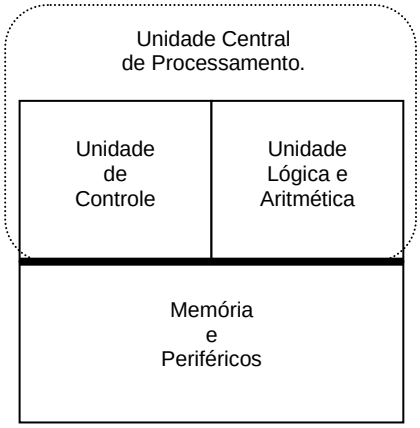
Modelos de arquitetura de computadores

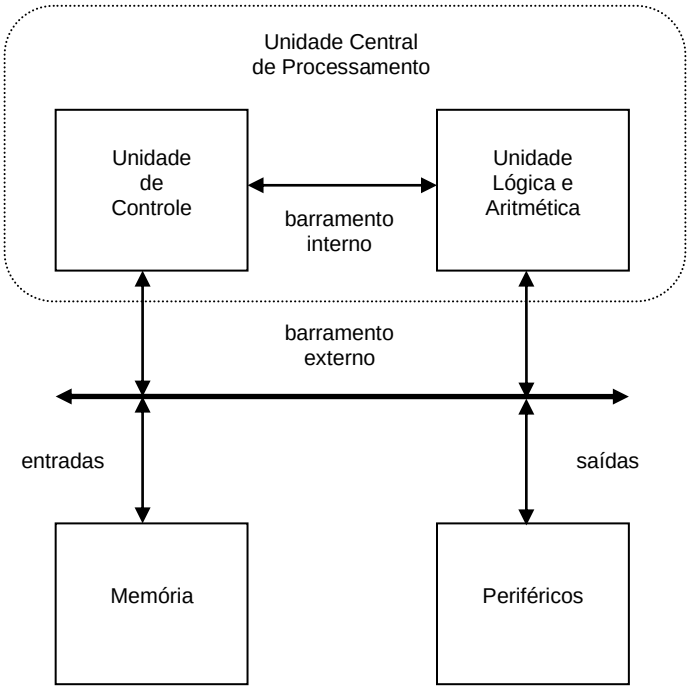
Um modelo simplificado de uma arquitetura genérica para computador procura agregar os seguintes componentes principais:

- Unidade Central de Processamento (UCP)
  - Unidade de Controle
  - Unidade Lógica e Aritmética
- Memória
- Periféricos



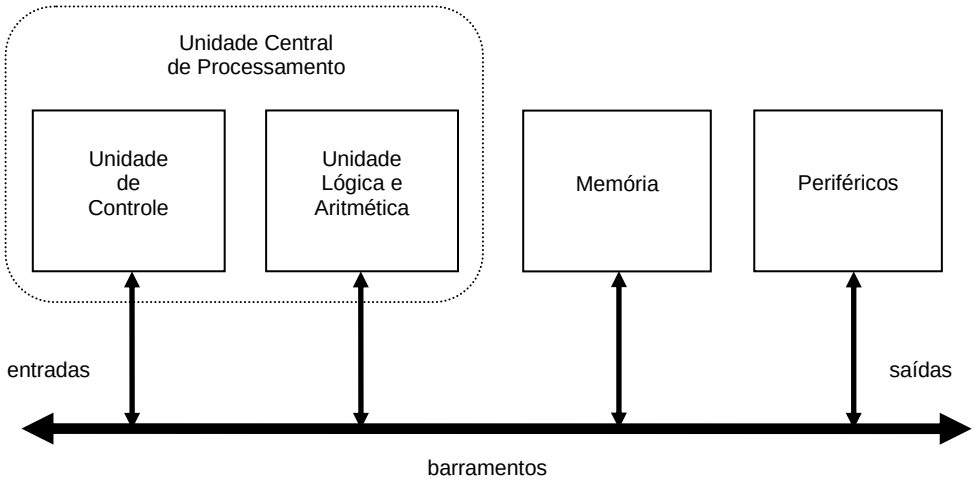
Organização de computador (1).

Esses componentes são conectados por vias ou barramentos, por onde passam as informações entre eles.



Organização de computador (2).

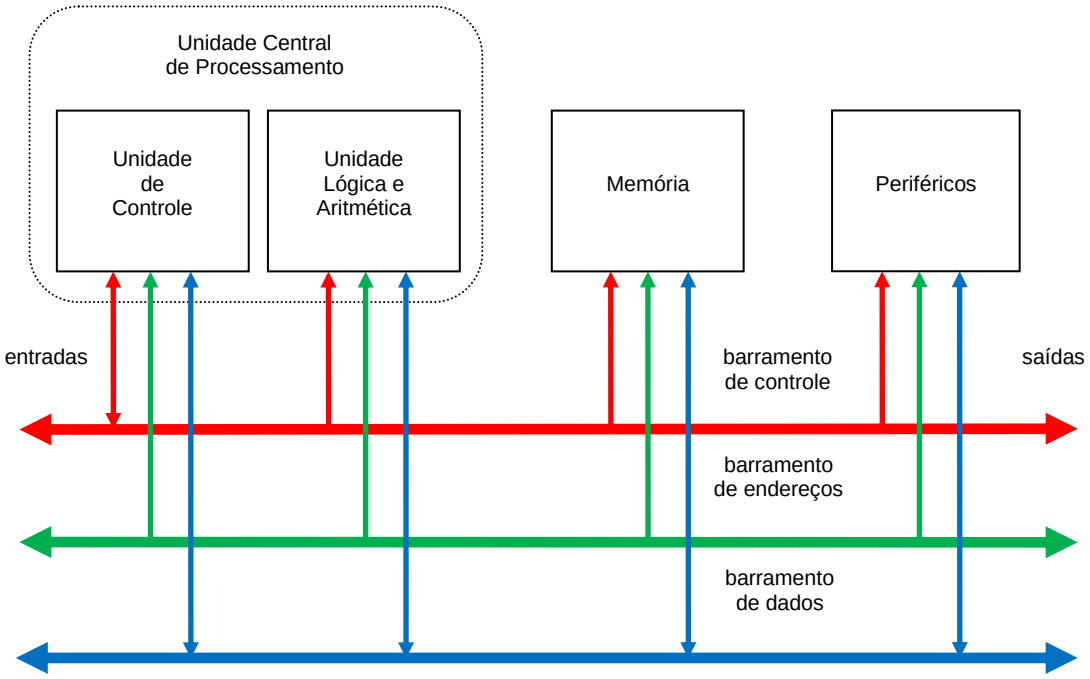
Embora a distinção entre barramento interno e externo exista, e a comunicação entre os componentes na Unidade Central de Processamento seja favorecida pela disponibilidade de um barramento interno exclusivo, a generalização das conexões por barramentos será empregada a seguir.



Os barramentos podem ser separados segundo especificidades em:

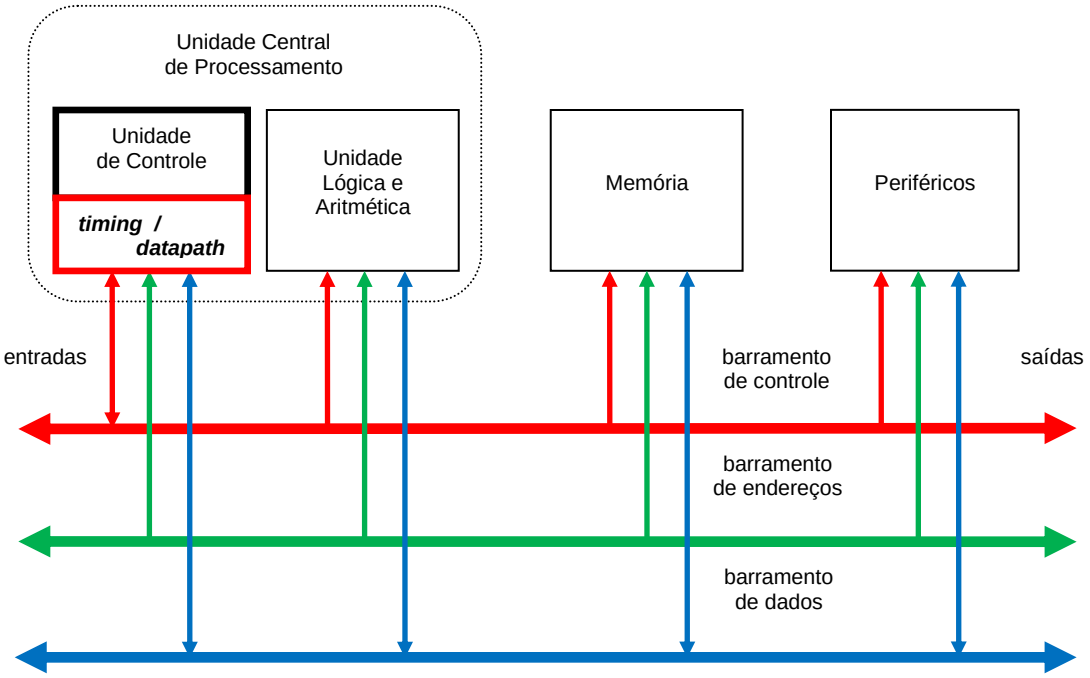
- de controle
- de endereços
- de dados

Organização de computador (3).



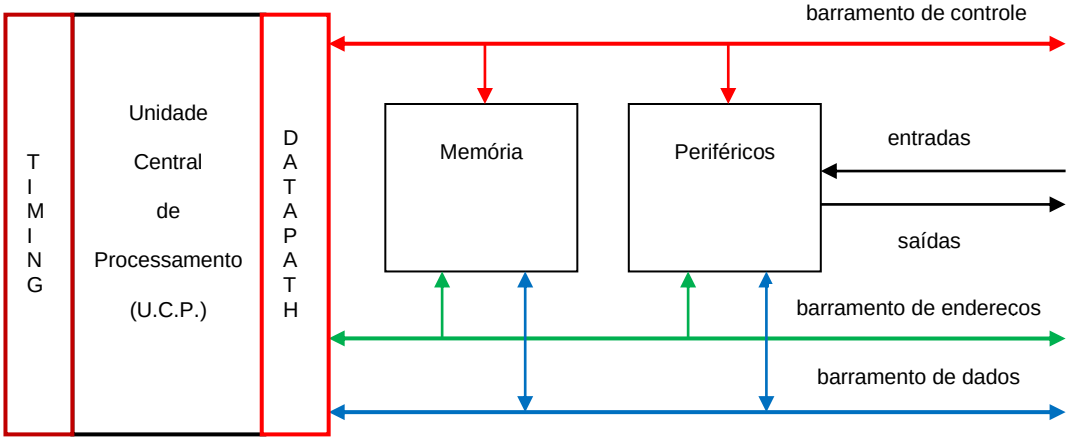
Organização de computador (4).

A Unidade de Controle conta com elementos adicionais para ajustar a sincronização das transferências (*timing*) entre componentes e para administrar os sinais para ativação/desativação de cada um deles (*datapath*).



Organização de computador (5).

O modelo a seguir dispõe os elementos para permitir futuras expansões.



Organização de computador (6).

Barramentos

Há três tipos básicos de barramentos: controle, endereços e dados.
O barramento de controle contém várias linhas de sinais:

- escrita e leitura na memória (R/W)
- escrita e leitura em porta de entrada e saída
- confirmação de transferência
- confirmação de interrrupção
- requisição e concessão de barramento
- temporização (*clock*)
- reinicialização (*reset*)

O barramento de endereços contém as linhas para

- origem e destino dos dados
- portas de entradas e de saídas

O barramento de dados contém as linhas por onde passarão os dados.

A largura do barramento de dados é fator importante para o desempenho. Por exemplo, se um barramento tiver largura de 08 bits e precisar transferir 16 bits, terá que fazer dois acessos à memória, um a cada ciclo.

Quando for necessário enviar dados, um módulo deverá obter o controle do barramento e transferir os dados.

