DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

UNIVERSIDADE ESTADUAL DE CAMPINAS

Organização de Computadores

(EA960)

Ivan Luiz Marques Ricarte

http://www.dca.fee.unicamp.br/courses/EA960/

1999

Ementa detalhada

Introdução à organização de computadores: Evolução de sistemas computacionais.

Classificação de arquiteturas: Estruturas de computadores. Máquinas de von Neumann. Medidas de avaliação de desempenho. Arquiteturas de alto desempenho.

Sistemas de memória hierárquica: Conceito de hierarquia de memória. *Bandwidth*, esquemas de endereçamento, memória virtual, paginação. Memória cache.

Sub-sistemas de entrada e saída: características, canais, processadores de E/S.

Princípios de pipelining: *pipelines* de instrução, *pipelines* aritméticos, projetos de sistemas *pipeline*.

Processadores vetoriais: características, exemplos.

Processadores matriciais: características, malhas de comunicação, algoritmos. Processadores associativos.

Multiprocessadores: sistemas fortemente acoplados e fracamente acoplados, exemplos; influências no sistema operacional e em linguagens de programação; algoritmos.

Arquiteturas alternativas: Arquitetura VLSI e Computadores *dataflow*.

Referências

1. **Livro-texto:**
   Advanced Computer Architecture: Parallelism, Scalability, Programmability
   Kai Hwang
   McGraw-Hill 1993, ISBN 0-07-031622-8

2. *Computer Architecture: Design and Performance, 2nd. ed.*
   Barry Wilkinson
   Prentice Hall 1996, ISBN 0-13-518200-X

3. *High-Performance Computer Architecture, 3rd. ed.*
   Harold S. Stone
   Addison Wesley 1993, ISBN 0-201-52688-3

Prefácio

Esse material foi preparado para a disciplina EA960, Organização de Computadores, que é oferecida pela FEEC/UNICAMP no segundo semestre de cada ano. O material foi elaborado para ser um **complemento** ao livro-texto, podendo ser encarado como um guia pré-leitura do livro-texto e como uma referência inicial de leitura.

A ênfase da disciplina está voltada para arquiteturas de alto desempenho. Desse modo, essa não é uma disciplina sobre *hardware* ou sobre *software* especificamente, mas sim sobre **arquiteturas**. Há várias definições para o termo "arquitetura de computadores":

- Estrutura e comportamento de computadores digitais (Hayes [6])
- Estrutura de módulos como eles estão organizados em um sistema computacional (Stone [10])
- Natureza de certas propriedades lógicas e abstratas de computadores, assim como a disciplina envolvida no projeto destas propriedades (Dasgupta)
- A interface entre o hardware de um computador e seu software (Goodman e Miller [5])
- O conjunto de tipos de dados, operações e características dos diversos níveis de projeto de um computador (Tanenbaum [11])
- A arte de fazer uma especificação [de computador] que irá viver ao longo de várias gerações de tecnologia (Baron e Higbie [1])
- O estudo de arquiteturas de conjuntos de instruções (requisitos de programação/software) e organizações de implementação de máquinas (hardware) (Hwang [7])
- O conceito sistêmico da integração de hardware, software, algoritmos e linguagens para realizar grandes computações (Hwang e Briggs [8])

Em resumo, a disciplina de arquitetura de computadores pode ser vista como o estudo, projeto e análise de sistemas computacionais digitais considerando suas características funcionais e o inter-relacionamento (a estrutura) de seus componentes.

Quando se trata de arquiteturas de alto desempenho, a ênfase está nos módulos e estruturas que permitem agilizar a velocidade de processamento, tais como processadores RISC, *pipelines*, memória *cache* e multiprocessadores. A compreensão sobre essas técnicas e módulos é fundamental para a efetiva utilização de sistemas computacionais modernos.

Sumário

Capítulo 1

Introdução à organização de computadores

O objetivo desse capítulo é apresentar os princípios que regem a organização de computadores. Para tanto, apresenta-se inicialmente alguns aspectos relativos à evolução de sistemas computacionais que tiveram impacto na organização atual de computadores.

A motivação para criar máquinas de computação sempre foi, essencialmente, melhorar a velocidade de cálculos, uma vez que a velocidade de um "computador humano" é limitada. Adicionalmente, buscava-se também reduzir fontes de erros, tais como distrações, descuidos e cansaço.

1.1 Primórdios

A pré-história das máquinas computacionais foi marcada pela criação das máquinas de calcular, tais como a *Pascalene* (Pascal, 1642–43), o *Stepped Reckoner* (Leibniz, 1674) e o *Engenho a diferenças* (Babbage, 1822–1842) [4, 9]. O marco tecnológico que precedeu esta era foi o desenvolvimento da engrenagem para cálculos com mecanismo para detecção de *carry* (W. Schickard, 1623) [2].

A organização dessas máquinas, de forma geral, assimilava-se ao diagrama apresentado na Figura 1.1.

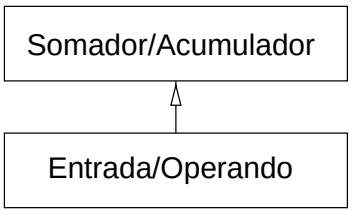


Figura 1.1: Organização das máquinas de calcular.

A *Pascalene* era capaz de realizar adição em cinco dígitos, com engrenagens em dois conjuntos de seis elementos e mecanismo de *carry* diferenciado daquele de Schickard — aparentemente Blai-

se Pascal (1623–1662) não conhecia esse outro mecanismo quando propôs sua máquina. Diversas cópias da máquina foram feitas, algumas chegando a operar com oito dígitos [2].

O *Stepped Reckoner*, de Gottfried Wilhelm von Leibniz (1646–1716), apresentava uma engrenagem cilíndrica em passos, usando a *Pascalene* como componente. Integrava ainda mecanismos para realizar multiplicação, com operandos de cinco e doze dígitos e resultados de até 16 dígitos. A operação requeria a intervenção do usuário para cada dígito do multiplicador, assim como para a operação do mecanismo de *carry*, o qual nem sempre apresentava resultados confiáveis [2].

O *Difference Engine* foi concebido inicialmente por J. H. Mueller em 1786 como uma calculadora de propósito geral aplicável a qualquer função que pudesse ser aproximada por um polinômio dado um intervalo de interesse. Posteriormente, Charles Babbage (1792–1871), em Londres, reinventou uma máquina baseada no mesmo princípio. A máquina realizava uma única operação aritmética (adição) e implementava um único algoritmo (método das diferenças finitas por polinômios), integrando ainda um dispositivo para impressão.

A motivação de Babbage em desenvolver o engenho a diferenças foi um projeto do governo francês iniciado em 1794, coordenado pelo Barão Gaspard de Prony para calcular manualmente diversas tabelas matemáticas. Este trabalho, que incluiu o cálculo de tabela de logaritmos para números naturais entre 1 e 200.000 com 19 casas decimais e o cálculo de tabelas trigonométricas, consumiu cerca de dez anos empregando entre 70 e 100 pessoas, tendo gerado 17 volumes manuscritos de grande porte.

Um protótipo de seis dígitos operando com diferenças de segunda ordem (permitindo a tabulação de polinômios quadráticos) foi apresentado em 1832, mas o projeto da máquina completa previa a operação com diferenças de sexta ordem com números de cerca de 20 dígitos.

Apesar de consumir cerca de 17000 libras do governo britânico no projeto, Babbage nunca concluiu efetivamente o engenho de diferenças, concentrando esforços em outro projeto (o *engenho analítico*). O projeto foi oficialmente cancelado pelo governo britânico em 1842. O engenho de diferenças foi efetivamente construído por outros em versões mais modestas (Scheutz, 1837–1853). Um protótipo operacional do engenho de diferenças foi construído em 1991 (bicentenário do nascimento de Babbage), no Museu da Ciência em Kensington (Inglaterra), usando tecnologia adequada à época de Babbage [2].

1.2 Primeiros Computadores

Os computadores mecânicos utilizam a mesma tecnologia das máquinas de calcular (engrenagens). Os marcos tecnológicos que precederam esta era da computação foram a introdução de sequência de cartões perfurados para controle de tear (J.-M. Jacquard, 1801) e a concepção da álgebra booleana (G. Boole, 1854).

O **Engenho Analítico** (Inglaterra, 1834–1871) de Charles Babbage foi o que se pode denominar de o primeiro computador (ou algo mais próximo de uma calculadora programável), não tendo sido efetivamente construído pela ambição excessiva do projeto e falta de interesse de possíveis clientes. Suas características básicas (Figura 1.2):

- capaz de computar qualquer operação matemática
- unidade para quatro operações básicas *(Mill)*, com dois acumuladores principais e alguns auxiliares;

- unidade para armazenar dados *(Store)*, com capacidade para algo em torno de cem números;
- cartões perfurados para operações e variáveis, com leitoras de cartão independentes;
- unidade de saída (impressa ou cartão perfurado).

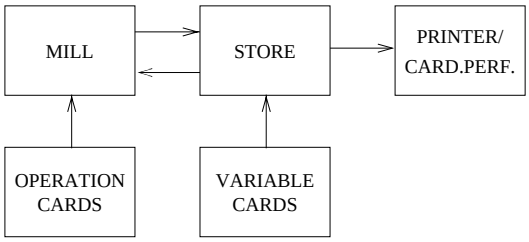


Figura 1.2: Organização do engenho analítico.

Para o projeto final, o *store* seria composto por engrenagens com dez posições, com capacidade prevista de 1000 números de 50 dígitos cada. O conceito de desvio de programas também já estava presente no projeto, através de um mecanismo que permitia reverter o movimento das leitoras de cartão. A especificação previa a realização de adições em três segundos e de multiplicações e divisões em dois a quatro minutos.

Partes de protótipos do *mill* e da impressora são produzidos por Babbage em 1871, mas após a morte de Babbage um comitê concluiu que seria impossível concluir o projeto sem seu autor. O filho de Babbage, Henry, conclui o protótipo do *mill* em 1906 e mostrou que o sistema funcionaria, mas a máquina completa nunca foi construída [2].

Outro esforço notável foi desenvolvido por Konrad Zuse na Alemanha. Os principais resultados de Zuse incluem o desenvolvimento do Z1 (1938) e do Z3 (1941). O Z1 foi o primeiro computador binário, ainda mecânico, operando com números em ponto flutuante com expoentes de sete dígitos e mantissa de 16 bits, além de um bit de sinal. O programa era lido de uma fita (filme 35mm) perfurada, com dados entrados de um teclado numérico e saída apresentada em lâmpadas elétricas. O Z2 (1939) substituiu a unidade aritmética mecânica do Z1 por uma unidade eletro-mecânica (operando com relês). O Z3 operava com números em ponto flutuante com expoente de sete dígitos, mantissa de 14 bits (com um '1' pré-fixado exceto pela representação do zero) e um bit de sinal. A memória utilizava cerca de 1400 relês para armazenar cerca de 64 números, enquanto as unidades de controle e aritmética utilizavam cerca de 1200 relês. No projeto do Z4, a memória mecânica foi retomada por ser mais compacta que a memória eletromecânica de capacidade equivalente. O Z4 foi concluído apenas em 1950, em Zurique, com um projeto que permitia desvios condicionais e uma forma básica de *pipeline* de instruções. Cerca de 300 dessas máquinas foram vendidas, sendo que posteriormente a firma montada por Zuse para sua produção foi adquirida pela Siemens [2].

Howard Hathaway Aiken (1900-1973), um estudante de pós-graduação em física em Harvard em 1936, motivou a criação de um computador para resolver seu problema (resolução de equações diferenciais não-lineares). Em Harvard, teve contato com o trabalho de Babbage. Conseguiu apoio financeiro e técnico da IBM para desenvolver os computadores Mark. O **Mark I** (1944) apresentava memória com rodas contadoras para representação decimal, com capacidade de 72 palavras de 23 dígitos. A velocidade de operação era de três adições/subtrações por segundo, uma multiplicação em

6 segundos, ou um logaritmo ou função trigonométrica em pouco mais de um minuto. As operações eram lidas de uma fitas perfurada (cerca de 7,5cm de largura) comum; o conceito de laço para operações repetitivas era suportando, bastando para isso colar os finais da fita perfurada. Dados eram lidos de fitas, cartões ou registradores com constantes. O Mark I era um equivalente eletromecânico do Engenho Analítico, tendo também se tornado conhecido como o *Automatic Sequence Controlled Calculator* (IBM ASCC). O orçamento, inicialmente previsto para cerca de US$15K, ultrapassou US$100K, envolvendo grande esforço da equipe de engenharia da IBM. Posteriormente, Aiken causou fúria a Watson (presidente da IBM) querendo assumir só o mérito no desenvolvimento de Mark I, o que motivou Watson a desenvolver máquinas que suplantassem estes esforços iniciais. O *Mark III* (1951) foi marcado pelo uso de memórias a tambor, um para instruções (capacidade para cerca de 4000 instruções), outros para dados (350 palavras de 16 bits no tambor principal, cerca de 4000 palavras nos tambores secundários) [2].

1.3 Os computadores eletrônicos

Marcos tecnológicos que precederam a era dos computadores eletrônicos foram o desenvolvimento da válvula triodo (L. de Forest, 1906) como uma extensão da válvula diodo (J. Fleming, 1904); e o desenvolvimento de circuitos binários baseados em álgebra booleana (G. Stibitz, 1937).

John Vincent Atanasoff e Clifford Berry, de *Iowa State University*, desenvolveram uma calculadora eletrônica dedicada para a resolução de sistemas de equações lineares simultâneas, denominado ABC *(Atanasoff-Berry Computer)*. A máquina operava a 60 Hz. O uso de capacitores como dispositivos de memória nessa máquina, 60 palavras de 50 bits requerendo *refreshing*, antecipou a utilização de memórias dinâmicas. A memória secundária usava cartões perfurados, movidos pelo usuário. O processo de perfurar os cartões, por calor, apresentava uma alta taxa de erros, nunca inferior a 0,001% [2]. Atualmente (desde 1973), Atanasoff é reconhecido como o criador do computador moderno.

Alan Turing, na Inglaterra, fez a especificação de computador mecânico que se tornou operacional em 1940. Um outro projeto, *Colossus*, foi desenvolvido por T. Flowers e M.H.A. Newman em 1943 com participação de Turing. *Colossus* foi o primeiro computador completamente eletrônico, tendo sido classificado pelos militares britânicos como segredo militar. Por este motivo, detalhes e características deste projeto permaneceram inacessíveis por cerca de 30 anos, sendo liberados apenas recentemente.

Maior impacto teve o Projeto **ENIAC** *(Electronic Numerator, Integrator, Analyzer, and Computer)*, desenvolvido na Moore School of Electrical Engineering (University of Pennsylvania) entre 1943 e 1946. Esse computador, desenvolvido por John W. Mauchly (1907-80) e J. Presper Eckert (1919-) e equipe, pesava cerca de 30 toneladas e apresentava um consumo de 140 KW. Utilizava aritmética decimal, com memória de 20 acumuladores de 10 dígitos, cada dígito usando 10 bits para sua representação, armazenados em flip-flops (duas válvulas por bit). A programação era manual, através de 6000 chaves e *plugs*. A velocidade de operação era cerca de 1000 vezes mais rápida que a do Mark I. Havia unidades separadas para a execução de multiplicações (cerca de 3 ms) e para divisão e raiz quadrada. 104 registros podiam ser usados para constantes de 12 dígitos. Dados podiam ser entrados através de cartões perfurados, que também eram utilizados para saída. A máquina operava a 100 KHz.

A patente para esse projeto só foi outorgada em 1964, mas em 1973 foi revogada, pois J. Mauchly

reconhecidamente teve contato com o trabalho de Atanasoff, durante uma visita a Iowa.

J.P. Eckert era o engenheiro eletrônico que realizava as concepções de J. Mauchly. No projeto ENIAC, realizado através de um convênio militar com o US Army's Ballistics Research Lab, ele teve papel fundamental em tornar operacional o conjunto de 18000 válvulas, 70000 resistores, 10000 capacitores, 6000 chaves e 1500 relés.

Em Fevereiro de 1946, o ENIAC foi revelado para o público, e no verão daquele mesmo ano foi oferecido o curso *Theory and Techniques for Design of Electronic Computers*, com aulas por Eckert, Mauchly, Stibitz, von Neumann, e Aiken, entre outros. Vários novos projetos surgiram a partir desse curso.

1.4 O conceito de programa armazenado

Os computadores eletrônicos, apesar de representar grande avanço em relação a seus similares eletromecânicos, apresentavam duas grandes limitações: baixa capacidade de memória e longo tempo de programação. A programação do ENIAC, por exemplo, exigia dias de trabalho, uma vez que várias modificações eram necessárias no painel de controle. O Mark I era fácil de "reprogramar" (troca de fita), porém velocidade de leitura de instruções de unidades mecânicas não era adequada à velocidade de processamento dos computadores eletrônicos.

O marco para quebrar essa barreira foi a concepção do conceito de programa armazenado, associada ao projeto EDVAC *(Eletronic Discrete Variable Automatic Computer)*, um sucessor do ENIAC com "ampla" capacidade de memória e que utilizava aritmética binária (Figura 1.3). A memória do EDVAC era composta por 1K palavras na memória principal (linha de atraso em mercúrio, tecnologia usada em outro projeto, o EDSAC) e 20K palavras de memória secundária (fio magnético). A tecnologia de armazenagem em linha de atraso em mercúrio, embora ampliasse a capacidade de armazenamento em relação aos flip-flops, era lenta, uma vez que convertia os sinais em som, que eram propagados pelo fluido. Entretanto, a sua velocidade de acesso era adequada aos processadores de então. A construção do EDVAC foi concluída em 1952, com cerca de 4000 válvulas, 10000 diodos a cristal, e 1024 palavras de 44 bits em memória ultrasônica; a sua velocidade de relógio era de 1 MHz [2].

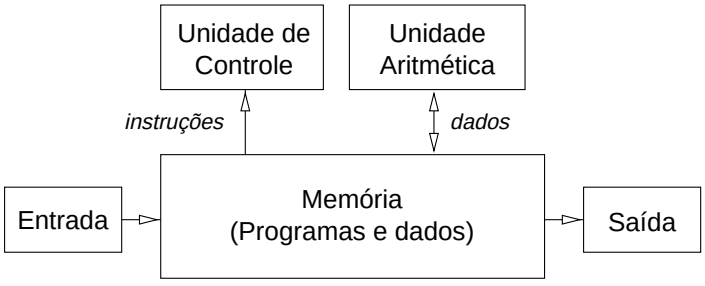


Figura 1.3: Organização do EDVAC.

Atribui-se normalmente a autoria do conceito de programa armazenado a von Neumann, exclusivamente. O motivo é que von Neumann escreveu um relatório de 101 páginas sobre o projeto EDVAC, *A First Draft of a Report on the EDVAC*, em junho de 1945, onde o conceito é formalmente

descrito pela primeira vez. No entanto, seria mais justo atribuir tal autoria à toda equipe de projeto do EDVAC, incluindo Mauchly e Eckert (Moore School), von Neumann (Institute for Advanced Study, Princeton), Herman H. Goldstine (inicialmente o elemento de *liaison* da Marinha) e Arthur W. Burks ("filósofo com inclinações matemáticas" da University of Michingan) [3].

A divulgação pública deste relatório impediu a obtenção de patentes sobre o conceito, o que causou a cizânia no grupo. Mauchly e Eckert, com uma visão de engenharia elétrica do projeto, gostariam de obter frutos transformando o EDVAC em produto comercial, enquanto von Neumann e Goldstine e Burks, com uma visão matemática e abstrata, visavam primordialmente a divulgação de resultados.

Embora o EDVAC tenha sido concluído apenas em 1952, algumas sugestões sobre o conceito de programa armazenado foram apresentadas durante a escola de verão do ENIAC. Isso permitiu que alguns participantes dessa escola, trabalhando em projetos mais modestos, viessem a construir computadores com programa armazenado antes da conclusão do EDVAC. Tais máquinas incluem:

- *Manchester Baby Machine*, da Universidade de Manchester (Inglaterra), de junho de 1948, por M. Newman e F.C. Williams;
- EDSAC *(Electronic Delay Storage Automatic Calculator)*, da Universidade de Cambridge (Inglaterra), de maio de 1949, por Maurice Wilkes;
- BINAC *(Binary Automatic Computer)*, da Eckert-Mauchly Computer Corporation (EMCC) construído sob encomenda da Northrop Aircraft Corporation, operacional em Setembro de 1949;
- UNIVAC *(Universal Automatic Computer)*, da Remington Rand Co. (que incorporou a EMCC), com a primeira unidade operacional em março de 1951;
- *Whirlwind*, do MIT por Jay Forrester, projetado como o primeiro computador para aplicações tempo-real. O Whirlwind tornou-se a base para projetos de minicomputadores;
- IBM 701, voltado para aplicações científicas (ex-*Defense Calculator*), foi o primeiro computador eletrônico da IBM (dezembro 1952);
- IBM 650 Magnetic Drum Computer, apresentado como o modelo barato da IBM (US$200K), anunciado em 1953. Essa máquina foi a base para o modelo IBM 1401 (transistorizado, anúncio em outubro de 1959, entrega no início de 1960 a um custo de US$150K).

A Manchester Baby Machine foi o primeiro computador de programa armazenado, sendo assim o que se pode chamar de o primeiro computador. Utilizava vários tubos de raios catódicos (CRT) como memória, com uma memória principal de 32 palavras de 32 bits. A programação era realizada bit-a-bit por um teclado, com a leitura de resultados também bit-a-bit (de um CRT). Tornou-se a base para um computador comercial inglês, o Ferranti Mark I (fevereiro de 1951). Posteriormente Turing juntou-se a essa equipe e desenvolveu uma forma primitiva de linguagem Assembly para essa máquina [2].

O EDSAC utilizava uma tecnologia de memória por linha de atraso em mercúrio, desenvolvida por William Shockley (Bell Labs) — com 16 tanques de mercúrio a capacidade era de 256 palavras de 35 bits, ou 512 palavras de 17 bits. Foi o primeiro computador de memória armazenado de uso

prático. Operava com uma taxa de relógio de 500 KHz. A entrada e saída de dados ocorria através de fita de papel. O primeiro programa armazenado foi imprimir quadrados dos primeiros números inteiros.

O BINAC foi projetado como um primeiro passo em direção aos computadores de bordo. Era um sistema com processadores duais (redundantes), com 700 válvulas cada e memória de 512 palavras de 31 bits. Já o UNIVAC tinha uma memória de 1000 palavras de 12 dígitos, com uma memória secundária de fitas magnéticas com capacidade de 128 caracteres por polegada. A primeira unidade do UNIVAC foi desenvolvida sob encomenda do Census Bureau norte-americano. O UNIVAC dominou o mercado de computadores na primeira metade dos anos 1950 [2].

O Whirlwind foi desenvolvido por Forrester e equipe para o US Navy's Office of Research and Inventions. A origem do projeto estava baseada em um simulador de vôo universal, com velocidade de operação adequada a aplicações de tempo real (500K adições ou 50K multiplicações por segundo). O projeto foi iniciado em setembro de 1943, tornando-se o computador operacional em 1951. Introduziu a tecnologia de memória a núcleos de ferrite (tempo de acesso de $9\mu s$), em 1953, em substituição à memória CRT original de 2048 palavras de 16 bits. O custo total do projeto superou vários milhões de dólares.

O IBM 701 estava disponível com memórias CRT de 2048 ou 4096 palavras de 36 bits. O IBM 702 Electronic Data Processing Machine estava voltado para aplicações comerciais (ex-*Tape Processing Machine*), tendo sido anunciado em setembro de 1953 e entregue no início de 1955.

1.5 Em direção aos computadores modernos

Ao mesmo tempo em que a tecnologia computacional começava a caminhar, outros desenvolvimentos tecnológicos que iriam afetar o futuro da computação também surgiam. Alguns marcos tecnológicos dignos de nota incluem a invenção do transistor, desenvolvido nos Bell Labs (J. Bardeen, W. Brattain e W. Shockley, 1947), com produção pela Texas Instruments a partir de 1954; e posteriormente o desenvolvimento de circuitos integrados (J. Bilby, Texas Instruments, e R. Noyce, Fairchild Semiconductors, 1958).

Os primeiros computadores transistorizados incluem:

- protótipos: TX-0 (MIT, 1956) e TX-2;
- DEC PDP-1 (1960), primeiro computador comercial com teclado e monitor de vídeo;
- IBM 7090 e 7094, versões transistorizadas do computador IBM709.

