ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Vinicius Ponte Machado
André Macedo Santana

edufpi

Ministério da Educação - MEC
Universidade Aberta do Brasil - UAB
Universidade Federal do Piauí - UFPI
Centro de Educação Aberta e a Distância - CEAD

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Vinicius Ponte Machado
André Macedo Santana

Reitor
José Arimatéia Dantas Lopes

Vice-Reitora
Nadir do Nascimento Nogueira

Superintendente de Comunicação
Jacqueline Lima Dourado

Editor
Ricardo Alaggio Ribeiro

EDUFPI - Conselho Editorial
Ricardo Alaggio Ribeiro (presidente)
Antonio Fonseca dos Santos Neto
Francisca Maria Soares Mendes
José Machado Moita Neto
Solimar Oliveira Lima
Teresinha de Jesus Mesquita Queiroz
Viriato Campelo

Diretor do Centro de Educação Aberta e a Distância - CEAD
Gildásio Guedes Fernandes

Vice-Diretora do Centro de Educação Aberta e a Distância - CEAD
Lívia Fernanda Nery da Silva

Coordenadora do Curso Licenciatura em Computação
Keylla Maria de Sá Urtiga Aita

Coordenadora de Tutoria do Curso Licenciatura em Computação
Aline Montenegro Leal Silva

EQUIPE TÉCNICA

Revisão de Originais
José Barbosa da Silva
Projeto Gráfico e Diagramação
Vilsselle Hallyne Bastos de Oliveira
Revisão Gráfica
Fabiana dos Santos Sousa

Dados internacionais de Catalogação na Publicação

M149a Machado, Vinicius Ponte.
Arquitetura e organização de computadores / Vinicius Ponte Machado, André Macedo Santana. – Teresina : EDUFPI, 2019.
232 p.

ISBN

1. Computadores - Arquitetura. 2. Computação. I. Santana, André Macedo. II. Título.

De acordo com a Lei n. 9.9610, de 19 de fevereiro de 1998, nenhuma parte deste livro pode ser fotocopiada, gravada, reproduzida ou armazenada num sistema de recuperação de informações ou transmitida sob qualquer forma ou por qualquer meio eletrônico ou mecânico sem o prévio consentimento do detentor dos direitos autorais.

Editora da Universidade Federal do Piauí - EDUFPI
Campus Universitário Ministro Petrônio Portella
CEP: 64049-550 - Bairro Ininga - Teresina - PI - Brasil

Apresentação

Muita gente ainda desconhece a composição e organização dos componentes que integram um instrumento vital para o seu trabalho: o computador. Tal desconhecimento é relacionado, de certa forma, com as inovações tecnológicas que ocorrem quase que diariamente.

Conhecer a organização e o funcionamento dos principias componentes desta máquina permitirá ao leitor saber se adquiriu ou indicou o computador mais adequado à determinada função. Com o intuito de permitir ao leitor conhecer melhor a Arquitetura de Computadores foi desenvolvido este material.

O objetivo desta apostila é proporcionar um entendimento da Arquitetura de Computadores, em especial Processadores e Sistemas de Memória. O texto foi escrito de forma simples e objetiva, e cada capítulo é acompanhado de embasamento teórico-prático, bem como de exercícios. A bibliografia e a webliografia ao fim das notas são mais do que suficientes para que o leitor se aprofunde na teoria apresentada em cada unidade.

Na Unidade I são apresentados os conceitos relacionados à Organização de Computadores. Nela, descrevemos o funcionamento básico de um computador bem como seus principais componentes. Apresentamos, ainda, nesta unidade a linguagem de montagem chamada Assembler. Já a Unidade II trata dos Processadores e contém um Capítulo relacionado a pipelines e outro comparando as arquiteturas RISC e CISC. Por fim, a Unidade III traz os Sistemas de Memória com explicações sobre tecnologias, memória cache e memória virtual.

Boa Leitura !!
Vinícius Machado
André Macedo Santana

SUMÁRIO

UNIDADE I - ORGANIZAÇÃO DE COMPUTADORES........13

1 ARQUITETURA DE COMPUTADORES...........................15

1.1 Introdução 15

1.2 Programas 18

1.3 Funcionamento Básico 22

1.4 Barramento 27

1.4.1 Barramento Local 28

1.4.2 Barramento de Expansão Internos 30

1.4.3 Barramento de Expansão Externos 32

1.5 Processador 34

1.6 Memória 40

1.7 Exercícios 46

2 PROGRAMAÇÃO EM lINGUAGEM DE MÁQUINA.........47

2.1 Linguagem Assembly 47

2.2 Estrutura dos programas em Assembly 48

2.3 Instruções 49

2.4 Diretivas 50

2.5 Processo de criação de programas 51

2.5.1 Registradores da CPU. 52

2.5.2 Programa Debug... 53

2.5.3 Estrutura Assembly 55

2.5.4 Modos de Endereçamento.. 56

2.5.5 Manipulação de Pilhas 57

2.5.6 Criando um programa simples em Assembly............. 58

2.5.7 Armazenando e carregando os programas.................. 61

2.6 Exercícios... 64

Webliografia 65

Referências... 66

UNIDADE II - PROCESSADORES.......................................69

3 PROJETO DE PROCESSADORES.....................................71

3.1 Introdução 71

3.2 Componentes 75

3.3 Funcionalidades 80

3.4 Ciclo de Instrução e Microprogramação...................... 82

3.5 Interrupções.. 85

3.5.1 Características básicas das Interrupções 86

3.6 Medidas de Desempenho 88

3.7 Mais de uma Instrução por Ciclo 89

3.8 Exercícios.. 92

***4. PIPELINE* E MÁQUINAS SUPERESCALARES................93**

4.1 Introdução 93

4.2 Duração de Ciclos 97
4.3 Latência 100
4.4 Bolhas 102
4.5 Previsão de Desvios 102
4.6 Processamento de Exceções 105
4.7 Bolhas causadas por dependência 106
4.8 Medidas de Desempenho 111
4.9 Máquinas Superpipeline e Superescalares 113
4.10 Processadores Superescalares 114
4.11 Exercícios 124

5. RISC e CISC 126

5.1 Introdução 126
5.2 CISC 128
5.3 RISC 135
5.4 RISC vs CISC – Comentários 143
5.5 Exercícios 149
Webliografia 150
Referências 151

UNIDADE III - SISTEMAS DE MEMÓRIA 153

6. MEMÓRIA 155

6.1 Introdução 155

6.2 Conceitos Básicos 155

6.3 Endereços de Memória.. 157

6.4 Ordenação dos Bytes... 159

6.5 Códigos com Correção de Erros................................ 161

6.6 Hierarquia de Memória... 164

6.7 Propriedades de uma Hierarquia 166

6.8 Tecnologias de Memória 169

6.9 Organização dos Chips de Memória.......................... 172

6.10 Exercícios... 178

7 MEMÓRIA CACHE..180

7.1 Introdução 180

7.2 Conceitos Básicos 182

7.3 Associatividade.. 189

7.4 Políticas de Substituição... 198

7.5 Políticas de Atualização 202

7.6 Caches em Vários Níveis.. 205

7.7 Exercícios.. 208

8 MEMÓRIA VIRTUAL...209

8,1 Introdução 209

8.2 Tradução de endereços 211

8.3 Swapping.. 214

8.4 Tabela de Páginas 216

8.6 Proteção 223
8.7 Exercícios 226
Webliografia 227
Referências 228
Sobre os autores 231

UNIDADE I
ORGANIZAÇÃO DE COMPUTADORES

Resumo

O alto nível de popularidade atingido pelos computadores nos últimos anos permitiu também quebrar uma série de barreiras, particularmente no que diz respeito à terminologia associada. Atualmente, expressões como bits, bytes, hard disk, RAM e outras deixaram de fazer parte vocabulário técnico dos especialistas para compor aquele de grande parte dos usuários destas máquinas. Entretanto, os aspectos básicos de funcionamento de um computador ainda são reservados aos profissionais da área e devem assim permanecer indefinidamente.

Nesta Unidade examinaremos alguns aspectos importantes dos componentes e do funcionamento de computadores, sendo que mais detalhes serão apresentados ao longo deste Curso. ***O texto desta Unidade é influenciado fortemente pelo livro Arquitetura de Computadores, de Nicholas Carter, por ser um exemplar de fácil leitura e com uma vasta quantidade exercícios e pelas notas de aula, muito bem elaboradas, do Professor Marcelo Rebonatto.***

O Capítulo é acompanhado de exercícios sem a solução. Preferimos deixar o prazer desta tarefa ao leitor. A bibliografia e a webliografia ao fim das notas são mais do que suficiente para adquirir um conhecimento

razoável sobre a organização de computadores. ***Ao término da leitura desta Unidade, o estudante deverá: a) Compreender o funcionamento básico de um computador; b) Entender o funcionamento de programas; e c) Ser capaz de identificar os componentes principais de um computador.***

1. ARQUITETURA DE COMPUTADORES

1.1. Introdução

Numa visão externa de grande parte dos usuários, um computador é composto de CPU, teclado, vídeo e mouse, como ilustrado pela figura 1.1. Esta é a visão da maior parte da população que tem algum tipo de interação com o computador, principalmente porque estes são os elementos de um microcomputador que se deve transportar, desconectando-os e reconectando-os para fazer o computador voltar a funcionar.

Figura 1.1: Visão externa de um computador.

Este tipo de visão dos componentes de um computador não vai afetar profundamente a vida e modo de utilização destas máquinas pela maior parte da população, é lógico que esta é uma visão extremamente superficial até para alguns usuários.

Para estudar organização de computadores alguns conceitos básicos devem ser assimilados/revisados. Estes conceitos abrangem os componentes funcionais básicos e as formas de representação/armazenamento de informações, além do funcionamento básico dos sistemas de computação.

A maioria dos sistemas de computadores pode ser dividido em três subsistemas: o **processador** (*Central Prossessing Unit - CPU*); a **memória**; e o **subsistema de entrada e saída** (E/S). O processador

é responsável pela execução dos programas, a memória fornece espaço de armazenamento para os programas e os dados aos quais ele referencia, e o subsistema de E/S permite que o computador e a memória controlem os dispositivos que interagem com o mundo externo ou que armazenam dados, como o DVD-ROM, discos rígidos, placa de vídeo/monitor, etc.

De forma bem resumida os componentes básicos de um computador são:

- ***Processador ou CPU***: é o componente vital do sistema de computação responsável pela realização das operações de processamento (os cálculos matemáticos com os dados, por exemplo) e pelo controle de quando e o que deve ser realizado, durante a execução de um programa;

- ***Memória***: é o componente de um sistema de informação cuja função é armazenar as informações que são, foram ou serão manipuladas pelo sistema. Os programas e os dados são armazenados na memória para execução imediata (memória principal) ou para execução ou uso posterior (memória secundária). Basicamente, há duas únicas ações que podem ser realizadas: a) a de guardar um elemento na memória, o que chamamos de armazenar, e a operação associada a esta ação é de escrita ou gravação (*write*); ou b) recuperação de um elemento da memória, ação de recuperar, e operação de leitura (*read*).

- ***Dispositivos de Entrada e Saída***: serve basicamente para permitir que o sistema de computação se comunique

com o mundo externo, realizando, ainda, a interligação, a conversão das linguagens do sistema para a linguagem do meio externo e vice-versa. Os seres humanos entendem símbolos como A, b, *, ?, etc., e o computador entende sinais elétricos que podem assumir um valor de +3 Volts para representar 1 e 0 Volts para representar 0. O teclado (dispositivo de ENTRADA) interliga o usuário e o computador, por exemplo, quando pressionamos a tecla A, os circuitos eletrônicos existentes no teclado "convertem" a pressão mecânica em um grupo de sinais elétricos, alguns com voltagem alta(bit 1) e outras com voltagem baixa (bit 0), o que corresponde, para o computador, ao caractere A. Os dispositivos de SAÍDA operam de modo semelhante, porém, em sentido inverso, ou seja, do computador para o mundo exterior, convertendo os sinais elétricos em símbolos conhecidos por nós.

Além dos três itens relacionados anteriormente, na maioria dos sistemas o processador tem um único barramento de dados que é conectado ao módulo comutador, embora alguns processadores integram diretamente o módulo de comutação no mesmo circuito integrado que o processador de modo a reduzir o número de chips necessários para construir o sistema, bem como o seu custo.

O comutador se comunica com a memória através de um barramento de memória, um conjunto dedicado de linhas que transfere dados entre estes dois sistemas. Um barramento de E/S distinto conecta o comutador com os dispositivos de E/S. Normalmente são utilizados barramentos separados porque o sistema de E/S geralmente é projetado de forma a ser o mais flexível possível para suportar diversos tipos

de dispositivos de E/S, e o de memória é projetado para fornecer a maior largura de banda possível entre o processador e o sistema de memória. A figura 1.2 apresenta a organização básica de um computador.

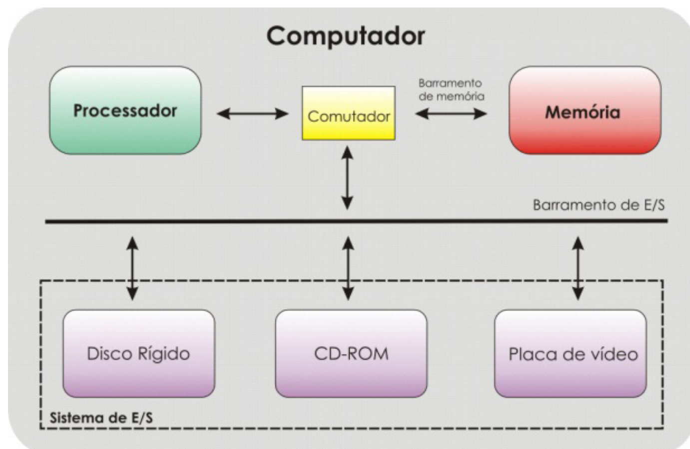


Figura 1.2: Organização básica de computadores
Figura adaptada de [Carter, N.]

1.2. Programas

Programas são sequências de instruções que dizem ao computador o que fazer, embora a visão que o computador tenha das instruções que compõem um dado programa seja muito diferente da visão de quem escreveu. Para o computador, um programa é composto de uma sequência de números que representam operações individuais. Essas operações são conhecidas como **instruções de máquina** ou apenas **instruções**, e o conjunto de operações que um dado processador pode executar é conhecido como **conjunto de instruções**.

Praticamente todos os computadores em uso atualmente são computadores com **memória de programa** que representam os pro-

gramas como números que são armazenados no mesmo espaço de endereçamento que os dados. A abstração de programas armazenados (representando instruções como números armazenados na memória) foi um dos principais avanços na arquitetura dos primeiros computadores. Antes disso, muitos computadores eram programados pelo ajuste de interruptores, ou refazendo a conexão das placas de circuito para definir o novo programa, o que exigia uma grande quantidade de tempo, além de ser muito sujeito a erros.

A abstração de programas armazenados em memória fornece duas vantagens principais sobre as abordagens anteriores. Primeiro, ela permite que os programas sejam armazenados e carregados facilmente na máquina. Uma vez que o programa tenha sido desenvolvido e depurado, os números que representam as suas instruções podem ser escritos em um dispositivo de armazenamento, permitindo que o programa seja carregado novamente para a memória em algum momento. Segundo, e talvez de modo mais significativo, a abstração de programas armazenados em memória permite que os programas tratem a si mesmos ou a quaisquer outros programas como se fossem dados.

Os programas que tratam outros programas como dados são muito comuns, e a maioria das ferramentas de desenvolvimento de programas caem dentro desta categoria. Essas ferramentas incluem os compiladores que convertem programas em linguagem de alto nível, como C e JAVA, em linguagem de montagem (assembly); os **montadores** que convertem instruções de linguagem de montagem em representações numéricas utilizadas pelo processador; e os **ligadores**(*linkers*) que unem diversos programas em uma linguagem de máquina em um único arquivo executável. Também são incluídos nesta categoria os **depuradores** (*debuggers*), programas que apresentam o

estado de outro programa à medida que este é executado, de modo a permitir que os programadores acompanhem o progresso de um programa e encontre os erros.

Os primeiros computadores em linguagem de montagem eram programados em **linguagem de máquina**, que nada mais era do que a representação numérica das instruções utilizadas internamente pelo processador. Para escrever um programa em linguagem de máquina, o programador determinava a sequência de instruções de máquinas necessárias para gerar o resultado correto e dava entrada nos números que representavam no computador essas instruções.

O primeiro passo para simplificar o desenvolvimento de programas veio quando foram desenvolvidos os montadores, permitindo que os programadores codificassem em linguagem de montagem. Na linguagem de montagem, cada instrução de máquina tinha uma representação em texto (como ADD, SUB, ou LOAD) que representa o que ela fazia e os programas eram escritos utilizando essas instruções. Uma vez que o programa tivesse escrito, o programador executava o montador para converter o programa em linguagem de montagem para linguagem de máquina, o qual podia ser executado no computador. As linhas a seguir mostram um exemplo de uma instrução em linguagem de montagem e a instrução em linguagem de máquina gerada a partir dela.

Linguagem de montagem: ADD,r1,r2,r3
Linguagem de máquina: 0x04010203

Utilizar linguagem de montagem tornou a tarefa de programação muito mais fácil, ao permitir que os programadores utilizassem um formato de instruções que era mais fácil de ser entendida por hu-

manos. A programação era extremamente tediosa porque para executar operações mais complexas era necessário utilizar várias instruções; além disso, as instruções disponíveis para os programadores diferem de máquina para máquina. Ou seja, se um programador quisesse executar um programa em um tipo de computador diferente, o programa tinha que ser completamente reescrito na nova linguagem de montagem desse computador.

As linguagens de alto nível foram desenvolvidas para resolver este problema, uma vez que permitem que os programas sejam escritos em muito menos instruções que as linguagens de montagem. Outra vantagem de escrever programas em linguagem de alto nível é que elas são muito mais portáveis do que programas escritos em linguagem de máquina. Programas de alto nível podem ser convertidos para uso em diferentes computadores pela re-compilação do programa, utilizando-se o compilador adequado ao novo computador.

O problema com as linguagens de alto nível é que os computadores não podem executar diretamente instruções em linguagem de

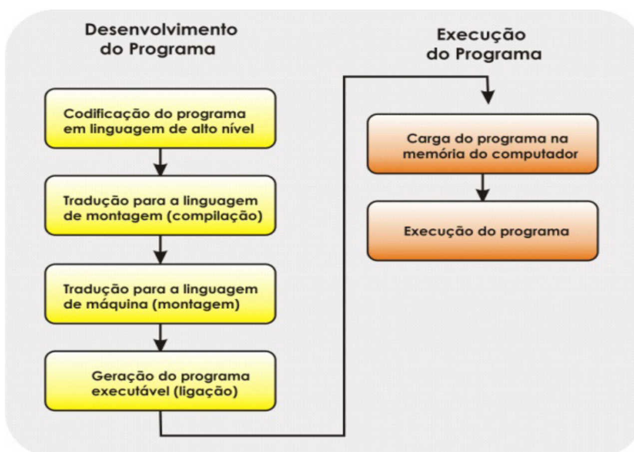


Figura 1.3: Desenvolvimento de um programa.
Figura adaptada de [Carter, N.]

alto nível. Assim, um programa chamado compilador é utilizado para converter o programa em seu equivalente em linguagem de montagem, que é, então, convertida em linguagem de máquina pelo montador. A figura 1.3 ilustra o desenvolvimento e a execução de um programa em linguagem de alto nível.

Uma alternativa para a compilação de um programa é utilizar um **interpretador** para executar a versão do programa em linguagem de alto nível. Os interpretadores são programas que tomam programas em linguagem de alto nível como entradas e executam os passos definidos para cada instrução do programa em linguagem de alto nível produzindo o mesmo resultado que compilar o programa e, então, executar a versão compilada. Os programas interpretados tendem a ser muito mais lentos do que programas compilados o que leva os interpretadores serem menos utilizados que os compiladores.

1.3. Funcionamento Básico

Os computadores, no seu funcionamento básico, executam quatro funções distintas sendo elas: a) Entrada; b) Processamento; c) Armazenamento/recuperação de dados; d) Saída. A figura 1.4 ilustra o funcionamento básico dos sistemas de computação.

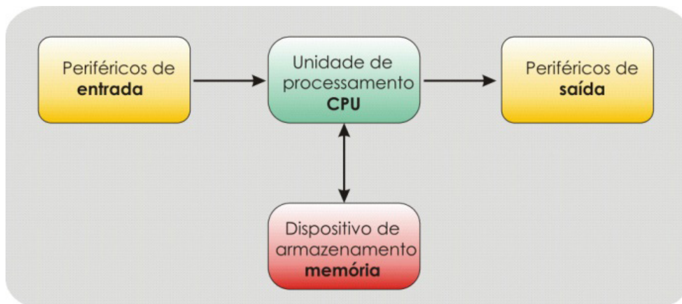


Figura 1.4: Componentes básicos de um computador.
Figura adaptada de [Rebonatto, M.]

Para que um computador trabalhe é necessária **a** inserção de informações (entrada). Seguindo as instruções fornecidas pelos programas, o computador processa os dados oriundos da entrada (processamento) armazenando-os logo em seguida para posterior utilização. As informações produzidas ficam disponíveis para utilização (saída) e a menos que se deseje utilizá-las e produzi-las novamente, elas devem ser armazenadas em um dispositivo de armazenamento estável.

O esquema geral da figura 1.4 é seguido por praticamente todos os computadores, sendo que os dados são produzidos através de instruções durante a etapa de processamento, realizada pela CPU. Cada processador tem um conjunto único de instruções para processar os dados, porém, geralmente utilizam a mesma forma de composição das instruções. As linhas a seguir mostram a forma das instruções comumente utilizada pelos processadores – *OPERAÇÃO* + *OPERANDOS* = *INSTRUÇÃO,* sendo a **operação** especifica a função a ser desempenhada, por exemplo, soma, armazene ou desvie, entre outras. Os **operandos** fornecem os dados a serem utilizados na operação e permitem alcançar a posição destes dados na memória.

Todas as informações manipuladas pelos computadores devem ser entendidas pela máquina. Como o computador é um dispositivo eletrônico, ele armazena e movimenta as informações de forma eletrônica, podendo utilizar um valor de corrente. Para que a máquina represente todos os símbolos da linguagem humana eletricamente são necessárias mais de 100 diferentes voltagens (corrente). Uma máquina desse tipo além de ser de custo elevado, seria difícil de construir e de baixa confiabilidade. Desta forma, opta-se por construir máquinas binárias capazes de entender apenas dois valores diferentes.

Os computadores digitais são totalmente binários, isto é, tra-

balham apenas com dois valores, tornando, dessa forma, simples o emprego da lógica booleana (Sim/Não, Verdadeiro/ Falso, Aberto/Fechado, etc.) tanto na construção de componentes quanto como base para a escrita de programas (programação).

Convencionou-se chamar os dois níveis elétricos de 0 e 1, sendo que cada algarismo da representação numérica binária é denominado de **bit**, correspondente a abreviatura de *binarydigit* (dígito binário). Obviamente, com apenas 1 bit isolado pode-se representar muito pouca coisa (apenas 2 valores), desta forma, usam-se agrupamentos ordenados de bits para a representação de informações úteis. A menor informação inteligível aos seres humanos é o **caractere**, como por exemplo, o número "5" ou a letra "a". Existem diversos agrupamentos de bits para representar caracteres, e o mais popularmente utilizado é chamado de **byte**. Um byte é uma sequência ordenada de 8 bits, sendo cada bit tratado de forma independente dos demais e com um valor fixo de acordo com sua posição. Qualquer sequência binária pode ser convertida para um número na base decimal, sendo utilizado este valor para encontrar o caractere correspondente, utilizando uma tabela de caracteres.

As memórias geralmente armazenam e recuperam informações byte a byte ou em múltiplos de bytes. A representação binária de valores também é utilizada para a representação de números dentro dos computadores. A metodologia é a mesma, sendo convertido o valor em base decimal para o correspondente em base binária. Por exemplo, o número 2310 pode ser armazenado no seguinte byte 00010111.

Os dispositivos de memória atuais utilizam agrupamentos de bytes para representar sua capacidade de armazenamento. Uma vez que tais agrupamentos são oriundos de uma base binária, o fator de

multiplicação utilizado é 1024 (2^{10}). Cada faixa possui também uma letra para abreviar a categoria. A Tabela 1 demonstra alguns agrupamentos de bits e bytes utilizados.

Tabela 01: Agrupamento de bits e bytes.

Agrupamento	*Símbolo*	*Representa*
Byte	*B*	*8 bits*
Kilo	*K*	*1024 Bbytes*
Mega	*M*	*1024 KBytes*
Giga	*G*	*1024 MBytes*
Tera	*T*	*1024 GBytes*
Penta	*P*	*1024 TBytes*

Estes agrupamentos são utilizados na descrição das capacidades de armazenamento dos computadores, hoje em dia, sendo que esta capacidade é referenciada por um número e um símbolo correspondente. Por exemplo, 8 GB de memória.

Outro conceito importante muito utilizado na descrição do mecanismo de transferência de informações entre a CPU e a memória principal é o conceito de **palavra**. A palavra é utilizada para indicar a unidade de transferência e processamento de um computador. As palavras são múltiplos de 1byte, sendo que os microprocessadores geralmente utilizam 32bits – 4 bytes como tamanho da palavra. Porém, já existem projetos e microprocessadores que utilizam palavras de 64 bits. Em outras palavras, os conceitos de bit, caractere, byte e palavra podem ser resumidos assim:

- Bit: é a menor unidade de informação armazenável em

um computador. Igualmente é o termo que representa a contração das palavras inglesas *BinaryDigi*t e pode ter, então, somente dois valores: 0 e 1. Evidentemente, com possibilidades tão limitadas, o bit pouco pode representar isoladamente; por essa razão, as informações manipuladas por um computador são codificadas em grupos ordenados de *bits*, de modo a terem um significado útil.

- Caractere: é o menor grupo de bits representando uma informação útil e inteligível para o ser humano. Qualquer caractere a ser armazenado em um sistema de computação é convertido em um conjunto de bits previamente definidos para o referido sistema (chama-se código de representação de caractere). Cada sistema poderá definir como (quantos bits e como se organizam) cada conjunto de bits irá representar um determinado caractere.

- Byte: é o grupo de 8 bits, tratados de forma individual como unidade de armazenamento e transferência. Como os principais códigos de representação de caracteres utilizam 8 bits por caractere, os conceitos de byte e caractere tornam-se semelhantes e as palavras quase sinônimas.

- Palavra: é um conjunto de bits que representam uma informação útil, mas está associada ao tipo de interação entre a memória principal e a CPU, que é individual (informação por informação). Ou seja, a CPU processa informação por informação, armazena e recupera número a número (cada uma estaria associada a uma palavra).

1.4. Barramento

Os processadores são circuitos integrados passíveis de serem programados para executar uma tarefa predefinida, basicamente manipulando e processando dados. A CPU manipula dados de acordo com programas que deverão, para isso, estar na memória. Um programa pode ordenar que dados sejam armazenados de volta na memória ou recuperar programas/dados armazenados em sistemas de memória de massa (disquetes, HD, etc.). Os caminhos por onde estas informações circulam em um computador é genericamente conhecido como barramento.

Em outras palavras, os barramentos nada mais são do que um conjunto de condutores (fios, trilhas) por onde passam os bits. Possuem duas principais características: A largura do barramento: número de bits transportados numa operação; A freqüência de operação: velocidade com que os dados são transmitidos. A figura 1.5 ilustra a funcionalidade dos barramentos.

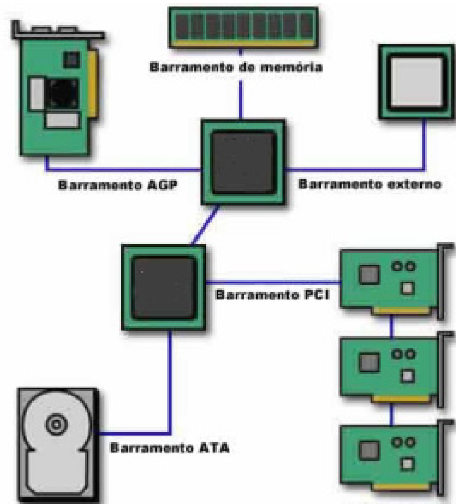


Figura 1.5: Barramentos de um computador. Figura adaptada de [Rebonatto, M.]

Existem diversos barramentos nos computadores atuais, sendo os principais o **barramento local** e **barramento de expansão**. O barramento local é vital para o funcionamento do computador, pois

interliga o processador a memória. Por outro lado, o barramento de expansão interliga os demais componentes, tais como periféricos de entrada, de saída e memória de armazenamento estável (secundária).

1.4.1 Barramento Local

O barramento local, também conhecido como interface CPU/Memória Principal, é de vital importância para o funcionamento do computador, uma vez que os blocos que formam a CPU não podem manipular diretamente os dados armazenados na memória, pois somente operam sobre valores mantidos em registradores. Desta forma, como as instruções e os dados a serem manipulados estão na memória, para que uma instrução seja executada, a memória deve ser acessada no mínimo uma vez. O barramento local é dividido em outros três: **Barramento de Dados**, **Barramento de Endereços** e **Barramento de Controle**. A interface CPU/MP se completa com os registradores auxiliares no acesso a memória. A figura 1.6 ilustra o esquema da interface CPU/MP.

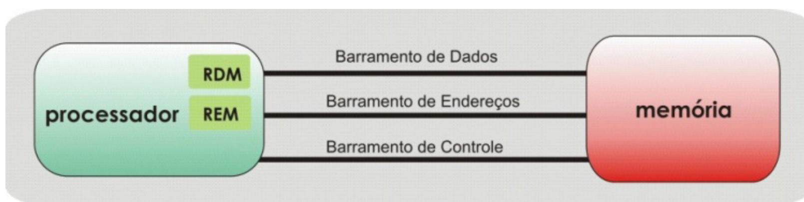


Figura 1.6: Barramento local.
Figura adaptada de [Rebonatto, M.]

- ***Registrador de Endereço da Memória (REM)***: armazena o endereço da célula onde deve ser feita a próxima operação de leitura ou escrita na memória;

- ***Barramento de Endereços***: liga o REM à memória para transferência do endereço da célula a ser lida ou escrita. Sua largura em bits deve ser igual ao REM;

- ***Registrador de Dados da Memória (RDM):*** armazena os dados que estão sendo transferidos de/para a memória;

- ***Barramento de Dados:*** liga o RDM à memória, possuindo a mesma largura deste. É o caminho por onde é feita a transferência do conteúdo;

- ***Barramento de Controle:*** interliga a CPU à memória para enviar os comandos de *READ* e *WRITE* e receber *WAIT*.

Para que uma simples placa de vídeo ou um HD possa ser utilizado em qualquer computador, independentemente do processador instalado, utiliza-se diversos modelos de barramentos de expansão. Os barramentos de expansão são disponibilizados na placa-mãe dos micros através de *slots que* nada mais são do que encaixes para que as conexões entre placas presentes no sistema computacional utilizem determinados padrões de barramento. Na parte superior dos *slots*, encontram-se ranhuras para a conexão de placas de circuito que funcionam com a placa-mãe. Sem as ranhuras, os micros ficariam limitados aos circuitos que estivessem permanentemente montados na placa-mãe.

Vale ressaltar que junto com a evolução dos computadores, o desempenho dos barramentos também evolui. Novos barramentos e/ou melhorias nos atuais estão sempre surgindo.

1.4.2 Barramento de Expansão Internos

Os barramentos de expansão são meios de comunicação utilizados em computadores para a interconexão dos mais variados dispositivos. Nos computadores modernos cabe aos *slots* de expansão a conexão de diversos dispositivos como placas de vídeo, por exemplo. O processador pode enviar comandos neste barramento, e cabe a cada periférico, conectados ao um slot, interpretar o seu comando.

Durante toda a evolução da arquitetura de computadores, diversos tipos de barramentos foram utilizados e continuam a ser utilizados:

- ***Barramento ISA (Industry Standard Architecture)***: o barramento ISA é um padrão não mais utilizado, sendo encontrado apenas em computadores antigos. Seu aparecimento se deu na época do IBM PC e essa primeira versão trabalha com transferência de 8 bits por vez e clock de 8,33 MHz.

- ***Barramento PCI (PeripheralComponentInterconnect):*** os slots PCI são menores que os slots ISA, assim como os seus dispositivos, obviamente. Mas há outra característica que tornou o padrão PCI atraente: o recurso Bus Mastering. Em poucas palavras, trata-se de um sistema que permite a dispositivos que fazem uso do barramento ler e gravar dados direto na memória RAM, sem que o processador tenha que "parar" e interferir para tornar isso possível. Note que esse recurso não é exclusivo do barramento PCI. Outra característica marcante do PCI é

a sua compatibilidade com o recurso Plugand Play (PnP), algo como "*plugar e usar*". Com essa funcionalidade, o computador é capaz de reconhecer automaticamente os dispositivos que são conectados ao slot PCI.

- ***Barramento AGP:*** para lidar com o volume crescente de dados gerados pelos processadores gráficos foi criado o padrão AGP. A primeira versão do AGP (chamada de AGP 1.0) trabalha com barramento de 32 bits e tem clock de 66 MHz, o que equivale a uma taxa de transferência de dados de até 266 MB por segundo, mas, na verdade, pode chegar ao valor de 532 MB por segundo. Explica-se: o AGP 1.0 pode funcionar no modo 1x ou 2x. Com 1x, um dado por pulso de *clock* é transferido. Com 2x, são dois dados por pulso de *clock*. Algum tempo depois, surgiu o AGP 3.0, que conta com a capacidade de trabalhar com alimentação elétrica de 0,8 V e modo de operação de 8x, correspondendo a uma taxa de transferência de 2.133 MB por segundo. Além da alta taxa de transferência de dados, o padrão AGP também oferece outras vantagens. Uma delas é o fato de sempre poder operar em sua máxima capacidade, já que não há outro dispositivo no barramento que possa, de alguma forma, interferir na comunicação entre a placa de vídeo e o processador. O AGP também permite à placa de vídeo fazer uso de parte da memória RAM do computador como um incremento de sua própria memória, um recurso chamado *Direct Memory Execute*.

1.4.3 Barramento de Expansão Externos

Para permitir a comunicação do computador com os dispositivos externos, alguns padrões de barramento foram implementados. Os mais conhecidos são USB e Firewire.

- ***USB (Universal Serial Bus):*** é o barramento externo mais usado atualmente. Além de ser usado para a conexão de todo o tipo de dispositivos, ele fornece uma pequena quantidade de energia, permitindo que os conectores USB forneçam energia para o funcionamento desses mesmos dispositivos. No USB 1.x, as portas transmitem apenas 12 megabits, o que é pouco para HDs, pendrives, drives de CD, placas wireless e outros periféricos rápidos. Mas, no USB 2.0, a velocidade foi ampliada para 480 megabits (ou 60 MB/s), suficiente para a maioria dos pendrives e HDs externos. Atualmente o USB 3.0, padrão mais atual, conta com velocidade de até 5Gbit/s (625 MB/s). Existem quatro tipos de conectores USB: o USB tipo A, que é o mais comum, usado por pendrives e todo tipo de dispositivo conectado ao PC; o USB tipo B, que é o conector "quadrado" usado em impressoras e outros periféricos; além do USB mini 5P e o USB mini 4P, dois formatos menores que são utilizados por câmeras, *mp3 players*, *smartfones* e outros *gadgets*.

- ***FireWire:*** o *Firewire* surgiu em 1995 (pouco antes do USB) como um concorrente do barramento SCSI. Ini-

cialmente ele foi desenvolvido pela Apple e depois submetido ao IEEE, quando passou a se chamar IEEE 1394. Embora seja o mais popularmente usado, o nome "*Firewire*" é uma marca registrada pela Apple, por isso não se encontra referência a ele em produtos ou documentação de outros fabricantes. Outro nome comercial para o padrão é o "*i.Link*", usado pela Sony. O *Firewire* também é um barramento serial muito similar ao USB em vários aspectos. A versão inicial do *Firewire* já operava a 400 megabits (ou 50 MB/s), enquanto o USB 1.1 operava apenas 12 megabits. Apesar disso, o USB utilizava transmissores e circuitos mais baratos e era livre de pagamento de royalties, o que acabou fazendo com que ele se popularizasse rapidamente. Na época, a indústria procurava um barramento de baixo custo para substituir as portas seriais e paralelas e, como de praxe, acabou ganhando a solução mais barata. Assim como o USB, o Firewire é um barramento *plug-and-play* e suporta a conexão de vários periféricos na mesma por porta, utilizando uma topologia acíclica, em que um periférico é diretamente conectado ao outro e todos se enxergam mutuamente, sem necessidade do uso de hubs ou centralizadores. Pode-se, por exemplo, conectar um HD externo (com duas portas Firewire) ao PC e conectar uma filmadora ao HD e o PC enxergaria ambos.

Para controlar os demais dispositivos é utilizado o driver, que é o programa a partir do qual uma unidade periférica cria uma interface com o sistema operacional para se conectar com o dispositivo do *hardware*.

Vejamos um exemplo prático: quando se conecta uma impressora a um computador, esta impressora requer a instalação do "*driver*" (que é instalado a partir de um CD que vem junto com o equipamento), sem o qual ela não conseguirá fazer a interface com as aplicações em geral. O "*driver*" é o elemento que faz esse comando.

Foi a solução encontrada para que os Sistemas Operacionais sejam compatíveis com diferentes tipos de equipamentos. Cada impressora, por exemplo, tem suas peculiaridades de *hardware*, logo, torna-se inviável que o Sistema Operacional tenha conhecimento sobre todos os equipamentos disponíveis. O Sistema Operacional disponibiliza bibliotecas de programação para que o fabricante possa criar uma interface entre seu equipamento e o *software*.

1.5. Processador

O processador é o componente vital de um sistema de computação responsável pela realização das operações de processamento (cálculos matemáticos, entre outros) e de controle durante a execução de um programa.

Como ilustrado na figura 1.7, os processadores são compostos por vários blocos. Sendo basicamente dividido em dois grandes grupos: **Blocos Operacionais** (unidades de execução e banco de registradores) e **Blocos de Controle** (lógica de controle). As unidades de execução contêm o hardware que executa as instruções. Isto inclui o hardware que busca e decodifica as instruções, bem como as unidades lógico-aritméticas (ULAs) que executam os cálculos.

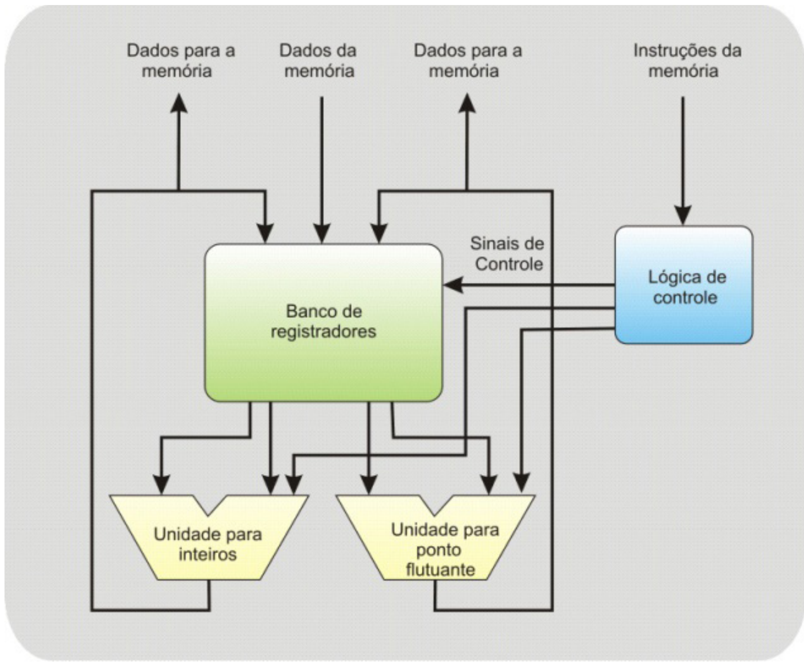


Figura 1.7: Diagrama em blocos de um processador.
Figura adaptada de [Carter, N.]

