Tecnologia de memória/Memória cache

Tecnologia para Memória

- **Acesso Randômico:**
  - "Randômico" é bom: tempo de acesso igual para todas localidades
  - DRAM: Dynamic Random Access Memory
    - Alta densidade, baixa potência, barata (1x), lenta(10x)
    - Dynamic: precisa ser "refrescada" regularmente
  - SRAM: Static Random Access Memory
    - Baixa densidade, alta potência, cara (100x), rápida (1x)
    - Static: conteúdo dura "para sempre"(até ter energia)
- **Tecnologia de Acesso "Não-tão-randômica":**
  - Tempo de acesso varia de localidade para localidade
  - Exemplos: Disco, CDROM

Tecnologia para Memória

- **Tecnologia de Acesso Seqüencial: tempo de acesso proporcional à localidade (ex.,Fita)**
- **Tecnologia de acesso randômico:**
  - Memória Principal: DRAMs + Caches: SRAMs

Random-Access Memory (RAM)

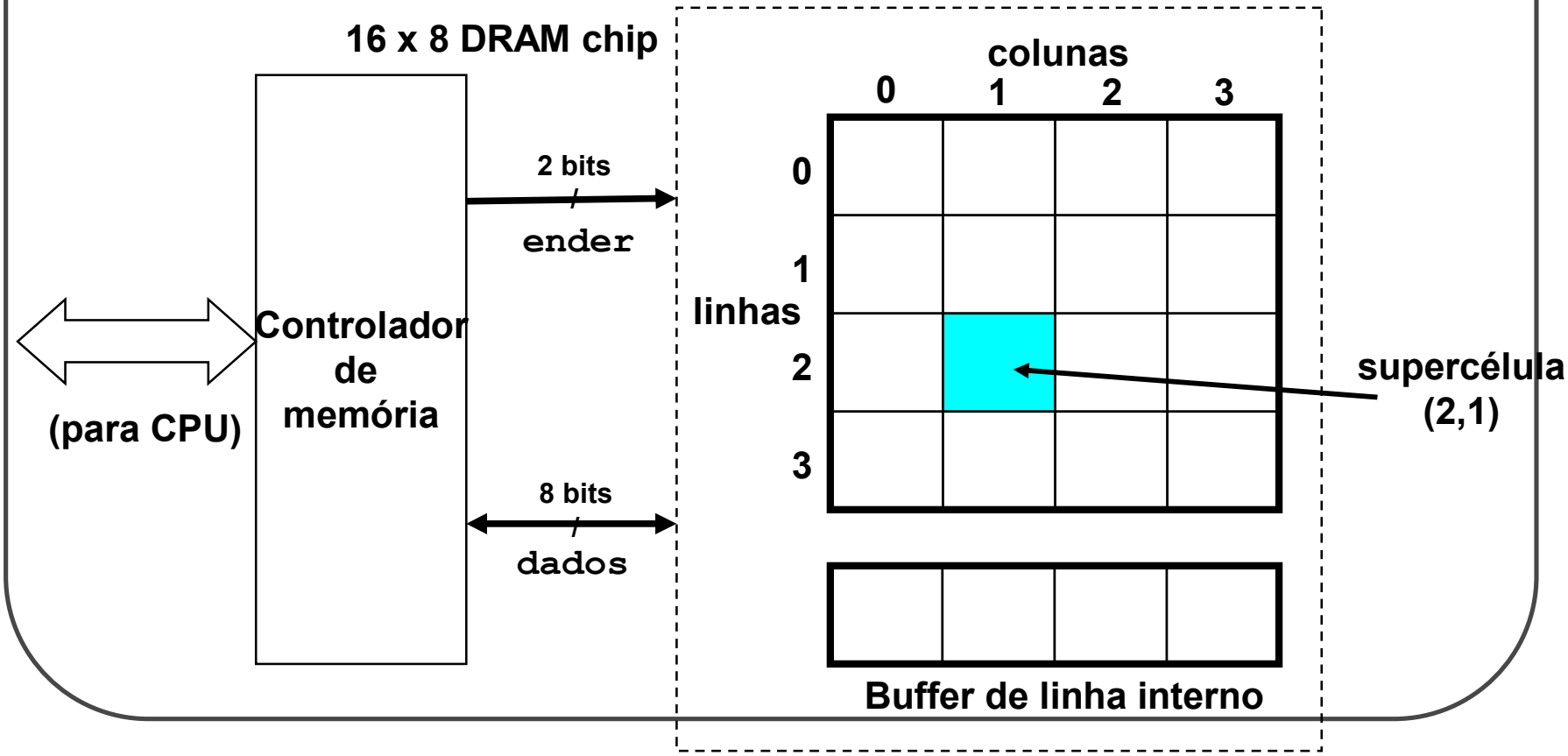
- **Características principais**
  - Unidade básica de armazenamento é a célula (um bit por célula)
  - Vários chips RAM formam uma memória
- **Static RAM (SRAM)**
  - Cada célula armazena um bit com circuito de seis transistores
  - Retém valor enquanto estiver ligada
  - Relativamente menos sensível a ruídos
  - Mais rápida e cara que DRAM
- **Dynamic RAM (DRAM)**
  - Cada célula armazena um bit com capacitor e transistor
  - Valor tem que ser atualizado cada 10-100 ms
  - Sensível a ruídos
  - Mais lenta e barata que SRAM

SRAM vs DRAM

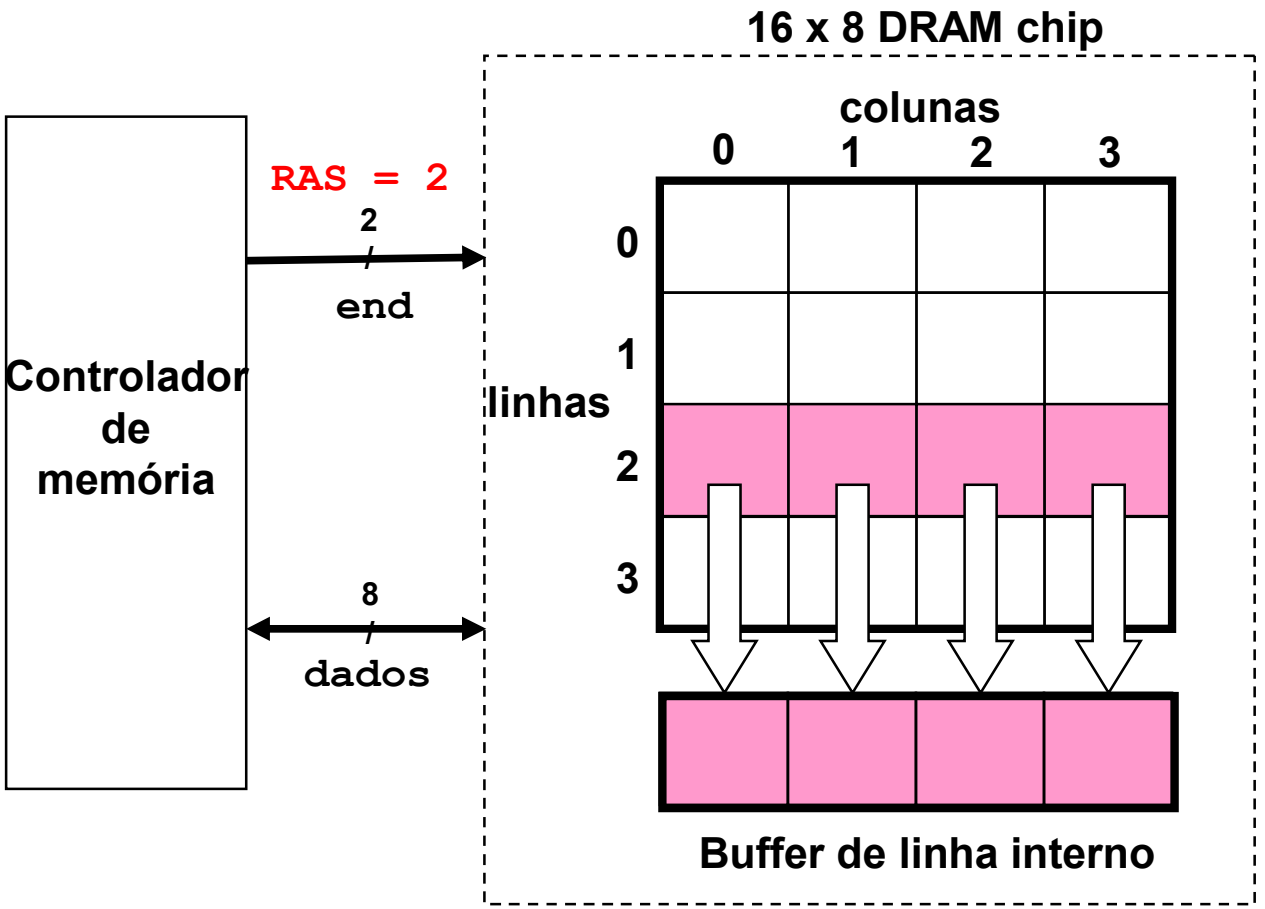
	Tran. por bit	Tempo de acesso	Persistente?	Sensível?	Custo	Aplicações
SRAM	6	1X	Sim	Não	100x	memórias cache
DRAM	1	10X	Não	Sim	1X	memórias principais

Organização convencional da DRAM

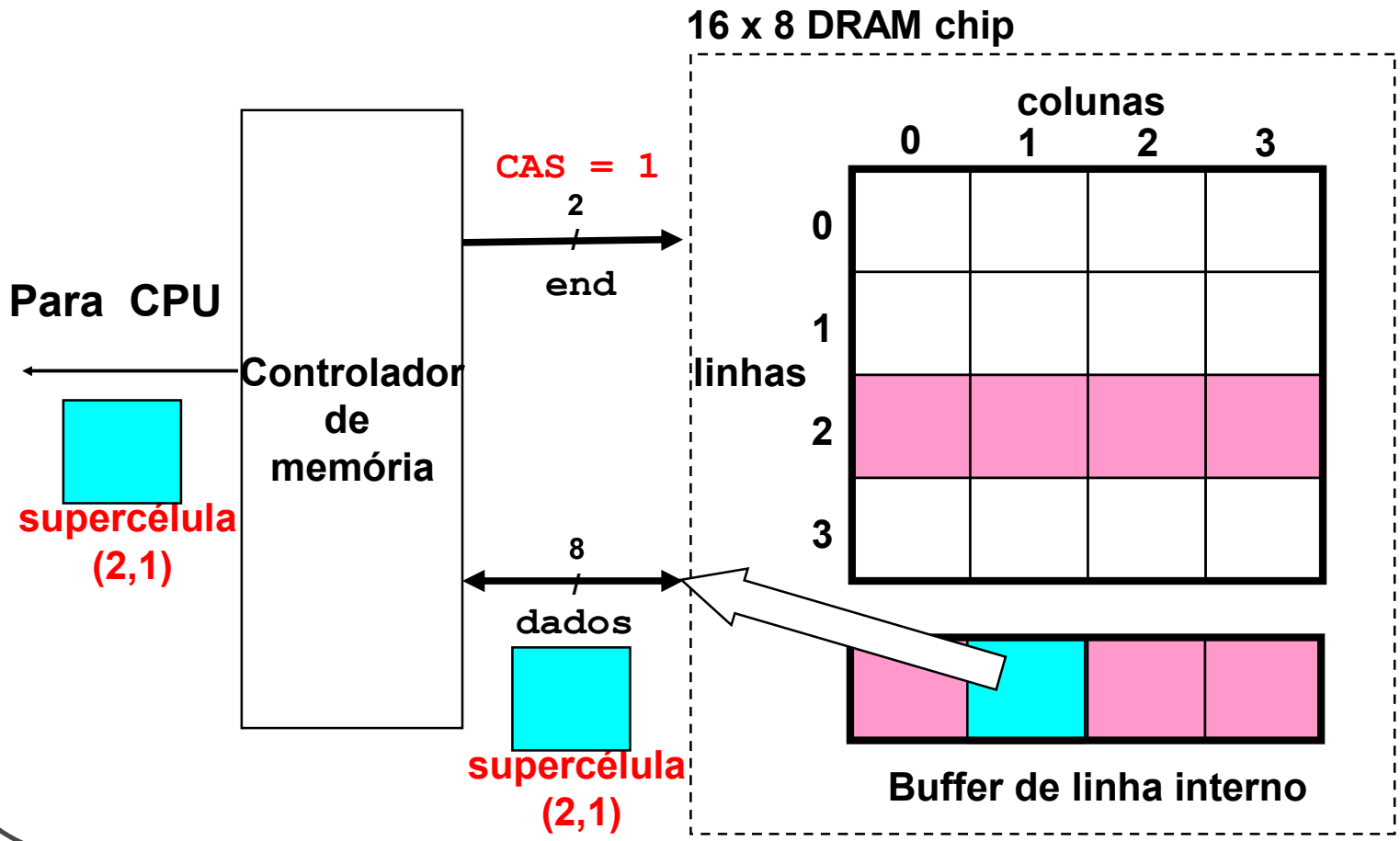
- **d x w DRAM:**
  - dw bits organizados como d supercélulas de tamanho w bits



