Introdução aos Sistemas Digitais e Microprocessadores

Guilherme Arroz José Monteiro Arlindo Oliveira

23 de Maio de 2003

Conteúdo

Capítulo 9

Circuitos para Transferência de Dados

As técnicas de projecto de circuitos digitais, combinatórios e sequenciais, apresentadas nos capítulos anteriores permitem a realização de sistemas de baixa e média complexidade. O nível de detalhe a que estas técnicas são aplicadas é demasiado elevado para que possam ser usadas na concepção de circuitos de grande dimensão. Assim, no projecto de sistemas com uma funcionalidade mais complexa é necessário um nível de abstracção mais elevado de forma a esconder muitos detalhes e a tornar o problema manejável.

Neste capítulo descreve-se o projecto de sistemas digitais em termos de duas componentes. Uma é a Unidade de Processamento, também chamada de circuito de dados (ou *datapath*, em inglês), que contém toda a lógica que faz os cálculos propriamente ditos bem como os registos onde os dados são guardados. A segunda é a Unidade de Controlo que gere quais as operações que a unidade de processamento deve efectuar em cada ciclo de relógio.

Esta abordagem pressupõe que uma complexidade de processamento mais elevada requer em geral vários ciclos de relógio para se completar. De facto, operações acima de um certo nível de complexidade podem implicar um circuito lógico específico com uma dimensão tal que tornaria incomportável a sua realização na prática. Estas operações são assim divididas numa sequência de operações mais simples, estas sim facilmente realizáveis em hardware. A unidade de processamento é o circuito que disponibiliza estas operações mais simples e a unidade de controlo é o circuito que as sequencia de forma a realizar a operação complexa. Para permitir descrever de maneira clara o algoritmo de realização de operações complexas em termos das operações básicas da unidade de processamento, foram desenvolvidas linguagens de descrição de hardware. Um exemplo simples deste tipo de linguagens é apresentado na Secção 9.3, a ser usada no resto deste livro.

Embora as unidades de processamento possam ser projectadas para um fim específico, em muitos casos opta-se por usar unidades que disponibilizam um conjunto de operações aritméticas e lógicas típicas, chamadas Unidades Aritméticas e Lógicas ou ULA (em inglês, *Arithmetic and Logic Unit* ou *ALU*). Na Secção 9.4 descreve-se o exemplo de uma ULA, que será usada no processador P3, estudado no Capítulo 12.

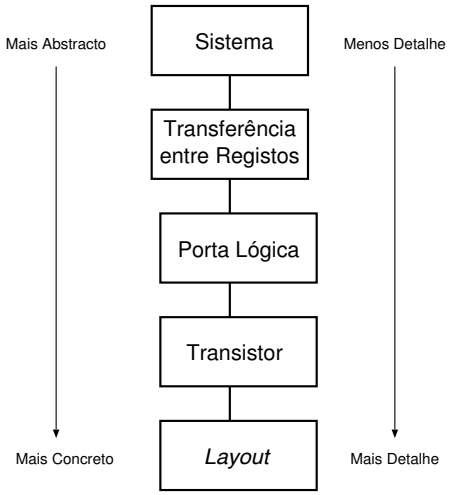


Figura 9.1: Diferentes níveis de abstracção no projecto de sistemas lógicos complexos.

9.1 Níveis de Abstracção

A abordagem usada neste capítulo para o projecto de circuitos corresponde a subir o nível de abstracção na descrição de circuitos lógicos. O projecto de sistemas complexos envolve diferentes níveis de abstracção, tendo o projecto início no mais abstracto, descendo-se sucessivamente para níveis mais concretos. A abordagem por níveis de abstracção torna possível o projecto de sistemas para os quais uma abordagem directa seria demasiado complexa.

De forma a dar uma ideia geral deste processo e um melhor contexto para os métodos de projecto apresentados neste livro, descrevem-se em seguida os diferentes níveis de abstracção tipicamente considerados no projecto de circuitos lógicos complexos. A Figura 9.1 representa estes níveis de abstracção ordenados de cima para baixo dos níveis mais abstractos para os níveis mais concretos. Dito de outra forma, nesta ordenação o grau de detalhe na descrição do sistema aumenta de cima para baixo.

O projecto tem início ao nível de sistema onde são feitas análises globais, mas muito gerais, sobre o projecto, nomeadamente sobre o número de sub-sistemas em que o sistema se deve dividir, qual o tipo de sub-sistemas e a sua forma de interligação. Termina ao nível de *layout* onde se vai a um grau de detalhe em que fica definido o caminho de cada interligação do circuito eléctrico, permitindo portanto a sua fabricação.

Em cada nível de abstracção, a análise é feita individualmente sobre módulos criados no nível imediatamente acima. Assim, o projecto é sucessivamente dividido em módulos mais pequenos, o que permite ir aumentando o nível de detalhe, mantendo-o sempre como um problema gerível em cada nível. É este processo de "dividir para conquistar" que faz o sucesso deste conceito de níveis de abstracção.

Por outro lado, num dado nível de abstracção é possível desenvolver o projecto sem necessitar de informação sobre níveis inferiores. Esta característica

esconde os detalhes de implementação, permitindo que o projectista se concentre em conseguir a melhor solução possível a esse nível.

Até ao capítulo anterior, toda a análise se debruçou sobre circuitos descritos ao nível da porta lógica. Este é o nível de abstracção intermédio, como se pode observar na hierarquia da Figura 9.1. Referiu-se atrás que a realização de portas lógicas é feita com o uso de transistores que, como se pode ver na figura, é um nível de abstracção imediatamente abaixo. No entanto, para a definição das técnicas de projecto apresentadas, nunca foi necessário saber em concreto como é que as portas lógicas iriam ser realizadas em termos de transistores.

Após obtida uma especificação ao nível de portas lógicas, o passo seguinte é convertê-las para transistores e fazer as ligações eléctricas. Para uma dada tecnologia de fabrico de circuitos, a cada porta lógica corresponde um esquema eléctrico de interligação de transistores. No entanto, para além deste mapeamento é necessário fazer o dimensionamento dos transistores de maneira a garantir, por exemplo, determinados tempos de atraso e consumos de potência.

Embora se possam realizar sistemas digitais com elementos discretos, é cada vez mais comum que a realização dos sistemas digitais seja em termos de circuito integrado. Neste caso, o passo final é converter esta descrição a nível de transistor, que no fundo é uma descrição de um circuito eléctrico, para máscaras que são usadas directamente no fabrico de circuitos integrados, a que se chama nível de *layout*. Estas máscaras definem linhas por onde passam certo tipos de elementos que constituem o circuito final. Por exemplo, as interligções são normalmente realizadas por linhas de metal, podendo existir várias camadas destas linhas e correspondendo a cada camada uma máscara. No caso da tecnologia CMOS, os transistores são formados pelo cruzamento de uma linha de polisilício por outra de dopagem de substracto, correspondendo a mais duas máscaras diferentes. Tipicamente, na passagem do nível de transistor para o nível de *layout*, o objectivo é escolher o caminho de cada linha de forma minimizar a área total ocupada pelo circuito. Estes dois níveis de abstracção, assim como o nível de sistema, saem fora do âmbito deste livro e portanto não serão discutidos.

Neste capítulo apresenta-se o nível de transferência entre registos, nível imediatamente acima do nível de porta lógica. De acordo com a filosofia dos níveis de abstracção, usam-se módulos de complexidade mais elevada, mas sem detalhar a forma como são construídos. Por exemplo, são utilizados directamente somadores sem haver a preocupação de como esses somadores serão realizados em termos de portas lógicas.

De referir que, para a maior parte das operações de projecto e optimização a cada nível de abstracção, o projectista pode recorrer a ferramentas de síntese de circuitos. A partir de uma descrição do circuito a um dado nível, estas ferramentas geram automaticamente o circuito correspondente num nível de abstracção mais baixo, optimizando-o tendo em conta um conjunto de parâmetros que o projectista pode especificar. A área de investigação de algoritmos de síntese lógica é uma área de intensa actividade, mas este é também um tema que sai fora do âmbito deste livro.

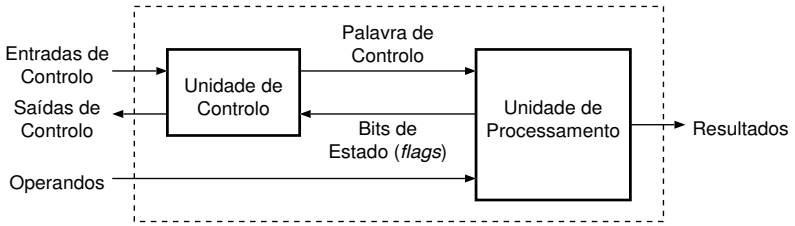


Figura 9.2: Estrutura de um sistema digital em termos de unidade de processamento e de unidade de controlo.

9.2 Separação entre Circuito de Dados e Circuito de Controlo

Um sistema digital pode ser construído de raiz usando básculas e portas lógicas discretas usando as técnicas de projecto de circuitos síncronos apresentadas nos capítulos anteriores. Porém, como discutido na secção prévia, esta aproximação tem grandes limitações. Por exemplo, qualquer dos métodos de geração de circuitos combinatórios descritos na Secção 3.3 é impraticável quando o número de variáveis de entrada é elevado.

Os sistemas digitais são, assim, normalmente estruturados em duas unidades principais, a *Unidade de Processamento* e a *Unidade de Controlo*, conforme está ilustrado na Figura 9.2. A unidade de processamento é constituída por pequenos módulos interligados para armazenar e processar a informação útil do sistema. Como indicado na figura, recebe as entradas a processar, ou seja, os operandos ou dados do exterior e calcula o resultado. A unidade de controlo é responsável por gerar os sinais de controlo que sequenciam as operações básicas da unidade de processamento de forma a que o sistema realize operações complexas. Ao conjunto de sinais de controlo que saem da unidade de controlo para a unidade de processamento chama-se *palavra de controlo*. A sequência de sinais de controlo gerada pela unidade de controlo pode depender dos resultados dos cálculos efectuados pela unidade de processamento. Esta informação é fornecida à unidade de controlo por *bits de estado* ou *flags*, cujo funcionamento será discutido mais à frente. A unidade de controlo pode também receber informação directamente do exterior, como por exemplo, um sinal externo que indica quando dar início à operação ou qual a operação a realizar. Em geral, existirão saídas de controlo para indicar para o exterior o estado da computação, como por exemplo, um sinal que indica o fim da operação.

Naturalmente, entre os ciclos de relógio necessários ao processamento de uma operação complexa ter-se-á que guardar valores temporários de cálculos intermédios. Estes são guardados em registos, disponíveis na unidade de processamento. As formas de interligação de registos e módulos de processamento foram apresentadas na Secção 7.5.4.

9.2.1 Exemplo de Motivação

Para ilustrar este processo de separação de uma operação complexa numa sequência de operações mais simples, considere-se o caso de uma multipli-

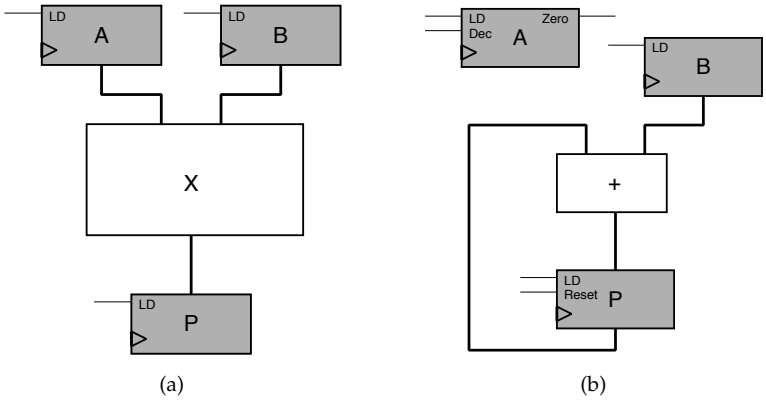


Figura 9.3: Circuitos multiplicadores: (a) bloco combinatório; (b) somas sucessivas.

cação. Na Secção 6.3 discutiu-se a realização lógica de multiplicadores, nomeadamente o facto de serem módulos complexos que requerem muito hardware.

Na Figura 9.3(a) é apresentada a realização directa para uma operação de multiplicação `P=A`$\times$`B`. Uma alternativa trivial a este circuito é realizar esta multiplicação através de somas sucessivas, somando `A` vezes o valor de `B`, conforme Figura 9.3(b). Assim, em vez de um módulo multiplicador será necessário apenas um somador, módulo significativamente mais simples. Para este exemplo, o circuito da Figura 9.3(b) funciona como a sua unidade de processamento.

O preço a pagar pela simplificação do hardware do circuito é a necessidade de um módulo adicional de controlo capaz de realizar a seguinte sequência de operações:

1. colocar registo `P` a zero.

2. carregar em `P` a soma de `P` com `B`.

3. decrementar o registo `A`.

4. se o registo `A` não chegou a zero, voltar para 2.

Para facilitar estas operações, os registos utilizados têm algumas funcionalidades adicionais. Assim, o registo `A` é de facto um contador, com uma entrada de controlo *Dec* para fazer a sua decrementação e uma saída de estado *Zero* que fica activa quando este registo contém o valor 0. Por seu lado, o registo `P` tem um sinal de controlo *Reset* que permite colocá-lo a zero.

Tendo disponíveis estes sinais de controlo, a unidade de controlo será um circuito sequencial que gera a sequência de sinais para realizar o algoritmo apresentado atrás. Na Figura 9.4 apresenta-se este algoritmo sob a forma de um fluxograma, ao qual foram adicionados dois sinais. A entrada *inicio* serve para indicar quando se deve dar início à operação. A saída fim indica quando a operação terminou. Este sinal é útil porque o número de ciclos de relógio necessários para esta operação não é fixo. Para cada operação de multiplicação

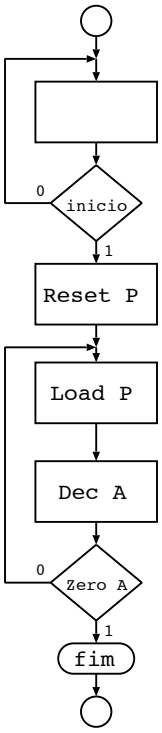


Figura 9.4: Fluxograma que descreve a unidade de controlo para a operação de multiplicação por somas sucessivas.

o tempo de processamento é determinado pelo valor inicial de `A` e portanto é importante que o sistema possa dar esta indicação para o exterior.

À descrição da Figura 9.4 é possível aplicar qualquer das técnicas de projecto de circuitos sequenciais apresentadas no Capítulo 8.3. Nas secções seguintes, apresenta-se uma linguagem de descrição que permite descrever as operações básicas da unidade de processamento, facilitando a composição de um circuito sequencial que realize a operação complexa desejada.

Este exemplo ilustra como se pode reduzir a complexidade do circuito a implementar, traduzindo operações complexas numa sequência de operações mais simples. Existe assim um compromisso que o projectista pode explorar entre a complexidade da unidade de processamento a implementar e o tempo (em termos de ciclos de relógio) que a operação demora a realizar. Em geral, quanto mais simples as operações disponíveis menor o hardware necessário na unidade de processamento, mas maior a sequência de operações, correspondendo portanto a um tempo total de execução maior.

O exemplo aqui apresentado é simplesmente ilustrativo. Embora na prática se evitem de facto os módulos multiplicadores combinatórios devido à sua complexidade, existem métodos sequenciais mais eficientes do que o aqui apresentado. Além disso, este exemplo discute duas alternativas possíveis, uma combinatória, outra sequencial. Em muitos casos, as operações são de tal modo complexas que a alternativa de um módulo combinatório puro está à partida fora de causa.

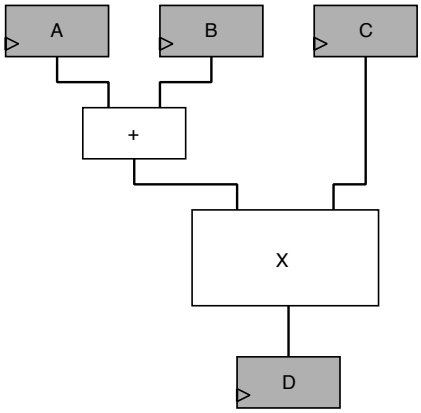


Figura 9.5: Unidade de processamento para realizar a operação $D = (A + B) \times C$.

9.2.2 Unidade de Processamento

As unidades de processamento são tipicamente construídas usando os módulos combinatórios de média complexidade estudados nos Capítulos 5 e 6 e registos estudados no Capítulo 7.5. Sobre a informação contida nos registos podem ser realizadas operações cujo resultado pode ser guardado no mesmo registo, noutro registo ou mesmo numa posição de memória. Por exemplo, se se pretender um sistema para uma funcionalidade específica como o cálculo de `D=(A+B)×C`, uma possível unidade de processamento seria simplesmente a indicada na Figura 9.5. Pela forma como o circuito da Figura 9.5 está desenhado, todos os ciclos de relógio o registo `D` é actualizado com o valor `(A+B)×C`, em que os valores de `A`, `B` e `C` são os destes registos no início de cada ciclo de relógio.

Existe alguma liberdade na construção da unidade de processamento. Em primeiro lugar, o projectista tem ao seu dispôr um conjunto de módulos, sejam eles aritméticos, lógicos ou outros, pelos quais poderá optar dependendo da operação a realizar. Poderá ainda construir módulos novos e específicos usando as técnicas de projecto do Capítulo 3. No exemplo anterior, assumiu-se que estavam disponíveis módulos somadores, módulos multiplicadores e registos.

Em segundo lugar, em muitos casos é possível fazer compromissos entre a complexidade da unidade de processamento e a complexidade da unidade de controlo, tentando-se arranjar um equilíbrio entre a quantidade de hardware necessário para o circuito e o desempenho deste. Concretizando para o exemplo anterior, para poupar hardware, em vez de se usar um módulo multiplicador, poder-se-ia optar por realizar a multiplicação por somas sucessivas, como discutido na secção anterior. Para isto seria necessário um registo temporário onde se somaria k vezes o valor do registo `C`, sendo k o valor do resultado `A+B`. A desvantagem óbvia é que a operação que antes se realizava num único ciclo de relógio agora necessita de n ciclos para a multiplicação, mais um ciclo para a soma original. A complexidade da unidade de controlo aumenta pois agora tem que controlar o número de vezes que a soma da multiplicação se efectua.

Outro ponto a considerar no projecto de uma unidade de processamento

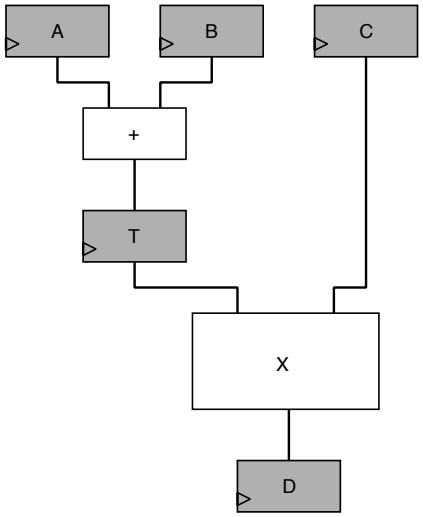


Figura 9.6: Redução do período de relógio para a unidade de processamento que realiza a operação $D = (A + B) \times C$.

está relacionado com a frequência máxima permitida para o sinal de relógio. Como referido na Secção 8.3, para o correcto funcionamento do sistema, o período de relógio terá que ser sempre superior ao maior atraso na lógica entre 2 registos da unidade de processamento. Para o exemplo da Figura 9.5, $T_{CLK} > t_{\text{soma}} + t_{\text{mult}} + t_{\text{reg}}$. Portanto, mesmo que existam operações muito simples no sistema, esta será sempre a granularidade mínima de duração de uma operação. Para reduzir esta granularidade, os módulos podem ser intercalados com registos, como ilustrado na Figura 9.6 onde o registo `T` guarda temporariamente o resultado da soma. Neste caso, a operação `D=(A+B)×C` demora 2 ciclos de relógio, um para calcular a soma `T=A+B` e outro para calcular a multiplicação `D=T×C`, mas o ciclo de relógio foi reduzido para $T_{CLK} > \max(t_{\text{soma}} + t_{\text{reg}}, t_{\text{mult}} + t_{\text{reg}}) = t_{\text{mult}} + t_{\text{reg}}$. De sublinhar que esta optimização pode ter um impacto enorme no desempenho do sistema pois esta redução de ciclo de relógio tem influência em todas as operações simples do sistema.

Uma vez escolhidos os módulos para a unidade de processamento, ficam definidas quais as operações que podem ser realizadas sobre a informação armazenada nos registos a cada ciclo de relógio. Estas operações são chamadas de *micro-operações* e estas definem os pontos de controlo para a unidade de controlo.

Apresenta-se mais à frente neste capítulo um exemplo de uma unidade de processamento construída para um fim específico (Secção 9.3.2). No entanto, podem-se contruir unidades de processamento para as quais não esteja definida à partida uma aplicação em particular, como é o caso das unidades de processamento de computadores. Para estas situações, define-se um conjunto de micro-operações suficientemente genérico e poderoso para acomodar eficientemente um conjunto grande de funcionalidades. A estas unidades de processamento dá-se o nome de Unidade Lógica e Aritmética ou ULA (em inglês, *Arithmetic and Logic Unit* ou *ALU*). No final deste capítulo exemplifica-se a construção de uma unidade de processamento deste tipo.

9.2.3 Unidade de Controlo

A unidade de controlo é responsável por definir quais as micro-operações que são executadas na unidade de processamento em cada ciclo de relógio. Estas micro-operações são definidas através de um conjunto de sinais de controlo aceites pela unidade de processamento, a cujo conjunto se chama palavra de controlo. As unidades de controlo têm dois tipos de entradas. Por um lado têm entradas externas que controlam o funcionamento global do sistema. Por outro lado, têm entradas provenientes da unidade de processamento, os bits de estado. A partir destes, a unidade de controlo tem informação acerca do resultado de uma dada operação, podendo desencadear diferentes operações com base nesta informação.

Pegando novamente no exemplo da Secção 9.2.1, como se pode observar pelo fluxograma da unidade de controlo apresentado na Figura 9.4, esta unidade tem duas entradas, uma entrada de controlo externa e um bit de estado da unidade de processamento. O sinal $inicio$ é uma entrada de controlo que indica quando se deve iniciar um novo cálculo. O sinal $ZeroA$ é uma entrada proveniente da unidade de processamento que informa a unidade de controlo do resultado da operação anterior, permitindo a esta decidir qual a acção a tomar.

Para além das saídas que formam a palavra de controlo, as unidades de controlo podem também ter saídas de controlo que comunicam com entidades externas indicando o estado do sistema. Um exemplo deste tipo de saídas é o fim da Figura 9.4. Neste caso, como o tempo de cálculo não é fixo, mas sim dependente dos valores dos operandos, é necessário que o sistema informe o exterior acerca do estado do cálculo, nomeadamente, se este já terminou ou não.

Estando definida a unidade de processamento, o projectista tem que definir a sequência de micro-operações necessárias para realizar a funcionalidade pretendida para o sistema. O projecto da unidade de controlo não é mais do que o projecto de um circuito sequencial, como estudado no Capítulo 8. Embora a sua realização possa ser feita usando qualquer das técnicas apresentadas nesse capítulo, o número de entradas (bits de estado e entradas de controlo) e saídas (largura da palavra de controlo e saídas de controlo) é em geral muito elevado, levando a que as técnicas mais comuns usadas no projecto de unidades de controlo assentem em controladores micro-programados, estudados na Secção 8.3.3.

9.3 Linguagem de Descrição de Hardware

A especificação de sistemas mais complexos necessita de uma linguagem de descrição com um nível de abstracção mais elevado do que as funções booleanas ou os diagramas de estado de forma a esconder os detalhes do sistema e assim permitir descrições sucintas. Estas linguagens chamam-se *linguagens de descrição de hardware*.

Embora o nível de abstracção possa variar, em geral esta descrição é feita ao nível de transferência entre registos, em que o projectista define quais as micro-operações efectuadas entre registos em cada ciclo de relógio. A forma como as micro-operações são realizadas em termos de portas lógicas não é de-

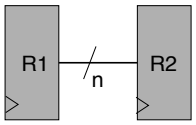


Figura 9.7: Circuito lógico correspondente à instrução R2 ← R1.

finida na descrição do sistema, pressupondo-se a criação de uma unidade de processamento que permita executar essas micro-operações.

Existem vários formatos possíveis para as linguagens de descrição de hardware. A linguagem adoptada neste livro e descrita em seguida é muito simples e permite apenas especificar as micro-operações que o sistema deve realizar. Outras linguagens, como *VHDL* e *Verilog*, têm um nível de descrição igual ao de uma linguagem de programação usual, mas com a diferença fundamental de terem que acomodar o facto do hardware ser intrinsecamente concorrente. Para isso, as linguagens de descrição de hardware definem blocos de código que descrevem uma operação e portanto são "sequenciais", mas os diferentes blocos funcionam em paralelo.

9.3.1 Linguagem de Transferência entre Registos

Define-se nesta secção a linguagem simples de descrição de hardware a nível de transferência entre registos a ser usada nos capítulos seguintes. Nesta linguagem, a especificação de um sistema digital é conseguida através de um conjunto de micro-operações que definem a funcionalidade pretendida.

A transferência de informação de um registo para outro, ou mais exactamente a replicação dessa informação, é designada em forma simbólica pela instrução:

$$\mathtt{R2} \leftarrow \mathtt{R1}$$

que significa `R2` recebe o conteúdo de `R1`. O valor guardado em `R1` não é alterado. Em termos de circuito lógico, a instrução anterior corresponde à Figura 9.7.

Se para além desta micro-operação simples de transferência houver algum processamento, isso é explicitado, por exemplo:

$$\mathtt{R1} \leftarrow \mathtt{R2} + \mathtt{R5}$$

ou

$$\mathtt{R7} \leftarrow \mathtt{R5} \vee \mathtt{R4}.$$

A primeira destas instruções será realizada pelo circuito lógico da Figura 9.8.

Quando se dão duas transferências em simultâneo, podem-se agrupar micro-operações separando-as por vírgulas:

$$\mathtt{R3} \leftarrow \mathtt{R6} - \mathtt{R2},\ \mathtt{R9} \leftarrow \mathtt{R2} \times \mathtt{R4}.$$

O circuito lógico correspondente é o da Figura 9.9.

Para além de registos, os operandos das micro-operações poderão ser também valores constantes ou referências a posições de memória. A indicação de uma posição de memória é feita usando `M[endereço]`, em que *endereço*

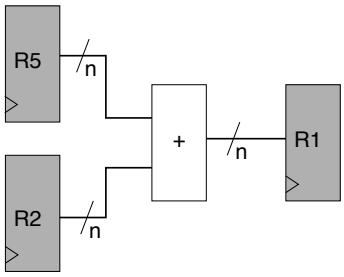


Figura 9.8: Circuito lógico correspondente à instrução $\mathtt{R1} \leftarrow \mathtt{R2} + \mathtt{R5}$.

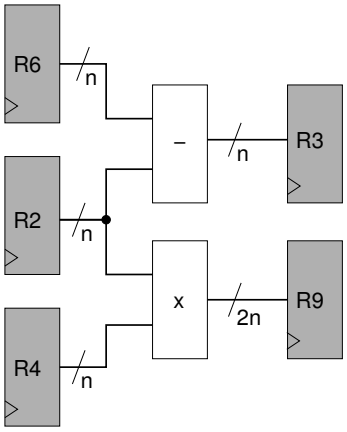


Figura 9.9: Circuito lógico correspondente à instrução $\mathtt{R3} \leftarrow \mathtt{R6} - \mathtt{R2}$, $\mathtt{R9} \leftarrow \mathtt{R2} \times \mathtt{R4}$.

poderá por seu lado ser um valor constante ou um registo. Exemplos possíveis de micro-operações serão:

$$\mathtt{R1} \leftarrow \mathtt{M}[\mathtt{32d}] \vee \mathtt{R7}$$

$$\mathtt{M}[\mathtt{R2}] \leftarrow \mathtt{R2} \oplus \mathtt{55h}$$

Pela forma como estão expressas, as micro-operações apresentadas até aqui ocorrem em *todos* os ciclos de relógio. Num sistema digital não se pretende normalmente que uma transferência, com ou sem processamento, ocorra sempre que surge um impulso de relógio. Para indicar que uma certa operação só deve ocorrer na presença de certas condições, usa-se o formato *condição: micro-operação*. Por exemplo, uma instrução como

$$K1 : \mathtt{R2} \leftarrow \mathtt{R1}$$

tem uma tradução num circuito lógico como o da Figura 9.10. Para haver apenas uma transferência, a linha $K1$ tem que permanecer a 1 apenas durante uma transição ascendente do sinal de relógio. Um exemplo de uma situação deste tipo está ilustrada na Figura 9.11, onde a transferência se dá apenas no instante t.

A condição de activação da micro-operação pode ser uma expressão lógica genérica. Por exemplo,

$$X \cdot Y : \mathtt{R5} \leftarrow \mathtt{R0}$$

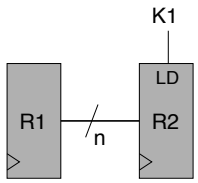


Figura 9.10: Circuito lógico correspondente à instrução $K1 : \texttt{R2} \leftarrow \texttt{R1}$.

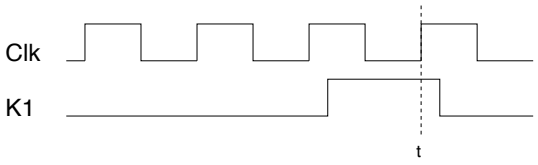


Figura 9.11: Diagrama temporal que garante uma transferência única de `R1` para `R2`.

significa que a cada ciclo de relógio apenas se transfere o conteúdo de `R0` para `R5` se ambos os sinais X e Y estiverem a 1.

Por vezes é útil trabalhar não com todos os bits guardados num registo, mas apenas com um subconjunto dos seus bits. Para indicar que uma micro-operação apenas se aplica a parte dos bits de um registo, a seguir ao nome do registo indica-se entre parêntesis quais os bits envolvidos. Por exemplo, se se prentender trabalhar apenas com o bit 5 de um registo `R2`, isso seria indicado por `R2(5)`. Para uma gama de bits, a representação é a seguinte:

$$\texttt{R3}(15\text{–}8) \leftarrow \texttt{R6}(7\text{–}0)$$

que neste caso indica que o octeto menos significativo de `R6` é copiado para o octeto mais significativo de `R3`. Naturalmente, terá que haver o cuidado de ser consistente no número de bits envolvidos numa micro-operação.

Por outro lado, pode ser necessário fazer a concatenação de vários registos para uma dada operação, o que é indicado pelo operador $|$. Por exemplo,

$$\texttt{R7}|\texttt{R6} \leftarrow \texttt{R3} \times \texttt{R2}$$

significa que o resultado da multiplicação de `R3` por `R2`, cujo resultado, como referido anteriormente, precisa do dobro dos bits dos operandos, ficará guardado em dois registos, `R7` e `R6`, em que `R7` tem a parte mais significativa e `R6` a menos significativa.

Com esta linguagem simples é possível descrever unidades de processamento com uma complexidade arbitrária. Esta descrição indica quais as micro-operações que podem ser executadas na unidade de processamento e sob que condições elas ocorrem. Dada esta descrição, o projectista pode desenhar a unidade de controlo que gera a sequência de sinais de controlo para realizar a operação pretendida.

De notar que esta linguagem não define completamente a implementação do sistema. Muitas vezes existe alguma liberdade de escolha pois é possível considerar diferentes circuitos que realizam uma mesma funcionalidade. Por

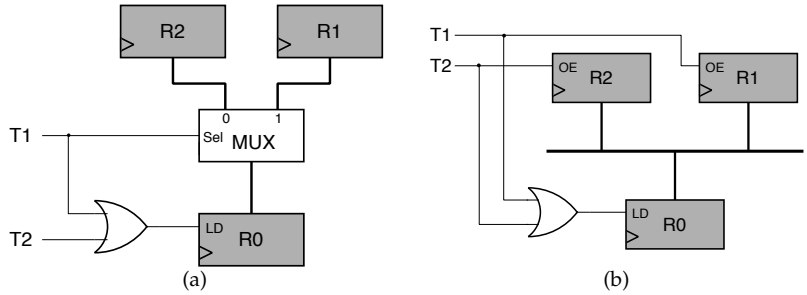


Figura 9.12: Circuitos equivalentes para realizar uma transferência de duas fontes possíveis.

Maior_Divisor_Comum(X, Y**)**
1. enquanto (Y≠0){
2. se X≥Y
3. então X=X-Y
4. se não, troca X com Y
5. }
6. resultado em X

Figura 9.13: Pseudo-código do algoritmo para o cálculo do máximo divisor comum.

exemplo, considere-se o caso comum de um registo que pode ser carregado a partir de uma de duas fontes:

$$\begin{aligned} T1: &\quad \texttt{R0} \leftarrow \texttt{R1} \\ \overline{T1} \cdot T2: &\quad \texttt{R0} \leftarrow \texttt{R2} \end{aligned}$$

Como discutido na Secção 7.5.4, estes registos podem estar ligados através de multiplexadores ou através de barramentos. Estas duas situações têm a representação em termos de circuito lógico indicada nas Figuras 9.12(a) e 9.12(b), respectivamente. Notar que no caso do barramento se usa menos hardware, mas há que ter o cuidado de garantir que os sinais $T1$ e $T2$ nunca estão activos em simultâneo.

9.3.2 Exemplo: Máximo Divisor Comum

De forma a ilustrar os conceitos apresentados atrás, desenvolve-se nesta secção um exemplo completo. Considere-se que se pretende projectar um sistema para calcular o máximo divisor comum de dois números inteiros positivos de n bits. Um algoritmo conhecido para realizar esta operação está indicado em pseudo-código na Figura 9.13.

Por palavras, neste algoritmo subtrai-se sucessivamente o menor dos números ao maior até que o resultado desta subtracção seja 0. Quando isso acontece, o algoritmo termina e o resultado é o valor final do outro operando. Não é

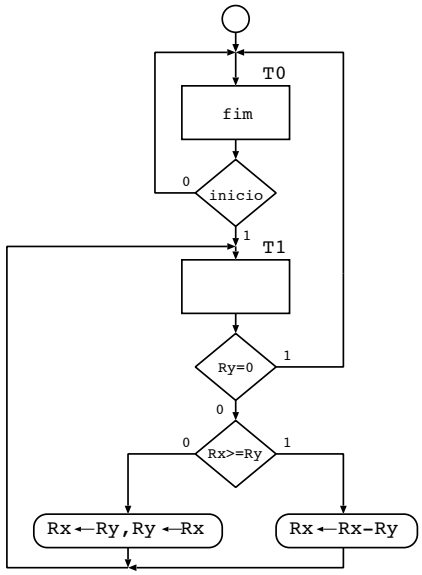


Figura 9.14: Fluxograma do algoritmo para o máximo divisor comum.

um algoritmo muito eficiente pois, por vezes, demora muito a terminar, mas é simples de realizar.

Assume-se que na especificação do sistema se indica que os operandos X e Y se encontram inicialmente guardados em dois registos, designados por `Rx` e `Ry`. Além disso, essa especificação indica que existem dois sinais de controlo, um sinal de entrada *inicio* para indicar que os registos `Rx` e `Ry` foram carregados com os operandos e que se deve dar início ao cálculo do maior divisor comum entre eles, um sinal de saída fim que assinala o fim deste cálculo. Para um funcionamento correcto do sistema, o sinal *inicio* e os registos `Rx` e `Ry` só devem ser alterados do exterior quando o sinal fim estiver activo.

Tendo em conta estas especificações, o algoritmo da Figura 9.13 pode ser apresentado sob a forma de fluxograma, como o apresentado na Figura 9.14. O estado $T0$ representa um estado de espera, onde nada acontece até que seja accionado o sinal *inicio*. Neste estado a saída de controlo fim está activa. Quando a entrada *inicio* vai a 1, o sistema avança para o estado $T1$, onde todo o processamento é realizado. O sistema mantém-se neste estado enquanto `Ry` não chegar a zero, situação em que terminou o cálculo, regressando o sistema a $T0$. Enquanto em $T1$, em cada ciclo os valores relativos de `Rx` e `Ry` são testados e, de acordo com este teste, ou se trocam os valores destes ou se subtrai `Ry` a `Rx`. Neste fluxograma os registos `Rx` e `Ry` estão a ser re-utilizados durante a operação uma vez que a cada ciclo de relógio não há novos valores a serem criados que necessitem de novos registos.

Neste ponto é necessário decidir a funcionalidade da unidade de processamento, ou seja, quais as micro-operações que estarão disponíveis. Para este exemplo, as micro-operações necessárias são simples, optando-se por incluí-las na unidade de processamento. Em geral, no entanto, poderá não ser este o caso, quer pela complexidade das operações quer por se estar a usar uma unidade de processamento predefinida. Nesse caso ter-se-á que subdividir as operações

1.		$xMy \leftarrow (\texttt{Rx} \geq \texttt{Ry}), Zy \leftarrow (\texttt{Ry} = 0)$
2.	$T0$:	$fim \leftarrow 1$
3.	$T0 \cdot inicio$:	$T0 \leftarrow 0, T1 \leftarrow 1, fim \leftarrow 0$
4.	$T1 \cdot Zy$:	$T0 \leftarrow 1, T1 \leftarrow 0$
5.	$T1 \cdot \overline{Zy} \cdot xMy$:	$\texttt{Rx} \leftarrow \texttt{Rx} - \texttt{Ry}$
6.	$T1 \cdot \overline{Zy} \cdot \overline{xMy}$:	$\texttt{Rx} \leftarrow \texttt{Ry}, \texttt{Ry} \leftarrow \texttt{Rx}$

Figura 9.15: Descrição a nível de transferência entre registos do algoritmo para o cálculo do máximo divisor comum.

em sub-operações mais simples que já sejam suportadas por micro-operações da unidade de processamento. Esta situação é estudada na Secção 9.4.7.

Por observação do fluxograma da Figura 9.14, as operações necessárias são apenas a transferência de valores entre registos e uma subtracção. Estas indicam a necessidade de registos com sinal de controlo *load*, LDx e LDy, e um subtractor.

Além das operações, há três condições a serem testadas: $inicio$ a 1, para o que não é necessário hardware específico; o valor do registo `Ry` ter chegado a 0; o valor `Rx` ser maior do que o valor de `Ry`. Para a segunda destas condições, pode-se assumir a existência de um comparador com a constante 0, o que é realizado através de uma porta NOR de n entradas, onde estão ligados todos os bits à saída de `Ry`. O mais usual é utilizar para `Ry` um registo que inclui já esta porta NOR e portanto com uma saída de controlo, Zy, que indica quando o valor lá guardado é zero. Para a terceira condição é necessário um comparador normal, ligado a `Rx` e `Ry`, e a cujo sinal de saída se atribuiu o nome xMy.

Tendo em conta a funcionalidade da unidade de processamento, o fluxograma da Figura 9.14 pode ser traduzido em linguagem de transferência entre registos como indicado na Figura 9.15. Nesta figura, os sinais $T0$ e $T1$ controlam a sequência de execução das restantes micro-operações e portanto serão gerados pela unidade de controlo.

Como referido na secção anterior, a cada ciclo de relógio qualquer das micro-operações das linhas da Figura 9.15 pode ocorrer. O que define quais as que de facto ocorrem são as condições indicadas na linha. Neste exemplo, para um funcionamento correcto, apenas um dos sinais da unidade de controlo, $T0$ e $T1$, estará activo. Com $T0$=1, apenas as operações das linhas 2 e 3 poderão ocorrer. Com $T1$=1, poderão ocorrer as das três últimas linhas. No entanto, é fácil ver que as condições destas três linhas, geradas pelos sinais Zy e xMy, são disjuntas, pelo que de facto apenas as micro-operações de uma das três últimas linhas da Figura 9.15 podem ocorrer em cada ciclo de relógio.

Examinando linha a linha este código, pode-se observar que na linha 1 os sinais de teste Zy e xMy são gerados em todos os ciclos de relógio. A linha 2 indica que o sinal fim se mantém activo no estado $T0$. A linha 3 só ocorre se o sistema estiver no estado $T0$ e o sinal de $inicio$ ficar activo, passando o controlo para o estado $T1$. Portanto, com o sistema no estado $T0$ e a linha $inicio$ desactiva, o sistema mantém-se indefinidamente neste estado e nada mais acontece. A linha 4 corresponde ao oposto, que é detectar que o registo `Ry` chegou a zero e que portanto o cálculo terminou. A acção correspondente é passar para o estado $T0$, que por sua vez irá activar a saída fim. As linhas 5 e 6 poderão ser activadas durante o decorrer do cálculo, em que o estado é $T1$. A linha 5 fica

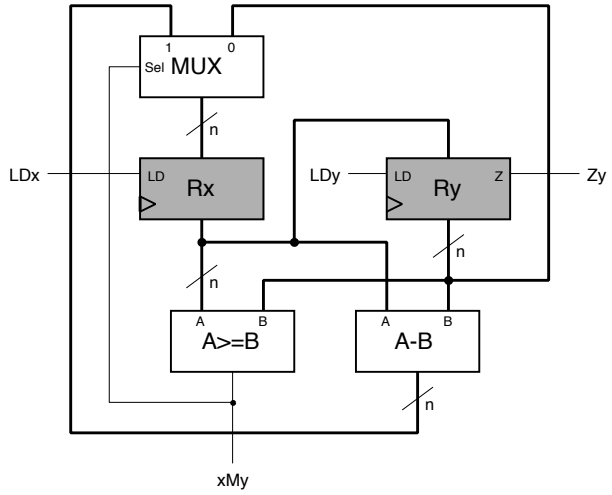


Figura 9.16: Unidade de processamento para o máximo divisor comum.

activa quando Rx $>$ Ry, subtraindo-se Ry a Rx. A linha 6 fica activa quando Rx $\leq$ Ry, activando-se a troca dos valores de Ry e Rx.

Traduzindo para circuito lógico a descrição a nível de transferência entre registos da Figura 9.15, apresenta-se na Figura 9.16 a unidade de processamento para este problema. Os sinais LDx e LDy formam a palavra de controlo da unidade de controlo, indicando em que ciclos de relógio os registos são actualizados. Os sinais Zy e xMy são os bits de estado da unidade de processamento e são entradas para a unidade de controlo. De notar que o controlo do multiplexador à entrada do registo Rx é feito directamente pela saída do comparador pois o carregamento para este registo é controlado pelo sinal LDx. Enquanto LDx estiver inactivo não importa qual o valor que está à entrada de Rx, e sempre que estiver activo a saída do multiplexador será a correcta.

O controlo desta unidade de processamento está indicado na Figura 9.17 e foi obtido a partir do fluxograma da Figura 9.14 por simples substituição das operações pelos sinais de controlo das micro-operações correspondentes e dos testes pelos bits de estado. Assim, o teste Ry=0 corresponde a testar se Zy=1 e Rx$>$Ry a testar se xMy=1. Por seu lado, para a operação Rx$\leftarrow$ Rx $-$ Ry basta activar o sinal de *load* do registo Rx, e para a operação de troca de valores entre os registos activam-se ambos os sinais de *load*. Notar que a selecção do valor à entrada de Rx, Rx$-$Ry ou Ry, é controlado directamente na unidade de processamento pelo sinal xMy.

Esta unidade de controlo é tão simples que não são necessárias técnicas sofisticadas para a sua construção. Usando a técnica de síntese de máquinas de estados da Secção 8.3.1, basta usar uma báscula tipo D para guardar o estado, codificando-se os estados da seguinte forma: 0 corresponde a $T0$; e 1 a $T1$. A síntese da lógica é trivial, chegando-se ao circuito da Figura 9.18.

Os circuitos das Figuras 9.16 e 9.18 juntos, conforme a Figura 9.2, realizam a funcionalidade pretendida de cálculo do máximo divisor comum entre dois números. A Figura 9.19 ilustra o funcionamento destes circuitos para o caso de $X = 54$ e $Y = 36$, em que se assumiu que os registos são actualizados no flanco

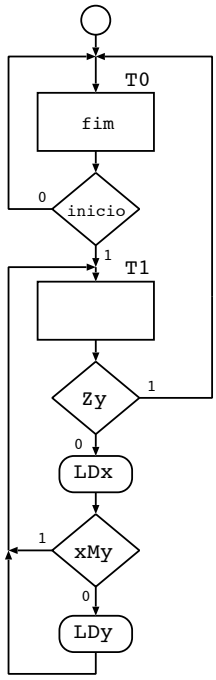


Figura 9.17: Fluxograma da unidade de controlo para o máximo divisor comum tendo em vista a unidade de processamento da Figura 9.16.

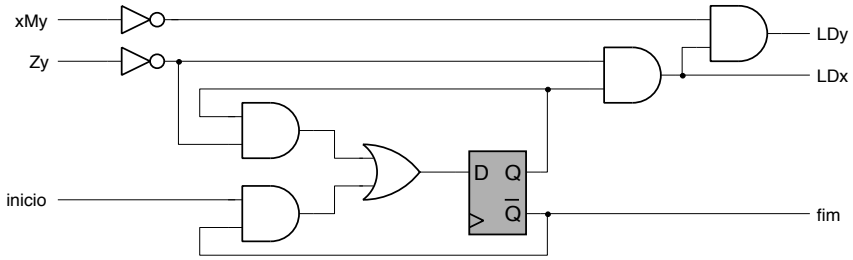


Figura 9.18: Unidade de controlo para o máximo divisor comum.

ascendente do sinal de relógio.

9.4 Unidades Lógicas e Aritméticas

No exemplo apresentado acima, a unidade de processamento foi projectada para um fim específico. De facto, essa será sem dúvida a abordagem mais eficiente para realizar um sistema digital pois assim inclui-se na unidade de processamento todos, e apenas, os módulos necessários às micro-operações a realizar, interligados de forma óptima. No entanto, em muitos casos, tal não é possível por o número de micro-operações ou a complexidade destas ser muito elevado.

Nestes casos, projecta-se para a unidade de processamento um circuito com

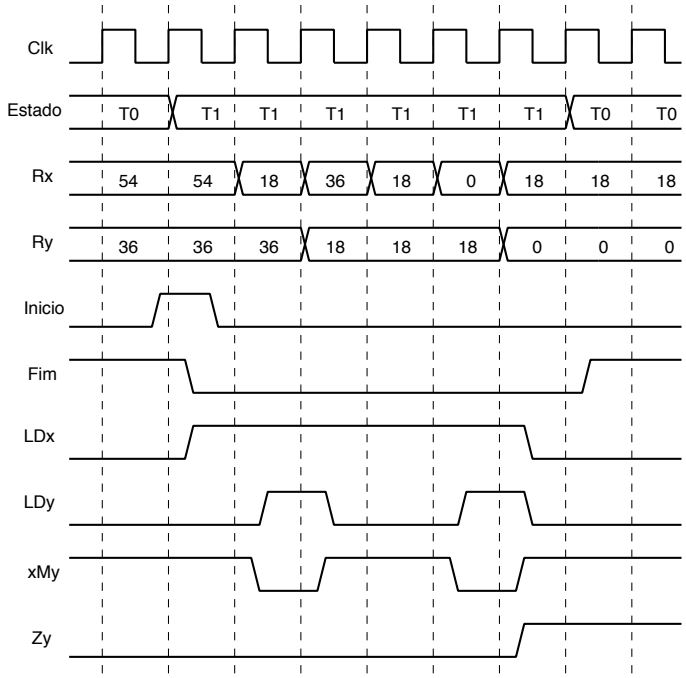


Figura 9.19: Diagrama temporal para o cálculo do máximo divisor comum entre $X = 54$ e $Y = 36$.

as funcionalidades básicas, em termos de operações quer aritméticas quer lógicas, associado a um conjunto de registos genéricos. Ao circuito que realiza as operações chama-se *Unidade Lógica e Aritmética* ou *ULA* (em inglês, *Arithmetic and Logic Unit* ou *ALU*). O número, tipo e complexidade das micro-operações incluídas numa ULA pode variar grandemente dependendo do problema em vista. Aplica-se aqui o princípio referido na Secção 9.2.2 de que é possível simplificar a unidade de processamento por transferência de alguma complexidade para a unidade de controlo. As operações mais complexas são assim realizadas por uma sequência de micro-operações disponíveis numa unidade de processamento mais simples.

A ULA trabalha normalmente junto com um banco de registos (ver Secção 7.5.4) para guardar os operandos, resultados das operações e valores temporários, necessários quando uma operação complexa é substituída por uma sequência de operações mais simples. Também aqui pode haver uma grande variação de caso para caso em termos do número de registos disponíveis neste banco.

Uma unidade de processamento deste tipo está representada na Figura 9.20. Assume-se o caso mais geral, em que a ULA aceita 2 operandos e tem um resultado único. A palavra de controlo conterá informação para a selecção de quais os 2 registos que servem como operandos e qual o registo destino para guardar o resultado. A especificação da micro-operação a realizar pela ULA é também parte da palavra de controlo. Por seu lado, a ULA gera um conjunto de bits de estado que, tal como referido anteriormente, podem ser usados pela unidade de controlo para tomar decisões em termos de execução dependendo

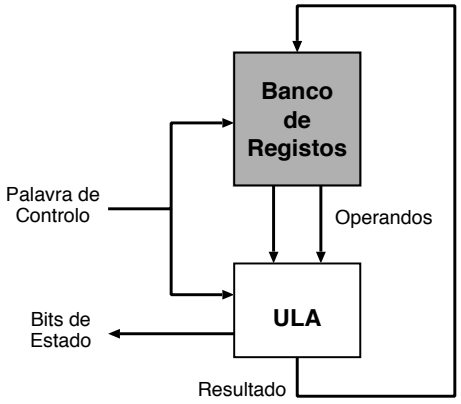


Figura 9.20: Exemplo de unidade de processamento com ULA mais banco de registos.

do resultado de uma micro-operação.

9.4.1 Estrutura de uma ULA

Para ilustrar estes conceitos, projecta-se em seguida uma ULA, que será posteriormente usada na unidade de processamento do processador P3 apresentado no Capítulo 12. A ULA definiu-se como um circuito que realiza operações aritméticas e lógicas básicas. Uma primeira observação é que estes dois tipos de operações não têm muito em comum. Assim, optou-se aqui por considerar em separado unidades para realizar cada um destes tipos de operações, a unidade aritmética e a unidade lógica. Além destas, considera-se uma terceira unidade para realizar o deslocamento à direita e à esquerda, funcionalidade semelhante à dos registos de deslocamento apresentados na Secção 7.5.2, mas neste caso efectuada por lógica puramente combinatória. Optou-se também por considerar esta unidade de deslocamento em separado pois corresponde a uma funcionalidade distinta das outras unidades. As operações realizadas pela unidade de deslocamento não se classificam em termos de operações
aritméticas e lógicas pois, dependendo do tipo de deslocamento efectuado, a operação pode ser considerada como de um tipo ou do outro, como se verá adiante.

A Figura 9.21 apresenta a estrutura da ULA em construção. Escolheu-se uma arquitectura em que as três unidades referidas trabalham em paralelo, entrando os operandos directamente em cada uma delas. De notar que esta opção é uma entre outras possibilidades de organização. Uma alternativa possível, com vantagens e desvantagens em relação à escolhida, seria colocar a unidade de deslocamento à saída do multiplexador, portanto em série com as unidades aritmética e lógica, o que permitiria executar micro-operações mais complexas.

A micro-operação realizada por qualquer destas unidades é especificada pela palavra de controlo. O número de bits de controlo que entra em cada unidade, p, q e r, depende do número de operações disponíveis em cada unidade.

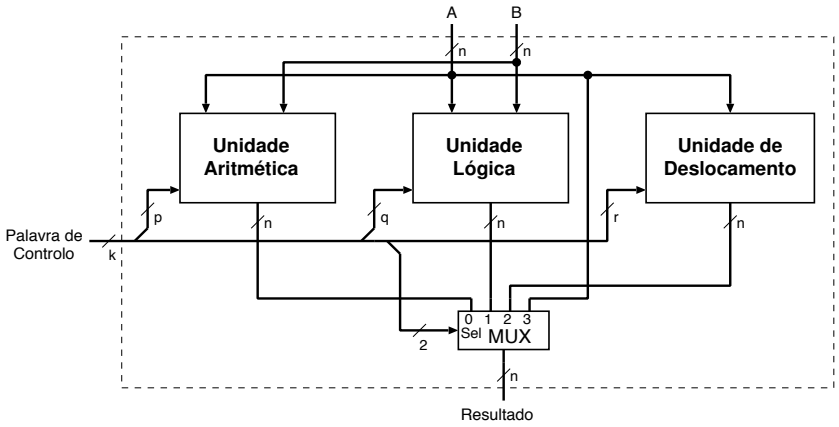


Figura 9.21: Estrutura da ULA.

Esta palavra de controlo controla também o multiplexador à saída, que escolhe de qual das unidades é que se pretende o resultado. De notar que só o resultado de uma vai ser usado na saída. Isto permite que os bits de controlo que entram em cada unidade possam ser partilhados, pois só é importante especificar os bits de controlo para a unidade que vai ter o seu resultado seleccionado à saída, sendo irrelevante qual a operação realizada pelas outras duas unidades. Notar também que, embora para não sobrecarregar a figura não esteja lá representado, existe também um multiplexador para seleccionar os bits de estado da unidade activa.

9.4.2 Bits de Estado

Sendo uma ULA uma unidade de processamento com micro-operações aritméticas e lógicas simples, os bits de estado passados para a unidade de controlo são também simples, e comuns a muitas destas micro-operações. Para a ULA em estudo, consideram-se 4 bits de estado, presentes na esmagadora maioria das ULAs.

Zero (*Z*): este bit fica a 1 quando o resultado da micro-operação foi 0. Este bit é actualizado em qualquer micro-operação da ULA. Em termos de realização, este bit pode ser gerado por uma porta lógica NOR entre todos os bits do resultado.

Transporte (*C*) (ou *carry*): este bit é usado pelas micro-operações aritméticas (como soma e subtracção) para indicar que há um bit de transporte para lá do bit mais significativo do resultado. É também usado nas micro-operações de deslocamento para guardar o bit que se perderia por sair para fora do resultado.

Sinal (*N*): para o bit de sinal é usado o bit mais significativo do resultado. Em notação de complemento para 2 ou sinal-magnitude, este bit indica quando o resultado deu negativo.

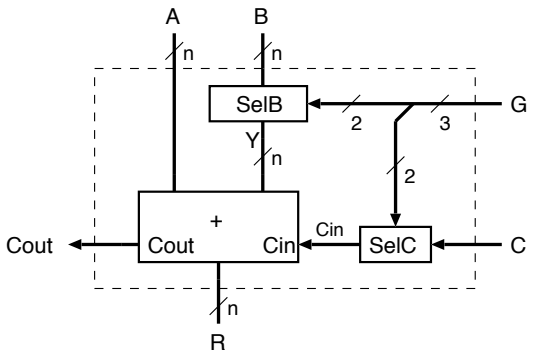


Figura 9.22: Esquema de uma unidade aritmética.

Excesso (*O*) (ou *overflow*): este bit só tem sentido para micro-operações aritméticas e fica a 1 quando o resultado tem uma magnitude que excede o valor máximo que possível de representar com o número de bits disponíveis para o resultado. Indica portanto que o valor de resposta está incorrecto. Este conceito de bit de excesso foi apresentado na Secção 6.2, onde se explicou como pode ser calculado pelo hardware.

Algumas ULAs podem apresentar um maior número de bits de estado, dependendo da aplicação em vista. Por exemplo, nalguns casos pode facilitar muito a existência de um bit de estado *paridade* que indique que o número de bits a 1 no resultado é par. A regra é normalmente a ULA fornecer informação que é útil à aplicação e que apenas é possível, ou substancialmente mais fácil, de obter directamente pelo hardware do que com (micro-)programação. Para a ULA em estudo, consideram-se apenas os 4 bits de estado descritos atrás.

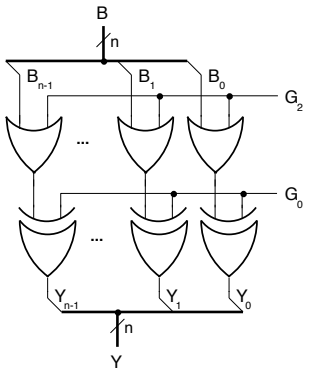
9.4.3 Unidade Aritmética

Quando se pensa em operações aritméticas básicas é natural pensar-se na adição, subtracção, multiplicação e divisão. De facto, a adição e a subtracção são quase que obrigatórias para as unidades aritméticas. A multiplicação, embora muito comum, não é incluída em todas devido à sua realização em hardware ser bastante mais complexa, conforme foi referido atrás. A implicação desta opção é que uma multiplicação terá que ser realizada por uma sequência de micro-operações mais elementares, como somas sucessivas, e portanto demorará bastante mais tempo a executar. A divisão também não é realizada directamente por muitas unidades aritméticas pois a sua realização é tão complexa como a multiplicação e é uma operação bastante menos utilizada. Assim, a penalização que advém da sua execução ser mais demorada não tem tanto peso no desempenho do sistema.

A Figura 9.22 apresenta um exemplo de uma unidade aritmética. Aqui, para simplificar, optou-se por não incluir as micro-operações de multiplicação e divisão. Assim, esta unidade aritmética é contruída em torno de um somador, que com a adição de blocos combinatórios simples que manipulam o operando

G_2G_0	Y_i
00	B_i
01	$\overline{B_i}$
10	1
11	0

(a)

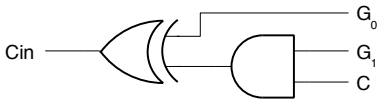


(b)

Figura 9.23: Bloco combinatório `SelB` (a) Descrição funcional. (b) Circuito lógico.

G_1G_0	C_{in}
00	0
01	1
10	C
11	$\overline{C}$

(a)



(b)

Figura 9.24: Bloco combinatório `SelC` (a) Descrição funcional. (b) Circuito lógico.

B e a entrada do sinal de transporte, permite utilizar o somador para executar um conjunto interessante de micro-operações.

Como foi discutido na Secção 6.2.2, um somador pode ser facilmente convertido num subtractor por aplicação do complemento para 2 de um dos operandos. Este complemento para 2 pode ser obtido por complementação lógica bit a bit desse operando e por colocação do bit de transporte C_{in} à entrada a 1. Os blocos combinatórios `SelB` e `SelC`, apresentados nas Figuras 9.23 e 9.24 respectivamente, foram definidos de forma a que, com os sinais de controlo $G_1 = 0$ e $G_2 = 0$, o sinal G_0 seleccione se a operação é uma soma ou uma subtracção. Como já referido, uma porta EXOR pode ser vista como um inversor controlado. Se o sinal G_0 estiver a 0, então $Y = B$. Com C_{in} também a 0, esta unidade realiza uma soma normal. Se o sinal G_0 estiver a 1, então cada bit de Y é o complemento do bit correspondente de B. Como $C_{\text{in}} = 1$, então tem-se o complemento para 2 de B e esta unidade realiza a subtracção $A - B$.

Por vezes, pretende-se fazer uma soma entre operandos com um número de bits b superior aos permitidos pela unidade aritmética, $b > n$. Uma forma simples de conseguir este objectivo é começar por somar os n bits menos significativos dos operandos e guardar o bit de transporte. Depois somam-se os n bits seguintes, tendo agora em conta o bit de transporte anterior. Este procedimento pode-se repetir indefinidamente, permitindo a soma de operandos com

$G_2G_1G_0$	Y_i	C_{in}	Micro-Operação	
000	B_i	0	$R \leftarrow A + B$	soma
001	$\overline{B_i}$	1	$R \leftarrow A - B$	subtracção
010	B_i	C	$R \leftarrow A + B + C$	soma com bit de transporte
011	$\overline{B_i}$	$\overline{C}$	$R \leftarrow A - B - C$	subtracção com bit de transporte
100	1	0	$R \leftarrow A - 1$	decremento
101	0	1	$R \leftarrow A + 1$	incremento
110	1	C	$R \leftarrow A - \overline{C}$	decremento, se $C = 0$
111	0	$\overline{C}$	$R \leftarrow A + \overline{C}$	incremento, se $C = 0$

Tabela 9.1: Conjunto de micro-operações da unidade aritmética.

um número arbitrário de bits. O mesmo raciocínio funciona para a subtracção. De maneira a acomodar esta situação, a unidade aritmética tem que poder aceitar um bit de transporte. Na unidade em questão, isto é conseguido com o sinal de controlo G_1 a 1, em que se têm as operações $A + B + C$ ou $A - B - C$ dependendo do valor de G_0.

Por fim, outras operações muito comuns são simples incrementos e decrementos de um operando. Para incrementar o operando A, basta colocar a entrada correspondente ao operando B a 0 e o bit de transporte a 1, o que é conseguido com as entradas de controlo G_2=1, G_1=0 e G_0=1. De forma semelhante, para decrementar o operando A faz-se a sua soma com B=-1 e C_{in}=0. Como o complemento para 2 de 1 é um valor com todos os bits a 1, a diferença para esta operação é o sinal de controlo G_0=0.

A Tabela 9.1 resume as micro-operações possíveis de realizar pela unidade aritmética apresentada na Figura 9.22, para cada combinação das entradas de controlo $G_2G_1G_0$. Note-se que as duas últimas micro-operações não foram projectadas, mas surjem como efeitos secundários das restantes operações. Embora à primeira vista não pareçam tão úteis como as seis que foram projectadas, estão disponíveis e podem ser usadas se necessário.

9.4.4 Unidade Lógica

A diferença fundamental entre as operações lógicas e as operações aritméticas é que as primeiras são operações binárias e as segundas operam sobre valores numéricos. Assim, para as operações lógicas os bits são tratados de forma independente enquanto que para as aritméticas são os bits no seu conjunto que têm significado, de acordo com a notação usada em cada caso para a representação de valores numéricos.

As micro-operações fornecidas pelas unidades lógicas aplicam-se individualmente a cada bit dos operandos de entradas. Por exemplo, a micro-operação $R \leftarrow A \wedge B$ significa um AND entre cada bit dos operandos A e B: $R \leftarrow A_{n-1} \wedge B_{n-1} | \cdots | A_1 \wedge B_1 | A_0 \wedge B_0$.

Qualquer operação lógica pode ser incluída numa unidade lógica. Para a ULA em projecto, consideram-se as micro-operações NOT, AND, OR e XOR. A Figura 9.25 ilustra como estas micro-operações são realizadas para um bit i. Naturalmente, a unidade lógica necessitará de tantos blocos iguais ao da Figu-

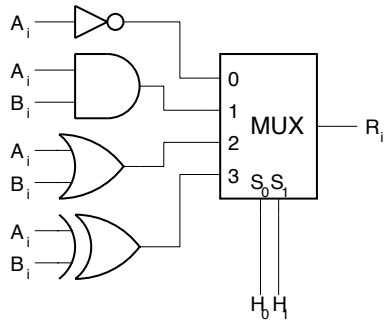


Figura 9.25: Esquema da unidade lógica.

H_1H_0	Micro-Operação	
00	$R \leftarrow \overline{A}$	complemento
01	$R \leftarrow A \wedge B$	conjunção
10	$R \leftarrow A \vee B$	disjunção
11	$R \leftarrow A \oplus B$	ou exclusivo

Tabela 9.2: Conjunto de micro-operações da unidade lógica.

ra 9.25 quanto o número de bits dos operandos. Os sinais de controlo H_0 e H_1 escolhem qual das micro-operações tem o seu resultado seleccionado à saída. A Tabela 9.2 indica qual a correspondência entre a combinação destes sinais e a micro-operação seleccionada.

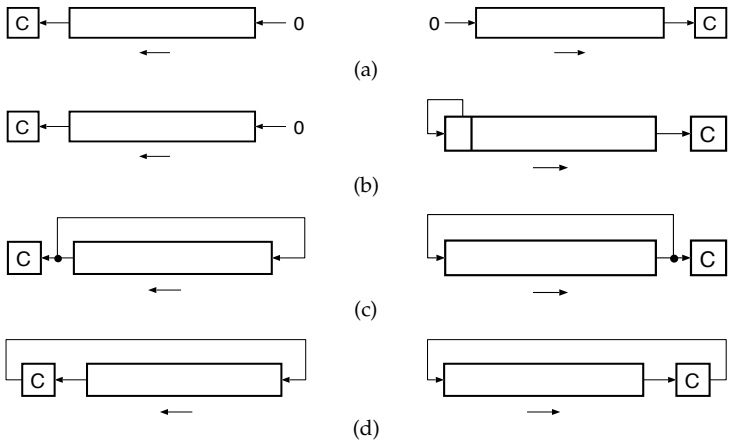


Figura 9.26: Tipos de deslocamento: (a) Deslocamento lógico; (b) Deslocamento aritmético; (c) Rotação; (d) Rotação com transporte.

9.4.5 Unidade de Deslocamento

A unidade de deslocamento apenas admite um operando de entrada e permite deslocar todos os bits deste operando uma posição à esquerda ou à direita. Existem vários tipos de deslocamento possíveis, indicados na Figura 9.26. Uma primeira observação é que em qualquer das formas de deslocamento o bit de estado transporte (C) recebe o bit que se perde com o deslocamento, o bit mais significativo do operando no caso de deslocamentos à esquerda, ou o bit menos significativo nos deslocamentos à direita.

Deslocamento lógico (Figura 9.26(a)): é o deslocamento mais simples em que cada bit passa para a posição seguinte, consoante seja deslocamento à esquerda ou à direita. O bit que sai do operando é copiado para o bit de estado transporte (C) e o bit que entra é sempre 0. Os bits de estado transporte (C), zero (Z) e sinal (N) são actualizados. Se se considerar o operando de entrada como um número sem sinal, um deslocamento de uma posição à esquerda ou à direita é o mesmo do que uma multiplicação ou divisão por 2, respectivamente.

Deslocamento aritmético (Figura 9.26(b)): este tipo de deslocamento tem em vista a utilização de um operando em formato de complemento para 2. O objectivo é manter a regra da multiplicação e divisão por 2, agora em números com sinal. No deslocamento à esquerda, o movimento dos bits é exactamente o mesmo do deslocamento lógico. A diferença está em que se o bit de sinal mudar de valor, isso significa que o resultado excedeu a capacidade para o número de bits do operando (uma multiplicação por 2 não pode trocar o sinal) e portanto o bit de estado excesso (O) ficará a 1. No deslocamento à direita, a diferença está também na forma como o bit mais significativo é tratado. Como o sinal numa divisão por 2 se tem que manter, em vez de entrar sempre 0 no bit de maior peso, este bit mantém o valor anterior. Sendo uma operação aritmética, todos os bits de estado são actualizados.