Muitos processadores possuem unidades de execução diferentes para cálculos com inteiros e com pontos flutuantes, porque é necessário um hardware muito diferente para estes dois tipos de dados. Além disso, como veremos na próxima Unidade, os processadores modernos, para melhorar o desempenho, frequentemente, utilizam várias unidades de execução para executar instruções em paralelo.

O banco de registradores é uma pequena área de armazenamento para os dados que o processador está usando. Os valores armazenados no banco de registradores podem ser acessados mais rapidamente que os dados armazenados no sistema de memória, sendo que os bancos de registradores geralmente suportam vários acessos simultâneos. Isto permite que uma operação, como por exemplo a adição, leia todas

suas entradas do banco de registradores ao mesmo tempo, ao invés de ter que lê-las uma por vez.

A lógica de controle controla o processador determinando quando as instruções podem ser executadas e quais operações são necessárias para executar cada instrução. Nos primeiros processadores, ela era uma parte muito pequena do hardware do processador, quando comparada com as ULAs e o banco de registradores, mas a quantidade de lógica de controle necessária cresceu significativamente à medida que os processadores tornaram-se mais complexos, fazendo com que seja uma das partes mais difíceis de ser projetada. Em resumo temos:

- ***Unidade Lógica e Aritmética ou ALU (AritmethicandLogic Unit):*** que assume todas as tarefas relacionadas às operações lógicas (ou, e, negação, etc.) e aritméticas (adições, subtrações, etc.) a serem realizadas no contexto de uma tarefa realizada através dos computadores; as primeiras ULAs eram de concepção bastante simples, realizavam um conjunto relativamente modesto de operação, com operandos de pequena dimensão (no que diz respeito ao tamanho da palavra). Com o passar do tempo, estes elementos foram tornando-se sofisticados para suportar operações mais complexas a maiores tamanhos de palavras de dados para permitir o grande potencial de cálculo oferecido pelos atuais microprocessadores;

- ***Registradores:*** que, como o nome indica, abriga o conjunto de registros dos microprocessadores essenciais para a realização das instruções dos programas do computador;

de forma mais superficial, pode-se subdividir o conjunto de registros de um microprocessador em dois grupos: os registros de uso geral, utilizados para armazenamento de operandos ou resultados de operações executadas pelo microprocessador, e os registros de controle, utilizados como suporte à execução dos programas do computador;

- ***Unidade de Controle:*** a responsável por tarefas como a interpretação das instruções de máquina a ser executada pelo computador, a sincronização destas instruções, o atendimento a eventos de hardware, etc. Esta unidade assume toda a tarefa de controle das ações a serem realizadas pelo computador, comandando todos os demais componentes de sua arquitetura, garantindo a correta execução dos programas e a utilização dos dados corretos nas operações que as manipulam. É a unidade de controle quem gerencia todos os eventos associados à operação do computador, particularmente as chamadas **interrupções**, tão utilizadas nos sistemas há muito tempo.

Execução de um Programa

Como comentado anteriormente, um programa para ser efetivamente executado por um processador deve ser constituído de uma série de instruções (em linguagem de máquina). Estas instruções devem estar armazenadas em posições sucessivas da memória principal. A execução é sequencial, ou seja, se a instrução executada está na posição x, a próxima instrução a ser executada deverá estar na posição x+1. A sequência de funcionamento de uma CPU é conhecida como

ciclo **Busca – Decodificação – Execução de Instruções**. A figura 1.8 ilustra este ciclo.

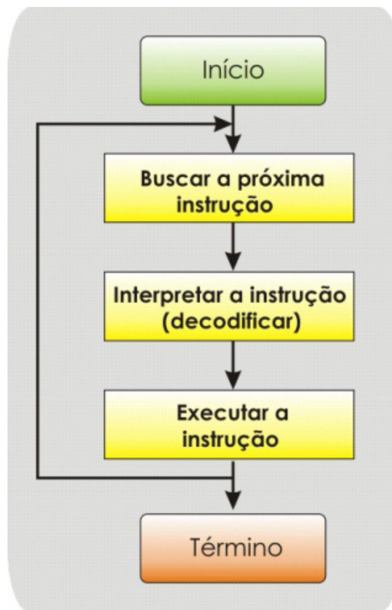


Figura 1.8: Ciclo básico de instrução.
Figura adaptada de [Rebonatto, M.]

- Um elemento dentro do processador, denominado contador de instruções, (*ProgramCounter-PC*) contém a posição da próxima instrução a ser executada. Quando uma sequência de execução de instruções tem início, a instrução cujo endereço está no contador de instruções é trazida da memória para uma área chamada registrador de instruções (RI). Este processo é conhecido como **busca da instrução**.

- A instrução é interpretada por circuitos de decodificação que fazem com que sinais eletrônicos sejam gerados no processador como resultado do valor do campo de operação,

isto é, **decodificam** a informação correspondente à operação a ser realizada.

- Esses sinais resultam na **execução da instrução**, isto é, aplicação da função contida pela operação sobre os operandos. Quando a execução de uma instrução é terminada, o contador de instruções é atualizado para o endereço da memória da próxima instrução (x + 1).

A sequência de instruções pode mudar com o resultado de uma instrução que direciona um desvio (também chamado de salto, *jump*). Instruções deste tipo contêm no campo operandos o endereço da próxima instrução a ser executada. Elas causam mudanças no fluxo do programa como resultado das condições dos dados. O desvio condicional representado por uma instrução de alto nível IF traduz-se em algum tipo de instrução de desvio.

As atividades realizadas pela CPU podem ser divididas em duas grandes categorias funcionais: **funções de processamento** e **funções de controle**. A função de processamento se encarrega de realizar as atividades relacionadas com a efetiva execução de uma operação, ou seja, processar (executar a instrução) de instruções. O principal componente da CPU que realiza a função de processamento é a ULA (unidade lógica e aritmética), sendo que ação dela é complementada pelo uso de registradores de processamento. A função de controle é exercida pelos componentes da CPU que se encarregam de atividades de busca, interpretação e controle da execução das instruções, bem como do controle da ação dos demais componentes do sistema de computação (memória, entrada/saída). O principal componente da CPU responsável pela função de controle é a UC (unidade de controle).

1.6. Memória

Uma das partes mais importante do computador é a memória. O processador apenas recebe dados e os processa segundo alguma pré-programação, e logo após os devolve, não importando de onde vem e para onde vão. Os programas a serem executados e os dados a serem processados (inclusive os que já o foram) ficam na memória, visto que a área para armazenamento de dados do processador é pequena.

Todo computador é dotado de uma quantidade (que pode variar de máquina para máquina) de memória a qual se constitui de um conjunto de circuitos capazes de armazenar (por períodos mais curtos ou mais longos de tempo) as unidades de dados e os programas a serem executados pela máquina. Nos computadores de uso geral, é possível encontrar diferentes denominações para as diferentes categorias de memória que neles são encontradas:

- ***Memória principal:*** também designada memória de trabalho, é onde normalmente devem estar armazenados os programas e dados a serem manipulados pelo processador;

- ***Memória secundária:*** que permitem armazenar uma maior quantidade de dados e instruções por um período de tempo mais longo; os discos rígidos são exemplos mais imediatos de memória secundária de um computador, mas podem ser citados outros dispositivos menos recentes como as unidades de fita magnética e os cartões perfurados introduzidos por Hollerith;

- ***Memória cache:*** conceito introduzido mais recentemente e que se constitui de uma pequena porção de memória com curto tempo de resposta, normalmente integrada aos processadores e que permite incrementar o desempenho na execução de um programa.

Há basicamente dois tipos de memória:

- ***ROM (Read-Only Memory):*** só permite a leitura de dados e são lentas; em compensação não perdem seu conteúdo quando desligadas;

- ***RAM (Random Access Memory)***: são rápidas, permitem leitura e escrita, porém, seu conteúdo é perdido quando são desligadas.

Em geral a ROM é utilizada para manter um programa que é executado pelo computador cada vez que ele é ligado ou reiniciado. Este programa é chamado de *bootstramp* e instrui o computador a carregar o SO. Dentro da ROM existem basicamente 3 programas: BIOS, POST e SETUP. É comum acontecer confusão em relação aos nomes, sendo usado atualmente o termo BIOS como algo genérico. Para acessar o programa de configuração, basta acessar um conjunto de teclas durante o POST (geralmente na contagem da memória). Na maioria das máquinas, basta apertar a tecla “DEL” ou “Delete”, porém, esse procedimento pode variar de acordo com o fabricante. Quando o computador é ligado, o POST (*Power On Self Test*) entra em ação, identificando a configuração instalada, inicializando os circuitos da placa-mãe (*chipset*) e vídeo, e executando teste da memória e

teclado. Após, ele carrega o SO de algum disco para a memória RAM, entregando o controle da máquina para o SO.

Na RAM, ficam armazenados o SO, programas e dados que estejam em processamento. O processador acessa a RAM praticamente o tempo todo. Atualmente a memória RAM, formada por circuitos de memória dinâmica (**DRAM** – *Dynamic* RAM), é mais lenta que o processador, ocasionando *waitstates* até que a memória possa entregar ou receber dados, diminuindo, assim, o desempenho do micro. Memórias mais rápidas amenizam este problema, assim como a utilização de cache de memória.

A **cache** é uma memória estática (**SRAM** - *Static*RAM) de alto desempenho utilizada para intermediar a comunicação com o processador. Na maioria das vezes, o processador não acessa o conteúdo da RAM, mas sim uma cópia que fica na cache. A cache é utilizada desde o 386DX e a partir dos 486, todos os processadores passaram a conter uma quantidade de memória estática, conhecida como L1 ou interna. A cache fora do processador é conhecida como L2 ou externa. Hoje, existem processadores com mais níveis de cache. Uma ressalva é que os processadores a partir do Pentium II possuem a L2 dentro da caixa que envolve o processador, não fazendo mais sentido as denominações interna e externa.

A DRAM é formada por capacitores fáceis de construir, baratos e podem-se aglomerar muitas células de memória em pequeno espaço físico. O problema é que, após algum tempo, eles descarregam, dessa forma deverá haver um período de recarga, chamado *refresh*. Durante este período, a memória geralmente não pode ser acessada, limitando, assim, com uma imposição física sua velocidade. Por outro lado, a SRAM é formada por caros circuitos digitais chamados *flip-flops*, que

armazenam dados sem a necessidade de ciclos para *refresh*. Um *flip-flop*, por ser um circuito completo, é maior que um capacitor, consequentemente, onde cabem muitos capacitores têm-se somente alguns *flip-flops*. Devido ao preço, tamanho e consumo, não é possível que um micro tenha toda sua RAM de memória estática, então a partir dos 386, utiliza-se um modelo híbrido com SRAM como cache e DRAM como RAM propriamente dita.

Operações no Sistema de Memória

Um modelo de sistema de memória simplificado pode ser dado por suportar somente duas operações: carga e armazenamento. As operações de armazenamento ocupam dois operandos: um valor a ser armazenado e o endereço onde o valor deve ser armazenado. Eles colocam o valor especificado na posição de memória especificada pelo endereço. Operação de carga tem um operando que especifica um endereço e retorna o conteúdo dessa posição de memória para o seu destino.

Utilizando este modelo pode-se imaginar a memória funcionando como uma grande folha de papel pautado, a qual cada linha na página representa um local de armazenamento para um byte. Para escrever (armazenar) na memória conta-se de cima para baixo na página até que se atinja a linha especificada pelo endereço, apaga-se o valor escrito naquela linha e escreve-se o novo valor. Para ler (carregar) um valor, conta-se de cima para baixo até que se atinja a linha especificada pelo endereço e lê-se o valor escrito naquele endereço. A maioria dos computadores permite que mais de um byte de memória seja armazenado ou carregado por vez. Geralmente, uma operação de carga ou de armazenamento opera sobre uma quantidade de dados igual à lar-

gura de bits do sistema, e o endereço enviado ao sistema de memória especifica a posição do byte de dados de endereço mais baixo a ser carregado ou armazenado. Por exemplo, um sistema de 32 bits carrega ou armazena 32bits (4 bytes) de dados em cada operação, nos 4 bytes começam o endereço da operação, de modo que uma carga, a partir da localização 424, retornaria uma quantidade de 32 bits contendo os bytes das localizações 424, 425, 426 e 427.

Para simplificar o projeto dos sistemas de memória, alguns computadores exigem que as cargas e armazenamentos sejam alinhados, significando que o endereço de uma referência de memória precisa ser múltiplo do tamanho do dado que está sendo carregado ou armazenado, de modo que uma carga de 4 bytes precisa ter um endereço que seja um múltiplo de 4, um armazenamento de 8 bytes precisa ter um endereço que seja múltiplo de 8, e assim por diante. Outros sistemas permitem cargas e armazenamentos desalinhados, mas demora mais tempo para completar tais operações do que com cargas alinhadas.

Uma questão adicional com carga e armazenamento de vários bytes é a ordem na qual eles são escritos na memória. Há dois tipos de ordenação diferentes que são utilizados em computadores modernos: *littleendian* e *big endian*. No sistema *littleendian* o byte menos significativo (o valor menor) de uma palavra é escrito no byte de endereço mais baixo, e os bytes são escritos na ordem crescente de significância. No sistema big endian, a ordem é inversa, o byte mais significativo é escrito no byte de memória com o endereço mais baixo. As linhas abaixo mostram o exemplo de como os sistemas *littleendian* e *big endian* escreveriam uma palavra de 32 bits (4 bytes) no endereço 0x1000.

```
Palavra = 0x90abcdef
Endereço = 0x1000
                  0x1000   0x1001   0x1002   0x1003
Little endian     ef       cd        ab        90
Big endian        90       ab        cd        ef
```

Exemplo retirado de [Carter, N.]

O projeto de sistemas de memória tem um impacto enorme sobre o desempenho de sistemas de computadores e é frequentemente o fator limitante para a rapidez de execução de uma aplicação. Tanto a largura de banda (quantos dados podem ser carregados ou armazenados em um período de tempo) quanto a latência (quanto tempo uma operação de memória em especial demora para ser completada) são importantes para o desempenho da aplicação.

Outras questões importantes no projeto de sistemas de memória incluem a proteção (evitar que diferentes programas acessem dados uns dos outros) e o modo como o sistema de memória interage com o sistema de E/S.

Existem várias tecnologias de construção de memória RAM: FPM (*Fast Page Mode*), EDO (*Extended Data Output*), BEDO (*Burst Extended Data Output*), SDRAM (*Synchronous Dynamic* RAM), PC-100, PC-133, DDR-SDRAM (*Double Data Rate* SDRAM), ES-DRAM (*Enhanced* SDRAM), RDRAM (*Rambus* DRAM), SLDRAM (*Sync Link* DRAM), etc.

Além da tecnologia também existem vários tipos de módulos de memória. Como exemplo temos: SIPP (*Single in Line Pin Package*), SIMM30-(*Single in Line Memory Module*), SIMM-72 (*Single in Line Memory Module*), DIMM (*Double in Line Memory Module*), RIMM (*Rambus In Line Memory Module*), etc.

1.7. Exercícios

1. Quais são os três componentes básicos de um sistema de computador? Faça um diagrama de blocos ilustrando os componentes básicos e comente sobre as funcionalidades de cada um deles.

2. Qual a diferença entre linguagem de máquina e linguagem de montagem? Por que a linguagem de montagem é considerada mais fácil para seres humanos programarem do que a linguagem de máquina?

3. Explique o processo de desenvolvimento e execução de um programa em linguagem de alto nível.

4. Explique as vantagens de utilizar um compilador em vez de um montador e diferencie compilador de interpretador.

6. Explique e mostre na forma de esquema gráfico o funcionamento básico dos sistemas de computador.

7. Para que servem barramentos nos computadores? Explique mostrando os dois principais tipos.

8. Qual a função dos barramentos de dados, de endereço e de controle?

9 - Quais os componentes básicos de um processador? Explique a função de cada uma.

10 - Explique o ciclo básico de execução de um programa.

11 - Qual a diferença entre memória RAM e ROM?

12 - Explique a diferença entre as ordenações *big endian* e *littleendian*.

2. PROGRAMAÇÃO EM LINGUAGEM DE MÁQUINA

2.1 Linguagem Assembly

A linguagem compreendida pelos computadores é a linguagem de máquina cujo alfabeto é formado apenas por duas letras: os dígitos binários[1] (ou bits) 0 e 1. Escrever programas em linguagem de máquina utilizando apenas 0s e 1s, entretanto, é uma tarefa chata e bastante sujeita a erros.

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

Cada instrução em linguagem Assembly corresponde a uma instrução de linguagem máquina, mas em vez de ser especificada em termos de zeros e uns, é especificada utilizando mnemónicas e nomes simbólicos. Por exemplo, a instrução que soma dois números guardados nos registos R0 e R1 e colocar o resultado em R0 poderá ser codificada como ADD R0, R1.

Tipicamente, quando um programador utiliza Assembly é porque a velocidade ou a dimensão do programa que está a desenvolver são críticas. Isto acontece muitas vezes na vida real, sobretudo quando os computadores são embebidos noutras máquinas (e.g. carros, aviões,

1 O uso de números binários para representar instruções e dados é a base da teoria computacional.

unidades de controlo de produção industrial...). Computadores deste tipo devem responder rapidamente a eventos vindos do exterior. As linguagens de alto nível introduzem incerteza quanto ao custo de execução temporal das operações, ao contrário do Assembly, onde existe um controlo apertado sobre que instruções são executadas.

Além deste motivo, existe outro que também está relacionado com a execução temporal dos programas: muitas vezes é possível retirar grandes benefícios da optimização de programas. Por exemplo, alguns jogos que recorrem a elaborados motores 3D são parcialmente programados em Assembly (nas zonas de código onde a optimização é mais benéfica que são normalmente as zonas de código mais frequentemente utilizado).

2.2 Estrutura dos programas em Assembly

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

- **Comentários:** estes são especialmente importantes quando se trabalha com linguagens de baixo nível, pois ajudam ao desenvolvimento dos programas e são utilizados exaustivamente. Os comentários começam com o caracter "#" ou ";" dependendo da versão do assembly ou processador;

- **Identificadores:** definem-se como sendo sequências

de caracteres alfanuméricos, underscores (_) ou pontos (.) que não começam por um número. Os códigos de operações são palavras reservadas da linguagem e não podem ser usadas como identificadores (e.g. addu;

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

2.3 Instruções

Instruções que o Assembly interpreta e traduz em uma ou mais micro-instruções (em linguagem máquina). Elas indicam as ordens que devem ser executadas pela CPU e são transcrições (ou notações) simplificadas que correspondem aos códigos binários das instruções de máquina.

Para os montadores MASM (Macro ASseMbler) e TASM (Turbo ASseMbler), as instruções devem ser escritas, obrigatoriamente, uma por linha, podendo ter até quatro campos, delimitados de acordo com a seguinte ordem:

[label:] mneumônico [operando(s)]   [; comentário]

Onde label é o rótulo dado ao endereço da instrução (no segmento de código), mneumônico representa a instrução, operandos são os dados operados pela instrução, e comentário é qualquer texto escrito para elucidar ao leitor do programa o procedimento ou objetivo

da instrução. Destes, apenas o campo mneumônico é sempre obrigatório. O campo operandos depende da instrução incluída na linha e os campos label e comentário são sempre opcionais. Todos os valores numéricos estão, por padrão, em base decimal, a menos que outra base seja especificada.

2.4 Diretivas

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

As instruções e diretivas de um programa Assembly podem ser escritas em maiúsculas ou minúsculas (Assembly é uma linguagem não case-sensitive, entretanto, as seguintes sugestões são dadas de forma a tornar o programa mais legível:

- Palavras reservadas (instruções e diretivas) devem ser escritas em letras maiúsculas; e

- Nomes em geral (comentários e variáveis) podem ser utilizados letras minúsculas ou maiúsculas, dando-se preferência às minúsculas.

O compilador implementa uma forma simplificada de definição de segmentos, a qual pode ser usada na maioria dos *programas.exe*. Muitos defaults assumidos são os mesmos usados pelos compilado-

res das linguagens de alto nível, facilitando a criação de módulos que serão ligados a programas criados em outras linguagens. Para que o montador assuma uma dada estrutura, é suficiente a definição de um modelo.

2.5 Processo de criação de programas

Para a criação de programas são necessários os seguintes passos:

- Desenvolvimento do algoritmo, estágio em que o problema a ser solucionado é estabelecido, e a melhor solução é proposta, criação de diagramas esquemáticos relativos … melhor solução proposta;

- Codificação do algoritmo, o que consiste em escrever o programa em alguma linguagem de programação; linguagem Assembly, neste caso específico, tomando como base a solução proposta no passo anterior;

- A transformação para a linguagem de máquina, ou seja, a criação do programa objeto, escrito como uma sequência de zeros e uns que podem ser interpretados pelo processador.

- O último estágio é a eliminação de erros detectados no programa na fase de teste. A correção normalmente requer a repetição de todos os passos, com observação atenta.

2.5.1 Registradores da CPU.

Para o propósito didático, vamos focar registradores de 16 bits. A CPU possui 4 registradores internos, cada um de 16 bits. São eles AX, BX, CX e DX. São registradores de uso geral e também podem ser usados como registradores de 8 bits. Para tanto, devemos referenciá-los como, por exemplo, AH e AL, que são, respectivamente, o byte high e o low do registrador AX. Esta nomenclatura também se aplica para aos registradores BX, CX e DX.

Os registradores, segundo seus respectivos nomes:

- AX Registrador Acumulador;
- BX Registrador Base;
- CX Registrador Contador;
- DX Registrador de Dados;
- DS Registrador de Segmento de Dados;
- ES Registrador de Segmento Extra;
- SS Registrador de Segmento de Pilha;
- CS Registrador de Segmento de Código;
- BP Registrador Apontador da Base;
- SI Registrador de Índice Fonte;
- DI Registrador de Índice Destino;
- SP Registrador Apontador de Pilha;
- IP Registrador Apontador da Próxima Instrução;
- F Registrador de Flag.

2.5.2 Programa Debug

Para a criação de um programa em assembler existem 2 opções: usar o TASM - Turbo Assembler da Borland ou o DEBUGGER. Nesta primeira seção vamos usar o DEBUGGER, uma vez que podemos encontrá-lo em qualquer PC.

DEBUGGER pode apenas criar arquivos com a extensão .COM, e por causa das características deste tipo de programa, eles não podem exceder os 64 Kb, e também devem iniciar no endereço de memória 0100H dentro do segmento específico. É importante observar isso, pois deste modo os programas .COM não são realocáveis. Os principais comandos do programa DEBUGGER são:

- A - Montar instruções simbólicas em código de máquina;
- D - Mostrar o conteúdo de uma área da memória;
- E - Entrar dados na memória, iniciando num endereço específico;
- G - Rodar um programa executável na memória;
- N - Dar nome a um programa;
- P - Proceder, ou executar um conjunto de instruções relacionadas;
- Q - Sair do programa debug;
- R - Mostrar o conteúdo de um ou mais registradores;
- T - Executar passo a passo as instruções;
- U - Desmontar o código de máquina em instruções simbólicas;
- W - Gravar um programa em disco.

É possível visualizar os valores dos registradores internos da CPU usando o programa Debug que é um programa que faz parte do pacote do DOS até o Windows XP. As versões mais recentes não vêm com este aplicativo. Contudo ele pode ser encontrado no endereço http://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/DOS-Debug.shtml. Para iniciá-lo, basta digitar Debug na linha de comando:

```
C:/>Debug [Enter]
-
```

Nota-se a presença de um hífen no canto inferior esquerdo da tela. Este é o prompt do programa. Para visualizar o conteúdo dos registradores, experimente:

```
-r[Enter]

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE
BP=0000  SI=0000  DI=0000
DS=0D62  ES=0D62  SS=0D62  CS=0D62  IP=0100   NV
UP EI PL NZ NA PO NC
0D62:0100 2E            CS:
0D62:0101 803ED3DF00    CMP     BYTE PTR
[DFD3],00                 CS:DFD3=03
```

É mostrado o conteúdo de todos os registradores internos da CPU; um modo alternativo para visualizar um único registrador é usar o comando "r" seguido do parâmetro que faz referência ao nome do registrador:

```
-rbx
BX 0000
:
```

Esta instrução mostra o conteúdo do registrador BX e muda o indicador do Debug de "-" para ":". Quando o prompt assim se tornar, significa que é possível, embora não obrigatória, a mudança do valor contido no registrador, bastando digitar o novo valor e pressionar [Enter]. Caso se pressione [Enter] o valor antigo se mantém.

2.5.3 Estrutura Assembly

Nas linhas do código em Linguagem Assembly há duas partes: a primeira é o nome da instrução a ser executada; a segunda, os parâmetros do comando. Por exemplo:

```
add ah bh
```

Aqui "add" é o comando a ser executado, neste caso uma adição, e "ah" bem como "bh" são os parâmetros. Por exemplo:

```
mov al, 25
```

No exemplo acima, estamos usando a instrução mov que significa mover o valor 25 para o registrador "al". O nome das instruções nesta linguagem é constituído de 2, 3 ou 4 letras. Estas instruções são chamadas mnemônicos ou códigos de operação, representando a função que o processador executar. Em algumas situações as instruções aparecem assim:

```
add al,[170]
```

Os colchetes no segundo parâmetro indicam-nos que vamos tra-

balhar com o conteúdo da célula de memória de número 170, ou seja, com o valor contido no endereço 170 da memória e não com o valor 170. Isso é conhecido como “endereçamento direto”.

2.5.4 Modos de Endereçamento

Um operando pode estar em diversas localizações na arquitetura tais como: na memória, no registrador, em um ponteiro, entre outras. Para cada forma de se especificar o operando na instrução Assembly, existe um modo de endereçamento. No chamado *endereçamento direto*, o endereço do operando na memória é especificado na instrução:

```
add 8088h, 8181h
```

No caso acima o valor contido no endereço 8088h é somado ao valor contido no endereço 8181h e alocado no endereço 8088h.

Já no *endereçamento de registradores* apenas registradores são referenciados nas instruções:

```
mov r1, r2
```

Neste caso o valor contido no registrador r2 é movido para o registrador r1. Um problema encontrado no endereçamento direto é que o comprimento do campo de endereço é geralmente inferior ao comprimento da palavra, limitando assim a gama de endereços. Uma solução é ter o campo de endereço referenciando o endereço de uma palavra na memória, a qual, por sua vez, contém o endereço completo do operando. Isso é conhecido como *endereçamento indireto*:

```
add r1,(r3)
```

Na instrução acima, r3 aponta para o endereço do operando real. Na verdade, esse modo implementa os ponteiros. Finalmente, temos o endereçamento indexado que é usado para operar vetores e matrizes. A instrução contém o endereço base do *array* e o deslocamento para mudar de célula, como por exemplo, a instrução abaixo:

```
add r1, [r2]end
```

O onde r1 deverá armazenar o somatório dos valores armazenados na matriz.

2.5.5 Manipulação de Pilhas

Chamamos área de pilha um espaço de memória especialmente reservado para organização de uma pilha de dados. Esta pilha é usada como memória auxiliar durante a execução de uma aplicação. As operações sobre esta área são push (empilha) e pop (desempilha).

Em geral, o hardware dá algum suporte à manutenção dessa pilha. No caso da máquina x86, um registrador específico (*esp)* é dedicado ao endereço do topo da pilha. Por motivos históricos, a pilha cresce em direção aos endereços mais baixos de memória. Ou seja, para alocar espaço para um endereço, devemos subtrair 4 de *esp* (supondo que um endereço ocupa 4 bytes) e para desalocar devemos somar 4 a *esp*.

Ao chamarmos um procedimento, precisamos passar dados e controle de uma parte do código para outra. A maior parte das arquiteturas tem instruções que facilitam algumas dessas tarefas, e o restante delas tem que ser programado explicitamente, usando a pilha. Geral-

mente as intruções *call* e *ret* facilitam a transferência de controle.

A instrução *call f* transfere o controle para f, antes, armazenando o endereço de retorno (endereço da instrução seguinte a *call f*), para que o controle possa retornar para este endereço ao final da execução da função. A instrução *ret* faz com que o controle retorne para o último endereço de retorno armazenado.

- ***call f*** armazena o endereço da próxima instrução na pilha de execução e faz um desvio para o endereço associado a f.

- ***ret*** desempilha um endereço da pilha e desvia para o endereço desempilhado.

Outro uso da pilha é salvar registradores durante chamadas de funções. Cada função utiliza os registradores para armazenar valores de variáveis e temporários. Mas ao fazer uma chamada, esses mesmos registradores são usados pela nova função. Para evitar que os valores anteriores se percam, podemos armazená-los na pilha. Deve existir uma convenção em cada sistema, de maneira que não se armazenem valores duas vezes e nem se deixe de armazená-los.

2.5.6 Criando um programa simples em Assembly.

Vamos, então, criar um cronograma para ilustrar o que vimos até agora. Adicionaremos dois valores. Para montar um programa no Debug, é usado o comando “a” (Assembler); quando usamos este comando, podemos especificar um endereço inicial para o nosso progra-

ma como o parâmetro, mas é opcional. No caso de omissão, o endereço inicial é o especificado pelos registradores CS:IP, geralmente no endereço 0100h, que é o local o qual programas com extensão .COM devem iniciar. E será este o local que usaremos, uma vez que o Debug só pode criar este tipo de programa.

Embora neste momento não seja necessário darmos um parâmetro ao comando "a", isso é recomendável para evitar problemas, logo:

```
a 100[enter]
mov ax,0002[enter]
mov bx,0004[enter]
addax,bx[enter]
nop[enter][enter]
```

O que o programa faz é mover o valor 0002 para o registrador ax, move o valor 0004 para o registrador bx, adiciona o conteúdo dos registradores ax e bx, guardando o resultado em ax e, finalmente, a instrução nop (nenhuma operação) finaliza o programa. No programa debug, a tela se parece com:

```
C:\>debug
-a 100
0D62:0100 mov ax,0002
0D62:0103 mov bx,0004
0D62:0106 add ax,bx
0D62:0108 nop
0D62:0109
```

Entramos com o comando "t" para executar passo a passo as instruções:

```
-t

AX=0002  BX=0000  CX=0000  DX=0000  SP=FFEE
BP=0000  SI=0000  DI=0000
DS=0D62  ES=0D62  SS=0D62  CS=0D62  IP=0103   NV
UP EI PL NZ NA PO NC
0D62:0103 BB0400        MOV     BX,0004
```

Vemos o valor 0002 no registrador AX. Teclamos "t" para executar a segunda instrução:

```
-t

AX=0002   BX=0004   CX=0000   DX=0000   SP=FFEE
BP=0000  SI=0000  DI=0000
DS=0D62   ES=0D62   SS=0D62   CS=0D62   IP=0106
NV UP EI PL NZ NA PO NC
0D62:0106 01D8          ADD     AX,BX
```

Teclando "t" novamente para ver o resultado da instrução add:

```
-t

AX=0006   BX=0004   CX=0000   DX=0000   SP=FFEE
BP=0000  SI=0000  DI=0000
DS=0D62   ES=0D62   SS=0D62   CS=0D62   IP=0108
NV UP EI PL NZ NA PE NC
0D62:0108 90            NOP
```

A possibilidade dos registradores conterem valores diferentes existe, mas AX e BX devem conter os mesmos valores acima descritos. Para sair do Debug usamos o comando "q" (quit).

2.5.7 Armazenando e carregando os programas.

Não seria prático ter que digitar o programa cada vez que iniciássemos o Debug. Ao invés disso, podemos armazená-lo no disco. Só que o mais interessante nisso é que um simples comando de salvar cria um arquivo com a extensão .COM, ou seja, executável - sem precisarmos efetuar os processos de montagem e ligação, como veremos posteriormente com o TASM.Eis os passos para salvar um programa que já esteja na memória:

- Obter o tamanho do programa subtraindo o endereço final do endereço inicial;
- Dar um nome ao programa;
- Colocar o tamanho do programa no registrador CX;
- Mandar o debug gravar o programa em disco.

Usando como exemplo o seguinte programa, vamos clarear a ideia de como realizar os passos acima descritos:

```
0C1B:0100 mov ax,0002
0C1B:0103 mov bx,0004
0C1B:0106 add ax,bx
0C1B:0108 int 20
0C1B:010A
```

Para obter o tamanho de um programa, o comando "h" é usado, já que ele nos mostra a adição e subtração de dois números em hexadecimal. Para obter o tamanho do programa em questão, damos como parâmetro o valor do endereço final do nosso programa (10A), e o endereço inicial (100). O primeiro resultado mostra-nos a soma dos endereços, o segundo, a subtração.

```
-h 10a 100
020a 000a
```

O comando “n” permite-nos nomear o programa.

```
-n test.com
```

O comando “rcx” permite-nos mudar o conteúdo do registrador CX para o valor obtido como tamanho do arquivo com o comando “h”, neste caso 000a.

```
-rcx
CX 0000
:000a
```

Finalmente, o comando “w” grava nosso programa no disco, indicando quantos bytes gravou.

```
-w

Writing 000A bytes
```

Para salvar um arquivo quando carregá-lo, é necessário dar o nome do arquivo a ser carregado e carregá-lo usando o comando “l” (load). Para obter o resultado correto destes passos, é necessário que o programa acima esteja criado. Dentro do Debug, escrevemos o seguinte:

```
-n test.com
-l
-u 100 109
0C3D:0100 B80200 MOV AX,0002
0C3D:0103 BB0400 MOV BX,0004
```

```
0C3D:0106 01D8 ADD AX,BX
0C3D:0108 CD20 INT 20
```

O último comando “u” é usado para verificar que o programa foi carregado na memória. O que ele faz é desmontar o código e mostrá-lo em assembly. Os parâmetros indicam ao Debug os endereços inicial e final a serem desmontados. O Debug sempre carrega os programas na memória no endereço 100h, conforme já comentamos.

2.6 EXERCÍCIOS

1. A linguagem Assembly (ou assembler) é conhecida como linguagem de montagem. Por quê?
2. Qual a vantagem de se usar Assembly quando comparada a linguagens de alto nível?
3. Explique porquê existem diferentes comandos e instruções.
4. Qual a diferença entre identificadores e Etiquetas na linguagem Assembly? Quando se deve usar um ou a outra?
5. O que são Diretivas em um programa Assembly?
6. O que são registradores e como eles são manipulados na linguagem Assembly?
7. Faça um programa em Assembly que execute o seguinte conjunto de instruções usando para isto apenas registros temporários:

   f=6;

   g=8;

   t=f+g;
8. Execute o programa da questão anterior passo-a-passo e veirifique os valores dos registros.

WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI
http://www.ufpi.br/uapi

Universidade Aberta do Brasil- UAB
http://www.uab.gov.br

Secretaria de Educação a Distância do MEC – SEED
http://www.seed.mec.gov.br

Associação Brasileira de Educação a Distância – ABED
http://www.abed.org.br

Apostilas, Tutoriais e Documentos
http://gabriel.sg.urcamp.tche.br/beraldo/arquitetura_2.htm

Guia do Hardware
http://www.guiadohardware.net

Laércio Vasconcelos
http://www.laercio.com.br

Gabriel Torres
http://www.gabrieltorres.com.br

César Augusto M. Marcon
http://www.inf.pucrs.br/~marcon/

Ivan L. M. Ricarte
http://www.dca.fee.unicamp.br/~ricarte/

Marcelo Trindade Rebonatto
http://vitoria.upf.tche.br/~rebonatto

Fabian Vargas
http://www.ee.pucrs.br/~vargas

Eduardo Moresi
http://www.intelingencia.blogspot.com

REFERÊNCIAS

CARTER, N. **Arquitetura de computadores**. Porto Alegre. Bookman, 2003.

HEURING, V. P; MURDOCCA, M. J.**Introdução à arquitetura de computadores**. Rio de Janeiro: Campus, 2002.

MORIMOTO, C. E. **Hardware: o guia definitivo**. Porto Alegre: Sulina, 2007.

MONTEIRO, M. A. **Introdução aorganização de computadores**. Rio de Janeiro: LTC, 2007.

PARHAMI, B. **Arquitetura de computadores:** de microprocessadores a supercomputadores. São Paulo:McGraw- Hill do Brasil, 2008.

PATTERSON, D. A ; HENNESSY, J. L. **Arquitetura de computadores: uma abordagem quantitativa**. Rio de Janeiro: Campus, 2003.

PATTERSON, D. A ; HENNESSY, J. L. **Organização e projeto de computadores**. Rio de Janeiro: Campus, 2005.

RIBEIRO, C ; DELGADO, J. **Arquitetura de computadores**.Rio de Janeiro: LTC, 2009.

STALLINGS, W. **Arquitetura e organização de computadores**. São Paulo: Prentice Hall, 2008.

TANENBAUM, A. S. **Organização estruturada de computadores**. Rio de Janeiro:Prentice Hall, 2007.

TORRES, G. **Hardware:** curso completo. Rio de Janeiro: Axcel Books, 2001.

WEBER, R. F. **Fundamentos de arquitetura de computadores**. Porto Alegre: Bookman, 2008.

UNIDADE II
PROCESSADORES

Resumo

Uma característica extremamente fascinante nos processadores é a sua capacidade de interpretar e executar instruções de programa. Na arquitetura de qualquer sistema computacional, o processador é o elemento que define a potencialidade da máquina, a partir de sua velocidade de processamento, do tamanho de palavra manipulada, da quantidade de memória interna (registradores) e do seu conjunto de instruções.

A maior parte destes fatores está intimamente ligada à forma como os diferentes componentes do microprocessador estão associados e como estes são controlados internamente.

***O conteúdo desta Unidade é influenciado fortemente pelos textos de Nicholas Carter, Eduardo Moresi, Marcelo Rebonatto, Ivan Ricarte e Fabian Vargas**. O Capítulo é acompanhado de exercícios sem a solução, preferimos deixar o prazer desta tarefa ao leitor. A bibliografia e a webliografia ao fim das notas são mais do que suficiente para adquirir um conhecimento razoável sobre a organização de computa-*

dores. ***Ao término da leitura desta Unidade, o estudante deverá: a) Entender a arquitetura de processadores, b) Compreender o funcionamento dos pipelines, e c) Ser capaz de diferenciar máquinas RISC de máquinas CISC.***

3. PROJETO DE PROCESSADORES

3.1 Introdução

A primeira característica a considerar num computador é sua Unidade Central de Processamento - UCP, que poderá fornecer uma série de indicações sobre o equipamento. A UCP ou CPU *(Central Processing Unit)*, também pode ser chamada de processador ou micro-processador, os dois termos são equivalentes.

Tudo o que acontece num computador provém da CPU que gerencia todos os recursos disponíveis no sistema. Seu funcionamento é coordenado pelos programas, que indicam o que deve ser feito e quando. Basicamente, a CPU executa cálculos muito simples como somas e comparações entre números, mas com uma característica muito especial: uma velocidade extremamente elevada.

