

# 04 - Hierarquia de Memória: Organização da *cache*

Arquitectura de Computadores  
Licenciatura em Engenharia Informática  
Luís Paulo Santos

# Material de apoio

(mesmo que para 03 – Hierarquia da Memória: Conceitos Básicos)

- “*Computer Organization and Design: The Hardware / Software Interface*”  
David A. Patterson, John L. Hennessy; 5th Edition, 2013
  - Secção 5.1 (pags. 372 .. 378) – Introduction
  - Secção 5.3 + 5.4 (pags. 383 .. 418) – The Basics of Caches + Performance
  - Secção 5.8 (pags. 454 .. 461) – A Common Framework for Memory Hierarchy
  - Secção 5.13 (pags. 471 .. 475) – Real Stuff: ARM Cortex-A8 and Intel Core i7
- “*Computer Systems: a Programmer's Perspective*”; Randal E. Bryant, David R. O'Hallaron--Pearson (2nd ed., 2011)
  - Secção 1.5 + 1.6 (pags. 12 .. 14) – Introduction
  - Capítulo 6 (pag. 560) – Preâmbulo
  - Secção 6.2 .. 6.7 (pags. 586 .. 630)

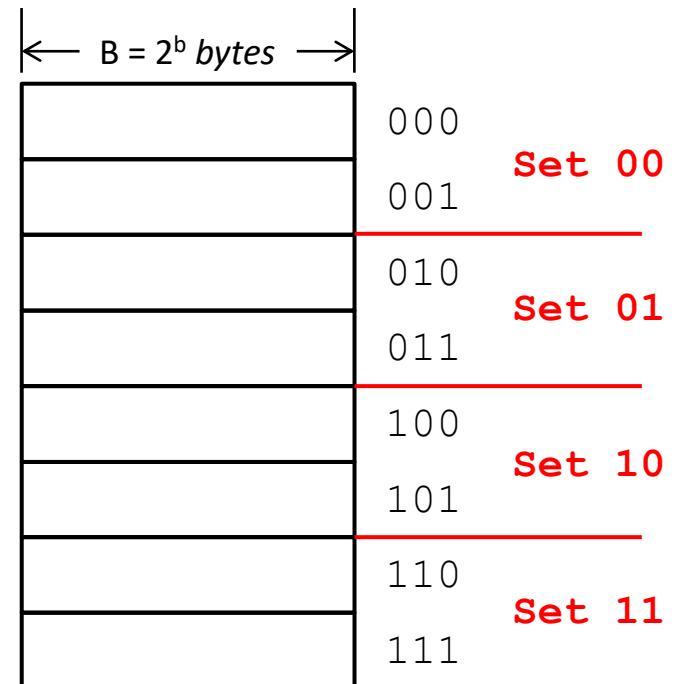
# Organização da Cache

- A memória (central) tem  $M$  bytes.  
O endereço tem  $m = \log_2(M)$  bits
- A cache organiza-se em linhas de  $B$  bytes.  
 $b = \log_2(B)$  bits identificam um byte.
- As linhas encontram-se agrupadas em  $S$  sets.  
 $s = \log_2(S)$  bits identificam um set.
- Cada set contém  $E$  linhas.
- A organização da cache é caracterizada por  
( $S, E, B, m$ )