A série IBM7000 marcou a entrada da IBM no mercado de computadores transistorizados. Uma das características inovadoras nesses produtos era a utilização de processadores de entrada e saída *(I/O Processors)*. As características genéricas destes primeiros computadores transistorizados incluiam memórias a núcleo de ferrite e tambores magnéticos, linguagens de programação de alto nível e o conceito de *sistemas de computadores.*

O desenvolvimento de várias dessas máquinas modernas incluiam contribuições à organização do sistema que hoje estão presentes em muitos computadores. Memória virtual foi introduzida no sistema Atlas (Inglaterra, 1962); o conceito de uma família de sistemas com periféricos compatíveis foi introduzido com o IBM System360 (1964); e o PDP-8 (1965) foi o minicomputador qu introduziu o

barramento único compartilhado, além de utilizar módulos transistorizados. Em direção ao processamento de alto desempenho, o CDC6600 (1964) continha diversas unidades funcionais e IOPs; o CDC STAR-100 (String Array Computer) fazia uso de *pipelines* e o ILLIAC-IV era um multiprocessador matricial com 64 unidades de processamento.

Os anos seguintes de desenvolvimento de sistemas computacionais foram marcados principalmente pela utilização de componentes com grau cada vez maior de integração, o que determinou uma das principais questões em relação a arquiteturas de computadores: dentre as funções alocadas a um sistema computacional, o que é alocado ao *hardware* e o que é alocado ao *software*?

Assim, a década de 1970 foi marcada pela utilização de circuitos integrados em larga escala, como exemplificado pelo Fairchild 3800 (ALU de 8 bits, 1967); pelos microprocessadores Intel 4004 (T. Hoff, S. Mazor e F. Fagin, 1971), 8008 (1972) e 8080 (1973); pelo minicomputador DEC PDP 11/45 (1972), com extenso uso de circuitos em *chips*; pelos dispositivos de memória dinâmica de 1 Kbit (Intel 1103, 1971) e 4 Kbit (1974); pelo DEC VAX 11/780 (1978), um minicomputador de 32 bits; e pelos microprocessadores Intel 8086 (1978), de 16 bits, e Motorola 68000 (1979), de 16/32 bits. Algumas funções específicas, como unidades de gerência de memória (MMU) e co-processadores aritméticos, passaram também a ser implementadas em hardware.

Na computação de alto desempenho, algumas contribuições dessa década incluem o computador 801 (J. Cocke; IBM, 1975), com arquitetura RISC, e o Cray-1 (S. Cray, 1976), com arquitetura vetorial.

Na década de 1980 houve principalmente a continuidade nesse processo iniciado na década anterior, com uma exploração ainda maior da capacidade de integração de circuitos. Surgiram dispositivos de memória dinâmica com maior capacidade: 64 Kbits (1981), 256 Kbit (NEC, 1984), 1 Mbit (IBM, 1984), 4 Mbit e 16 Mbit (1987); e microprocessadores mais poderosos: MC68020 (Motorola, 1984), 80286 (Intel, 1984), 80386 (Intel, 1985), MC68030 (Motorola, 1987), 80486 (Intel, 1989). Os sistemas de alto desempenho passaram a explorar de forma mais intensa o paralelismo, como os Cray X-MP (1982) e Cray-2 (1985); o computador Connection Machine CM-2 (1985), da empresa Thinking Machines; os microprocessadores Transputers T-414 (1985) e T-800, da empresa Inmos. Houve também um investimento em arquiteturas RISC, como marcado pelo processador Motorola 88000 (1988).

Os anos 1990 foram marcados pela disponibilização de dispositivos com altíssimo grau de integração, com microprocessadores com grande poder de processamento, tais como iPSC/860 (Intel, 1990), 68040 (Motorola, 1990), Alpha RISC/64 bits (DEC, 1992), Pentium (Intel, 1993). Computadores de alto desempenho surgiram, tais como o Cray Y-MP C90 (1991, 16 processadores, 16 GFlops) e o Thinking Machines CM-5; entretanto, o alto custo de tais sistemas podem ter decretado seu fim. Atualmente, multiprocessamento é utilizado em escalas modestas em estações de trabalho e servidores. O desenvolvimento de processadores ópticos ficou como uma promessa não cumprida nessa década, após o desenvolvimento de um protótipo no Bell Labs (1990).

Capítulo 2

Classificação de arquiteturas

No projeto de sistemas computacionais no nível de processador (trabalhando com grupos de palavras), os tipos de componentes considerados são processadores, memórias, dispositivos de entrada e saída, e meios de interconexão.

Processadores. Contemplam componentes tais como CPU, controladores e coprocessadores. Têm um conjunto de instruções (de propósito geral para a CPU, especializado para coprocessadores) operando sobre instruções e dados (obtidos e armazenados externamente) organizados em palavras.

Memórias. Incluem dois subsistemas principais, memória principal e memória secundária. O custo associado à memória está diretamente relacionado à sua capacidade de armazenamento e à sua velocidade de operação.

Dispositivos de entrada e saída. São conversores de representação física de dados. Em geral, são lentos quando comparados com o processador.

Meios de interconexão. Estabelecem a comunicação entre componentes através de barramentos sob seu controle. Um problema crítico na utilização dos meios de interconexão é a contenção, a disputa pelo uso simultâneo de recursos compartilhados.

2.1 Classificação básica

Com base no relacionamento e organização dos componentes de um sistema computacional, esses sistemas podem ser classificados entre máquinas von Neumann e máquinas não-von Neumann.

2.1.1 Máquinas von Neumann

A característica de máquinas von Neumann é a composição do sistema a partir de três subsistemas básicos: CPU, memória principal e sistema de entrada e saída [1].

A CPU (unidade central de processamento), por sua vez, tem três blocos principais: unidade de controle (UC), unidade lógico-aritmética (ALU) e registradores, incluindo-se aí um registrador contador de programa (PC) que indica a posição da instrução a executar.

São características das máquinas von Neumann a utilização do conceito de programa armazenado, a execução seqüencial de instruções e a existência de um caminho único entre memória e unidade de controle (Figura 2.1).

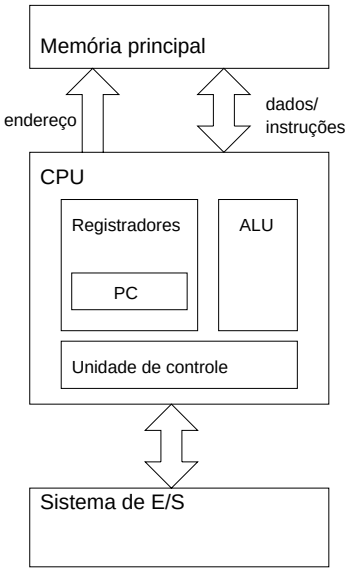


Figura 2.1: Arquitetura de máquinas von Neumann.

Durante sua operação, a execução de um programa é uma seqüência de *ciclos de máquina von Neumann*, compostos por:

1. Busca da instrução *(fetch)*: transfere instrução da posição de memória apontada por PC para a CPU

2. Execução da instrução: a unidade de controle decodifica a instrução e gerencia os passos para sua execução pela ALU

Uma variante do modelo básico de máquinas von Neumann é denominado de máquinas Harvard, onde há vias separadas para dados e instruções entre memória principal e CPU (Figura 2.2). A origem do termo vem dos computadores Mark I a Mark IV, desenvolvidos em Harvard, com memórias de dados e instruções separadas.

2.1.2 Máquinas não-von Neumann

As máquinas que não se enquadram na definição de máquinas von Neumann são denominadas máquinas não-von Neumann. Essa categoria é ampla, incluindo sistemas computacionais tais como:

Máquinas paralelas: várias unidades de processamento executando programas de forma cooperativa, com controle centralizado ou não;

Máquinas de fluxo de dados: não executam instruções de um programa, mas realizam operações de acordo com a disponibilidade dos dados envolvidos;

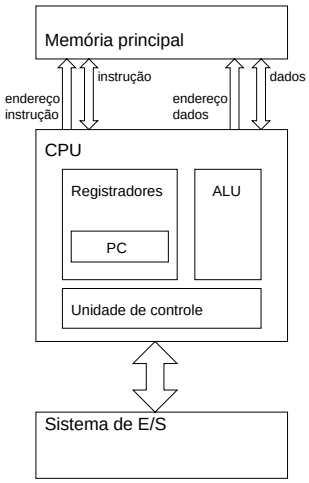


Figura 2.2: Máquina Harvard

Redes neurais artificiais: também não executam instruções de um programa, trabalhando com um modelo onde resultados são gerados a partir de respostas a estímulos de entrada; e

Processadores sistólicos (VLSI): processamento ocorre pela passagem de dados por arranjo de células de processamento executando operações básicas, organizadas de forma a gerar o resultado desejado.

2.2 Avaliação de arquiteturas

Como há um grande número de alternativas para a organização de um computador, é preciso ter mecanismos que permitam realizar a avaliação de cada arquitetura. Algumas medidas básicas de avaliação são necessárias para tanto.

O **desempenho** está usualmente associado à velocidade de execução de instruções básicas (taxas MIPS e FLOPS) ou à velocidade de execução de programas representativos das aplicações *(benchmarks)*.

O **custo** é geralmente analisado não em valor nominal, mas em termos de parâmetros que influenciam no custo final: número de pinos, área de chip, número de chips por sistema.

O **tamanho** de programas e dados em geral expressa a capacidade máxima e a eficiência de ocupação do código gerado.

Restrições físicas, tais como peso, volume e consumo de potência, também podem ser consideradas de acordo com a aplicação em vista. A facilidade de programação também pod ser um item importante, tomando em consideração que linguagens de programação são suportadas e que mecanismos estão presentes para explorar as características de alto desempenho do sistema.

Dificilmente uma medida é considerada isoladamente na avaliação de sistemas. A relação custo-desempenho é uma medida de comparação básica. Outra medida usualmente considerada é a adequação de uma arquitetura (de propósito geral ou dedicada) à aplicação-alvo.

Entretanto, se há uma medida que recebe maior peso nas avaliações de sistemas computacionais, esta medida certamente é o *desempenho* do sistema.

2.2.1 Avaliação de Desempenho

A medida de avaliação mais citada para comparar sistemas computacionais é o desempenho e, para o curso de EA960, essa será de fato a medida mais importante.

No processo de avaliação de desempenho, o objetivo é obter um modelo para estimar uma medida de desempenho a partir de parâmetros de projeto e parâmetros de entrada. Exemplos de parâmetros relevantes incluem tempo de processamento, tempo de espera, e a utilização de recursos.

Existem várias abordagens de avaliação, as quais podem ser classificadas em duas grandes categorias, o desenvolvimento de modelos analíticos e o desenvolvimento de modelos numéricos. Dentro desta última categoria, as bases para o desenvolvimento do modelo podem ser estabelecidas através de simulação (computacional ou física) ou através de medidas.

Algumas medidas de desempenho típicas envolvem grandezas tais como:

Banda de passagem de memória principal: expressa a máxima taxa de transferência de dados entre memória e CPU;

Tempo médio de execução de instrução: expressa a média ponderada do tempo de execução de instrução pela probabilidade de ocorrência da instrução. Geralmente, este valor é apresentado como a taxa de execução de instruções (recíproco do tempo médio), expresso em MIPS (milhões de instruções por segundo).

Benchmark: expressa o tempo total de execução de programas representativos para a aplicação de interesse. É uma medida mais global que as anteriores, pois incorpora o uso de vários componentes, podendo considerar até mesmo operações de entrada e saída de dados.

A medida do tempo de execução, em si, já não é tarefa simples. Um dos aspectos que devem ser considerados inclui o próprio modo de operação normal da aplicação: a ênfase é na execução de uma única tarefa ou na otimização da execução de um grupo de tarefas (ou seja, na vazão do sistema)? Outro aspecto que deve estar claro é qual o tempo que está sendo medido, o tempo de resposta ou o tempo de CPU. Nessa mesma direção, deve-se esclarecer que desempenho busca-se melhorar, o desempenho do sistema ou o desempenho de CPU.

A estratégia de avaliar um sistema por programas também oferece diversas alternativas. Podem ser utilizados programas reais completos ou segmentos de programas reais *(kernels)*. Caso a opção seja pela utilização de *benchmarks*, podem ser utilizados *"toy benchmarks"* ou *benchmarks* sintéticos, derivados a partir de aplicações significativas da categoria sob consideração.

É preciso considerar ainda que o tempo de execução de programas individuais é pouco representativo, uma vez que um mesmo programa em uma mesma máquina pode ter tempos diferenciados de execução devido a uma série de fatores externos. É importante, pois, obter uma medida que expresse uma combinação das medidas em diversas execuções, tais como:

- somatório dos tempos de execução;
- média ponderada dos tempos de execução;

- tempo relativo a uma máquina-referência, algumas vezes expressando aqui os valores em termos de média geométrica.

2.2.2 Como melhorar o desempenho?

Antes de mais nada, é preciso determinar que categoria de aplicação estará sendo atendida pelo sistema computacional sob estudo. É muito difícil fazer comparações utilizando valores que, por si, não expressam muita informação.

Considere por exemplo o uso da taxa MIPS como um parâmetro de comparação. O valor dessa taxa é computado pela expressão:

$$MIPS = \frac{\text{instr}}{\text{ciclo}} \times \frac{\text{ciclo}}{s} \times 10^{-6}$$

Para aumentar o valor dessa taxa, as duas estratégias possíveis são:

1. reduzir número de instruções para executar tarefa; ou
2. aumentar o número de instruções executadas por ciclo.

O problema é que esse valor pode não comparar adequadamente o desempenho efetivo de duas máquinas sob estudo, pois o conceito de "instrução" pode variar significativamente entre sistemas.

Outro aspecto que deve ser considerado também na otimização de uma aplicação em um sistema computacional é o **Princípio de Amdahl**. Segundo esse princípio, o impacto de uma otimização realizada sobre o desempenho global do sistema depende da fração da aplicação que está sendo beneficiada pela otimização. Assim, melhorar em 100 vezes o desempenho de 10% do sistema representa uma melhoria global de apenas 11%. Quanto menor for a fração atingida pela otimização, menor será o seu imapcto.

2.3 Arquiteturas de alto desempenho

O grande objetivo das técnicas de melhoria de desempenho é reduzir ineficiência. Algumas formas de atingir essa redução incluem: não suportar funções que não serão utilizadas; reduzir ociosidade dos componentes; e definir uma organização adequada para o sistema. Por exemplo, é preciso saber quando é melhor utilizar um processador de alto desempenho ou vários processadores para resolver um dado problema.

2.3.1 Aplicações

Técnicas adotadas em sistemas computacionais de alto desempenho dependem fortemente da aplicação. Aplicações que usualmente requerem o uso de técnicas de alto desempenho envolvem:

- computação numérica
  - estruturada
  - não-estruturada
- aplicações de tempo-real

- E/S intensiva
- atividades de projetos
- capacidade de inteligência artificial

O programa de computação e comunicação de alto desempenho do governo norte-americano (HPCC) identificou algumas aplicações caracterizadas como *Grand Challenge Applications* [7], tais como:

Tecnologia de gravação magnética: estudar magnetostática e interações de troca para reduzir ruído em discos de alta densidade

Projeto racional de remédios: desenvolver drogas para curar câncer ou AIDS através do bloqueio da ação do HIV

Transporte civil de alta velocidade: desenvolver jatos supersônicos através de modelos computacionais de dinâmica dos fluidos executando em supercomputadores

Combustíveis: projetar melhores modelos de motores através de cálculos de cinética química para revelar os efeitos mecânicos dos fluidos

Modelagem oceânica: simulação em ampla escala das atividades oceânicas e troca de calor com fluxos atmosféricos

Anatomia digital: sistemas computacionais operando em tempo-real no apoio à obtenção e análise de imagens clínicas, tomografias e ressonância magnética

Tecnologia para educação: educação de ciências ou engenharia apoiada por simulações computacionais em sistemas interconectadas por redes heterogêneas.

Estas são algumas das aplicações que necessitariam de sistemas computacionais com desempenho 3T [7]:

- 1 Teraflop de poder computacional
- 1 Terabyte de memória principal
- 1 Terabyte por segundo de capacidade de transferência de entrada e saída

2.3.2 Desenvolvimento das arquiteturas

O projeto de um sistema com arquitetura de alto desempenho oferece diversas alternativas. Em alguns casos, é possível atingir os resultados esperados através da manipulação do algoritmo para melhor ajustá-lo à organização do sistema. Em outros casos, mudanças na organização básica do sistema podem ser necessárias. Há ainda casos onde organizações alternativas, tais como o uso de paralelismo, podem ser mais adequadas.

Durante o processo de definição de arquitetura, algumas questões de interesse são:

- Qual o tempo necessário para executar um conjunto de programas representativo da aplicação?

- Qual o espaço de armazenamento necessário para um dado conjunto de programas ou dados?
- Qual o fator de utilização dos diversos componentes do sistema?

Não há respostas simples para tais questões. Elas podem requerer a análise de propriedades de um programa médio e também a probabilidade que o programa tenha uma dada propriedade — o modelo probabilístico é em geral adequado pois captura incertezas que podem se refletir na execução real dos programas.

É praticamente impossível desenvolver sistemas computacionais com arquitetura de alto desempenho adequadas ao uso em qualquer aplicação (propósito geral). O projetista (arquiteto) de um sistema computacional de alto desempenho deve ser capaz de identificar e remover os "gargalos" do sistema, ou seja, as condições de execução que limitam o desempenho global da aplicação-alvo. Em geral, a solução para este problema deriva-se de uma combinação das atividades relacionadas ao projeto.

Assim, no desenvolvimento de um sistema computacional de alto desempenho, os aspectos relevantes que serão considerados nesse curso são:

Projeto de processadores: os fatores que mais afetam o desempenho da unidade central de processamento. Como acelerar o processamento (decodificação e execução) de instruções. As contribuições da abordagem RISC.

Sistemas de memória hierárquica: estratégias para melhorar velocidade de acesso nos diversos níveis da hierarquia de memória, com ênfase para *cache*.

Sistemas paralelos: vantagens e dificuldades relacionadas.

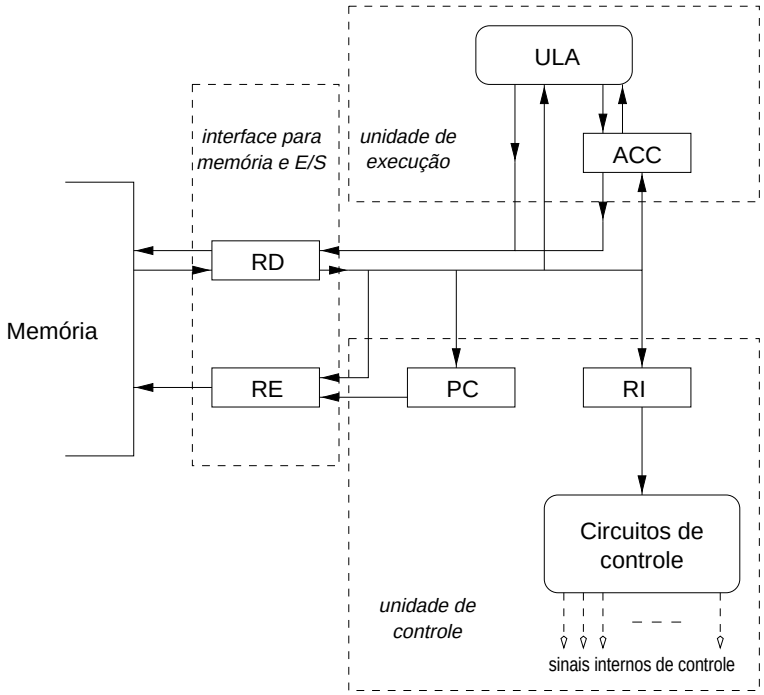
Capítulo 3

Projeto de Processadores

O objetivo desse capítulo é apresentar e discutir alternativas de projeto que permitem acelerar a operação das unidades centrais de processamento.

3.1 Revisão: Operação de Processadores

Arquitetura Interna. Ilustrada a seguir através de uma CPU muito simples mas que inclui os módulos presentes em qualquer processador: *unidade de execução*, *unidade de controle* e *registradores*.



RD Registrador de dados

RE Registrador de endereços da memória

ACC Registrador acumulador

PC Registrador contador de programa

RI Registrador de instruções

ULA Unidade lógica e aritmética

```
COMPORTAMENTOCPU()
 1  while CPUativada
 2      do RE ← PC
 3         RD ← MEMÓRIA[RE]
 4         RI ← RD.CodOp
 5         PC ← PC + incremento
 6         DECODIFICA INSTRUÇÃO
 7         if ADD
 8           then RE ← RD.Ender
 9                RD ← MEMÓRIA[RE]
10                ACC ← ACC + RD
11           else if JUMP
12                  then PC ← RD.Ender
13                  else if etc
14                         then ...
```

Ciclo de busca de instrução (FETCH): linhas de 1 a 6

Ciclo de execução de instrução (EXECUTE): linhas restantes

3.2 Fatores que afetam projeto de processadores

- Formato de dados
- Formato de instruções
- Estratégia de controle

3.2.1 Formatos de dados

Podem ser numéricos ou não-numéricos:

- Numéricos
  - inteiros, ponto fixo, ponto flutuante, complexos?
  - representação binária ou decimal?
  - faixa de valores representáveis?
  - precisão?

- Não-numéricos
  - Caracteres: EBCDIC, ASCII, UniCode, ...?
  - Strings?
  - Multimídia: imagens, vídeo, áudio, compressão, ...?

Qual o custo de incorporar *hardware* para processar e armazenar esses tipos de dados?

3.2.2 Formatos de instruções

A instrução tem usualmente um campo de *Código de Operação* e zero ou mais campos de endereços de *Operandos*:

- Código de Operação:
  - tamanho fixo ou variável?
- Operandos:
  - número máximo de operandos explícitos na instrução?
  - inclusão de operandos implícitos?
  - modos de endereçamento?

Código de operação

O código de operação de tamanho fixo simplifica o projeto da unidade de controle da CPU. A motivação para ter códigos de tamanho variável é minimizar a ocupação de memória para armazenamento de programas.

Huffman. Método para codificação de tamanho variável. Duas etapas:

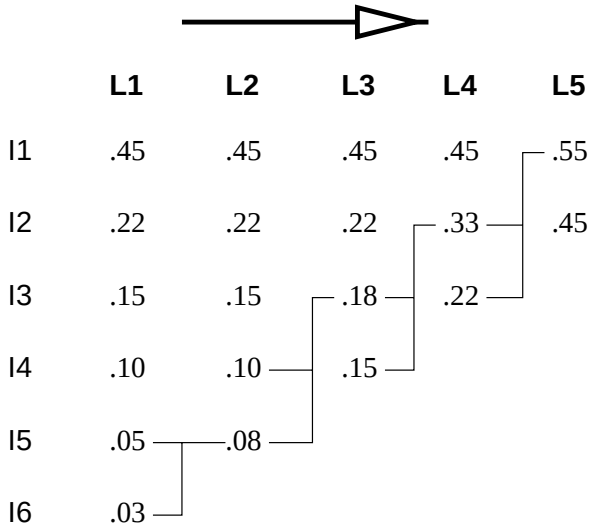
1. Construção de listas de instruções ordenadas pela freqüência de ocorrência em programas. Lista L_1 contém todas as n instruções. Lista L_i é construída a partir da combinação dos dois elementos de menor valor em L_{i-1}. Processo conclui com lista L_{n-1}, com apenas dois elementos.

2. Atribuição de códigos aos elementos da lista. A partir de L_{n-1}, atribui bit 1 para primeiro elemento e bit 0 para o segundo elemento (ou *vice versa*). Para codificar elementos na lista L_{i-1} a partir dos códigos dos elementos da lista L_i: se elemento em L_i foi resultado da combinação de dois elementos de L_{i-1}, então agrega bit 1 ao código do primeiro elemento combinado e bit 0 ao código do segundo elemento. Os demais códigos são preservados. Processo conclui com a atribuição de códigos aos elementos de L_1 (todas as instruções).

