

Aulas 22 e 23

- Organização da memória de um sistema computacional
- Hierarquia do sistema de memória
- Localidade temporal e espacial
- A memória cache
 - Princípio de funcionamento
 - Cache com mapeamento associativo
 - Cache com mapeamento direto
 - Cache com mapeamento parcialmente associativo

José Luís Azevedo, Bernardo Cunha, Tomás Oliveira e Silva

Introdução

- Pretensão do utilizador:
 - Uma memória rápida e com grande capacidade de armazenamento
 - Que custe o preço de uma memória lenta... ☺
- Solução perfeita para este dilema não existe

Tecnologia	Tempo Acesso	\$ / GB
SRAM	0,5 – 2,5 ns	\$500 - \$1000
DRAM	35 - 70 ns	\$10 - \$20

(Dados de 2012)

- A organização da memória de um sistema computacional resulta de um compromisso entre:
 - Velocidade, Capacidade, Custo, Consumo energético
- Menor tempo de acesso: maior custo por bit
- Maior capacidade: maior tempo de acesso

Introdução

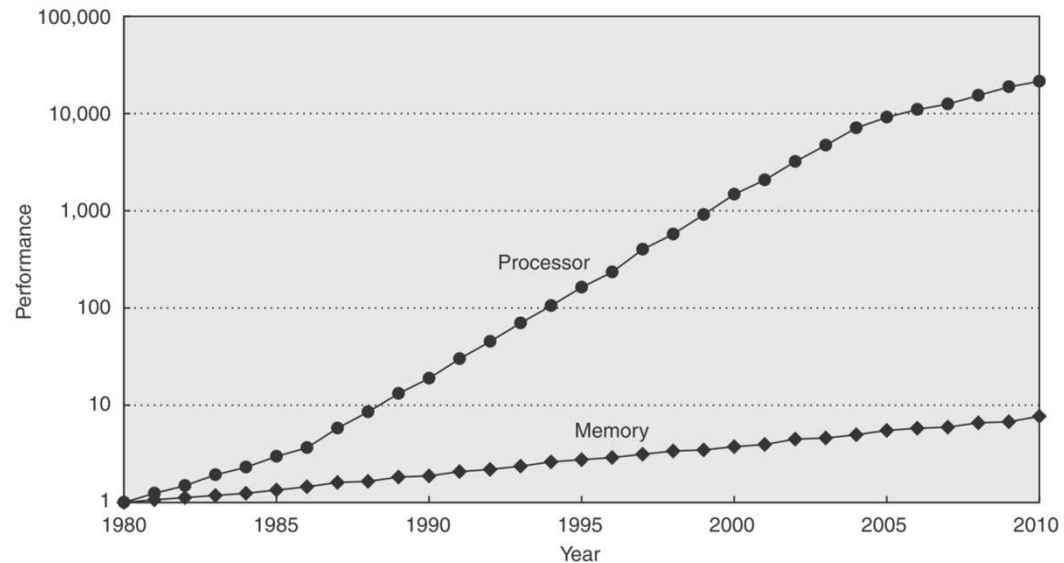
- Memória DRAM (Dynamic RAM)

Ano	Capacidade (max. por chip)	Access Time	\$ / Mb
1980	64 kbit	250 ns	\$1500
1983	256 kbit	185 ns	\$500
1985	1 Mbit	135 ns	\$200
1989	4 Mbit	110 ns	\$50
1992	16 Mbit	90 ns	\$15
1996	64 Mbit	60 ns	\$10
1998	128 Mbit	60 ns	\$4
2000	256 Mbit	55 ns	\$1
2004	512 Mbit	50 ns	\$0.25
2007	1024 Mbit	45 ns	\$0.05
2010	2 Gbit	40 ns	\$0.03
2012	4 Gbit	35 ns	\$0.001

Introdução

- O processador deve ser alimentado de instruções e dados a uma taxa que não comprometa o desempenho do sistema

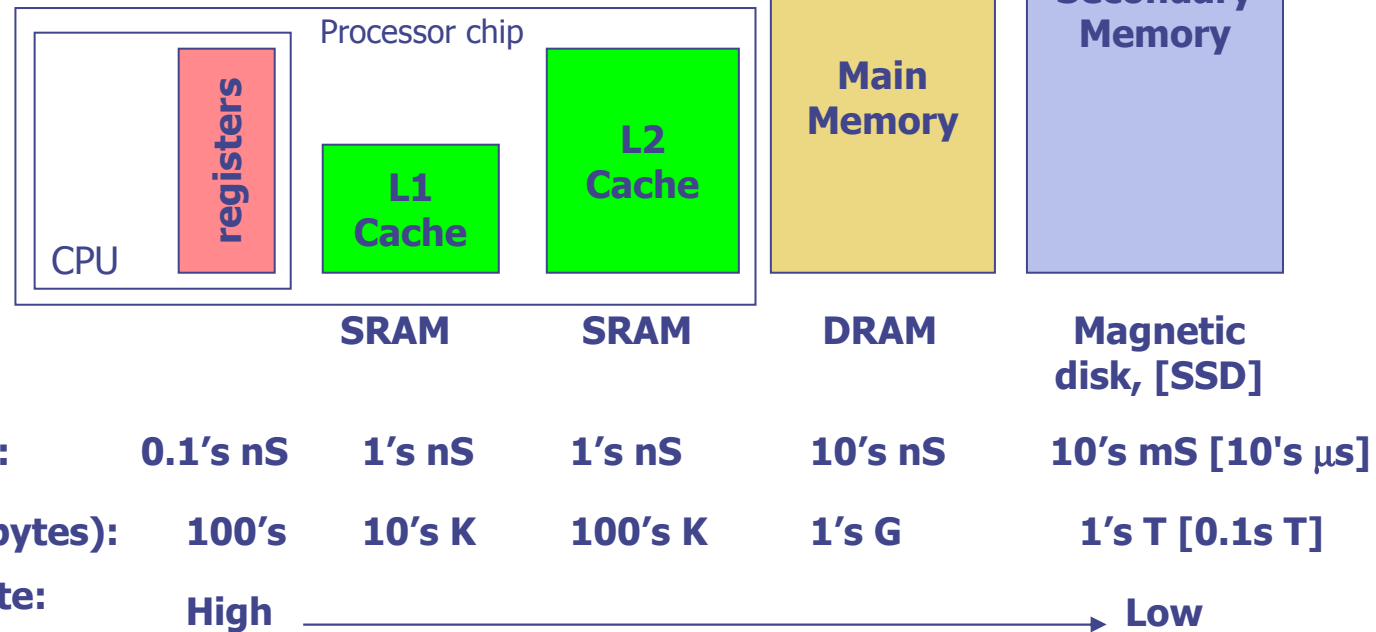
- A diferença entre o desempenho do processador e da memória (DRAM) tem vindo a aumentar