Número total de linhas =  $S * E$

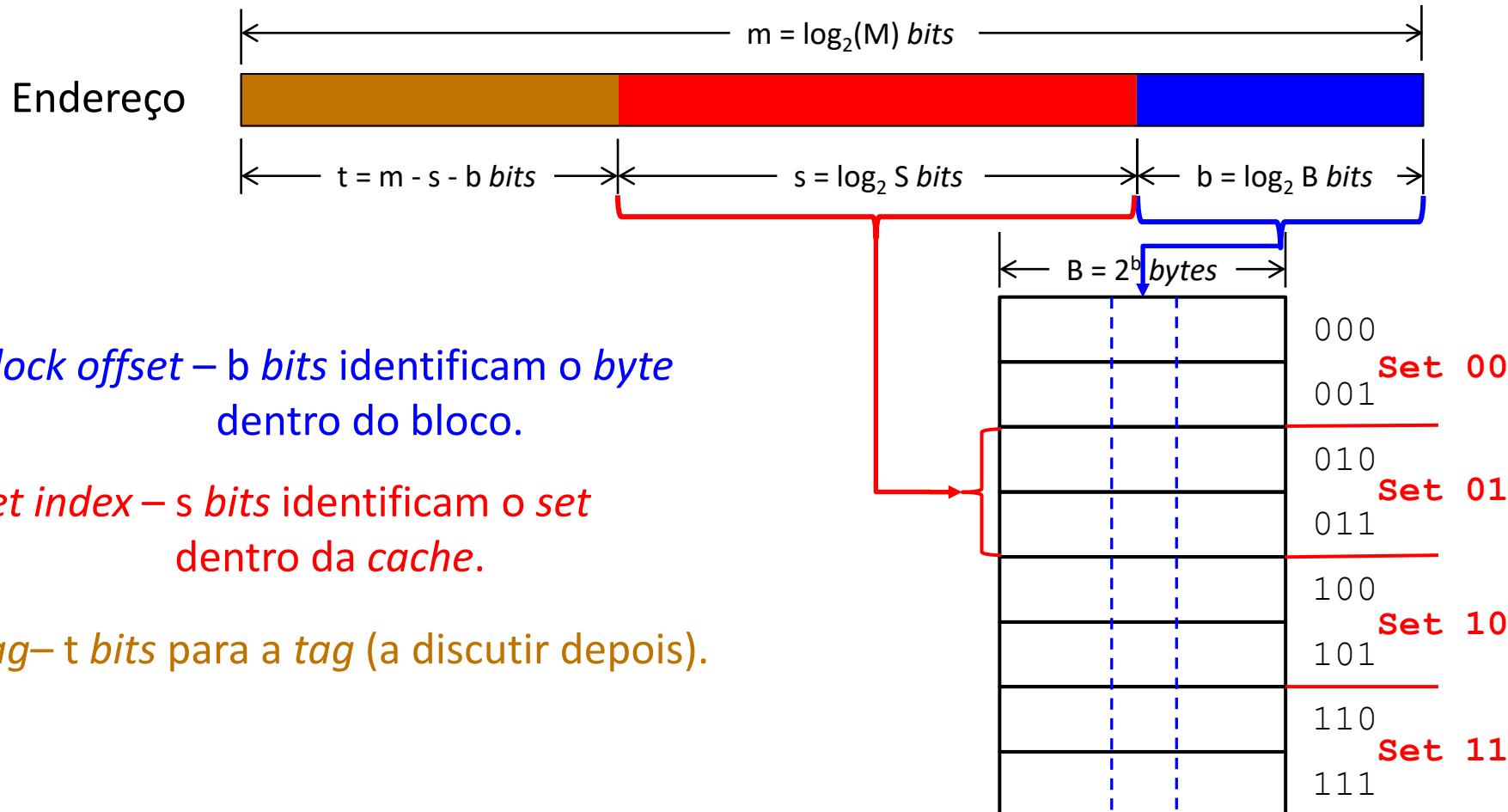
Capacidade da cache =  $S * E * B$

Cache (4, 2, B (?), m (?))



