

N_HARD COMPA6 - Texto de apoio

Site: [EAD Mackenzie](#)
Tema: HARDWARE PARA COMPUTAÇÃO {TURMA 01B} 2021/2
Livro: N_HARD COMPA6 - Texto de apoio

Impresso por: ANDRE SOUZA OCLECIANO .
Data: terça, 28 set 2021, 21:35

Descrição

Índice

1. UCP – COMPONENTES INTERNOS, CICLO DE INSTRUÇÃO E FUNCIONAMENTO

1. UCP – COMPONENTES INTERNOS, CICLO DE INSTRUÇÃO E FUNCIONAMENTO

COMPONENTE: Hardware para Computação

AULA: 6

MEMÓRIAS – MEMÓRIA CACHE

A memória cache é uma memória intermediária entre o processador e a MP com elevada velocidade de transferência.

E qual é o motivo de sua existência? Por que não deixar o processador “conversar” diretamente com a MP?

Existem dois motivos principais para a existência da memória cache:

- (1) Há uma diferença de velocidade muito grande entre o processador e a MP. A MP transfere bits para o processador em velocidades sempre inferiores às que o processador pode receber e operar os dados, o que pode deixar o processador ocioso e diminuir o desempenho de processamento.
- (2) Comportamento dos programas – princípios da localidade.

Características gerais

- É do tipo SRAM.
- Armazena uma cópia de partes da MP usadas com mais frequência ou mais recentemente pelo processador.
- Cache *hit* – acesso com acerto. A palavra requerida pelo processador está na cache.
- Cache *miss* – acesso com falha. A palavra requerida pelo processador não está na cache.

Importante!

Deseja-se sempre que o cache *hit* seja maior do que o cache *miss* para que o desempenho da máquina seja melhor.

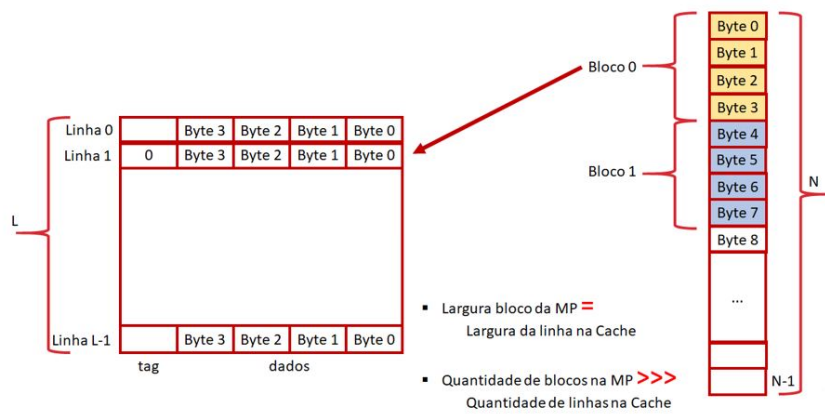
Tipos de cache

- Cache de Nível 1 (*Level 1* – L1) – está embutida no processador.
- Cache de Nível 2 (*Level 2* – L2) – localizada em um chip separado, acoplado ao processador.
- Cache de Nível 3 (*Level 3* – L3) – situada na placa-mãe.

Organização da cache

- Cache contém L linhas.
- Cada linha contém: uma *tag* (rótulo), um bloco de células da MP e bits de controle.
- A *tag* contém o endereço do bloco da MP que está armazenado na linha.
- A capacidade da cache é muito menor do que a capacidade da memória.

Figura 1 – Organização da memória cache



Fonte: Elaborada pela autora.