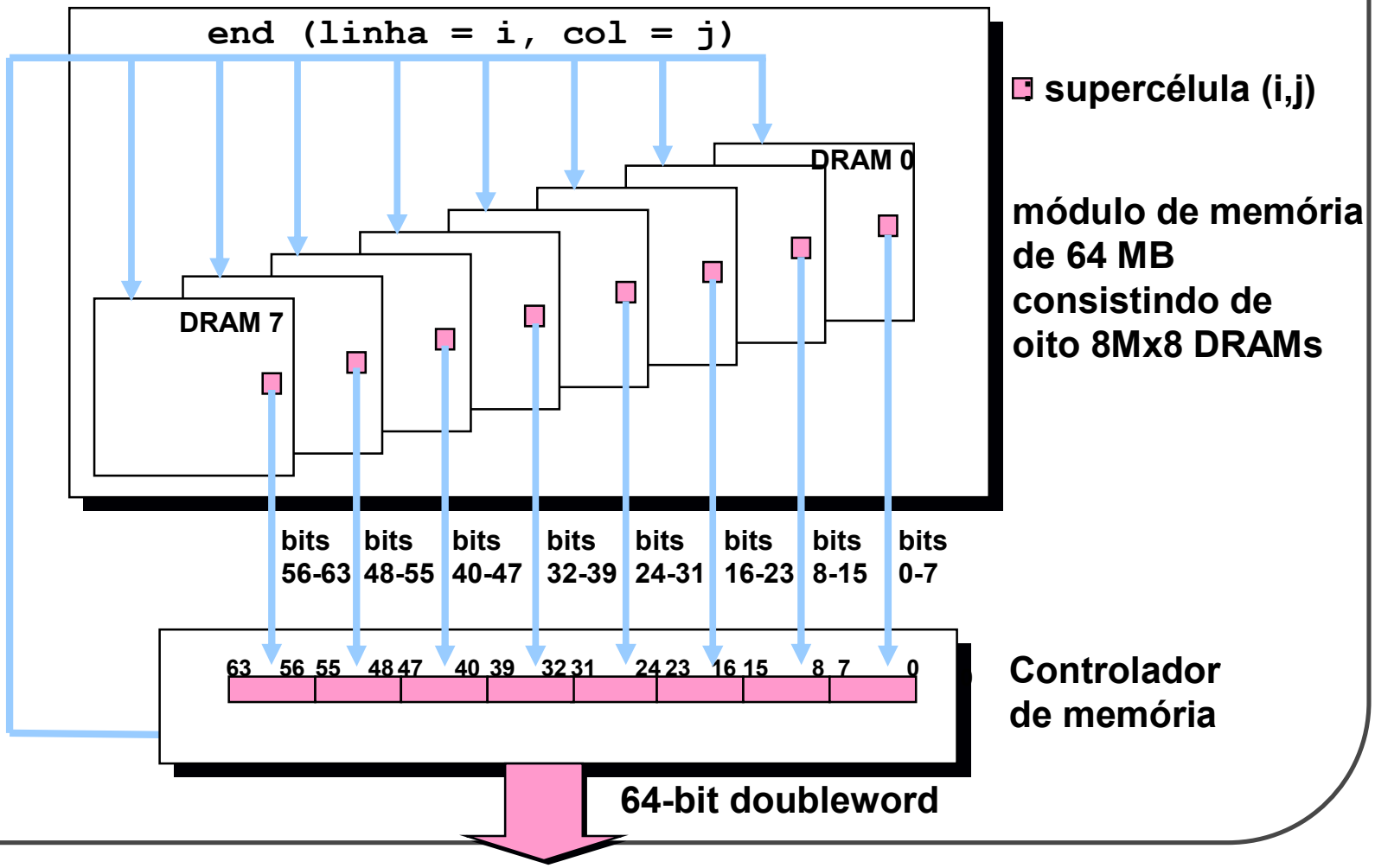
Lendo supercélula (2,1) da DRAM



Lendo supercélula (2,1) de DRAM



Módulos de memória



supercélula (i,j)

módulo de memória de 64 MB consistindo de oito 8Mx8 DRAMs

Controlador de memória

Enhanced DRAMs

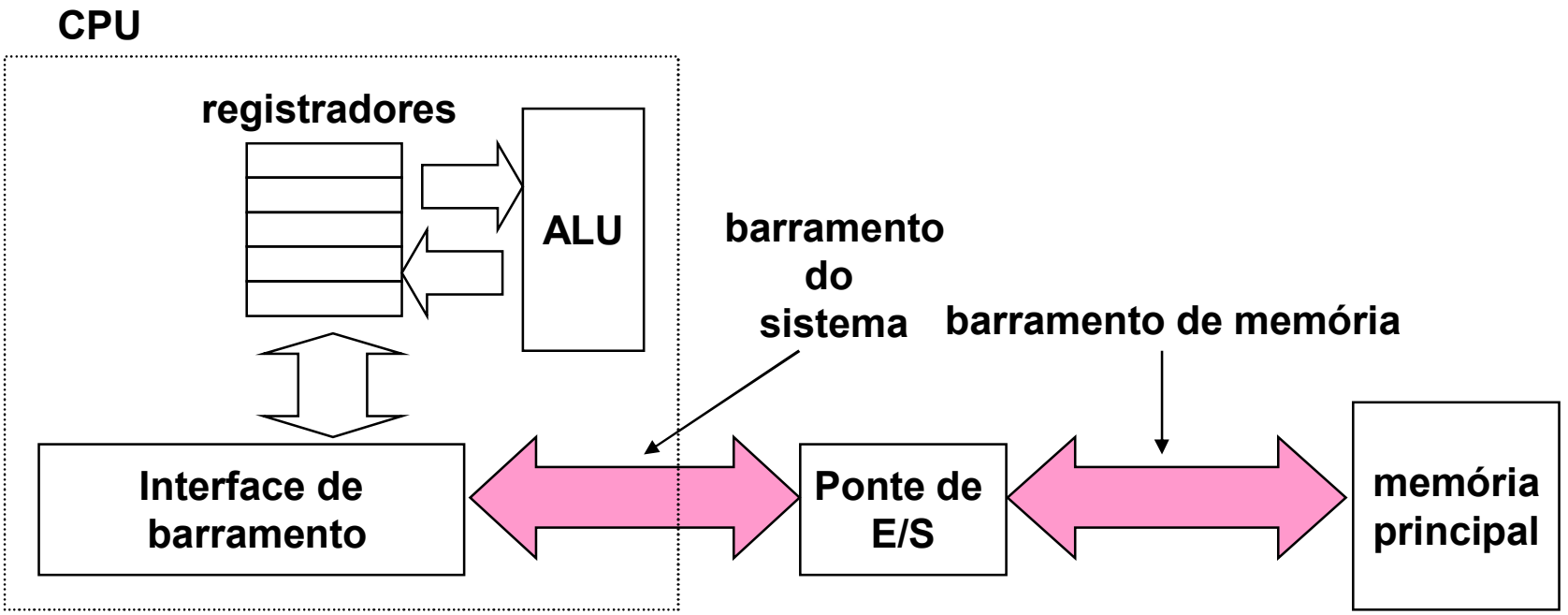
- **Dram aprimoradas derivam da DRAM básica**
  - Fast page mode DRAM (FPM DRAM)
    - Acesso a linha com [RAS, CAS, CAS, CAS, CAS] ao invés de [(RAS,CAS), (RAS,CAS), (RAS,CAS), (RAS,CAS)].
  - Extended data out DRAM (EDO DRAM)
    - FPM DRAM melhorada
  - Synchronous DRAM (SDRAM)
    - Trabalha com clock ao invés de ser assíncrona
  - Double data-rate synchronous DRAM (DDR SDRAM)
    - Melhoria da SDRAM
  - Video RAM (VRAM)
    - Semelhante à FPM DRAM
    - Dual ported (permite leituras e escritas concorrentes)

Memórias não voláteis

- **DRAM e SRAM são voláteis, perdem informação quando desligadas**
- **Memórias não voláteis retêm valor mesmo desligadas**
  - Nome genérico read-only memory (ROM), mas também podem ser modificadas
- **Tipos de ROMs**
  - Programmable ROM (PROM)
  - Eraseable programmable ROM (EPROM)
  - Electrically eraseable PROM (EEPROM)
  - Flash memory
- **Firmware**
  - Programa armazenado em uma ROM
    - Código de boot, BIOS (basic input/ouput system)
    - placas gráficas, controladores de disco

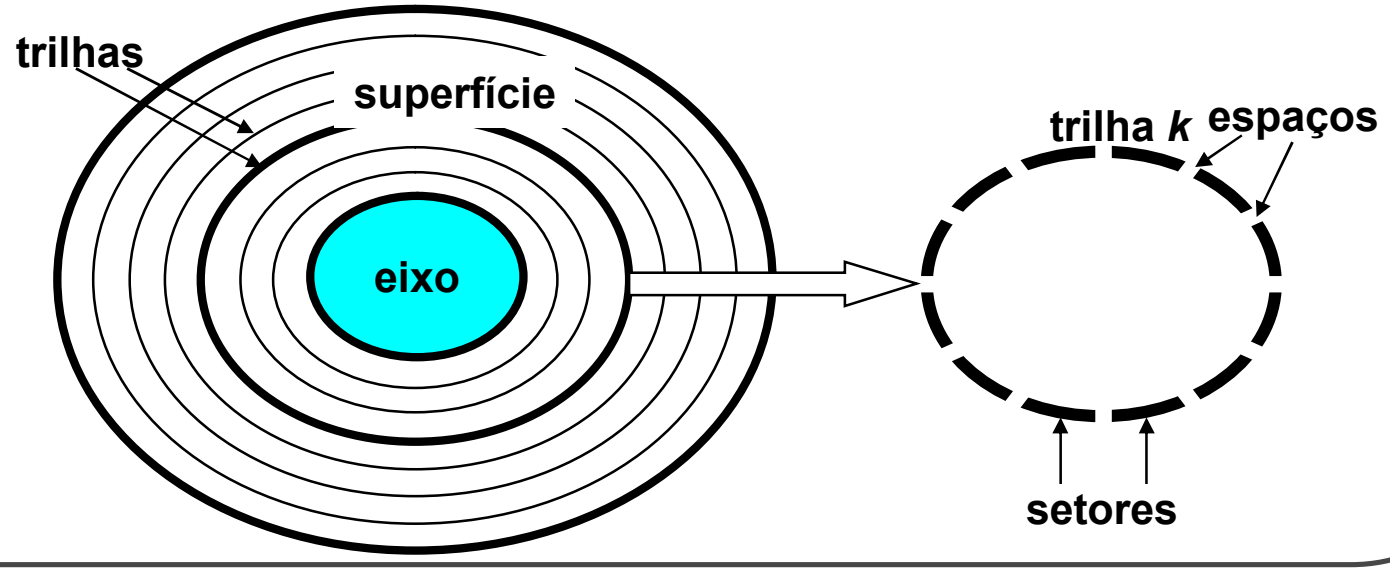
Estrutura típica de barramento conectando CPU e memória

- Um barramento é uma coleção de fios paralelos que carregam endereço, dados e sinais de controle



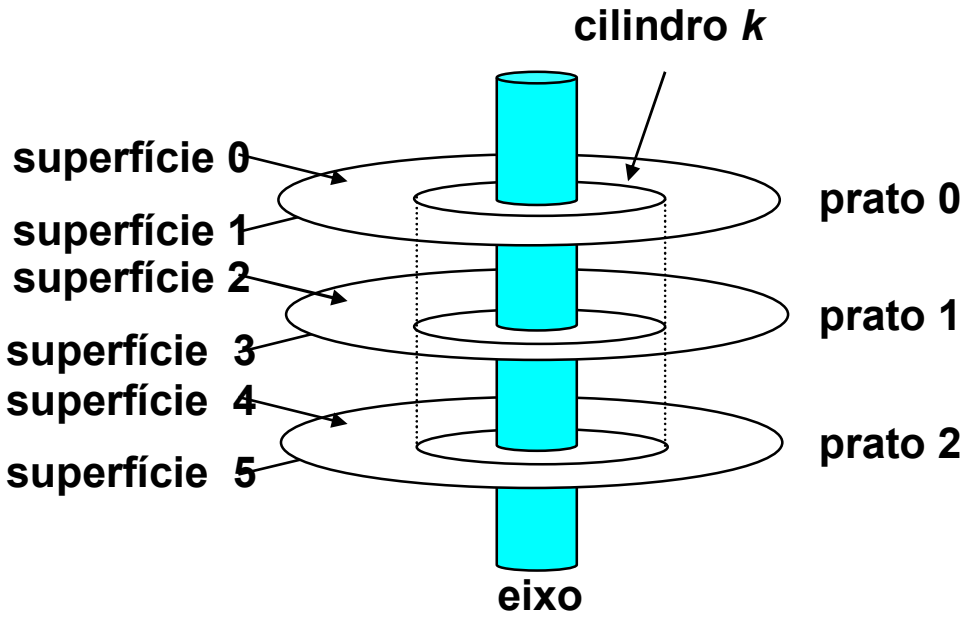
Geometria dos discos

- Um disco consiste de pratos, cada qual com duas superfícies
- Cada superfície consiste de anéis concêntricos denominados trilhas.
- Cada trilha consiste de setores separados por espaços.



Discos com múltiplos pratos

- **Trilhas alinhadas formam um cilindro**



Capacidade do disco

- **Capacidade: número máximo de bits que podem ser armazenados expresso em gigabytes (1 GB = 10^9)**
- **Fatores que determinam a capacidade**
  - Densidade de gravação (bits/in): número de bits que podem ser gravados em 1 polegada de uma trilha
  - Densidade de trilha(trilhas/in): número de trilhas que podem existir em um segmento radial
  - Densidade de armazenamento (bits/in2): produto da densidade de gravação com densidade de trilha
- **Disco modernos particionam as trilhas em conjuntos disjuntos denominados zonas de armazenamento**
  - Cada trilha em uma zona possui o mesmo número de setores, determinado pela circunferência da trilha mais interna
  - Cada zona possui um número diferente de setores/trilha

Calculando capacidade de disco

- **Capacidade = (# bytes/setor) x (méd. # setores/trilha) x (# trilhas/superfície) x (# superfícies/prato) x(# pratos/disco)**
- **Exemplo:**
  - 512 bytes/setor
  - 300 setores/trilha (em média)
  - 20.000 trilhas/superfície
  - 2 superfícies/prato
  - 5 pratos/disco

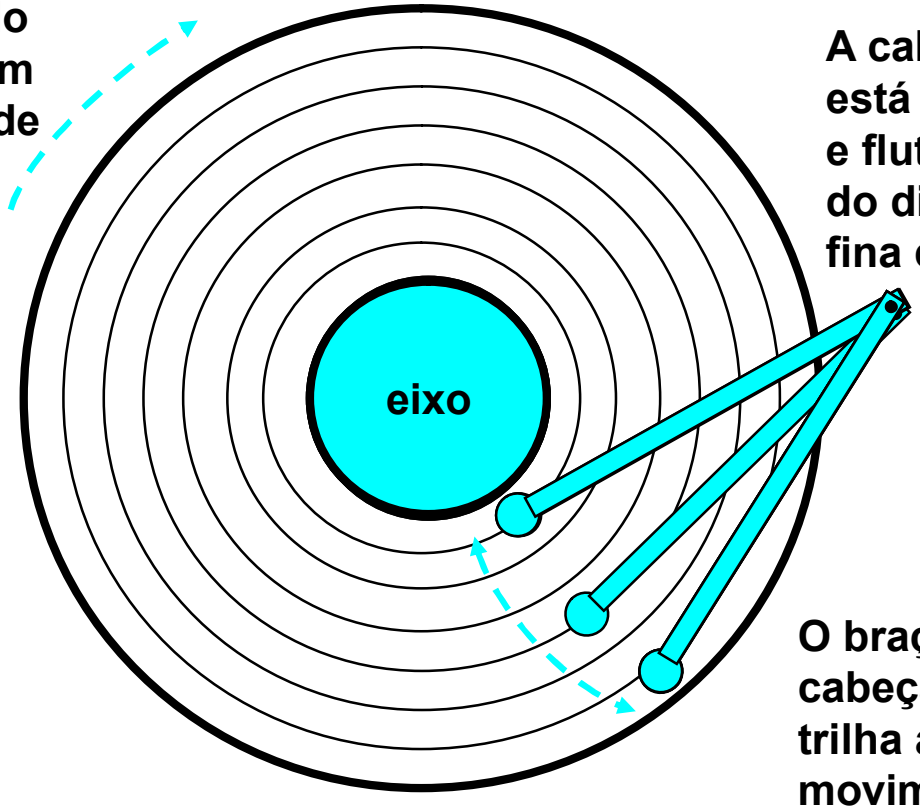
- **Capacidade = 512 x 300 x 20000 x 2 x 5**

  = 30.720.000.000

  = 30,72 GB

Operação do disco

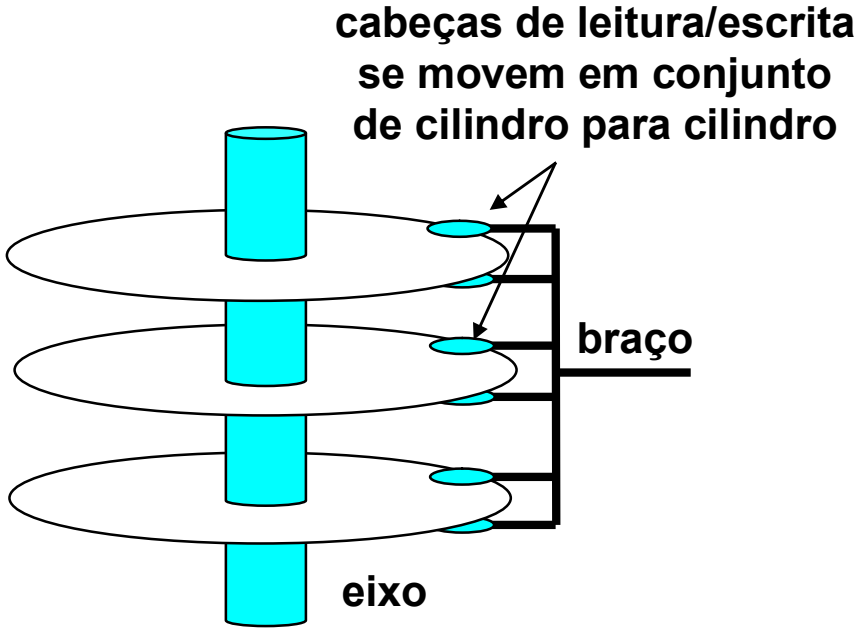
A superfície do disco gira com uma velocidade de rotação constante



A cabeça de leitura/escrita está ligada ao final do braço e flutua sobre a superfície do disco em cima de uma fina camada de ar

O braço pode posicionar a cabeça sobre qualquer trilha através de um movimento radial

Operação de disco com múltiplos discos



Tempo de acesso ao disco

- **Tempo médio de acesso a um setor desejado:**
  - Tacesso = Tmed procura + Tmed rotação + Tmed transferência
- **Tempo de procura (Tmed procura)**
  - Tempo para posicionar as cabeças no clindro que contém o setor desejado
  - Típico Tmed procura = 9 ms
- **Latência rotacional (Tmed rotação)**
  - Tempo de espera para que o primeiro bit do setor passe pela cabeça
  - Tmed rotação = 1/2 x 1/RPMs x 60 seg/1 min
- **Tempo de transferência (Tmed transferência)**
  - Tempo para ler os bits do setor
  - Tmed transferência = 1/RPM x 1/(med # setores/trilha) x 60 segs/1 min.

