho, nem a memória alterou a sua capacidade. Apenas a forma de contabilizar os endereços foi alterada.

| GAMA DE ENDEREÇOS DA MEMÓRIA | UTILIZAÇÃO              |
|------------------------------|-------------------------|
| 0 a 1599                     | 1.º bloco de instruções |
| 1600 a 1999                  | Livre (não usado)       |
| 2000 a 3999                  | 1.º bloco de dados      |
| 4000 a 4999                  | 2.º bloco de instruções |
| 5000 a 5999                  | Livre (não usado)       |
| 6000 a 7399                  | 2.º bloco de dados      |
| 7400 a 8191                  | Livre (não usado)       |

Tabela 4.5 - Mapa de utilização da memória com endereçamento de byte

A utilização do byte como unidade mínima de endereço proporciona uma forma de endereçamento que depende apenas da dimensão dos dados e é independente da largura em bits da palavra do processador. O efeito do endereçamento de byte é particularmente notório na escrita da memória. Se se pretender alterar apenas um byte dispondo apenas de endereçamento de palavra, tem de se ler a palavra inteira para um registo (que terá de estar disponível), isto é, não ter um valor que não se possa destruir), alterar o byte pretendido e escrever a palavra na memória. Com endereçamento de byte, basta apenas escrever o byte pretendido na memória. Normalmente, este conceito já não é estendido ao bit. Existem instruções capazes de alterar o valor de um bit individualmente, mas apenas num registo. Para acesso à memória, o byte é a unidade mínima de acesso.

**ESSENCIAL**

O acesso à memória é feito em palavras de tamanho fixo e não permite optimizar o acesso a quantidades de informação mais pequenas do que uma palavra, como bytes ou characters.

Solução: os endereços passam a ser de byte em byte e a interface de memória do processador passa a suportar acessos apenas a um byte ou palavra inteira.

- O mapa de endereços e a capacidade das memórias deve ser expresso em bytes (a unidade básica de endereçamento).
- Os endereços das células da memória (com 16 bits) avançam de 2 em 2 e devem ser par (acessos alinhados).
- O PC (Program Counter) avança também de 2 em 2.
- Há instruções específicas para aceder à memória em byte e em palavra. O endereçamento de byte não é transparente para a programação.
- O byte é a unidade mínima de acesso à memória.

## 4.5 CODIFICAÇÃO DAS INSTRUÇÕES

Já na secção 3.4.1, na página 149, se introduziu a necessidade de codificar em alguns valores de *opcode* (código de operação) as combinações de sinais de controlo internos do processador correspondentes às várias instruções. Este esquema é conhecido por codificação das instruções.

Dito de outra forma, deve atribuir-se um código a cada instrução, que depois o processador se encarrega de decodificar ao executar essa instrução e produzir os vários sinais de controlo necessários, tal como indicado pela ROM de descodificação na Fig. 4.2, por exemplo.

No PEPE-8 há menos de 16 instruções, sendo suficientes 4 bits para a codificação das suas instruções em *opcodes*. Mas o PEPE é mais poderoso e com mais instruções, pelo que 4 bits não chegam e o melhor é reservar 8 bits.

Como cada instrução tem tipicamente dois registos como operandos, os 8 bits que sobram dos 16 bits de largura das instruções dizem-nos rapidamente que só temos 4 bits para o índice dos registos, ou, por outras palavras, que o PEPE não poderá ter mais do que 16 registos.

E a constante que as instruções também especificam? Já não há espaço!

Felizmente, é possível organizar as instruções para que não especifiquem ao mesmo tempo dois registos e uma constante. Podemos ainda usar outro truque: usar menos bits de *opcode* para as instruções que precisarem de mais bits para os operandos.

A solução é codificar os 16 bits das instruções em 4 campos de 4 bits, de acordo com quatro tipos de instruções, tal como indicado na Tabela 4.6.

Sobre esta tabela é possível fazer os seguintes comentários:

- Sendo o número de registos 16, já não é práctico referenciá-los por letras. Por isso, adopta-se a nomenclatura típica R0, R1, R2, ..., R15;

| Tipo      | Exemplo     | 1.º Campo (4 bits) | 2.º Campo (4 bits) | 3.º Campo (4 bits)                  | 4.º Campo (4 bits)                    |
|-----------|-------------|--------------------|--------------------|-------------------------------------|---------------------------------------|
| INSTRUÇÃO |             |                    |                    |                                     |                                       |
| 1         | ADD R0, R1  |                    |                    | 1.º operando / resultado (registro) | 2.º operando (constante de 4 bits)    |
| 2         | ADD R0, 2   |                    |                    | Opcode secundário                   | Operando único (constante de 8 bits)  |
| 3         | IZ ... 100  |                    |                    | Opcode principal                    | 1.º operando / resultado              |
| 4         | MOV R0, 100 |                    |                    |                                     | 2.º operando (constante de 8 bits)    |
| 5         | JMP 2000    |                    |                    |                                     | Operando único (constante de 12 bits) |

Tabela 4.6 - Codificação das instruções mais comuns por tipos de operandos

O código de operação (*opcode*) é dividido em dois, principal e secundário, de 4 bits cada um. Todas as instruções têm o *opcode* principal, mas só as que gastam menos bits nos operandos (tipos 1, 2 e 3) têm *opcode* secundário. Outra forma de ver este esquema é considerar que cada instrução do tipo 4 ou 5 gasta todas as 16 codificações decorrentes do *opcode* secundário, que assim é omitido, libertando 4 bits para especificar os operandos (que nestes casos podem ocupar 12 bits);

As instruções do tipo 1 são as típicas de dois registos, em que o primeiro operando serve também para armazenar o resultado (exemplo: ADD R0, R1);

As instruções do tipo 2 são tipicamente as que usam uma constante para operar sobre um registo, que fica com o seu valor alterado. Um exemplo é a soma de um registo com uma constante imediata, especificada na própria instrução (exemplo: ADD R0, 4). Uma limitação óbvia é que, só havendo 4 bits para a constante, apenas é possível especificar valores entre -8 e +7 ou entre 0 e 15, consonante a instrução interpreta a constante em complemento para 2 ou como valor sem sinal.

Este tipo de codificação permite optimizar os casos (frequentes, aliás) de operações aritméticas com constantes de valor baixo (iniciamente um registo de uma unidade, por exemplo). Se se pretender uma operação aritmética com uma constante maior, que não caiba em 4 bits, tem de se (i) carregar primeiro outro registo com o valor pretendido e (ii) depois somar os dois registos (ver secção 4.10.3 para mais detalhes);

As instruções do tipo 3 são tipicamente as instruções que efectuam uma operação entre um registo fixo, predeterminado (e portanto pode inferir-se qual é o registo a partir do *opcode*, não sendo necessário especificá-lo como operando), e a constante. No caso do PEPE, as de uso mais frequente nesta categoria são as de salto

condicional, cujo salto é relativo, isto é, que somam a constante ao valor do PC (em vez de saltar para um endereço que, a ser especificado completamente, exigiria 16 bits e não apenas 8 bits). Isto permite apenas saltos de até 128 instruções para trás e 127 para a frente em relação ao valor actual do PC, o que permite contemplar muitos dos casos mais frequentes. Para saber como efectuar um salto condicional para qualquer endereço ver a secção 4.9;

- As instruções do tipo 4 são as que efectuam uma transferência de uma constante para um registo (instruções MOV registo, constante) e permitem inicializar qualquer dos 16 registos com qualquer valor de 16 bits. Uma questão interessante é saber como é que tal é possível se a constante é de apenas 8 bits (seria impossível especificar uma constante de 16 bits numa instrução também de 16 bits, pois não sobraria espaço para o opcode). A solução é fazê-lo em duas instruções, com alguns truques pelo meio (secção 4.10.3);

As instruções do tipo 5 são semelhantes às do tipo 3, mas agora o operando é de 12 bits e permite um maior leque de instruções para saltar em torno do PC. Este formato é usado pelas instruções de salto incondicional. Mesmo assim, o salto é relativo (sóman a constante ao valor do PC) e não permitem um salto absoluto, isto é, para qualquer endereço do espaço de endereçamento. A secção 4.9 ensina como vencer esta limitação.

**NOTA** Os processadores de 32 bits têm 32 bits para codificar uma instrução e portanto algumas das restrições aqui referidas desaparecem ou ficam aliviadas. É possível, por exemplo:

- Ter mais registos (mais bits para codificar os seus índices);
- Especificando três registos por instrução, dois operandos e um resultado (em vez de o resultado ter de ser armazenado no registo com o 1.º operando). Este é talvez o maior benefício, pois é possível reduzir o número de instruções necessárias (uma vez que o 1.º operando não precisa de ser destruído) e até mesmo reduzir o número de instruções que o processador sabe executar (por exemplo, a instrução MOV R1, R2 pode ser na realidade ADD R1, R2, 0, ou seja, R1  $\leftarrow$  R2 + 0, aproveitando-se assim a operação de soma em vez de se ter de definir também uma instrução de MOV);
- Usar mais bits para especificar as constantes (esta não é propriamente uma vantagem, pois as constantes agora também são maiores, de 32 bits, mas é possível por exemplo representar constantes de até 16 bits numa única instrução);
- Ter outras combinações de registos e constantes (por exemplo, poder especificar dois registos e uma constante de 16 bits);
- Ter bits de sobra para especificar outra informação (que depende da arquitectura e portanto não será referida aqui).

A desvantagem óbvia é que cada instrução gasta o dobro da informação (32 bits em vez de 16 bits). No entanto, os circuitos integrados de memória cada vez têm mais capacidade e o custo por byte de memória é cada vez mais baixo. Por isso, hoje em dia privilegia-se mais o desempenho (rapidez de processamento) do que o gasto de memória. Quantos mais bits um processador tiver de largura, mais informação consegue processar por unidade de tempo. Esta tem sido uma das razões fundamentais para o progressivo aumento da largura dos processadores.

#### ESSENCIAL

A codificação das instruções é um dos aspectos mais importantes no projecto de um processador;

Ter formato variável permite acomodar toda a informação necessária (registos, constantes, etc.) mas complica o circuito de descodificação de instrução, que acaba por ficar mais lento;

Um formato fixo implica que todas as instruções tenham o mesmo tamanho, o que simplifica muita coisa, mas reduz o número de bits disponível para o opcode e operandos, limitando entre outras coisas o número de registos que a arquitectura pode ter;

Com alguns pequenos truques, consegue-se optimizar a codificação para os casos mais frequentes, mantendo a simplicidade global.

## 4.6 REGISTOS

Devido à codificação das instruções, o PEPE está limitado a 16 registos. Estes são os registos especificáveis nas várias instruções que admitem registos como operando explícito, como por exemplo ADD R0, R1.

Outras instruções usam registos de forma implícita, como por exemplo as instruções de salto, que alteram o PC sem o referirem explicitamente. De facto, o PC nunca precisa de ser manipulado explicitamente, pois há instruções específicas que o fazem automaticamente. Isto reflecte o facto de o PC ser um registo específico (serve apenas para endereçar as instruções na memória), enquanto os restantes registos são de uso geral.

Por este motivo, o PC não está incluído no banco de registos e não conta para o limite dos 16 registos do banco de registos, que podem ser especificados nos campos das instruções em que devia ser especificado um registo.

Mesmo no banco de registos há alguns registos que, embora possam ser usados como registos de uso geral, têm algumas atribuições específicas face a certas instruções. Esses, para além da designação geral R0 a R15, têm também um nome para facilitade de referência, tal como indicado na Fig. 4.5.

Todos os registos têm 16 bits. Algumas instruções actuam ao nível do byte e tratam os dois bytes de cada registo de forma diferente. Esta é a única razão pela qual aparece uma divisão vertical na Fig. 4.5, pois cada registo é normalmente manipulável como um todo.

Os registos R0 a R10 são totalmente genéricos e podem ser livremente usados pelo programador, sem qualquer restrição. Os registos R11 a R15 têm algumas funcionalidades associadas. O seu uso pelo programador deve ser cuidado e com algumas restrições, sob pena de o funcionamento do programa não ser correcto.

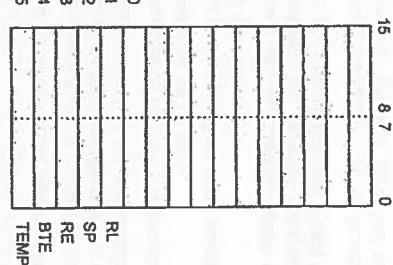


Fig. 4.5 - Registos do banco de registos do PEPE. Cada registo tem 16 bits, ou 2 bytes. A divisão vertical é meramente conceptual. Os cinco últimos registos têm algumas funcionalidades específicas, pelo que o seu uso deve ser controlado.

As especificidades associadas a estes cinco registos são fundamentalmente as seguintes (a introduzir nas secções indicadas):

- R11, ou RL (Registo de Ligação) – É usado para optimizar a chamada de rotinas (secção 5.7.2.2, na página 317). Pode ser livremente usado como registo geral desde que não se esteja a usar no âmbito desta optimização;
- R12, ou SP (Stack Pointer, ou Apontador de Pilha) – É usado na implementação de instruções que manipulam a pilha, uma estrutura de dados muito importante em qualquer processador e que será descrita apenas na secção 5.7.2.3, na página 321. Este registo pode ser usado livremente desde que o programa não use a pilha;
- R13, ou RE (Registo de Estado) – Contém alguns bits importantes para o funcionamento do processador. A Fig. 4.2 já especifica dois bits destes, o Z e o N usados nas instruções de salto condicional, mas há outros que serão introduzidos mais à frente, em particular nas secções 4.7 e 6.2.2.2 (na página 464). Este registo deve ser alterado pelo programador apenas em casos bem determinados (ainda a especificar, nomeadamente na secção 6.2.2.2). A Tabela A.2, na página 697, apresenta a funcionalidade completa dos bits deste registo;
- R14, ou BTE (Base da Tabela de Excepções) – É usado para conter o endereço de base de uma tabela, destinada a suportar eventos excepcionais (como erros de execução, por exemplo), ou exceções, cuja utilização é descrita na secção 6.2.1, na página 459. Não se deve usar como registo geral porque se ocorrer uma exceção e este registo não tiver um valor adequado o processador poderá ficar descontrolado (tentando executar instruções em endereços que não têm instruções válidas);

R15, ou TEMP (de temporário) – Designa um registo usado temporariamente na implementação de instruções mais complexas (como por exemplo a instrução SWAP, que troca os valores de dois registos, descrita na secção 4.10.2). O grande problema em usar este registo é que ele é alterado por algumas instruções e portanto o utilizador não pode assumir que lá guardou um valor e que esse valor lá estará quando precisar dele. Recomenda-se que não seja usado directamente pelo programador.

**NOTA** Em muitos processadores, registos específicos de implementação como o TEMP estão simplesmente escondidos, sendo usáveis apenas pelo hardware e não estando acessíveis ao utilizador. O PEPE também tem registos desses, tal como descrito no capítulo 7.

- A inclusão do TEMP no banco de registos, e portanto especificável nas instruções, deve-se essencialmente às seguintes razões:
- Facilitar a implementação das instruções, reduzindo o número de sinais de controlo;
  - Introduzir o problema de registos disponíveis ao utilizador mas com fins específicos, o que também acontece em alguns processadores.

## 4.7 BITS DE ESTADO

Um dos aspectos fundamentais de um processador é a capacidade de tomar decisões com base no resultado de uma dada operação. As decisões são implementadas por saltos condicionais (secção 4.9), que saltam para um dado endereço ou seguem para o endereço seguinte, com base numa ou mais condições estabelecidas pelo resultado da instrução anterior.

Estas condições são conhecidas por *bits de estado* (*flags*) e ficam memorizadas no RE (Registo de Estado).

Os bits de estado mais frequentes são os quatro indicados na Tabela 4.7. Os bits de estado são tipicamente referidos por uma só letra, adotando-se aqui as letras usuais em inglês, para ser mais fácil o seu reconhecimento ao ler manuais de processadores comerciais.

**NOTA** Alguns computadores não têm estes bits de estado, tendo apenas instruções que testam uma dada condição (se dois registos têm o mesmo valor, por exemplo) e saltam ou não, tudo na mesma instrução.

Outros, como o PEPE, permitem que um leque mais alargado de instruções produza informação para as instruções de salto condicional. Separam assim as instruções que estabelecem as condições (bits de estado), das que executam o salto ou não, dependendo dessas condições.

A discussão sobre as vantagens e desvantagens de um esquema e de outro é um assunto mais avançado que está fora do âmbito deste livro.

| BIT DE ESTADO | NAME               | VALOR APÓS UMA OPERAÇÃO<br>(1 = ACTIVO, 0 = INACTIVO)                                                 |
|---------------|--------------------|-------------------------------------------------------------------------------------------------------|
| Z             | Zero               | 1 se o resultado for zero                                                                             |
| N             | Negativo           | 0 se o resultado for diferente de zero                                                                |
| C             | Transporte (Carry) | Igual ao bit de transporte do resultado da operação (secção 2.8.1). 1 significa que houve transporte. |
| V             | Excesso (Overflow) | 1 se o resultado da operação produzir uma condição de excesso (secção 2.8.3)                          |

Tabela 4.7 - Bits de estado mais comuns

A Fig. 4.6 indica qual a posição destes bits no registo de estado (RE) do PEPE. Há outros bits com funcionalidade específica no RE, mas que só serão introduzidos mais tarde nomeadamente na secção 6.2.2.2, na página 464.



Fig. 4.6 - Disposição dos bits de estado no RE (Registo de Estado)

Algumas instruções afectam (alteram) apenas alguns destes bits, enquanto outras afectam todos os bits. As instruções de salto condicional podem testar apenas um ou vários bits de estado, dependendo da condição (secção 4.9). São suportadas não apenas as condições derivadas directamente dos bits de estado (zero e negativo) mas também outras como maior, menor, etc. É também possível ler/escrever o RE de uma só vez ou alterar um ou mais dos seus bits directamente, tal como em qualquer outro registo.

Como resultado de uma operação, os bits Z e N não podem ficar activos simultaneamente, mas podem ficar ambos a 0 se o número for estritamente positivo (maior que zero), Z só ficará activo se o resultado de uma operação tiver todos os seus bits a zero. É sempre fonte de alguma confusão o bit Z ficar a 1 quando o resultado é zero (e 0 quando o resultado é diferente de zero), pelo que pode ser útil pensar em Z como activo ou inactivo em vez de 1 e 0, respectivamente. O bit N presta-se menos a confusões porque na prática é igual ao bit de sinal do resultado.

Os bits C (transporte) e V (excesso) ficam activos (a 1) quando de alguma forma o resultado de uma operação é demasiado grande para "caber" (poder ser representado correctamente) em 16 bits, mas segundo duas perspectivas diferentes:

- O bit C encara os operandos e o resultado como números binários de N bits sem sinal, com valor entre 0 e  $2^N - 1$ . O bit C ficará activo quando o resultado de uma operação com N bits for superior a  $2^N - 1$  (ou seja, houver um transporte, "é vira urt", diferente de zero para além do bit de maior peso);
- O bit V encara os operandos e resultado como números binários de N bits com sinal, em notação de complemento para 2. O bit V ficará activo quando o sinal do

resultado for diferente do correcto (por exemplo, se a soma de dois números positivos der um número negativo, o que significa que o resultado era demasiado grande e "deu a volta" para os números negativos).

Dependendo dos valores dos operandos e da operação, é possível ter as quatro combinações dos dois bits C e V. O bit V activo é normalmente um erro, que pode ser ignorado ou não. Se for ignorado, é preciso ter a noção de que o resultado está errado e pode afectar severamente o programa. O bit C activo (com o bit V inactivo) pode ser uma situação normal (basta somar um número com o seu simétrico, o que dá zero) e pode ser usado para implementar operações aritméticas com mais bits do que a largura do processador (uma soma de 32 bits no PEPE, por exemplo – ver secção 4.11.1.2).

Para melhor compreensão, a Tabela 4.8 apresenta o valor em decimal dos números binários de 4 bits, quer sem sinal (só números positivos), quer com sinal (positivos e negativos) em notação de complemento para 2.

| NÚMERO BINÁRIO (4 BITS) | NÚMERO DECIMAL (SEM SINAL) | NÚMERO DECIMAL (COM SINAL) |
|-------------------------|----------------------------|----------------------------|
| 0000                    | 0                          | 0                          |
| 0001                    | 1                          | 1                          |
| 0010                    | 2                          | 2                          |
| 0011                    | 3                          | 3                          |
| 0100                    | 4                          | 4                          |
| 0101                    | 5                          | 5                          |
| 0110                    | 6                          | 6                          |
| 0111                    | 7                          | 7                          |
| 1000                    | 8                          | -8                         |
| 1001                    | 9                          | -7                         |
| 1010                    | 10                         | -6                         |
| 1011                    | 11                         | -5                         |
| 1100                    | 12                         | -4                         |
| 1101                    | 13                         | -3                         |
| 1110                    | 14                         | -2                         |
| 1111                    | 15                         | -1                         |

Tabela 4.8 - Duas formas de encarar um número binário: só valores positivos e com sinal (em complemento para 2)

A Tabela 4.9 permite exemplificar várias situações e os valores dos bits de estado resultantes de uma operação de adição. Usam-se números binários de apenas 4 bits, para ser mais simples, mas as regras são exactamente as mesmas para qualquer número de bits (incluindo os 16 do PEPE).

| CASO | VCNZ | OPERAÇÃO<br>(EM DECIMAL)                                                                                                                                                             | RESULTADO         | OPERAÇÃO<br>(EM BINÁRIO) |
|------|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|--------------------------|
| 0    | 0000 | 1 + 6                                                                                                                                                                                | 7                 | + 0110<br>0 011          |
| 1    | 0001 | 0 + 0                                                                                                                                                                                | 0                 | + 0000<br>0 0000         |
| 2    | 0010 | 5 + (-6)                                                                                                                                                                             | -1                | + 1010<br>0 1111         |
| 3    | 0011 | Impossível. Um resultado não pode ser simultaneamente zero e negativo                                                                                                                |                   |                          |
| 4    | 0100 | 7 + (-5)                                                                                                                                                                             | 2                 | + 1011<br>1 0010         |
| 5    | 0101 | 5 + (-5)                                                                                                                                                                             | 0                 | + 1011<br>1 0000         |
| 6    | 0110 | -3 + (-3)                                                                                                                                                                            | -6                | + 1101<br>1 1010         |
| 7    | 0111 | Impossível. Um resultado não pode ser simultaneamente zero e negativo                                                                                                                |                   |                          |
| 8    | 1000 | Impossível. Para o resultado ser positivo com excesso, é porque houve uma soma de operandos negativos. Mas a soma destes ( $\geq 8$ quando encarados sem sinal) dá sempre transporte |                   |                          |
| 9    | 1001 | Impossível. Se o resultado é zero, então não há excesso                                                                                                                              |                   |                          |
| 10   | 1010 | 7 + 3                                                                                                                                                                                | -6 (10 sem sinal) | + 0011<br>0 1010         |
| 11   | 1011 | Impossível. Um resultado não pode ser simultaneamente zero e negativo                                                                                                                |                   |                          |
| 12   | 1100 | -7 + (-3)                                                                                                                                                                            | +6                | + 1001<br>1 1101         |
| 13   | 1101 | Impossível. Se o resultado é zero, então não há excesso                                                                                                                              |                   |                          |
| 14   | 1110 | porque houve uma soma de operandos positivos. Mas a soma destas ( $\leq 7$ ) nunca pode dar transporte.                                                                              |                   |                          |
| 15   | 1111 | Impossível. Um resultado não pode ser simultaneamente zero e negativo                                                                                                                |                   |                          |

Tabela 4.9 - Exemplos de afectação dos bits de estado pela operação de adição

### SIMULAÇÃO – BITS DE ESTADO

Esta simulação ilustra o funcionamento dos bits de estado com a instrução ADD. Os respectos cobertos incluem os seguintes:

- Carregamento manual dos registos com os operandos, usando a Tabela 4.8 com base, mas agora com operandos de 16 bits;
- Verificação do funcionamento da instrução ADD e do valor dos bits de estado após a instrução;
- Repetição para vários valores de operandos.

#### ESSENCIAL

- O PEPE tem 16 registos no seu banco de registos. O actor limitativo do número de registos é o número de bus para referenciar os registos nas instruções (4);
- A maior parte dos registos é de utilização genérica, mas alguns (SP, BP, EM) têm utilizações específicas que o programador tem de respeitar;
- O PEPE faz parte do banco de registos. Tem uma utilização demasiado específica e instruções específicas para manipular;
- As registos de estado são linhas chamadas bits de estado, que memorizam informação sobre o resultado da última operação. A sua principal utilidade é servir de base às decisões executadas pelas instruções de salto condicional.

A operação usada é sempre a adição, por simplicidade e até porque os processadores não implementam realmente a subtração. Para implementar a instrução SUB R1, R2, o que é feito na realidade é complementar R2 para 2 e depois efectuam a soma de R1 com esse complemento. Note-se que:

- O caso 1 ( $Z=1$  e todos os outros a 0) só pode ser atingido somando zero com zero. Qualquer soma de um número com o seu simétrico (para dar zero como resultado) implica imediatamente transporte, pois por definição a soma de um número de  $N$  bits com o seu simétrico dá  $2^N$  como resultado, o que não é representável com  $N$  bits (apenas com  $N+1$ ). Ou seja, já é o caso 5, exemplificado com  $5 + (-5)$ ; O excesso no caso da soma só pode ser atingido somando dois números com o mesmo sinal (é o sinal do resultado for diferente do sinal dos operandos). Se um for positivo e outro negativo não é possível atingir a situação de excesso; Esta tabela pode também ser feita para as operações de multiplicação e divisão, com algumas adaptações. Por limitações de espaço e por ser um tópico específico, a elaboração destas tabelas é deixada como exercício para o leitor mais interessado.

## 4.8 CONJUNTO DE INSTRUÇÕES

Tal como o nome indica, o conjunto de todas as instruções da linguagem assembly de um processador designa-se **conjunto de instruções** desse processador.

Isto não quer dizer que o *hardware* em si não permita outras operações, quer por combinações diferentes dos sinais de controlo, quer por execução sequencial de séries de operações muito elementares (microprogramas – ver secção 2.6.7.4, na página 83) que no seu conjunto implementam instrução específicas.

No entanto, se compiliarmos muito o *hardware* aumentamos o tempo de atraso dos sinais nos circuitos e a frequência do relógio tem de baixar, além de que a arquitectura fica mais complexa e difícil de programar. Por isso, existe um compromisso entre funcionalidade e complexidade. A melhor solução é combinar simplicidade com uma funcionalidade mínima, numa solução de equilíbrio entre variedade de operações, simplicidade e rapidez de processamento. A secção 6.6.7, na página 552, elabora este aspecto.

O conjunto de instruções do PEPE procura atingir esse equilíbrio, implementando essencialmente as instruções que são mais comuns nos processadores comerciais, ao mesmo tempo que evita por algumas soluções menos optimizadas mas mais claras e didácticas. Essa é uma das razões porque este livro não adota uma arquitectura comercial como processador de base.

As secções seguintes descrevem o PEPE essencialmente do ponto de vista do programador em linguagem assembly, através dos recursos a que ele pode ter acesso (registos, memória, etc.) e da funcionalidade das diversas instruções que o PEPE suporta. Não se trata de uma descrição exaustiva nem sequer organizada em formato de manual de referência do PEPE, mas apenas de introduzir as características de programação do PEPE, gradualmente e com exemplos. O Apêndice A contém informação sobre o PEPE de forma mais sistematizada, incluindo uma tabela com todas as suas instruções.

O capítulo 7 apresenta uma visão dos blocos internos e da implementação detalhada da arquitectura do PEPE.

É importante notar que, na descrição que se segue, algumas das instruções em assembly apresentam uma única mnemónica para uma série de instruções máquina (*opcodes*) diferentes, a que correspondem funcionalidades na mesma classe mas com processamentos pelo hardware diferenciados (o exemplo mais expressivo desta variedade é a instrução MOV, descrita na secção 4.10).

Estas instruções, embora com a mesma mnemónica, diferem no número e/ou tipo dos operandos, de forma a evitar ambiguidades quanto ao significado de cada instrução. É apenas uma questão de facilitar a vida ao programador, que assim tem de fixar menos mnemónicas, concentrando-se na funcionalidade e não tanto na forma. O assembler gera o código-máquina com base não apenas nas mnemónicas mas também nos operandos identificando assim todas as situações.

Também há um caso em que uma instrução em linguagem assembly pode gerar uma ou duas instruções em código-máquina (também no caso da instrução MOV), dependendo se o

valor da constante envolvida seja pequeno (ou não) para poder ser representado em 8 bits). Mais uma vez, o assembler facilita a vida do programador, analisando a constante em causa e decidindo se pode gerar apenas uma instrução ou se tem de gerar duas.

Assim, para cada uma das instruções do PEPE, é indicado não apenas a mnemónica usada na linguagem assembly mas também uma mnemónica única para cada opcode que representa cada combinação de número e tipo de operandos dessa instrução.

Por outro lado, a notação RTL simples definida na Tabela 3.4, na página 128, tem de ser estendida para contemplar as novas características do PEPE, pelo que se deve agora ter em conta as convenções estabelecidas pela Tabela 4.10. Alguns detalhes da notação só serão explicados nas secções seguintes.

O leitor mais atento poderá notar que algumas instruções são implementadas por uma sequência de operações elementares, o que impede que essas instruções se executem num só ciclo de relógio, ao contrário do que sucede no PEPE-8 (mas não tem nada a ver com a largura do processador). Este assunto é muito importante, mas afecta essencialmente a implementação e muito pouco a funcionalidade, pelo que só será retomado no capítulo 7.

### ESSENCIAL

- Um processador tem um conjunto limitado de instruções, que são as operações que ele sabe fazer. Um programa é implementado por uma sequência arbitrariamente complexa destas instruções básicas;
- Embora haja instruções que implementam operações semelhantes, variando essencialmente no tipo de operandos, cada instrução tem um opcode único (para indicar o processamento diferenciado);
- Do ponto de vista do programador, é útil usar a mesma mnemónica para as operações que variam apenas nos operandos. O assembler suporta isto ao nível da linguagem assembly mas gera instruções máquina diferentes para cada caso;
- Ainda para facilitar a vida ao programador, o assembler pode gerar mais do que uma instrução máquina para uma dada instrução de assembly;
- As instruções devem ser especificadas não apenas em texto mas também em RTL, um conjunto de convenções específicas de cada processador;
- Algumas instruções envolvem uma sequência de operações elementares, pelo que nem todas são executadas apenas num ciclo de relógio.

As instruções descritas ao longo deste capítulo serão objecto de exemplos e de simulações. O objectivo é explicar a funcionalidade e não os detalhes de implementação. Por conseguinte, o PEPE é simulado exclusivamente a nível funcional (como uma caixa preta), sendo relevantes apenas aspectos como os registos e as ligações à memória.

| SÍMBOLOGIA     | SINÔNIMO                                                                                                                                                                                                                                                                                                                                           | EXEMPLO                                                                                     |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| Rd             | Registo (1.º operando/destino). Pode ser R0 a R15 (incluindo as designações específicas SP, RE e TEMP)                                                                                                                                                                                                                                             | R2                                                                                          |
| Rs             | Registo (2.º operando/origen). Pode ser R0 a R15 (incluindo as designações específicas SP, RE e TEMP)                                                                                                                                                                                                                                              | R2                                                                                          |
| R1             | Registo (3.º operando, usado em algumas instruções de acesso à memória. Pode ser R0 a R15 (incluindo as designações específicas SP, RE e TEMP))                                                                                                                                                                                                    | R2                                                                                          |
| PC             | Program Counter                                                                                                                                                                                                                                                                                                                                    | PC                                                                                          |
| SP, RE, TEMP   | Designações alternativas para os registos R12, R13 e R15, respectivamente                                                                                                                                                                                                                                                                          | SP, RE, TEMP                                                                                |
| k              | Constante com 4, 8 ou 12 bits, com ou sem sinal (de acordo com o indicado onde foi usado)                                                                                                                                                                                                                                                          | k                                                                                           |
| Mw[end]        | Célula de memória de 16 bits que ocupa os endereços end e end+1 (end tem de ser um endereço par)                                                                                                                                                                                                                                                   | Mw[R1]                                                                                      |
| Mb[end]        | Célula de memória de 8 bits cujo endereço é end (que pode ser par ou ímpar)                                                                                                                                                                                                                                                                        | Mb[R3]                                                                                      |
| Ra(i)          | Bits i do registo Ra; i ∈ [0..15]                                                                                                                                                                                                                                                                                                                  | R2(3)                                                                                       |
| Ra(j..j)       | Bits i a j (contíguos) do registo Ra; i, j ∈ [0..15]; i ≥ j                                                                                                                                                                                                                                                                                        | R2(15..7)                                                                                   |
|                | Concatenação de bits (os bits do operando da esquerda ficam à esquerda, ou com maior peso)                                                                                                                                                                                                                                                         | R1 ← R2(15..8)    00H                                                                       |
| bk(n)          | Attribuição do valor de uma expressão (expr) a uma célula de memória ou registo (dest). Um dos operandos da atribuição (expressão ou destino) tem de ser um registo ou um conjunto de bits dentro do processador. O operando da direita é todo calculado primeiro e só depois se destroi o operando da esquerda, colocando lá o resultado de expr. | 0{4} equivale a 0000<br>R1(15..12) equivale a R1(15..12)    R1(15)<br>R1(17..0) ← R2(15..8) |
| dest ← expr    | Attribuição do valor de uma expressão (expr) a uma célula de memória ou registo (dest). Um dos operandos da atribuição (expressão ou destino) tem de ser um registo ou um conjunto de bits dentro do processador. O operando da direita é todo calculado primeiro e só depois se destroi o operando da esquerda, colocando lá o resultado de expr. | R1 ← Mw[R2]<br>Mw[R0] ← R4 + R2<br>R1(17..0) ← R2(15..8)                                    |
| (expr) : accão | Executa accão se expr for verdadeira (expr tem de ser uma expressão booleana)                                                                                                                                                                                                                                                                      | (Z=1) : PC ← k                                                                              |
| ~, ∨, ⊕        | E, OU, OU-exclusivo                                                                                                                                                                                                                                                                                                                                | R1 ← R2 ∧ R3<br>Z ← 0                                                                       |
| Z, N, C, V     | Bits de estado no RE – Registo de Estado                                                                                                                                                                                                                                                                                                           |                                                                                             |

Tabela 4.10 - Notação RTL usada para descrever as instruções do PEPE

O circuito de base para as simulações é o da Fig. 4.7, com alguns detalhes explicados nos próprios guias práticos das simulações, quando relevante. Este circuito é essencialmente o da Fig. 4.3, em que os detalhes internos do processador foram omitidos e a transferência de dados/instruções para a (ou a partir da) memória usa apenas uma ligação bidireccional,

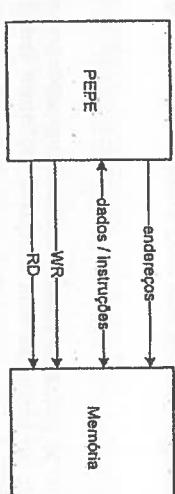


Fig. 4.7 - Circuito de base para exemplificar as instruções do PEPE

Os periféricos são conceptualmente idênticos à memória, pelo que o PEPE não tem instruções específicas para lidar com eles. Todas as instruções de acesso à memória podem também ser usadas em periféricos, com a mesma semântica.

## 4.9 INSTRUÇÕES DE SALTO

O contador de programa (PC) é um registo que não se refere explicitamente em nenhuma instrução, pois não é de utilização geral, mas isso não impede que seja usado em todas as instruções, indicando em que endereço da memória essa instrução está armazenada (Fig. 4.2, na página 188).

Normalmente, o PC é alterado automaticamente. Quando o PEPE arranca, após uma inicialização total (reset), o PC é inicializado com 0000H, endereço onde deve estar a primeira instrução a executar. Por cada instrução executada, o PC é incrementado de duas unidades (dado que o endereçamento no PEPE é de byte – ver secção 4.4), para passar a designar a instrução seguinte (2 bytes depois da instrução corrente).

Entretanto, há casos em que se pretende fazer um “salto” na sequência normal, colo- cando um valor diferente no PC, essencialmente nas seguintes situações:

- Implementar ciclos, tipicamente “voltando atrás” e repetindo a execução de algumas instruções;
- Tomar decisões booleanas (com duas hipóteses), em que numa hipótese se passa à instrução a seguir e noutra se salta para outro endereço qualquer;
- Salter por cima de instruções que não interesse executar (quando a sequência normal de instruções não constitui um bloco contíguo de instruções).

Esse é o papel das instruções de salto, que alteram o valor do PC. Há vários tipos de instruções de salto, de acordo com a Tabela 4.11, que estabelece as combinações possíveis. O salto pode ser classificado quanto à forma de determinação do endereço:
 

- Absoluto – Se o PC for carregado com um valor de 16 bits (conteúdo de um registo), que especifica completamente o endereço para onde saltar, independentemente do endereço que o PC continha anteriormente;

uma vez que a memória ou está a ser lida ou ser escrita, mas nunca as duas operações ao mesmo tempo (em cada acesso, apenas um dos dois sinais, WR ou RD, está activo).

- Relativo – Se o novo endereço for obtido somando uma constante  $k$  ao valor que o PC tem. Assume-se que a constante  $k$  está em complemento para 2 e permite saltar para "a frente" se  $k > 0$  e "para trás" se  $k < 0$ .

E quanto à sua realização:

- Incondicionais – Se o salto for sempre efectuado, independentemente de qualquer condição;
- Condicionais – Se o salto só for efectuado se uma dada condição for verdadeira.

| SALTOS         | ABSOLUTOS | RELATIVOS | CONDIÇÕES         | (APÓS ESTA INSTRUÇÃO) |                     |
|----------------|-----------|-----------|-------------------|-----------------------|---------------------|
|                |           |           |                   | ADD A, B              | CMP A, B            |
| Incondicionais | JMP       | Rs        | JMP               | $k$                   | Nenhuma             |
|                | JZ (JEQ)  | k         | Z = 1             | a + b = 0             | a = b               |
|                | JNZ (JNE) | k         | Z = 0             | a + b > 0             | a > b               |
|                | JN        | k         | N = 1             | a + b < 0             | a < b               |
|                | JNN       | k         | N = 0             | a + b ≥ 0             |                     |
|                | JP        | k         | (N ∨ Z) = 0       | a + b > 0             |                     |
|                | JNP       | k         | (N ∨ Z) = 1       | a + b ≤ 0             |                     |
|                | JC (JB)   | k         | C = 1             | Der transporte        | a < b <sup>29</sup> |
|                | JNC (JAE) | k         | C = 0             | Der excesso           | a ≥ b <sup>29</sup> |
|                | JV        | k         | V = 1             | Der excesso           |                     |
| Condicionais   | JNV       | k         | V = 0             | Não der excesso       |                     |
|                | JLT       | k         | (N ⊕ V) = 1       | a < b                 |                     |
|                | JLE       | k         | ((N ⊕ V) ∨ Z) = 1 | a ≤ b                 |                     |
|                | JGT       | k         | (N ⊕ V) = 0       | a > b                 |                     |
|                | JGE       | k         | ((N ⊕ V) ∨ Z) = 0 | a ≥ b                 |                     |
|                | JAE       | k         | (C ∨ V) = 0       | a > b <sup>29</sup>   |                     |
|                | JBE       | k         | (C ∨ V) = 1       | a ≤ b <sup>29</sup>   |                     |
|                | RTI       |           |                   | condição: PC ← PC + k |                     |
|                | PC ← RS   |           |                   |                       |                     |

Tabela 4.11 - Instruções de salto do PEPE

Os saltos absolutos utilizam um registo como operando (JMP R1, por exemplo), pois é a única forma de especificar um endereço de 16 bits numa só instrução. Isto tem a vantagem de que o endereço para onde se quer saltar pode variar ao longo da execução do programa e permite mesmo definir uma tabela de endereços para onde saltar, de acordo com um dado índice que indique o elemento da tabela que se pretende usar como endereço de salto. A secção 5.6.2 apresenta um exemplo de utilização desta possibilidade.

A constante  $k$  tem 12 bits no caso do JMP e 8 bits nas restantes instruções de salto (nestas há menos bits porque é preciso codificar o tipo de condição). Em complemento para 2, permite sair cerca de metade da gama de valores para trás e para a frente (valores positivos e negativos). No entanto, o PC é sempre par e o mesmo tem de suceder à

<sup>29</sup> No caso em que a e b são considerados sem sinal. Usado para comparar endereços, que podem variar entre 0 e 65535 e não entre -32768 e +32767. Ou seja, são números considerados sem sinal.

constante, pelo que não tem interesse incluir o bit de menor peso da constante na instrução (é sempre 0...). Logo, o que é guardado na instrução não é  $k$  mas sim  $k/2$  (ao executar a instrução, o hardware multiplica este valor por 2 antes de o somar ao PC). Isto permite, na prática, estender a gama de saltos, que no JMP varia entre  $-2^{12}$  e  $+2^{12}-1$  e nos saltos condicionais entre  $-2^8$  e  $+2^8-1$ .

As duas últimas colunas da Tabela 4.11 reflectem os dois tipos de operações que podem preceder as instruções de salto condicionais:

- Operações aritméticas, lógicas e de deslocamento – Produzem um resultado. Na penúltima coluna estão indicadas as condições que permitem testar o resultado, não apenas no valor em si (com os bits Z e N) mas também na sua validade (com os bits C e V);

- Operações relacionais (CMP e SUB) – O resultado expressa a relação de grandeza entre os dois operandos, que devem ser números em complemento para 2. Note-se que estas condições testam também o bit de excesso porque os operandos são entendidos como números em complemento para 2 e, como está uma subtração envolvida, se os operandos tiverem sinal contrário pode dar excesso. Mesmo nesse caso deve dar-se a resposta pretendida, pois cada operando é válido e a relação entre eles também.

Enquanto no primeiro caso as condições de salto são orientadas directamente aos bits de estado (JZ, JNC, etc.), no segundo as instruções de salto reflectem a relação entre os operandos, que podem ser:

- Números em complemento para 2 (positivos ou negativos, entre -32.768 e +32.767) – JEQ, JNE, JLT, JLE, JGT, JGE (salta se for igual, diferente, menor que, menor ou igual, maior que e maior ou igual, respectivamente);
- Endereços, considerados sem sinal (só positivos, entre 0 e 65.535) – JA, JAE, JB, JBE (salta se estiver acima, acima ou igual, abaixo ou igual, respetivamente). Tipicamente, estes saltos são usados pelos compiladores para efectuar operações aritméticas com apontadores (secção 5.5.4, na página 294).

Algumas destas instruções de salto condicional são iguais, na prática, a outras orientadas aos bits de estado, tal como indicado na Tabela 4.11.

Um aspecto fundamental do funcionamento é que o PC é alterado (somado com 2) ainda antes da execução de uma dada instrução. Ou seja, quando uma qualquer instrução é executada, já o PC tem o endereço da instrução seguinte. Por esse motivo, a instrução:

```
JMP 0 ; PC ← PC + 0
```

não faz nada, isto é, a execução continua normalmente na instrução seguinte. Pelo contrário, a instrução:

```
JMP -2 ; PC ← PC - 2
```

provoca um ciclo infinito, isto é, ao executar a instrução o processador salta para a própria instrução, que executa repetidamente. Este esquema não é nada intuitivo ("Então

para saltar para o endereço em que estou agora tenho de saltar para lá?" mas a sua razão de ser será explicada na secção 5.7.2.1, na página 316.

As instruções de salto são das mais usadas em todos os programas. Os exemplos do resto deste capítulo ilustram o funcionamento destas instruções.

#### ESENCEIA

- O PC indica qual a instrução corrente e evoluí normalmente de 2 em 2 (devido ao endereçamento de byte).

Um salto corresponde a colocar um valor diferente no PC, o que é efectuado pelas instruções de salto.

Um salto pode ser classificado como absoluto (no PC é colocado um endereço de 16 bits vindo de outro registo) ou relativo (o PC soma-se o valor de uma constante em complemento para 2);

- Um salto pode também ser classificado como unconditional (sai independente de qualquer condição) ou conditional (só salta se uma dada condição for verdadeira).
- As condições podem ser classificadas como de resultado (teste aos bits de estado após uma operação aritmética, lógica ou de deslocamento) ou de relação (após uma comparação entre os operandos).

- Quando uma instrução no PC já contém o endereço da instrução a seguir (é isso que o salto relativo faz) tem efeito e JMP = 2 faz voltar a executar a mesma instrução.

## 4.10 INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

### 4.10.1 COMBINAÇÕES DE OPERANDOS

Por instruções de transferência de dados entendem-se todas aquelas que copiam dados de um ponto para outro do computador, sem os transformar. Esta secção lida apenas com algumas instruções destas, em especial a instrução mov. Há mais instruções de transferência de dados, que por serem mais complexas são tratadas apenas na secção 5.7.3.1, na página 334. Uma instrução pode fazer transferência de dados entre os recursos seguintes:

- Registros (contidos no banco de registos da Fig. 4.2);
- Células de memória (em que se incluem os periféricos porque do ponto de vista do processador são indistinguíveis);
- Constantes (a contida na própria instrução, se for o caso).

Uma transferência de dados tem um recurso de origem e um recurso de destino. Esta operação transfere (copia) o valor do recurso de origem para o recurso de destino, o que pode ser expresso em RTL por:

*destino*  $\leftarrow$  *origem*

O valor de origem nunca é alterado, ao passo que o valor anterior de destino é sempre destruído e substituído por uma cópia do valor de origem.

A instrução swap altera origem porque troca os valores de origem e destino, mas faz duas transferências na mesma instrução (secção 4.10.2).

É possível estabelecer várias combinações para o par de recursos destino-origem<sup>30</sup>, tal como indicado na Tabela 4.12, que indica qual a secção em que cada uma é tratada.

Note-se que:

- Naturalmente, as combinações com constante do lado esquerdo não são possíveis (não se pode alterar o valor de uma constante);
- As combinações que não envolvem um registo (as duas últimas) não são na realidade possíveis ('ver porque na secção 4.10.5). São incluídas apenas para completar o cenário e para mostrar como a sua funcionalidade (bastante útil, aliás) pode ser implementada por uma sequência de instruções.

| COMBINAÇÃO DE DESTINO-ORIGEM | Descrição                                  | ONDE SE DETALHADA |
|------------------------------|--------------------------------------------|-------------------|
| Registo-registo              | Cópia de valores entre dois registo        | Secção 4.10.2     |
| Registo-constante            | Carregamento imediato de um registo        | Secção 4.10.3     |
| Registo-memória              | Lectura da memória                         |                   |
| Memória-registo              | Escrita na memória                         | Secção 4.10.4     |
| Memória-constante            | Initialização da memória com uma constante | Secção 4.10.5     |
| Memória-memória              | Cópia entre células de memória             |                   |

Tabela 4.12 - Combinações possíveis dos operandos numa transferência de dados

Estas instruções não afectam os bits de estado, essencialmente por duas razões:

- Há situações em que o RE, com os seus bits de estado, tem de ser guardado nouro registo ou na memória, para mais tarde se repor o seu valor. Esta simples operação de transferência em si não poderia obviamente mudar os bits de estado. Poder-se-ia tratar do caso do RE como uma situação especial, mas as exceções à regra são sempre de evitar em nome da simplicidade;

<sup>30</sup>destino aparece do lado esquerdo para ser consistente com a ordem na expressão em RTL.

Trata-se apenas de uma operação de cópia de um valor para outro recurso (registo ou memória), e não a produção de um novo valor. Isto é, não há processamento nem transformação dos dados.

#### 4.10.2 TRANSFERÊNCIAS ENTRE REGISTOS

Há duas instruções nesta categoria, cuja sintaxe em linguagem assembly e descrição em RTL estão representadas na Tabela 4.13:

- A instrução `MOV31` entre registos é muito usada, essencialmente para obter uma cópia de um valor (guardado num registo) para fazer alguma operação que possa alterar essa cópia, sem perder o valor original (que fica ainda no registo a partir do qual foi copiado), ou então para ficar com uma cópia após o que se pode alterar o registo inicial;
- A instrução `SWAP` ("trocar") é menos usada mas ainda assim útil para situações em que se pretende alterar o valor de um registo sem perder o seu valor antigo. Assim, produz-se o novo valor num outro registo temporário e depois executa-se a instrução. O registo em causa ficará com o novo valor e o registo temporário ficará com o valor antigo. Naturalmente, esta instrução poderia ser substituída por três transferências simples (instruções `MOV`) e recurso a um terceiro registo para realizar a troca, pelo que a sua existência se justifica apenas em termos de facilidade de programação e clareza dos programas. O Programa 4.9, na página 240, contém um exemplo de utilização da instrução `SWAP`.

| ASSEMBLY                 | EXEMPLO                  | MÁQUINA                                        | RTL                                              | EFEITO                                                                                                     |
|--------------------------|--------------------------|------------------------------------------------|--------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <code>MOV Rd, Rs</code>  | <code>MOV R0, R1</code>  | <code>MOV R<sub>d</sub>, R<sub>s</sub></code>  | <code>Rd ← Rs</code>                             | Rd fica com uma cópia do valor de Rs. O valor anterior de Rd é destruído                                   |
| <code>SWAP Rd, Rs</code> | <code>SWAP R0, R1</code> | <code>SWAP R<sub>d</sub>, R<sub>s</sub></code> | <code>TEMP ← Rd<br/>Rd ← Rs<br/>Rs ← TEMP</code> | Os registos Rd e Rs ficam com os valores trocados. Esta instrução destrói o valor anterior do registo TEMP |

Tabela 4.13 - Instruções de transferência de dados entre registos

Juntamente com a sintaxe em assembly é dada também a instrução máquina em termos simbólicos, com uma constante (que indica o opcode) e os parâmetros dessa instrução.

O funcionamento destas instruções é ilustrado pela Tabela 4.14, que contém um conjunto de instruções, executadas sequencialmente, e a evolução de alguns registos do processador ao longo dessa execução. Os restantes registos não são alterados por estas instruções (com exceção do PC, naturalmente).

Na primeira coluna mostra-se o endereço em que cada instrução está localizada em memória. A primeira instrução ocupa o endereço 0000H, que é o endereço onde o

processador começa a executar as instruções, quando arranca após uma inicialização total (reset). Cada instrução ocupa 16 bits, pelo que os endereços das instruções evoluem de 2 em 2 unidades, devido ao endereçamento de byte do PEPE. Em cada linha, mostra-se o valor de cada registo após a execução da instrução dessa linha, com os registos que mudaram de valor destacados a negrito.

Assume-se que os registos foram previamente inicializados com os valores (meramente exemplificativos) indicados como estado inicial.

| ENDEREÇO | INSTRUÇÃO                | RTL                                              | R0                | R1    | R2                | R3                | TEMP              |
|----------|--------------------------|--------------------------------------------------|-------------------|-------|-------------------|-------------------|-------------------|
|          | Estado inicial           |                                                  | 1234H             | 5678H | 9ABC <sup>H</sup> | FFFFH             | 0000H             |
| 0000H    | <code>MOV R0, R2</code>  | <code>R0 ← R2</code>                             | 9ABC <sup>H</sup> | 5678H | 9ABC <sup>H</sup> | FFFFH             | 0000H             |
| 0002H    | <code>MOV R2, R1</code>  | <code>R2 ← R1</code>                             | 5678H             | 5678H | FFFFH             | 0000H             |                   |
| 0004H    | <code>MOV R1, R0</code>  | <code>R1 ← R0</code>                             | 9ABC <sup>H</sup> | 5678H | FFFFH             | 0000H             |                   |
| 0006H    | <code>SWAP R1, R3</code> | <code>TEMP ← R1<br/>R1 ← R3<br/>R3 ← TEMP</code> | 9ABC <sup>H</sup> | FFFFH | 5678H             | 9ABC <sup>H</sup> | 9ABC <sup>H</sup> |

Tabela 4.14 - Efeito das instruções de transferência de registos. Cada linha mostra o novo conteúdo dos registos após a execução da instrução respectiva

#### 4.10.3 TRANSFERÊNCIA DE DADOS ENTRE REGISTOS

Esta simulação ilustra o funcionamento das instruções de transferência de dados entre registos. O circuito a usar é o da Fig. 4.7. Os aspectos cobertos incluem os seguintes:

- Verificação do valor aleatório dos registos quando se liga o processador (neste caso, arranca a simulação);
- Verificação do endereço em que se localiza cada instrução;
- Inicialização manual dos registos relevantes;
- Execução passo a passo do programa, verificando o valor dos registos após execução de cada instrução.

#### 4.10.3 TRANSFERÊNCIA DE UMA CONSTANTE PARA UM REGISTO

Quando um programa arranca (começa a execução) nunca se sabe que valor tem um registo (tal como acontece com o conteúdo das células de memória). Se o computador acabou de ser ligado, o valor inicial de cada registo é aleatório, isto é, não se pode determinar. Se um programa anterior acabou, sabe-se lá que valores ficaram nos registos. Mesmo durante a execução de um programa, surge várias vezes a necessidade de colocar um certo valor (especificado numa instrução) num registo para posterior processamento. O que se pretende basicamente é copiar o valor de uma constante para um registo. Isto permitiria, por exemplo, inicializar os valores dos registos na Tabela 4.14.

<sup>31</sup> Abreviatura de "mover" de um local para outro.

`MOV registo, constante ; registo ← constante`

O problema é que, de acordo com a Tabela 4.6, este tipo de instrução (tipo de codificação 4) admite apenas constantes de 8 bits. Como inicializar um registo com uma constante maior (16 bits)?

Alguns processadores resolvem este problema alterando o número de bits da instrução. Se esta tiver uma constante, para além da palavra de base (*opcode*, registo, etc.) tem uma segunda palavra de 16 bits com o valor da constante. O problema é que isto complica a desodinização das instruções. Umas vezes as instruções têm 16 bits (uma palavra), outras vezes 32 (duas palavras).

O PEPE resolve este problema de forma mais simples e regular com uma solução estatisticamente mais optimizada (para os casos mais frequentes) e mais adaptativa:

- Em linguagem assembly, o programador usa sempre a instrução indicada acima, qualquer que seja o valor da constante (dentro da gama permitida pelos 16 bits);
- O assembler analisa o valor da constante antes de gerar o código-máquina e separa duas situações:

- Para constantes no intervalo [-128 .. +127], ou seja, representáveis em 8 bits, ou seja, nos intervalos [-32.768 .. -129] e [+128 .. +32.767], o assembler gera duas instruções, com opcodes MOVL e MOVH (Tabela 4.15), em que concatena cada um dos dois bytes do registo destino.

Com este esquema, o PEPE consegue gerar apenas uma instrução de 16 bits para as constantes mais pequenas, que constituem os casos mais frequentes. Nos restantes casos, o PEPE gasta 32 bits, mas com duas instruções, pelo que mantém sempre o tamanho das instruções em 16 bits.

| ASSEMBLY                             | EXEMPLOS                         | MAQUINA                           | RTL | EFEITO                                                                                                                                          |
|--------------------------------------|----------------------------------|-----------------------------------|-----|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>MOV Rd, k</code>               |                                  |                                   |     |                                                                                                                                                 |
| <code>-128 ≤ k ≤ +127</code>         |                                  |                                   |     |                                                                                                                                                 |
| <code>MOV R0, -128</code>            | <code>MOVLI, Rd, k</code>        | <code>Rd[7..0] ← k[7..0]</code>   |     | Rd fica com uma cópia do valor da constante (de 8 bits) estendida para 16 bits com o seu bit de sinal, k[7]. O valor anterior de Rd é destruído |
| <code>k &lt; -128 ou k ≥ +128</code> | <code>MOVLI, Rd, k[15..8]</code> | <code>Rd[15..8] ← k[15..8]</code> |     | Rd fica com uma cópia do valor da constante (de 16 bits). O valor anterior de Rd é destruído                                                    |
| <code>MOV R2, -32768</code>          | <code>MOVLI, Rd, k[7..0]</code>  | <code>Rd[7..0] ← k[7..0]</code>   |     |                                                                                                                                                 |
| <code>MOV R2, -1000</code>           | <code>MOVLI, Rd, k[7..0]</code>  | <code>Rd[7..0] ← k[7..0]</code>   |     |                                                                                                                                                 |
| <code>MOV R5, +500</code>            | <code>MOVLI, Rd, k[15..8]</code> | <code>Rd[15..8] ← k[15..8]</code> |     |                                                                                                                                                 |
| <code>MOV R8, +32767</code>          |                                  |                                   |     |                                                                                                                                                 |

Tabela 4.15 - Instruções de transferência de uma constante para um registo

A Tabela 4.15 mostra a razão de ser dos nomes das mnemónicas das instruções:

`MOVL` move apenas o byte de menor peso (ou, como se costuma dizer em inglês, o *byte Low*). O byte de maior peso (o *High*) fica constituído todo por 0s ou por 1s, de acordo com o sinal da constante (extensão de 8 bits para 16 bits com sinal):

- Se a constante for positiva, o bit de sinal é 0, o byte de maior peso fica todo a zero e o portanto o valor final da constante no registo também fica positivo;
- Se a constante for negativa, o bit de sinal é 1, o byte de maior peso fica todo a 1 e o portanto o valor final da constante no registo também fica negativo.

`MOVH` altera apenas o byte de maior peso (o *byte High*), sem afectar o de menor peso. A Tabela 4.17 explica melhor o uso desta instrução.

A instrução MOVL revela que uma constante pode ser pequena e precisar de poucos bits para ser representada, mas tem sempre de ser estendida para 16 bits, pois essa é a largura dos registos. A extensão por replicação do bit do sinal permite estender para 16 (ou mais) bits preservando o valor inicial da constante, seja positiva ou negativa.

A Tabela 4.16 ilustra como é que a extensão para 16 bits com sinal preserva o valor de uma constante com 8 bits. Note-se que em cada linha o valor da constante é sempre o mesmo. Apenas varia a base (10, 2 e 16) e o número de bits usado para a representação (8, 16 e 32).

| DECIMAL | BINÁRIO   | HEXADECIMAL (EXTENSÃO PARA 16 BITS) | HEXADECIMAL (EXTENSÃO PARA 32 BITS) |
|---------|-----------|-------------------------------------|-------------------------------------|
| 0       | 0000 0000 | 00H                                 | 00 00 00 00H                        |
| +1      | 0000 0001 | 01H                                 | 00 00 00 01H                        |
| +64     | 0100 0000 | 40H                                 | 00 00 40H                           |
| +127    | 0111 1111 | 7FH                                 | 00 00 7FH                           |
| -1      | 1111 1111 | FFH                                 | FF FF FF FFH                        |
| -2      | 1111 1110 | FEH                                 | FF FF FEH                           |
| -64     | 1100 0000 | COH                                 | FF COH                              |
| -128    | 1000 0000 | 80H                                 | FF 80H                              |

Tabela 4.16 - Extensão de um valor binário com sinal de 8 bits para 16 e 32 bits

O princípio de funcionamento é simples. Num número binário, os 0s à esquerda num número positivo são irrelevantes, qualquer que seja a sua quantidade. De igual modo, os 1s à esquerda num número negativo são irrelevantes, qualquer que seja a sua quantidade.

Tal como indicado na Tabela 4.15:

No caso de constantes entre -128 e +127 o assembler gera apenas uma instrução `MOVL`, que não só coloca a constante (8 bits) no byte de menor peso do registo como estende esses 8 bits para 16 mantendo o sinal;

No caso de constantes menores que -128 ou maiores que +127, o *assembler* gera primeiro uma instrução `MOVL`, que coloca o *byte* de menor peso da constante no *byte* de menor peso do registo. Em seguida, gera uma instrução `MOVH` que coloca o *byte* de maior peso da constante no *byte* de maior peso do registo. Note-se que o sinal pode ficar incorrecto após o `MOVL`, mas o `MOVH` acaba a operação correctamente.

**NOTA**

No caso das constantes de 16 bits, a instrução `MOVL` estende o *byte* de menor peso para 16 bits porque esse é o seu comportamento normal (isto está indicado na Tabela 4.15). Mas tal não tem significado pois o *byte* de menor peso não é um número positivo nem negativo. É apenas uma parte (o *byte* de menor peso) de um número cujo bit de sinal está na outra parte (no *byte* de maior peso). De qualquer forma não tem importância, pois a instrução `MOVH` coloca depois o *byte* de maior peso no seu valor correcto.

A Tabela 4.17 contém alguns exemplos de constantes a transferir para registo, programador especifica sempre a mesma instrução *assembly* (`MOV`).

| ASSEMBLY                    | CONSTANTE (HEXADECIMAL, 16 bits) | INSTRUÇÕES MÁQUINA                           | COMENTÁRIOS                                                        |
|-----------------------------|----------------------------------|----------------------------------------------|--------------------------------------------------------------------|
| <code>MOV R1, 0</code>      | 00 00H                           | <code>MOVL, R1, 00H</code>                   |                                                                    |
| <code>MOV R1, +1</code>     | 00 01H                           | <code>MOVL, R1, 01H</code>                   |                                                                    |
| <code>MOV R1, +127</code>   | 00 7FH                           | <code>MOVL, R1, 7FH</code>                   |                                                                    |
| <code>MOV R1, +32767</code> | 7F FFH                           | <code>MOVL, R1, FFH<br/>MOVH, R1, 7FH</code> | Após o <code>MOV<sub>L</sub></code> , R1 fica com FFFFH (negativo) |
| <code>MOV R1, -1</code>     | FF FFH                           | <code>MOVL, R1, FFH</code>                   |                                                                    |
| <code>MOV R1, -128</code>   | FF80H                            | <code>MOVL, R1, 80H</code>                   |                                                                    |
| <code>MOV R1, -32768</code> | 80 00H                           | <code>MOVL, R1, 00H<br/>MOVH, R1, 80H</code> | Após o <code>MOV<sub>L</sub></code> , R1 fica com 0000H (positivo) |

Tabela 4.17 - Inicialização de registo com constantes de 16 bits

**SIMULAÇÃO – INICIALIZAÇÃO UM REGISTRO COM UMA CONSTANTE**

Esta simulação ilustra o funcionamento das instruções de transferência de constantes para registo, tendo por base as tabelas desta secção. Os aspectos cobertos incluem os seguintes:

- Extensão de sinal de 8 bits para 16 bits, com números positivos e negativos;
- Funcionamento das instruções `MOVL` e `MOVH`;
- Geração de instruções pelo *assembler* para a instrução `MOV` com vários valores de constantes.

#### 4.10.4 TRANSFERÊNCIAS ENTRE UM REGISTO E A MEMÓRIA

##### 4.10.4.1 ENDEREÇOS CONSTANTES EM REGISTOS

Até aqui, as instruções de acesso à memória nos programas de exemplo têm usado directamente endereços como constantes entre parênteses rectos ([endereço]), tal como ilustrado pelo Programa 4.1, na página 185, ainda no contexto de um só registo.

Extrapolando esta técnica para o PEPE, com os seus 16 bits e os seus 16 registos, as instruções de acesso à memória seriam algo do estilo:

```
MOV [endereço], registo ; registo ← [endereço]
MOV [endereço], registo ; [endereço] ← registo
```

A palavra "seriam" indica que não são. De facto, não podem ser. Os endereços são de 16 bits, pelo que só a constante gastaria 16 bits, para além da indicação do registo envolvido e do opcode. Logo, não caberiam numa instrução de apenas 16 bits.

Uma possível solução seria usar para as constantes de endereço a mesma técnica das constantes de dados (duas instruções), mas neste caso a probabilidade de se necessitar de duas instruções é muito maior. Só os acessos aos primeiros 256 bytes da memória poderiam usar apenas uma instrução. O resto dos 64 K endereços precisaria sempre de duas instruções, o que tornaria os acessos à memória bastante lentos.

Assim, a solução é muito mais pragmática: o PEPE não suporta instruções com constantes de endereço. Em vez disso, usa registo para conter os endereços das células a aceder, mantendo a sintaxe dos parênteses rectos:

```
MOV [registro1], [registro2] ; registro1 ← M[registro2]
MOV [registro1], registro1 ; M[registro1] ← registro1
```

Se o valor do `registro2` fosse 1000H, então a célula de memória accedida por estas instruções teria o endereço 1000H, enquanto o `registro1` teria o conteúdo, isto é, o valor lido ou escrito nessa célula.

Para inicializar o valor do `registro2` com um endereço de 16 bits é então possível usar a técnica das duas instruções, usada pela instrução *assembly* `MOV registo, constante` (com as instruções máquina `MOVL` e `MOVH`).

Claro que isto aparentemente é ainda pior! Para acceder à memória já são precisas três instruções, duas para inicializar o registo do endereço e outra para fazer o acesso propriamente dito.

Felizmente, não é preciso inicializar um registo com um endereço em todos os acessos, pois é frequente:

- Haver várias instruções que accedem a uma mesma célula de memória, tal como pode ser observado no Programa 4.1, na página 185, em que os oito acessos à memória são feitos a apenas duas células diferentes. Enquanto houver registo disponíveis, pode reservar-se um registo para conter o endereço dessa célula de

memória e passar a usar esse registo, sem o alterar, sempre que se tenha de aceder a essa célula;

Ter os dados organizados em tabelas de valores (células localizada em endereços contíguos), pelo que é muito útil poder dizer que um dado registo contém o endereço de base (o endereço do primeiro elemento) da tabela e depois ter um índice (uma constante ou outro registo, para o índice poder ser variável) que some à base para determinar o endereço real de um dado elemento na tabela, tal como representado na Fig. 4.8.

Para aceder aos vários elementos de uma tabela, a ideia é manter o registo com o endereço de base com um valor fixo e ir variando o índice, ou de uma maneira manual (índice especificado como uma constante na própria instrução de acesso à memória) ou algorítmica (índice especificado como um registo cujo valor vai variando). A secção 5.8.4.1, na página 363, apresenta mais detalhes.

#### 4.10.4.2 MÓDOS DE ACESSO À MEMÓRIA EM 16 BITS

A Fig. 4.8 representa uma tabela em memória com oito elementos de 16 bits, a começar no endereço 1000H. O primeiro elemento está no endereço 1000H, o segundo em 1002H, etc. (de 2 em 2, pois cada elemento tem 16 bits, ou 2 bytes).

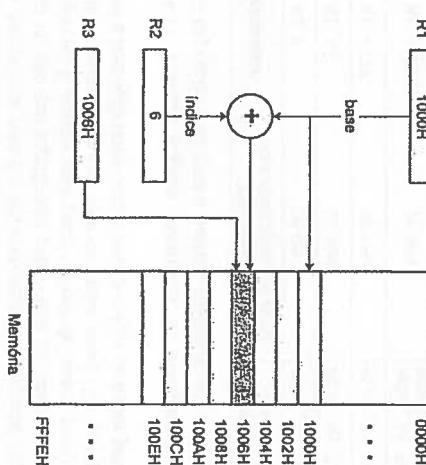


Fig. 4.8 - Acesso a tabelas em memória através de uma base (R1) e de um índice (R2) para obter o endereço do elemento pretendido. Também é possível aceder directamente, sem tabela (com R3)

O registo R1 "aponta" para (contém o endereço de) o primeiro elemento, ou seja, 1000H. O registo R2 contém 6 e o registo R3 contém 1006H. Nesta figura, pretende-se aceder à célula de memória a cinzento, localizada no endereço 1006H.

Há basicamente três formas de referenciar a célula de memória a cinzento, com uma instrução de leitura para o registo R0 (assumindo que R1, R2 e R3 foram previamente inicializados com os valores indicados na Fig. 4.8):

```

MOV    R0, [R3] ; R0 ← M[1006H] (acesso linear, sem reconhecer a tabela)
MOV    R0, [R1 + 6] ; R0 ← M[1000H + 6] (tabela com índice constante)
MOV    R0, [R1 + R2] ; R0 ← M[1000H + 6] (tabela com índice que pode variar)

```

- A primeira forma permite aceder a qualquer célula da memória, considerando o espaço de endereçamento linear (sem usar nenhuma estrutura de dados);
- O índice constante acede a uma tabela (cuja base está no endereço contido em R1) em que se conhece a função de cada elemento dentro da tabela e portanto se referencia directamente (neste exemplo, com o índice 6);
- A última forma, com índice variável (registo), permite a um programa percorrer a tabela, efectuando algum processamento sobre cada elemento da tabela.

Note-se que, com endereçamento em bytes e elementos de 16 bits, um índice com o valor não se refere ao N-ésimo elemento mas sim ao N-ésimo byte. O índice tem assim de ir variando de 2 em 2 e deve ser par. A dualidade de unidades de contagem, em elementos e em bytes, é uma das consequências da existência de endereçamento de byte e aparece constantemente em programação em assembly. É fundamental termos este aspecto sempre presente! Dada a importância destas formas de endereçamento para a boa programação, as secções seguintes dão alguns exemplos da sua utilização.

#### 4.10.4.3 ACESSO À MEMÓRIA EM 16 BITS COM ÍNDICE VARIÁVEL

O acesso à memória em 16 bits com índice variável é ilustrada pelo Programa 4.3, que soma todos os valores da tabela. Ao contrário do que seria de esperar, o programa começa pelo último elemento, uma vez que a soma não depende da ordem e tal facilita a condição de paragem do ciclo.

| base           | EQU | 1000H         | ; endereço de base (do 1.º elemento) da tabela     |
|----------------|-----|---------------|----------------------------------------------------|
| tam            | EQU | 8             | ; tamanho da tabela (em elementos de 16 bits)      |
| <b>índice:</b> | MOV | R3, 0         | ; inicializa soma                                  |
|                | MOV | R1, base      | ; inicializa registo de base da tabela             |
|                | MOV | R2, tam       | ; número de elementos de 16 bits da tabela         |
|                | SUB | R2, 1         | ; índice do último elemento (o 1.º tem índice 0)   |
| <b>maiúsm:</b> | MOV | R4, R2        | ; cópia para não destruir o índice                 |
|                | ADD | R4, R4        | ; duplica para obter o índice em bytes             |
|                | MOV | R0, [R1 + R4] | ; é um elemento da memória (no endereço R1+R4)     |
|                | ADD | R3, R0        | ; acumula o elemento na soma                       |
|                | SUB | R2, 1         | ; acumula o índice (em bytes) do elemento anterior |
|                | JNN | maiúsm        | ; se já for negativo, o índice já era o último (0) |
| <b>fim:</b>    | JMP | fim           | ; senão, ainda há mais elementos                   |

Programa 4.3 - Exemplo de acesso à memória com índice variável (em elementos): soma de todos os elementos de uma tabela, começando pelo último

Uma alternativa de varimento da tabela, desta vez a "andar para a frente", é apresentada pelo Programa 4.4. O teste de fim já não pode ser comparando com zero mas sim com um limite. Para facilitar o teste, o tamanho da tabela é logo especificado em *bytes*. Assim, o índice pode evoluir em *bytes* e comparar directamente com o tamanho da tabela. Neste exemplo, o primeiro elemento tem índice 0 e o último índice 14.

Note-se que o termo de comparação (*tam*, ou 16) tem de estar num registo (R4) porque a instrução *CMP* (*compare*, ou comparar) só admite constantes em complemento para 2 com 4 bits, entre -8 e +7. Desta forma, *tam* já pode tomar qualquer valor.

```
base EQU 1000H ; endereço de base (do 1.º elemento) da tabela
tam EQU 16 ; tamanho da tabela (em bytes)

início: MOV R3, 0 ; inicializa soma
        MOV R1, base ; inicializa registo de base da tabela
        MOV R4, tam ; inicializa registo de base da tabela
        MOV R2, 0 ; inicializa índice do elemento a aceder
        MOV R0, [R1 + R2] ; lê um elemento da memória (no endereço R1+R2)
        ADD R3, R0 ; acumula o elemento na soma
        ADD R2, 2 ; obtém índice (em bytes) do elemento seguinte
        CMP R2, R4 ; se forem iguais, já chegou ao fim
        ADD R2, 2 ; se R2 e R4 não forem iguais, ainda há mais
        ADD R3, R0 ; acumula o elemento
        JNZ maisUm ; se R2 não for 0, ainda há mais elementos para somar
        JNP fim ; acabou. R3 contém a soma de todos os elementos

fim:
```

**Programa 4.4 - Exemplo de acesso à memória com índice variável (em bytes): soma de todos os elementos de uma tabela, começando pelo primeiro**

#### SIMULAÇÃO 4.4 – ACESSO À MEMÓRIA EM 16 BITS COM ÍNDICE VARIÁVEL

Esta simulação ilustra o acesso à memória em 16 bits com índice variável, tendo por base o Programa 4.4. Os aspectos cobertos incluem os seguintes:

- Inicialização dos valores das tabelas;
- Funcionamento individual da instrução de acesso à memória em 16 bits;
- Execução passo a passo do programa.

#### 4.10.4.5 ACESSO À MEMÓRIA EM 16 BITS COM ÍNDICE CONSTANTE

O endereçamento com índice constante usa-se essencialmente quando se tem um conjunto solidário de dados que se complementam para armazenar uma dada informação, como por exemplo uma ficha de cliente, com diversos dados como número de cliente, idade, etc. O índice constante permite logo aceder ao elemento que se pretende (tem de se saber a priori a ordem relativa dos vários elementos na estrutura de dados). O Programa 4.6 apresenta um exemplo de utilização de um índice constante.

Cabe ao programador em cada caso tomar a decisão sobre qual o melhor método para aceder à estrutura de dados.

##### A título de exemplo, imaginemos uma tabela de fichas de empregados de uma empresa.

Cada ficha contém quatro campos: o número mecanográfico do empregado, a sua extensão telefónica, a sua idade e o valor do seu salário base mensal (em euros). Assume-se que cada um destes valores gasta uma célula de memória de 16 bits.

Trata-se de uma simplificação, pois normalmente estas fichas incluem ainda outras informações, como nome, morada, etc. Mas é o suficiente para este exemplo. A Fig. 4.9 ilustra a utilização da memória para conter várias destas fichas (das quais apenas três fichas estão representadas na figura), uma para cada empregado.

Assume-se que a tabela das várias fichas começa no endereço 1000H e que cada ficha gasta 4 palavras (8 bytes), pelo que a 1.ª ficha começa em 1000H, a 2.ª em 1008H, a 3.ª em 1010H, a 4.ª em 1018H, etc.

```
base EQU 1000H ; endereço de base (do 1.º elemento) da tabela
tam EQU 8 ; tamanho da tabela (em elementos de 16 bits)

início: MOV R3, 0 ; inicializa soma
        MOV R1, base ; inicializa registo de base da tabela
        MOV R0, [R1] ; lê um elemento da memória (no endereço dado por R1)
        ADD R3, R0 ; acumula o elemento na soma
        ADD R1, 2 ; endereço (em bytes) do próximo elemento
        SUB R2, 1 ; menos um elemento para somar
        JNZ maisUm ; se R2 não for 0, ainda há mais elementos para somar
        JNP fim ; acabou. R3 contém a soma de todos os elementos

fim:
```

**Programa 4.5 - Exemplo de acesso à memória sem índice (com endereçamento linear): soma de todos os elementos de uma tabela**

#### SIMULAÇÃO 4.5 – ACESSO À MEMÓRIA EM 16 BITS SEM ÍNDICE

Esta simulação ilustra o acesso à memória em 16 bits sem índice, tendo por base o Programa 4.5. Os aspectos cobertos incluem os seguintes:

- Inicialização dos valores das tabelas;
- Funcionamento individual da instrução de acesso à memória em 16 bits;
- Execução passo a passo do programa.

O Programa 4.6 ilustra o acesso a esta estrutura de dados. O objectivo é calcular o encargo mensal da empresa com todos os seus empregados, tendo em atenção que o valor pago a cada empregado é a soma do ordenado base com um bónus em euros igual à sua idade (por simplicidade, ignoram-se todos os outros encargos, como as contribuições para a segurança social).

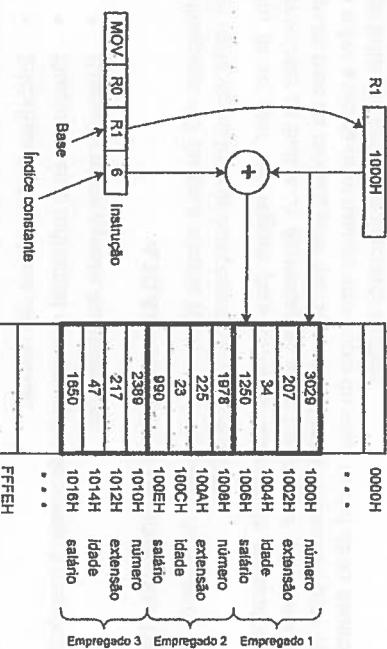


Fig. 4.9 - Acesso a tabelas em memória através de uma base e de um índice constante (instrução `MOV R0, [R1+6]`)

Dentro de cada ficha, cada um dos campos é accedido somando um índice constante ao endereço de base de cada ficha, tal como indicado na Tabela 4.18. Este índice pode ser constante porque a posição relativa dos vários campos dentro de uma ficha é fixa.

| FICHA | BASE DA FICHA | CAMPO    | VALOR | ÍNDICE DO CAMPO | ENDEREÇO DO CAMPO |
|-------|---------------|----------|-------|-----------------|-------------------|
| 1.a   | 1000H         | Número   | 3029  | 0               | 1000H             |
|       |               | Extensão | 207   | 2               | 1002H             |
|       |               | Idade    | 34    | 4               | 1004H             |
|       |               | Salário  | 1250  | 6               | 1006H             |
| 2.a   | 1008H         | Número   | 1978  | 0               | 1008H             |
|       |               | Extensão | 225   | 2               | 100AH             |
|       |               | Idade    | 47    | 4               | 1014H             |
|       |               | Salário  | 1650  | 6               | 1016H             |
| 3.a   | 1010H         | Número   | 2389  | 0               | 1010H             |
|       |               | Extensão | 234   | 2               | 1012H             |
|       |               | Idade    | 58    | 4               | 1014H             |
|       |               | Salário  | 2025  | 6               | 1016H             |

Tabela 4.18 - Acesso aos campos das fichas pela soma do endereço base de cada ficha com o índice de cada campo da ficha

#### Programa 4.6 - Exemplo de acesso à memória com índice constante

Assume-se que as fichas de todos os empregados estão contíguas em memória e foram previamente preenchidas com os dados dos empregados respectivos.

Note-se que o tamanho em bytes de cada ficha (tam, ou 8) tem de ser colocado num registo (R4) antes de ser somada com R1 (o endereço de base da ficha corrente) porque a instrução ADD (soma) só admite constantes em complemento para 2 com 4 bits, entre -8 e 8.

Assim, tam já pode tomar qualquer valor.

#### Simulação - ACESSO À MEMÓRIA EM 16 BITS COM ÍNDICE CONSTANTE

Esta simulação ilustra o acesso à memória em 16 bits com índice constante, tendo por base o Programa 4.6. Os aspectos cobertos incluem os seguintes:

- Inicialização dos valores das tabelas;
- Funcionamento individual da instrução de acesso à memória em 16 bits;
- Execução passo a passo do programa.

#### 4.10.4.6 INSTRUÇÕES DE ACESSO À MEMÓRIA EM 16 BITS

- O acesso com endereços lineares (primeira instrução) é na realidade um acesso de tabela com índice constante zero (como se a tabela tivesse apenas um elemento);

- O acesso à memória em palavra (16 bits) é explicitado pela notação `Mw[endereço]`, em que *endereço* tem de ser par;

apenas são previstas instruções com endereçamento linear, em que o endereço do byte a aceder está num registo.

A Tabela 4.20 especifica essas instruções de acesso à memória em 8 bits.

| L    | A S E M B L A G E    | R L                                                | M A Q U I N A            | E F E T O                                                 |
|------|----------------------|----------------------------------------------------|--------------------------|-----------------------------------------------------------|
| MOV  | Rd, [RS]             | Rd $\leftarrow$ Mw[Rs]                             | LDO, Rs, 0               | Leitura com endereçamento linear                          |
| MOV  | Rd, [RS + K]         | Rd $\leftarrow$ Mw[Rs + K]                         | LDO, Rs, k<br>(-16..+14) | Leitura de tabela, base Rs e índice k par                 |
| MOV  | Rd, [RS + RI]        | Rd $\leftarrow$ Mw[Rs + Rj]                        | LDR, RS, RI              | Lêitura de tabela, base Rs e índice RI                    |
| MOV  | [Rd], RS             | Mw[Rd] $\leftarrow$ Rs                             | STO, RS, 0               | Escrita com endereçamento linear                          |
| MOV  | [Rd + K], RS         | Mw[Rd + K] $\leftarrow$ RS                         | STO, RS, k<br>(-16..+14) | Escrita de tabela, base Rs e índice k par                 |
| MOV  | [Rd + RI], RS        | Mw[Rd + RI] $\leftarrow$ RS                        | STR, RS, RI              | Escrita de tabela, base RS e índice RI                    |
| SWAP | Rd, [RS] ou [RS], Rd | TEMP $\leftarrow$ Mw[RS]<br>Mw[RS] $\leftarrow$ Rd | SWAPM, Rd, RS            | Troca entre o valor de um registo e uma célula de memória |

Tabela 4.19 - Instruções de acesso à memória em 16 bits

- A própria constante no acesso a uma tabela com índice constante tem de ser pequena (tal como o R<sub>5</sub>, neste caso). Esta constante é codificada na instrução com apenas 6 bits (16 valores), mas para aumentar a sua gama de valores, uma vez que o ac-

- se destina a palavras e não a bytes, admite-se que o seu valor sempre par vai de  $-16\text{ a }+14$  (de 2 em 2) em vez de  $-8\text{ a }+7$  (de 1 em 1).<sup>32</sup> Para percorrer tabelas contém mais do que 16 elementos tem de se usar o acesso com índice variável (R1);

No acesso à memória com índice variável (R1), apenas é obrigatório que a soma da base e do índice seja par. O tamanho limite da tabela neste caso é 64 KB; isto ou seja, esta instrução permite aceder a qualquer endereço em memória;

A instrução SWAP entre um registo e a memória (também existe uma versão

registos – ver secção 4.10.2) permite efectuar de uma só assentada uma leitura

Para além da funcionalidade interessante que proporciona, tem uma aplicação

muito importante na implementação de sistemas operativos (secção 7.3.1, página 671).

#### **4.10.4.7 ACESSO À MEMÓRIA EM 8-BIT**

O suporte para enderecamento de *byte* implica poder aceder à memória *byte* a *byte*. Isto impõe limitações de codificação das instruções (o número de *opcodes* disponíveis é limitado).

<sup>32</sup> Na realidade, o que é codificado na instrução é metade do valor da constante. O hardware processador encarrega-se depois de duplicar o valor antes de o somar ao registo com o endereço base da tabela.

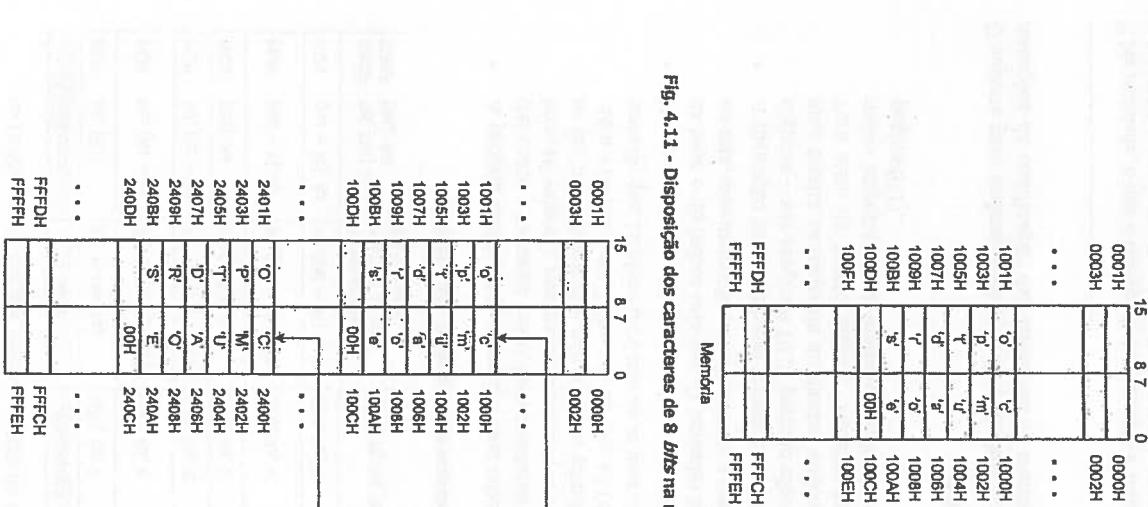
Table 4-20 - Instalaciones de casseno à plamártjor am 8 horeas

| <b>ASSEMBLEIA</b>         | <b>RTL</b>                                                      | <b>MÁQUINA</b> | <b>EFEITO</b>                                                                                           |
|---------------------------|-----------------------------------------------------------------|----------------|---------------------------------------------------------------------------------------------------------|
| <b>MOV B</b><br>Rd, [Rs]  | $Rd(7..0) \leftarrow Mb[Rs]$<br>$Rd(15..8) \leftarrow 0\{8\}$   | LDB, Rd, Rs    | Lectura de um byte. O byte de maior peso do registo destino é colocado a 0H                             |
| <b>MOV BS</b><br>Rd, [Rs] | $Rd(7..0) \leftarrow Mb[Rs]$<br>$Rd(15..8) \leftarrow Rd(7..0)$ | LDB, Rd, Rs    | Lectura de um byte. O byte de maior peso do registo destino é a extensão do sinal do byte de menor peso |
| <b>MOV</b><br>[Rd], Rs    | $Mb[Rd] \leftarrow Rs(7..0)$                                    | STB, Rd, Rs    | Escrita na memória do byte de menor peso do registo Rs                                                  |

Fig. 4.10 - Disposição de uma cadeia de caracteres (*string*) em memória, usando codificação ASCII. O valor 00H serve de terminator

A Fig. 4.11 mostra como estes caracteres se dispõem na memória, usando uma largura de 16 bits. Note-se que um endereço referencia um byte e não uma palavra de 16 bits.

A Fig. 4.12 ilustra o que se pretende, representando a situação após a execução do programa, com a cadeia de caracteres mais terminador copiada para a zona de memória destinada por destino.



Digitized by srujanika@gmail.com

A Fig. 4.11 deixa implícito que se o processador executar a instrução `MOV R0, [R1]` (leitura da memória em 16 bits, em que se assume que R1 contém 1000H) o registo R0 fica com o carácter 'ó' (endereço 1001H) no seu byte de maior peso e o carácter 'c' (endereço 1000H) no seu byte de menor peso.

Esta dualidade é sempre uma fonte de confusão mas não se pode deixar de a referenciar porque ela existe e é um aspecto importante nos processadores comerciais. Para já, não é relevante porque se estivermos sempre no mesmo processador as leituras e escritas na memória usam o mesmo método. O problema só se põe quando dados tirados da memória de um computador são levados, tal como estão, para outro computador que use o outro método. A secção 6.1.5.2, na página 438, introduz este problema de forma mais completa e perceptível.

0) Programa 4/ : percorre esta cadeia, copiando os caracteres para outra localização em memória depois de os passar de letras minúsculas para maiúsculas. Na codificação ASCII, isto é muito fácil de fazer, bastando subtrair 20H ao valor de cada carácter (Apêndice E). Por exemplo, o carácter 'a' tem o valor 61H, enquanto 'A' tem o valor 41H (menos 20H). A partir de 1000H (origem) está a cadeia de caracteres original. O programa copia-a (passando os caracteres para maiúsculas) para os endereços a partir de 2400H (destino), molhando cópia do terminador (0FH). Naturalmente, estes endereços podiam ser outros quaisquer (embora se deva ter cuidado para não haver sobreposição das duas cadeias de caracteres).

Por exemplo, o carácter 'a' tem o valor 61H, enquanto 'A' tem o valor 41H (menos 20H). A partir de 1000H (origem) está a cadeia de caracteres original. O programa copia-a (passando os caracteres para maiúsculas) para os endereços a partir de 2400H (destino), «molhando» cópia do terminador (00H). Naturalmente, estes endereços podiam ser outros quaisquer (embora se deva ter cuidado para não haver sobreposição das duas cadeias de memória).

oxygen  
destino  
terminador  
diferença

EQU ; 100OH ; endereço do primeiro carácter da cadeia original  
EQU ; 240OH ; endereço do primeiro carácter da cadeia destino  
ORH ; valor do byte que indica o fim da cadeia original  
EQU 20H ; valor da diferença entre minúsculas e maiúsculas

|            | Origem             | Destino | Opção                                                   |
|------------|--------------------|---------|---------------------------------------------------------|
| origem     | EQU                | 1000H   |                                                         |
| destino    | EQU                | 2400H   |                                                         |
| terminador | EQU                | 00H     |                                                         |
| diferença  | EQU                | 20H     |                                                         |
| Início:    | MOV R1, origem     |         | ; endereço do primeiro carácter da cadeia original      |
|            | MOV R2, destino    |         | ; endereço do primeiro carácter da cadeia destino       |
|            | MOV R3, diferença  |         | ; valor do byte que indica o fim da cadeia original     |
|            |                    |         | ; valor da diferença entre minúsculas e maiúsculas      |
| mai_UM:    | MOV R0, [R1]       |         | ; apontador para o primeiro carácter origem             |
|            | CMP R0, terminador |         | ; coloca diferença num registo (usado na instrução SUB) |
|            | JZ acaba           |         | ; obtém o próximo carácter (acesso de 8 bits)           |
|            | SUB R0, R3         |         | ; ve se já o byte terminador                            |
|            | MOV B [R2], R0     |         | ; se sim, vai acabar                                    |
|            | ADD R1, 1          |         | converte letra para maiúscula                           |
|            | ADD R2, 1          |         | guarda carácter na zona de memória destino              |
|            | JMP maisUm         |         | endereço (em bytes) do próximo carácter origem          |
| acaba:     | MOV B [R2], R0     |         | endereço (em bytes) do próximo carácter destino         |
|            | JMP fim            |         | ; ainda há mais caracteres para tratar                  |
|            |                    |         | ; guarda terminador na zona de memória destino          |
|            |                    |         | ; acabou                                                |

Ej. 4-13 - Sintaxis de extendiendo un tipo existente en el Programa 4.

## Algumas notas sobre o Programa 4.7:

As instruções MOV de inicialização dos registos são de 16 bits, mas notem-se as instruções MOVA nas leituras e escritas da memória, que só manipulam 8 bits de cada vez. Esta distinção é muito importante (em particular na escrita) para o bom funcionamento do programa;

O leitor mais atento poderá perguntar porque é que se usa a instrução CMP para ver se R0 contém o byte terminador, em vez de executar logo o JZ (se o R0 já tiver o valor zero após o MOV.B, o JZ não funciona?). Tal deve-se ao facto de as instruções MOVB não afectarem os bits de estado. Por outras palavras, os bits de estado são actualizados apenas após uma operação da ALU (ADD, CMP, etc.) e não pelas simples operações de transferências de dados. Portanto, não basta o R0 ter o valor zero. Tem de se actualizar o bit de estado Z;

Também poderá surgir a questão de na instrução CMP se comparar um registo de 16 bits (R0) com uma constante em cuja definição só se especificaram 8 bits (terminador) numa instrução que só tem 4 bits para codificar a constante! Isto como discutido na secção 4.10.3, qualquer constante especificada numa instrução é primeiramente estendida para 16 bits (mantendo o sinal e o valor) e só depois usada. A definição de terminador na directiva EQU só especifica 8 bits (00H), mas o assembler considera que os zeros à esquerda não foram especificados, pelo que o valor considerado é 0000H. Embora sendo um valor de 16 bits, terminador [15:0] na realidade um valor entre -8 e +7, que é a gama de valores que a instrução CMP admite para a constante, pelo que na instrução apenas 4 bits são usados (nesto caso, todos a 0). O assembler daria um erro se terminador tivesse um valor diferente dequele intervalo, caso em que se teria de aplicar a técnica do item seguinte;

A instrução SUB, que converte minúsculas em maiúsculas, não pode receber o numero 20H directamente na instrução, pois só admite constantes com 4 bits (-8 a +7), pelo que é preciso usar um registo auxiliar (R3). Esta é uma limitação específica do PEPE, porque tem apenas 16 bits para codificar as instruções, e poderia não existir noutra processador, nomeadamente se for de 32 bits. Convém no entanto não esquecer que a linguagem assembly é específica de um dado processador e todos os detalhes têm de ser revisados com muito cuidado. Aliás, do mesmo problema sofre a instrução CME. Sendo 00H o valor do terminador, poderia ser usado directamente na instrução, mas se decidissemos mudar o valor do terminador para FFFH, por exemplo, ter-se-ia de usar um registo auxiliar, tal como aconteceu no caso do SUB (felizmente, o assembler avisa quando uma constante demasiado grande para poder ser usada directamente numa instrução);

Por simplicidade, o programa não testa se um dado carácter é letra ou não, assim que todos os caracteres são letras. Num programa mais elaborado seria necessário comparar o carácter com as letras limites ('a' e 'z'), para ver se o carácter teria um valor entre ambos (em ASCII, as letras têm codificações contíguas e ordenadas);

A actualização de R1 e de R2 é feita simplesmente acrescentando uma unidade, pois estes registos são usados para aceder à memória byte a byte;

A instrução MOVB no endereço com a etiqueta Acaba escreve zero na memória, embora não seja muito explícito. Usa-se apenas o facto de que o R0 tem o valor zero quando o controlo executa esta instrução. Não é permitido especificar um endereço de memória e uma constante (zero, neste caso) numa mesma instrução (secção 4.10.5).

## SIMULAÇÃO 4.7 – ACESSO À MEMÓRIA EM 8 BITS

Esta simulação ilustra o acesso à memória em 8 bits, tendo por base o Programa 4.7. Os aspectos cobertos incluem os seguintes:

- Inicialização da cadeia de caracteres;
- Execução passo a passo do programa.

## 4.10.4.8 ACESSO À MEMÓRIA EM 8 BITS E 16 BITS

A codificação ASCII é já bastante antiga e limitada porque tem apenas 8 bits para codificar os caracteres. Desde há alguns anos, tem-se adoptado outra codificação, Unicode, em que cada carácter usa 16 bits e que permite até 65.536 caracteres diferentes, suportando melhor a panóptica de símbolos dos restantes alfabetos mundiais, nomeadamente os da escrita asiática.

```
ORIGEM: EQU 1000H ; endereço do primeiro carácter da cadeia original
destino EQU 2400H ; endereço do primeiro carácter da cadeia destino
terminador EQU 00H ; valor do byte que indica o fim da cadeia original
diferença EQU 20H ; valor da diferença entre minúsculas e maiúsculas

ÍCIO: MOV R1, origem ; apontador para o primeiro carácter origem
MOV R3, diferença ; apontador para o primeiro carácter destino
MOV R0, [R1] ; guarda diferença num registo (usado no SUB abaixo)
CMP R0, terminador ; vai buscar o próximo carácter (acesso de 8 bits)
JZ acaba ; se sim, vai acabar

SUB R0, R3 ; converte letra para maiúscula
MOV [R2], R0 ; guarda carácter na memória destino (16 bits)
ADD R1, 1 ; endereço (em bytes) do próximo carácter origem
ADD R2, 2 ; endereço (em bytes) do próximo carácter destino
JMP maisUm ; ainda há mais caracteres para tratar
    .cabal: MOV [R2], R0 ; guarda terminador na memória destino (16 bits)
    .fim:   JMP fin ; acabou
```

Programa 4.8 - Exemplo de acesso misto à memória (em 8 bits e 16 bits): cópia de uma cadeia de caracteres ASCII com conversão para Unicode

Um dos problemas que se colocam é a conversão de textos em codificação ASCII para Unicode. O Programa 4.8 explora este aspecto para ilustrar agora a mistura entre acessos

à memória de 8 e de 16 bits. Neste programa faz-se uma conversão da cadeia de caracteres da Fig. 4.10 (em ASCII, 8 bits por carácter) para uma cadeia de caracteres em Unicode, com 16 bits cada. Esta conversão é muito simples, uma vez que para manter a compatibilidade com a codificação ASCII os primeiros 256 códigos em Unicode (0000H

A Fig. 4.13 representa o conteúdo da memória após execução do Programa 4.8. Os caracteres Unicode terão o seu byte de menor peso igual à codificação ASCII do carácter correspondente e 00H no seu byte de maior peso (mas são valores de 16 bits e não dois

卷之六

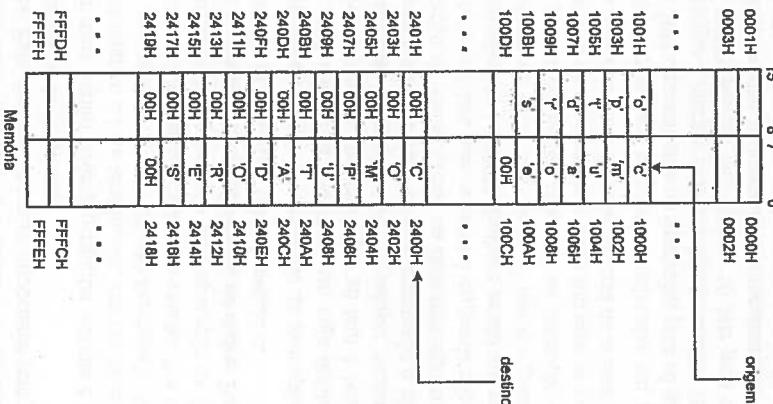


Fig. 4.13 - Situação pretendida após execução do Programa 4.8

Nesta figura representam-se também os endereços ímpares na zona dos caracteres Unicode (todos os bytes têm endereços individuais) mas o programa usa apenas os endereços pares nessa zona, pois o acesso aos caracteres Unicode é em 16 bits.

#### 4.10.5 TRANSFERÊNCIAS PARA MEMÓRIA DE UMA CONSTANTE OU

As transferências de dados com as combinações de operandos memória-constante e memória-memória não são admisíveis porque simplesmente não há espaço nas instruções para especificar duas constantes (endereço-valor ou endereço-endereço). Para conseguir a funcionalidade do MOV nas combinações memória-constante e memória-memória devem usar-se várias instruções e registos auxiliares. Exemplo:

| PRE-PROCESSOR           | DELEGATE                                                          |
|-------------------------|-------------------------------------------------------------------|
| MOV [contador], 1234H   | MOV R0, 1234H<br>MOV R1, contador<br>MOV [R1], R0                 |
| MOV [contador], [TOTAL] | MOV R0, total<br>MOV R1, contador<br>MOV R2, [R0]<br>MOV [R1], R2 |

**Tabela 4.21 - Exemplos de como vencer as limitações da instrução MOV em termos de combinações de operandos.** Poderiam ter sido usados outros registas quaisquer.

#### 4.11 INSTRUÇÕES ARITMÉTICAS

Essas instruções são fundamentais na medida em que têm a responsabilidade de efectuar os cálculos que normalmente qualquer programa tem. A Tabela 4.22 e a Tabela 4.23 apresentam as várias instruções aritméticas disponíveis no PEPE.

- Os acessos via R2 à cadeia de caracteres destinou passaram a usar `MOV` (acesso em 16 bits) em vez de `MOVB` (acesso em 8 bits);
  - R2 passou a ser incrementado de 2 em 2 unidades (acessos em 16 bits);
  - À instrução `MOVB R0, [R1]` lê apenas um byte da memória, colocando o byte de maior peso do R0 a 00H, o que deixa o carácter Unicode logo pronto para ser guardado em memória.

Esta simulação ilustra a mistura entre acessos à memória em 8 bits e em 16 bits, tendo por base o Programma 4.8. Os aspectos cobertos incluem os seguintes:

- Iniciização da Caixa de Valores;
  - Execução passo a passo do programa;
  - Verificação da evolução dos registos relevantes.

MEMÓRIA

Todas as instruções aritméticas afectam todos os bits de estado. A Tabela 4.9, na página 208, exemplifica com a soma como as operações aritméticas podem afectar os bits de estado, usando números de 4 bits por simplicidade, mas de forma facilmente extrapolável para 16 bits.

O PEPE não suporta representação de números reais e operações aritméticas em vírgula flutuante. O Apêndice D, na página 735, faz uma breve introdução a este tema.

#### 4.11.1 INSTRUÇÕES ARITMÉTICAS MAIS SIMPLES

A Tabela 4.22 apresenta as características das instruções aritméticas mais simples.

| ASSEMBLY | EXEMPLO | MÁQUINA     | RTL          | EFEITO           |
|----------|---------|-------------|--------------|------------------|
| ADD      | Rd, Rs  | ADD R0, R1  | ADDR, Rd, Rs | Rd ← Rd + Rs     |
| ADD      | Rd, k   | ADD R0, 1   | ADDI, Rd, k  | Rd ← Rd + k      |
| ADDC     | Rd, Rs  | ADDC R0, R1 | ADDC, Rd, Rs | Rd ← Rd + Rs + C |
| SUB      | Rd, Rs  | SUB R0, R1  | SUBR, Rd, Rs | Rd ← Rd - Rs     |
| SUB      | Rd, k   | SUB R0, 4   | SUBI, Rd, k  | Rd ← Rd - k      |
| SUBB     | Rd, Rs  | SUBB R0, R1 | SUBB, Rd, Rs | Rd ← Rd - Rs - C |
| CMP      | Rd, Rs  | CMP R0, R1  | CMPR, Rd, Rs | ZNCV ← Rd - Rs   |
| CMP      | Rd, k   | CMP R0, 4   | CMPF, Rd, k  | ZNCV ← Rd - k    |
| NEG      | Rd      | NEG R3      | NEG, Rd      | Rd ← -Rd         |

Tabela 4.22 - Instruções aritméticas do PEPE, excepto multiplicação e divisão

Algumas instruções (ADD, SUB, CMP) suportam uma constante de 4 bits, permitindo uma especificação imediata de uma constante para os casos mais frequentes (1, 0, -1, etc.) normalmente com valores baixos. A constante é estendida para 16 bits com sinal. Se a constante pretendida não estiver no intervalo -8 a +7 (4 bits), então em vez de:

ADD R0, k ; só serve se  $-8 \leq k \leq +7$

deve usar-se:

MOV R1, k ; usar um registo auxiliar que esteja livre ADD R0, R1 ; R0 ← R0 + k, qualquer que seja o valor de k em 16 bits

As instruções ADC e SUBB usam o bit de estado C (transporte) como operando adicional. O objectivo fundamental da existência destas instruções é permitir efectuar operações aritméticas com operandos mais largos do que a palavra do processador. Por exemplo, é possível fazer contas com dados de 32 bits no PEPE, que só suporta directamente dados de 16 bits. Como? Fazendo as contas com parte do dados de cada vez. A secção 4.11.2 mostra como se deve fazer para o caso da adição. O mesmo truque pode ser usado para a multiplicação, numa vez que a multiplicação é um conjunto de somas. O Programa 4.21, na página 265, tem outro exemplo de aplicação de ADC (soma condicional de uma unidade a um registo).

A instrução CMP (Compare, comparar) efectua uma subtracção entre os dois operandos, tal como SUB, mas com a diferença que o resultado não destrói o primeiro operando. Apesar os bits de estado são alterados, permitindo efectuar saltos condicionais com base na comparação entre os dois números (Por exemplo, o bit de estado Z ficará activo se os dois operandos forem iguais). A secção 4.9 descreve as instruções de salto.

A instrução NEG calcula o complemento para 2 (simétrico) do operando.

Pode ocasionar uma situação de excesso se o operando for o número mais negativo (8000H no caso do PEPE), pois este número não tem simétrico. Por exemplo, com 4 bits os números podem variar de -8 a +7. NEG aplicado a -8 geraria uma condição de excesso se o processador fosse de 4 bits, porque não é possível representar o número +8 em complemento para 2 com apenas 4 bits. O simétrico de zero é zero.

A instrução SUB R1, R1 (por exemplo) pode ser usada para colocar o valor 0000H num

registo, ao mesmo tempo que coloca o bit de estado a 1 (algo que o MOV R1, 0 não faz,

uma vez que as instruções MOV não afectam os bits de estado).

##### 4.11.1.1 SOMA E EXCESSO: SÉRIE DE FIBONACCI

O Programa 4.9 exemplifica o uso da soma através da série de Fibonacci.<sup>33</sup> Nesta série cujo número de elementos é ilimitado, cada elemento é obtido pela soma dos dois elementos anteriores, em que se define que os dois primeiros elementos são 1. Assim, a série será constituída por números inteiros, começando por

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

O programa gera a sequência, armazena em memória e conta o número de elementos da série até a soma dar excesso. Desta modo, ficamos a saber quantos elementos da série de Fibonacci devem ser calculados.

<sup>33</sup> Matemático italiano do século XIII (1170-1250).

Fibonacci conseguimos representar com 16 bits em complemento para 2 (embora os elementos sejam apenas positivos), o que é algo sempre útil de se saber...

Os dois primeiros elementos são tratados manualmente, mas depois o ciclo vai gerando cada um dos elementos. Note-se o uso da instrução SWAP, que em cada iteração repõe as funções relativas de R1 e R2, permitindo que a "janela" dos dois últimos elementos vá avançando à medida que eles vão sendo calculados.

```
; utilização dos registos:  
; R1 - último valor da série até um dado momento  
; R2 - penúltimo valor da série até um dado momento  
; R3 - endereço de memória onde os elementos (de 16 bits) são escritos  
; R4 - contador de elementos da série  
base EQU 1000H  
  
início: MOV R3, base ; inicializa apontador para o primeiro elemento  
MOV R2, 1 ; inicializa primeiro elemento da série  
MOV [R3], R2 ; armazena primeiro elemento da série  
ADD R3, 2 ; endereço para o segundo elemento (2 bytes depois)  
MOV [R3], R1 ; armazena segundo elemento da série  
MOV R4, 2 ; já foram armazenados 2 elementos  
maisUm: ADD R2, R1 ; soma os dois últimos elementos  
JVE fin ; se houve excesso, a série tem de acabar aqui  
ADD R3, 2 ; endereço para o próximo elemento (2 bytes depois)  
MOV [R3], R2 ; armazena mais este elemento da série  
ADD R4, 1 ; contabiliza mais este elemento  
SWAP R1, R2 ; troca os papéis de R1 e R2, para que R1 volte a ser o último valor da série e R2 o penúltimo  
JMP maisUm ; ainda há mais elementos para gerar  
fin: JMP fin ; acabou. Na memória estão todos os elementos gerados, e R4 indica quantos são
```

#### Programa 4.9 - Programa gerador da série de Fibonacci

##### SIMULAÇÃO 4.9 – SOMA E EXCESSO: SÉRIE DE FIBONACCI

Esta simulação ilustra o funcionamento da adição e da situação de excesso, tendo por base o Programa 4.9. Os aspectos cobertos incluem os seguintes:

- Execução passo a passo e com pontos de paragem do programa;
- Verificação da evolução dos registos relevantes e da memória, iteração a iteração;
- Verificação da condição de excesso.

#### 4.11.1.2 SOMA E TRANSPORTE: NÚMEROS GRANDES

Esta secção mostra como somar números maiores do que a palavra do processador. Neste caso, de 32 bits. Cada operando ocupa dois registos, um com os 16 bits de menor peso e o outro com os 16 bits de maior peso.

O resultado é que contém o sinal. O primeiro até pode ter o bit de maior peso a 1, mas isso não significa que o operando seja negativo. Este bit é apenas o bit 15 de um número cujo bit de maior peso está na posição 31.

Assim supor que dois operandos, A e B, ocupam os registos R1 a R4 da seguinte forma:

- A – R1 e R2 (R1 tem a parte de maior peso);
- B – R3 e R4 (R3 tem a parte de maior peso).

O resultado da soma A+B será armazenado, como na instrução ADD, no primeiro operando, ou seja, em R1 e R2, e pode ser obtido do seguinte modo:

```
ADD R2, R4 ; soma partes de menor peso (pode dar transporte)
ADD R1, R3 ; soma partes de maior peso, usando o transporte anterior
```

Nas estas instruções, R1 e R2 terão a soma pretendida. Ou seja, as partes de menor peso dos dois operandos devem ser encaradas como sem sinal, ligando apenas ao bit C (transporte) e ignorando o bit V (excesso).

Como 32 bits são difíceis de ler (são 8 dígitos hexadecimais), exemplificarmos aqui com dois números de 8 bits a ser somados em duas partes por um suposto processador de 4 bits. Os números são o 58 (3AH) e o 40 (28H). O programa nesse processador de 4 bits seria o seguinte:

```
MOV R1, 3H ; inicializa parte de maior peso do operando A
MOV R2, AH ; inicializa parte de menor peso do operando A
MOV R3, 2H ; inicializa parte de maior peso do operando B
MOV R4, 8H ; inicializa parte de menor peso do operando B
ADD R2, R4 ; soma partes de menor peso (pode dar transporte)
ADDC R1, R3 ; soma partes de maior peso, usando o transporte anterior
```

#### Programa 4.10 - Programa para somar dois números de 8 bits num processador de 4 bits

Note-se que, se interpretássemos R2 e R4 como números de 4 bits e não como partes de dois números de 8 bits, ambos seriam negativos e a sua soma daria excesso (daria um valor positivo, em 4 bits, apesar de os dois operandos da soma serem negativos). No entanto, como R2 e R4 são apenas partes de um número de 8 bits, o excesso é ignorado e (não se o transporte é aproveitado para a soma seguinte (tal como aliás aconteceria se a conta fosse feita manualmente), da forma indicada na Fig. 4.14:



Fig. 4.14 - Soma de um número em duas partes

O resultado, concatenando as duas partes, é 62H, ou 98 em decimal, como seria de esperar. O mesmo raciocínio poderia ser feito com mais bits.

### 4.11.2 MULTIPLICAÇÃO E DIVISÃO

A Tabela 4.23 apresenta as instruções de multiplicação e divisão do PEPE.

A instrução MUL calcula simplesmente o produto, assumindo que os operandos estão em complemento para 2. O resultado tem 16 bits, tal como os operandos, que não podem ser muito elevados sob pena de gerar uma situação de excesso.

O Programa 4.11 exemplifica o uso da multiplicação, implementando o cálculo do factorial de um dado número N. Neste programa, N=6, cujo factorial é 720. Se usar valores maiores de N, o factorial é também maior e pode gerar uma situação de excesso. O programa detecta isso e portanto tem duas formas de terminar, por fim (R1 terá factorial) ou por erro.

| ASSEMBLY   | EXEMPLO    | MACHINE     | RTU                    | EFEITO                                                                                                 |
|------------|------------|-------------|------------------------|--------------------------------------------------------------------------------------------------------|
| MUL Rd, Rs | MUL R0, R1 | MUL, Rd, Rs | Rd ← Rd * Rs           | Rd fica com o produto do valor anterior de Rd com o valor de Rs. O resultado continua a ter 16 bits    |
| DIV Rd, Rs | DIV R0, R1 | DIV, Rd, Rs | Rd ← quo-diente(Rd/Rs) | Rd fica com o quociente da divisão inteira de Rd por Rs. A divisão por zero é um erro (secção 6.2.3.2) |
| MOD Rd, Rs | MOD R0, R1 | MOD, Rd, Rs | Rd ← resto(Rd/Rs)      | Rd fica com o resto da divisão inteira de Rd por Rs. A divisão por zero é um erro (secção 6.2.3.2)     |

Tabela 4.23 - Instruções aritméticas de multiplicação e divisão no PEPE

```

; utilização dos registos:
; R1 - produto dos vários factores (valor do factorial no fim)
; R2 - factor auxiliar que começa com N-1, depois N-2, etc., até ser 2
; (1 já não vale a pena)

N      EQU 5          ; número de que se pretende calcular o factorial

início: MOV R1, N      ; valor inicial do produto
        MOV R2, R1      ; valor auxiliar
maisUm: SUB R2, 1      ; decrementa factor
        MUL R1, R2      ; acumula produto de factores
        JZV R2, 2        ; se houve excesso, o factorial tem de acabar aqui
        CMP R2, 2        ; vê se o factor já diminuiu até 2
        JGT maisUm       ; se ainda é maior do que 2, é preciso continuar
fim:   JMP fim         ; acabou. Em R1 está o valor do factorial
erro:  JMP ..erro       ; termina com erro

```

Programa 4.11 - Cálculo do factorial (de 6, neste exemplo). O programa pode terminar por erro de excesso se N for grande

```

N      EQU 7          ; número a determinar se é primo

início: MOV R2, N      ; divisor (primeiro valor a testar: N-1)
        CMP R2, 0      ; vê se N é 0 ou negativo
maisUm: JLE inválido  ; se for, o numero é considerado não válido
        SUB R2, 1      ; proximo valor do divisor
        CMP R2, 1      ; vê se divisor já chegou a 1
        JLE éPrimo       ; se sim, já testou os divisores todos e o número
                           ; é primo
        MOV R1, "N      ; coloca N num registo para se poder usar no MOD
        MOD R1, R2      ; resto da divisão de N pelo divisor em teste
        JNZ maisUm       ; se for diferente de zero, N não é divisível pelo
                           ; divisor em teste. Vai testar o divisor seguinte
nãoPrimo: JMP nãoPrimo
éPrimo:  JMP éPrimo
inválido: JMP inválido

```

Programa 4.12 - Programa para determinar se um dado número é primo

### SIMULAÇÃO 4.10 – MULTIPLICAÇÃO E EXCESSO: FACTORIAL

Esta simulação ilustra o funcionamento da multiplicação e da situação de excesso, tendo por base o Programa 4.11. Os aspectos cobertos incluem os seguintes:

- Execução passo a passo e com pontos de paragem do programa;
- Verificação da evolução dos registos relevantes, iteração a iteração;
- Determinação experimental (por tentativas) de qual o valor máximo de N sem dar erro de excesso.

As instruções DIV e MOD efectuam a divisão inteira em duas partes. DIV produz apenas o quociente, MOD apenas o resto. Qualquer deles tem 16 bits e é inteiro, o que significa que deve gerar um erro específico ou assistir apenas uma situação de excesso (secção 6.2.3.2, na página 257) 480). Para além do Programa 4.12 e do Programa 4.13, o Programa 4.19 (na página 257) ilustra também as instruções DIV e MOD.

O Programa 4.12 exemplifica o uso do resto da divisão (instrução MOD) para determinar se um número é primo ou não. O programa segue totalmente a definição: um número é primo se os seus únicos divisores forem 1 e ele próprio. Esta é provavelmente a forma mais inefficiente de saber se um número é primo ou não, mas é também provavelmente a mais simples. O programa começa por testar se N é divisível por N-1, depois por N-2, etc., até testar se é divisível por 2. Se chegar ao fim sem descobrir nenhum divisor (isto é, cujo resto da divisão dê zero), o número é primo, caso contrário não é. Como nota de requinte, o programa prevê o caso de N nulo ou negativo (note-se a necessidade do uso da instrução JPF, pois as instruções MOV não afectam os bits de estado).

```

; utilização dos registos:
; R1 - número a determinar se é primo
; R2 - divisor (para testar se N é múltiplo de R2)

N      EQU 7          ; número a determinar se é primo

início: MOV R2, N      ; divisor (primeiro valor a testar: N-1)
        CMP R2, 0      ; vê se N é 0 ou negativo
maisUm: JLE inválido  ; se for, o numero é considerado não válido
        SUB R2, 1      ; proximo valor do divisor
        CMP R2, 1      ; vê se divisor já chegou a 1
        JLE éPrimo       ; se sim, já testou os divisores todos e o número
                           ; é primo
        MOV R1, "N      ; coloca N num registo para se poder usar no MOD
        MOD R1, R2      ; resto da divisão de N pelo divisor em teste
        JNZ maisUm       ; se for diferente de zero, N não é divisível pelo
                           ; divisor em teste. Vai testar o divisor seguinte
nãoPrimo: JMP nãoPrimo
éPrimo:  JMP éPrimo
inválido: JMP inválido

```

**SIMULAÇÃO – DIVISÃO: NÚMEROS PRIMOS**

Esta simulação ilustra o funcionamento da instrução MOD, tendo por base o Programa 4.12. Os aspectos cobertos incluem os seguintes:

- Execução passo a passo e com pontos de paragem do programa;
- Verificação da evolução dos registos relevantes, iteração a iteração;
- Verificação das situações fronteira de N (negativo, 0, 1 e 2).

O Programa 4.13 é mais completo do que o Programa 4.12, pois não apenas determina se um dado número N é primo ou não como ainda, se não for primo, o factoriza em números primos (isto é, determina quais os números primos que multiplicados dão o valor N). Um dado factor pode aparecer mais do que uma vez. Por exemplo,  $60 = 2 \times 2 \times 3 \times 5$ .

Este programa exemplifica o uso das instruções de divisão inteira (MOD, resto, e DIV, quociente). O programa escreve os vários factores em memória, havendo um registo (R4) que diz quantos são. Se for apenas um, o número é primo.

```
; utilização dos registos:
; R1 - número a factorizar
; R2 - divisor (para testar se N é múltiplo de R2)
; R3 - registo auxiliar (porque o MOD destrói o primeiro operando)
; R4 - contador do numero de factores
; R5 - base da zona de memória onde colocar os factores

N      EQU 60          ; número a factorizar
base   EQU 1000H       ; base da zona de memória onde colocar os factores
início: MOV R1, N        ; inicializa registo com número a factorizar
        MOV R4, 0        ; inicializa contador do numero de factores
        CMP R1, 1        ; vê se N é inferior a 1
        JLT invalido    ; se N < 1, sai logo com R4=0 (não factorizável)
        MOV R5, base      ; inicializa registo com base da zona de memória
        ; onde colocar os factores
denovo: MOV R2, 2        ; divisor (primeiro valor a testar: 2)
        MOV R3, R1        ; copia do valor (MOD destrói o 1.º operando)
        MOD R3, R2        ; resto da divisão do número pelo divisor em teste
        ; se não for divisível, vai testar o próximo divisor
        JNZ próxIMO        ; factor encontrado. Tira esse factor do numero
        DIV R1, R2        ; armazena o factor na zona de memória
        MOV [R5], R2        ; contabiliza o factor encontrando
        ADD R4, 1        ; endereço onde armazenar o próximo factor
        ; recomeça procura de factores
        ; obtém o próximo divisor a testar
próximo: CMP R2, R1        ; mas só o vai testar se ...
        ; não superior ao próprio número a factorizar
        JLE maisUm        ; acabou. Se só se encontrou um factor, ...
        ADD R5, 2        ; ... então o numero era primo
        JZ éPrimo         ; aqui há vários factores (R4>1)
        ; se chegar aqui, N é primo (R4=1)
éPrimo: JMP éPrimo       ; se chegar aqui, N não é factorizável (R4=0)
invalido: JMP invalido    ;
```

Programa 4.13 - Factorização de um número

**ESSENCIAL**

O algoritmo é muito simples. Começa por tentar dividir N por um divisor (que começa em 2). Se não for divisível, incrementa o divisor e tenta de novo. Se for divisível, acha um factor, que armazena em memória, e divide o número inicial pelo factor, repetindo depois o algoritmo. O facto de começar pelo divisor mais baixo garante que os factores encontrados são números primos. Tal como no Programa 4.12, o valor de N é testado inicialmente para desistir os valores negativos e 0, que são considerados não factorizáveis (o contador de factores, R4, fica a zero). Um número primo fica apenas com um factor (ele próprio).

**SIMULAÇÃO – DIVISÃO: FACTORIZAÇÃO DE UM NÚMERO**

Esta simulação ilustra o funcionamento das instruções de divisão (DIV e MOD), tendo por base o Programa 4.13. Os aspectos cobertos incluem os seguintes:

- Execução passo a passo e com pontos de paragem do programa;
- Verificação da evolução dos registos relevantes e da memória, iteração a iteração;
- Verificação das situações fronteira de N (negativo, 0, 1 e 2).

| ESSENCIAL                                                                                                                                                                                                                                                |                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| ■ As instruções aritméticas afectam todos os bits de estado Z, N, C, V;                                                                                                                                                                                  | ■ As instruções ADPC e SUBS permitem incluir o transporte de uma operação anterior, superponendo resultados.       |
| ■ A instrução COMP compara dois números e é a única que não destrói o primeiro operando (afecta apenas os bits de estado);                                                                                                                               | ■ A instrução NEG permite calcular o simétrico (complemento para 2) com o sinal do número 8000H (o mais negativo). |
| ■ A soma de dois números de sinal contrário não pode dar excesso (overflow). Nasso o excesso nota-se quando os operandos têm o mesmo sinal, mas diferente sinal do resultado;                                                                            | ■ A multiplicação tem um resultado com apenas 16 bits, tal como os operandos. Gera excesso com facilidade.         |
| ■ A divisão (inteira) gera um quociente e um resto. Para evitar ter dois resultados, há duas instruções para produzir estes valores (DIV e MOD, respectivamente). Ambas produzem resultados de 16 bits. O excesso só pode ocorrer se o divisor for zero. |                                                                                                                    |

## 4.12 INSTRUÇÕES LÓGICAS

### 4.12.1 FUNCIONALIDADE DAS INSTRUÇÕES LÓGICAS

A Tabela 4.24 apresenta as instruções lógicas do PEPE. Este nome é devido às três principais funções lógicas que implementam: conjunção (AND), disjunção (OR e XOR) e negação (NOT).

O XOR (OU-exclusivo) difere do OR (OU) no facto de que exige que os dois bits em cada par de bits (um de cada operando) sejam diferentes, enquanto OR dá 1 também quando os dois bits são 1. Por esse motivo, designa-se OU-exclusivo (um deles tem de ser 1, mas apenas um de cada vez). Representa-se pelo sinal  $\oplus$  e aplicado a um bit cujo segundo operando seja 1 nega esse bit (o resultado só é 1 se os dois operandos forem diferentes; isto é, se o bit for 0, se o bit for 1, os operandos são iguais e o resultado dá 0).

As restantes instruções permitem testar ou manipular bits de forma mais específica. Nas instruções que têm a constante k, esta indica a posição do bit a manipular no registo Rd, pelo que pode variar entre 0 e 15.

| ASSEMBLY    | MÁQUINA      | RTL                                 | BITS DE ESTADO | EFEITO                                                                                                                    |
|-------------|--------------|-------------------------------------|----------------|---------------------------------------------------------------------------------------------------------------------------|
| AND Rd, Rs  | AND, Rd, Rs  | Rd $\leftarrow$ Rd $\wedge$ Rs      | Z, N           | Conjunção (E) bits a bits de Rd e Rs. Rd(1) só será 1 se Rd(1)=1 E Rs(1)=1                                                |
| OR Rd, Rs   | OR, Rd, Rs   | Rd $\leftarrow$ Rd $\vee$ Rs        | Z, N           | Disjunção (OU) bits a bits de Rd e Rs. Rd(1) será 1 se Rd(1)=1 OU Rs(1)=1                                                 |
| XOR Rd, Rs  | XOR, Rd, Rs  | Rd $\leftarrow$ Rd $\oplus$ Rs      | Z, N           | Disjunção exclusiva (OU-exclusivo) bits a bits de Rd e Rs. Rd(1) será 1 apenas se Rd(1) for diferente de Rs(1) (01 ou 10) |
| NOT Rd      | NOT, Rd, 0   | Rd $\leftarrow$ Rd $\oplus$ FFFFH   | Z, N           | Complementa (inverte) todos os bits de Rd. (Negação bits a bits)                                                          |
| TEST Rd, Rs | TEST, Rd, Rs | Rd $\wedge$ Rs                      | Z, N           | Faz a conjunção bits a bits de Rd e Rs, tal como o AND, mas afecta apenas os bits de estado (Rd não é alterado).          |
| BIT Rd, k   | BIT, Rd, k   | Z $\leftarrow$ Rd(k) $\oplus$ 1     | Z              | Coloca no bit Z a negação do bit Rd(k). Rd não é alterado.                                                                |
| SET Rd, k   | SET, Rd, k   | Rd(k) $\leftarrow$ 1                | Z, N           | Põe o bit k de Rd a 1                                                                                                     |
| CLR Rd, k   | CLR, Rd, k   | Rd(k) $\leftarrow$ 0                | Z, N           | Põe o bit k de Rd a 0                                                                                                     |
| CPL Rd, k   | CPL, Rd, k   | Rd(k) $\leftarrow$ Rd(k) $\oplus$ 1 | Z, N           | Complementa (inverte) o bit k de Rd                                                                                       |

Tabela 4.24 - Instruções lógicas do PEPE

**Nota** Se Rd = RE no caso das instruções que alteram o Rd, os únicos bits de estado afectados são os alterados pela operação em si (coluna RTL), não sendo aplicada a regra de alterar os bits Z ou N de acordo com o valor do resultado.

Ao contrário das instruções aritméticas, que consideram que os números binários estão representados em complemento para 2, as instruções lógicas consideram-nos apenas sequências de bits independentes, sem sinal nem sequer valor aritmético (só apenas bits). Note-se, no entanto, que o seu resultado afecta os bits de estado Z e N da mesma forma que numa operação aritmética.

As instruções AND, OR, XOR, NOT e TEST aplicam a respectiva operação a cada bit dos operandos de forma independente dos restantes. As outras instruções lidam apenas com um dos bits, o bit na posição k do registo Rd.

A Fig. 4.15 ilustra o funcionamento das instruções lógicas, usando números binários de 8 bits, apenas para ser mais simples. O funcionamento é o mesmo, qualquer que seja o número de bits dos operandos.

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| 1100 0101 | 1110 0101 | 1110 0101 | 1100 0101 |
| 0001 1101 | 0011 1101 | 0011 1101 | 0011 1101 |
| 0010 0101 | 0010 0101 | 1111 1101 | 1011 1000 |
| Z<0, N<0  | Z<0, N<0  | Z<0, N<1  | Z<0, N<1  |
| (a) AND   | (b) -TEST | (c) -OR   | (d) -XOR  |

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| 1110 0101 | 1111 1111 | 1111 1111 | 1100 0101 |
| 0000 0000 | 0001 1010 | 0001 1010 | 0000 0000 |
| Z<1, N<0  | Z<0, N<0  | Z<0, N<0  | Z<1, N<0  |
| (e) -AND  | (f) -NOT  | (g) -XOR  | (h) -XOR  |

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| 1110 0101 | 1110 0101 | 1110 0101 | 1110 0101 |
| 0         | 4         | 4         | 6         |
| Z<1       | Z<0, N<1  | Z<0, N<0  | Z<0, N<1  |
| (i) -BIT  | (j) -SET  | (l) -CLR  | (m) -CPL  |

Fig. 4.15 - Exemplos de instruções lógicas. Paralelamente ao resultado indica-se também o efeito nos bits de estado Z e N

Note-se que:

- (e) ilustra como o AND com bits desencontrados pode dar zero;
- (g) ilustra como XOR com 1s pode negar completamente o operando, produzindo o mesmo resultado que o NOT. Em (d), o XOR nega um operando apenas nos bits em que o outro operando tem 1s;
- (h) ilustra o efeito do XOR de um registo com ele próprio (XOR R1, R1, por exemplo). O resultado é zero. Este pequeno truque constitui uma forma típica de inicializar registos a zero, com a diferença face ao MOV R1, 0 de que inicializa logo os bits de estado Z e N, o que poderá ser útil em algumas situações.

Um dos objectivos fundamentais das instruções lógicas é implementar expressões booleanas, estabelecendo por exemplo que a expressão  $a \text{ AND } b$  só é verdadeira se simultaneamente a condição  $a$  e a condição  $b$  também o forem. Cada uma destas condições é essencialmente um bit, pois uma condição ou é falsa ou é verdadeira (dois valores possíveis). Normalmente, associa-se 0 a falso e 1 a verdadeiro, embora seja naturalmente possível arbitrar o contrário (é apenas uma questão de convenção de representação).

As instruções de decisão das linguagens de programação de alto nível (tipicamente a instrução `if`) tomam as decisões com base num valor booleano e precisam apenas de um bit para codificar esse valor. Isto é válido para linguagens como Java, que têm o tipo de dados booleano definido, mas linguagens como C têm apenas o tipo "inteiro" e implementam os valores booleanos convencionando que falso é zero e qualquer outro valor é verdadeiro.

O problema nesta convenção é que um simples AND de dois valores verdadeiros (não nulos) pode dar zero, e portanto falso! Basta imaginar os dois valores AAAAH e 5555H e verificar que um valor só tem 1s nas posições pares e outro só nas posições ímpares (a Fig. 4.15(e) apresenta outro exemplo). Conclusão: o AND  $b11 \text{ a } b11$  nestes casos dá 0000H, ou seja, falso, quando devia dar verdadeiro! A secção 4.12.2 mostra como lidar com este problema.

Ao nível da linguagem assembly, as instruções lógicas AND, OR, etc. permitem manipular  $N$  bits de uma só vez, em que  $N$  é a largura do processador. No entanto, os bits são independentes, isto é, um AND de  $N$  bits é na realidade um conjunto de  $N$  ANDs de um bit cada, independentes. O objectivo neste caso não é lidar com  $N$  valores booleanos simultaneamente, mas sim efectuar manipulação de bits.

Assim, estas instruções têm na realidade uma dupla função:

- Implementar expressões booleanas, com valores falso e verdadeiro. Para tal usa-se apenas um bit de cada operando (tipicamente o de menor peso) ou adopta-se a convenção de que um valor 0 representa falso e que um valor diferente de zero representa verdadeiro (como na linguagem C, por exemplo);
- Manipular determinados bits do operando, usando a funcionalidade lógica destas instruções para cada um dos bits individualmente, em dois tipos básicos de operações:
  - Alterar determinados bits de um operando para um valor fixo (0 ou 1), independentemente do valor que esses bits tinham antes;
  - Isolar determinados bits de um operando, mantendo o valor desses bits e forçando os restantes a um valor fixo (tipicamente 0), através de um outro operando designado máscara (precisamente porque força alguns bits e só mantém o valor de outros).

As secções seguintes ilustram estas funcionalidades.

## 4.12.2 EXPRESSESES BOOLEANAS

O Programa 4.14 ilustra a utilização de uma expressão booleana, usando a convenção da linguagem C de que um valor zero de um operando representa falso, e que diferente de zero representa verdadeiro.

O Programa 4.14 ilustra a utilização de uma expressão booleana, usando a convenção da linguagem C de que um valor zero de um operando representa falso, e que diferente de zero representa verdadeiro.

O Programa 4.14 ilustra a utilização de uma expressão booleana, usando a convenção da linguagem C de que um valor zero de um operando representa falso, e que diferente de zero representa verdadeiro.

O Programa 4.14 ilustra a utilização de uma expressão booleana, usando a convenção da linguagem C de que um valor zero de um operando representa falso, e que diferente de zero representa verdadeiro.

O Programa 4.14 ilustra a utilização de uma expressão booleana, usando a convenção da linguagem C de que um valor zero de um operando representa falso, e que diferente de zero representa verdadeiro.

O Programa 4.14 ilustra a utilização de uma expressão booleana, usando a convenção da linguagem C de que um valor zero de um operando representa falso, e que diferente de zero representa verdadeiro.

Programa 4.14 - Exemplo de utilização de uma expressão booleana com OR

O pretendido fosse um par de operandos com o valor verdadeiro e este fosse convertido ser 1 (0001H em 16 bits, por exemplo), bastaria substituir o OR por AND no Programa 4.14.

O entanto, na convenção adoptada na linguagem C o valor verdadeiro não é só um valor muito possível (na realidade, é qualquer valor diferente de zero), pelo que a operação tem de ser outra que não um simples AND. Neste caso, fazem-se testes separados a cada operando e basta um deles ser falso (zero) para o AND dos dois ser falso. Ou seja, em e de se testar AND a, b testa-se antes a negação desta operação, ou OR não-a, não-b, de acordo com as leis de Morgan (secção 2.2, na página 31).

O Programa 4.15 ilustra esta solução. Note-se o uso da instrução TEST com o próprio sítio, apenas para actualizar os bits de estado (ao contrário do AND, a instrução TEST não actualiza o 1.º operando), uma vez que o MOV não o faz. Há várias maneiras de fazer isto, sólido (no caso de R1):

```

AND    R1, R1
OR     R1, R1
CMP    R1, 0
mas não:
XOR    R1, R1      ; ATENÇÃO: isto coloca o registo R1 todo a zero!!!
;
```

pois XOR de um bit com ele próprio dá zero (Fig. 4.15b).

; utilização dos registos:  
; R1 - 1.º operando a testar  
; R2 - 2.º operando na memória do 1.º operando (o 2.º está logo a seguir)  
; R3 - endereço na memória do 1.º operando (o 2.º está logo a seguir)

```

início: MOV   R3, 0000H      ; inicializa R3 no inicio da memória
maisUm: MOV   R1, [R3]        ; obtém 1.º operando
TEST  R1, R1                ; actualiza os bits de estado
JZ    próximo             ; se é falso (zero), pode já passar ao próximo
MOV   R2, [R3 + 2]          ; obtém 2.º operando
TEST  R2, R2                ; actualiza os bits de estado
JNZ   achou               ; se chegou aqui e R2 é verdadeiro, então R1 e R2
                           ; são verdadeiros e achou o par pretendido
próximo: ADD  R3, 2           ; vai de continuar. Avança base da memória
JMP   maisUm              ; achou! Para aqui
achou: JMF  achou            ; achou! Para aqui
;
```

Programa 4.15 - Exemplo de utilização de uma expressão booleana com AND que não pode ser implementada com uma simples instrução ADD

### SIMPLIFICAÇÃO – EXPRESSESES BOOLEANAS

Esta simulação ilustra o funcionamento das expressões booleanas, tendo por base Programa 4.14 e o Programa 4.15. Os aspectos cobertos incluem os seguintes:

- Execução passo a passo e com pontos de paragem do programa;
- Verificação do conteúdo da memória e da existência de duas células de memória seguidas com o valor zero (se não existirem, têm de se criar alterando diretamente o conteúdo de uma ou duas células de memória);
- Verificação da evolução dos registos relevantes e da memória, iteração a iteração.

### 4.12.3 INSTRUÇÕES DE MANIPULAÇÃO DE UM SÓ BIT

Há situações em que se pretende alterar apenas um bit para conseguir um dado objectivo. Pode-se querer, por exemplo, trocar o seu valor (de 0 para 1, independentemente do seu valor anterior, complementá-lo, trocando o seu valor (de 0 para 1 ou vice-versa). Ou pode-se querer testar o bit antes, para saber se o seu valor deve ser alterado ou não. O PEPE suporta todas estas operações com as instruções SET (força a 1), CLR (força a 0), CPL (trocá o bit) e BIT (testa o valor do bit).

O Programa 4.16 mostra como usar estas instruções, usando a conversão entre letras maiúsculas e minúsculas. Este tópico já foi usado no Programa 4.7, na página 233, com

uma só palavra e convertendo à custa de uma soma (ou subtração) do factor de 20H. Por exemplo, o carácter 'A' na codificação ASCII tem o valor 61H, enquanto o carácter 'A' tem o valor 41H. Esta diferença de 20H mantém-se em todas as letras do alfabeto.



Dado que o RE (Registo de Estado) pode ser especificado nas instruções como qualquer outro registo, estas instruções de manipulação de um só bit podem ser usadas para manipular individualmente os bits deste registo. Bits como Z, N e V não se manipulam normalmente de forma direta, mas o bit C é muitas vezes usado em conjunção com as instruções de deslocamento (seção 4.13) e pode ser útil mutar ou saber o seu valor.

Existem ainda outros bits no RE que devem ser inicializados pelo programador se este pretender utilizar interrupções (seção 6.2.2, na página 462).

Uma alternativa a este esquema é comparar os bits das letras maiúsculas e minúsculas. A Tabela 4.25 mostra as codificações de algumas das letras do alfabeto, as suficientes para se perceber que as codificações das letras maiúsculas e minúsculas diferem apenas no bit 5. Este bit é precisamente aquele que define o factor 20H (0010 0000).

| BINÁRIO   | HEXADECIMAL | LETRA | LETRA | HEXADECIMAL | BINÁRIO   |
|-----------|-------------|-------|-------|-------------|-----------|
| 0100 0001 | 41H         | A'    | 'a'   | 61H         | 0110 0001 |
| 0100 0010 | 42H         | B'    | 'b'   | 62H         | 0110 0010 |
| ...       | ...         | ...   | ...   | ...         | ...       |
| 0100 1110 | 4EH         | W'    | 'w'   | 6EH         | 0110 1110 |
| 0100 1111 | 4FH         | Y'    | 'y'   | 6FH         | 0110 1111 |
| 0101 0000 | 50H         | 'P'   | 'p'   | 70H         | 0111 0000 |
| 0101 0001 | 51H         | Q'    | 'q'   | 71H         | 0111 0001 |
| ...       | ...         | ...   | ...   | ...         | ...       |
| 0101 1001 | 59H         | 'Y'   | 'y'   | 79H         | 0111 1001 |
| 0101 1010 | 5AH         | Z'    | 'z'   | 7AH         | 0111 1010 |

Tabela 4.25 - Codificação das letras em ASCII. As maiúsculas diferem das minúsculas apenas no bit 5

A diferença face ao Programa 4.7 é que não é preciso saber se a letra em questão é maiúscula ou minúscula, para decidir se é preciso somar (ou subtrair) o valor 20H. Se na Fig. 4.12, na página 232, algum dos caracteres na cadeia origem não fosse minúscula (por exemplo, "ComPUAdoRes" em vez de "computadores", como sucede na Fig. 4.16), o carácter correspondente na cadeia Destino não seria uma letra.

Por exemplo, se o carácter 'p' (70H – ver Tabela 4.25), que seria convertido em 'P' (50H), já fosse maiúsculo, como na Fig. 4.16, seria convertido erradamente em 30H (50H - 20H), que corresponde em ASCII ao carácter '0' (zero). Para suportar esta mistura entre maiúsculas e minúsculas, o Programa 4.7 tinha de testar o carácter para ver se era mesmo uma letra minúscula, antes de o converter.

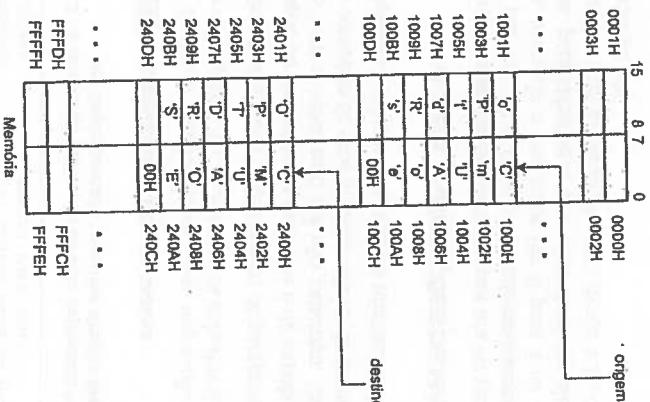


Fig. 4.16 - Conversão de letras maiúsculas e minúsculas para maiúsculas

Colocando 0 ou 1 no bit 5, obtém-se uma letra maiúscula ou minúscula, respectivamente, independentemente da letra que era antes, tornando o teste desnecessário. O Programa 4.16 faz precisamente isto, por adaptação do Programa 4.7, usando a instrução CLR para colocar o bit 5 a 0 em vez de subtrair 20H à codificação ASCII do carácter.

Note-se que, por simplicidade, o Programa 4.16 continua a assumir que a cadeia de caracteres contém apenas letras.

A Tabela 4.26 mostra qual o efeito de mudar a instrução CLR no Programa 4.16 por SET e CPL, que colocam o bit a 1 ou trocam o seu valor, respectivamente.

### SIMULAÇÃO – INSTRUÇÕES SET, CLR E CPL

Esta simulação ilustra o funcionamento das instruções de manipulação de um só bit SET, CLR e CPL, tendo por base o programa e tabelas anteriores. Os aspectos cobertos incluem os seguintes:

- Execução passo a passo e com pontos de paragem do programa;
- Inicialização do conteúdo da memória (cadeia de caracteres);
- Verificação da evolução dos registos relevantes e da memória, iteração a iteração.

| ORIGEM     | EQU | 1000H          | ; endereço do primeiro carácter da cadeia original     |
|------------|-----|----------------|--------------------------------------------------------|
| DESTINO    | EQU | 2400H          | ; endereço do primeiro carácter da cadeia destino      |
| TERMINADOR | EQU | 00H            | ; valor do byte que indica o fim da cadeia original    |
| INÍCIO:    | MOV | R1, origem     | ; apontador para o primeiro carácter da cadeia destino |
|            | MOV | R2, destino    | ; apontador para o primeiro carácter destino           |
|            | MOV | R0, [R1]       | ; obtém o próximo carácter (acesso de 8 bits)          |
| MAISUM:    | MOV | R0, terminador | ; se já é 0 byte terminador                            |
|            | CMP | R0, terminador | ; se sim, vai acabar                                   |
|            | JZ  | R0, 5          | acaba                                                  |
|            | CLR | R0, 5          | converte letra para maiúscula (põe bit 5 a zero)       |
|            | MOV | [R2], R0       | guarda carácter na zona de memória destino             |
|            | ADD | R1, 1          | endereço (em bytes) do próximo carácter origem         |
|            | ADD | R2, 1          | ainda há mais caracteres para tratar                   |
|            | JMP | maisUm         | guarda terminador na zona de memória destino           |
|            | MOV | [R2], R0       | acabou                                                 |
|            | JMP | fin            |                                                        |

Programa 4.16 - Cópia de uma cadeia de caracteres com conversão para maiúsculas  
(mesmo que as letras originais já o sejam)

| CADEIA DE CARACTÉRES ORIGEM | CADEIA DE CARACTÉRES DESTINO |             |             |
|-----------------------------|------------------------------|-------------|-------------|
|                             | (PARA MAIÚSCULAS)            | SET R0, 5 ; | CPL R0, 5 ; |
| 'C'                         | 'C'                          | 'C'         | 'C'         |
| 'o'                         | 'O'                          | 'O'         | 'O'         |
| 'm'                         | 'M'                          | 'M'         | 'M'         |
| 'u'                         | 'U'                          | 'U'         | 'U'         |
| 'a'                         | 'T'                          | 'T'         | 'T'         |
| 'r'                         | 'R'                          | 'R'         | 'R'         |
| 'e'                         | 'E'                          | 'E'         | 'E'         |
| 's'                         | 'S'                          | 'S'         | 'S'         |
| 00H                         | 00H                          | 00H         | 00H         |

Tabela 4.26 - Utilização das instruções de manipulação de um só bit para alterar a codificação das letras em ASCII

○ Programa 4.17 conta o número de letras maiúsculas na cadeia de caracteres origem na Fig. 4.16, testando o bit 5 de cada carácter com a instrução BIT.

|            |     |                |                                                     |
|------------|-----|----------------|-----------------------------------------------------|
| origen     | EQU | 1000H          | ; endereço do primeiro carácter da cadeia original  |
| terminador | EQU | 00H            | ; valor do byte que indica o fim da cadeia original |
| início:    | MOV | R1, origem     | ; apontador para o primeiro carácter origem         |
|            | MOV | R2, 0          | ; inicializa contador de letras maiúsculas          |
| maiúsm:    | MOV | R0, [R1]       | ; vai buscar o próximo carácter (acesso de 8 bits)  |
|            | CMP | R0, terminador | ; se sim, acabou                                    |
|            | JZ  | fin            | ; vê qual o valor do bit 5 do carácter              |
|            | BIT | R0, 5          | ; ao bit não for 0 (não for maiúscula) passa        |
|            | JNZ | próximo        | ; ao próximo                                        |
|            | ADD | R2, 1          | ; contabiliza mais uma letra maiúscula              |
| próximo:   | ADD | R1, 1          | ; endereço (em bytes) do próximo carácter origem    |
|            | JMP | maiúsm         | ; ainda há mais caracteres para tratar              |
| fim:       | JMP | fin            | ; acabou. Em R2 está o número de letras maiúsculas  |

Programa 4.17 - Contagem do número de letras maiúsculas com a instrução BIT

### SIMULAÇÃO 4.15 – INSTRUÇÃO BIT

Esta simulação ilustra o funcionamento da instrução BIT, tendo por base o Programa 4.17.

- Execução passo a passo e com pontos de paragem do programa;
- Inicialização do conteúdo da memória (cadeia de caracteres);
- Verificação da evolução dos registos relevantes e da memória, iteração a iteração.

## 4.12.4 OPERAÇÕES LÓGICAS COM MÁSCARAS

### 4.12.4.1 FUNCIONAMENTO DAS MÁSCARAS

A palavra "máscara" é aqui usada no sentido normal do termo, designando algo que impede uma operação em certos pontos e a permite noutras. No sentido corrente, operação é "ver", mas aqui a operação é uma das operações lógicas com dois operandos de 16 bits definidas na Tabela 4.24 (AND, OR, XOR e TEST).

Considera-se que o 1.º operando (aquele em que o resultado é armazenado) é o verdadeiro operando e que o 2.º operando é a máscara, que indica sobre que bits do 1.º operando operação deve ser feita.

A máscara é formada por:

- Bits activos, que deixam passar os bits do 1.º operando para o resultado sem alterações;
- Bits neutros, que transformam os bits do 1.º operando de acordo com a operação lógica aplicada.

A Tabela 4.27 estabelece as regras. AND e TEST usam a mesma regra. A diferença é que TEST usa o resultado apenas para actualizar os bits Z e N mas não o armazena no registo do 1.º operando.

| OPERAÇÃO  | BIT NEUTRO (DEIXA PASSAR IGUAL) | BIT ACTIVO (ALTERA BIT) | EFEITO DO BIT ACTIVO    |
|-----------|---------------------------------|-------------------------|-------------------------|
| AND, TEST | 1                               | 0                       | Força a 0               |
| OR        | 0                               | 1                       | Força a 1               |
| XOR       | 0                               | 1                       | Nega o bit (roca valor) |

Tabela 4.27 - Princípio de funcionamento das máscaras, em cada um dos seus bits neutros e activos, para cada uma das operações lógicas

As máscaras utilizam-se quando se pretende actuar "cirurgicamente"<sup>14</sup> sobre certos bits de um registo sem alterar nem sequer saber o valor dos restantes bits desse registo.

O Programa 3.7, na página 176, já contém um exemplo de utilização de máscaras (apesar de o processador ser apenas de 8 bits) aplicado à leitura de um valor de um periférico (botão de um semáforo), para isolar os bits que se querem testar dos restantes lidos (pois estes podem não ser 0 e interferir depois em eventuais operações de comparação do valor lido com valores predefinidos).



Fig. 4.17 - Efeito de uma máscara (0FH) com as várias operações lógicas

Fig. 4.17 ilustra o efeito de uma máscara (0000 1111, para separar melhor os bits neutros dos activos, mas podia ser uma qualquer) com cada uma destas operações lógicas.

### 4.12.4.2 MÁSCARAS AND

Estas máscaras deixam passar os bits correspondentes aos bits a 1 na máscara e forçam a 0 os restantes. Usam-se normalmente para isolar campos de bits num registo (colocando a 0 os bits que não interessam) ou colocar a 0 determinados bits (os que interessam).

O Programa 4.18 ilustra o uso de uma máscara AND para contar todos os bytes numa dada zona de memória cujos 4 bits de maior peso são 0111 (isto é, os bytes entre 70H e 7FH).

<sup>14</sup>Em medicina, nas operações cirúrgicas, também se usa uma máscara (um pano com um buraco) para melhor delimitar a zona de actuação.

```

primeiro EQU 1000H ; endereço do primeiro byte a testar
último EQU 1FFFH ; endereço do último byte a testar
padrão EQU 70H ; valor do byte que serve de termo de comparação
máscara EQU 0F0H ; máscara para isolar os 4 bits de maior peso

início: MOV R1, primeiro ; inicializa apontador para o primeiro byte
        MOV R2, [R1] ; inicializa registo com máscara
        MOV R3, máscara ; inicializa contedor de bytes
maisUm: MOV R0, [R1] ; vai buscar o próximo byte (acesso de 8 bits)
        AND R0, R3 ; elimina os 4 bits de menor peso (isola os 4 de maior peso)
        CMP R0, R4 ; compara com o padrão
        JNZ próximo ; se não for igual, passa ao próximo byte
próximo: ADD R5, 1 ; se for igual, contabiliza este byte
        ADD R1, R2 ; já estamos no último byte?
        JZ fim ; se este já era o último byte, termina
        ADD R1, 1 ; obtém endereço (em bytes) do próximo byte
        JMP maisUm ; vai testar mais um byte
fim:    JMP fim ; acabou. Em R5 está o número de bytes contados

```

Programa 4.18 - Exemplo de aplicação de uma máscara AND

Note-se que:

- Se a instrução AND não existisse, este programa contaria apenas os bytes com o valor 70H. O AND força os 4 bits de menor peso a 0, pelo que todos os bytes com valor entre 70H e 7FH são transformados pelo AND em 70H (valor do termo padão de comparação), e portanto todos eles são contabilizados em R5;
- A máscara é na realidade de 16 bits, pois a instrução AND usa registos de 16 bits, apesar de ter definido apenas 8 bits da máscara na directiva EQU. A directiva considera que o operando é de 16 bits, 00F0H, e não FFF0H, pois não faz extensão com sinal (isso é exclusivo das instruções que só admitem constantes de menos de 16 bits, como o MOV, ADD, CMP, etc.). A directiva EQU é uma forma de definir constantes sempre de 16 bits (secção 5.5.1, na página 285), pelo que assume que F0H<sup>35</sup> apenas deixou de especificar os zeros à esquerda. Este aspecto é importante para o funcionamento do CMP R0, R4.

### SIMULACRUM — MÁSCARA AND

Esta simulação ilustra o funcionamento das instruções de manipulação de um só bit, tendo por base os programas e tabelas anteriores. Os aspectos cobertos incluem os seguintes:

- Execução passo a passo e com pontos de paragem do programa;
  - Inicialização do conteúdo da memória (cadeia de caracteres);
  - Verificação da evolução dos registos relevantes e da memória, iteração a iteração.
- #### 4.12.4.3 MÁSCARAS OR
- Estas máscaras deixam passar os bits correspondentes aos bits a 0 na máscara e forcaram os restantes. Usam-se normalmente para colocar a 1 determinados bits dum registo.
- O Programa 4.19 gera uma cadeia de caracteres (é preenche uma zona de memória com elas), com os dígitos de um determinado número em decimal, usando o facto de a codificação ASCII dos dígitos 0 a 9 ser 30H a 39H, respectivamente. Portanto, nos 4 bits de menor peso está um número entre 0 e 9 e nos 4 bits de maior peso está 3.

```

início: MOV R1, N ; número a converter para caracteres
        MOV R2, base ; exemplo de número a converter para caracteres
        MOV R3, maxdiv ; endereço em memória onde colocar o 1.º carácter
        MOV R4, 0 ; máscara para converter dígito em carácter ASCII
        MOV R5, 10 ; usado a seguir para dividir o divisor por 10
maisUm: MOV R0, R1 ; inicializa apontador para converter divisor de 10 para números em 16 bits
        DIV R0, R3 ; obtém divisor potência de 10 para números em 16 bits
        OR R5, R0 ; inicializa indicador de zero à esquerda
        MOV R6, 10 ; usado a seguir para dividir o divisor por 10
        DIV R6, R3 ; obtém divisor potência de 10 para números em 16 bits
        OR R5, R6 ; inicializa indicador de zero à esquerda
        MOV R7, R0 ; obtém dígito de maior peso (em base 10)
        AND R7, R4 ; armazena carácter ASCII
        ADD R2, 1 ; actualiza endereço (em bytes) do próximo carácter
        DIV R1, R3 ; divide dígito de maior peso ao número a converter
        OR R5, R1 ; ainda há mais caracteres para tratar
        MOV R8, R0 ; converte último dígito (menor peso) para carácter ASCII
        MOV R9, R1 ; armazena último carácter
        MOV R10, R0 ; esta instrução é dispensável pois neste altura R3 já é 0, mas sem ela fica mais obscuro e é uma potencial fonte de erros em caso de alteração do programa!
        MOV R11, R3 ; escreve 0 (terminador) no fim da cadeia de caracteres
        JMP fim ; acabou

```

Programa 4.19 - Geração de uma cadeia de caracteres com os dígitos de um número, usando máscaras OR

<sup>35</sup> O zero antes do F evita que a constante seja confundida com um identificador (ver Nota na página 286), não tendo impacte no valor da constante.

O programa obtém cada dígito (0 a 9) pelo resto da divisão por uma potência de 10 (cujo valor depende da posição do dígito) e depois gera a codificação ASCII forçando o 3 com

uma máscara OR. Neste caso em particular o mesmo efeito podia ser obtido somando 3000 ao digito, mas a solução de máscara é mais geral em termos de modificar *bitwise operations*.

SIMPLY EASY 4.17 - MÁSCARA OR

Esta simulação ilustra o funcionamento das máscaras OR, tendo por base o Programa 4.19.



varmente o numero por 10 e ir guardando os restos das divisões, mas os algarismos apareceriam pela ordem inversa do pretendido;

Para que não apareçam zeros à esquerda<sup>36</sup>, usa-se um registo (R5) que começará zero mas deixá de o ser mal apareça um dígito diferente de zero, gracias a um

O último dígito tem de ser tratado aparte nessa altura  $R3$  (o divisor) é zero e não se pode usar o algoritmo normal, que faz uma divisão por  $R3$ .  
 JÁ

A escrita do byte a zero no fim da cadeia de caracteres pode usar o facto de que R3 nessa altura tem o valor 0. Mas em nome da clareza e robustez descreveremos

mas, é recomendável que não se usem truques (tal como indicado no comentário) pois é extremamente fácil esquecer este truque ao alterar o mapeamento e introduzir novas rotas.

eros sem se dar por isso, que depois saem bastante caros em termos de tempo gasto na depuração (testes para descobrir a origem dos erros).

A Tabela 4.28 mostra o conteúdo dos registos e das células de memória em cada iteração.

aparece quatro vezes, uma por cada iteração). A verificação dos conteúdos é deixada para o leitor e pode ser feita usando a Simulação 4.17. Os conteúdos das células não inicializadas devem ser zeros.

| ETIQUETA   | R0    | R1    | R2    | R3    | R4    | R5    | R6  | R7  | R8  | R9  | R10 | R11 | R12 | R13 | R14 |
|------------|-------|-------|-------|-------|-------|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| maisJnt(1) | 5746  | 5745  | 1000H | 10000 | 0030H | 0000H | xx  |
| maisJnt(2) | 0000H | 5745  | 1000H | 1000  | 0030H | 0000H | xx  |
| maisJnt(3) | 0035H | 746   | 1001H | 100   | 0030H | 0035H | 35H | xx  |
| maisJnt(4) | 0037H | 46    | 1002H | 10    | 0030H | 0037H | 35H | 37H | xx  |
| ultimo     | 0034H | 0036H | 1003H | 0     | 0030H | 0037H | 35H | 37H | 34H | xx  | xx  | xx  | xx  | xx  | xx  |
| fim        | 0034H | 0036H | 1005H | 0     | 0030H | 0037H | 35H | 37H | 34H | 36H | 00H |     |     |     |     |

**Tabela 4.28 - Valores dos registos e das células de memória ao longo da execução do Programa 4.19**

Neste exemplo, com N=5746, apareceria um zero à esquerda, pois com 16 bits os números positivos podem ir até 32767 (em decimal), ou seja, até 5 dígitos.

Fig. 4.18 - Cifra de caracteres (negação de alguns bits) com uma máscara XOR

Cifra mono-alfabética simples, utilizada desde o império romano,

| LETRAS<br>ORIGINAL | CODIFICAÇÃO<br>ASCII | CODIFICAÇÃO APOSSXOR<br>COM 3CH (0011 1100) | CARACTERE ASCII<br>RESULTANTE |
|--------------------|----------------------|---------------------------------------------|-------------------------------|
| ç'                 | 0110 0011            | 0101 1111                                   | —                             |
| ó'                 | 0110 1111            | 0101 0011                                   | 5                             |
| m'                 | 0110 1101            | 0101 0001                                   | Q                             |
| p'                 | 0111 0000            | 0101 1100                                   | L                             |
| ú'                 | 0111 0101            | 0100 1001                                   | H                             |
| p'                 | 0111 0100            | 0101 1101                                   | J                             |
| á'                 | 0110 0001            | 0101 1000                                   | X                             |
| ó'                 | 0110 0100            | 0101 0011                                   | S                             |
| ç'                 | 0110 1111            | 0101 1101                                   | N                             |
| é'                 | 0110 0010            | 0100 1110                                   | ó                             |
| ç'                 | 0111 0011            | 0100 1111                                   | ó                             |

Tabela 4.29 - Utilização da máscara XOR (3CH) para conseguir uma cifra muito simples dos caracteres ("computadores" transforma-se em "SQLIñHjsNvO")

| origem                                                                    | EQU                                                          | 1000H | ; endereço do primeiro carácter da cadeia original  |
|---------------------------------------------------------------------------|--------------------------------------------------------------|-------|-----------------------------------------------------|
| destino                                                                   | EQU                                                          | 2400H | ; endereço do primeiro carácter da cadeia destino   |
| terminador                                                                | EQU                                                          | 00H   | ; valor do byte que indica o fim da cadeia original |
| máscara                                                                   | EQU                                                          | 3CH   | ; máscara (0011 1100) para negar os bits do meio    |
| <b>início:</b> MOV R1, origem ; apontador para o primeiro carácter origem |                                                              |       |                                                     |
|                                                                           | MOV R2, destino ; apontador para o primeiro carácter destino |       |                                                     |
|                                                                           | MOV R3, máscara ; inicializa registo com máscara             |       |                                                     |
| maisUm:                                                                   | MOVB R0, [R1] ; obtém o próximo carácter (acesso de 8 bits)  |       |                                                     |
|                                                                           | CMP R0, terminador ; vê se já é o byte terminador            |       |                                                     |
|                                                                           | JZ acaba ; se sim, vai acabar                                |       |                                                     |
|                                                                           | XOR R0, R3 ; nega bits do meio do byte de menor peso         |       |                                                     |
|                                                                           | MOV B [R2], R0 ; guarda carácter na zona de memória destino  |       |                                                     |
|                                                                           | ADD R1, 1 ; endereço do próximo carácter origem              |       |                                                     |
|                                                                           | ADD R2, 1 ; endereço (em bytes) do próximo carácter destino  |       |                                                     |
|                                                                           | JMP maisUm ; ainda há mais caracteres para tratar            |       |                                                     |
| acaba:                                                                    | MOVB [R2], R0 ; guarda terminador na zona de memória destino |       |                                                     |
|                                                                           | JMP fin: ; acabou                                            |       |                                                     |

Programa 4.20 - Cifra dos caracteres de uma cadeia usando uma máscara XOR

Note-se que, dado que os bits são alterados, nem todos os caracteres resultantes são letras mas continuam a ter uma representação ASCII.

### SIMULAÇÃO – MASCARA XOR

- Esta simulação ilustra o funcionamento das máscaras XOR, tendo por base o Programa 4.20. Os aspectos cobertos incluem os seguintes:
- Execução passo a passo e com pontos de paragem do programa;
  - Verificação do funcionamento da instrução XOR;
  - Verificação da evolução dos registos relevantes e da memória, iteração a iteração.

### ESSENCIAL

- As instruções lógicas manipulam os números binários como simples sequências de bits independentes (não há noção de complemento para 2) e afectam apenas os bits Z e N (não podem gerar nem transportar nem excesso);
- As instruções AND, OR, XOR, NOT e TEST lidam com todos os bits do operando SET, CLR, CPL e BTR lidam com apenas um dos bits;
- Estas instruções permitem suportar as expressões booleanas das linguagens de alto nível, mas nem sempre uma simples instrução lógica consegue implementar uma operação booleana básica;
- As instruções AND, OR e XOR permitem manipular valores de 16 bits com uma máscara, que especifica os bits a que operação deve ser aplicada;
- As instruções de manipulação de um só bit permitem de uma forma expedita testar ou alterar um dado bit de um registo.

## 4.13 INSTRUÇÕES DE DESLOCAMENTO

Estas instruções permitem deslocar o operando (um registo, obrigatoriamente) de vários bits para a esquerda ou para a direita, o que tem uma série de aplicações, em particular quando se conjugam com as instruções lógicas, como por exemplo:

- Enviar os bits do operando em sequência, um a um, através de uma ligação de um só bit (usado em redes informáticas);
- Analisar os bits de um operando, um a um (contar o número de bits a 1, por exemplo);
- Máscaras que por deslocamento vão variando ao longo do programa, permitindo ir analisando diferentes partes de um dado operando;
- Isolar um grupo de bits do operando (por meio de uma máscara com AND, por exemplo) e mudá-los de posição (para ocuparem os bits de menor peso, por exemplo).

A Tabela 4.30 apresenta as instruções de deslocamento do PEPE. Trata-se de instruções que tipicamente todos os processadores suportam e que podem dividir-se em dois grupos:

- Deslocamentos circulares, ou rotações;
- Deslocamentos lineares, ou simples;
- Em qualquer deles o deslocamento pode ser para a:

  - Esquerda (na direcção do bit de maior peso), ou
  - Direita (na direcção do bit de menor peso).

| ASSESSORAMENTO | FILTRO                                                                                               | BITS DE ESTADO | EFEITO                                                              |
|----------------|------------------------------------------------------------------------------------------------------|----------------|---------------------------------------------------------------------|
| SHR Rd, k      | $k > 0 : C \leftarrow Rd(k-1)$<br>$k > 0 : Rd \leftarrow 0(k) \parallel Rd(15..k)$                   | Z, N, C        | Desloca k bits para a direita (entram zeros)                        |
| SHL Rd, k      | $k > 0 : C \leftarrow Rd(15..k+1)$<br>$k > 0 : Rd \leftarrow Rd(15..k, 0) \parallel 0(k)$            | Z, N, C        | Desloca k bits para a esquerda (entram zeros)                       |
| SHRA Rd, k     | $k > 0 : C \leftarrow Rd(k-1)$<br>$k > 0 : Rd \leftarrow Rd(15..k) \parallel Rd(15..k)$              | Z, N, C        | Desloca k bits para a direita, mantendo o sinal                     |
| SHLA Rd, k     | $k > 0 : Rd \leftarrow Rd(15..k, 0) \parallel 0(k)$                                                  | Z, N, C<br>V   | Desloca k bits para a esquerda (entram zeros, mas pode dar excesso) |
| ROR Rd, k      | $k > 0 : C \leftarrow Rd(k-1)$<br>$k > 0 : Rd \leftarrow Rd(k-1..0) \parallel Rd(15..k)$             | Z, N, C        | Roda k bits para a direita                                          |
| ROL Rd, k      | $k > 0 : C \leftarrow Rd(15..k+1)$<br>$k > 0 : Rd \leftarrow Rd(15..k, 0) \parallel Rd(15..15..k+1)$ | Z, N, C        | Roda k bits para a esquerda                                         |
| RORC Rd, k     | $k > 0 : (Rd \parallel C) \leftarrow Rd(k-2..0) \parallel C \parallel Rd(15..15..k-1)$               | Z, N, C        | Roda k bits para a direita, incluindo o bit C                       |
| ROLCK Rd, k    | $k > 0 : (C \parallel Rd) \leftarrow Rd(15..k+1..0) \parallel C \parallel Rd(15..15..k+2..0)$        | Z, N, C        | Roda k bits para a esquerda, incluindo o bit C                      |

Tabela 4.30 - Instruções de deslocamento do PEPE

Rd é o registo cujos bits são deslocados e k uma constante de 4 bits que especifica quantos bits Rd devem ser deslocados. Esta constante pode variar entre 0 e 15. Se k=0, não há lugar a deslocamento e apenas os bits de estado Z e N são actualizados (de acordo com o valor de Rd, que neste caso não é alterado), enquanto C se mantém.

Em todas as instruções de deslocamento (com  $k > 0$ ), o bit de estado C fica com o último valor que saiu do registo. Com a ajuda das instruções de salto condicional JC e JNC, isto permite tomar logo uma decisão com base neste bit que saiu do operando, sem necessidade de uma instrução de teste específica (BIT, por exemplo).

#### 4.13.1 INSTRUÇÕES DE DESLOCAMENTO LINEAR

No deslocamento linear, para a esquerda ou para a direita, cada bit é deslocado para a posição do bit ao lado, tal como indicado na Fig. 4.19 (para simplificar apenas 8 bits são representados, mas o princípio é válido para qualquer número de bits).

Um bit sai, sendo transferido para o bit de estado C. Um bit entra, podendo ser 0 ou 1, de acordo com o tipo de instrução de deslocamento, que pode ser de dois tipos:

- Lógico (SHL, SHR) – Encaram o operando como uma simples sequência de bits entrando sempre um 0 pelo lado oposto ao do bit que saiu;

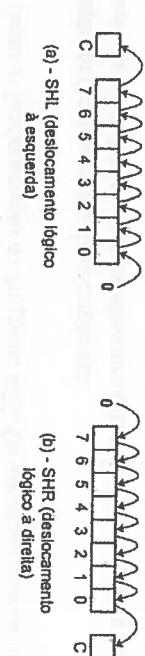
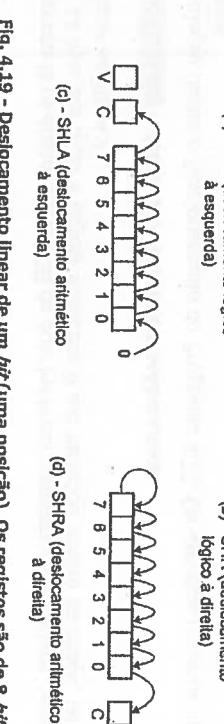


Fig. 4.19 - Deslocamento linear de um bit (uma posição). Os registos são de 8 bits, mas o princípio é o mesmo para qualquer número de bits. Um deslocamento de N bits repete o deslocamento de um bit N vezes ( $1 \leq N \leq 15$ )



Aritmético (SHRA, SHLA) – Encaram o operando como um número inteiro representado em complemento para 2, de tal modo que um deslocamento de um bit para a esquerda ou para a direita equivale a multiplicar ou dividir, respectivamente, o operando por 2, mantendo o sinal no processo. No deslocamento para a direita (SHRA), o bit que entra do lado esquerdo é igual ao bit que já lá estava (mantém o sinal). O deslocamento para a esquerda (SHLA) pode dar origem a um excesso se o bit que sai (para o bit C) for diferente do bit que ficar na posição de maior peso do registo após o deslocamento (isto é, um número positivo passar a negativo ou vice-versa), sendo esta a única diferença face à instrução SHL.

Todos os bits mudam ao mesmo tempo, pelo que nenhum destroi outro. Para deslocar um registo N bits para a esquerda (ou para a direita), repete-se o deslocamento de um bit N vezes. N pode variar entre 0 e 15, mas só com  $N \geq 1$  há lugar a deslocamento efectivo. Se  $N=0$ , actualiza apenas os bits de estado.

No caso da instrução SHLA, a situação de excesso com  $N \geq 1$  ocorre se algum dos bits que sair for diferente do bit que ficar na posição de maior peso após o deslocamento.

A Fig. 4.20 ilustra estes deslocamentos em várias situações. Note-se que:

- A Fig. 4.20e produz uma situação de excesso, apesar de o bit de maior peso que fica no registo ser 1, tal como antes do deslocamento. O excesso ocorre porque dos bits que saíram nem todos eram 1 (e portanto não foi apenas uma questão de eliminar 1s à esquerda);
- Nas Fig. 4.20d e Fig. 4.20f, o bit que entra é o bit de maior peso, ou o bit do sinal. Neste exemplo o número é negativo e o bit é 1, mas se o número for não negativo (nulo ou positivo) o bit que entra será 0.

O Programa 4.21 ilustra o funcionamento dos deslocamentos lineares lógicos, contando o número de bits a 1 no valor contido num registo. O princípio de funcionamento é simples:

- O registo é deslocado um bit de cada vez. O bit que sai vai para o bit C;

- Se o bit é 1, soma-o ao valor do contador de bits;
- Repete os passos anteriores até o registo ser zero.

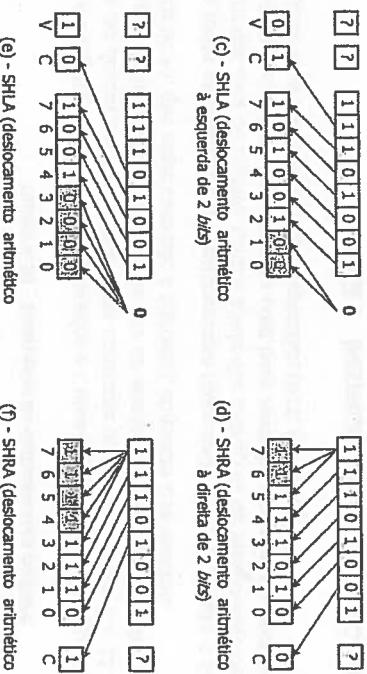
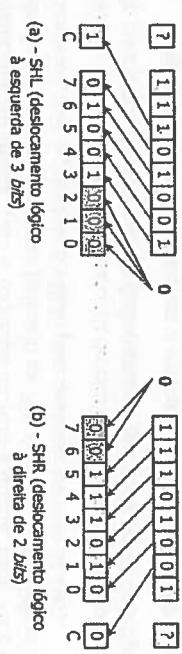


Fig. 4.20 - Deslocamentos lineares, lógicos e aritméticos, à esquerda e à direita. Os registos são de 8 bits, mas o princípio é o mesmo para qualquer número de bits

**SHLA** A instrução SHLA (ou SHRA) constitui uma forma mais fácil e eficiente de multiplicar (ou dividir) um número por uma potência de 2 do que usar a instrução MUL (ou DIV). Deslocar  $N$  bits corresponde a multiplicar (ou dividir) um número por  $2^N$ . Note-se no entanto que se  $2^N$  for maior (em valor absoluto) que o próprio número a dividir (SHRA) dá -1 (tudo a 1s) quando o número for negativo e zero (tudo a 0s) em caso contrário. Cuidado com o potencial excesso no caso de multiplicação (SHLA).

As instruções SHL e SHR não são adequadas para multiplicação e divisão porque SHL não detecta a situação de excesso e SHR não preserva o sinal (entra sempre 0). Estas instruções são mais adequadas para manipulação de bits, especialmente em cooperação com as instruções de máscaras lógicas (secção 4.12.4).

Pode contrastar-se este programa com o Programa 4.2, na página 186, que usa uma instrução lógica com uma máscara que vai evoluindo ao longo das iterações.

Note-se que:

- A instrução de deslocamento usada é SHR (para a direita), mas neste caso era igual usar a instrução SHL (para a esquerda);

|         |     |           |                                                        |
|---------|-----|-----------|--------------------------------------------------------|
| valor   | EQU | 6AC5H     | ; valor cujos bits a 1 vão ser contados                |
| início: | MOV | R1, valor | ; inicializa registo com o valor a analisar            |
|         | MOV | R2, 0     | ; inicializa contador de número de bits a 1            |
|         | MOV | R3, 0     | ; valor 0 auxiliar                                     |
| maisUm: | CMP | R1, 0     | ; isto é só para actualizar os bits de estado          |
|         | JZ  | final     | ; se o valor já é zero, não há mais bits a 1           |
|         | SHR | R1, 1     | ; retorna o bit de menor peso do valor e coloca-o      |
|         | ADD | R2, R3    | ; soma mais 1 ao contador, mas só se o bit             |
|         | JMP | maisUm    | ; vai analisar o próximo bit                           |
|         | JMP | final     | ; acabou. R2 tem o número de bits a 1 no valor inicial |

Programa 4.21 - Contagem do número de bits a 1 num registo com uma instrução de deslocamento

#### SIMULAÇÃO 4.20 - DESLOCAMENTOS LINEARES LÓGICOS

Esta simulação permite analisar o funcionamento do Programa 4.21. Os aspectos cobertos incluem os seguintes:

- Verificação do funcionamento da instrução SHR;
- Verificação da evolução dos registos relevantes, iteração a iteração.

O Programa 4.22 ilustra o funcionamento dos deslocamentos lineares 16 bits em conjunção com máscaras. O objectivo é compactar dois caracteres em ASCII que representam dígitos de 0 a 9 (codificações 30H a 39H) num único byte, em codificação BCD (Binary Coded Decimal, ou decimal codificado em binário), em que cada dígito ocupa só os 4 bits imprescindíveis, pelo que podem ser colocados aos pares em cada byte. Por exemplo, os dígitos 8 e 2 representam-se por 38H e 32H em ASCII e por 82H em BCD.

Este programa lê os dígitos em ASCII de dois bytes em memória consecutivos, compacta-os num só byte em BCD e depois armazena-o em memória noutra endereço, tal como representado na Fig. 4.21.

#### SIMULAÇÃO 4.20 - DESLOCAMENTOS E MÁSCARAS

Esta simulação ilustra o funcionamento do Programa 4.22. Os aspectos cobertos incluem os seguintes:

- Verificação do funcionamento da instrução SHL;
- Verificação da evolução dos registos relevantes e da memória, iteração a iteração.

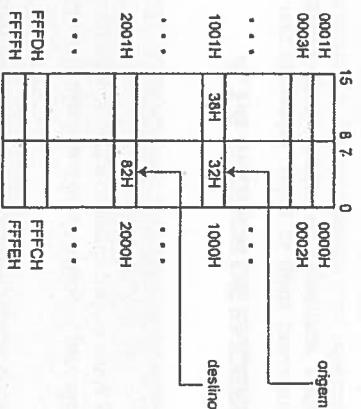


Fig. 4.21 - Compactação de dois dígitos em BCD num só byte a partir de dois dígitos em ASCII (38H e 32H, ou '8' e '2')

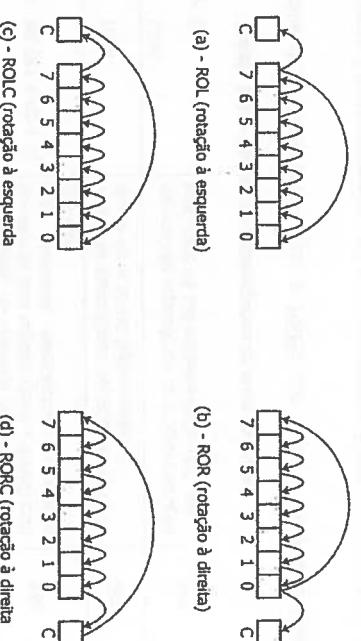


Fig. 4.22 - Rotação de um bit (uma posição). Os registos são de 8 bits, mas o princípio é o mesmo para qualquer número de bits. Uma rotação de N bits repete a rotação de um bit N vezes ( $1 \leq N \leq 15$ )

A rotação tem duas variantes face ao comportamento do bit de estado C:

\* Sem transporte – O bit C é actualizado com o bit que sai, mas não interfere na rotação;

\* Com transporte – O bit C participa ele próprio na rotação. Os bits do registo passam pelo bit C antes de voltarem a entrar no registo. É como se o registo rodasse mais um bit.

A Fig. 4.23 ilustra estas rotações em várias situações.

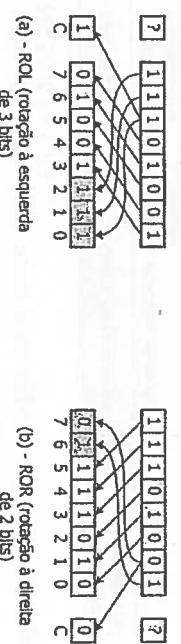


Fig. 4.23 - Rotações, com e sem transporte, à esquerda e à direita. Os registos são de 8 bits, mas o princípio é o mesmo para qualquer número de bits

Programa 4.22 - Compactação de dois dígitos em BCD num só byte a partir de dois dígitos em ASCII, usando deslocamentos e máscaras

```

origem EQU 1000H ; endereço do primeiro dígito em ASCII
destino EQU 2000H ; endereço onde colocar o byte com dois dígitos em BCD
máscara EQU 0FH ; máscara (0000 1111) para retirar o dígito propriamente dito dos caracteres em codificação ASCII

início:
    MOV R1, origem ; apontador para o primeiro dígito em ASCII
    MOV R2, destino ; apontador para o byte com dois dígitos em BCD
    MOV R3, máscara ; inicializa registo com máscara
    MOV R0, [R1] ; vai buscar o 1.º dígito em ASCII (acesso de 8 bits)
    AND R0, R3 ; isola o dígito propriamente dito, usando a máscara
    MOV R4, R0 ; guarda-o para já noutra registo
    ADD R1, 1 ; endereço do 2.º dígito em ASCII
    MOVE R0, [R1] ; obtém o 2.º dígito em ASCII (acesso de 8 bits)
    AND R0, R3 ; isola o dígito propriamente dito
    SHL R0, 4 ; desloca o dígito 4 bits para a esquerda, para poder...
    OR R4, R0 ; ... juntar os dois dígitos sem colidirem
    MOVE R0, [R2], R4 ; guarda o resultado na célula de memória destino
    fim: JMP fim ; acabou
  
```

#### 4.13.2 INSTRUÇÕES DE DESLOCAMENTO CIRCULAR (ROTAÇÕES)

No deslocamento circular, ou rotação, para a esquerda ou para a direita, cada bit é deslocado para a posição ao lado, tal como no deslocamento linear. A diferença está em que o bit que sai não se perde, pois "dá a volta" e entra de novo pelo lado oposto do registo, tal como indicado na Fig. 4.22 (com registos de 8 bits para ser mais simples, mas será idêntico com qualquer número de bits).

Uma das utilizações das instruções de rotação é mudar a posição relativa dos bits de um registo, sem perder nenhum bit. Por exemplo, para trocar as posições relativas dos bytes do registo R1 (de 16 bits) basta fazer:

```
MOV R1, 1234H ; inicializa R1 com um valor
ROR R1, 8 ; roda R1 de 8 posições (troca os bytes), ficando com 3412H
```

Neste caso (8 bits de rotação), rodar à esquerda (ROL) daria o mesmo resultado:

A rotação com transporte permite, por exemplo, inserir um dado bit (o valor inicial do bit C, que terá de ser inicializado) no meio de um registo. Nada que não se possa fazer com um deslocamento linear adequado e manipulação directa do bit pretendido. A sua existência justifica-se mais com motivos históricos, quando os processadores não tinham instruções para manipulação directa de bits.

| ESSENCIAL                                                                                                                                                                                                                                                                |  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| ■ As instruções de deslocamento são fundamentais porque permitem mudar bits de posição num registo, o que não é possível fazer com outras instruções;                                                                                                                    |  |
| ■ Em conjugação com as operações lógicas e máscaras, as instruções de deslocamento permitem facilmente compor valores a partir de outros, inclusivamente juntando ou separando valores de e para registo diferentes;                                                     |  |
| ■ Muitas vezes, o que se desloca nem é o valor a processar, mas sim a máscara usada para tratar alguns dos seus bits;                                                                                                                                                    |  |
| ■ Os deslocamentos podem ser lineares (os bits que saem perdem-se) ou circulares (os bits que saem voltam a entrar);                                                                                                                                                     |  |
| ■ Qualquer dos deslocamentos pode realizar-se para a direita ou para a esquerda;                                                                                                                                                                                         |  |
| ■ Normalmente, os valores são encarados como simples sequências de bits independentes. No entanto, os deslocamentos lineares aritméticos encaram-nos como números em complemento para 2, para constituir formas eficientes de multiplicar ou dividir por potências de 2; |  |
| ■ As instruções de deslocamento afectam o bit de estado C, para além dos deslocamentos lineares aritméticos, para a esquerda podem ainda gerar excesso;                                                                                                                  |  |

#### 4.14 MODOS DE ENDEREÇAMENTO

A maior parte das instruções tem operandos, sobre os quais executam operações. Por modos de endereçamento entende-se as formas de especificar onde está um dado operando. Embora haja processadores que possuem modos mais rebuscados, adaptados à sua própria arquitectura (como, por exemplo, acessos à memória através de registo auto-

-incrementados em cada acesso), existe um conjunto de modos mais usado e que geralmente a maior parte dos processadores suporta.

Note-se que o modo de endereçamento se pode aplicar a cada um dos operandos. No entanto, as instruções envolvem sempre um registo (aquele em que o resultado é armazenado), pelo que quando se fala do modo de endereçamento de uma instrução se está na realidade a falar do modo de endereçamento do outro operando.

A Tabela 4.31 especifica quais os modos mais usados e dá exemplos. Note-se que as constantes podem ser numéricas ou simbólicas (assumindo-se neste caso que os respetivos valores foram já definidos por uma directiva EQU).

O modo implícito refere instruções (RET, PUSH) e um registo (SP), que serão apenas explicados nas secções 5.7.2.3 e 5.7.3.1.

| MODO             | EXEMPLOS                              | QUE ESTÁ O OPERANDO?                                                                                                                  | PEPE |
|------------------|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|------|
| Imediato         | MOV R0, 10<br>ADD R2, mediaDuzia      | Constante imediata, especificada na própria instrução                                                                                 | Sim  |
| Registo          | MOV R0, R1<br>CMP R2, R7              | Num registo                                                                                                                           | Sim  |
| Directo          | MOV R0, [1000H]<br>MOV [contador], R3 | Na memória, no endereço especificado como constante na própria instrução                                                              | Não  |
| Indirecto        | MOV R0, [R1]<br>MOV R3, [R2]          | Na memória, no endereço contido num registo                                                                                           | Sim  |
| Baseado          | MOV R0, [R1 + 3]                      | Na memória, no endereço obtido pela soma do valor de um registo (base) com uma constante especificada na instrução                    | Sim  |
| Indexado         | MOV R0, [R1 + R2]                     | Na memória, no endereço obtido pela soma do valor de um registo com o valor-de-outro registo (um deles actua como índice variável)    | Sim  |
| Baseado Indexado | MOV R0, [R1 + R2 + 6]                 | Na memória, no endereço obtido pela soma do valor de dois registo (base e índice) com uma constante especificada na própria instrução | Não  |
| Relativo         | JMP 100H                              | Constante especificada na própria instrução, que se soma ao PC (saltos relativos)                                                     | Sim  |
| Implicito        | RET<br>PUSH R3                        | Registo não especificado na própria instrução mas usado na sua implementação (PC, SP)                                                 | Sim  |

Tabela 4.31 - Modos de endereçamento mais comuns

Sobre os dois modos de endereçamento que o PEPE não suporta (directo e baseado indexado):

O modo directo é particularmente útil para especificar directamente endereços de memória ou de periféricos. A sua vantagem é ser simples e claro. O problema é

que exige uma constante de 16 bits (endereço) especificada directamente na instrução, o que não é possível com um tamanho fixo de 16 bits para as instruções, tal como especificado na Tabela 4.6. Nestes casos, há basicamente duas soluções possíveis (sumarizadas na Tabela 4.32):

- Usar endereçamento indirecto, em que se carrega primeiro o endereço da célula de memória a aceder num registo e depois se acede através dele. Se o programa fizer vários acessos à mesma célula e houver registos disponíveis, pode mesmo reservar-se um registo para conter esse endereço enquanto for necessário;
- Reunir as várias células de memória (ou periféricos) em endereços contíguos e usar endereçamento baseado, em que se carrega um registo com o endereço de base dessa zona de células (ou periféricos) a aceder e depois se fazem os acessos especificando esse registo e uma constante que indica a posição de cada célula (ou periférico) dentro dessa zona.

O modo baseado indexado pode ser útil para aceder a estruturas de dados (listas, vectores, tabelas, etc.) em que cada elemento seja não uma simples célula mas uma estrutura com vários campos (tal como o exemplo da Fig. 4.9). O PEPE não pode suportar este modo pois implica demasiada informação para ser codificada numa instrução de 16 bits. De qualquer modo, a variedade de estruturas de dados é grande, nem sempre adaptada a este modo, e é sempre possível usar os modos baseado ou indexado, em que se efectua o cálculo de parte do endereço antes de fazer o acesso (em vez de ter as três componentes – base, índice variável e índice constante – especificadas explicitamente). Neste caso, o modo indexado tem ainda a vantagem de suportar qualquer valor para a constante, ao contrário do baseado que no PEPE só suporta constantes pares entre -16 e +14.

A Tabela 4.32 ilustra estas possíveis soluções, que são idênticas para células de memória e para periféricos.

| PRETENDE-SE                  | NO PEPE PODE USAR-SE                                                                                                 |
|------------------------------|----------------------------------------------------------------------------------------------------------------------|
| ; EXEMPLO IMPOSSÍVEL NO PEPE | ; solução 1: endereçamento indirecto                                                                                 |
|                              | contador EQU 1000H<br>soma EQU 1002H<br>total EQU 1004H                                                              |
|                              | ; endereçamento directo<br>(não suportado)                                                                           |
|                              | MOV R9, contador<br>MOV R0, [R9]<br>MOV R3, total<br>MOV R1, [R9]                                                    |
|                              | ADD R0, R1<br>MOV R9, soma<br>MOV [R9], R0                                                                           |
|                              | ; solução 2: endereçamento baseado                                                                                   |
|                              | base EQU 1000H<br>contador EQU 0<br>soma EQU 2<br>total EQU 4                                                        |
|                              | ; solução 3: endereçamento baseado indexado                                                                          |
|                              | MOV R9, [R9 + contador]<br>MOV R1, [R9 + total]<br>ADD R0, R1<br>MOV [R9 + soma], R0                                 |
|                              | ; solução 1: endereçamento baseado<br>MOV R9, R1<br>ADD R9, R2<br>MOV R0, [R9 + 4]; só pares (-16...+14)             |
|                              | ; solução 2: endereçamento indexado<br>MOV R9, 4<br>ADD R9, R2<br>MOV R0, [R1 + R9]                                  |
|                              | ; solução 3: endereçamento indexado com<br>; constante <-16 ou >+14<br>; (não suportado)<br>MOV R0, [R1 + R2 + 1000] |

Tabela 4.32 - Possíveis soluções para os modos de endereçamento não suportados pelo PEPE

## 4.15 CONCLUSÕES

O PEPE é um processador muito simples quando comparado com os processadores actuais de alto desempenho. No entanto, inclui praticamente todos os mecanismos básicos para conseguir fazer programas genéricos, nomeadamente no que diz respeito ao conjunto de instruções, que é o foco deste capítulo.

- Há algumas restrições impostas pela codificação das instruções. Por exemplo, o modo imediato em algumas instruções só admite valores entre -8 e +7, e o modo directo não é suportado. Há sempre formas menos optimizadas de conseguir o objectivo pretendido.
- Para além dos modos de endereçamento mais comuns, alguns processadores suportam outros mais específicos, como por exemplo acessos à memória com auto-incremento dos registos que contêm os endereços a aceder.

Trata-se de um microprocessador de 16 bits, o que implica algumas limitações, nomeadamente em termos de capacidade de cálculo (40.000, por exemplo, já é um número demasiado grande, dada a escala de valores possíveis de -32.768 a +32.767), ausência de suporte para números não inteiros (o que realisticamente necessita de pelo menos 32 bits – ver Apêndice D, na página 735) e algumas restrições impostas às instruções pelo facto de terem de ser codificadas em 16 bits (número de registos limitado e constantes imediatas reduzidas).

Estas limitações são conscientes e motivadas sobretudo pelo objectivo de exercitar a programação manual em linguagem assembly. Se este processador se destinasse a ser programado em C ou a um nível de abstracção mais elevado teria com certeza 32 bits. Existe um compilador de C para o PEPE, que teve de fazer uma série de restrições à linguagem para poder ser mais facilmente implementado (é sempre possível conseguir a funcionalidade que o hardware não suporta por meio de software). Lidar manualmente com constantes de 32 bits é complicado, pelo que a um nível introdutório é preferível suportar as limitações dos 16 bits em nome da simplicidade.

O mesmo espírito presidiu à escolha de uma arquitectura especificamente desenhada para fins pedagógicos face às comercialmente existentes, cujo conjunto de instruções está optimizado para tamanho de código ou desempenho e não se destina a programação manual mas sim a código gerado por compiladores.

Por outro lado, o PEPE inclui características que exercitem as várias técnicas existentes, não descurando no entanto os detalhes necessários para poder ser realmente implementado. O simulador é uma forma crucial de suprir a falta de um microprocessador real, facto que motivou o esforço de vários anos que já foi investido. A aprendizagem requer interacção efectiva com a máquina, mesmo que simulada, pois só com programas a funcionar se consegue provar que efectivamente se domina as várias técnicas envolvidas.

Os próximos capítulos continuarão à descoberta das capacidades do PEPE.

## 4.16 EXERCÍCIOS

- 4.1 Explique porque é que o PEPE tem:**
  - a) Uma largura de 16 bits;
  - b) 16 bits nos barramentos, tanto no de dados como no de endereços;
  - c) Endereçamento de byte.
- 4.2 Indique em hexadecimal e decimal qual o primeiro e último endereço do espaço de endereçamento do PEPE.**
- 4.3 Suponha que as primeiras quatro instruções executadas pelo PEPE após o arranque (reset) foram ADD, MOV, MOV e JNE. Indique o endereço em que cada uma destas instruções se localiza.**

**4.4** Se um conjunto de 1000 palavras (contíguas numa memória ligada ao PEPE) começam no endereço 1000H, qual é o último endereço (em hexadecIMAL) que ainda pertence a este conjunto de dados?

**4.5** Dê um exemplo (indique uma instrução e valor dos registos relevantes) de um acesso à memória:

- a) Alinhado;
- b) Desalinhado;
- c) Em byte.

**4.6** Suponha que um computador baseado no PEPE tem 8 KBytes de ROM, 16 KBytes de RAM, 512 bytes de periféricos de saída e 128 bytes de periféricos de entrada.

**4.7** Indique qual o efeito prático de usar o PEPE com uma RAM de largura de:

- a) 8 bits;
- b) 32 bits.

**4.8** Usando a Tabela A.9, que termina na página 707, converte para linguagem assembly do PEPE o seguinte programa em código-máquina: C102H, C203H, 5012H, 6618H, 6925H, C300H, D310H, 9133H, B132H, 200AH. Não use constantes simbólicas de dados, mas use etiquetas para as constantes de endereços.

**4.9** Explique porque é que a instrução CALL 1000H pode ser válida num programa e invalida noutras (pista: consulte a Tabela A.9).

**4.10** Com base na Tabela A.9, justifique porque é que o PEPE só pode ter 16 registos.

**4.11** Converta o Programa 3.6, na página 175, para linguagem assembly do PEPE (mantenha os valores definidos com EQU). Se o simulasse no PEPE com um relógio de 1 Hz, os tempos das diversas cores dos semáforos seriam substancialmente maiores. Porque?

**4.12** Usando a instrução ADD R1, R2, indique um possível par de valores destes dois registos que activem cada um dos bits de estado Z, N, C e V (mesmo que active outros bits de estado), em quatro exemplos independentes. Distinga a semântica dos bits C e V (em que situações ocorrem?).