- Solução:
 - Guardar a informação mais vezes utilizada pelo CPU numa memória rápida (*static* RAM) de pequena capacidade
 - Aceder raramente à memória principal (mais lenta) para obter a informação em falta (apenas quando necessário)
 - Transferir blocos de informação da memória principal para a memória rápida
- Conceito: **cache**

Hierarquia de memória

- Memória organizada em níveis
- A informação nos níveis superiores é um subconjunto da dos níveis inferiores



- A informação circula apenas entre níveis adjacentes da hierarquia
- **Bloco** – quantidade de informação que circula entre níveis adjacentes (n bytes)

Hierarquia de memória

- Solução 1 – expor a hierarquia
 - Alternativas de armazenamento: registos internos do CPU, memória rápida, memória principal, disco
 - Cabe ao programador utilizar racionalmente estas alternativas de armazenamento
 - Exemplo de processador que usa esta técnica: Cell microprocessor (Cell Broadband Engine Architecture) que equipa a PlayStation 3 e algumas televisões
- Solução 2 – esconder a hierarquia
 - Modelo de programação:
 - Tipo de memória único
 - Espaço de endereçamento único
 - A máquina gere automaticamente o acesso ao sistema de memória
 - Solução usada na maioria dos processadores contemporâneos

Hierarquia de memória

- 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
- A memória rápida armazena um sub-conjunto da informação residente na memória principal.
- Uma vez que a memória com que o processador interage diretamente é de pequena dimensão, a eficiência da hierarquia resulta do facto de se copiar informação para a memória rápida poucas vezes e de se aceder a essa informação muitas vezes (antes de surgir a necessidade de a substituir)
- Para se tirar partido deste esquema, a probabilidade de a informação (que o processador necessita) estar nos níveis mais elevados da hierarquia tem que ser elevada
- O que torna essa probabilidade elevada ?

Localidade

- **Princípio da localidade:** os programas não acedem à memória (dados e instruções) de forma aleatória mas usam tipicamente endereços que se situam na vizinhança uns dos outros
- Ou seja, num dado intervalo de tempo um programa acede a uma zona reduzida do espaço de endereçamento
- O princípio da localidade manifesta-se de duas formas:
 - **Localidade no espaço (spatial locality):** Se existe um acesso a um endereço de memória então é provável que os endereços contíguos sejam também acedidos
 - **Localidade no tempo (temporal locality):** Se existe um acesso a um endereço de memória então é provável que esse mesmo endereço seja acedido novamente no futuro próximo

Localidade

- **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 (zona contígua de memória) de informação existente na memória rápida, melhor

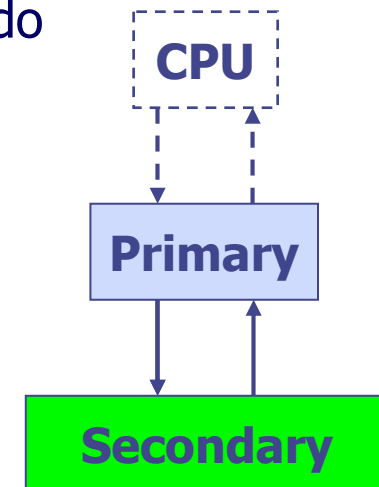
- **Localidade temporal:**

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

- Exemplos: 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 pelo CPU
- 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" acede-se ao nível secundário e transfere-se o bloco que contém a informação pretendida



Hierarquia de memória com 2 níveis

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

$$hit_{ratio} = \frac{nr_{hits}}{nr_{accesses}}$$

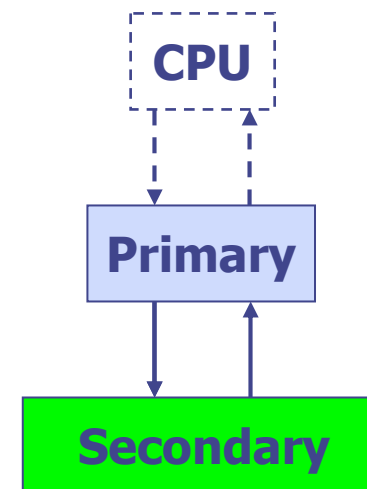
- A taxa de insucesso (**miss ratio**) é dada por:

$$miss_{ratio} = 1 - hit_{ratio}$$

- 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 **penalty time (miss penalty)**
- De um modo simplificado, o "penalty time" é dado por:

$$penalty_{time} = hit_{time} + t_{mem}$$

em que " t_{mem} " é o tempo de acesso ao nível secundário



Hierarquia de memória com 2 níveis

- O tempo médio de acesso à informação é então:

$$T_{access} = hit_{ratio} * hit_{time} + (1 - hit_{ratio}) * penalty_{time}$$

$$T_{access} = hit_{ratio} * hit_{time} + (1 - hit_{ratio}) * (hit_{time} + t_{mem})$$

- **Exercício:** Assumindo um *hit_ratio* de 95%, um tempo de acesso ao nível superior de 5ns e um tempo de acesso ao nível inferior de 50ns, calcular o tempo médio de acesso à memória
 - $T_a = 0,95 * 5 + 0,05 * (50 + 5) = 7,5ns$
Ou seja, aproximadamente 6,7 vezes mais rápido que o acesso direto ao nível secundário
- Quando o acesso a uma palavra no nível superior falha, acede-se ao nível seguinte não apenas a essa palavra, mas também às que estão em endereços adjacentes (vizinhas) - **bloco**

Memória cache

- **cache**: nível de memória que se encontra entre o CPU e a memória principal (primeiras utilizações no início da década de 60)

- Exemplo - **cache read**:

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

SIM (hit):

