EA075
Hierarquia de Memória / Memória Cache

Faculdade de Engenharia Elétrica e de Computação (FEEC)
Universidade Estadual de Campinas (UNICAMP)

Prof. Rafael Ferrari

(Documento baseado nas notas de aula do Prof. Levy Boccato)

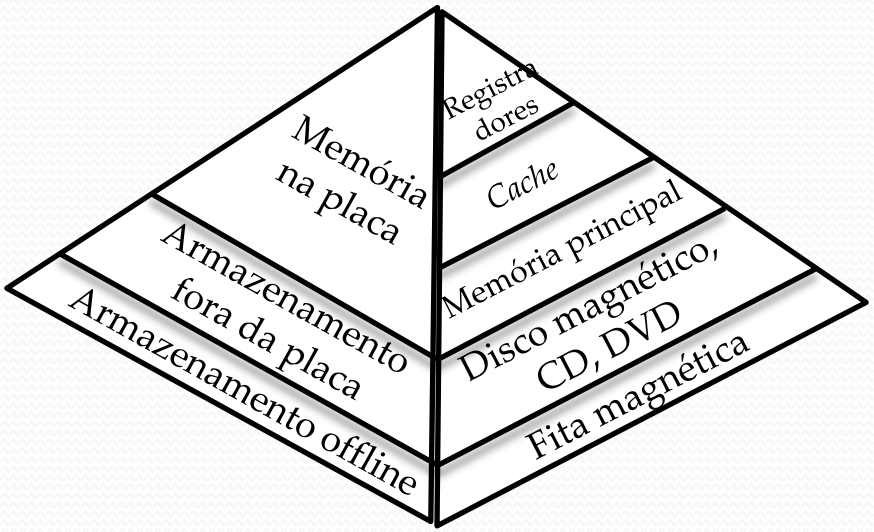
Introdução

- Idealmente, um programador/usuário de um sistema de computação gostaria de usar tecnologias de memória que oferecessem:
  - Grande capacidade de armazenamento.
  - Alto desempenho, i.e., tempo de acesso baixo o suficiente para acompanhar a velocidade do processador.
  - Baixo custo.

- Porém,
  - ↑ desempenho normalmente significa ↑ custo.
  - ↑ capacidade leva a ↓ desempenho.

Introdução

- Para contornar este dilema, utiliza-se uma **hierarquia de memória**.



- ✓ Diminui o custo por bit.
- ✓ Aumenta a capacidade (em particular, a densidade) de armazenamento.
- ✓ Aumenta o tempo de acesso (desempenho mais lento).
- ✓ Diminui a frequência de acesso à unidade pelo processador.

- Memórias menores, mais caras e mais rápidas são complementadas por memórias com maior capacidade, mais baratas, porém mais lentas.

Hierarquia de memória

- A **chave** para o sucesso de uma hierarquia de memória consiste em construir uma estratégia de uso dos elementos de memória de forma a garantir que a **frequência de acesso diminua à medida que a distância dos dispositivos de memória em relação ao processador aumenta.**

- Em outras palavras, a frequência de acesso a dispositivos de armazenamento rápidos deve ser muito maior que a frequência de acesso a dispositivos mais lentos.

- A base para o avanço de desempenho em uma hierarquia de memória é o chamado **princípio da localidade**.

- Segundo este princípio, um programa tende a acessar uma porção relativamente pequena do seu espaço de endereçamento durante qualquer intervalo de tempo. Ou, em outras palavras, um programa não acessa todo o seu código (ou seus dados) de uma vez com igual probabilidade.