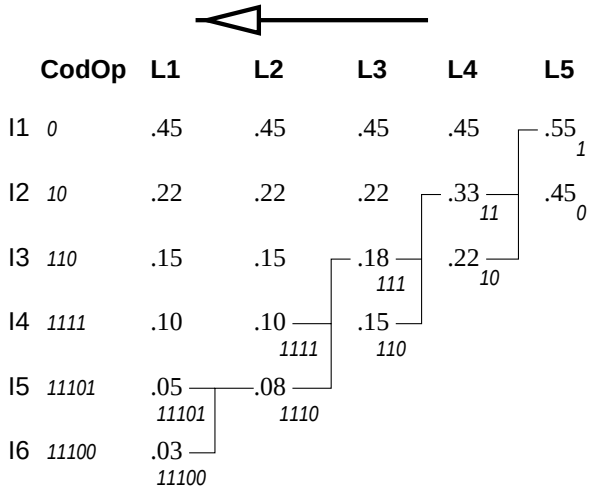
Exemplo: Processador hipotético com seis instruções

Instrução	**Freqüência**
I_1	0.45
I_2	0.22
I_3	0.15
I_4	0.10
I_5	0.05
I_6	0.03

Etapa 1:



Etapa 2:



Se código de operação de tamanho fixo fosse utilizado, 3 bits seriam necessários para codificar as seis instruções.

Com a codificação acima, o valor médio é

$$\sum_{i=1}^{6} f_i \times |I_i| = 2,1$$

Operandos

Um processador cujas instruções podem ter no máximo m operandos explícitos é classificada como uma *máquina de m-operandos*. Quanto menor o número de operandos explícitos, mais primitivas são as instruções (e portanto de execução mais rápida), porém maior quantidade de instruções é necessária para realizar uma mesma operação.

Exemplo. Para a instrução de alto nível

```
X = X + Y * Z
```

Em uma máquina de 3-operandos, onde `OP A,B,C` representa "`A` recebe o resultado de `B OP C`":

```
MULT  W,Y,Z
ADD   W,W,X
```

Em uma máquina de 2-operandos, onde `OP A,B` representa "`A` recebe o resultado de `A OP B`":

```
MOVE  Z,W
MULT  W,Y
ADD   W,X
```

Em uma máquina de 1-operando, onde `OP A` representa "`Acc` recebe o resultado de `Acc OP A`," onde `Acc` é o registrador acumulador:

```
LOAD  Y
MULT  Z
ADD   X
STORE W
```

Em uma máquina de 0-operandos *(stack machine)*, onde a operação binária é realizada consumindo os dois elementos no topo da pilha de registradores e o resultado é armazenado no topo da pilha:

```
PUSH  Z
PUSH  Y
MULT
PUSH  X
ADD
POP   W
```

3.2.3 Unidade de Controle

Controle hardwired: sinais internos de controle gerados através de circuitos seqüenciais controlados por sinais de *status* e pelo conteúdo do registrador de instruções.

Controle microprogramado: sinais internos de controle gerados pela execução de micro-instruções armazenadas em uma memória de controle.

Controle *hardwired* permite obter maior velocidade de execução de instruções, enquanto que o controle microprogramado torna o projeto da unidade de controle mais simples e flexível, facilitando a implementação de instruções complexas.

3.3 Evolução de processadores

Com desenvolvimento da tecnologia de *hardware*, principalmente na tecnologia de integração, projetistas de processadores introduziam no *hardware* novas operações e novos modos de endereçamento para:

- substituir seqüências de operações aritméticas primitivas
- suportar operações repetitivas
- fornecer métodos alternativos de endereçamento
- auxiliar invocação de procedimentos e passagem de parâmetros
- auxiliar execução de funções do sistema operacional
- facilitar construção de multiprocessadores

A presença destas instruções complexas caracteriza as chamadas máquinas CISC *(Complex Instruction Set Computer)*.

Exemplos:

- VAX-11/780: 303 instruções, 16 modos de endereçamento
- MC68020: 109 instruções, 18 modos de endereçamento
  (68040: 113 e 18, respectivamente)
- Intel i386: 111 instruções, 8 modos de endereçamento
  (i486: 157 e 12, respectivamente)

Para representar esta ampla gama de instruções, o processador deve utilizar instruções de tamanho variável:

- Intel i432: 6 a 321 bits por instrução
- Intel 8086: 1 a 6 bytes por instrução
- Motorola 68000: 1 a 5 words (16 bits) por instrução
- VAX-11/780: 2 a 57 bytes por instrução

Na segunda metade da década de 1970, esta estratégia de evolução de processadores começou a ser questionada:

- Qual o efeito da incorporação de instruções adicionais no projeto do processador?
- Há efetivamente melhoria no desempenho?

Foram realizados diversos estudos analisando o uso das instruções dos processadores, mostrando resultados como:

- atribuição corresponde, em média, a 47% das instruções executadas, sendo que destas 80% é de atribuição simples;
- mais de 50% das instruções opera com endereçamento a registradores, cerca de 30% correspondem a operações de transferência (MOVE), 12% a transferência para a pilha (PUSH, POP), e cerca de 10% a desvios condicionais;
- no VAX-11/780, 20% das instruções eram responsáveis por cerca de 60% do microcódigo; essas instruções eram utilizadas 0,2% do tempo.

Na linha tradicional de evolução de processadores, resultados como estes motivaram a codificação Huffman de instruções e a inclusão de instruções em "versão *quick*".

A utilização da codificação de Huffman permitia economia de espaço ocupado por código tipicamente em torno de 30 a 35%, com estudos demonstrando em alguns casos economias de até 43%.

A incorporação de instruções *quick* tinham impacto no microcódigo, que ficava mais complexo. Por exemplo, a memória de controle do VAX-11/780 era de 456 Kbytes.

Outro impacto devido ao suporte a instruções de tamanho variável: o processo de decodificação de instruções é inerentemente seqüencial.

3.4 A abordagem RISC

Filosofia. Transferir complexidade das operações para *software*, mantendo em *hardware* apenas as operações primitivas.

RISC ≡ *Reduced Instruction Set Computer*

3.4.1 Conceitos motivadores

Instruções complexas: apenas incluir quando o benefício no desempenho compensar a degradação de velocidade;

Uso de transistores: área de VLSI pode ser utilizada para novas instruções ou para aumentar número de registradores, incluir memória cache no chip do processador, adicionar unidades de execução;

Uso de microcódigo: deve ser evitado, pois o *overhead* associado ao tempo de acesso a microinstruções na memória de controle passou a ser considerável a partir do momento em que a tecnologia da memória principal passou de núcleos de ferrite para dispositivos semicondutores;

Papel do compilador: deve substituir eficientemente as operações complexas eliminadas do *hardware*. Para atingir este objetivo, otimização é fundamental; projeto de compiladores realizado juntamente com o projeto dos processadores.

Características comuns à maior parte dos processadores RISC:

- número de instruções limitado;
- codificação de instruções em uma palavra de tamanho fixo;
- execução sem micro-código;
- altas taxas de execução (próximas a 1 instrução/ciclo)
  - uso intenso de *pipelines*;
- poucos modos de endereçamento;
- operações envolvendo a memória principal restritas a transferências (LOAD, STORE);
- operações lógicas e aritméticas entre registradores, tipicamente com instruções de três endereços.

3.4.2 Exemplos

IBM801

- Período 1975–79, divulgação em 1982;
- Instruções:

  - 120 instruções de 32 bits;
  - instruções lógicas e aritméticas de três endereços de registradores;
  - memória: instruções LOAD e STORE apenas
- 32 registradores de propósito geral;
- dois modos de endereçamento (base+índice, base+imediato);
- pipeline de quatro estágios:
  1. instruction fetch
  2. register read ou address calculation
  3. ALU operation
  4. register write
- tecnologia ECL (*Emitter-Coupled Logic*), SSI/MSI (*Small/Medium Scale of Integration*), com ciclo de 66ns;
- compilador otimizado, com alocação de registradores com base em tempo de vida (*lifetime*) de variáveis e coloração de grafos:
  - variável $\equiv$ nó no grafo;
  - variáveis com sobreposição no tempo de vida $\equiv$ nós ligados por aresta;
  - número de registradores disponíveis $\equiv$ quantidade de cores para colorir o grafo.

Limitação da abordagem de alocação: tempo de vida não necessariamente corresponde à freqüência de uso.

Protótipos acadêmicos

Berkeley: RISC-I (1982), RISC-II (1983)

Stanford: MIPS (1983)

Implementações VLSI de processadores RISC de 32 bits, instruções registrador a registrador, sem micro-código.

RISC-I — 31 instruções, 78 registradores (em janelas de 32 registradores), dois modos de endereçamento, dois formatos de instrução, pipeline de dois estágios.

RISC-II — 39 instruções, 138 registradores (em janelas de 32 registradores), dois modos de endereçamento, dois formatos de instrução, pipeline de três estágios.

MIPS *(Microprocessor without Interlocked Pipeline Stages)* — 55 instruções, 16 registradores, dois modos de endereçamento, quatro formatos de instrução, pipeline de cinco estágios.

Processadores comerciais

Sun SPARC — derivado de RISC-II, com 39 instruções.

MIPS Computer System Corporation — R2000, R3000, R4000,...

MC88100 (Motorola, 1988) — instruções registrador a registrador com três operandos, memória apenas através de LOAD a STORE, 51 instruções de comprimento fixo (32 bits), sem micro-código, quatro unidades de execução, arquitetura Harvard.

Tendências correntes na evolução de processadores RISC apontam para a operação super-escalar, ou seja, obter a execução de mais de uma instrução por ciclo. Nesta linha incluem-se IBM-RS6000, DEC 21064 (Alpha), PowerPC, com taxas de duas a quatro instruções por ciclo através de duas a seis unidades funcionais.

Outras tendências são a utilização de palavras de 64 bits e endereços de 40 bits ou mais. Processadores representantes desta linha incluem Alpha 21164, MIPS 10000, PowerPC 620 e UltraSPARC, com quatro unidades funcionais e 40 ou 41 bits de endereçamento.

3.4.3 Instruções em RISC

Instruções de um processador são usualmente classificadas em uma das seguintes categorias:

1. transferência de dados;
2. aritméticas;
3. lógicas;
4. controle de fluxo de programas;
5. entrada e saída.

Em alguns casos, há instruções que não são classificadas como de um único tipo.

Os primeiros computadores apresentavam conjuntos de instruções pequenos e simples, pela necessidade de minimizar o uso de *hardware*. Com a evolução da capacidade de integração e, conseqüentemente, com o barateamento de *hardware*, a quantidade e a complexidade das instruções em processadores cresceram.

O compromisso que existe na implementação de uma operação F como uma única instrução I_F *ou* como uma rotina P_F composta de instruções mais simples.

A favor de I_F. Em geral, a execução de I_F é mais rápida que a execução de P_F, a qual necessitaria de um maior número de buscas de instruções da memória *(fetch)* e de manipulação de resultados intermediários, além de ocupar um maior espaço em memória.

A favor de P_F. Por outro lado, o processamento de I_F requer uma complexidade muito maior da parte do processador, o que traz impactos para o tamanho, tempo e custos de projeto — além do risco de ocorrência de problemas. P_F favorece o uso de otimizações do compilador, algo que I_F não permite — por exemplo, em uma multiplicação onde um dos operandos é zero. Finalmente, não se deve comparar diretamente o tempo de *fetch* e execução de uma operação complexa com o tempo necessário para buscar e executar uma instrução simples.

Na abordagem RISC, a segunda opção é favorecida. Assim, o código de um programa gerado para uma máquina RISC tem em geral mais instruções que aquele gerado para uma máquina CISC, mas tem o potencial para executar de forma mais eficiente — principalmente em instruções simples, com operandos inteiros. O maior tráfego de instruções entre CPU e memória em máquinas RISC é uma motivação para o uso de *caches* de instruções.

3.5 Janelas de Registradores

3.5.1 Motivação

- Chaveamento de contexto (salvar/restaurar valores de registadores) é responsável por tempo razoável de processamento entre chamadas de procedimentos.
- Maior número de registradores disponível em processadores RISC só agravaria o impacto desta sobrecarga *(overhead)* no processamento.

Solução (Projeto RISC/Berkeley, Sun SPARC):

$\Rightarrow$ Cada procedimento tem acesso a apenas parte do conjunto total de registradores (uma **janela** de registradores);

- Há uma sobreposição entre janelas consecutivas que facilita passagem de argumentos e de valores de retorno de funções.

3.5.2 Exemplo

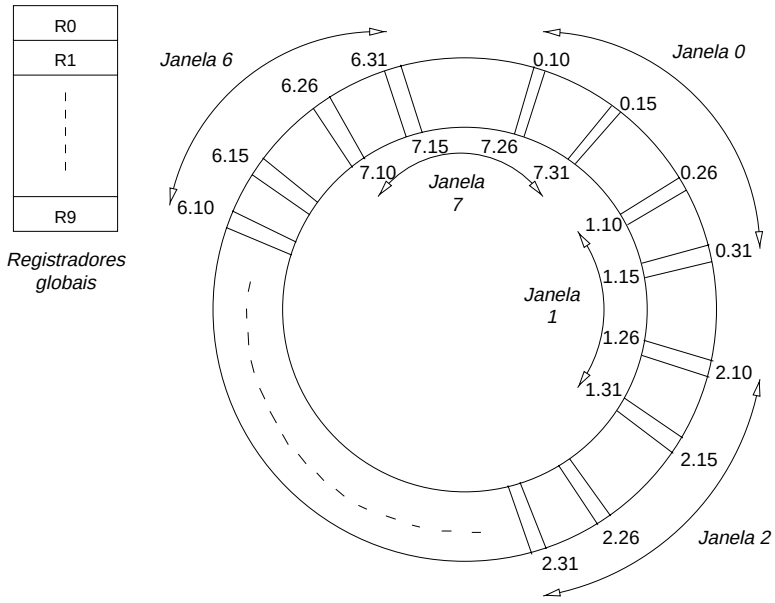
Organização de registradores em RISC-II:

- 138 registradores organizados em
  - 10 registradores globais (R0...R9)
  - 8 janelas de 22 registradores locais cada (R10...R31)
- Endereço de registrador tem dois campos:
  1. janela: 3 bits
  2. registrador: 5 bits
- Registradores sobrepostos (janelas consecutivas) têm dois endereços. Por exemplo, para as janelas 0 e 1:

1. 0.R26 ≡ 1.R10
2. 0.R27 ≡ 1.R11
3. 0.R28 ≡ 1.R12
4. 0.R29 ≡ 1.R13
5. 0.R30 ≡ 1.R14
6. 0.R31 ≡ 1.R15

. . . e da mesma forma para as janelas 1 e 2, 2 e 3, 3 e 4, 4 e 5, 5 e 6, 6 e 7, e 7 e 0.

- Organização circular das janelas de registradores:



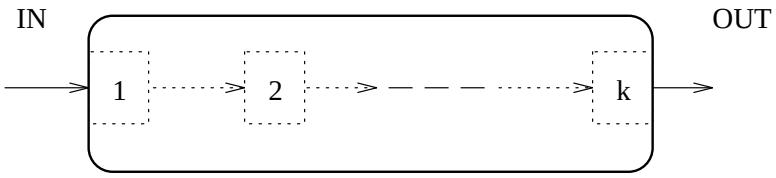
3.6 Pipelines

Pipelining é uma técnica de aceleração de execução de operações inicialmente associada a processadores RISC, mas atualmente utilizada de forma extensa em todos processadores modernos.

Em computadores, *pipelines* são utilizados principalmente no processamento de instruções e de operações aritméticas, que podem ser convenientemente subdivididas em passos menores de execução.

3.6.1 Princípio de operação

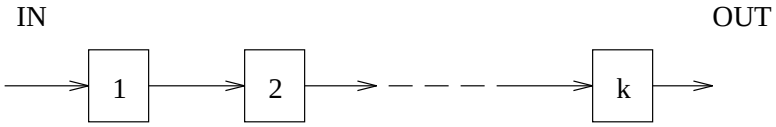
Motivação. Considere uma tarefa que utiliza, para sua execução, um módulo de processamento executando k subtarefas:



No esquema apresentado acima, se cada subtarefa requer um tempo de execução τ segundos, a tarefa como um todo irá ser executada em $k\tau$ segundos.

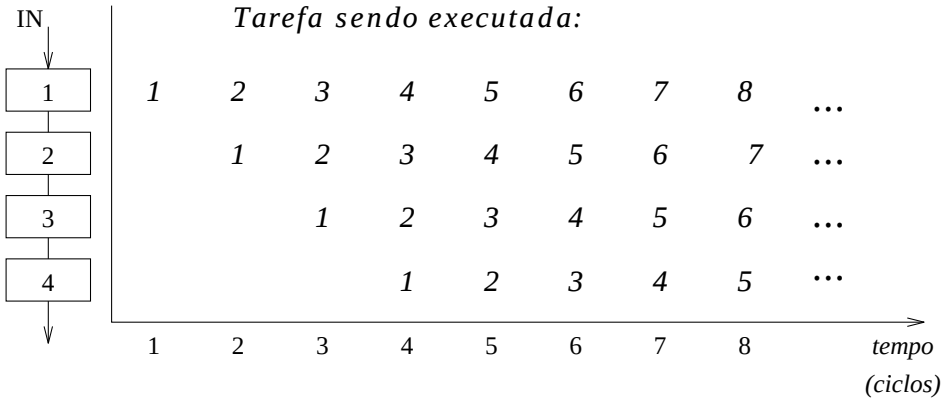
Para a execução de uma sequência de tais tarefas, cada nova tarefa na sequência só poderá ter sua execução iniciada após $k\tau$ segundos, o tempo necessário para a liberação do módulo de processamento.

Princípio de *pipeline*. Considere uma abordagem alternativa na qual o módulo de processamento é dividido em k módulos menores (estágios), cada um responsável por uma subtarefa, cada um operando concorrentemente aos demais estágios:



O princípio da técnica de *pipelining* é poder iniciar 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,



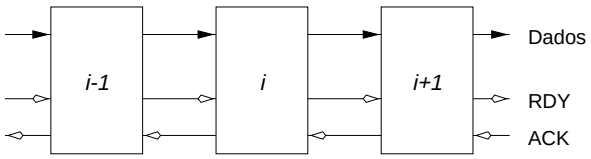
Na execução em pipeline, cada tarefa individualmente ainda requer $k\tau$ segundos. Entretanto, o módulo é capaz de gerar um novo resultado a cada τ segundos

3.6.2 Métodos de transferência de dados

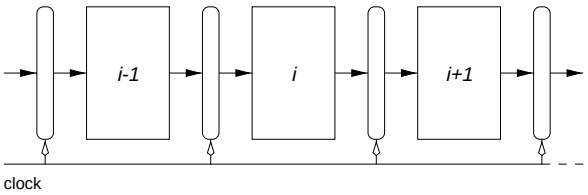
O tempo total para a execução de uma operação em *pipeline* é, em geral, ligeiramente maior que o tempo para executar a mesma operação monoliticamente (sem *pipeline*). Um dos *overheads* associados à operação de um *pipeline* é decorrente da necessidade de se transferir dados entre os estágios.

Há duas estratégias básicas para controlar a transferência de dados entre os estágios de um *pipeline*: o método assíncrono e o método síncrono.

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).



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*.



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.

3.6.3 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

$$L_k = k \times \tau$$

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 executando n operações com um ciclo de τ, o primeiro resultado só estará disponível após um período L_k — é o tempo de "preenchimento", ou *fill-up*, 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

$$H_k = \frac{n}{[k+n-1]\tau}$$

Quando o número de operações executadas tende para um valor muito grande, a vazão tende para o seu valor máximo

$$\lim_{n\to\infty} H_k = \frac{1}{\tau}$$

Fator de aceleração (*Speedup*). Expressa o ganho em velocidade na execução de um grupo de tarefas em um pipeline de k estágios comparada com a execução seqüencial deste mesmo grupo de tarefas.

Para a execução de n tarefas em um *pipeline* de k estágios, sendo o tempo seqüencial $kn\tau$, o fator de aceleração é

$$\begin{aligned} S_k &= \frac{\text{Tempo sequencial}}{\text{Tempo pipeline}} \\ &= \frac{nk}{k+n-1} \end{aligned}$$

Speedup é máximo quando o número de operações executadas é grande,

$$\lim_{n\to\infty} S_k = k$$

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,

$$\begin{aligned} E_k &= \frac{S_k}{k} \\ &= \frac{n}{k+(n-1)} \end{aligned}$$

A eficiência é máxima quando o número de operações executadas é grande,

$$\lim_{n\to\infty} E_k \to 1$$

3.6.4 Pipelines de instruções

O processamento de instruções é um dos maiores beneficiários do uso das técnicas de *pipelining*. O ciclo de máquina de von Neumann é facilmente decomposto como uma seqüência de passos menores, sendo um forte candidato à execução em *pipelines*.

A decomposição do processamento de uma instrução em passos elementares não é única, havendo várias abordagens possíveis. Exemplos:

Dois estágios: FETCH e EXECUTE;

Três estágios: FETCH, DECODE e EXECUTE;

Cinco estágios: FETCH, DECODE, FETCHOPER, EXECUTE, STORE;

Seis estágios: FETCH, DECODE, GENADDRESS, FETCHOPER, EXECUTE, STORE.

As variantes possíveis são muitas, tais como incluir um estágio INCREMENTPC ou mesmo particionar estágios com ciclos mais longos em estágios elementares, como ter dois ou três estágios para EXECUTE.

Máquinas RISC dependiam extensivamente de *pipelines* de instrução para atingir a meta de taxas de execução próximas a uma instrução por ciclo. Entretanto, há três grandes causas para a degradação de desempenho em *pipelines* de instruções:

1. Conflitos na utilização de recursos;
2. Dependências de controle em procedimentos;
3. Dependências de dados entre instruções.

Conflitos de recursos

Exemplos típicos de conflitos incluem:

Acesso à memória: busca de instrução, busca de operandos e armazenamento de resultados estão sujeitos a conflitos na utilização da memória e do barramento do sistema;

Unidades de execução: incremento do registrador PC e execução de operações aritméticas podem ambos requerer o uso simultâneo da ULA.

A solução para este tipo de conflitos usualmente envolve a duplicação de recursos, maior uso de registradores (restringindo acessos à memória principal a um único estágio, *fetch* de instrução) e a utilização de memória *cache* sob a forma de arquitetura Harvard, com separação entre *caches* e barramentos internos de dados e de instruções.

Dependências de controle

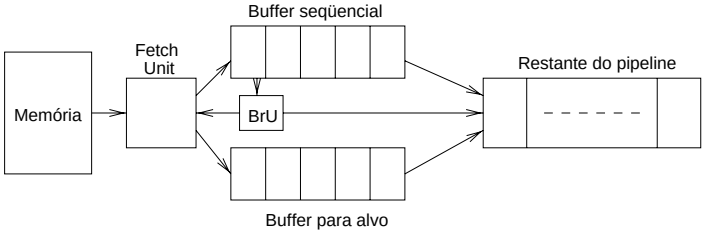
Problemas de dependência de controle estão associados principalmente a instruções de desvio em programas, principalmente a desvios condicionais. Quando a instrução é um desvio, não há como o primeiro estágio do *pipeline* conhecer o endereço da próxima instrução, uma vez que este valor só

estará disponível após a execução da operação em algum estágio posterior. Quando o endereço alvo de um desvio condicional é computado no *pipeline*, várias outras instruções podem estar presentes nos estágios anteriores. Se nenhum mecanismo especial for adotado, estas instruções deverão ser descartadas sempre que o endereço alvo do desvio for diferente da posição seguinte de memória.

O período durante o qual a saída do *pipeline* permanece ociosa devido ao descarte das operações pós-desvio é denominado uma *bolha* do *pipeline*. Como estatísticas mostram que desvios condicionais correspondem de 10 a 20% das instruções de programas típicos, a degradação decorrente desse problema não é desprezível.

Interlock. A técnica "ingênua" para se lidar este problema é o *interlocking*. Quando se detecta que a instrução é um desvio, a entrada de novas instruções no *pipeline* é bloqueada até a definição do endereço alvo da instrução de desvio. Obviamente, *interlock* é uma estratégia que impõe severa degradação no desempenho do *pipeline*, não sendo portanto uma solução para o problema.

Buffers de instruções. Outra estratégia envolve a utilização de *buffers* de instruções entre o estágio de FETCH e os demais estágios. Para lidar com o problema de desvios condicionais, *hardware* especial é incorporado para detectar instruções de desvio e computar os endereços alternativos neste estágio. Enquanto o endereço alvo do desvio não for computado (por um estágio especial, BRU, para acelerar o procedimento), os dois fluxos alternativos de instruções são transferidos para os *buffers*:



Além de exigir a duplicação de *hardware*, esta estratégia não resolve o problema de desvios encadeados, ou seja, quando a instrução transferida da memória para o *pipeline* antes da resolução do desvio também for um desvio. Para tratar n instruções de desvio simultaneamente, 2^n *buffers* seriam necessários.

