

Aulas 24 e 25

◆ A memória *cache*

- Hierarquia de memória com 2 níveis
- Princípio de funcionamento da cache
 - ◆ localidade temporal
 - ◆ localidade espacial
- Cache com mapeamento associativo
- Cache com mapeamento directo
- Cache com mapeamento parcialmente associativo

Introdução

- ◆ A hierarquia de memória combina uma memória adaptada à velocidade do processador (de pequena dimensão) com uma (ou mais) menos rápida, mas de maior dimensão
- ◆ Uma vez que a memória com que o processador interage directamente é de pequena dimensão, a eficiência da hierarquia resulta do facto de se mover informação para a memória rápida poucas vezes e de se aceder a essa informação muitas vezes (antes de a substituir)
- ◆ Ou seja, para se tirar partido deste esquema, a **probabilidade** de que a informação necessária esteja no nível mais elevado da hierarquia tem que ser **elevada**
- ◆ O que torna essa probabilidade elevada ?

Introdução (2)

- ◆ Um programa não acede, tipicamente, a todo o seu código (ou dados) ao mesmo tempo com igual probabilidade
- ◆ Assim, um programa acede a uma zona reduzida do espaço de endereçamento numa dada fracção de tempo (princípio da localidade)
- ◆ Há dois tipos diferentes de localidade:
 - **Localidade no espaço (*spatial locality*)**
 - ◆ Se existe um acesso a uma zona de memória então é provável que as zonas contíguas sejam também acedidas
 - **Localidade no tempo (*temporal locality*)**
 - ◆ Se existe um acesso a uma zona de memória então é provável que essa mesma zona seja acedida novamente num futuro próximo

Introdução (3)

◆ Localidade espacial:

"a informação que o processador necessita de seguida, tem uma elevada probabilidade de estar próxima da que consome agora"

- Exemplos: instruções de um programa; processamento de um *array*
- Quanto maior for o bloco de informação existente na memória rápida, melhor

◆ Localidade temporal:

"a informação que o processador consome agora tem uma elevada probabilidade de ser novamente necessária num curto espaço de tempo"

- Exemplo: variável de controlo de um ciclo, instruções de um ciclo
- Quantos mais blocos de informação estiverem na memória rápida, melhor

Hierarquia de memória com 2 níveis

- ◆ Nível primário (superior) rápido e de pequena dimensão
- ◆ Nível secundário (inferior) mais lento mas de maior dimensão
 - O nível primário contém os blocos de memória mais recentemente utilizados
- ◆ Os pedidos de informação são sempre dirigidos ao nível primário, sendo o nível secundário envolvido apenas quando a informação pretendida não está nesse nível
 - Se os dados pretendidos se encontram num bloco do nível primário então existe um **hit**
 - Caso contrário ocorre um **miss**
- ◆ Na ocorrência de um *miss* é efectuado o acesso ao nível secundário para obter os dados (transferindo o bloco que contém a informação pretendida)

Hierarquia de memória com 2 níveis (2)

- ◆ A taxa de sucesso (*hit ratio*) é dada por:

$$\text{hit_ratio} = \frac{\text{Nr_hits}}{\text{Nr_total_acessos}}$$

- ◆ A taxa de insucesso (*miss ratio*) é dada por:

$$\text{miss_ratio} = 1 - \text{hit_ratio}$$

- ◆ O tempo de acesso no caso de um *hit* designa-se **hit time**
- ◆ O tempo de substituir um bloco do nível superior e enviar os dados para o processador é designado por **miss penalty**
- ◆ O **tempo médio de acesso** à informação é então:

$$t_a = \text{hit_ratio} * \text{hit_time} + (1 - \text{hit_ratio}) * \text{penalty_time}$$

Cache

- ◆ **cache:** nível de memória que se encontra entre o CPU e a memória principal (primeiras utilizações no início da década de 60)
- ◆ É comum os computadores recentes incluírem 1 ou mais níveis de *cache*; a *cache* é muitas vezes integrada no mesmo *chip* do processador

Ex. Cache read

Cache recebe endereço (MAddr) do CPU
O bloco que contém MAddr está na cache?

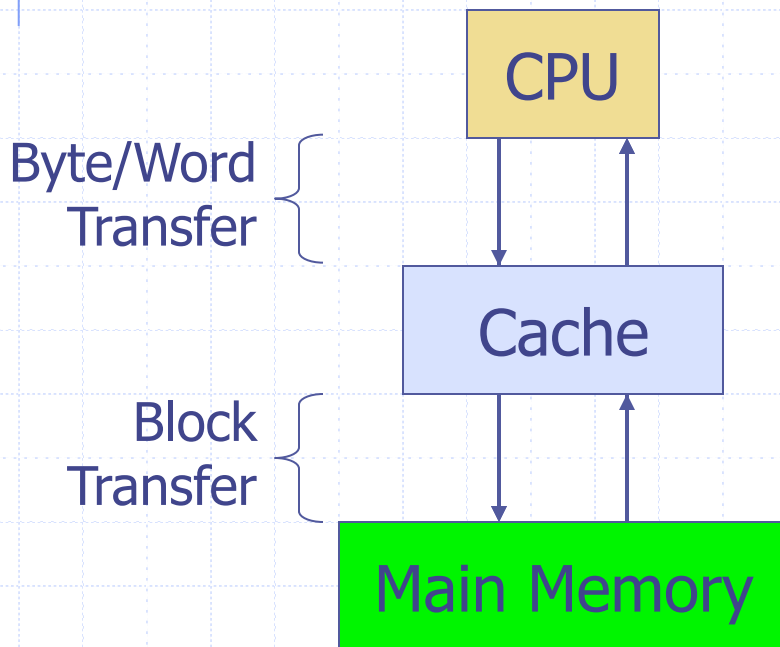
SIM:

