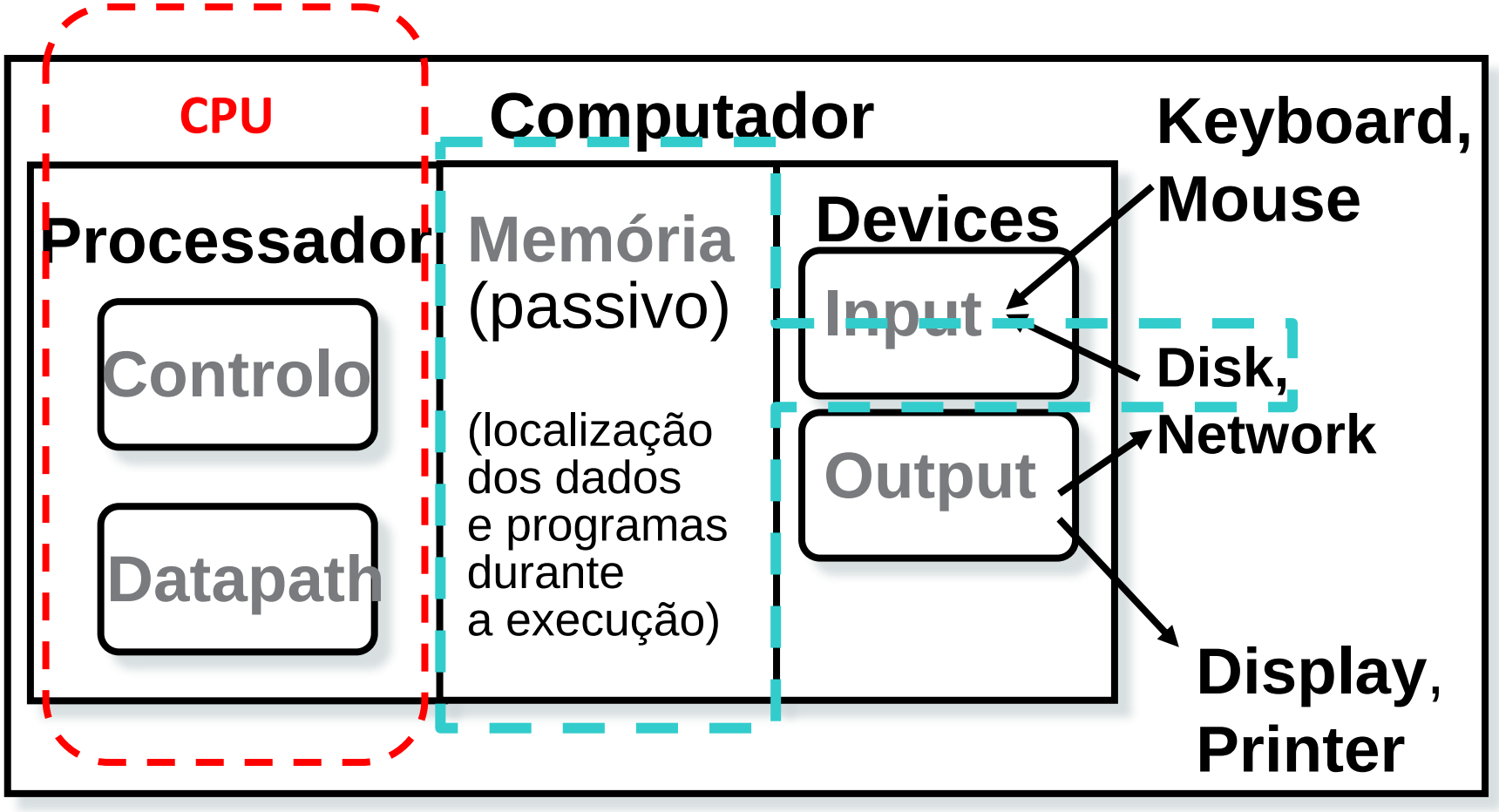
# Introdução à Arquitetura de Computadores
# - Hierarquia de Memória -

Arquitetura de Computadores 2024/2025

# Recordando…

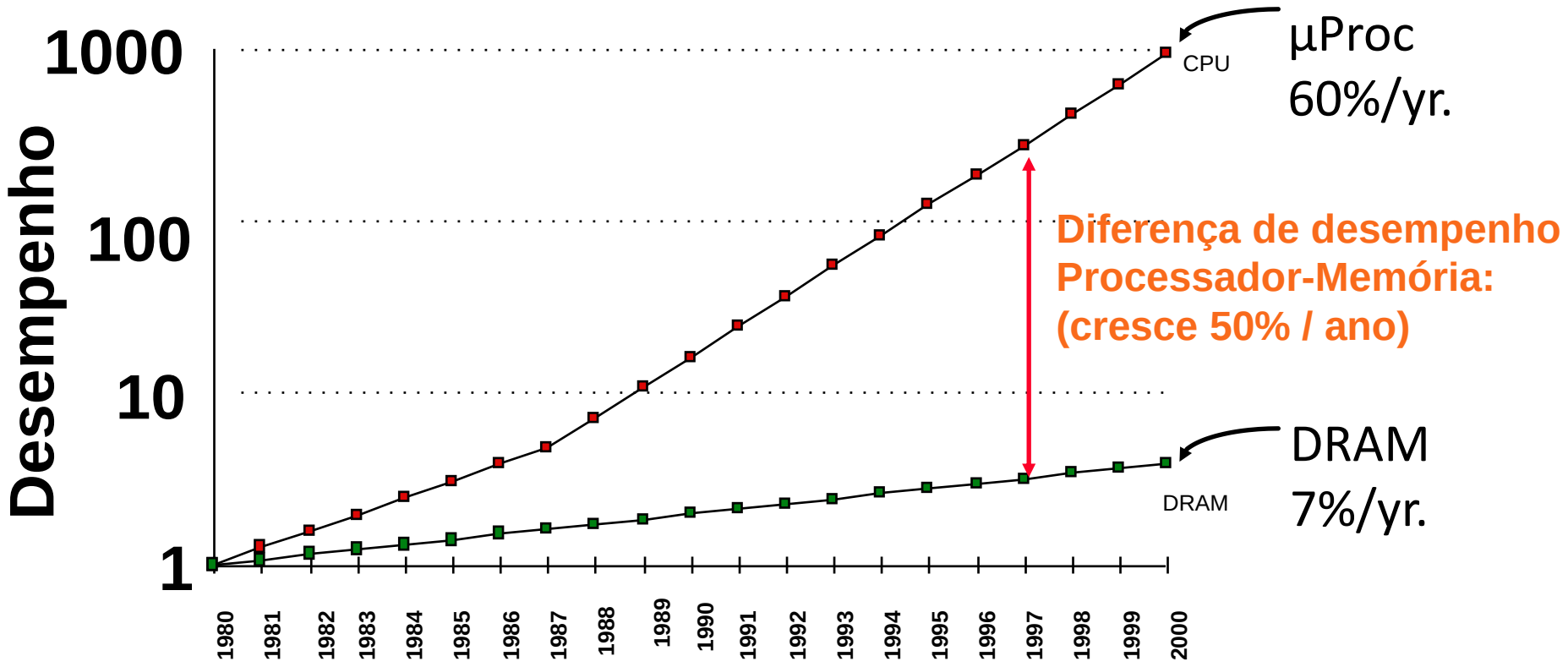

CPU
Computador
Keyboard,
Mouse
Processador
Memória
(passivo)
Devices
Input
Controlo
Disk,
Network
(localização
dos dados
e programas
durante
a execução)
Output
Datapath
Display,
Printer

# Hierarquia de Memória

*Armazenamento nos Sistemas Informáticos:*

- Processador
  - Guarda dados em registos (~100 Bytes)
  - O acesso aos registos é feito na casa dos nanossegundos
- Memória (chamada "memória principal")
  - Muito mais capacidade do que os registos (~Gbytes)
  - Tempo de Acesso ~50-100 ns
  - Centenas de ciclos de relógio por acesso à memória?!
- Disco
  - ENORME capacidade (virtualmente ilimitado)
  - MUITO MAIS lento: na casa dos milissegundos

# Motivação: Porque utilizar caches?




- 1989 Primeiro CPU da Intel com cache incorporada
- 1998 Pentium III com dois níveis de cache incorporados

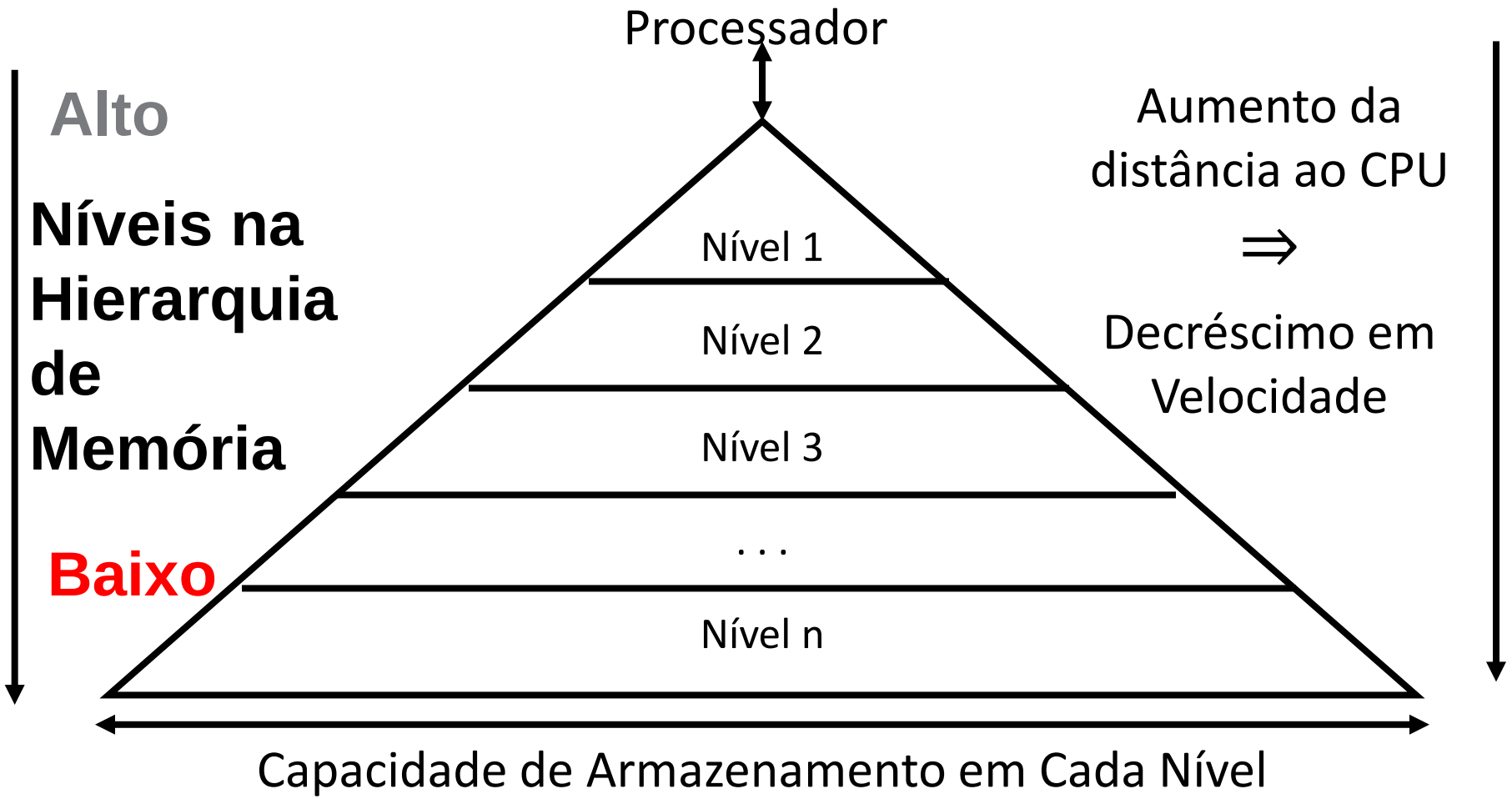
# Memória Cache

- O desfasamento entre a velocidade dos processadores e das memórias levou à introdução de um novo nível de memória: a cache

- Implementada recorrendo à mesma tecnologia de fabrico do CPU (tipicamente integrada no mesmo *chip*): mais rápida, mas mais cara do que a memória DRAM

- A cache é uma cópia de um subconjunto da memória principal

- A maior parte dos processadores tem caches separadas para instruções e dados




**À medida que descemos na hierarquia de memória, a latência sobe e o preço por bit desce**

# Hierarquia de Memória

- Se o nível está próximo do processador, ele será:
  - Mais pequeno;
  - Mais rápido;
  - Um subconjunto dos níveis mais baixos (contém dados recentemente usados)

- O nível mais baixo (tipicamente o disco) contém toda a informação disponível (ou será que ela está para além do disco?)