Hierarquia de memória

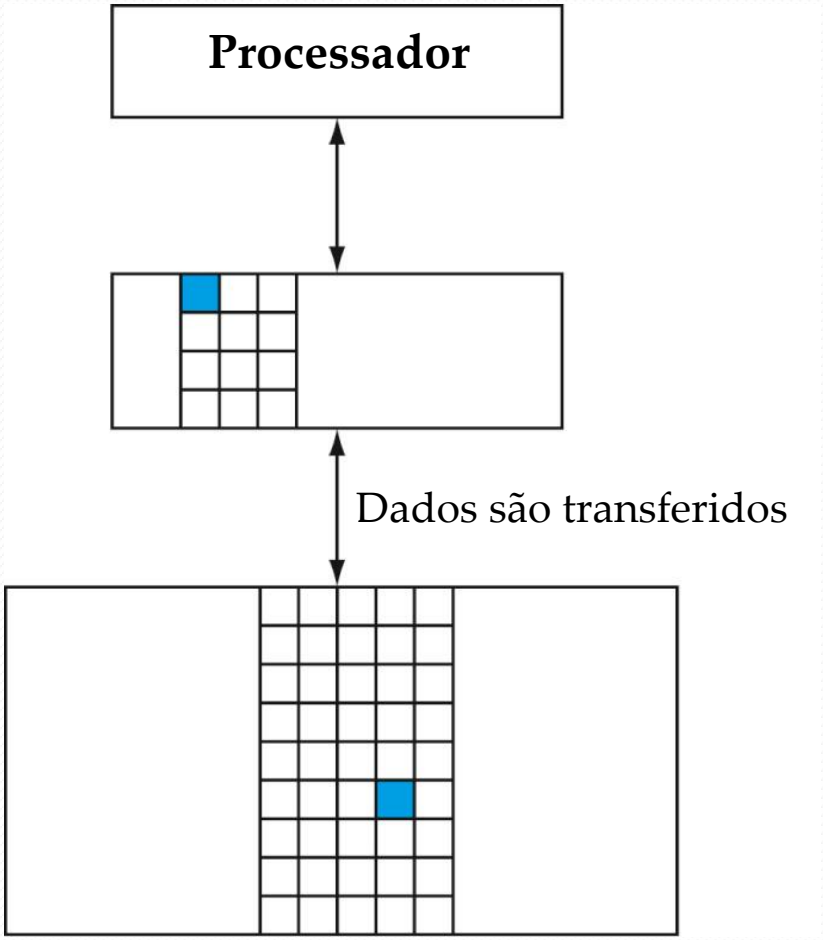
- **Localidade espacial:** refere-se à tendência de a execução de um programa envolver uma série de locais de memória que estão próximos. Esta propriedade decorre:
  - Da tendência de um processador acessar as instruções sequencialmente.
  - Da tendência de um programa acessar locais de dados sequencialmente, como ao processar um vetor ou uma tabela de dados.
- **Localidade temporal:** refere-se à tendência de um processador acessar locais de memória que foram usados recentemente.
  - Por exemplo, durante a execução de um laço de instruções.

Hierarquia de memória

- Possíveis argumentos a favor da localidade:
  - Na maior parte dos casos (exceto desvios e chamadas de procedimentos), a próxima instrução a ser apanhada da memória vem imediatamente após a última instrução lida.
  - É raro ter sequências longas de chamadas de procedimento seguidas por retornos de subrotina.
  - As construções iterativas normalmente executam um número relativamente pequeno de instruções.
  - Grande parte das operações envolve estruturas de dados do tipo *array* ou sequências de registros, cujas referências são localizadas (i.e., os itens estão próximos).

Hierarquia de memória

- **Níveis da hierarquia:**



• **Bloco (linha):** unidade de informação – pode ser formado por múltiplas palavras (*bytes*).

• Se o dado solicitado pelo processador está em algum bloco guardado no nível superior da hierarquia, ocorre um acerto (*hit*).

- *Hit rate*: número de *hits*/total de acessos.
- *Hit time*: tempo necessário para acessar o nível superior da hierarquia de memória, que inclui o tempo consumido para determinar se, para um certo acesso, ocorrerá um *hit* ou um *miss*.

• Se o dado não é encontrado, ocorre uma ausência (miss).

- *Miss penalty*: tempo para substituir um bloco do nível superior pelo bloco correspondente vindo do nível inferior, mais o tempo para entregar o dado ao processador.