Rotação (Figura 9.26(c)): nesta micro-operação o movimento dos bits é o mesmo do deslocamento lógico, mas o bit que entra não é 0 e sim igual ao bit que sai, fechando-se assim o círculo. Numa rotação à esquerda, o bit de maior peso entra no bit de menor peso. Numa rotação à direita, o bit de menor peso entra no bit de maior peso. Os bits de estado transporte (C), zero (Z) e sinal (N) são actualizados.

Rotação com transporte (Figura 9.26(d)): este tipo de rotação é igual ao anterior, com a diferença que o bit de estado transporte é incluído no círculo de rotação. O bit que entra é, portanto, o bit que estava anteriormente no bit de estado transporte. Também aqui os bits de estado transporte (C), zero (Z) e sinal (N) são actualizados.

Um circuito que realiza estas micro-operações de deslocamento está representado na Figura 9.27. Existem 8 micro-operações possíveis, portanto são necessários 3 sinais de controlo, J_0, J_1 e J_2. Neste circuito, J_0 indica se o deslocamento é à direita (0) ou à esquerda (1). Para os bits intermédios do operando, este é o único sinal de controlo relevante, pois simplesmente se vai buscar o bit de peso imediatamente maior ou menor. Por outras palavras, o bit de saída i

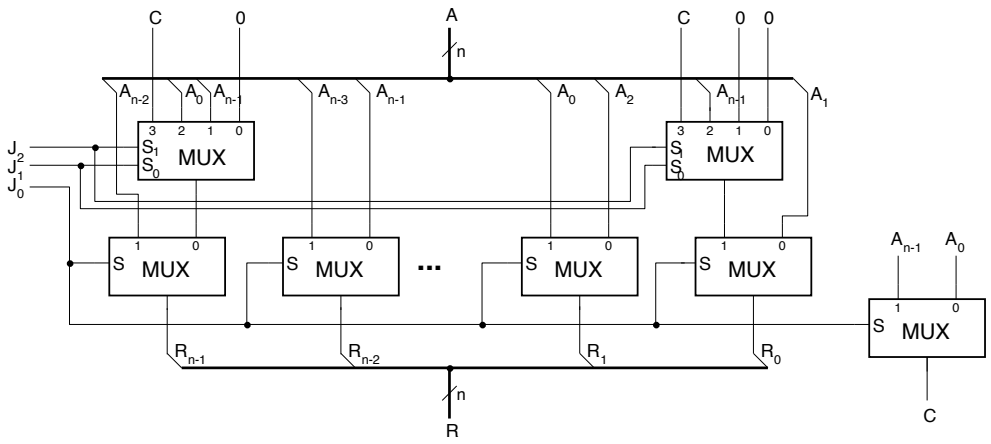


Figura 9.27: Esquema da unidade de deslocamento.

$J_2J_1J_0$	Micro-Operação	
000	$R \leftarrow \text{SHR } A$	deslocamento lógico à direita
001	$R \leftarrow \text{SHL } A$	deslocamento lógico à esquerda
010	$R \leftarrow \text{SHRA } A$	deslocamento aritmético à direita
011	$R \leftarrow \text{SHLA } A$	deslocamento aritmético à esquerda
100	$R \leftarrow \text{ROR } A$	rotação à direita
101	$R \leftarrow \text{ROL } A$	rotação à esquerda
110	$R \leftarrow \text{RORC } A$	rotação à direita com transporte
111	$R \leftarrow \text{ROLC } A$	rotação à esquerda com transporte

Tabela 9.3: Micro-operações possíveis na unidade de deslocamento.

fica igual ao bit de entrada $i+1$ ou $i-1$ consoante J_0 é 0 ou 1. Este é também o único sinal de controlo para a geração do bit de estado transporte, que se carrega com o bit mais ou menos significativo dependendo se o deslocamento é à esquerda ou à direita, respectivamente.

Os sinais J_1 e J_2 servem para definir qual o tipo de deslocamento a executar, o que se consegue pelo controlo do bit que entra, o bit mais significativo nos deslocamentos à direita e o bit menos significativo nos deslocamentos à esquerda. No caso de um deslocamento à direita, para um:

- deslocamento lógico: entra sempre um 0.
- deslocamento aritmético: o bit mantém-se, pois é um bit de sinal.
- rotação: entra o bit de menor peso do operando.
- rotação com transporte: entra o bit de estado transporte.

De forma semelhante para um deslocamento à esquerda:

- deslocamento lógico: entra sempre um 0.
- deslocamento aritmético: entra sempre um 0.
- rotação: entra o bit de maior peso do operando.
- rotação com transporte: entra o bit de estado transporte.

$S_4S_3S_2S_1S_0$	Micro-Operação	
00000	$R \leftarrow A + B$	soma
00001	$R \leftarrow A - B$	subtracção
00010	$R \leftarrow A + B + C$	soma com bit transporte
00011	$R \leftarrow A - B - C$	subtracção com bit transporte
00100	$R \leftarrow A - 1$	decremento
00101	$R \leftarrow A + 1$	incremento
00110	$R \leftarrow A - \overline{C}$	decremento, se $C = 0$
00111	$R \leftarrow A + \overline{C}$	incremento, se $C = 0$
01-00	$R \leftarrow \overline{A}$	complemento
01-01	$R \leftarrow A \wedge B$	conjunção
01-10	$R \leftarrow A \vee B$	disjunção
01-11	$R \leftarrow A \oplus B$	ou exclusivo
10000	$R \leftarrow \text{SHR } A$	deslocamento lógico à direita
10001	$R \leftarrow \text{SHL } A$	deslocamento lógico à esquerda
10010	$R \leftarrow \text{SHRA } A$	deslocamento aritmético à direita
10011	$R \leftarrow \text{SHLA } A$	deslocamento aritmético à esquerda
10100	$R \leftarrow \text{ROR } A$	rotação à direita
10101	$R \leftarrow \text{ROL } A$	rotação à esquerda
10110	$R \leftarrow \text{RORC } A$	rotação à direita com transporte
10111	$R \leftarrow \text{ROLC } A$	rotação à esquerda com transporte
11- - -	$R \leftarrow A$	transferência

Tabela 9.4: Tabela das micro-operações da ULA.

A micro-operação realizada pela unidade de deslocamento para cada combinação dos sinais de controlo encontra-se discriminada na Tabela 9.3.

9.4.6 Tabela de Controlo da ULA

Tendo-se estabelecido o funcionamento de cada uma das suas unidades internas, pode-se definir o funcionamento global da ULA representada na Figura 9.21. Assim, a unidade aritmética disponibiliza 8 micro-operações seleccionadas pelos sinais $G_2G_1G_0$, a unidade lógica disponibiliza 4 micro-operações seleccionadas pelos sinais H_1 e H_0, e a unidade de deslocamento 8 micro-operações seleccionadas pelos sinais J_2, J_1 e J_0. Observe-se que o multiplexador à saída da ULA escolhe o resultado de uma destas unidades, e portanto para cada micro-operação apenas se tem que garantir que a unidade responsável pela sua execução tem os bits de controlo certos. Ou seja, para cada micro-operação de facto só existe a preocupação de controlar uma destas três unidades. Isto significa que os sinais de controlo destas unidades podem ser partilhados: $G_0 = H_0 = J_0$, $G_1 = H_1 = J_1$ e $G_2 = J_2$. A nível da ULA, a referência a estes sinais será S_0, S_1 e S_2, respectivamente.

Para além destes 3 sinais de controlo, são necessários mais 2 sinais, S_3 e S_4,

para controlar os sinais de selecção do multiplexador. Arbitrariamente define-se que a combinação 00 selecciona a unidade aritmética, a combinação 01 a unidade lógica e 10 a unidade de deslocamento. A combinação 11 fica disponível, tendo-se optado por fazer seleccionar a entrada A de forma a se fornecer uma micro-operação que se limita a copiar um operando directamente para a saída. A Tabela 9.4 apresenta as micro-operações disponibilizadas por esta ULA e qual a palavra de controlo que as define.

A forma como os bits de estado são actualizados depende de qual das unidades da ULA é usada. Os bits zero (Z) e sinal (N) são calculados de igual forma para todas as unidades, de acordo com o definido na Secção 9.4.2. O bit zero é um NOR de todos os bits do resultado. Apesar de não ter muito significado para as micro-operações lógicas, o bit de sinal é sempre calculado devido a ser simplesmente igual ao bit mais significativo do resultado.

Já o bit de transporte (C) é calculado para as unidades aritmética e de deslocamento, e de forma diferente para estas. No caso da unidade aritmética, este bit é o bit de transporte à saída do somador, o sinal C_{out} da Figura 9.22. A forma de cálculo deste bit na unidade de deslocamento foi definida na Secção 9.4.5. Tipicamente é o bit que sai do operando devido à operação de deslocamento.

Por último, o bit excesso (O) só tem sentido nas micro-operações da unidade aritmética e nas micro-operações de deslocamento aritmético da unidade de deslocamento. Nesta última, foi também explicado na Secção 9.4.5 que o bit excesso fica a 1 quando uma operação de deslocamento aritmético provoca uma mudança do bit de sinal. Para a unidade aritmética, este bit é calculado através de um EXOR entre os dois bits de transporte mais significativos do somador da Figura 9.22, como explicado no Capítulo 6.

Assume-se que os bits de transporte e excesso não são definidos quando a micro-operação executada utiliza uma unidade em que eles não são calculados. Naturalmente, está aqui incluída a micro-operação de transferência, $R \leftarrow A$, pois não utiliza nenhuma destas unidades. Na prática, uma possibilidade, tão boa como qualquer outra, é assumir que estes ficam a zero.

O hardware necessário à geração destes bits de estado e à sua selecção à saída da ALU não está representado explicitamente nas figuras desta secção apenas para as não sobrecarregar.

9.4.7 Exemplo Revisitado: Máximo Divisor Comum

Considerando novamente o exemplo da Secção 9.3.2, ilustra-se agora como seria a realização de um sistema que calcula o Máximo Divisor Comum entre dois operandos utilizando como unidade de processamento a ULA definida nesta secção. A descrição deste sistema utilizando apenas as micro-operações que a ULA fornece encontra-se na Figura 9.28. Aqui assumiu-se que a ULA tem associado um banco de registos com pelo menos 3 registos, em que `R1` e `R2` têm inicialmente os dados de entrada, respectivamente X e Y, e `R3` serve como registo temporário. O resultado final fica em `R2`.

Pode-se observar que esta descrição é um pouco mais complexa do que na versão original. Como mencionado atrás, ao utilizar-se uma unidade de processamento genérica como esta ULA, em oposição a uma unidade de processamento especificamente desenhada para um dado problema, são necessárias mais micro-operações para atingir um dado objectivo. Neste caso particular, na versão original era possível no estado $T1$ fazer uma comparação e efectuar

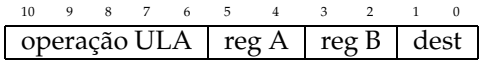
1.	$T0$:	$fim \leftarrow 1$
2.	$T0.inicio$:	$T0 \leftarrow 0, T1 \leftarrow 1$
3.	$T1$:	R1$\leftarrow$ R1 $-$ R2, $fim \leftarrow 0$
4.	$T1.Z$:	$T1 \leftarrow 0, T0 \leftarrow 1$
5.	$T1.N$:	$T1 \leftarrow 0, T2 \leftarrow 1$
6.	$T2$:	R3$\leftarrow$ R1 $+$ R2, $T2 \leftarrow 0, T3 \leftarrow 1, fim \leftarrow 0$
7.	$T3$:	R1$\leftarrow$ R2, $T3 \leftarrow 0, T4 \leftarrow 1, fim \leftarrow 0$
8.	$T4$:	R2$\leftarrow$ R3, $T4 \leftarrow 0, T1 \leftarrow 1, fim \leftarrow 0$

Figura 9.28: Descrição a nível de transferência entre registos do algoritmo para o cálculo do máximo divisor comum.

ainda no mesmo ciclo uma subtracção ou uma troca de registos, dependendo do resultado desta comparação. Com a ULA, a comparação tem que ser feita primeiro e só no ciclo seguinte se pode fazer uma nova operação. No código da Figura 9.28 foi feita uma optimização trivial pois como a comparação tem que ser realizada através de uma subtracção, caso o resultado ainda seja positivo a subtracção já está feita. Repare-se que caso o resultado seja negativo, há que somar a R3 o valor de R2 para recuperar o valor inicial de R1. A outra diferença está em que na ULA não é possível fazer uma troca de valores entre dois registos. Assim, são necessários três ciclos de relógio e um registo temporário para efectuar esta operação.

A sequência de operações a realizar está descrita no fluxograma da Figura 9.29. Ainda é um fluxograma simples, pelo que qualquer das técnicas de projecto de máquinas de estados pode ser utilizada para a unidade de controlo. Opta-se aqui por recorrer a um controlo micro-programado (ver Secção 8.3.3).

A palavra de controlo terá que seleccionar a operação da ULA, os dois registos com os operandos e o registo destino. Para a ULA são necessários 5 bits para a escolha da micro-operação, conforme indicado na Figura 9.21. Assumindo os 3 registos do banco de registos (R1, R2 e R3), bastam 2 bits para seleccionar cada um dos registos de cada operando e do registo para guardar o resultado. No total, a palavra de controlo será composta por 11 bits:



O número de estados no fluxograma da Figura 9.29 é cinco, o que corresponde ao número de posições na ROM de controlo, pelo que o número de bits para os endereços de controlo será três. Pode-se verificar neste fluxograma que o estado $T1$ pode ter 3 estados seguintes diferentes. Portanto, na micro-instrução terão de existir pelo menos dois endereços seguintes (assumindo que o terceiro pode ser obtido por incremento do CAR). Outra consequência desta observação é que para decidir entre 3 endereços são necessários 2 bits para a escolha do endereço seguinte a carregar no CAR. Por último, as condições de salto são: o sinal de controlo *inicio* no estado $T0$; os bits de estado Z e N em $T1$; o estado seguinte em $T2$ e $T3$ (sem salto); salto incondicional em $T4$. Para cobrir estas quatro hipóteses são necessários 2 bits de controlo para seleccionar a condição de salto. Finalmente, terá que existir um bit para a saída de controlo fim.

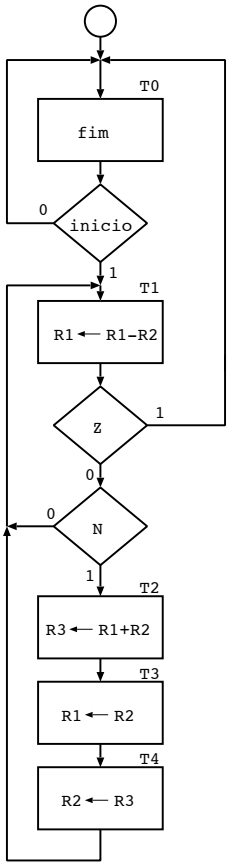


Figura 9.29: Fluxograma da unidade de controlo para o máximo divisor comum.

Adicionando à palavra de controlo os dois endereços seguintes para o `CAR` ($ES1$ e $ES0$), os dois bits para a selecção da condição de salto (SEL) e o bit da saída de controlo, tem-se o formato completo da micro-instrução para a unidade de controlo:

19–17	16–15	14–13	12	11–10	9–7	6–5	4–3	2–0
ES0	ES1	SEL	fim	operação ULA	reg A	reg B	dest

A Figura 9.30 apresenta um controlador micro-programável ajustado às condições deste problema. À saída da ROM de controlo tem-se a micro-instrução com a palavra de controlo que é enviada directamente para a unidade de processamento, com os campos $ES0$ e $ES1$ que guardam 2 possíveis endereços seguintes para o `CAR` e com o campo SEL que indica qual a condição de salto. Para este campo foi escolhido arbitrariamente que:

00 - corresponde a um salto incondicional. A saída do multiplexador `MUXS` tem sempre o valor 00, combinação que selecciona o campo $ES0$ da micro-instrução.

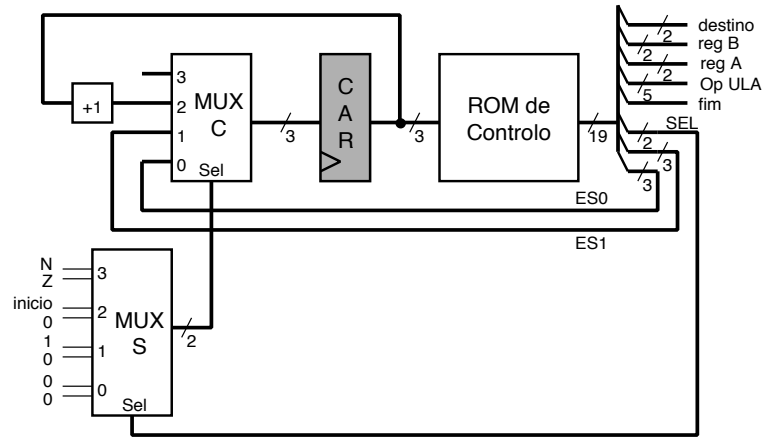


Figura 9.30: Unidade de controlo micro-programada para o máximo divisor comum.

01 - corresponde a não haver salto. O `CAR` é simplesmente incrementado, porque o `MUXS` apresenta sempre o valor 10 à saída, o que leva a que a entrada `CAR`+1 seja seleccionada no `MUXC`.

10 - corresponde a testar a entrada de controlo *inicio*. Quando *inicio*=0 é seleccionado no `MUXC` a entrada $ES0$ e portanto dá-se o salto. Se *inicio*=1, o `CAR` é incrementado.

11 - corresponde a testar simultaneamente os bits de estado N e Z. Se:

$NZ = 00$, o salto é feito para o endereço no campo $ES0$.

$NZ = 01$, o salto é feito para o endereço no campo $ES1$.

$NZ = 10$, não há salto e o `CAR` é simplesmente incrementado.

$NZ = 11$, combinação impossível.

Falta agora definir o micro-programa que realiza o fluxograma da Figura 9.29 (as posições sublinhadas indicam valores que são de facto indiferenças):

Posição ROM	ES0	ES1	SEL	fim	op ULA	reg A	reg B	reg dest
0:	00	<u>00</u>	10	1	<u>00000</u>	<u>00</u>	<u>00</u>	00
1:	01	00	11	0	00001	01	10	01
2:	<u>00</u>	<u>00</u>	01	0	00000	01	10	11
3:	<u>00</u>	<u>00</u>	01	0	11000	10	<u>00</u>	01
4:	01	<u>00</u>	00	0	11000	11	<u>00</u>	10

Neste micro-programa existe uma correspondência entre a posição de endereço i da ROM de controlo e o índice do estado Ti do fluxograma. Assim:

Posição 0: fim←1, $\overline{inicio}$: `CAR`← 0, *inicio*: `CAR`←`CAR`+1

Neste estado a unidade de controlo está constantemente a carregar o `CAR` com 0 até que a entrada *inicio* venha a 1, altura em que se deixa incrementar o `CAR`. Para isso, o campo SEL tem que ter o valor 10 de forma

a permitir que o sinal *inicio* passe no multiplexador MUXS para servir de selecção no MUXC. Enquanto *inicio* for 0, o multiplexador MUXC selecciona a entrada 0, campo ES0 da micro-instrução que contém o endereço 0. Se *inicio* vier a 1, então é a entrada 3 do MUXC que é seleccionada. Nesta espera activa, a saída de controlo fim é mantida a 1 e a unidade de processamento não faz nenhuma operação, o que se consegue especificando como registo destino um registo não existente, neste caso utilizou-se o índice 0.

Posição 1: R1←R1−R2, Z: CAR←0, $\overline{Z}.\overline{N}$: CAR←1, $\overline{Z}.N$: CAR← CAR+1

Aqui a ULA faz a subtracção entre R1 e R2, guardando-se o resultado em R1. Caso o resultado seja 0, o algoritmo terminou e portanto carrega-se no CAR através do campo $ES1$ o endereço 0 para voltar ao início (no multiplexador MUXS é seleccionada a entrada 3 e no MUXC a entrada 1). Enquanto o resultado for positivo (bit de estado $N = 0$), a unidade de controlo mantém-se no endereço 1 por carregamento do campo $ES0$ no CAR. Se o resultado for negativo, em vez da subtracção deve-se trocar os valores de R1 e R2, o que é feito a partir do endereço 2, ou seja, por incremento do CAR. O teste que determina qual destes três endereços seguintes é usado é conseguido por SEL=11.

Posição 2: R3←R1+R2, CAR←CAR+1

Neste estado, é colocado em R3 o valor que R1 tinha antes da subtracção, somando-lhe R2. O CAR é simplesmente incrementado, o que se consegue fixando a entrada 1 do multiplexador MUXS.

Posição 3: R1←R2, CAR←CAR+1

Aqui há uma simples transferência entre registos, o valor de R2 é copiado para R1. Novamente incrementa-se o CAR.

Posição 4: R2←R3, CAR←1

Finalmente neste estado, efectua-se outra simples transferência entre registos, copiando-se o valor de R3 para R2. Executa-se um salto incondicional para o estado no endereço 1, o que se consegue seleccionando a entrada 0 do multiplexador MUXS, obrigando à selecção do campo ES0 da micro-instrução.

Capítulo 10

Arquitectura de um Computador

Embora um sistema digital para qualquer função específica possa ser concebido de raiz interligando registos e blocos funcionais, usando as técnicas estudadas no Capítulo 9, por razões de flexibilidade é preferível dispor de um sistema adaptável e facilmente reprogramável. Um computador é exactamente isso, um sistema digital cujo funcionamento é especificado por um programa guardado em memória. Este capítulo descreve de uma maneira resumida a evolução dos computadores e a sua organização interna.

Um computador é um sistema digital programável através de uma sequência de instruções guardadas em memória. Através desta sequência de instruções, o computador é instruído para executar operações elementares de manipulação de dados. Um elemento crucial num computador é a sua unidade central de processamento, ou processador.

O processador, em si mesmo, é um sistema digital constituído por um circuito de dados e um circuito de controlo, tal como os sistemas estudados no capítulo anterior. A sua flexibilidade resulta do facto de estes circuitos serem projectados por forma a poderem executar todas as operações básicas de manipulação de dados que são necessárias para a execução de tarefas de processamento de dados, por mais complexas que estas sejam.

10.1 Perspectiva Histórica

A ideia de um sistema de processamento de informação que seja reconfigurável através de um programa guardado em alguma forma de memória é muito antiga, remontando aos princípios do século XIX. Charles Babbage projectou o primeiro computador mecânico capaz de desempenhar sequências de operações automaticamente. Este computador, conhecido como o *difference engine*, usava uma tecnologia totalmente baseada em elementos mecânicos. O sistema podia ser programado para construir tabelas matemáticas, de acordo com instruções especificadas pelo programador. A complexidade dos sistemas mecânicos necessários à realização de funções matemáticas fez com que computadores mais poderosos não viessem a ser fabricados até à existência de tecnologias

baseadas em circuitos electrónicos, embora tenham sido feitos planos para um computador mecânico ainda mais poderoso, denominado *analytical engine*.

Embora tenham existido diversas tentativas de construção de computadores electrónicos digitais, a primeira máquina a funcionar efectivamente foi o ENIAC, que ficou operacional em 1946 e podia ser programado para calcular tabelas matemáticas relacionadas com aplicações militares. Este computador usava uma tecnologia baseada em válvulas electrónicas e era programado através de um conjunto de cabos que, conforme estivessem ligados ou desligados, especificavam o conteúdo da memória do programa. Os seus resultados eram escritos em cartões perfurados ou numa máquina de escrever eléctrica.

O ENIAC guardava o programa e os dados numa memória, que, para a época, era de alta velocidade. Este conceito de usar a memória para guardar tanto o programa como os dados é atribuída a John von Neumann, que trabalhou no projecto. Por essa razão, chama-se a este tipo de arquitectura, em que uma só memória guarda tanto os dados como os programas, uma *arquitectura de von Neumann*. Esta arquitectura veio a impôr-se à alternativa de ter duas memórias separadas para os dados e para o código, que é conhecida como *arquitectura de Harvard*.

Um número de outros computadores, cada vez mais avançados, mas ainda pertencentes ao que se convencionou designar por *primeira geração*, vieram a ser projectados e construídos, mas a utilização de válvulas de vácuo impunha limites sérios à fiabilidade e ao tamanho mínimo com que podiam ser construídos. Estes computadores ocupavam tipicamente uma sala de grandes dimensões e podiam funcionar continuamente apenas durante poucas horas, até que um dos sub-sistemas deixasse de funcionar por falha de uma válvula.

A utilização de transístores como tecnologia de base veio permitir a evolução para a segunda geração de computadores, que já usavam tecnologias de discos magnéticos para guardar a informação de forma não volátil e tinham memórias baseadas em ferrites.

A terceira geração de computadores apareceu com o advento dos circuitos integrados, em 1961, permitindo uma grande redução no tamanho dos computadores e o consequente aumento do número de portas lógicas que era possível empacotar num dado volume. Um dos computadores que mais sucesso alcançou foi o IBM/360. Com este computador, a IBM introduziu o conceito de separar o conjunto de instruções da implementação física de um computador, criando assim a primeira de uma série de famílias que executavam o mesmo conjunto de instruções em processadores com diferentes organizações internas. A introdução deste conceito permitiu uma muito maior reutilização do esforço de programação de sistemas, uma vez que um programa codificado para um computador de uma dada família passou a poder ser executado por outros computadores da mesma família.

O desenvolvimento da tecnologia e a criação do circuito integrado com grande escala de integração (VLSI[1]) veio permitir a criação de processadores inteiramente contidos num só circuito integrado, o que fez baixar o preço de um computador por forma a permitir o seu uso como um computador pessoal. Embora o primeiro processador disponível num único circuito integrado tenha sido o 4004, da Intel, e o primeiro computador pessoal tenha sido baseado no processador 8080, do mesmo fabricante, o passo mais importante para

[1] Very Large Scale Integration.

a divulgação deste tipo de computadores terá sido a definição, pela IBM, de um modelo padrão para computadores pessoais, ou PCs[2], baseados no processador 8086. O sucesso deste tipo de computadores, baseados em diversos processadores desta família da Intel (80286, 80386, 80486, Pentium, Pentium II, Pentium III e Pentium IV), é conhecido.

10.2 Tipos de Computadores

Apesar da dominância, em termos numéricos, dos computadores pessoais baseados em processadores da Intel, continuam a existir alternativas baseadas noutros tipos de processadores. Entre estas contam-se os computadores da Apple, baseados inicialmente nos processadores da série 680X0 da Motorola e, mais recentemente, na família de processadores PowerPC, também da Motorola. Com os computadores pessoais coexistem outros tipos de computadores, tais como servidores baseados em processadores MIPS e Compaq (antes Digital), embora, neste momento, a diferença entre servidores e computadores pessoais tenha mais a ver com o sistema operativo que executam do que com as capacidades intrínsecas do processador.

A criação dos circuitos integrados veio também possibilitar a utilização de processadores para controlar sistemas electrónicos autónomos. Estes processadores, geralmente conhecidos como micro-controladores, são tipicamente mais simples, menos dispendiosos e mais flexíveis no que respeita aos sistemas de entrada/saída, o que permite o seu uso em aplicações tais como o controlo de electrodomésticos, onde o baixo preço é um importante factor. Processadores como o 8051, também da Intel, são de facto computadores feitos num único circuito integrado, uma vez que incluem num único circuito o processador, a memória e os meios de comunicação com o exterior.

Um outro tipo de processadores dedicados são os processadores de sinal, que são projectados por forma a serem extremamente eficientes em aplicações de processamento de sinal, e são comummente usados em sistemas para telecomunicações tais como telefones e televisões.

10.3 Organização Interna de um Computador

Como foi referido acima, um computador é um sistema digital programável através de um sequência de instruções guardada em memória. Estas instruções especificam qual a sequência de operações de manipulação de dados que deve ser executada.

Cada instrução especifica, de forma única, qual a operação que deve ser efectuada e quais os operandos aos quais a mesma deve ser aplicada. Por exemplo, uma dada instrução poderá especificar que o conteúdo da posição `10FAh` da memória deve ser somado ao conteúdo do registo `R3`, devendo o resultado ser guardado na mesma posição de memória, `10FAh`.

A unidade que processa a sequência de instruções é a *unidade central de processamento*, UCP (em inglês, *Central Processing Unit* ou *CPU*). As unidades centrais de processamento têm capacidade para executar um conjunto de instruções que, embora variando fortemente de computador para computador,

[2]Personal Computer.

tem um certo número de características comuns. A UCP é constituída por circuitos combinatórios que permitem efectuar operações lógicas e aritméticas, e por registos que permitem guardar os dados e os resultados dessas operações.

Nos computadores modernos, a sequência de instruções a executar é guardada em memória. Segundo o modelo de Von Neumann, esta memória serve também para guardar os dados, quer os de entrada no programa quer os que resultam da execução do mesmo.

As instruções estão guardadas sequencialmente em memória, e são, de uma forma geral, executadas pelo ordem em que se encontram. No entanto, existem também instruções que alteram a ordem de execução. Por exemplo, uma dada instrução poderá especificar que a próxima instrução a ser executada é a instrução guardada na posição de memória `AAF0h`.

A memória encontra-se ligada à unidade central de processamento através de dois barramentos, tal como está descrito na Figura 10.1.

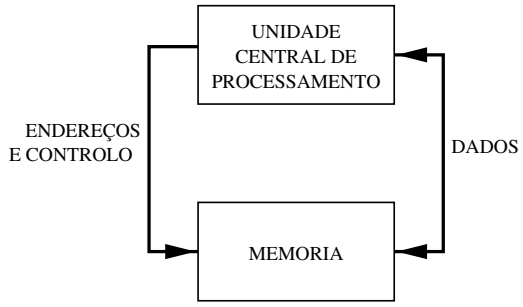


Figura 10.1: Interligação da unidade central de processamento com a memória.

Esquematicamente, a execução de uma instrução guardada em memória passa pelas seguintes fases:

- A UCP envia para a memória o endereço da próxima instrução a executar e recebe uma palavra de memória que guarda a codificação da instrução a executar.
- A UCP descodifica a instrução, identificando quais os operandos e qual o tipo de operação.
- A UCP faz um ou mais acessos à memória para carregar os operandos especificados pela instrução para registos internos.
- Na posse de todos os operandos, a unidade central de processamento executa a operação especificada na instrução.
- Após obter o resultado, a UCP escreve o mesmo em memória, caso seja necessário, ou num registo interno.

Esta descrição deve ser vista como uma descrição do princípio geral do funcionamento, podendo não se aplicar directamente a processadores modernos, que são extremamente complexos e que usam um número de diferentes técnicas para acelerar a velocidade de processamento.

Existem, no entanto, algumas alternativas a esta organização simples que é importante referir nesta fase e que são tipicamente utilizadas em processadores comerciais.

Em primeiro lugar, não é geralmente verdade que cada posição de memória seja suficiente para guardar a totalidade de uma instrução ou de cada operando. Assim, é por vezes necessário fazer vários acessos a memória para carregar uma instrução ou um operando.

Em segundo lugar, existem numerosos casos em que o fluxo normal de funcionamento descrito acima também não é respeitado. Existem muitos processadores que, por razões de eficiência, tentam executar mais que uma operação em paralelo. Nestes casos, descritos em mais detalhe no Capítulo 15, a unidade central de processamento pode intercalar as diversas fases de execução de diferentes instruções, podendo, por exemplo, carregar a próxima instrução a executar antes de escrever o resultado da anterior.

Uma outra alteração significativa à organização descrita acima é a não utilização de uma única memória para guardar os dados e o programa. É comum, especialmente em micro-controladores, utilizar a arquitectura de Harvard descrita atrás, ou seja, duas memórias diferentes, uma para os dados e outra para o programa. Em alguns casos isto justifica-se pelo facto de o programa ser fixo e poder ser guardado em memória não volátil, enquanto que os dados têm de ser guardados em memória de leitura e escrita.

10.4 Interacção com o Exterior

A unidade central de processamento e a memória, ilustradas na Figura 10.1 são o núcleo central de um computador, mas, por si só, não podem desempenhar qualquer função útil. Para comunicar com os utilizadores, o computador usa diversos dispositivos periféricos tais como teclados, ratos, monitores e impressoras. Usa também discos e fitas magnéticas para armazenar dados de forma permanente e interfaces de rede para comunicar com outros computadores.

Estes dispositivos encontram-se ligados a *portos de entrada/saída* (em inglês, *Input/Output ports* ou *IO ports*). Os portos de entrada e saída controlam estes periféricos usando protocolos que variam de periférico para periférico. Do ponto de vista da UCP, os portos de entrada/saída são acedidos, quer para leitura quer para escrita, de uma forma semelhante à memória. Conceptualmente, a cada porto de entrada/saída está atribuído um ou mais endereços, usados pela UCP quando pretende ler dados desse porto ou escrever dados para o mesmo. A UCP, a memória e os portos de entrada/saída são tipicamente acondicionados numa única unidade física, para que possam comunicar através de barramentos curtos e de alta velocidade, tal como está ilustrado na Figura 10.2.

A distinção entre um acesso a memória e um acesso a um porto de entrada/saída é feita, quer pelo endereço utilizado, quer pelo valor das linhas de controlo, como se verá no Capítulo 14.

O tratamento que cada porto de entrada/saída dá aos dados depende do periférico que lhe está ligado. Por exemplo, um porto de entrada/saída que corresponda a uma porta série envia os dados recebidos para uma linha série. Esses dados são depois interpretados por um periférico que entenda o protocolo série, como, por exemplo, um modem. Por seu lado, os dados enviados

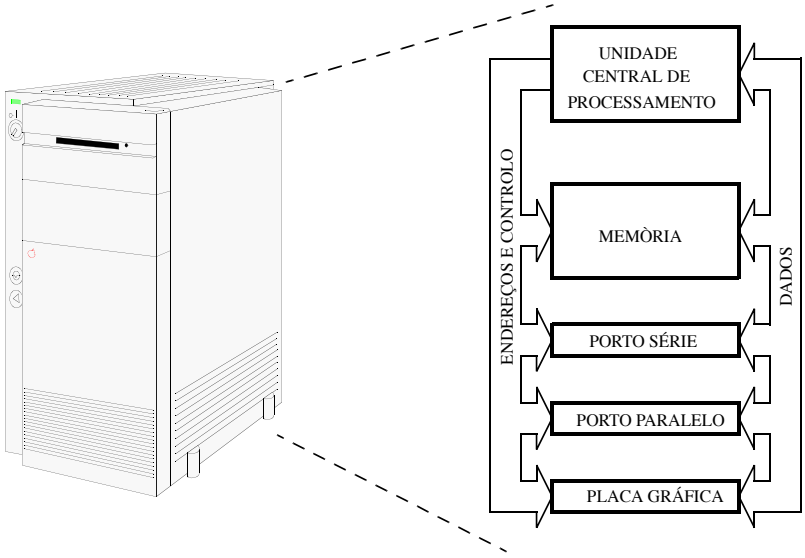


Figura 10.2: Processador, memória e portos de entrada/saída.

pelo rato pela linha série são lidos do porto de entrada/saída pela UCP quando faz um acesso ao porto correspondente.

Outros portos de entrada/saída tratam os dados de forma diferente. Um porto paralelo pode ser usado para comunicar com uma impressora, através de um conjunto de linhas. Outros portos correspondem a periféricos mais complexos, como, por exemplo, placas gráficas e controladores de disco. Uma placa gráfica interpreta os dados recebidos como comandos para desenhar pontos no monitor e gera o sinal vídeo que faz aparecer no mesmo a imagem correspondente, enquanto que um controlador de disco escreve os dados recebidos em localizações específicas do disco magnético ou lê os dados guardados nas localizações especificadas pela UCP. Na prática, por razões de desempenho e de modularidade do sistema, existem diversas alternativas para a leitura e escrita de dados em portos de entrada/saída, que serão estudadas em detalhe no Capítulo 14.

A Figura 10.3 descreve a forma como os periféricos exteriores são ligados ao computador. Note-se que, embora os periféricos sejam por vezes ligados à unidade central através de cabos, existem diversos periféricos que se encontram fisicamente dentro da unidade que alberga a UCP. Em geral, discos magnéticos, leitores de discos compactos e unidades de discos flexíveis estão alojadas dentro da unidade onde se encontra a unidade central de processamento, embora sejam periféricos ligados da mesma forma que um rato ou uma impressora, que são externos.

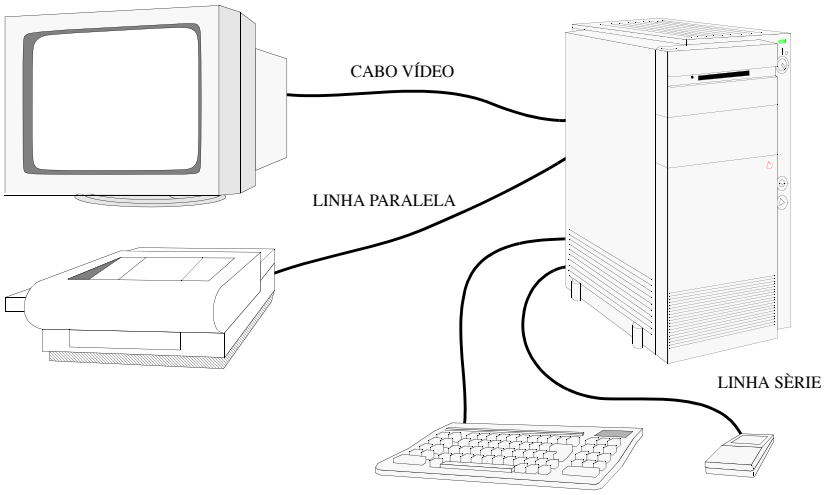


Figura 10.3: Computador e periféricos.

10.5 Níveis de Abstracção de um Computador

Um computador é geralmente utilizado a um nível de abstracção muito elevado. A maioria dos utilizadores de um computador não conhece, nem deseja conhecer, qualquer detalhe da sua organização interna ou da sua arquitectura. Tipicamente, um utilizador de um computador faz uso do mesmo executando uma aplicação, como, por exemplo, um editor, um processador de texto ou uma folha de cálculo. Estas aplicações foram desenvolvidas com o objectivo de tornar os computadores acessíveis a utilizadores que não saibam, ou não queiram, programar, e, através da utilização das mais recentes tecnologias de interacção com o utilizador, conseguiram de facto tornar a utilização dos computadores acessível a todos.

Porém, um computador é, de facto, um sistema digital programável, e, como tal, pode ser utilizado, programado ou configurado a diversos níveis de abstracção. A Figura 10.4 ilustra os diversos níveis de abstracção a que um computador pode ser visto.

O nível superior, o da aplicação, já foi referido. Um utilizador que utilize o computador a este nível interage com uma aplicação, tipicamente utilizando metáforas da vida real, tais como pastas, áreas de trabalho, folhas, etc.

Na maior parte dos casos, esta aplicação foi programada usando uma *linguagem de alto nível*, tal como Java ou C. É da responsabilidade dos programadores interpretarem as especificações que definem o funcionamento de uma aplicação e escreverem o código em linguagem de alto nível que as realiza.

Este código de alto nível não é directamente executado pelo computador,

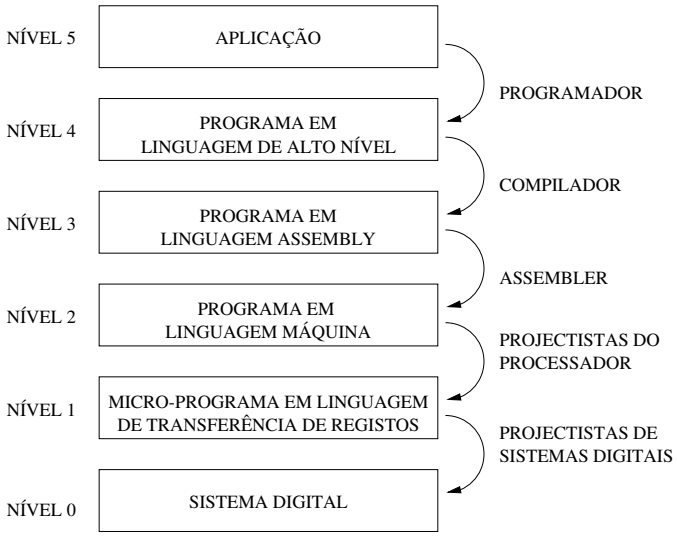


Figura 10.4: Níveis de abstracção a que um computador pode ser utilizado.

sendo primeiro traduzido, por um outro programa, chamado *compilador*, para uma linguagem muito mais simples, a *linguagem* assembly. Esta linguagem permite apenas especificar operações muito simples, tais como uma soma de duas posições de memória ou a cópia de uma posição de memória para outra.

Apesar da sua simplicidade, a linguagem *assembly* não é ainda directamente executada pelo processador. Tem de ser traduzida para *linguagem máquina*, que, essa sim, é executada directamente pelo processador. Esta linguagem máquina é, normalmente, guardada em memória e representa um programa que é executado directamente pelo processador. A tradução de linguagem *assembly* para linguagem máquina é realizada por um outro programa, o *assembler*.

Existem ainda níveis de abstracção mais baixos que a linguagem máquina. Os bits de um programa em linguagem máquina não são, em geral, usados directamente para controlar os registos e circuitos de dados do processador. Na maioria dos casos, a cada instrução em linguagem máquina correspondem diversas operações elementares de manipulação de dados e de transferência entre registos. Estas operações constituem o micro-programa, e são definidas pelos projectistas do processador.

Finalmente, definidas as micro-operações que têm de ser suportadas pelo hardware, há que projectar o sistema digital que as realiza. Este projecto é tipicamente efectuado por engenheiros especializados no projecto de sistemas digitais, que podem ou não ser os mesmos que projectaram e definiram o processador a um nível superior de abstracção.

Esta descrição simplificada do funcionamento de um computador omite diversos aspectos de maior ou menor importância, existindo, no entanto, um deles que não pode ser ignorado, mesmo numa descrição superficial como a que foi feita. Quase todos os computadores executam diversos programas, quer em simultâneo quer em sequência. A execução dos diversos programas é controlada por um programa especial, o sistema operativo do computador. O sistema operativo é ele mesmo um programa que tem como missão agendar

e gerir as diversas tarefas que o processador tem de executar. O estudo dos sistemas operativos merece, por si só, um livro, pelo que este assunto não é abordado com qualquer profundidade neste contexto. Porém, de um ponto de vista simplista, o sistema operativo é simplesmente um programa que distribui o tempo do processador entre as diversas tarefas que ele tem de efectuar. Estas tarefas incluem não só a execução de aplicações, mas também o atendimento de pedidos de dispositivos periféricos, a gestão do sistema de memória e a gestão de recursos partilhados entre as diversas aplicações e/ou utilizadores.

Conceptualmente, um processador pode ser utilizado ou programado a qualquer dos níveis superiores ao nível 0. Este nível é, tradicionalmente, fixo e inalterável. No entanto, existem tecnologias recentes que permitem reconfigurar as ligações entre os componentes digitais por forma a conseguir alterar a funcionalidade dos mesmos. A programação ou utilização a níveis superiores requer, tipicamente, menos esforço por parte do utilizador ou programador, mas o uso destas técnicas de baixo nível permite atingir grande eficiência.

10.6 Componentes de um Computador

Os capítulos que se seguem a este são dedicados ao estudo da arquitectura de computadores. Começa-se por analisar o funcionamento das unidades centrais de processamento do ponto de vista do programador em linguagem *assembly*. Para tal, o Capítulo 11 descreve a forma como uma típica unidade central de processamento é programada a este nível, usando para tal um hipotético microprocessador, o processador P3[3]. O processador P3 é semelhante a processadores comerciais, exibindo muitas das suas características, evitando, no entanto, as idiossincrasias inerentes a estes. Além de se apresentar a linguagem *assembly* deste processador, descreve-se também a forma como uma instrução *assembly* é traduzida para linguagem máquina, sendo assim abordados os níveis 2 e 3 da Figura 10.4.

A opção de usar um processador não comercial para ilustrar o funcionamento das unidades centrais de processamento foi tomada com a consciência que é uma solução que tem vantagens e desvantagens. A desvantagem mais significativa é a de o leitor não obter o valor acrescentado de conhecer profundamente um processador real, o que poderia ser de grande utilidade no futuro. Entenderam os autores que esta desvantagem seria mais do que compensada pelas vantagens pedagógicas de não ter de lidar, durante o processo de aprendizagem, com as complexidades inerentes ao uso de um processador comercial, muitas delas motivadas por razões puramente históricas.

Optou-se assim por definir este processador, utilizando uma abordagem em que os critérios de desempenho, realismo ou elegância assumiram uma posição secundária face aos critérios de clareza pedagógica e simplicidade. Na opinião dos autores, esta opção foi justificada pelo resultado final obtido, um processador simples, fácil de programar e com uma arquitectura de grande regularidade, o que permite simplificar grandemente o processo de aprendizagem.

Uma unidade central de processamento, tal como o processador P3, é um sistema digital complexo, que, embora sendo projectado usando as técnicas

[3] P3 = PPP = Pequeno Processador Pedagógico.

descritas em capítulos anteriores, merece um estudo mais detalhado. Assim, a estrutura interna do processador P3, ou seja, a sua micro-arquitectura, é descrita no Capítulo 12, sendo este capítulo usado não só para descrever este processador em particular, mas também para introduzir um número de técnicas de projecto de processadores que são utilizadas num grande número de sistemas. Este capítulo cobre assim os níveis 0 e 1 da Figura 10.4.

O Capítulo 13 descreve a forma como o sistema de memória de um processador moderno se encontra estruturado. De facto, a descrição feita acima apresenta uma visão excessivamente simplificada do sistema de memória de um computador moderno.

O Capítulo 14 descreve, com algum detalhe, a forma como a unidade central de processamento se interliga com os periféricos e quais os métodos e protocolos de comunicação mais utilizados. Descreve também, de forma necessariamente breve, alguns periféricos que são especialmente importantes, tais como discos e teclados.

Finalmente, o Capítulo 15 serve de breve introdução a tópicos mais avançados da área da arquitectura de computadores. Neste capítulo, serão abordados, muito brevemente, tópicos importantes para o desempenho de computadores, como *pipelining*, as filosofias CISC e RISC, assim como diversas técnicas que são usadas no projecto dos processadores actuais.

10.7 Sumário

Este capítulo introduz o conceito de computador como um sistema digital programável e descreve, de forma breve, a evolução histórica dos computadores.

Descreveu-se brevemente a arquitectura de um computador em termos dos seus diversos componentes, entre os quais se contam necessariamente a unidade central de processamento (UCP, ou processador), a memória e os portos de entrada/saída ligados a dispositivos periféricos.

Finalmente, foram referidos os diversos níveis de abstracção a que um computador pode ser conceptualizado, níveis estes que vão desde o nível da aplicação até ao nível da transferência de dados entre registos de um sistema digital.

Capítulo 11

Conjuntos de Instruções

Este capítulo é dedicado ao estudo de algumas alternativas possíveis para a *arquitectura do conjunto de instruções* de um processador. Como caso de estudo concreto, utiliza-se o conjunto de instruções do processador P3, um processador de 16 bits, micro-programado, concebido unicamente com fins didácticos, mas que exibe muitas das características de processadores comerciais simples.

Para cada processador ou família de processadores é definido um conjunto de instruções. A escolha de quais as instruções que devem ser incluídas no conjunto de instruções de um processador representa um problema complexo e intimamente relacionado com as funcionalidades pretendidas e a tecnologia utilizada. Este problema é resolvido por equipas de projectistas, os arquitectos do conjunto de instruções, após efectuarem uma análise das alternativas existentes e das limitações impostas pelas especificações.

11.1 Linguagens de Programação

Como foi visto no capítulo anterior, um computador pode ser programado usando linguagens com níveis de abstracção muito diversas.

O nível de abstracção mais baixo que se considera neste capítulo é o da *linguagem máquina*. Cada instrução de linguagem máquina é constituída por um conjunto de bits, directamente interpretável pelo processador. Considere-se, a título de exemplo, a sequência de instruções de linguagem máquina do processador P3, representada na Tabela 11.1, que soma três números inteiros guardados nas posições de memória `N1=00A0h`, `N2=00B0h`, `N3=00B1h`, e guarda o complemento do resultado na posição `N4=F000h`. Esta sequência de bits não parece ter qualquer significado para um programador, embora especifique de forma não ambígua as operações a efectuar. Com efeito, um programa em linguagem máquina tem a desvantagem de ser muito difícil de entender por seres humanos. Por essa razão, programar directamente em linguagem máquina exige dos programadores um esforço muito grande de memorização e de consulta de documentação, revelando-se muito ineficiente em termos do tempo que é necessário investir.

No nível de abstracção imediatamente superior encontra-se a linguagem *assembly*. Cada instrução nesta linguagem corresponde a uma instrução de linguagem máquina, mas, em vez de ser especificada em termos de zeros e

Endereço		Valor	
Base 2	Base 16	Base 2	Base 16
0001000000000000	1000	1010111001110000	AE70
0001000000000001	1001	0000000010100000	00A0
0001000000000010	1002	1000011001110000	8670
0001000000000011	1003	0000000010110000	00B0
0001000000000100	1004	1000011001110000	8670
0001000000000101	1005	0000000010110001	00B1
0001000000000110	1006	0100000000000001	4001
0001000000000111	1007	1010110001110000	AC70
0001000000001000	1008	1111000000000000	F000

Tabela 11.1: Instruções em linguagem máquina do processador P3.

uns, é especificada utilizando mnemónicas e nomes simbólicos. Por exemplo, a instrução de somar dois números guardados nos registos `R1` e `R2` e depositar o resultado em `R1` poderá ser codificada como `ADD R1,R2`. Para o programador, é muito mais fácil memorizar esta instrução do que o seu equivalente em linguagem máquina que é, no caso do P3, `1000011001000010`, ou seja, `8642h`. Ao programa descrito em linguagem máquina na Tabela 11.1 corresponde o programa em *assembly* descrito na segunda coluna da Tabela 11.2. Em

Endereço	Código *assembly*	Código máquina
1000h	`MOV R1, M[00A0h]`	AE70
1001h		00A0
1002h	`ADD R1, M[00B0h]`	8670
1003h		00B0
1004h	`ADD R1, M[00B1h]`	8670
1005h		00B1
1006h	`NEG R1`	4070
1007h	`MOV M[F000h], R1`	AC70
1008h		F000

Tabela 11.2: Correspondência entre as instruções *assembly* e máquina.

geral, além do uso de mnemónicas para as instruções, as linguagens *assembly* permitem definir nomes simbólicos para endereços de memória, constantes numéricas, constantes alfa-numéricas e endereços. Neste exemplo, usaram-se directamente os endereços pretendidos, mas teria sido possível definir nomes simbólicos `N1`, `N2`, `N3` e `N4` para representarem os valores `N1=00A0h`, `N2=00B0h`, `N3=00B1h` e `N4=F000h` usando a directiva `EQU` da linguagem *assembly*. Admitiu-se, no exemplo anterior, que o programa começava na posição `1000h`, o que em *assembly* se especifica usando a directiva `ORIG`. Se se usarem estas directivas, a seguinte sequência de instruções *assembly* corresponde exactamente à sequência de instruções máquina da Tabela 11.1. Por análise e comparação entre o código do Programa 11.1 e as instruções máquina da Tabela 11.1, é possível verificar que, embora ambos descrevam exactamente a mesma sequência de instruções, o código em *assembly* é muito mais compreen-

```
ORIG  1000h
N1    EQU   00A0h
N2    EQU   00B0h
N3    EQU   00B1h
N4    EQU   F000h

      MOV   R1, M[N1]
      ADD   R1, M[N2]
      ADD   R1, M[N3]
      NEG   R1
      MOV   M[N4], R1
```

Programa 11.1: Programa em *assembly* que soma três números em memória.

sível e fácil de editar, depurar ou modificar. Note-se que a cada instrução pode corresponder uma ou mais palavras de memória, dependendo do tipo de instrução e dos seus operandos. Pode também observar-se que as directivas não se traduzem directamente para instruções de linguagem máquina, mas podem traduzir-se na ocupação de uma ou mais posições de memória.

Quando programa em *assembly*, o programador continua a ter de conhecer a arquitectura e os recursos do processador de forma detalhada, uma vez que as instruções *assembly* só fazem sentido para aquele processador. Embora semelhantes entre si, as linguagens *assembly* são diferentes de processador para processador.

A tradução de um programa escrito em *assembly* para um programa escrito em linguagem máquina é um processo relativamente simples, uma vez que a cada instrução *assembly* corresponde uma e uma só instrução em linguagem máquina. Esta tradução pode ser feita manualmente, mas geralmente é efectuada por um programa chamado *assembler*. O *assembler* aceita um programa escrito em *assembly* e gera um programa em linguagem máquina, processando as directivas e traduzindo as instruções descritas no ficheiro fonte. A Secção 11.8 descreve em detalhe o funcionamento de um *assembler* para o processador P3.

Como foi referido no capítulo anterior, a programação de um computador pode ser feita utilizando níveis de abstracção superiores usando linguagens de alto nível que são, na maior parte dos casos, independentes do processador. Exemplos de linguagens deste tipo são as linguagens *C*, *Pascal* e *Java*. Programas escritos nestas linguagens têm de ser traduzidos para linguagem *assembly* e daí para linguagem máquina, antes de serem executados pelo processador. Este processo de tradução é muito mais complexo que o referido anteriormente, e é efectuado por programas chamados *compiladores*. Em geral, a cada instrução de uma linguagem de alto nível correspondem várias instruções *assembly*. Em alguns casos, uma linguagem de alto nível poderá ser interpretada directamente por um programa, o *interpretador*, não havendo assim necessidade de compilação, mas conduzindo a uma execução do programa consideravelmente mais lenta. Linguagens que, embora possam ser compiladas, são muitas vezes utilizadas desta forma são o *LISP* e o *Scheme*, para as quais existem interpretadores muito eficientes. Também o *Java* é por vezes usado com um interpretador, embora o programa original seja, neste caso, traduzido para

uma linguagem intermédia que se aproxima de linguagem máquina.

11.2 Instruções *Assembly*

Uma vez que existe uma correspondência directa entre uma instrução *assembly* e uma instrução em linguagem máquina, usar-se-á a partir de agora o termo "*instrução*" para designar qualquer uma das duas. Tipicamente, o contexto será suficiente para indicar se a referência é a uma instrução de linguagem máquina ou a uma instrução *assembly*.

Uma instrução é guardada em memória como uma combinação de bits que especifica, de forma única, uma sequência de operações de transferências entre registos que deve ser executada pelo processador. De um modo geral, as instruções são executadas pela ordem em que estão ordenadas em memória. Em diversos casos, porém, esta sequência de execução é alterada, quer porque é executada uma instrução de controlo, quer porque o processador recebe um pedido externo e tem de alterar a ordem de execução das instruções. Existem três grandes classes de instruções:

- As *instruções de transferência de dados* transferem informação de um local (registo, posição de memória ou porto de entrada/saída) para outro, sem alterar a informação original.
- As *instruções de manipulação de dados* aplicam um operador aritmético ou lógico ao *operando* ou *operandos*, especificados pela sua localização, e guardam o resultado da mesma.
- As *instruções de controlo* permitem alterar a sequência normal de instruções e definir qual a próxima instrução a ser executada.

As instruções de transferência de dados são usadas para copiar ou salvaguardar dados. São tipicamente usadas para copiar valores para registos ou posições de memória onde possam ser manipulados, para criar diversos tipos de estruturas de dados como, por exemplo, *arrays* e listas, e, ainda, para efectuar operações de entrada e saída.

No processador P3, a instrução de transferência de dados mais simples é a instrução `MOV`. Por exemplo, a instrução `MOV R1, M[00A0h]` copia o conteúdo da posição de memória `00A0h` para o registo `R1`.

As instruções de manipulação de dados são as que efectivamente executam as operações úteis num programa. Todos os processadores suportam instruções aritméticas básicas tais como adição e subtracção. São também suportadas por todos os processadores operações lógicas básicas, como a disjunção e a conjunção, e operações de deslocamento. Muitos processadores executam também operações aritméticas mais complexas como multiplicação, divisão e outras funções matemáticas sobre números inteiros ou reais.

Entre estas instruções encontram-se, tipicamente, instruções como `ADD`, `SUB`, `INC`, `MULT`, `AND`, `OR` e `XOR`. Por exemplo, no processador P3, a instrução `ADD R1,M[00B0h]` soma o conteúdo do registo `R1` ao conteúdo da posição de memória `00B0h`, guardando o resultado no registo `R1`.

A sequência de instruções a executar é controlada por um registo especial, o *contador de programa*. Na maioria dos processadores, este registo é designado por `PC` (*program counter*). Este registo contém o endereço de memória

onde se encontra a próxima instrução a ser executada. Tipicamente, as instruções de transferência e de manipulação de dados incrementam o contador de programa para o valor correspondente à posição de memória onde se inicia a instrução que se segue na sequência normal de execução, o que faz com que essa instrução seja a próxima a ser executada.

As instruções de controlo permitem alterar a ordem de execução de instruções, incondicionalmente ou, em alternativa, apenas se a operação anterior produzir um resultado que satisfaça certas condições. Mais especificamente, as instruções de controlo permitem especificar qual o endereço da próxima instrução a executar, através da alteração do registo contador de programa, `PC`. Estas instruções são estudadas em mais detalhe na Secção 11.5. Este tipo de instruções permite tomar decisões em função dos resultados de cálculos anteriores ou de acontecimentos externos, sendo portanto fundamentais para o funcionamento correcto de qualquer programa.

Um exemplo de uma instrução de controlo no processador P3 é a instrução `JMP`. Por exemplo, a instrução `JMP 00CCh` faz com que a próxima instrução a ser executada seja a instrução na posição de memória `00CCh`.

A combinação de bits que corresponde a cada instrução codifica, de forma única, quais as operações a executar, quais os operandos e qual a localização onde deve ser guardado o resultado. Assim, cada instrução máquina contém, necessariamente, três componentes, que se reflectem na estrutura da instrução:

- O *código de operação* (*operation code* ou *opcode*), que especifica qual a operação a executar. Por exemplo, no processador P3, o código para somar dois números é representado pela combinação de bits 100001.

- A especificação dos operandos aos quais deve ser aplicada a operação. Por exemplo, se num processador com 8 registos se pretender somar o registo `R1` com o registo `R2`, seria possível usar 6 bits (3 + 3) para representar os dois operandos, o que corresponderia à sequência 001010. O número de operandos de cada instrução varia com o tipo de instrução e com o processador, desde processadores em que as instruções não têm operandos explícitos (todos os operandos são guardados em localizações pré-estabelecidas) até processadores em que as instruções aceitam um, dois ou três operandos.

- A especificação da localização (registo ou memória) onde deverá ser guardado o resultado da operação. Por exemplo, se se pretender guardar o resultado no registo `R1`, num processador com 8 registos, a sequência de bits 001 poderá ser usada para o indicar.

A título de exemplo, no processador P3, a instrução `ADD R1,M[N2]`, usada no exemplo da Secção 11.1, é codificada com duas palavras de 16 bits. A primeira palavra contém o código de operação (100001) nos bits mais significativos. Contém ainda, nos três bits menos significativos, o número do registo que deve ser usado como primeiro operando e como destino do resultado (001). Os restantes bits indicam, usando uma codificação que será discutida na Secção 11.7.3, que o segundo operando deve ser obtido a partir da posição de memória cujo endereço está guardado na segunda palavra desta instrução. Assim, esta instrução é codificada com as palavras `8670h` e `00B0h`, que correspondem aos valores em binário `1000011001110000` e `0000000010110000`.

Na prática, a especificação dos operandos e da localização do resultado pode ser consideravelmente complexa. Isto deve-se ao facto de que as instruções de manipulação de dados e as instruções de transferência operarem sobre valores localizados numa das seguintes possíveis posições:

- Registos internos do processador
- Constante, especificada na própria instrução
- Localizações de memória
- Portos de entrada/saída

Para que seja possível especificar de forma flexível qualquer uma destas localizações são geralmente utilizadas formas mais complexas de codificação. A Secção 11.7 descreve o mecanismo de codificação das instruções no processador P3.

Muitos processadores comerciais possuem a possibilidade de executar instruções mais complexas. Por exemplo, o processador 8086 dispõe de uma instrução que copia um determinado número de posições de memória para uma outra zona de memória. Uma instrução deste tipo requer pelo menos três operandos: dois para especificar os endereços de origem e destino, e um para especificar o número de posições de memória a copiar.

Para instruções complexas deste tipo, muitas vezes um ou mais operandos são fixos e pré-definidos, para evitar a necessidade de codificar todos os operandos na instrução. Este tipo de endereçamento, chamado *endereçamento implícito*, é também usado em instruções mais simples mas muito comummente utilizadas, como por exemplo as instruções de manipulação da pilha descritas na Secção 11.3.5. Existem também processadores, chamados *processadores de pilha*, em que todos os operandos são guardados numa pilha, e todas as operações aritméticas operam sobre os dois operandos guardados no topo da pilha. Estes operandos são, assim, especificados de forma implícita. Noutras máquinas, um dos operandos é sempre um registo especial, denominado o *acumulador*, pelo que uma operação aritmética apenas necessita ter um operando definido explicitamente.

11.3 Especificação dos Operandos

Existem quatro origens possíveis para os operandos de uma instrução: registos internos do processador, constante especificada na própria instrução, uma localização em memória, e portos de entrada/saída.

11.3.1 Registos Internos

A utilização de operandos em registos é vantajosa por duas razões. Em primeiro lugar, o acesso a dados guardados num registo interno é muito mais rápido do que o acesso a dados em memória. Em segundo lugar, existem menos registos internos do que posições de memória, o que permite utilizar menos bits para especificar um registo do que uma posição de memória. Isto leva a que as instruções que utilizam apenas registos necessitem de menos bits para

a sua codificação, ocupando menos memória e sendo lidas e executadas mais rapidamente.

Os registos internos podem estar organizados de diversas formas. Na sua forma mais simples, os registos internos são simplesmente um conjunto de registos, com igual funcionalidade, organizados num banco de registos e especificados pelo seu número. Este tipo de organização foi usado no banco de registos descrito na Secção 7.5.4, e será utilizado no processador P3.

Outros processadores têm organizações mais complexas. Podem existir registos com funções específicas, que são sempre usados em determinadas operações. Em processadores que tenham um registo especial denominado acumulador, este guarda o resultado das operações e é também muitas vezes usado para especificar o endereço de acesso à memória. Processadores complexos, como, por exemplo, a família x86 da Intel, têm diversos registos com funções específicas, como a manipulação de cadeias de caracteres em memória e a gestão da memória do processador.

Mesmo os processadores simples e com organizações muito regulares dos registos possuem, geralmente, dois registos de uso específico, que podem ou não ser acessíveis directamente ao programador. Estes dois registos são o contador de programa (`PC`), cuja funcionalidade é descrita em mais detalhe na Secção 11.5 e o *apontador para a pilha* (em inglês, *stack pointer*, `SP`), cuja utilidade será estudada na Secção 11.3.5.

11.3.2 Constantes Especificadas na Própria Instrução

Quase todos os processadores podem executar instruções em que um ou mais operandos são valores constantes, especificados na própria instrução. Por exemplo, a instrução `ADD R1,0005h` soma ao conteúdo do registo `R1` o valor `5`, valor este que se encontra especificado na própria instrução.

Embora este método possa parecer tanto ou mais eficiente que a utilização de um valor em registo, isso não é, geralmente, verdade. De facto, a especificação de uma constante na própria instrução obriga, na prática, a utilizar mais uma palavra de memória para codificar a instrução, onde é guardada a constante.

Assim, para carregar este valor, é geralmente necessário fazer um acesso extra a memória[1] o que se revela muito mais lento que um acesso a um registo interno. Desta forma, em termos de velocidade, o uso de uma constante especificada na própria instrução é, na prática, equivalente, em termos de velocidade de execução, ao uso de um operando guardado em memória.

11.3.3 Memória e Portos de Entrada/Saída

Existem duas formas possíveis de tratar os dados provenientes de portos de entrada/saída. Uma filosofia, adoptada em muitas famílias de processadores, consiste em disponibilizar um conjunto de instruções especiais para a entrada e saída de dados. Esta solução que equivale a ter entradas/saídas independentes do sistema de memória (*independent IO*) foi adoptada em muitos processadores, e, nomeadamente, na família x86 da Intel. Neste caso, existe um espaço de

[1] O carregamento da constante pode ser mais rápido se a constante for especificada de uma forma que não exija uma palavra extra na instrução.

endereçamento específico para operações de entrada saída. Um acesso a um dado porto de entrada/saída é especificado colocando um dado valor nos bits de endereço do processador (ou em parte deles) e controlando uma ou mais linhas adicionais que especificam que a operação é uma operação de entrada ou saída e não um acesso a memória.

Em alternativa, é possível considerar que os portos de entrada/saída são mapeados para o espaço de memória do processador (*memory-mapped IO*), sendo os dados disponíveis nestes portos manipulados através das mesmas operações que manipulam os dados em memória. Esta solução permite reduzir a complexidade do conjunto de instruções, tendo como desvantagem principal a redução do espaço de memória endereçável pelo processador. Quando esta solução é adoptada, a leitura de dados de um porto de entrada é executada como se tratasse de uma leitura de determinadas posições de memória, enquanto que a escrita de dados em portos de saída é executada como se se tratasse de uma escrita em determinadas posições de memória. Compete aos dispositivos exteriores ao processador distinguir, através da correcta descodificação dos endereços, acessos feitos a portos de entrada/saída e acessos feitos a memória. À menor complexidade do conjunto de instruções vai corresponder uma maior sofisticação da arquitectura do sistema de memória, como será estudado no Capítulo 13. Quando esta solução é adoptada as instruções de entrada/saída podem usar todos os modos de endereçamento disponíveis, sendo os portos tratados pelo programador como se fossem posições de memória. Esta abordagem tem ainda a vantagem adicional de se poder processar dados directamente a partir de (ou com destino a) portos de entrada/saída. Esta foi a solução adoptada no processador P3, onde todos os acessos a portos de entrada e saída são feitos usando as operações normais de transferência de dados.

Para aceder a operandos em memória ou a portos de entrada/saída, é necessário especificar o endereço da memória ou o porto onde o operando se encontra, o que pode ser feito de diversas formas. As diferentes alternativas existentes para a especificação da localização de um operando, ou seja, os possíveis *modos de endereçamento*, são estudadas em detalhe na secção seguinte.