Lê a cache e envia para o CPU o conteúdo de MAddr

NÃO:

Encontra espaço na cache para um novo bloco

Accede à memória principal e lê o bloco que contém o ender. MAddr
Lê a cache e envia para o CPU o conteúdo de MAddr



Cache (2)

- Exemplo:
memória de 64K (16 bits de endereço),
cache de 8 linhas de 4 bytes

	Cache			
Line 7				
Line 6	0A	49	6E	73
Line 5				
Line 4				
Line 3				
Line 2	77	B3	6F	8B
Line 1				
Line 0				

← cache line (4 bytes) →

Quais os endereços da memória principal representados nesta cache?

Memory	Address (16 bits)	
8B	FFFF	Block 3FFF
6F	FFFE	
B3	FFFD	
77	FFFC	
...		
6B	000B	Block 2
73	000A	
69	0009	
64	0008	
20	0007	Block 1
74	0006	
72	0005	
65	0004	
73	0003	Block 0
6E	0002	
49	0001	
0A	0000	

Cache (3)

- ◆ As operações da cache são transparentes para o programa; o programa emite um endereço e pede que seja feita uma operação de leitura ou escrita e esse pedido é satisfeito pelo sistema de memória: é desconhecido para o programa se é a cache ou a memória principal a satisfazer o pedido
- ◆ Na implementação da unidade de controlo da cache é necessário ter em atenção os seguintes aspectos:
 - Como saber se um determinado endereço está na cache?
 - Se está na cache, onde é que está?
 - Onde colocar um novo bloco na cache?
 - Qual o bloco a retirar da cache quando ocorre um miss?
 - Como tratar o problema das operações de escrita de modo a manter a coerência da informação nos vários níveis?
- ◆ Implementação tem de ser eficiente! Realizável em *hardware*

Organização dos sistemas de cache

- ◆ Há basicamente 3 formas de organizar os sistemas de cache:
 - Cache **totalmente associativa** ("fully associative")
 - Cache com **mapeamento directo** ("direct mapped")
 - Cache **parcialmente associativa** ("set associative")
- ◆ Os slides seguintes apresentam a metodologia de organização de cada uma delas, partindo do seguinte conjunto de especificações-base:
 - Espaço de endereçamento de 16 bits (memória principal com 64 kBytes)
 - Memória cache com 256 linhas (posições) de 8 bytes (2^3) cada (ou seja, cada bloco tem uma dimensão de 8 bytes).

Cache (3)