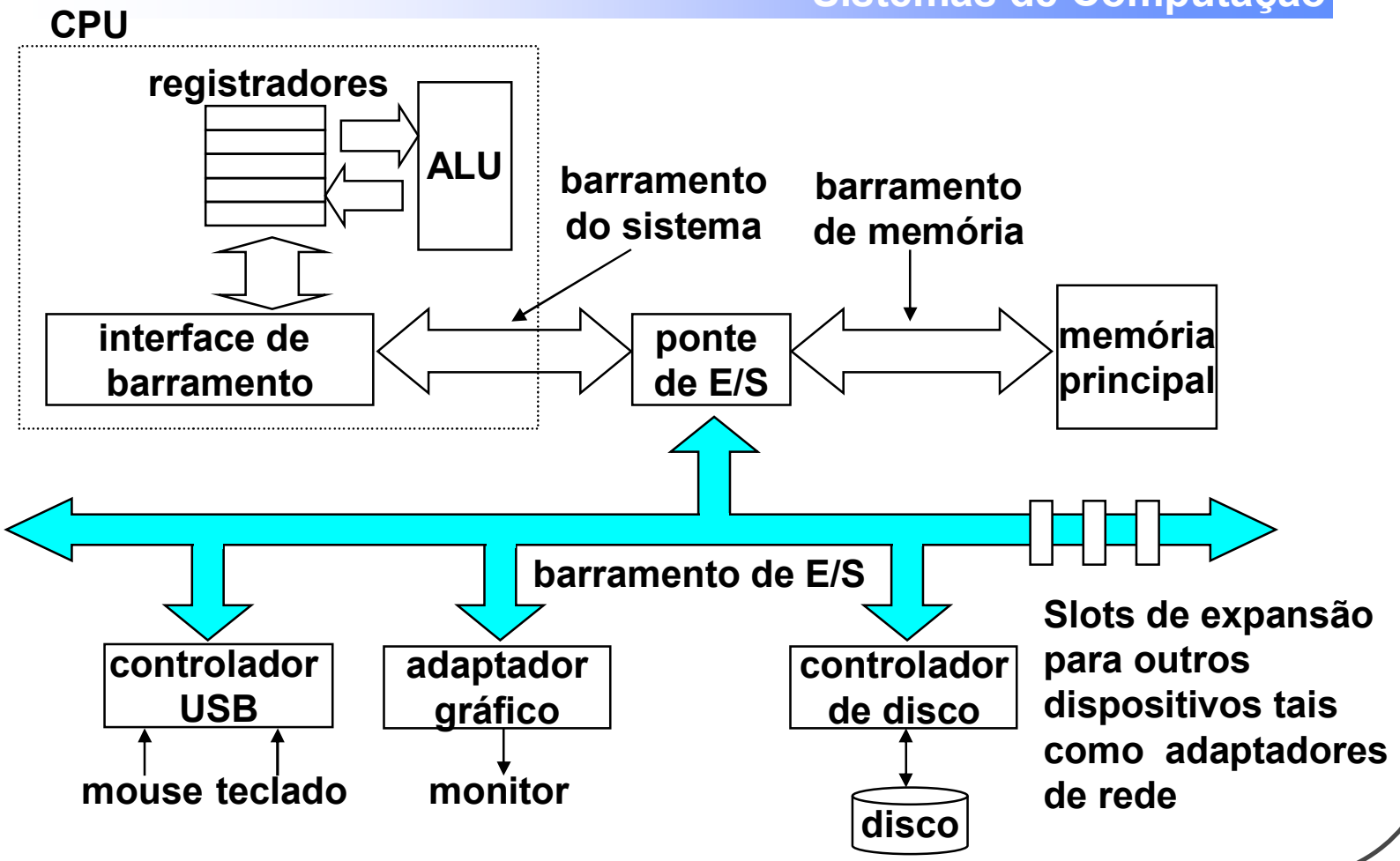
Exemplo

- **Dados:**
  - Velocidade de rotação= 7.200 RPM
  - Tempo médio de procura = 9 ms.
  - Med # setores/trilha = 400.
- **Teremos:**
  - Tmed rotação = 1/2 x (60 segs/7200 RPM) x 1000 ms/seg = 4 ms.
  - Tmed transferência = 60/7200 RPM x 1/400 segs/trilha x 1000 ms/seg = 0.02 ms
  - Tacesso = 9 ms + 4 ms + 0.02 ms
- **Pontos importantes:**
  - Tempo de acesso dominado pelo tempo de procura e latência rotacional
  - Tempo de acesso da SRAM é 4 ns/doubleword, DRAM por volta de 60 ns
    - Disco é aprox. 40.000 vezes mais devagar que SRAM, e 2.500 vezes mais devagar que DRAM

Blocos lógicos

- **Discos modernos apresentam uma visão abstrata mais simples da geometria complexa de setores:**
  - O conjunto de setores disponíveis é modelado como uma seqüência de blocos lógicos de tamanho **b**
- **O mapeamento entre os blocos lógicos e físicos é realizado pelo hardware/firmware dos controladores de disco**
- **Permite que o controlador separe cilindros sobressalentes para cada zona**

Barramento de E/S



Tendências de armazenamento

Sistemas de Computação

SRAM

métrica	1980	1985	1990	1995	2000	*2000:1980*
$/MB	19,200	2,900	320	256	100	*190*
acesso (ns)	300	150	35	15	2	*100*

DRAM

métrica	1980	1985	1990	1995	2000	*2000:1980*
$/MB	8,000	880	100	30	1	*8,000*
acesso (ns)	375	200	100	70	60	*6*
tam. típico(MB)	0.064	0.256	4	16	64	*1,000*

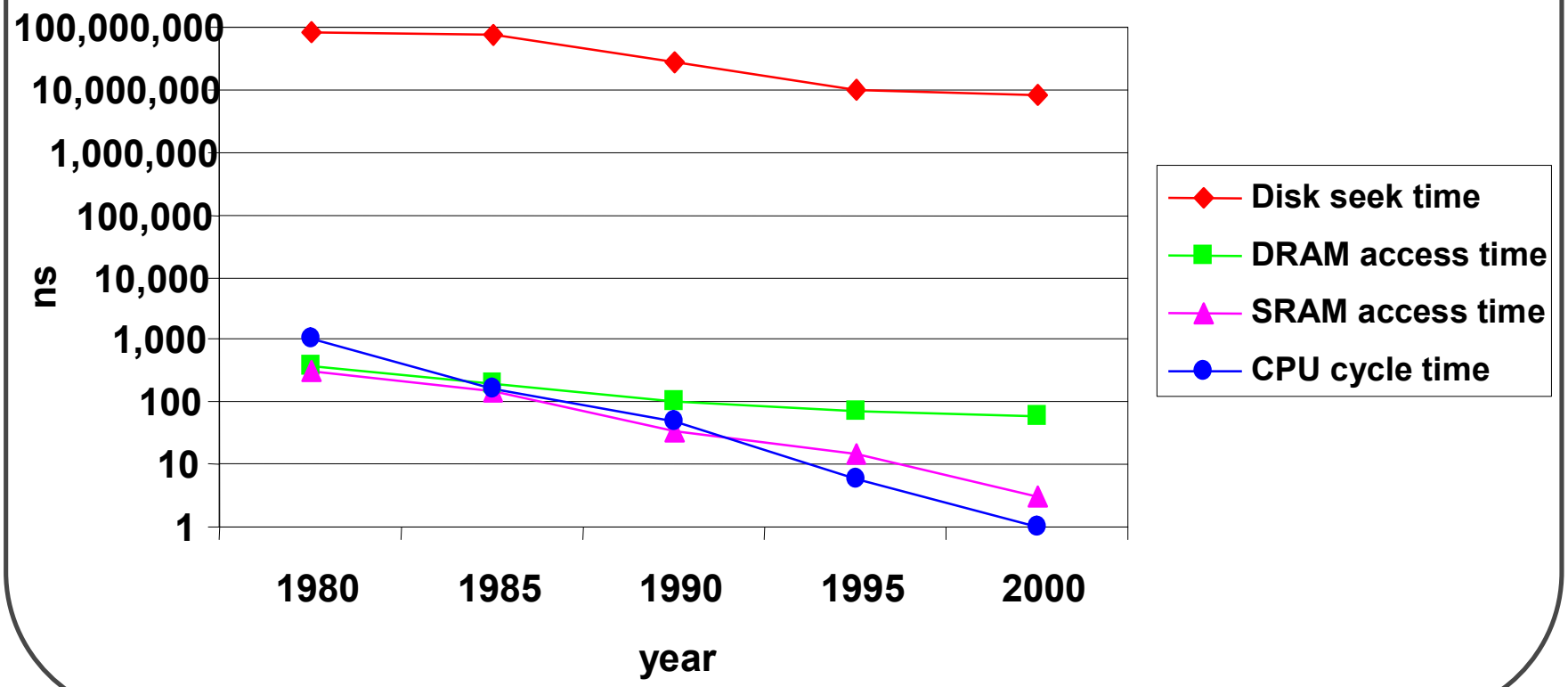
Disco

métrica	1980	1985	1990	1995	2000	*2000:1980*
$/MB	500	100	8	0.30	0.05	*10,000*
acesso (ms)	87	75	28	10	8	*11*
tam. típico (MB)	1	10	160	1,000	9,000	*9,000*

Relógios de CPU

	1980	1985	1990	1995	2000	*2000:1980*
processador	8080	286	386	Pent	P-III	
freqüência (MHz)	1	6	20	150	750	750
tempo de ciclo(ns)	1,000	166	50	6	1.6	750

O gap memória-CPU



Princípio da localidade

- **Programas tendem a reutilizar dados e instruções perto daqueles que foram utilizados recentemente**
- **Localidade temporal: Itens recentemente referenciados irão ser provavelmente referenciados em futuro próximo**
- **Localidade espacial: Itens com endereços próximos tendem a ser referenciados em instantes de tempo próximos**

Exemplo de localidade

Exemplo de localidade

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Dados**
  - Acessar elementos em série (padrão de referência passo 1): **Localidade espacial**
  - Referência à `sum` em cada iteração: **Localidade temporal**
- **Instruções**
  - Instruções executadas em seqüência: **Localidade espacial**
  - Passa pelo loop repetidamente: **Localidade temporal**

Exemplo de localidade

- **Pergunta:** A função abaixo tem boa localidade ?