- A hierarquia de memória fornece ao processador a ilusão de uma memória rápida e de grande capacidade

# Pressupostos da Hierarquia de Memória

- A Cache contém cópias dos dados armazenados na memória e que estão a ser utilizados no momento

- A Memória contém cópias dos dados armazenados no disco e que estão a ser utilizados no momento

- As Caches funcionam baseadas no princípio da localidade espacial e temporal:
  - Localidade temporal: se estamos a utilizar agora é muito provável que voltemos a utilizar em breve
  - Localidade espacial: se estamos a utilizar uma zona da memória, então é bem provável que utilizemos zonas vizinhas em breve

# Projecto da Cache

- Como organizamos então a cache?
- Como é que o endereçamento de memória é mapeado?

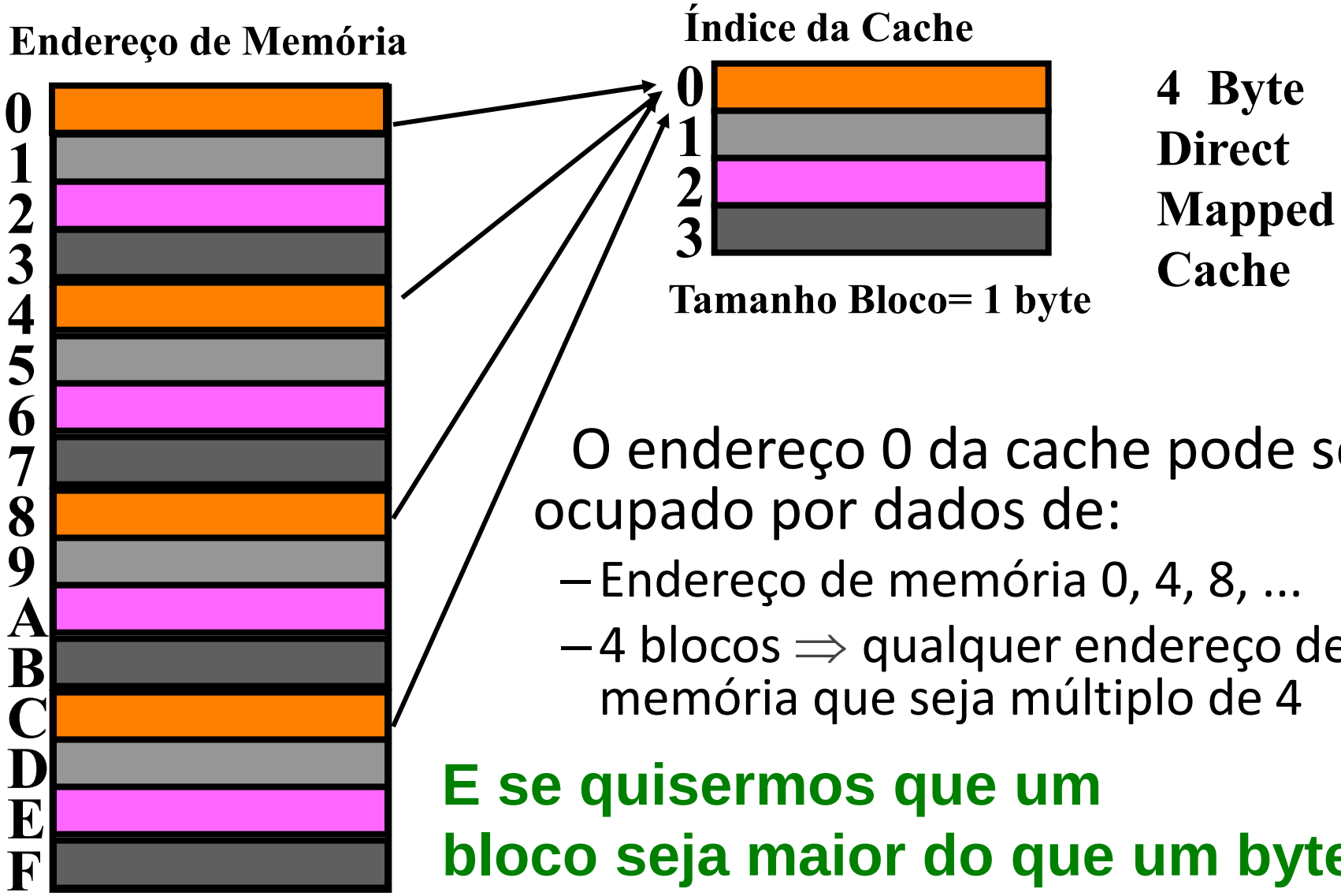
  (lembrar que a cache é um subconjunto da memória, logo múltiplos endereços podem mapear a mesma localização na cache)

- Como sabemos que elementos estão na cache?
- Como os localizamos rapidamente?

# *Direct-Mapped Cache* (1/4)

- Numa *cache de mapeamento direto*, cada endereço de memória é associado a um único bloco de memória na *cache*.
  - Precisamos apenas de verificar um único local para confirmar se os dados existem ou não na *cache*.
  - O *Bloco* é a unidade mínima de transferência entre o *cache* e a memória

# Direct-Mapped Cache (2/4)

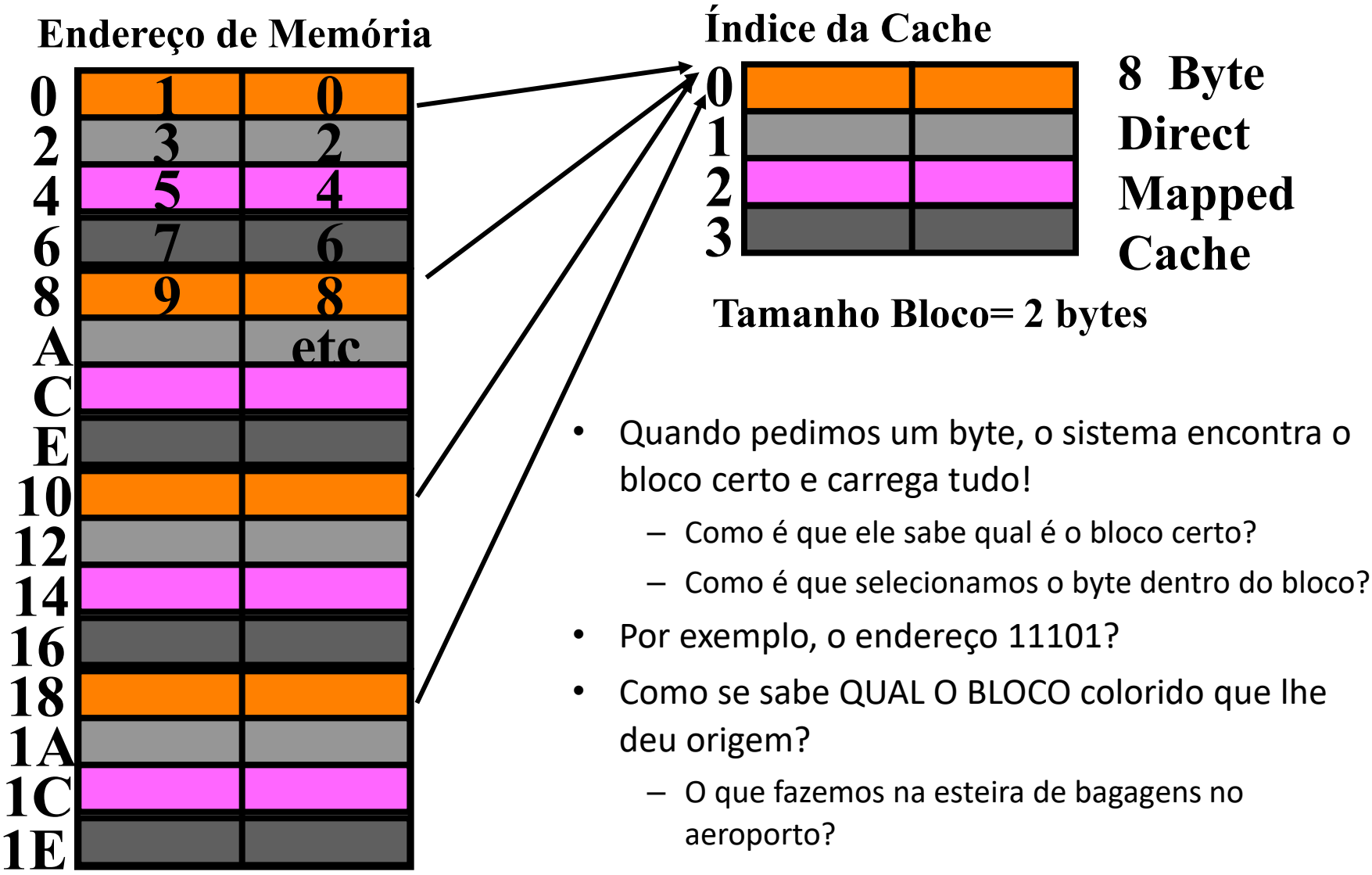



O endereço 0 da cache pode ser ocupado por dados de:

- Endereço de memória 0, 4, 8, ...
- 4 blocos ⇒ qualquer endereço de memória que seja múltiplo de 4