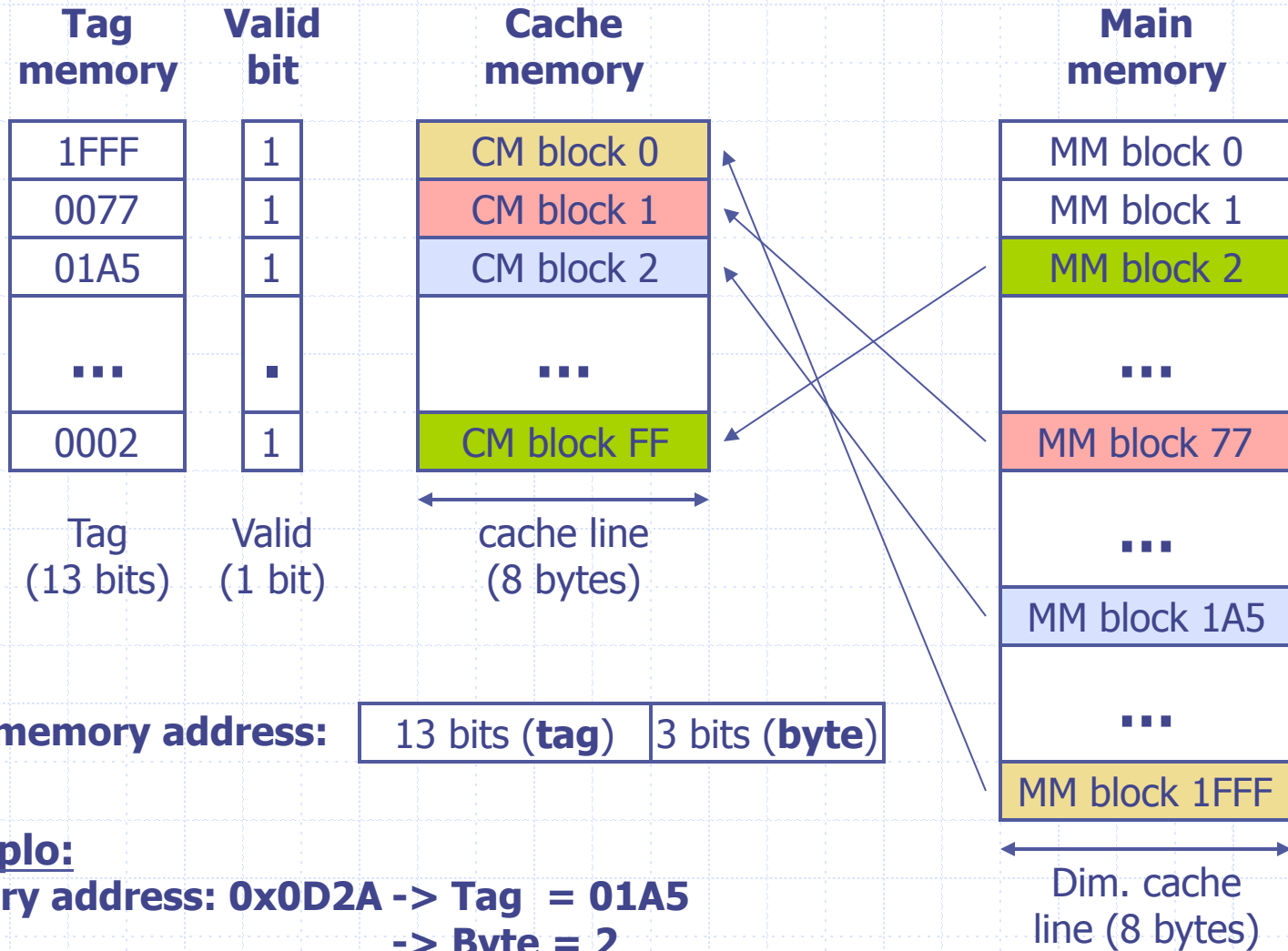
◆ Especificações-base:

- Espaço de endereçamento de 16 bits
- Memória cache com 256 posições de 8 bytes

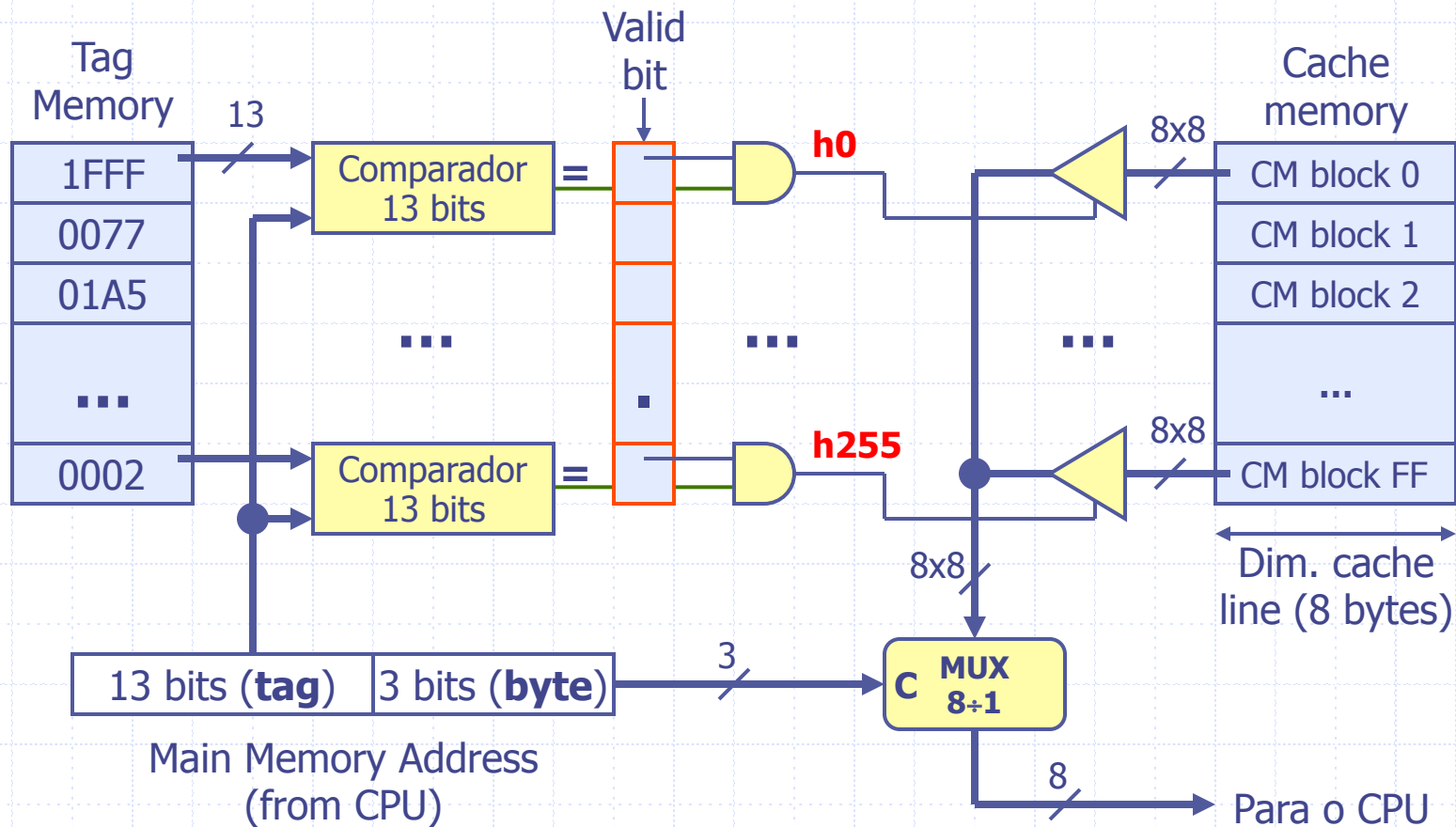
◆ Com estes valores, a memória principal pode ser vista como sendo constituída por um conjunto de 2^{13} blocos contíguos, de 8 bytes cada ($2^{16}/2^3 = 2^{13}$): 8k blocos (numerados de 0000 a 0x1FFF)