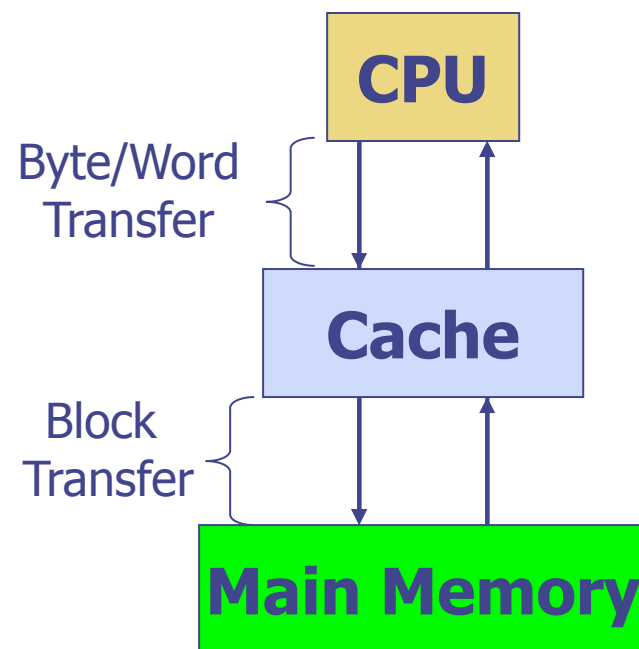
Lê a cache e envia para o CPU o conteúdo de MemAddr

NÃO (miss):

Encontra espaço na cache para um novo bloco

Acede à memória principal e lê o bloco que contém o endereço MemAddr

Conteúdo de MemAddr é enviado para o CPU



- É comum os computadores recentes incluírem 2 ou mais níveis de cache; a cache é normalmente integrada no mesmo circuito integrado do processador

Memória cache

- Exemplo: memória de 64K (16 bits de endereço), cache de 8 linhas de 4 bytes (32 bytes)

Cache

Line 7	77	B3	6F	8B
Line 6				
Line 5				
Line 4				
Line 3				
Line 2	64	69	73	6B
Line 1				
Line 0				

← cache line (4 bytes) →

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	

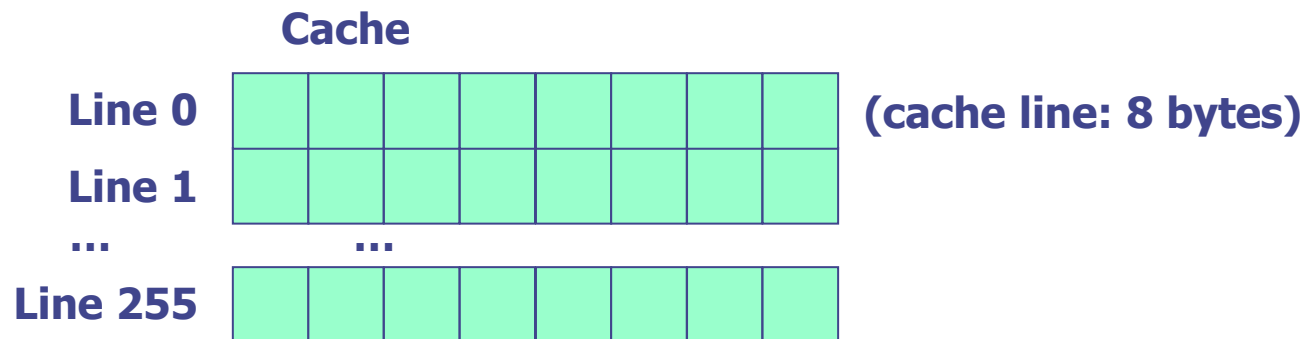
- Quais os endereços da memória principal representados nesta cache?

Memória cache

- As operações da cache são transparentes para o processador
- O processador gera 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 processador se é a cache ou a memória principal a satisfazer o pedido
- Na organização da cache e na implementação da respetiva unidade de controlo é necessário tomar em consideração um conjunto de aspetos, nomeadamente:
 - Como saber se um determinado endereço está na cache?
 - Se está na cache, onde é que está?
 - Quando ocorre um miss, onde colocar um novo bloco na cache?
 - Quando ocorre um miss, qual o bloco a retirar da cache?
 - 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 direto** ("direct mapped")
 - Cache **parcialmente associativa** ("set associative")
- A metodologia de organização de cada uma delas é apresentada a seguir, partindo do seguinte conjunto de especificações-base:
 - Espaço de endereçamento de 16 bits (memória principal organizada em bytes, com 64 kBytes)
 - Memória cache de 2 kBytes, em que cada linha tem 8 bytes (2^3) (ou seja, cada bloco tem uma dimensão de 8 bytes)



Organização dos sistemas de cache

- Especificação-base para os exemplos de organização que se seguem:
 - Espaço de endereçamento de 16 bits (0x0000 a 0xFFFF)
 - Memória cache de 2 kBytes, em que cada linha tem 8 bytes (2^3), significa que a cache tem 256 linhas ($2^{11} / 2^3 = 256$)
- 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
 - 3 bits menos significativos identificam o **byte dentro do bloco**
 - 13 bits mais significativos identificam o **bloco**