# Organização da Cache

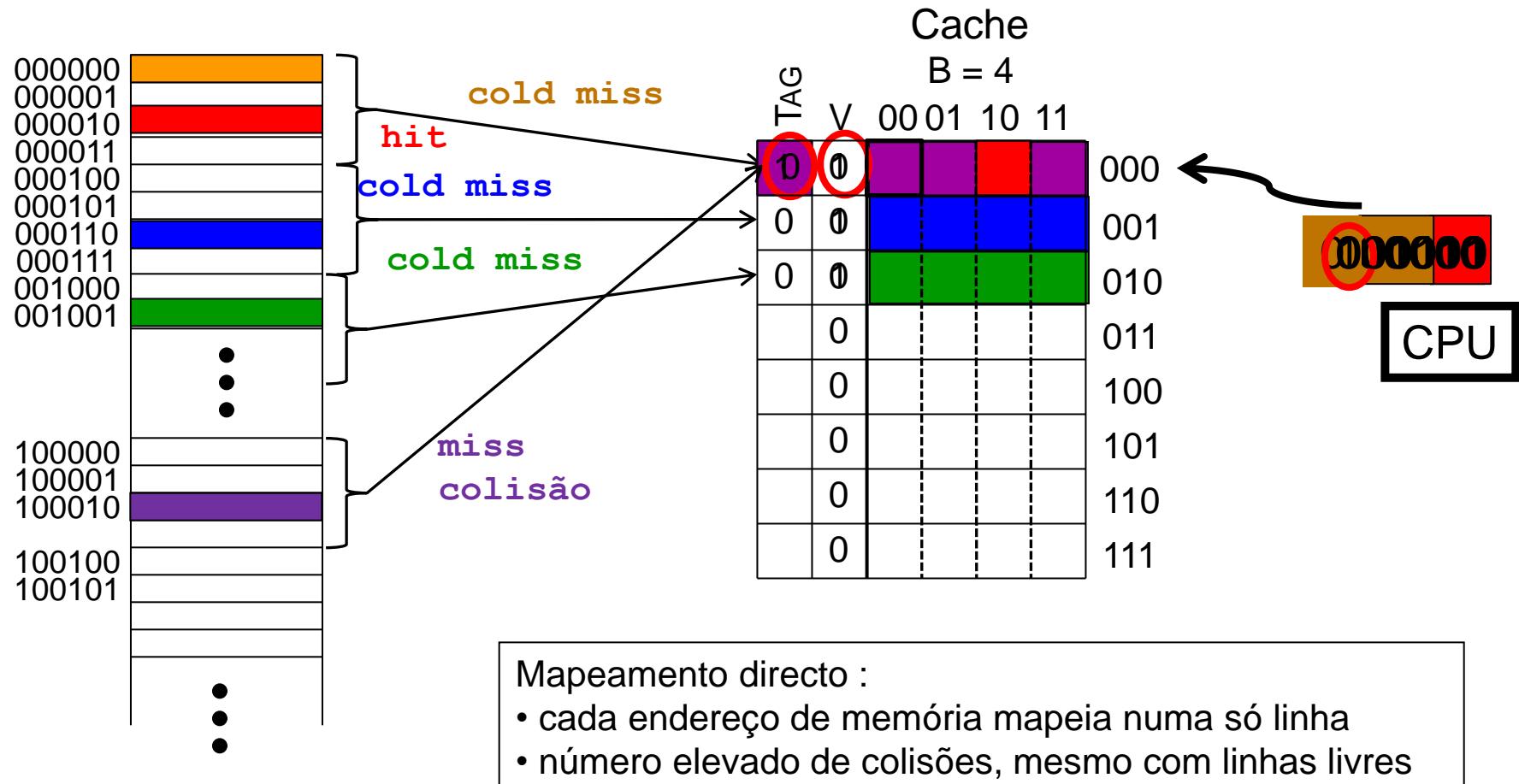
- Cache ( $S, E, B, m$ )



# Mapeamento Directo

Organização da *cache* com E=1. Cada *set* só tem 1 linha.

Exemplo: (S = 8, E = 1, B = 4, m=6)



# Mapeamento Directo

Na máquina anterior com endereços de  $m=6$  bits, 8 sets ( $S=8$ ), mapeamento directo ( $E=1$ ) e linhas de 4 bytes ( $B=4$ ), todos os endereços da forma ?000?? mapeiam no set com o índice 000. Como é que o CPU determina qual o endereço que está na cache?