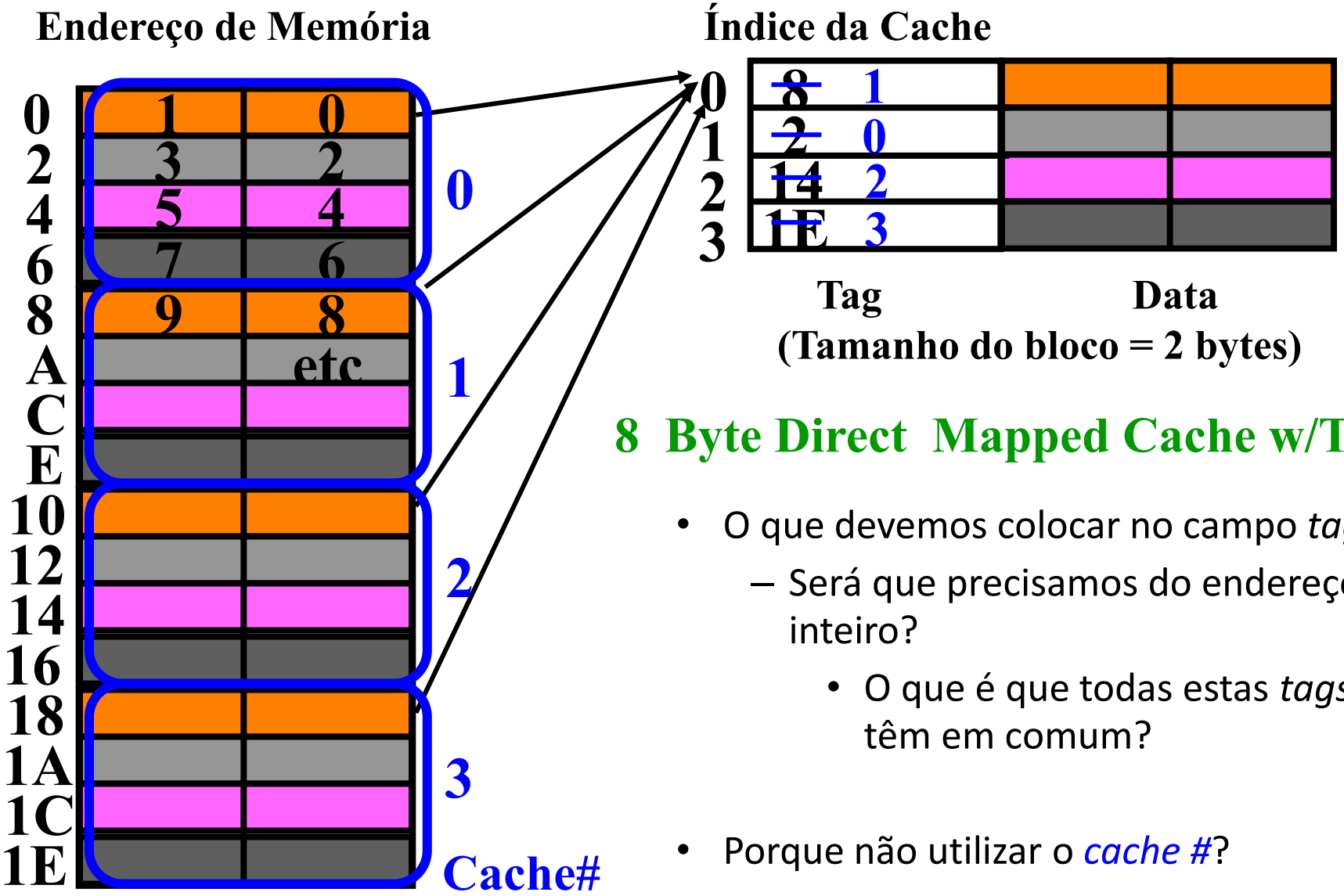
**E se quisermos que um bloco seja maior do que um byte?**

# Direct-Mapped Cache (3/4)




- Quando pedimos um byte, o sistema encontra o bloco certo e carrega tudo!
  - Como é que ele sabe qual é o bloco certo?
  - Como é que selecionamos o byte dentro do bloco?
- Por exemplo, o endereço 11101?
- Como se sabe QUAL O BLOCO colorido que lhe deu origem?
  - O que fazemos na esteira de bagagens no aeroporto?

# Direct-Mapped Cache (4/4)




**8 Byte Direct Mapped Cache w/Tag!**

- O que devemos colocar no campo *tag*?
  - Será que precisamos do endereço inteiro?
    - O que é que todas estas *tags* têm em comum?
- Porque não utilizar o *cache #*?

# Questões sobre o mapeamento direto

- Visto que vários endereços de memória são mapeados para o mesmo índice da cache, como podemos saber qual deles está lá?

- E se tivermos um tamanho de bloco > 1 byte?

- Solução: dividir o endereço de memória em três campos


**tag**
verifica de temos
o bloco correcto

**index**
selecciona
o bloco na cache

**offset**
indica o byte
dentro do bloco

# Terminologia das caches de mapeamento direto

- Index: especifica o índice da cache (em qual "linha"/bloco da cache devemos procurar)
- Offset: depois de encontramos o bloco correto, especifica qual o byte dentro do bloco que queremos
- Tag: os bits restantes são usados para identificar quais os endereços de memória que são mapeados no mesmo bloco da cache

Todos os campos são lidos como inteiros sem sinal.

# Exemplo de uma *Direct-Mapped Cache* (1/4)

- Suponha que temos uma memória *cache* de mapeamento direto com 16 KB com blocos de 16 bytes.
- Determine o tamanho dos campos de *tag*, *index* e *offset* se estivermos a utilizar uma arquitetura de 32 bits
- *Offset*
  - especifica o byte correto dentro de um bloco
  - Cada bloco contém 16 bytes

    16 bytes = $2^4$ bytes
  - Vão ser necessários 4 bits para especificar o byte dentro do bloco

# Exemplo de uma *Direct-Mapped Cache* (2/4)

- *Index*: (basicamente especifica o endereço de cada bloco na cache)
  - A *cache* contém 16 KB = $2^{14}$ bytes
  - Cada bloco contém $2^4$ bytes (16 bytes)
  - O número de blocos na *cache* será:

$$Nº\ Blocos = \frac{Tamanho\ Cache}{Nº\ Bytes\ por\ Bloco} = \frac{2^{14}}{2^4} = 2^{10}$$

  - São necessários 10 bits para especificar este número de blocos

# Exemplo de uma *Direct-Mapped Cache* (3/4)

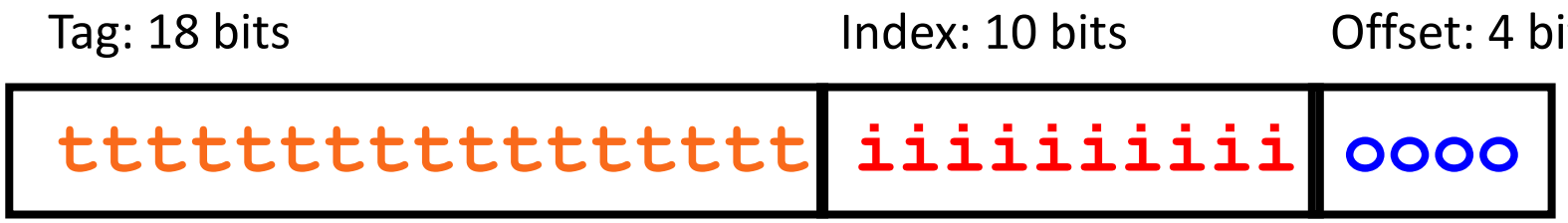
- *Tag*: (basicamente os *bits* restantes que identificam de forma única cada bloco em memória)
  - Comprimento do *tag* = Comprimento Endereço - *offset* - *index*

    = 32 - 4 - 10 bits
    = 18 bits

  - Logo o campo *tag* são os 18 bits mais à esquerda do endereço de memória

# Exemplo de uma *Direct-Mapped Cache* (4/4)

- Resultado final:




- Por que não um endereço completo de 32 *bits* como *tag*?
  - Todos os *bytes* dentro do bloco precisam do mesmo endereço (4*bits*)
  - O *index* deve ser o mesmo para todos os endereços dentro de um bloco, por isso é redundante na verificação da *tag*, portanto, pode ser ignorado para economizar memória (10 *bits*)

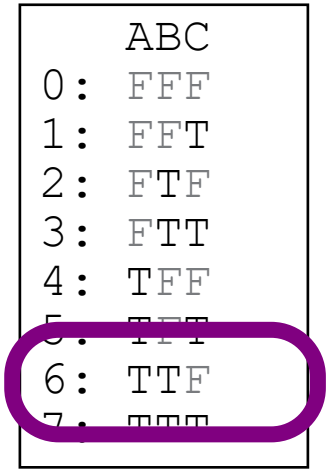
# QUIZ

A. O número de bits do *tag* depende apenas do tamanho da cache. Nunca depende do tamanho de cada bloco.

B. Se souber o tamanho da cache do seu computador então pode frequentemente fazer com que o seu código corra mais rápido.

C. As hierarquias de memória tiram partido da localidade espacial ao manter sempre os dados mais recentes próximos do processador.

É falsa porque os dados mais recentes definem localidade temporal e não espacial!



# Conclusão

- Gostaríamos de ter a capacidade do disco na velocidade do processador: infelizmente isso não é viável

- Então, criamos uma hierarquia de memória:
  - Cada nível sucessivamente inferior contém os dados "mais usados" do nível superior
  - Explora-se a localidade *temporal* e *espacial*
  - Objectivo: lidar de forma rápida com o caso mais comum, preocupando-nos com as excepções depois (princípio de design do MIPS)

# Terminologia das Caches

- Quando tentamos ler a memória, três cenários podem acontecer:

1. cache hit: o bloco de *cache* é válido e contém o endereço apropriado, então basta ler a *word* desejada da *cache*;

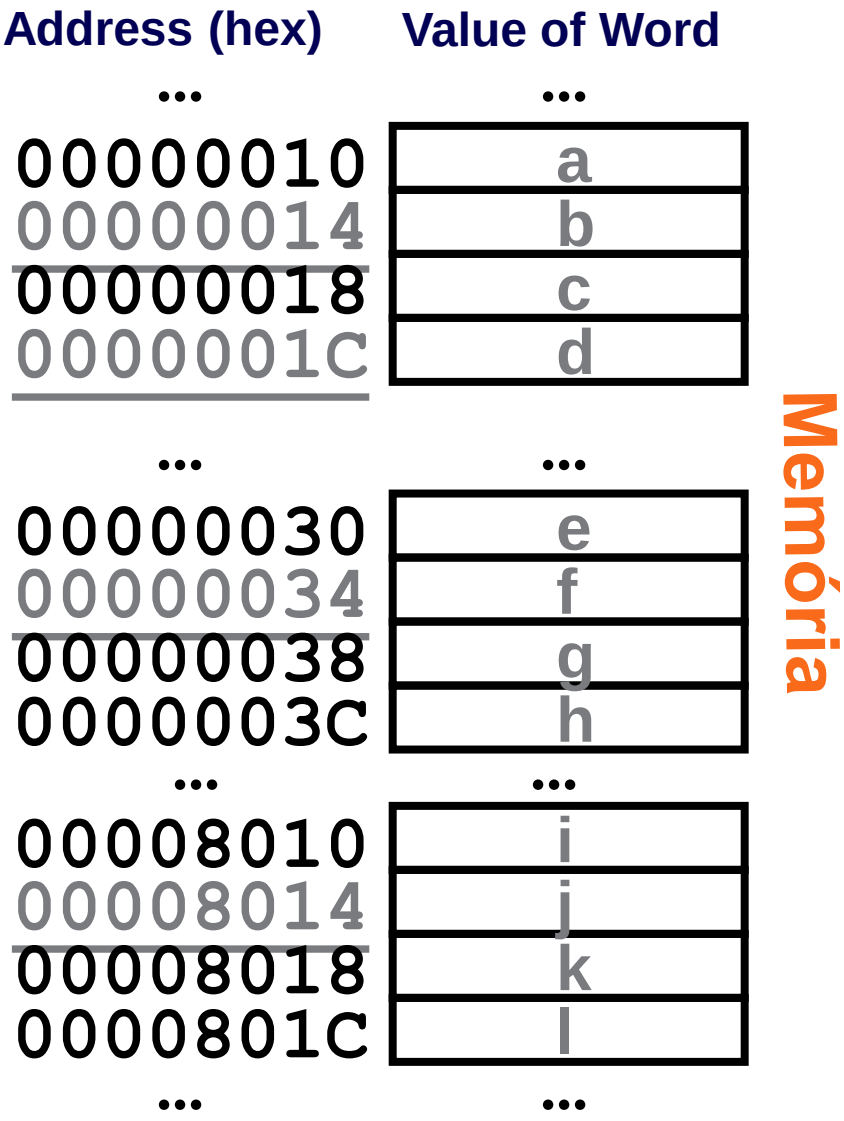
2. cache miss: nada no *cache* no bloco apropriado, então carregar o bloco da memória principal;

3. cache miss, block replacement: o bloco na *cache* não tem a *tag* certa, então substituir o bloco pelo correcto;

# Aceder a dados numa cache de mapeamento direto

- Ex.: cache com 16KB direct-mapped cache, blocos de 16 bytes

- Vamos ler os seguintes 4 endereços
  1. 0x00000014
  2. 0x0000001C
  3. 0x00000034
  4. 0x00008014



