

Hierarquia de Memória: Organização da *cache*

Arquitectura de Computadores
Lic. em Engenharia Informática
Luís Paulo Santos

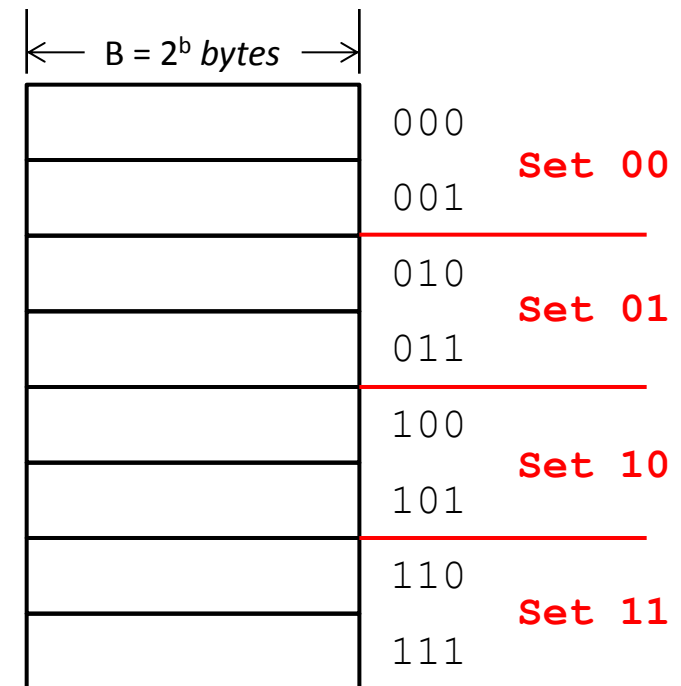
Organização da Cache

- 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 = $S * E$

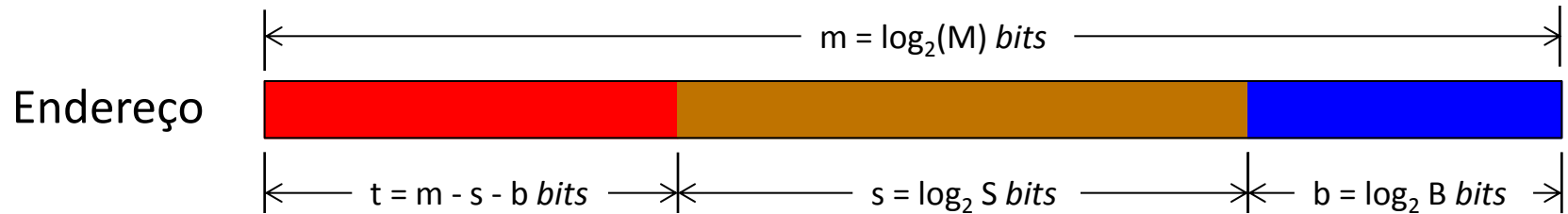
Capacidade da *cache* = $S * E * B$

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



Organização da Cache

- Cache (S, E, B, m)



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

set index – s bits identificam o *set* dentro da *cache*.