Os restantes *bits* mais significativos do endereço (apenas 1 bit neste exemplo) são colocados na *tag*. Número de *bits* da *tag*  $t=m-s-b$

Como é que o CPU determina se uma linha da cache contém dados válidos?

Cada linha da cache tem um bit extra (*valid*) que indica se os dados dessa linha são válidos.

Cache

	Valid	Tag	00	01	10	11	
1	0						000
0							001
0							010
0							011
0							100
0							101
0							110
0							111

# Mapeamento Directo

(S = 8, E = 1, B = 4, m=7)

Indique, dentro do tempo limite, qual o endereço mapeado na posição indicada da cache:

- $11\ 100\ 00_2 = 112_{10}$
- $00\ 001\ 01_2 = 5_{10}$
- $11\ 011\ 10_2 = 110_{10}$
- $00\ 111\ 00_2 = 28_{10}$
- Inválido (V==0)



49 segs 29 segs 15 segs 5 segs

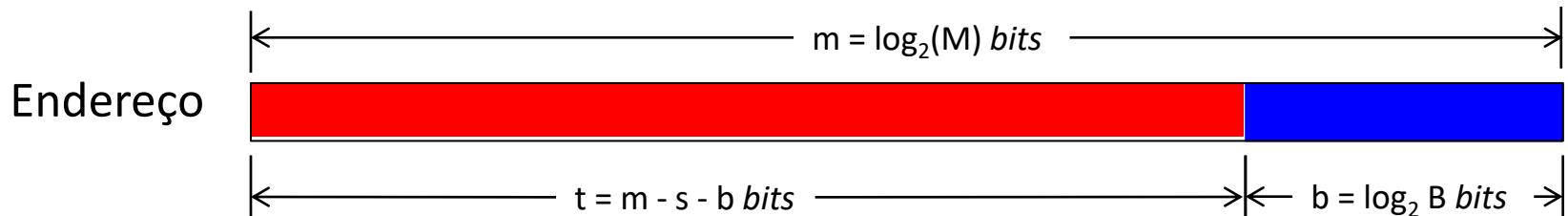
Cache B = 4		V	00	01	10	11	
TAG							
01	1						000
00	1						001
10	1						010
11	1						011
11	1						100
10	1						101
11	0						110
00	1						111

# Mapeamento Directo

- Número de sets ( $S$ ) igual ao número de linhas
- Uma linha por set ( $E=1$ )
- Um endereço de memória mapeia em uma e uma só linha da *cache*
  - **Consequência:** potencial para elevado número de colisões
- O *set index*, e a linha correspondente da *cache*, é determinado de forma unívoca por  $s$  *bits* do endereço
  - **Consequência:** *hit time* reduzido pois só é necessária a comparação de uma *tag*

# Mapeamento completamente associativo

- 1 único *set* ( $S=1$ ) que contém todas as linhas
- qualquer endereço pode mapear em qualquer linha
- *set index* não é usado
- Cache (1, E, B, m)



*block offset – b bits* identificam o *byte* dentro do bloco.

