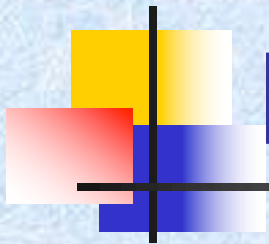




# Fundamentos de Sistemas Computacionais

---

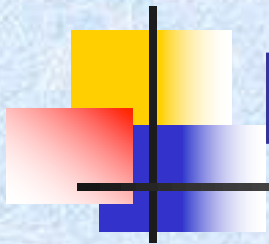
Revisão – Caches e  
Memória RAM



# Princípio da Localidade

---

- O uso da hierarquia de memória, na qual a cache está incluída, justifica-se pelo Princípio da Localidade.
- Princípio da Localidade: os programas tendem a reusar dados e instruções que foram usadas recentemente.
  - Possibilidade de se prever as instruções que serão utilizadas em um futuro próximo com base nos acessos feitos em um passado recente

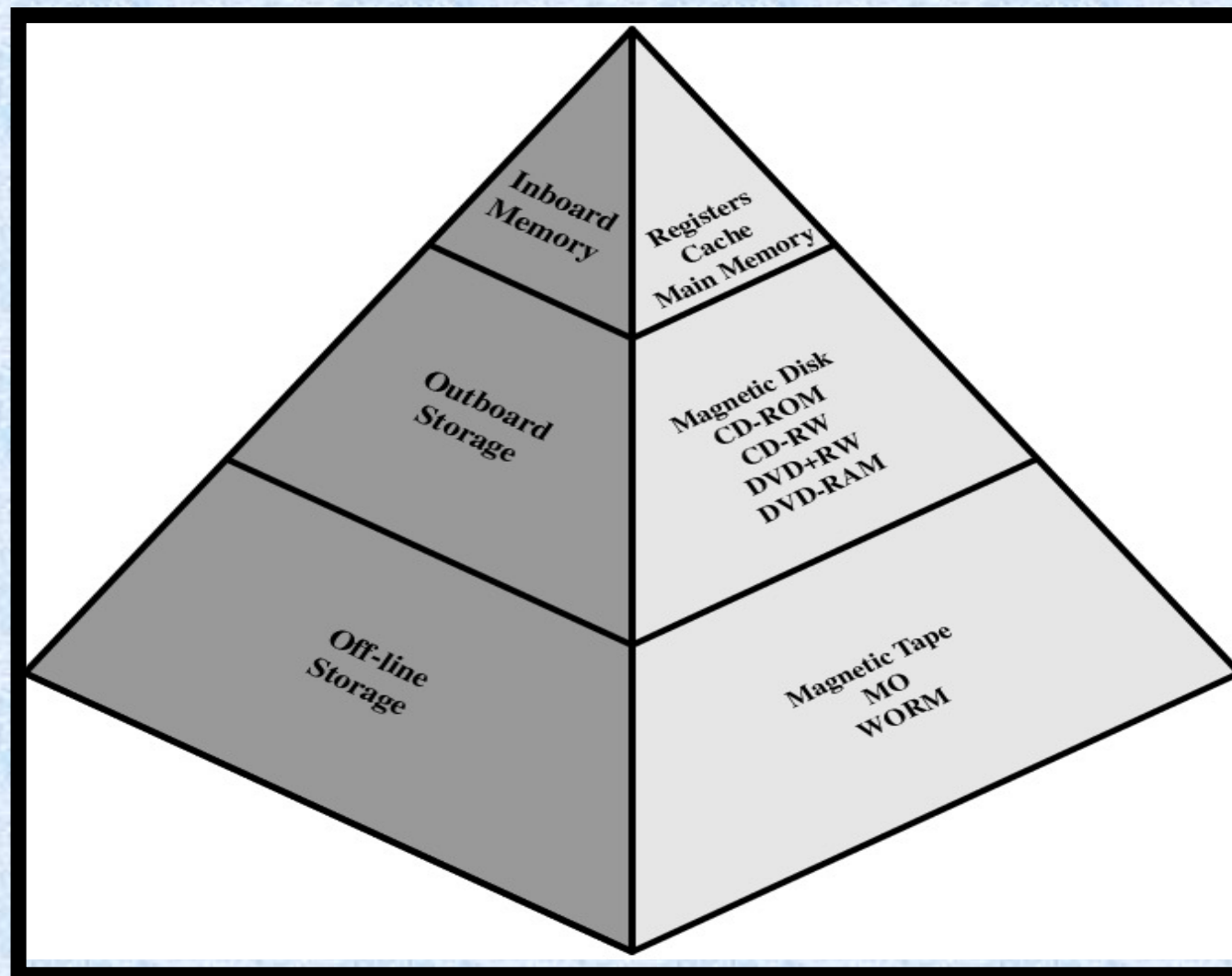


# Princípio da Localidade

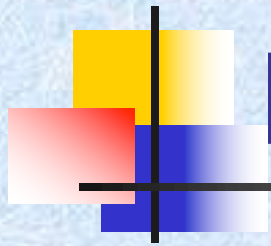
---

- Localidade Temporal
  - Itens acessados recentemente têm grande probabilidade de serem acessados em um futuro próximo
- Localidade Espacial
  - Itens cujos endereços estão próximos tendem a ser acessados em tempos próximos