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum
}
```

Exemplo de localidade

- **E esta ?**

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```

Exemplo de localidade

- **Pergunta:** É possível trocar os loops abaixo de modo que a função acesse o array de 3 dimensões com um padrão de referência de passo 1 ?

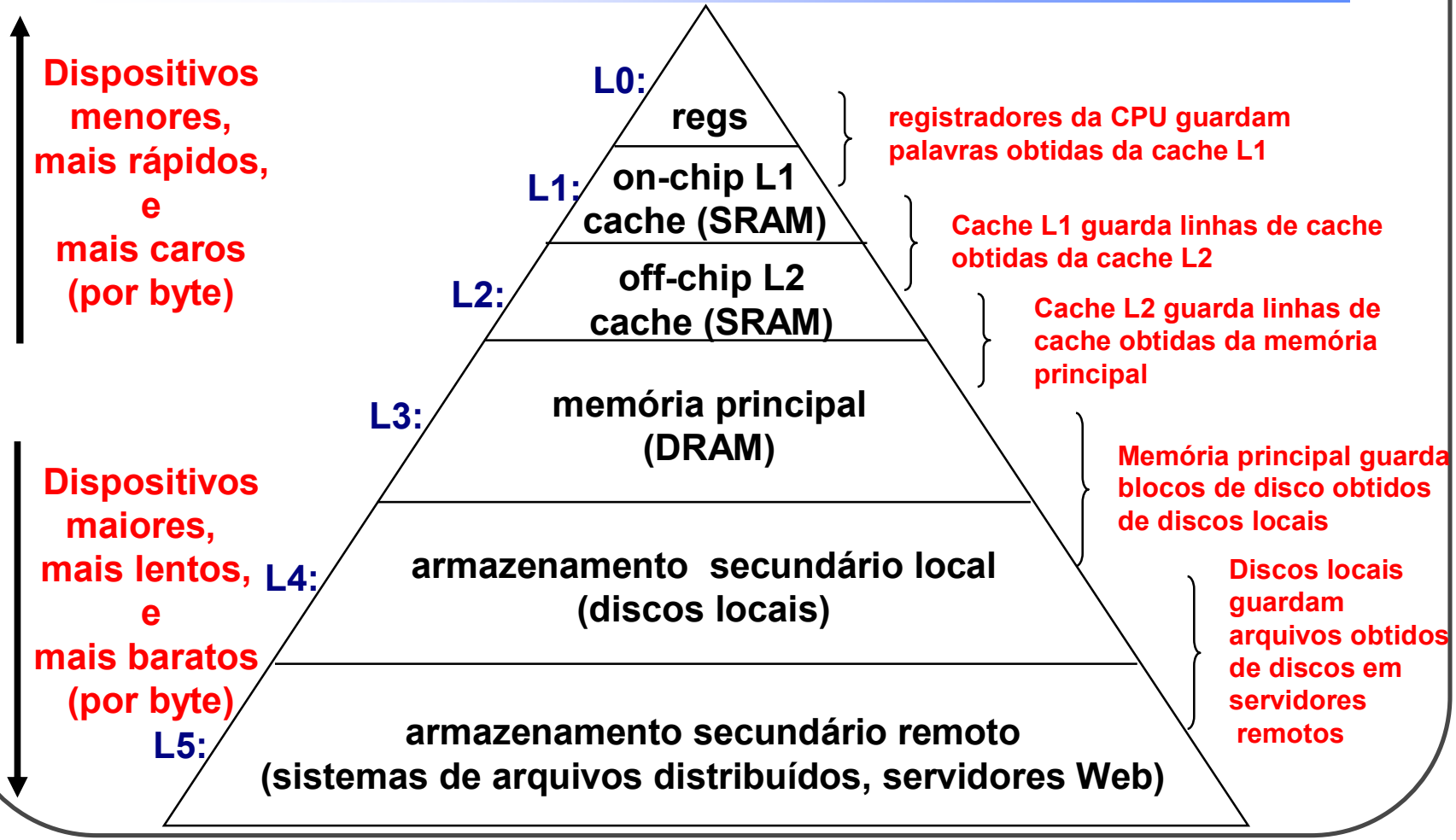
```
int sumarray3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum
}
```

Hierarquia de memórias

- **Algumas propriedades fundamentais e permanentes de hardware e software:**
  - Tecnologias rápidas para armazenamento custam mais por byte
  - O gap entre as velocidades de CPU e memória principal está aumentando
  - Programas bem escritos tendem a ter boa localidade

- **A partir destes fatores surge a idéia de organizar a memória e sistemas de armazenamento como uma hierarquia de memórias.**

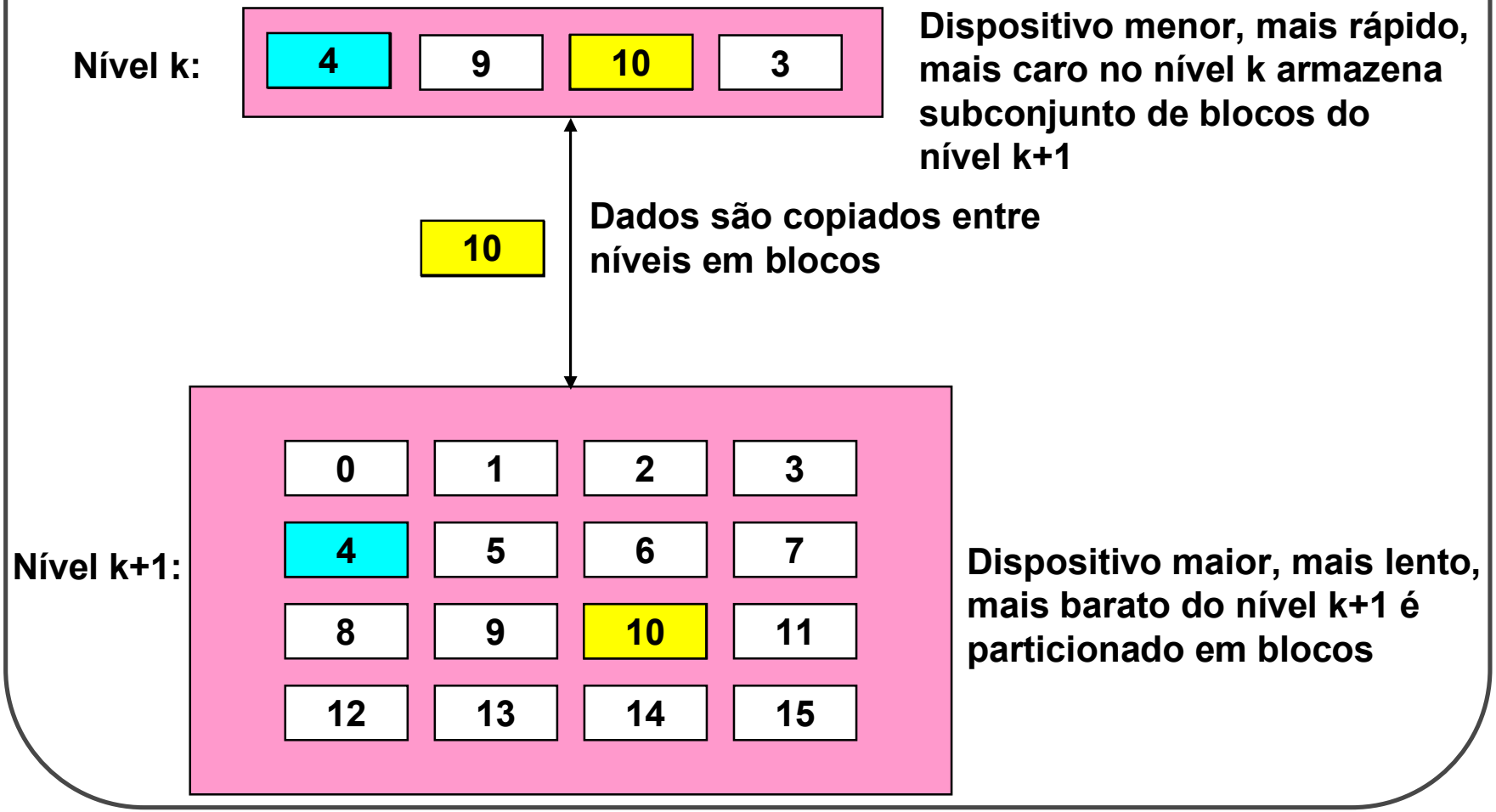
Exemplo de hierarquia de memórias



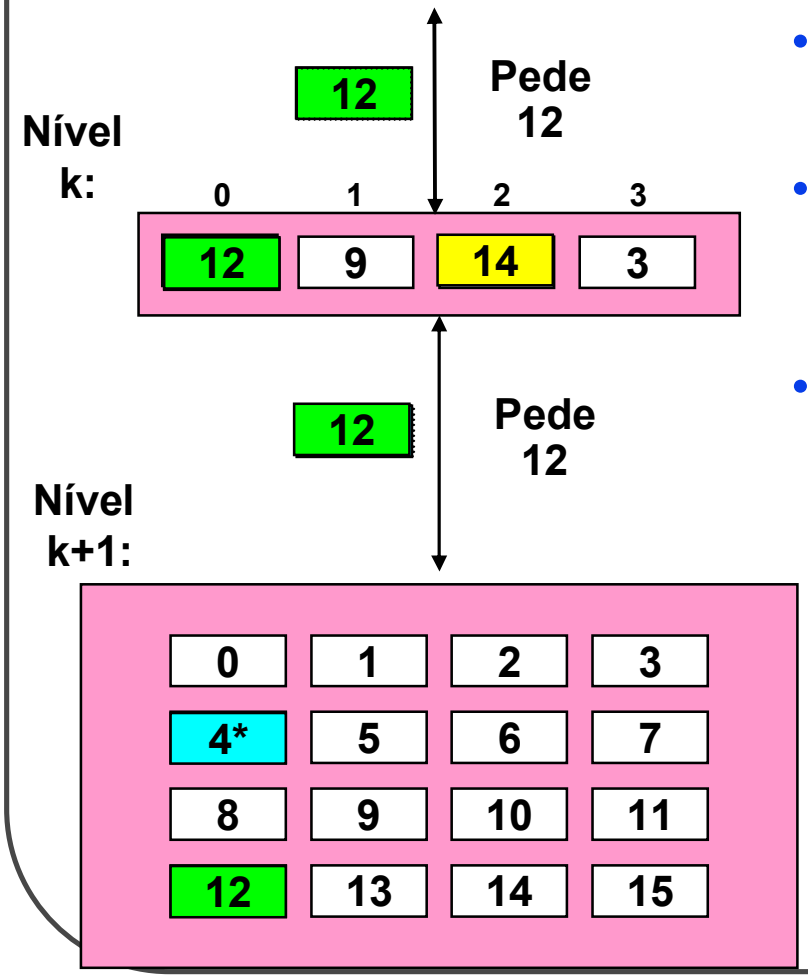
Caches

- **Cache: Um dispositivo menor e mais rápido que age como uma plataforma de acesso para um subconjunto de dados de um dispositivo maior e mais lento**
- **Idéia fundamental:**
  - Para cada k, o dispositivo mais rápido e menor no nível k serve com uma cache para um dispositivo maior e mais lento no nível k+1
- **Porque funciona ?**
  - Programas tendem a acessar dados no nível k mais freqüentemente que no nível k+1
  - Então, o armazenamento em k+1 pode ser mais lento e portanto maior e mais barato por bit
  - Resultado: Uma memória que custa tão barato quanto o dispositivo mais barato, mas que fornece dados para programas em uma taxa perto do dispositivo mais rápido

Caching na hierarquia de memórias



Conceitos gerais de caching

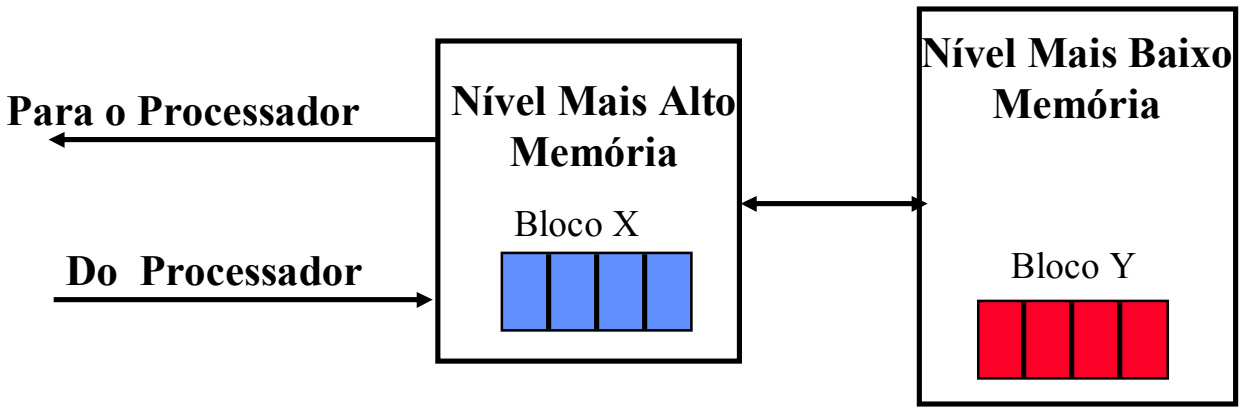


- **Programa precisa do objeto d, que está armazenado em algum bloco b**
- **Acerto de cache (hit)**
  - Programa encontra b na cache nível k. E.x., bloco 14.
- **Falta de cache (miss)**
  - b não está no nível k, então a cache do nível k precisa pegar o bloco do nível k+1. Ex., bloco 12.
  - Se a cache do nível k está cheia, então algum bloco corrente terá que sair para dar lugar ao novo. Quem será a vítima?
    - Política de colocação: onde o novo bloco deve ir ? Ex., b mod 4
    - Política de substituição: qual bloco deve sair ? Ex., LRU

Terminologia para Hierarquia de Memória

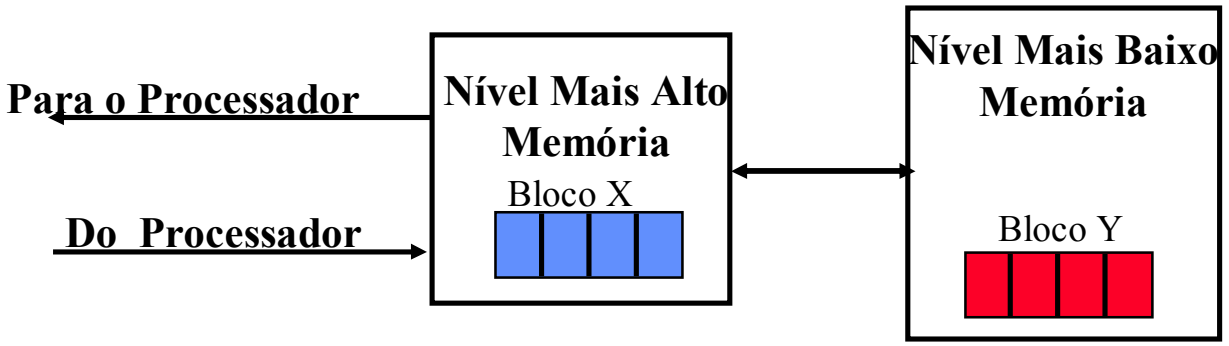
- **Acerto (hit): dado presente em algum bloco do nível mais alto (exemplo: Bloco X)**