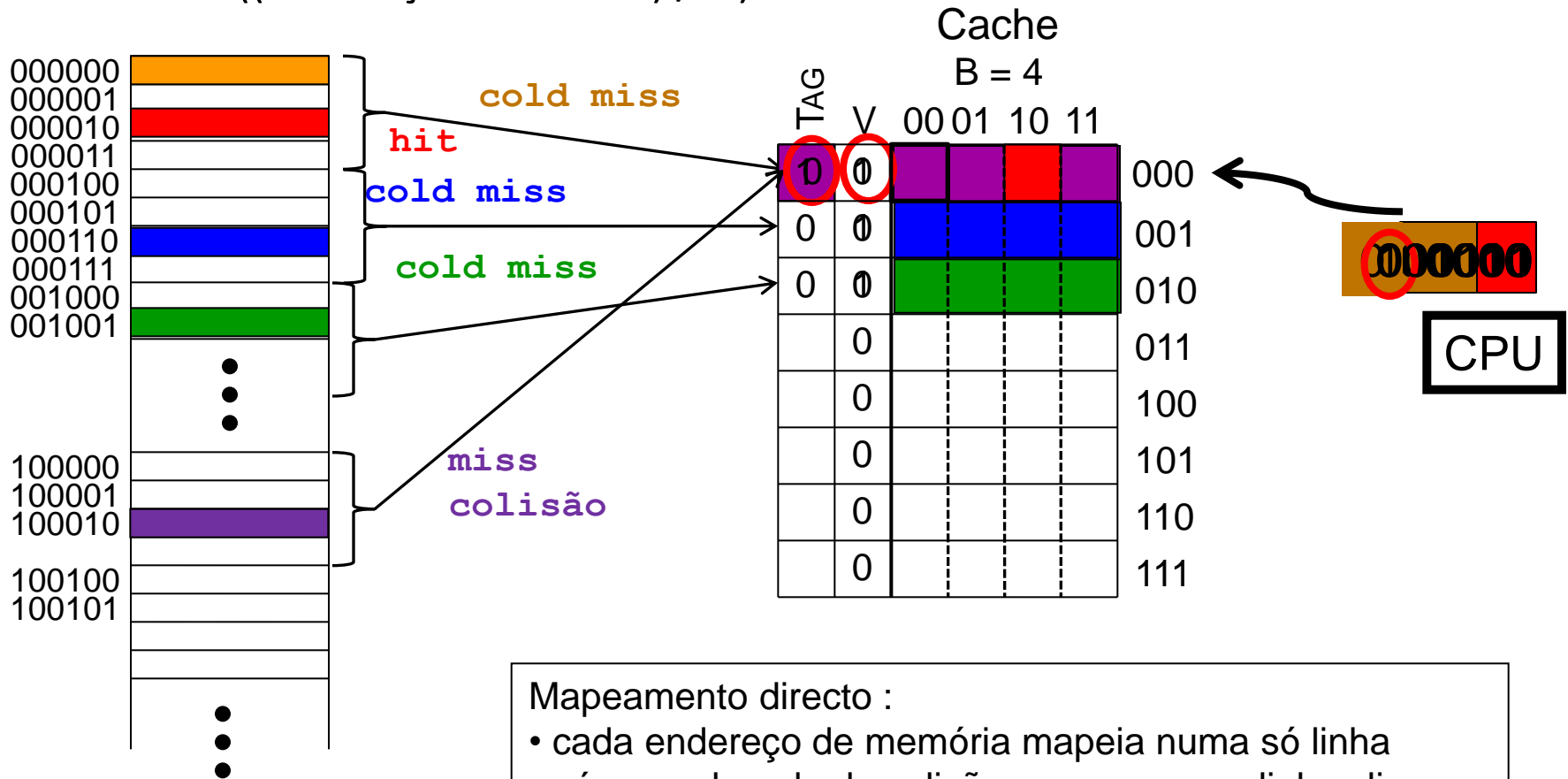
tag – t bits para a *tag* (a discutir depois).

Mapeamento Directo

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

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

$set = \text{resto}((\text{endereço sem } b \text{ bits}) / S)$



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 (1 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

Considere uma máquina com um espaço de endereçamento de 32 *bits*, uma cache de 64 Kbytes, mapeamento directo e blocos de 32 bytes. Quantos *bits* são necessários para a tag?

A cache tem $64\text{ K} / 32\text{ sets}$, ou seja, $2^6 \cdot 2^{10} / 2^5 = 2^{11}$, logo o *set index* são 11 *bits*. O *bloco offset* são $b = \log_2(32) = 5\text{ bits}$. A tag será de $t = m - s - b = 32 - 11 - 5 = 16\text{ bits}$.

Qual a capacidade total desta cache, contando com os bits da tag mais os valid bits?

Temos 64 Kbytes de dados.