# Hierarquia de Memória (Stallings)

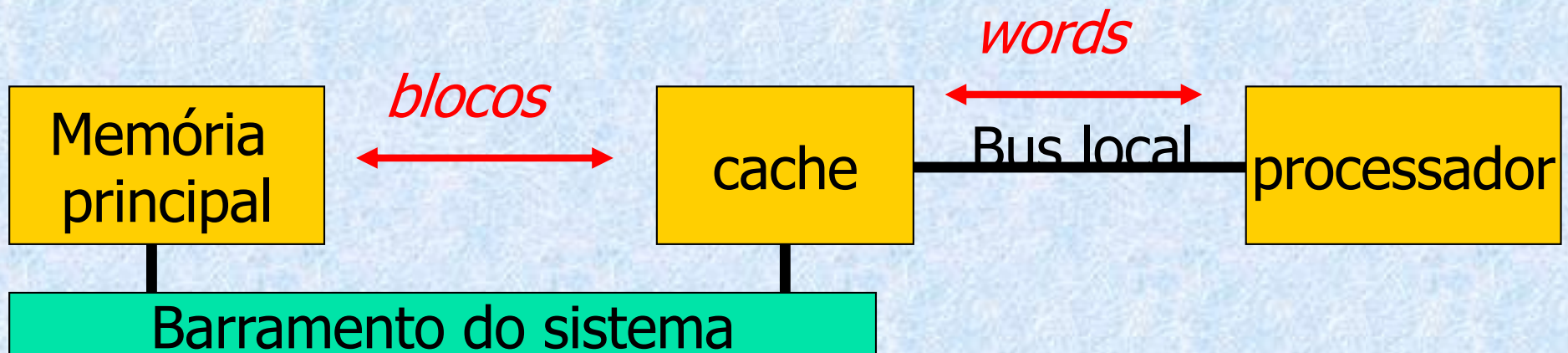


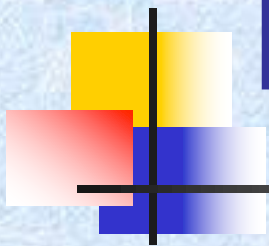




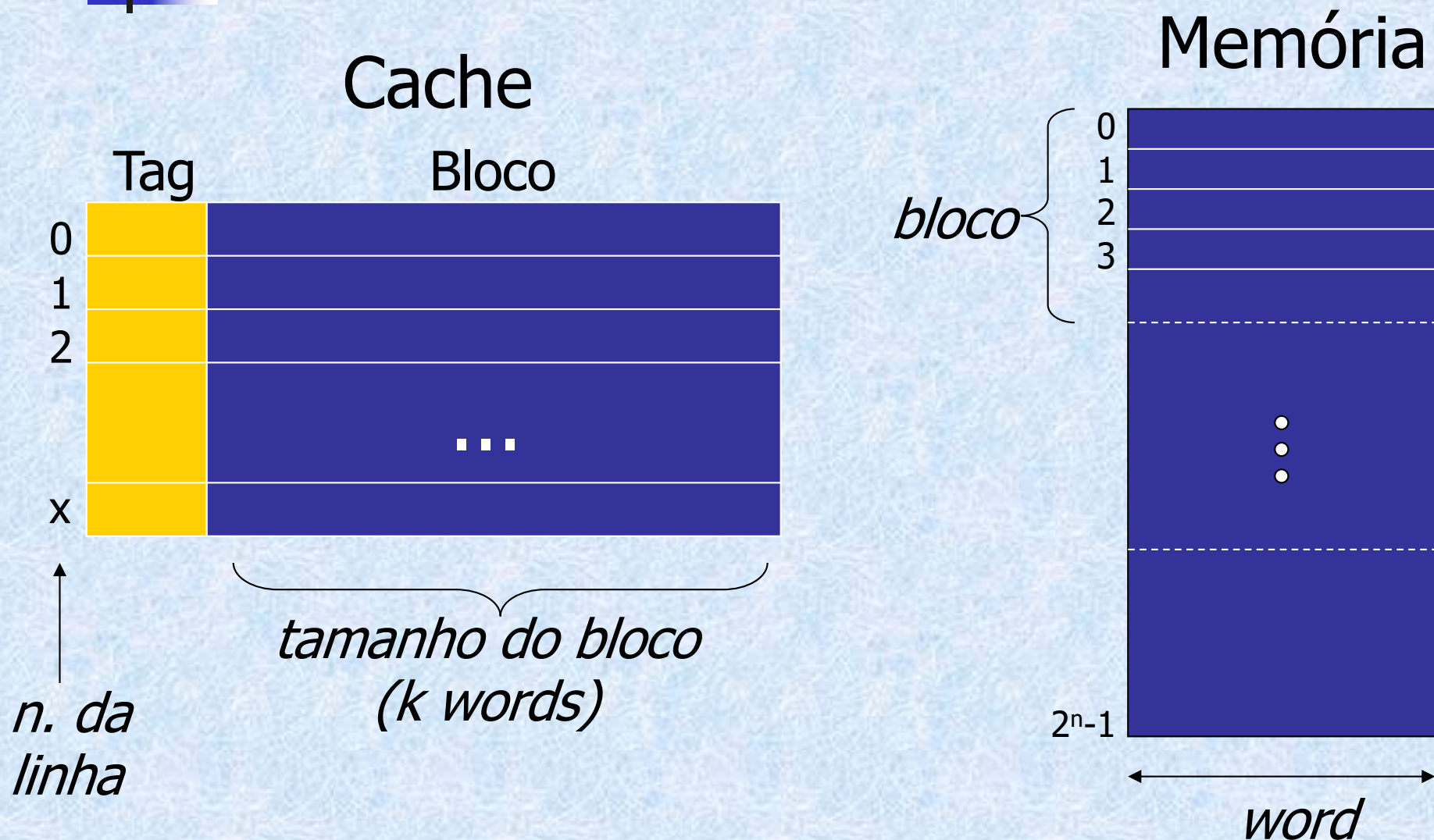
# Memória Cache

- É uma memória rápida e com pouca capacidade de armazenamento que é interposta entre a CPU e a memória principal.

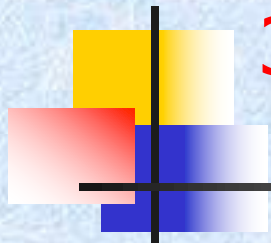




# Estrutura da Memória Cache



Exemplo: Memória Principal de 8MB (8388608), blocos de 8 words, words de 1 byte e cache de 32K linhas. Acesso ao endereço 524288 (512K)



## Memória Principal

0	
524288	a
524289	b
524290	c
524291	d
524292	e
524293	f
524294	g
524295	h
8388607	

word

## Cache

	Tag	Bloco							
		0	1	2	3	4	5	6	7
0	2	a	b	c	d	e	f	g	h
1									
2									
32767									

n. da linha      tamanho do bloco (8 words)

Tag:  $512K/32K/8 = 2$

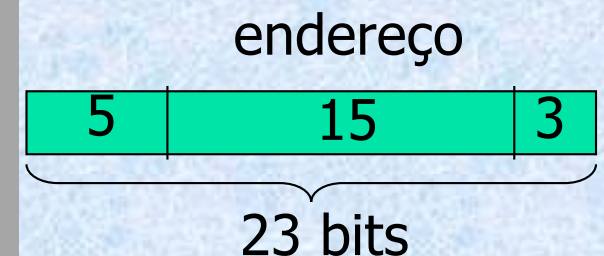
Acesso:

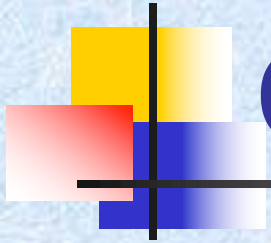
$524288/8 = 65536$

$65536/32768 = 2$  (tag)

Resto = 0 (linha)

Resto  $524288/8 = 0$  (word)



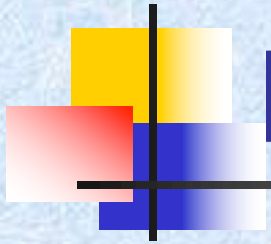


# Colocação do Bloco em Cache

---

- Para determinar onde colocar um bloco de memória em cache, temos as seguintes abordagens:
  - Mapeamento Direto
  - Set Associative
  - Fully Associative





# Mapeamento Direto

---

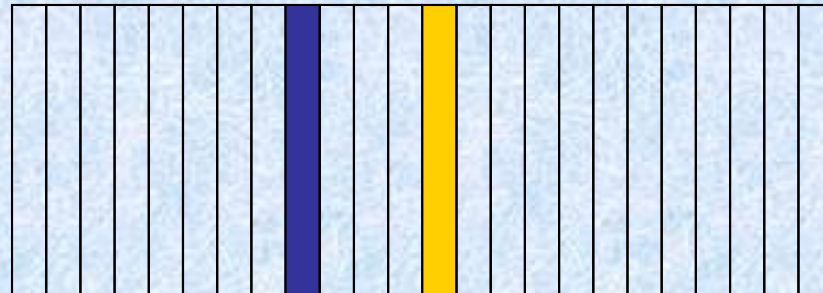
- Neste caso, o bloco só pode ser colocado em um lugar na cache.
- O local é geralmente calculado por  $ba \text{ MOD } n$ , onde  $ba$  é o endereço do bloco e  $n$  é o número de blocos em cache.

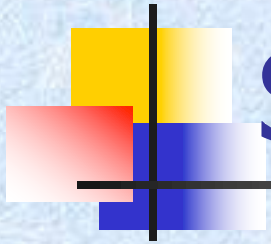


0 1 2 3 4 5 6 7



0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3

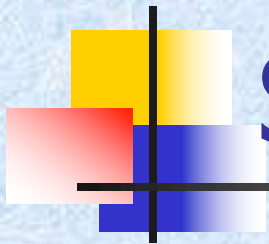




# Set Associative

---

- Neste caso, o bloco pode ser colocado em um conjunto restrito de posições.
- Um conjunto (set) é um grupo de blocos em cache.
- Inicialmente, o bloco é mapeado para um conjunto. Dentro deste conjunto, ele pode ser colocado em qualquer posição.



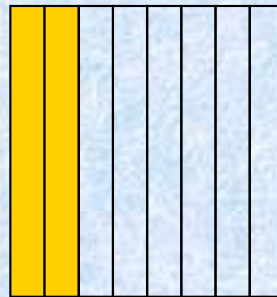
# Set Associative

---

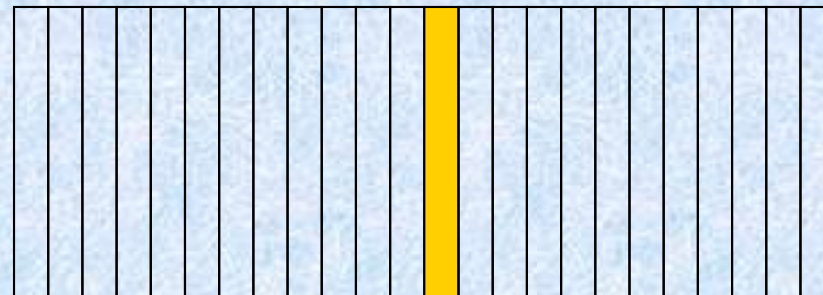
- O local é geralmente calculado por  $ba \text{ MOD } k$ , onde  $ba$  é o endereço do bloco e  $k$  é o número de conjuntos em cache.
- Se há  $m$  blocos em cada conjunto, a colocação em cache (cache placement) é chamada *m-way* set associative.



0 1 2 3 4 5 6 7

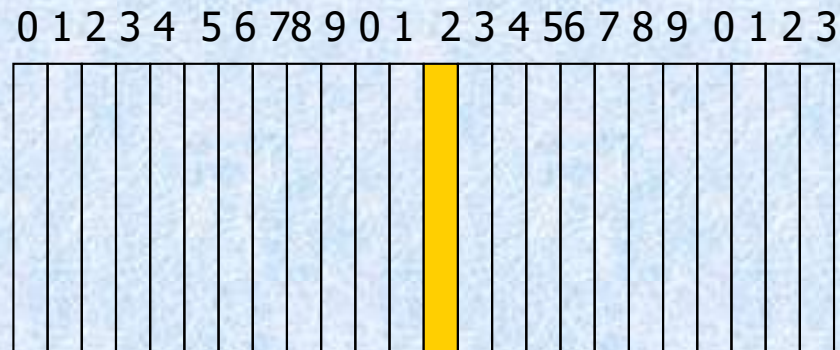


0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3





- # Cache

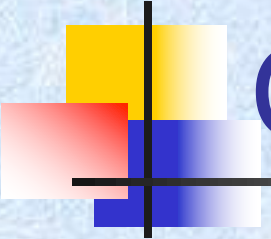


# Localização do Bloco em Cache

- Formato de um endereço do ponto de vista da cache:



- O índice determina o conjunto ao qual o bloco pertence
- O tag do endereço é comparado com o tag que está em cache. No caso de um hit, o deslocamento é utilizado para determinar a palavra ou byte desejado.

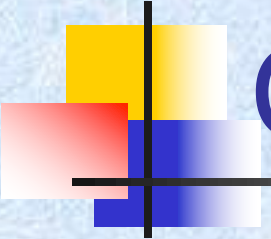


# Substituição de Blocos em Cache

---

- Caso haja um cache miss, o controlador da cache deve selecionar um bloco para ser substituído.
- Em caches de mapeamento direto, não há escolha pois somente um bloco pode ser substituído.



