

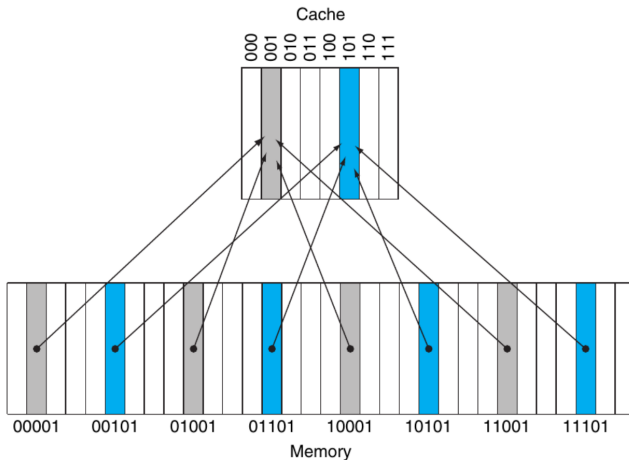
Blocos da Cache e Associatividade

Yuri Kaszubowski Lopes

UDESC

Cache com blocos de uma palavra

- Montamos uma cache com mapeamento direto
- Cada posição (bloco) da cache, armazena exatamente uma palavra



Cache com blocos de uma palavra

Localidade Temporal

- Nossa cache (com blocos de uma palavra) se beneficia disso
- Carregamos o dado para a cache, e se no futuro (próximo) ele for necessário novamente, ele já está na cache
 - ▶ Desde que ninguém o tire de lá

Localidade Espacial

- Ao carregarmos o dado em um endereço, é provável que seus vizinhos também sejam úteis
 - ▶ Isso não é explorado na cache com mapeamento de palavras
 - ▶ Como podemos tirar vantagem da localidade espacial?
- Única vantagem de localidade espacial é que vizinhos não concorrem pela mesma posição na cache

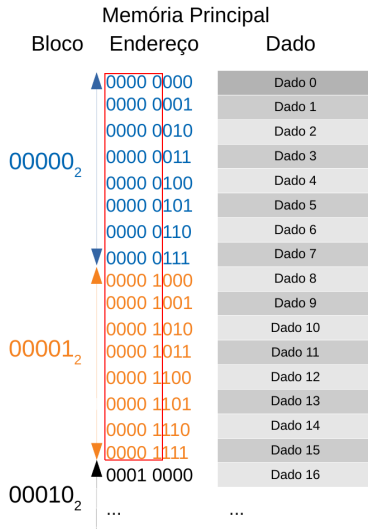
Mapeamento por blocos

- Vamos dividir a memória em blocos de n bytes
- Essa divisão vai ser utilizada para realizar o mapeamento da cache
- Exemplo considerando que cada endereço da memória suporta 1 byte, e cada bloco possui 8 bytes
- **Como obter o endereço de bloco?**

| Memória Principal | | |
|--------------------|-------------|---------|
| Bloco | Endereço | Dado |
| 00000 ₂ | ▲ 0000 0000 | Dado 0 |
| | 0000 0001 | Dado 1 |
| | 0000 0010 | Dado 2 |
| | 0000 0011 | Dado 3 |
| | 0000 0100 | Dado 4 |
| | 0000 0101 | Dado 5 |
| | 0000 0110 | Dado 6 |
| | ▼ 0000 0111 | Dado 7 |
| 00001 ₂ | ▲ 0000 1000 | Dado 8 |
| | 0000 1001 | Dado 9 |
| | 0000 1010 | Dado 10 |
| | 0000 1011 | Dado 11 |
| | 0000 1100 | Dado 12 |
| | 0000 1101 | Dado 13 |
| | 0000 1110 | Dado 14 |
| | ▼ 0000 1111 | Dado 15 |
| 00010 ₂ | ▲ 0001 0000 | Dado 16 |
| | ... | ... |