Quando for necessário requisitar dados, um módulo deverá obter o controle do barramento, transferir uma requisição para outro módulo via linhas de endereço e de controle apropriadas e aguardar o envio dos dados.

Quanto maior o número de módulos conectados, maior o comprimento do barramento, bem como o atraso na propagação de sinais, o que definirá o tempo para se obter o controle.

Para se melhorar o desempenho há várias soluções que podem ser empregadas.

A primeira é ampliar o espaço ocupado com o aumento da largura, o que traz consequências sobre o projeto de circuitos.

A segunda é aumentar a velocidade de transferência, só que há limitações nas velocidades individuais de cada módulo que podem ser comprometidas ao se trabalhar em altas velocidades.

Uma terceira solução, mais complexa, é criar uma hierarquia de barramentos, com vários níveis de prioridades e de velocidades, que se comunicam através de interfaces, o que requer circuitos dedicados e maior detalhamento do controle, principalmente para arbitrar a concessão do barramento aos diferentes módulos. Isso poderá ser feito de forma centralizada ou distribuída.

Entretanto, se for possível abrir mão de uma função fixa, uma mesma via pode transferir endereços em certo momento, por exemplo no início de uma transferência, e transmitir dados a seguir. O custo do projeto poderá ser menor, pois poderá manter a largura, porém o tempo de transferência poderá ser maior.

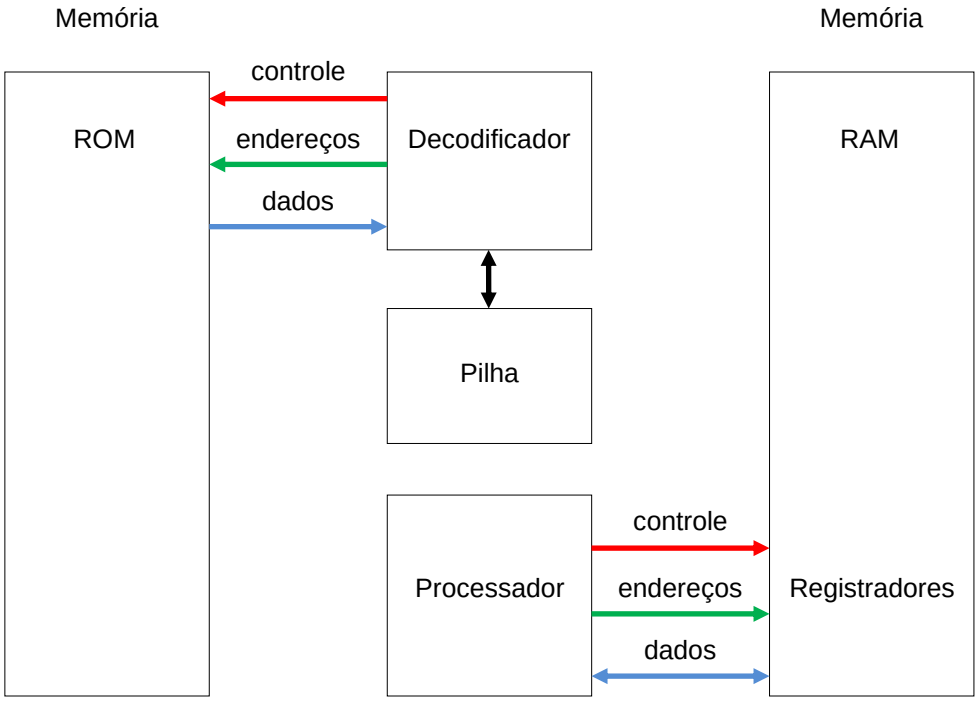
O controle da temporização poderá ser feito de forma síncrona ou assíncrona. A forma síncrona padroniza o número de ciclos, é mais barata e de fácil implementação, porém limita o uso de módulos mais rápidos. A forma assíncrona permite que o tempo das transferências possa ser variável, conforme as velocidades dos módulos envolvidos, contudo, é mais difícil de sem implementar e, por isso, nem sempre é uma boa opção a ser utilizada.

Os tipos de transferências também são fatores a serem considerados. Um tipo é transferir uma palavra por vez. Outro é transferir um bloco, o que pode ser mais eficiente, mas requer maior controle sobre as requisições (quantidade, tamanho), envios e comunicações entre módulos. Em sistemas multiprocessados é necessário assegurar o acesso a dados compartilhados, de tal forma que um processador deverá ser capaz de ler e alterar um valor na memória, por exemplo, sem ter que liberar o barramento, enquanto durar essas ações.

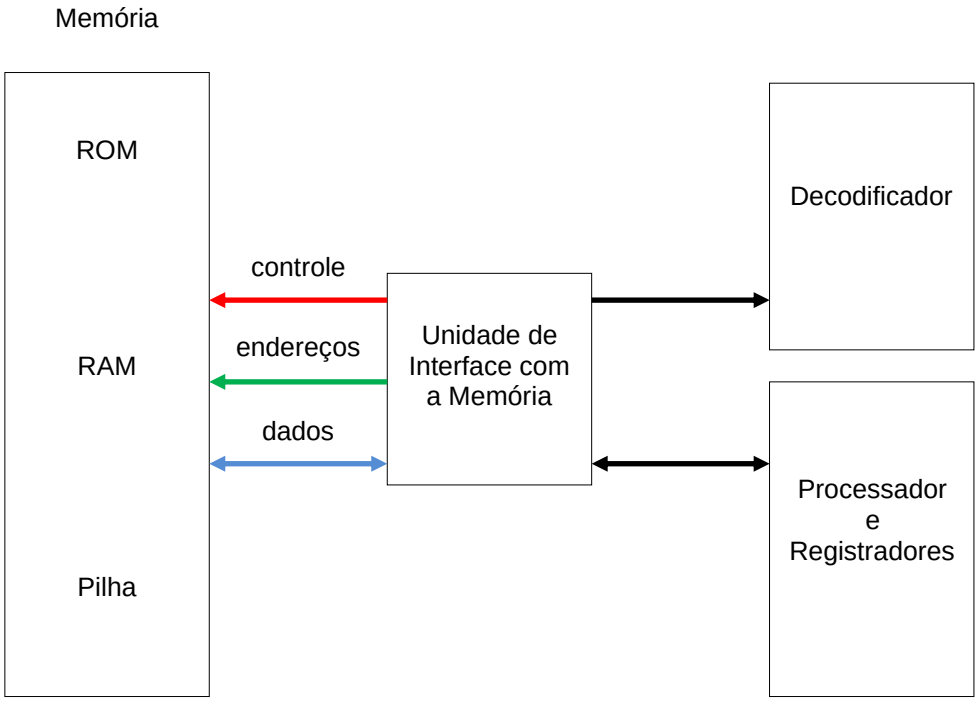
O projeto de barramentos tem influências diretas sobre modelos de arquiteturas.

Modelos de arquiteturas de computador sequencial

Arquitetura de Computador
(Harvard)



Arquitetura de Computador
(Princeton / von Neumann)



Comparação entre modelos de arquiteturas de computador sequencial

Harvard	**Princeton**
memória com código e dados com barramentos separados	memória com apenas um barramento
potencialmente mais eficiente (paralelismo) menor quantidade de ciclos de memória para se executar instruções	instruções e dados são tratados iguais uso de caches de instruções e dados pré-buscas de instruções e dados
modelo de projeto de processador com mais detalhes	modelo de projeto de processador mais simples
exemplo: Microchip PIC	exemplo: Motorola 68HC11 80x86/Pentium (exceto IO)

Comparação entre os principais tipos de arquitetura de conjunto de instruções (ISA's)
segundo o conjunto de instruções

CISC (*Complex Instruction Set Computer*)	**RISC** (*Reduced Instruction Set Computer*)
conjunto numeroso e variado de instruções	conjunto simples e pequeno de instruções
conjunto único de registradores (16)	múltiplos conjuntos de registradores (256)
fracamente paralelizado	altamente paralelizado (*pipeline*)
tempos de execução diferentes	quase o mesmo tempo de execução
instruções com múltiplos ciclos (mais lentas)	instruções com menos ciclos (mais rápidas)
instruções com tamanho variado	instruções com tamanho fixo
instruções típicas com 1 ou 2 operandos	instruções típicas com 3 operandos
complexidade no código	complexidade no compilador
acesso à memória por várias instruções	acesso à memória por poucas instruções
múltiplos modos de endereçamento	poucos modos de endereçamento
controle microprogramado	controle embutido no *hardware*
menos ortogonal (instruções específicas e pouco usadas)	mais ortogonal (mais potência e maior flexibilidade)

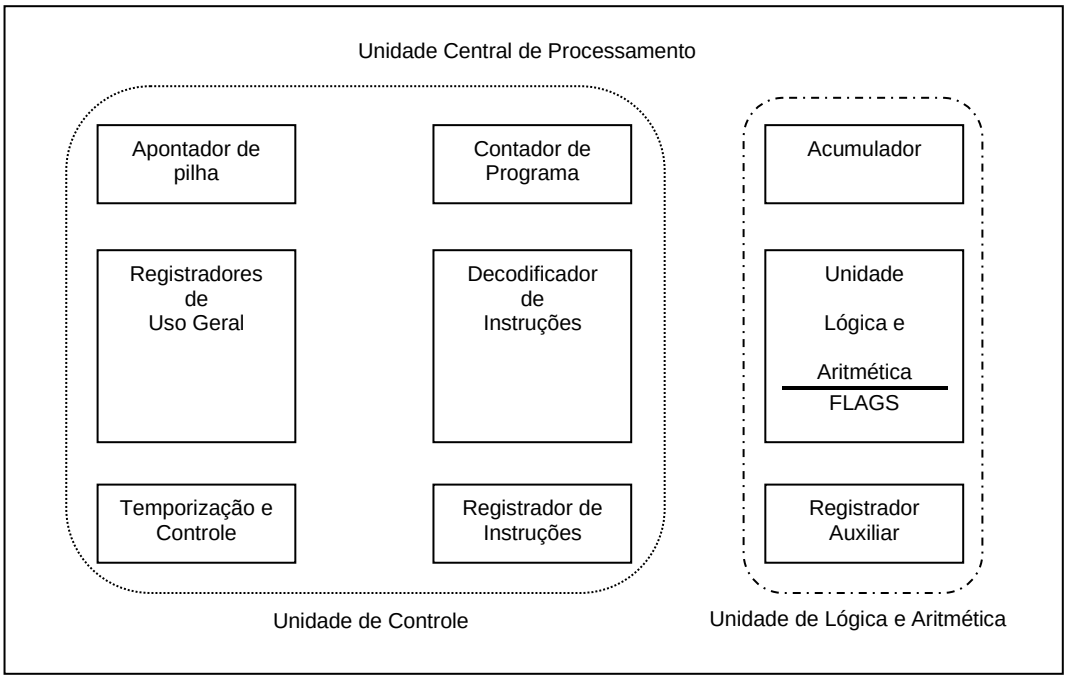
Combinações dos tipos acima constituem arquiteturas híbridas e servem:

- para possibilitar melhor compatibilidade entre *hardware* e *software*;
- para possibilitar a construção e uso de diferentes linguagens;
- para aumentar a capacidade de processamento.