Hierarquia de memória

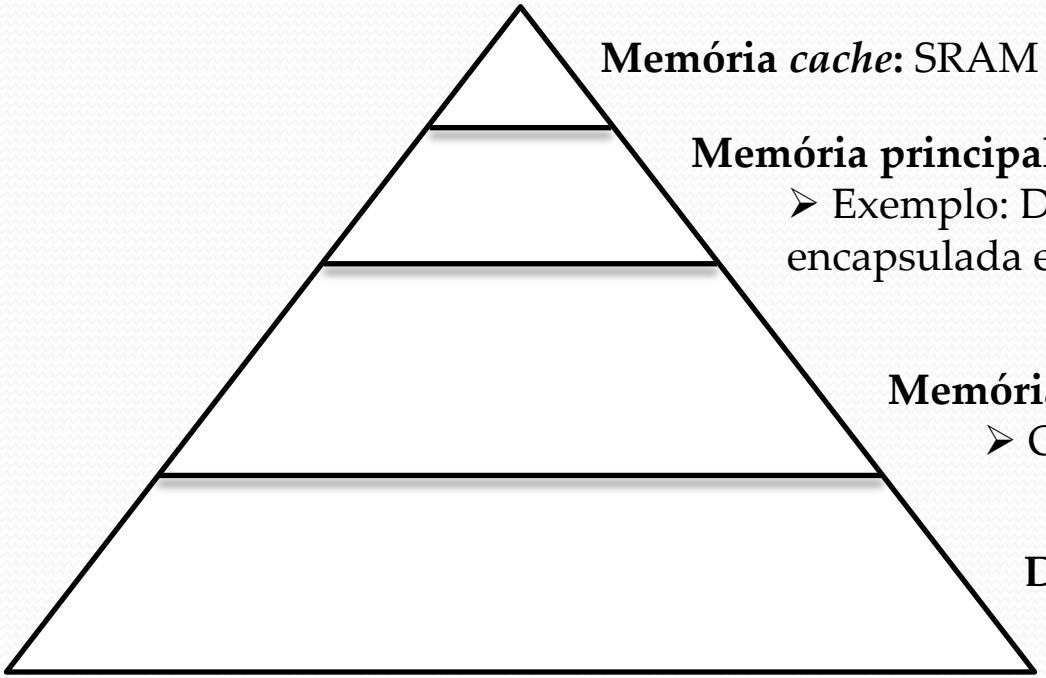
- **Tecnologias de memória:**

  - Os diferentes níveis em uma hierarquia de memória exploram diferentes tecnologias.

Tecnologia de memória	Tempo de acesso típico	$ por GiB (2012)
SRAM	0,5 – 2,5 ns	$500 - $1000
DRAM	50 – 70 ns	$10 - $20
Flash	5.000 – 50.000 ns	$0,75 - $1,0
Disco magnético (HD)	5.000.000 – 20.000.000 ns	$0,05 - $0,10

Hierarquia de memória

- **Panorama:**



Memória *cache*: SRAM

Memória principal: DRAM

- Exemplo: DDR4 SDRAM organizada ou encapsulada em DIMMs (*dual inline memory modules*)

Memória *flash*: SSD (*solid-state drive*)

- Opção mais cara que HDs.

Disco rígido: HDD (*hard disk drive*)

Hierarquia de memória

- **Objetivo:** por meio da construção de uma hierarquia de memória, cria-se a "ilusão" de que existe uma memória suficientemente grande (ilimitada) que pode ser acessada tão rapidamente quanto uma memória pequena e extremamente veloz.

- Os dados também são armazenados de forma hierárquica:
  - Um nível mais próximo do processador contém um subconjunto dos dados armazenados nos níveis mais afastados.
  - Dados são transferidos entre dois níveis adjacentes.

- Uma hierarquia de memória bem sucedida apresenta uma taxa de acerto alta o suficiente de modo que o tempo de acesso efetivo à memória é aproximadamente o tempo de acesso da memória mais rápida do sistema.

Hierarquia de memória

- **Como explorar o princípio da localidade?**

  - A localidade espacial geralmente é explorada usando blocos de memória maiores que são guardados na memória *cache*, e incorporando mecanismos de pré-busca (busca antecipada de itens) na lógica de controle da *cache*.

  - A localidade temporal é explorada mantendo valores de dados e instruções que foram usados recentemente na memória *cache* e empregando uma hierarquia de *cache*.

Cache

- **Memória *cache*:** unidade de armazenamento de baixa capacidade, alto custo (cara) e de rápido acesso que armazena cópias de partes da memória principal mais comumente ou provavelmente acessadas pelo processador.
  - Usualmente, é projetada usando SRAMs.

- **Funcionamento:**
  - Quando o processador tenta acessar uma palavra da memória, faz-se uma verificação para determinar se aquela palavra (i.e., se aquele endereço) está na *cache*.

  - Se estiver, ela é rapidamente entregue ao processador. Caso contrário, um bloco da memória principal, que consiste de um número fixo de palavras, é trazido para a *cache* e, então, a palavra solicitada é fornecida ao processador.