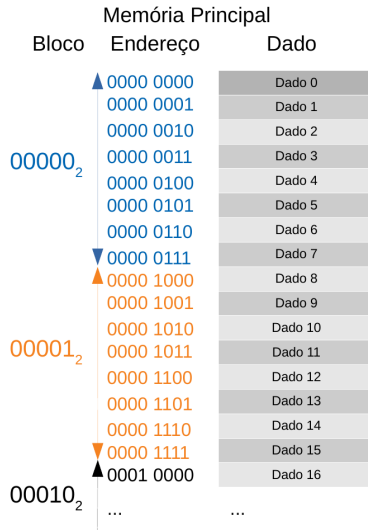
Mapeamento por blocos

- Exemplo considerando que cada endereço da memória suporta 1 byte, e cada bloco possui 8 bytes
- Como obter o endereço de bloco?**
 - $\text{Bloco} = \frac{\text{Endereço}}{\text{Tamanho do Bloco}}$
 - Divisão inteira
 - Em binário é ainda mais fácil se tudo for múltiplo de 2
 - ★ Basta usarmos os bits certos do endereço



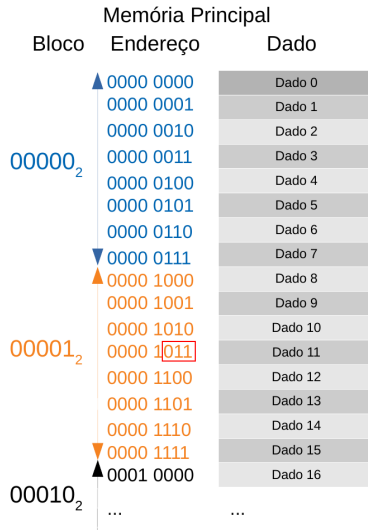
Mapeamento por blocos

- Quando o processador solicita o dado em determinado endereço
- Verifica o endereço do bloco
 - ▶ Utiliza os **bits mais baixos do bloco** para procurar na cache
 - ▶ Os bits mais altos **do bloco** são comparados com o tag



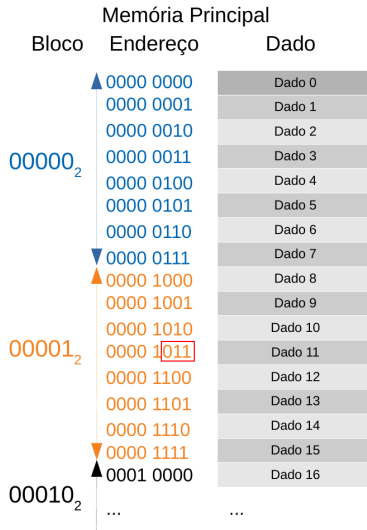
Mapeamento por blocos

- Em caso de hit:
 - ▶ O bloco está na cache
 - ▶ Mas o processador não solicitou um bloco, mas sim o dado em um endereço de memória específico



Mapeamento por blocos

- Em caso de hit:
 - ▶ O bloco está na cache
 - ▶ Mas o processador não solicitou um bloco, mas sim o dado em um endereço de memória específico
 - ▶ Os bits que foram descartados para se obter o endereço do bloco, podem ser usados para se obter um “deslocamento dentro do bloco”
 - ★ *Offset*



Mapeamento por blocos

- Em caso de hit:
 - ▶ O bloco está na cache
 - ▶ Mas o processador não solicitou um bloco, mas sim o dado em um endereço de memória específico
 - ▶ Os bits que foram descartados para se obter o endereço do bloco, podem ser usados para se obter um “deslocamento dentro do bloco”

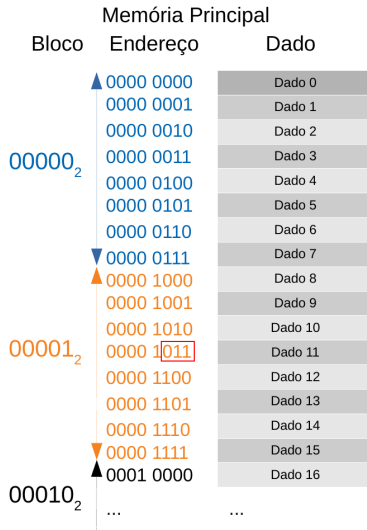
★ *Offset*

- Se a CPU solicita 0000 1011
 - ▶ Está no bloco 00001₂
 - ▶ Deslocado 011₂ dentro desse bloco