11.3.4 Modos de Endereçamento

A escolha dos modos de endereçamento suportados por um processador tem um grande impacto na sua estrutura interna e na flexibilidade do conjunto de instruções. Considere-se um acesso a memória em que o valor do registo `RX` e/ou o valor de uma palavra `W` são usados para especificar a localização do operando. Existem diversas formas de usar o valor de `RX` e de `W` para definir o valor do operando ou a localização do mesmo. Caso o operando se encontre numa posição de memória, o endereço onde ele se encontra é chamado de *endereço efectivo*. A Tabela 11.3 sumaria alguns modos de endereçamento comummente utilizados.

No *endereçamento por registo*, o operando é o valor guardado num registo interno. Este modo de endereçamento é eficiente, uma vez que, como foi atrás referido, obter um operando a partir de um registo é mais eficiente que obter um operando a partir de memória.

No modo de *endereçamento indirecto por registo*, o conteúdo do registo especificado indica o endereço efectivo de memória onde os dados se encontram, no caso de uma leitura, ou onde devem ser guardados, no caso de uma escrita.

Modo de endereçamento	Operação
Por registo	op ← RX
Indirecto por registo	op ← M[RX]
Imediato	op ← W
Directo	op ← M[W]
Indexado	op ← M[RX+W]
Relativo	op ← M[PC+W]
Baseado	op ← M[SP+W]
Indirecto	op ← M[M[W]]
Duplamente indirecto por registo	op ← M[M[RX]]
Implícito	

Tabela 11.3: Principais modos de endereçamento utilizados.

No modo de *endereçamento imediato*, o valor do operando encontra-se codificado na própria instrução, usando, se necessário, palavras de memória adicionais. Este modo de endereçamento só pode ser usado em operações de leitura de operandos, uma vez que o uso deste modo de endereçamento para definir a localização do resultado implicaria uma escrita na zona de memória onde está guardado o código máquina, com a consequente alteração do mesmo.

No modo de *endereçamento indexado*, o conteúdo do registo indicado é adicionado a um valor codificado na própria instrução para obter o endereço efectivo que deve ser usado pela operação para ler o operando e/ou guardar o resultado na memória.

No modo de *endereçamento directo*, o endereço efectivo de memória que deve ser usado pela operação para ler o operando e/ou guardar o resultado é especificado na própria instrução.

Os modos de *endereçamento relativo* e *endereçamento baseado* são casos especiais do endereçamento indexado, em que o registo `RX` toma um valor particular. No endereçamento relativo o registo `RX` é o contador de programa, `PC`, e no endereço baseado, o registo `RX` é o apontador para a pilha do processador, `SP`.

Os dois últimos modos descritos na Tabela 11.3 são menos utilizados e são suportados apenas por um pequeno número de processadores. No *endereçamento indirecto*, o valor especificado na instrução indica a posição de memória que guarda o endereço efectivo. No *endereçamento duplamente indirecto* por registo, o conteúdo do registo indica, de forma similar, a posição de memória que guarda o endereço efectivo onde se encontra o operando. Estes dois modos de endereçamento requerem assim dois acessos a memória, um para obter o endereço efectivo e outro para obter o operando (ou guardar o resultado) pelo que são consideravelmente menos utilizados que os anteriormente descritos.

No modo de *endereçamento implícito*, diversos registos, não especificados na instrução, são utilizados para especificar a posição dos operandos. Uma vez que a utilização particular que é feita deste modo de endereçamento varia de acordo com a instrução em que é utilizado, não é possível sistematizar a sua utilização da mesma forma que para os outros modos de endereçamento referidos.

As instruções do processador P3 podem especificar operandos usando qualquer um dos modos de endereçamento da Tabela 11.3, com excepção dos dois

últimos, que não são suportados por este processador. No entanto, no processador P3, apenas um dos operandos pode usar um dos modos mais complexos, enquanto que o outro operando deve usar necessariamente o modo de endereçamento por registo, opção esta muito comum em processadores comerciais.

Diferentes processadores comerciais exibem diferentes filosofias no que respeita aos modos de endereçamento suportados e à forma como o endereço dos operandos da instrução é obtido a partir dos dados codificados na mesma. Em particular, a utilização do conceito de memória segmentada, utilizada, por exemplo, nos processadores da Intel, implica a existência de registos que são usados implicitamente no processo de endereçamento. Estes e outros mecanismos de endereçamento serão estudados em mais detalhe no Capítulo 13.

11.3.5 Utilização de Pilhas

Uma opção muito utilizada para aceder e guardar operandos em memória consiste no uso de uma *pilha* (em inglês, *stack*.). Uma pilha é um conjunto contíguo de posições de memória cujo acesso é gerido por um registo especial, o apontador de pilha, geralmente denominado `SP` (em inglês, *stack pointer*). Conceptualmente, a pilha consiste num conjunto de posições de memória sobrepostas umas às outras, e às quais apenas é possível aceder uma a uma, a partir do topo, quer para colocar um dado (operação de `PUSH`) quer para o retirar (operação de `POP`). A pilha tem uma base que corresponde à posição inferior, que não deve ser ultrapassada, o que significa que não devem ser retirados dados que não foram lá colocados. Em alguns processadores, a base da pilha é especificamente considerada e acessos para lá desse limite são impedidos, mas, no caso do P3, esta verificação não é efectuada pelo processador. A forma mais simples de utilização permite apenas aceder ao valor guardado no topo da pilha, e que corresponde ao último valor lá colocado. Assim, é possível guardar-se um valor no topo da pilha usando a operação de `PUSH` (ou semelhante) ou recuperar o valor guardado no topo da pilha através da operação de `POP` (ou semelhante).

Quando um novo valor é guardado na pilha, o valor do registo `SP` é incrementado e quando um valor é retirado da pilha, o valor do apontador é decrementado, conforme exemplificado na Figura 11.1. Desta forma, o valor

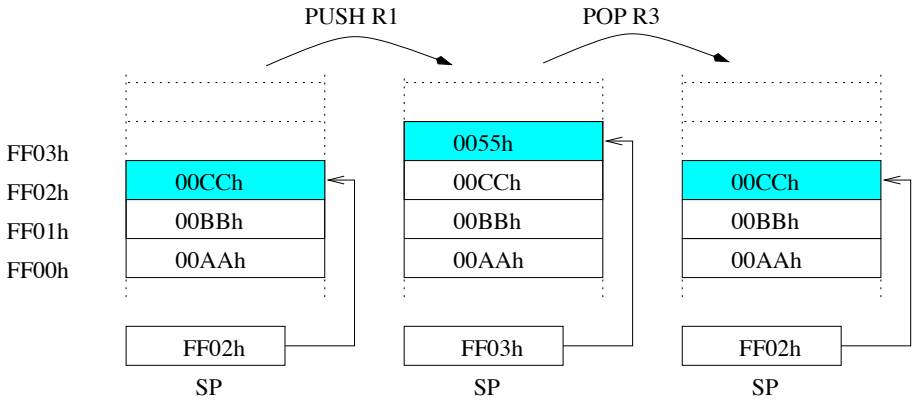


Figura 11.1: Exemplo de uso da pilha do processador.

do apontador de pilha indica sempre a posição de memória que representa o topo da pilha. Considere a sequência de operações exemplificada nesta figura. Inicialmente, o registo `R1` tem o valor `0055h`, e o apontador de pilha (`SP`) o valor `FF02h`. Após a instrução de `PUSH R1`, o valor do registo `SP` é incrementado para `FF03h` que é a posição de memória onde fica guardado o último valor colocado na pilha. A instrução `POP R3` tem como resultado carregar em `R3` o valor que se encontrava no topo da pilha (`0055h`) e decrementar o valor do registo `SP`. Note-se que, embora tal não se encontre representado na figura, o valor da posição de memória `FF03h` mantém o valor `0055h` após a operação de `POP`. No entanto, este valor já não está, de um ponto de vista lógico, na pilha, pelo que não é representado.

Existem numerosas utilizações para a pilha do processador. Uma das mais comuns consiste em utilizar a pilha como um espaço temporário para guardar o valor de registos que são momentaneamente necessários para outras operações. Assim, o programador pode decidir colocar os conteúdos do registos `R1`, `R2` e `R3` na pilha, com o objectivo de libertar estes registos para alguma operação. Quando esta operação estiver terminada, pode retirar os valores da pilha, pela ordem inversa, repondo assim os valores originais nos registos.

Uma outra aplicação comum para a pilha é a passagem de parâmetros para subrotinas. Esta aplicação será descrita em detalhe na Secção 11.5.2.

Dado que a pilha tem numerosas aplicações, alguns processadores podem possuir diversas pilhas, que podem ou não ser acessíveis ao programador. Neste caso, existirá mais do que um registo apontador de pilha.

Embora o mecanismo básico de utilização de uma pilha seja muito simples, é necessário tomar algumas precauções para que a mesma seja correctamente utilizada e não interfira com o funcionamento correcto do processador.

Em primeiro lugar, a pilha deve utilizar uma zona de memória que não seja utilizada por mais nenhum programa ou subrotina. Para garantir que a zona de memória usada pela pilha não entra em conflito com as zonas de memória utilizadas por outros módulos, é importante inicializar o valor do registo `SP` (definindo assim implicitamente a base da pilha) com um valor tal que garanta que, dentro das condições normais de utilização, o topo da pilha nunca atingirá a zona onde o código se encontra nem a zona reservada para dados. É também necessário garantir que existe uma operação de remoção de dados do topo da pilha para cada operação de inserção de dados na mesma. Se não existir esta correspondência, o valor do apontador de pilha crescerá (ou decrescerá) sem parar, acabando as operações de escrita por corromper zonas de memória reservadas para outras utilizações.

É de notar também que o funcionamento lógico das operações de `PUSH` e `POP` não se altera se a pilha crescer na direcção inversa à ilustrada na Figura 11.1, isto é, se o registo `SP` for decrementado quando se insere um valor na pilha, ao invés de ser incrementado. De igual forma, o valor do registo `SP` pode ser alterado antes ou depois da escrita em memória propriamente dita. No primeiro caso, o registo `SP` aponta para a primeira posição livre, enquanto que no segundo caso, aponta para a valor no topo da pilha.

No processador P3, o valor do registo `SP` é **decrementado** após uma operação de `PUSH`. Este decremento é efectuado **após** a escrita do valor, pelo que o registo `SP` aponta sempre para a próxima posição livre.

11.4 Codificação das Instruções

Na maioria dos processadores, as instruções básicas de manipulação de dados actuam sobre um máximo de dois operandos. Por exemplo, uma instrução de adição poderá calcular $RES = OP1 + OP2$. Para especificar completamente uma instrução deste tipo, é necessário especificar qual o tipo de operação, quais os operandos $OP1$ e $OP2$ e qual a localização de destino do resultado.

Para analisar as diversas possibilidades de codificação das instruções de um processador, considere-se um processador com a seguinte configuração:

- Registos: o processador possui 8 registos internos de 16 bits cada um, numerados de 0 a 7.
- Memória: o processador acede a uma memória de 2^{16} palavras de 16 bits cada, o que significa que cada posição de memória é acedida com um endereço de 16 bits. Isto corresponde a uma capacidade total de endereçamento de 64K palavras ou 128K octetos.
- Portos de entrada/saída: os portos de entrada saída são mapeados em memória, fazendo-se o acesso aos mesmos através de instruções normais de transferência de dados.
- Instruções: o conjunto de instruções contém 40 instruções.

Suponha-se que, para este hipotético processador, se pretende que cada instrução de dois operandos possa ter como origem e destino qualquer das possíveis localizações, e observe-se qual o impacto que esta opção tem na codificação das instruções. Analise-se, assim, quantos bits são necessários para codificar cada instrução:

- Código de operação: uma vez que existem 40 instruções diferentes, serão necessários 6 bits para codificar a operação a realizar.
- Operandos: cada um dos operandos pode ser especificado usando um dos 4 primeiros modos de endereçamento especificados na Tabela 11.3. Podem ser utilizados dois bits para indicar qual o modo de endereçamento. Para alguns modos de endereçamento, é necessário especificar um registo e um endereço.

Com estas opções, e se se pretender uma codificação uniforme para todas as instruções, obtém-se a codificação ilustrada na Figura 11.2. A vantagem de

	Resultado			Operando 1			Operando 2		
Código	Modo	Reg	Ender	Modo	Reg	Ender	Modo	Reg	Ender
6	2	3	16	2	3	16	2	3	16

Figura 11.2: Codificação de uma instrução com três operandos especificados com qualquer um dos quatro modos de endereçamento permitidos.

um formato uniforme deste tipo é clara, uma vez que esta abordagem permite uma total flexibilidade na especificação dos operandos, que podem estar em qualquer localização. Note-se, porém, que esta codificação exige um total de

69 bits por cada instrução o que implica que cada instrução ocupe 5 palavras de memória, ou seja, 10 octetos.

Considere-se, por exemplo, o Programa 11.2, que, utilizando instruções deste hipotético processador, soma o conteúdo das posições de memória `N1`, `N2` e `N3`, e guarda o resultado, complementado, na posição de memória `N4`:

```
ADD  R1, M[N1], M[N2]
ADD  R1, R1, M[N3]
NEG  M[N4], R1
```

Programa 11.2: Programa em *assembly* que soma três números em memória, para um processador que permite modos de endereçamento complexos para todos os operandos.

Note-se que, embora este código contenha apenas três instruções, o código máquina correspondente utiliza 15 palavras de memória.

Em termos de desempenho, esta opção revela-se, em geral, desvantajosa, uma vez que, não só os programas em linguagem máquina ocupam mais memória do que seria realmente necessária, como acabam por se tornar mais lentos. Com efeito, a superior flexibilidade dos modos de endereçamento permitidos é contrabalançada pelo tempo que demora a carregar da memória e a executar uma instrução.

É assim vantajoso restringir o número de operandos e os modos de endereçamento permitidos, ou, pelo menos, permitir a existência de instruções com modos de endereçamento mais limitados. Uma restrição comummente adoptada consiste em obrigar a que o resultado da operação seja guardado na mesma localização que o primeiro operando. Embora isto implique a destruição do valor guardado neste operando, esta restrição reduz consideravelmente o número de bits necessário para codificar cada instrução.

Adicionalmente, é possível obrigar a que pelo menos um dos operandos esteja disponível em registo. Novamente, isto reduz fortemente o número de bits necessário para codificar a instrução, embora implique uma considerável restrição no tipo de operações que podem ser efectuadas.

Com estas duas restrições, é possível usar um formato mais compacto para representar cada instrução. É necessário utilizar um bit de direcção que indicará se o modo de endereçamento sem restrições é aplicado ao primeiro ou ao segundo operando. Por exemplo, a instrução `ADD M[R1+N1],R2` terá o valor deste bit igual a 1 para indicar que o modo de endereçamento genérico se aplica ao primeiro operando e, consequentemente, à localização do resultado.

Com esta abordagem, é agora possível utilizar o formato descrito na Figura 11.3. Com esta codificação, cada instrução gasta agora apenas duas palavras

Código	Direcção	Modo	Reg	Reg	Ender
6	1	2	3	3	16

Figura 11.3: Codificação de uma instrução em que um dos operandos deve ser endereçado por registo.

de memória. Como contrapartida, já não é possível representar uma instru-

ção como `ADD R1,M[N1],M[N2]`. No entanto, é possível representar todas as instruções utilizadas no exemplo da página 156, conduzindo ao Programa 11.3 que ocupa apenas 10 palavras de memória para realizar a mesma tarefa, contra 15 na versão anterior. Na prática, é geralmente necessário utilizar mais do que

```
MOV  R1, M[N1]
ADD  R1, M[N2]
ADD  R1, M[N3]
NEG  R1
MOV  M[N4], R1
```

Programa 11.3: Programa em *assembly* que soma três números em memória, para um processador que permite um modo de endereçamento complexo apenas para um operando.

um formato para codificar instruções. Assim, uma instrução em que os dois operandos sejam registos pode ser codificada numa única palavra de memória, enquanto que uma instrução que use um modo de endereçamento indexado ou imediato (entre outros) pode necessitar de duas palavras de memória. Nestes casos, o valor de um bit ou combinação de bits na primeira palavra especifica o modo de endereçamento e a existência ou não de uma segunda palavra.

Em alguns processadores onde se define como um dos objectivo fundamentais que as instruções mais frequentes sejam codificadas o mais uniformemente possível e em pouco espaço, ainda que perdendo flexibilidade, impõe-se mesmo que todas as operações lógicas e aritméticas sejam realizadas entre registos. Neste caso, todos os acessos a memória são realizados através de operações de transferência de dados (`LOAD` e `STORE`), pelo que a arquitectura destes processadores é geralmente referida como do tipo *load/store*.

Num processador deste tipo, poder-se-ia utilizar o seguinte formato para as instruções que especificam operações aritméticas, sendo neste caso possível codificar instruções como `ADD R1,R2,R3` numa só palavra de memória. Já as

Código	Reg	Reg	Reg
6	3	3	3

Figura 11.4: Codificação de uma instrução com três operandos em registos.

instruções de `LOAD` e `STORE` necessitam de usar um outro formato, dado que especificam apenas um registo, mas necessitam de especificar um endereço:

Código	Reg	Ender
6	3	16

Figura 11.5: Codificação de instruções de `LOAD` e `STORE`.

A maior eficiência com que são codificadas as instruções aritméticas tem um custo elevado, uma vez que passa a existir a necessidade de carregar todos os operandos em registos (usando instruções de `LOAD`, ou similares) antes de

efectuar qualquer operação. Da mesma forma, quando é necessário guardar o resultado em memória tem de se executar explicitamente uma operação de `STORE`.

A tarefa de somar as três posições de memória e complementar o resultado que tem sido usada como exemplo seria implementada pelo Programa 11.4, num processador deste tipo:

```
LOAD   R1, M[N1]
LOAD   R2, M[N2]
LOAD   R3, M[N3]
ADD    R4, R1, R2
ADD    R4, R4, R3
NEG    R4, R4
STORE  M[N4], R4
```

Programa 11.4: Programa em *assembly* que soma três números em memória para um processador do tipo *load/store*.

Dado que as instruções de `LOAD` e `STORE` usam duas palavras de memória, enquanto que as operações aritméticas usam apenas uma, este código ocuparia 11 palavras de memória.

Como se pode depreender destes exemplos, a escolha da filosofia a adoptar no que respeita à codificação das instruções obedece a muitas restrições e compromissos, tendo tipicamente um grande impacto no desempenho final do processador. Na Secção 11.7 serão estudados em detalhe os formatos de codificação de instruções do processador P3.

11.5 Controlo da Sequência de Execução

Normalmente, os processadores executam sequencialmente instruções que se encontram em endereços consecutivos de memória. Porém, este fluxo de execução pode ser interrompido em duas condições: quando é executada uma instrução de controlo e quando é atendida uma interrupção.

11.5.1 Instruções de Salto

As instruções de controlo mais simples são os saltos incondicionais, normalmente designados em linguagem *assembly* por `JUMP` ou `BRANCH`. Estas instruções especificam qual o endereço da próxima instrução a ser executada, através da manipulação do valor do `PC`. Em linguagem *assembly*, este endereço é normalmente especificado através da utilização de um nome simbólico. Assim, por exemplo a sequência de instruções do processador P3 representada no Programa 11.5 representa um ciclo infinito que incrementa continuamente o valor do registo `R1`.

Em muitos casos, é necessário transferir o controlo apenas quando uma dada condição se verifica. Por exemplo, pode-se querer terminar um ciclo quando o valor de um contador guardado num dado registo atinge o valor 0. Nestes casos, utilizam-se instruções de salto condicional, que apenas transferem controlo para o endereço especificado quando uma dada condição se

```
Etiq1:  INC   R1
        BR    Etiq1
```

Programa 11.5: Exemplo de utilização do salto incondicional.

verifica. Por exemplo, no processador P3, a instrução `BR.Z Etiq` transfere controlo para a instrução com rótulo `Etiq` apenas se a última operação aritmética ou lógica executada teve como resultado o valor 0.

O troço de código representado no Programa 11.6 ilustra a utilização das instruções de salto condicional. Quando a instrução `DEC` é executada, o valor

```
        DEC   R1
        BR.Z  Etiq
        MOV   R1, 55AAh
Etiq:   ADD   R2,R1
```

Programa 11.6: Exemplo de utilização do salto condicional.

de `R1` é decrementado, ou seja, reduzido de uma unidade. Caso o valor que resulta desta operação seja igual a `0`, a instrução `BR.Z Etiq` transfere o controlo para a instrução com rótulo `Etiq`. Caso contrário, a instrução seguinte é executada. Neste caso, é a instrução que carrega o registo `R1` com o valor `55AAh`.

É comum fazer com que a condição de salto dependa da última operação efectuada, uma vez que isso evita que seja necessário especificar qual a localização do valor que deve ser testado. Assim, é necessário guardar num registo bits que mantenham o valor das condições que podem ser testadas por instruções de controlo.

Alguns processadores, porém, adoptam outra estratégia e permitem que a condição seja obtida por teste de um valor de um registo. Porém, mesmo neste caso, é útil guardar o resultado de condições que podem ser testadas em instruções posteriores. Este resultado é guardado num registo chamado *registo de estado* do processador. A cada condição corresponde, em princípio, um bit de estado. Entre as condições que podem ser testadas por instruções deste tipo encontram-se, tipicamente, as seguintes:

- Zero (`Z`): a condição é verdadeira se o resultado da última operação foi zero.
- Negativo (`N`): a condição é verdadeira se o resultado da última operação foi negativo.
- Transporte (`C`): esta condição (*carry*) é verdadeira se o resultado da última operação aritmética teve transporte.
- Excesso (`O`): esta condição (*overflow*) é verdadeira se o resultado da última operação excede a capacidade de representação do processador.
- Positivo (`P`): esta condição é verdadeira se o resultado da última operação foi estritamente positivo.

Considere-se, a título de exemplo, a sequência de instruções do Programa 11.7. A instrução `SUB R1,0004h` tem como resultado o valor -1, que é guardado

```
        MOV   R1, 0003h
        SUB   R1, 0004h
        BR.Z  Etiq1
        BR.N  Etiq2
Etiq1:  NOP
Etiq2:  NOP
```

Programa 11.7: Efeitos da instrução `SUB` nos valores dos bits de estado.

no registo `R1`.

Esta operação activa os bits de estado negativo, `N`, uma vez que o valor obtido é negativo, e o bit de transporte, `C`, uma vez que a subtracção é conseguida somando o complemento aritmético de 4. Ao mesmo tempo, coloca os bits de estado `Z`, `P` e `O` a 0, uma vez que o resultado não foi zero, não é positivo, nem resultou num valor não representável.

Desta forma, a instrução `BR.Z Etiq1` não vai transferir controlo para a instrução com rótulo `Etiq1`, uma vez que o bit de estado `Z` está inactivo. Já a instrução `BR.N Etiq2` vai transferir controlo para a instrução com rótulo `Etiq2`, dado que o bit de estado `N` está activo. Note-se que, de uma forma geral, só as instruções de manipulação de dados que envolvem a unidade lógica e aritmética (Secção 9.4) alteram os valores dos bits de estado.

Muitos processadores permitem também condições de salto que testam diversos bits de estado do processador, para além dos referidos acima. Pode, por exemplo, testar-se se o resultado de uma operação foi par, ou se uma dada condição interna ao processador se verifica.

Existem duas possibilidades para a especificação do endereço de salto. A primeira possibilidade é especificar o endereço de forma absoluta, dando directamente um endereço em memória que contém a próxima instrução a ser executada. A segunda possibilidade consiste em especificar este endereço de forma relativa, sendo que a constante usada representa um valor que deve ser adicionado ao `PC`. Quando esta possibilidade é utilizada, são muitas vezes usados menos bits que os necessários para descrever um endereço arbitrário em memória, uma vez que muitos saltos são para posições de memória relativamente próximas da instrução que está a ser executada.

A vantagem da primeira abordagem é permitir que qualquer endereço em memória seja especificado, independentemente da sua proximidade da instrução que está a ser executada. A segunda abordagem, além de permitir poupar espaço na codificação das instruções, permite que o programa seja *realojável*. Isto significa que o programa continua a funcionar mesmo que seja copiado para posições de memória diferentes daquela onde foi inicialmente localizado.

Na prática, muitos processadores suportam ambos os métodos de especificação, cabendo ao *assembler* escolher, de forma transparente para o programador, qual o tipo de salto que resulta da codificação de uma dada instrução.

No processador P3, existem os dois tipos de instruções de salto. Quando o salto é especificado com a instrução de `BR`, trata-se de uma salto relativo, e o valor especificado é adicionado ao conteúdo do `PC`. Quando o salto é espe-

cificado com a instrução de `JMP`, trata-se de um salto absoluto, sendo o valor especificado carregado directamente no registo `PC`. Muitas vezes os saltos relativos podem ser codificados usando menos bits, uma vez que muitos dos endereços de salto são próximos do endereço actual do `PC`.

Note-se que, do ponto de vista do programador, estas duas instruções são praticamente indistinguíveis, uma vez que o *assembler* tem a responsabilidade de codificar a instrução. Porém, em alguns casos particulares, pode existir interesse em utilizar um dos tipos de salto, especialmente nos casos em que haja interesse em realojar código máquina, sem utilizar o *assembler*.

11.5.2 Chamadas a Subrotinas

Um programa bem estruturado é tipicamente constituído por blocos de programa que desempenham uma tarefa bem definida e que são executadas repetidas vezes num programa. Em *assembly* esta estruturação de um programa em módulos é conseguida através do uso de subrotinas.

Uma subrotina é chamada através de uma instrução `CALL`, que, tal como faria uma instrução de `JMP`, transfere controlo para a instrução inicial da subrotina. Adicionalmente, porém, a instrução `CALL` causa o armazenamento do endereço de onde a subrotina foi chamada para que, quando esta terminar, seja possível continuar a execução com a instrução seguinte à instrução `CALL`. Uma subrotina é terminada com uma instrução de retorno, (`RET` ou `RETURN`), que transfere controlo para a instrução seguinte àquela que chamou a subrotina.

Embora diferentes processadores adoptem diferentes alternativas, uma solução muito comum é guardar o endereço de retorno no topo da pilha. Assim, a instrução de retorno tem simplesmente de repôr no contador de programa o valor guardado no topo da pilha para que seja retomada a sequência normal de execução. Outros processadores utilizam mecanismos mais complexos que permitem melhorar o desempenho, mas um estudo destas alternativas está fora do âmbito deste texto.

A Figura 11.6 ilustra a sequência de execução de instruções durante o processo de chamada e retorno de uma subrotina, tal como ela acontece no processador P3. Neste exemplo, a instrução `CALL Etiq1` transfere o controlo para a instrução cujo rótulo é `Etiq1`, carregando este valor no registo `PC`. O topo da pilha fica com o valor do endereço da instrução `ADD R1,R2`, que é a instrução que se segue à instrução de `CALL` e o valor do registo `SP` é decrementado para apontar para a próxima posição de memória[2]. Quando, após a execução de todas as instruções da subrotina, a instrução `RET` é finalmente executada, o endereço de retorno encontra-se no topo da pilha, pelo que basta carregar o valor do `PC` com este valor. Para um funcionamento correcto, é importante que a instrução `RET` encontre a pilha no mesmo estado em que ela se encontrava quando a subrotina foi chamada, para que o endereço de retorno seja correctamente recuperado. Isto significa que na execução da subrotina o número de instruções de `PUSH` e de `POP` tem de ser igual.

As subrotinas podem ter parâmetros, cujos valores são definidos pelo programa principal na altura da chamada da subrotina. A passagem de parâmetros para subrotinas pode ser feita de diversas formas. Uma forma comum de

[2]No processador P3, a pilha do processador é invertida, sendo o valor do apontador de pilha decrementado quando é executada uma instrução de `PUSH`

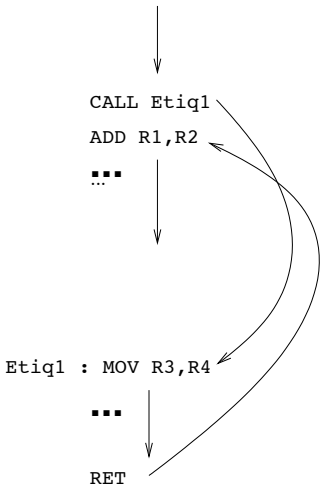


Figura 11.6: Ilustração do processo de chamada a uma subrotina.

passar parâmetros é através da pilha do processador. Neste caso, define-se a convenção de que se uma subrotina aceita um certo número de parâmetros, estes parâmetros se encontram no topo da pilha quando a rotina se inicia. Naturalmente, isto implica que o programa que chamou a subrotina coloque os parâmetros na pilha antes de a chamar. Em relação à passagem de parâmetros por registos, esta solução tem a vantagem de permitir subrotinas recursivas e de não limitar o número de parâmetros ao número de registos disponíveis. O modo de endereçamento baseado permite o acesso a valores que se encontram na pilha, mas não no topo da mesma. Este modo de endereçamento é especialmente útil para aceder directamente aos parâmetros de uma subrotina, sem que seja necessário executar explicitamente operações de `POP`.

11.5.3 Interrupções

Uma outra situação em que a sequência normal de execução das instruções é alterada acontece quando o processador recebe e atende um *pedido de interrupção*.

Em muitos sistemas, o processador tem de reagir a sinais vindos do exterior, que sinalizam a necessidade de efectuar algum processamento específico ou operações de entrada/saída. Por exemplo, um periférico pode querer sinalizar o processador que uma tecla foi premida, ou que uma palavra foi recebida num porto série.

Uma alternativa ao uso de interrupções é garantir que o processador amostra, a intervalos regulares, o valor de alguma linha, ou linhas, de entrada/saída. A esta amostragem regular chama-se, em inglês, *polling*. O uso de *polling* impõe uma sobrecarga grande ao processador, uma vez que é necessário garantir que um ou mais portos de entrada/saída são lidos a intervalos de tempo que não podem ser demasiado longos, conforme discussão na Secção 14.5.1.

Por esta razão, a esmagadora maioria dos processadores permite que um dispositivo exterior sinalize, de alguma forma, que a sequência normal de instruções deve ser interrompida e que o controlo deve ser transferido para uma

subrotina de atendimento. Em sistemas complexos, este dispositivo é geralmente um controlador de interrupções, que centraliza os pedidos provenientes de diversas fontes, os codifica, e os envia para o processador, de acordo com as suas prioridades.

Embora o mecanismo exacto através do qual as interrupções são sinalizadas, identificadas e atendidas varie de processador para processador, a seguinte descrição de alto nível aplica-se à maioria dos processadores comerciais e ao P3.

Em primeiro lugar, cada processador disponibiliza uma ou mais linhas de interrupção, que são activadas pelos dispositivos exteriores. O número de linhas não corresponde, porém, ao número total de origens de interrupções possíveis que é, em geral, muito superior. Isto é possível porque, após ter recebido indicação que a interrupção vai ser atendida, o dispositivo utiliza um dos barramentos exteriores para identificar qual a interrupção que foi activada. Desta forma é possível, por exemplo, usar apenas uma linha de interrupção e 8 linhas do barramento de dados para assinalar 256 tipos diferentes de interrupção.

Após receber a interrupção e o número que a identifica, o processador acede a uma tabela que indica qual o endereço da subrotina que deve ser usada para atender o pedido. O controlo de execução é então transferido para esta subrotina, após salvaguarda do conteúdo do contador do programa e de outra informação sobre o estado do processador. Esta informação inclui necessariamente o valor dos bits de estado do processador, mas pode também incluir o valor de outros registos internos cujo conteúdo possa ser destruído pela subrotina de interrupção. A salvaguarda desta informação é necessária para que a execução possa continuar, sem alterações, na instrução seguinte à que estava a ser executada quando a interrupção foi recebida.

Repare-se que, ao contrário das chamadas a subrotinas, as interrupções externas ocorrem em qualquer momento sem que o programador o possa prever. Assim, as interrupções podem surgir em qualquer instante, por exemplo entre a execução de uma instrução aritmética e uma instrução de salto condicional que testa o resultado dessa instrução.

No processador P3, como na maioria dos outros processadores, o programador pode actuar um bit de estado por forma a impedir que as interrupções sejam atendidas. Isto pode ser necessário em zonas de código onde a ocorrência de uma interrupção pudesse causar comportamentos indesejados.

Existem numerosas variações a este modo de operação e diversos detalhes de funcionamento que variam de processador para processador, mas o princípio geral de funcionamento é comum a todos eles. O mecanismo de interrupções do processador P3 será estudado em mais detalhe quando forem descritas as instruções que controlam o atendimento das mesmas.

Uma funcionalidade também existente em muitos processadores consiste em disponibilizar uma instrução (por exemplo, `INT` ou `TRAP`) que aceita um argumento (o número da interrupção) e que faz com que o processador se comporte exactamente como se tivesse recebido uma interrupção. Embora esta instrução possa, em princípio, ser substituída por uma instrução de chamada a subrotina, algumas diferenças nos detalhes de funcionamento são suficientes para justificar a sua existência como uma instrução separada. Este tipo de instruções é também útil na fase de *depuração* de um programa, em que as rotinas de interrupção podem ser usadas para analisar o valor de diversos registos e variáveis internas que podem não ser acessíveis de outra forma.

Em geral, as interrupções podem ter 3 origens distintas: podem ser externas, causadas pelo *hardware*; podem ser internas, despoletadas por uma instrução de `TRAP`; ou, em processadores mais complexos que o P3, podem ser causadas por excepções, como por exemplo um acesso incorrecto a memória ou a execução de uma divisão por zero.

11.6 Conjunto de Instruções do Processador P3

O processador P3 dispõe internamente de 16 registos, `R0` a `R15`, dos quais apenas 8 (`R0` a `R7`) podem ser usados directamente por instruções *assembly*. O processador P3 é um processador de 16 bits, o que significa que manipula, em cada instrução, dados de 16 bits.

Neste processador tanto os registos de dados como a memória também se encontram organizados em palavras de 16 bits, pelo que uma operação de transferência de dados manipula sempre um valor de 16 bits. Noutros processadores, o número de bits transferido em operações de manipulação de dados e acessos a memória nem sempre é igual ao número de bits dos registos. Por exemplo, no processador Intel 8088, os registos internos são de 16 bits, mas as transferências de memória são feitas em blocos de 8 bits, para simplificar a organização da memória exterior. No entanto, e de uma forma geral, quando um processador é designado como sendo de N bits, isto significa que tem capacidade para transferir e manipular dados em blocos de N bits.

No processador P3, o registo `R0` é um registo fictício, e contém sempre o valor `0`. Isto revela-se muito útil uma vez que a constante 0 é muitas vezes utilizada e um acesso a um registo é mais rápido que o acesso a uma constante guardada em memória. Os registos com números entre 8 e 15 são registos de uso especial cuja utilidade será estudada no capítulo seguinte. Estes registos desempenham funções específicas, não podendo ser manipulados directamente por instruções *assembly* genéricas. Dois destes registos são os registos `PC` e `SP` que podem ser usados em acessos a memória efectuados com os modos de endereçamento relativo e baseado. Para uso geral, o programador tem acesso aos registos `R1` a `R7`. Existe ainda um registo de estado que guarda os bits de estado do processador.

O processador P3 disponibiliza todas as instruções básicas de transferência de dados, manipulação e controlo que são comuns em processadores comerciais simples. Estas instruções podem classificar-se nas seguintes classes:

- Instruções aritméticas: aplicam um operador aritmético ao operando ou operandos. Exemplos de operações aritméticas de dois operandos são a soma, a subtracção e a multiplicação. Entre as operações aritméticas de um operando encontram-se as operações de incremento e decremento.

- Instruções lógicas: aplicam um operador lógico ao operando ou operandos. Exemplos de operações lógicas de dois operandos são a disjunção e conjunção, enquanto que o complemento Booleano é uma operação lógica de um só operando.

- Instruções de deslocamento: aplicam uma operação de deslocamento a um registo. Os deslocamentos podem ser à direita, à esquerda, circulares ou não, aritméticos ou lógicos.

- Instruções de controlo: controlam a sequência de instruções executada pelo processador, eventualmente com base no resultado de operações anteriores.
- Instruções de transferência: copiam ou movimentam dados de uma localização para outra.
- Instruções genéricas: agrupam-se nesta categoria um conjunto de instruções que executam diversas operações sobre o estado do processador, e que serão analisadas detalhadamente mais tarde.

A Tabela 11.4 contém as mnemónicas das instruções disponíveis no processador P3.

Aritméticas	Lógicas	Deslocamento	Controlo	Transferência	Genéricas
NEG	COM	SHR	BR	MOV	NOP
INC	AND	SHL	BR.cond	MVBH	ENI
DEC	OR	SHRA	JMP	MVBL	DSI
ADD	XOR	SHLA	JMP.cond	XCH	STC
ADDC	TEST	ROR	CALL	PUSH	CLC
SUB		ROL	CALL.cond	POP	CMC
SUBB		RORC	RET		
CMP		ROLC	RETN		
MUL			RTI		
DIV			INT		

Tabela 11.4: Conjunto de instruções do processador P3.

11.6.1 Instruções Aritméticas

O processador P3 disponibiliza as operações aritméticas descritas na Tabela 11.5. Todas as operações aritméticas disponíveis actuam sobre operandos

Instrução	Mnemónica	Exemplo
Complemento aritmético	NEG	NEG R1
Incrementar	INC	INC M[R2]
Decrementar	DEC	DEC M[R3+A5A5h]
Adicionar	ADD	ADD R3, M[R5+4]
Adicionar com transporte	ADDC	ADD R3, M[R6]
Subtrair	SUB	SUB R3, M[R5+4]
Subtrair com transporte	SUBB	SUBB R1, R2
Comparar	CMP	CMP R1,R2
Multiplicar	MUL	MUL R3,R4
Dividir	DIV	DIV R3,R4

Tabela 11.5: Instruções aritméticas do processador P3.

de 16 bits, que, quando representam números com sinal, são descritos em notação de complemento para dois. Em todas as operações, com excepção da

multiplicação e divisão, o resultado é também de 16 e é guardado no primeiro operando, destruindo o valor que se encontrava no mesmo. Por exemplo, a instrução `ADD R1,R2` executa a operação `R1` ← `R1 + R2`.

No caso da multiplicação, o resultado que tem, naturalmente, 32 bits, é guardado nas localizações que foram usadas para especificar os operandos, ficando a parte mais significativa guardada no primeiro operando. No caso da divisão, o resultado é guardado no primeiro operando enquanto que o resto da mesma é guardado no segundo operando. Esta opção de projecto leva a que as operações de multiplicação e divisão não possam ser usadas com operandos do tipo imediato. Por outro lado, o algoritmo utilizado para estas operações leva a que o seu resultado só faça sentido quando os operandos são números sem sinal. Pelas mesmas razões, os operandos não podem estar colocados fisicamente no mesmo local, o que significa que uma instrução `MUL R1,R1`, se utilizada, resultará num valor incorrecto.

Para simplificar as operações aritméticas com operandos de mais de 16 bits, operações de soma e subtracção com transporte estão também disponíveis. O bit de transporte, `C`, é um bit de estado gerado pela unidade lógica e aritmética, cujo valor é colocado a 1 quando existe um transporte numa operação aritmética ou de deslocamento.

Uma operação de adição com transporte, como por exemplo, `ADDC R1,R2` calcula o resultado de `R1+R2+C`, onde `C` é o valor guardado no bit de transporte. De forma idêntica, a subtracção com transporte, `SUBB R1,R2` calcula `R1-R2-C`.

Para exemplificar o funcionamento destas operações, considere-se um caso simplificado em que se pretendem somar dois números sem sinal, de 8 bits, mas utilizando apenas registos de 4 bits e utilizando uma unidade lógica e aritmética de 4 bits. Mais concretamente, suponha-se que se pretende somar o valor 00101111, guardado nos registos `R1` e `R2`, com o valor 00110011, guardado nos registos `R3` e `R4`.

Quando se soma o valor 1111 com o valor 0011, o resultado é 10010, que não cabe num registo de 4 bits. O bit mais significativo corresponde ao valor do bit de transporte, e é utilizado pela próxima instrução de adição ou subtracção com transporte. Isso significa que quando se somarem os valores 0010 com 0011 adicionando o bit de transporte se obtêm o valor 0110, sendo assim obtido o valor final correcto de 01100010.

Assim, no processador P3 a sequência de instruções do Programa 11.8 calcula correctamente uma soma de dois valores de 32 bits, sem sinal, guardados, respectivamente, nos registos `R1`, `R2` e `R3`, `R4`, ficando o valor resultante guardado no par de registos `R1` e `R2`.

```
ADD   R2, R4
ADDC  R1, R3
```

Programa 11.8: Soma de dois números de 32 bits.

Existem também instruções aritméticas unárias. A instrução `NEG` calcula o complemento aritmético do seu operando, enquanto que as instruções `INC` e `DEC` incrementam e decrementam, respectivamente, o seu operando.

Finalmente, a instrução de comparação `CMP` efectua uma subtracção, mas

sem guardar o resultado. É útil quando se pretende actualizar os bits de estado do processador sem alterar nenhum dos operandos envolvidos. Por exemplo, o troço de código descrito no Programa 11.9 determina e guarda em `R2` o endereço da primeira posição de memória que contém um valor igual ao do registo `R1`.

```
        MOV     R2, 0
Loop:   INC     R2
        CMP     R1, M[R2]
        BR.NZ   Loop
```

Programa 11.9: Determinação da primeira posição de memória que contém um valor igual ao registo `R1`.

Os operandos de todas as instruções, excepto a divisão e multiplicação, podem ser números inteiros sem sinal ou números com sinal em representação de complemento para 2. A interpretação dos resultados depende apenas do programador, sendo que o bit de estado `O` (excesso) só tem significado se se interpretarem os operandos como números com sinal.

Todas as instruções aritméticas alteram o valor dos bits de estado do processador, nomeadamente dos bits `Z`, `N`, `C` e `O`, que indicam, respectivamente, se o resultado foi zero, se foi negativo, se gerou transporte e se gerou um valor maior que o que é possível armazenar.

11.6.2 Instruções Lógicas

A Tabela 11.6 descreve as instruções lógicas do processador P3. As três primeiras instruções lógicas descritas nesta tabela aplicam aos seus operandos as

Instrução	Mnemónica	Exemplo
Conjunção	`AND`	`AND R1, M[R3]`
Disjunção	`OR`	`OR R1, 00FFh`
Disjunção exclusiva	`XOR`	`XOR M[R1], R2`
Complemento lógico	`COM`	`COM M[R2+4]`
Teste	`TEST`	`TEST R5, M[R4]`

Tabela 11.6: Instruções lógicas do processador P3.

operações de conjunção, disjunção e disjunção exclusiva, respectivamente. Estas operações são aplicadas bit a bit, sendo o resultado guardado na mesma posição do primeiro operando. A instrução lógica `COM` calcula o complemento, bit a bit, do seu único operando. Finalmente, a instrução `TEST` aplica o operador conjunção aos seus dois operandos, não guardando o seu resultado, mas alterando o valor dos bits de estado.

As operações lógicas alteram o valor dos bits de estado `Z` e `N`, mas deixam inalterados os bits de estado `C` e `O`. Com efeito, uma operação lógica nunca gera transporte nem um valor que não possa ser representado, pelo que estes bits são deixados inalterados.

11.6.3 Instruções de Deslocamento

As instruções de deslocamento disponíveis neste processador estão descritas na Tabela 11.7. O funcionamento das instruções de deslocamento foi explicado

Instrução	Mnemónica	Exemplo
Deslocamento lógico à direita	`SHR`	`SHR R1, 4`
Deslocamento lógico à esquerda	`SHL`	`SHL M[R1], 2`
Deslocamento aritmético à direita	`SHRA`	`SHRA M[R1], 2`
Deslocamento aritmético à esquerda	`SHLA`	`SHLA M[R2], 4`
Rotação para a direita	`ROR`	`ROR R4, 15`
Rotação para a esquerda	`ROL`	`ROL R4, 1`
Rotação para a direita, com transporte	`RORC`	`RORC R4, 15`
Rotação para a esquerda, com transporte	`ROLC`	`ROLC R2, 15`

Tabela 11.7: Instruções de deslocamento do processador P3.

na Secção 9.4.5, que deve ser consultada para uma descrição mais detalhada do funcionamento das mesmas.

Para todas estas instruções, um deslocamento à direita significa um deslocamento na direcção dos bits menos significativos. Estas instruções têm dois operandos. O primeiro é necessariamente um registo ou posição de memória que é o objecto do deslocamento, enquanto que o segundo é uma constante positiva que especifica o número de bits do deslocamento que deve ser aplicado ao primeiro operando. A constante pode tomar um valor entre 1 e 15. Em todas as operações de deslocamento, o bit de transporte `C` fica com o valor que sai do registo.

As instruções de `SHR` e `SHL` deslocam à direita e esquerda, respectivamente, o seu operando. Na operação `SHR` o bit mais significativo é preenchido com o valor 0. O mesmo acontece para o bit menos significativo na operação `SHL`.

O deslocamento aritmético à direita é semelhante ao deslocamento lógico, excepto no valor que é introduzido no bit mais significativo. No deslocamento aritmético, o valor do bit mais significativo após o deslocamento é igual ao seu valor antes do deslocamento. A diferença entre os deslocamentos lógicos e aritméticos é importante quando os mesmos são aplicados a números com sinal. Com efeito, um deslocamento aritmético à direita tem como efeito dividir por dois o número deslocado, quer o mesmo seja positivo ou negativo. Pelo contrário, o deslocamento lógico à direita não corresponde a uma divisão por dois quando é aplicado a um número negativo. Os deslocamentos à esquerda lógicos e aritméticos são equivalentes no que respeita ao resultado, mas alteram os bits de estado de forma diferente, uma vez que o primeiro é considerado uma operação lógica, alterando apenas os bits de estado `Z`, `N`, e `C` enquanto que o segundo, tal como o deslocamento aritmético à direita, é considerado uma operação aritmética, alterando todos os bits de estado.

As operações de rotação `ROR` e `ROL` representam deslocamentos circulares, onde os bits que são deslocados para fora do registo são re-injectados pelo outro extremo do registo. As operações de rotação com transporte aplicam uma operação de rotação ao conjunto do operando mais o bit de transporte. Assim, numa rotação à direita, o bit de transporte é injectado na parte alta do

registo, e o bit menos significativo passa para o bit de transporte. Numa rotação à esquerda passa-se o oposto. Estas operações afectam os bits de estado `Z`, `N` e `C`.

11.6.4 Instruções de Controlo

As instruções de controlo disponíveis neste processador são as descritas na Tabela 11.8.

Instrução	Mnemónica	Exemplo
Salto relativo incondicional	`BR`	`BR Pos1`
Salto relativo condicional	`BR.cond`	`BR.cond R3`
Salto incondicional absoluto	`JMP`	`JMP M[R3+1]`
Salto condicional absoluto	`JMP.cond`	`JMP.cond Rot1`
Chamada a subrotina	`CALL`	`CALL Rotina1`
Chamada condicional a subrotina	`CALL.cond`	`CALL.cond Rot2`
Retorno de subrotina	`RET`	`RET`
Retorno de subrotina com N parâmetros	`RETN`	`RETN 4`
Interrupção	`INT`	`INT 55`
Retorno de interrupção	`RTI`	`RTI`

Tabela 11.8: Instruções de controlo do processador P3.

A instrução de salto incondicional `JMP Pos` transfere o controlo da execução para a instrução guardada na posição `Pos`. Esta instrução efectivamente carrega o contador de programa com o valor especificado. Normalmente, é usado um nome simbólico para especificar o endereço de destino, embora possa ser usada qualquer constante numérica ou mesmo um operando especificado com qualquer um dos modos de endereçamento suportados pelo processador.

A instrução de salto condicional `JMP.cond Etiq` transfere controlo para a instrução especificada, mas apenas se se verificar a condição `cond`. É possível especificar qualquer uma das condições descritas na Tabela 11.9. Um teste de condição refere-se sempre ao resultado da última operação que afectou os bits de estado. Geralmente, esta é uma operação aritmética, lógica ou de deslocamento, embora também possa ser uma instrução de outro tipo, como por exemplo, `CMC`. Por exemplo, a instrução `BR.Z Dest` transfere controlo para a instrução no endereço `Dest` apenas se a última operação que alterou o registo de estado deu como resultado zero.

A condição `C` testa o resultado guardado no bit de transporte. A condição `N` testa se o último resultado foi negativo, o que em representação de complemento para 2 é equivalente a testar se o bit mais significativo do resultado é 1. A condição `P` testa se o resultado é estritamente positivo. Finalmente, um teste à condição `O` (excesso ou, em inglês, *overflow*) dá um resultado verdadeiro se a última operação aritmética deu um resultado que, interpretado como um número inteiro com sinal, não pôde ser adequadamente representado pelo processador.

Os bits que definem o valor das condições são guardados no registo de

Condição	Mnemónica	Descrição
Zero	`Z`	Última operação deu resultado zero
Não zero	`NZ`	Última operação deu resultado não zero
Transporte	`C`	Última operação gerou transporte
Não transporte	`NC`	Última operação não gerou transporte
Negativo	`N`	Última operação deu resultado negativo
Não negativo	`NN`	Última operação deu resultado não negativo
Excesso	`O`	Última operação gerou excesso (*overflow)*
Não excesso	`NO`	Última operação não gerou excesso (*overflow)*
Positivo	`P`	Última operação deu resultado positivo
Não positivo	`NP`	Última operação não deu resultado positivo
Interrupção	`I`	Existe uma interrupção pendente
Não interrupção	`NI`	Não existe interrupção pendente

Tabela 11.9: Condições de salto para o processador P3.

estado do processador, que, no entanto, não é directamente acessível ao programador.

As instruções de chamada a subrotina transferem controlo para a posição do programa especificada, mas apenas depois de guardarem na pilha o conteúdo do contador de programa. Isto permite que a instrução `RET` retorne o controlo para a instrução que se segue à instrução de chamada, copiando para o contador de programa o valor guardado no topo da pilha. Para que este mecanismo funcione correctamente, é importante que, dentro de cada rotina, o número de operações de inserção na pilha seja igual ao número de remoções.

É comum usar a pilha para passar parâmetros para uma rotina. Assim, existe também a instrução `RETN` que, além de repôr o valor do contador de programa, actualiza o apontador para a pilha por forma a que o número de valores especificado deixe de estar no topo da pilha do processador. Assim, por exemplo, a instrução `RETN 3` retira 3 valores da pilha do processador, e poderá ser usada para retornar de uma rotina que tem 3 parâmetros passados através da pilha. A instrução `RETN 0` é equivalente à instrução `RET`.

A instrução de `INT Intnum` comporta-se de forma semelhante a uma chamada a uma subrotina, uma vez que transfere controlo para a posição do programa associada à interrupção especificada com `Intnum`. A execução desta instrução tem o mesmo efeito que a recepção da interrupção `Intnum`. Como foi referido na Secção 11.5.3, isso é conseguido guardando na pilha não só o valor do contador de programa, mas também o registo de estado do processador. Isto permite que a instrução de retorno de interrupção (`RTI`) reponha o estado completo do processador[3], recuperando o valor do contador do programa e o registo de estado do processador. Esta instrução deve ser sempre e exclusivamente usada para efectuar o retorno de rotinas de interrupção, que podem ser chamadas quer através da instrução `INT` quer através do mecanismo de interrupções do processador.

As instruções de controlo não alteram o valor dos bits de estado do proces-

[3]No que respeita ao valor do contador do programa e do registo de estado. A rotina de interrupção poderá ter alterado o valor de outros registos.

sador, ao contrário do que acontece com as operações aritméticas, lógicas e de deslocamento estudadas nas secções anteriores.

11.6.5 Instruções de Transferência de Dados

Estas instruções permitem copiar palavras ou octetos entre posições de memória ou registos do processador. Também são consideradas instruções de transferência as instruções de manipulação da pilha do processador. O processador P3 dispõe das instruções de transferência de dados descritas na Tabela 11.10. A

Instrução	Mnemónica	Exemplo
Copiar o conteúdo	`MOV`	`MOV R1, M[R2]`
Copiar octeto menos significativo	`MVBL`	`MVBL M[Pos1], R3`
Copiar octeto mais significativo	`MVBH`	`MVBL R3, R4`
Trocar o conteúdo	`XCH`	`XCH R1, M[R2]`
Colocar na pilha	`PUSH`	`PUSH R1`
Remover da pilha	`POP`	`POP M[R5+4]`

Tabela 11.10: Instruções de transferência de dados do processador P3.

instrução mais básica de transferência de dados é a instrução `MOV POS1,POS2` que copia a palavra guardada em `POS2` para `POS1`. A instrução `MVBH POS1, POS2` copia o octeto mais significativo da posição `POS2` para o octeto mais significativo da posição `POS1`, deixando o octeto menos significativo inalterado. A instrução `MVBL` funciona de modo semelhante para o octeto menos significativo. A instrução de `XCH` troca os conteúdos das duas localizações especificadas.

Finalmente, as instruções de `PUSH` e `POP` são usadas para manipular a pilha. Mais especificamente, a instrução `PUSH Val` coloca na posição de memória apontada pelo registo `SP` o valor especificado, e, em seguida, decrementa o valor do registo `SP`. A instrução `POP Loc` começa por incrementar o valor do registo `SP`, e em seguida guarda em `Loc` (que pode ser um registo ou uma posição de memória) o valor da posição de memória apontada por `SP`.

As instruções de transferência de dados não alteram o valor dos bits de estado, uma vez que a sua principal funcionalidade é copiar dados de uma localização para outra. Caso seja necessário alterar o valor dos bits de estado de acordo com um valor manipulado por uma destas instruções é necessário executar uma instrução lógica ou aritmética que opere sobre o valor copiado. Tipicamente, esta instrução será a instrução `CMP` ou `TEST`.

11.6.6 Outras Instruções

O processador P3 disponibiliza ainda algumas instruções, descritas na Tabela 11.11, que manipulam diversos aspectos do funcionamento do processador. A instrução `ENI` dá ao processador ordem para aceitar interrupções a partir deste momento. A instrução `DSI` faz com que o processador deixe de aceitar interrupções. Ambas as instruções actuam modificando o valor de um bit do registo de estado, o bit `E`, que funciona como bit de controlo do sistema de interrupções.

Instrução	Mnemónica
Activar interrupções	`ENI`
Desactivar interrupções	`DSI`
Activar bit de transporte	`STC`
Desactivar bit de transporte	`CLC`
Complementar bit de transporte	`CMC`
Operação nula	`NOP`

Tabela 11.11: Outras instruções do processador P3.

Outras instruções que modificam directamente o valor de bits no registo de estado são as instruções de `STC`, `CLC` e `CMC` que, respectivamente, activam, desactivam e complementam o bit de transporte do processador.

Finalmente, a instrução `NOP` não executa qualquer operação nem altera o estado do processador. Pode ser usada para preencher temporariamente zonas de código que possam vir a ser alteradas mais tarde, mas não é geralmente utilizada excepto quando se pretenderem introduzir atrasos de curta duração em programas.

11.6.7 Exemplos de Utilização

Considere-se, a título de exemplo, que se pretende programar o processador P3 para somar 10 posições de memória consecutivas, com início na posição `Start`. O troço de código do Programa 11.10 executa essa operação.

```
        MOV    R1, 9
        MOV    R2, R0           ; Limpa o registo R2
Ciclo:  ADD    R2, M[Start+R1]
        DEC    R1
        BR.NN  Ciclo            ; Continua se R1>=0
```

Programa 11.10: Soma das 10 posições de memória com início na posição `Start`.

Inicializando o registo `R1` com o valor 9 e iterando até que este registo atinja um valor negativo, este código soma todas as posições de memória entre `M[Start]` até `M[Start+9]`, começando com as posições de endereço mais alto.

Em alternativa, seria possível inicializar `R1` com o valor 0 e iterar até que se atinja o valor 10, que já não deve ser adicionado. Neste caso, o código seria o representado no Programa 11.11. A instrução `CMP R1, 10` actualiza os registos de estado do processador da mesma maneira que a execução de uma subtracção. Assim, quando `R1` atingir o valor 10, o bit de estado `Z` irá ficar a 1 e o ciclo termina.

```
        MOV    R1, R0
        MOV    R2, R0           ; Limpa o registo R2
Ciclo:  ADD    R2, M[Start+R1]
        INC    R1
        CMP    R1, 10           ; Compara R1 com 10
        BR.NZ  Ciclo            ; Continua se R1-10 <> 0
```

Programa 11.11: Soma das 10 posições de memória com início na posição `Start`, incrementando o contador.

11.7 Formato das Instruções do Processador P3

As instruções do processador P3 são codificadas em uma ou duas palavras de memória. A segunda palavra de memória só é usada quando o modo de endereçamento requer a especificação do endereço de uma posição de memória ou de um operando imediato, nomeadamente nos modos de endereçamento imediato e indexado. Assim, todas as instruções que usem um destes modos de endereçamento usam duas posições de memória, a segunda das quais especifica o valor da palavra `W` usada no endereçamento.

A Figura 11.7 descreve o formato genérico de uma instrução do processador P3. Nesta figura, os campos marcados com um ponto de interrogação podem ou não estar presentes numa dada instrução. Os primeiros seis bits (bits 15 a 10)

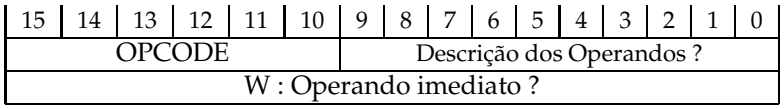


Figura 11.7: Formato genérico das instruções do processador P3.

da primeira palavra de cada instrução especificam qual o código da operação a executar (em inglês, *operation code*, ou mais simplesmente, *opcode*). A lista completa dos códigos de operação para o processador P3 encontra-se na Tabela 11.12. As instruções que não têm operandos nem parâmetros apenas usam o campo que especifica o código de operação. Nas instruções que têm um operando, existe, na descrição dos operandos, um campo M com dois bits, que controla o modo de endereçamento usado, de acordo com a Tabela 11.13.

Nas operações que têm dois operandos, o campo M especifica também o modo de endereçamento de um dos operandos, sendo o outro operando necessariamente um registo.

A descrição dos operandos inclui outros campos para além do campo M que especificam os operandos e os parâmetros de cada instrução, de uma forma que será detalhada em seguida. Esta especificação varia conforme o tipo de operação e é especificada pelo valor dos três primeiros bits do código de operação.

Mnemónica	Código	Mnemónica	Código
NOP	000000	CMP	100000
ENI	000001	ADD	100001
DSI	000010	ADDC	100010
STC	000011	SUB	100011
CLC	000100	SUBB	100100
CMC	000101	MUL	100101
RET	000110	DIV	100110
RTI	000111	TEST	100111
INT	001000	AND	101000
RETN	001001	OR	101001
NEG	010000	XOR	101010
INC	010001	MOV	101011
DEC	010010	MVBH	101100
COM	010011	MVBL	101101
PUSH	010100	XCH	101110
POP	010101	JMP	110000
SHR	011000	JMP.cond	110001
SHL	011001	CALL	110010
SHRA	011010	CALL.cond	110011
SHLA	011011	BR	111000
ROR	011100	BR.cond	111001
ROL	011101		
RORC	011110		
ROLC	011111		

Tabela 11.12: Códigos de operação do processador P3.

M	Endereçamento	Operação
00	Por registo	op = RX
01	Por registo indirecto	op = M[RX]
10	Imediato	op = W
11	Indexado, directo, relativo ou baseado	op = M[RX+W]

Tabela 11.13: Modos de endereçamento do processador P3.

11.7.1 Instruções sem Operandos

As instruções NOP, ENI, DSI, STC, CLC, CMC, RET e RTI não utilizam qualquer operando e são codificadas como se encontra ilustrado na Figura 11.8. As

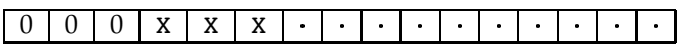


Figura 11.8: Codificação de instruções sem operandos.

posições definidas com X X X tem um valor diferente para cada uma destas instruções, de acordo com os valores descritos na Tabela 11.12. Para estas operações, os 10 bits menos significativos não são usados e o seu valor é ignorado.

As instruções `INT` e `RETN` aceitam um parâmetro como argumento, que é necessariamente um número inteiro, entre 0 e 1023. Este argumento é codificado no campo *ARG*, como se ilustra na Figura 11.9.

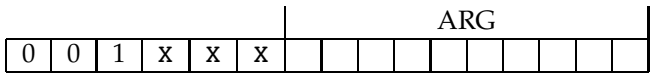


Figura 11.9: Codificação de instruções com um parâmetro.

11.7.2 Instruções com Um Operando

As instruções `NEG`, `INC`, `DEC`, `COM`, `PUSH` e `POP` aceitam um operando. Para as quatro primeira instruções, este operando especifica simultaneamente a origem dos dados aos quais vai ser aplicada a operação e o destino a dar ao resultado da mesma. Para a instrução `PUSH`, o operando determina o valor que será escrito na pilha. Para a instrução `POP`, o operando indica a localização onde deverá ser guardado o valor que se encontra no topo da pilha. Em qualquer dos casos, o operando pode ser especificado usando qualquer dos modos de endereçamento descritos na Secção 11.3.4. O modo de endereçamento é especificado pelos dois bits do campo M, de acordo com a Tabela 11.13 e o esquema da Figura 11.10. O valor do campo $IR1$, de 4 bits, é usado para especificar o va-

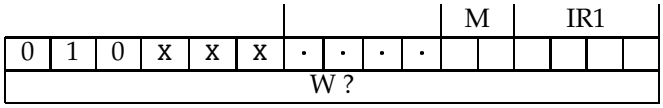


Figura 11.10: Codificação de instruções com um operando.

lor de `RX` (ver Tabela 11.13), usado de acordo com o modo de endereçamento. Isto permite codificar qualquer valor entre 0 e 7, para os registos de uso geral, e também dois outros valores para especificar os registos `PC` e `SP` nos modos de endereçamento relativo e baseado. Os quatro bits restantes não são utilizados.

As instruções `SHR`, `SHL`, `SHRA`, `SHLA`, `ROR`, `ROL`, `RORC`, `ROLC` aceitam, além do operando, um parâmetro que pode ser um número inteiro entre 1 e 15. Este segundo operando é codificado nos bits 6 a 9 da instrução, de acordo com a Figura 11.11.

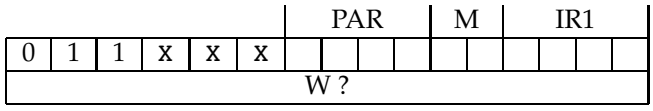


Figura 11.11: Codificação de instruções com um operando e um parâmetro.

11.7.3 Instruções com Dois Operandos

As instruções `CMP`, `ADD`, `ADDC`, `SUB`, `SUBB`, `MUL`, `DIV`, `TEST`, `AND`, `OR`, `XOR`, `MOV`, `MVBL`, `MVBH` e `XCH` usam dois operandos, sendo que o primeiro operando especifica simultaneamente um dos operandos fonte e a localização onde deverá ser guardado o resultado. Estas instruções são codificadas de acordo com o esquema da Figura 11.12. Uma vez que, como foi discutido na Secção 11.4,

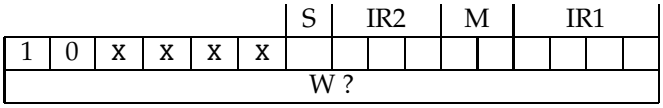


Figura 11.12: Codificação de instruções com dois operandos.

possibilitar a utilização de um modo arbitrário de endereçamento para ambos os operandos exigiria instruções demasiado longas, neste processador um dos operandos é, como foi referido anteriormente, especificado usando endereçamento por registo. Assim, o valor do campo M serve para especificar o modo de endereçamento de apenas um dos operandos. Se o valor de S for 0, o primeiro operando é endereçado com o registo $IR1$ de acordo com o modo de endereçamento especificado e $IR2$ é usado para aceder ao segundo operando. Se o valor de S for 1, $IR1$ especifica (de acordo com M) qual o segundo operando, e $IR2$ é usado para especificar o primeiro operando.

Em todos os casos, o registo ao qual se aplica o modo de endereçamento especificado pelo campo M é o registo especificado no campo $IR1$ do registo de instrução.

11.7.4 Instruções de Controlo

As instruções de controlo `JMP`, `JMP`.cond, `CALL` e `CALL.`cond são codificadas de acordo com o esquema da Figura 11.13.

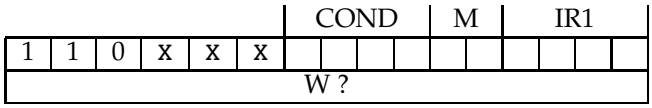


Figura 11.13: Codificação das instruções de salto absoluto

Os campos M e $IR1$ são interpretados da mesma forma do que nas instruções de um operando. O campo $COND$, quando necessário, é codificado de acordo com a Tabela 11.14.

As instruções de controlo `BR` e `BR`.cond são codificadas de acordo com o esquema da Figura 11.14. O valor do campo $OFFSET$ especifica um valor, relativo ao `PC`. Um valor de $OFFSET$ igual a 0 é equivalente à instrução de `NOP`.

Como discutido na Secção 11.5.1, esta instrução tem a vantagem de ocupar sempre apenas uma posição de memória. A limitação é que, como campo $OFFSET$ tem apenas 6 bits, os saltos relativos só são possíveis até 32 posições de memória atrás e 31 posições de memória à frente da posição actual do `PC`.

Condição	Mnemónica	Código
Zero	`Z`	0000
Não zero	`NZ`	0001
Transporte	`C`	0010
Não transporte	`NC`	0011
Negativo	`N`	0100
Não negativo	`NN`	0101
Excesso	`O`	0110
Não excesso	`NO`	0111
Positivo	`P`	1000
Não positivo	`NP`	1001
Interrupção	`I`	1010
Não interrupção	`NI`	1011

Tabela 11.14: Codificação das condições de teste.

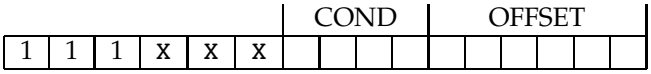


Figura 11.14: Codificação das instruções de salto relativo.

11.7.5 Exemplos de Codificação

A título de exemplo, considere-se a instrução `JMP.NZ R3` que salta para a posição de memória apontada por `R3` se o último resultado na unidade lógica e aritmética não foi 0. A codificação desta instrução será obtida notando que:

- O código de operação é 110001 (Tabela 11.12).
- O modo de endereçamento é por registo, o que significa que o valor do campo M é 00.
- O valor do campo $IR1$ é 0011, por forma a especificar o registo `R3`.
- O valor do campo `COND` é 0001, de acordo com a Tabela 11.14.