# Aceder a dados numa cache de mapeamento direto

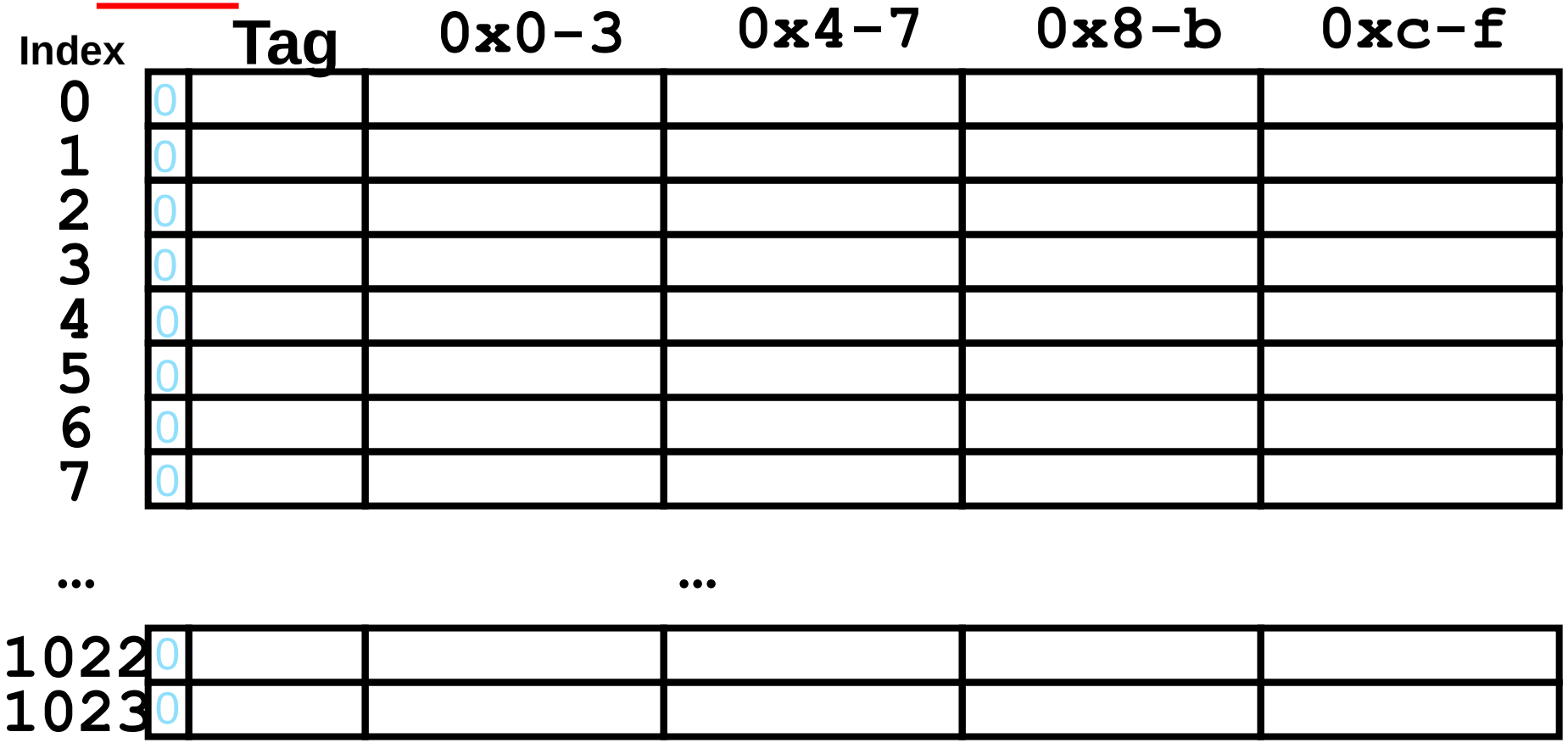
- 4 Endereços:
  - `0x00000014, 0x0000001C, 0x00000034, 0x00008014`

- 4 Endereços divididos (por conveniência) nos campos *Tag*, *Index* e *Offset*:

```
00000000000000000 0000000001 0100
00000000000000000 0000000001 1100
00000000000000000 0000000011 0100
00000000000000010 0000000001 0100
      Tag              Index    Offset
```

# 16 KB Direct Mapped Cache, 16B blocks

- Valid bit: determina se nesta linha o valor armazenado é válido (quando o computador é ligado pela primeira vez, todas as entradas são inválidas)



# 1. Ler 0x00000014

- 00000000000000000000 0000000001 0100

Tag Field  Index Field  Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 0 | | | | | |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# O endereço especifica o bloco 1 (0000000001)

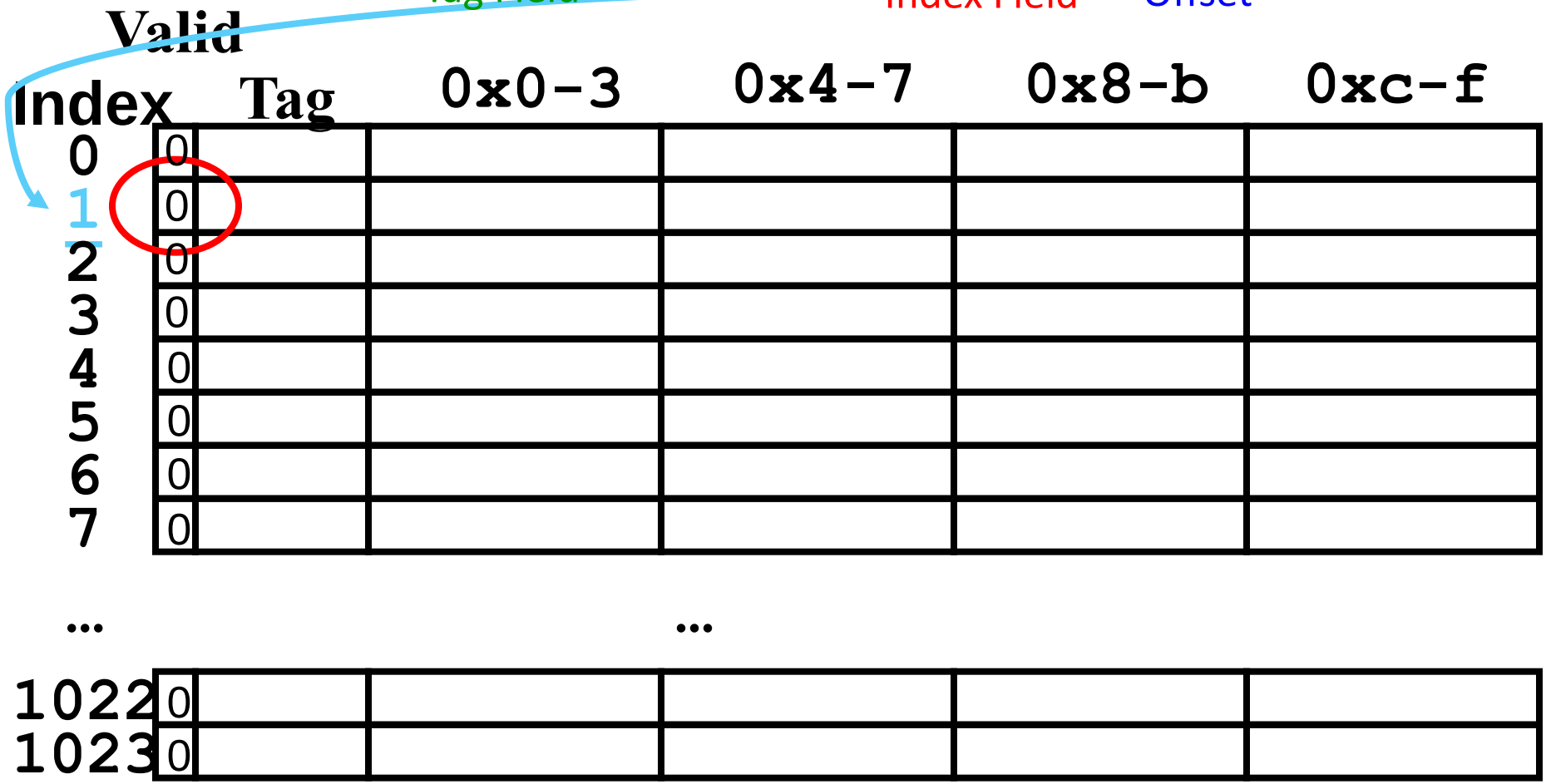
- 00000000000000000000 0000000001 0100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 0 | | | | | |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# O bloco não é válido - *Cache Miss*

- 00000000000000000000 0000000001 0100

Tag Field | Index Field | Offset



# Carregamos o bloco e colocamos o *valid bit* a 1

- 00000000000000000000 0000000001 0100

Tag Field | Index Field | Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# Lemos o *offset* 4 da cache, devolve a word *b*

- 00000000000000000000 0000000001 0100

Tag Field | Index Field | Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | | ... | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# 2. Ler 0x0000001C = 0…00 0..001 1100

- 00000000000000000000 0000000001 1100

Tag Field | Index Field | Offset

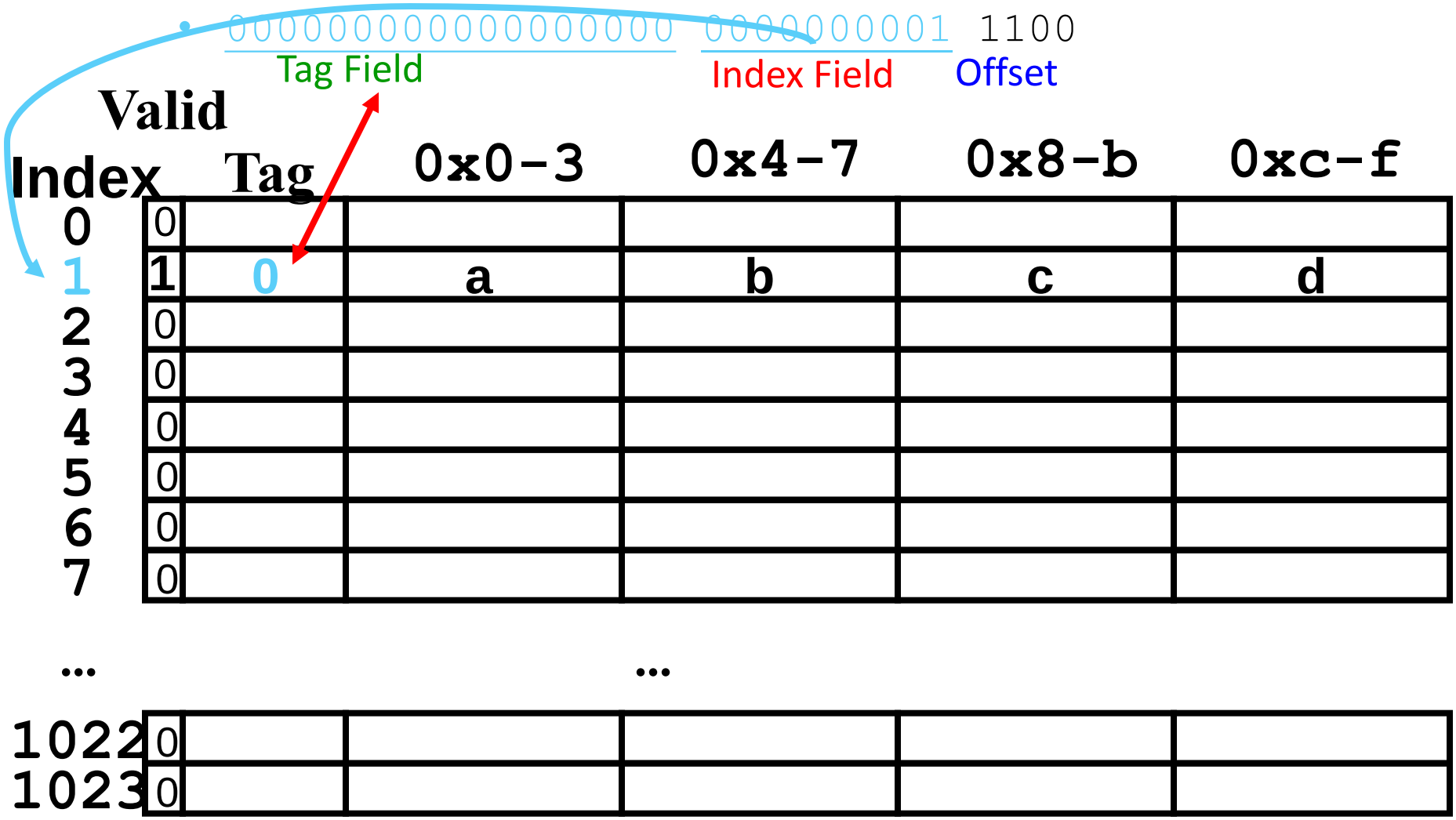
| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| … | | | … | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# O *Index* é válido