  - Taxa de acertos: a fração de acessos à memória encontrados no nível mais alto
  - Tempo de acerto: Tempo para acessar nível mais alto que consiste de:

    Tempo de acesso à memória + Tempo para determinar acerto



Terminologia para Hierarquia de Memória

- **Falta (miss): dado precisa ser recuperado de um bloco localizado em um nível mais baixo (Bloco Y)**
  - Taxa de faltas = 1 - (Taxa de acertos)
  - Penalidade por falta: Tempo para substituir um bloco de nível mais alto + Tempo para enviar informação ao processador
- **Tempo de acerto << Penalidade por falta (nível mais alto menor e com memória mais rápida)**

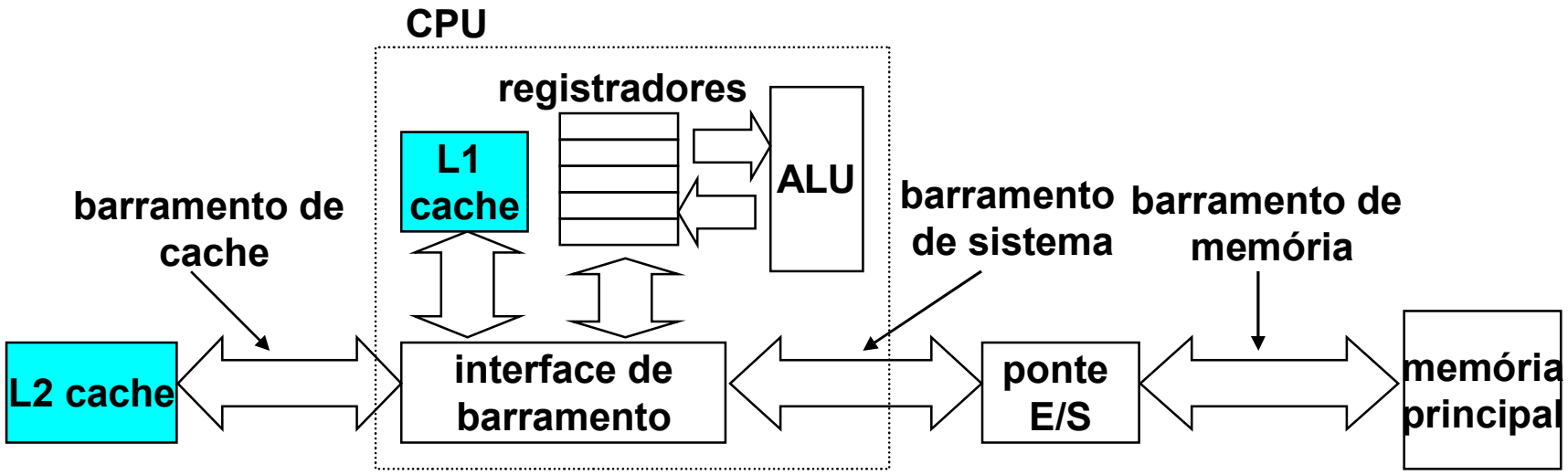


Conceitos gerais de caching

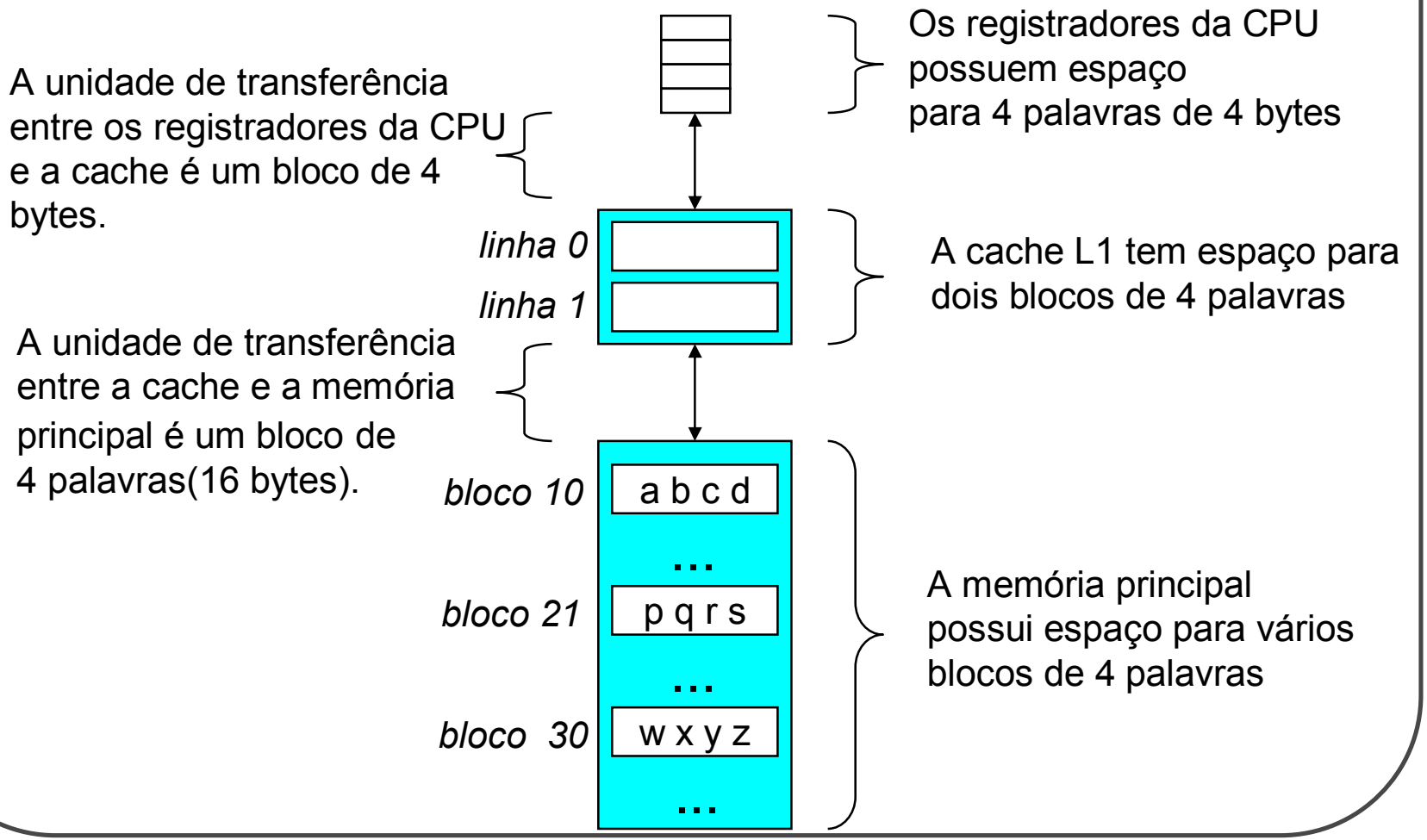
- **Tipo de faltas:**
  - Falta compulsória
    - Ocorrem porque a cache está vazia.
  - Falta por conflito
    - Ocorrem quando múltiplos objetos de dados são mapeados em um mesmo bloco no nível k
    - E.g. Suponha que bloco i seja referenciado por i mod 4 no nível k, então as referências aos blocos 0, 8, 0, 8, 0, 8, ... sempre acarretarão faltas
  - Falta por capacidade
    - Ocorrem quando o número de blocos ativos da cache é maior que a capacidade da cache.

Memórias cache

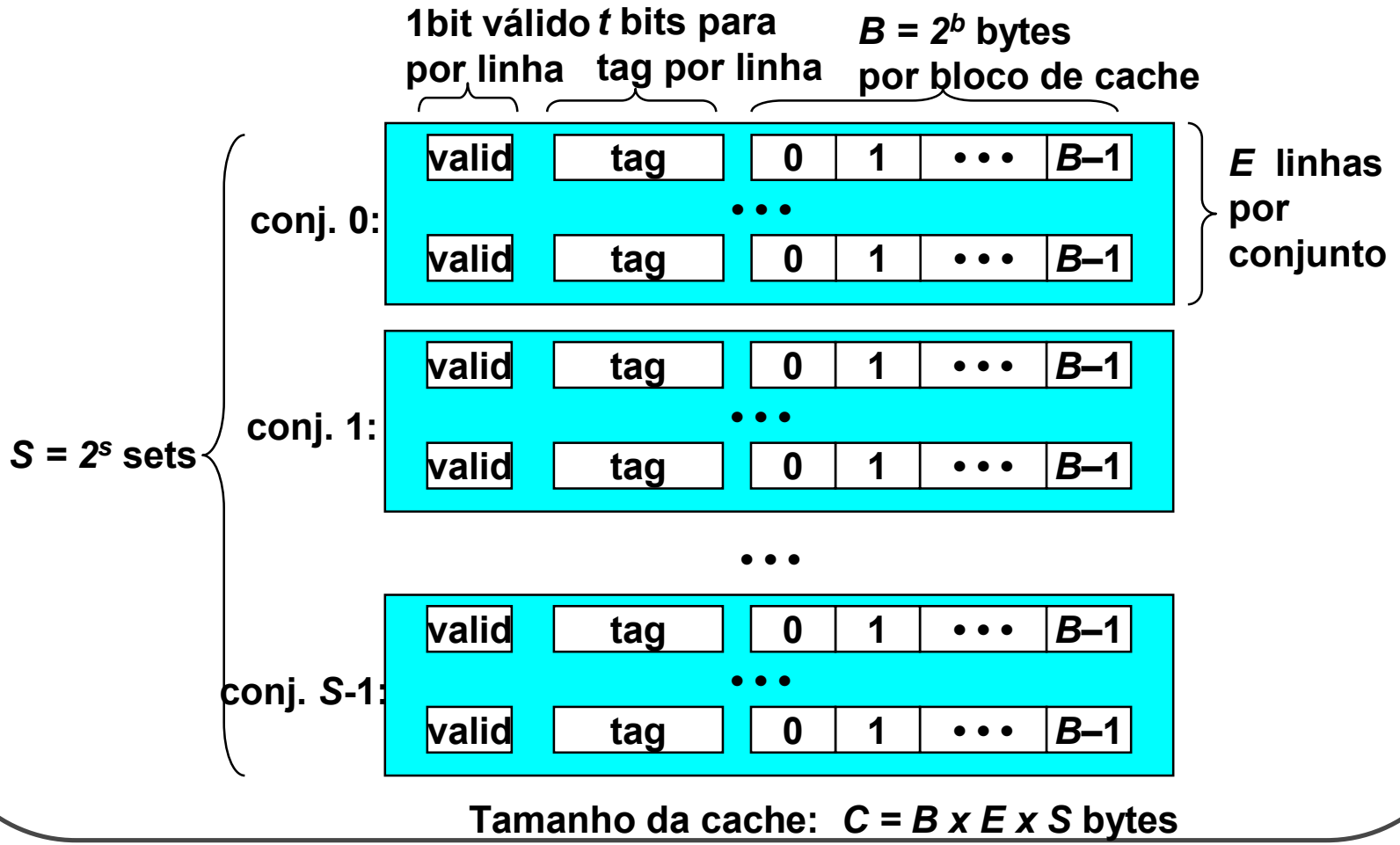
- **CPU procura por dados em L1, depois emL2 e finalmente na memória principal**



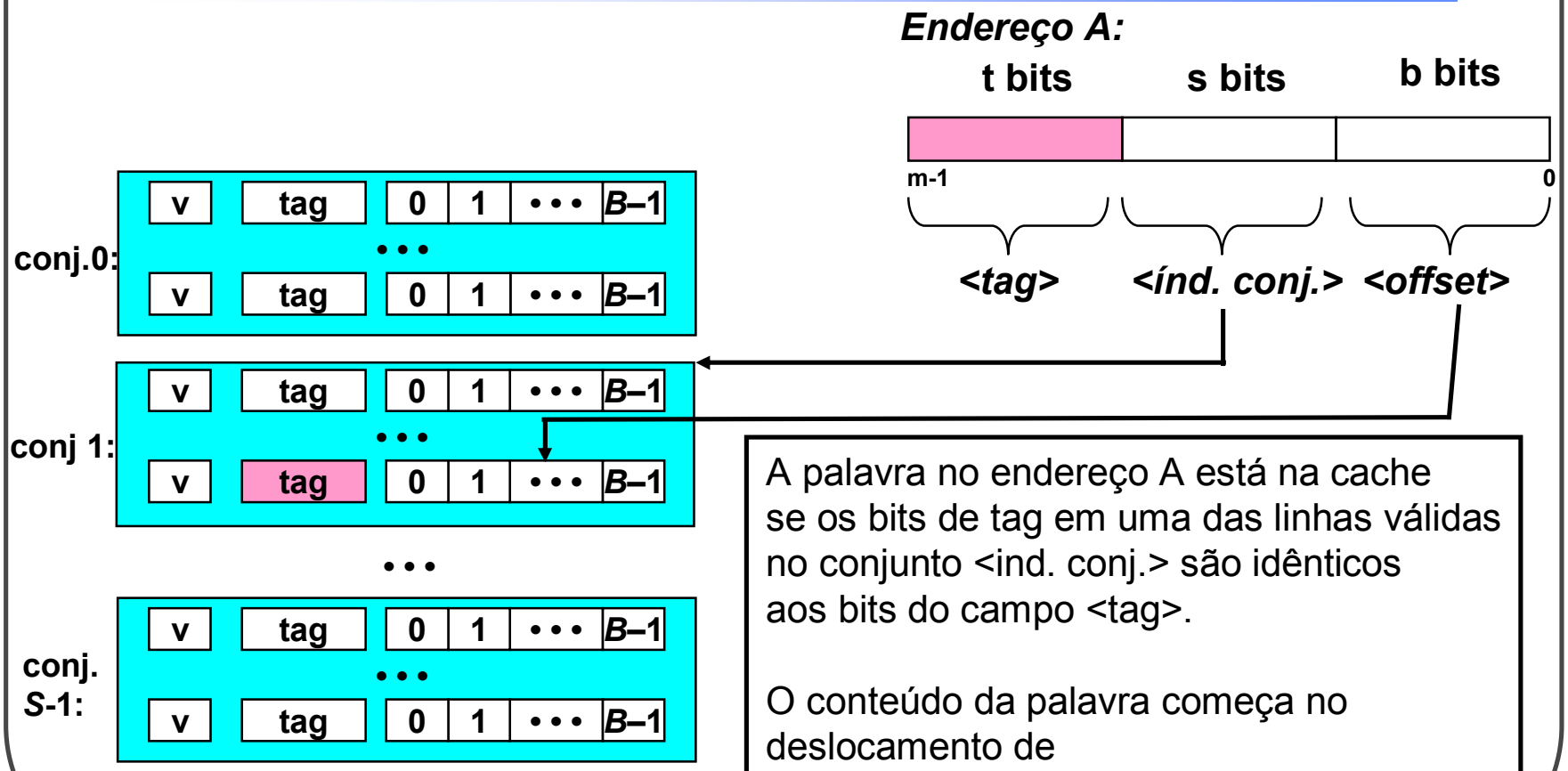
Inserindo Cache L1 entre CPU e memória principal



Organização geral da memória cache



Endereçamento de cache

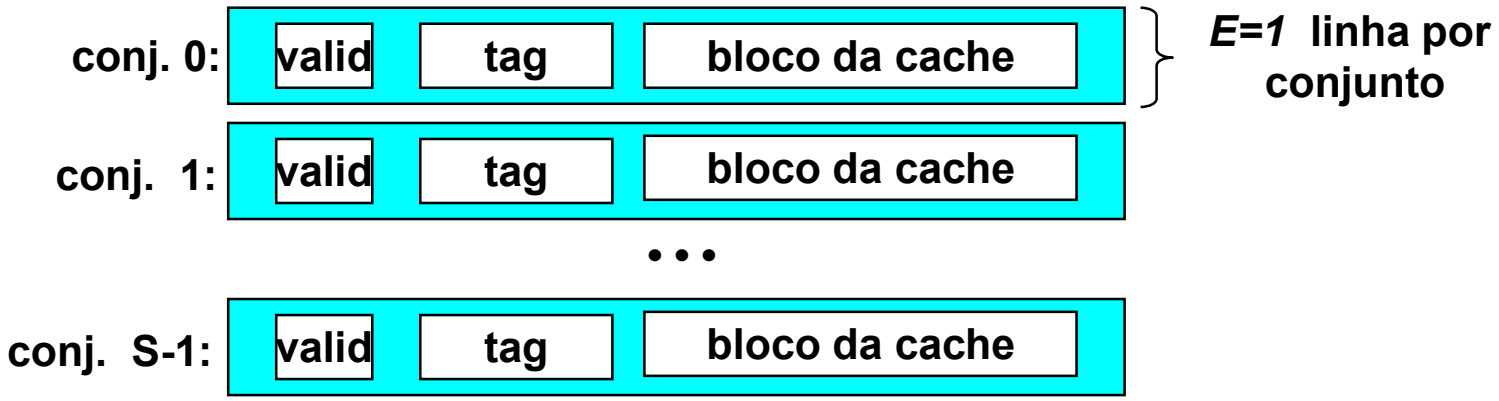


A palavra no endereço A está na cache se os bits de tag em uma das linhas válidas no conjunto <ind. conj.> são idênticos aos bits do campo <tag>.

O conteúdo da palavra começa no deslocamento de <offset> bytes a partir do início do bloco

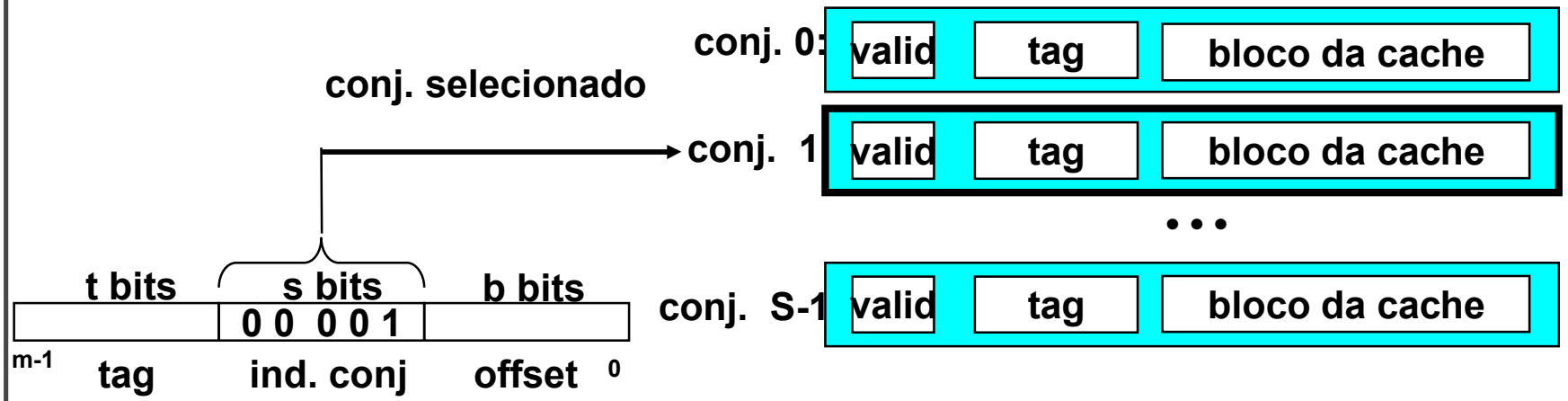
Cache mapeada diretamente

- **Forma mais simples**
- **Caracterizada por ter uma linha por conjunto**



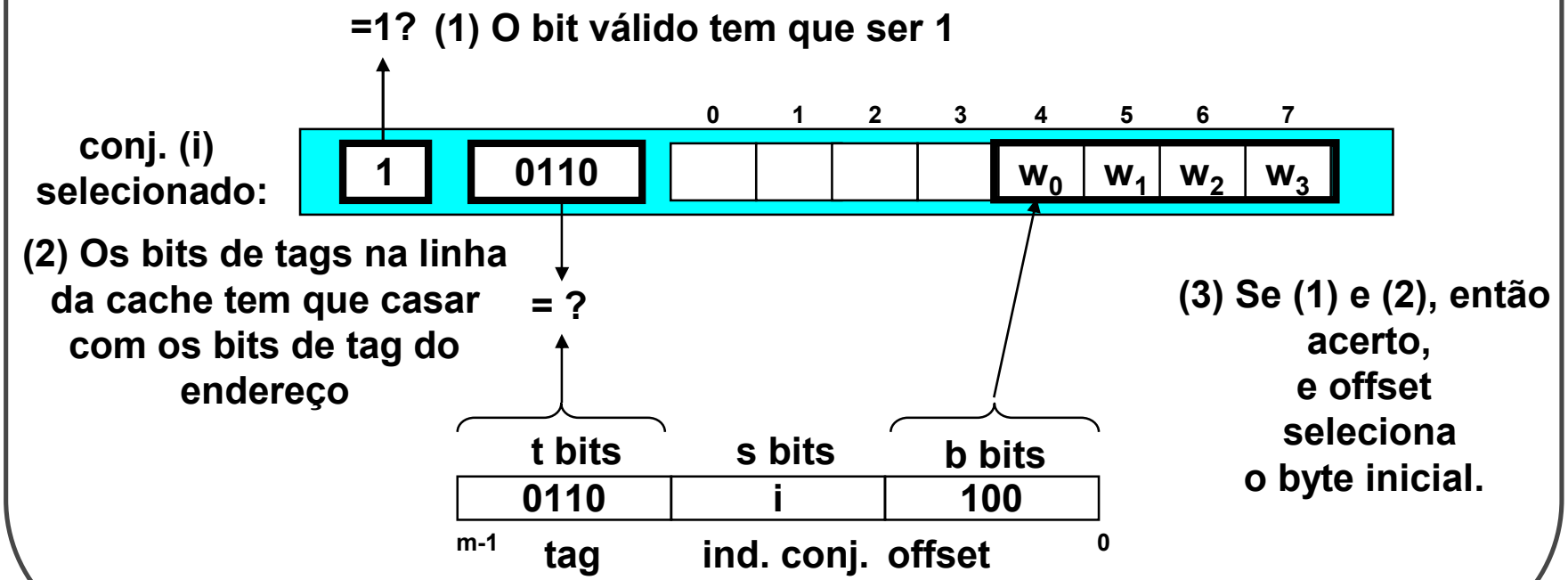
Accesso a caches mapeadas diretamente

- **Seleção do conjunto feita pelos bits de índice de conjunto**



Accesso a caches mapeadas diretamente

- **Encontre uma linha válida no conjunto selecionado com campo tag idêntico a bits de tag e extrai a palavra desejada**



Exemplo de cache mapeada diretamente

t=1	s=2	b=1
x	xx	x

M=endereço de 16 bytes, B=2 bytes/bloco,
S=4 conj., E=1 linha/conj.
Acessos a endereços (leituras):
0 [0000_2], 1 [0001_2], 13 [1101_2], 8 [1000_2], 0 [0000_2]

0 [0000_2] *(miss)*

(1)

v	tag	data
1	0	M[0-1]