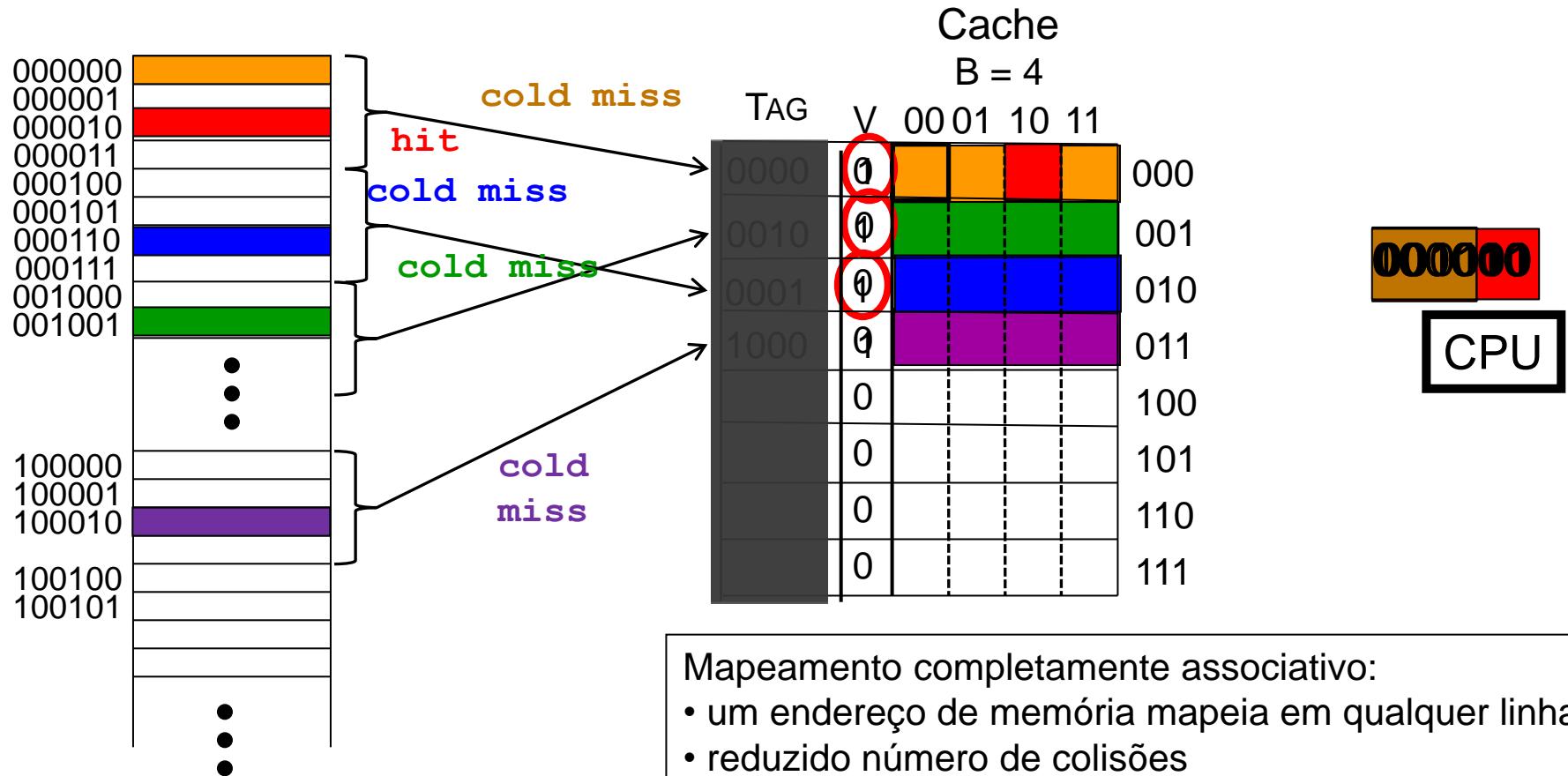
*set index – s=0 bits* não é utilizado pois só existe 1 *set*.

*tag – t bits para a tag.*

# Mapeamento Completamente Associativo

Organização da *cache* com  $S=1$ . Só existe 1 *set* que contém todas as linhas.

Exemplo: ( $S = 1$ ,  $E = 8$ ,  $B = 4$ ,  $m=6$ )



# Mapeamento Completamente Associativo

- Um único set ( $S=1$ )
- Todas as linhas pertencem ao mesmo set
- Um endereço de memória mapeia em qualquer linha da *cache*
  - **Consequência:** reduzido número de colisões
  - Como não existe *set index* a *tag* tem  $m-b$  *bits* e a procura de um endereço implica a comparação com todas as *tags*
    - **Consequência:** *hit time* aumenta pois é necessário comparar com todas as *tags*, que têm um número elevado de *bits*

# Mapeamento *n-way set associative*

O mapeamento directo resulta num grande número de colisões.