Classificação de tipos de arquitetura de conjunto de instruções (ISA's)
segundo a complexidade

ISA	Detalhes
CISC	**Complex Instruction Set Computer** muitas instruções especializadas, algumas raramente utilizadas
RISC	**Reduced Instruction Set Computer** processador simplificado com implementação eficiente apenas do que for mais usado
VLIW	**VLIW - Very Long Instruction Word** projetada para explorar o paralelismo no nível de instruções (ILP) compiladores mais complexos, hardware simplificado para aproveitar o paralelismo conceito inventado por Josh Fisher e sua equipe na Universidade de Yale, no começo dos anos 1980 Princípios: - identificar paralelismo além de um bloco básico e desenvolver um compilador capaz de organizar a ordem de execução - o compilador e a arquitetura alvo deverão ser desenvolvidos em conjunto Exemplos de implementações: Intel i860 (1990) HP PA-RISC (1990) SOC (*System-On-a-Chip*) GPU (*Graphics Processing Unit*) da Nvidia e da AMD/ATI
EPIC	**Explicity Parallel Instruction Computing** termo cunhado em 1997 pela aliança entre HP e Intel para o desenvolvimento de uma arquitetura com maior independência entre o hardware e o software (compilador) Princípios: - explorar o paralelismo ao nível de instruções - eliminar a necessidade de circuitos complexos para organizar a ordem de execução de instruções em paralelo Exemplo: Intel Itanium (IA-64)
MISC	**Minimal Instruction Set Computer** conjunto bem pequeno de instruções básicas, geralmente orientadas para execução em pilha e com modo de endereçamento do tipo *load/store* Exemplos: Manchester Baby (University of Manchester, 1948) EDSAC (University of Cambridge, 1949) Mark1 (University of Manchester, 1949) EDVAC (Ballistic Research Laboratory, 1951) ORDVAC (University ou Illinois, 1951) IAS (Princeton University, 1952) MANIAC I (Los Alamos Scientific Laboratory, 1952) ILLIAC (University of Illinois, 1952)
OISC	**One Instruction Set Computer** máquina abstrata que tem apenas uma instrução, dispensando a necessidade de um código de operação em linguagem de máquina empregado no ensino de arquitetura de computadores e em modelos computacionais para pesquisas em computação estrutural Categorias máquinas para manipulação de bits máquinas ativadas por transporte (*TTA-Transport Triggered Architecture*) máquinas de Turing completas baseadas em aritmética Exemplos de instruções subtrair e desviar se menor ou igual a zero subtrair e desviar se diferente de zero subtrair e desviar se negativo subtrair e desviar se positivo

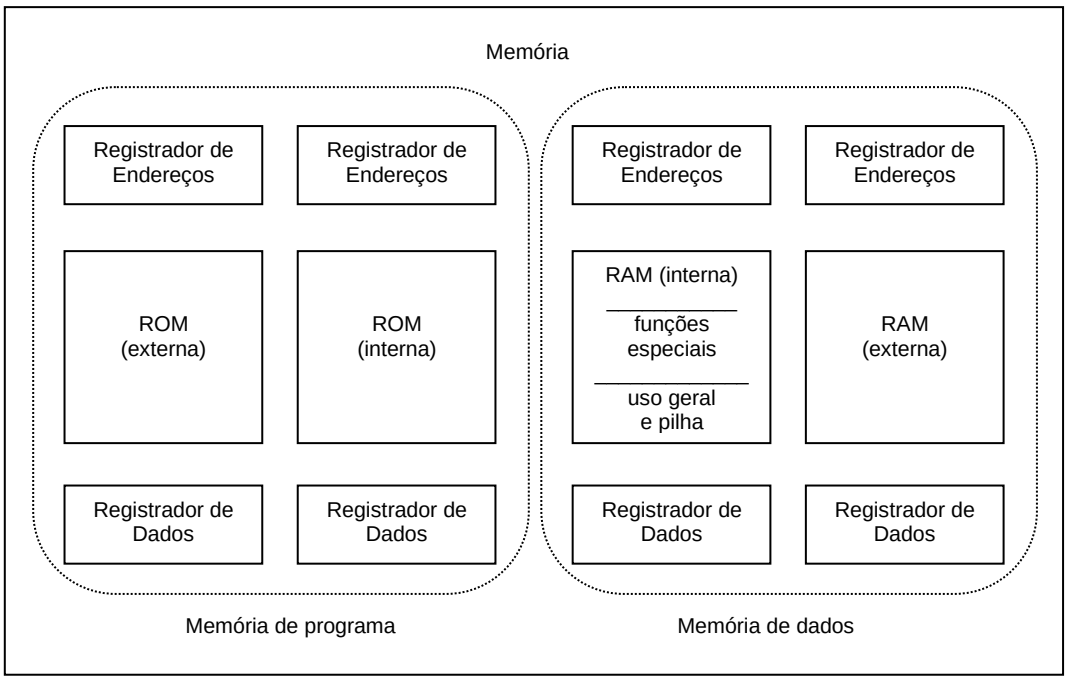
Os componentes principais para controle, operações e memória apresentam-se melhor agrupados por unidades funcionais.



Componentes principais para controle e operações

A unidade da Memória segundo sua constituição pode ser separada em:

- memória de programa (Read-Only Memory – ROM)
- memória de dados (Random-Access Memory – RAM)



Tipos de componentes de memória

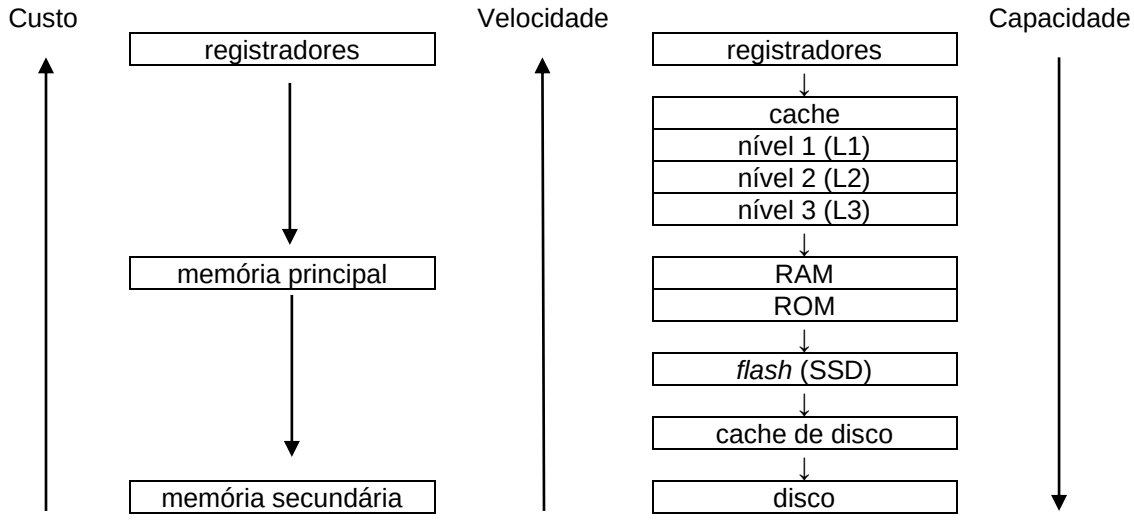
Tipos de memórias

Tipo de memória	Operação	Apagamento	Escrita	Volatilidade
RAM	leitura e escrita	elétrico/byte	elétrica	volátil (temporária)
ROM	leitura	(indisponível)	por máscara	não volátil (microprogramação)
PROM			elétrica	
EPROM	leitura (preferencial)	ultravioleta/*chip*		
EEPROM		elétrico/byte		
flash		elétrico/bloco		

As memórias do tipo RAM podem ser classificadas como dinâmicas ou estáticas.

RAM dinâmica	RAM estática
necessita refrescamento	dispensa refrescamento
construção simples	construção complexa
menor área/bit	maior área/bit
mais barata	mais cara
mais lenta	mais rápida
memória principal	memória cache

Hiearquia de memória



Princípio da localidade de referência

A hierarquia de memória poderá funcionar bem, se a grande maioria de dados/instruções puderem estar disponíveis nos níveis mais altos, enquanto as requisições que necessitem de acessos à memória em níveis mais baixos sejam bem reduzidas.

É tendência da execução sequencial de programas que haja reutilização de dados e instruções que tenham sido referenciados recentemente. Isso se deve, em boa parte, ao uso de estruturas de controle repetitivas, geralmente curtas e executadas numerosas vezes; ou, também, pelo uso de estruturas de dados como arranjos, listas e tabelas.

É possível prever, com razoável precisão, quais instruções ou dados serão usados em futuro próximo, e com isso aperfeiçoar o desempenho de certos programas.

Há dois aspectos comuns a serem considerados:

1. Localização temporal
   representa a (alta) probabilidade de que uma instrução executada recentemente seja realizada novamente em um curto espaço de tempo (repetições, por exemplo);

2. Localização espacial
   representa a (alta) probabilidade de que instruções próximas àquela executada recentemente sejam executadas também em futuro próximo.

Do primeiro aspecto decorre o objetivo de se manter instruções/dados acessados recentemente em local mais próximo para execução.

Do segundo aspecto decorre o objetivo de se mover blocos de instruções/dados para níveis mais altos na hierarquia de memória, na expectativa de que venham a ser usados proximamente.

Para favorecer a obtenção desses objetivos, memórias caches e memórias virtuais vêm sendo projetadas e utilizadas.

No caso das caches, quando um item (instrução/dado) é requerido, pela primeira vez, também será instalado na cache, para permanecer em disponibilidade, caso houver nova solicitação para uso. Por outro lado, para se reduzir cargas frequentes da memória principal (mais lenta), é conveniente carregar, em cache (mais rápida), um grupo de itens (bloco) que residam em endereços adjacentes.

O projeto de uma hierarquia de memória deve levar em consideração:

- a unidade mínima de informação presente (ou ausente) em cada nível (bloco, para memória cache; páginas, para memória virtual);
- as taxas de tentativas de acesso com sucesso (*hit rate*), no nível superior;
- as taxas de tentativas de acesso sem sucesso (*miss rate*), no nível superior (*miss*, para memória cache; *page fault*, para memória virtual);
- penalização no desempenho quando a tentativa não tiver sucesso (*miss penalty*), incluindo o tempo (*clocks*) necessário para substituir aquilo que estiver em falta.

O tempo médio de execução (*CPU_time*) dependerá do número de instruções, do período de *clock* (T_{clock}) e dos custos médios por instrução (CPI) para sua execução e para a espera por acesso aos dados:

$$CPU_{time} = \text{instruções} * T_{clock} * (CPI_{execução} + CPI_{memória})$$

e

$$CPI_{memória} = \text{acessos} / \text{instruções} * \textit{miss rate} * \textit{miss penalty}$$

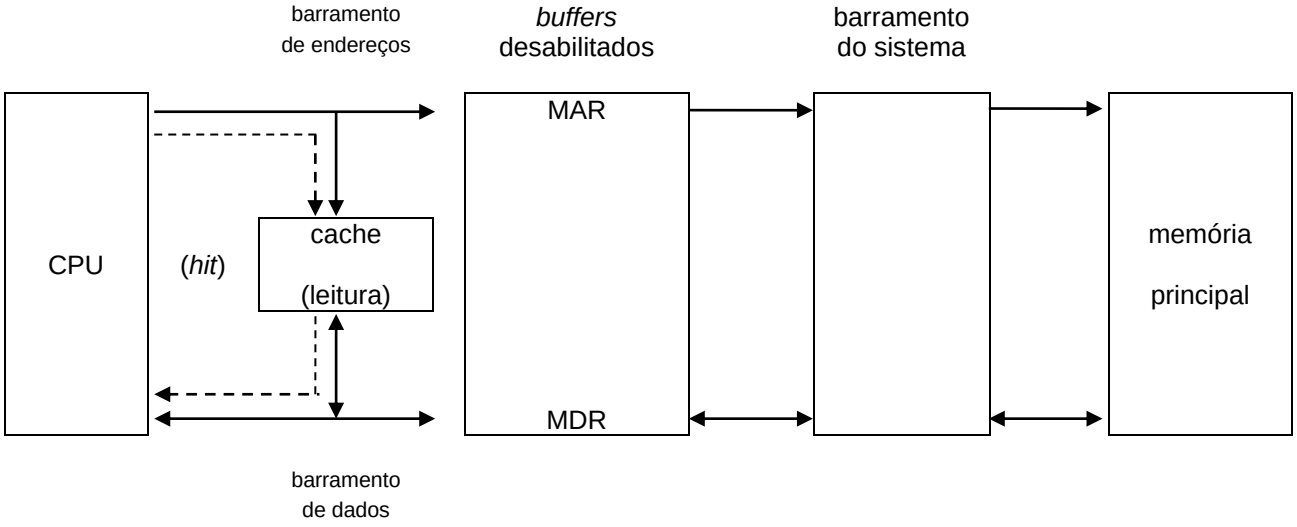
Melhor desempenho poderá resultar da minimização da *miss rate* e/ou *miss penalty*, bem como da seleção adequada das dimensões dos blocos, pelo melhor aproveitamento da largura de banda para acesso.