**4.13** Indique qual o valor dos bits de estado Z, C, N e V após a execução das seguintes instruções (valores antes da execução indicados entre parênteses):

- a) SUB R1, R1;
- b) ADC R1, R2 (C=1, R1=A5A5H e R2=5A5AH);
- c) ADD R1, R2 (R1=8AF3H, R2=C31FFH);

**4.14** Explique qual é o efeito da instrução máquina 2FFFFH (pista: consulte a informação detalhada sobre as instruções do PEPE, na Tabela A.9).

**4.15** A tabela seguinte mostra a evolução dos valores de três registos e de uma posição de memória ao longo da execução de um conjunto de instruções. Acabe de preencher a tabela. Em cada linha, coloque uma instrução (em sintaxe assembly) cuja execução possa ter originado os valores dos registos e da memória indicados nessa linha (cada instrução altera o estado deixado pela anterior).

| INSTRUÇÃO EXECUTADA | R1    | R2    | R3    | M1000H |
|---------------------|-------|-------|-------|--------|
| Estado Inicial      | FFFFH | FFFFH | FFFFH | FFH    |
|                     | FFFFH | FFFFH | FFFFH | FFH    |
|                     | 1000H | FFFFH | FFFFH | FFH    |
|                     | 1000H | 0010H | FFFFH | FFH    |
|                     | 1000H | 0010H | 00AFH | FFH    |
|                     | 0010H | 1000H | 00AFH | FFH    |
|                     | 0010H | 1000H | 00EH  | AFH    |

**4.16** Faça um programa que, dada uma sequência contígua em memória de números de 8 bits (interpretados sem sinal, de 0 a 255), determine o maior deles. Simule esse programa, usando o circuito de base das simulações deste capítulo, e verifique seu funcionamento.

**4.17** *Idem*, com um programa que escreva em memória, a partir de um dado endereço, todos os pares de letras maiúscula/minúscula.

**4.18** *Idem*, com um programa que, dado um carácter em ASCII (Apêndice E, na página 743), devolva num registo um dos caracteres "L", "D", "S" ou "C", consoante esse carácter seja uma letra (maiúscula ou minúscula), um dígiro, um sinal de pontuação ou um carácter de controlo (com valor menor do que 20H), respectivamente.

**4.19** Modifique o Programa 4.18 para usar uma máscara OR em vez de AND, mas mantendo a funcionalidade do programa. Simule-o para verificar a sua correcção.

**4.20** Algém teve a ideia brilhante de simplificar o Programa 4.21 para a versão seguinte, que afinal tem alguns problemas. Explique quais.

```

valor EQU 6AC5H ; valor cujos bits a 1 vão ser contados
início: MOV R1, valor ; inicializa registo com o valor a analisar
maisUm: SHR R1, 1 ; retira o bit de menor peso do valor para o bit C
      ADDC R2, 0 ; soma mais um ao contador, se esse bit para 1
      JNZ maisUm ; se valor=0, não há mais bits a 1 para contar
      fin:   JMP fin ; acabou. R2 tem o nº de bits a 1 no valor inicial
  
```

Os seres humanos adquirem conhecimento através da aprendizagem. Conseguem aplicar o conhecimento que têm em novas situações, recolher informação sobre os resultados obtidos de forma a reforçar (ou eliminar) conceitos e associações, actualizando o seu conhecimento, e continuar este ciclo de aprendizagem.

Pelo contrário, e com exceção de alguns pequenos protótipos experimentais e de alguns programas específicos, a engenharia maior da computadores tem de ser programada. Para saberem o que fazer precisam de um programa que preveja todas as situações que poderão ocorrer. Um computador não tem inteligência própria, e não sabe o que fazer quando surge uma situação que o seu programa não antecipou.

O capítulo 4 mostrou as características fundamentais de um processador (o cérebro de um computador) e as operações mais básicas que ele consegue efectuar. Não é nada parecido com o modelo de funcionamento humano. Isso significa que programar um computador não é uma tarefa nada fácil para um ser humano (o programador).

Após conceber a ideia do que pretende, o programador tem de a transformar num conjunto de pequenas operações que o computador consiga processar, quer executando-as directamente, quer efectuando transformações de modo a produzir sequências de operações mais elementares mas que no seu conjunto são funcionalmente equivalentes.

O objectivo deste capítulo é estabelecer os passos necessários para que, dado um computador baseado num processador, como por exemplo o PEPE, se possam programar e executar programas arbitrariamente complexos.

Hoje em dia já existem linguagens de programação de alto nível, que evitam muitos dos detalhes da programação em assembly. Este capítulo mostra como mapear uma linguagem de alto nível em linguagem assembly, de modo a que se perceba quais os passos envolvidos na programação de um computador, desde a ideia até ao programa pronto e a executar.

No entanto, a ênfase de um livro sobre arquitetura de computadores é a linguagem assembly, pois tem uma relação muito mais directa com os recursos do hardware e com o seu funcionamento. Assim, este capítulo mostra como se deve programar numa linguagem, apresentando uma série de exemplos que não só complementam os do capítulo 4 como introduzem novas funcionalidades decorrentes das potencialidades do assembler.

Se o capítulo 4 teve as instruções como ponto central, este capítulo coloca a sua ênfase no programa como sequência de instruções, que implementa uma dada funcionalidade, e na imprescindível boa documentação que faz parte intrínseca da arte de bem programar, seja em alto nível ou em linguagem assembly.

## 5 - PROGRAMAÇÃO DE UM COMPUTADOR

## 5.1 UM PROBLEMA SIMPLES

O capítulo 4 apresentou uma série de pequenos exemplos, destinados simplesmente a ilustrar as instruções básicas do PEPE. Os computadores conseguem executar programas muito mais elaborados, feitos exclusivamente por equipas de centenas de programadores. No entanto, o PEPE tem ainda muitas características por revelar, pelo que o melhor é começarmos com um exemplo simples.

Imaginemos que a um ser humano e a um computador é dado um conjunto de números inteiros entre 0 e 9999, para ordenar por ordem crescente.

### 5.1.1 MODO DE ACTUAÇÃO DE UM SER HUMANO

Embora possa naturalmente haver variantes, o algoritmo tipicamente usado pelas pessoas para ordenar estes números é o seguinte:

1. Separa os vários números em dez grupos, de acordo com o seu algarismo dos milhares;
2. Em cada grupo dos milhares, separa os números em dez grupos, de acordo com o algarismo das centenas;
3. Em cada grupo das centenas, separa os números em dez grupos, de acordo com o algarismo das dezenas;
4. Em cada grupo das dezenas, ordena os números por comparação directa;
5. No final, é só coligir todos os grupos das dezenas pela ordem crescente dentro das centenas e dentro dos milhares, ficando tudo ordenado.

### 5.1.2 MODO DE ACTUAÇÃO DE UM COMPUTADOR

Um computador pode ser programado para executar exactamente este algoritmo. Mas só para imitar o ser humano, pois esta não é a forma mais natural de actuar por parte de um computador.

Um computador não funciona em base 10, mas sim em base 2. Também não precisa de organizar os números em grupos (um truque usado pelos humanos para conseguir lidar mais facilmente com muitos números), além de que consegue executar muitas operações repetitivas de forma muito rápida e sem se aborrecer ou cansar.

Existem algoritmos de ordenação bastante optimizados para o funcionamento de um computador, mas aqui vamos ilustrar um bastante simples, normalmente designado ordenação de bolha (*bubble sort*).

Este algoritmo consiste simplesmente em:

1. Ir analisando os vários números da sequência dada, dois a dois, e trocar a sua posição na sequência se o primeiro for maior que o segundo (se forem iguais não vale a pena trocá-los);

## 5.2 MODELAÇÃO DO PROBLEMA COM FLUXOGRAMAS

A Fig. 5.1 podia ser usada para simplesmente explicar o algoritmo a um ser humano, que compreenderia por aprendizagem. Um computador é muito mais básico e não consegue entender a figura directamente, pelo que se torna necessário desenvolver um processo de transformar o conceito do algoritmo ao nível humano até chegar ao código-máquina, constituído por instruções que o processador consegue executar directamente, tal como representado na Fig. 1.4, na página 13.

O conceito é normalmente expresso primeiro por uma simples descrição textual, gráfica ou tabular (a chamada especificação). Depois, com alguma prática e em casos muito

**Fig. 5.1 - Ilustração do algoritmo de ordenação de bolha para quatro números.**  
Os rectângulos indicam os pares de números testados. Aquelas em que houve troca de números face à iteração anterior estão a cinzento

| Sequência original |                 | 10 | 5  | 6  | 2  |
|--------------------|-----------------|----|----|----|----|
| Ronda 1            | Após iteração 1 | 5  | 10 | 6  | 2  |
|                    | Após iteração 2 | 5  | 6  | 10 | 2  |
|                    | Após iteração 3 | 5  | 6  | 2  | 10 |
| Ronda 2            | Após iteração 1 | 5  | 6  | 2  | 10 |
|                    | Após iteração 2 | 5  | 6  | 10 | 2  |
|                    | Após iteração 3 | 5  | 2  | 6  | 10 |
| Ronda 3            | Após iteração 1 | 2  | 5  | 15 | 6  |
|                    | Após iteração 2 | 2  | 5  | 6  | 10 |
|                    | Após iteração 3 | 2  | 5  | 6  | 10 |
| Ronda 4            | Após iteração 1 | 2  | 5  | 6  | 10 |
|                    | Após iteração 2 | 2  | 5  | 6  | 10 |
|                    | Após iteração 3 | 2  | 5  | 6  | 10 |

simples, o programador pode passar directamente da especificação para o programa numa linguagem de alto nível.

No entanto, na maior parte dos casos o programador precisa de um auxiliar, um mapa que o guie no processo de programação, algo que expresse o problema de forma mais detalhada e sistematizada do que a especificação textual. Para casos relativamente pouco complexos, como o da ordenação por bolha, pode usar-se uma representação do problema bastante simples e eficaz: os fluxogramas.

Um fluxograma não é mais do que uma representação, meio textual, meio gráfica, do fluxo de controlo ao longo do algoritmo (isto é, que operações o algoritmo faz e qual o seu sequenciamento).

A Fig. 5.2 representa os principais elementos gráficos de um fluxograma. Existem mais símbolos, mas essencialmente são especializações do símbolo Operações. Estes quatro são os mais gerais e são suficientes para representar qualquer algoritmo. Testes mais complexos, com várias hipóteses de saída, podem sempre ser decompostos em vários testes binários.

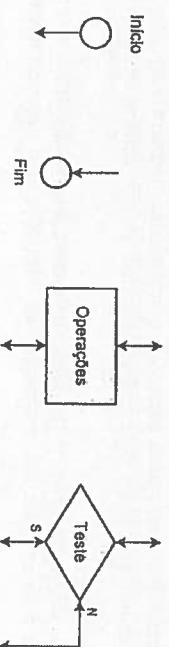


Fig. 5.2 - Elementos básicos de um fluxograma (S – Sim, N – Não)

**NOTA**

O fluxograma é uma notação muito simples, usada essencialmente para descrever o comportamento com um só fluxo de controlo. Para problemas mais complexos, torna-se necessário desenvolver um modelo do problema mais completo, que contemple os comportamentos individuais de cada uma das entidades referidas na especificação e as relações entre elas. Esse modelo expressa-se numa notação designada UML (*Unified Modeling Language*). No entanto, esta notação é complexa e está completamente fora do âmbito deste livro. Existe muita bibliografia sobre esta notação, usada universalmente para modelação ([Nunes 2004], [Silva 2005] e [Rational]).

A Fig. 5.3 apresenta um fluxograma adequado ao algoritmo da Fig. 5.1, assumindo os seguintes nomes:

- **houveTroca** – Elemento de memória que memoriza o facto de ter havido uma troca de números numa dada ronda. Conterá o valor verdadeiro se tiver havido troca, falso em caso contrário;
- **seq** – Sequência de números a ordenar;
- **i** – Índice do elemento de seq que se vai comparar com o elemento seguinte;
- **N** – Indica quantos números estão em seq.

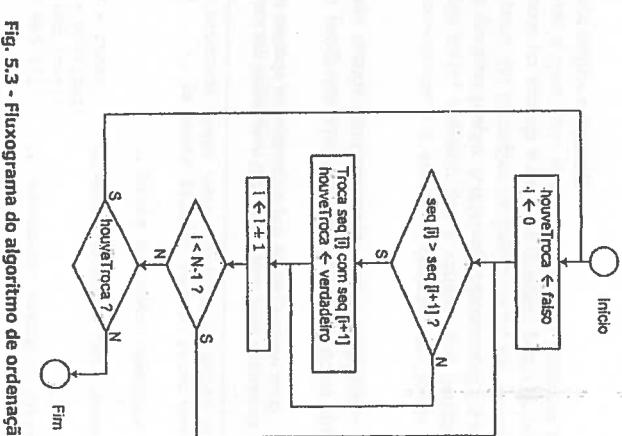


Fig. 5.3 - Fluxograma do algoritmo de ordenação por bolha

**NOTA**

Note-se que o termo de comparação com  $i$  no penúltimo teste é  $N-1$  e não  $N$ , pois no algoritmo acede-se a  $seq[i]$  e  $seq[i+1]$ . Como os índices dos vectores começam em zero, o valor máximo para um índice é  $N-1$  (em que  $N$  é o número de elementos do vetor) e neste caso o valor máximo de  $i$  para fazer nova iteração é  $N-2$  (pois o algoritmo faz referência a  $i+1$ , cujo valor máximo será o índice do último elemento do vector,  $N-1$ ).

Os fluxogramas são (devem ser) normalmente fáceis de perceber, até porque não há previsão para comentários. A disposição dos diversos símbolos, os textos no seu interior e as setas que definem os caminhos possíveis para o sequenciamento das operações contribuem para que os fluxogramas sejam uma forma eficaz de especificar o algoritmo de forma mais sistemática do que uma simples descrição textual.

De qualquer modo, um fluxograma não é um programa, e podem aparecer operações não totalmente especificadas em todos os seus detalhes. Na Fig. 5.3, a troca de  $seq[i]$  com  $seq[i+1]$  está apenas referida mas não detalhada. Esta é também uma forma de lidar com a complexidade. Se os detalhes aparecessem todos, não haveria vantagem em ter os fluxogramas como passos intermédios entre a especificação textual e os programas.

Um fluxograma deve assim ter mais detalhes e estar mais estruturado do que a especificação textual, mas deve ser mais simples do que o programa, para que também possa ser

mais facilmente compreensível. A Fig. B.2, na página 722, apresenta outro exemplo de fluxograma.

#### ESSENCIAL

- O modo de "pensar" de um computador é muito diferente do de um ser humano. Este preferiria terceias mais abstratas e de nível alto envolve o computador sem problemas. Mas perde rapidamente em termos de capacidade de memória e velocidade de processamento. Um tanto operativas é muito lento e precisa de muitas ajudas de memória.
- O computador funciona em base 2, para facilidade de implementação electrónica. Isto não é uma desvantagem face ao ser humano. Qualquer base numérica serve para fazer cálculos.
- Os seres humanos evoluem por aprendizagem e infetam novos conhecimentos a partir dos que já têm. Os computadores têm de ser programados com um algoritmo que preveja exatamente todas as situações possíveis.
- O programador deve adaptar os algoritmos ao modelo de computação do computador. Operações simples executadas de forma repetitiva constituem normalmente uma boa estratégia.
- Converter um conceito num algoritmo executável por um computador é normalmente uma tarefa de muito alto nível. É conceptualmente difícil de se dar a um ser humano.
- Para que o programador não perca este processo, deve usar ajudas de memória que lhe permitem especificar o problema num formato interno (modelo do problema), facilitando a compreensão.
- Os fluxogramas costumam uma forma simples de representar o comportamento do programa, apurando o seu fluxo de controlo numa forma muito textual gráfica, que é mais sintética e estruturada do que uma simples descrição textual em linguagem natural.

### 5.3 PROGRAMAÇÃO EM ALTO NÍVEL

Con os compiladores, os computadores podem ser programados em linguagens de programação de mais alto nível do que a linguagem *assembly*, como por exemplo C [Guerreiro 2001], C++ [Guerreiro 2003], Java [Mendes 2003] ou C# [Marques 2005]. O Programa 5.1 apresenta o algoritmo subjacente ao fluxograma da Fig. 5.3 em linguagem C, uma das linguagens de alto nível mais simples e mais divulgada. Os conjuntos /\* \*/ são comentários para benefício do programador, sendo ignorados pelo compilador.

Obvio que é mais fácil desenvolver este programa a partir do fluxograma do que com base apenas na Fig. 5.1 e no algoritmo enunciado na secção 5.1.2. O fluxograma é já um pre-programa!

```
NOTA
/* declaração de constantes */ /* quantidade de números a ordenar */
#define N 4
/* declaração de variáveis globais */
int seq [N] = {10, 5, 6, 2}; /* vetor (array) de N inteiros que conterá os
números a ordenar */

/* programa principal */
main ()
{
    /* declaração de variáveis locais */
    int houveTroca; /* indica se houve troca de numeros numa dada ronda */
    int i; /* posição (começa em 0) de um dado número no vector */
    int auxiliar; /* variável usada para a troca de números */

    Instruções */
    do
        houveTroca = false; /* ainda não houve trocas nesta ronda */
        /* começa a testar o vetor a partir da posição 0 */
        while (i < N-1)
            /* testa os números todos até ao fim do vetor */
            if (seq [i] > seq [i+1]) /* se o número seguinte é maior... */
            {
                auxiliar = seq [i];
                seq [i] = seq [i+1];
                seq [i+1] = auxiliar;
                houveTroca = true; /* agora já houve pelo menos uma troca */
                i = i + 1; /* passa ao número seguinte no vector */
            }
    while (houveTroca); /* se houve trocas, tem de fazer mais rondas */
    /* o programa acaba aqui. Os números estão todos ordenados */
}
```

Programa 5.1 - Programa em linguagem C para ordenar uma sequência de números usando o método de ordenação por bolha (bubble sort)

Apesar de este ser um programa simples, há alguns aspectos que convém frisar, em particular para os leitores menos familiarizados com a linguagem C, o que é feito pela Tabela 5.1.

As variáveis houveTroca, i e auxiliar designam-se "locais" por estarem definidas dentro da função main, enquanto que a variável seq designa-se "global" por estar definida fora de qualquer função. A diferença fundamental é que as primeiras só podem ser acedidas a partir das instruções da função main, não sendo conhecidas fora desta, enquanto seq pode ser acedida a partir de qualquer ponto do programa. A distinção no Programa 5.1 não é clara, pois só há instruções no programa principal, mas a secção 5.7.3.2 estabelece melhor a diferença.

| Conceito                    | Exemplos no programa                                    | Descrição                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------------|---------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| main                        | main () { <i>Instruções</i> }                           | Engloba as instruções do programa e define o ponto em que começa a sua execução (primeira instrução executável dentro do bloco entre chaves).                                                                                                                                                                                                             |
| /* */                       | /* programa principal */                                | Comentário (texto entre "/*" e "*/" é ignorado pelo compilador)                                                                                                                                                                                                                                                                                           |
| i<br>(terminador)           | i = i + 1;                                              | Carácter que indica o fim de uma instrução. Ao contrário das instruções em assembly em C as instruções podem começar numa linha e acabar noutra. A indentação é usada como elemento visual de estruturação de código, mas é ignorada pelo compilador.                                                                                                     |
| Constante símbólica literal | #define N 4                                             | Valor fixo, não alterável, valor atribuído na definição (com a directiva para o compilador #define). Não gera instruções para o processador, indica apenas ao compilador que sempre que encontrar N é o mesmo que encontrar 4 (não exemplo do Programa 5.1)                                                                                               |
| Constante literal           | N                                                       | Representação específica de um valor constante                                                                                                                                                                                                                                                                                                            |
| Varável simples             | int houveTroca ;<br>int auxiliar;                       | Elemento (célula) de memória, onde se pode escrever (armazenar) um valor para o ler mais tarde. Não gera instruções para o processador, apenas reserva espaço em memória para guardar o valor. O termo "variável" deriva precisamente do facto de se poder variar o valor do elemento de memória. O termo int indica que as variáveis são do tipo inteiro |
| Varável vectorial           | int seq [N] = {10, 5, 6, 2};<br>seq [i] = seq [i+1];    | Conjunto de variáveis simples reunidas sob o mesmo nome, passando cada uma das variáveis simples a ser conhecida pelo seu número de ordem (índice) dentro do conjunto. Os parênteses rectos permitem especificar qual o índice da variável (dentro do conjunto) que se pretende ler ou escrever.                                                          |
| Atribuição (=)              | houveTroca = false;<br>i = 0;                           | Operação de escrita de um valor (no lado direito do sinal de igual) numa variável (no lado esquerdo do sinal). Operação de leitura de uma variável faz-se pela simples utilização do seu nome numa expressão.                                                                                                                                             |
| Operação aritmética         | + (soma)                                                | Permite fazer contas com valores de variáveis ou constantes (para além da soma, também é possível especificar outras operações aritméticas)                                                                                                                                                                                                               |
| Operação relacional         | > (maior), < (menor)                                    | Operação que compara dois valores, produzindo como resultado um dos dois valores lógicos: verdadeiro ou falso.                                                                                                                                                                                                                                            |
| Círculo com teste no final  | do<br>{<br><i>Instruções</i><br>} while (houveTroca);   | Executa as instruções dentro das chaves e no final avalia a expressão booleana. Se for falsa, continua na instrução seguinte. Se for verdadeira, volta a executar as instruções e o teste à expressão.                                                                                                                                                    |
| Círculo com teste no inicio | while (i < N-1)<br>{<br><i>Instruções</i><br>}          | Identico ao anterior, mas com a diferença de que o teste à expressão booleana é feito no inicio e se der logo falso não executa nenhuma instrução do círculo.                                                                                                                                                                                             |
| Decisão                     | If (seq [i] > seq [i-1])<br>{<br><i>Instruções</i><br>} | Testa o resultado de uma expressão booleana. Se for verdadeiro, executa as instruções dentro das chaves. Se for falso, passa simplesmente à operação seguinte, sem executar as instruções.                                                                                                                                                                |

Tabela 5.1 - Principais aspectos da linguagem C no Programa 5.1

## 5.4 MAPEAMENTO DA PROGRAMAÇÃO DE ALTO NÍVEL EM LINGUAGEM ASSEMBLY

O objectivo de qualquer programa, seja em linguagem de alto nível, seja em linguagem assembly, é produzir código-máquina, isto é, instruções que o processador consiga executar directamente.

A Fig. 1.4, na página 13, mostra que os compiladores geram directamente código-máquina, pois todo o processo é automático após a programação em alto nível. No entanto, e para ilustrar a forma como um compilador analisa o programa em C e gera as instruções do PEPE correspondentes, vamos ver que instruções em linguagem assembly um programador deveria usar para produzir as mesmas instruções em código-máquina que o compilador irá gerar.

Visto de outra forma, isto equivale a supor que o compilador geraria primeiro o programa em linguagem assembly, que depois seria traduzido para código-máquina por um assembler. Afinal, a linguagem assembly não passa de uma representação simbólica do código-máquina, para que os humanos percebam as instruções que estão a ser dadas ao processador.

Neste processo, convém nunca esquecer que a linguagem C é de alto nível e independente da arquitetura do computador onde os programas correm. É responsabilidade do compilador gerar o código-máquina (ou instruções em assembly) adequado para o processador desse computador. Isto significa que cada computador tem de ter o seu próprio compilador de C (pelo menos a parte de geração de código tem de ser específica). Neste capítulo usa-se o PEPE como arquitetura alvo<sup>38</sup>, mas com outro processador as conclusões seriam semelhantes, com as necessárias adaptações. A portabilidade<sup>39</sup> é uma das grandes vantagens das linguagens de alto nível.

A Tabela 5.2 mostra um programa em que é feita a correspondência entre cada instrução no programa em C e as instruções assembly que lhe são equivalentes. Esta tabela é internamente ilustrativa e não deve ser interpretada como o resultado de um compilador real de C, pois num computador real há aspectos que não podem ser tratados de forma tão simplista. O programa assembly da Tabela 5.2 ilustra apenas a forma como as instruções de C podem ser implementadas no PEPE e tem a informação necessária para poder ser executado no simulador.

As secções seguintes explicam esta tabela em detalhe.

<sup>38</sup> Processador que irá executar o programa e cujo código deve ser gerado pelo compilador. Para o programa ser executado num processador diferente, o compilador tem de gerar outro código-máquina, com as instruções específicas desse processador.

<sup>39</sup> Capacidade de um programa poder ser executado em vários computadores sem alterações significativas.

| PROGRAMA EM C                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | PROGRAMA EM ASSEMBLY                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#define N 4 int seq [N] = {10, 5, 6, 2};  main () {     int houveTroca ;     int i;     int auxiliar;      do         houveTroca = false;         i = 0;         while (i &lt; N-1)         {             if (seq[i] &gt; seq[i+1])                 auxiliar = seq[i];                 seq[i] = seq[i+1];                 seq[i+1] = auxiliar;                 houveTroca = true;                 i = i + 1;             }             while (houveTroca);             fin: JMP fin ; acaba aqui         }</pre> | <pre>N EQU 4 ; #define N 4 PLACE 1000H seq: WORD 10 ; int seq [N] = {10, 5, 6, 2}; WORD 5 WORD 6 WORD 2 ; R0 - base do vector seq ; R1 - variável houveTroca ; R2 - variável i ; R3 - variável auxiliar PLCE 0000H main: MOV R0, seq ; base do vector ronda:     MOV R1, 0 ; houveTroca = false;     MOV R2, 0 ; i = 0; iteração:     MOV R7, N ; N     SUB R7, 1 ; N-1     CMP R2, R7 ; i &lt; N-1 ?     JGE teste ; se não, acabou o ciclo     MOV R9, R2 ; obtém cópia de i     SHL R9, 1 ; 2*i (ender. em bytes)     MOV R7, [R0+R9] ; seq[i]     MOV R10, R2 ; obtém cópia de i     ADD R10, 1 ; i+1     SHL R10, 1 ; 2*(i+1)     MOV R8, [R0+R10] ; seq[i+1]     CMP R7, R8 ; seq[i] &gt; seq[i+1] ?     JLE próximo ; não, Passa ao próximo     MOV R3, R7 ; auxiliar = seq [i];     MOV [R0+R9], R8 ; seq[i] = seq[i+1];     MOV [R0+R10], R3 ; seq[i+1]=auxiliar;     R1, 1 ; houveTroca = true; próximo:     ADD R2, 1 ; i = i + 1;     JMP iteração ; próxima iteração teste: CMP R1, 0 ; houveTroca = falso?     JNZ ronda ; mais uma ronda     fin: JMP fin ; acaba aqui</pre> |

Tabela 5.2 - Programa 5.1, em C, e as instruções correspondentes em linguagem assembly

**SIMULAÇÃO I – ORDENAÇÃO POR BOLHA**

Esta simulação ilustra o funcionamento do programa em linguagem assembly da Tabela 5.2. Os aspectos cobertos incluem os seguintes:

- Inicialização dos valores da memória;
- Localização dos blocos de dados e código;
- Reutilização de registos;
- Execução passo a passo e com pontos de paragem;
- Evolução dos registos relevantes e da memória, iteração a iteração.

**ESSENCIAL**

- Uma linguagem de programação de alto nível é uma notação textual (caracteres) que descreve de um conjunto de regras permitir especificar um algoritmo num formato suficientemente preciso para ser executado por um computador.
- Os computadores não executam directamente o programa de alto nível. Existe uma correspondência entre cada instrução da linguagem de alto nível e a linguagem assembly, que é também uma descrição textual mas já com um mapeamento directo nas instruções do código-máquina, que o computador sabe executar directamente.

A tradução entre as instruções da linguagem de alto nível para instruções em linguagem assembly (ou instruções máquina directamente) é feita pelo compilador.

- As grandes vantagens de utilizar uma linguagem de alto nível face a programar em linguagem assembly são fundamentalmente as seguintes:
  - Fáceis de escrever e de ler.
  - Fáceis de editar e de corrigir.
  - Fáceis de entender e de memorizar.
  - Fáceis de aprender e de aplicar.
  - O programa fica independente do processador em que é executado (o compilador é que faz a conversão adequada), suportando a portabilidade.

**5.5 DADOS, DECLARAÇÕES E DIRECTIVAS**

O Programa 5.1 começa pelo que se pode chamar declarações. Trata-se de informação para o compilador que não corresponde directamente a instruções mas que é importante para a produção do código-máquina. Estas declarações contemplam constantes e variáveis.

**5.5.1 CONSTANTES SIMBÓLICAS E A DIRECTIVA EQU**

A directiva EQU já foi introduzida na programação em linguagem assembly do PEPE-8 (secção 3.4.4, na página 160) e tem sido usada nos exemplos subsequentes, mesmo já no contexto do PEPE, permitindo definir o valor das constantes simbólicas. Sempre que no resto do programa (após a definição) aparecer uma destas constantes, o compilador substitui-a automaticamente pelo seu valor.

Em linguagem C, as constantes simbólicas são definidas de forma semelhante, por uma directiva para o compilador. Assim, se após:

```
#define N 4
/* quantidade de números todos a ordenar */
```

Aplicar:

```
while (i < N-1) /* testa os números todos até ao fim do vector */
```

O compilador interpreta como sendo:

```
while (i < 3)          /* testa os números todos até ao fim do vector */
    . . .
A grande vantagem das constantes simbólicas é que para mudar o seu valor basta mudá-la na declaração. Recompilando o programa, o novo valor entra em vigor automaticamente em todos os pontos do programa em que a constante for usada.
```

Em linguagem *assembly* do PEPE existe uma directiva para fazer a definição de constantes simbólicas, EQU<sup>40</sup>, bastando fazer:

```
N EQU 4                ; quantidade de números a ordenar
```

para obter o mesmo efeito, tal como indicado na Tabela 5.2.

O valor da constante deve ficar com 16 bits, em complemento para 2, o que significa que a gama de valores que se podem especificar nesta directiva vai de -32.768 (8000H) a +32.767 (7FFFH). O valor pode ser indicado em decimal, hexadecimal ou em binário.

Não há extensão com sinal. Se se indicar:

```
N EQU 0FFH              /* 255, ou 00FFH */
```

não fica com o valor 00FFH (255) e não FFFFH (-1 em 16 bits). A extensão com sinal só acontece nas instruções que podem receber uma constante em que não é possível especificar os 16 bits (ADD, por exemplo). Na directiva EQU as constantes podem ser especificadas com 16 bits.

**NOTA** Nos programas em linguagem *assembly*, as constantes hexadecimais que começam por uma letra (A a F) devem ter um 0 (zero) antes, mesmo que já tenha os quatro dígitos hexadecimais. Caso contrário, o assembler interpreta-as como um identificador e o resultado provável é dar um erro de identificador desconhecido. Este zero adicional à esquerda não tem qualquer efeito no valor da constante.

## 5.5.2 VARIÁVEIS

### 5.5.2.1 TIPOS DAS VARIÁVEIS

Ao contrário das constantes, as variáveis correspondem a elementos de memória, que podem memorizar valores para mais tarde serem lidos ou escritos novamente.

Cada variável numa linguagem de alto nível tem um tipo de dados associado (inteiro, carácter, etc.), que indica as características dos valores que podem ser armazenados nessa variável. O compilador dá um erro se o programador tentar armazenar um valor de um tipo numa variável de outro tipo.

Esta é uma diferença fundamental face à linguagem *assembly*, que não tem tipos de dados. Uma variável é apenas uma sequência de bits. O valor que lá está pode ser

<sup>40</sup> Do inglês equals ou equates, que significa "é igual a" ou "é equivalente a".

interpretado de uma forma ou outra consoante a instrução que usa esse valor. Por exemplo, uma instrução ADD interpreta os seus operandos como números em complemento para 2, em que 0000H (zero) é um número maior do que FFFFH (-1, em 16 bits). Uma instrução de AND, por exemplo, interpreta os operandos como números binários sem sinal, isto é, desde 0000H a FFFFH.

Assim, é o compilador que tem de fazer as verificações de tipo necessárias. Uma vez gerado o código, nem o assembler nem o hardware do processador verificam mais nada. Se o compilador tiver erros e não verificar bem os tipos (ou se o programador desenvolver o programa em linguagem *assembly* e usar um número em complemento para 2 onde devia usar um número sem sinal, por exemplo), não há mais avisos. Simplesmente o programa pode funcionar mal.

### 5.5.2.2 ACESSO A VARIÁVEIS DE TIPOS DE DADOS ESTRUTURADOS

Por outro lado, nas linguagens de alto nível os dados são estruturados. Um vector, como no Programa 5.1, é um conjunto de variáveis reunidas sob um nome comum, podendo aceder-se a cada elemento do vector por um número de ordem (índice) dentro do vector (seq[i]), por exemplo. Muitos compiladores verificam inclusivamente o número de elementos do vector, de modo que um acesso a partir do N-ésimo elemento gera um erro.

Em linguagem *assembly*, existe apenas a noção de elementos de memória individuais (células de memória ou registos). Apesar de existir suporte para acesso a vectores em memória, com os modos de endereçamento baseado e indexado (Tabela 4.31, na página 269), não há qualquer verificação de que a base tem o endereço do primeiro elemento do vector nem o uso de um valor de índice para além do último elemento do vector dá qualquer erro. Tem de ser o programador de linguagem *assembly* ou o compilador a inicializar o registo com o endereço de base do vector, como no caso de R0 na Tabela 5.2, e a garantir que o índice tem um valor adequado:

```
MOV R0, seq             ; inicializa base do vector
```

```
MOV R7, [R0+R9]          ; acesso a seq[1], no endereço R0+2*i
```

No acesso a vectores por meio de um índice, é fundamental lembrarmo-nos de que o índice se mede em número de elementos e não em bytes, tal como já indicado na secção 10.4.2. Essa é a razão pela qual o acesso às componentes do vector seq[i] e seq[i+1] na Tabela 5.2 se faz somando R0 (base do vector) não a R2 (o índice) mas sim aos registos R9 (que contém 2\*i) e R10 (que contém 2\*(i+1)), respectivamente, valores estes que têm de ser calculados em cada iteração porque o i vai variando. O factor 2 deriva do facto de cada componente do vector ter 2 bytes (16 bits). Seria 4 se cada elemento do vector tivesse 32 bits.

Este pormenor complica o acesso ao vector, tal como se pode ver na Tabela 5.2 na tradução para *assembly* da expressão booleana (seq[1] > seq[i+1]). Note-se o uso de uma instrução de deslocamento à esquerda (SHL) para efectuar a multiplicação por 2.

**NOTA** As instruções de deslocamento, à esquerda e à direita, (*SHL* e *SHR*) são muitas vezes usadas para efectuar multiplicações e divisões, respectivamente, por factores que são potências de 2 (ver dica na página 264). Uma instrução *ADD* constitui uma boa alternativa neste caso particular de multiplicação por 2.

Uma das decisões fundamentais a tomar pelo compilador (ou pelo programador de assembly) é onde colocar as variáveis, se em memória ou em registos. E sempre melhor usar registos, pois os acessos aos registos são sempre mais rápidos do que à memória, além de que os acessos à memória só podem ser feitos por instruções *MOV* enquanto que os registos estão acessíveis para a generalidade das instruções. O problema é que os registos são poucos, e por isso é preciso geri-los como recurso escasso que são.

Na Tabela 5.2, pode ver-se que se colocou o vector (*seq*) em memória (os vectores tipicamente têm muitos elementos, além de que não se pode aceder aos registos por índice) e as variáveis simples (houve troca, i e auxiliar) em registos, para o acesso ser mais eficiente.

Para colocar uma variável num registo basta tomar nota disso e começar a usar (é o compilador ou o programador que tem de verificar se está a aceder ao registo certo e que o mesmo registo não é usado ao mesmo tempo para mais de uma variável). Essa é a razão pela qual a declaração destas variáveis em C não corresponde a nada no programa em assembly (para além do comentário). Trata-se apenas de informação para o compilador.

**NOTA** Um pormenor digno de nota é a reutilização de registos. Na Tabela 5.2, *R9* é usado para conter primeiro *N-1* e depois *seq[1]*. Este duplo significado não tem qualquer inconveniente. Ambos são valores temporários, válidos apenas durante parte uma dada interação, e quando se usa *R9* para *seq[1]*, já o valor *N-1* deixou de ser preciso e portanto o registo pode ser reutilizado. Os compiladores mantêm uma lista de utilizações dos registos, e do conjunto de instruções em que o seu valor tem de ser válido, conseguindo assim uma utilização mais eficiente de um recurso que é escasso (os registos).

### 5.5.2.3 DIRECTIVAS WORD, TABLE E STRING

Para colocar variáveis em memória, tem de se reservar as células necessárias (a noção de reserva é explicada na secção 5.5.3). Para o vector *seq* precisamos de 4 células de 16 bits (4 palavras), que além disso devem ser inicializadas com os valores indicados no programa em C. Tanto a reserva de espaço como a inicialização com um valor podem ser conseguidas com a directiva WORD.

```
seq: WORD 10 ; int seq [N] = {10, 5, 6, 2};
      WORD 5
      WORD 6
      WORD 2
```

Cada directiva WORD reserva uma palavra (16 bits) e inicializa essa palavra com o valor indicado. Se o valor for omitido, reserva apenas o espaço da palavra mas não inicializa o seu conteúdo.

Em linguagem assembly, seq passa a ser uma etiqueta, cujo valor é o endereço da célula de memória em que o 10 é escrito (ou seja, o primeiro elemento do vector). Esse endereço pode depois ser usado para aceder ao vector, como indicado anteriormente.

**NOTA** WORD e EQU são duas directivas completamente diferentes.

```
N EQU 5 ; define um nome alternativo para a constante 5
var: WORD N ; reserva espaço em memória e inicializa-a com o valor de N (5)
      MOV R1, var ; endereço da variável definida pelo WORD
      MOV R2, [R1] ; copia o valor da variável para R2
```

EQU não passa de uma definição de um nome alternativo. Neste exemplo, *N* é apenas uma outra designação do número 5, com a vantagem de que se se alterar o valor da constante na directiva EQU, todas as referências a *N* passam a ser equivalentes a designar o novo valor. EQU não gera código-máquina nem sequer gasta espaço de memória na zona de dados.

A directiva WORD tem mais consequências:

- Reserva espaço para uma palavra em memória, na zona de dados, que pode ser acedida através da etiqueta *var*, tal como ilustrado no exemplo;
- Inicializa automaticamente (quando o programa inicia a sua execução) essa palavra com o valor indicado. Para implementar isto, o compilador pode de inserir automaticamente uma instrução de *MOV* para escrever o valor de inicialização nessa palavra de memória.

Note-se que o primeiro *MOV* não faz nenhum acesso à memória. Simplesmente, coloca no *R1* o endereço da palavra em memória que contém o *N*. O acesso à memória é feito pelo segundo *MOV*.

Existem outras duas directivas (TABLE e STRING) que permitem reservar espaço para variáveis em memória, uma sem e outra com inicialização do conteúdo. Exemplos:

```
etiqueta1: TABLE 20 ; reserva 20 palavras de memória sem
      etiqueta2: STRING "Bom dia!", 00H ; reserva espaço para a cadeia de
   ; caracteres, inicializa-o e coloca o byte
   ; terminador (00H) no fim
etiqueta3: STRING 6AH, 'A', 'B', 23H ; coloca estes 4 bytes no endereço
   ; etiqueta3 e nos três seguintes
```

- A directiva TABLE reserva o número indicado (20, no exemplo) de palavras (de 16 bits) de memória, sem as inicializar. A primeira palavra fica localizada no endereço especificado (etiqueta1, no exemplo) e as restantes vêm imediatamente a seguir, em endereços contíguos (isto é, de 2 em 2 pois cada palavra ocupa dois endereços). Esta directiva destina-se assim a reservar espaço para tabelas, que depois tem de ser inicializado pelo programa.
- Ao contrário das anteriores, a directiva STRING trabalha ao nível do byte individual. Permite especificar uma ou mais constantes, separadas por vírgulas, que podem ser:
  - Valores numéricos individuais (de um byte);

- Caracteres (em codificação ASCII, de um byte);

- Cadeias de caracteres (em que cada carácter é representado em ASCII por um só byte).

Os bytes são colocados sequencial e contiguamente a partir do endereço correspondente à etiqueta da directiva. No primeiro exemplo de STRING, no endereço correspondente a etiqueta2 ficará localizado o valor 42H (representação ASCII do carácter 'B'). No endereço etiqueta2 + 1 ficará localizado o valor 6FH (representação ASCII do carácter 'o') e assim sucessivamente. O número de bytes reservados é o necessário para conter todas as constantes especificadas. A primeira directiva STRING precisa de 9 bytes, enquanto que a segunda precisa apenas de 4 bytes. Note-se que se o programador quiser colocar um terminador (tipicamente 00H) após uma cadeia de caracteres, convenção usada na linguagem C para saber quando a cadeia acaba, terá de o fazer explicitamente, como indicado no exemplo.

Nenhuma destas directivas gera instruções. Apenas reservam espaço em memória algumas (WORD e STRING) inicializam-no. Esta inicialização ocorre quando o programa é carregado em memória (antes da sua execução, portanto), ao passo que o espaço reservado por TABLE terá de ser inicializado já com o programa em execução, usando instruções MOV.

A secção 5.8.4, na página 363, contém exemplos de manipulação de tabelas que podem ser criadas com estas directivas.

**NOTA**

As etiquetas que aparecem antes de uma directiva WORD ou TABLE ou de uma instrução têm de corresponder a um endereço par, pois referem-se a palavras de 16 bits e o PFCP suporta apenas endereçamento de palavra alinhado (secção 4.4, na página 196). A directiva STRING lida com sequências de bytes individuais, que podem começar e acabar em qualquer endereço (par ou ímpar), uma vez que estas sequências podem ocupar um número par ou ímpar de bytes. Por esta razão, o assembler adopta a seguinte política quando encontra uma etiqueta e pretende determinar a que endereço corresponde (usando o Contador de Endereços, CE – ver a secção 5.5.3):
 

- Se a etiqueta pertencer a uma directiva STRING, o assembler atribui-lhe o valor de CE, seja par ou ímpar;
- Caso contrário (se a etiqueta pertencer a uma directiva de palavras ou a uma instrução), força o alinhamento automaticamente, somando uma unidade ao próximo endereço utilizável se este for ímpar (de modo a que etiqueta fique sempre com um endereço par).

Um valor de CE ímpar pode ser provocado por uma directiva STRING com um número ímpar de bytes, como sucede nos exemplos acima, em que etiqueta1 será necessariamente par (devido a esta política), etiqueta2 será também par porque a TABLE reserva 20 palavras inteiras, mas etiqueta3 terá um valor ímpar, pois a primeira STRING ocupa 9 bytes (número ímpar). Como a segunda STRING ocupa 4 bytes (par), o assembler terá de somar 1 ao CE se logo a seguir vier uma instrução ou directiva de palavra (WORD ou TABLE).

Esta directiva aparece duas vezes no programa em assembly da Tabela 5.2 e tem por objectivo fazer a atribuição de endereços, quer aos dados, quer às instruções, indicando o endereço a partir do qual o conjunto de directivas (WORD, TABLE, etc.) ou instruções seguintes são colocadas.

Ao traduzir o programa assembly para código-máquina, o assembler vai mantendo um contador de endereços (CE), que tem 16 bits e no início da tradução está a 0000H. O CE é incrementado sempre que aparece uma instrução ou uma directiva que reserve espaço em memória e é usado para atribuir valores às etiquetas que vão aparecendo ao longo do programa.

Se o assembler encontrar uma directiva PLACE, coloca no CE o valor do endereço indicado nessa directiva. A Tabela 5.3 indica o que acontece ao CE de acordo com as directivas e instruções que o assembler encontrará ao traduzir o programa em assembly para código-máquina. Note-se que a tradução é feita analisando o programa assembly sequencialmente, desde o seu inicio até ao seu fim.

| SE O ASSEMBLER ENCONTRAR    |                         | O QUE O ASSEMBLER FAZ                                                                                                                                                                                              |  |
|-----------------------------|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Início do programa assembly |                         | CE < 0000H                                                                                                                                                                                                         |  |
| símbolo EQU valor (16 bits) |                         | símbolo < valor                                                                                                                                                                                                    |  |
| PLACE                       | endereço                |                                                                                                                                                                                                                    |  |
|                             |                         | CE < endereço                                                                                                                                                                                                      |  |
| etiqueta:                   | WORD valor (16 bits)    | Se CE for ímpar, CE < CE + 1 (para CE fitar par)<br>etiqueta < CE (só se a etiqueta for especificada)<br>Mv[CE] < valor<br>CE < CE + 2                                                                             |  |
| etiqueta:                   | TABLE número            | Se CE for ímpar, CE < CE + 1 (para CE fitar par)<br>etiqueta < CE (só se a etiqueta for especificada)<br>CE < CE + (2 × número)                                                                                    |  |
| etiqueta:                   | STRING valor1... valorK | etiqueta < CE (só se a etiqueta for especificada)<br>M[CE] < 1.º byte<br>M[CE+1] < 2.º byte<br>...<br>M[CE+N-1] < N-ésimo byte<br>(N ≥ K devido às cadeias de caracteres)<br>CE < CE + N (N pode ser par ou ímpar) |  |
| etiqueta:                   | ASCIIZ                  | Se CE for ímpar, CE < CE + 1 (para CE fitar par)<br>etiqueta < CE (só se a etiqueta for especificada)<br>Mv[CE] < Instrução/máquina gerada (16 bits)<br>CE < CE + 2                                                |  |
| Cadeia de caracteres ASCII. |                         |                                                                                                                                                                                                                    |  |

Tabela 5.3 - Geração de endereços durante o processo de tradução de um programa em linguagem assembly para código-máquina. As etiquetas são opcionais. Mv e Mb significam acesso à memória em palavra e byte, respectivamente

A Tabela 5.4 ilustra todo este processo, indicando os endereços atribuídos durante a tradução do programa assembly da Tabela 5.2 para código-máquina.

| ENDEREÇO (CE) | PROGRAMA EM ASSEMBLY                     |
|---------------|------------------------------------------|
| N             | EQU 4 ; #define N 4                      |
| PLACE         | 1000H ; localiza bloco de dados          |
| 1000H         | WORD 10 ; int seq [N] = {10, 5, 6, 2};   |
| 1002H         | WORD 5                                   |
| 1004H         | WORD 6                                   |
| 1006H         | WORD 2                                   |
|               | ; R0 - base do vector seq                |
|               | ; R1 - variável houveTroca               |
|               | ; R2 - variável i                        |
|               | ; R3 - variável auxiliar                 |
|               | ; PLACE 0000H ; localiza bloco de código |
| main:         | MOV R0, seq ; base do vector             |
| 0000H         | MOV R0, seq ; houveTroca = false;        |
|               | iteração:                                |
| 0002H         | MOV R1, 0 ; obtém cópia de i             |
| 0004H         | MOV R2, 0 ; i = 0;                       |
| 0006H         | MOV R7, N ; N                            |
| 0008H         | SUB R7, 1 ; N-1                          |
| 000AH         | CMP R2, R7 ; i < N-1 ?                   |
| 000CH         | JGE teste ; se não, este ciclo acabou    |
| 000EH         | MOV R9, R2 ; obtém cópia de i            |
| 0010H         | SHL R9, 1 ; 2*i (endereço em bytes)      |
| 0012H         | MOV R7, [R0+R9] ; seq [i]                |
| 0014H         | MOV R10, R2 ; obtém cópia de i           |
| 0016H         | ADD R10, 1 ; i+1                         |
| 0018H         | SHL R10, 1 ; 2*(i+1)                     |
| 001AH         | MOV R8, [R0+R10] ; seq [i+1]             |
| 001CH         | CMP R7, R8 ; seq [i] > seq [i+1] ?       |
| 001EH         | JLE próximo ; não, passa ao próximo      |
| 0020H         | MOV R3, R7 ; auxiliar = seq [i];         |
| 0022H         | MOV [R0+R9], R8 ; seq [i] = seq [i+1];   |
| 0024H         | MOV [R0+R10], R3 ; seq [i+1] = auxiliar; |
| 0026H         | MOV R1, 1 ; houveTroca = true;           |
| 0028H         | i = i + 1; ; proxima iteração            |
| 002AH         | TESTE: CMP R1, 0 ; houveTroca = falso?   |
| 002CH         | JNZ ronda ; mais uma ronda.              |
| 002EH         | FIM: JNE FIM ; acaba aqui.               |

Tabela 5.4 - Endereços atribuídos pelo assembler ao traduzir este programa assembly para código-máquina. Note-se o efeito das directivas PLACE no valor de CE (coluna da esquerda)

Sobre estas tabelas importa salientar os aspectos seguintes:

- Tudo isto acontece durante o processo de conversão das instruções em linguagem assembly para instruções em código-máquina. Não envolve qualquer execução do programa, apenas trabalho do assembler;
- O principal objectivo do CE (contador de endereços) é permitir ao assembler atribuir valores às etiquetas que o programador especifica no programa assembly;

No início da tradução do programa assembly para código-máquina, o CE é inicializado a 0000H e evolui de acordo com as instruções e directivas que o assembler vai encontrando.

Uma directiva PLACE altera o valor do CE para o endereço indicado na directiva, cuja função é permitir localizar os blocos de dados e de código (instruções) nos endereços pretendidos. Noteadamente, deve ter-se em atenção que o PEPE começa a sua execução após um *reset* no endereço 0000H. Por isso, o programa assembly deve usar uma de duas soluções:

- Colocar instruções logo no início, antes de qualquer directiva de reserva de espaço de dados em memória, aproveitando o facto de que o CE é inicializado a 0000H (corresponde a uma directiva inicial PLACE 0000H por omissão);
  - Colocar uma directiva PLACE 0000H antes de um conjunto de instruções, não importando a sua posição relativa no programa assembly. Esta é a solução usada no programa da Tabela 5.2.
- Não há qualquer problema em o CE "voltar atrás". Significa apenas que atribuiu primeiro endereços numa zona de dados e depois passou para uma zona de instruções que fica localizada numa zona de endereços mais baixos. O fundamental é que não haja sobreposição entre as duas zonas. Se por engano tal acontecer, escrever numa zona de dados (ao executar o programa) pode na realidade destruir instruções do código-máquina!
- Não confundir o CE com o PC (*Program Counter*):

- O PC existe apenas durante a execução do programa, enquanto o CE existe apenas durante a tradução do programa assembly para código-máquina (antes da execução);
- É normal o PC voltar ao mesmo valor (ciclos no programa), mas o CE só pode voltar ao mesmo valor por erro das directivas PLACE, acarretando sobreposição de zonas de dados e/ou código. Como exemplo de um erro típico, suponhamos que um dado programa tem um bloco de instruções localizado no endereço 1000H por meio de uma directiva PLACE e um bloco de dados localizado no endereço 2000H por meio de outra directiva PLACE. O programador começa a desenvolver o seu código, que inicialmente ocupa menos de 1000H endereços (espaço disponível entre os dois blocos, de instruções e dados). Com o evoluir do programa (mais e mais instruções), se o número de endereços gerado pelas instruções do bloco de código chegar aos 1000H, passará a haver sobreposição entre endereços de código e dados, o que é um erro grave. Felizmente, os assemblers podem detectar esta situação facilmente e reportar o erro ao utilizador, que deve alterar os endereços nas directivas PLACE de modo a afastar mais os dois blocos, de instruções e dados, e a evitar a sobreposição.
- A directiva EQU não tem qualquer influência no CE nem este tem qualquer efeito no valor do símbolo que é definido;

- As directivas de 16 bits (WORD e TABLE) e as instruções são automaticamente alinhadas (o CE é incrementado de uma unidade para ficar com um valor par) caso a última evolução do CE tenha sido provocada por uma directiva de *byte* (STRING) que tenha deixado o CE com um valor ímpar;
- Uma etiqueta é uma constante simbólica cujo valor é dado pelo CE quando a directiva ou instrução em que ela aparece é processada pelo assembler;
- Nem todas as acções indicadas para cada directiva ou instrução são executadas imediatamente. Como algumas das constantes simbólicas podem ser etiquetas, é necessário efectuar uma primeira passagem pelo programa para calcular todos os endereços e só depois, numa segunda passagem, atribuir os valores correctos às etiquetas.

A Tabela 5.5 é a tabela de símbolos deste programa, indicando para cada constante simbólica qual o seu valor.

| SÍMBOLO  | VALOR | TIPO DE CONSTANTE     | QUEM ATRIBUIU O VALOR |
|----------|-------|-----------------------|-----------------------|
| N        | 4     | Dado                  | Programador (com EQU) |
| seq      | 1000H | Endereço de dado      | Assembler             |
| main     | 0000H | Endereço de instrução | Assembler             |
| ronda    | 0002H | Endereço de instrução | Assembler             |
| iteração | 0006H | Endereço de instrução | Assembler             |
| próximo  | 0028H | Endereço de instrução | Assembler             |
| teste    | 002CH | Endereço de instrução | Assembler             |
| fin      | 0030H | Endereço de instrução | Assembler             |

Tabela 5.5 - Tabela de símbolos. Só os definidos com EQU têm o seu valor atribuído directamente pelo utilizador. Os símbolos de endereço (etiquetas) têm o seu valor atribuído pelo assembler, embora com base nas directivas PLACE.

### 5.5.2 - DIRECTIVA PLACE

Esta simulação ilustra o funcionamento da directiva PLACE no programa em linguagem assembly da Tabela 5.4. Os aspectos cobertos incluem os seguintes:

- Localização dos blocos de dados e código;
- Verificação dos valores da Tabela de Símbolos;
- Variação dos valores das directivas PLACE;
- Problema da não existência de instruções no endereço 0000H;
- Variante com localização do bloco de dados no meio de dois blocos de código (imediatamente antes da etiqueta teste).

Esta simulação ilustra o funcionamento da directiva PLACE no programa em linguagem assembly da Tabela 5.4. Os aspectos cobertos incluem os seguintes:

- Localização dos blocos de dados e código;
- Verificação dos valores da Tabela de Símbolos;
- Variação dos valores das directivas PLACE;
- Problema da não existência de instruções no endereço 0000H;
- Variante com localização do bloco de dados no meio de dois blocos de código (imediatamente antes da etiqueta teste).

Esta simulação ilustra o funcionamento da directiva PLACE no programa em linguagem assembly da Tabela 5.4. Os aspectos cobertos incluem os seguintes:

- Localização dos blocos de dados e código;
- Verificação dos valores da Tabela de Símbolos;
- Variação dos valores das directivas PLACE;
- Problema da não existência de instruções no endereço 0000H;
- Variante com localização do bloco de dados no meio de dois blocos de código (imediatamente antes da etiqueta teste).

Esta simulação ilustra o funcionamento da directiva PLACE no programa em linguagem assembly da Tabela 5.4. Os aspectos cobertos incluem os seguintes:

- Localização dos blocos de dados e código;
- Verificação dos valores da Tabela de Símbolos;
- Variação dos valores das directivas PLACE;
- Problema da não existência de instruções no endereço 0000H;
- Variante com localização do bloco de dados no meio de dois blocos de código (imediatamente antes da etiqueta teste).

Esta simulação ilustra o funcionamento da directiva PLACE no programa em linguagem assembly da Tabela 5.4. Os aspectos cobertos incluem os seguintes:

- Localização dos blocos de dados e código;
- Verificação dos valores da Tabela de Símbolos;
- Variação dos valores das directivas PLACE;
- Problema da não existência de instruções no endereço 0000H;
- Variante com localização do bloco de dados no meio de dois blocos de código (imediatamente antes da etiqueta teste).

- As directivas de 16 bits (WORD e TABLE) e as instruções são automaticamente alinhadas (o CE é incrementado de uma unidade para ficar com um valor par) caso a última evolução do CE tenha sido provocada por uma directiva de *byte* (STRING) que tenha deixado o CE com um valor ímpar;
- Uma etiqueta é uma constante simbólica cujo valor é dado pelo CE quando a directiva ou instrução em que ela aparece é processada pelo assembler;
- Nem todas as acções indicadas para cada directiva ou instrução são executadas imediatamente. Como algumas das constantes simbólicas podem ser etiquetas, é necessário efectuar uma primeira passagem pelo programa para calcular todos os endereços e só depois, numa segunda passagem, atribuir os valores correctos às etiquetas.

O uso de apontadores tem sido advogado essencialmente nos seguintes tipos de situação:

- Referência a blocos de dados criados dinamicamente, isto é, durante a execução do programa (secção 5.8.5 na página 378). As variáveis declaradas no programa são estáticas, isto é, são aloçadas (o seu espaço é reservado) pelo compilador ou pelo assembler, com directivas do tipo WORD ou TABLE. No entanto, as linguagens de alto nível também suportam alocação dinâmica de novas variáveis durante a execução do programa (por exemplo, através da função malloc em C ou do operador new em Java), e a referência a essas variáveis só pode ser feita pelo seu endereço, uma vez que não foram declaradas com um dado nome;
- Passagem de parâmetros por referência para funções (secção 5.7.1.1), em que uma função recebe um apontador para o parâmetro em vez de uma cópia desse parâmetro;
- Acesso à zona de dados de uma forma mais directa, sem as restrições próprias da estruturação de dados. O problema é que o compilador não pode saber que valor irá ter um apontador quando o programa correr, pelo que não pode verificar se o programador está a aceder de forma correcta, ou sequer se está a aceder à zona de dados!

Esta última utilização permite todas as barbaridades em termos de programação e é algo que deve ser evitado, já que deixa facilmente de funcionar quando se declara uma nova variável ou se troca a ordem de declaração (o que altera a ordem relativa da localização das variáveis em memória). É muito fácil o programador esquecer-se das assunções que fez ao programar com apontadores, alterar o programa numa coisa tão simples como declarar uma nova variável e este deixar de funcionar "inexplicavelmente". Estes problemas são muito difíceis de detectar.

Com as estruturas de dados dinâmicas (que podem ser criadas e eliminadas pelo programa), existe um problema adicional, que conduz a erros demasiado frequentes: quer aceder a uma estrutura de dados que já se eliminou, usando o apontador que estava válido antes de se eliminar estes dados. O apontador ainda aponta para o mesmo endereço, mas a célula de memória correspondente pode já ter sido usada para alocar

Um dos aspectos mais clássicos das linguagens de programação de alto nível é a dicotomia entre variáveis e apontadores para essas variáveis. Em linguagem assembly,

outras variáveis dinâmicas, e portanto os dados que lá estão já não têm significado do ponto de vista desse apontador.

Para resolver estes problemas, linguagens mais recentes (como Java) pura e simplesmente retiraram os apontadores enquanto construções explícitas suportadas pela linguagem, proporcionando ao programador um ambiente de programação mais automático e controlado. Note-se que em baixo nível os apontadores continuam a existir (a memória continua a ter endereços), só que controlados automaticamente pelo compilador e pelo software de apoio ao programa durante a sua execução.

Meramente em termos didácticos, esta secção pretende mostrar como se podem usar apontadores e a sua correspondência em linguagem assembly. Para esse efeito, o Programa 5.2 mostra como é possível implementar em C a mesma funcionalidade do Programa 5.1 com apontadores para a estrutura de dados a ordenar, em vez de usar uma variável de dados estruturada. O vector ainda é declarado, mas apenas para reservar espaço para ele e inicializar os seus elementos. A sua utilização faz-se apenas com apontadores.

Notas sobre este programa:

- num é uma variável do tipo "apontador para valores do tipo inteiro". Não contém um valor inteiro, mas sim o endereço da célula de memória em que se localiza o valor inteiro para que num aponta;
- Em C, o nome de um vector pode também ser usado como apontador para o primeiro elemento desse vector. Assim, a instrução num = seq; permite inicializar num com o endereço do primeiro elemento do vector;
- O operador \* permite referenciar o valor para que o apontador aponta, pelo que a expressão \*num equivale a especificar o nome da variável para que num aponta;
- A linguagem C suporta aritmética de apontadores. Isto quer dizer que somar 1 a num significa na realidade obter o apontador para o próximo elemento (neste caso do tipo inteiro, que é o tipo de base com que num foi declarado) e não o próximo endereço. Sendo os inteiros de 16 bits, ocupam dois endereços (o endereçamento é de byte). Portanto, num+1 significa na realidade somar 2 unidades no endereço que está em num. Somar assim um valor a um apontador não é realmente uma simples soma de valores. As unidades a somar a um apontador medem-se assim em elementos e não em bytes. O objectivo é fazer com que o utilizador não tenha de se preocupar com os bytes que cada tipo de dados ocupa. Desde que o apontador seja declarado como apontando para um tipo X, somar 1 a esse apontador equivale a somar K ao endereço desse apontador, em que K é o número de bytes que esse tipo X ocupa;
- A variável i está agora a ser usada apenas como contador das iterações, e não como índice de acesso ao vector. Para passar ao próximo elemento do vector é preciso ir actualizando o próprio apontador num, o que é feito com a instrução num = num + 1; (que na realidade soma 2 ao valor armazenado em num pelas razões expostas no item anterior);

Em cada ronda, num é inicializado com o endereço de base do vector para que possa percorrer novamente o vector desde o seu primeiro elemento.

```
* declaração de constantes */
#define N 4 /* quantidade de números a ordenar */

/* declaração de variáveis globais */
int seq [N] = {10, 5, 6, 2}; /* vector (array) de N inteiros a ordenar */
int auxiliar; /* programa principal */

int * num; /* apontador para um dos elementos do vector */
int houveTroca; /* indica se houve troca de números numa ronda */
int i; /* posição (começa em 0) de um número no vector */
/* variável usada para a troca de números */

instruções */
do {
    /* início do ciclo de ronda (faz pelo menos uma) */
    num = seq; /* inicializa o apontador com o endereço do 1.º elemento */
    houveTroca = false; /* ainda não houve trocas nesta ronda */
    i = 0; /* começa a testar o vector a partir da posição 0 */
    while (i < N-1)
        if (*num > *(num+1)) /* se o número seguinte é maior... */
            {
                auxiliar = *num; /* ... troca-os, usando a variável auxiliar */
                *num = *(num+1);
                *(num+1) = auxiliar;
                houveTroca = true; /* agora já houve pelo menos uma troca */
            }
        i = i + 1; /* mais um número tratado */
        num = num + 1; /* passa a apontar para o número seguinte no vector */
    } while (houveTroca); /* se houve trocas, tem de fazer mais rondas */
} /* o programa acaba aqui. Os números estão todos ordenados */
```

**Programa 5.2 - Programa em C para ordenar uma sequência de números usando o método de ordenação por bolha (bubble sort) e apontadores para aceder ao vector**

A Tabela 5.6 mostra como esta versão do programa em C, com apontadores, pode ser convertida para linguagem assembly. O acesso ao vector fica mais simples porque existe um apontador que vai percorrendo o vector, não havendo necessidade de estar a somar um índice ao apontador de base do vector.

### 5.5.3 - ORDENAÇÃO POR BOLHA COM APONTADORES

A simulação ilustra o funcionamento do programa em linguagem assembly da Tabela 5.6. Os aspectos cobertos incluem os seguintes:

- Execução passo a passo e com pontos de paragem;
- Verificação da evolução dos registos relevantes e da memória, iteração a iteração, nomeadamente o apontador e a célula para que ele aponta.

| PROGRAMA EM C                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | PROGRAMA EM ASSEMBLY                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#define N 4 int seq [N] = {10, 5, 6, 2}; main () {     int * num;     int houveTroca ;     int i;     int auxiliar;     do         num = seq;         houveTroca = false;         i = 0;         while (i &lt; N-1)         {             if (*num &gt; *(num+1))                 auxiliar = *num;                 *num = *(num+1);                 *(num+1) = auxiliar;                 houveTroca=true;             }             i = i + 1;             num = num + 1;         while (houveTroca);     } }</pre> | <pre>N EQU 4 ; #define N 4 PLACE 1000H ; localiza bloco de dados seq WORD 10 ; int seq [N] = {10, 5, 6, 2}; WORD 5 WORD 6 WORD 2 ; R0 - variável num (apontador) ; R1 - variável houveTroca ; R2 - variável i ; R3 - variável auxiliar PLACE 0000H ; localiza bloco de código ronda: MOV R0, seq ; num = seq; MOV R1, 0 ; houveTroca = false; i = 0; iteração: MOV R7, N ; N SUB R7, 1 ; N-1 CMP R2, R7 ; i &lt; N-1 ? JGE teste ; se não, o ciclo acabou MOV R7, [R0] ; num MOV R8, [R0+2] ; *(num+1) CMP R7, R8 ; *num &gt; *(num+1)? JLE próximo ; não, passe ao próximo MOV R3, R7 ; auxiliar = *num; MOV [R0], R8 ; *num = *(num+1); MOV [R0+2], R3 ; *(num+1) = auxiliar; MOV R1, 1 ; houveTroca = true; próximo: ADD R2, 1 ; i = i + 1; ADD R0, 2 ; num = num + 1; JMP Iteração ; próxima iteração teste: CMP R1, 0 ; houveTroca = false? JNZ ronda ; mais uma ronda fim: JMP fim ; acaba aqui</pre> |

Tabela 5.6 - Programa 5.2, em C e em linguagem assembly

## 5.6 INSTRUÇÕES

As linguagens de programação de alto nível têm instruções que permitem implementar operações de manipulação de dados, nomeadamente atribuição a variáveis, tomar decisões e repetir sequências de instruções (ciclos), tipicamente para iterar estruturas de dados.

A linguagem *assembly* implementa um conjunto muito mais reduzido e básico de operações, no qual as instruções das linguagens de alto nível têm de se mapar. As secções seguintes descrevem como é que tipicamente isto é feito.

### ESSENCIAL

- Como boa regra de programação, em alto nível ou em linguagem assembly, devem usar-se constantes simbólicas (definidas com a directiva EQU) em vez de constantes literais. O nome explica melhor o seu significado e mudar o valor na declaração muda automaticamente em todas as suas utilizações.

- Em linguagem assembly os dados (variáveis ou constantes) não têm tipo. Um valor não passa de um conjunto de bits; cujo significado depende da operação que o usa, como operando.

As variáveis em assembly também não são estruturadas. Um array de C, por exemplo, é mapeado numa sequência de variáveis simples (célula de memória) que têm de ser accedidas individualmente;

Os índices das variáveis estruturadas em linguagens de alto nível contam elementos (mesmo que estes sejam estruturas de dados complexas). Em linguagem assembly, os índices usados nas instruções de acesso à memória (MOV) medem-se em endereços, ou seja, em bytes. Tem de ser o compilador (ou o programador de linguagem assembly) a fazer a conversão.

As variáveis simples mais usadas colocam-se em registos. As restantes têm de ser colocadas em memória (o número de registos é muito limitado).

As directivas WORD, TABLE e STRING usam-se para reservar espaço de memória e inicializá-lo (excepto TABLE), tornando a estrutura de dados do programa mais clara. Tipicamente, estas declarações colocam-se no início do programa, usado como variável (lido e escrito).

As directivas EQU e WORD são diferentes: EQU define apenas o valor de uma constante simbólica, que é substituída por esse valor sempre que o seu nome aparecer no programa. WORD reserva um endereço em memória para poder ser usado como variável (lido e escrito).

A directiva PLACE permite controlar manualmente a localização de blocos de dados ou de instruções no espaço de memória. É o assembler que vai atribuindo (sequencialmente) os endereços às instruções ou dados que aparecem declarados no programa.

A ordem pela qual os vários blocos aparecem no programa não interessa e os blocos de dados podem estar entrelaçados com os de instruções. No entanto, tem de haver um bloco de instruções com um PLACE:0000H, pois este é o endereço onde o processador arranca após uma re-inicialização (reset).

Um apontador não passa de um endereço em memória através do qual é possível aceder a dados. Tem sempre a mesma dimensão (uma palavra), seja qual for a estrutura de dados para que aponta, permitindo referenciar qualquer estrutura de dados (simples ou não) por um mero registo.

## 5.6.1 ATRIBUIÇÃO E EXPRESSES

A instrução em C

$i = i + 1;$

soma 1 à variável  $i$  e o resultado dessa soma é atribuído à (armazenado na) variável  $i$ . O sinal “ $=$ ” define uma instrução de atribuição. A expressão no lado direito do sinal “ $=$ ” é avaliada primeiro e só depois o valor resultante é armazenado na variável referida no lado esquerdo desse sinal. Esta sequência permite que se possa usar uma dada variável em ambos os lados do sinal “ $=$ ”. No lado direito é lida, no lado esquerdo é escrita com um novo valor.

Em linguagem assembly, uma atribuição traduz-se normalmente por uma instrução de transferência de dados, MOV, que copia o valor de um lado para outro (secção 4.10).

No caso particular da instrução indicada atrás, é ainda mais simples, pois basta somar 1 ao registo que contém a variável  $i$ , continuando a soma no mesmo registo (R2, nos exemplos anteriores). Em assembly até nem parece haver uma atribuição, mas repare-se na mesma instrução em RTL. Na realidade, todas as instruções em assembly que produzam um resultado que deva ser armazenado têm uma atribuição implícita.

$i = i + 1; /* em assembly: ADD R2, 1 em RTL: R2 <- R2 + 1 */$

| ATRIBUIÇÃO                       | EXEMPLO EM C               | EXEMPLO EM ASSEMBLY                                                                                                                                   | COMENTÁRIOS                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Operação especial                | $i = i + 1;$               | ADD R2, 1                                                                                                                                             | A variável $i$ está localizada no R2                                                                                                                                                                                                                                                                                                                                   |
| Variável simples em registo      | $a = b;$                   | MOV R1, R3                                                                                                                                            | Variáveis $a$ e $b$ estão nos registos R1 e R3. Se $b$ não estiver num registo, tem de ser copiada para lá primeiro                                                                                                                                                                                                                                                    |
| Variável simples em memória      | $c = d;$                   | MOV [R4], R5                                                                                                                                          | R4 tem de ser inicializado previamente com o endereço de $c$ . Se $d$ não estiver num registo, tem de ser copiada para lá primeiro                                                                                                                                                                                                                                     |
| Elemento de variável estruturada | $e[i] = g;$<br>$e[i] = g;$ | MOV [R6+R7], R8<br>MOV [R5+R1], R8                                                                                                                    | Variáveis $e$ , $f$ e $g$ estão nos registos R6, R7 e R8. Se $g$ não estiver num registo, tem de ser copiada para lá primeiro                                                                                                                                                                                                                                          |
| Variável estruturada             | $h = m;$                   | MOV R7, base de $h$<br>MOV R8, base de $m$<br>MOV R9, n.º bytes<br>A: MOV B, [R8]<br>MOV B, [R7], R10<br>ADD R8, 1<br>ADD R7, 1<br>SUB R9, 1<br>JNZ A | h e m são vectores ou estruturas (necessariamente em memória). R7 e R8 são inicializados com os endereços de base de cada variável estruturada e R9 com o tamanho em bytes (n.º bytes). Deverá ser do mesmo tipo de dados, por isso devem ocupar o mesmo número de bytes. Depois, é só copiar (pode haver implementações mais eficientes que copiem palavra a palavra) |

Tabela 5.7 - Conversão de alguns casos da instrução de atribuição em C para linguagem assembly

A Tabela 5.7 apresenta alguns casos de instruções de atribuição, em que a variável atribuída pode ser simples (em registo ou em memória) ou estruturada (necessariamente em memória). As instruções de assembly variam consoante os casos, e podem mesmo constituir um pequeno programa, no caso de atribuição de variáveis estruturadas. É responsabilidade do compilador gerar as instruções correctas, consoante o caso.

No lado direito da atribuição aparece normalmente uma expressão. As secções 5.4, 5.5.2.2, 5.5.4 e 5.8.2, e em particular a Tabela 5.2 e a Tabela 5.6 dão indicações e exemplos de como fazer este mapeamento. O estudo sistemático da geração de código por parte de um compilador está fora do âmbito deste livro, podendo consultar-se [Crespo 2001].

## 5.6.2 DECISÃO

Sem testes a condições e respectivas tomadas de decisão os programas não passariam de uma sequência linear de instruções, sem hipótese de reagirem à evolução dos acontecimentos.

### 5.6.2.1 DECISÃO SIMPLES

Os testes usam tipicamente expressões booleanas, que produzem um de dois valores (verdadeiro ou falso). A instrução de decisão por exceção, i.e., pode contemplar dois conjuntos de instruções, executando apenas um deles, dependendo do valor que a expressão booleana de teste retornar numa dada altura.

Esta instrução implementa-se em linguagem assembly recorrendo a saltos condicionais. Por exemplo, a instrução em C:

```
if (expressão-booleana) { /* avalia expressão-booleana */
    instrução-A;
} else { /* instrução-B; */
    instrução-C;
    instrução-D;
}
```

/\* só chega aqui se expressão-booleana for falsa \*/

Na Tabela 5.7 apresenta-se nas seguintes instruções em linguagem assembly (apenas a título exemplificativo):

expressão-booleana ; avalia expressão-booleana e produz Z=1 se falso

instrução-A ; se for falso, salta para as instruções do else

instrução-B ; só chega aqui se expressão-booleana for verdadeira

JMP proxima ; salta por cima das instruções do else

instrução-C ; só chega aqui se expressão-booleana for falsa

instrução D ; instrução a seguir ao if

Nos casos em que a instrução if não tem cláusula else (a Tabela 5.2 mostra um exemplo), fica simplesmente:

```

expressão-booleana ; avalia expressão-booleana e produz Z=1 se falso
JZ próxima ; se for falso, não executa nenhuma instrução
instrução-a ; só chega aqui se expressão-booleana for verdadeira
instrução-B ; instrução a seguir ao if
próxima: . . .
    
```

### 5.6.2.2 DECISÃO MÚLTIPLE

As linguagens de alto nível suportam também instruções de decisão múltipla, que permitem testar expressões que possam tomar vários valores (e não apenas dois como as booleanas) e agilhar para um de vários blocos de instruções, sendo executado apenas um o correspondente ao valor que a expressão tiver na altura que for avaliada. No caso da linguagem C, a instrução de decisão múltipla é a instrução switch, cuja sintaxe pode ser exemplificada por:

```

switch (expressão) {
    case valor1 : instruções_caso1; /* chega aqui apenas se expressão = valor1 */
        break;
    case valor2 : instruções_caso2; /* chega aqui apenas se expressão = valor2 */
        break;
    case valor3 : instruções_caso3; /* chega aqui apenas se expressão = valor3 */
        break;
    case valor4 : instruções_caso4; /* chega aqui apenas se expressão = valor4 */
        break;
    default : instruções_omissão; /* chega aqui apenas se expressão não for igual a nenhum dos valores referidos */
}
    
```

As instruções break permitem sair da instrução switch, continuando na instrução que se lhe seguir. Se não se usar break no caso valor2, por exemplo, sempre que expressão for igual a valor2 são executadas as instruções\_caso2 e instruções\_caso3 (se não houver break, a execução de um caso continua pelas instruções do caso seguinte).

Em linguagem assembly, há dois métodos básicos de implementação desta instrução:

- Com testes múltiplos;
- Com uma tabela de endereços (secção 5.8.4.3, na página 374).

Com testes múltiplos, verifica-se sucessivamente se expressão é igual a cada um dos valores possíveis. É o caso mais geral, mas se por acaso o valor de expressão callhar a ser dos últimos nos testes tem se de efectuar todos os testes anteriores, ficando menos eficiente do que o caso dos primeiros valores a ser testados. Neste caso, as instruções em assembly correspondentes às instruções em C do switch poderiam ser as seguintes:<sup>41</sup>

```

switch (expressão) {
    case 0: instruções_caso0; break; /* chega aqui apenas se expressão = 0 */
    case 1: instruções_caso1; break; /* chega aqui apenas se expressão = 1 */
    case 2: instruções_caso2; break; /* chega aqui apenas se expressão = 2 */
    case 3: instruções_caso3; break; /* chega aqui apenas se expressão = 3 */
    default: instruções_omissão; /* chega aqui apenas se expressão não for */
}
    
```

As correspondentes instruções em assembly poderiam ser as seguintes:

```

PLACE 1000H ; zona de dados
tabela: WORD caso0 ; endereço da 1.ª instrução da sequência que trata o caso 0
WORD caso1 ; endereço da 1.ª instrução da sequência que trata o caso 1
WORD caso2 ; endereço da 1.ª instrução da sequência que trata o caso 2
WORD caso3 ; endereço da 1.ª instrução da sequência que trata o caso 3
    
```

<sup>41</sup> Algo simplificadas. Por exemplo, a instrução CMP só poderá ser usada com constantes desde que o valor destas esteja entre -8 e +7. Esta é uma restrição específica do PEPÉ, devida ao facto de apenas haver 16 bits para codificar uma instrução. Outros processadores poderão não ter esta restrição ou ter outras diferentes.

```

expressão ; calcula expressão, deixando o valor no R1
CMP R1, valor1 ; expressão = valor1?
JNZ caso2 ; se não, vai testar o caso seguinte
instruções_caso1 ; executa as instruções do caso 1
JMP Próxima ; break (sai do switch)
caso2: CMP R1, valor2 ; expressão = valor2?
JNZ caso3 ; se não, vai testar o caso seguinte
instruções_caso2 ; executa as instruções do caso 2
JMP Próxima ; break (sai do switch)
caso3: CMP R1, valor3 ; expressão = valor3?
JNZ caso4 ; se não, vai testar o caso seguinte
instruções_caso3 ; executa as instruções do caso 3
JMP Próxima ; break (sai do switch)
caso4: CMP R1, valor4 ; expressão = valor4?
JNZ omissão ; se não, vai tratar de todos os restantes casos
instruções_caso4 ; executa as instruções do caso 4
JMP próxima ; break (sai do switch)
omissão: instruções_omissão ; executa as instruções dos restantes casos
próxima: . . .
    
```

No caso da tabela de endereços, define-se uma entrada nessa tabela para cada valor possível de expressão contendo o endereço da primeira instrução do bloco correspondente a esse valor. Depois indexa-se a tabela com o valor que expressão tiver numa dada altura para determinar o endereço das instruções que tratam desse caso. Este método é mais eficiente porque todos os casos demoram igual tempo a testar (basta indexar a tabela) mas só é aplicável em casos bem comportados (mas que ainda assim são frequentes), em que os valores sejam inteiros contíguos e em número de casos limitado (por causa do tamanho da tabela). Os valores podem começar em zero (caso do exemplo seguinte) ou não, caso em que simplesmente o compilador subtraia automaticamente o menor valor a todos, de modo a considerar que começam em zero, para facilitar o acesso à tabela. As instruções em C seriam algo como neste exemplo:

```

switch (expressão) {
    case 0: instruções_caso0; break; /* chega aqui apenas se expressão = 0 */
    case 1: instruções_caso1; break; /* chega aqui apenas se expressão = 1 */
    case 2: instruções_caso2; break; /* chega aqui apenas se expressão = 2 */
    case 3: instruções_caso3; break; /* chega aqui apenas se expressão = 3 */
    default: instruções_omissão; /* chega aqui apenas se expressão não for */
}
    
```

<sup>41</sup> Algo simplificadas. Por exemplo, a instrução CMP só poderá ser usada com constantes desde que o valor destas esteja entre -8 e +7. Esta é uma restrição específica do PEPÉ, devida ao facto de apenas haver 16 bits para codificar uma instrução. Outros processadores poderão não ter esta restrição ou ter outras diferentes.

```

JGE  omissão ; se R1 não for inferior, não está contemplado na tabela
SHL  R1, 1 ; multiplica R1 por 2 (um endereço na tabela são 2 bytes)
MOV  R2, [R2+R1] ; obtém endereço de base da tabela de endereços
JMP  R2 ; salta para lá
caso: instruções_caso0
caso1: instruções_caso1 ; executa as instruções do caso 0
caso2: instruções_caso2 ; executa as instruções do caso 1
caso3: instruções_caso3 ; executa as instruções do caso 2
        JMP  próxima ; break (sai do switch)
        JMP  próxima ; executa as instruções do caso 3
        JMP  próxima ; break (sai do switch)
        JMP  omissão ; executa as instruções dos restantes casos
próxima: instruções_omissão ; instrução a seguir ao switch
    
```

Note-se a construção automática da tabela através das etiquetas e das directivas WORD. Se houver alteração nos endereços das instruções que tratam cada caso (por se terem acrescentado ou apagado instruções, por exemplo), basta correr de novo o assembler para os novos valores das etiquetas caso0 a caso3 serem calculados e a tabela ficará automaticamente com os novos valores.

### 5.6.3 ITERAÇÃO

Outro dos aspectos imprescindíveis numa linguagem de programação é a provisão de instruções para iteração, que permitem implementar ciclos de processamento. A linguagem C prevê três instruções para este fim:

- while
- do-while
- for

O Programa 5.1, na página 281, contém exemplos das duas primeiras, podendo ver-se o código correspondente em assembly na Tabela 5.2. No caso geral, um ciclo while como este:

```

while (expressão-booleana) { /* enquanto expressão-booleana for verdadeira... */
    instruções
}
    
```

Pode traduzir-se para linguagem assembly por instruções como estas:

```

ciclo: expressão-booleana ; calcula expressão-booleana. Se for falsa (0),
                    ; z fica a 1
                    ; JZ próxima ; se expressão-booleana for falsa, sai da instrução
                    ; instruções ; executa as instruções dentro do while
                    ; JMP ciclo ; vai avaliar expressão-booleana de novo para repetir
próxima: . . . ; próxima instrução
    
```

A outra forma de iteração, do-while, é semelhante mas o teste vem no fim, o que quer dizer que, mesmo que a expressão-booleana seja falsa à partida e assim se mantenha, as instruções são executadas pelo menos uma vez.

#### ESSENÇIA

■ A linguagem de alto nível liberta o programador dos detalhes do programa, que em baixo nível (assembly) fazem parte de um encadeamento supressivo dos níveis.

■ A instrução de atribuição de variáveis é simplesmente uma operação normalmente num simples MOV. O compilador encarrega-se de a atravésar o localizado em registos ou em memória para gerar as instruções demandadas. A atribuição com estruturas de dados complexas pode ser feita através de instruções comuns de memória.

■ As instruções de decisão simples (IF) aparecem em instruções de comando de salto condicional. As instruções de decisão múltipla (IF...) podem implicar-se em sucessivos testes e, a los condicionais sueltas, podem optar por parar indexação de uma tabela eender o resultado para um deles.

■ As instruções de iteração (WHILE...) são divididas em dois tipos: encadeamentos condicionais (com um salto condicional para uma instrução já ejecutada) e uma dada condição (que tem um bloco). As instruções de loop só avançam uma variável de índice, mas elas próprias podem operar num bloco interno.

■ As expressões usadas nestas instruções implicam consequências de instruções do processador que procuram um ou vários operandos, cada vez guardando resultados intermédios em registos intermediários e produzindo o resultado final.

Na prática, uma instrução for é equivalente a uma instrução while, podendo reescrever-se o programa desta forma equivalente:

```

#define N 4           /* definição do valor de N */
int vector[N];      /* declaração de um vector de N inteiros */
int i;              /* declaração da variável de índice */
/* inicializa a variável de índice */
while (i < N) {
    /* controlo do fim do ciclo */
    /* inicializa a zero todas as componentes do vector */
    vector[i] = 0;
    /* incrementa de uma unidade a variável de índice */
    i++;
}

```

A instrução `i++` é equivalente a `i=i+1`, e traduzir-se-á em linguagem assembly ADD R1, 1 (assumindo que a variável i se localiza no registo R1).

5.7 ROTINAS

## 5.7.1 ESTRUTURAÇÃO DO CÓDIGO

### 5.7.1.1 FUNÇÕES NAS LINGUAGENS DE ALTO NÍVEL

É comum um programa necessitar de executar uma dada funcionalidade em vários pontos do algoritmo. Nos primeiros tempos da programação (década de 60, grosso modo), ou se repetiam as instruções que implementavam essa funcionalidade (mas tal gasto de memória, nessa altura escassa e cara) ou fazia-se simplesmente um salto para as instruções onde a funcionalidade pretendida estava implementada.

Havia no entanto dois problemas fundamentais:

- Poder fazer saltos para qualquer ponto do programa sem critérios de estruturação do código originava grandes problemas, pois era frequente o programador alterar um dado conjunto de instruções esquecendo-se de alguma situação em que esse código também era usado (uma vez que não havia controlo sobre a origem dos saltos) e que era incompatível com essa alteração. A proliferação de saltos de estilo de programação originou mais tarde em retrospectiva um termo sugestivo “Código esparrente”.

- Uma vez executada a funcionalidade, ainda havia o problema de retornar a execução no ponto original de onde se tinha saltado. Isto implicava mais saltos, com recurso a mecanismos *ad hoc* e de difícil manutenção em caso de alterações.

C) que organizaram as instruções em instruções compostas e em funções (descrições a seguir) e passaram a restringir os saltos, não deixando sair para dentro de uma instrução composta ou de uma função. A partir da década de 80, apareceram outras linguagens de programação (itáis orientadas para objectos), como Smalltalk-80, Eiffel, C++, Java e C# que além do código estruturaram também os dados.

No entanto, isto passa-se apenas em termos de programação de alto nível. Em linguagem assembly, tudo continua a ser permitido, não apenas porque estas regras são sobrejetivas para evitar os erros dos programadores, cuja programação é quase exclusivamente em alto nível, mas também por motivos de flexibilidade. Por este motivo o programador

**Fig. 5.4 - Chamadas de uma função.** Os retângulos são instruções executadas sequencialmente. Os retângulos mais escuros são da chamada da função, que retorna sempre para a instrução seguinte aquela que a chamou

Uma função é um conjunto de instruções logicamente relacionadas, uma espécie de subprograma que pode ser chamado para executar uma dada funcionalidade, com

- Existe uma instrução específica para chamar uma função; Quando se chama uma função, o processador interrompe a sequência de instruções que estava a executar e passa a executar as instruções dessa função. Quando a função termina, o processador retoma a execução do programa na instrução a seguir à que chamou essa função;

Uma mesma função pode ser chamada a partir de vários pontos do programa, voltando sempre para a instrução seguinte àquela que a chamou;

Uma função pode ter parâmetros (ou, nome alternativo, argumentos), que são variáveis cujos valores são determinados imediatamente antes da sua chamada. Isto permite que as instruções da função possam ser usadas em vários casos diferentes, em que os valores dos parâmetros podem variar de caso para caso;

Uma função pode retornar um valor (o resultado do seu processamento). Se a função for chamada numa expressão, o valor retornado substitui a chamada dessa função, permitindo o cálculo do valor final dessa expressão.

linguagem *assembly* deve ser auto-disciplinado, evitando truques obscuros e preferindo clareza nos programas, tal como nas linguagens de alto nível.

O Programa 5.1 ordena um vector de números e constitui um programa completo. Trata-se apenas de um exemplo académico, sem interesse real. Nenhum faz um programa para

ordenar numeros e termina esse programa sem fazer nada com esses numeros. Sera mais natural as instruções de ordenação constituiram uma função, que depois pode ser chamada sempre que se necessitar de ordenar um vector.

As funções constituem um mecanismo fundamental de estruturação das instruções de um programa, havendo duas razões fundamentais para se usarem:

- Partilha de código - Se se pretender usar a funcionalidade que a função implementa em vários pontos do programa, basta chamar essa função várias vezes, em vez de repetir as instruções da função em cada um desses pontos;
- Clareza do programa - Mesmo que um dado conjunto de instruções seja usado apenas uma vez em todo o programa, mesmo assim pode ter interesse em organizar-lo como uma função, pois o programa fica mais estruturado e claro. Uma sequência potencialmente extensa de instruções é substituída por uma ou mais chamadas a funções (que contêm estas instruções). Não só essa sequência fica agora muito mais curta como também o nome das funções pode ajudar a transmitir a funcionalidade dessas instruções.

As funções não são uma forma de conseguir executar um programa mais depressa. Bem pelo contrário, o simples facto de ter de chamar uma função e depois retornar ao ponto onde foi chamada acarreta alguns atrasos de processamento. Mas em programação há muito tempo que aprendemos que mais vale ter um programa bem estruturado e que siga as regras do que um muito eficiente mas cheio de truques (que facilmente se transformam em erros), mesmo que isso acarrete algumas penalizações em tempo de execução.

O Programa 5.3 ilustra estes princípios, estruturando o Programa 5.1 em duas funções, ficando o programa principal (main) apenas com instruções para chamar estas funções.

```
#define N 4           /* quantidade de numeros a ordenar no vector seq1 */
#define P 6           /* quantidade de numeros a ordenar no vector seq2 */
int seq1[N] = {10, 5, 6, 2};    /* vector (array) de N inteiros */
int seq2[P] = {20, 7, 5, 10, 4, 8}; /* vector (array) de P inteiros */

main () {
    /* programa principal */
    ordenaBolha (seq1, N);
    ordenaBolha (seq2, P);
}

void ordenaBolha (int a[], int n) { /* função que ordena um vector */
    int houveTroca; /* indica se houve troca de números numa dada ronda */
    int i; /* posição (começa em 0) de um dado número no vector */
    do {
        /* inicio do ciclo de ronda (pelo menos uma tem de fazer) */
        houveTroca = false; /* ainda não houve trocas nesta ronda */
        i = 0; /* começa a testar o vector a partir da posição 0 */
        /* testa os numeros todos até ao fim do vector */
        /* ordena um dado número e o seguinte. houveTroca fica com true caso
         * tenha havido troca dos dois números */
        houveTroca = houveTroca || ordena (&a[i], &a[i+1]);
        i = i + 1; /* passa ao número seguinte no vector */
    } while (houveTroca); /* se houve trocas, tem de fazer mais rondas */
}
```

Programa 5.3 - Programa em linguagem C, estruturado em funções, para ordenar uma sequência de números usando o método de ordenação por bolha (*bubble sort*)

O algoritmo de ordenação é encerrado na função ordenaBolha, para poder ser chamado várias vezes. A função ordena é chamada apenas a partir de um sítio (da função ordenaBolha), mas torna o programa mais claro porque a função ordenaBolha fica mais simples (desse ponto de vista, uma função tão simples não era particularmente necessária, mas ilustra o princípio). Por outro lado, o facto de a função ordena retornar um valor (booleano<sup>42</sup>) permite usá-la numa expressão, simplificando o código.

Neste programa, os seguintes aspectos devem ainda ser tidos em conta:

- O programa principal (função main) contém apenas duas chamadas à função ordenaBolha, passando-lhe como parâmetros em cada caso a informação relativa a cada um dos vectores definidos como variáveis globais;
- A função ordenaBolha:
  - Não retorna valor nenhum (por isso tem o tipo de retorno como void);
  - Recebe dois parâmetros, o vector a ordenar e a sua dimensão (número de elementos). Quando retornar, o vector estará ordenado. O parâmetro a é na realidade o endereço do vector (isto é, do seu primeiro elemento) e não uma cópia do vector, pois em C qualquer uso do nome de um vector sem especificar um índice é o mesmo que ter o endereço desse vector e não um dado elemento nem uma cópia completa do vector. Assim, todas as operações que esta função efectuar sobre o vector a são na realidade feitas sobre os vectores seq1 e seq2 (na primeira e segunda chamadas a ordenaBolha, respectivamente). Esta passagem de parâmetros para uma função (apontadores em vez de cópias dos valores) designa-se passagem por referência. Já o parâmetro n usa passagem por valor, isto é, o que é passado como parâmetro é uma cópia dos valores N e

<sup>42</sup> Mas neste caso usa-se int porque a linguagem C não tem nenhum tipo de dados booleano. O valor zero corresponde a falso e qualquer outro valor corresponde a verdadeiro.

```
int ordena (int * x, int * y) { /* função que ordena dois valores, passados por
referência */
    int auxiliar; /* se o número corrente é maior do que o seguinte... */
    auxiliar = *x;
    /* ...então troca-os, usando a variável auxiliar */
    *x = *y;
    *y = auxiliar;
    return true;
}
else
    /* retorna falso se não tiver havido troca */
```

P e não o endereço das células de memória que eventualmente os contenham.

A secção 5.7.3.3 contém detalhes adicionais sobre este assunto;

- Esconde a ordenação dos elementos do vector na função ordena (nem sequer os testa para ver se já estão pela ordem correcta, confiando em ordena para isso), ficando mais simples;

- Chama a função ordena, passando-lhe como parâmetros os apontadores para dois elementos consecutivos do vector (passagem por referência) em vez dos elementos em si. O operador “&” permite obter os endereços em memória destes elementos, e são esses endereços que são passados à função ordena;
- Acumula em houveTroca qualquer valor verdadeiro retornado por ordena, através de uma operação de OR lógico (operador “||”).

A função ordena:

- Tem dois parâmetros, correspondentes a dois números que tem de testar. Se o primeiro for maior do que o segundo, troca-os, para que a ordenação seja por ordem crescente. Se for menor ou igual não faz nada (porque nesse caso os valores já estão ordenados);
- Estes parâmetros são apontadores para esses números (para as células de memória que os contêm – ver secção 5.5.4) e não os números em si. Ou seja, os parâmetros são passados por referência. Desta forma, esta função troca directamente os valores no vector a, ou na realidade os vectores seq1 e seq2 (na primeira e segunda chamadas a ordenaBolha no programa principal, respectivamente);
- Retorna um valor booleano, verdadeiro se tiver trocado os números e falso se o não tiver feito. Em C, os valores booleanos não existem como tipo de dados autónomo, pelo que se usa o tipo int para este efeito (embora existam as constantes predefinidas true e false, com os valores 1 e 0, respectivamente);
- Tem uma variável local, auxiliar, que só é válida enquanto a função estiver a ser executada.

### 5.7.1.2 ROTINAS EM LINGUAGEM ASSEMBLY

Em linguagens de programação de alto nível, os blocos de instruções chamam-se funções (ou procedimentos, quando não retornam valor). Em linguagem assembly, o termo mais comum é rotina. No fundo, são a mesma coisa: um conjunto de instruções que no seu conjunto implementam uma dada funcionalidade e que se podem chamar a partir de qualquer ponto do programa, havendo um mecanismo que permite retornar para a instrução a seguir à que fez a chamada.

Como é que se especificam e chamam rotinas em linguagem assembly? A Tabela 5.8 estabelece a correspondência, instrução a instrução, entre o Programa 5.3, em C, e um possível programa equivalente em linguagem assembly.

| PROGRAMA EM C                           | PROGRAMA EM ASSEMBLY                                                             |
|-----------------------------------------|----------------------------------------------------------------------------------|
| #define N 4<br>#define P 6              | N EQU 4 ; #define N 4<br>P EQU 6 ; #define P 6                                   |
| int seq1 [N] = {10, 5, 6, 2};           | PLACE 1000H ; localiza bloco de dados seq1: WORD 10 ; int seq [N]={10, 5, 6, 2}; |
| int seq2 [P] = {20, 7, 5, 10, 4, 8};    | seq2: WORD 20 ; int seq2 [P] = {20, 7, 5, 10, 4, 8};                             |
| ordenabolha (seq2, P);                  | WORD 5<br>WORD 7<br>WORD 10<br>WORD 4<br>WORD 8                                  |
| }                                       | PLACE 0000H ; localiza bloco de código main: MOV SP, 2000H ; inicializa SP       |
| void ordenabolha (int a[], int n)       | MOV R6, seq1 ; 1.º parâmetro                                                     |
| {                                       | MOV R7, N ; 2.º parâmetro                                                        |
| int houveTroca ;                        | CALL ordenabolha ; chama a função                                                |
| do                                      | MOV R6, seq2 ; 1.º parâmetro                                                     |
| houveTroca = false;                     | MOV R7, P ; 2.º parâmetro                                                        |
| i = 0;                                  | CALL ordenabolha ; chama a função                                                |
| while (i < n-1)                         | fim: JMP fim ; fim do programa                                                   |
| houveTroca = houveTroca                 | ordenabolha: ; a vem no R6, n vem no R7                                          |
| ordena (&a[i], &a[i+1]);                | ; i usa o R9                                                                     |
| i = i + 1;                              | R10 é usado para valores temporários                                             |
| }                                       | MOV R8, 0 ; houveTroca = false;                                                  |
| i = 0;                                  | MOV R9, 0 ; i = 0;                                                               |
| while (i < n-1)                         | iteração:                                                                        |
| MOV R10, R7 ; cópia de n                | MOV R10, R7                                                                      |
| SUB R10, 1 ; n-1                        | SUB R10, 1                                                                       |
| CMP R9, R10 ; i < n-1 ?                 | CMP R9, R10                                                                      |
| JGE teste ; se não, o ciclo acaba       | JGE teste                                                                        |
| MOV R10, R9 ; obtém cópia de i          | MOV R10, R9                                                                      |
| SHL R10, 1 ; 2*i (ender. em bytes)      | SHL R10, 1                                                                       |
| ADD R10, R6 ; endereço de a [i]         | ADD R10, R6                                                                      |
| MOV R1, R10 ; 1.º Parâmetro (&a[i])     | MOV R1, R10                                                                      |
| MOV R10, R9 ; obtém cópia de i          | MOV R10, R9                                                                      |
| ADD R10, 1 ; i+1                        | ADD R10, 1                                                                       |
| SHL R10, 1 ; 2*(i+1)                    | SHL R10, 1                                                                       |
| ADD R10, R6 ; endereço de a [i+1]       | ADD R10, R6                                                                      |
| MOV R2, R10 ; 2.º Parâmetro (&a[i+1])   | MOV R2, R10                                                                      |
| CALL ordena ; chama a função            | CALL ordena                                                                      |
| OR R8, R1 ; houveTroca    ordena        | OR R8, R1                                                                        |
| ADD R9, 1 ; i = i + 1;                  | ADD R9, 1                                                                        |
| iteração ; proxima iteração             | iteração                                                                         |
| teste: CMP R8, 0 ; houveTroca = falso ? | teste: CMP R8, 0                                                                 |
| JNZ ordenabolha ; mais uma roda         | JNZ ordenabolha                                                                  |
| }                                       | RET                                                                              |
| ordenabolha (int *x, int *y)            | ordenab: ; retorna da função                                                     |
| {                                       | ordena: ; x vem em R1, y vem em R2                                               |
| int auxiliar;                           | ; R3 e R4 usam-se como registos temporários                                      |
| if (*x > *y)                            | auxiliar usa o R5                                                                |
| }                                       | MOV R3, [R1]; *x                                                                 |

| PROGRAMA EM C                                                                                                                            | PROGRAMA ASSEMBLY                                                                                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> int auxiliar; int *x, *y; int a[10]; int n;  auxiliar = *x; *x = *y; *y = auxiliar; return true; } else     return false; } </pre> | <pre> MOV R4, [R2] ; *y CMP R3, R4 ; (*x &gt; *y) ? JLE else ; se não, executa o else MOV R5, R3 ; auxiliar = *x; MOV [R1], R4 ; *x = *y; MOV [R2], R5 ; *y = auxiliar; MOV R1, 1 ; true; RET else:     MOV R1, 0 ; false;     RET ; o valor retornado está no R1 } </pre> |

Tabela 5.8 - Programa 5.3, em C, e as instruções em linguagem assembly que lhe correspondem

Notas sobre o programa principal (função main):

- A instrução `MOV SP, 2000H` destina-se a inicializar um registo fundamental para as rotinas: o apontador de pilha, ou *SP* (*Stack Pointer*), que será descrito na secção 5.7.2.3;

- As instruções `CALL` são responsáveis pela implementação das setas “chamada” na Fig. 5.4. Estas instruções são descritas na secção 5.7.2. Antes de a rotina ser chamada, os parâmetros têm de ser preparados (*R6* e *R7*), com os valores que depois a rotina vai ler para poder efectuar o seu processamento;

- Em cada chamada da função os parâmetros são preenchidos com valores diferentes, o que se traduz em processamento diferente por parte da função. No entanto, as instruções chamadas são as mesmas, pelo que organizando-as numa rotina se evita ter de duplicar o código com apenas pequenas diferenças (correspondentes aos parâmetros).

- NOTA** Numa computadora real, o programa principal (*main*) é apenas uma rotina invocada pelo sistema operativo, que é responsável por inicializar o *SP*. A forma como a rotina *main* está implementada destina-a apenas a simplificar a realidade, para que o programa em linguagem *assembly* seja executado no simulador.

Notas sobre a rotina ordenaBolha:

- Esta rotina não retorna nenhum valor (o tipo de retorno em C é `void`), pelo que a rotina retorna sem preocupações de colocar um dado valor num dado registo (ao contrário do que sucede na função ordena);
- A função em C não tem uma instrução `return` explícita. Simplesmente chega ao fim e retorna. O compilador insere automaticamente o código necessário para o retorno, pois a instrução `RET` tem de lá estar, explicitamente (é esta instrução que faz o controlo regressar à instrução seguinte à que chamou esta função), tal como se pode ver na parte da linguagem *assembly*;

- O parâmetro *a* é passado por referência. Tal é indicado pelos parênteses rectos (`int a[]`), que indicam tratar-se de um vector, o que significa que o que é passado para a rotina é o endereço do primeiro elemento do vector e não o vector em si. Por isso, o *R6* (que corresponde a este parâmetro) é usado nesta rotina não como dado mas sim como endereço, para calcular os endereços das componentes do vector (sendo somado com  $2 \cdot i$ , em que o factor 2 se deve ao facto de os inteiros ocuparem 2 bytes cada um);
- O parâmetro *n* é passado no *R7* por valor (não há apontadores envolvidos), sendo usado como um dado (neste caso, o número de componentes do vector) e não como um endereço;
- O cálculo dos endereços de *a[i]* e *a[i+1]* são feitos de forma independente, isto é, começando por determinar o índice, multiplicando-o por 2 por causa do endereço de base do vector. Para nós, humanos, é fácil reconhecer que estes são dois elementos do vector consecutivos, e que portanto, uma vez conhecido o endereço de *a[i]*, o endereço de *a[i+1]* poderia ser obtido simplesmente somando 2 ao endereço de *a[i]* e evitar repetir os cálculos anteriores. Já não é tão fácil para o compilador reconhecer esta situação, embora seja uma optimização possível de incorporar no compilador. Outra forma de conseguir isto seria programar a função `ordenaBolha` em termos de apontadores, tal como se fez no Programa 5.2, em vez de usar o índice para aceder às componentes do vector. A secção 5.7.1.3 apresenta esta variante.

Notas sobre a rotina ordena:

- A instrução `RET` é o equivalente em linguagem *assembly* da instrução `return` em C e implementa as setas de “retorno” da Fig. 5.4. A secção 5.7.2 explica o funcionamento desta instrução. O valor retornado está no *R1*, onde pode ser accedido por “quem” chamou esta rotina;
- Os compiladores actuais fazem muitas optimizações no código gerado. O uso de *R5* era dispensável, podendo usar-se o *R3* para o mesmo fim. Isto evitaria a instrução `MOV R5, R3` (é `MOV [R2], R5` ficaria `MOV [R2], R3`). Trata-se apenas de um detalhe, mas ilustra o facto de que o código gerado pode não ser o que se espera. Tudo depende de como o compilador gera o código e das optimizações que faz;
- Esta rotina destrói o valor anterior (antes da sua chamada) dos registos *R3*, *R4* e *R5*. Isto pode ser gravoso porque a rotina não sabe de onde pode ser chamada, e a sequência de instruções que a chamou pode ter valores úteis armazenados nesses registos, que assim são destruídos pela rotina. Uma boa regra de programação é que uma rotina não deve alterar nenhum registo para não interferir com o contexto em que é chamada. Mas como, se a rotina precisa de usar os registos para efectuar o seu processamento? A resposta será dada na secção 5.7.3.1.

### 5.7.1.3 VARIANTE COM APONTADORES

A função ordenaBolha do Programa 5.3 recebe o vector de números a ordenar como um parâmetro do tipo vector de inteiros. Tal como se fez no Programa 5.2, é possível usar apontadores directos para os elementos do vector em vez de lhes aceder através de uma indexação do vector.

O Programa 5.4 mostra a variante à função ordenaBolha, usando agora apontadores. Tanto o programa principal (main) como a função ordena do Programa 5.3 se mantêm sem alterações. A variável i é agora usada apenas para contar os elementos do vector já testados e não para aceder ao vector.

A Tabela 5.9 apresenta a correspondência entre esta variante da função ordenaBolha e a correspondente rotina em linguagem assembly, em que se pode verificar a implementação dos apontadores. Recordemos que somar 1 a um apontador equivale a somar ao endereço que lhe corresponde um número de unidades igual ao número de bytes que ocupa o elemento para que o apontador aponta (secção 5.5.4). Neste caso o elemento é um inteiro de 16 bits, logo tem de se somar 2 ao endereço.

```
#define N 4           /* quantidade de números a ordenar no vector seq */
#define P 6           /* quantidade de números a ordenar no vector seq */

int seq1 [N] = {10, 5, 6, 2};    /* vector (array) de N inteiros */
int seq2 [P] = {20, 7, 5, 10, 4, 8}; /* vector (array) de P inteiros */

main ()
{
    ordenaBolha (seq1, N);
    ordenaBolha (seq2, P);
}

void ordenaBolha (int * a, int n)
{
    /* função que ordena um vector */
    /* parametos a aponta para o 1.º elemento do vector */
    /* int * p;          /* apontador auxiliar, a usar apenas nesta função */
    /* int houveTroca;   /* indica se houve troca de números numa dada ronda */
    /* int i;            /* contador de elementos do vector */

    do
    {
        /* inicio do ciclo de ronda (pelo menos uma tem de fazer) */
        /* p = a;          /* inicializa o apontador auxiliar (1.º elemento do vector) */
        /* houveTroca = false; /* ainda não houve trocas nesta ronda */
        i = 0;           /* ainda não testou elemento nenhum */
        while (i < n-1)
        {
            /* ordena um dado número e o seguinte. houveTroca fica com true caso
               tenha havido troca dos dois números */
            houveTroca = houveTroca || ordena (&p, p+1);
            p = p + 1;
            /* passa ao número seguinte no vector */
            i = i + 1;
            /* mais um número do vector já testado */
            while (houveTroca);
            /* se houve trocas, tem de fazer mais rondas */
        }
    }
}
```

**Programa 5.4 - Variante do Programa 5.3, com uma alternativa à função ordenaBolha, usando apontadores em vez do acesso a um vector com índice (main e ordena não têm qualquer alteração)**

| Função em C                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | Rotina em Assembly                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>void ordenaBolha (int * a, int n) {     int * p;     int houveTroca;     int i;     do     {         p = a;         houveTroca = false;         i = 0;         while (i &lt; n-1)         {             /* inicio do ciclo de ronda (pelo menos uma tem de fazer) */             /* p = a;          /* inicializa o apontador auxiliar (1.º elemento do vector) */             /* houveTroca = false; /* ainda não houve trocas nesta ronda */             i = 0;           /* ainda não testou elemento nenhum */             while (i &lt; n-1)             {                 /* ordena um dado número e o seguinte. houveTroca fica com true caso                    tenha havido troca dos dois números */                 houveTroca = houveTroca    ordena (&amp;p, p+1);                 p = p + 1;                 /* passa ao número seguinte no vector */                 i = i + 1;                 /* mais um número do vector já testado */                 while (houveTroca);                 /* se houve trocas, tem de fazer mais rondas */             }         }     } }</pre> | <pre>ordenaBolha: ; a vem no R6. n vem no R7     ordenaBolha: ; a vem no R6. n vem no R7     int auxiliar; /* variável local usada para a troca de números */     if (*x &gt; *y) /* se o número corrente é maior do que o seguinte... */     {         auxiliar = *x; /* ...então troca-os, usando a variável auxiliar */         *x = *y;         *y = auxiliar;         return true; /* retorna verdadeiro se tiver havido troca */     }     else         return false; /* retorna falso se não tiver havido troca */     }      /* função ordenaBolha */     ordenaBolha:     ; p usa o R0     ; houveTroca usa o R8     int i;     do     {         p = a;         houveTroca = false;         i = 0;         while (i &lt; n-1)         {             iteração:             MOV R10, R7 ; cópia de n             SUB R10, 1 ; n-1             CMF R9, R10 ; i &lt; n-1 ?             JGE teste; ; se não, o ciclo acabou             MOV R1, R0 ; 1.º parâmetro (p)             MOV R2, R0 ; 2.º parâmetro             ADD R2, 2 ; p+1 (igual a R0+2)             CAL ordene ; chama a função             OR R8, R1 ; houveTroca    ordena             ADD R0, 2 ; p = p + 1;             ADD R9, 1 ; i = i + 1;             JMF iteração; próxima iteração             teste:             CMP R8, 0 ; houveTroca = falso ?             JNZ ordenaBolha ; mais uma ronda     }     RET ; retorna da função }</pre> |

**Tabela 5.9 - Rotina ordenaBolha do Programa 5.4, em C, e as instruções em linguagem assembly que lhe correspondem**

O uso de apontadores em linguagens de alto nível deve ser feito com cuidado, pois ignora a noção de estrutura de dados como um todo e não permite ao compilador fazer certas verificações, como por exemplo detecção de acesso fora dos limites do vector.

## 5.7.2 MECANISMO DE CHAMADA E RETORNO

### 5.7.2.1 ENDEREÇO DE RETORNO

Dado que o mesmo código (instruções de uma rotina) pode ser chamado a partir de vários pontos do programa, pretendendo-se que após a execução da rotina a execução continue na instrução seguinte à que chamou a rotina (tal como representado na Fig. 5.4), torna-se necessário implementar um mecanismo que suporte este regresso automático.

Obviamente, a própria rotina não pode terminar fazendo um salto para a instrução a que deve regressar, pois a rotina não sabe, a partida, de que instrução terá sido chamada e por conseguinte para que instrução deve retornar. Portanto, a única solução é guardar (memorizar num registo ou numa célula de memória) o endereço da instrução para que deve retornar na altura em que se chama a função.

Se só houver um nível de rotinas, isto é, se apenas o programa principal chamar rotinas, e uma rotina não chamar outra, basta apenas um registo (ou uma célula de memória) para memorizar o endereço de retorno. Esse registo/célula é necessário apenas enquanto a rotina estiver a ser executada pois, uma vez executado o seu retorno, o valor guardado no registo/célula deixa de ser necessário. Se ocorrer uma nova chamada de uma rotina, um novo valor de retorno é armazenado nesse registo/célula.

Este mecanismo tem de ser implementado pelo par de instruções de chamada e retorno, CALL e RET. É importante reconhecer que ambas são, na prática, instruções de salto. A instrução CALL salta para a rotina chamada, e RET salta para a instrução seguinte ao CALL, que chamou a rotina, tal como ilustrado pela Fig. 5.4. No entanto, são formas controladas de salto:

- A instrução CALL permite especificar para onde saltar (endereço da rotina a chamar), mas é mais elaborada do que uma simples instrução de salto, pois guarda o endereço de retorno (coisa que a instrução de salto não faz);
  - A instrução RET salta, mas é automática, saltando para o endereço guardado pela instrução CALL. Não é o programador que diz para onde deve saltar.
- Na realidade, o PEPE não tem apenas o par de instruções CALL/RET para tratar da chamada/retorno das rotinas. O PEPE suporta dois mecanismos de chamada de funções, dependendo da forma como o endereço de retorno é guardado, num registo ou na memória:
- Num registo – Existe um registo específico para memorizar o endereço de retorno, o RU (registo de ligação – ver Fig. 4.5, na página 204). As instruções de chamada e retorno para este mecanismo são CALRF e RETF, respectivamente (o “F” nas mnemónicas distingue este mecanismo do seguinte);
  - Na memória – O endereço de retorno é guardado em memória, numa tabela designada pilha, cujo funcionamento é descrito na secção 5.7.2.3. As instruções de chamada e retorno para este mecanismo são CALL e RET, respectivamente (sem o “F”).

Os dois mecanismos podem coexistir no mesmo programa mas um não é compatível com o outro. Uma rotina que termine com RETF só pode ser chamada com CALRF e uma que termine com RET só pode ser chamada com CALL. A secção 5.7.2.4 discute a razão de existência destes dois mecanismos e quando usar um ou outro.

Em dois aspectos fundamentais deste mecanismo é saber como é que se determina o endereço de retorno, ou seja, o endereço da instrução seguinte. Tal como indicado na secção 4.9, na página 215, quando uma instrução é executada já o PC tem o endereço da instrução seguinte, isto é, já foi incrementado de 2 unidades. Assim, para guardar o endereço de retorno numa instrução CALL ou CALRF basta simplesmente memorizar o valor do PC (pois este já conterá o endereço da instrução seguinte).

### 5.7.2.2 CHAMADA DE ROTINAS COM ENDEREÇO DE RETORNO EM REGISTRO

Tabela 5.10 ilustra este mecanismo com um programa muito simples, em que o programa principal chama a rotina abc duas vezes.

| ENDEREÇOS | PROGRAMA       | AÇÕES                                                          |
|-----------|----------------|----------------------------------------------------------------|
| 0000H     | PLACE 000H     | Localiza o código                                              |
| 0002H     | main: CALL abc | RU ← PC (endereço de retorno, 0002H – o da instrução seguinte) |
| 0004H     | instrução      | Executa esta instrução                                         |
| 0006H     | CALF abc       | RU ← PC (endereço de retorno, 0006H – o da instrução seguinte) |
| 0008H     | fim: JMP fim   | PC ← 0008H (chama rotina abc)                                  |
| 000AH     | abc: instrução | PC ← 0006H (sai da rotina abc)                                 |
| 000CH     | instrução      | Executa esta instrução                                         |
| 000EH     | RET            | Executa esta instrução                                         |

Tabela 5.10 - Mecanismo de chamada/retorno com o RU (registo de ligação)

Note-se que:

- Em cada instrução CALRF sucedem duas acções:
  - O endereço da instrução seguinte ao CALRF, o endereço de retorno, é guardado no registo RU (registo de ligação);
  - O endereço da primeira instrução da rotina chamada (abc, neste exemplo) é colocado no PC. Na prática, isto corresponde a um salto. A próxima instrução a ser lida da memória e a ser executada será precisamente a primeira instrução da rotina chamada.
- A instrução RETF coloca simplesmente no PC o valor do RU. Desta forma, executa um salto para a instrução no endereço de retorno, isto é, a instrução seguinte ao CALRF, que foi o que a instrução CALRF colocou no RU. Neste exemplo, o end-

reço de retorno será de 0002H ou de 0006H, consoante a chamada efectuada.

Assim, a instrução RETF sabe sempre para onde regressar.

Este mecanismo é muito simples e funciona bem, mas apenas enquanto a rotina abc não se lembrar de, também ela, chamar outra rotina. O problema que surge é que é preciso guardar um novo endereço de retorno no RL, mas a primeira rotina chamada (abc) ainda não regressou, pelo que o endereço de retorno guardado no RL ainda é preciso. Em resumo: o RL está ocupado.

A solução para a rotina abc poder chamar outra rotina, usando o mesmo mecanismo, é a seguinte:

1. Guardar o valor actual do RL noutra sítio (noutro registo ou, se estiverem todos ocupados, numa célula de memória);
2. Chamar a outra rotina, que há-de executar e regressar (usando o RL para guardar/recuperar o seu próprio endereço de retorno);
3. Recuperar o valor anterior do RL (o guardado no ponto 1) e colocá-lo em RL;
4. Executar o resto da rotina abc até chegar ao RETF, altura em que usará o RL para poder regressar à instrução seguinte à que chamou esta rotina.

| ENDERECOS       | PROGRAMA                                                       | ACÇÕES                                |
|-----------------|----------------------------------------------------------------|---------------------------------------|
| PLACE 0000H     | PLACE 0000H                                                    | Localiza o código                     |
| main: CALLF abc | RL ← PC (endereço de retorno, 0002H – instrução seguinte)      |                                       |
| 0002H           | PC ← PC (chama rotina abc)                                     | Executa esta instrução                |
| 0004H           | CALLF abc                                                      |                                       |
| 0006H           | fim: JRP fim                                                   | PC ← 0008H (sai da rotina abc)        |
| abc: instrução  | PC ← 0006H (sai para fim)                                      |                                       |
| 0008H           | b0V R1, RL                                                     | Executa esta instrução                |
| 000AH           | Guarda RL noutro sítio, pois vai ser usado pela CALLF seguinte |                                       |
| 000CH           | RL ← PC (endereço de retorno, 000EH – instrução seguinte)      |                                       |
| 000EH           | PC ← 0014H (chama rotina def)                                  | Executa esta instrução                |
| 0010H           | M0V RL, R11                                                    | Recupera RL anterior – 0002H ou 0006H |
| 0012H           | instrução RETF                                                 |                                       |
| def: instrução  | PC ← RL (sai para o endereço de retorno – 0002H ou 0006H)      | Executa esta instrução                |
| 0014H           | executa esta instrução                                         |                                       |
| 0016H           | RET                                                            |                                       |
| 0018H           |                                                                |                                       |

Tabela 5.11 - Mecanismo de chamada/retorno com o RL (registo de ligação) com mais do que um nível de chamada de rotinas

A Tabela 5.11 ilustra esta situação. O RL é guardado em R11 (que não deve ser alterado pela rotina def) e o seu valor recuperado após def retornar.

Este esquema resulta, mas é pouco geral (tudo depende da profundidade<sup>43</sup> de níveis de chamadas) e é propenso a erros de programação (o programador tem de controlar manualmente onde estão todos os RLs guardados à espera de serem recuperados).

A situação em que este mecanismo funciona melhor é na chamada a uma rotina que não chama outras, isto é, quando é a rotina final numa eventual cadeia de rotinas que vão chamando outras (o “\_” nas mnemónicas CALLF e RETF deriva da palavra “final”).

A Tabela 5.12 é obtida com base na Tabela 5.8, mas agora usando o mecanismo CALLF/RETf. As instruções relevantes para este mecanismo estão sublinhadas.

| PROGRAMA EM C                       | PROGRAMA EM ASSEMBLY                      |
|-------------------------------------|-------------------------------------------|
| #define N 4                         | N EQU 4 ; #define N 4                     |
| #define P 6                         | P EQU 6 ; #define P 6                     |
| int seq1[N] = {10, 5, 6, 2};        | PLACE 1000H ; localiza bloco de dados     |
| int seq2[P] = {20, 7, 5, 10, 4, 8}; | seq1: WORD 10 ; int seq[N]={10, 5, 6, 2}; |
|                                     | WORD 5                                    |
|                                     | WORD 6                                    |
|                                     | WORD 2                                    |
|                                     | seq2: WORD 20 ; int seq2[P] = {20, 7, 5,  |
|                                     | WORD 5                                    |
|                                     | 10, 4, 8};                                |
|                                     | WORD 10                                   |
|                                     | WORD 4                                    |
|                                     | WORD 8                                    |
| main () {                           | PLACE 0000H ; localiza bloco de código    |
| ordenaBolha (seq1, N);              | main: MOV R6, seq1 ; 1.º parâmetro        |
| }                                   | MOV R7, N ; 2.º parâmetro                 |
| ordenabolha (seq2, P);              | CALLF ordenaBolha ; chama a função        |
|                                     | MOV R6, seq2 ; 1.º parâmetro              |
|                                     | MOV R7, P ; 2.º parâmetro                 |
|                                     | CALLF ordenaBolha ; chama a função        |
| fim: JMP fim ; fim do programa      | fim: JMP fim ; fim do programa            |
| }                                   |                                           |
| void ordenaBolha (int a[], int n)   | ordenaBolha: ; a vem no R6. n vem no R7   |
| {                                   | int houveTroca ;                          |
| int i;                              | ; houveTroca usa o R8                     |
| do                                  | ; i usa o R9                              |
| houveTroca = false;                 | R10 é usado para valores temporários      |
| i = 0;                              | MOV R8, 0 ; houveTroca = false;           |
| while (i < n-1)                     | MOV R9, 0 ; i = 0;                        |
| ...                                 | iteração:                                 |
|                                     | MOV R10, R7 ; cópia de n                  |
|                                     | SUB R10, 1 ; n-1                          |
|                                     | CMP R9, R10 ; i < n-1 ?                   |
|                                     | JGE teste ; se não, o ciclo acabou        |

<sup>43</sup> Número de rotinas chamadas e ainda não regressadas, por terem sucessivamente invocado outras antes de regressarem.

| PROGRAMA EM C                                                                                                                                                                                                                                                                                                                         | PROGRAMA EM ASSEMBLY                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>    ordena (ka[i], &amp;a[i+1]);     houveTroca = houveTroca;     i = 1 + 1; }  while (houveTroca) {     int ordena (int * x, int * y)     {         int auxiliar;         if (*x &gt; *y)             auxiliar = *x;         *x = *y;         *y = auxiliar;         return true;     }     else         return false; } </pre> | <pre> MOV R10, R9 ; obtém cópia de i SHL R10, 1 ; 2*i (ender. em bytes) ADD R10, R6 ; endereço de a [i] MOV RL, R10 ; 1.º parâm. (ka[i]) MOV R10, R9 ; obtém cópia de i ADD R10, 1 ; i+1 SHL R10, 1 ; 2*(i+1) ADD R10, R6 ; endereço de a [i+1] MOV R2, R10 ; 2.º parâm. (&amp;a[i+1]) MOV R0, RL ; guarda RL CALLF ordena ; chama a função MOV RL, R0 ; recupera RL OR R8, RL ; houveTroca    ordena ADD R9, 1 ; i = i + 1 JMP iteração ; proxima iteração teste: CMP R8, 0 ; houveTroca = falso ? JNZ ordenaBolha ; mais uma ronda RETF ; retorna da função ordena: ; x vem em RL, y vem em R2 ; R3 e R4 são registos temporários ; auxiliar usa o R5 MOV R3, [R1] ; *x MOV R4, [R2] ; *y CMP R3, R4 ; (*x &gt; *y) ? JLE else ; se não, executa o else MOV R5, R3 ; auxiliar = *x; MOV [RL], R4 ; *x = *y; MOV [R2], R5 ; *y = auxiliar; MOV RL, 1 ; true RETF ; return true; else: MOV RL, 0 ; false RETF ; return false; ; o valor retornado está no RL </pre> |

Tabela 5.12 - Programa 5.3 e as instruções em linguagem assembly que lhe correspondem, usando as instruções CALLF e RETF e o RL (registo de ligação). As instruções sublinhadas são as relevantes para o mecanismo de chamada/retorno

É de notar que:

- Na rotina ordenaBolha se usou o RL para guardar o RL enquanto se chama a rotina ordena. Era o único registo geral ainda disponível;
- Se em vez do Programa 5.3 se tivesse usado a variante com apontadores da rotina ordenaBolha, no Programa 5.4, nem o RL estaria disponível, pois este é usado para conter o apontador p. Neste caso, o RL não poderia ser guardado num registo, pois todos os registos gerais estariam ocupados, e teria de ser guardado numa célula de memória. Mas tal seria também problemático, pois para aceder à memória precisa-se, na prática, de um registo para indicar qual o endereço de memória a aceder (é não havendo nenhum registo geral disponível). A secção 5.7.2.3 mostra como resolver este problema.

Esta simulação ilustra o funcionamento do programa em linguagem assembly da Tabela 5.12. Os aspectos cobertos incluem os seguintes:

- Execução com pontos de paragem;
- Verificação do funcionamento das instruções CALLF e RETF;
- Verificação da evolução dos registos relevantes (RL e PC em particular).

### 5.7.2.3 CHAMADA DE ROTINAS COM ENDEREÇO DE RETORNO NA MEMÓRIA (PILHA)

A secção anterior mostrou como era possível chamar uma rotina e depois regressar, guardando o endereço de retorno. A chamada de rotinas é muito frequente nos programas, é vulgar uma rotina A chamar outra B, que por sua vez chama outra C, e assim sucessivamente.

Normalmente não há registos que cheguem para conseguir guardar vários endereços de retorno, correspondentes às rotinas pendentes (chamadas mas ainda não regressadas), pelo que nestes casos o mais vulgar é os sucessivos valores do endereço de retorno serem guardados em memória.

Este é um modelo diferente do que usa o RL (a secção 5.7.2.4 explica como os dois modelos podem ser combinados), em que a instrução de chamada guarda o endereço de retorno numa tabela e a instrução de retorno recupera o endereço para onde retornar a partir dessa tabela. Essa tabela tem um funcionamento específico (descrito a seguir) e designa-se pilha (stack).

A Tabela 5.12 mostrou como é que o Programa 5.3 se pode implementar com o mecanismo baseado no RL, com as instruções CALL e RET. A Tabela 5.13 explica as diferenças entre as duas tabelas. Basicamente, no caso da pilha (Tabela 5.8) não é preciso guardar o RL porque as instruções CALL e RET tratam de guardar e recuperar o endereço de retorno na pilha, de forma automática.

| ROTINA      | COM RL (TABELA 5.12)                                                                      | COM PILHA (TABELA 5.8)                                  |
|-------------|-------------------------------------------------------------------------------------------|---------------------------------------------------------|
| main        | CALL ordenaBolha ; chama a função                                                         | CALL ordenaBolha ; chama a função                       |
| ordenaBolha | MOV RL, RL ; guarda RL<br>CALL ordena ; chama a função<br>MOV RL, RL ; recupera RL<br>RET | CALL ordena ; chama a função<br>RET ; retorna da função |

Tabela 5.13 - Diferenças do mecanismo de chamada/retorno com RL (Tabela 5.12) e com pilha (Tabela 5.8) na implementação em assembly do Programa 5.3

A Tabela 5.14 compara os dois modelos de guarda de endereço de retorno, em registo RL e em memória (pilha).

| TOPICO                           | ENDERECO DE RETORNO POR REGISTRO (RL)                                                  | ENDERECO DE RETORNO POR MEMÓRIA (PILHA)                                                   |
|----------------------------------|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| Instruções                       | CALL, RET                                                                              | CALL, RET                                                                                 |
| Chamada                          | RL $\leftarrow$ PC (endereço de retorno)<br>PC $\leftarrow$ endereço da rotina chamada | Pilha $\leftarrow$ PC (endereço de retorno)<br>PC $\leftarrow$ endereço da rotina chamada |
| Retorno                          | PC $\leftarrow$ RL                                                                     | PC $\leftarrow$ endereço de retorno guardado na pilha                                     |
| Supõe rotinas que invocam outras | Directamente não (só guardando o RL noutro sítio)                                      | Sim                                                                                       |

Tabela 5.14 - Comparação entre os mecanismos de chamada/retorno de rotinas com endereço de retorno guardado em registo (RL) e em memória (pilha)

Para perceber como é que a pilha está organizada, podemos fazer uma analogia com a receção de um supermercado onde os clientes, antes de entrarem, têm de deixar quaisquer sacos que tenham consigo. A Tabela 5.15 faz a comparação com o mecanismo de chamada/retorno das rotinas.

| CLIENTE NO SUPERMERCADO                                                                                  | CHAMADA/RETORNO DE ROTINAS                                                                                                                      |
|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| • Quando entra – Deixa os sacos na receção, recabendo uma chapinha numerada                              | • Quando chama uma rotina – Guarda o endereço de retorno numa das células de memória da pilha e salta para lá.                                  |
| • Quando sai – Entrega a chapinha e recebe os seus sacos de volta                                        | • Quando retorna – Recupera o endereço de retorno e salta para lá.                                                                              |
| • Clientes podem entrar e sair por qualquer ordem (por isso, precisam da chapinha para se identificarem) | • A ordem de chamada/retorno é estritamente LIFO (Last In, First Out) – A última rotina a ser chamada é a primeira a ser retornada.             |
| • Sacos são guardados em caixas ou bengaleiros, de acesso por ordem aleatória ou                         | • Endereços de retorno são guardados numa pilha, com acesso apenas pelo topo. O que está no topo é sempre o da rotina que está a ser executada. |
| • Quando um cliente entrega os sacos, guardam-se em qualquer caixa que esteja livre                      | • Quando uma rotina é chamada, o endereço de retorno é sempre colocado no topo da pilha                                                         |
| • Quando o cliente quer recuperar os sacos, a chapinha identifica o caixão onde eles estão               | • Quando uma rotina retorna, o seu endereço de retorno está sempre no topo da pilha                                                             |

Tabela 5.15 - Analogia e diferenças entre o funcionamento da receção de um supermercado e o mecanismos de chamada/retorno de rotinas

No supermercado, cada cliente recebe uma chapinha que lhe permite identificar os seus sacos. Isto é necessário porque os clientes podem entregar as chapinhas (para recuperar os seus sacos) por uma ordem totalmente diferente daquela em que entregaram os sacos na receção, pelo que tem de haver um meio de identificar quais os sacos pretendidos.<sup>44</sup>

<sup>44</sup> Esta analogia do supermercado é mais adequada a outra estrutura de dados, também importante, designada montão, descrito na secção 5.8.5, na página 378.

No caso das rotinas, a situação é diferente. A ordem de retorno é exactamente a inversa da ordem de chamada, isto é, a última rotina a ser chamada é a primeira a ser retornada. Por isso, os endereços de retorno podem ser "empilhados" como se fossem cartões com o endereço de retorno escrito:

- Cada rotina que é chamada coloca o seu endereço de retorno no topo da pilha;
- Quando essa rotina retorna, obtém o seu endereço de retorno simplesmente retirando o cartão que nessa altura estiver no topo da pilha.



Pilha vazia

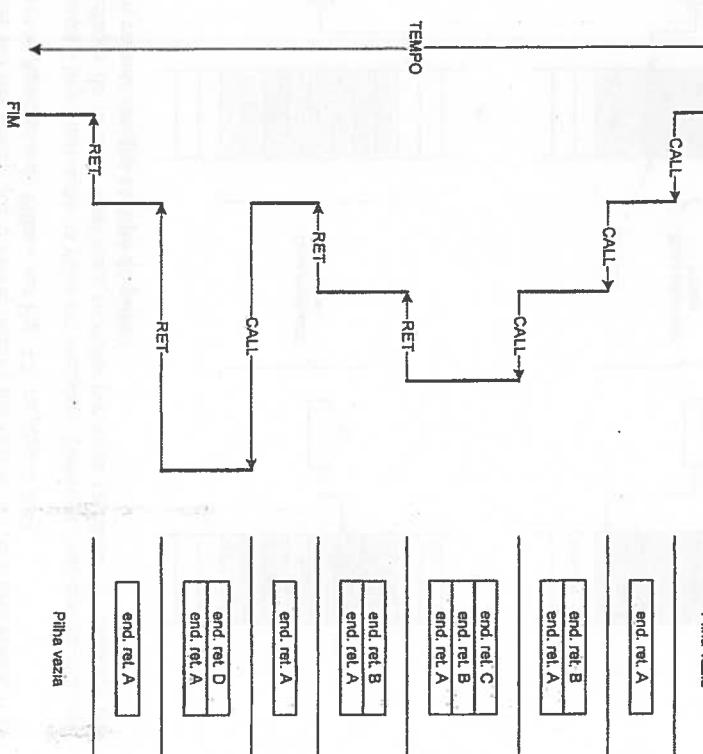


Fig. 5.5 - Mecanismo de chamada/retorno de rotinas com endereço de retorno guardado em memória (na pilha), com a evolução do estado da pilha ao longo do tempo. As relações de chamada estão aparentes na Fig. 5.5. A Tabela 5.17 apresenta um programa com esta estrutura de rotinas.

A Fig. 5.5 ilustra este processo de forma gráfica, representando a execução de um programa principal e quatro rotinas, A, B, C e D ao longo do tempo. As relações de chamada estão aparentes na Fig. 5.5. A Tabela 5.17 apresenta um programa com esta estrutura de rotinas.

Do lado esquerdo, notam-se as instruções a serem executadas ao longo do tempo, de cima para baixo, com destaque para as instruções de chamada (CALL) e retorno (RET). Quando uma rotina chama outra, a sua execução é suspensa até a rotina chamada retornar. A semântica do retorno implica que a rotina que é retornada é sempre a que está em execução, ou seja, a que foi chamada mais recentemente e portanto está há menos tempo em execução. Não se pode estar a executar uma rotina e retornar-se de outra qualquer. A ordem de chamada/retorno utiliza uma política LIFO (Last In, First Out – a última a entrar é a primeira a sair ou, melhor dizendo, a última a colocar o seu endereço de retorno na pilha é a primeira a retirá-lo).

No lado direito, representa-se o estado da pilha entre instruções de chamada/retorno, em que se pode verificar precisamente esta política:

- A pilha começa por estar vazia, quando ainda não foi chamada nenhuma rotina;
- A rotina A é a primeira a colocar o endereço de retorno, quando é chamada, seguindo-se as rotinas B e C, por esta ordem, altura em que a pilha está na sua profundidade (número de endereços de retorno memorizados) máxima;
- A rotina C é a primeira a retornar (pois não chama nenhuma outra), seguindo-se a rotina B, altura em que na pilha volta a estar apenas o endereço de retorno da rotina A, que ainda não retornou;
- A rotina A invoca ainda a rotina D, aumentando de novo a profundidade da pilha. Note-se que as posições na pilha são reutilizadas sempre que a profundidade volta a subir. O endereço de retorno da rotina D está agora precisamente na mesma posição que antes era ocupada pelo endereço de retorno da rotina B (que agora já não é necessário);

Finalmente, a rotina D retorna, seguida do retorno de A e a pilha volta ficar vazia;

- Cada instrução CALL (chamada) acrescenta um endereço de retorno no topo da pilha;
- Cada instrução RET (retorno) retira um endereço de retorno da pilha;
- Na pilha, os endereços de retorno mais antigos estão por baixo. No topo, está sempre o endereço de retorno da rotina de invocação mais recente, precisamente a que está em execução.

A pilha é assim uma tabela dinâmica, cujo número de elementos (profundidade da pilha) aumenta e diminui ao longo da execução do programa. A pilha é implementada:

- Reservando em memória um conjunto de células para este fim. Note-se que a dimensão da tabela é fixa. As células de memória têm de estar reservadas para a pilha, mesmo que não sejam usadas. O número de células ocupadas (as que contêm um endereço de retorno válido) é que vai variando. Quantas células devem ser reservadas? Depende da profundidade máxima esperada, e cabe ao programador garantir que essa profundidade não é ultrapassada, senão alguns endereços de retorno podem ser escritos fora da área reservada para a pilha, desvirtuando o conteúdo de zonas de memória adjacentes à zona reservada para a pilha;

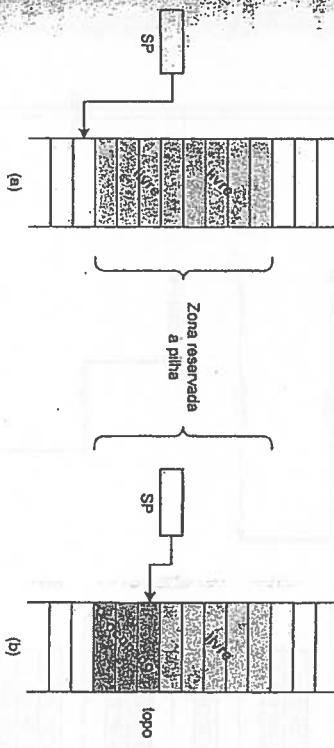


Fig. 5.6 - Implementação da pilha, em várias situações: (a) – Pilha vazia; (b) – Pilha com três endereços de retorno; (c) – Pilha com seis endereços de retorno; (d) – Pilha cheia

As células a cinzento são as reservadas para a pilha. As células brancas são células adjacentes à pilha e podem conter valores de variáveis, instruções ou estarem livres (não dependem do programa e de como o programador localizou as diversas zonas do programa no espaço de endereços).

Usando um registo que mantenha o endereço da célula de memória em que se situa o topo da pilha. Este registo irá variar consonante o número de endereços de retorno armazenados na pilha aumente ou diminua (isto é, à medida que sejam executadas instruções CALL e RET, respectivamente).

Na realidade, estas operações de guardar/recuperar endereços de retorno em memória acabam por ser tão frequentes que o PEPE reserva um registo só para este efeito: o SP – Stack Pointer, ou Apontador de Pilha – ver Fig. 4.5, na página 204).

Fig. 5.6 ilustra a implementação da pilha em memória. Parece-se fisicamente com uma pilha de endereços de retorno, com mais recentes por cima (a célula de memória que contém o mais recente designa-se topo da pilha).

Dentro da pilha, as células mais claras estão livres (estão reservadas para uso da pilha mas não contêm nenhum endereço de retorno válido), enquanto as mais escuas indicam endereços de retorno correspondentes a rotinas chamadas e ainda não retornadas. Naturalmente, as zonas a cíntezo são apenas uma convenção para facilitar a explicação. Do ponto de vista do computador, todas as células de memória são iguais.

O registo SP aponta para (contém o endereço de) a célula de memória onde está o topo da pilha (o endereço de retorno mais recente, o da rotina em execução). Dentro das células da pilha:

- Acima do topo, todas as células da pilha estão livres;
- O topo e todas as células abaixo estão ocupados.

Note-se que "acima" quer dizer endereços menores e "abaixo" quer dizer endereços maiores (ver Fig. 4.18 na página 259, por exemplo).

Obviamente, o registo SP tem de ser inicializado pelo programa principal antes de qualquer rotina ser chamada. Tal como se pode ver na Fig. 5.6a, o valor inicial de SP deve ser o endereço imediatamente abaixo na figura (isto é, mais 2) em relação ao da primeira célula da pilha. É importante não esquecer que a pilha memoriza endereços, que são de 16 bits. Logo, os valores de SP devem ser sempre pares.

A Fig. 5.6 descreve quatro situações:

- Pilha vazia. Todas as células da pilha estão livres. Não há endereços de retorno válidos armazenados na pilha. O SP aponta para a célula imediatamente abaixo à primeira célula da pilha, o que quer dizer que na realidade nem existe topo. Esta é a situação que se verifica quando se está a executar o programa principal, sem nenhuma rotina chamada;
  - A pilha tem três células ocupadas, o que quer dizer que nesta altura há três rotinas chamadas e ainda não regressadas. A instrução em execução pertence à rotina cujo endereço de retorno está no topo da pilha. O SP contém o endereço do topo;
  - Situação idêntica à (b), mas em que há seis endereços de retorno armazenados (a profundidade da pilha é maior);
  - A pilha está cheia, isto é, o topo está já a ocupar a última célula reservada para a pilha. Se a rotina em execução chamar outra, a célula de memória imediatamente acima da pilha será escrita com o endereço de retorno dessa chamada, podendo assim destruir informação válida. Tem de se reservar espaço suficiente para a pilha de modo a que isto não suceda.
- A Tabela 5.16 apresenta a descrição das operações elementares (em RTL) das instruções de chamada/retorno que usam a pilha (CALL e RET). Note-se que:
- No CALL, o acesso à memória é feito com SP-2, mas o SP só é alterado após acesso. Funcionalmente é equivalente, mas esta ordem é importante por causa do mecanismo de memória virtual (ver comentário à Tabela 6.15, na página 480);

| OPERAÇÃO | SINTAXE        | ACÇÕES (RTL)                                        | COMENTÁRIOS                                                                                                   |
|----------|----------------|-----------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| Chamada  | CALL constante | Mw[SP-2] ← PC<br>SP ← SP - 2<br>PC ← PC + constante | Guarda endereço de retorno<br>Passa à próxima posição livre da pilha<br>Endereço (relativo) da rotina chamada |
|          | CALL Rs        | Mw[SP-2] ← PC<br>SP ← SP - 2<br>PC ← Rs             | Guarda endereço de retorno<br>Passa à próxima posição livre da pilha<br>Endereço (absoluto) da rotina chamada |
| Retorno  | RET            | PC ← Mw[SP]<br>SP ← SP + 2                          | Recupera endereço de retorno<br>Volta à posição anterior da pilha                                             |

Tabela 5.16 - Instruções de chamada/retorno de rotinas com endereço de retorno na pilha

As duas formas de CALL diferem apenas na forma como se especifica o endereço da rotina chamada, de forma:

- Relativa, com uma constante que se soma ao valor do PC. Esta constante pode ter até 12 bits, pois 4 bits são gastos com o opcode da instrução CALL (embora nas instruções em linguagem assembly se especifique uma etiqueta a seguir à numérica da instrução CALL, o que o assembler coloca na instrução máquina correspondente é um valor que permite atingir o valor da etiqueta a partir do valor do PC e não o valor da etiqueta em si);
  - Absoluta, referindo um registo que deverá conter o endereço da rotina chamada. O Programa 5.13, na página 377, dá um exemplo.
- Dado que o registo SP aponta normalmente para o topo da pilha (ultimo endereço de retorno lá guardado), a operação de "guardar endereço de retorno na pilha" efectuada nas instruções CALL envolve (por esta ordem):
- Ajustar o SP para apontar para a posição seguinte ao topo (que assim passa a ser ela o novo topo);
  - Guardar o endereço de retorno nessa posição.
  - A instrução RET usa a ordem inversa, recuperando primeiro o endereço de retorno e repondo o valor anterior do SP depois;
  - A instrução RET não sabe qual a forma de CALL usada, nem depende dessa informação;
  - A inicialização do PC quando o processador arranca é automática (o PC é posto a 0000H, endereço em que a execução começa);
  - O SP tem de ser inicializado explicitamente pelo programador, tipicamente através de um MOV.

A Tabela 5.17 apresenta um programa com a estrutura de rotinas que já foi ilustrada pela Fig. 5.5. Estão também representados os endereços em memória das várias instruções.

| ENDEREÇO | PROGRAMA                                                                                                                                          |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| 0000H    | apósPilha EQU 1000H ; o endereço logo após a pilha (porque a pilha principal: MOV SP, apósPilha ; inicializa SP com o endereço logo após a pilha) |
| 0002H    | MOV R0, 0 ; chama a rotina A                                                                                                                      |
| 0004H    | CALL A                                                                                                                                            |
| 0006H    | MOV RL, 1 ; fim do programa                                                                                                                       |
| 0008H    | fin: Jmp fin                                                                                                                                      |
| A: 000AH | MOV R2, 2 ; chama a rotina B                                                                                                                      |
| 000CH    | CALL B                                                                                                                                            |
| 000EH    | MOV R3, 3 ; chama a rotina D                                                                                                                      |
| 0010H    | CALL D                                                                                                                                            |
| 0012H    | MOV R4, 4 ; retorna da rotina A                                                                                                                   |
| 0014H    | RET                                                                                                                                               |
| B: 0016H | MOV R5, 5 ; chama a rotina C                                                                                                                      |
| 0018H    | CALL C                                                                                                                                            |
| 001AH    | MOV R6, 6 ; retorna da rotina B                                                                                                                   |
| 001CH    | RET                                                                                                                                               |
| C: 001EH | MOV R7, 7 ; retorna da rotina C                                                                                                                   |
| 0020H    | MOV R8, 8                                                                                                                                         |
| 0022H    | RET                                                                                                                                               |
| D: 0024H | MOV R9, 9 ; retorna da rotina D                                                                                                                   |
| 0026H    | MOV R10, 10                                                                                                                                       |
| 0028H    | RET                                                                                                                                               |

Tabela 5.17 - Programa com quatro rotinas, útil apenas para ilustrar o mecanismo de chamada/retorno. Este é o programa que serviu de base à Fig. 5.5

Note-se a inicialização do SP, logo no início. O endereço escolhido (1000H, neste caso), tem de contemplar:

- O facto de a pilha crescer "para trás" nos endereços, ou seja, um CALL faz diminuir o SP. Logo, o SP tem de ser inicializado no fim do espaço de memória reservado para a pilha (mais concretamente, no endereço logo após o espaço da pilha, pois o CALL decrementa primeiro o SP e só depois coloca o endereço de retorno na pilha);
- Espaço suficiente para a pilha, assumindo-se que as N palavras ( $2^N$  bytes) anteriores a 1000H estão reservadas para uso exclusivo da pilha (sendo N a profundidade máxima esperada para a pilha);
- A existência real de memória em toda esta gama de endereços. O computador pode não ter memória em todo o seu espaço de endereçamento, e por isso tem de se verificar que toda a gama de endereços atribuídos à pilha está dentro da gama de endereços com memória.

## NOTA

Dado que o espaço reservado para a pilha não precisa de ser inicializado (só o SP, pois a pilha em si é considerada inicialmente vazia), é possível simplesmente inicializar o SP e assumir que as N palavras anteriores podem ser usadas para a pilha, desde que essa gama de endereços não se sobreponha (nem que seja apenas parcialmente) com nenhum outro bloco de dados ou de instruções (é uma questão de verificar as directivas PLACE e a dimensão de cada um dos blocos de dados e de instruções).

Esta solução é a usada no programa da Tabela 5.17 apenas por uma questão de simplicidade. No entanto, a sua correção está dependente da atenção do programador. A solução seguinte é mais segura, pois usa as directivas do assembler para fazer a reserva do espaço para a pilha. Se houver qualquer sobreposição entre blocos de dados efetuados, o assembler gera uma mensagem de aviso.

tampilha EQU 100H ; tamanho da pilha (em palavras)

PLACE pilha: TABLE tampilha ; localiza bloco de dados  
apósPilha: WORD 0000H ; reserva espaço para a pilha (tampilha palavras)  
var1: WORD 0000H ; eventual variável a seguir à pilha

PLACE principal: MOV SP, apósPilha ; inicializa SP com o endereço logo após a pilha  
MOV R0, 0 ; chama a rotina A  
CALL A  
MOV RL, 1 ; fim do programa

fin: Jmp fin ; resto do programa

Neste exemplo, a zona de memória reservada para utilização da pilha está entre os endereços 1000H e 11FEH. apósPilha, o endereço imediatamente após a pilha, é 1200H, pois tampilha é o número de palavras da pilha e não a sua dimensão em bytes. A directiva TABLE reserva 100H palavras, ou 200H bytes.

Se uma variável for declarada logo a seguir à pilha, tal como var1, fica localizada no endereço 1200H. SP é inicializado com este valor, mas a primeira instrução de CALL decrementa primeiro o SP (em 2 unidades) e só depois escreve na pilha. Assim, o princípio endereço de retorno será guardado na pilha no endereço 11FEH (1200H - 2).

A Tabela 5.17 corresponde à estrutura estática do programa. A Tabela 5.18 contém a estrutura dinâmica, isto é, a sequência de instruções que são executadas ao longo do tempo, e ainda a evolução dos registos mais relevantes (PC e SP) e das palavras da pilha que são usadas.

A tabela é muito importante em termos de ilustração do funcionamento do mecanismo de chamada/retorno das rotinas e deve ser percebida em detalhe. Os aspectos seguintes são os mais importantes:

- A coluna "Endereço" indica o endereço em que cada instrução reside em memória. Ao contrário do que acontece na Tabela 5.17, os endereços não estão ordenados por ordem crescente, mas sim por ordem de execução;
- A coluna "Programa" lista as instruções pela ordem com que são executadas. Por simplicidade, os comentários são omitidos. As etiquetas e a indentação das ins-

instruções ajudam a manter a perspectiva da estrutura do programa original. A instrução com a etiqueta `fim` é executada indefinidamente até o processador ser reinitializado;

| ENDERECO | PROGRAMA          | PC     | SP    | OFFH  | OFFCH | OFFFAH |
|----------|-------------------|--------|-------|-------|-------|--------|
|          | estado inicial    | 0000H  | xxxx  | xxxx  | xxxx  | xxxx   |
| 0000H    | principal:        | 0000H  | xxxx  | xxxx  | xxxx  | xxxx   |
| 0002H    | MOV SP, apóspilha | 0002H  | 1000H | xxxx  | xxxx  | xxxx   |
| 0004H    | MOV R0, 0         | 0004H  | 0004H | 0FFEH | 0006H | 0006H  |
| 000AH    | CALL A            | 000CH  | 0005H | 0FFCH | 0006H | 0006H  |
| A:       | MOV R2, 2         | 000EH  | 0018H | 0FFAH | 0006H | 0006H  |
| 000CH    | CALL B            | 000CH  | 001AH | 0FFAH | 0006H | 0006H  |
| 0016H    | MOV R5, 5         | 000EH  | 001AH | 0FFAH | 0006H | 0006H  |
| 0018H    | CALL C            | 000CH  | 001AH | 0FFAH | 0006H | 0006H  |
| C:       | MOV R7, 7         | 0002H  | 002AH | 0FFCH | 0006H | 0006H  |
| 001EH    | MOV R8, 8         | 0002H  | 002AH | 0FFCH | 0006H | 0006H  |
| 0020H    | RET               | 0002H  | 002AH | 0FFCH | 0006H | 0006H  |
| 0022H    | RET               | 0001CH | 0FFEH | 0006H | 0006H | 0006H  |
| 001AH    | MOV R6, 6         | 0001CH | 0FFEH | 0006H | 0006H | 0006H  |
| 001CH    | RET               | 0001CH | 0FFEH | 0006H | 0006H | 0006H  |
| 000EH    | MOV R3, 3         | 0010H  | 0012H | 0FFCH | 0006H | 0006H  |
| 0010H    | CALL D            | 0010H  | 0012H | 0FFCH | 0006H | 0006H  |
| 0024H    | MOV R9, 9         | 0026H  | 0028H | 0FFCH | 0006H | 0006H  |
| 0026H    | RET               | 0026H  | 0028H | 0FFCH | 0006H | 0006H  |
| 0028H    | RET               | 002AH  | 0FFEH | 0006H | 0006H | 0006H  |
| 0012H    | MOV R4, 4         | 0014H  | 0016H | 0FFCH | 0006H | 0006H  |
| 0014H    | RET               | 0014H  | 0016H | 0FFCH | 0006H | 0006H  |
| 0006H    | MOV R1, 1         | 0008H  | 0008H | 0FFCH | 0006H | 0006H  |
| 0008H    | JMP fim           | 0008H  | 000AH | 0FFCH | 0006H | 0006H  |
| 0008H    | fim:              | 000AH  | 000AH | 0FFCH | 0006H | 0006H  |
| ...      | ...               | ...    | ...   | ...   | ...   | ...    |

Tabela 5.18 - Evolução do estado de execução do programa da Tabela 5.17. Por motivos de clareza, nas quatro colunas da direita omitimos os valores nas linhas em que a pilha ou o SP não estão envolvidos

- A coluna "PC" indica o valor que o PC tem no início de cada instrução, e que é sempre 2 unidades acima do endereço em que a instrução reside. Ou seja, quando uma instrução começa a ser executada já o PC foi actualizado e preparado para ir buscar a próxima instrução;
- Nas colunas seguintes, os valores representados são os que ficam após a execução de cada instrução. Para ser mais perceptível, são indicados os valores apenas nas instruções de chamada e retorno. Nas restantes, os valores mantêm-se iguais aos valores anteriores. O valor "xxxx" significa "valor não inicializado", sendo portanto indeterminado;
- A coluna "SP" indica o valor com que o registo SP fica após uma instrução de CALL ou de RET. Note-se como desce após os CALLs e sobe após os RETs, voltando no fim ao valor original. Tem sempre um valor par, pois contém endereços de palavras (na pilha);

- As três colunas da direita representam as três palavras da pilha usadas. O seu valor começa por ser indeterminado, sendo inicializado pela instrução CALL. Só ao fim de três CALLs consecutivos (sem retorno) as três palavras são usadas. As zonas a cinzento indicam a gama de instruções em que cada palavra da pilha é válida (não inclui as instruções de RET porque no fim dessas instruções já a pilha da pilha não é válida). Note-se:
- O paralelismo das zonas a cinzento com a Fig. 5.5;

- Que endereços de retorno na pilha são sempre os da instrução seguinte ao CALL que chamou uma dada rotina, tal como indicado pela coluna "PC";

Que as instruções de retorno (RET) não retiram fisicamente o endereço de retorno da pilha. Apenas o copiam para o PC e actualizam o SP (soma-n-lhe 2 unidades), o que faz com que a palavra da pilha onde estava o endereço de retorno seja declarada como estando livre. O endereço de retorno armazenado nessa palavra só será destruído (por um novo endereço de retorno) se houver um novo CALL, como sucede na instrução CALL D, no endereço 0010H.

#### SIMULAÇÕES – CHAMADA DE ROTINAS COM A PILHA

Esta simulação ilustra o mecanismo subjacente às instruções CALL e RET, com base em dois exemplos:

- programa da Tabela 5.17, tendo em conta a Tabela 5.18;
- programa da Tabela 5.8, que mostra como se pode implementar o Programa 5.3 (ordenação de números com o método por bolha) em linguagem assembly.

Os aspectos cobertos por esta simulação incluem os seguintes:

- Execução com pontos de paragem;
- Verificação da evolução dos registos relevantes (SP e PC em particular);
- Verificação do funcionamento das instruções CALL e RET;
- Verificação do funcionamento da pilha.

#### 5.7.2.4 QUAL DOS MECANISMOS DE CHAMADA DE ROTINAS SE DEVE USAR?

A Tabela 5.14 já comparou os dois mecanismos de chamada/retorno, usando o registo de ligação (RL) ou a pilha para guardar o endereço de retorno. As secções 5.7.2.2 e 5.7.2.3, respectivamente, já detalharam cada um dos mecanismos.

Claramente, a pilha é muito mais geral do que o RL, funcionando bem em qualquer caso e não apenas no caso das rotinas que não chamam outras. Pode então perguntar-se porque é que havemos de ter o mecanismo com RL e não usarmos exclusivamente a pilha. A resposta é muito simples: desempenho (rápidez de processamento).

O problema da área de Arquitectura de Computadores não é propriamente a funcionalidade. A programação é tão flexível que os processadores conseguem fazer quase tudo o

que quisermos, com mais ou menos instruções. O grande problema é velocidade de processamento. Os processadores executam milhões e milhões de operações elementares para conseguir implementar funcionalidades complexas, e por conseguinte devem executar essas operações o mais rápido possível. O cerne da área de Arquitectura de Computadores é conseguir a organização interna dos computadores que lhes permita maximizar o seu desempenho (para uma dada frequência de relógio).

O mecanismo de chamada/retorno com o RL é interessante porque não envolve acessos à memória (no caso de rotinas que não chamam outras). Um acesso a um registo é sempre mais rápido do que um acesso à memória. O problema é se a rotina chamada quiser chamar outra. Neste caso, podemos usar uma das seguintes soluções:

1. Guardar o RL noutro registo antes de chamar a outra rotina e recuperá-lo depois do retorno dessa rotina (ver exemplo na Tabela 5.1, na página 318). Também muito rápido, este esquema é muito pouco flexível e nem sempre praticável, já que os registos são poucos e portanto tendem a estar sempre ocupados;
2. Guardar o RL na pilha (a secção 5.7.3.1 explica como isto é possível). Solução bastante flexível, pois o esquema pode ser repetido várias vezes, pelo que rotinas que chamam outras são facilmente suportadas, mas perde-se a vantagem do não acesso à memória;
3. Não usar CALL e RET (as instruções que usam o RL) para chamar rotinas que chamem outras, usando CALL e RET nestes casos.

Cabe ao programador de linguagem assembly (ou, mais frequentemente, ao compilador) escolher em cada caso qual a melhor das soluções. Estas podem coexistir facilmente no mesmo programa, mas é imprescindível usar a mesma solução na chamada e no retorno de uma dada rotina.

Como o mecanismo da pilha é o mais fácil de usar e o mais geral, o programador de assembly tende a usá-la. O mecanismo com RL é tipicamente uma optimização reservada para os compiladores, que podem fazer uma gestão automática dos registos e decidir em cada caso qual a melhor solução (algo muito difícil de fazer manualmente).

Uma optimização óbvia é usar o mecanismo CALL-RET apenas nas rotinas que não invoquem outras. Mesmo assim, ainda é preciso ter outros factores em conta, como a alocação de variáveis locais às funções, ferramentas de desenvolvimento que envolvam inspecção da estrutura de dados do programa (depuradores, por exemplo), etc.

Em alto nível (linguagem C, por exemplo), este problema não se coloca. A semântica da chamada e de retorno das funções está bem definida mas não específica como o retorno é feito, nem onde é que a informação necessária para o retorno é guardada. É um problema da implementação da linguagem. Nuns computadores pode ser feito de uma maneira, noutras computadores poderá ser feito de forma diferente. O compilador está sempre adequado ao computador onde o programa irá correr e escolherá qual a melhor implementação para esse computador e para esse programa.

**5.7.2.5 VARIANTES DO FUNCIONAMENTO DA PILHA**  
 O funcionamento da pilha descrito até aqui pelas instruções CALL e RET (Tabela 5.16) e pela Fig. 5.6 pode sofrer variantes, referidas nesta secção apenas porque nem todos os processadores implementam este mecanismo exactamente da mesma maneira. A Fig. 5.7 apresenta uma das variantes em várias situações (e-h), ao mesmo tempo que inclui a variante da Fig. 5.6-a-d para servir de termo de comparação.

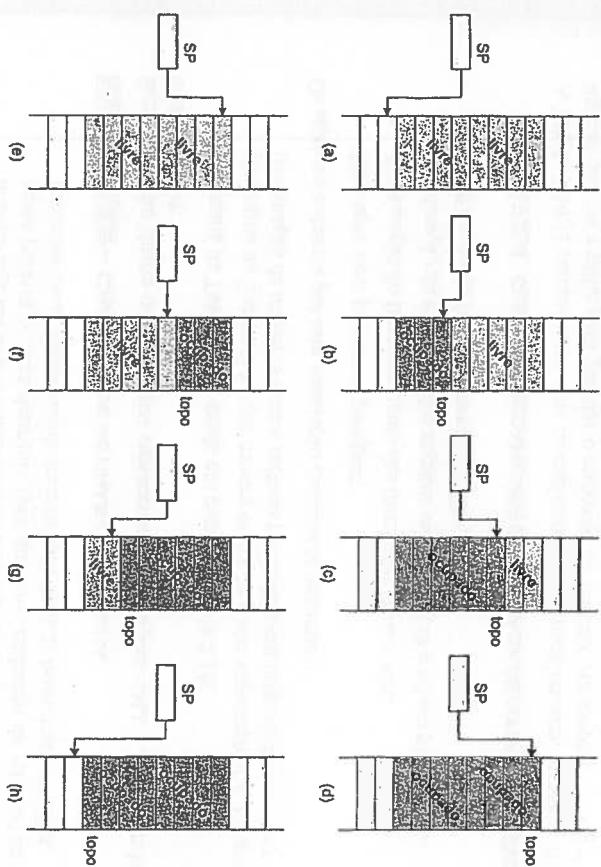


Fig. 5.7 - Variantes da implementação da pilha, em várias situações:  
 (a) a (d) – Pilha com topo em cima e SP a apontar para a primeira posição livre

Há duas formas de variar a implementação:

- O SP aponta para o topo da pilha (como na Fig. 5.7-a-d) ou para a primeira posição livre logo a seguir ao topo (como na Fig. 5.7-e-h);
- A pilha cresce para cima (Fig. 5.7-a-d) ou para baixo (Fig. 5.7-e-h).

Há assim quatro hipóteses, duas das quais representadas na Fig. 5.7. As outras duas correspondem a trocar as combinações das duas formas de variação. No fundo, todas as hipóteses acabam por ser equivalentes (implementam a mesma funcionalidade), sendo apenas uma questão de convenção.

A hipótese da Fig. 5.7-a-d parece ser mais intuitiva do que a da Fig. 5.7-e-h (porque "topo" na figura fica mesmo em cima), e essa é uma das razões porque tradicionalmente o SP é

inicializado com o endereço mais alto e depois as instruções de CALL fazem o SP diminuir, subindo na figura (convençionalmente que os endereços crescem de cima para baixo).

Mas o importante é o CALL e o RET serem compatíveis entre si. Se o RET funcionar de forma inversa ao CALL, então são compatíveis (mas atenção porque é preciso saber onde inicializar o SP). O PEPE utiliza o esquema da Fig. 5.7a-d.

#### ESSENCIAL

- As rotinas são o elemento básico de estruturação do código em linguagem assembly e correspondem às funções das linguagens de alto nível. As rotinas podem ter parâmetros (ou argumentos) e valores de retorno, que são passados tipicamente em registos.

As rotinas são chamadas com instruções de CALL, que guardam o endereço da instrução para onde devem retornar e saltam para o endereço da rotina. Terminam com instruções de RET, que obtem o endereço de retorno, para onde saltam.

- O endereço de retorno é sempre o da instrução seguinte ao CALL que invocou a rotina, que pode ser chamada a partir de muitos CALLs diferentes;
- O endereço de retorno é guardado em memória ou num registo (de acesso mais rápido mas que não suporta rotinas que chamam rotinas). Em memória, o endereço de retorno é guardado na pilha, numa zona de memória com acesso sequencial controlado por um apontador; o SP (*Stack Pointer*), que aponta para a última célula escrita (topo da pilha, graficamente por cima mas com endereço mais pequeno, usando a convenção de endereços mais altos por baixo);
- O CALL guarda o endereço de retorno por cima do topo (ficando a ser o novo topo e diminuindo o SP). O RET retira o endereço de retorno da pilha e o SP aumenta, passando a célula imediatamente por baixo a ser o novo topo.

### 5.7.3 OUTRAS UTILIZAÇÕES DA PILHA EM ROTINAS

#### 5.7.3.1 GUARDA DE REGISTOS NAS ROTINAS

Uma das decisões básicas que um compilador (ou programador de linguagem assembly) tem de tomar é atribuir variáveis a registos. Isto é, decidir que variável fica em que registo ou, de forma mais abrangente, decidir qual o significado de cada registo.

O ideal era cada rotina usar o seu próprio conjunto de registos, não partilhado com nenhuma outra rotina. Infelizmente, isto não é tecnicamente viável. O número de registos é muito limitado e, apesar de os compiladores tentarem a melhor utilização possível para os registos, o facto é que é frequente uma rotina chamar outra que use (altere) registos que estão a ser usados (e cujos valores não devem ser destruídos).

Naturalmente, este conflito só traz problemas se a rotina chamada alterar o valor do registo em questão, pois estraga o valor que a rotina tinha guardado nesse registo.

Isto pode suceder mesmo quando há um ou mais registos livres, que não estão mesmo a ser usados. Quando o compilador compila uma rotina, tem de decidir quais registos usar. Como uma rotina pode chamar várias, que por sua vez podem chamar outras, é virtualmente impossível (exceptuando os casos trivialmente simples) encontrar uma solução em que em nenhuma combinação de chamada de rotinas haja conflitos de registos.

Deve ainda ter-se em atenção o seguinte:

- Conflito é diferente de partilha. A semântica do programa pode ser tal que duas rotinas cooperem na definição do valor de um registo, e portanto é uma situação desejável;
- Só há conflito com problemas se houver escrita no registo. Se o registo só for lido pelas duas rotinas, então é um caso de partilha em leitura;
- Conflito pode assim definir-se como uma partilha em escrita não desejável;
- Os conflitos podem não ser directamente visíveis. Por exemplo, imagine-se que a rotina A usa o R2 e chama a rotina B, que não usa o R2 mas chama outra rotina C que, essa sim, usa e altera o R2. Do ponto de vista da rotina A, o valor de R2 foi alterado durante a chamada a B, destruindo o valor que A tinha guardado em R2.

Como exemplo, repare-se na rotina ordena da Tabela 5.8, na página 312. Esta rotina usa os registos indicados na Tabela 5.19, que analisa a utilização de cada um deles. Os registos R1 e R2 têm significado quer na rotina ordena quer na rotina que a chamou. São usados para passar os parâmetros para a rotina e retornar o respectivo resultado (seção 5.7.3.2). A rotina que chama tem de os inicializar com os valores correctos e a rotina chamada tem de retornar o resultado num dos registos. As duas podem também combinar que é a rotina chamada que produz valor(es) e o(s) armazena em determinado(s) registo(s). No fundo, é um contrato entre as duas rotinas, uma partilha dos registos entre elas.

| REGISTO | SIGNIFICADO               | ALTERA REGISTO | REGIME   |
|---------|---------------------------|----------------|----------|
| R1      | Parâmetro (x) e resultado | Sim            | partilha |
| R2      | Parâmetro (y)             | Não            | partilha |
| R3      | Valor temporário          | Sim            | confílio |
| R4      | Valor temporário          | Sim            | confílio |
| R5      | Variável local (auxiliar) | Sim            | confílio |

Tabela 5.19 - Registos usados pela rotina ordena e significado dessa utilização

Os registos R3, R4 e R5 já têm uma utilização diferente, que é apenas local à rotina ordena. Esta altera o seu valor mas não sabe se estes registos tinham conteúdo significativo.

tivo para a rotina que a chamou. Por seu turno, muito provavelmente a rotina que chama não faz ideia que a rotina chamada irá alterar estes registos, e poderá passar a funcionar mal se a informação contida nestes registos for destruída. É um caso de potencial conflito.

Neste exemplo simples, em que a rotina ordena faz parte de um pequeno programa (Tabela 5.8), não há conflito entre registos. No entanto, isto deve-se à dimensão extremamente reduzida do programa (e mesmo assim neste programa só RO está livre) e por o programador ter evitado usar os registos em mais do que uma rotina. Em qualquer programa de funcionalidade minimamente significativa o conflito é inevitável.

Assumindo que uma rotina A chama outra B, os conflitos em termos de utilização de registos só podem eliminar-se numa de três situações possíveis:

1. Alguém (o compilador ou o programador) verificou que não há, efectivamente, utilização de um mesmo registo por A e por B, e então não é preciso fazer mais nada;
2. A rotina A guarda o valor dos registos em memória antes de chamar a rotina B, recuperando o valor destes registos mal a rotina B retorna;
3. A rotina B guarda em memória os registos que precisar de alterar (antes de os alterar), tendo o cuidado de repor os valores originais antes de retornar.

A hipótese 1 só é viável em programas trivialmente simples.

A hipótese 2 tem o problema de replicar as operações de guarda/recuperação de registos em todas as chamadas a uma dada rotina, além de que eventuais alterações na rotina B (à medida que o programa é desenvolvido) podem obrigar a alterações em todos os pontos do programa em que B é chamada.

A hipótese 3 é a melhor porque a rotina B (chamada) é que sabe que registos irá alterar e portanto quais é que deve guardar/recuperar, embora esteja sujeita a perder tempo a guardar/recuperar registos que afinal a rotina A (a que chama) não estava a usar. No entanto, é um método seguro, pois a rotina pode ser chamada a partir de várias outras rotinas e nem todas usam os mesmos registos. Guardando/recuperando todos os registos que altera, a rotina garante que não interfere com o contexto da rotina que a chamou.

Como medida de boa programação em linguagem assembly, qualquer rotina que altere um registo só deve ser por mútuo acordo com a rotina que chama deve guardá-lo antes de o usar (tipicamente no inicio da rotina) e repor o seu valor antes de retornar. Mesmo que na altura em que a rotina é feita nem todos os registos precisem de ser guardados (porque a rotina que a chama não precisa disso), mesmo assim deve fazer-se, em antecipação para eventuais novas chamadas (por parte de outras rotinas). Esta regra não se aplica ao código gerado por compiladores, pois em cada compilação o compilador gera o código todo de novo e por isso pode facilmente adaptar-se às alterações efectuadas.

A zona de memória ideal para guardar os registos é a pilha, pois a necessidade de guardar/repor registos sucede-se ao ritmo das chamadas e retornos. O PEPE tem duas instruções específicas para este fim, PUSH e POP, descritas na Tabela 5.20.

**NOTA** Tal como no CALL (Tabela 5.16 e Tabela 5.21), o PUSH usa SP-2 para o acesso à memória e só depois actualiza o SP, por causa do mecanismo de memória virtual (ver comentário à Tabela 6.15, na página 48).

A Tabela 5.21 mostra a equivalência (apenas em termos de funcionalidade) de CALL e RET em instruções mais simples e mostra que, na prática:

- Chamar uma rotina (CALL) não passa de um salto (JMP) em que se tem o cuidado de guardar o valor do PC antes de saltar (o salto destrói o valor anterior do PC);
- Retornar de uma rotina (RET) não é mais do que recuperar o valor anterior do PC, após execução da rotina.

| INSTRUÇÕES | ACÇÕES RTL                                                              | IGUALDADE FUNCIONALMENTE                                     |
|------------|-------------------------------------------------------------------------|--------------------------------------------------------------|
| JMP Rs     | PC ← Rs ; salta sem preocupações de recuperar o valor anterior do PC    |                                                              |
| CALL Rs    | Mw[SP-2] ← PC<br>SP ← SP - 2<br>PC ← Rs<br>JMP Rs ; salta para a rotina | PUSH PC ; guarda PC na pilha (esta instrução não existe!)    |
| RET        | PC ← Mw[SP]<br>SP ← SP + 2                                              | POP PC ; recupera o valor do PC (esta instrução não existe!) |

Tabela 5.21 - Equivalência funcional das instruções CALL e RET e sua relação com JMP

**NOTA** As instruções PUSH PC e POP PC não existem. Nas instruções PUSH e POP apenas se podem especificar os registos do banco de registos (Fig. 4.5 na página 204), e o PC é o único que não está incluído. Foram indicadas desta forma apenas para se perceber a funcionalidade, que existe mas está enbebida nas instruções CALL e RET, não existindo em instruções separadas.

O Programa 5.5 serve apenas para ilustrar o mecanismo de guarda/recuperação de registos na pilha. O programa principal usa dois registos, R1 e R2, e chama a rotina A, que por sua vez chama a rotina B. Ambas guardam na pilha os registos que alteram e recuperam os seus valores antes de regressarem. Os números do lado esquerdo são os endereços das instruções.

A Tabela 5.22 mostra a evolução dos registos relevantes e da pilha após a execução de cada uma das instruções do Programa 5.5.

```

0000H MOV SF, 1000H ; inicializa o SP
0002H MOV R1, 111H ; inicializa R1
0004H MOV R2, 222H ; inicializa R2
0006H CALL A ; chama a rotina A
0008H fim: JRP fim ; programa acabou

A: PUSH R1 ; guarda o valor de R1 na pilha
000CH PUSH R2 ; guarda o valor de R2 na pilha
000EH MOV R1, 123H ; altera R1
0010H MOV R2, 456H ; altera R2
0012H CALL B ; chama a rotina B
0014H POP R2 ; repõe valor anterior de R2
0016H POP R1 ; repõe valor anterior de R1
0018H RET ; retorna da rotina A

B: PUSH R2 ; guarda o valor de R2 na pilha
001AH MOV R2, 5 ; altera R2
001EH POP R2 ; repõe valor anterior de R2
0020H RET ; retorna da rotina B

```

Programa 5.5 - Programa simples, com o único fim de ilustrar a guarda/recuperação de registos pelas rotinas

| END.  | INSTRUÇÃO      | R1    | R2    | SP    | OFFEH | OFFCH | OFFAH | OFF8H | OFF6H |
|-------|----------------|-------|-------|-------|-------|-------|-------|-------|-------|
|       | estado inicial | xxxx  |
| 0000H | MOV SP, 1000H  |       |       | 1000H |       |       |       |       |       |
| 0002H | MOV R1, 111H   | 0111H |       |       |       |       |       |       |       |
| 0004H | MOV R2, 222H   |       | 0222H |       |       |       |       |       |       |
| 0006H | CALL A         |       |       | 0FFEH | 0008H |       |       |       |       |
| 000AH | A: PUSH R1     |       |       | 0FFCH | 0111H |       |       |       |       |
| 000CH | PUSH R2        |       |       | 0FFAH | 0222H |       |       |       |       |
| 000EH | MOV R1, 123H   | 0123H |       |       |       |       |       |       |       |
| 0010H | MOV R2, 456H   |       | 0456H |       |       |       |       |       |       |
| 0012H | CALL B         |       |       | 0FFBH | 0004H |       |       |       |       |
| 001AH | B: PUSH R2     |       |       | 0FF6H | 0456H |       |       |       |       |
| 001CH | MOV R2, 5      |       | 0005H |       |       |       |       |       |       |
| 001EH | POP R2         |       | 0456H | 0FF8H |       |       |       |       |       |
| 0020H | RET            |       |       | 0FFAH |       |       |       |       |       |
| 0014H | POP R2         |       | 0222H | 0FFCH |       |       |       |       |       |
| 0016H | POP R1         | 0111H | 0FFEH |       |       |       |       |       |       |
| 0018H | RET            |       | 1000H |       |       |       |       |       |       |
| 0008H | fim: JRP fim   | 0111H | 0222H | 1000H | 0008H | 0111H | 0222H | 0014H | 0456H |

Tabela 5.22 - Evolução dos registos relevantes e da pilha após a execução de cada instrução do Programa 5.5

Este último ponto chama a atenção para o facto de ser preciso muito cuidado na utilização da pilha. Basta um pequeno descuido e o programa pode baralhar-se completamente. Os erros mais típicos relacionados com a guarda de registos na pilha são os seguintes:

- Não guardar/recuperar todos os registos. Este erro pode suceder logo quando se faz a rotina pela primeira vez, mas é mais típico de quem altera uma rotina, quando passa a usar mais um registo mas se esquece de acrescentar os respectivos PUSH e POP. Como o registo é estragado, partes do programa que já funcionavam podem deixar de funcionar;
- Recuperar os registos sem ser pela ordem inversa. O efeito prático é trocar os valores dos registos. Como exemplo, imagine-se que a rotina A era não como no Programa 5.5 mas sim da forma seguinte, em que a ordem do POPS é a mesma dos PUSHs, em vez de ser a inversa. R1 e R2 ficarão com os valores trocados!

```

A: PUSH R1 ; guarda o valor de R1 na pilha
     PUSH R2 ; guarda o valor de R2 na pilha
     MOV R1, 123H ; altera R1
     MOV R2, 456H ; altera R2
     CALL B ; chama a rotina B
     POP R1 ; repõe valor anterior de R2!
     POP R2 ; repõe valor anterior de R1!
     RET

```

- Emparelhamento incorrecto entre PUSHs e POPS (isto é, não há um POP para cada PUSH). Isto pode ocorrer essencialmente por duas razões típicas:
  - O programador começou a fazer a rotina e colocou os PUSHs. Entretanto, ao fazer a rotina, acabou por se esquecer dos POPS (ou pelo menos de alguns);

Para ser mais explícito, apenas os valores iniciais, finais e os que mudam estão representados (os outros são iguais ao valor anterior). O valor "xxxx" significa "não inicializado" ou "indeterminado". A zona a cíntento corresponde à utilização da pilha (palavras válidas). Note-se que:

- Com este esquema, as rotinas podem alterar os registos à vontade, desde que os guardem primeiro na pilha e reponham os respectivos valores antes de retornarem;

A zona a cíntento traduz o conjunto das palavras válidas na pilha, e por conseguinte a sua profundidade ao longo do tempo;

- Um RET ou um POP apenas reduz o número de palavras válidas na pilha (incrementando o SP), mas não destrói o conteúdo da pilha. Esta é a razão pela qual a última linha reflecte os valores que ficaram na pilha, apesar de esta terminar vazia;

Os dados (valores dos registos) são guardados na pilha à mistura com os endereços de retorno. O programa não precisa de saber onde estão os dados e os endereços de retorno. Basta que as operações de retirar valores da pilha sejam feitas exactamente pela ordem inversa das correspondentes operações de colocar valores na pilha.

- Um PUSH ou um POP está dentro de uma parte de código condicional (isto é, que pode ser ou não executada, dependendo de uma condição). Imagine-se por exemplo, que a rotina B era não como no Programa 5.5 mas sim da forma seguinte, em que o POP pode ou não ser executado. Se não for, o RET irá retornar da pilha e colocar no PC não o endereço de retorno da rotina B mas sim o valor que nessa altura estiver no topo da pilha, que nesse caso será o valor de R2 lá colocado pelo PUSH R2;

```
B:    PUSH R2 ; guarda o valor de R2 na pilha
      JZ SB ; retorna se o bit z for 1
      MOV R2, 5 ; altera R2
      POP R2 ; repõe valor anterior de R2
SB:   RET ; retorna da rotina B
```

Em qualquer destas duas situações, a consequência é o controlo retornar não para o endereço certo mas sim para uma instrução no endereço dado pelo valor anterior do último registo guardado (que, inclusivamente, poderá nem ser numa zona de instruções), e o mais provável é o programa baralhar-se completamente.

Outros efeitos possíveis são:

- POP com a pilha vazia (se faltar alguma instrução de PUSH ou erradamente houver POPS a mais);

- PUSH com a pilha cheia. Também se pode dever a um insuficiente dimensionamento da pilha, mas o mais provável é haver um ciclo em que falte um POP de modo que em cada iteração a pilha vai ficando com uma palavra (ou mais) em excesso ao que devia ter. Ao fim de algumas iterações, a pilha fica cheia. Outra causa possível é recursividade infinita (secção 5.7.3.5).

**NOTA** Estes erros são razoavelmente frequentes em programação manual em linguagem assembly (em particular se o programador tiver pouca prática). Os compiladores geram o código correcto, e só por erro grosseiro de quem programou o compilador (que rapidamente será detectado nos primeiros testes do compilador) é que o compilador cometerá um erro de utilização da pilha. A guarda/gestão/recuperação de registos nas linguagens de alto nível não existe para o programador. O compilador trata de tudo e o programador só tem de chamar as rotinas e retornar delas. O compilador insere automaticamente os PUSHs e POPS que forem necessários.

### SIMULAÇÃO – GUARDA DE REGISTOS NAS ROTINAS

Esta simulação ilustra o funcionamento da guarda/recuperação de registos nas rotinas, usando o Programa 5.5 e a Tabela 5.22 como base. Os aspectos cobertos incluem os seguintes:

- Execução com pontos de paragem;
- Verificação do funcionamento das instruções PUSH e POP;

- Verificação da evolução dos registos relevantes e da pilha;
- Verificação do efeito dos erros referidos sobre a utilização das instruções PUSH e POP.

Como exemplo aplicado ao programa de ordenação de números pelo método da bolha, a Tabela 5.23 mostra como a rotina ordena pode ser implementada (i) sem se preocupar em guardar os registos (reprodução desta rotina na Tabela 5.8, na página 312, para facilitar a comparação) e (ii) de forma mais correcta, preservando o valor dos registos.

| Modo                         | FUNÇÃO EM C                                                                                                                                                                                                                             | ROTEIRO EM ASSEMBLY                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| COM PRESERVAÇÃO DOS REGISTOS | <pre>int ordena (int* x, int* y) {     int auxiliar;     if (*x &gt; *y)         auxiliar = *x;         *x = *y;         *y = auxiliar;     return true; } else: MOV R1, 0 ; false       RET       ; o valor retornado está no R1</pre> | <pre>ordena: ; x vem em R1, y vem em R2 ; auxiliar usa o R5 MOV R3, [R1] ; *x MOV R4, [R2] ; *y CMP R3, R4 ; (*x &gt; *y) ? JLE else ; se não, executa o else else: MOV R5, R3 ; auxiliar = *x;       MOV [R1], R4 ; *x = *y;       MOV [R2], R5 ; *y = auxiliar;       MOV R1, 1 ; true       RET       ; return true;       ; return false;</pre>                                                                                                      |
| SEM GUARDAR REGISTOS         | <pre>int ordena (int* x, int* y) {     int auxiliar;     if (*x &gt; *y)         auxiliar = *x;         *x = *y;         *y = auxiliar;     return true; } else: MOV R1, 0 ; false       RET       ; o valor retornado está no R1</pre> | <pre>ordena: ; x vem em R1, y vem em R2 ; auxiliar usa o R5 PUSH R3 ; guarda o valor de R3 MOV R4, [R1] ; guarda o valor de R4 MOV R3, [R1] ; *x MOV R4, [R2] ; (*x &gt; *y) ? JLE else ; se não, executa o else else: MOV R5, R3 ; auxiliar = *x;       MOV [R1], R4 ; *x = *y;       MOV [R2], R5 ; *y = auxiliar;       MOV R1, 1 ; true       JMP acaba ; return true; acaba: MOV R1, 0 ; false       RET       ; o valor retornado está no R1</pre> |

Tabela 5.23 - Comparação entre duas implementações da rotina ordena, sem e com guarda/recuperação dos registos usados pela rotina.

Note-se que:

- A função em C é igual nos dois casos (é apenas uma questão do código gerado pelo compilador);
- Os POPS são feitos pela ordem inversa dos respectivos PUSHs;
- Na primeira versão há dois pontos de retorno, enquanto que na segunda só há um, com a etiqueta acaba (faz um salto para lá do outro ponto de retorno original). Isto é fundamental para o programador não se esquecer que no retorno os POPS têm de ser feitos. A alternativa é duplicar os POPS em ambos os pontos de saída, mas visualmente aparecem dois POPS para cada PUSH, o que, sem estar errado (só um dos POPS é executado em cada chamada à rotina), é nitidamente mais confuso;
- A guarda/recuperação de R5 é feita apenas dentro da instrução if. Podia estar junta com a das registos R3 e R4, mas assim evita-se perder tempo com o R5 caso a expressão booleana do if seja falsa. Verifica-se assim que a guarda/recuperação deve ser feita no âmbito mais restrito possível e não necessariamente a nível global da rotina.

#### **SIMULAÇÃO – ORDENAÇÃO POR BOLHA COM GUARDA DE REGISTOS**

Esta simulação aplica o mecanismo de guarda/recuperação de registos nas rotinas ao programa de ordenação de números pelo método de bolha. O programa de base é o da Tabela 5.8, com as adaptações necessárias para incluir preservação de registos. A Tabela 5.23 mostra a rotina ordenada da Tabela 5.8 já modificada para incluir este aspecto. O programa completo está incluído nesta simulação.

Os aspectos cobertos incluem os seguintes:

- Execução com pontos de paragem;
- Verificação do funcionamento das instruções PUSH e POP;
- Verificação da evolução dos registos relevantes e da pilha (em termos de conteúdo e de profundidade).

#### **5.7.3.2 VARIÁVEIS LOCAIS**

Nas linguagens de alto nível, uma variável local define-se como uma variável que só é válida num determinado âmbito restrito, normalmente uma função. Uma variável local não pode ser acedida a partir do exterior desse âmbito. O termo "local" aplica-se em oposição ao termo "global". Uma variável global é válida e acessível em qualquer ponto do programa. Para que tal aconteça, deve estar definida fora de qualquer função, incluindo a função principal (main).

Para uma função aceder a uma variável, ou (i) é global ou (ii) é passada como parâmetro ou (iii) é local (declarada dentro da função). No exemplo em C que temos visto neste capítulo (ordenação de números pelo método de bolha) não há variáveis globais. O programa é pequeno e a função ordenaBolha acede às variáveis seq1 e seq2 (Tabela 5.8).

na página 312) porque lhe são passadas como parâmetros, o que torna o programa mais robusto e claro por explicitar a que variáveis a função pode aceder.

Em termos de programação, as variáveis locais são preferíveis às globais, pois em cada ponto do programa o programador só tem acesso às variáveis de que precisa e não a todas, indiscriminadamente. Isto permite ao compilador confirmar os acessos às variáveis e detectar mais facilmente erros de programação. No entanto, as variáveis locais são mais difíceis de implementar.

As variáveis globais podem corresponder simplesmente a uma célula de memória, num dado endereço fixo, ficando acessível para leitura ou escrita durante todo o programa, por um simples MOV.

As variáveis locais (a uma função) têm uma semântica mais complexa, pois existem apenas enquanto o seu âmbito existir (no caso de uma função, entre o instante em que é chamada e aquele em que retorna). Por exemplo, a variável auxiliar é local à função ordena, na Tabela 5.23. Isto quer dizer que, de acordo com esta semântica, o espaço de memória que lhe corresponde não existe enquanto ordena não está em execução.

Isto é o que a semântica diz. Em termos de implementação, há várias opções:

1. O espaço de memória correspondente às variáveis locais de uma função é criado (reservado) sempre que a função é chamada e destruído (declarado livre) sempre que a função retorna. Estando o mecanismo de criação/destruição das variáveis locais intimamente ligado com o mecanismo de chamada/retorno de funções, o local óbvio para alocar espaço para as variáveis é a pilha;
2. Reservar espaço de forma fixa para todas as variáveis locais de todas as funções, em que só se acede às variáveis das funções cuja chamada estiver activa (as outras estão lá, mas não se lhes acede se a função a que pertencem não estiver em execução);
3. Usar registos para conter variáveis locais. Quando a função é chamada, guardam-se na pilha os valores dos registos que se pretendem usar como variáveis locais. Antes de a função retornar, os valores são repostos e cumpre-se o requisito de que quando a função retorna as variáveis locais deixam de existir.

O método 3 é o usado pela rotina ordena na Tabela 5.23 (em particular a 2.ª versão, que guarda e repõe os registos). A grande vantagem de usar registos é que durante a execução da função o acesso às variáveis locais é muito rápido, além de que muitas instruções podem manipular directamente os registos (ao contrário da memória, que só é acessível por MOV e cujos valores têm de ser trazidos para registos para serem processados). Tem de haver acessos à memória (para guardar e repor os registos, mas é só uma vez no inicio e no fim da rotina), sendo uma questão de ver se o número de acessos às variáveis locais corresponde a um benefício que compense o tempo gasto na preservação dos registos. A grande desvantagem deste método reside no facto de os registos serem em número muito limitado, além de que não contempla variáveis locais de tipos estruturados (vectores, por exemplo).

O método 2 não tem problemas com o número limitado de registos, mas o acesso a cada variável local (em memória) é mais lento do que se estivesse num registo. O facto de ter espaço de memória reservado para todas as variáveis locais possíveis, a ser usadas ou não, não é grande problema (os computadores actuais têm normalmente memória em quantidade suficiente). Uma grande desvantagem deste método é não suportar recursividade (chamada de uma rotina por ela própria – ver secção 5.7.3.5), pois todas as chamadas a uma rotina usam as mesmas variáveis locais.

O método 1 tem a grande vantagem de suportar recursividade, pois cada nova chamada da rotina implica um novo conjunto de variáveis locais. O acesso às variáveis locais não é tão rápido como se estivessem em registos (a pilha é na memória), mas em compensação não tem problemas com o número limitado de registos.

Assim, compete ao compilador ou ao programador de linguagem assembly escolher qual o método que pretende. Os compiladores tentam normalmente optimizar o desempenho, pelo que se compensar usam o método 3 (que suporta recursividade, dado que os registos são preservados na pilha), senão recorrem tipicamente ao método 1, que é o mais geral e funciona em qualquer circunstância. Os programadores de linguagem assembly usam tipicamente o que for mais fácil, o que significa muitas vezes usar apenas variáveis globais...

**NOTA** A função `ordena`, em C (Tabela 5.23), só tem uma variável local, auxiliar. No entanto, na rotina em linguagem assembly vê-se que são precisas mais duas (representadas por R3 e R4) para manter valores temporários. Assim, tem de se reservar espaço para três variáveis locais e não apenas para uma.

A reserva e a libertação de espaço para variáveis locais na pilha são extremamente simples:

- No início da rotina, decrementa-se o SP de 6 unidades (3 palavras). Isto corresponde a fazer três PUSHs, mas sem tempo gasto a escrever na memória. Estas palavras ficam assim ocupadas, e podem ser acedidas pela rotina (que terá ainda de as inicializar);
- No fim da rotina (antes de retornar), incrementa-se o SP de 6 unidades, o que faz libertar as 3 palavras das variáveis locais. Esta operação equivale a três POPs, mas sem perder tempo a ler o conteúdo das variáveis na memória.

O problema está no acesso. As variáveis locais ficam algures na pilha, mas o SP varia tanto que não serve de referência para fazer acesso a valores que estão na pilha. Para isso é necessário outro registo. Este aspecto, bem como um exemplo de criação e destruição das variáveis locais, é apresentado na secção 5.7.3.4, que inclui também uma simulação.

### 5.7.3.3 PASSAGEM DE PARÂMETROS E DO RESULTADO

A rotina que chama e a rotina chamada têm âmbitos diferentes. Nenhuma tem acesso às variáveis locais da outra. Partilhando apenas as variáveis globais. No entanto, a rotina que chama tem de passar informação à chamada, ou seja, tem de preencher os valores dos

parâmetros da rotina chamada. Por seu turno, a rotina chamada poderá ter de devolver um valor (resultado) à rotina que a chamou.

Em termos de linguagem de alto nível, quer os parâmetros quer o resultado podem ser passados por:

- Valor – O que é passado é um valor de dados, seja simples (um inteiro), seja estruturado (um vector, por exemplo). A semântica diz que na passagem é feita uma cópia do valor. Se, por exemplo, a rotina chamada alterar o valor do parâmetro que recebe, está a alterar a cópia e portanto a rotina que chama não sente essa alteração;
- Referência – O que é passado não é um valor de dados, mas sim um apontador para (endereço de) a célula de memória onde está o valor de dados (ou da primeira célula, no caso de dados estruturados). O que é copiado na passagem é apenas o apontador (endereço, que ocupa sempre uma palavra, sejam os dados simples ou estruturados). Tanto o original do apontador como a sua cópia aponta para o mesmo valor de dados. Se, por exemplo, a rotina chamada alterar o valor do parâmetro que recebe (através do apontador), está a alterar o original e portanto a rotina que chama sente essa alteração. Ou seja, o valor de dados é partilhado entre a rotina que chama e a rotina chamada.

Como exemplo, veja-se a função `ordenaBolha` no Programa 5.4 (na página 315). O parâmetro `n` é passado por valor. O que a rotina `ordenaBolha` recebe é uma cópia do valor que a rotina que a chamou (neste caso, o programa principal) lhe passou. Pode

alterar o valor de `n` à vontade (embora este exemplo não o faça), sem qualquer interferência no valor original que foi copiado para `n` (`N` e `P`, que são constantes, aliás, e portanto nunca poderiam ser alteradas).

O parâmetro `a` é um apontador que permite aceder a um vector, o mesmo a que o programa principal (que passa este parâmetro a `ordenaBolha`) tem acesso. Ou seja, se `a` é alterado, o programa principal sente essa alteração após `ordenaBolha` retornar. Portanto, o vector é passado por referência (semântica de partilha) e não por valor (semântica de cópia). As vantagens da passagem por referência são essencialmente as seguintes:

- Permitem à rotina chamada alterar vários dados (através de vários parâmetros) e várias vezes durante a sua execução;
- Evitam a cópia (na passagem dos parâmetros e do resultado) de grandes estruturas de dados, que se pode tornar pesada em termos de tempo de execução.

Idêntico raciocínio se pode seguir em relação ao retorno de um resultado, que pode ser um valor de dados (passagem por valor) ou um apontador para um valor de dados (passagem por referência). Neste caso, há que ter cuidado em não retornar apontadores para variáveis locais a uma função, pois elas deixam de estar válidas quando a função retorna.

**NOTA**

Na realidade, a passagem por referência não existe. Neste caso, é passado um apontador, que é copiado (o que significa que o apontador em si é passado por valor). No entanto, estamos interessados no dado para que o apontador aponta, não no apontador em si, e por isso dizemos que "o dado é passado por referência" quando deveríamos dizer "é passada uma referência (o apontador) para o dado".

Em termos práticos, põe-se o problema de saber onde é que a rotina que chama coloca os parâmetros (sejam os valores propriamente ditos, sejam os apontadores para eles) para que a rotina chamada os possa ir lá buscar e vice-versa em relação ao resultado.