O mapeamento *fully associative* implica um grande *overhead* em termos de tag e pesquisas mais longas nas linhas da cache.

O mapeamento *n-way set associative* representa um compromisso entre os 2.

A cache é dividida em S sets de E linhas.

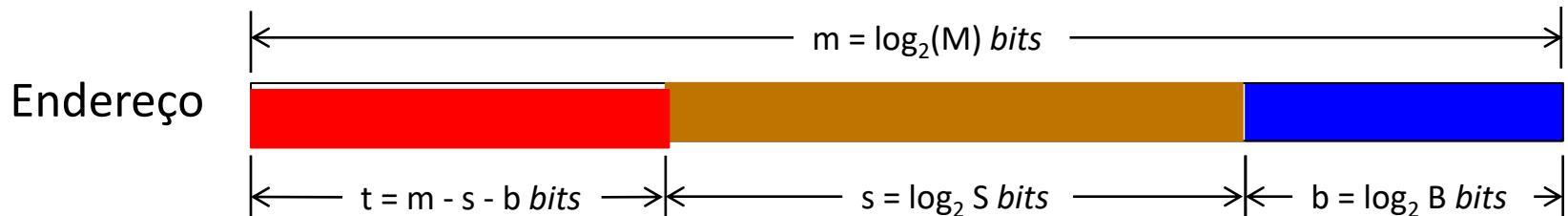
Dentro de cada set o bloco pode ser colocado em qualquer linha.

Relativamente ao mapeamento directo - reduz-se o número de colisões

Relativamente ao mapeamento *fully associative* – reduz-se o número de *bits* de tag e o tempo de procura na cache (*hit time*)

# Mapeamento $n$ -way set associative

- $S$  sets com  $E$  linhas cada
- *set index* determina o set em que um endereço mapeia
- um endereço mapeia em qualquer linha do respectivo set
- Cache ( $S, E, B, m$ )



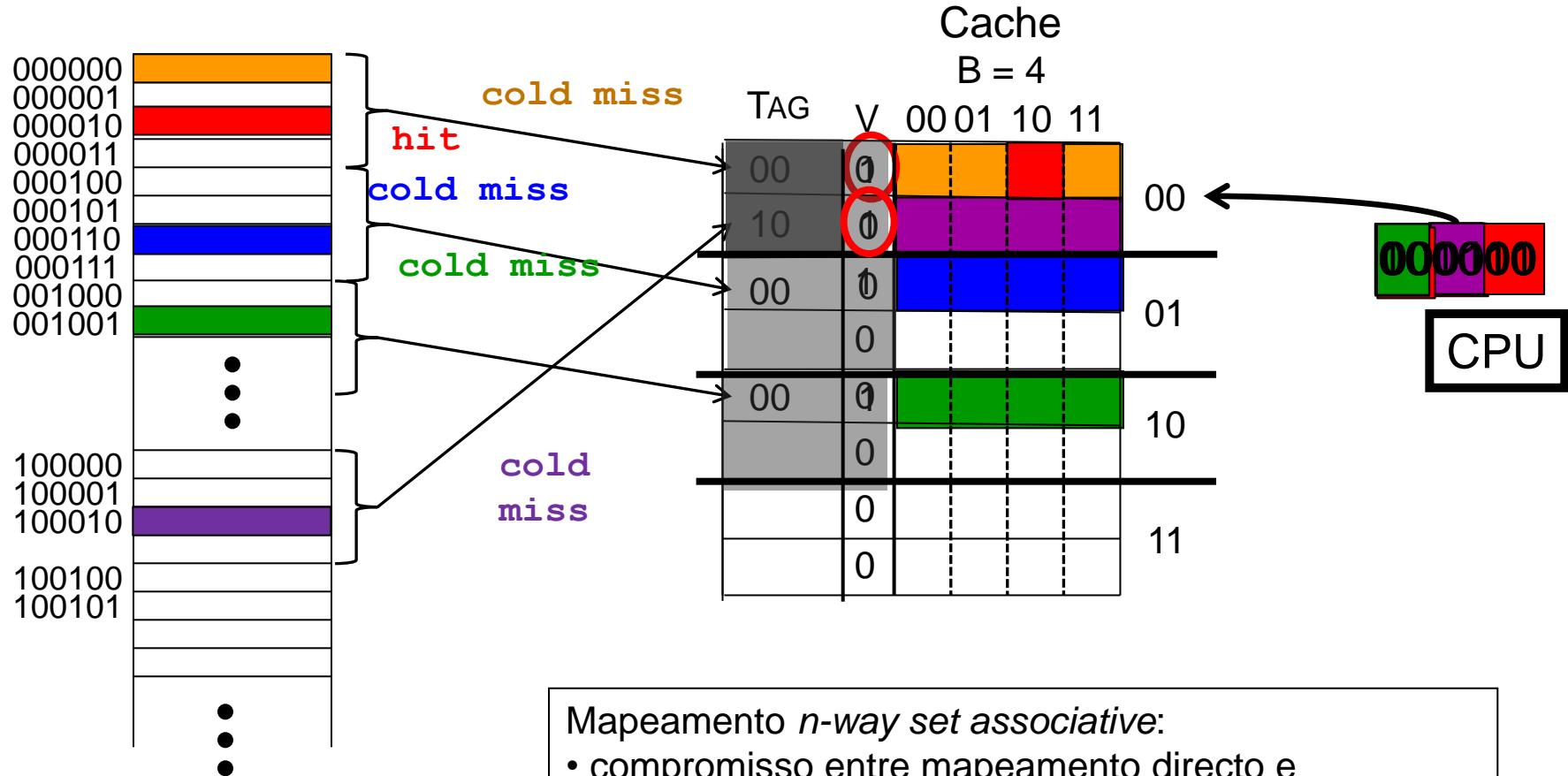
*block offset – b bits* identificam o *byte* dentro do bloco.