| Memória Principal | | |
|--------------------|-----------|---------|
| Bloco | Endereço | Dado |
| 00000 ₂ | 0000 0000 | Dado 0 |
| | 0000 0001 | Dado 1 |
| | 0000 0010 | Dado 2 |
| | 0000 0011 | Dado 3 |
| | 0000 0100 | Dado 4 |
| | 0000 0101 | Dado 5 |
| | 0000 0110 | Dado 6 |
| 00001 ₂ | 0000 0111 | Dado 7 |
| | 0000 1000 | Dado 8 |
| | 0000 1001 | Dado 9 |
| | 0000 1010 | Dado 10 |
| | 0000 1011 | Dado 11 |
| | 0000 1100 | Dado 12 |
| | 0000 1101 | Dado 13 |
| 00010 ₂ | 0000 1110 | Dado 14 |
| | 0000 1111 | Dado 15 |
| | 0001 0000 | Dado 16 |
| | ... | ... |

Mapeamento por blocos

- Em caso de miss:
 - ▶ Buscamos todo o bloco da memória e carregamos para a cache



Vantagens e desvantagens

Vantagens

Desvantagens

Vantagens e desvantagens

Vantagens

- Aumentamos a **localidade espacial**
 - ▶ Carregamos o dado e seus vizinhos em caso de miss

Desvantagens

Vantagens e desvantagens

Vantagens

- Aumentamos a **localidade espacial**
 - ▶ Carregamos o dado e seus vizinhos em caso de miss

Desvantagens

- Aumentamos a competição na cache
 - ▶ O tamanho total da cache não muda
 - ★ O pior caso seria uma cache de um bloco somente
 - ★ Se precisarmos de qualquer dado que esteja fora desse bloco, precisamos jogar toda a cache fora, para carregar um bloco completo que está em outro lugar
 - ★ Diminui a localidade temporal
 - ★ É mais provável que um dado que foi utilizado no passado, mas que está “longe” dos últimos carregados, seja substituído

Vantagens e desvantagens

Vantagens

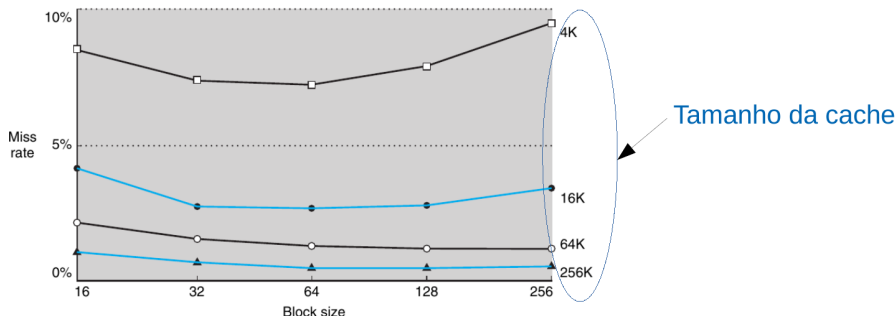
- Aumentamos a **localidade espacial**
 - ▶ Carregamos o dado e seus vizinhos em caso de miss

Desvantagens

- Aumentamos a competição na cache
 - ▶ O tamanho total da cache não muda
 - ★ O pior caso seria uma cache de um bloco somente
 - ★ Se precisarmos de qualquer dado que esteja fora desse bloco, precisamos jogar toda a cache fora, para carregar um bloco completo que está em outro lugar
 - ★ Diminui a localidade temporal
 - ★ É mais provável que um dado que foi utilizado no passado, mas que está “longe” dos últimos carregados, seja substituído
- A penalidade de falta (miss penalty) se torna maior
 - ▶ Precisamos carregar mais dados da memória principal