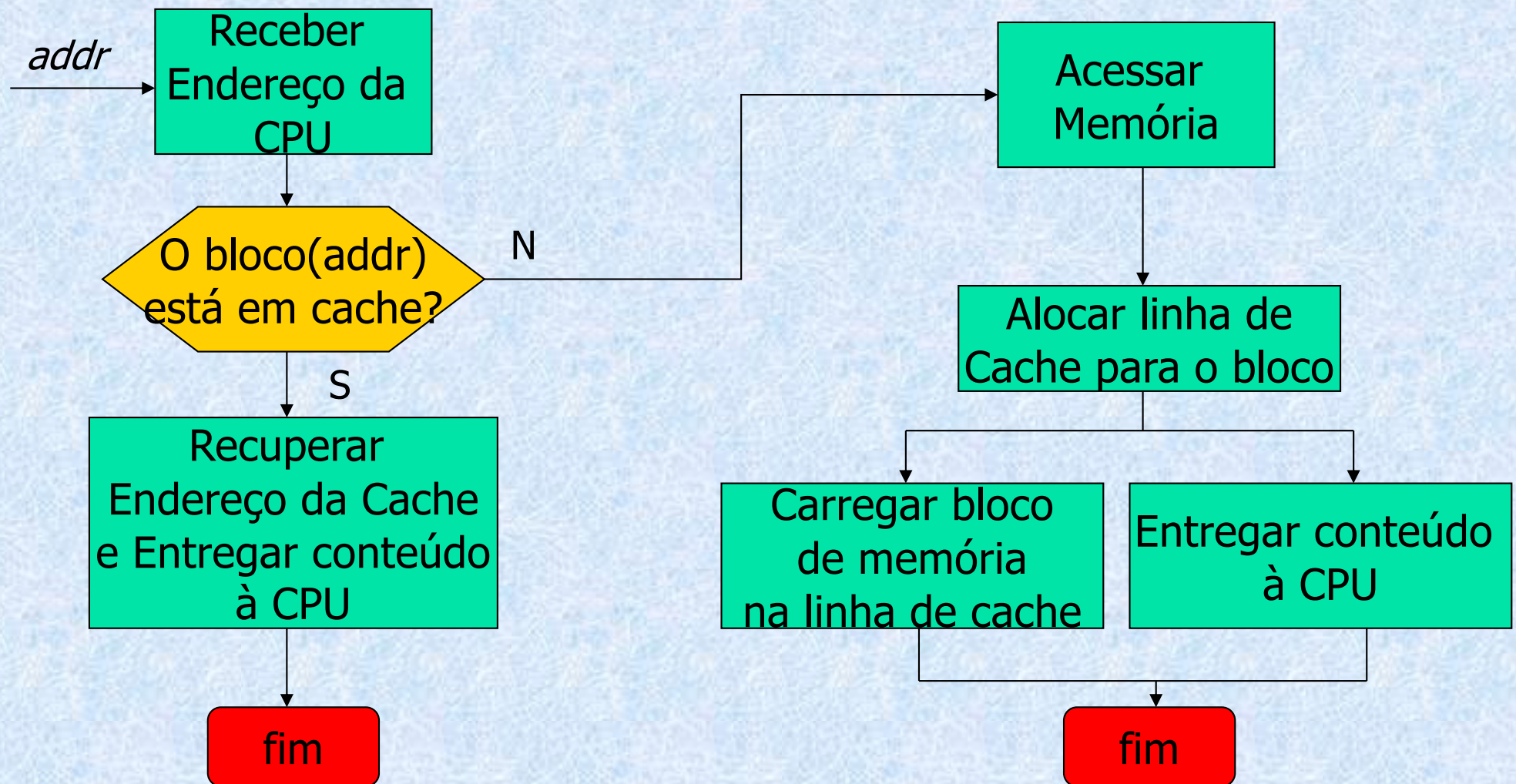


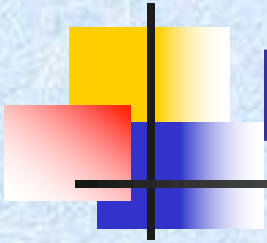
# Substituição de Blocos em Cache

---

- Nos esquemas associativos, podemos utilizar as seguintes abordagens:
  - *Random*: de maneira a conseguir uniformidade, os blocos a serem substituídos são selecionados aleatoriamente.
  - *LRU*: substitui os blocos menos recentemente utilizados. O custo de implementação do LRU é alto e geralmente são implementadas aproximações.

# Leitura de Blocos em Cache

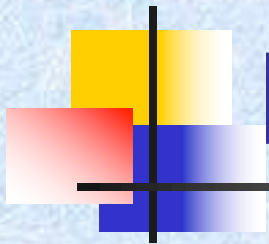




# Leitura de Blocos em Cache

---

- A leitura do bloco começa quando o endereço está disponível, já que um bloco pode ser lido ao mesmo tempo em que o tag é lido e comparado (memória associativa).
- Se o read é um hit, o dado é repassado a CPU imediatamente.
- Se o read é um miss, a leitura prévia é ignorada e o bloco é carregado da memória

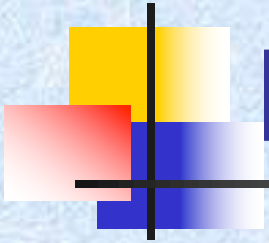


# Escrita de Blocos em Cache

---

- Um bloco não pode ser alterado antes de se ter certeza que o tag é o desejado.
- Por esta razão, os hits de escritas levam mais tempo que os hits de leituras.
- Os processadores também tem que informar o número de bytes a serem escritos e somente estas posições serão modificadas.
- No caso das leituras, o acesso a mais dados do que o solicitado não causa problema.

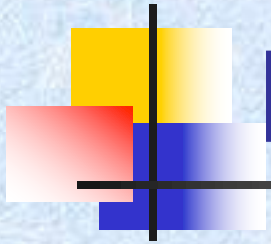




# Escrita de Blocos em Cache

---

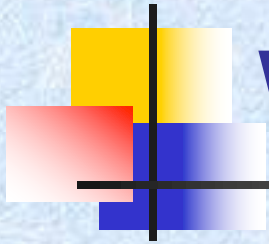
- Existem basicamente duas políticas de escrita em cache:
  - *Write through* – o dado é escrito tanto no bloco em cache como na memória.
  - *Write back* – O dado é somente escrito em cache. Somente no momento da substituição, o bloco modificado é escrito em memória



# Dirty Bits

---

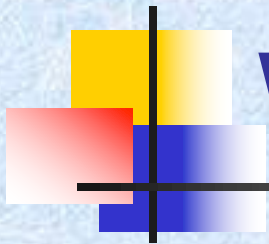
- Indicam se o bloco foi modificado enquanto estava em cache (dirty) ou não (clean).
- Os blocos clean não são escritos em memória na substituição.



# Write-back x Write-through

---

- As escritas são mais rápidas com write-back, pois elas ocorrem na velocidade da cache.
- Menor largura de banda é consumida com write back, devido à menor frequência de acesso à memória.
- Com write through, os read misses nunca ocasionam escritas para a memória.
- Além disso, políticas write-through são mais simples de se implementar.

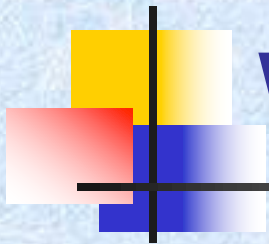


# Write stalls

---

- Ocorrem quando a CPU tem que esperar que o dado seja escrito em memória (write-through).
- Para reduzir este tempo, são frequentemente usados write-buffers.
- A CPU pode continuar após escrever o dado no write-buffer e, de maneira assíncrona, o dado é transferido para a memória

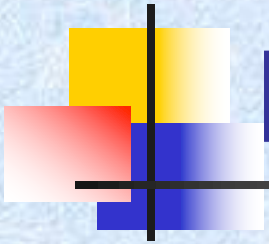




# Write Misses

---

- Já que o dado antigo não é necessário na escrita, podem haver duas abordagens para um write miss:
  - Write allocate: o bloco é carregado em cache no momento do write miss, seguido pelas ações do write hit.
  - No-write allocate: o bloco é modificado em memória e não é carregado em cache
- Tradicionalmente, políticas write-back usam write-allocate e write-through usam no-write allocate

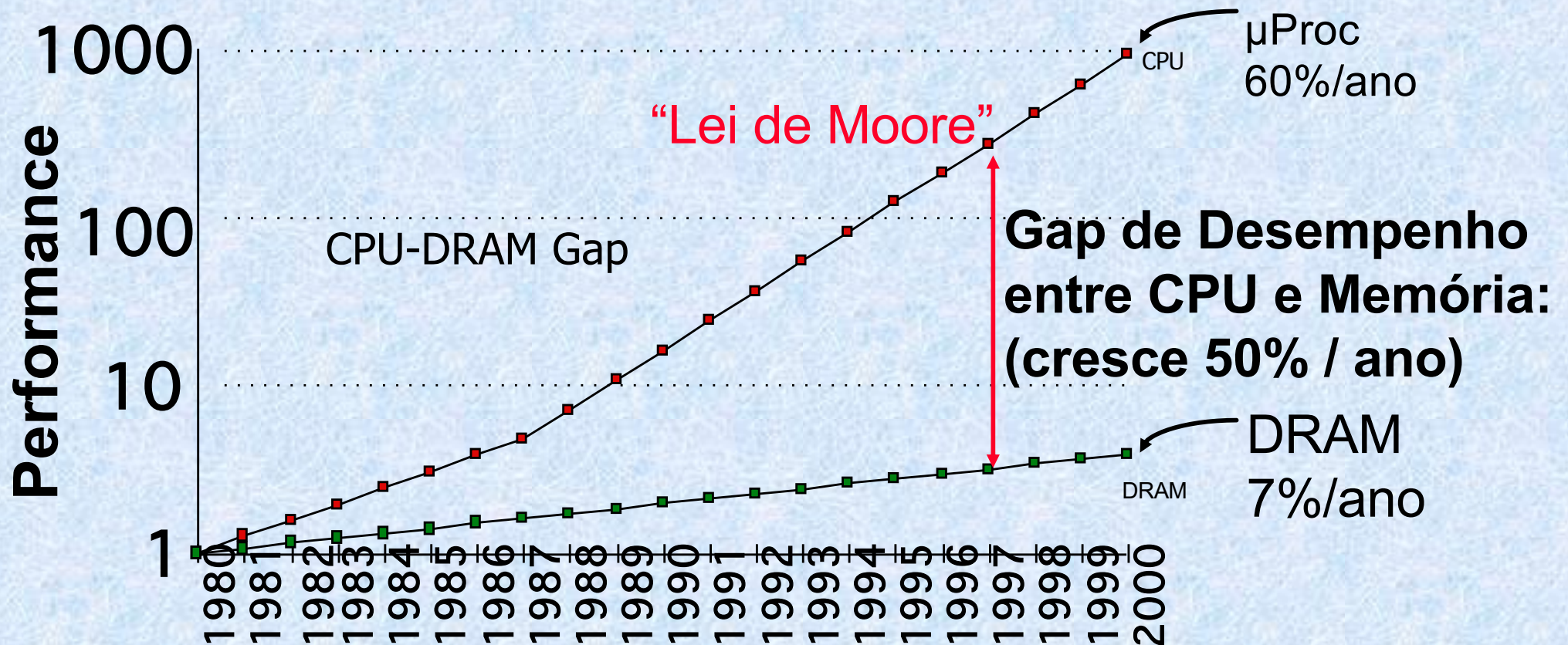
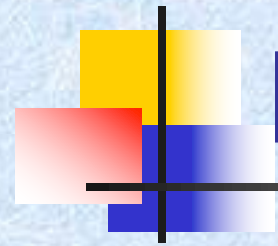