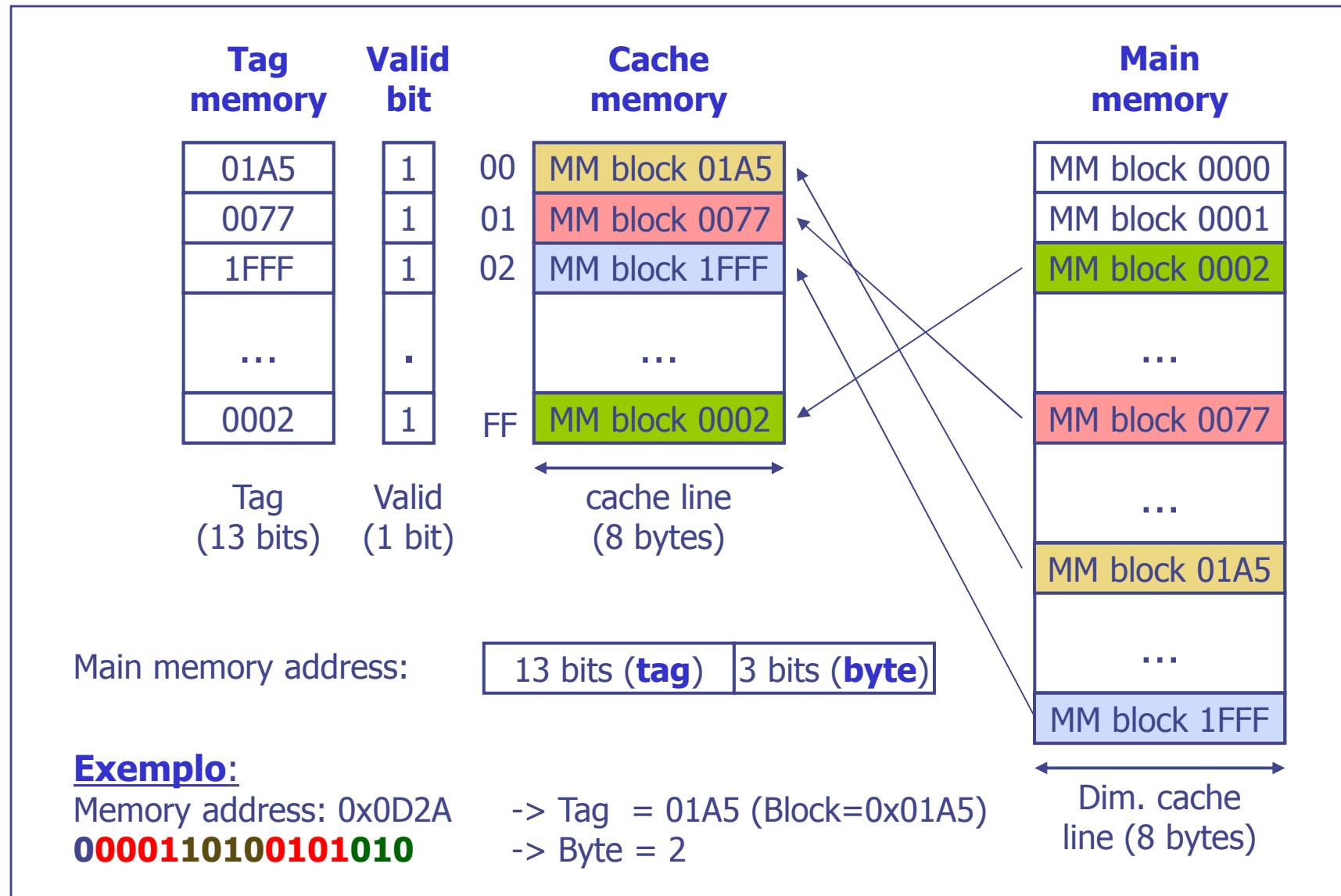
Main memory address:

15	0
13 bits (bloco)	3 bits (byte)

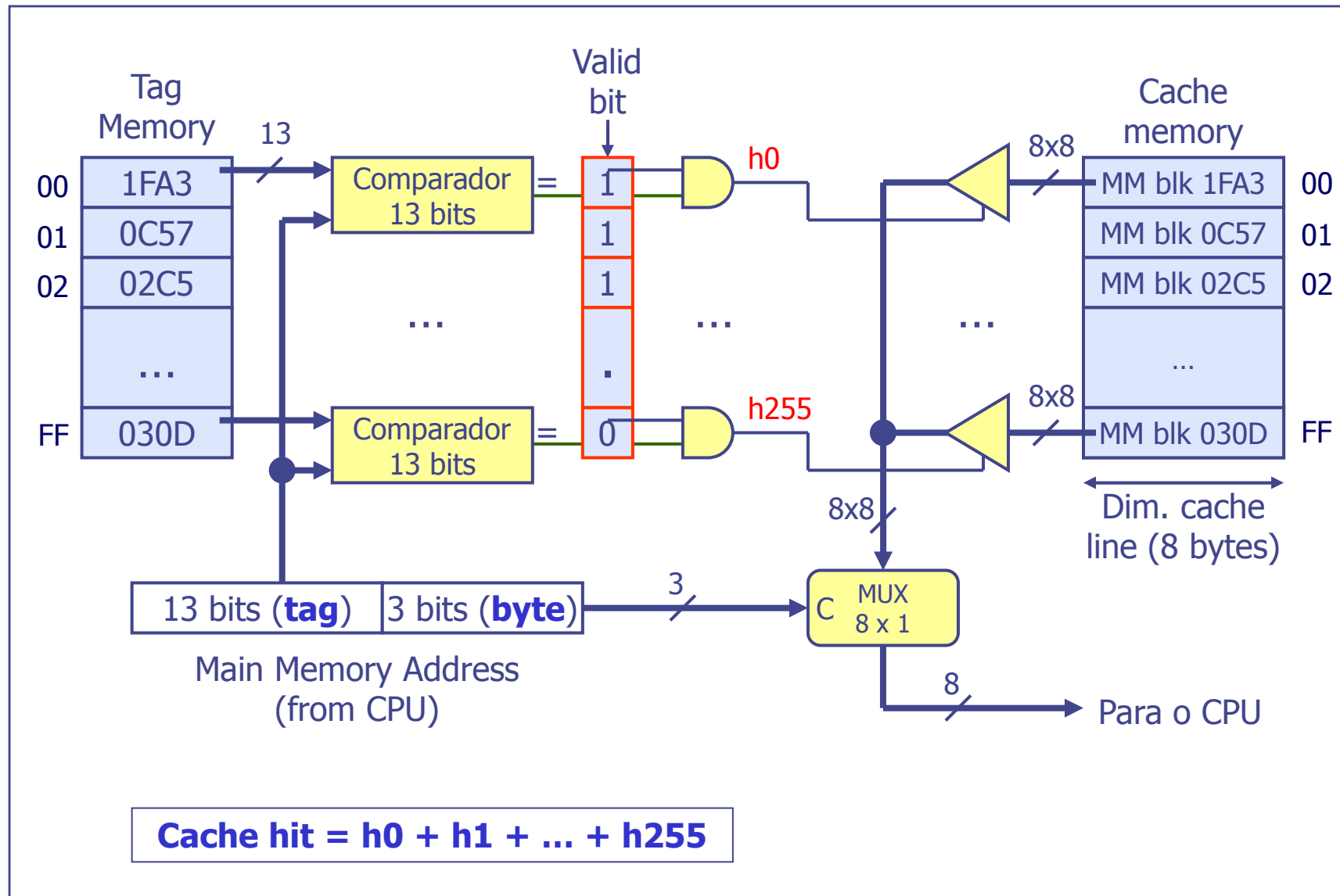
- Exemplo. Address = 0x001A : **00000000000011010**