Memória cache

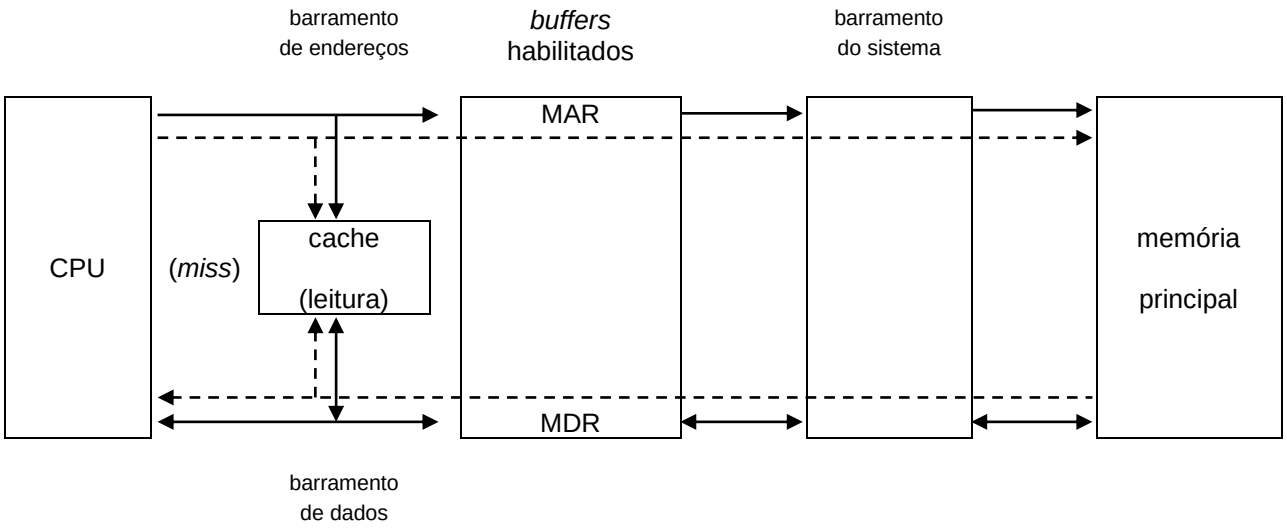
A memória cache serve como dispositivo auxiliar para melhorar a velocidade de acesso à informação.

Com capacidade menor que a da memória principal, guarda cópia de uma porção dessa.

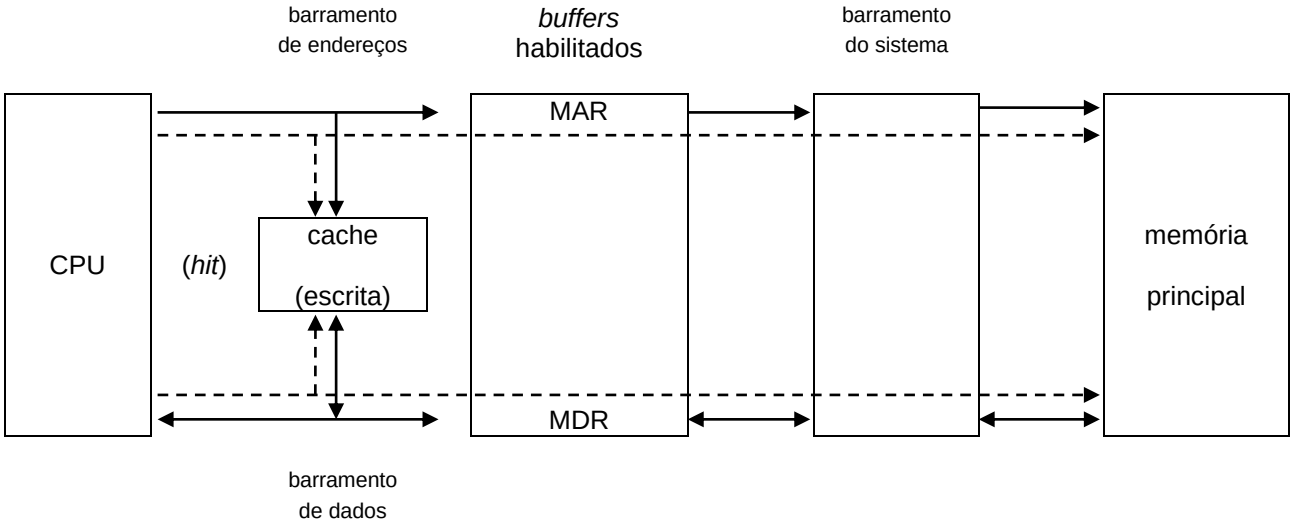
Se, durante uma operação de leitura, o conteúdo já estiver na mesma (*hit*), não é necessário buscá-lo na memória principal.



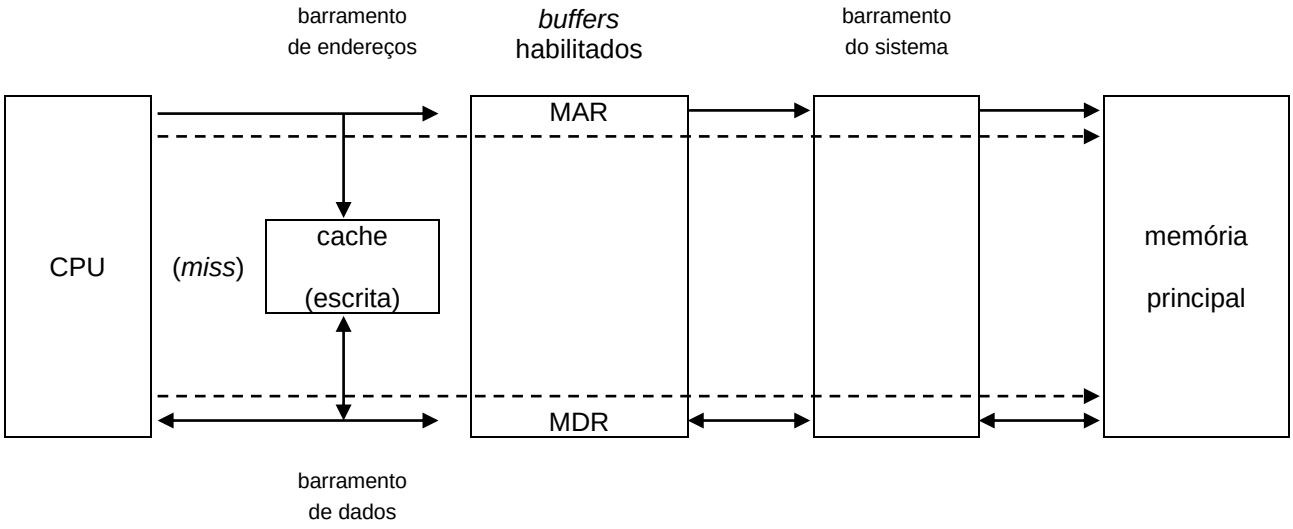
Caso contrário, ao ser necessário buscar um conteúdo não disponível (*miss*), ela também deverá ser atualizada.



Ao ser realizada uma operação de escrita, tanto o conteúdo da cache (*hit*) quanto da memória principal devem ser atualizados para manter coerência.



Caso o conteúdo não esteja disponível na cache (*miss*), a atualização será executada apenas na memória principal.



Tipos de memórias caches

Os tipos de memórias caches mais conhecidos são:

1. mapeamento direto
2. mapeamento associativo
3. mapeamento associativo por grupo (N-*way*)

Exemplos:

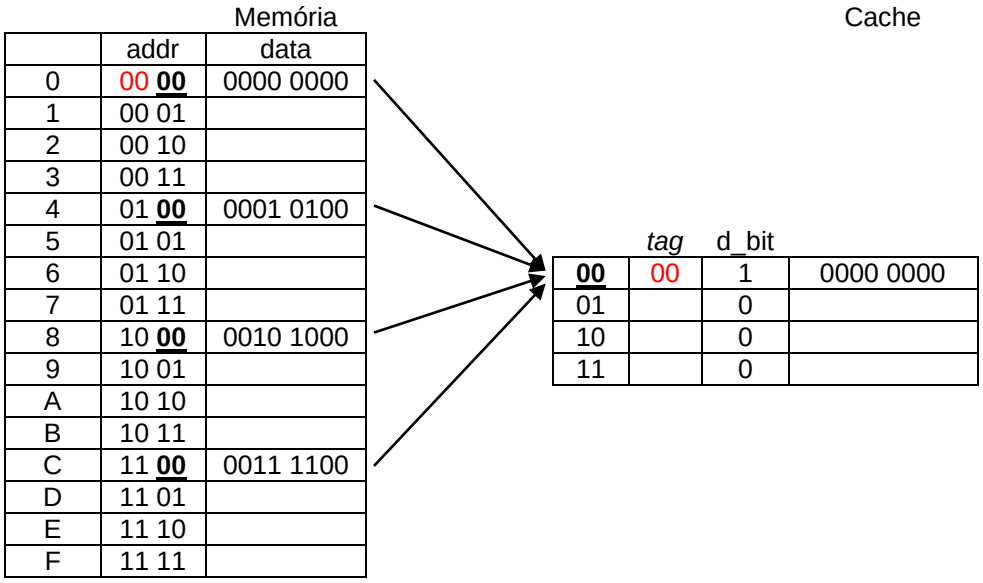
1. Mapeamento direto

Cada bloco só pode ser colocado em um lugar na cache.

A função de mapeamento é geralmente uma função modular (%).

Os bits de mais alta ordem podem ser recuperados pela etiqueta (*tag*).

O bit de validade (*dirty_bit*) serve para indicar se o dado é válido ou não.



Para ser instalada na cache, juntamente com a etiqueta (*tag*), na posição indicada pela função de mapeamento, verifica-se se a linha está livre; caso ocupada, será substituída.

A procura por uma palavra é feita comparando-se os bits de etiqueta (*tag*), se houver coincidência (*hit*), o bloco terá sido identificado.

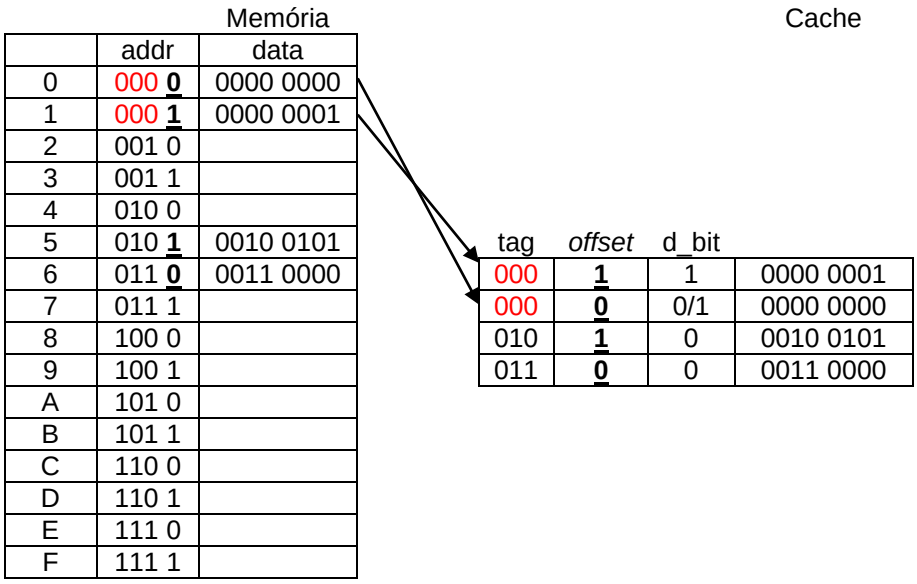
Tem por vantagens requerer um circuito simples para testar etiqueta, uma de cada vez, economizar energia (pois não testa todos os endereços), e ter políticas simples para instalação e substituição.