# Medidas de Desempenho

---

- Tempo médio de acesso à memória
  - $A_{mat} = \text{hit time} + \text{miss rate} * \text{miss penalty}$ 
    - Hit time: tempo gasto em um hit em cache
    - Miss rate: taxa de misses na cache
    - Miss penalty: penalidade associada a cada miss
  - O tempo  $A_{mat}$  pode ser medido em frações de segundos ou ciclos de clock

# Desempenho da Memória Principal x CPU (Patterson)

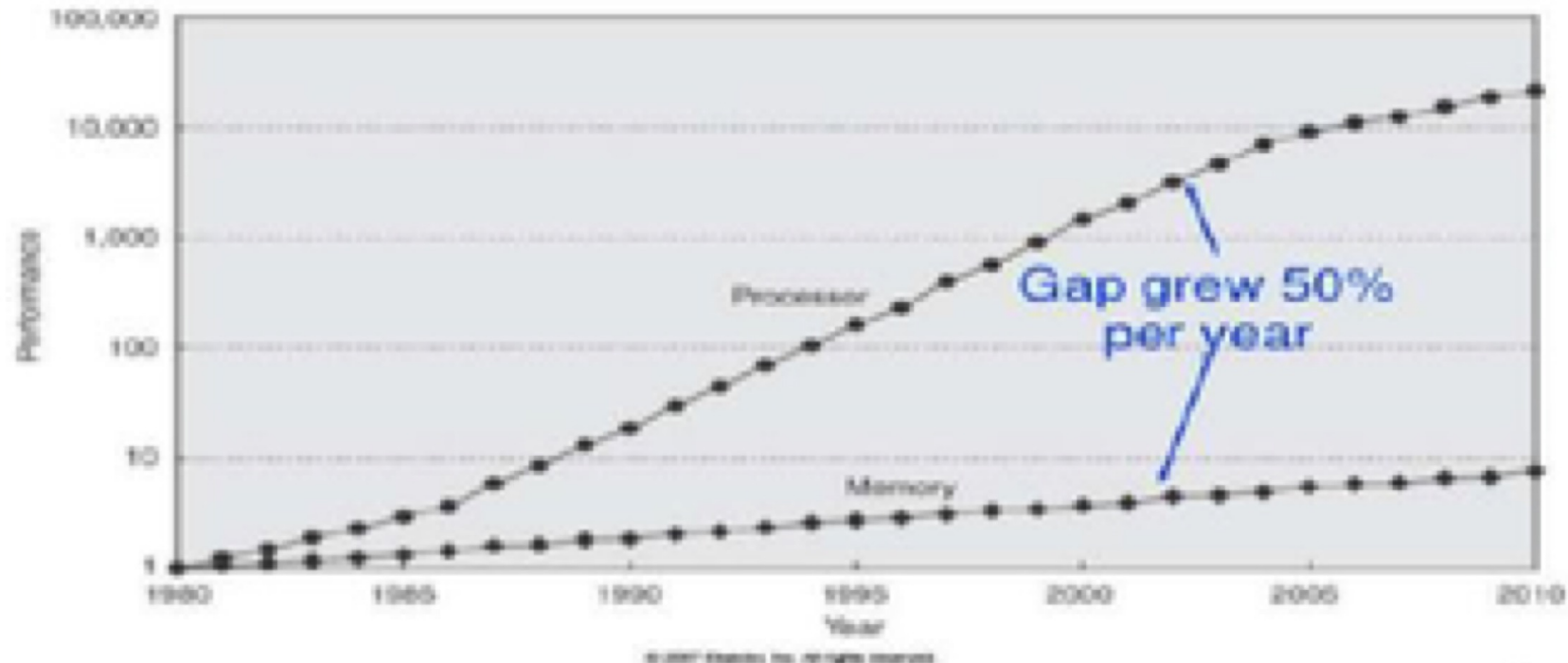


1980: sem caches; 1995 cache de dois níveis



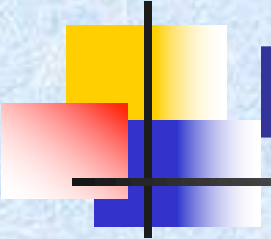
# Desempenho da Memória Principal x CPU (Patterson)

## Processor Memory Gap



Source: Computer Architecture, A Quantitative Approach by John L. Hennessy and David A. Patterson

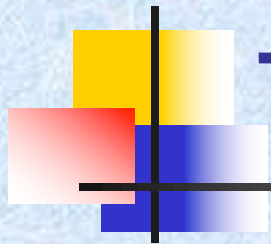




# Técnicas para Melhorar o Desempenho das Caches

---

- As caches foram introduzidas para reduzir o “gap” entre os tempos de acesso à memória principal e a velocidade do processador.
- O ideal de um projeto de cache seria obter hits rápidos e poucos misses.
- As principais técnicas para melhorar o desempenho das caches são:
  - Reduzir a taxa de cache misses
  - Reduzir a penalidade do cache miss
  - Reduzir o tempo de hit

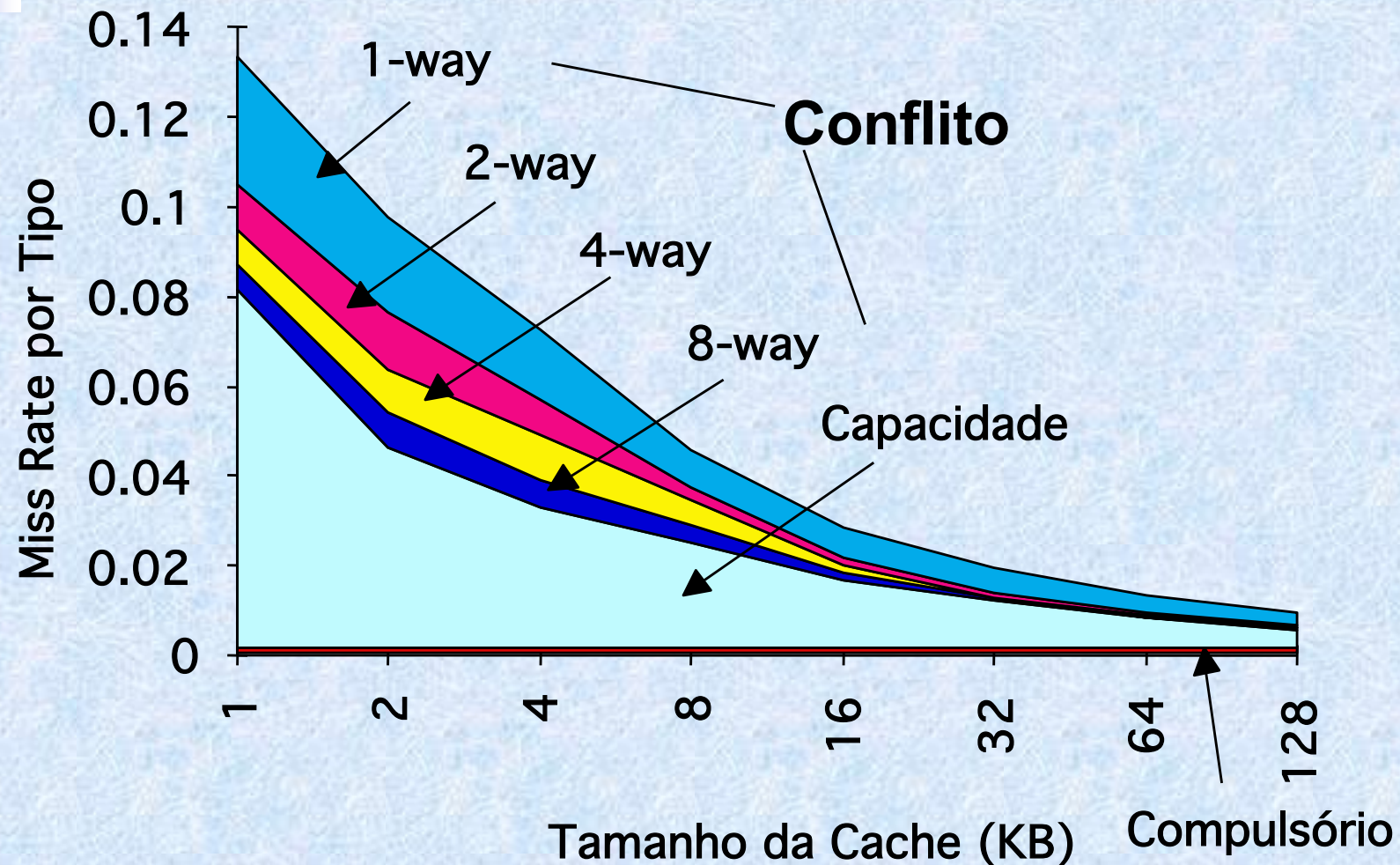


# Tipos de cache misses

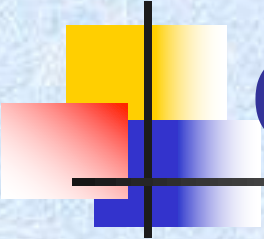
---

- *Compulsórios (cold misses ou first reference misses)*: no primeiro acesso ao bloco, o mesmo não estará em cache e, assim, deve ser trazido para a memória.
- *Capacidade*: ocorrem quando um bloco escolhido para substituição é acessado novamente.
- *Conflito (collision misses ou interference misses)*: ocorrem em caches com mapeamento direto ou associativas, se muitos dos blocos referenciados são mapeados para o mesmo conjunto.