Isto conduz a que a codificação desta instrução seja conseguida com uma só palavra que é obtida por concatenação destes valores, ou seja, 1100010001000011 o que é equivalente a `C443h`.

Considere-se agora a instrução `ADD R1,M[R7+00A0h]`, ligeiramente mais complexa, que tem dois operandos e usa um modo de endereçamento diferente. A codificação desta instrução é obtida considerando que:

- O código de operação para a instrução `ADD` é 100001. Note-se que o valor dos dois bits mais significativos é 10, o que indica que é uma instrução de dois operandos.
- Esta instrução tem dois operandos, dos quais o primeiro é endereçado por registo (`R1`) e o segundo é obtido através do uso de endereçamento indexado a partir dos valores do registo `R7` e da constante `00A0h`. Isto significa que:

  - O valor do campo M é 11, para indicar modo de endereçamento indexado.
  - O valor do campo S é 1, para indicar que o modo de endereçamento se aplica ao segundo operando.
  - O valor do campo $IR1$, que, neste caso, é usado na especificação do segundo operando, é 0111, uma vez que o registo utilizado é `R7`.
  - O valor do campo $IR2$ é 001, especificando o registo usado na definição do primeiro operando.

- O valor do campo W, na segunda palavra da instrução, será `00A0h`.

Esta instrução é assim codificada com duas palavras de memória, que contém os valores 1000011001110111 e 0000000010100000, ou seja, `8677h` e `00A0h`. Para outros exemplos de codificação, pode consultar-se o exemplo da Secção 11.1, que contém o código em linguagem máquina do P3 e o correspondente código em *assembly*.

11.8 Um *Assembler* para o Processador P3

Embora exista uma correspondência directa entre uma instrução *assembly* e uma instrução de linguagem máquina, a tradução manual de um programa em *assembly* para a sequência de bits que constituem a linguagem máquina é um processo penoso e sujeito a erros.

Por esta razão, é geralmente utilizada uma ferramenta, o *assembler*, que traduz um programa de linguagem *assembly* para linguagem máquina. Além de traduzir as mnemónicas das instruções e os valores dos argumentos e dos operandos, o *assembler* permite que o programador utilize nomes simbólicos em vez de constantes, o que simplifica a tarefa de programar e torna mais legíveis os programas.

A primeira e talvez a mais importante característica de um *assembler* é o facto de permitir ao programador usar mnemónicas para as instruções e nomes simbólicos para os endereços das mesmas. O uso de mnemónicas é de óbvia vantagem para o programador, já que evita a memorização dos códigos de operação. Também o uso de nomes simbólicos para os endereços simplifica consideravelmente a tarefa do programador que, na especificação de uma instrução de controlo, pode usar um rótulo que é o nome simbólico do endereço da instrução para onde pretende transferir controlo. Caso não existisse esta possibilidade, teria de se usar o endereço real da instrução que, por vezes, não é ainda conhecido e está, em todo o caso, sujeito a ser alterado no futuro, quando outras partes do programa forem modificadas.

Considere-se, por exemplo, o troço de código em *assembly* descrito no Programa 11.12.

Este troço de código tem como objectivo identificar a primeira posição de memória cujo conteúdo é igual ao conteúdo do registo `R1`, começando na posição `start=0F00h`.

Neste troço de código foi usada a directiva `ORIG` para definir o endereço de memória onde irá iniciar-se a escrita em memória do código máquina do programa e a directiva `EQU` que define a constante `start` com o valor `0F00h`.

```
              ORIG  0A00h
start         EQU   0F00h

; Procura:   Localiza a primeira posição de memória
;            que contém o valor do registo R1
; Entradas:  R1 - Valor a procurar
; Saídas:    R2 - Endereço do resultado

Procura:      MOV   R2, start  ; Inicializa R2
CProcura:     CMP   R1, M[R2]  ; Compara os valores
              BR.Z  FimProc    ; Termina se encontrou
              INC   R2         ; Incrementa o ponteiro
              BR    CProcura   ; Próxima iteração
FimProc:      RET              ; R2 contém o resultado
```

Programa 11.12: Programa em *Assembly* que localiza a primeira posição de memória cujo conteúdo é igual ao do registo `R1`.

Esta constante pode ser usada mais tarde em qualquer posição onde for necessário usar o valor `0F00h`. As definições de constantes devem vir no início do programa, para que possam ser facilmente alteradas. Os nomes das constantes permitem tornar a leitura do código mais fácil por serem mais inteligíveis que um valor numérico. Para além disso, se se pretender alterar o valor da constante basta alterar a definição e todo o código fica automaticamente actualizado. Por esta razão, não devem existir constantes numéricas dispersas pelo programa, devendo sempre ser usadas definições das mesmas no princípio do programa ou módulo respectivo.

Foram também usados comentários para documentar o programa. Um comentário começa pelo caracter `;`, que indica ao *assembler* que todo o texto que se segue nessa linha deverá ser ignorado no processo de tradução do código *assembly*.

Note-se que as directivas `ORIG` e `EQU` não são instruções *assembly*. Uma análise da Tabela 11.15, que contém o código máquina resultante deste código *assembly*, revela que não foram geradas nenhumas instruções que correspondam a estas directivas. O resultado das directivas só é visível quando se analisa a primeira instrução máquina gerada (Tabela 11.15), verificando-se que esta instrução é equivalente à instrução *assembly* `MOV R2, 0F00h` e se localiza na posição de memória `0A00h`. De uma forma geral, podem ser usadas constantes tanto como operandos de instruções como para especificar endereços de memória. Em segundo lugar, repare-se que foram usados nomes simbólicos para três endereços de instruções, os rótulos `Procura`, `CProcura` e `FimProc`. Dois destes rótulos foram utilizados no programa, nas instruções com mnemónica `BR.Z` e `BR`. Desta forma, o programador não tem de lidar com o valor numérico dos endereços das instruções para onde pretende transferir controlo. O rótulo `Procura`, por outro lado, não é usado dentro da subrotina, mas poderá ser usado mais tarde por outra subrotina que pretenda usar esta. A chamada à subrotina `Procura` poderá ser efectuada através da instrução `CALL Procura`, não necessitando o programador de conhecer o endereço onde esta rotina irá

Endereço	Instrução		Mnemónica
0A00h	1010111010100000b	AEA0h	MOV R2, start
0A01h	0000111100000000b	0F00h	
0A02h	1000001001010010b	8252h	CMP R1,M[R2]
0A03h	1110010000000010b	E402h	BR.Z FimProc
0A04h	0100010000000010b	4402h	INC R2
0A05h	1110000000111100b	E03Ch	BR CProcura
0A06h	0001100000000000b	1800h	RET

Tabela 11.15: Linguagem máquina da subrotina `Procura`.

ficar localizada.

O *assembler* do processador P3 aceita várias directivas, descritas na Tabela 11.16. Além das directivas `ORIG` e `EQU`, já usadas, o *assembler* do processador

`ORIG`	Especifica o endereço de origem do código que se segue
`EQU`	Define o valor de uma constante
`WORD`	Reserva uma posição de memória para uma variável
`STR`	Guarda uma cadeia de caracteres em memória
`TAB`	Reserva posições de memória

Tabela 11.16: Directivas para o *assembler* do processador P3.

P3 aceita três outras directivas, cujo uso está exemplificado no Programa 11.13.

```
Abc      WORD  0055h
Xyz      WORD  0011h
Texto1   STR   'Hel','lo'
Tabela1  TAB   3
```

Programa 11.13: Exemplo de uso das directivas.

A directiva `WORD` reserva uma posição de memória para conter uma variável, que pode mais tarde ser referenciada usando um nome simbólico. Permite ainda inicializar essa posição de memória. No exemplo acima, reserva uma posição de memória para a variável `Abc`, que é inicializada a `0055h` e uma posição de memória para a variável `Xyz`, inicializada a `0011h`.

A directiva `STR` permite guardar uma cadeia de caracteres em memória. No caso acima, são reservadas 5 posições de memória, que são preenchidas com os valores ASCII da cadeia de caracteres *Hello* e o nome simbólico `Texto1` fica definido com o valor do endereço da posição de memória onde começa a cadeia. O terceiro argumento desta directiva consiste numa lista de elementos, separada por vírgulas, onde cada elemento pode ser uma cadeia de caracteres, começada e terminada pelo símbolo `'` ou uma constante numérica. As cadeias de caracteres são substituídas pelos seus valores ASCII, concatenadas umas com as outras (ou com argumentos numéricos, caso existam) e o resultado é guardado em memória.

Finalmente, a directiva `TAB` reserva posições de memória, inicializadas a 0, que poderão ser usadas para guardar uma tabela, cujo princípio pode ser referenciado com um nome simbólico. No exemplo acima, são guardadas três posições de memória e definido o nome simbólico `Tabela1`.

Se se assumir que as directivas acima apareciam no princípio do programa, antecedidas de uma directiva `ORIG 0000h`, as primeiras 10 posições de memória ficariam preenchida de acordo com a Tabela 11.17. Ficariam ainda definidas as constantes `Abc`, `Xyz`, `Texto1` e `Tabela1` com os valores `0000h`, `0001h`, `0002h` e `0007h`, respectivamente, constantes estas que podem ser utilizadas noutro contexto para referenciar os respectivos endereços.

Endereço	Instrução
0000h	0055h
0001h	0011h
0002h	0048h
0003h	0065h
0004h	006Ch
0005h	006Ch
0006h	006Fh
0007h	0000h
0008h	0000h
0009h	0000h

Tabela 11.17: Conteúdo das posições de memória, de acordo com as directivas do Programa 11.13.

11.9 Programação em Linguagem *Assembly*

Como em qualquer linguagem de programação, o uso de técnicas de programação estruturada é fundamental para que seja possível levar a bom termo um projecto de programação. No entanto, o uso de uma estrutura modular num programa em *assembly* é talvez ainda mais importante do que em linguagens de alto nível, por duas razões. Em primeiro lugar, quando se usa linguagem *assembly* são geralmente necessárias mais instruções para realizar uma operação do que numa linguagem de alto nível, o que torna os programas consideravelmente mais extensos. Em segundo lugar, a legibilidade de um programa escrito em *assembly* é consideravelmente menor, dada as maiores restrições impostas ao formato do programa e ao uso de nomes de variáveis.

É assim importante fugir à tentação de começar a programar imediatamente, sem passar por uma fase de definição da estrutura do programa. Esta definição, que pode ser feita quer em termos de fluxograma quer em termos de pseudo-código, é a parte mais importante de qualquer projecto de programação em *assembly* e deve ser aquela a que se dedica mais tempo.

Se este tempo não for usado aqui, a fase de programação detalhada será, com elevada probabilidade, muito mais extensa, acabando o projecto por demorar mais tempo na sua totalidade. A fase de definição da estrutura do programa permitir definir e perceber bem o problema e dividi-lo em problemas

mais simples, sem a preocupação de conhecer detalhes relacionados com a linguagem de programação.

11.9.1 Programação Estruturada em *Assembly*

O uso de subrotinas está directamente relacionado com a estruturação do programa. A cada bloco definido na fase de estruturação do programa corresponderá uma ou mais subrotinas. Uma subrotina deverá ter bem definido:

- A sua funcionalidade
- Os parâmetros de entrada e saída
- Registos e posições de memória alterados na subrotina

Com uma boa estruturação do trabalho, cada subrotina pode e deve ser desenvolvida e testada independentemente do resto do código. Isto permitirá que a construção do programa final, obtido através da ligação de subrotinas, seja feita com relativa simplicidade e rapidez uma vez que a maioria dos erros de programação já foram detectados na fase de teste de cada módulo. Pelo contrário, é sempre de evitar o procedimento que consiste em juntar várias subrotinas ainda não testadas, com o objectivo de depurar o programa na sua totalidade. Verifica-se que os erros são extremamente difíceis de identificar e resolver nesta fase, mesmo que sejam relativamente simples quando analisados ao nível de uma subrotina.

Dada a relativa ilegibilidade de um programa em *assembly*, o uso de comentários que documentem o funcionamento das subrotinas e de partes importantes das mesmas é fundamental.

Cada subrotina deverá, assim, ter um cabeçalho que documente devidamente cada um dos três pontos identificados acima. Além disso, partes complexas ou menos óbvias de cada subrotina deverão ser comentadas independentemente.

A utilização de constantes numéricas embebidas no código dificulta a manutenção e alteração posterior do mesmo, além de ser uma fonte de possíveis erros que são difíceis de identificar. A metodologia que deverá ser adoptada é a definição de todos os valores de constantes numéricas no início do programa utilizando a directiva `EQU`.

Embora seja possível usar os registos ou posições de memória para passar parâmetros para subrotinas, esta solução não permite a utilização de subrotinas recursivas. Por essa razão, em muitos casos é utilizada a pilha para efectuar a passagem de parâmetros. Quando se utiliza esta abordagem, os parâmetros deverão ser colocados na pilha e, dentro da rotina, deverão ser acedidos através de acessos à pilha. O tipo de parâmetros aceites e a ordem pela qual eles devem ser passados deve ser documentados no cabeçalho da subrotina.

11.9.2 Exemplo de Programação em *Assembly*

Para ilustrar os conceitos descritos acima, projecta-se em seguida um programa em *assembly* para um problema específico. O programa a desenvolver deverá copiar um texto de um porto de entrada e escrevê-lo num porto de saída com a primeira letra de todas as palavras em maiúsculas. Neste problema, o único

separador de palavras é o espaço e o fim do texto é indicado por um caracter especial, `@`.

A primeira fase consiste em definir a estrutura global do programa. Numa primeira abordagem, o programa pode ser dividido em três blocos, que correspondem aos blocos do fluxograma na Figura 11.15. Este fluxograma, embora

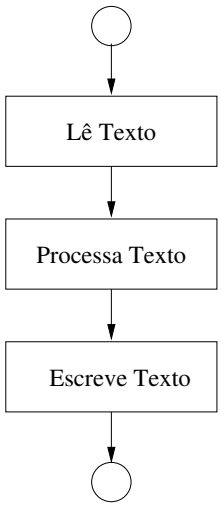


Figura 11.15: Fluxograma para o programa principal.

simples, define que, numa primeira fase o texto é lido do porto de entrada e é escrito em memória; na segunda fase, o texto é processado em memória; e na terceira fase, o texto modificado em memória é escrito para o porto de saída. Nesta fase, deve também ser definido que a subrotina que lê o texto retorna o número de caracteres lidos, para uso dos módulos seguintes. Os três passos indicados na figura ainda não são tão simples que a sua realização seja óbvia, pelo que cada um desses blocos deve agora ser refinado.

O bloco que lê o texto pode ser realizado de acordo com o fluxograma da Figura 11.16. Esta subrotina lê o texto, caracter a caracter, e escreve-o em memória, testando cada caracter para ver se é o caracter que indica o fim do texto e verificando se se atingiu o tamanho máximo permitido para o texto[4]. Com este nível de detalhe já é fácil transpor o fluxograma para linguagem *assembly* e criar a subrotina correspondente, descrita no Programa 11.14. Uma vez que o programa em *assembly* é geralmente escrito usando um editor de ficheiros que não tem suporte especial para a programação neste ambiente, é conveniente utilizar uma formatação que melhore a legibilidade do programa. Em geral, código *assembly* é escrito em quatro colunas: rótulos, mnemónica da instrução *assembly*, operandos e comentários.

Note-se que no programa não aparecem constantes numéricas, mas sim constantes que serão definidas no cabeçalho do programa. Neste caso, definiram-se: `fim_txt`, código do caracter que indica o fim de texto; `Texto`, primeira posição de memória reservada para o texto; e `max_car`, número de

[4]Nesta solução simples para o problema não é devolvida qualquer indicação se a leitura terminou por ter sido recebido o carácter `@` ou por ter sido atingido o número máximo de caracteres possível.

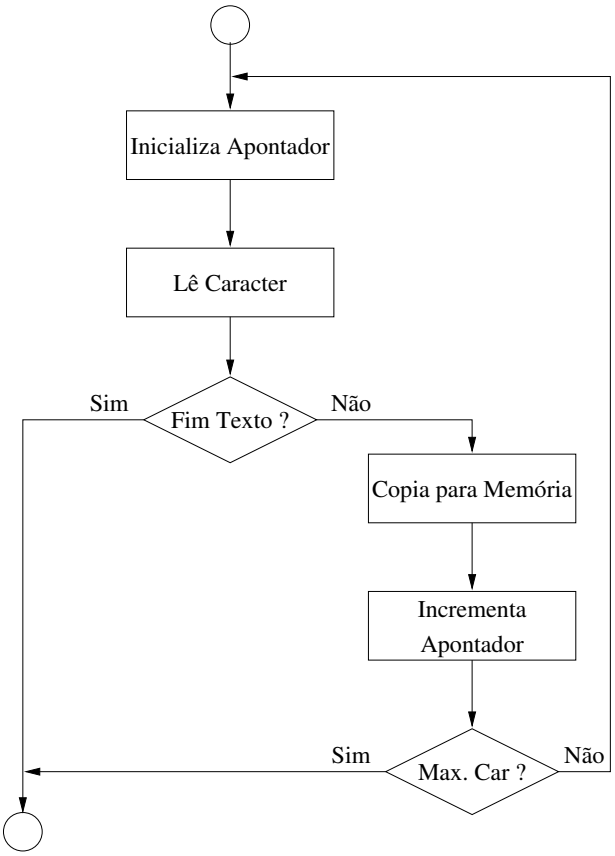


Figura 11.16: Fluxograma da subrotina de leitura de caracteres.

```
; LeTexto:   efectua a leitura caracter a caracter
;            e preenche a zona Texto
; Entradas:
; Saídas:    R2 - numero de caracteres lidos
; Efeitos:   altera o registo R1
LeTexto:     MOV    R2, R0           ; inicializa R2
CicloLeit:   CALL   LeCaracter       ; lê um caracter
             CMP    R1, fim_texto
             BR.Z   FimLeTexto
             MOV    M[R2+Texto], R1
             INC    R2
             CMP    R2, max_car
             BR.NZ  CicloLeit
FimLeTexto:  RET
```

Programa 11.14: Subrotina de leitura de texto *assembly*.

caracteres máximo para o texto. A leitura de um caracter é feita pela subrotina do Programa 11.18. Esta subrotina espera que exista um caracter no porto de

entrada, até o retornar. Assumiu-se aqui que uma leitura do porto de controlo devolve 0 caso nenhum caracter tenha sido introduzido desde a última leitura do porto de entrada que corresponde ao teclado. É necessário agora especi-

```
; LeCaracter:  efectua a leitura de um caracter
; Entradas:
; Saídas:      R1 - caracter lido
; Efeitos:     altera o registo R1
LeCaracter:    CMP   R0, M[controlo]
               BR.Z  LeCaracter       ; Ciclo de espera
               MOV   R1, M[in_port]   ; Lê o caracter
               RET
```

Tabela 11.18: Subrotina de leitura de um caracter.

ficar o bloco que processa o texto. Um possível fluxograma para esse bloco é o da Figura 11.17. Neste bloco avança-se caracter a caracter, em memória,

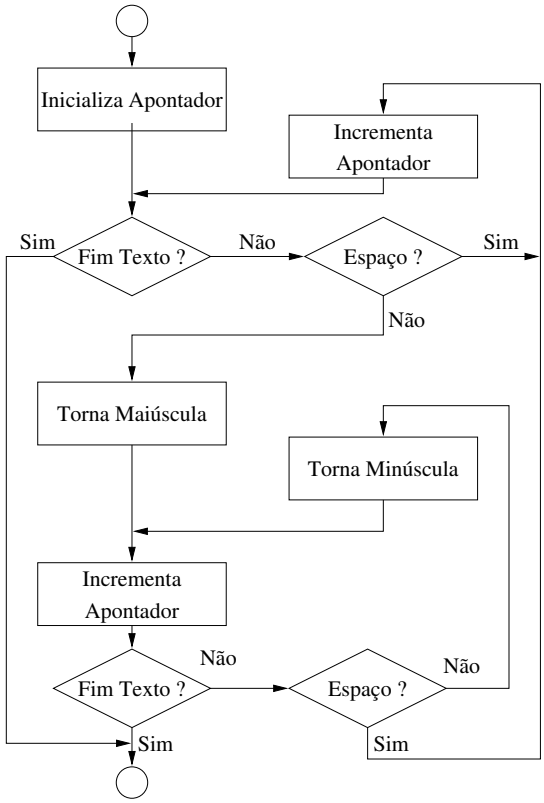


Figura 11.17: Fluxograma do bloco de processamento de texto.

convertendo cada letra a seguir a um espaço numa letra maiúscula até que se tenha processado o número de caracteres lidos. A conversão de minúsculas para maiúsculas pode ser efectuada aplicando a operação `AND`, que usa o facto

de, no código ASCII, os códigos para a minúscula e maiúscula de uma dada letra diferirem apenas no sexto bit, que é, respectivamente, 1 para minúscula e 0 para maiúscula. A operação inversa é conseguida com a operação `OR`. A alteração do valor do sexto bit é efectuada com a ajuda de uma máscara, que tem apenas esse bit a `1`.

Também aqui se chegou a uma representação suficientemente simples para que seja possível escrever o código *assembly* para esta subrotina, descrito no Programa 11.15. Note-se que existe uma correspondência entre o fluxograma e

```
; ProcTexto:   Converte em maiúscula o primeiro caracter
               de cada palavra
; Entradas:    R2 - numero de caracteres
; Saídas:
; Efeitos:     modifica o texto em memória;
               altera registos R1, R3 e R4
ProcTexto:     MOV   R3, R0           ; inicializa R3
               MOV   R4, mascara      ; Complemento da
               COM   R4               ; máscara
CicloExt:      CMP   R3, R2
               BR.Z  FimProcTexto
               MOV   R1, M[R3+Texto]
               CMP   R1,' '
               BR.Z  Incr2
               AND   R1, R4           ; torna maiúscula
               MOV   M[R3+Texto], R1
Incr1:         INC   R3
               CMP   R3, R2
               BR.Z  FimProcTexto     ; Fim de texto
               MOV   R1, M[R3+Texto]
               CMP   R1,' '
               BR.Z  Incr2
               OR    R1, mascara      ; torna minúscula
               MOV   M[R3+Texto], R1
               BR    Incr1
Incr2:         INC   R3               ; próxima palavra
               BR    CicloExt
FimProcTexto:  RET
```

Programa 11.15: Subrotina de processamento do texto.

o código.

Finalmente, falta especificar o bloco de impressão do texto. Um possível fluxograma para este bloco é o da Figura 11.18. Dada a simplicidade deste bloco, é possível fazer uma tradução imediata para linguagem *assembly*, resultando no código do Programa 11.16.

O programa principal, descrito no Programa 11.17, corresponde ao fluxograma da Figura 11.15, será simplesmente uma sequência de chamadas às subrotinas já definidas, antecedido das directivas necessárias, que definem a origem do programa e os valores das constantes e variáveis usadas no programa

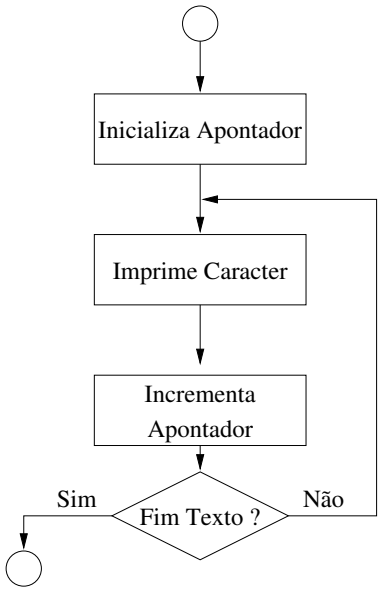


Figura 11.18: Fluxograma do bloco de escrita de texto.

```
; Imprime:   escreve o texto no porto de saída.
; Entradas:  R1 - início do texto
;            R2 - número de caracteres no texto
; Saídas:
; Efeitos:   altera os registos R3 e R4

Imprime:    MOV    R3, R0           ; inicializa R3
CicloImp:   MOV    R4, M[R1]
            MOV    M[out_port], R4  ; escreve caracter
            INC    R1
            INC    R3
            CMP    R3, R2           ; verifica terminou
            BR.NZ  CicloImp
            RET
```

Programa 11.16: Subrotina de escrita do texto modificado.

e subrotinas.

Sumário

Neste capítulo estudaram-se algumas das possíveis alternativas para a arquitectura do conjunto de instruções de micro-processadores e apresentou-se o conjunto de instruções do processador didáctico P3. Este conjunto de instruções, típico de processadores simples de segunda geração, inclui instruções de controlo, transferência e manipulação de dados. Foi também descrita a lin-

```
; Definição de constantes
fim_texto  EQU   '@'
controlo   EQU   FFFDh
max_car    EQU   100h
in_port    EQU   FFFFh
out_port   EQU   FFFEh
mascara    EQU   0020h

; Definição de variáveis
           ORIG  8000h
Texto      TAB   max_car

           ORIG  0000h
Inicio:    MOV   R1, F000h
           MOV   SP, R1     ; Inicializa a pilha
           CALL  LeTexto
           CALL  ProcTexto
           MOV   R1, Texto  ; Parâmetro para Imprime
           CALL  Imprime
           BR    Inicio
```

Programa 11.17: Programa principal.

guagem *assembly* do processador P3 que permite efectuar a sua programação utilizando um conjunto de mnemónicas em vez de linguagem máquina.

Foi também estudada a forma como a linguagem *assembly* é traduzida para linguagem máquina, tendo sido descrita a forma como cada instrução é codificada.

Finalmente, foi apresentado um conjunto de regras e recomendações que deverão ser adoptadas sempre que se desenvolvam programas em linguagem *assembly*, uma vez que tornam o processo de desenvolvimento mais simples e isento de erros.

Capítulo 12

Estrutura Interna de um Processador

No capítulo anterior definiu-se a arquitectura do conjunto de instruções do processador P3. A arquitectura do conjunto de instruções define a interface visível para o programador, especificando as instruções que estão disponíveis, quais os registos internos, os modos de acesso à memória e outras características relevantes do processador.

Para uma dada arquitectura do conjunto de instruções, existem muitas realizações possíveis para a estrutura do processador. As numerosas realizações possíveis resultam do grande número de escolhas que estão disponíveis ao projectista de sistemas digitais e implicam diferentes compromissos entre o número de ciclos de relógio necessários para executar cada instrução, a frequência máxima de relógio possível para o sistema e a área ocupada em silício para a realização física do processador.

Este capítulo descreve em detalhe uma realização particular do processador P3 e descreve os diversos compromissos que são inerentes às escolhas tomadas, focando, quando tal é julgado oportuno, as alternativas mais óbvias e as razões pelas quais as mesmas foram preteridas em detrimento da implementação descrita.

Como qualquer circuito complexo, é útil analisar o processador P3 considerando separadamente duas componentes principais, tal como foi estudado no Capítulo 9: o circuito de dados e o circuito de controlo.

No circuito de dados encontra-se toda a lógica regular que é usada para armazenar e processar dados do utilizador, lógica esta que opera, de uma forma geral, sobre conjuntos de dados organizados em octetos e palavras. Fazem parte do circuito de dados o banco de registos, a unidade lógica e aritmética, os circuitos de acesso a memória e portos de entrada/saída e ainda os barramentos de interligação internos.

O circuito de controlo gera os sinais que controlam o circuito de dados, por forma a que este execute a sequência de operações que são necessárias para carregar e executar cada instrução *assembly*, tendo em conta o estado do circuito de dados.

12.1 Circuito de Dados

O circuito de dados do processador P3, esquematizado na Figura 12.1, tem cinco componentes principais: o banco de registos, a unidade lógica e aritmética, o *registo de instrução* (`RI`), o registo de estado (`RE`), e, por último, os circuitos de interligação e multiplexagem de dados.

O banco de registos, cuja estrutura interna foi descrita na Secção 7.5.4, contém 16 registos, `R0` a `R15` de 16 bits cada e é acedido através de dois portos de leitura (portos A e B) e um porto de escrita (porto D). Dois destes registos são registos de uso especial, o contador de programa `PC` e o apontador para a pilha `SP`.

A unidade lógica e aritmética, descrita em detalhe na Secção 9.4, é utilizada para realizar todas as operações lógicas e aritméticas sobre os operandos que são fornecidos pelo banco de registos.

O registo de instrução, `RI`, é um registo de uso especial que não está integrado no banco de registos. Este registo não precisa de ser acedido directamente pelo circuito de dados. No entanto, todos os seus bits são usados pela unidade de controlo.

O registo de estado, `RE`, agrupa os diversos bits de estado do processador, ligados ao circuito de dados através de dois barramentos de 5 bits que permitem ler e escrever este registo.

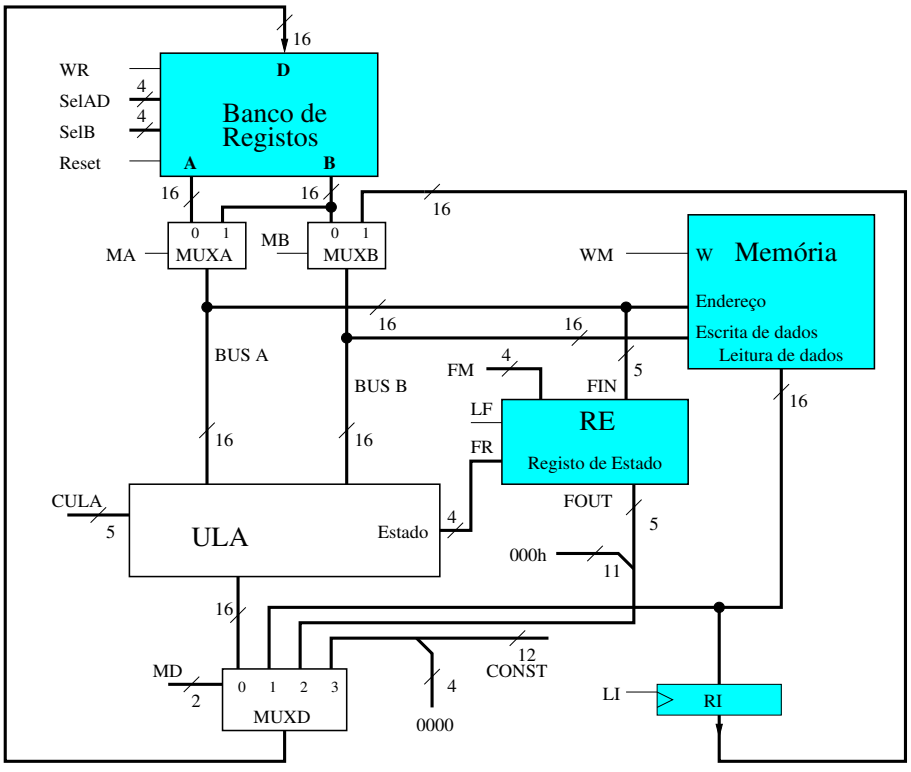


Figura 12.1: Circuito de dados do processador P3.

Os circuitos de acesso à memória são constituídos pelo barramento de en-

dereços e por dois barramentos de dados, um para escrita, outro para leitura. O barramento de endereços está ligado ao porto A do banco de registos, enquanto que o barramento de dados está ligado ao porto B. O barramento de leitura de dados está ligado ao porto de escrita do banco de registos. Estas ligações permitem executar uma leitura de memória para o banco de registos, através do controlo dos portos A e D do banco de registos, ou uma escrita na mesma, através do controlo dos portos A e B.

O funcionamento do circuito de dados é controlado pela palavra de controlo, sendo o funcionamento de cada um dos blocos descrito em detalhe nas secções seguintes.

12.1.1 Banco de Registos

O banco de registos foi estudado na Secção 7.5.4 e é utilizado quase sem modificações no circuito de dados do processador. A única alteração é que o registo `R0` toma, neste caso, sempre o valor `0`. O sinal de controlo de escrita, WR, é gerado pela unidade de controlo, assim como os valores nos barramentos $SelAD$ e $SelB$, de 4 bits cada. O valor de $SelAD$ especifica qual o registo cujo conteúdo é colocado no porto A e, simultaneamente, qual o registo em que deverá ser escrito o valor contido no porto D, se o sinal WR estiver activo. O valor de $SelB$ especifica qual o registo cujo conteúdo deverá ser colocado no porto B.

Para que seja possível flexibilizar o controlo da unidade lógica e aritmética, o registo `R0` contém sempre o valor 0. Isto permite seleccionar 0 como um dos operandos e efectuar diversas operações que, doutro modo, necessitariam de uma unidade lógica e aritmética mais complexa. Na prática, o registo `R0` não é um verdadeiro registo, uma vez que é implementado ligando as linhas do barramento directamente ao nível lógico `0`, através de *buffers* de três estados.

Com a excepção do registo `R0`, todos os outros registos podem ser usados para guardar valores. No entanto as funções de alguns dos registos estão pré-definidas, de acordo com a Tabela 12.1.

Note-se que todos os registos que estão destinados a funções específicas tem número superior a 7. Isto impede que um programa codificado ao nível do *assembly* tenha acesso aos mesmos e perturbe o funcionamento normal do processador, uma vez que, ao nível do *assembly*, o programador apenas pode aceder aos registos com números entre 0 e 7.

Entre os registos de uso especial, encontram-se o `R14` e o `R15`. O registo `R14` é o registo apontador da pilha, `SP`. É este registo que é usado para endereçar a memória quando se executa uma operação que manipule directa (`POP` ou `PUSH`) ou indirectamente (`CALL`, `INT`, `RET` e `RTI`) a pilha do processador.

O registo `R15` guarda o valor do contador de programa, `PC`, que, após a execução de cada instrução, aponta sempre para a próxima instrução que o processador irá executar.

Alterar o valor de qualquer um destes registos fora do seu contexto normal de utilização irá interromper o funcionamento normal do processador. Assim, o seu uso para quaisquer outras funções deverá ser sempre evitado.

Os registos `R11` a `R13` estão também destinados a funções específicas, mas o seu significado só se tornará claro quando se analisar a forma como são executadas as instruções *assembly*, na Secção 12.3.

Registo	Descrição
R0	Constante 0
R1	Registo de uso geral
R2	Registo de uso geral
R3	Registo de uso geral
R4	Registo de uso geral
R5	Registo de uso geral
R6	Registo de uso geral
R7	Registo de uso geral
R8	Registo de uso restrito
R9	Registo de uso restrito
R10	Registo de uso restrito
R11	Operando (SD)
R12	Endereço destino (EA)
R13	Resultado (RD)
R14	Apontador da pilha (SP)
R15	Contador programa (PC)

Tabela 12.1: Banco de registos.

12.1.2 Unidade Lógica e Aritmética

A unidade lógica e aritmética (ULA) usada por este processador é a que foi estudada na Secção 9.4, e que se reproduz na Figura 12.2.

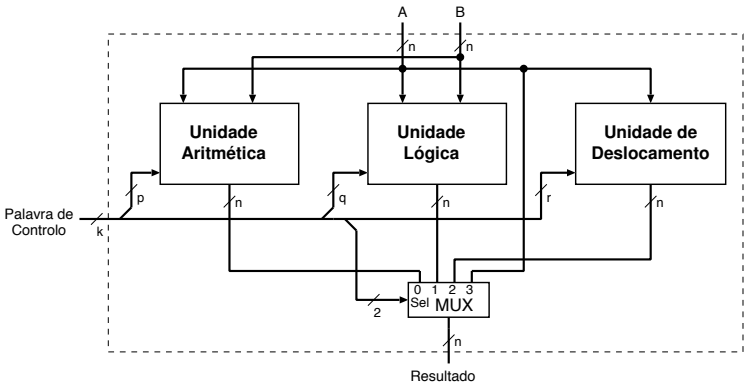


Figura 12.2: Estrutura da unidade lógica e aritmética.

A unidade lógica e aritmética é controlada por 5 bits de controlo, $CULA$. O valor destes 5 bits especifica, de acordo com a Tabela 9.1, qual a operação que a ULA executa sobre os dois operandos na sua entrada. Estes dois operandos são provenientes dos portos A e B do banco de registos.

Os quatro bits de estado gerados pela ULA estão ligados ao registo de estado, cujo funcionamento será detalhado na Secção 12.1.4.

12.1.3 Registo de Instrução

O registo de instrução, `RI`, encontra-se ligado directamente ao barramento de leitura de dados a partir da memória, é utilizado para guardar o código máquina da instrução *assembly* que está a ser executada.

Os 16 bits deste registo codificam, como foi descrito na Secção 11.4, qual a operação que deve ser executada e quais os operandos aos quais a mesma deve ser aplicada. De uma forma geral, o conteúdo deste registo não passa no circuito de dados, excepto na primeira fase de execução de uma instrução, em que o registo de instrução é carregado a partir de memória. O carregamento deste registo é controlado pelo valor do sinal LI, gerado pela unidade de controlo.

Para algumas operações, é necessário ler um ou mais campos do registo de instrução para o circuito de dados. Assim, é possível seleccionar o conteúdo do registo de instrução para a entrada no porto B da ULA, activando o sinal MB.

12.1.4 Registo de Estado

O registo de estado, `RE`, guarda os bits de estado do processador, permitindo ao programador testar o resultado da operação anterior e manter diversos bits de estado, que são actualizados de acordo o resultado das operações efectuadas pela unidade lógica e aritmética.

Quando o sinal de controlo LF está a 0, o valor dos bits de estado é actualizado de acordo com o resultado da última operação efectuada pela ULA. Para isso, o correspondente bit na máscara FM deverá estar a 1, de acordo com a Figura 12.3 definida pela unidade de controlo e não visível pelo programador ao nível do *assembly*. Os *bits de micro-estado*, z e c, são actualizados em todos

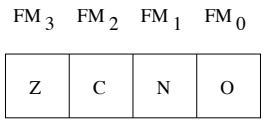


Figura 12.3: Bits da máscara FM que controla a actualização dos bits de estado.

os ciclos de relógio. A unidade de controlo define, em cada ciclo de relógio, quais os bits de estado que devem ser actualizados, de acordo com o que foi referido na Secção 11.6.1 sobre o modo como cada instrução *assembly* actualiza os mesmos.

Quando o valor de LF está a 1, o registo de estado é carregado com um valor proveniente do barramento $BUS\ A$, através do barramento FIN. Independentemente do valor dos sinais de controlo, o registo de estado pode ser carregado no banco de registos, através do porto D, usando para tal o barramento $FOUT$.

A Figura 12.4 descreve o esquema interno do registo de estado. O funcionamento do registo de estado é controlado por sinais activados pela unidade de controlo e por quatro bits de estado ZR, CR, NR, OR, gerados pela unidade lógica e aritmética.

Quando o sinal LF é activado pela unidade de controlo, o registo de estado é carregado a partir do barramento $BUS\ A$.

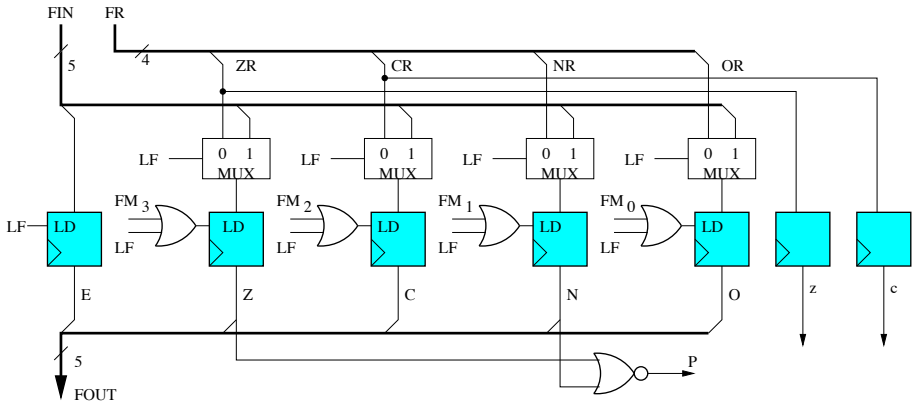


Figura 12.4: Esquema interno do registo de estado.

Além dos bits `Z`, `C`, `N` e `O`, cujo significado foi descrito na Secção 11.6.4, existe ainda o bit `E` que controla se o processador deve ou não responder a interrupções e cujo valor não vem da ULA, mas sim através da entrada FIN. Note-se que o sinal `P` não corresponde a um bit no registo de estado, uma vez que este bit é gerado pela lógica representada a partir dos valores dos bits `Z` e `N`.

12.1.5 Barramentos de Interligação

Os dois portos de leitura do banco de registos alimentam a unidade lógica e aritmética através dos multiplexadores `MUXA` e `MUXB`, que atacam os barramentos $BUS\ A$ e $BUS\ B$.

Quando é efectuado um acesso à memória, o $BUS\ A$ contém o endereço que deverá ser acedido. No caso de uma operação de escrita (sinal WM a 1), o valor no barramento $BUS\ B$ especifica qual o valor que deverá ser escrito. No caso de uma leitura (sinal WM a 0), o multiplexador $MUXD$ deverá ser controlado por forma a seleccionar o valor devolvido pela memória, colocando o valor de MD a 01. Neste caso, o valor lido da memória é escrito para o registo especificado pelo sinal de controlo $SelAD$, devendo o sinal de escrita no banco de registos, WR, estar activado.

12.1.6 Controlo do Circuito de Dados

O controlo do circuito de dados é feito através da palavra de controlo, descritos na Tabela 12.2. Cada um dos sinais descritos nesta tabela controla o funcionamento de um dos elementos do circuito de dados, tal como descrito nas secções anteriores.

Para se perceber melhor o funcionamento deste circuito de dados, consideram-se em seguida alguns exemplos que permitem ilustrar as operações de transferência entre registos possibilitadas pelo mesmo.

Suponha-se, por exemplo, que se pretende somar o conteúdo do registo `R3` com o conteúdo do registo `R7` e deixar o resultado no registo `R3`, alterando os valores dos bits de estado `C` e `Z`. Que valores devem tomar as variáveis de controlo do circuito de dados?

Sinal	# bits	Função
$Reset$	1	Inicializa o conteúdo dos registos a 0
$SelAD$	4	Controla os portos A e D do banco de registos
$SelB$	4	Controla o porto B do banco de registos
MA	1	Controlo do multiplexador A
MB	1	Controlo do multiplexador B
MD	2	Controlo do multiplexador D
WR	1	Escrita no banco de registos
WM	1	Escrita em memória
LF	1	Carrega os bits de estado
LI	1	Carrega o registo de instrução
FM	4	Controla a actualização dos bits de estado
$CULA$	5	Controla operação a executar na ULA
$CONST$	12	Valor de constante numérica

Tabela 12.2: Palavra de controlo do circuito de dados.

Em primeiro lugar, para que o banco de registos coloque nos portos A e B os registos `R3` e `R7`, os sinais $SelAD$ e $SelB$ devem tomar os valores 0011 e 0111, respectivamente. Adicionalmente, e como se pretende que tenha lugar uma escrita no banco de registos, o valor do sinal WR deverá ser 1.

Para que os valores presentes nos portos A e B do banco de registos cheguem às entradas da unidade lógica e aritmética, o valor dos sinais MA e MB deverá ser 0.

A operação da ULA é controlada pelo valor do sinal $CULA$. Da Tabela 9.1, tira-se que $CULA$ deverá tomar o valor 00000 para que seja realizada uma operação de adição. Para conseguir que os bits de estado sejam actualizados com os valores desta operação, o sinal LF deverá tomar o valor 0 e o sinal FM deverá tomar o valor 1100.

Finalmente, e para que o porto de escrita do banco de registos receba o valor de saída da unidade lógica e aritmética, é necessário controlar MD por forma a que o multiplexador D seleccione na sua saída o valor da sua entrada 0, o que consegue colocando MD a 00.

Resta agora controlar o valor dos sinais ainda não definidos por forma a que não se realizem operações indesejadas. Uma vez que não se pretende escrever na memória nem no registo de instrução os sinais WM e LI deverão tomar o valor 0. O campo constante, não utilizado, pode tomar qualquer valor.

Concluí-se assim que a micro-operação `R3←R3+R7` é executada se os sinais de controlo tomarem os valores da segunda coluna da Tabela 12.3.

Para um segundo exemplo, suponha-se agora que se pretende endereçar a memória usando o conteúdo do registo `R5` e guardar o valor dessa posição de memória no registo `R3`, sem mexer no conteúdo de qualquer dos outros registos.

Nestas condições, será necessário forçar $SelB$ a 0101 e $SelAD$ a 0011, activando também o sinal WR.

O multiplexador `MUXA` deverá agora seleccionar a sua entrada 1, pelo que o valor do sinal MA deverá ser 1. Para efectuar uma leitura de memória e seleccionar o valor lido como aquele que entra no porto D do banco de registo,

Sinal	R3 ← R3+R7	R3 ← M[R5]
$Reset$	0	0
$SelAD$	0011	0011
$SelB$	0111	0101
MA	0	1
MB	0	x
MD	00	01
WR	1	1
WM	0	0
LF	0	0
LI	0	0
FM	1100	0000
$CULA$	00000	xxxxx
$CONST$	xxxxxxxxxxx	xxxxxxxxxxx

Tabela 12.3: Sinais que controlam a execução das micro-operações R3 ← R3+R7 (coluna 2) e R3 ← M[R5] (coluna 3).

o sinal WM deverá tomar o valor 0 e o sinal MD o valor 01.

Para que não haja escrita no registo de instrução nem no registo de estado, os sinais LF e LI deverão tomar o valor 0. Também o sinal FM deverá ter todos os seus bits a 0.

Neste exemplo, como em muitos outros, os valores de alguns dos sinais são irrelevantes, uma vez que controlam partes do circuito de dados que não são lidos nem guardados. É o caso dos sinais de controlo $CULA$, MB e $CONST$, que podem tomar qualquer valor sem afectar o funcionamento do circuito. A terceira coluna da Tabela 12.3 descreve os valores que os sinais devem tomar para que seja executada a micro-operação R3 ← M[R5].

12.2 Unidade de Controlo

Os sinais que controlam o circuito de dados são gerados por uma unidade de controlo micro-programada, descrita nesta secção. Como foi visto na Secção 8.3.3, a utilização de uma unidade de controlo micro-programada permite uma maior flexibilidade da unidade de controlo, e também uma organização mais estruturada que no caso em que a unidade de controlo é feita sintetizando uma máquina de estados.

A Figura 12.5 descreve a estrutura geral da unidade de controlo do processador P3. O coração da unidade de controlo é o micro-sequenciador, que controla a ordem pela quais são executadas as micro-instruções guardadas na memória de controlo. As micro-instruções definem o valor dos sinais utilizados para controlar o circuito de dados, o próprio micro-sequenciador, e diversos aspectos do funcionamento dos outros módulos que aparecem na Figura 12.5.

O funcionamento do micro-sequenciador é controlado pela unidade de teste de condições e pela unidade de mapeamento, além dos sinais de controlo gerados directamente pelas micro-instruções. A unidade de teste de condições permite testar os bits de estado do processador e, também, outros bits internos

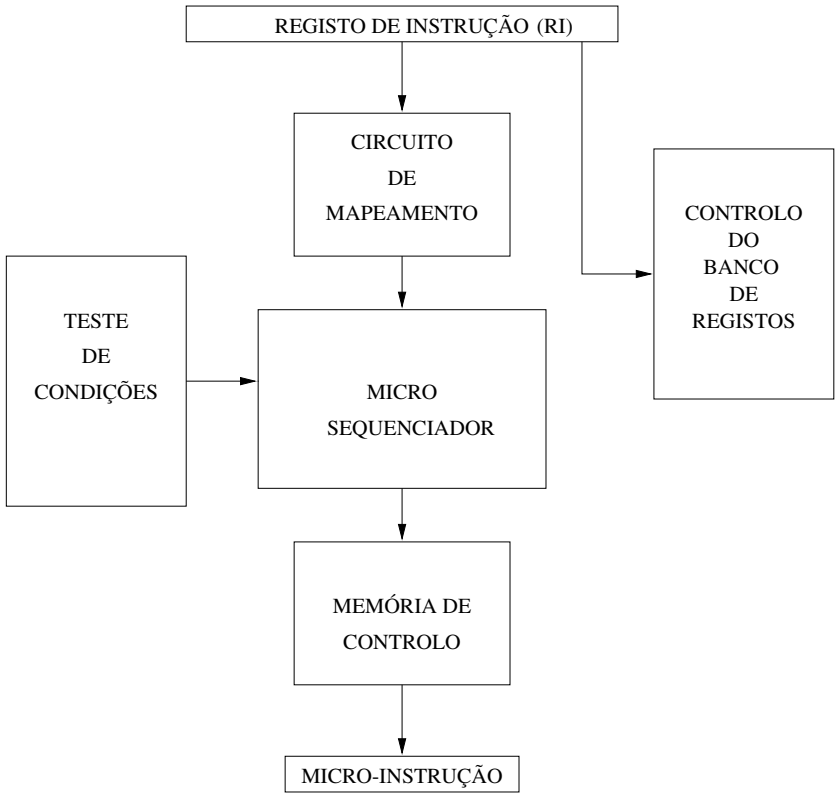


Figura 12.5: Esquema geral da unidade de controlo.

à unidade de controlo.

O controlo do banco de registos é feito por um circuito dedicado, controlado directamente por bits do registo de instrução e do registo de micro-instrução.

12.2.1 Formato das Micro-instruções

Tanto o circuito de dados como partes do próprio circuito de controlo são controlados por um conjunto de sinais que constituem a micro-instrução. Numa unidade micro-programada como esta, as micro-instruções que constituem o micro-programa são guardadas na memória de controlo, endereçada pelo registo `CAR`.

Uma opção possível é seleccionar uma micro-instrução que tenha um bit por cada um dos sinais de controlo que se pretendem gerar. No entanto, esta opção nem sempre se revela a mais adequada, uma vez que nem todas as combinações dos sinais de controlo são necessárias ou úteis.

A título de exemplo, no circuito da Figura 12.1, o valor do sinal $CULA$ não é importante quando se pretende carregar o valor de uma constante usando o sinal de controlo $CONST$.

Nesta realização, optou-se por utilizar dois formatos para a micro-instrução, que se distinguem entre si pelo valor do bit mais significativo da micro-instrução, F. A Figura 12.6 descreve os dois formatos possíveis para a micro-instrução.

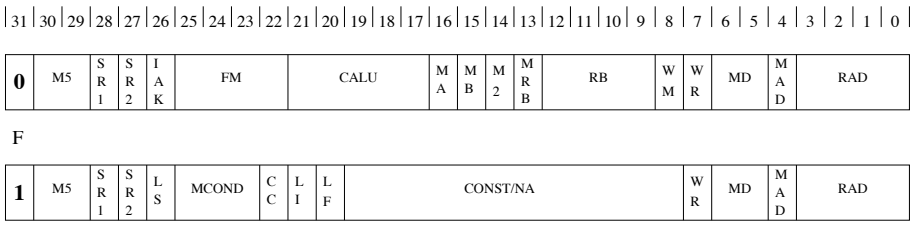


Figura 12.6: Formato das micro-instruções.

O formato correspondente a F=0 corresponde a micro-instruções que controlam fundamentalmente o circuito de dados, enquanto que o formato correspondente a F=1 é, prioritariamente, destinado a controlar a unidade de controlo, nomeadamente a unidade de teste de condições e o micro-sequenciador.

Muitas outras opções teriam sido possíveis, desde um formato único em que todos os sinais de controlo estivessem sempre disponíveis, até uma solução em que existissem mais do que dois formatos de micro-instrução. A primeira alternativa iria requerer uma micro-instrução com 50 bits, sem, no entanto, ser significativamente mais rápida, uma vez que raramente é necessário controlar todas as unidades do processador simultaneamente. Já a utilização de uma alternativa com mais formatos de micro-instrução poderia de facto reduzir o número de bits nas micro-instruções, mas implicaria uma redução significativa da velocidade de execução, por passarem a existir demasiados sinais que não podem ser controlados simultaneamente.

Pode observar-se que, na solução adoptada, alguns sinais estão presentes em ambos os formatos da micro-instrução, como, por exemplo, o sinal WR que controla a escrita no banco de registos. Estes sinais podem ser activados quer em micro-instruções do tipo 0 (F = 0) quer em micro-instruções do tipo 1 (F = 1).

Outros sinais estão presentes apenas num dos tipos de micro-instruções, como por exemplo o sinal que controla a escrita em memória, WM. Estes sinais só podem ser activados em micro-instruções desse tipo, devendo permanecer inactivos nas restantes. Isto significa que, no circuito de dados da Figura 12.1, os sinais que aparecem apenas num dos formatos de micro-instrução devem resultar da conjunção do sinal F, negado ou não, com o valor do bit da micro-instrução.

O circuito de dados, modificado para incluir explicitamente as portas lógicas que executam esta função, encontra-se representado na Figura 12.7. Note-se que a lógica adicionada força a que o controlo do multiplexador $MUXA$ e o sinal de escrita em memória fiquem activos apenas em micro-instruções do tipo 0, uma vez que o sinal que controla de facto o circuito de dados é a conjunção do sinal original com o complemento do bit F. Do mesmo modo, os 4 bits do sinal FM devem ficar activos apenas quando F=0, o que se consegue com 4 portas `AND`, na figura representadas apenas por uma delas aplicada aos 4 bits do barramento.

Os sinais de carregamento do registo de instrução e de carregamento do registo de estado, que só são gerados pela unidade de controlo em micro-instruções do tipo 1, são mascarados de forma análoga, sendo feita a conjunção com o bit F.

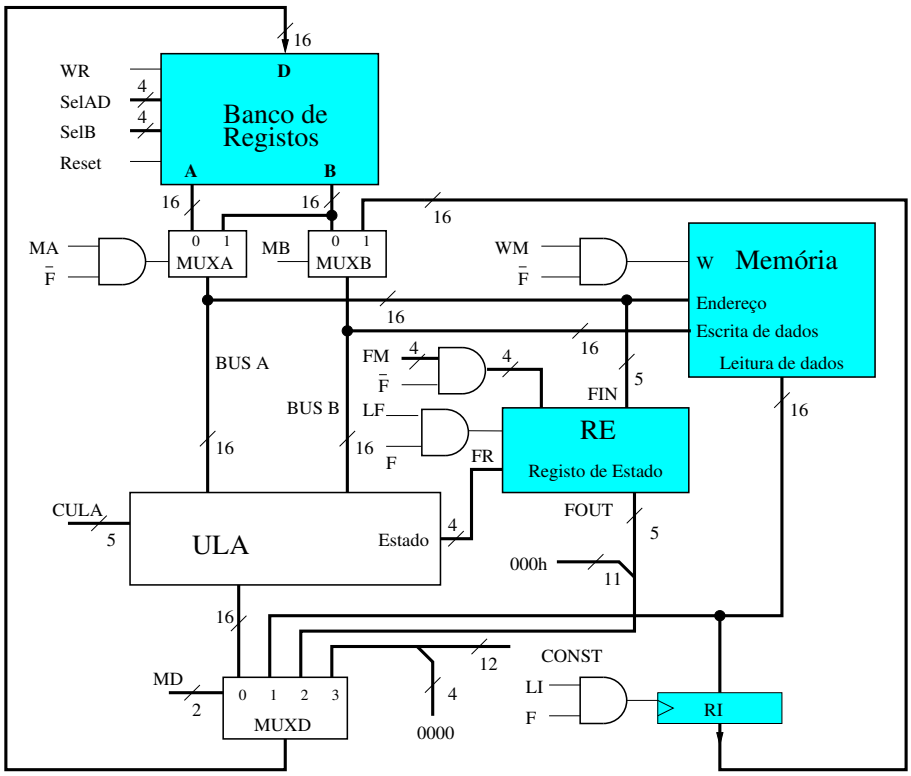


Figura 12.7: Circuito de dados e sinais de controlo.

A mesma regra é aplicável a sinais que controlam diversas partes da unidade de controlo propriamente dita.

De uma forma geral, quando um dado sinal é gerado pela unidade de controlo apenas no formato F=1, deverá ser feita uma conjunção com o sinal F, enquanto que quando esse sinal existe apenas no formato F=0, é feita a conjunção com o complemento do sinal F, isto é, $\overline{F}$.

Em alguns casos é possível poupar esta lógica se o sinal actuar partes do circuito que que não têm efeitos devido ao valor de outros sinais de controlo. Por exemplo, o sinal de controlo MB, que controla o multiplexador $MUXB$ está activo apenas no formato de micro-instrução que tem F=0. Assim, embora pudesse existir uma porta `AND` que fizesse a conjunção do sinal MB com a negação de F, esta porta revela-se inútil. De facto, uma análise cuidada do circuito de dados na Figura 12.1 mostra que, quando F=1, não existe controlo nem sobre a unidade lógica e aritmética nem sobre o sinal de escrita em memória, que está necessariamente inactivo. Assim, o valor que fica no barramento B nunca é usado, pelo que não é necessário controlar o multiplexador `MUXB`, poupando-se uma porta lógica.

12.2.2 Micro-Sequenciador

No centro da unidade de controlo encontra-se o micro-sequenciador, que gera a sequência de endereços de micro-instruções que deve ser executada. O micro-

sequenciador pode, em cada ciclo de relógio, executar uma das seguintes operações, de acordo com os sinais de controlo:

- Incrementar o endereço da micro-instrução a executar ou saltar para um endereço especificado na micro-instrução, de acordo com o valor do sinal $COND$, gerado pela unidade de teste de condições.
- Retornar de uma micro-rotina.
- Saltar para um dos endereços fornecidos pela unidade de mapeamento.

A estrutura do micro-sequenciador está descrita na Figura 12.8. O micro-

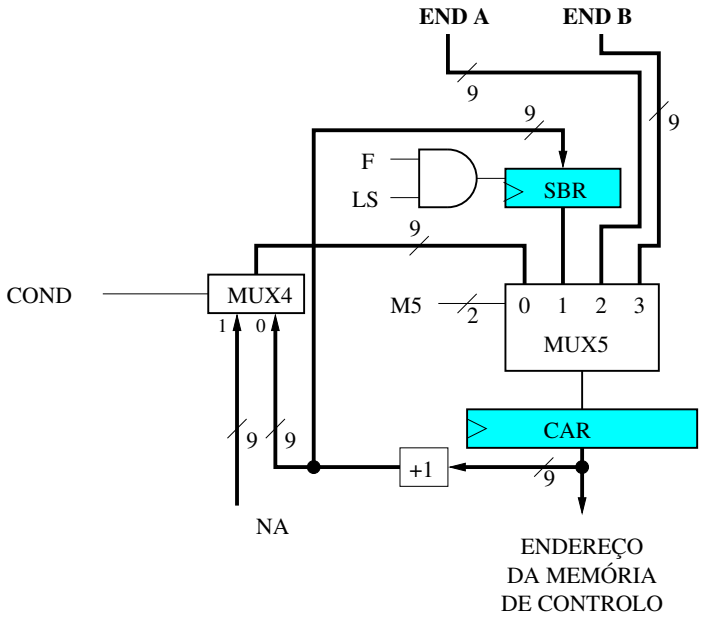


Figura 12.8: Esquema do micro-sequenciador.

sequenciador usa dois registos. O registo `CAR` (*Control Address Register*) contém o endereço da memória de micro-instruções onde está a micro-instrução que está a ser executada. Este registo desempenha para um micro-programa o mesmo papel que o contador de programa desempenha para um programa em *assembly* e pode também ser referido por *contador de micro-programa*. O registo `SBR` guarda o endereço de micro-programa para onde deverá ser transferido controlo após terminar a execução de uma micro-rotina. Uma vez que existe só um registo para guardar o endereço de retorno, apenas é possível utilizar um nível de profundidade de micro-rotinas, o que significa que uma micro-rotina não pode chamar outra.

É importante sublinhar que não existe qualquer relação entre uma micro-rotina e uma subrotina definida ao nível do *assembly*.

O funcionamento do micro-sequenciador é controlado pelos dois bits do sinal $M5$ que controla o carregamento do registo `CAR`, da seguinte forma:

- $M5 = 00$: `CAR` é incrementado se $COND$ for 0, e é carregado com o valor de NA (o endereço da memória de controlo para onde se deve saltar), se

$COND$ for 1, permitindo assim a execução de saltos condicionais dentro de micro-programas.

- $M5$ = 01 : `CAR` é carregado com o valor contido em `SBR`, sendo assim executado um retorno de micro-rotina.
- $M5$ = 10 : `CAR` é carregado com o valor especificado em $ENDA$ pela unidade de mapeamento.
- $M5$ = 11 : `CAR` é carregado com o valor especificado em $ENDB$ pela unidade de mapeamento.

A maioria das micro-instruções são executadas de forma sequencial, pelo que após a execução de uma micro-instrução o `CAR` deve ser incrementado para ficar a apontar para a próxima posição da memória de controlo. Este comportamento consegue-se colocando o valor de $M5$ a 00 na micro-instrução, e controlando o bloco de teste de condições por forma a que o valor de $COND$ seja 0. Caso se pretenda executar um salto condicional no micro-programa, o valor de $M5$ deverá ser 00 na mesma, e a unidade de teste de condições deverá seleccionar a condição de salto desejada.

A possibilidade de carregar o registo `CAR` com o conteúdo do registo `SBR` permite a existência de micro-rotinas. Quando se pretende chamar uma micro-rotina, o sinal LS deve ser activado e o registo CAR deve ser carregado com o valor de NA, que especifica o endereço da micro-rotina que se pretende chamar. Como o sinal LS só existe nas micro-instruções com F=1, o sinal de carregamento do registo `SBR` deve ser inibido quando F=0. A activação deste sinal faz com que o registo `SBR` seja carregado com o valor de `CAR`+1, que representa o endereço da primeira micro-instrução a ser executada quando a execução da micro-rotina termina. O retorno de micro-rotina é executado seleccionado $M5$ = 01 o que força o carregamento do registo `CAR` com o endereço de retorno.

O conteúdo do registo `CAR` também pode ser carregado com os valores $ENDA$ (colocando $M5$=10) ou $ENDB$ (colocando $M5$=11), gerados pela unidade de mapeamento, e cuja utilidade será estudada na Secção 12.2.4.

12.2.3 Teste de Condições

A unidade de teste de condições, descrita na Figura 12.9, tem como função seleccionar qual a condição que é testada pelo micro-sequenciador quando o mesmo executa uma micro-instrução de salto ou chamada a micro-rotina, condicional ou não.

Esta unidade tem um único bit de saída, o sinal $COND$, que indica ao micro-sequenciador se deve ou não executar um salto, tal como foi descrito na secção anterior.

Esta unidade é fundamentalmente constituída por dois multiplexadores, e algumas portas lógicas auxiliares. O multiplexador, `MUXCOND`, controlado pelo campo $MCOND$ da micro-instrução, permite seleccionar um dos seguintes bits:

- A constante 1, o que permite ao micro-sequenciador incrementar ou executar saltos incondicionais, dependendo do valor do sinal CC (*Complementar Condição*).

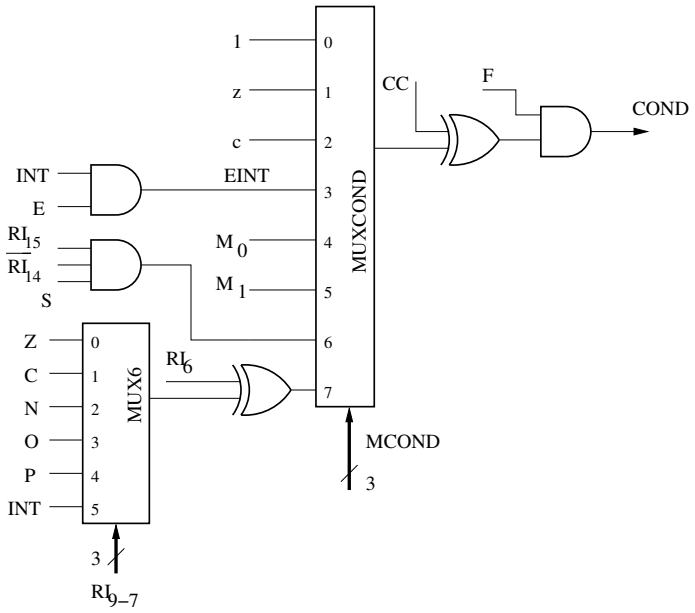


Figura 12.9: Unidade de teste de condições.

- Os bits z e c do registo de estado. Estes bits são também conhecidos por bits de micro-estado. O bit z está a 1 se a última operação na ULA deu resultado zero, enquanto que o bit c está a 1 se a última operação na ULA gerou transporte.
- A conjunção do bit `E` do registo de estado com o sinal INT que assinala a presença de uma interrupção, indicando se deve atender uma interrupção presente.
- Os bits do campo M do registo de instrução, que codificam o modo de endereçamento da instrução *assembly* que está a ser executada.
- O bit S do registo de instrução, mascarado pela expressão $RI_{15}\overline{RI_{14}}$ que indica qual o operando ao qual deve ser aplicado o modo de endereçamento da instrução, no caso de instruções de dois operandos.
- Um dos bits do registo de estado, mais os sinais P e INT, escolhido de acordo com o valor dos bits 7 a 9 do registo de instrução.

A entrada 7 do multiplexador `MUXCOND` é controlada pela saída do multiplexador `MUX6`. Uma vez que os bits 7 a 9 do registo de instrução especificam, conforme Figura 11.13) e de acordo com a Tabela 11.14, qual a condição que é testada numa dada instrução, é possível seleccionar directamente qual a condição que deverá ser testada pelo micro-sequenciador.

Uma análise da Tabela 11.14 revela que os bits 7, 8 e 9 do registo de instrução seleccionam qual a condição, enquanto que o bit 6 indica se a condição deve ser complementada. A ligação do multiplexador `MUX6` conforme indicado na figura e o uso de uma porta `XOR`, que funciona como uma negação condicional, permite que, na entrada 7 do multiplexador `MUXCOND`, esteja já seleccionada a

condição correcta. Isto leva a que o teste de uma destas condições seja efectuado numa só micro-instrução, não sendo necessário efectuar diversos testes aos valores dos bits de estado e do valor dos bits 6 a 9 do registo de instrução.

Quando se pretende que o micro-sequenciador execute as instruções de forma sequencial, deverá ser colocado o valor 0 na saída $COND$. Isto é conseguido colocando o valor 000 no campo $MCOND$ da micro-instrução e o valor 1 no campo CC da micro-instrução.