Há essencialmente as seguintes hipóteses:

- **Em variáveis globais** (endereços de memória fixos), que desempenham um papel semelhante a caixas de correio – Embora funcione, é uma forma pobre de conseguir o objectivo. Os valores a passar à rotina chamada têm de ser copiados de onde estão para as variáveis que servem de parâmetros, onde depois têm de ser lidos, além de que estão acessíveis a todo o programa (e portanto sujeitos a ser alterados inadvertidamente por alguma rotina, devido a erro de programação). Este método tem ainda a desvantagem de não suportar recursividade (chamada de uma rotina por ela própria – ver secção 5.7.3.5);

**Por registos** – Continua a não suportar recursividade, mas tem o grande atractivo de ser muito mais rápido do que em memória. Além disso, os compiladores conseguem normalmente optimizar o processo fazendo coincidir os registos em que se produzem os valores a passar à rotina chamada com os registos que servem de parâmetro a essa rotina, evitando cópia de valores. Este é o método mais eficiente, sendo normalmente o método usado quando não há recursividade envolvida (ou então tem de se combinar com guarda de registos na pilha) e o compilador consegue gerir os (poucos) registos existentes para que haja suficientes para o número de parâmetros a passar;

**Pela pilha** – Este é o mecanismo mais geral, suportando facilmente a recursividade e sem limitações de número de parâmetros, mas tem duas desvantagens:

- Exige cópia de valores para a pilha (por parte da rotina que chama, para colocar os parâmetros) e da pilha para registos (a rotina chamada precisa de buscar os parâmetros à pilha, para poder fazer algo com eles), o que tem impacte no desempenho;
  - Na rotina chamada, os registos para os quais os parâmetros precisam de ser lidos têm, por sua vez, de ser guardados na pilha antes de ser usados (para preservar o seu conteúdo), o que ainda tem mais impacte no desempenho.
- A rotina ordena, na Tabela 5.8 (página 312), usa passagem de parâmetros e do resultado por registos. A rotina que chama (`ordenaBolha`) preenche os registos R1 e R2 com os valores correctos e a rotina chamada (`ordena`) lê-os. No retorno do resultado procede de forma semelhante. A rotina ordena não tem de preservar estes dois registos porque ambos são usados para passagem de parâmetros, por mísimo acordo entre ordena e `ordenaBolha`. Quando muito, `ordenaBolha` teria de guardar na pilha os seus valores.

anteriores antes de os usar para passar os parâmetros a ordena. No entanto, ordena deverá preservar os outros registos que utilizar, nomeadamente o R3, R4 e R5, tal como ilustrado pela Tabela 5.23.

Os princípios básicos de passagem de parâmetros e resultados pela pilha são os seguintes:

- A rotina que chama coloca os parâmetros na pilha e espera que a rotina chamada lá deixe o resultado, de onde tipicamente o tira para colocar num registo;
- A rotina chamada retira os parâmetros da pilha, executa as suas instruções e deixa o resultado na pilha.

O programa da Tabela 5.24 é mais simples que o da Tabela 5.8 e destina-se exclusivamente a ilustrar e comparar de forma mais específica a passagem de parâmetros, quer por registos quer por pilha. Note-se que o programa em C na Tabela 5.24 é exactamente o mesmo nos dois casos. Estamos apenas a discutir qual a melhor forma de implementar a passagem de parâmetros. É notório que a passagem de parâmetros por registos é mais simples (menos instruções), pois a passagem pela pilha também tem de circular os valores pelos registos (os processadores só processam dados se estiverem em registos) e ainda colocá-los em memória, para a seguir os ler novamente para registos na rotina chamada!

Ainda assim a implementação da passagem de parâmetros por pilha na Tabela 5.24 não está correcta, pois não pode ser feita de forma tão simplista. A Tabela 5.25 ajuda a perceber porquê. É constituída por três partes: uma que descreve os princípios básicos, outra a implementação mais óbvia mas errada (a usada na Tabela 5.24) e a terceira, já no caminho para a implementação correcta. X, Y, Z, W e V são nomes genéricos para os registos (cabem ao programador ou compilador decidir quais e verificar que estão disponíveis).

A implementação mais óbvia deste esquema é usar directamente PUSHs e POPs, tanto na rotina que chama como na rotina chamada (parte do meio da Tabela 5.25), tal como usado na Tabela 6.24. Se do ponto de vista da rotina que chama isto está correcto (note-se que a rotina que chama é a mesma nas duas implementações na Tabela 5.25), está completamente errado do ponto de vista da rotina chamada, pois (Fig. 5.8):

- Quando a rotina chamada começa a executar, já não são os os parâmetros que estão no topo da pilha, mas sim o endereço de retorno, que o CALL lá colocou após os parâmetros (Fig. 5.8c). Logo, não basta apenas fazer POPs para obter os parâmetros. É preciso aceder à pilha directamente, por baixo do topo;
- Os parâmetros não se podem colocar em registos sem guardar esses registos primeiro, o que afasta ainda mais os parâmetros do topo da pilha (Fig. 5.8d);
- Antes de retornar, não basta fazer PUSH do resultado. Isto coloca-lo-ia no topo da pilha, em cima dos registos guardados no inicio da rotina, do endereço de retorno e dos parâmetros. Ora o que se pretende é que após o retorno tudo tenha desaparecido da pilha, ficando apenas o resultado no topo para a rotina que chamou fazer POP do resultado. Ou seja, exactamente na célula onde estava o primeiro parâmetro guardado na pilha antes do CALL (Fig. 5.8e), o que implica mais um acesso directo à pilha, sem ser pelo topo, para lá colocar o resultado.

| PROGRAMA EM C                                                                                                                                                                                                       | PROGRAMA EM ASSEMBLY (PASSAGEM POR REGISTOS)                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | PROGRAMA EM ASSEMBLY (PASSAGEM PELA PILHA)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>int x; int y;  main () {     x = max (3, 5);      y = dobro (x);  }  int max (int a, int b) {     if (a &gt;= b)         return a;     else         return b; }  int dobro (int n) {     return 2 * n; }</pre> | <pre>PLACE 1000H ; localiza bloco de dados x: WORD ; int x; y: WORD ; int y;  PLACE 0000H ; localiza bloco de código main: MOV SP, 2000H ; limite superior da pilha MOV R0, 3 ; 1.º parâmetro (a) MOV R1, 5 ; 2.º parâmetro (b) CALL max ; chama função MOV R8, x ; endereço de x (1000H) MOV [R8], R0 ; x = max (3, 5);  fim: JMP fim ; fim do programa  max: CMP R0, R1 ; (a&gt;=b)? JLT else RET  else: MOV R0, R1 ; return a; RET  dobra: MOV R2, 2 ; parâmetro e resultado em R0 MUL R0, R2 ; 2 * n RET</pre> | <pre>PLACE 1000H ; localiza bloco de dados x: WORD ; int x; y: WORD ; int y;  PLACE 0000H ; localiza bloco de código main: MOV SP, 2000H ; limite superior da pilha MOV R0, 3 ; 1.º parâmetro (a) PUSH R0 ; coloca-o na pilha MOV R0, 5 ; 2.º parâmetro (b) PUSH R0 ; coloca-o na pilha CALL max ; chama a função POP R0 ; obtém resultado MOV R8, x ; endereço de x (1000H) MOV [R8], R0 ; x = max (3, 5); PUSH R0 ; coloca x na pilha CALL dobro ; chama a função POP R0 ; obtém resultado MOV R8, y ; endereço de y (1002H) MOV [R8], R0 ; y = dobro (x); JMP fim ; fim do programa  fim: POP R2 ; obtém 2.º operando (b) POP R1 ; obtém 1.º operando (a) CMP R1, R2 ; (a&gt;=b)? JLT else RET  else: PUSH R1 ; if (a&gt;=b)       ; a       ; return a;       ; b       ; return b;  dobra: POP R3 ; obtém operando (n), MOV R4, 2 ; 2 * n MUL R3, R4 ; 2 * n PUSH R3 ; coloca resultado na pilha RET</pre> |

Tabela 5.24 - Programa simples destinado apenas a ilustrar a passagem de parâmetros (por registos e pela pilha). A passagem por registos é notoriamente mais simples, e mesmo assim a passagem por pilha está errada (ver texto)

| CLAS. | ROTEIRA QUE CHAMA                                                                                                                                                                 | ROTEIRA CHAMADA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|       | <p>Pilha &lt;- parâmetro 1<br/> pilha &lt;- parâmetro 2<br/> ... ; outros parâmetros<br/> pilha &lt;- parâmetro N<br/> CALL rotina<br/> registo V &lt;- pilha<br/> ;resultado</p> | <p>rotina: registo Z &lt;- pilha ; parâmetro N<br/> registro Y &lt;- pilha ; parâmetro 2<br/> registo X &lt;- pilha ; parâmetro 1<br/> ... ; instruções da rotina<br/> ... ; instruções da rotina<br/> pilha &lt;- registo W ; resultado</p>                                                                                                                                                                                                                                                                                           |
|       | <p>PUSH parâmetro 1<br/> PUSH parâmetro 2<br/> ... ; outros parâmetros<br/> PUSH parâmetro N<br/> CALL rotina<br/> POP registo V ; resultado</p>                                  | <p>rotina: POP registo Z ; parâmetro N<br/> POP registo Y ; parâmetro 2<br/> POP registo X ; parâmetro 1<br/> ... ; instruções da rotina<br/> ... ; instruções da rotina<br/> PUSH registo W ; resultado</p>                                                                                                                                                                                                                                                                                                                           |
|       | <p>PUSH parâmetro 1<br/> PUSH parâmetro 2<br/> ... ; outros parâmetros<br/> PUSH parâmetro N<br/> CALL rotina<br/> POP registo V ; resultado</p>                                  | <p>rotina: PUSH registo X ; guarda registo X<br/> PUSH registo Y ; guarda registo Y<br/> ... ; PUSH de outros registos a guardar<br/> registo z &lt;- pilha ; parâmetro N<br/> ... ; obtém outros parâmetros<br/> registo Y &lt;- pilha ; parâmetro 2<br/> registo X &lt;- pilha ; parâmetro 1<br/> ... ; instruções da rotina<br/> ... ; instruções da rotina<br/> pilha &lt;- registo W ; resultado<br/> ... ; POP de outros registos a repor<br/> POP registo Y ; repõe registo Y<br/> POP registo X ; repõe registo X<br/> RET</p> |

Tabela 5.25 - Passagem de parâmetros e resultado pela pilha: princípio básico e duas implementações, simplista (mais errada) e correcta (embora simplificada)

A Fig. 5.8 revela o que acontece na pilha ao longo da execução da chamada da rotina da Tabela 5.25. A partir da Fig. 5.8f, o SIP começa a regressar. Note-se que os valores já escritos na pilha e que ficam por cima não são destruídos. Ficam na pilha à mesma, mas na zona livre (na figura, esse facto é representado com letras a traço mais ténue), o que significa que provavelmente serão destruídos quando houver novas chamadas de rotinas. Isto é, a divisão com traço mais forte na pilha é meramente conceptual, para marcar o ponto que divide as zonas da pilha com a informação já existente (antes da chamada à rotina) e com a informação sobre a rotina chamada na Tabela 5.25.

Há ainda alguns problemas por resolver (o que será feito na secção 5.7.3.4, que inclui

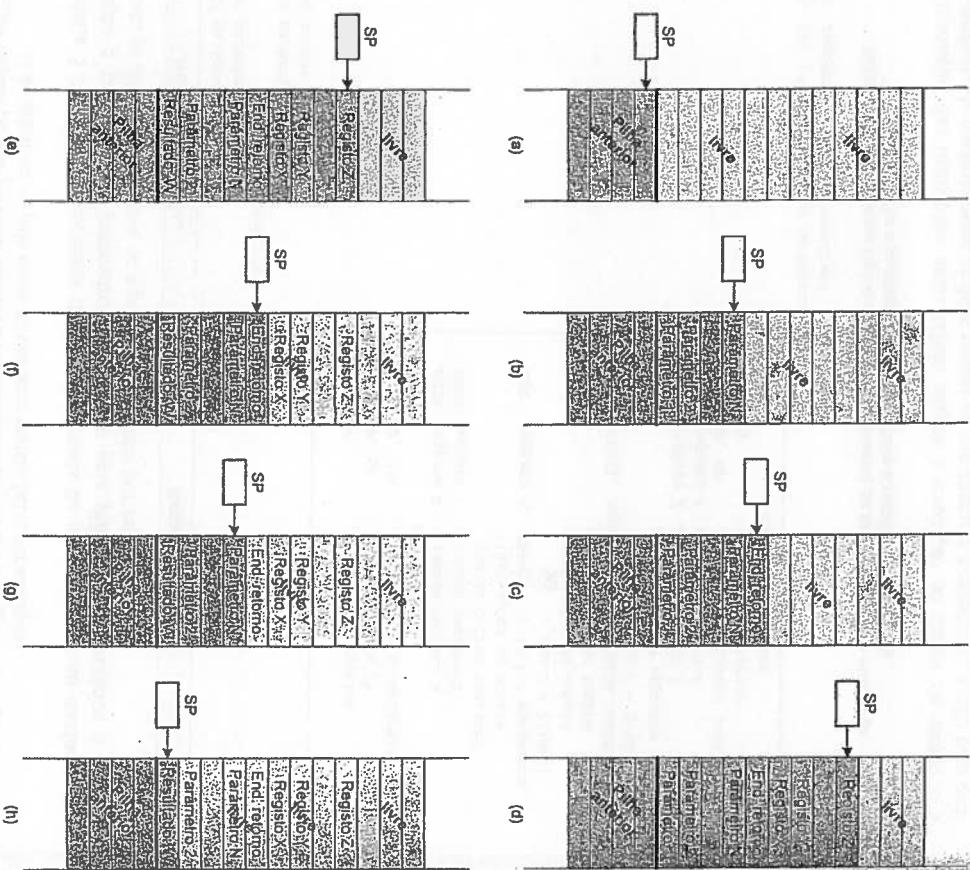


Fig. 5.8 - Evolução da pilha ao longo da chamada da rotina da Tabela 5.25:

- (a) - Estado Inicial; (b) - Após colocar parâmetros na pilha; (c) - Após o CALL;
- (d) - Após a rotina guardar os valores de registos na pilha; (e) - Após escrever o resultado
- (f) - Após repor os registos guardados na pilha; (g) - Após o RET
- (h) - Estado que a pilha deveria ter após o RET

- A parte de baixo da Tabela 5.25 mostra a solução correcta, mas não explica os detalhes do acesso directo à pilha (sem ser pelo topo com PUSH ou POP), para aceder aos parâmetros e ao resultado;

- \* Também já na secção 5.7.3.2 se tinha posto o problema do acesso directo à pilha, a propósito das variáveis locais;
- \* A Tabela 5.25 acaba com a instrução RET, ou seja, com a Fig. 5.8g, ficando por esclarecer como é que se passa para a situação da Fig. 5.8h (isto é, como é que se consomem os parâmetros de modo a que imediatamente após o retorno apenas o resultado fique na pilha).

Pelas razões expostas nesta secção, o método mais comum é usar registos para a passagem de parâmetros e resultados por registos. Como estes são em número limitado, acaba por ter de se guardar alguns na pilha para os recuperar mais tarde, mas apenas à medida do estritamente necessário. Também é possível usar passagem pela pilha quando houver muitos argumentos, por exemplo.

**Nota** A passagem de parâmetros é um dos factores que contribui para diminuir o desempenho do mecanismo de chamada/retorno de funções, pelo que alguns processadores comerciais optimizam este mecanismo reservando alguns registos especificamente para este efeito. Se a função tiver mais parâmetros do que o número destes registos, então o compilador recorre à pilha para passar os parâmetros adicionais, mas verifica-se estatisticamente que isto ocorre apenas numa minoria das chamadas de funções.

#### 5.7.3.4 CONTEXTOS DE CHAMADA DAS ROTINAS

A solução para os problemas passa por encontrar uma forma de manipular os valores da pilha de forma aleatória, como se fosse uma memória normal. Cada chamada a uma rotina precisa de guardar na pilha informação sobre essa chamada, que inclui:

- \* Parâmetros (os que não forem passados por registos);
- \* Endereço de retorno;
- \* Endereço de contexto;
- \* Variáveis locais;
- \* Registos guardados (para os repor no fim da execução da rotina chamada).

A este conjunto de informação chama-se **contexto de chamada** (*stack frame*, ou simplesmente contexto) da rotina. Alguns dos itens são de acesso normal numa pilha (POIS por ordem inversa dos PUSH), como o endereço de retorno e os valores dos registos guardados, mas outros devem poder ter acesso aleatório, como se de uma vulgar memória se tratasse, como os parâmetros e as variáveis locais (para poderem ser acedidos de forma fácil pelas instruções da rotina).

Falta um ponto de referência, que seja fixo para cada contexto. O SP não serve, pois qualquer operação sobre a pilha o altera. O que faz falta é outro registo cujo valor mude apenas quando uma função é chamada ou retornada e que possa servir de referência para os acessos pretendidos. Normalmente, utiliza-se um registo chamado FP (*Frame Pointer*, ou Apontador de Contexto). Alguns processadores têm um registo dedicado exclusivamente a este fim (podendo a sua manipulação estar incluída na implementação das instruções CALL e RET), enquanto outros recorrem simplesmente a um registo de

utilização geral, com a sua manipulação em *software*, através de instruções inseridas automaticamente pelo compilador.

O PEPE utiliza este segundo esquema, por questões de simplicidade e flexibilidade. Dado que num programa as chamadas a rotinas são uma constante, na prática o registo que se usa como FPR deve estar dedicado em exclusividade para este fim.

**NOTA.** O FPEB não tem um registo chamado FP. Nos exemplos que se seguem, é apenas para se perceber melhor a sua funcionalidade. Para o programar -ia de substituir FP pelo nome de um outro registo (R10, por exemplo).

A Tabela 5.26 mostra o esquema típico da chamada de uma rotina, com mudança de contexto e passagem de parâmetros e resultado pela pilha (os comentários a seguir indicam as diferenças no caso de a passagem ser feita por registos).

| ROTEIRA QUE CHAMA                                                                                       | ROTEIRA CHAMADA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> PUSH parâmetro 1 PUSH parâmetro 2 ...     ; outros parâmetros PUSH parâmetro N CALL rotina </pre> | <pre> rotina: PUSH FP      ; guarda FP anterior MOV  FP, SP   ; novo valor de FP SUB  SP, 2*K  ; espaço para K variáveis                 ; locais PUSH registo X ; guarda registo X PUSH registo Y ; guarda registo Y ...                 ; guarda outros registros ...                 ; instruções da rotina MOV  registo Y, [FP+C] ; 1.º parâmetro, ...           ; com C = 2*(N+1) ...           ; instruções da rotina MOV  [FP+C], registo W ; guarda resultado, ...           ; com C = 2*(N+1) ...           ; repõe outros registros POP  registo Y ; repõe registo Y POP  registo X ; repõe registo X MOV  SP, FP   ; elimina variáveis locais POP  FP      ; repõe FP anterior RET             ; retorna da rotina </pre> |

**Tabela 5.26 - Esquema típico de chamada/retorno de uma rotina com mudança de contexto e passagem de parâmetros e resultado pela pilha**

A disposição das instruções nesta tabela sugere a evolução ao longo do tempo e a transferência de controlo entre as duas rotinas (a que chama e a chamada), razão pela qual as instruções da rotina que chama aparecem separadas (no programa, o ADD aparece logo

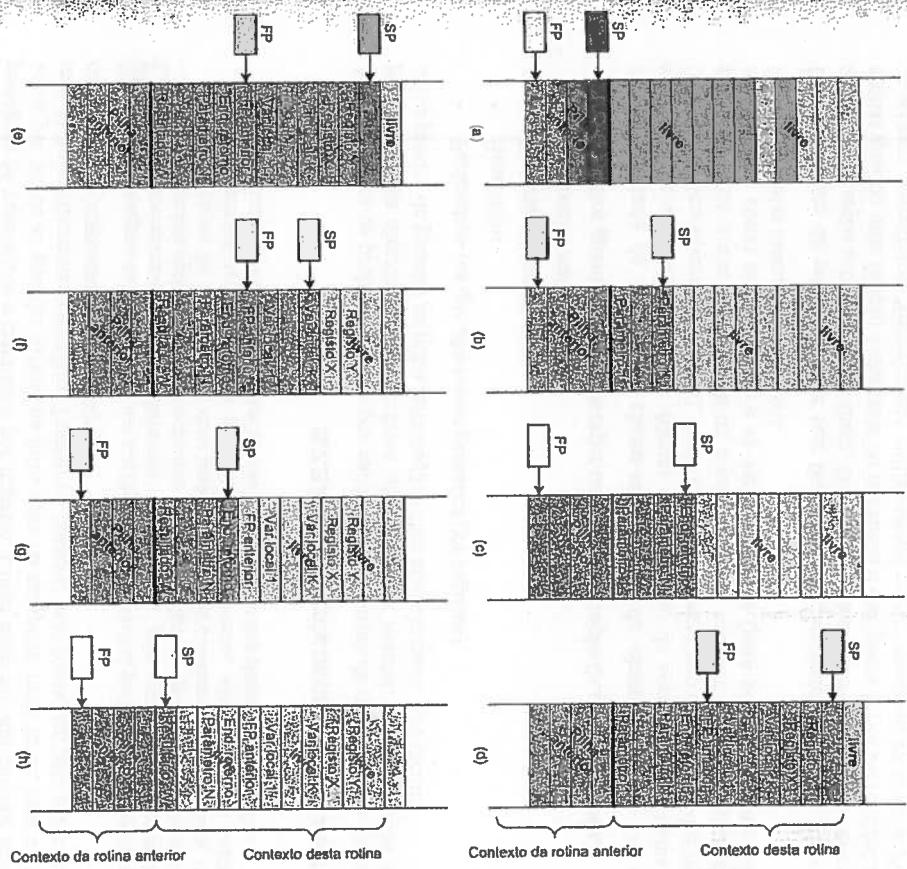


Fig. 5.9 - Evolução do contexto de uma rotina ao longo da sua chamada:

- (d) — Após a rotina registrar os registros na pilha; (e) — Após escrita do resultado (registro W); (f) — Após repor os registros guardados na pilha; (g) — Após o retorno; (h) — Estado que a pilha deveria ter após o retorno

Este esquema obedece essencialmente à sequência seguinte:

■ Antes de fazer a chamada, a rotina que chama coloca os parâmetros na pilha, com **PUSH** (Fig. 5-9b):

- A instrução CALL coloca o endereço de retorno acima dos parâmetros na pilha, ficando no topo (Fig. 5.9c);

A primeira coisa que a rotina chamada faz é mudar o contexto, guardando primeiro o valor do FP do contexto anterior, que pertence à rotina que chamou esta, e colocando o novo valor de FP, a referência de localização do contexto, precisamente a apontar para a posição da pilha em que o FP anterior foi memorizado (Fig. 5.9d);

Em seguida, a rotina chamada reserva espaço para as variáveis locais. Na Tabela 5.26 consideram-se K variáveis simples (de 16 bits cada), sendo portanto necessário subtrair  $2 \times K$  unidades (bytes) ao SP (conveniente lembrar que os endereços crescem de cima para baixo). Se as variáveis locais forem estruturadas, o valor poderá ser diferente. Em qualquer dos casos, tem de se reservar o número de bytes necessários (Fig. 5.9d);

Tem ainda de guardar na pilha o conteúdo dos registos que é preciso preservar, fazendo PUSH desses registos (Fig. 5.9d);

O contexto desta rotina começa no primeiro parâmetro colocado na pilha e termina no topo da pilha (Fig. 5.9d);

Depois disto, começa a execução da rotina propriamente dita. Quer os parâmetros, quer as variáveis locais, podem facilmente ser acedidas através de um MOV, especificando como endereço a soma de FP com uma constante (positiva para os parâmetros, negativa para as variáveis locais). O compilador sabe qual a posição (que é fixa) relativa ao FP de cada parâmetro e cada variável local, por isso pode especificar esta constante em cada acesso a um parâmetro ou variável local. Exemplos (conferir com a Fig. 5.9d):

```
MOV [FP-2], R3 ; copia R3 para a 1.ª variável local
MOV R5, [FP+4] ; copia o último parâmetro para R5
```

O resultado deve ser armazenado na posição em que o primeiro argumento está (Fig. 5.9e), para ser o último a retirar da pilha no fim de todo o processo de chamada e retorno. Isto é feito por um MOV, como se fosse uma operação de escrita no primeiro parâmetro;

- O retorno da rotina deve efectuar as operações da chamada em ordem inversa, começando por repor os valores de todos os registos guardados, com POPS (Fig. 5.9f); Segue-se a destruição das variáveis locais, o que é feito incrementando o SP de  $2 \times K$  unidades (o inverso da operação efectuada na chamada) ou simplesmente colocando em SP o valor de FP (pois este aponta para a posição imediatamente por baixo das variáveis locais – ver a Fig. 5.9f);
- O contexto anterior é reposto com um simples POP FP (Fig. 5.9g), ficando o endereço de retorno no topo da pilha;
- A rotina chamada pode simplesmente executar um RET e retornar à rotina que chamou. No entanto, ainda é preciso consumir os parâmetros, se estes forem mais do que um (Fig. 5.9g), havendo duas hipóteses:

Na Tabela 5.26 usou-se a hipótese mais simples, que é a rotina que chamou fazer isso, incrementando o SP de um valor adequado;

Mais elegante é ser a rotina chamada a fazer isto, mas para tal tinha de copiar o endereço de retorno para a posição do segundo parâmetro (logo acima do resultado), actualizar o SP para apontar para esse endereço de retorno e retornar, podendo terminar no modo indicado a seguir (usando o SP como registo auxiliar, uma vez que os restantes já foram repostos e o SP vai ser inicializado a seguir), em que  $n\delta$  é uma constante cujo valor (determinado pelo compilador) é o dobro do número de parâmetros (representa o número de bytes que os parâmetros ocupam).

```
POP registo X ; repõe registo X
MOV SP, [FP-2] ; endereço de retorno (aqui SP é apenas um
                ; registo auxiliar, para não estragar outro)
MOV [FP+n\delta], SP ; copia endereço de retorno para a posição acima
                    ; do resultado
MOV SP, FP ; recupera SP
ADD SP, n\delta ; SP fica a apontar para a nova posição do
                ; endereço de retorno
MOV FP, [FP] ; repõe o FP anterior
RET ; retorna da rotina chamada
```

Note-se que esta compensação, decorrente do consumo dos parâmetros, só é necessária se o número de parâmetros for superior a 1. No caso oposto, em que não há parâmetros mas há um resultado, o compilador tem de inserir espaço para o resultado quando a rotina é chamada, inserindo uma instrução SUB SP, 2 logo no início da rotina chamada, ainda antes do PUSH FP;

Finalmente o resultado pode ser obtido fazendo um simples POP, e a pilha volta a estar no estado em que estava antes desta chamada de rotina (Fig. 5.9h).

Se a passagem de parâmetros e resultado for por registos, então as alterações na Tabela 5.26 são as seguintes:

- A rotina que chama não coloca os parâmetros na pilha, mas sim em registos. Também não tem de consumir os parâmetros na pilha. De igual modo, o resultado não virá na pilha, mas sim num registo;
- A rotina chamada não sofre qualquer alteração, excepto nas instruções que accedem aos parâmetros, que não os leem da memória mas antes dos registos respectivos.

A passagem de parâmetros por registos é sempre mais simples. A passagem pela pilha usase essencialmente quando não há mais registos disponíveis para passar parâmetros. Quando há vários contextos na pilha (várias rotinas chamadas mas ainda não retomadas), é possível usar o FP e os seus valores anteriores memorizados na pilha para ter acesso à informação de cada um destes contextos. Isto é muito importante para o desenvolvimento dos programas. Quando se pára o programa num ponto de paragem (*breakpoint*), é possível ver toda a informação sobre a rotina que estava em execução e sobre todas as

que a chamaram,<sup>45</sup> permitindo assim ver a história do programa e dando informação preciosa para poder perceber qual o erro que está a ocorrer nesse programa.

O mecanismo dos contextos é ortogonal ao mecanismo de chamada/retorno de rotinas e de passagem de parâmetros. É perfeitamente possível usar contextos (com o registo FP) com CALL/RET ou CALFR/RET, com passagem de parâmetros pela pilha ou por registos, e misturar as diversas combinações no mesmo programa, o que permite optimizar o desempenho de funções simples e que não invoquem outras. Claro que ao invocar uma rotina que não use a pilha (com CALLE, que não precisa de guardar registos nem temha variáveis locais na pilha) muito provavelmente o compilador nem sequer gerará as instruções para mudar o FP, pois essa rotina não o usará. Em termos de depuração do programa (em caso de ponto de paragem nessa rotina), o sistema consegue mesmo assim estabelecer a história das chamadas de funções porque verifica a que rotina a instrução em que parou pertence e que o último contexto activo não é o dessa rotina, mas sim da rotina a que pertence a instrução cujo endereço está nessa altura no RT. Mas isso já é funcionalidade do ambiente de desenvolvimento (não é o programa que faz esta reconstituição), que tem de conhecer os detalhes do processador e de como o compilador gera código-máquina para esse processador.

### SIMULAÇÃO – CONTEXTOS DE ROTINAS

Esta simulação ilustra o funcionamento não apenas dos contextos de rotinas mas também das variáveis locais e dos parâmetros, usando um programa simples mas adequado a este fim (não reproduzido aqui por limitações de espaço). Os aspectos cobertos incluem os seguintes:

- Inicialização do SP e do FP;
- Acompanhamento passo a passo da criação e destruição de um contexto;
- Paragem do programa e visualização da pilha, com navegação entre contextos;
- Demonstração do uso de variáveis locais;
- Demonstração da passagem de parâmetros e do resultado.

#### 5.7.3.5 RECURSIVIDADE

É perfeitamente possível uma rotina chamar-se a ela própria, caso em que se diz que a chamada é recursiva. A recursividade pode ser:

- Directa – se a rotina se chamar directamente;
- Indirecta – se uma rotina A chamar outra B, que directa ou indirectamente chama A (por exemplo, A chama B, que chama C, que chama A).

As chamadas recursivas podem consumir bastante espaço na pilha. Tudo depende do número de chamadas pendentes antes de começarem a regressar. Deste ponto de vista, a recursividade pode ser:

- Finita – se a chamada recursiva estiver condicionada por um teste que permita efectuar algumas chamadas recursivas, mas que depois deixe de as fazer, permitindo a todas as chamadas pendentes retornarem. A Tabela 5.27 apresenta um exemplo;
- Infinita – se não houver nenhum mecanismo que interrompa as chamadas recursivas. Este caso só pode ser um erro e a pilha é consumida por sucessivos CALLS até que o seu espaço se esgota. A única solução, num caso destes, é parar o programa.

Há muitos exemplos de algoritmos com chamadas recursivas, mas o exemplo canónico é um dos mais simples é o cálculo do factorial de um número natural<sup>46</sup> n, representado por  $n!$  e cujo valor pode ser obtido pela multiplicação de n e de todos os números naturais menores que ele. Ou seja,

$$n! = n \times (n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1$$

A Tabela 5.27 apresenta duas implementações de uma função em C e da rotina em linguagem Assembly que lhe corresponde para efectuar o cálculo do factorial de um número, de forma iterativa (com um ciclo) e recursiva (com chamadas sucessivas).

Tabela 5.28 ilustra o funcionamento da recursividade em linguagem Assembly, mostrando as sucessivas invocações da rotina factorial. Apenas as instruções executadas são mostradas e pela ordem temporal de execução. Os valores dos registos R1, R2 e SP representados após a execução da instrução da linha respetiva permitem acompanhar a evolução do programa. Essencialmente, em cada chamada à rotina factorial é apenas feito o teste ao valor de n (R1) e a chamada recursiva para calcular o factorial de n-1. Esta rotina é sucessivamente chamada até R1 ser apenas 1, altura em que as invocações começam a regressar, e é aí que o cálculo do factorial é feito. Note-se que:

- R1 é guardado na pilha antes de se efectuar cada chamada recursiva e só é reposto quando essa chamada retorna, após o factorial respectivo ter sido calculado;
- Ao contrário, R2 é usado de forma transversal a todas as chamadas à rotina factorial para acumular os sucessivos produtos, de forma bastante semelhante à versão iterativa;
- Sem o teste a R1, a recursividade seria infinita. O programa consumiria a pilha (o valor de SP iria sucessivamente ficando menor) até à exaustão. Este erro obriga a terminar o programa, pois não há recuperação possível (só corrigindo o erro e recompilando);

<sup>45</sup> Normalmente designado call stack, ou pilha de chamada de rotinas.

<sup>46</sup> Número inteiro maior que zero.

| FUNÇÃO EM C                                                                                                                                                     | ROTAÇÃO EM ASSEMBLY                                                                                                                                                                                                                                                                                                                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>int factorial (int n) {     int produto;     int i;     produto = 1;     for (i=n; i&gt;1; i--)         produto = produto * i;     return produto; }</pre> | <pre>factorial: ; o valor de retorno sai no R1 ; produto usa o R2 ; i usa o R3     MOV R2, 1; ; produto = 1;     ciclo: CMP R3, 1; ; i=n;             JLE fim; ; se não, sai do ciclo             MULT R2, R3; ; produto = produto * i;             SUB R3, 1; ; i--             JMP ciclo; ; continua o ciclo fim: RET ; retorna da função</pre> |

Tabela 5.27 - Versões iterativa (em cima) e recursiva da função factorial, em C, e as instruções em linguagem assembly que lhe correspondem

- A profundidade máxima da pilha atingida depende do valor inicial de n, tal como se pode depreender facilmente da coluna da direita da Tabela 5.28. Cada invocação da rotina factorial consome duas palavras da pilha, com exceção da primeira, que não precisa de guardar o R1, e da última (com n=1), que pára a recursividade, pelo que o cálculo do factorial de n exige que a pilha tenha no mínimo  $2 * (n-1) + 1$  palavras (7, neste exemplo). Isto significa que mesmo com o programa correcto a pilha se pode esgotar, se n for suficientemente alto. Nestas circunstâncias, deve usar-se a versão iterativa da rotina factorial, na Tabela 5.27, que não tem este problema.

### SIMULAÇÃO – RECURSIVIDADE

Esta simulação ilustra o funcionamento da invocação recursiva de uma rotina em linguagem assembly, usando a versão recursiva do programa da Tabela 5.27 e tendo em conta a Tabela 5.28. Os aspectos cobertos incluem os seguintes:

- Execução do programa passo a passo;
- Evolução dos valores dos registos relevantes e da pilha;

| DESCRICAÇÃO                                                    | INSTRUÇÃO                                                 | R1    | R2    | SP | PROFOUNDADE DA FILHA |
|----------------------------------------------------------------|-----------------------------------------------------------|-------|-------|----|----------------------|
| Programa que chama a rotina factorial a 1.ª vez, factorial (4) | MOV SP, 1000H<br>MOV RL, 4<br>CALL factorial              | 4     | 1000H | 0  | 0FFEH                |
| 1.ª invocação (n = 4) (parte inicial)                          | factorial:<br>CMP RL, 1<br>JGT fact                       | 3     | 0FFCH | 2  | 0FFAH                |
| 2.ª invocação (n = 3) (parte inicial)                          | fact: PUSH RL<br>SUB RL, 1<br>CALL factorial              | 2     | 0FFBH | 4  | 0FFAH                |
| 3.ª invocação (n = 2) (parte inicial)                          | factorial:<br>CMP RL, 1<br>JGT fact                       | 1     | 0FFAH | 6  | 0FFAH                |
| 4.ª invocação (n = 1) (fim da recursividade)                   | factorial:<br>CMP RL, 1<br>JGT fact                       | 1     | 0FFAH | 6  | 0FFAH                |
| 3.ª invocação (n = 2) (conclusão)                              | POP RL<br>MULT RL, R2, R1                                 | 2     | 0FF6H | 5  | 0FFAH                |
| 2.ª invocação (n = 3) (conclusão)                              | POP RL<br>MULT RL, R2, R1                                 | 3     | 0FFAH | 3  | 0FFAH                |
| 1.ª invocação (n = 4) (conclusão)                              | POP RL<br>MULT RL, R2, R1                                 | 4     | 0FFCH | 2  | 0FFAH                |
| Programa que chama a rotina factorial a 1.ª vez (conclusão)    | instruções seguintes. Valor do factorial disponível em R2 | 1000H | 0     | 0  | 0                    |

Tabela 5.28 - Execução temporal das instruções da versão recursiva da função factorial, em linguagem assembly. Apenas as alterações de valores dos registos estão indicadas (conclusão)

**ESSENCIAL**

- O funcionamento da pilha está intimamente ligado ao mecanismo de chama e retorna das rotinas. É uma estrutura que cresce e diminui quando se chama ou uma rotina é retornada, respectivamente;
- A chamada de uma rotina pode implicar as seguintes utilizações da pilha:
  - Guarda de registos que a rotina vai usar mas cujos valores devem ser preservados e recuperados mal a rotina retorne. Como métrica geral, uma rotina deve fazer PUSH de todos os registos que usar (para além dos que são parâmetros ou resultados) quando inicia e POP dos mesmos por ordem inversa, antes de retornar;
  - Variáveis locais, que correspondem a espaço alocado obrigatoriamente na pilha pois a mesma função pode chamar-se a ela própria (recursividade), directa ou indirectamente, e por conseguinte cada chamada tem de ter a sua própria cópia das variáveis locais;
  - Parâmetros ou resultados, passados pela pilha. Pouco frequente, pois é mais simples e mais eficiente fazer a passagem por registos;
  - Ligação entre contextos de chamada de funções, guardando na pilha um registo (FP, ou Frame Pointer) que permite a uma função aceder de forma estável às suas variáveis locais e parâmetros na pilha (o SP é uma denisão) para permitir este acesso sem ser via PUSH ou POP;
  - O manuseamento da pilha requer muito cuidado: Os PUSHs e os POPS devem estar emparelhados e por ordem inversa. Qualquer falha em fazer um PUSH ou POP faz com que a partir daí todas as operações de recuperação de informação da pilha produzam valores errados. Uma troca de ordem faz trocar os valores;
  - Um erro típico consiste em retornar de uma rotina a partir de vários locais, enquanto nem todos fazem POPS de forma consistente (isto é, muito provável, quando se fazem alterações e se muda o conjunto de registos). Guardar/recuperar, pois é fácil alterar num lado e esquecer de fazer o mesmo nouro);

## 5.8 GESTÃO DOS DADOS

### 5.8.1 QUANDO OS REGISTOS NÃO CHEGAM

O número de registos de um processador é limitado e sempre pequeno para as necessidades (o ideal era ter toda a memória dentro do processador, em registos, mas tal não é tecnologicamente viável). Ao gerar o código-máquina, os compiladores tentam gerir os registos que houver da melhor forma possível, reutilizando cada um dos registos sempre

que ele deixar de ser usado numa dada situação (isto é, o valor que ele memoriza deixar de ser necessário).<sup>47</sup>

No entanto, há dois casos fundamentais em que a única solução possível é guardar os valores de um ou mais registos em memória:

1. Quando o número de registos não chega para memorizar todos os valores que um dado conjunto de instruções necessita numa dada altura. Os valores guardados em memória são tipicamente os menos usados nessa altura, sendo lidos novamente para registos quando forem necessários (altura em que outros valores serão memorizados em memória, por troca);
2. Quando uma rotina A chama outra B que usa o mesmo registo que a rotina A estava a usar.<sup>48</sup> Isto pode acontecer mesmo que ainda haja registos livres.

O problema básico é sempre o mesmo. O programa está a manipular  $N$  valores, mas o processador só dispõe de  $P$  registos, em que  $N > P$ . Portanto, a única solução possível é guardar pelo menos  $N-P$  valores em memória.

A terminologia anglo-saxónica usa um termo curioso para esta operação, *register spilling*, que significa "entornar registos", como se os valores fossem água e o recipiente (o banco de registos) enchesse e fosse necessário entornar alguns para a memória.

Há duas formas básicas de implementar esta operação, com domínios de aplicabilidade relacionados com as duas situações anteriores:

1. Em zona de memória explícita, com operações de acesso à memória indicando o endereço (MOV). O compilador (ou programador) indica explicitamente qual ou quais as localizações envolvidas na transferência de ou para memória (os registos e os endereços);
2. Na pilha, com operações de PUSH e POP. Tem as seguintes vantagens:
  - a) ser mais geral (suporta recursividade, por exemplo);
  - b) aproveitar o mecanismo de chamada/retorno das rotinas, reduzindo a complexidade da operação;
  - c) usar um endereço de memória já carregado num registo (o SP), o que evita o carregamento de um outro registo com o endereço da posição a utilizar.

### 5.8.2 CÁLCULO DE EXPRESSES

O cálculo de expressões numa linguagem de alto nível levanta o problema do armazenamento dos valores intermédios. A Tabela 5.29 ilustra a conversão para linguagem

<sup>47</sup> Idêntica situação podem enfrentar os programadores de linguagem assembly, embora tipicamente a programação manual a um nível tão baixo não envolva programas muito complexos.

<sup>48</sup> Na realidade, isto não se aplica apenas a rotinas, mas também a blocos de instruções imbricadas (Secção 5.8.3).

*assembly* do cálculo de uma expressão em linguagem C. Embora na linguagem de alto nível, sejam apenas referidas três variáveis, o cálculo do valor da variável z obriga a introduzir mais variáveis intermédias para guardar os valores das parcelas da expressão. Além disso, é preciso não esquecer que a semântica das expressões nas linguagens de alto nível assume que os operandos não são alterados (ao contrário do que sucede normalmente nas instruções *assembly* de dois operandos, em que o primeiro operando é substituído pelo resultado).

| CÓDIGO EM C                                                    | CÓDIGO EM ASSEMBLY                                                                                                                                                                                                                                                                                                                                       |
|----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> int x, y, z; x = 3; y = 1; z = 3 + (y+2) * (4+x); </pre> | <pre> ; assume-se que se usa R1, R2 e R3 para x, y e z, ; respectivamente MOV R1, 3 ; x = 3; MOV R2, 1 ; y = 1; MOV R4, R2 ; cópia de y, para não ser ; alterado ADD R4, 2 ; y + 2 MOV R5, R1 ; cópia de x, para não ser ; alterado ADD R5, 4 ; 4 + x MUL R4, R5 ; (y+2) * (4+x) ADD R4, 3 ; 3 + (y+2) * (4+x) MOV R3, R4 ; z = 3 + (y+2) * (4+x) </pre> |

Tabela 5.29 - Cálculo de uma expressão em C e as instruções em linguagem *assembly* que lhe correspondem

Verifica-se assim que o número de variáveis realmente necessárias é superior ao número das variáveis definidas pelo programador, o que aumenta o problema da limitação do número de registos de um processador.

Note-se que as instruções em linguagem *assembly* da Tabela 5.29 usam os registos R1 a R5 sem qualquer preocupação sobre os valores anteriores destes registos. Se estes forem relevantes, terão de se usar outros registos disponíveis cujos valores não sejam necessários ou, caso não existam, os registos usados terão de ser guardados previamente em memória e restaurados após estas instruções, tipicamente usando a pilha.

### 5.8.3 EXECUÇÃO DE INSTRUÇÕES IMBRICADAS

As linguagens de alto nível suportam instruções compostas de outras instruções, incluindo declaração de variáveis locais a uma dada instrução, em estruturas de controlo arbitrariamente complexas. Por exemplo, é perfeitamente possível que as instruções em C da Tabela 5.29 formem o corpo de uma instrução if que faz parte de uma função que ainda pode ser recursiva.

Nestas condições, o compilador analisa o caso concreto e, se houver necessidade, faz PUSH dos registos que usar dentro de um bloco de instruções no início desse bloco e POP desses registos no fim desse bloco.

O corpo de uma função pode ser considerado como um bloco de instruções, com as suas próprias variáveis locais, que pode ser invocado a partir de vários locais do programa e suporta parâmetros. A secção 5.7.3.2 já discutiu a implementação das variáveis locais neste contexto.

## 5.8.4 TABELAS

As tabelas são provavelmente as estruturas de dados mais usadas em programação de computadores, pelo que se justifica uma análise da forma mais adequada de as criar, gerir e aceder em linguagem *assembly*.

### 5.8.4.1 TABELAS DE UMA SÓ DIMENSÃO

As tabelas mais comuns têm apenas uma dimensão, sendo constituídas por uma sequência linear e contígua de elementos na memória. São também normalmente designadas vectores, embora do ponto de vista matemático tal esteja incorrecto<sup>49</sup> e uma designação mais correcta seja matriz monodimensional. No entanto, este é um dos casos em que o rigor matemático não traz vantagens face à terminologia corrente, já muito implantada neste contexto, razão pela qual este livro também a adopta.

Em linguagem *assembly* do PEPE há três formas básicas de declarar tabelas de uma dimensão (secção 5.5.2.3):

- Elementos de uma palavra:
  - Directiva TABLE, que reserva o número especificado de palavras de memória sem as inicializar. Os vários elementos da tabela deverão ser inicializados antes de serem usados;
  - Directiva WORD, que reserva e inicializa com o valor especificado uma palavra de memória. Para construir uma tabela basta especificar várias directivas WORD consecutivas.
  - Elementos de um byte:
  - Directiva STRING, que permite especificar várias constantes de um byte, produzindo portanto uma seqüência de bytes. Uma só directiva STRING permite especificar toda uma tabela de bytes. Também é possível especificar várias directivas STRING consecutivas, e considerar que a tabela é constituída pela totalidade de bytes especificados, pois ficarão localizados na memória de forma contígua.
- O mecanismo básico de utilização de uma tabela passa por considerar uma dada zona de memória e o endereço do primeiro byte dessa zona, que é encarado como a base da

<sup>49</sup> Um vector é um segmento de recta orientado num espaço n-dimensional, definido por dois pontos (origem e fim) de N coordenadas cada.

tabela. Um elemento de uma tabela pode ser accedido somando um índice (número inteiro) ao valor da base para determinar o endereço onde esse elemento está localizado em memória, tal como já ilustrado pela Fig. 4.8, na página 224.

**NOTA**

Normalmente, encara-se a base de uma tabela como o endereço mais baixo dessa tabela, para que os restantes endereços em que a tabela se localize possam ser obtidos somando um índice positivo à base. Esta convenção é a usada normalmente pelas linguagens de alto nível. No entanto, em linguagem assembly a flexibilidade é total e nada obsta a que a base seja em qualquer endereço da tabela (é apenas um ponto de referência), caso em que o índice poderá ter um valor positivo, nulo ou negativo.

No entanto, é fundamental perceber que este índice conta bytes e portanto só coincide com o índice da tabela (o que conta elementos) no caso das tabelas de elementos de uma byte (criadas com a directiva STRNG, por exemplo). No caso das tabelas de elementos de 16 bits, cada elemento tem 2 bytes e portanto para determinar o endereço de um elemento tem de se multiplicar o índice (em elementos) por 2 antes de se somar ao endereço de base da tabela.

O Programa 4.3, na página 225 já ilustra este aspecto fundamental, mas não explica como declarar a tabela (assume apenas que a tabela já existe, inicializada e com um dado endereço de base). O Programa 56 dá outro exemplo de manipulação de tabelas (procura de um elemento), em que a tabela é construída (espaço reservado e inicializado) por directivas WORD.

NOTAS SOBRE ESTE PROGRAMA

- Não é preciso especificar etiquetas para todas as directivas WORD. Basta a primeira, para haver uma forma de acedermos ao primeiro elemento da tabela (ende-reço de base). Os restantes elementos podem ser accedidos somando um índice à base da tabela;

  - A constante do 4.º elemento da tabela, 0CDEFH, tem de ser especificada com um zero à esquerda, para que o assembler não interprete o símbolo CDEFH como um identificador (não declarado, neste caso). As constantes literais numéricas têm de começar com um dígito. A constante do 1.º elemento não precisava do 0 à esquerda (está lá apenas para todos os elementos terem os 16 bits especificados);
  - O índice de acesso (em elementos) à tabela é duplicado para efectuar o acesso propriamente dito, pois os endereços são de *byte* e os elementos são de 16 bits;
  - Testar se o índice já chegou ao limite da tabela logo no inicio destina-se a con-templar o caso de tabelas vazias (com 0 elementos). Às vezes reserva-se o espaço para uma tabela com N palavras, mas só P elementos (com  $P < N$ ) lá estão armaze-nados (obrigatoriamente nas P primeiras posições, para a rotina funcionar bem). Neste caso, o valor a considerar como 2.º parâmetro para a rotina seria P e não N;
  - O registo R0 é reutilizado na instrução de acesso à memória. Não há qualquer conflito ou erro. Primeiro é calculado o endereço a acceder ( $R1+R0$ ) e só depois é feita a leitura da memória com escrita no R0. Esta reutilização permite não gastar mais um registo, evitando mais um PUSH no início da rotina e POP no fim.

```

MOV    R4, 0      ; inicializa índice de procura na tabela
CMP    R4, R2      ; já chegou ao fim?
JZ     ; não
        MOV    R0, R4      ; termina sem ter achado o elemento
        ADD    R0, R0      ; copia índice para não o destruir
        ADD    R0, [R1 + R0]  ; duplica índice porque os endereços são
        MOV    R0, [R1 + R0]  ; lê um elemento da tabela (no endereço
        CMP    R0, R3      ; este elemento é a chave procurada?
        JZ     ; acaba
        ADD    R4, 1      ; termina, já achou o elemento no índice
        ADD    R4, 1      ; índice do elemento seguinte
        ADD    R4, 1      ; ainda não achou, passa ao elemento seguinte
        MOV    R4, -1      ; indica que não achou
        POP    R0      ; repõe registo
RET

```

**Programa 5.6 - Exemplo de acesso a uma tabela com elementos de 16 bits declarada e inicializada com a directiva WORD**

```

MOV    R4, 0          ; inicializa índice de procura na tabela
CMP    R4, R2         ; já chegou ao fim?
JZ     ; termina sem ter achado o elemento
MOV    R0, R4         ; copia índice para não o destruir
ADD    R0, R0         ; duplica índice porque os endereços são em bytes
MOV    R0, [R1 + R0]   ; lê um elemento da tabela (no endereço R1+R0)
CMP    R0, R3         ; este elemento é a chave procurada?
JZ     ; termina, já achou o elemento no índice dado por R4
ADD    R4, 1           ; índice do elemento seguinte
JMP    maisUm         ; ainda não achou, passa ao elemento seguinte
MOV    R4, -1          ; indica que não achou
POP    R0             ; repõe registro
RET

```

**Programa 5.6 - Exemplo de acesso a uma tabela com elementos de 16 bits, declarada e inicializada com a directiva `word`**

O Programa 5.7 permite ilustrar o uso da directiva TABLE para reservar espaço para uma tabela sem o inicializar, o que obriga a fazê-lo depois explicitamente com instruções. Imediatamente, tabelas de valores constantes são mais fáceis de construir com directivas TABLE. A directiva TABLE é mais útil quando se pretende reservar espaço para conter uma tabela de variáveis, cujos valores vão sendo escritos pelo programa de acordo com o seu algoritmo.

```

MOV    R4, 0          ; inicializa índice de procura na tabela
CMP    R4, R2         ; já chegou ao fim?
JZ     ; termina sem ter achado o elemento
MOV    R0, R4         ; copia índice para não o destruir
ADD    R0, R0         ; duplica índice porque os endereços são em bytes
MOV    R0, [R1 + R0]   ; lê um elemento da tabela (no endereço R1+R0)
CMP    R0, R3         ; este elemento é a chave procurada?
JZ     ; termina, já achou o elemento no índice dado por R4
ADD    R4, 1           ; índice do elemento seguinte
JMP    maisUm         ; ainda não achou, passa ao elemento seguinte
MOV    R4, -1          ; indica que não achou
POP    R0             ; repõe registro
RET

```

**Programa 5.6 - Exemplo de acesso a uma tabela com elementos de 16 bits, declarada e inicializada com a directiva `word`**

O Programa 5.7 permite ilustrar o uso da directiva TABLE para reservar espaço para uma tabela sem o inicializar, o que obriga a fazê-lo depois explicitamente com instruções estaticamente, tabelas de valores constantes são mais fáceis de construir com directivas **TABLE**. A directiva TABLE é mais útil quando se pretende reservar espaço para conter uma tabela de variáveis, cujos valores vão sendo escritos pelo programa de acordo com o seu algoritmo.

```
tam EQU 4 ; número de elementos de 16 bits na tabela (>=0)
```

```
elem1 EQU 0123H ; 1.º elemento da tabela
elem2 EQU 4567H ; 2.º elemento da tabela
elem3 EQU 89ABH ; 3.º elemento da tabela
elem4 EQU OCDEFH ; 4.º elemento da tabela
```

```
chave EQU elem3 ; valor a procurar na tabela
pilha EQU 2000H ; valor inicial do SP
```

```
PLACE tam base: TABLE tam ; localiza bloco de dados
; reserva espaço para uma tabela com tam elementos
; de 16 bits cada (sem o inicializar)
```

```
PLACE 0000H inicio: MOV SP, pilha ; localiza SP
MOV R0, base ; endereço do 1.º elemento da tabela
MOV R1, elem1 ; 1.º elemento
MOV [R0], R1 ; inicializa 1.º elemento na tabela
```

```
MOV R2, elem2 ; 2.º elemento
MOV [R0+2], R1 ; inicializa 2.º elemento na tabela
MOV R3, elem3 ; 3.º elemento
MOV [R0+4], R1 ; inicializa 3.º elemento na tabela
MOV R4, elem4 ; 4.º elemento
MOV [R0+6], R1 ; inicializa 4.º elemento na tabela
```

```
MOV R1, base ; 1.º parâmetro (base da tabela)
MOV R2, tam ; 2.º parâmetro (número de elementos da tabela)
MOV R3, chave ; 3.º parâmetro (valor a procurar)
```

```
fim: JMP fim ; fim do programa
;***** ; ProcuraPalavra - Rotina omitida por ser idêntica à do Programa 5.6.
```

### Programa 5.7 - Alternativa ao Programa 5.6 com declaração da tabela com a directiva TABLE e inicialização dos elementos por instruções

As tabelas de *bytes* podem ser declaradas com a directiva STRING. Para melhor comparação, o Programa 5.8 apresenta o mesmo exemplo que o Programa 5.6, mas agora no reino dos elementos de 8 bits. As diferenças fundamentais face ao Programa 5.6 são as seguintes:

- A directiva STRING permite logo especificar várias constantes (seria equivalente especificar uma constante por directiva, desde que estivessem todas consecutivas e pela mesma ordem);
- O índice em elementos é igual ao índice em endereços, pois ambos se referem a bytes (ao contrário do Programa 5.6, não é preciso duplicar o índice em elementos para obter o índice em bytes);
- O acesso à tabela faz-se com uma instrução MOV e não MOV, para que apenas o byte em causa seja acedido.

### NOTA

As directivas TABLE, WORD e STRING permitem reservar espaço para uma tabela, garantindo que nenhum outro dado ou instrução seja declarado nesse espaço. No entanto, nada impede que se considere qualquer endereço como base da tabela (como aliás foi feito no Programa 4.3, na página 225), incluindo no espaço do código. Em linguagem assembly (quase) tudo é possível. Em particular, ler ou escrever instruções na memória como se fossem dados é algo fundamental, usado por exemplo para carregar em memória programas armazenados no disco, e é mesmo um dos pontos fundamentais que a arquitetura de von Neumann introduziu e garantiu o seu sucesso (secção 1.4).

```
tam EQU 8 ; número de elementos de 8 bits na tabela (>=0)
```

```
chave EQU 0ABH ; valor inicial do SP
```

```
PLACE 1000H base: STRING 01H, 23H, 45H, 67H, 89H, 0ABH, 0CDH, 0EFH ; elementos da
```

```
PLACE 0000H inicio: MOV SP, pilha ; localiza SP
MOV R1, base ; 1.º parâmetro (base da tabela)
MOV R2, tam ; 2.º parâmetro (número de elementos da tabela)
MOV R3, chave ; 3.º parâmetro (valor a procurar)
```

```
Entradas - R1 - endereço do 1.º elemento da tabela (base). Pode ser ímpar.
R2 - número de elementos de 8 bits na tabela (>=0)
R3 - chave (valor de 8 bits a procurar)
```

```
Saídas - R4 - índice do primeiro elemento que encontrar igual à chave
(-1 se não encontrar)
```

```
;***** ; ProcuraByte - Procura um elemento de 8 bits numa tabela, começando pelo
```

```
primeiro elemento da tabela
Entradas - R1 - endereço do 1.º elemento da tabela (base). Pode ser ímpar.
R2 - número de elementos de 8 bits na tabela (>=0)
```

```
R3 - chave (valor de 8 bits a procurar)
```

```
ProcuraByte:
PUSH R0 ; guarda registo
PUSH R1 ; guarda registo (para não alterar o parâmetro R1)
MOV R4, 0 ; inicializa índice de procura na tabela
```

```
maisUm: CMP R4, R2 ; já chegou ao fim?
JZ acaba ; termina, já achou o elemento no índice dado por R4
```

```
JZ nãoHá ; termina sem ter achado o elemento
; 1º um elemento da tabela
; este elemento é a chave procurada?
```

```
CMP R0, R3 ; termina, já achou o elemento no índice dado por R4
JZ acaba ; endereço do elemento (byte) seguinte
ADD R4, 1 ; índice do elemento (byte) seguinte
ADD R4, 1 ; índice não achou, passa ao elemento seguinte
JMP maisUm ; indica que não achou
acaba: MOV R4, -1 ; repõe registo
POP R1
POP R0
RET
```

Programa 5.8 - Exemplo de acesso a uma tabela com elementos de 8 bits, declarada e inicializada com a directiva STRING

**SIMULAÇÃO – TABELAS DE UMA SÓ DIMENSÃO**

Esta simulação ilustra o funcionamento das tabelas monodimensionais, tornando os programas desta secção como base. Os aspectos cobertos incluem os seguintes:

- Funcionamento das directivas TABLE, WORD e STRING;
- Disposição dos elementos das tabelas em memória;
- Cálculo dos endereços dos elementos de 8 e de 16 bits;
- Acesso à memória em 8 e 16 bits;
- Execução do programa passo a passo;
- Funcionamento dos programas e da evolução dos valores dos registos relevantes.

**5.8.4.2 TABELAS MULTIDIMENSIONAIS**

Embora menos comuns, há situações em que se precisam de tabelas de mais de uma dimensão, tipicamente duas. Casos mais complexos já não são normalmente implementados em assembly, mas sim numa linguagem de alto nível. Mas o princípio de funcionamento é exactamente o mesmo. Esta secção exemplifica o caso de tabelas de duas dimensões, sendo fácil de extrapolar para outros casos.

A linguagem assembly dos processadores contempla normalmente apenas mecanismos para acesso a tabelas de uma dimensão, com instruções de acesso à memória que permitem somar uma base a um índice (MOV R0, [R1+R2], por exemplo). Isto é consistente com o modelo básico das memórias, que consiste numa tabela de células de uma só dimensão. Isto significa que o problema não é apenas o acesso à tabela multidimensional, mas também a sua própria implantação numa memória que só tem uma dimensão.

Para implementar uma tabela multidimensional na memória, o truque básico é “linearizar” essa tabela, colocando linhas (ou colunas) umas a seguir às outras. A Fig. 5.10 ilustra este procedimento para uma tabela de três linhas por quatro colunas. A linearização por linhas (Fig. 5.10b) é normalmente equivalente, mas o algoritmo por colunas (Fig. 5.10d) pode ficar mais simples se se usar uma técnica ou outra do programa que acede à tabela pode ficar mais simples se se usar uma técnica ou outra ou haver outras considerações (desempenho, por exemplo)<sup>50</sup>, que ajudem a decidir qual a melhor forma.

**NOTA** Alguns processadores mais antigos, como o 8086 (e por compatibilidade os Pentiums dos PCs actuais), têm instruções de acesso à memória que suportam uma base e dois índices, o que permite suportar directamente em hardware tabelas de duas dimensões. No entanto, não é um mecanismo geral (não dá para todos os casos) e esta complexidade adicional tem o seu custo em termos de complexidade do hardware, que por sua vez se traduz em frequências de relógio mais baixas. Os microprocessadores mais modernos suportam apenas o mecanismo básico de base + índice.

Naturalmente, quem acede tem de saber como se faz a implementação, para usar a mesma técnica, senão pode obter os elementos errados. Para aceder à tabela, podem encarar-se as tabelas linearizadas de duas perspectivas distintas:

- Como uma só tabela, com uma base e um índice únicos. Neste caso, para aceder a um elemento é preciso calcular em que índice da tabela linearizada ele está, tendo em conta se a linearização foi feita por linhas ou por colunas;
- Como várias tabelas consecutivas, cada uma correspondente a uma linha (linearização por linhas) ou a uma coluna (linearização por colunas). Neste caso, determina-se primeiro o endereço de base de cada linha ou coluna e depois acede-se ao elemento em causa com o índice que o elemento tinha na tabela original dentro dessa linha ou coluna.

|     |  |
|-----|--|
| (a) |  |
| (b) |  |
| (c) |  |
| (d) |  |

Fig. 5.10 - Implementação de uma tabela de duas dimensões em memória.  
(a) - Tabela em duas dimensões; (b) - Tabela linearizada por linhas; (c) - Índice dos elementos nas tabelas linearizadas; (d) - Tabela linearizada por colunas

Estas duas perspectivas são equivalentes. A maior diferença é que na primeira se fazem as contas todas em cada acesso (adequada em acessos aleatórios), enquanto que na segunda se assume que a base de cada tabela (linha ou coluna) é armazenada num registo e depois se reutiliza em vários acessos, em particular se a tabela for iterada numa dada linha ou coluna (com uma rotina de procura de um elemento, por exemplo). É uma questão de optimização, que poderá interessar ou não, dependendo do programa.

Além destas variantes ainda há o caso da dimensão (8 ou 16 bits) de cada elemento. Se quisermos fazer uma rotina para aceder ao elemento  $E_{ij}$  ( $i$  – índice da linha,  $j$  – índice da coluna) da tabela, a informação de que precisamos é a seguinte:

<sup>50</sup> Em particular para tabelas de grandes dimensões. A página 660 permite detalhar este aspecto.

- Dimensão dos elementos (*byte* ou palavra, 8 ou 16 bits);
- B – Endereço de base da tabela (linearizada). Tem de ser par se os elementos forem de 16 bits;
- I – Índice da linha do elemento (começando em zero);
- J – Índice da coluna do elemento (começando em zero);
- D – Tamanho máximo da dimensão usada para linearizar a tabela (dimensão da linha na linearização por linhas, por exemplo).

A Tabela 5.30 mostra como calcular o endereço de memória do elemento a aceder.

| BITS | PERSPECTIVA DA TABELA | LINEARIZAÇÃO |               |             |               |
|------|-----------------------|--------------|---------------|-------------|---------------|
|      |                       | POR LINHAS   | POR COLUNAS   | BASE        | ÍNDICE        |
| B    | ÚNICA                 | B            | B             | $I*D + J$   | B             |
|      | SEQUÊNCIA DE TABELAS  | $B + I*D$    | J             | B           | $B + J*D$     |
| 16   | ÚNICA                 | B            | $2*I*D + 2*j$ | B           | $2*I*D + 2*j$ |
|      | SEQUÊNCIA DE TABELAS  | $B + 2*I*D$  | $2*j$         | $B + 2*j*D$ | $2*I$         |

Tabela 5.30 - Cálculo do endereço de um elemento na tabela linearizada, que é o mesmo (base + índice) seja esta encarada como uma só tabela ou como uma sequência de tabelas

A secção 4.10.4.5, na página 227, apresenta um exemplo de acesso a uma tabela bidimensional, composta por fichas de empregados, em que cada ficha tem quatro elementos de 16 bits. Nessa secção assumia-se apenas que a tabela já existia. O Programa 5.9 recupera este exemplo, mas agora apresenta a construção da tabela, usando directivas word. Os dados sobre os empregados já foram indicados na Tabela 4.18, na página 228.

Este programa ilustra o caso em que a tabela linearizada é vista como uma sequência de tabelas (as fichas). A rotina que calcula o encargo mensal de base de cada ficha com um bónus igual à idade) já recebe como parâmetro o endereço de base de cada ficha. Aliás, a própria construção da tabela já é feita não em forma matricial mas sim de forma linearizada, ficha a ficha.

```

numEmp EQU 4 ; número de empregados (e de fichas)
salario EQU 6 ; índice em bytes (dentro da ficha) do campo salário
idade EQU 4 ; índice em bytes (dentro da ficha) do campo idade
tam EQU 8 ; tamanho em bytes de cada ficha
pilha EQU 2000H ; valor inicial do SP

PLACE 1000H ; localiza bloco de dados
base: word 3029 ; n.º mecanográfico (empregado 1)
word 207 ; extensão telefónica
word 34 ; idade
word 1250 ; n.º mecanográfico (empregado 2)
word 1978 ; extensão telefónica
word 225 ; idade
word 23 ; idade

```

```

PLACE 0000H ; localiza bloco de instruções
início: MOV SP, pilha ; inicializa SP
        MOV R1, base ; n.º mecanográfico (empregado 3)
        MOV R4, numEmp ; extensão telefónica
        CALL EncargoTotal ; calcula encargo total dos empregados
fim:    JMP fim ; acabou. R3 contém a soma de todos os encargos
;***** ; inicializa R3 para zero
;***** ; EncargoTotal - Calcula o encargo mensal de todos os empregados
; Entradas - R1 - Endereço de base da tabela com informação sobre os empregados
; R4 - Número de fichas dos empregados
; Sáídas - R3 - Encargo mensal de todos os empregados
; Destroi - R1, R2, R4
;***** ; calcula encargo mensal de todos os empregados
;***** ; EncargoTotal:
MOV R3, 0 ; inicializa soma dos encargos salariais
maiJuma: CMP R4, 0 ; se não acaba
        JZ acaba ; não mais fichas para tratar?
        CALL EncargoEmp ; Calcula encargo mensal deste empregado
        ADD R3, R2 ; acumula o encargo na soma
        MOV R2, tam ; tamanho em bytes de cada ficha
        ADD R1, R2 ; obtém endereço da base da próxima ficha
        SUB R4, 1 ; menos uma ficha para tratar
        JMP maiJuma ; vai tratar da próxima ficha
acaba: RET

;***** ; EncargoEmp - Calcula o encargo mensal de um dado empregado, somando o ordenado
; Entradas - R1 - Endereço de base da ficha com informação do empregado
; Entradas - R2 - Encargo mensal do empregado
; Sáídas - R2 - Encargo mensal do empregado
;***** ; guarda registo
EncargoEmp: PUSH R0 ; guarda registo
        MOV R2, [R1 + salário] ; le o campo salário desta ficha
        MOV R0, [R1 + idade] ; le o campo idade desta ficha
        ADD R2, R0 ; acumula o bónus (igual à idade) na soma
        POP R0 ; repõe registo
        RET

Programa 5.9 - Reformulação do Programa 4.6 para exemplificar o acesso a uma tabela bidimensional com elementos de 16 bits (encarada como uma sequência de tabelas monodimensionais)
```

O Programa 5.10 ilustra a outra perspectiva indicada na Tabela 5.30, calculando o somatório dos salários de todos os empregados com acessos a toda a tabela com informação sobre os empregados como se tivesse uma simples sequência linear de elementos se tratasse. O índice do elemento a aceder tem de ser calculado completamente em cada acesso. Note-se que neste exemplo em particular não há a multiplicação por 2 referida na Tabela 5.30 porque a dimensão de cada ficha (tam) é o índice dentro de cada ficha (campo salário) já estão expressos em bytes.

```

; EQUS e WORDS omitidos por serem iguais ao Programa 5.9

PLACE 0000H ; localiza bloco de instruções
início: MOV SP, pilha ; inicializa SP
        MOV R1, base ; 1.º ficha (base da tabela)
        MOV R2, numEmp ; número de fichas da tabela
        CALL Salariototal ; Calcula encargo mensal total
fim:   JMP fim ; acabou. R3 contém a soma de todos os salários

;***** Salariototal - Calcula o total mensal dos salários dos empregados
; Entradas - R1 - Endereço de base da tabela com informação sobre os empregados
;           R2 - Número de empregados
;           R3 - Encargo mensal total
;***** Saláriototal:
SalarioTotal: ; guarda registos
PUSH R0
PUSH R1
PUSH R2
MOV R3, 0 ; inicializa soma dos salários
MOV R4, 0 ; inicializa contador das fichas de empregado
maisuma: CMP R4, R2 ; já tratou das fichas todas?
        JZ acaba ; se já não há mais fichas para tratar, acaba
        MOV R0, tam ; dimensão em bytes de cada ficha
        MUL R0, R4 ; dimensão em bytes das fichas anteriores
        ADD R0, salário ; índice em bytes (na tabela) do campo salário
        MOV R0, [R1 + R0] ; 16 o campo salário
        ADD R3, R0 ; acumula o valor na soma
        ADD R4, 1 ; mais uma ficha já tratada
        JMP maisuma ; vai tratar da próxima ficha
acaba:  POP R4
        POP R0 ; reapega registos
RET

```

**Programa 5.10 - Exemplo de acesso a uma tabela bidimensional com elementos de 16 bits (encarada como uma só tabela monodimensional linearizada)**

O Programa 5.11 ilustra o acesso a tabelas bidimensionais de elementos de 8 bits, em que a possibilidade de a directiva STRING especificar várias constantes na mesma linha já dá um aspecto mais matricial à construção da tabela (embora na memória os elementos sejam memorizados de forma linearizada, um após outro pela ordem com que foram especificados). Este exemplo calcula o dia do mês (de um dado mês) dando o número da semana dentro do mês (0 a 4) e o dia da semana (0 a 6). Se a combinação de parâmetros da rotina não for válida, devolve 0 (de acordo com a tabela). Com os parâmetros indicados, 3 e 5, o dia do mês correspondente é 24.

|       |        |                            |                                           |
|-------|--------|----------------------------|-------------------------------------------|
| pilha | EQU    | 2000H                      | ; valor inicial do SP                     |
| PLACE | 1000H  | ; localiza bloco de dados  |                                           |
| base: | STRING | 0, 0, 0, 1, 2, 3, 4        | ; Seg., Ter., Qua., Qui., Sex., Sáb., Dom |
|       | STRING | 5, 6, 7, 8, 9, 10, 11      | ; (0), (1), (2), (3), (4), (5), (6)       |
|       | STRING | 12, 13, 14, 15, 16, 17, 18 | ; 1.ª semana (0) do mês                   |
|       | STRING | 19, 20, 21, 22, 23, 24, 25 | ; 2.ª semana (1) do mês                   |
|       | STRING | 26, 27, 28, 29, 30, 0, 0   | ; 3.ª semana (2) do mês                   |
|       | STRING | 5, 6, 7, 8, 9, 10, 11      | ; 4.ª semana (3) do mês                   |
|       | STRING | 12, 13, 14, 15, 16, 17, 18 | ; 5.ª semana (4) do mês                   |

```

;***** Salariototal - Calcula o dia do mês com base nos parâmetros
; Entradas - R1 - Endereço da base da tabela com informação sobre os dias
;           R2 - Número da semana dentro do mês (0 a 4)
;           R3 - Dia da semana (0 a 6, ou segunda a domingo)
;***** Saláriototal:
Saláriototal: ; guarda registos
PUSH R0
PUSH R1
PUSH R2
MOV R3, 0 ; inicializa soma dos salários
MOV R4, 0 ; inicializa contador das fichas de empregado
maisuma: CMP R4, R2 ; já tratou das fichas todas?
        JZ acaba ; se já não há mais fichas para tratar, acaba
        MOV R0, tam ; dimensão em bytes das fichas anteriores
        MUL R0, R4 ; dimensão em bytes (na tabela) do campo salário
        ADD R0, salário ; índice em bytes (na tabela) do campo salário
        MOV R0, [R1 + R0] ; 16 o campo salário
        ADD R3, R0 ; acumula o valor na soma
        ADD R4, 1 ; mais uma ficha já tratada
        JMP maisuma ; vai tratar da próxima ficha
acaba:  POP R4
        POP R0 ; reapega registos
RET

```

**Programa 5.11 - Exemplo de acesso a uma tabela bidimensional com elementos de 8 bits**  
(encarada como uma só tabela monodimensional linearizada)

Esta simulação ilustra o funcionamento das tabelas multidimensionais, tornando os programas desta secção como base. Os aspectos cobertos incluem os seguintes:

- Disposição dos elementos das tabelas em memória;
- Cálculo dos endereços dos elementos de 8 e de 16 bits;
- Acesso à memória em 8 e 16 bits;
- Funcionamento dos programas e da evolução dos valores dos registos relevantes.

### 5.8.4.3 TABELAS DE APONTADORES

As tabelas de apontadores são estruturas de dados comuns. São tabelas em que cada elemento é um apontador (endereço de uma variável – ver secção 5.5.4, na página 294 – ou de uma rotina) e podem ser úteis em programas de comportamento dinâmico, em que o endereço do valor a aceder ou o endereço para onde saltar é determinado em tempo de execução pelo cálculo de um valor inteiro, usado para indexar a tabela. Estas tabelas têm geralmente uma só dimensão, embora nada impeça que sejam multidimensionais, e usam-se essencialmente nos seguintes tipos de situações:

- Acesso a diversas estruturas de dados, de dimensões variadas, em que a escolha da estrutura de dados a aceder só é feita em tempo de execução. Um valor inteiro é usado como índice na tabela para obter o endereço onde está a estrutura de dados, que pode então ser acedida usando esse endereço. Não é possível usar um simples vector em que cada elemento é uma destas estruturas de dados, pois num vector todos os elementos têm de ter o mesmo tamanho. O Programa 5.12 ilustra este tipo de situação com uma rotina que conta o número de caracteres de uma mensagem de erro correspondente a um dado código (um valor inteiro). Cada mensagem de erro tem uma dimensão diferente (o terminador 00H marca o fim), pelo que se tem de montar uma tabela dos endereços iniciais de cada mensagem e não um simples vector de mensagens;
- Tabela de saltos, em que os endereços não dizem respeito a dados mas sim a instruções. Este caso pode subdividir-se em dois:
  - Decisão múltipla, em que se deve saltar (com uma instrução JMP) para uma de várias instruções. As instruções de salto condicionais só suportam duas escolhas. A secção 5.6.2.2 ilustra este caso;
  - Invocação vectorizada de funções, em que se invoca uma de várias funções, usando a instrução CALL mas em que em vez de especificar uma constante se usa um registo para indicar o endereço da rotina a invocar (Tabela 5.16, na página 327). O Programa 5.13 ilustra este caso com uma máquina de calcular rudimentar, implementada como uma rotina que recebe três operandos (os dois operandos e a operação) e invoca uma de quatro rotinas, consoante a operação em causa. Um exemplo bem mais interessante é o da invocação de rotinas de um sistema operativo (system calls), em que o utilizador não invoca directamente uma rotina de sistema mas sim uma rotina fixa passando-lhe como parâmetro o código da rotina que realmente quer invocar. Tal é feito essencialmente por motivos de protecção.<sup>51</sup> A própria tabela pode ser alterada com instruções MOV durante a execução do programa de acordo com resultados obtidos, por exemplo, o que introduz ainda mais um nível de flexibilidade.

<sup>51</sup> Se o programa do utilizador pudesse fazer CALL para qualquer endereço do código do sistema operativo, poderia facilmente executar operações incorrectas e baralhar todo o sistema. A secção 7.7.5, na página 682, dá mais detalhes sobre este aspecto.

```

pilha EQU 2000H ; valor inicial do SP
PLACE 1000H ; zona de dados
messag1: STRING "Tamanho inválido", 00H ; mensagens de erro, codificadas
messag2: STRING "Objecto não existe", 00H ; em ASCII (um byte por carácter) e
messag3: STRING "Erro", 00H ; terminadas com o byte 00H (todas
messag4: STRING "", 00H ; têm dimensão diferente)
messag5: STRING "Não aplicável", 00H

tabela: WORD mensag0 ; endereço da mensagem 0
        WORD mensag1 ; endereço da mensagem 1
        WORD mensag2 ; endereço da mensagem 2
        WORD mensag3 ; endereço da mensagem 3
        WORD mensag4 ; endereço da mensagem 4
        WORD mensag5 ; endereço da mensagem 5

PLACE 0000H ; zona de instruções
MOV SP, pilha ; limite superior da pilha
MOV RL, 1 ; código da mensagem 1
CALL Tammensagem ; conta caracteres da mensagem 1
MOV RL, 2 ; código da mensagem 2
CALL Tammensagem ; conta caracteres da mensagem 2
MOV RL, 4 ; código da mensagem 4
CALL Tammensagem ; conta caracteres da mensagem 4
CALL Tammensagem ; conta caracteres da mensagem 4
JMP fin ; fin do programa

***** ; ****
; Tammensagem - Obtém o tamanho em caracteres de uma mensagem (sem terminador)
; Entradas - RL - Código da mensagem (0 a 5)
; Saídas - R2 - Número de caracteres da cadeia
; Destroi - RL, R3
***** ; ****

tammensagem:
MOV R3, tabela ; base da tabela de apontadores
ADD RL, RL ; duplica RL, pois a tabela é de apontadores e cada
             ; elemento é um endereço que ocupa 2 bytes
MOV R1, [R3+RL] ; obtém o endereço da mensagem pretendida
CALL Conta ; conta quantos caracteres tem
RET ; já está! O resultado foi deixado por Conta no R2

***** ; ****
; Conta - Conta o número de caracteres de uma cadeia de caracteres
; Entradas - R1 - Apontador para o 1.º carácter da cadeia
; Saídas - R2 - Número de caracteres da cadeia
; Destroi - RL, R3
***** ; ****
Conta: MOV R2, 0 ; inicialmente, o número de caracteres é, zero
maisUm: MOVB R3, [RL] ; obtém próximo carácter da cadeia
        CMP R3, 0 ; já é o terminador 00H?
        JZ termina ; se sim, a contagem está feita
        ADD R2, 1 ; conta mais um carácter
próximo: ADD RL, 1 ; prepara apontador para o próximo carácter
termina: RET ; vai tratar do próximo carácter

```

<sup>51</sup> Programa 5.12 - Exemplo de utilização de uma tabela de apontadores para estruturas de dados de dimensão variável

Note-se que este programa assume que cada carácter está representado em ASCII, ocupando apenas um byte em memória e portanto não funcionaria se cada carácter estivesse representado em Unicode, ocupando dois bytes. Em particular, são relevantes as instruções da rotina Conta nas seguintes etiquetas:

- maiSum – A instrução MOV.B (secção 4.10.4.7, na página 230) é fundamental, pois lê da memória apenas o byte correspondente ao carácter pretendido, mesmo que o seu endereço (valor de R1) seja ímpar. O byte do carácter é lido para o byte de menor peso de R3, cujo byte de maior peso fica a 0H. Portanto, pode ser directamente comparado com o terminador 00H. O uso de mov (acesso em 16 bits) em vez de MOVEB teria os seguintes problemas:

- Se R1 fosse par, R3 ficaria com dois caracteres. Seria preciso isolar um carácter usando uma máscara AND com o valor 00FFH, para eliminar o carácter no byte de maior peso de R3;
- Se R1 fosse ímpar, a instrução MOV daria um erro (acessos em 16 bits têm de ser alinhados, isto é, o endereço têm de ser par). Isto significa que para aceder aos caracteres em endereços ímpares era preciso fazer um deslocamento à direita de 8 bits no valor de R3 lido com R1 par (e R1 tinha de evoluir de 2 em 2).
- próximo – O registo R1 contém o endereço do próximo carácter da cadeia, e evolui de 1 em 1 porque cada carácter ocupa apenas um byte. Se ocupasse dois bytes, teria de evoluir de 2 em 2.

### SIMULAÇÃO 5.12 – TABELA DE APONTADORES PARA DADOS

Esta simulação ilustra o funcionamento da tabela de apontadores para estruturas de dados, usando o Programa 5.12 como base. Os aspectos cobertos incluem os seguintes:

- Conteúdo das estruturas de dados em memória;
- Execução do programa passo a passo e com pontos de paragem;
- Evolução dos valores dos registos relevantes;
- Caso da mensagem vazia.

O Programa 5.13 exemplifica o uso de uma tabela de apontadores para rotinas.

|       |       |           |                              |
|-------|-------|-----------|------------------------------|
| pilha | EQU   | 2000H     | ; valor inicial do SP        |
| PLACE | 1000H |           |                              |
| base: | WORD  | Soma,     | ; zona de dados              |
|       | WORD  | Subtrai,  | ; endereço da rotina Soma    |
|       | WORD  | Mult,     | ; endereço da rotina Subtrai |
|       | WORD  | Divide,   | ; endereço da rotina Mult    |
| PLACE | 0000H |           |                              |
| fazl: | MOV   | SP, pilha | ; zona de instruções         |
|       | MOV   | R1, 7     | ; limite superior da pilha   |
|       | MOV   | R2, 4     | ; 2.º operando               |

```

MOV    R3, 0      ; código da soma
CALL   Trata      ; efectua soma
MOV    R1, 5      ; 1.º operando
MOV    R2, 8      ; 2.º operando
MOV    R3, 2      ; código da multiplicação
CALL   Trata      ; efectua multiplicação
Fin:  JMP   fin    ; fim do programa

```

```

Trata - Invooca a operação indicada pelo 3.º parâmetro
Entradas - R1 - 1.º operando
            R2 - 2.º operando
R3 - Operação (0 soma, 1 subtração, 2 multiplicação, 3 divisão)
Saídas - R1 - Resultado da operação
Destruí - R0, R3

```

```

Trata: MOV  R0, base ; base da tabela de apontadores
        ADD  R3, R3 ; duplica R3, pois a tabela é de apontadores e cada
                      ; elemento é um endereço que ocupa 2 bytes
MOV  R3, [R0+R3] ; obtém o endereço da rotina pretendida
CALL  R3 ; invoca-a. Os operandos já estão em R1 e R2
RET

```

```

Soma - R1 - Resultado da soma
Soma:  ADD  R1, R2 ; executa operação
        RET   ; termina, com resultado em R1

```

```

Subtrai - Subtrai o 2.º operando do 1.º operando
Entradas - R1 - 1.º operando
            R2 - 2.º operando
Saídas - R1 - Resultado da subtração
Subtrai: SUB  R1, R2 ; executa operação
        RET   ; termina, com resultado em R1

```

```

Mult - Multiplica os dois operandos
Entradas - R1 - 1.º operando
            R2 - 2.º operando
Saídas - R1 - Resultado da multiplicação
Mult:   MUL  R1, R2 ; executa operação
        RET   ; termina, com resultado em R1

```

```

Divide - Divide o 1.º operando do 2.º operando (divisão inteira)
Entradas - R1 - 1.º operando
            R2 - 2.º operando
Saídas - R1 - Resultado da divisão
Divide: DIV   R1, R2 ; executa operação
        RET   ; termina, com resultado em R1

```

Programa 5.13 - Exemplo de utilização de uma tabela de apontadores para rotinas

**SIMULAÇÃO 5.3 – TABELA DE APONTADORES PARA ROTINAS**

Esta simulação ilustra o funcionamento da tabela de apontadores para rotinas, usando o Programa 5.13 como base. Os aspectos cobertos incluem os seguintes:

- Conteúdo da tabela na memória;
- Execução do programa passo a passo e com pontos de paragem;
- Evolução dos valores dos registos relevantes;
- Efeito da alteração da tabela de endereços durante a execução do programa.

**5.8.5 ESTRUTURAS DE DADOS DINÂMICAS (MONTÃO)**

A pilha é uma estrutura de dados muito simples, pois baseia-se essencialmente num apontador (o SP), mas aplica-se apenas a casos de utilização específica, em que os valores armazenados na pilha sejam de utilização “ultimo a entrar, primeiro a sair”<sup>52</sup>. Felizmente, este é um caso muito frequente, abrangendo não apenas o mecanismo de chamada-retorno das rotinas como também os casos normais de guarda-restauro de valores de registos (incluindo os blocos de instruções imbricadas).

No entanto, a pilha tem uma desvantagem: os valores lá armazenados têm uma duração limitada ao contexto respectivo. Por exemplo, os valores das variáveis locais perdem-se quando uma função retorna. Para armazenar valores de forma mais persistente usam-se normalmente variáveis globais (definidas ao nível do programa principal), que só perdem o contexto quando o programa termina.

O problema é que o programador tem de especificar essas variáveis globais explicitamente e de forma estática, fixa durante toda a execução do programa. Isto é demasiado limitativo para alguns programas e por essa razão as linguagens de alto nível suportam a criação dinâmica (durante a execução do programa) de variáveis. Esta criação é feita de forma explícita, pelo programador, usando construções específicas para esse fim, como malloc em C e new em Java e C++, que retornam uma referência para cada variável criada dinamicamente (na prática, o endereço de memória em que a variável ficou localizada).

As variáveis criadas dessa forma duram até serem destruídas pelo utilizador<sup>53</sup>, potencialmente até ao fim do programa e de forma independente da pilha. Por isso têm de usar outra estrutura de dados, designada montão (*heap*).

De acordo com os dicionários correntes da Língua Portuguesa, um montão é um conjunto de coisas agrupadas de forma desordenada, sem preocupações de ordem (ao contrário da pilha, em que a ordem é crucial).

Tal como a pilha, o montão não é mais do que uma zona de memória. No entanto, enquanto na pilha a criação de uma variável é feita sempre no seu topo (basta decrementar o SP do número unidades correspondentes à dimensão da variável em palavras<sup>54</sup>), no montão as operações de criação de variáveis (malloc, new, etc.) têm de percorrer essa zona de memória à procura de um espaço disponível que tenha no mínimo a dimensão desejada.<sup>55</sup>

É o facto de as variáveis não serem criadas por uma ordem fixa que garante a durabilidade dessas variáveis enquanto não forem destruídas explicitamente, mas a criação/destruição sucessiva de variáveis provoca a fragmentação da memória destinada ao montão<sup>56</sup>, que obriga periodicamente à sua recompactação (operação que consiste em mudar as variáveis usadas de endereço de forma a juntá-las, o mesmo acontecendo aos espaços livres).

O acesso às variáveis criadas na pilha (normalmente parâmetros e variáveis locais) é feito tipicamente através de um registo (FP, ou *Frame Pointer*) que mantém uma referência para o contexto de cada rotina (secção 5.7.3.4) mas que só é válida enquanto essa rotina não retornar. Pelo contrário, as variáveis criadas no montão têm de sobreviver ao retorno das rotinas, mas não podem ser acedidas pelo nome, como as variáveis globais, uma vez que foram criadas dinamicamente, com o programa em execução.

As primitivas de criação de variáveis no montão, como malloc em C ou new em Java, devolvem um apontador que não é mais do que o endereço de base do espaço de memória onde cada variável foi criada e que pode depois ser usado para aceder a essa variável. Esse apontador é válido apenas até essa variável ser dealocada (explicitamente com a primitiva free em C ou de forma automática em Java – ver a nota de rodapé 53 na página 378).

O Programa 5.2, na página 297, já ilustrou o uso de apontadores, embora tenha usado um apontador para uma variável global, declarada com nome e portanto criada automaticamente pelo compilador e pelo sistema de suporte à execução do programa. O Programa 5.14 ilustra a utilização das funções malloc e free em C para melhor compreensão deve ser contrastado com o Programa 5.2.

<sup>52</sup> LIFO – Last In, First Out.

<sup>53</sup> Algumas linguagens, como Java, suportam reciclagem automática de memória, evitando trabalho ao programador e alguns erros típicos de gestão de memória. O sistema detecta automaticamente quando uma variável pode ser destruída analisando as referências para ela. Se já não houver nenhuma, já não é possível aceder a essa variável e pode destruir-se, libertando a memória correspondente que fica disponível para ser alocada a uma nova variável dinâmica.

<sup>54</sup> Nas linguagens de alto nível, uma variável pode ser estruturada e ocupar várias palavras de memória.

<sup>55</sup> Uma analogia que espelha esta diferença básica entre o montão e a pilha foi já referida na Tabela 5.15, na página 322.

<sup>56</sup> Quando se destrói uma variável entre outras que ainda perdurem cria-se um espaço livre de dimensão N palavras. Ao usar esse espaço para criar uma nova variável de dimensão P necessariamente menor ou igual a N, poderá ficar um espaço livre ainda menor (N-P).

```

/* declaração de constantes */
#define N 4 /* quantidade de números a ordenar */

main ()
{
    /* declaração de variáveis */
    /* programa principal */
    int * seq; /* apontador para a base vector que conterá os
    int * num; /* números a ordenar */
    int houveTroca; /* apontador para um dos elementos do vector */
    int i; /* indica se houve troca de números numa dada ronda */
    int auxiliar; /* posição (começa em 0) de um dado número no vector */

    /* instruções */
    seq = malloc (2*N); /* cria vector no montão com 2*N bytes, pois cada
    *seq = 10; /* elemento é um inteiro com 2 bytes. Retorna apontador */
    *(seq+1) = 5;
    *(seq+2) = 6;
    *(seq+3) = 2;

    /* inicio do ciclo de ronda (faz pelo menos uma) */
    houveTroca = false;
    /* inicializa apontador com o endereço do 1.º elemento */
    i = 0;
    while (i < N-1) /* testa os números todos até ao fim do vector */
    {
        if (*num > *(num+1)) /* se o número seguinte é maior... */
        {
            auxiliar = *num; /* ... troca-os, usando a variável auxiliar */
            *num = *(num+1);
            *(num+1) = auxiliar;
            houveTroca = true;
            /* agora já houve pelo menos uma troca */
        }
        i = i + 1; /* mais um número tratado */
        num = num + 1; /* passa a apontar para o número seguinte no vector */
    }
    houveTroca ? /* se houve trocas, tem de fazer mais rondas */
    free (seq); /* liberta espaço de memória no montão */
}

```

#### Programa 5.14 - Variante do Programa 5.2 com alocação dinâmica de variáveis no montão

Note-se que:

- O espaço de memória criado com malloc (que recebe como parâmetro o número de bytes a alocar) tem de ser inicializado após a sua criação;
- A função free liberta espaço de memória no montão, recebendo como parâmetro o apontador que o malloc devolveu. Esse espaço de memória passa a estar livre para novas invocações de malloc e o apontador deixa de ser válido. Qualquer acesso ao vetor após a invocação de free poderá ter resultados imprevisíveis, pois entretanto o espaço de memória em que o vetor se encontrava pode já ter sido usado por outra invocação a malloc;

A versão em linguagem assembly do Programa 5.14 não é mostrada porque as diferenças face à Tabela 5.6 na página 298 correspondem basicamente à invocação das funções malloc e free, que são complexas e extravasam o âmbito deste livro;

A criação dinâmica de variáveis é útil apenas em programas mais complexos, que exigam criação e libertação de espaço de memória com alguma frequência (por exemplo, quando se criam estruturas de dados que mudam de tamanho ao longo do tempo).

Uma outra diferença crucial entre a pilha e o montão é que a pilha é gerida em hardware. Dista inicializar o SP por um simples MOV, enquanto o montão é totalmente gerido em software, através da invocação das funções do sistema de suporte malloc e free.

Embora sejam estruturas de dados independentes, uma solução comum é reservar uma zona de memória para o conjunto pilha e montão, começando a usar-se a pilha de um dos dois e o montão do outro, o que permite uma gestão mais dinâmica das duas estruturas de dados. Um programa pode usar mais pilha se usar menos espaço no montão, e vice-versa. Se se abusar em qualquer delas, o sistema termina o programa com um erro do tipo "colisão montão-pilha", o que quer dizer que uma das estruturas cresceu tanto que chegou à zona de memória ocupada pela outra. Razões típicas para que tal aconteça:

Recursividade infinita (que gasta pilha sem parar);

Successivas alocações de memória (malloc) sem as libertações (free) correspondentes;

Dimensionamento insuficiente do espaço de memória necessário, quer da pilha quer do montão.

Fig. 5.11 ilustra esta organização típica.

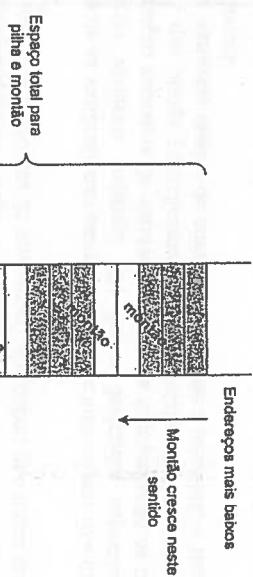


Fig. 5.11 - Organização típica da pilha e do montão

Note-se que, ao contrário da pilha, o montão pode ter zonas de memória livre no seu interior, tal como ilustrado pela figura. Isto deve-se a libertações de espaço de memória causadas pelas invocações a `free`. No entanto, estas zonas livres interiores são usadas preferencialmente por novas invocações a `malloc`. O montão só cresce se no seu interior não houver nenhuma zona livre de tamanho maior ou igual do que o pretendido por uma operação `malloc`. Devido a sucessivas alocações/libertações de memória, o espaço montão tende a ficar fragmentado, com pequenos espaços livres no seu interior que acabam por não ser úteis, devido à sua pequena dimensão (ver a nota de rodapé 56 na página 379). De vez em quando é preciso desfragmentar (compactar) o espaço do montão, mudando as zonas ocupadas de sítio, actualizando os apontadores no programa para essas zonas, e juntando todos os pequenos espaços livres num só, maior.

A descrição em detalhe da programação com variáveis dinâmicas e do funcionamento do montão está fora do âmbito deste livro, podendo consultar-se outros livros, como:

[Guerreiro 2001] e [Tannenbaum 2001].

### 5.8.6 LISTAS LIGADAS

As tabelas são estruturas de dados fixas, com um número imutável de elementos, todos de igual dimensão (que também não varia). Por conseguinte, não permitem uma gestão dinâmica dos dados. Por exemplo, inserir um elemento entre outros dois configura obriga a copiar todos os elementos subsequentes da tabela para uma posição à frente, de modo a criar espaço para o elemento novo, o que pode ser uma operação lenta. Isto pode ser visto na Tabela 4.18, na página 228, em que as fichas sobre cada empregado estão contiguas em memória, dispostas numa tabela.

Por motivos de dinamicidade de gestão, usam-se muitas vezes listas de elementos ligados em série por apontadores<sup>57</sup>, em vez das tabelas. O nome correto para esta estrutura de sequência designa-se cabeça da lista e o último cauda da lista. O apontador da cauda não apontará para nenhum outro nó (é o último) e por isso contém um valor especial (designado geralmente por null e com o valor típico 0000H) que indica essa situação. Em algum sitio (um registo ou uma célula de memória de endereço conhecido pelo programa) terá de haver um apontador para a cabeça da lista.

**NOTA** Para além de um apontador no sentido cabeça → cauda, cada nó pode ter também um apontador no sentido cauda → cabeça. A vantagem é permitir percorrer os nós da lista nos dois sentidos, o que pode ser útil em alguns algoritmos. Estas listas designam-se listas duplamente ligadas. É ainda possível colocar a cauda a apontar para a cabeça, caso em que a lista se designa lista circular.

A Tabela 5.31 compara as tabelas e as listas ligadas nas suas principais características.

| CARACTERÍSTICA                              | TABELAS                                                 | LISTAS LIGADAS                                                                          |
|---------------------------------------------|---------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Tempo de acesso                             | Rápido e constante, independente do elemento accedido   | Mais lento e variável (depende da posição na lista)                                     |
| Inserção e remoção de elementos intermédios | Mais difícil e potencialmente moroso (duração variável) | Fácil e duração constante após acesso ao elemento anterior ao ponto de inserção/remoção |
| Tamanho                                     | Fixo (na altura da criação)                             | Varável (ao longo do programa)                                                          |
| Eficiência de memória                       | 100% (só gasta espaço com os dados)                     | <100% (tem de gastar espaço de memória com os apontadores)                              |
| Tamanhos grandes                            | Obriga a um só espaço de memória contíguo               | Só os bytes de cada nó é que têm de estar contíguos                                     |

Tabela 5.31 - Comparação das principais características das tabelas e das listas ligadas

A Fig. 5.12a ilustra o mecanismo das listas ligadas, usando o mesmo exemplo da Tabela 4.18. A informação sobre cada empregado é colocada em tabelas individuais (fichas), mas em vez de colocar as fichas todas de forma contígua em memória (na prática, uma tabela bidimensional linearizada em memória – ver secção 5.8.4.2) ligam-se umas às outras por apontadores. Além dos campos de informação sobre cada empregado, cada ficha inclui um apontador que contém o endereço da próxima ficha. Cada nó da lista (a ficha) é assim uma tabela.

**Nota** Em termos formais, cada nó da lista devia ter apenas um apontador para o nó seguinte e outro para o elemento propriamente dito, que poderia ser uma simples célula de memória ou uma estrutura de dados complexa. No entanto, ao nível da linguagem assembly este rigor só introduz complexidade adicional, pelo que neste livro o nó inclui directamente o elemento junto com o apontador.

A Fig. 5.13 representa o cenário da Fig. 5.12 nos seus três estágios de evolução, mas agora em termos de células de memória, mostrando os valores dos apontadores (que poderiam ser quaisquer outros sem afectar o funcionamento da lista).

As operações típicas sobre uma lista são as seguintes:

- Aceder a um nó da lista, que não pode ser directamente como nas tabelas. Tem de se conectar pela cabeça e depois ir percorrendo a lista, de nó em nó, pois só a partir de um nó se pode conhecer o apontador para o nó seguinte. A travessia terá forçosamente de terminar na cauda, quando for encontrado um apontador com o valor null, e terá de ser feita quer seja motivada por uma iteração sobre os nós da lista quer simplesmente para chegar ao *n*-ésimo nó da lista. O Programa 5.15 exemplifica esta operação, com procura de uma ficha ao longo da lista;
- Inserir um novo nó na lista, como no caso da Fig. 5.12b, em que se inseriu um novo nó (Ficha4) entre a Ficha2 e a Ficha3. Tal é feito simplesmente colocando o apontador que estava na Ficha2 na Ficha4 e um novo apontador para Ficha4 na Ficha2. O Programa 5.15 exemplifica esta operação;

<sup>57</sup> Uma analogia algo sugestiva é uma fila de vários elefantes num circo, em que a tromba de um elefante agarra a cauda do elefante seguinte.

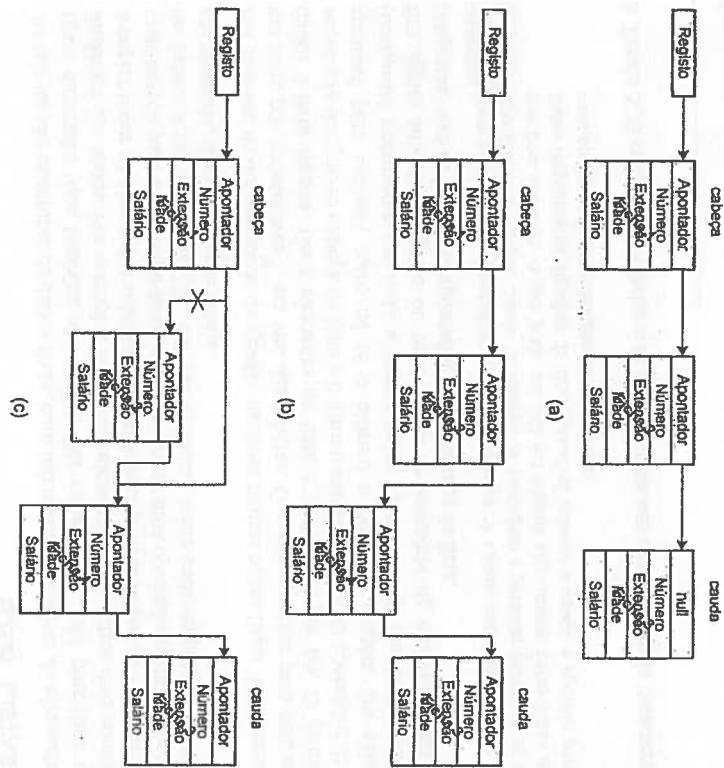


Fig. 5.12 - Exemplo de lista ligada. Tem de haver um registo ou célula de memória que aponte para a cabeça. (a) – Lista original; (b) – Após inserção de um novo elemento antes da cauda; (c) – Após remoção da Ficha2 da lista

- Acrescentar um novo nó no fim da lista (não ilustrado), que é um caso particular do anterior e bastante frequente. Neste caso, basta colocar o apontador para o novo nó na antiga cauda e nulo no apontador do novo nó, que passa a ser a nova cauda;
- Remover um nó da lista, como no caso da Fig. 5.12c, em que removeu o nó Ficha2. Tal é feito simplesmente colocando no apontador de Ficha1 o apontador que estava na Ficha2. Assim, Ficha1 passa a apontar para Ficha4. Note-se que não há necessidade de destruir o apontador para Ficha2 na Ficha2. O nó Ficha2 deixa de ser acessível, pois deixa de haver apontador para ele, e todo o seu espaço deve ser declarado livre. O Programa 5.15 exemplifica esta operação.

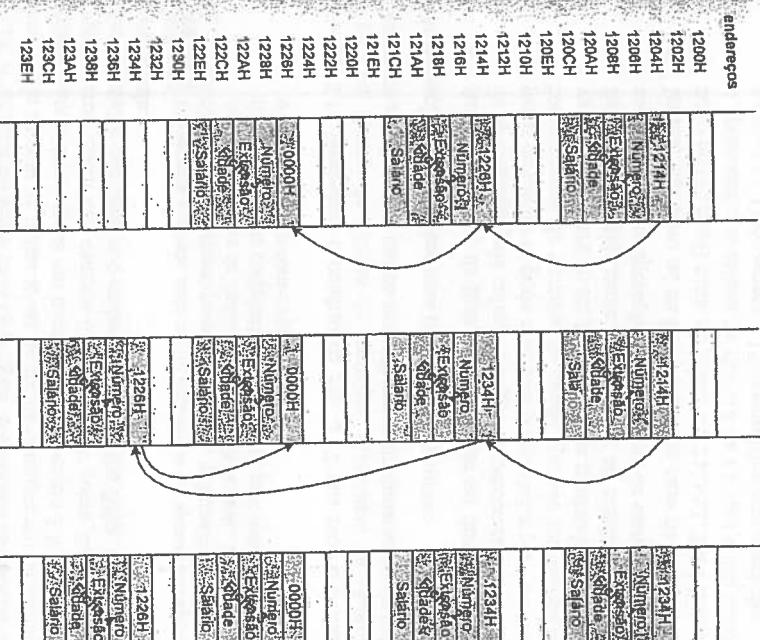


Fig. 5.13 - Uma possível implementação em memória do exemplo da Fig. 5.12, mostrando o valor dos apontadores

**NOTA** Tipicamente, o espaço de memória de cada nó é alocado no montão com as primitivas de gestão de memória dinâmica (secção 5.8.5). Retirar um nó da lista não liberta o seu espaço de memória, embora possa ficar inacessível (por não haver apontador para ele). Um programa que vá criando nós de listas e reutilizando outros desses listas sem os libertar tem fugas de memória. O problema é o aumento progressivo da memória gasta no montão, podendo chegar ao ponto de ocorrer um erro de colisão entre o montão e a pilha. Assim, qualquer nó que seja eliminado de uma lista e não seja necessário em mais lado nenhum deve ser libertado, quer explicitamente, com uma função de sistema apropriada (free em C, por exemplo), quer de forma automática, com reciclagem automática de memória – ver a nota de rodapé 53 na página 378 – quando para tal houver suporte (em Java, por exemplo).

## ESSENCIAL

- a O número de registos de um processador é muito limitado e não chega para conter as variáveis de um programa. Cabe ao compilador (ou programador) decidir que variáveis ficam em registos (normalmente as mais usadas) e quais têm de ser colocadas na memória, seja na pilha ou noutra zona qualquer.
- As tabelas são estruturas de dados muito usadas, mesmo em programas simples em linguagem assembly. São normalmente criadas com as directivas TABLE, WORD e STRING. A primeira reserva apenas o espaço, pelo que com o programa em execução este espaço tem de ser inicializado com instruções MOV.
- As tabelas são acedidas com instruções MOV, normalmente especificando (i) o endereço de base da tabela num registo e o índice como uma constante ou (ii) o registo, que se soma à base, ou (ii) apenas o endereço do elemento pretendido.
- Nun processador com endereçamento de bytes, como o PEPE, e numa tabela com elementos de 16 bits, o índice em elementos dentro da tabela é apenas metade do índice em bytes, pelo que tem de se multiplicar por 2 antes de somar à base ou usar numa instrução MOV.
- Uma tabela multidimensional (com vários índices) implementa-se por linearização, colocando sequencialmente em memória as tabelas mono-dimensionais constituintes, cada uma delas obtidas por fixação de todos os índices (excepto um) num dado valor, obtendo-se uma tabela global linearizada, de uma só dimensão. O acesso a esta tabela multidimensional, especificando vários índices tem de ser convertido num só índice, fazendo cálculos com os índices que permitem localizar o elemento pretendido na tabela linearizada.
- As tabelas de apontadores permitem referenciar um de vários elementos cuja dimensão pode inclusivamente não ser homogénea (basta os apontadores referirem todos do mesmo tamanho, uma palavra). Esses elementos podem ser dados ou mesmo instruções, o que permite, por exemplo, efectuar um CALL "dinâmico" (endereço da rotina a chamar pode variar conforme o valor de uma variável).
- Existe uma zona de memória (o montão) reservada para criar dados dinamicamente, memorizando depois apontadores para eles, de forma independente da pilha e do mecanismo de chamada e retorno das rotinas. É usual reservar um espaço total para pilha e montão, de modo a que a gestão do espaço ocupado por cada um deles possa ser ajustada ao comportamento do programa.
- As listas ligadas são sequências de dados dinâmicas, cuja dimensão pode variar ao longo do programa, podendo inserir-se ou remover elementos em qualquer ponto da lista. Os elementos não têm de estar contiguos em memória. O acesso é sequencial e mais lento do que nas tabelas, sendo necessário atravessar todos os elementos desde a cabeça da lista até ao elemento pretendido.

Nada impede que os elementos de uma tabela sejam apontadores para outras estruturas de dados (tabelas ou listas ligadas) ou que os nós de uma lista ligada sejam tabelas ou contenham também apontadores para outras estruturas de dados. No fundo, tudo são células de memória e respectivos endereços. Todas as estruturas de dados que inventarmos não passam de abstracções sobre a realidade mais básica, sem a alterar.

O Programa 5.15 exemplifica o acesso à lista (procurando a ficha de um empregado com um dado número mecanográfico para saber a sua idade) e a inserção e a remoção de uma ficha, seguindo a sequência de operações ilustrada pela Fig. 5.13. Os valores usados para o conteúdo das fichas são os indicados na Tabela 4.18, na página 228. Note-se que:

- As fichas estão preenchidas nos endereços indicados na Fig. 5.13 usando directivas PLACE e WORD. Trata-se de um mero truque para suprir a falta de primitivas de alocação de memória dinâmica (algo equivalente ao malloc em C) e permitir que o Programa 5.15 seja executado. Se essas primitivas existissem, cada ficha seria construída durante a execução do programa pedindo ao sistema para alocar 10 bytes de memória (para conter as cinco palavras de cada ficha) e inicializando essas palavras com instruções MOV.
- A rotina Remove recebe como parâmetro o apontador para a ficha anterior (na sequência da lista) à que se quer remover. Embora seja isto que dá jeito para a rotina, o programador poderá ter apenas o apontador para a ficha que quer remover. Para a rotina ser alterada para receber este parâmetro era preciso receber também como parâmetro um apontador para a cabeça da lista, para poder percorrê-la até achar uma ficha cujo apontador para a ficha seguinte seja igual ao apontador para a ficha a remover (ou seja, era preciso procurar a ficha anterior); Por simplicidade, estas rotinas não têm protecções contra apontadores null. Por exemplo, a rotina Remove não funciona bem se lhe for passado como parâmetro um apontador para a cauda da lista, pois irá remover a ficha seguinte (que não existe). Normalmente, as rotinas que lidam com apontadores fazem alguns testes para detectarem casos como este.

---

```

número EQU 2 ; índice em bytes (dentro da ficha) do campo número
extensão EQU 4 ; índice em bytes (dentro da ficha) do campo extensão
idade EQU 6 ; índice em bytes (dentro da ficha) do campo idade
salário EQU 8 ; índice em bytes (dentro da ficha) do campo salário
null EQU 0000H ; apontador null
pilha EQU 2000H ; valor inicial do SP

PLACE 1204H ; localiza ficha 1, cabeça da lista (Fig. 5.13).
ficha1: word 1214H ; apontador para a ficha 2
word 3029 ; n.º mecanográfico
word 207 ; extensão telefónica
word 34 ; idade
word 1250 ; salário
word 1214H ; localiza ficha 2 (Fig. 5.13)
ficha2: word 1226H ; apontador para a ficha 3

```

```

word 1978 ; n.º mecanográfico
word 225 ; extensão telefónica
word 23 ; idade
word 990 ; salário

PLACE 1226H ; localiza ficha 3 (Fig. 5.13)
ficha3: word null ; apontador null (cauda da lista)
word 2389 ; n.º mecanográfico
word 217 ; extensão telefónica
word 47 ; idade
word 1650 ; salário

PLACE 1234H ; localiza ficha 4 (Fig. 5.13)
ficha4: word null ; apontador null (para já, pois esta ficha ainda
; não está na lista).
word 1027 ; n.º mecanográfico
word 234 ; extensão telefónica
word 58 ; idade
word 2025 ; salário

PLACE 0000H ; localiza bloco de instruções
início: MOV SP, pilha ; inicializa SP
MOV R1, fichal ; apontador para a cabeça da lista
MOV R2, 2389 ; número mecanográfico do empregado 3
CALL Idade ; obtém idade desse empregado
MOV RL, fichal ; apontador para a ficha após a qual se vai inserir
MOV R2, ficha4 ; apontador para a ficha a inserir
CALL Insere ; insere esta ficha

MOV RL, fichal ; apontador para a ficha anterior à que se pretende
remover ; remove a ficha 2
CALL Remove ; o programa acabou
fim: JMP fim

;***** Idade - Obtém a idade de um empregado dado o seu número mecanográfico
; Entradas - R1 - Apontador para a cabeça da lista
; R2 - Número mecanográfico do empregado
; Saídas - R1 - Apontador para a ficha do empregado com esse n.º mac.
; R2 - Idade do empregado (0 se o empregado não existir)
;***** Idade:
PUSH R0 ; guarda registo
maiÚma: MOV R0, [R1+índice] ; guarda registo
        CMP R0, R2 ; obtém n.º mecanográfico
        JNZ próxima ; vê se é o número pretendido
        ; se não, passa à próxima ficha
        MOV R2, [R1+índice] ; obtém idade do empregado
        JMP acaba ; já está
        ; obtém apontador para a próxima ficha
        ; ainda há proxima ficha?
        ; se houver (não for null), continua à procura
        ; não achou nenhum empregado com este n.º mac.
        ; repõe registo
acaba: MOV R1, [R1] ; repõe registo
        CMP R1, null ; se null, continua
        JNZ maiÚma ; não achou nenhum empregado com este n.º mac.
        ; repõe registo
RET

```

**Sistema 5.14 - LISTAS LIGADAS**

Esta simulação ilustra o funcionamento das listas ligadas, tomando o Programa 5.15 como base. Os aspectos cobertos incluem os seguintes.

- Disposição das fichas em memória e seu conteúdo, incluindo apontadores;
- Funcionalidades básicas das listas (acesso, travessia, inserção e remoção);
- Evolução dos valores dos registos relevantes.

Programa 5.15 - Exemplo de utilização de uma lista ligada

```

Inserire - Insere uma nova ficha na lista
Entradas - R1 - Apontador para a ficha a inserir
Saidas - nenhuma
Inserire: PUSH R0 ; guarda registo
          MOV R0, [R1] ; obtém apontador para a próxima ficha
          MOV [R2], R0 ; coloca o apontador da ficha a inserir
          MOV [R1], R2 ; põe ficha a apontar para a nova ficha
          POP R0 ; repõe registo
RET

```

```

Remove: PUSH R0 ; guarda registo
          MOV R0, [R1] ; obtém apontador para a próxima ficha (a que se
                      ; vai Remover)
          MOV R0, [R0] ; obtém apontador para a ficha a seguir àquela
          MOV [R1], R0 ; que se vai remover
          ; põe ficha anterior a apontar para a ficha a
          ; seguir
          POP R0 ; repõe registo
RET

```

## 5.9 DESENVOLVIMENTO DE PROGRAMAS

### 5.9.1 CICLO DE DESENVOLVIMENTO

Até aqui, temos lidado com pequenos programas, muito simples e destinados apenas a ilustrar aspectos específicos. Os programas reais são normalmente bem mais complexos e têm um ciclo de desenvolvimento com mais passos do que uma simples tradução para linguagem máquina.

Isto deve-se essencialmente ao facto de que, com excepção de aplicações muito simples em computadores muito pequenos (como os que controlam um forno de microondas, por exemplo), é normal hoje em dia:

Um computador ter vários programas independentes carregados em memória e executá-los simultaneamente;

O programador não ter de programar rigorosamente tudo. Qualquer computador tem disponível um conjunto de pequenos programas que permitem efectuar operações básicas, como por exemplo lidar com os periféricos (seção 7.7.6, na página 685), e que o programador pode usar (assim, já tem parte do trabalho feito).

Estas duas características reflectem a evolução da tecnologia (nos primórdios, os programadores tinham de programar rigorosamente todos os detalhes e só um programa podia correr de cada vez) e permitiram conseguir que os computadores fizessem mais e melhores coisas, mas complicam todo o ambiente de programação. Felizmente, muitas das tarefas a executar estão já automatizadas. As consequências básicas são as seguintes:

- Se um computador pode ter vários programas carregados, não se pode exigir que cada um deles tenha de ser carregado numa zona de endereços específica, sendo rapidamente surgiram conflitos. O mesmo se pode dizer em relação aos endereços de dados. Poder-se-ia pensar em instruções de salto relativas ou recurso condado a directivas PLACE, mas a secção 7.6, na página 643, descreverá uma solução mais adequada;
- Só em casos muito simples um computador pode funcionar apenas com o programa do utilizador, sem software de suporte. Além disso, é de grande utilidade ter o computador controlado por um sistema operativo, que trate de gerir não apenas os vários programas do utilizador como também os recursos do computador;
- A existência de partes do programa já desenvolvidas (biblioteca de módulos, em que um módulo é um conjunto de rotinas, tipicamente organizadas num ficheiro formalmente já compilados, nem sequer as instruções em texto são fornecidas) exige um mecanismo de ligação entre os módulos fornecidos com o sistema os produzidos pelo utilizador.

A Fig. 1.4, na página 13, mostra o processo de conversão de uma ideia para um programa em linguagem de alto nível, daí para linguagem assembly e finalmente para código-máquina. No entanto, refere-se apenas à transformação da informação que descreve o que se pretende que o computador faça. Não indica os passos fundamentais que constituem todo o processo de desenvolvimento de um programa, incluindo testes e correção de erros. A Fig. 5.14 estabelece o cenário de forma mais completa.

Após o programador saber o que pretende do programa (especificação), faz um modelo adequado. Pode usar um simples fluxograma (seção 5.2) ou uma notação mais completa, como a UML (*Unified Modeling Language* [Silva 2005]). Esta notação é universalmente usada para descrever sistemas mas é demasiado complexa para ser descrita no âmbito deste livro introdutório, pelo que ao nível de programas simples, em particular em linguagem assembly, se usam apenas fluxogramas, mais simples.

Com base na especificação e no modelo, o programador produz um algoritmo implementado numa linguagem de programação, como por exemplo C ou linguagem assembly. A

Fig. 5.14 representa as duas alternativas, mas tipicamente só se usa uma ou outra. Em alguns casos especiais, uns módulos do programa são feitos em C, para programar em alto nível, e outros módulos são feitos em linguagem assembly, para permitir uma implementação mais eficiente (tipicamente no acesso aos periféricos, em módulos designados gestores de periféricos, ou *device drivers* – secção 7.7.6, na página 685).

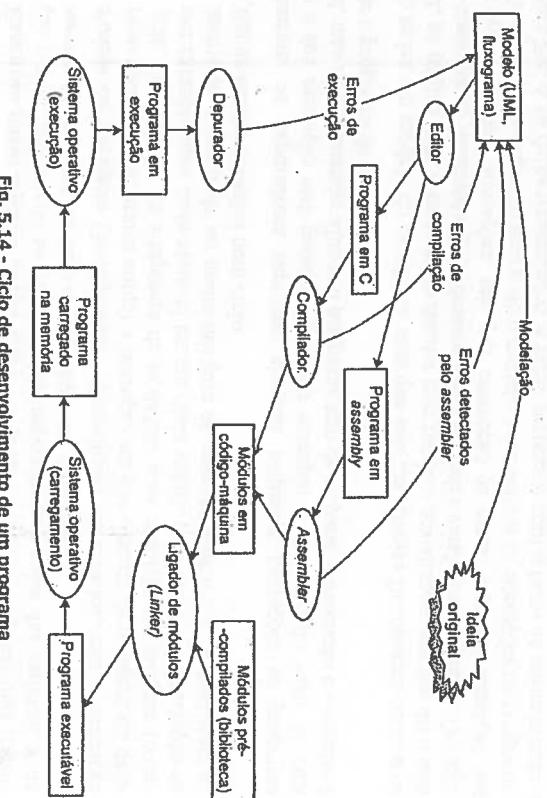


Fig. 5.14 - Ciclo de desenvolvimento de um programa

O programa é feito num editor de texto, que pode ser independente do resto das ferramentas ou integrado (IDE – *Integrated Development Environment*, ou Integrado), caso em que o editor reconhece as construções da linguagem e oferece algumas vantagens, como cores diferentes para as palavras reservadas e para os comentários.

De seguida, o compilador ou o assembler (consoante a linguagem em que o programa foi feito) analisa o programa, reconhece as instruções e gera código-máquina. Normalmente, os compiladores geram código-máquina directamente, sem usar linguagem assembly como passo intermédio. Esta destina-se apenas a constituir uma forma de programar um computador ao mais baixo nível, de forma a gerir directamente os seus recursos, geralmente por motivos de eficiência (rapidez de processamento). As funcionalidades menos críticas são sempre programadas em alto nível.

Programas grandes podem ser divididos em vários módulos, cada um em seu ficheiro, que são compilados separadamente. Cada módulo compilado tem código, mas não é um programa completo. Quer o compilador quer o assembler são programas que leem o ficheiro com instruções na linguagem respectiva e produzem outro ficheiro com as instruções do código-máquina que lhes correspondem.

Normalmente, já há módulos pré-compilados fornecidos pelo fabricante do computador ou por uma terceira empresa, que incluem funções de grande utilização (por exemplo, funções de baixo nível que lidam com os periféricos – secção 7.7.6, na página 685).

Os vários módulos compilados pelo programador são de seguida ligados entre si e com os módulos pré-compilados fornecidos com o sistema, de modo a formar um programa completo. Só nesta fase de ligação dos vários módulos é que são determinados os endereços em que as diversas partes do código-máquina e as zonas de dados ficam localizados. Quando um módulo é compilado, é produzida uma tabela de símbolos, semelhante à Tabela 5.5 mas em que os endereços não estão preenchidos. Só quando os diversos módulos se ligam é que essas tabelas podem ser completadas. Nessa altura preenchem-se no código-máquina as referências a endereços absolutos para as quais se tinha reservado espaço com um valor provisório. Obviamente, só pode haver saltos relativos entre instruções dentro do mesmo módulo.

Um programa executável é um ficheiro que contém um conjunto de módulos já ligados (nos casos mais simples só há um módulo), a que se acrescenta automaticamente alguma informação, como por exemplo qual o endereço em que o programa deve começar a ser executado.

Um ficheiro não passa de um conjunto de bytes em disco. Para executar um programa executável, o sistema operativo tem de carregar o programa na memória principal e executá-lo, usando a informação contida nesse ficheiro sobre o endereço de arranque de execução.

Durante o desenvolvimento de um programa, é normal que surjam erros. Tirando casos triviais, é virtualmente impossível desenvolver um programa totalmente correcto logo à primeira tentativa.<sup>53</sup> Esses erros podem ser de dois grandes tipos:

- **Compilação** – São os mais fáceis de corrigir. São detectados pelo compilador (ou *assembler*) e podem ser de dois tipos:
  - **Sintaxe** – O programa não cumpre as regras gramaticais da linguagem. A falta de um sinal de separação de instruções ou de um identificador, identificador inválido, etc., são erros típicos;
  - **Semântica** – Geralmente devido ao uso incorrecto de um identificador, como por exemplo utilização de uma variável do tipo A quando se estava à espera de uma variável do tipo B.
- **Execução** – Se o programador tiver sorte, o erro é detectado pelo sistema e reportado, indicando a instrução em que correu (por exemplo, uma divisão por zero). Caso contrário, normalmente em casos de erro no algoritmo (por exemplo, o programador esqueceu-se de incrementar uma variável), o programa pura e simples-

<sup>53</sup> Piada clássica: Toda a gente sabe que um programa que não temia pelo menos um ciclo e um erro só pode ser tão trivial que não chega a ser um programa...

Boas regras de programação e teste individual de pequenas partes do programa constituem meio caminho andado para não haver erros de execução, mas é normal ter de recorrer à depuração. A sua vertente mais básica, se não houver outro recurso, é sempre possível ir colocando instruções em pontos estratégicos do programa para fazer sair informação sobre o que se vai passando no programa (em C, é típico usar *printf*s). No entanto, o melhor é ter um ambiente de programação integrado (IDE – *Integrated Development Environment*) que inclua um depurador que suporte:

- **Execução passo a passo (single-step)** – O programa executa apenas uma instrução e para, permitindo ao programador inspecionar o valor das variáveis e até mesmo dos registos (no caso de depuração em *assembly*);
- **Pontos de paragem (breakpoints)** – Semelhante ao anterior, mas a execução prossegue até o programa atingir a instrução em que o ponto de paragem foi definido. Geralmente, há a hipótese de se definir uma condição de paragem (como por exemplo uma dada variável ter um dado valor), para além da localização da instrução, o que pode ser muito útil para os casos em que o erro se manifeste ao fim de muitas iterações num ciclo.

Normalmente, os depuradores suportam algumas pequenas alterações ao programa quando a sua execução está parada, como por exemplo a alteração do valor de uma variável, mas se se pretender alterar o programa tem de se parar a execução e compilar e executar o programa de novo.

Quando se faz um modelo do problema, nem que seja um simples fluxograma como o da Fig. 5.3, na página 279, é importante manter esse modelo actualizado, em vez de o usar apenas como passo intermédio para desenvolver a primeira versão do programa. Ou seja, sempre que se fazem alterações, seja por correção de erros, seja por alteração das especificações, deve primeiro alterar-se o modelo e daí derivar as alterações no programa, tal como a Fig. 5.14 deixa transparecer. A razão prende-se com o facto de normalmente o modelo ser mais fácil de compreender, pois não tem todos os detalhes, pelo que é mais fácil de alterar do que directamente no programa. Desta forma, continua a actuar como passo intermédio e guia auxiliar na tarefa de programar. Este aspecto é tão mais importante quanto mais complexo o programa for.

## 5.9.2 PROGRAMAÇÃO EM ALTO NÍVEL OU EM LINGUAGEM ASSEMBLY?

Uma das primeiras decisões ao programar um sistema é escolher que linguagem de programação usar. Tipicamente, a melhor escolha recai sobre uma linguagem de alto nível, que permite abstrair muitos dos detalhes do computador e aumentar a produtividade dos programadores. No entanto, é preciso ter os seguintes factores em conta:

- As linguagens de programação mais recentes e evoluídas (Java, C#) não estão disponíveis em todas as plataformas, em particular as baseadas em microprocessadores de gama mais baixa, em que a diversidade do mercado é maior. Se a aplicação a desenvolver se destinar a ser portada para vários sistemas diferentes, este aspecto é essencial. A linguagem C continua a ser de longe a linguagem de programação mais divulgada, existindo virtualmente em todas as plataformas, mas da menos suporte ao programador;
- A linguagem assembly é necessária apenas em situações especiais, em que a inteligência do programador é necessária para construir uma rotina que precise de estar muito optimizada, ou que seja de tão baixo nível que manipule recursos físicos, de tal forma que não consiga ser expressa na linguagem de alto nível.

Neste livro, usa-se essencialmente a linguagem assembly, para que se possa ilustrar os vários aspectos do funcionamento de um computador e não porque os computadores devam ser programados na sua própria linguagem específica. Os compiladores das linguagens de alto nível conseguem ser bastante eficientes e são muito mais fáceis de programar, pelo que devem ser usados sempre que possível.

## 5.9.3 DESENVOLVIMENTO EM LINGUAGEM ASSEMBLY

As linguagens de alto nível têm instruções que quase se assemelham à nossa escrita natural, tornando clara boa parte da semântica do programa. Em linguagem assembly, as instruções são de muito baixo nível. O número de instruções a efectuar é assim muito mais elevado do que numa linguagem de alto nível, pelo que é muito mais fácil o programador "perder-se" nos meandros do algoritmo. Desse modo, as regras mais básicas do bom programador em linguagem assembly são as seguintes:

- \* Documentação abundante:
- Praticamente todas as instruções devem ter um comentário, que não deve fazer a simples transcrição do que cada instrução faz mas sim o que ela significa. A Tabela 5.32 ilustra este aspecto crucial;
- Cada rotina deve ter um cabeçalho que indique o que a rotina faz, no seu conjunto, informação que entra, informação que sai e informação que a rotina altera (registos, variáveis em memória, etc.). Também poderá ser útil incluir nome do autor, data em que foi alterada, número da versão, histórico das alterações, etc. (em particular em programas feitos por vários programadores)
- Com umas linhas de caracteres repetidos, o cabeçalho serve também para melhor identificar e separar as rotinas, tornando mais fácil a sua localização e,

compreensão. O formato do cabeçalho varia consoante o programador, mas a Tabela 5.33 ilustra com um exemplo: a rotina factorial da Tabela 5.27, mas desta vez com um cabeçalho em vez de uns meros comentários. Será útil comparar esta versão com a da Tabela 5.27, na página 358. Note-se que também em C se usam cabeçalhos e comentários, embora a um nível de abstracção mais elevado.

| TIPO DE COMENTÁRIOS                                                             | CÓDIGO EM ASSEMBLY                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Sem comentários                                                                 | PLACE Saldo WORD 0<br>PLACE 0000H<br>Deposita:<br>MOV R3, 100<br>MOV R1, Saldo<br>MOV R2, [R1]<br>ADD R2, R3<br>MOV [R1], R2<br>...                                                                                                                                                                                                                                                |
| (o que é que isto faz?)                                                         | PLACE 1000H ; Localiza endereços em 1000H<br>Saldo WORD 0 ; Variável Saldo inicializada a 0<br>PLACE 0000H ; localiza endereços em 0000H<br>Deposita:<br>MOV R3, 100 ; Coloca 100 em R3<br>MOV R1, Saldo ; Coloca o endereço de Saldo em R1<br>MOV R2, [R1] ; Lê memoria endereçada por R1<br>ADD R2, R3 ; Soma R2 e R3<br>MOV [R1], R2 ; Escreve memoria endereçada por R1<br>... |
| Comentários inúteis (não fazem mais do que repetir a instrução assembly)        | PLACE 1000H ; Início da zona de dados<br>Saldo WORD 0 ; Variável Saldo da conta bancária<br>PLACE 0000H ; Início do programa<br>Deposita:<br>MOV R3, 100 ; Valor a depositar na conta<br>MOV R1, Saldo ; Endereço da variável Saldo<br>MOV R2, [R1] ; Lê o valor actual do saldo<br>ADD R2, R3 ; Soma-lhe o valor depositado<br>MOV [R1], R2 ; Actualiza a variável Saldo<br>...   |
| Comentários úteis (explicam o papel de cada instrução no contexto da aplicação) | PLACE 1000H ; Início da zona de dados<br>Saldo WORD 0 ; Variável Saldo da conta bancária<br>PLACE 0000H ; Início do programa<br>Deposita:<br>MOV R3, 100 ; Valor a depositar na conta<br>MOV R1, Saldo ; Endereço da variável Saldo<br>MOV R2, [R1] ; Lê o valor actual do saldo<br>ADD R2, R3 ; Soma-lhe o valor depositado<br>MOV [R1], R2 ; Actualiza a variável Saldo<br>...   |

Tabela 5.32 - Exemplo simples que ilustra o efeito de ausência de comentários.

- Não usar truques para ficar mais rápido ou ocupar menos memória. Os processadores estão cada vez mais rápidos e a memória cada vez mais barata e maior. O pouco que se pode ganhar em eficiência quase nunca compensa o tempo (quase sempre demasiado) que se demora a corrigir um erro introduzido ao fazer uma alteração porque o programador se esqueceu de todas as restrições necessárias para o truque funcionar. Deve sempre evitar-se usar truques optimizantes mas obscuros e rebuscados, e preferir código mais simples e estruturado, possivelmente menos eficiente mas mais claro e fácil de alterar.

```

FUNÇÃO EM C
***** ROTINA EM ASSEMBLY *****
Factorial - Calcula o factorial de um numero N
Autor: José Delgado
Versão 1.0
Última modificação: 11/04/2006
Parâmetros: n - Número N
Retorna - Valor do factorial
*****
int factorial (int n) {
    int produto;
    int i;
    produto = 1; /* inicializa factorial*/
    for (i=n; i>1; i--) {
        /* acumula o produto dos vários
         factores, de n a 2 */
        produto = produto * i;
    }
    return produto; /* retorna factorial*/
}

```

```

; ***** ROTINA EM ASSEMBLY *****
; Factorial - Calcula o factorial de um numero N
; Autor: José Delgado
; Versão 1.0
; Última modificação: 11/04/2006
; Entradas: R1 - Valor do numero N
; Saidas: R2 - Valor do factorial
; Destròis: R3 (variável i de iteração)
factorial:
    MOV R2, 1      ; produto = 1;
    MOV R3, R1      ; i=n;
    ciclo: CMP R3, 1   ; i > 1?
                JLE fim; ; se não, acaba ciclo
                SUB R3, 1   ; i--
                MULT R2, R3 ; produto*produto*i;
                JMP ciclo ; continua ciclo
fim: RET          ; retorna

```

Tabela 5.33 - Exemplo de utilização de um cabeçalho

O Programa 6.1, na página 474, e o Programa B.1, na página 724, constituem exemplos de rotinas e programas completos em linguagem assembly, onde pode ser melhor apreciado o estilo correto de programação a este nível. Naturalmente, cada programador tem o seu estilo de indentação, comentários, cabeçalhos, convenções para os nomes dos identificadores e até mesmo em termos de utilização de letras maiúsculas ou minúsculas. O assembler do PEPE é sensível a este último aspecto em termos de identificadores, mas em termos de mnemónicas das instruções e dos identificadores reservados (nomes de registos, nomeadamente) tanto aceita letras maiúsculas como minúsculas. Há programadores que preferem usar letras maiúsculas para as mnemónicas e identificadores reservados porque se distinguem melhor dos restantes identificadores e dos comentários, e essa é a razão do uso desta convenção na maior parte dos exemplos deste livro. No entanto, dá trabalho estar constantemente a alternar entre letras maiúsculas e minúsculas, pelo que muitos programadores preferem usar apenas minúsculas.

## 5.9.4 AMBIENTES DE DESENVOLVIMENTO

### 5.9.4.1 COMPUTADOR ALVO E HOSPEDERIO

É importante perceber que para desenvolver um programa (editar, compilar, executar, etc.) é preciso ter já um computador a executar um sistema operativo, que receba comandos do utilizador e permita executar o editor, compilador, etc. que em si também

são programas. Apenas com o hardware de um computador, sem programas de desenvolvimento, não se consegue fazer nada.

Assim, há que fazer distinção entre dois tipos de computadores (do ponto de vista do desenvolvimento de programas):

- Computador hospedeiro (host)** – Necessariamente tem que ter interface com utilizador (ecrã, rato, teclado), sistema operativo, disco, programas de desenvolvimento (editor, compilador, depurador, etc.). Em suma, um computador usável. Actualmente, o computador de desenvolvimento mais divulgado é o PC.

- Computador alvo (target)** – É o computador que irá executar o programa que está a ser desenvolvido. O código-máquina gerado tem de ser o código-máquina específico<sup>59</sup> desse computador, que nos casos mais simples pode não ter qualquer software de apoio (em que o programa a desenvolver será o único a executar).

- Os objectivos do computador hospedeiro são fundamentalmente os seguintes:
  - Producir um programa executável para o computador alvo, por meio das ferramentas (programas) de desenvolvimento;
  - Servir de interface de depuração (reporte de erros, mensagens de depuração, etc.) ao longo do desenvolvimento do programa.

Há assim dois tipos de ambiente de desenvolvimento, tal como é indicado na Fig. 5.15:

O computador alvo é também o computador hospedeiro, isto é, o programa desenvolvido é executado no próprio computador em que é desenvolvido. Neste caso, quer o compilador quer o assembler geram código-máquina para o processador desse computador, e o programa desenvolvido é mais um entre todos os que o sistema operativo desse computador executa, em pé de igualdade com todas as ferramentas do ambiente de desenvolvimento (editor, compilador, depurador, etc.). Os ficheiros são guardados no disco desse computador e o programa executável é carregado do disco para a memória principal desse mesmo computador para ser executado;

O computador alvo é diferente do computador hospedeiro (ambiente de desenvolvimento cruzado – cross development environment). Nesse caso, o programa executável é desenvolvido no computador hospedeiro (gerando código-máquina do computador alvo), mas depois tem de ser enviado para o computador alvo para aí ser executado. Isto implica uma ligação física entre os dois, que também pode ser usada para o computador alvo enviar mensagens de erro ou de teste (auxiliares de depuração) para o ecrã do computador hospedeiro. Esta ligação pode ser uma simples porta série, USB ou ethernet (rede local, ou LAN). A Fig. 5.16 ilustra um computador alvo típico deste ambiente.

<sup>59</sup>Normalmente designado "nativo".

sem necessidade de software do lado deste último nem sequer de retirar o microprocessador do circuito.

Há basicamente três formas de encarar o computador alvo:

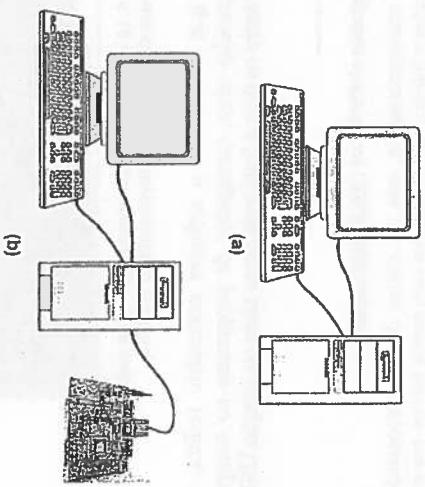


Fig. 5.15 - Ambientes da desenvolvimento de um programa: (a) - Para o próprio computador; (b) - Para outro computador (desenvolvimento cruzado - cross development)

Normalmente, o primeiro tipo de ambiente usa-se para programas com alguma complexidade e que pela sua natureza devem ser executadas num computador com sistema operativo, disco, interface de utilizador (normalmente gráfica, com menus e janelas), etc. Podem ser programas de contabilidade, planeamento, gestão, bases de dados, etc. Hoje em dia os computadores usam-se para tudo e há milhares de programas tão complexos quanto úteis.

O ambiente de desenvolvimento cruzado usa-se tipicamente para desenvolver programas para computadores pequenos, normalmente sem disco, com uma interface de utilizador muito rudimentar (pelo menos quando comparada com a de um PC) e em que muitas vezes não há sequer sistema operativo e o programa desenvolvido é o único a ser executado no computador alvo (é por estas razões que é preciso haver um computador hospedeiro).

No computador hospedeiro, é o disco (ou qualquer outro sistema de memória de massa) que garante que a informação não se perde quando o computador é desligado. A memória principal é volátil e não pode ser usada para este fim. Quando o computador é ligado, começa por ler do disco e executar um pequeno programa (o *boot loader*), que é depois responsável por ler o sistema operativo (também ele um programa), que de seguida torna conta do computador é responsável por carregar em memória principal e executar os programas, de acordo com os comandos que o utilizador for dando.

Geralmente, nos pequenos computadores alvo não há disco, pelo que o programa tem de ser guardado em circuitos integrados de memória não volátil. Hoje em dia usa-se tipicamente ROM Flash (seção 6.5.3.1, na página 534), uma tecnologia que permite transferir os programas por hardware do computador hospedeiro para o computador alvo.

**Real** - É o computador alvo propriamente dito. Normalmente, consiste numa placa com base num microprocessador comercial, formando um computador completo, incluindo os periféricos necessários (sensores, interruptores, LEDs, LCDs, interfaces para actuadores externos, como lampadas, motores, etc.). Esta placa é feita à medida de cada aplicação específica;

**Simulado** - O programa destinado ao computador alvo é executado no hospedeiro, tipicamente por um interpretador, que é um programa que interpreta e simula em software a funcionalidade de cada instrução que o computador alvo executaria em hardware. É uma simulação do sistema real, em que se usa a interface gráfica do computador hospedeiro para simular os diversos periféricos do computador alvo real (por exemplo, um LED representa-se como uma pequena bolha no ecrã cuja cor varia consoante o estado do LED). Todas as simulações neste livro são exemplo desta técnica, cujas principais características, que constituem as suas vantagens e limitações, são:

- Experimentar antes de construir o sistema real, o que permite logo obter alguns resultados que podem evitar erros que obrigariam a refazer o sistema real;
- Desenvolver e testar programas para o computador alvo em paralelo com o desenvolvimento do hardware do sistema real, o que permite ganhar tempo;
- Determinismo e controlo, isto é, a simulação comportar-se sempre da mesma forma, ao contrário do sistema real em que há factores que podem não ser reproduutíveis ou controláveis;
- O tempo de simulação desenvolve-se de forma mais lenta do que no sistema real, dependendo essencialmente das capacidades de processamento do computador hospedeiro, que é que corre o programa destinado ao computador alvo. Isto é feito tipicamente através de um interpretador de código-máquina, que simula funcionalmente o hardware real mas de forma muito mais lenta. Isto poderá impedir o teste de situações mais reais e não revelar logo à partida limitações do sistema real. Ou seja, uma simulação é apenas uma estimativa do que se irá efectivamente passar com o sistema real.
- **Emulado** - Consiste num computador alvo real mas em que em lugar do microprocessador se usa uma ponta de prova que o substitui de forma praticamente real. Esta ponta de prova faz parte de um ICE (*In-Circuit Emulator*) que inclui

<sup>61</sup> Dependendo da implementação do interpretador, do código-máquina interpretado, da forma como este codifica as instruções e de vários outros factores, a diferença entre os tempos de execução real e interpretada poderá variar entre dezenas e dezenas de milhares de vezes.

um microprocessador real e *hardware* especial que permite ao hospedeiro um controlo completo e em tempo-real sobre o computador alvo. Trata-se de reunir o melhor dos dois mundos (controlo total, como no computador alvo simulado, mas com execução em tempo real, como no computador alvo real). Naturalmente, estes sistemas são bastante dispendiosos, até porque cada microprocessador tem de ter o seu ICE específico.

A Fig. 5.16 apresenta um exemplo de um computador alvo real usado num sistema de desenvolvimento cruzado. Trata-se de uma placa da ATMEL<sup>61</sup>, com 16x11 cm, que inclui um microprocessador também da ATMEL, baseado no ARM<sup>62</sup>, ROM Flash, RAM e alguns LEDs e interruptores para interacção com o utilizador, bem como alguns bits de entrada/saída para ligar a outros dispositivos. Aproximadamente ao meio, com uma etiqueta em cima, pode observar-se o microcontrolador. Do lado esquerdo, vê-se uma bateria de lítio, redonda, para servir de alimentação ao relógio de tempo real mesmo quando a placa está desligada. Na parte de trás, a ficha mais pequena termina o cabo de alimentação da placa, enquanto a ficha maior é o terminador do cabo que liga ao computador hospedeiro.

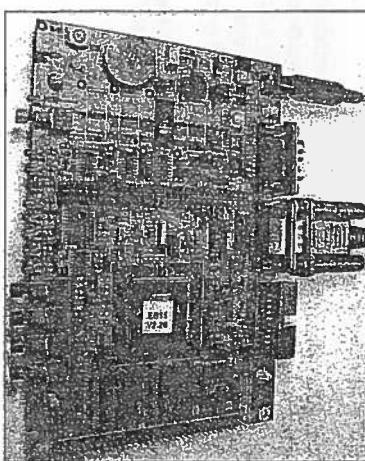


Fig. 5.16 - Exemplo de um computador alvo usado num sistema de desenvolvimento cruzado

Este último é um PC e que inclui todo o software necessário (editor, compilador, comunicação com o computador alvo, carregador de programas no computador alvo, depurador, etc.). A ligação entre os dois é feita pela porta série ou ligação USB do PC.

<sup>61</sup> Fabricante de circuitos integrados electrónicos ([www.atmel.com](http://www.atmel.com)).

<sup>62</sup> O ARM, em si não é um microprocessador, mas um núcleo de microprocessador de 32 bits disponível para outros fabricantes o integrarem, juntamente com outros blocos, em microprocessadores comerciais. É um conceito com base europeia, que tem conhecido um grande sucesso a nível mundial ([www.arm.com](http://www.arm.com)).

O utilizador desenvolve os programas para a placa, usando o PC como hospedeiro para o sistema de desenvolvimento e como interface de utilizador para depuração.<sup>63</sup> Quando o programa estiver pronto, a placa poderá funcionar autonomamente. A ROM Flash mantém o programa de forma não volátil (pode desligar-se a alimentação que o programa não se perde) e a RAM permite armazenar variáveis que o programa utilize. Passa a ser um computador completo e autónomo, embora com funcionalidade específica.

#### 5.9.4.2 SISTEMAS EMBEDEDOS

Os computadores alvo são cada vez mais baratos e pequenos, verificando-se que na maior parte das vezes nem os vemos. No entanto, eles estão por toda a parte, desde um simples controlador da intensidade luminosa de um candeeiro numa sala, passando pelo controlador de microondas ou de uma torradeira, computador de bordo de um automóvel, monitor de sinais vitais em equipamento médico, etc., até a computadores mais elaborados que controlam aviões e naves espaciais. Portanto, a nossa vida depende deles.

Não vemos estes computadores porque eles estão incluídos dentro de outros aparelhos que não se parecem com o conceito que tipicamente temos de um computador (um PC, por exemplo). Uma máquina de lavar roupa não se parece com um computador mas o mais provável é ter um computador lá dentro que detecta quando o utilizador carrega nos botões e gera a sequência de ações (mete água, centrifuga, etc.) de um dado programa (esf., de lavagem) seleccionado pelo utilizador. Estes pequenos computadores, tipicamente usados em aplicações de controlo, designam-se sistemas ebebidos (por estarem "embebidos" nouros sistemas). Cerca de 99% de todos os processadores vendidos actualmente destinam-se ao mercado dos sistemas ebebidos.

**NOTA** Note-se no entanto que em alguns sistemas ebebidos os "pequenos" computadores são PCs. Por exemplo, as máquinas Multibanco não passam de um PC com periféricos adequados (leitor de cartões, impressora de talões, dispensário de notas, etc.). A vantagem de usar um PC é facilitar o ambiente de programação e a integração dos diversos equipamentos. Portanto, a noção de "pequeno computador" é relativa. O factor identificativo de um sistema ebebido é o facto de o computador se destinar a executar uma aplicação específica em vez de ser usado como computador programável de utilização geral.

Por outro lado, com o avanço da tecnologia acaba por suceder o mesmo que aconteceu na evolução do reino animal: há dispositivos que são híbridos, o que complica a sua classificação. Estão neste caso os computadores integrados nos telefones móveis mais recentes, que por um lado são sistemas ebebidos (tem funcionalidade específica, incluída nos menus, e fazem parte de uma caixa pequena mas que inclui ainda todo o sistema de comunicações sem fios) mas por outro são já programáveis com código Java descarregado a partir de um servidor.

<sup>63</sup> A placa dispõe de alguns periféricos, nomeadamente LEDs, que também podem ser usados para depuração (fazendo-os acender ou apagar quando o programa passa por determinadas instruções e/ou em determinadas circunstâncias).

Geralmente, os computadores alvo que são programados por um sistema de desenvolvimento cruzado destinam-se a aplicações de sistemas embutidos, que dispõem de menos recursos do que um computador pessoal típico (por essa razão é que o sistema de desenvolvimento é cruzado). Mesmo assim, há muitas classes de sistemas embutidos desde sistemas críticos em termos de fiabilidade, passando por sistemas que requerem já elevadas capacidades de cálculo mas baixo consumo, até aos sistemas produzidos em grandes quantidades e em que a funcionalidade pretendida é muito pequena e o factor mais importante é o custo.

Assim, as características do sistema pretendido são fundamentais na determinação do tipo de processador a usar, desde as arquiteturas mais modernas de 32 bits até às arquiteturas já antigas de 8 bits (cujo investimento no seu desenvolvimento já foi largamente amortizado e requerem apenas tecnologia barata para serem fabricadas). Por outro lado, o nível de integração conseguido tem grande impacte no custo, no consumo e nas dimensões físicas, todos factores importantes, em particular nos sistemas embutidos móveis. A placa da Fig. 5.16, por exemplo, já tem algumas capacidades importantes em termos de processamento e memória<sup>64</sup>, o que obriga a um conjunto importante de circuitos integrados externos ao processador e impõe dimensões mínimas e custo apreciável. Há no entanto aplicações em que o muito baixo custo é o aspecto mais fundamental (controlador de um forno de microondas, por exemplo).

Um microprocessador clássico inclui apenas o processador em si, mas não a RAM, nem ROM, nem periféricos, que têm de ser externos. Como estes elementos têm de existir em qualquer computador, desde os primeiros tempos da evolução dos microprocessadores que se começou a integrar periféricos no mesmo circuito integrado que o microprocessador. Com a evolução da tecnologia foi ainda possível integrar RAM e ROM, nascendo assim o microcontrolador, que é um pequeno computador num só circuito integrado, constituído tipicamente por um processador, ROM Flash, EEPROM, RAM e periféricos.

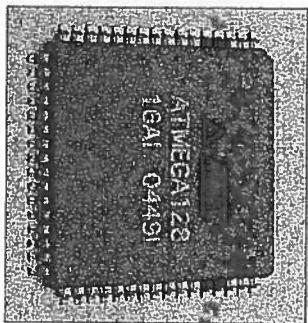


Fig. 5.17 - Exemplo de um microcontrolador, um computador completo num só circuito integrado

O microcontrolador é capaz de suportar directamente pequenas cargas como LEDs. Para construir muitos sistemas simples basta apenas ligar interruptores (para o utilizador actuar) ou outros sensores e LEDs (para visualização de informação). Para actuação de cargas com maior potência (motores, lampadas, etc.), é preciso juntar circuitos electrónicos de potência, mas todo o controlo está incluído no microcontrolador, tornando os sistemas mais simples do que usando microprocessadores, memórias e periféricos não integrados.

<sup>64</sup> Processador ARM de 32 bits a 33 MHz, 256 KBytes de RAM e 4 MBytes de ROM Flash.

- ESSENCIAL**
- O desenvolvimento de um programa envolve várias fases, que incluem modelação, edição, documentação, compilação, ligação com módulos pré-compilados, carregamento em memória, execução, teste e depuração. Qualquer modificação ou correção de erro deve reconhecer o ciclo do princípio.
  - A programação em linguagem de alto nível é preferível à programação em linguagem assembly, mesmo em sistemas pequenos, pois o programador não portável e é mais fácil de desenvolver. Exceptuam-se pequenos programas de muito baixo nível ou em que o desempenho seja crucial.
  - É preferível ter um programa mais claro, modular e bem estruturado do que ter um muito eficiente mas obscuro e difícil de alterar.
  - Em qualquer programa, mas muito particularmente em linguagem assembly, deve ter-se uma boa documentação. Pouco tempo depois de fazer uma rotina já ninguém se lembra bem do que fez.
  - Os fluxogramas são uma ajuda importante, não só em termos de estruturação do programa mas também de documentação.

■ O computador hospedeiro é o que corre as ferramentas de desenvolvimento (editor, compilador, etc.). O computador alvo é o que irá correr o programa (o compilador gera código-máquina para este computador). Os computadores alvo pequenos (sem sistema operativo, disco, etc.) precisam de um computador hospedeiro (sistema de desenvolvimento cruzado).

■ Estes pequenos sistemas têm tipicamente o programa em ROM (para não serem volátil) e funcionam autonomamente com uma funcionalidade fixa dentro de outro sistema, não se destinando a desenvolvimento de programas. São sistemas embutidos.

■ Os microcontroladores são circuitos integrados com um processador, RAM, ROM e periféricos, integrando praticamente tudo o que é necessário para o sistema de controlo de um sistema embutido. Não apenas conduzem a um sistema mais barato mas também mais compacto (menor espaço ocupado) como ainda a menor consumo (importante em muitas aplicações).

O microcontrolador é capaz de suportar directamente pequenas cargas como LEDs. Para construir muitos sistemas simples basta apenas ligar interruptores (para o utilizador actuar) ou outros sensores e LEDs (para visualização de informação). Para actuação de cargas com maior potência (motores, lampadas, etc.), é preciso juntar circuitos electrónicos de potência, mas todo o controlo está incluído no microcontrolador, tornando os sistemas mais simples do que usando microprocessadores, memórias e periféricos não integrados.

Os fabricantes de microcontroladores (por exemplo, a ATME<sup>65</sup> e a MICROCHIP<sup>66</sup>) vendem sistemas de desenvolvimento cruzados que permitem facilmente desenvolver programas (em C ou em linguagem assembly) num PC para executar nos respetivos microcontroladores, oferecendo assim uma forma expedita de implementar pequenos sistemas embalados num único circuito integrado.

Não há grandes diferenças entre programar um sistema baseado num microcontrolador e outro num microprocessador, e se se usar uma linguagem de alto nível a programação é praticamente transparente a este aspecto. As diferenças fundamentais prendem-se com a forma de aceder aos periféricos (no microprocessador serão acedidos por um vulgar acesso à memória, mas no microcontrolador os periféricos aparecem tipicamente como registos adicionais, o que torna o acesso mais rápido) e no tamanho e tempo de acesso das memórias (que no microcontrolador são mais pequenas, pois têm de caber dentro do circuito integrado juntamente com tudo o resto, mas que por essa mesma razão são de acesso bastante mais rápido).

A secção 0, na página 534, e o Apêndice B, na página 711, descrevem pormenores adicionais sobre microcontroladores, em particular sobre o CREPE, um microcontrolador baseado no PEPE, essencialmente através da inclusão de memórias internas e de alguns periféricos.

## 5.10 CONCLUSÕES

A grande vantagem dos computadores é o facto de poderem ser (re)programmados, não estando limitados a uma funcionalidade fixa. Com base num número limitado de operações básicas (conjunto de instruções) é possível programar uma sequência suficientemente longa e que implemente aplicações arbitrariamente complexas.

Já lá vão os tempos em que a grande dificuldade estava em construir o hardware, apenas para implementar programas razoavelmente simples, pelo menos pelos padrões actuais. Hoje em dia a grande fatia do esforço está no desenvolvimento do software, em que se concentra a dificuldade da funcionalidade a implementar. Não apenas os computadores fazem mais coisas, como em mais domínios de aplicação, como ainda de forma mais elaborada e mais amiga do utilizador, com interfaces pessoa-máquina mais elaboradas, incluindo interfaces gráficas e a cores, em dispositivos cada vez mais pequenos e que se produzem em cada vez maior número. Os telemóveis e os computadores de bolso são um bom exemplo da complexidade e poder que conseguimos hoje em dia ter literalmente na nossa mão.

Os compiladores já permitem programar um computador a um nível mais alto do que o das instruções do computador, mas o programador ainda tem de converter a especificação da funcionalidade pretendida para um algoritmo e para o programa que o implementa.

Existe um mapeamento entre as instruções de um programa de alto nível para um conjunto de instruções da linguagem assembly de cada computador. A tarefa de um compilador é analisar um programa de uma linguagem de alto nível e, usando esse mapeamento, gerar as instruções máquina do computador necessárias, gerindo recursos como os registos e a memória e usando estruturas de dados como a pilha e o montão.

O objectivo deste capítulo foi mostrar como é que este processo de geração das instruções máquina funciona tipicamente, usando o PEPE como exemplo, em programas que utilizam exclusivamente o processador e a memória. O capítulo 6 completa este cenário com os periféricos, mostrando de que forma estes afectam a programação e o funcionamento de um computador.

## 5.11 EXERCÍCIOS

- 5.1 Faça modelos com fluxogramas de diversos programas ou rotinas em linguagem assembly deste livro. Os capítulos 4 e 5 contêm os exemplos mais adequados.
- 5.2 Escreva em linguagem C um pequeno programa que declare um vector de inteiros e contenha uma função para inicializar cada um desses inteiros com o dobro do valor da sua posição no vetor. Compile manualmente, escrevendo em linguagem assembly um programa que implemente a mesma funcionalidade e estabelecendo a correspondência entre as instruções/directivas em C e as instruções/directivas em linguagem assembly.
- 5.3 Programe uma rotina que recebe um número em binário de 16 bits em complemento para 2 e produz uma cadeia de caracteres em ASCII (terminada com 00H) com o valor em decimal desse número (deve incluir o sinal se for negativo). Use o simulador para verificar o seu funcionamento (o resultado pode ser visto por inspecção da memória na sua interface gráfica).
- 5.4 Use a directiva STRING para criar uma tabela bidimensional de caracteres em ASCII (Apêndice E, na página 743). Em cada linha coloque os cinco primeiros caracteres dos seguintes alfabetos: caracteres da numeração romana, algarismos decimais, letras maiúsculas, letras minúsculas e caracteres ASCII a partir de “!”.
  - a) Assumindo que a tabela comece no endereço 1000H, indique:
    - i) O endereço do primeiro carácter em cada alfabeto;
    - ii) Qual o endereço do último byte da tabela e o seu conteúdo.
  - b) Faça um programa com a declaração da tabela e uma rotina que obtenha o carácter  $x$  ( $0 \leq x < 5$ ) do alfabeto Y;
  - c) Suponha que no seu programa substitui STRING por WORD. Explique todas as restantes alterações que terá de fazer para que o programa continue a funcionar com a mesma semântica.

<sup>65</sup> www.microchip.com

5.5 Faça um programa para converter um número entre 1 e 3999 para uma cadeia de caracteres em ASCII (terminada com 00H) com o valor correspondente em numeração romana. Use o simulador para verificar o seu funcionamento.

5.6 Imagine que o controlador de uma máquina de lavar roupa é um computador baseado no PEPE, com 8 KBytes de ROM, 2 KBytes de RAM e 128 endereços de periféricos. Estes dispositivos estão localizados em endereços contíguos, sem espaços livres entre eles.

- Quantas directivas PLACE usa e com que valores (em hexadecimal)?
- Se imagina o programa dividido em secções delimitadas pelas directivas PLACE, explique que directiva deve usar e em que secção deve colocar (gastando o mínimo possível de memória):

(i) A declaração de uma variável para conter o número de rotações por minuto de centrifugação seleccionado (400 a 1600 r.p.m.);

(ii) A declaração de uma variável do tipo vector, que contém diversos metros sobre o programa (temperatura actual da água, tempo em segundos que decorreu desde o inicio do programa, etc.);

(iii) A indicação de que o tempo da fase de centrifugação, usado em diversos pontos do programa, é 70 segundos;

(iv) A declaração de uma tabela (pредефинida) da temperatura da água pretendida em cada programa;

(v) A declaração de uma tabela (pредефинida) dos tempos de duração (em segundos) dos vários programas;

(vi) A indicação do endereço do periférico que liga aos sensores;

(vii) A rotina que lê esses sensores.

5.7 A palavra de memória que começa no endereço 100H tem o valor 5012H, que corresponde a (indique quais são as hipóteses possíveis):

- Instrução ADD R1, R2;
- Célula de memória ainda não inicializada (após reset do processador);
- Valor definido com EQU 5012H;
- Valor definido com EQU 5012H;
- Valor após PLACE 5012H;
- Valor após MOV [R1], R2, em que R1=5012H e R2=100H;
- Valor após MOV [R1], R2, em que R1=100H e R2=5012H;
- Valor após PUSH R1, R2, em que inicialmente R1=5012H e SP=100H;
- Valor após SWAP [R1], R2, em que inicialmente R1=5012H e R2=100H.

5.8 Indique a que instrução do PEPE equivale (em funcionalidade) a sequência de instruções (assumindo que o SP está inicializado):

```
PUSH R1  
RET
```

5.9 Pretende fazer uma rotina que troque os conteúdos de duas posições de memória, mas está indeciso sobre usar passagem de parâmetros por valor ou por referência. Faga as duas versões da rotina (use passagem de parâmetros em registos) e explique se deve usar uma, outra ou tanto faz.

5.10 Refaça o exercício 4.18, mas em formato de rotina. Preserve o valor de todos os registos usados (com exceção do de saída).

5.11 Transforme o Programa 4.18, na página 256, numa rotina que receba como parâmetros dois endereços x e y e um valor entre 0 e FH e devolva o número de bytes em memória entre esses dois endereços (inclusive) que têm esse valor nos seus 4 bits de maior peso. A rotina deve preservar o valor de todos os registos não usados como parâmetros e não deve assumir que  $y > x$ .

5.12 Pretende-se saber qual o menor dos valores dos bytes em memória entre dois endereços x e y (inclusive), em que  $y \geq x$ .

- Faça uma versão recursiva de uma rotina que implemente esta funcionalidade, comparando o valor do byte num dado endereço k com o valor devolvido pela rotina aplicada à gama de endereços [k+1, y]. Use passagem de valores por registos (R1=x, R2=y, R3=resultado). A rotina deve preservar o valor de todos os restantes registos que use;
- Se imediatamente antes da primeira invocação da rotina o SP valer 1000H e os endereços forem 2000H e 20FFH, indique o endereço da palavra mais profunda da pilha gasta pela rotina ao longo da execução de todas as suas invocações. Justifique a sua resposta;
- Qual seria o efeito no funcionamento do programa se em vez de 20FFH o limite superior da gama de endereços fosse 2FFFH?

5.13 Mostre (em linguagem assembly do PEPE) como declarar uma matriz de 20 linhas x 30 colunas de valores inteiros (16 bits) e como aceder ao elemento  $(i, j)$ , em que  $0 \leq i < 20$  e  $0 \leq j < 30$ , supondo que a matriz está organizada por:

- Linhas (os elementos de cada linha estão contíguos em memória);
- Colunas (os elementos de cada coluna estão contíguos em memória).

5.14 As tabelas são muitas vezes usadas para implementar filas. Uma fila é uma estrutura de dados linear, com dois extremos (a cabeça e a cauda), duas operações (escrita, que acrescenta um elemento à fila, e leitura, que retorna um elemento da lista na cabeça) e uma política de acesso FIFO (First In, First Out), em que a escrita é feita na cauda e a leitura na cabeça. Não há inserção ou remoção de elementos no meio da fila. A fila pode estar vazia (se não tiver elementos, caso em que a leitura não pode ser efectuada) ou cheia (se houver uma capacidade máxima

e esta já tiver sido atingida, caso em que a escrita de um novo elemento não pode ser efectuada). A fila pode ser implementada por uma lista ou, de uma forma mais eficiente, por uma tabela usada de forma circular com recurso a dois índices. Neste caso, a capacidade da fila é o número de elementos da tabela e os dois índices são as posições na tabela do elemento mais antigo na tabela (cabeça) e do mais recente (cauda). A forma mais simples de usar os índices é inicializá-los com 0 e incrementar a cauda após uma escrita ou a cabeça após uma leitura. Se qualquer dos índices já estiver no seu valor máximo (última posição da tabela) antes de ser incrementado, o "incremento" é na realidade voltar a 0<sup>66</sup>. A fila está vazia se os dois índices forem iguais (normalmente a 0, quando a fila é inicializada) e cheia quando a cauda estiver na posição anterior à cabeça (incluindo a situação cabeça = 0 e cauda = última posição da tabela).

- Implemente um programa que declare uma tabela com N palavras e rotinas para escrever, ler e saber o número de elementos da fila, em que a escrita e leitura só são efectuadas se a fila não estiver cheia nem vazia, respectivamente (assuma que os elementos da fila são números inteiros);
- Simule este programa no simulador, verificando o seu funcionamento e evolução do estado da fila.

### 5.15 Complete o Programa 5.15 com mais algumas funcionalidades.

- Tome as rotinas de manipulação das listas mais robustas, contemplando os casos de apontadores null (tal como referido na página 387);
- Acrescentar rotinas para:
  - Adicionar um elemento no fim da lista;
  - Contar o número de funcionários com mais de x anos (passado como parâmetro);
  - Converter a lista numa tabela, dados os endereços iniciais de ambas e sabendo quantas palavras cada ficha ocupa. Esta rotina deve percorrer os elementos da lista e copiá-los, por ordem e de forma contígua, para a tabela (assumindo que esta tem espaço suficiente);
  - Percorrer a lista numérica, executar uma rotina diferente para cada valor da idade, com o fim de efectuar um processamento diferenciado consoante a idade do funcionário. Use uma tabela de apontadores com endereços das rotinas que tratam cada idade. Como é que se faz para seguir o mesmo tratamento para idades dentro de uma dada gama, por exemplo 30-39, 40-49, etc.?
- Quais alterações teria de fazer ao programa se um dos campos da ficha do empregado fosse o seu nome (cadeia de caracteres ASCII terminada por zero)?

<sup>66</sup> O que origina o termo "circular".

## 6 O COMPUTADOR COMPLETO

O comportamento global de um computador, em termos de funcionalidade, capacidade e desempenho, depende não apenas de cada um dos seus componentes básicos mas também da forma como estão interligados e como cooperam para o objectivo comum. Este capítulo analisa essa interacção e descreve os vários problemas que se colocam para conseguir um comportamento global equilibrado.

Um dos aspectos fundamentais de um computador é o seu desempenho (rapidez com que executa os programas). Nunca chega. À medida que a tecnologia evolui, produzindo circuitos electrónicos cada vez mais rápidos e com maiores capacidades, e as arquitecturas organizam os diversos circuitos de formas cada vez mais complexas, as aplicações passam logo a implementar funcionalidades que antes não eram possíveis, conduzindo ainda a uma maior necessidade de desempenho.

As aplicações multimédia, em particular as gráficas de 3 dimensões, constituem bons exemplos. As arquitecturas dos computadores pessoais têm evoluído no sentido de proporcionar o máximo suporte em hardware para estas aplicações, quer em termos de processamento, com extensões sucessivas do conjunto de instruções para suportar processamento encadeado de amostras de som e pixels de imagens e de vídeo, quer em termos de taxa de transferência de dados da memória para a placa gráfica. Também em termos de rede de comunicação se evoluiu muito. Dos 10 Mbit/s já se passou para a escala de 10 Gbit/s, pelo que é fundamental ter uma ligação rápida com a memória.

Os processadores estão já na escala dos 3 a 4 GHz de frequência de relógio. A ligação do processador aos periféricos e à memória já se faz acima de 1 GHz. Os processadores de maior desempenho já não ligam directamente à memória e periféricos pelo clássico trio de barramentos de endereços, dados e controlo. Há controladores especiais que tomam conta dos barramentos e permitem uma comunicação o mais rápido possível entre os vários componentes do computador, mesmo directamente entre periféricos e memória, sem passar pelo processador. Neste cenário global de alto desempenho, o maior objectivo é o equilíbrio. Não adianta ter um processador muito rápido se a memória for lenta, ou a ligação aos periféricos constituir um estrangulamento. Tem de se identificar quais os pontos mais críticos e optimizar a arquitectura nesses pontos.

Este capítulo foca os pontos fundamentais do funcionamento da arquitectura como um todo (barramentos, endereçamento, interrupções, periféricos e transferência de dados de e para os periféricos), terminando com uma breve análise da avaliação de desempenho dos computadores. Para melhor estabelecer o contacto com os sistemas reais, é ainda feita uma breve descrição das principais características dos dois tipos de computadores mais vulgares no mercado: os computadores pessoais e os microcontroladores dos sistemas embutidos.

## 6.1 INTERLIGAÇÃO DOS COMPONENTES DE UM COMPUTADOR

### 6.1.1 BARRAMENTOS

Qualquer computador é constituído pelo menos pelos seguintes componentes:

- Processador;
- Memória, que poderá estar dividida em RAM e ROM;
- Periféricos, que poderão ser só de entrada, só de saída ou de entrada/saída.

**NOTA**

Com exceção de casos muito triviais ou específicos, qualquer computador precisa de RAM para poder memorizar valores. A ROM tem a vantagem de ser não volátil e usarse para conter as instruções que o processador comece a executar após ligar o computador, mas não serve para armazenar variáveis do programa.

Dada a sua flexibilidade, os PCs contêm todos os programas em RAM, pois estes são carregados a partir de disco (que armazena os programas em ficheiros). No entanto, até os PCs têm uma pequena ROM que lhes permite ler do disco as instruções que depois carregam do disco para a RAM todos os restantes programas necessários (nomencladamente, o sistema operativo).

No caso do PC, a ROM serve apenas para arrancar, mas em sistemas mais pequenos, de funcionalidade fixa (sistemas embutidos), todo o programa está em ROM e a RAM é apenas usada para dados.

Note-se que os circuitos usados nas simulações com o PEPE neste livro não têm normalmente ROM, apenas RAM. Tal destina-se apenas a tornar o circuito mais simples ( só há um tipo de memória) e só é possível dizer que estamos numa simulação e o programa é artificialmente carregado em RAM ainda antes de começar a execução do programa. Num circuito real isto não seria possível pois, mal se liga o sistema, o processador começa logo a executar o programa, mesmo se em este ter sido carregado em memória ( só com a ROM se garante que o programa já lá está quando o sistema é ligado).

Um computador pode ter múltiplas memórias (várias ROMs e várias RAMs) e vários periféricos. Os periféricos podem ser muito simples (um registo com os bits disponíveis para ligar ao mundo exterior) ou subsistemas completos (que incluem muitas vezes o seu próprio computador especializado). Um PC, por exemplo, é já um computador complexo com muitos recursos de hardware. Os computadores mais recentes até já têm mais do que um processador.

Com tantos dispositivos, o computador tem de definir:

- Um meio físico de interligação destes dispositivos, composto por:
- Barramento de endereços – Permite indicar a qual dos dispositivos o processador precisa de aceder e, se esse dispositivo for constituído por várias células individuais (RAM, por exemplo), qual a célula pretendida;

- Barramento de dados – Usado pelo processador para ler ou escrever os dados. Ao contrário dos outros barramentos, este é bidireccional (os dados podem fluir do processador para um dispositivo, no caso de uma escrita, ou no sentido oposto, no caso de uma leitura);

- Barramento de controlo – Constituído pelos sinais que controlam o acesso do processador a memória e periféricos.
- Regras de utilização do meio físico que permitam a comunicação entre os vários dispositivos.

Barramento (*bus*) é a designação dada a um conjunto de ligações relacionadas que ligam em paralelo a cada um dos dispositivos interligados. Assume-se implicitamente que um barramento interliga mais do que dois dispositivos, sendo tratado de uma ligação ponto-a-ponto, em que um dispositivo liga apenas a um outro.

Qualquer processador tem três barramentos, que tratam dos endereços, dos dados e do controlo dos dispositivos a que o processador liga. No caso do PEPE:

- Os barramentos de endereços e de dados são constituídos por 16 bits cada um;
- O barramento de controlo é constituído (para já) por dois sinais de um bit cada:

  - RD, activo a 0 quando o processador está a fazer uma leitura de um dispositivo;
  - WR, activo a 0 quando o processador está a fazer uma escrita num dispositivo.

A Fig. 4.7, na página 213, mostra o circuito que tem sido usado para as simulações do PEPE e que já contempla estes sinais, mas inclui apenas uma memória. A Fig. 6.1 mostra um exemplo com mais dispositivos. Assume-se, para já, que todos os dispositivos têm uma largura (dos dados) de 16 bits, embora alguns possam ter apenas 8 bits (a secção 6.1.5.1 tratará deste aspecto). Sobre esta figura podem fazer-se os seguintes comentários:

- O computador é constituído por uma ROM (ROM1), dois módulos de RAM (RAM1 e RAM2) e quatro periféricos (P1 a P4);
- O barramento de dados (16 bits, D15..D0) liga a todos os dispositivos, pois é por aqui que o processador comunica com todos os dispositivos. Este barramento é uma espécie de corredor com várias portas, em que cada uma dá acesso a um dispositivo. Apenas os dados circulam pelo corredor. Cada dispositivo "vê" o resto do sistema a partir da sua porta, sem se deslocar;

Apenas duas operações podem ser efectuadas com este sistema:

- Escrita – O processador coloca um valor no barramento de dados. Todos os dispositivos "vêm" o valor (todos estão a "olhar" para o corredor), mas assume-se que apenas um dispositivo memorizará esse valor;
- Leitura – O processador solicita que um determinado dispositivo lhe forneça um valor. Como resposta, esse dispositivo envia o valor pretendido pelo barramento de dados. Mais uma vez todos os dispositivos "vêm" esse valor, pois

todos ligam ao barramento de dados, mas apenas o processador memoriza esse valor, num dos seus registos;

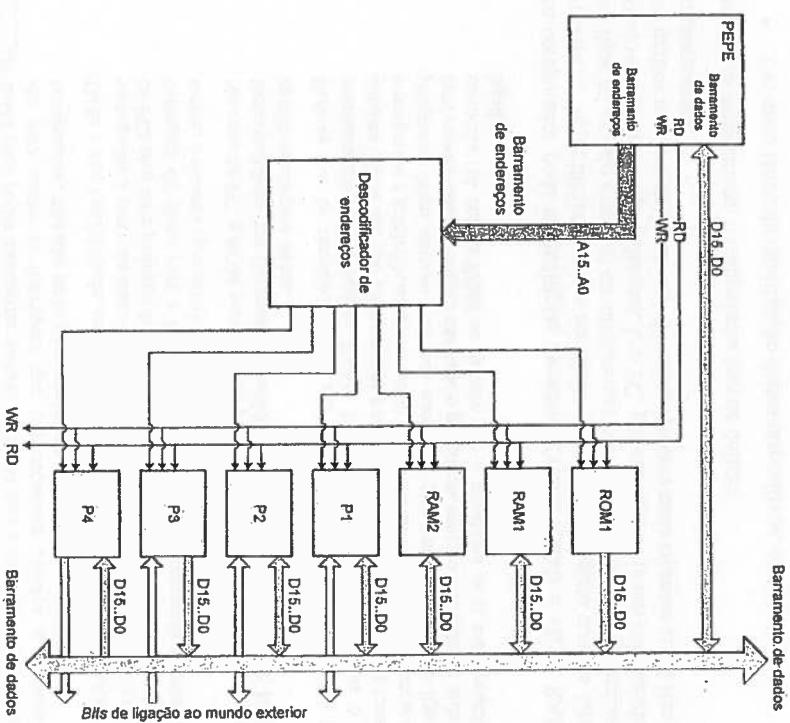


Fig. 6.1 - Barramentos do PEPE, ilustrando a ligação do processador aos dispositivos

- O barramento de controlo (sinais RD e WR) liga a todos os dispositivos, para que estes saibam qual das operações o processador está a efectuar;
- O barramento de endereços (16 bits, A15..AO) liga a um sistema que descodifica os endereços, produzindo sinais que identificam e activam o dispositivo a que o processador quer aceder (e só esse). A descodificação de endereços é tratada já na secção 6.1.3);
- Este sistema não é uma democracia! O processador (o mestre) é o único dispositivo que pode tomar a iniciativa de enviar informação para os barramentos. Os restantes dispositivos (os escravos) só o podem fazer (e apenas no barramento de dados) como resposta a um pedido por parte do processador;

O processador não distingue a memória dos periféricos. Para ele, todos os dispositivos são iguais, suportando as duas operações referidas. Cabe ao programador (através das instruções do programa) saber se está a fazer a operação certa sobre o dispositivo correcto. Assim, a designação "acesso à memória" abrange implicitamente os acessos à memória e aos periféricos;

**Nota**

Alguns processadores mais antigos ainda suportam um espaço de endereçamento separado para os periféricos. Isto significa que há instruções diferentes para acceder à memória e aos periféricos e o processador tem barramentos de controlo separados para acesso à memória e aos periféricos.

Esta organização usou-se nos primeiros tempos dos microprocessadores, em que se encarava as entradas/saiadas como um conceito separado do de processamento. Por outro lado, o número de bits disponíveis para endereço era muito limitado e assim evitava-se que os periféricos gastassem endereços que deviam ser reservados para memória.

Depois, a tecnologia evoluiu e actualmente já não se justifica a complexidade adicional de tratar os periféricos de forma diferente da memória, pelo que os periféricos e memória estão mapeados no mesmo espaço de endereço e são tratados de forma uniforme (*periféricos mapeados em memória, ou memory-mapped input/output*).

Com apenas duas operações este sistema nem parece complicado, mas levanta alguns problemas, que são enunciados e resolvidos nas secções seguintes.

## 6.1.2 OPERAÇÕES DE LEITURA E ESCRITA

Olhando para a Fig. 6.1, consegue inferir-se que nem todos os dispositivos são iguais em termos de suporte às operações de leitura e escrita. Nomeadamente:

A ROM1 só pode ser lida, por definição<sup>57</sup> (razão pela qual a sua ligação ao barramento de dados não é bidireccional);

A RAM1 e RAM2 devem poder ser lidas e escritas, no seu normal funcionamento; Os periféricos P1 e P2 são de entrada/saída (leitura/escrita), o que é indicado pelas setas bidirecionais que os ligam ao barramento de dados (e também nas ligações ao mundo exterior);

O periférico P3 é só de entrada (leitura), pois a seta que o liga ao barramento de dados tem apenas o sentido periférico → processador e a seta de ligação ao mundo exterior é apenas de entrada;

<sup>57</sup>ROM = Read Only Memory (memória só de leitura). Esta designação destina-se a fazer o contraste com as RAMs, que podem ser escritas pelo processador. A informação é colocada nas ROMs por mecanismos próprios (secção 6.5.3.1, na página 534), e que não são os acessos normais de escrita do processador.

- O periférico P4 é só de saída (escrita), pois a seta que o liga ao barramento de dados tem apenas o sentido processador → periférico e a seta de ligação ao mundo exterior é apenas de saída.

**PROBLEMA 5** Como é que se impede o processador de efectuar uma operação de leitura ou escrita sobre um dispositivo que não a suporta?

**SOLUÇÃO** Não se consegue impedir, pois é sempre possível haver erros no programação. Pode é proteger-se o dispositivo contra operações incorrectas. Liga-se ao dispositivo o sinal (RD ou WR) correspondente à operação que ele suporta mas não se liga o outro. Se o processador efectuar a operação errada sobre esse dispositivo, ele ignorá-la-á pois não tem o sinal da sua operação activa. O programa funcionará provisivelmente de forma incorrecta, mas é responsabilidade do programador testá-lo e corrigir os erros até estar tudo correcto.

Mesmo que o dispositivo só suporte uma operação, precisa da informação de qual operação o processador está a efectuar, senão podem ocorrer as situações seguintes (admitindo que já se sabia que o processador pretenda efectuar a leitura ou escrita sobre esse dispositivo):

- Se o programa ler um dispositivo que só suporta escrita, nem o processador nem o dispositivo colocam um valor no barramento de dados, pelo que este assume um valor aleatório (dependente do ruído electromagnético, como o produzido por um motor eléctrico ou pelo interruptor da lâmpada de uma sala). O processador le assim um valor incorrecto e o dispositivo memoriza um valor que não devia (destruindo o valor que lá devia estar). Com o dispositivo ligado apenas ao WR, o processador continua a ler um valor incorrecto, mas pelo menos o dispositivo não destrói o valor que lá tinha;
- A escrita num dispositivo que só suporta leitura constitui uma situação mais grave, podendo eventualmente causar avarias no hardware. Tanto o processador como o dispositivo colocariam um valor no barramento de dados, causando um conflito entre sinais que poderiam danificar os circuitos de ligação ao barramento e imitizando o computador. Se o dispositivo ligar apenas ao RD, o resultado da operação de escrita é o valor a escrever perder-se, isto é, não ser memorizado pelo dispositivo. O efeito é apenas uma eventual incorrecção no funcionamento do programa, que o programador pode corrigir sem receio de entretanto ter danificado o computador.

A Tabela 6.1 sumariza as operações suportadas por cada um dos dispositivos e os sinais que ligam a cada um deles.

A Fig. 6.2 mostra o circuito corrigido em termos de barramento de controlo (os sinais RD e WR).

| DISPOSITIVO | OPERAÇÃO | BARRAMENTO DE CONTROLO |
|-------------|----------|------------------------|
| RÔM1        | Ler      | Escrita                |
|             | Sim      | Não                    |
|             |          | RD                     |
| RAM1        | Sim      | Sim                    |
| RAM2        | Sim      | Sim                    |
| P1          | Sim      | Sim                    |
| P2          | Sim      | Sim                    |
| P3          | Sim      | Não                    |
| P4          | Não      | Sim                    |
|             |          | WR                     |

Tabela 6.1 - Operações suportadas pelos dispositivos e relação com os sinais do barramento de controlo

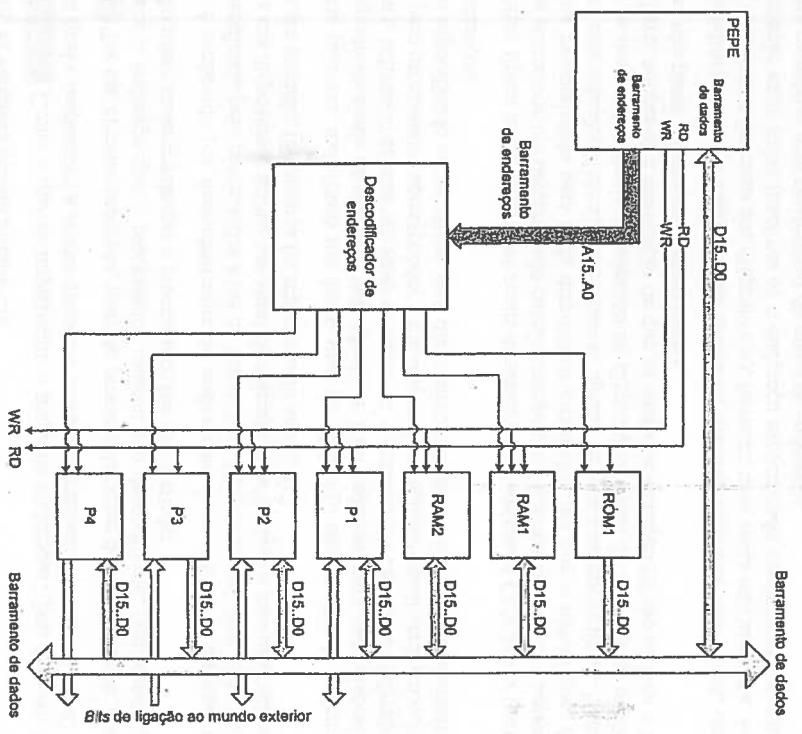


Fig. 6.2 - Circuito da Fig. 6.1 corrigido em termos de barramento de controlo (os sinais RD e WR só devem ligar aos dispositivos que suportam a operação respectiva)

### 6.1.3 DESCODIFICAÇÃO DE ENDEREÇOS (DE PALAVRA)

**NOTA/MERCHANTE** – Toda esta secção (até à secção 6.1.4, na página 430) assume que o processador suporta apenas endereçamento de palavra. Não é esse o caso do PEPE nem da generalidade dos processadores comerciais, que suportam endereçamento de byte. No entanto, esta limitação temporária é essencial para não introduzir todos os detalhes ao mesmo tempo. A secção 6.1.4, na página 430, volta a introduzir o endereçamento de byte.

#### 6.1.3.1 SELECCÃO DE DISPOSITIVO A ACEDER

**PROBLEMA 2** Como é que o processador especifica qual o dispositivo a que quer aceder? Em escrita só quer escrever num deles, não em todos ao mesmo tempo. Em leitura, apenas um dispositivo poderá enviar o seu dado, senão cada dispositivo tenta enviar o seu e há conflitos no barramento (é como várias pessoas a falar simultaneamente – ninguém se entende).

**SOLUÇÃO** Especificar um endereço em cada acesso à memória que identifique unicamente um e um só dispositivo.

Esta solução já foi abordada na secção 4.2.2, na página 192, e não é exclusiva dos computadores. Para efectuar um telefonema, por exemplo, é preciso identificar o interlocutor pelo seu "endereço" (número de telefone).

O PEPE tem um barramento de endereços de 16 bits, o que significa que consegue especificar  $2^{16}$  (64 K, ou 65.536) endereços diferentes. Ou seja, tem um espaço de endereçamento que varia do endereço 0000H até ao endereço FFFFH.

O processador não sabe, na realidade, que dispositivos ligam aos barramentos. Quando quer aceder à memória/periféricos, coloca os valores correctos nos barramentos e espera pela resposta (acesso em leitura) ou simplesmente que tudo tenha corrido bem e que algum dispositivo tenha recebido e memorizado o valor (acesso em escrita). A secção 6.1.6 descreve estas operações em maior detalhe.

Por outro lado, cada computador terá o seu conjunto específico de memórias e periféricos, disponíveis em endereços que também variam de computador para computador. A tabela que indica que dispositivos estão disponíveis em quais endereços designa-se mapa de endereços. A Tabela 4.1 (na página 194) exemplifica, podendo ver-se que há endereços ocupados, em que o processador pode aceder a células de memória e a periféricos, e outros livres, em que nenhum dispositivo será acedido se o processador os utilizar. Este mapa de endereços é específico de um dado computador. Quem o projecta deve garantir que não há dois dispositivos mapeados no mesmo endereço, senão haverá conflitos no acesso a esse endereço. Portanto:

- O funcionamento do processador não pode depender de quantos nem quais dispositivos lhe estão ligados;
- O funcionamento de cada dispositivo não deve variar consoante esteja disponível num ou outro endereço ou esteja inserido num computador com mais ou menos dispositivos.

Caso contrário, as variantes seriam infundáveis.

**PROBLEMA 3** Como é que se implementa o mapa de endereços? Isto é, quem é determina qual o dispositivo a aceder quando o processador especifica um endereço?

**SOLUÇÃO** Ter um circuito separado, quer do processador quer dos dispositivos, que "olhe" para o endereço que o processador indica e o descodifique, notificando o dispositivo nesse endereço de que o processador lhe está a aceder.

Este circuito é conhecido por descodificador de endereços. A sua missão é analisar o endereço especificado pelo processador à luz do mapa de endereços e, caso esse endereço corresponda a um dispositivo, activar um sinal de selecção que indique a esse dispositivo que ele está a ser acedido, tal como já foi representado na Fig. 6.2.

Dito por outras palavras, transforma um único sinal de 16 bits (o barramento de endereços) num conjunto de sinais de selecção individuais de 1 bit cada, um para cada dispositivo do sistema, activando apenas um sinal de cada vez e somente no caso de o endereço especificado pelo processador corresponder a um dispositivo (nenhum sinal será activado se o endereço especificado estiver livre, sem dispositivo atribuído). Ou seja, implementa o mapa de endereços.

O termo genérico típico usado para estes sinais de selecção individuais é *Chip Select* (CS), indicando que selecciona um determinado circuito integrado.<sup>58</sup> O nome de cada um destes sinais é depois especializado para cada dispositivo incluindo algo que o identifique. A Fig. 6.3 ilustra esta solução. Naturalmente, trata-se apenas de uma convenção típica. Nada obriga a que os normes dos sinais de selecção de dispositivo sejam de uma forma ou de outra. Neste livro, adopta-se a convenção de que os sinais de selecção CS são activos a 0 (mas uma vez não passa de uma convenção usual).

Assim, o descodificador deste exemplo terá activa no máximo uma das oito saídas, das quais só são usadas sete (número dos dispositivos), havendo uma saída não utilizada. As sete saídas deverão estar todas inactivas se o endereço especificado corresponder a uma zona livre (sem memórias nem periféricos) do mapa de endereços.

#### 6.1.3.2 IMPLEMENTAÇÃO DO MAPA DE ENDEREÇOS

Entretanto, o sistema não tem apenas sete endereços, mas sim sete dispositivos, cada um com potencialmente muitos endereços. Considere-se o caso de uma RAM, por exemplo, com várias células de memória. A geração de um sinal CS individual para cada célula está fora de questão, pois uma memória actual pode ter milhões de células. Teremos de nos contentar com um só CS para cada dispositivo do sistema, mas depois é preciso arranjar forma de endereçar cada célula individualmente.

Embora o dispositivo seleccionado possa ser mais complexo do que um só circuito integrado, é uma designação histórica.

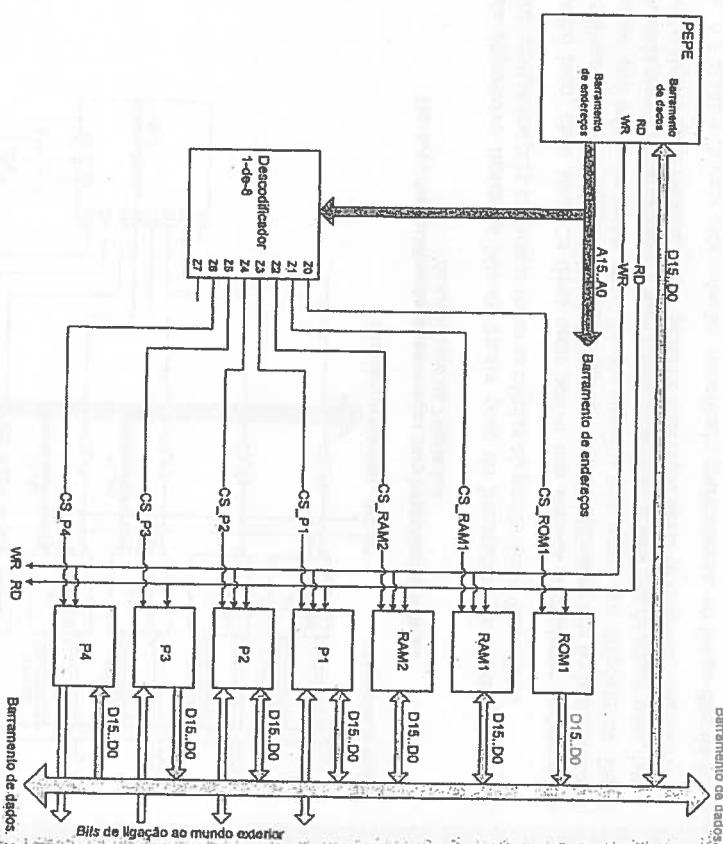


Fig. 6.3 - Descodificação de endereços, activando um dos sinais individuais de seleção

**PROBLEMA:** No caso de um dispositivo que tem várias células, como as memórias, como é que ele sabe a qual das suas células o processador está a querer aceder?

**SOLUÇÃO:** O circuito de descodificação de endereços deve activar o sinal CS desse dispositivo durante toda a gama de endereços correspondentes às suas células individuais e os P bits de menor peso do barramento de endereços devem ligar ao dispositivo para especificar exactamente qual a célula a aceder ( $P = \log_2 N$ , em que  $N$  é o número de células do dispositivo). Por exemplo, uma RAM com 4 K células precisa de 12 bits, uma vez que  $12 = \log_2 4096$ , ou  $4096 = 2^{12}$ .

Para resolver este problema precisamos de saber quantas células tem cada dispositivo e em que gama de endereços o pretendemos colocar. Ou seja, é preciso fazer o mapa de endereços. Como exemplo, podemos assumir o mapa da Tabela 6.2, que também indica a dimensão em células (palavras) de cada dispositivo. É de realçar que nesta fase ainda estamos a considerar apenas endereçamento de palavra (o endereçamento de byte será reintroduzido na secção 6.1.4).

| DISPOSITIVO | DIMENSÃO (CÉLULAS) | ENDERECO INICIAL | ENDERECO FINAL | NÚMERO DE BITS DO BARRAMENTO DE ENDEREÇOS |
|-------------|--------------------|------------------|----------------|-------------------------------------------|
| ROM1        | 4 K (4096)         | 1000H            | 0000H          | 0FFFH                                     |
| RAM1        | 2 K (2048)         | 800H             | 1000H          | 17FFH                                     |
| RAM2        | 1 K (1024)         | 400H             | 200H           | 23FFH                                     |
| P1          | 8                  | 8                | 300H           | 3007H                                     |
| P2          | 4                  | 4                | 400H           | 4003H                                     |
| P3          | 1                  | 1                | 500H           | 500H                                      |
| P4          | 1                  | 1                | 600H           | 600H                                      |
|             |                    |                  |                | 0                                         |

Tabela 6.2 - Mapa de endereços usado no circuito da Fig. 6.4

Os periféricos P1 e P2 incluem várias células, que nos periféricos são conhecidas por portos. Um porto é a unidade mínima de acesso a um periférico e pode corresponder a:

- Um conjunto de bits disponíveis para ligar ao mundo exterior (de entrada ou de saída);
- Registros internos ao periférico, usados tipicamente para programar o seu comportamento ou verificar o seu estado.

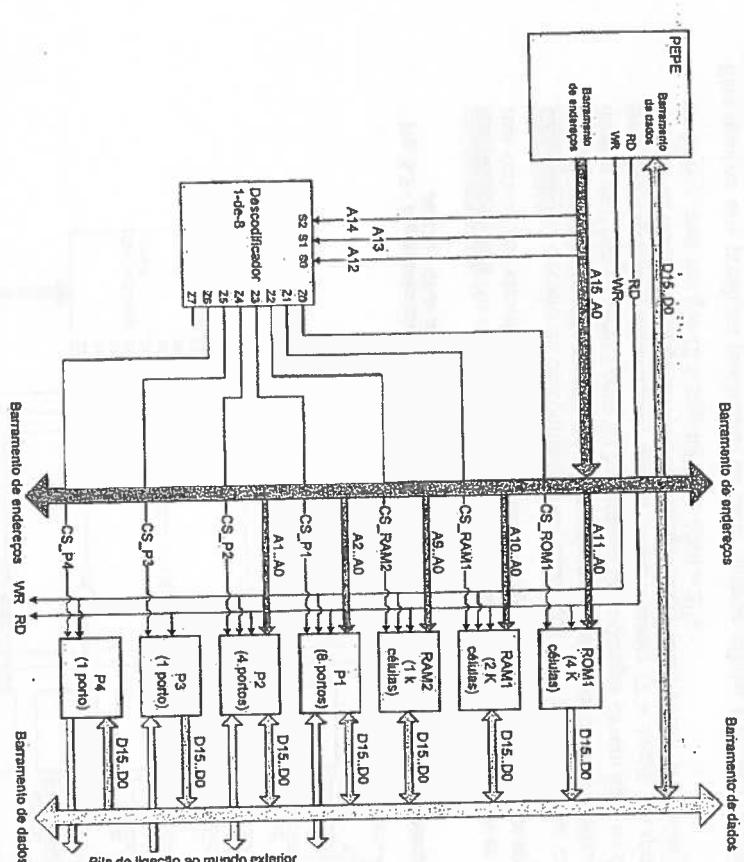
Todos os portos de um periférico são accedidos através do mesmo sinal CS desse periférico e são individualizados através do endereço, tal como uma célula dentro de uma memória. No entanto, uma memória pode ter milhões de células, enquanto os portos de um periférico se contam normalmente às unidades. Neste exemplo específico, só os periféricos P1 e P2 têm vários portos (oito e quatro, respectivamente), enquanto os periféricos P3 e P4 têm apenas um porto cada um.