2048 *sets* (== num. de linhas), cada com 2 bytes de tag = 4 Kbytes.

Cada linha tem um valid bit logo 2048 bits = 0,25 Kbytes

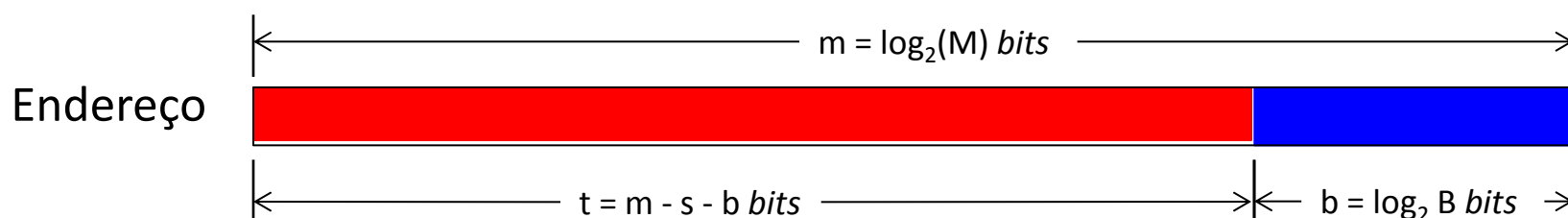
Capacidade total = $64 + 4 + 0,25 = 68,25\text{ KBytes}$

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 uma comparação de *tags*

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.

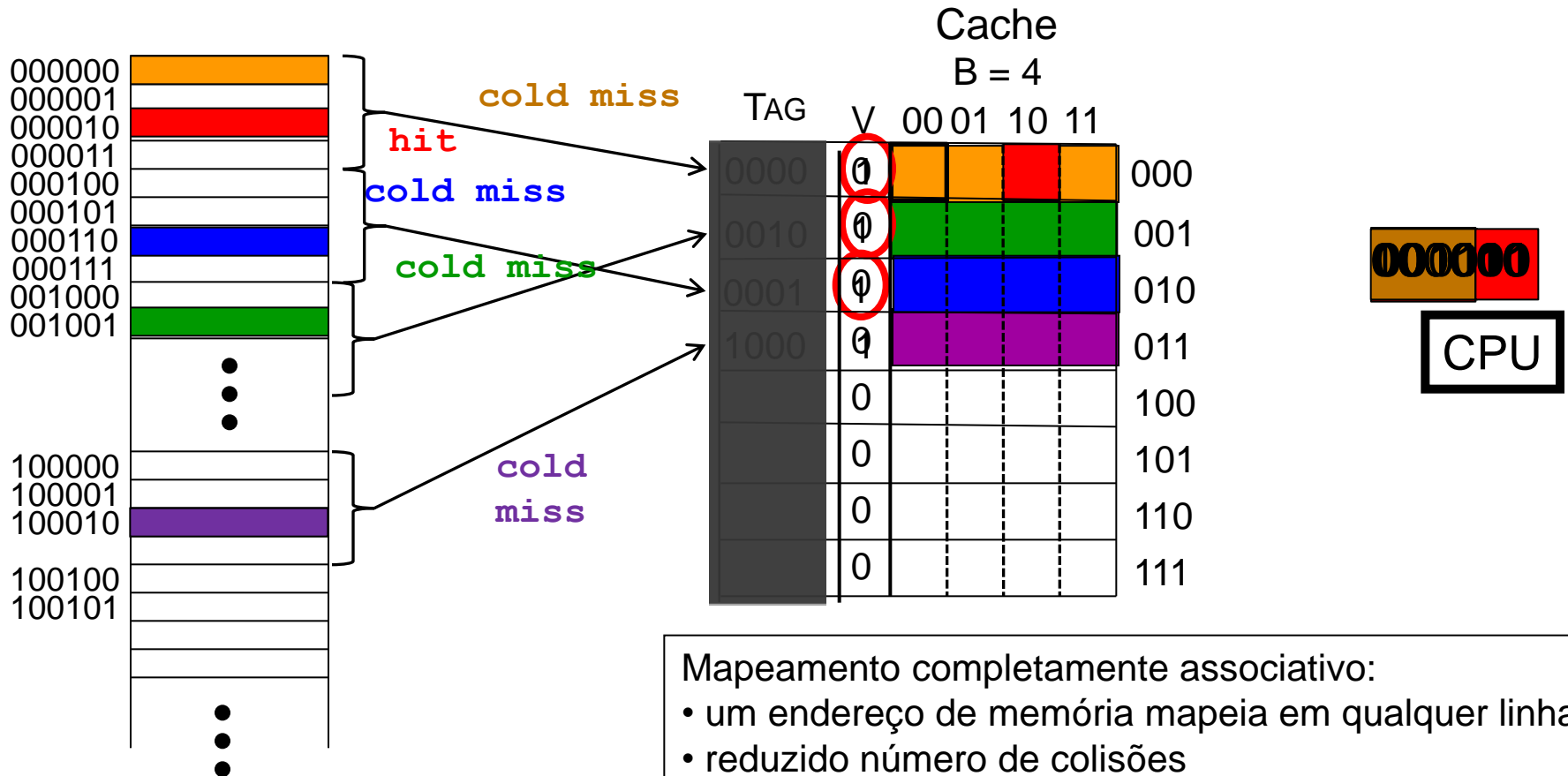
set index – $s=0$ bits não é utilizado pois só existe 1 *set*.