# Taxas de Misses por Tipo de Acesso (SPEC 92) - Patterson





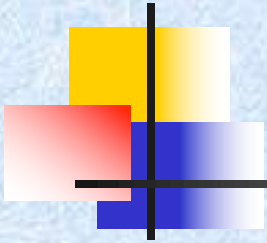


# Técnicas para Reduzir a Taxa de Cache Misses

---

1. Aumentar o tamanho do bloco
2. Aumentar a associatividade
3. Victim Caches
4. Caches Pseudo-associativas
5. Pré-carga por hardware de instruções e dados
6. Pré-carga controlada por compilador
7. Otimizações do compilador

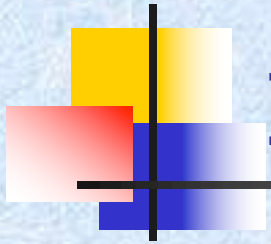




# 1. Aumentar o Tamanho do Bloco

---

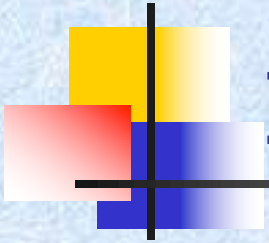
- Esta técnica é motivada pelo fato de que, com blocos de tamanho maior, o número de misses compulsórios tende a ser menor.
  - Blocos maiores tiram grande proveito da localidade temporal
- No entanto, blocos grandes podem aumentar o número de misses de conflito e de capacidade.
  - Blocos grandes aumentam a penalidade do miss



# 1. Aumentar o Tamanho do Bloco

## Miss Rate x Tamanho da Cache (Patterson)

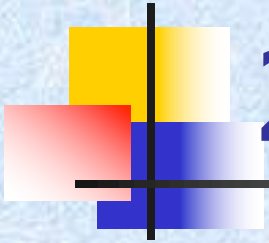
Tamanho do Bloco	Tamanho da Cache				
	1K	4K	16K	64K	256K
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%



# 1. Aumentar o Tamanho do Bloco

---

- A escolha do tamanho do bloco depende de dois fatores:
  - Latência da memória (tempo de acesso à memória)
  - Largura de banda da memória
- Em geral, se a largura de banda e a latência da memória são grandes, blocos grandes são escolhidos.

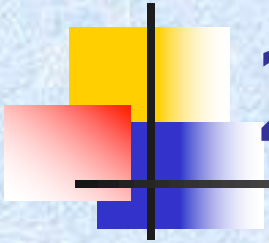


## 2. Aumentar a Associatividade

---

- A utilização desta técnica justifica-se porque, com maior associatividade, há mais blocos que potencialmente podem ser escolhidos para substituição, possibilitando o uso de algoritmos mais elaborados.

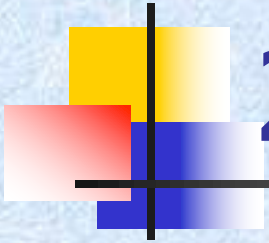




## 2. Aumentar a Associatividade

---

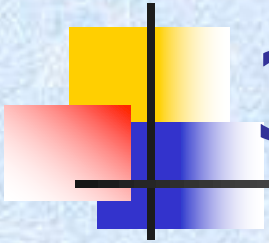
- Para se utilizar esta técnica, deve-se observar duas regras gerais obtidas empiricamente:
  - Uma cache 8-way set associative reduz a taxa de misses tanto quanto uma cache totalmente associativa (até 2010)
  - Uma cache de mapeamento direto de tamanho  $N$  apresenta aproximadamente o mesmo miss rate que uma cache 2-way de tamanho  $N/2$ .



## 2. Aumentar a Associatividade

---

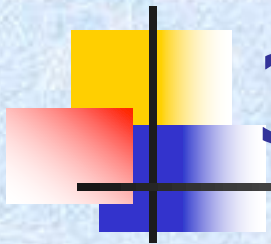
- Deve-se ter em mente que, ao aumentar a associatividade, o projeto da cache fica mais complexo e, portanto, o tempo de hit fica maior.
- Sendo assim, pode ser que a taxa de misses seja reduzida porém, como o tempo de hit é maior, o tempo médio de acesso à memória aumente.



### 3. Caches Vítimas

---

- A cache vítima é uma pequena cache totalmente associativa que é adicionada entre a cache e o nível de memória imediatamente inferior.
- A cache vítima contém somente blocos que foram substituídos da cache por causa de um miss.

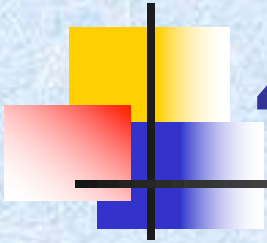


## 3. Caches Vítimas

---

- Na ocorrência de um miss, verifica-se se o bloco encontra-se na cache vítima.
- Caso afirmativo, o bloco vítima substitui um bloco da cache
- Alguns estudos mostram que caches vítimas de poucos blocos (4) conseguem reduzir o número de misses de conflito de caches pequenas de mapeamento direto.

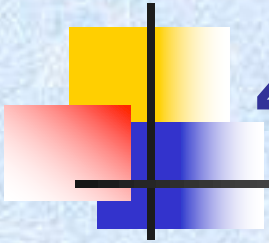




## 4. Caches Pseudo-Associativas

---

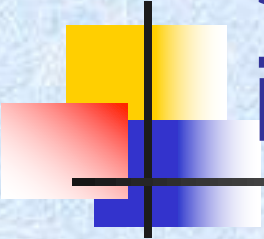
- Caches pseudo-associativas ou associativas por coluna tentam obter a taxa de misses da cache associativa e o tempo de hit da cache de mapeamento direto.
- No caso de hit, o acesso a esta cache funciona como o acesso a cache de mapeamento direto.
- Em um miss, o “pseudo set” é obtido pela inversão dos bits mais significativos do índice e a entrada da cache obtida desta maneira é verificada.



## 4. Caches Pseudo-Associativas

---

- Estas caches possuem um tempo de hit rápido e um tempo de hit lento.
- Tempos de hit variáveis podem complicar o projeto de pipelines.
- Por esta razão, esta técnica é mais adequada a caches de segundo nível.

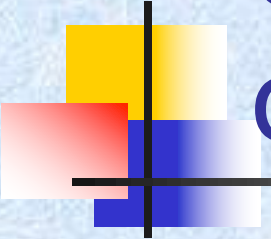


## 5- Pré-carga por hardware de instruções e dados

---

- Intuitivamente, a taxa de misses pode ser reduzida se trouxermos os dados e instruções para cache antes dos mesmos terem sido referenciados.
- Uma técnica muito comum consiste em se trazer dois blocos a cada miss: o bloco referenciado e o próximo bloco.
- O bloco referenciado é colocado na cache e o próximo bloco é colocado em um “stream buffer”.



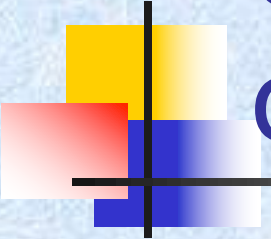


## 6- Pré-carga com auxílio do compilador

---

- Nesta abordagem, o compilador insere instruções de pré-carga no código do programa.
- Em geral, deve-se garantir que o dado ou instrução pré-carregados não causarão exceções de memória virtual (reais ou de proteção).
- Por esta razão, as pré-cargas mais efetivas são as que não causam alterações programa
  - Non-binding prefetching

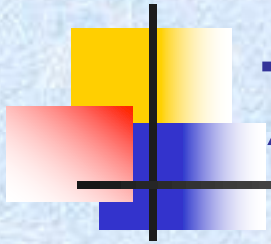




## 6- Pré-carga com auxílio do compilador

---

- A pré-carga só faz sentido se o processador puder continuar a operação enquanto o dado a ser pré-carregado não chega.
  - As caches devem ser capazes de tratar diversas requisições sem se bloquear: lockup-free caches.
- A execução de instruções de pré-carga adiciona um overhead à execução. Deve-se então tomar cuidado para que este overhead não seja maior que o ganho obtido.