*set index – s bits* identificam o *set*.

*tag – t = m – s – b bits* para a *tag*.

# Mapeamento $n$ -way set associative

Exemplo: ( $S = 4$ ,  $E = 2$ ,  $B = 4$ ,  $m=6$ )



Mapeamento  $n$ -way set associative:  
• compromisso entre mapeamento directo e mapeamento completamente associativo

# Mapeamento $n$ -way set associative

(S = 4, E = 2, B = 4, m=7)

Indique, dentro do tempo limite, qual o endereço mapeado na posição indicada da cache:

- $111\ 10\ 00_2 = 120_{10}$
- $010\ 00\ 01_2 = 33_{10}$
- $001\ 01\ 10_2 = 22_{10}$



49 segs 29 segs 15 segs

Cache B = 4		V	00	01	10	11	
TAG							
011	1						000
010	1						001
110	1						010
001	1						011
111	1						100
010	1						101
111	0						110
100	1						111

The diagram shows a cache organization with 4 sets (S=4) and 4 ways per set (m=4). The cache has 8 slots. The address fields are: TAG (11 bits), V (1 bit), and index (2 bits). The index field is split into two 1-bit fields: 00, 01, 10, and 11. The cache is divided into four sets by the index fields. The V bit indicates if a slot is valid. The TAG field is compared with the memory address to find a matching entry. The cache state is as follows:

- Set 0 (index 00): Slot 011 (V=1, TAG=111) is highlighted with a blue circle.
- Set 1 (index 01): Slot 010 (V=1, TAG=010) is highlighted with a purple circle.
- Set 2 (index 10): Slot 110 (V=1, TAG=110) is highlighted with a red circle.
- Set 3 (index 11): Slot 111 (V=0, TAG=111) is highlighted with a green circle.

Red lines on the right side of the cache columns indicate the boundaries of the sets.