A Tabela 6.2 explicita o número (em hexadecimal) de células/portos de cada dispositivo, de modo a facilitar a determinação da gama de endereços de cada dispositivo, depois de decidir qual o endereço da primeira célula/porto de cada dispositivo (coluna "Endereço inicial").

Note-se que os vários dispositivos não ocupam os endereços de forma contígua. Há "buracos" livres entre as gamas de endereços dos dispositivos, para que os endereços iniciais de cada dispositivo sejam localizados com 1000H endereços de intervalo. Pode parecer que seria mais fácil se se atribuissem os endereços aos dispositivos de forma contínua, mas tal só seria verdade se todos os dispositivos tivessem o mesmo número de células. O que é mais fácil para a implementação da descodificação de endereços é os endereços iniciais dos vários dispositivos estarem igualmente espalhados (secção 6.1.3.3).

A Fig. 6.4 ilustra este aspecto. Cada dispositivo tem indicada a sua dimensão (em células ou portos). Pode notar-se que o barramento de dados liga a cada dispositivo apenas com o número de bits necessário para endereçar todas as suas células/portos. Assim (conferir com a Tabela 6.2), a ROM1 tem 4 K células e precisa de 12 bits (A11..A0). A RAM1 tem 2 K células e precisa de 11 bits (A10..A0). A RAM2 tem 1 K células e precisa de 10 bits

(A9..A0). O periférico P1 tem oito portos e portanto precisa de 3 bits (A2..A0). Os periféricos P3 e P4 só têm um porto e por conseguinte não precisam de todo do barramento de endereços.



**Fig. 6.4 - Descodificação de endereços para implementar o mapa de endereços da Tabela 6.2**

A cada dispositivo ligam os bits de menor peso do barramento de endereços, tanta quantos necessários para endereçar todas as células ou portos desse dispositivo.

Por outro lado, cada sinal CS deve estar activo nos acessos a qualquer endereço do dispositivo respectivo, o que se consegue ligando ao descodificador *bits* do barramento de endereços que sejam de maior peso do que aqueles que ligam aos dispositivos. Desta forma, cada sinal CS corresponde a uma combinação fixa desses *bits* de maior peso, isto é, que se mantenha igual durante toda a gama de endereços desse (e só desse) dispositivo.

A Fig. 6.5 ajuda a ilustrar esta solução, mostrando graficamente, ao longo dos 64 K do espaço de endereçamento do PEPE:

**A evolução dos bits de maior peso do barramento de endereços, A15..A11.**

Para melhor perceber esta figura, pode imaginar-se o processador a efectuar sequencialmente acessos a todos os endereços, começando em 0000H e terminando em FFFFH. A Fig. 6.5 mostra (i) qual o dispositivo que será accedido à medida que os endereços vão subindo (tal como indicado pela Tabela 6.2) e (ii) a evolução dos bits de maior peso do barramento de endereços.

Pode facilmente constatar-se que o bit A12 é o mais baixo que mantém o seu valor para todos os endereços da gama de qualquer dispositivo. Não admira, pois a ROM precisa de ligar aos bits A11..A0 para endereçar todas as suas células. O A11 mantém-se constante

Como precisamos de sete sinais de CS, usamos um descodificador 1-de-8 (ficando um livre, não usado), que requer 3 bits para seleccionar uma das oito possibilidades. Portanto, os bits que ligam ao descodificador são o A12, A13 e A14.

A Tabela 6.3 completa a Tabela 6.2, mostrando o número de bits de menor peso do barramento de endereços que cada dispositivo precisa e a evolução dos três bits ao longo do mapa de endereços.

| Dispositivo | Dimensão (células) | Endereço INICIAL | Endereço FINAL | BITS DO BARRAMENTO DE ENDEREÇOS | A14, A13, A12 |
|-------------|--------------------|------------------|----------------|---------------------------------|---------------|
|             | DECIMAL            | HEXA.            | HEXA.          |                                 |               |
| RAM1        | 4 K (4096)         | 1000H            | 0000H          | 0FFH                            | 12 (A11..A0)  |
| RAM1        | 2 K (2048)         | 800H             | 100H           | 17FFH                           | 11 (A10..A0)  |
| RAM2        | 1 K (1024)         | 400H             | 200H           | 23FFH                           | 10 (A9..A0)   |
| P1          | 8                  | 8                | 300H           | 307H                            | 3 (A2..A0)    |
| P2          | 4                  | 4                | 400H           | 403H                            | 2 (A1..A0)    |
| P3          | 1                  | 1                | 500H           | 500H                            | 0             |
| P4          | 1                  | 1                | 600H           | 600H                            | 0             |
| Livre       | ?                  | ?                | 700H           | ?                               | 111           |

Tabela 6.3 - Evolução dos bits A14, A13 e A12 (do barramento de endereços) usados

para seleccionar um dos sinais CS na saída do descodificador

### 6.1.3.3 DESCODIFICAÇÃO PARCIAL DOS ENDEREÇOS

A Fig. 6.5 expressa graficamente o mapa de endereços referido na Tabela 6.3 e parece assumir que o processador pode encontrar dispositivos apenas nos endereços indicados. Aliás, parece que apenas uma fração pequena do espaço de endereços está atribuída a dispositivos, estando o resto livre.

É apenas uma ilusão! O que a Fig. 6.5 e a Tabela 6.3 expressam é o conjunto de endereços em que o processador pode aceder aos vários dispositivos, mas na realidade só omissoas no que toca aos restantes endereços.

Consideremos, por exemplo, a gama de endereços que vai de 1800H até 1FFFH. Que corresponde ao espaço de endereços entre a RAM1 e a RAM2. Se o processador fizer um acesso ao endereço 1800H, por exemplo, será que algum dispositivo é seleccionado?

Este endereço difere do endereço 1000H apenas no bit A11, que está a 1 em 1800H e a 0 em 1000H. Isto quer dizer que:

- O valor dos bits A14, A13 e A12 é o mesmo nos dois casos (001), o que significa que a saída do descodificador activada é a mesma ( $z_1$ , ou CS\_RAM1);
- Os bits A10..A0, que são os que identificam a célula individual dentro da RAM1, têm o mesmo valor (todos a 0).

Ou seja, o único bit diferente (o A11) é precisamente um que o sistema de descodificação de endereços da RAM1 não usa.

Na prática, isto quer dizer que a primeira célula da RAM1 (bits A10..A0 todos a 0) pode ser accedida tanto pelo endereço 1000H como pelo endereço 1800H. Na realidade, todos os endereços entre 1000H e 17FFH podem ser accedidos também pelos endereços entre 1800H e 1FFFFH, respectivamente. É como se esta segunda gama de endereços fosse um espelho da primeira.

Continuando a olhar para a Fig. 6.5, descobre-se facilmente que, se a RAM1 tem um espelho:

- A RAM2 tem três (usa apenas 1/4 da gama de endereços 2000H a 3000H). Por exemplo, os endereços 2000H, 2400H, 2800H e 2C00H conduzem todos à mesma célula;
- O periférico P1 tem 511 espelhos (usa apenas 1/512, ou 8 em 4096, dos endereços). Por exemplo, o primeiro porto deste periférico pode ser accedido por qualquer dos seguintes endereços: 3000H, 3008H, 3010H, 3018H, etc.;
- O periférico P4 tem 4095 espelhos (usa apenas 1/4096 dos endereços possíveis). Qualquer endereço entre 6000H e 6FFFFH permite aceder ao periférico.

Tudo isto porque o descodificador 1-de-8 da Fig. 6.4 usa apenas os bits A14, A13 e A12, em que cada uma das saídas se mantém activa em 4096 endereços contíguos, e cada dispositivo usa apenas o número de bits do barramento de endereços estritamente necessário para endereçar cada uma das suas células, sem olhar aos restantes.

No realidade, o cenário é ainda mais "espelhado", pois toda a metade superior do espaço de endereçamento (8000H a FFFFH, em que o bit A15 está a 1) é um espelho da metade inferior! Isto sucede porque em lado nenhum se usa o A15 para seleccionar ou não os dispositivos, pelo que tanto faz estar a 0 como a 1. Afinal, o espaço de endereçamento está todo gasto, com exceção dos endereços correspondentes à saída z7 do descodificador (7000H a 7FFFH, ou F000H a FFFFH).

O espelhos de gamas de endereços não são em si uma coisa errada ou sequer indesejável. O projectista do circuito tem é de estar consciente de que os espelhos existem e de que as gamas de endereços correspondentes não estão livres. De resto, não têm qualquer consequência. A descodificação de endereços em que se admite ter espelhos designa-se descodificação parcial dos endereços.

O que se ganha é facilidade de descodificação. Aliás, o mapa de endereços inicialmente apresentado na Tabela 6.2 distribui os dispositivos de 1000H em 1000H precisamente para que a descodificação se resuma ao descodificador e aos bits de endereço necessários em cada dispositivo com várias células ou portos. No entanto, é sempre possível discriminar todos os bits de endereço e eliminar totalmente os espelhos, usando essencialmente dois pequenos truques (já representados na Fig. 6.6):

- O espelho maior, o que deriva de não se usar os bits de endereço de peso maior do que os usados no descodificador, eliminando simplesmente, garantindo que o descodificador só funciona quando esses bits de maior peso têm um determinado valor. Nos restantes casos, o descodificador deve ter todas as suas saídas inactivas. Neste exemplo, isto quer dizer que o descodificador só deve funcionar quando o a15 estiver a 0. Normalmente, os descodificadores têm uma entrada de controlo (e) que permite ligar ou desligar o funcionamento do descodificador. Na Fig. 6.6, admitiu-se que a entrada de controlo é activa a 1, pelo que basta inserir uma negação para que o descodificador funcione com a15 a 0;

change on taxes, health insurance costs, or employer taxes. In addition, the state may choose to change its tax base, or it may change its tax rates. In either case, the state may have to make up the difference between the new tax base and the old one. This may mean higher taxes on individuals, corporations, or other entities.

#### Conclusion

In this article, we have examined the economic consequences

of the proposed changes in the individual mandate. We have shown that the proposed changes will result in significant increases in the cost of health insurance premiums for most individuals.

#### Notes

##### 1. *Id.*

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

See, e.g., *Proposed Rule, Health Insurance Premium Subsidies Under the Affordable Care Act, 75 Fed. Reg. 10,000 (Feb. 11, 2010)* (hereinafter "Proposed Rule").

### **6.1.3.4 DESCODIFICAÇÃO DE MAPAS DE ENDEREÇOS IRREGULARES**

Nem sempre é possível distribuir os vários dispositivos de forma uniforme pelo espaço de endereçamento como se pode ver na Tabela 6.2. Basta imaginar, por exemplo, o sistema da Fig. 6.2 mas em que se pretende agora que todos os dispositivos estejam em endereços consecutivos a partir de  $0000H$ , tal como indicado na Tabela 6.4.

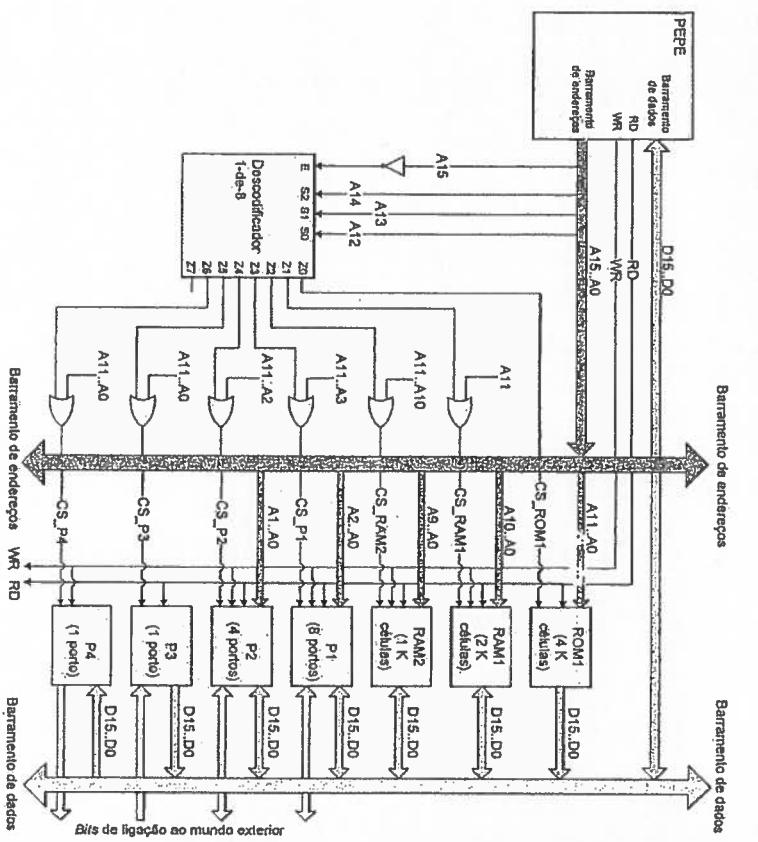


Tabela 6.4 - Mapa de endereços usado no circuito da Fig. 6.4

Note-se que o que varia são os endereços iniciais dos dispositivos. As suas dimensões (em células ou portos) e os bits do barramento de endereços que a eles ligam manufaturam-se. Agora já não se pode usar apenas o simples descodificador de 1-de-8, mas continua a ser necessário gerar sete sinais CS, que devem ficar activos apenas na gama de endereços especificada para cada dispositivo. Como o mapa de endereços é irregular e sem espaços intermédios, não pode haver espelhos pelo meio das gamas de endereços dos vários dispositivos. Se o mapa de endereços é irregular, a solução provavelmente também o será.

Uma hipótese, com base em descodificadores, é apresentada pela Fig. 6.7. O truque nesta solução é reconhecer dois grupos de dispositivos, um com um número significativo de endereços (as memórias) e outro com poucos endereços (os periféricos). Assim, usa-se um descodificador 1-de-8 para cada um dos grupos, em que o segundo é controlado a partir do primeiro.

descritores são ac-

- Como a menor das memórias tem 1 K palavras de capacidade, as saídas do primeiro descodificador estão activas precisamente em 1 K endereços, o que permite cobrir 8 K endereços (o que chega para contemplar todas as memórias). Isto implica que:

Habilmente os espelhos que ocorrem dentro de cada gama de endereços, em que uma dada saída do descodificador está activa, obrigando a que o CS que chega a cada dispositivo só fique activo quando simultaneamente a saída correspondente do descodificador esteja activa e todos os bits de peso menor que os usados no descodificador e que não estejam ligados ao dispositivo tenham o valor 0. No caso do periférico P1, por exemplo, tanto a saída Z3 como os bits A11..A3 devem ser 0 (o que se detecta com um simples OR). A ROM não precisa disto, pois usa todos os bits peso menor do que os usados no descodificador.

## **na descodificação de endereços**



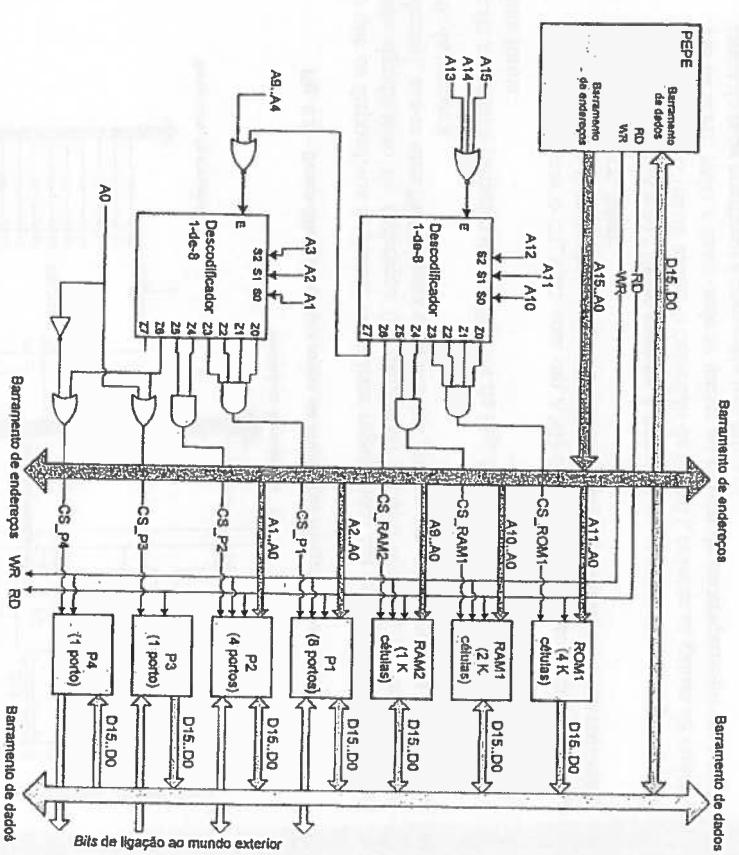


Fig. 6.7 - Decodificação de um mapa de endereços irregular com descodificadores resíduos.

- O sinal CS\_BA11, que deve estar activo em 2 K endereços, tenha de ser obtido pelo AND das duas saídas seguintes do descodificador;
  - O sinal CS\_RAM2, que deve estar activo em 1 K endereços, liga directamente à saída z6 do descodificador;
  - Os bits de seleção do descodificador sejam os bits A12-A10 (o A10 varia em cada 1 K endereços);
  - A entrada de controlo do descodificador seja o OR dos bits A15-A13, o que significa que o descodificador só funciona quando estes bits são 0 (no infinito do espaço de endereçamento).

O segundo descodificador só deve funcionar durante os primeiros 16 endereços do último 1 K dos endereços descodificados pelo primeiro descodificador (conferir com Tabela 6.4). Por esta razão, a sua entrada de controlo é obtida pelo OR da

sáida 27 do primeiro descodificador e dos bits de endereço A9..A4 (que têm de estar a 0), o que tem as seguintes consequências:

- Os bits de selecção do segundo descodificador são os bits A3..A1, o que faz com que este descodificador tenha cada uma das suas oito saídas activas em dois endereços apenas (o que perfaz os 16 endereços referidos);
  - O sinal CS\_P1, que deve estar activo em oito endereços, é obtido pelo AND das primeiras quatro saídas do descodificador;
  - O sinal CS\_P2, que deve estar activo em quatro endereços, é obtido pelo AND das duas saídas seguintes do descodificador;
  - Os sinais CS\_P3 e CS\_P4, que devem estar activos apenas num endereço, já estão numa granularidade de descodificação de endereços abaixo do que o segundo descodificador permite (cada uma das suas saídas está activa durante dois endereços). Para respeitar o mapa de endereços da Tabela 6.4, a solução é usar a mesma saída, Z6, mas discriminar um endereço e o seguinte à custa do A0 (que varia entre endereços consecutivos).

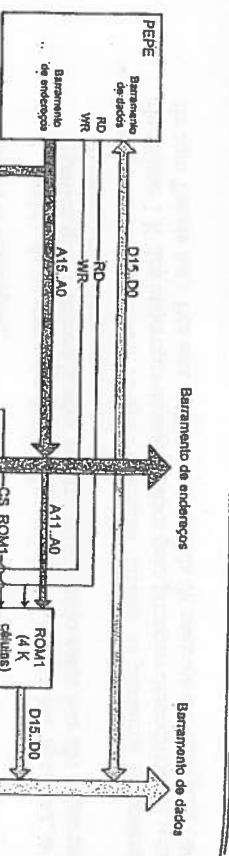
### **6.1.3.5 DESCODIFICAÇÃO DE ENDEREÇOS PROGRAMÁVEL**

### 6.1.3.5 DESCODIFICAÇÃO DE ENDEREÇOS PROGRAMAVEL

- A PROM está sempre activa, isto é, o seu CS (enquanto dispositivo) está sempre a 0. Desta forma, há sempre um valor presente nas saídas da PROM, o seleccionado pelos seus próprios bits de endereço;
  - Cada bit de saída da PROM liga a um dispositivo a seleccionar (é o seu CS). Em cada palavra da PROM apenas um bit pode estar a 0 (todos a 1 significa que nenhum dispositivo está a ser seleccionado, o que é útil para implementar zonas livres do espaço de endereços);
  - Alguns dos bits do barramento de endereços ligam aos bits de endereço da PROM, seleccionando a palavra da PROM adequada, com o valor dos sinais pretendido.

Uma forma trivial seria ligar todos os bits de endereço à PROM, reservando uma palavra para cada um dos endereços possíveis. Isto permitiria a flexibilidade máxima de especificar qual o CS activado para cada um desses endereços. O problema é que isso exigiria uma PROM gigantesca, razão pela qual se usa a PROM como um descodificador programável e se liga apenas a alguns dos bits do barramento de endereços.

Uma forma trivial seria ligar todos os bits de endereço à PROM, reservando uma palavra para cada um dos endereços possíveis. Isto permitiria a flexibilidade máxima de especificar qual o CS activado para cada um desses endereços. O problema é que isso exigiria uma PROM gigantesca, razão pela qual se usa a PROM como um descodificador programável e se liga apenas a alguns dos bits do barramento de endereços.



A Tabela 6.5 apresenta o conteúdo da PROM1 nas suas palavras iniciais (as restantes têm todos os bits todos a 1, isto é, com todos os CS inactivos).

| $A_{15}A_4A_3A_2A_1A_0$ | $D_3D_2D_1D_0$ | DISPOSITIVO SELECIONADO |
|-------------------------|----------------|-------------------------|
| 000 000                 | 1110           | ROM1                    |
| 000 001                 | 1110           | ROM1                    |
| 000 010                 | 1110           | ROM1                    |
| 000 011                 | 1110           | ROM1                    |
| 000 100                 | 1101           | RAM1                    |
| 000 101                 | 1101           | RAM1                    |
| 000 110                 | 1011           | RAM2                    |
| 000 111                 | 0111           | periféricos             |
| 001 000                 | 1111           | nenhum                  |
| 111 111                 | 1111           | nenhum                  |

A PROM2 substitui o segundo descodificador da Fig. 6.7 e as portas lógicas associadas, da seguinte forma:

- Recebe todos os bits de endereço  $A_9..A_0$  (ou seja, todos os que a PROM1 não recebe). Isto permite a flexibilidade total de geração dos sinais CS dos periféricos dentro deste bloco de 1 K palavras.

Para gerar o CS\_P1, por exemplo, basta ter 8 palavras com o bit  $D_0$  a 0 (e os restantes bits a 1) para contemplar os 8 portos deste periférico. Estas 8 palavras têm de ser consecutivas e começar num endereço múltiplo de 8, mas aparte estas restrições pode mudar-se facilmente o mapa de endereços, substituindo a PROM2 por outra com outro conteúdo;

O bit  $A_{10}$  da PROM2 é usado como sinal de controlo para desactivar ( $P_0$  a 1) todas as suas saídas, o que implica que a PROM2 deverá ter toda a segunda metade do seu conteúdo (com o bit  $A_{10}$  da PROM2 a 1) preenchida exclusivamente com 1s. Esta pode parecer uma forma muito inefficiente de activar e desactivar a parte dos periféricos, mas poderá ser muito mais barata em termos de implementação, pois a alternativa seria ter componentes externos à PROM2, o que tem sempre custos acrescidos. A capacidade da PROM2 (2 K palavras de 4 bits) é baixa face às existentes comercialmente, pelo que não há qualquer custo adicional em usar este pequeno truque.

A Tabela 6.6 apresenta o conteúdo da PROM2 nas suas palavras iniciais (as restantes têm todos os bits todos a 1, isto é, com todos os CS inactivos).

Fig. 6.8 - Descodificação de um mapa de endereços irregular com PROMs

Dado que os dispositivos se podem considerar organizados em dois grupos, um com um número significativo de endereços (as memórias) e outro com poucos endereços (os periféricos), usa-se uma PROM para cada um dos grupos, em que a segunda é controlada a partir da primeira.

A PROM1 substitui o primeiro descodificador da Fig. 6.7 e as portas lógicas associadas, da seguinte forma:

- Em vez de gerar o CS\_ROM1 com um AND das primeiras quatro saídas, a PROM1 tem o bit  $D_0$  a 0 (activo) durante as primeiras quatro palavras, com um raciocínio idêntico para CS\_RAM1;

- Os bits  $A_{15}, A_{14}$  e  $A_{13}$  ligam também à PROM1. Isto permite, sem qualquer alteração do circuito (basta mudar o conteúdo da PROM1), colocar as gamas de endereços da ROM1, RAM1 e RAM2 onde se quiser no espaço de endereçamento em vez de terem de estar contíguos e com estes três bits a 0;

| A10, A9, ..., A3, A2, A1, A0<br>(CONTROLO, A9, ..., A3, A2, A1, A0) | D3, D2, D1, DO | Dispositivo<br>SELECIONADO |
|---------------------------------------------------------------------|----------------|----------------------------|
| 00 ... 0000                                                         | 1110           | P1                         |
| 00 ... 0010                                                         | 1110           | P1                         |
| 00 ... 0011                                                         | 1110           | P1                         |
| 00 ... 0100                                                         | 1110           | P1                         |
| 00 ... 0101                                                         | 1110           | P1                         |
| 00 ... 0110                                                         | 1110           | P1                         |
| 00 ... 0111                                                         | 1110           | P1                         |
| 00 ... 1000                                                         | 1101           | P2                         |
| 00 ... 1001                                                         | 1101           | P2                         |
| 00 ... 1010                                                         | 1101           | P2                         |
| 00 ... 1011                                                         | 1101           | P2                         |
| 00 ... 1100                                                         | 1011           | P3                         |
| 00 ... 1101                                                         | 0111           | P4                         |
| 00 ... 1110                                                         | 1111           | nenhum                     |
| 00 ... 1111                                                         | 1111           | nenhum                     |
| ...                                                                 | ...            | ...                        |
| 11 ... 1111                                                         | 1111           | nenhum                     |

Tabela 6.6 - Conteúdo da PROM2 para implementar o mapa de endereços da Tabela 6.4

### 6.1.4 DESCODIFICAÇÃO DE ENDEREÇOS (DE BYTE)

Toda a secção 6.1.3, da página 416 até aqui, foi feita com base no endereçamento de palavra, para não introduzir os problemas todos ao mesmo tempo. A cada palavra fez-se corresponder um só endereço, assumindo que só era possível aceder a uma palavra inteira (e não a cada byte individualmente).

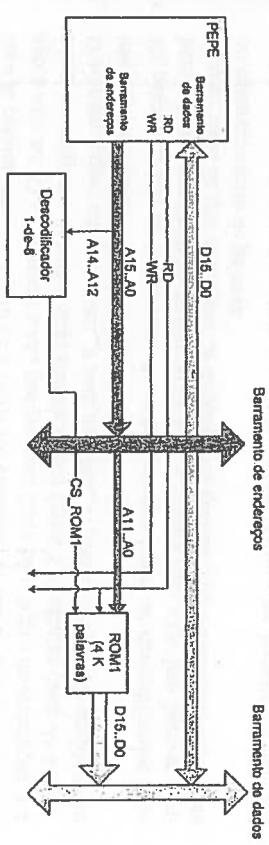
No entanto, o PEPE funciona com endereçamento de byte (secção 4.4, na página 196), o que tem consequências em termos de descodificação de endereços. Cada palavra gasta agora dois endereços, pois é possível aceder apenas a um byte de cada vez e como tal todos os bytes precisam de ter o seu próprio endereço.

**PROBLEMA 6.5** O PEPE suporta acessos em palavra (com MOV) e em byte (com MOVE), o que implica não apenas poder enviar dados completos (palavras) pelo barramento de dados mas também "metades de dados" (bytes). Como é que se implementa esta funcionalidade?

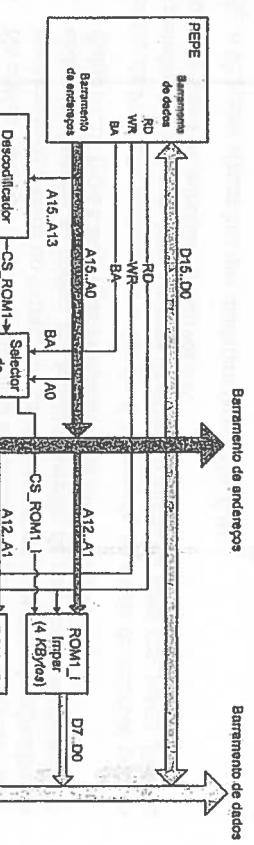
**SOLUÇÃO 6.5** Dividir o barramento de dados do PEPE e dos dispositivos ao meio, com sinais de controlo que permitam especificar se se quer aceder apenas a um byte (usando metade do barramento de "dados") ou à uma palavra inteira (usando o barramento de dados completo).

Como uma palavra em memória tem dois bytes com endereços consecutivos, um dos bytes terá endereço par (bit A0 = 0) e outro ímpar (bit A1 = 1). Metade dos 16 bits do barramento de dados (D15..D8) liga a um dos bytes, enquanto a outra metade (D7..D0) liga ao outro byte. Uma vez que isto sucede em todas as palavras, toda uma metade da

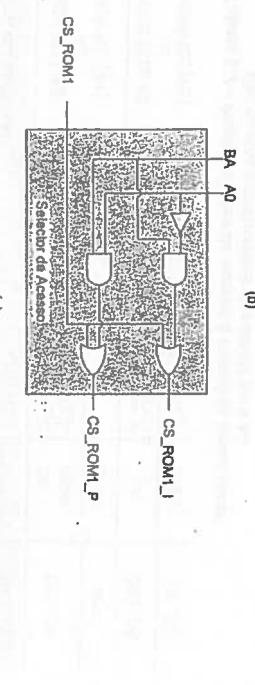
memória (endereços pares) liga a uma metade do barramento de dados, e o mesmo sucede à metade dos endereços ímpares face à outra metade do barramento de dados. No caso do PEPE, é a metade par dos endereços que liga à metade alta do barramento de dados (D15..D8), enquanto a metade ímpar liga à metade baixa (D7..D0). Poderia também ser ao contrário. A secção 6.1.5.2 discute este tópico.



(a)



(b)



(c)

Fig. 6.9 - Descodificação de endereços. (a) - De palavra; (b) - De byte;

(c) Detalhe de geração dos sinais de seleção par e ímpar

A Fig. 6.9 ilustra a solução, usando a ROM1 como exemplo de dispositivo acedido e fazendo o contraste com a forma como a secção 6.1.3 lidou com a descodificação dos endereços de palavra.

A Fig. 6.9a representa a ROM1 como um só dispositivo de 16 bits de largura e com um só sinal de selecção. É um dispositivo monolítico, que só suporta acesso a uma palavra inteira (16 bits) de cada vez. Não é possível, portanto, aceder (nãoeadamente em escrita) apenas a um byte. Em suma, não suporta endereçamento de byte.

Na Fig. 6.9b, a ROM1 foi dividida em duas metades, isto é, duas ROMs de 8 bits de largura, cada uma com 4 KBytes, que no total da sua capacidade são equivalentes à ROM1 da Fig. 6.9a, com 4 K palavras (4 KBytes + 4 KBytes).

Como cada uma delas tem apenas 8 bits de dados, liga apenas a uma metade do barramento de dados (D15..D8 ou D7..D0). A ROM1\_P (par) contém os bytes com endereço par a ROM1\_I (ímpar) contém os bytes com endereço ímpar.

Possuem sinais de selecção individuais, produzidos a partir do sinal de selecção global da ROM1 (CS\_ROM1) pelo circuito da Fig. 6.9c (descrito a seguir), e é esta característica que permite ao processador aceder-lhes individualmente, ou seja, suportar endereçamento de byte. Por outro lado, também é possível aceder a uma palavra inteira (acesso em 16 bits), actuando os dois sinais de selecção simultaneamente.

O processador também tem de estar preparado para o endereçamento de byte. Para além dos sinais RD e WR, o barramento de controlo passa a ter mais um sinal, BA (Byte Address sing), que fica:

- Activo (a 1) quando o processador faz um acesso de byte (com uma instrução MOV.B – ver secção 4.10.4.7, na página 230);
- Inactivo (a 0) quando o processador faz um acesso de palavra (com uma instrução MOV – ver secção 4.10.4.6, na página 229).

Por outro lado, quando o processador executa uma instrução MOV.B, é a paridade do endereço a aceder (na prática, o valor do bit A0) que determina a metade da memória a que se accede e por conseguinte a metade do barramento de dados que se deve usar. Se o endereço for par (A0=0), deve aceder-se à parte da memória par e usar-se a metade do barramento de dados D15..D8. Se o endereço for ímpar (A0=1), deve aceder-se à parte da memória ímpar e usar-se a metade D7..D0.

A Tabela 6.7 indica o que sucede nas quatro combinações possíveis de acesso, de acordo com o tipo de acesso (palavra ou byte, que o processador expressa no sinal BA) e paridade do endereço acedido (indicada por A0).

Notas sobre esta tabela:

- Se o acesso for de palavra (MOV), o endereço acedido tem de ser par, caso em que as duas metades da memória são acedidas simultaneamente pelos 16 bits do barramento de dados, tal como indicado na Fig. 6.9b. Ou seja, são acedidos dois bytes ao mesmo tempo: o que fica no endereço par indicado na instrução MOV e o que fica no endereço seguinte, que é ímpar;

bytes ao mesmo tempo: o que fica no endereço par indicado na instrução MOV e o que fica no endereço seguinte, que é ímpar;

| EXEMPLOS DE INSTRUÇÃO |               | R1    | BA | A0 | SINAL DE SELECCÃO ACTIVO | MEMÓRIA ACEDIDA | BARRAMENTO DE DADOS USADO |
|-----------------------|---------------|-------|----|----|--------------------------|-----------------|---------------------------|
| Palavra               | MOV [R1], R2  | par   | 0  | 0  | CS_ROM1_P                | par + impar     | D15..D0                   |
|                       | MOV R2, [R1]  | ímpar | 0  | 1  | erro!                    | erro!           | erro!                     |
| byte (8 bits)         | MOVB [R1], R2 | par   | 1  | 0  | CS_ROM1_P                | par             | D15..D8                   |
|                       | MOVB R2, [R1] | ímpar | 1  | 1  | CS_ROM1_I                | ímpar           | D7..D0                    |

Tabela 6.7 - Possibilidades de acesso à memória resultantes das quatro combinações dos sinais BA e A0

O acesso de palavra com endereço ímpar é um erro e nunca deve suceder. A secção 6.1.5.3 explica porquê;

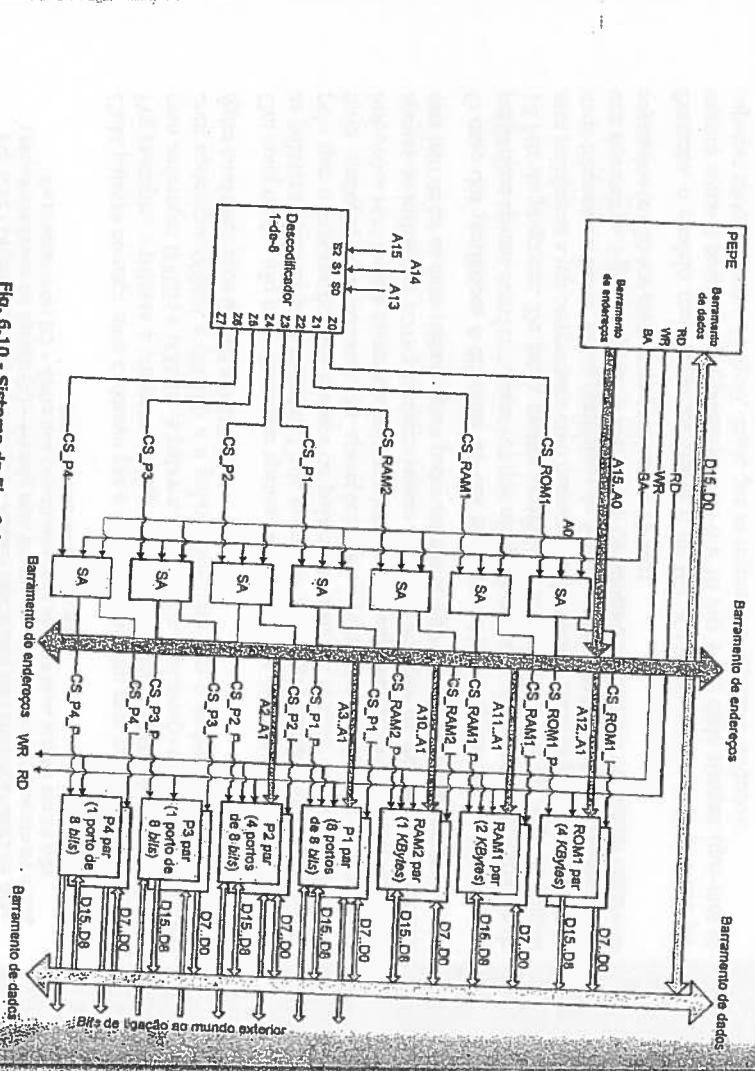
No acesso em byte, os bytes localizados em endereços pares só podem ser acedidos através da metade do barramento de dados D15..D8, e os bytes localizados em endereços ímpares só podem ser acedidos através da metade do barramento de dados D7..D0 (pois é assim que as ligações estão feitas fisicamente).

O circuito da Fig. 6.9c implementa a Tabela 6.7, gerando os sinais de selecção para as duas metades da ROM1 com base no sinal de selecção global da ROM1 (CS\_ROM1). Note-se que todos os sinais de selecção são activos a 0.

No Fig. 6.9b houve alguns bits que mudaram face à Fig. 6.9a:

- Cada metade da ROM1 continua a receber 12 bits de endereço, visto que continua a ter 4 K células (embora agora de 8 bits). No entanto, em vez do A11..A0, recebem agora A12..A1. O A11 passou a A12 porque com bytes há o dobro de endereços, e a ROM1, no conjunto das suas duas metades, tem agora 8 K células (mas de 8 bits). O A0 nem sequer entra na ROM1\_P nem na ROM1\_I, pois é este bit que decide se se usa uma ou outra;
- Os pesos dos bits do barramento de endereços que ligam ao descodificador também sobem numa unidade, passando de A14..A12 para A15..A13. Isto deve-se também ao facto de que com bytes os dispositivos gastam o dobro dos endereços face ao endereçamento de palavra.

A Fig. 6.10 mostra como implementar o endereçamento de byte no circuito da Fig. 6.4, com vários dispositivos, usando a mesma técnica que na Fig. 6.9b (um selector de acesso, que com base no BA, no A0 e no sinal de selecção de um dispositivo gera os dois sinais de selecção para as duas metades, par e ímpar, desse dispositivo). Para poupar espaço, as duas metades foram quase sobrepostas, assumindo-se que ligam aos mesmos bits (do barramento de dados e de controlo) e a única diferença reside no sinal de selecção e na metade do barramento de dados.



Mj. 6.10 - Sistema da Fig. 6.4 mas agora com descodificação de endereços.

**C**LIQUE SA (Selectar de Acesso) é o descrito pela Flg. 6-9.

SIMPLY CIO 96 - DESCODIFICANDO

círculo da Fig. 6.11 segue bem o comportamento da descodificação de endereços. Tomando

- Comportamento dos dispositivos em relação a uma operação (leitura ou escrita);
  - que não suportem);
  - Funcionamento do descodificador;
  - Endereçamento interno dos dispositivos com vários endereços;
  - Mapa de endereços e espelhos;
  - Acessos à memória em *byte* e em palavra.

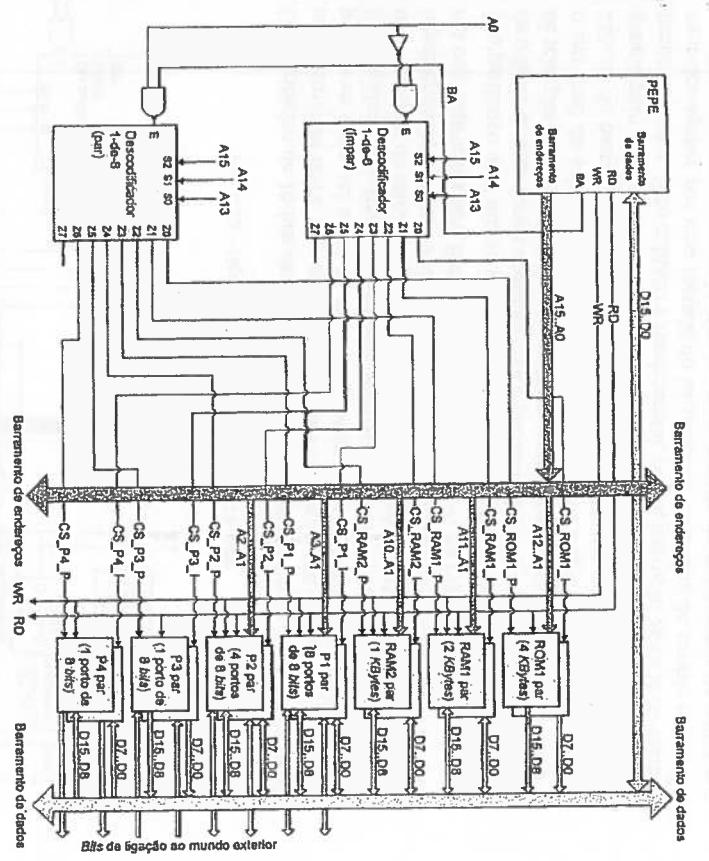


Fig. 6.11 - Descodificação de endereços de byte separados para os endereços pares e ímpares

**3.1.3 INSTRUÇÕES DE ENDEREÇAMENTO**

## **6.1.5 IMPACTE DO ENDEREÇAMENTO DE BYTE**

**6.15.1 ORGANIZAÇÃO DA MEMÓRIA EM BYTES**

A Fig. 6.12 descreve em maior detalhe a organização do espaço de endereçamento da memória e dos periféricos quando há endereçamento de *byte*.

Conceptualmente, uma memória é uma sequência linear de células de largura de 8 bits e endereços consecutivos (Fig. 6.12a). No entanto, em termos físicos está dividida ao meio, em que cada metade liga à sua metade do barramento de dados (Fig. 6.12b).

A divisão da memória (e periféricos) em duas metades de 8 bits de largura cada não é feita em grandes blocos mas sim em endereços alternados, sendo o bit 00 que indica se

### 6.15.1 ORGANIZAÇÃO DA MEMÓRIA EM BYTES

## 6.1.5 IMPACTE DO ENDEREÇAMENTO DE BYTE

um valor num dado endereço está na metade par ou na metade ímpar. Isto significa que cada palavra está mapeada sempre nas duas metades (Fig. 6.12c).

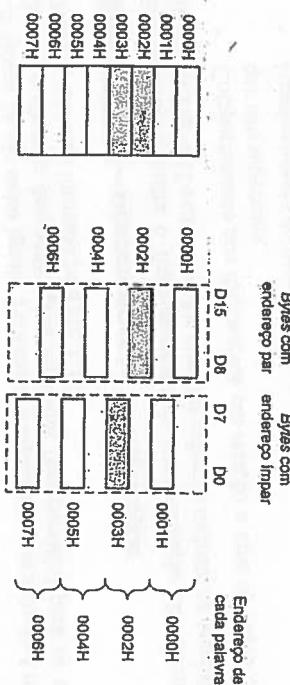


Fig. 6.12 - Organização da memória. (a) - Modelo conceptual com bytes com endereço par e bytes com endereço ímpar; (b) - Modelo real, ilustrando a ligação ao barramento de dados (sequência linear de bytes); (c) - Bytes que constituem cada palavra num acesso em 16 bits (sempre com endereço par)

Cada palavra começa num endereço par e inclui o byte com esse endereço e o seguinte. Por exemplo, a palavra a círculo na Fig. 6.12 tem o endereço 0002H e inclui os bytes com endereços 0002H e 0003H. A palavra anterior tem o endereço 0000H (com os bytes com endereços 0000H e 0001H) e a palavra seguinte tem o endereço 0004H (com os bytes com endereços 0004H e 0005H).

Um computador não pode funcionar apenas com metade da memória, uma vez que todas as palavras usam as duas metades. Cada instrução de código-máquina tem uma palavra, pelo que o processador faz acessos de palavra à memória sempre que vai buscar uma nova instrução para executar. As operações com a pilha têm a palavra como base. As instruções MOV leem e escrevem palavras inteiras. E assim sucessivamente. Na realidade, apenas as instruções MOVB permitem acesso a apenas um byte da memória. No entanto, esta tem de ter as duas metades para poder ser usada por todas as outras operações.

O caso dos periféricos é diferente do das memórias. Nada impede que se lide com os periféricos apenas com MOVB, uma vez que não podem conter programas nem variáveis de 16 bits do programa. De facto, muitos periféricos têm apenas 8 bits de largura, e ligam sem problemas a um computador cujo barramento de dados é mais largo. Aliás, a situação mais frequente é haver vários periféricos de 8 bits que são usados de forma independente uns dos outros, ligando apenas a metade do barramento de dados e ocupando endereços separados (e não aos pares para perfazer os 16 bits).

Portanto, o cenário mais frequente não é o da Fig. 6.11 mas sim o da Fig. 6.13, que mostra como é possível ligar periféricos de 8 e 16 bits a um dado sistema (que tem de suportar endereçamento de byte), desde que se tenham os seguintes cuidados:

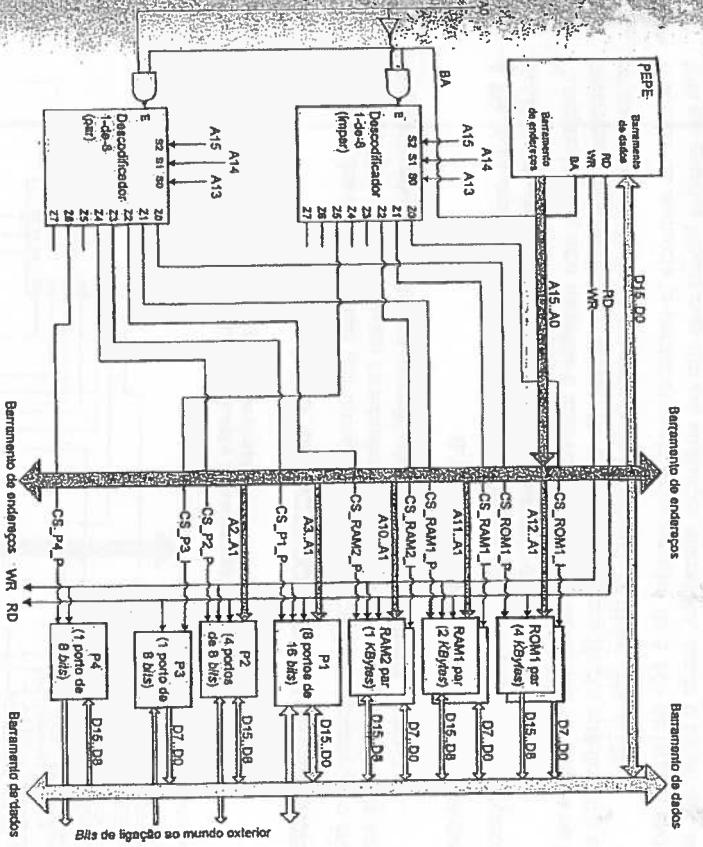


Fig. 6.13 - Ligação de periféricos de 8 e 16 bits

Un periférico de 16 bits deve ser accedido com instruções MOV. Também se pode usar instruções MOVB, mas apenas se o periférico tiver o seu barramento de dados partido ao meio, tal como as memórias. Nem todos os periféricos suportam isto (que implica dois sinais de seleção separados). Se um periférico de 16 bits só tiver um sinal de seleção, este deve ligar-se ao descodificador de endereços par e o periférico deve ligar a todo o barramento de dados (D15,D0) e ser accedido apenas com instruções MOV. Este é o caso do periférico P1 na Fig. 6.13;

Um periférico de 8 bits só pode ligar a metade do barramento de dados, mas tem de o fazer de uma forma coerente com a paridade do endereço em que o seu sinal de seleção estiver activo. Por exemplo, os periféricos P2 e P4 têm apenas 8 bits e o seu sinal de seleção vem do descodificador par. Logo, devem ligar aos bits D15,D8 do barramento de dados. Já o periférico P3 tem o seu sinal de seleção gerado pelo descodificador ímpar, razão pela qual liga aos bits D7,D0 do barramento de dados. Este cuidado é fundamental, pois a instrução MOVB leu ou escreve os dados apenas por uma metade do barramento, escolhida de acordo com o bit AC. Caso contrário, o processador usa uma metade do barramento de dados e o

periférico a outra, o que naturalmente impede os dados de circularem correctamente;

No acesso aos periféricos de 8 bits deve usar-se apenas a instrução MOVB. O uso de MOV num acesso em leitura<sup>69</sup>, por exemplo, fará com que 8 bits sejam lidos correctamente do periférico, mas os 8 restantes ficarão com valores aleatórios (e os 8 bits ficarão a 0);

Os endereços em que os vários portos de um periférico de 8 bits estão disponíveis não são consecutivos, mas sim de 2 em 2. O periférico P2, por exemplo, que está nesse caso, tem o sinal de selecção (CS\_P2\_P) ligado ao descodificador par, o que implica que só é accedido em endereços pares (também não adiantaria ser accedido em endereços ímpares, pois liga apenas aos bits D15..D8 do barramento de dados). O bit A0 é usado para distinguir as duas metades dos dispositivos e nem sequer entra no P2 (que recebe apenas os bits de endereço A2..A1), mas terá de ser forçadamente 0 (endereço par) para o periférico ser accedido correctamente. Consulte o periférico P2 podem ser accedidos nos endereços 8000H, 8002H, 8004H e 8006H e não nos endereços ímpares (só o MOVB é suportado).

| Dispositivo | Dimensão (bytes) | Endereço Inicial | Endereço Final | Pode Usar-se MOV? | Pode Usar-se MOVB? |
|-------------|------------------|------------------|----------------|-------------------|--------------------|
| ROM1        | 8 K (8192)       | 2000H            | 0000H          | Sim               | Sim                |
| RAM1        | 4 K (4096)       | 1000H            | 2000H          | Sim               | Sim                |
| RAM2        | 2 K (2048)       | 800H             | 4000H          | Sim               | Sim                |
| P1          | 16               | 10H              | 6000H          | Sim               | Não                |
| P2          | 4                | 4                | 800H           | Sim               | Não                |
| P3          | 1                | 1                | A000H          | A000H             | Não                |
| P4          | 1                | 1                | C000H          | C000H             | Não                |

Tabela 6.8 - Mapa de endereços do circuito da Fig. 6.13, com endereçamento de byte

### 6.1.5.2 ENDEREÇAMENTO LITTLE-ENDIAN E BIG-ENDIAN

A Fig. 6.12 indica claramente que no PEPE o byte de maior peso do barramento de dados (D15..D8) liga à metade par da memória, enquanto o byte de menor peso (D7..D0) liga à metade ímpar. Porque é que é assim e quais são as implicações resultantes?

Na realidade, não tem que ser assim. A Fig. 6.14 mostra os dois métodos de ligação possíveis, o da Fig. 6.12 (que o PEPE usa) na metade superior e o oposto na metade

<sup>69</sup> Necessariamente a um endereço par, tal como referido na página 198.

inferior. A diferença fundamental entre os dois consiste na definição de qual metade do barramento de dados (D15..D8 ou D7..D0) liga a qual metade da memória (par ou ímpar). Como exemplo, assume-se que o processador efectuou a seguinte instrução (em ambos os métodos):

```
MOV R1, 002H ; endereço da célula de memória a aceder
MOV R1, 456H ; valor a escrever na célula de memória
MOV [R1], R2 ; escreve a palavra na memória
```

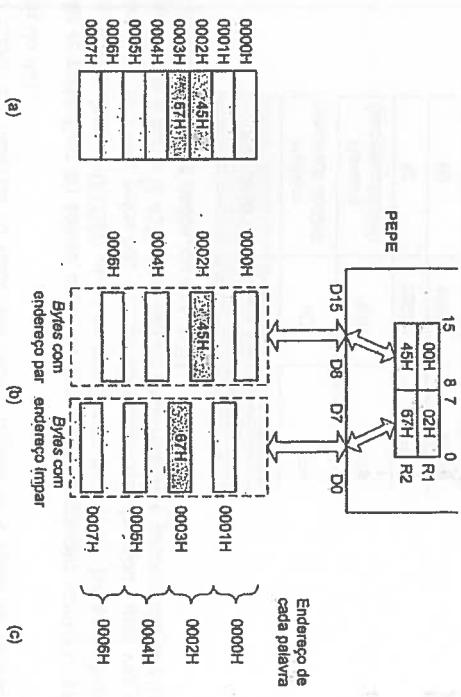


Fig. 6.14 - Métodos de ligar o processador à memória: (a, b, c) – Big-endian; (d, e, f) – Little-endian (o processador foi omitido para não estar repetido)

Os dois métodos de ligar o processador à memória têm as seguintes designações e semântica:

- Big-endian (Fig. 6.14a, b, c) – A metade de maior peso do barramento de dados (D15..D8) liga à metade par da memória. Isto quer dizer que o byte 45H (o de

maior peso de R2) é escrito no byte par da palavra com endereço 0002H (Fig. 6.14c), que é o byte com endereço 0002H. Dito de outra forma, o byte de maior peso do R2 vem primeiro que o byte de menor peso, quando considerarmos a sequência crescente de endereços do modelo do endereçamento de byte (Fig. 6.14a);

**Little-endian** (Fig. 6.14d, e, f) – A metade de menor peso do barramento de dados (D7..D0) liga à metade par da palavra. Isto quer dizer que o byte 67H (o de menor peso de R2) é escrito no byte par da palavra com endereço 0002H (Fig. 6.14f), que é o byte com endereço 0002H. Dito de outra forma, o byte de menor peso do R2 vem primeiro que o byte de maior peso, quando considerarmos a sequência crescente de endereços do modelo do endereçamento de byte (Fig. 6.14d).

Na prática, portanto, a diferença está em qual byte do registo é escrito primeiro na sequência crescente de endereços: o de maior peso (*big-endian*, Fig. 6.14a) ou o de menor peso (*little-endian*, Fig. 6.14d).

Outra forma equivalente de ver esta diferença é considerar o endereço de uma palavra (sempre par) e ver qual byte (o de maior ou o de menor peso) de um registo escrito nessa palavra em memória fica localizado nesse endereço par. Nos processadores *big-endian* será o byte de maior peso, enquanto nos *little-endian* será o byte de menor peso. Mais uma vez, esta diferença é ilustrada pela Fig. 6.14a e pela Fig. 6.14d, com a palavra no endereço de memória 0002H (palavra que ocupa os endereços 0002H e 0003H).

Mas não se pense que basta trocar as ligações aos dispositivos para um mesmo processador passar de *big-endian* a *little-endian* ou vice-versa. Há uma diferença intrínseca no comportamento do processador, usando um método ou outro, em relação ao acesso de byte (com MOVB), descrita na Tabela 6.9.

É importante reconhecer dois aspectos que são iguais nos dois métodos e que podem ser confirmados na Fig. 6.14:

- Nos acessos de palavra, o processador usa os bits do barramento de dados exactamente pela mesma ordem do registo (Fig. 6.14b). Ou seja, os bits 15..0 de um registo ligam exactamente pela mesma ordem aos bits 15..0 do barramento de dados (não há nenhuma troca de bits interna ao processador).
- Na sequência linear crescente dos endereços de byte, o endereço par de uma palavra vem sempre imediatamente antes do endereço ímpar.

Enquanto um dado computador usar o mesmo método para aceder à memória em 16 bits, não há problemas em qualquer dos métodos. A diferença reside na ordem pela qual os bytes estão dispostos em memória, mas como as leituras usam o mesmo método que a escrita e o acesso é em palavra (os dois bytes são acedidos), os valores escritos na memória são perfeitamente recuperados.

Os problemas podem surgir em dois tipos de situações:

- No mesmo computador, quando um programa acede em modo de byte (com MOVB) a bytes escritos na memória em modo de palavra (com MOV);
- Ao copiar dados da memória de um computador para outro que use um método diferente de ligação à memória. Esta operação relaciona-se tipicamente com formatos de ficheiros de dados e corresponde a um problema que não pode ser ignorado, obrigando a efectuar uma reordenação adequada dos bytes.

Para ilustrar o cuidado a ter quando se misturam acessos em palavra e em byte no mesmo computador, vamos usar a situação da Fig. 6.14, executando as instruções seguintes no caso dos dois métodos, *big-endian* ou *little-endian*:

```
MOV R1, 0002H ; endereço da célula de memória a aceder
MOV R2, 4567H ; valor a escrever na célula de memória
MOV [R1], R2 ; escreve a palavra na memória
MOVB R3, [R1] ; leitura da memória em byte
MOV R4, [R1] ; leitura da memória em palavra
```

A Tabela 6.9 indica, para cada um dos dois métodos, os valores da memória e dos registos após a execução de todas estas instruções. Pode verificar-se que, apesar das diferenças no conteúdo da memória entre os dois métodos, cada método é consistente consigo próprio no que toca ao acesso em palavras (o valor lido para o registo R4 é igual ao escrito na memória a partir do R2).

O mesmo já não se pode dizer do acesso em byte, apesar de a instrução MOVB R3, [R1] especificar o mesmo endereço (0002H) que a instrução MOV R4, [R1]. No caso do *big-endian*, o processador lê o byte pelos bits D15..D8 do barramento de dados, uma vez que o endereço especificado é par, e é 45H. No caso do *little-endian*, o processador lê o byte dos bits D7..D0 do barramento de dados, ou seja, 67H.

| RECURSO | REGISTRO         | VALOR   |
|---------|------------------|---------|
|         | Byte 15..0       | LEITURA |
| Memória | (endereço 0002H) | 45H     |
| Memória | (endereço 0003H) | 67H     |
| R1      |                  | 0002H   |
| R2      |                  | 4765H   |
| R3      |                  | 0045H   |
| R4      |                  | 4765H   |

Tabela 6.9 - Valores na memória e nos registos depois a execução das instruções anteriores, com os métodos *big-endian* e *little-endian*

Os processadores de 32 bits suportam normalmente acessos de byte, 16 bits (meia-palavra) e 32 bits (palavra). As regras atrás enunciadas para a ligação *big-endian* e *little-endian* mantêm-se. Cada palavra da memória ocupa agora 4 endereços, mas continua a ser conhecida pelo endereço múltiplo de 4, que deve ser o mais baixo. Nesse endereço em

memória fica localizado o byte de maior peso de uma palavra (*big-endian*) ou de menor peso (*little-endian*), tal como indicado na Fig. 6.15.

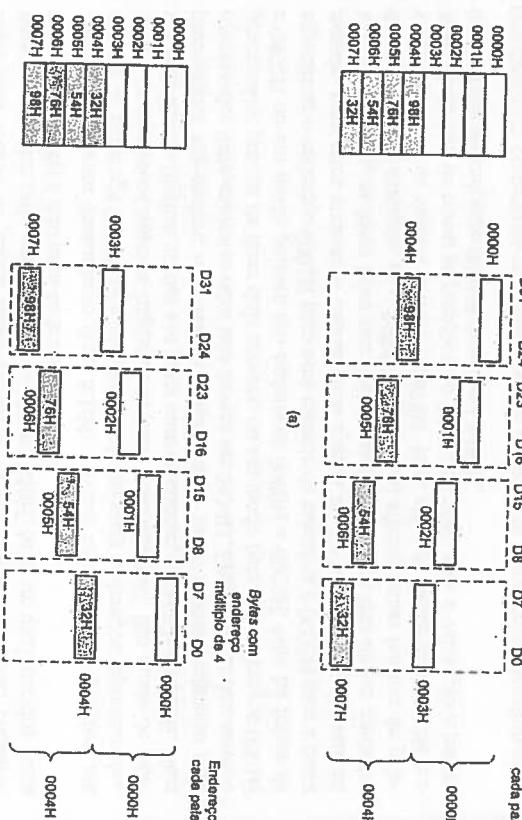


Fig. 6.15 - Disposição em memória da palavra de 32 bits 98 76 54 32H com um processador de 32 bits. (a) – *Big-endian*; (b) – *Little-endian*

Os termos *big-endian* e *little-endian* referem-se à extremidade de uma palavra que aparece primeiro numa escala crescente de endereços. Se essa extremidade for o byte de maior peso, como na Fig. 6.15a, é *big-endian*. Se for o byte de menor peso, como na Fig. 6.15b, é *little-endian*.

A grande questão é: qual é o melhor método? Não há resposta clara. Ambos funcionam e são alternativas válidas.

O método *big-endian* tem a vantagem de que num registo os bytes aparecem pela mesma ordem (lendo o registo da esquerda para a direita) que na memória, lida de cima para baixo (Fig. 6.15a) ou da esquerda para a direita (usual nos mapas de visualização dos conteúdos de memória).

O método *little-endian*, por seu lado, tem a vantagem de ambas as instruções

```
MOV [R1], R2
MOVE [R1], R2
```

colocarem no endereço indicado por R1 o byte de menor peso de R2. Com o método *big-endian*, o MOV coloca nesse endereço o byte de maior peso de R2, ao contrário do MOVE, que coloca lá o byte de menor peso de R2. Idêntico raciocínio se pode estabelecer para o caso da leitura da memória.

Os processadores modernos suportam normalmente os dois métodos, sob seleção de um pino. Os compiladores precisam de saber qual o método usado pelo processador, para gerarem os bytes do código-máquina pela ordem correcta.

O PEPE suporta apenas o método *big-endian*, não porque seja complicado de implementar este pino de seleção em hardware mas sim para evitar a complexidade resultante de poder configurar de forma separada o hardware e o software, com a possibilidade de erros derivados da utilização de configurações incompatíveis.

**NOTA** A palavra *endian* vem de *end*, ou topo. Esta questão foi satirizada no livro "As viagens de Gulliver", de Jonathan Swift, em que se retratam longas discussões sobre a forma de começar a partir a casca aos ovos cozidos, algo tipicamente inglês e que normalmente é feito colocando o ovo cozido com casca num cálice. Um dos tops do ovo é mais afilado (*little end*) e o outro mais estérico (*big end*). Uns defendiam que o ovo devia ser colocado no cálice com o topo mais afilado para cima e os outros ao contrário. Discussão inútil, pois qualquer lado serve e o que se pretende é comer o ovo, o que se consegue seja qual for o lado por que se comece a partir a casca.

Da mesma forma, discutir se os computadores devem ser *little-endian* ou *big-endian* é em larga medida uma inutilidade, pois o computador funciona com qualquer delas (desde que escolha uma), e ambas têm vantagens e desvantagens.

### 6.15.3 ALINHAMENTO DOS ACESSOS

No acesso à memória em modo de byte (com MOVE), é possível especificar qualquer endereço, par ou ímpar. Dependendo do método de ligação à memória (*big-endian* ou *little-endian*) e da paridade do endereço acedido (par ou ímpar), o processador usa uma das metades do barramento de dados para aceder à metade da memória onde esse endereço se encontra fisicamente (Fig. 6.14).

A semântica do acesso à memória em modo de palavra (com MOV) especificando um endereço X implica que os bytes acedidos sejam os contidos nos endereços X e X+1. Neste caso, há duas situações possíveis:

- O endereço X especificado é par, bastando efectuar um acesso de 16 bits com os sinaliz BX=0 e A0=0. O valor de 16 bits é transferido tal como está de um registo para a memória (ou vice-versa). Este acesso designa-se alinhado;
- O endereço especificado é ímpar, caso em que não se consegue aceder aos dois bytes desta palavra num só acesso. Tal como se pode ver na Fig. 6.9b, as duas metades da memória recebem os mesmos bits de endereço, e só se conseguem ler apenas no A0 (isto é, valores iguais dos bits de endereço nas duas metades da memória), o que implica X ser par. Se X for ímpar, X e X+1 diferem não apenas no

A0 mas também no A1. A palavra (16 bits) a que se pretende aceder encontra-se distribuída por duas palavras alinhadas (um byte em cada uma). Este acesso designa-se desalinhado.

**SOLUÇÃO** Efectuar dois acessos e somar o valor final aproveitando apenas os bytes relevantes e trocando a sua ordem.

A Fig. 6.16 ilustra o alinhamento dos acessos, em que se podem exemplificar os seguintes acessos:

- Acesso de byte – Acesso individual a qualquer dos bytes, em endereço par (Fig. 6.16a) ou ímpar (Fig. 6.16b). Qualquer processador com endereçamento de byte suporta isto, não interessando se é big-endian ou little-endian. Por exemplo, uma leitura do endereço 0001H dará o valor 34H, tal como uma leitura do endereço 0002H dará o valor 56H.

■ Acesso de palavra alinhado (Fig. 6.16c) – Neste exemplo há dois possíveis, especificando o endereço 0000H ou o 0002H. Num processador big-endian os valores lidos num acesso de leitura serão 1234H e 5678H, respectivamente, enquanto num processador little-endian serão 3412H e 7856H. Mas em qualquer dos casos o valor de 16 bits é lido num só acesso;

■ Acesso de palavra desalinhado (Fig. 6.16d) – Neste exemplo só é possível um acesso, ao endereço 0001H (apenas porque não há mais endereços representados. Todos os endereços ímpares implicam acessos desalinhados). Em modo de palavra, este acesso implica aceder aos bytes com os endereços 0001H e 0002H. Num processador big-endian, uma leitura deverá dar o valor 3456H, enquanto num processador little-endian o valor lido deverá ser 5634H. Em qualquer dos casos, o processador precisa de fazer dois acessos, ou em modo byte aos endereços 0001H e 0002H ou em modo palavra aos endereços 0000H e 0002H. Não há forma de especificar o endereço 0001H para uma metade da memória e 0002H para a outra metade, pois estes endereços não diferem apenas no bit A0 (também diferem no A1). Além disso, os bytes lidos nos dois acessos de palavra têm de ser mudados de posição. Por exemplo, se o PEPE (big-endian) efectuar duas leituras de palavra, ficará com os valores 1234H e 5678H. Para além de deitar fora o 12H e o 78H, ainda tem de trocar as posições do 34H e do 56H, pois o valor que a leitura desalinhada ao endereço 0001H deve dar é 3456H.

Muitos processadores comerciais suportam acessos em palavra a endereços desalinhados, para serem mais flexíveis, mas a interface de memória (que tem de efectuar os dois acessos e as trocas dos bytes automaticamente) fica mais complexa, razão pela qual outros processadores suportam apenas acessos alinhados. O suporte para o desalinhamento é algo não fundamental, de que se consegue prescindir com alguns cuidados ao nível do compilador ou do programador em linguagem assembly.

Por simplicidade, o PEPE não suporta acessos desalinhados. As consequências em termos práticos são as seguintes:

- O compilador e o assembler só podem gerar instruções em endereços pares, pois a busca de instruções é sempre feita palavra a palavra (todas as instruções ocupam uma palavra);

■ As directivas WORD e TABLE, que reservam espaço para palavras em memória, só podem usar endereços alinhados (a directiva STRING só lida com bytes e não precisa de endereços alinhados);

■ A instrução MOV só consegue aceder à memória (em palavra) se o endereço a aceder for par. Note-se que instruções do género MOV [R1+R2], R3 não precisam em rigor que R1 e R2 sejam par, mas apenas que a soma deles o seja. No entanto, em abono da clareza, é melhor que ambos sejam pares. A instrução MOV suporta endereços ímpares e acesso em byte, pelo que não tem nada a ver com o problema do alinhamento.

Ao gerar o código, o compilador de uma linguagem de alto nível ou o assembler vão usando sequencialmente os endereços. Se uma dada construção (a directiva STRING) deixar como último byte ocupado um endereço par e a seguir for necessário um endereço ímpar (para colocar uma instrução, por exemplo), avança-se para o próximo endereço. Desperdiça-se um byte de memória (o que tem endereço ímpar, que era o próximo byte) mas resolve-se o problema do alinhamento.

**NOTA** O conceito de alinhamento estende-se facilmente a processadores de maior largura. Por exemplo, um acesso alinhado num processador de 32 bits exige que o endereço seja múltiplo de 4 (porque cada palavra tem 4 bytes) e não apenas par.

|     |       |     |     |       |
|-----|-------|-----|-----|-------|
| (a) | 0000H | 12H | 34H | 0001H |
|     | 0002H | 56H | 78H | 0003H |
| (b) | 0000H | 12H | 34H | 0001H |
|     | 0002H | 56H | 78H | 0003H |
| (c) | 0000H | 12H | 34H | 0001H |
|     | 0002H | 56H | 78H | 0003H |
| (d) | 0000H | 12H | 34H | 0001H |
|     | 0002H | 56H | 78H | 0003H |

Fig. 6.16 - Alinhamento dos acessos à memória. (a) e (b) – Acessos individuais a um byte, par ou ímpar, sem problemas de alinhamento; (c) – Acesso a uma palavra alinhada; (d) – O acesso a um endereço ímpar (desalinhado) obriga a efectuar dois acessos para aceder correctamente à palavra intala

Os processadores de palavra de 32 bits suportam normalmente também acessos em modo de meia palavra (16 bits), com instruções específicas para o efeito. Nesse caso, faz sentido falar em alinhamento de meia palavra, que exige apenas que o endereço a aceder seja par (tal como no PEPE).

Ainda falta esclarecer o que é que sucede, por exemplo, se o PEPE executar a instrução

`MOV R1, [R2]`

em que o valor de R2 é ímpar. Trata-se de um erro, de que o hardware do processador não consegue recuperar. A solução, descrita com mais detalhe na secção 6.2.3, consiste em invocar uma rotina para tratar da situação. Cabe ao programador especificar o que essa rotina deverá fazer para transpor o problema (terminar o programa, avisar o utilizador, eventualmente efectuar alguma correção, etc.).

### SIMULADOR – ENDEREÇAMENTO DE BYTE

Esta simulação ilustra as implicações do endereçamento de byte, tornando o circuito da Fig. 6.13 como base. Os aspectos cobertos incluem os seguintes:

- Periféricos de 8 e 16 bits e respectivos endereços;
- Ligação do processador aos dispositivos (*big-endian* e *little-endian*);
- Alinhamento dos acessos.

## 6.1.6 CICLOS DE ACESSO À MEMÓRIA/PERIFÉRICOS

### 6.1.6.1 LIGAÇÃO AO BARRAMENTO DE DADOS

O barramento de dados é bidireccional, isto é, poderá ser lido ou escrito por cada um dos dispositivos que a ele ligam (incluindo processador). Isto implica que cada dispositivo deve poder ligar ao barramento de dados tanto com uma entrada como com uma saída. Várias saídas ligadas entre si podem originar conflitos, se pelo menos duas saídas tentarem impor valores diferentes no barramento.

**PROBLEMA** Como é que se implementa a bidireccionalidade, ao mesmo tempo que se evitam os conflitos?

**SOLUÇÃO** Cada dispositivo que ligue ao barramento de dados com uma saída é obrigado a usar uma saída *tristate* (com 3 estados: 0, 1 e desligado), sob controlo de um sinal que permita ligar e desligar essa saída (secção 2.6.3, na página 65). Cada dispositivo deve ser comandado pelo processador e pelo circuito de descodificação de endereços de forma a garantir que:

- Cada dispositivo (excepto o processador) leia ou escreva o barramento de dados, de acordo com o indicado pelo processador. Quando ler, não escrever (desligando a sua saída *tristate*). Quando escrever, não ler;

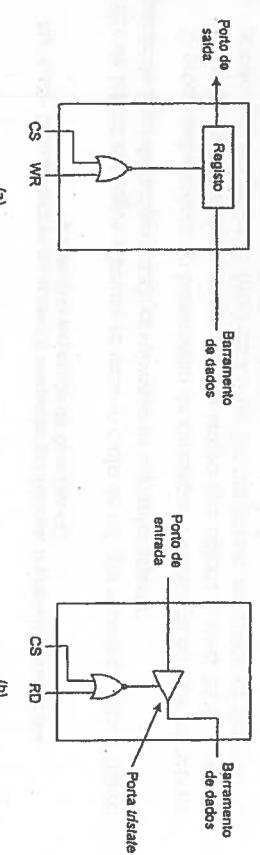


Fig. 6.17 - Ligação de um dispositivo a um barramento de dados e respectivo controlo, com recurso a uma interface *tristate* caso o dispositivo possa ser lido. (a) – Periférico só de saída; (b) – Periférico só de entrada; (c) – Periférico de entrada/saída; (d) – Memória

Cada um dos dispositivos (incluindo o processador) da Fig. 6.13 possui um circuito de controlo deste género, com os seguintes sinais:

- RD – Sinal de leitura, que indica (se activo) que o processador está a ler do barramento de dados;
- WR – Sinal de escrita, que indica (se activo) que o processador está a escrever no barramento de dados;
- CS – Sinal de selecção, que indica (se activo) que a operação em curso (leitura ou escrita) está a ser efectuada sobre este dispositivo.

Alguns dispositivos poderão não ter o sinal de escrita ou de leitura, se não suportarem a operação correspondente, mas todos têm um sinal de seleção individual, uma vez que os sinais RD e WR ligam a todos os dispositivos. Naturalmente, é responsabilidade do sistema de descodificação de endereços garantir que nunca há mais do que um dispositivo com o seu sinal de seleção activo. Numa operação de leitura por parte do processador, isto garante que só um dispositivo de cada vez poderá ter a sua interface-tristate ligada, de forma a evitar conflitos no barramento de dados.

A Fig. 6.17 representa genericamente um dispositivo de  $N$  bits. Se considerarmos endereçamento de byte, isto é, a divisão em duas metades de 8 bits dos dispositivos de 16 bits, como sucede na Fig. 6.13, então a Fig. 6.17 representa um dispositivo de 8 bits, que poderá ser metade de um de 16 bits (como a RAM ou o P1) ou o dispositivo completo (como o P3 ou o P4). Cada dispositivo (ou parte dele) endereçável individualmente em leitura tem de ter controlo tristate individual.

Note-se que a Fig. 6.17b representa um periférico só de entrada e como tal não tem memória. Limita-se a incorporar uma porta tristate, que quando está activa (numa operação de leitura) liga directamente os pinos exteriores ao barramento de dados, de modo a que estes possam ser memorizados pelo processador durante essa operação.

A secção 2.6.3, na página 65, e a [MEMÓRIA](#), na página 66, ilustram o funcionamento das portas tristate.

### 6.1.6.2 CICLOS DE LEITURA E ESCRITA

É sempre o processador que toma a iniciativa de efectuar uma operação de leitura ou de escrita no barramento de dados. No entanto, o processador não se pode "deslocar" ao interior de um dispositivo para lá manipular directamente os dados. O máximo que pode fazer é actuar os sinais RD ou WR e o barramento de endereços (que fará actuar o sinal de seleção CS do dispositivo acedido) e esperar que esse dispositivo reaja de forma adequada para implementar essa operação.

**PROBLEMA 6.6** Como é que o processador e os dispositivos acedidos conseguem coordenar-se para trocar informação? Numa operação de escrita, como é que o dispositivo sabe quando é que o valor escrito pelo processador está disponível para o memorizar? Em leitura, como é que o processador sabe quando é que o dispositivo já disponibilizou o valor a ler, para o poder memorizar? E como é que ambos sabem que a operação terminou?

**SOLUÇÃO** Definir protocolos que envolvam os vários barramentos do processador e estableça em detalhe a sequência de evolução dos sinais desses barramentos de forma a implementar as operações de leitura e escrita. Tanto o processador como os restantes dispositivos têm de obedecer a esse protocolo.

Estes protocolos designam-se **ciclo de leitura** e **ciclo de escrita** e envolvem uma sequência de operações elementares. O termo "ciclo" advém do facto de que cada acesso à memória implica repetir esta sequência.

A Fig. 6.18 ilustra o ciclo de leitura do PEPE, que se completa num só ciclo do sinal de relógio (CLK) do processador (o eixo horizontal é o tempo). Outros processadores terão provavelmente diferenças de pormenor, mas o princípio básico é o mesmo. Esta figura representa na realidade dois ciclos de acesso à memória, nos ciclos de relógio T1 e T3, assumindo-se neste exemplo que no período T2 o processador não acedeu à memória (por estar ocupado internamente a executar uma instrução, por exemplo). Esta situação mostra que o processador não acede à memória em todos os ciclos de relógio.

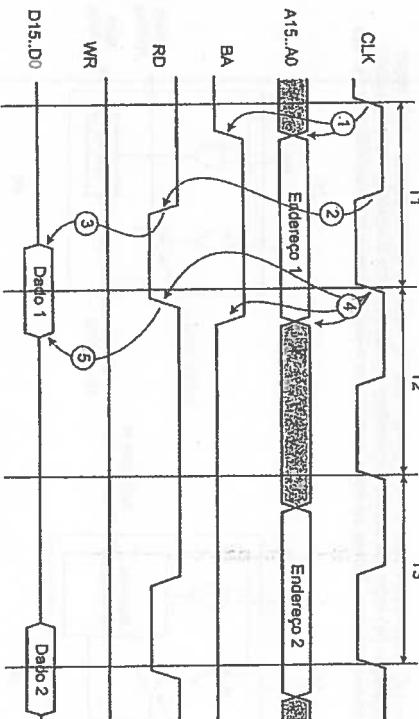


Fig. 6.18 - Ciclo de leitura no acesso à memória/periféricos. O primeiro ciclo acece

à memória em modo de byte (BA=1)

Um ciclo de leitura consegue efectuar-se num só ciclo de relógio e começa com o flanco ascendente do sinal de relógio (CLK), envolvendo os seguintes passos:

- O processador coloca no barramento de endereços o endereço do byte ou palavra a aceder, aproximadamente ao mesmo tempo que coloca o sinal BA a 1 (caso o acesso seja em modo de byte) ou a 0 (se for um acesso em modo de palavra). O primeiro ciclo de leitura da Fig. 6.18 é em modo de byte;
- No flanco descendente do relógio, o processador activa o sinal de leitura (RD=0);
- O dispositivo acedido (aquele cujo sinal de seleção tiver sido activado, em função dos bits de endereço) detecta o sinal RD activo e coloca o valor a ser lido no barramento de dados;
- No fim do ciclo de relógio (flanco ascendente), o processador desactiva o sinal RD, ao mesmo tempo que memoriza o valor presente no barramento de dados.<sup>70</sup>

<sup>70</sup> O primeiro acesso, em modo de byte, o processador lê apenas metade do barramento de dados. Como o PEPE é um processador big-endian, usa a metade alta (D15..D8) se o endereço for par e a metade baixa (D7..D0) se o endereço for ímpar (secção 6.1.5.2).

Também a partir desta altura o barramento de endereços e o sinal BA podem mudar (não estando nem o RD nem o WR activos, nenhum dispositivo será acedido);

5. O dispositivo acedido detecta a desactivação do sinal RD e retira o seu valor do barramento de dados, deixando-o livre para os próximos ciclos de acesso à memória.

O processador pode efectuar ciclos de leitura da memória em ciclos de relógio contíguos, embora (tal como ilustrado na Fig. 6.18) haja alguns ciclos em que o processador está ocupado internamente e não tem necessidade de efectuar ciclos de acesso à memória.

Note-se que o acesso à memória em leitura é feito não apenas na execução de algumas instruções (MOV, MOVA, SWAP, POP, etc.) mas também durante a busca de todas as instruções, que têm de ser lidas da memória para serem executadas.

O ciclo de escrita é exemplificado pela Fig. 6.19, com dois ciclos consecutivos (o primeiro é feito em modo de byte, com BA=1). Tal como o ciclo de leitura, o de escrita inicia-se com o flanco ascendente do sinal de relógio (CLK) e envolve os seguintes passos:

1. O processador coloca no barramento de endereços o endereço do byte ou palavra a acceder, aproximadamente ao mesmo tempo que coloca o sinal BA a 1 (caso o acesso seja em modo de byte) ou a 0 (se for um acesso em modo de palavra). O primeiro ciclo de escrita da Fig. 6.19 é em modo de byte;
2. No flanco descendente do relógio, o processador activa o sinal de escrita (WR=0) e coloca no barramento de dados o valor a escrever;<sup>71</sup>
3. O processador desactiva o sinal WR, altura em que o dispositivo acedido deve memorizar o valor presente no barramento de dados;
4. No fim do ciclo, o processador retira o seu valor do barramento de dados, deixando-o livre para os próximos ciclos de acesso à memória. Também a partir desta altura o barramento de endereços e o sinal BA podem mudar (não estando nem o RD nem o WR activos, nenhum dispositivo será acedido).

A parte a sequência de sinais e o sentido de circulação dos dados, uma diferença fundamental entre os ciclos de leitura e escrita reside no facto de este último demorar dois ciclos de relógio, enquanto o primeiro demora um. A justificação para este facto será dada na secção 6.16.3.

Diz-se que o barramento de dados está em estado de alta impedância<sup>72</sup> quando nem o processador nem qualquer outro dispositivo está a colocar dados no barramento de dados.

<sup>71</sup> No primeiro acesso, em modo de byte, o processador escreve apenas em metade do barramento de dados. O PEPE é um processador big-endian e como tal usa a metade alta (D15..D8) se o endereço for par e a metade baixa (D7..D0) se o endereço for ímpar (secção 6.1.5.2).

<sup>72</sup> Indicando com isto que nenhum dos dispositivos (incluindo o processador) está a forçar um valor no barramento de dados.

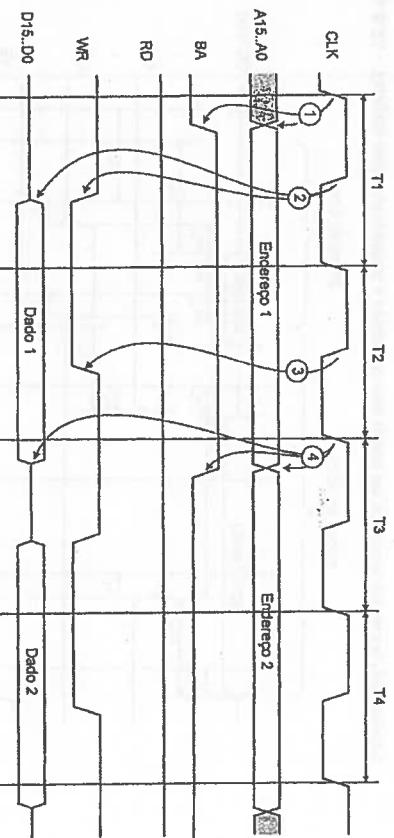


Fig. 6.19 - Ciclos de escrita no acesso à memória/periféricos. O primeiro ciclo acede à memória em modo de byte (BA=1)

Os ciclos de leitura e/ou escrita podem ser contíguos, sem necessidade de ciclos de relógio de intervalo. Note-se que em cada ciclo há sempre períodos em que o barramento está em estado de alta impedância, o que permite a comutação de leitura para escrita (inversão do sentido de circulação no barramento de dados) quando um ciclo de escrita sucede imediatamente a um de leitura, tal como alias é exemplificado pela Fig. 6.20, em que a leitura é em modo de palavra (BA=0) e a escrita em modo de byte (BA=1).

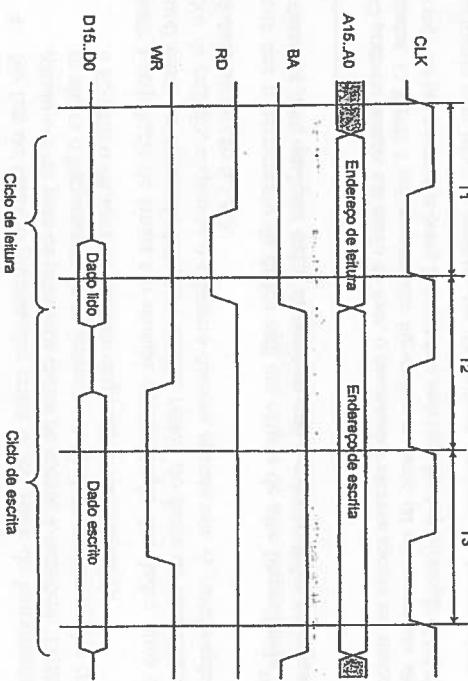
### SIMULAÇÃO 6.3 – CICLOS DE ACESSO À MEMÓRIA E PERIFÉRICOS

Esta simulação exemplifica o funcionamento do processador nos ciclos de acesso à memória. Os aspectos cobertos incluem os seguintes:

- Evolução detalhada dos sinais durante um ciclo de leitura;
- Evolução detalhada dos sinais durante um ciclo de escrita;
- Verificação da evolução dos sinais nos vários barramentos durante o processamento completo de uma instrução swap (troca de um valor de um registo com o de uma célula de memória), incluindo leitura da própria instrução da memória.

### 6.1.6.3 TEMPORIZAÇÕES NO ACESSO AOS DISPOSITIVOS

Nada neste mundo é instantâneo. Para que os ciclos de leitura e escrita funcionem correctamente, não basta combinar a sequência dos vários sinais. A evolução temporal destes sinais é também de crucial importância.



**Fig. 6.20 - Os ciclos de acesso à memória/periféricos em leitura e escrita**

**PROBLEMA 9** Numa operação de escrita, como é que o processador sabe que o dispositivo destino já teve tempo de memorizar o valor, para poder passar à operação seguinte? Em leitura, como é que o processador sabe quando é que o valor fornecido pelo dispositivo acedido já está disponível, para o poder memorizar? E como é que este dispositivo sabe que o processador já leu esse valor, para dar a operação por terminada e ficar à espera de novos pedidos do processador?

**SOLUÇÃO:** Definir tempos mínimos de duração e de sequenciamento que o processador e os restantes dispositivos têm de cumprir para os vários sinais envolvidos no acesso, de modo a garantir que todos têm tempo de reagir em cada um dos ciclos de acesso à memória.

Um dos aspectos mais notórios na Fig. 6.20 é o facto de o ciclo de escrita durar dois ciclos de relógio, enquanto o ciclo de leitura demora apenas um ciclo de relógio. Isto é necessário, essencialmente devido às seguintes razões:

Numa memória não se pode activar o sinal de wr imediatamente após disponibilizar o barramento de endereços. Tem de se esperar algum tempo até que os circuitos internos de selecção da memória estabilizem para se ter a certeza que só a célula correcta é seleccionada. Caso contrário, corre-se o risco de destruir o conteúdo de células de memória que fossem seleccionadas transitoriamente enquanto os bits de endereço estivessem a mudar (porque não mudam todos exactamente ao mesmo tempo);

Também para garantir a estabilização interna dos sinais e uma escrita correcta, as memórias requerem normalmente que os dados permaneçam estáveis no buffer.

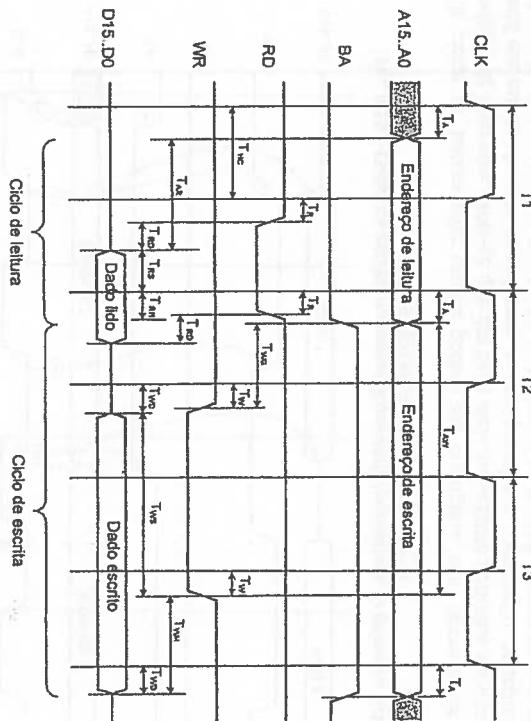


Fig. 6.21 - Tempos mais relevantes a respeitar nos ciclos de acesso à memória/periféricos

mento de dados ainda durante algum tempo após o sinal de escrita (WR) ser desactivado. Por isso, o processador não deve desactivar o sinal WR e os dados no mesmo flanco de relógio (precisa de pelo menos mais metade do ciclo de relógio);

Nas memórias, o tempo de acesso<sup>73</sup> em escrita é normalmente superior ao tempo de acesso em leitura;

O ciclo de leitura tem de ter o sinal RD actuado apenas na segunda metade do relógio. Se o fosse apenas na primeira metade, o tempo de acesso à memória ficaria muito curto e seria necessário mais um ciclo de relógio. Isto faz com que a memória continue a forçar um valor no barramento de dados ainda durante algum tempo do ciclo de relógio seguinte. Portanto, um acesso em escrita imediatamente a seguir ao de leitura não pode começar a actuar o sinal de WR e a colocar os dados no barramento de dados logo no início do ciclo de escrita, pois haveria um potencial conflito no barramento de dados entre a memória e o processador;

O barramento de dados do processador tem normalmente os circuitos internos de acesso aos pinos exteriores em modo de leitura, e numa escrita tem de activar os circuitos no sentido contrário, mudança que demora algum tempo.

A Fig. 6.21 representa os principais tempos envolvidos nestes acessos. Cada processador terá as suas especificidades, altamente dependentes da forma como está implementado. No entanto, os tempos aqui indicados para o PEPE são representativos do que é normal existir em todos os processadores.

A Tabela 6.10 explica o significado de cada tempo e as restrições ao período do relógio que não pode ser menor do que o valor que cumpre todas essas restrições. A frequência do sinal de relógio de um processador, que se pretende que seja a maior possível, depende assim não apenas dos detalhes internos da sua arquitetura (assunto do capítulo 7) mas também da sua interacção com a memória e os periféricos.

| TEMPO                   | DESCRIÇÃO                                   | IMPOSTO POR:          | GAMA   | RESTRIÇÕES                                            |
|-------------------------|---------------------------------------------|-----------------------|--------|-------------------------------------------------------|
| $T_1, T_2, T_3$         | Período do sinal de relógio                 | Gerador de relógio    | exacto |                                                       |
| $T_{HC}$                | Metade do período do sinal de relógio       | Gerador de relógio    | exacto |                                                       |
| $T_A$                   | Atraso do endereço                          | Processador           | máx.   |                                                       |
| $T_M$                   | Tempo de acesso da memória em leitura       | Memória ou periférico | máx.   | $T_1 \geq T_A + T_M + T_{RS}$                         |
| $T_R$                   | Atraso de RD                                | Processador           | máx.   | $T_{HC} \geq T_A + T_R + T_{RS}$                      |
| $T_D$                   | Atraso de dados após flanco                 | Memória ou periférico | máx.   | $T_{HC} \geq T_A + T_M - (T_R + T_D)$                 |
| <b>CICLO DE LEITURA</b> |                                             |                       |        |                                                       |
| $T_{RS}$                | Tempo com dados estáveis                    | Processador           | mín.   | $T_A \geq T_{RS}$                                     |
| $T_{RD}$                | Tempo com dados estáveis antes de RD        | Processador           | mín.   |                                                       |
| $T_{WR}$                | Tempo de acesso da memória em escrita       | Memória ou periférico | máx.   |                                                       |
| <b>CICLO DE ESCRITA</b> |                                             |                       |        |                                                       |
| $T_{WG}$                | Tempo de guarda entre endereço estável e WR | Memória ou periférico | mín.   | $T_2 + T_3 \geq T_A + T_{WG} + T_{WH} - T_B$          |
| $T_W$                   | Atraso de WR                                | Processador           | máx.   | $T_2 + T_3 \geq T_A + T_{WG} - T_W + T_{WS} + T_{WH}$ |
| $T_B$                   | Atraso do barramento de dados em escrita    | Processador           | máx.   | $T_{HC} \geq T_A + T_{WG} - T_W$                      |
| $T_{WS}$                | Tempo com endereço estável antes de WR      | Memória ou periférico | mín.   | $T_{HC} \geq T_W + T_{WH} - T_B$                      |
| $T_{WH}$                | Tempo com dados estáveis após flanco de WR  | Memória ou periférico | mín.   |                                                       |

Tabela 6.10 - Explicação dos tempos mais relevantes nos ciclos de acesso à memória/periféricos e restrições a cumprir.

O processador não espera. Cumple as temporizações expressas na Fig. 6.21, tendo como base o período do sinal de relógio. É responsabilidade do projectista do sistema verificar que os tempos mínimos exigidos para o acesso à memória e aos periféricos são respeitados pelo processador, nem que para isso tenha de diminuir a frequência do relógio (ou usar dispositivos com acesso mais rápido). Se isto não for garantido, os acessos não

funcionarão bem, as leituras darão valores errados, os valores escritos pelo processador não serão correctamente memorizados pelos dispositivos ou serão escritos na célula errada, etc. O sistema ficará simplesmente não usável.

Todos os sinais mudam com algum atraso em relação ao sinal que provocou a sua mudança, razão pela qual os sinais mudam de forma não alinhada com as mudanças de flanco do relógio. O tempo de atraso é função da complexidade (quantidade de portas lógicas) dos circuitos internos (do processador ou de qualquer outro dispositivo) e resulta do tempo que um sinal leva a propagar-se pelas várias portas lógicas por que tem de passar. Tem de ser medido e fornecido pelo fabricante. Estes tempos de atraso são normalmente indicados com dois valores, típico e máximo (que o fabricante garante não exceder). O sistema deve ser projectado para o valor máximo (caso mais desfavorável), mas o valor típico dá uma indicação da folga que se terá na maior parte dos casos. Com a tecnologia actual, estes tempos medem-se em nanosegundos ou mesmo em décimas de nanosegundo. Por exemplo, só algum tempo após o flanco ascendente do ciclo de relógio inicial de um ciclo é que o novo valor do barramento de endereços fica estável (tempo de atraso  $T_A$ ). O mesmo acontece no flanco descendente do sinal de relógio no ciclo de leitura, em que RD fica activo apenas  $T_A$  unidades de tempo depois.

Por outro lado, os dispositivos também demoram algum tempo a reagir aos sinais de controlo que recebem do processador (barramentos, de endereços e de dados, e RD e WR). Por exemplo, uma memória demorará  $T_{RS}$  unidades de tempo a colocar no barramento de dados o valor endereçado pelo barramento de endereços.

Para além dos tempos de atraso à reacção de um sinal, há que ter em conta que a memorização de um dado valor (presente no barramento de dados, tipicamente) tem outros dois tipos de restrições em relação à activação do sinal de memorização (tipicamente, o RD e o WR):

- Tempo de preparação** – O sinal a memorizar tem de estar estável (bits fixos, sem estarem a variar) algum tempo antes de o sinal que controla a memorização variar). Isto resulta do facto de que o circuito de memorização é geralmente mais complexo que o circuito do sinal que controla essa memorização. Pelo que é preciso disponibilizar o dado a memorizar algum tempo antes de o sinal de controlo mudar. Por exemplo, os tempos  $T_{RS}$  e  $T_{WS}$  especificam este aspecto em relação aos ciclos de leitura (sinal RD) e escrita (sinal WR), respectivamente. O  $T_{RS}$  é exigido pelo processador, enquanto  $T_{WS}$  é imposto pela memória.
- Tempo de manutenção** – O sinal a memorizar não pode ser retificado imediatamente após o flanco do sinal que disparou a memorização, pois esta operação não é instantânea e corre-se o risco de o circuito memorizar alguns dos bits entretanto já alterados. Os tempos  $T_{RD}$  e  $T_{WH}$  expressam este aspecto. O  $T_{RD}$  é exigido pelo processador no ciclo de leitura, enquanto  $T_{WH}$  é imposto pela memória durante o ciclo de escrita. Note-se que no caso do  $T_{RD}$  a referência é o flanco do relógio e não o RD, pois o sinal que actua o registo dentro do processador é o relógio. O RD serve apenas para avisar a memória de que o ciclo de leitura já terminou.

Estes tempos especificam-se pelo valor mínimo e típico. A frequência do relógio do processador não pode ultrapassar o limite acima do qual estes valores mínimos já não podem ser garantidos, sob pena de o sistema não conseguir efectuar correctamente todos os acessos.

Da Fig. 6.21 ainda são aparentes dois aspectos relevantes da transição entre o ciclo de leitura e o de escrita:

- O sinal RD é desactivado mais ou menos ao mesmo tempo que o barramento de dados passa a reflectir o novo valor para o acesso de escrita, pelo que pode suceder que o valor lido pelo processador ainda esteja no barramento de dados algum tempo depois de o barramento de endereços estar já a exhibir um novo valor. Tudo isto sucede já no período de relógio T2. Por outras palavras, com os dois ciclos contíguos o ciclo de leitura "invade" o ciclo de escrita. Isto não constitui problema porque já no período T2 o ciclo de escrita só começa efectivamente quando o barramento de endereços muda, e mesmo assim todo o resto do circuito ( nomeadamente a parte de descodificação de endereços) O sinal de WR só é activado na segunda metade do período de relógio T2;

Para garantir o tempo  $T_{\text{WR}}$ , o melhor é desenhar o processador de forma que  $T_{\text{WR}}$  não seja maior que  $T_1$  (apesar de haver alguma folga, pois os circuitos de descodificação de endereços também têm o seu atraso). O perigo aqui seria mudar os endereços ainda durante o tempo de manutenção do valor lido, o que poderia mudar o sinal de selecção de dispositivo e invalidar antes de tempo o valor que deveria estar estável no barramento de dados.

#### 6.1.6.4 ACESSO A DISPOSITIVOS LENTOS

**PERGUNTA** Se um dispositivo for significativamente mais lento do que os restantes, as temporizações têm de ser dimensionadas para o seu caso (o mais desfavorável), atrasando todos os acessos?

**SOLUÇÃO** Para suportar dispositivos muito lentos face aos restantes, e para evitar dimensionar as temporizações pelo dispositivo mais lento deve-se prever um sinal de pedido de espera ao processador (WAIT) que o dispositivo lento tem de accionar.

Este sinal deve ser accionado por um dispositivo lento (que saiba à partida que não consegue acompanhar a rapidez dos ciclos de leitura ou escrita do processador) mal seja activado o seu sinal de selecção e durar o tempo necessário para se completar o acesso ao dispositivo. Este sinal deve ser ligado ao pino WAIT do PEPE, que é activo a 1 e constitui o quarto sinal do barramento de controlo (depois do RD, WR e BA). Ao contrário dos anteriores, que são produzidos pelo processador, este sinal de WAIT é produzido pelos dispositivos (e lido pelo processador).

A activação do pino WAIT prolonga o sinal de RD ou WR por um número inteiro de ciclos de relógio (o prolongamento termina quando o relógio tiver um flanco ascendente em que WAIT já não esteja a 1).

A Fig. 6.22 mostra um ciclo de leitura normal e outro em que o dispositivo actuou o pino de WAIT durante algum tempo, resultando no prolongamento do acesso por mais um ciclo de relógio. O tempo de acesso neste caso ( $T_{\text{acc}}$ ) é nitidamente superior ao de um dispositivo que consiga acompanhar as temporizações mínimas do processador ( $T_{\text{acc}}$ ). O prolongamento funciona de modo equivalente num ciclo de escrita. Os ciclos de relógio a mais resultantes do prolongamento designam-se estados de espera (wait states).

Se houver vários dispositivos lentos, cada um deles deve gerar o seu próprio sinal de pedido de espera, de acordo com as suas próprias restrições. O pino WAIT deve ligar a um AND de todos estes sinais. O circuito de descodificação de endereços garante que só um dispositivo em cada ciclo de acesso é seleccionado, pelo que só um dispositivo de cada vez poderá accionar o seu pedido de espera. Se todos os dispositivos do sistema forem suficientemente rápidos, o sinal WAIT pode ser deixado sempre inactive (ligado em permanência a 0).

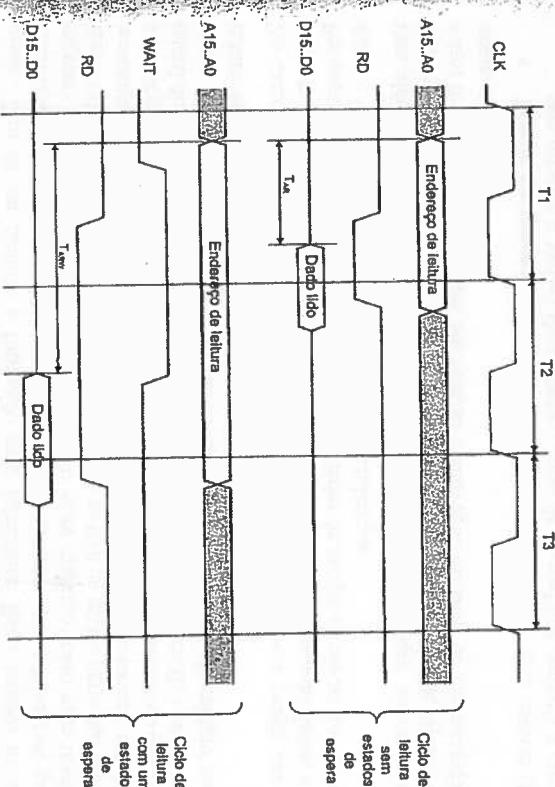


Fig. 6.22 - Prolongamento do tempo de acesso num ciclo de leitura quando o pino WAIT do processador é activado.

Deve-se que este mecanismo bloqueia completamente o processador durante os ciclos de relógio de prolongamento, pelo que só deve ser usado para dispositivos cujo tempo de acesso esteja no máximo na ordem de poucos ciclos de relógio. Dispositivos mesmo muito lentos face à velocidade do processador devem ser accedidos através de portos de periféricos (cujos bits substituirão os barramentos do processador e em que o processador tem de fazer vários acessos ao periférico para gerar a sequência de todos os sinais relevantes, em vez de uma simples instrução mov).

**NOTA**

Alguns computadores possuem um circuito especial (gerador de estados de espera) que insere sistematicamente um ou mais estados de espera em todos os acessos, quando as memórias são mais lentas que o processador. Nos PCs, o número de estados de espera é inclusivamente programável por software, memorizando esse número num periférico que depois liga ao gerador de estados de espera. Uma solução mais simples seria diminuir a frequência de relógio do processador, mas assim consegue-se que o processador funcione à velocidade máxima no acesso às suas memórias internas (*caches*) – ver secção 7.5, na página 622) e só teria de esperar nos acessos à memória exterior (bastante menos frequentes que os acessos às *caches*).

**ESSENCIAL**

- O processador liga aos dispositivos (memórias e periféricos) por meio de barramentos, endereços, dados e controlo. O processador liga as memórias e periféricos da mesma forma, não os distinguindo nos acessos que faz.
- É sempre o processador que comanda os acessos, especificando um endereço e um tipo de acesso (leitura ou escrita). É o software que tem de saber se, num dado endereço há algum dispositivo e se suporta o acesso efectuado.
- Cada dispositivo tem um sinal de selecção que o activa. Tem de haver um circuito (descodificador de endereços) que liga ao barramento de endereços e que em cada acesso activa apenas um sinal de selecção de dispositivo. De acordo com o mapa de endereços estabelecido para o sistema.
- O descodificador de endereços usa tipicamente os bits de menor peso do barramento de endereços. Os bits de menor peso ligam aos vários dispositivos para endereçar células individuais dentro deles.
- O PEPE suporta endereçamento de *byte*, em que os endereços identificam bytes individuais e não palavras. Há instruções diferentes para aceder aos dispositivos em bytes e em palavras (neste caso têm de ser acessos alinhados – endereço par).
- Com endereçamento de *byte*, o PEPE divide o espaço de endereçamento em dois bancos, os das células pares e ímpares. Os sinais *(Byte Addressing)* e *(bit of least weight)* de menor peso do barramento de endereços indicam se o processador está a aceder ao banco par ou ímpar (acesso em *byte*) ou aos dois (acesso em palavra).
- Os processadores com endereçamento de *byte* podem ser *little-endian* ou *big-endian*, dependendo do modo como mapeiam uma palavra no espaço de endereçamento. Os *little-endian* colocam o byte de menor peso dessa palavra no endereço mais baixo, enquanto os *big-endian* colocam a/o byte de maior peso.
- Os acessos fazem-se por meio de um ciclo com temporizações bem determinadas (ciclos de leitura ou escrita), que têm de ter a duração suficiente para que as memórias ou periféricos tenham tempo de reagir. Para dispositivos mais lentos é possível atrasar os ciclos de acesso por meio do sinal de *WAIT*.

**SEÇÃO 6.4 – PROLONGAMENTO DOS ACESSOS**

Esta simulação ilustra o funcionamento do prolongamento dos acessos com estados de espera. Os aspectos cobertos incluem os seguintes:

- Geração do sinal de espera por dispositivos lentos e sua ligação ao pino *WAIT*.
- Funcionamento detalhado dos ciclos de acesso de leitura e escrita com prolongamento;
- Verificação do bloqueamento do processador durante o prolongamento.

**6.2 EXCEPÇÕES****6.2.1 PRINCÍPIOS BÁSICOS**

O modelo de computação subjacente à arquitectura básica de um computador estabelece que as instruções são executadas sequencialmente sempre que o programa não efectuar um salto (mudança explícita dessa ordem através da alteração do PC para um dado valor, em vez de um simples incremento). No entanto, também há situações em que excepcionalmente a ordem de execução das instruções é alterada sem ser por vontade do programa. Os eventos que dão origem a estas situações são genericamente designados exceções (por ocorrerem de forma muito menos frequente do que a situação normal de execução de instruções sob controlo do programa).

A forma de atender (o termo mais usual para designar "tratar")<sup>14</sup> uma exceção é invocar uma rotina, que ao regressar retornará a execução do programa no ponto em que a interrompeu. Programa principal é o termo usual para designar o programa cuja execução é interrompida temporariamente e depois retornada.

As exceções podem ter duas origens fundamentais:

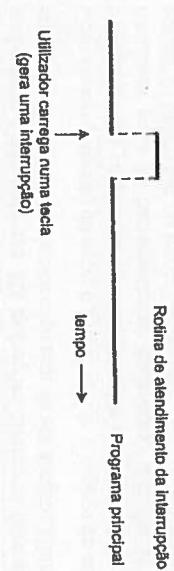
- Externa ao programa, por activação em hardware de pinos próprios do processador, normalmente ligados a periféricos que querem assinalar algo ao processador. Estas exceções têm o nome específico de interrupções e podem ocorrer de forma totalmente assíncrona ao programa (em qualquer altura). A grande vantagem das interrupções é que o programa não tem de estar continuamente a ter o estado dos periféricos. Pode ignorar esse estado completamente e fazer outras actividades sem preocupações, pois quando algo relevante acontecer o programa será automaticamente avisado. Por exemplo, quando se carrega numa tecla de um PC é gerada uma interrupção, que invoca uma rotina para tratar essa tecla. Se o processador do PC tivesse de estar constantemente a ler o estado do teclado para ver se o utilizador tinha carregado numa tecla, quase não fazia mais nada... A Fig. 6.23 ilustra esta situação. A tecla é tratada quando necessário (pode suceder em qualquer altura), sem que o programa principal se tenha de preocupar com isso ou

<sup>14</sup> Também se usa a expressão "servir uma exceção".

sequer se aperceba que tal aconteceu (retorna a execução no ponto em que foi interrompido);

O próprio programa, quando ocasiona situações que o *hardware* não consegue resolver e precisam de ser tratadas em *software* (divisão por zero, por exemplo).

Estas exceções ocorrem apenas durante a execução de instruções específicas, de forma síncrona com o programa, portanto, é só geralmente designadas armadilhas (*traps*), no sentido de que as instruções em que ocorrem possuem mecanismos para apanhar as situações fora do comum.



#### As exceções, quer interrupções quer armadilhas, constituem uma forma de tratar em

*software* situações que o *hardware* consegue detectar mas não resolver. Deste ponto de vista, são uma extensão às capacidades básicas do processador. As interrupções permitem especificar o que o processador deve fazer quando no mundo exterior alguém activa um sinal e as armadilhas dão ao programa a possibilidade de especificar as ações a tomar quando uma dada instrução encontra uma situação que lhe impede de adoptar o seu comportamento básico.

A Tabela 6.11 compara as principais características das exceções e das chamadas normais de rotina. As instruções CALL chamam rotinas indicando qual o endereço onde estas se encontram. As rotinas de exceção (que tratam as exceções) são invocadas directamente pela unidade de controlo do processador, pelo que este precisa de saber quais os endereços das rotinas que tratam as várias exceções. Cada exceção (interrupção ou armadilha) tem um número único, começando em zero. Quando uma exceção ocorre, o processador:

1. usa esse número de exceção para indexar a Tabela de Excepções (que não é mais do que uma enumeração dos endereços das várias rotinas de exceção);
2. obtém o endereço da rotina respectiva;
3. invoca essa rotina (mas de forma mais complexa que um CALL, tal como indicado na Tabela 6.11).

A Tabela de Excepções pode residir em qualquer ponto da memória. O PEPE tem um registo (BTE – Base da Tabela de Excepções) com o fim específico de apontar para esta tabela (contém o endereço da primeira palavra da tabela), registo este que, tal como o conteúdo da tabela, tem de ser inicializado pelo utilizador. O Programa 6.1, na página 474, contém um exemplo de como fazer estas initializações. A Fig. 4.5, na página 204, descreve o conjunto de registos do PEPE.

Tabela 6.11 - Principais características dos mecanismos das excepções

| CHARACTERÍSTICA                             | INTERRUPÇÕES                                                              | ARMADILHAS                                       | CHAMADA DE ROTINA                                                                                            |
|---------------------------------------------|---------------------------------------------------------------------------|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| Objetivo básico                             | Reagir a estímulos exteriores sem estar continuamente a testar se existem | Instruções específicas (DIV, por exemplo)        | Estruturar o código, permitindo invocar o mesmo conjunto de instruções a partir de vários pontos do programa |
| Quem as causa                               | Hardware exterior                                                         | Durante a execução destas instruções             | Instrução CALL                                                                                               |
| Quando podem ocorrer                        | Em qualquer instante                                                      | Durante a execução do CALL                       |                                                                                                              |
| Forma de as tratar                          | Invocar uma rotina de exceção                                             | Invocar uma rotina normal                        |                                                                                                              |
| Quem invoca a rotina                        | Processador, de forma automática                                          | Programador, através do CALL                     |                                                                                                              |
| Onde está a rotina?                         | Endereço está na Tabela de Excepções                                      | CALL indica o endereço                           |                                                                                                              |
| Informação guardada/reposta automaticamente | Contador de Programa (PC) + Registo de Estado (RE)                        | Contador de Programa (PC)                        |                                                                                                              |
| Instrução de retorno da rotina              | RETF (Return From Exception)                                              | RET (Return)                                     |                                                                                                              |
| Onde são tratadas                           | Logo que possível, entre instruções                                       | Imediatamente, durante a execução das instruções | Durante a execução do CALL                                                                                   |
| Mascaráveis?                                | Sim                                                                       | Algumas                                          | Não                                                                                                          |

Tabela 6.11 - Principais características das exceções e das rotinas normais

A Fig. 6.24 mostra como indexar a Tabela de Excepções com o número da exceção para obter o endereço da rotina que atende essa exceção. Neste exemplo, ocorreu a exceção numero 2. Cada entrada na tabela ocupa 2 bytes, o que obriga a multiplicar o número da exceção por 2 antes de o somar à base da tabela, contida no registo BTE.

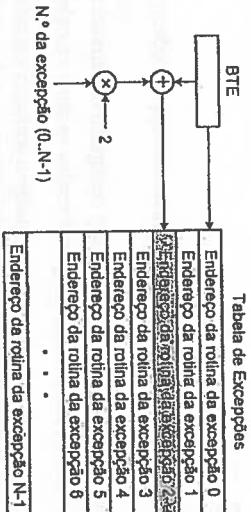


Fig. 6.24 - Uso do registo BTE, da Tabela de Excepções e do número da exceção que ocorreu para determinar o endereço da rotina que a atende

As exceções (e em particular as interrupções) podem ocorrer quando menos se espera, para que o comportamento do programa não se compore de uma forma ou de outra, consoante a altura em que as exceções ocorrem, é fundamental que as rotinas de exceção efectuem o seu processamento e retornem repontudo exactamente todo o contexto (conteúdo dos registos) que estava em vigor quando foram invocadas. Cabe ao programador das rotinas de exceção guardar e depois repor todos os registos cujo conteúdo alterar (como aliás já deve fazer nas rotinas normais como boa prática de programação, mas no caso das rotinas de exceção é imprescindível).

No entanto, os bits de estado (e outros bits) do RE (Registo de Estado) variam de forma tão frequente que o próprio mecanismo de chamada das rotinas de exceção guarda também na pilha o RE, para além do PC. Isto quer dizer que estas rotinas não podem terminar com RET (que só repõe o PC). Existe uma instrução específica (REI) para retorno das rotinas de exceção. Isto quer dizer que não se pode usar uma rotina normal para atender uma exceção nem chamar uma rotina de atendimento de exceção com uma instrução CALL.

Desse ponto de vista, é fundamental que as interrupções nunca sejam atendidas a meio de uma instrução, mas sempre entre duas instruções (senão, tinha também de se guardar o estado interno da instrução. Já as armadilhas têm de ser atendidas mal o erro acontece, pois o hardware (unidade de controlo do processador) não sabe o que há-de fazer para prosseguir. Nesta situação, uma forma típica de recuperar é perder o estado interno da execução dessa instrução, corrigir (se possível) o que originou o erro e repetir (desde o início) a instrução, tal como sucede no caso das exceções de falta de página, no mecanismo de memória virtual (secção 7.6.3, na página 647).

As interrupções são normalmente mascaraveis (é possível o programa dizer ao processador que não quer atender interrupções, mesmo que o hardware as peça), mas muitas das armadilhas possíveis não são normalmente mascaraveis, pois há erros que não é possível ignorar. Estes conceitos são detalhados nas secções seguintes.

## 6.2.2 INTERRUPÇÕES

### 6.2.2.1 PINOS DE INTERRUPÇÃO

Tal como indicado na Fig. 6.25, o PEPE suporta quatro interrupções, pois tem quatro pinos com esta funcionalidade (INT0, INT1, INT2 e INT3). Nesta figura, os periféricos representam qualquer dispositivo que produza um sinal a 0 ou a 1 (pode ser um sensor, um interruptor, outro computador, etc.).

Qualquer dos periféricos A, B, C ou D na Fig. 6.25 pode efectuar um pedido de interrupção, de forma independente dos restantes. Podem inclusivamente fazê-lo todos ao mesmo tempo. O PEPE permite programar a forma como um periférico pode fazer o seu pedido de interrupção, de forma independente para cada um dos pinos, com as quatro hipóteses indicadas na Tabela 6.12.

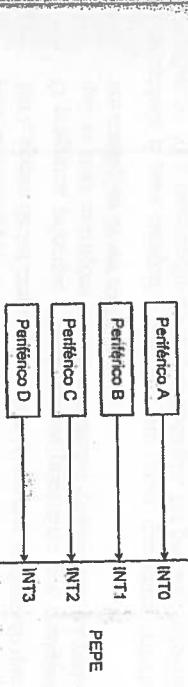


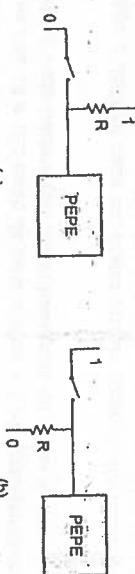
Fig. 6.25 - Pinos de interrupção do PEPE

| DISSENSÍVEL A: | DESCRIPÇÃO                               |
|----------------|------------------------------------------|
| Flanco         | ascendente<br>(0 → 1)                    |
|                | descendente<br>(1 → 0)                   |
| Nível          | <i>Idem</i> , mas para o flanco 1 para 0 |
|                | <i>Idem</i> , mas para o valor 0 do pino |

Tabela 6.12 - Hipóteses de efectuar um pedido de interrupção

**NOTA** É preciso algum cuidado quando se liga um simples interruptor (botão de pressão, por exemplo) a um dos pinos de interrupção, pois ao contrário de qualquer porta lógica um interruptor não é um dispositivo que force 0 e 1. Trata-se apenas de um interruptor que liga dois terminais entre si ou os isola galvanicamente, consoante o estado. O problema é que quando liga os dois terminais consegue forçar um valor num pino de interrupção, se o outro terminal do interruptor ligar a 1 ou a 0, mas quando abre deixa o pino de interrupção "no ar", isto é, ligado a nada. Do ponto de vista eléctrico, dada a alta impedância de uma entrada digital, tal potencia o efeito do ruído electromagnético e o valor realmente "lido" pelo processador nesse pino é aleatório.

Para resolver o problema, basta colocar uma resistência, ligada ao nível a que o interruptor não liga, que garante um valor quando o interruptor está desligado. A Fig. 6.26a mostra como conseguir normalmente 1 e 0 quando se carrega no interruptor, enquanto a Fig. 6.26b mostra o caso inverso.

Fig. 6.26 - Ligação de um interruptor a um dos pinos de interrupção do PEPE.  
(a) – Resistência de pull-up; (b) – Resistência de pull-down

Esta solução não é específica dos pinos de interrupção e é usada sempre que se quer ligar um interruptor a qualquer pino de entrada de um circuito digital (um bit de um periférico de entrada, como por exemplo no caso da Fig. 6.31).

Por omissão, os pinos de interrupção são sensíveis ao flanco ascendente. O PEPE permite mudar esta sensibilidade, alterando os bits de um registo auxiliar designado RCN (Registo de Configuração do Núcleo – ver Tabela A.4, na página 698), e que controla alguns dos recursos internos do núcleo do PEPE. Os 8 bits de menor peso deste registo controlam a sensibilidade das quatro interrupções, de acordo com a Tabela 6.13.

| BITS  | SIGLA | DESCRIÇÃO                                                                                                                                                                                                                                                               |
|-------|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 e 0 | NS10  | Nível de sensibilidade da interrupção 0 (pino INT0). <ul style="list-style-type: none"> <li>* 00=flanco de 0 para 1 (com memória);</li> <li>* 01=flanco de 1 para 0 (com memória);</li> <li>* 10=nível 1 (sem memória);</li> <li>* 11=nível 0 (sem memória).</li> </ul> |
| 3 e 2 | NS11  | <i>Idem</i> , para o pino de interrupção 1 (INT1)                                                                                                                                                                                                                       |
| 5 e 4 | NS12  | <i>Idem</i> , para o pino de interrupção 2 (INT2)                                                                                                                                                                                                                       |
| 7 e 6 | NS13  | <i>Idem</i> , para o pino de interrupção 3 (INT3)                                                                                                                                                                                                                       |

Tabela 6.13 - Formato dos 8 bits de menor peso do RCN  
(Registo de Configuração do Núcleo)

Não há instruções específicas para manipular os bits deste registo. No entanto, é possível ler e alterar o seu valor por meio de uma simples instrução MOV, pelo que a solução é copiar o seu conteúdo para um registo normal, alterar os bits pretendidos e copiar de novo para o RCN. Por exemplo, para alterar a sensibilidade do pino de interrupção INT1 para o nível 1, pode usar-se o código seguinte:

```
MOV R1, 000BAH ; máscara com bits 3 e 2 (NS11) com o valor 10 (nível 1)
MOV R2, RCN ; vai buscar o conteúdo do RCN
OR R2, R1 ; coloca os bits 3 e 2 (NS11) com o valor 10
MOV RCN, R2 ; actualiza o RCN
```

A partir deste momento, a sensibilidade do pino de interrupção INT1 fica em nível 1, sem afectar a sensibilidade dos restantes pinos de interrupção.

### 6.2.2.2 CONTROLO DO ATENDIMENTO DE INTERRUPÇÕES

É possível desligar a sensibilidade do PEPE a cada um dos pinos de interrupção individualmente ou em relação a todos. O RE (Registo de Estado) tem 5 bits para este efeito, tal como ilustrado pela Fig. 6.27, que inclui também os bits de estado já descritos na Tabela 4.7, na página 206.

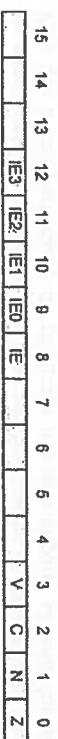


Fig. 6.27 - Bits de controlo das interrupções no RE (Registo de Estado)

O bit IE controla todas as interrupções, enquanto os bits IE0 a IE3 controlam cada uma das quatro interrupções, individualmente. Para o PEPE responder a um pedido da interrupção INT*i* (*i* ∈ [0..3]), os bits IE e IE*i* terão de estar ambos a 1 obrigatoriamente. Assim, é possível ligar ou desligar cada interrupção individualmente ou todas de uma só vez, alterando apenas o bit IE.

O PEPE tem instruções para ligar e desligar cada um destes bits de controlo das interrupções. As instruções EI (Enable Interrupts) e DI (Disable Interrupts) colocam o bit IE a 1 e a 0, respectivamente. As instruções para actuar nos outros bits têm nomenclaturas diferentes, mas acrescidas do número do pino de interrupção respectivo (por exemplo, EI2

EI3, EI4, ...). Um aspecto extremamente importante é que, como parte do mecanismo de atendimento das interrupções (secção 6.2.2.4), o bit IE é colocado a 0 automaticamente quando uma interrupção é atendida e reposto a 1 quando a rotina de atendimento da interrupção é retomada. O objectivo é garantir ao programador que a própria rotina de atendimento de interrupção não é interrompida, dando oportunidade a essa rotina de lidar com recursos gastos do programa sem correr o risco de outra interrupção interferir com essa operação. No entanto, o programador tem a possibilidade de indicar que quer permitir o atendimento de outras interrupções ainda durante a execução da rotina de atendimento da primeira interrupção, desde que execute a instrução EI (que coloca o bit IE a 1) em algum ponto dessa rotina. Isto significa que o comportamento básico é o atendimento sequencial das várias interrupções, mas sob controlo do programador é possível as rotinas de interrupção interromperem-se mutuamente.

Outro aspecto importante é a prioridade no atendimento de interrupções quando há várias pedidas (e com os bits respectivos a 1 no RE). A ordem pela qual várias interrupções pendentes (pedidas mas ainda não atendidas) são servidas tem de estar definida à partida, pois a decisão de qual atender é tomada pelo hardware do processador. O normal é que processadores desejem atender a interrupção 0 a mais prioritária. No PEPE, a prioridade é descendente da interrupção INT0, a mais prioritária, até à interrupção INT3, a menos prioritária.

Notese que:

- A decisão de qual interrupção atender primeiro só tem em conta as interrupções cujos bits no RE estão activos (a 1). Por exemplo, se houver pedidos simultâneos para as interrupções INT0 e INT1, com IE0=0 e IE1=1, a interrupção atendida é a INT1, apesar de ser menos prioritária (claro que assumindo que IE=1);

- O programa principal é ainda menos prioritário que qualquer interrupção, pelo que as suas instruções só são executadas quando não há pedidos de interrupção em condições de ser atendidos.

A prioridade é para cumprir e não tem em conta quaisquer critérios de equidade de atendimento. Se a interrupção INT1, por exemplo, for pedida consecutivamente de forma tão frequente que quando a sua rotina de atendimento termina já houver novo pedido

Nem o programa principal será executado, nem sequer os eventuais pedidos de interrupções INT1 e eventuais interrupções INTO serão executadas.

Alguns processadores usam um esquema de gestão de prioridades que impede uma interrupção de interromper uma rotina de interrupção de maior prioridade que entretanto tenha colocado explicitamente o bit IE a 1 antes de retornar. Isto faz sentido mas é mais complexo de implementar. O PEPE toma uma atitude mais simplista, não distinguindo a prioridade da rotina de interrupção que está a executar. Desde que os bits IE e IE1 estejam a 1 e surja o pedido da interrupção i, esta é atendida. Após colocar o bit IE a 1 durante a execução de uma rotina de interrupção, a única forma de impedir selectivamente o atendimento de certas interrupções é colocar explicitamente os respectivos bits IE1 a 0. Quando a rotina de interrupção regressar, todos os bits de controlo das interrupções serão restaurados ao repor o RE.

### 6.2.2.3 COMPORTAMENTO DAS INTERRUÇÕES

A Fig. 6.23 ilustra o funcionamento básico das interrupções, analisando o comportamento de um programa em vários cenários possíveis de pedidos de interrupção. O programa é composto por apenas algumas instruções, identificadas com as letras A..Q. Há duas rotinas de interrupção, para tratar das interrupções INT0 (instruções X, Y e Z) e INT1 (instruções S e T). Ambas as rotinas terminam com a instrução RET (Return From Exception), simbolizada pela letra R (a negrito). Assume-se, por simplicidade, que todas as instruções são executadas sequencialmente (aparte as invocações das rotinas de interrupção).

Os cenarios representados na Fig. 6.28 sao apenas alguns entre muitos possiveis e destinam-se a exemplificar alguns dos aspectos principais, podendo fazer-se os seguintes comentários:

- Cenário (a) – Os bits IE, IEO e IE1 até podem estar activos (a 1), mas sem haver pedidos de interrupção as rotinas de atendimento das interrupções não são invocadas;

Cenário (b) – Houve um pedido de interrupção INT1 mas a rotina correspondente não foi invocada. Um dos bits IE1 ou IE não estava activo (tem de estar ambos activos, a 1);

Cenário (c) – A interrupção INT1 está programada para ser sensível ao flanco ascendente do sinal no pino INT1. Este flanco ocorre duas vezes durante a execução do programa, o que faz invocar duas vezes a rotina correspondente. Esses flancos ocorrem durante as instruções E e N. Repare-se que a interrupção é atendida apenas no fim da execução destas instruções. O processador invoca a rotina de atendimento da interrupção, que após regressar permite ao programa continuar a execução no ponto em que tinha sido interrompido. Pode confirmar-se que nenhuma instrução do programa deixa de ser executada. O tempo total de execução é que aumenta, devido às duas execuções da rotina;

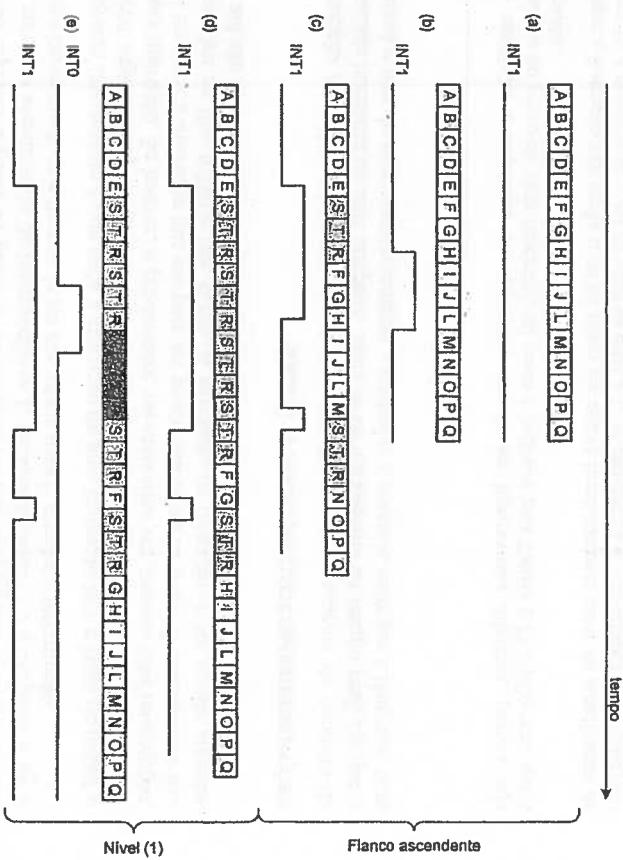


Fig. 6.28 - Exemplos que ilustram o funcionamento das interrupções

- Cenário (d) –** Semelhante ao anterior, mas com a diferença que a interrupção é sensível ao nível 1 e não ao flanco. Isto quer dizer que enquanto o pino INT1 estiver a 1 o pedido de interrupção manter-se-á, mesmo que ela seja atendida. Isto justifica as 4 invocações da rotina. Note-se que não é preciso que o pino esteja activado durante toda a execução da rotina de atendimento, mas apenas até ao atendimento. Uma vez que a rotina comece a ser executada, o pino INT1 pode ser desactivado (com o valor 0) que tal não impede a rotina de executar até ao fim;

**Cenário (e) –** Agora há duas interrupções, ambas sensíveis ao nível 1. A interrupção INT0 só consegue interromper a sequência de invocações da rotina de atendimento de INT1 porque é mais prioritária (qualquer pedido em INT2, por exemplo, teria de esperar que o pino INT1 fosse desactivado para finalmente ser atendido).

Nestes dois últimos cenários nota-se perfeitamente o facto de a rotina de atendimento ser efectuada com o bit IE=0, bit este colocado automaticamente a 0 pelo mecanismo de invocação da rotina e reposto a 1 pela instrução RET. Se não fosse assim, a interrupção INT1 seria atendida de novo ainda antes de a rotina começar a ser executada, pois o pino INT1 está sensível ao nível 1 (e o pino INT1 continua a 1). A consequência seria uma invocação da rotina de atendimento de INT1 com recursividade infinita, esgotando a pilha a breve

#### 6.2.2.4 MECANISMO BÁSICO DE ATENDIMENTO DE INTERRUPÇÕES

Ao atender uma interrupção, o processador suspende o fluxo normal de instruções, invoca e executa a rotina que trata dessa interrupção e ao retornar continua o processamento na instrução seguinte à última executada antes de atender a interrupção. Desse ponto de vista, a invocação de uma rotina de interrupção parece semelhante à invocação das rotinas normais. No entanto, há diferenças fundamentais, que já foram expressas na Tabela 6.11.

A Tabela 6.14 estabelece as operações a efectuar durante o atendimento de uma interrupção, em três fases entre *hardware* e *software*: invocação, processamento e retorno (embora com algumas simplificações que a Tabela 6.15 detalha e corrige).

| FASE DO ATENDIMENTO                                      | OPERAÇÕES A EFECTUAR                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |        |
|----------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|
|                                                          | FASE 1                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | FASE 2 |
| Invocação da rotina de atendimento ( <i>hardware</i> )   | <ul style="list-style-type: none"> <li>Guardar o endereço de retorno (PC) na pilha;</li> <li>Guardar o registo RE na pilha;</li> <li>Colocar o bit IE a zero, desactivando todas as interrupções. Este mecanismo é fundamental para garantir que a rotina de atendimento de outra interrupção pode gerir recursos do computador sem o perigo de indexar a Tabela de Excepções com o número da interrupção a obter o endereço da rotina de atendimento desta interrupção;</li> <li>Colocar no PC o endereço da rotina de atendimento da interrupção, cuja primeira instrução vai ser executada em primeiro lugar na fase seguinte;</li> </ul>                                                                                                                                                                                                                          |        |
| Execução da rotina de atendimento ( <i>software</i> )    | <ul style="list-style-type: none"> <li>Guardar na pilha todos os registos a usar nesta rotina;</li> <li>Efectuar operações críticas, sem interferência de outras potenciais interrupções;</li> <li>Caso seja necessário, avisar o periférico de que o seu pedido de interrupção foi atendido, para que retire o seu pedido do pino de interruptão (tipicamente, isto é feito escrevendo num registo de controlo do periférico);</li> <li>Se houver necessidade de permitir outras interrupções ainda durante esta rotina: <ul style="list-style-type: none"> <li>Colocar a 1 os bits de controlo de interrupções do RE necessários para permitir outras interrupções;</li> <li>Efectuar operações não críticas, podendo ser interrompida por outras interrupções;</li> </ul> </li> <li>Repor os registos guardados no início da rotina, por ordem inversa;</li> </ul> |        |
| Instrução de retorno de exceção, REE ( <i>hardware</i> ) | <ul style="list-style-type: none"> <li>Repor o registo RE a partir da pilha com o valor que tinha antes de atender a interrupção (o bit IE volta a 1);</li> <li>Colocar no PC o endereço de retorno antes guardado na pilha.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |        |

Tabela 6.14 - Operações a efectuar durante o ciclo completo do atendimento de uma interrupção

No entanto, esta tabela não esclarece qual a relação entre o processamento normal de instruções e o atendimento de interrupções. A Fig. 6.29 torna este aspecto mais claro, completando o ciclo normal de processamento (em *hardware*) de uma qualquer instrução (incluindo as da própria rotina de atendimento) com a parte da invocação das rotinas de atendimento.

Todas as instruções (incluindo as das rotinas de atendimento das interrupções) passam pelo ciclo de leitura da memória, descodificação e execução. As instruções são lidas da memória para o RI (Registo de Instruções, não acessível ao programador e a partir do qual as instruções são descodificadas – ver secção 7.2.3, na página 580). Só no fim da execução de cada instrução (nunca no meio da sua execução) é que o processador testa se há interrupções pedidas e permitidas. A ordenação dos testes já sugere a respectiva ordem de prioridades (enquanto houver interrupções INT1, por exemplo).

Se não há interrupções para tratar (não há pedidos ou as interrupções não estão permitidas), o processador prossegue para a próxima instrução directamente. Se houver uma interrupção para processar, o processador guarda o valor do endereço de retorno e do RE na pilha, desactiva todas as interrupções (colocando o bit IE a zero) e indexa a Tabela de Excepções para obter o endereço da rotina a invocar, que coloca no PC.<sup>75</sup>

De seguida, prossegue com o ciclo normal das instruções. Dado o novo valor do PC, a próxima instrução a ser executada será a primeira da rotina de atendimento da interrupção (é assim que se faz a invocação desta rotina).

Por outro lado, a Fig. 6.29 também mostra o que sucede quando a instrução a ser executada é REE, no fim da execução de uma rotina de atendimento de interrupção. O RE e o PC são repostos a partir da pilha. O bit de controlo das interrupções IE volta a estar como antes do atendimento da interrupção (a 1, necessariamente<sup>76</sup>) e o PC passa a ter o endereço da instrução que deixou de ser executada para ir atender a interrupção.

Um aspecto interessante é que após a execução de uma instrução REE e antes de passar à instrução seguinte o processador testa se há interrupções pedidas, tal como faz após qualquer instrução. Se houver, o processador vai mais uma vez atender uma interrupção, e mais uma vez a instrução que era para ser executada deixa de o ser. É exactamente isto que sucedeu na Fig. 6.28d e Fig. 6.28e. A execução da instrução R foi adiada algumas vezes até não haver mais interrupções para atender.

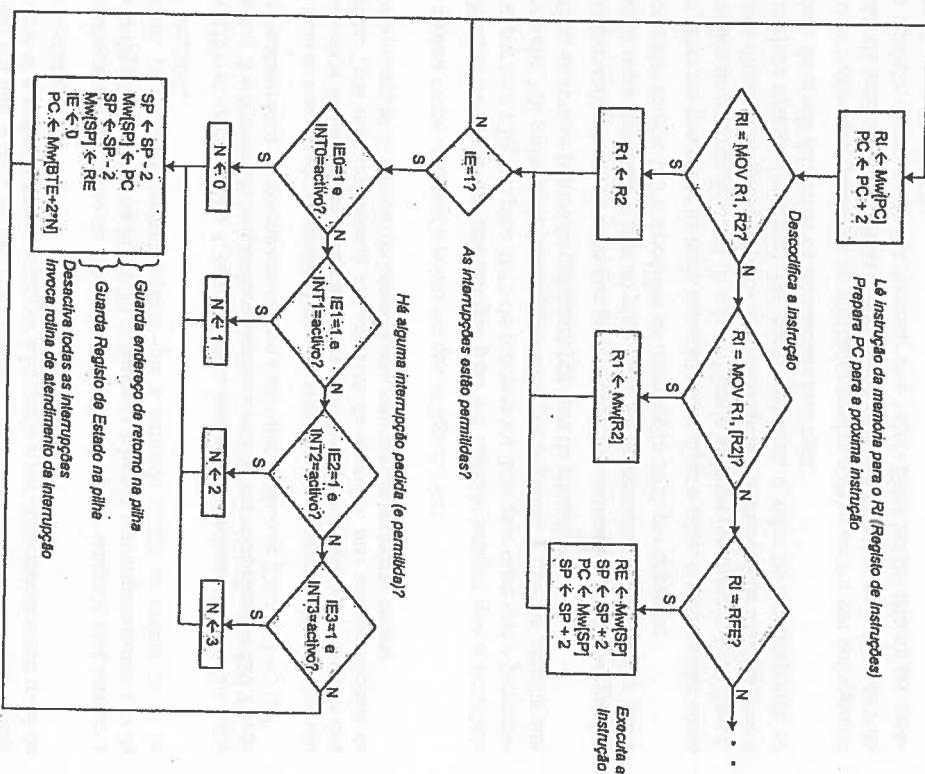
#### 6.2.2.5 PROGRAMAÇÃO COM INTERRUPÇÕES

Esta secção exemplifica a utilização de interrupções com um sistema de controlo da intensidade luminosa de uma lâmpada. Trata-se de um sistema de tempo real, em que o processador tem de estar constantemente a controlar a potência entregue à lâmpada, com

<sup>75</sup> Na realidade, a sequência de operações tem de ser ligeiramente diferente, embora seja equivalente do ponto de vista funcional, tal como é indicado pela Tabela 6.15 e pelo texto que a acompanha.

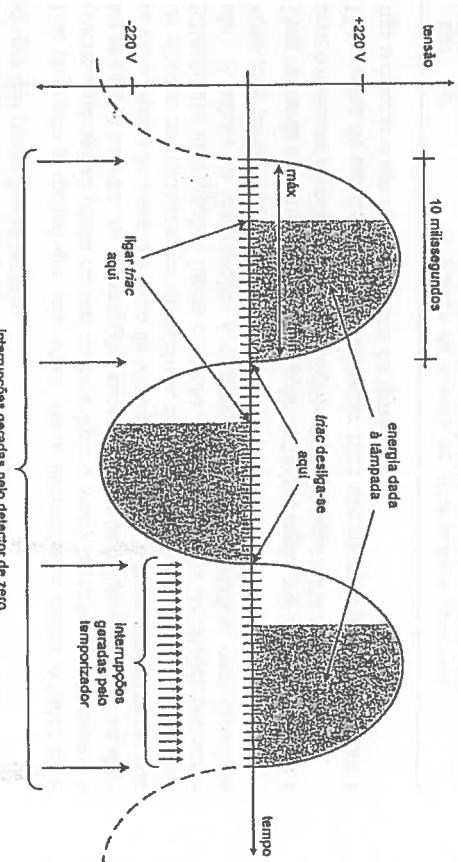
<sup>76</sup> Porque a instrução RE repõe o RE tal como ele estava imediatamente antes do atendimento da interrupção, e obviamente o bit IE tinha de estar a 1. Se estivesse a 0, a interrupção não teria sido atendida.

temporizações sincronizadas com a sinusóide da tensão de alimentação de 220 V e 50 Hz das nossas tomadas de electricidade.



processador tem de tomar decisões em cada 0,1 milissegundos, ou 100 microsegundos, ou 10.000 vezes por segundo. São temporizações bastante rápidas que têm de ser precisas (tem de estar sincronizadas com os 50Hz). As interrupções são a melhor forma de

A Fig. 6.30 ilustra o princípio básico de funcionamento do controlo digital da lâmpada. A tensão de 220V é alternada, sinusoidal e com uma frequência de 50Hz. Do ponto de vista da lâmpada, a polaridade da tensão (positiva ou negativa) é irrelevante. Por isso, o controlo tem de ser feito em cada metade sinusóide, ou a cada 10 milissegundos.



**Fig. 6.30 - Controlo da intensidade luminosa de uma lâmpada de incandescência com recurso a interrupções.** O triac é um interruptor electrónico que permite ligar e desligar a lâmpada

A lampada liga aos 220V por meio de um *triac*, um interruptor electrónico que tem as seguintes particularidades:

- A única forma de desligar o triac é deixar a corrente através dele reduzir-se a zero. Isto acontece sempre que a sinusóide da tensão passa por zero. Depois, mesmo que a tensão volte a subir, o triac só volta a ligar-se com novo impulso na porta;

O triac funciona de igual modo quer na metade positiva quer na metade negativa do período da tensão.

Designam-se sistemas de tempo real todos aqueles em que há temporizações certas a cumprir, não bastando executar logo que possível. A tensão eléctrica na Europa tem uma frequência de 50Hz, o que quer dizer que repete o ciclo em cada 20 milissegundos. O controlo da intensidade luminosa tem de ser feito em cada meia sinusóide, ou a cada 10 milissegundos. Se houver 100 níveis diferentes de intensidade luminosa por exemplo, o

O triac está sempre a ligar e desligar a lâmpada (100 vezes por segundo). Em cada meia sinusóide da tensão, o triac é ligado em algum ponto (que pode variar) e desliga-se sempre quando a sinusóide passa por zero.

O ponto onde se liga o triac determina a quantidade de energia dada à lâmpada durante esse meia sinusóide. Se for logo no início, a lâmpada está na intensidade máxima. Se for quase no fim, a lâmpada acende durante tão pouco tempo que já nem se nota acesa. Se se varia o ponto em que o triac se liga (relativamente ao início de cada meia sinusóide), consegue variar-se a potência média entregue à lâmpada e por conseguinte o seu nível de intensidade luminosa.

Com a persistência do filamento da lâmpada, nem se nota que a lâmpada está sempre a ser ligada e desligada. No caso da Fig. 6.30, a lâmpada é ligada aproximadamente a  $\frac{1}{4}$  da meia sinusóide, pelo que se poderá esperar que a lâmpada esteja na ordem de  $\frac{1}{4}$  da intensidade luminosa.

**Nota** Os 220V são eficazes, isto é, equivalentes a uma tensão contínua de 220V (um valor médio). A sinusóide não se mantém constante, e para ter um valor eficaz de 220 V varia (em módulo) entre o valor mínimo de zero e um valor máximo de  $220 \times 1.41 = 310V$ .

O facto de a tensão não ser contínua mas sim sinusoidal tem apenas como impacte a não linearidade da intensidade luminosa com a percentagem de tempo que a lâmpada está ligada. Varia mais rapidamente nos extremos da sinusóide, mas também depende da nossa percepção de variação da luminosidade, mas precisa em baixos valores.

A Fig. 6.31 mostra como controlar o ponto em que se liga o triac:

- O detector de zero é um circuito que gera um impulso sempre que a sinusóide passa por zero e liga ao pino INT0 do processador. Isto quer dizer que o processador recebe 100 pedidos de interrupção INT0 por segundo, o que lhe permite sincronizar-se com os pontos de passagem por zero da sinusóide;
- O temporizador é um circuito que gera N impulsos durante os 10 milissegundos de cada meia sinusóide e liga ao pino INT1 do processador. Isto significa que o processador recebe  $100 \times N$  pedidos de interrupção INT1 por segundo;
- Para saber em que ponto da meia sinusóide deve ligar o triac, o processador conta P pedidos de interrupção INT1 ( $0 < P < N$ ) desde a última interrupção INT0 (isto é, desde a última passagem por zero). P=N corresponde à intensidade máxima, P=1 corresponde ao número de níveis diferentes de intensidade luminosa;
- Há dois botões que permitem ao utilizador mudar o valor de P e por conseguinte o valor da intensidade luminosa. Cada clique no botão B1 aumenta o valor de P de uma unidade, enquanto B2 faz diminuir este valor. Estes botões têm de ser monitorados como na Fig. 6.26b;
- Os periféricos P1 (saída) e P2 (entrada) permitem ao processador comandar a porta do triac e ler o estado dos botões B1 e B2.

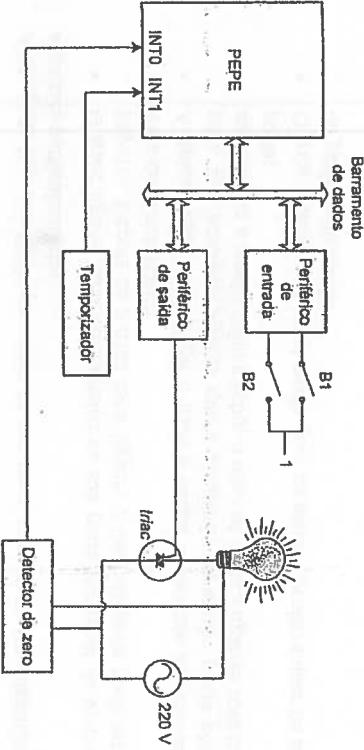


Fig. 6.31 - Circuito de controlo da intensidade luminosa de uma lâmpada usando um triac comandado por um microprocessador. Por simplicidade, a memória está omitida

O Programa 6.1 mostra como se pode implementar a funcionalidade pretendida, sendo constituído por três partes fundamentais:

- Um programa principal que tem como única incumbência alterar o valor de P (correspondente ao ponto em que se deve ligar o triac), estando continuamente a ler os botões B1 e B2. As interrupções encarregam-se de ir ligando o triac na altura certa. Antes de entrar no ciclo de leitura dos botões, o programa principal tem de efectuar as inicializações necessárias, nomeadamente o registo BTE e a sensibilidade das interrupções (neste caso não, pois usam-se as definições por omissão). A tabela de interrupções é construída de forma estática, com directivas WORD, mas também poderia ser construída com instruções MOV;
- Uma rotina de atendimento da interrupção INT0 (passagem por zero) que coloca a zero o contador do número de interrupções INT1 até ligar o triac;
- Uma rotina de atendimento da interrupção INT1 que incrementa este contador e liga o triac se o seu valor for maior ou igual a P.

```

N EQU 10 ; número de níveis de intensidade luminosa
BTE EQU 8000H ; endereço do periférico de saída onde liga o triac
botões EQU 0A000H ; (no bit 0, ligado = 1)
          ; endereço do periférico de entrada onde ligam
          ; os botões (B1 - bit 1, B2 - bit 0). Se o botão
          ; estiver carregado, o bit correspondente vem a 1
          ; valor inicial do SP
PLACE 1000H ; localiza Tabela de Excepções
base: word Rot_int0 ; endereço da rotina de atendimento da interrupção 0
          ; word Rot_int1 ; endereço da rotina de atendimento da interrupção 1
          ; localiza bloco de instruções
PLACE 0000H ; inicializações
início: MOV SP, pilha ; inicializa SP

```

```

MOV BTE, base ; inicializa apontador para a Tabela de Excepções
MOV R8, triac ; endereço do periférico de saída onde liga o triac
MOV R7, botões ; endereço do periférico de entrada onde ligam os botões
MOV R3, N ; valor que indica ao fim de quantas interrupções INT1 se deve ligar a lâmpada (P, no texto)
MOV R2, 0 ; contador de interrupções INT1
EIO - ; permite interrupções INT1
EI ; permite interrupções INT1
; ciclo do programa principal.
; a partir de agora pode haver interrupções
; lâbotões:
MOV R0, [R7] ; lê estado do periférico dos botões
BIT R0, 1 ; vê se o botão B1 foi carregado
B1: JNZ R0, 0 ; se sim, vai tratar deste caso
BIT R0, 0 ; vê se o botão B2 foi carregado
JZ lâbotões ; se não, continua à espera de qualquer botão
CMP R3, 1 ; P <= 1?
JLE B2a ; se sim, não pode decrementar mais
SUB R3, 1 ; decremente valor de P
B2a: MOV R0, [R7] ; lê estado do periférico dos botões
BIT R0, 0 ; vê se o botão B2 continua carregado
B2a: JNZ R0, 0 ; se sim, espera que o botão B2 seja largado
JMP lâbotões ; B2 tratado, volta a testar os dois botões
B1: MOV R1, N ; P >= N ?
CMP R1, RL ; se sim, não pode incrementar mais
JGE Bla ; incrementa valor de P
ADD R3, 1 ; lê estado do periférico dos botões
MOV R0, [R7] ; R0, 1 ; vê se o botão B1 continua carregado
BIT R0, 1 ; se sim, espera que o botão B1 seja largado
Bla: JNZ R0, 0 ; se sim, espera que o botão B1 seja largado
JMP lâbotões ; B1 tratado, volta a testar os dois botões
; Rot_int0 - Rotina de atendimento da interrupção INT0.
; Invocada sempre que a sinusóide da tensão passa por zero.
; Coloca a zero o contador de interrupções INT1 (no registo R2).
; Conta o número de interrupções INT1 (no registo R2). Se chegar ao
; valor P (indicado por R3) liga o triac
; *****lâbotões***** ; retorna a zero o contador de interrupções INT1
; Rot_int1 - Rotina de atendimento da interrupção INT1.
; Invocada N vezes em cada meia sinusóide da tensão.
; Conta o número de interrupções INT1 (no registo R2). Se chegar ao
; valor P (indicado por R3) liga o triac
; *****lâbotões***** ; retorna a zero o contador de interrupções INT1
Rot_int1:
ADD R2, 1 ; incrementa o contador de interrupções INT1
CMF R2, R3 ; já chegou ao valor P?
JLT acaba ; se ainda não, já não há mais nada a fazer, agora
PUSH R0 ; tem de guardar o valor do R0
MOV R0, 1 ; liga o triac
MOV R0, 0 ; termina impulso para o triac
MOV R0, [R8], R0 ; repõe o valor de R0
POP R0 ; retorna da rotina de interrupção
acaba:
RFE

```

Programa 6.1 - Programa que implementa o controlo da intensidade luminosa da lâmpada

## Notas sobre este programa:

- O valor de N para um circuito real e para permitir variações suaves de intensidade luminosa deveria ser na ordem de 100. Neste programa considerou-se apenas 10 por causa da Simulação 6.5, pois o simulador não consegue tratar tantas interrupções por segundo. Mas como N é definido num EQU é fácil mudar o seu valor.
- A Tabela de Excepções é construída simplesmente com directivas WORD, especificando os endereços das várias rotinas de atendimento de interrupção. Note-se que neste exemplo são apenas definidas as entradas da tabela referentes às interrupções INT0 e INT1. Se se permitissem interrupções INT2, por exemplo, sem declarar a rotina correspondente na tabela, e se gerassem interrupções INT2 (ligando algum sinal ao pino INT2), o processador acederia à posição 2 da tabela e usaria o valor que obtivesse como endereço da rotina invocada. O mais provável seria o programa baralhar-se completamente;
- É obrigatório inicializar o registo BTE (base da Tabela de Excepções) com o endereço da primeira palavra da tabela;
- As interrupções também não funcionam sem serem explicitamente permitidas. Tem de se permitir cada uma das interrupções, individualmente, e depois permitir globalmente (com a instrução EI). Estas permissões devem ser feitas apenas após as initializações, para garantir que as interrupções só começam a ser atendidas após estar tudo preparado;
- As rotinas de interrupção não devem ser chamadas por CALL e devem terminar obrigatoriamente por RET em vez de RETI;

- Uma rotina de interrupção nunca deve estragar qualquer registo, pois nunca se sabe em que ponto pode ocorrer uma interrupção. Exceptua-se o RE, pois esse é guardado automaticamente na invocação da rotina de interrupção. Todos os outros usados têm de ser guardados e repostos, tal como R0 na rotina Rot\_int1. Note-se que basta guardar apenas quando necessário, tal como ilustrado nesta rotina. Note-se ainda que R2 é alterado nestas rotinas de interrupção, mas obviamente não é preciso guardá-lo! Trata-se de uma variável global, e as rotinas de interrupção devem mesmo alterar o seu valor.

## SIMULAÇÃO – FUNCIONAMENTO BÁSICO DAS INTERRUPOES

Esta simulação ilustra o funcionamento básico das interrupções, com dois exemplos:

- Exemplo académico, apenas com as initializações básicas, geração manual das interrupções (com botões ligados aos pinos de interrupção) e um ciclo infinito como programa principal. Com execução passo a passo e pontos de paragem, este exemplo permite verificar manualmente e ao ritmo do utilizador o que se passa quando surgem as interrupções;
- Exemplo do Programa 6.1, como aplicação completa de tempo real. Na prática, a frequência de trabalho da lâmpada é apenas de 1 Hz, para limitar o número de

interrupções por segundo e não exceder a capacidade de cálculo do simulador. Mas o princípio é o mesmo e as conclusões a extrair também.

### 6.2.2.6 CONTROLADOR DE INTERRUPÇÕES

Alguns processadores incluem um pino de interrupção cujo funcionamento é mais complexo do que o descrito até aqui para os pinos de interrupção do PEPE, permitindo suportar vários pedidos de interrupção num só pino com recurso a um controlador de interrupções externo (PIC – *Programmable Interrupt Controller*, ou Controlador de Interrupções Programável). Nestes casos, o processador tem também outro pino (de saída), com o nome típico de INTA – *Interrupt Acknowledge*, ou Reconhecimento de Interrupção, para indicar ao controlador que está a atender um dos seus pedidos de interrupção. O controlador tem de indicar qual a interrupção (na tabela, ou vetor, de interrupções) a atender, razão pela qual este tipo de pino de interrupções recebe geralmente a designação de interrupção vectorizada, enquanto que as interrupções descritas nas secções anteriores são conhecidas por interrupções simples.

Um PIC consegue tipicamente lidar com oito interrupções, e alguns sistemas permitem ligar vários PICs entre si de modo a suportar mais. A programabilidade do PIC tem a ver com a permissão ou inibição de cada uma das oito interrupções, modo de sensibilidade dos pinos de interrupção, prioridades relativas, número das interrupções geradas, etc.

A Fig. 6.32 mostra o esquema de ligação do PIC aos periféricos e a um processador e permite descrever o seu funcionamento. Neste exemplo, o processador suporta três interrupções simples e uma vectorizada.

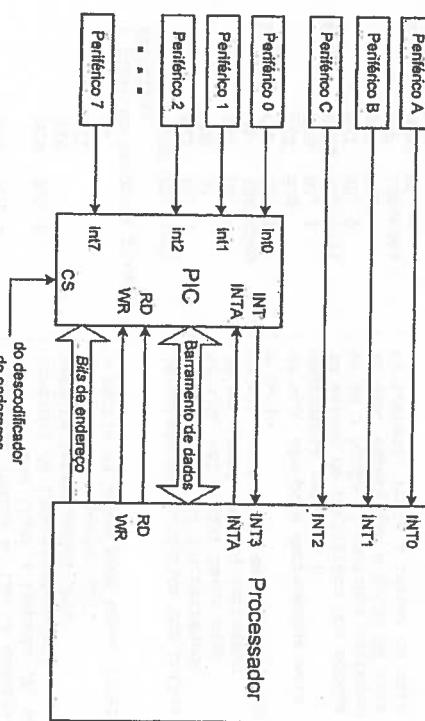


Fig. 6.32 - Controlador de Interrupções programável (PIC) e coexistência entre pinos de interrupção simples e vectorizados

O processador "vê" o PIC em duas perspectivas:

Periférico com alguns portos, com todos os sinais correspondentes: RD, WR, sinal de seleção (CS), barramento de dados e alguns bits do barramento de endereços (os necessários para suportar os portos). Cada porto corresponde a um registo interno, com as seguintes funcionalidades típicas:

- Bits de permissão/inibição de cada linha de interrupção (porto de escrita);
- Configuração da sensibilidade de cada linha de interrupção: flanco, nível, etc. (porto de escrita);
- Número base da interrupção INT0 na Tabela de Excepções (porto de escrita). Os números das restantes interrupções obtêm-se somando 0 a 7 (consoante a interrupção) a esse número base;
- Estado das linhas de interrupção, indicando quais os pedidos de interrupção pendentes (porto de leitura);
- Definição das prioridades (porto de escrita).

Ronte de interrupções, no pino INT3, cujo funcionamento básico é tipicamente o seguinte (poderá ter variantes de processador para processador):

O processador programa o PIC accedendo-lhe como periférico, escrevendo nos seus registos internos especificando em quais dos seus pinos aceita pedidos de interrupção, qual a sua sensibilidade, número base das interrupções, etc.;

O PIC recebe pedidos de interrupção nos seus pinos e activa o pino INT3 para assinalar ao processador que há pelo menos um pedido pendente;

Quando o processador puder atender esse pedido de interrupção INT3, faz um ciclo de leitura do espaço de endereçamento da memória, mas activando o pino INTA em vez do pino RD (Fig. 6.33);

Reagindo à activação do INTA, o PIC coloca no seu barramento de dados um byte (designado vector) com o número de interrupção a gerar (correspondente ao periférico mais prioritário e cujo pedido de interrupção deve ser atendido em primeiro lugar). O valor do barramento de endereços neste acesso é irrelevante. Dado que RD não foi acuado, nem a memória nem os periféricos são seleccionados (isto significa que só pode haver um controlador de interrupções principal no sistema, embora tipicamente possam ser ligados em cascata, com a saída de um ligado no pino de interrupção de outro);

No flanco ascendente de INTA, o processador lê esse byte, multiplica-o por 2 (se for um processador de 16 bits com endereçamento de byte, como o PEPE) e soma-o com endereço de base da Tabela de Excepções (BTE, se fosse o PEPE), obtendo o endereço da palavra nesta tabela que contém o endereço da rotina de atendimento da interrupção (Fig. 6.24). Após guardar o endereço de

retorno e o RE, o processador coloca esse endereço no PC e vai processar a instrução seguinte, como em qualquer interrupção simples (Fig. 6.29).

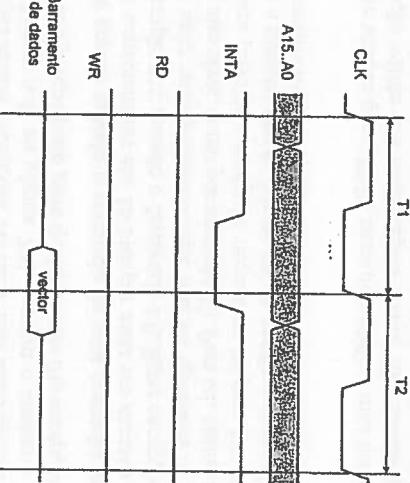


Fig. 6.33 - Ciclo de leitura do vetor do controlador de interrupções programável (PRC)

Note-se que a partilha de um só pino de interrupção simples de um processador por vários periféricos (fontes de interrupção, de uma forma mais genérica) é perfeitamente possível, com *hardware* extra (OR de todos os pedidos, por exemplo) e algumas instruções (que leiam os pedidos individuais das várias fontes de interrupção, por exemplo) que decidam qual a interrupção concreta que deve ser atendida. O mecanismo descrito nesta secção destina-se apenas a implementar esta partilha em *hardware*, de forma mais eficiente. Dada esta alternativa, e uma vez que precisa de ser simples, o PEPE não suporta interrupções vectorizadas (quatro interrupções simples são suficientes para a esmagadora maioria das aplicações pedagógicas).

## 6.2.3 OUTRAS EXCEPÇÕES

**6.2.3.1 INVOCAÇÃO EXPLÍCITA E RETORNO DE UMA EXCEPÇÃO**

A Tabela 6.15 mostra as operações elementares envolvidas na execução das instruções SWE e RFE (a Tabela 6.14 e a Fig. 6.29 têm apenas uma versão simplificada). A invocação automática (por *hardware*) de uma rotina de atendimento no caso de ocorrer uma exceção faz desenrolar as mesmas operações que na instrução SWE.

Notas importantes:

- Na instrução SWE, o PC (endereço de retorno) é o primeiro registo a ser guardado na pilha. No entanto, o mecanismo de protecção do PEPE (assunto tratado na secção 7.7.5, na página 682) obriga a que o bit NP do RE seja colocado a 0 antes, tal como os bits IE (para impedir novas interrupções) e DE (para impedir operações de DMA, descritas na secção 6.4.2.3, na página 511). Para que depois se possa também guardar na pilha o valor que o RE tinha imediatamente antes do atendimento da interrupção, este é guardado temporariamente no registo TEMP. A mesma razão está na base da recuperação do valor do RE (na instrução RFE) para o registo TEMP primeiro e só depois para o registo RE;

Esta instrução desempenha um papel semelhante ao da instrução CALL para as rotinas normais, mas com algumas diferenças fundamentais:

- Para além de guardar o endereço de retorno também guarda o RE;
- O endereço da rotina a invocar é indicado pela Tabela de Excepções e não pela própria instrução;
- Só pode ser usada para invocar rotinas de excepção (terminadas com RFE).

Note-se que esta instrução pode ser usada para invocar qualquer excepção, incluindo as interrupções. Basta indicar o número da excepção que se pretende. A sua utilidade revela-se sobretudo em dois campos:

- Teste e depuração do programa. Para exercitar as interrupções ou qualquer outra excepção, pode fazer-se uma rotina de teste que invoque explicitamente essa excepção com a instrução SWE. Assim, é possível testar se a rotina de atendimento dessa excepção funciona bem, sem necessidade de ter a causa real que normalmente gera essa excepção (isto pode ser particularmente útil com interrupções, pois como são assíncronas nem sempre é fácil gerar as interrupções de forma sistemática e a ritmo controlado. Note-se que SWE invoca uma rotina de atendimento de uma interrupção mesmo com as interrupções inibidas pelos bits IE do RE);

Chamadas a rotinas do sistema operativo ou de uma biblioteca de funções de apoio ao programa. A grande vantagem face a um simples CALL é que o programa não tem de especificar o endereço real onde a rotina se encontra, mas apenas o seu número na Tabela de Excepções. Por um lado, isto permite alterar e recompilar o sistema operativo ou a biblioteca de funções sem alterar o programa do utilizador. Por outro, constitui um mecanismo de protecção contra um eventual mau comportamento do programa que invoca essas rotinas (por erro ou por malícia – caso dos vírus, por exemplo), algo muito importante em qualquer computador e que é descrito na secção 7.7.5, na página 682.

pilha), testar o código de operação usado e proceder de acordo com a funcionalidade pretendida.

| SINTAXE ASSEMBLY |                 | OPERAÇÕES (RTL)                                                                                                       |  |
|------------------|-----------------|-----------------------------------------------------------------------------------------------------------------------|--|
| SWE              | número-excepção | TEMP ← RE<br>RE (NP, IE, DE) ← 0<br>Mw[SP-2] ← PC<br>Mw[SP-4] ← TEMP<br>PC ← Mw[BTE+2*número-excepção]<br>SP ← SP - 4 |  |
| RFE              |                 | TEMP ← Mw[SP]<br>PC ← Mw[SP+2]<br>SP ← SP + 4<br>RE ← TEMP                                                            |  |

Tabela 6.15 - Operações elementares envolvidas numa instrução SWE (Software Exception)

- As instruções que fazem vários acessos à memória são problemáticas quando o mecanismo de memória virtual está ligado (seção 7.6, na página 643), pois poderá ocorrer uma excepção se um dos acessos falhar e deixar o SP meio alterado e portanto inconsistente. Por este motivo, o SP só é alterado no fim da instrução. Se algum acesso à memória falhar, a execução da instrução pode ser abortada sem ter de desfazer alterações já feitas e mais tarde recomeçada sem ter de recuperar de um estado intermédio.

### 6.2.3.2 EXCEPÇÕES PREDIFINIDAS

Há várias situações anormais em que o hardware do PEPE não sabe o que fazer e invoca uma excepção para permitir ao programador especificar o que fazer nesses casos. A Tabela 6.16 descreve algumas dessas excepções. Há mais, mas serão apenas descritas nas secções 7.6.7 e 7.7.5. A Tabela A.8, na página 702, contém o conjunto completo. A exceção SWE não tem número porque pode gerar qualquer uma das excepções.

A excepção de EXCESSO é gerada quando o resultado de uma operação aritmética não consegue ser representado correctamente em 16 bits por estar em excesso. A excepção DIV ocorre apenas na instrução DIV quando o dividendo é 0. Estas excepções podem ser inhibidas, para permitir que estes erros possam ocorrer mas ser ignorados (o que acontece em algumas linguagens de alto nível, como por exemplo C). Para tal, usam-se dois bits no RE, que têm de estar a 1 para permitir a excepção respectiva: TV para EXCESSO e TD para DIV. A Fig. 6.34 mostra a posição destes bits no RE. A seguir a uma inicialização do PEPE (*reset*), estes bits encontram-se a 0 e devem ser colocados explicitamente a 1 para permitir estas excepções.

A exceção COD\_INV ocorre sempre que o PEPE tenta descodificar uma instrução que não tem um código de operação válido (há codificações livres, não usadas por nenhuma instrução). Isto constitui um excelente mecanismo de estender o conjunto de instruções do PEPE em software. É apenas uma questão de o compilador ou *assembler* gerar as instruções extra com códigos de operação não usados e de a rotina de atendimento desta excepção ir ler a instrução que provocou a excepção (usando o valor do PC guardado na

| Nº MÉRIO | NOME          | CALCA                                                | OCORRE EM          | MÁSCARA | ATENDIMENTO                  |
|----------|---------------|------------------------------------------------------|--------------------|---------|------------------------------|
| —        | SWE           | Execução desta instrução                             | Instrução          | —       | Imediato                     |
| 0        | INT0          | Activação externa do pino INT0                       | Qualquer altura    | Sim     | Após Instrução em que ocorre |
| 1        | INT1          | Activação externa do pino INT1                       | Qualquer altura    | Sim     | Após Instrução em que ocorre |
| 2        | INT2          | Activação externa do pino INT2                       | Qualquer altura    | Sim     | Após Instrução em que ocorre |
| 3        | INT3          | Activação extrema do pino INT3                       | Qualquer altura    | Sim     | Após Instrução em que ocorre |
| 4        | EXCESSO       | Excesso em operação aritmética                       | Instrução          | Sim     | Imediato                     |
| 5        | DIV0          | Divisão (DIV) por zero                               | Instrução          | Sim     | Imediato                     |
| 6        | COD_INV       | Código de operação Inválido                          | Descodificação     | Não     | Imediato                     |
| 7        | D_DESALINHADO | Acesso em 16 bits à memória (MOV) com endereço ímpar | Instrução          | Não     | Imediato                     |
| 8        | I_DESALINHADO | Busca de instrução com PC ímpar                      | Busca de instrução | Não     | Imediato                     |

Tabela 6.16 - Algumas das excepções predefinidas (em hardware) no PEPE

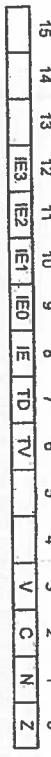


Fig. 6.34 - Registo de Estado (RE) incluindo os bits TD e TV, que controlam as excepções DIV0 e EXCESSO, respectivamente

A exceção D\_DESALINHADO é gerada sempre que uma instrução MOV de acesso à memória, em escrita ou em leitura, especificar um endereço ímpar. O PEPE suporta apenas acessos de 16 bits alinhados, com endereço par (seção 6.1.5.3). A exceção I\_DESALINHADO é semelhante, mas ocorre durante uma busca de instrução da memória, em que o processador usa o PC como endereço, caso este tenha um valor ímpar. Os acessos de busca de instrução são em palavras (16 bits), pelo que têm de ser alinhados. Ao contrário das anteriores, as excepções COD\_INV, D\_DESALINHADO e I\_DESALINHADO não podem ser mascaradas (estão sempre permitidas). Se ocorrerem, a rotina de excepção é invocada. Isto significa que a inicialização do processador deve preencher correctamente a Tabela de Excepções tão cedo quanto possível, antes que qualquer destas excepções ocorra.

Na Tabela 6.16 pode verificar-se que as interrupções esperam que a instrução em execução acabe para que sejam atendidas, ao contrário das restantes excepções, ocorridas durante a execução de uma instrução, que abortam a execução dessa instrução e são atendidas imediatamente.

**ESSENCIAL**

As exceções são eventos que ocorrem excepcionalmente, pelo que os programadores não testam explicitamente a sua ocorrência. O processador invoca automaticamente uma rotina para cada uma das exceções possíveis, sempre que ela ocorre.

As exceções podem ser causadas pelo próprio programa (erro numa instrução) ou por periféricos externos ao processador (activando pinos deste).

Estas últimas são conhecidas por interrupções.

A Tabela de Excepções é uma tabela com os endereços das rotinas a invocar para cada exceção. Quando uma exceção ocorre, o processador usa o seu número para indexar a tabela e obter o endereço da rotina a invocar.

Nesta invocação é guardado automaticamente na pilha não apenas o endereço de retorno mas também o RE (Registo de Estado). Todos os restantes registos que a rotina invocada altere devem ter antes guardados na pilha e restaurados antes de a rotina regressar, pois nunca se sabe quando uma exceção ocorre.

As interrupções podem ser mascaradas (inibidas) por meio de bits adequados no RE. Para aumentar o grau de controlo do programador, quando uma rotina de exceção é atendida, as interrupções são automaticamente inibidas (voltando ao estado anterior quando a rotina retorna).

As rotinas de exceção devem terminar pela instrução RETF (Return From Exception) e só podem ser invocadas automaticamente pela ocorrência de uma exceção ou explicitamente pela instrução SWI (nunca por CALL).

- Tal como isso se podem usar rotinas (como GEL) sem inicializar o SP, não se podem usar exceções (normalmente interrupções) sem programar todas as rotinas das exceções a usar;
- construir a tabela de exceções (com directivas WORD, por exemplo) para inicializar os registos SP e BP e os bits do RE que controlam a inibição das interrupções (o geral, IE, e o específico de cada interrupção);
- Uma interrupção nunca interrompe a execução de uma instrução.

**6.3 TIPOS DE PERIFÉRICOS****6.3.1 O QUE É UM PERIFÉRICO?**

Convém neste ponto fazer uma distinção entre os dois significados normalmente dados à palavra "periférico":

O exemplo canónico deste tipo de periféricos é o disco magnético, mas há vários outros exemplos:

- Discos ópticos (CD-ROMs e DVDs, com variantes graváveis e regraváveis);
- Discos magneto-ópticos;
- Dispositivos totalmente electrónicos (*pen-drives*).

Estes últimos têm a vantagem de não ter componentes mecânicos, mas ainda não conseguem competir com os discos magnéticos em fiabilidade, capacidade de armazenamento e custo por megabyte de capacidade.

Por "disco" deve entender-se o conjunto do disco propriamente dito e do seu controlador, um circuito que gere o disco e implementa as suas várias operações.

Os discos magnéticos armazenam dados à custa da magnetização da superfície do disco por meio de uma cabeça magnética. Cada unidade pode ter vários discos, e cada disco tem duas superfícies. O conjunto está montado de forma sólida num viseiro que gira a uma velocidade de 5400, 7200 ou 10000 rotações por minuto (valores típicos). Por cada disco há dois braços, cada um com uma cabeça magnética na ponta e com o disco a deslizar apenas a cerca de 0,5 micrões (0,0005 milímetros) de distância. O braço pode mover a cabeça para qualquer posição radial do disco, o que associado ao movimento

Dispositivo de entrada/saída, que liga directamente aos barramentos (de dados, de endereços e de controlo) do computador e que pode ser lido ou escrito directamente pelo processador com instruções de acesso à memória (como MOV). Pode incluir vários portos (endereços individuais) que correspondem normalmente a registos internos de controlo desse periférico e a registos cujos bits estão disponíveis para ligar a hardware específico. É o caso dos dispositivos P1 a P4 da Fig. 6.2 na página 415, por exemplo:

Subsistema (normalmente complexo e frequentemente incluindo o seu próprio processador) que liga ao computador principal por meio de uma interface adequada (que será do tipo anterior). É o caso de um disco de um PC, por exemplo. Os periféricos simples são aqueles que tipicamente possuem menos inteligência própria, oferecendo menos funcionalidade e precisam de menos desempenho em termos de transferência de informação. Qualquer computador tem periféricos de bits de entrada/saída, temporizadores, controladores de interrupções, etc.

Os periféricos mais complexos interligam normalmente ao computador propriamente dito por interfaces e protocolos normalizados, o que permite aos fabricantes desses periféricos usá-los em vários computadores e aos fabricantes de computadores poderem escolher o fabricante dos periféricos sem estarem dependentes dele.

As secções seguintes descrevem muito brevemente os aspectos mais essenciais desses tipos de periféricos.

**6.3.2 PERIFÉRICOS DE MEMÓRIA DE MASSA**

O exemplo canónico deste tipo de periféricos é o disco magnético, mas há vários outros exemplos:

- Discos ópticos (CD-ROMs e DVDs, com variantes graváveis e regraváveis);
- Discos magneto-ópticos;
- Dispositivos totalmente electrónicos (*pen-drives*).

Estes últimos têm a vantagem de não ter componentes mecânicos, mas ainda não conseguem competir com os discos magnéticos em fiabilidade, capacidade de armazenamento e custo por megabyte de capacidade.

angular do disco consegue, na prática, levar a cabeça a qualquer ponto da superfície do disco (Fig. 6.35b). Se as cabeças tocarem nos discos danificam-se, pelo que não suportam grandes choques ou vibrações. Quando o disco se desliga, os braços recolhem automaticamente para as cabeças ficarem fora dos discos, ficando mais protegidas (Fig. 6.35a).

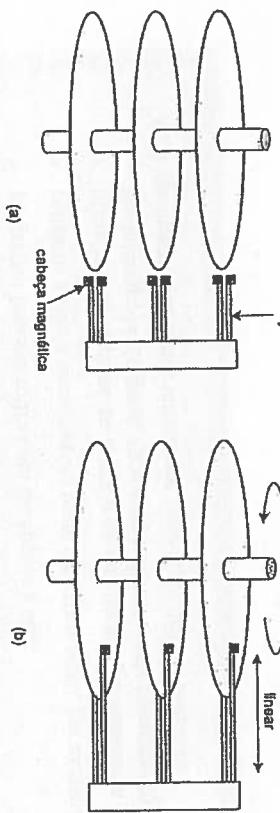


Fig. 6.35 - Com movimento de rotação do disco e linear do braço conseguem posicionar-se a cabeça magnética em qualquer ponto do disco. (a) - Posição de repouso; (b) - Braço posicionado num cilindro intermédio

Na realidade, isto só é metade verdade. O braço consegue mudar a posição axial da cabeça (distância ao eixo central), mas terá de esperar que o disco rode até o ponto pretendido do disco passar junto à cabeça.

Mantendo o braço fixo com o disco a rodar, a cabeça passa periodicamente junto a todas as posições angulares à mesma distância axial. Isto define uma pista circular em cada disco (Fig. 6.36b), contendo uma sequência de bytes que pode ser lida ou escrita à medida que o disco roda sem alterar a posição da cabeça. O conjunto das pistas correspondentes nas duas superfícies dos vários discos designa-se cilindro (todas as cabeças estão posicionadas na mesma pista). O número de cilindros existentes depende do diâmetro dos discos, da largura de cada pista, da precisão de posicionamento do braço, etc. São questões tecnológicas como estas que limitam a capacidade do disco.

No acesso aos discos nunca se accede a um só byte. Cada pista está organizada em vários blocos de bytes, designados sectores (Fig. 6.36c). As pistas exteriores têm mais sectores do que as interiores, pois o seu diâmetro é maior (Fig. 6.36a) e o factor limitativo é a densidade de bits. Um acesso a um disco implica ler ou escrever pelo menos um sector. A unidade de disco tem uma pequena memória, do tamanho de um sector (pelo menos), a partir da qual se pode ler o byte pretendido. Para escrever um byte é preciso ler o seu sector inteiro, alterar esse byte e escrever de novo. Dado que estatisticamente se verifica que os programas fazem várias leituras ou alterações de bytes em endereços perto uns dos outros, esta organização optimiza globalmente os acessos.

A título de exemplo, um disco de 1 TByte (1000 GBytes) poderá estar organizado em 3 discos (6 superfícies), com cerca de 167 GBytes por superfície. Dado que o limite é a densidade de bits por unidade de superfície, a distribuição por pistas e sectores é variável

com a localização radial dos dados no disco (os fabricantes já nem divulgam a geometria interna). Actualmente o tamanho do sector usado é de 512 bytes mas, devido ao aumento das capacidades dos discos, esse valor passará a ser de 4 KBytes a partir de 2011.

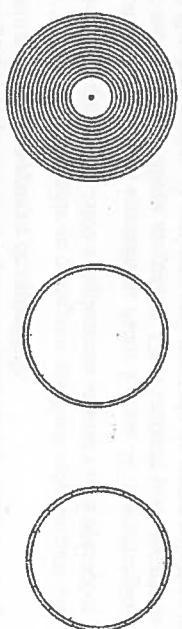


Fig. 6.36 - Organização dos dados do disco. (a) - Conjunto de pistas concêntricas; (b) - Uma única pista, a que é possível aceder com o braço fixo; (c) - Pista com sectores marcados

tempo de acesso a um sector é fundamentalmente o somatório de três tempos:

**Tempo de procura de pista (seek time)** – Será mínimo se o braço estiver na pista ao lado, e tanto maior quanto maior for o número de pistas a atravessar. O tempo de procura médio típico é da ordem de 10 ms ou inferior (6 a 8 ms são valores típicos);

**Atraso rotacional (rotational delay)** – O braço posiciona-se mecanicamente, mas para chegar ao sector pretendido dentro de uma pista basta esperar que o disco rode até ao ponto certo. O valor mínimo é zero, se o sector aparecer logo que se faça o pedido, e o valor máximo é o tempo de uma rotação do disco, se o sector pretendido tiver acabado de passar. Em média, é meia rotação. Se a velocidade de rotação do disco for de 7200 rpm, ou de 120 rotações por segundo, meia rotação demora cerca de 4 ms;

**Tempo de transferência (transfer time)** – Tempo de leitura ou escrita do sector, que se pode obter aproximadamente multiplicando a taxa de transferência (*transfer rate*) do disco (indicada pelo fabricante) pelo tamanho desse sector. Esta taxa depende essencialmente da velocidade de rotação do disco e da densidade de bits no disco (na prática, o número de bytes por pista). Considerando o exemplo do disco de 1 TBytes com 512 KBytes em média por pista (1000 sectores de 512 bytes) e uma velocidade de rotação de 7200 rpm (leitura de 120 pistas/segundo), poder-se-á esperar uma taxa de transferência de uns 60 MBytes/s. Um sector de 512 bytes será lido em cerca de 0,008 ms. Nitidamente, não é este tempo o factor dominante num tempo de acesso médio na ordem dos 10 ms.

A maioria do tempo de acesso a um disco é portanto gasta à espera, o que se designa por latência. Felizmente, o processador poderá executar outras tarefas enquanto o controlador do disco executa o pedido de acesso ao sector. Note-se que o controlador de disco tem uma zona de memória interna para conter pelo menos um sector. Um sector é primeiro lido para esta zona de memória e só depois transferido para a memória do processador. Depois de escrito, o sector é primeiramente escrito nessa zona de memória e só depois escrito no disco. Este esquema é necessário por duas razões fundamentais:

A velocidade de rotação do disco é constante. Quando o sector a aceder passa pela cabeça magnética, tem de ser acedido nessa altura. Se o processador estivesse a atender uma interrupção ou a executar outra tarefa qualquer, poderia perder a oportunidade e ter de esperar por mais uma rotação do disco até o sector passar de novo. O controlador do disco garante que o acesso é feito quando for a altura certa.

A taxa de transferência de e para o disco não é necessariamente a mesma com que o processador lê ou escreve dados na memória. A zona de memória do controlador funciona à velocidade do disco ao aceder ao disco, e à velocidade do processador quando é acedido por este.

Apesar de serem dispositivos electromecânicos e rodarem a alta velocidade com uma cabeça magnética quase a roçar a superfície, os discos magnéticos são muito fiáveis. Os computadores portáteis, actualmente tão divulgados, representam um desafio para estes periféricos, pelo perigo de vibrações mecânicas que representam. Os computadores mais recentes incluem sensores electrónicos de movimento (acelerómetros) que recolhem automaticamente o braço (protegendo as cabeças magnéticas e evitando que elas "aterrrem" nos discos) em caso de ser detectado um movimento mais brusco do que o suportável normalmente pelo disco em funcionamento.

### 6.3.3 PERIFÉRICOS GRÁFICOS

Longe vão os dias em que os computadores tinham apenas uma interface de linha (comandos de texto) com o utilizador. Hoje em dia todos os computadores de uso geral têm uma interface gráfica com grandes capacidades em termos de manipulação de janelas, imagens de duas dimensões, vídeo, objectos tridimensionais com texturas, etc., e resoluções de  $1280 \times 1024$  pixels com 32 bits por pixel (24 bits de cor). Tudo isto com capacidade de produzir várias dezenas de imagens completas por segundo.

Os processadores modernos incluem já uma série de instruções optimizadas para manipulação de imagens e outra informação multimédia, mas muito continua a depender da placa gráfica, o periférico responsável pela gestão do ecrã em baixo nível. Os detalhes do funcionamento das placas gráficas e software associado estão fora do âmbito deste livro, sendo aqui referidos apenas os aspectos essenciais:

- Uma interface gráfica é um subsistema completo, incluindo um processador especializado (capaz de muitas operações gráficas, libertando o processador central dessas tarefas), memória de processamento, para uso desse processador (na ordem de dezenas ou mesmo centenas de MBytes), e uma memória de vídeo com capacidade para conter todos os pixels do ecrã (Fig. 6.37);
- Esta memória de vídeo está continuamente a ser lida sequencialmente, pixel a pixel, linha a linha, imagem a imagem, para gerar o sinal de vídeo que é dado ao monitor, que constrói continuamente as imagens, linha a linha;

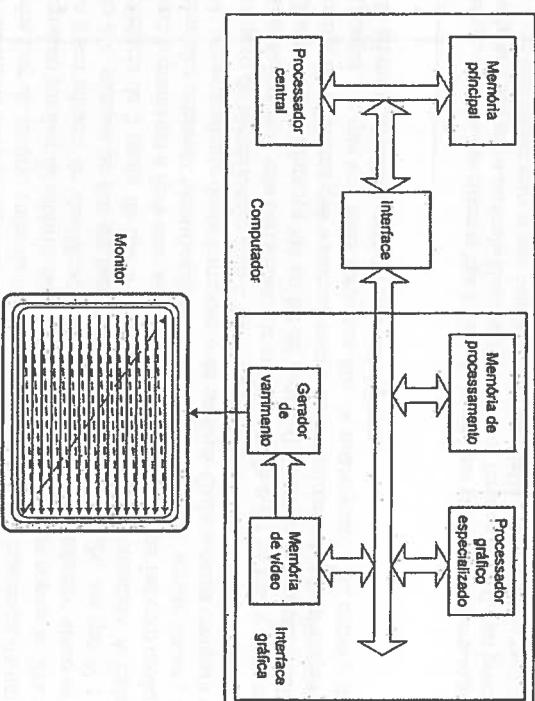


Fig. 6.37 - Estrutura de uma interface gráfica e geração das imagens no monitor

Por outro lado, a memória de vídeo vai sendo actualizada pelo processador gráfico especializado da interface gráfica ou pelo próprio processador central, de acordo com os algoritmos de geração das imagens;

Estas operações envolvem movimentações de dados massivas entre a memória central do computador e a memória de processamento da interface gráfica. Os PCs actuais incluem uma ligação especial do processador central e da sua memória à interface gráfica, que permite que a memória central do computador possa ser usada como se de memória da interface gráfica se tratasse;

O que torna a interface gráfica especial é o facto de geralmente ser o periférico com maiores necessidades de desempenho, pelo que normalmente funciona à velocidade do barramento do processador, enquanto todos os outros (com exceção das interfaces de rede de comunicação mais rápidas) funcionam a velocidades mais baixas, em barramentos específicos (secção 6.4.1).

### 6.3.4 PERIFÉRICOS DE COMUNICAÇÃO

#### 6.3.4.1 PRINCÍPIOS BÁSICOS

Estes periféricos destinam-se essencialmente a permitir a interligação de dados entre sistemas computacionais, incluindo neste conceito dispositivos como impressoras (que têm um computador lá dentro).

Um dos aspectos fundamentais é o facto de, ao contrário do que se passa dentro do universo de um só computador, não ser geralmente possível assumir que há apenas um mestre incontestado e um só sinal de relógio. Os sistemas em comunicação são independentes, autónomos e muitas vezes com direitos iguais (o que quer dizer que não se pode identificar um mestre).

A ligação pode ser ponto-a-ponto, com apenas dois intervenientes, ou em barramento, em que vários sistemas estão interligados e partilham os mesmos sinais. Sempre que mais do que um sistema puder tomar a iniciativa (até apenas com dois, numa ligação bidireccional) há que resolver o problema do árbitrio, isto é, quem é que em cada altura tem direito a colocar os seus dados no barramento. O que não pode suceder é dois ou mais sistemas forçarem um valor no barramento de dados simultaneamente, senão há conflito de dados e os sistemas não conseguem comunicar.

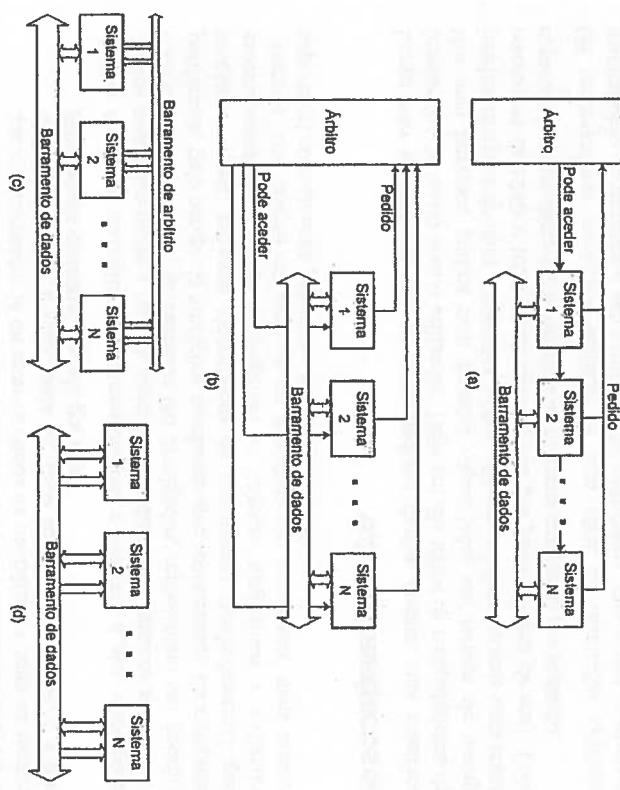


Fig. 6.38 - Principais métodos de árbitrio. (a) - Em cadeia; (b) - Centralizado; (c) - Conflito na identificação; (d) - Colisão nos dados

Os métodos fundamentais de árbitrio são os seguintes, ilustrados pela Fig. 6.38:

- **Centralizado** – Assume a existência de um circuito que desempenha o papel de um árbitro que centraliza os pedidos e que decide qual dos sistemas que competem pelo acesso ao barramento de dados terá prioridade no acesso;

**Distribuído** – Os vários sistemas entendem-se para tomar uma decisão sem um árbitro central. O truque aqui é deixar que os sistemas entrem em potencial conflito. O árbitro baseia-se depois na recuperação desse conflito. Há dois métodos fundamentais:

- **Conflito na identificação** dos sistemas (Fig. 6.38c) – Este método pressupõe uma fase de árbitro antes da transmissão dos dados e a existência de um barramento auxiliar de árbitro, no qual cada sistema a querer aceder ao bus coloca o seu número de identificação (cada sistema tem um número único atribuído e o número de bits do barramento de árbitro é  $\log_2 N$ , em que  $N$  é uma potência de 2 maior do que o número de sistemas concorrentes). A ligação a esse barramento é feita com saídas *open-collector*,<sup>77</sup> o que faz com qualquer 0 num dos números identificadores dos sistemas a querer aceder forceem a 0 o bit correspondente desse barramento de árbitro. Cada sistema compara o seu número de identificação com o valor no barramento de árbitro e descarta as suas saídas para este barramento de árbitro desde o bit de maior peso em que detectar uma diferença até ao bit de ordem 0. Este procedimento é feito por todos, de tal forma que o sistema com o menor número de identificação vence (porque é o que tem mais zeros nos bits de maior peso). Tal como o método de árbitro em cadeia, este método não é justo;

Para este efeito, cada um dos sistemas liga à linha de pedido por meio de uma porta lógica com saída *open-collector*, que tem a particularidade de apenas poder forçar o valor 0. Isto permite que várias saídas deste tipo liguem entre si sem conflito. Basta uma delas forçar 0 para o valor da linha 0. A linha só será 1 se nenhuma das saídas quiser forçar 0 (caso em que não haverá pedidos de acesso ao barramento).

**Em cadeia** (Fig. 6.38a) – Os vários sistemas fazem o seu pedido de acesso ao barramento numa única linha,<sup>77</sup> que o árbitro recebe. O árbitro pode detectar que há um pedido de acesso mas não sabe de qual sistema veio. Indica a permissão de acesso a um dos sistemas, aquele a que está fisicamente ligado. Se ele tiver feito um pedido, acede. Caso contrário, indica ao sistema seguinte que ele pode aceder, efectuando este igual procedimento, até chegar ao próximo sistema que tiver feito um pedido. O principal problema deste método é não ser justo. Os sistemas mais perto do árbitro têm obviamente prioridade;

**Paralelo** (Fig. 6.38b) – Cada um dos sistemas concorrentes pelo acesso ao barramento envia um sinal de pedido individual a um circuito que desempenha o papel de um árbitro central. Este decide, de acordo com algum algoritmo, e selecciona um dos sistemas, por notificação com sinais individuais. O problema deste método é que o árbitro é um recurso crítico, que se pode revelar um estrangulamento em termos de desempenho e fiabilidade. Funciona bem com um número de sistemas limitados e barramentos muito curtos. O PCI, um barramento paralelo normalizado muito usado nos PCs, usa este método.

**Conflito nos dados (Fig. 6.38d)** – Este método dispensa a fase de arbitrio.

Um sistema que queria enviar dados para o barramento começa por esperar pelo fim do acesso que algum sistema esteja a fazer. Quando tal acontece acede ao barramento, sem verificar se algum dos outros sistemas também fazem o mesmo. No entanto, tem o cuidado de ir lendo do barramento os valores que estão a passar. Caso detecte alguma diferença face ao que para lá enviou, porque houve uma colisão. Outro sistema tentou também enviar os seus dados, com outros valores, e os valores resultantes destes conflitos serão diferentes dos que os sistemas para lá enviaram. Quando isto sucede, ambos os sistemas desistem, esperam um tempo aleatório, espearam pelo fim de qualquer acesso em curso e tentam de novo. Este tempo aleatório resolve o conflito (algum há-de aceder ao barramento primeiro) e garante a justezza do arbitrio (nenhum sistema tem prioridade face a outros). Este é o método usado pelas redes locais de computadores (protocolo *Ethernet*<sup>18</sup>).

Outra característica típica da comunicação é a ausência de um barramento de endereços, uma vez que cada sistema tem as suas características próprias e não se pode definir um espaço global de endereçamento. É no barramento de dados que tem de se enviar informação sobre a quem se destinam os dados. Isto implica definir um protocolo de comunicação em que se estabeleça qual o formato da informação, que é enviada não byte a byte mas sim em pacotes, que englobam bytes de dados e bytes de controlo.