Tem por desvantagem uma pequena taxa de acessos com sucesso (*hit rate*); e toda vez que houver não encontrar (*miss*), promoverá substituição.

2. Mapeamento associativo

Cada bloco só pode ser colocado em um lugar na cache.

A função de mapeamento é geralmente uma função modular (%).



Para ser instalada na cache, juntamente com a etiqueta (*tag*) e o deslocamento (*offset*), na posição indicada pela função de mapeamento, verifica-se se a linha está livre; caso ocupada, outra posição será usada. Se todas estiverem ocupadas, se aplicará a política de substituição.

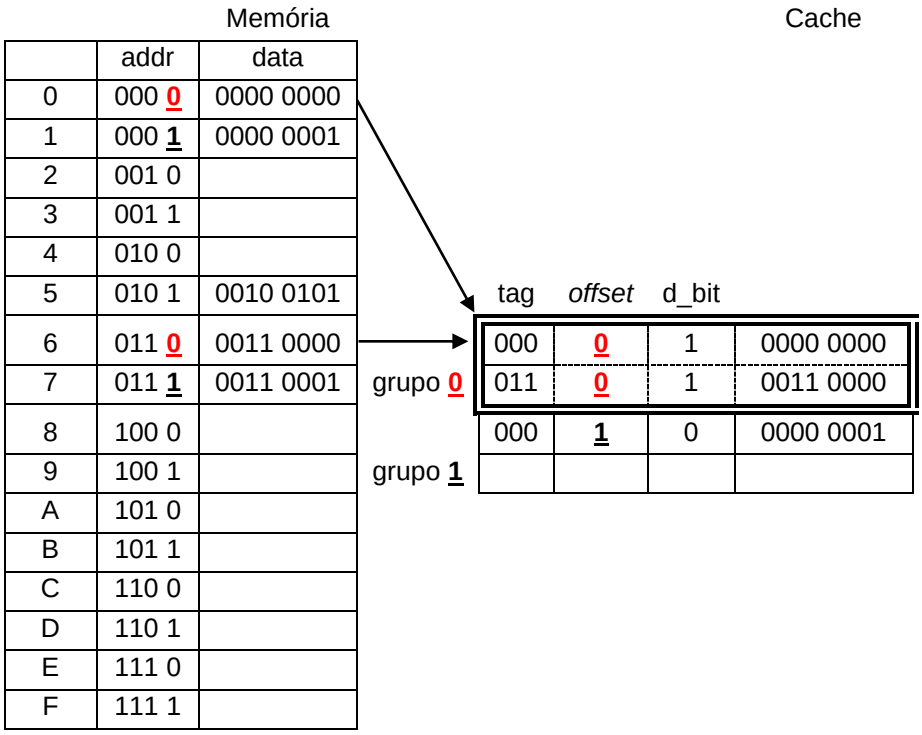
A procura por uma palavra é feita comparando-se todos os bits de etiqueta (*tag*), em paralelo, se houver coincidência (*hit*), o bloco terá sido identificado pelo uso do deslocamento.

Tem por vantagens oferecer flexibilidade para instalação em qualquer posição e, portanto, permitir o uso integral da cache; ter taxa de acessos com sucesso melhor; e permitir que a política para substituição possa empregar vários algoritmos.

Tem por desvantagens o custo, maiores gastos de tempo e energia para localizar um bloco.

3. Mapeamento associativo por grupos (N-*way*)

Solução de compromisso entre as anteriores, buscando unir vantagens de ambos.



Para ser instalada na cache, juntamente com a etiqueta (*tag*) e o deslocamento (*offset*), no grupo indicado pela função de mapeamento; verifica-se se a linha está livre; caso ocupada, outra posição será usada. Se todas estiverem ocupadas, se aplicará a política de substituição.

A procura por uma palavra é feita comparando-se todos os bits de etiqueta (*tag*), em paralelo, apenas para os elementos pertencentes ao grupo, se houver coincidência (*hit*), o bloco terá sido identificado pelo uso do deslocamento.

Tem por vantagens a combinação de características das outras formas de mapeamento, além de permitir que a política para substituição possa empregar vários algoritmos.

Tem por desvantagens a possibilidade de não usar todas as posições disponíveis e promover substituição quando houver falta (*miss*).

Algoritmos para substituição

Dentre os algoritmos para substituição destacam-se:

1. LRU (*Least Recently Used*)
   - a linha há mais tempo utilizada será a escolhida
2. LRR (*Least Recently Replaced*)
   - a linha trocada há mais tempo (FIFO) será a escolhida
3. LFU (*Least Frequently Used*)
   - a linha menos referenciada será a escolhida
4. Random
   - uma linha qualquer será escolhida aleatoriamente.

Técnicas de escrita de dados em cache

Técnicas de *Write Hit*

1. *Write-Back Cache*

Essa esta técnica escreve dados diretamente em cache, cabendo ao sistema, posteriormente, a atualizar a memória principal. Fica-se livre mais rapidamente para executar outras operações, embora a latência do controlador possa trazer problemas de consistência de dados na memória principal, como em sistemas multiprocessados com memória compartilhada, que deverão ser tratados por protocolos de consistência de cache.

Vantagens
- a escrita ocorre conforme a velocidade da cache;
- múltiplas escritas em um endereço requerem apenas uma escrita na memória;
- menor consumo de largura de banda.

Desvantagens
- mais difícil para implementar;
- problemas de consistência entre os dados existentes no cache e na memória;
- leituras de blocos em cache podem resultar em escritas de blocos inválidos (*dirty*) na memoria.

2. *Write-Through Cache*

Ao escrever em uma região de memória, que está contida em cache, escreve-se tanto na linha específica de cache como na região memória correspondente, ao mesmo tempo.

Vantagens
- mais fácil de implementar;
- uma falta (*miss*) nunca resulta em escritas na memória;
- a memória tem sempre a informação mais recente (sincronizada).

Desvantagens
- a escrita é lenta;
- cada escrita necessita de um acesso à memória;
- maior consumo de largura de banda da memória.

Técnicas de *Write Miss*

1. *Write Allocate*

O bloco de endereços é carregado, seguido de uma ação de *write hit*.
Frequentemente usado em caches com *Write Back*.

2. *No Write Allocate*

O bloco de endereços é diretamente modificado na memória, não é carregado em cache. Frequentemente usado em caches com *Write Through*.

Memória virtual

Memória virtual é uma técnica que busca combinar recursos de *hardware* e *software* para usar a memória secundária como cache para armazenamento secundário. Com isso é possível o compartilhamento seguro e eficiente da memória entre vários programas, e superar limites de memória principal.

Três funções básicas são oferecidas:

1. expansão
   permitir a utilização de memória maior que a fisicamente disponível;

2. realocação
   proporcionar um espaço de endereçamento próprio para cada processo (aplicação);

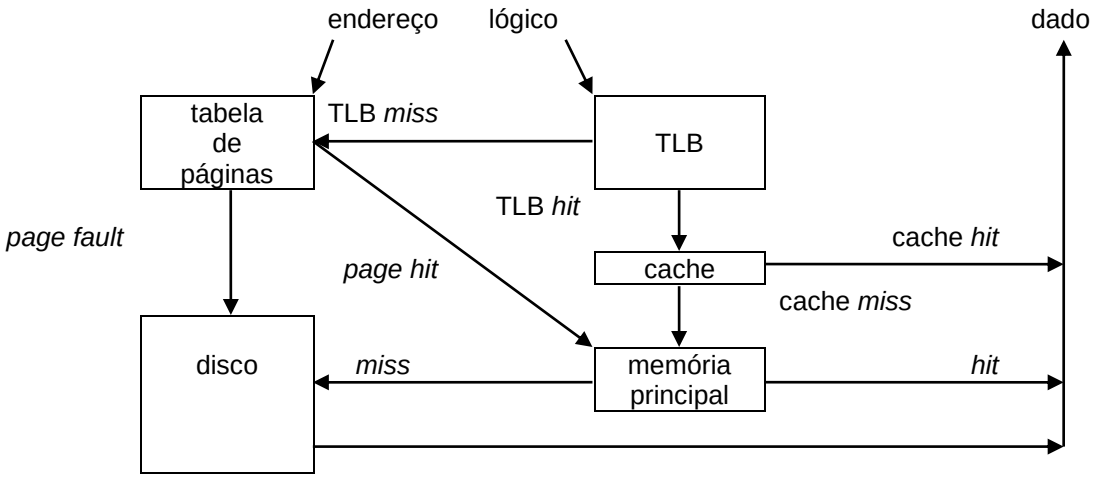
3. proteção
   oferecer limites para utilização apenas do espaço concedido e evitar invasões de outros.

Os principais mecanismos empregados para implementá-la são

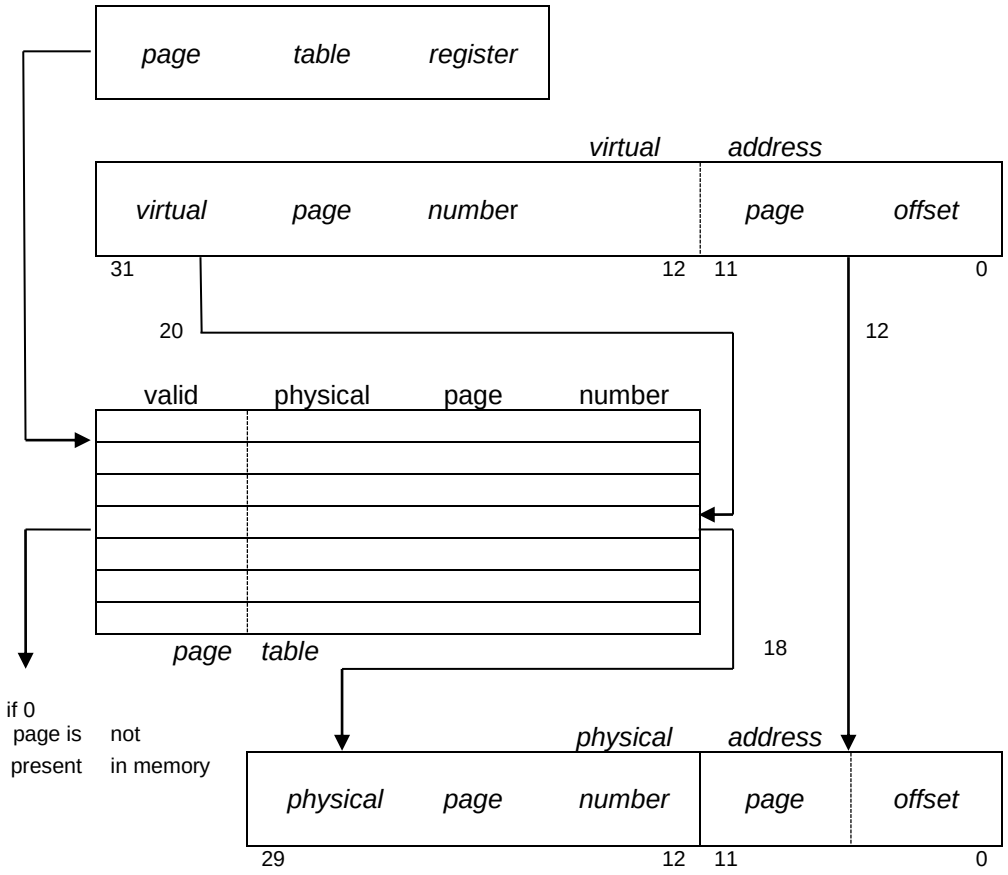
1. paginação (*paging*)
   divisão da memória física em blocos de bytes (*page frames*)
   e ajuste da memória de processo em páginas nesses blocos
   valores típicos podem ser 4K (x86-x64) ou 8K (RISC)

2. segmentação
   divisão dos espaços de endereçamento em porções (segmentos) para cada aplicação
   o endereço será composto de uma base (origem do segmento) e de um deslocamento
   o par [segmento:deslocamento] é convertido para um endereço linear (virtual)
   o endereço virtual é convertido para o correspondente físico enquadrado em um bloco.