A função das CPUs é sempre a mesma. O que as diferenciam é sua estrutura interna e, o mais importante, o fato de cada uma ter seu conjunto de instruções próprio. Ou seja, um programa escrito para uma CPU dificilmente poderá ser executado diretamente em outra - esse é um dos principais motivos da incompatibilidade entre os computadores.

Os microprocessadores são geralmente circuitos integrados dispostos em pastilhas os quais o número de pinos varia de 40 a 132. Como ilustrado na figura 3.1, os sinais associados a estes pinos permitirão ao microproces-

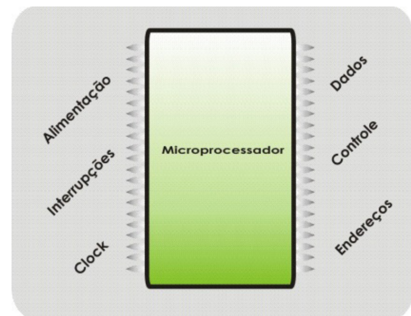


Figura 3.1: Microprocessador
Figura adaptada de [Moresi, E.]

sador a troca de informação com o seu ambiente, ou seja, memória e circuitos de E/S.

Uma análise detalhada dos sinais disponíveis nos pinos de um microprocessador permite conhecer os seus diferentes modos de funcionamento (lógico e elétrico) e as suas possibilidades em termos de interfaces. Os sinais externos dos computadores são organizados em três grupos: os sinais de endereço, os sinais de dados e os sinais de controle. A cada um destes grupos é associada uma linha de comunicação (ou um barramento), respectivamente. Estes barramentos permitirão conectar os pinos do microprocessador aos pinos de mesma função dos circuitos de memória ou de E/S.

A figura 3.2 mostra, de forma bem simplificada, os barramentos em um microprocessador. Temos o chamado barramento de dados, através do qual trafegam os dados que são transmitidos ou recebidos pelo microprocessador. Os dados transmitidos podem ser enviados para a memória ou para um dispositivo de saída, como o vídeo.

Os dados recebidos podem ser provenientes da memória, ou de um dispositivo de entrada, como o teclado. Cada uma das "perninhas" do microprocessador pode operar com um bit. No microprocessador da figura 1.2, temos um barramento de dados com 16 bits. Observe que as linhas desenhadas sobre o barramento de dados

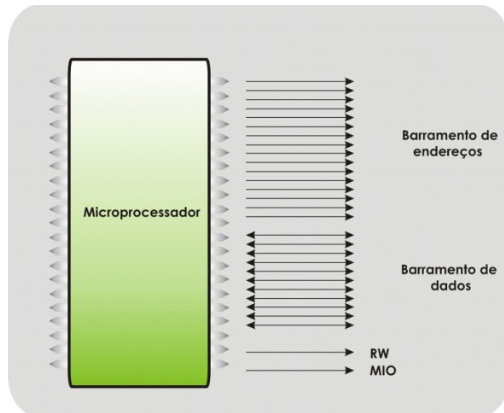


Figura 3.2: Representação simplificada de um microprocessador.
Figura adaptada de [Moresi, E.]

possuem duas setas, indicando que os bits podem trafegar em duas direções, saindo e entrando no microprocessador. Dizemos então que o barramento de dados é bidirecional.

Um exemplo de funcionamento é o carregamento de uma instrução. O microprocessador inicialmente carrega o endereço da instrução no barramento de endereços; em seguida, ele ativa um sinal no barramento de controle para especificar à memória uma operação de leitura; em resposta, a memória vai colocar, no barramento de dados, a palavra representando a instrução requisitada e previne o microprocessador (com um sinal no barramento de controle); ao perceber o sinal de validação, o microprocessador lê a palavra no barramento de dados e a armazena num de seus registros internos.

Além dos barramentos, a figura 3.2 mostra ainda dois sinais de controle que servem para definir se a operação a ser realizada é uma leitura ou uma gravação e se deve atuar sobre a memória ou sobre um dispositivo de E/S. São eles:

- ***MIO:*** este sinal indica se a operação diz respeito à memória ou a E/S;
- ***RW:*** este sinal indica se a operação é uma leitura ou uma gravação;

Através desses dois sinais, podem ser definidas 4 operações básicas: a) leitura da memória, b) escrita na memória, c) leitura de E/S (Ex: do teclado), d) escrita em E/S (Ex: no vídeo).

Outros exemplos de sinais do barramento de controle:

- ***INT:*** este sinal é uma entrada que serve para que dispo-

sitivos externos possam interromper o microprocessador para que seja realizada uma tarefa que não pode esperar. Por exemplo, a interface de teclado interrompe o microprocessador para indicar que uma tecla foi pressionada. Esta tecla precisa ser lida e seu código deve ser armazenado na memória para processamento posterior. As interfaces de drives e do disco rígido interrompem o microprocessador para avisar o término de uma operação de leitura ou escrita. Vários outros dispositivos também precisam gerar interrupções. Como existe apenas uma entrada INT, o microprocessador opera em conjunto com um chip chamado Controlador de Interrupções. Este chip é encarregado de receber requisições de interrupção de vários dispositivos e enviá-las ao microprocessador, de forma ordenada, através do sinal INT.

- ***NMI:*** este é um sinal de interrupção especial para ser usado em emergências. Significa Non MaskableInterrupt ou Interrupção não mascarável. Em outras palavras, esta interrupção deve ser atendida imediatamente. Ao contrário do sinal INT, que pode ser ignorado pelo microprocessador durante pequenos intervalos de tempo (isto se chama "mascarar a interrupção"), o sinal NMI é uma interrupção não mascarável. Nos PCs, o NMI é usado para informar erros de paridade na memória.

- ***INTA:*** significa *InterruptAcknowledge*, ou seja, reconhecimento de interrupção. Serve para o microprocessador in-

dicar que aceitou uma interrupção e que está aguardando que o dispositivo que gerou a interrupção identifique-se, para que seja realizado o atendimento adequado.

Existem ainda mais de uma dúzia de sinais no barramento de controle. Seu estudo é muito interessante para quem está preocupado em aprender detalhadamente como funciona um microprocessador e uma placa de CPU. Aqui não iremos abordá-los. Nosso objetivo é apenas fazer uma apresentação simplificada.

3.2 Componentes

As funções de controle e processamento necessitam de componentes compostos de circuitos digitais para sua realização. Estes componentes são interligados interna e externamente através de barramentos. A seguir detalharemos os principais componentes de um processador.

Unidade Lógica e Aritmética (ULA):

A ULA é o dispositivo da CPU que executa realmente as operações matemáticas com os dados. Estas operações podem ser, por exemplo, soma, subtração, multiplicação, divisão, operações lógicas AND, OR, XOR, NOT, deslocamento de bits à direita e esquerda, incremento, decremento e comparações. A ULA é um aglomerado de circuitos lógicos e componentes eletrônicos simples que, integrados, realizam as operações já mencionadas. Ela pode ser uma parte pequena da pastilha do processador usada em pequenos sistemas, ou pode compreender um considerável conjunto de componentes lógicos de alta velocidade.

A despeito da grande variação de velocidade, tamanho e complexidade, as operações aritméticas e lógicas realizadas por uma ULA seguem sempre os mesmos princípios fundamentais.

O circuito de cálculo mais simples é o somador que adiciona dois números de *n* bits. A ULA é o circuito que efetua diversas operações aritméticas e lógicas entre dois operandos. O tipo de tratamento a efetuar deve ser informado através de sinais de seleção de operação. Para revisar, a figura 3.3 mostra a composição de uma ULA, capaz de realizar quatro diferentes operações com dois números binários A e B: A e B, A ou B, Complemento de B e A+B.

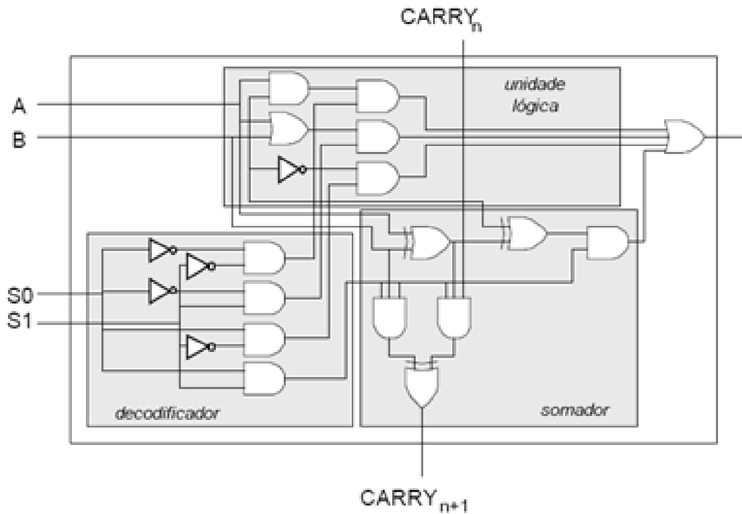


Figura 3.3: ULA de quatro operações sobre 1 bit.
Figura retiradade [Moresi, E.]

Registradores:

É um dispositivo capaz de memorizar uma informação. Na arquitetura de um microprocessador, os registradores, geralmente numerosos, são utilizados para assegurar o armazenamento temporário

de informações importantes para o processamento de uma dada instrução.

Para que um dado possa ser transferido para a ULA, é necessário que ele permaneça, mesmo que por um breve instante, armazenado em um registrador. Além disso, o resultado de uma operação aritmética ou lógica realizada na ULA deve ser armazenado temporariamente, de modo que possa ser utilizado mais adiante, ou apenas para ser, em seguida, transferido para a memória. Para entender estes propósitos, a CPU é fabricada com certa quantidade de registradores destinados ao armazenamento de dados. Servem, pois, de memória auxiliar da ULA.

No projeto de processadores, os registradores são normalmente divididos de forma a trabalharem com dados inteiros e em ponto flutuante. Os processadores implementam o banco de registradores separados por dois motivos: primeiro, isto permite que sejam colocados fisicamente próximos às unidades de execução que os utilizam. O banco de registradores para inteiros pode ser colocado próximo às unidades que executam operações inteiras, e o de ponto flutuante próximo às unidades de execução de ponto flutuante. Isto reduz o comprimento da fiação que liga o banco de registradores às unidades de execução e, portanto, reduz o tempo necessário para enviar dados de um para outro; o segundo motivo é que bancos de registradores separados ocupam menos espaço nos processadores que executam mais de uma instrução por ciclo. Os detalhes disto estão além do escopo deste curso, mas o tamanho de um banco de registradores cresce aproximadamente com o quadrado do número de leituras e escritas simultâneas que o banco permite.

Conceitualmente, registrador e memória são semelhantes. São a localização, a capacidade de armazenamento e os tempos de acesso às

informações que os diferenciam. Os registros se localizam no interior de um microprocessador, enquanto a memória é externa a este.

Unidade de Controle (UC):

É o dispositivo mais complexo da CPU. Além de possuir a lógica necessária para realizar a movimentação de dados e instruções dele para a CPU, através dos sinais de controle que emite em instantes de tempo programados, esse dispositivo controla a ação da ULA. Os sinais de controle emitidos pela unidade de controle ocorrem em vários instantes durante o período de realização de um ciclo de instrução e, de modo geral, todos possuem uma duração fixa e igual, originada em um gerador de sinais usualmente conhecido como relógio. Ao contrário de circuitos integrados mais comuns, cuja função é limitada pelo hardware, a unidade de controle é mais flexível. Ela recebe instruções da unidade de E/S, as converte em um formato que pode ser entendido pela unidade de aritmética e lógica, e controla qual etapa do programa está sendo executado.

Relógio ou clock:

É o dispositivo gerador de pulsos cuja duração é chamada de *ciclo.* A quantidade de vezes em que este pulso básico se repete em um segundo define a unidade de medida do relógio, denominada *frequência* a qual também usamos para definir velocidade na CPU. A unidade de medida usual para a frequência dos relógios de CPU é o Hertz (Hz), que significa 1 ciclo por segundo. Como se trata de frequências elevadas, abrevia-se os valores usando-se milhões de Hertz ou de ciclos por segundo (MegaHertz ou simplesmente, MHz). Assim, por exemplo, se um determinado processador funciona como seu relógio osci-

lando 25 milhões de vezes por segundo, sua frequência de operação é de 25 MHz. E como a duração de um ciclo, seu período, é o inverso da freqüência, então cada ciclo, neste exemplo, será igual ao inverso de 25.000.000 ou 1/25.000.000=0,00000004 ou 40 nanossegundos.

Registrador de Dados da Memória (RDM):

Armazena os dados que estão sendo transferidos de/para a memória. Em geral o RDM possui um tamanho igual ao da palavra do barramento de dados.

Registrador de Endereços da Memória (REM):

Armazena o endereço da célula onde deve ser feita a próxima operação de leitura ou escrita na memória. O REM possui um tamanho igual ao dos endereços da memória.

Registrador de Instruções (RI):

É o registrador que tem a função específica de armazenar a instrução a ser executada pela CPU. Ao se iniciar um ciclo de instrução, a UC emite o sinal de controle que acarretará a realização de um ciclo de leitura para buscar a instrução na memória, e que será armazenada no RI, via barramento de dados e RDM.

Contador de Instruções ou Program Counter (PC):

É o registrador cuja função específica é armazenar o endereço da próxima instrução a ser executada. Tão logo a instrução que vai ser executada seja buscada (lida) da memória para a CPU, o sistema providencia a modificação do conteúdo do PC de modo que ele passe a armazenar o endereço da próxima instrução na sequência. Por isso,

é comum definir a função do PC como sendo a de "armazenar o endereço da próxima instrução", que é o que realmente ele faz durante a maior parte da realização de um ciclo de instrução.

Decodificador de instruções:

É um dispositivo utilizado para identificar as operações a serem realizadas, que estão correlacionadas a instrução em execução. Em outras palavras, cada instrução é uma ordem para que a CPU realize uma determinada operação. Como são muitas instruções, é necessário que cada uma possua uma identificação própria e única. A unidade de controle está, por sua vez, preparada para sinalizar adequadamente aos diversos dispositivos da CPU, conforme ela tenha identificado a instrução a ser executada. O decodificador recebe na entrada um conjunto de bits previamente escolhido e específico para identificar uma instrução de máquina e possui 2^N saídas, sendo N a quantidade de algarismos binários do valor de entrada. Se o valor presente sobre as entradas é k, apenas a saída de ordem k será ativa. O decodificador analisa a informação nas suas entradas e fornece na saída, de maneira exclusiva, a indicação ou significado desta informação dentre várias possibilidades.

3.3 Funcionalidades

Muitos autores dividem as atividades realizadas pela CPU em duas grandes categorias funcionais:

Função Processamento

Encarrega-se de realizar as atividades relacionadas com a efetiva execução de uma operação, ou seja, *processar*. O dispositivo principal

desta área de atividades de uma CPU é chamado de Unidade Lógica Aritmética - ULA. Os demais componentes relacionados com a função processamento são os registradores, que servem para armazenar dados a serem usados pela ULA. A interligação entre estes componentes é efetuada pelo barramento interno da CPU.

A capacidade de processamento de uma CPU é em grande parte determinada pelas facilidades embutidas no *hardware* da ULA para realizar as operações matemáticas projetadas. Um dos elementos fundamentais é a definição do tamanho da palavra. Um tamanho maior ou menor de palavra acarreta, sem dúvida, diferenças fundamentais no desempenho da CPU, e, por conseguinte, do sistema como um todo.

Função Controle

É exercida pelos componentes da CPU que se encarregam das atividades de busca, interpretação e controle da execução das instruções, bem como do controle da ação dos demais componentes do sistema de computação. A área de controle é projetada para entender o que fazer, como fazer e comandar quem vai fazer no momento adequado. Uma analogia pode ser feita com os seres humanos, imaginando que a área de controle é o cérebro que comanda o ato de andar, enquanto a área de processamento são os músculos e ossos das pessoas que realizam efetivamente o ato. Os nervos podem ser relacionados com os barramentos entre os diversos elementos envolvidos.

O componente vital para as funções de controle é a Unidade de Controle (UC). Ela recebe como entrada o valor do registrador de instruções e decodifica-o (através do decodificador de instruções). Para cada código de instruções ele gera uma sequência de sinais diferentes, ativando os circuitos correspondentes para cada uma das tarefas neces-

sárias para a busca e execução da instrução a ser executada.

Cada sinal de controle comanda uma **microinstrução** (que denota uma tarefa a ser executada para a execução de uma operação). Uma microinstrução pode ser responsável pela realização de uma carga em um registrador, uma seleção de um dado para entrada em um determinado componente, uma ativação da memória, a seleção de uma operação da ULA ou a habilitação de um circuito lógico, para citar alguns exemplos.

Os dispositivos básicos que devem fazer parte daquela área funcional são: unidade de controle, decodificador, registrador de instrução, contador de instrução, relógio e os registradores de endereço de memória e de dados da memória.

3.4 Ciclo de Instrução e Microprogramação

A CPU trabalha diretamente com a memória principal e o processamento é feito por ciclo regulado pelo *clock* (relógio). Estas etapas compõem o que se denomina **ciclo de instrução**. Este ciclo se repete indefinidamente até que o sistema seja desligado, ou ocorra algum tipo de erro, ou seja, encontrada uma instrução de parada.

A sequência resumida desse ciclo é:

- Buscar (cópia) instrução na memória principal;
- Executar aquela instrução;
- Buscar a instrução seguinte;
- Executar a instrução seguinte;
- E assim por diante (milhões de vezes por segundo).

Primeiro, o processador busca a instrução na memória. Então a instrução é decodificada para determinar qual instrução ela é e quais os seus registradores de entrada e saída. A instrução decodificada é representada por um conjunto de bits que dizem ao hardware como executar a instrução. Estes padrões de bits são enviados para a próxima seção da unidade de execução a qual lê as entradas da instrução a partir dos registradores. A instrução decodificada e os valores dos registradores de entrada são encaminhados para o hardware, que calcula o resultado da instrução, e o resultado é escrito de volta no banco de registradores.

As instruções que acessam o sistema de memória têm fluxo de execução semelhante, exceto que a saída da unidade de execução é enviada ao sistema de memória por ser, ou endereço de uma operação de leitura, ou endereço e o dado de uma operação de escrita.

A figura 3.4 exemplifica a execução de instruções.

Figura 3.4: Execução de instruções.
Figura adaptada de [Carter, N.]

Os módulos que implementam os diferentes passos na execução da instrução são fisicamente dispostos próximos um ao outro interconectados através de barramento. À medida que a instrução é executada, os dados fluem pelo barramento de um módulo para o seguinte com cada módulo executando o seu trabalho em uma sequência.

Este ciclo de instrução pode ser descrito em LTR (Linguagem de Transferência entre Registradores), de modo que possamos acompanhar sua realização com a movimentação de informações entre os componentes da CPU. As linhas abaixo ilustram este "Algoritmo" de funcionamento.

```
Iniciar
        CICLO RI   ←   MEM[PC]
        PC   ←   PC + 1
        Interpretar o Código da Operação
        Enquanto houver operando não
        carregados
         Buscar Operandos
         PC   ←   PC + 1
        Executar a instrução
        Ir para CICLO
Fim
```

Inicialmente, o conteúdo de memória no endereço da próxima instrução a ser executada (PC) tem seu valor transferido para RI. Logo após, o valor de PC é incrementado para o endereço da próxima instrução a ser executada. O decodificador de instruções irá receber os bits referentes ao código da operação e decodificá-lo, dando entrada na UC este valor. A UC gera os sinais necessários para a execução da Instrução.

Em um processador microprogramado, o hardware não precisa executar diretamente as instruções do conjunto de instruções. Ao invés disso, o hardware executa microoperações muito simples, e cada instrução determina uma sequência de microoperações que são utilizadas para implementar a instrução. Essencialmente, cada instrução do conjunto de instruções é traduzida pelo hardware em um pequeno programa de microinstruções, de modo semelhante ao modo como o compilador traduz cada instrução de um programa em linguagem de alto nível para uma sequência de instruções em linguagem de máquina.

Processadores microprogramados possuem uma pequena memória que mantém a sequência de microoperações utilizadas para implementar uma instrução do conjunto de instruções. Para executar uma instrução, o processador acessa essa memória para localizar o conjunto de microinstruções necessárias para implementar a instrução e, então, executa as microinstruções em sequência.

Processadores modernos tendem a não utilizar microprogramação por dois motivos: primeiramente porque agora se tornou prático implementar a maior parte das instruções diretamente no hardware, isso devido aos avanços na tecnologia VLSI, o que torna o microcódigo desnecessário; Em segundo lugar, processadores microprogramados tendem a ter um desempenho pior que os processadores não microprogramados por causa do tempo adicional envolvido na busca de cada microinstrução na memória de microinstruções.

3.5 Interrupções

O trabalho da CPU é executado em laço infinito conforme já foi afirmado, porém, os periféricos precisam de atenção da CPU de

vez em quando. Para chamar a atenção da CPU para si, um periférico usa um cógido próprio chamado de código de interrupção, este gera na CPU uma operação de interrupção. Ao ser interrompida, a CPU precisa salvar todo o seu conteúdo em alguma área de memória para atender ao periférico. Após realizar o atendimento do periférico, a CPU retoma os valores armazenados na memória ao receber a interrupção e continua o processamento normalmente. A esse processo dá-se o nome de TROCA DE CONTEXTO, e será estudado com mais detalhes na disciplina de Sistemas Operacionais.

Uma interrupção também pode acontecer devido à execução normal de um programa. O próprio sistema operacional gera interrupções constantemente para a CPU. No passado, muitos periféricos recém-adicionados ao PC não funcionavam a contento, pois usavam a mesma interrupção já usada por outros já instalados no sistema. Esse fenômeno é conhecido como conflito de hardware. Esse problema não existe no barramento PCI (visto na seção 1.4.2) junto com os Sistemas Operacionais lançados a esta época que implementaram uma tecnologia conhecida como *plug-and-play.* Quando novas placas são adicionadas aos computadores não se tem mais a preocupação de resolver problemas de conflitos de interrupções ou de endereços-base para identificação do periférico, tudo é atribuído pelo BIOS que suporta *plug-and-play* e configura automaticamente.

3.5.1 Características básicas das Interrupções

A requisição de interrupção pode ocorrer a qualquer momento (assincronamente), sendo ela indicada pela ativação de um "*flag*" pelo dispositivo periférico. O processador reconhece a interrupção, envia

sinais de controle, completa a execução da instrução corrente, salva o conteúdo dos registradores de interesse (contador de programa, status, etc) e atende ao dispositivo periférico que solicitou a interrupção, transferindo o controle para a rotina de tratamento da interrupção. Ao término da execução desta rotina, o microprocessador desativa o "flag" de indicação de interrupção, restaura os registradores que foram salvos e transfere o controle para a instrução seguinte ao ponto de interrupção do programa.

Algumas aplicações de tempo real envolvem módulos de programas críticos que não podem ser interrompidos durante sua execução. Algumas interrupções podem ter seu tratamento postergado, enquanto outras necessitam de tratamento imediato (por exemplo, coleta de dados e alarmes). Em função disso, a maioria dos processadores apresentam *interrupções mascaráveis* e *interrupções não-mascaráveis*. Através de instruções apropriadas o programa pode habilitar ou desabilitar uma interrupção mascarável, enquanto que a não-mascarável deverá ser sempre atendida, devendo ser reservada, assim, apenas para eventos de alta importância. As interrupções dos microprocessadores costumam ser dos seguintes tipos: *vetoradas* ou *não-vetoradas*.

Interrupção Vetorada

Virtualmente cada processador reserva uma área de memória especifica para tratar cada uma das interrupções. Estas localizações são chamadas vetores de interrupção. Este tipo exige a identificação do dispositivo periférico que solicita interrupção. Essa identificação é utilizada para a localização do endereço da subrotina de tratamento da interrupção, em uma tabela localizada em uma região determinada da memória do microprocessador (vetor de interrupções).

Interrupção não vetorada

Sem vetor de interrupção, o programa deve verificar cada possível fonte de interrupção para ver quem causou a interrupção, aumentando, assim, o tempo de resposta. Em geral uma causa da interrupção é gerada e colocada em um registrador causa. Como resultado, o Sistema operacional (ou Kernel) pode imediatamente determinar a identidade do dispositivo que o interrompeu.

3.6 Medidas de Desempenho

A medida geral de desempenho de um sistema de computação depende fundamentalmente da capacidade e velocidade de seus diferentes componentes, da velocidade com que estes componentes se comunicam entre si e do grau de compatibilidade entre eles (p. ex., se a velocidade da CPU de um sistema é muito maior que a da memória).

Considerando a existência de tantos fatores que influenciam o desempenho de um sistema de computação, desenvolveram-se diversos meios de medir seu desempenho, entre os principais relacionados exclusivamente com a CPU destacam-se: MIPS e FLOPS.

O desempenho dos processadores, em geral, é medido em termos de sua velocidade de trabalho. Como o trabalho da CPU é executar instruções, criou-se a unidade MIPS – Milhões de instruções por segundo. O MIPS é muito questionado, pois mede apenas a execução de instruções, sendo que existem diferentes instruções com tempos de execução distintos, como por exemplo, multiplicação de números inteiros e multiplicação de números reais (ponto flutuante). Em contraste com o MIPS, o FLOPS – Operações de Ponto Flutuante por Segundo mede basicamente o desempenho da ULA, analisando apenas

as instruções mais complexas (as que envolvem ponto flutuante). Hoje em dia, os microprocessadores estão na faixa de MFLOS (milhões de flops) onde são encontradas máquinas com GFLOPS (supercomputadores).

3.7 Mais de uma Instrução por Ciclo

Descrevendo o funcionamento da CPU na realização de seus ciclos de instrução foi observado que, embora o ciclo de instrução seja composto de várias etapas, ele é realizado de forma sequencial, isto é, uma etapa se inicia após a conclusão da etapa anterior. Desta forma, enquanto a fase de decodificação da instrução estava sendo executada, a busca (RDM e REM) e a execução (ULA) estavam ociosas.

A situação acima leva a crer que a CPU não pode executar mais de uma instrução por ciclo, mas isso não é verdade. Graças aos avanços da tecnologia, existem CPUs que executam mais de uma instrução por ciclo, sendo chamadas de superescalares.

Pipeline

O *pipeline* é uma técnica de implementação de CPU na qual múltiplas instruções estão em execução ao mesmo tempo. O processador é construído com diversos estágios distintos, cada um responsável pela execução de uma parte da instrução e possuindo o seu próprio bloco de controle. Assim que um estágio completa sua tarefa com uma instrução passa esta para o estágio seguinte e começa a tratar da próxima instrução. Uma vez que diversas instruções são executadas ao mesmo tempo, obtêm-se um acréscimo no desempenho do processador.

A figura 3.5 exemplifica um *pipeline* na produção em serie de

bicicletas. A divisão das tarefas resultou em 5 etapas: na primeira é feita a estrutura da bicicleta com pedais e correia; na segunda é instalada a roda dianteira; na terceira é colocada a roda traseira; na quarta é colocado o assento; e, por fim, na quinta e última etapa é colocado o guidom. É conveniente lembrar que o tempo para a produção de uma bicicleta não diminui, mas o tempo entre bicicletas sim. A primeira bicicleta demora cinco unidades de tempo para ser produzida, mas logo a seguir, em cada unidade de tempo uma nova bicicleta é produzida (se não houverem problemas).

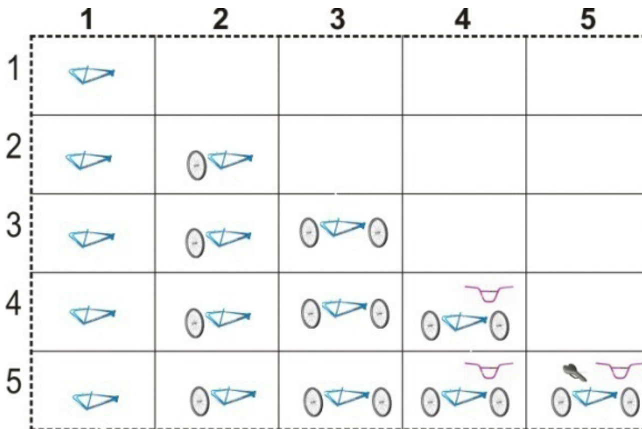


Figura 3.5: Pipeline.
Figura adaptada de [Rebonatto, M.]

Processadores Superescalares

Processadores superescalares possuem várias unidades de execução permitindo que diversas instruções sejam executadas ao mesmo tempo. Um aspecto importante neste tipo de processador é que, como mais de uma instrução é executada simultaneamente e devido a tempos de execução diferentes entre elas, ou a conflitos ocorridos, pode se dar à execução de instruções fora da ordem original do programa.

A figura 3.6 ilustra este tipo de organização de CPUs.

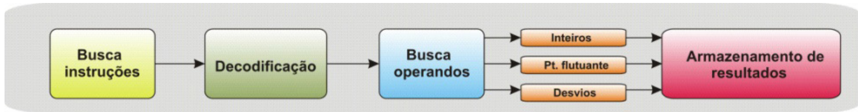


Figura 3.6: Maquina Superescalar.
Figura adaptada de [Rebonatto, M.]

Estes processadores possuem unidades funcionais específicas para cada tipo de instrução. As unidades funcionais podem internamente ser divididas em várias etapas cada uma. Desta forma, têm-se pipelines distintos: para execução de instruções envolvendo números inteiros, reais e desvios.

3.8 EXERCÍCIOS

1. Defina processador enfocando por qual tipo de operações ele é responsável.

2. Mostre e explique de forma resumida os componentes de um processador.

3. Explique as funções de controle e processamento de um processador.

4. Quais as etapas de ciclo de instrução? Explique-as.

5. Se ler uma instrução da memória gasta 5 ns, decodificar gasta 2ns, 3 ns para ler o banco de registradores, 4 ns para executar o cálculo exigido pela instrução e 2 ns para escrever o resultado no banco de registradores, qual é a frequência máxima de relógio do processador sabendo que fmax = 1/duração do ciclo?

6. O que é microprogramação? Por que foi utilizada em computadores antigos? Por que os processadores atuais abandonaram essa técnica?

7. Explique a ideia de *pipeline* e processadores superescalares.

8. Por que processadores implementam bancos de registradores separados para inteiros e ponto flutuante?

4. *PIPELINE* E MÁQUINAS SUPERESCALARES

4.1 Introdução

A maioria das arquiteturas e processadores projetadas depois de 1990 possui uma organização chamada *pipeline*. A tradução literal de "*pipeline*" seria "linha de dutos", uma tradução mais aproximada do significado que este termo assume em arquiteturas de computadores seria "linha de montagem". Organizar a execução de uma tarefa em *pipeline* significa dividir a execução dessa tarefa em estágios sucessivos, exatamente como ocorre na produção de um produto em uma linha de montagem.

Considere uma tarefa que utiliza, para sua execução, um módulo de processamento executando k sub-tarefas. No esquema apresentado na figura 4.1, se cada sub-tarefa requer um tempo de execução *t* segundos, a tarefa como um todo irá ser executada em *kt* segundos. Para a execução de uma sequência de tais tarefas, cada nova tarefa na sequência só poderá ter sua execução iniciada após *kt* segundos, o tempo necessário para a liberação do módulo de processamento.

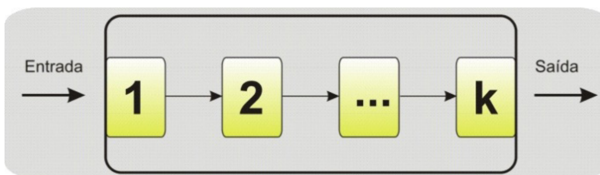


Figura 4.1: Execução de tarefas.
Figura adaptada de [Ricarte, I.]

Considere uma abordagem alternativa (figura 4.2) na qual o módulo de processamento é dividido em *k* módulos menores (está-

gios), cada um responsável por uma sub-tarefa, cada um operando concorrentemente aos demais estágios:

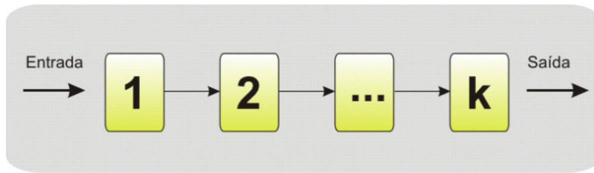


Figura 4.2: Princípio de pipeline.
Figura adaptada de [Ricarte, I.]

O princípio da técnica de *pipeline* inicia uma nova tarefa antes que o resultado para a tarefa anterior na sequência de tarefas tenha sido gerado. A utilização dos estágios de um *pipeline* pode ser graficamente representada através de um **diagrama de ocupação espaço tempo**. Por exemplo, para um *pipeline* de quatro estágios mostrado na figura 4.3. Na execução em *pipeline*, cada tarefa individualmente ainda requer *kt* segundos. Entretanto, o módulo é capaz de gerar um novo resultado a cada *t* segundos

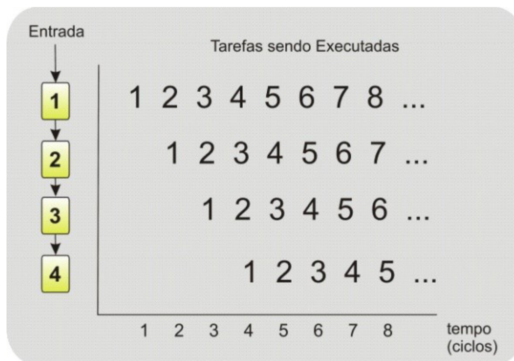


Figura 4.3: Pipeline de quatro estágios.
Figura adaptada de [Ricarte, I.]

Existem algumas regras muito importantes que devem ser seguidas quando se projeta uma arquitetura em *pipeline*:

- Todos os estágios devem ter a mesma duração de tempo;
- Deve-se procurar manter o *pipeline* cheio a maior parte do tempo;
- O intervalo mínimo entre o término de execução de duas instruções consecutivas é igual ao tempo de execução do estágio que leva mais tempo;
- Dadas duas arquiteturas implementadas com a mesma tecnologia, a arquitetura que é construída usando *pipeline* não reduz o tempo de execução de instruções, mas aumenta a frequência de execução de instruções (*throughput*).

O tempo total para a execução de uma operação em *pipeline* é, em geral, ligeiramente maior que o tempo para executar a mesma operação monoliticamente (sem *pipeline*). Um dos *overheads* associados à operação de um *pipeline* é decorrente da necessidade de se transferir dados entre os estágios.

Há duas estratégias básicas para controlar a transferência de dados entre os estágios de um *pipeline*: o **método assíncrono** (figura 4.4) e o **método síncrono** (figura 4.5). No método assíncrono, os estágios do *pipeline* comunicam-se através de sinais de *handshaking*, indicando a disponibilidade de dados do estágio corrente para o próximo estágio (RDY) e indicando a liberação do estágio corrente para o estágio anterior (ACK).

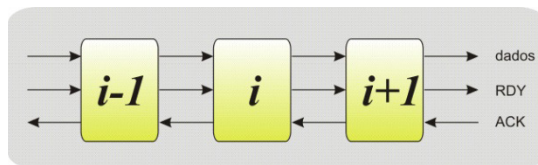


Figura 4.4: Método assíncrono.
Figura adaptada de [Ricarte, I.]

No método síncrono, os estágios do *pipeline* são interconectados por *latches* que armazenam os dados intermediários durante a transferência entre estágios, que é controlada por um sinal de relógio. Neste caso, o estágio com operação mais lenta determina a taxa de operação do *pipeline*.

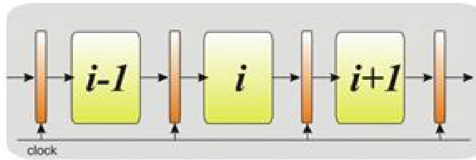


Figura 4.5: Método síncrono.
Figura adaptada de [Ricarte, I.]

O método assíncrono é o que permite maior velocidade de operação do *pipeline*. Entretanto, o método síncrono é o mais adotado devido à sua simplicidade de projeto e operação.

A figura 4.6 mostra a execução típica de instruções em um *pipeline.* Cada um dos retângulos mostrados representa um banco de *flip-flops* que são elementos de memória utilizados para armazenar os resultados no final de cada estágio do *pipeline*.

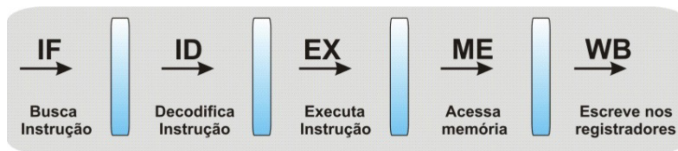


Figura 4.6: Estrutura de uma máquina pipeline.
Figura adaptada de [Vargas, F.]

Os *pipelines* utilizados em microprocessadores normalmente são síncronos, portanto, um sinal de relógio não mostrado na figura 4.7 habilita o elemento de memória a "passar" seus resultados para o estágio seguinte. Como existe um único relógio para comandar o *pipeline*,

o tempo gasto em todos os estágios do *pipeline* é idêntico e não pode ser menor que o tempo gasto no estágio mais lento. Este tempo gasto em cada estágio é o período de *clock* utilizado para comandar o *pipeline* determinando a velocidade de execução das instruções.

Ciclo de *clokc*	*Estágio do pipeline onde a instrução se encontra*							
	0	1	2	3	4	5	6	7
Instrução 01	IF	ID	EX	ME	WB			
Instrução 02		IF	ID	EX	ME	WB		
Instrução 03			IF	ID	EX	ME	WB	
Instrução 04				IF	ID	EX	ME	WB

Figura 4.7: Execução de uma seqüência de instruções em um pipeline.
Figura adaptada de [Vargas, F.]

4.2 Duração de Ciclos

Para implementar *pipelines*, os projetistas dividem o caminho de dados de um processador em seções e colocam *latches* entre cada seção, como mostra a figura 2.6. No início de cada ciclo, os *latches* leem as suas entradas e as copiam para suas saídas, as quais permanecerão constantes por todo o ciclo. Isto divide o caminho de dados em diversas seções, cada uma das quais tem latência de um ciclo de relógio, uma vez que a instrução não pode passar através de um *latch* até que o próximo ciclo seja iniciado.

Se for considerado apenas o número de ciclos necessários para executar um conjunto de instruções, pode parecer que um *pipeline* não melhora o desempenho do processador. De fato, como veremos adiante, projetar um *pipeline* para um processador aumenta o número de ciclos de relógio que são necessários para executar um programa,

porque algumas instruções ficam presas no *pipeline* esperando que as instruções que geram suas entradas sejam executadas. O benefício de desempenho do *pipeline* vem do fato que em um estágio menos lógica necessita ser executada sobre os dados, o que possibilita processadores com *pipelines* ter ciclos de tempo mais reduzidos do que implementações sem *pipeline* do mesmo processador. Uma vez que um processador com *pipeline* tem uma taxa de rendimento de uma instrução por ciclo, o número total de instruções executadas por unidade de tempo é maior em um processador com *pipeline* que, assim, dão um desempenho melhor.

A duração do ciclo em um processador com *pipeline* é dependente de quatro fatores: a duração do ciclo por parte do processador que não tem *pipeline*, o número de estágios do *pipeline,* a homogeneidade com que a lógica do caminho dos dados é dividida entre os estágios e a latência dos *latches*. Se a lógica pode ser dividida homogeneamente entre os estágios do *pipeline*, o período de tempo de um processador com *pipeline* é:

$$duraçãodociclo_{compipeline} = \frac{duraçãodociclo_{compipeline}}{númerodeestágiosdepipeline} + latênciadotatch$$

Cada estágio contém a mesma fração da lógica original, mais um *latch*. À medida que o número de estágios aumenta, a latência do *latch* torna-se uma parte cada vez menor da duração do ciclo, limitando o benefício de dividir um processador em um número muito grande estágios de *pipeline*.