13 [1101_2] *(miss)*

(3)

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

8 [1000_2] *(miss)*

(4)

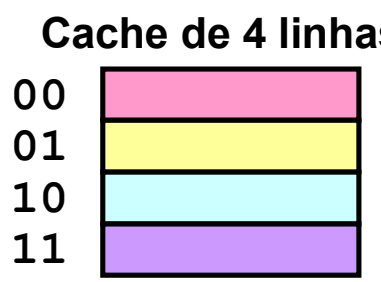
v	tag	data
1	1	M[8-9]
1	1	M[12-13]

0 [0000_2] *(miss)*

(5)

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

Porque utilizar bits do meio como índice?

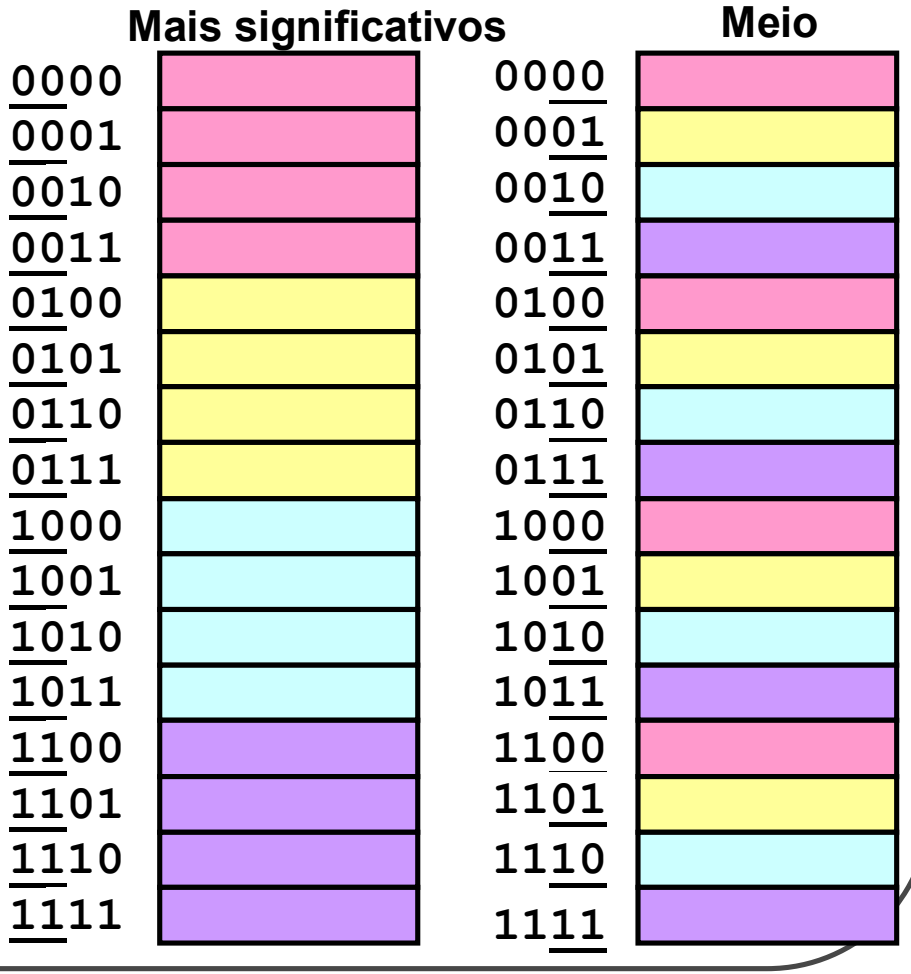


Indexação por bits mais significativos

- Linhas de memória adjacentes podem ser mapeadas no mesmo lugar (mau uso da localidade espacial)

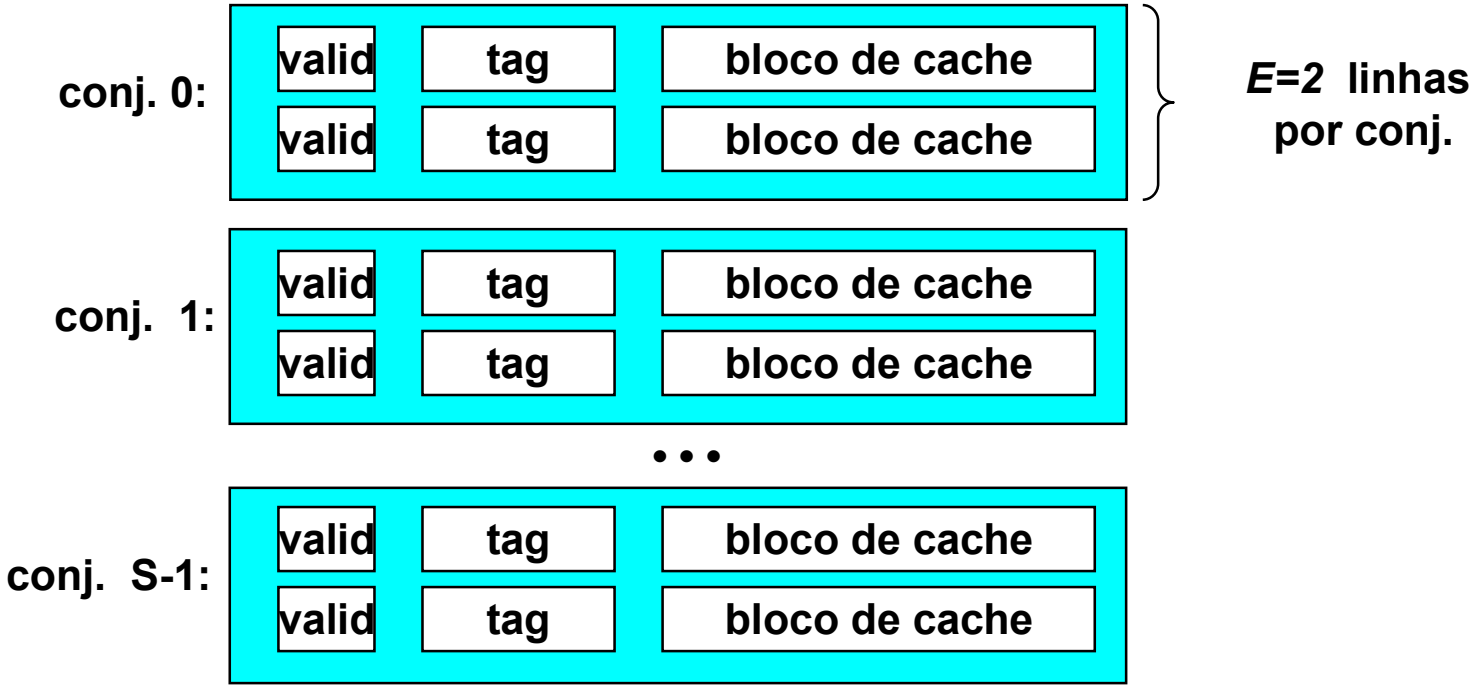
Indexação por bits do meio

- Linhas consecutivas da memória são mapeadas em linhas diferentes da cache



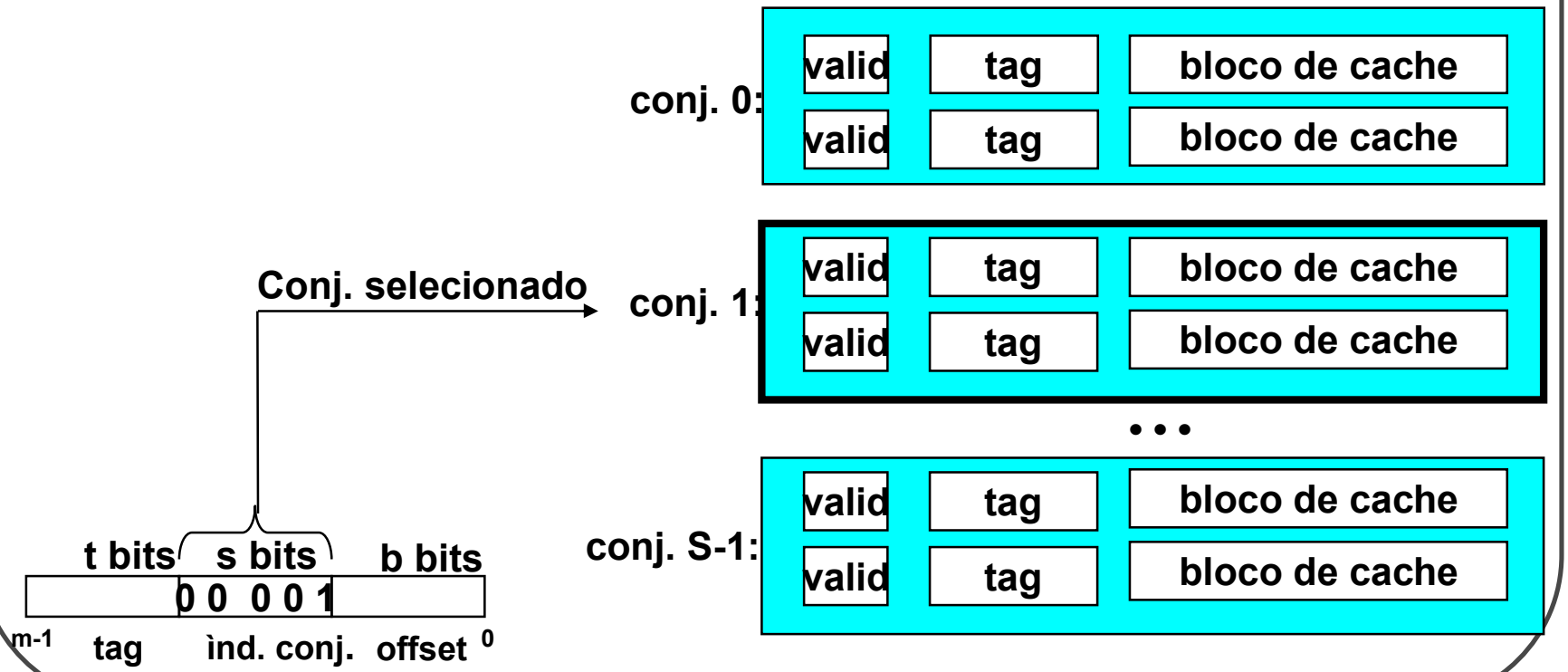
Caches associativas por conjunto

- **Caracterizadas por mais de uma linha no conjunto**



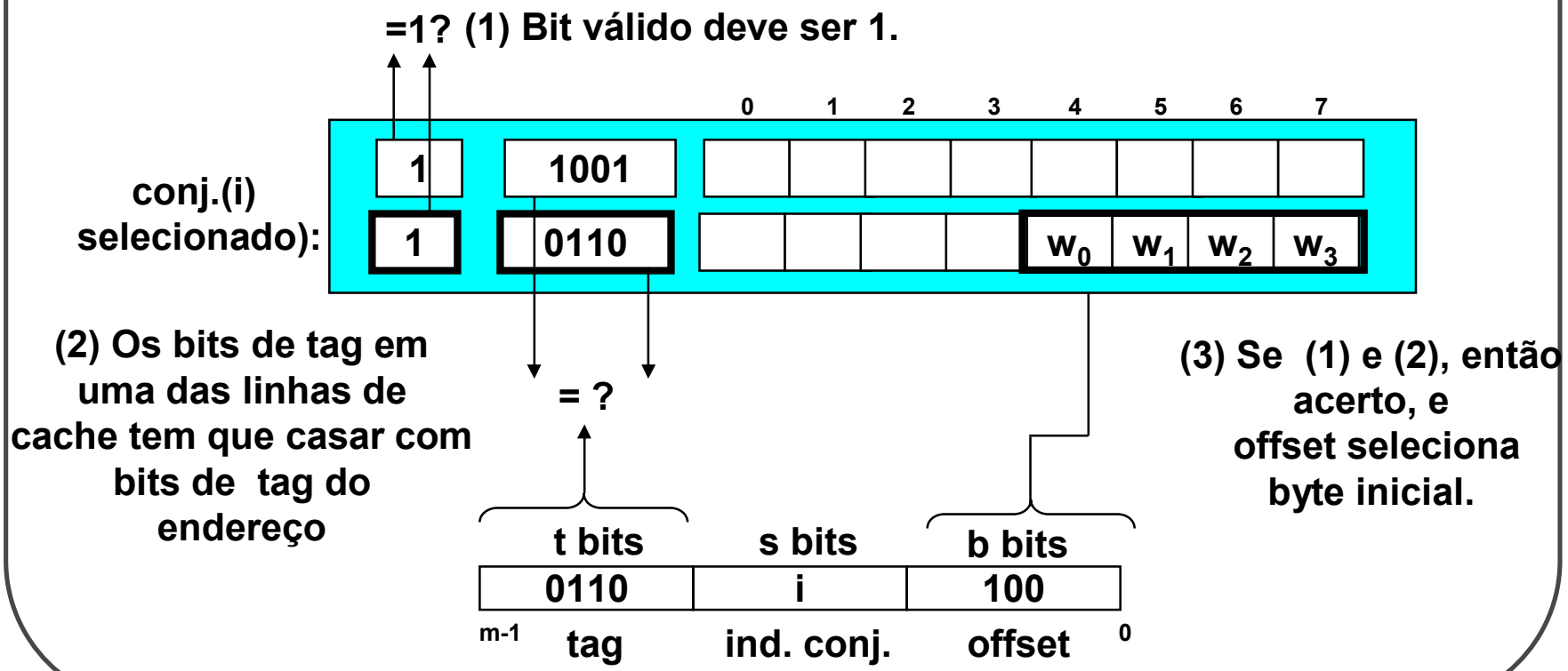
Acesso a caches associativas por conjuntos

- **Seleção do conjunto é igual à memória mapeada diretamente**



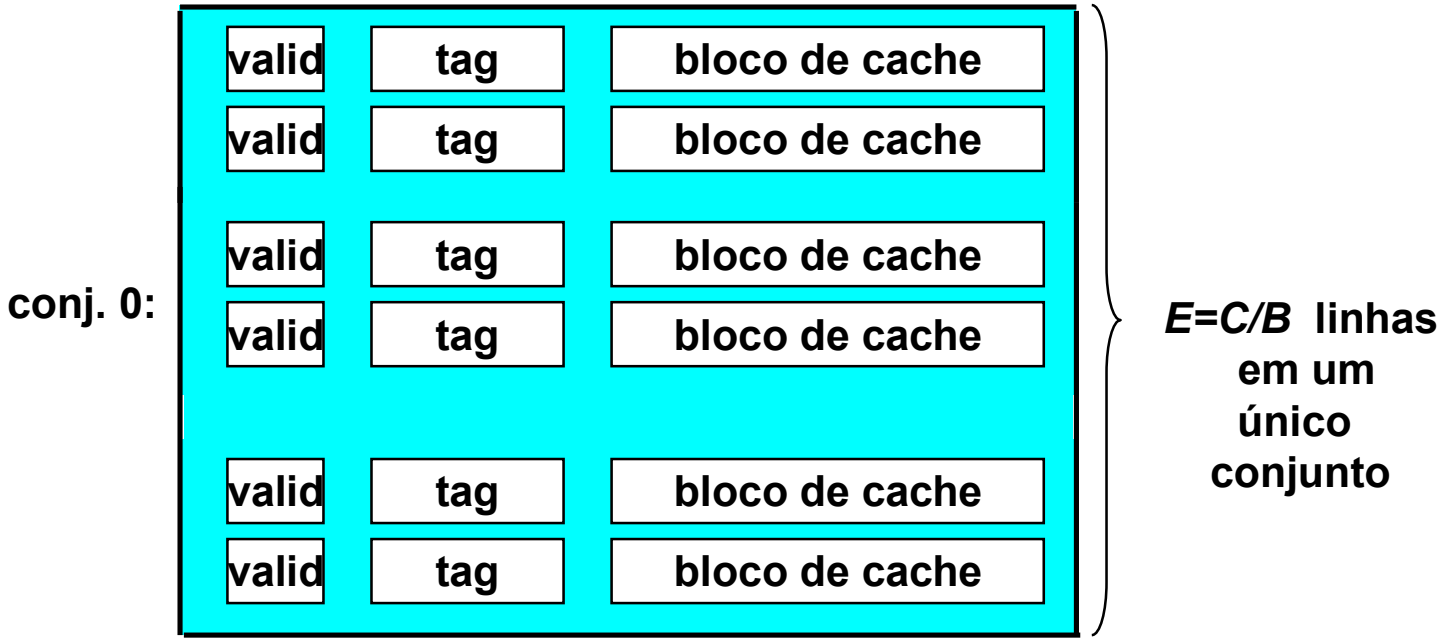
Acesso a caches associativas por conjunto

- **Compara o tag de cada linha válida do conjunto selecionado**



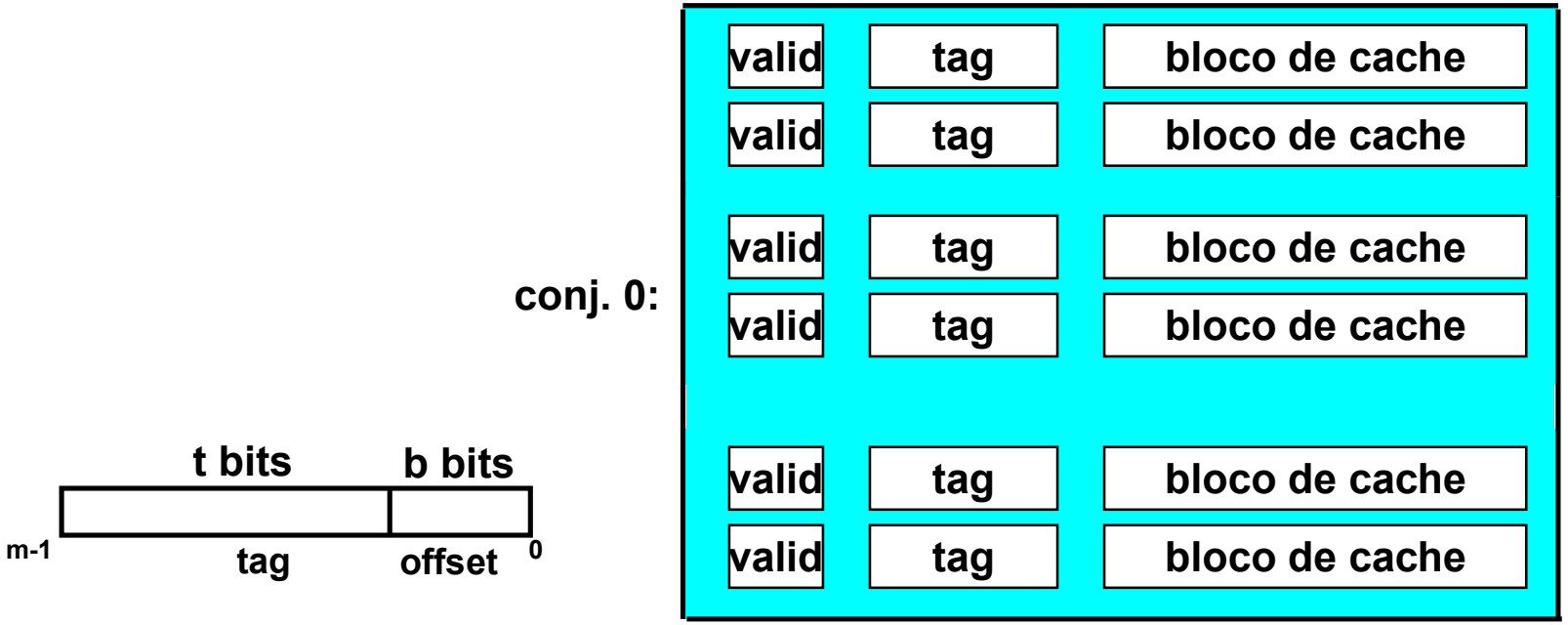
Caches totalmente associativas

- **Caracterizadas por um único conjunto conter todas as linhas**



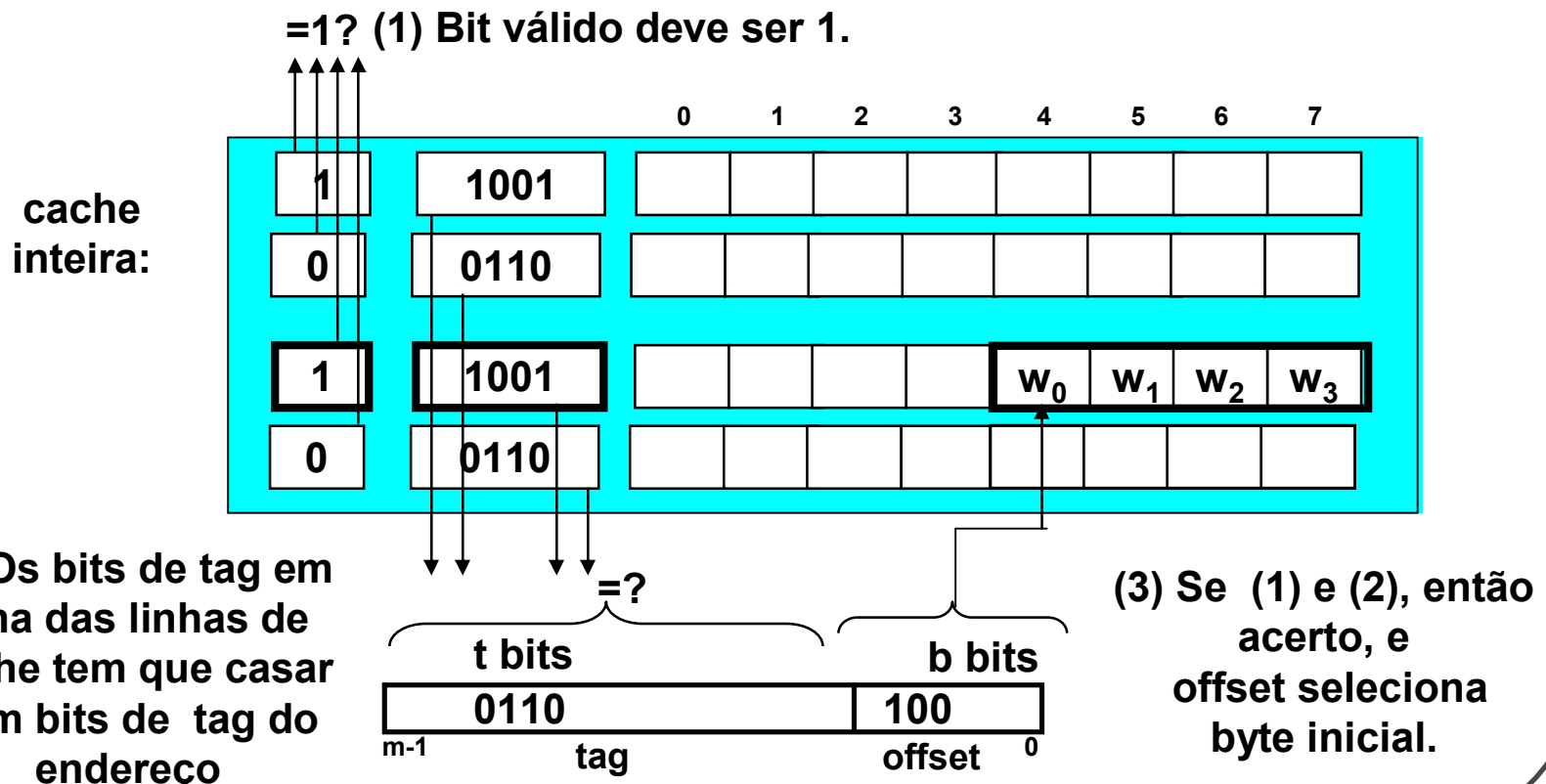
Acesso a caches totalmente associativas

- **Conjunto 0 é sempre selecionado**



Acesso a caches totalmente associativas

- Compara tag do endereço com campo tag da cache



Algoritmo de substituição de dados

- **Consiste em determinar qual o bloco da memória cache deve ser retirado para ceder lugar a outro por ocasião de uma falta**
- **No mapeamento direto não há opção!**
- **Nos demais mapeamentos …**
  - LRU - *Least Recently Used*
  - FIFO - *First In First Out*
  - LFU - *Least Frequently Used*
  - Escolha aleatória (*)

Política de escrita I

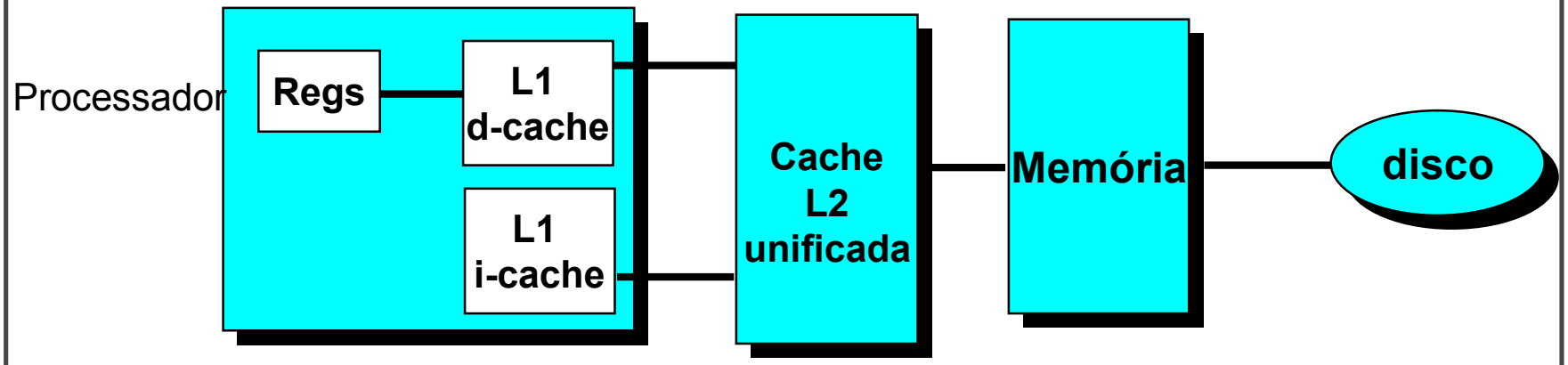
- **A escrita é sempre realizada na cache pela CPU. Quando deve ser realizada na memória principal?**
- **Problema: vários processos em várias CPU ou dispositivos de E/S podem acessar um mesmo bloco na MP**
  - valores diferentes para um mesmo dado!
- **Políticas de escrita...**
  - *write through*
  - *write back*
  - *write once*

Política de escrita II

- **Escrita em ambas (*write through*)**
  - Sempre que se escreve na cache, escreve-se na memória principal
  - Pode haver queda no desempenho
- **Escrita somente no retorno (*write back*)**
  - Escreve apenas quando o bloco é substituído: há bit de alteração