Bloco 3, de 0x0018 a 0x001F, byte 2 dentro do bloco

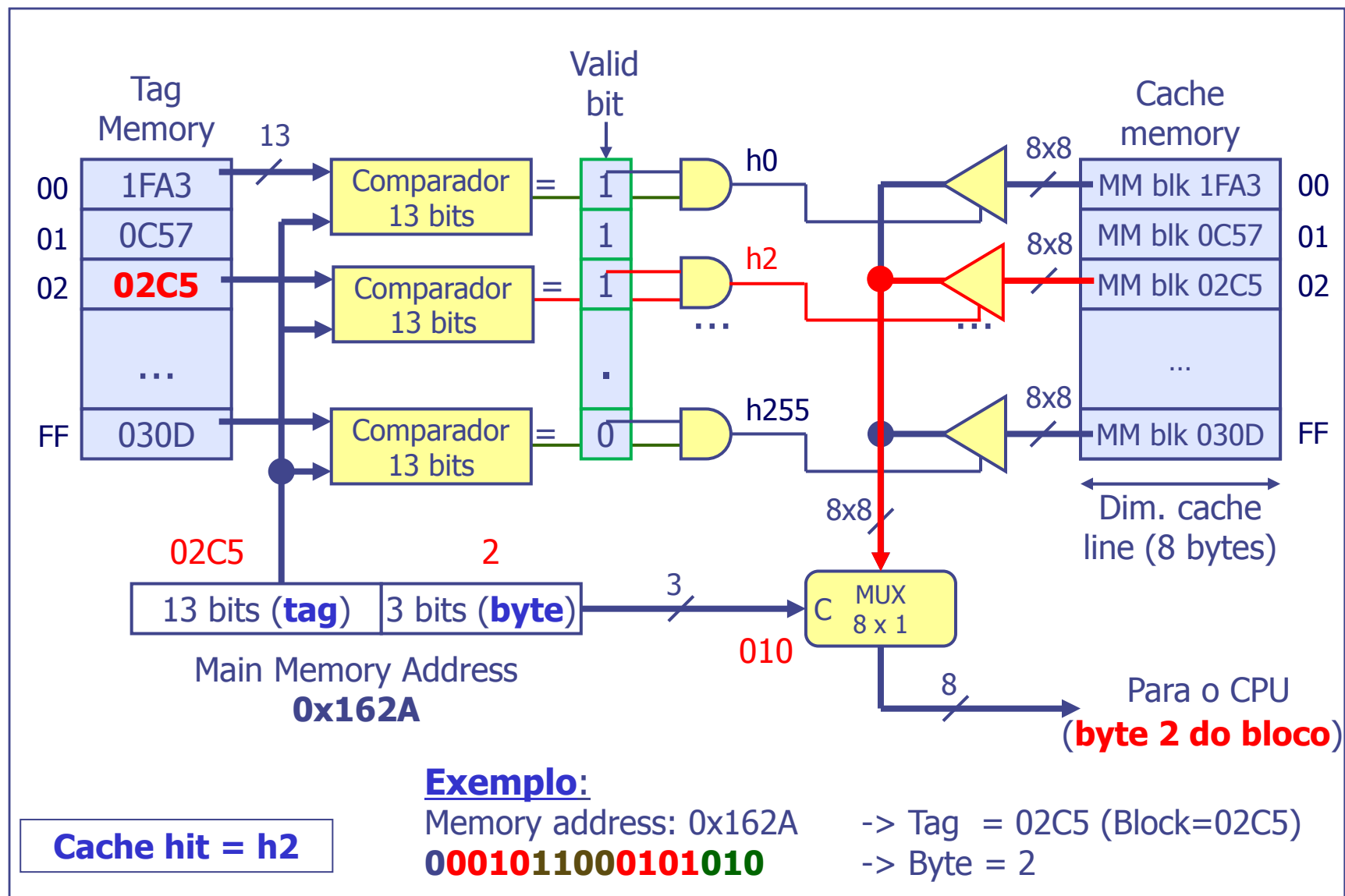
Cache com mapeamento associativo



Cache com mapeamento associativo



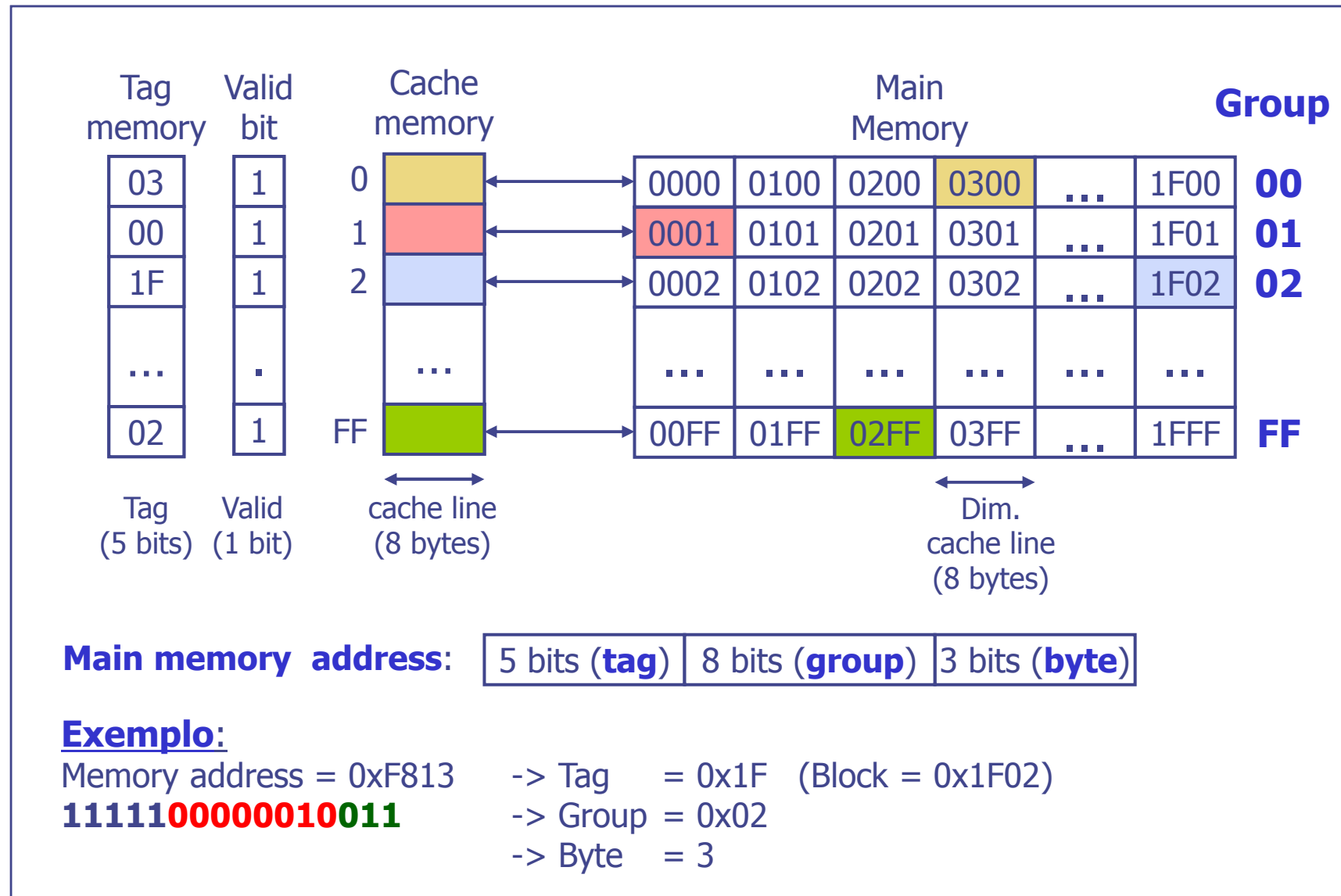
Cache com mapeamento associativo (exemplo)