Funcionamento da cache

- (1) Processador inicia a operação de leitura e coloca o endereço desejado da MP no barramento de endereços.
- (2) Controlador de memória intercepta o endereço e interpreta seu conteúdo.
- (3) É verificado se o dado solicitado está ou não armazenado na cache.
- (4) Se o dado solicitado estiver na cache (cache *hit*), uma cópia do dado é transferida, pelo barramento de dados, da memória cache para o processador.
- (5) Se o dado solicitado não estiver na cache (cache *miss*), o controlador de memória busca pela informação na MP. Ao encontrar o dado, o bloco da MP que contém os bytes desejados é copiado e armazenado em uma linha da cache. Quando o processador solicitar novamente o dado ou algum dado vizinho, já está na cache, onde o acesso é rápido e não precisa buscar na MP (acesso lento).
- (6) Em seguida, apenas o dado solicitado é copiado para o processador.

Mapeamento de cache

Como o número de linhas na cache é menor do que a quantidade de blocos da MP, é necessário determinar em qual linha da cache um bloco específico da MP será inserido, ou seja, é preciso mapear os blocos da MP para as linhas da memória cache.

Existem três funções de mapeamento:

- Mapeamento direto
- Mapeamento associativo
- Mapeamento associativo por conjunto

MAPEAMENTO DIRETO

Cada bloco da MP tem uma linha da cache específica, previamente definido onde será armazenado.

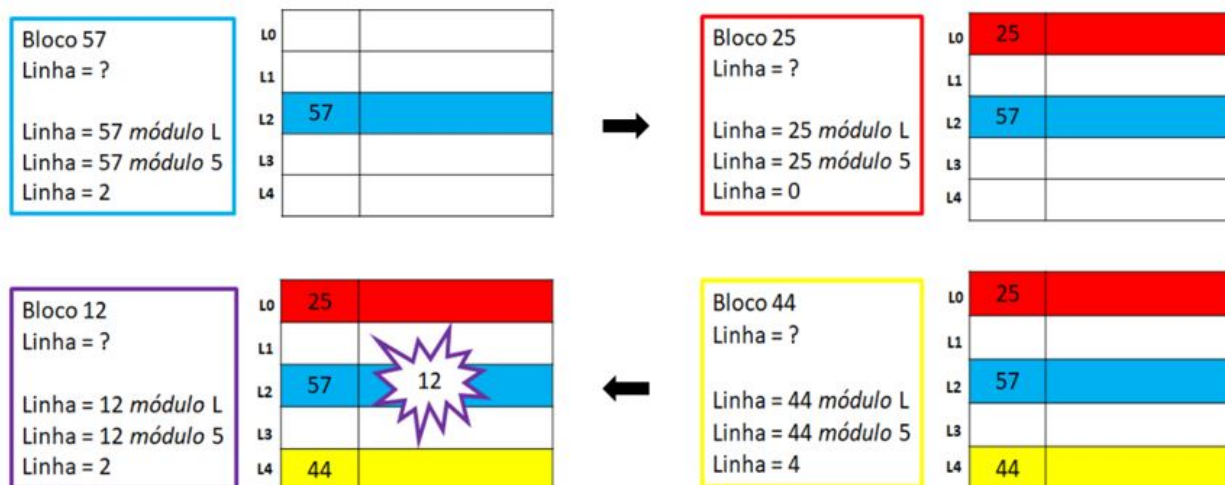
$$\text{Linha (cache)} = \text{Bloco (MP)} \bmod \text{Linhas (L)}$$

Vantagem: mapeamento simples. Cada bloco de memória é mapeado para uma posição fixa na cache.

Desvantagem: número elevado de colisão de informação. Se o programa faz referência repetidamente a palavras que estão em dois blocos de memória, mapeados à mesma posição na cache, então esses blocos serão continuamente introduzidos e retirados da cache.

Considere uma memória cache com L = 5 (total de linhas da cache). Deseja-se inserir a sequência de blocos da MP indicada: 57, 25, 44 e 12.

Figura 2 – Mapeamento direto para uma sequência de blocos



Fonte: Elaborada pela autora.

Ao tentar inserir o bloco 12, há colisão de linha. É necessário tirar o bloco 57 para poder inserir o bloco 12, mesmo havendo linhas livres na cache.