Vantagens e desvantagens

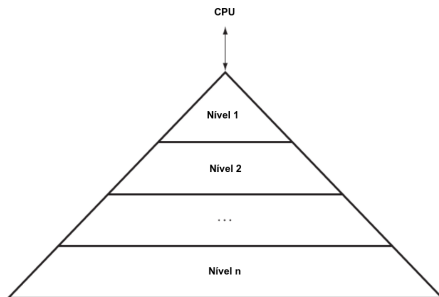
- Precisamos de um equilíbrio
- Blocos muito pequenos diminuem a **localidade espacial**
- Blocos muito grandes diminuem a **localidade temporal**



Taxa de miss para diferentes tamanhos de bloco e diferentes tamanhos de cache no SPEC92 (Patterson, Henessy; 2014)

Leituras versus escritas

- Ler um dado com a memória cache é relativamente simples
 - ▶ Em caso de hit, lemos o dado
 - ▶ Em caso de miss
 - ★ Carregamos o dado para a cache (Enquanto isso o pipeline pode entrar em **stall**)
 - ★ Depois lemos o dado
- Mas em caso de escritas, as coisas não são tão simples
- `sw $t0, 4($t1)`
- Qual a dificuldade considerando os múltiplos níveis de memória?



Tratando escritas

- Quando uma instrução escrever algo, vamos escrever apenas na cache
 - ▶ Lembre-se que numa hierarquia real, a CPU se comunica apenas com o nível de memória mais alto
- Agora para o mesmo endereço de memória, temos dois dados diferentes
 - ▶ Um na cache (atualizado)
 - ▶ Um na memória de nível mais baixo (desatualizado)
 - ▶ **A memória fica inconsistente**

Tratando escritas

Write-Through

- Maneira simples de corrigir:
 - ▶ Sempre propagar escritas para os níveis mais baixos de memória
 - ▶ Esquema chamado de **Write-Through**
 - ▶ **Problemas?**

Tratando escritas

Write-Through

- Maneira simples de corrigir:
 - ▶ Sempre propagar escritas para os níveis mais baixos de memória
 - ▶ Esquema chamado de **Write-Through**
 - ▶ **Problemas?**
 - ★ Nossa cache serve apenas para leituras
 - ★ Toda escrita deve ser propagada para os níveis mais lentos
 - ★ Precisamos esperar os níveis mais lentos terminarem a operação
 - ★ O mesmo (ou pior) que não ter uma cache
 - ▶ Como melhorar?

Tratando escritas

Write-Back

- Escrevemos apenas na cache
- Esquema chamado de **Write-Back**
- O dado é atualizado nos níveis mais baixos apenas quando o dado na cache é substituído
- Utilizado na maioria das CPUs atuais
- Em alguns cenários write-throughs podem ser mais eficientes.
- Existem ainda outros métodos, como o no write allocate que escreve na memória, mas não na cache
 - ▶ A CPU comumente utiliza write-back, mas deixa o Sistema Operacional modificar o sistema de escrita de setores da memória específicos quando conveniente

Tratando escritas

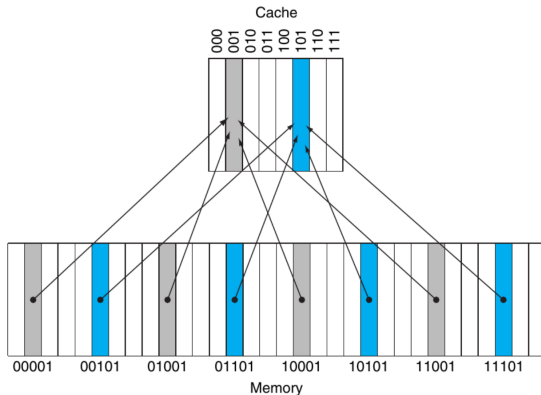
Write-Back

Write-Back

- Mais complexo de tratar
 - ▶ Se torna um pesadelo especialmente considerando nossas máquinas com múltiplas CPUs
 - ▶ Cada CPU tem uma cópia do dado em sua própria cache (e.g. L1)
 - ★ Se outra CPU requisita o mesmo dado, e simplesmente o carregar da memória, estamos carregando uma versão desatualizada!