◆ Assim, no endereço de 16 bits, os 13 bits mais significativos identificam o **bloco** e os 3 bits menos significativos o **byte** dentro desse bloco

Cache com mapeamento associativo



Cache com mapeamento associativo (2)



$$\text{Cache hit} = h0 + h1 + \dots + h255$$

Cache com mapeamento associativo (3)

- ◆ (*Fully associative cache*)

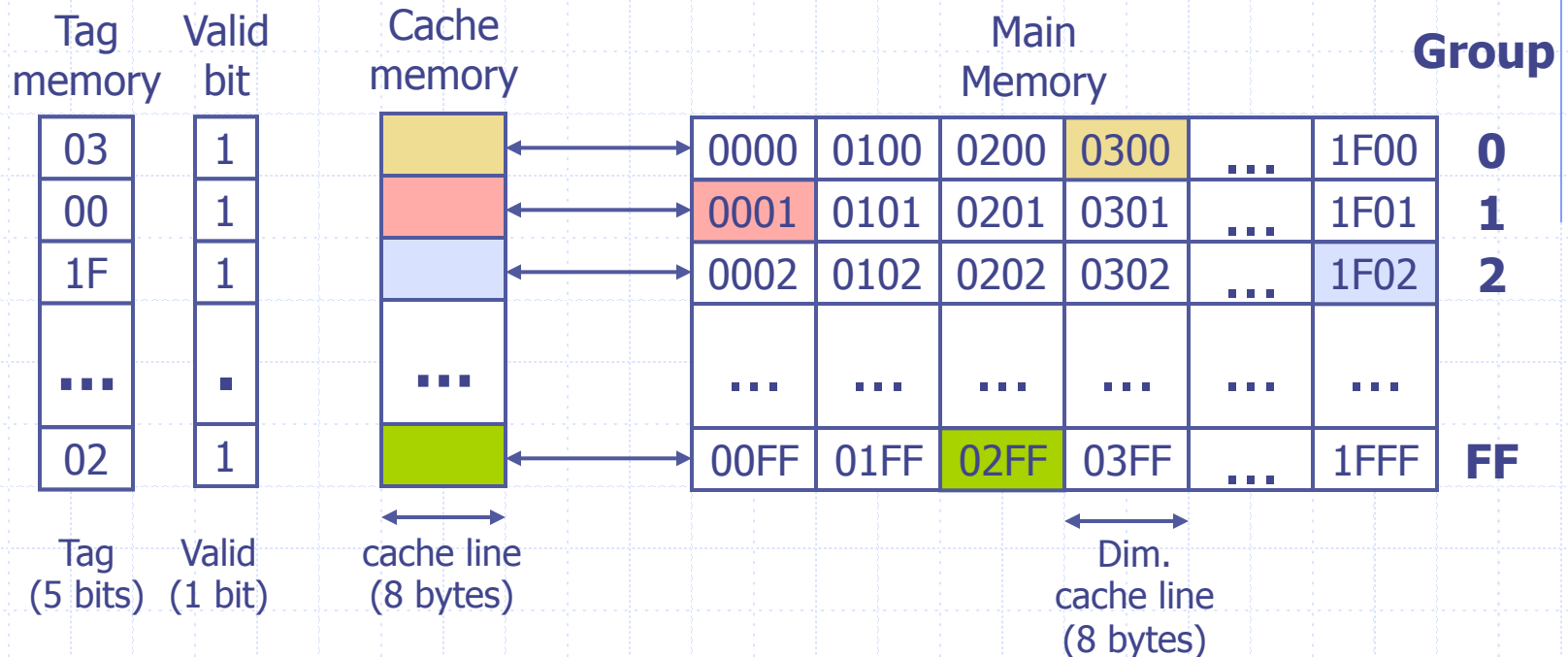
- ◆ Vantagens:

- Qualquer bloco da memória principal pode ser colocado em qualquer posição da cache

- ◆ Inconvenientes:

- Todas as entradas da memória "tag" têm de ser analisadas, de forma a verificar se um endereço se encontra na cache
- Muitos comparadores, alto custo

Cache com mapeamento directo



Main memory address: 5 bits (**tag**) | 8 bits (**group**) | 3 bits (**byte**)

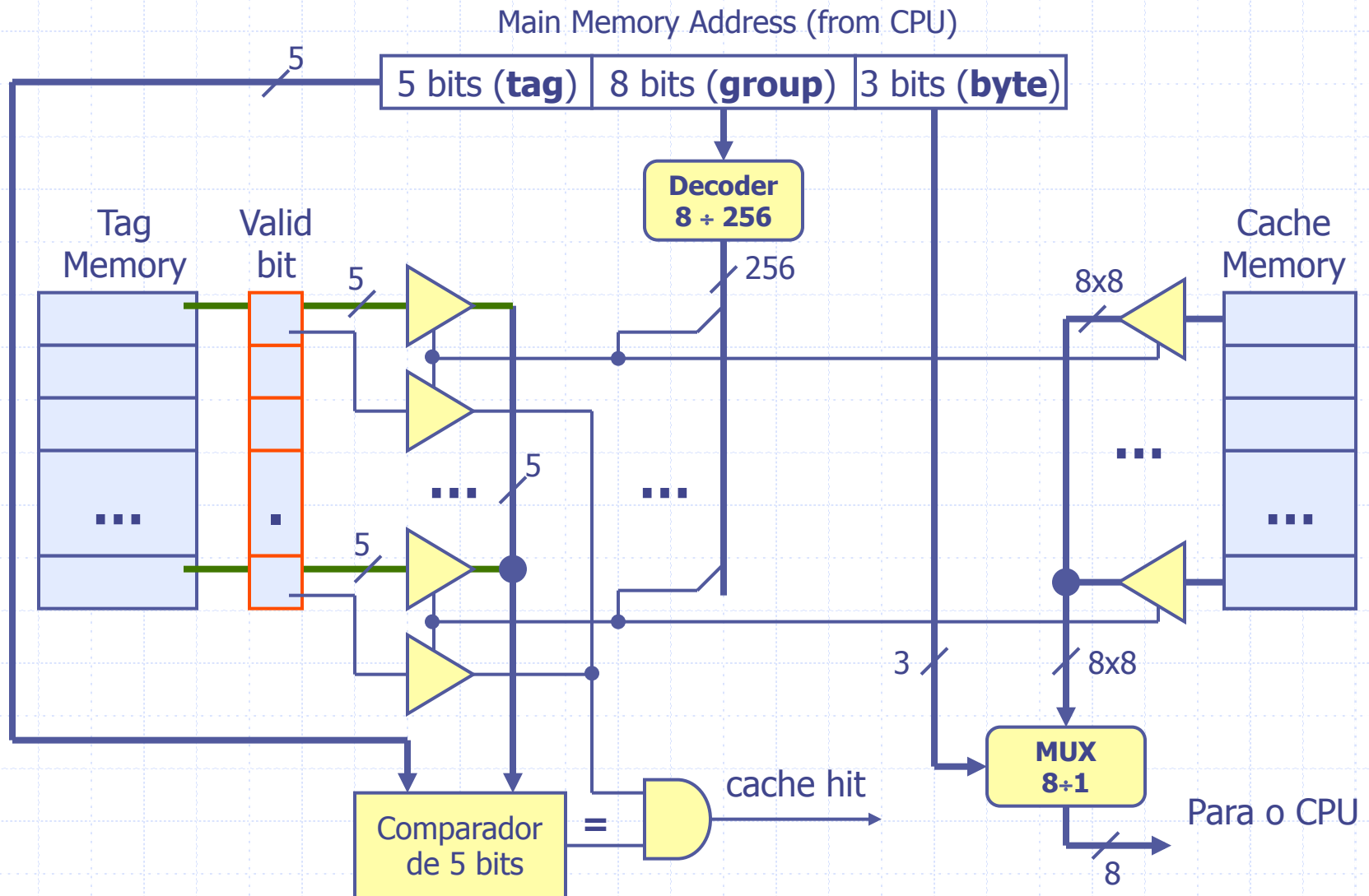
Exemplo:

Memory address = 0xF816 -> **Tag = 0x1F**
 -> **Group = 0x02**
 -> **Byte = 6**

Cache com mapeamento directo (2)