Cache

- **Funcionamento:**

  - Como o número de blocos que a *cache* consegue armazenar é inferior à capacidade da memória principal, é necessário elaborar um mecanismo para **mapear** os **blocos da memória principal** nas **linhas** da *cache*.

  - Além disso, é preciso criar um mecanismo para determinar qual bloco da memória principal ocupa, de fato, uma linha específica da *cache* – para que seja possível descobrir se um bloco de memória está ou não na *cache*.

Cache

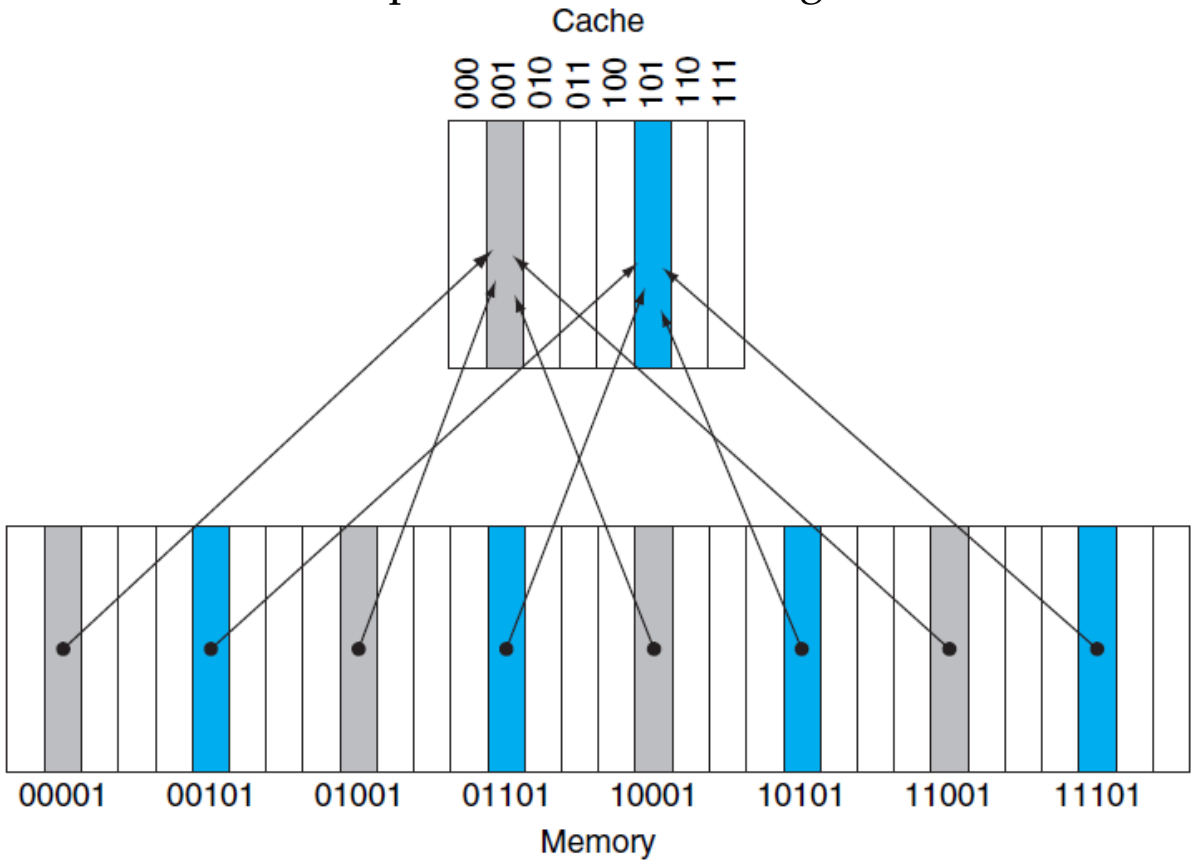
- **Mapeamento direto:**
  - Mapeia cada bloco da memória principal em apenas uma linha da *cache*. Para isto, o endereço de memória é dividido em três partes:

Tag	Índice / Linha	Offset / Palavra

  - **Índice / Linha:** identifica a linha da *cache*.
    - Se a *cache* possui m linhas, este campo tem $\log_2 m$ bits.
  - **Offset / Palavra:** identifica uma palavra (ou *byte*) dentro de um bloco de memória principal (portanto, dentro da linha da *cache*).
    - Se o bloco tem K palavras, este campo requer $\log_2 K$ bits.
  - ***Tag* (Etiqueta):** identifica o bloco de memória.