Associatividade da cache

- Na cache de diretamente mapeada, cada bloco da memória pode ser “encaixado” em apenas uma posição na cache
 - ▶ Se essa posição já estiver ocupada, precisamos substituir seu conteúdo
 - ★ **Mesmo que haja várias outras posições livres na cache que poderíamos utilizar**
 - ★ Colisão de endereço
- Solução: associatividade da cache



Cache **totalmente associativa**

- Uma cache **totalmente associativa** pode carregar um bloco da memória para **qualquer** bloco da cache
 - ▶ Qualquer bloco livre
 - ▶ Se a cache estiver totalmente cheia, podemos escolher o bloco “menos útil” para substituir
- Qual as vantagens e desvantagens de uma cache totalmente associativa quando comparada a uma diretamente mapeada?

Cache **totalmente associativa**

- Uma cache **totalmente associativa** pode carregar um bloco da memória para **qualquer** bloco da cache
 - ▶ Qualquer bloco livre
 - ▶ Se a cache estiver totalmente cheia, podemos escolher o bloco “menos útil” para substituir
- Qual as vantagens e desvantagens de uma cache totalmente associativa quando comparada a uma diretamente mapeada?
 - ▶ + Redução de misses
 - ★ Maior flexibilidade, podendo escolher os blocos menos úteis para substituir na cache
 - ▶ - Quando a CPU solicita um endereço, precisamos procurar na cache toda
 - ★ O endereço não vai para um bloco específico da cache
 - ★ Uma abordagem comum é colocar comparadores paralelos no hardware
 - ★ Custo de energia, espaço e complexidade
 - ★ Ainda sim perdemos um pouco de tempo

Cache associativa por conjunto

- **Cache associativa por conjunto:**
 - ▶ Meio termo entre um modelo totalmente associativo e o diretamente mapeado
- Cache separada em **conjuntos**
 - ▶ Cada conjunto suporta até **n blocos**
 - ▶ Cache **associativa de n vias**
- Os blocos da memória são mapeados para os conjuntos
 - ▶ O bloco da memória pode estar dentro de **qualquer bloco do conjunto**
- Quando a CPU precisa de um dado no endereço X
 - ▶ O conjunto que precisa ser pesquisado é fixo
 - ▶ Agora é necessário pesquisar só em **todos os blocos do conjunto**

Exemplos

- Associatividades possíveis em uma cache de 8 blocos

Associativa de 1 via
(diretamente mapeada)

| Conjunto | Tag | Dado |
|----------|-----|------|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |

Associativa de 2 vias

| Conjunto | Tag | Dado | Tag | Dado |
|----------|-----|------|-----|------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

Associativa de 4 vias

| Conjunto | Tag | Dado | Tag | Dado | Tag | Dado | Tag | Dado |
|----------|-----|------|-----|------|-----|------|-----|------|
| 0 | | | | | | | | |
| 1 | | | | | | | | |

Associativa de 8 vias (totalmente associativa)

| Conjunto | Tag | Dado | Tag | Dado | Tag | Dado | Tag | Dado | Tag | Dado | Tag | Dado | Tag | Dado | Tag | Dado |
|----------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|
| 0 | | | | | | | | | | | | | | | | |

Exemplo

- Blocos de 4 bytes
- Cache associativa de 2 vias
- Cache com capacidade para armazenar 8 blocos no total
 - ▶ $8 \times 4 = 32$ bytes de capacidade para dados
- Onde o byte no endereço 00001001_2 pode ser mapeado

Exemplo

Cache

| Conjunto | Tag | Dado (Bloco) | Tag | Dado (Bloco) |
|----------|-----|--------------|-----|--------------|
| 00_2 | | | | |
| 01_2 | | | | |
| 10_2 | | | | |
| 11_2 | | | | |

- Onde o byte no endereço 00001001_2 pode ser mapeado
- Bloco de 4 bytes: 2 bits menos significativos são offset

Memória Principal

Endereço Dado(1 byte)

| | |
|-----------|---------|
| 0000 0000 | Dado 0 |
| 0000 0001 | Dado 1 |
| 0000 0010 | Dado 2 |
| 0000 0011 | Dado 3 |
| 0000 0100 | Dado 4 |
| 0000 0101 | Dado 5 |
| 0000 0110 | Dado 6 |
| 0000 0111 | Dado 7 |
| 0000 1000 | Dado 8 |
| 0000 1001 | Dado 9 |
| 0000 1010 | Dado 10 |
| 0000 1011 | Dado 11 |
| 0000 1100 | Dado 12 |
| 0000 1101 | Dado 13 |
| 0000 1110 | Dado 14 |
| 0000 1111 | Dado 15 |
| 0001 0000 | Dado 16 |

...