Caso se pretenda que o micro-sequenciador execute um salto incondicional, deverá ser colocado o valor 1 na saída $COND$, colocando 000 em $MCOND$ e 0 em CC.

Finalmente, caso se pretenda que o micro-sequenciador execute um salto na condição de um dado bit tomar um dado valor, o multiplexador `MUXCOND` deverá ser controlado de forma a seleccionar o bit pretendido, enquanto que o sinal CC define se a condição deverá ser complementada ou não.

Tanto o salto incondicional como o salto condicional só podem ser executados pelo micro-sequenciador quando a micro-instrução é do tipo F=1, uma vez que só neste formato estão disponíveis os campos $COND$, CC e o endereço de salto NA.

Caso a micro-instrução seja do tipo 0, o valor de $COND$ é colocado a 0 e o micro-sequenciador incrementa sempre o contador de micro-programa.

12.2.4 Unidade de Mapeamento

A unidade de mapeamento é utilizada para gerar, de forma rápida, os endereços das micro-rotinas chamadas durante a execução de instruções. Com efeito, em diversos passos da execução de uma instrução *assembly* torna-se necessário saltar para uma micro-rotina ou troço de micro-código, de acordo com o valor presente num dado campo do registo de instrução.

Por exemplo, o valor contido nos seis bits mais significativos do registo de instrução representa o código de instrução e define qual a operação que deverá ser executada. Este valor é utilizado na fase de descodificação de uma instrução *assembly*, para gerar o endereço da memória de controlo que corresponde às micro-instruções que implementam a instrução *assembly*.

Noutra fase da execução da instrução, é necessário saltar para um dado endereço de micro-código, de acordo com o modo de endereçamento utilizado, e especificado no campo M do registo de instrução.

A unidade de mapeamento é utilizada em diversas fases da execução de uma instrução. No processador P3, esta execução é feita nos seguintes passos:

1. Carregamento do registo de instrução.
2. Descodificação do código de operação e carregamento dos operandos.
3. Execução do micro-programa que implementa a instrução.
4. Escrita do resultado.
5. Teste de pedidos de interrupção.

Com esta sequência de operações, é necessário descodificar o código de operação para saber quais os operandos a carregar e qual o endereço da memória de micro-programa que contém as micro-instruções a executar. Dado

que o micro-sequenciador pode testar de cada vez apenas o valor de um bit, seleccionado pela unidade de teste de condições, e que o código de operação tem 6 bits, a escolha da micro-rotina utilizando este mecanismo iria requerer seis micro-instruções só para descobrir qual o endereço da micro-rotina a efectuar. Seriam ainda necessárias mais micro-instruções para decidir quais as micro-rotinas que deveriam ser chamadas para fazer a leitura dos operandos e a escrita do resultado, o que se revelaria muito ineficiente. A unidade

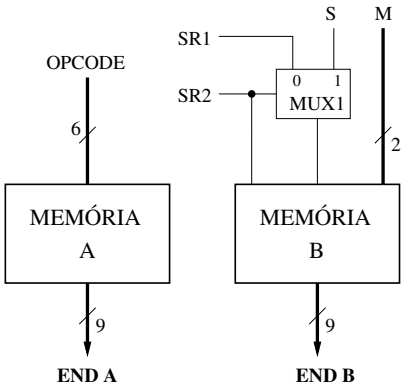


Figura 12.10: Unidade de mapeamento.

de mapeamento, esquematizada na Figura 12.10, permite que a transferência de controlo seja feita numa única micro-instrução. Isso consegue-se utilizando duas memórias de mapeamento, que são endereçadas de acordo com o código da operação, o modo de endereçamento e o valor do bit de direcção presentes no registo de instrução.

A unidade de mapeamento pode gerar dois endereços, qualquer um dos quais pode ser seleccionado pelo micro-sequenciador como o próximo endereço de micro-instrução a executar (ver Figura 12.8).

A memória A é endereçada directamente pelo código de operação (campo $OPCODE$ da instrução *assembly*) e implementa uma tabela que contém os endereços das micro-rotinas que executam as operações de transferência entre registos que realizam cada uma das instruções.

As linhas de endereço da memória de mapeamento B são controladas pelo valor dos bits $SR1$ e $SR2$ da micro-instrução e também pelo valor dos bits M e S do registo de instrução. De acordo com os valores dos bits $SR1$ e $SR2$, esta memória gera o endereço das micro-rotinas de carregamento de operandos ou de escrita do resultado.

Isto permite que, através do controlo dos bits $SR1$ e $SR2$, seja possível gerar quatro endereços diferentes. Estes endereços são usados para especificar diversas micro-rotinas de leitura e escrita, conforme especificado na Tabela 12.4.

Assim, caso se pretenda que a memória de mapeamento B gere o endereço da micro-rotina de leitura de um operando, a micro-instrução deverá colocar $SR2$ e $SR1$ a 00. Neste caso, o valor de S será ignorado, o que é correcto uma vez que este campo não tem significado quando a instrução é de um operando. Caso se pretenda o endereço da micro-rotina de leitura de dois operandos, basta forçar $SR2$ a 1, sendo nestas condições o valor de S usado para endereçar a memória, através do multiplexador `MUX1`.

SR2	SR1	S	Endereço seleccionado
0	0	-	Micro-rotina de leitura de um operando
0	1	-	Micro-rotina de escrita do resultado
1	-	0	Micro-rotina de leitura de dois operandos para S=0
1	-	1	Micro-rotina de leitura de dois operandos para S=1

Tabela 12.4: Funcionamento da memória de mapeamento B.

Para que o circuito de controlo funcione de acordo com o especificado, a memória B deverá ser carregada com os endereços das micro-rotinas correspondentes a cada uma das operações desejadas, de acordo com o esquema da Figura 12.11.

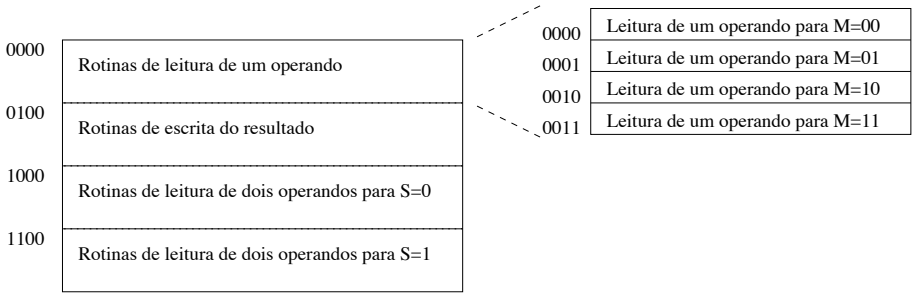


Figura 12.11: Preenchimento da memória de mapeamento B.

Para se analisar o funcionamento da unidade de mapeamento, suponha-se que se pretende controlar o micro-sequenciador por forma a transferir controlo para a primeira micro-instrução que implementa a instrução *assembly* guardada no registo de instrução. O endereço desta micro-instrução está guardado na memória de mapeamento A, que é endereçada pelos 6 bits mais significativos (campo $OPCODE$) do registo de instrução.

É necessário agora assegurar que este valor é carregado no registo `CAR` no próximo flanco do relógio. Por análise da Figura 12.8, verifica-se que o sinal de controlo $M5$ deverá tomar o valor 10, por forma a que o multiplexador $MUX5$ seleccione o valor gerado pela memória de mapeamento A como o próximo valor do registo `CAR`.

O controlo da memória de mapeamento B é ligeiramente mais complexo, mas também fácil de perceber. Suponha-se que se pretende transferir o controlo para a sequência de micro-instruções que carrega um operando. Por análise da Tabela 12.4, verifica-se que é necessário colocar os bits de controlo $SR2$ e $SR1$ a 00 para que a memória de mapeamento B gere o endereço dessa micro-rotina. Dado que valor do campo M do registo de instrução endereça directamente esta memória, basta agora colocar o sinal $M5$ a 11 para que a próxima micro-instrução a ser executada seja a desejada.

12.2.5 Controlo do Banco de Registos

A unidade de controlo controla o banco de registos através do circuito de controlo do banco de registos, descrita na Figura 12.12. Controlando o valor do

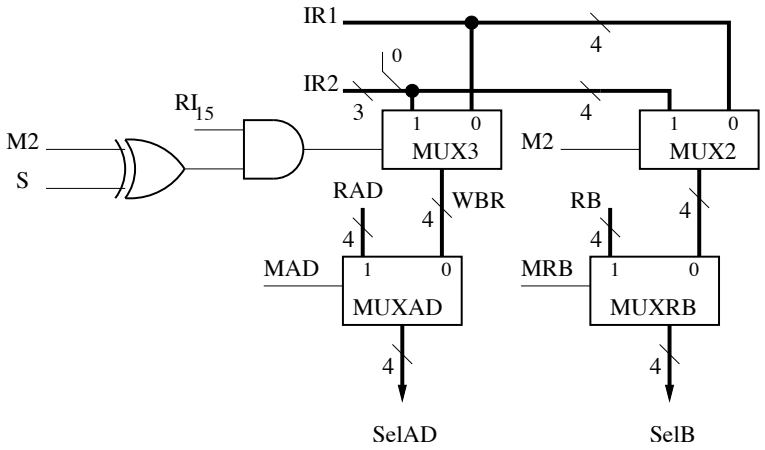


Figura 12.12: Circuito de controlo do banco de registos.

sinal MRB, disponível na micro-instrução, a unidade de controlo escolhe se o endereço do porto B do banco de registos é igual a RB (especificado na micro-instrução) ou aos valores especificados no registo de instrução. A situação é idêntica para o controlo dos portos A e D do banco de registos, sendo desta vez a escolha controlada pelo sinal MAD.

Quando o endereço do porto B é especificado directamente pelo registo de instrução, o valor do sinal de controlo $M2$ escolhe qual dos campos do registo de instrução deverá controlar este endereço.

No caso dos portos A e D, esta escolha é feita directamente por lógica que usa os valores do bit S e do bit mais significativo do código de operação. Esta lógica permite escolher o valor de WBR como sendo igual a $IR1$ ou $IR2$, de acordo com a Tabela 12.5. Apesar da aparente complexidade desta tabela, a

RI_{15}	S	M2	WBR
0	-	-	IR1
1	0	0	IR1
1	0	1	IR2
1	1	0	IR2
1	1	1	IR1

Tabela 12.5: Controlo do multiplexador `MUX3`.

sua utilização é muito simples, e é descrita na Tabela 12.6.

A lógica descrita na Figura 12.12 é necessária porque, nas instruções com dois operandos, o campo do registo de instruções que contém o registo ao qual é aplicado o modo de endereçamento pode ser $IR1$ ou $IR2$, de acordo com o valor de S. Estas instruções têm todas o bit mais significativo do registo de instrução a 1, conforme descrito na Secção 11.7.3. Todas as instruções que têm

M2	Valor seleccionado
0	Registo usado pelo primeiro ou único operando
1	Registo usado pelo segundo operando, quando exista

Tabela 12.6: Valor seleccionado pelo multiplexador `MUX3`.

o bit mais significativo do registo de instrução a 0 devem escrever o resultado no registo especificado pelo campo $IR1$, o que é conseguido com a porta `AND` da figura.

Para ilustrar o funcionamento desta unidade do micro-controlador, suponha-se que se pretende operar com registos definidos pelos valores dos campos da micro-instrução (`RAD` e `RB`), sem ter em atenção quais os registos definidos pela instrução *assembly* propriamente dita. Neste caso, há que endereçar o banco de registos com o valor definido pelos campos RAD e RB da micro-instrução, pelo que os sinais MAD e MRB deverão ser colocados a 1.

Para uma utilização mais complexa, suponha-se que se está na fase final de execução da instrução *assembly* `ADD R1, M[R7+00A0h]`. A codificação desta instrução já foi estudada na Secção 11.7.5, onde se viu que esta instrução é representada pela sequência de duas palavras `1000011001110111b(8677h)` e `0000000010100000b(00A0h)`. Uma vez que se está na fase final de execução da instrução, o resultado da adição, já calculado, está guardado no registo `R13`. Pretende-se carregar este resultado no registo de destino especificado pela instrução *assembly*. Como se viu na Secção 11.7.5, o registo de destino encontra-se especificado no campo $IR2$ do registo de instrução. Isto acontece porque, na codificação desta instrução, o modo de endereçamento indexado se aplica ao segundo operando da instrução, o que é indicado pelo valor S=1.

Para conseguir executar esta operação, há, em primeiro lugar, que garantir que o valor presente no porto B do banco de registos se propague até ao porto de escrita do mesmo. Por análise da Figura 12.1 e da Tabela 9.1, verifica-se que é necessário colocar os valores no campos da micro-instrução de acordo com a Tabela 12.7.

Sinal	Valor
MB	0
MD	00
WR	1
WM	0
LF	0
LI	0
FM	0000
CULA	11XXX

Tabela 12.7: Sinais que controlam a execução da micro-operação `R1` ← `R13`.

É agora necessário controlar o circuito da Figura 12.12 por forma a conseguir que o registo usado como primeiro operando da instrução *assembly* seja escrito com o resultado. O valor de MRB deverá ser 1, para que o campo RB da micro-instrução possa especificar o registo `R13`. Já o valor de MAD deverá

ser 0, para que o valor seleccionado pelo multiplexador $MUXAD$ seja o que provêm do multiplexador $MUX3$.

Para controlar adequadamente o multiplexador $MUX3$, basta colocar $M2$ a 0, como descrito na Tabela 12.6. Pode verificar-se que o valor do bit RI_{15} do registo de instrução toma o valor necessário para que o controlo do multiplexador $MUX3$ fique com o valor adequado, que, neste caso, é 1.

12.2.6 Circuito de Controlo

O circuito completo de controlo do processador P3, descrito na Figura 12.13 resulta da ligação dos diversos blocos descritos nas secções anteriores. Note-

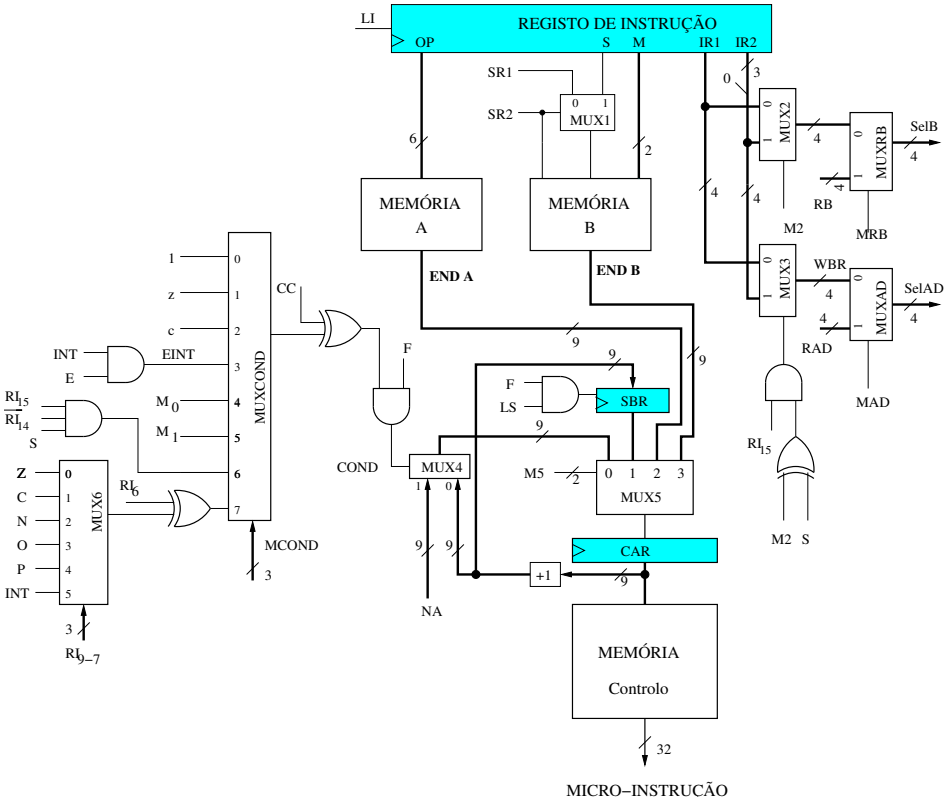


Figura 12.13: Circuito de controlo do processador P3.

se que o controlo do multiplexador `MUX5` permite seleccionar, como próximo endereço da micro-instrução a ser executada, quer o que vem do multiplexador `MUX4`, controlado pela unidade de teste de condições, quer o conteúdo do registo `SBR`, quer qualquer um dos endereços especificados pela unidade de mapeamento.

O sinal LI permite carregar o registo de instrução, que se encontra duplicado nesta figura e na Figura 12.1, que ilustra o circuito de dados.

O registo `CAR` é carregado em todos os ciclos de relógio, pelo que não são necessários sinais para controlar o seu carregamento.

12.3 Micro-Programação

Uma vez percebido o funcionamento da unidade de controlo, a programação da memória de controlo é relativamente simples. Com efeito, por análise do circuito de controlo, é possível identificar o valor que cada bit da micro-instrução deve tomar para se obtenha o funcionamento desejado do circuito de dados, assim como um comportamento correcto do próprio circuito de controlo.

O primeiro passo para a definição do conteúdo da memória de controlo é a definição da estrutura de alto nível dos micro-programas. Esta estrutura é, em grande parte, imposta pelas características do circuito de controlo e pelos tipos de operações que podem ser efectuados por este.

A sequência de operações efectuada quando uma instrução *assembly* é executada foi descrita na Secção 12.2.4. A esta sequência de operações corresponde o fluxograma da Figura 12.14. A execução de uma instrução do processador

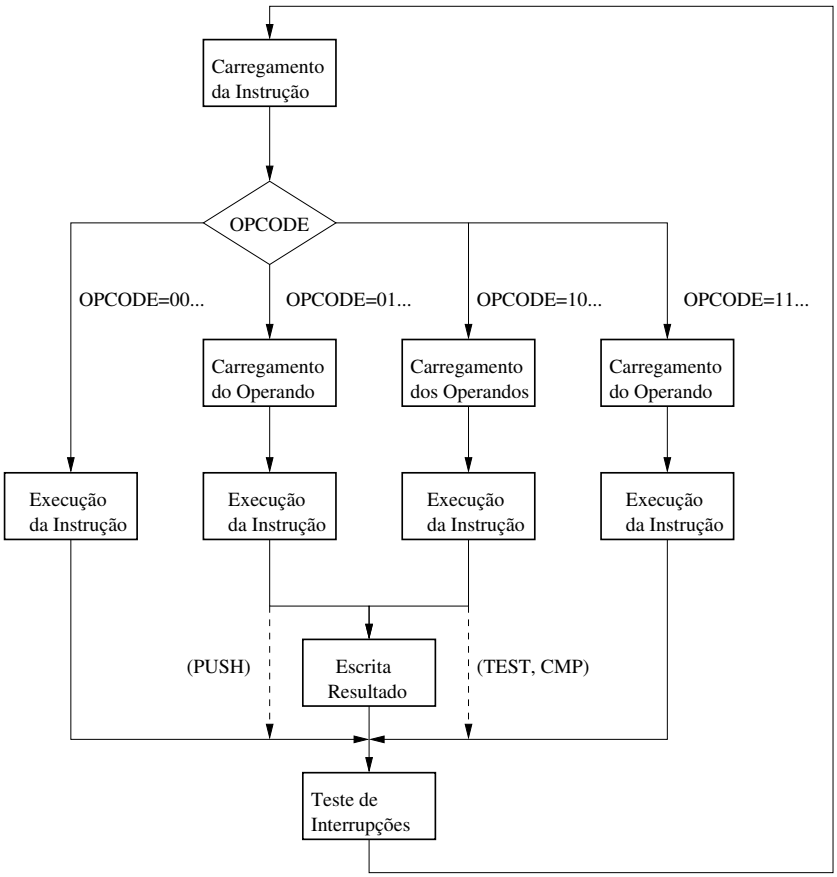


Figura 12.14: Fluxograma da execução de uma instrução *assembly*.

começa com o carregamento da instrução, da memória para o registo de instrução. Com base no código da operação, é efectuado um salto para o endereço da primeira micro-instrução que executa essa operação, usando para tal a me-

mória A do circuito de mapeamento.

Para as instruções que têm um ou dois operandos, a execução prossegue com o carregamento dos mesmos. Este carregamento é conseguido chamando uma micro-rotina, cujo endereço é fornecido pela memória B do circuito de mapeamento. A fase seguinte consiste na execução da instrução propriamente dita, usando uma sequência de micro-instruções que são específicas para cada instrução.

Após a execução da instrução, o controlo é transferido para a secção de micro-código que escreve o resultado, nos casos em que a instrução requer esta operação[1]. Para isto, é novamente utilizada a memória B do circuito de mapeamento, uma vez que a micro-rotina de escrita do resultado varia com o modo de endereçamento utilizado na instrução.

Finalmente, após a escrita do resultado (ou após a execução, no caso de instruções que não necessitam desse passo), o controlo é transferido para uma sequência de micro-instruções que verificam se existe uma interrupção pendente. Em caso afirmativo, o controlo é transferido para a micro-rotina de tratamento de interrupção. A excepção é a instrução `INT`, que não necessita de verificar se existem interrupções pendentes uma vez que faz a desactivação do bit do registo de estado que permite o atendimento de interrupções.

12.3.1 Carregamento do Registo de Instrução

Como foi visto atrás, a primeira fase da execução de uma instrução é o carregamento do registo de instrução, `RI` (em Inglês, esta fase chama-se *instruction fetch*, ou, abreviadamente, *IF*). Nesta fase, é necessário carregar o registo de instrução com o conteúdo da posição de memória apontada pelo contador de programa, `PC`. Em seguida, deve ser incrementado o valor do contador de programa, para que o mesmo fique a apontar para a próxima instrução a ser executada. A última operação consiste em transferir controlo para o troço de micro-programa que executa as operações necessárias à execução da instrução *assembly*. Ao contrário do *assembly* que apenas permite que uma e uma só operação seja feita em cada ciclo de relógio, um passo de micro-programa pode executar mais do que uma operação de transferência entre registos, desde que os circuitos de dados e controlo assim o permitam. A Tabela 12.1 descreve a sequência de operações de transferência entre registos que implementam esta fase da execução de uma micro-instrução. Como se pode observar, esta sequência de operações é descrita utilizando uma sequência de micro-instruções descritas na linguagem de transferência entre registos que foi descrita no Capítulo 9.

```
IF0:  RI←M[PC]                      ; Carrega RI
IF1:  PC←PC+1, CAR←ROMA[OPCODE]     ; Incrementa PC
```

Programa 12.1: Micro-programa da fase de carregamento do registo de instrução.

[1] Algumas instruções, como por exemplo, as instrução de `JMP`, `TEST` e `CMP` não geram qualquer resultado que necessite de ser escrito no operando da instrução.

Cada micro-instrução corresponde a uma ou mais operações de transferência entre registos, que são executadas sempre os que sinais de controlo correspondentes a essa micro-instrução estão activos. Para evitar a listagem exaustiva de todos os sinais de controlo que se encontram activos em cada micro-instrução, cada micro-instrução é precedida de um rótulo simbólico. Este rótulo corresponde aos valores dos sinais de controlo que se encontram activos durante a execução dessa micro-instrução.

Embora cada micro-instrução seja descrita como um conjunto de transferências de valores entre registos, a programação da memória de controlo é feita com uma sequência de zeros e uns que especificam o valor de cada micro-instrução, de acordo com o formato descrito na Figura 12.6. Os valores de cada bit da micro-instrução deverão ser tal que causem as transferências entre registos indicadas.

A transformação do micro-programa, descrito ao nível de transferência entre registos, na combinação de bits que controla adequadamente os circuitos do processador pode ser feita manualmente ou com o auxílio de um programa, que, neste caso, se chama micro-*assembler*. Na prática, e dado que a programação a este nível requer um conhecimento profundo dos sinais e circuitos envolvidos, a utilidade de um micro-*assembler* é consideravelmente mais reduzida que a de um *assembler*, pelo que esta transformação é feita, na maior parte dos casos, manualmente.

No caso presente, ilustra-se o funcionamento das micro-rotinas usando a linguagem de transferência entre registos, sendo a Secção 12.3.6 dedicada ao estudo do processo de tradução entre a micro-instrução e o micro-código em formato binário.

12.3.2 Carregamento dos Operandos

A fase seguinte da execução de uma instrução consiste no carregamento dos operandos, de acordo com o número dos mesmos e o seu modo de endereçamento. A micro-rotina a chamar depende da instrução *assembly* que está a ser executada, embora o procedimento seja similar em todas elas. Como foi descrito na Figura 12.10 da Secção 12.2.4, a memória de mapeamento B deve conter o endereço das micro-rotinas de leitura de operandos, quer para os casos em que existe apenas um operando, quer para os casos em que existem dois operandos.

Assim, uma instrução *assembly* que necessite apenas de um operando deverá chamar a micro-rotina correspondente usando a memória B da unidade de mapeamento, com os bits de controlo $SR1$ e $SR2$ a 0. Uma instrução que necessite de dois operandos deverá endereçar a memória B da unidade de mapeamento com o sinal $SR2$ a 1. Neste caso, o segundo bit de endereço desta memória é o bit S do registo de instrução, de acordo com o circuito da Figura 12.10.

Por forma a comunicarem facilmente entre si, os diversos passos de execução de uma instrução *assembly* usam os registos temporários de uma forma regular, de acordo com a seguinte convenção e a Tabela 12.1.

- O registo `EA` (`R12`) é usado para guardar o endereço efectivo de um operando sempre que este operando provenha de memória (do inglês, *effective address*).

- O valor do primeiro operando deve ser copiado para o registo `RD` (`R13`). Após os cálculos, o resultado da operação deve ser guardado nesse mesmo registo.

- O valor do segundo operando deve ser copiado para o registo `SD` (`R11`), sempre que a instrução use dois operandos.

As instruções de carregamento de operandos deverão funcionar de forma a deixar o endereço do operando no registo `EA` (`R12`). O valor do primeiro ou único operando deverá ser guardado no registo de destino, `RD` (`R13`). Em alguns modos de endereçamento (por exemplo, no modo imediato), não é necessário o endereço do operando e, nestes casos, o registo `EA` não é carregado. Quando as instruções têm dois operandos, o segundo operando deve ser deixado no registo `SD` (`R11`).

Pode-se agora examinar as micro-rotinas de carregamento de operandos, começando pelas que carregam apenas um operando. De acordo com o modo de endereçamento, o operando pode estar em várias localizações:

- Endereçamento por registo: o operando encontra-se num registo. Deve ser copiado desse registo para o registo `RD`. Não existe necessidade de actualizar o registo `EA`.

- Endereçamento indirecto por registo: o operando encontra-se na posição de memória cujo endereço está contido no registo. O valor do registo deve ser copiado para o registo `EA` e o valor da memória apontado por este registo deve ser copiado para o registo `RD`.

- Endereçamento imediato: o operando encontra-se na própria instrução, ou, mais exactamente, na posição de memória apontada pelo contador de programa, que já foi incrementado na micro-instrução com rótulo `IF1`. Esta posição de memória deve ser copiada para o registo `RD`, não havendo necessidade de actualizar o registo `EA`.

- Endereçamento indexado: o operando encontra-se numa posição de memória cujo endereço é obtido somando o conteúdo de um registo com o valor da posição de memória apontada pelo contador de programa. Este endereço deve ser carregado no registo `EA` e o seu conteúdo deverá ser copiado para o registo `RD`.

Em todos os casos, o registo ao qual se aplica o modo de endereçamento é o registo especificado no campo $IR1$ do registo de instrução.

Após a execução das micro-instruções que carregam o operando, o controlo deve ser retornado para o código que chamou a micro-rotina de carregamento de operando.

Torna-se agora simples especificar os micro-programas para cada uma das micro-rotinas de carregamento de um operando, descritas no Programa 12.2. A mais complexa desta micro-rotinas é a que trata do carregamento de operandos quando o modo de endereçamento é indexado, caso em que há que somar a palavra `W` ao valor do registo especificado em $IR1$ e endereçar a memória com o valor resultante.

As micro-rotinas de carregamento de dois operandos funcionam de forma semelhante às de carregamento de um operando. Porém, neste caso, o valor

```
F1R0:   RD←R[IR1], CAR←SBR  ; Copia operando

F1RI0:  EA←R[IR1]           ; Carrega o endereço
F1RI1:  RD←M[EA], CAR←SBR   ; Copia operando

F1IM0:  RD←M[PC]            ; Carrega o operando
F1IM1:  PC←PC+1, CAR←SBR    ; Incrementa o PC

F1IN0:  EA←M[PC]            ; Carrega a constante W
F1IN1:  PC←PC+1             ; Incrementa PC
F1IN2:  EA←EA+R[IR1]        ; Guarda o endereço
F1IN3:  RD←M[EA], CAR←SBR   ; Carrega o operando
```

Programa 12.2: Micro-rotinas de carregamento de um operando.

do bit S do registo de instrução indica se o modo de endereçamento é aplicado ao primeiro operando ou ao segundo operando. Quando o modo de endereçamento é aplicado ao primeiro operando, também se aplica ao destino da operação, uma vez que o primeiro operando especifica simultaneamente um dos operandos e o destino a dar ao resultado. Em ambos os casos, o modo de endereçamento aplica-se ao registo especificado no campo $IR1$ do registo de instrução. O outro operando, especificado pelo campo $IR2$ do registo de instrução, é sempre um registo.

```
F2R0:   RD←R[IR1]           ; Copia primeiro operando
F2R1:   SD←R[IR2], CAR←SBR  ; Copia segundo operando

F2RI0:  EA←R[IR1]           ; Guarda endereço
F2RI1:  RD←M[EA]            ; Copia primeiro operando
F2RI2:  SD←R[IR2], CAR←SBR  ; Copia segundo operando

F2IM0:  RD←M[PC]            ; Copia primeiro operando
F2IM1:  PC←PC+1             ; Incrementa o PC
F2IM2:  SD←R[IR2], CAR←SBR  ; Copia segundo operando

F2IN0:  EA←M[PC]            ; Carrega a constante W
F2IN1:  PC←PC+1             ; Incrementa o PC
F2IN2:  EA←EA+R[IR1]        ; Guarda o endereço
F2IN3:  RD←M[EA]            ; Copia primeiro operando
F2IN4:  SD←R[IR2], CAR←SBR  ; Copia segundo operando
```

Programa 12.3: Micro-rotinas de carregamento de dois operandos para $S = 0$.

O primeiro conjunto de micro-rotinas, descrito no Programa 12.3, é utilizado quando o valor de S é 0. Neste caso, o modo de endereçamento aplica-se ao primeiro operando, como, por exemplo, na instrução `ADD M[R1+30],R3`. O procedimento a adoptar é semelhante ao que foi utilizado nas instruções de um operando, existindo, tal como antes, quatro casos distintos: endereçamento por registo, indirecto, imediato e indexado. A diferença consiste fundamentalmente em copiar para o registo `SD` o valor do segundo operando, por forma

```
F2RS0:   SD←R[IR1]            ; Copia segundo operando
F2RS1:   RD←R[IR2], CAR←SBR   ; Copia primeiro operando

F2RIS0:  EA←R[IR1]            ; End.  do seg.  operando
F2RIS1:  SD←M[EA]             ; Copia segundo operando
F2RIS2:  RD←R[IR2], CAR←SBR   ; Copia primeiro operando

F2IMS0:  SD←M[PC]             ; Copia segundo operando
F2IMS1:  PC←PC+1              ; Incrementa PC
F2IMS2:  RD←R[IR2], CAR←SBR   ; Copia primeiro operando

F2INS0:  EA←M[PC]             ; Carrega a constante W
F2INS1:  PC←PC+1              ; Incrementa PC
F2INS2:  EA←EA+R[IR1]         ; Guarda endereço
F2INS3:  SD←M[EA]             ; Copia segundo operando
F2INS4:  RD←R[IR2], CAR←SBR   ; Copia primeiro operando
```

Programa 12.4: Micro-rotinas de carregamento de dois operandos para $S = 1$.

a que as instruções passem a dispor dos dois operandos nos registos `RD` e `SD`. Nos casos em que tal se aplique, o registo `EA` deve, tal como anteriormente, ser carregado com o valor do endereço onde se encontrava o primeiro operando. Este valor será mais tarde utilizado pelo micro-código de escrita do resultado, descrito no Programa 12.8.

Quando o valor do bit S do registo de instrução é 1, o tratamento é algo diferente. Neste caso, o modo de endereçamento aplica-se ao segundo operando, como, por exemplo, na instrução `ADD R3,M[R4+30]`. Neste caso, o papel dos registos `RD` e `SD` deve ser trocado, e não há necessidade de guardar o endereço do operando no registo `EA`. Note-se que o modo de endereçamento continua a aplicar-se ao registo especificado em $IR1$, só que, neste caso, o campo $IR1$ codifica o segundo operando, não o primeiro. Estas micro-rotinas estão descritas no Programa 12.4.

12.3.3 Execução das Instruções

Após a execução das micro-rotinas de carregamento de operandos, o processador pode operar sobre os dados, de acordo com a operação especificada no código de instrução.

Uma vez que os operandos estão já disponibilizados nos registos `RD` e `SD`, a operação propriamente dita é, de uma forma geral, relativamente simples. No caso vertente, ilustra-se a execução de instruções usando para tal alguns exemplos que são representativos do conjunto de instruções do processador.

Como exemplo de uma instrução aritmética, considere-se a instrução `ADD`. Após chamar a micro-rotina de carregamento de operandos, usando a unidade de mapeamento, esta instrução deverá somar o conteúdo dos registos `RD` e `SD` e deixar o resultado no registo `RD`. Deverá também actualizar os bits do registo de estado, actuando para tal os bits do campo FM.

Após ter o resultado guardado no registo `RD`, o controlo deve ser transferido para a micro-rotina de escrita do resultado. Uma vez que o endereço destas

micro-rotinas está guardado nas posições 4 a 7 da memória B (ver Tabela 12.4) da unidade de mapeamento, esta transferência faz-se carregando o registo CAR com o conteúdo desta memória, endereçada com os bits mais significativos ($SR2$ e $SR1$) a 01.

```
ADD0:  CAR←ROMB[1|S|M], SBR←CAR+1          ; Copia Ops
ADD1:  RD←RD+SD, FM←Fh, CAR←ROMB[0|1|M]    ; Adição
```

Programa 12.5: Micro-programa da fase de execução da instrução ADD.

Dada a simplicidade da operação de soma, a execução da instrução propriamente dita resume-se à operação RD ← RD+SD e à actualização dos bits de estado, sendo as outras duas instruções as chamadas às micro-rotinas de carregamento de operandos e de escrita do resultado.

Um exemplo ligeiramente mais complexo é o da instrução de PUSH, descrito no Programa 12.6. Esta instrução deverá guardar o seu operando na posição de memória apontada pelo registo SP e, em seguida, decrementar o mesmo. Esta instrução não tem de escrever o resultado no seu operando, uma vez que o valor do mesmo não deve ser alterado. Desta forma, o controlo é transferido directamente para a micro-rotina de tratamento de interrupções.

```
PUSH0:  CAR←ROMB[0|0|M], SBR←CAR+1  ; Copia operando
PUSH1:  M[SP]←RD, SP←SP-1           ; Escrita
PUSH2:  CAR←IH0                     ; Salto para IH
```

Programa 12.6: Micro-programa da fase de execução da instrução PUSH.

As instruções de controlo são programadas utilizando a mesma estrutura. Neste caso, porém, estas instruções actuam directamente sobre o valor do registo PC. Por exemplo, o micro-programa que executa a instrução CALL é o do Programa 12.7.

```
CALL0:  CAR←ROMB[0|0|M], SBR←CAR+1  ; Carregar endereço
CALL1:  M[SP]←PC, SP←SP-1           ; Push do PC
CALL2:  PC←RD                       ; Carregamento do PC
CALL3:  CAR←IH0                     ; Salto para IH
```

Programa 12.7: Execução da instrução CALL.

12.3.4 Escrita do Resultado

Após a execução da instrução, o resultado deve ser escrito, em registo ou em memória, de acordo com o modo de endereçamento usado. As micro-rotinas de escrita do resultado (Programa 12.8) recebem o resultado no registo RD e escrevem-no na localização especificada pelos bits M e S do registo de instrução.

Caso o bit S seja 1, a escrita deve sempre ter lugar para um registo. O endereço deste registo é especificado directamente pela unidade de controlo do banco de registos, descrita na Figura 12.12. Caso o bit S seja 0, o valor deverá

ser escrito na posição de memória apontada pelo registo `EA`, caso o modo de endereçamento seja indirecto (M = 01) ou indexado (M = 11). Uma vez que só

```
WBR0:  R[WBR]←RD     ; Escrita em registo
WBR1:  CAR←IH0       ; Tratamento de interrupções

WBM0:  S: CAR←WBR0   ; Escrita em registo se S = 1
WBM1:  M[EA]←RD      ; Escrita do resultado
WBM2:  CAR←IH0       ; Tratamento de interrupções
```

Programa 12.8: Micro-rotina de escrita do resultado.

existem duas micro-rotinas de escrita do resultado, a tabela correspondente na memória de mapeamento B deve ser construída de tal forma que as entradas correspondentes aos modos de mapeamento indexado e indirecto por registo apontem para o micro-código com rótulo `WBM0`. A entrada nesta tabela correspondente ao modo de endereçamento imediato nunca é usado, uma vez que este modo de endereçamento não pode ser utilizado para especificar o destino de uma operação.

No caso em que o primeiro operando é especificado utilizando o modo de endereçamento por registo, a escrita do resultado é mais simples, bastando copiar o conteúdo do registo `RD` para o registo especificado na instrução *assembly*. O endereço deste registo é seleccionado directamente pelo circuito, de acordo com a Figura 12.12.

12.3.5 Teste de Interrupções

A fase final da execução de uma instrução é o teste à existência de interrupções pendentes. Nesta fase, verifica-se se o sinal $EINT$ está activo, o que significa que existe uma interrupção pendente e que o bit que assinala a disponibilidade para atender interrupções está activo. Em caso negativo, o controlo deverá ser transferido para a primeira micro-instrução da micro-rotina de carregamento de instruções, $IF0$, o que desencadeará a execução da próxima instrução.

Note-se que é possível realizar a primeira linha do Programa 12.9 numa só micro-instrução. A operação de transferência de registos, é sempre executada, enquanto que a operação de carregamento do `CAR` só é executada quando o teste ao complemento do sinal $EINT$ dá resultado verdadeiro.

Se se der início ao tratamento de uma interrupção, o registo de bits de estado e contador de programa deverão ser guardados na pilha. O bit do registo de estado que indica a disponibilidade do processador para receber interrupções deverá ser desactivado, o que se consegue carregando o registo de estado com o valor 0. Finalmente, deverá ser activado o bit IAK, indicando externamente que se vai dar início ao tratamento da última interrupção gerada.

Em resposta à activação deste sinal, o controlador de interrupções deverá colocar o vector de interrupção no barramento de dados, identificando assim o periférico responsável pela interrupção. As rotinas de tratamento às interrupções de cada periférico do sistema, para onde o processador deve passar o controlo da execução após uma interrupção de um periférico, têm os seus endereços guardados numa tabela de interrupções, com início no endereço `FE00h`.

O vector de interrupção serve como índice para esta tabela. Assim, este valor deverá ser somado a FE00h para se obter o endereço de memória onde se encontra o endereço com que deverá ser carregado no contador de programa (notar que $VECTINT$+FE00h=$VECTINT$-0200h e este estratagema é utilizado por, ao contrário de FE00h, o valor 0200h ser possível de representar no campo $CONST$ de 12 bits). Finalmente, deverá ser transferido controlo para a primeira micro-instrução da micro-rotina de carregamento de instrução, que, neste caso, executará a primeira instrução da subrotina de interrupção.

```
IH0:  R8←RE, EINT:  CAR←IF0      ; Guarda RE
IH1:  M[SP]←R8, SP←SP-1
IH2:  M[SP]←PC, SP←SP-1, IAK←1
IH3:  R9←VECTINT
IH4:  R8←0200h
IH5:  R9←R9-R8
IH6:  PC←M[R9]
IH7:  RE←R0, CAR←IF0
```

Programa 12.9: Micro-rotina de tratamento de interrupções.

12.3.6 Geração do Micro-código

Definida a estrutura dos micro-programas e conhecidos os detalhes de cada um dos blocos, basta agora definir o valor dos bits de cada micro-instrução. Considere-se, por exemplo, a primeira micro-instrução a ser executada durante a fase de carregamento de instrução:

```
IF0:  RI←M[PC]  ; Carrega RI
```

Em primeiro lugar, há que identificar o tipo de micro-instrução que poderá ser utilizado para efectuar as transferências entre registos indicadas. Neste caso, pretende-se carregar o registo de instrução com o valor da posição de memória apontada pelo contador de programa. O sinal que controla o carregamento do registo de instrução, LI, só está disponível no formato $F = 1$, o que define imediatamente o tipo de micro-instrução a utilizar.

Analisando agora o circuito de dados na Figura 12.1, pode-se observar que, para conseguir o funcionamento pretendido, é necessário garantir que:

1. O porto A do banco de registos seja endereçado com o número do registo que guarda o PC, ou seja, $15d = 1111b$.
2. O multiplexador MUXA seleccione a entrada 0, colocando o valor do PC no barramento de endereços da memória.
3. O sinal LI esteja activo.
4. Os sinais de controlo de escrita em memória e no banco de registos estejam inactivos.
5. O sinal que controla a escrita no registo de estado, LF, esteja inactivo.
6. O sinal que controla a escrita no registo SBR, LS, esteja inactivo.

Uma vez que o formato da micro-instrução é o formato 1, isso força automaticamente os valores pretendidos nos sinais que controlam a escrita na memória e o multiplexador `MUXA`.

Analisando o circuito de controlo, na Figura 12.13, verifica-se que é necessário garantir que:

1. O registo `CAR` seja incrementado. Para tal, é necessário:
   - Controlar o multiplexador `MUX5` com o valor 00
   - Colocar 0 no controlo do multiplexador `MUX4`, seleccionando a entrada 0 do multiplexador `MUXCOND` e colocando o sinal CC a 1, por forma a que $CCOND$ seja 0.
2. O multiplexador `MUXAD` deverá seleccionar a sua entrada RAD, que deverá tomar o valor $1111b$, uma vez que se pretende este valor em $SelAD$.

Chega-se assim à conclusão que, nesta micro-instrução, os seguintes valores deverão estar definidos:

$F = 1$
$M5 = 00$
$MCOND = 000$
$CC = 1$
$LI = 1$
$LF = 0$
$LS = 0$
$WR = 0$
$MAD = 1$
$RAD = 1111$

Estes valores definem a micro-instrução ilustrada na Figura 12.15, onde os valores que não são relevantes foram deixados em branco. Arbitrando agora

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

1	M5	SR1	SR2	LS	MCOND	CC	LI	LF	CONST/NA	WR	MD	MAD	RAD
1	00			0	000	1	1	0		0		1	1111

Figura 12.15: Codificação da micro-instrução `RI←M[PC]`.

que os valores não relevantes são colocados a 0, obtém-se finalmente o valor que os bits deverão ter para esta micro-instrução: `8060001Fh`.

Um exemplo ligeiramente mais complexo permite-nos ilustrar a utilização de uma micro-instrução do tipo F=0 e o uso das memórias de mapeamento. Considere-se então a micro-instrução:

```
IF1:  PC←PC+1, CAR←ROMA[OPCODE]
```

No circuito de dados, é necessário controlar a unidade aritmética por forma a que esta efectue um incremento, o que significa colocar $CULA$ igual a 00101 e seleccionar o `PC` no porto A do banco de registos. É também necessário seleccionar a entrada 0 do multiplexador `MUXA` e a entrada `0` do multiplexador `MUXD`.

Finalmente, é necessário activar o sinal de escrita nos registos WR e garantir que todos os outros sinais de escrita em registos estão desactivados.

Na unidade de controlo é necessário seleccionar o valor 1111 no sinal $SelAD$ assim como forçar $M5$ a 10, por forma a que a saída da memória de mapeamento A seja seleccionada. É ainda necessário garantir que o sinal IAK fica a 0.

Estas considerações conduzem à definição dos valores dos bits descritos na Figura 12.16. Arbitrando, como anteriormente, que os campos não definidos

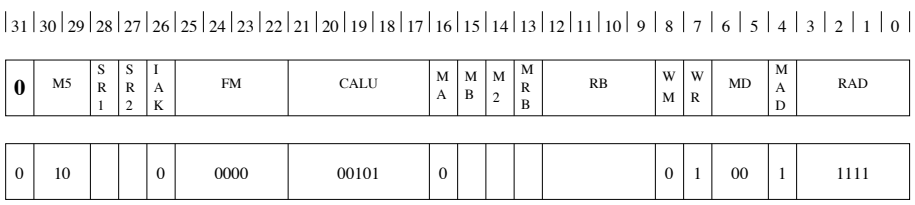


Figura 12.16: Micro-instrução `PC ← PC+1, CAR ← ROMA[OPCODE]`.

são preenchidos com o valor 0, obtém-se o valor final para esta micro-instrução, `400A009Fh`.

Sumário

Este capítulo descreveu a estrutura interna do processador P3, um processador micro-programado de 16 bits, cujo conjunto de instruções tinha sido estudado no capítulo anterior.

As duas componentes mais importantes deste processador são o circuito de dados e a unidade de controlo. O circuito de dados é constituído por um banco de registos e uma unidade lógica e aritmética, já estudadas em capítulos anteriores, além dos registos de instrução e de estado e dos diversos barramentos de interligação. A unidade de controlo é baseada num micro-sequenciador que gera a sequência de sinais que controlam o funcionamento do circuito de dados. O micro-sequenciador usa uma unidade de teste de condições, uma unidade de mapeamento e uma unidade que controla o banco de registos.

A parte final do capítulo foi dedicada a estudar a forma como se definem os micro-programas que controlam o funcionamento deste micro-processador e a forma como é realizada cada uma das suas instruções.

Capítulo 13

Sistemas de Memória

Nos capítulos anteriores, foi examinado o funcionamento de um processador, sendo a memória encarada como um conjunto uniforme de registos, cada um dos quais endereçável individualmente. Esta visão simplista não corresponde à realidade excepto em sistemas muito simples mas permitiu descrever o funcionamento do processador sem entrar em linha de conta com as complexidades do sistema de memória.

Em particular, qualquer computador actual, quer seja um sistema embarcado, um computador pessoal ou um servidor, utiliza numerosos dispositivos para armazenamento de dados. Estes dispositivos consideram-se usualmente divididos em duas grandes classes: memória primária e memória secundária.

A *memória primária*, também referida como *memória principal*, construída com base em circuitos integrados, tem um tempo de acesso mais curto, mas é mais cara (por bit) e de menor dimensão, não tendo a capacidade de guardar os dados quando o sistema se encontra desligado.

A *memória secundária*, também conhecida como *memória auxiliar*, que recorre a dispositivos baseados em tecnologia magnética (discos e fitas magnéticas) ou ópticas (CDROMs e DVDs) é mais lenta, mas tem um custo por bit menor, maior capacidade e a possibilidade de conservar os dados mesmo sem o computador estar ligado. Esta memória, além de poder ser utilizada pelo processador na execução de programas, através do uso de *memória virtual*, é também usada para guardar dados em sistemas de ficheiros. Esta utilização da memória secundária não será estudada neste capítulo, uma vez que implica um conhecimento de sistemas operativos que vai para além do âmbito de um livro introdutório.

Neste capítulo, será analisado o sistema de memória de um computador, começando pela organização da memória primária, que, em alguns sistemas simples, representa a única forma de armazenamento de dados.

Sistemas mais complexos exibem outras componentes, que serão estudados nas secções seguintes. Em particular, serão estudados os sistemas de *cache*, que têm como objectivo acelerar os acessos a memória primária, e os sistemas de *memória virtual*, que permitem utilizar a memória secundária de forma transparente para o processador.

13.1 Organização de Sistemas de Memória

No Capítulo 7 foi analisada a forma como um conjunto de registos pode ser organizado em bancos e em módulos de memória. Um módulo de memória é constituído por um conjunto de registos (de um ou mais bits) e um sistema de descodificação que permite seleccionar um e apenas um dos registos, para operações de leitura ou escrita. Estes registos são, em geral, constituídos por dispositivos com comportamentos análogos aos das básculas actuadas por nível.

A memória principal de um computador, na sua versão mais simples, consiste num conjunto (ou banco) de módulos de memória, organizados por forma a serem visíveis pelo processador como um conjunto de posições onde dados podem ser escritos ou lidos.

Na prática, a organização do sistema de memória pode ser relativamente complexa, dado que, tipicamente, o espaço de endereços é utilizado por diversos dispositivos de memória, e também, em alguns processadores, como é o caso do P3, pelo sistema de entradas/saídas.

A secção seguinte descreve a forma como a memória de um computador é construída através da interligação de diversos módulos e como diversos tipos de memórias podem ser interligados entre si por forma a partilharem o espaço de endereçamento do processador.

13.1.1 Planos de Memória

Em geral, não é possível encontrar um único circuito integrado de memória que satisfaça, só por si, as necessidades de um dado sistema computacional que se pretende projectar. Assim, é necessário utilizar diversos módulos de memória ligados entre si, por forma a oferecer a funcionalidade e capacidade desejada.

Módulos de memória comerciais têm, para além das linhas de endereço, uma ou mais linhas que permitem seleccionar esse módulo, tipicamente denominadas CS (*chip select*) ou CE (*chip enable*). Quando este sinal não está activo, as saídas de dados do módulo de memória estão no estado de alta impedância, não sendo possível aceder às memórias. Isto permite ligar, de forma muito simples, diversos módulos de memória por forma a conseguir criar memórias de capacidade e configurações arbitrárias.

A Figura 13.1 ilustra as interligações disponibilizadas por um módulo de memória simples, com uma entrada de selecção CS. As linhas de endereço permitem endereçar uma das posições de memória do módulo, e as linhas de dados são ligadas ao barramento de dados do processador. A entrada de $R/\overline{W}$ selecciona o modo desejado de funcionamento. Quando esta linha está activa, a memória está em modo de leitura e disponibiliza para o exterior os dados da posição especificada nas linhas de endereços. Quanto esta linha está a `0`, a memória é colocada em modo de escrita e escreve na posição especificada o valor que se encontra nas linhas de dados. Módulos de memória deste tipo podem ser interligados por forma a conseguir uma variedade de configurações de memória.

Existem fundamentalmente dois tipos de interligação de módulos de memória: uma interligação que aumenta o número de bits de cada posição de memória e uma interligação que aumenta o número de posições de memória.

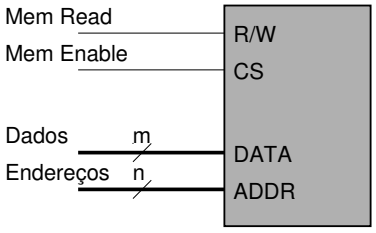


Figura 13.1: Esquema de um módulo de memória.

O primeiro tipo de ligação, ilustrado na Figura 13.2, permite compor diversos módulos de memória, por forma a conseguir um número de bits em cada posição superior ao que é disponibilizado por cada módulo de memória isoladamente. Neste tipo de ligação, as palavras devolvidas por cada um dos módulos são concatenadas numa palavra de maior dimensão.

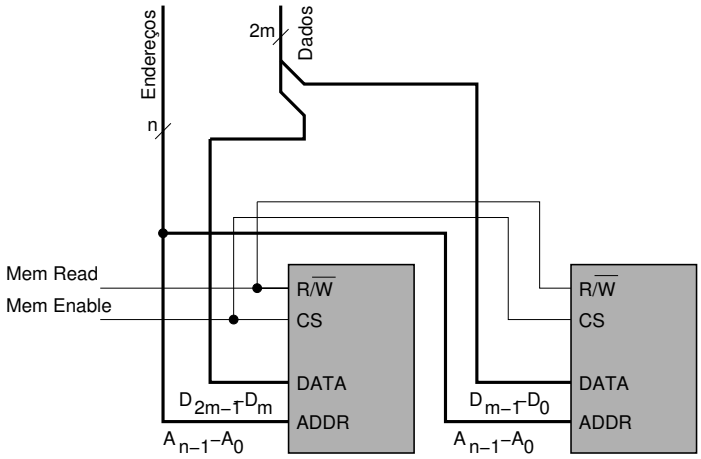


Figura 13.2: Ligação de módulos de memória por forma a aumentar o número de bits em cada posição de memória.

O segundo tipo de ligação, ilustrado na Figura 13.3, permite ligar dois módulos de memória, por forma a conseguir um número de posições de memória superior ao que é fornecido por cada módulo individualmente. Neste caso, o barramento de endereços é constituído pelas linhas que entram directamente nos módulos e pela linha que é usada para seleccionar o módulo activo.

Os esquemas anteriores de interligação podem ser combinados entre si, por forma a obter um sistema de memória com as características desejadas.

A utilização de um exemplo concreto ilustra este mecanismo com maior clareza. Suponha-se que existem disponíveis módulos de memória de 16k octetos (ou seja, 2^{14} posições de memória com 8 bits cada) e que se pretende construir um sistema de memória com 64k octetos, organizado em 2^{16} posições de memória de 8 bits cada uma. Neste caso, a interligação de quatro módulos, tal como está exemplificado na Figura 13.4 disponibiliza a funcionalidade pretendida. Note-se que é uma interligação do mesmo tipo que a da Figura 13.3, onde, portanto, as linhas de dados são interligadas entre si e é usado um des-

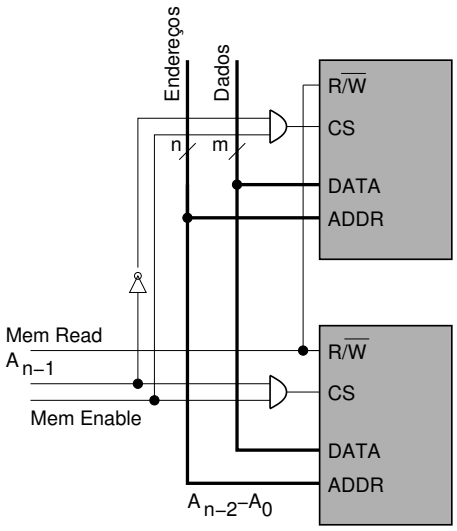


Figura 13.3: Ligação de módulos de memória por forma a aumentar o número de posições de memória disponíveis.

codificador para seleccionar qual dos módulos de memória deverá responder a um dado endereço. De referir que esta interligação das linhas de dados é possível por estar garantido à partida, pelo próprio funcionamento do descodificador, que apenas um dos circuitos de memória está activo em cada acesso à memória.

Na Figura 13.5 estão ilustradas as interligações necessárias para combinar 8 módulos de memória do mesmo tipo do utilizado no exemplo anterior por forma a obter uma memória de 128k octetos, organizada em 64k palavras de 16 bits cada. Neste caso, aos 14 bits de endereço que são enviados directamente para as memórias, juntam-se os dois bits que controlam o descodificador, o que conduz a um total de 16 linhas de endereço.

13.1.2 Mapas de Memória

Foi analisado na secção anterior como diversos módulos de memória podem ser interligados entre si por forma a formarem uma memória da dimensão desejada.

Nesta secção, será estudada a forma como as memórias são interligadas com o processador, por forma a disponibilizarem o espaço de endereçamento desejado pelo projectista.

Admita-se uma situação muito simples, em que se pretende utilizar o módulo de memória da Figura 13.4 para disponibilizar um espaço de endereçamento de 64k octetos num processador que tem 20 bits de endereço e tem um barramento de dados de 8 bits. Note-se que, embora a memória possa ter uma estrutura interna complexa, ela apresenta-se para o processador de uma forma semelhante ao do módulo simples da Figura 13.1, neste caso com 16 linhas de endereço e 8 de dados.

Uma vez que o processador tem um espaço de endereçamento total de 2^{20}

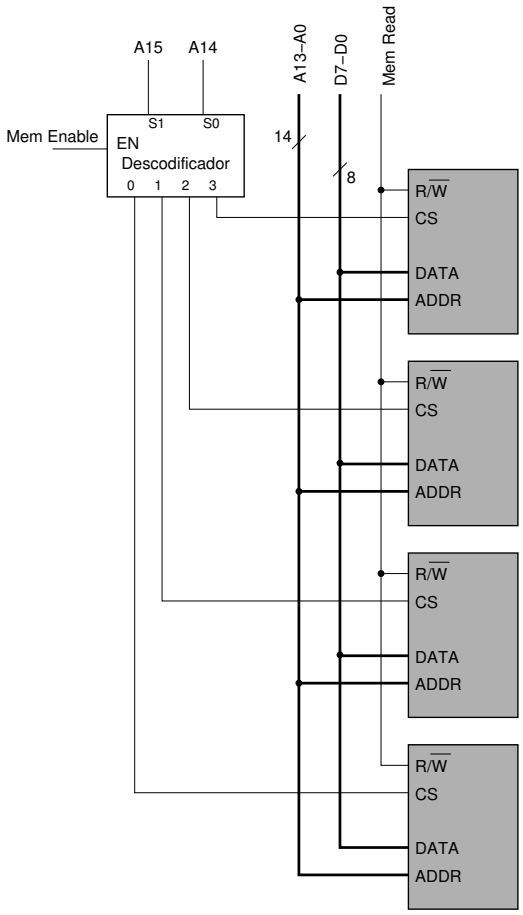


Figura 13.4: Esquema de uma memória de 64k octetos, construída com quatro módulos de 16k octetos cada.

octetos, de `00000h` a `FFFFFh` (superior à capacidade do módulo de memória), o projectista tem, em primeiro lugar, de decidir o espaço de endereçamento que deverá corresponder à memória. Suponha-se que a opção do projectista é a de que as 2^{16} posições de memória deverão estar localizadas nas posições `F0000h` a `FFFFFh`, isto é, no topo do espaço de endereçamento utilizável pelo processador.

À correspondência que é necessário realizar entre endereços de memória e módulos é habitual chamar-se o *mapa de memória*. Muitas vezes, este mapa é ilustrado de forma gráfica, como é o caso da Figura 13.6, que corresponde ao exemplo em estudo. Para conseguir que a memória fique visível nesta zona do espaço de endereçamento, há que criar um circuito de descodificação, ilustrado na Figura 13.7. A porta lógica utilizada faz com que apenas os endereços que têm os 4 bits mais significativos a 1 activem a memória.

Em geral, os mapas de memória de um processador são mais complexos e não são usadas portas lógicas individuais, mas sim descodificadores ou ROMs para seleccionar módulos individuais.

Quando se projecta o circuito que faz a descodificação dos endereços e a

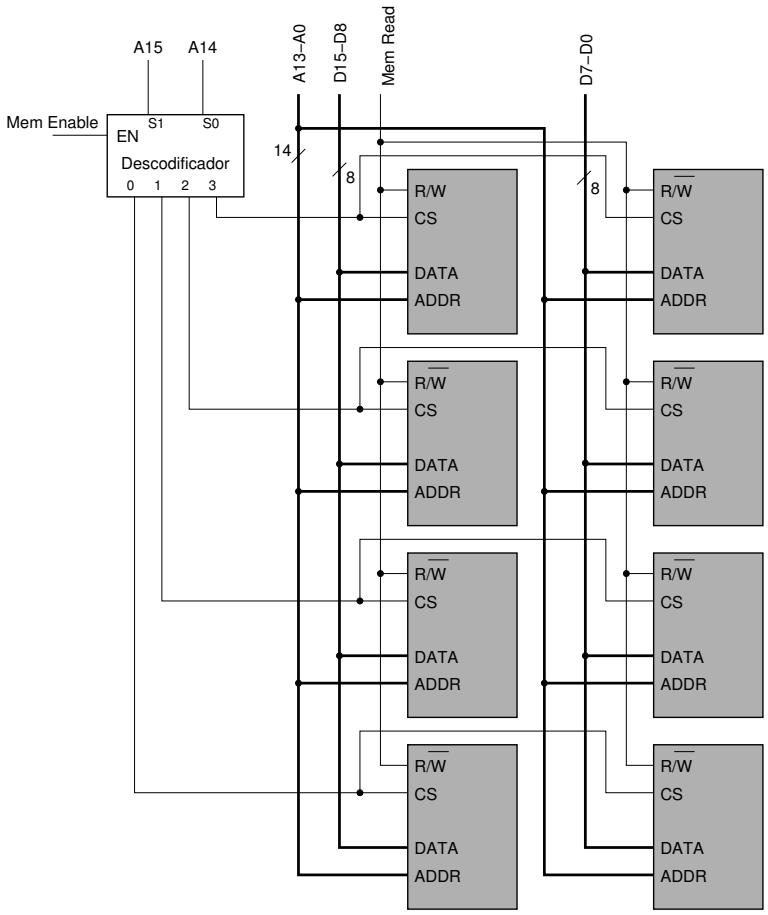


Figura 13.5: Esquema de uma memória de 128k octetos, organizada como 64k palavras de 2 octetos cada.

geração dos sinais de controlo dos módulos de memória, há que garantir que:

1. Cada módulo de memória é seleccionado quando o endereço corresponde a uma posição de memória guardada nesse módulo e existe uma operação de acesso à memória.

2. Os bits que são usados para endereçar as palavras dentro do módulo são adequadamente controlados pelas linhas correspondentes do barramento de endereços.

Considere-se, por exemplo, um sistema baseado no mesmo processador do exemplo anterior, em que se pretende realizar o mapa de memória da Figura 13.8. Neste mapa de memória, existe uma memória ROM de 32k octetos na parte baixa da memória, e duas áreas de RAM não contíguas, ambas de 64k octetos. A realização da descodificação deste sistema de memória poderia também ser feita com portas lógicas discretas, mas isso resultaria num circuito relativamente complexo.

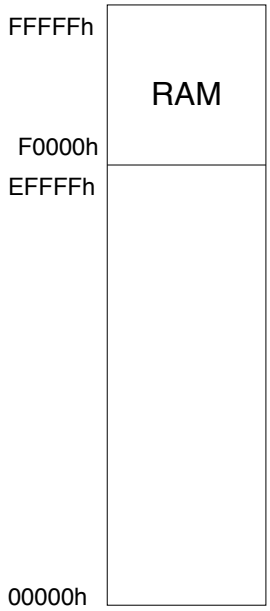


Figura 13.6: Mapa de memória de um processador um espaço de endereçamento de 2^{20} octetos, mas com apenas 64k octetos de memória RAM instalada entre as posições `F0000h` e `FFFFFh`.

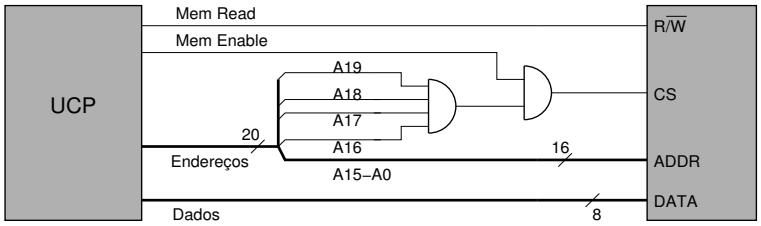


Figura 13.7: Circuito correspondente ao mapa de memória da Figura 13.6.

Se se utilizar um descodificador, controlado pelos 4 bits mais significativos do endereço, é possível utilizar as suas saídas para controlar, directamente, os bits de selecção das memórias RAM. O controlo da ROM é ligeiramente mais complexo, uma vez que é preciso assegurar que a ROM é seleccionada apenas quando os 5 bits mais significativos são 0 e a operação é uma operação de leitura. A Figura 13.9 ilustra a realização do circuito de descodificação que corresponde ao mapa de memória da Figura 13.8.

13.1.3 Geração dos Sinais de Controlo

Até agora, assumiu-se que uma memória, logo que seleccionada em modo de leitura, coloca na sua saída o valor das posições de memória desejadas. Na prática, o funcionamento de uma memória é ligeiramente mais complexo, uma vez que há que adaptar o funcionamento da memória a diversas velocidades de funcionamento dos processadores e dos barramentos de controlo.

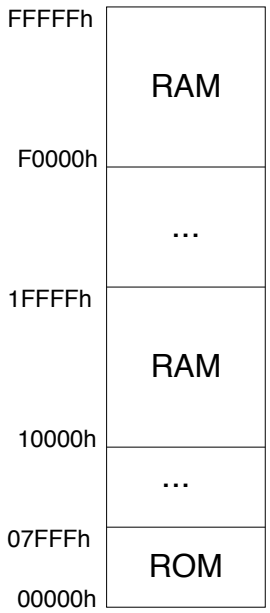


Figura 13.8: Mapa de memória de um processador com RAM e ROM.

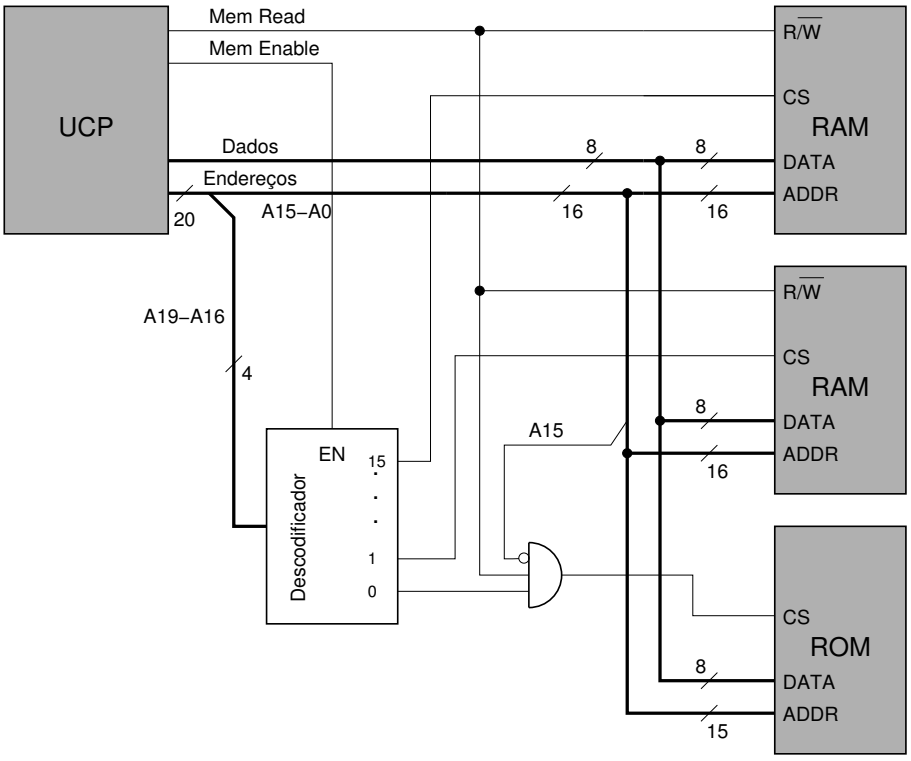


Figura 13.9: Circuito correspondente ao mapa de memória da Figura 13.8.