<u>Exemplo</u>: *um processador sem pipeline tem uma duração de ciclo de 25 ns. Qual é a duração do ciclo de uma versão desse processador com pipeline de 5 estágios divididos homogeneamente, se cada latch tem uma latência de 1 ns? E se o processador for dividido em 50 estágios? Que conclusão*

podemos tirar quando comparado os dois processadores?

__**Solução:**__ *aplicando a equação, a duração do ciclo para o pipeline de 5 estágios é (25ns/5) + 1 = 6 ns. Para o pipeline de 50 estágios a duração do ciclo é (25ns/50)+1 = 1,5 ns. No pipeline de 5 estágios a latência do latch é de apenas 1/6 da duração global do ciclo, enquanto que no pipeline de 50 estágios a latência é de 2/3 da duração total do ciclo. Isso mostra que aumentando a quantidade de estágios, a latência do latch influencia muito na duração total do ciclo.*

Exemplo retirado de [Carter, N.]

Com frequência, a lógica do caminho de dados não pode ser dividida em estágios de *pipeline* com latência igual. Por exemplo, acessar o banco de registradores pode demorar 3 *ns*, enquanto decodificar uma instrução pode demorar 4 *ns*. Quando estão decidindo a forma de dividir o caminho de dados em estágio de *pipeline*, os projetistas devem equilibrar o seu desejo de que cada estágio contenha a mesma latência e a quantidade de dados que precisa ser armazenada no *latch*.

Algumas partes do caminho de dados, como a lógica de decodificação das instruções, são irregulares, fazendo com que seja difícil dividi-las em estágios. Para outras partes que geram grande quantidade de valores intermediários é necessário colocar os *latches* em locais onde haja menos resultados e, assim, menos bits precisam ser armazenados.

Quando um processador não pode ser dividido em estágios de *pipeline* com latência igual, a duração do ciclo de relógio do processador é igual à latência do estágio de *pipeline* mais longo, mais o atraso do *latch*, uma vez que a duração do ciclo deve ser longa o suficiente para que o estágio de *pipeline* mais longo possa completar e armazenar os resultados no *latch* que está entre ele e o próximo estágio.

<u>Exemplo</u>: suponha que um processador sem pipeline, com uma duração de ciclo de 25 ns, esteja dividido em 5 estágios com latências de 5, 7, 3, 6 e 4 ns. Se a latência do latch for de 1 ns qual é a duração do ciclo?

<u>Solução</u>: o estágio de pipeline mais longo demora 7 ns. Somando o atraso de latch de 1 ns a este estágio, resulta em uma latência de 8 ns, que é a duração do ciclo.

Exemplo retirado de [Carter, N.]

4.3 Latência

A **latência** de um *pipeline* é o tempo necessário para uma instrução atravessar todo o *pipeline*. Portanto para calcular o tempo de latência de uma máquina com *pipeline* basta multiplicar o período do *clock* pelo número de estágios no *pipeline*. A latência é importante apenas para determinar quanto tempo a execução da primeira instrução leva para ser completada.

Enquanto o pipeline pode reduzir a duração do ciclo de um processador, aumentando, assim, a taxa de rendimento das instruções, ele aumenta a latência do processador em, pelo menos, a soma de todas as latências dos latches. A latência de um pipeline é a soma do tempo que cada instrução demora para passar através do pipeline, o que é o produto do número de estágios pela duração do ciclo de relógio.

<u>Exemplo</u>: se processador sem pipeline com duração de ciclo de 25 ns for dividido homogeneamente em 5 estágios utilizando latches de latência de 1 ns, qual é a latência total do pipeline? E se o processador fosse dividido em 50 estágios?

Solução: aplicando a equação, a duração do ciclo para o pipeline de 5 estágios é (25ns/5)+1 = 6 ns. Para o pipeline de 50 estágios a duração do ciclo é (25ns/50)+1 = 1,5 ns. Dado isto, podemos calcular a latência de cada pipeline multiplicando a duração do ciclo pelo número de estágios. Isto resulta em uma latência de 30 ns para o pipeline de 5 estágios e 75 ns para o pipeline de 50 estágios.

Exemplo retirado de [Carter, N.]

Este exemplo mostra o impacto que o pipeline pode ter sobre a latência, especialmente à medida que o número de estágios cresce. O pipeline de 5 estágios tem uma latência de 30 ns, 20% mais demorado que o processamento sem pipeline, cuja duração do ciclo é de 25 ns; enquanto o pipeline de 50 estágios tem latência de 75 ns, três vezes a do processador original.

Pipelines com estágios não uniformes utilizam a mesma fórmula, embora tenham um aumento ainda maior na latência, porque a duração do ciclo precisa ser longa o suficiente para acomodar o estágio mais longo do *pipeline*, mesmo que os outros sejam mais curtos.

Exemplo: suponha que um processador sem pipeline, com uma duração de ciclo de 25 ns, esteja dividido em 5 estágios com latências de 5, 7, 3, 6 e 4 ns. Se a latência do latch for de 1 ns, qual é a latência resultante?

Solução: o estágio de pipeline mais longo demora 7 ns. Somando o atraso de latch de 1 ns a este estágio, resulta em uma latência de 8 ns, que é a duração do ciclo. Para obtermos a latência resultante basta multiplicá-la pelo número de ciclos obtendo 40 ns de latência total.

Exemplo retirado de [Carter, N.]

4.4 Bolhas

Uma **bolha** em um *pipeline* consiste em uma sequência de um ou mais períodos de *clock* em que um estágio do *pipeline* está vazio. Se um estágio do *pipeline* estiver vazio no ciclo de *clock* n, consequentemente o estágio seguinte estará vazio no ciclo de *clock*n+1. Desta forma bolhas formadas na entrada de um *pipeline* propagam-se através do *pipeline* até desaparecerem no último estágio. Situações que geram bolhas em *pipelines* incluem:

- a execução de instruções de desvio;
- o atendimento de interrupções;
- o tratamento de exceções;
- o retardo na execução de instruções devido a dependências existentes com instruções que a precedem.

No caso de atendimento de exceções e interrupções não existem muitas técnicas efetivas para minorar o problema da formação de bolhas no *pipeline,* pois estas ocorrências são bastante imprevisíveis. Quando da execução de desvios condicionais, a formação de bolhas pode ser reduzida através da utilização de **predição de ocorrência e desvio**. Já em se tratando de dependências, a formação de bolhas pode ser minorada através do **re-ordenamento de instruções**.

4.5 Previsão de Desvios

O problema introduzido por desvios condicionais em arquiteturas organizadas em *pipeline* é que quando o desvio é decodificado na

unidade de decodificação de instruções é impossível decidir se o desvio será executado ou não. Isto é, não se pode determinar *a priori* qual a próxima instrução a ser executada. Existem duas possibilidades:

- a condição que determina o desvio é falsa e o desvio não é executado, neste caso a próxima instrução a ser executada é a instrução seguinte à instrução de desvio no programa;
- condição é verdadeira e o endereço da próxima instrução a ser executada é determinada pela instrução de desvio.

A forma mais simples e menos eficaz de tratar um desvio condicional consiste em simplesmente paralisar a busca de instruções quando uma instrução de desvio condicional é decodificada. Com esta forma de tratamento de desvio, garantimos que todo desvio condicional gerará uma bolha no pipeline, pois a busca de instrução ficará paralisada até que se possa decidir se o desvio será executado ou não. Esta técnica só deve ser utilizada quando o custo de buscar e depois ter que descartar a instrução errada é muito alto.

Outra forma é tentar prever o que vai acontecer com o desvio. Neste caso a previsão pode ser **previsão estática** ou **previsão dinâmica**. A primeira é aquela em que, dada uma instrução de desvio em um programa, nós vamos sempre fazer a mesma previsão para aquela instrução. A segunda permite mudar a previsão para uma mesma instrução à medida que o programa é executado.

Previsão Estática

A forma mais simples de previsão estática é aquela em que a mesma previsão é feita para um dado desvio condicional. Esta técnica

de previsão é simples de implementar e pode tomar duas formas:

Aquela em que os desvios condicionais nunca ocorrem: assumindo que os desvios condicionais nunca ocorrem, simplesmente se continua o processamento normal de instruções incrementando o PC e buscando a instrução seguinte. Se for determinado mais tarde que o programa deve desviar as instruções buscadas terão que ser descartadas e os efeitos de quaisquer operações realizadas por elas devem ser anulados.

Aquela em que os desvios condicionais sempre ocorrem*:* assumindo que os desvios condicionais sempre ocorrem, é necessário calcular o endereço de desvio muito rapidamente já na Unidade de Decodificação de Instrução para dar tempo de buscar a nova instrução no endereço especificado pela instrução de desvio.

Algumas observações feitas por projetistas de computadores após analisar diversos programas indicam que um grande número de desvios ocorre no final de laços de programa. Se um laço de programa for executado *n* vezes, o desvio que se encontra no final do laço irá ocorrer *n* vezes e não somente uma vez no final do laço. Alguns programadores também observaram que desvios condicionais originados por comandos IF em linguagens de alto nível tendem a não ocorrer. Portanto, existem evidências de que seria desejável possuir a capacidade de fazer previsão estática diferenciada para diferentes instruções de desvio em um mesmo programa.

Uma solução para esta situação consiste em adicionar um bit no código de instruções de desvio condicional para informar o hardware

se aquele desvio será provavelmente executado ou provavelmente não executado. Este bit deverá ser especificado pelo compilador. Como a determinação de que a previsão será de execução ou não do desvio é feita em tempo de compilação, este método de previsão também é estático.

Previsão Dinâmica

Na previsão dinâmica de desvios condicionais uma mesma instrução de desvio pode receber uma previsão de ser ou não executada em diferentes momentos do programa. Uma solução comum é criar uma tabela de desvios em hardware. Essa tabela pode ser gerenciada na forma de uma cache. A cada vez que uma instrução de desvio é executada, ela é adicionada à tabela e um bit é setado ou resetado para indicar se o desvio foi executado ou não. Na tabela também é registrado o endereço para qual o desvio foi realizado, desta forma na próxima vez que a mesma instrução de desvio for decodificada, esta tabela é consultada e a previsão feita é de que a mesma coisa (execução ou não execução do desvio) vai ocorrer de novo. Se a previsão for de execução do desvio o endereço no qual a nova instrução deve ser buscada já se encontra nesta tabela.

4.6 Processamento de Exceções

Exceções são eventos síncronos, resultado direto da execução do programa corrente e são frutos de instruções mal-sucedidas no *software* como, por exemplo, divisão por zero e *overflow* em operação aritmética.

A ocorrência de exceções em um computador é imprevisível e,

portanto, numa máquina em *pipeline* uma exceção normalmente resulta na formação de bolhas. Uma complicação adicional é que como existem várias instruções em diferentes estágios de execução ao mesmo tempo, é difícil decidir qual foi a instrução que causou a geração da exceção. Uma solução para minorar este problema é categorizar as exceções de acordo com o estágio do *pipeline* em que elas ocorrem. Por exemplo, uma instrução ilegal só pode ocorrer na unidade de decodificação de instruções (ID) e uma divisão por zero ou uma exceção devida a *overflow* só pode ocorrer na unidade de execução (EX).

Algumas máquinas com arquitetura em *pipeline* são ditas ter exceções imprecisas. Nestas máquinas não é possível determinar qual a instrução que causou a exceção. Outra complicação nas arquiteturas em *pipeline* é o aparecimento de múltiplas exceções no mesmo ciclo de *clock*. Por exemplo, uma instrução que causa erro aritmético pode ser seguida de uma instrução ilegal. Neste caso a unidade de execução (EX) gerará uma exceção por erro aritmético e a unidade de decodificação (ID) gerará uma exceção por instrução ilegal, ambas no mesmo ciclo de *clock*. A solução para este problema consiste em criar uma tabela de prioridade para o processamento de exceções.

4.7 Bolhas causadas por dependência

Outra causa de formação de bolhas em arquiteturas com *pipeline* são as dependências existentes entre instruções num programa. Para ilustrar este problema, vamos considerar o exemplo de pseudo programa *assembler* apresentado a seguir. O trecho de programa abaixo permuta os valores armazenados nas posições $800 e $1000 da memória.

A MOVE $800, D0 copia o valor do endereço $800 no registrador D0

B MOVE $1000, D1 copia o valor do endereço $1000 no registrador D1

C MOVE D1,$800 copia o valor do registrador D1 no endereço $800

D MOVE D0,$1000 copia o valor do registrador D0 no endereço $1000

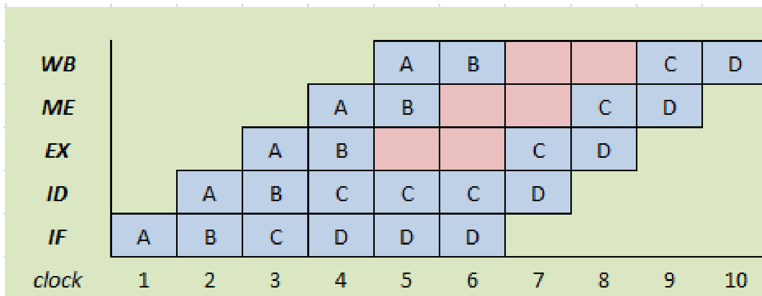


Figura 4.8: Formação de bolha devido à dependência de instruções. Exemplo retirado de [Vargas, N.]

Observe na figura 4.8 que no período de *clock* 6 quando a instrução C deveria executar no estágio de escrita na memória (ME) para escrever o valor de D1 no endereço $800. O valor lido do endereço $1000 ainda não está disponível em D1, pois ele só será escrito quando a instrução B tiver executado no estágio de escrita nos registradores (WB). Isso ocorre também no período de *clock* 6. Portanto, a execução da instrução C necessita ser atrasada por dois ciclos de relógio, gerando uma bolha no *pipeline*.

Uma forma simples de resolver este problema consiste em reordenar as instruções no programa, conforme ilustrado a seguir.

B	*MOVE $1000, D1*	*copia o valor do endereço $1000 no registrador D1*
A	*MOVE $800, D0*	*copia o valor do endereço $800 no registrador D0*
C	*MOVE D1,$800*	*copia o valor do registrador D1 no endereço $800*
D	*MOVE D0,$1000*	*copia o valor do registrador D0 no endereço $1000*

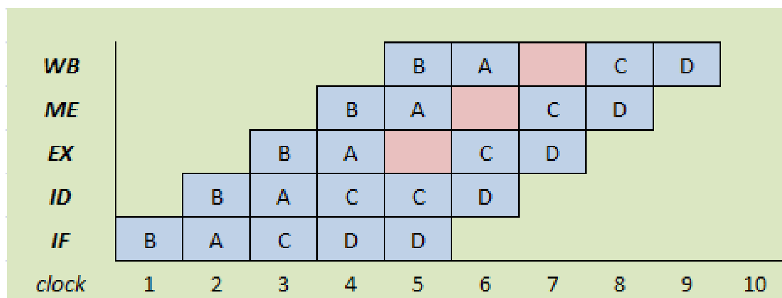


Figura 4.9: Minimização de bolhas por reordenação de instruções.
Exemplo retirado de [Vargas, N.]

A figura 4.9 ilustra que um simples re-ordenamento de instruções é suficiente neste caso para minimizar a bolha gerada pela dependência. Observe que o tempo total para a execução da sequência de instruções foi reduzido de 1 ciclo de *clock*.

Existem situações em que um simples re-ordenamento de instruções não é suficiente para minimizar (ou eliminar) bolhas causadas por dependências. Considere o programa abaixo que realiza a inversão de uma tabela de palavras localizadas entre os endereços $1000 e

```
A         MOVEA       #$1000, A0      inicializa A0
B         MOVEA       #$9000, A1      inicializa A1
C  loop   MOVE (A0), D0               copia em D0 dado apontado por A0
D         MOVE  (A1), D1              copia em D1 dado apontado por A1
E         MOVE  D0, (A1)              copia o valor em D0 no end. Indicado em A1
F         MOVE  D1, (A0)              copia o valor em D1 no end. Indicado em A0
G         ADDQ  #2, A0                incrementa o valor de A0
H         SUBQ  #2, A1                decrementa o valor de A0
I         CMPA  A0, A1                compara os endereços em A0 e A1
J         BGT LOOP                    enquanto A1 é maior que A0, continua
K         MOVE  #$500, A0             inicia outro procedimento
L         MOVE  #$17000, A1           inicia outro procedimento
```

WB					.	.	C	D		.	X	G	H			.	.
ME				.	.	C	D		E	F	G	H			.	.	
EX			A	B	C	D		E	F	G	H			I	J		
ID		A	B	C	D	E	E	F	G	H	I	I	I	J	K		C
IF	A	B	C	D	E	F	F	G	H	I	J	J	J	K	L	C	D
clock	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figura 4.10: Execução do programa com loop.
Exemplo retirado de [Vargas, N.]

Conforme ilustrado na figura 4.10, o laço do programa que efetivamente transfere os dados para fazer a inversão da tabela de palavras necessita de 12 ciclos de *clock* para executar. Para inverter uma tabela com *n* palavras é necessário executar este laço *n/2* vezes. Portanto o número de ciclos de *clock* necessário para realizar a inversão da tabela é *6n* (supondo que cada instrução é executada em apenas 1 ciclo de *clock*).

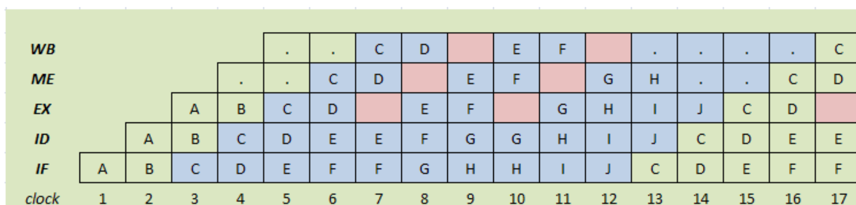
A figura indica a formação de 3 bolhas dentro da execução do loop:

1) a primeira bolha aparece no ciclo de *clock* 7 quando a instrução E tem que esperar até o fim do WB da instrução C. Em outras palavras, ela tem que aguardar até que o novo valor do registrador D0 produzido pela instrução C seja escrito no estágio de escrita em registradores WB, o que só vai ocorrer no ciclo de *clock* 6. Esta bolha tem o tamanho de 1 ciclo de *clock*.

2) a segunda bolha, com tamanho de 2 ciclos de *clock*, aparece no ciclo de *clock* 12 quando a instrução I tem que esperar até o fim do WB da instrução H.

3) a terceira bolha aparece no final do loop devido ao uso de uma previsão de que o desvio não será executado, o que faz com que o processador busque as instruções K e L que não serão executadas. Quando é determinado que estas instruções não serão executadas, elas são eliminadas do pipeline, gerando uma bolha de dois ciclos de clock.

O programa foi reescrito para que fosse eliminada a dependência entre as instruções de atualização dos endereços em A0 e A1 e a instrução de comparação. Para que isto ocorresse, a atualização dos valores dos registradores foi feita no início do laço e a inicialização de A0 e A1 foi alterada apropriadamente. O novo programa é apresenta-



WB					.	.	C	D		E	F		.	.	.	.	C
ME				.	.	C	D		E	F		G	H	.	.	C	D
EX			A	B	C	D		E	F		G	H	I	J	C	D	
ID		A	B	C	D	E	E	F	G	G	H	I	J	C	D	E	E
IF	A	B	C	D	E	F	F	G	H	H	I	J	C	D	E	F	F
clock	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figura 4.11: Eliminação de bolhas dentro do loop.
Exemplo retirado de [Vargas, N.]

Para eliminar a bolha no final do *loop*, as instruções foram reordenadas dentro do laço, e a previsão estática do desvio foi alterada para uma previsão de que o desvio sempre ocorre. Assim obtemos a execução mostrada na figura 4.11.

Conforme mostrado na figura 4.11, a modificação da previsão estática de desvio para desvio executado e a eliminação de bolha por reordenação de instruções dentro do laço causaram uma redução de

30 no número de ciclos de *clock* necessários para executar o laço. Com um laço de 8 ciclos de *clock*, uma tabela com *n* palavras pode ser invertida em 4n ciclos de *clock*. Observe que a instrução C não faz nada e foi inserida apenas como uma reserva de espaço de tempo para que o laço pudesse começar ordenadamente. A inserção da instrução C é necessária por causa da dependência existente entre a instrução B e a instrução D.

4.8 Medidas de Desempenho

As medidas para comparar desempenhos de *pipelines* não diferem daquelas para comparação de sistemas computacionais em geral. As principais métricas utilizadas são latência, vazão, *speedup* e eficiência.

- Latência: é o tempo necessário para completar uma operação ao longo do *pipeline*. Em geral, é maior para sistemas com *pipeline* do que para sistemas monolíticos. Entretanto, é uma medida de interesse menor, pois raramente um *pipeline* seria utilizado para realizar uma única operação. Para um *pipeline* com *k*, estágios executando com um ciclo de τ, a latência para a execução de uma instrução é:

$$L_k = k \cdot \tau$$

- Vazão (throughput): expressa o número de operações executadas por unidade de tempo. Esta é uma medida mais utilizada para avaliação de *pipelines*, pois usualmente grupos de tarefas são neles executados. Para um *pipeline* com *k*, estágios execu-

tando *n* operações com um ciclo de τ, o primeiro resultado só estará disponível após um período L_k (tempo de preenchimento do *pipeline*). Após este período, as *n-1* operações restantes são entregues a intervalos de τ segundos. Portanto, para a execução do grupo de n operações, a vazão é:

$$H_k = n \,/\, [\,(k + n-1)\,\tau\,]$$

- Fatar de aceleração (Speedup): (e)expressa o ganho em velocidade na execução de um grupo de tarefas em um *pipeline* de *k* estágios comparada com a execução sequencial deste mesmo grupo de tarefas. Para a execução de *n* tarefas em um *pipeline* de *k* estágios, sendo o tempo sequencial *kn* , o fator de aceleração é:

$$S_k = nk \,/\, (\,k+n-1)$$

Quanto maior o tempo de uso consecutivo (*n*) de um *pipeline*, melhor será o fator de aceleração obtido. Quanto maior o número de estágios em um *pipeline* (*k*), maior o potencial para melhorar o fator de aceleração. Na prática, o número de estágios em *pipelines* varia entre quatro e dez, dificilmente ultrapassando 20 estágios. As principais limitações para a ampliação do número de estágios em um *pipeline* incluem custo, complexidade de controle e implementação de circuitos. *Superpipelining* é a estratégia de processamento que busca acelerar o desempenho através da utilização de grandes números de estágios simples.

- Eficiência: medida de *speedup* normalizada pelo número de estágios.

$$E_k = S_k/k = n/\,[\,k + (n - 1)\,]$$

4.9 Máquinas Superpipeline e Superescalares

O período de clock de uma máquina pipeline é determinado pelo estágio que leva maior tempo para ser executado. Uma técnica utilizada para acelerar uma máquina pipeline é subdividir os estágios mais lentos em subestágios de menor duração. Uma máquina com um número de estágios maior do que cinco é chamada de superpipeline.

A figura 4.12 ilustra uma organização superpipeline obtida pela divisão do estágio de busca de instruções (IF) em dois subestágios e do estágio de acesso à memória em três subestágios.

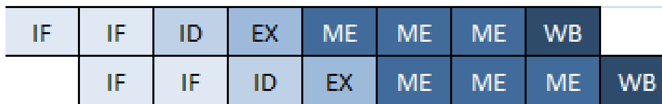


Figura 4.12: Arquitetura Superpipeline.
Figura adaptada [Vargas, N.]

Outra técnica para acelerar o processamento em máquinas pipeline é a utilização de múltiplas unidades funcionais que podem operar concorrentemente. Numa arquitetura deste tipo múltiplas instruções podem ser executadas no mesmo ciclo de clock. É necessário realizar análise de dependências para determinar se as instruções começadas ao mesmo tempo não possuem interdependências.

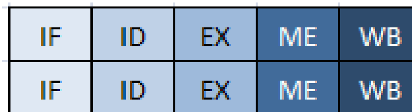


Figura 4.13: Arquitetura Superescalar.
Figura adaptada [Vargas, N.]

A figura 4.13 ilustra uma arquitetura superescalar com dois pipelines que podem operar em paralelo. Uma preocupação que surge com a implementação de máquinas superescalar é a possibilidade de

término de execução *fora de ordem*. Isto ocorreria quando suas instruções são iniciadas ao mesmo tempo em dois pipelines e uma bolha surge em um deles causando que uma instrução leve mais tempo do que a outra. Uma solução adotada para este problema consiste em criar um *buffer de reordenação* que armazena o resultado das instruções até que os resultados possam ser escritos na ordem correta.

A maioria dos processadores RISC são arquiteturas superescalar e superpipelined. Isto é, eles possuem mais de cinco estágios no pipeline devido à subdivisão de alguns estágios e eles possuem múltiplas unidades funcionais em cada estágio. Uma configuração típica é o uso de quatro unidades funcionais em cada estágio.

4.10 Processadores Superescalares

Paralelismo no nível de instrução

O pipeline melhora o desempenho ao aumentar a taxa pela qual as instruções podem ser executadas. No entanto, existem limites para um pipeline melhorar o desempenho. À medida que mais e mais estágios do pipeline são inseridos no ao processador, o atraso nos registros de pipeline necessários em cada estágio torna-se uma parte significativa da duração do ciclo, reduzindo o benefício de aumentar a profundidade do pipeline. De modo mais significativo, aumentar a profundidade do pipeline aumenta o atraso de desvios e a latência de instruções, aumentando o número de ciclos de adiamento que ocorrem entre instruções dependentes.

Devido a estas restrições, projetistas voltaram-se para o paralelismo de modo a melhorar o desempenho pela execução de várias tarefas ao mesmo tempo. Sistemas de computadores em paralelo tendem a ter uma de duas formas: vários processadores e processadores com

paralelismo no nível de instrução. Em sistemas com vários processadores tarefas relativamente grandes, como procedimentos ou iterações de laços, são executadas em paralelo. Em contraste, processadores com paralelismo no nível de instrução executam instruções individuais em paralelo.

A figura 4.13 mostra um diagrama de bloco de alto nível de um processador com paralelismo no nível da instrução. O processador contém várias unidades de execução para executar as instruções, cada uma das quais lê os seus operandos e escreve o seu resultado no conjunto único de registradores centralizados. Quando uma operação escreve o seu resultado no conjunto de registradores, aquele resultado torna-se visível para todas as unidades de execução do próximo ciclo, permitindo que operações sejam executadas em diferentes unidades a partir das operações que geraram suas entradas. Processadores com paralelismo no nível de instrução frequentemente tem um hardware complexo de vias secundárias, que transmite o resultado de cada instrução para todas as unidades de execução, de modo a reduzir o atraso entre instruções pendentes.

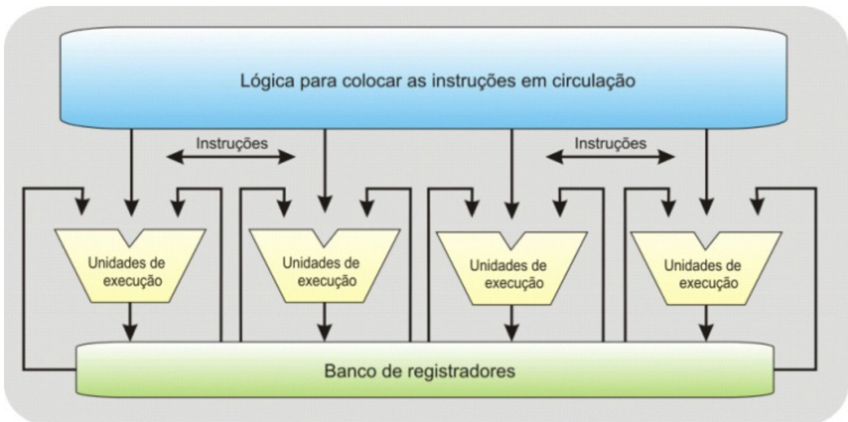


Figura 4.14: Processador com paralelismo no nível de instrução.
Figura adaptada [Carter, N.]

Na figura 4.14 todas as unidades de execução foram desenhadas de modo idêntico. Na maioria dos processadores reais, algumas ou todas as unidades de execução são capazes de executar apenas um subconjunto de instruções do processador. A divisão mais comum é entre operações de inteiros e de ponto flutuante, porque estas operações exigem um hardware muito diferente. Implementar esses dois conjuntos de hardware em unidades de execução aumenta o número de instruções que podem ser executadas simultaneamente sem aumentar significativamente a quantidade de hardware exigido.

Processadores com paralelismo no nível da instrução exploram o fato que muitas das instruções em um programa sequencial não dependem das instruções que a precedem imediatamente no programa. Por exemplo, considere o fragmento de programa abaixo. As instruções 1, 3 e 5 são dependentes umas das outras porque a instrução 1 gera valores que são utilizados pela instrução 3 a qual gera resultado que é utilizado na instrução 5. As instruções 2 e 5 não utilizam os resultados de quaisquer outras instruções no fragmento e não geram quaisquer resultados que sejam utilizados neste fragmento. Estas dependências exigem que as instruções 1, 3 e 5 sejam executadas em ordem para gerar o resultado correto, mas as instruções 2 e 4 podem ser executadas antes, depois, ou em paralelo com quaisquer outras instruções, sem modificar os resultados do fragmento de programa.

```
1: LD r1,(r2)
2: ADD r5, r6, r7
3: SUB r4, r1, r4
4: MUL r8, r9, r10
5: ST (r11), r4
```

```
Ciclo 1:   LD r1,(r2)        ADD r5, r6, r7
Ciclo 2:   SUB r4, r1, r4    MUL r8, r9, r10
Ciclo 3:   ST (r11), r4
```

Exemplo retirado de [Carter, N.]

Este exemplo ilustra tanto os pontos favoráveis quanto os desfavoráveis do paralelismo no nível de instrução. Processadores PNI podem, ao executar instruções em paralelo, atingir aumentos de velocidade significativos em uma ampla variedade de programas, mas a melhora máxima do seu desempenho está limitada pelas dependências de instruções.

Limitações

O desempenho de processadores PNI está limitado pela quantidade de paralelismo no nível da instrução que o compilador e o hardware podem localizar no programa. O paralelismo no nível de instrução é limitado por diversos fatores: dependência de dados, dependência de nomes e desvios. Além disso, uma determinada capacidade do processador de explorar o paralelismo no nível de instrução pode ser limitada pelo número e pelo tipo de unidades de execução que estão presentes e por restrições aos tipos de instrução do programa que podem ser executadas em paralelo.

Desvios limitam o paralelismo no nível da instrução porque o processador não sabe quais instruções serão executadas depois de um desvio, até que ele seja completado. Isso exige que o processador espere a conclusão do desvio antes que outras instruções sejam executadas.

Exemplo: Considere o seguinte fragmento de programa:

```
ADD   r1, r2, r3
LD    r4, (r5)
SUB   r7, r1, r9
MUL   r5, r4, r4
SUB   r1, r12, r10
ST    (r13), r14
OR    r15, r14, r12
```

Quanto tempo este programa demoraria para ser executado por um processador que permita duas instruções executando simultaneamente? Como seria se o processador permitisse quatro instruções simultaneamente? Assuma que o processador pode executar instruções em qualquer ordem que não viole a dependência de dados, que todas as instruções têm latência de um ciclo e que todas as unidades de execução do processador podem executar todas as instruções no fragmento.

Solução: *em um processador que permita duas instruções realizadas simultaneamente, este programa demoraria quatro ciclos para ser posto em execução. Um exemplo seria:*

```
Ciclo 01: ADD r1, r2, r3      LD r4, (r5)
Ciclo 02: SUB r7, r1, r9      MUL r5, r4, r4
Ciclo 03: SUB r1, r12, r10    ST (r13), r14
Ciclo 04: OR r15, r14, r12
```

Se o processador pudesse executar quatro instruções simultaneamente, o programa poderia ser posto em circulação em dois ciclos. Um exemplo seria:

```
Ciclo 01:   ADD r1, r2, r3    LD r4, (r5) ST (r13), r14     OR r15, r14, r12
Ciclo 02:   SUB r7, r1, r9    MUL r5, r4, r4     SUB r1, r12, r10
```

Note que, independente do número de instruções que o processador possa executar simultaneamente, não é possível colocar o programa em circulação em apenas um ciclo por causa das dependências.

Por exemplo, as instruções SUB r7, r1, r9 e SUB r1, r12, r10.

Exemplo retirado de [Carter, N.]

Processadores Superescalares

Processadores superescalares baseiam-se no hardware para extrair paralelismo no nível de instrução de programas sequenciais. Durante cada ciclo, a lógica de circulação de instruções de um processador superescalar examina as instruções em um programa sequencial para determinar quais instruções podem ser postas em circulação naquele ciclo. Se existir suficiente paralelismo no nível da instrução dentro de um programa, um processador superescalar pode executar uma instrução por unidade de execução por ciclo, mesmo se o programa foi compilado originalmente para um processador que executa apenas uma instrução por ciclo.

Esta é uma das maiores vantagens de processadores superescalares e é o motivo pelo qual virtualmente todas as CPUs de PCs e de estações de trabalho são processadores superescalares. Processadores superescalares podem executar programas que foram compilados originalmente para processadores puramente sequenciais e podem atingir desempenho melhor nestes programas do que processadores que não são capazes de explorar o paralelismo no nível da instrução.

A capacidade que processadores superescalares tem de explorar o paralelismo no nível da instrução em programas sequenciais não significa que os compiladores sejam irrelevantes para sistemas construídos com processadores superescalares. De fato, bons compiladores são ainda mais críticos para o desempenho de computadores superescalares do que eles são para processadores puramente sequenciais. Processadores superescalares podem examinar em um dado momento apenas uma pequena janela de instruções de um programa para determinar quais instruções podem ser executadas em paralelo. Se um compilador for capaz de organizar as instruções de um programa de modo que grande

quantidade de instruções ocorra dentro desta janela, um processador superescalar será capaz de atingir um bom desempenho no programa.

Execução Em-Ordem e Fora-De-Ordem

Uma diferença significativa de complexidade e desempenho no projeto de processadores superescalar é que o processador executa as instruções na ordem em que elas aparecem no programa e que ele as executa em qualquer ordem, desde que não mude o resultado do programa. A execução fora-de-ordem pode fornecer um desempenho muito melhor que a execução em-ordem, mas exige um hardware muito mais complexo para ser implementada.

Processadores Em-Ordem

Em processadores superescalares em-ordem, o tempo para colocar um programa em circulação pode ser determinado ao se percorrer sequencialmente o código para determinar quando cada instrução pode ser posta em circulação. De modo semelhante á técnica utilizada para processadores com pipeline que executam apenas uma instrução por ciclo. A diferença entre processadores superescalares em-ordem e processadores com pipeline que não são superescalares é que um processador superescalar pode colocar uma instrução em circulação no mesmo ciclo que a instrução anterior no programa. Se as dependências de dados permitirem e desde que o número de instruções colocada em cada ciclo não exceda o número de instruções, o processador pode executar as instruções simultaneamente.

Exemplo:

Quanto tempo a seguinte sequência de instruções deveria ser executada em um processador em-ordem, com duas unidades de execução,

cada uma das quais podendo executar qualquer instrução? As operações de carga têm latência de dois ciclos e as demais operações têm latência de um ciclo.

```
LD    r1, (r2)
ADD   r3, r1, r4
SUB   r5, r6, r7
MUL   r8, r9, r10
```

Solução: *assumindo que a operação LD é posta em circulação no ciclo n, a ADD não pode ser posta em circulação até n+2, porque ela é dependente de LD. A SUB é independente de ADD e LD, de modo que também não pode ser posta em circulação no ciclo n+2 devido a necessidade de o processador gastar um ciclo para carregar as instruções em-ordem. A MUL também é independente de todas as instruções anteriores, mas precisa de n+3 para ser posta em circulação, porque o processador só pode colocar duas instruções em circulação. Portanto, demora quatro ciclos para que todas as instruções do programa sejam postas em circulação.*

Exemplo retirado de [Carter, N.]

Processadores Fora-de-Ordem

Determinar o tempo de circulação de uma sequência de instruções em um processador fora-de-ordem é mais difícil que determinar o tempo de circulação em um processador em-ordem porque há muitas possibilidades de ordens nas quais as instruções podem ser executadas. Geralmente a melhor abordagem é começar pela sequência de instruções para localizar as dependências entre as instruções. Uma vez que as dependências entre as instruções sejam entendidas, elas podem ser designadas para ciclos de circulação de modo a minimizar o atraso entre a execução da primeira e da última instrução na sequência.

O esforço para encontrar a melhor ordem possível de um conjunto de instruções cresce exponencialmente com o número de instruções no conjunto, uma vez que todas as ordens possíveis devem ser potencialmente consideradas. Assim, assumiremos que a lógica de instruções em um processador superescalar estabelece algumas restrições sobre a ordem na qual as instruções circulam, de modo a simplificar a lógica de circulação das instruções. A suposição é que o processador tentará executar uma instrução tão logo as dependências de um programa permitam. Se existirem mais instruções que unidades de execução, o processador privilegiará as instruções que ocorrem primeiro no programa, mesmo que eventualmente houvesse uma ordem que implicasse em um menor tempo de execução.

Com esta suposição para a circulação de instruções, torna-se muito mais fácil encontrar o tempo de circulação de uma sequência de instruções em um processador fora-de-ordem. Iniciando com a primeira instrução de uma sequência, seguimos instrução por instrução, atribuindo cada uma ao primeiro ciclo no qual suas entradas já estão disponíveis. Deve-se respeitar as limitações de que o número de instruções que circulam não seja maior que o número máximo de instruções simultâneas que o processador suporta e que não ultrapasse o número de unidades de execução.

Exemplo: *em quanto tempo a seguinte sequência de instruções deveria ser executada em um processador fora-de-ordem com duas unidades de execução, cada uma das quais podendo executar qualquer instrução? As operações de carga têm latência de dois ciclos e as demais operações têm latência de um ciclo.*

```
LD      r1, (r2)
ADD     r3, r1, r4
SUB     r5, r6, r7
MUL     r8, r9, r10
```

<u>Solução:</u>

A única dependência nesta sequência é entre as instruções LD e ADD. Por causa dessa dependência, a instrução ADD precisa entrar em circulação pelo menos dois ciclos antes de LD. A SUB e a MUL poderiam ser postas em circulação no mesmo ciclo que a LD. Assumindo que SUB e LD são postas em circulação no ciclo n, a MUL é posta em circulação no ciclo n+1 e ADD posta em circulação no ciclo n+2, dando um tempo de circulação de três ciclos para este programa.

Exemplo retirado de [Carter, N.]

EXERCÍCIOS

1. Explique o princípio por trás da técnica de pipeline e mostre porquê melhoram o desempenho.

2. Explique a estrutura de execução de uma máquina pipeline.

3. Dado um processador sem pipeline, com duração de ciclos de 10 ns e latches com 0,5 ns de latência, quais as durações de ciclos das versões do processador com pipeline de 2, 4, 8 e 16 estágios? Qual a latência de cada uma das versões? Que conclusões podemos tirar sobre isso?