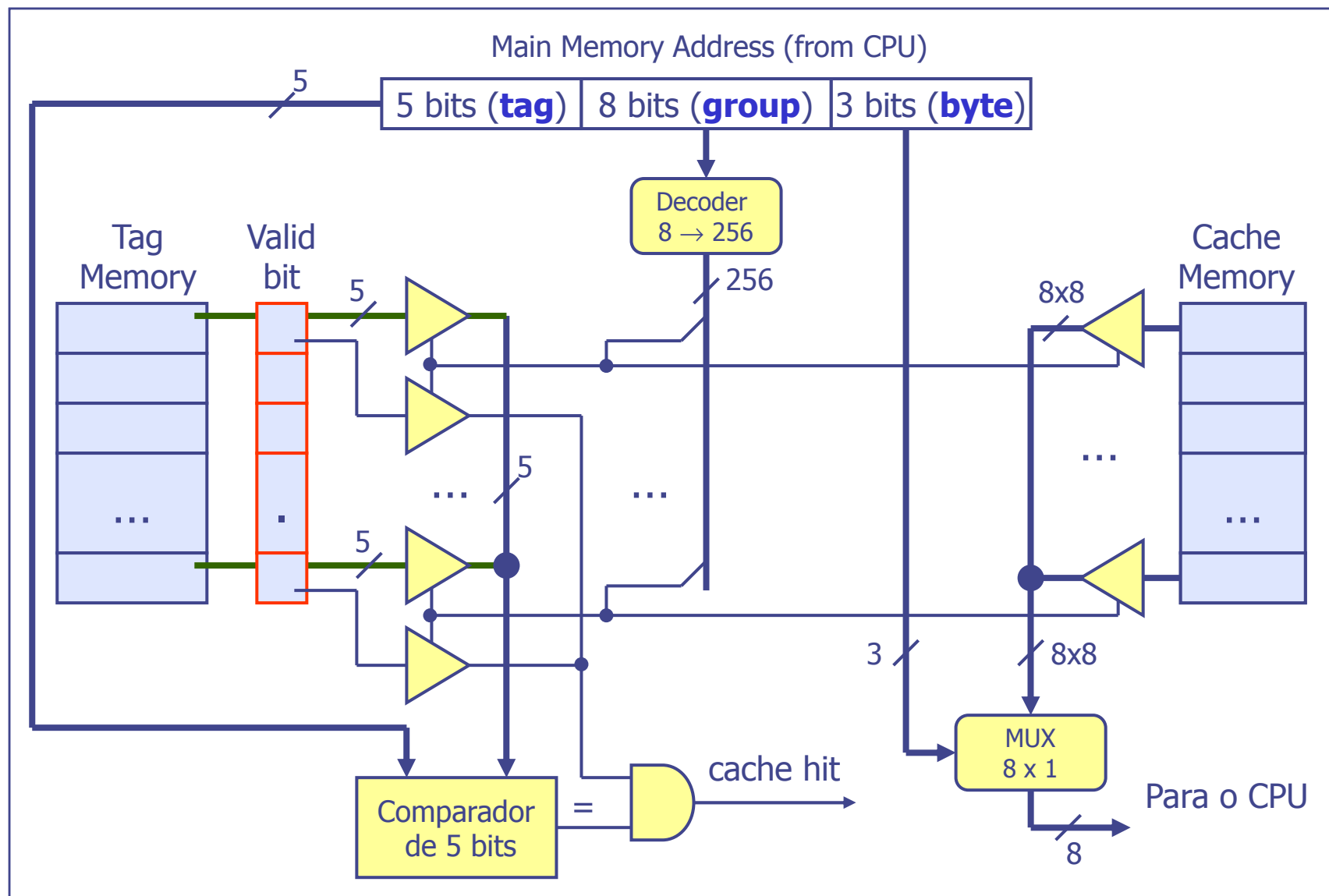
Cache com mapeamento associativo

- Vantagens:
 - Qualquer bloco da memória principal pode ser colocado em qualquer posição da cache
- Inconvenientes:
 - A "tag" tem que ter todos os bits do número do bloco, o que numa implementação realista origina comparadores com uma dimensão elevada
 - Todas as entradas da memória "tag" têm de ser analisadas, de forma a verificar se um endereço se encontra na cache
 - Muitos comparadores, custo elevado