  - A memória principal pode ficar desatualizada (ex: E/S via DMA)
- **Escrita uma vez (*write once*)**
  - É utilizada quando há múltiplas CPUs
  - A CPU que primeiro alterar o bloco autaliza-o na memória local, avisando às demais
  - As demais CPUs não utilizam mais o dado da cache
  - A atualização final ocorre na substituição

Caches multiníveis

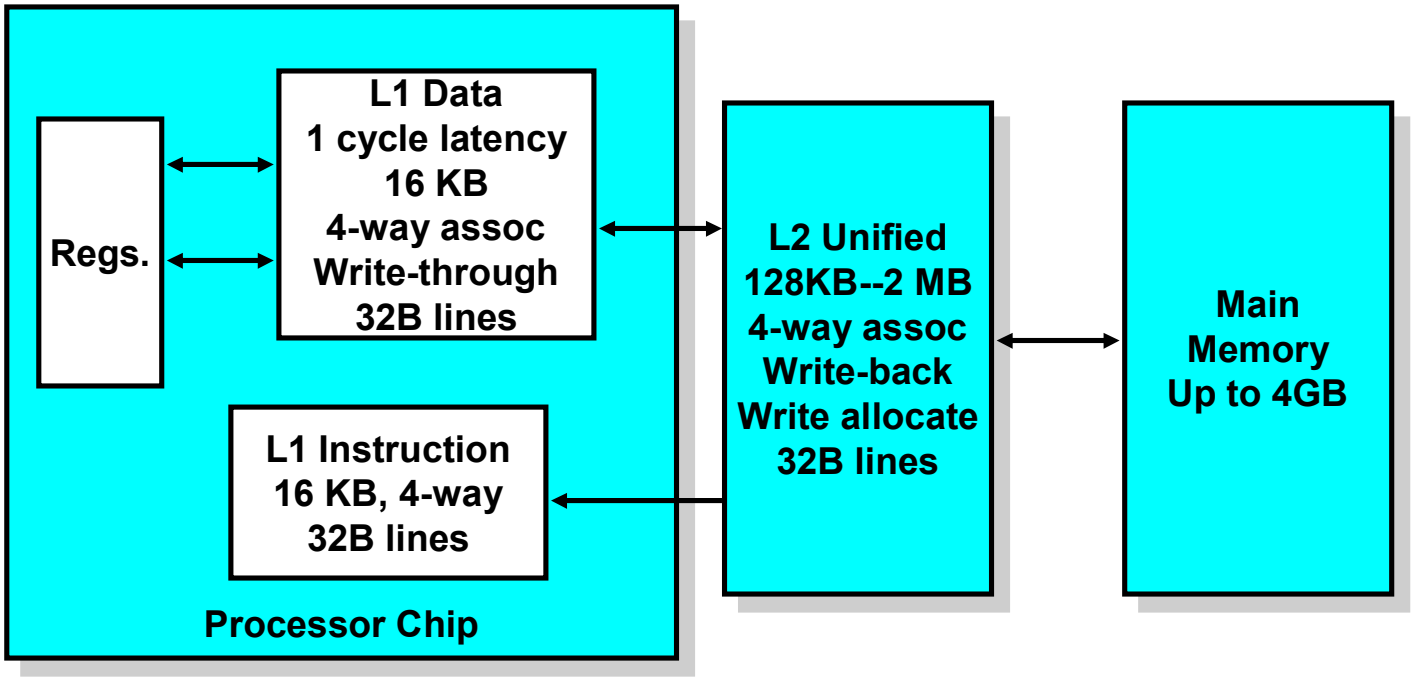
- **Caches para dados e instruções separadas ou juntas**



tamanho:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
velocidade:	3 ns	3 ns	6 ns	60 ns	8 ms
$/Mbyte:			$100/MB	$1.50/MB	$0.05/MB
tam. linha:	8 B	32 B	32 B	8 KB	

maior, mais lenta, mais barata

Hierarquia do Pentium



Características de código eficiente para caching

- **Referências repetidas a variáveis (localidade temporal)**
- **Padrões de referência com passo 1 (localidade espacial)**

Exemplo de desempenho de código em relação a caching

- **Cache inicialmente vazia, palavras de 4-bytes, blocos na cache de 4 palavras**

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Taxa de falta = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Taxa de falta = 100%

Exemplo de multiplicação de matrizes

- **Explore localidade temporal e mantenha o conj. de trabalho pequeno utilizando blocos**
- **Explore localidade espacial pelo tamanho do bloco**

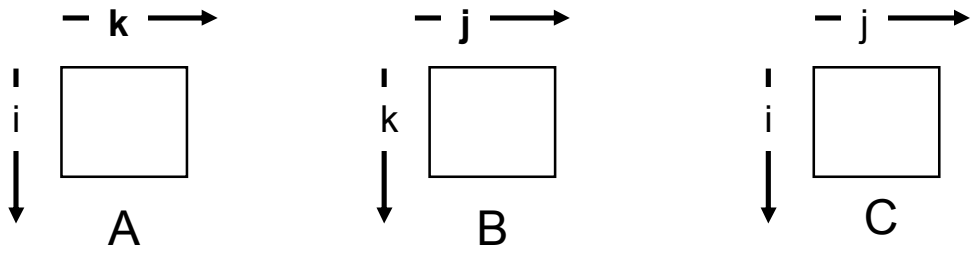
- **Descrição:**
  - Multiplique matrizes N x N
  - O(N3) operações no total
  - Acessos
    - N leituras por elemento fonte
    - N valores somados para destino

Variável ***sum*** *em registrador*

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Análise de falta em cache para multiplicação de matrizes

- **Assuma:**
  - Tamanho de linha = 32B (pode armazenar 4 64-bit palavras)
  - Dimensão N da matriz muito grande (1/N tende a 0.0)
  - Cache não tem capacidade para armazenar várias linhas
- **Método de análise:**
  - Veja o padrão de acesso do loop interno



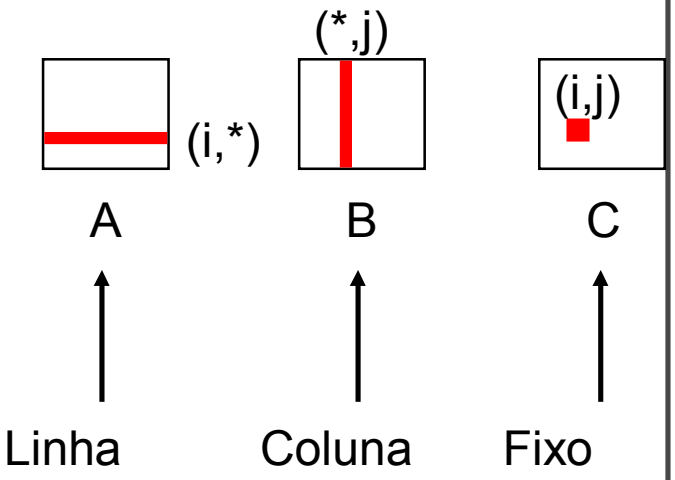
Arrays em C

- **Arrays em C são alocados por linha em lugares adjacentes na memória**
- **Acesso a colunas em uma linha:**
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - acessa elementos sucessivos
  - se tamanho de bloco (B) > 4 bytes, explora localidade espacial
    - taxa de falta compulsória = 4 bytes / B
- **Acesso a linhas em uma coluna:**
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - acessa elementos distantes, não apresenta localidade espacial
    - taxa compulsória de falta = 1 (i.e. 100%)

Multiplicação de matrizes (ijk)