4. O que são bolhas em pipelines? Quais situações as geram?

5. Explique as técnicas de previsão de desvios estáticas e dinâmicas.

6. Mostre três métricas utilizadas para medir desempenho de processadores.

7. O que é paralelismo no nível de instrução? Como os processadores exploram para melhorar o desempenho?

8. Comente sobre as limitações do paralelismo no nível de instrução.

9. Quanto tempo demora para que as seguintes instruções sejam executadas por um processador superescalar em ordem, com duas unidades de execução, em que qualquer unidade de execução pode executar todas as instruções, as operações de carga têm latência de 3 ciclos e

todas as outras têm latência de 2 ciclos? Assuma que o processador tem pipeline de 6 estágios.

```
LD r4, (r5)
LD r7, (r8)
ADD r9, r4, r7
LD r10, (r11)
MUL r12, r13, r14
SUB r2, r3, r1
ST (r2), r15
MUL (r22), r23
ST (r22), r23
ST (r24), r21
```

10. Repita a questão anterior considerando um processador fora de ordem.

5. RISC E CISC

5.1 Introdução

Antes da década de 1980, havia um grande foco na redução do intervalo semântico entre as linguagens utilizadas para programar computadores e as linguagens de máquina. Acreditava-se que tornar as linguagens de máquina mais parecidas às linguagens de alto nível resultaria num melhor desempenho, pela redução do número de instruções exigidas para implementar um programa e tornaria mais fácil compilar um programa em linguagem de alto nível para a linguagem de máquina. O resultado final disto foi o projeto de conjuntos de instruções que continhas instruções muito complexas.

À medida que a tecnologia de compiladores era aperfeiçoada, os pesquisadores começaram a questionar estes sistemas com instruções complexas, conhecidos como computadores com conjunto de instruções complexas CISC (*ComplexInstruction Set Computer*). Os mesmos forneciam desempenho melhor que sistemas baseados em conjuntos de instruções mais simples. Esta segunda classe ficou conhecida como computadores de conjunto de instruções reduzidas RISC (*ReducedInstruction Set Computer*).

O argumento principal a favor dos computadores CISC é que esses geralmente exigem menos instruções que os computadores RISC para executar uma dada operação, de modo que um computador CISC tem um desempenho melhor que um computador RISC que executa instruções a mesma taxa. Além disso, programas escritos para arquiteturas CISC tendem a tomar menos espaço na memória que programas escritos para a arquitetura RISC. O principal argumento a favor dos computadores RISC é que os seus conjuntos de instruções mais

simples frequentemente permitem que eles sejam implementados com frequência de relógio mais altas, permitindo executar mais instruções na mesma quantidade de tempo. Com frequência de relógio maior, um processador RISC permite que ele execute programas em menos tempo do que um processador CISC para executar os programas.

Durante a década de 80 e início dos anos 90, houve muita controvérsia na comunidade de arquitetura de computadores no que diz respeito a qual das duas abordagens era a melhor e dependendo do ponto de vista, qualquer uma das duas podia ser considerada vencedora. Ao longo dos últimos 20 anos, tem havido certa convergência entre às arquiteturas, tornando difícil determinar se uma arquitetura é RISC ou CISC. As arquiteturas RISC incorporam algumas instruções complexas, mas úteis as arquiteturas CISC que, por sua vez, abandonaram instruções complexas que não eram utilizadas com frequência suficiente para justificar a sua implementação.

Uma diferença clara entre as arquiteturas CISC e RISC refere-se ao acesso a memória. Em muitas arquiteturas CISC, instruções aritméticas e outras podem ler as suas entradas ou escrever as suas saídas diretamente na memória, em vez de fazê-lo sobre os registradores. Por exemplo, uma arquitetura CISC pode permitir uma operação ADD na forma ADD(r1)(r2)(r3), a qual onde os parênteses em volta do nome do registro indicam que o registro contém o endereço de memória onde um operando pode ser encontrado, ou o resultado pode ser escrito. A mesma operação para ser realizada nas arquiteturas RISC, que utilizam o modelo carga-armazenamento ou *load-store*, necessita da seguinte seqüência de instruções:

```
LD r4, (r2)        :. Carregar r2 em r4
LD r5,(r3)         :. Carregar r3 em r5
ADD r6,r4,r5       :. Somar r4 e r5 e guardar o valor em r6
ST (r1), r6        :. Gravar r6 em r1
```

Exemplo retirado de [Carter, N.]

Este exemplo mostra que uma arquitetura RISC pode exigir muito mais operações para implementar uma função do que uma arquitetura CISC, se bem que o exemplo é bem extremo. Este exemplo também mostra que arquiteturas RISC normalmente utilizam mais registradores para implementar uma função que as arquiteturas CISC, uma vez que todas as entradas de uma instrução precisam ser carregadas em um bando de registradores antes que a instrução possa ser executada. No entanto os processadores RISC têm a vantagem de "dividir" uma operação CISC complexa em várias operações RISC, permitindo ao compilador organizar as operações RISC para um desempenho melhor.

Atualmente, vemos processadores híbridos, que são essencialmente processadores CISC, porém que possuem internamente núcleos RISC. Assim, a parte CISC do processador pode cuidar das instruções mais complexas, enquanto o núcleo RISC pode cuidar das mais simples, as quais são mais rápidas. Parece que o futuro nos reserva uma fusão destas duas tecnologias. Um bom exemplo de processador híbrido é o Pentium Pro.

5.2 CISC

No inicio dos anos 70, quer porque os compiladores eram mui-

to pobres e pouco robustos, quer porque a memória era lenta e cara causando sérias limitações no tamanho do código, previu-se uma crise no software. O hardware era cada vez mais barato e o software cada vez mais caro. Um grande número de investigadores e projetistas defendiam que a única maneira de contornar os grandes problemas que se avizinhavam era mudar a complexidade do (cada vez mais caro) software e transportá-la para o (cada vez mais barato) hardware.

Se houvesse função mais comum que o programador tivesse de escrever várias vezes em um programa, por que não implementar essas funções em hardware? Afinal de contas o hardware era barato e o tempo do programador não. Esta ideia de mover o fardo da complexidade do software para o hardware foi impulsionadora por trás da filosofia CISC, e quase tudo o que um verdadeiro CISC faz tem este objetivo. Alguns pesquisadores sugeriram que uma maneira de tornar o trabalho dos programadores mais fácil seria fazer com que o código *assembly* se parecesse mais com o código das linguagens de alto nível (C ou Pascal).

Os mais extremistas falavam já de uma arquitetura de computação baseada numa linguagem de alto nível. Este tipo de arquitetura era CISC levado ao extremo. A sua motivação primária era reduzir o custo global do sistema fazendo computadores mais fáceis de programar. Ao simplificar o trabalho dos programadores, pensava-se que os custos seriam mantidos em um nível razoável.

Uma lista das principais razões para se promover este tipo de arquitetura é:

- reduzir as dificuldades de escrita de compiladores;
- reduzir o custo global do sistema;

- reduzir os custos de desenvolvimento de *software*;
- reduzir a diferença semântica entre linguagens de programação e máquina;
- fazer com que os programas escritos em linguagens de alto nível corressem mais eficientemente;
- melhorar a compactação do código;
- facilitar a detecção e correção de erros.

Resumindo, se uma instrução complexa escrita numa linguagem de alto nível fosse traduzida exatamente em uma instrução *assembly*, então:

- Os compiladores seriam mais fáceis de escrever. Isto pouparia tempo e esforço para os programadores, reduzindo, assim, os custos de desenvolvimento de software;

- O código seria mais compacto, o que permitiria poupar em memória, reduzindo o custo global do hardware do sistema;

- Seria mais fácil fazer a detecção e correção de erros o que, de novo, permitiria baixar os custos de desenvolvimento de software e de manutenção.

Até este momento, centramos a atenção nas vantagens econômicas da arquitetura CISC, ignorando a questão do desempenho. A abordagem utilizada neste tipo de arquitetura para melhorar o desempenho das máquinas CISC foi, conforme já referido, transferir a complexidade do software para o hardware. Para melhor se compre-

ender como é que este tipo de abordagem afeta o desempenho vamos analisar um pouco a seguinte equação - equação do desempenho de um processador – uma métrica normalmente utilizada para avaliar o desempenho de um sistema de computação:

$$\frac{Tempo}{Programa} = \frac{instruções}{programa} x \frac{ciclos}{instrução} x \frac{tempo}{ciclo}$$

Melhorar o desempenho significa reduzir o termo à esquerda do sinal de igualdade, porque quanto menor for o tempo que um programa demora a ser executado, melhor será o desempenho do sistema. As máquinas CISC tentam atingir este objetivo reduzindo o primeiro termo à direita do sinal de igualdade, isto é, o número de instruções por programa. Os pesquisadores pensaram que ao reduzir o número de instruções que a máquina executa para completar uma determinada tarefa, poder-se-ia reduzir o tempo que ela necessita para completar essa mesma tarefa, aumentando, dessa forma, o seu desempenho.

Destarte, ao reduzir o tamanho dos programas conseguiam-se dois propósitos: por um lado era necessária uma menor quantidade de memória para armazenar o código; e por outro o tempo de execução era, também, diminuído, pois havia menos linhas de código para executar.

Além de implementar todo o tipo de instruções que faziam um variado número de tarefas como copiar *strings* ou converter valores, havia outra táctica que os projetistas utilizavam para reduzir o tamanho do código e a sua complexidade: os modos de endereçamento complexos. A figura 5.1 ilustra um esquema de armazenamento para um computador genérico. Se quiséssemos multiplicar dois números, teríamos primeiro que carregar cada um dos operandos de uma locali-

zação na memória para um dos registradores.

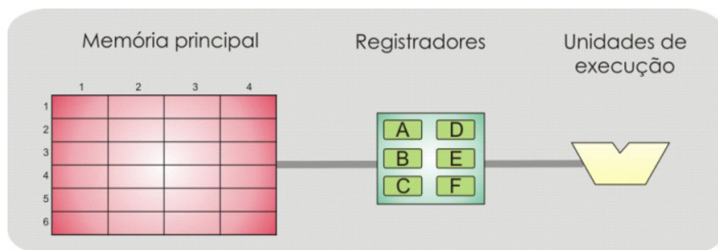


Figura 5.1: Esquema de armazenamento.

Vejamos o seguinte exemplo, meramente ilustrativo, da multiplicação de dois números armazenados em memória. Uma vez carregados nos registradores, os operandos poderiam ser multiplicados pela unidade de execução. Utilizaríamos as seguintes linhas de código para multiplicar o conteúdo das posições de memória [2:3] e [5:2] e armazenar o resultado na posição [2:3]:

```
MOV [A, 2:3]
MOV [B, 5:2]
MUL [A, B]
MOV [2:3, A]
```

Este exemplo de código explicita os passos que têm de ser dados pelo processador para multiplicar os conteúdos de duas posições de memória. Há que carregar os dois registros com o conteúdo da memória principal, multiplicar os dois números e guardar o resultado de novo na memória principal. Se quiséssemos tornar o *assembly* menos complicado e mais compacto, poderíamos modificar a arquitetura por forma a realizar a operação descrita com uma instrução apenas. Para atingir esse objetivo, a instrução MUL teria que ser alterada por forma a aceitar como parâmetros duas posições de memória:

MUL [2:3, 5:2]

Evoluir de quatro instruções para apenas uma é uma grande economia. Apesar da "nova arquitetura" ainda ter que carregar o conteúdo das duas posições de memória para os registradores, multiplicá-los e voltar a armazenar na memória o resultado (não há como contornar isso). Todas essas operações de baixo nível são feitas em hardware e são invisíveis para o programador. Este é um exemplo de endereçamento complexo. Uma instrução *assembly*, na realidade, leva em consideração uma série complexa de operações. Uma vez mais, isto é um exemplo da filosofia CISC de transferir a complexidade do software para o hardware.

Outra característica das máquinas CISC é a utilização de microcódigo. A microprogramação é uma das características primordiais que permite aos projetistas a implementação de instruções complexas em hardware. Para melhor compreender o que é a microprogramação, vamos considerar, resumidamente, a execução direta.

Usando execução direta, a máquina carrega a instrução da memória, descodifica-a e envia-a para a unidade de execução. Esta unidade de execução "pega" na instrução a sua entrada e ativa alguns circuitos. Se, por exemplo, a máquina carrega a instrução de ADD com operadores em ponto flutuante e a fornece à unidade de execução, existe um circuito que a carrega e direciona as unidades de execução para garantir que todas as operações de deslocamento, adição e normalização sejam executadas corretamente. A execução direta é, na realidade, o que se poderia esperar que acontecesse dentro de um computador se não houvesse conhecimento do microcódigo.

A principal vantagem da execução direta é que ela é rápida. Não existe qualquer tipo de abstração ou tradução extras; a máquina apenas

descodifica e executa as instruções em hardware. O seu maior problema é que pode ocupar algum espaço. De fato, se todas as instruções têm que ter um circuito que as execute, então, quanto maior for o número de instruções, maior vai ser o espaço ocupado pela unidade de execução. Por isso, executar diretamente as instruções não **é** uma boa abordagem para o projeto de uma máquina CISC. Até porque os recursos (transistores) disponíveis eram bastante escassos.

Chegamos assim à microprogramação. Com esta, é possível dizer que quase temos um miniprocessador dentro do processador. A unidade de execução é um processador de microcódigo que executa microinstruções. Os projetistas usam estas microinstruções para escrever microprogramas que são armazenados numa memória de controle especial.

Quando uma instrução normal de um programa é carregada da memória, descodificada e entregue ao processador de microcódigo, este último executa a sub-rotina de microcódigo adequada. Esta sub-rotina "diz" às várias unidades funcionais o que fazer e como fazer.

Com a evolução da tecnologia, o microcódigo está cada vez mais rápido. Os processadores de microcódigo nos processadores modernos conseguem velocidades da ordem de 95% em relação à execução direta. Como o microcódigo está cada vez melhor, faz-se cada vez mais sentido transferir funcionalidades do *software* para o *hardware*. Dessa forma, os conjuntos de instruções cresceram rapidamente e o número médio de instruções por programa decresceu.

Contudo, à medida que os microprogramas cresceram para fazer frente ao crescente número de instruções, alguns problemas começaram a surgir. Para manter um bom desempenho, o microcódigo tinha que ser altamente otimizado, eficiente e bastante compacto para

que os custos de memória não começassem a crescer demasiadamente. Como os microprogramas eram, agora, tão grandes, era mais difícil testá-los e detectar e corrigir erros. Como resultado, o microcódigo incluído nas máquinas que vinham para o mercado tinha, algumas vezes, erros e tinha que ser corrigidos. Foram estas dificuldades de implementação do microcódigo que levaram os pesquisadores a questionar se a implementação de todas estas instruções complexas e elaboradas em microcódigo seria realmente o melhor caminho para fazer uso dos limitados recursos (transistores) existentes.

5.3 RISC

Como já foi referido, muitas das implementações da arquitetura CISC eram tão complexas que eram distribuídas por vários *chips*. Esta situação não era, por razões óbvias, ideal. Era necessária uma solução em um único *chip*, uma solução que fizesse melhor uso dos escassos recursos disponibilizados (transistores). No entanto, para que todo um processador coubesse num só *chip*, algumas das suas funcionalidades teriam que ser deixadas de fora. Nesse intuito, realizaram-se estudos destinados a descobrir que tipos de situações ocorrem mais frequentemente na execução de aplicações. A ideia era descobrir em que tipo de tarefas o processador passava mais tempo e otimizar essas mesmas tarefas. Se tivessem que ser feitos compromissos, estes deviam ser feitos em favor da velocidade de execução das tarefas nas quais o processador passa mais tempo a trabalhar, ainda que isso pudesse atrasar outras tarefas não tão frequentes.

Esta abordagem quantitativa de fazer mais rápidas as tarefas mais comuns provocou a inversão da filosofia iniciada pelos CISC e

a complexidade teve que ser retirada do hardware e ser passada para o software. A memória estava mais barata e os compiladores eram cada vez mais eficientes, por isso muitas das razões que conduziram os projetistas a "complicar" o conjunto de instruções deixaram de existir. Os pesquisadores diziam que o suporte para as linguagens de alto nível poderia ser mais eficiente se fosse implementado em software, gastar recursos (transistores) preciosos para suportar as linguagens de alto nível em hardware era um desperdício. Esses recursos poderiam ser utilizados em outras tecnologias para melhorar o desempenho.

Quando os investigadores tiveram que decidir quais as funcionalidades teriam de ser retiradas, o suporte para o microcódigo foi o primeiro a sair, e com ele saíram também um grupo de instruções complexas que tornava o trabalho dos compiladores e dos programadores mais fácil. A ideia era que quase ninguém utilizava aquelas instruções tão complexas. Ao compilar o código, os compiladores preteriam este tipo de instruções em favor da geração de um conjunto de instruções mais simples que realizassem a mesma tarefa.

O que os pesquisadores concluíram dos estudos realizados foi que um pequeno conjunto de instruções estava fazendo a maioria do trabalho. Aquelas instruções que raramente eram usadas poderiam ser eliminadas sem que houvesse perda de qualquer funcionalidade.

Esta ideia da redução do conjunto de instruções, deixando de fora todas as instruções que não fossem absolutamente necessárias, substituindo as instruções mais complexas por conjuntos de instruções mais simples, foi o que esteve na origem do termo *ReducedInstruction Set Compute - RISC*. Ao incluir apenas um pequeno e criteriosamente escolhido grupo de instruções numa máquina, poder-se-ia deixar de fora o suporte do microcódigo e passar a usar a execução direta.

Não só o número de instruções foi reduzido, mas também o tamanho das mesmas. Foi decidido que todas as instruções RISC deveriam, sempre que possível, demorar apenas um ciclo de relógio a terminar a sua execução. A razão por trás desta decisão foi baseada em algumas observações feitas pelos pesquisadores. Em primeiro lugar, percebe-se que tudo o que poderia ser feito usando as instruções de microcódigo, também poderia ser feito com pequenas e rápidas instruções de *assembly*. A memória que estava sendo usada para armazenar o microcódigo poderia simplesmente ser usada para armazenar o *assembler*, assim, a necessidade de microcódigo seria pura e simplesmente eliminada. É por esta razão que muitas das instruções de uma máquina RISC correspondem a microinstruções numa máquina CISC. A segunda razão que levou o formato a ser uniformizado e demorar apenas um ciclo de relógio foi a observação de que a implementação do *pipelining* só é realmente viável se não tiver que lidar com instruções de diferentes graus de complexidade. O pipelining permite a execução de várias instruções em paralelo. Dessa forma, uma máquina que o suporte consegue reduzir drasticamente o número médio de ciclos por instrução (CPI – Cycles Per Instruction). Baixar o número médio de ciclos que as instruções necessitam para terminar a sua execução é uma maneira efetiva de baixar o tempo total que é necessário à execução de um programa.

Relembremos de novo a equação do desempenho. Os projetistas deste tipo de arquitetura tentaram reduzir o tempo por programa reduzindo o segundo termo à direita do sinal de igualdade (ciclos/instrução), permitindo que o primeiro termo (instruções/programa) aumentasse ligeiramente. Pensava-se que uma redução no número de ciclos por instrução, alcançada à custa da redução do conjunto de ins-

truções, a introdução da técnica de *pipelining*e outras funcionalidades (das quais já falaremos) compensariam largamente o aumento do número de instruções por programa. Verificou-se que esta filosofia estava correta.

Além da técnica de *pipelining*, houve duas inovações importantes que permitiram o decréscimo do número de ciclos por instrução mantendo o aumento do tamanho do código em um nível mínimo: a eliminação dos modos de endereçamento complexos e o aumento do número de registradores internos do processador. Nas arquiteturas RISC existem apenas operações registrador-registrador e apenas as instruções LOAD e STORE podem acessar a memória. Poder-se-ia pensar que o uso de LOAD's e STORE's em vez de uma única instrução que operasse na memória iria aumentar o número de instruções de tal modo que o espaço necessário em memória e o desempenho do sistema seriam afetados. Como posteriormente se verificou, existem algumas razões que fazem com que este pensamento não seja correto.

Verificou-se que mais de 80% dos operandos que apareciam em um programa eram variáveis escalares locais. Isto significa que, se fossem adicionados múltiplos bancos de registradores à arquitetura, estas variáveis locais poderiam ficar armazenadas nos registradores, evitando ter que ir à memória todas as vezes que fosse necessária alguma delas. Deste modo, sempre que uma sub-rotina é chamada, todas as variáveis locais são carregadas para um banco de registradores, sendo aí mantidas conforme as necessidades.

Esta separação das instruções LOAD e STORE de todas as outras permite ao compilador "programar" uma operação imediatamente a seguir ao LOAD (por exemplo). Assim, enquanto o processador espera alguns ciclos para os dados serem carregados para o(s) registra-

dor(es), pode executar outra tarefa, em vez de ficar parado à espera. Algumas máquinas CISC também tiram partido desta demora nos acessos à memória, mas esta funcionalidade tem que ser implementada em microcódigo.

Como se pode ver da discussão acima, o papel do compilador no controle dos acessos à memória é bastante diferente nas máquinas RISC em relação às máquinas CISC. Na arquitetura RISC, o papel do compilador é muito mais proeminente. O sucesso deste tipo de arquitetura depende fortemente da "inteligência" e nível de otimização dos compiladores que se "aproveitam" da maior responsabilidade que lhes é concedida para poderem gerar código mais otimizado. Este fato de transferir a competência da otimização do código do hardware para o compilador foi um dos mais importantes avanços da arquitetura RISC.

Como o hardware era, agora, mais simples, isto significava que o software tinha que absorver alguma das complexidadesexaminando agressivamente o código e fazendo um uso prudente do pequeno conjunto de instruções e grande número de registradores típicos desta arquitetura. Dessa maneira, as máquinas RISC dedicavam os seus limitados recursos (transistores) para providenciar um ambiente em que o código poderia ser executado tão depressa quanto o possível, confiando no compilador para fazer o código compacto e otimizado.

Em resumo, existe um conjunto de características que permite uma definição de arquitetura básica RISC. São elas:

Execução de uma instrução por ciclo de clock:

Esta é a característica mais importante das máquinas RISC (e que as distingue das máquinas CISC). Como consequência disto,

qualquer operação que não possa ser completada em um ciclo não pode ser incluída no conjunto de instruções. Assim, muitas máquinas RISC não possuem instruções de multiplicação e divisão, executando estas operações através de sequências de somas e deslocamentos criadas em tempo de compilação ou através de procedimentos de bibliotecas.

Arquitetura LOAD/STORE

Dada a condição anterior de que todas as instruções devem ser executadas em um ciclo de *clock*, cria-se um problema com as instruções que acessam a memória, pois este tipo de operação leva muito tempo. A solução adotada é que todas as instruções comuns (ADD, MOV, AND) devem ter apenas registradores como operandos, ou seja, apenas o endereçamento de registrador é permitido. Entretanto, algumas instruções têm que referenciar a memória, de modo que duas instruções especiais, *LOAD* e *STORE*, são adicionadas a esta arquitetura. Estas duas instruções são a única forma de ler ou escrever dados na memória.

Uso intenso de Pipelining

Proibir que as instruções comuns acessem a memória não resolve o problema de como fazer com que as instruções *LOAD* e *STORE* operem em um ciclo. A solução neste caso é alterar ligeiramente o objetivo: ao invés de requerer que toda instrução deva ser executada em um ciclo, deve-se buscar a capacidade de iniciar uma nova instrução a cada ciclo, sem levar em conta quando ela é concluída. Dessa forma, se em N ciclos consegue-se iniciar N instruções, na média se obtém a execução de uma instrução por ciclo. Para atingir este objetivo modificado, todas as máquinas RISC possuem *pipeline*. No entanto, alguns

cuidados devem ser tomados com o uso do *pipeline*. Por exemplo, o compilador deve garantir que qualquer instrução que venha depois de um LOAD não utilize o dado que está sendo buscado da memória.

Unidade de Controle Implementada por hardware

Em uma máquina RISC, as instruções são executadas diretamente pelo *hardware*, eliminando a necessidade de qualquer microprograma na UC. Eliminar a necessidade de decodificação das instruções em microinstruções é o segredo da velocidade das máquinas RISC. Se uma instrução complexa for realizada em uma máquina CISC utilizando 10 microinstruções de 0,1s cada, conclui-se que tal instrução leva 1s para ser executada. Em uma máquina RISC, para realizar a mesma operação seria necessário utilizar 10 instruções RISC, e se cada uma levar 0,1s para ser executada, se gasta então o mesmo tempo que a máquina CISC. A única desvantagem da máquina RISC neste caso é a ocupação um pouco superior de memória, visto que os programas tendem a ter mais instruções.

Instruções de Formato Fixo

Como em uma máquina RISC a UC é implementada através de hardware, os bits individuais de cada instrução são utilizados como entradas no bloco decodificador de instruções. Neste caso, não faz sentido utilizar instruções de tamanho variável, pois não existe microprograma embutido capaz de retirar bytes da fila de instruções e analisá-los um de cada vez por software.

Conjunto de Instruções e Modos de Endereçamentos Reduzidos

Este é o motivo pelo qual as máquinas RISC receberam tal nome.

No entanto, não existe nem uma objeção em criar muitas instruções, desde que cada instrução seja executada em um ciclo. Na prática, a limitação na quantidade de instruções recai sobre a complexidade do decodificador de instruções, que cresce gradativamente conforme o número de instruções, aumentando inclusive a área ocupada da pastilha. Por razões de velocidade e complexidade, não é desejável ter mais do que um número mínimo de modos de endereçamento.

Múltiplos Conjuntos de Registradores

Por não possuir nem um microprograma, grande parte da área da pastilha é liberada para outros propósitos. Muitas máquinas RISC utilizam este espaço para implementar um grande número de registradores de CPU que são utilizados para diminuir a quantidade de acessos à memória (LOAD e STORE).

Necessidade de Compiladores Complexos

Ao manter o hardware tão simples quanto possível de modo a obter a maior velocidade possível, paga-se o preço de tornar o compilador consideravelmente mais complicado. Boa parte da complexidade do compilador reside na grande distância semântica existente entre as linguagens de alto nível e o conjunto de instruções reduzido das máquinas RISC. Além disso, o compilador tem que lidar com características difíceis, como cargas atrasadas e o gerenciamento do grande número de registradores.

5.4 RISC vs CISC – Comentários

Armazenamento e Memória:

A memória, hoje em dia, é rápida e barata; qualquer pessoa que tenha instalado recentemente um programada Microsoft sabe que muitas das companhias que desenvolvem software já não têm em consideração aslimitações de memória. Assim, as preocupações com o tamanho do código que deram origem ao vasto conjuntode instruções da arquitetura CISC já não existem. Defato, os processadores da era pós-RISC têm conjuntos de instruções cada vez maiores de um tamanho e diversidade sem precedentes, e ninguém pensa duas vezes no efeito que isso provoca no uso da memória.

Compiladores:

O desenvolvimento dos compiladores sofreu um tremendo avanço nos últimos anos. De fato, chegou a um ponto tal que a próxima geração de arquiteturas (como o IA-64 ou Merced da Intel) depende apenas do compilador para ordenar as instruções tendo em vista a máxima taxa de instruções executadas.

Os compiladores RISC tentam manter os operandos em registradores de forma a poderem usar simples instruções registrador-registrador. Os compiladores tradicionais, por outro lado, tentam descobrir o modo de endereçamento ideal e o menor formato de instrução para fazerem os acessos à memória. Em geral, os programadores de compiladores RISC preferem o modelo de execução registrador-registrador de forma que os compiladores possam manter os operandos que vão ser reutilizados em registradores, em vez de repetirem os acessos à memória cada vez que for necessário um operando. Usam,

por isso, LOAD's e STORE's para acessar a memória para que os operandos não sejam implicitamente rejeitados após terminada aexecução de uma determinada instrução, como acontece nas arquiteturas que utilizam um modelo de execução memória-memória.

VLSI:

O número de transistores que "cabem" numa placa desilício é extremamente elevado e com tendência acrescer ainda mais. O problema agora já não é a falta de espaço para armazenar as funcionalidades necessárias, mas o que fazer com todos os transistores disponibilizados. Retirar das arquiteturas as funcionalidades que só raramente são utilizadas já não é uma estratégia moderna de projeto de processadores. De fato, os projetistas procuram afincadamente mais funcionalidades para integrarem nos seus processadores para fazerem uso dos recursos (transistores) disponíveis. Eles procuram não o que podem tirar, mas o que podem incluir. A maioria das funcionalidades pós-RISC são uma consequência direta do aumento do número de transistores disponíveis e da estratégia "incluir se aumentar o desempenho".

Conjunto de Instruções:

Uma instrução é um comando codificado em 0's e 1's que leva o processador a fazer algo. Quando um programador escreve um programa em C, por exemplo, o compilador traduz cada linha de código C em uma ou mais instruções do processador. Para que os programadores possam (se quiserem) "ver" estas instruções não tendo que lidar com 0's e 1's as instruções são representadas diferente – por exemplo MOV, que copia um valor de uma localização para outra ou ADD, que adiciona dois valores. A seguinte linha de código adiciona dois

valores (b e c) e coloca o resultado em a:a=b+*c*. Um compilador de C poderia traduzir isto na seguinte sequência de instruções:

```
MOV ax, b
ADD ax, c
MOV a, ax
```

A primeira instrução copia o conteúdo da localização de memória que contém o valor **b** para o registrador **ax** do processador (um registrador é uma localização de armazenamento dentro do processador que pode conter certa quantidade de dados, normalmente 16 ou 32 bits. Sendo uma parte integrante do processador, os acessos aos registradores são muitíssimo mais rápidos que os acessos à memória). A segunda instrução adiciona o valor **c** ao conteúdo de **ax**e a terceira copia o resultado, que está em **ax**, para a localização onde a variável **a** está armazenada. Qualquer programa, por mais complexo que seja, é traduzido, em última análise, em séries de instruções do gênero da anterior.

Os programas de aplicação modernos contêm centenas de milhares de linhas de código. Os sistemas operativos são ainda mais complexos: o Microsoft Windows 95 contém cerca de 10 milhões de linhas de código, a maior parte dele escrito em C, e o Windows NT tem mais de 5 milhões de linhas de código escritas em C e C++. Imagine-se o que seria ter que traduzir 1 milhão de linhas de código C num conjunto de instruções com uma a vinte ou trinta instruções por linha de código e é fácil perceber o porquê de o softwarede hoje em dia ser tão complicado – e tão difícil de corrigir.

Quando um programa roda, o processador carrega as instruções uma a uma e as executa. Leva tempo carregar uma instrução e

mais tempo ainda descodificar essa instrução para determinar o que representam os 0's e 1's. E quando começa a execução, é necessário um determinado número de ciclos para completar a execução. Em um 386 a 25 *MHz* um ciclo de relógio é igual a 40 *ns*, em um PENTIUM a 120 *MHz* um ciclo é igual a menos de 9 *ns*. Uma maneira de fazer com que um processador rode o software mais rapidamente é aumentar a velocidade do relógio. Outra é diminuir o número de ciclos que uma instrução requer para completar a execução. Se todo o resto fosse igual, um processador que funcionasse a 100 *MHz* teria apenas metade do desempenho de outro que funcionasse a 50 *MHz*, se o primeiro requeresse 4 ciclos de relógio por instrução e o segundo apenas um ciclo.

Desde o início da era dos microprocessadores, o grande objetivo dos projetistas de *chips* é desenvolver uma CPU que requeira apenas 1 ciclo de relógio por instrução, não apenas certas instruções, mas TODAS as instruções. O objetivo original dos projetistas de *chips* RISC era limitar o número de instruções suportadas pelo *chip* de modo a que fosse possível alocar um número suficiente de transistores a cada uma delas, para que a sua execução precisasse apenas de um ciclo de relógio para se completar.

Em vez de disponibilizar uma instrução MUL, por exemplo, o projetista faria com que a instrução ADD executasse em 1 ciclo de relógio. Então o compilador poderia fazer a multiplicação de **a** e **b** somando **a** a ele próprio **b** vezes ou vice versa. Uma CPU CISC poderia multiplicar **10** e **5** da seguinte forma:

```
MOV ax, 10
MOV bx, 5
MUL bx
```

Enquanto um CPU RISC faria o mesmo de outro modo:

```
MOV ax, 0
MOV bx, 10
MOV cx, 5
begin:
ADD ax, bx
loop begin; loop cx vezes
```

É claro que este é apenas um exemplo ilustrativo, pois os *chips* RISC atuais têm, de fato, instruções de multiplicação.

Medida de Desempenho:

Como a medida de desempenho está diretamente relacionada a um programa em específico, cria-se uma dificuldade em comparar processadores RISC e CISC dadas as diferenças conceituais entre ambas as arquiteturas. Como comparar adequadamente medidas de comportamento desigual? A unidade de medida MIPS (Milhões de Instruções Por Segundo) não é eficaz para esta comparação, pois pode facilmente iludir o observador com os resultados, uma vez que um mesmo programa possui muito mais instruções em uma máquina RISC do que em uma CISC. Dessa forma, utiliza-se com maior frequência a unidade MFLOPS (Milhões de Operações de Ponto Flutuante Por Segundo), por medir a velocidade com que o processador efetivamente realiza os cálculos matemáticos.

Recentemente, a controvérsia envolvendo RISC x CISC diminuiu, devido à grande convergência que vem ocorrendo entre as duas tecnologias. As máquinas RISC tornaram-se mais complexas, em virtude do aumento da densidade das pastilhas e das velocidades do

hardware, e as máquinas CISC passaram a utilizar conceitos tradicionalmente aplicados às máquinas RISC, como o número crescente de registradores e maior ênfase no projeto do *pipeline* de instruções.

5.5 EXERCÍCIOS

1. Quais as principais características dos processadores com arquitetura RISC?

2. Quais as principais características dos processadores com arquitetura CISC?

3. Discorra sobre as principais diferenças entre processadores RISC e CISC enfatizando as vantagens e desvantagens de cada uma dessas arquiteturas.

4. Atualmente, é fácil diferenciar claramente se um processador segue uma arquitetura RISC ou CISC? Por quê?

WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI
http://www.ufpi.br/uapi

Universidade Aberta do Brasil- UAB
http://www.uab.gov.br

Secretaria de Educação a Distância do MEC – SEED
http://www.seed.mec.gov.br

Associação Brasileira de Educação a Distância – ABED
http://www.abed.org.br

Apostilas, Tutoriais e Documentos
http://gabriel.sg.urcamp.tche.br/beraldo/arquitetura_2.htm

Guia do Hardware
http://www.guiadohardware.net

Laércio Vasconcelos
http://www.laercio.com.br

Gabriel Torres
http://www.gabrieltorres.com.br

César Augusto M. Marcon
http://www.inf.pucrs.br/~marcon/

Ivan L. M. Ricarte
http://www.dca.fee.unicamp.br/~ricarte/

Marcelo Trindade Rebonatto
http://vitoria.upf.tche.br/~rebonatto

Fabian Vargas
http://www.ee.pucrs.br/~vargas

Eduardo Moresi
http://www.intelingencia.blogspot.com

REFERÊNCIAS

CARTER, N. **Arquitetura de computadores**. Porto Alegre:Bookman, 2003.

HEURING, V. P; MURDOCCA, M. J. **Introdução à arquitetura de computadores**. Rio de Janeiro: Campus, 2002.

MORIMOTO, C. E. **Hardware: o guia definitivo**. Porto Alegre: Sulina, 2007.

MONTEIRO, M. A. **Introdução aorganização de computadores**. Rio de Janeiro: LTC, 2007.

PARHAMI, B. **Arquitetura de computadores: de microprocessadores a supercomputadores**. São Paulo:McGraw- Hill do Brasil, 2008.

PATTERSON, D. A ; HENNESSY, J. L. **Arquitetura de computadores: uma abordagem quantitativa**. Rio de Janeiro: Campus, 2003.

PATTERSON, D. A ; HENNESSY, J. L. **Organização e projeto de computadores**. Rio de Janeiro: Campus, 2005.

RIBEIRO, C ; DELGADO, J. **Arquitetura de computadores**. Rio de Janeiro: LTC, 2009.

STALLINGS, W. **Arquitetura e organização de computadores**. São Paulo: Prentice Hall, 2008.

TANENBAUM, A. S. **Organização estruturada de computadores**. Rio de Janeiro:Prentice Hall, 2007.

TORRES, G. **Hardware: curso completo**. Rio de Janeiro: Axcel Books, 2001.

WEBER, R. F. **Fundamentos de arquitetura de computadores**. Porto Alegre: Bookman, 2008.

UNIDADE III
SISTEMAS DE MEMÓRIA

Resumo

Os sistemas de memória têm pa

pel crítico no desempenho de um sistema computacional. Estes sistemas recebem dados do mundo externo transferidos para processador e também recebe dados do processador a transferir para mundo externo.

A memória é um conjunto de células, todas com o mesmo número de bits, sendo cada célula identificada por um número único que é seu endereço. Os acessos à memória são realizados através de palavras de memória, sendo que a velocidade da memória, indicada pelo seu tempo de acesso, é significativamente inferior à velocidade com que o processador trabalha.

O conteúdo desta Unidade é influenciado fortemente pelos textos de Nicholas Carter, Rossano Pinto, Jean Laime, Ivan Ricarte e Alexandre Casacurta. *O Capítulo é acompanhado de exercícios, sem a solução, preferimos deixar o prazer desta tarefa ao leitor. A bibliografia e a webliografia ao fim das notas são mais do que suficiente para adquirir um conhecimento razoável sobre a organização de computadores.* ***Ao término da leitura desta Unidade, o estudante deverá: a) Entender o fun-***

cionamento de um sistema de memória, b) Compreender o porquê do uso de hierarquia de memórias; c) Ser capaz de implementar os algoritmos de substituição de páginas e d) Conhecer a técnica de memória virtual.

6. MEMÓRIA

6.1 Introdução

Até agora, temos tratado os sistemas de memória como uma caixa preta na qual o processador pode colocar dados para posterior recuperação. Vamos explorar esta caixa preta para examinar como são implementados os sistemas de memória nos sistemas de memória nos sistemas de computadores modernos.

Este capítulo começa com uma discussão sobre latência, taxa de transferência e largura de banda, as três grandezas utilizadas para medir o desempenho de sistemas de memória. Em seguida, abordaremos hierarquia de memória explicando como e porquê várias tecnologias de memória são utilizadas para implementar um único sistema de memória. Finalmente, cobriremos as tecnologias de memória explicando como os *chips* de memória são implementados, a diferença entre SRAMs e DRAMs e como os diferentes modos de acesso encontrados em DRAMs são implementados.

6.2 Conceitos Básicos

Quando discutimos *pipelines* de processadores, utilizamos os termos **latência** e **taxa de transferência** para descrever o tempo utilizado para completar uma operação individual e a taxa na qual as operações podem ser completadas. Esses termos também são utilizados na discussão de sistemas de memória com o mesmo significado. Outro termo também utilizado é largura de banda, que descreve a taxa total pela qual os dados podem ser movimentados entre o processador e o sistema de memória. A **largura de banda** pode ser vista como o

produto da taxa de transferência e a quantidade de dados referidos por cada operação de memória.

Exemplo:

Se o sistema de memória tem uma latência de 10 ns por operação e uma largura de banda de 32 bits, qual é a taxa de transferência e a largura de banda do sistema de memória, assumindo que apenas uma operação pode ser realizada por vez e não existe retardo entre as operações?

Solução:

Sabendo que taxa de transferência = 1/latência quando as operações são realizadas sequencialmente, então a taxa de transferência do sistema de memória é de 100 milhões de operações por segundo. Uma vez que cada operação faz referência a 32 bits de dados, a largura de banda é de 3,2 bilhões de bits por segundo.