## 7 – Otimizações do Compilador

---

- Esta técnica não utiliza hardware adicional.
- Pode-se utilizar técnicas de profiling para rearrumar o código, reduzindo as taxas de misses das instruções.
- As taxas de misses de dados podem ser reduzidas pelas seguintes técnicas:
  - Merging de arrays
  - Trocas de loops
  - Fusão de loops
  - Blocagem



# 7 – Otimizações do Compilador

## Merging de Arrays

---

- Alguns programas referenciam múltiplos arrays com o mesmo índice no mesmo loop e isso pode levar a misses de conflito:

```
int val[TAM];  
int chave[TAM];  
...  
for (i=0; i<MAX; i++)  
    a = a + val[i] + chave[i];
```



# 7 – Otimizações do Compilador

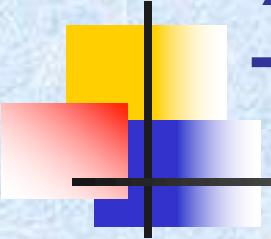
## Merging de Arrays

---

- O compilador pode rearrumar o código de maneira que os dados referenciados estejam no mesmo bloco ou em blocos próximos na cache:

```
struct merge {  
    int val;  
    int chave;  
};  
struct merge array_merge[TAM];  
...  
for (i=0; i<MAX; i++)  
    a = a + array_merge[i].val + array_merge[i].chave;
```





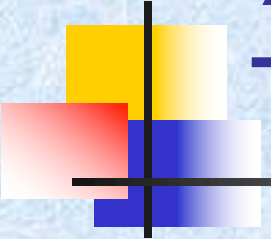
# 7 – Otimizações do Compilador

## Troca de loops

---

- Alguns programas possuem código com loops aninhados que não acessam dados de maneira sequencial.
- Exemplo (armazenamento de matrizes por linha):

```
for (j=0; j<100; j++)  
    for (i=0; i<5000; i++)  
        a[i][j] = a[i][j] * 2;
```



# 7 – Otimizações do Compilador

## Troca de loops

---

- O compilador pode alterar o código de maneira que o acesso seja sequencial.

```
for (i=0; i<5000; i++)  
    for (j=0; j<100; j++)  
        a[i][j] = a[i][j] * 2;
```



# 7 – Otimizações do Compilador

## Junção de loops

---

- Alguns programas acessam as mesmas matrizes com os mesmos loops porém fazendo cálculos diferentes:

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        a[i][j] = b[i][j] + c[i][j];  
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        d[i][j] = b[i][j] * c[i][j];
```



# 7 – Otimizações do Compilador

## Junção de loops

---

- Na junção de loops, tira-se proveito da localidade temporal, fazendo que os dados sejam acessados diversas vezes enquanto estão em cache:

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
    {  
        a[i][j] = b[i][j] + c[i][j];  
        d[i][j] = b[i][j] * c[i][j];  
    }
```





# 7 – Otimizações do Compilador

## Blocagem

---

- Existem alguns casos onde diversas matrizes são acessadas, umas por linha e umas por coluna.
- Nestes casos, uma otimização mais complexa deve ser utilizada.
- Consiste em se operar em submatrizes (ou blocos), ao invés de se operar em matrizes inteiras



# 7 – Otimizações do Compilador

## Blocagem

---

- Exemplo (código original):

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
    {  
        r = 0;  
        for (k=0; k<N; k++)  
            r = r + b[i][k] * c[k][j];  
        a[i][j] = r;  
    }
```

- *a* e *b*: percorrimento por linha
- *c*: percorrimento por coluna



# 7 – Otimizações do Compilador

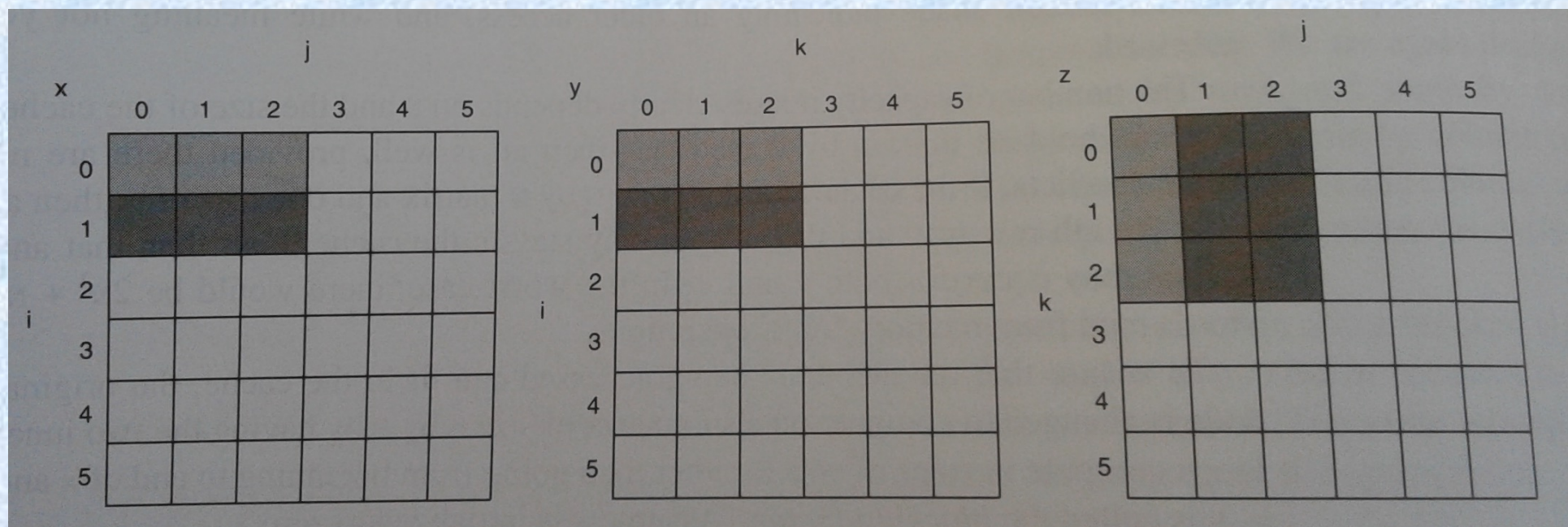
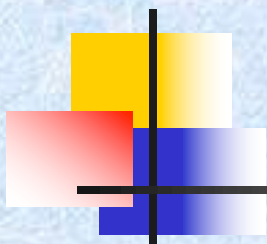
## Blocagem

---

- Exemplo (código alterado):