```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Loop interno:



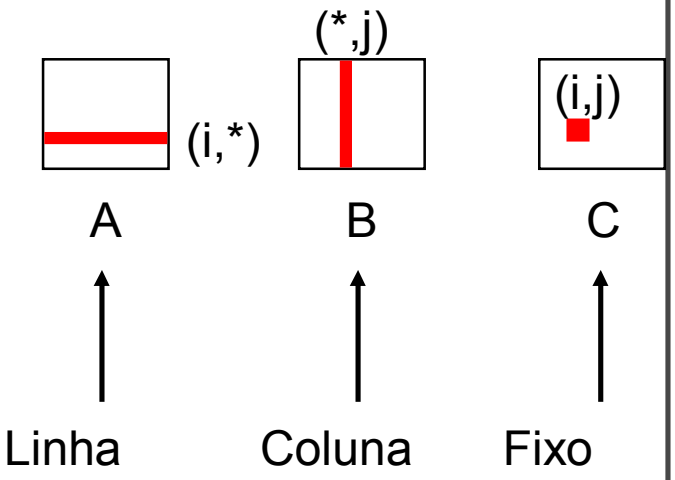
- **Faltas em cada iteração do loop interno:**

A	B	C
0.25	1.0	0.0

Multiplicação de matrizes (jik)


```
/* jik */
for (j=0; j<n; j++)  {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Loop interno:



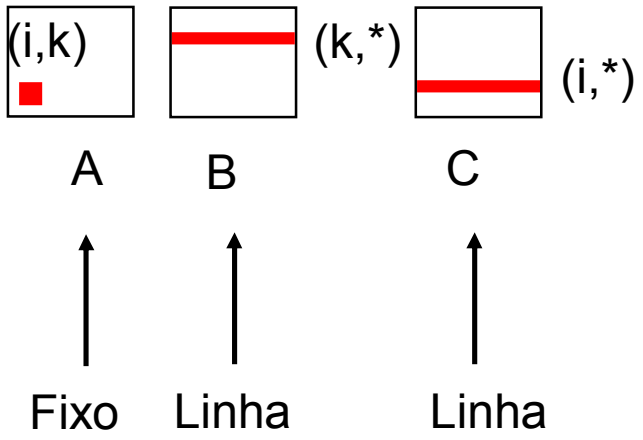
- **Faltas em cada iteração do loop interno:**

A	B	C
0.25	1.0	0.0

Multiplicação de matrizes (kij)


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

Loop interno:



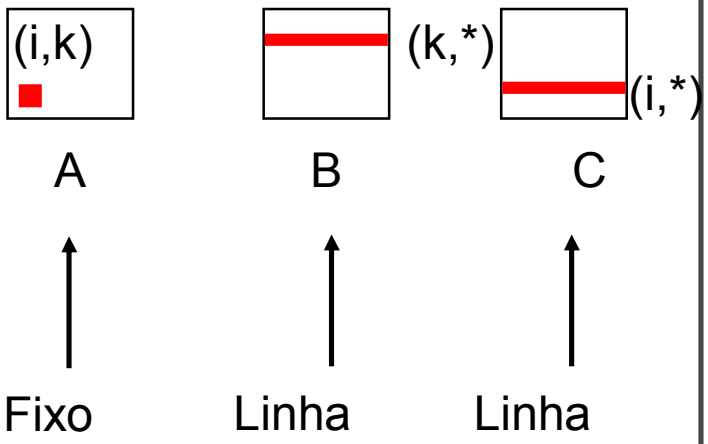
- **Faltas em cada iteração do loop interno:**

A	B	C
0.0	0.25	0.25

Multiplicação de matrizes (ikj)


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

Loop interno:



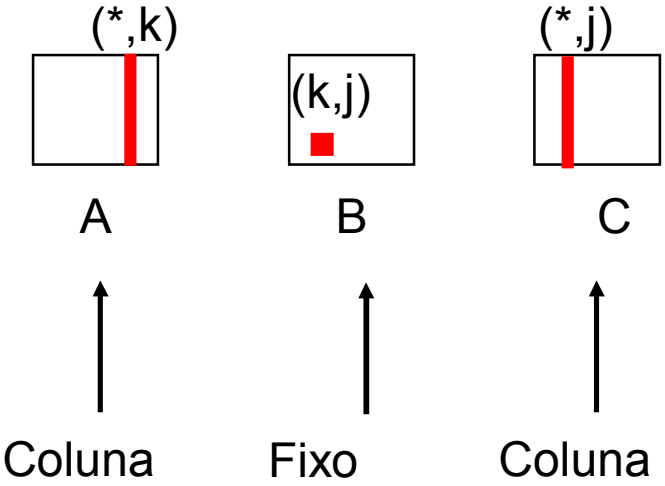
- **Faltas em cada iteração do loop interno:**

A	B	C
0.0	0.25	0.25

Multiplicação de matrizes (jki)


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

Loop interno:



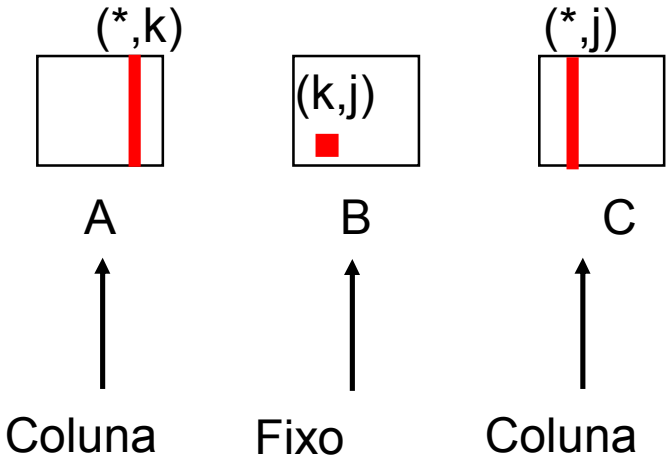
- **Faltas em cada iteração do loop interno:**

A	B	C
1.0	0.0	1.0

Multiplicação de matrizes (kji)


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

Loop interno:



- **Falta em cada iteração do loop interno**

A	B	C
1.0	0.0	1.0

Multiplicação de matrizes

ijk (& jik):
- 2 loads, 0 stores
- faltas/iter = **1.25**

```
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

kij (& ikj):
- 2 loads, 1 store
- faltas/iter = **0.5**

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

jki (& kji):
- 2 loads, 1 store
- faltas/iter = **2.0**

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