### 6.3.4.2 COMUNICAÇÃO PARALELA

A comunicação paralela assume que os dois sistemas se interligam por um barramento de dados de 8, 16 ou mais bits, complementados por sinais de controlo que permitem controlar o fluxo dos dados. Um sistema não envia dados sem ter a informação de que o seu interlocutor está pronto para os receber. É um barramento paralelo assíncrono, em que o sistema mais lento marca o ritmo.

**[Nota]** Os barramentos dos computadores são também uma forma de comunicação paralela entre o processador e os restantes dispositivos (memória e periféricos). Estes são barramentos paralelos síncronos, pois trabalham com um mesmo sinal de relógio.

O funcionamento destes barramentos já foi discutido na secção 6.1.6. Aqui discute-se apenas a perspectiva de comunicação entre sistemas diferentes, em que um não pode assumir nada sobre as temporizações do outro.

A Fig. 6.39 ilustra a comunicação paralela assíncrona, assumindo um sistema A (emissor) e outro B (receptor). Para além do barramento de dados, existem dois sinais, Pedido (gerado pelo emissor) e Confirmação (gerado pelo receptor).

<sup>18</sup> Não confundir com *Internet*, um conjunto de infra-estruturas e protocolos que asseguram actualmente as comunicações informáticas globais no planeta.

O protocolo de comunicação de um valor do emissor para o receptor pode ser descrito em quatro fases, assumindo que inicialmente os dois sinais de controlo estão inactivos (a 0):

1. O emissor coloca o valor a enviar no barramento de dados e activa o seu sinal Pedido (de envio);
2. Logo que possa, o receptor memoriza esse valor e activa o sinal Confirmação, assinalando assim que já recebeu o valor;
3. O emissor assinala que recebeu a confirmação do receptor, desactivando o sinal Pedido e libertando o barramento de dados;
4. O receptor desactiva o sinal Confirmação para assinalar que (i) já processou o valor e está pronto a receber outro e (ii) que notou que o emissor recebeu a sua confirmação.

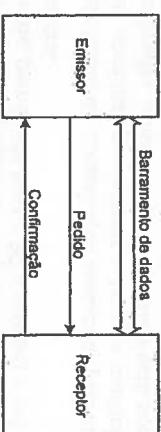


Fig. 6.39 - Comunicação paralela assíncrona. (a) – Ligação entre emissor e receptor; (b) – Sequência temporal dos sinais

Este protocolo designa-se *aperto-de-mão* (*handshaking*), dada a existência dos dois sinais de controlo complementares, como os dois braços se tratassesem.

**[Nota]** Normalmente, o emissor é que decide quando tem algo para enviar, mas nada impede que se inverta a semântica do protocolo e seja o receptor a gerar o sinal Pedido (que agora indica que o receptor está pronto para receber valores e está a pedi-los), e o emissor a gerar o sinal Confirmação (confirmar o envio de um valor). A sequência dos sinais é semelhante à Fig. 6.39b, mas em que o barramento de dados está activo durante toda a duração do sinal Confirmação.

Para comunicar na direcção oposta (suportando comunicação bidireccional), o barramento de dados em cada sistema deve ser bidireccional com interface *tristate* (o que aliás

já é aparente na Fig. 6.39b) e a função relativa dos dois sinais de controlo deve ser invertida. Naturalmente, assume-se que haverá uma forma de cada sistema saber em que papel se encontra (ou por um sinal adicional ou pela sequência dos valores enviados).

#### 6.3.4.3 COMUNICAÇÕES SÉRIE

A comunicação paralela só funciona em distâncias curtas (poucos metros) e com temporizações de relativa baixa frequência, devido à interacção entre os vários sinais. Para distâncias maiores ou frequências mais elevadas tem de se usar comunicação série, em que se transmite apenas um bit de cada vez.

O cabo de comunicação precisa apenas de dois condutores, embora na prática os cabos incluam normalmente dois pares de condutores, para permitir comunicação *full-duplex* (nos dois sentidos simultaneamente). A comunicação *simplex* (apenas num sentido) é mais simples mas não permite interacção e poderá ser útil apenas em aplicações específicas, como por exemplo telemetria (leitura remota de sensores). A comunicação *half-duplex* consiste em partilhar um mesmo par de condutores para permitir comunicação nos dois sentidos, mas apenas um de cada vez. Poupa-se no cabo, mas complica-se no emissor e no receptor. Com a evolução da tecnologia e das necessidades de comunicação, a comunicação *full-duplex* é a mais usada.

Na comunicação série, os bytes são serializados no emissor, isto é, transmite-se um bit de cada vez. Cada byte a transmitir é armazenado num registo que depois (em cada ciclo de relógio) é sucessivamente deslocado de um bit para a direita (começando a transmissão pelo bit de menor peso) ou para a esquerda (começando a transmissão pelo bit de maior peso). Em cada ciclo de relógio, o bit transmitido é o presente no extremo (direito ou esquerdo, respectivamente) desse registo. Para transmitir os 8 bits de um byte são precisos 8 ciclos de relógio. Do lado do receptor faz-se a operação inversa e ao fim de 8 ciclos de relógio tem-se o byte recuperado.

Para além dos bits dos dados propriamente ditos, é preciso incluir bits adicionais para sincronização entre emissor e receptor ou enviar outra informação de controlo. Estes bits também consomem tempo de comunicação e como tal constituem um elemento de ineficiência (*overhead*) indesejável mas necessário, que naturalmente se tenta minimizar.

O mundo das redes de comunicação e dos seus protocolos é na realidade um universo, completamente fora do âmbito deste livro. Esta secção resume-se aos níveis mais básicos de dois exemplos concretos, muito usados em computadores pessoais:

- Norma RS-232C (usada na porta série dos PCs);
- Barramento USB (*Universal Serial Bus*), usado para ligar a periféricos [USB].

Embora não seja descrita aqui, a norma IEEE 1394, mais conhecida por *FireWire*, é também referida pela Tabela 6.17. Originalmente desenvolvida pela Apple, equipa os computadores desta empresa e, dada a sua elevada taxa de transmissão e o facto de as interfaces possuirem igual estatuto e poderem comunicar entre si, sem necessidade de um computador pessoal, é muito usada no mercado do consumo, em particular para vídeo

digital, enquanto o USB é mais usado no mercado de computadores pessoais, para dados. Em termos de implementação, tem bastantes semelhanças com o USB.

| Protocolo                 | APARECEU EM | RITMO DE TRANSMISSÃO | TIPO DE TRANSMISSÃO | LIGAÇÃO       |
|---------------------------|-------------|----------------------|---------------------|---------------|
| RS-232C                   | 1962        | 192 Kbytes           | Assíncrona          | Ponto-a-ponto |
| FireWire 400 (IEEE 1394)  | 1995        | 400 Mbytes           | Síncrona            | Barramento    |
| USB 1.0                   | 1996        | 1,5 Mbytes           | Síncrona            | Barramento    |
| USB 1.1                   | 1998        | 12 Mbytes            | Síncrona            | Barramento    |
| USB 2.0                   | 2000        | 480 Mbytes           | Síncrona            | Barramento    |
| FireWire 800 (IEEE 1394b) | 2001        | 800 Mbytes           | Síncrona            | Barramento    |

Tabela 6.17 - Comparação de algumas características de dois protocolos de comunicação série: RS232 e USB, nas suas várias versões

#### COMUNICAÇÃO SÉRIE ASSÍNCRONA COM A NORMA RS-232C

A norma RS-232C é um dos protocolos de comunicação mais antigos (20 anos anterior aos PCs) e está orientado para a interligação ponto-a-ponto de dois dispositivos, encarando cada byte como independente dos restantes, e sem restrições de tempo entre dois bytes transmitidos em sequência (de forma assíncrona entre eles, portanto). Era o protocolo usado pelos terminais de vídeo remotos ligados por modems aos computadores nos centros de cálculo na era pré-PC. Quando os computadores pessoais apareceram, foi adoptada para uso na porta série, permitindo ligar o PC a modems e usá-lo como terminal remoto de um computador central.

A Fig. 6.40 ilustra o princípio básico da comunicação assíncrona pela norma RS-232C. O emissor e o receptor têm de combinar previamente a taxa de transmissão, medida em bits/s. Há várias taxas normalizadas, sendo a máxima prevista pela norma 19.200 bits/s, caso em que cada bit demora  $1/19.200 = 52.1$  microsegundos a transmitir. Em repouso, o sinal está a 1.

Quando o emissor pretende enviar um byte (Fig. 6.40a), começa por enviar um bit a 0 (bit inicial, ou *start bit*), depois os 8 bits do byte (começando pelo de menor peso), seguidos de um bit de paridade (opcional, usado para detecção de erros) e de dois bits de guarda (*stop bits*), que se destinam a garantir um período mínimo com o sinal a 1 entre bytes, mesmo que estes sejam transmitidos continuamente.

**NOTA** A norma original era orientada ao carácter de texto, em codificação ASCII de 7 bits, pelo que nem sequer contemplava a emissão de 8 bits de dados (apenas 5, 6 ou 7). No entanto, esta era uma limitação grande e os sistemas passaram a permitir os 8 bits.

O bit de paridade destina-se a detectar um número ímpar de bits trocados (por ruído na transmissão). A norma permite escolher paridade par ou ímpar (caso em que o número de bits a 1 no conjunto byte transmitido + bit de paridade deve ser par ou ímpar, respectivamente) ou nem sequer transmitir este bit.

O tempo dos bits de guarda também é configurável, em 1,  $1 \frac{1}{2}$  ou 2 bits.

As taxas de transmissão originalmente normalizadas eram 75, 150, 300, 600, 1200, 2400, 4800, 9600 e 19200 bits/s. Embora tenham aparecido normas subsequentes que melhoraram em muito estes valores, nada se compara às taxas de transmissão atingidas pelo USB.

Tudo isto tem de ser configurado de igual modo no emissor e no receptor, sob pena de o receptor não entender correctamente os bytes enviados pelo emissor.

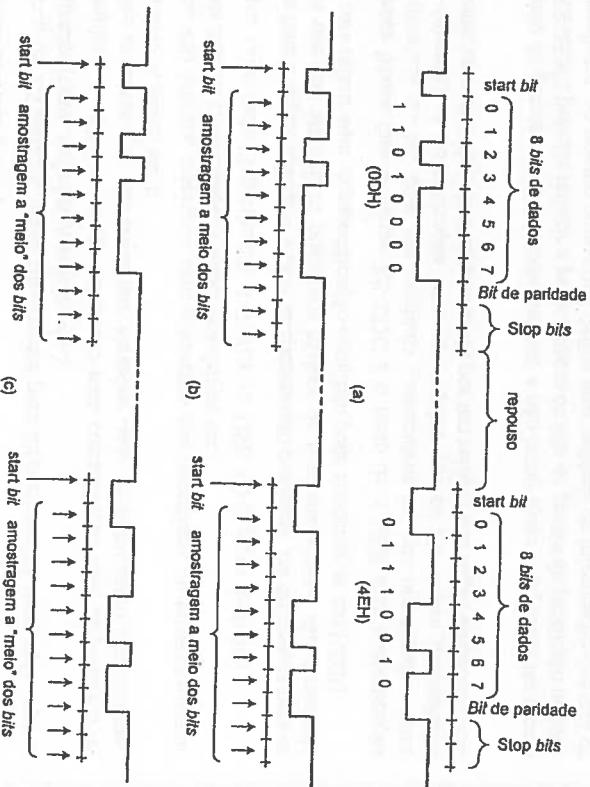


Fig. 6.40 - Comunicação série assíncrona. (a) - Emissor (o tempo de repouso pode variar entre 0 e infinito); (b) - Recuperação dos bytes no receptor por amostragem do sinal;

(c) - Receptor com frequência de receção ligeiramente inferior à frequência de emissão

Não há qualquer limitação ao tempo entre o envio de dois bytes consecutivos (tempo de repouso, na Fig. 6.40a). Pode variar entre zero e infinito. É esta variabilidade que lhe confere a designação de comunicação assíncrona, pois dentro de cada byte os bits são transmitidos com ritmo constante, à taxa de transmissão combinada. Nesta figura, são transmitidos dois bytes 0DH e 4EH. O receptor (Fig. 6.40b) está continuamente a monitorizar o sinal até detectar uma transição para zero, altura que marca como inicio de cada byte. Sabendo a taxa de transmissão, vai depois amostrar o sinal aproximadamente a meio longo de um bit, para conseguir eliminar potenciais picos causados por ruído.

Depois de o byte ser totalmente recebido, o receptor compara o bit de paridade com a paridade do que foi recebido. Se for diferente do esperado, ocorre um erro (compete aos níveis superiores do protocolo usado recuperar da situação – pedir uma retransmissão, por exemplo). Note-se que a detecção de erros não é garantida. Se em vez de um bit estar errado estiverem dois, a paridade volta a estar correcta e não há detecção de erro.

**NOTA** É usual complementar esta detecção de baixo nível com outras mais elaboradas, como somar todos os bytes transmitidos e transmitir essa soma ( vulgarmente designada checksum) no final. O receptor soma todos os bytes recebidos e compara com a soma recebida. Se houver alguma diferença, houve erro. A probabilidade de detecção é muito mais elevada do que no caso da paridade. Esta soma é apenas um caso particular de um mecanismo mais geral, o CRC (Cyclic Redundancy Code), em que a função a aplicar sucessivamente aos vários bytes é mais complexa do que a soma (envolve geralmente polinómios), mas o princípio é o mesmo.

Apesar de o emissor e o receptor combinarem previamente a taxa de transmissão, não há garantia de que as frequências medidas de um lado e de outro sejam rigorosamente iguais. Haverá sempre uma pequena diferença, que se ilustra na Fig. 6.40c, em que o receptor tem a sua frequência ligeiramente mais lenta do que a do emissor, caso em que se vai atrasando na amostragem dos bits.

Desde que a diferença não seja muito grande, não há qualquer problema. Quando se chega ao final do byte, os bits de guarda e o bit de início do byte seguinte garantem que o receptor fica novamente sincronizado (tal como representado na Fig. 6.40c). Se a diferença for tal que o receptor amostre o sinal de entrada já fora dos bits, o protocolo deixa de funcionar correctamente.

A Simulação B.1, na página 774, apresenta um exemplo de utilização da norma RS-232C. É importante reconhecer que o número de bytes de dados que se consegue transmitir por segundo não é a taxa de transmissão a dividir por 8. Por cada 8 bits de dados enviados, têm de ir também outros 4 bits (start bit, bit de paridade e 2 stop bits). Isto representa uma sobrecarga do protocolo de 50%. Por exemplo, 19.200 bits/s equivale, no máximo, a 1600 bytes/s ou 12.800 bits/s = 19.200 bits/s / 1.5.

**NOTA** Associada à taxa de transmissão aparece normalmente a unidade baud<sup>19</sup>, que indica o número de símbolos enviados por segundo. Na norma RS-232C, baud equivale a bits, porque cada símbolo é um bit. No entanto, os esquemas de codificação de bits transmitidos sobre canais analógicos (caso dos modems, por exemplo) permitem codificar mais do que um bit em cada símbolo transmitido, pelo que a unidade bits é mais adequada para descrever a quantidade de informação enviada por unidade de tempo.

<sup>19</sup> Termo derivado do nome do engenheiro francês Jean-Maurice-Émile Baudot (1845-1903), que dedicou a vida ao aperfeiçoamento do telegrafo.

## COMUNICAÇÃO SÉRIE SÍNCRONA COM O BARRAMENTO USB

O barramento série universal (USB) apareceu em 1996, essencialmente como resposta às dificuldades de instalar periféricos num PC (até então, a forma de estender as capacidades de um PC era abri-lo, instalar uma placa do periférico pretendido e reconfigurar os vários periféricos existentes, para poderem coexistir sem conflitos).

O USB veio permitir uma maior taxa de comunicação do PC com os periféricos (dispositivos USB) e ligá-los com o PC em funcionamento, introduzindo a configuração automática (*plug and play*). Outra característica interessante é o facto de o USB contemplar a possibilidade de alimentação (a 5 V) dos dispositivos. Em muitos casos, basta ligar o dispositivo ao USB, mas dispositivos que requeriam mais potência terão de ter a sua própria alimentação.

Há quatro taxas de transmissão possíveis:

- Baixa, de 1,5 Mbit/s, usada tipicamente para dispositivos de interacção com o utilizador (ratos, teclados, joysticks, etc.);
- Completa, 12 Mbit/s, usada sobretudo para comunicação com dispositivos de dados massivos que não requerem elevadas taxas de transmissão (impressoras, *pen-drives*, áudio, etc.);
- Alta, 480 Mbit/s, reservada para aplicações que necessitem ou possam usufruir desta taxa de comunicação (vídeo, *pen-drives*, etc.);
- Super velocidade (SuperSpeed), 5 Gbit/s, no USB 3.0 (a partir de 2010).

A descrição feita a seguir aplica-se à taxa de transmissão completa. Há diferenças para as outras taxas que não alteram os princípios básicos de funcionamento e são demasiado detalhadas para referir aqui. A especificação completa pode encontrar-se em [USB].

Uma diferença básica face à norma RS-232C é o facto de o USB usar comunicação síncrona orientada ao bit, com um conjunto consecutivo de bits organizados numa mensagem (pacote). A comunicação síncrona implica que os bits sejam transmitidos continuamente, com um ritmo fixo determinado por um relógio que segue enbebido com os dados.

A transmissão de pacotes com um dado formato e não como bytes independentes (como na norma RS-232C) permite reduzir a percentagem de bits de gestão do protocolo no total de bits transmitidos, ao mesmo tempo que define uma unidade de informação que terá de ser retransmitida em caso de erro.

Cada um dos pacotes tem um formato específico, que permite ao hardware detectar o seu princípio e fim (Fig. 6.41).

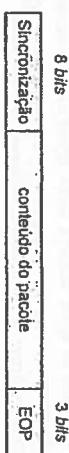


Fig. 6.41 - Envolvente (início e fim) de cada pacote, com indicação do número de bits gastos antes e depois do pacote propriamente dito

Ao contrário do RS-232C, os dados no USB são transmitidos diferencialmente<sup>80</sup>, o que permite maiores frequências de trabalho, com dois condutores designados D+ e D-. Um 1 verifica-se quando D+ > D- e um 0 quando D+ < D- (por D+ e D- entendê-se aqui as tensões eléctricas em cada condutor). Definem-se ainda os estados de repouso (nem o dispositivo nem o concentrador forcam algum valor nos condutores) e de tudo-0, em que ambos condutores estão na tensão mínima possível. A tensão intermédia em torno da qual D+ e D- giram não é zero, tendo de estar entre 1,3 e 2,0 V.

O papel dos Os e Is inverte-se entre as velocidades baixa e completa, pelo que a sinalização se faz em termos de estados do cabo USB designados J e K. Na velocidade completa, OJ corresponde ao 1 e OI ao 0. A sequência que define o fim de um pacote e o início do próximo é a seguinte:

- Após o envio de todos os bits de conteúdo de um pacote, o cabo USB é colocado no estado tudo-0 durante dois bits, seguido de um bit J, após o que o dispositivo ou concentrador que acabou de enviar o pacote se desliga do cabo (que fica em repouso);
- O estado de repouso tem de durar pelo menos um bit, após o qual se pode começar a enviar um novo pacote (pelo mesmo dispositivo/concentrador ou outro). O intervalo entre pacotes da mesma transacção (Fig. 6.46) tem de ser no máximo na ordem dos 7 bits. Não há limite para o intervalo entre pacotes de transacções diferentes (embora o protocolo garanta que é enviado pelo menos um pacote em cada milissegundo, para não se perder o sincronismo);
- Um pacote começa pela transição do estado de repouso para o estado K, o que constitui o primeiro bit de uma sequência de 8 bits que serve de byte inicial de pacote, com o objectivo de sincronização. Esse byte tem o valor binário 0000 0001 (sete bits a 0 seguidos de um bit a 1), a que se seguem de imediato os bits do conteúdo do pacote.

Há aqui algum paralelismo com os bits de guarda (*stop bits*) e o bit de inicio (*start bit*) do RS-232C. As grandes diferenças são o tamanho do "byte" transmitido no USB, que é um pacote inteiro, com milhares de bytes, e o facto de o receptor nunca perder o sincronismo, mesmo entre pacotes.

Na realidade, o que é enviado para o cabo USB não é rigorosamente o byte de sincronização e os bits do pacote. O problema é que cada pacote pode ter mais de 8000 bits seguidos (Fig. 6.45b), sem informação de relógio entre emissor e receptor, o que pode causar grandes problemas se houver algum (mesmo que pequeno) desfasamento entre as frequências de trabalho do emissor e do receptor. Este problema já tinha sido identificado na norma RS-232C (Fig. 6.40c), mas aí estavam em causa apenas 8 bits.

O USB usa comunicação síncrona (bits todos seguidos), mas tem de incluir alguma informação sobre o relógio. Para tal, usa as transições do sinal (que ocorrem quando o

valor dos bits a transmitir muda) para sincronizar o relógio e utiliza um esquema que garante um limite máximo de tempo sem transições de modo a não deixar os relógios do emissor e do receptor perderem o sincronismo.

Este esquema contempla duas técnicas básicas:

- Codificação da sequência de bits a enviar com o método NRZI (*Non Return to Zero Invert*, ou Não Retorno a Zero Invertido), em que um 0 à entrada faz mudar o estado à saída e um 1 à entrada faz manter o estado. Assim, sempre que a saída do NRZI muda, sabemos que o bit à entrada mudou para um 0 (independentemente do bit anterior), e esta mudança identifica uma transição do relógio. O problema é que uma longa sequência de 1s à entrada produz um sinal constante a saída, o que naturalmente não é desejável;
- Enchimento de bits a zero. Por cada seis 1s consecutivos à entrada, insere-se um 0. Desta forma, garante-se que nunca há mais de seis bits a 1 na sequência de saída, embora à custa de transmitir mais uns bits pelo meio, e o receptor continua com informação sobre o relógio, no máximo de sete bits. O receptor que recebe a sequência de bits NRZI deve analisar o sétimo bit após uma sequência de seis bits a 1. Se for 0, simplesmente descarta-o. Se for 1, há um erro e o pacote terá de ser retransmitido.

A Fig. 6.42 ilustra estas duas técnicas. O enchimento faz-se a partir da transição para 1 do padrão de 8 bits de sincronização.

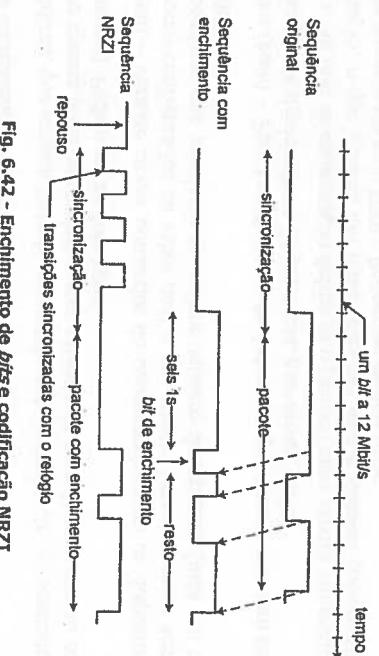


Fig. 6.42 - Enchimento de bits e codificação NRZI

Note-se que o barramento USB é lógico e não físico. Cada dispositivo USB liga a uma interface individual (porto) de um concentrador (hub) USB, que faz a gestão dos vários dispositivos, permitindo ao processador central comunicar com qualquer deles e fazendo com que na prática pareça que estão todos ligados no mesmo cabo, como se constituíssem um barramento real.

Um PC tem incluído um concentrador raiz, com um controlador que liga ao processador (tal como os outros periféricos internos) e é responsável por gerir a configuração.

automática e dinâmica dos dispositivos USB. Esse concentrador raiz disponibiliza portas USB, onde podem ligar dispositivos USB ou concentradores USB externos, que são essencialmente repetidores dos sinais e permitem adicionar portas USB ao sistema (Fig. 6.43).

O concentrador raiz pode controlar até 127 dispositivos, cada um com um número único, entre 1 e 127. O número 0 é reservado para permitir a ligação de um novo dispositivo. Qualquer dispositivo assume ser o número 0 quando é ligado ao barramento, situação que é detectada pelo hardware. Quando tal acontece, o concentrador raiz comunica com o dispositivo 0, pede-lhe a sua descrição e atribui-lhe um novo número, ficando o número 0 livre de novo.

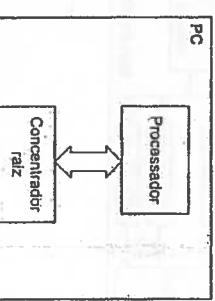


Fig. 6.43 - Topologia física do USB

Não há comunicação entre dois dispositivos. Cada um comunica apenas com o concentrador raiz, estabelecendo com este canais virtuais chamados condutas (*pipes*). Cada dispositivo pode incluir diversas funcionalidades, a que correspondem terminais (*endpoints*) lógicos. Uma conduta estabelece-se entre um terminal e o software que controla o concentrador raiz.

Em cada dispositivo há 16 números (0 a 15) para definir terminais, mas a cada número correspondem dois terminais, um em cada direcção da informação. (concentrador → dispositivo e vice-versa). Todos os dispositivos têm de ter pelo menos os dois terminais com o número 0, pois é através dele que o concentrador estabelece as duas condutas básicas para comunicação bidireccional com o dispositivo. Poderá ter mais, dependendo das suas características. As condutas correspondentes são estabelecidas quando o dispositivo é ligado ao barramento, fase essa denominada enumeração. Há dois tipos de condutas:

- **Conduitas de mensagem** – São bidirecionais (usam os dois terminais com um dado número) e em que se define um formato para a transferência de informação. As transferências são exclusivamente iniciadas pelo concentrador;
- **Conduitas de corrente (stream)** – São unidirecionais e não têm um formato de transferência de informação definido. Os dados podem ser enviados por iniciativa do concentrador ou dos dispositivos.
- Há quatro tipos de transferências de dados através das conduitas:
- Não periódicas, que surgem quando necessário, sob iniciativa do concentrador (o dispositivo apenas pode responder) e sem garantia de taxa de transmissão ou de atraso (latência). Podem ser de dois tipos:
  - **Controlo** – Permite enviar comandos do concentrador para os dispositivos, bem como configurá-los ou saber qual o seu estado. Este tipo exige sempre duas conduitas que terminem no mesmo número de terminal (uma em cada direcção);
  - **Massivo (bulk)** – Para transferência de dados em quantidade mas sem restrições temporais significativas (impressoras, pen-drives, etc.).
- Periódicas. Na fase de enumeração define-se o período entre envios consecutivos de informação, o que garante um limite máximo para a latência (isto é, atraso máximo de transmissão pelo USB). Podem ser de dois tipos:
  - **Interrupção** – Não há sinais físicos dos periféricos para o controlador de interrupções do computador, pelo que tem de ser o controlador do concentrador raiz a testar periodicamente os dispositivos para saber se algum tem informação para transmitir. O período pode ser programado entre 1 e 255 milissegundos, sendo tipicamente usado para dispositivos de interacção com o utilizador (ratos, teclados, joysticks, etc.). A informação a transmitir é tipicamente pequena (até 64 bytes);
  - **Isócrono** – Para este tipo de transferência aloca-se uma percentagem da largura de banda do USB, de forma a garantir um determinado ritmo de transmissão sustentado. Isto é conseguido através da definição da quantidade de informação a transferir em cada período e pelo facto de eventuais erros não serem corrigidos com retransmissão de informação. São usadas sobretudo em aplicações de áudio e vídeo, em que garantir o ritmo de transmissão é mais importante do que garantir que todos os dados chegam. Este é o único tipo de transferência de dados que não faz controlo de erros. O período pode ser programado entre 1 e 32.768 milissegundos.

As conduitas de mensagem suportam apenas o tipo de transferência de controlo, enquanto as de corrente suportam apenas os outros três tipos de transferência de dados. Como as características de transferência de uma conduta são estabelecidas na fase de enumeração e ficam fixas subsequentemente, pode falar-se em conduitas de controlo, isócronas, de interrupção e massivas.

As conduitas periódicas não podem ocupar mais de 90% da largura de banda (percentagem de utilização) do USB. Na fase de enumeração, quando um dispositivo se liga ao USB, o concentrador raiz verifica se há largura de banda suficiente para as pretensões do dispositivo. Nos restantes 10% (ou mais, se as conduitas periódicas não usarem tudo a que têm direito), as conduitas de controlo têm prioridade sobre as de dados massivos, que só conseguem transmitir informação quando o canal está livre de tráfego das outras conduitas.

A Fig. 6.44 mostra o diagrama de blocos básico (apenas do ponto de vista lógico) relevante para a comunicação entre concentrador e dispositivos. Cada dispositivo inclui percentagem de utilização das conduutas isócronas. A separação entre conduutas é apenas lógica, baseada na identificação do terminal envolvido (incluída em cada pacote). O concentrador actua como árbitro central, distribuindo o tráfego de acordo com as prioridades.

A Fig. 6.44 mostra o diagrama de blocos básico (apenas do ponto de vista lógico) relevante para a comunicação entre concentrador e dispositivos. Cada dispositivo inclui percentagem de utilização das conduutas isócronas. A separação entre conduutas é apenas lógica, baseada na identificação do terminal envolvido (incluída em cada pacote). O concentrador actua como árbitro central, distribuindo o tráfego de acordo com as prioridades.

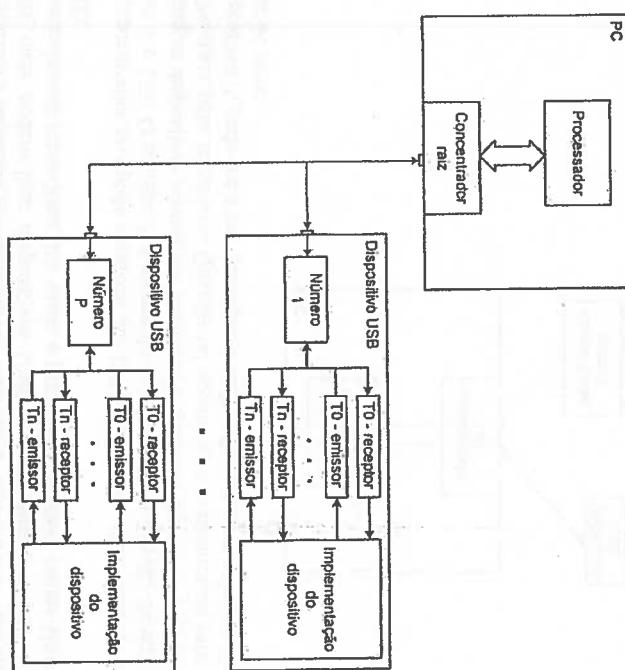


Fig. 6.44 - Topologia lógica do USB

Os terminais são os pontos de ligação ao dispositivo das várias condutas que o concentrador estabelece com ele e podem ser do tipo controlo, masivo, interrupção ou isócrono (que determina o tipo da conduta). Cada terminal inclui uma zona de memória suficiente para guardar o maior pacote que pode circular na sua conduta (valor este que pode ser configurado na fase da enumeração).

Por exemplo, uma câmara de vídeo poderá ter, para além dos dois terminais com número 0, apenas mais dois terminais de emissão, por exemplo (para transmitir o sinal de vídeo num e o de áudio outro), que neste caso serão numerados 1 e 2. Este dispositivo terá três condutas: uma de controlo (tipo mensagem, bidireccional), que usará os dois terminais com o número 0, e duas isócronas (tipo corrente, unidireccionais), que ligarão os terminais (emissores) com os números 1 e 2.

Seja qual for o terminal ou tipo de conduta utilizado, cada comunicação (designada transacção) é sempre iniciada pelo concentrador e envolve tipicamente três pacotes:

- **Símbolo (token)** – Indicam que tipo de operação o concentrador pretende. Este primeiro pacote é gerado pelo concentrador e pode ser um destes:
  - IN (o concentrador está a pedir dados);
  - OUT (o concentrador vai enviar dados);
  - SETUP (o concentrador vai iniciar uma operação de controlo do dispositivo);
  - SOF (*Start Of Frame*, ou início de quadro, que é explicado em baixo).
- **Dados** – Este segundo pacote inclui os dados a transferir. O número de bytes definido na fase de enumeração. Se o pacote anterior for de IN, este pacote é gerado pelo dispositivo, caso contrário é gerado pelo concentrador;
- **Aperto-de-mão (handshake)** – Este pacote é gerado<sup>81</sup> pelo elemento que não gerou o pacote anterior e pode ser de um de três tipos:
  - ACK (Acknowledge) – Dados bem recebidos;
  - NAK (Not Acknowledge) – O dispositivo não está preparado para aceitar os dados enviados por um OUT ou não tem dados para responder a um IN. O protocolo repetirá esta transacção. O concentrador nunca pode gerar um NAK;
  - STALL (empatar) – O dispositivo não pode efectuar esta comunicação, por não suportar o pedido efectuado ou por ter sido colocado em estado de halt (parado) devido a um erro na comunicação.

A Fig. 6.45 mostra qual o formato dos pacotes mais relevantes do protocolo do USB. Qualquer pacote começa pelo PID (Packet ID, ou identificador de pacote), que pode implementar controlo de fluxo ou de erros.

tomar um de 16 valores (4 bits), e do qual depende a interpretação do resto do pacote, cujo formato varia consoante o valor do PID. Para aumentar a probabilidade de detecção de um erro (uma vez que tudo o que vem a seguir depende da correcta transmissão do PID), este campo tem 8 bits, 4 com o valor real e os restantes com os primeiros 4 negados.

A detecção de erros no resto dos pacotes (após o PID) é feita com um campo de CRC (Cyclic Redundancy Check, ou Verificação de Redundância Cicólica), em que cada bit enviado é passado por uma função (envolvendo polinómios) que vai actualizando um valor (de 5 ou 16 bits, CRC5 ou CRC16). O mesmo é feito no receptor, ao receber cada um dos bits. Após receber todo o pacote, compara o valor produzido pela função com o valor CRC recebido. Se for diferente, houve erro. Nos pacotes de dados (Fig. 6.45b) usam-se 16 bits em vez de 5 porque estes pacotes são normalmente muito maiores, pelo que se tem de usar mais bits no CRC para manter a probabilidade de detecção de erros. As funções usadas garantem que todos os erros de um ou dois bits são detectados. Se houver mais bits errados ao longo do pacote, há uma certa probabilidade de o erro não ser detectado.

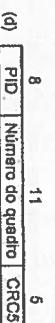
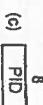
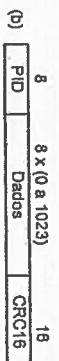
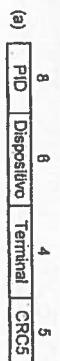


Fig. 6.45 - Formatos dos pacotes mais relevantes do protocolo do USB.  
Cada campo tem indicado em cima o número de bits que ocupa.

(a) – Símbolo; (b) – Dados; (c) – Aperto-de-mão; (d) – Início de quadro

O primeiro pacote (Fig. 6.45a, símbolo) de qualquer transacção entre o concentrador e um dispositivo, efectuada numa dada conduta, tem de especificar (i) o número do dispositivo envolvido (dispositivo), (ii) o número do terminal correspondente a essa conduta (Terminal) e, se o terminal não for de controlo, (iii) qual dos terminais (emissor ou receptor) vai ser usado na transmissão dos dados propriamente ditos (Fig. 6.45b), o que é indicado pelo valor do PID (emissor se PID=IN ou receptor se PID=OUT). Se PID=SETUP, a comunicação é bidireccional e usa os dois terminais.

O segundo pacote (Fig. 6.45b) é o que transmite os dados, com uma dimensão variável, dependendo dos dados que houver para transmitir e de um valor máximo, definido quando a conduta é estabelecida, mas que não pode ultrapassar 1023 bytes (se houver mais dados para transmitir, têm de ser divididos por várias transacções). O primeiro pacote é sempre gerado pelo concentrador, mas este segundo é gerado pelo dispositivo se o primeiro for IN.

<sup>81</sup> Excepto nas transacções das condutas isócronas, que não têm este terceiro pacote porque não implementam controlo de fluxo ou de erros.

As zonas de memória em cada terminal são fundamentais. Se os dados forem enviados a partir do concentrador, o terminal receptor memoriza o conteúdo do pacote de dados até que o dispositivo o possa ler. Se o dispositivo tiver dados para enviar pelo controlador, armazena-os na zona de memória do terminal de emissão. Quando o controlador decidir, há-de enviar um pacote IN para esse terminal, que envia então os dados como resposta.

O terceiro pacote indica apenas se os dados foram bem recebidos e aceites, pelo que este pacote é apenas constituído pelo PID (Fig. 6.45c), sendo gerado pelo dispositivo ou pelo concentrador (quem receber os dados). Se houver um erro, a transacção será repetida. As transacções de dados nas condutas isocronas só têm os dois primeiros pacotes, pois não há controlo de erros nem de fluxo.

O USB divide o tempo em fatias precisas de 1 milissegundo. Em cada  $1 \pm 0,0005$  milissegundos, o concentrador raiz difunde (*broadcast*) para todos os dispositivos USB um quadro (*frame*), que pode incluir tantas transacções (cada uma com os pacotes já referidos) couberem neste milissegundo (desde que haja informação para transmitir), mesmo que de várias conutas (o software encarrega-se depois de separar o tráfego). O tempo que sobrar fica desperdiçado (Fig. 6.46a).

O quarto tipo de pacote, SOF (Fig. 6.45d), é usado para indicar o começo de cada quadro. Um quadro vazio (quando não há nada para transmitir) tem pelo menos o pacote SOF, desperdigando o resto do tempo, para manter a sincronização (Fig. 6.46b).

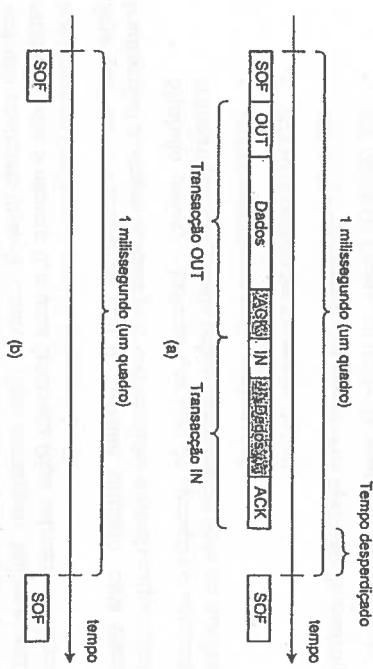


Fig. 6.46 - Exemplos de utilização do tempo do USB. Os pacotes a cinzento são enviados pelo dispositivo e os restantes pelo concentrador; (a) - Quadro com duas transacções, ficando o resto do quadro não usado; (b) - Quadro totalmente vazio

Actualmente o desenvolvimento do USB continua activo, com actividades de que se destacam o USB-On-The-Go, que permite a dispositivos USB comunicarem entre si sem necessitarem de um PC, e o WUSB (Wireless USB), que permite a dispositivos interagirem com o PC mas dispensando o cabo USB. A informação está disponível em [USB].

#### ESSENCIAL:

- De ponto de vista do processador, um periférico é apenas um conjunto de portos que ele pode ler ou escrever. Muitas vezes falamos num periférico como "ndo o conjunto desses portos" e de todo o hardware que eles controlam como "o exemplo um disco é o seu controlador, um teclado ou uma impressora", que
- Pode inclusivamente possuir o seu próprio processador e constituírem si, outro computador embbebido no primeiro. É um abuso de linguagem admissível em termos macroscópicos, mas não nos podemos esquecer de que aquilo a que chamamos "computador" é na realidade um conjunto de vários computadores, mas num nível especializado e integrados por protocolos específicos.

- Os discos (incluindo controladores) e a interface gráfica são exemplos de periféricos complexos que têm evoluído muito. Os discos, apesar de eletrónicos e com tempos de acesso na ordem dos milissegundos, têm alta capacidade e são bastante rápidos. As interfaces gráficas são cada vez mais sofisticadas, em particular com animações 3D, e precisam de cada vez mais capacidade de comunicação com o processador e memória central,

- Há então vários sistemas interligados por um barramento. É preciso decidir (algoritmo) qual o sistema que envia os dados a seguir (só um deles o pode fazer de cada vez). Esse algoritmo pode ser centralizado ou distribuído e ser feito antes da comunicação ou mesmo, após esta começar, com detecção de colisões (é o caso da ethernet, protocolo usado nas redes locais de computadores).
- Alimentação aos periféricos pode ser paralela ou série, síncrona ou assíncrona.
- As ligações paralelas síncronas com um relógio comum usam-se para barramentos internos do computador, sob comando do processador central ou de um controlador do barramento. As ligações paralelas assíncronas são mais comuns: ligações entre dois sistemas independentes em que se usam más de controlo para um sistema saber quando o seu interlocutor já recebeu um dado e está pronto para receber mais um.
- As ligações série assíncronas (RS232) e síncronas (USB, Firewire) permitem ligações maiores do que um barramento paralelo sendo as duas ligadas exteriormente ao sistema do computador.

- O USB em particular é plug & play e tornou a ligação aos periféricos muito flexível, incorporando não apenas transmissão de dados mas também video e som, com garantia de tempo de entrega dos dados. Apesar de ser uma norma complexa, está muito divulgada e constitui o principal meio de interligação dos PCs aos periféricos externos (incluindo câmaras de vídeo, discos magnéticos, telemóveis...), cuja alimentação é geralmente fornecida pelo próprio USB.

## 6.4 ARQUITECTURA DO SISTEMA DE PERIFÉRICOS

### 6.4.1 BARRAMENTOS HIERÁRQUICOS

A secção 6.1 descreveu a interligação dos vários componentes de um computador, tendo como premissa que todos os periféricos ligavam directamente aos barramentos do processador (endereços, dados e controlo) e portanto assumindo que todos conseguiam acompanhar o ritmo do processador. Este inclui um mecanismo para esperar por algum dispositivo mais lento (o sinal de WAIT – ver Fig. 6.22, na página 457). No entanto, enquanto o processador está à espera está completamente bloqueado, pelo que isto só funciona para dispositivos que precisem apenas de um pouco mais tempo de acesso.

Esta é a situação descrita na Fig. 6.47a. No entanto, só funciona nos computadores mais simples. Nos computadores já com algumas capacidades, como um vulgar PC, já não pode ser assim. A continua pressão para maior desempenho dos computadores tem feito a tecnologia de circuitos integrados evoluir bastante, o que se tem traduzido em frequências de relojão dos processadores cada vez mais elevadas, o que torna impraticável a ligação directa de todos os periféricos aos barramentos do processador.

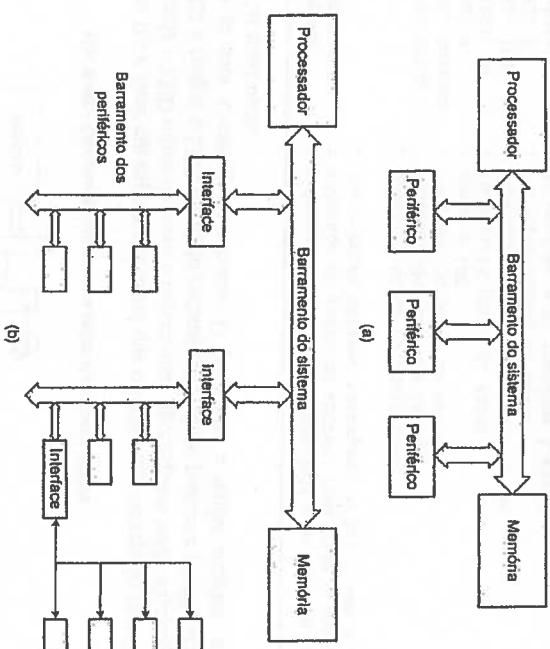


Fig. 6.47 - Arquitectura do sistema de periféricos. (a) - Sistema simples com periféricos ligados directamente ao barramento do sistema; (b) - Sistema de periféricos com barramentos hierárquicos e interfaces para ligar ao barramento de sistema

Para serem rápidos, os barramentos de sistema devem ser curtos e estar optimizados para transferências de e para a memória, com detalhes que variam com a marca e modelo do computador. Seria quase impossível aos fabricantes dos periféricos contemplarem todas

as variantes e acompanhar continuamente as evoluções tecnológicas dos processadores. Por outro lado, um barramento curto não permite espaço para as ligações dos vários periféricos, e se é mais longo tem de ser mais lento.

Assim, rapidamente apareceram barramentos normalizados e mais lentos do que o barramento do sistema, permitindo desacoplar os detalhes do barramento de sistema do funcionamento dos periféricos à custa de interfaces que interligam os dois barramentos. É também possível ter mais do que um nível de barramentos de periféricos, de forma a contemplar periféricos mais lentos (tipicamente mais antigos) ou que usem outro modelo de comunicação (série, por exemplo).

A secção 6.5.2 exemplifica esta arquitectura no âmbito dos PCs.

### 6.4.2 MODOS DE TRANSFERÊNCIA DE DADOS

Um dos aspectos mais fundamentais de qualquer computador é a forma como o processador transfere informação entre a memória e os dispositivos periféricos (disco, rede de comunicação, periféricos de entrada/saída, etc.).

O processador é o mestre incontestado e nada se faz sem a sua autorização. No entanto, o processador tem de ter em conta que um periférico é normalmente um dispositivo lento (pelos seus padrões) e altamente dependente da interacção com dispositivos exteriores. Por outro lado, a interacção com os periféricos não deve gastar uma fração do tempo do processador demasiado elevada (deve mesmo ser a menor possível).

As questões fundamentais neste tema são as seguintes:

- Latência – Como é que o processador sabe quando é que um dado periférico está pronto a transferir dados? Um simples porto de entrada/saída está sempre pronto, mas periféricos como o disco, por exemplo, poderão ter de esperar que o braço seja posicionado e o disco rode até à posição certa, o que pode demorar um tempo imprevisível;
- Taxa de transferência – Qual a forma mais rápida de transferir dados de e para um periférico, tendo o menor impacte na execução das actividades do processador?

#### 6.4.2.1 TRANSFERÊNCIA POR TESTE (POLLING)

A forma mais simples de o processador saber quando um dado periférico está pronto a transferir dados, ou quando um dado bit muda de valor, é testar essa condição continuamente, em ciclo (*polling*). Esta situação é designada espera activa porque o processador fica dedicado a essa espera.

O Programa 6.2 ilustra a espera activa, imaginando um porto de entrada de 8 bits em cujo bit 0 está ligado um botão de pressão e um porto de saída de 8 bits em cujo bit 0 está ligado um LED (Fig. 6.48).

Para serem rápidos, os barramentos de sistema devem ser curtos e estar optimizados para transferências de e para a memória, com detalhes que variam com a marca e modelo do computador. Seria quase impossível aos fabricantes dos periféricos contemplarem todas

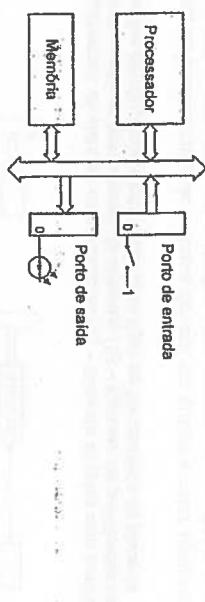


Fig. 6.48 - Circuito simples para teste da espera activa

O que o programa faz é ficar em espera activa até que o botão seja carregado (bit 0 a 1), altura em que acende o LED e fica de novo à espera, mas agora que o botão seja largado, altura em que volta a apagar o LED e o ciclo recomeça. O efeito prático é o LED acender apenas enquanto se está a carregar no botão. O programa é muito simples, mas o processador não faz mais nada.

```

saída EQU 8000H ; endereço do porto de saída (bit 0 a 1)
Entrada EQU 0A000H ; endereço do porto de entrada onde liga o botão
                     ; (se o botão estiver carregado, o bit 0 vem a 1)

PLACE 0000H ; localiza bloco de instruções
início: MOV R2, Saída ; endereço do porto de saída
        MOV R3, Entrada ; endereço do porto de entrada
apaga:  MOV RO, 0 ; apaga o LED
        MOVB [R2], RO ; inicializa porto de saída
ciclo1: MOVB RL, [R3] ; obtém estado do porto de entrada
        BIT RL, 0 ; verifica estado do bit 0
        JZ ciclo1 ; se estiver a 0, continua à espera
acende: MOV RO, 1 ; botão foi carregado. Acende o LED
        MOVB [R2], RO ; põe a 1 o bit 0 do porto de saída
ciclo2: MOVB RL, [R3] ; obtém estado do porto de entrada
        BIT RL, 0 ; verifica estado do bit 0
        JNZ apaga ; se estiver a 1, espera que passe para 0
        ; volta a esperar pelo botão carregado
    
```

Programa 6.2 - Espera activa

Este é um exemplo académico (para ser simples), mas é fácil imaginar outro mais complexo, como por exemplo um sistema em que o processador pretende efectuar uma leitura de um disco. Indica ao controlador do disco qual o sector que pretende ler, e fica em espera activa a ler um bit que o controlador do disco fornece e que há-de mudar de estado quando o controlador do disco tiver lido o sector para a sua zona de memória interna. Nessa altura, o processador poderá ler essa zona de memória do controlador com os dados pretendidos.

O problema é que o acesso ao disco terá demorado vários milissegundos, durante os quais o processador ficou bloqueado em espera activa, sem fazer mais nada, mesmo que houvesse outras tarefas que pudesse executar enquanto o controlador do disco se

encarregava de ler o sector. Felizmente, este é um desperdício de tempo que já não ocorre em qualquer computador moderno.

### SIMULAÇÃO 6.6 - ESPERA ACTIVA

Esta simulação ilustra a espera activa, tornando o Programa 6.2 como base. Os aspectos abertos incluem os seguintes:

- Verificação do comportamento do programa com um botão de pressão;
- Execução do programa passo a passo, com um botão que mantém o estado mesmo quando não carregado (para ser mais fácil testar o comportamento).

### 6.4.2.2 TRANSFERÊNCIA POR INTERRUPÇÕES

A espera activa pode ser resolvida com recurso às interrupções, o meio por exceléncia de um processador não se preocupar com um dado evento mas ser avisado quando ele ocorre. A Fig. 6.49 liga o sinal que vem do botão às interrupções INT0 e INT1, mas com a primeira sensível ao flanco ascendente e a segunda ao flanco descendente. Desta forma, o processador pode ser avisado quando o botão é carregado (gera uma interrupção INT0) e quando é largado (gera uma interrupção INT1).

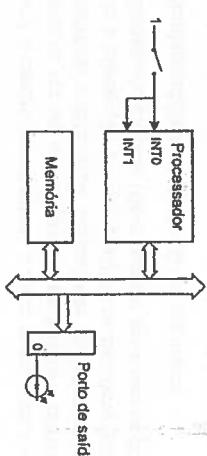


Fig. 6.49 - Circuito simples para teste da espera não activa (por interrupções)

O Programa 6.3 ilustra este circuito. Há uma actividade principal (aqui representada simplesmente por um ciclo infinito que soma uma unidade a R3, mas poderia ser outra actividade mais complexa) que está a ser executada, sem ligar ao botão. Quando este é premido ou largado, as rotinas de interrupção acendem ou apagam o LED, respectivamente. Este programa tem de configurar o INT1 para ser sensível ao flanco descendente do sinal do botão.

Numa aplicação mais séria, a interrupção poderia ser gerada pelo controlador de disco da secção anterior. Assim, o processador faz o pedido ao controlador do disco para ler um seCTOR, por exemplo, e de seguida vai executar outras actividades enquanto o controlador do disco faz a leitura.

Quando os dados estiverem prontos, o controlador activa o pino de interrupção e a respectiva rotina de atendimento pode fazer a leitura desses dados. De seguida, o processador pode voltar às actividades que estava a executar.

```

Saída EQU 8000H ; endereço do porto de saída (bit 0 a 1 = LED aceso)
pilha EQU 2000H ; valor inicial do SP

PLACE 1000H ; localiza Tabela de Excepções
base: word Rot_int0 ; endereço da rotina de atendimento da interrupção
word Rot_int1 ; endereço da rotina de atendimento da interrupção 1

PLACE 0000H ; localiza bloco de instruções
SP, pilha ; inicializa SP
início: MOV SP, pilha ; endereço da rotina de atendimento da interrupção 0
MOV BTE, base ; inicializa apontador para a Tabela de Excepções
MOV R2, Saída ; endereço do porto de saída
MOV R3, 0 ; inicializa contador
MOV R0, 0004H ; mascara com bits 3 e 2 (MSB) com o valor 01
MOV RL, RCN ; (flanco descendente)
OR RL, RO ; vai buscar o conteúdo do RCN
MOV RCN, RL ; programa INTL para o flanco descendente
EIO ; permite interrupções INT0
EI ; a partir de agora pode haver interrupções
ciclo: ADD R3, 1 ; incrementa contador (actividade principal)
JMP ciclo ; continua ciclo principal

;***** - Rotina de atendimento da interrupção INT0.
;***** - Invocada sempre que o botão é carregado (flanco ascendente do sinal).
Rot_int0:
;***** - Rotina de interrupção INT0.
;***** - Invocada sempre que o botão é carregado. Acende o LED.
;***** - Retorna da rotina de interrupção

;***** - Rotina de atendimento da interrupção INT1.
;***** - Invocada sempre que o botão é largado (flanco descendente do sinal).
;***** - Apaga o LED.

Rot_int1:
;***** - Rotina de interrupção INT1.
;***** - Invocada sempre que o botão é largado. Apaga o LED.
;***** - Restaura o R0
;***** - retorna da rotina de interrupção
    
```

Programa 6.3 - Espera não activa (por interrupções)

**SIMULAÇÃO - TRANSFERÊNCIA DE DADOS POR INTERRUPOES**

Esta simulação ilustra a transferência de dados por interrupções, tornando o Programa 6.3 como base. Os aspectos cobertos incluem os seguintes:

- Verificação do comportamento do programa com um botão de pressão;
- Verificação da actividade principal, com execução do programa passo a passo;
- Verificação das interrupções com pontos de paragem nas respectivas rotinas.

```

    
```

```

    
```

```

    
```

```

    
```

As interrupções resolvem o problema da espera activa, mas no exemplo do controlador do disco os dados têm de ser lidos para a memória pelo próprio processador, palavra a palavra, com um ciclo de leitura de uma palavra do controlador seguida de uma escrita na memória, para além da actualização dos endereços da palavra origem e destino (o que pode feito com um registo usado como índice).

O Programa 6.4 ilustra como poderia ser no exemplo do controlador de disco se fosse a rotina de interrupção (invocada por uma interrupção gerada pelo controlador quando o sector já tivesse sido lido) a assegurar a cópia do sector do controlador para a memória.

```

Rot_int0:
    
```

```

    
```

Programa 6.4 - Exemplo de rotina de transferência de dados de um periférico para memória

Desta forma, a transferência de uma palavra gasta uma leitura, uma escrita e as leituras (busca) de cinco instruções da memória (incluindo actualização do índice). Ou seja, sete acessos à memória para transferir uma só palavra, para além do tempo de execução das instruções.

A transferência de dados por software é fácil (é só programar umas instruções) mas é lenta, em particular se há muitos dados a transmitir. Para optimizar, só com suporte em hardware, com acesso directo à memória (DMA - Direct Memory Access).

A Fig. 6.50 ilustra o conceito. A ideia básica é efectuar a transferência por meio de um circuito especializado, em que só a leitura e escrita da palavra em si gastam tempo. Não há instruções para copiar os dados de um lado para o outro (a transferência é feita em hardware), pelo que não há tempo gasto com a busca dessas instruções na memória nem com a sua execução.

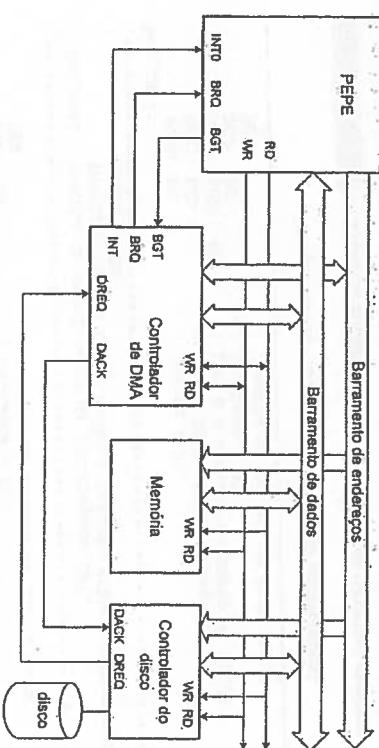


Fig. 6.50 - Transferência de dados por DMA (*Direct Memory Access*)

O controlador de DMA pode comportar-se como:

- Um periférico, com registos internos que o processador pode ler e escrever para configurar uma transferência de dados ou saber informação sobre o estado dessa transferência;
  - Um processador especializado (que só sabe fazer transferências de dados). Nesta qualidade, tem a capacidade de ser ele a gerir os barramentos (endereços, dados e controlo), como se fosse um processador normal a aceder à memória.
  - Neste segundo papel, o controlador de DMA entra em conflito com o processador, habituado a ser o único a gerir os barramentos, pelo que tem de haver um processo de garantir que só um deles em cada instante gere os barramentos. O processador continua a ser o mestre, pois só com autorização deste o controlador de DMA pode entrar em acção. Uma transferência de dados efectuada por DMA envolve tipicamente os seguintes passos (concretizados aqui com o caso do PEPE e um exemplo de uma leitura de disco):
1. O processador programa o controlador de DMA para a transferência, escrevendo nos seus registos internos (usando-o como um periférico qualquer) a informação necessária, nomeadamente o endereço base da zona de dados a transferir, o endereço de base da zona de dados destino e o número de bytes a transferir;
  2. O processador dá o comando ao controlador do disco para ler um dado sector (accedendo-lhe como qualquer periférico, escrevendo nos seus registos internos a informação necessária – número da pista, número do sector, etc.);

3. O processador vai executar outras actividades, “esquecendo-se” desta transferência;
4. Quando o controlador de disco tiver lido o sector para a sua zona de dados interna, activa um sinal (DREQ) para o controlador de DMA, indicando-lhe que já tem os dados prontos para a transferência;
5. O controlador de DMA pede autorização ao processador para começar a transferência, activando o sinal BRQ (Bus Request, ou pedido de acesso aos barramentos);
6. Logo que possa (uma instrução nunca é interrompida a meio), o processador coloca os seus barramentos em estado de alta impedância (tristate) e activa o sinal de BGT (Bus Grant, indicando ao controlador de DMA que tem permissão para gerir os barramentos de acesso à memória e aos periféricos). O PEPE possui um bit (DE) que tem de estar a 1 para permitir o DMA (Tabela A.2, na página 697);
7. O controlador de DMA activa o sinal DACK (DMA Acknowledge), para o controlador de disco saber que a transferência se vai iniciar, ao que o controlador do disco responderá desactivando o sinal DREQ;
8. O controlador de DMA efectua a transferência, lendo uma palavra do endereço origem e escrevendo-a no endereço destino, repetindo depois a operação para a palavra seguinte, até ter transferido toda a informação (de acordo com o que o processador programou no passo 1). Nesta fase, é o controlador de DMA que se assume como mestre do sistema e gera todos os sinais dos barramentos (nomeadamente de endereços e de controlo), como se de um processador de tratasse (mas não faz busca de instruções, apenas leituras e escritas da memória), razão pela qual na Fig. 6.50 o barramento de endereços e os sinais de RD e WR aparecem como sinais bidirecionais;
9. Quando terminar a transferência, o controlador de DMA:
  - a) Desactiva o sinal DACK (DMA Acknowledge), para o controlador de disco saber que a transferência terminou;
  - b) Muda a sua interface de acesso aos barramentos, deixando de ser mestre para se converter num simples periférico (barramento de endereços e de controlo passam a ser apenas entradas e coloca o barramento de dados em alta impedância – tristate);
  - c) Desactiva o sinal de BRQ, para o processador saber que o controlador de DMA já não precisa de aceder aos barramentos;
  - d) Activa o pedido de interrupção para o processador, sinalizando que a transferência já foi efectuada.
10. O processador recomeça a sua actividade normal, desactivando o sinal BGT e accedendo se necessário aos seus barramentos;
11. Quando puder, o processador atende o pedido de interrupção, dando início ao processamento dos dados transferidos.

A sequência exacta de operações depende do sistema e de algumas opções usuais em transferências de DMA, em que tipicamente são suportados os seguintes modos:

- Contínuo, ou em bloco – Toda a transferência é feita de uma só vez (caso deste exemplo). Este modo deve ser usado apenas quando os dados já estão todos disponíveis;

A pedido, ou rajada (*burst*) – São transferidas palavras até se chegar ao limite de palavras a transferir ou o periférico não tiver mais palavras prontas para transferir. Este é um modo adequado para periféricos que vão produzindo os dados ao mesmo tempo que eles vão sendo transferidos. Se o periférico se atrasar, pode usar os sinais DREQ e DACK para estabelecer um protocolo com o controlador de DMA que faz com que este possa libertar os barramentos (desactivando o sinal BRQ) se o periférico desactivar o sinal DREQ. Logo que este sinal seja activado de novo, o controlador de DMA volta a activar o sinal BRQ e continua no ponto em que interrompeu, até transferir todas as palavras que o processador indicou inicialmente. Só então a interrupção para o processador é gerada;

- Simples, ou palavra a palavra – O controlador de DMA transfere uma só palavra e desactiva o sinal BRQ, para o activar de novo a seguir se o periférico tiver mais palavras para transferir. A ideia é permitir ao processador ir executando algumas instruções pelo meio das transferências de DMA (ou vice-versa – tudo depende do ritmo de pedidos de DMA) técnica conhecida por *cycle stealing* (roubo de ciclos). Com suporte adequado em hardware, é inclusivamente possível o controlador de DMA aceder ao barramento apenas nos ciclos em que o processador está a executar uma instrução internamente e não acede à memória. Neste caso a transferência não é tão rápida, mas o peso do DMA no desempenho do processador fica praticamente reduzido a zero porque aproveita os ciclos de acesso à memória que o processador de qualquer modo não usaria.

As operações de DMA têm precedência sobre as interrupções, que não são atendidas enquanto o sinal BRQ estiver activo (o processador não poderá aceder à pilha). Assim, é importante que as operações de DMA sejam curtas. As caches (secção 7.5, na página 622) podem permitir que o processador continue a executar, internamente, enquanto o DMA usa os barramentos exteriores. No entanto, mal o processador precise de aceder à memória externa terá de parar até a operação de DMA acabar.

Normalmente, cada controlador de DMA suporta vários canais de DMA, através de pinos DREQ e DACK individuais para vários periféricos. Muitos sistemas também suportam vários controladores de DMA através de suporte para vários mestres (quer processadores quer controladores de DMA) nos barramentos.

#### SIMULACRUM – TRANSFERÊNCIA DE DADOS POR DMA

Esta simulação ilustra a transferência de dados por DMA, tornando o Programa 6.4 e a Fig. 6.50 como base. Os aspectos cobertos incluem os seguintes:

- Verificação do comportamento da transferência de dados por software;

Verificação da programação e do comportamento do PEPE e do controlador de DMA durante uma transferência, usando os vários modos de funcionamento do controlador de DMA;

Medição do desempenho dos vários modos de transferência por DMA, e comparação com a transferência em software.

#### 6.4.2.4 TRANSFERÊNCIA POR PROCESSADOR DE ENTRADAS/SAÍDAS

Os controladores de DMA já conseguem aliviar o processador das transferências de dados, executando-as de forma muito mais rápida. No entanto, são pouco inteligentes, no sentido que só sabem fazer transferências de dados, independentemente dos aspectos funcionais dessas transferências. Tem de ser o processador central a assumir todas as rotinas de entradas, saídas, interrupções, etc.

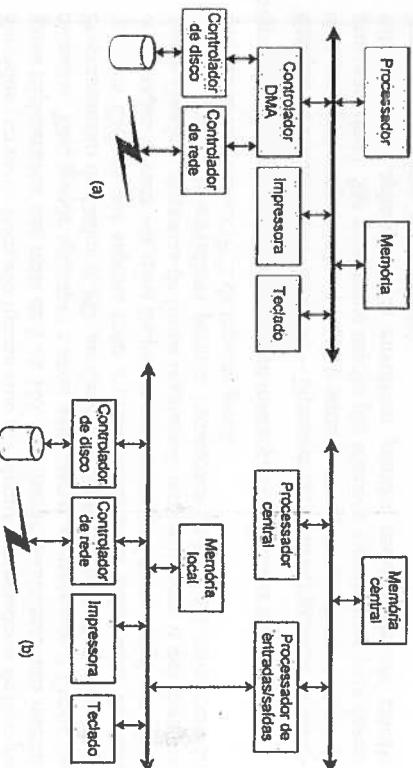


Fig. 6.51 - Arquitectura de entradas/saídas: (a) - Com controlador de DMA; (b) - Com processador de entradas/saídas (que inclui controlador de DMA)

A Fig. 6.51a apresenta esta visão, em que um controlador de DMA optimiza a transferência de dados de e para periféricos que envolvem quantidades apreciáveis de informação, como discos e redes de comunicação. Periféricos de menor débito (ou menos críticos) são tratados directamente pelo processador. Todas as interrupções passam pelo processador.

A Fig. 6.51b mostra outra solução, em que se usou um processador especializado para tratar de todos os aspectos de entradas/saídas. Um processador destes inclui um controlador de DMA para tratar dos periféricos com maior débito de dados. A grande vantagem em relação a usar apenas um controlador de DMA é que as rotinas que tratam directamente com os periféricos podem correr neste processador, que tem a sua própria memória, de forma simultânea com o software de nível mais alto, que corre no processador central. Este tem apenas de lançar o processador de entradas/saídas nas suas tarefas específicas e ficar assim liberto para executar outras actividades.

**ESSENCIAL**

- Dentro de um computador (PC, por exemplo), usam-se normalmente barramentos paralelos para ligar os subsistemas pertencentes ao processador central. Estes barramentos são *standard* (PCI, por exemplo) para permitir ligar periféricos de vários fabricantes e funcionam a frequências mais baixas que o barramento de sistema, que liga o processador às memórias e aos periféricos de maior desempenho (interface gráfica e de rede de comunicação).

- A interacção entre o sistema central (processador e memória) e os periféricos envolve essencialmente transferência de dados, o que implica não apenas copiar os dados de origem para o destino mas também saber quando os dados estão disponíveis para a cópia. Pode ser feita essencialmente de quatro formas distintas:

- teste (polling)** – O processador está em ciclo (esperaactivité) a testar uma condição que indique que a transferência pode ser feita. Esta espera pode ser gravosa porque o processador não pode fazer mais nada;
- interrupções** – O processador está à fazer o seu processamento normal e o periférico avisa o processador quando está pronto para a transferência, gerando uma interrupção;
- DMA (acesso directo à memória)** – Um controlador especializado copia os dados de forma autónoma, em hardware, depois de ser programado pelo processador. O processador não pode fazer acessos à memória ou periféricos durante a transferência. O aviso de que a transferência terminou é feito por interrupção ao processador central;
- processador de entradas/saídas** – Permite não apenas efectuar a transferência de dados por hardware como também algum processamento específico dos dados. Geralmente é um processador especializado, que inclui capacidade de DMA e libera o processador central das tarefas de mais baixo nível das entradas/saídas.

**6.5 EXEMPLOS DE COMPUTADORES COMPLETOS****6.5.1 CLASSES DE COMPUTADORES**

Os computadores são usados em quase todos os aspectos da nossa vida, embora tal não implique necessariamente uma interacção directa com um monitor e um teclado. Para abranger toda a panóplia de aplicações possíveis existe um conjunto de classes de computadores, cada uma com um conjunto de características que a tornam mais adequada para uma dada gama de aplicações. Naturalmente, as diversas classes não são mutuamente exclusivas, nem em termos de características nem em termos de aplicações em que constituem uma opção válida.

As classes de computadores geralmente reconhecidas são as seguintes, por ordem decrescente de capacidades e custo (indicam-se também os termos normais em inglês):

- Grandes computadores**, para servir muitos utilizadores simultaneamente ou aplicações de cálculo muito intensivo;
- Supercomputador (supercomputer)** – Outra reino de processadores muito específicos, custando uma fortuna, são hoje feitos exclusivamente com processadores de topo de gama emergentes da linha dos computadores pessoais. Estes computadores têm tipicamente muitos processadores (centenas ou mesmo milhares), interligados por redes rápidas (*gigabit LAN*, por exemplo). Usam-se sobretudo em aplicações especiais que requerem muitas operações de cálculo científico;
- Servidor (server)** – Formado tipicamente por alguns processadores de topo de gama (tipicamente, não mais de 8 ou 16), partilhando geralmente uma mesma memória. Têm grande aplicação a nível empresarial, constituindo o Centro de Processamento de Dados (CPD, ou Centro de Dados, Centro de Informática, ou *Data Center*) que suporta todo o processamento de informação de uma organização. É cada vez mais frequente usarem-se servidores em placas compactas (*blades*), dispostas de forma organizada em bastidores, o que para um grande número de servidores permite instalações muito mais compactas do que caixas individuais (PCs de topo de gama).
- Computadores pessoais**, tipicamente com interacção com apenas um utilizador;
- Estação de trabalho (workstation)** – Tipicamente, é um computador de secretaria com mais capacidades do que o normal, com processadores de topo de gama (até dois). São geralmente usadas por técnicos especialistas para desenvolvimento de aplicações que envolvam grandes necessidades de cálculo (aplicações gráficas, por exemplo);
- Computador de secretária (desktop)** – São os computadores mais usados, quer a nível profissional quer a nível de lazer. Os computadores actuais já têm boas capacidades, quer em termos de cálculo quer em termos de interacção multimédia com o utilizador;
- Computador de colo (laptop ou notebook)** – Os cada vez mais divulgados "portáteis" são essencialmente computadores de secretaria adaptados à mobilidade, com processadores que consomem menos, caixas mais pequenas, maior nível de integração, etc;
- Computador de internet (netbook)** – Constituem a evolução mais recente dos computadores portáteis, com menor custo devido a usarem ecrãs de menor dimensão e processadores de menor desempenho. Destinam-se essencialmente

a aplicações de baixos requisitos computacionais, como edição de documentos e pesquisas na Internet (o que deu origem ao nome desta categoria);

- **Computador de mão (palm-top)** – Também designados PDAs (*Personal Digital Assistant*), ou nos casos mais simples por agendas electrónicas, cabem literalmente na palma da mão e destinam-se essencialmente às pequenas tarefas pessoais em ambiente totalmente móvel (agenda, bloco de notas, aplicações de gestão pessoal, acesso móvel à Internet, em particular correio electrónico, etc.). Actualmente, assiste-se à convergência entre os PDAs e os telemóveis, reunindo todas as funcionalidades úteis a nível pessoal num só aparelho (incluindo até recepção directa de televisão digital terrestre).

Computadores embutidos, normalmente escondidos dentro de sistemas com aplicações específicas e que não são normalmente reconhecidos como sistemas informáticos:

- **Microcontrolador (microcontroller)** – Um computador num só circuito integrado que reúne processador, memória e periféricos. Usam-se aos biliões por todo o mundo, desde simples brinquedos até aviões, passando por telemóveis, automóveis, edifícios, etc.

As secções seguintes descrevem mais em pormenor os exemplos dos computadores pessoais (vulgarmente designados PCs) e dos microcontroladores.

## 6.5.2 OPC

### 6.5.2.1 ARQUITECTURA ORIGINAL

Vendem-se actualmente na ordem de 300 milhões de PCs por ano em todo o mundo (segundo a IDC - *International Data Corporation*, 295 milhões em 2008 e 282 milhões em 2009). É de longe o computador de uso geral mais divulgado. O termo PC (*Personal Computer*) é abusivo, pois há outras arquiteturas de computadores destinadas a ser usadas apenas por um utilizador, como por exemplo os computadores da Apple (vulgarmente conhecidos por Macs).

O computador pessoal tem este nome por oposição aos grandes sistemas de vários utilizadores, exclusivos de centros de informática e utilizados apenas por técnicos especialistas nas décadas de 60 e 70. O facto de hoje em dia qualquer pessoa, mesmo leiga em termos informáticos, conseguir usar um computador para a sua vida pessoal ou profissional, no conforto da sua casa, no seu escritório ou mesmo em qualquer lado (com os computadores portáteis), contribuiu definitivamente para a transformação da sociedade em que vivemos.

O PC é apenas a arquitetura que mais sucesso comercial teve, sobretudo devido ao facto de a IBM, em cujo contexto apareceu, ter divulgado as especificações. Isto permitiu a outras empresas (em particular asiáticas) produzir computadores compatíveis com o PC original (os chamados *clones*), em grandes quantidades e com preços mais reduzidos do que a IBM o faria. A Apple quase falhou porque durante muito tempo insistiu em ser

única a fabricar os seus computadores. Hoje está mais revitalizada, à custa de uma qualidade muito boa de hardware e software, mas perdeu completamente ein quanitidade e quota de mercado.

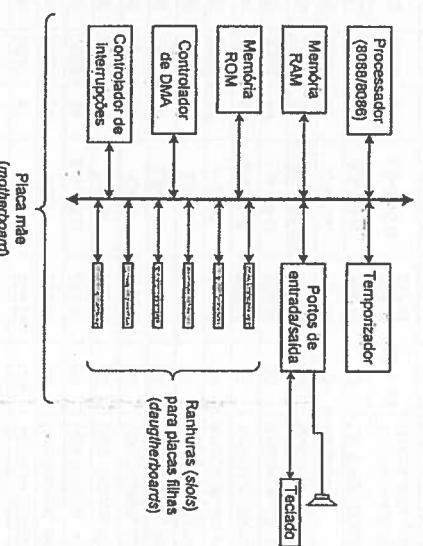


Fig. 6.52 - Arquitectura original do PC (PC/XT)

A arquitetura original do IBM-PC tinha um 8088<sup>52</sup> da Intel como processador. Rapidamente se chegou à conclusão de que em termos de desempenho valia a pena ter o 8086 e os seus 16 bits de barramento de dados, aparecendo o PC/XT. A Fig. 6.52 representa esta arquitetura original, baseada numa placa-mãe (*motherboard*), com algumas ranhuras onde podiam ser encaixadas algumas placas-filha (*daughterboards*), constituindo assim uma arquitetura extensível.

A placa mãe incluía os dispositivos mais básicos, enquanto os subsistemas de periféricos mais elaborados (como o controlador de discos e do disco duro e a placa gráfica de interface com o monitor) ficavam nas placas filhas. O controlador de DMA optimizava a comunicação com estas placas filhas e o controlador de interrupções suportava o DMA, teclado, temporizações, etc. O temporizador gerava as temporizações de sistema, incluindo o som do altifalante, ligado aos portos de entrada/saída, tal como o teclado. Com um 8088 a 4,77 MHz e com um barramento de dados de 8 bits, 64 KBytes de RAM, uma unidade de disquetes de 160 KBytes, monitor de vídeo sem gráficos, PC-DOS 1.0 (nome original do MS-DOS) e algumas aplicações (incluindo um compilador de Basic e uma folha de cálculo, VisiCalc), o IBM-PC arrancou em 1981 para dominar o mundo.

<sup>52</sup> Um 8088 era um processador derivado do 8086, de 16 bits de palavra mas com um barramento externo de dados de apenas 8 bits, para ser mais barato (menos pinos). Um acesso de 16 bits à memória implicava dois acessos em byte. Embora o hardware tratasse disto automaticamente, ficava mais lento do que um 8086.

O PC não tem parado de evoluir nestes seus primeiros 30 anos, desde as primeiras versões com um barramento de dados de 8 bits até às versões actuais de 64 bits e frequências de relógio 1000 vezes superiores. Um dos principais segredos do sucesso está na sua linha evolucionária, sempre preocupada em manter a compatibilidade com as versões anteriores. Por esta razão, pode dizer-se que a arquitectura actual é a possível, num equilíbrio resultante da evolução tecnológica por um lado e de muitos compromissos por outro.

Como qualquer computador, um PC é constituído por um processador, memória, periféricos e barramentos. Todos evoluíram muito, em capacidades e rapidez. Esta completamente fora do âmbito deste livro descrever os detalhes da arquitectura do PC, sobre a qual existe abundante informação ([Buchanan 2001], [Thompson 2004], [Reagan 2005], [Minasi 2005], [PCTechGuide], [PCGuide], [Anandtech] e [PCStats]). Aqui descrevem-se apenas os aspectos fundamentais, e em termos de evolução de modo a perceber-se o que foi conseguido nestes 30 anos.

### 6.5.2.2 EVOLUÇÃO NOS PROCESSADORES

A Tabela 6.18 mostra bem a evolução ocorrida nos processadores ao longo destes anos. Em termos tecnológicos, basta olhar para:

- As frequências dos relógios e o ritmo da sua evolução;
- O nível de integração, medido pelo crescente número de transístores (elementos básicos de todos os sistemas digitais) em cada processador;
- A tecnologia de fabrico, expressa pela dimensão da pista mais estreita no circuito integrado, cada vez mais pequena, consegue actualmente (2010) 0,032 micrón<sup>83</sup>, prevendo-se 0,022 micrón para 2011-12 e 0,016 micrón eventualmente até 2018. Dados os valores, a unidade mais corrente já é o nanómmetro<sup>84</sup> (ou seja, 32 nm, 22 nm e 16 nm).

Outros factores tecnológicos não expressos nesta tabela têm sido extremamente importantes, como por exemplo a tensão de trabalho dos processadores, que tem descido dos 5 V do 8086 para cerca de 1 V nos processadores com menor consumo (importante nos computadores portáteis) e potência dissipada. A redução da largura das pistas nos circuitos integrados e da tensão de trabalho têm sido os factores fundamentais para suportar o aumento da frequência do relógio e da integração.

Dado o seu custo e características, alguns destes processadores são essencialmente usados em servidores, mas nada os impede de ser usados em computadores pessoais. A arquitectura de base do computador é a mesma, diferindo depois nas capacidades do processador, da memória e dos periféricos.

<sup>83</sup> Um micrón é a milionésima parte de um metro, ou um milésimo de milímetro. Símbolo: µ.

<sup>84</sup> Milésima parte de um micrón.

| REFERÊNCIA         | FABR. CANTE | ANO  | BITS (MHZ/32H2) | RELÓGIO (MHZ/32H2) | TRANSC. (NM) | PISTA (NM) | CARACTERÍSTICAS EM <sup>9</sup>                              |
|--------------------|-------------|------|-----------------|--------------------|--------------|------------|--------------------------------------------------------------|
| 8086               | Intel       | 1978 | 16              | 5,8 M              | 29 K         | 3000       | 16 bits, 20 bits endereço                                    |
| 80186              | Intel       | 1982 | 12              | 10-12 M            | 100 K        | 1500       | Primeiro microcontrolador                                    |
| 80286              | Intel       | 1982 | 16              | 6-20 M             | 134 K        | 1500       | Modo protegido, 24 bits endereço                             |
| 80386              | Intel       | 1985 | 32              | 16-33 M            | 275 K        | 1500       | Processador 32 bits                                          |
| 80486              | Intel       | 1989 | 32              | 25-120 M           | 1,2 M        | 1000       | Cache e co-processador matemático internos                   |
| Pentium            | Intel       | 1993 | 32              | 60-200 M           | 3,1 M        | 800        | Superescalair, acesso à memória de 64 bits                   |
| Pentium Pro        | Intel       | 1995 | 32              | 150-200 M          | 5,5 M        | 600        | Cache L2 integrada                                           |
| Pentium MMX        | Intel       | 1996 | 32              | 166-233 M          | 4,5 M        | 350        | Instruções de multimédia                                     |
| Pentium II         | Intel       | 1997 | 32              | 233-450 M          | 7,5 M        | 350        | Cache L2 não integrada, barramento dedicado                  |
| Celeron            | Intel       | 1998 | 32              | 266-433,1 G        | 7,5 M        | 250        | Sem cache L2 ou reduzida instruções para SIMD                |
| Pentium III        | Intel       | 1999 | 32              | 533 M-1 G          | 28,1 M       | 180        | Processador de 7.ª geração                                   |
| Athlon             | AMD         | 1999 | 32              | 500 M-2,2 G        | 22 M         | 250        | Mais instruções, 20 estágios                                 |
| Pentium 4          | Intel       | 2000 | 32              | 1,4-2,8 G          | 42 M         | 180        | Hiperfluxo                                                   |
| Titanium           | Intel       | 2001 | 64              | 733-800 M          | 25 M         | 180        | Processador 64 bits                                          |
| Pentium 4          | Intel       | 2002 | 32              | 3-3,8 G            | 55 M         | 130        | Cache L3: 1,5 - 3 M                                          |
| Titanium 2         | Intel       | 2002 | 64              | 0,9-1,6 G          | 220 M        | 180        | Relógio e tensão variáveis                                   |
| Pentium-M          | Intel       | 2003 | 32              | 1,3-2,1 G          | 77 M         | 130        | Processadores de 64 bits, compatíveis em binário com Pentium |
| Opteron            | AMD         | 2003 | 64              | 1,2-2,4 G          | 100 M        | 130        | Dois núcleos, virtualização                                  |
| Athlon 64          | AMD         | 2003 | 64              | 2,0-2,2 G          | 106 M        | 130        | Dois núcleos, hiperfluxo                                     |
| Titanium Madison   | Intel       | 2004 | 64              | 1,6 G              | 592 M        | 130        | Dois núcleos, L3: 2x12 M                                     |
| Pentium Extreme    | Intel       | 2005 | 32              | 3,2 G              | 230 M        | 90         | Dois núcleos, virtualização                                  |
| Athlon 64 X2       | AMD         | 2005 | 64              | 2,0-2,4 G          | 233 M        | 90         | Dois núcleos, 2 processadores                                |
| Dothan Xeon        | Intel       | 2005 | 64              | 2,8 G              | 230 M        | 90         | Dois núcleos, hiperfluxo                                     |
| Titanium Montecito | Intel       | 2006 | 64              | 1,7 G              | 1720 M       | 90         | Dois núcleos, L3: 2x12 M                                     |
| Core               | Intel       | 2006 | 64              | 1,8-3,3 G          | 291 M        | 65         | Dois núcleos, virtualização                                  |
| Kentsfield         | Intel       | 2006 | 64              | 2,66 G             | 582 M        | 65         | 4 núcleos, 2 processadores                                   |
| Dothan Barcelona   | AMD         | 2007 | 64              | 2-2,5 G            | 463 M        | 65         | 4 núcleos, um processador                                    |
| Pheim Xeon         | Intel       | 2007 | 64              | 2-3,16 G           | 820 M        | 45         | 4 núcleos, tecnologia 45 nm                                  |
| Phenom X4          | AMD         | 2008 | 64              | 1,8-2,6 G          | 450 M        | 65         | 4 núcleos, L3: 2 MB                                          |
| Bloomfield         | Intel       | 2008 | 64              | 2,66-3,33 G        | 731 M        | 45         | QuickPath, hiperfluxo, DDR3                                  |
| ATOM               | Intel       | 2008 | 32              | 0,8-2 G            | 47 M         | 45         | Netbooks                                                     |
| Phenom II X4       | AMD         | 2009 | 64              | 2,5-3,4 G          | 758 M        | 45         | L3: 6 MB, DDR3                                               |
| Clarkdale          | Intel       | 2010 | 64              | 2,8-3,46 G         | 383 + 177 M  | 32 + 45    | 2 núcleos em 32 nm, processador gráfico a 45 nm              |
| Phenom II X6       | AMD         | 2010 | 64              | 2,6-3,2 G          | 904 M        | 45         | 6 núcleos, L3: 6 MB, DDR3                                    |
| Titanium Tukwila   | Intel       | 2010 | 64              | 1,6-1,73 G         | 2066 M       | 65         | 4 núcleos, L3: 24 MB                                         |
| Opteron 6100       | AMD         | 2010 | 64              | 1,7-2,3 G          | 1810 M       | 45         | 12 núcleos, L3: 12 MB                                        |
| Nehalem-EX         | Intel       | 2010 | 64              | 1,7-2,6 G          | 2300 M       | 45         | 8 núcleos, L3: 24 MB                                         |

Tabela 6.18 - Evolução das características dos principais processadores da Intel e da AMD.

Em termos arquitecturais (organização interna dos vários blocos do processador), os marcos mais importantes (que não serão aqui detalhados) são os seguintes:

- Evolução para uma arquitectura de 32 bits com o 80386, que se manteve ainda nos Pentium (arquitectura IA-32);

Integracão no processador das *caches* de dados e instruções (Fig. 4.3, na página 191) e do co-processor matemático, permitindo que passassem a funcionar à velocidade máxima. A partir do 486, a frequência interna do processador passou a ser superior a dos barramentos para acesso à memória;

Funcionamento superescalar com o Pentium, com várias unidades em execução simultânea, o que permitiu executar mais instruções por ciclo de relógio, em média. Também com o Pentium o barramento de dados foi alargado para 64 bits para permitir aceder à memória de forma mais rápida (mas o processador continuou a ser de 32 bits);

- Introdução de um segundo nível de *cache* (L2) de grande capacidade (256 KBytes a 1 MByte), dentro do próprio processador, a partir do Pentium III;

Instruções específicas para suportar multimédia e correntes (*streams*) de vídeo (extensões multimédia – MMX – e de processamento de corrente – SIMD);

- Evolução para arquitecturas de 64 bits, com o Itanium, o Athlon 64, o Opteron, o Xeon e o Core 2;

Exploração do paralelismo ao nível das instruções, com o Itanium, permitindo executar simultaneamente várias instruções, desde que unas não dependam dos resultados das outras;

- Hiperfluxo (*hyperthreading*), permitindo executar duas, quatro ou mais tarefas de forma independente no mesmo processador, conduzindo a um melhor aproveitamento dos recursos. Os processadores incluem normalmente várias unidades que podem funcionar simultaneamente, mas as instruções têm dependências de dados (secção 7.3.5, na página 613) que impedem a utilização contínua de todas as unidades. A técnica de hiperfluxo permite executar simultaneamente instruções de vários processos independentes, de modo a maximizar o aproveitamento da execução dessas unidades. O hardware garante que os dados e controlo dos vários processos não se misturam, o que implica ter várias réplicas de alguns recursos, como por exemplo o banco de registos, e coordenação estreita na utilização partilhada de outros, como as unidades aritméticas e as *caches*. Do ponto de vista lógico, tudo se passa como se o processador implementasse vários processadores, embora tal não corresponda à realidade. É melhor do que só um processador, mas tem a limitação de que há um conjunto de recursos partilhados pelo que um processador com duas vias de hiperfluxo, por exemplo, não executa dois processos independentes com o dobro do desempenho. Dependendo dos processos, a melhoria do desempenho neste caso face a não ter hiperfluxo poderá ser da ordem dos 20% a 30%;

- Virtualização, que permite que o mesmo processador seja partilhado por vários sistemas operativos (mesmo diferentes, como Windows e Linux) de forma total-

mente transparente (sem alterações aos sistemas operativos ou aos seus programas) e isolada (sem interferência de uns nos outros), dando a ilusão de que existem realmente vários processadores. Esta característica, com suporte em hardware nos processadores mais recentes e de mais alto desempenho, permite aproveitar melhor os recursos computacionais (reduzindo os tempos mortos de processamento) e gerir os sistemas informáticos de forma muito mais dinâmica; Gestão dinâmica do consumo e desempenho, em que um processador suporta diversas características de controlo não funcional que permitem obter mais desempenho (à custa do consumo) ou maior autonomia num computador portátil (à custa do desempenho), nomeadamente através da variação da tensão de alimentação e da frequência de relógio do processador. É ainda possível recorrer a outros truques, como desligar unidades internas do processador quando não estão a ser usadas e aumentar temporariamente a frequência de relógio acima do valor máximo, desde que a temperatura interna do processador e potência dissipada estejam dentro dos limites admissíveis;

- Multiprocessamento, em que cada processador possui dois, quatro ou mais núcleos, permitindo a execução realmente simultânea de vários programas – ver secção 7.7.1, na página 664);

Integracão resultante da evolução tecnológica e consequente capacidade de colocar mais transistores no mesmo circuito integrado. Inicialmente, os muitos circuitos integrados que rodeavam o processador foram aglutinados num conjunto de poucos circuitos integrados (*chipsset*), mas depois os mecanismos mais críticos, como os controladores de ligação à memória e ao barramento de periféricos e até a unidade de processamento de gráficos (GPU), foram integrados no mesmo circuito do próprio processador.

Em termos de mercado, apenas a Intel<sup>85</sup> e a AMD<sup>86</sup> (ambas formadas por filiais da Fairchild Semiconductors) têm assegurado, desde o inicio, o desenvolvimento de processadores para os PCs. A AMD tem estado sempre atrás da Intel, mas desde o lançamento do Athlon, em 1999, tem ganho notoriedade tecnológica, reforçada com os processadores Opteron e Phenom (embora neste campo haja sempre muitas opiniões) e preços mais baixos. A Intel está mais avançada em termos de tecnologia de fábrica, tendo já lançado em 2010 processadores com tecnologia de 32 nm. No entanto, a AMD adiantou-se no número de núcleos, tendo lançado a última família de processadores Opteron com 12 núcleos. Nada a que a Intel não seja capaz de responder num curto espaço de tempo, nesta luta contínua pela supremacia e quota de mercado.

Uma diferença significativa entre as duas empresas é a forma como evoluíram para a arquitectura de 64 bits. Com o Itanium (arquitectura IA-64), a Intel rompeu a compatibilidade binária (código executável), permitindo aos programas já compilados correrem por

<sup>85</sup> Constituída em 1968 e cujo nome deriva de *Integrated Electronics* ([www.intel.com](http://www.intel.com))

<sup>86</sup> Constituída em 1969 e cujo nome significa *Advanced Micro Devices* ([www.amd.com](http://www.amd.com))

meio de uma tradução entre binários efectuada em software. No entanto, o aproveitamento dos benefícios arquitecturais só é real por meio de recompilação dos programas fonte. A AMD preferiu manter compatibilidade mesmo ao nível do código executável (arquitectura AMD64), o que permite aos programas executáveis já existentes correrem directamente sem alterações nem perda de desempenho e portanto uma transição mais suave do mundo de 32 bits para o de 64 bits. O facto é que a Intel incluiu as extensões da AMD para 64 bits na sua linha de processadores mais comum, mantendo a visão mais evolucionária (primeiro, arquitectura Netburst, depois Core e actualmente Nehalem).

Outro aspecto extremamente importante tem sido a invasão do mercado dos servidores pelos processadores usados nos PCs. Outra o reino dos supercomputadores com tecnologias específicas que custavam fortunas, hoje em dia todo o esforço colocado no avanço tecnológico dos PCs (em termos de processadores e do resto) faz com que os grandes servidores e mesmo os supercomputadores sejam feitos com base nos processadores de topo de gama derivados da tecnologia dos PCs. Os servidores empresariais usam tipicamente entre um e oito processadores em regime de memória partilhada, enquanto os grandes supercomputadores usam milhares, com redes de interligação adequadas.

Por outro lado, a gama de processadores de alto desempenho existentes no mercado tem tendência a concentrar-se. A pouco e pouco, os que estão fora do mercado dos PCs têm vindo a perder terreno ou mesmo a desaparecer, por não conseguirem acompanhar o ritmo de investimento na evolução tecnológica.

No verão de 2004, a Hewlett-Packard<sup>87</sup> anunciou o fim do processador de 64 bits Alpha, um dos primeiros desta classe a aparecer, em 1992. Na altura, passou a usar os Itanium 2 da Intel nos seus servidores de topo de gama.

A Silicon Graphics Inc.<sup>88</sup> produzia servidores baseados no MIPS, outro processador de 64 bits, mas passou a usar os processadores da Intel (Itanium 2 e Xeon).

A Sun Microsystems<sup>89</sup> vende servidores baseados no seu processador UltraSPARC, mas no entanto não deixa de ser relevante o facto de a sua linha de servidores incluir também processadores da AMD e da Intel.

No verão de 2005, a Apple anunciou a mudança de processadores PowerPC para processadores da Intel em todos os seus novos modelos de Macs. A mudança ficou completa em Agosto de 2006. A IBM continuou o desenvolvimento da arquitectura POWER, subjacente aos processadores PowerPC, mas apenas no reino dos servidores. Em 2010 lançou o POWER7, que usou para constituir um supercomputador (com muitos processadores) capaz de executar mais de 1 petaFLOPS ( $10^{15}$  operações de vírgula flutuante por segundo).

O mercado dos processadores de alto desempenho está assim reduzido a quatro fabricantes: Intel, AMD, IBM e SUN. A frente da "guerra" da concorrência, mais acesa entre a Intel e a AMD, dada a abrangência do mercado dos PCs, centra-se actualmente no reino dos processadores com vários núcleos e na redução das dimensões da tecnologia.

O mercado começou com uma série de processadores com dois núcleos, incluindo o Power4 da IBM, já desde 2001; em 2004 o UltraSPARC T4 da SUN e em 2005 o dual-core Opteron e Athlon 64 X2 (ambos da AMD) e o dual-core Xeon (Intel). Desde então, até Abril de 2010, os desenvolvimentos mais interessantes foram os seguintes:

- Em 2006, o núcleo duplo de 64 bits chegou aos PCs portáteis, com o Core 2 Duo da Intel e o Turion 64 X2 da AMD;
- A Sun Microsystems lançou o T1 (também conhecido por Niagara), um processador com oito núcleos tipo UltraSPARC, cada um com capacidade para quatro tarefas em execução simultânea;

A Intel iniciou a produção do Montecito, um processador com dois núcleos Itanium 2 Madison, cada um com caches L2 de 1 MByte e L3 de 12 MBytes, tudo empacotado num circuito integrado com 1720 milhões de transistores;

No fim de 2006, a Intel lançou processadores com quatro núcleos, num esforço de resposta ao adiantamento por parte da AMD, que foi a primeira a dispor de dois núcleos em arquitecturas x86;

Já em 2007, a IBM lançou o Power6, sucessor do Power5, com 4,7 GHz de frequência de relojão, 790 milhões de transistores e tecnologia de 65 nm, duplicando o desempenho do seu antecessor, o Power5, praticamente sem aumento do consumo;

A SUN lançou o UltraSPARC T2 (também conhecido por Niagara II), com oito vias de hiperfluxo em cada um dos oito processadores;

A AMD só chegou aos quatro núcleos em Setembro de 2007, com o Opteron Barcelona, mas com quatro núcleos nativos numa só pastilha, ao passo que a Intel tinha justaposto na mesma embalagem dois processadores de dois núcleos cada;

No final de 2007, a Intel apresentou a família Penryn, com a novidade do fabrico com tecnologia de 45 nm;

Já em 2008, a AMD lançou o Phenom, com quatro núcleos para computadores pessoais, ainda com 65 nm, enquanto a Intel lançou o Bloomfield em 45 nm, com QuickPath (ligação rápida, correspondente ao Hypertransport da AMD), hiperfluxo e suporte para memórias DDR3;

Ainda em 2008, a Intel lançou o Atom, processador de baixa gama para a classe dos netbooks e dispositivos de acesso à Internet, que veio a revelar-se um enorme sucesso de mercado;

Em 2009, a AMD chegou à tecnologia de 45 nm e ao suporte à memória DDR3, com o Phenom II;

Em 2010, a concorrência entre a Intel e AMD parece ter-se acentuado, apesar de ambas as empresas terem celebrado em Novembro de 2009 um acordo que resol-

ve as disputas legais que tinham até então. Em Janeiro, a Intel lançou o Clarkdale, usando pela primeira vez a tecnologia de 32 nm, mas apenas no processador. Na mesma embalagem, incluiu ainda um processador gráfico com tecnologia 45 nm. Em Fevereiro, lançou a nova versão do Itanium, o Tukwila, com mais de 2 mil milhões de transistores, e em Março lançou o Gulftown com seis núcleos, em 32 nm, e o Nehalem-EX com oito núcleos, em 45 nm. Pelo seu lado, a AMD, ainda em 45 nm, lançou o Phenom II X6, com seis núcleos, e o Opteron 6100, para os servidores, com 12 núcleos, adiantando-se à Intel em número de núcleos;

Em Fevereiro de 2010, a IBM apresentou o Power7, sucessor do Power6, em 45 nm, com 1200 milhões de transistores, oito núcleos e frequências de 3,0 a 4,14 GHz; ao mesmo tempo, a SUN apresentou o UltraSPARC T3 (Niagara 3), com 1000 milhões de transistores em tecnologia de 40 nm, 16 núcleos (todos na mesma pastilha) e ligações de alto débito que permitem interligar até quatro processadores sem circuitos adicionais.

Os planos futuros dos vários fabricantes passam por tecnologia de menor dimensão, mais núcleos, mais integração, maior desempenho e menor consumo.

A tecnologia de silício parece não ter limites imediatos, com o processo de 22 nm previsto já para 2011 ou 2012 e o de 16 nm lá para 2018. A Intel tem algum avanço nesta área, pois foi o primeiro fabricante de processadores a atingir 32 nm. Os limites da tecnologia, cujas dimensões mínimas começam a aproximar-se das do átomo (diâmetro na ordem dos 0,1 a 0,7 nm), poderão começar a impor-se nos próximos anos. Mas já antes pelo limite tinha sido anunciado e a evolução tecnológica foi solucionando os problemas, pelo que poderá haver desenvolvimentos que continuem a sustentar a lei de Moore, sabe-se lá até quando.

Um dos aspectos interessantes da crescente miniaturização é o facto de, para além de permitir mais circuitos numa só pastilha (vários núcleos e caches cada vez maiores), conduzir a uma maior frequência de processamento. No entanto, o consumo de potência aumenta muito com a frequência, e o consumo por processador já há muito que ultrapassou os 100 Watts, tendo mesmo em alguns casos até 130 Watts, que já é considerado nos limites (mais é possível, mas com recurso a técnicas específicas de arrefecimento). A tendência é para procurar mais desempenho no aumento do número de núcleos e não no aumento da frequência. A eficiência energética está a assumir cada vez mais importância, com todos os fabricantes a tentarem produzir processadores com menor consumo, mesmo nos servidores de topo de gama.

O problema do consumo não é apenas dos dispositivos portáteis mas também dos servidores fixos, em que a miniaturização também tem acontecido ao nível dos computadores completos. Nos centros de dados (*data centers*), os servidores compactos (*blades*<sup>90</sup>) permitem uma grande densidade de poder computacional, mas conduzem também a

consumos da ordem de 30 kW ou mais por cada bastidor, o que se traduz num problema muito sério de remoção do calor produzido. Em Outubro de 2006, a SUN lançou um centro de dados (*Blackbox*) de grande capacidade, em servidores (até 250) e em discos (até 1,5 Petabytes<sup>91</sup>), num contedor de transportes internacionais normalizado, que, para além de uma ligação a uma rede de comunicações e de uma tomada de electricidade, precisa de uma ligação de água para arrefecimento. Desde então, vários fabricantes têm igualmente produzido centros de dados compactos e pré-montados.

Os fabricantes de processadores têm vindo a fazer um esforço na redução do consumo e têm conseguido cada vez mais desempenho por watt de potência consumido, o que tem permitido aumentar o desempenho e reduzir o consumo simultaneamente.

Mas nem sempre o desempenho é o factor primordial. Em 2008, a Intel lançou um novo processador, o Atom, em que os aspectos fundamentais são a redução de custo e do consumo, destinado essencialmente à categoria de *netbooks* (ver secção 6.5.1). Inicialmente, foi concebido para computadores portátiles de muito baixo custo para países menos desenvolvidos e em particular para crianças (na linha da iniciativa OLPC – *One Laptop Per Child*), tendo sido desenvolvido um computador específico para o efeito (o Classmate). No entanto, acabou por ser um sucesso nos países mais desenvolvidos, onde foi visto como um PC totalmente funcional mas de baixo custo, suficiente para acesso à Internet e factor de redução da competição pelo uso do computador familiar. O computador Magalhaes usa este processador.

Mas os *netbooks* estão mesmo a converter-se num fenômeno comercial, pois grande parte das pessoas não precisa de elevar desempenho dos processadores topo de gama. De acordo com a Strategy Analytics, em 2009 venderam-se 30,2 milhões de *netbooks*, um aumento de 79% face a 2008, com Acer, Asus, HP e Dell como os fabricantes de topo. Em compensação, e segundo a IDC, o número total de PCs vendido em 2009 foi de 282 milhões, uma descida face aos 295 milhões de 2008. Há quem defende que a proporção dos *netbooks* nos PCs irá aumentar e a ABI estima que em 2013 se venderão cerca de 139 milhões de *netbooks*. Trata-se, portanto, de um mercado em grande expansão e que revela uma tendência que poderá ser comparada à que já ocorreu na aviação comercial, com as companhias *low-cost*. Para o público em geral, o custo é um factor dominante.

### 6.5.2.3 EVOLUÇÃO NAS MEMÓRIAS

Um dos maiores dramas dos processadores é o facto de precisarem de aceder constantemente à memória, pois têm um número muito limitado de registos. Para reduzir esses acessos, foi fundamental a existência de *caches* dentro dos próprios processadores, que contêm uma cópia das palavras de memória mais usadas e evitam o acesso externo se a palavra pretendida lá estiver (o que tipicamente ocorre em mais de 95% dos acessos).

A partir do 80486, os processadores passaram a dispor de *caches* internas (tipicamente separadas, uma para código e outra para dados), designadas por nível 1 (L1) porque mais

<sup>90</sup> Formato compacto, apropriado para ser montado num bastidor e não numa caixa individual. Um bastidor pode conter dezenas de servidores blade.

<sup>91</sup> 1 Petabyte =  $10^{15}$  bytes = 1000 Terabytes = 1.000.000 GigaBytes.

tarde apareceram outros níveis de cache (níveis 2 e 3, ou L2 e L3), que primeiro eram externas e depois foram sendo integradas dentro do processador. As caches L2 e L3 são unificadas, isto é, servem tanto para dados como para código. Cada núcleo tem o seu próprio conjunto de caches L1 e L2, enquanto a cache L3 poderá ser partilhada pelos vários núcleos, em particular se forem mais de dois. A dimensão das caches de níveis L1 e L2 é da ordem das "dezenas e centenas de Kbytes, tipicamente, enquanto a das caches de nível L3 é da ordem das unidades ou mesmo dezenas de MBytes. As caches são detalhadas na secção 7.5, na página 622.

As caches internas funcionam à velocidade do processador (actualmente na ordem dos 2 a 4 GHz), mas a memória externa (designada memória principal ou memória de sistema) é mais lenta. As caches usam memória estática (ou SRAM), gastando tipicamente entre quatro e seis transistores por cada bit de memória para o fazer. Desde que alimentado, o circuito mantém o valor indefinidamente.

A memória principal tem de ter uma capacidade muito superior, pelo que é obrigada a usar apenas um transistor, mantendo o bit memorizado apenas por uma carga eléctrica que se vai esvaindo ao longo do tempo devido a fugas por o isolamento não ser perfeito (tal como um depósito de água com um furo). Esta carga tem de ser refreshada periodicamente, tal como um artista de circo com vários pratos a rodar no alto de varas flexíveis devido ao atrito. Esta memória designa-se memória dinâmica (DRAM) e precisa de estar sempre a ser acedida de modo a refreshar o valor guardado em cada célula. No entanto, como está organizada numa matriz de linhas e colunas, é possível refreshar-se toda uma linha só com um acesso. As DRAMs incluem normalmente circuitos internos que permitem que elas próprias se encarreguem do seu refreshamento interno.

As memórias dinâmicas para os PCs, com 16, 32 e 64 bits de largura, são feitas com vários circuitos integrados e vêm em módulos, pequenas placas de circuito impresso que se encaixam em suportes adequados na placa mãe. Desta forma é possível dotar um PC de mais ou menos memória. Os primeiros módulos tinham contactos de apenas um lado, sendo designados SIMMs (Single Inline Memory Module), com 30 e depois com 72 contactos, mas com o aumento da capacidade e largura das memórias passaram a DIMMs (Dual Inline Memory Module), com 168 contactos, depois 184 e actualmente 240.

As memórias dinâmicas são razoavelmente lentas, com um tempo de acesso a uma célula individual que se reduziu na ordem de 10 vezes no mesmo período em que a frequência de relógio dos processadores aumentou na ordem de 1000 vezes. A razão é simples: nesse período, a capacidade das memórias aumentou na ordem das 100.000 vezes, e quanto maior é o circuito mais tempo os sinais demoram a propagar-se.

No entanto, e graças às caches e aos controladores de DMA, estas memórias não são tipicamente usadas em acessos individuais mas sim em blocos de várias palavras, ou acesso em rajada (*burst*). Isto significa que o que tem de se optimizar é o tempo de acesso a um conjunto de vários endereços de palavra consecutivos.

As primeiras memórias dinâmicas usadas nos PCs suportavam já acessos em rajada (FPM<sup>92</sup> DRAMs), especificando o número de uma linha e depois variando o número da coluna, mas desactivando os circuitos da DRAM antes de passar à próxima coluna. Isto introduzia um atraso que as DRAMs EDO<sup>93</sup> resolveram, desligando os circuitos apenas no fim do acesso em rajada.

Uma das dificuldades das DRAMs era a sua sincronização com os sinais do processador, pois tanto as FPM como as EDO eram assíncronas. As SDRAMs (Synchronous DRAMs), que passaram a ser suportadas nos PCs a partir de 1996, vieram permitir o aumento da taxa de transferência em rajada, tendo a DRAM permanentemente sincronizada com um relógio do sistema, de 100 MHz e 133 MHz depois. Estas memórias passaram a ser confeccionadas por PC100 (10 ns de tempo de acesso) e PC133 (7,5 ns de tempo de acesso). É importante notar que estes tempos de acesso são já durante a rajada (um novo valor por cada ciclo de relógio). Nestas memórias, o primeiro acesso demora tipicamente 5 a 10 vezes estes valores, resultante da soma dos tempos necessários para a especificação do número da linha e da coluna e para a preparação para o acesso. Nos acessos subsequentes, basta pulsar um sinal para as diversas palavras serem accedidas sequencialmente, em rajada, sem necessidade de especificar novo endereço. Nas memórias mais recentes, com frequências de acesso mais elevadas mas ainda com tempos de preparação de acesso não muito mais curtos, a diferença entre o primeiro acesso e os seguintes é até maior, na ordem de 10 a 20 vezes.

O próximo passo da Intel foi desenvolver a RDRAM (Direct Rambus DRAM), que deveria proporcionar maiores taxas de transferência (cerca do dobro). No entanto, dificuldades e atrasos nesse desenvolvimento, em conjugação com o elevado custo dessa tecnologia e com o facto de a AMD e outras empresas terem apostado na SDRAM PC133, acabaram por fazer a Intel desistir no princípio de 2000 e passar a apostar também nas SDRAMs.

Entretanto, no fim de 1999 apareceram as DDR<sup>94</sup> SDRAM, que permitiram aceder aos dados em rajada nos dois flancos do relógio, enquanto as memórias anteriores só disponibilizavam os dados num dos flancos do relógio. Isto é conseguido fazendo uma leitura simultânea de duas palavras em endereços consecutivos (aproveitando a forma designada prefetch buffer) e fazendo-as sair depois ao dobro da velocidade, nos dois flancos do relógio. Isto obriga a ler sempre duas palavras consecutivas, o que melhora os acessos a endereços consecutivos mas não a endereços aleatórios. No entanto, dado que os programas tendem a trabalhar num conjunto de endereços consecutivos e não de forma dispersa pela memória (secção 7.5.1, na página 622), em termos práticos este esquema

<sup>92</sup> FPM = Fast Page Mode (ou Modo de Página Rápido – naturalmente, rápido nessa altura...).

<sup>93</sup> EDO = Extended Data Out (ou Dados de Saída Extendidos, numa alusão a manter os circuitos activados da DRAM activos até ao fim da rajada).

<sup>94</sup> DDR = Double Data Rate, ou taxa de transferência dupla.

compensa. Estas memórias passaram a ser usadas nos PCs em 2000, pela mão da AMD, enquanto a Intel só introduziu suporte para elas em 2001.

Em 2003 surgiram as memórias DDR2, que passaram a ler simultaneamente quatro palavras consecutivas para o *prefetch buffer*, o que para a mesma frequência de trabalho interna da memória duplica a frequência máxima com que o processador pode ler da memória (mas tendo de ler um bloco de quatro palavras consecutivas). Em 2007 apareceram os primeiros computadores com suporte para memórias DDR3, com um *prefetch buffer* de 8 palavras, e em 2010 são já a norma nos processadores de maior desempenho. Em 2012 deverão surgir as primeiras memórias DDR4.

Enquanto o tamanho do *prefetch buffer* tem permitido aumentar a taxa de transferência entre a memória e o processador, a redução da tensão de alimentação das memórias e das dimensões da tecnologia tem permitido aumentar a capacidade de cada circuito integrado (4 Gbit/s em 2010) e a frequência de trabalho interna das memórias, já superior a 200 MHz. Um módulo DDR3 de 64 bits de largura a funcionar internamente a 200 MHz consegue transferir, no máximo, 8 palavras de 64 bits em 5 ns, ou 12,8 Gbytes/s.

#### 6.5.2.4 EVOLUÇÃO NOS PERIFÉRICOS

Também nesta área tem havido uma grande evolução, embora nem sempre com grande impacte na arquitetura do PC (aparte a placa gráfica, que tem sido sempre a grande cliente do crescente desempenho do PC). Os periféricos podem ser de diversos tipos:

- Memória não volátil (disquetes, discos duros, CD-ROMs, DVDs, *pen-drives*, etc.);
- Multimédia (placa gráfica, som, digitalização de vídeo, digitalização de imagens, etc.);
- Comunicações (porta série, USB, LAN, redes sem fios, etc.);
- Outros periféricos (teclado, rato, impressoras, aquisição de sinais, etc.).

A secção 6.3, na página 482, já descreveu algumas destes periféricos. Está fora do âmbito deste livro aprofundar este tópico, referindo-se apenas alguns exemplos mais paradigmáticos e representativos da evolução:

- As disquetes evoluíram desde 5,25 polegadas e 160 KBytes de capacidade em 1981 até 3,5 polegadas e 1,44 MBytes em 1987, altura em que pararam. Em 1995 apareceram as unidades Zip, da Iomega, com cerca de 100 MBytes de capacidade, estendida para 250 MBytes em 1999 com uma unidade com ligação USB. Depois disso, as *per-drives*, construídas com ROM Flash, sem partes mecânicas e interface USB, tomaram conta do sector das unidades de memória de massa removíveis. Em 2010 já vão em capacidades na ordem das centenas de GBytes;
- Os discos magnéticos duros, apesar de velhinhos (a primeira unidade foi construída em 1954 pela IBM, com 5 MBytes), continuam a dar conta do recado e a ser o cavalo de batalha da memória de massa. O primeiro disco duro usado nos PCs (no PC/XT) tinha 10 MBytes de capacidade. Até agora, os discos aumenta-

ram a sua capacidade na ordem das 200.000 vezes (para cerca de 1 TByte, em 2010) e melhoraram muito os seus tempos de acesso e taxas de transferência. Definiram-se interfaces apropriadas para os discos, que foram sendo introduzidas à medida que a tecnologia evoluía e se atingiam os limites das interfaces anteriores (IDE, EIDE, ATA, ATA séria – SATA). Actualmente, os PCs suportam vários canais SATA com capacidades de transferência na ordem de 3 Gbytes/s. Também se usam unidades externas, com ligação USB, mas essencialmente para cópias de segurança (*backups*), uma vez que os tempos de acesso (incluindo o tempo de transferência dos dados) são em média superiores às unidades internas;

- Os sistemas RAID (Redundant Array of Independent Disks) incluem um conjunto de discos com várias técnicas para optimizar a memória de massa em servidores, incluindo suporte para redundância e recuperação em caso de falhas;
- Os SSD (Solid State Drives) são dispositivos de memória de massa sem partes mecânicas, com capacidades já na ordem do 1 TByte. Têm a vantagem de ser mais rápidos do que os discos e serem totalmente electrónicos, embora ainda sejam mais caros por Mybyte do que os discos e tenham um número de escritas limitado que introduz algumas restrições ao seu funcionamento. O futuro dirá se conseguirão destronar os discos magnéticos;
- As velhas portas série e paralelo já desapareceram, em favor do USB (secção 6.3.4.3);
- As interfaces de comunicação em rede local já vão na casa dos 10 Gbit/s;
- As placas gráficas não param de evoluir, empurradas pelas sempre “guilosas” necessidades de manipulação de imagens (em jogos 3D, nomeadamente) e vídeo em tempo real, já com suporte para HDTV (televisão de alta definição). Actualmente, os PCs suportam uma ligação à placa gráfica com uma taxa de transferência na ordem dos 8 Gbytes/s, em pé de igualdade, em termos práticos, com a memória principal;
- Em 2010, a Intel lançou dois processadores (Arrandale para PCs portáteis e Clarkdale para PCs de secretaria) já com um processador gráfico integrado na mesma embalagem. A AMD tem previsto algo semelhante para 2011, com uma nova arquitetura designada Fusion.

#### 6.5.2.5 EVOLUÇÃO NOS BARRAMENTOS

Um dos componentes mais críticos e limitativos tem sido a infra-estrutura de interligação (os barramentos), não apenas porque toda a informação por lá passa mas também por questões de normalização e aceitação por parte do mercado (fabricantes de periféricos). As ranhuras (*slots*) para as placas filhas no IBM-PC original (Fig. 6.52) disponibilizavam apenas 8 bits de barramento de dados e constituíam uma grande limitação ao desempenho. Este aspecto foi melhorado com o barramento ISA (Industry Standard Architecture) de 16 bits quando apareceu o PCI/AT, baseado no processador 80286. A normalização do barramento ISA permitiu a produção de inúmeras placas para o PC, o que contribuiu também para o seu sucesso, mas fixou a frequência de trabalho do ISA em 8,33 MHz, o

que implicou desacoplá-lo do barramento de sistema por meio de um circuito adaptador, para permitir a evolução da frequência do relógio do processador. O ISA permitia apenas um ritmo de transferência máximo de cerca 16 MBytes/s, o que mostrou ser muito limitado mal apareceram as placas gráficas. Ainda houve uma extensão, o EISA (Extended ISA), que permitiu transferências de 32 bits, mas a frequência de trabalho era a mesma e continuou a ser insuficiente.

O PCI (*Peripheral Component Interconnect*) foi desenvolvido pela Intel no início da década de 90, começando com 32 bits a 33 MHz mas estendendo depois as características para 64 bits a 66 MHz, conseguindo 528 MBytes/s de taxa de transferência máxima. O PCI ofereceu ranhuras para placas filhas, tal como o ISA, embora não compatíveis. Durante alguns anos, os PCs ofereceram os dois tipos de ranhuras para permitir tanto ligar as placas mais antigas como as novas, baseadas no PCI. As placas gráficas foram naturalmente das primeiras a aproveitar o desempenho do PCI.

A crescente integração dos vários componentes dos PCs levou ao aparecimento de conjuntos de circuitos integrados (*chipsets*) que incluíam muitos destes componentes, e em particular todos os circuitos de suporte ao PCI, nesta altura a espuma dorsal (*back-bone*) dos PCs e que servia de ponto de ligação de outros barramentos dos PCs, como o ISA, o USB (que entretanto estava a surgir) e o EIDE (barramento interno para ligação aos discos, que depois evoluiu para ATA até chegar a SATA – Serial ATA).

A Fig. 6.53 mostra a arquitectura típica dos barramentos de um PC da altura do Pentium II, com um *chipset* dividido em duas partes (pontes): a ponte norte, que interliga a cache, o processador, a memória e o PCI, e a ponte sul, de menor desempenho, que permite a ligação aos periféricos não ligados directamente ao PCI. A placa de áudio podia também já estar ligada ao PCI, mas assim ilustra a coexistência de periféricos mais antigos com os mais modernos, para PCI.

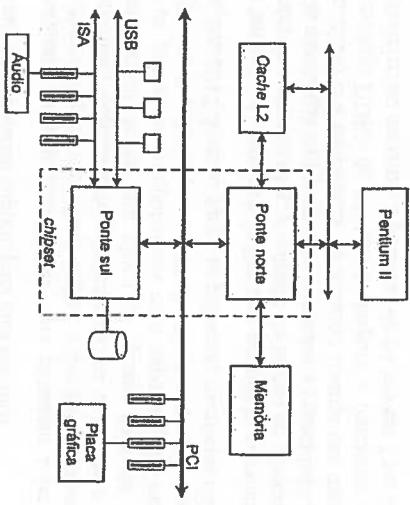


Fig. 6.53 - Arquitectura do PC com PCI e o seu *chipset*

Os próximos passos da evolução, ilustrados pela Fig. 6.54 correspondem à integração da cache L2 no processador e à introdução de uma ligação especial na ponte norte (porto AGP, *Accelerated Graphics Port*) para suportar o apetite voraz da placa gráfica em termos de transferência de dados da memória, evitando as limitações do PCI.

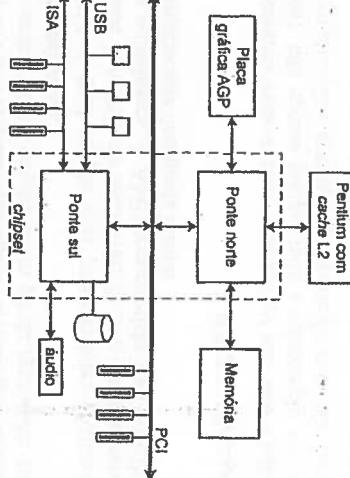


Fig. 6.54 - Evolução da arquitectura do PC, com porto AGP

Mesmo assim, o PCI constituía um estrangulamento na ligação aos periféricos, pelo que a partir de 1999, com o Pentium III, o *chipset* passou a adoptar uma tecnologia baseada em concentradores (*hubs*) directamente interligados entre si (Fig. 6.55), ficando o PCI relegado para um plano lateral, de extensibilidade em termos de placas filhas. O suporte directo para o ISA desapareceu.

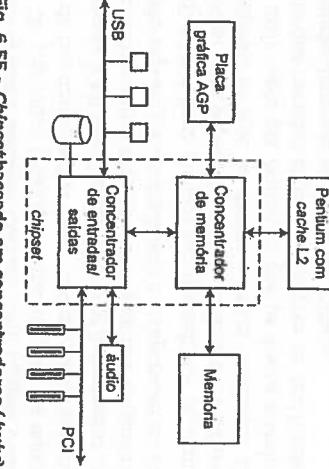


Fig. 6.55 - *Chipset* baseado em concentradores (*hubs*)

A arquitectura baseada nestes concentradores manteve-se actualmente na sua essência. As evoluções mais significativas nos últimos anos foram as seguintes:

- Aumento das frequências de trabalho e suporte para as memórias DDR2 e DDR3;
- Maior integração, com mais funcionalidades integradas no *chipset* (alguns processadores integraram mesmo a ponte norte dentro do processador);

- Ligaçao directa ao concentrador de memória de um controlador de LAN gigabit (tal como já tinha sucedido com o AGP);
- Suporte para hiperfluxo (*hyperthreading*);
- Suporte para processadores com vários núcleos;
- Introdução do PCI-Express (2004), uma ligação série, ponto-a-ponto, que permitiu cerca de 250 MBytes/s de taxa de transferência (500 com o PCI 2.0 em 2007). Estas ligações podem ser emparelhadas, de modo a transmitir mais informação por segundo. O porto AGP foi substituído por 16 canais PCI-Express, num total máximo de 8 GBytes/s. Em 2010, o PCI 3.0 duplicou a taxa de transferência.

No entanto, a AMD optou por integrar o concentrador de memória no próprio processador, o que se traduziu por reduzir substancialmente o estrangulamento na ligação à memória. Apesar de a memória não funcionar mais depressa, a latência é reduzida e o desempenho aumenta. Embora a Intel tenha seguido inicialmente uma via mais tradicional, com os dois concentradores no *chipset*, acabou por também incorporar o concentrador de memória no próprio processador, com o processador Bloomfield, em 2008.

A interligação entre o processador e o resto do sistema, e em particular entre núcleos nos processadores da AMD, é feita por um sistema de ligação chamado HyperTransport, constituído por ligações série, ponto a ponto, que podem ser agregadas até 32 bits para maior desempenho. Mais tarde, este sistema de ligação foi estendido para ligação entre placas e até entre bastidores. Actualmente na sua versão 3.1, é apoiado por um consórcio de vários fabricantes e permite, na sua configuração máxima, uma taxa de transmissão bidireccional de 51,2 GBytes/s. É possível fazer a ligação a barramentos já existentes, como por exemplo o PCI-Express, através de pontes de interface adequadas.

Mais uma vez, a Intel acabou por reconhecer o mérito desta solução e em 2008 (também no processador Bloomfield) lançou o QuickPath, como concorrente do HyperTransport, para substituir o FSB (*Front Side Bus*). O QuickPath consegue apenas 25,6 GBytes/s no máximo, pois tem menos ligações (e este valor já inclui os bits gastos com o protocolo).

### 6.5.3 O MICROCONTROLADOR

#### 6.5.3.1 CARACTERÍSTICAS BÁSICAS

Por muitos milhões de PCs que se vendem anualmente, cerca de 99% de todos os processadores que se vendem destinam-se ao mercado dos sistemas embebidos (secção 5.9.4.2, na página 401), em que, ao contrário dos processadores dos PCs, os aspectos fundamentais não são capacidades e desempenho mas sim integração, custo e em muitos casos consumo. Um automóvel de topo de gama poderia ter na ordem de 100 microcontroladores. Não se vêem, mas elas estão lá, desde o motor até aos controladores dos bancos com ajuste eléctrico, passando por todos os sistemas de visualização, controlo, navegação, protecção (*airbags*, ABS, etc.), conforto, entretenimento, comunicações, etc.

Quantos mais componentes electrónicos (normalmente circuitos integrados) um sistema tiver, mais caro será (em termos de componentes e de montagem) e mais espaço ocupará.

Por esta razão, desde o princípio da era do microprocessador (já com o 80186, em 1982) que se começou a integrar no mesmo circuito integrado não apenas o processador mas também memória (RAM e ROM) e periféricos, constituindo um computador completo num só circuito integrado, que se designa vulgarmente por microcontrolador. Em muitos sistemas, o microcontrolador é o único componente de controlo, ligando directamente a interruptores, LEDs, interfaces de controlo de motores, etc. O programa está armazenado de forma não volátil na ROM (pré-programmada de fábrica se o volume de cópias justificar – tipicamente para cima de 10.000 unidades – ou programada depois de o circuito estar montado, para séries de menor volume).

Cada microcontrolador inclui normalmente, para além do processador propriamente dito, os seguintes recursos:

- ROM Flash, programável por blocos, não tendo que ser toda (re)programmada de uma só vez. Pode ser programada com o microcontrolador montado no circuito, através de pinos e transferência (tipicamente série) a partir de um PC, o que é óptimo em termos de desenvolvimento (Secção 5.9.4, na página 396). Em termos de capacidade, há desde 1 KByte até centenas de KBytes. Estas memórias suportam na ordem de 100.000 reprogrammações;
- RAM, para variáveis e estruturas de dados voláteis. Normalmente é um recurso escasso, com capacidades típicas desde poucos bytes até alguns KBytes;
- EEPROM (ou E<sup>2</sup>PROM), uma ROM programável byte a byte a partir das instruções do microcontrolador, o que contrasta com a ROM Flash, programável apenas a partir do exterior e em grandes blocos de bytes. A E<sup>2</sup>PROM serve essencialmente para guardar dados de forma não volátil. A escrita é lenta (alguns milissegundos), sendo necessário testar se a escrita já foi efectuada antes de tentar nova escrita;
- Controlador de interrupções, com vários pinos disponíveis e possibilidade de programar pinos dos periféricos de entrada/saída para gerar interrupções quando um desses pinos muda de valor. São ainda suportadas várias interrupções internas, geradas pelos vários periféricos;
- Bits de periféricos de entrada/saída. Normalmente organizados em portos de 8 bits, podem variar entre poucos bits até várias dezenas de bits. São normalmente programáveis individualmente como entrada ou saída. É também usual num microcontrolador alguns pinos terem várias funções, entre bits de entrada/saída e outras funcionalidades mais específicas (pinos de uma UART<sup>95</sup>, por exemplo), sendo necessário programar a função que se pretende para esses pinos;
- Vários temporizadores programáveis; podendo gerar interrupções internas quando chegam ao fim da contagem;

<sup>95</sup> Uma UART (*Universal Asynchronous Receiver and Transmitter*) é uma unidade de comunicação adaptada ao protocolo RS-232C. O processador lida com a UART ao nível do byte. A UART transmissora envia cada byte em série, bit a bit, e a UART receptora reconstrói o byte.

- Uma ou mais UARTs para comunicação série, que podem gerar interrupções internas quando precisam de indicar que receberam um byte ou já acabaram a transmissão de um byte;

Outros dispositivos de comunicação entre microcontroladores, como I<sup>2</sup>C ou SPI (*Serial Peripheral Interface*), o que é fundamental em sistemas com vários microcontroladores (como o exemplo do automóvel), em que há normas específicas que permitem implementar uma autêntica rede local;

- Vários canais de conversão analógica/digital (A/D) ou digital/analógica (D/A), para interface com o mundo analógico (tensões contínuas);
- RTC (*Real Time Clock*), um relógio normal (ano, dia, hora, minutos e segundos), geralmente com recurso a um cristal de 32 KHz;

- Cão-de-guarda (*watchdog*), que permite reiniciar automaticamente o microcontrolador caso este deixe de funcionar bem. Para este efeito, o programa tem de efectuar periodicamente operações reconhecidas pelo cão-de-guarda. Se decorrer mais de que um certo tempo (1 segundo, por exemplo) sem essas operações serem efectuadas (por o microcontrolador se ter baralhado, devido a erro do programa ou a ruído electromagnético), o cão-de-guarda reinicia o microcontrolador, que recomeça a execução do programa.

Em termos de extensibilidade, há duas classes fundamentais de microcontroladores:

- Os mais pequenos e de mais baixo custo, que não dispõem de barramentos para o exterior e cujos pinos ligam apenas aos periféricos de entrada e saída, digitais e analógicos, de comunicação, de temporização, etc. Não podem ligar a memórias nem periféricos exteriores e portanto a sua gama de aplicação está limitada às suas próprias capacidades internas, de memórias e de periféricos. Mesmo assim, a panoplia de aplicações para que são adequados é colossal;
- Os destinados a aplicações que exigem mais capacidades, desempenho ou recursos que não se consigam encontrar integrados no microcontrolador, exigindo assim a ligação a hardware externo. Estes microcontroladores não só incluem todos os recursos normalmente encontrados nos microcontroladores mais pequenos, em maior quantidade e variedade, como ainda incluem todos os barramentos necessários para poderem funcionar como processadores e ligar a dispositivos externos.

A Tabela 6.19 ilustra as características de alguns dos microcontroladores existentes no mercado. Há uma grande variedade porque o mercado principal dos microcontroladores é dos sistemas embebidos, com aplicações específicas muito variadas. A inclusão de recursos em excesso significa custo ou consumo a mais, pelo que é necessário adaptar as capacidades do microcontrolador a cada aplicação em concreto.

Muitos microcontroladores ainda são de 8 ou mesmo 4 bits, para terem um baixo custo, embora alguns sistemas embebidos já usem 16 e 32 bits como norma, para maior capacidade de cálculo. As frequências de relógio são normalmente na casa de poucas dezenas de MHz, embora alguns sistemas já usem várias centenas de MHz.

| FABRICANTE E REFERÊNCIA | BITS PROC. | RELÓGIO (MAX) | Nº PINOS | INTERFACE MEMÓRIA | MEMÓRIAS INTERNAS |        | BOTS EXTRAS (MAX) | TEMPO RIZADA DOPAS | COMUNICAÇÕES |        | A/D | I2C |
|-------------------------|------------|---------------|----------|-------------------|-------------------|--------|-------------------|--------------------|--------------|--------|-----|-----|
|                         |            |               |          |                   | ROM               | RAM    |                   |                    | UARTS        | OUTRAS |     |     |
| Atmel MARC 4            | 4          | 8 MHz         | 44       | Não               | 4 KB              | 128 B  | Não               | 34                 | 2            | Não    | Não | Não |
| NEC uPD75P0116          | 4          | 2 MHz         | 44       | Não               | 16 KB             | 256 B  | Não               | 34                 | 4            | Não    | Sim | Não |
| Atmel ATTiny12          | 8          | 6 MHz         | 8        | Não               | 1 KB              | Não    | 64 B              | 6                  | 1            | Não    | Não | Não |
| Microchip PIC10F200     | 8          | 4 MHz         | 6        | Não               | 256 B             | 16 B   | Não               | 4                  | 1            | Não    | Não | Sim |
| Atmel ATmega1280        | 8          | 16 MHz        | 100      | Sim               | 128 KB            | 8 KB   | 4 KB              | 86                 | 6            | 4      | Sim | Sim |
| Intel 80C51             | 8          | 16 MHz        | 40       | Sim               | 8 KB              | 256 B  | Não               | 32                 | 3            | Não    | Não | Não |
| Zilog Z8F6422           | 8          | 20 MHz        | 64       | Não               | 64 KB             | 4 KB   | Não               | 46                 | 4            | 2      | 2   | 12  |
| Philips PXAC37          | 16         | 32 MHz        | 44       | Sim               | 32 KB             | 1 KB   | Não               | 32                 | 3            | 1      | 2   | Não |
| Intel 83C196            | 16         | 40 MHz        | 160      | Sim               | 8 KB              | 4 KB   | Não               | 83                 | 4            | 2      | 1   | 16  |
| Fujitsu MB91F467D       | 16         | 96 MHz        | 208      | Sim               | 1 MB              | 64 KB  | Não               | 192                | 19           | 5      | 6   | 24  |
| Philips LPC2220         | 32         | 75 MHz        | 144      | Sim               | Não               | 64 KB  | Não               | 112                | 4            | 2      | 3   | 8   |
| NEC UPD703111GM         | 32         | 150 MHz       | 176      | Sim               | Não               | 144 KB | Não               | 77                 | 6            | 2      | 1   | 8   |

Tabela 6.19 - Características de alguns microcontroladores disponíveis no mercado



Mesmo os microcontroladores sem interface de memória podem aceder a uma memória externa, mas para tal têm de usar os bits dos portos de entrada/saída para simular os barramentos de dados, endereços e controlo, fazendo evoluir no tempo os sinais de escrita e leitura na memória. Naturalmente, os acessos serão lentos, mas em casos específicos este esquema poderá ser uma opção.

Alguns microcontroladores têm muitos recursos, não apenas em termos de memória mas também em termos de periféricos, enquanto outros são muito mais limitados. É tudo uma questão de necessidade e de custo e de numa dada aplicação usar o microcontrolador mais adequado (modelo de menor custo, consumo, etc., que contemple os recursos e rapidez de processamento necessários).

### 6.5.3.2 CREPE: UM MICROCONTROLADOR BASEADO NO PEPE

O CREPE (Controlador Recomendado Especialmente Para Ensino) é um microcontrolador de 16 bits concebido para ser compatível em termos de instruções com o PEPE, mas permitir construir sistemas embutidos apenas com o microcontrolador, sem necessidade de outros componentes de computador (para além dos dispositivos controlados).

O CREPE não é mais do que um núcleo do PEPE (sem caches nem interface de memória), a que se juntam ROM, RAM e alguns periféricos, tudo dentro do mesmo circuito integrado. O CREPE pertence assim à classe dos microcontroladores mais simples, que não tem interface de memória e portanto não pode ser estendido em termos de recursos, mas é representativo da sua classe e permite simular as aplicações típicas dos sistemas mais simples (que na realidade são os mais numerosos).

A Fig. 6.56 mostra em termos gerais as diferenças de arquitetura entre o PEPE e o CREPE. O conjunto de instruções é igual. O que varia é a forma como os recursos são usados. Um acesso à memória no PEPE, por exemplo, implica uma transferência de dados pela interface de memória. Já no CREPE esse acesso faz-se internamente, com memórias de dados (RAM) e de instruções (ROM) separadas, e o acesso aos periféricos faz-se por meio de um conjunto de registos auxiliares, diferentes dos que existem no PEPE (são os registos internos dos periféricos, que ficam acessíveis directamente ao programador do CREPE), embora a instrução para os usar continue a ser o MOV.

Com o PEPE (Fig. 6.56a), ter o estado de um interruptor ou acender um LED exige a ligação de periféricos de entrada/saída apropriados aos barramentos de dados, de endereços e de controlo, para além do sistema de descodificação de endereços (secção 6.1.4, na página 430). Com o CREPE (Fig. 6.56b), é só ligar o interruptor e o LED directamente ao circuito integrado, e depois é só software.

Os microcontroladores sem interface de memória têm a vantagem de que está tudo feito e é simples. É só ligar os dispositivos a controlar. Em contrapartida, estão limitados aos recursos que oferecem. Os microprocessadores são muito mais flexíveis, permitindo ligar o que for necessário (dentro das limitações do microprocessador, nomeadamente em relação ao barramento de endereços), mas exigem trabalho de interligações.

Os microcontroladores com interface de memória oferecem o melhor dos dois mundos, com recursos internos mas com capacidade de expansão, mas são mais caros, pelo que não se usam nos pequenos sistemas (que também são os que mais facilmente se satisfazem com os recursos oferecidos pelos microcontroladores).

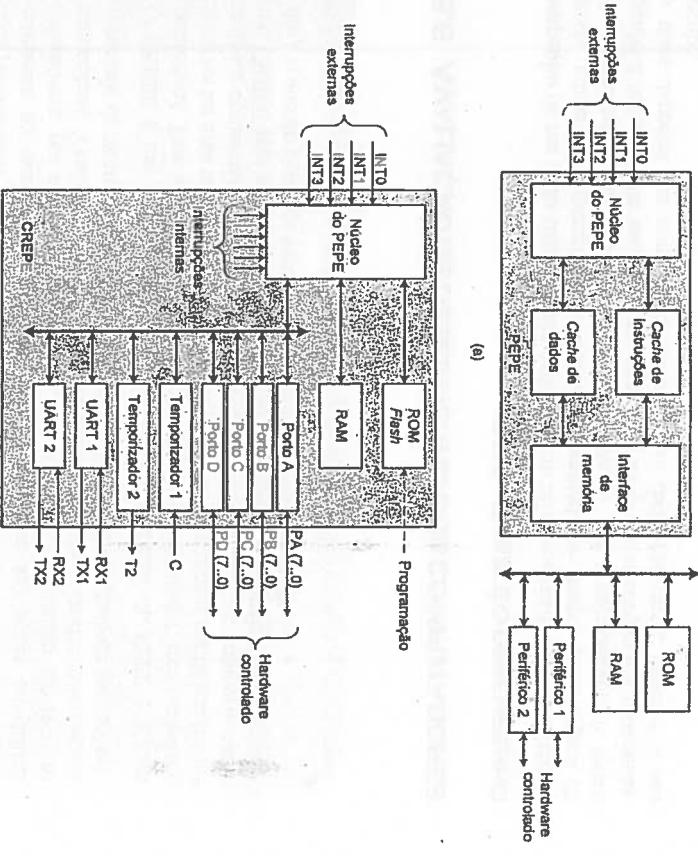


Fig. 6.56 - Arquitecturas genéricas. (a) - Do PEPE; (b) - Do CREPE

**NOTA**

Mesmo os microcontroladores sem interface de memória podem aceder a uma memória externa, mas para tal têm de usar os bits dos portos de entrada/saída para simular os barramentos de dados, endereços e controlo, fazendo evoluir no tempo os sinais de escrita e leitura na memória. Naturalmente, os acessos serão lentos, mas em casos específicos este esquema poderá ser uma opção.

Os recursos oferecidos pelo CREPE são os seguintes:

- 32 KBytes de ROM, mapeados a partir do endereço 0000H, que é onde o CREPE começa a sua execução após uma inicialização (tal como o PEPE). Num microcontrolador a sério, seria ROM Flash e haveria pinos para a programar, cartando um programa via sistema de desenvolvimento. No simulador, em que o CREPE só existe em software, o único mecanismo disponível é o carregamento de programas através da leitura de um ficheiro. No entanto, qualquer escrita (por MOV) nesta gama de endereços não altera o valor da célula de memória acedida;

- 32 KBytes de RAM, mapeados na segunda metade do espaço de endereçamento do CREPE (que é de 64 KBytes, tal como o PEPE). Nada impede o carregamento de programas em RAM (especificando directivas PLACE adequadas);

Quatro portos de entrada/saída de 8 bits cada, com os pinos PA, PB, PC e PD. Cada um é configurável de forma independente para entrada ou saída, mesmo durante o funcionamento (o que permite tornar os portos bidireccionais). Também é possível gerar uma interrupção interna sempre que um dos bits de um porto de entrada mude de valor. Os bits de cada porto não são configuráveis individualmente;

Dois temporizadores, constituídos por contadores de 16 bits cuja frequência de relógio e valor de fim de contagem é programável. Ambos geram uma exceção quando chegam ao fim da contagem. O temporizador 1 tem ainda um modo cronômetro (contagem de tempo enquanto o pino C estiver a 1) e o temporizador 2 permite gerar formas de onda quadradas de frequência programável no pino T2; Duas UARTs, com os pinos RX1, TX1, RX2 e TX2, úteis na comunicação série com outros CREPEs, por exemplo. Na comunicação, o programa precisa de saber quando chegou um byte ou quando um byte acaba de ser enviado. Tal pode ser feito por teste (*polling*) de bits de estado num dos registos auxiliares ou por geração de interrupções internas.

Esta é apenas uma descrição sumária do CREPE. O Apêndice B, na página 711, contém a descrição em detalhe e um exemplo de utilização.

Em termos de implementação, existem as seguintes alternativas:

- Simulador** – O CREPE é um módulo do simulador, tal como PEPE, e pode ligar-se a dispositivos periféricos (LEDs, interruptores, etc.);
- Placa de microcontrolador** – Utiliza-se uma placa com um microcontrolador comercial para simular o funcionamento do CREPE, cujos dispositivos internos se mapeiam nos dispositivos reais desse microcontrolador. O objectivo é simular o CREPE em termos reais (podendo interagir com periféricos reais e não apenas simulados). Esta placa pode ser utilizada em dois regimes:
  - Desenvolvimento** – O PC continua a executar o simulador e o módulo do CREPE, mas pode escolher quais os periféricos que funcionam no módulo e na placa. Desta forma, o CREPE pode interagir tanto com periféricos simulados como com periféricos reais. Uma das UARTs do microcontrolador comercial é usada para ligar ao PC e a sua ROM Flash possui apenas um programa para interagir com o PC;
  - Produção** – O programa do CREPE é carregado na ROM Flash do microcontrolador comercial. Desta forma, consegue-se implementar um sistema autônomo, com programa em memória não volátil, tal como se o CREPE existisse enquanto microcontrolador real.

#### ESSENCIAL

Existe uma grande diversidade de computadores, mas em que podemos identificar as principais classes: supercomputadores, servidores, cestos de trabalho, computadores de secretaria, portátiles, de mão e microcontroladores. Estes computadores destinam-se a classes de aplicações distintas:

- A arquitetura do PC é, longe, a mais divulgada e tem evoluído no sentido de aumentar o desempenho (de computação, gráfico, de comunicação, etc.). Os grandes computadores já se baseiam, em grande maioria, na arquitetura do PC. A tendência actual está na utilização de processadores com vários núcleos;
- Os microcontroladores são computadores completos num só circuito integrado, com processador, memórias (RAM e ROM) e periféricos. Destinam-se especialmente ao mercado dos sistemas embutidos, para aplicações específicas (telemoveis, por exemplo), em que o custo, tamanho e consumo são factores primordiais. Vendem-se em muito maior número que os restantes sistemas e a variedade de microcontroladores existentes reflecte a diversidade de aplicações;
- O CREPE é um microcontrolador, que inclui um núcleo do PEPE e alguns periféricos. Tem uma implementação em software no simulador, mas também é possível ter uma implementação em hardware real, com uma placa de microcontrolador comercial, que interpreta as instruções do CREPE e se compra como um CREPE real a interagir com dispositivos reais (e não apenas num simulador), o que do ponto de vista laboratorial é sempre interessante.

## 6.6 AVALIAÇÃO DE DESEMPENHO DOS COMPUTADORES

### 6.6.1 O QUE É O DESEMPENHO

O desempenho de um dado sistema pode definir-se em relação a muitos factores (tempo de resposta, número de operações efectuadas por unidade de tempo, percentagem de operações efectuadas com sucesso, etc.). No contexto dos computadores, o termo desempenho é normalmente associado ao inverso do tempo de execução dos programas. Quanto mais depressa um computador executar um dado programa, maior é o seu desempenho.

Existe ainda uma implementação em linguagem C do interpretador de instruções do CREPE que pode ser portada para qualquer sistema comercial, havendo apenas necessidade de adaptar as partes que têm a ver com o hardware específico desse sistema.

Se um computador A executar um programa no tempo  $T_A$  e um computador B executar o mesmo programa no tempo  $T_B$ , então pode definir-se, em igualdade de circunstâncias<sup>96</sup>, para este programa em particular, um desempenho relativo dado por:

$$\frac{\text{Desempenho B}}{\text{Desempenho A}} = \frac{T_A}{T_B}$$

Há constantemente uma procura insaciável de desempenho. Não basta um computador funcionar. Tem de o fazer o mais rápido possível. Os programas são cada vez mais complexos e hoje em dia fazem coisas que são possíveis apenas porque os computadores conseguem executar milhares de milhões de operações por segundo. As evoluções têm sido tecnológicas, com relógios de frequência cada vez maior, mas também têm havido muitos melhoramentos a nível da arquitectura (organização interna) dos computadores, que permitem aproveitar melhor cada ciclo de relógio.

Qualquer computador tem vários blocos constituintes, cada um afectando o desempenho desse computador de modo diferente, e não há dois computadores iguais, pois cada fabricante de computadores tem as suas próprias soluções, o que levanta dois problemas fundamentais:

**PROBLEMA 1** Dados dois computadores diferentes, como é que se podem comparar para saber qual é o mais rápido, ou qual o seu desempenho relativo?

**PROBLEMA 2** Num dado computador, como é que se pode saber qual o impacte de cada um dos seus blocos constituintes no desempenho global, ou dito de outra forma, quanto é que se ganha no desempenho global melhorando X% o desempenho desse bloco?

Neste contexto, o que interessa medir é o desempenho global, visto pelos programas, pelo que tanto o hardware como o compilador têm impacte no tempo de execução dos programas. Assim, os componentes fundamentais que podem ser objecto de avaliação desse desempenho são os seguintes:

- Processador – O elemento mais rápido do sistema, em que se pretende maximizar a frequência do relógio e minimizar o número médio de ciclos de relógio por instrução;

Memória – Normalmente mais lenta que o processador, a questão fundamental é conseguir minimizar o tempo médio de acesso à memória;

Periféricos – Geralmente, o factor mais lento do desempenho e por conseguinte um dos mais determinantes nos programas que os usam intensamente;

Compilador – O que está em causa é a qualidade do código gerado; em termos das instruções usadas e da gestão de recursos, nomeadamente os registos.

O tempo  $T$  que um programa demora a executar num computador pode ser obtido por:

$$T = T_{\text{proc}} + T_{\text{mem}} + T_{\text{per}}$$

en que:

- $T_{\text{proc}}$  – Tempo gasto pelo processador em processamento interno, sem aceder à memória;
- $T_{\text{mem}}$  – Tempo gasto pelo processador nos acessos à memória, incluindo a busca de instruções;
- $T_{\text{per}}$  – Tempo gasto pelo processador nos acessos aos periféricos, incluindo o tempo de espera (sem poder prosseguir) que uma operação sobre um periférico se complete.

Um computador real procura não apenas minimizar cada um destes tempos mas também, tanto quanto possível, sobrepor-lhos para que o tempo de execução seja inferior ao seu somatório. Exemplos:

- Execução simultânea ou parcialmente sobreposta de várias instruções no processador, desde que não haja dependências (o operando de uma instrução ser o resultado doutra). Este aspecto é tratado na secção 7.3, na página 595;
  - Carregamento numa memória mais rápida (cache, secção 7.5, na página 622) não apenas da instrução ou dado de que se precisa num dado momento mas também de instruções ou dados em endereços contíguos, numa antecipação de que não serão necessários a seguir (o que estatisticamente é muito frequente);
  - Execução de outros programas nos tempos mortos à espera de uma operação sobre um periférico (secção 7.7, na página 664).
- As secções seguintes dão resposta aos dois problemas atrás enunciados e analisam o impacte de cada um dos quatro componentes identificados, mas para já sem nehum destas optimizações, que são tratadas nas secções referidas.

## 6.6.2 PROGRAMAS DE AVALIAÇÃO (BENCHMARKS)

Quando se precisa escolher um computador, poderá haver a tendência para olhar para a frequência de relógio do processador, num raciocínio de que maior frequência implica maior desempenho. Não é necessariamente verdade, pois há computadores que demoram em média mais ciclos de relógio a executar uma instrução do que outros e esta métrica não tem em conta os restantes factores, como o compilador ou os periféricos.

Existe também a métrica simplista de usar MIPS (Milhões de Instruções Por Segundo)<sup>97</sup>. Só que normalmente o fabricante divulga o valor de MIPS assumindo que o processador executa repetidamente a instrução mais rápida (tipicamente o NOP), sem ter em conta as restantes instruções, o que indica apenas quanto rápido um computador consegue fazer

<sup>96</sup> Nomeadamente, assumindo que ambos os computadores estão exclusivamente a executar este programa.

<sup>97</sup> Como se pode ver, MIPS não é o plural de MIP...

coisa nenhuma. Outras instruções demoram mais ciclos de instrução a ser executadas do que o NOP, pelo que tem de se fazer uma média ponderada pela frequência relativa de ocorrência de cada instrução no programa.

Também se usa o termo MFLOPS (*Millions of Floating-Point Operations Per Second*)<sup>98</sup> para indicar o número máximo de operações de vírgula flutuante (com valores não inteiros – ver Apêndice D, na página 735) que o processador consegue executar por segundo. Estas operações são importantes em programas de cálculo intensivo, mas mais uma vez estamos a falar de valores máximos, inatingíveis com programas reais, para além de que este factor ignora o facto de os programas não serem exclusivamente constituídos por operações de vírgula flutuante.

Para complicar o cenário, um computador não é apenas processador, memória e algoritmos. A generalidade dos programas usa também os periféricos, em particular os que envolvem comunicação por rede ou acesso a bases de dados.

Nítidamente, o que interessa é avaliar os computadores como um todo, em cenário real com programas reais. Uma solução é executar o(s) programa(s) que o utilizador normalmente usa em cada um dos computadores a testar e comparar os tempos de execução. O problema é que normalmente isto não é praticável (ninguém quer comprar vários computadores só para experimentar e nem sempre as empresas que os vendem os disponibilizam para os testes dos compradores).

A alternativa é definir programas de avaliação normalizados (*benchmarks*), que cada um dos fabricantes pode testar nos seus computadores, e divulgar os tempos de execução. Isto permite ter um termo de comparação entre computadores sem implicar comprá-los primeiro.

Os *benchmarks* existem já há muitos anos. Em 1976 apareceu o Whetstone e em 1984 o Dhrystone. Trata-se de pequenos programas, não aplicações realistas mas sim conjuntos de instruções escolhidas com base na frequência estatística de ocorrência dessas instruções em programas reais (razão pela qual se chamam "sintéticos" a estes *benchmarks*).

O desempenho de um computador é medido em Whetstones/segundo ou Dhrystones/segundo, ou seja, o número de vezes que um computador consegue executar o *benchmark* por segundo. O grande problema destes *benchmarks* é, por um lado, poderem não ser representativos das características dos programas do utilizador e, por outro, serem tão pequenos e executarem tão rápido que pequenos factores que se diluiriam num programa grande acabam por assumir um papel preponderante.<sup>99</sup>

<sup>98</sup> ... nem MFLOPS é plural de MFLOP.

<sup>99</sup> Para além do facto de que alguns fabricantes já optimizavam artificialmente as arquitecturas e os compiladores para melhor executarem estes *benchmarks*...

Em 1988, várias empresas formaram a SPEC<sup>100</sup> (*System Performance Evaluation Cooperative*), que passou a usar uma média do desempenho de vários programas reais como *benchmark* segundo várias perspectivas, incluindo desempenho do processador, processamento gráfico, supercomputação, aplicações Java, sistemas de cliente-servidor, sistemas de ficheiros e servidores Web. Cada *benchmark* é composto por um conjunto de programas numa dada classe de aplicações e destina-se a comparar o desempenho dos vários sistemas de computadores nessas classes de aplicações, desde computação pura em memória até processamento de dados e servidores de informação. A SPEC disponibiliza os dados sobre os valores obtidos para cada *benchmark* para inúmeros sistemas.

As aplicações de processamento de dados são na realidade as mais usadas no mercado empresarial. A unidade de processamento destas aplicações é usualmente a transacção, um conjunto de operações encadeadas que envolvem não apenas processamento por parte do processador mas também acessos a periféricos, nomeadamente redes ou bases de dados. No caso de uma transacção afectar o estado do sistema, tem de garantir cumulativamente que:

- Não pode ser interrompida por outra transacção em execução concorrente;
- As alterações da transacção não podem ficar a meio devido a uma falha de rede, por exemplo (ou completa a sequência de operações ou não altera nada, podendo neste caso ter de desfazer operações já efectuadas).

© TPC<sup>101</sup> (*Transaction Processing Council*) apareceu em 1992 com o objectivo de criar vários *benchmarks* para ambiente transactional, incluindo acesso a bases de dados, com ou sem optimizações, e a servidores Web. Nas transacções, os factores importantes são o ritmo de execução de transacções e a escalabilidade, estando em causa não apenas o computador em si mas também todo o software de suporte, incluindo sistema operativo, sistema de gestão de base de dados e servidor.

### 6.6.3 A LEI DE AMDAHL

Os *benchmarks* permitem comparar o desempenho de dois computadores, englobando todos os factores que contribuem para esse desempenho, mas não indicam de que forma cada factor contribui, nem aqueles que mais afectam o desempenho e que portanto são os que mais interessam optimizar.

A lei de Amdahl indica a melhoria que se pode esperar no tempo  $T$  de execução de um programa num dado computador, se se melhorar de  $N$  vezes apenas alguns dos factores que afectam esse desempenho:

$$\text{Melhorado} = \frac{T_{afetado}}{N} + T_{não\ afetado}$$

<sup>100</sup> [www.spec.org](http://www.spec.org)

<sup>101</sup> [www.tpc.org](http://www.tpc.org)

O tempo de execução do programa melhora apenas na parte afectada pelo melhoramento.

A parte que não depende desse melhoramento continua a demorar o mesmo tempo.

Esta lei é muito importante porque diz que não se pode esperar que o tempo de execução se reduza proporcionalmente à melhoria de apenas parte do programa. Para dar um exemplo, consideremos a fórmula já usada na página 543 para exprimir o tempo de execução de um programa:

$$T = T_{proc} + T_{mem} + T_{per}$$

e assumamos que  $T_{proc} = 10$  seg,  $T_{mem} = 20$  seg,  $T_{per} = 70$  seg. Com estes tempos, este programa de exemplo gasta 10% em processamento interno do processador, 20% nos acessos à memória e 70% nos acessos aos periféricos. Se passarmos a frequência de relógio do processador para o dobro, e assumindo que tal não tem impacte noutras factores, o tempo gasto pelo processador em processamento interno será metade e o tempo de execução melhorado do programa passará a ser:

$$T_{melhorado} = \frac{T_{proc}}{2} + T_{mem} + T_{per}$$

ou

$$T_{melhorado} = 5 + 20 + 70 = 95 \text{ seg}$$

A melhoria global do desempenho, medida pelo rácio dos tempos de execução, foi de apenas:

$$\frac{T}{T_{melhorado}} = \frac{100}{95} = 1,053$$

Ou seja, a melhoria do tempo de execução do programa foi pouco mais de 5%, apesar de o tempo gasto pelo processador se ter reduzido para metade. A diferença resulta de o tempo que realmente foi melhorado ocupar apenas uma pequena fração do tempo total de execução.

Por outro lado, se esta redução para metade tivesse sido feita no tempo gasto com os periféricos, a melhoria global teria sido mais de 50%:

$$\frac{T}{T_{melhorado}} = \frac{100}{10 + 20 + 35} = 1,538$$

Portanto, a lei de Amdahl tem duas consequências muito importantes:

- A melhoria global é inferior à melhoria de apenas um (ou alguns) dos factores aditivos que afectam o tempo de execução;
- O impacte da melhoria de um factor é tanto maior quanto maior for o peso desse factor no tempo total de execução. Dito por outras palavras, o que vale a pena é optimizar os casos mais frequentes ou mais relevantes.

$$T_{proc} = \frac{N \times \sum_{i=1}^K P_i \times C_{mi}}{F} = \frac{\text{ciclos}}{\text{ciclos/segundo}} = \frac{5000000 \times (0,35 \times 2 + 0,4 \times 1 + 0,25 \times 3)}{1000000} = 9,25 \text{ seg.}$$

en que:

- F – Frequência do relógio do processador, medida em ciclos/segundo;
- N – Número de instruções do programa (excluindo as gastas com acessos à memória e aos periféricos);
- $C_{mi}$  – Número de ciclos de relógio para a instrução i. Umas instruções são mais complexas que outras e demoram mais ciclos de relógio a executar que as outras.

Para facilitar a análise, também se pode usar esta equação considerando não instruções individuais mas classes de instruções (saltos, por exemplo).

$$T_{proc} = \frac{N \times \sum_{i=1}^K P_i \times C_{mi}}{F} = \frac{N \times D}{F} = \frac{\text{ciclos}}{\text{ciclos/segundo}}$$

en que (o significado de N e F mantém-se):

- K – Número de classes de instruções consideradas no programa;
- $P_i$  – Percentagem média de ocorrência de instruções desta classe no programa;
- $C_{mi}$  – Número médio de ciclos de relógio para a classe de instruções i;
- D – Média pesada (pela percentagem de ocorrência) do número de ciclos de relógio de cada instrução.

Se, apenas a título de exemplo, considerarmos um programa com 5 milhões de instruções (excluindo acessos à memória e periféricos), divididas em três classes (aritméticas, transferências entre registos e controlo), com percentagem de ocorrência de 35%, 40% e 25% e número de ciclos/instrução médio (excluindo a busca das instruções) de 2, 1 e 3, respectivamente, a executar num processador com frequência de relógio de 1 MHz (um milhão de ciclos por segundo), o tempo de execução interna do processador será, previsivelmente, na ordem de:

$$T_{proc} = \frac{N \times \sum_{i=1}^K P_i \times C_{mi}}{F} = \frac{\text{ciclos}}{\text{ciclos/segundo}} = \frac{5000000 \times (0,35 \times 2 + 0,4 \times 1 + 0,25 \times 3)}{1000000} = 9,25 \text{ seg.}$$

ou

$$T_{proc} = \frac{N \times D}{F} = \frac{5000\,000 \times (0,35 \times 2 + 0,4 \times 1 + 0,25 \times 3)}{1\,000\,000} = \frac{5000\,000 \times 1,85}{1\,000\,000} = 9,25 \text{ seg}$$

Neste exemplo, cada instrução demora, em média, 1,85 ciclos de relógio a ser executada. O mesmo programa, expresso numa linguagem de alto nível, poderá demorar mais ou menos tempo noutra computador em que o número de ciclos/instrução médio (D) de cada classe seja diferente ou em que o respectivo compilador gere um conjunto de instruções diferente (afectando NxD). No caso geral, o tempo de execução de um programa num computador é afectado essencialmente pelos factores descritos na Tabela 6.20.

| FATOR                       | AFFECTA | MODO COMO INFLUENCIA O DESEMPENHO DO PROCESSADOR                                                                                                                                                                                                                           |
|-----------------------------|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Algoritmo                   | N, D    | Os algoritmos mais eficientes implicam menos operações a efectuar, logo menos instruções                                                                                                                                                                                   |
| Linguagem de programação    | N, D    | Linguagens de programação de alto nível implicam operações mais abstractas, mas demoram de implementar, mas envolvem menos esforço de programação/manutenção. A linguagem assembly é a mais eficiente, mas o esforço de programação só é admisível para pequenos programas |
| Compilador                  | N, D    | O compilador pode fazer mais ou menos optimizações, gerir os recursos (registos, por exemplo) de forma mais ou menos eficiente, gerar sequências de instruções que gastam mais ou menos ciclos de relógio (seção 6.5.6)                                                    |
| Conjunto de instruções      | N, D, F | Um computador pode ter instruções complexas mas que demorem mais ciclos a executar, ou instruções mais simples e mais rápidas a executar mas necessitam em maior número (seção 6.5.7)                                                                                      |
| Arquitectura do processador | D, F    | Usando algumas técnicas, nomeadamente processamento em estágios (seção 7.3, na página 595), é possível reduzir em média o número de ciclos de relógio para cada instrução/classe de instruções                                                                             |
| Frequência do relógio       | F       | É sobretudo um factor com um limite tecnológico, mas também depende da arquitectura do processador, pois arquitecturas mais simples permitem frequências de relógio mais rápidas (seção 6.5.7)                                                                             |

Tabela 6.20 - Factores que influenciam o tempo de execução dos programas (considerando apenas o tempo de execução no processador).

Este livro preocupa-se essencialmente com o conjunto de instruções e a arquitectura do processador. O objectivo básico em termos de desempenho é um processador ter o menor número possível de ciclos/instrução médio e oferecer um conjunto de instruções que, em conjunto com o compilador, permita reduzir globalmente o número de ciclos de relógio a executar por um programa. Note-se que estes factores acabam também por influenciar a frequência de relógio máxima possível. Uma arquitectura complexa introduz mais atrasos nos circuitos, o que limita a frequência (todos os circuitos têm de ter tempo de reagir antes do próximo ciclo de relógio).

### 6.6.5 AVALIAÇÃO DO DESEMPENHO DA MEMÓRIA

A memória constituiu sempre um factor de estrangulamento do desempenho, pelo facto de um acesso a uma célula de memória (externa ao processador) demorar bastante mais

tempo do que um acesso a um registo (interno ao processador). Tal como referido na secção 6.5.2.3, o tempo de acesso às memórias não tem acompanhado a evolução da frequência de trabalho dos processadores porque estas têm aumentado muito a sua capacidade.

O grande truque neste aspecto tem sido o uso de *caches* dentro do próprio processador. Uma *cache* é uma memória de uso estatístico, contendo apenas as palavras mais usadas pelo processador, pelo que pode ser bastante mais pequena que toda a memória principal. Se a palavra com o endereço accedido estiver na *cache*, o tempo de acesso é pouco mais que o acesso a um registo. Se não estiver, tem de fazer o acesso à memória principal e demora mais tempo (mas no próximo acesso a palavra já lá estará).

Dentro dos parâmetros normais, verifica-se que na esmagadora maioria dos acessos tipicamente mais do que 95%) as palavras pretendidas estão na *cache*, o que evita um acesso à memória principal, externa ao processador. Colocada dentro do processador, a *cache* tem um tempo de acesso muito mais rápido do que a memória principal, o que permite um tempo médio de acesso muito inferior. As *caches* são detalhadas na secção 5, na página 622.

**NOTA:** As *caches*, e em particular a de nível 2 (L2), ocupam uma enorme fatia da área da pastilha de silício dos microprocessadores de alto desempenho, o que se traduz no custo do circuito integrado. Numa tentativa de baixar o custo e competir com a concorrência de outros fabricantes como a AMD, a Intel lançou os Pentiums Celeron em Abril de 1998, sem *cache* L2 e com 7,5 milhões de transistores. O efeito da ausência da *cache* é que os acessos têm de ser feitos à memória principal com muito mais frequência, o que piora o tempo médio de acesso. O fraco desempenho determinou uma má aceitação pelo mercado, e apenas quatro meses depois, em Agosto desse mesmo ano, a Intel lançou os Celeron com 128 KBytes de *cache* L2 e 19 milhões de transistores. Mesmo assim, bastante menos que o Pentium II Xeon, lançado em Junho de 1998 em duas versões, com 512 KBytes e 1 MByte de *cache* L2. Em 2000, a AMD lançou o Duron para competir com o Celeron, com apenas 64 KBytes de *cache* L2, o suficiente para o desempenho ser aceitável mas ser bastante atrativo em preço face ao Celeron.

No entanto, as *caches* não são só por si uma solução completa, pois a memória principal continua a desempenhar um papel fundamental, em particular quando se consideram periféricos que envolvem transferências de dados massivas, como a placa gráfica e a interface de rede, em que o DMA é fundamental.

Interligação entre o processador, a memória principal e os periféricos mais exigentes tem merecido a atenção dos fabricantes, manifestada sobretudo através dos desenvolvimentos referidos na secção 6.5.2.3.

O impacte do desempenho da memória, em que o modo de raiada é fundamental, depende muito da aplicação, e pode ser analisado quer do ponto de vista do processador (incluindo a cache), quer do ponto de vista das operações de DMA com periféricos. Como exemplo, e para se perceber a diferença de desempenho entre as várias formas de aceder à memória, considere-se um programa que faz 100 leituras a palavras de memória em

endereços consecutivos, copiando-as para outros 100 endereços consecutivos, nas seguintes condições:

- O tempo de acesso a uma palavra de memória, se estiver na *cache*, é de 1 ns. Se não estiver, é o tempo de acesso à memória principal.
- A *cache* está inicialmente vazia;

O tempo de acesso à memória principal é de 40 ns para um acesso individual ou primeiro acesso de uma rajada (endereços consecutivos), em que cada acesso a seguir ao primeiro em modo de rajada demora 2 ns;

Cada cópia de uma palavra (leitura + escrita) é feita por uma iteração de um ciclo de 8 instruções (incluindo as instruções de acesso à memória), num total de 100 iterações.

Nestas condições, o tempo  $T_{mem}$  (um dos termos da equação da página 543) gasto pelo processador nos acessos à memória para conseguir realizar estas 100 leituras e 100 escritas (sem entrar em linha de conta com o tempo de execução das instruções que não acedem à memória, mas contando com o tempo gasto com a sua busca da memória) é obtido da seguinte forma:

Acessos individuais (um a um) do processador, por instruções de acesso à memória, admitindo que não há *cache*. O tempo gasto pelo processador em acessos à memória (8 buscas das instruções + leitura + escrita) será:

$$T_{mem} = 100 \times ((8 + 2) \times 40\text{ ns}) = 40\,000\text{ ns}$$

*Idem*, mas com *cache*. Neste caso, a primeira busca de cada instrução tem de ser feita na memória principal, mas na segunda iteração do ciclo as instruções já estão na *cache*, pelo que a busca já é muito mais rápida. Por outro lado, os acessos às 100+100 palavras não aproveitam a *cache*, pois todos os endereços são diferentes, pelo que o tempo gasto com os acessos à memória será:

$$T_{mem} = 8 \times (1 \times 40\text{ ns} + 99 \times 1\text{ ns}) + 2 \times 100 \times 40\text{ ns} = 9112\text{ ns}$$

Acessos por um controlador de DMA, em que por simplicidade se ignoram as instruções de programação do controlador de DMA (executadas apenas uma vez) e se contabiliza apenas o tempo em que o processador está parado à espera que a operação de DMA acabe (assumindo que o processador não continua em processamento interno durante esta operação, senão o tempo perdido pelo processador ainda será menor). Em DMA os acessos à memória funcionam em rajada (quer lendo primeiro as palavras todas para memória interna do controlador e escrevendo-as todas depois quer tendo dum lado a escrevendo do outro palavra a palavra), mas em rajada), pelo que o tempo gasto será:

$$T_{mem} = 2 \times ((1 \times 40\text{ ns}) + (99 \times 2\text{ ns})) = 476\text{ ns}$$

Embora naturalmente as relações entre estes tempos dependam do programa usado, é notida a vantagem da *cache* (23% do tempo gasto) e a diferença entre transferências por software, com acessos individuais a cada palavra, e por hardware, em rajada (5% do

tempo gasto, mesmo com *cache*). Por isso, não basta ter um processador com uma elevada frequência de relógio. O desempenho global do sistema depende muito do suporte em hardware no acesso à memória.

## 6.6.6 O IMPACTE DO COMPILADOR

A principal função de um compilador (Fig. 5.14, na página 391) é converter uma sequência de caracteres (texto simples) que representa instruções de um programa, de acordo com as regras de uma dada linguagem, em código-máquina (instruções em binário que o hardware sabe executar directamente). Os *assemblers* são também compiladores, tal como a linguagem *assembly* é uma linguagem de programação, embora de baixo nível. Quanto mais alto é o nível da linguagem de programação, mais longe está do hardware e mais perto do nível do programador chega (Fig. 1.4, na página 13), o que significa que mais trabalho o compilador tem que ter. No entanto, mais margem de manobra tem para fazer optimizações e para melhor gerir os recursos disponíveis (registros e outras unidades internas do processador, memória, etc.).

Em linguagem *assembly*, o programador especifica quase tudo e o *assembler* praticamente só tem de fazer um mapeamento quase directo para as instruções do código-máquina. Numa linguagem de alto nível, é o compilador que decide que instruções do processador gerar, que registos deve usar, como organizar as estruturas de dados, quando aceder à memória e em que endereços, como invocar rotinas e passar parâmetros, etc.

O compilador é assim um elemento fundamental no desempenho de um computador. O mesmo programa compilado por dois compiladores diferentes e executado no mesmo computador pode proporcionar tempos de execução significativamente distintos, dependendo das optimizações feitas e do grau de conhecimento que o compilador tem da arquitetura em que o programa vai ser executado.

Isto significa que os *benchmarks* avaliam não apenas o *hardware* em si mas todo o conjunto, incluindo compilador e software de sistema (biblioteca de funções da linguagem de programação, sistema operativo, plataforma de suporte, etc.).

Está fora do âmbito deste livro explicar o funcionamento de um compilador e a forma como ele optimiza e gera o código-máquina. Referem-se aqui apenas algumas das optimizações mais frequentes, apenas a título de exemplo e sem pretender uma descrição exaustiva (a ordem não indica importância de optimização):

- Desenvolver de ciclos, substituindo ciclos com poucas iterações por cópias separadas das instruções de cada iteração. Gasta mais memória, mas evita os testes e os saltos inerentes às iterações;
- Invocação de rotinas em linha. Em vez de invocar uma rotina, insere-se uma cópia das instruções da função. Evita as instruções de chamada e retorno e as passagens de parâmetros e do resultado. Normalmente, só compensa no caso de rotinas pequenas;

- Reutilização de registos, usando registos com resultados de uma expressão ou rotina como os registos com os parâmetros de entrada para outra expressão ou rotina (evitando cópias de valores);
- Substituição de instruções de multiplicação/divisão por uma potência de 2 por instruções de deslocamento à esquerda/direita, que normalmente são de execução mais rápida;
- Eliminação de ocupação de registos quando já não são usados, ou de variáveis ou instruções/rotinas que não são usadas;
- Troca de ordem de execução de instruções, quando tal não afectar a semântica do programa, o que permite depois fazer outras optimizações ou passar para fora de um ciclo instruções cuja execução é independente das iterações e produz o mesmo valor em cada iteração (e assim só se faz uma vez);
- Resolução em software de aspectos que têm a ver com o funcionamento dos diversos blocos internos do processador, nomeadamente no que respeita ao processamento em estágios (secção 7.3, na página 595);
- Substituição de expressões nos índices das matrizes (*arrays*) por cálculos com apontadores, que normalmente são mais eficientes;
- Escolha de instruções mais adequadas de acordo com os valores de constantes. Por exemplo, a instrução ADD no PEPE só suporta constantes entre -8 e +7. Se o valor estiver fora deste intervalo, o código gerado já tem de usar um registo adicional para conter esse valor. O compilador tem de saber isto para poder optimizar para o caso das constantes pequenas.
- O programador é mais inteligente do que um compilador e consegue detectar situações específicas de optimização que um compilador pode não reconhecer. Mas é menos fiável, cometendo frequentemente mais erros, e os compiladores actuais incluem inúmeras optimizações que em termos globais compensam perdas em casos mais rebuscados.
- A programação em linguagem *assembly*, como alternativa à programação em alto nível, não se justifica em termos gerais, pois dá muito mais trabalho e potencia muito mais erros que saem muito caros em termos de desenvolvimento. Assim, a programação em linguagem *assembly* deve estar reservada para rotinas pequenas, simples e críticas em termos de desempenho, o que normalmente acontece apenas em rotinas de muito baixo nível de um sistema operativo ou de interacção com periféricos (*device drivers*).

### 6.6.7 A FILOSOFIA RISC

Historicamente, os processadores eram concebidos por equipas que se preocupavam essencialmente com o *hardware*, tornando-o o mais completo possível, na assunção de que quanto mais funcionalidade o *hardware* tivesse (mais instruções e mais completas):

- Mais fácil seria para o compilador gerar código (o *hardware* estaria mais perto do nível das linguagens de programação de alto nível);

- Maior desempenho o computador teria (as operações implementadas directamente em *hardware* são mais eficientes do que executar uma rotina).

O expoente máximo desta filosofia foi a arquitectura VAX, da Digital Equipment Corporation, que em 1977 permitia que ambos os operandos das instruções pudesssem estar quer em registos quer na memória, com um número demasiado elevado de instruções, modos de endereçamento e dimensão das instruções variável (entre 1 e 53 bytes!), de acordo com os parâmetros que recebiam. Esta preocupação de reduzir o tamanho do código derivava do custo das memórias, que na altura era elevado.

Outra vertente consistiu na desenho de arquitecturas de alto nível (*High Level Language*, ou HLL), que subiam ainda mais o nível do *hardware*, tentando aproximar-lo de uma linguagem de programação de alto nível que lhes servia de modelo. Apareceram implementações em *hardware* para linguagens de programação como Lisp, Smalltalk-80, C e Java.

No entanto, a complicação do *hardware* resultante desta funcionalidade acrescida implicava circuitos muito complexos, o que impedia a frequência do relógio de subir, e arquitecturas em que mesmas operações elementares demoravam vários ciclos de relógio a executar. Num VAX, a média do número de ciclos de relógio por instrução era na ordem de 10.

Por outro lado, a funcionalidade das instruções era projectada a pensar no caso mais complexo, para servir para a generalidade dos casos, sem ter em conta a frequência com que cada caso ocorria. Deste modo, mesmo os casos mais simples (que se vieram a revelar os mais frequentes) ficavam mais lentos e executavam mais operações do que o necessário.

Também as arquitecturas HLL, em especial as das linguagens mais evoluídas, tendiam a asear-se muito na pilha para armazenamento de valores intermédios e passagens de parâmetros, fazendo um uso intensivo da memória em detrimento dos registos, com a penalização correspondente em termos de tempo de acesso.

Pode dizer-se que nessa altura os engenheiros estavam mais preocupados com a funcionalidade do que com optimizações e desempenho. Também os compiladores não estavam tão desenvolvidos, pelo que o principal cavalo de batalha era tornar o *hardware* mais complexo e poderoso.

No final da década de 70, início da de 80, apareceram alguns projectos que seguiram a via oposta, com *hardware* simples, poucas instruções, todas de igual dimensão (32 bits), poucos modos de endereçamento, processamento em estágios e exposição da arquitectura ao compilador, que era responsável pela resolução dos problemas levantados por arquitecturas tão pouco dotadas de recursos: o IBM 801 (1975), RISC-I (1980) e MIPS (1981).

Estas arquitecturas tinham de executar na ordem de duas vezes mais instruções que o VAX, uma vez que a funcionalidade era mais rudimentar, mas conseguiam executar em média uma instrução em menos de dois ciclos de relógio, o que, tendo em conta que o VAX demorava em média 10 ciclos de relógio por cada instrução, ainda fazia com que o

VAX fosse cerca de três vezes mais lento para a mesma frequência de relógio. E isto com arquitecturas muito mais simples, mais baratas e com maior potencial de subida da frequência do relógio.

Dada a simplicidade, estas arquitecturas ficaram conhecidas como RISC (*Reduced Instruction Set Computers*) e em retrospectiva as arquitecturas como o VAX passaram a chamar-se CISC (*Complex Instruction Set Computers*), grupo do qual a arquitectura IA-32, a que serve de base aos Pentiums, é um membro destacado (o 8086, que está na sua base, apareceu em 1978, e por razões de compatibilidade perdurou até hoje).

Como produtos comerciais RISC apareceram arquitecturas como MIPS (usado até 2006 nas estações de trabalho da Silicon Graphics Inc.), o Power-PC (que serviu de base aos computadores da Apple até esta mudar para a Intel, e hoje continua com a designação de base, POWER, apenas em servidores), o PA-RISC (usado até 2008 em servidores da Hewlett-Packard) e o UltraSPARC (base dos servidores da Sun Microsystems).

No entanto, mais importante que uma pureza de linha ou filosofia, o importante é reconhecer que a principal lição da filosofia RISC é que o caso mais frequente deve ser mais otimizado, e que a complexidade só é boa enquanto for realmente útil e não tornar simples.

O PEPE foi também desenhado segundo a filosofia RISC, o que se pode traduzir fundamentalmente pelas seguintes regras:

- Instruções do código-máquina todas do mesmo tamanho e com o menor número possível de formatos, para simplificar a descodificação das instruções na unidade de controlo;
- Acesso à memória apenas por instruções de transferência de dados (MOV). Nunca por instruções de processamento de dados (ADD, por exemplo);
- Suporte para poucos modos de endereçamento, assumindo os parâmetros das instruções sempre em registos;
- Um número generoso de registos, para minimizar os acessos à memória;
- Optimização de casos frequentes. No caso do PEPE, um exemplo consiste no par de instruções CALLF e RETF, que evitam usar a pilha para invocar uma rotina que não chame outra (secção 5.7.2, na página 317);
- Optimização da arquitectura para que cada instrução, em média, seja executada no menor número de ciclos de relógio possível, para o que é fundamental o processamento em estágios (secção 7.3, na página 595)...

Em termos de mercado, as arquitecturas CISC foram progressivamente desaparecendo incorporando as lições RISC. O caso surpreendente e paradigmático é a arquitectura IA-32, que é a base dos Pentiums. Embora seja CISC de raiz, foi progressivamente incorporando mecanismos que aprendeu dos seus primos RISC (como por exemplo o processamento com um número significativo de estágios), com vista ao aumento do desempenho. Usando o nível de investimento só possível com base no quase monopólio do mercado

dos computadores pessoais, as empresas que mais a desenvolvem (Intel e AMD) levaram os microprocessadores baseados nesta arquitectura a um nível de desempenho tal que, longe de desaparecer, está na base da evolução para os 64 bits que poderá talvez (o futuro o dirá) ditar o triunfo sobre as outras arquitecturas RISC de raiz.

Esta evolução para os 64 bits processou-se essencialmente da seguinte forma:

- A AMD enveredou pela arquitectura AMD64, que é totalmente compatível com a IA-32, com apenas algumas alterações para suportar os 64 bits. Actualmente, os processadores Athlon e Opteron implementam esta arquitectura;
- A Intel acabou por ter de adoptar as extensões da AMD para 64 bits, sob a designação Intel 64, expressa nas arquitecturas Netburst, Core e Nehalem;
- A Intel produziu também uma nova arquitectura (IA-64), usada apenas no Itanium, que é incompatible em código binário com a IA-32, e que se baseia na execução de várias instruções em paralelo, sob controlo do compilador (que poderá trocar a ordem de execução de instruções se tal não afectar a semântica do programa). Os programas já existentes têm de ser recompilados.

A experiência passada mostra que quando se introduz uma nova arquitectura que é incompatible ao nível do código-máquina com a anterior surgem problemas que podem ditar o fracasso (e a Intel já tem algumas experiências neste campo). O desenvolvimento do Itanium tem sofrido muitos atrasos e o seu desempenho, embora de topo, acaba por sofrer a concorrência de outros processadores compatíveis com o 8086 da própria Intel (não obstante, o Nehalem-EX), sob constante pressão da AMD. Embora a Intel tenha já divulgado alguma informação sobre os sucessores do Itanium, com lançamentos previstos para 2012 (Poulson) e 2014 (Kitton), só um dos grandes fabricantes produz actualmente computadores baseados no Itanium (a Hewlett Packard) e a Microsoft e a Red Hat já anunciaram o fim do suporte a novos desenvolvimentos para o Itanium.

### 6.6.8 AVALIAÇÃO DO DESEMPENHO DOS PERIFÉRICOS

Os periféricos de interesse do ponto de vista do desempenho não são os simples portos de entrada/saída mas sim os periféricos complexos, como os discos, as redes e as placas gráficas. Não está em causa apenas o hardware, mas também toda uma panóplia de camadas de software, desde as rotinas de mais baixo nível (*device drivers*) até às funções mais complexas de todo o sistema operativo, plataformas de suporte e aplicações. A informação é normalmente lida num lado, eventualmente processada e escrita e/ou transmitida para outro lado. Do ponto de vista do desempenho, o factor mais relevante é o analise completa do desempenho de um sistema de periféricos sem dados de um sistema real ou de uma simulação desse sistema. No entanto, é possível obter uma avaliação aproximada, efectuando simplificações ao funcionamento do sistema para permitir alguns cálculos estatísticos. Esta secção usa esta última via para estabelecer algumas conclusões. Vejamos o funcionamento de um disco magnético. Um dos factores mais importantes é o tempo de acesso, normalmente na ordem dos milissegundos, uma eternidade para os

processadores actuais que conseguem executar vários milhões de instruções durante esse tempo. Quanto tempo demora, em média, um acesso de leitura a um sector de um disco, desde que o processador faz o pedido até esse sector estar lido em memória e pronto para ser processado?

A secção 6.3.2, na página 483, contém os detalhes do funcionamento dos discos relevantes para responder a esta pergunta, considerando que o tempo de acesso é o somatório do tempo de procura de pista (*seek time*), dado pelo fabricante, do atraso rotacional médio (meia rotação) e do tempo de transferência do sector, normalmente desprezável face aos outros dois.

Essa secção dá um exemplo para um disco de 100 GBytes, com um tempo de acesso médio na ordem de 10 milisegundos e um tempo de transferência na ordem de 0,016 ms, o que significa que cerca de 99,8% do tempo de acesso é gasto à espera. Com um processador de 1 GHz, isto corresponde a cerca de 10 milhões de ciclos de relógio. A uma média de dois ciclos de relógio por instrução, por exemplo, o processador pode executar 5 milhões de instruções enquanto espera que o sector esteja disponível em memória, após uma operação de DMA (a leitura propriamente dita). É fácil, portanto, constatar o peso que os acessos ao disco têm no desempenho dos programas.

Outra forma mais transaccional de olhar para este exemplo é reconhecer que no máximo o sistema consegue fazer cerca de 100 transacções por segundo (se cada uma envolver apenas um acesso ao disco, o que demora cerca de 10 ms).

Por outro lado, também é importante reconhecer que os programas acedem a ficheiros, constituídos normalmente por vários blocos, cada um com vários sectores contíguos. Portanto, o tempo de espera é comum para todos os sectores de um bloco, que podem ser lidos em acessos seguidos sem novo posicionamento da cabeça magnética.

No entanto, os ficheiros estão partidos em blocos, normalmente espalhados pelo disco, o que provoca vários acessos em que o tempo de procura de pista pode pesar. Esta é a razão pela qual periodicamente se deve compactar um disco (em particular se tiver pouco espaço livre, muito fragmentado), o que envolve colocar os vários blocos de um ficheiro de forma contígua num disco de forma a reduzir os tempos de procura nos acessos aos vários blocos.

Uma aplicação que envolvia o envio de informação pela rede poderá ter os seguintes factores a condicionar o tempo total gasto numa comunicação:

- Tempo de acesso à informação (poderá envolver uma ou mais leituras do disco);
- Tempo de processamento local (normalmente desprezável, mas pode ser importante se os dados tiverem muito processamento – compressão, cifra, etc.);
- Tempo de comunicação (tal como no acesso aos discos, inclui latência e tempo de transmissão).

Normalmente o factor limitativo é o disco, mas uma rede lenta pode estrangular a comunicação. Como exemplo, consideremos um servidor Web a que vários clientes fazem

pedidos de páginas HTML. Alguns clientes estão na rede local do servidor, mas outros fazem acessos remotos via Internet. Suponhamos as seguintes condições:

- A rede local funciona a 100 Mbit/s de taxa de transferência máxima, mas a ligação à Internet não vai além de 1 Mbit/s. Assume-se que a rede tem uma latência de 10.000 bits (isto é, em média cada mensagem tem de esperar o tempo equivalente à transmissão de 10.000 bits antes de ser enviada):

As páginas HTML têm uma dimensão média de 8 KBytes e assume-se que todos os bytes de cada página estão totalmente contíguos no disco;

O disco tem um tempo típico de procura de pista de 4 ms, gira a 10.000 rpm e tem uma taxa de transferência de 50 MBytes/s;

Os barramentos de entrada/saída do computador têm capacidade suficiente para não constituir uma limitação.

Nestas condições, qual o tempo médio de obtenção de uma página a partir do disco?

- O tempo médio de achar o início da página:
- 4 ms (*seek time*) para achar a pista no disco;
- 3 ms (tempo de meia volta, em média).

O tempo médio de leitura de uma página HTML é de 8 KBytes/50 MBytes/s  $\equiv$  0,16 ms;

Resposta final =  $7 + 0,16 = 7,16$  ms (na prática, este tempo ignora uma série de pequenos factores, como os atrasos introduzidos pelo controlador do disco e pelo controlador de DMA. No entanto, todos estes factores são residuais face ao tempo de espera até a informação começar a ser lida do disco).

Supondo pedidos sequenciais (sem sobreposição), quantos pedidos de páginas exclusivamente por parte de clientes na rede local consegue o servidor satisfazer por segundo?

- Cada página demora cerca de 7,16 ms a ser lida de disco;
- O tempo de transmissão de cada página através da rede é o somatório de um tempo de espera de cerca de 10.000 bits (a um ritmo de transmissão de 100 Mbit/s) e de  $(8 \times 8)$  Kbytes/100 Mbit/s para ser enviada, ou cerca de 0,1 ms de espera mais 0,64 ms para ser enviada, ou ainda 0,74 ms de tempo total de comunicação;
- O tempo total de serviço será em média,  $7,16 + 0,74 = 7,9$  ms, o que permite servir cerca de 126 pedidos por segundo ( $1\text{ s} / 7,9\text{ ms}$ );
- O factor limitativo, neste caso, é o disco.

Supondo pedidos sequenciais (sem sobreposição), quantos pedidos de páginas exclusivamente por parte de clientes na Internet consegue o servidor satisfazer por segundo?

- Cada página demora cerca de 7,16 ms a ser lida de disco;
- O tempo de transmissão de cada página através da rede é o somatório de um tempo de espera de cerca de 10.000 bits (a um ritmo de transmissão de 1 Mbit/s)

e de (8x8 K) bits/1 Mbit/s para ser enviada, ou cerca de 10 ms de espera mais 64 ms para ser enviada, ou ainda 74 ms de tempo total de comunicação;

- O tempo total de serviço será em média,  $7,16 + 74 \approx 81$  ms, o que permite servir cerca de 12 pedidos por segundo (1 s / 81 ms);

O factor limitativo, neste caso, é a rede.

Há várias formas de aumentar o desempenho de um servidor Web. Por exemplo:

- Usar vários discos (RAID – *Redundant Array of Independent Disks*), de forma a suportar vários acessos simultâneos a páginas. Os RAID substituem um grande disco num servidor por uma bateria de discos mais pequenos e baratos. Usam-se várias técnicas (níveis), como por exemplo, guardar cada página em dois discos diferentes (RAID nível 1), lendo-se a página do disco que na altura do acesso levar a um menor tempo de espera (o que tiver a cabeça mais perto dos sectores pretendidos). No entanto, como isto duplica o espaço necessário, o nível mais usado em servidores de informação, incluindo servidores de Web, é o RAID nível 5, que distribui os dados por vários discos e optimiza o acesso de vários pedidos simultâneos (e portanto o número de transacções por segundo);
- Usar *caches* de páginas (que mantêm em memória principal as páginas mais acedidas, ficando estas com um acesso muito mais rápido);
- Melhorar a taxa de comunicação da rede.

As conclusões a tirar sobre o desempenho de um sistema de periféricos, sem medição real através de *benchmarks*, dependem do realismo e precisão do modelo usado e das aplicações a executar, seja em simulação, seja num raciocínio simples como o que se faz nesta secção. Às vezes tiram-se medidas num sistema para tentar extrapolar para outro.

O objectivo fundamental de qualquer planeamento ou avaliação de um sistema de periféricos é identificar quais os estrangulamentos que condicionam tudo o resto, e optimizar essencialmente esses casos, de forma a maximizar a melhoria do desempenho de todo o sistema.

Não adianta ter componentes individuais do sistema com elevado desempenho se depois existem outros que estrangulam o percurso dos dados e fazem o resto do sistema esperar. Desta forma, o mais importante é ter um sistema equilibrado, em que a largura de banda de cada componente é adequada para o contexto em que está inserido. As sucessivas evoluções da arquitectura do PC (Fig. 6.52 até à Fig. 6.55) foram essencialmente neste sentido. Os periféricos mais rápidos, como a placa gráfica, ficaram no barramento do sistema. Os restantes ficaram em barramentos secundários, mais lentos.

**ESSENCIAL**

- O desempenho de um computador mede-se geralmente em relação ao tempo de execução (inversamente proporcional);

- MIPS e MFLOPS medem apenas o máximo possível de operações por segundo mas não exprimem o desempenho real. Para este efeito usam-se *benchmarks*, constituídos tipicamente por um conjunto de aplicações reais em que se faz uma média dos tempos de execução e se produz um número que serve depois de termo de comparação com outros computadores;
- Os *benchmarks* medem o desempenho de um computador como um todo. A lei de Amdahl indica a melhoria que se pode esperar no desempenho de um programa quando se melhora o desempenho de apenas um dos componentes que afetam esse desempenho;
- Há várias classes de *benchmarks*, consoante o tipo de aplicações que se pretende executar (algoritmos em memória, uso intensivo dos periféricos, etc.);

- Os processadores têm melhorado o seu desempenho à custa não apenas da tecnologia (aumento da frequência do relógio) mas também da arquitectura (redução do número de ciclos de relógio por instrução, um dos princípios base da filosofia RISC, que revolucionou as arquitecturas de computadores);
- As *caches* são fundamentais para o desempenho de um computador, pois são muito mais rápidas do que a memória principal (cujo tempo de acesso tem diminuído pouco pois a sua capacidade tem aumentado muito). Os processadores de maior desempenho incluem já vários MBytes de cache;
- O tempo de acesso à um disco magnético (na ordem de alguns milisegundos, mesmo para os discos mais rápidos) é totalmente dominado pela espera do posicionamento da cabeça magnética sobre a pista pretendida e da rotação do disco até o sector pretendido passar junto à cabeça. Em comparação, o tempo de transferência dos dados é insignificante;
- As redes locais de computadores já funcionam a 10 GHz. As placas gráficas, com vídeos e animações 3D, necessitam de taxas de transferência de dados da memória na ordem dos GB/s. Estes periféricos têm ligações especiais ao barramento de sistema, de modo a suportar estes ritmos. Os periféricos mais lentos ligam a barramentos secundários, mais lentos;

- O segredo do desempenho está em evitar pontos de estrangulamento no caminho crítico dos dados, mais importantes. O desempenho de um sistema é apenas tão bom quanto o do elo de menor desempenho da cadeia. O objectivo de qualquer sistema é equilibrar o desempenho dos vários componentes de um computador, o que pode variar de acordo com o contexto de utilização do computador.

## 6.7 CONCLUSÕES

Este capítulo apresentou essencialmente a visão de sistema de um computador, com ênfase na interacção dos vários componentes, em particular os periféricos, uma vez que o processador, com exemplos de programação envolvendo apenas a memória, já tinha sido objecto dos capítulos anteriores.

Para além do conjunto de instruções, há todo um conjunto de aspectos fundamentais para um projectista de um sistema ou simplesmente para perceber e saber usar um sistema já feito, nomeadamente:

- Endereçamento, envolvendo essencialmente:
  - O mapa de endereços (em que gama de endereços se pode aceder a cada um dos dispositivos, memórias ou periféricos)
  - A descodificação de endereços (como seleccionar o dispositivo correcto, e apenas esse);
  - A dicotomia endereçamento em *byte/palavra*, que tem grandes implicações a nível de hardware e software.
  - Ciclos de acesso do processador à memória e periféricos, com os respectivos tempos de acesso e temporizações do processador;
  - Excepções e as suas vantagens nas interacções com os periféricos (interrupções) e processamento de situações anormais causadas pelo programa;
  - Classes de computadores, que apesar de obedecerem todos à mesma arquitectura de base diferem na sua estrutura em termos de processador, memórias, periféricos e outros componentes (nomeadamente estruturas de interligação), de forma a estarem mais adequados à aplicação em vista;
  - Desempenho do computador, em que o importante não é analisar cada componente de forma isolada mas sim o comportamento global de todo o sistema, devendo identificarse os componentes que mais contribuem para o estrangulamento do desempenho e optimizar-se esses componentes.

Desde o aparecimento da arquitectura básica de von Neumann, há já 60 anos, a evolução dos computadores tem sido suportada pelo avanço tecnológico da electrónica, que permite uma integração cada vez maior de componentes, mais rápidos e com maior capacidade, mas tal seria muito limitado se não tivesse sido acompanhado por uma miriade de técnicas ao nível organizacional que permitem tirar partido da tecnologia para produzir computadores que fazem cada vez mais e melhor.

## 6.8 EXERCÍCIOS

- 6.1** Considere um computador com base no PEPE com uma RAM com 2 KBytes de capacidade cujo primeiro byte está no endereço A800H. Qual é o endereço (em hexadecimal) do seu último byte?

## 6.0 COMPUTADOR COMPLETO

**6.2** Suponha que num dado instante R1=1234H, R2=100H e SP=1000H. Indique o endereço em que o byte 34H é escrito quando se executa a seguinte instrução:

- a) MOV [R2+6], R1  
c) SWAP [R2], R1

- b) MOVB [R2], R1  
d) PUSH R1

**6.3** Suponha que não sabe se o PEPE é big-endian ou little-endian. Faça um programa para o PEPE que permita descobrir qual destas hipóteses está correcta.

**6.4** Suponha que declarou

```
PLACE 50H
STRING "Computador", 00H
```

A directiva STRING usa ASCII para converter os caracteres em bytes. Mostre qual o conteúdo da memória gerado por esta directiva, byte a byte, a partir do endereço 1000H para os seguintes processadores (todos suportam endereçamento de byte):

- a) PEPE;  
b) PEPE-8;  
c) Processador de 16 bits, little-endian;  
d) Processador de 32 bits, big-endian;  
e) Processador de 32 bits, little-endian.