Cache

- **Mapeamento direto:**
  - Exemplo: memória de 32 (2^5) palavras e *cache* de 8 (2^3) palavras.
    - Cada linha armazena uma única palavra (0 bits de offset)
    - A tag são os dois bits mais significativos do endereço
    - A linha é identificada pelos 3 bits menos significativos do endereço.



Cache

- **Mapeamento direto:**
  - Esta técnica é relativamente simples para se implementar.
  - **Vantagem:** como um bloco de memória só pode ocupar uma linha específica, para saber se ele está na *cache*, basta comparar a *tag* do bloco desejado com aquela presente na linha da *cache*.
  - **Desvantagem:** ainda que haja espaço disponível na *cache*, um bloco só pode ser colocado numa posição específica, sobrescrevendo, assim, um eventual bloco que já ocupe aquela linha.

Cache

- **Mapeamento totalmente associativo:**
  - Permite que cada bloco da memória principal seja carregado em qualquer linha da *cache*.

    - A *tag* identifica o bloco da memória principal. Usa $\log_2 B$ bits, onde B é o número de blocos (de memória) existentes.
    - O campo de offset contém $\log_2 K$ bits, onde K é o número de palavras contidas em um bloco.
    - O número de linhas da memória *cache* não é determinado pelo formato do endereço.

  - **Desvantagem:** complexidade do circuito necessário para comparar as *tags* de todas as linhas da *cache* em paralelo.

Cache

- **Mapeamento associativo em conjunto:**

  - Representa um meio-termo que aproveita os pontos fortes das técnicas direta e associativa, enquanto ameniza suas desvantagens.

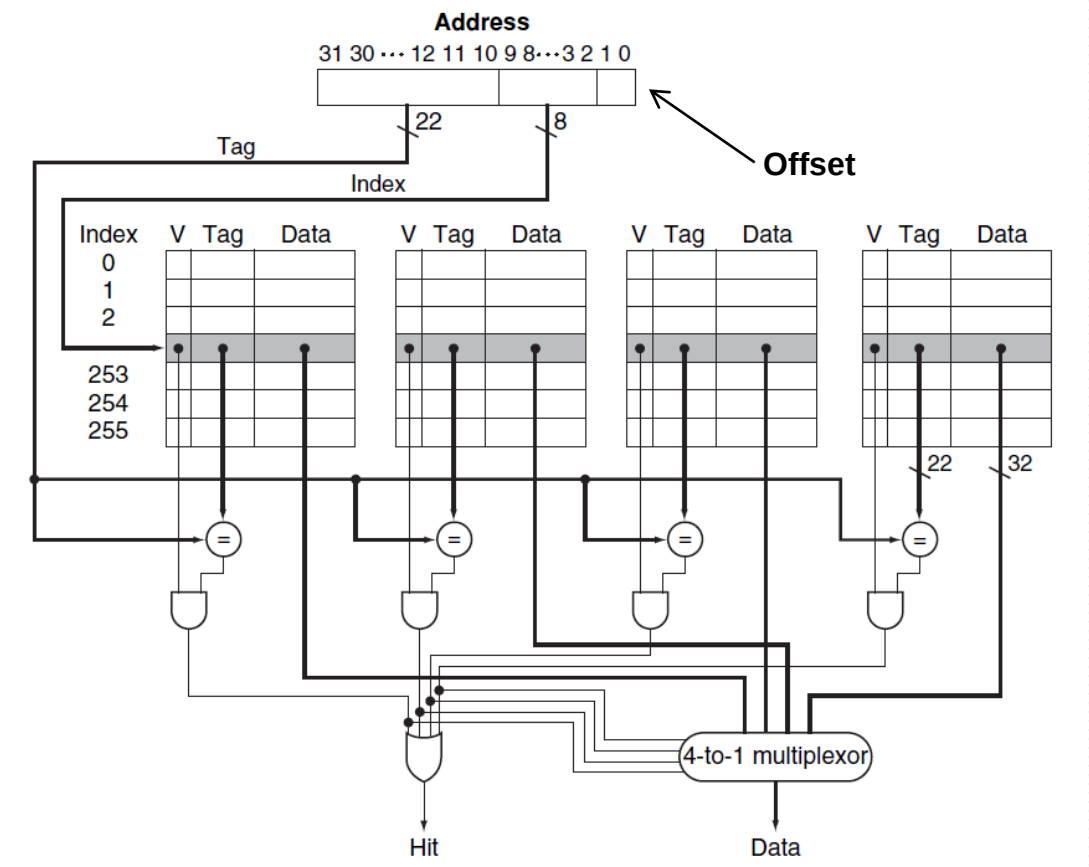
  - Cada bloco da memória é mapeado a um único conjunto, definido pelos bits de índice.