- ◆ O endereço do bloco é obtido, a partir do endereço real, por:
 - **Endereço do bloco = endereço real / dimensão do bloco**
(**Endereço do bloco = [tag group], i.e. "tag" concatenado com "group")**
- ◆ A posição da cache (i.e. a linha) associada a um dado endereço é dada por:
 - **Pos = endereço do bloco % número de blocos da cache**
(**pos = group**)
- ◆ O tamanho do bloco e o número de blocos da cache são potências de 2

Cache com mapeamento directo (3)

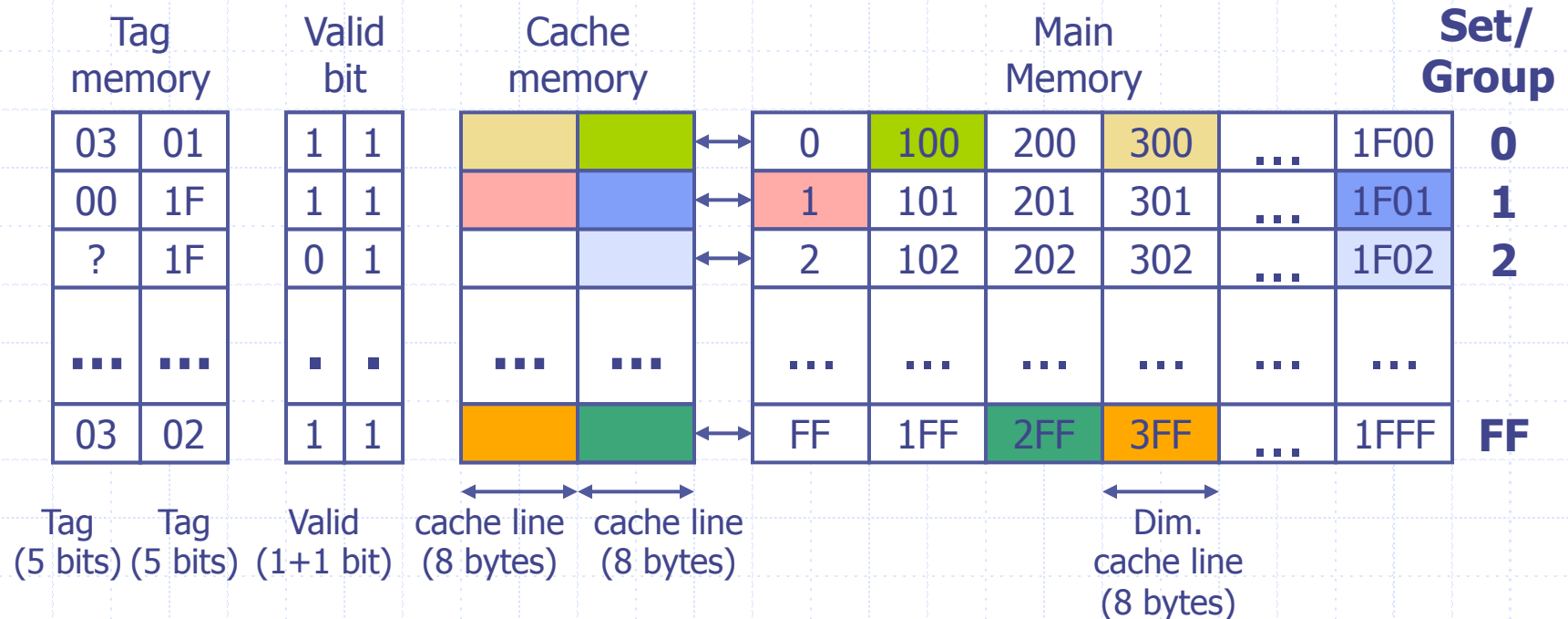


Cache com mapeamento directo (4)

- ◆ (*Direct-mapped cache*)
- ◆ A cada bloco da memória principal é associada uma linha da cache; o mesmo bloco é sempre colocado na mesma linha
- ◆ Vários blocos têm associada a mesma linha
- ◆ Maior vantagem: simplicidade de implementação
- ◆ Maior desvantagem:
 - Num dado instante apenas um bloco de um dado grupo pode residir na cache, o que tem como consequência que alguns blocos podem ser substituídos e recarregados várias vezes, mesmo existindo espaço na cache para guardar todos os blocos em uso
- ◆ Uma melhoria óbvia deste tipo de cache seria permitir o armazenamento simultâneo de mais que um bloco do mesmo grupo; é esta melhoria que é explorada na cache **parcialmente associativa**

Cache com mapeamento parcialmente associativo

(exemplo com associatividade de 2)



Main memory address:

5 bits (**tag**)

8 bits (**set**)

3 bits (**byte**)

Exemplo:

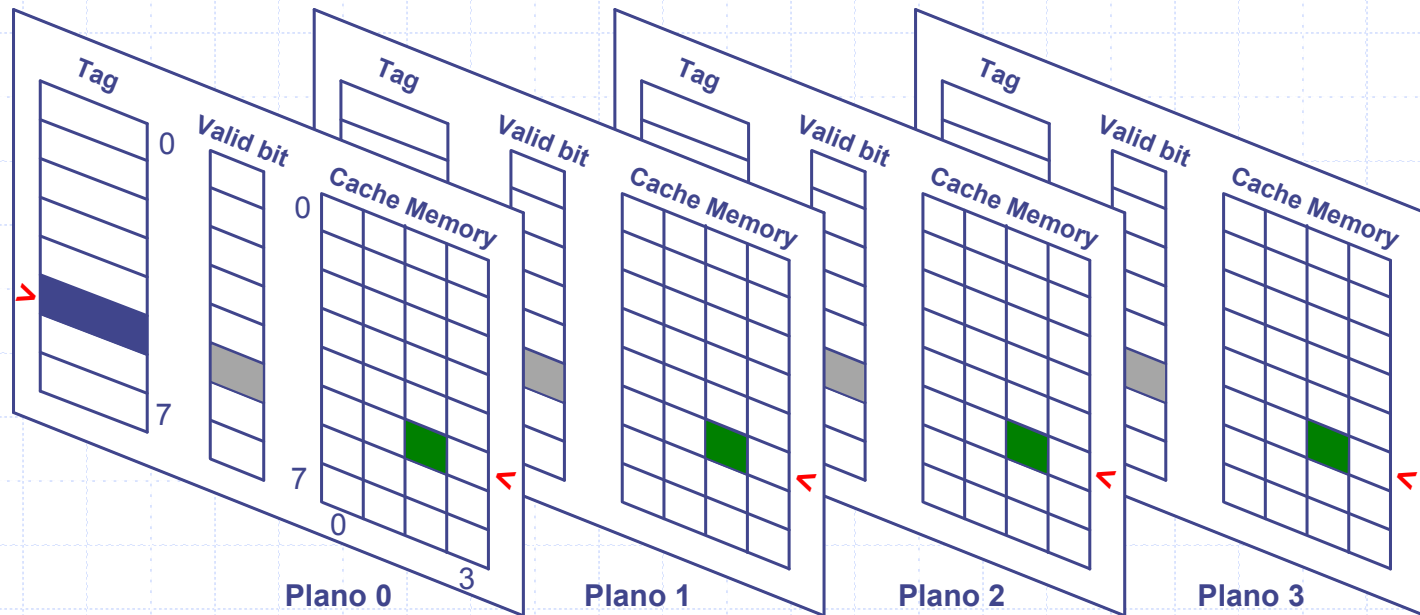
Memory address = 0x17FC -> Tag = 0x02

-> Set/Group = 0xFF

-> Byte = 4

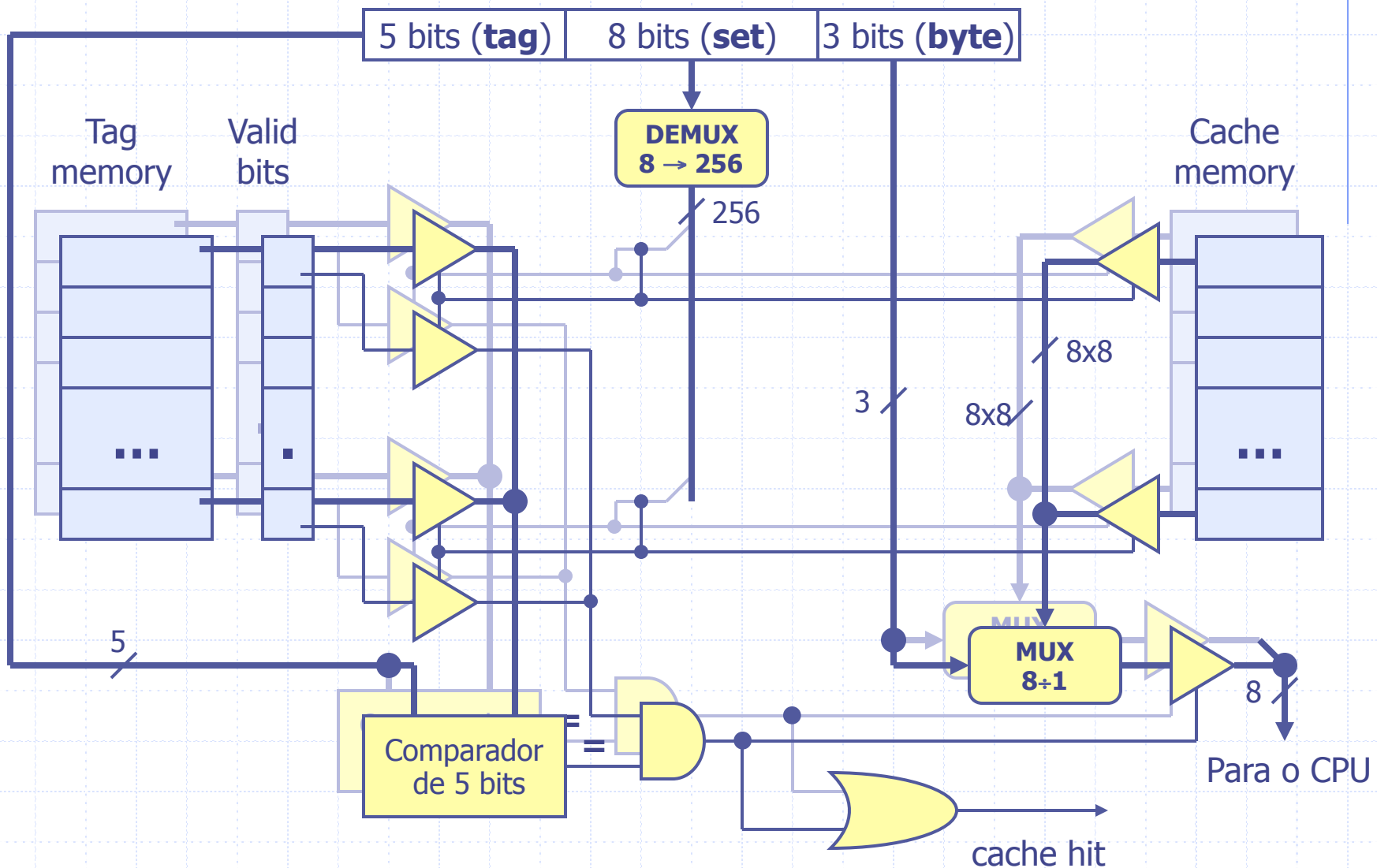
Cache com mapeamento parcialmente associativo