...

Exemplo

Cache

| Conjunto | Tag | Dado (Bloco) | Tag | Dado (Bloco) |
|----------|-----|--------------|-----|--------------|
| 00_2 | | | | |
| 01_2 | | | | |
| 10_2 | | | | |
| 11_2 | | | | |

- Onde o byte no endereço 00001001_2 pode ser mapeado
- 8 blocos e 2 vias: 4 conjuntos: 2 bits para endereço do conjunto na cache

Memória Principal

| Bloco | Endereço | Dado(1 byte) |
|------------|-------------|--------------|
| 000000_2 | ▲ 0000 0000 | Dado 0 |
| | 0000 0001 | Dado 1 |
| | 0000 0010 | Dado 2 |
| | ▼ 0000 0011 | Dado 3 |
| 000001_2 | ▲ 0000 0100 | Dado 4 |
| | 0000 0101 | Dado 5 |
| | 0000 0110 | Dado 6 |
| | ▼ 0000 0111 | Dado 7 |
| 000010_2 | ▲ 0000 1000 | Dado 8 |
| | 0000 1001 | Dado 9 |
| | 0000 1010 | Dado 10 |
| | ▼ 0000 1011 | Dado 11 |
| 000011_2 | ▲ 0000 1100 | Dado 12 |
| | 0000 1101 | Dado 13 |
| | 0000 1110 | Dado 14 |
| | ▼ 0000 1111 | Dado 15 |
| 000100_2 | ▲ 0001 0000 | Dado 16 |
| | ... | ... |

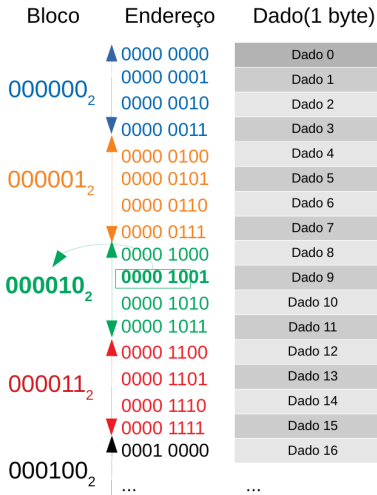
Exemplo

Cache

| Conjunto | Tag | Dado (Bloco) | Tag | Dado (Bloco) |
|----------|-----|--------------|-----|--------------|
| 00_2 | | | | |
| 01_2 | | | | |
| 10_2 | | | | |
| 11_2 | | | | |

- Onde o byte no endereço 00001001_2 pode ser mapeado
 - ▶ Está no bloco 000010_2
- 2 bits de offset
- 6 bits de endereço do bloco
 - ▶ 2 bits mapeamento para endereço do conjunto
 - ▶ 4 bits para Tag

Memória Principal



Exemplo

Cache

| Conjunto | Tag | Dado (Bloco) | Tag | Dado (Bloco) |
|----------|-----|--------------|-----|--------------|
| 00_2 | | | | |
| 01_2 | | | | |
| 10_2 | | | | |
| 11_2 | | | | |

- Onde o byte no endereço 00001001_2 pode ser mapeado
- Temos 4 conjuntos. Como precisamos de 2 bits para endereçar os conjuntos, “olhamos” para os 2 bits menos significativos do bloco
 - ▶ Está no conjunto 10_2