MAPEAMENTO ASSOCIATIVO

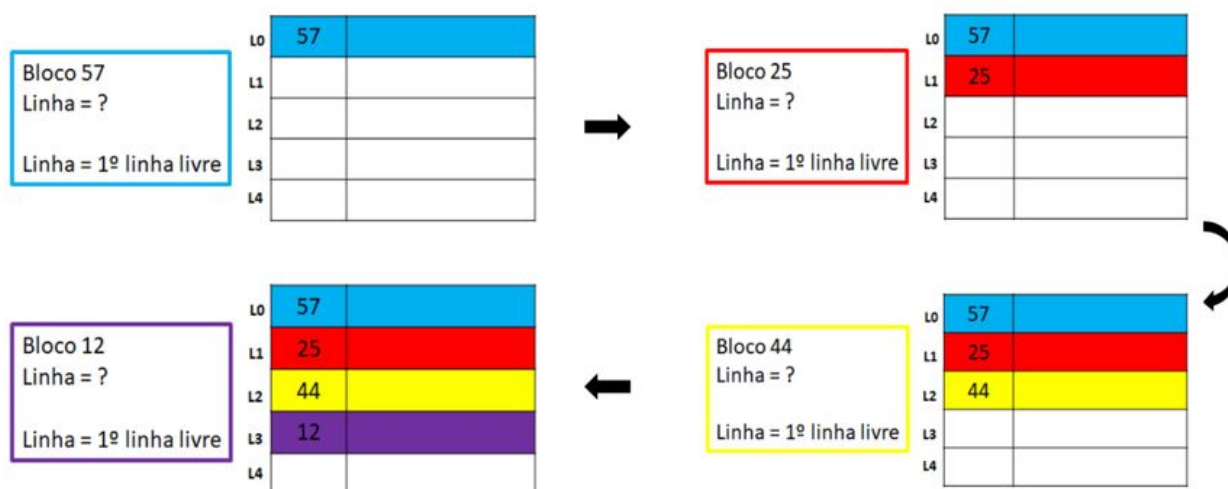
Não há local fixo na memória cache para alocação de um bloco da MP. Cada bloco pode ser armazenado em qualquer linha da cache.

Vantagem: número de colisão de informação é reduzido.

Desvantagem: a busca por um bloco na memória cache deve ser feita verificando linha a linha. Para otimizar o processo de busca, todas as linhas são examinadas simultaneamente. Essa busca simultânea é complexa e custosa.

Considere uma memória cache com $L = 5$ (total de linhas da cache). Deseja-se inserir a sequência de blocos da MP indicada: 57, 25, 44 e 12.

Figura 3 – Mapeamento associativo de uma sequência de blocos



Fonte: Elaborada pela autora.