- ◆ Esquematização de uma *cache* com associatividade de 4, de 128 bytes, com 8 linhas (128 / 4 = 32 bytes por plano, 32 / 8 = 4 bytes por linha)
 - Se o endereço gerado pelo CPU for, em binário (16 bits):
 - ◆ **0110101010010110**



- A comparação dos 11 bits mais significativos (A_{15} - A_5) do endereço com as *tags* é feita **simultaneamente** nos 4 planos
- A posição da memória *tag* onde é feita a comparação é determinada pelos bits A_4 - A_2 do endereço (posição 5 no exemplo)
- O plano onde ocorre o *hit* (se ocorrer) fornece o *byte* armazenado na *cache* na posição determinada pelos bits A_1 - A_0 do endereço (posição 2 no exemplo)

Cache parcialmente associativa (2)



Cache parcialmente associativa (3)

- ◆ (*Block-set associative cache / n-way-set associative cache*)
- ◆ A cache parcialmente associativa é semelhante à cache com mapeamento directo, mas a primeira permite que mais que um bloco de um mesmo grupo possa estar na cache
- ◆ Uma cache com associatividade de 2 permite que 2 blocos do mesmo grupo possam estar simultaneamente na cache
- ◆ A divisão do endereço de memória é igual ao do mapeamento directo (o campo "group" é normalmente designado por "set" - conjunto), mas agora há "**n**" possíveis lugares onde um dado bloco de um mesmo grupo pode residir (para uma cache com associatividade de "**n**")
- ◆ Os "**n**" possíveis lugares onde o bloco pode residir têm que ser procurados simultaneamente

Estratégias de substituição

- ◆ As estratégias de substituição de blocos na cache, na ocorrência de um miss, são várias; as mais utilizadas são as seguintes:
 - **LRU** - *Least Recently Used*
 - ◆ É substituído o bloco da cache que está há mais tempo sem ser referenciado
 - **LFU** - *Least Frequently Used*
 - ◆ É substituído o bloco menos acedido
 - **FIFO** - *First-in-first-out*
 - ◆ É substituído o bloco que foi carregado há mais tempo
 - **Random:**
 - ◆ Substituição aleatória (testes indicam que não é muito pior do que LRU)

Políticas de escrita

◆ Write-through

- Todas as escritas são realizadas simultaneamente na cache e na memória principal
- A memória principal está sempre consistente
- Se dado ausente na cache, actualiza apenas a memória principal (**write-no-allocate**)

◆ Write-back

- Valor escrito apenas na cache; novo valor é escrito na memória quando o bloco da cache é substituído
- **Dirty bit** (esta flag é activada quando houver uma escrita em qualquer endereço do bloco presente na linha da cache)
- Mais complexo do que *write-through*
- Se dado ausente na cache, carrega o bloco para a cache e actualiza-o (**write-allocate**)

Exercícios

1. Qual a posição dos endereços de bloco 1 e 29 numa cache de mapeamento directo com 8 linhas?
2. Considere um sistema computacional com um espaço de endereçamento de 32 bits e uma cache com 256 blocos de 8 bytes cada um.
 - Qual o tamanho em bits dos campos *tag*, *group* e *byte* supondo que a cache é: 1) associativa; 2) mapeamento directo; 3) com associatividade de 2?
3. Para a implementação de uma cache com 64KB de dados e blocos de 4 bytes num sistema computacional com um espaço de endereçamento de 32 bits, quantos bits de armazenamento são necessários supondo: 1) uma cache associativa; 2) uma cache com mapeamento directo; 3) uma cache com associatividade de 2.