**6.5** A instrução MOV R1, [R2] não demora o mesmo tempo a executar do que a instrução MOV R1, R2. Porque? Faça um programa com um ciclo que execute N vezes uma destas instruções e outro idêntico mas com a outra instrução, para verificar qual é mais lento. Execute os dois no simulador. Use um relógio de tempo real para o PEPE com período de 50 ms, ajuste N para o tempo total de execução do programa com MOV R1, R2 ser na ordem de 30 segundos e defina um ponto de paragem (*breakpoint*) para detectar o fim do programa. Se dividir a diferença de tempos de execução dos programas por N obtém a diferença de tempos de execução das duas instruções? Justifique a resposta.

**6.6** Idem, mas agora com ADD R1, R2 e MUL R1, R2. Sabendo que uma multiplicação

fazida pelo método manual (Fig. 2.52, na página 103) é substancialmente mais complexa do que uma adição, dê uma explicação para a diferença de tempos de execução dos dois programas.

**6.7** Considere o computador da Fig. 6.57, em que tanto a ROM como a RAM têm 16 bits de largura mas estão divididos em duas metades (de 8 bits de largura cada) para suportar o endereçamento de byte (as capacidades indicadas dizem respeito apenas a cada metade). Cada periférico tem apenas um porto, de 8 bits de largura.

- a) Que bits do barramento de endereços devem ligar à ROM? E à RAM?  
b) Porque é que os periféricos não ligam ao barramento de endereços?  
c) Explique o esquema de descodificação de endereçamento de byte e de palavra, respeita ao suporte para endereçamento de byte e de palavra;

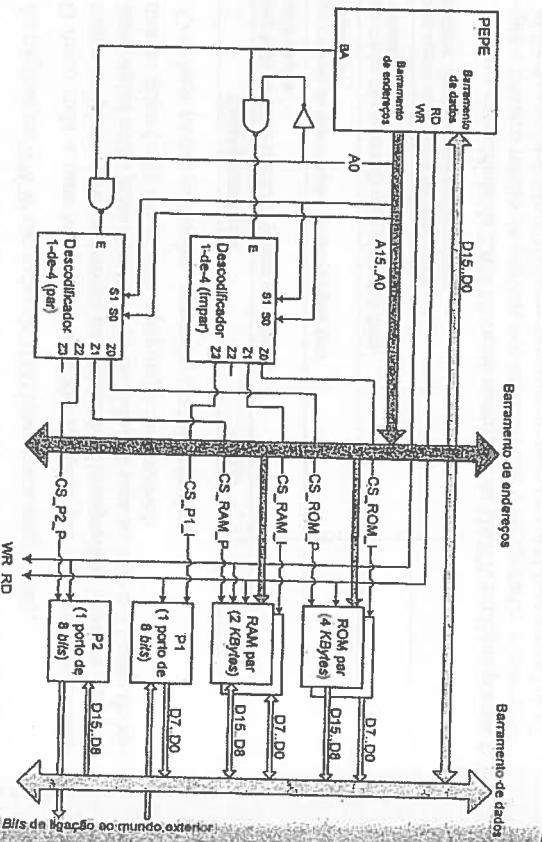


Fig. 6.57 - Arquitectura de um computador baseado no PEPE

- d) Suponha que a ROM deve estar mapeada a partir do endereço 0000H e a RAM a partir de 2000H. A que bits do barramento de endereço devem ligar os bits S0 e S1 dos descodificadores 1-de-4? Justifique.
- e) Imagine que o processador executa a instrução MOV [R1], R2, com R1=3000H. A que dispositivo (se algum) é que a instrução está a aceder? E se R1=8000H?
- f) Preencha a tabela seguinte com o valor dos sinais durante o acesso à memória de cada instrução indicada e o valor do R2 e memória indicada após a execução de cada instrução. A execução das instruções é sequencial (não são independentes).
- | R1    | Instrução    | A0 | BA | RD | WR | A15-A0 | D15-D8 | D7..D0 | R2 | M[2000H] |
|-------|--------------|----|----|----|----|--------|--------|--------|----|----------|
| 2000H | MOV [R1], R2 |    |    |    |    |        |        |        |    | 1234H    |
| 2000H | MOV [R1], R2 |    |    |    |    |        |        |        |    | 1234H    |
| 2000H | MOV R2, [R1] |    |    |    |    |        |        |        |    |          |
| 2000H | MOV R2, [R1] |    |    |    |    |        |        |        |    |          |
| 2001H | MOV R2, [R1] |    |    |    |    |        |        |        |    |          |
| 2001H | MOV R2, [R1] |    |    |    |    |        |        |        |    |          |
- g) Destas instruções, indique quais são adequadas para aceder a cada um dos portos e com que valor de R1.

### 6.8 No simulador, monte um computador com o PEPE (o da Fig. 6.57, por exemplo)

e coloque LEDs em todos os bits do periférico de saída. Escreva um programa que leia sequencialmente várias bytes de RAM e as escreva no periférico de saída, voltando depois ao princípio. No simulador, inicialize manualmente estas células de RAM, com valores adequados para formar padrões de evolução dos LEDs ao longo do tempo (por exemplo, um só LED ligado que circula da direita para a esquerda e depois ao contrário). Implemente ainda o que for necessário para que o ritmo de evolução dos LEDs seja rápido (5 mudanças por segundo) e regular.

### 6.9 No simulador, monte um computador com o PEPE (o da Fig. 6.57, por exemplo)

e ligue um periférico de saída a um descodificador e mostrador de sete segmentos (Fig. 2.14, na página 52). Use um relógio normal para o PEPE, sem ser de tempo real. Faça um programa para ir mostrando sucessivamente cada dígito hexadecimal de 0 a F, em ciclo. Implemente o tempo de duração de cada dígito (de aproximadamente 1 segundo) das formas seguintes (discuta cada um dos métodos em termos de flexibilidade de programação e precisão temporal):

- Com um ciclo em software;
- Por leitura de um periférico de entrada ligado a um relógio de tempo real de período adequado;
- Por interrupções geradas por um relógio de tempo real ligado a um dos pinos de interrupção (sensível ao flanco, configuração por omissão).

### 6.10 Usando o registo RCN (Tabela 6.13, na página 464), mude a sensibilidade do pino de interrupção de flanco para nível (0 ou 1, tanto faz) no exercício 6.9c e explique a diferença de comportamento no mostrador.

### 6.11 Imagine um computador que digitaliza som, a um ritmo de 10.000 amostras/segundo. Cada amostra resultante é enviada para outro computador por meio de uma UART, que utiliza comunicação série assíncrona (página 493), com bit de paridade e dois stop bits, a um ritmo de 150.000 bits/segundo. Para enviar uma amostra, o processador escreve o valor na UART e espera que esta indique que já o acabou de enviar, o que é feito por meio de uma interrupção. Nessa altura, o processador envia nova amostra.

- Qual o ritmo máximo de amostras que é possível enviar pela UART?
- A primeira amostra tem de ser enviada explicitamente, senão nenhuma amostra é enviada. Porque?
- De quanto em quanto tempo, em média, é que o PEPE recebe uma interrupção da UART?
- Qual o tempo máximo disponível para processamento de cada amostra, assumindo que tudo o resto (digitalização, interrupção da UART, escrita na UART, etc.) é instantâneo?
- Nas condições da alínea d), que fração do tempo do processador fica para outras tarefas se o processamento de cada amostra for 70 microssegundos?

**6.12** Considere o Programa 6.4, na página 511, e a Fig. 6.50. Assuma que o PEPE demora 3 ciclos de relógio a executar uma instrução (incluindo a busca), excepto se for uma leitura de memória (*mov*), em que gasta 4, ou uma escrita, em que preceisa de 5. Suponha ainda que o controlador de DMA precisa de 1 ciclo de relógio para fazer uma leitura da memória e 2 para fazer uma escrita.

- a) A rotina de interrupção do Programa 6.4 tem de guardar o estado do PEPE (PC, RE e outros registos, já na rotina), o que não acontece numa transferência de DMA. Explique porque;

- b) Estime o tempo gasto numa transferência de um sector de 1024 bytes de um disco para memória, tanto por DMA (em bloco) como por software (com o Programa 6.4). Ignore os tempos gastos pelo PEPE na programação do controlador de DMA. Indique aproximadamente o ganho de tempo (em ciclos de relógio) nesta transferência proporcionado pelo DMA.

**6.13** Suponha que dois computadores clientes ligados a um servidor, um por uma linha série com uma UART e outro por uma LAN (rede local, com *ethernet*), fazem um pedido de uma mesma página WWW a esse servidor. Este deve efectuar uma transferência de dados entre periféricos (leitura do disco e escrita no periférico de comunicação, UART ou LAN). Assuma que:

- A UART usa um bit de paridade (para detecção de erros) e dois STOP bits, a uma taxa de transmissão de 10.000 bits/segundo;
- A LAN transmite dados a 100.000 bits/segundo, em que por cada 100 bytes de dados transmitidos são enviados mais 50 bytes para controlo da transmissão;
- As páginas de WWW são todas de 1000 bytes (para simplificar);
- O disco roda a uma velocidade de 6000 rotações por minuto, o posicionamento na pista correcta demora em média 5 milisegundos, todos os bytes de uma página estão juntos no disco (e são lidos de uma só vez) e o tempo de leitura do disco (uma vez achada a página) é desprezável.

- a) Com base nestas assunções, preencha a tabela seguinte:

| CARACTERÍSTICA                                                                      | UART | LAN |
|-------------------------------------------------------------------------------------|------|-----|
| Número total de bits a transmitir (dados + controlo)                                |      |     |
| Tempo mínimo de transmissão de uma página (em milissegundos)                        |      |     |
| Tempo médio de procura de uma página no disco (em milissegundos)                    |      |     |
| Número médio de páginas por segundo que o computador cliente poderá ler do servidor |      |     |

- b) Sendo a capacidade da LAN 10 vezes superior à da UART, explique porque é que o número médio de páginas por segundo que o computador cliente poderá ler do servidor não é 10 vezes superior no caso da LAN face ao da UART.

## 7 - O PROCESSADOR EM DETALHE

Os capítulos anteriores descreveram o PEPE sobretudo de um ponto de vista externo, de quem o utiliza para construir um computador e programar aplicações básicas. No entanto, e apesar de o PEPE ser um microprocessador muito simples face aos que existem no mercado, há ainda muitos aspectos a descobrir, que não influenciam de forma aparente o desempenho e na sua funcionalidade.

Primeiro que tudo, é preciso perceber como é que os vários componentes internos de um processador conseguem cooperar para implementar o conjunto de instruções, não apenas do ponto de vista da execução estrita das instruções, nomeadamente em termos do processamento de dados, mas também como é que se estabelece o controlo e o sequenciamento das várias operações elementares (microinstruções) necessárias.

Um aspecto sempre fundamental é o desempenho do processador, que executa muitas operações de forma sequencial. Aqui a ideia é processar as instruções como numa linha de montagem industrial, usando processamento em estágios (*pipelining*), em que várias instruções estão em execução simultânea, embora em estágios de evolução diferentes. A existência de uma só memória (quer para dados, quer para instruções) e de uma só ligação (quer para leitura, quer para escrita) limita severamente o desempenho do processador. A solução é incluir pequenas memórias dentro do próprio processador, designadas *caches*, cujo acesso é muito mais rápido que o da memória externa, com a vantagem de existirem duas (uma para instruções, outra para dados) e com entrada e saída separadas. O princípio básico é que devem armazenar apenas as células mais usadas (a sua capacidade tem de ser muito menor do que a da memória externa, senão não cabem dentro do processador), mas tal implica um funcionamento automático que é preciso implementar.

Por outro lado, os programas são cada vez mais complexos e nem sempre a memória tem capacidade necessária, o que implica usar uma hierarquia de memória, distinguindo entre memória principal (a que o processador pode aceder directamente) e memória de massa (disco magnético, tipicamente), de muito maior capacidade mas de acesso demasiado lento para suportar o acesso directo por parte do processador. É preciso delinear um mecanismo (designado memória virtual) para gerir esta hierarquia de forma automática.

Finalmente, é preciso reconhecer a importância do sistema operativo como programa middleware, no próprio processador, para as suas funcionalidades básicas.

Existem ainda muitos outros temas, mais especializados ou avançados, que por limitações de espaço não são abordados neste livro. Recomenda-se [Patterson 2008, Stalling 2006].

## 7.1 DIAGRAMA DE BLOCOS GERAIS

A Fig. 4.2, na página 188, estabelece o diagrama de blocos básicos para um processador com um banco de registos, mas com algumas limitações, nomeadamente em termos de controlo (cada instrução tem de ser executada num só ciclo de relógio, o que não permite a implementação de algumas instruções mais complexas) e de acesso à memória (exige memórias de dados e de instruções separadas, cada uma com entrada e saída distintas).

A Fig. 4.3, na página 191, ilustra o princípio que permite usar uma só memória externa com apenas uma ligação de dados (quer para escrita, quer para leitura), ao mesmo tempo que oferece uma vista de duas memórias separadas ao processador propriamente dito. A Fig. 7.1 representa o diagrama de blocos geral do PEPE e detalha a Fig. 4.3, incorporando basicamente tudo o que foi explicado nos capítulos anteriores e satisfazendo os respectivos requisitos. Embora mais completo do que o processador simples da Fig. 4.2, na página 188, pode facilmente reconhecer-se, na parte de cima da figura, o banco de registos, a ALU e os multiplexers que controlam o caminho dos dados.

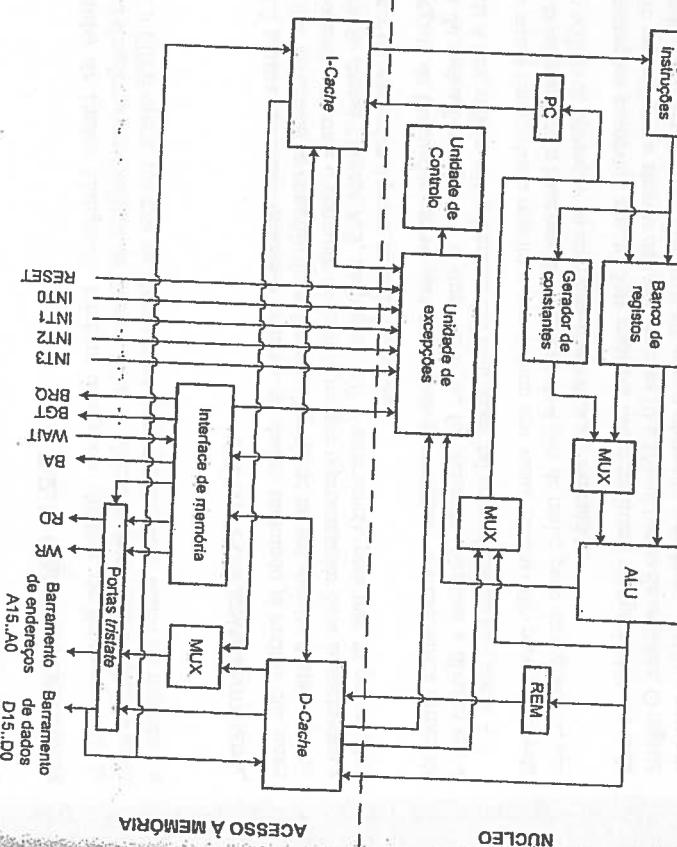


Fig. 7.1 - Diagrama geral do PEPE

O PEPE é constituído essencialmente pelos seguintes blocos, cujo funcionamento é detalhado nas secções seguintes:

- RI (Registo de Instruções) – Onde cada instrução é armazenada após ter sido lida da cache de instruções;
- PC (Program Counter) – Vai sendo incrementado de 2 em 2 e é usado para indicar em que endereço deve ser lida a próxima instrução;
- Banco de Registros (secção 7.2.1.2) – Contém os registos principais, que servem para armazenar quer os operandos das instruções quer o resultado de muitas das instruções. Contém ainda alguns registos auxiliares, tipicamente com funções de configuração de alguns dos blocos do PEPE;
- Gerador de Constantes (secção 7.2.1.3) – Permite gerar constantes de 16 bits a partir da informação (com 4, 8 ou 12 bits) codificada nas instruções. Assim, consegue-se codificar algumas constantes de 16 bits, que até são as mais usadas, dentro de uma instrução de apenas 16 bits;
- Unidade Aritmética e Lógica – Vulgarmente designada ALU (secção 7.2.1.4), responsável pela transformação dos dados;
- Unidade de Controlo (secção 7.2.3) – Cérebro do sistema e responsável pelo sequenciamento de todos os sinais que controlam o processador;
- Unidade de Excepções (secção 7.2.2) – Recebe informação sobre os vários pontos do processador onde podem ocorrer situações anómalias e assinala à Unidade de Controlo a existência de excepções (secção 6.2, na página 459);
- Registo de Endereço de Memória (REM) – Usado para memorizar os vários endereços de memória em acesso de dados. Não faz parte do banco de registos e é necessário porque em algumas instruções o endereço da célula de memória a aceder tem de ser calculado por uma operação da ALU (mov R1, [R2+R3], por exemplo);
- Cache de Dados (D-cache) e de Instruções (I-cache) (secção 7.5) – Permitem que na maior parte dos casos a célula acedida se encontre já dentro do processador, reduzindo substancialmente os acessos à memória. A existência de caches separadas permite suportar acessos simultâneos (do ponto de vista do núcleo) à memória de busca de instruções e de acesso a dados, o que é importante para o funcionamento e desempenho do processador (secção 7.3);
- Interface de Memória (secção 7.4) – Arbitra e coordena os acessos à memória principal pelas caches quando estas não têm a informação pretendida ou é necessário actualizar a memória principal. Também suporta o endereçamento de byte e DMA (secção 6.4.2.3, na página 511);
- Portas tristate de acesso aos barramentos da memória externa, para poder colocar esses barramentos em alta impedância e suportar as operações de DMA;
- Multiplexers diversos para seleção de sinais alternativos.

## 7.2 NÚCLEO DO PROCESSADOR

### 7.2.1 CAMINHO DE DADOS

O Caminho de Dados (*datapath*), também designado Unidade de Processamento de Dados ou simplesmente Unidade de Dados, inclui os blocos onde circulam os dados sobre os quais o PEPE opera, em que os blocos mais importantes são o banco de registos e a ALU.

A Fig. 7.2 mostra em maior detalhe o caminho de dados, incluindo já muitos dos sinal internos que a unidade de controlo deverá gerar. O projecto de um circuito destes deve ter essencialmente em conta o conjunto de instruções que o processador deve implementar e a codificação destas (Tabela A.9, na página 707 e anteriores), para que se possa saber como obter os operandos a partir de cada instrução.

O RI (Registo de Instruções) serve para armazenar a instrução a ser executada ( vindia da memória de instruções), quando o sinal ESCR\_RI for activado. Note-se a divisão em 4 campos de 4 bits, reflectindo o formato das instruções, tal como definido na Tabela A.9. Todos os sinais (indicados a negrito e que entram nos vários blocos) são produzidos pela unidade de controlo, com base nos dois campos de 4 bits de maior peso do registo RI, que contém o código de operação da instrução que está a ser executada.

Muitas instruções envolvem um ou dois registos, que constituem os operandos A e B da ALU e são especificados através dos seus índices (0 a 15) no banco de registos. O registo do operando B está sempre nos bits RI(3..0), mas o operando A pode ser obtido a partir de vários campos de 4 bits do RI, pelo que há necessidade de utilizar um *multiplexer* para seleccionar o campo correcto, através do sinal SEL\_A. Há ainda hipótese de ser a própria unidade de controlo a indicar que registo quer usar como operando A, independentemente do que a instrução específica, caso em que será seleccionado o sinal REG\_A (de 4 bits). O mesmo se passa em relação ao registo onde deve ser armazenado o resultado da instrução, que no banco de registos se designa operando C e é seleccionado por um *multiplexer* através do sinal SEL\_C. Tipicamente é o mesmo registo que o especificado pelo operando A, mas para manter a flexibilidade permite-se escolher qualquer dos três campos de 4 bits de menor peso do RI e ainda um registo especificado pela unidade de controlo, através do sinal REG\_C (de 4 bits). Este operando C é de escrita e como tal precisa de um sinal para indicar quando escrever (ESCR\_C).

Para além dos 16 registos principais (Fig. 4.5, na página 204), o PEPE suporta 16 registos auxiliares (secção A.2.2, na página 697), para configurar os vários recursos internos. Os sinais PA\_A e PA\_C indicam se as entradas do banco de registos IND\_A e IND\_C, respectivamente, se referem a um registo principal ou auxiliar (32 registos no total).

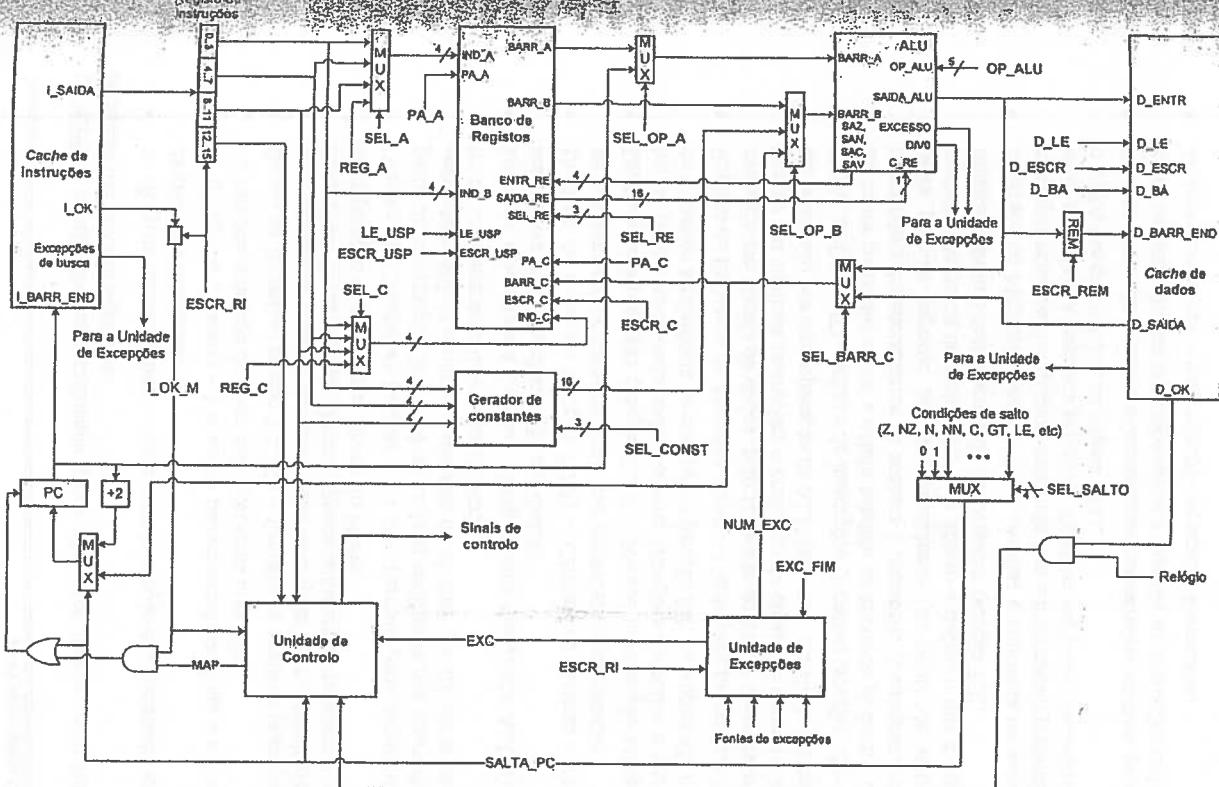


Fig. 7.2 - Diagrama geral do núcleo do PEPE, com ênfase no caminho de dados, e ligações às caches

Por limitações do número de bits de cada instrução, a instrução `MOV` é a única que pode usar os registos auxiliares (não sendo possível efectuar operações da ALU directamente sobre eles), pelo que a entrada `IND_B` do banco de registos se refere sempre a um registo principal e não é necessário um sinal `PA_B`.

A ALU recebe os dois operandos, A e B. Dependendo das instruções, o primeiro pode ser um dos registos do banco de registos ou o próprio PC (no caso dos saltos relativos, em que o PC tem de ser somado com uma constante). O segundo pode ser um registo, uma constante ou um número de uma exceção (para somar ao registo `BTE`), vindo da unidade de exceções (seção 7.2.2). Para efectuar estas selecções há dois multiplexers, controlados pelos sinais `SEL_OP_A` e `SEL_OP_B`.

Devido às limitações de espaço nas instruções, as constantes ocupam apenas 4, 8 ou 12 bits, dependendo da instrução. Em cada caso, estas constantes têm de ser estendidas (com zeros) para 16 bits. Por outro lado, a própria unidade de controlo tem necessidade de, em alguns casos, ser ela a especificar uma constante. Tudo isto motiva a introdução de uma unidade geradora de constantes, cujo funcionamento é controlado pelo sinal `SEL_CONST` (de 3 bits, pois há oito hipóteses diferentes de gerar constantes – Tabela 7.2).

A ALU está projectada para saber executar várias operações e precisa de 5 bits para seleccionar um delas (sinal `OP_ALU`). Para além do resultado, a ALU recebe o valor de transporte (*carry, bit C* do RE) e produz também informação para actualizar os bits de estado `Z, N, C` e `V` (com os sinais com o nome SA – Saída da ALU – seguido do bit respectivo). É a unidade de controlo que indica quais destes bits são afectados (alterados como resultado de cada instrução, através do sinal `SEL_RE` (3 bits, pois há oito situações diferentes)). A ALU detecta ainda duas exceções: excesso e divisão por zero.

O valor a escrever no registo de resultado ("operando" C do banco de registos) pode ser um resultado de uma operação da ALU ou de uma leitura da memória (*cache de dados*, activando o sinal `DLE`). A selecção é feita por um *multiplexer* controlado pelo sinal `SEL_BARR_C`. O resultado da operação da ALU pode também ser escrito na memória (*memória cache de dados*, activando o sinal `D_ESCR`) ou ainda no registo REM (Registo de Endereço de Memória), nas instruções em que o endereço tem de ser calculado através de uma soma de dois registos (`MOV [R1 + R2], R3`, por exemplo), caso em que primeiro é calculado o endereço e guardado no REM e só depois é feito o acesso. O sinal `D_BA` indica se o acesso à memória/*cache* de dados é feito em *byte* ou em palavra.

O PC é o registo usado para endereçar a memória de instruções, sendo incrementado de unidades (devido ao endereçamento de *byte*, sendo o processador de 16 bits) para endereçar a próxima instrução. Esta operação é tão frequente (em todas as instruções) que o PC dispõe de um somador com 2 só para si.<sup>102</sup> Em certas instruções (saltos, chamadas de

rotinas e retornos), o PC deve ser actualizado com um valor que não é igual ao anterior mais 2 mas sim determinado a partir de um resultado da ALU ou de uma leitura à memória (dependendo do sinal `SEL_BARR_C`). A escolha entre um cenário ou outro é determinada pelo sinal `SEL_SALTO`, que decide não apenas de onde vem o novo valor mas também em que condições o PC deve ser actualizado.

Nos casos em que o PC deve ser carregado com um determinado valor (escoolido por `SEL_BARR_C`), mesmo que dependente de condições (saltos condicionais), a decisão sobre se o PC deve ser actualizado ou não é tomada pelo sinal `SALTA_PC` (saída do *multiplexer* controlado pelo sinal `SEL_SALTO`), que depende das condições das instruções de salto (com base nos bits de estado), que podem ser vistas na página 706:

- Se a entrada deste *multiplexer* com 0 for seleccionada, a sua saída `SALTA_PC` a 0 indica que o PC deve ser incrementado de 2 unidades, mas tal só acontece quando o sinal `MAP` é activado, após buscar uma nova instrução da *cache* e este acesso tiver tido sucesso (`I_OK_M=1`);
- Se seleccionarmos a entrada com 1, `SALTA_PC` fica activo (a 1) e o PC memoriza o novo valor presente na sua entrada. Esta opção é usada nas instruções de salto incondicional (`JMP`), nas instruções de chamada e retorno de rotina (`CALL, RET`, etc.);
- A selecção de qualquer outra entrada, correspondente a um dos casos de salto condicional, faz o PC memorizar um novo valor (caso em que o salto é efectuado) ou manter o seu valor, dependendo da condição seleccionada. Esta opção é usada nas instruções de salto condicional (`JZ`, por exemplo).

As *caches* (quer de dados quer de instruções) desempenham um papel fundamental no processador, mantendo as células de memória mais usadas para tornar os acessos mais rápidos. No entanto, poderá suceder que uma *cache* não tenha a célula pretendida carregada e esta tenha de ser acedida na memória principal, caso em que demora mais tempo. As *caches* recuperam automaticamente (seção 7.5.1), indo buscar o valor pretendido, mas durante esse tempo o núcleo do processador tem de esperar. A solução adoptada é diferente para as duas *caches*:

- Se um acesso à *cache* de dados falhar, a única solução é parar a unidade de controlo, pois não se pode prosseguir enquanto o dado não estiver disponível. Por essa razão, o sinal `D_OK`, que fica inactivo sempre que a *cache* de dados não tem o dado pretendido, corta o sinal de relógio à unidade de controlo;
- Do lado das instruções, quando uma instrução começa a executar já o PC foi actualizado para endereçar a instrução seguinte, o que significa que durante a execução de uma instrução já a *cache* de instruções poderá estar à procura da instrução seguinte na memória principal, se a não tiver carregada. O núcleo pode continuar a execução da instrução armazenada no RI (o que pode demorar uma série de ciclos de relógio) e só precisa mesmo de parar se quiser carregar a instrução seguinte no RI e esta ainda não estiver disponível. Se for o caso, o sinal `I_OK` estará inactivo e será memorizado num registo de 1 bit (`I_OK_M`) quando `ESCR_RI`

<sup>102</sup> Poderia usar-se a ALU para este efeito, seleccionando 2 no gerador de constantes, mas desenvolvimentos subsequentes, na secção 7.3.3, obrigam a um somador específico, pelo que aquela solução não é usada.

for activado, impedindo o PC de ser incrementado. Ao mesmo tempo,  $\text{I\_OK\_M}$  indica a unidade de controlo que deve esperar (mas só quando esta quiser executar a instrução que a cache ainda está a obter).

A unidade de controlo (secção 7.2.3) gera todos os sinais de controlo necessários ao funcionamento do sistema, indicados na Fig. 7.2 por etiquetas a negrito.

A unidade de excepções (secção 7.2.2) controla as várias excepções geradas pelo hardware interno e pelos pinos de interrupção, gerando o sinal EXC quando há uma excepção pendente, cujo número (um dos indicados na Tabela A.8, na página 702) é indicado pelo sinal NUM\_EXC, que na ALU é multiplicado por 2 e somado ao registo BTR (Base da Tabela de Excepções) para aceder à memória e obter o endereço da rotina de atendimento da excepção. Caso a excepção seja uma interrupção, esta unidade espera que o sinal ESCR\_RI esteja activo para a assinalar em EXC, pois só nesta altura, entre duas instruções, é que as interrupções podem ser atendidas.

Depois de iniciar o atendimento da excepção (incluindo guardar o RE e o PC na pilha) e antes de invocar a rotina de atendimento da excepção, a unidade de controlo activa o sinal EXC\_FIM para informar a unidade de excepções que a excepção já está a ser atendida e que caso haja mais excepções pendentes em condições de ser atendidas já pode assinalar a próxima. As excepções podem provir dos circuitos associados às caches (excepções nos acessos à memória), da ALU e dos pinos de interrupção.

### 7.2.12. BANCO DE REGISTOS

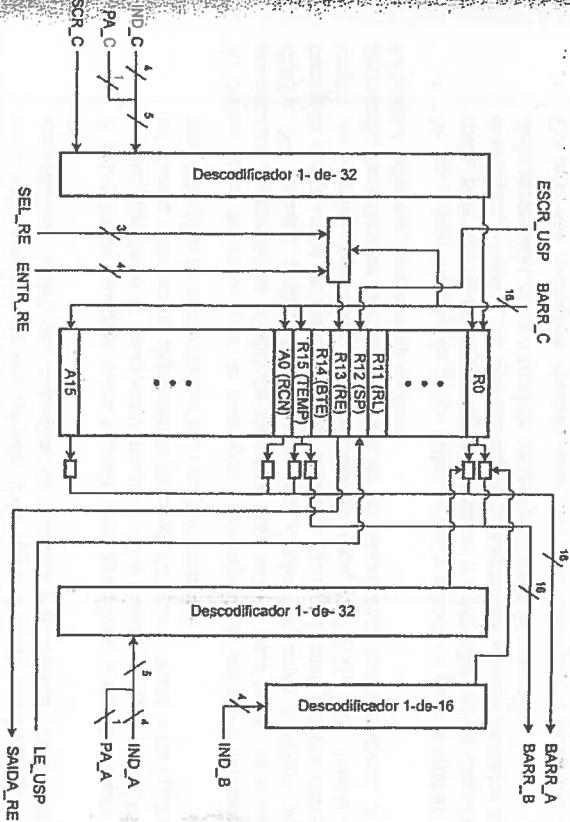
A Fig. 7.3 detalha o banco de registos da Fig. 7.2, podendo ver-se os 16 registos principais e os 16 auxiliares (secção A.2.2, na página 697). Os descodificadores 1-de-32 permitem seleccionar um só registo no conjunto, conjugando os sinais IND\_C e PA\_C por um lado e IND\_A e PA\_A por outro. Os registos auxiliares não podem circular pelo barramento BARR\_B, o que faz com que o descodificador correspondente seja apenas 1-de-16.

Existe ainda uma série de detalhes não representados, pois trata-se já de um nível muito pormenorizado, sem grande valor acrescentado em termos de compreensão do circuito e essencial apenas em termos de eventual implementação em hardware. Noteadamente, cada registo auxiliar tem uma saída directa para ligação aos recursos cuja configuração controla (ligação omitida por simplicidade), independentemente da ligação ao barramento BARR\_A.

As pequenas caixas não identificadas junto às saídas dos registos são portas *tristate* que só são activadas pelos descodificadores quando o registo respectivo é seleccionado para leitura. Os sinais de controlo destas portas estão apenas representados para o registo RO, por simplicidade, mas naturalmente o cenário repete-se para cada porta *tristate*. Cada registo principal tem duas portas *tristate* para permitir lê-lo através do barramento BARR\_A ou BARR\_B.

O registo de estado (RE) tem ainda uma entrada directa (ENTR\_RE) para os bits de estado ( $Z$ ,  $N$ ,  $C$  e  $V$ ) que vêm da ALU e uma saída (SAIDA\_RE) directa para que os seus bits

possam ligar à ALU, à Unidade de Controlo e à Unidade de Excepções. Os bits SEL\_RE indicam quais os bits de RE que podem ser alterados através da entrada directa (porque nem todas as instruções afectam todos os bits de estado), de acordo com a Tabela 7.1. O pequeno módulo à entrada do RE coordena a escrita neste registo.



| SEL.RE | BITS DE ESTADO | COMENTÁRIOS                                                                  |
|--------|----------------|------------------------------------------------------------------------------|
| 000    | Nenhum         | RE não alterado                                                              |
| 001    | Z, N, C, V     |                                                                              |
| 010    | Z, N           |                                                                              |
| 011    | Z              |                                                                              |
| 100    | Z, N, C        | Resultado de uma operação da ALU (estes 4 bits fazem parte do sinal ENTR.RE) |
| 101    | C              |                                                                              |
| 110    | NP, IE, DE     | Coloca os bits NP, IE e DE a 0 (sem afectar os restantes)                    |
| 111    | RE_0           | Coloca todos os bits do RE a 0                                               |

Tabela 7.1 - Codificação dos bits de seleção SEL.RE, que indicam quais os bits de estado que podem ser alterados directamente (sem ser pelas operações normais de escrita num registo qualquer)

### 7.2.1.3 GERADOR DE CONSTANTES

As constantes normalmente necessárias na ALU ou são obtidas a partir da constante especificada pela própria instrução, estendendo-a para 16 bits, ou geradas localmente, de acordo com o indicado pela unidade de controlo.

A codificação das instruções foi cuidadosamente planeada para que as instruções que especificam constantes o façam sempre nos bits menos significativos. Há instruções que contêm constantes de 4, 8 e 12 bits. O ideal seria as constantes serem sempre de 16 bits, mas para conseguir que todas as instruções tenham 16 bits, incluindo código de operação e operandos, sem recorrer a uma palavra adicional com o valor da constante, adoptaram-se valores máximos de constantes com os números de bits referidos, que conseguem cobrir os casos mais frequentes.

As instruções MOVL e MOVH podem ser usadas sempre que se precise de uma constante em que todos os 16 bits sejam especificados, tal como indicado na secção 4.10.3, na página 219.

A Tabela 7.2 indica as várias situações possíveis para gerar os 16 bits da constante a partir dos 4, 8 e 12 bits de menor peso da instrução, que são aqueles que algumas instruções usam para especificar as constantes. Tal é feito em grupos de 4 bits (nibbles). Cabe depois à unidade de controlo seleccionar o caso que, pretende, através do sinal SEL\_CONST (Fig. 7.2).

Convém não esquecer que quando a constante é usada para endereçamento (instruções de acesso à memória e de fluxo de controlo) deve ser multiplicada por 2, o que é conseguido em conjunção com a operação ADD\_Ex2 da ALU (Tabela 7.3). As instruções de acesso à memória com operando imediato usam sempre endereçamento por palavra e essas constantes têm sempre de ser par para o acesso ser alinhado. A constante guardada na

instrução é sempre metade do valor real, para melhor aproveitar a gama de valores permitida pelos bits disponíveis, mas ao ser usada na execução da instrução tem de ser multiplicada por 2 (ver página 214).

| SEL.CONST | OPERAÇÃO | EXTENSÃO | EXEMPLO DE USO:     | BITS DA CONSTANTE GERADA |
|-----------|----------|----------|---------------------|--------------------------|
| 0         | E4_16    | 4 → 16   | 000H SHR Rd, n      | 15..12 11..8 7..4 3..0   |
| 1         | E4_16S   |          | 0000 ADD Rd, k      | RJ(3..0)                 |
| 2         | E8_16    | 8 → 16   | 00H Sinal SWE       | RJ(3..0)                 |
| 3         | E8_16S   |          | 0000 0000 RJ(3..0)  | RJ(7..4)                 |
| 4         | E12_16S  | 12 → 16  | Sinal JMP etiqueta  | RJ(7..4) RJ(3..0)        |
| 5         | UM       | 1        | 0000 0000 RJ(11..8) | RJ(11..8)                |
| 6         | DOIS     | 2        | 0000 0000 RJ(7..4)  | RJ(7..4)                 |
| 7         | QUATRO   | 4        | 0000 0000 RJ(3..0)  | RJ(3..0)                 |
|           |          |          | PUSH Rd             | 0010                     |
|           |          |          | RFE                 | 0100                     |

Tabela 7.2 - Geração de constantes de 16 bits

A Tabela 7.2 fornece exemplos do uso das constantes na implementação das instruções do PEPE. A constante UM não é usada na implementação das instruções do PEPE, mas está prevista para a implementação de novas instruções, usando microprogramação (secção 7.2.4).

A implementação da Tabela 7.2 consiste num circuito combinatório que pode ser feito por uma tabela, com SEL\_CONST e os 12 bits de menor peso de RI (Registo de Instruções) como entradas e simplificação dos sinais de saída (com mapas de Karnaugh), ou um circuito misto com portas lógicas e multiplexers que para cada nibble da constante seleccionam o valor indicado na tabela. Esta implementação é um projecto mais de sistemas digitais do que de arquitectura de computadores e não é detalhada neste livro.

### 7.2.1.4 UNIDADE ARITMÉTICA E LÓGICA (ALU)

A ALU não é mais do que um circuito combinatório com capacidade de efectuar várias operações sobre os seus operandos BARR\_A e BARR\_B (Fig. 7.2), em que a operação realizada é seleccionada pelo sinal OP\_ALU. As operações a suportar derivam do conjunto de instruções (Tabela A.9, na página 707) e estão descritas na Tabela 7.3, que inclui algumas operações extra (não necessárias para implementar as instruções do PEPE mas que poderão ser úteis na criação de novas instruções).

Na Tabela 7.3, “P” significa negação, “X” um valor que não interessa especificar (tanto faz), “>>” e “<<” deslocamento e “>>>” e “<<<” rotação. “C” é na realidade C.RE. Esta tabela representa ainda alguns dos sinais internos da ALU, cujo diagrama detalhado está representado na Fig. 7.4. Estes sinais internos são gerados por um circuito combinatório (omitido por simplicidade) com base no código de operação da ALU (OP\_ALU).

| OP_ALU<br>(4,0) | OPERAÇÃO | SAÍDA_ALU | X                         | Y | CARRY            | MUX_C | Br2 | NA | NB | MASC_B | PERMITE |
|-----------------|----------|-----------|---------------------------|---|------------------|-------|-----|----|----|--------|---------|
| 000             | ADD      | a, b      | a + b                     | a | b                | 0     | 10  | 1  | 0  | 0      | 0       |
| 001             | ADDC     | a, b      | a + b + C                 | a | b                | C     | 00  | 1  | 0  | 0      | X       |
| 010             | SUB      | a, b      | a - b                     | a | /b               | 1     | 11  | 1  | 0  | 1      | 0       |
| 011             | SUBB     | a, b      | a - b - C                 | a | /b               | /C    | 01  | 1  | 0  | 1      | X       |
| 00              | NEG      | a         | /a+1                      | a | 0                | 1     | 11  | 1  | 1  | 0      | 1       |
| 101             | MUL      | a, b      | a * b                     | a | b                | 0     | 10  | 1  | 0  | 0      | 0       |
| 110             | DIV      | a, b      | Quociente (a / b)         | a | b                | X     | XX  | 1  | 0  | 0      | X       |
| 111             | MOD      | a, b      | Resto (a / b)             | a | b                | X     | XX  | 1  | 0  | 0      | X       |
| 000             | OP_A     | a         | a                         | a | X                | X     | XX  | x  | 0  | X      | X       |
| 001             | NOT_A    | a         | /a                        | a | X                | X     | XX  | x  | 1  | X      | X       |
| 010             | OP_B     | b         | b                         | a | b                | X     | XX  | 1  | X  | 0      | X       |
| 011             | NOT_B    | b         | /b                        | a | b                | X     | XX  | 1  | X  | 1      | 0       |
| 01              | LBTOLA   | a, b      | b(7..0)    a(7..0)        | a | b                | X     | XX  | 1  | 0  | 0      | X       |
| 101             | LBTOLA   | a, b      | a(15..8)    b(7..0)       | a | b                | X     | XX  | 1  | 0  | 0      | X       |
| 110             | HBTOLA   | a, b      | b(15..8)    a(7..0)       | a | b                | X     | XX  | 1  | 0  | 0      | X       |
| 111             | HBTOLA   | a, b      | a(15..8)    b(15..8)      | a | b                | X     | XX  | 1  | 0  | 0      | X       |
| 000             | AND      | a, b      | a $\wedge$ b              | a | b                | X     | XX  | 1  | 0  | 0      | X       |
| 001             | OR       | a, b      | a $\vee$ b                | a | b                | X     | XX  | 1  | 0  | 0      | X       |
| 010             | ADD_8x2  | a, b      | a + 2 <sup>b</sup>        | a | b                | X     | XX  | 0  | 0  | 0      | X       |
| 10              | XOR      | a, b      | a $\oplus$ b              | a | b                | 0     | 10  | 1  | 0  | 0      | X       |
| 100             | BIT      | a, n      | a $\wedge$ 2 <sup>n</sup> | a | 2 <sup>n</sup>   | X     | XX  | 1  | 0  | 0      | 1       |
| 101             | SETBIT   | a, n      | a $\vee$ 2 <sup>n</sup>   | a | 2 <sup>n</sup>   | X     | XX  | 1  | 0  | 0      | 1       |
| 110             | CLRBIT   | a, n      | a $\wedge$ 2 <sup>n</sup> | a | 1/2 <sup>n</sup> | X     | XX  | 1  | 0  | 1      | 1       |
| 111             | CPLBIT   | a, n      | a $\oplus$ 2 <sup>n</sup> | a | 2 <sup>n</sup>   | X     | XX  | 1  | 0  | 0      | 1       |
| 000             | SHR      | a, n      | a >> n+1                  | a | b                | X     | XX  | x  | X  | X      | X       |
| 001             | SHL      | a, n      | a << n+1                  | a | b                | X     | XX  | x  | X  | X      | X       |
| 010             | SRA      | a, n      | a >> n+1                  | a | b                | X     | XX  | x  | X  | X      | X       |
| 011             | ---      | X, X      | X                         | X | X                | X     | XX  | x  | X  | X      | X       |
| 11              | ROR      | a, n      | a >>> n+1                 | a | b                | X     | XX  | x  | X  | X      | X       |
| 101             | ROL      | a, n      | a << n+1                  | a | b                | X     | XX  | x  | X  | X      | X       |
| 110             | RORC     | a, n      | C    a >>> n+1            | a | b                | X     | XX  | x  | X  | X      | X       |
| 111             | ROLC     | a, n      | C    a << n+1             | a | b                | X     | XX  | x  | X  | X      | X       |

Tabela 7.3 - Definição da codificação das operações da ALU e de alguns sinais internos.

"X" significa "tanto faz".

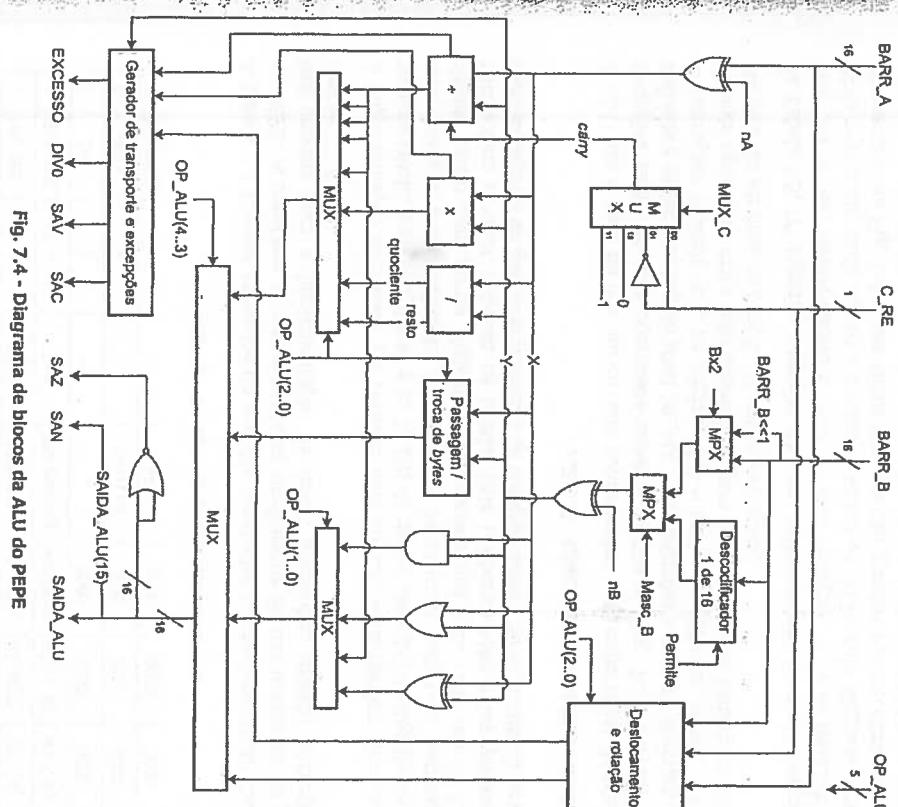


Fig. 7.4 - Diagrama de blocos da ALU do PEPE

Esta ALU é mais complexa do que a do PEP-E-8 (Fig. 3.11, na página 127), mas pode verificar-se que o princípio é o mesmo. Há várias unidades que calculam um valor com base nos operandos, mas só um deles é escolhido para a saída por meio de multiplexers. A negação (bit a bit) dos operandos também é conseguida com OU-exclusivos.

A subtração é implementada com o somador, complementando o segundo operando. Na prática, o segundo operando é simplesmente negado, usando um OU-exclusivo, e um 1 é adorado na soma colocando a entrada de transporte (carry) do somador a 1. A operação NEG é conseguida negando o operando A (na=1) e forçando a entrada de transporte (carry) do somador a 1 (MUX\_C=1), ao mesmo tempo que se força o operando

Y a zero, o que é conseguido pelo descodificador 1-de-16 com `Permite=0`, o que faz com que todas as suas 16 saídas estejam a 0.

A implementação das instruções `MUL` (multiplicação), `DIV` (divisão) e `MOD` (resto) é feita com unidades próprias, cujos detalhes estão fora do âmbito deste livro. As secções 2.8.4 (na página 103) e 2.8.5 (na página 105) lançam algumas pistas sobre o assunto, de forma simplificada, usando unidades com somas e subtrações sucessivas.

**NOTA** Os microprocessadores comerciais que precisam de ter um alto desempenho fazem estas operações com unidades complexas que conseguem produzir o resultado de forma bem mais rápida. A rapidez paga-se com complexidade.

As operações de manipulação de bits são simples, bastando providenciar os circuitos respectivos. A unidade mais complexa é a que suporta o deslocamento/rotação de  $N$  bits num só ciclo de relógio, o que é conseguido por um conjunto de multiplexers que seleccionam para a saída os bits pretendidos. Este conjunto designa-se *barrel-shifter* e permite implementar os deslocamentos lineares e rotações descritos na secção 4.13, na página 261.

O descodificador 1-de-16 permite gerar um número binário apenas com um bit a 1 (o operando B especifica qual), de forma a poder manipular apenas um dos bits do operando A (usado nas operações que manipulam um só bit, como `SETBIT` ou `CLRBIT`).

A duplicação do valor do operando B é apenas efectuada numa operação de soma e destina-se a implementar acessos à memória somando um registo a uma constante cujo valor tem de ser multiplicado por 2 (ver página 214 e secção 7.2.2). Requer um simples *multiplexer*, que selecciona o operando B mas um bit deslocado para a esquerda (em que o bit 15 não é ligado e no bit de menor peso se liga 0). É apenas uma questão de ligação dos bits, sob controlo do sinal interno `Bx2`.

As operações aritméticas com a ALU podem originar duas exceções, caso os respectivos bits de controlo no RE (Fig. 6.34, na página 481) estejam activos:

- Excesso, quando o resultado sai fora da gama representável em complemento para 2 nos 16 bits do processador;
- Divisão por zero, quando o segundo operando de DIV ou MOD é zero.

Para este efeito, a ALU produz duas saídas, EXCESSO e DIV0, que ligam à unidade de exceções (Fig. 7.1 e Fig. 7.2). A geração destas saídas, bem como dos sinais `SAVE` e `SAC`, que se destinam a actualizar os bits de estado `V` e `C`, é um circuito combinatório com algumas portas lógicas, pelo que está apresentada de forma simplificada.

Os diversos *multiplexers* representados permitem seleccionar para a saída o valor correcto. É de realçar que, apesar da complexidade aparente da Fig. 7.4, a ALU é um circuito combinatório, sem estado interno, e que muda as suas saídas mal as suas entradas mudem (aparte tempos de atraso). Só os registos conseguem memorizar estado.

#### ESSENCIAL

- O Caminho de Dados (*datapath*), ou Unidade de Processamento de Dados, é a parte do processador em que os dados são processados, sendo constituído essencialmente pelo banco de registos pelo gerador de Constantes e pela ALU e pela ligação à memória (ou "tacfe" de dados).
- O banco de registos permite ler dois registos de uma só vez (tem quais saídas) e escrever o resultado de uma operação sobre esses dois registos num terceiro.
- O gerador de constantes permite gerar algumas constantes específicas de 6 bits ou estender (com ou sem sinal) constantes com menos bits que vão incluir das instruções:

  - A ALU (Unidade Aritmética e Lógica) permite efectuar várias operações sobre um ou dois operandos. É um circuito combinatório.

#### 7.2.2 UNIDADE DE EXCEPÇÕES

A unidade de exceções recebe indicação de vários pontos do microprocessador caso haja alguma situação excepcional a assinalar à unidade de controlo, como por exemplo uma interrupção ou uma divisão por zero. Tem a capacidade de memorizar o pedido de cada exceção e de ordenar por prioridade os vários pedidos, quando ocorrem vários em simultâneo. Esta unidade trata apenas as exceções geradas em hardware, activando o sinal EXC quando há um pedido de exceção que deva ser atendido.

No caso das exceções de execução de uma instrução (divisão por 0, acesso à memória em 16 bits com endereço ímpar, etc.), o sinal EXC é activado imediatamente e apenas durante um ciclo de relógio, o que é aproveitado pela unidade de controlo (secção 7.2.3) para abortar a execução da instrução e iniciar o atendimento da exceção. Para as interrupções, que não devem interromper uma instrução a meio mas esperar por uma altura em que o processador as possa atender, a unidade de exceções possui uma entrada, INT\_OK (que fica activa durante um ciclo de relógio sempre que o processador vai executar uma nova instrução) lhe indique quando pode activar o sinal EXC, o que fará também apenas durante um só ciclo de relógio. O sinal INT\_OK liga ao ESCR\_R1, pois é nesta altura, quando o processador está a colocar no R1 a próxima instrução, que as interrupções podem ser atendidas.

Sempre que activar o sinal EXC, a unidade de exceções assume que a unidade de controlo inicia imediatamente as operações tendentes a invocar a rotina de atendimento. O sinal EXC\_FTM, gerado pela unidade de controlo, indica à unidade de exceções que a última exceção que assinalou (e cujo número estava a indicar) já foi tratada. Se houver mais exceções pendentes, a unidade de exceções deve passar a assinalar a próxima, mais prioritária (se for uma interrupção, tem de esperar pelo sinal INT\_OK). Para saber que

excepções pode pedir, esta unidade liga também aos bits do RE que controlam a permissão/inibição de excepções.

O número da excepção (sinal NUM\_EXC) gerado pela unidade de excepções tem de ser multiplicado por 2, por causa do endereçamento de byte, e somado com o registo BTE (Base da Tabela de Excepções) para aceder a esta tabela na memória e obter o endereço da rotina a invocar. Tal é feito na ALU usando a operação ADD\_Bx2 (Fig. 7.6).

A Fig. 7.5 descreve o circuito simplificado desta unidade (note-se a ligação a alguns dos bits dos registos RE e RCN – Tabela 6.13, na página 464), em que por simplicidade apenas estão representadas algumas excepções. As restantes são semelhantes de acordo com a Tabela 7.4. A Tabela A.8, na página 702, descreve as excepções predefinidas do PEPE.

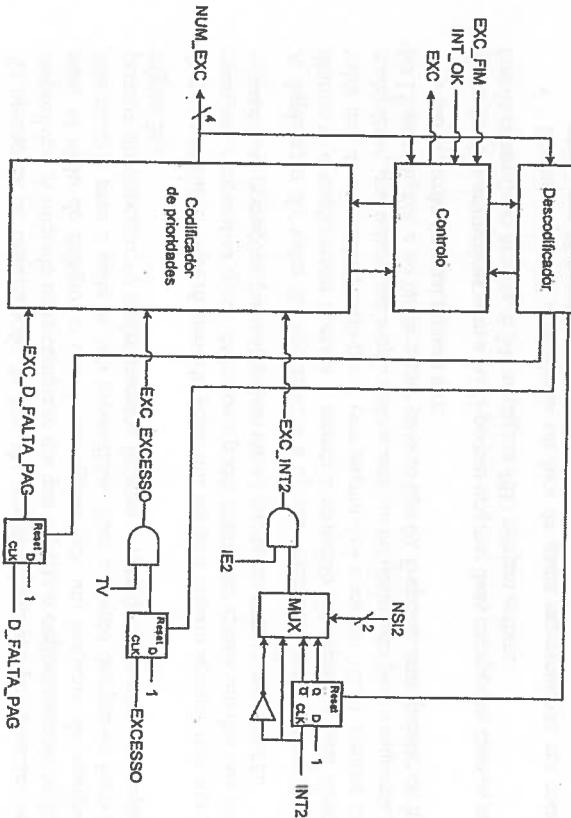


Fig. 7.5 - Diagrama de blocos simplificado da unidade de excepções. Apresentadas algumas excepções, a título exemplificativo

### 7.2.3 UNIDADE DE controlo

O papel da unidade de controlo é gerar todos os sinais que controlam os recursos do processador (Fig. 7.2), de modo a que todo o conjunto funcione correctamente.

No PEPE-8, uma simples ROM de descodificação é suficiente para gerar todos os sinais a partir do opcode da instrução (Fig. 3.18, na página 150), uma vez que se trata de uma arquitetura em que as instruções são executadas num só ciclo de relógio. No PEPE, de 16 bits, as instruções são mais complexas (chamadas a rotinas, por exemplo) e podem

demorar vários ciclos de relógio a executar, havendo ainda casos especiais para tratar, como as interrupções.

| EXCEPÇÃO EXEMPLO | COMENTÁRIOS                                                                                             | OUTRAS EXCEPÇÕES CONCILIADO COM CIRCUITO DE ENTRADA SEMELHANTE                              |
|------------------|---------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| INT2             | Mascarável pelo bit IE2 do RE. Os bits NS12 do RCN configuram a sensibilidade (flanco ou nível, 0 ou 1) | INT0<br>INT1<br>INT3                                                                        |
| EXCESSO          | Mascarável pelo bit TV do RE                                                                            | DIV0<br><br>D DESALINHADO<br>L DESALINHADO<br>L_FALTA_PAG<br>D PROT<br>L PROT<br>SO LEITURA |
| D_FALTA_PAG      | Não mascarável                                                                                          |                                                                                             |

Tab. 7.4 - Características dos circuitos de entrada na unidade de excepções

solução é usar uma unidade de controlo que implemente uma máquina de estados microprogramada (Fig. 2.44b, na página 86), que seja capaz de sequenciar todas as operações elementares (microinstruções) necessárias, tal como representado na Fig. 7.6.

Cada instrução em código-máquina é implementada por uma sequência de microinstruções (cada uma executada num ciclo de relógio), armazenadas numa ROM. O conjunto de todas as palavras desta ROM designa-se microcódigo. A ROM de microcódigo contém uma microinstrução, o que se reflecte numa combinação de sinais que implementam uma operação elementar (ou mais do que uma, desde que possam ser executadas no mesmo ciclo de relógio). Modificando as microinstruções, é possível mudar/acrescentar as instruções disponíveis sem alterar o hardware do processador.

O registo MPC (Micro Program Counter, também muitas vezes designado CAR – Control Address Register), é responsável por indicar qual a microinstrução a ser executada e determina no microcódigo funções equivalentes ao PC no código-máquina. A unidade de controlo é um processador dentro do processador, mas com funções exclusivamente de controlo. Este registo é actualizado (memorizado um novo valor) em cada ciclo de relógio do processador.

As execuções das instruções obedecem a um ciclo, já delineado na Fig. 6.29, na página 470, com os seguintes passos fundamentais:

1. Busca instrução (Ie-a da cache de instruções para o RI, registo de instruções, usando o PC como endereço).
2. Incrementa o PC de 2 unidades, preparando-o já para a próxima instrução.
3. Descodifica a instrução presente no RI, agilizando o controlo para a sequência de microinstruções que implementa essa instrução.

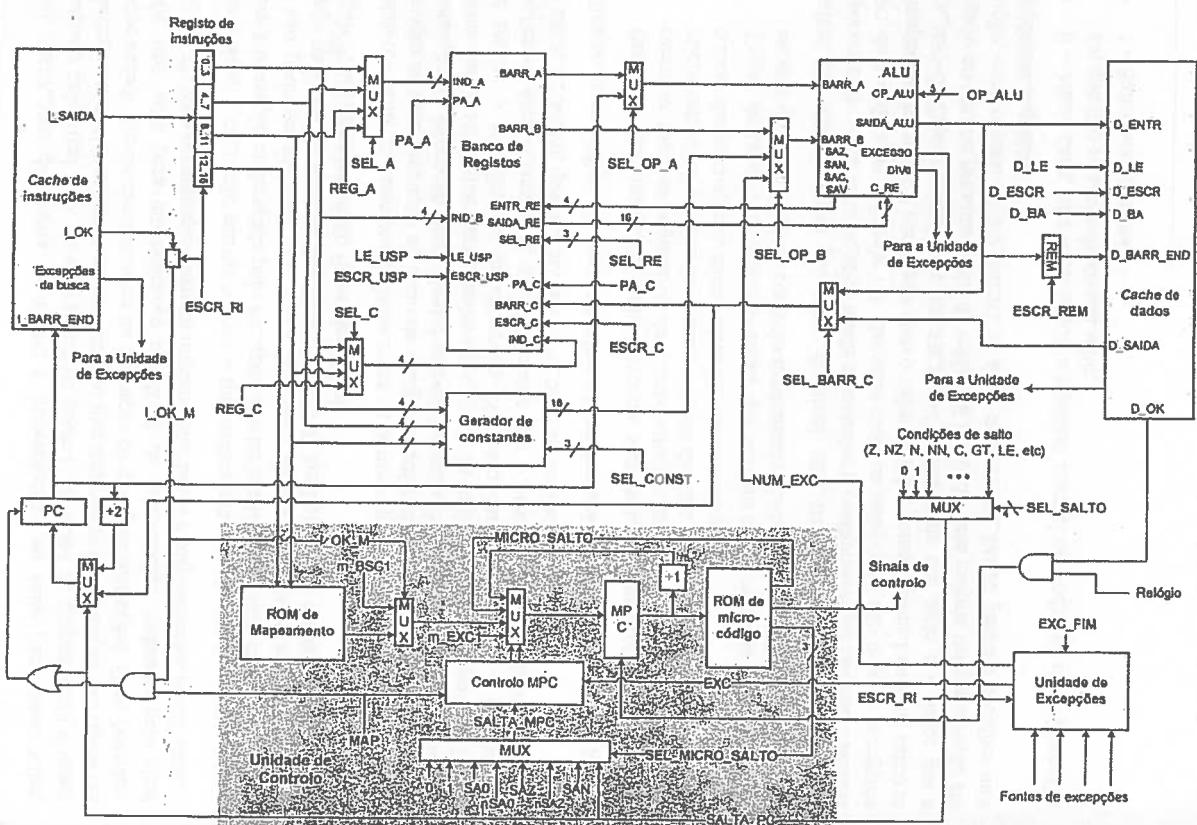


Fig. 7.5 - Inserção da Unidade de controlo no diagrama geral do núcleo do REPE.

4. Executa as várias microinstruções que implementam esta instrução. Em caso de erro, termina a instrução imediatamente.
5. Verifica se há algum pedido de exceção pendente (no caso de uma interrupção, os bits de permissão respectivos no RE terão de estar activos). Se sim, vai executar as microinstruções que permitem invocar a rotina de atendimento.
6. Volta ao passo 1.

Este ciclo é em parte implementado à custa das microinstruções e do MPC, e o resto usa algum hardware de suporte (Tabela 7.5) para ser mais eficiente.

O passo 3 é efectuado pela ROM de Mapeamento, que não é mais do que uma tabela que indica, para cada instrução (através do seu código de operação, ou *opcode*, contido nos bits de maior peso do RI), qual o endereço na ROM de microcódigo da primeira microinstrução da sequência que implementa essa instrução. A ROM de mapeamento é necessária porque o número de microinstruções que implementam várias instruções não é sempre o mesmo e alguém tem de saber em que microinstrução começar, para cada uma das instruções.

Esta ROM só é usada para, dado o *opcode* (8 bits de maior peso) de uma instrução, obter a primeira microinstrução da execução dessa instrução (activando o sinal MAP). Nas microinstruções seguintes, são elas próprias que mantêm o sequenciamento, ou simplesmente incrementando o MPC ou indicando qual o endereço (na ROM de microcódigo) da microinstrução seguinte (o que corresponde a um salto no microcódigo). A Tabela 7.9 exemplifica o conteúdo da ROM de mapeamento.

Note-se que, ao contrário do PC, o MPC é incrementado de apenas uma unidade. O MPC endereça apenas palavras da ROM de microcódigo, seja qual for a sua largura em bits, e o conceito de acesso byte a byte não existe. Todos os bits da palavra da ROM de microcódigo são necessários em cada instante, pois ligam directamente aos recursos de hardware que controlam.

Em cada ciclo de relógio, o novo valor de MPC é indicado pelo *multiplexer* à sua entrada, controlado por um pequeno bloco combinatório ("Controlo MPC") que implementa a Tabela 7.5.

| EXC | MAP | SALTA_MPC | PRÓXIMO VALOR DE MPC    | FUNÇÃO                 |
|-----|-----|-----------|-------------------------|------------------------|
| 0   | X   | 0         | MPC + 1                 | Segue                  |
| 0   | 0   | 1         | MICRO_SALTO             | Salta (no microcódigo) |
| 0   | 1   | 1         | ROM Mapeamento [opcode] | Mapa instrução         |
| 1   | X   | X         | m_EXC1                  | Trata exceção          |

Tabela 7.5 - Condições que determinam o novo valor do MPC em cada ciclo de relógio (caixa "Controlo MPC" na Fig. 7.6). "X" significa "tanto faz".

O sinal **SALTA\_MPC** determina se o **MPC** é incrementado ou salta para outro valor, determinado pelo sinal **MAP**. Se estiver activo (**MAP=1**), a ROM de mapeamento é usada para determinar qual a primeira microinstrução que implementa a instrução carregada em **RI** (o que sucede imediatamente antes da execução de qualquer instrução). Caso contrário (**MAP=0**), MPC salta para um endereço na ROM de microcódigo indicado pelo sinal **MICRO\_SALTO**. Esta organização permite implementar saltos e mapeamentos condicionais.

Mas se o sinal **I\_OK\_M** não estiver activo, o que indica que a cache ainda não conseguiu fornecer a instrução endereçada pelo PC, após este ter mudado de valor, então a instrução ainda não pode ser mapeada e o MPC é forçado a ficar com o valor **n\_BSC1**, que é o endereço na ROM de microcódigo (ver Tabela 7.7) em que é feita a busca de uma nova instrução<sup>103</sup>, até que a instrução fique disponível.

No entanto, o sinal **EXC** tem precedência sobre os outros e pode mesmo terminar (abrir) a execução de uma instrução no ciclo de relógio imediatamente seguinte à ocorrência de uma situação de exceção. Este sinal é activado apenas durante um ciclo de relógio, o suficiente para o MPC sair para o endereço **n\_EXC1** da ROM de microcódigo (ver Tabela 7.8) e iniciar o atendimento da exceção. No caso das interrupções, a unidade de excepções só gera o sinal **EXC** quando souber que o processador está entre instruções (sinal **ESCR\_RI** activo, que liga ao sinal **INT\_OK** desta unidade – Fig. 7.5).

Os saltos no microcódigo, tal como os saltos na programação em assembly, servem para:

- Partilhar microinstruções entre diversas sequências e implementar ciclos. Por exemplo, todas as sequências de microinstruções terminam com um salto para as microinstruções que implementam a busca de uma nova instrução (recomeçando o ciclo das instruções). Estes saltos são incondicionais;
  - Tomar decisões com base em sinais que actuam como entradas da máquina de estados que a unidade de controlo implementa. Estes saltos são condicionais.
- Os saltos são seleccionados por meio do sinal **SEL\_MICRO\_SALTO**, que controla um multiplexor cuja saída (**SALTA\_MPC**) ajuda a controlar o multiplexor que escolhe a entrada do MPC, de acordo com a Tabela 7.5. Em cada ciclo de relógio, se não houver excepções nem mapeamento, o MPC é carregado com o valor anterior somado com 1 (não há salto) se **SALTA\_MPC=0**. Caso contrário, se **SALTA\_MPC=1**, trata-se de um salto e o valor que é carregado no MPC no próximo ciclo de relógio é indicado pela própria microinstrução em execução, com o sinal **MICRO\_SALTO**. O sinal **SEL\_MICRO\_SALTO** permite escolher uma das hipóteses seguintes:
- 0 – Neste caso, não salta, sendo a hipótese normal em todas as microinstruções em que não está envolvido um salto;
  - 1 – Salto incondicional;

**SA0 e nSA0** – Salta se o bit 0 (o de menor peso) da saída da ALU for 0 ou 1, respectivamente. Testa se o resultado de uma operação é par ou ímpar;

**SAZ, nSAZ e SAN** – Salta se o resultado de uma operação na ALU for zero, diferente de zero, negativo ou não negativo, respectivamente. Note-se que estes sinais referem-se à saída da ALU e não aos bits do RE, permitindo testar o resultado da ALU sem ter de destruir os bits de estado quando se quer testar resultados inter-médios de uma dada instrução;

**SALTA\_PC** (saída do multiplexor controlado pelo sinal **SEL\_SALTO**) – Salta se uma das condições de salto (a seleccionada por **SEL\_SALTO**) for verdadeira. Note-se que as hipóteses 0 e 1 de **SEL\_SALTO** (PC não alterado e alteração incondicional) não podem ser aproveitadas para o multiplexor da unidade de controlo, pois **SEL\_SALTO** é usado para controlar a escrita de novos valores no PC.

## 7.2.4 MICROPROGRAMAÇÃO

A microprogramação consiste em especificar os bits contidos em cada microinstrução (saída da ROM de microcódigo) de forma a controlar os diversos recursos de hardware.

A secção 2.6.7.4, na página 83, já deu um exemplo muito simples de um circuito microprogramado, apresentando o conteúdo da ROM de microcódigo directamente em bits. Os é 1s. Na Fig. 7.6 é nítido que o controlo de um processador é bem mais complexo, com muitos sinais para gerar (é alguns deles com vários bits), embora o princípio seja o mesmo. No entanto, é para lidar com a complexidade, tal como se programa um processador em linguagem assembly e não em código-máquina (sendo a conversão uma tarefa do assembler) também se usa um microassembler para gerar o microcódigo a partir de uma linguagem simbólica (muitas vezes designada *microassembly*), em que o (micro)programador lida com constantes simbólicas em vez de 0s e 1s (senão, a microprogramação seria demasiado fastidiosa e sujeita a erros).

A especificação simbólica das microinstruções é designada microprograma, que o microassembler transforma no microcódigo memorizado na ROM da unidade de controlo. A ROM de mapeamento é também produzida pelo microassembler, tal como num nível superior o assembler produz uma tabela de símbolos (com os endereços atribuídos a cada etiqueta).

A notação simbólica de muito baixo nível já foi usada na Tabela 3.11, na página 151, que mostra os valores que os sinais devem ter para cada uma das instruções do PEPE-8 (na ROM de descodificação). No entanto, o PEPE-8 pode considerar-se um caso trivial de microprogramação, pois todas as instruções são executadas num só ciclo de relógio.

### 7.2.4.1 CIRCUITO SIMPLES MICROPROGRAMADO

Como preparação para o estudo mais detalhado do controlo do PEPE, a Fig. 7.7 apresenta um caso muito simples de sistema microprogramado em que a operação pretendida (deslocamento de um valor de  $N$  bits para a direita) se faz em vários ciclos de relógio, bit

<sup>103</sup> Na realidade, pode verificar-se na Tabela 7.7 e na Unidade de Controlo da Fig. 7.6 que o **MPC** fica a alternar entre **n\_BSC1** (busca) e **n\_BSC2** (mapeamento) até o **I\_OK\_M** ficar activo.

a bit. Este sistema só sabe fazer esta função e está muito longe das capacidades de um microprocessador, mas pode reparar-se que a estrutura da sua unidade de controlo é semelhante à do PEPE (Fig. 7.6).

O registo RA memoriza o valor  $x$  a deslocar de  $N$  bits, em que  $N$  é memorizado no registo RB. A primeira operação a efectuar é memorizar estes valores nestes registos, activando os sinais ESCR\_RA e ESCR\_RB. Em seguida, por cada ciclo de relógio deve deslocar-se o RA de um bit para a direita (activando o sinal SHR\_RA) e decrementar RB (activando o sinal DEC\_RB), repetindo-se estas duas operações até RB ser zero, o que é detectado pelo sinal nz (obtido pelo NOR de todos os bits de RB), que constitui uma entrada da unidade de controlo. A sua negação, nz, é também disponibilizada à unidade de controlo para mostrar que é possível testar as duas condições (RA ser zero e diferente de zero).

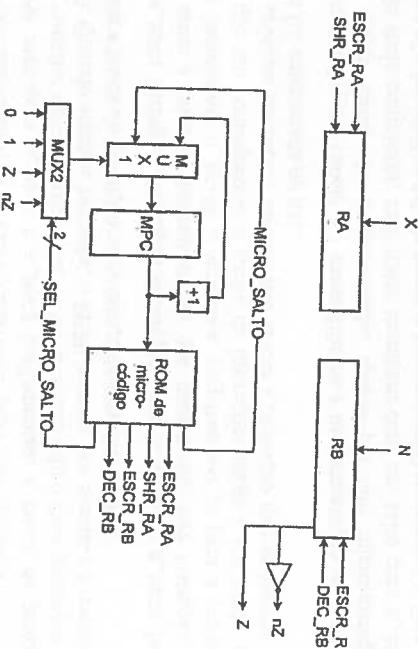


Fig. 7.7 - Circuito microprogramado para deslocar um registo de  $N$  bits

A ROM de microcódigo gera todos os sinais, não apenas os que controlam os registos (parte controlada) mas também os usados para a própria gestão da unidade de controlo:

- MICRO\_SALTO – Especifica o endereço da microinstrução para onde saltar, se for o caso;
- SEL\_MICRO\_SALTO – Especifica se é para saltar ou não e, se for, em que condições.

O registo MPC mantém o endereço da microinstrução a ser executada e o multiplexer MUX2 selecciona o seu novo valor para o próximo ciclo de relógio. Ou é igual ao anterior mas, [não há salto] ou a própria ROM de microcódigo indica um novo endereço. O sinal SEL\_MICRO\_SALTO permite seleccionar a entrada de controlo do MUX1, que será 0 (não há salto), 1 (há salto incondicional), z (salta se  $z=1$ ) e nz (salta se  $nz=1$ , ou se  $z=0$ ).

#### 7-O PROCESSADOR EM DETALHE

O número de bits necessário para MPC (e para MUX1 para o sinal MICRO\_SALTO) depende apenas do número de microinstruções da ROM de microcódigo.

A Tabela 7.6 mostra o microprograma que implementa este comportamento (assumindo, para simplificar, que só é executado uma vez) e o respectivo microcódigo, em binário.

| Endereço | Acceso                              | MICROPROGRAMA |        |         |         | ROM de microcódigo |
|----------|-------------------------------------|---------------|--------|---------|---------|--------------------|
|          |                                     | ESCR_RA       | SHR_RA | ESCR_RB | DEC_RB  |                    |
| Início   | RA $\leftarrow X$                   | SIM           | SIM    |         |         | 0 1010 0000        |
| desloca  | RA $\gg 1$                          | SIM           | SIM    | SIM     |         | 1 0101 0000        |
| testa    | (RB > 0) : MPC $\leftarrow$ desloca |               |        | JNZ     | desloca | 2 0000 1101        |
| final    | MPC $\leftarrow$ fim                |               |        | JMP     | fim     | 3 0000 0111        |

Tabela 7.6 - Microprograma (símbólico) e microcódigo (binário) para o circuito da Fig. 7.7. Os números por baixo dos sinais indicam o número de bits que cada sinal tem

Para tornar mais explícito quais os sinais relevantes em cada microinstrução, só se especificam os sinais activos, assumindo que os restantes têm um valor por omissão. Assim, os sinais de um bit (ESCR\_RA, SHR\_RA, ESCR\_RB e DEC\_RB) em branco têm o valor inactivo (não, ou 0), enquanto os activos têm o valor SIM (1). Assumiu-se também que o valor por omissão do sinal SEL\_MICRO\_SALTO é 00 (não salta, incrementa apenas o MPC). O valor de MICRO\_SALTO por omissão (quando não há salto) é irrelevante, tendo sido usado 00.

Os 8 bits da palavra da ROM de microcódigo são formados pela justaposição dos valores dos vários sinais, pela ordem com que aparecem na Tabela 7.6. É óbvio que é preferível programar de forma símbólica e deixar o microassembler produzir o conteúdo da ROM de microcódigo do que especificar manualmente os bits da ROM.

#### SIMULACRO 7.1 - CIRCUITO SIMPLES MICROPROGRAMADO

Esta simulação ilustra o funcionamento do circuito da Fig. 7.7, usando o microcódigo da Tabela 7.6, e envolve essencialmente fazer evoluir o relógio passo a passo (um ciclo de cada vez) e verificar o que se vai passando nos diversos pontos do circuito (não necessariamente, os sinais que vão ficando activos).

## 7.2.4.2 MICROPROGRAMAÇÃO NO PEPE

A unidade de controlo de um microprocessador como o PEPE tem mais sinais para controlar, mais microinstruções, mais bits no MEC e mais algumas pequenas diferenças, mas o princípio de funcionamento é o mesmo. Os sinais de um só bit têm geralmente os valores 0 e 1, mas outros com mais bits têm constantes simbólicas para representar os valores possíveis. A Tabela 7.7 e a Tabela 7.8 ilustram a micropogrammación no PEPE.

A Tabela 7.7 contém apenas algumas das microinstruções que implementam as instruções do PEPE, incluindo as que fazem busca das instruções. Os símbolos representados são os mais relevantes da Fig. 7.6. O conjunto completo está incluído no próprio simulador do PEPE (Simulação 7.2), que permite executar as microinstruções passo a passo. A Tabela 7.8 ilustra as microinstruções de tratamento das interrupções e exemplifica como definir duas novas instruções. A localização dos operandos em cada instrução máquina consta das colunas "Campos da instrução" da Tabela A.9, na página 707.

E importante saber que a inicialização do PEPE (através do pino RESET) coloca o PC, o RE e o MPC a zero. As instruções  $m_{BSC1}$  e  $m_{BSC2}$  implementam a busca da próxima instrução e são as primeiras microinstruções da ROM de microcódigo (nos endereços 0 e 1, respectivamente). Quando arranca, portanto, o PEPE começa por executar a microinstrução  $m_{BSC1}$  e fazer a busca da instrução no endereço de memória 0000H.

A microinstrução  $m_{BSC1}$  carrega a instrução endereçada por PC no RI e  $m_{BSC2}$  faz o mapeamento para achatar a primeira microinstrução que implementa essa instrução, ao mesmo tempo que incrementa o PC de 2 unidades, preparando-o já para a próxima instrução (note-se que em consequência a cache de instruções muda a sua saída, que passará a conter a próxima instrução, mas tal não afecta a instrução que se buscou em  $m_{BSC1}$ , pois esta já foi armazenada no RI).

As restantes microinstruções da Tabela 7.7 exemplificam a implementação de algumas das instruções do PEPE. Unas são implementadas apenas por uma microinstrução, enquanto outras são mais complexas, mas todas terminam com um salto para  $m_{BSC1}$  (incluindo o NOP, que não faz mais nada), de forma a buscar a próxima instrução. Este é o ciclo básico do funcionamento do processador.

**NOTA**

Início das instruções em que o PC memoriza um novo valor (caso de salto ou de chamada de rotina) o PC é alterado duas vezes. A primeira destina-se a preparar o PC para a instrução seguinte (+2), o que é feito em m\_BSCC, e a segunda para colocar no PC o endereço destino do salto ou do inicio da rotina, já na execução da instrução (escollhendo em SEL\_SALTO uma das condições que fazem activar SALTA\_PC). Esta segunda alteração desfaz o efeito da primeira, que ocorre por omissão em todas as instruções. Isto pode ser visto na Tabela 7.7, nas microinstruções da busca de instruções e na instrução JNP\_RS.

Ao contrário do que a Fig. 6.29 (página 470) sugere, o teste à existência de exceções (sinal EXC=1 na unidade de controlo) não é feito pelas microinstruções, mas sim em hardware, tal como descrito na Tabela 7.5. É uma optimização de desempenho, mas em termos de sequenciamento de operações o efeito é semelhante.

| INSTRUÇÃO       |         | NAME                                | MICRO-OPERAÇÕES |       |       |        |         |         |       |     |      |     | SEL_A |  | SEL_B |  | SEL_C |  | SEL_D |  | SEL_E |  | SEL_F |  | SEL_G |  | SEL_H |  | SEL_I |  | SEL_J |  | SEL_K |  | SEL_L |        | SEL_M   |        | SEL_N  |  | SEL_O |  | SEL_P |  | SEL_Q |  | SEL_R |  | SEL_S |  | SEL_T |  | SEL_U |  | SEL_V |  | SEL_W |  | SEL_X |  | SEL_Y |  | SEL_Z |  | SEL_BALTO |  | SEL_MORBALTO |  |
|-----------------|---------|-------------------------------------|-----------------|-------|-------|--------|---------|---------|-------|-----|------|-----|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--------|---------|--------|--------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-------|--|-----------|--|--------------|--|
| Busca           | m_BSC1  | RI ← [MPC]                          |                 |       |       |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  | SIM   |        |         |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
|                 | m_BSC2  | MPC ← MAP[apcaddr];<br>PC ← PC + 2  |                 |       |       |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  | SIM   | SALTA  |         |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| NOP             | m_NOP   | MPC ← BSC1                          |                 |       |       |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |        |         | SALTA  | m_BSC1 |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| ADD Rd, Rs      | m_ADD   | Rd ← Rd + Rs;<br>MPC ← BSC1         | RI_7_4          | REG   | REG   |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |        |         | SALTA  | m_BSC1 |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| ADD Rd, k       | m_ADDI  | Rd ← Rd + k;<br>MPC ← BSC1          | RI_7_4          | REG   | CONST | E4_16S | RI_7_4  | SIM     | ALU   | ADD | ZNCV |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       | SALTA  | m_BSC1  |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| SHR Rd, n       | m SHR   | Rd ← Rd >> k;<br>MPC ← BSC1         | RI_7_4          | REG   | CONST | E4_16  | RI_7_4  | SIM     | ALU   | SHR | ZNC  |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  | SALTA | m_BSC1 |         |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| MOVL Rd, k      | m_MOVL  | Rd ← K(7)(8)    k;<br>MPC ← BSC1    |                 |       |       | CONST  | E8_16S  | RI_11_8 | SIM   | ALU | OP_B |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  | SALTA | m_BSC1 |         |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| MOV Rd, [Rs+RJ] | m_LDR1  | REM ← R + RI                        | RI_7_4          | REG   | REG   |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |        | SIM     |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
|                 | m_LDR2  | Rd ← Mw[REM];<br>MPC ← BSC1         |                 |       |       |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |        |         | SALTA  | m_BSC1 |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| MOV [Rd+n], Rs  | m_ST01  | REM ← R + 2*(n/2)                   | RI_7_4          | REG   | CONST | E4_16S |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       | SIM    |         |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
|                 | m_ST02  | Mw[REM] ← Rs;<br>MPC ← BSC1         | RI_7_4          | REG   |       |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |        |         | SALTA  | m_BSC1 |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| JZ etiqueta     | m_JZ    | Z=1: PC ← PC + 2*dif;<br>MPC ← BSC1 |                 | PC    | CONST | E8_16S |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       | Z      | SALTA   | m_BSC1 |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| JMP Rs          | m_JMPR  | PC ← Rs;<br>MPC ← BSC1              | RI_3_0          | REG   |       |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |        |         | SALTA  | m_BSC1 |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
| CALL etiqueta   | m_CALL1 | REM ← SP - 2                        | REG_A           | SP    | REG   | CONST  | DOIS    |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       | SIM    |         |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
|                 | m_CALL2 | Mw[REM] ← PC                        |                 |       | PC    |        |         |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |        | OP_A    |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
|                 | m_CALL3 | PC ← PC + 2* dif                    |                 |       | PC    | CONST  | E12_16S |         |       |     |      |     |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       | ALU    | ADD_Bx2 |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |
|                 | m_CALL4 | SP ← SP - 2;<br>MPC ← BSC1          |                 | REG_A | SP    | REG    | CONST   | DOIS    | REG_C | SP  | SIM  | ALU | SUB   |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       | SALTA  | m_BSC1  |        |        |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |       |  |           |  |              |  |

**Tabela 7.7 - Microprogramação simbólica.** A ROM de microcódigo é gerada a partir de uma tabela como esta (que representa apenas algumas microinstruções e sinais a controlar). O simulador inclui o microcódigo completo

Na Tabela 7.8, a sequência de microinstruções que trata do início do atendimento de uma rotina de exceção (`m_EXC1` em diante) é semelhante à implementação da instrução `SWE` (*Software Exception*) descrita na Tabela 6.15, na página 480 (mas incluindo agora o detalhe da utilização do `REM`, Registo de Endereço de Memória). Difere apenas no facto de o número da exceção entrar no *multiplexer* antes da entrada `BARR_B` da ALU (operando B), enquanto que no caso da instrução `SWE` esse número está no `RT`, na própria instrução.

Note-se que, embora seja preciso aceder aos endereços de memória (na pilha) `SP-2` e `SP-4`, o `SP` só é actualizado no fim da instrução, para permitir gerar uma exceção de falta de página (decorrente do mecanismo de memória virtual – secção 7.6, na página 643), abortando a instrução que a originou, sem ter de desfazer alterações já feitas ao `SP`. Isto permite recomençar a execução dessa instrução mais tarde (quando a página já estiver disponível) sem ter de recuperar de um estado intermédio.

Aliás, o mesmo cuidado foi já posto na instrução `CALL`, na Tabela 7.7, e em todas as instruções do PEPE em que possam ser feitas alterações ao estado do processador antes de um acesso à memória (que pode falhar por falta de página).

Para além do atendimento das exceções, a Tabela 7.8 ilustra a implementação de duas novas instruções, que não existem no PEPE de base:

- `ADDM [Rd], k` – Adiciona uma constante `k` de 4 bits (-8 a +7) no endereço de memória indicado pelo registo `Rd`. A constante é estendida para 16 bits com sinal. O PEPE não consegue somar directamente com a memória, pelo que o truque é ler o valor para o registo `TEMP`, fazer a soma e armazenar o resultado nessa mesma posição de memória;
- `SUM Rc, [Rs], Rd` – Soma todos elementos (de 16 bits) de um vector em memória, a partir do endereço indicado pelo registo `Rs`. O registo `Rc` indica quantos elementos se considera que o vector tem e o registo `Rd` é usado para acumular o resultado. Os três registos são alterados pela instrução. O algoritmo usado é simples. O `Rd` é inicializado com zero e depois entra num ciclo em que se testa se o valor de `Rc` já é zero (caso em que termina). Se ainda não for, lê o valor da memória e soma-o ao `Rd`, incrementa `Rs` de duas unidades (por causa do endereço de *byte*) e decrementa `Rc` de uma unidade (menos um elemento do vetor para tratar). Esta instrução ilustra os ciclos num microprograma.

Para que cada uma destas duas novas instruções possa ser usada num programa do PEPE é preciso:

1. Definir a sequência de microinstruções que a implementa.
2. Definir qual o código de operação (*opcode*) e formato da instrução máquina (localização dos operandos nos 16 bits da instrução). Em princípio, dever-se-á usar um código de operação livre, ainda não usado (embora o *assembler* do PEPE permita reutilizar um já usado, redefinindo a instrução).

| INSTRUÇÃO                     | NOME                 | MICROOPERAÇÕES |           |           |       |        |        |         |       |       |       |
|-------------------------------|----------------------|----------------|-----------|-----------|-------|--------|--------|---------|-------|-------|-------|
|                               |                      | SEL_OP_A       | SEL_OP_B  | SEL_CONST | REG_A | REG_B  | REG_C  | REG_D   | REG_E | REG_F | REG_G |
| Atendimento da exceções       | <code>m_EXC1</code>  | RE             | REG       |           |       |        |        |         |       |       |       |
|                               | <code>m_EXC2</code>  | RE             | REG       | CONST     |       |        |        |         |       |       |       |
|                               | <code>m_EXC3</code>  | PC             | REG       | CONST     |       |        |        |         |       |       |       |
|                               | <code>m_EXC4</code>  | REG            | SP        | REG       |       |        |        |         |       |       |       |
|                               | <code>m_EXC5</code>  | REG            | TEMP      | REG       |       |        |        |         |       |       |       |
|                               | <code>m_EXC6</code>  | REG            | BTE       | REG       | EXC   |        |        |         |       |       |       |
|                               | <code>m_EXC7</code>  | PC             | Mw[REM]   | REG       |       |        |        |         |       |       |       |
|                               | <code>m_EXC8</code>  | SP             | SP - 4    | REG       | CONST | QUATRO | REG_C  | SP      | SIM   | ALU   | SUB   |
| <code>ADDM [Rd], k</code>     | <code>m_ADDM1</code> | REM            | Rd        | REG       |       |        |        |         |       |       |       |
|                               | <code>m_ADDM2</code> | TEMP           | Mw[REM]   | REG       |       |        |        |         |       |       |       |
|                               | <code>m_ADDM3</code> | TEMP           | TEMP + k  | REG       | REG   | CONST  | E1_16S | REG_C   | TEMP  | SIM   | MEM   |
|                               | <code>m_ADDM4</code> | Mw[REM]        | TEMP      | REG       |       |        |        |         |       |       |       |
| SUM <code>Rc, [Rs], Rd</code> | <code>m_SUM1</code>  | Rd             | 0         | REG       | REG   |        |        | RI_3_0  |       |       |       |
|                               | <code>m_SUM2</code>  | Rc = 0         | MPC       | REG       |       |        |        |         |       |       |       |
|                               | <code>m_SUM3</code>  | REM            | Rs        | REG       |       |        |        |         |       |       |       |
|                               | <code>m_SUM4</code>  | TEMP           | Mw[REM]   | REG       |       |        |        | REG_C   | TEMP  | SIM   | MEM   |
|                               | <code>m_SUM5</code>  | Rd             | Rd + TEMP | REG       | REG   | REG    |        | RI_3_0  | SIM   | ALU   | ADD   |
|                               | <code>m_SUM6</code>  | Rs             | Rs + 2    | REG       | CONST | DOIS   | RI_7_4 |         | SIM   | ALU   | SUB   |
|                               | <code>m_SUM7</code>  | Rc             | Rc - 1    | MPC       | REG   | CONST  | UM     | RI_11_B | SIM   | ALU   | SUB   |

Tabela 7.8 - Microinstruções que tratam do atendimento de exceções e de duas instruções que não existem no PEPE de base mas podem ser definidas, alterando a ROM do microcódigo, e usadas em programas de linguagem assembly

3. Incluir a instrução na ROM de mapeamento, para que quando a instrução for buscada pelo PEPE este saiba onde começa a sequência de microinstruções que a implementa.
4. Definir uma mnemónica e um formato da instrução em linguagem assembly, e de alguma forma fornecer essa informação ao assembler, para que este reconheça a instrução quando ela aparecer num programa e gere a respectiva instrução máquina.

O passo 1 já está feito na Tabela 7.8.

A Tabela A.9, na página 707 e anteriores, inclui a descrição de todas as instruções predefinidas no PEPE, incluindo os respectivos códigos de operação (opcodes). Os 4 bits de maior peso (15..12) das instruções são sempre de opcode (primário), mas os 4 bits seguintes (11..8) podem ser de operandos, quando estes gastam 12 bits, ou de opcode (secundário), quando os operandos gastam 8 bits ou menos. Só há 16 opcodes primários, pelo que o número de instruções com 12 bits de operandos (sem opcode secundário) tem de ser muito limitado. É o opcode secundário que permite aumentar o número de instruções.

A instrução SUM R<sub>c</sub>, [R<sub>s</sub>], R<sub>d</sub> tem três operandos, que exigem 4 bits cada (no PEPE há 16 registos), o que significa que só pode ter um opcode primário. Felizmente, as codificações 1110 e 1111 estão livres, pelo que se reserva o opcode primário 1110 para esta instrução.

O opcode primário 1111 ainda está livre, o que permitiria reservá-lo para codificar a instrução ADDM (R<sub>d</sub>), k, que assim poderia ter 8 bits para a constante k (e 4 bits para R<sub>d</sub>). No entanto, para ilustrar o caso de uma instrução com opcode primário e secundário vamos atribuir-lhe o opcode primário 0101 (já usado para as instruções aritméticas), que ainda tem dois opcodes secundários livres, o 1110 e o 1111 (por inspecção da Tabela A.9).

O formato da instrução máquina para cada uma destas instruções está representado na Fig. 7.8.

| (a) | 0101 | 1110           | R <sub>s</sub> | R <sub>d</sub> |
|-----|------|----------------|----------------|----------------|
| (b) | 1110 | R <sub>c</sub> | R <sub>s</sub> | R <sub>d</sub> |

Fig. 7.8 - Formato da instrução máquina para as novas instruções definidas.

(a) – ADDM [R<sub>d</sub>], k; (b) – SUM R<sub>c</sub>, [R<sub>s</sub>], R<sub>d</sub>

A ROM de mapeamento não é mais do que uma tabela que para cada instrução máquina no PEPE (identificada pelo seu opcode) indica qual o endereço (na ROM de microcódigo) onde está a primeira microinstrução da sequência que a implementa. Para contemplar todos os casos, esta ROM tem 8 bits de endereço, usando tanto o opcode primário como secundário. As 16 entradas de uma instrução só com opcode primário apontam para a mesma microinstrução.

A Tabela 7.9 ilustra este mapeamento, usando as instruções da Tabela 7.7 e da Tabela 7.8 e as codificações da Tabela A.9 e da Fig. 7.8. A tabela completa tem 256 entradas, correspondentes aos 8 bits dos dois opcodes. A Tabela 7.9 é apresentada em formato umbólico, com etiquetas em vez de endereços da ROM de microcódigo, mas naturalmente o microassembler fará a correspondência e gerará automaticamente o conteúdo da ROM de mapeamento.

| INSTRUÇÃO                                              | NÚMERO DE ENTRADAS NA ROM DE MAPEAMENTO | OPCODE | MICRO-INSTRUÇÃO INICIAL |
|--------------------------------------------------------|-----------------------------------------|--------|-------------------------|
| ADD R <sub>d</sub> , R <sub>s</sub>                    | 1                                       | 0101   | m_ADD                   |
| ADD R <sub>d</sub> , k                                 | 1                                       | 0101   | m_ADDI                  |
| SHR R <sub>d</sub> , n                                 | 1                                       | 0110   | m_SHR                   |
| MOVL R <sub>d</sub> , k                                | 16                                      | 1100   | m_MOVL                  |
| MOV R <sub>d</sub> , [R <sub>s</sub> +R <sub>t</sub> ] | 16                                      | 1000   | m_LDR1                  |
|                                                        |                                         | 1001   | m_LDR1                  |
| MOV [R <sub>d</sub> +n], R <sub>s</sub>                | 16                                      | 1001   | m_STOI                  |
|                                                        |                                         | 1011   | m_STOI                  |
| JZ etiqueta                                            | 1                                       | 0001   | m_JZ                    |
| JMP R <sub>s</sub>                                     | 1                                       | 0000   | m_JMPR                  |
| CALL etiqueta                                          | 16                                      | 0011   | m_CALLI                 |
| NOP                                                    | 1                                       | 0000   | m_NOP                   |
| SWE                                                    | 1                                       | 0000   | m_SWI                   |
| ADDM R <sub>d</sub> , k                                | 1                                       | 0101   | m_ADDM1                 |
| SUM R <sub>c</sub> , [R <sub>s</sub> ], R <sub>d</sub> | 16                                      | 1110   | m_SUM1                  |
|                                                        |                                         | 1110   | m_SUM1                  |

Tabela 7.9 - Exemplos de mapeamento entre instruções e sequências de microinstruções, a usar na ROM de mapeamento. Apenas algumas instruções estão representadas.

ote-se que as microinstruções que efectuam a busca de instruções não estão incluídas na Tabela 7.9, pois não correspondem a uma instrução que esteja no RI e seja preciso saber onde se começa a executar. A sequência de busca de instruções é invocada directamente a partir de outras microinstruções. O mesmo se passa em relação às microinstruções que implementam a invocação da rotina de atendimento de exceções (m\_EXC1 em diante), pois este atendimento é feito em hardware e não como resultado de uma instrução SWE dessa sim, está na tabela).

Finalmente, é preciso indicar ao *assembler* que as duas novas instruções existem, com que formato (mnemónica e operandos) e que instrução máquina gerar quando uma delas aparecer no programa. Tal é feito pela directiva *DEF*, com indicação da mnemónica, do código de operação (*opcode*) a gerar e da estrutura e sintaxe dos parâmetros da instrução. A instrução máquina a gerar pelo *assembler* tem 16 bits e está dividida em partes de 4 bits cada, tal como indicado na Fig. 7.8. Por exemplo, estas duas instruções podem ser definidas por:

```
ADDM DEF C8=5EH [R], C4
SUB DEF C4=0EH R, [R], R
```

C8 indica uma constante de 8 bits (no caso de ADDM é o *opcode*), [R] designa um dos registos do PEPE entre parênteses rectos (indicando um acesso à memória), R designa um dos registos e C4 um valor de 4 bits, a especificar pela instrução. Note-se que o *opcode* da instrução SUB, 0EH é de apenas 4 bits. O zero destina-se apenas a distinguir a constante numérica de um identificador.

A documentação do PEPE, no site de apoio do livro, contém as regras completas para poder definir novas instruções.

#### ESSENCIAL

- A Unidade de Excepções recebe sinais de vários pontos do processador (com indicação de excepção, incluindo as interrupções. Prioriza-as (para só gerar uma de cada vez) e em alguns casos memoriza os pedidos de excepção. Sempre que quer gerar uma excepção, activa um sinal para a Unidade de Controlo.
- A Unidade de Controlo é tipicamente microprogramada, em que cada instrução é implementada por uma ou mais microinstruções. Cada microinstrução é uma combinação de valores de todos os sinais que controlam os recursos internos do processador e é executada num só ciclo de relógio.
- Esta unidade inclui uma ROM com as microinstruções e um registo (MPC) que endereça essa ROM, indicando que microinstrução está a ser executada. Em cada ciclo de relógio, o MPC é incrementado (passando à microinstrução seguinte) ou salta para outra microinstrução. Este salto pode ser condicional, dependendo de algumas condições que a própria microinstrução em execução selecciona.

#### SIMULAÇÃO – MICROPROGRAMAÇÃO

Esta simulação toma como base o conteúdo desta secção e das anteriores e exemplifica como a microprogramação funciona, utilizando o próprio PEPE no simulador, que permite:

- Visualizar o microprograma, de forma simbólica;

- Executar as microinstruções passo a passo (uma microinstrução de cada vez), examinando o estado dos recursos de hardware após a execução de cada microinstrução;
- Modificar as microinstruções já definidas, alterando a funcionalidade das instruções do PEPE;
- Definir novas instruções e implementá-las em microprograma, mesmo complexas (incluindo por exemplo ciclos nas microinstruções), desde que a arquitetura possua os necessários recursos de hardware para as implementar;
- Indicar ao *assembler* a estrutura das novas instruções, de modo a que este as reconheça e permita escrever programas de linguagem assembly usando as novas instruções.

### 7.3 PROCESSAMENTO EM ESTÁGIOS

#### 7.3.1 PRINCÍPIOS DE FUNCIONAMENTO

Olhando para a Tabela 7.7, pode notar-se que qualquer instrução demora no mínimo três ciclos de relógio a ser processada. Veja-se o exemplo da instrução ADD Rd, Rs, que obrigatoriamente envolve as seguintes microinstruções:

- m\_BSC1 – Faz a busca da instrução para o RI (Registo de Instruções);
  - m\_BSC2 – Faz a descodificação da instrução (indicando o endereço na ROM de microcódigo da primeira microinstrução que executa a instrução) e prepara o PC para a próxima instrução (somando-lhe 2 unidades);
  - m\_ADD – Microinstrução que executa a instrução ADD.
- Esta instrução gasta apenas um ciclo de relógio para a sua execução, correspondente à microinstrução m\_ADD, mas no total são gastos três ciclos de relógio (por causa da busca da instrução e da sua descodificação), o que revela uma eficiência de apenas 33%. Este facto tem um grande impacte no desempenho do processador.
- Existem aqui três tipos de operações diferentes mas que têm de ser executadas sequencialmente porque cada uma depende do resultado da anterior:
- B – Busca de instrução;
  - D – Descodificação e actualização do PC (estas duas operações são independentes e podem ser executadas simultaneamente);
  - E – Execução da instrução.

A Fig. 7.9a ilustra este funcionamento sequencial, considerando a execução de quatro instruções do género do ADD (em que cada operação demora 1 ciclo de relógio a executar). Cada instrução demora assim 3 ciclos e só de 3 em 3 ciclos é que uma instrução acaba. As quatro instruções desta figura demoram 12 ciclos de relógio a ser totalmente processadas. A unidade de controlo da Fig. 7.6 funciona desta forma, executando à vez

cada uma das microinstruções. Só após acabar a execução de uma instrução é que passa à seguinte.

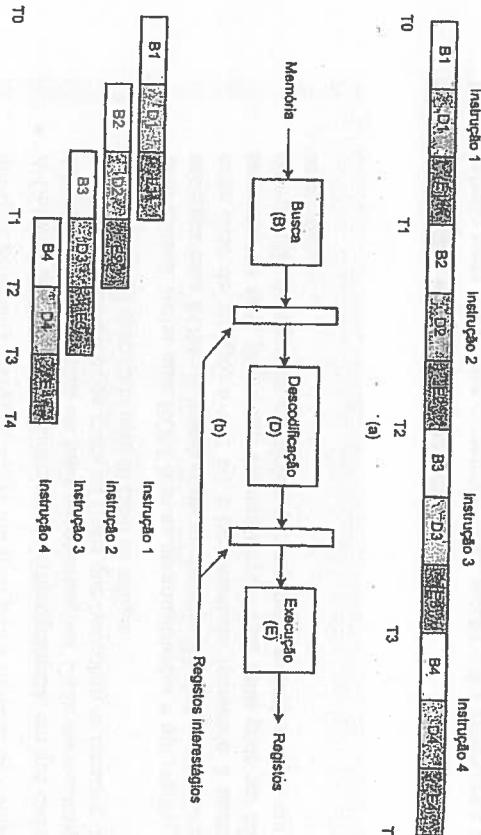


Fig. 7.9 - Princípio do processamento em estágios. (a) – Processamento sequencial (uma só instrução de cada vez); (b) – Cadeia de estágios, com unidades especializadas (registos interestágio); (c) – Processamento em estágios

No entanto, se cada uma destas operações for executada por uma unidade de *hardware* diferente, estas operações podem operar como uma linha de montagem industrial, em que, mal acaba a sua operação, cada unidade passa o seu resultado à unidade seguinte e pode imediatamente começar a executar a mesma operação sobre a instrução seguinte, ainda antes de a instrução anterior ter acabado totalmente a sua execução.

Esta organização designa-se **processamento em estágios** (*pipelining*), em que o processamento total de uma instrução é dividido em tarefas mais pequenas (estágios), executadas por unidades especializadas numa só tarefa (operação), a do estágio que implementam. O conjunto destas unidades em linha designa-se **cadeia de estágios** (*pipeline*).

A Fig. 7.9b ilustra esta organização com as três operações referidas (B, D e E). Cada unidade passa o seu resultado à unidade seguinte através de um registo interestágio, cuja única função é guardar temporariamente esse resultado, permitindo a sua evolução temporal e espacial ao longo da cadeia, de acordo com o ritmo de evolução desta. Assume-se que o estágio de execução guarda o resultado final nos registos (que é o caso da maior parte das instruções, embora o destino de algumas seja a cache/memória). A Fig. 7.9c mostra o que sucede na escala de tempos, ao longo da execução das operações, podendo notar-se que:

Há três unidades que podem funcionar simultaneamente, ao contrário do que acontece na Fig. 7.9a, em que só há uma unidade (a de controlo);

Cada instrução continua a demorar 3 ciclos de relógio a ser executada, desde a primeira até à última operação;

Como cada unidade não espera que as outras acabem mas comece logo a trabalhar na próxima instrução mal acabe a sua operação, acaba uma nova instrução em cada ciclo de relógio e não de 3 em 3. É como se cada instrução demorasse apenas um ciclo de relógio a ser executada!

No entanto, tal só acontece enquanto a cadeia está "cheia", isto é, com todas as unidades ocupadas. Neste exemplo (Fig. 7.9c), esta situação só ocorre nos dois ciclos de relógio à volta de T1. No início há uma fase de encherimento da cadeia, em que ainda não há resultados, razão pela qual o primeiro resultado só sai em T1, 3 ciclos de relógio após o início da sequência. No fim há uma fase de esvaziamento da cadeia, em que deixam de entrar instruções para executar e as unidades vão sucessivamente ficando inactivas;

O tempo total de processamento foi substancialmente reduzido, para metade (neste exemplo), o que ilustra o potencial impacte do processamento em estágios no desempenho de um computador.

Uma cadeia com N estágios proporciona como limite teórico uma melhoria de desempenho de N vezes, pois possui N unidades a funcionar simultaneamente, mas na prática tal é impossível de conseguir devido a algumas restrições que não deixam a cadeia estar sempre cheia (reduzindo a sua eficiência), nomeadamente as seguintes:

- Dependências entre instruções que impeçam a execução simultânea das suas operações. Com a cadeia cheia, num dado instante estarão N instruções em execução, em diversos estágios de evolução da cadeia. Poderá suceder, por exemplo, que uma instrução precise de um valor que a anterior ainda não produziu, pois ainda não acabou de se executar. Neste caso, a instrução que não pode prosseguir tem de empatar a cadeia, atrasando o seu processamento (e o das instruções seguintes) do número de ciclos de relógio necessário até o valor necessário estar disponível;

- Tipicamente, a cadeia de estágios lê instruções da cache de instruções (Fig. 7.6 e secção 7.5), o que pode fazer com que às vezes essa leitura demore mais tempo do que o habitual (quando a instrução pretendida não está carregada na cache). Durante esse tempo (até a instrução ser lida da memória e carregada na cache, o que a cache consegue autonomamente), o processador tem de esperar que a instrução esteja disponível. Uma solução seria parar a cadeia de estágios, mas isso faria parar também as instruções que já estão em processamento para a frente na cadeia. Por isso, a solução geralmente adoptada consiste em introduzir na cadeia um estádio de inactividade (ou mais, se necessário), que se vão propagando ao longo da cadeia, razão pela qual se designam *bolhas* (*bubbles*). Assim, conseguem empatar a cadeia sem afetar as tarefas que já estavam em execução. A Fig. 7.10 ilustra esta técnica com a introdução de uma bolha antes da instrução 2

(assumindo que esta instrução não estava carregada na cache na altura do acesso), provocando uma falha no acesso à cache que demora algum tempo a recuperar.

A cadeia é assim atrasada de um ciclo, em que a bolha de inactividade vai passando de unidade em unidade como se de uma instrução normal se tratasse (mas que não faz nada<sup>104</sup>). O único efeito é a redução da eficiência face ao valor máximo possível. A cache de dados também pode ter falhas nos acessos de dados (leitura ou escrita), mas nesse caso tem mesmo de se parar toda a cadeia de estágios, até que os dados estejam carregados na cache. A interacção com a cache de dados é feita no fim da cadeia e, se esta não parasse, as instruções que viriam a seguir estragariam o contexto do estágio que estivesse à espera do acesso à cache de dados. No PEPE, a paragem é conseguida retirando temporariamente o relógio à unidade de controlo (ao MPC), pelo que durante esse tempo pára tudo (Fig. 7.6).

As instruções executadas não constituem sempre uma sequência linear. Há instruções de salto, chamadas a rotinas e exceções, que mudam o rumo do fluxo de controlo em relação ao que era esperado (alteram o valor do PC). Neste caso, a execução das instruções seguintes, que já estavam nos estágios iniciais, tem de ser abandonada. A cadeia tem de ser esvaziada e reconhecer o seu encheramento, o que é ilustrado na Fig. 7.10 em que se supõe que a instrução 2 é um salto. As bolhas introduzidas nas instruções 3 e 4 fazem com que o trabalho já feito nestas instruções seja ignorado (só no estágio E2 se sabe que era um salto). A cadeia só começa a encher de novo com a instrução 5 (no endereço para onde a instrução 2 saltou), no instante T3, e só no instante T5 que a cadeia volta a ficar cheia.

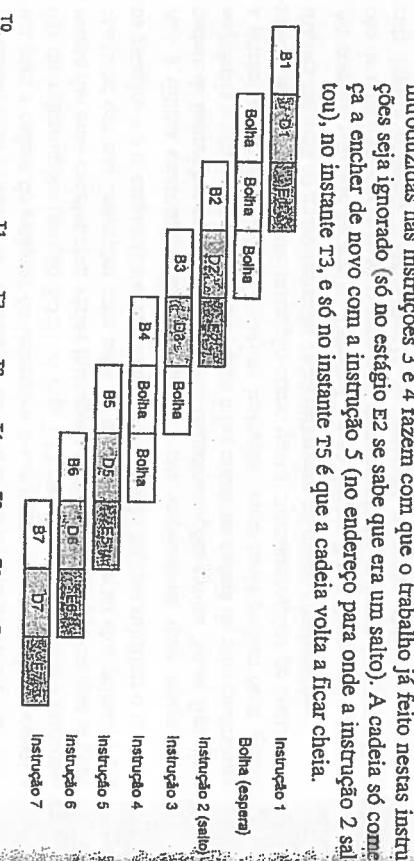


Fig. 7.10 - Efeito das bolhas na cadeia de estágios. Nos instantes T2, T4 e T5 nenhuma instrução fica pronta, perdendo-se eficiência

Embora cada processador lide com a técnica de processamento em estágios de forma diferente (o número de estágios não é sempre o mesmo e há outras especificidades), os princípios básicos são os mesmos.

<sup>104</sup> Para este efeito usa-se a microinstrução `M_NOP`, que implementa a instrução `NOP` e em que todos os sinais estão inactivos.

Nem todas as instruções têm o mesmo tempo de execução (no estágio E). Por exemplo, na Tabela 7.7 pode verificar-se que a instrução ADD demora apenas um ciclo de relógio, enquanto a instrução CALL demora quatro. Numa cadeia de estágios, a cadênciia de evolução tem de ser igual para todas as unidades, uma vez que a entrada de uma liga à saída da anterior. A mais lenta domina e as outras têm de esperar. Por conseguinte, é muito importante dividir tarefas complexas noutras mais pequenas (mesmo que tal aumente o número de estágios da cadeia) de tempo de execução aproximadamente igual, de modo a que não haja um estágio muito mais lento do que os outros que os restantes e a frequência do relógio possa ser a mais alta possível. Este aspecto é fundamental para os processadores microprogramados, como o PEPE, pois o número de microinstruções necessárias varia de instrução para instrução.

Por outro lado, coloca-se a questão de as tarefas a considerar para cada estágio serem ao nível de cada instrução ou de cada microinstrução. A solução adoptada no PEPE é considerar duas cadeias de estágios, tal como ilustrado pela Fig. 7.11:

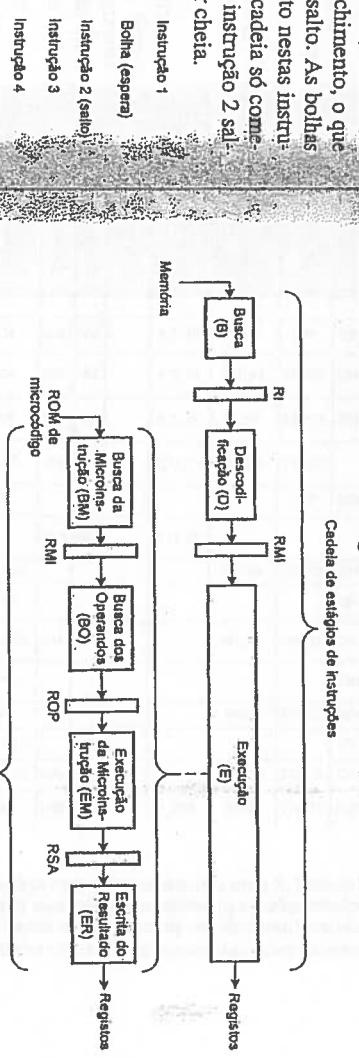


Fig. 7.11 - Estrutura das cadeias de estágios no PEPE. A cadeia de estágios de microinstruções pode evoluir várias vezes durante um estágio da cadeia de instruções

- Cadeia de estágios de instruções – Correspondente à da Fig. 7.9 e que evolui de cada vez que se faz a busca de uma nova instrução, a um ritmo não constante (porque depende do tempo de execução de cada instrução). Esta cadeia tem três estágios, o que significa que em qualquer instante podem estar em processamento três instruções diferentes (em estágios diferentes);
- Cadeia de estágios de microinstruções – Implementa o estágio de execução de uma instrução, executando as várias microinstruções em regime de processamento de estágios. Esta cadeia é de mais baixo nível, funciona à velocidade máxima a um ritmo constante (evolui em cada ciclo do relógio do processador) e inclui quatro estágios, o que permite ter quatro microinstruções a ser processadas simultaneamente, mas em diferentes estágios de evolução.

Note-se que:

- A informação que passa nos registos interestágio não consiste apenas de dados, mas também de sinais de controlo;

  - Os estágios D (Descodificação) e BM (Busca de Microinstrução) são em boa parte coincidentes, pois o resultado da descodificação de uma instrução é a obtenção (busca) da primeira microinstrução da sequência que implementa essa instrução.
  - O estágio BM propriamente dito só existe nas instruções que são implementadas por mais do que uma microinstrução. Desta forma, os registos interestágio que terminam estes estágios são também o mesmo, embora parte da informação evoluia com a cadeia de estágios de instruções e a restante com a cadeia de estágios das microinstruções (seção 7.3.3). A Fig. 7.13 na página 615 ilustra o funcionamento das duas cadeias com várias instruções ao longo do tempo, em que se pode constatar que nas instruções com mais de uma microinstrução só a primeira tem os estágios B e D, enquanto nas seguintes só há estágio BM, que substitui o D;
  - Até à última microinstrução da sequência que implementa uma instrução, a cadeia de instruções está parada e só a de microinstruções evolui. Só no fim dessa sequência a cadeia de instruções avança e faz entrar na cadeia de microinstruções a primeira microinstrução da próxima instrução, onde tanto podem estar quatro microinstruções da mesma instrução como quatro microinstruções de quatro instruções (de uma só microinstrução cada) diferentes ou outras combinações;
  - Ao contrário do que sucede na Tabela 7.7 e na Tabela 7.8, a última microinstrução (activando o sinal MAP – Tabela 7.5) a partir dessa instrução que entre tanto entrou na cadeia de instruções. A última microinstrução de cada instrução faz o mapeamento da próxima instrução. A Tabela 7.10 e a Tabela 7.11 apresentam novas versões da Tabela 7.7 e da Tabela 7.8, respectivamente, com esta alteração. Os saltos em microcódigo ficam agora reservados para as instruções que explicitamente precisarem deles, como por exemplo a instrução SUM na Tabela 7.11 (em M\_SUM). O mapeamento condicional na microinstrução M\_SUM2 decide se a instrução termina (fazendo o mapeamento para a seguinte) ou prossegue no seu ciclo interno. Note-se que a Tabela 7.11 ainda tem de ser corrigida em termos de dependência de dados, assunto que será tratado na secção 7.3.5;
  - Já não há sinal ESCR\_RI. É o sinal MAP que faz escrever nos registos interestágio da cadeia de instruções e a faz avançar;
  - A funcionalidade de um processador não muda pelo facto de suportar processamento de estágios ou não. O processamento em estágios é apenas uma forma de aumentar o desempenho, permitindo que tarefas que antes tinham de ser executadas sequencialmente agora possam ser executadas simultaneamente, por haver suporte em hardware para tal. O PEPE permite ligar e desligar este mecanismo, mas o seu microcódigo é basicamente igual com e sem estágios, com apenas algumas diferenças ao nível da gestão e da transição entre microinstruções.

**Tabela 7.10 - Adequação da microprogramação da Tabela 7.7 para processamento com cadeia de estágios.** A última microinstrução de uma instrução faz o mapeamento da primeira microinstrução da próxima instrução, que já percorreu a cadeia de estágios de instruções. As microinstruções  $m_{BSC1}$  e  $m_{BSC2}$  desapareceram (deixa de ser preciso gastar tempo com a busca explícita das instruções), tal como o sinal  $ESCR\_RI$ . As últimas três colunas foram alteradas e o resto da tabela manteve-se sem alterações

| INSTITUIÇÃO | NOME        | MICRO-OPERAÇÃO | SEL_A                                                  | REG_A   | SEL_OP_A | REG_A | SEL_B   | REG_B   | SEL_C   | REG_C | SEL_BARR | REG_C | SEL_ALU | REG_ALU | SEL_B2 | REG_B2 | SEL_BCR | REG_BCR | SEL_MICRO_BALTO | REG_BCR |
|-------------|-------------|----------------|--------------------------------------------------------|---------|----------|-------|---------|---------|---------|-------|----------|-------|---------|---------|--------|--------|---------|---------|-----------------|---------|
|             | NOP         | m_NOP          | MPC←MAP[opcode]                                        |         |          |       |         |         |         |       |          |       |         |         |        |        |         |         | SIM             | SALTA   |
| ADD         | Rd, Rs      | m_ADD          | Rd ← Rd + Rs;<br>MPC←MAP[opcode]                       | RI_7_4  | REG      | REG   |         | RI_7_4  | SIM     | ALU   | ADD      | ZNCV  |         |         |        |        |         | SIM     | SALTA           |         |
| ADD         | Rd, k       | m_ADDI         | Rd ← Rd + k;<br>MPC←MAP[opcode]                        | RI_7_4  | REG      | CONST | E4_16S  | RI_7_4  | SIM     | ALU   | ADD      | ZNCV  |         |         |        |        |         | SIM     | SALTA           |         |
| SHR         | Rd, n       | m_SHR          | Rd ← Rd >> k;<br>MPC←MAP[opcode]                       | RI_7_4  | REG      | CONST | E4_16   | RI_7_4  | SIM     | ALU   | SHR      | ZNC   |         |         |        |        |         | SIM     | SALTA           |         |
| MOVL        | Rd, k       | m_MOVL         | Rd ← k(7)(8)    k;<br>MPC←MAP[opcode]                  |         |          | CONST | E8_16S  | RI_11_8 | SIM     | ALU   | OP_B     |       |         |         |        |        | SIM     | SALTA   |                 |         |
| MOV         | Rd, [Rs+RJ] | m_LDRI         | REM ← Rs + RJ                                          | RI_7_4  | REG      | REG   |         |         |         |       | ADD      |       |         |         |        |        | SIM     |         |                 |         |
|             |             | m_LDR2         | REM ← Mw[REM];<br>MPC←MAP[opcode]                      |         |          |       |         |         | RI_31_8 | SIM   | MEM      |       |         |         |        |        | SIM     |         | SIM             | SALTA   |
|             |             | m_STO1         | REM ← Rd + 2 <sup>k</sup> (N/2)                        | RI_7_4  | REG      | CONST | E4_16S  |         |         |       | ADD_Bn2  |       |         |         |        |        | SIM     |         |                 |         |
| MOV         | [Rd+n], Rs  | m_STO2         | Mw[REM] ← Rs;<br>MPC←MAP[opcode]                       | RI_11_8 | REG      |       |         |         |         |       | OP_A     |       | SIM     |         |        |        | SIM     |         | SIM             | SALTA   |
| JZ          | etiqueta    | m_JZ           | Z=1: PC ← PC + 2 <sup>k</sup> diff;<br>MPC←MAP[opcode] |         | PC       | CONST | E8_16S  |         |         | ALU   | ADD_Bn2  |       |         |         |        |        | Z       | SIM     | SALTA           |         |
| IMP         | Rs          | m_IMPR         | PC ← Rs;<br>MPC←MAP[opcode]                            | RI_3_0  | REG      |       |         |         |         |       | OP_A     |       |         |         |        |        | SALTA   | SIM     | SALTA           |         |
| CALL        | etiqueta    | m_CALL1        | REM ← SP - 2                                           | REG_A   | SP       | REG   | CONST   | DOIS    |         |       | SUB      |       |         |         |        |        | SIM     |         |                 |         |
|             |             | m_CALL2        | Mw[REM] ← PC                                           |         | PC       |       |         |         |         |       | OP_A     |       | SIM     |         |        |        |         |         |                 |         |
|             |             | m_CALL3        | PC ← PC + 2 <sup>k</sup> diff                          |         | PC       | CONST | E12_16S |         |         | ALU   | ADD_Bn2  |       |         |         |        |        | SALTA   |         |                 |         |
|             |             | m_CALL4        | SP ← SP - 2;<br>MPC←MAP[opcode]                        | REG_A   | SP       | REG   | CONST   | DOIS    | REG_C   | SP    | SIM      | ALU   | SUB     |         |        |        |         | SIM     | SALTA           |         |

**Tabela 7.11 - Adequação da microprogramação da Tabela 7.8 para processamento com cadeia de estágios.** O salto para a busca de instruções ( $m_{BSC1}$ ) foi substituído pelo mapeamento (com o sinal  $MAP$ ) da próxima instrução. A microinstrução  $m_{SUM2}$  tem um mapeamento condicional. O resto da tabela manteve-se sem alterações, mas ainda tem de ser corrigida em termos de dependência de dados. (seção 7.3.5).

Os registos interestágio são os seguintes:

- RI – Registo de Instruções**, que memoriza a última instrução lida da cache de instruções;

**RMI – Registo de Microinstruções**, que abrange as duas cadeias de estágios. Na parte da cadeia de instruções contém informação sobre os operandos da instrução em execução (índices de registos e/ou uma constante), enquanto na parte da cadeia de microinstruções corresponde ao MPC e mantém o endereço na ROM de microcódigo da microinstrução em execução. O MPC evolui à velocidade do relógio, enquanto a parte dos operandos evolui apenas quando a instrução em execução muda;

### 7.3.3 IMPLEMENTAÇÃO DAS CADEIAS DE ESTÁGIOS

A implementação do processamento em estágios num processador envolve, primeiro que tudo, decidir que estágios existem e onde devem ser colocados os registos interestágio. No caso do PEPE, a Fig. 7.11 já define a estrutura dos estágios. A Fig. 7.12 é semelhante à da Fig. 7.6, em que os sinais gerados pela unidade de controlo foram omitidos (por simplicidade) e os registos interestágio foram introduzidos.

Estes registos dividem toda a arquitectura em secções (estágios), abrangendo a parte de dados e de controlo. Sempre que uma cadeia evolui, os sinais memorizados num dos registos interestágio e transformados pelos circuitos intermédios (caso da ALU, por exemplo) são memorizados no registo interestágio seguinte, num deslocamento constante e sincronizado da esquerda para a direita. Cada instrução/microinstrução vai sendo sucessivamente processada em cada estágio, até ter passado por todos os estágios da cadeia. Todos os registos de uma cadeia de estágios são escritos ao mesmo tempo (os sinais de escrita estão ligados entre si).

### **7.3.3.1 CADEIA DE ESTÁGIOS DE INSTRUÇÕES**

A zona cinzenta na Fig. 7.12 corresponde à cadeia de estágios das instruções, cujo objectivo básico é o de realizar a funcionalidade das microinstruções  $m_{BSC1}$  (busca) e  $m_{BSC2}$  (mapeamento e actualização do PC) da Tabela 7.7, mas de tal forma que quando se acaba de executar a última microinstrução de uma dada instrução já a busca e o mapeamento da instrução seguinte estão feitos e não há intervalo entre execução de instruções. Ou seja, a execução de uma instrução sobrepõe-se no tempo à busca e mapeamento das instruções seguintes. As microinstruções  $m_{BSC1}$  e  $m_{BSC2}$  não são usadas com processamento em estágios.

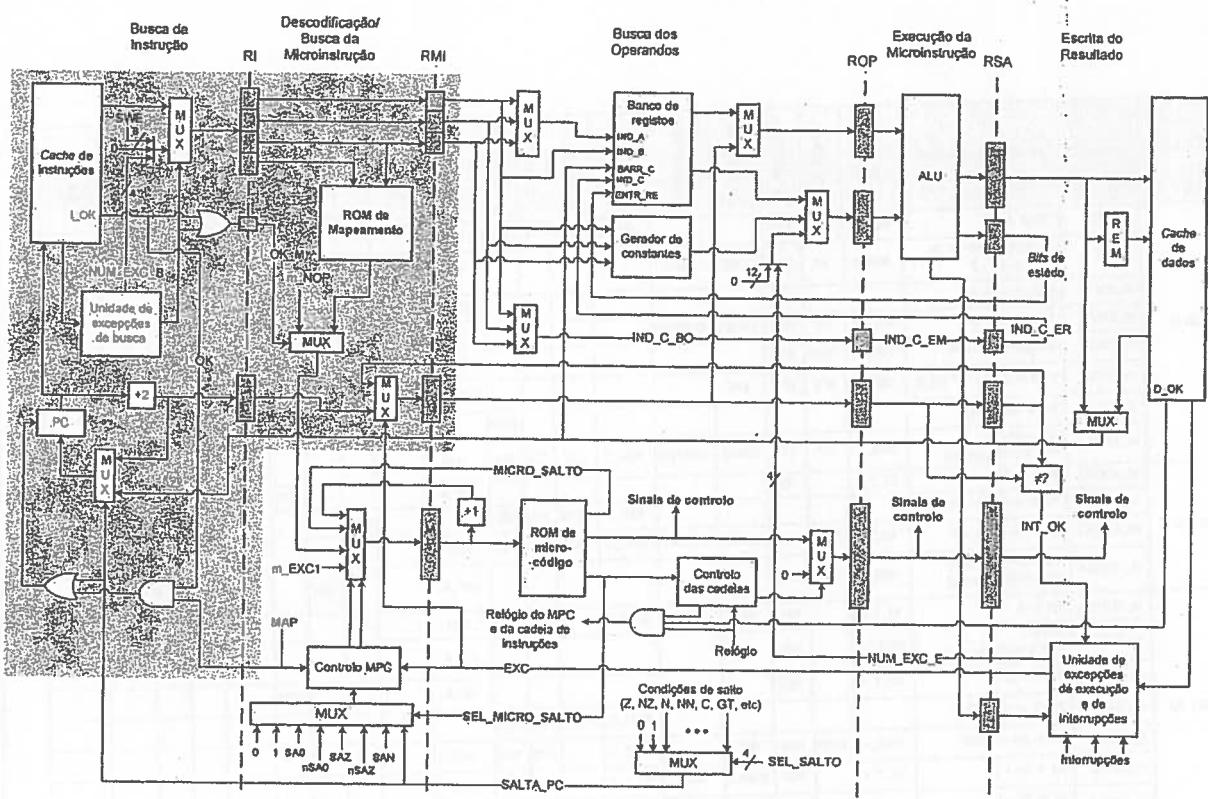


Fig. 7.12 - Arquitectura geral do núcleo do PEPE com cadeias de estágios.  
A zona cinzenta corresponde à cadeia de estágios de instruções

A cadeia de estágios de instruções só evolui no último ciclo de relógio (microinstrução) da execução de qualquer instrução. PC, RI e RMI são actualizados ao mesmo tempo (cada um lê o valor do anterior antes de a saída deste mudar). Quando esta cadeia evolui:

- No estágio D (Descodificação), os operandos da instrução no endereço  $x$  são escritos na parte de dados do RMI (registo de microinstruções) e o endereço (na ROM de microcódigo) da primeira microinstrução que implementa esta instrução é carregado no MPC, que é a parte de controlo do RMI;

No estágio B (Busca de Instrução), uma nova instrução (no endereço  $x+2$ ) entra no RI;

- O PC é actualizado para  $x+4$ .

O PC vai assim em avanço em relação à instrução em execução, o que significa que tem de ser propagado ao longo da cadeia (Fig. 7.12), de modo a acompanhar a instrução respectiva (necessário caso a instrução use o PC ou haja uma exceção). Numa dado instante, cada estágio está a processar uma instrução diferente e cada uma tem de ser acompanhada pelo respectivo valor de PC (o valor que o PC teria durante a execução dessa instrução se não houvesse processamento em estágios e que é igual ao endereço dessa instrução mais 2).

Esta sequência crescente do PC ao longo do tempo pode ser interrompida pelas seguintes razões:

A cadeia evoluiu antes de a cache ter tido tempo de ir buscar à memória a instrução endereçada pelo PC desde a última vez que este foi alterado. Nessa altura, no RI entra um valor incorrecto, mas  $I_{OK\_M}$  fica inactivo (a 0) e no próximo ciclo de relógio o que entra no MPC é  $m\_Nop$ , que implementa uma bolha de inactividade. A situação prolonga-se até que a cache recupere a instrução endereçada pelo PC, cujo valor foi entretanto impedido de evoluir pelas portas lógicas ligadas ao seu sinal de escrita (Fig. 7.12);

Sempre que o PC é alterado (numa instrução de salto, por exemplo), esta cadeia tem de ser esvaziada, pois as instruções que nela já estavam em processamento, em antecipação para serem executadas a seguir, terão de ser descartadas. O programa deve prosseguir a partir da instrução indicada pelo novo PC e não nos endereços seguintes ao da instrução que provocou a alteração do PC. "Esvaziar" significa preencher os vários registos interestágio com sinais inactivos, simulando uma instrução sem efeitos no estado do processador (nop), e colocar  $I_{OK\_M}$  a 0 (até RI ser carregado com a instrução endereçada pelo novo valor do PC). Por simplicidade, os circuitos que implementam o esvaziamento não estão representados na Fig. 7.12;

Durante a busca de uma instrução pode ocorrer uma de várias situações de erro ( $I_{DESLINHADO}$ ,  $I_{FALTA\_PAG}$  ou  $I_{PROT}$  – ver Tabela A.8, na página 702). A solução neste caso não pode ser gerar uma exceção mal a situação ocorre, pois o PC vai em avanço e a instrução que gerou o problema pode nunca chegar a

ser executada, pois entretanto pode haver um salto. Assim, a solução é substituir essa instrução ao entrar no RI por uma instrução SWI (que gera uma exceção com o número de exceção adequado (o sinal NUM\\_EXC\_B fornece os quatro bits de menor peso da instrução). Se essa "instrução" chegar a ser processada, gera-se a exceção correspondente. Se entretanto houver um salto, nada acontece porque a cadeia é esvaziada. Pelo facto de estas exceções serem inseridas na cadeia de estágios esperando a sua vez, sem abortar as instruções já em processamento, o seu atendimento designa-se encadeado. A unidade de exceções foi dividida em duas, uma para tratar deste aspecto das exceções de busca de instrução e a outra para tratar das restantes exceções (erros de execução e interrupções).

A Tabela 7.12 ilustra a evolução dos registos interestágio desta cadeia de estágios, longo do tempo, durante a execução do Programa 7.1, assumindo que as suas instruções se encontram nos endereços indicados e que o endereço no microcódigo da microinstrução que implementa cada instrução é representado pelo símbolo "m\_" seguido da mnemónica, tal como exemplificado na Tabela 7.7 e na Tabela 7.8.

|       |      |                                                       |                                                  |  |  |  |
|-------|------|-------------------------------------------------------|--------------------------------------------------|--|--|--|
| 1000H | ADD  | R1, R2                                                |                                                  |  |  |  |
| 1002H | SHR  | R1, 3                                                 |                                                  |  |  |  |
| 1004H | JNZ  | L                                                     |                                                  |  |  |  |
| 1006H | ADD  | R1, 2                                                 | ; salto condicional para o endereço L (PC + 1AH) |  |  |  |
| 1008H | OR   | R7, R8                                                | ; não chega a ser executada se houver salto      |  |  |  |
| 100AH | SUB  | R6, 4                                                 |                                                  |  |  |  |
| 1020H | L:   | MOV R2, [R1+R3]                                       |                                                  |  |  |  |
| 1022H | ADDM | [R2], 2 ; nova instrução. Soma 2 ao conteúdo de M[R2] |                                                  |  |  |  |
| ...   |      |                                                       |                                                  |  |  |  |

Programa 7.1 - Sequência de instruções para exemplificar o processamento em estágios.

| TEMPO | PC    | R1              | R2    | R3         | RMI   | RMF    |
|-------|-------|-----------------|-------|------------|-------|--------|
| T0    | 1000H | X               | X     | X          | X     | X      |
| T1    | 1002H | ADD R1, R2      | 1002H | X          | X     | X      |
| T2    | 1004H | SHR R1, 3       | 1004H | R1, R2     | 1002H | m_ADD  |
| T3    | 1006H | JNZ 1AH         | 1006H | R1, 3      | 1004H | m SHR  |
| T4    | 1008H | ADD R1, 2       | 1008H | 1AH        | 1006H | m_JNZ  |
| T5    | 1020H | NOP             | 1020H | 0          | 1020H | m_NOP  |
| T6    | 1022H | MOV R2, [R1+R3] | 1022H | 0          | 1020H | m_NOP  |
| T7    | 1024H | ADDM [R2], 2    | 1024H | R2, R1, R3 | 1022H | m_LDR  |
| T8    | 1026H | ...             | 1026H | R2, 5      | 1024H | m_ADDM |

Tabela 7.12 - Exemplo de evolução da cadeia de estágios de instruções. "X" refere-se a valores anteriores que são irrelevantes para este exemplo. A zona cinzenta em T5 e T6 indica o esvaziamento da cadeia com simulação de NOPs.

Os tempos indicados referem-se aos instantes imediatamente após cada evolução da cadeia (e não a uma escala linear de tempo). Neste exemplo, algumas instruções são simples e são executadas num só ciclo de relógio, mas outras (Mov e ADDM) são mais complexas e demoraram vários ciclos de relógio a ser executadas. A Tabela 7.13 detalha esta tabela com todas as operações em cada ciclo de relógio.

Em T1, a instrução ADD (localizada no endereço 1000H) já entrou no RI e o PC já foi actualizado para 1002H. O valor do PC propagado é também 1002H, pois durante a execução de cada instrução o PC aponta já para a instrução seguinte (tal como quando não há cadeias de estágios).

No entanto, na transição de T4 para T5 a instrução JNZ é executada, numa altura em que o ADD R1, 2 já tinha entrado na cadeia, em antecipação à espera de ser executada a seguir. Este exemplo assume que a instrução JNZ salta. Como o PC é alterado, a cadeia tem de ser esvaziada para descartar as instruções que assim não serão executadas, com dois aspectos muito importantes:

- A nova instrução, o destino do salto (no endereço L, neste exemplo), tem de passar pelos dois estágios da cadeia antes de chegar à cadeia das microinstruções. Para preencher o vazio nestes dois estágios após o esvaziamento, inserem-se bolhas de inactividade, simulando NOPs, que depois são propagados pela cadeia de microinstruções, mas sem alterar o estado do processador (pela definição de NOP);
- O valor do PC propagado destes falsos NOPs tem de ser o valor de L, para que se durante o enchimento da cadeia houver uma interrupção (o que implica novo esvaziamento desta cadeia porque o PC será alterado para a primeira instrução da rotina de atendimento da interrupção), o endereço de retorno memorizado seja L e assim esta instrução possa finalmente ser executada a seguir à instrução de retorno (RET). Este requisito implementa-se facilmente fazendo com que o valor escrito no PC seja também escrito simultaneamente nos dois registos interestágio do PC propagado (mas apenas quando o sinal SALTA\_PC vale 1, indicando um salto no PC). Estes detalhes dos sinais de escrita nestes registos estão omitidos na Fig. 7.12 por simplicidade mas o seu efeito está indicado na Tabela 7.12.

Note-se ainda que na parte dos operandos do RMF não estão os operandos propriamente ditos mas sim os elementos que vão depois permitir obter esses operandos. No caso dos registos, o que lá está são os índices dos registos no banco de registos (0 a 15). No caso das constantes, o que lá está é o valor da constante, mas apenas com 4, 8 ou 12 bits, dependendo da instrução, tendo ainda de passar pelo gerador de constantes para se produzir o operando constante de 16 bits (por extensão dos bits à esquerda).

### 7.3.3.2 CADEIA DE ESTÁGIOS DE MICROINSTRUÇÕES

O estágio D (Descodificação) coincide com a primeira execução do estágio EM (Busca de microinstrução), que já faz parte da cadeia das microinstruções. No caso das instruções implementadas por várias microinstruções, o valor dos operandos no RMI mantém-se durante toda a execução dessas microinstruções, enquanto o MPC vai evoluindo de acordo com a ROM de microcódigo. A Tabela 7.5, na página 583, indica como é que o MPC evolui face às várias situações possíveis (incluindo exceções). É igual ao caso em que não há processamento em estágios.

Cada microinstrução passa sucessivamente pelos vários estágios desta cadeia, em que cada estágio lida apenas com parte dos sinais gerados pela unidade de controlo (Tabela 7.7 e Tabela 7.8) e não com todos ao mesmo tempo. Os sinais de controlo saem todos ao mesmo tempo da ROM de microcódigo, mas têm de ser propagados ao longo da cadeia até ao estágio em que vão ser usados (e no tempo certo), tal como indicado genericamente na Fig. 7.12, acompanhando o sucessivo processamento da microinstrução respectiva ao longo da cadeia.

Com a cadeia cheia, em cada instante há três microinstruções em processamento:

- O estágio BO (Busca de Operandos) produz os operandos em si e armazena-os no registo ROP (Registo dos Operandos);
- O estágio EM (Execução da Microinstrução) executa a operação na ALU e armazena o resultado no registo RSA (Registo de Saída da ALU);
- O estágio ER (Escrita do Resultado) escreve o resultado no sítio definitivo, seja num registo do banco de registos, no registo REM ou na memória (depende da microinstrução). Note-se que, no caso de o destino ser um registo, o seu índice no banco de registos foi propagado ao longo da cadeia (sinais IND\_C\_BO, IND\_C\_EM e IND\_C\_ER na Fig. 7.12), pois entretanto no RER (parte dos operandos) já devem estar outra instrução, a ser processada a seguir, com outro valor de IND\_C. Note-se também que os bits de estado são actualizados ao mesmo tempo que o resultado (no estágio ER) e não no ciclo de relógio em que a ALU os produz.

A Tabela 7.13 reproduz o exemplo da Tabela 7.12, mas agora incluindo as duas cadeias de estágios (arquitetura completa) e mais detalhes, nomeadamente as microinstruções das instruções MOV e ADD. Os tempos marcados duram um ciclo de relógio.

A zona cinzenta a partir de T8 ilustra o esvaziamento das duas cadeias de estágios motivada pela escrita no PC resultante de um salto, notando-se o número de ciclos de relógio que as cadeias necessitam até voltar a encher. As instruções ADD, OR e SUB, que estão a seguir ao JNZ e começam a ser processadas na cadeia de estágios, são simplesmente descartadas se o salto for efectuado (a instrução ADD chega mesmo a fazer a soma, mas não atinge o estágio ER, pelo que não chega a alterar o R1), obrigando a esvaziar toda a cadeia e esperar que encha de novo. Em compensação, se a instrução JNZ não saltar já boa parte do trabalho foi feita. Por este exemplo pode verificar-se o peso que as mudanças no PC (saltos, chamadas de rotina, etc.) têm no desempenho da cadeia de estágios.

| TEMPO | PC    | RI (SAÍDA BO)     |               | RMI (SAÍDA D/BM) |               |         | ROP (SAÍDA BO)      | RSA        | DESTINO ALTERADO COM O RESULTADO |
|-------|-------|-------------------|---------------|------------------|---------------|---------|---------------------|------------|----------------------------------|
|       |       | INSTRUÇÃO         | PC PROPA-GADO | OPERANDO DO RMI  | PC PROPA-GADO | MPC     |                     |            |                                  |
| T0    | 1000H | X                 | X             | X                | X             | X       | X                   | X          | X                                |
| T1    | 1002H | <b>ADD R1, R2</b> | <b>1002H</b>  | X                | X             | X       | X                   | X          | X                                |
| T2    | 1004H | SHR R1, 3         | 1004H         | R1, R2           | <b>1002H</b>  | m_ADD   | <b>R1 ← R1 + R2</b> | X          | X                                |
| T3    | 1006H | JNZ 1AH           | 1006H         | R1, 3            | 1004H         | m_SHR   | R1 ← R1 >> 3        | X          | X                                |
| T4    | 1008H | ADD R1, 2         | 1008H         | 1AH              | 1006H         | m_JNZ   | Z=0: PC ← PC + 1AH  | R1, 3      | R1+R2                            |
| T5    | 100AH | OR R7, R1         | 100AH         | R1, 2            | 1008H         | m_ADDI  | R1 ← R1 + 2         | 1006H, 1AH | R1>>3                            |
| T6    | 100CH | SUB R6, 4         | 100CH         | R7, R1           | 100AH         | m_OR    | R7 ← R7 ∨ R1        | R1, 2      | 1006H+1AH                        |
| T7    | 100EH | ...               | 100EH         | R6, 4            | 100CH         | m_SUBI  | R6 ← R6 - 4         | R7, R8     | R1+2                             |
| T8    | 1020H | NOP               | 1020H         | 0                | 1020H         | m_NOP   | não faz nada        | 0          | nenhum                           |
| T9    | 1022H | MOV R2, [R1+R3]   | 1022H         | 0                | 1020H         | m_NOP   | não faz nada        | 0          | nenhum                           |
| T10   | 1022H | MOV R2, [R1+R3]   | 1022H         | R2, R1, R3       | 1022H         | m_LDR1  | REM ← R1 + R3       | 0          | nenhum                           |
| T11   | 1024H | ADDM [R2], 5      | 1024H         | R2, R1, R3       | 1022H         | m_LDR2  | R2 ← Mw[REM]        | R1, R3     | 0                                |
| T12   | 1024H | ADDM [R2], 5      | 1024H         | R2, 5            | 1024H         | m_ADDM1 | REM ← R2            | X          | R1+R3                            |
| T13   | 1024H | ADDM [R2], 5      | 1024H         | R2, 5            | 1024H         | m_ADDM2 | TEMP ← Mw[REM]      | R2         | X                                |
| T14   | 1024H | ADDM [R2], 5      | 1024H         | R2, 5            | 1024H         | m_ADDM3 | TEMP ← TEMP + 5     | X          | R2                               |
| T15   | 1026H | ...               | 1026H         | R2, 5            | 1024H         | m_ADDM4 | Mw[REM] ← TEMP      | TEMP, 5    | REM                              |
| T16   | 1028H | ...               | ...           | ...              | ...           | ...     | ...                 | TEMP       | TEMP                             |
| T17   | 102AH | ...               | ...           | ...              | ...           | ...     | ...                 | TEMP       | TEMP                             |
| T18   | 102CH | ...               | ...           | ...              | ...           | ...     | ...                 | ...        | Mw[REM]                          |
| T19   | 102EH | ...               | ...           | ...              | ...           | ...     | ...                 | ...        | ...                              |

Tabela 7.13 - Evolução detalhada ao longo do tempo e do espaço das instruções nos estágios do PEPE. A instrução ADD, a negrito, exemplifica a passagem de uma instrução. A zona cinzenta ilustra o esvaziamento de todos os estágios, devido a um salto.

Note-se a paragem nos registos RI e RMI (operandos) durante a execução das microinstruções de mesma instrução.

Note-se que um salto no microcódigo (como na microinstrução `m_SUM7`, Tabela 7.11) esvaziar apenas a cadeia de microinstruções, sem afectar a cadeia de instruções, uma vez que a instrução em execução se mantém.

Durante a execução das microinstruções de uma mesma instrução, a cadeia de instruções está parada e o valor do PC e do REM mantém-se (com exceção do MPC).

No entanto, no caso de um salto condicional no microcódigo, como em `m_SUM2` da Tabela 7.11, a própria cadeia de microinstruções tem de parar até o resultado do teste ser conhecido, o que só acontece no estágio ER (Escrita do Resultado). Os bits que controlam as condições de saída do MPC (`SAO`, `SAZ`, etc.) ligam ao registo RSA, que memoriza a saída da ALU. Desde que se sabe que no MPC está uma microinstrução com um salto condicional até que esta chegue ao estágio ER decorrem dois ciclos de relógio, durante os quais não se pode deixar a cadeia de microinstruções (nem a de instruções) continuar. Caso contrário, se esta fosse uma das últimas microinstruções da instrução, poderiam entrar microinstruções de outras instruções, avançando o PC, RI e RMR e destruindo o contexto, que se deve manter se o salto no microcódigo se der dentro de uma dada instrução (`CASE`). As instruções que contêm ciclos internos) ou se se tratar de um mapeamento condicional que salte para a instrução seguinte, como por exemplo a microinstrução `m_SUM2` da Tabela 7.11, em que nesse caso o PC poderia estar já a apontar não para a instrução seguinte mas para uma posterior.

Nestas condições, o mais seguro é parar o MPC e a cadeia de estágios, quando o sinal `SEL_MICRO_SALTO` indica um salto condicional no microcódigo, durante dois ciclos de relógio, o que é feito pelo bloco "Controlo das cadeias" da Fig. 7.12, através do controlo do sinal de relógio durante dois ciclos de relógio do processador. Este bloco é uma máquina de estados simples, que gera ainda (no segundo e terceiro ciclos de relógio) um sinal que faz entrar zeros na parte de controlo do ROP a seguir à passagem da microinstrução de salto condicional (que serão depois propagados para o RSA). Isto garante que todos os sinais ficam inactivos até a próxima microinstrução fluir novamente, o que equivale na prática, a um esvaziamento controlado da cadeia de microinstruções.

Note-se que os saltos incondicionais no microcódigo (incluindo o mapeamento da próxima instrução) não usam este esquema, pois o salto é feito imediatamente no MPC e o fluxo de microinstruções é contínuo, sem necessidade de esperar por uma decisão.

O sinal `D_OK`, que indica se o acesso à cache de dados teve sucesso, pode também impedir o avanço das cadeias de estágio, tal como é indicado na Fig. 7.12. No entanto, neste caso também o sinal de relógio dos restantes estágios é afectado (todo o processador pára a espera que a cache complete o seu acesso à memória principal). Este detalhe está omitido na figura, por simplicidade.

### 7.3.4 EXCEPÇÕES COM PROCESSAMENTO EM ESTÁGIOS

Para além dos saltos no PC e no MPC, as excepções constituem eventos que interferem com a regularidade do funcionamento em linha das cadeias de estágios e que requerem muitos

cuidados para serem tratados da forma correcta, uma vez que quando ocorrem as cadeias estão normalmente cheias, com várias actividades simultâneas.

O principal desafio é conseguir que um núcleo com processamento em estágios se comporte, do ponto de vista das excepções, como um núcleo sem estágios. Deve guardar o endereço da instrução seguinte àquela em cuja execução a excepção ocorreu, na pilha o endereço da instrução seguinte, e descartando qualquer processamento acabando as actividades das instruções anteriores e descartando qualquer processamento já feito nas instruções seguintes pelos primeiros estágios da cadeia. Deve também guardar o valor do RE correspondente à altura em que a excepção ocorreu.

Sem processamento em estágios, só há uma instrução em processamento de cada vez e portanto é fácil associar uma excepção à instrução em que ocorre. Uma interrupção é atendida no fim do processamento da instrução em que o pedido é feito ao processador. Com processamento em estágios, há normalmente várias instruções em processamento ao mesmo tempo. Destas, qual é a instrução em que se considera que uma dada interrupção ocorreu?

O primeiro aspecto a considerar é que é só no estágio ER que as microinstruções afectam estado do processador (incluindo os bits de estado). Até lá chegarão, as microinstruções em processamento podem ser descartadas sem problemas para efeitos de esvaziamento da cadeia, podendo considerar-se que não tiveram qualquer processamento.

Por outro lado, as excepções podem ser divididas nos seguintes grupos:

1. Excepções de busca, que ocorrem durante a busca de uma instrução. Tal como descrito na página 606, uma excepção deste tipo é inserida na cadeia de estágios na forma de uma instrução SWI que substitui a instrução em que a excepção ocorreu. Isto permite evitar que uma situação de erro causada em antecipação durante a busca de uma instrução cause logo uma excepção, pois essa instrução ainda não está em execução (e uma instrução de salto pode fazer com que esta instrução nem chegue a ser executada). Portanto, estas excepções ocorrem no momento certo, quando o processador precisa de processar uma instrução e não o consegue fazer. No entanto, as instruções anteriores, em fases mais avançadas na cadeia de estágios, têm de acabar o seu processamento normal. As que vinham depois já não podem ser processadas.
2. Excepções de execução, nomeadamente excesso e divisão por zero. Estas situações ocorrem durante o processamento do estágio EM (Execução da Instrução) e devem afectar o estágio seguinte (ER), gerando a excepção e evitando que o resultado erróneo seja escrito no registo resultado (que normalmente é o mesmo que o primeiro operando), o que se consegue ligando o sinal `EXC` (gerado pela unidade de excepções durante apenas um ciclo de relógio) ao sinal de inicialização dos registos interestágio, o que coloca todos os sinais inactivos, tal como num NOP.
3. Excepções de acesso à memória (via cache) de dados. Os acessos à memória são sempre feitos em duas fases; primeiro coloca-se o endereço no REM e só na microinstrução seguinte é que se faz o acesso propriamente dito. Os erros dos

acessos de dados só são detectados após o REM ser escrito, no estágio ER, mas uma eventual escrita na memória (*cache*) não se chega a realizar porque enquanto a exceção é gerada e o estágio de ER (escrita do resultado) da microinstrução seguinte é abortado, tal como no caso anterior.

4. Interrupções. Considera-se que uma interrupção ocorre durante a instrução que estiver nesse instante (quando o pino de interrupção do processador é activado) em processamento no estágio ER, que é o último, garantindo-se assim que a interrupção é atendida na primeira altura possível. Quando esta instrução acabar (o pedido de interrupção ao PEPE pode ocorrer numa microinstrução que não seja a última da instrução), a próxima microinstrução a ser executada é a n\_EXC1, que inicia o tratamento das exceções.

O caso 1 é o mais fácil de tratar. As exceções entram pelo princípio da cadeia, como qualquer instrução (neste caso, SWE), não sendo preciso efectuar qualquer operação especial. A cadeia será esvaziada, mas simplesmente como consequência da escrita no PC do endereço da rotina de tratamento das exceções.

Os casos 2, 3 e 4 acabam por ser idênticos, apesar de a origem ser diferente, e obrigam a esvaziar as duas cadeias de estágiosinal surja o sinal EXC (o que pode acontecer mesmo a meio de uma instrução ou apenas entre instruções, dependendo da exceção em causa), para impedir que as (micro)instruções que já estão em processamento ao longo da cadeia tenham qualquer efeito. Sendo as exceções geradas no último estágio, já não há que esperar que eventuais microinstruções em curso acabem. O sinal EXC é gerado pela unidade de exceções durante apenas um ciclo de relógio e é usado para esvaziar as cadeias. O circuito "Controlo MPC", que implementa a Tabela 7.5, indica que o MPC é inicializado com n\_EXC1 (microinstrução descrita na Tabela 7.11). O número da exceção é indicado pelo sinal NUM\_EXC\_E (Fig. 7.12).

No caso 4, a geração do sinal EXC está dependente do sinal INT\_OK (entrada da unidade de exceções), que deve apenas ficar activa quando se muda de instrução. Neste caso, torna-se como referência o estágio ER, considerando-se que a instrução muda no ciclo de relógio seguinte àquele em que a microinstrução no estágio EM não pertence à mesma instrução que a microinstrução no estágio EM. Para detectar isto, o PC continua a ser propagado ao longo da cadeia de microinstruções. Sempre que os PC propagados nos registos RSA e ROP forem diferentes, há uma instrução que tem a sua última microinstrução no estágio ER e no ciclo de relógio seguinte já terminou. O número da exceção é também indicado pelo sinal NUM\_EXC\_E (Fig. 7.12).

Continuamos com o requisito de guardar na pilha o PC e o RE correctos. Se analisarmos a sequência de microinstruções que invoca a rotina de tratamento da exceção, n\_EXC1 a n\_EXC8 na Tabela 7.11, constata-se que essa sequência é semelhante a uma instrução, em que as microinstruções entram no MPC e a cadeia de microinstruções vai avançando enquanto que a de instruções está parada. O RE que é guardado na pilha é lido directamente do banco de registos, enquanto o PC guardado é o PC propagado no registo RMR (via multiplexer antes do ROP), que se mantém constante durante toda a execução de uma dada

instrução (o PC propriamente dito vai já duas instruções à frente, como consequência do processamento em estágios).

O caso do RE fica automaticamente contemplado. A instrução em que a exceção ocorreu foi a última a passar pelo estágio ER, pelo que nesta altura o valor do RE será o que essa instrução tiver deixado.

O caso do PC é diferente. Quando se considera que a exceção ocorre, o único sítio onde garantidamente está o PC correcto é no PC propagado do estágio ER (registo RSA). Nas exceções do tipo 2 ou 3, a microinstrução que esse estágio está a tratar é aquela que originou a exceção. Nas de tipo 4, essa é a última microinstrução da instrução durante a qual a execução o pedido de interrupção foi feito. O valor desse PC propagado é igual ao endereço dessa instrução na memória mais 2 unidades, o que aliás é válido para qualquer microinstrução em qualquer estágio. Portanto, basta apenas copiar o conteúdo do PC propagado do registo RSA para o do RMI, o que é feito usando o multiplexer antes do PC propagado do RMI e o próprio sinal EXC para o actuar (Fig. 7.12).

Este valor fica agora inalterado até a primeira instrução da rotina de atendimento da exceção entrar no RR, que é quando a cadeia de microinstruções evolui, mas volta logo pouco depois quando o PC propagado do RMI lá chega de novo.

### 7.3.5 DEPENDÊNCIAS DE DADOS

Inevitavelmente, os problemas criados pelo processamento em estágios não se ficam por aqui. Veja-se a Fig. 7.9, na página 596. Na execução linear (Fig. 7.9a), uma instrução só começa a ser processada após todas as microinstruções da anterior terem sido executadas, num processamento em estágios (Fig. 7.9b), a execução das instruções é parcialmente sobreposta, o que faz com que a ordem de execução de algumas microinstruções seja alterada, o que pode ter efeitos desastrosos sobre o funcionamento do programa.

A Tabela 7.13 proporciona exemplos concretos. A instrução ADD entra na cadeia de instruções em T1, mas só em T5 altera o R1 com o valor que produz (coluna mais à direita). A instrução seguinte no programa, SHR, usa (deve usar) o valor de R1 produzido pelo ADD. No entanto, para fazer o deslocamento à direita precisa de ler o R1 primeiro, o que faz no estágio BO (Busca dos Operandos), armazenando-o no registo ROP (Fig. 7.12). O problema é que isto é feito no instante T3 e só em T5 é que a instrução ADD produz o valor que SHR devia usar!

Há aqui uma dependência de dados inerente ao algoritmo, que assume uma execução sequencial das instruções, uma de cada vez, que não está a ser respeitada. Uma instrução precisa de ler um dado que ainda não foi produzido. A responsabilidade é exclusivamente do processamento em estágios, que altera as relações entre instruções. Isolada, cada instrução tem a mesma semântica. O programa também não foi alterado. O problema está em que cadeias de estágios não respeitarem a ordem cronológica natural de execução das operações elementares das instruções, devido a esta diferença entre os modelos de

execução do programa e da cadeia de estágios e que se traduz num conflito de dados (*data hazard*).

A instrução JNZ parece ser independente da anterior, pois não usa o R1, mas não depende do valor do bit de estado Z, que é alterado pela instrução SHR. Felizmente, a instrução JNZ vai fazendo as contas (somando o valor do PC ao valor da constante 1AH) e só no último estágio, quando o novo valor de PC já está pronto na saída da ALU, é que torna a decisão se o escreve no PC ou o descarta. Nesta altura já o SHR actualizou o bit Z, pelo que embora estas duas instruções sejam consecutivas não originam um conflito de dados.

Tal como a Tabela 7.13, a Fig. 7.13 representa a evolução das instruções do Programa 7.1, mas de uma forma mais esquemática e sintética e evidenciando as dependências dados por meio de setas entre o estágio que produz um valor e o estágio que precisa dele. Setas para a frente não constituem problema, pois significa que um valor é produzido antes de ser lido. No entanto, setas verticais e para trás (no tempo) indicam que um dado valor é preciso no mesmo ciclo de relógio em que é produzido ou até antes, respectivamente, situações com que é preciso lidar para não alterar a semântica do programa.

Se JNZ não saltar, a instrução ADD R1, 2 é no seu estágio BO o valor de R1 produzido pela instrução SHR no seu estágio ER, mas como tem a instrução JNZ entre as duas estes dois estágios ocorrem no mesmo ciclo de relógio (seta vertical).

Estes conflitos aparecem quando uma instrução precisa de ler operandos dum registo num estágio anterior àquele em que outra actualizou esse registo. A maior parte das instruções escreve os registos no estágio ER e lê os dois estágios antes, no BO. Note-se que problema ocorre apenas quando se verificam cumulativamente as seguintes situações:

- O registo de resultado de uma instrução é o mesmo que o registo de operando de uma instrução posterior;
- As instruções estão suficientemente próximas. Com dois estágios de separação entre BO e ER, só instruções adjacentes ou com uma entre elas podem originar conflitos. Por exemplo, a instrução MOV também lê o R1, mas está demasiado afastada da última instrução que alterou este registo (SHR ou ADD, dependendo do JNZ).

Obviamente, o modelo de execução do programa tem de ter prioridade e não pode ser alterado por causa do modelo de execução do processamento em estágios, que terá de adaptar. Há diversas formas de resolver este problema, quer em software quer em hardware.

Em software, o compilador adquire mais relevância (pois tem de conhecer bem a arquitetura do processador para poder detectar e resolver os conflitos) e pode adoptar as seguintes estratégias:

- Alterar a ordem de execução de instruções, se estas forem independentes (não se pode alterar o algoritmo), de forma a afastar entre si instruções em que a saída de uma seja a entrada de outra;

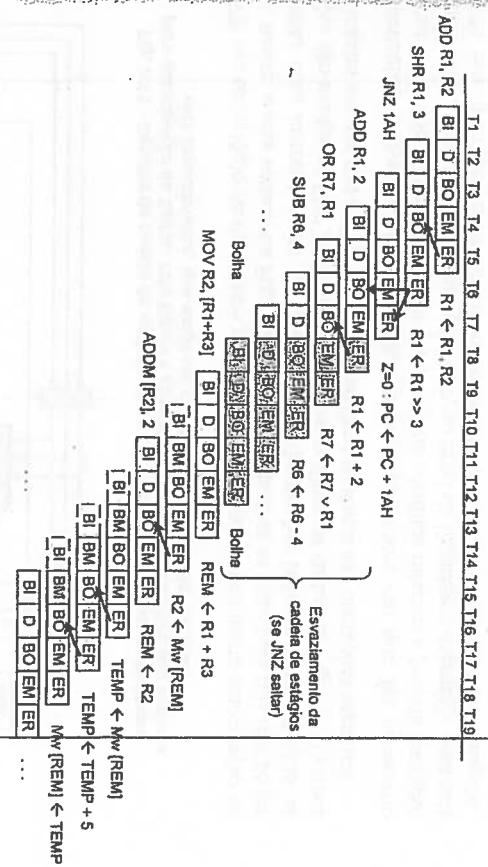


Fig. 7.13 - Evolução das cadeias de estágios com o Programa 7.1 da página 606, evidenciando as dependências de dados. Nas instruções com mais de uma microinstrução, só a primeira tem estágio BI a D, este último substituído pelo BI.

Nas microinstruções seguintes

- Inserir duas instruções NOP (que não afectam nada) entre as instruções que tenham conflito, de modo a que estas tenham duas instruções de intervalo. Esta solução é pior do que a anterior porque reduz o desempenho (perde-se tempo ao executar os NOP), mas se a solução anterior não for possível (se não houver instruções independentes em quantidade suficiente), poderá ser o único recurso;
- As soluções em hardware envolvem projectar o processador de raiz a pensar na cadeia de estágios e incluir circuitos que detectem os conflitos e recuperem automaticamente da situação. As soluções mais usadas são:

- Adopção de um relógio de duas fases não sobrepostas. São dois sinais (em vez de um só), em que ora está um activo ora está outro, com uma pequena fração de tempo em que estão os dois inactivos. As básculas e demais circuitos são projectados para suportar este tipo de relógio. A vantagem deste esquema é que os circuitos podem escrever os registos na primeira metade do ciclo de relógio e lê-los na segunda (leitura-apos-escrita, ou *read-after-write*), o que significa que o caso das setas verticais (dependências no mesmo ciclo de relógio) está automaticamente resolvido. Além disso, o número de instruções NOP a inserir, nos casos em que é necessário, é reduzido de dois para um (as setas das dependências já não têm de ir para a frente, basta serem verticais). Esta técnica é usada pela generalidade dos processadores, incluindo o PEPE,

**NOTA**

A Fig. 7.14a ilustra o diagrama temporal de um relógio de duas fases não sobrepostas ( $\phi_1$  e  $\phi_2$ ). Cada bit de um registo interestágio (Fig. 7.14b) é controlado por dois trincos (seção 2.6.1.2, na página 58), controlados pelas duas fases do relógio. Um registo de 16 bits, por exemplo, é constituído por 16 conjuntos destes, todos ligados às duas fases do relógio.

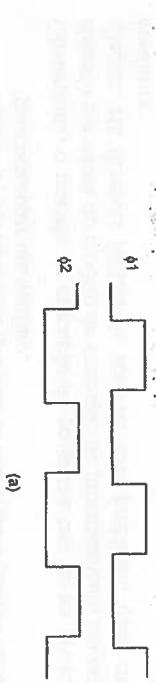


Fig. 7.14 - Uso de um relógio de duas fases, permitindo leitura após escrita no mesmo ciclo de relógio. (a) – Diagrama temporal; (b) – Registo interestágio; (c) – Registo do banco de registos

Durante a primeira fase ( $\phi_1$ ), o trinco da esquerda está aberto (transparente), isto é, a sua saída segue o sinal de entrada. Quando  $\phi_1$  acaba, o trinco fecha e memoriza o último valor da entrada. Durante  $\phi_1$ , a saída do segundo trinco (que estava fechado) não sofre alterações. Durante a segunda fase ( $\phi_2$ ), o segundo trinco abre e o valor antes memorizado aparece na saída do bit do registo. Quando  $\phi_2$  acaba, o segundo trinco memoriza também este valor. O facto de as duas fases do relógio não serem sobrepostas permite a um trinco fechar antes de o seguinte abrir. As pequenas diferenças entre os flancos de  $\phi_1$  e  $\phi_2$  podem ser facilmente conseguidas a partir de um só sinal de relógio usando algumas portas lógicas para produzir atrasos.

Os registos do banco de registos do processador (Fig. 7.3, na página 573), cuja única função é memorizar valores, são trincos simples com uma porta trinco na saída (Fig. 7.14c), que é activada pela segunda fase ( $\phi_2$ ), enquanto a escrita é activada pela primeira fase ( $\phi_1$ ). Desta forma, consegue-se o efeito pretendido, isto é, uma microinstrução ter no estágio B0 o valor escrito (no mesmo ciclo de relógio) no estágio ER por uma microinstrução anterior (assumindo que cada ciclo de relógio comece com o início de  $\phi_1$ ). Isto corresponde à seta vertical na Fig. 7.13.

- Antecipação dos dados (*data forwarding*). O estágio ER pode ainda não ter escrito um dado valor no registo destino, mas se esse valor foi produzido pela ALU já está disponível no registo RSA. Em caso de conflito, basta o estágio B0 ter o valor

do RSA em vez de utilizar o valor do ROP e o problema fica resolvido. Para este efeito, a entrada da ALU possui multiplexers adicionais controlados pelo circuito de detecção de conflitos (Fig. 7.15), que analisa os índices dos registos usados por instruções consecutivas, que para o efeito também têm de ser propagados ao longo da cadeia. Por simplicidade, o PEPE não suporta esta característica. Note-se que não é possível antecipar os dados decorrentes de uma leitura da memória, situação que ocorre duas vezes na Fig. 7.13. Aqui, a única solução é esperar e atrasar a cadeia, com instruções NOP (solução de *software*) ou bolhas de inactividade (solução de *hardware*).

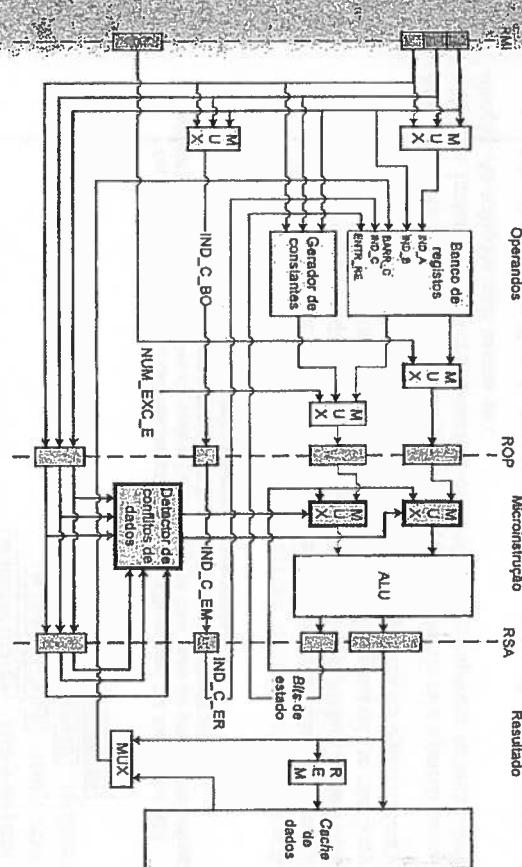


Fig. 7.15 - Detalhe da cadeia de estágios de microinstruções com os *multiplexers* para antecipação de dados. Esta figura é apenas ilustrativa, pois o PEPE não suporta esta característica, que exige um circuito de detecção de conflitos

O PEPE usa um relógio de duas fases mas, por simplicidade, não suporta antecipação de dados. Assim, a seta vertical da Fig. 7.13 fica resolvida, mas as setas para trás têm de ser resolvidas com instruções NOP ou por troca de instruções independentes. Ambas as soluções são usadas na Fig. 7.16, que resolve os conflitos de dados da Fig. 7.13 (a troca de instruções assume que os bits de estado não têm influência nas instruções seguintes).

Em particular, note-se a implementação da instrução ADDM, em que foi necessário introduzir duas microinstruções m NOP para resolver conflitos internos à própria instrução e que não eram resolvíveis com instruções NOP, ao nível da linguagem assembly. Sem esta ilustração, esta instrução funcionaria de forma incorrecta com processamento em estágios.

Note-se também que os atrasos pioram o desempenho do processador face à situação ideal da cadeia (sempre cheia e a executar operações úteis). Mesmo assim, o tempo de execução é muito menor do que uma simples execução sequencial, pelo que a cadeia de estágios é um elemento fundamental em qualquer processador.

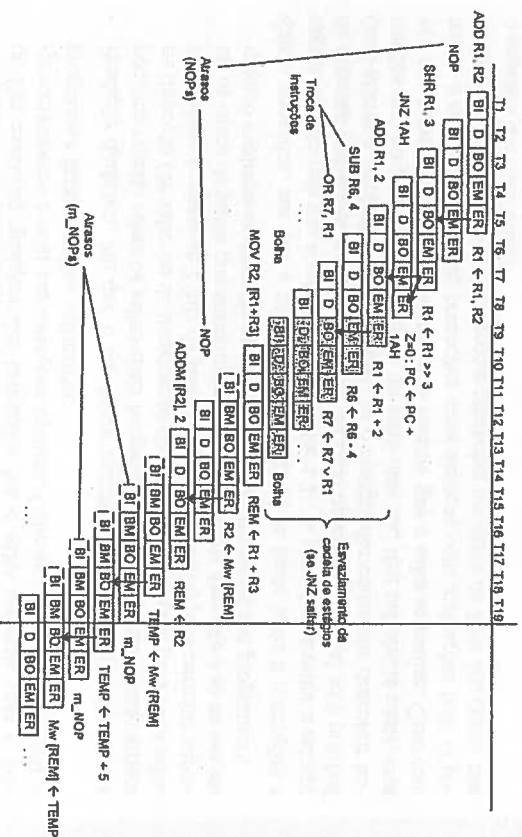


Fig. 7.16 - Resolução dos conflitos de dados da Fig. 7.13 com atrasos e troca de instruções independentes (assumindo que os bits de estado não têm influência nas instruções seguintes)

Na Tabela 7.11, a instrução SUM também tem os seus problemas com o processamento em cadeia de estágios. A microinstrução m\_SUM5 lê o registo TEMP ainda antes de ele ser escrito por m\_SUM4 e (embora de forma menos óbvia) a microinstrução m\_SUM7 (excepto na primeira iteração). A solução mais óbvia seria introduzir microinstruções m\_NOP, mas neste caso é possível eliminar as dependências de dados simplesmente trocando a ordem de m\_SUM5, m\_SUM6 e m\_SUM7 (mas mantendo o salto para m\_SUM2 na última da sequência). A Tabela 7.14 mostra a Tabela 7.11 corrigida em termos de dependência de dados.

### 7.3.6 DEFENDÊNCIAS DE CONTROLO

Além das dependências de dados, também há dependências de controlo, em que a execução ou não de algumas instruções está condicionada por uma (eventual) mudança de fluxo de controlo que acontece nas instruções de salto, chamadas de rotinas e retornos. Quando tal acontece, é preciso evançar toda a cadeia, perdendo todo o esforço efectuado, com as instruções seguintes já em processamento na cadeia de estágios, e esperar que a cadeia encha de novo.

| INSTRUÇÃO | NOME | MICRO-OPEAÇÕES | SEL CONST | SEL OP\_A | SEL OP\_B | REG\_A | REG\_B | REG\_C | REG\_D | REG\_E | REG\_F | REG\_G | REG\_H | REG\_I | REG\_J | REG\_K | REG\_L | REG\_M | REG\_N | REG\_O | REG\_P | REG\_Q | REG\_R | REG\_S | REG\_T | REG\_U | REG\_V | REG\_W | REG\_X | REG\_Y | REG\_Z | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ | REG\_AA | REG\_BB | REG\_CC | REG\_DD | REG\_EE | REG\_FF | REG\_GG | REG\_HH | REG\_II | REG\_JJ | REG\_KK | REG\_LL | REG\_MM | REG\_NN | REG\_OO | REG\_PP | REG\_QQ | REG\_RR | REG\_SS | REG\_TT | REG\_UU | REG\_VV | REG\_WW | REG\_XX | REG\_YY | REG\_ZZ |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Embora por simplicidade o PEPE não suporte qualquer método para tentar reduzir esta penalização, há uma série de técnicas que são normalmente usadas nos processadores de alto desempenho (lista não exaustiva):

- Pré-processamento das instruções de mudança de fluxo (saltos incondicionais, chamadas e retornos), feito tipicamente numa unidade de pré-busca de instruções, que faz busca especulativa (em avanço) das instruções para uma fila que se insere entre a cache e o núcleo do processador e que reconhece os opcodes das instruções de mudança de fluxo, fazendo logo as contas para determinar qual o endereço para onde o controlo irá mudar, continuando aí a pré-busca. Desta forma, a cadeia de estágios recebe já a sequência de instruções correcta e não precisa de ser esvaziada. Note-se que esta técnica não funciona com saltos condicionais nem com instruções de mudança de fluxo com base num registo (JMP R1, por exemplo), pois nestes casos só após executar a instrução anterior se pode calcular o salto (ou mesmo saber se há salto ou não);
- Predição de salto, em que estatisticamente se tenta adivinhar se um salto condicional vai saltar ou não (quando se acerta, evita-se esvaziar a cadeia):
- Predição estática. O processador dispõe de dois conjuntos de instruções condicionais, um que assume que a instrução salta e outro que assume que não salta. Em execução qualquer das instruções faz o salto correcto, mas o pré-processamento evolui em direções opostas. Cabe ao compilador analisar o programa e fazer a melhor estimativa;
- Predição dinâmica, em que o hardware mantém alguma informação (numa pequena tabela) sobre os saltos mais frequentes e o pré-processamento arrisca na direcção (de salto ou de incremento do PC) mais frequente para cada salto. Normalmente usam-se 2 bits de predição para cada salto, para permitir suportar os casos de saltos que normalmente evoluem numa direcção e só de vez em quando evoluem na direcção oposta (caso típico dos ciclos nos programas).
- Saltos atrasados, em que o efeito de um salto só se sente após N instruções a seguir ao salto (em que N raramente é superior a 2). A ideia é aproveitar o esforço de algumas instruções que vêm a seguir ao salto na cadeia, mas tal só é possível (sem alterar a semântica do programa) se o compilador conseguir descobrir instruções que no programa vêm antes do salto mas não têm influência neste, uma vez que estas instruções são executadas mesmo que o salto se realize. Caso contrário, o compilador tem de preencher estes espaços com instruções NOP. O primeiro espaço consegue-se tipicamente preencher em cerca de 50% dos casos, mas o segundo já é mais difícil.

Os processadores têm cadeias com cada vez mais estágios, aumentando a penalização do esvaziamento das cadeias e aumentando cada vez mais a importância de uma boa predição. O pré-processamento dos saltos incondicionais e a predição dinâmica são as técnicas mais populares, em conjugação com optimizações por parte dos compiladores que tentam mudar a ordem de instruções ou mesmo o programa em si (mas sem alterar a semântica do programa) para minimizar o impacte dos saltos.

#### ESSENCIAL

- O processamento em estágios baseia-se na divisão das tarefas a efectuar pelo processador (em operações elementares) e na execução destas em paralelo em unidades especializadas (em estágios de evolução diferentes), aumentando o ritmo de execução das instruções;
- Em termos de hardware, a alteração mais significativa é a introdução de registos entre partes distintas do Caminho de Dados e da Unidade de Controlo para delimitar os estágios (que no seu conjunto formam uma cadeia linear). Em cada ciclo de relojão, o valor dos registos à entrada de cada estágio são processados e escritos nos registos de saída desse estágio;
- Cada microinstrução passa por todos os estágios, tenha ou não operações a efectuar em cada um deles;
- Saltos nas instruções (ou saltos condicionais) nas microinstruções obrigam a esvaziar a cadeia, reconhecendo o seu encerramento e reduzindo o desempenho;
- Outro factor de perda de eficiência é a dependência entre instruções. Como uma instrução começa a sua execução antes de a anterior terminar, poderá precisar de um valor que a instrução anterior ainda não produziu (dependência de dados). Nesses casos, atrasa-se o processamento da cadeia de estágios, introduzindo bofões de inactividade, ou altera-se a ordem das microinstruções de modo a evitar que microinstruções com dependências estejam contíguas.

## 7.4 INTERFACE DE MEMÓRIA

A interface de memória não é detalhada neste livro, porque essencialmente é uma máquina de estados complexa, sendo um assunto mais de projecto de sistemas digitais do que de arquitetura de computadores, pelo menos a um nível introdutório. Neste

<sup>105</sup> A um nível mais avançado, as interfaces de memória dão um contributo muito relevante para o desempenho do processador, permitindo inclusivamente vários acessos em execução simultânea, em diversos estágios de evolução.

contexto, o importante é a funcionalidade deste módulo e a sua interface com as *caches* (secção 7.5). A Fig. 7.1 mostra a interligação da interface de memória aos restantes blocos do PEPE (*caches* e núcleo). As principais funções da interface de memória são:

- Implementar a interface de memória descrita na secção 6.1, incluindo:
  - Suporte para endereçamento de *byte*, com o pino *BA*, e formato *big-endian* (secção 6.1.5, na página 435);
  - Acesso à memória principal (sinais *RD*, *WR* e *WAIT*, ciclos de acesso e temporizações, secção 6.1.6, na página 446).
- Implementar a interface de DMA (que é totalmente transparente para o núcleo do PEPE), de acordo com a secção 6.4.2.3, na página 511), garantindo que nenhum acesso a DMA é autorizado a meio de uma sequência de vários acessos à memória correspondentes a uma actualização da *cache*. Durante uma operação de DMA, o núcleo pode mesmo prosseguir o seu processamento interno, desde que as *caches* não precisem de aceder à memória principal. Se tal acontecer, os sinais *I\_OK* e *D\_OK* fazem o núcleo esperar até que a transferência de DMA termine, tal como descrito no fim da secção 7.2.1.1. Neste contexto, a referência a estes sinais é meramente ilustrativa do que poderá suceder num processador mais elaborado, uma vez que por simplicidade as operações de DMA no PEPE esvaziam ambas as *caches* (secção 7.5.7).
- Arbitrar os acessos à memória principal por parte das *caches* de dados e instruções. A *cache* de dados tem sempre prioridade, pois qualquer falha no acesso à *cache* de dados empata o processador (as instruções já estão em execução).

## 7.5 CACHES

### 7.5.1 PRINCÍPIOS DE FUNCIONAMENTO DAS CACHES

Os registos constituem os elementos de memória (para armazenamento de dados) de acesso mais rápido que os computadores possuem, sendo accedidos num só ciclo de relógio. A sua capacidade, infelizmente, é extremamente reduzida (na ordem das dezenas de registos ou no máximo poucas centenas).

A memória principal, externa ao processador, possui uma capacidade muito mais elevada (um computador pessoal actual tem uma memória com uma capacidade na ordem de  $64\text{GB}$  e um servidor ainda mais), mas em compensação o tempo de acesso é substancialmente superior.<sup>105</sup> Uma vez que, a tecnologia de fabrico de circuitos integrados

fundamentalmente a mesma para os processadores e para as memórias, a lentidão destas face aos processadores deve-se essencialmente à grande capacidade das memórias, que ligam muitos milhões de células no mesmo barramento, aumentando o tempo necessário para cada bit do barramento mudar de valor.

As *caches* (Fig. 7.1) são memórias que se colocam entre o núcleo do processador e a memória principal com o objectivo primordial de tornar os acessos à memória mais rápidos, em média, do ponto de vista do processador. O seu funcionamento baseia-se no princípio da localidade dos programas, que estabelece que os programas não accedem à memória (quer de dados, quer de instruções) de forma aleatória mas antes usam preferencialmente endereços que se situam na proximidade uns dos outros. Este princípio manifesta-se em duas vertentes:

- Localidade temporal – Se um endereço foi accedido recentemente, é muito provável que seja accedido novamente no futuro próximo;
  - Localidade espacial – Se um endereço foi accedido, os endereços que se situam nas proximidades têm mais probabilidades de ser accedidos no futuro do que endereços mais afastados.
- No fundo, o princípio da localidade não é mais do que o reconhecimento de que um programa accede às suas variáveis e instruções, e que:
- Os programas têm geralmente ciclos, que executam instruções sequencialmente e repetem instruções e accedem repetidamente a uma ou mais variáveis ou percorrem sequencialmente estruturas de dados, em vez de a instrução seguinte a executar ou a variável seguinte a acceder ser uma escolhida aleatoriamente dentro de todo o espaço de endereçamento;
  - Tanto as variáveis como as instruções são colocadas pelo compilador de forma contígua em memória (ocupando apenas uma fração do espaço de endereço) e não uniformemente espalhadas por todo o espaço.
- Mesmo dentro de um programa, verifica-se que em cada instante ele está a trabalhar numa dada zona e tende mais a evoluir de zona de trabalho de forma razoavelmente lenta e localizada do que em cada instante mudar aleatoriamente para outra zona (embora naturalmente o comportamento específico varie bastante de programa para programa).

A Fig. 7.17 ilustra o esquema básico de ligação das *caches* ao processador e à memória, em que:

- O processador liga apenas à *cache*, e esta é que liga à memória principal;
- Se a palavra de memória pretendida estiver na *cache* (acesso com sucesso, ou *hit*), a memória principal nem é accedida (pelo menos em leitura, e as leituras são muito mais frequentes que as escritas);
- Se a palavra pretendida não estiver na *cache* (fallha no acesso, ou *miss*), esta faz automaticamente um acesso à memória principal e memoriza essa palavra (que já estará disponível nos próximos acessos), satisfazendo em seguida o acesso pre-

<sup>105</sup> Não é o caso do PEPE, em que se assume que a memória é suficientemente rápida (ou o PEPE é suficientemente lento) para os acessos demorarem um ou dois ciclos de relógio. Mas nos processadores comerciais (um Pentium, por exemplo), o tempo de acesso a uma palavra de memória pode ser várias vezes superior ao tempo de acesso a um registo (secção 6.5.2.3, na página 527).

tendido como se nada se tivesse passado (apenas com o atraso do acesso à memória principal);

A cache tem muito menor capacidade do que a memória principal, mas:

- O seu conteúdo é estatisticamente mais relevante, isto é, tem apenas as palavras de memória cujo acesso é mais provável no futuro próximo (por terem sido as mais acedidas no passado recente);
- São de acesso muito mais rápido do que a memória principal, não apenas por serem mais pequenas mas também por usarem uma tecnologia diferente (são memórias estáticas, enquanto a memória principal usa memória dinâmica, que permite uma maior capacidade mas aumenta os tempos de acesso – secção 6.5.2.3, na página 527).

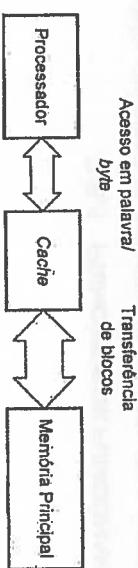


Fig. 7.17 - Esquema básico de ligação das caches

Ter cache é até pior que não ter cache quando a palavra pretendida não está na cache, pois além do tempo de acesso à memória ainda se tem o tempo de memorizar essa palavra na cache. No entanto, é um investimento para os próximos acessos, em que a palavra já estará memorizada na cache e se poupa o acesso à memória.

Verifica-se na prática que a taxa de sucesso (*hit ratio*), ou a percentagem de vezes que a palavra pretendida já está na cache face ao número total de acessos, se situa tipicamente acima dos 95%. Também é usual usar o termo complementar, a taxa de insucesso (*miss ratio*), que estabelece a percentagem média de vezes em que a palavra pretendida não está na cache.

Assumindo este valor para a taxa de sucesso ( $T_s$ ), e que os tempos de acesso à memória principal ( $T_m$ ) e à cache ( $T_c$ ) são 50 ns e 5 ns, respectivamente, o tempo de acesso ( $T_a$ ) que o processador experimenta, em média, é dado por:

$$T_a = T_s \times T_c + (1 - T_s) \times (T_m + T_c) = T_c + (1 - T_s) \times T_m = 5 + 0.05 \times 50 = 7.5 \text{ ns}$$

Ou seja, apenas 15% dos 50 ns que todos os acessos à memória demorariam se não houvesse cache (ou cerca de 6,7 vezes mais rápido). Portanto, e apesar de em alguns acessos demorar mais tempo (55 ns em vez de 50 ns), em média há grande vantagem em usar uma cache.

Na prática, joga-se ainda mais na antecipação e quando o acesso a uma palavra falha acede-se à memória não apenas a essa palavra mas também às suas vizinhas (em endereços adjacentes), num conjunto que se designa bloco, na assunção de que se uma palavra foi acedida é muito provável que as suas vizinhas (companheiras do mesmo bloco) sejam também acedidas no futuro próximo.

A utilização de blocos em vez de palavras individuais tem as seguintes vantagens:

Aumento da taxa de sucesso (*hit ratio*), através da antecipação (jogando no princípio da localidade);

Redução da penalização por insucesso, aproveitando o modo de rajada das memórias dinâmicas (secção 6.5.2.3, na página 527), em que o segundo acesso e seguintes são muito mais rápidos do que o primeiro;

Redução da informação de gestão da cache para uma dada capacidade, pois essa informação é necessária apenas por cada bloco e não para cada palavra individual.

O endereço de base do bloco está sempre alinhado com a dimensão em elementos endereçáveis do bloco. Ou seja, com endereçamento de byte é sempre um múltiplo do número de palavras de um bloco pelo número de bytes de cada palavra.

Por exemplo, se uma cache do PEPF tiver blocos de 4 palavras (de 16 bits), então cada bloco tem 8 bytes e os endereços de base dos blocos têm de terminar em 0 ou em 8. A execução da instrução MOV R1, [R2], em que R2=3C2AH, fará com que todo o bloco com as palavras nos endereços 3C28H, 3C2AH, 3C2CH, e 3C2EH seja carregado na cache. Enquanto este bloco permanecer na cache, qualquer acesso a estas a palavras terá sucesso (*hit*).

Blocos grandes permitem aproveitar melhor a localidade espacial e minimizar os insucessos (*misses*) nos acessos, mas se o seu número for pequeno (depende da capacidade da cache) podem não conseguir cobrir o espectro de acessos do processador e a taxa de insucesso (*miss ratio*) volta a aumentar. Os valores típicos de tamanho de bloco situam-se entre 8 e 128 bytes.

## 7.5.2 ORGANIZAÇÃO DAS CACHES

### 7.5.2.1 PRINCÍPIOS DA ORGANIZAÇÃO

Na memória principal, para aceder a uma palavra basta indicar o seu endereço. Uma cache memoriza apenas algumas palavras (as mais acedidas), cujos endereços não são necessariamente contíguos (normalmente não são, pelo menos entre blocos), pelo que é preciso implementar um mapeamento entre os endereços das palavras na memória principal e a sua localização na cache (naturalmente, apenas para as palavras que lá estão).

Primeiro que tudo, a unidade básica de trabalho da cache é o bloco. Basta aceder a um byte desse bloco para todos os bytes desse bloco terem de estar carregados na cache. Como funcionam todos de forma solidária, a cache só tem de se preocupar com a gestão do endereço de base (o do primeiro byte) de cada bloco, pois sabe que os outros estão colocados de forma contígua, quer na cache quer na memória principal.

Uma cache é essencialmente uma memória com espaço para  $N$  blocos, mas que para além de memorizar cada bloco também tem de memorizar o endereço de base desse bloco, pois

cada posição disponível pode ser ocupada por muitos dos blocos da memória principal, e preciso saber qual está lá em cada momento.

Fazendo uma analogia, é como se a cache fosse um parque de estacionamento com lugares, em que cada lugar pudesse ser ocupado por um dos muitos veículos existentes, e o guarda do parque mantivesse um registo de que veículo está estacionado em qual lugar do parque.

A organização básica de uma cache está representada na Fig. 7.18. Cada posição para um bloco designa-se *linha da cache*, devido à organização gráfica e inclui, para além do bloco propriamente dito, uma etiqueta (que contém o endereço de base do bloco) e um bit de validade, que indica se a linha correspondente tem conteúdo válido ou está vazia. Neste exemplo, a cache tem capacidade para 8 blocos (cada um com 4 palavras de 4 bytes) e a etiqueta são 13 bits, pois os endereços são de 16 bits e cada bloco tem 8 bytes (precisando de 3 bits para identificar cada byte), mas estes números são facilmente extrapoláveis para outras situações.

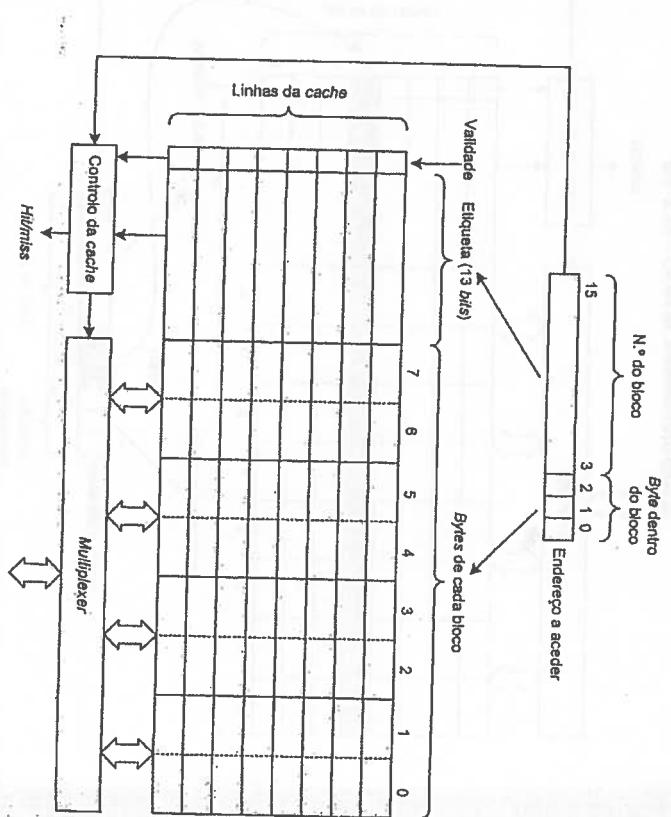


Fig. 7.18 - Organização básica de uma cache

Quando o processador acede a um determinado endereço na memória, está na prática a fazê-lo à cache (Fig. 7.17), que verifica se o byte/palavra (depende do modo de acesso)

acedido está carregado (*hit*) ou não (*miss*). Neste último caso, que tem de ir buscar à memória todo o bloco a que o endereço acedido pertence.

Para saber se o acesso tem sucesso ou não (*hit* ou *miss*), a unidade de controlo interna da cache separa os bits de menor peso do endereço acedido ( neste caso 3 bits) e que só são precisos para efectuar o acesso dentro do bloco, e usa os restantes bits (número do bloco) para procurar o bloco acedido dentro das várias linhas da cache, comparando com o número de cada bloco armazenado na etiqueta de cada linha (ignorando as que têm o bit de validade inactivo).

Se a cache encontrar uma etiqueta igual aos 13 bits de maior peso do endereço a aceder, é só usar os 3 bits de menor peso do endereço para acabar o acesso, dentro do bloco, usando um *multiplexer* para seleccionar a informação pretendida. Se não encontrar, i.e. o bloco da memória principal para uma linha da cache, activa o bit de validade e preenche a etiqueta com o número desse bloco. Dependendo do tipo de cache, a linha a usar pode depender do endereço a aceder ou ser uma das que estiverem livres (se a cache tiver todas as linhas ocupadas, tem de libertar uma e preenche-la com a informação do bloco acedido).

#### NOTA

O acesso visto pelo processador demora mais tempo se houver uma falha (*miss*), mas não tanto mais que se justifique dar outras tarefas ao processador enquanto espera que o acesso se complete. A gestão da cache é feita automaticamente pelo seu circuito de controlo, em hardware e da forma mais rápida possível, pelo que a melhor solução é o processador parar tudo até o controlo da cache indicar que o acesso está pronto.

O mecanismo é semelhante ao já descrito a propósito do sinal WAIT (secção 6.1.6.4, na página 456), e implica acuar mesmo ao nível da unidade de controlo, por exemplo impedindo o relógio de actuar o MPC ou introduzindo estados de espera, normalmente designados bolhas (secção 7.3.1, na página 595).

Os detalhes completos da implementação do PEPE a este nível não são descritos neste livro por já ultrapassarem o nível da arquitectura, sendo já uma questão de projecto de um sistema digital complexo.

A existência da etiqueta é fundamental para a cache conseguir fazer o mapeamento entre um bloco da memória e a linha em que este se encontra armazenado. Há três tipos básicos de mapeamento: directo, associativo e associativo por conjuntos.