Exemplo retirado de [Carter, N.]

Se todas as operações de memória fossem executadas sequencialmente, calcular a latência e a largura de banda de um sistema de memória seria simples. No entanto, muitos desses sistemas são projetados de forma que o relacionamento entre latência e largura de banda sejam mais complexos. Sistemas de memória podem utilizar *pipelining* do mesmo modo que os processadores utilizam, permitindo que as operações sobreponham a sua execução, de modo a melhorar a taxa de transferência. Algumas tecnologias de memória empregam um tempo fixo entre sucessivos acessos à memória. Esse tempo é usado para preparar o hardware para o próximo acesso. Tal procedimento é denominado **pré-carregamento** e tem o efeito de adiantar parte das tarefas envolvidas no acesso à memória. Isso reduz o retardo entre o tempo no qual um endereço é enviado para o sistema de memória até

que a operação de memória seja completada. Se o sistema de memória estiver ocioso muito tempo, fazer o pré-carregamento ao final de cada operação de memória melhora o desempenho, porque normalmente não existe outra operação esperando para utilizar tal sistema.

6.3 Endereços de Memória

A memória é formada por um conjunto de **células** (ou posições)**,** cada uma das quais podendo guardar uma informação. Cada célula tem um número associado a ela, número esse conhecido como **endereço** da célula. É por meio desse número que os programas podem referenciar a célula. Se a memória tiver *n* células, elas terão endereços de *0* a *n–1*. Todas as células de uma memória dispõem do mesmo número de bits. Se uma célula tiver *k* bits ela poderá armazenar qualquer uma das *2k* combinações possíveis para os bits. A figura 1.1 mostra três organizações diferentes para uma memória de 96 bits. Observe que células adjacentes têm endereços consecutivos (por definição).

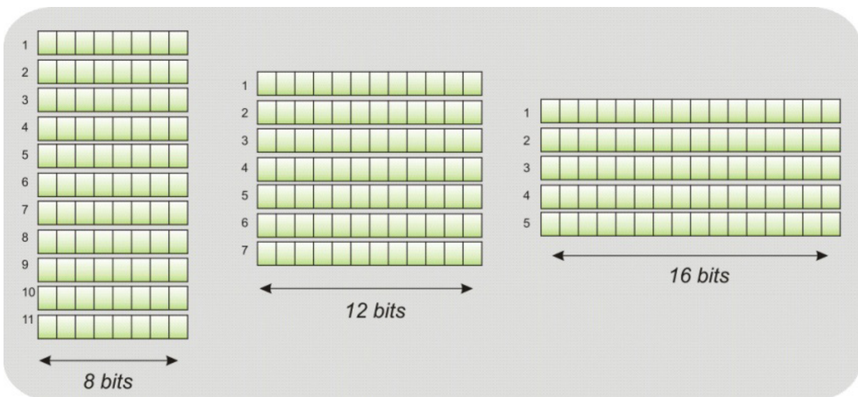


Figura 6.1: Organizações de memória.
Figura adaptada de [Pinto, R.]

Os computadores que usam o sistema de numeração binária (incluindo aqueles que utilizam a notação octal e/ou a hexadecimal para números binários) expressam os endereços de memória como números binários. Se um determinado endereço tem *m* bits, o número máximo de células endereçáveis é de *2m*. Por exemplo, um endereço utilizado para referenciar a memória da figura 6.1 (a) precisa de no mínimo 4 bits para expressar todos os números binários entre 0 e 11. No entanto, no caso da figura 1.1 (b) e da figura 1.1 (c), três bits de endereço serão suficientes. O número de bits no endereço determina o número máximo de endereços que poderão ser referenciados diretamente na memória sendo completamente independente do número de bits por célula. Tanto uma memória com 212 células de 8 bits quanto outra memória com 212 células de 64 bits precisam de 12 bits para representar o endereço da célula.

A célula é a menor unidade endereçável em um computador. Nos últimos anos, quase todos os fabricantes de computadores padronizaram o tamanho da célula em 8 bits, chamando essa célula de **byte.** Os bytes são agrupados em **palavras.** Um computador com uma palavra de 32 bits tem 4 bytes/palavra, enquanto um computador com uma palavra de 64 bits tem 8 bytes/palavra. A maioria das instruções de uma máquina opera sobre palavras, daí a significância do conceito. Por exemplo, uma instrução de soma muito provavelmente vai somar dois valores de 32 bits. Portanto, uma máquina de 32 bits deverá ter registradores e instruções para tratar palavras de 32 bits, enquanto uma máquina de 64 bits deve ter registradores de 64 bits e instruções para mover, somar, subtrair etc. palavras de 64 bits.

6.4 Ordenação dos Bytes

Os bytes de uma palavra podem ser numerados da esquerda para a direita ou da direita para a esquerda. À primeira vista, pode parecer que essa escolha não tem a menor importância, mas, conforme será visto em breve, ela tem implicações muito sérias e importantes. A figura 6.2 (a) mostra parte de uma memória de um computador de 32 bits, cujos bytes são numerados da esquerda para a direita, a exemplo da arquitetura do SPARC ou dos mainframes da IBM**.** A figura 6.2 (b) mostra a representação da memória de um computador de 32 bits, cujos bytes de uma palavra são numerados da direita para a esquerda, a exemplo das máquinas da família Intel. O computador em que a numeração dos bytes das palavras atribui um número maior à parte mais significativa da palavra é chamado de ***big endian*****,** enquanto o computador da figura 1.2(b) é conhecido como computador ***littleendian*****.**

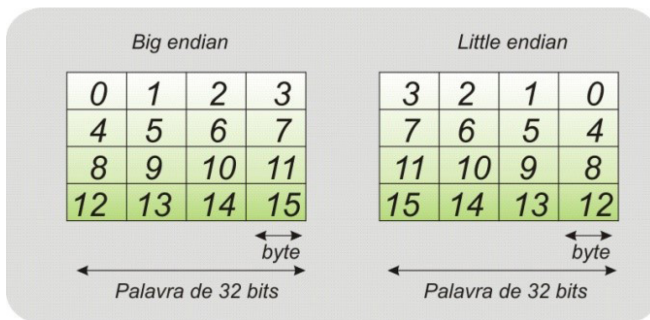


Figura 6.2: Big endian e Little endian.
Figura adaptada de [Pinto, R.]

É muito importante que o leitor entenda que, tanto no sistema *big endian* quanto no *littleendian*, um número inteiro de 32 bits, com um determinado valor numérico, digamos 6, é representado pelos bits 110 nos três bits mais à direita (de mais baixa ordem) de uma pala-

vra, sendo os demais 29 bits preenchidos com zeros. No esquema *big endian*, os bits 110 estarão no byte 3 (ou 7, ou 11 etc.), enquanto no *littleendian* esses bits estarão no byte 0 (ou 4, ou 8 etc.). Em ambos os casos, a palavra contendo esse número inteiro estará armazenada no endereço 0.

Ambas as representações funcionam muito bem e são consistentes. Os problemas começam quando uma máquina *big endian* tenta enviar um registro a outra máquina *littleendian* por meio de uma rede. Vamos admitir que a máquina *big endian* envie o registro byte a byte para a *littleendian*, começando do byte 0 e terminando no byte 19. (Sejamos otimistas e suponhamos que todos os bits transmitidos chegarão incólumes à outra ponta.) Portanto, o byte 0 da máquina *big endian* vai para a memória da máquina *littleendian* no byte 0, e assim por diante, conforme mostra a figura 6.3 (c).

Uma solução óbvia seria um software para inverter os bytes de uma palavra, depois de eles serem copiados. Isso passa a tratar corretamente os números, mas incorretamente a string, conforme mostra a figura 6.3 (d). Observe que, ao ler a string de caracteres, o computador primeiro lê o byte 0 (um espaço), depois o byte 1 (caractere C), e assim por diante. Em função disso, é fácil constatar que a leitura é feita na ordem inversa.

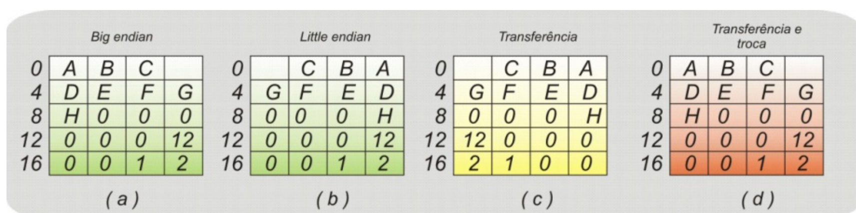


Figura 6.3: Transmissão entre Big endian e Little endian.
Figura adaptada de [Pinto, R.]

Não há uma solução simples para esse problema. Se incluirmos um cabeçalho na frente de cada item de dados, indicando o tipo (caractere, número inteiro etc.) e o tamanho do dado que se segue, teremos uma solução que funciona, mas que é extremamente ineficiente. A inclusão desse cabeçalho permitiria que o receptor fizesse somente as conversões necessárias. Seja como for, deve ter ficado claro que a falta de um padrão para a ordenação dos bytes da palavra dos computadores é um dos principais problemas para a troca de dados entre máquinas diferentes.

6.5 Códigos com Correção de Erros

Os dados armazenados na memória dos computadores podem ocasionalmente ser alterados em razão de oscilações na tensão de alimentação ou de outras causas que, no momento, não são importantes. Para se prevenir contra esses tipos de erro, algumas memórias armazenam as informações usando um código que permita a correção ou a detecção de erros. Quando esses códigos são usados, há necessidade de se acrescentarem bits extras a cada palavra de memória, de modo a permitir a verificação da exatidão da informação armazenada. Quando uma palavra é lida da memória, os bits armazenados permitem verificar a ocorrência eventual de erros que tenham corrompido a informação armazenada.

Para entender o tratamento dado aos erros, é necessário examinar o que vem a ser um erro em uma informação armazenada na memória de um computador. Suponha que uma determinada palavra de memória contenha m bits de dados aos quais vamos acrescentar mais r bits de redundância ou de verificação. Seja n o novo tamanho da

palavra (ou seja, $n = m + r$) Uma unidade de n bits, sendo m de dados e r de redundância será chamada de **palavra de código.**

Dadas duas palavras de código quaisquer, digamos *10001001* e *10110001*, é possível determinar o número de bits diferentes que **ocupa** a mesma posição nessas duas palavras. Nesse caso, existem três bits diferentes em posições correspondentes. Para determinar isso, basta que seja calculada a função booleana OU EXCLUSIVO correspondente às duas palavras de código, contando a seguir o número de bits 1 gerados no resultado.

Chama-se **distância de Hamming** ao número de bits correspondente que difere em duas palavras de código quaisquer. O significado desse resultado é muito importante, pois se duas palavras distam d, será necessária a ocorrência de d erros para que uma palavra seja transformada na outra. Por exemplo, as palavras de código *11110001* e *00110000* distam 3 unidades de Hamming, pois são necessários três erros para que uma se converta na outra.

Com uma palavra de memória de m bits, aparentemente todos os $2m$ padrões de bits são legais, mas, se considerarmos que vamos acrescentar a esses m bits os r bits de verificação, então, das $2n$ combinações possíveis somente $2m$ serão válidas. Dessa maneira, quando a memória lê um código inválido, o sistema sabe que houve um erro no valor armazenado naquela posição. Se conhecermos o algoritmo para cálculo dos bits de verificação, será possível construir uma tabela com todas as palavras de código válidas e dessa tabela extrair as duas palavras de código com a menor distância de Hamming. Essa distância será à distância para todo o código.

As propriedades de detecção de erros e de correção de erros de um código dependem fundamentalmente da sua distância de Ham-

ming. Para detectar *d* erros, é necessária uma distância de Hamming de *d + 1*, pois em tal código não há como a ocorrência de *d* erros transformar uma palavra de código válida em outra palavra de código válida. Assim também, para corrigir *d* erros, precisamos de um código com distância de *2d + 1* para permitir que, mesmo em presença de *d* erros, a palavra de código original possa ser recomposta por meio dos bits redundantes.

Vamos examinar um exemplo simples de um código com detecção de erros. Para isto, considere um código no qual um **bit de paridade** seja acrescentado aos bits de dados do código. Esse bit de paridade deve ser escolhido de modo a fazer com que a quantidade de bits 1 na palavra de código seja sempre um número par (ou ímpar). Tal código tem distância de Hamming igual a 2 — portanto, a ocorrência de um único erro produz uma palavra de código inválida. Ou seja, são necessários dois erros para transformar uma palavra de código válida em outra palavra de código válida. Esse esquema pode ser usado para detectara ocorrência de um único erro na informação armazenada. Toda vez que uma informação com paridade errada for lida da memória, é sinalizada uma condição de erro. Nesse caso, como o erro só é detectado, e não corrigido, o programa não pode continuar seu processamento, porém a ação de detecção impede o cálculo de resultados errados.

Como exemplo de um código com correção de erro, considere o caso de um código com somente quatro palavras de código válidas:

***Exemplo**: 0000000000, 0000011111, 1111100000, e 1111111111*

Esse código tem uma distância de Hamming igual a 5, o que

significa que ele pode corrigir dois erros. Quando o receptor recebe uma palavra de código igual a *0000000111*, ele sabe que o correto deveria ser 0000011111 (considerando a ocorrência de no máximo dois erros). No entanto, se ocorrerem três erros, transformando, por exemplo, a palavra de código 0000000000 em 0000000111, os mesmos não poderão ser corrigidos.

6.6 Hierarquia de Memória

Até agora, tratamos os sistemas de memória como estruturas de um único nível, como mostra a figura 6.4 (a). Na realidade, sistemas de memória de computadores modernos têm hierarquias de memórias de vários níveis, como ilustrado na figura 6.4 (b) e figura 6.4 (c). A figura 6.4 (b) mostra uma hierarquia de memória em dois níveis: uma memória principal e uma memória secundária. Já a figura 1.4 (c) apresenta uma hierarquia em três níveis, consistindo de uma cache, uma memória principal e uma memória secundária. A razão principal pela qual sistemas de memória são construídos como hierarquias é que o custo por *bit* de uma tecnologia de memória é geralmente proporcional à velocidade da tecnologia.

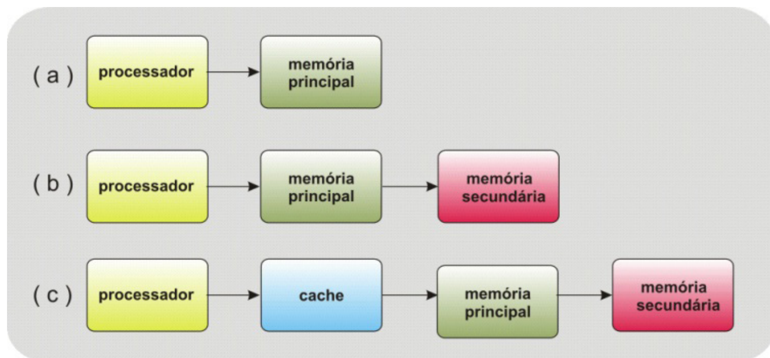


Figura 6.4: Hierarquia de memória.

Em uma hierarquia de memória, os níveis mais próximos ao processador contêm uma quantidade relativamente pequena de memória que é implementada em uma tecnologia de memória rápida, de modo a fornecer baixo tempo de acesso. Progredindo para baixo na hierarquia, cada nível contém mais capacidade de armazenamento e o acesso demora mais que o nível acima dele. O objetivo de uma hierarquia de memória é manter os dados que serão mais referenciados por um programa nos níveis superiores da hierarquia, de modo que a maioria das solicitações de memória possam ser tratadas no nível ou nos níveis superiores. Isto resulta em um sistema de memória que tem um tempo de acesso médio semelhante ao tempo de acesso do nível mais rápido, mas com o custo médio por *bit* semelhante àquele do nível mais baixo.

Em geral, não é possível prever quais localizações de memória serão acessadas com mais frequência, de modo que os computadores utilizam um sistema baseado em demanda para determinar quais dados manter nos níveis mais altos da hierarquia. Quando a solicitação de memória é enviada para a hierarquia, o nível mais alto é verificado para ver se ele contém o endereço. Se for assim, a solicitação é completada. Caso contrário, o próximo nível mais baixo é verificado, com o processo sendo repetido até que, ou o dado seja encontrado, ou o nível mais baixo da hierarquia seja atingido, no qual se tem a garantia que o dado está contido.

Tradicionalmente, uma hierarquia de memória incluindo os registradores e colocando informações referentes à capacidade, custo e tempo de acesso é expressa na fora de pirâmide como mostra a figura 6.5. A representação em forma de pirâmide facilita a compreensão, pois no topo estão os registradores de acesso quase imediato com capa-

cidade reduzida, enquanto na base encontra-se a memória secundária, disponível em grandes quantidades, porém de acesso mais lento.

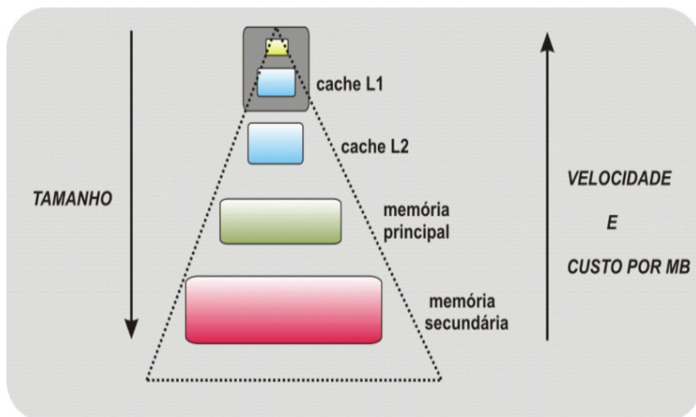


Figura 6.5: Hierarquia de memória- Pirâmide.

6.7 Propriedades de uma Hierarquia

As principais propriedades de um sistema de hierarquia de memória são: Inclusão, Coerência e Localidade de Referência. Considere uma hierarquia de memória como mostra a figura 6.6 onde M_1 , M_2 , . . . , M_n tal que M_i é mais rápida, porém menor que M_{i+1}, e os registradores do processador podem ser considerados como o nível 0 da hierarquia de memória.

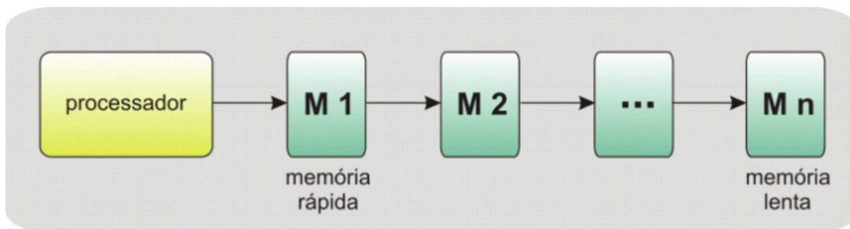


Figura 6.6: Hierarquia de memória.
Figura adaptada de [Ricarte, I.]

Inclusão:

Cada conjunto de dados em M_i deve estar contido no conjunto de dados em M_{i+1}:$M_1 \subset M_2 \subset \ldots \subset M_n$. Assim, M_n seguramente contém todos os itens de dados, e os subconjuntos de M_n são copiados para os outros níveis M_i, $i < n$, ao longo do processamento. Um item em M_i seguramente terá cópias em M_{i+1} , M_{i+2} , ..., M_n , porém, um item em M_{i+1} não está necessariamente em M_i quando requisitado àquele nível. Um *item miss* caracteriza uma falha de acesso ao item no nível em que ele foi solicitado.

Coerência:

Cópias de um mesmo item devem ser consistentes ao longo de níveis sucessivos da hierarquia de memória. Há dois métodos básicos para manutenção da consistência entre os níveis da hierarquia:

- *write-through*: atualização imediata em M_{i+1} quando item é modificado em M_i;

- *write-back*: atualização só é realizada quando o item estiver sendo retirado de M_i.

Localidade de Referência:

Localidade é o conceito fundamental para o funcionamento adequado da hierarquia de memória. É definida como o comportamento de programas segundo o qual referências à memória geradas pela CPU, para acesso a instruções ou a dados, fazem com que estes acessos estejam agrupados em certas regiões (ou seqüência) no tempo ou no espaço. As duas principais formas de localidade são:

Localidade temporal: os itens referenciados no passado recente têm maior chance de serem novamente referenciados em um futuro próximo. Ocorre, por exemplo, em sub-rotinas, iterações e referências a variáveis temporárias.

Localidade espacial: tendência de processos em acessar itens cujos endereços estão próximos. Ocorre, por exemplo, em operações em arranjos e referências a segmentos de programas. A localidade espacial está relacionada com a localidade sequencial pela qual elementos em posições consecutivas da memória têm chance de serem acessados em sequência, como ocorre, por exemplo, em referências a instruções de um programa e a elementos de um arranjo.

Pelo princípio da localidade, é possível construir sistemas de memória hierárquica de forma eficiente. Exemplos de seu uso incluem sistemas de memória *cache*, entre processador e memória principal; e sistemas de memória virtual, operando com as memórias principal e secundária. Memórias *cache* podem estar localizadas no *chip* do processador (internas), ou ser construídas externamente com dispositivos SRAM. A memória principal geralmente utiliza dispositivos DRAM, enquanto que a memória secundária é usualmente constituída por discos magnéticos.

Tempo Médio de Acesso

Foi desenvolvido um conjunto de termos para descrever hierarquia de memória. Quando um endereço que está sendo referenciado por uma operação é encontrado em um nível da hierarquia de memó-

ria, diz-se que ocorreu um acerto naquele nível. Caso contrário diz-se que ocorreu uma falha. De modo semelhante, a taxa de acertos (*hit ratio*) de um nível é a porcentagem de referência em um nível que resultam em acertos, e a taxa de falhas (*miss ratio*) é o percentual de referências em um nível que resultam em falhas.

Se conhecemos a taxa de acerto e o tempo de acesso (tempo para completar uma solicitação) para cada nível na hierarquia de memória podemos calcular o tempo médio de acesso da hierarquia. Para cada nível na hierarquia o tempo médio de acesso é:

$$(T_{acerto} \ x \ P_{acerto}) + (T_{falha} \ x \ P_{falha})$$

Onde, T_{acerto} é o tempo para completar uma solicitação que acerte, P_{acerto} é a taxa de acertos do nível (expressa como uma probabilidade), T_{falha} é o tempo de acesso médio dos níveis abaixo deste na hierarquia, e P_{falha} é a taxa de falha do nível. Uma vez que a taxa de acertos do nível mais baixo é 100%, podemos começar no nível mais baixo e trabalhar em direção ao topo para calcular o tempo de acesso médio de cada nível na hierarquia.

6.8 Tecnologias de Memória

Hoje em dia, a memória principal é fabricada a base de pastilhas de semicondutores. Contudo, existem vários outros tipos de memórias que utilizam os semicondutores. A Tabela 1 apresentada a seguir contém uma lista dos principais tipos de memórias de semicondutores.

Tabela 1: Memória de semicondutores.

<table>
<tr><th>Tipo de memória</th><th>Categoria</th><th>Mecanismo para apagar</th><th>Mecanismo de escrita</th><th>Volatilidade</th></tr>
<tr><td>RAM</td><td>Memória de leitura e escrita</td><td>Eletricamente, em nível de bytes</td><td>Eletricamente</td><td>Volátil</td></tr>
<tr><td>ROM</td><td rowspan="2">Memória apenas de leitura</td><td rowspan="2">Não é possível</td><td>Máscaras</td><td rowspan="5">Não-volátil</td></tr>
<tr><td>PROM</td><td rowspan="4">Eletricamente</td></tr>
<tr><td>EPROM</td><td rowspan="3">Memória principalemnte de leitura</td><td>Luz UV, em nível de pastilha</td></tr>
<tr><td>Flash</td><td>Eletricamente, em nível de blocos</td></tr>
<tr><td>EEPROM</td><td>Eletricamente, em nível de bytes</td></tr>
</table>

Tabela adaptada de [Laime, J.]

Dentre os tipos de memórias ilustrados (figura 6.7), a **RAM** (*Random-Access Memory*) é a mais conhecida. O objetivo da RAM é permitir que os dados sejam lidos e escritos rapidamente e de modo fácil. Conforme mostra a tabela, tanto a leitura quanto a escrita são feitas por meio de sinais elétricos. Para manter seus dados, ela requer o fornecimento de energia constante (volátil). Além disso, existem RAMs dinâmicas (usam capacitores para armazenar informações, portanto requerem sinais de*refresh*) e estáticas (construídas através de *flip-flops*), mas ambas são voláteis. Em geral, as RAMs estáticas são mais rápidas que as dinâmicas.

O segundo tipo ilustrado foi a memória **ROM** (*Read-OnlyMemory*), memória apenas de leitura. Sua principal vantagem é que os dados ficam permanentemente armazenados, não precisando ser carregados a partir de um dispositivo secundário. Os dados são gravados

na pastilha durante a fabricação através de máscaras que contém as informações. Portanto, não é permitido erros durante o processo de fabricação. Consequentemente, para que o processo se torne mais barato é necessário a fabricação de muitas unidades com a mesma informação.

Por outro lado, quando poucas unidades serão fabricadas, a melhor alternativa é a memória ROM programável (*ProgrammableROM*). A **PROM** permite que os dados sejam gravados apenas uma vez, só que isso é feito eletricamente pelo fornecedor ou pelo cliente após sua fabricação. Para isso é necessário um aparelho de gravação de PROM. Outra classe de memórias são aquelas destinadas principalmente para operações de leitura, tais como: EPROM (*Erasable Programmable Read Only Memory*), EEPROM (*Electrically Erasable Programmable Read Only Memory*) e FLASH.

A **EPROM** é uma PROM que pode ser apagada. Pode ser programável (com queimadores de PROM) e apagada (com máquinas adequadas à base de raios ultravioleta). Tem utilização semelhante à da PROM, para testar programas no lugar da ROM, ou sempre que se queira produzir ROM em quantidades pequenas, com a vantagem de poder ser apagada e reutilizada. Já a **EEPROM** é uma EPROM que pode ser apagada por processo eletrônico, sob controle da UCP, com equipamento e programas adequados. É mais cara e é geralmente utilizada em dispositivos aos quais se deseja permitir a alteração via modem possibilitando a carga de novas versões de programas à distância, ou, então, para possibilitar a reprogramação dinâmica de funções específicas de um determinado programa, geralmente relativas ao *hardware* (p.ex., a reconfiguração de teclado ou de modem, programação de um terminal, etc.).

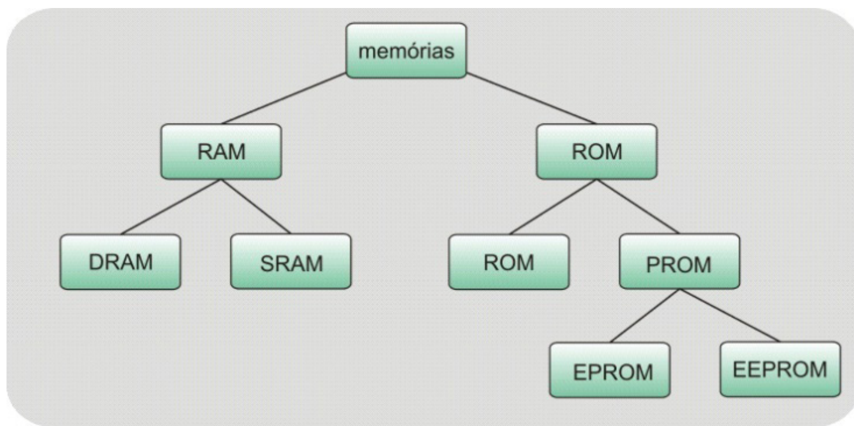


Figura 6.7: Tecnologia de memória.
Figura adaptada de [Casacurta, A.]

6.9 Organização dos Chips de Memória

Os *chips* de memória SRAM e DRAM têm a mesma estrutura básica, que é mostrada na figura 6.8. Os dados são armazenados em uma matriz retangular de ***células de bit***, cada uma das quais retém um *bit* de dados. Para ler dados da matriz, metade do endereço a ser lido (geralmente os bits de ordem mais alta) é enviado para um **decodificador**. Este aciona (vai para o nível alto) a **linha de palavra** correspondente ao valor dos *bits* de entrada, o que faz com que todas as *células de bit* na linha correspondente acionem os seus valores sobre as ***linhas de bit*** as quais elas são conectadas. Então, a outra metade do endereço é usada como entrada para um **multiplexador** que seleciona a *linha de bit* apropriada e orienta a sua saída para os pinos de saída do *chip*. O mesmo processo é utilizado para armazenar dados no *chip*, exceto se o valor a ser escrito é orientado para a *linha de bit* adequada. Neste caso é escrito na *célula de bit* escolhida.

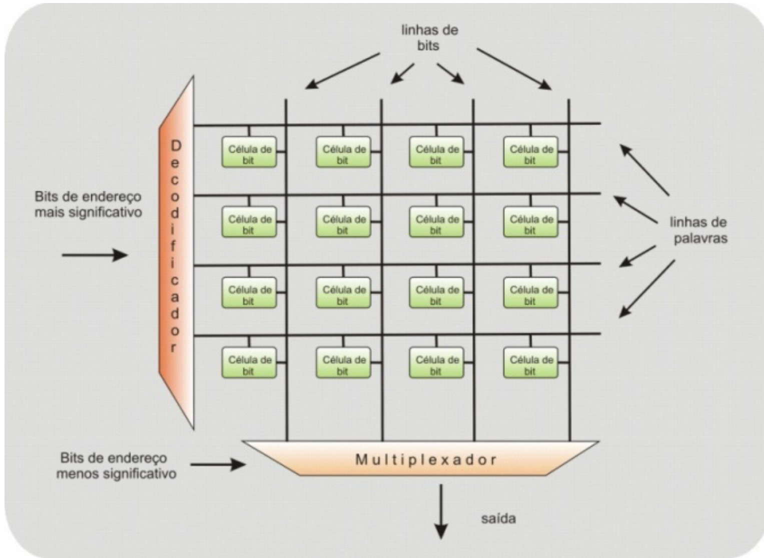


Figura 6.8: Organização de chips de memória.
Figura adaptada de [Carter, N.]

A maioria dos chips de memória gera mais de um bit de saída. Isto é feito, ou construindo-se diversas matrizes de células de bit, cada uma das quais produz um bit de saída, ou projetando um multiplexador que seleciona as saídas de diversas linhas de bit e as orienta para a saída do chip.

A velocidade de um chip de memória é determinada por alguns fatores, incluindo o comprimento das linhas de bit, de palavra e como as células de bit são construídas. Linhas de palavras e de bits mais longas têm capacitância e resistência mais altas, de modo que, à medida que seus comprimentos aumentam, demora mais para acionar um sinal sobre elas. Por esse motivo, muitos chips de memória modernos são construídos com muitas matrizes de células de bit pequenas para manter as linhas de palavra e de bit curtas. As técnicas utilizadas para construir tais células afetam a velocidade do chip de memória porque elas interferem em quanta corrente está disponível para acionar a saída

da célula de bit sobre as linhas de bit. Isso que determina quanto tempo demora para propagar a saída da célula de bit para o multiplexador. Como veremos nas duas próximas seções, células de bit SRAM podem acionar muito mais corrente do que células de bit DRAM, o que é um dos princípios pelos quais as SRAMs tendem a ser muito mais rápidas do que as DRAMs.

SRAMs

A principal diferença entre SRAMs e DRAMs é como as suas células de bit são construídas. Como mostra a figura 6.9, o núcleo de uma célula de bit SRAM consiste de dois inversores conectados em uma configuração "*back-to-back*". Uma vez que um valor tenha sido colocado na célula de bit, a estrutura em anel dos dois inversores manterá o valor indefinidamente porque cada entrada de um inversor é oposta do outro. Este é o motivo pelo qual SRAMs são chamadas de RAMs estáticas. Os valores armazenados na RAM permanecem lá enquanto houver energia aplicada ao dispositivo. Por outro lado, DRAMs perderão o seu valor armazenado ao longo do tempo, motivo pelo qual são conhecidas como RAMs dinâmicas.

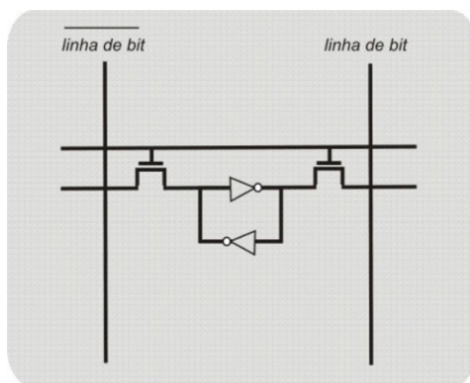


Figura 6.9: Célula de bit SRAM. Figura adaptada de [Carter, N.]

Para ler o valor de uma célula de bit, a linha de palavra tem que ser colocada no nível alto, o que faz com que os dois transistores liguem as saídas dos inversores à linha de bit e à linha de bit invertido.

Estes sinais, então, podem ser lidos pelo multiplexador e enviados para fora do chip. Escrever em uma célula de bit SRAM é feito ativando a linha de palavra e acionando os valores apropriados sobre a linha de bit e a linha de bit invertido. Enquanto o dispositivo que está acionando alinha de bit for mais forte que o inversor, os valores na linha de bit serão dominados pelo valor originalmente armazenados na célula de bit e, então, serão armazenados na célula de bit quando a linha de palavra deixar de ser ativada.

DRAMs

A figura 6.10 mostra uma célula de bit DRAM. Em vez de um par de inversores é utilizado um capacitor para armazenar os dados na célula de bit. Quando a linha de palavra é ativada, o capacitor é conectado à linha de bit, permitindo que o valor armazenado na célula seja lido ao se examinar a tensão armazenada no capacitor, ou escrever colocando uma nova tensão sobre o mesmo. Esta figura mostra porque as DRAMs geralmente têm capacidades muito maiores que as SRAMs construídas com a mesma tecnologia de fabricação: as SRAMs exigem muito mais componentes para implementar uma célula de bit. Tipicamente, cada inversor exige dois transistores para um total de seis transistores em uma célula de bit (algumas implementações utilizam pouco mais ou menos transistores). Em contraste,

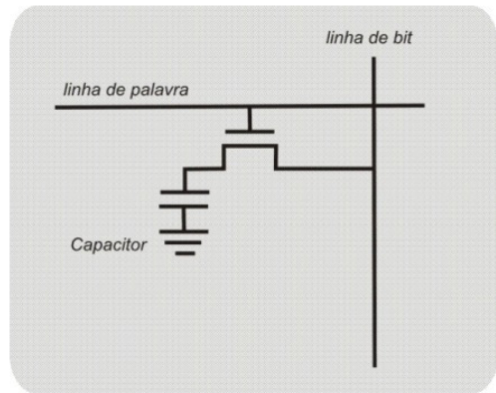


Figura 6.10: Célula de bit DRAM.
Figura adaptada de [Carter, N.]

uma célula de bit DRAM exige apenas um transistor com capacitor, o que toma menos espaço no chip.

As figuras 6.9 e 6.10 também servem para mostrar porque as SRAMs são mais rápidas que as DRAMs. Nas SRAMs, um dispositivo ativo (o inversor) aciona o valor armazenado na célula de bit sobre a linha de bit e a linha de bit invertido. Nas DRAMs, o capacitor é conectado à linha de bit quando a palavra é ativada, e esse sinal é muito mais fraco do que o produzido pelos inversores na célula de bit SRAM. Assim, a saída de uma célula de bit DRAM demora muito mais para que seja acionada sobre a alinha de bit do que a célula de bit SRAM demora para acionar a equivalente linha de bit.

As DRAMs são chamadas de RAMs dinâmicas porque os valores armazenados em cada célula de bit não são estáveis. Ao longo do tempo, fugas de corrente farão com que as cargas armazenadas nos capacitores sejam drenadas e perdidas. Para evitar que o conteúdo de uma DRAM seja perdido, ela precisa ser refrescada. Essencialmente uma operação de refrescamento (*refresh*) lê os conteúdos de cada célula de bit em uma linha da matriz de células de bit e, então, escreve os mesmos valores de volta nas células de bit, restaurando-os aos valores originais.

Desde que cada linha na DRAM seja refrescada com frequência suficiente para que nem uma das cargas do capacitor caia baixo o suficiente para que o hardware interprete mal os valores armazenando em uma linha, a DRAM pode manter o seu conteúdo indefinidamente. Uma das especificações de uma DRAM é o tempo de refrescamento (*refresh time*), que é a frequência pela qual uma linha pode ficar sem ser refrescada antes que esteja correndo o risco de perder o seu conteúdo.

Exemplo:

Se uma DRAM tem 512 linhas e o seu refresh time é de 10 ms, com que frequência, em média, uma operação de refresh de linha precisa ser feita?

Solução:

Dado que o refresh time é de 10 ms, cada linha precisa ser "refrescada"no mínimo a cada 10 ms. Uma vez que há 512 linhas, temos que fazer 512 operações de "refrescamento" de linha em um período de 10 ms que resulta em um refresh de linha a cada 1,95 . 10^{-5} s. É importante comentar que os projetistas devem definir como a operação de refresh é realizada se uniformemente por blocos ou outras alternativas.

Exemplo retirado de [Carter, N.]

6.10 EXERCÍCIOS

1. Explique: latência, taxa de transferência e endereçamento para sistemas de memória.

2. Explique as técnicas de armazenamento *big endian* e *littleendian*. Considere que a memória de um computadore é uma matriz 3x3, como seria a representação, nos dois sistemas, da palavra "MEMORIA" sabendo que cada célula pode conter apenas um caractere?

3. O que é e para que serve a distância de Hamming? Qual a distância de Hamming de 10001001 e 10110001.

4. Por que as memórias são organizadas em níveis hierárquicos? Quais as propriedades de cada nível?

5. Em uma hierarquia de memória de dois níveis, se o nível mais alto tem tempo de acesso de 8 ns e o nível mais baixo tem um tempo de acesso de 60 ns, qual a taxa de acerto no nível mais alto para obtermos um tempo de acesso médio de 10 ns?

6. Se um sistema de memória tem um tempo de acesso médio de 12 ns, o nível mais alto tem taxa de acertos de 90 % e um tempo de acesso de 5 ns, qual o tempo de acesso do nível mais baixo?

7. Mostre na forma de uma árvore as tecnologias de memória e comente sobre cada uma delas.

8. Explique a organização dos chips de memória.

9. Quais as diferenças entre as células de bit de uma memória DRAM e SRAM? Há diferença de velocidade entre elas? Qual a mais rápida? Por quê?

10. Para os seguintes casos determine se SRAMs ou DRAMs seriam os blocos de memória mais adequados para o sistema de memória e explique o porquê. Assuma que existe apenas um nível de memória.
a. Um sistema de memória no qual o desempenho é o dispositivo mais importante;
b. Um sistema de memória no qual o custo é o fator mais importante;
c. Um sistema no qual os dados sejam armazenados por longos períodos sem qualquer atividade por parte de processador.

7 MEMÓRIA CACHE

7.1 Introdução

Ao longo do tempo, os processadores têm sido sempre mais rápidos que as memórias. É fato que o tempo de operação das memórias tem melhorado bastante, mas como o mesmo ocorre com os processadores, permanece o descompasso entre as velocidades desses dois importantes componentes de um sistema computacional. Na medida em que se torna tecnicamente possível a colocação de mais e mais circuitos dentro de um chie, os projetistas dos processadores vêm usando essas novas facilidades para implementar o processamento pipeline e o processamento superescalar, tornando os processadores ainda mais rápidos.