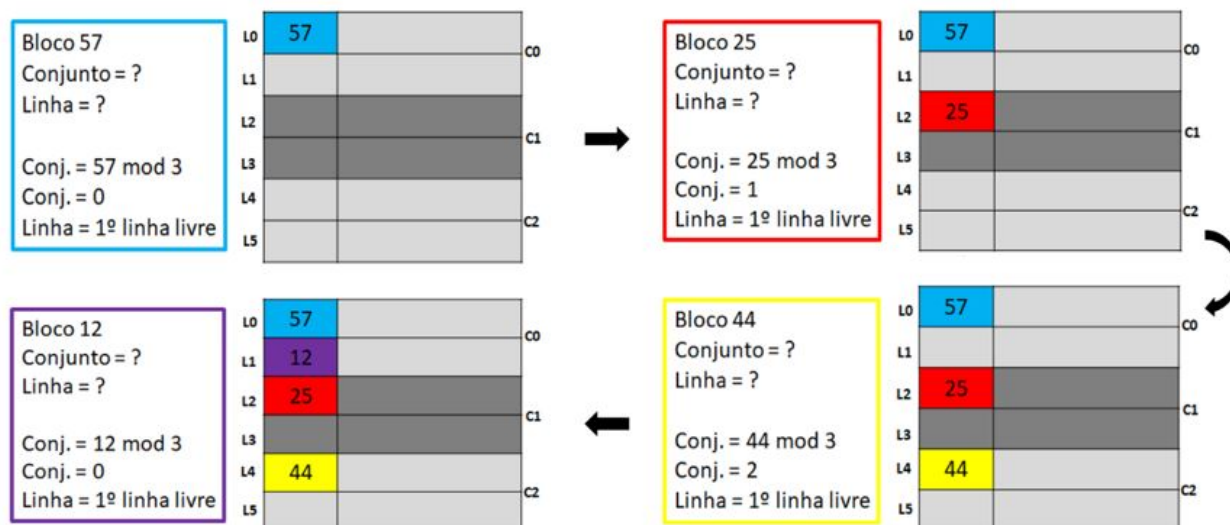
MAPEAMENTO ASSOCIATIVO POR CONJUNTO

Tenta-se resolver o problema de conflito de blocos em uma mesma linha (mapeamento direto), e o problema do mapeamento associativo relativo à custosa busca e comparação simultânea do campo rótulo em toda memória cache.

A memória cache é dividida em conjuntos, e cada conjunto possui uma certa quantidade de linhas. Cada conjunto é tratado pelo controlador de memória como no mapeamento direto, ou seja, cada bloco da MP ocupará um conjunto específico determinado pela operação **conjunto = bloco módulo C** (onde C = quantidade de conjuntos na memória cache). Dentro do conjunto, o bloco da MP pode ser armazenado em qualquer uma das linhas (método associativo).

Considere uma memória cache com L = 6 (total de linhas da cache). Deseja-se inserir a sequência de blocos da MP indicada: 57, 25, 44 e 12.

Figura 4 – Mapeamento associativo por conjunto de uma sequência de blocos



Fonte: Elaborada pela autora.

ALGORITMOS DE SUBSTITUIÇÃO

Quando um dos blocos da MP precisa ser armazenado na memória cache e ela está cheia, uma das linhas da cache deve ser substituída. A linha que receberá o novo bloco da MP é determinada por um algoritmo de substituição.

Os algoritmos de substituição são aplicados somente quando os mapeamentos associativo e associativo por conjunto são utilizados, pois, nestes mapeamentos, há uma flexibilidade de alocação de blocos de MP em linhas da cache.

Algoritmos de substituição existentes: LRU, LFU, FIFO e aleatório.

LRU (*Least Recently Used*)

- A linha a ser substituída é a que está há mais tempo sem ser referenciada na memória cache (Figura 5).

LFU (*Least Frequently Used*)

- A linha a ser substituída é a que foi referenciada menos vezes pelo processador (Figura 5).

FIFO (*First In First Out*)

- A linha a ser substituída é a que está há mais tempo na cache, independente de estar sendo usada ou não com frequência pelo processador (Figura 5).

ALEATÓRIO

- A substituição é realizada aleatoriamente e não é baseada em nenhum histórico de uso.

Figura 5 – Algoritmos de substituição

Linha	Tag	Dados	FIFO	LFU	LRU
			Tempo total na memória (ns)	Frequência de acessos	Último acesso (há ns atrás)
0			300	15	200
1			10	94	1
2			1500	36	150
3			187	82	58
4			690	07	47
5			268	24	239
6			792	61	12
7			43	43	73
8			90	02	05
9			814	70	63
10			225	13	172

Fonte: Elaborada pela autora.

Como mencionado no item Organização da Cache, cada linha da memória cache é composta por uma tag, o bloco de informação e bits de controle. Dependendo do algoritmo de substituição utilizado, os bits de controle armazenarão a informação necessária. Por exemplo, quando o algoritmo FIFO é utilizado pela memória cache, os bits de controle armazenam o tempo total que aquele bloco está armazenado naquela linha específica da cache.