Com efeito, enquanto que um processador pode funcionar com ciclos de relógio de cerca de 1ns, os tempos de acesso a memória variam, conforme os tipos de memória, entre poucos nano-segundos para as RAMs estáticas e as dezenas de nano-segundos, para as RAMs dinâmicas, de maior capacidade. Isso significa que um módulo de memória tipicamente não consegue disponibilizar os dados num ciclo de relógio do processador, o que obriga à existência de um protocolo para transferência dos dados.

A forma mais simples de garantir que os dados são lidos correctamente consiste em projectar o sistema de forma que o processador espere o tempo suficiente para que a operação de leitura ou escrita se conclua com sucesso.

Suponha-se, a título de exemplo, que um processador que trabalha a 100 MHz (o que corresponde um ciclo de relógio de 10ns) deve comunicar com uma memória que tem um tempo de acesso, medido a partir da activação do sinal CS, de 25ns. Neste caso, há que garantir que, numa operação de leitura, o processador espera três ciclos de relógio antes de ler os dados provenientes da memória. De igual forma, numa operação de escrita, deverá manter os dados válidos durante três períodos de relógio, a partir do momento em que seleccionou a memória.

A Figura 13.10 ilustra as formas de onda de uma operação de escrita num sistema com esta configuração. Após activação do sinal que selecciona o módulo de memória respectivo, os dados deverão manter-se válidos no barramento durante 3 ciclos de relógio do processador.

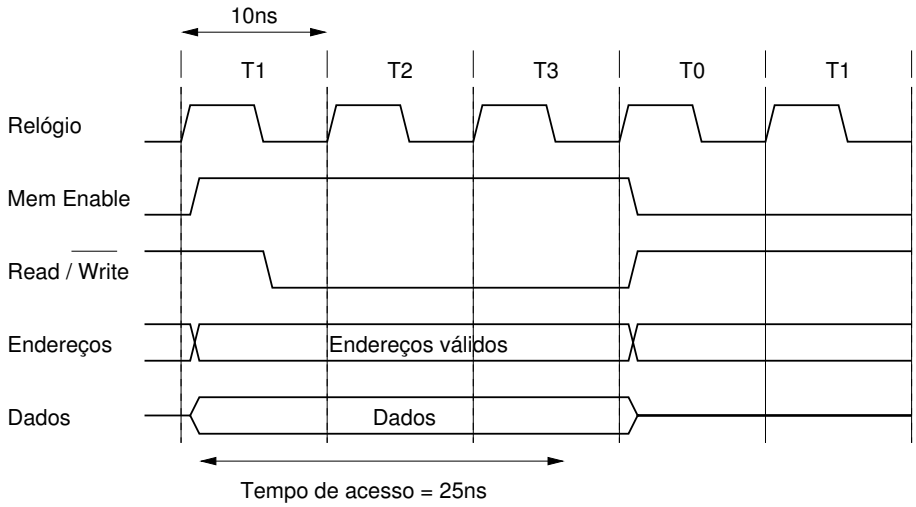


Figura 13.10: Formas de onda para uma operação de escrita em memória.

A operação de leitura funciona de forma similar. Neste caso, os dados apenas ficam garantidamente válidos no barramento depois de passar um tempo igual ao tempo de acesso à memória, pelo que o processador só pode ler os dados ao fim do terceiro ciclo de relógio.

As formas de ondas das Figuras 13.10 e 13.11 ilustram os acessos efectuados por um processador a memórias estáticas, que, tipicamente, têm apenas as linhas de controlo CS e $R/\overline{W}$. A realidade de um computador actual é, no entanto, bastante mais sofisticada.

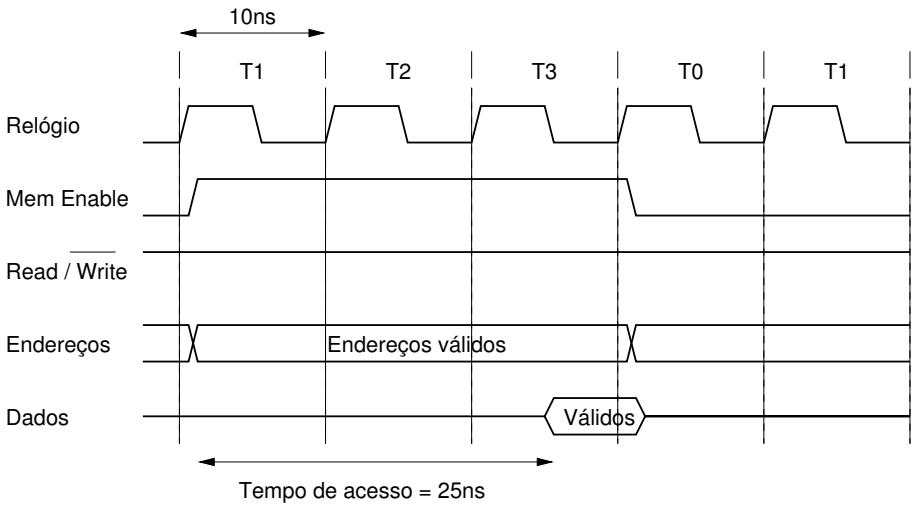


Figura 13.11: Formas de onda para uma operação de leitura de memória.

Por um lado, as memórias utilizadas como memória primária são, na maioria dos casos, memórias dinâmicas, cujo controlo é significativamente mais complexo do que o das memórias estáticas. Estas memórias são mais baratas (por bit) e exibem diversos modos de acesso, que permitem, entre outras coisas, a transferência rápida de blocos de dados. O estudo detalhado das formas de controlo de memórias está fora do âmbito de um texto introdutório, pelo que não será prosseguido aqui.

Por outro lado, o acesso de um processador não é feito a uma memória simples, mas a uma hierarquia de memórias, como se descreve em seguida.

13.2 Hierarquia de Memória

Por razões que se prendem com o desempenho do sistema, a estudar nesta secção, a memória de um computador tem, na maior parte dos casos, diversos níveis, ilustrados na Figura 13.12.

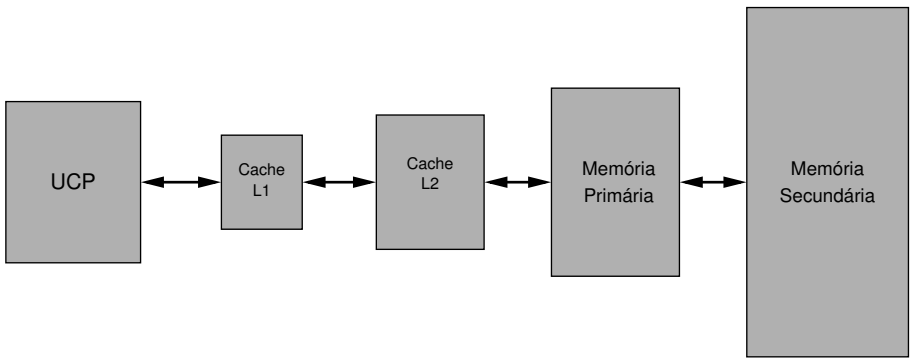


Figura 13.12: Hierarquia de memória num computador.

Do ponto de vista do desempenho do processador, a situação ideal corresponde a ter a maior quantidade de memória disponível possível a funcionar à velocidade mais rápida possível. Porém, como o custo por bit e a velocidade de funcionamento estão inversamente relacionados, a solução adoptada passa, geralmente, pelo uso de diversos tipos de memória, com diferentes velocidades de acesso. O sistema de memória está estruturado por forma a que os dados e instruções mais comummente utilizados, em cada passo da execução de um programa, estejam em memórias mais rápidas, enquanto que os menos frequentemente acedidos estejam em memórias mais lentas.

Na sequência, irá ser analisada uma hierarquia de memória simples, onde existe apenas um nível de cache, uma memória principal de grande capacidade e uma memória secundária que é usada como memória virtual. No caso geral, podem existir vários níveis de cache, mas isso não altera o mecanismo de funcionamento.

Quando é feito um acesso a uma posição de memória, quer para leitura quer para escrita, o sistema começa por verificar se essa posição está disponível na memória mais rápida, que é denominada de cache. Se essa posição não estiver disponível na cache, é consultada a memória principal, que é geralmente do tipo RAM dinâmica. Se a posição também não estiver disponível na memória principal, é feito um acesso à memória secundária, maior e mais lenta, e geralmente a funcionar com base em discos magnéticos.

Se o sistema de memória estiver bem estruturado, este sistema hierárquico é apenas ligeiramente mais lento, em média, que um sistema de memória em que a totalidade da memória funcionasse à velocidade da cache. O custo total, porém, é muito menor, uma vez que as memórias mais lentas têm um custo por bit muito inferior ao das memórias mais rápidas.

Como é possível conseguir um objectivo próximo do original (memória muito grande e muito rápida) investindo apenas uma fracção do montante que seria necessário se toda a memória fosse rápida? A possibilidade de obter este resultado prende-se com um facto que se verifica em praticamente todos os programas, e que é conhecido pelo *princípio da localidade*. Este princípio baseia-se na verificação empírica de que os acessos a memória feitos por um processador são altamente correlacionados entre si, e não são distribuídos uniformemente pela memória disponível. Existem dois tipos de localidade:

- *Localidade temporal*: se é feito um acesso a um determinado endereço de memória, é muito provável que haja um novo acesso a este mesmo endereço num futuro próximo.
- *Localidade espacial*: se é feito um acesso a um determinado endereço de memória, é muito provável que haja um novo acesso a um endereço próximo deste.

A localidade temporal pode ser utilizada para acelerar os acessos a dados e instruções, mantendo nas memórias mais rápidas as posições de memória que foram utilizadas mais recentemente. Com efeito, uma fracção muito significativa do tempo de execução de um programa é gasta em ciclos de dimensão relativamente pequena, onde são executadas repetidamente as mesmas instruções e re-utilizadas as mesmas variáveis.

A localidade espacial pode ser utilizada com o mesmo fim, copiando para memórias mais rápidas posições que estão, espacialmente, próximas daqueles

que foram recentemente utilizados. A localidade espacial nos dados resulta do uso de estruturas de dados, como, por exemplo, *arrays*, onde os valores são acedidos em sequência. A localidade espacial nas instruções deriva do modo de funcionamento de um microprocessador, que executa as instruções de uma forma essencialmente sequencial.

O aproveitamento da localidade (espacial e temporal) está na base do uso de caches, que têm como objectivo reduzir os tempos de acesso do processador à memória primária. A maioria dos computadores actuais utilizam também um sistema de gestão de memória denominado memória virtual, que utiliza o princípio da localidade para permitir a utilização pelos programas de uma quantidade de memória superior àquela que existe, fisicamente, no computador.

A conjugação destas duas técnicas, caches e memória virtual, cuja realização será estudada em detalhe na Secção 13.3 e Secção 13.4, respectivamente, leva a que, do ponto de vista da unidade central de processamento, esteja disponível virtualmente uma memória muito rápida, de capacidade muito superior à memória física presente no computador.

13.2.1 Caches

A função de uma cache num sistema de memória é permitir que a maioria dos acessos a memória sejam feitos mais rapidamente do que é possível utilizando directamente a memória primária do computador. Este conceito de uma memória pequena e mais rápida pode ser generalizado para incluir vários níveis de cache, como descrito na Figura 13.12. Os níveis de cache mais próximos do processador são feitos com memórias mais rápidas, mas de menor capacidade. Os níveis mais exteriores são realizados com memórias progressivamente mais lentas.

A memória primária do computador é, tipicamente, realizada com memórias dinâmicas, cujos tempos de acesso se encontram tipicamente na gama dos 60ns a 70ns. As memórias de cache encontram-se muitas vezes no mesmo circuito integrado que o processador, e têm assim tempos de acesso compatíveis com a velocidade do processador. Em processadores modernos, é comum existirem diversos níveis de cache internos ao processador, que podem ser ou não complementados com caches exteriores. Estas caches podem ter tempos de acesso da ordem da duração do ciclo de relógio do processador, que, com a presente tecnologia, é menor que 1ns.

Numa operação de leitura, a função da cache é disponibilizar dados e/ou instruções num período de tempo mais curto do que o que é possível quando o acesso é feito à memória principal. Numa operação de escrita, a cache deve poder ser acedida rapidamente para guardar os dados, devendo os mesmos ser posteriormente escritos em memória.

Uma vez que a cache tem menor dimensão que a memória principal, só uma pequena fracção da totalidade do conteúdo da memória está presente, em cada momento, na cache. Assim, é necessário substituir frequentemente dados e instruções presentes na cache por outros, que estão a ser mais recentemente acedidos.

Para quantificar a aceleração que é possível conseguir através do uso de uma cache, considere-se um exemplo simplificado de um sistema computacional que tem apenas um nível de cache e uma memória primária.

Para usar um exemplo concreto, admita-se que um dado processador funciona a 500 MHz, que faz um acesso à memória de dados em cada instrução e que cada instrução demora 4 ciclos de relógio a ser executada. Nestas condições, o processador faz um acesso à memória de dados em cada 8ns[1].

Admita-se que a memória primária tem um tempo de acesso médio de 70ns, enquanto que a cache, realizada com uma tecnologia semelhante à do processador, disponibiliza os dados em 2ns, o que corresponde ao período de relógio do processador. Admita-se ainda que, neste sistema, os acessos a memória têm um padrão tal que 95% dos acessos feitos a memória podem ser substituídos por um acesso à cache, ou seja, que a cache exibe um *taxa de sucesso* (em inglês, *hit-rate*) de 95%.

Nestas condições, é possível calcular o número de ciclos de relógio que demora, em média, a executar cada instrução, neste sistema e num sistema semelhante mas em que não fosse utilizada a cache.

Cada instrução demora 4 ciclos de relógio a ser executada, dos quais um ciclo é de acesso à memória. O tempo médio para um acesso à memória é dado por

$$\overline{T}_{\text{mem}} = 0,95 \times 2 + 0,05 \times 70 = 5,4\text{ns} \tag{13.1}$$

O número médio de ciclos de relógio por cada instrução executada é assim de $3 + 5,4/2,0$, ou seja, $5,7$ (esta medida de desempenho é denominada de *CPI*, ciclos por instrução, ou, em inglês, *cicles per instruction*). Significa isto que este processador executa uma instrução a cada $5,7$ ciclos de relógio, ou seja, a cada $11,4$ns. Este processador executa assim $87,7$ milhões de instruções por segundo (ou *MIPS*, outra medida de desempenho de processadores - em inglês, *millions of instructions per second*).

Se não existisse uma cache, cada acesso a memória teria de ser feito à memória primária. Isto significa que o processador demoraria, para executar cada instrução, $3 + 70/2,0$ ciclos de relógio, ou seja, 38 ciclos de relógio. O processador executaria assim apenas $13,2$ MIPS, o que significa que ficaria aproximadamente 7 vezes mais lento.

Este exemplo ilustra bem a importância das caches no desempenho de um sistema computacional, e mostra como a inclusão de uma memória pequena e rápida entre o processador e a memória principal aumenta significativamente a velocidade com que as instruções são executadas.

Nestas análises, não foi focado o aspecto crítico da dimensão das caches, mas apenas a sua velocidade. A dimensão das caches, no entanto, é crítica para um bom funcionamento do sistema de memória. O impacto da dimensão aparece, indirectamente, no valor da taxa de sucesso, que é maior para caches de maior dimensão, e menor para caches mais pequenas. Em geral, no dimensionamento de uma cache para um sistema de memória, é necessário considerar dois factores fundamentais:

- A velocidade da cache, que condiciona a velocidade máxima de funcionamento do sistema.
- A dimensão da cache, que condiciona a taxa de sucesso da mesma, e, consequentemente, o número de acessos à memória principal.

[1] Ignore-se, por agora, que o processador precisa também de aceder às instruções, que também estão guardadas em memória.

A melhor solução provém de um compromisso entre estes dois factores.

No caso do processador estudado no capítulo Capítulo 12, admitiu-se que é possível executar um acesso a memória em cada ciclo de relógio. Assim, a memória representada na Figura 12.1 representa, numa arquitectura típica, a cache mais interna do processador, que se assumiu ter um tempo de acesso não superior ao ciclo de relógio do processador.

Por questões de clareza na exposição, foram omitidos na descrição do processador, as linhas de controlo necessárias para fazer a interface com o controlador de cache. Destas, a mais importante é um sinal de $READY$, que deverá ser testado em cada leitura da cache. Este sinal, gerado pelo controlador da cache, indica que a leitura se encontra concluída e que os dados pretendidos se encontram disponíveis no barramento de dados. No caso em que os dados não se encontram presentes na cache, o processador deverá esperar antes de prosseguir o processamento. No caso particular da arquitectura estudada no Capítulo 12, o processador deverá continuar a executar a mesma micro-instrução até que o sinal de $READY$ fique activo. A utilização de um linha de $READY$ é, geralmente, necessária quando são usadas caches um acesso à cache, pois este é muito mais demorado quando os dados não estão disponíveis na mesma e têm de ser recuperados da memória principal.

Numa operação de escrita, a situação é um pouco mais simples. Neste caso o processador pode prosseguir imediatamente, desde que seja possível guardar os dados em cache ou o controlador disponha de um registo que guarde os dados temporariamente. Esse é geralmente o caso, embora os detalhes de funcionamento do controlador possam ser complexos. Este assunto será estudado com algum detalhe na Secção 13.3.4.

13.2.2 Memória Virtual

De forma a que o processador não fique com um espaço de endereçamento limitado à quantidade de memória instalada no sistema, é definido um espaço de endereçamento virtual com uma dimensão tipicamente muito superior à memória primária. Assim, todo o processamento da UCP é feito utilizando *endereços virtuais*, em oposição aos *endereços físicos* que permitem o acesso à memória primária e à cache.

Sempre que o processador faz um acesso à memória, o endereço que coloca no barramento de endereços será portanto um endereço virtual. Para efectuar esta leitura ou escrita da memória, será necessário primeiro traduzir este endereço virtual para um endereço físico. Esta tarefa está a cargo de uma unidade especial no computador, a *Unidade de Gestão de Memória*, *UGM* (em inglês, *Memory Management Unit*, ou *MMU*). Logo, a sequência de operações no acesso à memória será[2]:

1. a UCP coloca no barramento de endereços o endereço virtual onde pretende aceder;

2. a UGM traduz este endereço para um endereço físico;

[2]Em certos sistemas, a cache funciona directamente com os endereços virtuais, situação em que os pontos 2 e 3 estão invertidos

3. é realizado um acesso à cache para testar se este endereço físico lá se encontra; caso se encontre na cache, o acesso de leitura ou escrita termina aqui;

4. caso contrário, é realizado um acesso à memória primária.

Naturalmente, como o espaço virtual é muito maior que o espaço físico, nem todas as posições de memória virtuais terão um correspondente endereço físico. Neste caso, não é possível realizar a tradução pela UGM, sendo necessário fazer um acesso ao disco, local onde se encontra armazenado o espaço de endereçamento virtual que não cabe em memória primária. Neste acesso ao disco, transferem-se para memória primária os dados correspondentes aos endereços virtuais acedidos. De facto, uma forma de olhar para este sistema é pensar na memória primária como uma cache para o espaço de endereçamento virtual, sendo válidas as observações feitas na secção anterior.

O tempo de acesso ao disco, da ordem das dezenas de milisegundos, é muito superior ao tempo de acesso à memória primária, que, como se viu, é da ordem das dezenas de nanosegundos. Portanto, existe um factor de 1 milhão de diferença entre os tempos de acesso à memória primária e secundária. Assim, sempre que um dado endereço virtual não se encontra em memória primária há uma grande penalidade em termos de desempenho do processador. Felizmente, este impacto é reduzido pois, devido aos princípios da localidade temporal e espacial, a taxa de faltas nos acessos a posições de memória virtuais é muito baixa. Um valor típico para a taxa de faltas é da ordem de 0,0001%, ou seja, apenas um em cada um milhão de acessos não se encontra em memória primária.

Com os valores utilizados na Equação 13.1, tempos de acessos à memória e à cache de 70ns e 2ns, respectivamente, e uma taxa de sucesso na cache de 95%, e assumindo que o acesso ao disco demora 10ms (ou 10^7ns), com uma taxa de faltas de endereços virtuais de 0,0001% (ou 10^{-6}), o tempo de acesso à memória passa a ser[3]:

$$\overline{T}_{\text{mem}} = 0,95 \times 2 + 0,05 \times (0,999999 \times 70 + 0,000001 \times 10^7) = 5,9\text{ns} \quad (13.2)$$

Portanto, 95% das vezes o acesso continua a ser resolvido pela cache. Dos restantes 5% dos casos, em que é necessário fazer um acesso à memória primária, 99,9999% das vezes o acesso é feito apenas à memória primária, e em apenas 0,0001% dos casos é necessário fazer um acesso ao disco. Como se pode observar, a grande penalidade dos acessos ao disco é diluída pelo baixo número de acessos a este.

De qualquer forma, sempre que acontece uma falta a UCP não fica, em geral, bloqueada à espera da leitura do disco, o que poderia significar o desperdício de milhões de instruções. Nos computadores actuais, existem em geral vários *processos* a executar-se concorrentemente. Isto significa que, embora a cada instante só se esteja a executar um destes processos, existe um conjunto de outros processos à espera da sua vez. O tempo de processamento é assim dividido, de forma controlada pelo sistema operativo, entre estes diferentes processos. Se o processo que se está a executar encontrar uma falta de endereço virtual, este processo é bloqueado e é colocado em execução um dos processos em espera. Desta forma, o carregamento da informação do disco ocorre

[3]Para simplificar, foi aqui propositadamente ignorado o tempo de tradução do endereço virtual para físico por parte da UGM.

durante a execução útil de outro processo, diminuindo-se significativamente o despedício do tempo de processamento.

13.3 Organização de Sistemas de Cache

Como foi visto atrás, existem dois tipos de localidade de acessos: localidade temporal e localidade espacial. Para aproveitar a localidade temporal, a cache deve guardar dados que foram recentemente acedidos. Para aproveitar a localidade espacial, a cache deverá guardar dados que se encontram próximos, em termos de endereço, de dados recentemente acedidos.

As diversas alternativas que existem para arquitecturas de cache empregam diferentes soluções para conseguir aproveitar, da melhor forma, cada uma destas características dos acessos. Para tornar mais clara a discussão que se segue, assuma-se um sistema com uma hierarquia de memória que consiste em apenas dois níveis: um primeiro nível, a cache, e um segundo nível, a memória primária. Na prática, o nível seguinte poderá não corresponder efectivamente à memória primária mas apenas a um segundo nível de cache. Isso, no entanto, não afecta o comportamento do primeiro nível de cache, que será analisado em seguida. Ignore-se também para já a memória secundária, assumindo-se portanto que todos os endereços são endereços físicos.

Existem fundamentalmente quatro graus de liberdade que afectam a escolha de uma arquitectura para o sistema de cache.

O primeiro é a forma como o espaço de endereçamento da memória principal é mapeado para o espaço mais reduzido de endereçamento da cache. Uma vez que a cache tem necessariamente um menor número de posições que a memória, há que definir um processo através do qual se possa mapear cada posição de memória primária para uma posição da cache.

O segundo aspecto prende-se com o dimensionamento dos blocos de cache, que tem como objectivo fazer o melhor uso possível da localidade espacial dos acessos a dados. Um bloco de cache representa a quantidade mínima de octetos que são carregados da memória principal para a cache, quando é necessário efectuar um carregamento.

O terceiro aspecto é a forma como são substituídos dados na cache, por forma a maximizar a exploração que é feita das localidades temporais.

O quarto aspecto está relacionado com a forma como as operações de escrita são tratadas. Uma operação de escrita é fundamentalmente diferente de uma operação de leitura porque o processador pode prosseguir a execução de um programa sem que a operação de escrita tenha terminado e porque afecta dados em memória que podem ser acedidos por outros dispositivos.

13.3.1 Mapeamento de Dados em Caches

De uma forma geral, o mapeamento de um espaço de endereçamento de maior dimensão (que corresponde à memória principal) para um espaço de endereçamento mais pequeno (a cache) baseia-se em considerar apenas um subconjunto dos bits de endereço.

Um endereço de 32 bits deverá assim ser visto como partido em duas partes: o *índice* (*index*) e a *etiqueta* (*tag*). A Figura 13.13 ilustra a partição de um endereço de 32 bits em duas partes, para uma cache que disponibilize 1024 posições.

A dimensão da etiqueta depende do tamanho máximo de memória disponível, e só corresponde aos restantes 22 bits porque se assumiu que o espaço total de endereçamento é, como foi referido, de 2^{32} octetos. Quando é feita uma leitura

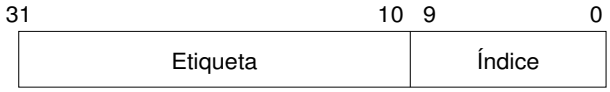


Figura 13.13: Campos etiqueta e índice.

a um dado endereço, o subconjunto de bits que corresponde ao índice é usado para endereçar a cache. Nessa posição da cache estará guardado o valor que se encontra guardado na posição de memória pretendida.

Porém, uma vez que um número de endereços diferentes irá corresponder a uma mesma posição na cache, é importante poder distinguir se o dado guardado nessa posição da cache corresponde efectivamente à posição de memória pretendida. Isso consegue-se guardando na cache não só os dados, mas também o campo etiqueta do endereço correspondente aos dados que lá estão.

Quando é feito o acesso à cache, usando apenas os bits de índice, basta comparar o campo etiqueta da posição de memória a que se quer aceder com a etiqueta que está guardada na cache. Se as etiquetas forem iguais, o conteúdo dessa posição de cache é o conteúdo pretendido. A Figura 13.14 ilustra de forma esquemática o mecanismo de funcionamento de uma cache deste tipo.

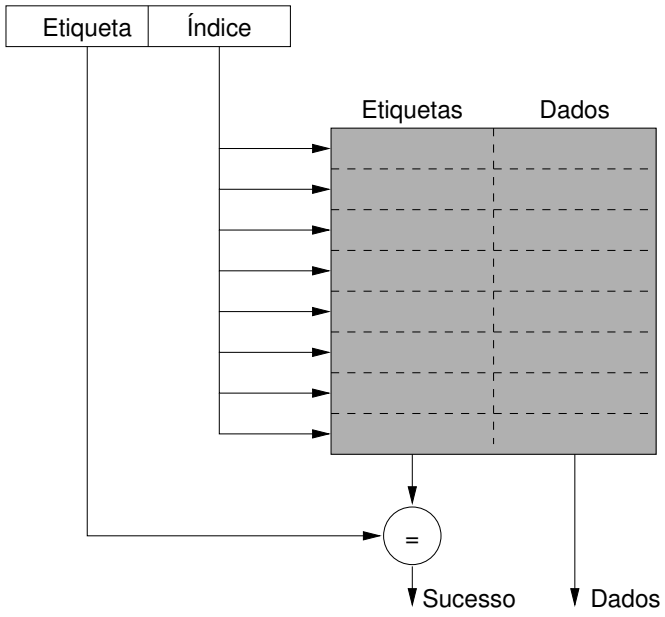


Figura 13.14: Esquema de funcionamento de uma cache de mapeamento directo.

Se as etiquetas forem diferentes, os dados que estão na cache não são os pretendidos e será necessário efectuar uma leitura da memória principal para obter os dados correctos. Uma vez que, por causa da localidade temporal, estes

dados tem grande probabilidade de vir a ser usados dentro de pouco tempo, há que enviar os dados não só para o processador, mas também para esta posição da cache, e actualizar o valor da etiqueta.

Este tipo de mapeamento, a que se chama *mapeamento directo*, não representa a única possibilidade de mapear o espaço total de endereçamento para um conjunto mais reduzido de posições de cache.

As caches de mapeamento directo são as mais simples. No entanto, são as mais restritivas pois, como cada posição de memória só pode estar numa dada posição da cache, o desempenho será muito mau se o programa aceder consecutivamente a duas posições de memória que, por azar, calhem na mesma posição da cache.

Uma alternativa às caches de mapeamento directo são as caches *completamente associativas*. As caches associativas são uma consequência da seguinte observação: o ideal, de um ponto de vista de flexibilidade da cache, é que uma dada posição de memória possa ser colocada em qualquer posição da cache. Neste caso, o endereço é interpretado na totalidade como uma etiqueta, não existindo o campo índice. A identificação da posição correcta da cache faz-se, unicamente através da comparação das etiquetas. Se existir uma posição da cache que contenha uma etiqueta com valor igual à etiqueta do endereço que está a ser acedido, então essa posição de cache contém o valor pretendido.

Para realizar estas caches usam-se *memórias associativas*, cujo funcionamento é diferente das habituais. Numa memória associativa o valor que é usado para endereçar a memória não corresponde a uma posição, mas sim ao conteúdo de uma posição de memória. Cada posição nas memórias associativas tem não só os dados, mas também uma etiqueta. O endereço que é colocado à entrada de uma memória associativa é comparado simultaneamente com as etiquetas de todas as posições de memória, sendo disponibilizados os dados que correspondem à posição onde a etiqueta coincide com o valor usado para endereçar a memória. A Figura 13.15 ilustra o esquema interno de uma memória totalmente associativa.

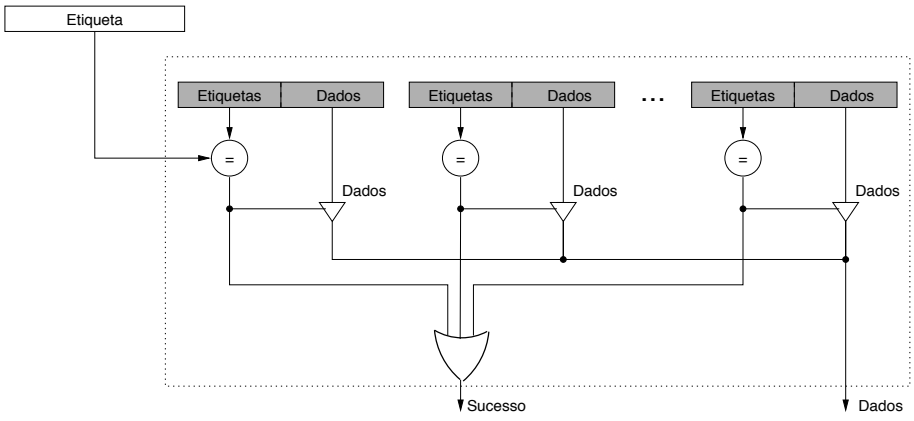


Figura 13.15: Esquema de uma memória totalmente associativa.

Apesar de mais flexíveis, estas caches são mais lentas e mais complexas do que as de mapeamento directo devido à necessidade de ser feita uma comparação simultânea de todas as posições da memória com a etiqueta do endereço.

Por essa razão, caches de dimensão significativa não são, geralmente, completamente associativas. Existe, no entanto, uma forma de obter muitas das vantagens do mapeamento totalmente associativo mantendo a velocidade de acesso e o custo de uma cache de mapeamento directo.

Uma cache de *mapeamento associativo por conjuntos* consiste num conjunto de caches de mapeamento directo, endereçadas em paralelo. Todas elas são acedidas, em simultâneo, com o campo índice do endereço, mas apenas aquela que tiver a etiqueta correcta disponibiliza os dados. Se existirem n caches de mapeamento directo em paralelo, uma dada posição de memória pode ser mapeada para qualquer uma n caches, na posição que corresponde ao valor do índice. A este tipo de cache chama-se *cache de mapeamento associativo de n vias*.

Estes três tipos de caches podem ser vistos como pertencendo todas ao mesmo tipo, variando apenas o número de vias de associatividade. Numa cache de mapeamento directo, o número de vias de associatividade é 1. Numa cache totalmente associativa, o número de vias de associatividade é M, onde M é o número de posições da cache. O endereço é partido em dois campos, cuja dimensão varia com o tamanho das caches e o número de vias de associatividade. Em particular, o número de bits no índice é igual a $\log_2 \frac{M}{n}$, onde M é a dimensão da cache e n o número de vias de associatividade.

Em caches que tenham diversas vias de associatividade, a escrita de novos dados em cache tem de ser antecedida da escolha da via que irá conter esses dados. Com efeito, qualquer das vias poderá ser escolhida, dependendo a decisão da política de substituição, que irá ser estudada na Secção 13.3.3.

Por exemplo, uma cache de 4096 posições usará 12 bits de índice se for de mapeamento directo, 10 bits de índice se for associativa de 4 vias e nenhum bit de índice se for totalmente associativa. Neste último caso, o número de vias de associatividade é igual ao número de posições na cache.

13.3.2 Blocos de Cache

Uma cache em que apenas seja carregada uma posição de memória de cada vez não faz uso da localidade espacial. Com efeito, se, imediatamente a seguir for acedida pela primeira vez a posição de memória seguinte, esta não se encontrará em cache. No entanto, pelo princípio da localidade espacial, é muito provável que a posição de memória $i + 1$ seja acedida em breve se acabou de ser feito um acesso à posição de memória i.

As caches fazem uso desta característica dos padrões de acesso mapeando para a cache blocos de posições consecutivas de memória, e não posições individuais. Estes conjuntos de posições de memória, cuja dimensão varia de cache para cache, chamam-se *blocos de cache* ou *linhas de cache*. O uso de blocos de cache está também relacionado com uma maior eficiência nos acessos a memória primária, que, tipicamente, disponibilizam mecanismos de acesso mais rápidos para conjuntos de posições consecutivas.

Quando uma cache funciona por blocos, que é o caso mais comum, o endereço deve ser visto como dividido em três partes: a etiqueta, o índice e o *deslocamento* dentro do bloco. O campo deslocamento tem um número de bits suficiente para endereçar uma posição individual dentro de um bloco e é usado para seleccionar qual das posições do bloco deverá ser transferida para o processador.

Tal como anteriormente, os bits do índice são usados para seleccionar qual dos blocos poderá conter a posição de memória pretendida. O número de blocos na cache passa a ser igual à dimensão da cache dividida pela dimensão de cada bloco. O número de bits no índice passa assim a ser igual a $\log_2 \frac{M}{nB}$, onde B é a dimensão de cada bloco, e, tal como anteriormente, M é a dimensão da cache e n o número de vias de associatividade. Uma cache com a mesma dimensão do exemplo anterior, com 4096 posições, que use blocos de 16 posições, usará apenas 8 bits de índice se for de mapeamento directo. Com efeito, as 4096 posições correspondem a 256 blocos de 16 posições cada, sendo cada um dos blocos endereçado pelo campo de 8 bits do índice. A partição do endereço nos campos de etiqueta, índice e deslocamento para este exemplo está ilustrada na Figura 13.16.

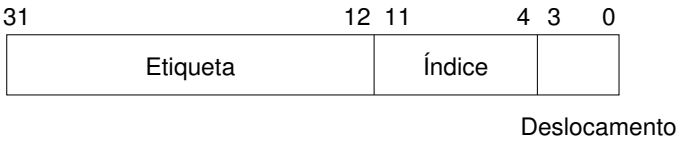


Figura 13.16: Campos etiqueta, índice e deslocamento para o exemplo do texto.

A Figura 13.17 ilustra a correspondência que existe entre as posições de uma memória primária de 64M e os blocos de uma cache com estas características. Note-se que a dimensão das etiquetas não se altera com o uso de blocos na

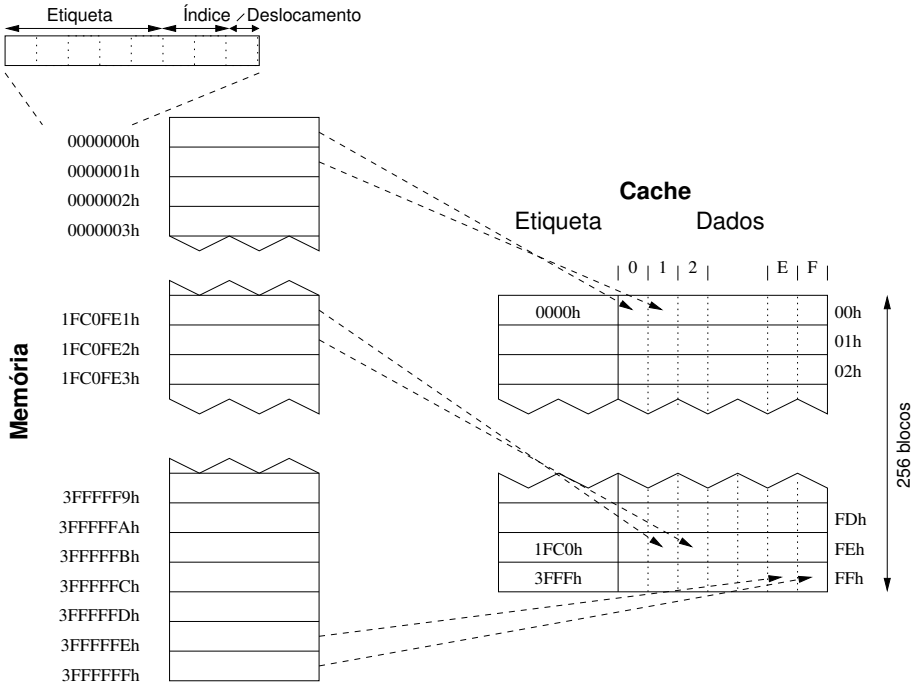


Figura 13.17: Exemplo de mapeamento de uma memória de 64M octetos para uma cache de mapeamento directo de 4k octetos com blocos de 16 octetos.

cache, uma vez que o número de bits de índice é menor, mas os bits retirados ao índice são utilizados para endereçar dentro do bloco de cache. Porém, passa a ser necessária apenas uma etiqueta para cada conjunto de B posições na cache, o que significa que existe uma penalização mais pequena pela necessidade de guardar as etiquetas, além dos dados propriamente ditos.

É agora possível perceber a estrutura interna de uma cache de mapeamento directo, que se encontra representada na Figura 13.18. A cache consiste num conjunto de blocos, cada um dos quais contém diversas posições de memória e um conjunto de bits que guarda a etiqueta que corresponde ao referido bloco.

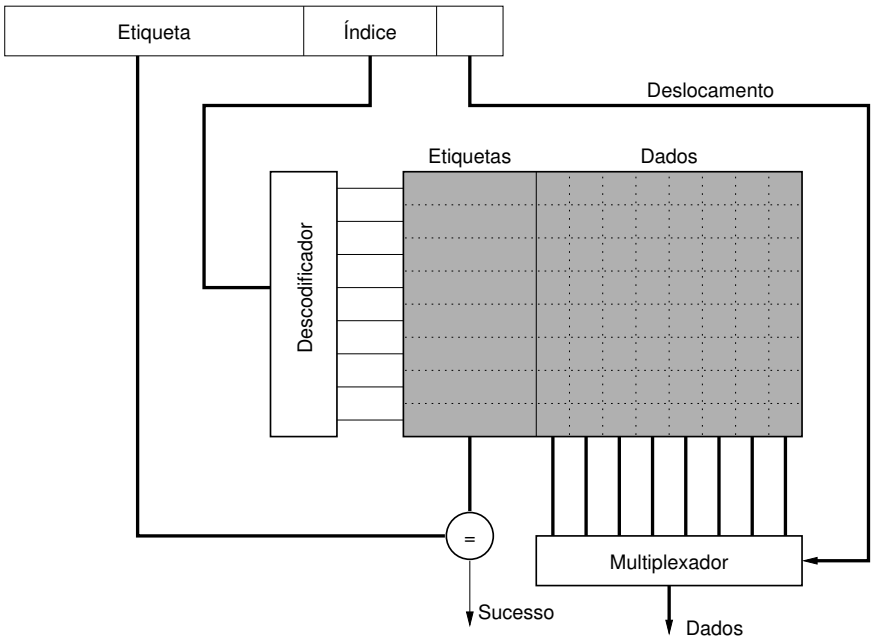


Figura 13.18: Estrutura de uma cache de mapeamento directo.

Os bit de endereço são usados de forma diferente, conforme o campo a que pertencem. Numa primeira fase, os bits de índice são usados para endereçar, através de um descodificador, o bloco de cache que poderá conter os dados. Numa segunda fase, a etiqueta guardada junto ao bloco seleccionado é comparada com a etiqueta do endereço pretendido. Se a comparação der um resultado positivo, a posição de memória pretendida reside no bloco escolhido, sendo seleccionada pelo campo deslocamento do endereço e enviada para o processador.

Caches com mais vias de associatividade funcionam de forma semelhante. A Figura 13.19 descreve a estrutura de uma cache com duas vias de associatividade. Neste caso, os bits de índice seleccionam dois blocos, um de cada um dos dois conjuntos de blocos. A comparação com as etiquetas é feita em paralelo para cada uma das etiquetas nas posições indicadas pelo campo índice. Apenas uma destas comparações poderá dar um resultado positivo, uma vez que a gestão da cache garante que uma posição de memória nunca está presente simultaneamente em mais do que uma posição da cache. O uso de *buffers* de três-estados simplifica a lógica que escolhe qual das vias deverá enviar os

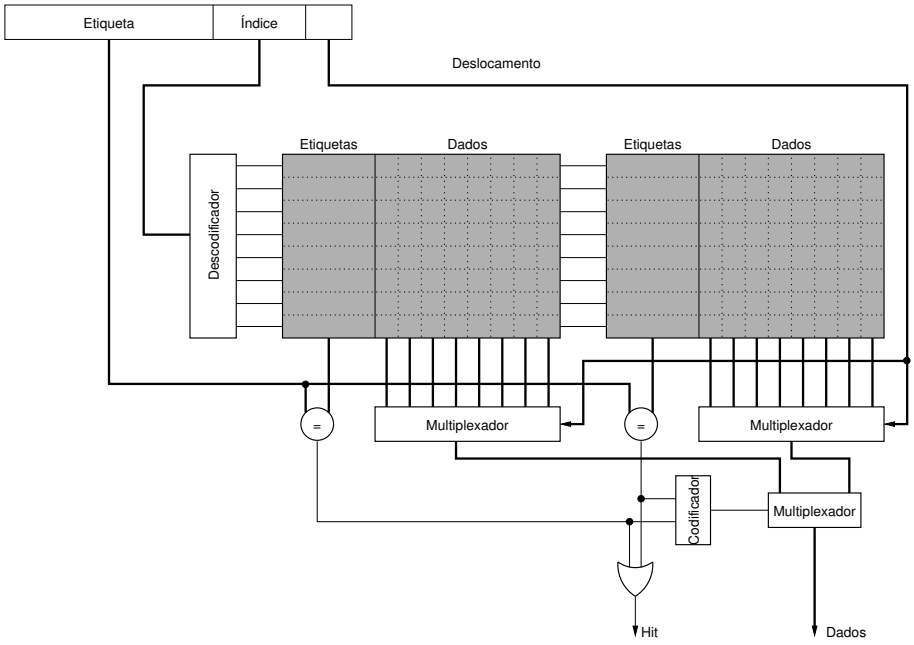


Figura 13.19: Estrutura de uma cache com duas vias de associatividade.

dados para a cache. Se não for usada lógica três-estados, será necessário usar um codificador cuja saída controla um multiplexador que selecciona a via que deverá estar activa, como representado na Figura 13.19.

13.3.3 Políticas de Substituição

Nas caches de mapeamento directo não existe necessidade de tomar qualquer decisão quando uma dada posição de memória não está presente na cache. Com efeito, uma vez que se pretendem guardar na cache os dados mais recentemente acedidos, é necessário fazer uma substituição dos dados na cache sempre que existe uma falta na cache. Uma vez que os dados pretendidos só podem ir para uma posição na cache, o conteúdo dessa posição terá de ser substituído.

Nas caches que têm duas ou mais vias de associatividade, há que decidir em qual das vias irão ser guardados os novos dados, já que isso implica deitar fora dados que já estão na cache. Em geral, a melhor solução consiste em substituir os dados que foram acedidos há mais tempo, pois de acordo com o princípio da localidade temporal, estes serão os que têm menos probabilidade de virem a ser acedidos num futuro próximo. Esta política de substituição é denominada de *LRU* (do inglês, *least recently used*). No entanto, em caches com muitas vias de associatividade, esta política é difícil de implementar em hardware, uma vez que obrigaria a manter informação sobre a altura em que foi acedido cada um dos blocos de memória e a determinar, de forma muito rápida, qual a via usada há mais tempo.

Existem diversas alternativas para resolução deste problema. Uma possibilidade, que na prática tem um comportamento bom, é simplesmente selec-

cionar de forma aleatória uma das vias como aquela que contém o bloco que deverá ser substituído. Uma outra solução bastante eficaz, mas ligeiramente mais complexa, consiste em manter um contador, para cada conjunto de n blocos de cache, que é incrementado sempre que o seu conteúdo é igual ao número do bloco que é acedido. Desta forma, blocos frequentemente acedidos têm menos probabilidade de serem substituídos, uma vez que o contador permanece pouco tempo a apontar para eles.

13.3.4 Políticas de Escrita

Até agora analisou-se o comportamento de uma cache quando o processador executa instruções de leitura. Nas operações de escrita, os compromissos são significativamente diferentes. Existem, fundamentalmente, dois graus de liberdade no que respeita às operações de escrita.

O primeiro grau de liberdade prende-se com a forma como é tratada uma operação de escrita quando a posição de memória que é escrita se encontra em cache. Nas caches *write-back*, a escrita é feita somente para a cache. Nas caches *write-through*, a escrita é feita simultaneamente na cache e na memória.

Uma vantagem das caches *write-back* é que a escrita é executada à velocidade da cache, não havendo necessidade de esperar pela escrita em memória. Esta vantagem, no entanto, não é tão grande como possa parecer, uma vez que o processador não precisa de ficar parado à espera que termine a operação de escrita, ao contrário do que acontece com uma operação de leitura. Uma outra vantagem das caches *write-back* é a possível redução do tráfego no barramento causado por escritas na memória, uma vez que duas escritas para uma posição na cache não geram escritas separadas na memória. Porém, a esta vantagem corresponde uma desvantagem na altura em que é necessário substituir o bloco de cache, uma vez que a memória tem de ser actualizada com os dados alterados que estão na cache. Isto pode atrasar significativamente a operação de leitura que causa a substituição do bloco, ou, em alternativa, torna a estrutura da cache consideravelmente mais complicada.

O segundo grau de liberdade está relacionado com a acção a tomar quando existe uma escrita para um bloco que não está em cache. Pode ter-se como política que o bloco deverá ser copiado de memória para cache (caches *write-allocate*) ou, pelo contrário, que os dados deverão ser escritos directamente em memória, sem que o bloco seja copiado para cache.

Geralmente, as caches *write-back* são do tipo *write-allocate*, enquanto que muitas das caches *write-through* são do tipo *no-write-allocate*. No entanto, outras opções são possíveis, sendo necessário analisar o impacto no desempenho de cada uma das opções, face ao perfil típico de execução dos programas.

13.3.5 Bits de Controlo

Num bloco de cache, para além dos dados propriamente ditos e da etiqueta, existe necessariamente um conjunto de bits de controlo que mantém informação diversa.

No caso de caches *write-back*, um desses bits é necessariamente o *dirty bit*. Este bit indica se os dados nesse bloco de cache foram alterados e deverão ser escritos para a memória quando esse bloco de cache for necessário para outros dados.

Outro bit de controlo presente em todas as caches é o *valid bit*, que indica se os dados que estão nessa posição da cache são válidos ou não. Existem diversas razões pelas quais os dados podem deixar de estar válidos. O computador pode estar a começar a executar um programa, situação em que todos os dados estão inválidos. Podem também existir outros dispositivos que alterem posições de memória, situação em que a cópia em cache fica inválida. Em sistemas operativos que possam executar mais de um programa de cada vez (*multi-tasking*), os blocos de cache poderão também ser marcados como inválidos quando existe uma mudança de contexto.

Poderão existir outros bits de controlo, que controlam o funcionamento da cache em outros aspectos. Por exemplo, poderão existir bits de controlo que permitem realizar a política de substituição, ou bits que permitam implementar mecanismos de protecção de memória. O número e a função desses bits varia de sistema para sistema.

13.4 Memória Virtual

As caches, que são os elementos da hierarquia de memória mais perto do processador, têm como objectivo primário acelerar a velocidade de acesso a dados por parte do processador. No outro extremo da hierarquia existe a memória secundária, tipicamente constituída por discos magnéticos, cuja função é aumentar a dimensão da memória vista pelo processador para além da dimensão da memória primária instalada no computador.

A memória principal e o disco são dispositivos bastante diferentes. No entanto, em muitos sistemas, é transparente para o processador (e para o programador) qual a localização física dos dados que são utilizados. Um programa pode aceder a dados guardados num espaço de endereçamento que é muito maior que a memória primária disponível. A disponibilização de um espaço virtual de endereçamento maior que o fisicamente disponível é conseguida utilizando endereços virtuais.

Quando um sistema disponibiliza memória virtual, todos os acessos feitos pelo processador à memória são feitos com endereços virtuais. Este sistemas dispõem de uma Unidade de Gestão de Memória, UGM (em inglês, *Memory Management Unit*, ou MMU) que traduz o endereço virtual para um endereço físico de memória, caso o endereço virtual corresponda a uma posição de memória. Se o endereço virtual corresponder a uma posição que se encontra, nesse momento, guardada em disco, o sistema vai buscar essa posição de memória ao disco e copia-o para a memória principal.

Para facilitar esta tradução de endereços, e para optimizar a transferência de dados de e para memória secundária, os espaços virtual e físico são divididos em grupos contíguos de endereços, a que se chamam, nesta caso, *páginas*. A dimensão de uma página, naturalmente igual nos espaços virtuais e físicos, é, tipicamente, significativamente maior que a dimensão de um bloco de cache, uma vez que a transferência de dados entre memória secundária e primária incorre numa penalização muito maior do que a transferência entre memória principal e cache. A dimensão de uma página de memória virtual varia, tipicamente, entre 1k octetos e 16k octetos, dependendo do sistema.

Sempre que é feito um acesso a um endereço virtual, a UGM verifica se a página correspondente a esse endereço está em memória ou não. Se não es-

tiver, diz-se que houve uma *falta de página* (em inglês, *page fault*) e a página é carregada para memória. O carregamento de uma página, desencadeado através da execução de uma rotina específica para o efeito, pode demorar diversos milisegundos, tempo suficiente para executar milhões de instruções num processador moderno. Isto significa que o processador fica livre para outras tarefas, tais como executar outros programas (em sistemas multi-tarefa) ou atender a pedidos pendentes.

O processo de tradução de endereços virtuais para endereços físicos usa diversas estruturas físicas e lógicas no seu funcionamento. As mais importantes são a *tabela de páginas* e a *TLB* (do inglês, *Translation Look-Aside Buffer*).

13.4.1 Tabelas de Páginas

Um dado endereço virtual especifica uma posição no espaço de endereçamento virtual. Tal como nas caches, este endereço pode ser decomposto em diversos campos, que, no caso de memória virtual, são dois: o número da página e o deslocamento dentro da página.

O endereço físico pode também ser visto como repartido nestes dois campos. Uma vez que as páginas têm a mesma dimensão, o deslocamento dentro da página é igual no endereço físico e no endereço virtual. Já o número de página pode exigir um número de bits diferente, uma vez que as dimensões do espaço de endereçamento físico e virtual são, tipicamente, bastante diferentes.

Desta forma, a tradução de endereços virtuais para físicos consiste, fundamentalmente, em traduzir o número da página virtual para um número de página física.

Tabela de Páginas Plana

O esquema mais simples de tradução baseia-se no uso de uma tabela (ver Figura 13.20), com um número de entradas igual ao número de páginas virtuais. Em cada entrada dessa tabela encontra-se guardado um *descritor de página* (em inglês, *Page Table Entries* ou *PTE*) que inclui informação sobre o endereço físico da página, ou informação que permite localizar a referida página em disco, possivelmente através do uso de tabelas auxiliares.

Quando é feito um acesso a uma página de memória virtual, a tabela é consultada para verificar se a referida página existe em memória física. Em caso afirmativo, o endereço físico é construído através da concatenação da entrada correspondente na tabela com o deslocamento dentro de página. Em caso negativo, existe uma falta de página, e é chamada uma sub-rotina do sistema operativo para copiar a página para memória principal.

Este esquema simples funciona bem apenas quando o espaço de endereçamento virtual não é demasiado grande. Considere-se, por exemplo, um sistema que disponha de um espaço de endereçamento virtual de 2^{32} octetos e uma memória física de 256M octetos. Assuma-se ainda que a dimensão das páginas é de 4k octetos. Este sistema dispõe de um total de 2^{20} ($2^{32}/2^{12}$) páginas virtuais, o que significa que a tabela de páginas tem de ter 2^{20} entradas, qualquer que seja a quantidade de memória virtual que esteja a ser utilizada por um programa.

Cada entrada na tabela de páginas terá de ter a dimensão suficiente para identificar qual das páginas em memória física corresponde à página virtual

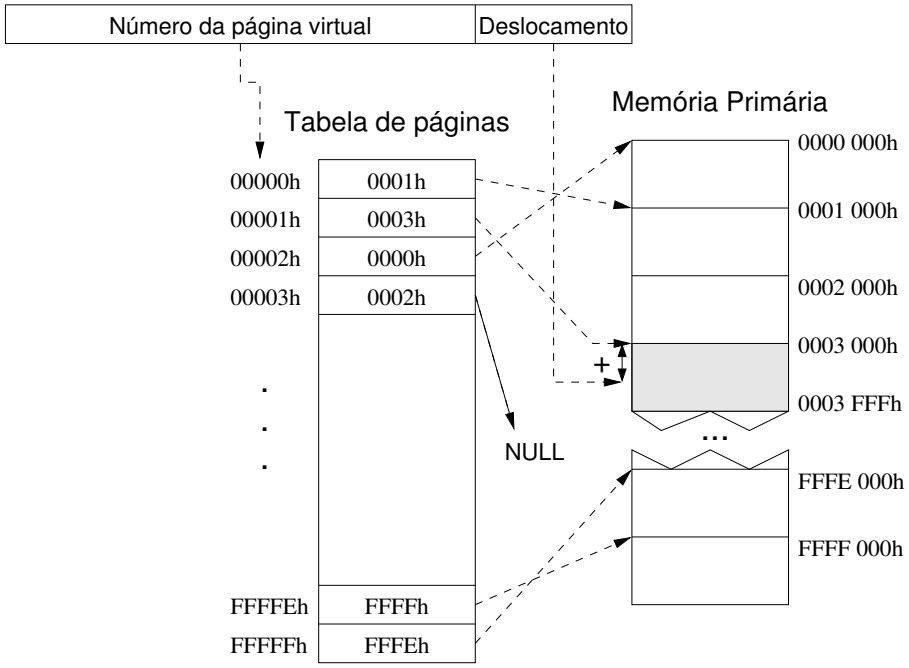


Figura 13.20: Tabela para tradução de endereços virtuais.

que se pretende aceder. A memória física tem capacidade para 2^{16} páginas ($2^{28}/2^{12}$). Por simplicidade, admita-se que todas as páginas são alinhadas em memória, o que significa que o primeiro endereço de cada página tem os últimos 12 bits a 0. Neste caso, basta guardar os 16 bits mais significativos do endereço na tabela de páginas, logo cada descritor de página necessita de, pelo menos, dois octetos.

Isto significa que uma tabela desta dimensão irá ocupar um espaço de memória pelo menos igual a 2M octetos, qualquer que seja a dimensão do programa que está a ser executado. Isto representa um uso muito ineficiente de recursos, especialmente no caso em que o programa que está a ser executado usa pouca memória.

Tabela de Páginas Hierárquica

A utilização de *tabelas hierárquicas* elimina a ineficiência que foi apontada nas tabelas planas, tornando, no entanto, o acesso aos dados um pouco mais lento.

Numa tabela de páginas hierárquica de dois níveis, o primeiro nível da tabela de páginas é constituído por uma única tabela, denominada de *directório*. O directório contém referências para tabelas do segundo nível que contêm, essas sim, os endereços físicos das páginas. Este esquema, ilustrado na Figura 13.21, obriga a que sejam feitos dois acessos à memória para recuperar o endereço físico das páginas. Em compensação, as tabelas do segundo nível apenas precisam de estar presentes quando as correspondentes páginas de memória estão efectivamente a ser utilizadas.

Considere-se o mesmo caso do exemplo anterior. Numa tabela plana, a ta-

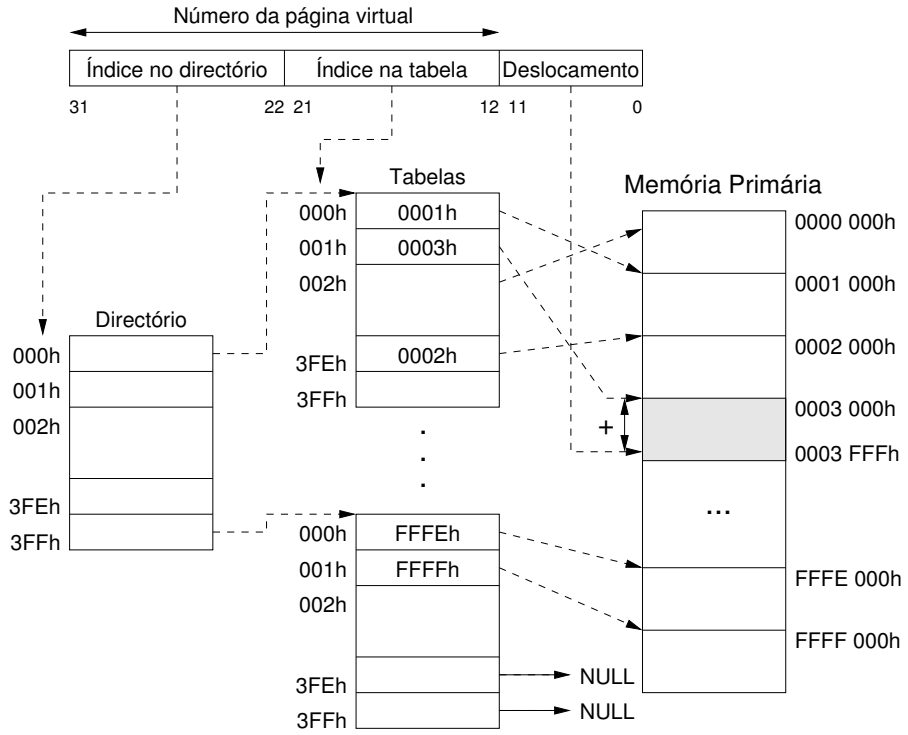


Figura 13.21: Tabela hierárquica de dois níveis para tradução de endereços virtuais.

bela de páginas para qualquer programa ocuparia, como se viu, 2M octetos. No caso de uma tabela hierárquica, os bits de endereço virtual da página seriam divididos em dois grupos de 10 bits cada. Os primeiros 10 bits, mais significativos, endereçariam a tabela do primeiro nível, onde estariam guardados os endereços (físicos) das tabelas do segundo nível. Estas seriam endereçadas com os 10 bits menos significativos do número de página virtual. Cada uma destas tabelas teria 2^{10} entradas, cada uma das quais com o descritor de página que, como se observou atrás, ocuparia pelo menos dois octetos para o número de página física.

Para ilustrar a vantagem das tabelas hierárquicas, considere um hipotético programa que use apenas 6M octetos, contíguos em memória virtual, a partir do endereço `00000000h`. Este programa iria utilizar apenas $6 \times 2^{20}/2^{12} = 1,5 \times 2^{10}$ páginas de memória. Os endereços de todas estas páginas encontram-se nas duas primeiras tabela do segundo nível. Isto significa que é apenas necessário guardar a tabela de nível 1 (o directório) e duas tabelas de nível 2, ocupando um total de $3 \times 2^{10} \times 2$ octetos, ou seja, 6k octetos, em vez dos 2M octetos necessários quando se usa uma tabela de páginas plana.

O conceito de tabelas de páginas hierárquicas pode ser generalizado para tabelas hierárquicas com mais níveis, o que permite espaços de endereçamento virtual de grande dimensão. Neste caso, o directório funciona como anteriormente, enquanto que o último nível continua a manter os endereços físicos das páginas. Os níveis intermédios permitem aceder às tabelas do nível seguinte.

Em geral, três níveis de tabelas são suficientes para todos os sistemas, independentemente da dimensão da memória virtual e física do sistema, embora existam arquitecturas que disponibilizam já, para futuras realizações, a utilização de quatro níveis de tabelas.

Regra geral, com a excepção do directório, as tabelas de páginas têm a mesma dimensão de uma página. Esta situação permite que tabelas de tradução que não estejam a ser utilizadas possam ser guardadas em disco, tal como as páginas de dados e código, reduzindo a fracção de memória ocupada pelas tabelas de tradução.

Existem também sistemas que utilizam outras formas de mapeamento de tabelas de páginas, denominadas de *tabelas invertidas*. No entanto, esta técnica, pouco utilizada em sistemas recentes, não será estudada neste texto.

13.4.2 Política de Substituição

Tal como no projecto de caches, existem diversas opções a tomar quando se projecta um sistema de memória virtual. As opções a tomar estão relacionadas com as questões que já foram estudadas no projecto de sistemas de caches: políticas de substituição, mecanismos de protecção, e manutenção da coerência entre memória e disco em operações de escrita.

A gestão destes mecanismos é, em geral, da responsabilidade do sistema operativo. Sempre que há um acesso que não encontra a página desejada em memória primária, ou seja, sempre que ocorre uma falta de página, é chamada uma rotina do sistema operativo que copia do disco essa página para uma dada zona da memória primária. Esta operação implica muitas vezes copiar para disco a página que lá se encontra.

Uma vez que copiar uma página entre disco e memória é uma operação demorada, justifica-se utilizar uma política de substituição de páginas muito mais sofisticada do que em caches. O facto desta decisão ser realizada pelo sistema operativo, portanto, em software, permite manter para cada página de memória informação detalhada sobre os acessos mais recentes, e decidir qual página deve ser substituída usando um algoritmo mais complexo. Em particular, é possível utilizar o algoritmo LRU, que envia para disco a página menos recentemente acedida, um método que tipicamente não é utilizável para gerir a substituição de blocos de cache.

O uso continuado e intenso de uma quantidade de memória muito superior ao disponível em memória primária leva a uma situação em que é necessário enviar continuamente páginas de memória para disco e vice-versa, conhecida como *thrashing*. Nesta situação, o computador continua a executar o programa pretendido, mas passa a maior parte do tempo (por vezes, mais de 99% do tempo) a gerir o sistema de memória virtual, levando a um aparente bloqueio do sistema. Esta situação é, em geral, de evitar.

13.4.3 Política de Escrita

Devido também ao elevado tempo de acesso ao disco, um sistema de memória virtual com uma política de escrita *write-through* não é tipicamente realizável. Assim, todas as escritas são feitas na memória primária, numa abordagem semelhante à política *write-back write-allocate* das caches. Esta política implica a

existência de um bit nos descritores de página das tabelas de tradução que indica se essa página foi alterada em memória ou não, chamado aqui também de *dirty* bit.

Na substituição de uma dada página, este bit é testado. Se estiver a 0, isso indica que a página não foi alterada desde que foi trazida para memória primária. Neste caso, a cópia em disco continua válida, significando que se pode simplesmente escrever por cima da cópia da página em memória primária. Se, pelo contrário, esse bit estiver a 1, então antes de escrever por cima da página é necessário escrevê-la para disco.

13.4.4 Bits de Controlo

Além do endereço físico da página, os descritores de página incluem um conjunto de bits que indicam diversas propriedades da página. Embora a informação extra que é guardada varie de sistema para sistema, os seguintes bits estão, tipicamente, sempre presentes:

- Um bit que indica se o endereço físico é válido. Caso não seja, isso significa que a página está em disco e que terá de ser carregada para memória principal antes de o seu conteúdo ser utilizado pelo processador.
- Um bit que indica se a página foi alterada enquanto em memória, o que implica que deverá ser modificada em disco, quando for substituída.
- Bits de protecção, que indicam se a página é acessível para escrita, leitura e execução, em modo utilizador.
- Bits de protecção, que indicam se a página é acessível para escrita, leitura e execução, em modo sistema.
- Um bit que indica que a página foi acedida recentemente.

Outros campos associados a cada página de memória existem e são utilizados pelo sistema operativo para operações específicas.

Note-se o paralelismo que existe entre a informação que é guardada para páginas de memória virtual e para blocos de cache. Em ambos os casos, uma memória mais rápida funciona como armazenamento temporário de uma memória mais vasta mas mais lenta, e, em ambos os casos, é necessário guardar informação que permita decidir quais são os dados que devem ser substituídos ou copiados para a memória mais lenta.

13.4.5 *Translation Lookaside Buffers*

Se cada acesso a memória passasse por um processo de tradução que envolvesse acessos aos diversos níveis da tabela de páginas, o impacto no desempenho do sistema seria demasiado grande, fazendo com que um acesso a memória num sistema de memória virtual fosse três ou quatro vezes mais lento que num sistema sem memória virtual.

Para contornar esta dificuldade, usa-se uma pequena memória cache, chamada *Translation Lookaside Buffer*, que devolve ao processador o endereço físico que corresponde a uma dada página virtual. Dado que cada página corresponde a um número elevado de posições de memória, mesmo uma pequena

cache irá ter uma taxa de sucesso muito elevada, pois o número de páginas diferentes que são usadas num dado troço de programa é tipicamente pequeno, dada a propriedade da localidade dos acessos.

A existência desta cache permite evitar, na maior parte das traduções, os acessos à tabela de páginas que são necessariamente mais lentos.

13.4.6 Interligação da Memória Virtual com as Caches

Em geral, o sistema de memória virtual co-existe com as caches, pelo que o carregamento dos dados a partir de um dado endereço desencadeia, efectivamente dois processos: a procura desses dados numa das caches, e a tradução dos endereços virtuais para endereços físicos.

Existem fundamentalmente duas possibilidades de interligar o sistema de cache com o sistema de memória virtual. A primeira alternativa endereça as caches com endereços virtuais enquanto que a segunda utiliza endereços físicos.

No primeiro caso, os endereços que são enviados para as caches são os endereços emitidos pelo processador, antes de qualquer processo de tradução. Esta alternativa tem a vantagem de tornar mais rápido o acesso aos dados, uma vez que não é necessário traduzir os endereços antes de endereçar as caches. Uma vez que as caches contêm os dados que correspondem a um dado endereço virtual, o seu conteúdo tem de ser invalidado sempre que exista uma alteração do mapeamento de memória virtual, por exemplo, quando muda o processo que está a ser executado.