#### 7.5.2.2 MAPEAMENTO DIRECTO

O mapeamento directo é a forma mais simples de determinar onde um dado bloco se encontra armazenado na cache, dado o endereço de base desse bloco, tal como ilustrado pela Fig. 7.19. Usa-se simplesmente o resto da divisão do número do bloco (endereço de base sem os bits que identificam o byte dentro do bloco) pelo número de linhas da cache ou, dito de outra forma, os *P* bits de menor peso do número do bloco (no seu conjunto designados índice), em que o número de linhas da cache é  $2^P$ . Neste exemplo, como a cache tem 8 linhas, o índice tem 3 bits. A etiqueta tem de continuar a existir (agora reduzida a 10 bits), pois vários blocos podem estar mapeados na mesma linha da cache. O

índice não tem que ser guardado na etiqueta porque é derivado do endereço a aceder e é sempre o mesmo para uma dada linha da cache.

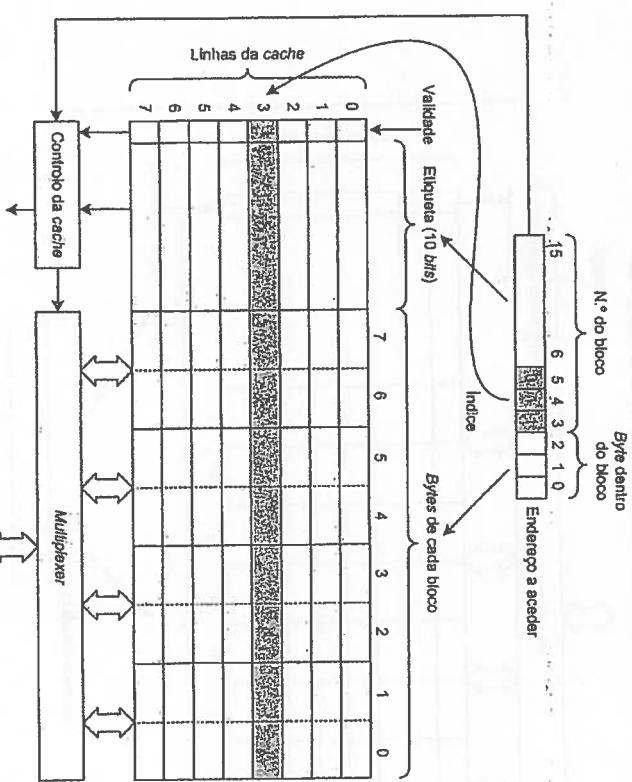


Fig. 7.19 - Cache de mapeamento directo

Para o controlo da cache verificar se um dado bloco está carregado, basta usar os bits de índice retirados do endereço para indexar a cache e obter a linha (para o que se pode usar um simples descodificador 1-de-N – secção 2.5.3, na página 51), verificar se o respectivo bit de validade está activo e comparar a etiqueta dessa linha com o resto do número do bloco (sem os bits do índice). Não é preciso lidar com nenhuma outra linha da cache, pois este bloco só poderia estar nesta linha (que é a que tem o mesmo índice).

No entanto, a simplicidade deste mapeamento é também a sua limitação. Um dado bloco só pode ser armazenado numa linha da cache específica (a indicada pelo índice do endereço), o que significa que todos os blocos cujos números tenham o mesmo índice se mapelam na mesma linha (Fig. 7.20) e simplesmente não podem estar carregados ao mesmo tempo na cache, mesmo que mais nenhuma linha esteja ocupada! Estatisticamente os blocos distribuir-se-ão pelas várias linhas, mas poderá haver linhas não ocupadas enquanto outras serão disputadas por blocos a que o processador está a aceder, reduzindo os benefícios da cache e o desempenho do processador.

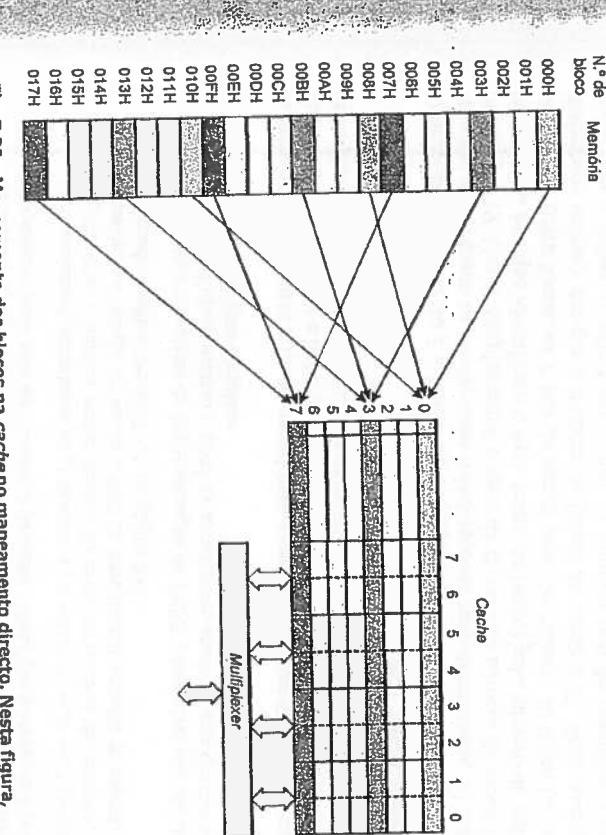


Fig. 7.20 - Mapeamento dos blocos na cache no mapeamento directo. Nesta figura, os elementos de memória representados são blocos e não células individuais

A Tabela 7.15 ilustra o funcionamento de uma cache de mapeamento directo, com 8 blocos de 8 bytes (4 palavras de 16 bits cada). A sequência de acessos representada é apenas um exemplo, pois depende do comportamento do programa. Note-se que:

- Neste caso o número de bloco é igual ao endereço a dividir por 8 (sem os 3 bits de menor peso), tal como indicado na Fig. 7.19;
- Há acessos a endereços diferentes que pertencem ao mesmo bloco (diferem apenas nos 3 bits de menor peso) e há acessos em byte (endereços pares e ímpares) e em palavra (endereços pares). Do ponto de vista da cache, só interessa o bloco a que pertence, pois um bloco ou está na cache (e então todos os seus 8 bytes estão disponíveis) ou não está (é nenhum dos seus 8 bytes lá está);
- Como os acessos em palavra (16 bits) têm de ser alinhados (endereço par), não há risco de um acesso envolver dois blocos (último byte de um bloco e primeiro byte do seguinte);
- Por simplicidade, a Tabela 7.15 representa apenas as etiquetas da cache, mas a cada uma estará associado um bloco de dados (8 bytes, neste exemplo). Nas etiquetas, a tabela representa o número do bloco, para ser mais claro, mas na realidade o que é memorizado é apenas os bits do número do bloco para a esquerda dos bits do índice (Fig. 7.19);

- A cache começa inicialmente vazia mas vai sendo preenchida. A quantidade de falhas (misses) é muito mais elevada do que uma situação normal e estável de um programa já carregado em memória, o que é típico de um programa que começou a sua execução e cujos blocos ainda estão a ser acedidos pela primeira vez.
- Alguns blocos têm de ser substituídos, não por a cache estar cheia mas devido ao mecanismo do mapeamento directo, que mapeia todos os blocos de 8 em 8 na mesma linha da cache, e destes só um pode lá estar em cada instante.

| ACCESSO FEITO |          | NÚMERO DO BLOCO MEMORIZADO EM CADA LINHA DA CACHE |      |     |     |   |   |     |     |
|---------------|----------|---------------------------------------------------|------|-----|-----|---|---|-----|-----|
|               | HT       | 0                                                 | 1    | 2   | 3   | 4 | 5 | 6   | 7   |
| ENDEREÇO      | Nº bloco |                                                   |      |     |     |   |   |     |     |
|               | MISS     |                                                   |      |     |     |   |   |     |     |
| 32H           | Palavra  | 06H                                               | miss |     |     |   |   | 06H |     |
| 6CH           | Palavra  | 0DH                                               | miss |     |     |   |   | 0DH | 06H |
| 35H           | Byte     | 06H                                               | hit  |     |     |   |   | 0DH | 06H |
| 208H          | Palavra  | 41H                                               | miss | 41H |     |   |   | 0DH | 06H |
| 2DCH          | Byte     | 7BH                                               | miss | 41H | 7BH |   |   | 0DH | 06H |
| 20AH          | Palavra  | 41H                                               | hit  | 41H | 7BH |   |   | 0DH | 06H |
| 2B0H          | Palavra  | 56H                                               | miss | 41H | 7BH |   |   | 0DH | 56H |
| 6CH           | Palavra  | 0DH                                               | hit  | 41H | 7BH |   |   | 0DH | 56H |
| 36H           | Byte     | 06H                                               | miss | 41H | 7BH |   |   | 0DH | 56H |
| 14AH          | Palavra  | 29H                                               | miss | 29H | 7BH |   |   | 0DH | 56H |
| 2F0H          | Palavra  | 5EH                                               | miss | 29H | 7BH |   |   | 0DH | 5EH |
| 2B0H          | Byte     | 56H                                               | miss | 29H | 7BH |   |   | 0DH | 56H |
| 6EH           | Palavra  | 0DH                                               | hit  | 29H | 7BH |   |   | 0DH | 56H |
| 2B5H          | Byte     | 56H                                               | hit  | 29H | 7BH |   |   | 0DH | 56H |
| 20AH          | Palavra  | 41H                                               | miss | 41H | 7BH |   |   | 0DH | 56H |
| 32H           | Palavra  | 06H                                               | miss | 41H | 7BH |   |   | 0DH | 56H |
| 184H          | Palavra  | 30H                                               | miss | 30H | 7BH |   |   | 0DH | 06H |
| 218H          | Palavra  | 43H                                               | miss | 30H | 41H |   |   | 43H | 0DH |

Tabela 7.15 - Exemplo de evolução do estado de uma cache de mapeamento directo. Os blocos acedidos estão representados a cinzento e os alterados a negrito. As células em branco representam as linhas da cache vazias (bit de validade inactivo)

O mapeamento associativo tenta resolver este problema, deixando qualquer bloco ser mapeado em qualquer linha da cache. Desta forma, uma linha só precisa de ser substituída por outra se a cache ficar cheia (todas as linhas usadas), independentemente dos números dos blocos a que o processador aceder. As consequências fundamentais são as seguintes:

### 7.5.2.3 Mapeamento associativo

A quantidade de hardware necessário para efectuar todas estas comparações em simultâneo é apreciável, o que na prática limita a capacidade (em linhas) das caches associativas.

A Tabela 7.16 ilustra o funcionamento de uma cache de mapeamento associativo, com 8 blocos de 8 bytes (4 palavras de 16 bits cada). Pode ser contrastada com a Tabela 7.15, para comparação com o funcionamento das caches de mapeamento directo, pois a sequência de endereços acedidos é igual.

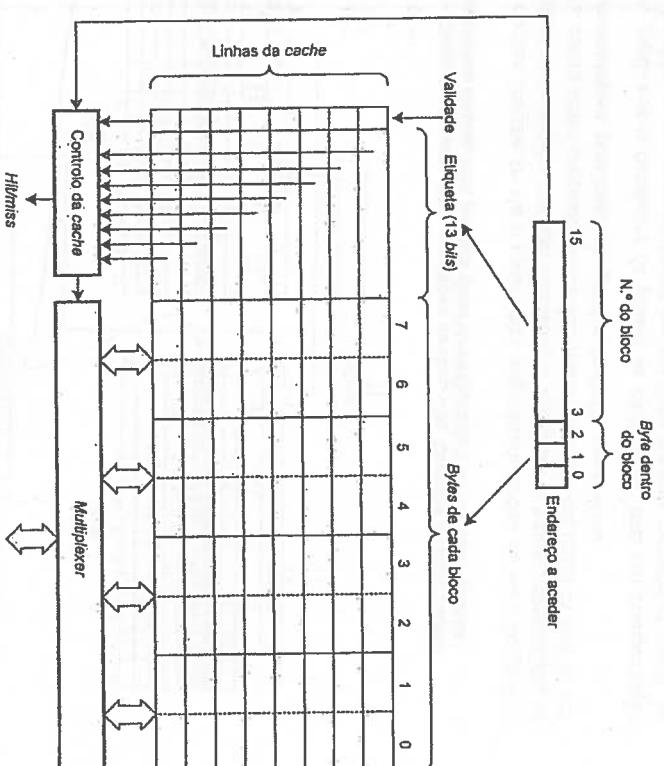


Fig. 7.21 - Estrutura de uma cache com mapeamento associativo

A Fig. 7.21 ilustra este aspecto, com ligações directas de todas as etiquetas ao controlo da cache, em que por cada linha existe um comparador da etiqueta com o número do bloco que o processador quer aceder.

A etiqueta tem de ter todos os bits do número do bloco, pois já não há relação entre parte nenhuma do número do bloco e a sua posição na cache;

O número do bloco a ser acedido tem de ser comparado com todas as etiquetas ao mesmo tempo, pois o acesso tem de ser muito rápido e as procura sequenciais demoram tempo.

| ACCESSO FEITO |         | NÚMERO DO BLOCO MEMORIZADO EM CADA LINHA DA CACHE |      |     |     |     |     |     |     |     |     |
|---------------|---------|---------------------------------------------------|------|-----|-----|-----|-----|-----|-----|-----|-----|
| ENDEREÇO      | Tipo    | N.º do bloco                                      | Hit  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| 32H           | Palavra | 0DH                                               | miss | 06H |
| 6CH           | Palavra | 0DH                                               | miss | 06H | 0DH |
| 35H           | Byte    | 0EH                                               | hit  | 05H | 0DH |
| 208H          | Palavra | 41H                                               | miss | 06H | 0DH | 41H | 0DH | 0DH | 0DH | 0DH | 0DH |
| 2DCH          | Byte    | 7BH                                               | miss | 06H | 0DH | 41H | 7BH | 0DH | 0DH | 0DH | 0DH |
| 20AH          | Palavra | 41H                                               | hit  | 06H | 0DH | 41H | 7BH | 0DH | 0DH | 0DH | 0DH |
| 2B0H          | Palavra | 56H                                               | miss | 06H | 0DH | 41H | 7BH | 56H | 0DH | 0DH | 0DH |
| 6CH           | Palavra | 0DH                                               | hit  | 06H | 0DH | 41H | 7BH | 56H | 0DH | 0DH | 0DH |
| 36H           | Byte    | 06H                                               | hit  | 06H | 0DH | 41H | 7BH | 56H | 0DH | 0DH | 0DH |
| 14AH          | Palavra | 29H                                               | miss | 06H | 0DH | 41H | 7BH | 56H | 29H | 0EH | 0EH |
| 2F0H          | Palavra | 5EH                                               | miss | 06H | 0DH | 41H | 7BH | 56H | 29H | 0EH | 0EH |
| 2B0H          | Byte    | 56H                                               | hit  | 06H | 0DH | 41H | 7BH | 56H | 29H | 0EH | 0EH |
| 6EH           | Palavra | 0DH                                               | hit  | 06H | 0DH | 41H | 7BH | 56H | 29H | 0EH | 0EH |
| 2B5H          | Byte    | 56H                                               | hit  | 06H | 0DH | 41H | 7BH | 56H | 29H | 0EH | 0EH |
| 20AH          | Palavra | 41H                                               | hit  | 06H | 0DH | 41H | 7BH | 56H | 29H | 0EH | 0EH |
| 32H           | Palavra | 06H                                               | hit  | 06H | 0DH | 41H | 7BH | 56H | 29H | 0EH | 0EH |
| 184H          | Palavra | 30H                                               | miss | 06H | 0DH | 41H | 7BH | 56H | 29H | 5EH | 30H |
| 218H          | Palavra | 43H                                               | miss | 43H | 0DH | 41H | 43H | 56H | 29H | 5EH | 30H |

Tabela 7.16 - Exemplo de evolução do estado de uma cache de mapeamento associativo. Os blocos acedidos estão representados a cinzento e os alterados a negrito. As células em branco representam as linhas da cache vazias (bit de validade inactivo)

Note-se que:

- As linhas são tipicamente preenchidas por ordem sequencial e o número de linha não tem relação com o número do bloco;
- Um bloco só sai da cache para ser substituído por outro, depois de a cache encher. É o que aconteceu ao bloco 06H, no último acesso, pois já não havia nenhuma linha livre;
- Na Tabela 7.15 há cinco hits (acesso em que o bloco já está na cache). Na Tabela 7.16, para a mesma sequência de acessos, há nove hits, o que ilustra a vantagem das caches associativas. Na Tabela 7.15, estes quatro hits a menos resultam do facto de alguns blocos que já estavam na cache terem sido substituídos por outros mapeados nas mesmas linhas da cache, apesar de haver três linhas da cache que nunca são usadas, ao contrário da cache associativa, que usa todas as linhas disponíveis e assim aproveita melhor a capacidade da cache. A desvantagem das caches associativas é a quantidade de hardware que necessitam.

### 7.5.24 MAPEAMENTO ASSOCIATIVO POR CONJUNTOS

Uma solução intermédia, mais flexível que o mapeamento directo mas fácil de implementar, é usar  $k$  caches de mapeamento directo e fazer a procura em todas elas em paralelo. Esta solução designa-se mapeamento associativo por conjuntos de  $k$  vias e está ilustrada na Fig. 7.22.

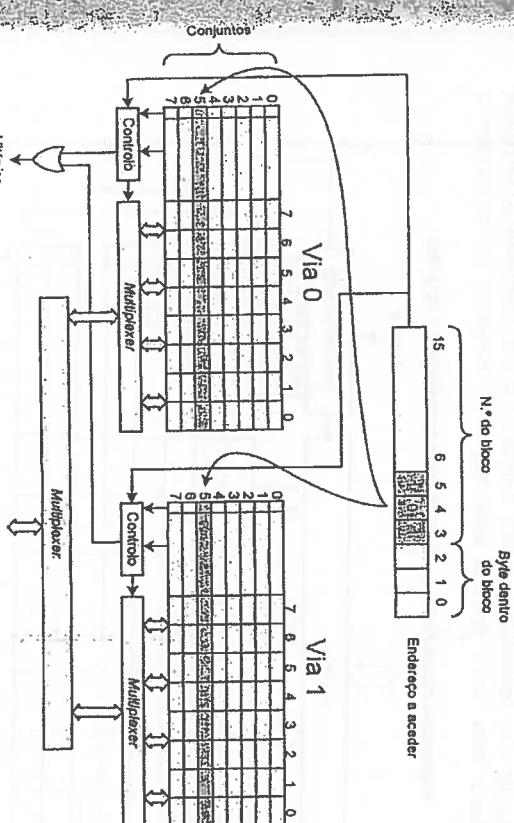


Fig. 7.22 - Cache de mapeamento associativo de duas vias. Consiste de duas caches de mapeamento directo, com procura simultânea (associativa) nas duas caches

Cada uma das caches designa-se via e cada linha, englobando todas as vias, designa-se conjunto. O índice no endereço a aceder identifica o conjunto. A palavra pretendida, se estiver na cache, estará nesse conjunto, numa das vias. A procura em todas as vias de um dado conjunto é simultânea, pois estas são geridas de forma associativa.

Neste exemplo, duplica-se o hardware de gestão de cada linha, mas em compensação para que dois blocos que se mapeiem na mesma linha possam coexistir na cache. Se por exemplo se quiser admitir quatro blocos com o mesmo índice ao mesmo tempo na cache, esta tem de ter quatro vias.

Uma cache de uma só via é uma cache de mapeamento directo. Uma cache com tantas vias como blocos é totalmente associativa. Pelo meio há várias hipóteses de combinação entre mapeamento directo e totalmente associativo, mantendo constante a capacidade total da cache em blocos (usar uma cache com mais vias não implica necessariamente aumentar a capacidade total da cache). A Fig. 7.23 ilustra as variantes possíveis com oito blocos (por simplicidade, apenas as etiquetas estão representadas), desde uma cache de mapeamento directo até uma de oito vias, totalmente associativa.

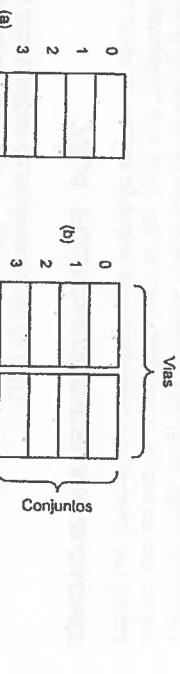


Fig. 7.23 - Variantes da associatividade com oito blocos (apenas estão representadas as etiquetas). (a) – Mapeamento directo (só uma via); (b) – Mapeamento associativo com quatro vias; (c) – Mapeamento associativo com oito vias; (d) – Mapeamento totalmente associativo (um só conjunto)

A Tabela 7.17 ilustra o funcionamento de uma cache de mapeamento associativo de duas vias, com 8 blocos de 8 bytes (4 palavras de 16 bits cada). Pode ser contrastada com a Tabela 7.15 e a Tabela 7.16, para comparação com o funcionamento das caches de mapeamento directo e totalmente associativo, pois a sequência de endereços accedidos é igual.

Note-se que:

- Em cada conjunto, um bloco novo é escrito sequencialmente em cada via (com duas, equivale a alternadamente);

Apesar de a existência de duas vias permitir a dois blocos com o mesmo índice coexistirem na cache, ainda se nota algumas saídas e reentradas de blocos devido a existirem mais de dois blocos com o mesmo índice. Quando um bloco volta a entrar, pode não ir para a mesma via, tal como ilustrado pelos blocos 06H e 0DH.

Com a mesma sequência de acessos que nos casos anteriores, esta cache tem seis hits enquanto a cache de mapeamento directo tem cinco e a cache de mapeamento associativo tem nove, o que reflecte alguma melhoria devido à introdução de mais hardware, sem o desempenho da cache associativa mas também sem todo o hardware correspondente. Também se nota um melhor aproveitamento das linhas da cache em relação à cache de mapeamento directo (só há quatro índices diferentes em vez de oito, pelo que a probabilidade de aceder a um bloco com um dado índice é maior), mas pior do que o da cache totalmente associativa;

Este exemplo é muito simples e totalmente inútil em termos estatísticos, despendendo apenas a ilustrar o funcionamento. Na prática, os benefícios (em termos de taxa de insucesso, ou *miss ratio*) de ter duas vias, face às caches de mapeamento directo, situam-se na ordem dos 25%, enquanto a passagem para quatro

vias contribui com menos de 10% e acima disto ainda menos. Quanto mais vias houver, menos conflitos haverá no posicionamento de blocos na cache, mas também mais complexo o hardware necessário será e cada vez compensará menos. As caches dos processadores comerciais tendem assim a limitar o número de vias a 4, 8 ou 16.

| ACÉSSO FEITO | ENDEREÇO | TIPO    | Nº DO BLOCO | MISS OU | NÚMERO DO BLOCO MEMORIZADO EM CADA LINHA DA CACHE |     |     |     |   |     |     |     |
|--------------|----------|---------|-------------|---------|---------------------------------------------------|-----|-----|-----|---|-----|-----|-----|
|              |          |         |             |         | 0                                                 | 1   | 2   | 3   | 0 | 1   | 2   | 3   |
|              | 32H      | Palavra | 06H         | miss    |                                                   |     | 06H |     |   |     |     |     |
|              | 6CH      | Palavra | 0DH         | miss    |                                                   | 0DH | 06H |     |   |     |     |     |
|              | 35H      | Byte    | 06H         | hit     |                                                   | 0DH | 05H |     |   |     |     |     |
|              | 208H     | Palavra | 41H         | miss    |                                                   | 0DH | 06H |     |   | 41H |     |     |
|              | 2DCH     | Byte    | 7BH         | miss    |                                                   | 0DH | 06H | 7BH |   | 41H |     |     |
|              | 20AH     | Palavra | 41H         | hit     |                                                   | 0DH | 06H | 7BH |   | 41H |     |     |
|              | 2B0H     | Palavra | 56H         | miss    |                                                   | 0DH | 06H | 7BH |   | 41H | 56H |     |
|              | 6CH      | Palavra | 0DH         | hit     |                                                   | 0DH | 06H | 7BH |   | 41H | 56H |     |
|              | 36H      | Byte    | 06H         | hit     |                                                   | 0DH | 06H | 7BH |   | 41H | 56H |     |
|              | 14AH     | Palavra | 29H         | miss    |                                                   | 29H | 06H | 7BH |   | 41H | 56H |     |
|              | 2FH      | Palavra | 5EH         | miss    |                                                   | 29H | 5EH | 7BH |   | 41H | 56H |     |
|              | 2B0H     | Byte    | 56H         | hit     |                                                   | 29H | 5EH | 7BH |   | 41H | 56H |     |
|              | 6EH      | Palavra | 0DH         | miss    |                                                   | 29H | 5EH | 7BH |   | 41H | 56H |     |
|              | 2B5H     | Byte    | 56H         | hit     |                                                   | 29H | 5EH | 7BH |   | 41H | 56H |     |
|              | 20AH     | Palavra | 41H         | miss    |                                                   | 41H | 5EH | 7BH |   | 41H | 56H |     |
|              | 32H      | Palavra | 06H         | miss    |                                                   | 41H | 5EH | 7BH |   | 41H | 56H |     |
|              | 184H     | Palavra | 30H         | miss    | 30H                                               | 41H | 5EH | 7BH |   | 41H | 56H |     |
|              | 218H     | Palavra | 43H         | miss    |                                                   | 41H | 5EH | 7BH |   | 41H | 56H | 43H |

Tabela 7.17 - Exemplo de evolução do estado de uma cache de mapeamento associativo de duas vias. Os blocos accedidos estão representados a cinzento e os alterados a negrito. As células em branco representam as linhas da cache vazias (*bit* de validade inactivo)

### 7.5.3 POLÍTICAS DE SUBSTITUIÇÃO DE BLOCOS

Quando se faz um acesso a um bloco que não está carregado na cache, é provável que todas as linhas em que o novo bloco podia ser carregado estejam já ocupadas. Quando tal acontece, para o novo bloco ser carregado outro tem de sair. Qual deles?

No mapeamento directo só há uma linha possível (a do índice que corresponde ao bloco), pelo que a resposta é óbvia, mas no mapeamento associativo qualquer uma serve e no

mapamento associativo com  $K$  vias há  $K$  hipóteses de escolha, tornando-se necessário definir uma política de substituição de blocos.

Na Tabela 7.16 e na Tabela 7.17, em que há escolha de onde colocar o novo bloco, assumiu-se que os novos blocos eram colocados na via seguinte à do último bloco carregado (encarando a cache totalmente associativa como tendo um só conjunto com  $N$  vias), dando a volta quando chega à última via. É uma forma de ir distribuindo os blocos pela cache, mas não tem em conta o grau de utilização dos blocos, pelo que se pode substituir um bloco muito usado (que provocará previsivelmente um miss logo a seguir, o que pode ser visto na Tabela 7.17 com os blocos 06H e 0DH) em vez de outro raramente acedido (e que portanto não fará tanta falta).

As hipóteses mais comuns da política de substituição são as seguintes:

- **FIFO (First In, First Out)**, ou sequencial – É a utilizada nas tabelas anteriores, pela sua simplicidade. O bloco que está há mais tempo na cache (o primeiro a ter sido carregado) é o primeiro a sair;
- **LRU (Least Recently Used), Usado Menos Recentemente** – O bloco a ser substituído é o que está há mais tempo sem ser usado, o que, atendendo ao princípio da localidade, dá uma medida indirecta do seu grau de utilização e por conseguinte da sua necessidade de estar na cache. Uma implementação rigorosa desta política precisava de um temporizador para cada bloco, o que é incomportável. Por isso adoptam-se aproximações mais simples, que envolvem associar a cada bloco de um conjunto um contador (com  $\log_2 K$  bits, em que  $K$  é o número de vias) que é posto a zero quando um bloco é acedido ou carregado de novo (bloco jovem), enquanto os contadores dos restantes blocos desse conjunto são incrementados (blocos ficam mais velhos). É possível demonstrar que todos os contadores de um mesmo conjunto têm valores diferentes, e quando for necessário substituir um bloco usa-se aquele cujo contador tem o maior valor (o menos recente a ser acedido). Há ainda aproximações mais simples, envolvendo bits individuais que distinguem qual das metades do conjunto é mais velha (foi acedida há mais tempo), dentro de cada metade qual é a submetade mais velha e assim sucessivamente até se distinguir um dos blocos;
- **Aleatório (Random)** – O bloco a ser substituído é um qualquer do conjunto, seleccionado de forma aleatória. Embora pareça uma escolha pouco científica, o facto é que em média pouco pior é que a política LRU e é muito fácil de implementar, em particular com mais de quatro ou oito vias de associatividade, que tipicamente é o limite prático de implementação da LRU.

#### 7.5.4 POLÍTICAS DE ESCRITA NAS CACHES

O que foi dito até aqui sobre as caches aplica-se essencialmente aos acessos em leitura (incluindo buscas de instruções), que aliás são os mais frequentes. A escrita nas caches envolve o desafio adicional de manter a consistência entre a cópia de um bloco na cache e o original na memória principal, havendo duas políticas básicas:

- **Escrita imediata (write-through)** – O valor é sempre escrito imediatamente na memória principal, atravessando a cache. Se o bloco em causa está carregado na cache, actualiza também o valor na cache. Se não estiver, há a hipótese de nem o carregar ou carregar a versão actualizada com o valor escrito. Nesta política nunca há inconsistência entre as versões de um bloco na cache e na memória principal;
- **Escrita atrasada (write-back)** – A escrita é apenas feita na cópia do bloco na cache (esse bloco tem de ser carregado caso lá não esteja) e esse bloco só é actualizado na memória principal quando esse bloco tiver de sair da cache (por o espaço ser preciso para outro bloco, por exemplo).

A política de escrita atrasada é mais complexa de implementar, mas a escrita imediata tem grandes problemas de desempenho, pois cada escrita na memória principal é muito lenta. Uma escrita de uma só palavra na memória principal pode demorar muitos ciclos de relógio, durante os quais o processador terá de ficar bloqueado à espera que o acesso em escrita se complete.

Para evitar isto, usa-se normalmente um **tampão de escrita (write buffer)**, uma pequena zona de memória interna à cache, do tamanho de uma palavra. Depois de o processador escrever a palavra no bloco da cache com escrita imediata e no tampão de escrita, pode imediatamente prosseguir, enquanto a cache se encarrega de autonomamente escrever a palavra do tampão de escrita na memória principal. Uma vez que esta escrita esteja acabada, o tampão de escrita é libertado. É normal este tampão ser antes uma fila de espera, e não uma simples palavra, para a cache não bloquear em eventuais escritas do processador efectuadas logo a seguir à primeira.

Na política de escrita atrasada, todos os acessos em escrita são feitos na cache, muito mais rápido que a memória principal. A escrita de um bloco é só feita uma vez e pode utilizar o modo de rajada das RAMs (secção 6.5.2.3, na página 527) em vez de acessos individuais, o que também é muito mais rápido.

A implementação da política de escrita atrasada assume que, para além do bit de validade, cada linha da cache tem associado um bit de "sujidade" (dirty bit) que é posto a 1 em todas as escritas no bloco carregado nessa linha. Quando um bloco tem de sair da cache, só precisa de actualizar a memória caso o bloco tenha sido alterado, ou seja, o seu dirty bit esteja a 1. Se este bit estiver a 0, basta substituir o bloco por outro, pois a memória contém o bloco original caso seja necessário de novo.

Por outro lado, a política de escrita atrasada tem problemas de consistência entre a memória principal e a cache, no caso de acessos a periféricos, tal como descrito na secção 7.5.6, e ainda no caso de computadores com vários processadores (multiprocessadores) que partilham uma mesma memória (mas com caches independentes). No entanto, a política de escrita imediata não é viável, na prática, com sistemas de memória virtual (secção 7.6), pois o tempo de escrita em blocos que residem em disco é demasiado elevado, pelo que cada vez mais as caches usam sobretudo a política de escrita atrasada ou permitem configuração da política.

### 7.5.5 EVOLUÇÃO DO SUBSISTEMA DE CACHES

As *caches* constituem um nível intermédio de memória entre os registos e a memória principal, num compromisso entre rapidez de acesso e capacidade. A medida que a velocidade dos processadores (expressa pela sua frequência de relógio – Tabela 6.18, na página 521) foi aumentando mais do que a das memórias (expressa pelo seu tempo de acesso – secção 6.5.2.3, na página 527) a capacidade destas foi aumentando substancialmente, a *cache* foi ganhando importância enquanto elemento adaptador entre o ritmo do processador e o das memórias, tendo de cobrir um fosso cada vez maior. Deixou de ser uma pequena memória auxiliar para passar a constituir um subsistema completo, parte integrante e fundamental da estratégia de desenvolvimento de qualquer processador de alto desempenho. A Fig. 7.24 ilustra a evolução deste subsistema ao longo dos tempos visível também na evolução dos processadores da Intel (Tabela 6.18):

- Na época pré-386 não havia suporte do processador para *cache* (Fig. 7.24a);
- Com o 386 apareceu um controlador de *caches* que permitiu implementar uma *cache* externa ao processador (Fig. 7.24b), com memória estática (SRAM, muito rápida), enquanto a memória principal era feita com memória dinâmica (DRAM), mais lenta e com necessidade de refreshamento, mas com maior capacidade);
- O 486 integrhou pela primeira vez a *cache* dentro do processador, permitindo um acesso mais rápido do que numa *cache* exterior e aproximando o seu tempo de acesso do dos registos (Fig. 7.24c);
- No entanto, as limitações de espaço no circuito integrado limitaram a capacidade da *cache*, e rapidamente surgiu a necessidade de utilizar uma *cache* externa, com SRAM, mais lenta que a *cache* interna mas mais rápida do que a memória principal (Fig. 7.24d). Assim, a *cache* interna (L1, nível 1) contém os blocos mais usados dos conteúdos na *cache* externa (L2, nível 2). Cada bloco na *cache* interna tem assim um original na memória principal e duas cópias, uma em cada *cache*; No Pentium, a *cache* interna (L1) foi dividida em duas, uma para instruções, outra para dados (Fig. 7.24e). Isto permite a uma unidade de pré-busca de instruções estar continuamente a fazer busca de instruções em antecipação para uma pequena fila de espera (para quando forem precisas já estarem prontas a ser executadas) sem entrar em grande contenção com os acessos de dados resultantes das instruções em execução. Naturalmente, tem de haver uma unidade interna de arbitrio entre as duas *caches* (não representada por simplicidade) para acesso à *cache* externa (L2), que serve tanto para dados como para instruções;

Graças ao aumento do número de transistores que a tecnologia consegue colocar num circuito integrado, a *cache* L2 passou a estar incorporada dentro do próprio processador no Pentium III, o que para além de aumentar a integração permitiu reduzir os tempos de acesso à *cache* L2 (Fig. 7.24f). Já no Pentium II a *cache* estava no módulo do processador, interligada por um barramento dedicado, mas a funcionar a metade da velocidade em relação à *cache* integrada no próprio processador. A *cache* gasta muitos transistores e tem um significativo impacte no custo, o que motivou o aparecimento de uma versão do Pentium II sem *cache* L2.

(Celeron), mais barata, mas cujo impacte negativo no desempenho forçou a Intel rapidamente a incluir-lhe uma *cache* L2, embora com metade do tamanho da versão mais cara. Isto demonstra a importância da *cache* no processador, tanto no custo como no desempenho;

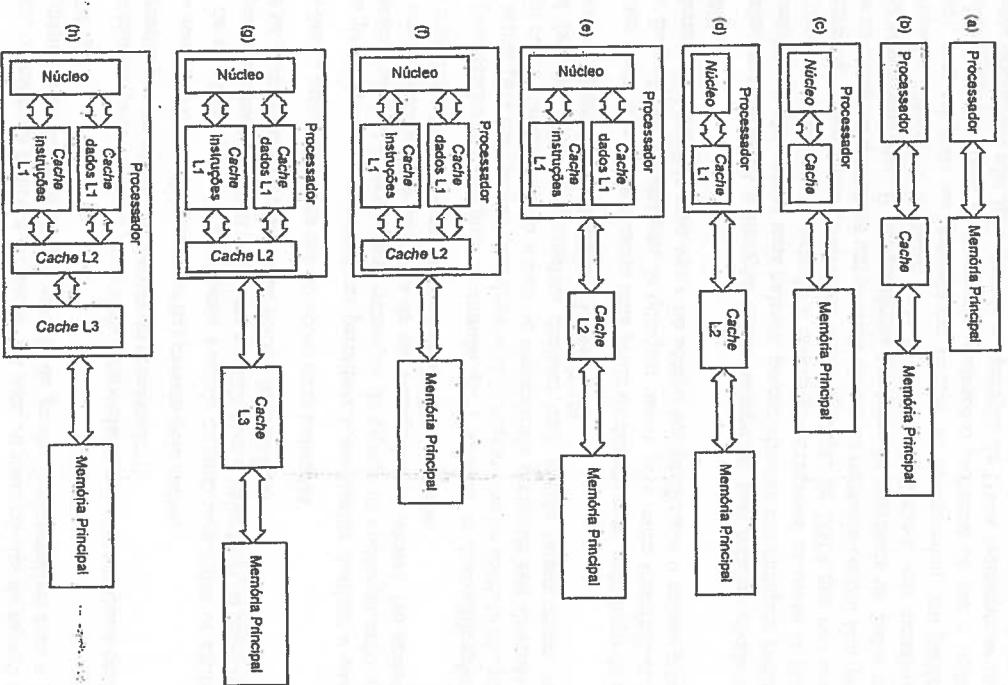


Fig. 7.24 - Evolução do subsistema de caches. (a) - Sem cache; (b) - Cache externa; (c) - Cache interna; (d) - Cache interna (L1) e externa (L2); (e) - Separação da cache L1 em dados e instruções; (f) - Integração da cache L2 no processador; (g) - Cache L3; (h) - Integração da cache L3 no processador

- Com o aumento da complexidade dos programas e da capacidade da memória principal, a cache teve também de aumentar a sua capacidade, o que motivou o aparecimento de uma cache de nível três (L3), externa ao processador (Fig. 7.24g) no Pentium 4. Entretanto, a cache L2, interna, já tinha 1 MByte;
- Os processadores mais recentes acabaram por integrar a cache L3 no próprio processador (Fig. 7.24h), com autênticos monstros em termos de pastilha do circuito integrado e em que as caches ocupam a maior parte da área. Em 2010, os processadores da Intel Itanium Tukwila e o Nehalem-EX, com 1810 e 2300 milhões de transistores, respectivamente, incluem caches L3 de 24 MBytes!
- A tendência actual é a evolução para processadores multi-núcleo, ligados a memórias principais que já há muito ultrapassaram o gigabyte, o que provavelmente motivará o aparecimento de mais níveis de cache.

**NOTA** A cache de um processador é um mecanismo de hardware e não deve ser confundida com as caches também usadas em inúmeros programas, que são mecanismos de software puro. O objectivo é idêntico (guardar uma cópia de alguma informação a que se accede para nos acessos subsequentes ser mais fácil ou mais rápido aceder-lhe), mas a implementação é bem diferente. Por exemplo, um web browser (Internet Explorer, Firefox, Safari, Opera, etc.) guarda tipicamente informação em ficheiros no computador local para que no próximo acesso não tenha de ir buscar a um servidor toda a informação de novo, reduzindo o tempo de carregamento nos acessos subsequentes. No entanto, não há procuras associativas (simultâneas) e para decidir se a cópia local que tem ainda é idêntica ao original deve aceder ao servidor para saber se o ficheiro em causa foi mudado ou não. Algo que é muito mais lento e complexo que um simples acesso à memória, mas mesmo assim compensa porque o carregamento completo do ficheiro pretendido pode demorar ainda muito mais tempo que a simples verificação de validade.

### 7.5.6 CASOS EM QUE NÃO SE QUER CACHE

- As caches são imprescindíveis em qualquer processador moderno de alto desempenho, mas constituem um problema fora dos casos normais de acesso à memória pelo processador (busca de instruções e leitura/escrita de dados), o que sucede essencialmente nas seguintes situações:
- Acesso aos periféricos, quer em leitura quer em escrita. Se o sistema não souber que um dado endereço pertence a um periférico, à primeira leitura dá *miss* e efectivamente faz a leitura do periférico, mas nas leituras seguintes lê o valor em cache em vez de ler novamente do periférico, cujo estado pode já ter mudado. Nos acessos em escrita, este problema não existe se a política de escrita for imediata, pois o periférico é sempre escrito, mas, se a política utilizada for a escrita atrasada, o periférico só é realmente escrito quando o bloco correspondente tiver de sair da cache, perdendo-se todas as escritas intermédias efectuadas na cache;
  - Transferências de dados por DMA (secção 6.4.2.3, na página 511), quer entre memória e periféricos quer entre zonas diferentes de memória, pois há alteração

de blocos da memória principal fora do controlo do processador e que não passam pela cache.

Em todos os casos, o problema básico é o mesmo. Há uma ou mais células do espaço de endereçamento (memória ou periférico) cujo modelo de gestão é incompatível com o da cache. Esta assume que:

- Só o processador pode aceder a essa célula (imediatamente, o estado dessa célula só pode mudar por um acesso em escrita do processador);
- Todos os acessos à célula (leitura e escrita) passam pela cache;
- A cópia dessa célula na cache representa a célula original para todos os efeitos, sendo apenas necessário no fim actualizar a célula com o valor final da cópia dessa célula na cache (não interessando os valores intermédios).

Pelo menos uma destas assunções falha nas situações atrás indicadas. Para acessos aos periféricos, os processadores permitem normalmente desligar a cache quando o processador accede a determinados endereços do espaço de endereçamento, que são designados *non-cacheable* (não passíveis de ser carregados na cache). No acesso a estes endereços, o processador comporta-se como se não tivesse cache.

Típicamente, o processador tem um pino de entrada que o sistema de descodificação de endereços pode actuar para indicar que num determinado acesso o valor accedido não pode ser carregado em cache. Outra solução é usar os mecanismos presentes nos sistemas de memória virtual (secção 7.6), que permitem declarar uma página inteira como *non-cacheable*, através do descriptor dessa página (Tabela 7.19).

Note-se que nestes casos não está em causa uma perda de desempenho pelo facto de não usar a cache. Esta não teria vantagens de qualquer modo, pois numa transferência de dados cada endereço só é accedido uma vez e no acesso aos periféricos o acesso precisa sempre de ser feito.

Para as transferências com DMA a situação é mais complexa, pois está envolvida toda uma zona de memória, e não apenas uma palavra, potencialmente em qualquer gama de endereços. O problema coloca-se em relação a blocos já carregados na cache (e potencialmente já alterados) e que sejam envolvidos numa operação de DMA que não leia os blocos alterados na cache ou que altere um ou mais blocos na memória central sem que as cópias na cache sejam alteradas. É um problema genérico de coerência de dados entre cópias. Uma solução possível é invalidar toda a cache (actualizando em memória os blocos alterados), mas tal pode ser gravoso em termos de desempenho, em particular quando há caches grandes e com vários níveis. Acresce o problema de que o software (mesmo o sistema operativo) não sabe quando a operação de DMA realmente se inicia. Assim, a solução típica é usar um protocolo de manutenção de coerência de caches, normalmente usado em sistemas multiprocessador (com vários processadores) com partilha de memória principal, em que neste ponto de vista o controlador de DMA conta como um processador pelo facto de poder mudar o estado da memória. Este protocolo, designado MESI, está fora do âmbito deste livro, podendo consultar-se [Patterson 2008, Stallings 2006] para mais detalhes.

## 7.5.7 CACHES NO PEPE

O PEPE suporta apenas duas *caches* L1, de instruções e dados, que têm de ser configuradas antes de poderem ser usadas. Por omissão, as *caches* estão desligadas após o reset para que o funcionamento básico do PEPE seja o mais simples possível.

Esta configuração está descrita na secção A.2.2, na página 697, permitindo ainda algum controlo, através da invalidação (desencadeada pelo utilizador) de todas as entradas da *cache*, com cópia dos blocos entretanto alterados para a memória principal, no caso da *cache* de dados e se a sua política de escrita em vigor for escrita atrasada (*write-back*).

O PEPE suporta os dois mecanismos referidos na secção anterior para acesso aos periféricos sem passar pela *cache*. O da memória virtual é descrito na Tabela 7.20. O pino IC (*Ignore Cache*) a 1 durante um acesso à memória implica que a *cache* é ignorada durante esse acesso e é utilizado o ciclo normal de acesso a uma palavra individual. Esse pino deve ser gerado pelo sistema de descodificação de endereços (secção 6.14, página 430) sempre que for feito um acesso a um periférico. Note-se que:

- O PEPE não sabe se está a aceder a um periférico, mas apenas que os acessos à memória/periféricos em que o pino IC estiver activo não passam pela *cache*.

Por este motivo, a política de escrita na *cache* que for utilizada é irrelevante para estes acessos. Mesmo que seja *write-back*, os dados são escritos imediatamente nos periféricos;

Nada impede a utilização deste mecanismo com memória, o que poderá ser útil para algumas aplicações específicas, como por exemplo implementar zonas de dados críticos fisicamente localizados em memória RAM não volátil (com apoio de uma bateria).

A solução implementada pelo PEPE nas transferências de DMA para evitar a incoerência entre as *caches* e a memória principal envolve invalidar automaticamente as duas *caches* (o DMA pode ser usado para carregar código) quando atende um pedido de DMA, tendo o cuidado de antes actualizar na memória principal os blocos que já tinham sido alterados na *cache* de dados, caso esta esteja configurada em política de escrita arrasada (*write-back*). Naturalmente, não é um método aceitável para um processador comercial com vários megabytes de *cache*, mas tem o condão de ser simples.

### **INTRODUÇÃO – CACHES**

Esta simulação toma como base o conteúdo desta secção e das anteriores e exemplifica como as *caches* do PEPE funcionam, em particular no que toca aos seguintes aspectos:

- Interface de utilizador, que permite em qualquer altura, com simulação passo a passo, visualizar o estado das *caches* e de algumas estatísticas da sua utilização;
- Configuração das *caches*, em termos de estrutura e políticas;
- Verificação do funcionamento dos vários tipos de *caches* e políticas;
- Avaliação do impacte no desempenho dos programas da existência das *caches*;
- Integração das *caches* num sistema com periféricos.

### ESSENCIAL

- Uma *cache* é uma memória mais pequena, mas, mais rápida (uma ordem de 10 vezes, tipicamente) do que a memória principal e contém as palavras mais recentemente usadas. Se um valor estiver na *cache*, o processador não precisa de fazer um acesso à memória principal. Estatisticamente, isto melhora substancialmente o tempo médio de acesso.

- As *caches* exploram o facto de os programas exhibirem localidade temporal e espacial e conseguem taxas de sucesso (hit ratio) de quase 100%. Fazem parte de qualquer processador que tenha o desempenho como objectivo.
- Uma *cache* funciona como um dicionário. Dá-se-lhe um endereço e ela permite aceder ao conteúdo respetivo, se o tiver carregado. Se não o tiver, varre todo o núcleo do processador só vez as *caches*. São estas ligadas com a memória principal. Os processadores modernos têm *caches* internas;
- As *caches* podem ser de mapeamento directo, associativo ou misto (associativo por conjuntos). O mapeamento associativo permite aproveitar melhor a capacidade da *cache*, mas é mais difícil de implementar. O mapeamento directo é simples mas tem uma taxa de sucesso pior.
- Num acesso em escrita, a *cache* pode actualizar logo a memória principal (*write-through*), ou mais tarde, com todas as alterações de uma só vez (*write-back*).

## 7.6 MEMÓRIA VIRTUAL

### 7.6.1 HIERARQUIA DE MEMÓRIAS

A capacidade de memorizar valores, a par da capacidade de os processar, é fundamental em qualquer computador, sem o que não seria possível lidar com estruturas de dados complexas.

Do ponto de vista do conjunto de instruções de um processador, existem dois tipos de elementos de memória:

- O banco de registos (sobre os quais a estimadora maioria das instruções opera);
  - A memória propriamente dita (sobre a qual se pode apenas executar duas operações: leitura para um registo e escrita a partir de um registo).
- Os registos são de acesso fácil e rápido mas são poucos, enquanto a memória, em relação aos registos, só serve para ler ou escrever e é de acesso lento mas possui uma grande capacidade.

As *caches* constituem um mecanismo de reduzir o tempo de acesso médio da memória, aproximando-o do dos registos (em termos estatísticos), sem perder a capacidade de armazenamento da memória. É um mecanismo automático e transparente às instruções, embora o programador/compilador o devam conhecer e saber utilizar para optimizar o desempenho dos programas, reduzindo as falhas (*misses*) tanto quanto possível.

No entanto, a memória principal tem ainda uma capacidade muito pequena e é demasiado dispendiosa para guardar toda a informação de que um computador necessita, nomeadamente ficheiros com programas e dados. Desde os primeiros tempos da informática que os discos magnéticos têm suprido esta necessidade, e continuam a fazê-lo devido às evoluções tecnológicas que têm permitido aumentar a sua capacidade de forma assombrosa. Em comparação com a memória principal, são menos dispendiosos (por MByte) e têm mais capacidade, para além de serem não voláteis, mas são dispositivos electromecânicos de acesso muito mais lento.

Como medida contra potenciais falhas e avarias, deve fazer-se periodicamente uma cópia de segurança do conteúdo do disco (*backup*), para o que tipicamente se usam fitas magnéticas, que têm uma grande capacidade (tanto como os discos) mas um custo de meio de armazenamento muito mais baixo. O acesso à informação numa fita magnética é sequencial e demasiado lento para servir de memória de utilização (em acesso aleatório) para o computador. No entanto, estes dispositivos são muito adequados para as operações de cópia de segurança, em que os ficheiros são escritos e/ou lidos em sequência.

Assim, a "memória" de um computador não é apenas uma, mas sim um conjunto de memórias, a vários níveis, formando uma hierarquia em que as características evoluem em sentidos opostos. Num extremo, o reduzido tempo de acesso dos registos obriga a uma baixa capacidade e alto custo, enquanto no outro a alta capacidade e baixo custo só se conseguem com um tempo de acesso muito alto.

A Tabela 7.18 expressa esta visão, comparando as características dos vários níveis e estabelecendo o paralelo com uma biblioteca, como exemplo da vida real em que também existem vários níveis de tempos de acesso, frequência de utilização e de capacidade e custo de armazenamento, tornando o livro como unidade básica de informação (equivalente à palavra do computador).

### 7.6.2 PRINCIPIOS DE FUNCIONAMENTO DA MEMÓRIA VIRTUAL

A memória principal e a memória de massa são muito diferentes em termos de capacidade e de tempos de acesso, havendo um factor de cerca de 100 vezes entre as duas em cada característica, mas infelizmente em direções opostas. A memória principal tem um acesso cerca de 100 vezes mais rápido, mas cerca de 100 vezes menos capacidade.

O mecanismo de memória virtual apareceu para dar a ilusão<sup>107</sup> de a memória principal ser quase tão grande como a memória de massa, com um tempo de acesso não muito pior

(estatisticamente) do que o da memória principal. Para o conseguir, usam-se os mesmos princípios que as *caches*, embora com uma implementação diferente:

| Função                          | Tecnologia              | Ordem de valor       | Equivale a        |
|---------------------------------|-------------------------|----------------------|-------------------|
|                                 | Modo usual              | Tempo de acesso      | Custo/mbyte       |
| Banco de registos               | Registos                | 0,25 ns              | Centenas de bytes |
| <i>Cache</i>                    | Memória estática (SRAM) | 1 ns                 | 1 MByte           |
| Memória principal               | Memória dinâmica (DRAM) | 50 ns <sup>108</sup> | —                 |
| Memória de massa <sup>109</sup> | Disco magnético         | 5 ms                 | 100 GBytes        |
| Memória de arquivo              | Fita magnética          | Segundos             | 100 GBytes        |

Tabela 7.18 - Hierarquia de memórias de um computador; principais características de cada nível e comparação com um exemplo da vida real (biblioteca)

\* Princípio da localidade, quer espacial quer temporal. Um programa não precisa de toda a sua informação (quer dados, quer instruções) ao mesmo tempo. Isto significa que parte desta informação (a mais usada) pode estar em memória principal, enquanto a restante (a menos usada) pode estar em disco;

Função de mapeamento, neste caso considerando que os endereços a que um programa se refere pertencem a um espaço de endereçamento virtual, que tem existência física apenas parcialmente e em que os endereços virtuais se mapeiam em endereços físicos (que constituem o espaço de endereçamento físico a que as RAMs e os periféricos pertencem). A memória física é a memória principal. Este mapeamento designa-se tradução de endereços, porque um endereço virtual tem de ser traduzido para um endereço físico;

Funcionamento automático, libertando o programador dos detalhes das limitações da memória principal (embora deva conhecer o impacte deste mecanismo nos programas se quiser optimizar o desempenho).

No fundo, é considerar a memória principal como uma *cache* (de nível L4) do disco, que realmente é a grande memória (não volátil) do computador. Tudo o resto (memória principal, *caches* L3, L2, e L1, e registos) não passam de memórias auxiliares, que só são válidas enquanto a alimentação do computador estiver ligada e que só servem para tornar os acessos à memória do disco mais rápidos. Dos 5 ms típicos para um acesso a disco,

<sup>107</sup> Tempo para um acesso individual. Para vários acessos seguidos, em rajada, o tempo do segundo acesso e seguintes é mais reduzido (secção 6.5.2.3, na página 527).

<sup>108</sup> Nome típico, devido à alta capacidade de armazenamento de dados (face à memória principal).

consegue passar-se para os 0,25 ns dos registos em quase 100% dos acessos (com o programa já em regime estacionário). É uma melhoria de quase 20 milhões de vezes no mesmo tempo que se mantém o acesso de um programa a potencialmente toda a grande memória de um disco. Vale a pena ter uma hierarquia de memória, tanto mais que, com exceção dos registos, que possuem um conceito específico diferente da memória, os mecanismos são automáticos.

Note-se que, do ponto de vista da memória principal, o mecanismo das *caches* destina-se essencialmente a aumentar o desempenho (reduzindo o tempo de acesso médio) enquanto o mecanismo de memória virtual se destina essencialmente a aumentar a capacidade de memória aparente aos programas (sem piorar significativamente o tempo de acesso médio).

A Fig. 7.25 ilustra o conceito básico de memória virtual, que requer suporte em hardware do processador para ser implementado. Todos os endereços que um programa manda nos registos do processador e inclui nas suas instruções máquina passam a ser considerados endereços virtuais e não físicos, como até aqui. Quando o processador faz um acesso à memória, fá-lo com um endereço virtual, que tem de ser traduzido para (mapeado em) uma localização física, que tanto pode ser na memória principal como no disco. Neste último caso, a falta do valor na memória principal impede o acesso directo, sendo detectada pelo hardware, que gera uma exceção. A rotina correspondente lê do disco o valor com o endereço acedido e carrega-o em memória física, após o que volta a executar a instrução que gerou a exceção, e desta vez o acesso já pode ser feito. O problema é resolvido em software (pela mesma rotina de exceção) porque lidar com o disco é uma operação demasiado complexa para ser feita totalmente em hardware.

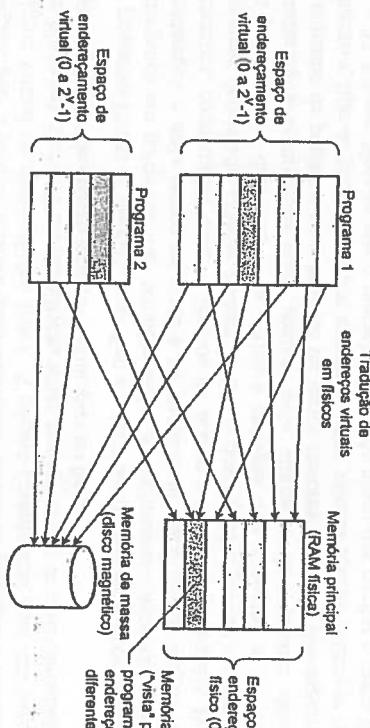


Fig. 7.25 - Conceito de memória virtual

Poderá também suceder que a memória principal esteja toda ocupada e que para um novo valor ser carregado na memória principal outro tenha de sair, tal como já sucedia nas caches, actualizando a versão em disco se o valor tiver sido alterado entretanto.

Sempre que um valor entra ou sai de memória principal, o mecanismo de tradução de endereços tem de ser actualizado. Este mecanismo actua como uma indirecção. Um determinado endereço virtual pode ser mapeado onde se quiser no espaço físico. Aliás, tal como nas *caches* associativas, quando um valor é carregado novamente, pode ir parar a outra localização (embora o mecanismo de implementação seja diferente). Quem decide onde carregar um dado valor e portanto qual a forma de traduzir os endereços é o sistema operativo.

Note-se que nem os programas nem a memória física têm de usar todo o espaço de endereçamento respectivo, que apenas define os limites do endereçamento. Cabe ao mecanismo de tradução de endereços saber que gamas de endereçamento estão a ser usadas, quer a nível virtual quer a nível físico.

As vantagens essenciais da memória virtual são as seguintes:

- O espaço de memória virtual ( $0 \cdot 2^V-1$ ) pode ser muito maior que o espaço de endereçamento físico ( $0 \cdot 2^F-1$ ), embora nada obrigue que seja. O primeiro limita o conjunto de dados/instruções a que o programa pode aceder. O segundo limita a parte deste conjunto que pode estar simultaneamente carregada em memória (que pelo princípio da localidade poderá ser uma parte relativamente pequena). Note-se que o espaço físico pode ser reaproveitado em qualquer altura para outra parte do espaço virtual. Basta alterar a informação para tradução de endereços;

A memória virtual constitui também um excelente suporte para a coexistência de vários programas (processos, secção 7.7) no mesmo computador. Cada programa (processo) tem o seu próprio espaço de endereçamento virtual, ou seja, todos eles podem aceder à mesma gama de endereços virtuais ( $0 \cdot 2^V-1$ ). No entanto, o mecanismo de tradução de endereços pode mapear um dado endereço virtual em cada um dos vários processos em endereços físicos diferentes, separando-os assim totalmente. Esta é uma forma de protecção, pois permite ao sistema operativo isolar completamente os processos uns dos outros. Por outro lado, como os espaços de endereçamento virtual dos processos são independentes, não há necessidade de mudar a gama de endereços (virtuais) que um processo usa devido a outros processos já estarem carregados em memória;

Mas o mesmo mecanismo permite também partilhar memória física ou periféricos entre processos, mesmo em endereços virtuais diferentes. Basta traduzir as gamas de endereços virtuais nos vários processos para uma mesma gama de endereços físicos (Fig. 7.25). Assim, o que um processo escrever os outros podem ler. Este esquema é usado para a comunicação de dados ou outras interacções entre processos (secção 7.7.3.3), sempre de forma controlada pelo sistema operativo através do mecanismo de tradução de endereços.

### 7.6.3 TRADUÇÃO DE ENDEREÇOS VIRTUAIS PARA FÍSICOS

A tradução entre os endereços virtuais e físicos é feita com recurso a uma tabela, que para cada endereço virtual indica qual o endereço físico correspondente. Esta tabela seria impraticável se para cada endereço a traduzir existisse uma entrada na tabela (gastaria

mais espaço que os próprios dados). Felizmente, não há necessidade de fazer uma tradução independente por cada endereço. Basta partir os espaços de endereçamento em zonas de endereços contíguos, chamadas páginas. Desta forma, basta traduzir o endereço de base da página, pois a posição relativa (deslocamento) dentro da página é o mesmo, seja numa página virtual ou numa página física. A página desempenha aqui um papel semelhante ao bloco nas caches. Para simplificar, todas as páginas são de igual dimensão (em bytes), quer no espaço de endereçamento virtual quer no físico.

Quanto mais pequenas forem as páginas, mais fino é o controlo de que endereços estão e não estão carregados em memória, mas também maior é a informação necessária para traduzir os endereços e mais tempo se gasta a ler páginas do disco (cujos tempos de acesso são enormes, comparados com os tempos de acesso à memória principal). Por outro lado, páginas demasiado grandes podem afectar o desempenho porque a simples falta de uma palavra num dado endereço obriga a carregar toda a página a que ele pertence e aumenta a competição entre páginas pela utilização da memória física existente. O tamanho de página a usar depende de vários factores (tamanho dos espaços de endereçamento e tipo de programas, por exemplo), mas valores típicos situam-se na ordem dos 4 a 32 KBytes. Note-se que o tamanho típico de bloco das caches é cerca de 1000 vezes inferior, o que reflecte o facto de o mecanismo de memória virtual se preocupar normalmente com um espaço de memória bastante maior que o das caches.

**NOTA** A paginização (partição do espaço de endereçamento em páginas, todas de igual tamanho) é apenas uma das técnicas usadas em memória virtual. A segmentação permite juntar blocos de tamanho variável, segmentos, cuja principal vantagem é permitir definir características (proteção, detecção de acessos fora do segmento, etc.) de uma forma adaptada ao tamanho de cada segmento que o programa define, em vez de forçar um tamanho fixo. No entanto, este mecanismo não é transparente para a programação e levanta alguns problemas de implementação, pelo que acaba por ser menos geral que a paginização. Também é possível combinar os dois, paginando segmentos grandes (segmentação paginada).

A Fig. 7.26 ilustra o princípio de tradução de um endereço virtual para um físico, assumindo um espaço de endereçamento virtual de 32 bits, um espaço físico de 24 bits e um tamanho de página de 4 KBytes (12 bits). Os dados deste exemplo sugerem que:

- O processador é de 32 bits (porque os endereços que o programa manipula são de 32 bits, e um endereço tem de poder ser armazenado e manipulado como um valor qualquer, logo os registos, ALU, etc., deverão ser de 32 bits);

- O barramento de endereços (físicos) terá apenas 24 bits. Pelos padrões actuais este valor já é pequeno (tal como os 32 bits nos endereços virtuais, alias<sup>110</sup>), mas ilustra o facto de ser comum o espaço de endereçamento virtual ser maior do que o físico, uma vez que a utilização do espaço virtual está limitada pela capacidade

do disco, enquanto a utilização do espaço físico está limitada pela capacidade da memória principal, e esta é sempre bastante menor que a do disco;

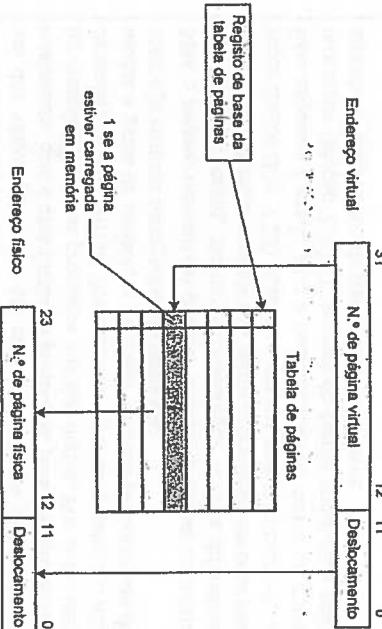


Fig. 7.26 - Tradução de endereços numa memória virtual paginada

Os 12 bits de menor peso dos endereços (quer virtual, quer físico), indicam o deslocamento (posição relativa) de cada endereço dentro da página (que é o mesmo na página virtual e na página física);

- O espaço de endereçamento virtual tem 1 M páginas virtuais ( $2^{20} = 2^{32}/2^{12}$ ), enquanto o físico tem 4 K páginas físicas ( $2^{12} = 2^{24}/2^{12}$ );
- O mecanismo de tradução de endereços tem de mapear 1 M páginas virtuais em 4 K páginas físicas. Isto é, das 1 M páginas virtuais só 4 K de cada vez é que podem estar carregadas em memória principal.

Suponhamos que o processador executa a instrução

MOV R1, [R2]

em que R2 = 3C7A25C4H é o endereço a aceder e tem 32 bits (neste exemplo). É um endereço virtual, sem qualquer correspondência à partida com qualquer endereço físico. Cabe ao mecanismo de tradução de endereços indicar o endereço físico em que o valor a aceder se encontra.

O algoritmo de tradução de endereços pode ser descrito da seguinte forma (Fig. 7.26):

- O endereço virtual é partido em duas partes: número de página virtual (20 bits de maior peso) e deslocamento dentro da página virtual (12 bits de menor peso);
- Existe uma tabela (a tabela de páginas) com uma entrada por cada página virtual (nesta exemplo com  $2^{20} = 1\text{M}$  entradas), com a seguinte informação:
  - Um bit de validade v (tal como nas caches), que indica se a página virtual correspondente está carregada em memória física ou não. Note-se que a tabela de páginas não é uma cache, pois tem de ter todas as entradas possíveis;

<sup>110</sup> A utilização de valores menores que os reais da actualidade nos processadores de alto desempenho destina-se apenas a tornar os exemplos mais manejáveis e compreensíveis.

- Um campo que indica o número de página física em que essa página virtual está carregada (se  $v=1$ ) ou uma referência que permite localizar a página em disco (se  $v=0$ ).

O processador contém um registo que mantém o endereço de base da tabela de páginas. Cada processo (secção 7.7) tem a sua própria tabela de páginas, pois todos os processos podem aceder à mesma gama de endereços virtuais e os espaços de endereçamento acessíveis aos processos são independentes. O registo da base da tabela de páginas faz parte do estado de cada processo e o seu valor muda quando o processo em execução muda;

O número de página virtual ( $20\ bits$ ) é usado para indexar a tabela e ter acesso à informação constante da entrada correspondente a essa página virtual;

Se  $v=1$ , a página virtual está carregada em memória e o número de página virtual é convertido no número de página física ( $12\ bits$ ) indicado nessa entrada da tabela. O número de página física obtido é concatenado ao deslocamento dentro da página para obter o endereço físico final;

Se  $v=0$ , a página virtual não está carregada em memória, o que se designa falta de página (*page fault*). É gerada uma exceção, cuja rotina deve usar a referência para o disco contida na entrada da tabela para ir ao disco ler a página virtual em falta e lê-la para a memória física, após o que altera por bit  $v$  para 1 e coloca nessa entrada o número de página física usada. Quando a rotina de exceção regressa, a instrução que provocou a falta de página (o *MOV*, neste exemplo) deve ser re-exectuada, já não dando origem a uma falta de página desta vez.

Neste exemplo, o número de página virtual é  $3C7A2H$  (os  $20\ bits$  – ou 5 dígitos hexadecimais – de maior peso de  $R2$ ) e o deslocamento dentro da página é  $5C4H$  (os  $12\ bits$  de menor peso de  $R2$ ). Se o número de página física em que está carregada em memória for  $42DH$ , então o endereço físico accedido pela instrução *MOV* e que corresponde ao endereço virtual indicado pelo valor de  $R2 \& 42D5C4H$ .

Note-se que o número máximo de entradas da tabela de páginas com o bit  $v=1$  em qualquer instante é igual ao número de páginas físicas que a capacidade da memória principal suporta e não necessariamente igual ao total de páginas de todo o espaço físico. As faltas de página podem ocorrer na busca de instruções ou durante acessos a dados, quer em leitura quer em escrita. Todas as instruções que de alguma forma accedem à memória podem originar uma falta de página, o que as obriga a ser re-excutáveis. Isto quer dizer que:

- Antes de o último acesso à memória estar completo (algumas instruções fazem mais do que um acesso) não devem alterar o estado do processador, para que a

sua execução possa ser abortada sem ter de estar a desfazer alterações, o que seria muito complicado;

O endereço de retorno da rotina de exceção deve ser não a instrução seguinte mas sim a própria instrução em que a falta de página ocorreu. Ou o mecanismo de exceções faz a diferença e guarda logo o endereço de retorno correcto, consequente às exceções, ou a própria rotina de exceção se encarrega de ajustar o endereço de retorno guardado na pilha. Esta última hipótese demora um pouco mais tempo (mas que é irrelevante face ao tempo de leitura de uma página de disco) e é mais simples em termos de hardware.

## 7.6.4 GESTÃO DAS PÁGINAS

As páginas virtuais não carregadas em memória não estão propriamente espalhadas pelo disco todo, pois isso dificultaria a gestão do carregamento das páginas. Quando um processo é criado, o sistema operativo cria as páginas virtuais que ele necessita num espaço próprio, designado *swap file*.<sup>112</sup> Quando uma destas páginas é referenciada pelo processo respetivo é carregada em memória principal, e se tiver de sair de memória principal volta para este espaço. É esta troca de foco da página, entre a memória principal e esta zona de disco, que está na origem do termo *swap* (trocá). Na tabela de páginas, as referências para o disco que as entradas com  $v=0$  possuem indicam o número de página neste espaço.

**NOTA** Algunhas páginas com informação proveniente de ficheiros estão mapeadas nos próprios ficheiros. Por exemplo, o código dos programas é copiado para memória a partir de um ficheiro e não precisa de ser alterado, pelo que não tem sentido utilizar uma página adicional no *swap file* para manter em disco algo que já lá está, embora noutro local.

Tal como nas caches, há o problema de ter de decidir qual a página que tem de sair quando a memória física está toda ocupada e um programa faz referência a uma página virtual que não está carregada em memória. Normalmente, usa-se uma política de substituição de páginas do tipo LRU (Least Recently Used), na assunção de que a página com utilização menos recente será também a menos provável de ser preciso a seguir. Normalmente a informação usada para este fim resume-se a um bit de utilização (*use bit*, ou *reference bit*) em cada entrada da tabela de páginas, que periodicamente o sistema operativo desactiva em todas as entradas e o hardware activa (numa dada entrada) de cada vez que há um acesso à página respectiva).

Também como nas caches, a página que sai tem de substituir a cópia no *swap file* se tiver sido alterada desde que foi carregada na memória principal. Para este efeito, cada entrada da tabela de páginas tem um bit (*dirty bit*) que é desactivado quando a página é carregada e é activado automaticamente pelo hardware quando a página é accedida em escrita. A política de escrita no *swap file* é sempre escrita atrasada (*write-back*), pois o tempo de

<sup>111</sup> Não apenas com as instruções de transferência de dados (*MOV*, por exemplo) mas também com as instruções que manipulam a pilha implicitamente (*CALI*, *RET*, *PUSH*, *POP*, etc.).

<sup>112</sup> Dependendo do sistema, poderá ter outras designações, como *swap space* ou *page file*.

acesso em escrita de uma palavra individual no disco é incompatível. A escrita de uma página inteira autoriza o acesso pelas várias palavras alteradas nessa página, além de que os vários valores escritos no mesmo endereço só o último tem de ser copiado para disco. Por outro lado, havendo uma entrada na tabela de páginas por cada página virtual, é possível incluir nessa entrada informação adicional que faz com que também possa actuar como descritor das propriedades dessa página. Assim, cada entrada da tabela de páginas pode conter a informação indicada na Tabela 7.19. Nem todas as combinações de bits são válidas. Por exemplo, se  $V=0$ , os bits A e D não são relevantes.

| Campo              | Significado | Função                                                                                                                                                                                                                                                                 |
|--------------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Válida             | V           | Se $V=1$ , a página está carregada em memória principal, caso contrário está no swap file                                                                                                                                                                              |
| Acedida            | A           | Se $A=1$ , a página foi acedida desde a última vez que o sistema operativo colocou os bits A das várias entradas a 0. Este bit é usado para a política LRU                                                                                                             |
| Alterada           | D           | Se $D=1$ , pelo menos uma palavra da página foi alterada desde que foi carregada em memória principal. Se a página tiver de sair de memória principal, tem de ser escrita em disco para actualizar a cópia no swap file                                                |
| Alterável          | W           | Se $W=0$ , nenhuma palavra da página pode ser alterada. Se o programa o tentar, é tipicamente gerada uma exceção. É uma protecção usada, por exemplo, para páginas de código (instruções)                                                                              |
| Nível de protecção | P           | Se $P=0$ , a página é de nível de sistema. Se $P=1$ , é de nível de utilizador (secção 7.7.5, na página 682). Esta protecção pode impedir a um programa de utilizador de aceder (mesmo em leitura) a páginas do sistema (geradas por uma exceção em caso de tentativa) |
| Residente          | R           | Se $R=1$ , a página não pode participar do mecanismo de substituição de páginas, ficando residente na memória principal. Esta característica é reservada para páginas importantes e que têm de estar sempre acessíveis                                                 |
| Cacheable          | C           | Se $C=0$ , os acessos à página não passam pela cache, sendo sempre lidos ou escritos no endereço especificado. $C=1$ é usado tipicamente em páginas localizadas nos endereços dos periféricos (secção 7.5.6)                                                           |
| Referência         | Ref         | Número de página física (se $V=1$ ) ou número de página no swap file (se $V=0$ )                                                                                                                                                                                       |

Tabela 7.19 - Informação típica contida numa entrada da tabela de páginas

Cada entrada na tabela gasta tipicamente não mais que uma palavra, e mesmo assim o espaço gasto pela tabela já é muito grande. No exemplo da secção anterior, a tabela de páginas tem  $1M(2^{22}/2^3)$  entradas de uma palavra cada, ou  $4MB$  ( $2^{22}$ ), uma vez que cada palavra tem 32 bits. A tabela ocupa assim 1024 páginas ( $2^{10}/2^3$ ) de memória física, que seriam muito mais bem aproveitadas em páginas de dados ou de instruções.

Há várias técnicas para reduzir o espaço gasto com a tabela de páginas, mas a mais simples consiste em dividir a própria tabela em páginas (de entradas) e inclui-las no mecanismo de swapping. Isto quer dizer que apenas a parte da tabela mais usada precisa de estar carregada em memória principal e não a tabela toda. Para este efeito, tem de haver uma tabela com as entradas correspondentes às páginas da tabela, designado diretório, que tem de estar sempre residente em memória (mas que neste exemplo ocupa apenas 4 KBytes, ou a dimensão de uma página).

A Fig. 7.27 ilustra o conceito. O número de página virtual (20 bits) é dividido em dois conjuntos de 10 bits, uma vez que cada página de 4 KBytes suporta 1 K entradas de 32 bits cada. Os 10 bits da esquerda indexam o diretório e os outros 10 bits indexam a página de entradas que contém a referência para a página virtual acedida.

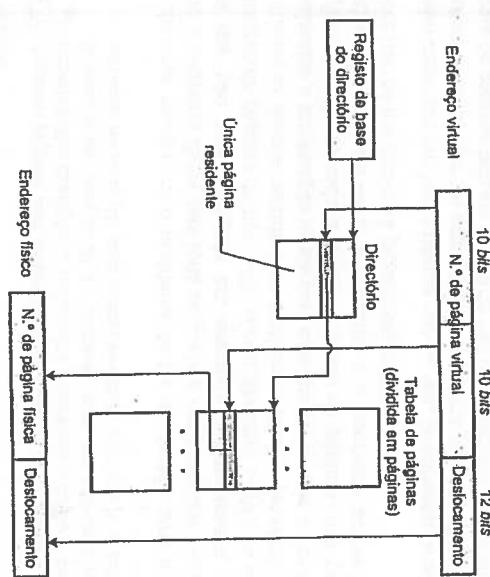


Fig. 7.27 - Tradução de endereços com tabela de páginas multível paginada

A tabela de páginas fica assim multível, em que apenas o diretório está residente. Tudo o resto pode ir para disco se não for acedido. Se uma página de dados ou instruções não for acedida durante algum tempo será remetida para o swap file quando o espaço for necessário para outras páginas mais activas. O mesmo pode acontecer a uma página de entradas (parte da tabela de páginas) se nenhuma das páginas de dados ou instruções que elle referencia for acedida durante algum tempo. O diretório também é uma página, mas residente, pelo que se garante que o mecanismo de tradução de endereços tem sempre o seu ponto de partida carregado em memória.

O diretório só precisa de estar carregado em memória quando o processo está em execução. Há, no entanto, situações em que o processo tem de parar à espera que um evento ocorra (que o utilizador carregue numa tecla, por exemplo) ou que um determinado recurso esteja disponível (secção 7.7.2). Se essa paragem demorar muito tempo, o sistema operativo acaba por passar até o diretório desse processo para o swap file, para aproveitar o espaço dessa página para os processos que estão em execução. Quando tal acontece, diz-se que o processo foi swapped out, isto é, foi totalmente relegado para o swap file. Quando o processo for reactivado, o seu diretório volta a ser carregado na memória principal e só fica residente até o processo ser terminado ou swapped out de novo. Este mecanismo é desencadeado apenas pela paragem prolongada do processo e é diferente do mecanismo normal de swapping (troca de páginas entre o swap file e a memória) das páginas não residentes.

As faltas de página podem agora suceder não apenas no acesso à página de dados ou instruções, mas também durante a própria tradução de endereços, ao tentar aceder à tabela de páginas. É mais uma sobrecarga para o desempenho de todo o mecanismo, mas com efeito relativamente raro (dado o princípio da localidade dos programas) não é estatisticamente muito relevante. Por outro lado, conduz a ganhos significativos em espaço de memória, o que por sua vez contribui para melhorar o desempenho global por ficar mais espaço de memória física livre e reduzir o número de vezes em que uma página tem de ir para disco por a memória estar toda ocupada. Além disso, a maioria dos processos usa apenas uma pequena parte do seu espaço de endereçamento, pelo que muitas das páginas da tabela nunca chegam sequer a ser criadas (as páginas da tabela vão sendo criadas à medida que o seu espaço de endereçamento vai sendo utilizado e as suas entradas vão sendo precisas).

A Fig. 7.28 ilustra a utilização deste mecanismo para aceder a um dado endereço físico. Note-se que, para estabelecer o caminho pela tabela de páginas, a própria página da tabela também teve de ser carregada em memória física e acedida através do directorio, numa operação de recuperação da falta de página executada pelo sistema operativo e que faz com que a página física da palavra acedida pelo programa seja carregada em memória física, para finalmente o acesso se realizar. Nenhuma página pode ser acedida sem estar carregada em memória física.

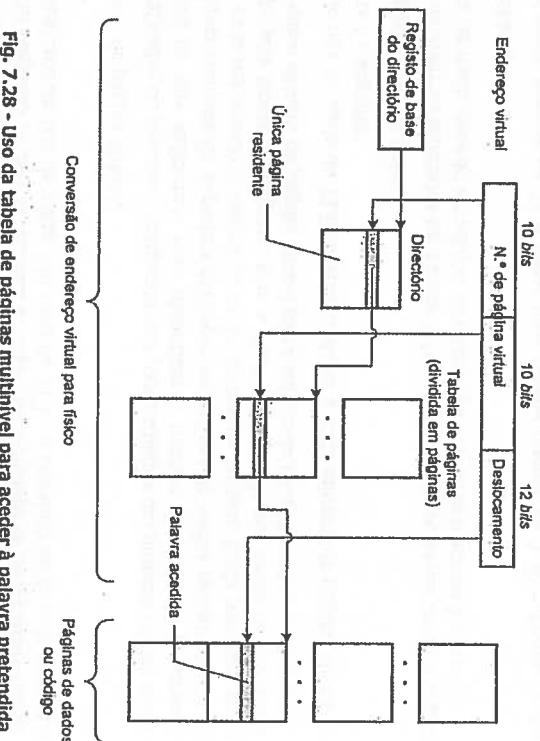


Fig. 7.28 - Uso da tabela de páginas multihilvel para aceder à palavra pretendida

## 7.6.5 ATLB E O SEU PAPEL NA TRADUÇÃO DE ENDEREÇOS

Na Fig. 7.26 já é aparente que para fazer a tradução de endereço virtual para físico é preciso fazer um acesso à tabela de páginas. No caso da Fig. 7.27 é preciso fazer dois acessos, um ao directorio e outro a uma página da tabela de páginas. A tabela de páginas

encontra-se em memória, o que significa que para fazer um acesso à memória, usando um endereço virtual, é preciso fazer um acesso adicional (ou mais dois). Esta é uma situação demasiado gravosa, pois aumenta o tempo de acesso à memória aproximadamente para o dobro ou mesmo para o triplo.

A solução é recorrer mais uma vez ao princípio da localidade e incluir no sistema de memória virtual uma pequena *cache* que contenha os mapeamentos mais usados de número de página virtual para número de página física. Esta *cache* tem a designação habitual de TLB (*Translation Lookaside Buffer*)<sup>113</sup> e a sua integração na tradução de endereços está representada na Fig. 7.29.

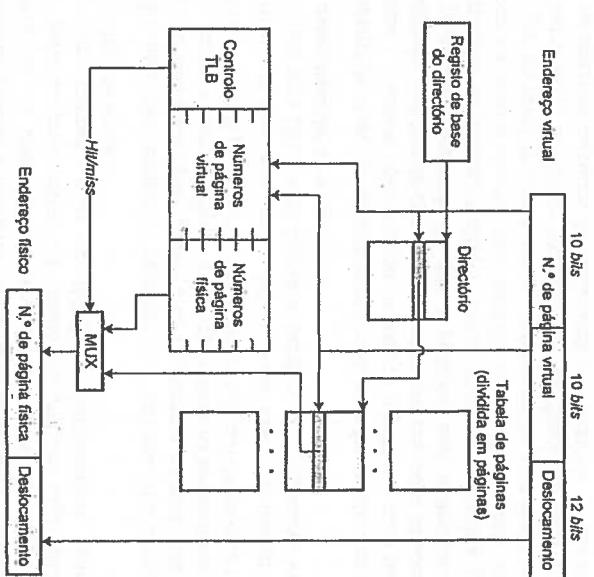


Fig. 7.29 - Integração da TLB na tradução de endereços. Se o número de página virtual constar da TLB, o acesso ao número de página físico é imediato, senão tem de se percorrer a tabela de páginas

Desde que o mapeamento esteja presente na TLB (devido a um acesso anterior), a conversão de endereço virtual para físico consegue fazer-se num tempo muito reduzido e sem acessos à memória. Caso não esteja (TLB miss), é preciso atravessar a tabela de páginas (acedendo à memória) até obter o número de página física, altura em que se

<sup>113</sup> Nome pouco óbvio hoje em dia mas que perdurou desde os primeiros tempos em que apareceu. Actualmente seria mais lógico algo como TLC (*Translation Lookaside Cache*).

memoriza esse mapeamento na TLB. O próximo acesso a um endereço desta página já será muito mais rápido. Ao contrário das outras caches, a TLB memoriza mapeamentos de números de página virtual-física e não dados ou instruções. No entanto, tal como as outras, é um dicionário que recebe um conjunto de bits de entrada e, caso esse conjunto lá esteja memorizado, fornece um conjunto de bits de saída. No caso da TLB, o conjunto de bits de entrada é formado por:

- Número de página virtual;
- Identificador do processo, designado PID (normalmente um número inteiro com o número de bits suficientes para identificar o número máximo de processos). Qualquer computador suporta a execução simultânea de vários processos (secção 7.7), cada um usando a mesma gama de endereços virtuais. Cada processo tem a sua própria tabela de páginas, mas a TLB é partilhada por todos os processos, pelo que o número de página virtual não é suficiente para identificar uma página. O conjunto de bits de saída da TLB, correspondente a cada número de página virtual de cada processo, é o seguinte:
  - Número de página física;
  - Bits de controlo indicados na Tabela 7.19, para que a informação sobre a página possa ser usada quando a tradução de endereços é feita, sem aceder à memória.
- É ainda de notar que:
  - Uma única entrada na TLB serve uma página inteira, ou 1 K palavras, o que significa que a TLB pode ser uma cache relativamente pequena, na ordem nas dezenas a centenas de entradas, e atingir taxas de falha (*miss ratios*) tipicamente inferiores a 1%;
  - Como qualquer outra cache, a TLB também precisa de uma política de substituição de entradas, normalmente aleatório ou uma aproximação de LRU, e de escrever, pois alguns dos bits indicados na Tabela 7.19 podem ser alterados no decurso dos acessos à memória e é preciso actualizar a tabela de páginas. Normalmente usa-se escrita imediata (*write-through*) por ser mais simples e porque o sistema de memória virtual está integrado com as caches de dados e instruções, o que significa que uma escrita imediata por parte da TLB é escrita na cache L1, a mais rápida de todas;
  - Uma falha (*miss*) no acesso à TLB não implica que a página não esteja carregada na memória. A entrada da TLB pode ter sido simplesmente retirada da TLB pelo algoritmo de substituição, mas a página ter ficado carregada em memória. Neste caso, recuperação de uma falha no acesso é bem mais rápida do que se além da entrada da página virtual não estar carregada na TLB a página em si não estiver carregada em memória principal. O que não pode suceder é a página existir na TLB sem estar carregada na memória principal;

A abstracção de memória, tal como os programas a vêem, é unificada e automática. Aparte às vezes uns acessos demorarem mais do que os outros, a funcionalidade de um processo é sempre a mesma. Os dois mecanismos básicos (caches e memória virtual) têm de cooperar para conseguir implementar esta abstracção. A questão fundamental que se coloca é qual destes mecanismos está mais perto do processador e é tratado em primeiro lugar quando um acesso à memória ocorre, havendo duas possibilidades fundamentais quando um processador lança um pedido de acesso:

- As caches estão antes (são verificadas primeiro) e a memória virtual depois. Ou seja, as caches contêm endereços virtuais e os respectivos valores. Em caso de sucesso (*hit*), o acesso é bastante rápido pois nem sequer há necessidade de converter o endereço de virtual para físico, poupando-se o tempo de procura na TLB;
- A memória virtual está antes e as caches depois. Neste caso, o endereço virtual é primeiro convertido para físico e só depois se verifica se o respectivo valor está na cache, que assim trabalha ao nível físico.

Naturalmente, o mecanismo deve ser optimizado para o caso mais frequente, que devido à localidade dos programas é o acesso com sucesso. À partida o primeiro caso parece ser mais eficiente que o segundo, que tem de converter os endereços em todos os acessos. No entanto, apresenta a potencial dificuldade de que é possível dois endereços virtuais diferentes estarem mapeados no mesmo endereço físico, o que se traduziria por duas cópias do mesmo bloco na mesma cache, sem controlo da coerência entre ambas. A existência de mais do que um nível de cache dentro do processador traz ainda complicações adicionais. Por outro lado, as caches externas têm de funcionar com endereços físicos, pois a TLB e os restantes circuitos do mecanismo de memória virtual só conseguem ser suficientemente rápidos de forma interna ao processador.

Também as operações de DMA ao nível de endereços virtuais seriam muito mais complexas, pelo que geralmente se fazem ao nível físico. No entanto, tal implica geralmente invalidar o conteúdo da cache, pois algum dos blocos pertencente à gama de endereços alterados pela operação de DMA poderá estar carregado na cache e ficar incorreto.

Assim, e ignorando algumas optimizações comuns mas mais avançadas que o nível deste livro, esta secção considera que todas as caches de dados e instruções estão ao nível físico apenas a TLB trabalha ao nível dos endereços virtuais. A Fig. 7.30 ilustra a forma como os dois mecanismos estão integrados e se completam, desde o pedido de acesso com um

endereço virtual (numa instrução do tipo MOV, por exemplo), passando pelo endereço físico em que esse endereço virtual está mapeado e até obter o valor pretendido. Esta figura é apenas ilustrativa e não tem como objectivo um grande rigor de representação (número de bits dos campos, número de linhas da cache, tipo de cache, etc.).

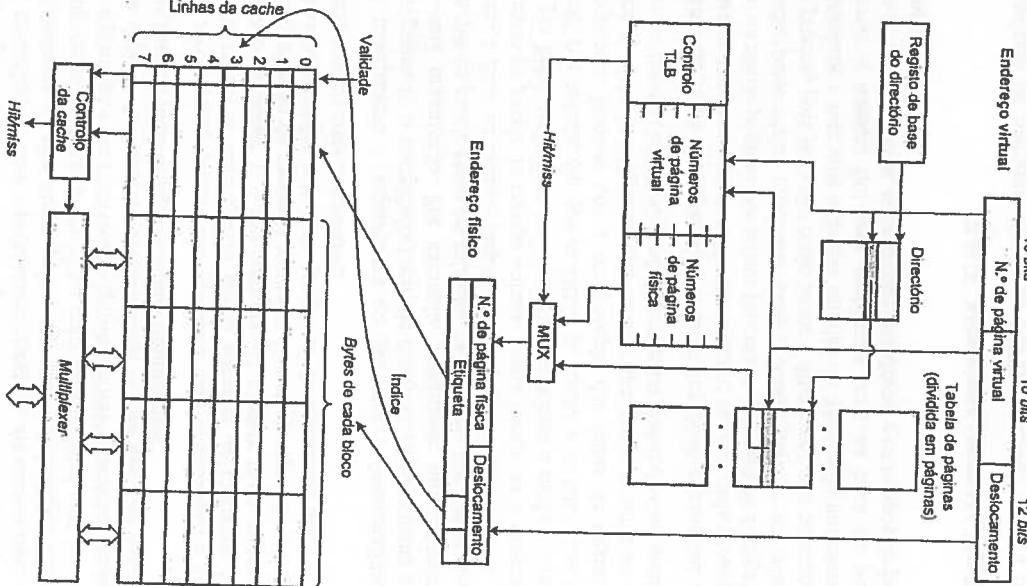


Fig. 7.30 - Integração dos mecanismos de memória virtual e caches

O endereço físico produzido é interpretado de forma diferente pelo mecanismo de memória virtual e pela cache (mas o valor é o mesmo). Nomeadamente, percebe-se que o tamanho da página é superior ao do bloco, pois os bits de menor peso da etiqueta ainda se sobreponem aos bits de maior peso do deslocamento.

O caminho a percorrer desde o pedido de acesso até a sua execução estar completa pode ser mais longo e tortuoso do que a Fig. 7.30 deixa antever, para além de alguns cuidados fundamentais a ter em relação à coerência da informação entre os vários mecanismos. Por exemplo, se uma página sair de memória para disco, quaisquer blocos dessa página presentes nas caches têm de ser invalidados (porque os endereços físicos dessa página serão reutilizados para mapar outra página virtual, com outro conteúdo).

O caminho mais rápido consegue-se se o mapeamento virtual-físico estiver presente na TLB, a página referenciada estiver carregada em memória principal e o bloco accedido (dentro dessa página) estiver presente na cache. Um acesso pode falhar em qualquer destes três pontos:<sup>114</sup>

- TLB – Uma falha do mapeamento virtual-físico na TLB envolve atravessar a tabela de páginas. Com o directório e uma tabela com um nível, como sucede na Fig. 7.30, isto implica:
- Ler a entrada no directório (esta página é residente);
- Se a página da tabela de páginas não estiver em memória, tem de ser carregada;
- Carregar na TLB a entrada com o mapeamento virtual-físico da página accida pelo processador.

**Página** – Uma falta de página é uma operação muito morosa em relação à velocidade de processamento do processador. Enquanto as falhas na TLB e nas caches de dados e instruções podem ser resolvidas rapidamente, com o processador a esperar, uma falta de página envolve gerar uma exceção, cuja rotina irá desencadear um pedido de acesso ao disco que bloqueará o processo onde ocorreu, ficando à espera da conclusão, altura em que a instrução que desencadeou o acesso (e a excepção de falta de página) será re-executada. Note-se que um acesso por parte do processador pode ocasionar duas faltas de página<sup>115</sup>, uma relativa à página da tabela de páginas (no caso de uma falta na TLB) e outra relativa à página a que o processador pretende aceder. A instrução pode assim recomeçar a sua execução mais de uma vez. Enquanto a leitura do disco decorre, tipicamente o processador passa a executar outro processo;

<sup>114</sup> Embora haja algumas combinações que não podem suceder, como por exemplo um bloco estar presente na cache mas a respectiva página não estar carregada em memória principal.

<sup>115</sup> As páginas de código e dados usadas pelo sistema operativo na execução do pedido de leitura das páginas em falta têm de ser residentes.

**Cache** – O bloco pretendido não está carregado na cache. A recuperacão é feita em hardware, normalmente com o processador à espera. As caches mais evoluídas, nos processadores de alto desempenho, suportam continuação dos acessos durante a recuperacão pela cache do bloco cujo acesso falhou até haver outra falha antes de a primeira estar resolvida. Algumas permitem mesmo várias falhas pendiientes, que depois vão sendo resolvidas. Estas caches que não bloqueiam o processador em caso de falha designam-se não bloqueantes.

Todo este conjunto de mecanismos encerra um equilíbrio cuja optimização é difícil de conseguir para todos os casos. O desempenho global depende de inúmeros factores, incluindo não apenas o computador e o seu software de suporte (tipo, capacidade e políticas das caches, organização da memória virtual e do sistema operativo, tempos de acesso do disco, etc.) como também o próprio programa do utilizador, que pode estar mais ou menos adequado a todos estes mecanismos.

A programação só é transparente à arquitectura no que toca à funcionalidade. No desempenho, quem programa (e o compilador) tem de conhecer minimamente a forma como o computador está organizado. Por exemplo, imagine-se um programa que manipula imagens, em que os pixels estão organizados na memória por linhas. Os pixels da mesma linha tendem a ficar na mesma página ou em páginas consecutivas, mas o mesmo não se passa entre os pixels da mesma coluna. Desta forma, se o programa for processando os pixels por linha, terá muito mais sucessos no acesso à cache e à memória virtual do que se o fizer por coluna, em que as falhas na cache e as falhas nas páginas serão muito mais frequentes. Note-se que a recuperacão das falhas de página é tão drasticamente lenta face ao ciclo de relógio do processador que qualquer aumento na taxa de falhas de página tem um impacte substancial no desempenho global do computador.

Naturalmente, uma memória física de capacidade insuficiente pode estrangular rapidamente o sistema quando se atinge um ponto de funcionamento em que cada processo que entra em execução tem de retirar páginas dos outros processos (por falta de espaço). Dada a partilha do processador pelos vários processos, cada processo gasta o seu tempo a ir buscar as páginas que precisa, pois já lá não estão desde a última vez que executou. Esta situação designa-se *thrashing* e tem uma analogia simples no trânsito de um cruzamento (com poucos automóveis o trânsito flui sem problemas, mas na hora de ponta os automóveis bloqueiam-se uns aos outros, mesmo com semáforos, e os tempos de percurso aumentam drasticamente).

### 7.6.7 MEMÓRIA VIRTUAL NO PEPE

A existéncia do mecanismo de memória virtual no PEPE destina-se mais a permitir verificar o seu comportamento do que a suportar um ambiente de exploração de políticas e estatísticas. Acresce o facto de este mecanismo exigir suporte de um sistema operativo e assumir a existéncia de um dispositivo de memória de massa (disco, nomeadamente), envolvendo já algum nível de complexidade. Assim, as capacidades do PEPE neste campo limitam-se ao essencial para suportar o mecanismo.

Os endereços virtuais têm apenas 16 bits, pois têm de circular pelos registos. A página é de 256 bytes (128 palavras), o que quer dizer que cada processo pode aceder a 256 páginas virtuais. A tabela de páginas só tem um nível, ocupando 256 palavras (uma palavra por cada entrada).

Os endereços físicos têm também 16 bits (maior não podia ser, pois o processador só suporta um espaço de endereçamento físico de 16 bits, e também não há interesse em o restringir a menos bits). Assim, o espaço físico também tem 256 páginas (embora a capacidade da memória física possa limitar o número de páginas virtuais carregadas simultaneamente).

O formato das entradas na tabela de páginas é indicado pela Tabela 7.20, com base na descrição da Tabela 7.19.

| BIT  | NAME               | SIGNS | FUNÇÃO                                                                                                                                                                                                                                                                                                                                         |
|------|--------------------|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 15   | Válida             | V     | Se $V=1$ , a página está carregada em memória principal, caso contrário está no swap file                                                                                                                                                                                                                                                      |
| 14   | Acedida            | A     | Se $A=1$ , a página foi acedida desde a última vez que o sistema operativo colocou os bits A das várias entradas a 0. Este bit é usado para a política LRU                                                                                                                                                                                     |
| 13   | Alterada           | D     | Se $D=1$ , pelo menos uma palavra da página foi alterada desde que foi carregada em memória principal. Se a página tiver de sair de memória principal, tem de ser escrita em disco para actualizar a cópia no swap file                                                                                                                        |
| 12   | Alterável          | W     | Se $W=0$ , nenhuma palavra da página não pode ser alterada. Se o programa tentar efectuar uma escrita com $W=0$ é gerada a exceção SO\_LETTURA (Tabela A.8, na página 702)                                                                                                                                                                     |
| 11   | Nível de protecção | P     | Se $P=0$ , a página é de nível de sistema. Se $P=1$ , é de nível de utilizador (seção 7.7.5, na página 682). Se o programa o tentar aceder a uma página de sistema ( $P=0$ ) com o bit $NP=1$ , é gerada a exceção D\_PROT ou I\_PROT (Tabela A.8, na página 702), consignante o acesso seja de dados ou de busca de instrução, respetivamente |
| 10   | Residente          | R     | Se $R=1$ , a página não pode participar do mecanismo de substituição de páginas, ficando residente na memória principal                                                                                                                                                                                                                        |
| 9    | Cacheable          | C     | Se $C=0$ , os acessos à página não passam pela cache, sendo sempre lidos ou escritos no endereço especificado. $C=1$ é usado tipicamente em páginas localizadas nos endereços dos periféricos (seção 7.5.6) e constitui uma alternativa ao mecanismo de inhibição da cache por hardware (seção 7.5.7)                                          |
| 8..0 | Referência         | Ref   | Número de página física (se $V=1$ ) ou número de página no swap file (se $V=0$ ). Se $V=1$ , o bit 8 é ignorado                                                                                                                                                                                                                                |

Tabela 7.20 - Formato das entradas na tabela de páginas

A base da tabela de páginas está contida no registo auxiliar RTP (Registo da Tabela de Páginas – Tabela A.3, na página 698). Note-se que cada processo tem o seu próprio espaço de memória virtual, o que implica a sua própria tabela de páginas. Sempre que o processo em execução muda, também o valor do RTP tem de mudar.

Dado que os circuitos do PEPE para acesso à memória em leitura/escrita de dados (MOV) e em busca de instruções são separados, existem duas TLBs separadas, uma para dados e outra para instruções. Ambas são completamente associativas, com blocos de uma palavra, e a sua capacidade e política de substituição de blocos pode ser configurada pelo registo auxiliar RCMV (Registo de Configuração da Memória Virtual – Tabela A.7, na página 700). As falhas nas TLBs são tratadas em hardware.

Este mesmo registo permite ligar ou desligar o mecanismo de memória virtual. Por omisso, está desligado após um reset, caso em que o processador considera que os endereços usados para fazer os acessos são físicos. Ao ligar a memória virtual, as TLBs são inicializadas (sem entradas carregadas).

O PEPE possui ainda o registo auxiliar RPID (Registo do Identificador do Processo – Tabela A.3, na página 698), que contém o PID, um número entre 0 e 255 (apenas é usado o byte de menor peso do RPID) que deve identificar unicamente o processo em execução. Este valor é guardado nas TLBs juntamente com o número de cada página virtual para identificar a que processo essa página virtual pertence, uma vez que todos os processos têm a mesma gama de endereços virtuais. Um acesso a um endereço virtual (em dados ou em busca de instruções) só reconhece um mapeamento virtual-físico já existente na TLB respectiva se tanto o número de página virtual como o número do processo conferirem.

A etiqueta das TLBs é formada pelo número da página virtual e pelo PID do processo a que esta pertence. Sem o PID, em cada mudança de processo ter-se-ia de invalidar todas as entradas das TLBs. A parte de dados de cada entrada da TLB é uma cópia da entrada da tabela de páginas, que tem de ser actualizada caso os bits A ou D tenham sido alterados. O bit V refere-se à entrada na TLB e não à tabela de páginas, mas de qualquer modo uma entrada não pode estar carregada na TLB sem a página correspondente estar carregada na memória principal.

Se, durante um acesso o bit V da entrada correspondente à página virtual acedida estiver a 0, é gerada a exceção D\_FALTA\_PAG ou I\_FALTA\_PAG (Tabela A.8, na página 702), consoante o acesso tenha sido em dados ou em busca de instruções, respectivamente. Aí, rotinas de tratamento destas interrupções têm de ler a página do disco (do swap file) em capacidade para 512 páginas, uma vez que só há 9 bits disponíveis na referência para páginas em disco (Tabela 7.20). Cabe ao software de suporte garantir que os processos em execução não usam, no total, mais páginas do que este limite.

Todas as instruções no PEPE são re-exectáveis. Na Tabela A.9, na página 707, pode observar-se que instruções como CALL ou RET só alteram o registo SP após todos os acessos à memória terem sido concluídos (usando endereços parciais calculados com base no valor do SP antes de ter sido alterado).

O algoritmo de substituição de páginas é muito simples, do tipo LRU, tendo por base os bits A das entradas da tabela de páginas, que periodicamente devem ser postos a 0.

Para utilizar a memória virtual, é preciso executar uma série de passos:

- Criar um swap file, com uma tabela interna de 512 entradas (para 512 páginas). Esta estrutura de dados destina-se a ser accedida pelo software de suporte à memória virtual (que inclui as rotinas de tratamento das exceções de falta de página) e deve ser projectada em conjunto. Pode ser implementada num periférico específico (para melhor simular um disco) ou numa zona de memória separada, não incluída no conjunto de páginas de memória física a usar pelo mecanismo de memória virtual;
  - Criar uma tabela de páginas, com 256 entradas, com o formato estabelecido pela Tabela 7.20;
  - Inicializar o RTP (Registo da Tabela de Páginas) com o endereço de base desta tabela;
  - Programar as rotinas de exceção associadas à memória virtual (D\_FALTA\_PAG ou I\_FALTA\_PAG) e incluir os seus endereços na tabela de exceções (secção 6.2);
  - Implementar o resto do software de suporte à memória virtual, incluindo política de substituição de páginas, rotina de exceção de protecção de escrita, etc.;
  - Implementar sistema de suporte de processos, incluindo gestão do registo RPID;
  - Criar e executar os processos.
- ### SIMULAÇÃO – MEMÓRIA VIRTUAL
- Esta simulação toma como base o conteúdo desta secção e das anteriores e exemplifica o funcionamento do mecanismo de memória virtual no PEPE, implementando em linguagem assembly do PEPE todo o software necessário para suportar processos com memória virtual. São focados em particular os seguintes aspectos:
- Verificação do funcionamento do mecanismo de memória virtual e da tradução de endereços;
  - Interface de utilizador, que permite em qualquer altura, com simulação passo a passo, visualizar o estado do mecanismo de memória virtual (imediatamente a TLB e a tabela de páginas) e de algumas estatísticas da sua utilização;
  - Análise de um acesso favorável (tudo carregado) e desfavorável (com faltas);
  - Substituição de páginas, com actualização do swap file com uma página modificada;
  - Protecção de escrita e de acesso a páginas de sistema;
  - Partilha de memória entre dois processos;
  - Acesso a periféricos;
  - Análise do comportamento do sistema com programas bem e mal concebidos.

**ESSENCIAL**

- A memória virtual é um mecanismo concebido para eliminar as limitações da memória física (endereços físicos em número limitado), usando a memória principal como se fosse uma cache do disco. Não aumenta o desempenho em relação a usar apenas memória principal física (pelo contrário, é diminui).
- Este mecanismo é transparente para a funcionalidade dos programas, mas estes beneficiam de um aumento aparente da capacidade de memória, podem ser carregados (em qualquer endereço) e até partilhar zonas de memória física ou periféricos em endereços virtuais diferentes, de forma protegida;
- A unidade de transferência entre o disco e a memória principal é a página. Uma falta de página gera uma exceção, que permite ao sistema operativo ir ao disco buscar a página e repetir o acesso.

- O mapeamento entre endereços virtuais e físicos faz-se por meio de uma tabela de páginas. Para acelerar o processo, usa-se uma TLB, uma pequena cache que contém os mapeamentos entre páginas virtuais e físicas mais frequentes;
- Quando alguma palavra de uma página é alterada, marca-se essa página. Se tiver de sair de memória principal (porque está cheia e o espaço é preciso), tem de actualizar a versão em disco primeiro.

## 7.7 SUPORTE PARA PROCESSOS

### 7.7.1 MODELOS DE PROGRAMAÇÃO E DE EXECUÇÃO

Os primeiros computadores conseguiam executar apenas um programa de cada vez. Hoje em dia, qualquer computador pessoal executa sem problemas várias dezenas de programas e um grande servidor pode estar a executar milhares de programas simultaneamente. A secção 6.2.2.5, na página 469, já apresentou um exemplo de um programa que executa várias tarefas independentes (ler botões, detectar passagens por zero da tensão da rede eléctrica e contar o tempo em que o triciclo está ligado). No entanto, trata-se apenas de um programa, que está apenas a fazer uma tarefa (ler os botões). As outras são invocadas periodicamente à custa de interrupções. Estas tarefas contribuem para um fim comum ( controlo da intensidade luminosa de uma lâmpada) e têm de ser pensadas de forma coordenada para o conjunto funcional de forma satisfatória.

Com o aumento da complexidade do software, em que um computador tem de lidar com programas totalmente independentes mas que partilham os mesmos recursos físicos (processador, memória e periféricos), já não é possível programar tudo de forma coordenada e a única solução é o processador proporcionar suporte para a execução

simultânea de programas independentes, designados processos. Um processo engloba a informação sobre um programa em execução.

Este termo "simultânea" sugere que em qualquer instante o computador está realmente a executar  $n$  instruções, uma de cada processo. De facto, os processadores modernos de alto desempenho possuem mais do que um recurso de cada tipo (ALUs ou mesmos núcleos, no mesmo tempo. Quando o computador possui vários processadores (ou um processador com vários núcleos), diz-se que há multiprocessamento, permitindo a execução ao mesmo tempo de vários processos (tanto quantos os processadores ou núcleos do processador). Um computador que suporta multiprocessamento designa-se multiprocessador [Patterson 2008, Stallings 2006].

Um processador com um único núcleo só consegue executar um processo de cada vez (ou mais, embora de forma fixa, com a técnica de hiperfluxo – página 522). No entanto, consegue simular a execução simultânea de muitos processos, dedicando algum tempo a cada processo e percorrendo-os a todos, rotativamente. Desta forma, o processador é partilhado por todos os processos. Se a rotatividade for muito rápida face à escala humana (tipicamente, poucos milissegundos dedicados a cada processo), parece que todos os processos estão a correr ao mesmo tempo, embora na realidade em cada instante só um esteja a ser executado (tal como sucede numa sessão simultânea de xadrez, em que o mestre dedica um pouco do seu tempo a cada um dos participantes). Os processos não são realmente executados ao mesmo tempo, mas apenas num regime de partilha de tempo designado multiprogramação. Cada processador de um multiprocessador pode executar vários processos em regime de multiprogramação.

Os programas devem ser construídos de forma a que os seus resultados não dependam do regime em que são executados nem de quantos nem quais processos estejam a ser executados também no mesmo computador. Desta forma, a terminologia que se usa em termos de modelo de programação dos programas é programação sequencial (em que um processo admite que é o único a executar) ou programação concorrente (em que um processo sabe que há outros processos também em execução e está preparado para interagir com eles, independentemente de ser em regime de multiprocessamento ou de multiprogramação).

### 7.7.2 MULTIPROGRAMAÇÃO

A multiprogramação requer um gerador de processos (que faz parte do sistema operativo) para gerir a partilha dos recursos do processador pelos vários processos. Cada processo é executado como se fosse o único no processador e sem se preocupar quando é que os outros correm. Periodicamente, um temporizador gera uma interrupção que invoca uma rotina que:

- Guarda os registos do processador em memória, incluindo o PC, SP e RE, e ainda o RTP (se estiver a usar a memória virtual) e o RPID (que contém o identificador do processo), o que corresponde ao estado do processo em execução e se designa por contexto<sup>116</sup> desse processo;
- Lê de memória o contexto de outro processo;
- Transfere o controlo para esse processo, que recomeça no ponto em que tinha sido interrompido, como se não tivesse havido entretanto mudança para outro processo.

O objectivo desta interrupção é retirar o controlo do processador (preempção) do processo em execução e dá-lo a outro processo, de modo a que todos os processos tenham oportunidade de ser executados, mesmo que um dado processo esteja a executar um ciclo sem fim.

O período da interrupção define o tempo que cada processo pode correr de cada vez. Designa-se fatia de tempo (*time slice*). Os contextos dos vários processos constam de uma lista de processos que vai sendo percorrida por esta rotina, de cada vez que é invocada, voltando ao princípio quando chega ao fim. Pela função de despachar controlo do processador para um dos processos, esta rotina é normalmente designada despachante (*dispatcher*) e a operação que desempenha por despacho (*dispatch*).

A Fig. 7.31 ilustra a partilha de tempo por quatro processos, mostrando o tempo em que ocorrem as interrupções de fatia de tempo. Pretende-se que o tempo dedicado ao despacho seja o menor possível, de forma a reduzir o peso do próprio sistema operativo no desempenho global do processador, pelo que os tempos nesta figura não estão necessariamente à escala.

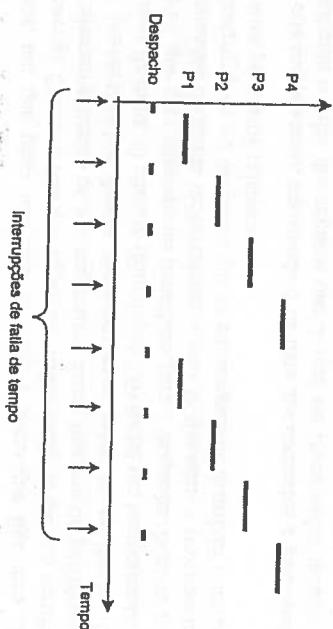


Fig. 7.31 - Multiprogramação: despacho e partilha de tempo por vários processos

A multiprogramação consiste assim numa alternância, por parte de cada processo, entre dois estados em termos de utilização do processador. Ou está em execução ou está executável (à espera da sua vez). No entanto, durante a execução, um processo pode ter de esperar que um recurso (um periférico, por exemplo) esteja disponível (tipicamente por estar a ser usado por outro processo ou por ainda estar a fazer alguma tarefa) ou que um dado evento ocorra (o utilizador carregue numa tecla do teclado, decorram N unidades de tempo, etc.). Se tal acontecer a um processo, o sistema operativo passa-o para o estado bloqueado, retirando-o de execução e passando-o para a lista dos processos bloqueados. Desta forma, evita-se que este processo consuma inutilmente a sua fatia de tempo em espera activa (teste de uma dada condição repetido continuamente, em ciclo) de algo que pode demorar um longo tempo a acontecer. Quando o processo puder prosseguir (o recurso ficar disponível ou o evento esperado ocorrer), o seu contexto é mudado para a lista dos processos executáveis, entrando depois em execução quando chegar a sua vez. A Fig. 7.32 ilustra o modelo básico<sup>117</sup> de estados dos processos.

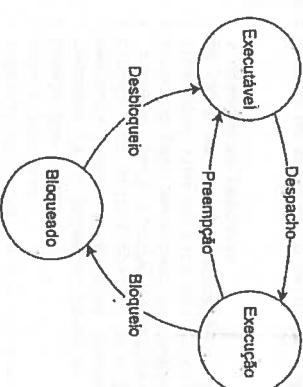


Fig. 7.32 - Modelo básico de estados dos processos. Só pode haver um processo em Execução. Os processos no estado Executável constam de uma lista e os que estão no estado Bloqueado constam de outra. Mudança de estado implica passagem para outra lista

O mecanismo circular (*round robin*), de atravessamento da lista de processos, em que cada um corre à vez e depois repete-se a volta, não é a única política de escolha (escalonamento, ou *scheduling*) do próximo processo a correr. Entre outras variantes possíveis, a generalidade dos sistemas operativos suporta prioridades, em que a cada processo é atribuída uma prioridade e esse processo só é executado quando nenhum processo de maior prioridade (se houver) estiver em condições de ser executado (tipicamente, os processos de maior prioridade passam a maior parte do tempo bloqueados, correndo apenas quando um dado recurso – normalmente um semáforo, secção 7.7.1 – fica disponível). Esta operação de escalonamento é normalmente separada da operação de despacho e é executada por outra rotina designada escalonador (*scheduler*). No exemplo desta secção, por ser muito simples, as duas operações são executadas pela mesma rotina

<sup>116</sup> Poderá ter outros nomes, dependendo do sistema operativo, mas é apenas uma questão de nomenclatura.

<sup>117</sup> Os sistemas operativos suportam normalmente mais estados, como por exemplo suspenso, mas tal está fora do âmbito deste livro.

O Programa 7.2 ilustra o funcionamento da multiprogramação e do despacho de uma forma extremamente simples em relação a qualquer sistema operativo digno desse nome. O contexto de cada processo consiste apenas no valor do SP (o resto do estado de cada processo é todo guardado na sua pilha) e a lista dos contextos não passa de uma tabela com os valores do SP, que é percorrida ciclicamente.

Só há dois processos, um que pisca um LED (Pisca) e outro que liga dois LEDs alternadamente (Alternar). Como os bits que ligam aos LEDs estão no mesmo periférico, ambos os processos alteram primeiramente os bits respectivos numa imagem do periférico dos LEDs na memória (valorLEDs) e depois escrevem esse valor no periférico. Cada processo tem a sua zona de pilha. O circuito (Simulação 7.6) inclui um temporizador que liga à interrupção INT0, que gera interrupções periódicas para o despacho. Note-se que os processos são programados de forma independente, como se tivessem o processador só para si, e são estas interrupções e o despacho que se encarregam de distribuir o tempo do processador pelos vários processos existentes.

A rotina CriaProc cria um processo, registando-o na lista dos contextos e guardando na pilha que lhe for atribuída o valor dos registos que devem ser recuperados quando esse processo for executado. Cada processo tem um identificador único (PID – process identifier), um número entre 0 e 255. O PID de cada novo processo é igual ao anterior mais um.

A rotina Despacho guarda o contexto de um processo na sua pilha e repõe o contexto do processo seguinte. No programa principal, que faz as inicializações, são também criados os processos e transferido o controlo para o primeiro processo, o que envolve ir buscar o seu contexto e retornar tal como Despacho o faz. Este retorno usa os valores pré-guardados na pilha. Os restantes processos são depois invocados pela rotina Despacho. O PID do processo corrente é armazenado no registo auxiliar RPID (Registo do Identificador do Processo – Tabela A.3, na página 698).

Este exemplo nunca destrói processos, mas tal não é difícil, bastando eliminar o registo do processo na tabela dos contextos. A maior dificuldade é gerir os "buracos" que esta operação cria na tabela dos contextos. Por esta razão, os sistemas operativos usam listas ligadas (secção 5.8.6, na página 382), que suportam facilmente a inserção e remoção de elementos (contextos dos processos).

A Simulação 7.6 permite verificar que os dois processos executam "simultaneamente", parecendo independentes, e contém detalhes adicionais sobre o funcionamento deste exemplo.

```

PLACE          1000H
tabCont:      TABLE
pilha:        TABLE
                800
                ; reserva espaço para os contextos dos processos
                ; (MAXNUMP * TAMPILA)
pilhapp:      TABLE
                30
                ; pilha para o programa principal
                ; valor de inicialização do SP do prog. principal
numP:         WORD
                ; numero actual de processos
valorLEDs:   WORD
                ; imagem na memória do valor dos LEDs
tabInt:       WORD  Despacho ; tabela de exceções (só tem INT0)
                ; programa principal, inicializações, etc.
início:      MOV    BTE, tabInt ; inicializa BTE com tabela de exceções
                ; SP, fimpilhapp; inicializa SP com pilha do programa principal
                ; R0, 0
MOV          R0, numP
MOV          R1, [R1], R0 ; inicializa número actual de processos
MOV          R0, RPID, R0 ; inicializa número do processo a correr
MOV          R1, valorLEDs
MOV          [R1], R0 ; inicializa imagem na memória do valor dos LEDs
MOV          R0, Pisca ; endereço do processo que pisca o LED 0
CALL         Criaproc ; cria o processo
MOV          R0, Alternar ; endereço do processo que alterna os LEDs
CALL         Criaproc
MOV          R0, tabCont ; cria o processo
MOV          R0, tabCont ; tabela dos contextos
MOV          SP, [R0] ; recupera SP do primeiro processo
POP          PC ; restaura todos os registos (excepto PC e RE, que
                ; tinham sido salvos aparte, SP e TEMP)
RE           ; "retorna" para o primeiro processo (o outro será
                ; invocado pelo mecanismo do despacho). Esta
                ; instrução liga os bits das interrupções devido
                ; ao valor de RE pre-guardado na pilha durante
                ; a criação do processo
;
```

```

; processo que pisca o LED 0
Pisca:
    MOV    R1, valorLEDs ; imagem na memória do valor dos LEDs
    MOV    R2, LEDS ; endereço do periférico dos LEDs
    mov    R3, [R1] ; 1é imagem na memória do valor dos LEDs
    ciclo:  MOV    R3, 0 ; troca o bit 0 (LED deste processo)
            CPL   R3, 0 ; actualiza imagem na memória do valor dos LEDs
            MOV    R1, R3 ; actualiza valor dos LEDs no periférico
            MOVB  [R2], R3 ; valor de espera
            MOV    R0, ESPERA ; rotina que espera um tempo
            CALL   Espera ; continua o ciclo
            JMP    ciclo ; continua o ciclo
;
```

processo que liga alternadamente os LEDs 2 e 3

```

; alterna:
    MOV    R1, valorLEDs ; imagem na memória do valor dos LEDs
    MOV    R2, LEDS ; endereço do periférico dos LEDs
    ciclo:  MOV    R3, [R1] ; 1é imagem na memória do valor dos LEDs
            CPL   R3, 2 ; troca o LED 2
            BTR  R3, 2 ; vê estado do LED 2 para o bit 3 ficar ao contrário
            JZ    bit1 ; se o LED 2 está a 0...
            CLR   R3, 3 ; senão, coloca-o a 0...
            JMP    alterna ; de um processo (tem de ter IEO e TE a 1)
;
```

```

bit1: SET R3, 3 ; ... coloca o bit 3 a 1
altera: MOV [R1], R3 ; actualiza imagem na memoria do valor dos LEDs
        MOV [R2], R3 ; valor de espera
        MOV R0, ESPERA ; rotina que espera um tempo
        CALL Espera ; continua o ciclo
        JMP cicl0

; rotina que espera algum tempo por meio de um ciclo. Recebe em R0 o numero
; de interacções a esperar
Espera: SUB R0, 1 ; menos uma iteração para esperar
        JNZ Espera ; executa ciclo até R0 chegar a 0
        RET

; rotina que cria um processo. No R0 recebe o endereço de entrada do processo
; (da sua primeira instrução). Na imagem de R0 guardada na pilha do novo
; processo fica o seu número do processo.

CriaProc:
        PUSH R1 ; guarda registos usados
        PUSH R2
        PUSH R3
        MOV RL, numP ; número actual de processos
        MOV R2, [R1]
        ADD R2, 1 ; passa a haver mais um processo
        MOV R3, MAXNUMP ; numero máximo de processos
        CMP R2, R3
        JGE fin ; se já não pode criar mais processos, retorna
        MOV [R1], R2 ; actualiza o número actual de processos
        MOV R3, TAMPILHA ; tamanho da pilha de cada processo em palavras
        MUL R3, R2 ; espaço de pilha ocupado por todos os processos
        SHL R3, 1 ; base das pilhas dos processos
        MOV RL, pilha ; endereço logo após a pilha do novo processo
        ADD RL, R3 ; primeira palavra útil da pilha deste processo
        SUB RL, 2 ; coloca lá o endereço do processo
        MOV [RL], RO ; palavra seguinte da pilha deste processo
        MOV RO, IMAGEMRE ; valor de RE com interrupções permitidas
        MOV [RL], RO ; coloca esse valor de RE na pilha
        RESV RL, 26 ; reserva espaço necessário para 13 registos
        MOV R3, 26 ; principais (todos excepto RE, TEMP e SP)
        SUB RL, R3 ; base da tabela dos contextos
        MOV R3, tabCont ; número do processo actual (n.º de processos - 1)
        SUB R2, 1 ; multiplica n.º de processos por 2
        ADD R3, R2 ; endereço na tabela do contexto deste processo
        MOV [R3], RL ; coloca lá o valor do SP deste processo
        POP R3 ; restaura o valor dos registos
        POP R2
        POP R1
        RET

```

CriaProc:

PUSH R1 ; guarda registos usados

PUSH R2

PUSH R3

MOV RL, numP

MOV R2, [R1]

ADD R2, 1

MOV R3, MAXNUMP

JGE fin

MOV [R1], R2

MOV R3, TAMPILHA

MUL R3, R2

SHL R3, 1

MOV RL, pilha

ADD RL, R3

SUB RL, 2

MOV [RL], RO

MOV RO, IMAGEMRE

MOV [RL], RO

RESV RL, 26

MOV R3, tabCont

SUB R2, 1

SHL R2, 1

ADD R3, R2

MOV [R3], RL

POP R3

POP R2

POP R1

RET

; rotina que espera algum tempo por meio de um ciclo. Recebe em R0 o numero

de interacções a esperar

Espera: SUB R0, 1

JNZ Espera

RET

; rotina que cria um processo. No R0 recebe o endereço de entrada do processo

(da sua primeira instrução). Na imagem de R0 guardada na pilha do novo

processo fica o seu número do processo.

Programa 7.2 - Programação com processos (multiprogramação), com recurso a uma rotina de despacho invocada por interrupções de fatia de tempo

### NOTA

A mudança de contexto (*context switch*) de processo é uma actividade crítica em que não pode haver exceções, como por exemplo falta de página. Por este motivo, um sistema operativo que suporte memória virtual deverá ter o cuidado de colocar a lista dos contextos dos processos numa página declarada residente (Tabela 7.19, na página 652).

### Simulação – MULTIPROGRAMAÇÃO

Esta simulação toma como base o Programa 7.2 e exemplifica o funcionamento da multiprogramação no PEPE. Em particular, são focados os seguintes aspectos:

- Criação de processos;
- Verificação do funcionamento da multiprogramação e de que um processo não bloqueia outro, apesar de incluir ciclos infinitos, nem interfere com os valores dos registos dos outros processos, apesar de utilizar os mesmos registos;
- Distinção entre tempo de processo e tempo real (aumento do tempo real de processamento com o aumento do número dos processos);
- Funcionamento do mecanismo do despacho;
- Efeito do aumento exagerado da fatia de tempo de cada processo;
- Marcação das temporizações por interrupção (temporizador) em vez de ser por contagem de iterações (variante ao Programa 7.2).

### 7.7.3 INTERACÇÃO ENTRE PROCESSOS

#### 7.7.3.1 SINCRONIZAÇÃO DE BAIXO NÍVEL

Embora aparentemente o Programa 7.2 funcione bem, padece de um problema clássico na programação concorrente, quando vários processos interagem. Na Simulação 7.6, o efeito

```

MOV R2, RPID ; numero do processo a correr
SHL R2, 1 ; multiplicar por 2 (endereçamento de byte)
MOV [R0+R2], SP ; guarda o valor actual do SP
MOV R3, numP ; numero actual de processos
MOV R4, [R3] ; repõe numero do processo a correr
SER R2, 1 ; passa ao processo seguinte
ADD R2, 1
CMP R2, R4 ; se já chegou ao último, ...
JLT prox ; ... é preciso voltar ao princípio
MOV RPID, R2 ; actualiza o numero do processo a correr
SHL R2, 1 ; multiplicar por 2 (endereçamento de byte)
MOV SP, [R0+R2] ; obtém o valor de SP do novo processo
POPC ; restaura todos os registos (excepto PC e RE, que tinham sido salvos aparte, SP e TEMP)
REE ; retorna da rotina (para o novo processo)

```

dese problema nota-se por uma irregularidade quase imperceptível, de vez em quando, no ritmo dos LEDs.

Para se perceber melhor qual o problema, note-se que o algoritmo de cada processo é genericamente o seguinte:

1. Lê imagem (na memória) dos LEDs para um registo.
2. Altera o(s) bit(s) do(s) LED(s) desse processo nessa imagem.
3. Actualiza essa imagem na memória.
4. Escreve essa imagem no periférico para actualizar o valor dos LEDs.
5. Espera um tempo.
6. Repete o ciclo.

Como a interrupção de fáta de tempo e a mudança de processo pode acontecer em qualquer altura, é perfectamente possível que a situação da Fig. 7.33 (ou outra semelhante) ocorra. O processo Piscia lê a imagem dos LEDs e começa a alterá-la num registo, mas antes que possa registar essa alteração na imagem, em memória, ocorre uma mudança de processo e passa a correr o processo Alterna, que faz outras modificações e regista-as na imagem dos LEDs na memória e nos próprios LEDs, no periférico.

Pouco tempo depois de a esperar o processo Alterna começar, volta a executar o processo Piscia, que não sabe nada das alterações nos LEDs do processo Alterna e calmamente actualiza a imagem dos LEDs e o valor dos próprios LEDs. Os LEDs 2 e 3 voltam a ter o valor anterior à última execução do processo Alterna.

O resultado visual é estes LEDs manterem o último valor deixado pelo processo Alterna durante muito menos tempo do que o que devia (o tempo de espera do passo 5 do algoritmo), dando a sensação de um piscadelha rápido.

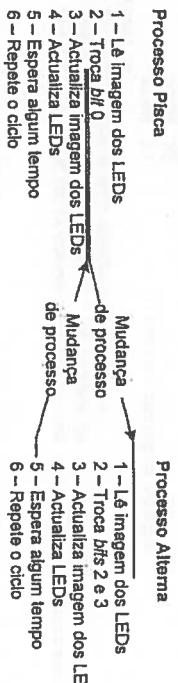


Fig. 7.33 - Interferência na sequência temporal do processamento dos processos causada pela multiprogramação quando mais do que um processo usa a mesma informação

Naturalmente, isto só ocorre de vez em quando, quando o acaso do ritmo de mudança de processo determina uma situação desse género, mas a probabilidade de ocorrer pode ser muito aumentada se, por exemplo, o passo 5 for mudado para antes do passo 3, pois o tempo entre ler a informação partilhada e a actualizar (depois de a ter modificado) aumenta devido ao tempo de espera (Simulação 7.7).

Este problema é crucial em muitas actividades da vida real. Imagine, por exemplo, que um cliente vai a uma agência de viagens e pergunta se há lugares num dado voo. Dizem-

lhe que só há mais um lugar disponível mas, enquanto decide se quer aproveitar ou não, outro cliente, noutra agência, faz a mesma pergunta e decide mais depressa. Quando finalmente o primeiro cliente diz que sim, a agência tem de lhe dizer que afinal já não há lugar, o que naturalmente é desagradável, tendo em conta que tinha acabado de dizer que havia um lugar.

A solução é garantir que enquanto um processo está a alterar informação partilhada (leendo, alterando e actualizando), nenhum outro pode estar a fazer o mesmo. Esta restrição designa-se exclusão mútua (o primeiro a começar a fazer este procedimento exclui todos os outros). O objectivo é garantir que todas as operações, desde a leitura da informação até à sua actualização final, sejam efectuadas de forma atómica, sem interrupções ou divisões. Ou se faz tudo ou não se faz nada. Uma secção de um programa que tenta de ser executada deste modo (com exclusão mútua) designa-se secção crítica.

Uma solução seria desligar as interrupções antes de um processo entrar numa secção crítica, mas isso desligaria também a multiprogramação para processos que não manipulam a informação partilhada em causa. A solução universalmente adoptada consiste em cada processo tentar reservar a secção crítica antes de entrar nela (se já estiver reservada, fica à espera até o conseguir) e libertar essa reserva quando terminar a execução da secção crítica. Em cada instante, só um processo (no máximo) pode deter essa reserva.

Para este efeito, cada secção crítica usa uma variável, designada trinco lógico (*lock*), que está aberta se tiver o valor 0 e fechada se tiver outro valor. Há duas rotinas, Entrar e sair, que cada processo tem de invocar imediatamente antes de entrar e após sair de uma secção crítica, respectivamente.

Para Sair, basta colocar o trinco lógico a 0, pois tal é feito numa única operação indivisível (basta uma instrução mov). Para Entrar, no entanto, é preciso ler o valor do trinco lógico e colocar lá 1 (por exemplo), entrando na secção crítica apenas se o valor lido for 0. No entanto, entre a leitura e escrita do trinco lógico pode haver uma interrupção de mudança de processo, pelo que o problema se mantém.

A solução é dada pela instrução SWAP entre um registo e a memória (SWAP R0, [R1], por exemplo – página 705), que permite trocar os valores de uma posição de memória e de um registo (uma leitura e uma escrita dessa posição de memória) numa só instrução de forma atómica (indivisível).<sup>118</sup>

Desta forma, a rotina Entrar coloca 1 no registo e faz um SWAP. Se leu 1, a secção crítica já estava reservada por outro processo e tem de continuar a tentar. Se leu 0, a secção estava livre e entretanto já a reservou (a instrução SWAP deixou 1 no trinco lógico), podendo então regressar para o processo entrar na secção crítica.

<sup>118</sup> Outros processadores usam outras instruções semelhantes, como por exemplo TAS (Test and Set), que afecta um bit de estado e permite efectuar um salto condicional com base no valor do trinco lógico (evitando assim uma instrução de comparação).

**NOTA**

O trinco lógico corresponde ao valor de uma célula de memória e às operações Entrar e Sair. É um mecanismo de software (embora necessite de suporte em hardware para a operação atómica) e não deve ser confundido com o trinco introduzido na secção 2.6.1.1, na página 57, que consiste num bloco de hardware. O termo anglo-saxónico, lock, seria provavelmente traduzido de forma mais correcta por fechadura, fecho ou cadeado. Um ferrolho ou um trinco é apenas a lingueta que corre na fechadura e garante mecanicamente o fecho. No entanto, o termo "trinco lógico" é o que tem mais tradição em português na área de sistemas operativos, pelo que é o termo usado neste livro.

O Programa 7.3 ilustra esta solução, corrigindo o problema da exclusão mútua no Programa 7.2. Apenas as alterações e partes relevantes são indicadas, mantendo-se o resto do programa. A Simulação 7.7 contém o programa completo e mostra que as irregularidades do ritmo dos LEDs desaparecem.

```

trinco: WORD 0          ; variável para exclusão mútua
. . .
MOV R0, 0                ; rotina que usa uma variável de exclusão mútua para proteger uma secção crítica
MOV R1, trinco           ; (faz um processo esperar até conseguir entrar na secção crítica)
. . .
MOV [R1], R0              ; guarda registos
. . .
; inicializa variável de exclusão mútua
; processo que pisca o LED 0
; rotina que liberta uma secção crítica
Pisca:
    MOV R1, valorLEDs      ; imagem na memória do valor dos LEDs
    MOV R2, LEDS            ; endereço do periférico dos LEDs
    Entrar: CALL R3, [R1]     ; espere até entrar na secção crítica
    ; le imagem na memória do valor dos LEDs
    CPL R3, 0               ; troca o bit 0 (LED deste processo)
    MOV [R1], R3             ; actualiza imagem na memória do valor dos LEDs
    MOVB [R2], R3             ; actualiza valor dos LEDs no periférico
    CALL Sair                ; liberta a secção crítica
    MOV R0, ESPERA           ; valor de espera
    CALL Espera              ; rotina que espera um tempo
    JMP ciclo                ; continua o ciclo
; processo que liga alternadamente os LEDs 2 e 3
; Alterna:
    MOV R1, valorLEDs      ; imagem na memória do valor dos LEDs
    MOV R2, LEDS            ; endereço do periférico dos LEDs
    Entrar: CALL R3, [R1]     ; espere até entrar na secção crítica
    ; le imagem na memória do valor dos LEDs
    CPL R3, 2               ; troca o LED 2
    BIT R3, 2                ; vé estado do LED 2 para o bit 3 ficar ao contrário
    JZ bit1                 ; se o LED 2 está a 0...
    CLR R3, 3                ; senão, coloca-o a 0
    altera: SET R3, 3          ; ... coloca o bit 3 a 1
    MOV [R1], R3              ; actualiza imagem na memória do valor dos LEDs
    MOVB [R2], R3             ; actualiza valor dos LEDs no periférico
    CALL Sair                ; liberta a secção crítica
    MOV R0, ESPERA           ; valor de espera
. . .

```

```

CALL Espera              ; rotina que espera um tempo
JMP ciclo                ; continua o ciclo
. . .
; rotina que usa uma variável de exclusão mútua para proteger uma secção crítica
; (faz um processo esperar até conseguir entrar na secção crítica)
Entrar: PUSH R0           ; guarda registos
        PUSH R1
        MOV R0, trinco         ; variável de exclusão mútua
        MOV R1, 1               ; valor para fechar o trinco lógico
        SWAP R1, [R0]           ; tenta fechar o trinco lógico (operação atómica)
        JNZ deNovo             ; se estava fechado, tem de continuar à espera
        deNovo: SWAP R1, 0       ; estava aberto (valor 0)?
        POP R1, [R0]             ; conseguiu! Pode recuperar os registos...
        POP R0
        RET                     ; ... e regressar
. . .
; rotina que liberta uma secção crítica
Sair: PUSH R0             ; guarda registos
      PUSH R1
      MOV R0, trinco         ; variável de exclusão mútua
      MOV R1, 0               ; valor para abrir o trinco lógico
      MOV [R0], R1             ; abre o trinco lógico
      POP R1                 ; Já está. Pode recuperar os registos...
      POP R0
      RET                     ; ... e regressar
. . .

```

**Programa 7.3 - Utilização de um trinco lógico para impor exclusão mútua aos processos na execução de uma secção crítica****SIMULAÇÃO 7.7 – EXCLUSÃO MÚTUA**

Esta simulação toma como base o Programa 7.3 e exemplifica o funcionamento do mecanismo de exclusão mútua no PEPE. São focados em particular os seguintes aspectos:

- Efeito da inclusão do tempo de espera antes da actualização da imagem dos LEDs;

- Atomicidade da instrução SWAP;
- Funcionamento das rotinas de acesso ao trinco lógico.

**7.7.3.2 SÍNCRONIZAÇÃO COM SEMÁFOROS**

Note-se que, com a solução do Programa 7.3, um processo que recebe o processador pode ficar a tentar entrar e a consumir a sua fatia de tempo do processador inutilmente, por o processo que detém a reserva de uma secção crítica ter deixado de executar (devido à mudança de processo) antes de libertar essa secção (espera activa, semelhante ao já discutido na secção 6.4.2.1, na página 507). Por este motivo, este mecanismo deve usar-se para secções críticas de pequena duração.

Quando se pretende proteger secções de duração mais longa ou coordenar o acesso a recursos de utilização mais complexa, usa-se um mecanismo de mais alto nível e que está

presente em qualquer sistema operativo, que é designado semáforo (por analogia com os semáforos do trânsito, que num cruzamento impõem exclusão mútua entre ruas que se cruzam). O princípio é o mesmo (só um processo pode entrar na secção de cada vez), mas com a vantagem de que se um processo não puder entrar fica bloqueado (Fig. 7.32), sem consumir tempo do processador. O Programa 7.4 ilustra a funcionalidade das funções (na linguagem C, por se tratar de um mecanismo de nível alto) de acesso a um semáforo, normalmente designadas Esperar (Wait) e Assinalar (Signal).<sup>119</sup> Semáforo é uma estrutura de dados (struct, em C) que inclui um valor (um inteiro) e uma lista dos contextos dos processos bloqueados neste semáforo, tal como ilustrado pela Fig. 7.34.

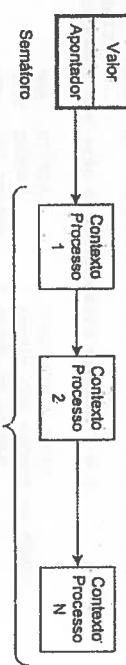


Fig. 7.34 - Um semáforo é essencialmente constituído por uma variável inteira e por uma lista dos contextos dos processos bloqueados neste semáforo (que pode estar vazia)

Quando um processo tenta entrar na secção crítica protegida por este semáforo, deve verificar se o semáforo está livre, invocando a função Esperar. Se estiver, ocupa o semáforo e pode avançar. Caso contrário, o processo é bloqueado, transferindo-se o seu contexto para a lista de processos do semáforo. Quando o processo que estava a ocupar o semáforo sai da secção crítica, invoca a função Assinalar, que desbloqueia (passando a executável) o primeiro processo da lista dos bloqueados nesse semáforo ou, caso não haja nenhum processo bloqueado nesta lista, liberta o semáforo.

Os semáforos binários são usados para exclusão mútua e só deixam passar no máximo um processo de cada vez. A sua variável pode ter o valor 1 (livre) ou 0 (ocupado). No entanto, o mecanismo pode ser generalizado para deixar passar no máximo N processos (semáforos N-ários), o que é útil em muitas aplicações (como por exemplo quando só há N cópias de um recurso e só os primeiros N processos têm direito a usá-lo, tendo os outros de esperar). A variável é inicializada com o valor N e cada invocação da primitiva Esperar reduz o valor da variável de uma unidade (até chegar a 0). Os semáforos binários são um caso particular destes, com N=1.

Como as funções podem servir para qualquer semáforo, recebem a variável correspondente como argumento. O semáforo deve ser inicializado com o N pretendido e com a lista de contextos vazia (a declaração e inicialização do semáforo estão omitidas por simplicidade).

\* Espera até conseguir entrar na secção protegida por este semáforo \*/  
esperar (Semáforo sem)

```

Entrar (trinco); /* só um processo de cada vez pode entrar aqui */
if (sem.valor > 0) /* se o semáforo ainda tem lugares disponíveis */
sem.valor--;
/* reserva um lugar para este processo e prossegue */
else
    /* senão, tem de bloquear o processo */
    {
        remover (listaExecutáveis, processo); /* deixa de estar executável */
        inserir (sem.lista, processo); /* e fica bloqueado neste semáforo */
    }
}

```

Sair (trinco); /\* liberta a secção do semáforo em si \*/
assinalar (Semáforo sem)
 /\* Indica que já libertou este semáforo \*/
}

```

Entrar (trinco); /* só um processo pode entrar de cada vez */
if (tamanho (sem.list) == 0) /* se não há processos bloqueados */
sem.valor++; /* aumenta o número de lugares disponíveis */
else
    /* senão, activa logo o primeiro processo da lista */
    {
        inserir (listaProcessosExecutáveis, cabeça (sem.lista));
        remover (sem.lista, cabeça (sem.lista));
    }
}
Sair (trinco); /* liberta a secção do semáforo em si */

```

#### Programa 7.4 - Funções de gestão de semáforos N-ários

Note-se que:

- As funções Entrar e Sair correspondem às rotinas com o mesmo nome do Programa 7.2, em que se pode especificar como argumento o endereço da variável usada como trinco lógico. Processo é um apontador para o contexto do processo em execução.
- Há diferenças fundamentais entre um semáforo binário e um trinco lógico, apesar de ambos se destinarem a exclusão mútua (Tabela 7.2):
- O trinco lógico é um mecanismo de muito baixo nível, que usa espera activa e geralmente suporte em hardware (instrução atómica). A sua estrutura de dados resume-se a uma variável. Destina a proteger secções críticas curtas em que o tempo de espera é por regra pequeno, não compensando bloquear o processo (operação que também gasta algum tempo);
- O semáforo é um mecanismo de não tão baixo nível, implementado totalmente em software, com recurso a um trinco lógico para implementar exclusão mútua na gestão da sua estrutura de dados interna (composta por uma variável e uma lista de contextos de processos). Destina-se a bloquear processos quando for necessário impedi-los de continuar, o que evita espera activa quando o

<sup>119</sup> Este último nome expressa a analogia de assinalar ao processo que pode prosseguir (desbloquear-se) como resultado de um outro processo ter libertado a secção protegida. Na realidade, quem faz tudo (passar o processo para a lista dos processos executáveis) é esta função.

tempo de espera por parte de um processo é desconhecido e potencialmente longo.

Tanto Esperar como Assinalar incluem uma secção crítica, pois manipulam a estrutura de dados do semáforo, pelo que a sua protecção por meio de um trinco lógico é essencial. Desta forma, o acesso à secção crítica destas funções é uma espera activa, feita pelo trinco lógico, mas só à primeira vez, pois se o processo não conseguir passar no semáforo (se seu valor já for 0) fica bloqueado e não consome tempo de processador até a função Assinalar o transferir para a lista de processos executáveis.

| CARACTERÍSTICA             | TRINCO LÓGICO                        | SEMÁFORO BINÁRIO                 |
|----------------------------|--------------------------------------|----------------------------------|
| Estrutura de dados         | Variável                             | Variável + lista de processos    |
| Primitiva de implementação | Instrução de leitura/escrita atómica | Trinco lógico                    |
| Mecanismo de espera        | Espera activa                        | Bloqueio do processo             |
| Destinha-se a proteger     | De curta duração                     | De duração arbitrariamente longa |
| Secções críticas           |                                      |                                  |

Tabela 7.21 - Comparação entre as principais características dos trincos lógicos e as dos semáforos binários

Mais detalhes sobre o funcionamento dos semáforos, bem como sobre outros mecanismos de sincronização entre processos, podem encontrados em qualquer livro sobre sistemas operativos, como por exemplo [Tanenbaum 2001].

### 7.7.3.3 COMUNICAÇÃO

A transferência de dados entre processos faz-se por memória partilhada (zona de dados em que ambos os processos podem ler e escrever). Se houver memória virtual, essa zona partilhada faz-se mapeando páginas de processos diferentes na mesma página física, tal como indicado pela Fig. 7.25.

A memória partilhada pode ser simplesmente uma zona de dados não estruturada (um vector de células de memória que pode ser escrito ou lido quantas vezes se quiser) ou estar organizada por mensagens (com capacidade para várias mensagens, organizadas numa fila), em que o acto de ler uma mensagem por parte de um processo a consome, retirando-a da fila de mensagens (que suporta os conceitos de vazia e cheia). O exercício 5.14, na página 407, descreve o conceito de fila.

En qualquer dos casos, a sincronização entre processos é fundamental. Um processo não pode ler uma informação antes de outro a colocar na memória partilhada, nem durante a operação de escrita. Para esta coordenação usam-se semáforos ou outras construções de sincronização.

Esta área já diz mais respeito ao sistema operativo [Tanenbaum 2001] do que à arquitetura e não é tratada em detalhe neste livro.

## 7.7.4 PROGRAMAÇÃO COOPERATIVA

A multiprogramação (secção 7.7.2) requer um gestor de processos (que faz parte do sistema operativo) para gerir a partilha dos recursos do processador pelos vários processos. Nas aplicações mais simples, típicas dos sistemas enbebidos (secção 5.9.4.2, na página 401), nem sequer se usa um sistema operativo, que consome memória e pode ter custos de licenças de utilização (e estes sistemas são muito sensíveis ao custo), optando-se por uma gestão directa dos processos que se designa programação cooperativa.<sup>120</sup>

Não se trata realmente de multiprogramação, mas apenas de uma aproximação em que só há um programa e os "processos"<sup>120</sup> não passam de rotinas chamadas à vez por um ciclo principal, executado repetidamente. Tal como na multiprogramação, cada rotina recebe a atenção do processador de forma repetitiva, mas a grande diferença é que não há uma interrupção para mudança de processo. Cada rotina a desempenhar o papel de processo tem de ser concebida para não ter ciclos bloqueantes. De cada vez que é invocada, deve executar uma iteração do seu processamento e retornar para que o ciclo principal possa invocar os outros processos. Esta invocação repetitiva de todos os processos é que dá a ideia de multiprogramação. No entanto, se algum não retornar, o sistema bloqueia todo. Cada processo é responsável por passar o processador ao processo seguinte. Esta é a razão pela qual esta programação se designa "cooperativa".

Coloca-se o problema de manutenção do estado interno de cada processo entre invocações sucessivas, algo que na multiprogramação o despacho faz automaticamente (guardando todos os registos na pilha). Na programação cooperativa, cada processo tem de ser concebido como uma máquina de estados, guardando (antes de retornar de uma iteração) o número do estado para que deve ir na próxima iteração (quando for invocado de novo). Quando um processo (rotina) é invocado, deve começar por determinar o estado para que deve ir e fazer uma escolha múltipla baseada nesse estado, agulhando para o conjunto de instruções que trata desse estado (como uma instrução switch em C – secção 5.6.2.2, na página 302).

Este valor do estado seguinte deve ser guardado numa variável em memória e não na pilha, pois quando a rotina retorna deve deixar a pilha no mesmo estado em que estava quando foi chamada (pela mesma razão, também não é possível guardar os registos todos na pilha). Ao contrário da multiprogramação, só há uma pilha, partilhada por todos os "processos". Cada processo pode ainda ter a sua zona específica de variáveis, que contenham informação adicional que precise de ser mantida entre invocações sucessivas (contadores, por exemplo).

O Programa 7.5 ilustra a programação cooperativa, usando o mesmo exemplo que o do Programa 7.2. Desta forma, é possível contrastar os dois tipos de programação. No Programa 7.5 o primeiro estado é para a inicialização de cada processo e só é executado

<sup>120</sup> Também chamados "falsos processos".

uma vez. O processo pisca usa apenas outro estado, em que troca o valor do LED quando o contador chega ao fim, mas para ilustrar a existência de mais estados o processo Alterna, fazendo justiça ao seu nome, alterna entre dois estados em que em cada um coloca os LEDs nos valores pretendidos, em vez de trocar os seus valores.

uma vez. O processo Písca usa apenas outro estado, em que troca o valor do LED quando o contador chega ao fim, mas para ilustrar a existência de mais estados o processo Alterna, fazendo justiça ao seu nome, alterna entre dois estados em que em cada um coloca os LEDs nos valores pretendidos, em vez de trocar os seus valores.

---

|                                              |                                                                    |                                                   |
|----------------------------------------------|--------------------------------------------------------------------|---------------------------------------------------|
| PLACE                                        | 1000                                                               | i zona de dados                                   |
| pilhaAPP:                                    | TABLE 30                                                           | i pilha para o programa principal                 |
| valorPilhAPP:                                | WORD 0                                                             | i valor de inicialização do SP do prog. principal |
| estpiscas:                                   | WORD 0                                                             | i imagem na memória do valor dos LEDs             |
| estALT:                                      | WORD 0                                                             | i palavra com o estado do processo Písca          |
| contpiscas:                                  | WORD 0                                                             | i palavra com o estado do processo Alterna        |
| contalt:                                     | WORD 0                                                             | i contador de espera do processo Písca            |
| PLACE                                        | 0000H                                                              | i programa principal, inicializações, etc.        |
| início:                                      | MOV SP, fimpilhAPP ; inicializa SP com pilha do programa principal |                                                   |
|                                              | MOV R0, valorLEDs ; imagem na memória do valor dos LEDs            |                                                   |
|                                              | MOV R2, LEDS ; endereço do periférico dos LEDs                     |                                                   |
| ciclo:                                       | MOV RO, 0 ; inicializa imagem na memória do valor dos LEDs         |                                                   |
|                                              | MOV [R8], R0 ; inicializa estado do processo Písca                 |                                                   |
|                                              | MOV R1, RO ; inicializa estado do processo Alterna                 |                                                   |
|                                              | CALL Písca ; executa uma iteração do processo Písca                |                                                   |
|                                              | JMP ciclo ; repete o ciclo                                         |                                                   |
| i processo que pisca o LED 0                 |                                                                    |                                                   |
| Písca:                                       | MOV R1, estPísca ; vê em que-estado está                           |                                                   |
|                                              | MOV R0, [R1] ; se ainda não chegou a zero, vai iterar de novo      |                                                   |
|                                              | CMP R0, 0 ; se é 0, vai para o final                               |                                                   |
|                                              | JZ Piscado ; se não, continua                                      |                                                   |
|                                              | MOV R0, 1 ; se é 0, vai para o final                               |                                                   |
|                                              | JZ Piscado ; se não, continua                                      |                                                   |
|                                              | RET ; se chegar aqui, houve algum erro!                            |                                                   |
| i processo que alterna entre os LEDs         |                                                                    |                                                   |
| Alterna:                                     | MOV R2, contalt ; le contador de espera                            |                                                   |
|                                              | SUB R0, 1 ; e decrementa-o                                         |                                                   |
|                                              | MOV [R2], R0 ; actualiza contador de espera                        |                                                   |
|                                              | JNZ final ; se ainda não chegou a zero, vai iterar de novo         |                                                   |
|                                              | MOV R0, ESPERA ; se é 0, vai para o final                          |                                                   |
|                                              | CMP R0, 1 ; se não, continua                                       |                                                   |
|                                              | JZ final ; se é 0, vai para o final                                |                                                   |
|                                              | MOV R0, [R8] ; lê imagem na memória do valor dos LEDs              |                                                   |
|                                              | SET R0, 2 ; força os bits 2 e 3 (LEDs deste processo)              |                                                   |
|                                              | CLR R0, 3 ; actualiza imagem na memória do valor dos LEDs          |                                                   |
|                                              | MOV [R8], R0 ; actualiza valor dos LEDs no periférico              |                                                   |
|                                              | MOV R0, [R9] ; passa ao processo seguinte                          |                                                   |
|                                              | MOV R0, 2 ; próximo estado é 0 2                                   |                                                   |
|                                              | RET ; passa ao processo seguinte                                   |                                                   |
| i processo que reinicia o contador de espera |                                                                    |                                                   |
| Reinicia:                                    | MOV R1, estALT ; le contador de espera                             |                                                   |
|                                              | SUB R0, 1 ; e decrementa-o                                         |                                                   |
|                                              | MOV [R2], R0 ; actualiza contador de espera                        |                                                   |
|                                              | JNZ final2 ; se ainda não chegou a zero, vai iterar de novo        |                                                   |
|                                              | MOV R0, ESPERA ; reinitializa contador de espera                   |                                                   |
|                                              | MOV [R2], R0 ; reinicializa contador de espera                     |                                                   |

```

MOV    R0, [R8]      ; 1a imagem na memória do valor dos LEDs
CLR    R0, 2          ; força os bits 2 e 3 (LEDs deste processo)
SET    R0, 3          ; actualiza imagem na memória do valor dos LEDs
MOV    [R8], R0        ; actualiza valor dos LEDs no periférico
MOVB   [R9], R0
MOV    R0, 1
MOV    R1, estalt
fimaz2: MOV    R1, [RL1], R0      ; próximo estado é o 1
                                ; passa ao processo seguinte
RET

```

#### Programa 7.5 - Programação com processos em regime de programação cooperativa.

Este exemplo implementa a mesma funcionalidade que o Programa 7.2.

Deve notar-se que os ciclos de espera para implementar as temporizações dos LEDs continuam a existir, mas em vez de ser internos a cada processo têm de ser externos, isto é, passando pelo círculo principal (que deve ser o único ciclo no programa). Em vez de um processo iterar um ciclo à espera de uma condição, deve fazer uma iteração e retornar confiando no ciclo principal para o invocar de novo para a iteração seguinte.

A grande vantagem da programação cooperativa, para além de não precisar de sistemas operativo, é a garantia do processo corrente de que o processador não lhe é retirado em qualquer altura, sendo ele que controla quando tal acontece (quando ele retorna). Por este motivo, os problemas que motivaram o aparecimento dos trincos lógicos e dos semáforos na secção 7.7.3 não existem. Uma das coisas que se nota na execução deste programa, na Simulação 7.8, é que os LEDs mudam de forma sincronizada, ao contrário do que sucedeu nas simulações anteriores, o que evidencia desde logo um ambiente muito mais controlado.

A comunicação entre processos ocorre da mesma forma que na multiprogramação, por memória partilhada, mas sem problemas de sincronização.

#### SIMULAÇÃO 7.8 – PROGRAMAÇÃO COOPERATIVA

Esta simulação toma como base o Programa 7.5 e exemplifica o funcionamento da programação cooperativa. São focados os seguintes aspectos, entre outros:

- Efeito da utilização de um ciclo de espera interno a um dos processos;
- Marcação das temporizações por interrupção (temporizador) em vez de ser por contagem de iterações (variante ao Programa 7.5).

#### 7.7.5 PROTECÇÃO

Uma das áreas dos sistemas operativos que requer suporte em hardware por parte do processador é a protecção (dos processos e do próprio sistema operativo) dos efeitos eventualmente malefícios por parte de outro processo, seja por erro de programação ou intencionalmente (caso dos vírus, por exemplo).

Se um processo puder mudar dados de outro processo ou mesmo do sistema operativo pode facilmente baralhar o computador todo, com perda de serviço e/ou informação ou

os seus utilizadores, ou ter acesso indevidamente a informação sensível. A segurança na área da informática é dos tópicos mais importantes actualmente, pelo que a protecção básica, restringindo as capacidades de um processo exclusivamente às que deviam ser, é algo fundamental em qualquer computador. O hardware dá o suporte, mas o sistema operativo é que é responsável por o usar e gerir os processos de forma protegida.

Cada processador tem as suas especificidades, mas tipicamente os processadores suportam as seguintes características, que o PEPE também suporta:

- Dois níveis de privilégio, vulgarmente designados Sistema e Utilizador;
  - Instruções e recursos do processador (registos, por exemplo) de nível de Sistema;
  - Pilhas separadas para nível de Sistema e Utilizador (dois registos SP);
  - Descritor de nível de privilégio nas páginas de memória virtual.
- O RE (Tabela A.2, na página 697) inclui o bit NP (Nível de Privilégio), que se estiver a 0 indica que se está no nível de Sistema e se estiver a 1 no nível de Utilizador. Por "utilizador" entende-se software que usa o processador mas não serve para a sua gestão (pode ser um compilador, um processador de texto, etc.), o software de nível de Sistema faz parte do sistema operativo e destina-se a gerir o computador.<sup>121</sup> A diferença entre os dois é que o nível de Sistema permite aceder a todos os recursos e funcionalidades, ao passo que ao nível de Utilizador estão barradas algumas dessas características. Se uma rotina de nível Utilizador tentar aceder a algo a que não tem direito é gerada a exceção SISTEMA (Tabela A.8, na página 702), que permite ao sistema operativo gerar um erro e eventualmente terminar o processo correspondente.

As instruções e recursos do PEPE acessíveis apenas ao nível de Sistema são os seguintes:

- Instrução MOV (página 705) envolvendo registos auxiliares (secção A.2.2, na página 697), quer a ler quer a escrever esses registos;
- Alteração do registo RE (Tabela A.2, na página 697), com exceção dos bits de estado (Z, N, C e V). Qualquer alteração dos outros bits, seja por que instrução for (há muitas maneiras de alterar o RE), gera a exceção SISTEMA;
- Instrução SWI (secção 6.2.3.1, na página 478) com número de exceção inferior a 32. Isto quer dizer que as exceções com número até 31 são ou predefinidas (Tabela A.8, na página 702) ou reservadas para o nível de Sistema. As exceções 32 a 255 podem ser invocadas a partir do nível de Utilizador e são reservadas para invocação de rotinas do sistema operativo.

A seguir a uma inicialização (reset) do PEPE, o bit NP está a 0, o que se significa que arranca em nível de Sistema. Quando o sistema operativo cria um processo, pode fazê-lo

<sup>121</sup> Apenas as camadas de mais baixo nível do sistema operativo (as que lidam com os recursos mais críticos) são de nível de Sistema. As partes de mais alto nível (tipicamente as que lidam menos com recursos críticos) são também de nível de Utilizador.

em nível de Utilizador, bastando colocar este bit a 1 na imagem do RE guardada na pilha desse processo (constante `IMAGEMRE` no Programa 7.2).

Naturalmente, um processo em nível de Utilizador não pode mudar o bit NP directamente. No entanto, a ocorrência de uma exceção faz com que o bit NP seja colocado a 0, sendo reposto a 1 pela instrução de retorno (RE), tal como sucede com o bit de permissão das interrupções (IE). Esta mudança para nível de Sistema numa exceção justifica-se pelo facto de as exceções serem um mecanismo normalmente usado para resolver problemas de temporizações ou acesso a periféricos, tarefas do sistema operativo e que não devem estar acessíveis ao nível de utilizador, tanto mais que todos os processos partilham os mesmos recursos físicos (periféricos, nomeadamente) e essa partilha tem de ser feita de modo controlado.

Este facto permite a uma rotina de nível de Utilizador invocar uma funcionalidade do sistema operativo que exija nível de Sistema sem comprometer a protecção de todo o sistema. Tal é feito não com uma simples instrução CALL mas sim com SWE, com um número de exceção entre 32 e 255. Esta chamada indirecta (chamada ao sistema, ou `system call`) tem duas grandes vantagens:

- A rotina de nível de Utilizador não precisa de saber qual o endereço da rotina de Sistema invocada (basta o seu número), o que permite compilar e recompilar o sistema operativo (novas versões, ou releases) sem necessidade de alterar os programas de nível de Utilizador;
- A invocação de rotinas do sistema operativo é feita de modo controlado e protegido. É o sistema operativo que controla a tabela de exceções e que rotinas esta permite invocar (naturalmente, tem de preencher todas as entradas, mesmo que não use todas as exceções possíveis).

Um processo criado com nível de Utilizador pode assim ser executado em nível de Sistema, quando invoca rotinas de nível Sistema. Também acontece que o sistema operativo precisa de invocar rotinas de nível de utilizador (normalmente designadas callbacks), o que poderia permitir a essas rotinas alterar a imagem do RE guardada na pilha, nomeadamente o bit NP (caso em que a rotina de Sistema poderia retornar para a rotina de Utilizador mas com o bit np igual a 0, em nível de Sistema), constituindo um grave erro em termos de protecção.

Para resolver este problema, a pilha de nível de Utilizador não é a mesma que a de nível de Sistema e o registo SP não é apenas um mas dois, USP (SP de nível de Utilizador) e SSP (SP de nível de Sistema), com selecção automática pelo bit NP. As instruções que manipulam o SP (CALL, RET, PUSH, POP, etc.) usam o USP ou SSP de acordo com o nível correto, indicado pelo bit NP. Na implementação da instrução SWE (página 707) e nas microinstruções que lançam o atendimento de uma exceção (Tabela 7.8, na página 591), que podem mudar de nível de Utilizador para o nível de Sistema, pode constatar-se que antes de usar a pilha o bit NP é colocado a 0, sendo o valor anterior de RE guardado provisoriamente no registo TEMP. Pelo contrário, a última microinstrução da instrução RE (página 707) é repor o valor do RE. Isto permite guardar a imagem do RE já na pilha de

Sistema, mesmo quando a exceção ocorre durante a execução de uma rotina em nível de Utilizador.

Poderá ser necessário copiar dados de uma pilha para a outra, nomeadamente quando uma rotina de um nível invoca uma do outro nível, com parâmetros passados pela pilha. Nesse caso, os dados têm de ser copiados de uma pilha para a outra. As instruções MOV RD, USP e MOV USP, RS (página 705) permitem aceder ao USP a partir do nível de Sistema (em que SP se refere a SSP). Estas instruções não estão vedadas ao nível de Utilizador porque o USP é um recurso de nível de Utilizador (nessa altura, é o mesmo que SP).

Mesmo com todos estes cuidados, nada impedia a rotinas e dados de nível de Sistema, desde que se soubesse o endereço. Isto permitiria ter acesso às tabelas do sistema operativo e periféricos, com todas as consequências daí resultantes. Por esta razão, restringe-se a gama de endereços a que uma rotina de nível de Utilizador tem acesso, usando o mecanismo da memória virtual. Cada página tem um descritor na tabela de páginas que a descreve (Tabela 7.19). O bit P nesse descritor permite indicar se a página é de nível de Sistema ou de Utilizador. Qualquer acesso (leitura ou escrita) à memória com P=0 (página de Sistema) com NP=1 (nível de Utilizador) gera uma exceção. D PROT se for com MOV e L PROT se for numa busca de instrução (Tabela A.8, na página 702). Os periféricos, recursos críticos e partilhado por todos os processos, devem ser geridos pelo sistema operativo e mapeados em páginas virtuais de nível de Sistema.

Finalmente, mesmo dentro do nível de Utilizador é possível declarar certas páginas (de instruções e de dados que não devem ser alterados) como só de leitura, usando o bit W do descritor da Tabela 7.19. Qualquer acesso em escrita com W=0 é uma violação desta protecção e gera a exceção SO\_LERTRRA (Tabela A.8).

### SIMULAÇÃO – PROTECÇÃO

Esta simulação exemplifica as técnicas de protecção descritas nesta secção, com pequenos exemplos que exercitam os vários aspectos referidos, nomeadamente:

- Níveis de privilégio e utilização do bit NP;
- Verificação da conutação do SP com o valor de NP;
- Geração de uma exceção SISTEMA com execução de acções não permitidas ao nível de Utilizador;
- Implementação (com microprogramação) de uma instrução de nível de Sistema; manipulam o SP (CALL, RET, PUSH, POP, etc.) usam o USP ou SSP de acordo com o nível correto, indicado pelo bit NP. Na implementação da instrução SWE (página 707) e nas microinstruções que lançam o atendimento de uma exceção (Tabela 7.8, na página 591), que podem mudar de nível de Utilizador para o nível de Sistema, pode constatar-se que antes de usar a pilha o bit NP é colocado a 0, sendo o valor anterior de RE guardado provisoriamente no registo TEMP. Pelo contrário, a última microinstrução da instrução RE (página 707) é repor o valor do RE. Isto permite guardar a imagem do RE já na pilha de

<sup>2</sup>Nos exemplos anteriores, nomeadamente o Programa 7.2, são os próprios processos da aplicação que lidam directamente com os periféricos, misturando-se partes do programa que são independentes do computador com outras que são específicas e com exigência de

nível de protecção diferentes (as rotinas de nível de utilizador não devem ter acesso directo e incondicional aos periféricos).

Os sistemas operativos estão estruturados em camadas, em que apenas as de mais baixo nível estão ligadas ao hardware. As restantes, de nível superior, não só não accedem directamente ao hardware como são independentes deste. O sistema operativo fornece uma abstracção do hardware que possibilita o seu uso e gestão coerente por parte de todos os programas. Deste modo, consegue-se não apenas estruturar melhor o programa (facilitando o seu desenvolvimento e manutenção) como também implementar a protecção necessária em qualquer sistema e aumentar a sua portabilidade (entre computadores diferentes).

Assim, a interacção com os periféricos é feita pelos chamados gestores de periféricos (*device drivers*), que podem tomar duas formas essenciais:

- Rotina de nível de sistema, que é invocada pelas rotinas da aplicação (normalmente, de nível de utilizador) por chamada ao sistema (*system call*), gerando uma exceção em software (SW). Isto pode implicar mudança de nível de protecção e cópia de informação entre as pilhas de sistema e de utilizador. A rotina do sistema só regressa quando a operação foi efectuada e deve incluir sincronização (exclusão mutua, tipicamente, implementada por um semáforo) para impedir alterações descoordinadas à estrutura de dados associada ao periférico acedido por parte dos vários processos existentes. No Programa 7.3, isto corresponde a substituir o MVBs (de actualização do valor dos LEDs no periférico) por uma chamada ao sistema, com SW. Neste exemplo em particular a chamada ao sistema já beneficiaria da exclusão mutua fornecida pela variável trinco, mas no caso geral um gestor de periférico não pode depender das rotinas que o invocam para garantir a exclusão mutua, pelo que deve incluir o seu próprio semáforo;

Processo independente. Neste caso, um processo que pretenda interactivar com o periférico deve fazer o pedido ao gestor de periférico enviando-lhe uma mensagem (dados colocados numa zona partilhada e coordenação efectuada por semáforos). A resposta (no caso de leitura de dados de um disco, por exemplo) poderá ser dada também por mensagem. Este esquema é mais flexível que o anterior, permitindo nomeadamente que o processo que fez o pedido prossiga o seu processamento (enquanto o gestor do periférico executa esse pedido de forma automática e independente) até precisar mesmo dos dados, mas é mais exigente em termos de desempenho de todo o sistema (há mais operações envolvidas), o que impede que seja adoptado para as interacções mais frequentes e com mais requisitos de rapidez.

Os gestores de periféricos é tudo o que os suporta são mecanismos fundamentais em qualquer computador, mas já começam a sair do âmbito da arquitetura de computadores. Este tema (e todo o suporte para processos) é incluído aqui para fazer a ligação aos sistemas operativos, que são detalhados pelos livros da especialidade [Tanenbaum 2001].

#### ESSENCIAL

- Os processadores têm, normalmente, suporte para processos (programas em execução "simultânea", em regime de multiprogramação). A mudança de processo envolve guardar todo o contexto (registos) do processo em execução em memória e carregar nos registos o contexto previamente guardado do processo que vai ser executado a seguir (é que portanto retorna a execução no ponto em que tinha sido interrompido);

Um processo nunca sabe em que ponto sai de execução. Quando vários processos usam um mesmo recurso, têm de garantir que não são interrompidos a meio de uma secção crítica, cujo acesso tem de ser mutuamente exclusivo. Isto é conseguido com trincos e semáforos, primitivas de sincronização que se baseiam numa instrução atómica (*Test & Set*; *SWAP*, etc.), que efectua uma leitura e uma escrita numa sequência indivisível.

- O suporte para processos inclui também protecção que tipicamente envolve dois níveis de privilégio: em que o sistema operativo (ou pelo menos o seu núcleo) executa no nível mais privilegiado.

- Mesmo sem suporte em hardware para multiprogramação, é possível simular processos com rotinas que são chamadas à vez em ciclo pelo programa principal (processos cooperativos). Cada uma destas rotinas executa uma iteração e retorna, não podendo bloquear-se (senão as outras não-corriam). Antes de retornar, tem de guardar em memória o estado em que estava, para na próxima invocação poder continuar o processamento. Neste caso, não há necessidade de exclusão mutua porque cada rotina é que controla quando retorna.

## 7.8 CONCLUSÕES

Un computador é um sistema muito complexo, sendo capaz de realizar operações muito importantes para sociedade humana. Esta complexidade está em boa parte no software, a parte (re)programmável sem alterações físicas, mas o hardware é fundamental no suporte à funcionalidade, garantia de bom funcionamento e desempenho.

Este capítulo afiou apenas as características básicas do hardware de um processador e do software de mais baixo nível, nomeadamente do sistema operativo. Os computadores e os sistemas operativos actuais são muito mais complexos e incluem muito mais técnicas do que as descritas neste livro de nível introdutório. No entanto, são as que permitem compreender o funcionamento básico de qualquer computador, por serem as que estão presentes em todos os computadores.

O PEPE foi concebido exactamente nesta perspectiva. Tal como sabendo conduzir um automóvel de uma marca se consegue conduzir outro de qualquer marca (diferem nos detalhes mas na sua essência são idênticos), conhecendo as características e funciona-

mento do PEPE tem-se a ferramenta fundamental para compreender qualquer outro. Este é um esforço necessário sejam quais forem os computadores/processadores que se conhecam, pois todos acabam por diferir nas suas capacidades e soluções de detalhe encontradas para as implementar.

A arquitectura básica, derivada do modelo original de von Neumann, é quase universal. Memória, registo, ALU, PC e unidade de controlo são conceitos fundamentais que perduraram até hoje e que fazem parte da arquitectura de qualquer computador, embora haja diferenças significativas nos detalhes de implementação. Nem todos os computadores usam microprogramação, usando apenas instruções que se executam numa só operação.

O desempenho é o *Santo Graal* dos computadores. À medida que a tecnologia evolui e os computadores ficam mais rápidos e com mais capacidade de memória, os programas ficam também mais complexos e exigem mais desempenho. É uma espiral de inflação informática que parece não ter fim (bem pelo contrário, cada vez está mais rápida). O processamento em estágios é uma das contribuições mais importantes para o aumento do desempenho, mas muitas outras técnicas, nem sequer referidas aqui por limitações de espaço, fazem parte de qualquer dos processadores de alto desempenho (secção 6.5.2, na página 520) existentes no mercado actual.

A memória não tem conseguido acompanhar os aumentos sucessivos da frequência do relógio e do desempenho proporcionado pelas evoluções arquitecturais dos processadores, pois também tem aumentado drasticamente a sua capacidade, razão pela qual as caches internas dos processadores têm aumentado em capacidade e importância. A memória virtual não contribui para o desempenho (antes pelo contrário), mas é fundamental para a funcionalidade e correção dos programas de funcionalidade, abstraindo muitas das limitações físicas da memória.

O sistema operativo, que orquestra todo o computador, é essencialmente software, mas no seu âmbito envolve uma estreita cooperação com o hardware e o suporte que este proporciona. Noutros tempos, um fabricante fazia tudo, desde o hardware até ao sistema operativo, compiladores e aplicações de maior utilização. Hoje em dia, a especialização de cada empresa numa dada área é incontornável, pelo que as normas, a portabilidade e interoperacionalidade são aspectos de importância universal. Uma das características que garantiu o sucesso do Unix (juntamente com o Windows, um dos sistemas operativos que dominam o mundo) é a sua portabilidade e capacidade de correr em praticamente qualquer arquitectura, precisamente por limitar as suas exigências de suporte às técnicas mais básicas de suporte e quase universalmente disponíveis.

É provável que no futuro os computadores sejam muito diferentes dos actuais. A experiência passada diz-nos que a inteligência e criatividade humanas parecem não ter limites à vista. No entanto, o facto é que as evoluções tecnológicas e arquitecturais têm conseguido vencer (em desempenho) todas as tentativas de revolução em relação ao modelo computacional clássico de von Neumann, pelo que a matéria tratada neste livro é a fundamental em termos de computadores e da forma como estes funcionam.

## 7.9 EXERCÍCIOS

**7.1** Justifique a existência do REM (Fig. 7.2, na página 569). Pista: considere os vários modos de endereçamento do PEPE.

**7.2** Indique se a instrução CMP R1, -3 recorre ou não à extensão de bits da constante e, em caso, positivo, qual dos métodos da Tabela 7.2, na página 575, deverá ser usado. Justifique a resposta.

**7.3** Alguns processadores usam uma palavra extra além da instrução propriamente dita (opcode, operandos em registos, modo de endereçamento, etc.) para especificar um operando constante. No PEPE, todas as instruções têm uma e só uma palavra. Justifique a existência do gerador de constantes (Fig. 7.2, na página 569) à luz desta afirmação.

**7.4** Considere uma unidade de controlo microprogramada, como a da Fig. 7.7, na página 586, que controla um circuito com 11 sinais de 1 bit e 2 sinais de 3 bits. Suponha que a ROM de microcódigo tem 30 bits de largura e o multiplexer que controla os saltos no microcódigo tem 8 entradas.

- Divida os 30 bits da palavra da ROM de microcódigo em grupos e indique o número de bits e a funcionalidade de cada grupo;
- Quantos bits deve ter o MPC (*Micro Program Counter*)?
- Que capacidade (em palavras) deve ter a ROM de microcódigo?
- Quantas condições de salto (para os saltos condicionais) são suportadas por esta unidade de controlo?
- Suponha que quer suportar mais três condições de salto. Que alterações tem de fazer a esta unidade de controlo?
- Indique que tipo de alterações no circuito controlado por esta unidade de controlo podem exigir aumentar:

- i) A largura da ROM de microcódigo sem alterar a sua capacidade;
- ii) A capacidade da ROM de microcódigo sem alterar a sua largura.

**7.5** Considere o circuito da Fig. 7.35, em que R1, R2 e R3 são registos de 16 bits e se pretende implementar a divisão inteira  $X/Y$ . O algoritmo consiste simplesmente em ir subtraindo Y a X até o resto ser inferior a Y. O número de subtrações efectuadas será o quociente inteiro. Os bits Z e N ficam a 1 se  $R1-R2$  for zero ou negativo, respectivamente. LOAD\_R1 e LOAD\_R2 permitem carregar X e Y em R1 e R2, respectivamente. INIT\_R3, INCR\_R3 e LOAD\_R3 implementam a inicialização a 0, o incremento e o carregamento em R3, respectivamente.

- Construa e preencha uma tabela semelhante à Tabela 7.6, na página 587, com toda a informação necessária para a determinação do conteúdo da ROM de microprograma para implementar a divisão inteira;
- Idem*, mas para implementar o resto da divisão inteira;

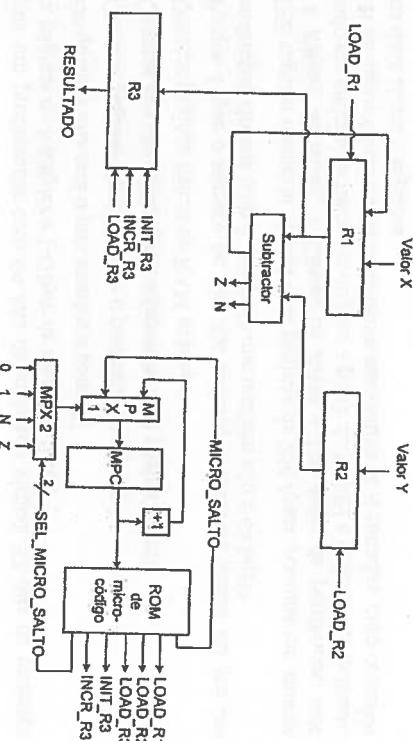


Fig. 7.35 - Divisor microprogramado

**7.6 Utilize o simulador do PEPE na parte de microcódigo para implementar novas instruções para o PEPE, tal como já foi feito na Simulação 7.2, na página 594.**

Para cada uma das instruções seguintes defina no assembler (página 594), determine uma instrução máquina adequada (Fig. 7.8, na página 592), implemente-a (indicando as microinstruções necessárias e os valores dos sinais de controlo) e teste a sua funcionalidade. Tenha atenção à gama de valores possíveis para a constante n.

- Fill [Rd], Rs, n (preenche com o valor que estiver no registo Rs as n palavras de memória que começam no endereço dado pelo registo Rd);
- SHRM [Rd], n (desloca de n bits para a direita a palavra de memória cujo endereço é indicado pelo registo Rd);
- MovSTR [Rd], [Rs], Rn (cópia uma cadeia de bytes, em número indicado pelo registo Rd, a partir do endereço dado pelo registo Rd, para a zona de memória que começa no endereço dado pelo registo Rd);
- SUM Rd, [Rs], Rn (soma todas as palavras a partir do endereço indicado pelo registo Rs e em número indicado pelo registo Rn, ficando o resultado no registo Rd).

**7.7 Num processador com cadeia de estágios, porque é que estes são projectados para executarem aproximadamente no mesmo tempo?**

- 7.8 A execução de um dado programa com 1 milhão de instruções executadas demora 5 minutos num processador sem cadeia de estágios e 2 minutos num processador com cadeia de estágios. Se se modificar o programa para de 100 em 100 instruções escrever num periférico, provocando um pedido de interrupção (cuja rotina corresponde a 20 instruções executadas), a proporção dos tempos de execução nos**

dois processadores manter-se-á ou um é mais afectado do que o outro? Justifique a resposta.

**7.9 Considere um processador com um relógio de 1 GHz, sem cadeia de estágios e que demora, em média, 4 ciclos de relógio a executar uma instrução. Se o processador executar 10 milhões de instruções:**

- Qual a duração do ciclo de relógio?
- Quanto tempo demorará a executar o programa?
- Quantos MIPS consegue o processador fazer?

**7.10 Considere que no processador do exercício 7.9 se introduziu uma cadeia de 5 estágios.**

- Quais as alterações fundamentais a efectuar para que tal seja possível?
  - A frequência de relógio (assumindo a mesma tecnologia de fabrico) deve de descer para 800 MHz. Porquê?
  - Quanto tempo demora agora, em média, a execução de uma instrução?
  - Que melhoria se deverá esperar no tempo de execução do programa? Quanto tempo demorará agora o programa a executar? Justifique;
  - Quantos MIPS faz agora o processador? Explique a diferença em relação à resposta da alínea c) do exercício 7.9 e face à resposta da alínea c) deste exercício;
  - Suponha agora que o programa tem cerca de 20% de instruções de salto executadas que efectivamente saltam. Que variação no tempo de execução será esperável face a não haver saltos? (Pista: use a Fig. 7.13, na página 615, para ter uma ideia do impacte de um salto);
  - Suponha ainda que, para além dos saltos, 10% das instruções executadas têm dependências de dados que obrigam a introduzir uma bolha. Nestas condições, quanto tempo demorará o programa a correr e que melhoria terá sido conseguida face ao tempo de execução no exercício 7.9?
- 7.11 Num dado computador, a memória principal tem 8 K blocos (de 256 bytes cada), sendo endereçável ao byte, em que cada bloco é a unidade mínima de transferência entre a memória principal e a cache.**
- De quantos bits será o processador? Justifique a resposta;
  - Qual a capacidade da memória? Qual o valor máximo para esta capacidade, com este processador?
  - Se aumentássemos a capacidade da memória era obrigatório, conveniente ou indiferente aumentar a capacidade da cache? Porque?
  - Nos computadores actuais, as memórias estão optimizadas para acessos de várias palavras seguidas e não para acessos a células individuais. Porque?
  - Supondo que este computador tem uma cache associativa por conjuntos com 4 vias e uma capacidade de 32 conjuntos, indique qual o formato dos endereços.

ços (campos com diferentes significados em que o controlo da cache divide o endereço);

f) Idem, mas supondo que a cache é de mapeamento directo (mantendo a capacidade da cache);

g) Identifique (de forma precisa) nas caches das alíneas anteriores o local ou locais onde a palavra com o endereço F7DCH (eventualmente estendido com zeros à esquerda) será localizada;

h) Indique o menor e o maior endereço das palavras que são transferidas juntamente com a palavra no endereço indicado na alínea anterior.

### 7.12 Imagine que o PEPE executa as seguintes instruções:

```
MOV    R0, 00AH ; valor para configurar cache de dados
MOV    R0C0, R0 ; configura cache de dados
```

- Qual a capacidade da cache de dados?
- Qual o formato dos endereços, em termos de campos relevantes para o funcionamento da cache de dados?

### 7.13 Um dado computador tem uma cache L1 (com blocos de 4 palavras) integrada no processador e uma cache L2 externa (com blocos de 16 palavras), que liga à memória principal, cujos tempos de acesso são, em média 1 ns, 10 ns e 60 ns, respectivamente. Estes dois últimos tempos referem-se apenas ao primeiro acesso (os acessos às palavras seguintes, em rajada, são 2 ns e 10 ns, respectivamente). A taxa de sucesso média das caches é de 85% e 95%. Assumindo que em caso de falha no acesso a uma cache o tempo total de acesso é o tempo de um acesso com sucesso (gasto apenas para se saber que falhou) seguido de um acesso ao nível seguinte e da repetição do acesso original, calcule o tempo de acesso médio sentido pelo núcleo do processador.

### 7.14 Imagine um processador com 64 bits de endereço virtual, 32 bits de endereço físico, páginas de 8 KBytes e 1 GByte de memória física.

- De quantos bits será o processador e porque?
- Quantos páginas virtuais é que o processador suporta?
- Quantos bits são usados para referenciar uma página física?
- Quantas páginas físicas de RAM existem?
- Porque é que o número de bits dos endereços físicos é menor do que dos endereços virtuais (isto é, porque é que também não é 64 bits)?

7.15 O PEPE suporta memória virtual com páginas de 256 bytes. Assuma um sistema com 8 KBytes de ROM, 4 KBytes de RAM e 128 bytes de periféricos, tudo mapeado em endereços físicos contíguos a partir de 0000H e pela ordem indicada. A TLB de dados é uma cache totalmente associativa de 4 entradas, cujo conteúdo é numa dada altura o seguinte:

| VÁLIDA | ALTERADA | NA PÁGINA VIRTUAL | NA PÁGINA FÍSICA |
|--------|----------|-------------------|------------------|
| 1      | 1        | 24H               | 26H              |
| 1      | 0        | 7CH               | 2EH              |
| 1      | 1        | 64H               | 30H              |
| 0      | 0        | E8H               | 0AH              |

| ENDERECO VIRTUAL | ENDERECO FÍSICO |
|------------------|-----------------|
| 6426H            | 3064H           |
| 240AH            |                 |
|                  | 267CH           |

- Indique as gamas de endereços do espaço de endereçamento virtual que originariam uma falta de página se fosse feito um acesso a um desses endereços;
  - Partindo de uma TLB vazia, indique uma das possíveis frequências de acesso à memória efectuados pelo programa (a que endereços, indicando se é de leitura ou de escrita) que levam a este conteúdo da TLB. Estes endereços são físicos ou virtuais?
  - Os acessos referidos na alínea anterior são de dados, de código (busca de instruções) ou podem ser de ambos os tipos?
  - A que dispositivo físico se refere cada uma das entradas na TLB?
  - Suponha agora que, com a TLB no estado indicado, o processador executa a instrução MOV R1, [R2], em que R2=E850H. Indique que acções (relevantes para a memória virtual) ocorrem durante este acesso e se este provoca alguma alteração na TLB, explicando porquê. Pode arbitrar todos os aspectos que não tiverem solução única.
- 7.16 Num dado processador, verificou-se que um acesso à cache demora 5 ns e a taxa de sucesso média nestes acessos é de 95%. O acesso à memória principal demora 50 ns, mas em cerca de 1% dos acessos à memória principal, em média, ocorre uma falta de página, cuja recuperação (desde a falta até a página estar disponível na memória principal) implica um tempo entre 6 e 12 ms, dependendo da posição da cabeça do disco face à zona em que a página se encontra, com um valor médio

na ordem dos 8 ms. Quando um acesso falha num nível da hierarquia de memória é feito um acesso no nível seguinte e depois repete-se o acesso desde o princípio.

- Quanto tempo demora o acesso mais longo (situação mais desfavorável)?
- Qual o tempo médio de acesso à memória física (visto pelo núcleo do processador)?
- Qual o tempo médio de acesso à memória virtual (visto pelo núcleo do processador)?
- Com base nas respostas anteriores, mostre que o mecanismo de memória virtual piora o desempenho de um computador. Então porque é que os computadores o suportam?

**7.17** Imagine que dois processos estão a cooperar no incremento de uma dada variável em memória. O processo Par testa continuamente a variável até descobrir que esta tem um valor ímpar, altura em que incrementa o seu valor e repete o ciclo. O processo Impar faz a acção complementar (espera até ser par e depois incrementa a variável, repetindo depois o ciclo).

- Programe as rotinas dos processos em regime de programação cooperativa;
- Idem*, mas em regime de multitprogramação. Precisa de usar sincronização ou não? Porque?
- Suponha, simplificando, que cada teste à variável (incremente-a ou não, seja num regime ou outro) demora sempre o mesmo tempo e que em cada fatia de tempo são executados 10 testes de um dado processo. Ignorando o tempo gasto fora das iterações (mudança do contexto do processo, por exemplo), calcule aproximadamente o número de iterações que o programa executa até a variável chegar ao valor 1000, para os casos das alíneas a) e b);
- Com base na resposta da alínea c) é ou não possível afirmar que um dos regimes é mais eficiente que o outro? Porque?

**7.18** Se se pudesse alterar o conteúdo da tabela de exceções a partir de um programa, no nível de Utilizador, o sistema não teria uma protecção eficaz. Explique porque é que é que o mecanismo de memória virtual pode ajudar.

**7.19** Os gestores de periféricos têm uma interface bem determinada, para que o sistema operativo não tenha de ser recompilado sempre que se muda esse gestor. Mas aqui vamos simplificar e admitir que o gestor é implementado por uma simples chamada ao sistema operativo. Programe um gestor de periférico que leia um byte de byte novo disponível. Programe uma rotina que apenas invoque o gestor de periférico, que só deve retornar quando houver um novo byte, e affixe esse carácter em dois mostradores de sete segmentos. Simule tudo no simulador. Para fazer entrar o byte use um módulo de entrada e um interruptor para gerar a interrupção para o PEPE. A rotina deve correr em nível de Utilizador e a chamada ao sistema deve ser feita com SWI.

## MANUAL DE PROGRAMADOR DO PEPE

### APÊNDICE A

O PEPE (Processador Especial Para Ensino) é um microprocessador de 16 bits, concebido com fins pedagógicos para a área de arquitectura de computadores e que serve de base a este livro. É descrito em detalhe ao longo dos vários capítulos. Este apêndice sistematiza e completa as suas principais características, em jeito de manual de referência.

O simulador descrito sucintamente no Apêndice C, na página 725, contém um módulo que implementa do PEPE com um pequeno manual *on-line*. O site de apoio a este livro ([www.fcapt.pt](http://www.fcapt.pt)) contém o software do simulador e a documentação mais recente sobre o PEPE, que poderá ter evoluído face à incluída neste livro. As diferenças eventualmente introduzidas estão referidas nas novas versões da documentação.

#### A.1 PINOS DO MÓDULO PEPE

A Tabela A.1 descreve os pinos do módulo PEPE existente no simulador descrito no Apêndice C. O conhecimento das suas características é fundamental para construir circuitos como, por exemplo, o da Fig. 6.4, na página 420. Como em qualquer sistema real, os pinos de entrada não usados devem ser ligados a 0 ou 1 e não deixados "no ar". O pino RESET destina-se a inicializar o módulo e tipicamente liga-se a um módulo específico, que inicializa o PEPE no início de cada simulação.

Os pinos INT0 a INT3 permitem gerar interrupções, com o funcionamento descrito na secção 6.2.2, na página 462.

O pino CLOCK permite ligar o PEPE a um relógio externo, sob controlo do RCN (Registo de Configuração do Núcleo), descrito na Tabela A.4. Por omissão, o sinal de relógio é interno ao PEPE.

O barramento de dados (D15..D0), o barramento de endereços (A15..A0) e os sinais RD (Read), WR (Write), BA (Byte Addressing), WAIT e IC são os pinos que suportam a ligação à memória e periféricos. O seu funcionamento detalhado está descrito na secção 6.1, a partir da página 410. O pino RC está descrito na secção 7.5.7, na página 642.

Os sinais BRQ (Bus Request) e BGR (Bus Grant) suportam as operações de acesso directo à memória (DMA), de acordo com a descrição feita na secção 6.4.2.3, na página 511.

| PINO    | N.º DE BITS | TIPO              | FUNCIONALIDADE                                                                                                                                       |
|---------|-------------|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| RESET   | 1           | Entrada           | Initialização (Reset) (activo a 1)                                                                                                                   |
| INT0    | 1           | Entrada           | Interrupção 0 (activação programável a 0, 1 ou um dos flancos)                                                                                       |
| INT1    | 1           | Entrada           | Interrupção 1 (activação programável a 0, 1 ou um dos flancos)                                                                                       |
| INT2    | 1           | Entrada           | Interrupção 2 (activação programável a 0, 1 ou um dos flancos)                                                                                       |
| INT3    | 1           | Entrada           | Interrupção 3 (activação programável a 0, 1 ou um dos flancos)                                                                                       |
| CLOCK   | 1           | Entrada           | Entrada de relógio externo do processador                                                                                                            |
| D15..A0 | 16          | Entrada/<br>Saída | Barramento de dados                                                                                                                                  |
| A15..A0 | 16          | Saída             | Barramento de endereços                                                                                                                              |
| BA      | 1           | Saída             | (Byte Addressing) Endereçamento de byte<br>BA=1 – acessos à memória em byte<br>BA=0 – acessos à memória em palavra                                   |
| RD      | 1           | Saída             | Activo a 0 nos ciclos de leitura à memória                                                                                                           |
| WR      | 1           | Saída             | Activo no flanco 0 para 1 nos ciclos de escrita na memória                                                                                           |
| WAIT    | 1           | Entrada           | WATT=1 – prolonga o ciclo de acesso à memória com duração mínima<br>WATT=0 – ciclo de acesso à memória com duração mínima                            |
| IC      | 1           | Entrada           | (Ignore Cache) Ignorar a cache nos acessos à memória<br>IC=1 – acessos devem ser directos sem passar pela cache<br>IC=0 – acessos devem usar a cache |
| BRQ     | 1           | Entrada           | (Bus Request) Pedido de DMA, activo a 1                                                                                                              |
| BGT     | 1           | Saída             | (Bus Grant) Autorização para DMA, activo a 1                                                                                                         |

Tabela A.1 - Pinos do módulo PEPE

| BIT | SIGLA | NOME E DESCRIÇÃO                                                                                                                                                                                      |
|-----|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0   | Z     | Zero – Este bit é colocado a 1 pelas operações da ALU que produzem zero como resultado                                                                                                                |
| 1   | N     | Negativo – Este bit é colocado a 1 pelas operações da ALU que produzem um número negativo (bit de maior peso a 1) como resultado                                                                      |
| 2   | C     | Transporte (Carry) – Este bit é colocado a 1 pelas operações da ALU que geram transporte                                                                                                              |
| 3   | V     | Excesso (Overflow) – Este bit é colocado a 1 pelas operações da ALU cujo resultado é demasiado grande (em módulo) para ser representado correctamente, seja positivo ou negativo                      |
| 4   | A     | Reservado para utilização futura                                                                                                                                                                      |
| 5   | B     | Reservado para utilização futura                                                                                                                                                                      |
| 6   | TV    | Exceção em caso de excesso (Trap on overflow) – Se este bit estiver a 1, é gerada a exceção que produz o excesso. Se estiver a 0, o excesso só actualiza o bit V                                      |
| 7   | TD    | Exceção em caso de divisão por 0 (Trap on DIV0) – Se este bit estiver a 1, é gerada a exceção DIV0 numa instrução DIV ou MOD com quociente 0 (não é gerada a exceção EXCESSO nem o bit V é posto a 1) |
| 8   | IE    | Permissão de Interrupções Externas (Interrupt Enable) – Só com este bit a 1 as Interrupções externas poderão ser atendidas                                                                            |
| 9   | IE0   | Permissão da Interrupção Externa 0 (Interrupt Enable 0) – Só com este bit a 1 os pedidos de Interrupção no pino INT0 poderão ser atendidos                                                            |
| 10  | IE1   | Idem, para a Interrupção INT1                                                                                                                                                                         |
| 11  | IE2   | Idem, para a Interrupção INT2                                                                                                                                                                         |
| 12  | IE3   | Idem, para a Interrupção INT3                                                                                                                                                                         |
| 13  | DE    | Permissão de acessos directos à memória (DMA Enable) – Só com este bit a 1 os pedidos de DMA no pino BRQ serão tidos em conta e eventualmente atendidos pelo processador                              |
| 14  | NP    | Nível de Proteção<br>0=Sistema; 1=Utilizador. Define o nível de proteção corrente                                                                                                                     |
| 15  | R     | Reservado para utilização futura                                                                                                                                                                      |

Tabela A.2 - Descrição dos bits do RE (Registo de Estado)

## A.2 REGISTOS PRINCIPAIS

São os registos do banco de registos acessíveis à generalidade das instruções, sendo referidos pelas siglas R0 a R15. Os registos R11 a R15 têm funcionalidades particulares e têm designações específicas (BL, SR, RE, BTE e TEMP, respectivamente). A Fig. 4.5, na página 204, descreve o conjunto destes registos e a funcionalidade destes últimos.