Branch folding. Nesta técnica, um *buffer* de instruções é inserido entre o estágio de FETCH/DECODE e o estágio de EXECUTE. Neste *buffer*, para cada instrução decodificada está também o endereço da próxima instrução. Quando uma instrução de desvio é computada, o endereço alvo "cobre" o campo de endereço da instrução anterior no *buffer*. Para desvios condicionais, dois campos de endereços são mantidos. Um dos endereços é selecionado pela unidade de execução como endereço alvo tentativo, sendo utilizado para buscar as próximas instruções; o outro endereço é mantido até que o endereço alvo efetivo possa ser computado. Se a escolha foi correta, o processamento continua normalmente. Caso contrário, as instruções no *pipeline* são descartadas.

Branch target buffer. Esta técnica está baseada na existência de uma tabela de *lookup* de alta velocidade de acesso que mantém endereços alvos associados a instruções de desvio. Quando a

instrução de desvio é encontrada pela primeira vez, uma predição inicial é feita e inserida na tabela. Se a predição foi correta, não há penalidade ao desempenho; caso contrário, as instruções posteriores no *pipeline* são descartadas e a entrada associada ao endereço alvo daquela instrução na tabela de *lookup* é atualizada. Da próxima vez que essa instrução de desvio for encontrada, o endereço alvo da tabela é utilizado como predição. Esta estratégia é adequada para instruções de desvio associadas a iterações.

Delayed branch. Nesta técnica, o programador ou compilador posiciona L instruções após a instrução de desvio para serem executadas independentemente do resultado do teste. Em outros termos, buscam-se instruções anteriores à instrução de desvio (no corpo de uma iteração, por exemplo) que não afetem a condição do desvio, sendo estas instruções colocadas após a instrução de desvio. Por exemplo, em um processador com $L = 1$, a seqüência

```
        ADD  R3,R4,R5
        SUB  R2,R2,1
        BEQ  R2,R0,L1
             .
             .
L1:          .
```

poderia ser convertida pelo compilador para

```
        SUB  R2,R2,1
        BEQD R2,R0,L1
        ADD  R3,R4,R5
             .
             .
L1:          .
```

onde `BEQD` é uma versão *delayed* da instrução `BEQ`. Alguns processadores apresentam apenas a instrução de desvio com *delay*, sendo que nestes casos instruções `NOP` *(no operation)* são introduzidas após o desvio quando não é possível transferir outras instruções para aquela posição. Idealmente, em um *pipeline* de k estágios, $L = k - 1$ para evitar a criação de bolhas.

Predição estática. Esta técnica baseia-se no conhecimento do programador ou compilador sobre o comportamento do programa. Por exemplo, em instruções de desvio associadas a iterações é alta a probabilidade do desvio ocorrer. Por outro lado, em instruções de desvio associadas a condições de erro esta probabilidade é menor. Assim, na codificação da instrução de desvio haveria um *bit* adicional indicando a "preferência" do desvio.

Dependências de dados

Quando operandos de uma instrução dependem de operandos de outra instrução, diz-se que há uma depêndencia de dados entre as instruções. Há três tipos de dependências que podem ocorrer: a dependência de fluxo de dados, a antidependência e a dependência de saída.

Dependência de fluxo. Esta é a forma de dependência onde o operando de uma instrução depende diretamente do resultado de uma instrução anterior, como em

```
ADD R3,R2,R1          ; R3 = R2 + R1
SUB R4,R3,1           ; R4 = R3 - 1
```

Neste caso, o valor de R3 só será conhecido após a conclusão da instrução ADD. Este tipo de dependência é também denominado de *hazard leitura-após-escrita* (RAW) ou de dependência real. A ocorrência deste tipo de dependência pode ocasionar a criação de bolhas no *pipeline*, a não ser que o programador ou compilador consiga inserir instruções independentes entre as duas instruções com dependência de fluxo de dados.

Antidependência. Também conhecida como *hazard escrita-após-leitura* (WAR), esta forma de dependência ocorre quando uma instrução posterior atualiza uma variável (registrador) que é lida pela instrução anterior, como acontece com R2 em

```
ADD R3,R2,R1          ; R3 = R2 + R1
SUB R2,R4,1           ; R2 = R4 - 1
```

O cuidado que se deve tomar é não permitir que o conteúdo de R2 seja alterado pela instrução SUB antes de ser utilizado pela instrução ADD. O impacto deste tipo de dependência em um *pipeline* depende muito de sua estrutura, não sendo um problema na maior parte dos casos. Outra situação onde a antidependência pode ocasionar problemas é quando a ordem seqüencial de execução de instruções não é garantida.

Dependência de saída. Também conhecida como *hazard escrita-após-escrita* (WAW), esta forma de dependência ocorre quando duas instruções atualizam uma mesma variável, como em

```
ADD R3,R2,R1          ; R3 = R2 + R1
SUB R2,R3,1           ; R2 = R3 - 1
ADD R3,R2,R5          ; R3 = R2 + R5
```

Neste exemplo, além da dependência de fluxo e da antidependência entre as instruções consectivas, há uma dependência de saída entre a primeira e a terceira instrução. Este tipo de dependência não causa problemas quando se garante que a ordem das instruções é preservada durante a execução. É também uma forma de conflito de recursos, que pode ser eliminado utilizando um outro registrador para armazenar o resultado da terceira instrução.

Condições de Bernstein Em 1966, Bernstein estabeleceu um conjunto de condições que indicam matematicamente a independência entre instruções. Sejam $O(x)$ e $I(x)$ os conjuntos de dados de escrita (saída, contra-domínio) e de leitura (entrada, domínio) associados à instrução x, respectivamente. Duas instruções i e j são independentes quando:

$$O(i) \cap I(j) = \emptyset$$
$$I(i) \cap O(j) = \emptyset$$
$$O(i) \cap O(j) = \emptyset$$

A primeira condição indica a ausência de *hazards* de leitura-após-escrita; a segunda, escrita-após-leitura; e a terceira, de escrita-após-escrita. A falha em qualquer uma das três condições indica um potencial de *hazard*, sendo condição necessária mas não suficiente para sua ocorrência.

As condições de Bernstein podem ser também utilizadas para grupos de instruções ou processos.

Interlock

A técnica de *interlocking* também pode ser utilizada para lidar com *hazards* em *pipelines*. Nesta técnica, um bit é associado a cada registrador de operandos para indicar (por exemplo, quando 1) que o conteúdo é válido. Quando uma instrução que foi buscada da memória vai escrever em um registrador, ela verifica o valor do bit de validade. Se estiver setado, então o bit é resetado para indicar que o valor será alterado; caso contrário, a instrução deve aguardar o bit ser alterado antes de prosseguir. Esta condição de espera é necessária para evitar *hazards* de escrita após escrita. Quando o resultado da instrução é gerado o *bit* é setado, liberando o acesso do registrador para leitura por outras instruções.

Forwarding

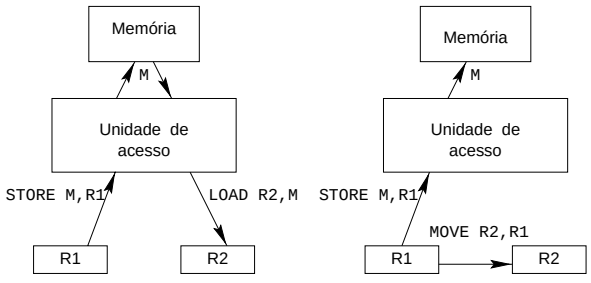
Data forwarding é uma técnica que permite acelerar o desempenho de *pipelines* oferecendo alternativas para acelerar a transferência de dados entre instruções, evitando acessos desnecessários à memória. Algumas destas técnicas podem ser aplicadas pelo compilador na fase de otimização.

Store-Load. Quando uma instrução `STORE` é seguida por outra instrução `LOAD` referente à mesma posição de memória, a segunda instrução pode ser substituída por `MOVE` entre registradores. Por exemplo,

```
STORE  M,R1
LOAD   R2,M
```

ficaria

```
STORE  M,R1
MOVE   R2,R1
```

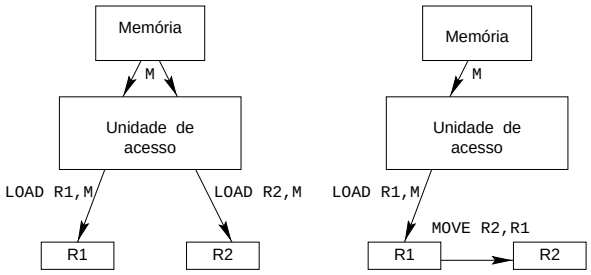


Load-Load. Quando duas instruções `LOAD` referem-se a uma mesma posição de memória, a segunda instrução é substituída por uma transferência entre registradores. Por exemplo,

```
LOAD   R1,M
LOAD   R2,M
```

ficaria

```
LOAD   R1,M
MOVE   R2,R1
```

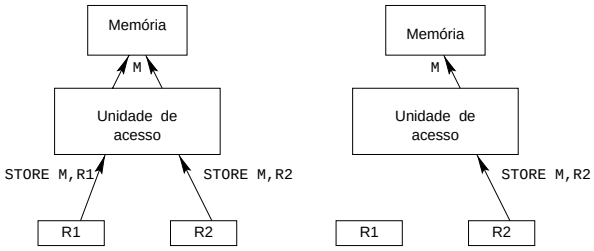


Store-Store. Quando duas instruções `STORE` consecutivas armazenam valores na mesma posição de memória, a primeira operação pode ser eliminada, pois seu resultado será superposto pelo resultado da segunda instrução. Por exemplo,

```
STORE  M,R1
STORE  M,R2
```

ficaria simplesmente

```
STORE  M,R2
```



Internal forwarding

O conceito de *forwarding* pode ser estendido para transferência de dados entre estágios de *pipelines*, reduzindo o impacto de *hazards* no desempenho. Neste caso, é preciso adicionar lógica e barramentos internos para detectar e explorar as transferências entre estágios do *pipeline*.

Considere a execução do seguinte trecho de código em um *pipeline* com estágios FETCHOPER, EXECUTE e STORE:

```
ADD R3,R2,R0
SUB R4,R3,8
```

A referência a R3 na segunda instrução, sem o uso de *internal forwarding*, implica em um atraso de dois ciclos no *pipeline*, pois apenas após a instrução ADD completar STORE é que a instrução SUB poderá prosseguir com FETCHOPER.

Internal forwarding oferece uma alternativa para essa situação onde um caminho é estabelecido entre a saída do estágio STORE para a entrada do estágio EXECUTE; a utilização deste caminho e a detecção de quando isto poderia acontecer ocorre por *hardware*. Deve-se observar que esta solução não elimina a bolha no *pipeline*, porém reduz o atraso de dois para um ciclo, neste caso.

Pipelines em processadores multithreaded

Problemas de *hazards* e dependências entre instruções ocorrem porque instruções em execução no *pipeline* pertencem a um mesmo processo. Entretanto, muitos computadores são utilizados em ambientes multi-usuários, onde vários processos podem estar sendo executados concorrentemente.

Processadores *multithreaded* operam com instruções de diversos processos simultaneamente. Cada *thread* é representada por um contexto, que consiste de um registrador contador de programa, um conjunto de registradores e um registrador de *status* do contexto. Por exemplo, um processador com capacidade de operar quatro *threads* (linhas de execução) simultâneas deveria oferecer quatro conjuntos de registradores independentes, um para cada *thread*.

A execução de instruções de diferentes *threads* no *pipeline* permite eliminar as interdependências entre as instruções. Entrelaçando *(interleaving)* a sequência de instruções executadas, cada estágio do *pipeline* estaria executando uma instrução de contexto diferente, sem risco de *hazards*. Por exemplo, é possível eliminar dependências em um *pipeline* de até quatro estágios simplesmente alternando a execução de instruções de quatro *threads* diferentes.

Idealmente, o número de *threads* deve ser pelo menos igual ao número de estágios no *pipeline* para atingir a sua máxima taxa de operação.

3.6.5 Pipelines aritméticos

Pipelines aritméticos são utilizados para a execução de operações aritméticas complexas, tais como multiplicação inteira, operações em ponto flutuante (multiplicação, adição) e operações vetoriais, que podem ser divididas em etapas de execução menores.

Pipelines aritméticos apresentam como vantagens a simplificação de circuitos necessários para implementar as operações complexas, permitindo o compartilhamento de estágios comuns do *pipeline* para a realização de operações distintas. A reutilização de estágios para executar uma mesma operação é outra forma de simplificação do *hardware*, levando à criação de *pipelines* **não-lineares**, ou *feedback pipelines*. Um *pipeline* sem realimentações de dados para estágios anteriores é classificado um *pipeline* **linear**.

Outro critério de classificação de *pipelines* aritméticos referem-se a quantas operações o *pipeline* pode executar. *Pipelines* **unifuncionais** são dedicados à execução de uma única operação; *pipelines* **multifuncionais** podem executar mais de uma operação. Neste caso, o circuito controlador do *pipeline* deve direcionar os dados entre as interconexões dos estágios do *pipeline* de acordo com a função sendo executada.

Operações em ponto flutuante

A utilização de *pipelines* para a realização de operações aritméticas será ilustrada inicialmente pela definição dos estágios para a execução de multiplicação em ponto flutuante.

A entrada para o *pipeline* são dois operandos em ponto flutuante, $(M_1 E_1)$ e $(M_2 E_2)$, com representação normalizada.

As etapas de execução da operação de multiplicação são:

1. Adicionar expoentes para obter o expoente de saída,

$$E_o = E_1 + E_2$$

2. Multiplicar mantissas,

$$M_o = M_1 \times M_2$$

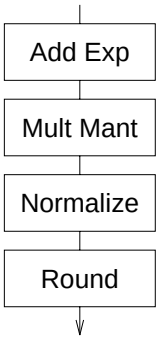
   O resultado M_o tem o dobro do número de bits da representação adotada para ponto flutuante

3. Normalizar resultado, se necessário

4. Arredondar o resultado para o comprimento normal

   - em caso de *overflow*, renormalização pode ser necessária.

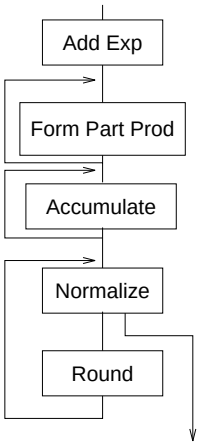
Associando a cada etapa um estágio do *pipeline*, obtém-se:



Refinamento do *pipeline*:

- Estágio *Multiply Mantissas* poderia ser implementado como um circuito combinacional monolítico. As desvantagens nesta abordagem seriam:
  - complexidade do circuito
  - tempo longo deste estágio iria determinar a máxima velocidade de operação do *pipeline*
- Abordagem alternativa: implementar este estágio por uma sequência de operações mais simples (e rápidas):
  - geração de produtos parciais
  - soma dos produtos parciais

- *Pipeline* resultante, também dividindo o estágio *Round* em operações mais elementares *Round* e *Normalize*:



- *Pipeline* resultante é um exemplo de *pipeline não-linear*
  - tem estrutura mais complexa
  - execução de uma nova operação pode colidir com a execução de outra operação em alguns dos estágios do *pipeline* devido à realimentação entre estágios

Outro exemplo de operação em ponto flutuante adequada para execução em *pipeline* é a adição de dois números normalizados, $N_1 = (M_1, E_1)$ e $N_2 = (M_2, E_2)$:

Passo 1. Comparar expoentes E_1 e E_2;

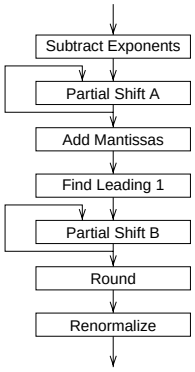
Passo 2. Alinhar mantissas M_1 e M_2 para igualar expoentes;

Passo 3. Adicionar mantissas;

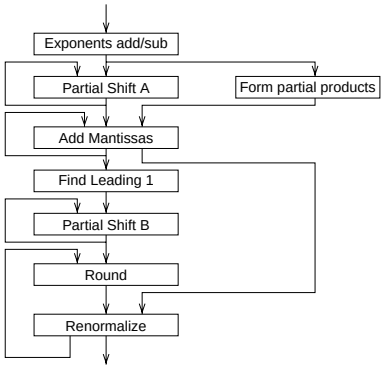
Passo 4. Normalizar mantissa resultante, ajustando expoente resultante;

Passo 5. Arrendondar e renormalizar, se necessário.

Um possível *pipeline* exclusivo para esta operação seria:



Como é possível observar, há muitos estágios comuns entre os *pipelines* propostos para as duas operações. Assim, suas estruturas podem ser combinadas para a formação de um *pipeline* multifuncional:



3.6.6 Controle de pipelines

Em um *pipeline* não-linear, é preciso incorporar algum mecanismo para indicar se nova operação pode ou não ser iniciada em um dado ciclo. O objetivo deste mecanismo é evitar colisões entre operações em progresso no *pipeline*.

A descrição da ocupação do *pipeline* é feita através de uma *tabela de reserva*:

- diagrama espaço-tempo descrevendo os estágios ocupados em cada ciclo para a realização de cada operação;
- linhas do diagrama correspondem a estágios do pipeline;
- colunas do diagrama correspondem a ciclos (escala de tempo);

Para o exemplo do *pipeline* para multiplicação em ponto flutuante, considerando que o produto das mantissas é formado através de duas passagens pelo estágio correspondente, a tabela de reserva seria

	1	2	3	4	5	6	7
Add Exp	X						
Form Part Prod		X	X				
Accumulate			X	X			
Normalize					X		X
Round						X	

Vetor de colisão

A partir da tabela de reserva de ocupação do pipeline, é possível derivar um *vetor de colisão* utilizado no controle da iniciação de novas operações. O vetor de colisão é um vetor binário, que contém um 1 na posição i do vetor (da esquerda para a direita) se uma nova operação pode ser iniciada i ciclos após o início da operação corrente.

O vetor de colisão é obtido a partir da tabela de reserva, podendo ser obtido por inspeção ou através da análise de **latências proibidas**. A latência é determinada pela distância entre dois inícios de operações no *pipeline*. Assim, uma latência ℓ significa que o início de duas operações estão separadas por ℓ ciclos. As latências proibidas são aquelas que causariam colisões no *pipeline*. Para obter quais são as latências proibidas de um *pipeline* a partir de sua tabela de reserva, basta verificar a distância entre todos os pares de marcas em uma mesma linha, ou seja, em um mesmo estágio.

O vetor de colisão resume o conjunto de latências proibidas e permitidas. Um bit na posição i do vetor de colisão tem 1 se a latência i é proibida, 0 se a latência i é permitida. Por convenção, adotaremos aqui a i-ésima posição da esquerda para a direita.

No exemplo da tabela para o *pipeline* de multiplicação, na segunda e terceira linhas obtém-se que 1 é uma latência proibida, e na quarta linha obtém-se que 2 também é uma latência proibida. Assim, o vetor de colisão para este *pipeline* é $C = 11$, indicando que:

1. nova operação não pode ser iniciada um ciclo após o início de uma operação por conflito no segundo estágio (formação dos produtos parciais)

2. nova operação não pode ser iniciada dois ciclos após o início de uma operação por conflito no estágio de normalização de operandos

3. entre três e seis ciclos após o início de uma operação, outras operações podem ser iniciadas sem conflito *com esta* operação

Implementação do controle

O controle de um *pipeline* pode ser implementado através de um controlador formado por um *shift register* S, portas OR e um registrador com o vetor de colisão C:

1. acesso ao pipeline é liberado se bit saindo de S é 0; caso contrário, a requisição deve aguardar até o próximo ciclo

2. se requisição foi aceita, novo valor de S será

$$S = (S \ll 1) + C$$

   onde $\ll b$ denota a operação de deslocamento à esquerda de b bits (com 0 entrando à direita) e $+$, a operação OR

3. se nenhuma requisição foi aceita, novo valor de S será simplesmente

$$S = (S \ll 1)$$

Observe que o tempo de espera para que uma requisição seja aceita é no máximo igual ao comprimento do vetor de colisão.

Diagramas de estado

Os diagramas de estado resumem todos os estados válidos possíveis do *shift register* do controle do pipeline e quais as transições que levam a cada estado. Nós no diagrama correspondem ao conteúdo do *shift register* após cada iniciação de operação válida. Arcos no diagrama correspondem às transições entre estados, tendo como peso a latência requerida para iniciar a operação que levou à transição.

Diagramas de estado são construídos pelas seguintes regras:

1. O estado inicial é o vetor de colisão;
2. A partir de cada estado não analisado, indica o próximo estado no *pipeline* (ou seja, o conteúdo do *shift register*) se uma novas operação fosse iniciada em $t = i$ ciclos, onde i é cada posição na representação do estado com valor 0.

A partir da análise dos diagramas de estado, é possível obter qual a melhor estratégia de iniciar as operações para cada *pipeline*. Para tanto, é preciso:

1. analisar os ciclos do diagrama de estados
2. selecionar o ciclo que resulta em mínima latência média (MAL), onde a latência média (AL) é

$$\text{AL} = \frac{\sum \text{latências em cada arco no ciclo}}{\text{número de arcos no ciclo}}$$

Para analisar os ciclos, uma simplificação no método é restringir-se aos ciclos simples, onde cada estado aparece uma vez. Caso contrário o problema seria intratável, uma vez que o número total de ciclos não simples é infinito. Em especial, é preciso analisar os *ciclos gulosos*, que são os ciclos que se iniciam em cada estado seguindo o arco de menor latência.

Propriedades do ciclo MAL:

1. valor da MAL é superior ou igual ao máximo número de marcas em qualquer linha da tabela de reserva
2. valor da MAL é inferior ou igual à latência média de qualquer ciclo guloso do diagrama de estados
3. a latência média de qualquer ciclo guloso — e por consequência o valor da MAL — é inferior ou igual ao número de 1's no vetor de colisão inicial mais 1.

Exemplo

Considere o seguinte *pipeline* de quatro estágios:

A	X	X						X
B			X				X	
C				X	X			
D						X		

As latências proibidas são:

Estágio A: 1, 6 e 7

Estágio B: 4

Estágio C: 1

Estágio D: (nenhuma)

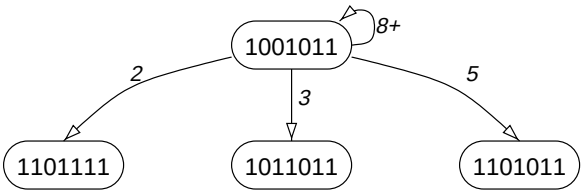
Portanto, o vetor de colisão para este *pipeline* é:

$$C = 1001011$$

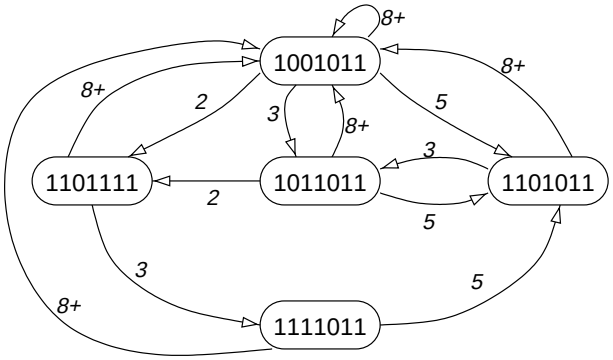
Para a construção do diagrama de estados, deve-se analisar a partir do estado inicial `1001011` os estados gerados quando uma nova operação for iniciada em $t = 2$, em $t = 3$ ou em $t = 5$. Se nova operação for iniciada quando $t = 2$, o estado do *pipeline* seria `0101100` e, após a operação iniciada, o novo estado gerado seria $0101100 + 1001011$, ou seja, `1101111`.

Similarmente, para $t = 3$ o novo estado seria `1011011` e para $t = 5$, `1101011`. Para operações iniciadas em $t \geq 8$, o novo estado seria novamente `1001011`, o próprio vetor de colisão.

Após a geração destes três novos estados, o diagrama de estados parcialmente construído seria:



Repetindo a análise para cada um dos novos estados, o diagrama completo é obtido:



Para este diagrama, um mesmo ciclo guloso é obtido a partir de cada um dos estados possíveis: `(2,3,5,3)`. Assim, obtém-se que a latência média mínima é $MAL = 3,25$ (13 ciclos para completar quatro operações), sendo que o mínimo teórico seria 3, pois há três marcas no estágio A.

Capítulo 4

Sistemas de memória

Memória tem papel crítico no desempenho de um sistema computacional. Ela recebe dados do mundo externo transferidos para processador, e também recebe dados do processador a transferir para mundo externo. Como a cada ciclo de instrução em uma máquina von Neumann requer que o processador obtenha o código da instrução e seus operandos, a velocidade de transferência de itens da memória para o processador é, de fato, crítica.

Uma limitação associada a dispositivos de memória é que um módulo de memória convencional não pode acessar mais que uma palavra durante um ciclo de acesso — é o chamado gargalo de von Neumann.