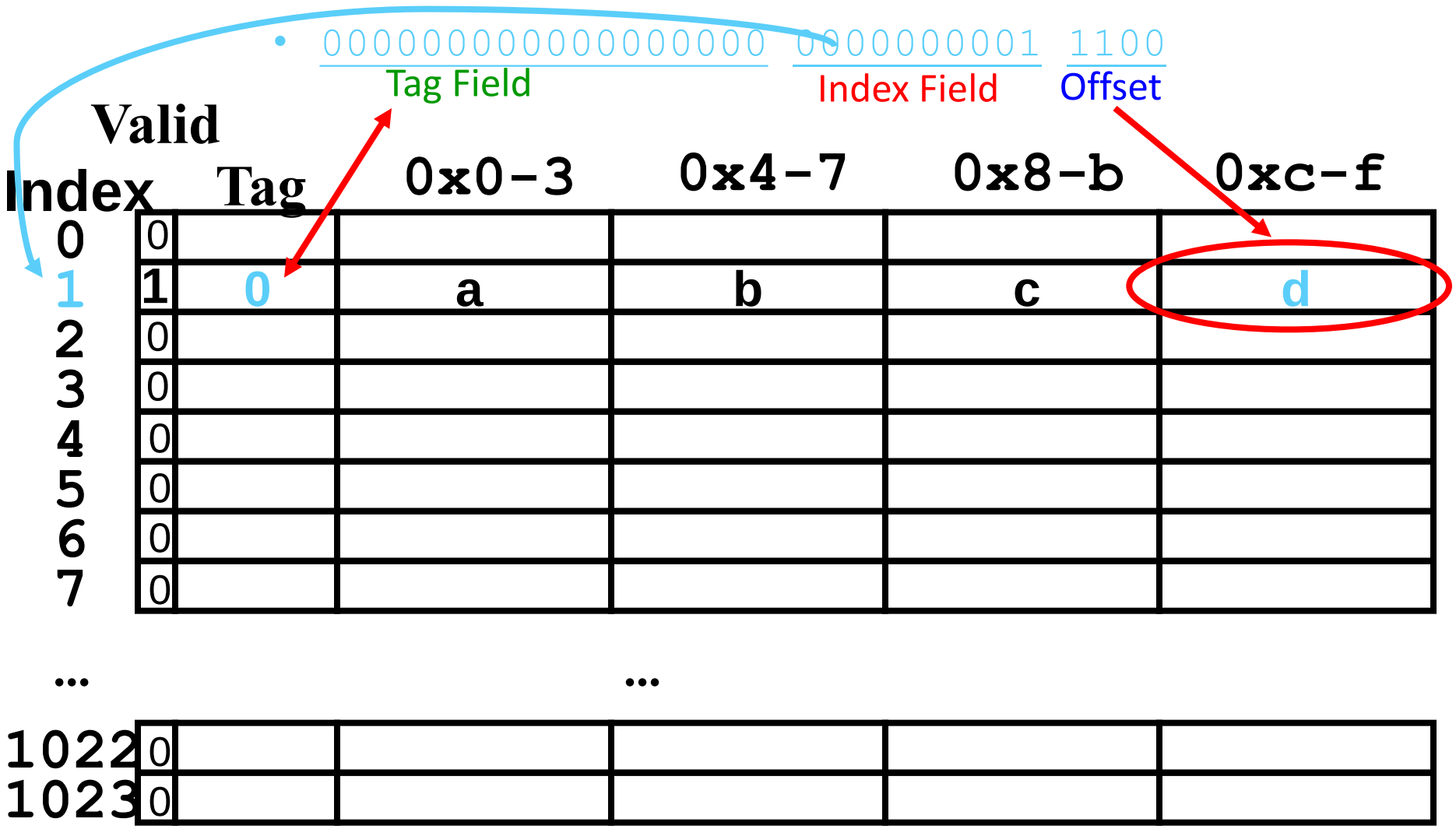
- 00000000000000000000 0000000001 1100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# *Index* válido e *Tag* correcta - *Cache Hit*



# Index válido e Tag correcta, devolve a word d



# 3. Ler 0x00000034 = 0…00 0..011 0100

- 00000000000000000000 0000000011 0100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| … | | | … | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# O endereço especifica o bloco 3

- 00000000000000000000 0000000011 0100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# Bloco não válido - *Cache Miss*

- 00000000000000000000 0000000011 0100

Tag Field | Index Field | Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# Carrega esse bloco para a cache e devolve a *word* f

- 00000000000000000000 0000000011 0100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | e | f | g | h |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# 4. Ler 0x00008014 = 0…10 0..001 0100

- 00000000000000010 0000000001 0100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | e | f | g | h |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| … | | | | … | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# Verificamos o Bloco 1 da Cache, *Valid Bit* Ok

- 00000000000000010 0000000001 0100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | e | f | g | h |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# As *Tags* não coincidem (0 != 2) - *Cache Miss BR*

- 00000000000000010 0000000001 0100
  Tag Field | Index Field | Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 0 | a | b | c | d |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | e | f | g | h |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# Substituir o bloco 1 e actualizar a *tag*

- 00000000000000010 0000000001 0100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 2 | i | j | k | l |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | e | f | g | h |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | ... | | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# Devolver word j

- 0000000000000010 0000000001 0100
  Tag Field — Index Field — Offset

| Index | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 2 | i | j | k | l |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | e | f | g | h |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | | ... | | |
| 1022 | 0 | | | | | |
| 1023 | 0 | | | | | |

# Outro exemplo. O que acontece?

- Escolher entre: Cache: Hit, Miss, Miss w. replace
  Valor devolvido: a ,b, c, d, e, ..., k, l
- Ler endereço 0x00000030 ? `00000000000000000 0000000011 0000`
  - Cache Hit; Devolve e
- Ler Endereço 0x0000001c ? `00000000000000000 0000000001 1100`
  - Cache Miss with Block Replacement

| | Valid | Tag | 0x0-3 | 0x4-7 | 0x8-b | 0xc-f |
|---|---|---|---|---|---|---|
| 0 | 0 | | | | | |
| 1 | 1 | 2 | i | j | k | l |
| 2 | 0 | | | | | |
| 3 | 1 | 0 | e | f | g | h |
| 4 | 0 | | | | | |
| 5 | 0 | | | | | |
| 6 | 0 | | | | | |
| 7 | 0 | | | | | |
| ... | | | | ... | | |

# O que fazer num *write hit*?

- Write-through
  - atualizar a *word* ou *byte* no bloco de cache e a *word* ou *byte* correspondente na memória
- Write-back
  - Actualizar a *word* apenas no bloco da *cache*
  - Permitir que a *word* na memória fique "obsoleta"

  ⇒Adicione um ‘dirty bit' a cada bloco, indicando que a memória precisa de ser atualizada quando o bloco for substituído

  ⇒ Os sistemas operativos fazem o *flush* da *cache* nas operações de I/O…
- Trade-offs no desempenho?

# Trade-off no Tamanho do Bloco (1/3)

- Benefícios de um tamanho de Bloco maior:
  - Localidade Espacial: se acedermos a uma determinada palavra, muito provavelmente acederemos a palavras próximas de seguida
  - Na execução de programas ao executarmos uma determinada instrução, é muito provável que também executemos as próximas de seguida.
  - Funciona também muito bem em acessos sequenciais como por exemplo em tabelas

# Trade-off no Tamanho do Bloco (2/3)

- Desvantagens de um tamanho de Bloco maior
  - Tamanho de bloco maior significa maior penalização no caso de um *miss* (*miss penalty)*
    - em caso de falha, leva mais tempo para carregar um novo bloco do próximo nível de memória
  - Se o tamanho do bloco for muito grande em relação ao tamanho da cache, então há poucos blocos
    - Resultado: a *miss rate* aumenta
- Em geral pretende-se minimizar o **tempo médio de acesso à memória** ou *Average Memory Access Time* (AMAT)

  = Hit Time + Miss Penalty x Miss Rate

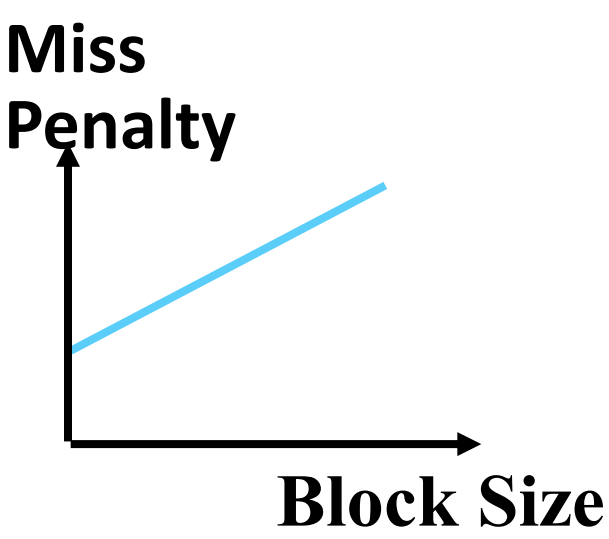
# Trade-off no Tamanho do Bloco (3/3)

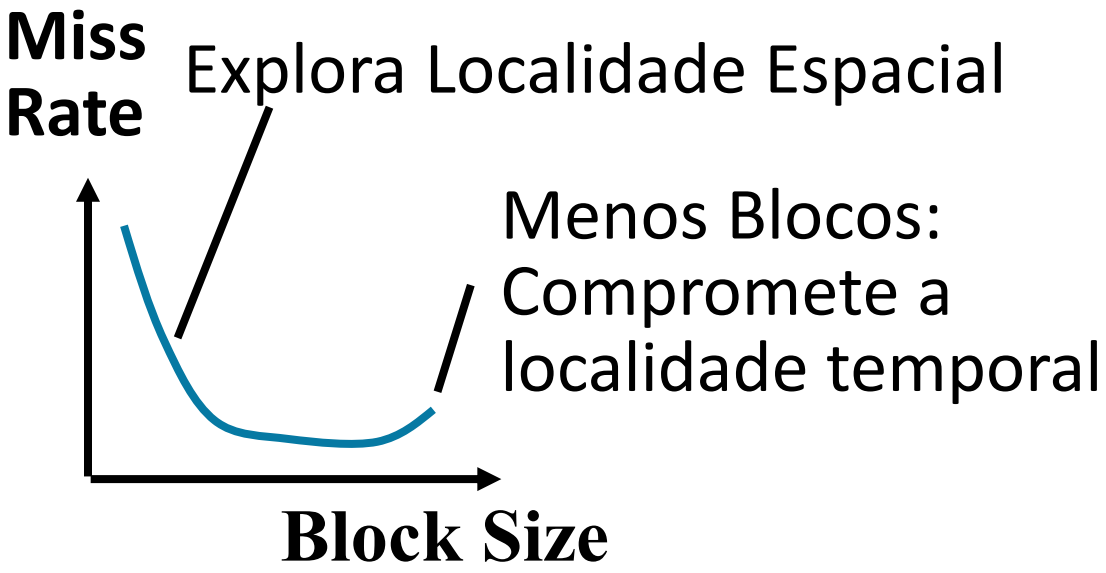
- *Hit Time* = tempo necessário para encontrar e recuperar dados da *cache* de nível atual
- *Miss Penalty* = tempo médio para recuperar dados em uma falha no nível atual da *cache* (inclui a possibilidade de perdas em níveis sucessivos da hierarquia de memória)
- *Hit Rate* = % de pedidos que são correspondidos no *cache* de nível atual
- *Miss Rate* = 1 - *Hit Rate*