Cache com mapeamento direto



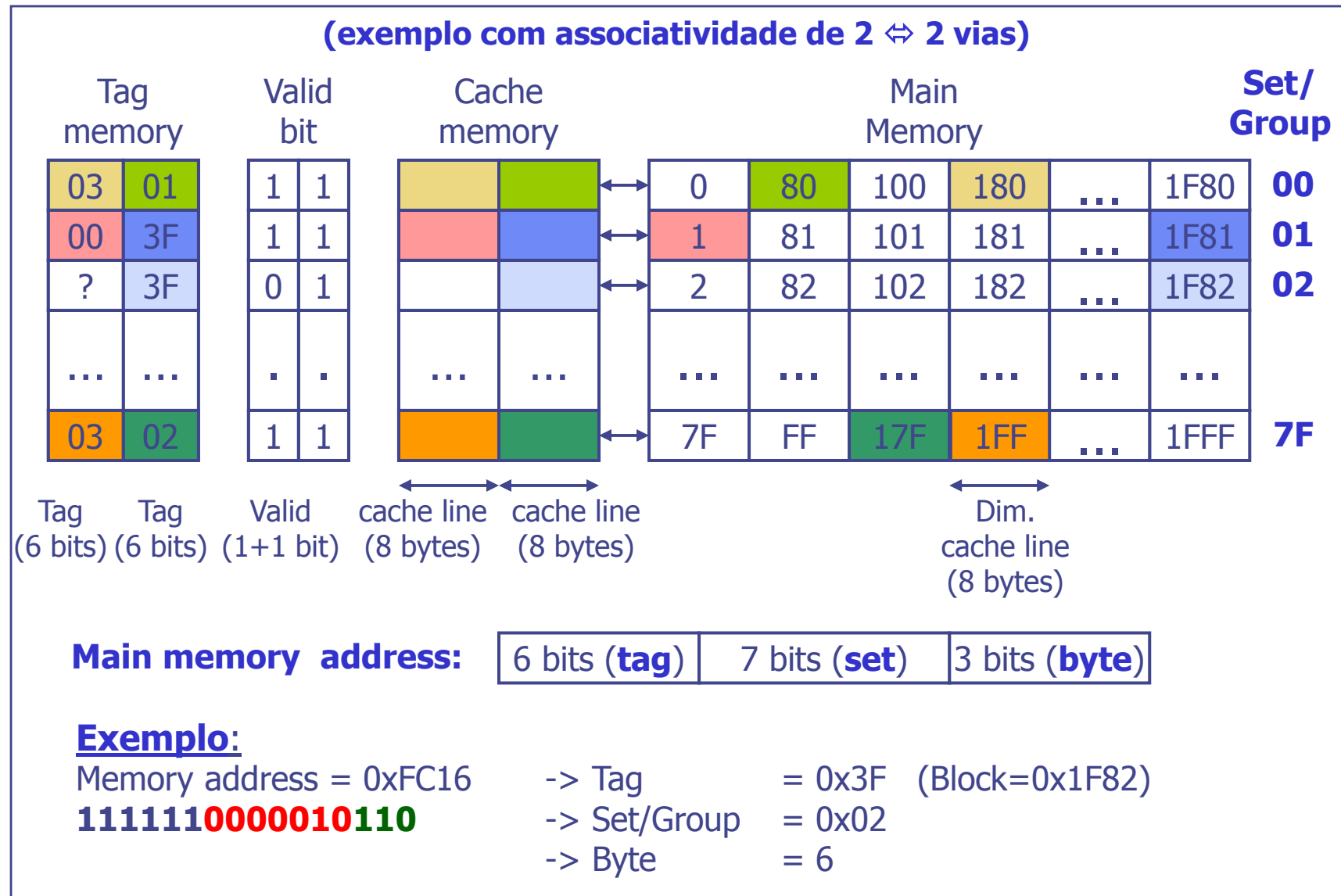
Cache com mapeamento direto



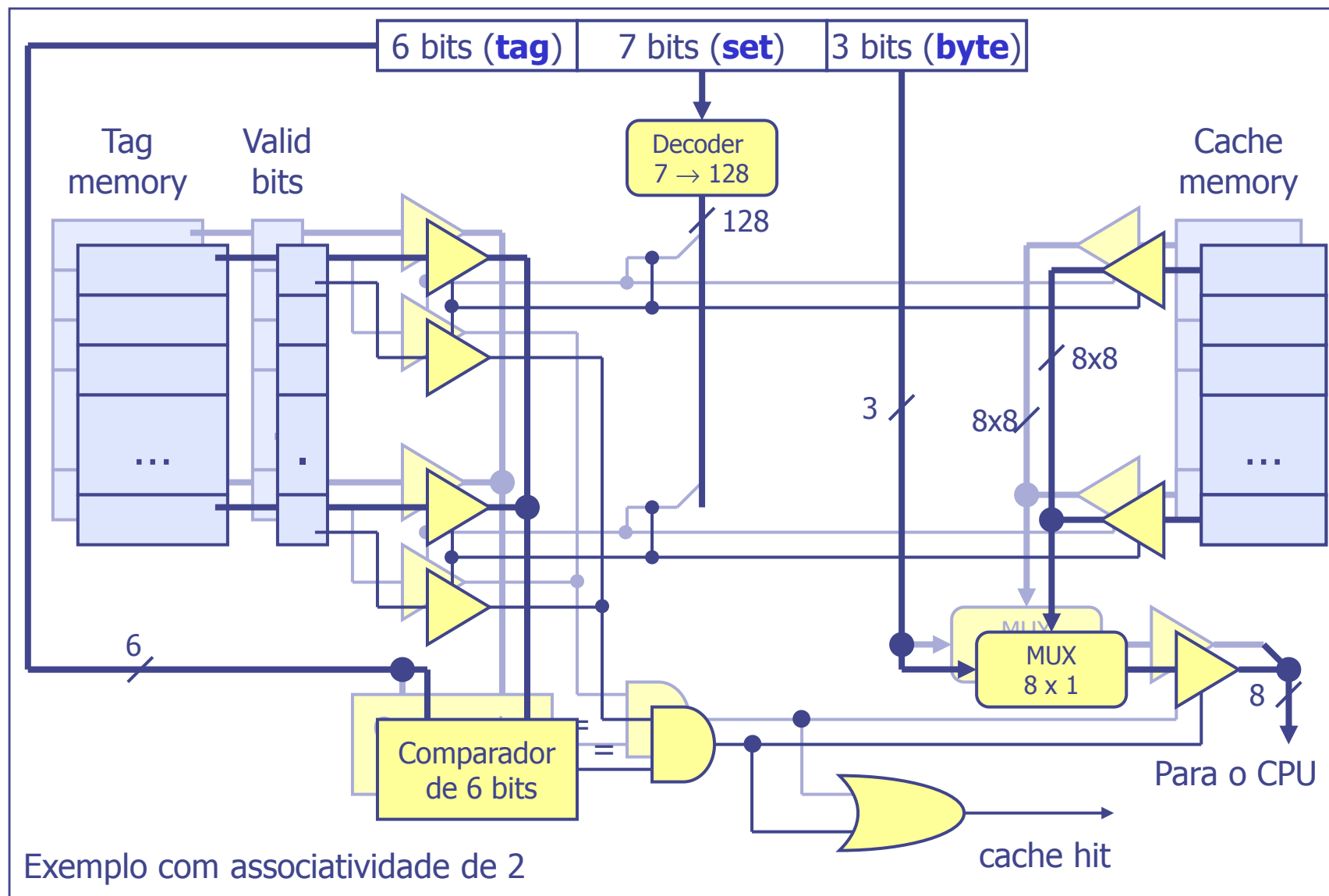
Cache com mapeamento direto

- A cada bloco da memória principal é associada uma linha da cache; o mesmo bloco é sempre colocado na mesma linha
- Maior vantagem:
 - Simplicidade de implementação
- Maior desvantagem:
 - Vários blocos têm associada a mesma linha
 - 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 ("set associative")**