As principais características tecnológicas que diferenciam os vários tipos de dispositivos de memória incluem custo, tempo e modo de acesso e persistência do armazenamento.

Custo. Está relacionado com a tecnologia utilizada para armazenar os bits de dados e sua velocidade de acesso. Apresenta grandes variações:

- SRAM: $400/Mbyte
- DRAM: $50/Mbyte
- Discos magnéticos: $0.50/Mbyte
- Fitas magnéticas: $0.005/Mbyte

Tempo de acesso. É o tempo para localizar, ler ou escrever um item de informação da (ou para a) memória. Usualmente separa-se tempo de acesso (apenas localização) do tempo de transferência (que depende da dimensão do item). Tempos típicos são da ordem de:

- SRAM: 12–25 ns
- DRAM: 60–70 ns
- Discos magnéticos: 8–10 ms

O tempo de ciclo é o tempo mínimo entre dois acessos consecutivos à memória. Este tempo pode ter que ser maior que o tempo de acesso; por exemplo, dispositivos DRAM requerem um tempo adicional para *refreshing*. É o tempo de ciclo que, juntamente com o tamanho da palavra, define a máxima taxa de transferência (banda de passagem) da memória.

Modo de acesso. Especifica o modo pelo qual a localização pode ser determinada para a transferência do dado:

Acesso a posições aleatórias: o acesso a posições arbitrárias ocorre em tempo independente da posição acessada. Por exemplo, dispositivos semicondutores de memória (RAM).

Acesso serial: acesso ocorre apenas em seqüências pré-estabelecidas, como ocorre com fitas magnéticas.

Acesso direto (ou semi-aleatório): combina características de acesso aleatório e seqüencial. O acesso a uma posição arbitrária é possível, porém afeta consideravelmente o tempo de acesso. Por exemplo, discos magnéticos.

Acesso associativo: definição da posição acessada depende de parte do conteúdo armazenado (chave).

Alterabilidade. Caracteriza se o processo de escrita de um item de informação é reversível ou não. Memórias de escrita e leitura podem ter seus conteúdos livremente alterados, como em memórias RAM e discos magnéticos. O processo irreversível é caracterizado por memórias apenas de leitura, como ROM, PROM, cartões perfurados e discos ópticos WORMs.

Permanência de armazenamento. Caracteriza se dispositivo de memória é capaz de manter informação armazenada na ocorrência de falta de energia para o sistema (armazenamento volátil vs. armazenamento persistente) e se dispositivo de memória é capaz de manter informação sem *refreshing* (memória estática vs. memória dinâmica).

Características físicas. Incluem a tecnologia de integração, a densidade de armazenamento (bits/área ou bits/volume), o consumo de energia a a confiabilidade do dispositivo.

4.1 Hierarquia de memória

Um dos problemas associados ao projeto de sistemas de memória é a **latência** de memória, definido como o tempo decorrido entre a requisição de um item de dado pelo processador e o efetivo recebimento deste item. Portanto, a latência é determinada pelo período que o processador deve esperar para que a memória esteja habilitada a responder à sua solicitação, pelo tempo de acesso ao dispositivo de memória e pelo período de transferência dos dados.

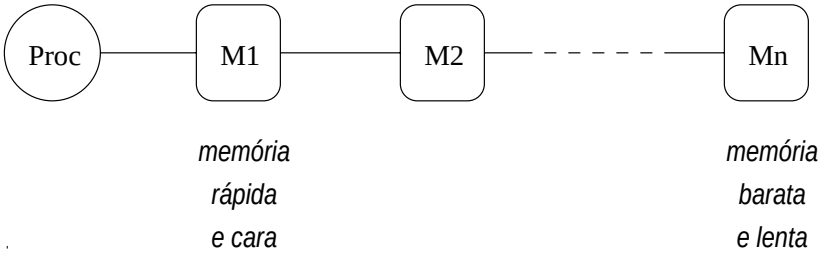
Fatores que afetam a latência incluem:

Fator tecnológico: disparidade entre a velocidade de processamento e a velocidade de acesso à memória;

Fator estrutural: em sistemas com mais de um processador, concorrência de acesso à memória pode atrasar recebimento do item por contenção na memória ou na rede de interconexão.

Para atingir boa banda de passagem de memória a custo razoável, utiliza-se a estrutura de *memória hierárquica*. O objetivo desta estrutura é estabelecer um sistema de memória que aparenta ao usuário ter grande capacidade de armazenamento e alta velocidade de acesso a um custo razoável. A dificuldade de se atingir tais características com um único tipo de dispositivo de memória é que quanto mais rápido for um dispositivo de memória, mais alto será seu custo.

A alternativa oferecida pela hierarquia de memória requer a estruturação em níveis de memória $M_1, M_2, \ldots, M_n$ tal que M_i é mais rápida porém menor que M_{i+1}:



Registradores do processador podem ser considerados como o nível 0 da hierarquia de memória.

4.1.1 Propriedades de uma hierarquia de memória

As principais propriedades de um sistema de hierarquia de memória são:

- Inclusão
- Coerência
- Localidade de referência

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 $M_{i+1}, M_{i+2}, \ldots, M_n$; 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 em M_{i+1} 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, faz com que estes acessos estejam agrupados em certas regiões (ou sequências) 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 subrotinas, 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 seqüencial, pela qual elementos em posições consecutivas da memória têm chance de serem acessados em seqüência, 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.

4.2 Memória *cache*

O princípio de funcionamento da memória *cache* é duplicar parte dos dados contidos na memória principal (a memória lenta, neste caso) em um módulo menor (o *cache*) composto por dispositivos de memória mais rápidos.

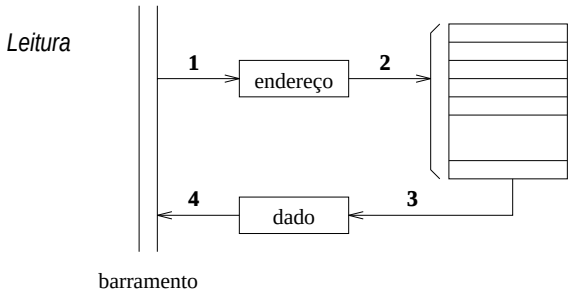
Quando o processador solicita um item de dado (gerando uma referência para seu endereço, que pode ser físico ou virtual), o gerenciador de memória requisita este item do *cache*. Duas situações podem ocorrer:

***cache hit*:** item está presente no *cache*, é retornado para o processador praticamente sem período de latência;

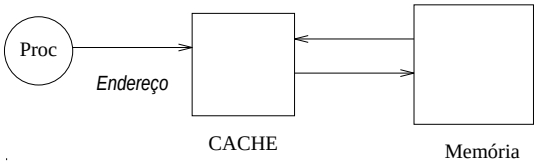
***cache miss*:** item não está presente no *cache*, processador deve aguardar item ser buscado da memória principal.

4.2.1 Localização de itens no *cache*

Em um acesso "convencional" à memória principal, um item buscado é selecionado com base em seu **endereço** na memória:



A idéia por trás do princípio do *cache* é trazer os dados da memória principal que estão sendo mais utilizados para a memória rápida e mais próxima do processador:

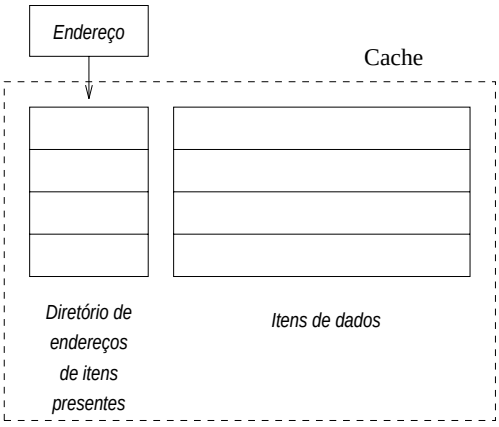


Como se pode observar deste diagrama, a presença do *cache* é transparente para o processador. Se o item solicitado está presente *(cache hit)*, ele é entregue à CPU com baixa latência. Caso esteja ausente, um ciclo de acesso convencional à memória principal é realizado, com o processador recebendo o item após um período de espera (latência alta).

Entretanto, há ainda um problema de localização de item no *cache*. Ao copiar um item da memória principal para o *cache*, a informação sobre a posição (endereço) do item na memória principal não é válida para a posição do item no *cache*. Como saber então se o item buscado está presente ou não no *cache*?

Estrutura básica

Para cada item de dado armazenado no *cache*, armazena-se também uma parte do endereço (o *tag*) do item na memória principal. O conjunto de *tags* armazenados constitui o **diretório** do *cache*; o item é armazenado em uma área de dados associada àquele *tag*:



Quando o módulo de *cache* recebe uma requisição de um item do processador, o endereço do item é buscado entre os endereços dos itens (na verdade, entre seus *tags*) armazenados no *cache*. Se

o *tag* estiver presente no diretório, o item solicitado é entregue. Se o *tag* não estiver no diretório, sinaliza-se um *cache miss* e um bloco da memória principal contendo o item solicitado é transferido para o *cache*. Pode ser necessário, neste caso, remover um bloco presente no *cache* para abrir espaço para o novo bloco transferido.

A busca de um endereço na área de diretório de um *cache* requer uma comparação do endereço buscado com os *tags* armazenados no diretório, o que caracteriza um modo de acesso **associativo**. Na Seção 4.2.2 serão apresentados os mecanismos para realizar este tipo de acesso de forma eficiente.

Em geral, quando um item é trazido da memória principal para o *cache*, alguns itens armazenados nas posições vizinhas também são trazidos para o *cache*. É a forma de aproveitar a propriedade de localidade seqüencial. O conjunto de itens que é transferido entre memória principal e *cache* é chamado de bloco ou uma **linha** do *cache*. O número de palavras em uma linha é um dos parâmetros no projeto do *cache*.

Tempo de acesso efetivo

Como foi observado, a latência para operações de leitura ao cache varia entre os dois valores correspondentes às situações de *cache hit* ou *miss*. Seja t_c o tempo de acesso a um item no módulo *cache*, e seja t_l o tempo de acesso a uma linha na memória principal. As duas situações possíveis são:

Leitura com *hit*: neste caso, o tempo de acesso visto pelo processador é t_c;

Leitura com *miss*: neste caso, o tempo observado pelo processador é $t_c + t_l$.

Seja h é a probabilidade de item buscado estar presente no *cache*, ou seja,

$$h = \frac{\text{número palavras encontradas no } \textit{cache}}{\text{número total de referências à memória}}$$

O parâmetro h é usualmente chamada de **taxa de acerto**, ou *hit ratio*.

Então, o tempo de acesso efetivo ao *cache* t_{eff} pode ser expresso por

$$t_{\text{eff}} = t_c + (1 - h)t_l$$

O termo $1 - h$ é usualmente denominado a **taxa de ausência** do *cache* (*miss ratio*).

No caso ideal, $h = 1.0$, o que implicaria em $t_{\text{eff}} = t_c$. Para valores muito baixos de h, a presença do *cache* acarretaria em degradação no desempenho de acesso à memória, com $t_{\text{eff}} > t_c$. Tipicamente, o valor de h está em torno de 0.85, embora este valor varie sensivelmente de acordo com o tipo de aplicação. Pequenas variações em h podem afetar significativamente o tempo efetivo de acesso.

Em geral, expressa-se a latência em termos de ciclos do processador. Tipicamente, a latência para um *hit* está em torno de 1 a 2 ciclos, enquanto que a latência com *miss* pode variar de 5 a 20 ciclos de CPU.

4.2.2 Organização

É fundamental para o bom desempenho do *cache* que a busca a um item seja realizada em um tempo curto. Por este motivo, o uso de **memórias associativas** (em *hardware*) para implementar o diretório do *cache* não é uma solução viável, pois seu tempo de acesso é longo.

Portanto, é preciso utilizar uma organização interna para o *cache* que permita fazer esta busca rápida usando dispositivos convencionais de memória. Evidentemente, uma busca seqüencial ao diretório é inconcebível. Uma das alternativas para agilizar a busca de um *tag* no diretório do *cache* é reduzir o número de posições que devem ser buscadas.

Existem três estratégias básicas de organização do *cache*:

1. Organização por mapeamento direto;
2. Organização completamente associativa;
3. Organização associativa por conjuntos.

Organização por mapeamento direto

Neste tipo de organização, há apenas uma posição no *cache* para onde um item de dado pode ser transferido.

A busca restringe-se a uma inspeção nesta posição, requerendo apenas a presença de um comparador. Se o *tag* do item solicitado for igual ao *tag* associado àquela posição de memória, o *cache hit* está caracterizado. Caso contrário, como nesta organização o item não pod estar em nenhuma outra posição, o *cache miss* é caracterizado. Então a linha contendo o item solicitado é transferida da memória principal para aquela posição do *cache*.

As vantagens deste tipo de organização incluem:

- simplicidade de *hardware*, e conseqüentemente custo mais baixo;
- não há necessidade de escolher uma linha para ser retirada do *cache*, uma vez que há uma única opção. Isto dispensa a implementação de um algoritmo de troca de linhas; e
- operação rápida

Apesar de simples, esta estratégia é pouco utilizada devido à sua baixa flexibilidade. Como há apenas uma posição no *cache* onde o item solicitado pode estar localizado, a probabilidade de presença no *cache* h pode ser baixa. Assim, pode haver queda sensível no desempenho em situações onde o processador faz referências envolvendo itens que são mapeados para uma mesma posição do *cache*.

Organização completamente associativa

Neste tipo de organização, um item de memória pode estar localizado em qualquer posição do *cache*. Assim, o processo de identificar se um item está ou não presente no *cache* requer uma comparação simultânea do *tag* buscado com todas as linhas do diretório.

A comparação com todas as linhas do diretório requer a utilização de memórias associativas. Por este motivo, a formação de um *cache* totalmente associativo eficiente e economicamente viável é limitado a *caches* de dimensões reduzidas, como em algumas implementações de *cache* interno (no *chip* do processador).

Organização associativa por conjuntos

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

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:

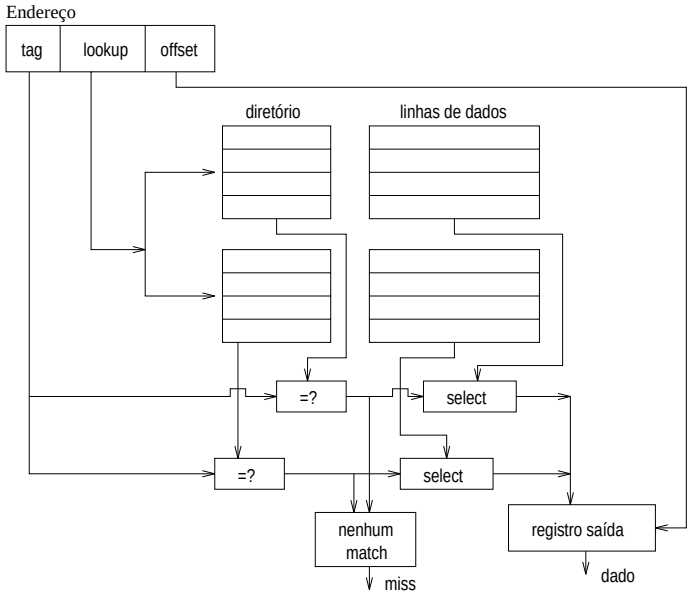
Offset: se uma linha com L bytes contém W palavras, então os $\log_2 W$ bits menos significativos identificam qual dos itens em uma linha está sendo buscado. Se o menor item endereçável for um byte, então $L = W$.

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 um *cache* de 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 do *cache* é $K = 4$, então este *cache* tem espaço para 512 conjuntos. Portanto, o tamanho do *tag* deverá ser 10 bits.

O controlador do *cache* 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. Por exemplo, com quatro conjuntos de duas linhas cada ($N = 4$ e $K = 2$), uma organização típica seria:



4.2.3 Aspectos da organização no projeto

Os principais parâmetros de projeto do *cache* são:

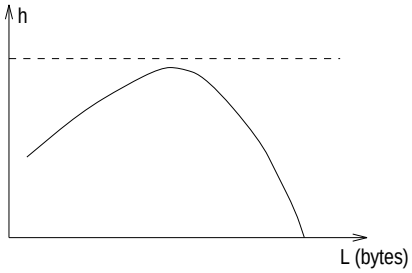
N: número de conjuntos

K: número de linhas em um conjunto

L: número de bytes em uma linha

A dimensão total do *cache* em bytes é LKN. Uma regra empírica determina que, cada vez que o tamanho do *cache* dobra, o número de ausências é reduzido em 30%. No entanto, esta é apenas uma aproximação grosseira, que não dve ser utilizada como referência em um projeto mas simplesmente como uma aproximação inicial.

A organização ideal do *cache* depende da característica de acesso aos dados. Em linha gerais, $K > 1$, porém mantido em um valor baixo; e L é relativamente baixo. Se L é muito baixo, aumenta custo proporcional do diretório; por outro lado, se for muito alto, pode trazer itens que não serão utilizados (perda da localidade):



O valor de N é em geral relativamente alto, sendo determinado pela capacidade total do *cache*, dimensão da linha e associatividade. Está portanto diretamente relacionado ao custo alocado para o *cache*.

4.2.4 Políticas de troca de linha

Quando todas as posições disponíveis para uma linha requisitada e ausente do *cache* estão preenchidas, uma das linhas deve ceder lugar à linha requisitada. Portanto, é necessário estabelecer uma política de troca de linhas.

As políticas de troca de linha usuais são LRU e FIFO. LRU *(Least Recently Used)* é a estratégia mais utilizada em sistemas comerciais, sendo passível de implementação em hardware. FIFO *(First In, First Out)*, apesar de permitir uma implementação mais simples, apresenta taxas de ausência cerca de 12% maior que para LRU.

LRU com diretório sombra

O diretório sombra é uma alternativa para contornar deficiência de LRU sob a presença de "ciclos longos".

Nesta abordagem, o cache tem duas áreas de diretório, principal e sombra. Quando um novo item é trazido para o *cache*, o *tag* da linha retirada do *cache* é armazenado no diretório sombra.

Quando um item solicitado ao *cache* está ausente, esta ausência pode ser de duas formas:

transitória: dado não está no *cache* e o *tag* correspondente não está no diretório sombra

sombra: dado não está no *cache* porém seu *tag* correspondente está no diretório sombra

A ocorrência de uma ausência sombra indica que o dado esteve no *cache* há algum tempo e agora está novamente sendo requisitado. Este comportamento pode caracterizar a detecção de um ciclo longo. A linha é trazida para o *cache* e sua entrada é marcada como oriunda de uma ausência sombra. Na próxima vez que um item tiver que ser selecionado para remoção, este item pode receber maior prioridade de permanência no *cache*.

Um bit é necessário para diferenciar ausência sombra de ausência transitória.

Em geral, para evitar as distorções inerentes a um sistema operando exclusivamente com base em prioridades, um esquema combinado (prioridade e LRU) é utilizado.

O custo adicional associado à implementação do diretório sombra é relativamente baixo, pois o maior custo do *cache* corresponde à área de dados.

A utilização do diretório sombra não acarreta em sobrecarga no tempo de processamento do *cache*. Como o diretório sombra só é consultado em caso de ausência do *cache*, haverá um tempo disponível de t_l para seu processamento, enquanto a linha solicitada é transferida da memória principal para a memória *cache*. Portanto, o uso do diretório sombra não acarreta em atrasos adicionais. Ao contrário, este diretório pode até mesmo ser construído com dispositivos não tão rápidos quanto o restante do *cache*, o que pode baratear seu custo.

Os benefícios associados à utilização do diretório sombra refletem-se na forma de uma redução do número de ausências entre 10 a 30%.

4.2.5 Políticas de atualização da memória

Para manter consistência entre *cache* e memória principal, uma das políticas de escrita em hierarquias de memória, *write-through* ou *write-back*, deve ser utilizada quando um dado é atualizado no *cache*.

Entretanto, pode acontecer uma situação em que o dado atualizado não está presente no *cache*. Neste tipo de situação, é preciso adicionalmente definir a política de **alocação em escrita**. As duas alternativas possíveis incluem:

1. *Fetch on write* ou com alocação em escrita: item é trazido para o *cache* após atualizado;
2. *No fetch on write* ou sem alocação em escrita: item é atualizado apenas na memória principal, não sendo trazido para o *cache*.

Write-through

Na estratégia *write-through*, quando um ciclo de escrita ocorre para uma palavra, ela é escrita no *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 um *cache* com a estratégia *write-through* com alocação em escrita, considere o seguinte modelo. Seja t_c o tempo de acesso ao *cache*, t_l o tempo de transferência para uma linha da memória principal para o *cache* e t_m o tempo de acesso a uma palavra da memória principal. Seja ainda h a probabilidade do item referenciado estar na memória e w a fração das referências à memória que correpondem a operações de escrita. As quatro situações que podem ocorrer são:

1. Leitura de item presente no *cache*, em tempo t_c com probabilidade $h \times (1 - w)$;
2. Leitura de item ausente do *cache*, em tempo $t_c + t_l$ com probabilidade $(1 - h) \times (1 - w)$;
3. Atualização de item presente no *cache*, em tempo t_m — uma vez que a atualização no *cache* ocorre concorrentemente com a atualização em memória, o tempo t_c está escondido no tempo maior, t_m. Esta situação ocorre com probabilidade $h \times w$;
4. Atualização de item ausente do *cache*, em tempo $t_m + t_l$ com probabilidade $(1 - h) \times w$.

A partir destas possibilidades, o tempo efetivo de acesso ao *cache* é

$$t_{\text{eff}} = (1 - w)t_c + (1 - h)t_l + wt_m$$

Write-back

Na estratégia *write-back*, quando um ciclo de escrita ocorre para uma palavra, ela é atualizada apenas no *cache*. A linha onde a palavra ocorre é marcada como **alterada**. Quando a linha for removida do 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 do *cache* para a memória.

Seja w_b a probabilidade de uma linha do *cache* ter sido atualizada. Em geral, $w_b < w$ uma vez que uma linha pode conter mais de uma palavra atualizada. Então, a partir de uma análise similar à realizada acima, obtém-se que o tempo efetivo de acesso ao *cache* é

$$t_{\text{eff}} = t_c + (1 - h)(1 - w_b)t_l$$

4.2.6 Diretrizes de projeto

Alguns sistemas suportam módulos separados para *cache* de instruções e para *cache* de dados, com o objetivo de aumentar banda de passagem entre processador e *cache*. Assim, dados e instruções poderiam ser acessados em paralelo. Outra vantagem neste tipo de projeto é que os *caches* separados podem ser colocados mais próximos dos pontos de utilização.

As desvantagens desse tipo de arquitetura Harvard para a memória *cache* incluem:

- Necessidade de manutenção de consistência
  - linhas contendo instruções e dados
  - programas auto-modificantes
  - operandos imediatos
- Potencial para uso ineficiente do espaço
  - um dos *caches* pode permanecer sub-utilizado enquanto o outro pode estar sobre-carregado.

Mais de um nível de *cache* pode co-existir em um sistema, mas dificilmente mais de dois níveis. Em geral, *caches* de dois níveis ocorrem em processadores que implementam *cache on-chip* e um outro módulo de *cache* maior, o *cache* externo, é implementado *off-chip*.

Em *caches* de dois níveis, a política de atualização do primeiro para o segundo nível é em geral *write-through*, e do segundo nível para a memória principal é *write-back*.

4.2.7 Coerência de *cache*

Quando apenas a CPU manipula *cache*, coerência é mantida pelas políticas de escrita. Problema surge quando há outras "entidades" manipulando a memória.

- Coprocessadores (Ex. DMA)
- Outras CPUs

Solução: monitoramento das atividades de memória *(snooping)*. Neste caso, o gerenciador de memória fica "espionando" as atividades de atualização da memória pelo barramento. Quando há operações de escrita à memória por outras entidades, o gerenciador invalida o conteúdo do *cache*.

4.2.8 Impacto para o programador

Cache é um exemplo típico de uma característica da arquitetura interna do sistema, não sendo controlável pelo programador ou projetista do compilador. Por outro lado, conhecimento sobre o *cache* permite que um programador reestruture seu programa de forma a explorar suas características.

Alguns exemplos onde o programador pode explorar características da presença do *cache*:

- multiplicação de grandes matrizes
- algoritmos de busca

Se não levarem em conta existência do *cache*, *thrashing* pode ocorrer.