A conversão de endereços fica a cargo de uma unidade para gerenciamento de memória (MMU - *Memory Management Unit),* que utilizará tabelas de páginas, pelo menos uma tabela por processo, relacionando o endereço virtual com o endereço físico do bloco correspondente, ou da localização da página em armazenamento secundário. Para ajudar na tradução de endereços, também faz uso de uma memória cache associativa TLB (*Translation Lookaside Buffer),* no qual mantém cópias dos valores mais recentes dos endereços virtuais e físicos solicitados.



Tradução do endereço virtual em endereço físico



Acesso Direto à Memória (DMA)

Através do Acesso Direto à Memória (DMA) é possível transferir dados entre a memória e dispositivos periféricos sem a participação do processador. O dispositivo responsável pelo controle da atividade de acesso direto à memória é chamado de controlador de DMA.

Há dois outros modos de se transferir dados entre memória e dispositivos de entrada e saída: o programado ou o controlado por interrupções.

No modo de transferência programado, o processador verifica se há algum dispositivo pronto para transferir dados. Se houver, o processador se ocupará totalmente da transferência, o que ocorrerá rapidamente, mas ficará empenhado durante todo o tempo em que isso durar.

No modo de transferência controlado por interrupções, quando um dispositivo estiver pronto para transferir dados, sinalizará uma interrupção ao processador. O processador continuará na execução da instrução atual, salvará o contexto, só então permitirá a transferência, o que ocorre, portanto, após certo atraso. O processador ainda estará totalmente envolvido no processo,

Esses modos não são úteis para se transferir blocos de dados maiores, o que poderá ser obtido pelo controlador de DMA, por meio de três modos:

1. *burst mode*
   uma vez no controle do barramento de sistema, o controlador de DMA só irá liberá-lo após completar a transferência, até então, o processador esperará pelo uso do barramento

2. *cycle stealing mode*
   nesse modo, o controlador de DMA forçará o processador a suspender suas operações e liberar o controle do barramento. Após concluir a transferência, devolverá o controle. Permanecerá nisso, subtraindo ciclos de *clock* a cada byte transferido.

3. *transparent mode*
   o controlador de DMA tomará o acesso do barramento do sistema somente se o processador não o estiver utilizando

Vantagens

1. A transferência sem o envolvimento do processador serve para aumentar velocidade da tarefa de leitura-escrita.

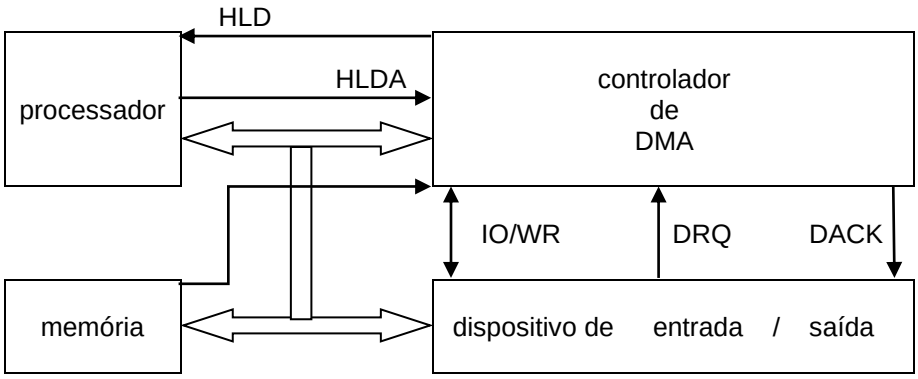
2. O acesso direto à memória reduz os ciclos de *clock* requeridos para se ler ou escrever um bloco de dados.

3. A implementação de acesso direto à memória reduz a sobrecarga do processador.

Desvantagens

1. Há um custo adicional para a implementação do controlador de DMA.

2. Pode ocorrer problema de coerência de cache durante o uso do controlador de DMA.

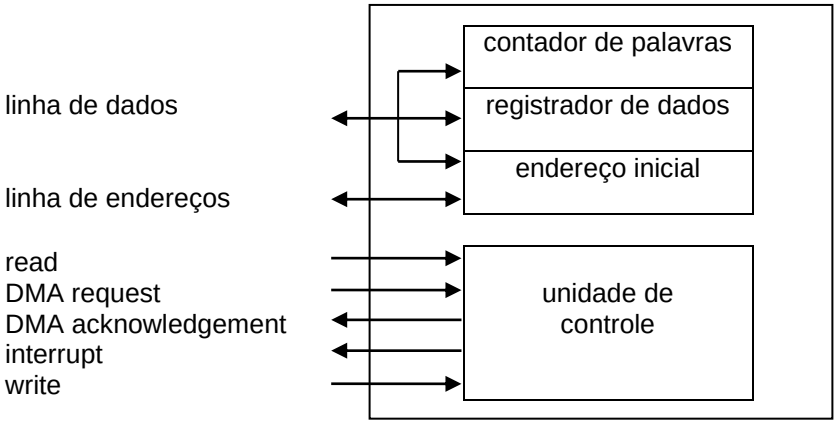


Quando um dispositivo de entrada/saída tiver quer transferir dados de/para a memória, enviará ao controlador de DMA uma requisição (DRQ). Se o controlador a aceitar, solicitará ao processador que aguarde alguns ciclos de *clock*, enviando-lhe uma solicitação (HLD).

Ao receber a solicitação (HLD) do controlador de DMA, o processador liberará o barramento e responderá com uma confirmação (HLDA).

Depois que receber a confirmação, o controlador de DMA informará ao dispositivo (DACK) que a transferência poderá ser realizada, e se encarregará do barramento durante a transferência.

Quando concluir a operação, o controlador de DMA informará ao processador e devolverá o controle do barramento, para que o processamento continue a partir de onde tiver parado.



Sempre que um processador receber solicitação para ler ou gravar um bloco de dados, deverá sinalizar o controlador DMA enviando as seguintes informações:

1. se os dados deverão ser lidos da memória, ou se serão gravados.
   Isso seguirá por meio de linhas de controle de leitura ou escrita
   gravação entre o processador e a unidade lógica do controlador de DMA.
2. O processador também fornecerá o endereço inicial de/para o bloco de dados
   na memória, bem como de onde deverá ser lido ou onde deverá ser escrito.
   O controlador DMA armazenará isso em seu registro de endereço.
3. O processador também enviará a quantidade de palavras, ou seja,
   quantas palavras deverão ser lidas ou escritas.
   Isso será armazenado no contador de dados ou registrador de contagem de palavras.
4. O mais importante é o endereço do dispositivo de entrada/saída que deseja
   ler ou gravar dados. Isso será armazenado no registrador de dados.

Paralelismo

Modelo de programação paralela

Um modelo de programação paralela é uma abstração de arquitetura de computadores também paralela, cujo objetivo é facilitar a expressão de algoritmos e a composição de programas.

As características de um bom modelo de programação podem ser avaliadas em pelo menos dois aspectos:

1. generalidade
   relativa à variedade de problemas que podem ter soluções expressas
   em diferentes arquiteturas

2. desempenho
   relativo à eficiência do código a ser executado
   em diferentes arquiteturas.

Formas pelas quais um modelo de programação paralela pode ser implementado por:

1. biblioteca adaptada para uma linguagem de programação sequencial
2. extensão de uma linguagem de programação convencional
3. linguagem de programação nova, naturalmente paralela.

A classificação de modelos de programação pode ser feita mediante o emprego de duas grandes categorias:

1. interação de processos
   relativa aos mecanismos pelos quais processos paralelos possam se comunicar:

   1.1 pelo uso eficiente de memória compartilhada

   - espaço global de endereçamento onde os processos
     possam ler e escrever de modo assíncrono,
     com arbítrio de condições para regular as tentativas de acessos simultâneos,
     a fim de evitar conflitos, mediante o emprego de intertravamentos, semáforos
     e monitores (exemplos: Cilk, CUDA, OpenMP)

   - processadores convencionais com vários núcleos
     geralmente oferecem suporte nativo à memória compartilhada

   1.2 por trocas de dados através de mensagens

   - assíncronas
     quando a mensagem pode ser enviada antes que o receptor esteja preparado
     para recebê-la, como no caso da formalização feita mediante
     CSP (Communicating Sequential Processes)
     (exemplos: D, Erlang, Scala)

   - síncronas
     quando a mensagem será enviada contando que o receptor esteja pronto
     para recebê-la, como no caso de emprego do modelo de atores
     (exemplos: Ada, Go, Occam, VerilogCSP)

1.3 por interação implícita
na qual a interação entre processos, ainda que complexa e difícil de gerenciar, não fique aparente ao programador, e dependa mais do compilador e/ou do ambiente de execução,

- linguagens para domínios específicos podem oferecer mecanismos para a execução concorrente de operações em níveis mais altos

- linguagens de programação funcionais procuram evitar a ocorrência de efeitos colaterais quando da execução em paralelo de funções não dependentes (exemplos: Concurrent Haskell, Concurrent ML)

2. decomposição de problemas em processos executáveis paralelamente

2.1 Paralelismo de tarefas
tem foco na execução de processos ou *threads* mediante comunicação por trocas de mensagens ou sinais em circuitos (exemplos: Verilog, VHDL)

2.2 Paralelismo de dados
tem foco na execução de operações sobre grupos de dados (arranjos ou matrizes) nas quais se possa trabalhar de modo independente sobre partições disjuntas (exemplo: Tensorflow)

2.3 Paralelismo implícito
depende mais do compilador (por exemplo, na paralelização automática de código), do ambiente de execução ou de características próprias do hardware (como o paralelismo em nível de instruções em arquiteturas superescalares).

Linguagens de programação paralela podem combinar mais de uma maneira para oferecer paralelismo.

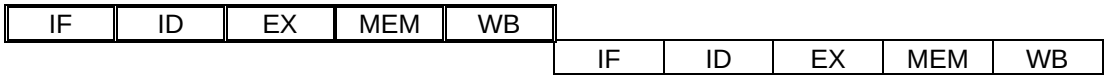
Tipos de paralelismo

Há quatro tipos básicos de paralelismo:

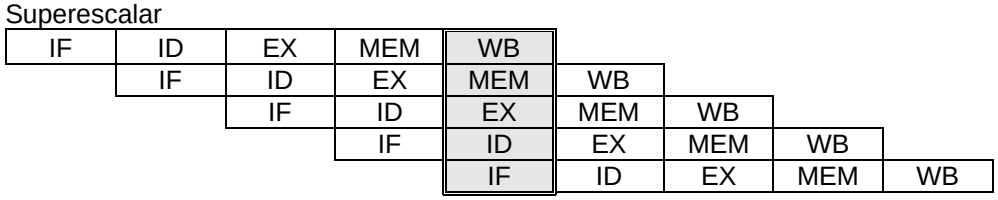
- ao nível de bits: o qual se obtém pelo aumento da palavra do processador, permitindo-se executar operações com dados maiores que o comprimento da mesma;

- ao nível de instrução: o qual se obtém otimizando os passos básicos para o tratamento de uma instrução: busca (IF – ***instruction fetch***), decodificação (ID – ***instruction decode***), execução (EX – ***execute***), acesso à memória (MEM) e reescrita (WB – ***writeback***).

Subescalar



Superescalar



- de dados: o qual se obtém distribuindo-se dados entre diversos nodos computacionais para que sejam processados em paralelo;

- de tarefas: o qual se obtém executando-se processamentos diferentes sobre um mesmo conjunto de dados ou sobre diferentes conjuntos de dados.

Tipos de computadores paralelos

Os tipos mais comuns de computadores paralelos são: com vários núcleos (***multicore***); simétricos (vários processadores com uma mesma memória compartilhada); distribuídos (conectados por rede: ***cluster***, massivamente paralelos; ***grid***); e os especializados (reconfiguráveis, de uso geral, específicos para aplicações e vetoriais).