```
for (jj=0; jj<N; jj=jj+B)
  for (kk=0; kk<N; kk=kk+B)
    for (i=0; i<N; i++)
      for (j=jj; j<(min(jj+B-1,N); j++)
      {
        r = 0;
        for (k=kk; k<(min(kk+B-1,N); k++)
          r = r + b[i][k] * c[k][j];
        a[i][j] = r;
      }
```





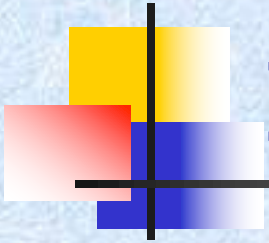




# Reduzir a Penalidade do Cache Miss

---

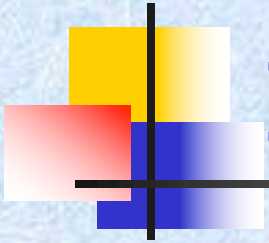
- O tempo médio de acesso à memória depende, além da taxa de cache misses, da penalidade associada a cada cache miss.
- Existem, assim, algumas técnicas que tentam reduzir esta penalidade:
  1. Priorizar misses de leitura
  2. Utilizar sub-blocos
  3. Early Restart e Palavra Crítica Primeiro
  4. Lockup-free caches
  5. Caches de segundo nível



# 1. Priorizar Misses de Leitura

---

- Para acelerar a conclusão da execução de instruções, geralmente “write buffers” são adicionados ao hardware.
- Assim, quando o store termina, geralmente o dado ainda se encontra no “write buffer” e não no bloco de cache associado.
- Esta decisão de projeto pode fazer com que, dependendo da ordem de acesso aos dados, valores antigos sejam lidos.



# 1. Priorizar Misses de Leitura

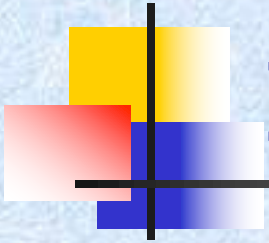
---

- Exemplo:

- (1) STORE #512, r3
  - (2) LOAD r1, #1024
  - (3) LOAD r2, #512

- Caso os endereços 512 e 1024 estejam mapeados no mesmo índice de cache e a escrita de (1) demorar a se completar, a instrução (3) pode ler o valor antigo da memória.

- Hazard de dados RAW

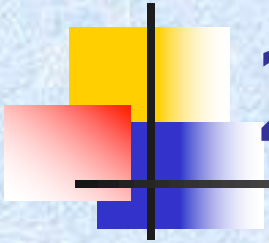


# 1. Priorizar Misses de Leitura

---

- A maneira mais simples de evitar que este problema aconteça consiste em fazer com que o read miss espere até que o “write buffer” esteja vazio.
  - Aumento da penalidade do read miss
- Para priorizar os misses de leitura, o hardware de muitos processadores verifica o conteúdo do “write buffer” e, caso não haja conflito, deixa o read miss continuar.

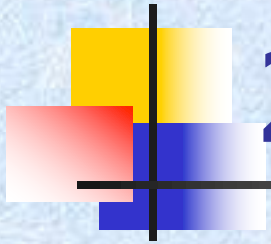




## 2. Utilizar Sub-Blocos

---

- Esta técnica consiste em se dividir cada bloco da cache em sub-blocos e adicionar um bit de validade a cada sub-bloco. O tag continua associado ao bloco
- Em um read miss, somente um sub-bloco é lido.
  - Redução da penalidade do miss



## 2. Utilizar Sub-Blocos

*Bloco*

100	1		1		1		1	
150	1		1		0		0	
060	0		1		0		1	
404	0		0		0		0	

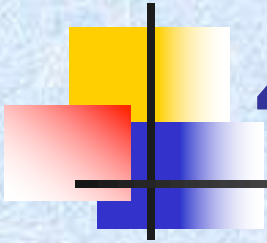
*Sub-bloco*



## 3. Early Restart e Palavra Crítica Primeiro

---

- Estas técnicas não esperam que o bloco inteiro seja colocado em cache, deixando a CPU continuar logo que a palavra desejada estiver carregada.
  - *Early Restart*: Carrega as palavras na ordem sequencial e, logo que a palavra desejada tiver sido carregada, a CPU continua a execução
  - *Palavra Crítica Primeiro*: solicita a palavra que causou o miss primeiro e permite que a CPU continue a execução logo que a mesma chegar.

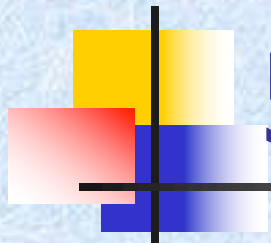


## 4. Lockup-free Caches

---

- Uma cache lockup-free ou non-blocking cache é capaz de fornecer dados que estão na cache mesmo durante um cache miss (hit under miss).
- Caches lockup-free mais complexas permitem o tratamento simultâneo de vários misses (miss under miss), se a memória for capaz de tratar diversos misses.
- O projeto de lockup-free caches é bastante complexo, pois deve lidar com diversos acessos incompletos.

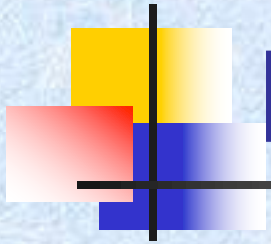




## 5. Caches de Segundo Nível

---

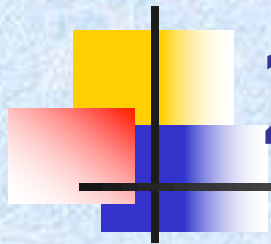
- Devido à grande diferença de desempenho entre a memória RAM e a CPU, a inclusão de uma cache mais lenta e com maior capacidade entre a cache tradicional e a memória é uma alternativa bastante interessante.
- As caches de segundo nível (L2), para serem efetivas, devem ser bem maiores que a cache de primeiro nível (L1).
- Geralmente, a propriedade de inclusão multinível é observada porém, caso se opte por tamanhos de blocos distintos, o projeto da hierarquia de memória fica complexo.



# Reduzir o tempo de hit

---

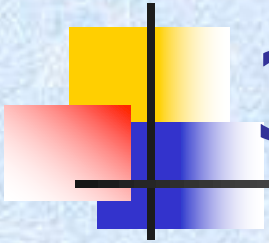
- O tempo de hit é crucial pois limita a taxa de clock do processador.
- Principais técnicas:
  - Caches simples e pequenas
  - Virtual Caches
  - Pipelining de escritas



## 2. Virtual Caches

---

- Geralmente, a CPU lida com endereços virtuais que devem ser convertidos para endereços físicos.
- As caches tradicionais utilizam endereços físicos.
- As caches virtuais utilizam endereços virtuais, não fazendo a tradução de endereços em um cache hit.