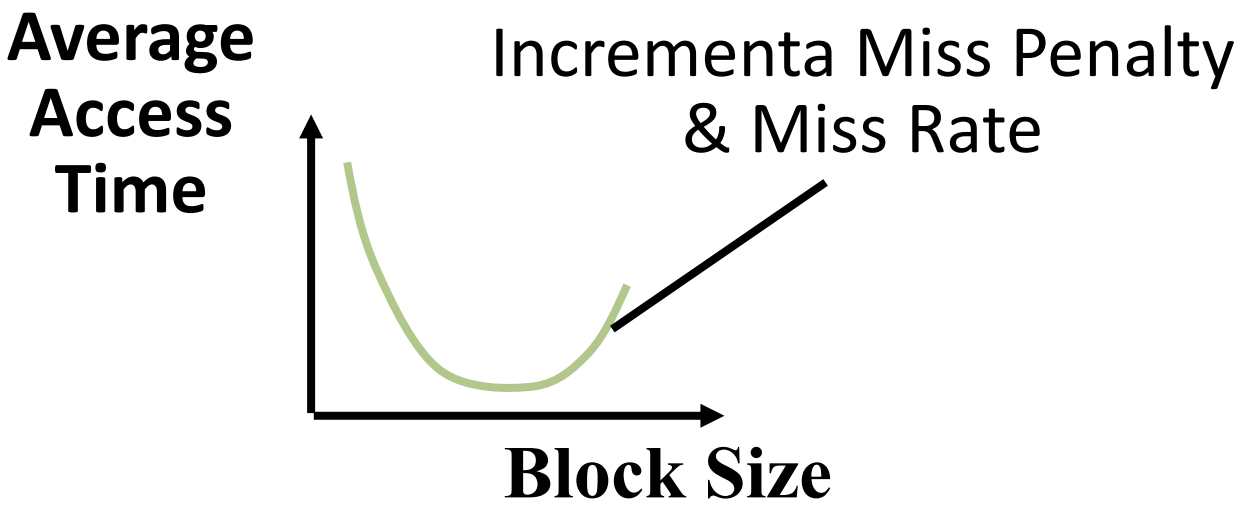
# Exemplo Extremo : Um único Bloco


- Tamanho da Cache = 4 bytes  Tamanho do Bloco = 4 bytes
  - Apenas **UMA** entrada (linha) na cache!
- Se acedermos a um determinado endereço, provavelmente ele voltará a ser acedido de novo em breve
  - Mas será bastante improvável ser acedido novamente de imediato!
- O próximo acesso será provavelmente um *miss*
  - Carregar um novo bloco para a *cache*, com o *overhead* associado.
  - Um pesadelo para o projetista da cache: Efeito Ping Pong

# Trade-off no Tamanho do Bloco. Conclusão



# Tipos de *Cache Misses* (1/2)

- Modelo dos "Três Cs" para as falhas na cache

- 1st C: **C**ompulsory Misses
  - Ocorre quando um programa é iniciado pela primeira vez
  - A *cache* não contém nenhum dos dados desse programa ainda, então podem ocorrer falhas
  - Não pode ser evitado facilmente. A solução está fora do âmbito da disciplina

- 2nd C: **C**apacity Misses
  - Falha que ocorre porque a cache tem um tamanho limitado
  - Falha que não ocorreria se aumentássemos o tamanho da *cache*

# Tipos de *Cache Misses* (2/2)

- 3rd **C**: Conflict Misses
  - Falha que ocorre quando dois endereços de memória distintos são mapeados no mesmo bloco da *cache*
  - Pode ocorrer sobretudo quando dois blocos que por acaso são mapeados para o mesmo local na cache são utilizados de forma alternada
  - É um desperdício caso existam outros blocos livres na cache, mas que correspondem a outros blocos de memória que não estão a ser acedidos no momento
  - Este é o grande problema das caches de mapeamento directo!
  - Como atenuamos as consequências dete problema?

- Como lidar com este tipo de falha?
  - Solução 1: Aumentar o tamanho da cache
    - Vai falhar em algum momento (é impossível aumentar de forma ilimitada a cache)
  - Solução 2: Vários blocos distintos no mesmo índice de cache?

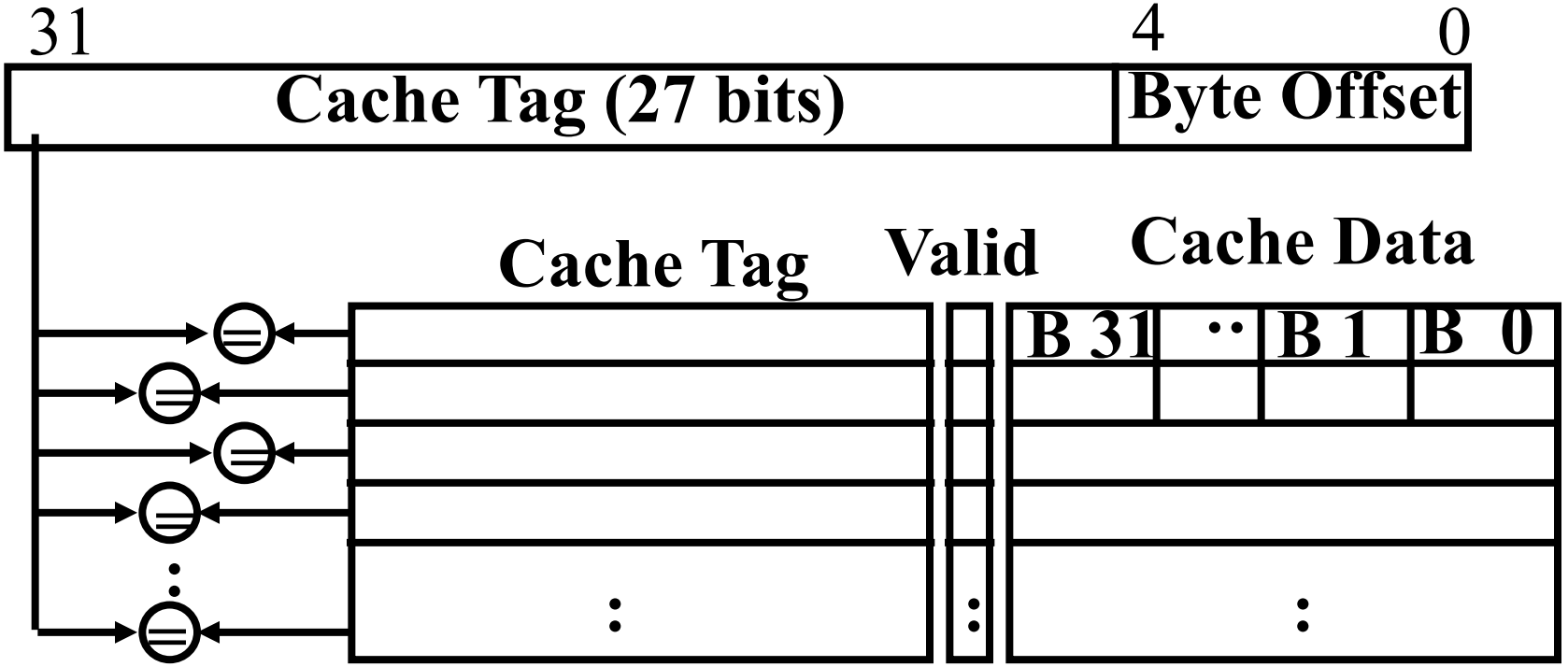
# *Cache Fully Associative* (1/3)

- Campos no Endereço de Memória:
  - ***Tag***: tal como anteriormente
  - ***Offset***: tal como anteriormente
  - ***Index***: não existe!

- O que isto significa?
  - Não há "linhas": qualquer bloco na memória principal pode ser mapeado em qualquer bloco na *cache*
  - Tem de se comparar com todas as *tags* na *cache* para descobrir onde o bloco da memória foi mapeado

# *Cache Fully Associative* (2/3)

- *Cache Fully Associative* (exemplo, 32 Bytes/block)
  - Compara as *tags* em paralelo



# *Cache Fully Associative* (3/3)

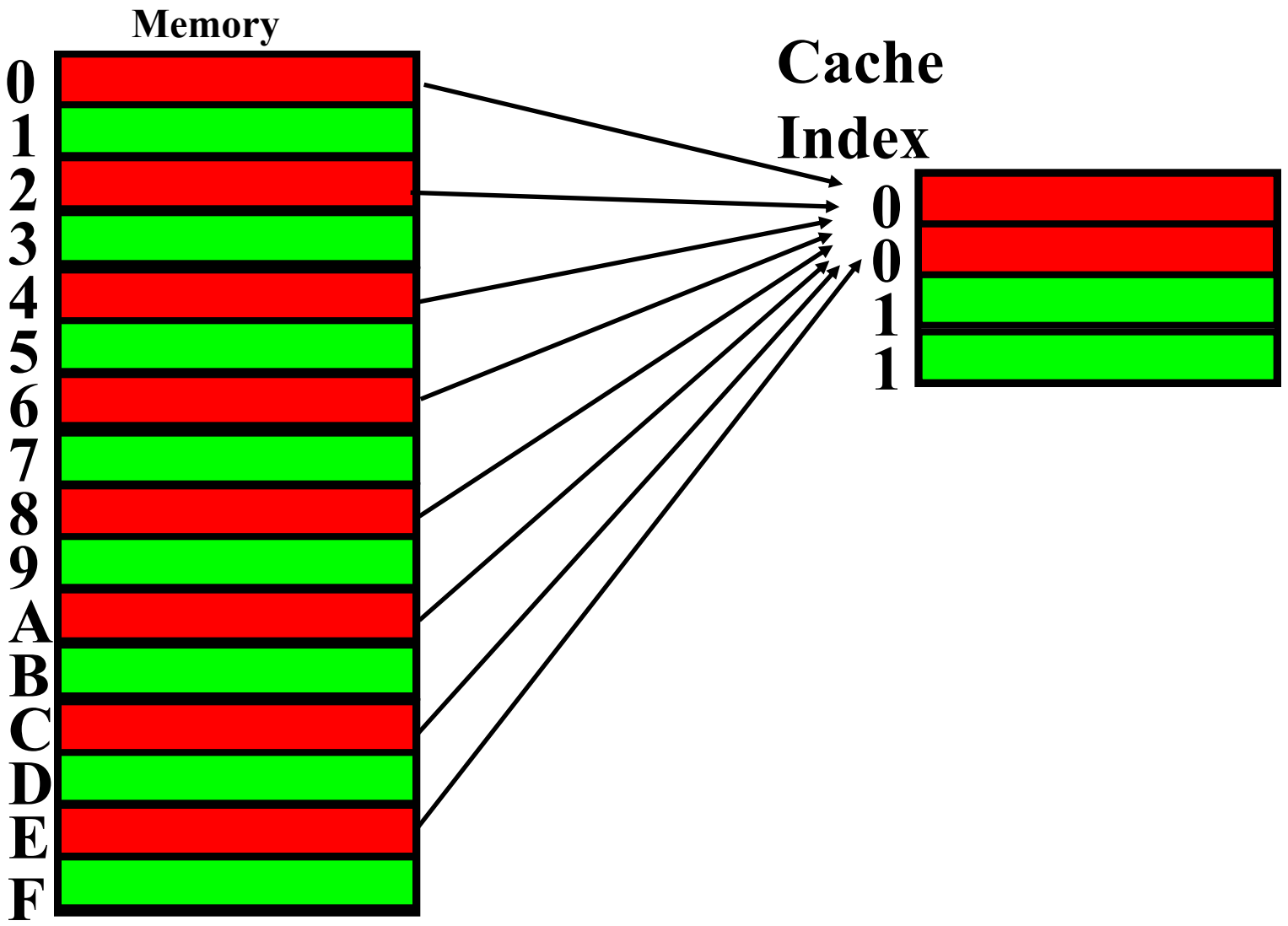
- Vantagens:
  - Não ocorrem *Conflit Misses* (já que os dados podem ir parar a qualquer lugar da *cache*)
  - O principal tipo de falha é o Capacity Miss

- Desvantagens:
  - Precisamos de ter um comparador em hardware para cada entrada na cache: se tivermos 64KB de dados numa cache com 4B de em cada bloco, precisaríamos de ter 16K comparadores: **impraticável**

# *Cache N-Way Set Associative* (1/4)

- Um compromisso entre as caches *Direct-Mapped* e as *Fully Associative*:
  - Agrupar blocos na cache em conjuntos de N blocos, que se comportam como uma cache *Fully Associative*.
  - Como **N** é normalmente pequeno então o hardware necessário não é complexo.
  - Resolve os *Conflict Misses* uma vez que no mesmo conjunto vamos ter espaço para vários blocos.