Taxonomia segundo Flynn

SISD	SIMD
Single Instruction Single Data	***Single Instruction Multiple Data***
MISD	MIMD
Multiple Instruction Single Data	***Multiple Instruction Multiple Data***

- ***Single Instruction – Single Data***

Single instruction: apenas uma sequência de instruções pode ser executada por uma unidade de controle durante um ciclo de ***clock***
Single data: apenas uma sequência de dados pode ser usada como entrada durante um ciclo de ***clock***

Tipo de problema: uso geral
Tipo de execução: determinística
Tipo de computador: mainframes, minicomputadores e a maioria dos computadores atuais

- ***Single Instruction – Multiple Data***

Single instruction: apenas uma sequência de instruções pode ser executada em todas as unidades de controle durante o mesmo ciclo de ***clock***
Multiple data: apenas uma sequência de dados pode ser usada como entrada durante o mesmo ciclo de ***clock***

Tipo de problema: com grande regularidade tais como os de processamento de imagens
Tipo de execução: síncrona (***lockstep***) e determinística
Tipo de computador: arranjos de processadores, ***pipelines*** vetoriais, processadores com GPU's

- ***Multiple Instruction – Single Data***

Multiple instruction: sequências independentes de instruções podem ser executadas em todas as unidades de controle durante o mesmo ciclo de ***clock***
Single data: apenas uma sequência de dados pode ser usada como entrada durante um ciclo de ***clock***

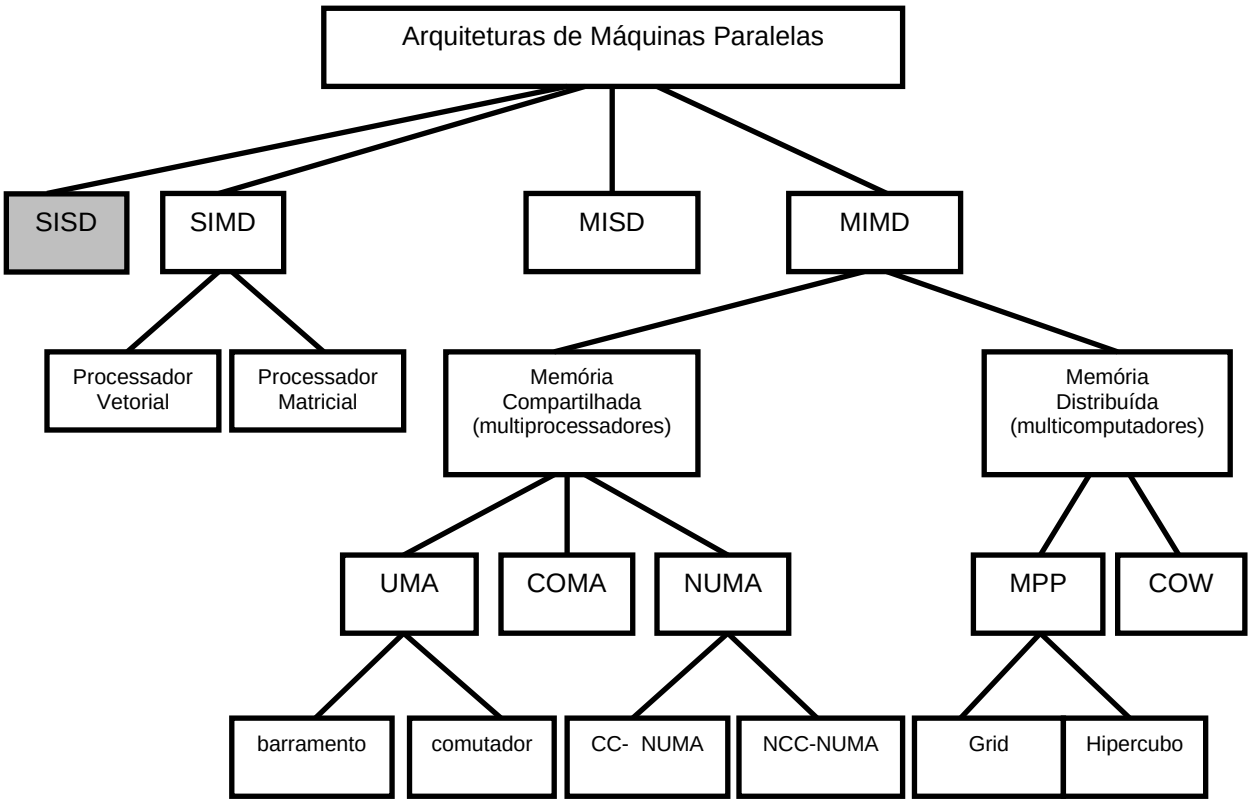
Tipo de problema: filtros de frequência e decifrar mensagem por vários algoritmos
Tipo de execução: assíncrona
Tipo de computador: Carnegie-Mellon C.mmp Computer (1971)

- ***Multiple Instruction – Multiple Data***

Multiple instruction: sequências independentes de instruções podem ser executadas em todas as unidades de controle durante o mesmo ciclo de ***clock***
Single data: sequências independentes de dados podem ser usadas como entrada durante um ciclo de ***clock***

Tipo de problema: filtros de frequência e decifrar mensagem por vários algoritmos
Tipo de execução: síncrona e assíncrona, determinística ou não-determinística
Tipo de computador: supercomputadores, ***clusters, grids***, computadores com múltiplos núcleos

Taxonomia adaptada segundo Tanenbaum



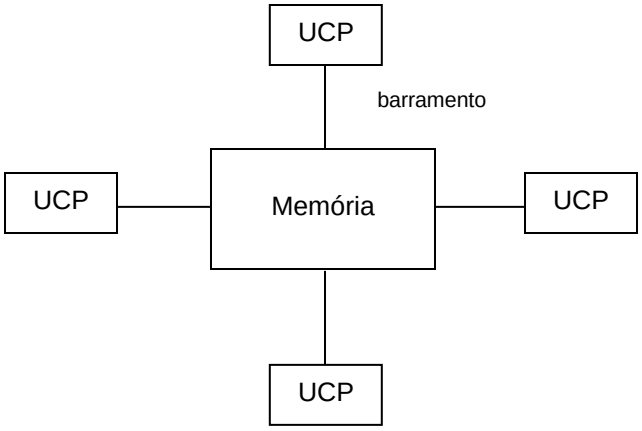
Arquiteturas de memória para computadores paralelos

- Memória compartilhada (***shared memory***)

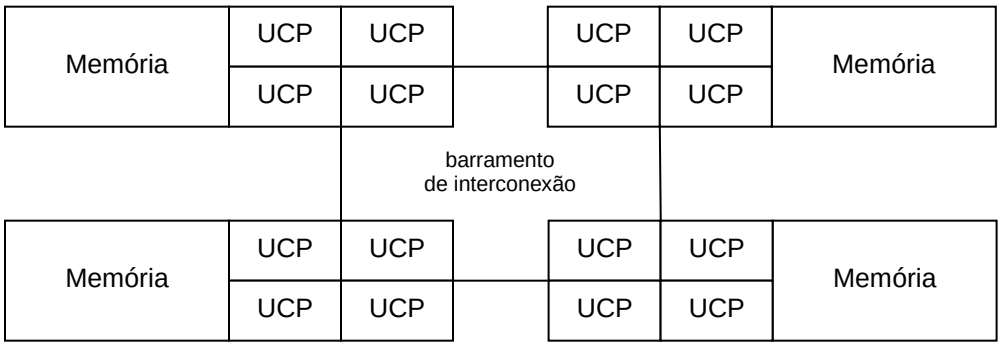
Vantagens: espaço de endereçamento global facilitado para o usuário
compartilhamento rápido e uniforme pelos processadores
coerência de cache mantida por ***hardware***

Desvantagens: falta de escalibilidade
(aumentar processadores implica maior tráfego no barramento)
maior responsabilidade para o programador para garantir acesso "correto"
(controle de sincronização)
maior custo
(para projetar e produzir memórias com múltiplos acessos)

- Modelo UMA (***Uniform Memory Access***)
  - Tipo de memória: a mesma com tempo de acesso igual para todos os processadores
  - Tipo de computador: multiprocessadores simétricos (SMP)



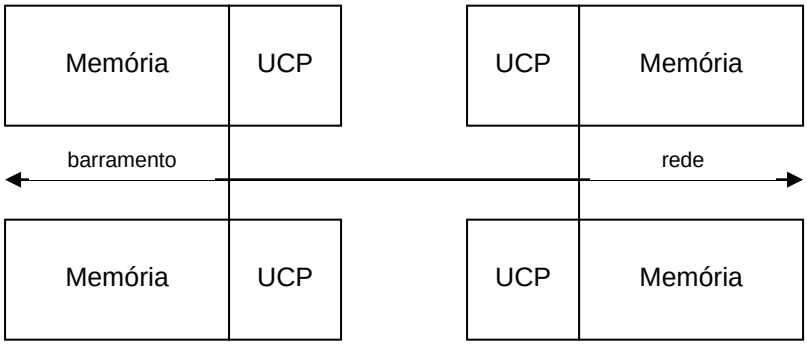
- Modelo NUMA (***Non-Uniform Memory Access***)
  - Tipo de memória: a mesma com tempo de acesso diferente entre processadores
  - Tipo de computador: conexões entre multiprocessadores simétricos (SMP)



- Memória distribuída (***distributed memory***)

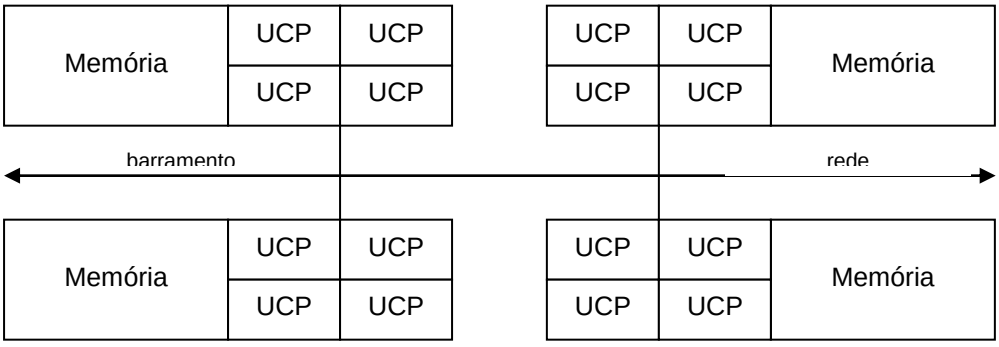
Vantagens: escalabilidade
acesso rápido à memória local pelo processador
não há trabalho extra (***overhead***) para manter coerência
custo compatível com processadores comuns e recursos de rede

Desvantagens: maior responsabilidade para o programador
para garantir comunicação de dados entre processadores
dificuldade em manter estruturas de dados
baseadas em memória global
tempo de acesso não uniforme



- Memória híbrida (***hybrid distributed-shared memory***)

Modelo comum à maioria dos processadores simétricos (SMP) atuais.
Os requisitos de acesso à memória dependem da infraestrutura de rede.



Modelos de programação paralela

Um modelo de programação paralela é uma abstração de arquitetura de computadores também paralela, cujo objetivo é facilitar a expressão de algoritmos e a composição de programas.

As características de um bom modelo de programação podem ser avaliadas pelo menos dois aspectos:

1. generalidade
   relativa à variedade de problemas que podem ter
   soluções expressas em diferentes arquiteturas
2. desempenho
   relativo à eficiência do código a ser executado
   em diferentes arquiteturas.

Formas pelas quais um modelo de programação paralela pode ser implementado podem ser:

1. biblioteca adaptada para uma linguagem de programação
   sequencial
2. extensão de uma linguagem de programação convencional
3. linguagem de programação nova, naturalmente paralela.