Cache com mapeamento parcialmente associativo



Cache com mapeamento parcialmente associativo

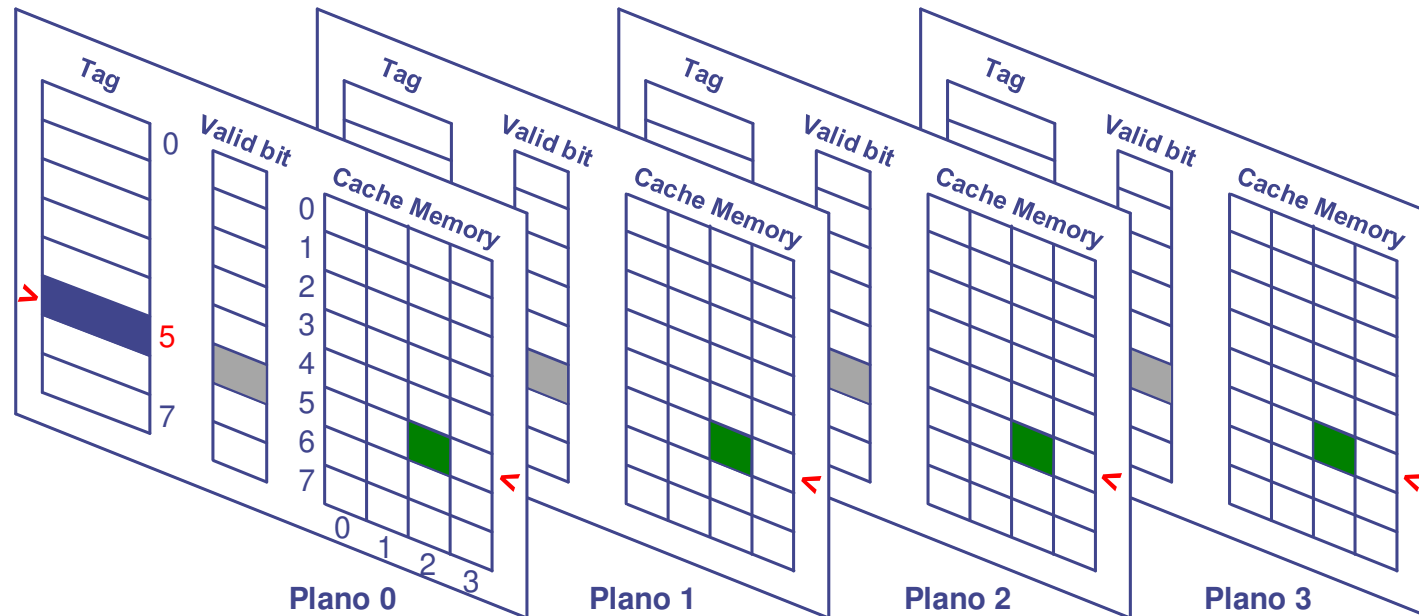


Cache parcialmente associativa

- A cache parcialmente associativa é semelhante à cache com mapeamento direto, mas a primeira permite que mais que um bloco de um mesmo grupo possa ser carregado na cache
- Uma cache com associatividade de 2, ou seja, com 2 vias, permite que 2 blocos do mesmo grupo possam estar simultaneamente na cache
- A divisão do endereço de memória é igual ao do mapeamento direto (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; "n" é o número de vias da cache e nesse caso diz-se que a cache tem associatividade de "n"
- As "n" vias onde o bloco pode residir têm que ser procuradas simultaneamente; o hit da cache resulta do OR dos hits de cada uma das vias
- (Block-set associative cache / n-way-set associative cache)

Cache com mapeamento parcialmente associativo

- Esquematização de uma cache com **associatividade de 4**, de **128 bytes**, com **blocos de 4 bytes** ($128 / 4 = 32$ bytes por via, $32 / 4 = 8$ linhas)
- Endereço gerado pelo CPU (16 bits) 0x6A96: **0110101010010110**



- A comparação dos 11 bits mais significativos (A15-A5) do endereço com as "tags" é feita simultaneamente nas 4 vias
- A posição da memória "tag" onde é feita a comparação é determinada pelos bits A4-A2 do endereço (posição 5 no exemplo)
- A via onde ocorre o hit (se ocorrer) fornece o byte armazenado na cache na posição determinada pelos bits A1-A0 do endereço (posição 2 no exemplo)

Políticas de substituição de blocos

- As políticas de substituição de blocos na cache, na ocorrência de um miss, são várias. As mais utilizadas são as seguintes:
- **FIFO** – First in first out: é substituído o bloco que foi carregado há mais tempo
- **LRU – Least Recently Used**: é substituído o bloco da cache que está há mais tempo sem ser referenciado
- **LFU – Least Frequently Used**: substituído o bloco menos acedido
- **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
- Se endereço ausente na cache, atualiza apenas a memória principal (**write-no-allocate**)
- A memória principal está sempre consistente

- **Write-back**

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

Exemplo – Intel Core i7-6500U

- Número de cores: 2
 - Core 1: L1 data (32 KB), L1 instr (32 KB), L2 data+instr (256 KB)
 - Core 2: L1 data (32 KB), L1 instr (32 KB), L2 data+instr (256 KB)
 - L3: (4 MB) partilhada pelos 2 cores

Cache	Size	Associativity	Line Size
L1 Data	2 x 32 KB	8-way set associative	64 bytes
L1 Instructions	2 x 32 KB	8-way set associative	64 bytes
L2	2 x 256 KB	4-way set associative	64 bytes
L3	4 MB (shared cache)	16-way set associative	64 bytes

- Qual a dimensão do bloco das caches L1, L2 e L3?
- Qual o número de linhas das caches L1, L2 e L3?

Exercícios

1. Qual a posição dos endereços de bloco 1 e 29 numa cache de mapeamento direto 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 direto; 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:
 - uma cache associativa;
 - uma cache com mapeamento direto;
 - uma cache com associatividade de 2.