- No entanto, a cada troca de contexto, a cache deve ser esvaziada (flushed), pois cada processo possui seu próprio espaço de endereçamento virtual.



### 3. Pipelining de Escritas

---

- O pipelining de escritas ocorre entre escritas distintas.
- Para que o dado seja escrito, deve ser feita primeiro a comparação com o tag e depois a escrita é feita no bloco correto.
- A comparação com o tag é feita no estágio 1 e a escrita dos dados é feita no estágio 2

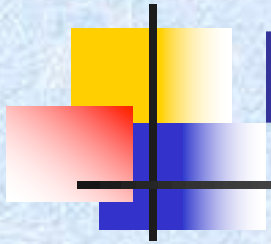




# Fundamentos de Sistemas Computacionais

---

Revisão – Memória RAM



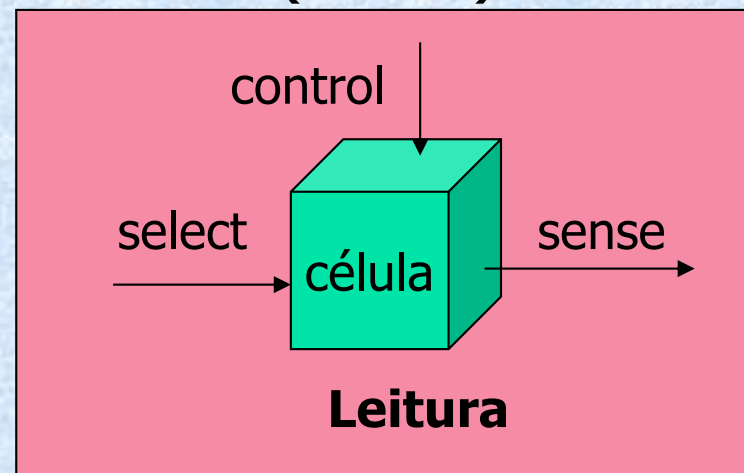
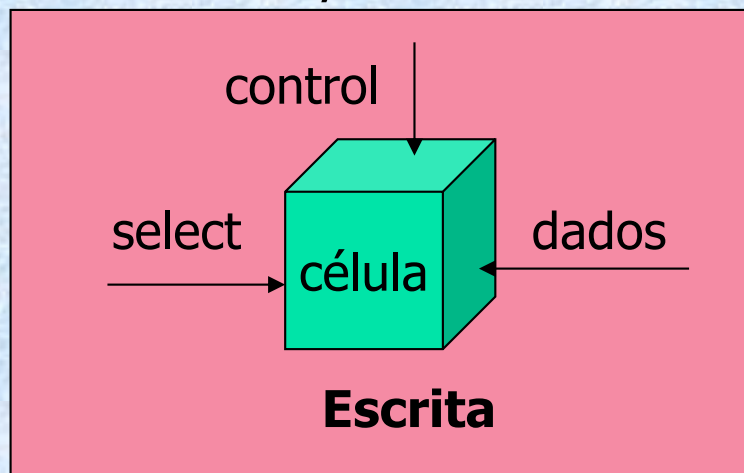
# Memórias Semicondutoras

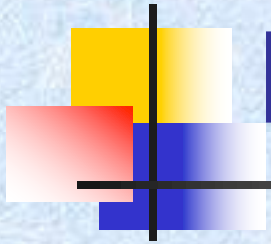
---

- A memória principal utiliza geralmente chips semicondutores e por isso é chamada memória semicondutora.
- O elemento básico da memória semicondutora é a célula
- As células possuem dois estados semiestáveis (0 e 1) e podem ser escritas, setando o estado, ou lidas, sentindo o estado.

# Operação de uma célula de memória

- Primeiramente, uma célula é selecionada para leitura ou gravação (select).
- O tipo de operação é indicado em controle.
- Na escrita, um sinal elétrico seta o estado da célula para 0 ou 1.
- Na leitura, o valor da célula é sentido (sense).



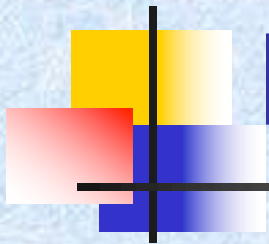


# Memória RAM

---

- A memória RAM é um tipo de memória semicondutora.
- RAM – Random Access Memory
- Estas memórias são voláteis, ou seja, necessitam de um fornecimento constante de energia para manter os valores

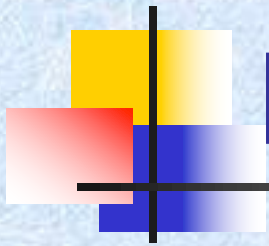




# Memória DRAM

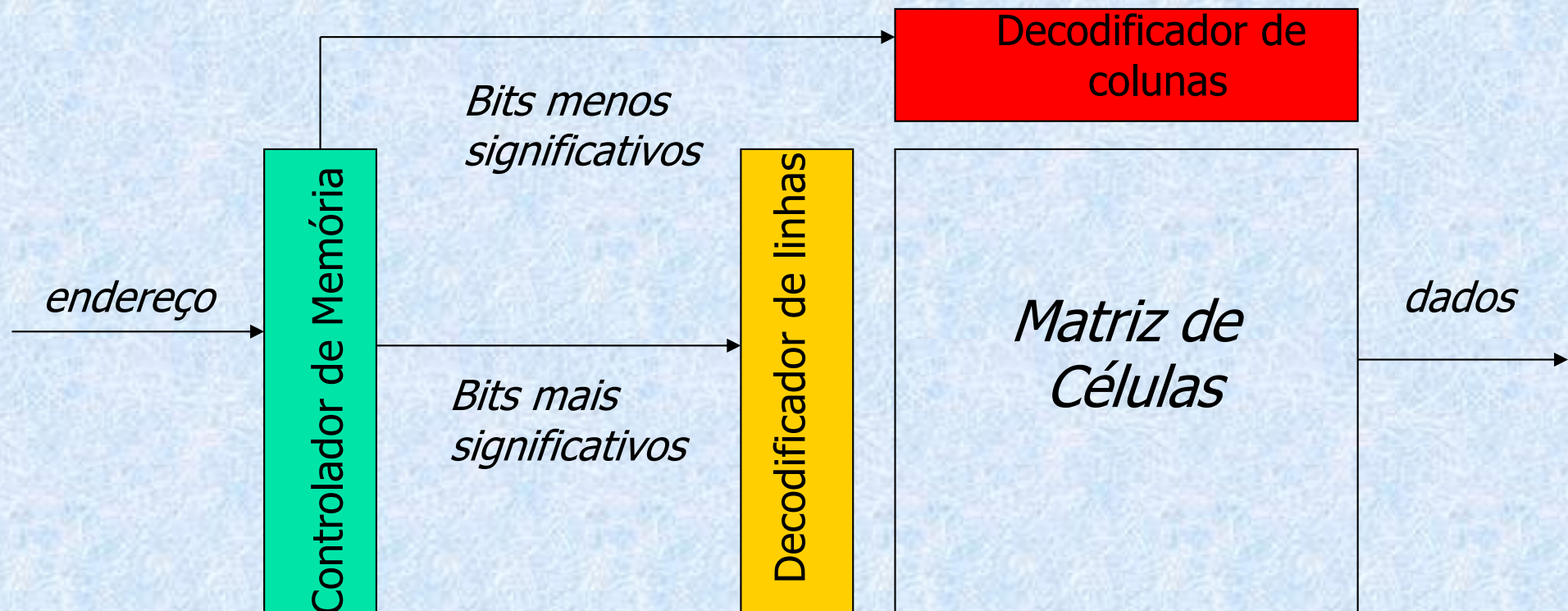
---

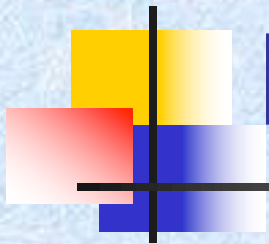
- A memória DRAM (Dynamic RAM) é composta de células que armazenam dados.
- Um transistor é utilizado para armazenar um bit.
- A memória DRAM é organizada como uma matriz retangular endereçada por linhas e colunas.
- O endereço é dividido em duas metades: RAS (row access strobe) e CAS (column access strobe).
- O protocolo de acesso à memória é assíncrono.



# Memória DRAM

- Esquema de acesso à memória

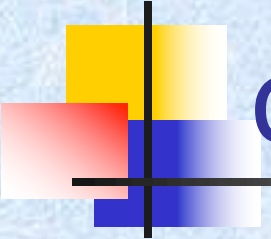




# Memórias DRAM

---

- As leituras podem danificar a informação contida nas células.
- Por esta razão, as DRAMs necessitam de refreshs periódicos para manter os dados.
- Todos os bits de uma linha podem ser “refreshed” pela leitura da linha.
- Durante os refreshs, a memória não está disponível.
- Utilizadas geralmente em memória principal.



# Técnicas para melhorar o desempenho das memórias RAM

---

- As técnicas geralmente se concentram em aumentar a largura de banda da memória:
  1. Aumentar o tamanho da palavra
  2. Memória Entrelaçada
  3. Bancos de Memória Independentes





# 1. Aumentar o Tamanho da transferência CPU/Memória

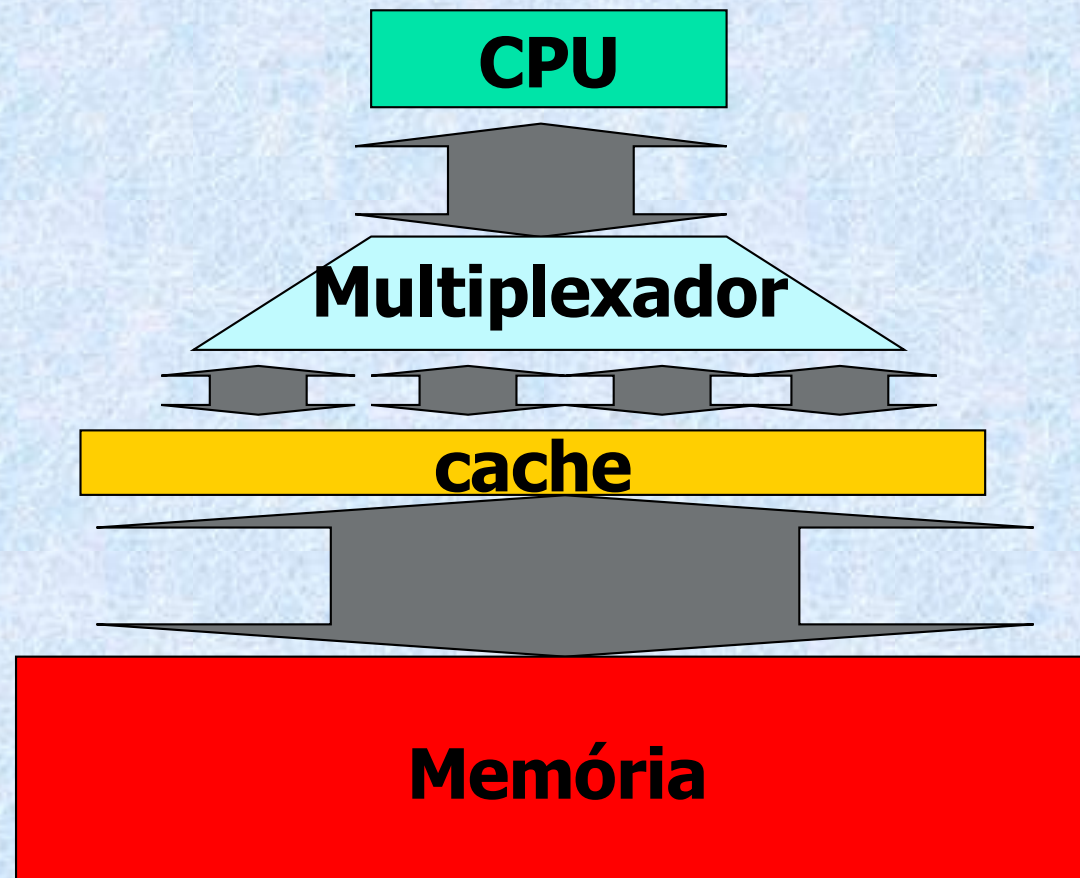
---

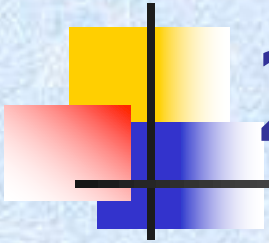
- Consiste em aumentar a largura de banda do barramento, transferindo assim mais dados em cada acesso.
- Como a CPU acessa ainda uma palavra, é necessário um multiplexador entre CPU e cache.
- O Alpha AXP usa barramentos de 256 bits para transferências entre memória RAM e cache L2.



# 1. Aumentar o Tamanho da transferência CPU/Memória (Patterson)

---

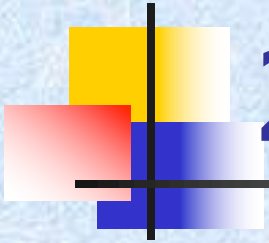




## 2. Memória Entrelaçada

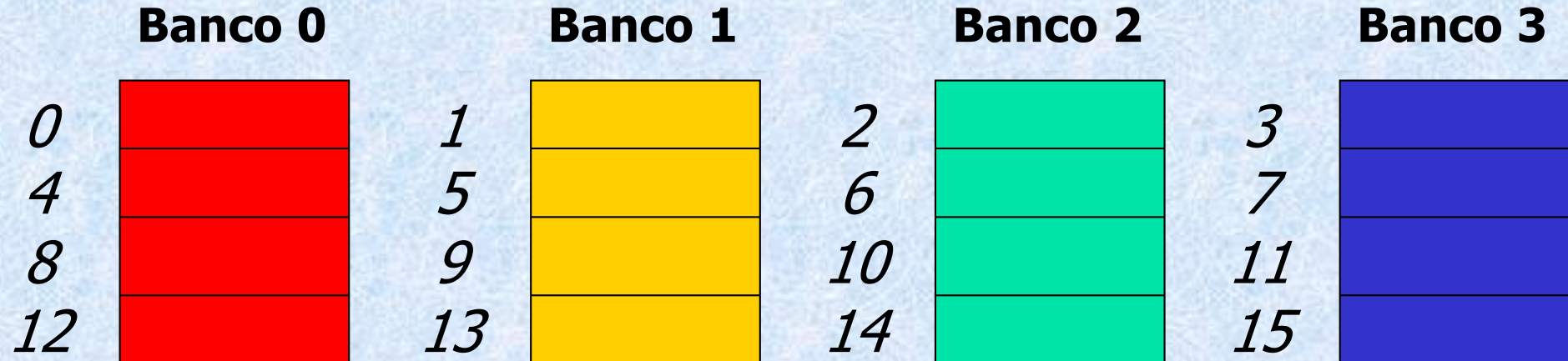
---

- Os chips de memória podem ser organizados em bancos para permitir que diversas palavras sejam lidas ou escritas simultaneamente.
  - Um único controlador de memória é utilizado
- Uma boa função de mapeamento dos endereços para os bancos de memória é crucial para o bom desempenho desta estratégia.
  - Geralmente, a função MOD é utilizada, favorecendo acessos sequenciais.

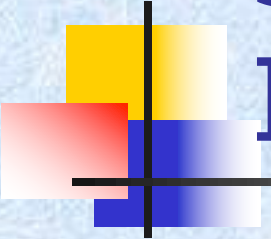


## 2. Memória Entrelaçada

- Exemplo (memória entrelaçada 4-way):



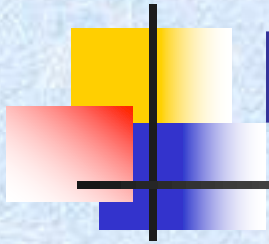




## 3. Bancos de Memória Independentes

---

- Neste caso, existem múltiplos controladores de memória, que permitem que os bancos operem de maneira independente.
- Cada banco necessita de linhas de endereço separadas e, frequentemente, de barramentos separados.



# Memória SRAM

---

- A memória SRAM (Static RAM) utiliza de 4 a 6 transistores por bit.
- Os valores binários são armazenados utilizando configurações de flip-flops.
- O SRAM não necessita de refreshs periódicos porém necessita de fornecimento constante de energia.
- SRAMs são mais rápidas, mais caras e menos densas que as DRAMs
- Utilizadas principalmente em memórias cache