No segundo caso, o endereço virtual é, em primeiro lugar, traduzido para um endereço físico, que é usado para endereçar as caches. O acesso às caches é mais lento, mas, em compensação, existe um mapeamento mais directo entre os dados que se encontram em cache e os dados que se encontram em memória. Se as páginas de memória se encontrarem alinhadas em endereços múltiplos do tamanho da página, os bits que correspondem ao deslocamento dentro da página não são alterados no processo de tradução. As somas que se encontram indicadas nos esquemas de tradução são, assim, realizadas como concatenações dos bits devolvidos pela TLB e dos bits de deslocamento. A Figura 13.22 ilustra o esquema de interligação entre a TLB e uma cache, endereçada fisicamente, numa situação em que as páginas se encontram alinhadas em memória.

Uma vez que a primeira fase de acesso a uma cache que não seja totalmente associativa utiliza apenas os bits de índice para escolher o bloco onde estão os dados, poderá ser possível iniciar o acesso à cache antes de o índice da tabela ter sido traduzido pela TLB. Para tal, basta que o número de bits de índice e de deslocamento usados pela cache não seja superior ao número de bits necessários para especificar o deslocamento dentro de cada página de memória virtual. A Figura 13.23 representa, de forma muito simplificada, o sistema de memória de um processador de 32 bits, com memória virtual, TLB e um nível de cache endereçada fisicamente.

Neste exemplo, o processador tem um espaço de endereçamento virtual de 4G octetos (2^{32} octetos) com páginas de 4k octetos. O endereço de página é enviado para uma TLB de mapeamento directo, com 16 entradas, que envia o endereço físico da página para uma cache de 4k octetos. Este endereço é separado em 4 componentes: 20 bits ($A_{31}-A_{12}$) para a etiqueta, 8 bits ($A_{11}-A_4$) para

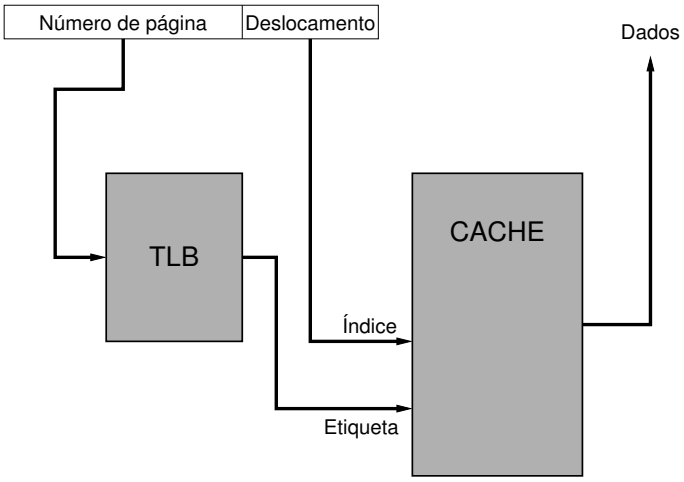


Figura 13.22: Interligação entre a TLB e a cache.

o índice e 2 bits ($A_3 - A_2$) para o deslocamento dentro do bloco. Os dois bits menos significativos (A_1 e A_0) não são utilizados, sendo apenas usados para seleccionar o octeto dentro da palavra, o que não foi considerado neste esquema. Como o número de bits necessários para especificar o deslocamento dentro de cada página é de 12, e as páginas se encontram alinhadas em memória, a selecção do bloco de cache pode ser iniciada (usando as linhas de endereço A_{11} a A_4) ainda antes da TLB traduzir o endereço da página de virtual para físico. Se os dados estiverem presentes em cache, estes são enviados directamente para o processador. Caso contrário, o controlador da memória primária verifica que a linha de HIT não foi activada, e desencadeia um acesso à memória primária de 64M octetos. Este acesso é feito activando as linhas de controlo das memórias dinâmicas e os respectivos endereços de linhas e colunas.

Neste esquema muito simplificado, não estão ilustrados os circuitos utilizados para as operações de escrita na cache de dados e na TLB, nem os diversos sinais de controlo necessários para controlo das memórias dinâmicas (DIMM). O leitor poderá no entanto observar que a complexidade de um sistema de memória rivaliza com a do próprio processador, especialmente se se considerar que, em sistemas reais, existem diversos níveis de caches, caches separadas para endereços e dados, arquitecturas complexas de memória primária e interligação dos barramentos de dados com dispositivos de entrada/saída.

Sumário

Neste capítulo, estudaram-se os diversos componentes do sistema de memória de um processador típico.

Foi analisada a forma como a memória primária é constituída a partir de módulos simples, interligados entre si por forma a permitir a construção da estrutura de memória desejada. Cada um destes módulos corresponde a uma dada parte da memória visível pelo processador, podendo a organização lógica dos módulos ser descrita pelo mapa de memória do processador.

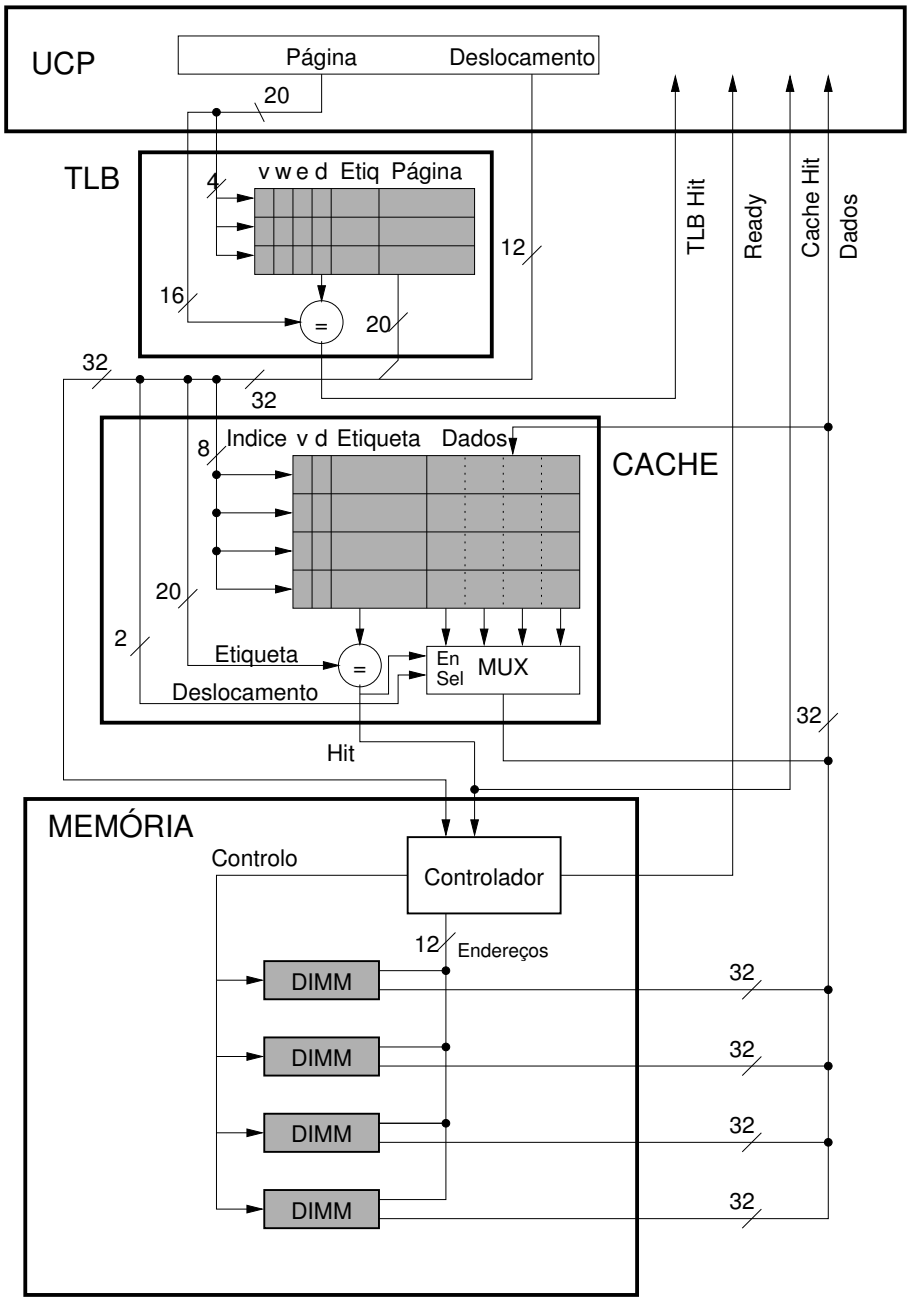


Figura 13.23: Sistema de memória de um processador de 32 bits.

Foi também estudada a forma como memórias de diversas velocidades e capacidades são interligadas entre si numa hierarquia de memória, sendo as memórias mais rápidas e de menor capacidade acedidas em primeiro lugar, por forma a diminuir o tempo médio de acesso a memória.

Finalmente, foram estudados os sistemas de memória virtual, que permi-

tem usar a memória secundária para alargar o espaço de endereçamento utilizável para além do que seria possível se apenas se usasse a memória primária.

Capítulo 14

Entradas, Saídas e Comunicações

Nos capítulos anteriores definiram-se dois dos componentes principais de um computador, a unidade central de processamento e o sistema de memória. Embora todo o processamento se efectue entre estas duas unidades, tal seria de utilidade reduzida caso não existisse alguma forma de interacção com o mundo exterior, que por um lado permita introduzir os dados a processar e por outro lado torne acessível os resultados.

Nos primeiros computadores, esta interface era rudimentar. As entradas eram especificadas por interruptores que definiam o seu valor bit a bit e o resultado era visualizado em lâmpadas que indicavam o seu valor também ao nível do bit. Desde então houve grande evolução não só nas interfaces pessoa-máquina como nas interfaces entre diferentes máquinas. O desenvolvimento de novas formas de interacção com os computadores é cada vez mais um assunto de intensa investigação.

O termo *periféricos* é normalmente utilizado para designar de forma genérica os dispositivos de entrada e saída de um computador. Hoje em dia existe uma grande diversidade de periféricos, com características muito diferentes a variados níveis. Para simplificar a interface do lado da UCP o acesso aos diferentes periféricos é definido como tendo o mesmo protocolo de um acesso a uma posição de memória (Secção 13.1.3). Devido ao protocolo de acesso ser o mesmo, é indiferente para o processador aceder a uma posição de memória ou a um periférico, possibilitando, portanto, o mapear no espaço de memória dos diferentes periféricos (Secção 14.1). Esta simplificação do lado do processador implica a existência do lado de cada periférico de um circuito de interface que converta a informação proveniente do periférico para o protocolo utilizado no acesso à memória.

Neste capítulo começa-se por descrever o funcionamento de alguns dos periféricos mais conhecidos. Em seguida, apresentam-se diferentes modos de comunicação entre o computador e os periféricos. Finalmente, discutem-se as formas como se transfere a informação para periféricos com diferentes ritmos de transmissão de informação.

14.1 Arquitectura de Entradas/Saídas

A organização do sistema de entradas e saídas num computador é um aspecto de central importância na definição da sua arquitectura. Como já foi anteriormente referido, um computador está organizado em torno de um sistema de barramentos que permite ao processador trocar informação com a memória e com todos os periféricos do sistema. Globalmente essa arquitectura está representada na Figura 14.1.

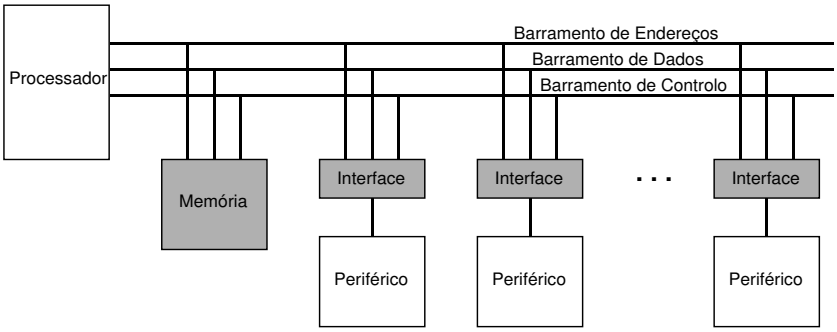


Figura 14.1: Representação geral da arquitectura de um computador.

O *barramento de endereços* permite ao processador indicar a posição de memória ou o periférico com que pretende interagir. Trata-se, numa abordagem inicial, de um barramento unidireccional em que a origem é sempre o processador.

O *barramento de dados* permite a circulação dos dados a transferir entre o processador e a memória ou os periféricos. Trata-se de um barramento bidireccional em que o processador pode ser a origem dos dados, quando se pretende escrever dados na memória ou transferi-los para periféricos de saída, ou o destino dos dados com origem na memória ou nos periféricos de entrada.

O *barramento de controlo* inclui todas as linhas que permitem gerir e coordenar a transferência. Nesse barramento estão presentes linhas que permitem indicar a direcção da transferência, sincronizar a transferência, definir a dimensão dos dados a transferir, determinar se se trata de transferências de dados envolvendo a memória ou os periféricos, sinalizar pedidos de atenção dos periféricos e outras funções de controlo. As linhas efectivamente presentes são, contudo, muito dependentes das soluções arquitecturais utilizadas.

14.1.1 Interfaces

A ligação dos periféricos é, na maior parte dos casos, assegurada por um módulo que pode assumir diversos graus de complexidade, denominado *interface*. A interface existe para isolar o processador das particularidades dos periféricos. De entre as funcionalidades pedidas à interface, podem ser referidas, nesta fase, as seguintes:

- Promover as adaptações necessárias do ponto de vista físico, nomeadamente ao nível das tensões e correntes envolvidas na comunicação entre o processador e os periféricos.

- Permitir esconder ao processador a velocidade dos periféricos, tipicamente mais baixa que a do processador.
- Descodificar os endereços presentes no respectivo barramento por forma a identificar uma transferência de dados que envolve o periférico a que está ligada.
- Controlar o periférico nos casos em que podem existir vários modos de funcionamento.
- Controlar a comunicação física com o periférico através de lógica adequada, diminuindo, assim as tarefas a desempenhar pelo processador.
- Sinalizar ao processador, através do barramento de controlo, a presença de dados provenientes do periférico ou a disponibilidade deste para receber dados através do *sistema de interrupções* do computador. Este tema será tratado adiante na Secção 14.5.2
- Permitir ao periférico participar em modos de transferência mais avançados com envolvimento reduzido do processador como é o caso do *acesso directo à memória* que será analisado na Secção 14.5.3.

A estrutura de uma interface está ilustrada na Figura 14.2.

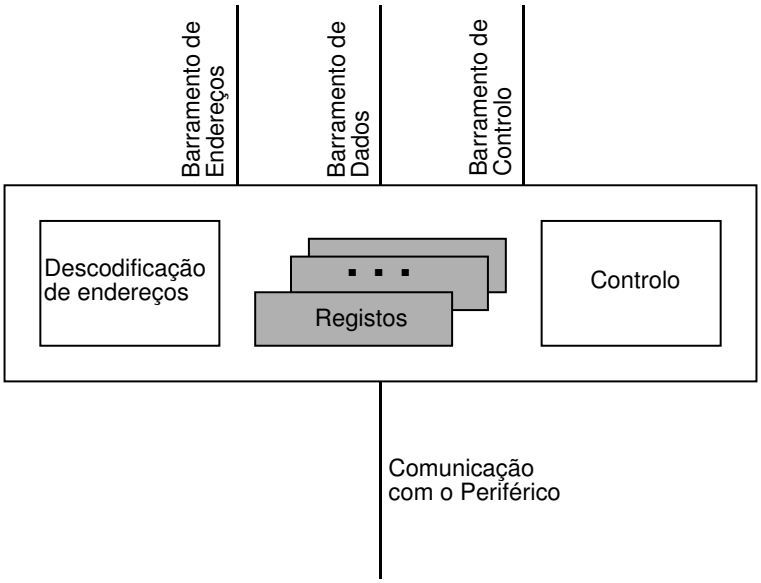


Figura 14.2: Representação geral da arquitectura de uma interface.

No caso de periféricos uni-direccionais, isto é, só de saída ou entrada, como uma impressora, ou um rato, a interface possui um registo onde o processador escreve os dados a enviar para o periférico ou onde lê os dados provenientes do periférico. Como é óbvio em periféricos bi-direccionais, como por exemplo, um modem, é de esperar a existência de dois registos, um em que o processador escreve os dados a enviar e outro em que o processador lê os dados recebidos. Estes registos, a unidade mais simples de entrada/saída endereçável pelo processador, têm o nome de *portos*.

O facto dos dados serem escritos nestes portos e deles lidos, em vez de tal ser feito no periférico propriamente dito permite que a operação seja feita à velocidade possível por parte do processador. Se, de facto, a escrita ou a leitura fossem feitas envolvendo directamente o periférico, o processador teria que realizar o seu ciclo de leitura ou escrita à velocidade permitida pelo periférico, tipicamente algumas ordens de grandeza abaixo da velocidade possível nos barramentos internos.

Muitas interfaces, possuem, para além dos *portos de dados*, isto é, os portos destinados a transferir dados, alguns outros portos onde o processador pode escrever informações de controlo que permitem configurar determinados aspectos do funcionamento do periférico ou da comunicação com ele, bem como ler informação sobre o estado do periférico, da ligação ou da própria interface. Trata-se, respectivamente, de *portos de controlo* e *portos de estado*.

Cada um destes portos é naturalmente acedido através de endereços diferentes. Conceptualmente o módulo de descodificação de endereços é responsável por identificar a presença no barramento de dados do endereço correspondente a cada um dos registos presentes na interface por forma a que o porto seleccionado seja escrito ou lido conforme a acção especificada no barramento de controlo. Na prática, as interfaces só descodificam os bits menos significativos do barramento de endereços em número necessário para endereçar os diversos portos internos da interface, sejam portos de dados, controlo ou estado. Estes portos têm, em geral, endereços seguidos. Tal como no caso da descodificação de endereços de memória parte da descodificação é realizada exteriormente às interfaces e em comum para todas elas.

O módulo de controlo da interface coordena as diversas acções desta com relevo para a gestão da comunicação com o periférico e, nos casos em que isso é relevante, nas transferências implicando o sistema de interrupções (Secção 14.5.2) ou o acesso directo à memória (Secção 14.5.3).

14.1.2 Tipos de Endereçamento dos Portos

Como se referiu, o endereçamento é, de forma geral, uma função dividida entre um bloco de descodificação de endereços externo às interfaces, muitas vezes partilhado com a descodificação de memória, e alguma descodificação interna à interface. Na Figura 14.3 exemplifica-se este tipo de estrutura.

Repare-se que a descodificação é feita primariamente por um circuito de descodificação externo à interface que actuará uma linha de *enable* do descodificador dentro da interface. Desse modo é possível conseguir que a interface só possa estar activa num determinado leque de endereços. Internamente, para endereçar cada um dos n portos, existe um segundo circuito de descodificação que descodifica i bits de endereço, sendo $n = 2^i$. Em geral, esses bits correspondem aos bits menos significativos do barramento de endereços. Na Figura 14.3 representa-se uma interface com n portos dos quais são representados o porto 0 e o porto $n-1$, sendo o primeiro um porto de saída e o segundo um porto de entrada.

Há três formas fundamentais de organizar o endereçamento dos periféricos. Nas considerações anteriores tem sido assumido que as interfaces dos periféricos partilham com a memória os barramentos de endereços, dados e controlo. Tal não é estritamente necessário. De facto, seria até conceptualmente interessante considerar a existência de um segundo conjunto de barramentos

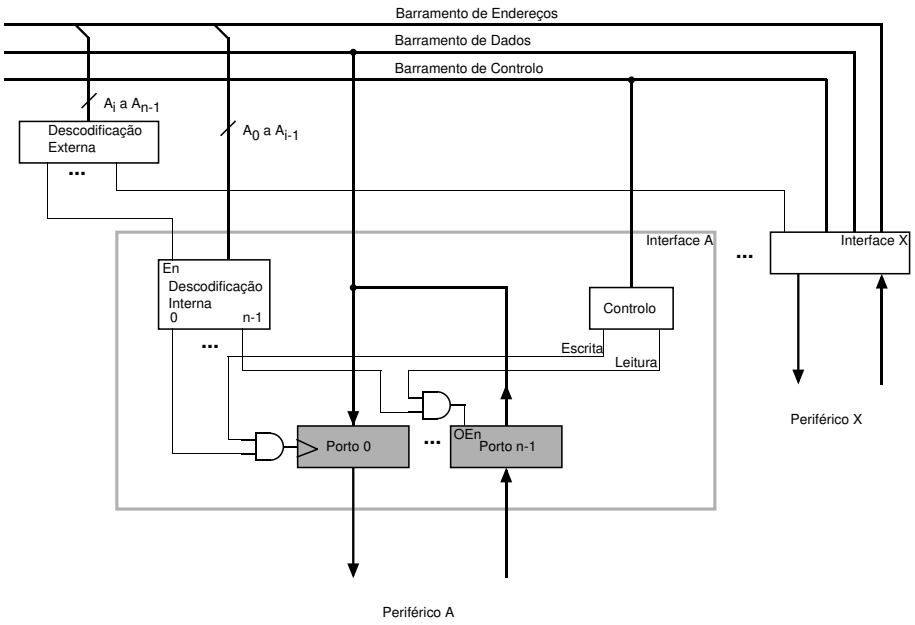


Figura 14.3: Descodificação de endereços para acesso aos portos de uma interface.

para as interfaces de entrada/saída fisicamente separado dos barramentos de memória. Essa opção permitiria um aumento do desempenho do computador, uma vez que, simultaneamente com uma comunicação com um periférico é possível manter os acessos à memória, quer para transferir dados, quer para ler instruções.

Esta opção, obriga, porém, para ser útil, a uma nova concepção do processador que passaria a ter uma capacidade de processamento paralelo que muito complicaria a sua arquitectura. Essa não é uma solução frequente. No entanto, o uso de um processador dedicado apenas às entradas/saídas, partilhando a memória com o processador principal, permite levar à prática esta solução com menor complicação e de uma forma perfeitamente satisfatória do ponto de vista do desempenho global. Este tipo de arquitectura será referida com um pouco mais de detalhe no Secção 14.5.4.

Assumindo, portanto, uma arquitectura com um único conjunto de barramentos partilhados entre a memória e as interfaces de entrada/saída, podem ainda ser conceptualizadas duas formas de organização.

Em muitas arquitecturas, como é também o caso do P3, assume-se a existência de um único espaço de endereçamento que, tal como os barramentos, é partilhado por memória e periféricos. Nesse tipo de arquitectura haverá endereços atribuídos a posições de memória e outros atribuídos a portos. Trata-se, como já se referiu no Secção 11.3.3 de uma arquitectura com os portos mapeados em memória (em inglês, *memory mapped*). Em arquitecturas deste tipo o endereçamento de um porto faz-se colocando no barramento de endereços o respectivo endereço e actuando no barramento de controlo uma linha que indique a direcção da transferência pretendida. Isso é conseguido habitualmente pela utilização de uma linha de escrita/leitura $RD/\overline{WR}$ que estará a 1, ou me-

lhor, em H, quando se pretende uma leitura de um porto e em 0, ou melhor, em L, quando se pretende uma escrita.

Do ponto de vista do conjunto de instruções de um processador deste tipo, não há, como é óbvio, necessidade de instruções de entrada/saída específicas, uma vez que todas as transferências se fazem como se de posições de memória se tratasse. É, portanto, utilizado para entradas/saídas todo o conjunto de instruções que permitem o acesso à memória.

Esta arquitectura tem a desvantagem de ter de ser necessário reservar, no espaço de endereçamento de memória, um conjunto de endereços para portos de entrada ou saída. Para além da diminuição consequente da dimensão da memória utilizável, isso obriga, por vezes, a soluções menos elegantes para obstar a que certos endereços atribuídos a portos provoquem simultaneamente o acesso a posições de memória.

A solução está em utilizar espaços de endereçamento separados para a memória e para os portos de entrada/saída, embora partilhando os mesmos barramentos. Para conseguir isso, terá o barramento de controlo de fornecer informação indicando se um determinado endereço é um endereço de memória ou de um porto. Isso consegue-se, por exemplo, através de uma linha que distinga os dois tipos de acesso. Uma linha desse tipo pode assumir o valor 1 quando se trata de um acesso à memória e um valor 0 quando se trata de um acesso a um porto. Uma designação adequada seria, por exemplo, $MEM/\overline{IO}$. Uma solução alternativa é a utilização de linhas separadas de escrita e leitura para memória e portos de entrada/saída. Ter-se-ia, então, linhas com designações do tipo $MEMREAD$, $IOREAD$, $MEMWRITE$ e $IOWRITE$, por exemplo. Este tipo de arquitectura é referido como de *entradas/saídas isoladas* ou *entradas/saídas separadas* (em inglês, *independent IO*). Neste tipo de arquitectura o conjunto de instruções tem de incluir instruções explícitas de entrada/saída. Essas instruções, quando executadas, provocam a activação das linhas adequadas ou do nível adequado da linha $MEM/\overline{IO}$ para que a transferência se dê envolvendo os portos e não a memória. Tipicamente essas instruções são mais limitadas na sua operatividade que as instruções de acesso à memória, limitando-se, por vezes, a uma instrução de entrada de dados e outra de saída.

14.2 Periféricos

Nas secções seguintes descreve-se o funcionamento básico de alguns dos periféricos mais importantes nos computadores de uso genérico. De sublinhar que, apesar da importância que estes têm nos computadores com que a maioria das pessoas trabalha diariamente, existem muitos outros sistemas computacionais em que estes periféricos nem sequer existem. É o caso dos sistemas computacionais existentes em muitos dos electrodomésticos. Para estes, a entrada e saída de dados processa-se através de periféricos específicos ao sistema em que estão integrados.

14.2.1 Teclados

O *teclado* é o periférico de entrada de dados mais importante nos computadores de uso genérico, ao ponto de ser normal os computadores não arrancarem caso

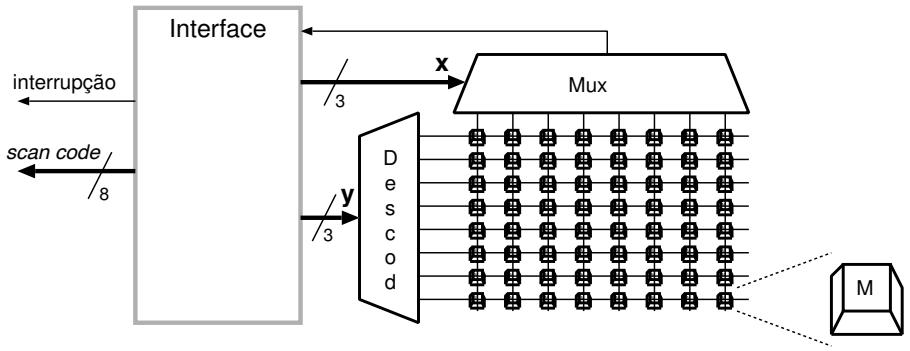


Figura 14.4: Esquema interno de um teclado.

não detectem um teclado ligado. A Figura 14.4 apresenta o esquema interno de um teclado.

Uma matriz de linhas e colunas de interligações eléctricas constitui o elemento básico de um teclado. Em cada cruzamento linha/coluna está colocado um interruptor correspondente a uma tecla. Ao premir-se a tecla, fecha-se um circuito entre uma linha e uma coluna.

Para fazer a leitura desta matriz, o circuito de interface do teclado inclui um controlador que contém dois contadores. Como mostra a figura, um dos contadores, y, faz o varrimento das linhas, colocando de cada vez uma, e só uma, ao valor lógico 1. Por cada contagem do contador y, o segundo contador, x, faz o varrimento das colunas, lendo o valor lógico destas. A detecção de uma coluna a 1 indica que a tecla na posição (x, y) da matriz está premida. A partir desta coordenada, o controlador identifica univocamente a tecla.

O controlador no teclado tem um conjunto de outras funções. Uma delas é resolver um problema associado aos interruptores mecânicos. Ao fechar-se um interruptor, antes da linha estabilizar no novo nível eléctrico, esta apresenta uma oscilação de sinal, o que pode erroneamente indicar uma sequência repetida de premir e libertar da tecla. O controlador filtra estes picos, esperando um tempo razoável entre uma alteração do estado de uma tecla e a leitura do seu estado definitivo. Esta operação tem o nome de *debounce*.

Para reduzir a quantidade de informação a transmitir à UCP, em vez deste controlador enviar constantemente o estado das diferentes teclas, são apenas enviadas alterações do estado das teclas. Por exemplo, se se premir a tecla 'M', é enviado ao processador o código `M_premida`. Ao se libertar a tecla, é enviado o código `M_liberta`. A estes códigos dá-se o nome de *make code* e *break code*, respectivamente. Este funcionamento aplica-se também às *silent keys*, como as teclas *shift* e *control*. Para poder detectar a mudança de estado de uma tecla, o controlador terá que manter numa memória local o estado actual de todas as teclas.

Estes códigos são chamados de *scan codes*. Associado ao teclado (como a qualquer outro periférico) existe um programa chamado *device driver*, responsável por fornecer a nível de software uma interface para o periférico. No caso do teclado, será o *device driver* que converterá as coordenadas (x, y) para um

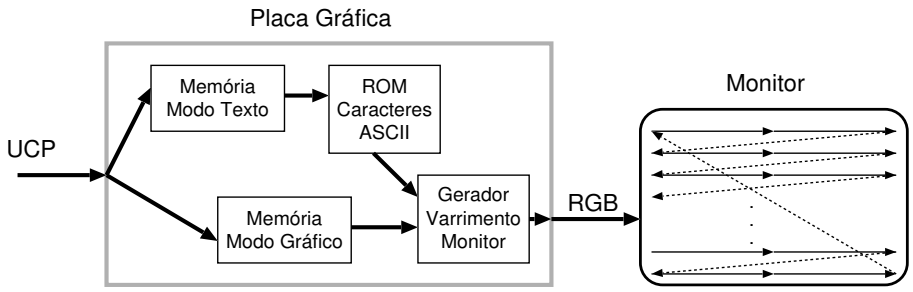


Figura 14.5: Esquema interno de uma placa gráfica e ligação a monitor.

dado *scan code*. É esta operação que permite que um teclado com a mesma disposição de teclas possa ter disposições diferentes de símbolos e assim adaptar-se facilmente a diferentes línguas. Por exemplo, o símbolo '-' no teclado português está na mesma tecla do símbolo '/'no teclado inglês. A configuração do *device driver* permitirá associar essa tecla ao *scan code* correspondente para a língua para a qual o sistema está configurado.

Outro aspecto dos *scan codes* é que não indicam se a letra pretendida pelo utilizador é maiúscula ou minúscula. Será o *device driver* que terá que verificar se o *make code* de uma letra ocorre entre um `shift_premida` e um `shift_liberta`. Se sim, então a letra será maiúscula, caso contrário será minúscula.

A conversão de *scan codes* para código ASCII é normalmente da responsabilidade da aplicação.

Sempre que existe uma alteração do estado de uma tecla, o controlador do teclado envia uma interrupção para a UCP indicando que tem dados para enviar. Por vezes, a UCP está ocupada com outras actividades e não consegue dar atenção imediata. Para evitar a perda de dados, o controlador do teclado possui um *buffer* capaz de armazenar uma sequência de eventos do teclado. Esta solução não resolve completamente o problema, pois este *buffer* tem tipicamente uma capacidade reduzida. Nos casos em que a UCP demora um pouco mais a ler os dados do teclado, pode acontecer que o *buffer* encha, fazendo com que todos os eventos posteriores sejam descartados. É comum o controlador do teclado fazer soar um aviso sonoro para indicar que essas teclas estão a ser ignoradas.

14.2.2 Monitores

À semelhança do teclado, o *monitor* ocupa o lugar principal na classe dos periféricos de saída de dados. A interface entre o monitor e a UCP é realizada pela *placa gráfica*. Um diagrama da organização interna de uma placa gráfica e sua ligação a um monitor está apresentada na Figura 14.5.

As placas gráficas têm, em geral, dois modos de funcionamento, programáveis pela UCP: *modo texto* e *modo gráfico*. Em qualquer dos modos de funcionamento, do ponto de vista da UCP o monitor é uma matriz (linha,coluna). No caso do modo texto, os elementos desta matriz são caracteres ASCII. No modo

gráfico, os elementos da matriz são pontos no monitor, o elemento de menor tamanho manipulável pela placa gráfica e a que se dá o nome de *pixel*.

Os primeiros monitores só funcionavam em modo texto. Chama-se a este tipo de monitores de *monitores alfanuméricos*. Hoje em dia o modo texto existe não só para manter compatibilidade com monitores antigos, como também para servir como o denominador comum mais simples que qualquer sistema consegue reconhecer, evitando-se assim potenciais problemas de incompatibilidade entre diferentes interfaces gráficas. Neste modo, para se ecoar um caracter no monitor basta enviar para a placa gráfica o seu código ASCII. O local onde esse caracter irá aparecer no monitor é determinada pela posição do *cursor*, uma referência (linha,coluna) mantida pela placa gráfica. O funcionamento típico é o valor da coluna do cursor ser incrementado por cada caracter escrito, sendo portanto os caracteres escritos em sequência. Se se exceder o número máximo de colunas, então o valor da linha do cursor é incrementado e o valor da coluna colocado a zero, fazendo com que a próxima escrita seja no início da linha seguinte. Se se exceder o número máximo de linhas, o texto no monitor é todo movido uma linha para cima (em inglês, *scroll up*), perdendo-se a linha de cima do monitor e ganhando-se uma linha em branco em baixo. O próprio código ASCII inclui alguns comandos para controlar o cursor dos monitores alfanuméricos, pois eram estes os monitores disponíveis na altura da definição deste código. Por exemplo, o código:

- BS (*backspace*) permite recuar o cursor uma posição, ou seja, decrementa o valor da coluna.
- LF (*linefeed*) passa o cursor para a linha seguinte, ou seja, incrementa o valor da linha.
- CR (*carridge return*) coloca o cursor no início da linha, ou seja, coloca o valor da coluna a zero.

Existem também códigos para colocar o cursor numa dada posição do monitor. No entanto, estes são específicos para cada sistema.

Para que seja possível a utilização dos monitores gráficos actuais, as placas gráficas, quando em modo texto, utilizam uma ROM para obter a descrição dos caracteres em termos de pixels.

Em modo gráfico, a UCP define ponto a ponto, isto é, pixel a pixel, o que deve aparecer no monitor. A *definição gráfica* é o número total de pixels no monitor, distribuídos por um número total de linhas e um número total de colunas. Em monitores a preto e branco (ou monocolores), basta um bit por posição (linha,coluna) para indicar se o pixel nessa posição está aceso (branco) ou apagado (preto). Para monitores a cores, é necessário definir para cada pixel qual é a cor que este deve assumir. Assim, para cada posição (linha,coluna) define-se um valor que indica uma entrada dentro de uma *palete de cores* (em inglês, *colormap*). O número de bits necessários por pixel depende portanto do tamanho desta palete de cores. Por exemplo, se a palete de cores tiver 256 entradas cada pixel fica definido por 8 bits. O tamanho da palete de cores determina o número máximo de cores diferentes que se podem utilizar em simultâneo. No entanto, cada aplicação pode definir as entradas na palete de cores de forma a que diferentes aplicações possam utilizar um conjunto de cores diferente.

A placa gráfica mantém o estado de cada pixel numa memória interna. A capacidade desta memória determina a definição máxima da placa gráfica. Por

exemplo, para uma definição de 1280×1024 (1280 colunas por 1024 linhas) com uma palete de 256 cores (8 bits) é necessário que a placa gráfica tenha uma memória de pelo menos $1280 \times 1024 \times 8$ bits, ou seja, 1,25M octetos. Se com esta memória se pretender uma maior quantidade de cores, será necessário reduzir a definição. Supondo que se pretende aumentar o tamanho da palete para 64k cores (16 bits), não é possível usar a definição padrão abaixo da anterior, 1024×768, pois exigiria $1024 \times 768 \times 16$ bits, ou seja, 1,5M octetos. Assim, ter-se-ia que optar pela definição padrão seguinte, 800×600, a que corresponde a memória $800 \times 600 \times 16$ bits, portanto, 0,9M octetos.

A UCP define a cor de cada pixel escrevendo na correspondente posição de memória gráfica. Para facilitar a tarefa aos programadores, existem normalmente bibliotecas gráficas que fornecem rotinas de alto nível para definir objectos complexos no monitor. Estas rotinas são específicas para cada sistema.

A ligação entre a placa gráfica e o monitor depende do tipo de monitor. Ainda hoje, a maioria dos monitores são baseados num *tubo de raios catódicos* (à semelhança das televisões). Nestes monitores, um feixe de electrões é emitido contra uma tela de fósforo, elemento que tem a propriedade de se tornar luminoso ao ser atingido pelo feixe. Por controlo do varrimento do feixe, e da intensidade deste durante o varrimento, é possível definir padrões na tela. Tipicamente, este feixe de electrões varre o monitor por linhas da esquerda para a direita, desligando-se quando faz o retorno da direita para a esquerda, avançando para a linha seguinte, como apresentado na Figura 14.5. Durante o varrimento de uma linha, o feixe é modulado de forma a acender ou apagar cada pixel dessa linha.

Esta descrição aplica-se a monitores monocolores. Nos monitores a cores, para cada pixel existem três telas de fósforo, correspondentes às três cores elementares: vermelho, verde e azul. Três feixes de electrões permitem controlar independentemente cada uma destas telas, através da intensidade do feixe correspondente. Assim, é possível definir para cada pixel um valor diferente de luminosidade para cada uma das três cores elementares, permitindo assim definir, em princípio, qualquer cor.

Para controlar o monitor, a placa gráfica lê em sequência as posições de memória interna e gera um sinal analógico para o monitor com os valores para as componentes vermelho, verde e azul de cada pixel. Este sinal tem o nome *RGB*, pois estas três cores elementares em inglês têm os nomes *Red*, *Green* e *Blue*, respectivamente. Em geral, a intensidade de cada uma destas cores elementares é definida com um octeto. Por esta razão, a utilização de uma palete de cores com 2^{24} entradas é chamada de *cor verdadeira* (em inglês, *true color*), pois define univocamente cada cor possível no monitor.

De forma a que uma pessoa não se aperceba do varrimento, o monitor deve ser completamente escrito pelo menos 24 vezes por segundo. Caso contrário o utilizador terá uma sensação de piscar do monitor. Esta exigência implica uma grande largura de banda entre a placa gráfica e o monitor, pois é necessário enviar 24 vezes por segundo três octetos para todos os pixels. Por exemplo, para uma definição 1280×1024, transferem-se 90M octetos/s.

Hoje em dia, começam a ficar populares os monitores baseados em LCD (do inglês, *Light Coupled Device*). Nestes monitores, cada pixel é um elemento activo cuja cor pode ser controlada independentemente. A interface com estes monitores é diferente em dois aspectos. Por um lado, em princípio não é necessário a placa gráfica ter memória, pois os valores definidos para os pixels do

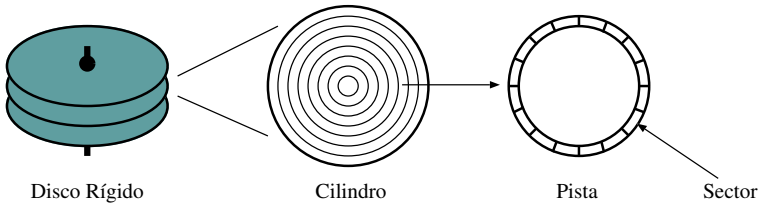


Figura 14.6: Diagrama da organização interna de um disco rígido.

monitor LCD são mantidos indefinidamente, ou seja, a memória gráfica poderá estar do lado do monitor. Por outro lado, cada pixel é endereçado individualmente, permitindo que a interface seja completamente digital. As tarefas da placa gráfica ficariam assim reduzidas a fazer a conversão para possíveis definições diferentes e conversão entre modo texto e modo gráfico. No entanto, para manter a compatibilidade com placas gráficas anteriores, os monitores LCD em geral aceitam como entrada o sinal RGB tradicional. Além disso, hoje em dia as próprias placas gráficas possuem algum processamento de imagens gráficas.

14.2.3 Discos Magnéticos

Por último, analisa-se nesta secção o funcionamento de um periférico de entrada e saída de dados, o *disco rígido*. Este periférico é também ubíquo nos computadores de uso geral. A sua utilização primária é no armazenamento de dados de forma permanente, pois os valores nos registos do processador, na cache ou na memória principal são perdidos quando a fonte de alimentação é desligada. Para além desta função, os discos foram já estudados no Capítulo 13 como o elemento de mais alto nível na hierarquia de memória. Qualquer destas duas funções é controlada pelo sistema operativo. A primeira é gerida pelo *sistema de ficheiros* e a segunda pelo sistema de memória. Assim, o espaço de armazenamento dos discos está normalmente dividido em pelo menos duas *partições*, uma para o sistema de ficheiros, e outra para o sistema de memória, ou espaço de *swap*. Poderá haver mais partições, pois os sistemas operativos em geral suportam mais do que uma partição para cada uma destas funções.

A Figura 14.6 apresenta um diagrama da organização interna de um disco rígido. Um disco rígido é de facto constituído por uma pilha concêntrica de discos magnéticos. Estes discos encontram-se permanentemente em rotação, a uma velocidade elevada e constante. Cada um destes discos está organizado em *pistas*, como mostra a figura. Às pistas dá-se também o nome de *cilindros*, significando de facto o conjunto de pistas com o mesmo raio de todos os discos. Cada um destes discos tem uma cabeça magnética que se pode deslocar radialmente e assim posicionar-se sobre qualquer das pistas desse disco. Cada pista, por seu lado, está dividida em *sectores*, que são o elemento mínimo de informação num disco.

Quando o sistema operativo acede ao disco, para escrita ou leitura, fá-lo sempre em termos de um, ou mais, sectores. Cada sector tem uma referência única que é utilizada pelo sistema operativo. Para fazer a interface entre a UCP

e a parte mecânica do disco, estes incluem internamente um controlador. A partir da referência indicada pela UCP, o controlador do disco determina qual dos cilindros contém o sector pretendido, e em qual das pistas deste cilindro esse sector se encontra. O controlador faz deslocar a cabeça desse cilindro de forma a colocá-la sobre essa pista e espera até que o sector em causa passe por baixo da cabeça magnética, altura em que este é lido ou escrito.

Assim, existem três componentes no tempo de acesso a um sector do disco:

- o *tempo de procura* (em inglês, *seek time*), que é o tempo que a cabeça magnética demora até chegar à pista onde o sector se encontra.

  Apesar das distâncias serem curtas e das cabeças magnéticas serem muito leves, e portanto demorarem pouco tempo a deslocarem-se para a pista correcta, o tempo de procura é mesmo assim uma componente importante no tempo total de acesso ao disco. Um valor máximo típico para esta componente para os discos de hoje é 8ms, correspondendo ao trajecto maior da cabeça. Naturalmente, em média este valor será menor e perto de metade deste.

- o *tempo rotacional* (em inglês, *rotational latency*), que é o tempo que demora desde que a cabeça magnética se encontra em posição até que o sector desejado passe por baixo desta.

  Actualmente, um valor típico para a velocidade de rotação de um disco rígido é de 10.000 rotações por minuto. Isto significa que uma rotação completa de um cilindro demora 6ms. Como em média se tem que esperar meia volta do cilindro para que o sector certo passe por baixo da cabeça, o tempo rotacional médio será de 3ms.

- o *tempo de leitura/escrita* (em inglês, *access time*), que é o tempo que demora a leitura ou a escrita de um sector.

  Para estimar o tempo de leitura/escrita é necessário saber o número de sectores por pista. Para um valor típico de 64 sectores por pista, e assumindo as mesmas 10.000 rotações por minuto, o tempo de leitura/escrita será 6ms/64=0,09ms.

Pelos valores apresentados, conclui-se que o tempo médio de acesso a um sector do disco é perto de 7,1ms. Duas observações em relação a este valor. Primeiro, quando comparado com o tempo de acesso à memória principal, que hoje em dia é inferior a 100ns, o acesso ao disco é cerca de 100.000 vezes mais lento! Portanto, os acessos ao disco fazem degradar muito o desempenho do sistema e devem ser minimizados.

A segunda observação é que o tempo de leitura/escrita é desprezável face aos tempos de procura e rotacional. Por esta razão, seria desejável aumentar o tamanho do sector. Há aqui um compromisso, pois embora se ganhe em eficiência, poderá haver um grande desperdício de espaço se os sectores forem muito grandes pois, nos casos em que se pretendam guardar pequenas quantidades de informação, o resto do sector ficará desaproveitado. Actualmente, valores típicos para os sectores são entre 512 octetos e 4k octetos.

Uma forma de aumentar a eficiência no acesso ao disco é manter o máximo possível a estrutura lógica dos dados em sectores contínuos. Para isso, muitos sistemas operativos permitem a *desfragmentação* do disco, o que não é mais do

que colocar os ficheiros em sectores consecutivos. Desta maneira, apenas se perde uma vez o tempo de procura e rotacional para a leitura/escrita de vários sectores.

De referir que a leitura/escrita dos dados não se faz directamente para a UCP, pois isso exigiria uma largura de banda muito grande. Por exemplo, se o tamanho do sector for de 1k octetos, então a leitura de um sector gera 1k/0,09ms=11M octetos/s. Assim, o controlador de disco possui um *buffer* para onde são copiados os sectores e só depois serão lidos pela UCP, ao ritmo de transmissão permitido por esta. Para as escritas o processo é semelhante. O sector a escrever é primeiro escrito para este *buffer* e só quando está completo é o mesmo copiado para o disco.

14.3 Comunicação Paralela

Do ponto de vista da comunicação entre o processador e a interface de um periférico, a comunicação faz-se, como se viu, através dos barramentos do computador. Este é um caso de comunicação paralela, isto é, uma comunicação envolvendo vários bits simultaneamente, tipicamente uma palavra do computador ou, em algumas arquitecturas, opcionalmente um octeto.

A comunicação entre a interface e o periférico, por sua vez, pode decorrer em paralelo, da mesma forma, ou em série. Na Secção 14.4 será estudada a *comunicação série*. Nesta secção ir-se-á analisar a *comunicação paralela*.

Um aspecto a ter em conta é o de que, por vezes, a largura da palavra do processador é desadaptada à comunicação com o periférico. Em muitos casos, por exemplo, o barramento de interface com o periférico assume a transferência de um octeto e o barramento interno pode ter 16 bits, como no caso do P3, ou mais, o que acontece em muitos dos microprocessadores comerciais disponíveis. Essa circunstância não cria qualquer problema. No caso de periféricos de saída, a interface ignora simplesmente os bits que não vão ser transmitidos e o programa terá que ter em conta que a informação deve ser formatada de acordo com esse facto. No caso dos periféricos de entrada, a interface não actua as linhas não utilizadas, uma vez que o barramento é construído de forma a que as linhas não actuadas assumam um valor por omissão. Do mesmo modo, o programa terá que ter em conta que a informação útil de entrada ocupa um subconjunto dos bits da palavra.

A complexidade da comunicação entre a interface e o periférico depende da complexidade do próprio periférico e do grau de funcionalidade pedido à interface. Um aspecto particular a ter em conta é o grau de *sincronização* necessário entre a interface e o periférico. A sincronização, a este nível, permite coordenar a transferência de informação, garantindo que a entidade que recebe informação é sinalizada de que a entidade que a envia tem informação disponível no barramento de interligação. Pode ainda ser necessário garantir que a entidade que envia receba uma confirmação explícita de que essa informação foi recebida.

Em periféricos muito simples não é necessária qualquer sincronização entre a interface e o periférico. Noutros tipos de periféricos, estes têm de ser sinalizados sempre que a interface possui dados para enviar ou, em alternativa, em periféricos de entrada, a interface tem de aceitar sinalização do periférico indicando que um dado está a ser enviado. Em periféricos mais complexos, é

necessário sincronizar completamente a comunicação entre a interface e o periférico de modo a, para além de sinalizar o envio, garantir que os dados são recebidos.

Por outro lado, a comunicação entre o processador e a interface pode, igualmente, necessitar de algum grau de sincronização.

Os exemplos apresentados em seguida permitem ter uma visão de interfaces com diferentes complexidades. Não se pretende ser exaustivo nesta questão, mas antes ilustrar alguns tipos de problemas encontrados neste contexto e soluções comuns para esses problemas.

14.3.1 Interfaces sem Sincronização

O primeiro exemplo refere-se a um tipo muito simples de periféricos sem qualquer necessidade de sincronização. Ir-se-á considerar um periférico de entrada constituído por um conjunto de interruptores e um periférico de saída constituído por um conjunto de díodos emissores de luz (LEDs). Quer os interruptores, quer os leds, permitem uma comunicação muito básica entre o utilizador de um sistema e o processador.

A entrada de dados faz-se através de uma instrução de entrada de dados dirigida ao porto ligado aos interruptores. O programa poderá, por exemplo, estar a executar um ciclo que inclui essa instrução, de forma que, em cada ciclo, tem em conta o estado de cada um dos interruptores.

Neste caso, não é necessária qualquer sincronização, não sendo sequer necessário um registo de entrada. Este pode ser substituído simplesmente por um conjunto de *buffers* de três estados que realizam a interface eléctrica. É esta entidade que, neste caso, assume o papel de porto. A Figura 14.7 ilustra a estrutura de uma interface com um porto de entrada deste tipo.

Repare-se que, neste caso, dada a simplicidade da interface, não é necessário existir descodificação interna, uma vez que há apenas um porto. Quando a instrução de leitura do porto é executada, os valores assumidos pelos interruptores são directamente introduzidos no barramento de dados.

Embora seja conceptualmente possível conceber interfaces com portos de saída desprovidos de registos, tal não é, em geral, uma opção útil. Estar-se-ia na presença de um porto de saída que assumiria os valores a ele enviados apenas durante um ciclo de escrita do processador o que, em geral, não é suficiente.

Na Figura 14.8 está representada uma interface de saída que permite ao processador escrever um octeto num conjunto de leds. Mais uma vez a simplicidade da interface dispensa descodificação interna. A instrução de escrita provoca a escrita no porto de um octeto que se manterá visível nos leds até à próxima escrita.

14.3.2 Protocolos de Sincronização

Sincronização por Impulso

Considere-se agora uma interface que está ligada a um conversor digital/analógico (D/A) e a um conversor analógico/digital (A/D). Um dispositivo do primeiro tipo permite converter uma sequência de palavras de valores binários num sinal analógico. Uma aplicação óbvia é a geração de audio num com-

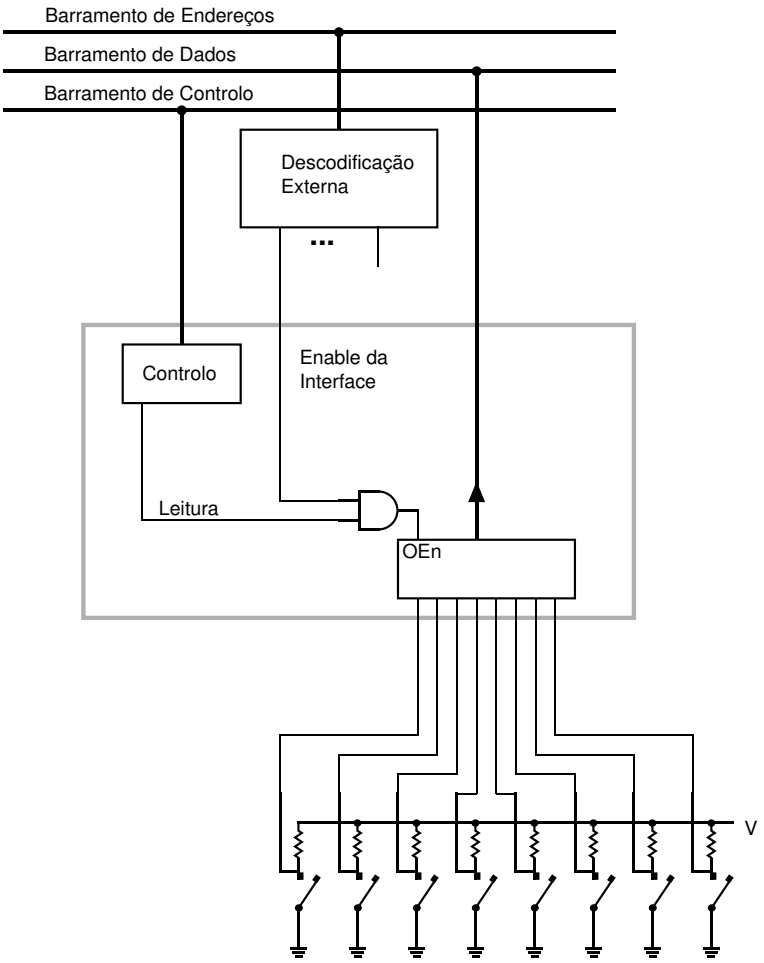


Figura 14.7: Interface de entrada de um conjunto de interruptores.

putador. Trata-se de um periférico de saída. Um conversor analógico/digital permite, por seu turno, converter um sinal analógico variável no tempo, num conjunto de valores numéricos que o representam. Uma aplicação óbvia é um sistema de digitalização de audio. Trata-se de um periférico de entrada.

Em ambos os casos é necessário garantir a transferência periódica de valores binários que permita gerar o sinal pretendido ou amostrar um sinal a digitalizar e guardar os valores sucessivamente obtidos em memória. A responsabilidade dessa geração pode ser atribuída, por exemplo, ao processador com auxílio do sistema de interrupções e de um temporizador, como se verá adiante. No caso da entrada de dados, pode também ser dada a função de gerir essa periodicidade ao circuito do próprio periférico. Na Figura 14.9 ilustra-se uma hipótese de interface e a sua ligação aos dois conversores. Neste exemplo, a geração dos sinais periódicos para o conversor analógico/digital é feita localmente pelo periférico.

Nesta interface existe um flip-flop D que gera um impulso de cada vez que o processador escreve uma palavra no registo da interface. Esse impulso é

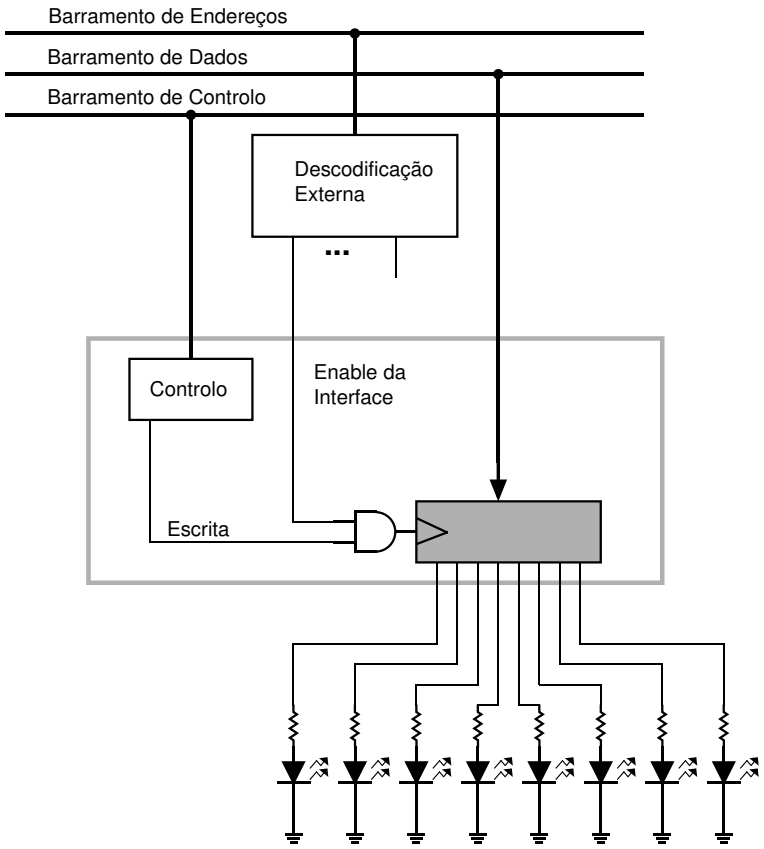


Figura 14.8: Interface de saída para um conjunto de leds.

temporizado a partir do relógio do barramento de controlo e é controlado pela linha de escrita no registo. Desse modo, sempre que há uma escrita (no endereço interno 0, correspondente ao registo de saída), o conversor D/A é avisado através da linha DOUTVAL que sinaliza a existência de um novo dado no barramento de ligação ao conversor.

Inversamente, sempre que o conversor A/D gera um novo dado, coloca-o no barramento de ligação com a interface e actua a linha DINVAL com um impulso. Esse impulso, por um lado, procede ao carregamento do dado no registo de entrada da interface e, por outro, permite actuar uma báscula. Essa báscula pode ser lida (com o endereço interno 3) pelo processador. Desse modo, o processador pode inquirir se há dados novos no registo de entrada. No caso de haver, o processador lê-os (actuando o endereço interno 2) o que simultaneamente desactiva a linha que indica a existência de dados disponíveis. A opção usada de sincronização entre a interface e o processador, através de um registo de estado de um bit com o endereço interno 3, não é a única. Como será posteriormente abordado, poderia ter sido usada a saída da báscula para actuar directamente a linha de interrupção do processador.

Em interfaces reais existem, como é natural, máquinas de estado mais complexas (e seguras) para garantir os processos de sincronização com os periféri-

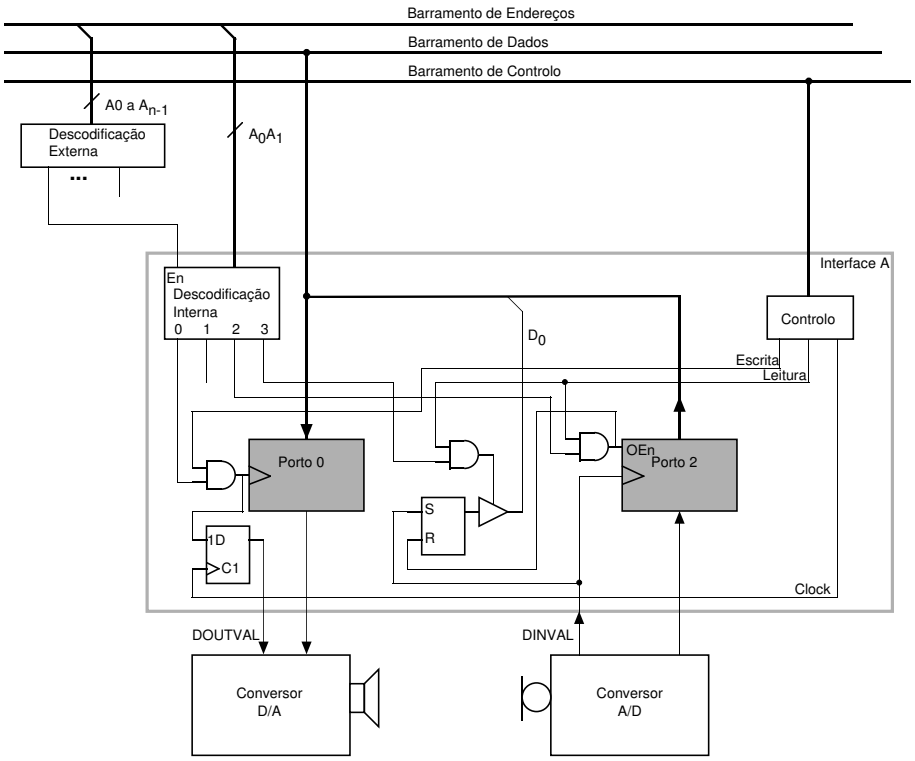


Figura 14.9: Interface bidireccional com um conversor A/D e um conversor D/A.

cos e com o processador.

A nível da troca de sinais entre a interface e o conversor D/A, o diagrama temporal da Figura 14.10 ilustra o funcionamento do protocolo. No instante 1 novos dados são escritos no registo. No instante 2, o facto é assinalado ao periférico. Em 3 não há qualquer sinalização específica, mas o periférico fica a aguardar nova escrita. O periférico deve ter já, nesta fase, guardado a palavra presente no barramento. Em 4, o ciclo recomeça. Esta forma de sincronização é habitualmente designada por *sincronização por impulso*. Este impulso é muitas vezes designado em inglês, por *strobe*.

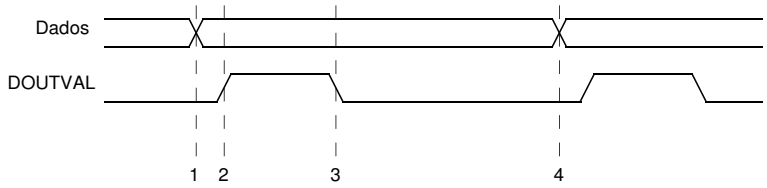


Figura 14.10: Sincronização por impulso.

A forma de sincronização da comunicação entre o conversor A/D e a interface é semelhante. No caso, porém, de, como se referiu atrás, a sincronização poder estar a cargo do processador, o protocolo seria ligeiramente diferente,

uma vez que, agora, o impulso funcionaria, não como confirmação da presença de dados, mas como pedido do seu envio. Na Figura 14.11 ilustra-se essa variante. Agora em 1, o destinatário dos dados pede que eles sejam enviados. Em 2, a fonte dos dados coloca-os no barramento. Em 3, o destinatário assumiu que os dados estavam presentes e armazenou-os. Em 4 o ciclo recomeça.

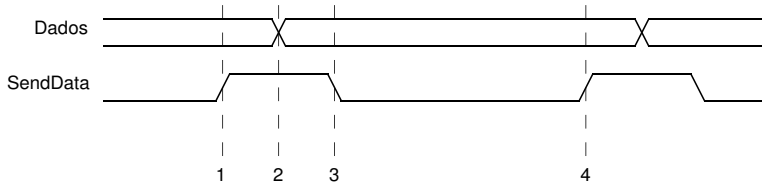


Figura 14.11: Variante da sincronização por impulso.

Protocolos de *Handshaking*

Há, como se pode ver, várias formas de sincronização entre a interface e o periférico. Nas formas apresentadas tem de haver, implícito nas interfaces um conhecimento das temporizações dos interlocutores, uma vez que não há, por parte destes, nenhuma confirmação explícita de que a comunicação se realizou com êxito. Isto quer dizer que é da responsabilidade da entidade que gera os dados saber que o receptor já está pronto para receber novos dados. Em algumas aplicações é impossível, porém, ter esse conhecimento porque não se conhece as características temporais do periférico a que uma interface pode ser ligada.

Para resolver este problema tem de existir, na comunicação entre a interface e o periférico, não só a indicação da entidade geradora de dados que estes se encontram disponíveis, mas também, por parte da entidade destinatária, a indicação de que os dados foram aceites. Isso conduz a um tipo de protocolo de comunicação entre a interface e periférico que tem por designação *handshake*. Nesse protocolo, a entidade geradora de dados (por exemplo, a interface, numa comunicação com um periférico de saída) sinaliza que os dados estão disponíveis por uma linha $DADVAL$, por exemplo, e a entidade receptora (no exemplo referido, o periférico) sinaliza que os dados foram aceites por uma linha reconhecimento, ACK (do inglês, *acknowledge*). A forma concreta como a sinalização é feita pode variar. Podemos considerar impulsos ou mudanças de nível de uma linha. Na Figura 14.12 ilustra-se o protocolo utilizando mudanças de nível das linhas.

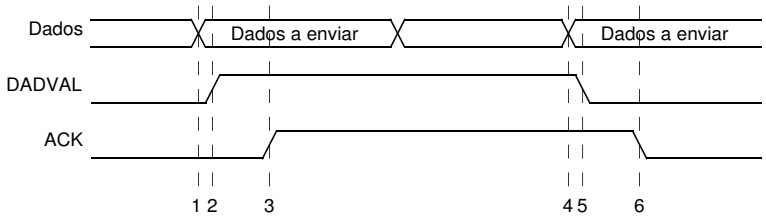


Figura 14.12: Exemplo de um protocolo de aperto de mão.

Na figura os dados a enviar são colocados no barramento no instante 1 e isso é assinalado pela mudança de nível da linha $DADVAL$ no instante marcado com 2. A entidade receptora reconhece a recepção no instante 3 alterando o nível da linha ACK. Assumiu-se que os dados deixavam de estar estáveis a partir do instante assinalado com 4. Nos instantes 5 a 7, repete-se o processo de envio com novos dados. O nível das linhas associadas ao protocolo, neste caso, não tem significado e são apenas as suas transições que suportam a informação. Como já referido podiam ter sido usados impulsos para garantir a sinalização.

Um caso mais interessante é um protocolo que garante uma dupla sincronização: para além da capacidade de dar à entidade geradora de dados a confirmação de que a entidade receptora, de facto, recebeu os dados, como acontece no caso anterior, garante-se ainda, que a entidade receptora é sinalizada de que a entidade emissora recebeu aquela confirmação. Este protocolo é habitualmente designado por *duplo handshaking* e uma das suas possíveis variantes está representada na Figura 14.13.

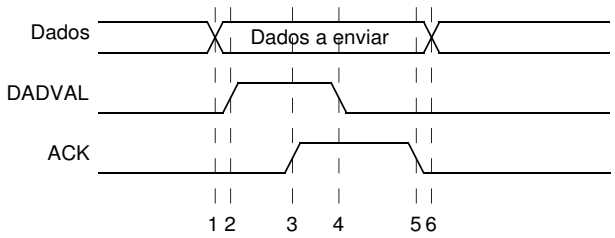


Figura 14.13: Exemplo de um protocolo de duplo aperto de mão.

Na figura os dados são disponibilizados em 1 e o facto é assinalado no instante marcado com 2 pelo activar da linha $DADVAL$. A entidade receptora pode agora ler e registar os dados, respondendo com a activação da linha ACK no instante marcado com 3. A entidade emissora pode, a partir de agora, desactivar a linha $DADVAL$ (instante 4) indicando, assim ao receptor que tomou conhecimento da sua confirmação. Este responde em 5 desactivando a linha ACK, terminando o ciclo e repondo a situação inicial. Na figura os dados foram retirados pela entidade emissora no instante 6, mas podiam ter sido retirados em qualquer momento a partir de 4, por coerência com o significado do nome da linha $DADVAL$, ou mantidos até à ocorrência de novo ciclo.