4.2.9 Considerações finais

Cache é também um exemplo de utilização de algumas regras gerais para sistemas de alto desempenho:

- dados acessados mais frequentemente devem estar fisicamente próximos de onde são usados
- a densidade de tráfego deve ser mantida baixa em interconexões longas e compartilhadas

O mecanismo de *cache* funciona porque programas exibem um tipo de comportamento (localidade) que pode ser explorado. Se o comportamento de programas fosse diferente, esta organização de *caches* poderia falhar consideravelmente. Outros aspectos de programas deveriam então ser explorados para atingir melhor desempenho, principalmente no projeto de processadores para aplicações específicas.

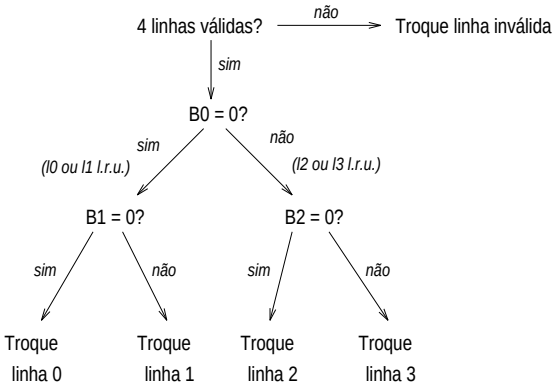
Um dos princípios do *cache* é adaptar-se a fluxos de execução, aprendendo quais itens foram usados e favorecendo aqueles usados recentemente sobre aqueles não usados recentemente. É possível incorporar outros tipos de hardware ao sistema para auxiliá-lo a adaptar-se ao comportamento observado em um fluxo de execução. A questão é *onde* e *que tipo* de hardware usar, e se o *custo* do hardware adicional é justificado pelo ganho de desempenho obtido.

4.2.10 Exemplo

O processador Intel i486 oferece *cache on-chip* com as seguintes características:

- módulo *cache* de 8 Kbytes
- *cache* de dados e instruções
- trabalha com endereços físicos
- organização associativa por conjuntos
  - fator de associatividade $K = 4$
  - linhas de $L = 16$ bytes
  - $N = 128$ conjuntos
  - *tags* de 21 bits
  - um bit de validade associado a cada linha
  - política de escrita *write-through* sem alocação
  - estratégia de troca de linhas: pseudo-LRU

Na estratégia pseudo-LRU, o controlador primeiro busca uma linha inválida para ser removida do *cache*. Se não há linha inválida, usa três bits de LRU (B0, B1 e B2) para selecionar linha a ser trocada:



Além do *cache on-chip*, o dispositivo controlador de *cache* 82485 oferece a possibilidade de se implementar um subsistema de *cache* externo para Intel 486. Esse dispositivo oferece as seguintes características:

- associativo por conjuntos, com $K = 2$
- linhas de $L = 16$ bytes
- estratégia de escrita *write-through* sem alocação
- 2K *tags* de 17 bits
- usa SRAMs 82485MA/MB
- configurações 64K ou 128 Kbytes

Outro dispositivo controlador de *cache* é o 82495, com as seguintes características:

- associativo por conjuntos, com $K = 2$
- diretório de *tags* (4k ou 8K *tags*)
- política de escrita configurável
- manutenção de estado das linhas
- coerência por *snooping*
- usa SRAMs 82490 (256 Kbit), *dual port*, com buffers para *write-back*, *snooping*, acesso à memória
- configurações 128K, 256K ou 512 Kbytes
- tamanho de linha 16, 32 ou 64 bytes
- largura de barramento de memória de 32, 64 ou 128 bits

4.3 Memória virtual

Um dos objetivos buscados na utilização de memória virtual é otimizar o uso da memória principal usando a memória secundária como *backup*. Neste caso, a memória principal é o nível rápido e de pequena capacidade da hierarquia de memória, enquanto que a memória secundária (discos magnéticos, principalmente) constituem o nível lento e de maior capacidade.

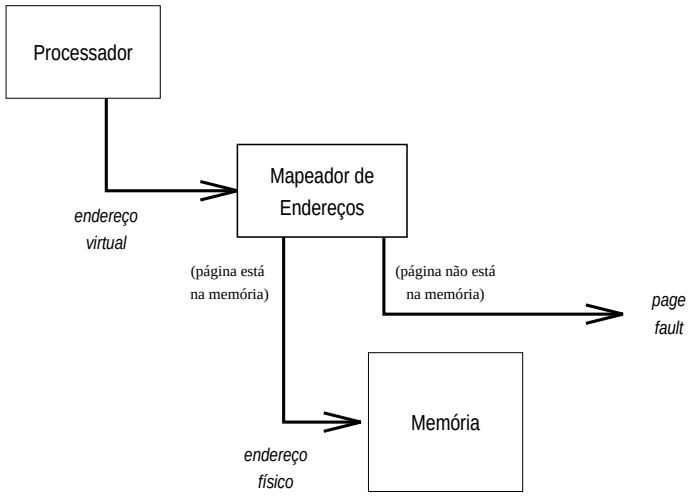
A aplicação típica de memória virtual é mapear um grande espaço de endereçamento a uma memória primária fisicamente menor. Por exemplo, com 32 bits de endereço, o espaço de endereçamento virtual é de 4 Gbytes — memórias primárias têm capacidades da ordem de dezenas de Mbytes. Outra aplicação surge na multiprogramação, onde pode-se mapear um espaço de endereçamento pequeno a uma memória primária maior.

Com a utilização de memória virtual, liberam-se programas e programadores de considerações sobre o posicionamento físico de suas variáveis na memória. Para tanto, opera-se com dois espaços de endereçamento:

espaço físico M: conjunto de endereços da memória principal

espaço virtual V: conjunto de endereços gerados pelo processador

Endereços virtuais precisam ser traduzidos em termos de endereços físicos durante a execução de programas:



O mapeamento é definido por uma função $f_t : V \rightarrow M \cup \{\emptyset\}$, onde endereço virtual $v \in V$ é traduzido por $f_t(v)$:

$$f_t(v) = \begin{cases} m, & \text{se } m \in M \text{ foi alocado para armazenar o dado} \\ & \text{identificado pelo endereço virtual } v \\ \emptyset, & \text{se dado } v \text{ não está em } M \end{cases}$$

Desempenho do sistema de memória virtual depende da eficiência da tradução de endereços. A operação da memória virtual envolve a cooperação do sistema operacional com os recursos do computador. O mapeamento de endereço virtual para endereço físico requer a manutenção da informação sobre quais dados estão em memória.

4.3.1 Conceitos básicos (revisão)

Algumas questões básicas envolvidas no projeto de sistemas de memória virtual são:

1. o tamanho e a natureza dos blocos de informação que são transferidos entre as memórias primária e secundária

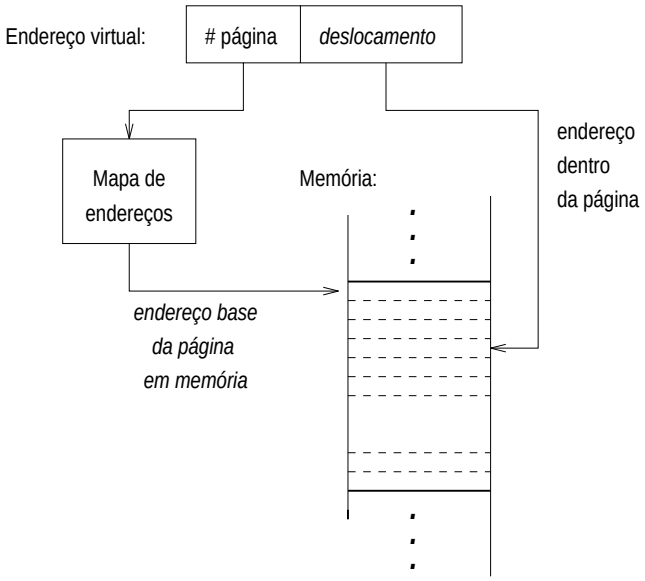
→ uso de *blocos* devido a características de acesso à memória secundária

- blocos de tamanho fixo: *paginação*
- blocos de tamanho variável: *segmentação*

2. política de alocação de espaço e de troca de blocos entre memórias primária e secundária
3. política de busca de blocos da memória secundária
   - por demanda
   - antecipada

Paginação

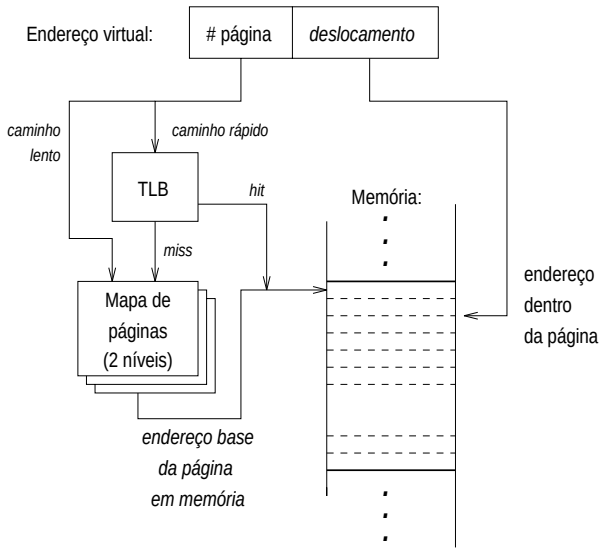
A informação é mantida para segmentos de tamanho fixo da memória primária e secundária. As *páginas* da memória secundária são transferidas para os *frames de páginas* da memória principal. Mapas de tradução de endereços são mantidos em memória pelo sistema, sendo que a informação associada a cada entrada é essencialmente um par *(página virtual, página física)*:



Um problema potencial é que esses mapas podem ocupar muito espaço de memória. Em um sistema com endereços de 32 bits, páginas de 4 Kbytes, são necessários 12 bits para manter informação do deslocamento na página, 20 bits para identificar página. Portanto, são necessárias 1M entradas na tabela.

Para lidar com este problema sem ocupar grande parte da memória principal, diversos níveis de paginação podem ser utilizados; em outras palavras, a própria tabela é paginada. Entretanto, neste caso aumenta-se número de acessos à memória secundária que podem ser necessários para acessar um endereço.

A solução para reduzir a quantidade de acessos em sistemas de diversos níveis de paginação é manter um *cache* privativo da tabela de páginas para as entradas referenciadas mais recentemente, usualmente denominado TLB *(Translation Lookaside Buffer)*:



Neste exemplo de mapeamento em dois níveis, o endereço da página virtual dividido em duas partes, uma correspondente a cada nível. O primeiro nível é um índice para a tabela de páginas; o segundo, um índice para memória primária.

No exemplo de endereço de 32 bits com 12 bits de deslocamento (páginas de 4 Kbytes), dividindo-se o número da página em duas partes de 10 bits cada, a tabela de cada nível tem 1K entradas — o primeiro nível sempre mantido em memória, o segundo nível pelo menos uma página da tabela em memória.

Outras informações que podem estar contidas na tabela de páginas, tais como se a entrada é válida (página ainda está presente em memória) e a proteção de acesso à página (leitura apenas, leitura e escrita, apenas execução).

Segmentação

Um dos objetivos na utilização de segmentação é melhorar o aspecto de localidade de referência em sistemas de memória virtual. Em um sistema paginado, os itens que são transferidos dentro de uma unidade de acesso (página) não estão necessariamente relacionados de forma lógica. Assim, itens que não serão utilizados podem estar sendo transferidos para a memória, com um possível desperdício de espaço em memória e de banda de passagem.

Com a segmentação, implementa-se um mecanismo para agrupar itens relacionados logicamente em unidades de acesso *(segmentos)* de tamanho variável. Essas unidades de acesso são mais próximas da visão do usuário (programas, dados) do que páginas. Uma outra vantagem é que segmentos de uso comum (por exemplo, o código de um programa do sistema operacional) podem ser compartilhados. Por outro lado, a memória virtual segmentada tem gerenciamento mais complexo. Cada entrada na tabela de segmentos tem que manter informação adicional sobre o comprimento do segmento, e a alocação de um segmento à memória deve gerenciar problema de fragmentação externa.

O espaço de endereçamento segmentado é bidimensional, com endereços sendo compostos por uma componente de identificação do segmento e outra de deslocamento dentro do segmento. Em um sistema paginado, quando um valor de deslocamento é incrementado além de seu valor máximo

um endereço na página seguinte é gerado; ou seja, o espaço de endereçamento é linear. No sistema segmentado, uma condição de tentativa de acesso fora de limites pode ser detectada.

Há duas possibilidades para suportar adequadamente o compartilhamento de segmentos:

1. O processo que for usar o segmento compartilhado tem que conhecer o endereço virtual alocado para o segmento — uma única tabela de segmentos é mantida;

2. O processo que for usar o segmento compartilhado pode acessá-lo através de seu índice particular — uma tabela de segmentos para cada processo. Neste caso, o endereço do início da tabela de segmentos é mantido em um *registrador base da tabela de segmentos*, uma informação associada ao contexto do processo.

Segmentação Paginada

Este é o esquema usualmente adotado, combinando as características de paginação e segmentação. A memória é organizada em segmentos, e cada segmento é particionado em páginas de tamanho fixo.

O endereço virtual é dividido em três partes: um número de segmento, um número da página e o deslocamento na página.

Determinação do tamanho da página

Dois fatores principais devem ser considerados: como reduzir o problema de fragmentação e como otimizar acesso à memória secundária.

Para otimizar o acesso à memória secundária, deve-se considerar que discos magnéticos são acessados em unidades de tamanho fixo (blocos, setores). Portanto, para reduzir número de acessos, o tamanho da página deve ser um múltiplo inteiro do tamanho de um bloco. Tipicamente, a dimensão de blocos varia de 512 a 8Kbytes.

Para reduzir a fragmentação, deve-se considerar o relacionamento entre a dimensão de segmentos e a dimensão de páginas. Sejam z e s os tamanhos de uma página e de um segmento em palavras, respectivamente. O número de páginas em um segmento é dado por

$$n(s,z) = \left\lceil \frac{s}{z} \right\rceil$$

O espaço desperdiçado por fragmentação interna (não utilizado na última página) é

$$I(s,z) = z \times n(s,z) - s$$

e o espaço ocupado pela tabela de páginas do segmento é

$$T(s,z) = c \times n(s,z)$$

onde c é uma constante expressando o número de palavras usado por entrada na tabela de páginas.

O espaço gasto em memória por segmento é

$$\begin{aligned} W &= I(s,z) + T(s,z) \\ &= (c+z) \times n(s,z) - s \end{aligned}$$

O valor médio para esta função é dado por

$$\begin{aligned} E[W] &= (c+z)E[n(s,z)] - E[s] \\ &= (c+z) \times \left(\frac{\bar{s}}{z} + \frac{1}{2}\right) - \bar{s} \end{aligned}$$

onde $\bar{s}$ é o tamanho médio de segmento.

Deseja-se minimizar este espaço gasto, portanto

$$\begin{aligned} \frac{dE[W]}{dz} &= \frac{1}{2} + \frac{\bar{s}}{z} - \frac{\bar{s}(c+z)}{z^2} \\ &= 0 \end{aligned}$$

de onde o tamanho ideal de página

$$z_o = \sqrt{2c\bar{s}}$$

Por exemplo, para $c = 2$

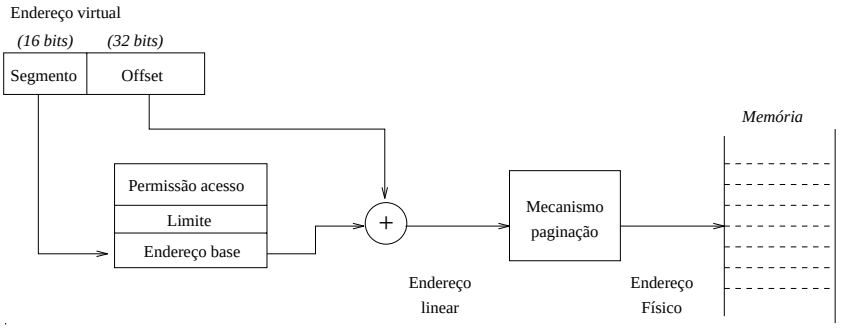
$\bar{s}$	z
16	8
256	32
4 K	128
16 K	256
64 K	512

Exemplo: Paginação e segmentação no i486

O Intel i486 oferece dois modos de operação, real e protegido. No modo real, o espaço de endereçamento está limitado até tamanho máximo de 1 Mbyte. No modo protegido, suporta-se memória virtual com paginação pura, segmentação pura, paginação segmentada, ou endereçamento físico.

A dimensão do número de segmento é 16 bits, com o tamanho do segmento podendo variar de 1 byte a 4 Gbytes. O tamanho máximo da memória física é 4 Gbytes, e o espaço total de endereçamento é 64 Tbytes. O endereço base de um segmento é adicionado ao deslocamento de 32 bits para gerar um **endereço linear**.

A utilização do mecanismo de paginação é opcional, podendo ser ativado ou desativado por software. A paginação opera sobre o endereço linear. A TLB oferece espaço para 32 entradas. O tamanho da página é fixo em 4 Kbytes. O diretório de tabela de páginas (primeiro nível) tem 1024 entradas (4 Kbytes), e a tabela de páginas (segundo nível) também tem 1024 entradas (4 Kbytes).



Se o mecanismo de paginação for desabilitado, trabalha-se então com um mecanismo de segmentação pura. O mecanismo de segmentação não pode ser desabilitado da mesma maneira, mas a paginação pura pode ser emulada ao se trabalhar com um único segmento de tamanho 4 Gbytes.

Política de troca de páginas

O objetivo desta política é definir a estratégia para seleção de página a ser substituída na memória de forma a minimizar o número de *page faults*.

Na ocorrência de um ausência de página referenciada em um instante de tempo t, uma das seguintes políticas poderia ser adotada para selecionar a página a ser substituída caso a memória já esteja completamente ocupada:

LRU *(Least recently used)*: substitui a página na memória cuja última referência é a mais antiga.

OPT *(Optimal)*: substitui a página na memória cuja a próxima referência está mais distante no tempo. Este algoritmo não é realizável, pois requer conhecimento do futuro; é usado apenas como algoritmo de referência para comparação *a posteriori*.

FIFO *(First-in, first-out)*: substitui a página mais antiga na memória.

LFU *(Least frequently used)*: substitui a página na memória cujo contador de referências tem o valor mais baixo.

FIFO Circular (ou Algoritmo do relógio): mantém páginas em uma fila circular. Se a página apontada para substituição (estratégia FIFO) foi referenciada recentemente (bit de referência setado), o bit de referência é zerado e o apontador move para a página seguinte. A primeira página encontrada com o bit de referência zerado é selecionada para substituição.

RAND: página selecionada para substituição é escolhida aleatoriamente.

Os primeiros sistemas implementando memória virtual analisaram estas diversas possibilidades, e concluiu-se que, em geral, LRU predizia o futuro satisfatoriamente.

Sob condições de uso intenso, estes sistemas entravam em períodos de instabilidade, caracterizando um fenômeno conhecido como *thrashing*. Quando uma página que foi substituída da memória tem de ser recuperada logo a seguir, gera-se um tráfego intenso de páginas entre memória principal e secundária. Eventualmente, pode-se perder mais tempo em gerenciamento da memória virtual do que na realização de trabalho útil.

A solução para o problema de *thrashing* utiliza o conceito de conjunto de trabalho *(working set)*. Este é o conjunto de páginas que compõe a área de trabalho de um processo. Se um processo tem este conjunto de páginas em memória, a taxa de ausência de páginas requisitadas será muito baixa. A limitação imposta nesta estratégia é que a combinação de todos os conjuntos de trabalhos de processos ativos não deve exceder a capacidade total da memória.

O cálculo do conjunto de trabalho deve ser realizado dinamicamente. O conjunto de trabalho $W(t, \Delta t)$ no instante de tempo t para janela de tempo Δt contém apenas as páginas que foram referenciadas nos últimos Δt segundos anteriores a t. Uma possível estratégia de gerenciamento envolve os seguintes passos:

1. Quando ocorre uma ausência de página requisitada, acrescente a nova página ao conjunto de trabalho
2. Do conjunto de páginas não referenciadas nos Δt segundos imediatamente anteriores à requisição, selecione uma página por LRU e a descarte. Caso todas as páginas do conjunto de trabalho tenham sido referenciadas no período, mantenha-as todas — o conjunto de trabalho cresce
3. Se duas ou mais páginas do conjunto de trabalho não foram referenciadas na janela referente ao conjunto de trabalho, selecione duas páginas por LRU e as descarte

A determinação do tamanho da janela Δt deve ocorrer experimentalmente.

Política de busca de páginas

A questão a ser respondida neste ponto é: quando que uma página deve ser trazida da memória secundária para a memória principal?

Na *paginação por demanda*, a transferência ocorre apenas quando a página é referenciada por um processo. Na política de *pré-paginação*, a página é carregada antes dela ser referenciada, com base em estimativas de acessos futuros a páginas.

Em sistemas de propósito geral, onde é difícil prever acessos futuros a páginas, a paginação por demanda é usualmente adotada. No entanto, dependendo das características do processo, pré-paginação pode reduzir sensivelmente a sobrecarga de transferência de páginas.

4.3.2 Interação entre *cache* e memória virtual

Cache pode operar com endereços físicos ou virtuais. Quando utilizando endereços físicos, o acesso ao *cache* é realizado apenas ao final da tradução do endereço (vrtual para físico) pela unidade gerenciadora de memória.

Com endereços virtuais, à custa de uma maior complexidade de projeto pode-se permitir a concorrência entre acesso ao *cache* e a tradução do endereço. Entretanto, introduz-se um problema potencial, conhecido como *aliasing*. Um *alias* caracteriza dados endereçados de forma lógica distinta (processos diferentes) que têm mesmo *tag* no diretório do *cache*.

A solução para evitar inconsistências de acesso ao *cache* por causa de *aliases* requer a inutilização de todo o todo o conteúdo do *cache* (*flushing*). Esta operação provoca queda sensível de desempenho. Por exemplo, implementações Unix em sistemas com *caches* virtuais realizam *flushing* a cada chaveamento de contexto ou operação de entrada ou saída.

Uma alternativa (adotada pela Sun a partir de seu modelo 3/200) é utilizar chaves de contexto, três bits que identificam a que contexto a linha do cache está associada. Isto permite *flushing* seletivo de um contexto. Dados compartilhados por diversos processos podem ser marcados como não passíveis de serem transferidos para o *cache*, evitando este problema.

4.3.3 Comparação entre *cache* e memória virtual

Apesar de representar dois níveis consecutivos da hierarquia de memória, há diferenças fundamentais entre o uso de *caches* e de memória virtual causadas pela discrepância entre os tempos de acesso entre os níveis envolvidos, consecutivos na hierarquia de memória.

O módulo de *cache* é entre 4 e 20 vezes mais rápido que o módulo de memória primária (tipicamente 5 vezes), enquanto que a memória primária é entre 1000 e 10000 vezes mais rápida que a memória secundária — e não se prevê redução deste *gap*, pois avanços em tecnologias de discos magnéticos melhoram velocidade de acesso a uma taxa menor do que as melhorias obtidas em memórias semicondutoras.

Quando o item buscado não está presente na memória rápida, a ação tomada depende desta diferença de velocidade de acesso. Para o *cache*, o processador espera enquanto a linha contendo o item é trazida da memória principal. No sistema de memória virtual, o tempo de busca ao próximo nível é extremamente longo, pois requer acesso a uma página inteira no disco. Duas estratégias são então possíveis:

1. Controle do processador passa a outras tarefas enquanto a ausência da página é tratada. Neste caso, o chaveamento de contexto necessário aumenta o tempo de espera para todas as tarefas.

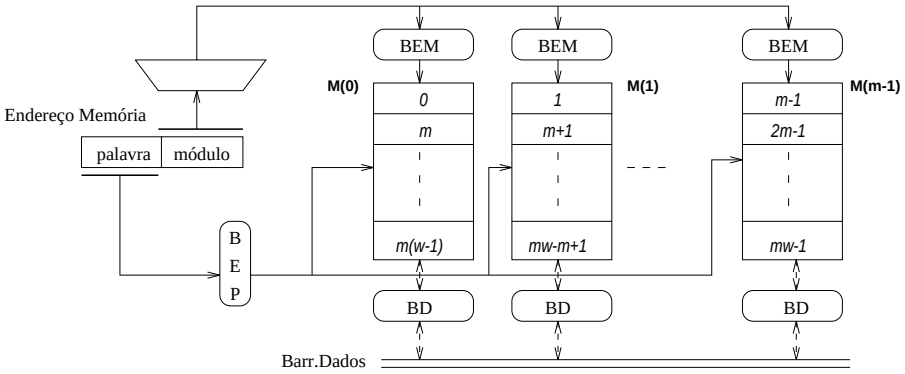
2. Processador espera página ser acessada. Esta solução é razoável quando ciclo de CPU é "barato", e pode ser necessária em sistemas de tempo real, onde o tempo de chaveamento de contexto não é um atraso tolerável.