# Exemplo de uma 2-Way Set Associative Cache



# *Cache N-Way Set Associative* (2/4)

- Campos no Endereço de Memória:
  - Tag: tal como anteriormente, um identificador único de cada bloco em memória.
  - Offset: tal como anteriormente
  - Index: indica qual **o conjunto** de blocos na *cache* onde o bloco de memória pode ser mapeado

- Então qual é a diferença?
  - cada conjunto na cache pode conter vários blocos
  - Assim que encontramos o conjunto correto na cache, temos de comparar com todas as *tags* desse conjunto para encontrar o local onde o bloco já está mapeado.

# *Cache N-Way Set Associative* (3/4)

- Ideia Basica:
  - A *cache* é *direct-mapped* se pensarmos em termos de *sets;*
  - Cada conjunto é *fully associative;*
  - Basicamente as *caches N-way Set Associative* trabalham em paralelo dentro de cada conjunto: cada bloco tem o seu próprio *valid bit* e dados.

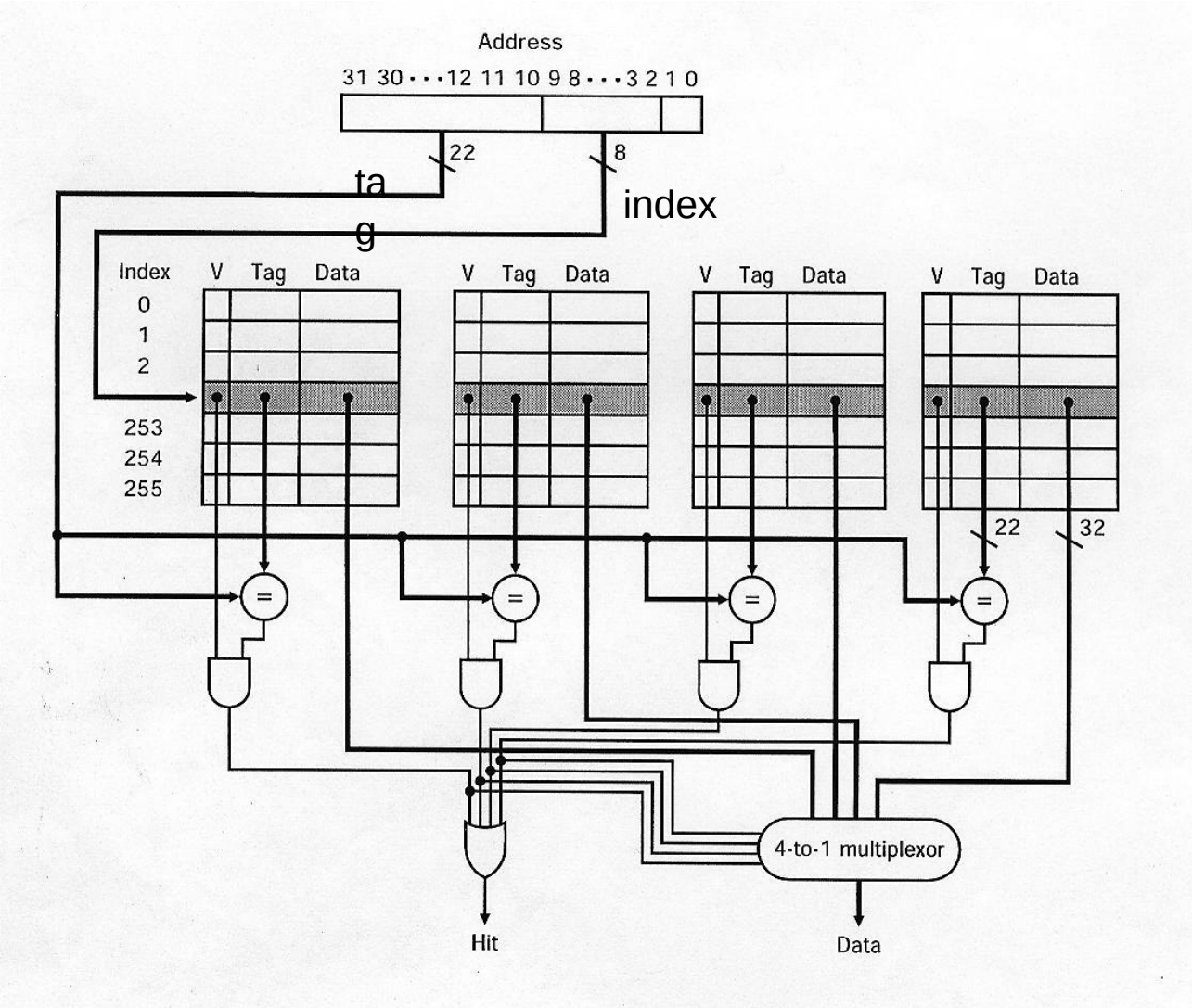
- Dado um endereço de memória:
  - Encontrar o conjunto correto na cache usando o valor do campo ***Index***.
  - Comparar o ***Tag*** do bloco em memória com todos os ***Tags*** no conjunto determinado anteriormente.
  - Se ocorrer um ***match***, então temos um ***hit***!, caso contrário, um ocorre um ***miss***.
  - Finalmente, usar o campo de ***offset*** como de costume para encontrar os dados desejados dentro do bloco.

# *Cache N-Way Set Associative* (4/4)

- Quais as grandes vantagens?
  - Mesmo uma cache 2-way Set Associative Evita muitos ***conflict misses***;
  - O custo do hardware não é significativo: apenas são necessários N comparadores.

- De facto uma *cache* com M blocos:
  - É ***Direct-Mapped*** se for uma *cache 1-way Set Associative*
  - É ***Fully Associative*** se for uma *cache M-way Set Associative*
  - Então estes dois exemplos são apenas casos especiais do modelo concepetual de uma cache N-Way Set Associative

# Circuito de uma *4-Way Set Associative Cache*



# Política de Substituição de Blocos

- Cache Direct-Mapped: o índice especifica completamente qual a posição em que um bloco pode entrar em caso de um ***miss***
- Cache N-Way Set Associative: o índice especifica um conjunto de blocos na cache, mas o bloco pode ocupar qualquer posição dentro do conjunto no caso de um ***miss***
- Fully Associative: Cada bloco pode ser escrito em qualquer posição na cache

- Pergunta: se tivermos escolha, onde devemos escrever o novo bloco?
  - Se houver algum local com o *valid bit a* ***off*** (assinalando que o bloco está vazio), geralmente escreve-se o novo bloco no primeiro bloco que aparecer nessa situação.
  - Se todas as localizações possíveis já tiverem um bloco válido, devemos escolher uma política de substituição (replacement policy), ou seja temos de definir uma regra pela qual determinamos qual bloco é substituido em caso de um ***miss***.

# Política de Substituição de Blocos: LRU

- LRU (Least Recently Used)
  - **Ideia:** substituir o bloco que foi acedido (leitura ou escrita) menos recentemente;
  - **Prós:** localidade temporal ⇒ bloco usado num passado recente implica um uso futuro provável: na verdade, esta é uma política muito eficaz
  - **Contras:** com uma *cache 2-way set associative* é fácil de identificar (um *bit* para indicar o LRU); com uma *cache 4-way* ou superior, requer *hardware* mais sofisticado e muito tempo para fazer a identificação.

# Big Idea

- Como escolher entre os diferentes modelos de *cache*, tamanho de bloco e política de substituição?

- Conceber um modelo de avaliação de desempenho:
  - Minimizar:

  *Average Memory Access Time (AMAT)* = Hit Time + Miss Penalty x Miss Rate

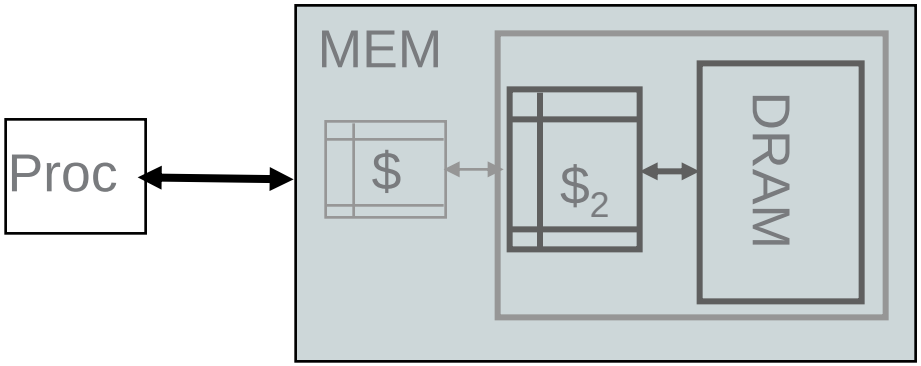
- Criar a ilusão de uma memória grande, barata e rápida (em média)

- Como podemos melhorar a *miss penalty*?

# Melhorando a *Miss Penalty*

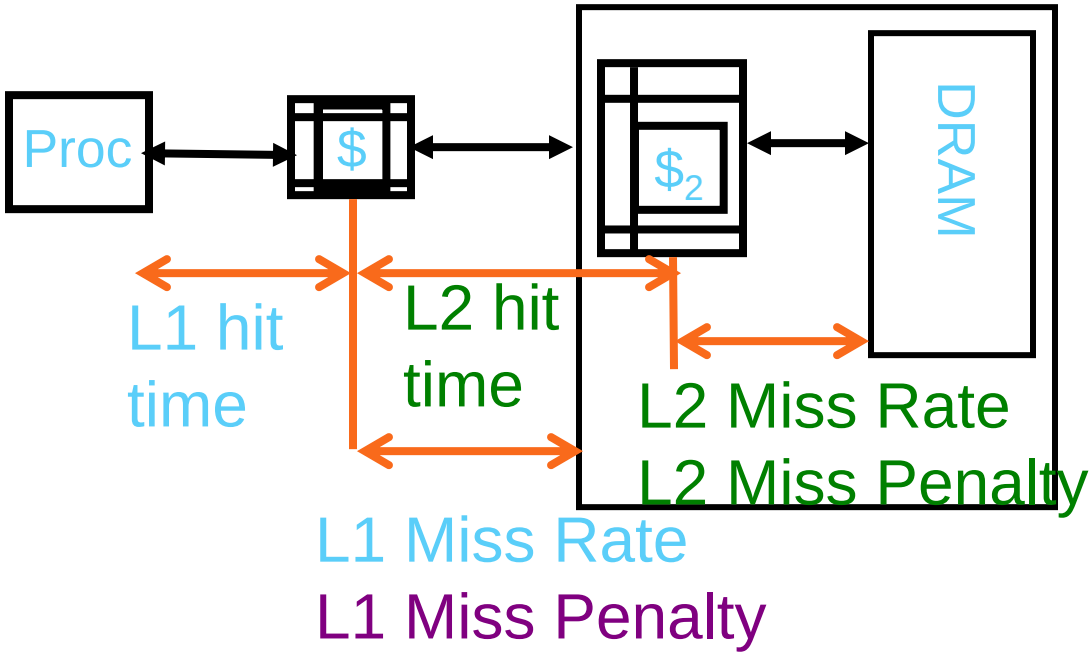
Quando as *caches* se tornaram populares, a ***Miss Penalty*** era ~ 10 ciclos de relógio do processador.