POLÍTICAS DE ATUALIZAÇÃO

Toda vez que o processador realiza uma operação de escrita, esta ocorre imediatamente na cache. É necessário que, em algum momento, a MP seja atualizada, para que o sistema mantenha sua coerência de informação.

Uma política de atualização deve ser utilizada para manter a coerência de cache, que é a coesão de informação entre a MP e a cache. E por que essa atualização é tão importante? A MP pode ser acessada tanto pela cache quanto por dispositivos de E/S (entrada e saída). É possível que uma célula da MP tenha sido alterada na cache e ainda não tenha sido alterada na MP e, assim, esta célula na MP está desatualizada. Ou um dispositivo de E/S pode alterar o conteúdo de uma célula na MP e, então, a cache é que fica desatualizada.

As políticas de atualização são:

- *Write through*
- *Write back*
- *Write once*
- Protocolo MESI

Vamos entender melhor como funcionam as políticas *write through* e *write back*.

-

WRITE THROUGH (ou escrita direta ou escrita em ambas)

Todas as operações de escrita são feitas tanto na memória cache quanto na MP. Dessa forma, consegue-se assegurar que a MP estará sempre com dados válidos.

Desvantagem: considerável tráfego de informação com a MP é gerado, podendo criar um gargalo no sistema, pois, conforme já vimos anteriormente, a MP é considerada uma memória lenta. Pode haver uma grande quantidade de escritas desnecessárias na MP e, consequentemente, uma redução de desempenho no sistema.

WRITE BACK (ou escrita de volta ou escrita somente no retorno)

As escritas do processador acontecem apenas na memória cache. Quando é feita uma atualização do processador, em uma linha da cache, é atribuído o valor 1 ao bit de atualização (um dos bits de controle na linha da cache) associado à linha atualizada na cache. Quando um bloco vai ser substituído na cache, ele apenas é escrito de volta na MP se seu bit de atualização tiver valor 1.

Desta forma, minimiza-se o número de operações de escrita na MP (que era um gargalo na política *write through*).

Desvantagem: partes da MP podem ficar inválidas por um tempo e, portanto, o acesso à MP pelos dispositivos de E/S só poderá ser efetuado por meio da memória cache. O acesso direto à cache requer um conjunto de circuitos mais complexo e custoso.

WRITE ONCE

É uma política utilizada em sistemas de multiprocessadores, que não são o foco deste material.

PROTOCOLO MESI

Política de atualização amplamente utilizada. Não estudaremos os detalhes, pois requer conhecimento de máquina de estados, mas é importante que saibam que existe e que é utilizado.

CÓDIGO HAMMING

Nenhum canal de comunicação ou meio de armazenamento pode ser completamente livre de erros. Esses erros podem ser desastrosos, como um erro de um bit no envio de informação ou código de um programa.

Quanto mais bits são compactados por milímetro quadrado de armazenamento, maiores as densidades de fluxo magnético; as taxas de erro também aumentam na porção direta do número de bits/segundo transmitidos ou do número de bits por milímetro quadrado de armazenamento magnético.

Em meios de transmissão, tal como redes de computadores, é suficiente a habilidade de detectar erros. Quando um erro é detectado, solicita-se a retransmissão da informação, se necessário. Em dispositivos de memória, é preciso ter a habilidade de detectar erros e de corrigi-los.

Deteção e correção de erros na transmissão e no armazenamento de informação dentro da máquina é um estudo a ser considerado no projeto de sistemas de computação. Utiliza-se, hoje, como algoritmo de detecção e correção de erros, o Cycle Redundancy Code (CRC), ou, em português, código por redundância cíclica.

Em nossa disciplina, precisamos apenas ter uma ideia de como um código de detecção e correção de erros funciona. Para isso, adotamos um algoritmo simples: o **Código Hamming**. Este algoritmo não é utilizado nas máquinas atuais porque apresenta algumas falhas e detecta **apenas um erro** na informação transmitida.

Para uma palavra original de M bits que queremos armazenar na memória, adiciona-se R bits, denominados de bits de redundância, formando um código de M + R bits. Esse código é, então, armazenado na memória (alguns autores denominam de palavra final).