Já os projetistas das memórias têm usado as novas tecnologias para aumentar a capacidade das memórias e não a sua velocidade operacional, de maneira que o problema do desbalanceamento das velocidades de operação entre esses dois dispositivos tem-se agravado com o tempo. Na prática, isso significa que um processador deve esperar vários ciclos de clock até que a memória atenda a uma requisição sua para leitura ou escrita. Quanto mais lenta for a memória em relação à velocidade do processador, mais ciclos de espera serão necessários para ele compatibilizar a operação desses dois elementos de hardware.

Conforme mencionamos anteriormente, existem duas maneiras de lidar com essa questão. Na mais simples delas, as operações de leitura da memória devem ser iniciadas sempre que o processador encontrar uma instrução de *READ*. Mesmo com a operação de leitura iniciada, o processador deve continuar seu processamento, só parando se uma instrução tentar usar a palavra requisitada da memória antes de

ela estar disponível. Quanto mais lenta a memória, mais frequentes se tornarão essas paradas e maior é a penalidade quando elas ocorrerem. Por exemplo, se o tempo de operação da memória for de 10 ciclos, é muito provável que uma das 10 instruções seguintes tentará usar a palavra que estiver sendo lida.

A outra solução é fazer com que o compilador gere seu código, evitando que instruções usem os valores requisitados à memória antes que esses valores estejam disponíveis, impedindo assim a parada do processador. O problema dessa solução é que ela é muito simples de descrever, mas muito difícil de implementar. Muitas vezes, depois de uma instrução de *LOAD* nada mais há a fazer, de maneira que o compilador será obrigado a inserir instruções *NOP* (*no operation*), instruções essas que nada fazem a não ser gastar o tempo do processador. Na verdade, essa solução substitui a parada por hardware prevista na solução anterior e institui a parada por software, mantendo inalterada a degradação da performance do sistema como um todo.

Vale observar que a raiz do problema anteriormente citado é econômica e não tecnológica. Os engenheiros já sabem como construir memórias que sejam tão rápidas quanto os processadores. Acontece que, para operar a essa velocidade, as memórias precisam estar implementadas dentro do chip do processador (a transferência de dados no barramento é um dos principais ofensores da velocidade das memórias). A colocação de uma memória muito grande dentro do chip do processador vai aumentar o seu tamanho, e evidentemente vai torná-lo mais caro, mas mesmo que o custo não venha a ser uma limitação, existem restrições fortes para o aumento indiscriminado do tamanho do chip. Portanto, a escolha fica entre ter uma pequena quantidade de memória rápida, dentro do chip, ou uma grande quan-

tidade de memória lenta, fora do chip. Na verdade, um bom projeto deveria buscar memórias rápidas, de alta capacidade, a preços baixos.

A busca de tais características levou a técnicas que combinam memórias pequenas e rápidas com memórias grandes e lentas, na tentativa de fazer com que o sistema opere com a velocidade da memória mais rápida e com a capacidade da memória mais lenta, tudo isso a um preço bastante acessível. A memória pequena e rápida ficou conhecida pelo nome de **cache** (da palavra francesa *cacher,* que significa esconder).

A ideia básica que há por trás do conceito de memória cache é muito simples: as palavras de memória mais usadas pelo processador devem permanecer armazenadas na cache. Quando o processador precisar de uma palavra, ele primeiro busca essa palavra na cache. Somente no caso de ela não estar armazenada na cache é que a busca se dará na memória principal. Se uma parte substancial dos acessos for satisfeita pela cache, o tempo médio de acesso será muito pequeno, próximo ao da cache.

7.2 Conceitos Básicos

Parte do problema de limitação de velocidade do processador refere-se à diferença de velocidade entre o ciclo de tempo da CPU e o ciclo de tempo da memória principal, ou seja, a memória principal transfere bits para a CPU em velocidades sempre inferiores às que a CPU pode receber e processar os dados, o que acarreta, muitas vezes, a necessidade de acrescentar-se um tempo de espera para a CPU (*waitstate*).

O problema de diferença de velocidade se torna difícil de solu-

cionar apenas com melhorias no desempenho das memórias principais devido a fatores de custo e tecnologia. Enquanto o desempenho dos microprocessadores, por exemplo, vem dobrando a cada 18/24 meses, o mesmo não acontece com a taxa de transferência e o tempo de acesso das memórias DRAM, que vem aumentando pouco de ano para ano.

Na busca de uma solução para este problema foi desenvolvida uma técnica que consiste da inclusão de um dispositivo entre a CPU e a memória principal, denominado de memória cache cuja função é acelerar a transferência de informações entre CPU e memória principal e, com isso, aumentar o desempenho do sistema de computação.

A cachê é um nível na hierarquia de memória entre a CPU e a memória principal. É construída com memória SRAM, que é muito mais rápida do que a DRAM, normalmente empregada na construção das memórias principais. A cache é fabricada com tecnologia semelhante à da CPU e, em consequencia, possui tempos de acesso compatíveis, resultando numa considerável redução da espera da CPU para receber dados e instruções. Como o custo da memória estática é muito alto, a cache normalmente é muito menor do que a memória principal.

Com a inclusão da cache pode-se enumerar, de modo simplista, o funcionamento do sistema como segue:

- Sempre que a CPU vai buscar uma nova informação (instruções ou dados), ela acessa a memória cache;
- Se a informação estiver na cache (*cache hit*), ela é transferida em alta velocidade (compatível com a da CPU);
- Se a informação não estiver na cache (*cache miss*), então o sistema está programado para transferir a informação

desejada da memória principal para a cache. Só que esta informação não é somente da instrução ou dado desejado, mas dele e de um grupo subsequente, na pressuposição de que as instruções/dados do grupo serão requeridas pela CPU em seguida e, portanto, já estarão na cache quando necessário. A figura 7.1 ilustra a troca de informações entre a CPU, cache e memória principal.

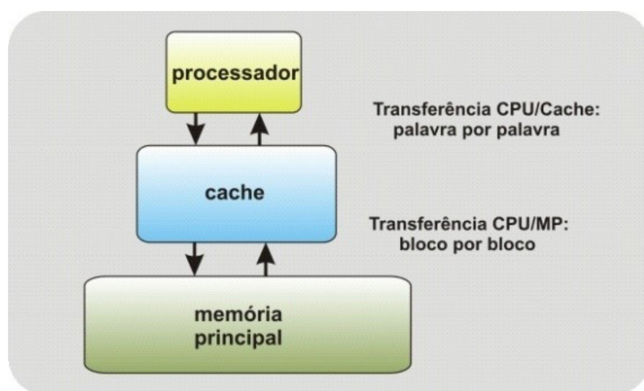


Figura 7.1: Transferência de dados CPU/Cache/MP.

Para haver algum aumento no desempenho de um sistema de computação com a inclusão da memória cache, é necessário que haja mais acertos (*hits*) do que faltas (*miss*). Isto é, a memória cache somente é produtiva se a CPU puder encontrar uma quantidade apreciável de palavras na cache suficientemente grande para sobrepujar as eventuais perdas de tempo com as faltas. Faltas estas que redundam em transferência de um bloco de palavras da memória principal para a cache, além da efetiva transferência da informação desejada.

A implementação de memórias foi referendada pela existência muito forte de localidades na execução dos programas. Existem dois tipos de localidades:

Localidade Temporal: *a*s posições da memória, uma vez acessadas, tendem a ser acessadas novamente num futuro próximo. Normalmente ocorrem devido ao uso de laços de instruções, acessos a pilhas de dados e variáveis como índices, contadores e acumuladores;

Localidade Espacial: se um programa acessa uma palavra de memória, há uma boa probabilidade de que o programa acesse num futuro próximo uma palavra subsequente ou um endereço adjacente àquela palavra que ele acabou de acessar. Em outras palavras, os endereços em acessos futuros tendem a ser próximos de endereços de acessos anteriores. Ocorre devido ao uso da organização sequencial de programas.

O tamanho da cache influencia diretamente no desempenho do sistema, uma vez que para existir um aumento real de desempenho o número de acertos deve superar e muito o número de faltas. Com uma cache maior, há maior probabilidade de a informação desejada estar contida nela, porém outros fatores também devem ser levados em conta, como o tempo de acesso da memória principal, da memória cache e a natureza dos programas em execução (principio da localidade).

Um fator que influencia diretamente o desempenho da cache é a forma de mapeamento dos dados da memória principal nela. Como o tamanho da cache é bem menor que o da memória principal, apenas uma parte dos dados da memória principal pode ser copiada na cache. Existem basicamente três formas de mapeamento:

Mapeamento Completamente Associativo: a palavra pode ser

colocada em qualquer lugar da cache. Neste caso deve ser armazenado na cache não somente o dado, mas também o endereço. Para descobrir se a posição procurada está armazenada na cache, é feita a comparação simultânea de todos os endereços. Caso seja localizado (*cache hit*) o dado é devolvido ao processador; Caso o endereço pesquisado não se encontre na cache (*cache mis*s), a memória principal é acessada.

Mapeamento Direto: cada palavra deve ser armazenada em um lugar específico na cache, o qual depende do seu endereço na memória principal. O endereço é dividido em duas partes: *tag* e *índice*. O índice é usado como endereço na cache e indica a posição onde pode estar armazenada a palavra. O tag é usado para conferir se a palavra que está na cache é a que está sendo procurada, uma vez que endereços diferentes com o mesmo índice serão mapeados sempre para a mesma posição da cache.

Mapeamento Set-Associativo: é um projeto intermediário entre os dois anteriores. Neste caso, existe um número fixo de posições onde a palavra pode ser armazenada (pelo menos duas) que é chamado um conjunto. Como na cache com mapeamento direto, o conjunto é definido pela parte do endereço chamada índice. Cada um dos tags do conjunto são comparados simultaneamente como tag do endereço. Se nenhum deles coincidir ocorre um *cache miss*.

Outro problema a ser considerado na implementação de uma memória cache é a política de substituição das palavras. Deve-se res-

ponder a seguinte pergunta: “Em que local da cache será colocada a nova linha?” A política de substituição define qual linha será tirada da cache para dar lugar a uma nova. No caso do mapeamento direto, cada palavra tem um lugar predefinido, então não existe escolha. Para o mapeamento completamente associativo, pode-se escolher qualquer posição da cache; e no set-associativo qualquer posição dentro do conjunto definido pelo índice. As principais políticas de substituição são:

Substituição Aleatória: neste caso é escolhida uma posição qualquer da cache aleatoriamente para ser substituída. É mais simples de implementa**r,** mas não leva em conta o princípio da localidade temporal.

FIFO (First-in First-out): remove a linha que está a mais tempo na cache. Exige a implementação de uma fila em hardware.

LRU (LeastRecentlyUsed): remove-se a linha que a mais tempo não é referenciada. Exige implementação de um contador para cada linha. Quando um *hit* ocorre na linha seu contador é zerado enquanto todos os demais são incrementados. Quando for necessário substituir uma linha, será retirada aquela cujo contador tiver o valor mais alto.

Uma vez que todas as solicitações feitas **à** memória são realizadas através da cache, caso a CPU fizer a escrita de uma posição que estiver armazenada na cache, se esta alteração não for repassada para a memória principal, pode-se perder a atualização quando a linha da cache for substituída. Para evitar este problema pode-se adotar duas estratégias:

Escrita em ambas (Write-Through): cada escrita na cache é imediatamente repetida na memória principal;

Escrita no retorno (Write-Back): as escritas são feitas apenas na cache, mas ela será escrita na MP quando for substituída. Pode-se escrevê-la mesmo se não foi alterada ou somente se tiver sido modificada.

Com a política *write-through* pode haver uma grande quantidade de escritas desnecessárias na memória principal, com natural redução do desempenho do sistema. Já a política *write-back* minimiza esta desvantagem, porém a memória principal fica potencialmente desatualizada para utilização por outros dispositivos a ela ligados, como módulos de E/S, o que os obriga a acessar dados na cache.

Uma vez que tanto os dados como as instruções são mantidos na memória principal, ambos tiram proveito da memória cache. Pode-se adotar cache unificada ou separada para cada tipo de conteúdo. A solução que considera uma **cache unificada** (instruções e dados usando a mesma cache) é mais simples de projetar e estabelece automaticamente um equilíbrio entre as buscas de instruções e as buscas de dados. No entanto, a tendência atual é para projetos que usem **caches divididas,** uma cache para instruções e outra para dados. Esse projeto é conhecido como arquitetura de Harvard**,** uma alusão ao computador Mark III projetado por Howard Aiken que tinha memórias diferentes para instruções e para dados. O uso cada vez mais difundido dos processadores pipeline é o principal responsável por levar os projetistas na direção da arquitetura de Harvard. A unidadede busca das instruções faz o acesso às instruções ao mesmo tempo que a unidade de busca

de operandos faz o acesso aos dados necessários à execução de outra instrução buscada anteriormente. A adoção do modelo de duas caches permite acesso em paralelo a instruções e a dados, enquanto o modelo da cache unificado não permite. Além disso, considerando-se que as instruções não são modificadas durante a execução de um programa, o conteúdo da memória de instruções não precisa ser escrito de volta na memória principal. A figura 7.2 ilustra a arquitetura de cache Harvard, onde caches são implementadas de maneira separadas.

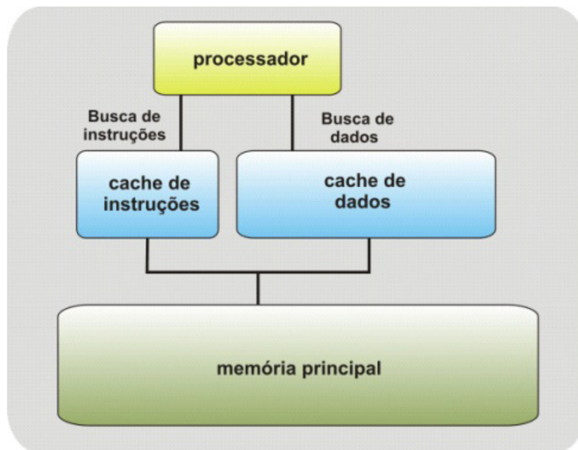


Figura 7.2: Arquitetura Harvard.
Figura adaptada de [Carter, N.]

7.3 Associatividade

A associatividade de uma cache determina quantas posições dentro dela podem conter um dado endereço de memória. Caches com alta associatividade permitem que cada endereço seja armazenado em muitas posições na cache, o que reduz as faltas de cache causadas por conflitos entre linhas que precisam ser armazenadas no mesmo conjunto de posições. Caches com baixa associatividade restringem o número de posições nos quais um endereço pode ser colocado, o que

aumenta o número de faltas, mas simplifica o hardware da cache, reduzindo a quantidade de espaço ocupado por ela e reduzindo o tempo de acesso.

Cache com Mapeamento Completamente Associativo

As caches com mapeamento completamente associativo, ou simplesmente caches associativas permitem que qualquer endereço seja armazenado em qualquer linha da cache. Quando uma operação de memória é enviada à cache, o endereço da solicitação precisa ser comparado a cada entrada na matriz de etiquetas para determinar se os dados referenciados pela operação estão contidos nela.

Cache com Mapeamento Direto

As caches com mapeamento direto são o extremo oposto das associativas. Nela, cada endereço de memória só pode ser armazenado em uma posição da cache. Este método é o mais simples, sendo cada bloco da memória principal mapeado em uma única linha da cache. Para que isso seja feito podemos pensar no seguinte mecanismo:

$$i = j \text{ módulo } m$$

Onde i é o número da linha da cache; j é o número do bloco da memória principal e; m é o número de linhas da cache. Assim, podemos implementar esta forma de mapeamento de maneira muito fácil, usando também o endereço da informação na memória principal.

Como ilustra a figura 7.3, quando uma operação de memória é enviada a uma cache com mapeada diretamente, um subconjunto dos bits do endereço é utilizado para selecionar a linha da cache que pode conter o endereço e outro subconjunto de bits é utilizado para selecionar o byte dentro de uma linha da cache para o qual o endereço

aponta. Em geral, os *n* bits menos significativos no endereço são utilizados para determinar a posição do endereço dentro de sua linha de cache, onde *n* é o *log* na base 2 do número de bytes na linha. Os *m* bits mais significativos seguintes, onde *m* é o log_2 do número de linhas na cache, são utilizados para selecionar a linha na qual o endereço pode ser armazenado, como mostra a figura 7.4.

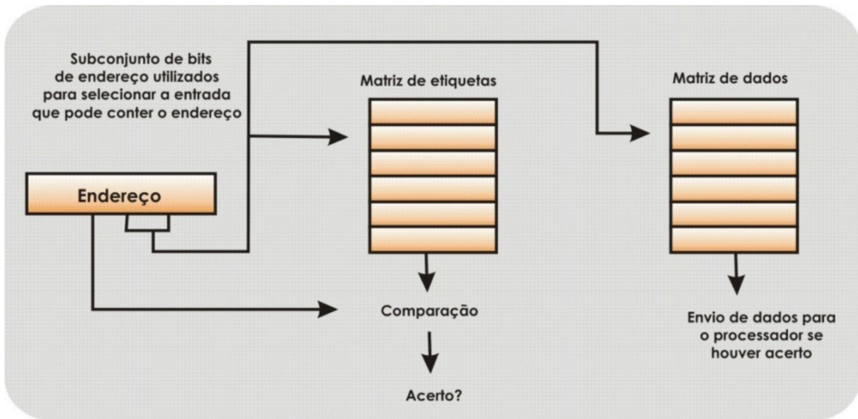


Figura 7.3: Cache com mapeamento direto.
Figura adaptada de [Carter, N.]

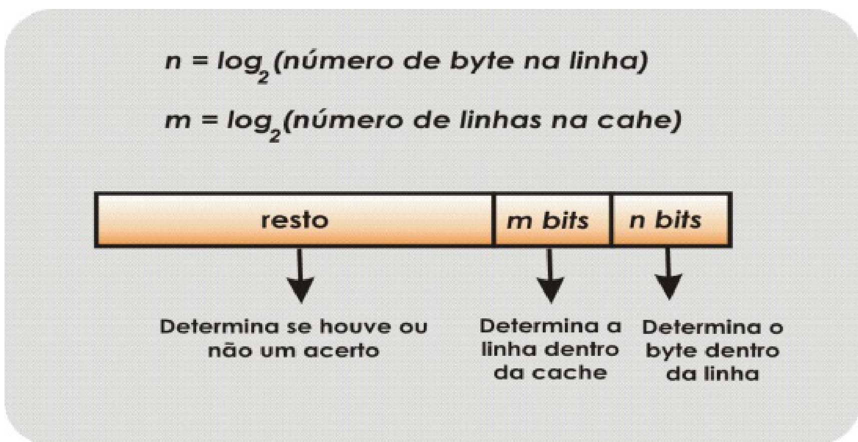


Figura 7.4: Desmembramento do endereço.
Figura adaptada de [Carter, N.]

Exemplo:

Em uma cache com mapeamento direto, com capacidade de 16 Kbytes e comprimento de linha de 32 bytes, quantos bits são utilizados para determinar, dentro de uma linha de cache, o byte ao qual uma operação de memória faz referência, e quantos bits são utilizados para selecionar a linha que pode conter os dados dentro da cache?

Solução:

log 32 na base 2 = 5, de modo que são necesários 5 bits para determinar qual p byte, dentro de uma linha de cache, está sendo referenciado. Com linhas de 32 bits, existem 512 linhas na cache de 16 Kbytes, de modo que são necessários 9 bits para selecionar qual a linha pode conter o endereço (log de 512 na base 2 = 9).

Exemplo retirado de [Carter, N.]

Caches mapeadas diretamente têm a vantagem de necessitar de uma área significativamente menor para ser implementada fisicamente no chip que as caches associativas, porque requerem apenas um operador para determinar se houve um acerto. Se o endereço do item solicitado for igual ao endereço associado àquela posição de memória, o *cachehit* está caracterizado. Caso contrário, como nesta organização o item não pode estar em nem uma outra posição, o *cache miss* é caracterizado. Além disso, caches mapeadas diretamente têm tempos de acesso menores porque exige uma só comparação a ser examinada para determinar se houve um acerto, enquanto que as caches associativas precisam examinar cada uma das comparações e escolher a palavra de dados adequada para enviar ao processador.

No entanto, caches mapeadas diretamente tendem a ter taxas de acerto menores do que as associativas, devido aos conflitos entre linhas que são mapeadas na mesma área da cache. Cada endereço só pode ser

colocado em um local, o que é determinado pelos *m* bits de endereço, como o ilustrado na figura 2.4. Se dois endereços têm o mesmo valor naquele bit, eles serão mapeados sobre a mesma linha da cache e não poderão residir ao mesmo tempo. Um programa que alterasse referências a estes dois endereços nunca geraria um acerto, uma vez que a linha contendo o endereço estaria sempre sendo descartada antes da próxima referência ao endereço. Assim, a cache poderia ter uma taxa de acerto igual a 0%, ainda que o programa fizesse referência a apenas dois endereços. Na prática, caches mapeadas diretamente, em especial de grande capacidade, podem obter boas taxas de acerto, embora tendam a ser menores que àquelas que fornecem várias posições possíveis para cada linha de dados.

Em resumo, as vantagens deste tipo de organização incluem:

- simplicidade de hardware, e consequentemente custo mais baixo;
- não há necessidade de escolher uma linha para ser retirada da *cache*, uma vez que há uma única opção; e
- operação rápida

Apesar de simples, esta estratégia é pouco utilizada devido à sua baixa flexibilidade.

Como há apenas uma posição na cache onde o item solicitado pode estar localizado, a probabilidade de presença na cache pode ser baixa. Desse modo, pode haver queda sensível no desempenho em situações as quais o processador faz referências envolvendo itens que são mapeados para uma mesma posição da cache.

Cache com Mapeamento Set-Associativo

As caches com mapeamento set-associativo ou grupo associativo são uma combinação de caches associativas e mapeadas diretamente. Nela existe um número fixo de posições (chamadas de conjuntos ou grupos), nas quais um dado endereço pode ser armazenado. O número de posições em cada conjunto é a associatividade da cache.

Este tipo de organização oferece um compromisso entre o desempenho da organização por mapeamento direto e a flexibilidade da organização completamente associativa. A cache particionada em *N* conjuntos, onde cada conjunto pode conter até *K* linhas distintas, onde *K* é a associatividade da cache; na prática, $1 < K <= 16$.

Uma dada linha só pode estar contida em um único conjunto, em uma de suas *K* linhas. O conjunto selecionado é determinado a partir do endereço do item sendo referenciado. O endereço de um item é particionado em três segmentos:

- *Offset*: se uma linha com *L* bytes contém *W* palavras, então os $log_2 W$ bits menos significativos identificam qual dos itens em uma linha está sendo buscado. Se o menor item endereçável for um byte, então $L = W$.

- *Lookup*: os $log_2 N$ bits seguintes identificam em qual dos *N* conjuntos, de 0 a *N-1*, a linha deve ser buscada.

- *Tag:* os bits mais significativos que restam do endereço servem como identificação da linha, que é armazenada no diretório do cache.

Por exemplo, considere uma cachede 4 Kbytes com linhas de 32 bytes, onde o menor item endereçável fosse um byte e os endereços gerados pelo processador sejam de 24 bits. Se a associatividade da cache K = 4, então esta cache tem espaço para 512 conjuntos. Portanto, o tamanho do *tag* deverá ser 10 bits. O controlador da cachê deve ser capaz de fazer a busca de uma linha em um tempo próximo a um tempo de ciclo dos dispositivos de memória rápida utilizados no módulo cache. A estratégia para tanto é comparar o valor de *lookup* simultaneamente com os *K* possíveis candidatos do conjunto selecionado (usando *K* comparadores). Ao mesmo tempo, *K* linhas são acessadas dos dispositivos de memória e trazidas para *K* registros rápidos — caso a linha buscada esteja presente no cache, ela é simplesmente selecionada de um destes registros. Caso contrário, um sinal de *cache miss* é ativado.

Para facilitar o entendimento, a figura 7.5 mostra uma cache set-associativo de dois caminhos, na qual existem duas posições para cada endereço. De modo semelhante a uma cache mapeada diretamente, um subconjunto dos bits de endereço é utilizado para selecionar o conjunto que pode conter o endereço. A cache set-associativo de dois caminhos implica no fato de existirem duas etiquetas que podem ser comparadas com o endereço de referência de memória para determinar se houve ou não um acerto. Se qualquer uma das etiquetas for igual ao endereço, houve um acerto e a matriz de dados é selecionada. Caches com maior associatividade têm estruturas semelhantes, possuindo apenas mais comparadores para determinar se houve ou não um acerto.

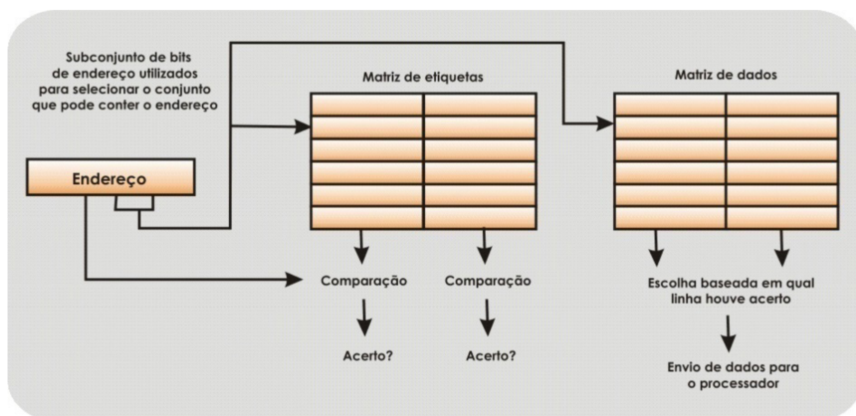


Figura 7.5: Cache com mapeamento set-associatvo de dois caminhos.
Figura adaptada de [Carter, N.]

Uma cache set-associativo, por agrupar linhas em conjuntos, necessita menos bits para identificar conjuntos do que seria necessário para identificar individualmente as linhas onde o endereço pode ser armazenado. O número de conjuntos de uma cache pode ser obtido calculando-se o número de linhas e dividindo-o pela associatividade.

Exemplo:

Quantos conjuntos existem em uma cache set-associativo de dois caminhos, com capacidade de 32 Kbytes e linhas de 64 bytes, e quantos bits de endereço são utilizados para selecionar um conjunto nesta cache? E em uma cache set-associativo de oito caminhos, com a mesma capacidade e comprimento de linha?

Solução: *Uma cache de 32 Kbytes, com linhas de 64 bytes, contém 512 linhas de dados. Em uma cache set-associativo de dois caminhos, cada conjuntos contém duas linhas, de modo que existem 256 conjuntos. Log na base 2 de 256 é igual a oito, de modo que são utilizados 8 bits de um*

endereço para selecionar um conjunto mapeado por um endereço. A cache set-associativo de oito caminhos tem 64 linhas e utiliza seis bits do endereço para selecionar o conjunto.

Exemplo retirado de [Carter, N.]

Em geral, caches set-associativo têm taxas de acerto melhores que as mapeadas diretamente, mas taxas de acerto piores que caches associativas do mesmo tamanho. As caches set-associativo, por permitir que cada endereço seja armazenado em vários locais, eliminam alguns dos conflitos que ocorrem com caches com mapeamento direto. A diferença nas taxas de acertos é uma função da capacidade da cache, do grau de associatividade e dos dados referenciados por um programa. Alguns programas fazem referência a grandes blocos de dados contíguos, conduzindo a poucos conflitos, enquanto outros fazem referência a dados dispersos, o que pode conduzir a conflitos, se os endereços de dados mapearem o mesmo conjunto na cache.

Quanto maior é uma cache, menor é o benefício que ela tende a ter da associatividade, uma vez que existe uma probabilidade menor que dois endereços sejam mapeados para a mesma área. Ir de uma cache diretamente mapeada para uma set-associativo de dois caminhos normalmente causa reduções significativas na taxa de faltas. Aumentar para quatro caminhos tem um efeito menos significativo, e crescer além disso tende a ter pouco efeito, exceto para caches extremamente pequenas. Por esse motivo, caches set-associativo de dois caminhos são mais comuns nos microprocessadores atuais.

7.4 Políticas de Substituição

Ocorre quando uma linha precisa ser descartada de uma cache para abrir espaço para os dados que estão entrando, ou porque a cache está cheia, ou ainda por causa de conflitos relacionados com um conjunto. A política de substituição determina qual linha será descartada.

Em caches mapeadas diretamente não existe escolha, uma vez que alinha está entrando só pode ser colocada em uma única posição, mas caches set-associativas e associativas contém várias linhas que poderiam ser descartadas para abrir espaço para a linha que está entrando. Nessas caches, o objetivo geral da política de substituição é minimizar as faltas futuras da cache ao descartar uma linha que não será referenciada no futuro.

Vários algoritmos são implementados como políticas de substituição de páginas. Por exemplo, temos:

> ***Algoritmo Ótimo:*** proposto por Belady em 1966, é o que apresenta o melhor desempenho computacional e o que minimiza o número de faltas de páginas. No entanto, sua implementação é praticamente impossível. A ideia do algoritmo é retirar da memória a página que vai demorar mais tempo para ser referenciada novamente. Para isso, o algoritmo precisaria saber, antecipadamente, todos os acessos à memória realizados pela aplicação, o que é impossível em um caso real. Por estes motivos, o algoritmo ótimo só é utilizado em simulações para se estabelecer o valor ótimo e analisar a eficiência de outras propostas elaboradas.

FIFO (First-in, First-out): é um algoritmo de substituição de páginas de baixo custo e de fácil implementação que consiste em substituir a página que foi carregada há mais tempo na memória (*aprimeira página a entrar é a primeira a sair*). Esta escolha não leva em consideração se a página está sendo muito utilizada ou não, o que não é muito adequado pois pode prejudicar o desempenho do sistema. Por este motivo, o FIFO apresenta uma deficiência denominada *anomalia de Belady*: a quantidade de falta de páginas pode aumentar quando o tamanho da memória também aumenta. Por estas razões, o algoritmo FIFO puro é muito pouco utilizado. Contudo, sua principal vantagem é a facilidade de implementação: uma lista de páginas ordenada pela "idade". Dessa forma, na ocorrência de uma falta de página, a primeira página da lista será substituída e a nova será acrescentada ao final da lista.

LRU (Least Recently Used): é um algoritmo de substituição de página que apresenta um bom desempenho substituindo a página *menos recentemente usada*. Esta política foi definida baseada na seguinte observação: se a página está sendo intensamente referenciada pelas instruções é muito provável que ela seja novamente referenciada pelas instruções seguintes e, de modo oposto, aquelas que não foram acessadas nas últimas instruções também é provável que não sejam acessadas nas próximas. Apesar de o LRU apresentar um bom desempenho, ele também possui algumas deficiências quando o padrão de acesso é sequencial (em estruturas de dados do tipo vetor, lista, árvore), dentro de loops, etc. Diante dessas deficiências foram

propostas algumas variações do LRU, dentre elas destacamos o LRU-K. Este algoritmo não substitui aquela que foi referenciada há mais tempo e sim quando ocorreu seu *k-último* acesso. Por exemplo, LRU-2 substituirá a página que teve seu penúltimo acesso feito há mais tempo, e LRU-3 observará o antepenúltimo e assim por diante. A implementação do LRU também pode ser feita através de uma lista, mantendo as páginas mais referenciadas no início (cabeça) e as menos referenciadas no final da lista. Portanto, ao substituir, retira-se a página que está no final da lista. O maior problema com esta organização é que a lista deve ser atualizada a cada nova referência efetuada sobre as páginas, o que torna alto o custo dessa manutenção.

- MRU (Most Recently Used): faz a substituição da última página acessada. Este algoritmo também apresenta algumas variações ao LRU. Por exemplo, o MRU-n escolhe a n-última página acessada para ser substituída. Dessa forma, é possível explorar com mais eficiência o princípio de localidade temporal apresentada pelos acessos.

- NRU (Not Recently Used): procura por páginas que não foram referenciadas nos últimos acessos para serem substituídas. Tal informação é mantida através de um bit. Este algoritmo também verifica, através de um bit de modificação, se a página teve seu conteúdo alterado durante sua permanência em memória. Esta informação também vai ajudar a direcionar a escolha da página. As substituições das páginas seguem

a seguinte prioridade: páginas não referenciadas e não modificadas, páginas não referenciadas, páginas não modificadas e páginas referenciadas e modificadas.

LFU (LeastFrequentlyUsed): escolhe a página que foi menos acessada dentre todas as que estão carregas em memória. Para isso, é mantido um contador de acessos associado a cada página (*hit*) para que se possa realizar esta verificação. Esta informação é zerada cada vez que a página deixa a memória. Portanto, o problema desse algoritmo é que ele prejudica as páginas recém-carregadas, uma vez que, por estarem com o contador de acessos zerado, a probabilidade de serem substituídas é maior. Qual uma possível solução para este problema? (Estabelecer um tempo de carência) Só páginas fora desse tempo é que podem ser substituídas. Tal estratégia deu origem ao algoritmo FBR (*Frequency-BasedReplacement*).

MFU (Most Frequently Used): substitui a página que tem sido mais referenciada, portanto, o oposto do LFU. O controle também é feito através de contadores de acesso. O maior problema deste algoritmo é que ele ignora o *princípio de localidade temporal*.

WS (Working Set): possui a mesma política do LRU. No entanto, este algoritmo não realiza apenas a substituição de páginas ele também estabelece um tempo máximo que cada página pode permanecer ativa na memória. Assim, toda página que tem seu tempo de permanência esgotado ela é retirada da me-

mória. Portanto, o número de páginas ativas é variável. O WS assegura que as páginas pertencentes ao *working set* processo permanecerão ativas em memória.

Os algoritmos apresentados são alguns dos disponíveis na literatura. Outras implementações ou variações dos destacados também podem ser encontradas. Deixamos essa tarefa a cargo do leitor.

7.5 Políticas de Atualização

Como já discutido, níveis em hierarquia de memória podem usar uma política *write-back* ou de *write-through* para tratar o armazenamento. Se um nível de hierarquia é *write-through*, quando a operação de escrita é executada, os valores armazenados são escritos no nível e enviados para o nível seguinte mais abaixo. Isso assegura que o conteúdo do nível e do próximo nível será sempre o mesmo.

As caches podem ser implementadas tanto como sistemas *write-back* quanto *write-through* e ambas as abordagens tem suas vantagens. As caches *write-through* têm a vantagem de que não é necessário registrar quais linhas foram escritas. Como os dados em uma cache *write-through* são sempre consistentes com o conteúdo do próximo nível, descartar uma linha pode ser feito escrevendo-se a nova linha sobre a velha, reduzindo o tempo para trazer uma linha para dentro da cache. Em contraste, caches *write-back* escrevem seu conteúdo apenas quando uma linha é descartada.

Se uma dada linha recebe várias solicitações de armazenamento enquanto ela está na cache, esperar até que a linha seja descartada pode reduzir significativamente o número de escritas enviadas para o próximo nível da hierarquia da cache. Este efeito pode ainda ser mais

importante, se o próximo nível da hierarquia for implementado com DRAM em modo de página, uma vez que a cache *write-back* pode usar o modo de página para reduzir o tempo de escrita uma linha no nível seguinte. No entanto, caches *write-back* exigem um hardware para manter controle sobre se cada linha foi escrita ou não desde que foi copiada.

Write-Through

Na estratégia *write-through*, quando um ciclo de escrita ocorre para uma palavra, ela é escrita na *cache* e na memória principal simultaneamente. A principal desvantagem desta estratégia é que o ciclo de escrita passa a ser mais lento que o ciclo de leitura. No entanto, em programas típicos a proporção de operações de escrita à memória é pequena, de 5 a 34% do número total de referências à memória.

Para analisar como fica o tempo efetivo de acesso para uma cache com a estratégia *write-through* com alocação em escrita, considere o seguinte modelo. Seja t_c o tempo de acesso a cache, t_l o tempo de transferência para uma linha da memória principal para a cache e t_m o tempo de acesso a uma palavra da memória principal. Seja ainda *h* a probabilidade do item referenciado estar (este "estar" está estranho aqui. Reveja!) na memória, e w a fração das referências à memória que correspondem a operações de escrita. As quatro situações que podem ocorrer são:

- leitura de item presente no cache: em tempo tc com probabilidade h x (1-w);
- leitura de item ausente do cache: em tempo tc + tlcom probabilidade (1-h) x (1-w);
- atualização de item presente na cache: em tempo tm

uma vez que a atualização na cache ocorre concorrentemente com a atualização em memória, o tempo t_c está escondido no tempomaior, t_m. Esta situação ocorre com probabilidade $h \times w$;

- atualização do item ausente da cache: em tempo $t_m + t_l$ com probabilidade $(1-h) \times w$.

A partir destas possibilidades, o tempo efetivo de acesso a cache é:

$$t_{ef} = (1-w)\, t_c + (1-h)\, t_l + w\, t_m$$

Write-Back

Na estratégia *write-back*, quando um ciclo de escrita ocorre para uma palavra, ela é atualizada apenas na cache. A linha em que a palavra ocorre é marcada como alterada. Quando a linha for removida da cache, a linha toda é atualizada na memória principal. A desvantagem desta estratégia está no maior tráfego entre memória principal e cache, pois mesmo itens não modificados são transferidos da cache para a memória.

Considere wb a probabilidade de uma linha da cache ter sido atualizada. Em geral, $wb < w$ uma vez que uma linha pode conter mais de uma palavra atualizada. Então, a partir de uma análise similar à realizada acima, obtém-se o tempo efetivo de acesso a cache:

$$t_{e_} = t_c + (1-h)(1-w_b)\, t_l$$

Em geral, caches *write-back* têm um desempenho melhor que as *write-through*. Isso ocorre porque uma linha que é escrita uma vez tem uma probabilidade grande de ser escrita inúmeras vezes, dessa forma o custo (em tempo) de escrever uma linha inteira no nível seguinte

na hierarquia de memória é menor do que escrever as modificações à medida que elas ocorrem. Sistemas mais antigos frequentemente utilizam caches *write-through* por causa da sua complexidade mais baixa de controle, mas as *write-back* tornaram-se dominantes nos computadores modernos. A figura 7.6 ilustra a diferença entre as duas políticas de atualização de memória apresentadas.

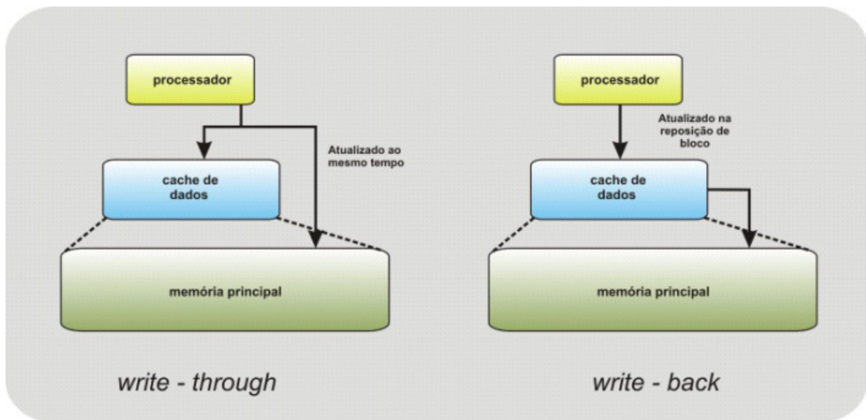


Figura 7.6: Write-Through e Write-Back.