  - Dentro de seu conjunto, o bloco pode ocupar qualquer uma das *n*-vias disponíveis.

  - É possível descobrir se um bloco está ou não na *cache* comparando sua *tag* com todas as *tags* existentes no conjunto.

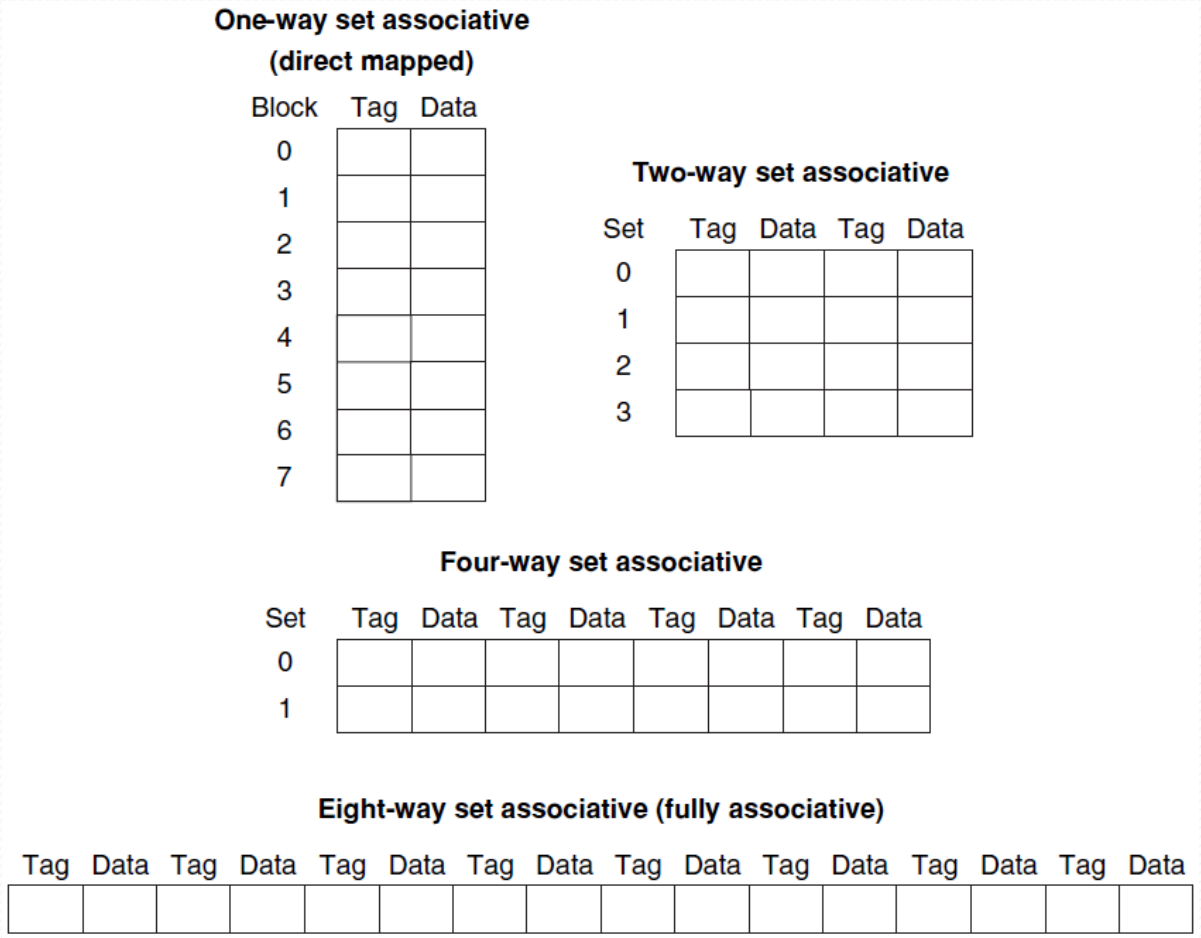
Cache

- **Mapeamento associativo em conjunto:**
  - Exemplo: cache associativa em conjunto com 256 conjuntos, 4 vias e 4 palavras por bloco.