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

Considere uma máquina com um espaço de endereçamento de 32 *bits*, uma cache de 64 Kbytes, mapeamento completamente associativo e blocos de 32 bytes.

Quantos *bits* são necessários para a tag?

- A cache tem 1 *set* (*set index* = 0 *bits*).
- *bloco offset* são $b = \log_2(32) = 5$ *bits*.
- A tag será de $t = m - s - b = 32 - 0 - 5 = 27$ *bits*.

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.

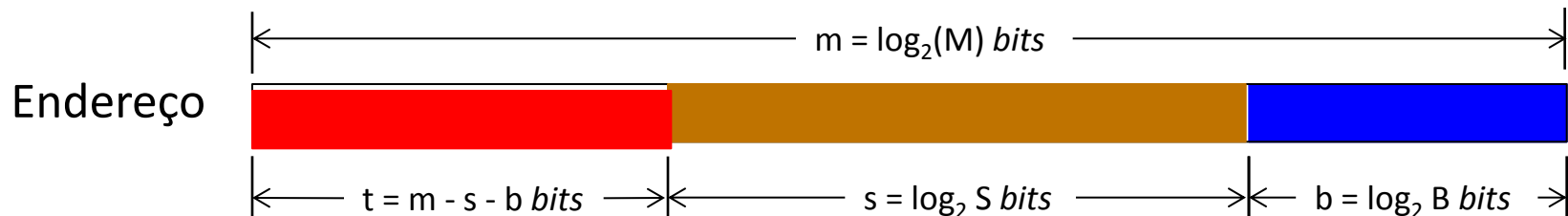
Dentro de cada *set* o bloco pode ser colocado em qualquer linha.

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)