Hoje com os processadores modernos na casa dos 5 GHz (0.2 ns por ciclo de relógio) e 80 ns de tempo de acesso às memória DRAM ➔ 400 ciclos de relógio do processador!




**Solução: adicionar outros níveis de memória *cache* entre a memória e a *cache* do processador: *Second Level (L2) Cache***

# Analisando uma Hierarquia de Memória *Cache* Multinível




**Avg Mem Access Time =**

**L1 Hit Time + L1 Miss Rate * L1 Miss Penalty**

---

**L1 Miss Penalty =**

**L2 Hit Time + L2 Miss Rate * L2 Miss Penalty**

---

**Avg Mem Access Time =**

**L1 Hit Time + L1 Miss Rate ***

**(L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)**

# Exemplo: com uma *cache* L2

- Assumindo:
  - L1 Hit Time = 1 ciclo
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 ciclos
  - L2 Miss rate = 15% (% L1 misses que são misses na cache L2)
  - L2 Miss Penalty = 200 ciclos
- L1 miss penalty = 5 + 0.15 * 200 = 35
- Avg mem access time = 1 + 0.05 x 35
  = 2.75 ciclos

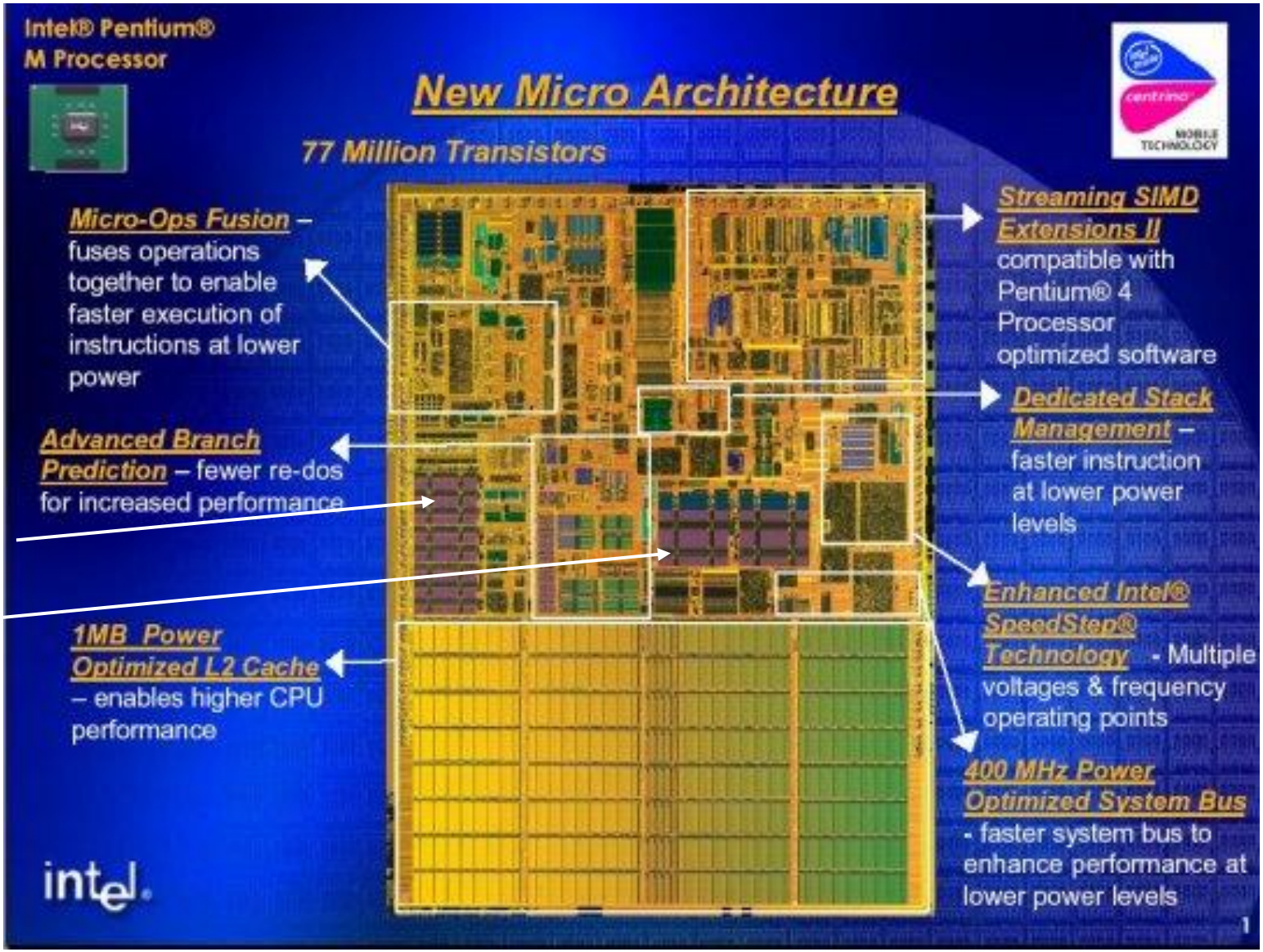
# Exemplo sem a cache L2

- Assumindo:
  - L1 Hit Time = 1 ciclo
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 200 ciclos
- Avg mem access time = 1 + 0.05 x 200 = 11 cycles
- 4x mais rápido com a cache L2 cache! (2.75 vs. 11)

# Conclusão

- Discutimos o conceito de *cache* de memória em detalhe. O conceito de *cache* em geral aparece repetidamente no ecossistema de um computador:
  - Cache no Sistema de Ficheiros
  - Cache em páginas Web
  - Cache em Bases de Dados
  - Memorização de Software
  - Outros?
- Big idea: se algo é dispendioso computacionalmente, mas queremos fazê-lo repetidamente, fazemo-lo uma vez e armazenamos o resultado em ***cache***.
- Escolhas no desenho de uma *Cache*:
  - Write through v. write back
  - size of cache: speed v. capacity
  - direct-mapped v. associative
  - for N-way set assoc: choice of N
  - block replacement policy
  - 2nd level cache?
  - 3rd level cache?
- Usamos o modelo de desempenho (o AMAT) para escolher entre as opções, dependendo dos programas, tecnologia, orçamento, etc..

# O Intel Pentium M CPU (antigo)

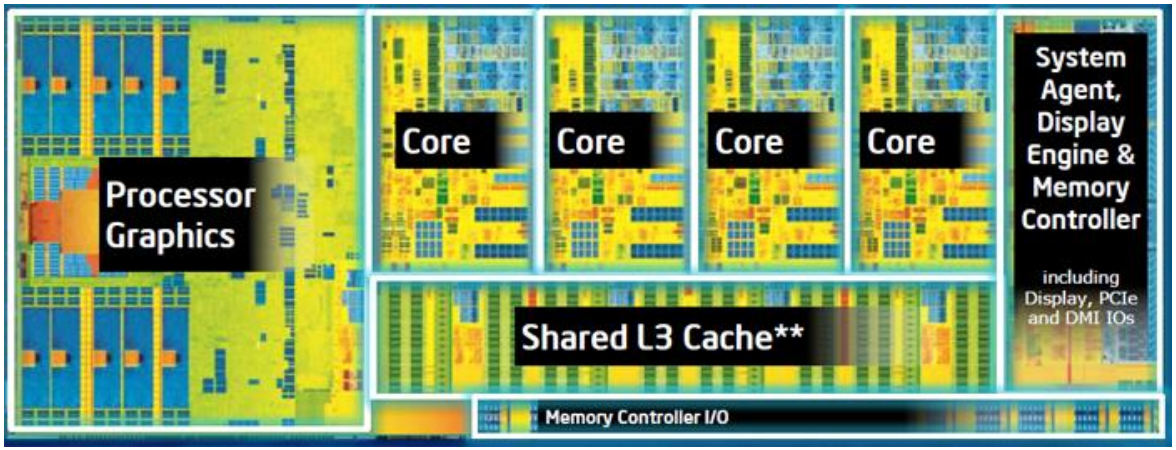



32KB I$

32KB D$

# A Arquitetura de uma CPU moderna

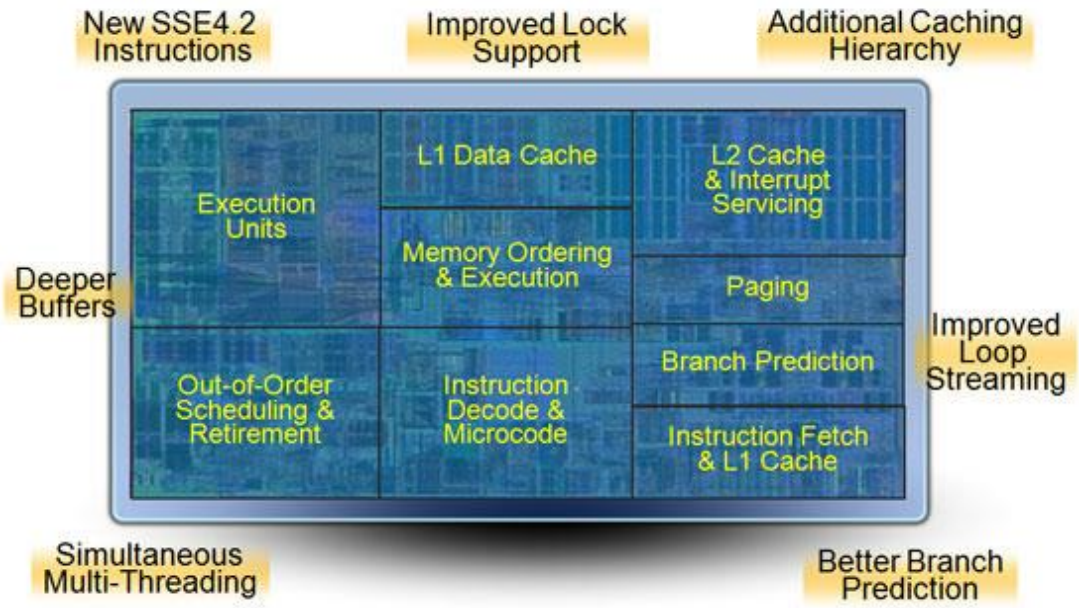
- 4 core number with Hyper-Threading, each core running 2 threads (e.g. core i7-4770K)
- Low Latency Processor
- Execute instructions sequentially
- Low memory Bandwidth (~25.6GB/s)

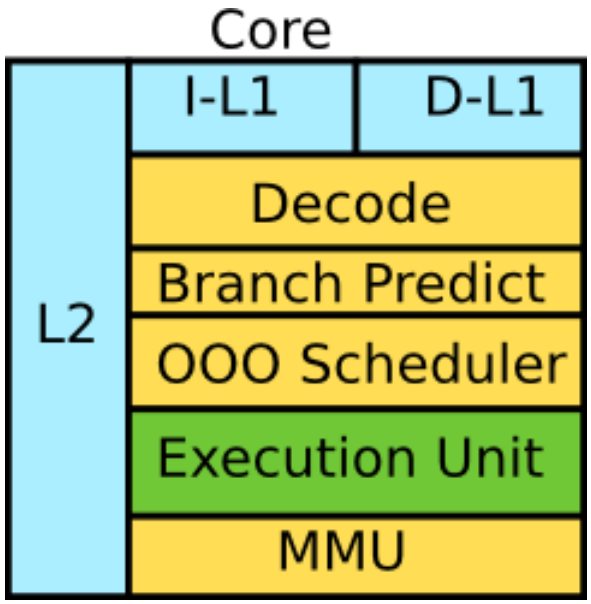




(Haswell Architecture @ Intel Core i7)

# A evolução dos CPU
## Multi-core CPUs




Menos de **10%** da área total do chip é usada para executar código.

# Para saber mais ...

- P&H - Capítulo 7.1, 7.2 e 7.3