Cache

- **Exemplo:** *cache* com capacidade de armazenamento de 8 blocos configurada usando mapeamento direto, mapeamento associativo em conjuntos com 2 e 4 vias e mapeamento totalmente associativo.



Cache

- Se a memória *cache* está cheia e um novo bloco deve ser trazido da memória principal, uma posição na *cache* precisa ser alocada, de maneira que o bloco ali residente seja substituído.

  - No caso do mapeamento direto, não há escolha: o bloco situado na linha selecionada é substituído.

  - Para as técnicas associativa e associativa em conjunto, o uso de um **algoritmo de substituição** se faz necessário.

Cache

- **Estratégias de substituição:**
  - Aleatória: a linha a ser substituída é escolhida aleatoriamente.
  - *First-in first-out* (FIFO): o bloco mais antigo, i.e., que permaneceu mais tempo na *cache*, é substituído.
    - Uma fila com os índices das linhas pode ser usada para indicar a próxima a ser substituída.
  - Menos frequentemente utilizado (*least-frequently used*, LFU): substitui a linha que teve menos referências.
  - Menos recentemente utilizada (*least-recently used*, LRU): substitui a linha que permaneceu mais tempo na *cache* sem ser acessada.

Cache

- Quando um dado é escrito na *cache*, é preciso atualizá-lo na memória principal em algum momento.

- A abordagem mais simples é denominada *write-through* (escrita através).
  - Todas as operações de escrita são feitas tanto na *cache* quanto na memória principal.
  - Esta opção mantém a consistência entre os dados, sendo de fácil implementação.
  - **Desvantagem:** o processador tem de esperar pela lenta operação de escrita na memória principal toda vez que alterar o conteúdo de um determinado bloco. Isto gera um tráfego de memória considerável que pode se tornar um gargalo.
  - Além disso, é possível que algumas operações de escrita potencialmente desnecessárias venham a ser realizadas.

Cache

- Uma alternativa é denominada *write-back* (escrita na volta).
  - As atualizações são feitas apenas na *cache*.
  - Toda vez que uma palavra tem seu conteúdo alterado na *cache*, um bit de modificação (bit de uso ou *dirty bit*) associado à linha é marcado.
  - Quando um bloco é substituído na memória *cache*, ele é escrito de volta na memória principal **se, e somente se,** o bit de uso estiver marcado.
  - Esta opção tende a reduzir o número de operações de escrita (lentas) na memória principal.

Cache

- **Coerência de *cache*:** em sistemas multiprocessadores, a memória principal é compartilhada e cada processador possui sua própria *cache*. Quando todas as unidades de *cache* e a memória principal mantêm dados válidos e consistentes uns com os outros, tem-se coerência de *cache*.

- ***Cache* de instrução e *cache* de dados**
  - **Trade-offs:** projeto e implementação de duas *caches*, além de, eventualmente, ter uma taxa de acerto menor que uma *cache* maior unificada.
  - **Atrativo:** pode eliminar a disputa pela *cache* entre a unidade de busca / decodificação de instruções e a unidade de execução em uma arquitetura de processador com *pipeline*.

Cache

- **Desempenho:**
  - Considere que um sistema possui hierarquia de memória com dois níveis: memória principal e *cache*.
    - Quando o dado desejado está na *cache*, ocorre um *hit* e a informação é acessada rapidamente (t_h).
    - Caso contrário, um bloco da memória principal é trazido para a *cache*, e o acesso é, então, feito a partir desta cópia local. A latência (penalidade) adicional devido ao acesso à memória principal é $t_{\text{MEM.}}$
  - **Tempo médio de acesso:** depende da probabilidade de o dado estar na memória *cache*.

$$t_{\text{médio}} = P\{\text{dado na cache}\}t_h + (1 - P\{\text{dado na cache}\})(t_h + t_{\text{MEM}})$$

  - Simplificando,

$$t_{\text{médio}} = t_h + (1 - P\{\text{dado na cache}\})t_{\text{MEM}}$$

Bibliografia

- Vahid, Frank, and Tony Givargis. *Embedded system design: a unified hardware/software introduction*. Vol. 4. New York, NY: John Wiley & Sons, 2002.
- Patterson, David A., and John L. Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013. (capítulo 5, seções 5.1, 5.3 e 5.4)