Como calcular o valor de R? R deve ser tal que $2^R - 1 - R \geq M$.

Exemplo: para M = 4 e R = 3, temos $2^3 - 1 - 3 = \underline{\underline{4}} \geq 4$

Para M = 8 e R = 4, temos $2^4 - 1 - 4 = \underline{\underline{11}} \geq 8$

(1) Sabendo a quantidade de bit de redundância que deve ser usada, é necessário calcular cada um desses bits. O cálculo é feito utilizando uma função que envolve os M bits da palavra original. Veja, no exemplo, o passo a passo dessa função.

(2) A palavra original mais os R bits de redundância são armazenados na memória. Cada bit tem uma posição pré-definida que deve ocupar no código final gerado.

(3) Após a leitura do código da memória (M + R bits), os bits de redundância são novamente calculados de acordo com os M bits recém lidos. O cálculo é feito utilizando a mesma função da etapa (1).

(4) Os novos bits de redundância, recém calculados, são comparados com os R bits de redundância lidos da memória.

(5) Se a comparação for igual, consideramos o código lido correto, ou seja, nenhum erro foi detectado.

Se a comparação não for igual, consideramos que houve um erro no armazenamento e o erro é corrigido utilizando a operação de XOR entre os valores das redundâncias armazenadas e os novos valores calculados.

Importante!

O código de Hamming é um código de correção que detecta e corrige apenas 1 bit errado.

Exemplo:

Palavra original - 01001011₂

Código final?

posição 8 posição 1

M = 8 bits

R = 4 bits → 4 bits de redundância

12 bits

Posição na palavra	Binário correspondente	Informação
12	1100	Dado 8
11	1011	Dado 7
10	1010	Dado 6
09	1001	Dado 5
08	1000	Redundância 4
07	0111	Dado 4
06	0110	Dado 3
05	0101	Dado 2
04	0100	Redundância 3
03	0011	Dado 1
02	0010	Redundância 2
01	0001	Redundância 1

Redundância1: $D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7$

$$1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$$

Redundância2: $D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7$

$$1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = 1$$

Redundância3: $D2 \oplus D3 \oplus D4 \oplus D8$

$$1 \oplus 0 \oplus 1 \oplus 0 = 0$$

Redundância4: $D5 \oplus D6 \oplus D7 \oplus D8$

$$0 \oplus 0 \oplus 1 \oplus 0 = 1$$

Código final a ser enviado:

D8	D7	D6	D5	R4	D4	D3	D2	R3	D1	R2	R1
0	1	0	0	1	1	0	1	0	1	1	0

Considerando o código lido:

D8	D7	D6	D5	R4	D4	D3	D2	R3	D1	R2	R1
0	1	0	0	1	0	0	1	0	1	1	0

Como descobrir se aconteceu ou não um erro na transmissão ou no armazenamento?

É necessário recalcular as redundâncias a partir do código lido e fazer o XOR com a redundância do código armazenado.

1ª etapa: recalcular as redundâncias.

Redundância1: $D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7$

$$1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 1$$

Redundância2: $D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7$

$$1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0$$

Redundância3: $D2 \oplus D3 \oplus D4 \oplus D8$

$$1 \oplus 0 \oplus 0 \oplus 0 = 1$$


Redundância4: $D5 \oplus D6 \oplus D7 \oplus D8$

$$0 \oplus 0 \oplus 1 \oplus 0 = 1$$

2ª etapa: fazer o XOR entre as novas redundâncias e as redundâncias armazenadas.

	R4	R3	R2	R1
Nova redundância	1	1	0	1
Redundância armazenada	1	0	1	0
	0	1	1	1

\oplus



Posição 0111 = Posição 07 → onde está armazenado o bit D4 → justamente o bit que estava errado (comparado ao código armazenado)

Descoberto o bit incorreto, é possível convertê-lo e processar a palavra corretamente.