# Mapeamento $n$ -way set associative

- $S$  sets com  $E$  linhas cada
- Um endereço de memória mapeia num único set
- Um endereço de memória mapeia em qualquer linha do seu set
  - **Consequência:** compromisso no número de colisões
  - Para localizar um endereço dentro do respectivo set é necessário comparar com as *tags* de todas as linhas desse set
    - **Consequência:** compromisso no *hit time*

# Escrita na *cache*

O que acontece quando o CPU altera um *byte* ou palavra na cache?

Existem 2 políticas alternativas:

1. *Write-through* – a informação é escrita na *cache* e na memória central;
2. *Write-back* – a informação é escrita apenas na *cache*. A memória central só é actualizada quando o bloco for substituído.

# Write-through

- Uma vez que a escrita é feita nos vários níveis da hierarquia, leva tanto tempo como o nível mais lento
- **SOLUÇÃO:** usar um *write-buffer*. A escrita é feita no nível superior da *cache* e enviada para este *buffer*.  
O processador continua a executar e a escrita nos níveis mais lentos da hierarquia é feita de forma assíncrona.  
Problema: o *write-buffer* pode encher
- Várias escritas sucessivas no mesmo bloco implicam **também** escritas sucessivas nos níveis mais caros da hierarquia

# Write-back

- Uma vez que a escrita é feita no nível superior da hierarquia é mais rápida
- Escritas consecutivas no mesmo endereço (mesmo bloco) não implicam múltiplas escritas nos níveis mais lentos.  
Esta escrita acontece apenas quando o bloco é substituído.
- Uma vez que quando ocorre um *miss* o respectivo bloco tem que ser actualizado na memória (se foi escrito anteriormente), aumenta a *miss penalty*
- Bastante mais complexo de implementar do que *write-through*

# *Write-misses*

- Como tratar os *write-misses*?
  - ***write-allocate*** - o bloco é copiado de memória para a *cache*, a escrita é feita na *cache* e, se se tratar de *write-through*, também na memória ;
  - ***no-write-allocate*** - o bloco é escrito apenas na memória (escritas consecutivas no mesmo ADDR pagam sempre a *miss penalty*)

# Escrita na cache

## *Write-through*

### **Vantagens**

1. Não aumenta a *miss penalty*, pois o bloco não tem que ser escrito num *miss*
2. Mais simples de implementar

### **Desvantagens**

1. Várias escritas no mesmo bloco implicam várias escritas na memória central
2. As escritas são feitas à velocidade da memória central e não à da cache  
(embora os processadores actuais usem *write-buffers* para diminuir a ocorrência deste problema)

## *Write-back*

### **Vantagens**

1. Minimização do número de escritas na memória central;
2. Cada palavra é escrita à velocidade da cache e não à velocidade da memória central

### **Desvantagens**

1. Aumenta a *miss penalty*
2. Mais complexo de implementar

# Políticas de Substituição

Numa cache com **algum grau de associatividade** qual o bloco que deve ser substituído quando ocorre uma colisão?

**Least Recently Used (LRU)** – é substituído o bloco que não é usado há mais tempo.

**Aleatoriamente (random)** – o bloco a substituir é escolhido aleatoriamente.

LRU: a implementação é muito custosa para associatividade > 2 (eventualmente 4).

Alguns processadores usam uma aproximação a LRU.

A diferença de *miss rate* entre o esquema aleatório e LRU não é elevada (1.1 vezes superior para 2-way associative).

Se a *miss penalty* não é muito elevada o esquema aleatório pode ser tão, ou mais, eficaz como as aproximações ao LRU.

# Casos reais: ARM Cortex-A8 e Intel Core I7 920

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	-	Unified (instruction and data)
L3 cache size	-	8 MiB, shared
L3 cache associativity	-	16-way set associative
L3 replacement	-	Approximated LRU
L3 block size	-	64 bytes
L3 write policy	-	Write-back, Write-allocate
L3 hit time	-	35 clock cycles

[Computer Organization and Design: the Hardware / Software Interface  
Patterson and Hennessy, 5th Edition, 2013]