Memória Principal

| Bloco | Endereço | Dado(1 byte) |
|------------|-------------|--------------|
| 000000_2 | ▲ 0000 0000 | Dado 0 |
| | 0000 0001 | Dado 1 |
| | 0000 0010 | Dado 2 |
| | ▼ 0000 0011 | Dado 3 |
| 000001_2 | ▲ 0000 0100 | Dado 4 |
| | 0000 0101 | Dado 5 |
| | 0000 0110 | Dado 6 |
| | ▼ 0000 0111 | Dado 7 |
| 000010_2 | ▲ 0000 1000 | Dado 8 |
| | 0000 1001 | Dado 9 |
| | 0000 1010 | Dado 10 |
| | ▼ 0000 1011 | Dado 11 |
| 000011_2 | ▲ 0000 1100 | Dado 12 |
| | 0000 1101 | Dado 13 |
| | 0000 1110 | Dado 14 |
| | ▼ 0000 1111 | Dado 15 |
| 000100_2 | ▲ 0001 0000 | Dado 16 |
| ... | ... | ... |

Exemplo

Cache

| Conjunto | Tag | Dado (Bloco) | Tag | Dado (Bloco) |
|----------|-----|--------------|-----|--------------|
| 00_2 | | | | |
| 01_2 | | | | |
| 10_2 | | | | |
| 11_2 | | | | |

- Onde o byte no endereço 00001001_2 pode ser mapeado
- Mapearmos para uns dos blocos do conjunto 10_2

Memória Principal

| Bloco | Endereço | Dado(1 byte) |
|-------|-------------|--------------|
| | ▲ 0000 0000 | Dado 0 |
| | 0000 0001 | Dado 1 |
| | 0000 0010 | Dado 2 |
| | ▼ 0000 0011 | Dado 3 |
| | ▲ 0000 0100 | Dado 4 |
| | 0000 0101 | Dado 5 |
| | 0000 0110 | Dado 6 |
| | ▼ 0000 0111 | Dado 7 |
| | ▲ 0000 1000 | Dado 8 |
| | 0000 1001 | Dado 9 |
| | 0000 1010 | Dado 10 |
| | ▼ 0000 1011 | Dado 11 |
| | ▲ 0000 1100 | Dado 12 |
| | 0000 1101 | Dado 13 |
| | 0000 1110 | Dado 14 |
| | ▼ 0000 1111 | Dado 15 |
| | ▲ 0001 0000 | Dado 16 |
| | ... | ... |

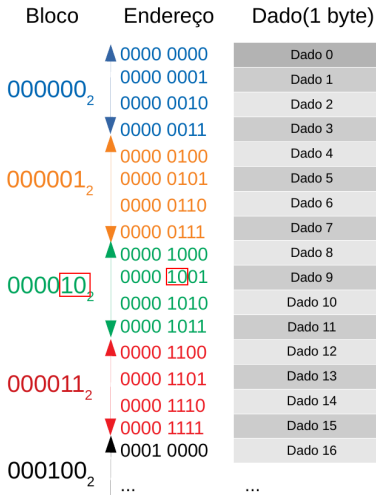
Exemplo

Cache

| Conjunto | Tag | Dado (Bloco) | Tag | Dado (Bloco) |
|----------|-----|--------------|------|------------------------------|
| 00_2 | | | | |
| 01_2 | | | | |
| 10_2 | | | 0000 | Dado8 Dado9 Dado10 Dado11 |
| 11_2 | | | | |

- Onde o byte no endereço 00001001_2 pode ser mapeado
- Mapearmos para uns dos blocos do conjunto 10_2
 - ▶ E.g., o segundo bloco

Memória Principal



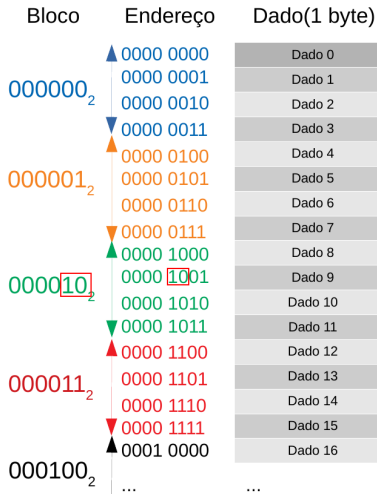
Exemplo

Cache

| Conjunto | V | Tag | Dado (Bloco) | V | Tag | Dado (Bloco) |
|----------|---|-----|--------------|---|------|------------------------------|
| 00_2 | 0 | | | 0 | | |
| 01_2 | 0 | | | 0 | | |
| 10_2 | 0 | | | 1 | 0000 | Dado8 Dado9 Dado10 Dado11 |
| 11_2 | 0 | | | 0 | | |