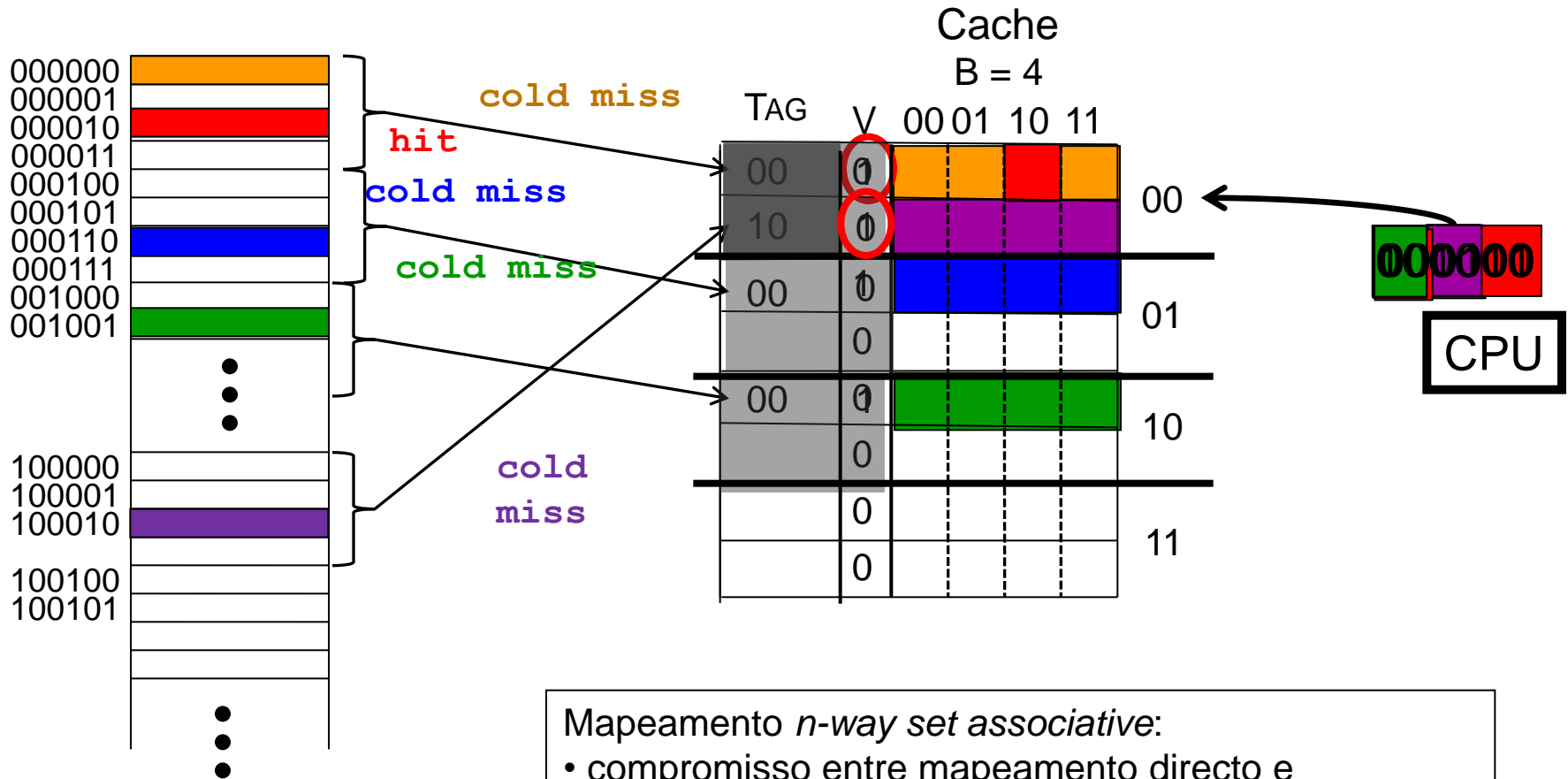
block offset – b bits identificam o byte dentro do bloco.

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*

Considere uma máquina com um espaço de endereçamento de 32 *bits*, uma cache de 64 Kbytes, mapeamento 8-way *set associative* e blocos de 32 bytes.

Quantos *bits* são necessários para a tag?

- 8 linhas por *set* ($E = 8$).
- $64K / (8 \cdot 32) = 2^6 \cdot 2^{10} / (2^3 \cdot 2^5) = 2^8$ *sets* ($s=8$ bits)
- *bloco offset* são $b = \log_2(32) = 5$ *bits*.
- A tag será de $t = m - s - b = 32 - 8 - 5 = 19$ *bits*.

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 lido de memória para a *cache*, escrito na *cache* e, se se tratar de *write-through*, também na memória ;
 - ***no-write-allocate*** - o bloco é escrito apenas na memória

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?

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

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

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]