A classificação de modelos de programação pode ser feita mediante o emprego de duas grandes categorias:

1. interação de processos
   relativa aos mecanismos pelos quais processos paralelos possam se comunicar:
   1.1 pelo uso eficiente de memória compartilhada
      - espaço global de endereçamento onde os processos possam ler e escrever
        de modo assíncrono, com arbítrio de condições para regular as tentativas
        de acessos simultâneos, a fim de evitar conflitos, mediante o emprego de
        intertravamentos, semáforos e monitores (exemplos: Cilk, CUDA, OpenMP)
      - processadores convencionais com vários núcleos geralmente oferecem suporte nativo
        à memória compartilhada

   1.2 por trocas de dados através de mensagens
      - assíncronas
        quando a mensagem pode ser enviada antes que o receptor esteja pronto para recebê-la,
        como no caso da formalização feita em CSP (Communicating Sequential Processes)
        (exemplos: D, Erlang, Scala)
      - síncronas
        quando a mensagem será enviada contando que o receptor esteja pronto para recebê-la,
        como no caso de emprego do modelo de atores
        (exemplos: Ada, Go, Occam, VerilogCSP)

   1.3 por interação implícita
      na qual a interação entre processos, ainda que complexa e difícil de gerenciar,
      não fique aparente ao programador, e dependa mais do compilador e/ou
      do ambiente de execução,
      - linguagens para domínios específicos podem oferecer mecanismos
        para a execução concorrente de operações em níveis mais altos
      - linguagens de programação funcionais procuram evitar a ocorrência de efeitos colaterais
        quando da execução em paralelo de funções não dependentes
        (exemplos: Concurrent Haskell, Concurrent ML)

2. decomposição de problemas em processos executáveis paralelamente

2.1 Paralelismo de tarefas
tem foco na execução de processos ou threads
mediante comunicação por trocas de mensagens
ou sinais em circuitos (exemplos: Verilog, VHDL)

2.2 Paralelismo de dados
tem foco na execução de operações sobre grupos de dados
(arranjos ou matrizes) nas quais se possa trabalhar de
modo independente sobre partições disjuntas
(exemplo: Tensorflow)

2.3 Paralelismo implícito
depende mais do compilador (por exemplo, na paralelização
automática de código), do ambiente de execução ou
de características próprias do hardware (como o paralelismo
em nível de instruções oferecidos por arquiteturas superescalares).

Linguagens de programação paralela podem combinar mais de uma maneira de oferecer paralelismo.

Principais tipos de paralelismos

- Memória compartilhada

Tarefas compartilham um espaço de endereçamento comum no qual podem ler e escrever assincronamente. Mecanismos como travas (***locks***) e semáforos (***semaphores***) podem ser usados para o controle de acesso à memória compartilhada.

Vantagens: simplicidade no desenvolvimento de programas
não há necessidade de um controle explícito da comunicação entre tarefas

Desvantagens: dificuldade em gerenciar a localidade dos dados
(manter dados locais conserva os acessos à memória, evita refrescamento de cache e tráfego no barramento, mas isso pode ser dificultado pelo uso de vários processadores ao mesmo tempo)

- ***Threads***

Um processo simples pode ter múltiplas sequências de execução concorrentes, cada um com acesso a dados locais, mas compartilhando recursos de um mesmo programa, como o espaço de memória global, através do qual se comunicam usando mecanismos de sincronização.

- Troca de mensagens

Um conjunto de tarefas que trabalham sobre suas próprias memórias locais podem se comunicar através de troca de mensagens, e assim agir de forma cooperativa.

- Dados paralelos

Um conjunto de tarefas trabalha sobre o mesmo conjunto de dados, mas cada uma delas lida com uma porção específica da estrutura de dados.

- Híbridos

Combinações de dois ou mais modelos de programação paralela, como por exemplo, troca de mensagens com ***threads*** ou memória compartilhada. Outro modelo pode combinar dados paralelos com troca de mensagens.

- SPMD (***Single Program Multiple Data***)

Um mesmo programa é executado por todas as tarefas simultaneamente, mas não exatamente as mesmas instruções ao mesmo tempo, uma vez que cada tarefa lidará com uma porção diferente do conjunto de dados.

- MPMD (***Multiple Program Multiple Data***)

Vários programas podem estar ativos, trabalhando em paralelo.

Prova de teoremas

As relações descritas podem ser utilizadas em Lógica Formal para provar teoremas.

Exemplo 1:
Provar (s') dados:

1. t - premissa
2. t $\rightarrow$ q' - premissa
3. q' $\rightarrow$ s' - premissa

4. q' - *modus ponens* entre 1 e 2
5. s' - *modus ponens* entre 3 e 4 (c.q.d.)

Exemplo 2:
Provar (a) dados :

1. a' $\rightarrow$ b - premissa
2. b $\rightarrow$ c - premissa
3. c' - premissa

4. b' - *modus tolens* entre 2 e 3
5. (a')' - *modus tolens* entre 1 e 4
6. a - dupla negação (c.q.d.)

Quantificadores

A Lógica de Predicados, ou Lógica de Primeira Ordem, associa o uso de quantificadores.

Quantificador Universal

$(\forall x \in S) \;\; P(x)$ - ***para todo*** x em S, P(x) é verdadeiro

Quantificador Existencial

$(\exists x \in S) \;\; P(x)$ - ***para algum*** x em S, P(x) é verdadeiro

Os quantificadores admitem complementação :

não $\left((\forall x) \;\; P(x) \right)$ = $(\exists x)$ (não P(x))

não $\left((\exists x) \;\; P(x) \right)$ = $(\forall x)$ (não P(x))

Exemplo :
Supor o predicado abaixo, devidamente quantificado :

$$(\forall x \in R) \;\; (x^2 + 2x - 1 > 0)$$

não $\left((\forall x \in R) \;\; (x^2 + 2x - 1 > 0) \right) \rightarrow (\exists x \in R) \;\; (x^2 + 2x - 1 \leq 0)$

o que pode ser verificado quando x = 0 !

Exercícios propostos

1. Provar, pela álgebra, que :

   a) $(p \cdot q') + p' = p' + q'$

   b) $p \cdot (p' + q) = p \cdot q$

   c) $p \cdot q + (p' + q')' = (p' + q')'$

   d) $(p \cdot q + r) \cdot ((p \cdot q)' \cdot r') = 0$

   e) $(p \cdot q + r') \cdot (p \cdot q \cdot r' + p \cdot r') = p \cdot r'$

2. Fazer as tabelas e os circuitos do exercício anterior.

3. Demonstrar as relações abaixo através de tabelas:

   a) $x + x \cdot y = x$ (absorção)

   b) $x \cdot (x + y) = x$ (absorção)

   c) $(x + y) \cdot (x + z) = x + y \cdot z$

   d) $x + x' \cdot y = x + y$

   e) $x \cdot y + y \cdot z + y' \cdot z = x \cdot y + z$

4. Verificar o resultado das seguintes relações :

   a) $(x \rightarrow y) \cdot x \rightarrow y$ ("modus ponens")

   b) $(x \rightarrow y) \cdot y' \rightarrow x'$ ("modus tolens")

   c) $(x + y) \cdot x' \rightarrow y$ (silogismo disjuntivo)

   d) $(x \rightarrow y) \cdot (y \rightarrow z) \rightarrow (x \rightarrow z)$ (silogismo hipotético)

   e) $((x \rightarrow y) \cdot (w \rightarrow z)) \cdot (x+w) \rightarrow (y+z)$ (dilema)

   f) $(x \rightarrow y) \rightarrow (x \rightarrow x \cdot y)$ (absorção)

   g) $x \cdot y \rightarrow x$ (simplificação)

   h) $x \rightarrow x + y$ (adição)

   i) $(x' \rightarrow x) \rightarrow x$ (contradição)

   j) $(x' \rightarrow y) \rightarrow ((x' \rightarrow y') \rightarrow x)$ (contradição)

   h) $y \cdot (y \cdot x' \rightarrow z) \cdot (y \cdot x' \rightarrow z') \rightarrow x$ (contradição)

   i) $(x \rightarrow y) = (x \cdot y') \rightarrow (y \cdot y')$ (redução ao absurdo)

5. Verificar o resultado das seguintes relações

a) $x \cdot (y \cdot z) \Leftrightarrow (x \cdot y) \cdot z$ (associatividade)

b) $x + (y + z) \Leftrightarrow (x + y) + z$ (associatividade)

c) $x \cdot y \Leftrightarrow y \cdot x$ (comutatividade)

d) $x + y \Leftrightarrow y + x$ (comutatividade)

e) $x \cdot (y+z) \Leftrightarrow (x \cdot y) + (x \cdot z)$ (distributividade)

f) $x+(y \cdot z) \Leftrightarrow (x+y) \cdot (x+z)$ (distributividade)

g) $(x \cdot y)' \Leftrightarrow x' + y'$ (De Morgan)

h) $(x+y)' \Leftrightarrow x' \cdot y'$ (De Morgan)

i) $(x')' \Leftrightarrow x$ (involução)

j) $x \cdot x \Leftrightarrow x$ (idempotência)

k) $x + x \Leftrightarrow x$ (idempotência)

l) $x \rightarrow y \Leftrightarrow x' + y$ (implicação)

m) $(x \Leftrightarrow y) \Leftrightarrow (x \rightarrow y) \cdot (y \rightarrow x)$ (equivalência)

n) $x \rightarrow y \Leftrightarrow x' \rightarrow y'$ (contraposição)

o) $(x \cdot y) \rightarrow z \Leftrightarrow x \rightarrow (y \rightarrow z)$ (exportação)

6. Transformar as afirmativas abaixo em proposições :

a) A soma de dois inteiros é maior que o primeiro.
b) A soma de dois inteiros é maior que a diferença entre eles.
c) O produto de um inteiro por zero é menor que outro inteiro.
d) Se um de dois números são nulos, o produto deles é zero.
e) Se dois números são iguais a um terceiro, então são iguais.

7. Construir um circuito para identificar se dois bits são iguais.

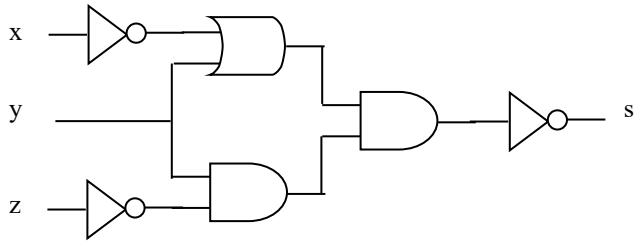
8. Construir um circuito capaz de fornecer a tabela abaixo :

x y z	f (x,y,z)
0 0 0	1
0 0 1	0
0 1 0	0
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	0
1 1 1	1

9. Simplificar a expressão lógica abaixo :

(x > 0 e y = 0) ou não (x < 0 ou z > 0)

10. Identificar a equação do circuito abaixo pela soma de produtos (SoP):



11. Identificar a equação do circuito anterior pelo produto das somas (PoS).

12. Construir um circuito equivalente ao da questão (10) usando apenas portas NAND.

13. Construir um circuito equivalente ao da questão (10) usando apenas portas NOR.

14. Montar um diagrama de um somador completo de 2 palavras de 2 bits cada.

15. Construir um circuito equivalente ao da questão (14) usando apenas portas NAND.

16. Construir um circuito equivalente ao da questão (14) usando apenas portas NOR.

17. Simplificar por mapa de Karnaugh: a'b'c'd'+a'b'c'd+a b'c'd+a b c'd'+a b c d'+a b c'd+a b c d

18. Demonstrar a validade do seguinte argumento:

(1) x < 6
(2) y > 7 ou x = y $\rightarrow$ (y = 4 e x < y)
(3) y = 4 $\rightarrow$ x < 6
(4) x < 6 $\rightarrow$ x < y
$\therefore$ x = y

19. Verificar se as proposições são falsas ou verdadeiras:

a) $(\forall n \in N)$ (n+4 > 3)

b) $(\forall n \in N)$ (n+3 > 7)

c) $(\exists n \in N)$ (n+4 < 7)

d) $(\exists n \in N)$ (n+3 < 2)

20. Sendo A = {1,2,3,4,5} determinar a negação de:

a) $(\exists n \in A)$ (x+4 = 10)

b) $(\forall n \in N)$ (x+4 < 10)