7.6 Caches em Vários Níveis

Em muitos sistemas, mais de um nível de hierarquia é implementado como uma cache, como mostra a figura 7.7. Quando isto é feito, o mais comum é que o primeiro nível de cache (mais próximo ao processador) seja implementado como caches distintas para dados e para instruções, enquanto os outros níveis sejam implementados como caches unificadas. Isso dá ao processador largura de banda adicional proporcionada por uma arquitetura Harvard no nível superior do sistema de memória, ao mesmo tempo em que simplifica o projeto dos níveis mais baixos.

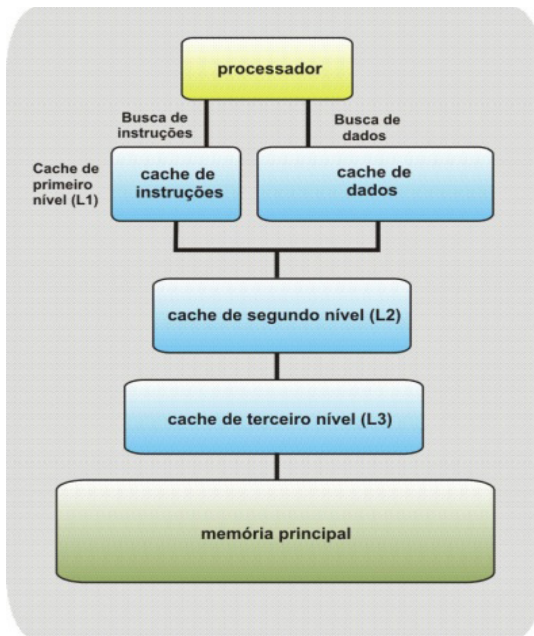


Figura 7.7: Hierarquia de caches em vários níveis.
Figura adaptada de [Carter, N.]

Para que uma cache de vários níveis melhore significativamente o tempo médio de acesso à memória de um sistema, cada nível precisa ter uma capacidade maior do que o nível acima dele na hierarquia, porque a localidade de referência vista em cada nível decresce à medida que se vai a níveis mais profundos na hierarquia. (Solicitações a dados referenciados recentemente são tratados pelos níveis superiores do sistema de memória, de modo que a solicitação feita aos níveis mais baixos tendem a ser mais distribuídos pelo espaço de endereçamento.) Caches com capacidades maiores tendem a ser mais lentas, de modo que o benefício de velocidade de caches distintas para dados e instruções não é tão significativo nos níveis mais baixos da hierarquia de memória, o que representa um outro argumento a favor de caches unificadas para estes níveis.

No início da década de 90, a hierarquia mais comum para computadores pessoais e para estações de trabalho era a cache de primeiro nível (L1) ser relativamente pequena e localizada no mesmo chip que o processador. Caches de nível mais baixo eram implementadas fora do chip, com o uso de SRAMs. Capacidades de 4 a 16 Kbytes não eram incomuns em caches L1, com caches L2 atingindo 64 a 256 Kbytes. À medida que o número de transistores que podia ser integrado em um chip de memória aumentou, níveis adicionais de cache foram movidos para dentro do chip do processador. Muitos sistemas atuais **têm** tanto caches de primeiro quanto de segundo nível no mesmo chip que o processador, ou ao menos no mesmo encapsulamento. Caches de terceiro nível são frequentemente implementadas externamente e podem ter vários megas de tamanho. Espera-se que nos próximos poucos anos este nível também seja integrado no encapsulamento do processador.

7.7 EXERCÍCIOS

1. O que motivou a criação da memória cache?

2. Diferencie Localidade Temporal de Localidade Espacial.

3. As caches podem ser unificadas ou separadas. Explique as vantagens de cada uma delas e mostre a arquitetura Harvard.

4. Discorra sobre: cache com mapeamento associativo, com mapeamento direto e com mapeamento set-associativo. Explique o processo de tradução de endereços e comente sobre as vantagens e desvatagens de cada uma delas.

5. Explique as políticas de substituição de páginas: FIFO, LRU, MRU, NRU, LFU, MFU E WS. Considere a seguinte string de referência de página: 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6. Simule a funcionalidade dos algoritmos listados e diga quantas faltas de página ocorrem em cada um, considerando uma memória com três páginas.

6. É possível afirmar que para um conjunto de aplicações X o algoritmo de substituição de páginas A seja mais adequado, e para um conjunto de aplicações Y o algoritmo de substituição de páginas B seja mais eficiente? Isso é verdade? Sim ou não? Justifique.

7. Diferencie as políticas de atualização write-back e write-through comentando as vantagens e desvantagens de cada uma.

8. Explique as vantagens de utilizar caches em vários níveis e mostre uma representação gráfica desta hierarquia.

8 MEMÓRIA VIRTUAL

8.1 Introdução

O custo da memória foi uma limitação significativa nos primeiros sistemas de computador. Antes do desenvolvimento de DRAMs baseadas em semicondutores, a tecnologia de memória predominante era a memória de núcleo, nas quais anéis toroidais de material magnético eram utilizados para armazenar cada bit de dados. O custo para produzir e montar esses anéis toroidais, como dispositivos de memória, conduziu a capacidades de memória limitadas em muitas máquinas.

Para resolver este problema, foi desenvolvida a **memória virtual**. Em um sistema de memória virtual, discos rígidos e outros meios magnéticos formam a camada inferior da hierarquia de memória, com as DRAMs formando o nível principal da hierarquia. Como programas não podem acessar diretamente dados armazenados em meios magnéticos, a área de endereço de um programa é dividida em **páginas**, blocos contíguos de dados que são armazenados em mídia magnética. Quando é feita referência a uma página de dados, o sistema a copia para a memória principal, permitindo que ela seja acessada.

A figura 8.1 ilustra muitos dos conceitos chaves de memória virtual. Cada programa tem seu próprio espaço de endereços virtuais, que é um conjunto de endereços que os programas utilizam para carregar e armazenar operações. O espaço de endereço físico é o conjunto de endereços que foi utilizado para fazer referência a posições na memória principal, e os termos endereço virtual e endereço físico são utilizados para descrever endereços que estão nos espaços de endereço virtual e físico. O espaço de endereços virtuais é dividido em páginas, algumas das quais são copiadas para quadros (janelas na memória principal

onde uma página de dados pode ser armazenada) porque recentemente foi feita referências a elas e algumas das quais estão residentes apenas no disco. As páginas são sempre alinhadas com relação a um múltiplo do tamanho da página, de modo que elas nunca se sobrepõem. Os termos página virtual e página física são utilizados para descrever uma página de dados nos espaços de endereços virtual e físico. Páginas que foram carregadas da memória a partir do disco são ditas terem sido mapeadas para dentro da memória principal.

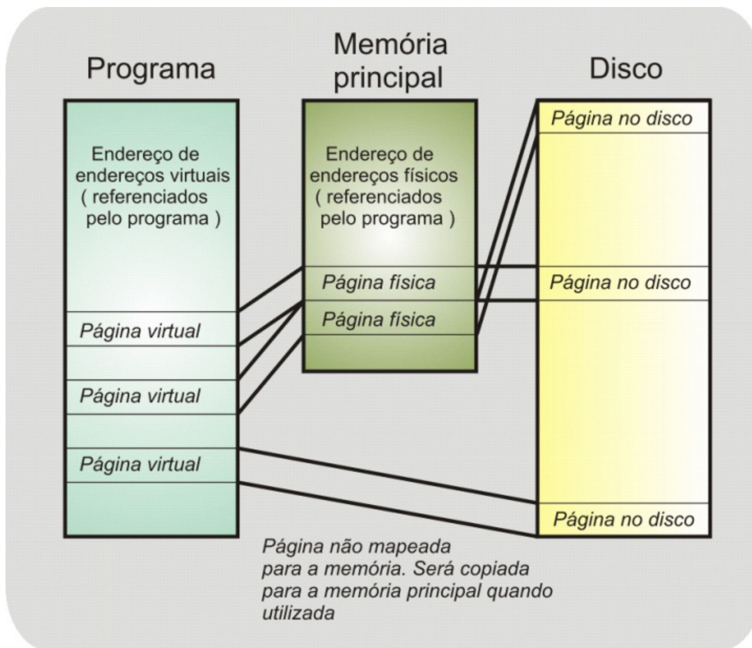


Figura 8.1: Memória virtual.
Figura adaptada de [Carter, N.]

A memória virtual permite que um computador aja como se a memória principal fosse muito maior do que ela é realmente. Quando um programa faz referência a um endereço virtual, ele não pode dizer, a não ser medindo a latência da operação, se um endereço virtual es-

tava residente na memória principal, ou se ele teve que ser buscado no meio magnético. Assim, o computador pode mover páginas de/para a memória principal, quando necessário, de modo semelhante como as linhas de cache são trazidas para dentro e para fora da cache, de acordo com a necessidade, permitindo que programas façam referências a mais dados do que podem ser armazenados em um único momento na memória principal.

8.2 Tradução de endereços

Programas que são executados em um sistema de memória virtual utilizam endereços virtuais como argumentos para as instruções de carga e armazenamento, mas a memória principal utiliza endereços físicos para registrar as posições onde os dados estão efetivamente armazenados. Quando um programa executa uma referência à memória, o endereço virtual utilizado precisa ser convertido para o endereço físico equivalente, um processo conhecido como tradução de endereços. A figura 8.2 mostra um fluxograma de tradução de endereços.

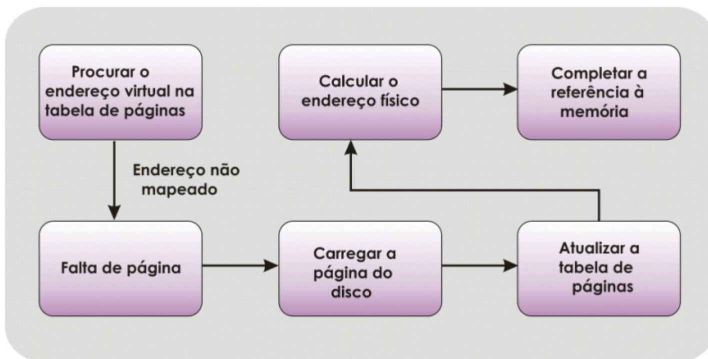


Figura 8.2: Tradução de endereços.
Figura adaptada de [Carter, N.]

Quando um programa de usuário executa uma instrução que

faz referência à memória, o sistema operacional faz acesso à tabela de páginas, uma estrutura de dados na memória que mantém o mapeamento dos endereços virtuais para os físicos, de modo a determinar se uma página virtual contém o endereço referenciado pela operação, ou se está ou não atualmente mapeada sobre uma página física. Se isto for verdade, o sistema operacional determina o endereço físico que corresponde ao endereço virtual, a partir da tabela de páginas, e a operação continua utilizando o endereço físico para fazer acesso à memória principal.

Se a página virtual que contém o endereço referenciado não está atualmente mapeada sobre uma página física, ocorre uma falta de página e o sistema operacional busca a página que contém os dados necessários na memória, carregando-a para uma página física e atualizando a tabela de páginas com a nova tradução. Uma vez que a página tenha sido lida para a memória principal a partir do disco e a tabela de páginas tenha sido atualizada, o endereço físico da página pode ser determinado e a referência à memória pode ser completada. Se todas as páginas físicas no sistema já contêm dados, o conteúdo de uma delas deve ser transferido para meio magnético para abrir espaço para a página que será carregada. As políticas de substituição utilizadas para escolher a página física que será transferida são semelhantes àquelas discutidas no Capítulo anterior.

Como tanto páginas virtuais quanto físicas são sempre alinhadas sobre um múltiplo do seu tamanho, a tabela de páginas não precisa armazenar completamente o endereço virtual, nem o endereço físico. Ao invés disso, os endereços virtuais são divididos em um identificador de página virtual, chamado número de página virtual ou NPV, e o conjunto de bits que descreve o deslocamento a partir do início da

página virtual até o endereço virtual. Páginas físicas são divididas de modo semelhante em número de páginas físicas (NPF) e um deslocamento a partir do início da página física até o endereço físico, como mostra a figura 8.3.

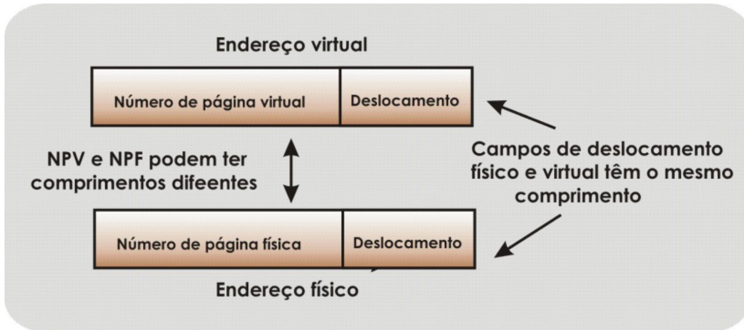


Figura 8.3: Endereços físicos e virtuais.
Figura adaptada de [Carter, N.]

As páginas virtual e física de um dado sistema são geralmente do mesmo tamanho, de modo que o número de bits (log_2 do tamanho da página) necessário para manter o campo de deslocamento dos endereços físicos e virtuais é o mesmo, embora o número de bits empregados para identificar o NPV e o NPF podem ter comprimentos diferentes. Muitos sistemas, em especial sistemas de 64 bits, têm endereços virtuais mais longos que os endereços físicos, dada a impraticabilidade atual de se construir um sistema com 2^{64} bytes de memória DRAM.

Quando um endereço virtual é traduzido, o sistema operacional procura pela entrada correspondente ao NPV na tabela de páginas e retorna o NPF correspondente. Os bits de deslocamento do endereço virtual são, então, concatenados ao NPF para gerar o endereço físico, como mostra a figura 8.4.

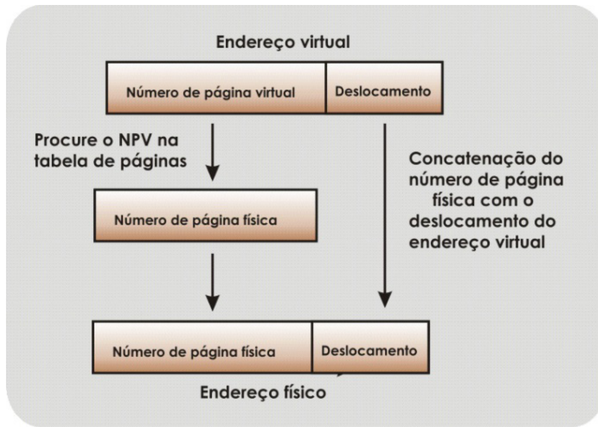


Figura 8.4: Convertendo endereços virtuais em físicos.
Figura adaptada de [Carter, N.]

8.3 Swapping

O sistema de memória virtual recém descrito é um exemplo de paginação por demanda atualmente o tipo de memória virtual mais comumente utilizado. Na paginação por demanda, as páginas de dados só são trazidas para a memória principal quando acessadas por um programa. Quando uma mudança de contexto ocorre, o sistema operacional não copia qualquer página do programa antigo para o disco ou do novo programa para a memória principal. Em vez disso, ele apenas começa a executar o novo programa, buscando as páginas à medida que são referenciadas.

Swapping é uma técnica relacionada que utiliza meio magnético para armazenar programas que não estão sendo atualmente executados no processador. Em um sistema que utiliza swapping, o sistema operacional trata todos os dados de um programa como uma unidade atômica e move todos os dados para dentro e para fora da memória principal em uma operação.

Quando um sistema operacional em um computador que utiliza swapping escolhe um programa para ser executado no processador, ele carrega todos os dados do programa na memória principal. Caso o espaço não disponível em memória não seja suficiente, o sistema operacional transfere outros programas para o disco, liberando assim espaço de memória.

Se todos os programas que estão sendo executados em um computador cabem na memória principal (contando tanto as suas instruções como os dados), tanto a paginação por demanda quanto o swapping permitem que o computador opere em modo multiprogramado sem ter que buscar dados no disco.

Sistemas de swapping têm a vantagem de que, uma vez que o programa tenha sido buscado no disco, todos os dados do programa estarão mapeados na memória principal. Isto faz com que o tempo de execução de um programa seja mais previsível, uma vez que faltas de página nunca ocorrerão durante a execução.

Sistemas de paginação por demanda têm a vantagem de que eles buscam do disco apenas as páginas que um programa efetivamente utiliza. Se um programa precisa fazer referência a apenas uma parte dos seus dados durante cada fatia do seu tempo de execução, isso pode reduzir significativamente a quantidade de tempo utilizada para copiar dados de/para o disco. Além disso, sistemas que utilizam swapping não podem, utilizar sua capacidade de armazenamento em meio magnético para permitir que um único programa faça referência a mais dados que cabem na memória principal, porque todos os dados de um programa devem ser transferidos para dentro ou para fora da memória principal, como uma unidade.

Em um sistema de paginação por demanda, as páginas indivi-

duais dos dados de um programa podem ser trazidas para dentro da memória, quando necessário, fazendo com que o espaço disponível no disco seja o limite da quantidade máxima de dados a que o programa pode fazer referência. Para a maioria das aplicações, as vantagens da paginação por demanda são maiores que as desvantagens, tornando a paginação por demanda a opção para a maioria dos sistemas operacionais atuais.

8.4 Tabela de Páginas

Como já discutido anteriormente, o sistema operacional utiliza uma estrutura de dados conhecida como tabela de páginas para identificar como os endereços virtuais são mapeados em endereços físicos. Cada programa tem seu próprio mapeamento virtuais-físico, e tabelas de páginas diferentes são necessárias para cada programa no sistema. A implementação mais simples de tabelas de páginas é apenas uma matriz de entrada de tabelas, uma entrada por página virtual, e é conhecida como tabela de páginas de nível único para diferenciá-las das tabelas de páginas multinível, que descreveremos adiante.

Para executar uma tradução de endereço, o sistema operacional utiliza o número de página virtual do endereço como um índice para acessar a matriz de entradas da tabela de páginas, de modo a localizar o número da página física correspondente à página virtual. Isto é mostrado na figura 8.5, que apresenta um sistema cujo espaço de endereços virtuais tem um tamanho de apenas oito páginas.

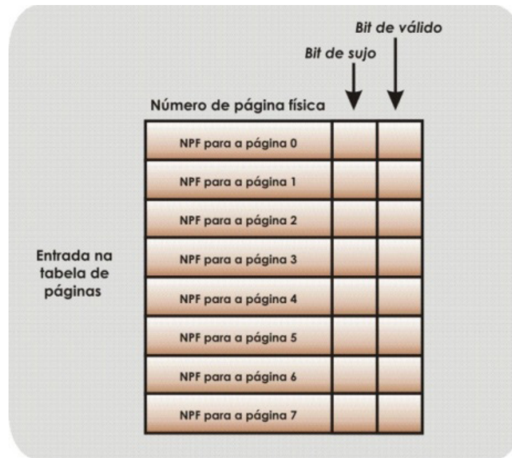


Figura 8.5: Tabela de páginas de nível único.
Figura adaptada de [Carter, N.]

Como ilustrado na figura 8.5 as entradas na tabela de páginas geralmente contêm um número de página física, um bit válido e um bit sujo. O bit válido indica se a página virtual corresponde à entrada está atualmente mapeada na memória física. Se o bit válido está ativo, o campo do número da página física contém o número da página física que contém os dados da página virtual. O bit sujo registra se a página foi ou não modificada desde que ela foi trazida para a memória principal. Isto é utilizado para determinar se o conteúdo da página deve ser escrito de volta no disco quando a página for retirada da memória principal.

A figura 8.6 traz um diagrama de uso de uma tabela de páginas para traduzir um endereço virtual. Sistemas com tabelas de página de nível único geralmente exigem que toda a tabela de páginas seja mantida, o tempo todo, na memória física, de modo que o sistema operacional possa acessar a tabela para traduzir endereços.

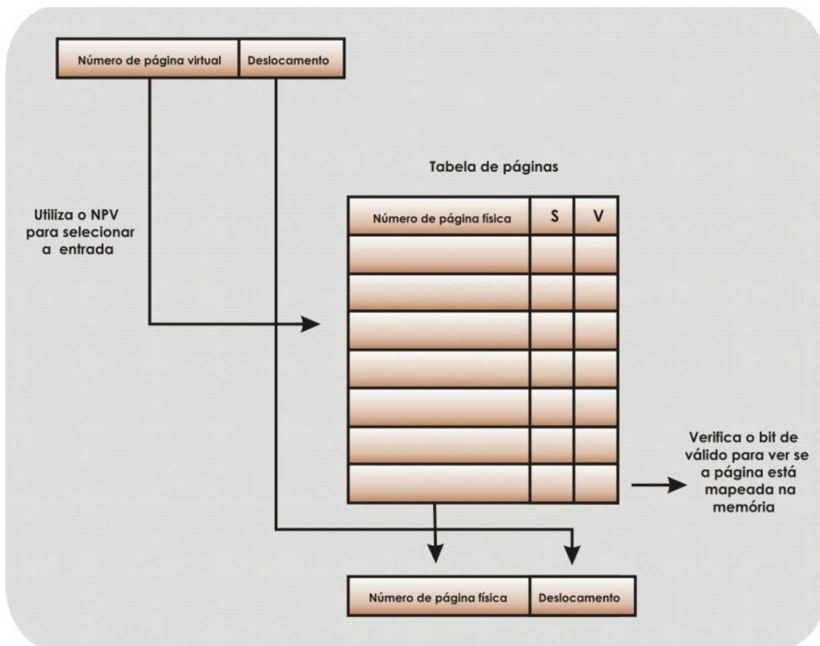


Figura 8.6: Tradução de endereços utilizando tabela de páginas.
Figura adaptada de [Carter, N.]

Tabelas de Páginas Multiníveis

Tabelas de páginas podem exigir um grande espaço de endereçamento. Para resolver este problema, os projetistas utilizam tabelas de páginas com vários níveis, as quais permitem que muito da tabela de páginas seja armazenada na memória virtual e mantida em disco quando não esteja em uso. Em uma tabela de páginas multinível, a própria tabela de páginas é dividida em páginas e distribuída em uma hierarquia. As entradas no nível mais baixo da hierarquia são semelhantes às entradas de um tabela de páginas de nível único, contendo o NPF da página, junto com os bits de válido e sujo. As entradas nos níveis da tabela de páginas identificam a página na memória que contém o próximo nível da hierarquia para o endereço que está sendo traduzido.

Quando se utiliza este sistema, apenas a página que contém o nível mais alto da tabela de páginas precisa ser mantida na memória o tempo todo. Outras páginas da tabela de páginas podem ser copiadas de/para o disco, quando necessário.

Para executar uma tradução de endereço, o NPV de um endereço é dividido em grupos de bits, onde cada grupo contém o número de bits igual ao log2 do número de entradas na tabela de páginas em uma página de dados, como mostrado na figura 8.7. Se o número de bits no NPV não é divisível pelo log2 do número de entrada na tabela de páginas de uma página de dados, é necessário arredondar o número de grupos para cima para o próximo número inteiro.

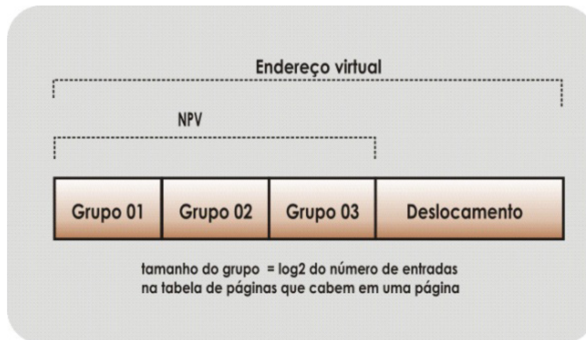


Figura 8.7: Divisão de endereços para tabela de páginas multinível.
Figura adaptada de [Carter, N.]

O número de bits mais significativo é então utilizado para selecionar uma entrada na página de nível mais alto da tabela de páginas. Esta entrada contém o endereço da página de dados que contém o próximo conjunto de entradas a ser pesquisado. O grupo seguinte de ordem menos significativa é então utilizado para indexar a página apontada pela entrada que está no nível superior da tabela de páginas. Este processo é repetido até que o nível mais baixo da tabela de pági-

nas seja encontrado, o qual contém o NPF para a página desejada. A figura 8.8 mostra este processo em um sistema com NPVs de 6 bits e quatro entradas em cada página. Se qualquer uma das páginas na tabela de páginas solicitadas durante a tradução do endereço não estiver mapeada na memória principal, o sistema simplesmente busca no disco e continua a tradução.

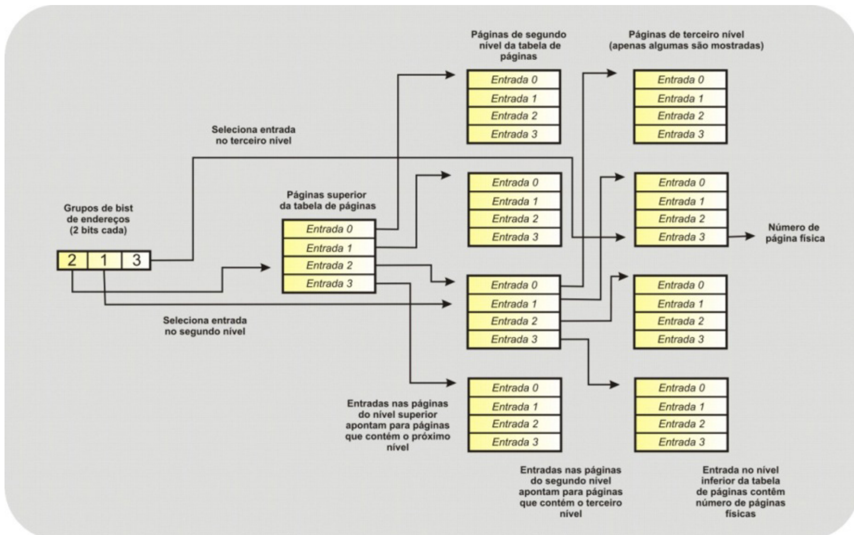


Figura 8.8: Tradução de endereços com tabelas multinível.
Figura adaptada de [Carter, N.]

8.5 TranslationLookaside Buffers - TLBs

Uma das desvantagens principais de usar tabelas de páginas para a tradução de endereços é que a tabela de páginas precisa ser acessada a cada referência feita à memória. Para reduzir esta desvantagem, processadores são projetados para utilizar memória virtual incorporando estruturas especiais denominadas de *Translation Lookaside Buffers (TLBs)* que atuam como caches para a tabela de páginas. Quando um

programa executa uma referência à memória, o endereço virtual é enviado ao TLB para determinar se ele contém a tradução do endereço. Se verdadeiro, o TLB retorna o endereço físico dos dados e a referência à memória continua. Caso contrário, ocorre uma falta no TLB e o sistema procura a tradução na tabela de páginas. Alguns sistemas fornecem hardware para executar o acesso à tabela de páginas em uma falta de TLB, enquanto outros exigem que o sistema operacional acesse a tabela de páginas via software. A figura 8.9 mostra o processo de tradução de endereços em um sistema contendo um TLB.

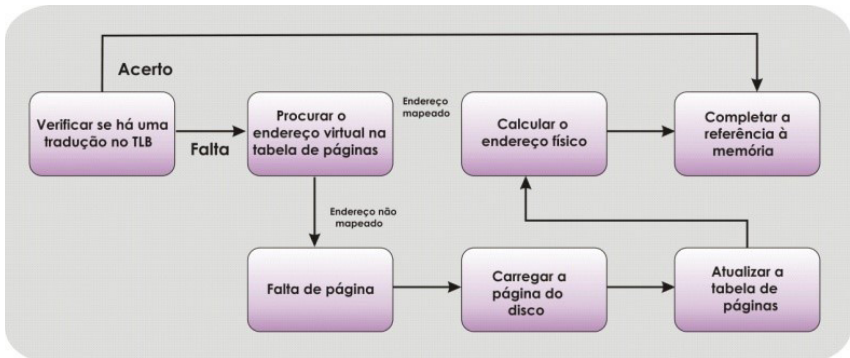


Figura 8.9: Tradução de endereços com TLB.
Figura adaptada de [Carter, N.]

<u>Organização de TLBs</u>

Os TLBs são organizados de modo semelhante às caches, possuindo uma associatividade e um número de conjuntos. Enquanto os tamanhos de cache são descritos em bytes, os tamanhos dos TLBs são normalmente descritos em termos do número de entradas ou traduções contidas no TLB, uma vez que o espaço ocupado para cada entrada é irrelevante para o desempenho do sistema.

A figura 8.10 mostra uma entrada típica de TLB. O seu for-

mato é semelhante a uma entrada na tabela de páginas e contém um NPF, um bit de válido e um bit de sujo. Além disto, a entrada do TLB contém o NPV da página, que é comparado ao NPV do endereço de uma referência à memória para determinar se ocorre um acerto. De forma semelhante a uma entrada na matriz de etiqueta de uma cache, os bits do NPV que são utilizados para selecionar uma entrada no TLB são geralmente omitidos do NPV armazenado na entrada para economizar o espaço. No entanto, todos os bits do NPF precisam ser armazenados no TLB porque eles podem ser diferentes dos bits correspondentes no NPV.

Figura 8.10: Entrada TLB.
Figura adaptada de [Carter, N.]

Os TLBs são geralmente muito menores que as caches, porque cada entrada em um TLB faz referência muito mais a dados do que uma linha de cache, permitindo que um número de entradas TLB relativamente menor descreva o conjunto de trabalho de um programa. Os TLBs contêm mais entradas do que seriam necessárias para descrever os dados contidos na cache, porque é desejável que eles contenham traduções para dados que residam na memória principal, bem como na cache.

8.6 Proteção

Além de permitir o emprego de meios magnéticos como um nível de hierarquia de memória, a memória virtual também é muito útil em sistemas de computadores multiprogramados, porque fornece proteção de memória, evitando que um programa acesse dados de outro. A figura 8.11 ilustra como isso funciona. Cada programa tem o seu próprio endereço virtual, mas o espaço de endereçamento físico é compartilhado entre todos os programas que estão sendo executados no sistema. O sistema de tradução de endereços assegura que as páginas virtuais utilizadas por cada programa sejam mapeadas sobre páginas físicas diferentes e posições diferentes no meio magnético.

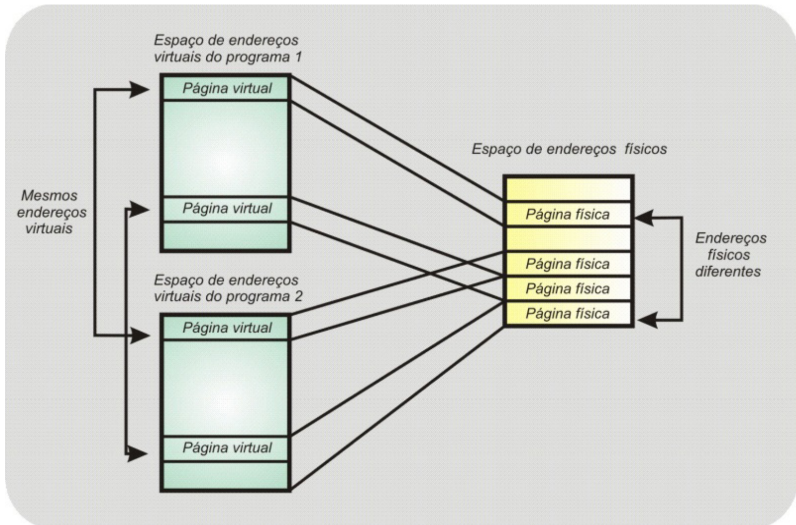


Figura 8.11: Proteção através de memória virtual.
Figura adaptada de [Carter, N.]

Isso traz dois benefícios:

Primeiro, evita que os programas acessem dados um do outro, porque qualquer endereço virtual ao qual o programa faz referência

será traduzido para o endereço físico pertencente a ele. Não existe maneira de um programa criar um endereço virtual que seja mapeado sobre um endereço físico que pertença a outro programa e, portanto, não há modo de um programa acessar os dados de outro programa. Se os programas querem compartilhar dados um com o outro, a maioria dos sistemas operacionais permite que eles solicitem especificamente que a parte das suas páginas virtuais sejam mapeadas sobre os mesmos endereços físicos.

O segundo benefício é que um programa pode criar e utilizar endereços no seu próprio espaço de endereçamento virtual sem interferência de outros programas. Assim, os programas independem de quantos programas mais estão sendo executados no sistema e/ou quanta memória aqueles outros programas estão utilizando. Cada programa tem o seu próprio espaço de endereçamento virtual e pode fazer cálculos de endereços e operações de memória naquele espaço de endereços, sem se preocupar com quaisquer outros programas que possam está sendo executados naquela máquina.

A desvantagem desta abordagem é que o mapeamento de endereços virtual-físicos torna-se parte do estado de um programa. Quando o sistema faz a comutação da execução de um programa para outro, ele precisa mudar a tabela de páginas que ele utiliza e invalidar quaisquer traduções de endereços no TLB. De outro modo, o novo programa utilizaria o mapeamento de endereços físico-virtuais do programa antigo, sendo capaz de acessar os dados. Isto aumenta o ônus de uma troca de contexto, devido ao tempo necessário para invalidar o TLB e trocar a tabela de páginas, e porque, também, o novo programa sofrerá um número maior de falta de TLB quando ele começar a ser executado com um TLB vazio.

Alguns sistemas tratam desta questão acrescentando, a cada entrada no TLB, bits adicionais para armazenar a identificação do processo ao qual a entrada se aplica. O hardware emprega o identificador do processo para o qual está sendo feita a tradução como parte da informação utilizada para determinar se houve um acerto de TLB, permitindo que traduções a partir de vários programas residam no TLB ao mesmo tempo. Isto elimina a necessidade de invalidar o TLB em cada troca de contexto, mas aumenta a quantidade de espaço de armazenamento exigido para o mesmo. Em sistemas modernos, a memória virtual é mais frequentemente utilizada como uma ferramenta para apoiar a multiprogramação do que como uma ferramenta para permitir que os programas utilizem mais memória do que aquela que é fornecida na memória principal do sistema. No entanto, é extremamente valioso ter memória virtual para fornecer proteção entre programas e permitir, em uma mudança de contexto, a substituição dos dados de cada programa entre a memória principal e o disco, o que explica porque praticamente todos os sistemas operacionais e hardware modernos suportam memória virtual.

EXERCÍCIOS

1. Explique o que é memória virtual.

2. Qual é a diferença entre endereço físico e endereço virtual?

3. Explique o processo de tradução de endereço virtual para físico.

4. Por que os sistemas de memória utilizam páginas físicas e virtuais do mesmo tamanho?

5. O que é Swapping? Para que é usado? Quais as vantagens e desvantagens?

6. Discorra sobre tabela de páginas.

7. Um sistema tem endereço virtual de 32 bits, endereço físico de 24 bits e páginas de 2KB.

   a. Qual é o tamanho de cada entrada na tabela de páginas (um único nível)?

   b. Quantas entradas na tabela de páginas são necessárias para este sistema?

   c. Quanta área de armazenamento é necessária para a tabela de página?

8. Explique o que são TLBs? Como se procede a tradução de endereço utilizando-as?

9. Como o sistema de memória virtual evita que os programas acessem dados uns dos outros?

WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI
http://www.ufpi.br/uapi
Universidade Aberta do Brasil- UAB
http://www.uab.gov.br
Secretaria de Educação a Distância do MEC – SEED
http://www.seed.mec.gov.br
Associação Brasileira de Educação a Distância – ABED
http://www.abed.org.br
Apostilas, Tutoriais e Documentos
http://gabriel.sg.urcamp.tche.br/beraldo/arquitetura_2.htm
Guia do Hardware
http://www.guiadohardware.net
Laércio Vasconcelos
http://www.laercio.com.br
Gabriel Torres
http://www.gabrieltorres.com.br
César Augusto M. Marcon
http://www.inf.pucrs.br/~marcon/
Ivan L. M. Ricarte
http://www.dca.fee.unicamp.br/~ricarte/
Marcelo Trindade Rebonatto
http://vitoria.upf.tche.br/~rebonatto
Fabian Vargas
http://www.ee.pucrs.br/~vargas
Eduardo Moresi
http://www.intelingencia.blogspot.com
Rossano Pablo Pinto
http://www.dca.fee.unicamp.br/~rossano
Alexandre Casacurta
http://www.inf.unisinos.br/~coordinf/professores/casacurta.html

REFERÊNCIAS

CARTER, N. **Arquitetura de computadores.** Porto Alegre: Bookman, 2003.

HEURING, V. P; MURDOCCA, M. J. **Introdução à arquitetura de computadores.** Rio de Janeiro: Campus, 2002.

MORIMOTO, C. E. **Hardware:** o guia definitivo. Porto Alegre: Sulina, 2007.

MONTEIRO, M. A. **Introdução aorganização de computadores.** Rio de Janeiro: LTC, 2007.

PARHAMI, B. **Arquitetura de computadores:** de microprocessadores a supercomputadores. São Paulo:McGraw- Hill do Brasil, 2008.

PATTERSON, D. A ; HENNESSY, J. L. **Arquitetura de computadores:** uma abordagem quantitativa. Rio de Janeiro: Campus, 2003.

PATTERSON, D. A ; HENNESSY, J. L. **Organização e projeto de computadores**. Rio de Janeiro: Campus, 2005.

RIBEIRO, C ; DELGADO, J. **Arquitetura de computadores**.Rio de Janeiro: LTC, 2009.

STALLINGS, W. **Arquitetura e organização de computadores**. São Paulo: Prentice Hall, 2008.

TANENBAUM, A. S. **Organização estruturada de computadores**. Rio de Janeiro:Prentice Hall, 2007.

TORRES, G. **Hardware: curso completo**. Rio de Janeiro: Axcel Books, 2001.

WEBER, R. F. **Fundamentos de arquitetura de computadores**. Porto Alegre: Bookman, 2008.

SOBRE OS AUTORES

Vinicius Ponte Machado

CV. http://lattes.cnpq.br/9385561556243194

Doutor em Engenharia Elétrica e de Computação pela Universidade Federal do Rio Grande do Norte, mestre em Informática Aplicada pela Universidade de Fortaleza (2003) e graduado em Informática pela mesma instituição (1999). Atualmente é professor adjunto da Universidade Federal do Piauí e docente pesquisador do Programa de Pós-Graduação em Ciência da Computação da UFPI. Tem experiência na área de Ciência da Computação, com ênfase em Gestão do Conhecimento e Inteligência Artificial, atuando principalmente nos seguintes temas: sistemas multiagente, aprendizagem de máquina e Redes Industriais.

André Macêdo Santana

CV. http://lattes.cnpq.br/5971556358191272

Doutorado em Engenharia da Computação pela Universidade Federal do Rio Grande do Norte (UFRN), Natal-RN, Brasil (2011), Mestrado em Engenharia Elétrica pela Universidade Federal do Rio Grande do Norte UFRN, Natal-RN, Brasil (2007) e graduação em Ciência da Computação pela Universidade Federal do Piauí (UFPI), Teresina-PI, Brasil (2004). Atualmente é Professor Adjunto do Departamento de Computação/UFPI, membro do Programa de Pós-Graduação em Ciência da Computação PPGCC/UFPI e coor-

denador do RAPOOZA (Laboratório de Robótica Aplicada, Pesquisa Operacional e Otimização). Atua principalmente nos seguintes temas: robótica móvel, visão computacional, filtragem estocástica, inteligência computacional aplicada à saúde e processamento de imagens médicas.

Ministério
da Educação