4.4 Organização entrelaçada de memória

O objetivo desta estrutura é oferecer um método de organização física de módulos de memória principal de forma a melhorar a banda de passagem de memória. Com módulos independentes, cada módulo de memória poderia receber um endereço simultaneamente, permitindo o acesso paralelo ou o acesso em pipeline. Esta estrutura é extremamente adequada para situações onde há freqüentemente acessos a posições consecutivas de memória, como ocorre na transferência de linhas de *cache* e em processamento vetorial.

A alocação de um item a um módulo de memória pode seguir duas estratégias. Se os bits mais significativos de endereço determinam a qual módulo o item pertence, diz-se que a estratégia de entrelaçamento *(interleaving)* é de ordem alta. Caso os bits menos significativos sejam utilizados para este fim, a estratégia de entrelaçamento é de ordem baixa.

A forma mais adotada é o entrelaçamento de ordem baixa, onde endereços consecutivos são colocados em módulos de memória distintos. Assim, uma memória principal com $m = 2^a$ módulos de capacidade $w = 2^b$ palavras cada utiliza os a bits menos significativos do endereço para identificar o módulo, e os b bits mais significativos identificam o endereço da palavra dentro do módulo. O mesmo endereço de palavra é apresentado aos módulos simultaneamente:



BEM: buffer endereço módulo
BEP: buffer endereço palavra
BD: buffer de dados

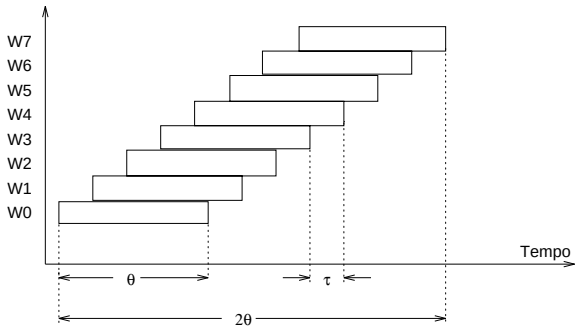
4.4.1 Acesso em pipeline

Uma vez que as palavras tenham sido selecionadas em cada módulo, é preciso entregá-las a quem as solicitou. Uma possibilidade seria ter um barramento com largura suficiente para transferir todas as palavras paralelamente. Esta solução de alto custo nem sempre é viável. Outra possibilidade é realizar a transferência em *pipeline*.

No acesso em *pipeline*, o ciclo de memória é subdividido em m *ciclos menores*, onde m é o grau de entrelaçamento; em outras palavras, a memória está dividida em m módulos. A relação entre o grau de entrelaçamento e o tempo de ciclo de memória θ define a duração do ciclo menor, τ:

$$\tau = \frac{\theta}{m}$$

O tempo de ciclo θ é o tempo para acessar uma palavra em um módulo; τ é o tempo necessário para o sistema de memória produzir uma palavra em um padrão de acesso a posições consecutivas. O tempo necessário para produzir a primeira palavra em um acesso a m posições consecutivas é θ. Como cada uma das palavras seguintes é entregue após um intervalo de tempo τ, o tempo total para acessar o bloco todo é 2θ. Por exemplo, para $m = 8$:



Assim, embora o tempo de acesso total a um único bloco de m palavras seja 2θ, o **tempo efetivo** de acesso aproxima-se de τ para uma sequência longa de acessos a posições consecutivas.

4.5 Subsistemas de entrada e saída

O subsistema de entrada e saída é a parte mais lenta da hierarquia de memória. Como já foi comentado, a diferença entre velocidade de acesso para memória principal e secundária não deverá ser reduzida através de fatores tecnológicos. Entretanto, a velocidade deste sistema de memória auxiliar é fator determinante no desempenho de sistemas de memória virtual e aplicações com uso intensivo de entrada e saída de dados — as chamadas aplicações limitadas por E/S *(I/O bound)*.

Em geral, componentes do subsistema de E/S são estruturados em duas grandes camadas, a interface de E/S e o dispositivo de E/S propriamente dito. Nem sempre a linha divisória entre estes componentes é clara. O conjunto de comandos associado ao diretamente ao dispositivo constitui seu *device driver*, responsável por tarefas tais como bufferização dos dados, conversão de formatos de representação e detecção e correção de erros.

4.5.1 Canais e processadores de E/S

Existem basicamente quatro níveis possíveis de interação entre a CPU e a execução da operação de E/S:

E/S programada: a CPU é responsável pelo controle da operação, usando principalmente a estratégia de espera ocupada (*busy-wait*);

E/S dirigida por interrupção: a CPU pode realizar processamento enquanto E/S ocorre, sendo notificada da ocorrência da operação, quando então assume controle para concluir a transferência ou tratamento do dado, se necessário.

Canais de dados de E/S: a CPU não intervem na operação de E/S — apenas requisita início da operação de transferência. O conceito de] canais de dados de E/S *(I/O channels)* oferece uma estratégia eficiente para transferência de blocos de dados, sendo utilizado para realizar transferências em DMA *(Direct Memory Access)*.

Processadores de E/S: este processador assume controle total da operação de E/S, sem intervenção da CPU. As transações de transferência de dados são controladas por *programas de E/S* que utilizam as instruções especiais oferecidas pelo processador de E/S.

Os canais de E/S, por sua vez, podem ser classificados como:

Canal seletor: manipula uma transação de E/S de cada vez. A inicialização da operação requer localização do início do bloco em memória, o comprimento do bloco a ser transferido e o endereço do dispositivo.

Canal multiplexador: pode manipular diversas transações concorrentemente, sendo composto por **subcanais**.

4.5.2 Características de discos magnéticos

Discos magnéticos são os componentes mais comuns em sistemas de memória secundária. De forma genérica, podem ser descritos como compostos por:

Mecanismo: é composto por componentes de gravação (superfícies magnéticas, cabeças de gravação e leitura) e por componentes de posicionamento (braços, motores).

Controlador: é composto por microprocessador (processador de entrada e saída), memória (buffer) e interfaces com barramento.

A superfície de gravação é organizada em trilhas, as quais são divididas em setores. Como a velocidade de rotação é fixa, o tempo total para transferência de uma trilha é constante. A transferência é realizada pela passagem da trilha sob a cabeça de gravação e leitura — em geral, há uma cabeça de gravação e leitura por superfície de gravação. Entretanto, a cabeça deve ser posicionada sobre setor para poder realizar a transferência de dados.

Assim, o tempo total para obter os dados de um setor do disco é uma composição de:

1. tempo de posicionamento *(seek)* da cabeça sobre a trilha;
2. atraso rotacional, para que o início do setor desejado passe sob a cabeça, e
3. tempo de transferência.

O tempo de posicionamento é afetado pelo mecanismo pelo qual o braço irá se posicionar sobre a trilha desejada. Para tanto, o braço executa um movimento de rotação sobre um *pivot*. Esta estratégia oferece características mecânicas melhores que os motores de movimento linear usados antigamente. O tempo de posicionamento, por sua vez, pode ser decomposto em

(a) tempo de aceleração

(b) tempo de movimento livre

(c) tempo de desaceleração

(d) tempo de ajuste

Os fatores que limitam a diminuição deste tempo de posicionamento incluem a potência do motor do *pivot* e a rigidez do braço (que determina sua capacidade de suportar aceleração). Com relação ao tempo de posicionamento, discos de diâmetro menor levam vantagem.

O atraso rotacional depende diretamente da velocidade de rotação do disco — usualmente entre 3600 (“padrão”) e 7200 rpm. A maior velocidade reduz atraso rotacional e melhora taxa de transferência de dados, porém aumenta consumo de potência e requer melhor mecânica.

A taxa de transferência também depende da velocidade de rotação, mas também da densidade de armazenamento. A densidade de gravação linear está atualmente da ordem de 50000 bits por polegada. Apesar de um disco ter diversas superfícies, cada uma delas com uma cabeça de leitura e gravação, apenas uma das cabeças está ativa em um dado instante — um único canal de dados pode ser oferecido de cada vez. Um sistema multicanal poderia ampliar a capacidade de transferência de dados, mas há dificuldades que precisariam ser contornadas para implementar tal sistema. Uma delas é o problema do alinhamento simultâneo das cabeças sobre as trilhas.

Taxas de transferência atuais estão na ordem de 5.3 Mbytes/s (trilhas mais afastadas do centro) a 3.1 Mbytes/s (trilhas mais próximas do centro).

4.5.3 Tecnologia de controladores de disco

As principais funções executadas por um controlador de disco incluem:

- acesso ao mecanismo,
- gerência do sistema dedicado de ajuste,
- transferência de dados entre disco e sistema, e
- controle do buffer do disco (se presente).

Em geral, controladores utilizam processadores dedicados, com *overhead* de processamento da ordem de 0.3 a 1 ms.

Buffers de disco tem capacidade típicas na ordem de 64 Kbytes a 1 Mbyte, usando dispositivos com tecnologia SRAM de porta dupla (alto custo).

As principais estratégias de acesso com buffer são:

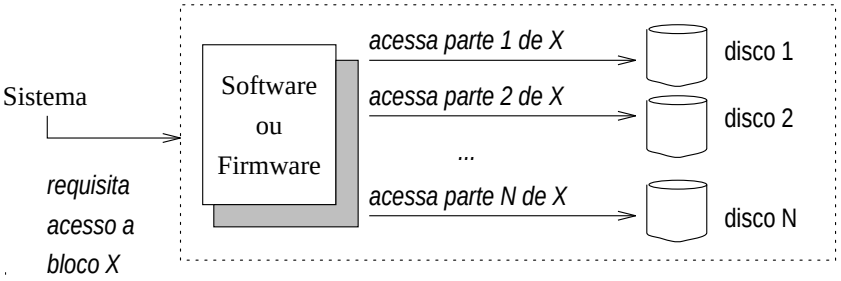
Leitura adiantada *(read-ahead)*: técnica adequada para fluxos de acesso seqüencial. Quando há mais de um fluxo sequencial previsto, buffer pode ser segmentado. O controlador pode ou não manter bloco no buffer uma vez que ele tenha sido acessado.

Leitura na chegada *(on-arrival)*: assim que a cabeça está posicionada sobre a trilha correta, inicia a transferência de dados para o buffer. O problema desta abordagem está no tamanho das trilhas, que tem aumentado muito — pode acontecer do buffer não ter capacidade para armazenar toda a trilha.

Escrita: apenas adotada se memória para o buffer é não volátil (possivelmente com bateria de backup) — caso contrário, há risco de perda dos dados. O uso do buffer para escrita reduz latência, evita acesso desnecessário ao mecanismo em caso de reescrita e permite escalonar acessos ao mecanismo.

4.5.4 Arranjos de discos

Um arranjo de discos é composto por um grupo de discos trabalhando como se fosse uma unidade lógica com melhores características de acesso e maior capacidade de armazenamento:



O problema associado a este trabalho em grupo é que a confiabilidade do sistema é reduzida. O MTTF *(Mean Time To Failure)* para um grupo de componentes trabalhando em conjunto é inversamente proporcional ao número de componentes. Por exemplo, o MTTF para um disco moderno é

de 200000 a 10^6 horas (cerca de 20 a 100 anos). Para um arranjo de discos, este tempo poderia ser reduzido a meses ou até mesmo a semanas.

No projeto de arranjos de discos, dois aspectos são considerados. O primeiro é como realizar a distribuição de dados de modo a melhorar o tempo de acesso. O outro aspecto é como utilizar redundância de dados para melhorar a confiabilidade do sistema.

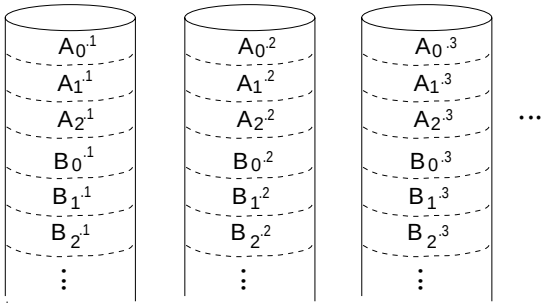
Distribuição de dados

A estratégia de distribuição de dados é a responsável pelo mapeamento de endereços lógicos, gerados pelo sistema,) para os endereços físicos, nos discos que compõem o arranjo.

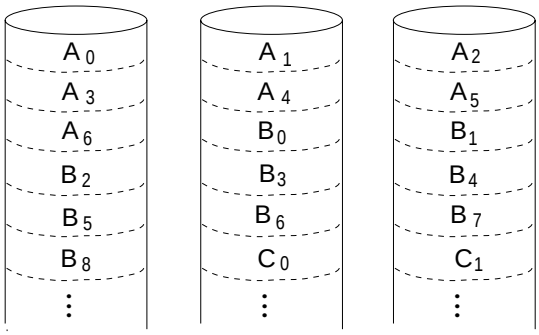
Na abordagem convencional, utilizada por exemplo em ambientes de redes locais de computadores, cada disco é tratado independentemente. Os usuários ou aplicações distribuem os dados explicitamente. Em geral, esta tarefa é delegada ao administrador do sistema, sendo difícil obter uma distribuição com acesso balanceado aos discos.

A estratégia de *striping* combina discos em um único espaço de endereçamento. As unidades lógicas de acesso consecutivas são distribuídas em *round-robin* entre os discos. Neste caso, o tamanho da unidade lógica de acesso determina o comportamento do sistema:

Striping com unidade pequena (menor que um bloco: bit, byte): todos os discos devem cooperar para atender a cada acesso. A taxa de transferência global com N discos com taxas individuais τ é aproximadamente $N \times \tau$. O tempo de posicionamento é em geral ou, na melhor das hipóteses, igual ao tempo para um disco.



Striping com unidade grande (pelo menos um bloco): discos trabalham cooperativamente para acessos a grandes volumes de dados, mas independentemente para acessos a blocos. O acesso ao primeiro bloco de uma série equivale à operação para um disco individual. Entretanto, acesso aos blocos seguintes pode ser agilizado durante o acesso ao primeiro bloco.



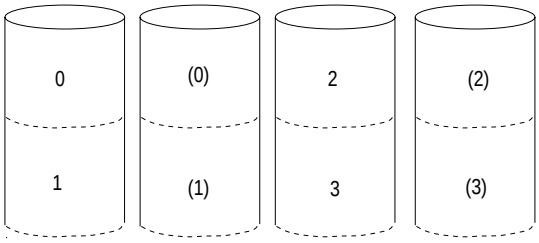
A escolha do tamanho da unidade é uma função principalmente das características do tipo de aplicação que se deseja atender.

Mecanismos de redundância

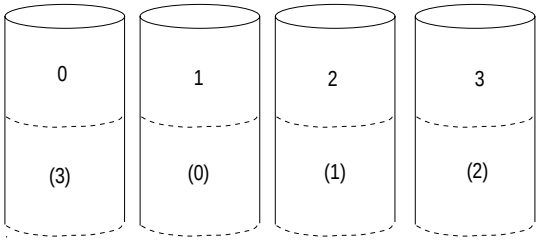
A abordagem tradicional de redundância é oferecer cópias *backup* em fitas magnéticas. Entretanto, em sistemas onde o acesso eficiente aos dados armazenados é crítico, deseja-se alguma forma de suportar falhas dinamicamente. As duas principais abordagens para tanto incluem a duplicação dos dados e a proteção por paridade.

Na estratégia de duplicação dos dados, múltiplas cópias dos dados são mantidas simultaneamente. Para manter N cópias de um disco, N discos adicionais são necessários. As caraterísticas de desempenho para escrita pode se degradar com esta estratégia, pois para cada operação vários discos deverão ser atualizados. No entanto, as características de leitura podem ser melhoradas, pois as cópias adicionais podem ser aproveitadas para agilizar acesso aos dados.

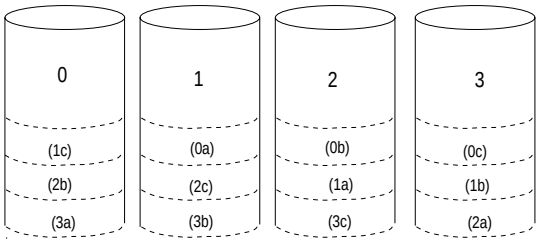
Os métodos básicos de duplicação incluem o espelhamento *(mirroring)* e o espalhamento *(declustering)*. No espelhamento, para cada disco de dados existe um disco imagem com conteúdo idêntico:



No espalhamento, os dados duplicados estão distribuídos entre todos os discos. No espalhamento encadeado, cada disco é dividido em duas seções, a primeira com dados e a segunda com suas cópias:



No espalhamento entrelaçado, as cópias dos dados são particionadas entre os discos que não contém os dados originais:



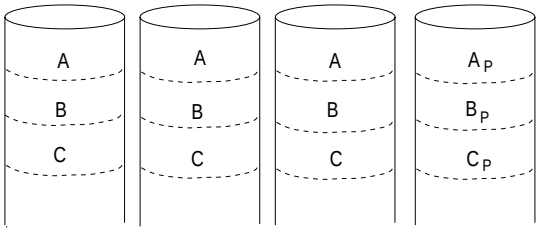
A estratégia de proteção por paridade usa os mesmos princípios adotados em códigos detectores e corretores de erros. Assim, um bit de paridade permite a recuperação de erro (em um bit) a partir da informação da paridade, dos bits corretos e do conhecimento da ocorrência de erro. Apesar de requerer um disco extra para manter a informação de paridade, é mais econômico que a estratégia de duplicação completa dos dados.

Há duas abordagens de manutenção da informação de paridade:

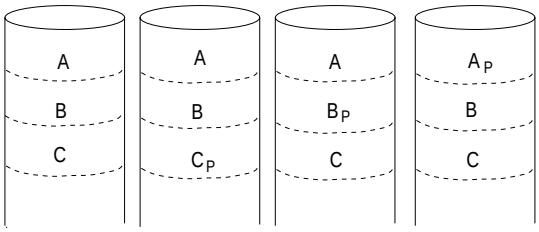
RMW *(Read-Modify-Write)*: nova informação de paridade é construída a partir dos valores novos e antigos dos dados sendo escritos e dos dados de paridade antigos;

RW *(Regenerate-Write)*: lê dados não sendo alterados e, juntamente com valores novos, gera nova informação de paridade.

A forma de armazenamento da informação de paridade também é um parâmetro de projeto. Uma abordagem utilizada é dedicar um disco exclusivamente para paridade:



A outra abordagem é distribuir a informação de paridade ao longo dos vários discos do arranjo:



RAID

RAID é a abreviatura para *Redundant Array of Inexpensive Disks.* É a forma de implementação mais comum de arranjos de discos.

Arranjos RAID são usualmente classificados em *níveis* de acordo com opções de organização e mecanismo de redundância adotados:

- RAID 0: striping com unidade grande, sem redundância
- RAID 1: endereçamento independente com duas cópias dos dados
- RAID 2: striping com unidade pequena, código corretor de erro (Hamming)
- RAID 3: striping com unidade pequena, disco de paridade
- RAID 4: striping com unidade grande, disco de paridade
- RAID 5: striping com unidade grande, paridade distribuída
- RAID 6: striping com unidade grande, dois códigos distintos de detecção de erros

Capítulo 5

Sistemas de Processamento Paralelo

Motivação. Processamento paralelo é utilizado para reduzir o tempo total de processamento de aplicações com grande demanda por recursos. Além de possibilitar a redução no tempo de execução, sistemas de processamento paralelo podem ser a única maneira de viabilizar computacionalmente as aplicações "grandes desafios" em áreas como medicina, biologia, física e engenharia.

Potencial de paralelismo. Considere um programa "grande", cuja execução em um processador de 100 MHz a uma taxa de execução de uma instrução por ciclo leve cerca de um dia. Se o programa tem cerca de um milhão de instruções distintas, mesmo um programa de tal dimensão apresenta um fator de repetição da ordem de 10^7. Considerando que mesmo uma pequena parte destas instruções repetidas pudesse ser executada paralelamente, um grande potencial de paralelismo está presente nestas aplicações intensivas.

5.1 Definições

Condições de paralelismo. Para que haja paralelismo, é preciso que haja independência entre as instruções executadas em paralelo. A forma essencial de independência é a independência de dados, estabelecida pelas condições de Bernstein. Formas associadas de independência incluem a independência de controle, segundo a qual dados de uma iteração não devem interferir com dados nas iterações sucessivas de um comando iterativo, e a independência de recursos, segundo a qual dois comandos não podem demandar a utilização simultânea de um recurso compartilhado.

Exemplo. A partir da análise de dependência de dados para as cinco instruções:

```
P1:    C = D * E
P2:    M = G + C
P3:    A = B + C
P4:    C = L + M
P5:    F = G / E
```

as instruções que podem ser realizadas em paralelo são P1 e P5, P2 e P3, P2 e P5, P3 e P5, P4 e P5. Consequentemente, P2, P3 e P5 podem ser executadas concorrentemente.

Relação de paralelismo. A relação de paralelismo apresentada acima é uma relação matemática, mas não uma relação de equivalência. Ela é comutativa e associativa, mas não é transitiva.

Granularidade. É uma medida da quantidade de computação envolvida em cada etapa do processamento paralelo. A granularidade varia de pequena (uma ou poucas instruções) a grande (programas). O grau de paralelismo é inversamente proporcional ao tamanho do grão de processamento.

Latência de comunicação. Como há a necessidade (em geral) de se comunicar os resultados entre grãos de processamento, o tempo dispendido nesta comunicação representa fator importante no desempenho global da aplicação. Quanto menor for o tamanho do grão de processamento, maior será o impacto da latência de comunicação no desemepenho. Um bom desempenho depende do balanceamento adequado entre as atividades de processamento e de comunicação.

Escalonamento. Outra atividade que não está diretamente ligada ao processamento mas que é necessária para a execução paralela é o escalonamento das atividades entre os nós de processamento. O impacto desta sobrecarga também está inversamente relacionada ao tamanho do grão de processamento.

5.2 Modelos de desempenho

Seja R o tempo dispendido no processamento de um grão (ou uma tarefa) de processamento e C o tempo de comunicação entre as tarefas. A razão R/C é um indicador da eficiência da computação paralela — valores altos indicam boa eficiência, enquanto valores baixos indicam alta sobrecarga de comunicação e, consequentemente, baixa eficiência.

Para desenvolver um modelo básico de desempenho, as seguintes considerações são assumidas: cada tarefa deve se comunicar com todas as demais tarefas, mas este tempo é nulo se duas tarefas estiverem alocadas a um mesmo processador; e não há sobreposição entre processamento e comunicação. O número total de tarefas é M e o número de processadores é N.

Considere inicialmente o caso de $N = 2$ processadores, onde um dos processadores recebe K tarefas. O tempo total de execução é

$$T_2 = R \times \max(K, M - K) + C \times K(M - K)$$

Há duas possíveis situações onde esta expressão é minimizada: quando $K = 0$ (ou equivalentemente, $K = M$); ou quando $K = M/2$. O limiar para definição de qual das duas situações corresponde ao mínimo ocorre quando

$$\frac{R}{C} = \frac{M}{2}$$

Para um número arbitrário de processadores, onde o processador $P_i,\ 1 \leq i \leq N$ recebe K_i tarefas, o tempo de execução será

$$T_N = R \times \max_i K_i + C \times \frac{1}{2} \sum_{i=1}^{N} K_i(M - K_i)$$

Similarmente, o limiar para que o mínimo ocorra para a execução de todas as tarefas em um processador ou para a execução balanceada das tarefas (cada processador com M/N tarefas) ocorre quando

$$\frac{R}{C} = \frac{M}{2}$$

5.3 Métricas de desempenho

Grau de paralelismo. Expressa o número de processadores que pode ser utilizado durante um intervalo de tempo para a execução de um programa. Um diagrama do GdP em função do tempo representa o **perfil de paralelismo** do programa. O **paralelismo médio** é expresso por

$$A = \frac{1}{\Delta T} \int_{\Delta T} GdP(t)\, dt$$

Fator de aceleração *speedup*. Representa o ganho no tempo de execução em paralelo. Se $T(1)$ é o tempo de execução em um processador e $T(n)$ é o tempo de execução da mesma tarefa com n processadores, então

$$S_n = \frac{T(1)}{T(n)}$$

O *speedup* assimptótico é definido como S_∞, e idealmente deve tender a A.