### A.2.1 REGISTOS PRINCIPAIS

Estes registos servem essencialmente para configuração de algumas das características do PEPE, que na sua maioria num processador comercial estariam fixas, mas que num processador pedagógico se justificam poder ser alteradas, para permitir experimentar uma gama maior de soluções. A instrução MOV é a única que permite escrever e ler estes registos, por cópia de valor de ou para, respectivamente, um registo principal (R0 a R15).

| R  | NP | DE | I <sub>E3</sub> | I <sub>E2</sub> | I <sub>E1</sub> | I <sub>E0</sub> | IE | TD | TV | B | A | V | C | N | Z |
|----|----|----|-----------------|-----------------|-----------------|-----------------|----|----|----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12              | 11              | 10              | 9               | 8  | 7  | 6  | 5 | 4 | 3 | 2 | 1 | 0 |

Fig. A.1 - Disposição dos vários bits no RE (Registo de Estado)

### A.2.2 REGISTOS AUXILIARES

Quando o PEPE arranca, a seguir a um *reset*, todos estes registos estão a 0000H, para omissão. A Tabela A.3 representa os registos auxiliares actualmente definidos para o PEPE, num total de 16 possíveis.

| REGISTRO<br>AUXILIAR | NOME<br>ALTERNATIVO | FUNCTIONALIDADE                                                                                                                                        |
|----------------------|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| A0                   | RCN                 | Registo de Configuração do Núcleo – Contém bits que controlam a configuração do núcleo do PEPE                                                         |
| A1                   | RCCD                | Registo de Configuração da Cache de Dados – Bits que controlam a configuração da cache de dados                                                        |
| A2                   | RCCI                | Registo de Configuração da Cache de Instruções – Bits que controlam a configuração da cache de instruções                                              |
| A3                   | RCMV                | Registo de Configuração da Memória Virtual – Bits para configurar o sistema de memória virtual                                                         |
| A4                   | RTP                 | Registo da Tabela de Páginas – Contém o endereço do primeiro nível de tabelas de páginas virtuais (directório), para utilização pelas TLBs             |
| A5                   | RPID                | Registo do PID – Contém o PID (Identificador de Processo) do processo em execução, para utilização nas TLBs. Apenas os 8 bits de menor peso são usados |
| A15 .. A6            |                     | Reservado                                                                                                                                              |

Tabela A.3 - Registos auxiliares do PEPE

O RCN (Registo de Configuração do Núcleo) permite configurar as características fundamentais do núcleo do processador (Fig. 7.1).

| BIT      | SIGLA | FUNCTIONALIDADE                                                                                                                                                                                                                                                                                     |
|----------|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0        | CDL   | Cache de Dados Ligada – Se CDL=0, o processador não usa cache de dados                                                                                                                                                                                                                              |
| 3, 2 e 1 | TCD   | Tamanho da Cache de Dados (em conjuntos) – 000=1, 001=2, 010=4, 011=8, 100=16, 101=32, 110=64, 111=128 (se TCD=1, então trata-se de uma cache completamente associativa)                                                                                                                            |
| 5 e 4    | TBCD  | Tamanho do Bloco da Cache de Dados (em palavras de 16 bits) – 00=1, 01=2, 10=4, 11=8                                                                                                                                                                                                                |
| 7 e 6    | ACD   | Associatividade (nº de bits) da Cache de Dados – 00=1, 01=2, 10=4, 11=8 (se ACD=1, então trata-se de uma cache de mapeamento directo)                                                                                                                                                               |
| 9 e 8    | PSCD  | Política de Substituição da Cache de Dados – 00=sequencial (FIFO); 01=aleatória; 10=LRU (Least Recently Used); ou Usada Menos Recentemente; 11=Reservado                                                                                                                                            |
| 10       | PEC0  | Política de Escrita da Cache de Dados – 0=escrita-imediata ( <i>write-through</i> ), 1=escrita-atrasada ( <i>write-back</i> )                                                                                                                                                                       |
| 11       | FCD   | Flush da Cache de Dados – Se FCD=1 e o bit PEC0 antes da escrita for 1, considera-se que a cache tem conteúdo válido com potenciais blocos alterados, pelo que todos os blocos com o bit de validade=1 e bit diff=1 são escritos na memória principal antes de invalidar todas as entradas da cache |
| 15 .. 12 | R     | Reservado                                                                                                                                                                                                                                                                                           |

Tabela A.5 - Bits do RCCD (Registo de Configuração da Cache de Dados).

De igual forma, o RCCI (Registo de Configuração da Cache de Instruções) configura a cache de instruções, de acordo com a Tabela A.6.

Com estes registos, é possível, para cada uma das caches:

- \* Ligar/desligar a cache. Sem cache, o acesso à memória principal é sempre feito.
- \* A interface de memória garante, caso seja necessário, o arbitrio entre os acessos de instruções e de dados;
- \* Configurar o número de conjuntos e de vias, tamanho do bloco, política de substituição de blocos e política de escrita (só para a cache de dados).

Tabela A.4 - Formato do RCN (Registo de Configuração do Núcleo)

Os bits 7..0 controlam a sensibilidade dos pinos de interrupção do PEPE, tal como descrito na Tabela 6.12, na página 463. Quando o processador arranca, a seguir a uma inicialização (*reset*), possui um relógio interno que não é de tempo real, funcionando no

simulador tão depressa quanto o computador que executa a simulação o permitir. Também é possível seleccionar um relógio externo (que pode ou não ser de tempo real), para maior flexibilidade. O bit E controla o uso de processamento em estágios. O valor inicial do RCN após uma inicialização do PEPE é 0000H. Por meio de uma instrução MOV é possível alterar a configuração do núcleo em qualquer altura.

### A.2.2.2 CONFIGURAÇÃO DAS CACHES

A secção 7.5, na página 622, descreve o funcionamento das caches. A cache de dados pode ser configurada através do registo auxiliar RCCD (Registo de Configuração da Cache de Dados). O significado dos bits deste registo está indicado na Tabela A.5.

Tabela A.5 - Bits do RCCD (Registo de Configuração da Cache de Dados).

Uma escrita neste registo invalida todo o conteúdo da cache.

De igual forma, o RCCI (Registo de Configuração da Cache de Instruções) configura a cache de instruções, de acordo com a Tabela A.6.

Com estes registos, é possível, para cada uma das caches:

- \* Ligar/desligar a cache. Sem cache, o acesso à memória principal é sempre feito.
- \* A interface de memória garante, caso seja necessário, o arbitrio entre os acessos de instruções e de dados;
- \* Configurar o número de conjuntos e de vias, tamanho do bloco, política de substituição de blocos e política de escrita (só para a cache de dados).

| BIT      | SIGLA | NOME E DESCRIÇÃO                                                                                                                                                              |
|----------|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0        | CLI   | Cache de Instruções Ligada – Se CLI=0, o processador não usa cache de instruções                                                                                              |
| 3, 2 e 1 | TCI   | Tamanho da Cache de Instruções (em conjuntos) – 000=1, 001=2, 010=4, 011=8, 100=16, 101=32, 110=64, 111=128 (se TCI=1, então trata-se de uma cache completamente associativa) |
| 5 e 4    | TBCI  | Tamanho do Bloco da Cache de Instruções (em palavras de 16 bits) – 00=1, 01=2, 10=4, 11=8                                                                                     |
| 7 e 6    | ACI   | Associatividade (n.º de vias) da Cache de Instruções – 00=1, 01=2, 10=4, 11=8 (se ACI=1, então trata-se de uma cache de mapeamento directo)                                   |
| 9 e 8    | PSCI  | Política de Substituição da Cache de Instruções – 00=sequencial (FIFO); 01=aleatória; 10=LRU (Least Recently Used, ou Usada Menos Recentemente), 11=Reservado                 |
| 15 .. 10 | R     | Reservado                                                                                                                                                                     |

Tabela A.6 - Bits do RCCT (Registo de Configuração da Cache de Instruções).

Uma escrita neste registo invalida todo o conteúdo da cache.

A escrita num destes registos tem como efeito invalidar todas as entradas da cache respectiva (a outra não é afectada), mesmo que o valor a escrever seja igual ao último escrito. Pode usar-se esta característica para invalidar completamente uma cache, em qualquer altura. É ainda possível, no caso da cache de dados, especificar se a invalidação deve ser precedida da cópia (*Flush*) dos blocos alterados para a memória principal (desde que a sua política em vigor seja escrita atrasada, ou *write-back*).

### A.2.2.3 CONFIGURAÇÃO DA MEMÓRIA VIRTUAL

O mecanismo da memória virtual é descrito na secção 7.6, na página 643, e está desligado por omissão. O RCMV (Registo de Configuração da Memória Virtual) permite ligá-lo e configura-lo, de acordo com a Tabela A.7.

| BIT     | SIGLA | NOME E DESCRIÇÃO                                                                                                                                             |
|---------|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0       | MVL   | Memória Virtual Ligada – Se MVL=0, o processador não usa memória virtual. Os endereços nos registos são considerados físicos                                 |
| 2 e 1   | DTLB  | Tamanho da TLB de Dados (em palavras) – 00=2, 01=4, 10=8, 11=16 (a TLB de dados é uma cache completamente associativa, de bloco de uma só palavra)           |
| 3       | PSTO  | Política de Substituição da TLB de Dados – 0=aleatória; 1=RU (Least Recently Used, ou Menos Usada Recientemente)                                             |
| 5 e 4   | ITLB  | Tamanho da TLB de Instruções (em palavras) – 00=2, 01=4, 10=8, 11=16 (a TLB de Instruções é uma cache completamente associativa, de bloco de uma só palavra) |
| 6       | PSTI  | Política de Substituição da TLB de Instruções – 0=aleatória; 1=LRU (Least Recently Used, ou Menos Usada Recientemente)                                       |
| 15 .. 7 | R     | Reservado                                                                                                                                                    |

Tabela A.7 - Bits do RCMV (Registo de Configuração da Memória Virtual).

O RTP (Registo da Tabela de Páginas) aponta para a base do directório e o RPID (Registo do Identificador de Processo) identifica o processo corrente. Note-se que estes dois registos, bem como a tabela de páginas (no mínimo, o directório), devem ser inicializados antes de ligar a memória virtual.

## A.3 EXCEPÇÕES

Designam-se por exceções (secção 6.2, na página 459) os eventos a que o processador é sensível e que constituem alterações, normalmente pouco frequentes, ao fluxo normal de instruções de um programa. As exceções podem ter origem:

- Externa – São geradas pela activação dos pinos INT0 a INT3 do PEPE e permitem lidar com eventos assíncronos ao programa e associados aos periféricos. Essas exceções designam-se por interrupções (secção 6.2.2, na página 462);

■ Interna – São geradas por erros na execução das instruções ou pela instrução SWI (Software Exception), que permite gerar explicitamente qualquer exceção, de 0 a 255. As exceções 0 a 31, em que se incluem as exceções predefinidas, só podem ser invocadas se o nível de privilégio correte for Sistema (bit NP=1 no RE). Se forem invocadas no nível Utilizador, é gerada a exceção SISTEMA.

A cada exceção está associada uma rotina de tratamento da exceção (ou rotina de serviço da exceção, ou simplesmente rotina de exceção), cujo endereço consta da Tabela de Excepções, que deve conter uma palavra (o endereço da rotina de tratamento) para cada uma das exceções que se pretender invocar. A Tabela de Excepções começa no endereço indicado pelo registo RTE (Base da Tabela de Excepções), que deverá ser previamente inicializado com um valor adequado.

A Tabela A.8 descreve as exceções predefinidas do PEPE.

## A.4 CONJUNTO DE INSTRUÇÕES

A Tabela A.9 ocupa as páginas seguintes e descreve as instruções do PEPE, seguindo uma notação RTL descrita na Tabela 4.10, na página 212, e com a seguinte informação:

- Sintaxe em linguagem assembly;
- Quatro campos de 4 bits cada, com o código de operação (*opcode*, que pode ser 4 ou 8 bits) e os operandos. Nem todas as instruções têm dois operandos, pelo que em alguns casos há campos desaproveitados que o PEPE ignora (marcados com "xxxx") e noutras há um único operando que gasta 8 bits ou mesmo 12 bits. Todas as instruções têm 16 bits. Os campos com 4 bits especificados correspondem aos códigos de operação gerados pelo assembler ou a operandos de valor fixo. Nem todas as codificações possíveis são usadas (há algumas livres);
- Descrição da instrução em operações elementares em RTL;
- Bits de estado afectados pela instrução;
- Comentários com informação adicional à descrição RTL.

| NÚMERO | NOME           | CAUSA                                                               | OCORRE EM          | MASCARA | ATENDIMENTO                  |
|--------|----------------|---------------------------------------------------------------------|--------------------|---------|------------------------------|
| 0..255 | SWE            | Execução desta instrução                                            | Instrução          | -       | Imediato                     |
| 0      | INTO           | Activação externa do pino INT0                                      | Qualquer altura    | Sim     | Após instrução em que ocorre |
| 1      | INT1           | Activação externa do pino INT1                                      | Qualquer altura    | Sim     | Após instrução em que ocorre |
| 2      | INT2           | Activação externa do pino INT2                                      | Qualquer altura    | Sim     | Após instrução em que ocorre |
| 3      | INT3           | Activação externa do pino INT3                                      | Qualquer altura    | Sim     | Após instrução em que ocorre |
| 4      | EXCESSO        | Excesso em operação aritmética                                      | Instrução          | Sim     | Imediato                     |
| 5      | DIV0           | Divisão (DIV) por zero                                              | Instrução          | Sim     | Imediato                     |
| 6      | COD_INV        | Código de operação inválido                                         | Descodificação     | Não     | Imediato                     |
| 7      | D._DESALINHADO | Acesso em 16 bits à memória (MOV) com endereço ímpar                | Instrução          | Não     | Imediato                     |
| 8      | L._DESALINHADO | Busca de Instrução com PC ímpar                                     | Busca de Instrução | Não     | Encadeado                    |
| 9      | D._FALTA_PAG   | Acesso de dados à memória virtual e à página não está em memória    | Execução           | Não     | Imediato                     |
| 10     | L.FALTA_PAG    | Busca de Instrução e a página não está em memória                   | Busca de Instrução | Não     | Encadeado                    |
| 11     | D._PROT        | Acesso do utilizador a página de sistema                            | Execução           | Não     | Imediato                     |
| 12     | L.PROT         | Acesso do utilizador a código de sistema                            | Busca de Instrução | Não     | Encadeado                    |
| 13     | SO._LEITURA    | Escrita em página protegida                                         | Execução           | Não     | Imediato                     |
| 14     | SISTEMA        | Execução de uma instrução de nível de sistema em modo de utilizador | Execução           | Não     | Imediato                     |

Tabela A.8 - Excepções predefinidas do PEPE

As instruções marcadas com cinzento são na realidade representações alternativas para instruções que já existem, apenas com o fim de facilitar a programação, pelo que não gastam códigos de operação (*opcodes*). Por exemplo, JEQ (salta se os operandos de uma instrução anterior de comparação, CMP, forem iguais) é idêntica a JZ (salta se o bit de estado Z estiver activo, o que acontecerá se os operandos de uma instrução anterior de comparação, CMP, forem iguais).

| SINTaxe em ASSEMBLY |        |      | CAMPOS DA INSTRUÇÃO (4x4 BITS) |    |      |  | ACÇÕES (RTL)                        |  | BITS DE ESTADO               |                                                        | COMENTÁRIOS        |  |
|---------------------|--------|------|--------------------------------|----|------|--|-------------------------------------|--|------------------------------|--------------------------------------------------------|--------------------|--|
| ADD                 | Rd, Rs | 0101 | 0000                           | Rd | Rs   |  | Rd $\leftarrow$ Rd + Rs             |  | Z, N, C, V                   |                                                        |                    |  |
|                     | Rd, k  | 0101 | 0001                           | Rd | k    |  | Rd $\leftarrow$ Rd + k              |  | Z, N, C, V                   | $k \in [-8 .. +7]$                                     |                    |  |
| ADDC                | Rd, Rs | 0101 | 0010                           | Rd | Rs   |  | Rd $\leftarrow$ Rd + Rs + C         |  | Z, N, C, V                   |                                                        |                    |  |
| SUB                 | Rd, Rs | 0101 | 0011                           | Rd | Rs   |  | Rd $\leftarrow$ Rd - Rs             |  | Z, N, C, V                   |                                                        |                    |  |
|                     | Rd, k  | 0101 | 0100                           | Rd | k    |  | Rd $\leftarrow$ Rd - k              |  | Z, N, C, V                   | $k \in [-8 .. +7]$                                     |                    |  |
| SUBB                | Rd, Rs | 0101 | 0101                           | Rd | Rs   |  | Rd $\leftarrow$ Rd - Rs - C         |  | Z, N, C, V                   |                                                        |                    |  |
|                     | Rd, Rs | 0101 | 0110                           | Rd | Rs   |  | (Rd - Rs)                           |  | Z, N, C, V                   | Rd não é alterado                                      |                    |  |
| CMP                 | Rd, k  | 0101 | 0111                           | Rd | k    |  | (Rd - k)                            |  | Z, N, C, V                   | $k \in [-8 .. +7]$<br>Rd não é alterado                |                    |  |
| MUL                 | Rd, Rs | 0101 | 1000                           | Rd | Rs   |  | Rd $\leftarrow$ Rd * Rs             |  | Z, N, C, V                   |                                                        |                    |  |
| DIV                 | Rd, Rs | 0101 | 1001                           | Rd | Rs   |  | Rd $\leftarrow$ quociente (Rd / Rs) |  | Z, N, C, V $\leftarrow$ 0    | Divisão inteira                                        |                    |  |
| MOD                 | Rd, Rs | 0101 | 1010                           | Rd | Rs   |  | Rd $\leftarrow$ resto (Rd / Rs)     |  | Z, N, C, V $\leftarrow$ 0    | Resto da divisão inteira                               |                    |  |
| NEG                 | Rd     | 0101 | 1011                           | Rd | x00x |  | Rd $\leftarrow$ -Rd                 |  | Z, N, C, V                   | Complemento para 2<br>V $\leftarrow$ 1 se Rd = 8000H   |                    |  |
| AND                 | Rd, Rs | 0110 | 0000                           | Rd | Rs   |  | Rd $\leftarrow$ Rd $\wedge$ Rs      |  | Z, N                         |                                                        |                    |  |
| OR                  | Rd, Rs | 0110 | 0001                           | Rd | Rs   |  | Rd $\leftarrow$ Rd $\vee$ Rs        |  | Z, N                         |                                                        |                    |  |
| NOT                 | Rd     | 0110 | 0010                           | Rd | x00x |  | Rd $\leftarrow$ Rd $\oplus$ FFFFH   |  | Z, N                         | Complemento para 1                                     |                    |  |
| XOR                 | Rd, Rs | 0110 | 0011                           | Rd | Rs   |  | Rd $\leftarrow$ Rd $\oplus$ Rs      |  | Z, N                         |                                                        |                    |  |
| TEST                | Rd, Rs | 0110 | 0100                           | Rd | Rs   |  | Rd $\wedge$ Rs                      |  | Z, N                         | Rd não é alterado                                      |                    |  |
| BIT                 | Rd, n  | 0110 | 0101                           | Rd | n    |  | Z $\leftarrow$ Rd(k) $\oplus$ 1     |  | Z                            | Rd não é alterado                                      |                    |  |
| SET                 | Rd, n  | 0110 | 0110                           | Rd | n    |  | Rd(n) $\leftarrow$ 1                |  | Z, N ou outro (se Rd for RE) | $n \in [0 .. 15]$<br>Se Rd for RE, afecta apenas RE(n) |                    |  |
| EI                  |        | 0110 | 0110                           | RE | IE   |  | RE(IE) $\leftarrow$ 1               |  |                              | EI                                                     | Enable interrupt   |  |
| EIO                 |        | 0110 | 0110                           | RE | IE0  |  | RE(IE0) $\leftarrow$ 1              |  |                              | EIO                                                    | Enable interrupt 0 |  |
| EI1                 |        | 0110 | 0110                           | RE | IE1  |  | RE(IE1) $\leftarrow$ 1              |  |                              | EI1                                                    | Enable interrupt 1 |  |
| EI2                 |        | 0110 | 0110                           | RE | IE2  |  | RE(IE2) $\leftarrow$ 1              |  |                              | EI2                                                    | Enable interrupt 2 |  |
| EI3                 |        | 0110 | 0110                           | RE | IE3  |  | RE(IE3) $\leftarrow$ 1              |  |                              | EI3                                                    | Enable interrupt 3 |  |
| SETC                |        | 0110 | 0110                           | RE | C    |  | RE(C) $\leftarrow$ 1                |  |                              | C                                                      | Set Carry flag     |  |
| EDMA                |        | 0110 | 0110                           | RE | DE   |  | RE(DE) $\leftarrow$ 1               |  |                              | DE                                                     | Enable DMA         |  |

| SÍNTAXE EM ASSEMBLY |       | CAMPOS DA INSTRUÇÃO (4x4 BITS) |      |    |     | ACÇÕES (RTL)                                                                               |  | BITS DE ESTADO               | COMENTÁRIOS                                                                                                                     |
|---------------------|-------|--------------------------------|------|----|-----|--------------------------------------------------------------------------------------------|--|------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| CLR                 | Rd, n | 0110                           | 0111 | Rd | n   | Rd(n) ← 0                                                                                  |  | Z, N ou outro (se Rd for RE) | $n \in [0..15]$<br>Se Rd for RE, afecta apenas RE(n)                                                                            |
| DI                  |       | 0110                           | 0111 | RE | IE  | RE(IE) ← 0                                                                                 |  | IE                           | Disable interrupt                                                                                                               |
| DIO                 |       | 0110                           | 0111 | RE | IE0 | RE(IE0) ← 0                                                                                |  | E10                          | Disable interrupt 0                                                                                                             |
| DII                 |       | 0110                           | 0111 | RE | IE1 | RE(IE1) ← 0                                                                                |  | E11                          | Disable interrupt 1                                                                                                             |
| DIZ                 |       | 0110                           | 0111 | RE | IE2 | RE(IE2) ← 0                                                                                |  | E12                          | Disable interrupt 2                                                                                                             |
| DI3                 |       | 0110                           | 0111 | RE | IE3 | RE(IE3) ← 0                                                                                |  | E13                          | Disable interrupt 3                                                                                                             |
| CLRC                |       | 0110                           | 0111 | RE | C   | RE(C) ← 0                                                                                  |  | C                            | Clear Carry flag                                                                                                                |
| DDMA                |       | 0110                           | 0111 | RE | DE  | RE(DE) ← 0                                                                                 |  | DE                           | Disable DMA                                                                                                                     |
| CPL                 | Rd, n | 0110                           | 1000 | Rd | n   | Rd(n) ← Rd(n) ⊕ 1                                                                          |  | Z, N ou outro (se Rd for RE) | $n \in [0..15]$<br>Se Rd for RE, afecta apenas RE(n)                                                                            |
| CPLC                |       | 0110                           | 1000 | RE | C   | RE(C) ← ~RE(C) ⊕ 1                                                                         |  | C                            | Complement Carry flag                                                                                                           |
| SHR                 | Rd, n | 0110                           | 1001 | Rd | n   | $n > 0 : C \leftarrow Rd(n-1)$<br>$n > 0 : Rd \leftarrow 0\{n\}    Rd(15..n)$              |  | Z, N, C                      | $n \in [0..15]$<br>Se $n=0$ , actualiza Z e N (C não)                                                                           |
| SHL                 | Rd, n | 0110                           | 1010 | Rd | n   | $n > 0 : C \leftarrow Rd(15-n+1)$<br>$n > 0 : Rd \leftarrow Rd(15-n..0)    0\{n\}$         |  | Z, N, C                      | $n \in [0..15]$<br>Se $n=0$ , actualiza Z e N (C não)                                                                           |
| SHRA                | Rd, n | 0101                           | 1100 | Rd | n   | $n > 0 : C \leftarrow Rd(n-1)$<br>$n > 0 : Rd \leftarrow Rd(15)\{n\}    Rd(15..n)$         |  | Z, N, C                      | $n \in [0..15]$<br>Se $n=0$ , actualiza Z e N (C não)                                                                           |
| SHLA                | Rd, n | 0101                           | 1101 | Rd | n   | $n > 0 : C \leftarrow Rd(15-n+1)$<br>$n > 0 : Rd \leftarrow Rd(15-n..0)    0\{n\}$         |  | Z, N, C, V                   | $n \in [0..15]$<br>Se $n=0$ , actualiza Z e N (C não)<br>V ← 1 se algum dos bits que sair for diferente do Rd(15) após execução |
| ROR                 | Rd, n | 0110                           | 1100 | Rd | n   | $n > 0 : C \leftarrow Rd(n-1)$<br>$n > 0 : Rd \leftarrow Rd(n-1..0)    Rd(15..n)$          |  | Z, N, C                      | $n \in [0..15]$<br>Se $n=0$ , actualiza Z e N (C não)                                                                           |
| ROL                 | Rd, n | 0110                           | 1101 | Rd | n   | $n > 0 : C \leftarrow Rd(15-n+1)$<br>$n > 0 : Rd \leftarrow Rd(15-n..0)    Rd(15..15-n+1)$ |  | Z, N, C                      | $n \in [0..15]$<br>Se $n=0$ , actualiza Z e N (C não)                                                                           |
| RORC                | Rd, n | 0110                           | 1110 | Rd | n   | $n > 0 : (Rd    C) \leftarrow Rd(n-2..0)    C    Rd(15..n-1)$                              |  | Z, N, C                      | $n \in [0..15]$<br>Se $n=0$ , actualiza Z e N (C não)                                                                           |
| ROLC                | Rd, n | 0110                           | 1111 | Rd | n   | $n > 0 : (C    Rd) \leftarrow Rd(15-n+1..0)    C    Rd(15..15-n+2)$                        |  | Z, N, C                      | $n \in [0..15]$<br>Se $n=0$ , actualiza Z e N (C não)                                                                           |

| SÍNTAXE EM ASSEMBLY |                      | CAMPOS DA INSTRUÇÃO (4x4 BITS) |      |      |          | ACÇÕES (RTL)                            |  | BITS DE ESTADO | COMENTÁRIOS                                                                   |
|---------------------|----------------------|--------------------------------|------|------|----------|-----------------------------------------|--|----------------|-------------------------------------------------------------------------------|
| MOV                 | Rd, [Rs + off]       | 0111                           | Rd   | Rs   | off/2    | Rd ← Mw[Rs + off]                       |  | Nenhum         | off ∈ [-16 .. +14]                                                            |
|                     | Rd, [Rs]             | 0111                           | Rd   | Rs   | 0000     | Rd ← Mw[Rs # 0]                         |  | Nenhum         |                                                                               |
|                     | Rd, [Rs + RI]        | 1000                           | Rd   | Rs   | RI       | Rd ← Mw[Rs + RI]                        |  | Nenhum         |                                                                               |
|                     | [Rd + off], Rs       | 1001                           | Rs   | Rd   | off/2    | Mw[Rd + off] ← Rs                       |  | Nenhum         | off ∈ [-16 .. +14]                                                            |
|                     | [Rd], Rs             | 1001                           | Rs   | Rd   | 0000     | Mw[Rd # 0] ← Rs                         |  | Nenhum         |                                                                               |
|                     | [Rd + RI], Rs        | 1010                           | Rs   | Rd   | RI       | Mw[Rd + RI] ← Rs                        |  | Nenhum         |                                                                               |
| MOVB                | Rd, [Rs]             | 1011                           | 0000 | Rd   | Rs       | Rd ← 0{8}    Mb[Rs]                     |  | Nenhum         |                                                                               |
|                     | [Rd], Rs             | 1011                           | 0001 | Rd   | Rs       | Mb[Rd] ← Rs(7..0)                       |  | Nenhum         | O byte adjacente a Mb[Rd] não é afectado                                      |
| MOVBS               | Rd, [Rs]             | 1011                           | 0010 | Rd   | Rs       | Rd ← Mb[Rs](7){8}    Mb[Rs]             |  | Nenhum         | Igual a MOVB Rd, [Rs] com extensão de sinal                                   |
| MOVL                | Rd, k                | 1100                           | Rd   |      | k        | Rd ← k(7){8}    k                       |  | Nenhum         | k ∈ [-128 .. +127]<br>k é estendido a 16 bits com sinal                       |
| MOVH                | Rd, k                | 1101                           | Rd   |      | k        | Rd(15..8) ← k                           |  | Nenhum         | k ∈ [0 .. 255]<br>O byte de menor peso não é afectado                         |
| MOV                 | Rd, k                | 1100                           | Rd   |      | k        | Rd ← k(7){8}    k                       |  | Nenhum         | Se k ∈ [-128 .. +127]                                                         |
|                     | -Rd, k               | 1100                           | Rd   |      | k(7..0)  | Rd ← k(7){8}    k(7..0)                 |  | Nenhum         | Gera duas instruções (se k ∈ [-32.768 .. -129] + 128 .. +32.767))             |
|                     | Rd, k                | 1101                           | Rd   |      | k(15..8) | Rd(15..8) ← k(15..8)                    |  | Nenhum         |                                                                               |
|                     | Rd, Rs               | 1011                           | 0011 | Rd   | Rs       | Rd ← Rs                                 |  | Nenhum         |                                                                               |
| Ad, Rs              | Ad, Rs               | 1011                           | 0100 | Ad   | Rs       | Ad ← Rs                                 |  | Nenhum         |                                                                               |
|                     | Rd, As               | 1011                           | 0101 | Rd   | As       | Rd ← As                                 |  | Nenhum         |                                                                               |
| MOV                 | Rd, USP              | 1011                           | 0110 | Rd   | xxxx     | Rd ← USP                                |  | Nenhum         |                                                                               |
|                     | USP, Rs              | 1011                           | 0111 | xxxx | Rs       | USP ← Rs                                |  | Nenhum         | O SP lido é o de nível utilizador, independentemente do bit NP do RE          |
| SWAP                | Rd, Rs               | 1011                           | 1000 | Rd   | Rs       | TEMP ← Rd; Rd ← Rs<br>Rs ← TEMP         |  | Nenhum         | O SP escrito é o de nível utilizador, independentemente do bit NP do RE       |
|                     | Rd, [Rs] ou [Rs], Rd | 1011                           | 1001 | Rd   | Rs       | TEMP ← Mw[Rs]; Mw[Rs] ← Rd<br>Rd ← TEMP |  | Nenhum         | Recomendável sem reposição de estado mesmo que um dos acessos à memória falhe |
| PUSH                | Rs                   | 1011                           | 1010 | Rs   | xxxx     | Mw[SP-2] ← Rs<br>SP ← SP - 2            |  | Nenhum         | SP só é actualizado no fim para ser re-exectável                              |
| POP                 | Rd                   | 1011                           | 1011 | Rd   | xxxx     | Rd ← Mw[SP]<br>SP ← SP + 2              |  | Nenhum         |                                                                               |

| SINTAXE EM ASSEMBLY |          | CAMPOS DA INSTRUÇÃO (4 x 4 BITS) |      |                         |      | ACÇÕES (RTL)                   |  | BITS DE ESTADO | COMENTÁRIOS                                                                                                      |
|---------------------|----------|----------------------------------|------|-------------------------|------|--------------------------------|--|----------------|------------------------------------------------------------------------------------------------------------------|
| PUSHC               |          | 1011                             | 1100 | xxxx                    | xxxx | Mw[SP-2] ← Ri<br>SP ← SP - 2   |  | Nenhum         | Executa esta acção para cada um dos registos principais (excepto RE, SP e TEMP). Também não guarda/restaura o PC |
| POPC                |          | 1011                             | 1101 | xxxx                    | xxxx | Ri ← Mw[SP]<br>SP ← SP + 2     |  | Nenhum         |                                                                                                                  |
| JZ                  | etiqueta | 0001                             | 0000 | dif = (etiqueta - PC)/2 |      | Z=1: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JNZ                 | etiqueta | 0001                             | 0001 | dif = (etiqueta - PC)/2 |      | Z=0: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JN                  | etiqueta | 0001                             | 0010 | dif = (etiqueta - PC)/2 |      | N=1: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JNN                 | etiqueta | 0001                             | 0011 | dif = (etiqueta - PC)/2 |      | N=0: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JP                  | etiqueta | 0001                             | 0100 | dif = (etiqueta - PC)/2 |      | (NvZ)=0: PC ← PC + (2*dif)     |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JNP                 | etiqueta | 0001                             | 0101 | dif = (etiqueta - PC)/2 |      | (NvZ)=1: PC ← PC + (2*dif)     |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JC                  | etiqueta | 0001                             | 0110 | dif = (etiqueta - PC)/2 |      | C = 1: PC ← PC + (2*dif)       |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JNC                 | etiqueta | 0001                             | 0111 | dif = (etiqueta - PC)/2 |      | C = 0: PC ← PC + (2*dif)       |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JV                  | etiqueta | 0001                             | 1000 | dif = (etiqueta - PC)/2 |      | V=1: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JNV                 | etiqueta | 0001                             | 1001 | dif = (etiqueta - PC)/2 |      | V=0: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JA                  | etiqueta | 0001                             | 1010 | dif = (etiqueta - PC)/2 |      | (CvZ)=0: PC ← PC + (2*dif)     |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JAE                 | etiqueta | 0001                             | 1011 | dif = (etiqueta - PC)/2 |      | C=0: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JB                  | etiqueta | 0001                             | 1100 | dif = (etiqueta - PC)/2 |      | C=1: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JBE                 | etiqueta | 0001                             | 1011 | dif = (etiqueta - PC)/2 |      | (CvZ)=1: PC ← PC + (2*dif)     |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JEQ                 | etiqueta | 0001                             | 0000 | dif = (etiqueta - PC)/2 |      | Z=1: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JNE                 | etiqueta | 0001                             | 0001 | dif = (etiqueta - PC)/2 |      | Z=0: PC ← PC + (2*dif)         |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JLT                 | etiqueta | 0001                             | 1100 | dif = (etiqueta - PC)/2 |      | NvV=1: PC ← PC + (2*dif)       |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JLE                 | etiqueta | 0001                             | 1101 | dif = (etiqueta - PC)/2 |      | ((NvV)vZ)=1: PC ← PC + (2*dif) |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JGT                 | etiqueta | 0001                             | 1110 | dif = (etiqueta - PC)/2 |      | ((NvV)vZ)=0: PC ← PC + (2*dif) |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JGE                 | etiqueta | 0001                             | 1111 | dif = (etiqueta - PC)/2 |      | NvV=0: PC ← PC + (2*dif)       |  | Nenhum         | etiqueta ∈ [PC-256 .. PC+254]                                                                                    |
| JMP                 | etiqueta | 0010                             |      | dif = (etiqueta - PC)/2 |      | PC ← PC + (2*dif)              |  | Nenhum         | etiqueta ∈ [PC-4096 .. PC+4094]                                                                                  |
|                     | Rs       | 0000                             | 0111 | xxxx                    | xxxx | PC ← Rs                        |  | Nenhum         |                                                                                                                  |

| SINTAXE EM ASSEMBLY |          | CAMPOS DA INSTRUÇÃO (4 x 4 BITS) |                         |      |      | ACÇÕES (RTL)                                                                               |  | BITS DE ESTADO         | COMENTÁRIOS                                                                                              |
|---------------------|----------|----------------------------------|-------------------------|------|------|--------------------------------------------------------------------------------------------|--|------------------------|----------------------------------------------------------------------------------------------------------|
| NOP                 |          | 0000                             | 0000                    | xxxx | xxxx |                                                                                            |  | Nenhum                 | Não faz nada                                                                                             |
| SWE                 | k        | 0000                             | 0001                    | k    |      | TEMP ← RE<br>RE ← 0<br>Mw[SP-2] ← PC<br>Mw[SP-4] ← TEMP<br>PC ← Mw[BTE+2*k]<br>SP ← SP - 4 |  | Todos colocados a zero | SP só é actualizado no fim para ser re-executável<br>k ∈ [0 .. 255]<br>Tem de se usar a pilha de sistema |
| CALL                | etiqueta | 0011                             | dif = (etiqueta - PC)/2 |      |      | Mw[SP-2] ← PC<br>PC ← PC + (2*dif)<br>SP ← SP - 2                                          |  | Nenhum                 | etiqueta ∈ [PC-4096 .. PC+4094]<br>SP só é actualizado no fim para ser re-executável                     |
|                     | Rs       | 0000                             | 0010                    | xxxx | Rs   | Mw[SP-2] ← PC<br>PC ← Rs<br>SP ← SP - 2                                                    |  | Nenhum                 | SP só é actualizado no fim para ser re-executável                                                        |
| CALLF               | etiqueta | 0100                             | dif = (etiqueta - PC)/2 |      |      | RL ← PC<br>PC ← PC + (2*dif)                                                               |  | Nenhum                 | etiqueta ∈ [PC-4096 .. PC+4094]                                                                          |
|                     | Rs       | 0000                             | 0011                    | xxxx | Rs   | RL ← PC<br>PC ← Rs                                                                         |  | Nenhum                 |                                                                                                          |
| RET                 |          | 0000                             | 0100                    | xxxx | xxxx | PC ← Mw[SP]<br>SP ← SP + 2                                                                 |  | Nenhum                 |                                                                                                          |
| RETF                |          | 0000                             | 0101                    | xxxx | xxxx | PC ← RL                                                                                    |  | Nenhum                 |                                                                                                          |
| RFE                 |          | 0000                             | 0110                    | xxxx | xxxx | TEMP ← Mw[SP]<br>PC ← Mw[SP+2]<br>SP ← SP + 4<br>RE ← TEMP                                 |  | Todos restaurados      | SP só é actualizado no fim para ser re-executável<br>Tem de se usar a pilha de sistema                   |

Tabela A.9 - Descrição detalhada das instruções do PEPE

## A.5 PROGRAMAÇÃO DO PEPE

O PEPE pode ser programado em linguagem assembly e em C.

- O assembler está integrado no simulador. Faz parte da interface do módulo PEPE, permitindo ler um ficheiro com um programa em linguagem assembly e gerar código-máquina, que pode ser carregado imediatamente na memória ligada ao PEPE ou armazenado num ficheiro para uso posterior.

- O compilador de C é um programa separado que gera linguagem assembly a partir de um programa em C. O ficheiro com linguagem assembly pode depois ser compilado pelo PEPE.

Para além das instruções descritas na Tabela A.9, o assembler aceita constantes e directivas (secção 5.5, na página 285) que permitem especificar informação para a geração do código-máquina:

- Identificador – Sequência de caracteres alfanuméricos, incluindo o carácter ‘\_’, em que o primeiro não pode ser um dígito. Exemplos: BOM\_DIA, s12, A2b4;
- Etiqueta – Identificador seguido do carácter ‘:’, que precede uma instrução ou directiva e a que o assembler atribui automaticamente o valor do endereço em que essa instrução fica localizada ou que está em vigor quando a directiva é processada;
- Litterais – São valores constantes (números ou caracteres) que podem ser especificados das formas seguintes:
  - Binário – Sequência de caracteres “0” ou “1” terminada pela letra “b” ou “B”. Exemplos: 101b, 00110110B;
  - Decimal – Um valor inteiro decimal entre -32.768 e 65.535, correspondente a uma sequência de caracteres “0” a “9”, eventualmente precedida do sinal (“+” ou “-”) e opcionalmente terminado com a letra “d” ou “D”, embora tal seja assumido quando nenhuma outra base for indicada. Nas instruções que assumem um valor aritmético, o valor máximo é +32.767;
  - Hexadecimal – Sequência de caracteres “0” a “9” e “a” (ou “A”) a “f” (ou “F”) terminada pela letra “h” ou “H”. Exemplos: 05Ah, 0FA46H. São válidos valores entre 0000H e 0FFFFH. As constantes em hexadecimal cujo algorísmo de maior peso é uma letra devem ser escritas com um zero adicional à esquerda, de modo a distinguir a constante de uma variável. Por exemplo, a constante A7H deverá ser escrita 0A7H, caso contrário o assembler gera um erro de identificador desconhecido;
  - Carácter – Uma letra, dígito ou sinal de pontuação representável em código ASCII de 8 bits (Apêndice E, na página 743, com extensão para acentos) e especificado entre plicas. Exemplos: ‘g’, ‘3’ e ‘#’;
  - Cadeia de caracteres – um conjunto de caracteres entre aspas, como por exemplo “olá”.

Internamente, todos os valores são convertidos para sequências de bits, em que o seu significado depende da instrução que os processa. Algumas instruções interpretam os valores como estando em notação de complemento para 2, em que o bit mais à esquerda determina o sinal. Se necessário, é feita uma extensão do número de bits. O assembler gera um erro se a constante estiver fora do intervalo admissível (não couper no número de bits disponíveis para a representar numa dada instrução, por exemplo).

Constantes simbólicas – Identificadores que representam um valor constante, definido com a directiva EQU ou de endereços (etiquetas). Constituem uma boa prática de programação, pela legibilidade acrescida (o símbolo associado à constante dá uma pista sobre a natureza do valor) e facilidade de alteração (apenas na sua definição e não em todos os sítios onde é usada). Note-se que em linguagem assembly as variáveis são células de memória que têm de ser identificadas por um endereço constante (o que pode variar é o conteúdo da célula de memória). Na prática, todos os identificadores em linguagem assembly são constantes simbólicas;

EQU – Directiva que permite definir o valor de uma constante simbólica de dados (identificador EQU constante). Sempre que no código-fonte aparecer identificador, o assembler interpreta-o como sendo constante. Exemplo (note-se que o identificador abc não é uma etiqueta e não deve ser seguido de ‘:’):

```
abc    EQU  3 ; atribui o valor 3 ao identificador abc
```

WORD – Directiva que permite reservar espaço para uma palavra em memória e inicializar esse espaço com um valor ([etiqueta] WORD constante). A etiqueta é opcional. Esta directiva é diferente de EQU, pois afecta a memória, enquanto que EQU define apenas um nome alternativo para uma constante. Exemplo:

```
xyz: WORD 3 ; reserva espaço para uma palavra, no endereço
```

TABLE – Directiva que permite reservar espaço contíguo para várias palavras em memória, sem inicializar esse espaço ([etiqueta] TABLE constante). A etiqueta é opcional. Exemplo:

```
tab: TABLE 10 ; reserva espaço para 10 palavras (a primeira fica localizada no endereço atribuído à etiqueta tab)
```

STRING – Directiva que permite inicializar um conjunto de bytes em memória ([etiqueta] STRING constante, constante). A etiqueta é opcional. Pode especificar-se várias constantes, separadas por vírgulas, que podem ser valores numéricos (representáveis em 8 bits), caracteres individuais ou cadeias de caracteres. Esta directiva não alinha o endereço da etiqueta, que pode assim ser ímpar. Exemplo:

```
b: STRING 3, 'y', "ola" ; reserva 5 bytes e armazena lá o valor das várias constantes, pela ordem com que aparecem (o 3 fica no endereço atribuído à etiqueta b)
```

- PLACE** – Directiva que indica ao *assembler* onde deve gerar o próximo endereço atribuindo-o a uma etiqueta ou localizando nesse endereço um dado ou instrução (*PLACE constante*). Podem existir vários PLACES no mesmo programa. Por omis-  
são, o *assembler* assume PLACE 0 no início do programa. Exemplo:

```
PLACE 100H ; a instrução que vier a seguir fica localizada...
; no endereço 100H
```

- O *assembler* permite ainda definir novas instruções, tal como já exemplificado na página 594. A documentação do PEPE, no site de apoio do livro, contém as regras completas para poder definir novas instruções.

## MANUAL DE PROGRAMADOR DO CREPE

### APÊNDICE B

O CREPE (Controlador Recomendado Especialmente Para Ensino) é um microcontrolador de 16 bits com o mesmo conjunto de instruções que o PEPE mas sem cache ou interface de memória externa, ligando ao mundo exterior apenas pelos pinos dos periféricos que incorpora. Permite construir sistemas simples, apenas com o microcontrolador e os dispositivos controlados.

O CREPE é mais eficiente em termos de simulação que o PEPE, pois o simulador não tem de simular todos os ciclos de acesso à memória nem o funcionamento de memória, periféricos, etc., pois estão todos contidos dentro do módulo CREPE.

A secção 6.5.3.2, na página 538, contém uma descrição sumária das características do CREPE, incluindo a sua arquitetura interna (Fig. 6.56b). Este apêndice descreve-o mais em detalhe, tomando o PEPE como base, e dá exemplos da sua programação, no contexto do simulador.

### B.1 PINOS DO MÓDULO CREPE

O módulo CREPE é um dos módulos disponíveis no simulador descrito no Apêndice C, com os pinos indicados pela Tabela B.1, dos quais alguns existem também no PEPE. A funcionalidade destes últimos foi já descrita na secção A.1, enquanto a dos pinos específicos do CREPE está descrita com mais detalhe na secção B.3.

### B.2 REGISTOS AUXILIARES

Os registos principais do CREPE são os mesmos que no PEPE (R0 a R15). Os registos auxiliares (A0 a A15) permitem configurar o processador e lidar com os periféricos, sendo acessíveis, quer em leitura quer em escrita, exclusivamente pela instrução MOV (tal como no PEPE).

Os registos auxiliares do CREPE são os indicados pela Tabela B.2, que refere também os nomes alternativos para os registos e que o *assembler* reconhece. Os restantes registos auxiliares são usados para controlar os periféricos (a secção B.3 descreve o funcionamen-

to dos vários periféricos e o formato dos registos auxiliares que os controlam) ou apenas para transferir dados (de 8 ou 16 bits, consoante o indicado na Tabela B.2).

| COMUNICAÇÃO AO PEPE | NOME | TIPO          | FUNCTIONALIDADE                                                                       |
|---------------------|------|---------------|---------------------------------------------------------------------------------------|
| RESET               | 1    | Entrada       | Inicialização (Reset) (activo a 1)                                                    |
| INT0                | 1    | Entrada       | Interrupção 0 (activação programável a 0, 1 ou um dos flancos)                        |
| INT1                | 1    | Entrada       | Interrupção 1 (activação programável a 0, 1 ou um dos flancos)                        |
| INT2                | 1    | Entrada       | Interrupção 2 (activação programável a 0, 1 ou um dos flancos)                        |
| INT3                | 1    | Entrada       | Interrupção 3 (activação programável a 0, 1 ou um dos flancos)                        |
| CLOCK               | 1    | Entrada       | Entrada de relógio externo do microcontrolador                                        |
| PA(7..0)            | 8    | Entrada/saída | Porto A de entrada ou saída de 8 bits                                                 |
| PB(7..0)            | 8    | Entrada/saída | Porto B de entrada ou saída de 8 bits                                                 |
| PC(7..0)            | 8    | Entrada/saída | Porto C de entrada ou saída de 8 bits                                                 |
| PD(7..0)            | 8    | Entrada/saída | Porto D de entrada ou saída de 8 bits                                                 |
| C                   | 1    | Entrada       | Sinal que controla a contagem de tempo pelo cronómetro do temporizador 1 (activo a 1) |
| T2                  | 1    | Saída         | Saída do Temporizador 2, para geração de formas de onda quadradas                     |
| RX1                 | 1    | Entrada       | Pino de recepção da UART 1                                                            |
| TX1                 | 1    | Saída         | Pino de transmissão da UART 1                                                         |
| RX2                 | 1    | Entrada       | Pino de recepção da UART 2                                                            |
| TX2                 | 1    | Saída         | Pino de transmissão da UART 2                                                         |

Tabela B.1 - Pinos do módulo CREPE

O registo A0 (RCN – Registo de Configuração do Núcleo) tem um formato igual ao do PEPE (já descrito na Tabela A.4). Os bits 7..0 controlam a sensibilidade dos pinos de interrupção do PEPE, tal como descrito na Tabela 6.12, na página 463.

Quando o processador arranca, a seguir a uma inicialização (reset), possui um relógio interno que não é de tempo real, funcionando no simulador tão depressa quanto o computador que executa a simulação o permitir. Também é possível seleccionar duas frequências de tempo real, permitindo um controlo da escala do tempo, ou um relógio externo, para maior flexibilidade.

| RÉGISTRO AUXILIAR | NOME ALTERNATIVO | FUNCTIONALIDADE                                                                                                                                                                                                                                                                         |
|-------------------|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A0                | RCN              | Registo de Configuração do Núcleo – Contém bits que controlam a configuração do núcleo do CREPE, da forma igual à do PEPE                                                                                                                                                               |
| A1                | RCL              | Registo de Configuração das UARTs – Contém os bits que configuram as UARTs                                                                                                                                                                                                              |
| A2                | RCT              | Registo de Configuração dos Temporizadores – Contém os bits que configuram os 4 portos de entrada/saída                                                                                                                                                                                 |
| A3                | RCP              | Registo de Configuração dos Portos – Contém os bits que configuram os 4 portos de entrada/saída                                                                                                                                                                                         |
| A4                | REP              | Registo de Estado dos Periféricos – Permite aceder a informação sobre o estado actual dos temporizadores, das UARTs e dos pinos de interrupção externas                                                                                                                                 |
| A5                | RVT1             | Registo de Valor do Temporizador 1 – Contém o valor correto do temporizador 1 (16 bits). A escrita neste registo afecta directamente o valor do contador e pode ser usada para o inicializar a zero, por exemplo                                                                        |
| A6                | RT1              | Registo do Tempo 1 – Contém o valor do tempo (16 bits) ao fim do qual o temporizador 1 dispara                                                                                                                                                                                          |
| A7                | RVT2             | Registo de Valor do Temporizador 2 – Contém o valor correto do temporizador 2 (16 bits). A escrita neste registo afecta directamente o valor do contador e pode ser usada para o inicializar a zero, por exemplo                                                                        |
| A8                | RT2              | Registo do Tempo 2 – Contém o valor do tempo (16 bits) ao fim do qual o temporizador 2 dispara                                                                                                                                                                                          |
| A9                | RDU1             | Registo de Dados da UART 1 – Quando se lê, obtém-se o byte recebido pela UART 1. Quando se escreve, indica à UART 1 qual o byte a enviar. Apenas o byte de menor peso é usado. Na leitura, o byte de maior peso vem a 0H                                                                |
| A10               | RDU2             | Registo de Dados da UART 2 – Quando se lê, obtém-se o byte recebido pela UART 2. Quando se escreve, indica à UART 2 qual o byte a enviar. Apenas o byte de menor peso é usado. Na leitura, o byte de maior peso vem a 0H                                                                |
| A11               | RPA              | Registo do Porto A – Quando se lê, obtém-se o byte disponível nos pinos do porto A. Quando se escreve (se o porto A estiver configurado para escrita), memoriza o valor e disponibiliza-o nesses pinos. Apenas o byte de menor peso é usado. Na leitura, o byte de maior peso vem a 0H. |
| A12               | RPB              | Idem, para o porto B                                                                                                                                                                                                                                                                    |
| A13               | RPC              | Idem, para o porto C                                                                                                                                                                                                                                                                    |
| A14               | RPD              | Idem, para o porto D                                                                                                                                                                                                                                                                    |
| A15               |                  | Reservado                                                                                                                                                                                                                                                                               |

Tabela B.2 - Registos auxiliares do CREPE

O valor inicial de todos os registos auxiliares após uma inicialização (reset) do PEPE é 000H. Por meio de uma instrução mov é possível alterar a configuração do núcleo ou de qualquer periférico em qualquer altura.

| BIT   | SIGLA | FUNCTIONALIDADE                                                                                                                                                                   |
|-------|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1..0  | NS10  | Nível de sensibilidade da interrupção 0 (pino INT0). 00=flanco de 0 para 1 (com memória); 01=flanco de 1 para 0 (sem memória); 10=nível 1 (sem memória); 11=nível 0 (sem memória) |
| 3..2  | NS11  | <i>Idem</i> , para o pino de interrupção INT1                                                                                                                                     |
| 5..4  | NS12  | <i>Idem</i> , para o pino de interrupção INT2                                                                                                                                     |
| 7..5  | NS13  | <i>Idem</i> , para o pino de interrupção INT3                                                                                                                                     |
| 8     | FR    | Frequência do relógio do processador. 0=relógio interno, sem frequência de tempo real; 1=relógio externo, dado pelo pino CLOCK                                                    |
| 15..9 |       | Reservados                                                                                                                                                                        |

Tabela B.3 - Formato do RCN (Registo de Configuração do Núcleo) do CREPE

## B.3 FUNCIONAMENTO DOS PERIFÉRICOS

### B.3.1 PORTOS DE ENTRADA/SAÍDA

Tal como indicado na Tabela B.4, o CREPE tem quatro portos de 8 bits, configuráveis individualmente por meio do RCP (Registo de Configuração dos Portos), num de quatro modos:

- Entrada (normal). Os pinos do porto não ligados (deixados "no ar") originam 0 ou 1 de forma aleatória;
- Entrada com valor 1 por omissão (*pull-ups* – ver Nota na página 463). Entrada em que os pinos não ligados (deixados "no ar") são sempre lidos como 1;
- Entrada activa. Igual à entrada normal, mas é gerada uma exceção (PORTO\_A, PORTO\_B, PORTO\_C ou PORTO\_D) sempre que um dos bits de um dado porto muda de valor.
- Saída. O porto memoriza o último valor lá escrito.

| BITS  | SIGLA | FUNCTIONALIDADE                                                                                                                                |
|-------|-------|------------------------------------------------------------------------------------------------------------------------------------------------|
| 1..0  | MPA   | Modo do Porto A – 00=entrada, 01=entrada com <i>pull-ups</i> , 10=entrada activa, 11=saida (não ficam a zero imediatamente após a programação) |
| 3..2  | MPB   | <i>Idem</i> , para o Porto B                                                                                                                   |
| 5..4  | MPC   | <i>Idem</i> , para o Porto C                                                                                                                   |
| 7..6  | MPD   | <i>Idem</i> , para o Porto D                                                                                                                   |
| 15..8 |       | Reservados                                                                                                                                     |

Tabela B.4 - Formato do RCP (Registo de Configuração dos Portos de entrada/saída)

Esta configuração pode mudar em qualquer altura, o que permite implementar portos bidirecionais e com funcionamento de três estados (*tri-state*). A configuração individual e

independente de cada bit dentro de um dado porto não é suportada (apenas para ser mais simples, pois os microcontroladores comerciais permitem normalmente fazê-lo).

Cada porto tem 8 bits e ignora os 8 bits de maior peso da palavra do processador quando é escrito. Na leitura, os 8 bits de maior peso vêm sempre a zero. Na escrita, os portos memorizam o valor escrito (excepto se estiver configurado para entrada, caso em que ignora). Na leitura, o valor lido pelo processador corresponde aos valores reais dos pinos do processador (mesmo se o porto estiver configurado para escrita, caso em que lê o último valor lá escrito).

Os registos RPA (Registo do Porto A), RPB, RRC e REP têm 8 bits e são usados para ler ou escrever do porto respectivo, usando a instrução MOV. Exemplos:

```
MOV RPA, R1 ; escreve o byte da menor peso de R1 no porto A
MOV R3, RPC ; 16 o porto C e escreve-o no byte de menor peso de R3,
             ; cujo byte de maior peso fica a 00H
```

### B.3.2 TEMPORIZADORES

No CREPE existem dois temporizadores, cujo sinal de relógio é igual ao do microcontrolador e que se podem configurar pelo RCT (Registo de Configuração dos Temporizadores), descrito na Tabela B.5.

- Temporizador 1, que usa o registo auxiliar RV1 (Registo de Valor do Temporizador 1) como um contador de 16 bits e tem dois modos de funcionamento:
  - Contínuo. O contador (RV1) começa a contagem em 0000H e em cada ciclo de relógio do contador o seu valor é comparado com o do registo auxiliar RT1 (Registo do Tempo 1), que deve ser carregado previamente com o valor da temporização pretendida. Quando ficam iguais, é gerada uma exceção TEMPOL e o contador RV1 é automaticamente reposto a zero, recomeçando o ciclo. O registo RT1 contém assim o número de ciclos de relógio entre duas exceções TEMPOL consecutivas (RT1=0 significa 64 K ciclos de relógio);
  - Cronómetro. O contador (RV1) começa em zero. Quando o pino C do processador passar para 1, RV1 passa a ser incrementado em cada ciclo do relógio, sendo gerada uma exceção CRONO quando este pino passar de 1 para 0 (altura em que interrompe a contagem). O valor de RV1 terá o valor da contagem de forma precisa, independentemente de o processador demorar mais ou menos tempo a atender e a processar a interrupção. A contagem de medição do tempo de C a 1 não recomeça se entretanto C voltar a 1 (porque poderia destruir o valor de RV1 antes de este ser lido pelo processador). Para a contagem recomeçar é necessário desligar o Temporizador 1 e voltar a ligá-lo (usando o bit T1L do RCT – ver Tabela B.5). O reconhecimento da contagem coloca RV1 a zero. O bit V7 do REP (Tabela B.8) é posto a 1 se a duração de C a 1 exceder a capacidade de contagem de RV1. Se o bit EORT1 do RCT (Tabela B.5) estiver a 1, é também gerada uma exceção (T1\_FIM) se esta situação acontecer.

Temporizador 2, semelhante ao Temporizador 1, mas usando os registos RVT2 (Registo de Valor do Temporizador 2) e RT2 (Registo do Tempo 2), gerando a exceção TEMPO2 no modo Contínuo e tendo o modo Gerador em vez de Cronômetro. No modo Gerador, incrementa o contador RVT2 até atingir o valor de RT2, altura em que RVT2 volta a 0000H e o pino do Processador T2 muda de valor (alternando entre 1 e 0), mas não é gerada nenhuma exceção. Assim, T2 produz um sinal quadrado com um período de  $2 \times RT2$  ciclos de relógio (RT2=0 significa 64 K ciclos de relógio).

Os bits CT1 (Controlo do Temporizador 1) e CT2 (Controlo do Temporizador 2) permitem configurar os dois temporizadores de forma independente a partir de um mesmo registo.

| BITS  | SIGLA | FUNCTIONALIDADE                                                                                                                                                                                   |
|-------|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0     | CT1   | Controlo do Temporizador 1 – 0=A escrita neste registo RCT não afecta o Temporizador 1 (não afecta bits T1L e MT1); 1 = Temporizador 1 é configurado.                                             |
| 1     | T1L   | Temporizador 1 Ligado – Se T1L=0, o temporizador 1 fica inativo; se T1L=1, actualiza configuração e recomeça contagem (coloca registo RVT1 a 0)                                                   |
| 2     | MT1   | Modo do Temporizador 1 – 0=Contínuo, 1 = Cronômetro                                                                                                                                               |
| 3     | EDT1  | Se EDT1=1, gera uma exceção se o Temporizador 1 exceder a sua capacidade (voltando a 0000H) quando está à espera que o sinal C volte a 0. A medição do tempo segue normalmente, sem ser afectada. |
| 4     | CT2   | Controlo do Temporizador 2 (não afecta bits T2L e MT2); 1 = Temporizador 2 é configurado.                                                                                                         |
| 5     | T2L   | Temporizador 2 Ligado – Se T2L=0, o temporizador 2 fica inativo; se T2L=1, actualiza configuração e recomeça contagem (coloca registo RVT2 a 0)                                                   |
| 6     | MT2   | Modo do Temporizador 2 – 0=Contínuo, 1 = Gerador                                                                                                                                                  |
| 15..7 |       | Reservados                                                                                                                                                                                        |

Tabela B.5 - Formato do RCM (Registo de Configuração dos Temporizadores)

### B.3.3 UARTS

O CREPE tem duas UARTs, para suportar a comunicação série assíncrona (secção 6.3.4.3, na página 492). As UARTs têm uma configuração simplificada e assumem, de forma fixa, que há 2 stop bits e 8 bits por carácter, sem bit de paridade. O único parâmetro que se pode realmente configurar é a taxa de transmissão, através da duração em ciclos de relógio de cada bit. Em termos de simulação, não é importante adoptar valores normalizados. É ainda possível indicar se se pretende que seja gerada uma exceção quando um byte é recebido ou quando a transmissão de um byte termina.

A Tabela B.6 descreve as configurações das UARTs, que são especificadas no RCU (Registo de Configuração das UARTs). A leitura dos bytes recebidos e a escrita dos bytes a transmitir é feita pelos Registos de Dados das UARTs (RDU1 e RDU2). Na realidade, cada um destes consiste em dois registos separados, um que só pode ser lido (0 de recepção) e outro que só pode ser escrito (0 de emissão), pelo que partilham a mesma designação. São

registos de 8 bits. Na leitura pelo processador, o byte de maior peso vem sempre a zero e na escrita o byte de maior peso da palavra do processador é ignorado.

| BITS  | SIGLA | FUNCTIONALIDADE                                                                                                                                                                                                     |
|-------|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0     | CTU1  | Controlo da UART 1 – 0=A escrita neste registo RCU não afecta a UART 1 (não afecta bits DB1, ERB1 e ETB1); 1 = UART 1 é configurada com este byte                                                                   |
| 4..1  | DB1   | A duração (em ciclos de relógio) de cada bit na UART 1 é igual a este campo (0 a 15), em que 0 equivale a 16. Portanto, cada bit pode durar entre 1 e 16 ciclos de relógio.                                         |
| 5     | ERB1  | Exceção na Recepção de um Byte da UART 1 – Se ERB1=1, o processador gera a exceção RXL_CIEGOU quando a UART 1 receber um byte (e ficar pronta para lhe poder ser dado outro byte para ser transmitido)              |
| 6     | ETB1  | Exceção na Transmissão de um Byte da UART 1 – Se ETB1=1, o processador gera a exceção TXL_FIM quando a UART 1 acabar de transmitir um byte                                                                          |
| 7     |       | Reservado                                                                                                                                                                                                           |
| 8     | CTU2  | Controlo da UART 2 – 0=A escrita neste registo RCU não afecta a UART 2 (não afecta bits DB2, ERB2 e ETB2); 1 = UART 2 é configurada com este byte                                                                   |
| 12..9 | DB2   | A duração (em ciclos de relógio) de cada bit na UART 2 é igual a este campo (0 a 15), em que 0 equivale a 16. Portanto, cada bit pode durar entre 1 e 16 ciclos de relógio                                          |
| 13    | ERB2  | Exceção na Recepção de um Byte da UART 2 – Se ERB2=1, o processador gera a exceção RXL_CIEGOU quando a UART 2 receber um byte                                                                                       |
| 14    | ETB2  | Exceção na Transmissão de um Byte da UART 2 – Se ETB2=1, o processador gera a exceção TXL_FIM quando a UART 2 acabar de transmitir um byte (e ficar pronta para lhe poder ser dado outro byte para ser transmitido) |
| 15    |       | Reservado                                                                                                                                                                                                           |

Tabela B.6 - Formato do RCU (Registo de Configuração das UARTs)

Os bits CTU1 (Controlo da UART 1) e CTU2 (Controlo da UART 2) permitem configurar as duas UARTs de forma independente a partir de um mesmo registo.

Há duas formas de saber se um byte foi recebido ou se a UART está pronta para transmitir mais um byte, tal como indicado na Tabela B.7 (que se refere a UART 1, mas para a UART 2 é só trocar o 1 pelo 2):

- Por teste (*polling*) de bits de estado no REP (Registo de Estado dos Periféricos – Ver Tabela B.8), mas tal implica ocupar o processador com espera activa;
- Por exceções, que avisam o processador do evento, mas apenas se tiverem sido permitidas pela configuração do RCU. Mesmo usando este método, devem usar-se os bits do REP na recepção de um byte para testar eventuais erros.

É preciso esperar que o byte anterior acabe de ser transmitido. Se se escrever novo byte no RDW1 antes de o anterior ter sido transmitido o resultado não está definido. Provavelmente, o byte finalmente transmitido será uma mistura dos dois.

| FORMA                          | OPERAÇÃO                       | Funcionalidade                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------------------|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Byte foi recebido<br>(polling) | Byte foi recebido              | Testar continuamente o bit ERX1 do REP, que se estiver a 1 indica que existe um byte novo recebido para o processador ler do RDU1 (Registo de Dados da UART 1). Este bit é automaticamente colocado a 0 quando o RDU1 é lido.                                                                                                                                                                         |
|                                | Byte acabou de ser transmitido | O REP tem ainda outros bits que se estiverem a 1 indicam que houve em na recepção de um byte por erros de transmissão (RX1) ou que pelo menos um byte foi perdido por não ter sido lido antes de o seguinte ter chegado (SRX1). Estes bits são igualmente colocados a 0 de forma automática por uma leitura do RDU1. O bit RX1 refere-se sempre ao último byte recebido (enquanto RDU1 não for lido). |
|                                | Byte foi recebido              | Testar continuamente o bit ETX1 do REP. Este bit está normalmente a 1 e vem a 0 automaticamente quando um byte é escrito em RDU1 e volta a 1 quando o byte acaba de ser transmitido.                                                                                                                                                                                                                  |
| Excepção                       | Byte acabou de ser transmitido | Quando um byte chega, é gerada uma excepção RX1_CHECK (se o bit ERX1 do RCU estiver a 1). Também nesta caso deverá ser consultados os bits RX1 e SRX1 (antes de ler o RDU1) para verificar se existiu um erro (só se o bit ETX1 do RCU estiver a 1).                                                                                                                                                  |

Tabela B.7 - Formas de controlar a emissão e recepção de bytes na UART 1. Para a UART 2, é só substituir o 1 pelo 2 nos nomes

**B.3.4 INFORMAÇÃO SOBRE O ESTADO DOS PERIFÉRICOS**

O REP (Registo de estado dos periféricos), descrito na Tabela B.8, concentra os bits que permitem observar o estado dos periféricos e dos pinos de interrupção. O facto de estarem todos juntos num mesmo registo (e não juntos com os bits de configuração dos periféricos respetivos, por exemplo) permite obter toda a informação de estado relevante sobre os periféricos de uma só vez e reflecte a diferença de utilização entre os bits de configuração e os de estado (estes últimos acedidos de forma muito mais frequente).

O REP é um registo que suporta apenas leitura (a escrita é ignorada). A evolução do estado dos seus bits é automática e está fora do controlo do programador, que no entanto pode inicializar alguns destes bits indirectamente, como é o caso dos bits referentes às UARTs, para o que basta reprogramar a UART em causa (actuando no RCU), e do bit VTI, que é inicializado quando se desliga o Temporizador 1 (actuando no RCT).

## B.4 EXCEPÇÕES

A Tabela B.9 descreve as exceções que o CREPE suporta. As exceções anteriores a TEMPO1 são comuns ao PEPE, tendo já sido descritas na Tabela 6.16, na página 481. As exceções específicas do CREPE correspondem a interrupções de periféricos, podem ocorrer várias simultaneamente e de forma assíncrona ao programa. Estas interrupções internas não são atendidas se o bit RE estiver a 0 (tal como as interrupções externas e podem ser mascaradas por configuração dos registos auxiliares).

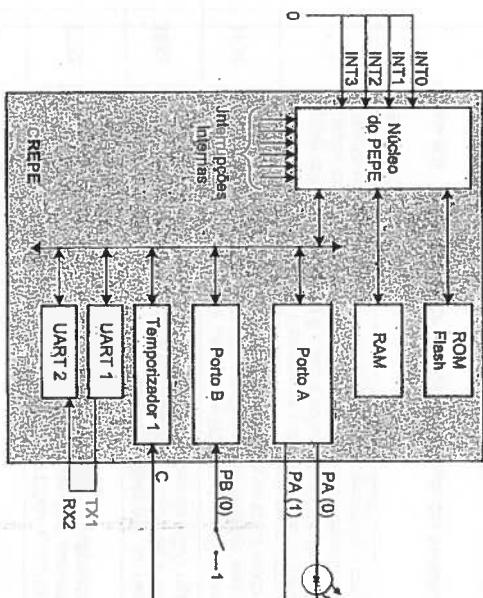
| BITS   | SIGNS      | Funcionalidade                                                                                                                                                                                                                                                                                                                     |
|--------|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0      | PINT0      | Pedido de interrupção 0. Se PTO=1, há um pedido pendente para a Interrupção 0 (pode não reflectir o estado do pino INT0, pois este pode ter sido eventualmente memorizado. Depende do modo do pino INT0, configurado no RCM). Este bit vem a zero automaticamente quando a interrupção 0 é atendida (até haver novo pedido).       |
| 1      | PINT1      | <i>Idem</i> , para a interrupção 1                                                                                                                                                                                                                                                                                                 |
| 2      | PINT2      | <i>Idem</i> , para a interrupção 2                                                                                                                                                                                                                                                                                                 |
| 3      | PINT3      | <i>Idem</i> , para a interrupção 3                                                                                                                                                                                                                                                                                                 |
| 4      | ERX1       | Estado da recepção de bytes da UART 1. Se ERX1=1, foi recebido um byte que ainda não foi lido pelo processador. Este bit é colocado automaticamente a 0 quando o processador lê o byte recebido.                                                                                                                                   |
| 5      | ETX1       | Estado da transmissão de bytes da UART 1. Se ETX1=1, a UART 1 está pronta para ser dado um novo byte para ser enviado. Este bit é colocado a 0 pela escrita de um byte no RDU1 (Registo de Dados da UART 1) e automaticamente reposto a 1 quando o byte acaba de ser transmitido.                                                  |
| 6      | IRX1       | Erro de recepção na UART 1. Se IRX1=1, o byte não foi recebido correctamente, talvez devido a ruído na linha de transmissão. Tipicamente, o erro é detectado por os stop bits não terem a duração correcta. Uma leitura do RDU1 repõe este bit a 0.                                                                                |
| 7      | SRX1       | Sobreposição na recepção na UART 1. Se SRX1=1, foi recebido um novo byte antes de o anterior ter sido lido da UART 1. O último byte recebido está correcto, e este bit é apenas de informação da ocorrência. Uma leitura do RDU1 repõe este bit a 0.                                                                               |
| 8      | ERX2       | Estado da recepção de bytes da UART 2. Se ERX2=1, foi recebido um byte que ainda não foi lido pelo processador. Este bit é colocado automaticamente a 0 quando o processador lê o byte recebido.                                                                                                                                   |
| 9      | ETX2       | Estado da transmissão de bytes da UART 2. Se ETX2=1, a UART 1 está pronta para ser dado um novo byte para ser enviado. Este bit é colocado a 0 pela escrita de um byte no RDU2 (Registo de Dados da UART 2) e automaticamente reposto a 1 quando o byte acaba de ser transmitido.                                                  |
| 10     | IRX2       | Erro de recepção na UART 2. Se IRX2=1, o byte não foi recebido correctamente, talvez devido a ruído na linha de transmissão. Tipicamente, o erro é detectado por os stop bits não terem a duração correcta. Uma leitura do RDU2 repõe este bit a 0.                                                                                |
| 11     | SRX2       | Sobreposição na recepção na UART 2. Se SRX2=1, foi recebido um novo byte antes de o anterior ter sido lido da UART 2. O último byte recebido está correcto, e este bit é apenas de informação da ocorrência. Uma leitura do RDU2 repõe este bit a 0.                                                                               |
| 12     | VTI        | Se VTI=1, o Temporizador 1 excede a sua capacidade de contagem (os 16 bits) ao esperar que o pino C volte a 0 e deu a volta para 0000H. Este bit é colocado a 1 nesta situação. Independentemente do bit ECT1 do RCT permitir que seja gerada uma excepção ou não e só volta a 0 desligando o Temporizador 1 com o bit TLL do RCT. |
| 15..13 | Reservados |                                                                                                                                                                                                                                                                                                                                    |

Tabela B.8 - Formato do REP (Registo de Estado dos Periféricos)

As interrupções são priorizadas de acordo com o indicado Tabela B.9. As exceções com menor número de prioridade são as mais prioritárias. A ordenação por prioridades deve em conta a potencial urgência dos eventos correspondentes. TEMPO1 e T1\_FIM têm a mesma prioridade porque correspondem a modos diferentes do Temporizador 1 e portanto não podem ocorrer simultaneamente.

| NUMERO | NAME       | CUSA                                                                          | PRIORIDADE DE INTERRUPÇÃO |
|--------|------------|-------------------------------------------------------------------------------|---------------------------|
| -      | SWE        | Execução desta instrução                                                      |                           |
| 0      | INT0       | Activação externa do pino INT0                                                | 3                         |
| 1      | INT1       | Activação externa do pino INT1                                                | 4                         |
| 2      | INT2       | Activação externa do pino INT2                                                | 5                         |
| 3      | INT3       | Activação externa do pino INT3                                                | 6                         |
| 4      | EXCESSO    | Excesso em operação aritmética                                                |                           |
| 5      | DIV0       | Divisão (DIV) por zero                                                        |                           |
| 6      | COD_INV    | Código de operação Inválido                                                   |                           |
| 7      | TEMPO1     | O Temporizador 1 chega ao fim da sua temporização em modo Contínuo            | 0                         |
| 8      | T1_FIM     | O Temporizador 1 chega ao seu valor máximo e volta a 0000H em modo Cronômetro | 0                         |
| 9      | CRONO      | A entrada C passa de 1 para 0                                                 |                           |
| 10     | TEMPO2     | O Temporizador 2 chega ao fim da sua temporização                             | 1                         |
| 11     | RX1_CHEGOU | Foi recebido um byte pela UART 1                                              | 7                         |
| 12     | RX2_CHEGOU | Foi recebido um byte pela UART 2                                              | 8                         |
| 13     | TX1_FIM    | A UART 1 acabou de transmitir um byte e está pronta para transmitir outro     | 13                        |
| 14     | TX2_FIM    | A UART 2 acabou de transmitir um byte e está pronta para transmitir outro     | 14                        |
| 15     | PORTO_A    | O valor do Porto A muda com este porto configurado para entrada activa        | 9                         |
| 16     | PORTO_B    | Idem, para o porto B                                                          | 10                        |
| 17     | PORTO_C    | Idem, para o porto C                                                          | 11                        |
| 18     | PORTO_D    | Idem, para o porto D                                                          | 12                        |

Fig. 5.1 - Esquema das dioceses e freguesias do concelho de Vila Franca de Xira



A Fig. B.2 mostra o fluxograma deste programa. Para ilustrar os dois modos de gestão das UARTs, a emissão dos bytes é controlada por teste (*polling*) do bit de estado da UART 1 (ETX1), enquanto a recepção dos bytes pela UART 2 utiliza uma interrupção (INT2\_CHANGE). Note-se que a passagem de 1 para 0 do bit C do Temporizador 1 também gera uma interrupção (CRONO), usada para ler o valor de contagem do tempo para um

Programa B.1 mostra a implementação deste fluxograma. Para poupar espaço, as constantes literais são usadas directamente nas instruções, mas as boas regras de programação indicam que se devem usar constantes simbólicas definidas no início do programa (tal como exemplificado pela constante N).

note-se que só é feito o teste de haver ou não mais bytes para transmitir depois de o

Note-se que só é feito o teste de haver ou não maus *bytes* para transmitir depois de o último ter sido completamente enviado, de modo a que o pino C só seja posto a 0 nessa altura e a interrupção CRONO obtenha o tempo total de transmissão. Se o teste fosse feito imediatamente após o envio de um *byte*, o tempo de envio do último *byte* praticamente não era contabilizado. Uma coisa é escrever um *byte* na UART para ser enviado, outra é este *byte* estar completamente enviado.

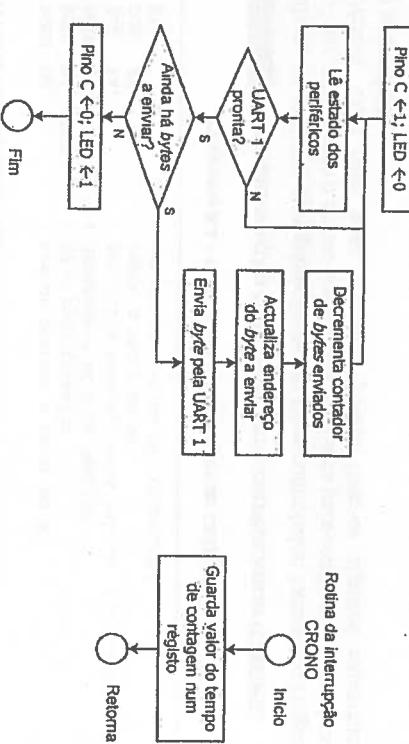
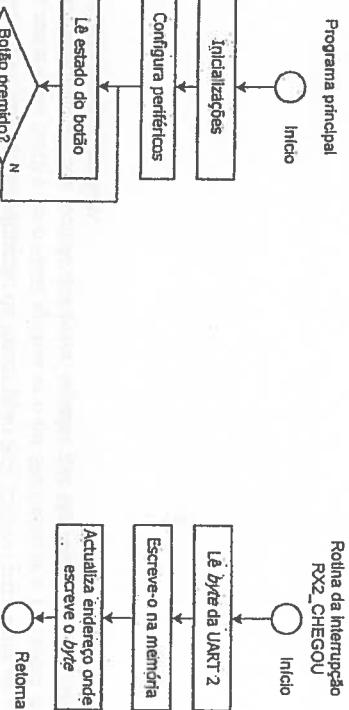
## B.5 EXEMPLO DE UTILIZAÇÃO

A Fig. representa um esquema de ligações do CREPE, incluindo os blocos internos relevantes para um exemplo de utilização simples que ilustra o funcionamento e a manipulação dos periféricos pelo programador.

O objectivo é transferir alguns *bytes* em memória para outra zona de memória, usando as duas UARTs como meio de comunicação e o Temporizador 1 em modo Cronómetro para medir o tempo de transferência. Esta operação começa quando se prime o interruptor.

## ARQUITECTURA DE COMPUTADORES

### APÊNDICE B - MANUAL DE PROGRAMADOR DO CREPE



**Fig. B.2 - Fluxograma do Programa B.1**

```

N EQU 14 ; número de bytes da cadeia de caracteres origem

PLACE 1000H ; localiza bloco de dados
; tabela de Excepções (ver Tabela B.9 para saber que excepções há)
int0: word Rot_nada ; endereço das rotinas das interrupções
      ; estas interrupções só são necessárias para
      ; deixar os endereços das rotinas de interrupção
      ; usadas no sítio correcto
excesso: word Rot_nada
div0: word Rot_nada
cod_inv: word Rot_nada
tempol: word Rot_nada
tl_fim: word Rot_nada
    
```

```

crono: word Rot_crono ; endereço da rotina da interrupção CRONO
tempo2: word Rot_nada
rx1_chegou: word Rot_nada
rot_rx2: word Rot_nada ; endereço da rotina da interrupção RX2_CHEGOU
origem: STRING "Teste do CREPE" ; cadeia de caracteres a enviar
PLACE 2000H ; localiza outro bloco de dados
destino: TABLE 20 ; reserva área para colocar caracteres recebidos
; localiza bloco de instruções
; inicializações
início: MOV SP, 2000H ; inicializa SP (pilha)
MOV R1, 000H ; inicializa apontador para a Tabela de Excepções
MOV R0, 000H ; ver Tabela B.3 para a definição dos bits
MOV RCN, R0 ; programa núcleo para relógio interno de 100 Hz
MOV R0, 003H ; ver Tabela B.4 para a definição dos bits
MOV RCP, R0 ; programa porta A para periférico de saída
MOV R0, 0007H ; ver Tabela B.5 para a definição dos bits
MOV RCT, R0 ; programa Temporizador 1 para modo Cronómetro
MOV R0, 311H ; ver Tabela B.6 para a definição dos bits
MOV RCJ, R0 ; programa UARTs (8 ciclos de relógio por bit)
; a partir de agora pode haver interrupções
lêBotão: EI ; lê porto B
MOV R0, RPB ; vê se o botão foi premido (bit 0 = 1?)
BIT R0, 0 ; se não, continua à espera
JZ lêBotão ; passa pino C para 1 (começa contagem de tempo no
            ; ... Temporizador 1) e acende o LED, no porto B
MOV R0, 3 ; número de bytes a ser transmitidos (cadeia origem)
MOV R2, N ; endereço da cadeia de caracteres a ser enviada
MOV R3, origem ; endereço da zona para colocar os bytes recebidos
MOV R4, destino ; endereço da zona para colocar os bytes recebidos
ciclo_envio: ; lê estado dos periféricos
    MOV R0, RCP ; vê se UART 1 está pronta a enviar (bit ETX1 = 1?)
    BIT R0, 5 ; se não, espera que acabe de enviar o byte anterior
    JZ ciclo_envio ; já enviou os bytes todos?
    CMP R2, 0 ; se sim, acabou
    JLE acabou ; vai buscar o próximo byte a enviar
    MOV R0, [R3] ; envia-o pela UART 1
    MOV RDUL, R0 ; menos um byte a enviar
    SUB R2, 0 ; endereço do byte a enviar a seguir
    ADD R3, 1 ; vai esperar que este byte acabe de ser enviado
    JMP ciclo_envio ; passa pino C para 0 (acaba a contagem de tempo) e
                    ; apaga o LED
acabou: MOV R0, 0 ; fim do programa
    MOV RPA, R0
    fim: JMP fim
    
```

\*\*\*\*\* Rot\_nada - Rotina de atendimento de interrupções não activas (só para preencher a Tabela de Excepções até à RX2\_CHEGOU). Se for invocada, é um erro.

```

;***** Rot_crone - Rotina de atendimento da interrupção CRONO.
; Rot_crone é chamada quando o bit C passa de 1 para 0 e o temporizador 1 está
; activo em modo Cronómetro.
;***** acaba: Rot_crone:
    PUSH R0 ; tem de guardar o valor do R0
    MOV R0, R71 ; vê quanto tempo demorou a transferência de bytes
    MOV R5 ; deixa esse valor em R5
    POP R0 ; reape o valor de R0
    REE ; retorna da rotina de interrupção

;***** Rot_RX2 - Rotina de atendimento da interrupção RX2_CHEGOU.
;***** Invocada quando a UART 2 acaba de receber um byte.
;***** acaba: Rot_RX2:
    PUSH R0 ; tem de guardar o valor do R0
    MOV R0, RDU2 ; lê o byte recebido
    MOVE [R4], R0 ; armazena-o na zona destino
    ADD R4, 1 ; actualiza endereço onde colocar o próximo byte
    POP R0 ; reape o valor de R0
    REE ; retorna da rotina de interrupção

```

**Programa B.1 - Exemplo de programação do CREPE****SIMULACRUM – PROGRAMAÇÃO COM UM MICROCONTROLADOR (CREPE)**

Esta simulação exemplifica a programação de um microcontrolador, tornando o Programa B.1 como base, explorando o acesso aos registos auxiliares para configuração e utilização dos periféricos. Com base neste programa podem fazer-se algumas experiências, incluindo as seguintes:

- Configurar a duração de cada bit de forma diferente em cada UART e ver qual a máxima diferença até a recepção deixar de ser correcta;
- Modificar a rotina de atendimento da interrupção RX2\_CHEGOU para antes de ler o RDU2 testar o bit SRX2 e ler o RDU2 apenas se o bit SRX2 estiver a 1 (Tabela B.8). O resultado é ler apenas metade dos bytes, aqueles que são recebidos na UART 2 sem o byte anterior ser lido.

O desenho de circuitos é uma tarefa complexa cujo resultado físico final (*hardware*) não é facilmente modificável. Por este motivo, a utilização de simuladores para validar e testar o desenho de novos circuitos é desde há muito uma tarefa fundamental para qualquer arietado de computadores.

O simulador de circuitos SIMAC (SIMulator de Arquitectura de Computadores) destina-se ao ensino de arquitectura de computadores e não ao desenvolvimento e implementação de arquitecturas. Por este motivo, algumas das tarefas não são tão automatizadas como o seriam num simulador comercial, os detalhes eléctricos não são tomados em consideração (o simulador é puramente digital) e os módulos que disponibiliza não correspondem a nenhum circuito físico existente no mercado.

O simulador é uma aplicação construída em Java™ com uma interface gráfica para desenhar e simular os circuitos. O ciclo de vida de um circuito tem duas fases distintas: desenho e simulação. A cada uma destas fases corresponde um painel gráfico com características distintas.

O simulador é genérico (e não específico do PEPE) e podem facilmente incorporar-se novos módulos de funcionalidade arbitrariamente complexa, desde que se programem em Java™ e respeitem as convenções de integração no simulador.

Este apêndice faz apenas uma breve introdução à funcionalidade e capacidades do simulador, nomeadamente as fases de desenho e de simulação e a utilização de alguns módulos complexos que podem ser utilizados no desenho de circuitos.

Dada a evolução natural de todos os sistemas, é natural que algumas características do simulador em si, disponível no site de apoio do livro ([www.fca.pt](http://www.fca.pt)) sejam diferentes do que aqui é descrito. Os princípios básicos, no entanto, mantêm-se válidos. Em caso de discrepância, o site de apoio tem a documentação mais recente.

## APÊNDICE C INTRODUÇÃO AO SIMULADOR (SIMAC)

### C.1 DESENHO DE CIRCUITOS

O objectivo da fase de desenho é criar um circuito através da interligação de instâncias de um conjunto de módulos predefinidos que o simulador disponibiliza. A biblioteca de módulos do SIMAC contém desde simples portas lógicas até módulos complexos como processadores (PEPE, por exemplo).

Para facilitar a demonstração das capacidades do simulador, quer durante a fase de desenho, quer durante a fase de simulação, este apêndice utiliza um circuito com uma funcionalidade semelhante à do circuito da Fig. 3.4, na página 121, que contém uma memória, um somador e um registo. Com ele é possível somar vários números contidos na memória e guardar o resultado final no registo.

No circuito a desenhar no SIMAC (Fig. C.1), a memória utilizada é uma PROM (de 16 bits de largura com 16 bits de endereço, embora pudéssemos usar uma mais pequena), para não ser necessário escrever primeiro os dados (basta programar a PROM). O somador e o registo (de basculas D) também são de 16 bits. O somador tem um bit de *carry in* e outro de *carry out* para suportar o transporte da soma.

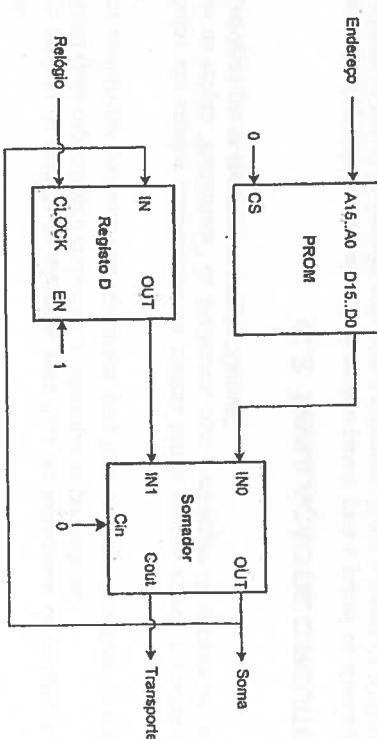


Fig. C.1 - Circuito a desenhar no SIMAC

A primeira tarefa a executar para construir o circuito da Fig. C.1 é localizar os módulos que são necessários, na interface descrita na Fig. C.2. Os módulos disponíveis estão descritos na janela da biblioteca de módulos (IC Library, ou biblioteca de circuitos integrados, como se de uma implementação física se tratasse). No presente caso são necessários os módulos "Somador", "Registo D", e "PROM". De seguida, é necessário colocar estes módulos no painel onde se irá construir o circuito (zona com a grelha de pontos à esquerda da biblioteca de módulos), fazendo clique sobre o nome do somador na biblioteca e arrastando o cursor para a grelha no local onde se deseja colocar o somador. Após esta operação, deve-se proceder da mesma forma para os outros dois módulos. Quem o somador quer os outros módulos manipulam, por omissão, valores de 1 bit. Para configurar cada um dos módulos de acordo com a Fig. C.1 é necessário efectuar um duplo clique sobre cada um dos módulos e ter acesso à sua janela de configuração própria. Nessa janela podemos configurar o somador para valores de 16 bit, bem como configurar os barramentos de endereços e dados da PROM com 16 bit.

O passo seguinte consiste em interligar os módulos. Cada módulo tem um ou vários portos (pontos de ligação), que podem ser de entrada, de saída, de saída com *tristate* ou

Para interligar dois portos (do mesmo módulo ou de módulos diferentes), é necessário fazer clique sobre um dos portos e arrastar o cursor até ao outro.

Para interligar dois portos (do mesmo módulo ou de módulos diferentes), é necessário fazer clique sobre um dos portos e arrastar o cursor até ao outro.

Para interligar dois portos (do mesmo módulo ou de módulos diferentes), é necessário fazer clique sobre um dos portos e arrastar o cursor até ao outro.

Para interligar dois portos (do mesmo módulo ou de módulos diferentes), é necessário fazer clique sobre um dos portos e arrastar o cursor até ao outro.

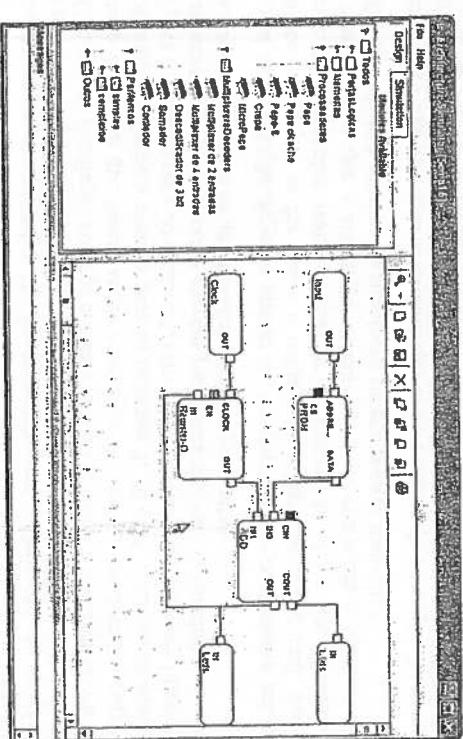


Fig. C.2 - Circuito da Fig. C.1, desenhado no SIMATIC.

Do ponto de vista de desenho, todos os tipos de portos são iguais. Não é feita qualquer verificação sobre a compatibilidade dos portos interligados. Assim, se forem interligados dois portos de saída, muito provavelmente irá ocorrer um erro durante a simulação (quando as duas saídas tentarem impor valores diferentes para a ligação).

Para interligar vários portos é necessário efectua-lo dois a dois, embora o resultado final, do ponto de vista de simulação, seja uma única ligação a interligar todos os portos envolvidos.

Cada módulo possui uma janela de configuração que pode ser acedida através de um duplo clique sobre o módulo. A janela de configuração varia de módulo para módulo. Por exemplo, no caso das portas lógicas simples é possível configurar o número de bits com que opera (número de *bits* de cada um dos portos). Todas as instâncias (dos módulos predefinidos) criadas pelo utilizador permitem mudar o seu nome e a configuração de

Por omisão, o 1.º bit do porto liga ao 1.º bit da ligação, o 2.º ao 2.º e assim sucessivamente até à dimensão máxima do porto. Mas é também possível, por configuração do porto (na janela de configuração do módulo), indicar que se pretende ligar outros bits do porto e da ligação e indicar um máximo de bits a ligar. Por exemplo, se se indicar que se

quer ligar o 3.º bit do porto ao 5.º da ligação, num máximo de 3 bits, serão ligados os bits  $P_2, P_3, P_4$  do porto aos bits  $L_4, L_5, L_6$  da ligação.<sup>123</sup>

Nalguns casos, é necessário ligar um porto de entrada directamente a 0 ou a 1. Por exemplo, na Fig. C.1 o pino EN (*Enable*) do registo deve estar a 1, enquanto que o CS (*Chip Select*) da PROM e o Cin (*Carry In*) do somador estão ligados directamente a 0. Se estes portos não forem ligados a nada, o valor lido pelos módulos é aleatório e pode ocasionar um funcionamento incorrecto. Para ligar um porto a 0 ou 1 é necessário utilizar a janela de configuração do porto e indicar que esse porto deve estar fixo a 0 ou 1. Os portos ligados directamente a 0 aparecem a preto no esquemático, enquanto os portos ligados a 1 aparecem a vermelho.

Para que seja possível testar o circuito, é necessário adicionar-lhe módulos de entrada e de saída de informação. Ao circuito da Fig. C.2 é necessário adicionar:

- Um módulo com um conjunto de 16 LEDs para observar o resultado à saída do somador;
- Outro módulo só com um LED para observar a saída Cout (*Carry Out*) do somador;
- Um módulo de entrada de números, para ligar ao barramento de endereços da PROM (para especificar o endereço da célula que se pretende ler);
- Um módulo que gera uma onda quadrada, para ligar ao porto do relógio do registo.

Estes módulos são criados exactamente da mesma forma que os outros e podem ser encontrados na secção "Periféricos" da biblioteca, com a exceção do gerador de onda quadrada (relógio) que se encontra na secção "Outros".

## C.2 SIMULAÇÃO DE CIRCUITOS

Para iniciar a simulação de um circuito, é necessário passar para o painel de simulação, que é semelhante ao painel de desenho mas com a diferença de que a janela da biblioteca de módulos é substituída por uma janela com um conjunto de botões de controlo da simulação e duas caixas com a identificação dos pontos de paragem de depuração (*breakpoints*) e com os eventos a tratar pela simulação.

O núcleo do simulador é um motor de tratamento de eventos que funciona por ciclos, ou interações. Cada interação ocorre integralmente no mesmo tempo de simulação e compreende três fases (executadas em ciclo enquanto a simulação estiver a correr):

- Ligações – Analisam-se todas as ligações do circuito para verificar quais as que mudaram de valor desde a última interação (quando a simulação arranca, considera-se que todas as ligações mudaram de valor). Em cada ligação que mudou de

valor, verifica-se que portos lá ligam e marcam-se os módulos a que esses portos pertencem como necessitando de ser recalcados;

**Módulos** – Percorrem-se todos os módulos e os marcados na fase anterior são executados, o que implica normalmente ler os portos de entrada e produzir valores para os portos de saída. No entanto, a escrita não é feita directamente, pois é preciso simular os tempos de atraso de cada módulo. Assim, de cada vez que um módulo pretende alterar o valor de um dos seus portos, gera um evento (com indicação do tempo em que deve ocorrer e do valor a escrever no porto) que é acrescentado à lista de eventos que irão ocorrer no futuro;

**Eventos** – A lista de eventos está organizada cronologicamente, pelo que na sua cabeça está o evento no tempo mais próximo em que um evento está programado. Assim, o tempo de simulação é avançado logo para esse tempo (uma vez que até lá nada ocorreu) e todos os eventos programados para esse tempo são executados. Os de um tempo posterior são retidos para futura execução. A execução de eventos pode alterar o valor de algumas ligações.

Depois da última fase repete-se o ciclo. A simulação só pára quando o utilizador a interrompe ou se acabam os eventos. Os módulos de entrada de informação podem gerar eventos quando o utilizador carrega em botões com o rato. Também há módulos de relógio que geram eventos de forma periódica.

Os tempos de atraso dos módulos são fundamentais para a definição do conceito de tempo de simulação, que é independente do tempo real durante o qual se está a simular o circuito. O tempo de simulação começa em zero quando se inicia a simulação e vai sendo incrementado à medida que forem sendo resolvidos os eventos escalonados para cada instante. Assim, se existirem eventos para os instantes 5, 20 e 35, o tempo de simulação salta de 0 para 5 e depois para 20 e para 35 sem nunca passar pelos instantes intermédios (excepto se entre tanto forem gerados eventos para esses tempos). O simulador só executa operações nos instantes para os quais existem eventos para tratar.

O tempo real de simulação depende apenas do desempenho do computador em que a simulação corre (excepto se for usado o módulo de relógio de tempo real, que gera eventos com temporizações reais). Aumentar os tempos de atraso dos módulos não aumenta o tempo real que a simulação demora, pois as operações a efectuar são as mesmas. O único efeito é o tempo de simulação subir a valores mais altos.

A caixa de eventos mostra a lista de eventos que estão à espera de execução, bem como o respectivo instante para o qual estão escalonados, os portos que vão alterar e o novo valor de cada porto. Por motivos de desempenho, a lista de eventos só é mostrada quando o simulador está parado.

A caixa de *breakpoints* (pontos de paragem) contém todos os *breakpoints* inseridos pelos diferentes módulos. Podem-se inserir *breakpoints*:

- Sobre o valor de uma ligação – É apenas necessário seleccionar a ligação, fazendo clique sobre ela, e pressionar o botão de adicionar novo *breakpoint* para quando a ligação for igual ou diferente de um determinado valor;

<sup>123</sup> O primeiro bit dos portos e das ligações é o bit número 0.

- Nos módulos, de forma específica – Por exemplo, basta fazer clique numa dada instrução na janela de interface do PEPE, para a simulação parar quando o processador fizer a busca da instrução no endereço selecionado. Desta modo, é possível colocar *breakpoints* no código do programa que o processador executa.

Para interagir com um módulo durante a simulação, é necessário ter acesso à sua interface, para o que basta efectuar um duplo clique sobre os respectivos módulos no painel de simulação.<sup>124</sup> A Fig. C.3 ilustra algumas destas interfaces (as dos módulos da Fig. C.2). De cima para baixo e da esquerda para a direita, são apresentadas as interfaces do relógio, do módulo de entrada de números, dos LEDs ligados à saída do somador, do LED ligado ao Cout do somador e da PROM. Esta última é necessária para introduzir os números a somar. Para colocar um valor à saída do módulo de entrada de dados é apenas necessário introduzir um número na caixa seguido de Enter. Os LEDs aparecem a verde quando o valor é 0 e vermelho quando o valor é 1. Caso o bit não esteja ligado ou esteja ligado a um porto em alta impedância (tristate), o respectivo LED aparece a cinzento.

iteração seguinte é necessário pressionar de novo o botão de passo a passo. Esta evolução controlada da simulação e a caixa de eventos por executar são as ferramentas por excelência para analisar com detalhe tudo o que se passa no simulador.

Existem ainda outros módulos com uma interface com o utilizador, como por exemplo as RAMs, o processador PEPE, uma variante deste, designada MicroPEPE, que permite observar o interior do processador, e o microcontrolador CREPE. A interface do MicroPEPE está ilustrada na Fig. C.4, sendo possível ver (e alterar) os registos principais e auxiliares, os bits do RE, o programa em linguagem *assembly* e a tabela de símbolos (e respectivos valores).

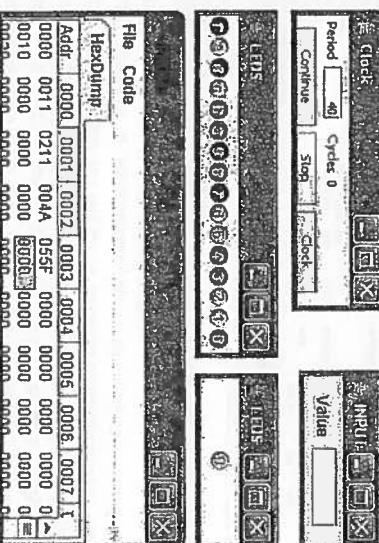


Fig. C.3 - Interfaces dos módulos da Fig. C.2

Depois de dar inicio à simulação, o relógio permite iniciar a geração de uma onda quadrada de período 40, o que conduz ao acumular dos valores da PROM no registo. Para ver o resultado do registo é necessário parar a simulação. Para ver o resultado de cada soma que é armazenada no registo, é possível utilizar o botão Clock do relógio para gerar um único ciclo de cada vez, e assim analisar cada uma das operações de adição.

E também possível executar a simulação passo a passo. Nesse caso, o simulador só executa os eventos de um única iteração, parando de seguida. Para executar os eventos da

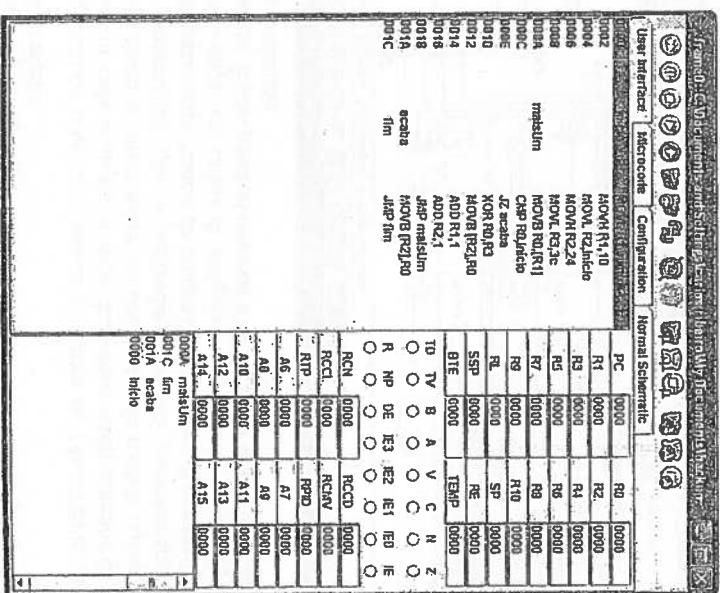


Fig. C.4 - Interface do MicroPEPE

Este painel do MicroPEPE é muito semelhante ao do PEPE e do CREPE, variando apenas o número de botões disponíveis na barra de botões no topo e os painéis disponíveis ao lado do painel "User interface". Os últimos seis botões só estão disponíveis na versão MicroPEPE e permitem carregar/gravar/reiniciar um novo microcódigo ou uma nova configuração para a ROM de mapeamento. Os restantes botões existem em todos os modelos e permitem executar, parar, reiniciar, avançar uma instrução, compilar e carregar

um programa, ligar/desligar o relógio interno e gerar um ciclo de relógio interno (útil para ver a evolução do microcódigo).

A interface do CREPE é muito semelhante à do PEPE. As únicas diferenças residem na existência de um painel adicional para mostrar o conteúdo da memória interna do CREPE e no número de registos auxiliares utilizados na configuração, que existem em maior número no CREPE. Já o MicroPEPE possui três painéis adicionais:

- Microcódigo do processador;

- Conteúdo da ROM de mapeamento;

- Esquemático interno do processador.

O painel do microcódigo (Fig. C.5) mostra, de forma simbólica, as microinstruções do processador. A primeira coluna contém o endereço de cada microinstrução. A segunda coluna contém o nome simbólico pelo qual a instrução é conhecida, para que possa ser referenciada para saltos, quer por outras microinstruções, quer pela ROM de mapeamento. A microinstrução assinalada com a barra escura indica qual a microinstrução em execução, aquela cujo endereço está no MPC. A terceira coluna contém um comentário em formato RTL que descreve a microinstrução e as restantes contêm os códigos simbólicos de cada um dos campos das microinstruções.

| Endereço | Nome    | Comentário                         |
|----------|---------|------------------------------------|
| 00000000 | RESERVA |                                    |
| 00000001 | MULPC   | TYPE = ALU; OP = MUL; PC = MPC - 1 |
| 00000002 | RESERVA |                                    |
| 00000003 | RESERVA |                                    |
| 00000004 | RESERVA |                                    |
| 00000005 | RESERVA |                                    |
| 00000006 | RESERVA |                                    |
| 00000007 | RESERVA |                                    |
| 00000008 | RESERVA |                                    |
| 00000009 | RESERVA |                                    |
| 0000000A | RESERVA |                                    |
| 0000000B | RESERVA |                                    |
| 0000000C | RESERVA |                                    |
| 0000000D | RESERVA |                                    |
| 0000000E | RESERVA |                                    |
| 0000000F | RESERVA |                                    |
| 00000010 | RESERVA |                                    |
| 00000011 | RESERVA |                                    |
| 00000012 | RESERVA |                                    |
| 00000013 | RESERVA |                                    |
| 00000014 | RESERVA |                                    |
| 00000015 | RESERVA |                                    |
| 00000016 | RESERVA |                                    |
| 00000017 | RESERVA |                                    |
| 00000018 | RESERVA |                                    |
| 00000019 | RESERVA |                                    |
| 0000001A | RESERVA |                                    |
| 0000001B | RESERVA |                                    |
| 0000001C | RESERVA |                                    |
| 0000001D | RESERVA |                                    |
| 0000001E | RESERVA |                                    |
| 0000001F | RESERVA |                                    |
| 00000020 | RESERVA |                                    |
| 00000021 | RESERVA |                                    |
| 00000022 | RESERVA |                                    |
| 00000023 | RESERVA |                                    |
| 00000024 | RESERVA |                                    |
| 00000025 | RESERVA |                                    |
| 00000026 | RESERVA |                                    |
| 00000027 | RESERVA |                                    |
| 00000028 | RESERVA |                                    |
| 00000029 | RESERVA |                                    |
| 0000002A | RESERVA |                                    |
| 0000002B | RESERVA |                                    |
| 0000002C | RESERVA |                                    |
| 0000002D | RESERVA |                                    |
| 0000002E | RESERVA |                                    |
| 0000002F | RESERVA |                                    |
| 00000030 | RESERVA |                                    |
| 00000031 | RESERVA |                                    |
| 00000032 | RESERVA |                                    |
| 00000033 | RESERVA |                                    |
| 00000034 | RESERVA |                                    |
| 00000035 | RESERVA |                                    |
| 00000036 | RESERVA |                                    |
| 00000037 | RESERVA |                                    |
| 00000038 | RESERVA |                                    |
| 00000039 | RESERVA |                                    |
| 0000003A | RESERVA |                                    |
| 0000003B | RESERVA |                                    |
| 0000003C | RESERVA |                                    |
| 0000003D | RESERVA |                                    |
| 0000003E | RESERVA |                                    |
| 0000003F | RESERVA |                                    |
| 00000040 | RESERVA |                                    |
| 00000041 | RESERVA |                                    |
| 00000042 | RESERVA |                                    |
| 00000043 | RESERVA |                                    |
| 00000044 | RESERVA |                                    |
| 00000045 | RESERVA |                                    |
| 00000046 | RESERVA |                                    |
| 00000047 | RESERVA |                                    |
| 00000048 | RESERVA |                                    |
| 00000049 | RESERVA |                                    |
| 0000004A | RESERVA |                                    |
| 0000004B | RESERVA |                                    |
| 0000004C | RESERVA |                                    |
| 0000004D | RESERVA |                                    |
| 0000004E | RESERVA |                                    |
| 0000004F | RESERVA |                                    |
| 00000050 | RESERVA |                                    |
| 00000051 | RESERVA |                                    |
| 00000052 | RESERVA |                                    |
| 00000053 | RESERVA |                                    |
| 00000054 | RESERVA |                                    |
| 00000055 | RESERVA |                                    |
| 00000056 | RESERVA |                                    |
| 00000057 | RESERVA |                                    |
| 00000058 | RESERVA |                                    |
| 00000059 | RESERVA |                                    |
| 0000005A | RESERVA |                                    |
| 0000005B | RESERVA |                                    |
| 0000005C | RESERVA |                                    |
| 0000005D | RESERVA |                                    |
| 0000005E | RESERVA |                                    |
| 0000005F | RESERVA |                                    |
| 00000060 | RESERVA |                                    |
| 00000061 | RESERVA |                                    |
| 00000062 | RESERVA |                                    |
| 00000063 | RESERVA |                                    |
| 00000064 | RESERVA |                                    |
| 00000065 | RESERVA |                                    |
| 00000066 | RESERVA |                                    |
| 00000067 | RESERVA |                                    |
| 00000068 | RESERVA |                                    |
| 00000069 | RESERVA |                                    |
| 0000006A | RESERVA |                                    |
| 0000006B | RESERVA |                                    |
| 0000006C | RESERVA |                                    |
| 0000006D | RESERVA |                                    |
| 0000006E | RESERVA |                                    |
| 0000006F | RESERVA |                                    |
| 00000070 | RESERVA |                                    |
| 00000071 | RESERVA |                                    |
| 00000072 | RESERVA |                                    |
| 00000073 | RESERVA |                                    |
| 00000074 | RESERVA |                                    |
| 00000075 | RESERVA |                                    |
| 00000076 | RESERVA |                                    |
| 00000077 | RESERVA |                                    |
| 00000078 | RESERVA |                                    |
| 00000079 | RESERVA |                                    |
| 0000007A | RESERVA |                                    |
| 0000007B | RESERVA |                                    |
| 0000007C | RESERVA |                                    |
| 0000007D | RESERVA |                                    |
| 0000007E | RESERVA |                                    |
| 0000007F | RESERVA |                                    |
| 00000080 | RESERVA |                                    |
| 00000081 | RESERVA |                                    |
| 00000082 | RESERVA |                                    |
| 00000083 | RESERVA |                                    |
| 00000084 | RESERVA |                                    |
| 00000085 | RESERVA |                                    |
| 00000086 | RESERVA |                                    |
| 00000087 | RESERVA |                                    |
| 00000088 | RESERVA |                                    |
| 00000089 | RESERVA |                                    |
| 0000008A | RESERVA |                                    |
| 0000008B | RESERVA |                                    |
| 0000008C | RESERVA |                                    |
| 0000008D | RESERVA |                                    |
| 0000008E | RESERVA |                                    |
| 0000008F | RESERVA |                                    |
| 00000090 | RESERVA |                                    |
| 00000091 | RESERVA |                                    |
| 00000092 | RESERVA |                                    |
| 00000093 | RESERVA |                                    |
| 00000094 | RESERVA |                                    |
| 00000095 | RESERVA |                                    |
| 00000096 | RESERVA |                                    |
| 00000097 | RESERVA |                                    |
| 00000098 | RESERVA |                                    |
| 00000099 | RESERVA |                                    |
| 0000009A | RESERVA |                                    |
| 0000009B | RESERVA |                                    |
| 0000009C | RESERVA |                                    |
| 0000009D | RESERVA |                                    |
| 0000009E | RESERVA |                                    |
| 0000009F | RESERVA |                                    |
| 000000A0 | RESERVA |                                    |
| 000000A1 | RESERVA |                                    |
| 000000A2 | RESERVA |                                    |
| 000000A3 | RESERVA |                                    |
| 000000A4 | RESERVA |                                    |
| 000000A5 | RESERVA |                                    |
| 000000A6 | RESERVA |                                    |
| 000000A7 | RESERVA |                                    |
| 000000A8 | RESERVA |                                    |
| 000000A9 | RESERVA |                                    |
| 000000AA | RESERVA |                                    |
| 000000AB | RESERVA |                                    |
| 000000AC | RESERVA |                                    |
| 000000AD | RESERVA |                                    |
| 000000AE | RESERVA |                                    |
| 000000AF | RESERVA |                                    |
| 000000B0 | RESERVA |                                    |
| 000000B1 | RESERVA |                                    |
| 000000B2 | RESERVA |                                    |
| 000000B3 | RESERVA |                                    |
| 000000B4 | RESERVA |                                    |
| 000000B5 | RESERVA |                                    |
| 000000B6 | RESERVA |                                    |
| 000000B7 | RESERVA |                                    |
| 000000B8 | RESERVA |                                    |
| 000000B9 | RESERVA |                                    |
| 000000BA | RESERVA |                                    |
| 000000BB | RESERVA |                                    |
| 000000BC | RESERVA |                                    |
| 000000BD | RESERVA |                                    |
| 000000BE | RESERVA |                                    |
| 000000BF | RESERVA |                                    |
| 000000C0 | RESERVA |                                    |
| 000000C1 | RESERVA |                                    |
| 000000C2 | RESERVA |                                    |
| 000000C3 | RESERVA |                                    |
| 000000C4 | RESERVA |                                    |
| 000000C5 | RESERVA |                                    |
| 000000C6 | RESERVA |                                    |
| 000000C7 | RESERVA |                                    |
| 000000C8 | RESERVA |                                    |
| 000000C9 | RESERVA |                                    |
| 000000CA | RESERVA |                                    |
| 000000CB | RESERVA |                                    |
| 000000CC | RESERVA |                                    |
| 000000CD | RESERVA |                                    |
| 000000CE | RESERVA |                                    |
| 000000CF | RESERVA |                                    |
| 000000D0 | RESERVA |                                    |
| 000000D1 | RESERVA |                                    |
| 000000D2 | RESERVA |                                    |
| 000000D3 | RESERVA |                                    |
| 000000D4 | RESERVA |                                    |
| 000000D5 | RESERVA |                                    |
| 000000D6 | RESERVA |                                    |
| 000000D7 | RESERVA |                                    |
| 000000D8 | RESERVA |                                    |
| 000000D9 | RESERVA |                                    |
| 000000DA | RESERVA |                                    |
| 000000DB | RESERVA |                                    |
| 000000DC | RESERVA |                                    |
| 000000DD | RESERVA |                                    |
| 000000DE | RESERVA |                                    |
| 000000DF | RESERVA |                                    |
| 000000E0 | RESERVA |                                    |
| 000000E1 | RESERVA |                                    |
| 000000E2 | RESERVA |                                    |
| 000000E3 | RESERVA |                                    |
| 000000E4 | RESERVA |                                    |
| 000000E5 | RESERVA |                                    |
| 000000E6 | RESERVA |                                    |
| 000000E7 | RESERVA |                                    |
| 000000E8 | RESERVA |                                    |
| 000000E9 | RESERVA |                                    |
| 000000EA | RESERVA |                                    |
| 000000EB | RESERVA |                                    |
| 000000EC | RESERVA |                                    |
| 000000ED | RESERVA |                                    |
| 000000EE | RESERVA |                                    |
| 000000EF | RESERVA |                                    |
| 000000F0 | RESERVA |                                    |
| 000000F1 | RESERVA |                                    |
| 000000F2 | RESERVA |                                    |
| 000000F3 | RESERVA |                                    |
| 000000F4 | RESERVA |                                    |
| 000000F5 | RESERVA |                                    |
| 000000F6 | RESERVA |                                    |
| 000000F7 | RESERVA |                                    |
| 000000F8 | RESERVA |                                    |
| 000000F9 | RESERVA |                                    |
| 000000FA | RESERVA |                                    |
| 000000FB | RESERVA |                                    |
| 000000FC | RESERVA |                                    |
| 000000FD | RESERVA |                                    |
| 000000FE | RESERVA |                                    |
| 000000FF | RESERVA |                                    |

Fig. C.5 – Painel de Microcódigo do MicroPEPE

Este painel permite não só a visualização dos códigos como a alteração de cada um dos campos de forma simbólica. É possível guardar as alterações ao microcódigo num ficheiro aparte, que ficará associado a esta arquitectura (sempre que se abrir a arquitectura, o módulo do MicroPEPE irá ler as suas microinstruções desse ficheiro).

Fig. C.6 – Esquemático interno do MicroPEPE, com sondas em pontos específicos

A maioria dos módulos pára sempre que o tempo de simulação pára, pois é o motor do simulador que executa as operações de reavaliação nesses módulos, mas existem módulos que têm motores independentes, controlados pelo relógio real do computador em que a simulação corre. Estes módulos são geralmente de controlo, como por exemplo um elevador ou uma pista de combolos, e têm como objectivo criar um cenário de controlo mais real, permitindo ao programa que corre no PEPE implementar aplicações interactivas.

vas e de tempo real. Por exemplo, se o simulador para e deixa de controlar o elevador (sem parar o motor) quando este está a descer, este continua a descer até bater no fundo.

Um dos módulos que possui um motor independente é um gerador de onda quadrada de tempo real, com um período que pode ser especificado em milissegundos. Desta forma, é possível controlar o ritmo da simulação, ao contrário do que sucede com um relógio normal, que trabalha apenas em tempo de simulação. Por exemplo, se se criar um circuito com apenas um LED ligado à saída de um relógio normal, o utilizador observará um comportamento errático e demasiado rápido do LED. Tal facto resulta da elevada frequência com que o circuito funciona e que impede o LED de acompanhar todas as alterações do sinal do relógio. Como os únicos eventos são os gerados pelo relógio, o tempo de simulação saíta de meio período em meio período (nas transições do sinal do relógio), pelo que a frequência com que muda não depende do período mas sim do desempenho do computador onde se está a efectuar a simulação. Com um relógio de tempo real, os eventos só são gerados quando o motor interno desse módulo o determinar, de onde resulta um efeito de pisca-pisca regular no LED.

## APÊNDICE D COMPUTAÇÃO EM VÍRGULA FLUTUANTE

Os números inteiros e a sua representação em binário (secção 2.7, na página 87) são fundamentais ao funcionamento de qualquer computador. No entanto, há inúmeras classes de aplicações que requerem o uso de números reais, frequentemente com grande precisão. Por outro lado, há números inteiros cuja grandeza não os permite representar nos *bits* da palavra do computador.

A norma IEEE 754 [IEEE 1985] define o formato de representação dos números em vírgula flutuante e permite resolver estes problemas. Esta representação assume que se usam alguns bits da palavra do computador para representar os dígitos mais significativos do número (que definem a precisão da representação) e outros para indicar o expoente de um factor multiplicativo (que define a escala de grandeza do número). A gama de representação de números é drasticamente maior do que a dos números inteiros, embora possam não ser representados todos os seus dígitos.

A generalidade dos processadores actuais (com exceção para os mais pequenos, típicos dos sistemas embedidos) inclui ALUs com suporte para operações com números em vírgula flutuante, em conformância com o IEEE 754. O PEPE não suporta estas operações, pois são já bastante complexas e excedem o nível introdutório deste livro. Este apêndice faz uma brevíssima introdução a este tema, cuja inclusão (face a outros) se justifica pela sua importância para muitas aplicações e pelo seu suporte generalizado no mercado dos processadores, constituindo um ponto de partida para a consulta de bibliografia mais especializada [Patterson 2008, Stallings 2006].

### D.1 REPRESENTAÇÃO EM VÍRGULA FLUTUANTE

Os números não inteiros (com vírgula e parte fraccionária) podem ser representados em duas notações básicas (em qualquer base, incluindo decimal e binária):

- **Vírgula fixa** – A vírgula está imediatamente à direita do algarismo das unidades (que é onde deve estar, por definição), dividindo o número em duas partes, inteira e fraccionária. Os números -12,375 e 0,0789 exemplificam esta notação;

- Vírgula flutuante<sup>125</sup>** – A vírgula pode estar em qualquer posição, desde que se compense a mudança de posição face à posição fixa com a multiplicação do número com a posição da vírgula modificada (designado significando<sup>126</sup>) por uma potência adequada (expoente) na base usada. Na prática, o que interessa é a notação em vírgula flutuante normalizada, em que a vírgula está imediatamente à direita do primeiro algarismo não zero (excepto se o número representado for zero). Em base 10, os números seriam  $-1,2375 \times 10^1$  e  $7,89 \times 10^2$ .

Um número N representa-se em vírgula flutuante normalizada na base B por:

$$N = \pm \text{significando} \times B^{\text{expoente}}$$

em que  $1 \leq \text{significando} < B$  (excepcionalmente o caso de o número ser zero).

Ao contrário da representação dos números inteiros, que se faz em complemento para 2 (secção 2.7.6, na página 95), em vírgula flutuante usa-se a notação em sinal e grandeza, pois facilita as operações aritméticas, bastante mais complexas do que as de números inteiros. Os processadores têm duas ALUs (cada uma com o seu banco de registo), uma interira e outra de vírgula flutuante, e instruções separadas para as operações de vírgula flutuante, tipicamente com as mnemónicas das instruções aritméticas interiras precedidas de "F" (FADD, FSUB, FMUL, FDIV, etc.).

A conversão de um número não inteiro de base decimal para binário (em vírgula flutuante) pode ser efectuada recorrendo à decomposição de um número em algarismos, válido para qualquer base, já apresentada na página 88, mas estendendo-a para posições menores que 0 (parte fraccionária):

$$(a_2 a_1 a_0 a_{-1} a_{-2} a_{-3})_B = (a_2 \times B^2) + (a_1 \times B^1) + (a_0 \times B^0) + (a_{-1} \times B^{-1}) + (a_{-2} \times B^{-2}) + (a_{-3} \times B^{-3})$$

em que em base B os factores para a esquerda da vírgula são  $B^0=1$ ,  $B^1=2$ ,  $B^2=4$ , etc., e para a direita são  $B^{-1}=0,5$ ,  $B^{-2}=0,25$ ,  $B^{-3}=0,125$ , etc.

Assim, o número atrás usado (-12,375) pode ser convertido desta forma:

$$-12,375 = 8+4+0,25+0,125 = 2^3+2^2+2^{-1}+2^{-3} = -1100,011$$

Note-se a representação em sinal e grandeza e não em complemento para 2. Para normalizar o número, basta andar com a vírgula para a esquerda e compensar com a potência adequada (tal como se faria em base 10):

$$-1100,011 = -1,100011 \times 2^3$$

A grande vantagem da notação em vírgula flutuante (já só considerando a versão normalizada) é separar a precisão (número de algarismos significativos) da grandeza (expoente da base). Por exemplo, 2,74036219473294657394785678 é um número pequeno (inferior a 3) mas requer grande precisão (muitos algarismos). Pelo contrário, o número 2.700.000.000.000.000 tem um valor grande mas só necessita de 2 algarismos significativos. Gasta-se menos informação com a representação  $2,7 \times 10^{21}$  do que a especificar todos os zeros explicitamente. Note-se que 0,000000000000000027 tem a mesma precisão que o número anterior mas uma grandeza bem mais pequena ( $2,7 \times 10^{-21}$ ). Uma representação em vírgula fixa gastaria demasiados bits com os zeros à esquerda ou à direita dos algarismos significativos. Com a notação em vírgula flutuante, números muito pequenos ou muito grandes podem ser representados com uma precisão adequada.

Note-se que um número inteiro também pode ser representado em vírgula flutuante. Basta encará-lo como um número real sem algarismos significativos para a direita da vírgula, com o benefício de uma gama de valores representáveis muito maior (embora os limites da precisão impeçam os números muito grandes de serem representados com o valor exacto).

## D.2 A NORMA IEEE 754

No entanto, o número de bits para representar quer o significando quer o expoente é limitado, estabelecendo os valores extremos para a precisão e a grandeza, respectivamente. A norma IEEE 754 define dois formatos, simples e duplo, com 32 e 64 bits, respectivamente (Fig. D.1). Se o significando tiver mais bits significativos do que os que cabem no campo respectivo, o valor tem de ser arredondado (convertido para o valor representável mais próximo).<sup>127</sup> Se a grandeza for tão grande que o número não seja representável há um excesso (overflow). Se grandeza for demasiado pequena para ser representada, ocorre um défice (underflow).

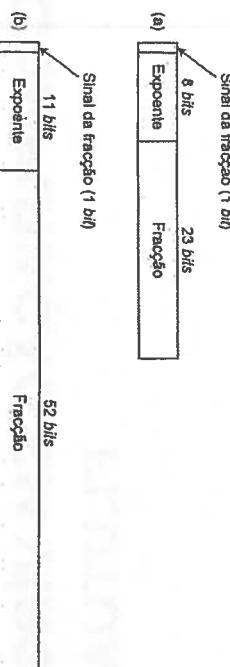


Fig. D.1 - Formatos dos números em vírgula flutuante definidos pela norma IEEE 754.  
(a) – Precisão simples (32 bits); (b) – Precisão dupla (64 bits)

<sup>125</sup> Na base 10, esta notação também se designa por notação científica, sendo comum nas máquinas de calcular.

<sup>126</sup> Nome na língua de "operando" e "multiplicando". Tradicionalmente designado mantissa na notação científica. O IEEE 754 recorre a este novo termo para evitar confusões, pois mantissa já era usado para designar a parte fraccionária de um logaritmo.

<sup>127</sup> O standard prevê várias formas de fazer arredondamento e bits de precisão adicional para cálculos intermédios, para reduzir os erros introduzidos pelos arredondamentos.

Com o número normalizado, o bit mais à esquerda (algarismo das unidades) do significando é sempre 1 (excepto se o valor for zero), pelo que é omitido e só se representa a sua parte fraccionária (no campo da fração). O sinal do significando está no bit mais significativo do número para ser mais fácil ver se este é positivo ou negativo. O expoente está logo a seguir, nos bits mais significativos do número, para facilitar a comparação entre dois números. Aquele com maior expoente é o maior e a fração só é usada para desempatar entre dois números com igual expoente. Naturalmente, só se podem comparar números de igual precisão, pelo que pode haver necessidade de converter um formato noutra. Para este efeito, note-se que os bits da fração estão alinhados à esquerda e podem acrescentar-se zeros nos bits mais à direita sem alterar o valor do número, pois são à direita da vírgula.

No expoente não se usa grandeza e sinal, mas antes deslocamento. Com os 8 bits do expoente do formato simples, por exemplo, dever-se-ia poder representar grandezas desde  $2^{-127}$  a  $2^{+127}$ , mas em vez de se usar 7 bits para o valor do expoente e um para o seu sinal soma-se um deslocamento de 127 ao valor pretendido para o expoente do número. Assim, o valor que fica no campo do expoente varia entre 1 e 254, para indicar expoentes reais entre -126 e +127 (os extremos 0 e 255 deste campo têm significados especiais). Este esquema é usado para permitir comparar números em vírgula flutuante directamente, como se fossem inteiros de 32 ou 64 bits. Os números ficam assim ordenados desde o mais infinito (expoente -126 para a precisão simples) até ao maior (expoente +127).

Como exemplo, o número normalizado  $-1.100011 \times 2^3$  (convertido anteriormente a partir do valor decimal -12,375), é representado em precisão simples na Fig. D.2a. O bit de sinal é 1 (número negativo), o campo do expoente é 130 ( $127 + 3$ ), ou 82H, e o campo da fração inclui a parte fraccionária do número com um enximento do número necessário de zeros à direita. Os números normalizados têm sempre o bit à esquerda da vírgula a 1, pelo que não é preciso representá-lo (fica implícito). A Fig. D.2b representa o mesmo número em precisão dupla e ilustra também a conversão entre formatos. O campo do expoente é agora 1026 ( $1023 + 3$ ), pois o campo tem 11 bits em vez de 8 bits e o deslocamento é maior, enquanto o campo da fração se mantém com o mesmo valor à esquerda mas agora com um enximento de 46 bits a 0.

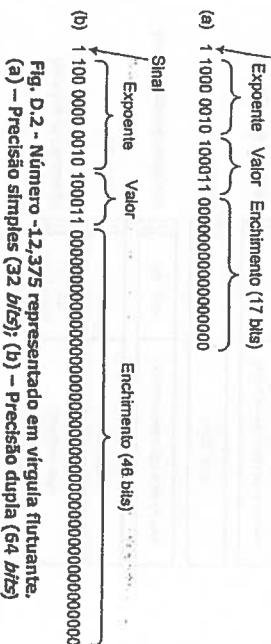


Fig. D.2 - Número -12,375 representado em vírgula flutuante:  
(a) - Precisão simples (32 bits); (b) - Precisão dupla (64 bits)

A Tabela D.1 compara os dois formatos em diversas características.

| PARÂMETRO                          | PRECISÃO SIMPLES                | PRECISÃO DUPLA                    |
|------------------------------------|---------------------------------|-----------------------------------|
| Nº de bits da palavra              | 32                              | 64                                |
| Nº de bits da fração               | 23                              | 52                                |
| N.º de bits do expoente            | 8                               | 11                                |
| Deslocamento no expoente           | 127                             | 1023                              |
| Exponentes utilizáveis             | -126 .. +127                    | -1022 .. +1023                    |
| Valores límite de grandeza (base2) | $2^{-128} .. 2^{+128}$ (aprox.) | $2^{-1022} .. 2^{+1023}$ (aprox.) |
| Menor número desnormalizado        | $10^{-38} .. 10^{+38}$          | $10^{-308} .. 10^{+308}$          |
| Menor número normalizado           | $10^{-38}$ (aprox.)             | $10^{-324}$ (aprox.)              |

Tabela D.1 - Características da representação dos números em vírgula flutuante na Norma IEEE 754

Os valores extremos do campo do expoente (bits todos a 0 e todos a 1) têm significados especiais, contemplados na Fig. D.3:

- O valor zero tem uma representação específica e é um valor exacto. Em muitos cálculos, é importante um valor ser mesmo zero e não um número arbitrariamente pequeno. Note-se que há dois zeros, um positivo e outro negativo, característica que não tem problemas numa representação em sinal e grandeza;
- Os números desnormalizados permitem representar valores mais pequenos que o menor valor normalizado, embora com menos precisão. Tem o campo do expoente com o valor zero (menor valor possível) e, ao contrário dos números normalizados, assume-se que o bit mais à esquerda é agora 0. O menor destes valores tem todos os bits do campo da fração a 0 excepto o mais à direita. O truque destes valores é terem zeros à esquerda na fração, que são zeros entre a vírgula e o primeiro bit significativo (diferente de 0). A escala dos números mais pequenos é assim refinada, mas sem conseguir manter a precisão;

- No inverso da escala, quando o maior número representável é atingido, reservou-se uma representação para o conceito de infinito (positivo e negativo). Este valor pode ser usado nas operações. Por exemplo, qualquer número dividido por infinito dá zero;

- Os NaNs (Not A Number), ou não-números, são entidades representáveis pelo IEEE 754 mas não correspondem a valores válidos, permitindo contemplar variáveis não inicializadas ou resultados de operações inválidas (divisão por zero) ou indefinidas (infinity dividido por infinity). Podem ser propagados através de expressões numéricas, permitindo assimilar um erro que pode ser testado mais tarde em vez de gerar logo uma exceção, cujo mecanismo varia de computador para computador.

|                       | Sinal | Exponente                | Fracção                          |
|-----------------------|-------|--------------------------|----------------------------------|
| Número normalizado    | 0     | 000...001 a<br>111...110 | Qualquer valor                   |
| Valor zero            | 0     | 000...000                | 0000...0000                      |
| Número desnormalizado | 0     | 000...000                | Qualquer valor diferente de zero |
| Infinito              | 1     | 111...111                | 0000...0000                      |
| Nan (Not A Number)    | 0     | 111...111                | Qualquer valor diferente de zero |

Fig. D.3 - Tipos de valores do IEEE 754

### D.3 OPERAÇÕES ARITMÉTICAS EM VÍRGULA FLUTUANTE

Estas operações são mais complexas do que as dos números inteiros. As operações incidem sobre os significandos (incluindo o bit 1 implícito e a fração) e os exponentes, que são processados separadamente. Os operandos devem estar normalizados e o valor do resultado deve ser normalizado antes de a operação terminar. Como o valor zero tem uma representação específica, os algoritmos das operações não funcionam com este valor. Por esta razão, antes de efectuar a operação tem de se testar se algum dos operandos é zero (ou os dois) e se for o caso produzir o valor de resultado adequado sem efectuar a operação propriamente dita (por exemplo, soma com zero é igual o número original e multiplicação por zero dá zero).

A multiplicação envolve multiplicar os significandos e somar os exponentes. Na divisão dividem-se os significandos e subtraem-se os exponentes. Podem originar excesso (*overflow*) e défice (*underflow*), respectivamente. A Fig. D.4 ilustra a operação de multiplicação. Note-se que o deslocamento tem de ser subtraído a soma dos exponentes, pois cada um dos exponentes dos operandos já tem um deslocamento somado, e este não deve ser incluído duas vezes no resultado. Por simplicidade, não estão representados alguns aspectos, como determinação da nulidade dos operandos e verificação de excesso no resultado.

Curiosamente, a adição é mais complexa do que a multiplicação, pois antes de fazer a soma tem de se colocar os dois números com o mesmo expoente, o que implica desnormalizar o número com menor expoente, deslocando o seu significando (a fração com o 1 implícito à sua esquerda) para a direita e aumentando o seu expoente do número de bits necessário até os dois exponentes terem o mesmo valor. Isto implica uma redução de precisão do menor número. Se a diferença de exponentes for maior do que os bits da fração, o menor dos números pode mesmo ficar o seu campo de fração a zero e a soma é igual ao outro operando (um dos operandos é desprezável face ao outro). A subtração é tratada como uma soma depois de trocar o sinal ao subtraendo. A Fig. D.5 ilustra a operação de adição. Tal como na figura anterior, não estão representados os testes de nulidade dos operandos e nem a verificação de excesso no resultado.

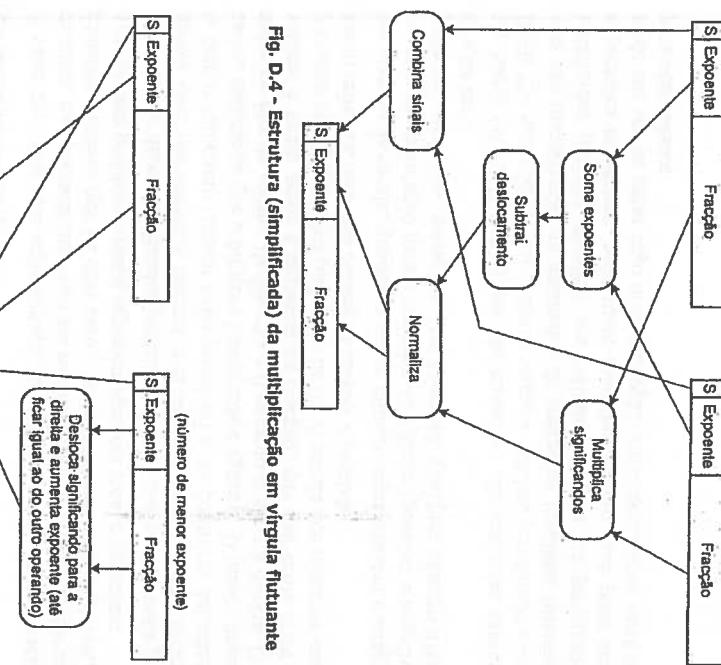


Fig. D.4 - Estrutura (simplificada) da multiplicação em vírgula flutuante

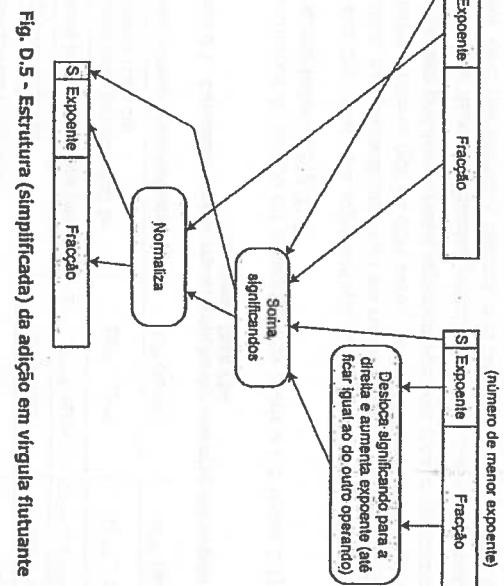


Fig. D.5 - Estrutura (simplificada) da adição em vírgula flutuante

## APÊNDICE E

# CODIFICAÇÃO DE CARACTERES EM ASCII

A Tabela E.1 especifica a codificação ASCII (*American Standard Code for Information Interchange*) em hexadecimal de cada um dos caracteres usados em textos (letras, dígitos, sinais de pontuação). É uma representação de 7 bits, assumindo-se que num byte o bit de maior peso está a 0. A primeira extensão do ASCII básico foi o Latin-1, que ocupou as 128 codificações com o bit de maior peso a 1 essencialmente com letras com acentos. As codificações de 00H a 1FH correspondem a caracteres de controlo de escrita e de comunicação, muitas agora com interesse apenas histórico, e não têm representação para imprimir. Destes, os mais importantes são NUL (null, 00H), CR (*carriage return*, 0DH) e LF (*line feed*, 0AH), em que os dois últimos são usados para mudança de linha.

|     |        |     |   |     |   |     |   |     |   |     |     |
|-----|--------|-----|---|-----|---|-----|---|-----|---|-----|-----|
| 20H | espaco | 30H | 0 | 40H | @ | 50H | P | 60H | : | 70H | p   |
| 21H | -      | 31H | 1 | 41H | A | 51H | Q | 61H | a | 71H | q   |
| 22H | "      | 32H | 2 | 42H | B | 52H | R | 62H | b | 72H | r   |
| 23H | #      | 33H | 3 | 43H | C | 53H | S | 63H | c | 73H | s   |
| 24H | \$     | 34H | 4 | 44H | D | 54H | T | 64H | d | 74H | t   |
| 25H | %      | 35H | 5 | 45H | E | 55H | U | 65H | e | 75H | u   |
| 26H | &      | 36H | 6 | 46H | F | 56H | V | 66H | f | 76H | v   |
| 27H | *      | 37H | 7 | 47H | G | 57H | W | 67H | g | 77H | w   |
| 28H | (      | 38H | 8 | 48H | H | 58H | X | 68H | h | 78H | x   |
| 29H | )      | 39H | 9 | 49H | I | 59H | Y | 69H | i | 79H | y   |
| 2AH | *      | 3AH | : | 4AH | J | 5AH | Z | 6AH | j | 7AH | z   |
| 2BH | +      | 3BH | : | 4BH | K | 5BH | L | 6BH | k | 7BH | l   |
| 2CH | ,      | 3CH | < | 4CH | L | 5CH | \ | 6CH | l | 7CH | \   |
| 2DH | -      | 3DH | > | 4DH | M | 5DH | / | 6DH | m | 7DH | /   |
| 2EH | .      | 3EH | > | 4EH | N | 5EH | ^ | 6EH | n | 7EH | ~   |
| 2FH | /      | 3FH | ? | 4FH | O | 5FH | _ | 6FH | o | 7FH | DEL |

Tabela E.1 - Codificação ASCII

## BIBLIOGRAFIA

- [Anandtech] [www.anandtech.com](http://www.anandtech.com)  
[Baptista 2002] Carlos Baptista, *Fundamental dos Sistemas Digitais*, FCA, 2002, ISBN 972-722-272-2
- [Botros 2005] Nazeh Botros, *HDL Programming Fundamentals : VHDL and Verilog*, Charles River Media, 2005, ISBN 158-450-855-8
- [Buchanan 2001] William Buchanan e Austin Wilson, *Advanced PC Architecture*, Addison-Wesley, 2001, ISBN 0-201-59858-3
- [Crespo 2001] Rui Crespo, *Processadores de Linguagens: da Concepção à Implementação*, 2.ª edição, IST Press, 2001, ISBN 972-8469-18-7
- [Farhat 2003] Hassan Farhat, *Digital Design and Computer Organization*, CRC Press, 2003, ISBN 0849311918
- [Gonçalves 2005] Victor Gonçalves, *Sistemas Electrónicos com Microcontroladores*, 2.ª edição, FCA, 2005, ISBN 972-8480-12-1
- [Guerreiro 2001] Pedro Guerreiro, *Elementos de Programação com C*, 2.ª edição, FCA, 2001, ISBN 972-722-300-1
- [Guerreiro 2003] Pedro Guerreiro, *Programação com Classes em C++*, 2.ª edição, FCA, 2003, ISBN 972-722-375-3
- [Hamacher 2002] Carl Hamacher, Zvonko Vranescic e Safwat Zaky, *Computer Organization*, 5.ª edição, McGraw-Hill, 2002, ISBN 0-07-112218-4
- [IEEE] [www.computer.org/history](http://www.computer.org/history)
- [IEEE 1984] "IEEE Standard Graphic Symbols for Logic Functions", ANSI/IEEE Std 91-1984
- [IEEE 1985] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, Agosto 1985
- [Mano 2003] Morris Mano, *Logic and Computer Design Fundamentals*, Prentice-Hall, 2003, ISBN 013140539X
- [Marques 2005] Paulo Marques e Hernâni Pedroso, *C# 2.0*, FCA, 2005, ISBN 972-722-508-x
- [Mendes 2003] António Mendes e Maria Macelino, *Fundamentos de Programação em Java 2, 2.ª edição*, FCA, 2003, ISBN 972-722-423-7
- [Mims 2003] Forrest Mims, *Getting Started in Electronics*, Master Publishing, 2003, ISBN 094-505-328-2
- [Minasi 2005] Mark Minasi, Faith Wempner e Quentin Docter, *The Complete PC Upgrade and Maintenance Guide*, Sybex, 2005, ISBN 078-214-431-4
- [Nunes 2004] Mauro Nunes e Henrique O'Neill, *Fundamental de UML*, 3.ª edição, FCA, 2004, ISBN 972-722-481-4
- [Panami 2005] Behrooz Parhami, *Computer Architecture: From Microprocessors to Supercomputers*, Oxford University Press, 2005, ISBN 0-19-515455-X
- [Patterson 2008] D. Patterson e J. Hennessy, *Computer organization & design: the hardware/software interface*, Morgan Kaufmann Publishers, 4.ª edição, 2008, ISBN 0123744938

[PCGuide] [www.pcguide.com](http://www.pcguide.com)[PCStats] [pcstats.com](http://pcstats.com)[PCTechGuide] [www.pctechguide.com](http://www.pctechguide.com)[Rational] [www.rational.com/unl](http://www.rational.com/unl)[Reagan 2005] Patrick Reagan, *Troubleshooting the PC with A+ Preparation*, 3.ª edição, Prentice-Hall, 2005, ISBN 013-119467-4[Silva 2005] Alberto Silva e Carlos Videira, *UML - Metodologias e Ferramentas CASE*, 2.ª edição, Centro Atlântico, 2005, ISBN 989-615-009-5[Shiva 2000] Sajjan Shiva, *Computer Design and Architecture*, 3.ª edição, Marcel Dekker, Inc., 2000, ISBN 0-8247-0368-5

[Stalling 2006] William Stallings, "Computer Organization &amp; Architecture", 7.ª edição, Pearson Prentice Hall, 2006, ISBN 0-13-183644-8

[Tanenbaum 1999] A. Tanenbaum, *Structured computer organization*, Prentice-Hall, 4.ª edição, 1999, ISBN 0-13-095990-1[Tanenbaum 2001] Andrew Tanenbaum, *Modern Operating Systems*, 2.ª edição, Prentice-Hall, ISBN 0130313580[Thompson 2004] Robert Thompson e Barbara Thompson, *Building The Perfect PC*, O'Reilly, 2004, ISBN 059-600663-2[USB] [www.usb.org/](http://www.usb.org/)[Wurster 2002] C. Wurster, *Computers: an Illustrated History*, Taschen GmbH, 2002, ISBN 3-8228-1293-5

## ÍNDICE REMISSIVO

### **A**

Accelerated Graphics Port. Ver AGP

Acesso Directo à Memória. Ver DMA

Acessos à memória. Ver Memória

alminhados . 198, 443

ciclos de

escrita . 451, 696

leitura . 449, 696

desalinhados . 198, 444

dispositivos lentos . 456

em 16 bits . 224

com índice constante . 227

instruções . 229

sem índice . 226

em 8 bits . 230

em 8 e 16 bits . 235

ADD . 153, 238, 703

ADDC . 238, 703

Address. Ver Endereço

AGP . 533, 534

Aleatório . 66, 115, 175, 219, 369, 490, 636, 728

Algariano . 87, 258

Algebra de Boole . 38, 109

axiomas . 40

operações . 38

Algortimo . 5, 89, 104, 164, 245, 548

Alta impedância . 66, 170, 450, 513, 567, 730,

Ver Portas tristate

ALU . 123, 145, 184, 566, 567, 575, 603, 648,

688

Ambiente de desenvolvimento . 163, 356, 397

cruzado . 397

integrado . Ver IDE

AMD . 23, 521, 549, 555

AND . 35, 124, 153, 188, 246, 247, 376, 576, 703

Anexação dos dados . 616

Aperito-de-mão . 491, 502

Apontadores . 294, 297, 314, 345, 375, 383

aritmética de . 215, 296

Apple . 21, 26, 492, 518, 524

Arbitrio . 191, 488, 638, 699

### **B**

Backbone . 532

Backups. Ver Cópias de segurança

Banco de registo . 66, 184, 187, 567, 572, 573,

643

Barraamentos . 410, 411, 483, 506, 538, 567

assistentes . 490

de controlo . 411

de dados . 411

de endereços . 410

escrita em . 411

hierárquicos . 506

leitura de . 411

síncronos . 490

Barrel-shifter . 578

Báscula . 59, 60, 615, 715

activada no fluxo . 59

Base . 363

binária . 87

|                                                |                                                              |
|------------------------------------------------|--------------------------------------------------------------|
| à Tabela de Excepções . Ver BTE                | integração com a memória virtual . 657                       |
| de numeração . 87, 221                         | não bloqueantes . 660                                        |
| decimal . 9                                    | no PEPE . 642                                                |
| hexadecimal . 89                               | organização . 625                                            |
| Baud . 495                                     | princípios de funcionamento . 622                            |
| BCD . 265                                      | Cadeia                                                       |
| Benchmark . Ver Programa de avaliação          | de caracteres . 231, 253, 291                                |
| BIGT . 513                                     | de catágoos . 595                                            |
| Big-endian . 438, 439, 442, 622                | de instruções . 599, 603                                     |
| Bill Gates . 16, 17, 21                        | de microinstruções . 599, 608                                |
| Binary Coded Decimal . Ver BCD                 | encilhamento . 597                                           |
| Bit . 9, 34, 57, 87, 91, 92, 93                | esvaziamento . 597                                           |
| de estado . 205, 206, 209, 217, 234, 239, 262, | estados . 713                                                |
| 673                                            | CALL . 312, 327, 707                                         |
| de paridade . 493                              | Callbacks . 684                                              |
| de sinal . 94, 98, 221, 738                    | Cartridge . 317, 707                                         |
| de sujidade . 637                              | Cão-de-guarda . 536                                          |
| de utilização . 651                            | Capacidade . 55, 91, 117, 272                                |
| de validade . 626, 627, 637, 649, 699          | CAR (Control Address Register) . Ver MTC                     |
| BTI . 246, 254, 703                            | Cartridge return . 743                                       |
| Bolha . 277, 597, 627                          | Centro de Dados . Ver Centro de Processamento de Dados       |
| Boole . 29, 38, 39, 40, 41, 109                | Centro de Informática . Ver Centro de Processamento de Dados |
| Boot loader . 398                              | Centro de Processamento de Dados . 517                       |
| Breakpoints . Ver Pontos de paragem            | Centros de dados . 526                                       |
| Broadcast . 504                                | Chamada                                                      |
| BRQ . 513                                      | ao sistema . 374, 684                                        |
| BTE . 204, 460, 473, 570, 696                  | de funções . 307, 312                                        |
| Bubble . Ver Bolha                             | de rotinas . 204, 313, 344, 357                              |
| Bug . 19                                       | Checksum . 495                                               |
| Broadcast . 504                                | Chip Enable . Ver Enable                                     |
| Burst . Ver Rajada                             | Chip Select . Ver Sinal de seleção                           |
| Bus . Ver Barramento                           | Chipset . 532, 533                                           |
| Grant . Ver BGT                                | Círculo . 484                                                |
| Request . Ver BRQ                              | Concentrador . 533                                           |
| Busca                                          | Conflito                                                     |
| de instruções . 330, 581, 685                  | de dados . 490                                               |
| de microinstruções . 600, 603                  | assíncrona . 494                                             |
| Byte . 10, 89, 93, 196, 230, 269, 430, 435     | full-duplex . 492                                            |
| addressing . Ver Endereçamento de byte         | half-duplex . 492                                            |
| Cache                                          | Componentes básicos . 114                                    |
| acesso com sucesso . 191, 571, 623, 657        | estreitura básica . 6                                        |
| bloco . 624, 629, 636                          | evolução dos . 16                                            |
| conjunto . 633, 636, 699                       | inteligência dos componentes . 410                           |
| falta no acesso . 571, 598, 623, 630           | supercomputadores . 17                                       |
| non-cocheable . 641                            | Comunicação                                                  |
| vía . 633, 636                                 | assíncrona . 494                                             |
| Caches . 190, 622                              | série . 492                                                  |
| casos em que não se querem . 640               | simples . 492                                                |
| de instruções . 522, 567                       | sintrona . 496                                               |
| Colisão . 490                                  | Concentrador . 533                                           |
| Código-fonte . 13, 709                         | Condito                                                      |
| Código-máquina . 13, 195, 283, 390             | de dados . 488, 614, 617                                     |
| Colisão . 490                                  | entre saídas . 65, 168, 414, 446                             |
| Comentários . 134, 176, 352, 395               | Conjunção . 36, 125. Ver AND                                 |
| Compare . 187, 226. Ver CM4P                   | Conjunto de instruções . 153, 210, 293, 480, 560             |
| Compilador . 13, 542                           | Constantes . 135, 216, 366                                   |
| de silício . 47                                | Contador                                                     |
| Complex Instruction Set Computers . Ver CISC   | de endereços . 290, 291                                      |
| Computador                                     | de impulsos de rebigo . 68, 540, 715                         |
| Contexto                                       | de programa . Ver PC (Program Counter)                       |
| de chamada de rotinas . 313, 336, 351, 356     | em software . 163, 184                                       |
| de processos . 666, 676                        | Context switch . Ver Mudança de contexto                     |
| Controlador                                    | Contexto                                                     |
| de disco . 485, 512, 557                       | de chamada de rotinas . 313, 336, 351, 356                   |
| de DMA . 512, 550                              | de processos . 666, 676                                      |
| de interrupções . 476, 500, 519, 535. Ver PIC  | Depuração . 258, 393                                         |
| Códigos de operação . 189, 200, 592            | Descomilação de endereços                                    |
| primário . 592                                 | espelhos . 422                                               |
| secundário . 592                               | mapas irregulares . 425                                      |
| Código-fonte . 13, 709                         | parcial . 423                                                |
| Cópias de segurança . 531, 644                 | programável . 427                                            |
| digital/análoga . 536                          | Descomilador de endereços . 417                              |
| digital/análogica . 536                        | Descodificadores . 51                                        |
| CPL . 246, 704                                 | Descriptor . 683                                             |
| Desenvolvimento                                | Défice . 737, 740                                            |

|                               |                                               |
|-------------------------------|-----------------------------------------------|
| CRC . 495, 503                | D_DESALINHADO . 481, 702                      |
| CREPE . 538, 711              | D_FALTA_PAG . 702                             |
| exceções . 718                | D_PROT . 702                                  |
| exemplo de utilização . 720   | DACK . 513                                    |
| periféricos . 714             | Dados                                         |
| estados dos . 718             | confílio de . 614                             |
| portos de entrada/saída . 714 | de entrada . 6, 114                           |
| temporizadores . 715          | gestão dos . 360                              |
| UARTs . 716                   | unidade de . 119, 127, 145, 148, 191, 568     |
| pinos . 711                   | Data . Ver Dados                              |
| registos auxiliares . 711     | center . Ver Centro de Processamento de Dados |
| hostpedro . 397               | Datas                                         |
| pessoal . 397                 | confílio de . 614                             |
| alvo . 397                    | de entrada . 6, 114                           |
| temporizadores . 715          | forwarding . Ver Anticipação dos dados        |
| UARTs . 716                   | hazard . Ver Confílio de dados                |
| pinos . 711                   | Daughterboard . Ver Placa-mãe                 |
| registos auxiliares . 711     | Declaração                                    |
| hostpedro . 397               | Dados                                         |
| pessoal . 397                 | forwarding . Ver Anticipação dos dados        |
| alvo . 397                    | hazard . Ver Confílio de dados                |
| temporizadores . 715          | Daughterboard . Ver Placa-mãe                 |
| UARTs . 716                   | Declaração                                    |
| pinos . 711                   | Défice . 737, 740                             |
| registos auxiliares . 711     | Delay time . Ver Tempo de Atraso              |
| hostpedro . 397               | Dependências                                  |
| pessoal . 397                 | de controlo . 618                             |
| alvo . 397                    | de dados . 613                                |
| temporizadores . 715          | Depuração . 258, 393                          |
| UARTs . 716                   | Descomilação de endereços                     |
| pinos . 711                   | espelhos . 422                                |
| registos auxiliares . 711     | mapas irregulares . 425                       |
| hostpedro . 397               | parcial . 423                                 |
| pessoal . 397                 | programável . 427                             |
| temporizadores . 715          | Descomilador de endereços . 417               |
| UARTs . 716                   | Descodificadores . 51                         |
| pinos . 711                   | Descriptor . 683                              |
| registos auxiliares . 711     | Défice . 737, 740                             |
| hostpedro . 397               | Delay time . Ver Tempo de Atraso              |
| pessoal . 397                 | Dependências                                  |
| alvo . 397                    | de controlo . 618                             |
| temporizadores . 715          | de dados . 613                                |
| UARTs . 716                   | Depuração . 258, 393                          |
| pinos . 711                   | Descomilação de endereços                     |
| registos auxiliares . 711     | espelhos . 422                                |
| hostpedro . 397               | mapas irregulares . 425                       |
| pessoal . 397                 | parcial . 423                                 |
| temporizadores . 715          | programável . 427                             |
| UARTs . 716                   | Descomilador de endereços . 417               |
| pinos . 711                   | Descodificadores . 51                         |
| registos auxiliares . 711     | Descriptor . 683                              |
| hostpedro . 397               | Défice . 737, 740                             |
| pessoal . 397                 | Delay time . Ver Tempo de Atraso              |
| temporizadores . 715          | Dependências                                  |
| UARTs . 716                   | de controlo . 618                             |
| pinos . 711                   | de dados . 613                                |
| registos auxiliares . 711     | Depuração . 258, 393                          |
| hostpedro . 397               | Descomilação de endereços                     |
| pessoal . 397                 | espelhos . 422                                |
| temporizadores . 715          | mapas irregulares . 425                       |
| UARTs . 716                   | parcial . 423                                 |
| pinos . 711                   | programável . 427                             |
| registos auxiliares . 711     | Descomilador de endereços . 417               |
| hostpedro . 397               | Descodificadores . 51                         |
| pessoal . 397                 | Descriptor . 683                              |
| temporizadores . 715          | Défice . 737, 740                             |
| UARTs . 716                   | Delay time . Ver Tempo de Atraso              |
| pinos . 711                   | Dependências                                  |
| registos auxiliares . 711     | de controlo . 618                             |
| hostpedro . 397               | de dados . 613                                |
| pessoal . 397                 | Depuração . 258, 393                          |
| temporizadores . 715          | Descomilação de endereços                     |
| UARTs . 716                   | espelhos . 422                                |
| pinos . 711                   | mapas irregulares . 425                       |
| registos auxiliares . 711     | parcial . 423                                 |
| hostpedro . 397               | programável . 427                             |
| pessoal . 397                 | Descomilador de endereços . 417               |
| temporizadores . 715          | Descodificadores . 51                         |
| UARTs . 716                   | Descriptor . 683                              |
| pinos . 711                   | Défice . 737, 740                             |
| registos auxiliares . 711     | Delay time . Ver Tempo de Atraso              |
| hostpedro . 397               | Dependências                                  |
| pessoal . 397                 | de controlo . 618                             |
| temporizadores . 715          | de dados . 613                                |
| UARTs . 716                   | Depuração . 258, 393                          |
| pinos . 711                   | Descomilação de endereços                     |
| registos auxiliares . 711     | espelhos . 422                                |
| hostpedro . 397               | mapas irregulares . 425                       |
| pessoal . 397                 | parcial . 423                                 |
| temporizadores . 715          | programável . 427                             |
| UARTs . 716                   | Descomilador de endereços . 417               |
| pinos . 711                   | Descodificadores . 51                         |
| registos auxiliares . 711     | Descriptor . 683                              |
| hostpedro . 397               | Défice . 737, 740                             |
| pessoal . 397                 | Delay time . Ver Tempo de Atraso              |
| temporizadores . 715          | Dependências                                  |
| UARTs . 716                   | de controlo . 618                             |
| pinos . 711                   | de dados . 613                                |
| registos auxiliares . 711     | Depuração . 258, 393                          |
| hostpedro . 397               | Descomilação de endereços                     |
| pessoal . 397                 | espelhos . 422                                |
| temporizadores . 715          | mapas irregulares . 425                       |
| UARTs . 716                   | parcial . 423                                 |
| pinos . 711                   | programável . 427                             |
| registos auxiliares . 711     | Descomilador de endereços . 417               |
| hostpedro . 397               | Descodificadores . 51                         |
| pessoal . 397                 | Descriptor . 683                              |
| temporizadores . 715          | Défice . 737, 740                             |
| UARTs . 716                   | Delay time . Ver Tempo de Atraso              |
| pinos . 711                   | Dependências                                  |
| registos auxiliares . 711     | de controlo . 618                             |
| hostpedro . 397               | de dados . 613                                |
| pessoal . 397                 | Depuração . 258, 393                          |
| temporizadores . 715          | Descomilação de endereços                     |
| UARTs . 716                   | espelhos . 422                                |
| pinos . 711                   | mapas irregulares . 425                       |
| registos auxiliares . 711     | parcial . 423                                 |
| hostpedro . 397               | programável . 427                             |
| pessoal . 397                 | Descomilador de endereços . 417               |
| temporizadores . 715          | Descodificadores . 51                         |
| UARTs . 716                   | Descriptor . 683                              |
| pinos . 711                   | Défice . 737, 740                             |
| registos auxiliares . 711     | Delay time . Ver Tempo de Atraso              |
| hostpedro . 397               | Dependências                                  |
| pessoal . 397                 | de controlo . 618                             |
| temporizadores . 715          | de dados . 613                                |
| UARTs . 716                   | Depuração . 258, 393                          |
| pinos . 711                   | Descomilação de endereços                     |
| registos auxiliares . 711     | espelhos . 422                                |
| hostpedro . 397               | mapas irregulares . 425                       |
| pessoal . 397                 | parcial . 423                                 |
| temporizadores . 715          | programável . 427                             |
| UARTs . 716                   | Descomilador de endereços . 417               |
| pinos . 711                   | Descodificadores . 51                         |
| registos auxiliares . 711     | Descriptor . 683                              |
| hostpedro . 397               | Défice . 737, 740                             |
| pessoal . 397                 | Delay time . Ver Tempo de Atraso              |
| temporizadores . 715          | Dependências                                  |
| UARTs . 716                   | de controlo . 618                             |
| pinos . 711                   | de dados . 613                                |
| registos auxiliares . 711     | Depuração . 258, 393                          |
| hostpedro . 397               | Descomilação de endereços                     |
| pessoal . 397                 | espelhos . 422                                |
| temporizadores . 715          | mapas irregulares . 425                       |
| UARTs . 716                   | parcial . 423                                 |
| pinos . 711                   | programável . 427                             |
| registos auxiliares . 711     | Descomilador de endereços . 417               |
| hostpedro . 397               | Descodificadores . 51                         |
| pessoal . 397                 | Descriptor . 683                              |
| temporizadores . 715          | Défice . 737, 740                             |
| UARTs . 716                   | Delay time . Ver Tempo de Atraso              |
| pinos . 711                   | Dependências                                  |
| registos auxiliares . 711     | de controlo . 618                             |
| hostpedro . 397               | de dados . 613                                |
| pessoal . 397                 | Depuração . 258, 393                          |
| temporizadores . 715          | Descomilação de endereços                     |
| UARTs . 716                   | espelhos . 422                                |
| pinos . 711                   | mapas irregulares . 425                       |
| registos auxiliares . 711     | parcial . 423                                 |
| hostpedro . 397               | programável . 427                             |
| pessoal . 397                 | Descomilador de endereços . 417               |
| temporizadores . 715          | Descodificadores . 51                         |
| UARTs . 716                   | Descriptor . 683                              |
| pinos . 711                   | Défice . 737, 740                             |
| registos auxiliares . 711     | Delay time . Ver Tempo de Atraso              |
| hostpedro . 397               | Dependências                                  |
| pessoal . 397                 | de controlo . 618                             |
| temporizadores . 715          | de dados . 613                                |
| UARTs . 716                   | Depuração . 258, 393                          |
| pinos . 711                   | Descomilação de endereços                     |
| registos auxiliares . 711     | espelhos . 422                                |
| hostpedro . 397               | mapas irregulares . 425                       |
| pessoal . 397                 | parcial . 423                                 |
| temporizadores . 715          | programável . 427                             |
| UARTs . 716                   | Descomilador de endereços . 417               |
| pinos . 711                   | Descodificadores . 51                         |
| registos auxiliares . 711     | Descriptor . 683                              |
| hostpedro . 397               | Défice . 737, 740                             |
| pessoal . 397                 | Delay time . Ver Tempo de Atraso              |
| temporizadores . 715          | Dependências                                  |
| UARTs . 716                   | de controlo . 618                             |
| pinos . 711                   | de dados . 613                                |
| registos auxiliares . 711     | Depuração . 258, 393                          |
| hostpedro . 397               | Descomilação de endereços                     |
| pessoal . 397                 | espelhos . 422                                |
| temporizadores . 715          | mapas irregulares . 425                       |
| UARTs . 716                   | parcial . 423                                 |
| pinos . 711                   | programável . 427                             |
| registos auxiliares . 711     | Descomilador de endereços . 417               |
| hostpedro . 397               | Descodificadores . 51                         |
| pessoal . 397                 | Descriptor . 683                              |
| temporizadores . 715          | Défice . 737, 740                             |
| UARTs . 716                   | Delay time . Ver Tempo de Atraso              |
| pinos . 711                   | Dependências                                  |
| registos auxiliares . 711     | de controlo . 618                             |
| hostpedro . 397               | de dados . 613                                |
| pessoal . 397                 | Depuração . 258, 393                          |
| temporizadores . 715          | Descomilação de endereços                     |
| UARTs . 716                   | espelhos . 422                                |
| pinos . 711                   | mapas irregulares . 425                       |
| registos auxiliares . 711     | parcial . 423                                 |
| hostpedro . 397               | programável . 427                             |
| pessoal . 397                 | Descomilador de endereços . 417               |
| temporizadores . 715          | Descodificadores . 51                         |
| UARTs . 716                   | Descriptor . 683                              |
| pinos . 711                   | Défice . 737, 740                             |
| registos auxiliares . 711     | Delay time . Ver Tempo de Atraso              |
| hostpedro . 397               | Dependências                                  |
| pessoal . 397                 | de controlo . 618                             |
| temporizadores . 715          | de dados . 613                                |
| UARTs . 716                   | Depuração . 258, 393                          |
| pinos . 711                   | Descomilação de endereços                     |
| registos auxiliares . 711     | espelhos . 422                                |
| hostpedro . 397               | mapas irregulares . 425                       |
| pessoal . 397                 | parcial . 423                                 |
| temporizadores . 715          | programável . 427                             |
| UARTs . 716                   | Descomilador de endereços . 417               |
| pinos . 711                   | Descodificadores . 51                         |
| registos auxiliares . 711     | Descriptor . 683                              |
| hostpedro . 397               | Défice . 737, 740                             |
| pessoal . 397                 | Delay time . Ver Tempo de Atraso              |
| temporizadores . 715          | Dependências                                  |
| UARTs . 716                   | de controlo . 618                             |
| pinos . 711                   | de dados . 613                                |
| registos auxiliares . 711     | Depuração . 258, 393                          |
| hostpedro . 397               | Descomilação de endereços                     |
| pessoal . 397                 | espelhos . 422                                |
| temporizadores . 715          | mapas irregulares . 425                       |
| UARTs . 716                   | parcial . 423                                 |
| pinos . 711                   | programável . 427                             |
| registos auxiliares . 711     | Descomilador de endereços . 417               |
| hostpedro . 397               | Descodificadores . 51                         |
| pessoal . 397                 | Descriptor . 683                              |
| temporizadores . 715          | Défice . 737, 740                             |
| UARTs . 716                   | Delay time . Ver Tempo de Atraso              |
| pinos . 711                   | Dependências                                  |
| registos auxiliares . 711     | de controlo . 618                             |
| hostpedro . 397               | de dados . 613                                |
| pessoal . 397                 | Depuração . 258, 393                          |
| temporizadores . 715          | Descomilação de endereços                     |
| UARTs . 716                   | espelhos . 422                                |
| pinos . 711                   | mapas irregulares . 425                       |
| registos auxiliares . 711     | parcial . 423                                 |
| hostpedro . 397               | programável . 427                             |
| pessoal . 397                 | Descomilador de endereços . 417               |
| temporizadores . 715          | Descodificadores . 51                         |
| UARTs . 716                   | Descriptor . 683                              |
| pinos . 711                   | Défice . 737, 740                             |
| registos auxiliares . 711     | Delay time . Ver Tempo de Atraso              |
| hostpedro . 397               | Dependências                                  |
| pessoal . 397                 | de controlo . 618                             |
| temporizadores . 715          | de dados . 613                                |
| UARTs . 716                   | Depuração . 258, 393                          |
| pinos . 711                   | Descomilação de endereços                     |
| registos                      |                                               |

## ARQUITECTURA DE COMPUTADORES

## ÍNDICE REMISSIVO

- Ambientes de** . 396  
**de programas** . 389  
**Desktop** . Ver *Computador de secretaria*  
**Deslocamento**  
 circular . 73, 261  
 com transporte . 267  
 sem transporte . 267  
**dentro da página** . 648  
**linear** . 73, 261  
**aritmético** . 263  
**lógico** . 262  
**Despachante** . 666  
**Despacho** . 666  
**Device drivers** . Ver *Gestores de Periféricos*  
**DI** . 465, 704  
**Diagrama**  
 de blocos . 566  
 de estados . 76  
**lógico** . 46  
 temporal . 33, 61, 69, 616  
**Dígitos** . 87  
**EQU** . 285  
**PLACE** . 291  
**STRING** . 288  
**TABLE** . 288  
**WORD** . 288  
**Directório** . 652, 653  
**Dirty bit** . Ver *Bit de sujidade*  
**Disable Interrupts** . Ver *DI*  
**Disco magnético** . 485  
**Disjuncão** . 36, 125, Ver *OR*  
**Dispatch** . Ver *Despacho*  
**Dispatcher** . Ver *Despachante*  
**Display** . Ver *Mostrador*  
**DIV** . 242, 703  
**DIV0** . 481, 702  
**DMA** . 511, 569, 556, 567, 622, 641, 696  
**acknowledge** . Ver *DACK*  
**canais de** . 514  
**Double word** . 93
- 
- E**
- E** . 34  
**Edge-triggered** . Ver *Báscula activada no flanco*  
**EEPROM** . 56, 535  
**EI** . 465, 703  
**EIDE** . 531  
**EISA** . 532  
**Enable** . 53, 169  
**interrupts** . Ver *EI*  
**Endereçamento**
- 
- F**
- big-endian** . 438  
**de byte** . 118, 197  
**de palavra** . 118, 197  
**alinhado** . 290, 376  
**little-endian** . 438  
**modos de** . 288  
**Enderço** . 118  
 de base . 224  
 de retorno . 316  
**físico** . 645  
**virtual** . 645  
**Endpoint** . Ver *USB: terminais*  
**ENIAC** . 19, 38  
**EQU** . 285  
**Escalonador** . 667  
**Escalonamento** . 667  
**Espaço de endereçamento** . 192  
**físico** . 645  
**lineares** . 225  
**virtual** . 645  
**Espera aciva** . 507, 675  
**Estação de trabalho** . 517  
**Estado** . 57, 74  
**de alta impedância** . 66, 170, 450, 513, 567,  
 730  
**de espera** . 457  
**Estágio**  
 de instrução . 605  
 BM (Busca de Microinstrução) . 600  
 BO (Busca de Operandos) . 608  
 D (Descodificação) . 605  
 EM (Execução da Microinstrução) . 603, 608  
 ER (Escrita do Resultado) . 603, 608  
**Estágios** . 596  
**Ethernet** . 20, 397, 490  
**Etiqueta** . 162, 626  
**Evolução**  
 das memórias . 527  
 dos barramentos . 531  
 dos periféricos . 530  
 dos processadores . 520  
**Excepções** . 193, 204, 429, 701  
 com processamento em estágios . 610  
 de acesso à memória . 611  
 de busca . 611  
 de execução . 611  
 encadeadas . 606  
 interrupções . 612  
 invocação explícita das . 478  
 mascaráveis . 462  
 predefinidas . 480  
 prioridade das . 465  
 Excesso . 101, 241, 737, 740  
**EXCESSO** . 481, 702
- 
- G**
- GBytes** . 484  
**Gerador de constantes** . 567, 574  
**Gestores de periféricos** . 391, 555, 686  
**Gigabytes** . 91  
**Grandeza** . 737
- 
- H**
- Exclusão mutua** . 673  
**Execução**  
 paralela . 665  
 passo a passo . 163, 165, 393  
**little-endian** . 438  
**módulos de** . 288  
**Enderço** . 118  
 de base . 224  
 de retorno . 316  
**físico** . 645  
**virtual** . 645  
**Endpoint** . Ver *USB: terminais*  
**ENIAC** . 19, 38  
**EQU** . 285  
**Escalonador** . 667  
**Escalonamento** . 667  
**Espaço de endereçamento** . 192  
**factorial** . 357  
**FALSO** . 34  
**Falta de página** . 650  
**Falta de tempo** . 666  
**FIFO** . 636  
**FireWire** . 493  
**First In, First Out** . Ver *FIFO*  
**Flag** . Ver *Bit de estado*  
**Flap** . Ver *ROM Flash*  
**Flash** . Ver *Báscula*  
**Flush** . 699, 700  
**Fluxogramas** . 278, 721  
**FP** . 379  
**Frame** . Ver *USB: quadro*  
**Frame Pointer** . Ver *FP*  
**Frequência** . 30, 34  
**Fugas de memória** . 385  
**Funções**  
 chamada de . 307  
 de gestão de semáforos . 677  
 lógicas . 35, 39  
 nas linguagens de alto nível . 306  
 simplificação de . 41
- 
- I**
- I\_DESALINHADO** . 481, 702  
**I\_FATTA\_PAG** . 702  
**I\_PROT** . 702  
**I<sup>2</sup>C** . 536  
**ICE** . See  
**IDE** . 391, 393, 531  
**IE** . 465  
**Índice** . 224, 287, 364, 627  
**constante** . 225  
**variável** . 225  
**Industry Standard Architecture** . Ver *ISA*  
**Instruções** . 298  
 aritméticas . 237  
 divisão . 244  
 multiplicação . 243  
 soma de números grandes . 240  
 soma, subtração, comparação . 238  
 codificação das . 200  
 conjunto de . 210  
 de decisão  
 multiplicação . 302  
 simples . 301  
 de deslocamento . 261  
 circular (rotações) . 266  
 linear . 262  
 de manipulação de um só bit . 250  
 de salto . 134, 145, 213  
 condicional . 134, 140, 187, 189, 214, 374,  
 571  
 incondicional . 134, 140, 214, 571

- relativo . 202, 214  
de transferência de dados . 216  
lagura das . 194  
lógicas . 246  
para iteração . 304  
re-excutáveis . 650
- INT0 . 481, 702  
INT1 . 481, 702  
INT2 . 481, 702  
INT3 . 481, 702
- INTA . 476
- Integrated Development Environment* . Ver IDE
- Intel . 20, 521, 555, 639
- Interface  
com o utilizador . 8, 11, 15  
de áudio . 532  
de memória . 192, 567, 621
- gráfica . 531
- Internet . 22, 490, 557
- Interrupções . 251, 459, 462, 701
- comportamento das . 466
- controlador de . 476
- controlo do aenditamento . 464
- mechanismo de atendimento . 468
- programação com . 469
- vectorizadas . 476
- Interrupt Acknowledge* . Ver INTA
- Interruptor . 32, 37, 65, 76, 166, 168, 414, 463,  
538, 720
- Inversor . 36
- ISA . 531
- Iatinum . 23, 521, 523, 555

- J**
- JA . 214, 706  
JB . 214, 706  
JE . 214, 706  
JC . 214, 706  
JEQ . 706  
JGE . 214, 706  
JGT . 214, 706  
JLE . 214, 706  
JLT . 214, 706  
JMP . 214, 706  
IN . 214, 706  
INC . 214, 706  
INE . 214, 706  
INV . 214, 706  
INV . 214, 706  
INZ . 214, 706
- K**
- Karnaugh . 41, 70, 575  
KBytes . 91, 250, 432
- L**
- Label . Ver Etiqueta
- LAN . 397, 517, 530
- Laptop . Ver Computador de coto, Ver  
Computador de coto
- Largura . 124, 145
- Last In, First Out . Ver LIFO
- Latência . Ver Trinco
- Least Recently Used . Ver LRU
- LIFO . 37, 166, 399, 507, 555, 338, 668, 680, 721,  
728
- Lei  
de Amdahl . 545, 546  
de Moore . 20, 24, 25
- LIFO . 322, 324
- Ligação ponto-a-ponto . 411
- LineFeed . 743
- Língua natural . 12
- Linguagem  
assembly . 13, 152  
C . 13, 20, 248, 281, 296, 332, 394, 676  
C# . 26, 155, 280, 306, 394  
de programação . 13  
de Transferência de Registros . Ver RTL  
Java . 13, 23, 155, 248, 280, 378, 545  
microassembly . 585
- Linux . 26
- Listas ligadas . 382  
listas circulares . 382  
listas duplamente ligadas . 382  
Little-endian . 438, 440, 446  
Load . 187
- Localidade  
espacial . 623  
princípio da . 645  
temporal . 623
- Lock . Ver Trinco lógico
- Logograma . 46, 48, 51
- LRU . 636, 651
- Microcontroller . Ver Microcontrolador
- Microinstrução . 84

- M**
- MP . 214, 706  
JV . 214, 706  
IZ . 214, 706
- Mantissa . 736
- Mapa  
de endereços . 173, 193, 416, 417  
de Karnaugh . Ver Karnaugh
- Mapeamento  
associativo . 630  
por conjuntos . 633  
directo . 627
- Máquinas de estados . 75, 679
- assíncronas . 75  
de Mealy . 75
- de Moore . 75
- microprogramadas . 84
- síncronas . 75
- sintetizadas . 83
- Máscaras . 248
- AND . 255
- funcionamento . 254
- OR . 257
- XOR . 259
- Matriz . 552
- Mega . 91, 118
- Megabytes . 91
- Memória . 542, Ver Acessos à memória  
avaliação de desempenho . 548  
células de . 131, 216
- de dados . 6, 115, 146
- de instruções . 6, 115, 147
- de sistema . Ver Memória principal  
dinâmica . 528
- estática . 528
- hierarquia de . 643
- não volátil . 13, 115, 398, 410, 530, 644, 645
- organização em bytes . 435
- partilhada . 678
- principal . 190, 191, 528
- reciclagem automática . 378
- virtual . 643, 644, 673
- integração com as caches . 657
- no PEPE . 660
- princípios de funcionamento . 644
- voltáil . 8, 117, 398, 535
- Memory-mapped input/output* . Ver Periféricos  
mapeados em memória
- Mensagens . 678
- MFLOPS . 544
- Microassembler . 585
- Microcódigo . 581
- Microcontrolador . 518, 525, 538
- CREEP . 711
- Microcontroler . Ver Microcontrolador
- Microinstrução . 84
- N**
- NAND . 35
- NEG . 238, 703
- NEGACÃO . 34
- Nibble . 89, 93, 575
- Nível de privilegio . 573, 683
- sistema . 683
- utilizador . 683
- Non Return to Zero Invert . Ver NRZI
- NOP . 153, 607
- NOR . 35
- Norma IEEE 754 . 737
- NOT . 35, 246, 703
- Notação científica . 736
- NRZI . 498
- Null . 743

- Números  
binários . 87  
decimais . 87  
em complemento para 2 . 95  
hexadecimais . 89  
negativos . 94
- O**
- Open-collector . 489
- Operações  
aritméticas . 125  
divisão . 105  
em vírgula flutuante . 740  
inteiros . 99
- multiplicação . 103
- soma . 99
- subtração . 100
- átomicas . 673
- lógicas . 125
- com máscaras . 254
- Operandos . 152, 216
- Operton . 23, 522
- Ordenação de bolha . 276
- OU . 34
- OU-Exclusivo . 36
- Output Enable . 66
- Overflow . Ver Excesso
- P**
- Pacote . 496
- Page fault . Ver Falta de página
- Page file . Ver Swap file
- Paginação . 648
- Páginas . 648
- físicas . 648, 649, 661
- gestão das . 651
- HTML . 57
- tabela de . 652, 655
- virtuais . 648, 649, 661
- Palavra . 54, 196
- Palmtop . Ver Computador de mão
- Parallel Load . 70
- Parâmetro . 152
- Passagem de parâmetros  
em registos . 346
- pela pilha . 346
- por referência . 345
- 0**
- PCI . 567
- PCI-Express . 534
- PDA . 518
- Pen-drives . 483, 496, 530
- Pentium . 22, 532, 554, 638
- PEPE . 147, 538, 695
- conjunto de instruções . 701
- exceções . 701
- Portos . 419
- pinos . 695
- programação . 708
- registos . 696
- auxiliares . 697
- principais . 696
- PEPE-8 . 119, 124, 145
- Periféricos . 165, 506, 542
- avaliação de desempenho . 555
- de comunicação . 487
- de entrada . 176
- de saída . 174
- mageados em memória . 413
- tipos de . 482
- Peripheral Component Interconnect . Ver PCI
- Personal Computer . Ver Computador pessoal
- Personal Digital Assistant . Ver PDA
- PIC . 476, 478
- Pico . 47
- PID . 656, 668
- Fila . 204, 321
- guarda temporária de registos . 334
- topo da . 325
- Pinos de interrupção . 462
- Pipeline . Ver Cadia de estágios
- Pipelining . Ver Processamento em estágios
- Pipes . Ver USB: condutas
- Pista . 484
- Pixels . 12, 409, 486, 660
- Placa-filha . 519
- Placa-mãe . 519
- PLACE . 291, 540
- Plug and play . 496
- Política  
de escrita . 636  
atrasada . 637, 642
- imediatas . 637
- de substituição  
aleatória . 636
- de blocos . 636
- FIFO . 636
- LRU . 636
- Polling . Ver Transferência por teste
- Ponte  
norte . 532
- sul . 532
- Pontos de paragem . 163, 165, 393, 728
- POP . 357, 705
- POPC . 706
- Porta série . 531
- Portabilidade . 283
- Portas  
lógicas . 34
- tristate . 65, 169, 446, 567
- Portos . 419
- de entrada . 507, 728
- de entradas/saída . 519, 537, 555, 713, 714
- de saída . 507, 727, 729
- tristate . 714
- Potências de 2 . 91
- PowerPC . 22, 27, 524
- Precisão . 737
- Predição  
de saito . 620
- dinâmica . 620
- estática . 620
- Princípio da localidade . 623
- Prioridade  
das exceções . 465
- dos processos . 667
- Process identifier . Ver PID
- Processador . 6, 542
- avaliação de desempenho . 547
- carregamento de dados . Ver Unidade de dados
- com dois núcleos . 525
- duo-core . Ver com dois núcleos
- núcleo do . 566, 568
- PEPE . 195, 204
- PEPE-8 . 145
- Processamento  
de informação . 5, 28, 114, 517
- em estágios . 565, 595, 596, 606, 665
- Processos . 665
- em execução . 667
- executáveis . 667
- sincronização . 671
- suporte para . 664
- supõe para . 664
- Programa . 6, 13
- de avaliação . 543, 545
- Dhrystone . 544
- Whetstone . 544
- principal . 459
- Programa de avaliação . 558
- Programação  
ciclo de desenvolvimento . 389
- com interrupções . 469
- com periféricos . 173
- concorrente . 665
- cooperativa . 679
- em alto nível . 280
- em linguagem assembly . 160, 394
- estruturação do código . 306
- sequencial . 665
- Programador . 13
- Programmable Interrupt Controller . Ver PIC
- Programmable Read Only Memories . Ver PROM
- Programmable Read Only Memory . See PROM
- PROM . 55, 115, 427
- Proteção . 682
- Pseudo-instrução . Ver Directiva
- Pull-down . 463
- Pull-up . 463
- Pull-ups . 714
- PUSH . 337, 705
- PUSHC . 706
- R**
- RAID . 531, 558
- Rajada . 514
- RAM . 115, 411, 535, Ver Memória
- Random . Ver Aleatório
- Random Access Memory . Ver RAM
- Ranbuta . 531
- RCCD . 698, 699
- RCCI . 698, 699
- RCMV . 662, 698, 700
- RCN . 464, 510, 581, 695, 698
- RE . 187, 204, 212, 217, 251, 461, 462, 464, 570,
- 612, 683
- Read Only Memory . Ver ROM, Ver ROM
- Recursividade . 356
- Rede local . 557
- Redes neuronais . 3
- Reduced Instruction Set Computers . Ver RISC
- Redundant Array of Independent Disks . Ver RAID
- Reference bit . Ver Bit de utilização
- Regime  
estacionário . 31, 646
- transitório . 31
- Register  
polling . 361
- Transfer Language . Ver RTL
- Transfer Notation . Ver RTN, RTL
- Registo . 63
- Apontador de Pilha . Ver SP

|                                          |                                                 |                                         |
|------------------------------------------|-------------------------------------------------|-----------------------------------------|
| Base da Tabela de Exceções - Ver BTE     | Contador de programa - Ver PC (Program Counter) | da Tabela de Páginas - Ver RTP          |
| de Configuração                          | da Cache de Dados - Ver RCD                     | da Cache de Instruções - Ver RCCI       |
| do Endereço de Memória - Ver RCMD        | da Memória Virtual - Ver RCMV                   | do Núcleo - Ver RCN                     |
| de Estado - Ver RE                       | de Instâncias - 568                             | Ver RI                                  |
| de Ligação - Ver RL                      | de Microinstruções - Ver RMI                    | do Identificador do Processo - Ver RPID |
| dos Operandos - Ver ROP                  | temporário - Ver TEMP                           | dos Registros - 63, 203                 |
| de deslocamento - 73                     | guarda temporária na pilha - 334                | banco de - 66, 184, 187, 204            |
| interseções - 596                        | reutilização de - 288, 360                      | de duas faces - 615                     |
| representação                            | de número - 87                                  | de um número em bits - 93               |
| em complemento para 2 - 94, 95           | em módulo e sinal - 94                          | em virgula flutuante - 735              |
| estrelado - 57, 162, 213, 480, 642, 683  | estrutura de dados - 62                         | de símbolos - 62                        |
| ET - 313, 327, 707                       | ETF - 317, 707                                  | de um número negativo - 94              |
| ETTF - 317, 707                          | return From Exception - Ver RFE                 | de uma pilha - 334                      |
| EFE - 466, 480, 707                      | Setup time - Ver Tempo de preparação            | SATA - 532                              |
| EL - 469, 567, 571, 581, 588, 603, 613   | Serial ATA - Ver SATA                           | Scheduling - Ver Escalonamento          |
| ESC - 552, 554                           | Serial Peripheral Interface - Ver SPI           | Seção crítica - 673                     |
| EL - 204, 316, 332, 696                  | Série de Fibonacci - 239                        | Sectores - 484                          |
| EL - 603, 605, 612                       | Servidores - 517                                | Segmentação - 648                       |
| OL - 262, 704                            | SET - 246, 703                                  | Segmentos - 648                         |
| OLC - 262, 704                           | Shannon - 38                                    | Serial ATA - Ver SATA                   |
| OM - 54, 115, 411                        | Shl - 262, 704                                  | Serial Peripheral Interface - Ver SPI   |
| de Mapeamento - 583                      | SHLA - 262, 704                                 | Série de Fibonacci - 239                |
| Flash - 398, 550, 555, 559               | Short - 93                                      | Servers - 262, 704                      |
| OP - 603, 608, 612                       | SHRA - 262, 704                                 | Servidores - 517                        |
| OR - 262, 704                            | SIMAC - 725                                     | Set - 57                                |
| ORC - 262, 704                           | Simétrico - 239                                 | Setup time - Ver Tempo de preparação    |
| de Mapeamento - 583                      | Simplificação algébrica - 41                    | Serial ATA - Ver SATA                   |
| Flash - 398, 550, 555, 559               | Simulador - 37, 272, 283, 312, 475, 539, 589,   | Serial Peripheral Interface - Ver SPI   |
| OP - 603, 608, 612                       | SIMAC - 725                                     | Serial ATA - Ver SATA                   |
| OR - 262, 704                            | desenho de circuitos - 725                      | Serial Peripheral Interface - Ver SPI   |
| ORC - 262, 704                           | simulação - 728                                 | Serial ATA - Ver SATA                   |
| de saída da ALU - Ver RSA                | Significando - 736                              | Serial ATA - Ver SATA                   |
| do identificador do processo - Ver RPID  | Sinal - 594, 621, 695, 708, 711, 725            | Serial ATA - Ver SATA                   |
| do tempo de vida - 667                   | Round robin - 667                               | Serial ATA - Ver SATA                   |
| do tempo de retorno - 316                | RPT - 662, 668, 698, 701                        | Serial ATA - Ver SATA                   |
| com CALL e RET (pilha) - 321             | RS - 232C - 492                                 | Serial ATA - Ver SATA                   |
| com CALLF e RETF (RL) - 317              | RSA - 603, 608, 612                             | Serial ATA - Ver SATA                   |
| contexto de chamada - 351                | RTC - 536                                       | Serial ATA - Ver SATA                   |
| Rotational delay - Ver Atraso rotacional | RTL - 128, 134, 154, 162, 212, 326, 480, 701    | Serial ATA - Ver SATA                   |
| Rotinas - 310                            | RTT - 128, Ver RTL                              | Serial ATA - Ver SATA                   |
| chamada e retorno - 316                  | RTP - 661, 663, 698, 701                        | Serial ATA - Ver SATA                   |

de controlo . 59, 128, 129, 130, 178, 429  
 de entrada . 33, 61, 495, 616  
 de saída . 34, 47, 61, 72, 117  
 de selecção . 169, 417  
 Sintonização  
 com sanfônicos . 675  
 com trincos lógicos . 671  
*Single-step* . Ver *Passo a Passo*  
 Síntese de circuitos combinatórios . 46  
 Sistemas  
 de tempo real . 469, 475  
 embutidos . 27, 401, 409, 518, 677  
 operativos . 15, 26, 396, 479, 641, 668  
 Site . 37, 695  
*Sior*. Ver *Ranhura*  
*Software* . 17, 26, 178, 272, See  
*Exception*. Ver *SWE*  
*open-source* . 25  
 SP . 204, 212, 312, 325, 356, 381, 666  
 SPEC . 545  
 SPI . 556  
 Stack . Ver *Pilha*  
 Stack Pointer . Ver *SP*  
 Start bit . 493  
 Stop bits . 493  
 Store . 187  
 String . Ver *Cadeia de caracteres*  
 STRING . 288  
 SUB . 238, 703  
 SUBB . 238, 703  
 Sun Microsystems . 25  
 Supercomputadores . 17, 517  
 SWAP . 218, 230, 673, 705  
 Swap file . 651, 653  
 Swapping . 652  
 SWE . 478, 480, 481, 590, 702, 707  
*System call*. Ver *Chamada ao sistema*

---

Tabela  
 de exceções . 460  
 de páginas . 649  
 de símbolos . 294, 855  
 de verdade . 35, 37, 47, 52, 100

Tabelas . 363  
 de apontadores . 374  
 de uma só dimensão . 363  
 multidimensionais . 368

TABLE . 288

Tampão de escrita . 637  
 Taxa  
 de acesso . 453  
 de atraso . 34, 37, 47, 60, 61, 73, 210, 455  
 de manutenção . 60, 455  
 de preparação . 60, 455  
 de procura de pista . 485  
 de transferência . 485  
 Temporizador . 540  
 Tensão  
 de alimentação . 30, 470  
 de entrada . 32  
 de saída . 32  
 margem de ruído . 32

Terabytes . 91  
 Termo mínimo . 40  
 Terra . 32, 36

TEST . 246, 703  
*Test and Set*. 673

Thrashing . 660  
*Time slice*. Ver *Faixa de tempo*  
 TLB . 654, 655, 663, 698, 700  
 Toggle . Ver *Transição alternada*  
 TPC . 545

Tradução de endereços . 645, 647  
 Transacção . 545

Transfer  
 rate . Ver *Taxa de transferência*  
*time* . Ver *Tempo de transferência*

Transferência  
 de uma constante para um registo . 219  
 entre regístos . 218  
 entre um registo e a memória . 223  
 modos de . 507

para memória de constante ou memória .  
 por DMA . 641, Ver *DMA*  
 por interrupções . 509  
 por processador de entradas/saiadas . 515  
 por teste . 507, 540, 717  
 taxa de . 485, 507

Transição  
 alternada . 61  
 no flanco ascendente . 59  
 no flanco descendente . 59

Transistor . 19, 30  
*Translation Lookaside Buffer*. Ver *TLB*  
 Transporte . 241, 570  
 Trapa . Ver *Armadilhas*  
 Triac . 471



## CURSO TÉCNICO DE HARDWARE – 5ª EDIÇÃO ACTUALIZADA



JOSÉ GOVEIA / ALBERTO MAGALHÃES

Os fabricantes de processadores são sempre os que têm mais protagonismo, pois estão constantemente a forjar os processadores até aos limites do silício. Com a colocação de dois e quatro núcleos nos processadores, chegaram a níveis de processamento nunca antes alcançados, pois uma nova tecnologia e arquitetura demoram a aparecer. Esta obra, actualizada com as mais recentes tecnologias, permite esclarecer o funcionamento e componentes de um processador e de outros componentes de um PC como memória, placas gráficas e placas principais, o componente mais importante de um PC. Todos os periféricos vêm aqui referidos, bem como a sua forma de funcionamento e como detectar e resolver avanços físicos ou por software. Se pretende montar uma rede de computadores e perceber o seu funcionamento com o protocolo TCP/IP, esta obra elucidá-o acerca deste assunto de uma forma simples e prática.

**Conteúdos:** Como montar um PC; Os processadores mais recentes e sua tecnologia, incluindo os Dual-Core; Os vários tipos de memórias existentes no mercado; Diferenças entre monitores CRT e TFT; Como optimizar um PC através do BIOS; Ferramentas de diagnóstico e resolução

## INTRODUÇÃO À PROGRAMAÇÃO EM JAVA

ANTÓNIO ADREGO DA ROCHA / OSVALDO ROCHA PACHECO

Esta obra tem como principal objectivo fornecer competências sólidas no desenvolvimento de programas de pequena e média complexidade usando a linguagem Java, versão 6. O entendimento dos autores que a introdução do ensino da programação não deve ser feita usando o paradigma da programação orientada a objectos, porque ele é demasiado complexo para uma primeira abordagem à programação. Em alternativa, acreditamos que é mais vantajoso introduzir a linguagem Java aplicando o paradigma da programação procedimental, cuja metodologia assenta na decomposição hierárquica das soluções, através de refinamentos sucessivos, como a forma mais natural de lidar com a complexidade. Especificamente, pretende-se atingir os seguintes objectivos:

**ISBN:** 978-972-772-823-9  
**Nº Pág.:** 432

**Conteúdos:** Compreensão clara do que é um computador; Desenvolvimento de estratégias para a especificação precisa do problema que se pretende resolver num computador; Normas de representação da informação num computador; Aprendizagem da linguagem de programação Java, versão 6; Estudo das principais estruturas de dados estáticas; Criação de tipos de dados

## INTRODUÇÃO À PROGRAMAÇÃO EM VISUAL BASIC 2010

ANTÓNIO GAMEIRO LOPES

O presente livro constitui um guia de iniciação ao fascinante mundo da Programação, tornando como base o Visual Basic 2010, na sua versão Express. Apresentando as principais ferramentas e funcionalidades desta linguagem, esta exposição não se limita a uma postura meramente descriptiva. Peço contrário, procura familiarizar o leitor com as diversas técnicas de programação, adopçando uma perspectiva fundamentalmente aplicada. Propõe diferentes possibilidades para a concretização de tarefas, salientando as respectivas vantagens e desvantagens, e alertando para os erros mais comuns. Inclui a implementação de vários Exemplos de aplicação, códigos completos que integram diferentes aspectos da matéria exposta, auxiliando, assim, a sua assimilação. Em complemento, os Problemas possibilham, ao leitor, uma avaliação dos seus conhecimentos, enquanto os Exercícios Propostos constituem sugestões para implementação, cobrindo diversos níveis de dificuldade.

**ISBN:** 978-972-772-444-3

**Nº Pág.:** 448  
**Conteúdos:** Conceitos fundamentais do paradigma da programação por objectos; Princípios e métodos de programação que permitem desenvolver aplicações JAVA; Construções da linguagem JAVA5 que suportam a PPO; Bluel, um ambiente muito simples de apoio ao desenvolvimento de aplicações JAVA

Contate-nos por e-mail:

[fca@fca.pt](mailto:fca@fca.pt)