Repare-se que, neste tipo de protocolo, não há nenhuma necessidade por parte de qualquer das entidades intervenientes de conhecer as características temporais da outra. De facto, se, por exemplo, uma interface estiver ligada a um periférico lento, isso significa apenas que o ciclo é mais longo do que seria com um periférico rápido. Se se tratasse de uma saída de dados, o que aconteceria seria que a linha $DADVAL$ ficaria um tempo longo activada com a interface à espera da activação do sinal ACK e, provavelmente mais tarde, seria necessário esperar de novo pela sua desactivação. No caso inverso de se tratar de uma interface de entrada, seria a linha $DADVAL$ a exibir aquele tipo de lentidão. Em qualquer caso, o ciclo seria longo, mas os dados seriam trocados com toda a segurança.

Nos protocolos de *handshake* ilustrados a iniciativa de realizar a transferência de dados está do lado da entidade emissora. Nada impede que a iniciativa

seja da entidade receptora. No caso do duplo *handshake*, por exemplo, ilustra-se na Figura 14.14 uma situação desse tipo.

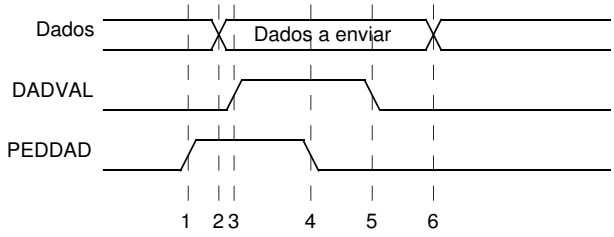


Figura 14.14: Exemplo de um protocolo de duplo aperto de mão com a iniciativa na unidade receptora.

Na figura, a entidade receptora inicia o ciclo no instante 1, pedindo dados à entidade emissora activando a linha $PEDDAD$. A entidade emissora coloca dados válidos no barramento no instante 2 e activa a linha $DADVAL$ no instante marcado com 3, avisando o receptor que os dados presentes no barramento são dados válidos. O instante 4 corresponde ao momento em que a entidade receptora terminou a leitura dos dados e informa o emissor que já não precisa deles. A entidade emissora confirma isso em 5, desactivando a linha que indicava que os dados estavam válidos. A partir daqui o emissor pode retirar os dados do barramento a qualquer momento. No exemplo, os dados foram alterados no instante 6.

A estrutura interna de interfaces capazes de comunicar segundo estes protocolos é, evidentemente, mais complexa que as anteriormente apresentadas. Com os conhecimentos obtidos nos capítulos anteriores, porém, o leitor deverá ser capaz de projectar qualquer delas.

É habitual, neste tipo de interfaces, dispor, para leitura pelo processador, de um registo de estado que permite ao processador saber em que fase se encontra a transferência e se pode, no caso de saídas, colocar mais dados na interface ou, no de entradas, se já existem dados disponíveis na interface. Tal como no caso anteriormente estudado, é possível recorrer ao sistema de interrupções para sinalizar o processador que deve interactuar com a interface.

14.3.3 Interfaces Síncronas

Na arquitectura de interfaces apresentada até aqui, assumiu-se que não existia transferência de relógio entre a interface e o periférico, o que leva à necessidade de estabelecer um processo de sincronização. É esse o caso geral quando um computador interage com um periférico através de uma interface assíncrona, mas não é a única possibilidade. Pode-se conceber, em alternativa, um protocolo de comunicação em que haja um relógio comum à interface e ao periférico. Esse tipo de comunicação tem, naturalmente, a designação de comunicação paralela síncrona.

A comunicação paralela síncrona coloca, ao nível dos periféricos, o mesmo tipo de problemas que os barramentos internos dos processadores, ainda que, em geral, o relógio seja mais lento. Em qualquer caso, algumas interfaces deste tipo permitem a vários periféricos partilhar a mesma interface, reforçando os aspectos comuns aos dois tipos de barramento.

Um aspecto importante a ter em conta neste tipo de interfaces é o que resulta do facto da interface ser interligada a dois barramentos, o barramento interno do computador e o barramento de ligação aos periféricos que têm, em geral, relógios de diferentes frequências. Isso conduz a uma maior complexidade interna deste tipo de interfaces, em relação às interfaces simples anteriormente descritas. Não cabe dentro dos objectivos deste livro explorar este tema em mais detalhe.

14.4 Comunicação Série

Na secção anterior analisou-se a estrutura das interfaces e dos protocolos de entrada e saída, assumindo que a comunicação entre a interface e o periférico se realiza em paralelo, isto é, sendo os vários bits de um octeto ou de uma palavra transferidos simultaneamente. Nesta secção analisar-se-á outro tipo de comunicação que assenta na transferência da informação bit a bit, sendo, por isso denominada *comunicação série*.

As vantagens deste tipo de comunicação sobre a comunicação paralela são as seguintes:

- São necessários menos fios para estabelecer a ligação, conseguindo-se, não só alguma economia, como também, cabos fisicamente mais flexíveis ocupando menos espaço.
- Não há problemas de sincronização entre os sinais das várias linhas existentes na comunicação paralela, o que pode ser um problema em alta velocidade com distâncias significativas.
- É economicamente viável, porque reduzida a menos linhas, a utilização de *drivers* eléctricos de maior potência ou sensibilidade que permitam a comunicação a maiores distâncias que as conseguidas com electrónica digital simples.
- A utilização de comunicação série permite a utilização de ligações através de redes de comunicação (inicialmente através de linhas telefónicas) que seriam de mais complexa utilização com comunicação paralela.
- A comunicação série facilita a utilização de redes locais de comunicação permitindo a relativamente fácil partilha de um canal físico de comunicação por um conjunto de equipamentos.
- Este tipo de comunicação potencia, também, a utilização de canais de comunicação sem fios, por razões semelhantes às já anteriormente referidas.

As interfaces destinadas a comunicação série são estruturadas em torno de um registo de deslocamento. No caso de interfaces de saída, o registo é um registo de carregamento paralelo e saída série. No caso de interfaces de entrada, pelo contrário, trata-se de um registo de entrada série e saída paralela. Obviamente que a estrutura da interface é mais complexa, mas estes registos permitem conciliar a necessidade de comunicação com um periférico com a comunicação em paralelo com o processador através do barramento de dados.

A comunicação série apresenta a vantagem, como foi atrás sugerido, de facilitar a ligação entre dois processadores através de linhas série que interligam interfaces de entrada/saída dos dois computadores. Se bem que uma ligação deste tipo seja concebível com interfaces de comunicação paralela, a utilização de comunicação série facilita muito a tarefa.

Quanto aos sentidos de comunicação entre duas entidades que comunicam em série, sejam uma interface e um periférico, sejam dois computadores, há três tipos de ligação. Quando a comunicação se faz apenas numa direcção, por exemplo, de uma interface para um periférico de saída, chama-se *comunicação simplex*. Não é, actualmente, comum excepto em aplicações muito específicas. A *comunicação half-duplex*, por sua vez, realiza-se nos dois sentidos, mas separadamente, isto é, com a entidade A a enviar dados para a entidade B e, posteriormente, com a entidade B a enviar dados para a entidade A. Neste caso nunca há comunicação simultaneamente nos dois sentidos. Embora em comunicações envolvendo computadores de uso geral seja já pouco usado, este tipo de comunicação é frequente em aplicações de controlo e automação. Por fim, quando entre duas entidades se permite a troca simultânea de comunicação nos dois sentidos, trata-se de *comunicação full-duplex*. É a mais frequente hoje em dia. Repare-se que uma comunicação *full-duplex* do ponto de vista lógico, pode ser fisicamente suportada em comunicações simplex, como acontece, por exemplo, na comunicação entre um processador e um terminal (incluindo monitor e teclado) ou com um modem.

Na comunicação série surge, tal como sucede com a comunicação paralela, a necessidade de sincronizar as duas entidades que trocam informação entre si. Há dois modos fundamentais de o fazer: na *comunicação síncrona*, para além de se transmitirem os sucessivos bits de informação é igualmente transmitido uma sequência de impulsos de relógio que permitem ao receptor extrair os bits de informação do sinal recebido; na *comunicação assíncrona*, pelo contrário, não é transmitido qualquer sinal de relógio e têm de ser assumidos pelas duas entidades envolvidas certos pressupostos no que diz respeito às características temporais do sinal transmitido.

14.4.1 Comunicação Assíncrona

A comunicação assíncrona é a forma de comunicação série que assume mais baixa complexidade e é, historicamente, a primeira a ser utilizada. Na comunicação assíncrona entre duas entidades, como foi já referido, não é transmitido o relógio. A necessária sincronização é garantida pela existência de relógios do lado emissor e do lado receptor com frequências tão próximas quanto possível. Se ambas as entidades estiverem de acordo quanto à frequência de transmissão, o receptor, usando o relógio local, vai amostrar o nível da linha em intervalos sucessivos, separados pela duração do bit. No entanto, como os dois relógios não são exactamente iguais, ao fim de alguns bits corre-se o risco de que um dos bits recebidos não seja lido ou seja lido duas vezes. Para evitar isso, na comunicação assíncrona o número de bits a enviar de cada vez é reduzido. É habitual enviar um caracter de cada vez ou, quando a informação a enviar não é textual, um octeto.

Como vão ser enviadas sucessivas sequências de bits, passa a ser necessário um segundo nível de sincronização que consiste em identificar perante a entidade receptora quando se inicia uma nova sequência. Enquanto que a anterior

se pode designar por *sincronização de bit*, esta nova forma pode ser designada por *sincronização de caracter* ou *sincronização de octeto*.

Uma forma clássica de resolver os problemas apontados está ilustrada na Figura 14.15.

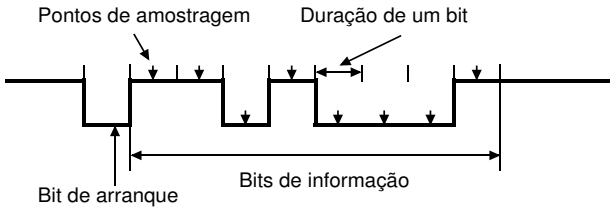


Figura 14.15: Exemplo de envio de um octeto em comunicação assíncrona.

A linha de transmissão permanece num nível determinado quando não estão a ser transmitidos dados. Normalmente opta-se pelo nível H, o nível elevado de tensão. Quando surgem dados para transmissão (no caso do exemplo, um octeto), a transmissão inicia-se obrigatoriamente com a passagem da linha ao nível oposto do nível de repouso durante a duração de um bit. A este intervalo é chamado *bit de arranque* (em inglês, *start bit*). Ao receber este bit de arranque, isto é, ao verificar a existência de uma transição do nível de repouso para o inverso, a entidade receptora calcula os instantes correspondentes ao meio do tempo de duração dos bits transmitidos e, nesses instantes, lê o nível da linha. Desse modo são recebidos os sucessivos valores dos bits transmitidos. No exemplo da figura, é recebido o octeto 10001011 assumindo que, como é usual, se envia o octeto com os bits menos significativos em primeiro lugar.

Para este esquema funcionar, a linha terá de ficar no estado de repouso por um tempo mínimo entre cada par de octetos ou de caracteres. Esse intervalo é, normalmente, o correspondente à duração de um a dois bits. Há, deste modo, a garantia que, mesmo que o último bit transmitido seja um 0, haverá um intervalo com uma duração garantida em que a linha permanece em repouso. A esse intervalo é habitual chamar *bits de guarda* (em inglês, *stop bits*).

Por outro lado, a provável ocorrência de erros de transmissão levou a juntar a todo este esquema, um mecanismo que permita detectar a eventual existência de um erro deste tipo. Para isso usa-se um bit suplementar que indica a *paridade* do conjunto dos bits. É usual designar esse bit por *bit de paridade* (em inglês, *parity bit*). Na emissão, verifica-se se o número de bits de informação a 1 é par ou ímpar e determina-se o valor do bit suplementar, de modo a que a paridade do conjunto total, incluindo-o, seja do tipo desejado. Por exemplo, se se estiver a transmitir octetos e o octeto a transmitir for 10001011, a paridade dos oito bits é par, uma vez que o número de bits a 1 é quatro. Se se pretender paridade par, o bit suplementar será 0. Se se pretender paridade ímpar, o bit será 1.

Na recepção verifica-se de novo a paridade. Se não ocorreu qualquer erro na transmissão, a paridade testada na entidade receptora estará correcta. Se ocorrer um erro num bit, haverá um bit que tem o valor trocado e, portanto, a detecção da paridade vai indicar que houve um erro. Nessas circunstâncias a entidade receptora tem a informação que houve um erro de transmissão e, consequentemente, pode desencadear as acções adequadas, nomeadamente possibilitando à entidade receptora pedir a retransmissão do caracter ou octeto

com erro. Repare-se, contudo, que este método tem uma limitação: se ocorrerem dois erros, ou mais geralmente, um número par de erros, a determinação da paridade não indica qualquer erro. A utilização do bit de paridade é, em geral, deixada como opcional numa transmissão assíncrona.

Na Figura 14.16 está ilustrada a estrutura da informação transmitida incluindo já o bit de paridade (com paridade par) e dois bits de guarda.

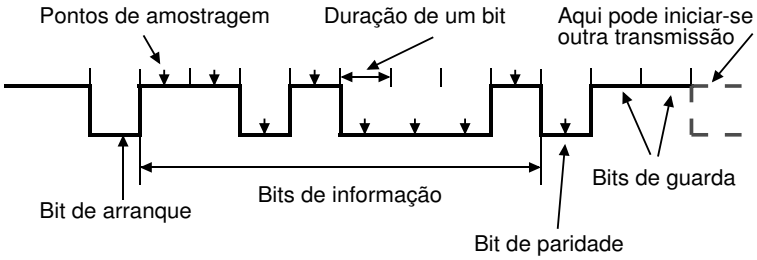


Figura 14.16: Exemplo de envio de um octeto em comunicação assíncrona com bit de paridade e bits de guarda.

Se o relógio de recepção fosse igual ao de emissão, a amostragem dos bits far-se-ia sempre no meio do intervalo de tempo correspondente a cada bit. Na realidade não é isso que se passa porque os relógios não têm exactamente a mesma frequência. Considere-se, por exemplo, o que acontece se o relógio de recepção for ligeiramente mais lento que o de emissão. Observe-se a Figura 14.17. Inicialmente, o aparecimento do bit de arranque permite iniciar a leitura por amostragem dos bits seguintes. Como o relógio é, porém, mais lento que o de emissão, cada bit vai sendo amostrado um pouco mais tarde que o anterior em relação ao meio do período de duração do bit. Resulta óbvio da figura que, por causa deste efeito, o número de bits transmitidos de cada vez tem de ser limitado.

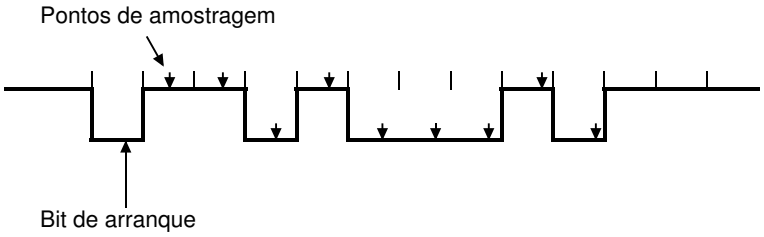


Figura 14.17: Recepção de um octeto com relógio de recepção de frequência ligeiramente inferior ao de recepção.

Como é evidente, nem todos os bits transmitidos numa comunicação assíncrona são bits úteis de informação. De facto, o bit de arranque, o bit de paridade e o intervalo denominado de bits de guarda, são necessários à comunicação, mas não transportam informação útil.

A *velocidade de transmissão* é usualmente referida em bits por segundo (bit/s) e refere-se à quantidade de bits transmitidos por segundo. Note-se que se consideram todos os bits transmitidos, incluindo os de informação útil e os bits de arranque, de paridade e de guarda. Esta velocidade de transmissão é, por

vezes, designada por *baud rate*, embora esta designação seja incorrecta. O *baud* corresponde à mais simples quantidade de informação transmitida. Sempre que 1 *baud* equivale a 1 bit, a designação está correcta. No entanto, certas formas de transmitir informação transmitem vários bits simultaneamente levando a que 1 *baud* corresponde a vários bits.

14.4.2 Comunicação Síncrona

Na comunicação síncrona o relógio que existe na entidade receptora tem exactamente a mesma frequência do usado na emissora. Desse modo, o fenómeno de escorregamento do momento de amostragem dos bits não existe. Há duas formas de garantir que o relógio é o mesmo nas duas entidades: ou ele é transmitido, do mesmo modo que os bits, ou, em alternativa, é transmitida suficiente informação para garantir que é reconstituído na recepção um relógio sincronizado com o relógio de emissão. Adiante se fará uma breve referência à forma de o conseguir.

De qualquer modo, a existência de um relógio sincronizado leva a dispensar a segmentação de informação em entidades de pequena duração, como acontece na comunicação assíncrona. Em princípio, pode agora transmitir-se informação de forma contínua. Claro que isso coloca novas exigências do ponto de vista das interfaces e do nível de prioridade que o processador dá a essa comunicação. Haverá que garantir que a emissão não é travada por falta de dados para transmitir e que, na recepção, não se perde informação por falta de recolha de informação da interface pelo processador.

Persistem, contudo, razões para segmentar a informação ainda que em blocos de maior dimensão do que era usual na comunicação assíncrona. Por um lado, isso resulta da necessidade de permitir que o processador controle a comunicação, gerindo a quantidade de informação que está disponível para enviar ou receber de cada vez. Por outro lado, há que ter em conta que os erros de transmissão existem. Enviar um longo bloco de informação sem o segmentar implica que um simples erro de transmissão comprometa todo o bloco, levando à necessidade de o retransmitir. Por outro lado, blocos demasiado pequenos têm o inconveniente de levar, como se verá adiante, a baixos rendimentos de transmissão. Há, portanto, que encontrar uma dimensão óptima. Actualmente as dimensões envolvidas nas normas mais correntes vão de algumas centenas a muitos milhares de bits.

Nas implementações correntes de comunicação síncrona, é normal o contínuo envio de relógio, mesmo quando não há informação útil a transmitir. Isso coloca o problema de diferenciar a ausência de dados da transmissão de dados úteis, uma vez que o receptor está permanentemente a amostrar a linha.

O desenvolvimento deste tipo de comunicação foi relevante para a implementação de redes de computadores de todos os tipos. Isso implica a necessidade, quando é enviada uma sequência de dados, de explicitar qual é o seu destino, isto é, a que receptor, de entre um conjunto de receptores possíveis, se destina a informação.

Por todas as razões apontadas foi necessário desenvolver *protocolos de comunicação* que dessem suporte a todas as funcionalidades requeridas. Faz-se seguidamente uma breve introdução a essa problemática, mas recomenda-se a consulta de textos sobre comunicação de dados ou redes de computadores

a quem pretenda aprofundar o assunto. Inicialmente desenvolveram-se *protocolos orientados ao caracter*, protocolos em que se assume que a informação a transmitir é constituída por caracteres de texto. Com o desenvolvimento da necessidade de transmitir outro tipo de informação, foram desenvolvidos *protocolos orientados ao bit*.

Protocolos Orientados ao Caracter

Protocolos deste tipo são já pouco usados, mas são aqui referidos por serem um passo importante que faz a ponte da comunicação assíncrona para os modernos protocolos orientados ao bit. Neste tipo de protocolos assume-se que a informação útil é constituída por texto sob a forma de um código e, eventualmente, por alguns caracteres de controlo como os que constam da Tabela 2.13 do código ASCII apresentada na Secção 2.3.3.

Na ausência de dados é transmitido sucessivamente o caracter SYN (*synchronization*), que mantém a sincronização entre a entidade emissora e a receptora. Quando há dados para transmitir, que no caso destes protocolos é, normalmente, texto, a transmissão inicia-se por um cabeçalho com informação liagada ao controlo de transmissão. O cabeçalho é precedido pelo caracter SOH (*Start Of Header*), início de cabeçalho. O texto é, por sua vez, precedido do caracter STX (*Start of TExt*), início de texto e é seguido do caracter ETX (*End of TExt*), fim de texto. O fim do pacote de informação é um caracter BCC (*Block Check Character*), caracer de verificação do bloco. Este não é um caracter específico mas sim um caracter calculado de forma a servir de detecção de erros de transmissão. A informação a transmitir é segmentada em blocos com um comprimento máximo definido. Na Figura 14.18 ilustra-se a estrutura de um pacote de informação neste tipo de protocolo.

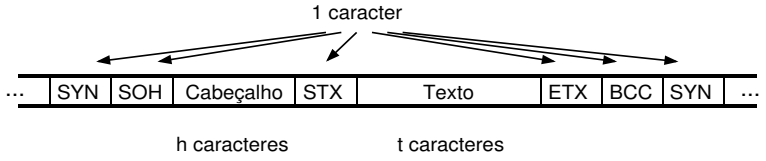


Figura 14.18: Estrutura de um pacote num protocolo de comunicaao síncrona orientado ao caracter.

Protocolos Orientados ao Bit

Na transmissão de informação não textual há o problema de não se poder transmitir configurações aleatórias de bits porque podem ser interpretados como caracteres de controlo (STX, SOH, etc.). De facto só caracteres (de texto ou controlo) podem ser transmitidos. Acontece que muitas vezes há necessidade de transmitir informação que não é "caracterizável", por exemplo, programas. Os protocolos orientados ao bit permitem resolver este problema.

Nos protocolos orientados ao bit perde-se a noção de caracter. Passa a haver uma sequência de bits. Quando não há informação útil a transmitir a entidade emissora transmite continuamente uma sequência de bits chamada *flag* e que é, em muitos dos protocolos correntes, 01111110. Quando surgem dados para

transmitir, transmite-se um cabeçalho e, após este (que tem um comprimento fixo) transmitem-se os dados. Após os dados é transmitido um bloco de verificação de erro, após o que se seguem *flags*. O aparecimento da *flag* marca, portanto, o fim da transmissão. A Figura 14.19 ilustra a estrutura de um pacote de informação neste tipo de protocolo.

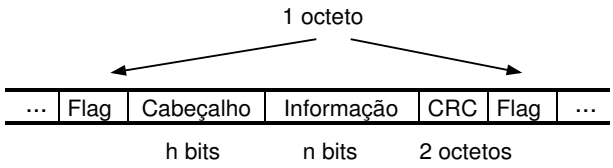


Figura 14.19: Estrutura de um pacote num protocolo de comunicaao síncrona orientado ao bit.

Há um problema com este procedimento. Trata-se do eventual aparecimento da sequência 01111110, correspondente à *flag* no meio da informação a transmitir. Se essa sequência de bits for transmitida, o protocolo assume que o pacote acabou. Desenvolveu-se, por isso, um procedimento adicional chamado *bit stuffing*. Com excepção das *flags*, sempre que o emissor constata a existência de cinco 1s seguidos, acrescenta um 0. Na recepção, sempre que se encontram cinco 1s seguidos, verifica-se o bit seguinte. Se é um 0 retira-se. Se for um 1, está-se perante uma *flag*.

Se, por exemplo, a sequência a transmitir for

$$01010011010111111111110110101,$$

a sequência efectivamente transmitida, por efeito do *bit stuffing* é

$$010100110101111101111\underline{0}10110101$$

em que os 0s sublinhados correspondem aos bits inseridos.

14.5 Modos de Transferência de Dados

Em geral, o processamento dos dados vindos de (ou destinados a) periféricos não é realizado imediatamente na altura da transferência. O normal é o processamento desses dados ser realizado em memória. Assim, existe uma fase de transferência de dados de ou para um bloco de memória, e uma outra fase de processamento destes dados em memória.

A fase de transferência de informação entre o computador e os periféricos destina-se a colocar no periférico informação que está na memória do computador ou, no sentido inverso, a transferir para a memória informação que chega do periférico. Nestas circunstâncias, a participação do processador é puramente instrumental, lendo de um lado para um registo interno e em seguida escrevendo esse registo no outro lado.

Analisam-se em seguida as diferentes alternativas para a fase de transferência de dados entre o computador e os periféricos.

14.5.1 Transferência Controlada por Programa

Neste modo de transferência de informação, o programa a ser executado pela UCP controla toda a fase de transferência. Para além de realizar a transferência propriamente dita, quer entre a UCP e o periférico quer entre a UCP e a memória, tem também a seu cargo a monitorização da interface do periférico para saber quando pode enviar dados ou quando novos dados estão prontos para serem lidos.

Para transferir um bloco de informação, por exemplo, da memória para um periférico, é necessário que o programa tenha um ciclo que lhe permita estar constantemente a testar o porto de estado da interface desse periférico para analisar se ela está pronta a receber dados e, quando isso acontecer, tem de fazer uma leitura de uma posição da memória para um registo seguida de uma escrita desse registo para o porto de dados da interface do periférico, recomeçando o ciclo até toda a informação do bloco ter sido transferida. Este método é chamado de *polling*, no sentido em que a iniciativa de verificar se um dado periférico está pronto é da UCP. Um fluxograma de um programa deste tipo está ilustrado na Figura 14.20. Uma transferência no sentido inverso terá um programa semelhante, em que no ciclo de espera se testa se o periférico tem dados prontos para enviar para a memória.

Esta é a abordagem mais simples em termos de hardware, uma vez que tudo é resolvido pelo software. A grande desvantagem é naturalmente que a UCP fica completamente monopolizada por esta tarefa que é, em geral, uma tarefa lenta. Para muitos casos, a maior parte do tempo a UCP estará no ciclo de espera em que se testa se o periférico está novamente disponível.

14.5.2 Transferência Controlada por Interrupções

Uma forma mais optimizada de realizar a transferência de dados é usar o sistema de interrupções do processador. Assim, mantém-se a transferência de informação a cargo da UCP, mas esta deixa de ter que ficar em ciclo lendo o bit de estado da interface à espera que esta indique a possibilidade de fazer uma nova transferência. Em vez disso, quando o periférico estiver de novo disponível, a sua interface coloca um sinal activo que provocará uma interrupção do processador. A interface deverá manter o sinal de interrupção activo até que a UCP a informe que vai tratar a sua interrupção, normalmente através de um sinal de *confirmação de interrupção* (em inglês, *interrupt acknowledge* – no caso do processador P3 do Capítulo 12, o sinal IAK).

Este modo de transferência permite, no contexto do exemplo anterior, que o processador esteja a executar qualquer actividade e, quando é interrompido, corra a rotina que procede à transferência de dados. A rotina terá a estrutura da Figura 14.21. Pode-se observar que esta rotina corresponde simplesmente à fase de transferência de dados entre a memória e o periférico do programa anterior.

Com esta implementação, a UCP deixa de ter ciclos desperdiçados à espera que o periférico fique disponível. A UCP continua, no entanto, a ter algum tempo de processamento dedicado à transferência, mas este é um tempo útil no sentido em que se está de facto a realizar a cópia dos dados.

Para além disso, esta implementação permite que existam várias transferências a decorrer em simultâneo com diferentes periféricos. Na situação anterior,

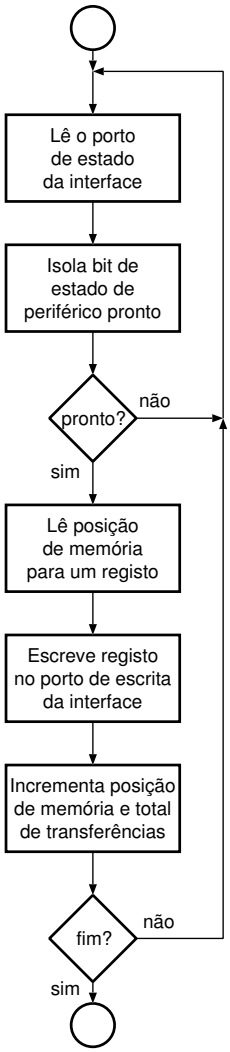


Figura 14.20: Fluxograma de um programa de controlo da transferência de dados por software.

tal não era possível pois a UCP estava dedicada à transferência de um bloco com um periférico, não dando atenção a mais nada, nomeadamente outros periféricos. Com transferências controladas através de interrupções, depois de se lançar a transferência de um bloco para um dado periférico, a UCP fica liberta para iniciar em paralelo uma nova transferência com outro periférico. Esta possibilidade levanta três problemas.

O primeiro problema é que, tal como referido na Secção 11.5.3, durante a execução de uma rotina de tratamento a uma interrupção, as interrupções são automaticamente inibidas de forma a que esta rotina não possa ser interrompida. Em muitos processadores, este problema é minimizado permitindo que

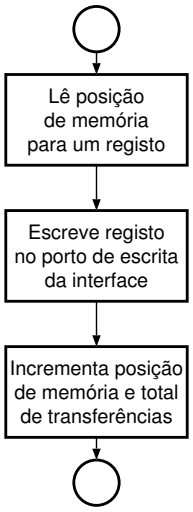


Figura 14.21: Fluxograma de rotina de tratamento de interrupção numa transferência de dados por interrupção.

periféricos mais prioritários possam interromper rotinas de interrupção com menor prioridade. De qualquer forma, pode acontecer a UCP não responder de imediato à interrupção de um periférico caso tenha sido interrompida por outro há pouco tempo e assim ainda esteja a tratar dessa transferência. Em geral, cada dispositivo tem um *tempo de resposta máximo* (em inglês, *timeout*). O comportamento caso esse tempo seja excedido é específico a cada dispositivo, desde o abortar a comunicação com um código de erro, passando pelos que fazem uma nova tentativa de comunicação (até se atingir um número máximo de tentativas), até ao ignorar puro e simples deste erro de transmissão e passagem imediata aos dados seguintes.

Como as rotinas de tratamento à interrupção utilizadas nas transferência de dados são extremamente simples (conforme Figura 14.21), logo com um tempo de execução muito baixo, este não é normalmente um problema sério. Portanto, mesmo em caso de conflito, os tempos de resposta a uma interrupção não excedem, em geral, os tempos de resposta máximos dos dispositivos.

O segundo problema que surge quando existe mais do que uma transferência a decorrer em simultâneo é que quando a UCP recebe uma interrupção tem que ter mecanismos para poder identificar qual foi o dispositivo que gerou essa interrupção. Esta identificação tem como finalidade a selecção da rotina de tratamento à interrupção que deve ser executada, específica para cada periférico.

Uma solução simples para este problema é a UCP disponibilizar para o exterior um conjunto de linhas de interrupções independentes, atribuindo-se a cada periférico do sistema uma dessas linhas. Esta solução pode não ser prática quando o número de periféricos é elevado. A alternativa é usar uma linha de interrupção única (como no caso do P3 estudado no Capítulo 12), mas nesse caso a UCP terá que passar por um processo para a identificação do perifé-

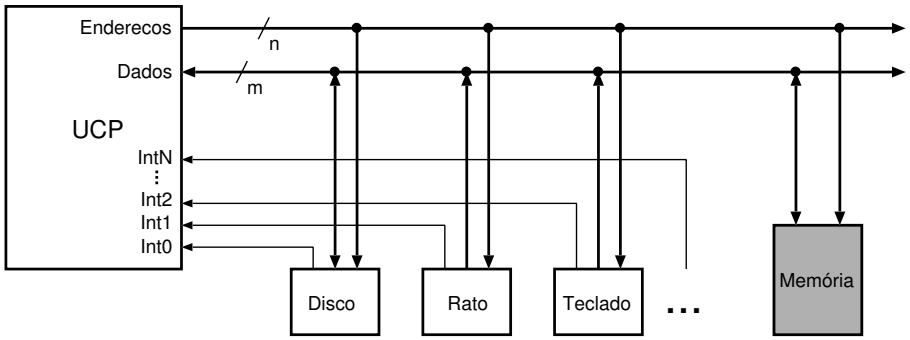


Figura 14.22: Diagrama de processador e periféricos ligados com linhas de interrupção individuais.

rico após a recepção de uma interrupção. Estas alternativas são analisadas em detalhe em seguida. Notar que é possível um processador conjugar as duas alternativas, com um conjunto de linhas de interrupção em que uma ou mais destas partilhem periféricos.

Finalmente, um terceiro problema está relacionado com a possibilidade de vários periféricos gerarem simultaneamente uma interrupção, colocando-se a questão de qual destas interrupções deve ser tratada em primeiro lugar. É necessário, portanto, definir uma ordem de prioridades no atendimento das interrupções. Ao contrário do que se poderia pensar à partida, os dispositivos com maior prioridade não são, em geral, os que interagem com o utilizador. Normalmente, os dipositivos mais rápidos (por exemplo, o disco) têm maior prioridade do que os mais lentos (por exemplo, o teclado), que, como foi observado atrás (ver Secção 14.2), são tipicamente os que fazem a interface com o utilizador. A razão desta ordem de prioridades deve-se a que os dipositivos mais rápidos exigem tempos de resposta máximos menores, pois necessitam que a sua informação seja lida ou escrita rapidamente para que possam ficar prontos para os próximos dados que chegam a ritmo elevado. A maneira como o tratamento destas prioridades é realizado depende da forma de identificação das interrupções e será discutido dentro de cada uma das secções que segue.

Linhas de Interrupção Independentes

A solução conceptualmente mais simples em termos de identificar o dispositivo que gera uma interrupção é o processador disponibilizar linhas individuais de interrupção. Nestas condições, basta ligar a linha de interrupção de cada periférico a uma das linhas de interrupção do processador, como exemplificado na Figura 14.22.

Cada linha de interrupção terá associado um endereço de início da rotina de tratamento à interrupção. Este endereço pode ser fixo e definido à partida. Por exemplo, a interrupção da linha $Int0$ da Figura 14.22 saltará sempre para o endereço 0000h, a da linha $Int1$ no endereço 0010h, etc. Portanto no endereço 0000h deverá estar o início da rotina de tratamento à interrupção do periférico ligado a essa linha. No caso da Figura 14.22 seria a rotina de tratamento das

interrupções do disco.

Uma alternativa mais flexível, e também mais comum, é através da utilização de uma *tabela de rotinas de interrupção*. Esta tabela reside numa posição fixa em memória e terá tantas entradas quantas as linhas de interrupção do processador. Cada entrada está associada a uma das linhas de interrupção e contém o endereço do início da rotina de tratamento ao periférico ligado nessa linha. É possível por software definir as entradas nessa tabela, dando portanto maior flexibilidade no posicionamento das rotinas de tratamento às interrupções.

Para exemplificar, e voltando à Figura 14.22, considere-se que o processador tem 16 linhas de interrupção, de 0 a $N = 15$. Neste caso, a tabela de rotinas de interrupção ocupará 16 posições de memória, estando cada posição desta tabela associada ao índice da linha de interrupção. Considere-se ainda que foi definido para esta UCP que esta tabela começa no endereço 5000h. Caso surja uma interrupção na linha de interrupção i, a execução da rotina de interrupção terá lugar a partir do valor guardado na posição de memória 5000h+i, ou seja, será realizada a transferência $\texttt{PC} \leftarrow \texttt{M}[\texttt{5000h} + \texttt{i}]$.

Com esta organização, está definida à partida uma prioridade fixa para as linhas de interrupção. Por exemplo, têm maior prioridade as linhas de interrupção com índice menor. Assim, caso surja mais do que uma interrupção em simultâneo, será atendida aquela com índice menor. As restantes ficarão pendentes e serão tratadas após a execução da rotina de interrupção desta. Este funcionamento é possível graças à inibição automática das interrupções quando se entra numa rotina de tratamento a uma interrupção. Portanto, como referido atrás, os dispositivos mais rápidos deverão ser ligados às linhas de interrupção com índice menor. No caso da Figura 14.22, pela forma como os periféricos foram ligados às linhas de interrupção, o disco será o periférico com maior prioridade, seguido pelo rato.

Linha de Interrupção Partilhada

A abordagem apresentada atrás limita o número de periféricos capazes de gerar interrupções ao número de linhas de interrupção disponíveis. Uma alternativa é ter uma única linha de interrupção partilhada por todos os periféricos. Quando esta linha é activada, a primeira tarefa da UCP é identificar qual o periférico que a gerou e em seguida executar a rotina de tratamento à interrupção correspondente.

Este processo pode, por sua vez, ser resolvido por duas formas. As interrupções podem ser vectorizadas ou não. Com interrupções vectorizadas, o periférico responsável pela interrupção informa a UCP que foi ele quem a gerou. No caso das interrupções não vectorizadas, terá que ser a UCP a varrer os periféricos até identificar aquele que gerou a interrupção. Analisam-se em seguida estas duas alternativas.

Interrupções Não Vectorizadas

Como referido, com uma linha de interrupção única e com interrupções não vectorizadas será necessário a UCP por software ter um processo de identificar o periférico que requisitou a sua atenção. A forma de o fazer é ler em sequência o registo de estado na interface de cada um dos periféricos até encontrar um

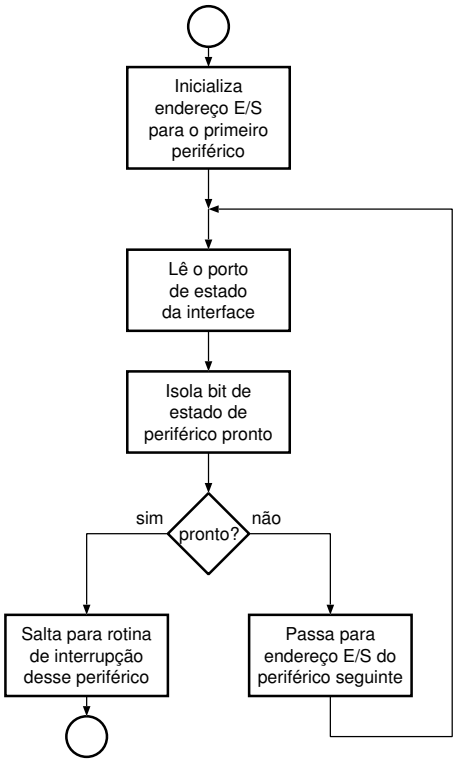


Figura 14.23: Fluxograma da fase inicial da rotina de interrupção para interrupções não vectorizadas que identifica qual é o periférico que deve ser servido.

que esteja pronto a transferir informação. Este procedimento está ilustrado na Figura 14.23.

Este processo é também chamado de *polling*, apesar de ser uma forma diferente *polling* da apresentada na Secção 14.5.1. Aqui sabe-se à partida que haverá um periférico pronto a transferir informação e o objectivo do *polling* é identidicar qual. Anteriormente, o *polling* foi usado no contexto de uma espera activa, em que se fica em ciclo até o periférico com quem se está a realizar a transferência esteja pronto.

Portanto, no caso de interrupções não vectorizadas, a rotina de interrupção é única, sendo o fluxograma da sua fase inicial o apresentado na Figura 14.23. No entanto, uma vez identificado o periférico a servir, esta salta para uma zona de código de tratamento da interrupção específico a esse periférico.

Neste método de interrupções, a maneira de definir prioridades é conseguida através da ordem com que os periféricos são testados. É fácil observar que, caso tenha havido mais do que um periférico a gerar uma interrupção, será atendido primeiro o que for testado em primeiro lugar. De facto, nessa altura, o processador nem se chega a aperceber que há mais periféricos a requerer a sua atenção. Depois de ter tratado do primeiro periférico, como a linha de interrupção continuará activa, haverá uma nova chamada à rotina de tra-

tamento da interrupção e desta vez, se entretanto não tiverem chegado novas interrupções de maior prioridade, já se chegará ao periférico de menor prioridade. Logo, a ordem de teste é igual à ordem de prioridade do periféricos.

Esta é uma abordagem simples e flexível, pois o controlo está do lado do software. Porém, tem o problema de, para a maioria das aplicações, ser demasiado lenta. O ciclo de teste dos periféricos pode demorar bastante tempo, o que é altamente indesejável numa rotina de tratamento a interrupções.

Interrupções Vectorizadas

O caso mais comum para gerir as interrupções num computador é através de interrupções vectorizadas. Com esta abordagem, após a UCP ter activado o sinal de confirmação de interrupção, o dispositivo que gerou a interrupção terá que se identificar, colocando o seu *vector de interrupção* no barramento de dados. Este vector de interrupção não é mais do que um valor numérico único, atribuído a cada periférico do sistema. Da mesma forma que com linhas de interrupção independentes, a UCP utiliza este identificador para obter o endereço da rotina a executar a partir de uma tabela de rotinas de interrupção.

Para gerir as interrupções e o diálogo com a UCP, utilizam-se normalmente *circuitos controladores de interrupções*, *PIC* (em inglês, *Programmable Interrupt Controller*). As funcionalidades típicas de um PIC são:

- aceitar um conjunto de linhas de pedido de interrupção.
- gerir a interface com o processador.
- fornecer um vector correspondente à interrupção mais prioritária, quando pedido pelo processador.
- permitir o mascaramento de interrupções.

Uma estrutura interna possível para um PIC com oito linhas de interrupção está representada na Figura 14.24.

As linhas $INTn$ são as linhas provenientes das interfaces dos periféricos, em que n define o vector do periférico aí ligado. A cada valor de n está associada a prioridade da linha. Assuma-se, por exemplo, que a um valor menor de n está associada uma maior prioridade da linha. Nesta situação, os periféricos de maior prioridade deverão estar ligados às linhas de índice menor.

Todas as linhas de interrupção ficam memorizadas no registo que mantém as interrupções pendentes. Na forma mais simples, bastaria uma porta OR entre todas as saídas deste registo para gerar o sinal INT para a UCP. No entanto, em geral, o PIC permite fazer o mascaramento de interrupções. Para isso, está disponível um registo de *máscara de interrupções* que permite filtrar as interrupções e assim impedir que parte das linhas $INTn$ gerem de facto uma interrupção. Para isso, basta colocar a 0 o bit deste registo com peso igual ao vector de interrupção que se pretende inibir. Este registo está mapeado no espaço de entradas/saídas da UCP, e portanto, por software, é possível definir a cada passo de um dado programa quais são os periféricos de que se aceitam interrupções.

As saídas do registo de interrupções pendentes entram também num codificador com prioridades, como o descrito na Secção 5.2.2. Este codificador gera na sua saída o valor binário correspondente à entrada com menor índice

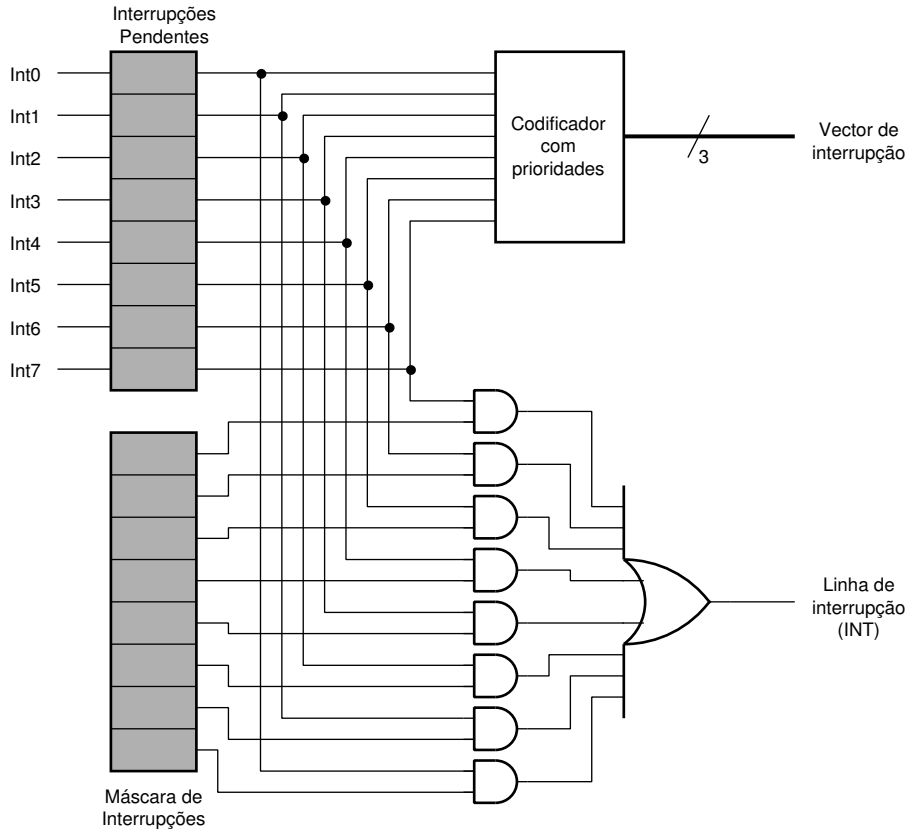


Figura 14.24: Diagrama da organização interna de um controlador de interrupções.

que está a 1 (caso a ordem das prioridades fosse o contrário da definida atrás, bastaria trocar a ordem de prioridades deste codificador). A saída deste codificador é colocada no barramento de dados quando a UCP envia o sinal IAK indicando que está a responder a um pedido de interrupção e que pretende saber o vector dessa interrupção.

Até esta altura do texto, assumiu-se sempre que uma rotina de serviço a uma interrupção não pode ser interrompida, pois o bit de estado que habilita as interrupções é automaticamente colocado a 0 quando se serve uma interrupção. Por vezes, este não é o comportamento ideal. Nomeadamente, num sistema com periféricos com diferentes prioridades é fácil perceber que pode ser desejável permitir que um dispositivo de maior prioridade interrompa a rotina de serviço à interrupção de outro periférico menos prioritário.

Para este fim, o PIC tem, regra geral, um registo extra que mantém informação de qual (ou quais, pois abriu-se agora a porta para uma rotina de interrupção poder ser por sua vez interrompida) o vector de interrupção que está a ser servido, como mostra esquematicamente a Figura 14.25. Este é actualizado com o valor do vector de interrupção activo com maior prioridade na altura em

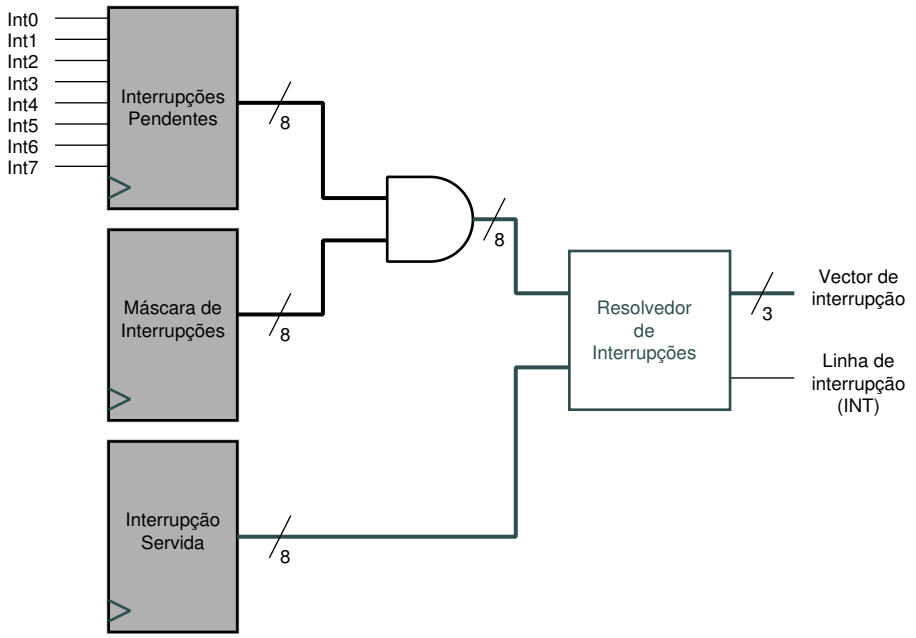


Figura 14.25: Diagrama da organização interna de um controlador de interrupções com registo de informação do vector a ser servido.

que vez o sinal IAK da UCP. Se entretanto chegar uma nova interrupção com prioridade inferior à que está a ser tratada, esta é ignorada até que termine o tratamento da interrupção actual. Caso a prioridade seja mais elevada, então a linha de interrupção INT para a UCP é imediatamente reactivada.

Para que este processo funcione correctamente, a UCP tem que ter um comportamento diferente do que foi apresentado para o caso do processador P3 do Capítulo 12 em dois pontos:

1. o bit de estado `E` que permite ou não o atendimento de novas interrupções não deve ser colocado a zero automaticamente quando da entrada numa rotina de serviço a uma interrupção. Em alternativa, é possível manter o mesmo comportamento do Capítulo 12 desde que a primeira instrução de cada rotina de serviço a interrupções seja um `ENI`.

2. a UCP tem que avisar o PIC do fim da rotina de interrupção, para que este possa limpar a entrada correspondente no registo que mantém informação das interrupções activas.

Naturalmente, existem alternativas ao modelo de PIC aqui apresentado. Nomeadamente, é possível realizar este modo de funcionamento em termos distribuídos. Um exemplo simples é o sistema de *daisy-chain*, em que o periférico mais prioritário recebe sempre o sinal de confirmação de interrupção, IAK. Se não tiver sido ele a gerar a interrupção, ele próprio envia o IAK para o segundo mais prioritário, e assim sucessivamente até se chegar ao dispositivo que gerou a interrupção.

14.5.3 Acesso Directo à Memória

Como se viu no caso da transferência por interrupção, a UCP foi aliviada da tarefa de teste da disponibilidade dos periféricos para nova transferência e pode concentrar-se em outras tarefas, sendo a sua atenção desviada apenas de vez em quando para realizar efectivamente a transferência de uma palavra de dados. Mas mesmo esta transferência utiliza a UCP apenas porque não pode ser feita directamente da memória para o periférico ou vice-versa, uma vez que a UCP se limita a ir buscar a palavra a um dos sítios e colocar no outro.

O objectivo dos dispositivos *DMA* (*direct memory access*, ou *acesso directo à memória*) é exactamente o de permitir esta transferência sem interferência da UCP. Para além de libertar de vez a UCP do processo de transferência de dados, uma segunda vantagem em realizar este tipo de transferência directamente é permitir que, em transferências com periféricos muito rápidos, a UCP não constitua o ponto de estrangulamento da transferência.

Um controlador de DMA é, portanto, um circuito que, sempre que é necessário realizar uma transferência, toma controlo dos barramentos do sistema e coordena a transferência de dados entre a memória primária e um periférico ou vice-versa.

Durante a transferência de dados, a UCP não pode aceder aos barramentos que ligam à memória primária e aos periféricos, logo não pode ir à memória, quer para buscar uma nova instrução, quer para ler dados. Isso não impede, porém, a UCP de continuar a executar uma instrução desde que esta não implique um acesso aos barramentos.

Nos sistemas com memórias cache, o impacto da transferência DMA pode ser substancialmente menor. Durante uma transferência DMA, a UCP pode continuar a aceder normalmente à hierarquia de caches. Assim, enquanto a UCP conseguir obter a partir das caches as instruções e dados de que necessita, pode continuar a sua execução normal. Se ocorrer uma falta na cache, então sim, bloqueia-se à espera de poder novamente aceder aos barramentos.

Antes de assumir o controlo dos barramentos do sistema, o controlador de DMA tem, porém, de pedir à UCP esse controlo e tem que esperar que este lhe seja concedido. Isso realiza-se através de duas linhas, que podem ter a denominação $BUS\_REQUEST$ e BUS_GRANT.

Assim, o controlador de DMA quando pretende realizar uma transferência activa o sinal $BUS\_REQUEST$. O processador, se estiver a realizar um ciclo de leitura ou escrita, termina o ciclo e, logo após, liberta os barramentos, colocando-os em alta impedância e activa a linha BUS_GRANT. O controlador de DMA realiza a transferência e, após isso, desactiva a linha $BUS\_REQUEST$ devolvendo o controlo dos barramentos ao processador.

Arquitectura de DMA

O funcionamento pode ser melhor compreendido com recurso à Figura 14.26, em que, para não complicar desnecessariamente a figura, não está representada a hierarquia de caches. A sequência de operações para uma transferência por DMA é a seguinte:

1. A UCP funciona normalmente, processando instruções e realizando escritas e leituras da memória.

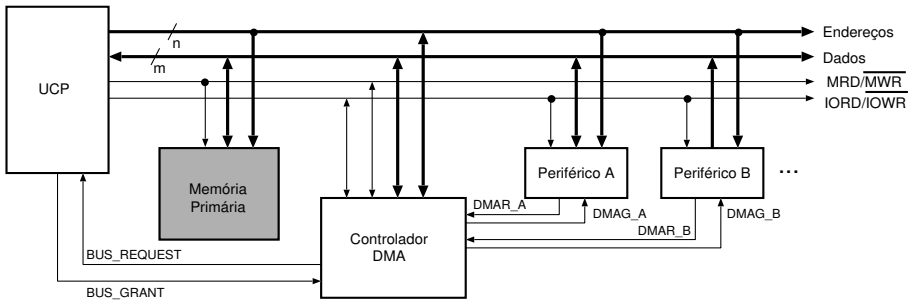


Figura 14.26: Arquitectura do sistema com um controlador DMA.

2. O periférico A, por exemplo, tem uma palavra de dados que deve ser transferida para a memória. A interface deste periférico activa a linha $DMAR\_A$ (em inglês, *DMA Request*), pela qual pede ao controlador de DMA autorização para a operação.

3. O controlador de DMA activa, por seu lado, a linha $BUS\_REQUEST$ e espera por autorização da UCP.

4. Quando possível, a UCP liberta os barramentos, deixando-os em alta-impedância e activa a linha BUS_GRANT.

5. O controlador de DMA coloca o endereço da posição de memória onde escrever no barramento de endereços, coloca a linha $RD/\overline{MW}$ a 0 e activa a linha $DMAG\_A$, dando assim indicação ao periférico da autorização para a transferência.

6. O periférico, recebendo a autorização, coloca os dados no barramento de dados e retira o $DMAR\_A$.

7. O controlador desactiva (respeitando as temporizações de escrita ou leitura da memória) o barramento de endereços, as linhas $RD/\overline{MW}$, $BUS\_REQUEST$ e $DMAG\_A$.

8. A UCP retira a linha BUS_GRANT e retorna à sua actividade normal.

Repare-se que tanto os sinais $BUS\_REQUEST/BUS\_GRANT$ como os sinais $DMAR\_A/DMAG\_A$ seguem um protocolo de *double hand-shaking*.

O Controlador de DMA

O controlador de DMA funciona, durante um ciclo de DMA, como mestre dos barramentos, mas durante o resto do tempo funciona como escravo. Nomeadamente, antes de realizar ciclos de DMA tem que ser programado pela UCP para conhecer os endereços para/de onde transferir dados na memória, qual a quantidade de informação a transferir e qual o tipo de DMA.

Para que esta programação seja possível, o controlador de DMA dispõe internamente de um conjunto de registos que podem ser escritos ou lidos pela

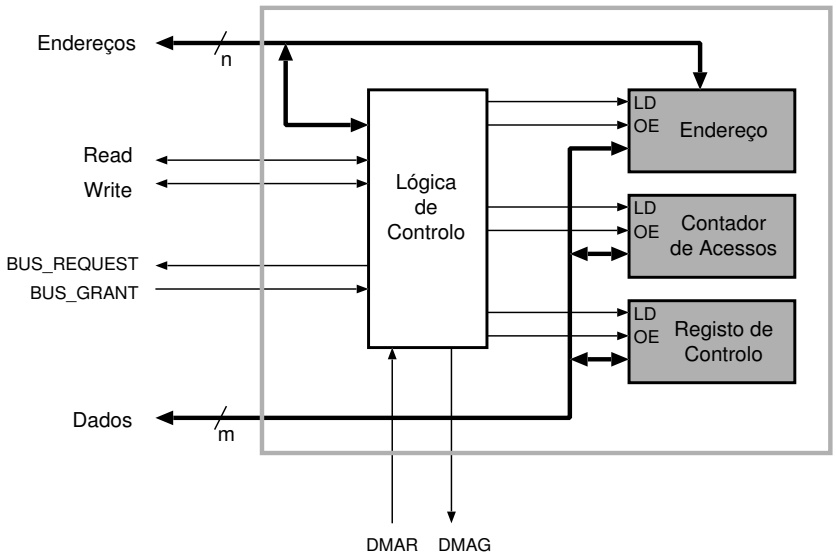


Figura 14.27: Estrutura interna de um controlador de DMA.

UCP. A estrutura interna típica de um controlador de DMA é descrita na Figura 14.27.

No início de uma transferência DMA o processador endereça os registos internos do controlador para programar o tipo e direcção das transferências no Registo de Controlo, o endereço do início da zona de memória envolvida na transferência e o número de acessos a realizar. O processador terá ainda que programar o periférico envolvido, ou melhor, a sua interface. A partir daqui o processador deixa de interferir no processo.

Sempre que um periférico pretende realizar uma transferência, activa a sua linha $DMAR$ (*DMA Request*). O controlo, como se viu, pede controlo dos barramentos através da linha $BUS\_REQUEST$. Quando este é dado pelo processador através da linha BUS_GRANT, o controlador:

1. activa o *buffer* de endereço de modo a endereçar a posição de memória envolvida;
2. activa a linha $READ$ ou $WRITE$, conforme se trate de leitura da memória para o periférico ou o inverso;
3. activa a linha $DMAG$ (*DMA Grant*) do periférico, o que o leva a receber a palavra proveniente da memória pelo barramento de dados ou a transferir a sua palavra interna para o barramento de dados.

Acabado este ciclo, o controlador:

1. retira o pedido $BUS\_REQUEST$;
2. incrementa o Registo de Endereço para ficar a apontar para a posição seguinte de memória;

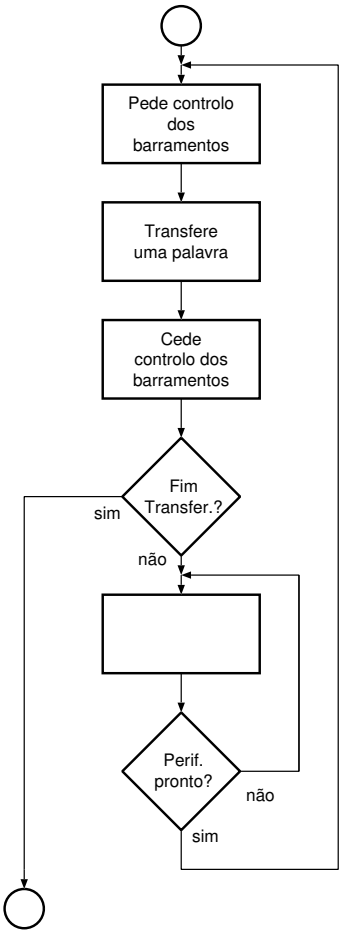


Figura 14.28: Fluxograma de transferência DMA por palavra.

3. decrementa o Contador de Acessos;
4. se o Contador de Acessos ficar a zero, activa a linha de Interrupção para avisar o processador que a transferência programada terminou.

Na figura está ilustrado um controlador com um canal de DMA, mas muitos controladores comerciais possuem mais que um canal. Nesse caso, apenas são replicados os módulos especificamente referentes a cada canal e o controlo tem que gerir prioridades no acesso dos vários periféricos para gerir pedidos concorrentes.

Tipos de DMA

Há vários aspectos que permitem definir as características de DMA. Uma transferência por DMA pode ser:

- De memória para periférico.

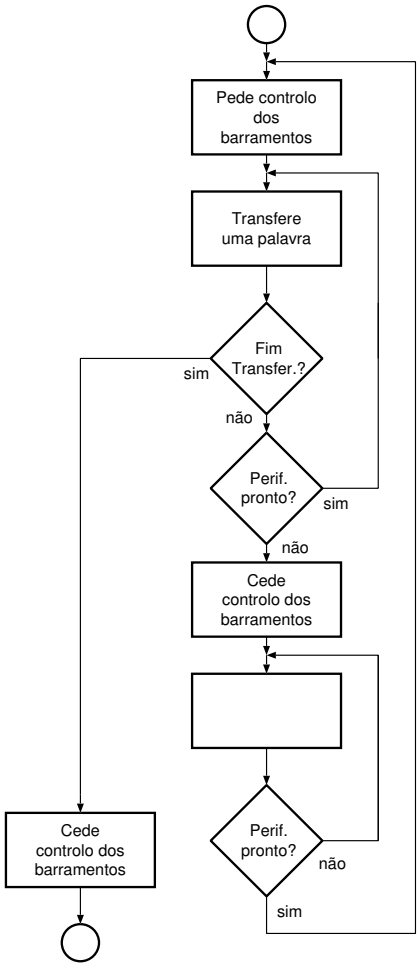


Figura 14.29: Fluxograma de transferência DMA por burst.

- De periférico para memória.

ou com alguma complexidade adicional e procedimentos diferentes:

- De memória para memória.
- De periférico para periférico.

As transferências como as descritas atrás são do *tipo simultâneo*. Neste tipo de transferência, numa transferência de periférico para memória, por exemplo, a leitura do periférico e a escrita de memória são feitas em simultâneo, utilizando o barramento de dados.

As transferência do *tipo sequencial* obrigam o controlador de DMA a realizar sequencialmente a leitura da palavra a transferir da sua origem para um registo interno ao controlador e a escrevê-lo imediatamente a seguir no seu destino.

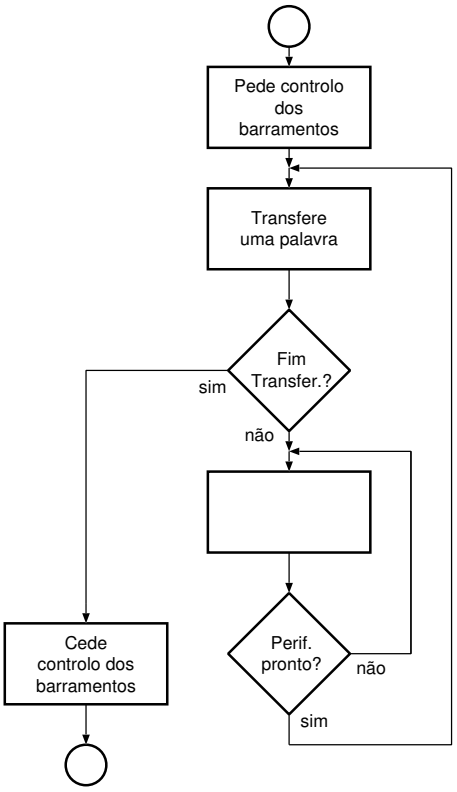


Figura 14.30: Fluxograma de transferência DMA por bloco.

O primeiro tipo é mais rápido e o controlador é menos complexo. No segundo tipo, porém, permite-se transferir informação entre dois periféricos, ou entre duas zonas de memória.

Por outro lado, as transferência por DMA podem ser realizadas em 3 modos diferentes: *modo Palavra*, *modo Rajada* e *modo Bloco*.

No modo de transferência por palavra (em inglês, *word*), cada ciclo de DMA é usado para transferir uma palavra após o que o controlo do barramento é devolvido à UCP, como descrito pelo fluxograma de Figura 14.28.

Na transferência por rajada (em inglês, *burst*), um ciclo de DMA é usado para transferir uma palavra após o que, a haver mais palavras prontas para ser transmitidas, se dá início a uma nova transferência. O controlo do barramento só é devolvido à UCP quando não houver mais dados para transmitir. Este comportamento está representado no fluxograma da Figura 14.29.

Na transferência por bloco (em inglês, *block*), quando arranca uma transferência o controlador DMA permanece com controlo sobre os barramentos até ser transferido o total das palavras para que foi programado, independentemente de entre cada par de palavras o periférico estar ou não pronto. Usa-se este modo para transferir dados de e para periféricos muito rápidos. A Figura 14.30 descreve este modo de transfererência.

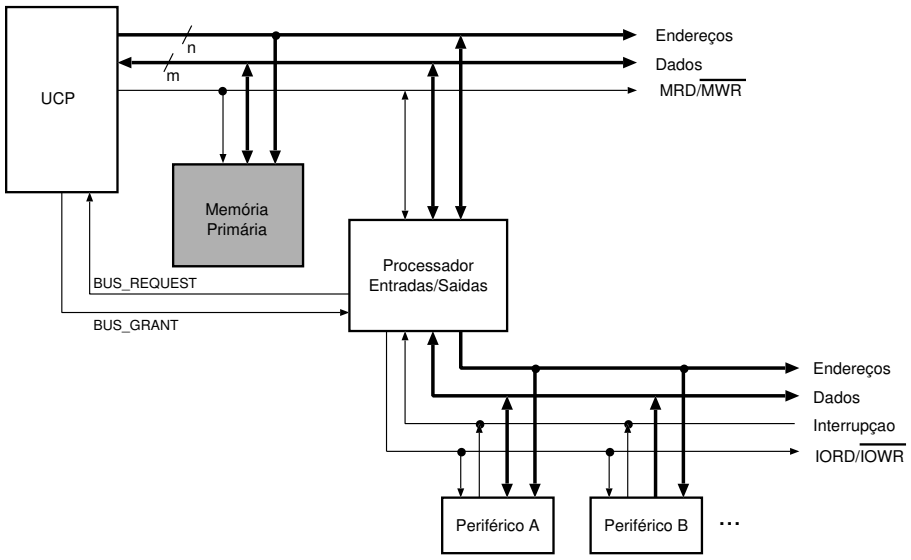


Figura 14.31: Interligação de um processador de entradas/saídas.

14.5.4 Transferência usando um Processador de Entrada/Saída

As transferências DMA permitem libertar a UCP do processo de copiar dados de periféricos para memória ou vice-versa. Um processador de entradas/saídas tem o mesmo objectivo, com a flexibilidade adicional de ser facilmente programável.

A interligação de um processador de entradas/saídas é apresentada na Figura 14.31. Como se pode observar pela figura, os periféricos do sistema comunicam todos com o processador de entradas/saídas. Este está ligado aos barramentos da memória do sistema, mas, em funcionamento normal, estes são controlados pela UCP. Tal como o controlador DMA, o processador de entradas/saídas só pode utilizar estes barramentos depois de ter pedido autorização, e esta ter sido concedida, à UCP.

A vantagem adicional do processador de entradas/saídas é a de, tal como um processador genérico, executar programas. O conjunto de instruções *assembly* que estes processadores executam pode ser mais reduzido que o de um processador genérico, mas, em geral, possuem instruções específicas para leitura e escrita de dados de diferentes fontes. Assim, este processador pode não só transferir informação de um periférico para memória, ou vice-versa, mas pode realizar por si algum tipo de processamento desta informação, reduzindo ainda mais o peso do sistema de entradas/saídas na UCP.

Tal como o controlador DMA, o processador de entradas/saídas funciona normalmente como escravo da UCP. Para além do processador de entradas/saídas ter que pedir autorização para aceder aos barramentos com a memória, é a UCP que define à partida qual o programa que o processador de entradas/saídas deve executar para controlar a transferência de um dado periférico.

Glossário