Eficiência. É uma medida normalizada do fator de aceleração,

$$E_n = \frac{S_n}{n} = \frac{T(1)}{nT(n)}$$

Redundância. Seja $O(1)$ o número de operações necessárias para executar o programa em um processador e $O(n)$ o número necessário para execução do mesmo programa em n processadores. A redundância é dada pela relação

$$R_n = \frac{O(n)}{O(1)}$$

Utilização. É uma indicação da fração de recursos que foi mantida ocupada durante a execução de um programa, expressa por

$$U_n = R_n \times E_n$$

Se adicionalmente assume-se que $O(1) = T(1)$, então a utilização pode ser expressa por

$$U_n = \frac{O(n)}{nT(n)}$$

Qualidade do paralelismo. É uma medida artificial que pretende capturar os principais aspectos relacionados ao desempenho de execução em sistemas paralelos, expressa por

$$Q_n = \frac{S_n \times E_n}{R_n}$$

Considerando que

$$\frac{1}{n} \leq E_n \leq U_n \leq 1$$

e que

$$1 \leq R_n \leq \frac{1}{E_n} \leq n$$

então observa-se que $Q_n \leq S_n$.

5.4 Classificação

O critério de classificação de Flynn, de 1966, é ainda o mais utilizado para categorizar sistemas paralelos. Por esse critério, computadores paralelos são analisados a partir do número de fluxos de instruções e de dados que são processados simultaneamente.

Máquinas SISD *(Single Instruction stream, Single Data stream)*. São as máquinas convencionais, que processam uma instrução operando sobre itens de dados simples.

Máquinas SIMD *(Single Instruction stream, Multiple Data stream)*. São os chamados sistemas de processamento matricial, com uma mesma instrução operando simultaneamente sobre dados distintos.

Máquinas MIMD *(Multiple Instruction stream, Multiple Data stream)*. Sistema paralelo de processamento com diversas instruções operando simultaneamente sobre dados distintos.

Máquinas MISD *(Multiple Instruction stream, Single Data stream)*. Não constituem uma categoria de máquinas paralelas no sentido usual da palavra, não havendo arquiteturas paralelas que se encaixem nesta classificação. Exemplos mais próximos de máquinas MISD incluem arquiteturas de processamento sistólico e computadores tolerantes a falhas.

5.5 Sistemas de processamento matricial

Em máquinas SIMD, um programa também é uma seqüência única de instruções armazenadas em memória. Cada instrução é decodificada em seu turno, indicado por um único registrador contador de programa, por uma única unidade de controle. A instrução decodificada é então executada atomicamente. Instruções escalares, que operam sobre um único item de dado, são executadas por uma ULA escalar, convencional. No entanto, algumas instruções (matriciais) operam sobre agregados de dados, sendo executadas simultaneamente por um arranjo de **elementos de processamento** (PE, de *Processing Element*).

5.5.1 Arquitetura básica

O programa SIMD é armazenado em uma **memória de controle**, que é também o repositório inicial para os dados. Os dados que serão processados pelos PEs devem ser distribuídos a diversos **elementos de memória** (ME).

A forma de conexão entre cada PE e cada ME caracteriza a arquitetura da máquina SIMD e o modo pelo qual os PEs trocam informação entre si. Se cada PE tem acesso exclusivamente a um ME, caracteriza-se um sistema SIMD de **memória local**. Neste caso, uma rede de interconexão interprocessadores suporta a troca de informação diretamente entre os PEs. Por outro lado, se todos os PEs estão conectados a todos os MEs através de uma rede de interconexão, então a troca de informação é realizada através do acesso a posições de memória compartilhadas, caracterizando assim um sistema SIMD de **memória global**.

5.5.2 Princípio de operação

Todos PEs operam sincronamente, iniciando e concluindo a instrução ao mesmo tempo. Cada PE tem seus registradores locais e pode executar operações lógicas e aritméticas sobre os itens de dados de um arranjo. Por exemplo, a instrução `ADD R1,X`, onde `X` é um vetor, faz com que o conteúdo de cada posição do vetor seja adicionada ao conteúdo de cada `R1` de cada PE.

Instruções de desvio são executadas pela unidade de controle, podendo estar associadas ao resultado de instruções escalares ou matriciais. No caso de condições associadas a instruções matriciais, são utilizados registradores de condições múltiplas (1 bit associado ao resultado de cada PE), e as condições envolvem combinações (todos, pelo menos um, nenhum) dos bits individuais.

A utilização de "máscaras" podem inibir a execução de uma instrução em PEs selecionadas. Para tanto, são fornecidas instruções especiais que permitem ajustar o conteúdo de um registrador de mascaramento.

5.5.3 Processadores *bit slice*

Uma estratégia de máquina SIMD é usar PEs cada vez mais simples e numerosas, como forma de ampliar o potencial de paralelismo usando elementos de processamento menos complexos. Levado ao extremo, este raciocínio leva aos computadores *bit slice*, onde cada PE opera sobre um único bit de dado. Tais máquinas adequam-se bem a aplicações onde bits são de fato a unidade de informação, como por exemplo em processamento de imagens. No entanto, máquinas *bit slice* também podem operar sobre valores escalares, uma vez que é possível agregar PEs para realizar uma operação em conjunto.

5.5.4 Exemplos

O **ILLIAC IV** *(Illinois Array Computer)* foi o primeiro sistema SIMD de ampla repercussão, sendo do final da década de 1960. O projeto inicial previa um sistema com até 256 PEs organizadas em quatro quadrantes de 64 PEs. Cada quadrante era organizado como uma matriz bidimensional de 8×8 PEs, numeradas de PE_0 a PE_{63}. A PE_i estava conectada diretamente às PEs PE_{i-1}, PE_{i+1}, PE_{i-8} e PE_{i+8} (operações em módulo 64). Comunicação com outras PEs envolviam PEs intermediárias, sendo que no pior caso seis PEs intermediárias seriam envolvidas.

O ILLIAC IV não apresentava uma memória de controle global — os MEs locais eram utilizados como a memória de programa, organizados como os módulos de um sistema de memória entrelaçado. Cada ME local consistia de 2048 palavras de 64 bits cada, com um barramento de 512 bits conectando as memórias à unidade de controle para permitir a transferência de até oito palavras simultaneamente. A unidade de controle apresentava ainda um buffer de instruções de 64 palavras (suficiente para 128 instruções de 32 bits) usando *prefetching*, um buffer de dados também de 64 palavras, uma ULA de 64 bits, um registrador PC e quatro registradores acumuladores. Os PEs podiam executar operações inteiras sem sinal sobre operandos de 8 ou 64 bits e operações em ponto flutuante com operandos de 32 ou 64 bits.

O ILLIAC IV foi efetivamente construído com um único quadrante pela Burroughs em 1972. Esta mesma empresa construiu posteriormente outra máquina SIMD, o Burroughs **Scientific Processor**. Esta máquina apresentava arquitetura SIMD com memória global, com uma estrutura de interconexão *crossbar* ligando 16 PEs a 17 MEs. O elemento de memória "adicional" permitia que os dados fossem arranjados nos MEs de modo a garantir os principais padrões de acesso aos dados sem conflitos.

O **GF-11** foi construído pela IBM no início da década de 1980, como um protótipo de pesquisa voltado para a realização de cálculos numéricos em cromodinâmica quântica. Era uma máquina SIMD de memória local, com 576 PEs de 32 bits, 20 MFlops, ligadas por uma rede de interconexão de três estágios (usando *crossbars* 24×24). Cada ME apresentava uma hierarquia de três níveis: registradores (256 palavras, 12.5ns), memória estática (16K palavras, 50ns) e memória dinâmica (512K, 200ns, dois bancos).

MPP *(Massively Parallel Processor)* é um exemplo de máquina *bit slice*, tendo sido construída pela Goodyear Aerospace no início da década de 1980. Consistia de um arranjo de 16384 PEs distribuídas em um arranjo 128×128. A CM-2 *Connection Machine 2*, da Thinking Machines, é outro exemplo de máquinas com PEs *bit slice* operando com grande número de processadores.

5.6 Máquinas MIMD

Em máquinas MIMD, cada unidade de processamento é um processador autônomo, com sua própria unidade de controle, contador de programa e unidade de execução. Como para sistemas SIMD, há duas arquiteturas básicas:

Multiprocessadores: máquinas MIMD de memória global compartilhada;

Multicomputadores: máquinas MIMD de memória local distribuída.

5.6.1 Redes de interconexão

As redes de interconexão representam um dos pontos críticos em sistemas MIMD. Ao contrário do que ocorre em máquinas SIMD, o padrão de comunicação em MIMD é assíncrono, requerendo redes de interconexão muito eficientes para tratar as diversas requisições independentes dos processadores.

Redes de interconexão são usualmente classificadas como dinâmicas ou estáticas. **Redes dinâmicas** são aquelas onde a conexão não é fixa, sendo determinada de acordo com a demanda de comunicação. Exemplos de redes dinâmicas incluem barramentos, *crossbars* e redes de múltiplos estágios.

Redes estáticas apresentam conexões fixas, que não podem ser alteradas durante a execução de um programa paralelo. Exemplos de redes estáticas incluem anéis, malhas e hipercubos.

Análise de redes dinâmicas

O desempenho de uma rede dinâmica é usualmente medido em termos de sua **banda de passagem** *(bandwidth)*, expressa em termos da quantidade de solicitações geradas pelas fontes (processadores) que foram aceitas pelos destinos (memórias ou processadores).

Para simplificar esta análise de desempenho, as seguintes assumpções são tomadas:

1. As requisições são geradas pelos processadores sincronamente, com uma probabilidade r;
2. As requisições são eventos aleatórios e independentes, com os destinos sendo uniformemente distribuídos;
3. As requisições não aceitas por problemas de contenção na rede de interconexão são simplesmente descartadas.

O **barramento** é o mecanismo de interconexão mais simples possível. Em um barramento, apenas um processador pode ter acesso à "rede" por vez. Para controlar o acesso ao barramento, mecanismos de arbitração devem ser utilizados. A banda de passagem para um barramento conectando p processadores é dada por

$$BW_{\text{bus}}(p) = 1 - (1 - r)^p$$

onde $1 - r$ é a probabilidade que um processador não esteja gerando uma requisição para algum módulo de memória, $(1 - r)^p$ é a probabilidade de que nenhum processador estar gerando uma requisição para algum módulo de memória e $1 - (1 - r)^p$ é a probabilidade de que pelo menos um processador esteja gerando uma solicitação de acesso a algum módulo de memória.

Um ***crossbar*** $p \times m$ é um arranjo de chaves com p entradas e m saídas, onde cada chave controla uma conexão completa de um dos barramentos de entrada com um dos barramentos de saída. Assim, $\min(p, m)$ conexões simultâneas podem ser efetivadas através de um *crossbar*. A banda de passagem para um *crossbar* $p \times m$ é dada por

$$BW_{\text{xbar}}(p, m) = m \times \left[1 - \left(1 - \frac{r}{m}\right)^p\right]$$

onde

r/m é a probabilidade de um processador gerar uma requisição específica para algum módulo de memória $M_j, 1 \leq j \leq m$.

$1 - r/m$ é a probabilidade de um processador não gerar uma requisição específica para algum módulo de memória M_j.

$(1 - r/m)^p$ é a probabilidade de nenhum processador gerar uma requisição específica para algum módulo de memória M_j.

$1 - (1 - r/m)^p$ é a probabilidade de pelo menos um processador gerar uma requisição específica para algum módulo de memória M_j, ou seja, de M_j estar aceitando alguma requisição de acesso.

A expressão para $BW_{\rm xbar}$ deriva-se da agregação das bandas de passagem dos módulos individuais de memória.

Para grandes valores de p e m,

$$BW_{\rm xbar} \rightarrow m \times \left(1 - e^{-rp/m}\right)$$

Crossbars oferecem o melhor desempenho de conexão, caracterizando uma rede não-bloqueante (todas as permutações possíveis são realizáveis). Entretanto, sua aplicabilidade é limitada por seu alto custo — para uma rede $n \times n$, $O(n^2)$ chaves são necessárias.

Redes de múltiplos estágios (MINs) utilizam diversos *crossbars* interligados como forma de reduzir o custo total da rede. Uma proposta pioneira neste sentido (rede Clos, 1953) tem origem na área de telefonia, onde através de uma rede de três estágios com vários *crossbars* configurava-se uma rede não bloqueante que, para redes grandes, apresentava um número de chaves menor que um *crossbar* de mesma dimensão.

As MINs utilizadas em multiprocessamento são em geral redes bloqueantes, compostas por células que são *crossbars* 2×2 com dois ou quatro estados possíveis de configuração, apenas. Os dois estados básicos são *through* (entrada 0 com saída 0, entrada 1 com saída 1) e *cross* (entrada 0 com saída 1, entrada 1 com saída 0). Os dois estados adicionais que podem ser suportados em algumas redes são *lower broadcast* (entrada 0 para saídas 0 e 1) e *upper broadcast* (entrada 1 para saídas 0 e 1). Uma rede com N entradas e N saídas requer $\log_2 N$ estágios com $N/2$ células deste tipo em cada estágio. Exemplos de MINs usadas em multiprocessamento incluem a Baseline e a Omega. Estas duas redes caracterizam-se pelo **auto-roteamento**, o fato de que o roteamento de mensagens é realizável considerando-se apenas o endereço de destino da mensagem.

O desempenho da MIN é obtido pela consideração de que cada célula é um *crossbar*. Sendo r_i a probabilidade de haver uma solicitação nas saídas das células do estágio i,

$$r_i = 1 - \left(1 - \frac{r_{i-1}}{2}\right)^2$$

com $r_0 = r$. A banda de passagem para a MIN $N \times N$ será portanto

$$BW_{\rm MIN}(N) = N \times \left[1 - \left(1 - \frac{r_{\log_2 N - 1}}{2}\right)^2\right]$$

Um dos fatos que limita o desempenho da MIN é o fato de que nem todas as permutações são realizáveis. Se s é o número total de células na MIN $N \times N$, então apenas 2^s das $N!$ possíveis permutações são realizáveis sem bloqueio, o que corresponde a uma fração

$$f_{\rm nb} = \frac{N^{N/2}}{N!}$$

5.6.2 Sincronização em multiprocessadores

O princípio da comunicação entre mutlprocessadores é o compartilhamento de variáveis globais. Para garantir a consistência no acesso a tais variáveis, mecanismos de sincronização são necessários.

Estes mecanismos de sincronização dependem de operações atômicas na forma Read-Modify-Write. Um exemplo típico é a instrução Test-and-Set, suportada por alguns processadores para fins de sincronização.

O problema com a sincronização desta forma é que ela implica em uma serialização no acesso à variável, podendo causar degradação sensível no desempenho de sistemas com um grande número de processadores. A alternativa é utilizar mecanismos de sincronização não-bloqueantes.

Fetch-and-Add é uma instrução que implementa esse tipo de mecanismo. Utilizado em redes combinantes, permite o acesso concorrente de vários processadores a valores consistentes de variáveis compartilhadas.

Uma rede combinante é essencialmente uma MIN cujos módulos de chaves incorporam lógica para

1. detectar pacotes com instrução Fetch-and-Add com mesmo endereço de destino,
2. realizar operação aritmética (soma), e
3. armazenar um dos valores envolvidos na operação.

Um grupo de instruções Fetch-and-Add executados concorrentemente acrescenta à variável global a soma correta dos incrementos de cada operação. Cada processador recebe como retorno da instrução um valor da variável global que equivale a alguma execução serial de todos os incrementos envolvidos, o que garante a consistência na operação concorrente.

Os problemas ainda existentes na utilização de redes combinantes são o custo e a maior latência de comunicação da rede.

5.6.3 Caches em multiprocessadores

A utilização de caches em multiprocessadores é motivada pela redução de tráfego pela rede de interconexão entre processadores e memória, uma vez que os caches são localizados junto aos processadores.

O problema ocorre quando caches armazenam cópias de dados compartilhados — como garantir que a cópia reflete de fato o valor mais recente da variável global, que pode ter sido modificado por outro processador? No caso de variáveis globais compartilhadas é preciso ter mecanismos adicionais de coerência de cache.

As políticas usuais de manutenção de coerência de cache baseiam-se ou na invalidação ou na atualização do valor armazenado em caches de processadores que não realizaram a modificação da variável compartilhada. Os mecanismos usuais para manutenção de caches em multiprocessadores incluem:

1. Manter um único cache junto à memória, compartilhado por todos os processadores — mantém alto tráfego pela rede, além de criar um ponto adicional de contenção no acesso ao cache.
2. Apenas variáveis compartilhadas não alteráveis podem ser movidas para o cache; as demais devem sempre ser manipuladas diretamente na memória global.

3. *Broadcast write*: cada operação de escrita em um cache é propagada para todos os outros caches, que avaliam a necessidade de invalidar (ou atualizar, se novo valor foi enviado) a linha correspondente no cache local.

4. *Snoop bus*: para sistemas com barramentos compartilhados, é possível ter um módulo de apoio ao cache que monitore todas as operações de escrita à memória no barramento e avalie a necessidade de invalidar o conteúdo do cache.

5. Manter diretórios globais identificando que variáveis estão presentes em quais caches. Quando ocorre uma operação de atualização de uma variável, o diretório é pesquisado para determinar quais caches devem ser notificados da modificação. Dependendo da estratégia de implementação pode ser *full* (cada entrada registra presença ou não da variável em todos os processadores do sistema) ou *limited* (cada entrada no diretório pode registrar presença de até n processadores, onde n é menor que o número total de processadores no sistema).

5.6.4 Multicomputadores

O compartilhamento de variáveis em multiprocessadores impõe restrições que limitam a escalabilidade daquele tipo de sistemas de processamento paralelo. Sistemas multiprocessadores dificilmente ultrapassam poucas dezenas de nós.

A alternativa é então utilizar sistemas sem memória compartilhada, ou sistemas multicomputadores. Em tais sistemas, a troca de informação entre processadores se dá por troca de mensagens. Sem as restrições impostas pelo compartilhamento de variáveis, tais sistemas podem apresentar um grande número de processadores — há sistemas com alguns milhares de nós.

Multicomputadores apresentam, no entanto, limitações. A primeira é a maior dificuldade de programação, pois os programadores devem explicitamente indicar trocas de mensagens através de pares de instruções Send e Receive. O balanceamento da carga de trabalho entre os nós também torna-se mais complexo, pois a migração de tarefas passa a apresentar um maior custo.

Outro aspecto envolvido em multicomputadores é a necessidade de manter cópias dos programas, assim como de um *kernel* do sistema operacional, em cada nó. A alocação dos programas aos nós pode representar uma sobrecarga sensível no tempo de execução, principalmente quando cada nó está executando um programa distinto. Este tempo pode ser reduzido com a adoção do modelo de programação SPMD *(Single Program, Multiple Data)* e o suporte a mecanismos de *broadcast* para a distribuição de programas aos nós.

Redes estáticas

Multicomputadores usualmente utilizam redes de interconexão interprocessadores estáticas, com caminhos fixos entre os nós. Cada conexão é um *link*, que pode ser unidirecional ou bidirecional. Alguns tipos de redes estáticas são apresentados na seqüência.

Rede completamente conectada. Cada um dos n nós tem conexão para todos os outros $n-1$ nós. Devido ao seu alto custo, é adequada apenas para redes com poucos nós.

Arranjos lineares. Cada nó tem conexão com dois vizinhos. Quando os nós extremos do arranjo também estão conectados entre si, o arranjo recebe a denominação de **anel**. Uma variante do anel é o **anel cordal**, onde uma conexão "saltando" alguns vizinhos está adicionalmente presente em cada nó.

Arranjos bidimensionais. Cada nó tem conexão com quatro vizinhos, configurando um arranjo em malha. Quando os extremos são conectados entre si, a rede é denominada um **toróide**. A vantagem deste tipo de rede é sua distribuição simples (adequada ao *layout* em VLSI) e boa escalabilidade, pois a quantidade de links em um nó não depende do tamanho total da rede.

Árvores. Cada nó tem conexão com m filhos e um pai, exceto pelo nó raiz, que só tem conexão com os filhos. Uma rede **estrela** é um caso particular de árvore com apenas um nível de profundidade. A principal limitação da organização em árvores é o alto tráfego de mensagens pelo nó raiz. Uma **árvore gorda** *(fat tree)* é uma alternativa de árvore onde o número de links entre os nós aumenta à medida que os nós vão se aproximando da raiz.

Hipercubos binários. Um hipercubo binário tem um número de nós $N = 2^n$, onde n é a dimensão do hipercubo. Cada nó no hipercubo tem uma representação binária de n bits e está conectado aos nós cuja identificação difere em apenas um bit. Hipercubos apresentam escalabilidade limitada, pois o número de *links* por nó depende do tamanho da rede. Uma variante do hipercubo que apresenta número fixo de *links* por nó é o **Cubo conectado em ciclos (CCC)**, onde cada nó do hipercubo é substituído por um anel com n nós.

Arranjos e hipercubos binários são casos particulares de **n-cubos k-ários**, uma estrutura de rede com n dimensões com k elementos em cada dimensão — k é a **raiz** da rede. O número total de nós neste sistema é $N = k^n$. Cada nó é identificado por uma n-upla $a_{n-1}a_{n-2}\cdots a_1a_0$, cujos elementos a_i são dígitos na base k. Cada nó está conectado aos nós vizinhos em cada dimensão, ou seja, aos nós cuja identificação só varia no dígito a_i para os valores $a_i + 1 \pmod k$ e $a_i - 1 \pmod k$, para i variando de 0 a $n - 1$.

Redes estáticas são usualmente avaliadas em termos de parâmetros do grafo que implementam, tais como

diâmetro: distância máxima entre dois nós;

distância média: a média das distâncias entre pares de nós;

número de links: a quantidade total de *links* na rede;

grau do nó: o número de *links* em cada nó;

largura de biseção: o número de *links* que devem ser interrompidos para dividir a rede em duas partes de igual tamanho.

Mecanismos de troca de mensagens

Em multicomputadores, a **mensagem** é a unidade lógica de troca de informação entre dois nós. A mensagem é usualmente dividida em pacotes de tamanho fixo, sendo que cada pacote contém informação adicional aos dados para fins de roteamento e recomposição da mensagem.

No roteamento **store-and-forward**, o pacote é a unidade física para a troca de mensagens. Cada nó deve oferecer um *buffer* com capacidade suficiente para armazenar um pacote. Cada nó intermediário no caminho entre o nó fonte e o nó destino armazena o pacote e verifica seu destino. Se não for endereçado para o nó local, o pacote é então re-encaminhado para a rede em direção ao seu destino.

Seja L o comprimento de um pacote em bits e W a velocidade de comunicação no canal, em bits por segundo. O tempo necessário para encaminhar um pacote de um nó fonte para um nó distante D links será

$$T_{\mathrm{SF}} = \frac{L}{W} \times D$$

Neste caso, a latência de comunicação é proporcional à distância entre os nós.

No roteamento **wormhole**, o pacote é subdividido em flits *(flow control digits)*, que podem ser um flit de cabeçalho e diversos flits de dados. A dimensão de um flit é determinada pelo tamanho da rede, sendo o número de bits necessários para endereçar um nó na rede. Neste tipo de roteamento, roteadores de hardware têm buffers para um flit apenas, mas mantém a conexão durante toda a transmissão da mensagem. Assim, flits são sempre transmitidos em ordem, em pipeline. OS processadores em cada nó não intervêm no roteamento, que é completamente decidido pelos chips roteadores.

Sejam L, W e D como acima, e seja F a dimensão de um flit em bits. A latência de comunicação para um pacote em wormhole será dado por

$$T_{\mathrm{WH}} = \frac{L}{W} + \frac{F}{W} \times D$$

Ou seja, dependendo dos valores relativos de L, F e D, a latência de comunicação é praticamente independente da distância.

Referências Bibliográficas

[1] Robert J. Baron and Lee Higbie. *Computer Architecture*. Addison-Wesley, 1992.

[2] Mark Brader. A chronology of digital computing machines (to 1952). `www.best.com/~wilson/faq/chrono.html`, April 1994.

[3] Martin Campbell-Kelly and William Aspray. *Computer: A History of the Information Machine*. Basic Books, 1996.

[4] Bob Carlson, Angela Burgess, and Christine Miller. Timeline of computer history. *IEEE Computer*, 29(10):25–110, October 1996. `www.computer.org/computer/timeline/`.

[5] James Goodman and Karen Miller. *A Programmer's View of Computer Architecture*. Saunders College, 1993.

[6] John P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, second edition, 1988.

[7] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[8] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[9] IEEE Computer Society. Events in the history of computing. `www.computer.org/50/history/`, 1996.

[10] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, third edition, 1993.

[11] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, third edition, 1990.