- Onde o byte no endereço 00001001_2 pode ser mapeado
- Também necessitamos do bit verificador

Memória Principal



Exemplo

Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Sabemos que o dado está deslocado 1 byte dentro do bloco (offset de 1 byte)

Bit de validade ativo

Cache

| Conjunt o | V | Tag | Dado (Bloco) | V | Tag | Dado (Bloco) |
|--------------|---|-----|--------------|---|------|------------------------------|
| 00_2 | 0 | | | 0 | | |
| 01_2 | 0 | | | 0 | | |
| 10_2 | 0 | | | 1 | 0000 | Dado8 Dado9 Dado10 Dado11 |
| 11_2 | 0 | | | 0 | | |

Memória Principal

| Bloco | Endereço | Dado(1 byte) |
|-------|-----------|--------------|
| | 0000 0000 | Dado 0 |
| | 0000 0001 | Dado 1 |
| | 0000 0010 | Dado 2 |
| | 0000 0011 | Dado 3 |
| | 0000 0100 | Dado 4 |
| | 0000 0101 | Dado 5 |
| | 0000 0110 | Dado 6 |
| | 0000 0111 | Dado 7 |
| | 0000 1000 | Dado 8 |
| | 0000 1001 | Dado 9 |
| | 0000 1010 | Dado 10 |
| | 0000 1011 | Dado 11 |
| | 0000 1100 | Dado 12 |
| | 0000 1101 | Dado 13 |
| | 0000 1110 | Dado 14 |
| | 0000 1111 | Dado 15 |
| | 0001 0000 | Dado 16 |
| | ... | ... |

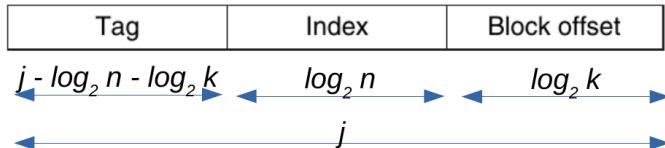
Qual o ganho?

| Associativity | Data miss rate |
|---------------|----------------|
| 1 | 10.3% |
| 2 | 8.6% |
| 4 | 8.3% |
| 8 | 8.1% |

Misses na cache de dados do Intrinsity FastMATH (64KB de cache e blocos de 16 palavras) considerando o benchmark SPEC2000 para diferentes associatividades (Patterson, Henessy; 2014).

Endereçamento com associatividade

- Considere que a memória principal é endereçada utilizando j **bits**
- Ao solicitar um endereço de memória:
 - ▶ Com **blocos de tamanho k** (número de endereços por bloco), os últimos $\lg k$ bits são utilizados para se descobrir o deslocamento no bloco (offset)
 - ▶ Tendo n **conjuntos** na memória, $\lg n$ bits após os bits de deslocamento (offset) são utilizados para se descobrir o conjunto da cache
 - ★ Índice na cache
 - ▶ Os demais bits fazem parte do campo Tag



Exemplo

- Considere uma máquina onde a memória é endereçada utilizando 64 bits. Considere ainda blocos de tamanho 16, e uma cache associativa de 4 vias com 8 conjuntos
- Quais e quantos bits são utilizados para o offset, o conjunto da cache, e o Tag?

Exemplo

- Considere uma máquina onde a memória é endereçada utilizando 64 bits. Considere ainda blocos de tamanho 16, e uma cache associativa de 4 vias com 8 conjuntos
- Quais e quantos bits são utilizados para o offset, o conjunto da cache, e o Tag?

$$\text{Tag} = 64 - 4 - 3 = 57$$

$$\log_2 8 = 3$$

$$\log_2 16 = 4$$

63 62 61 60 59 ... 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Endereço dos bits

Cache associativa de n vias

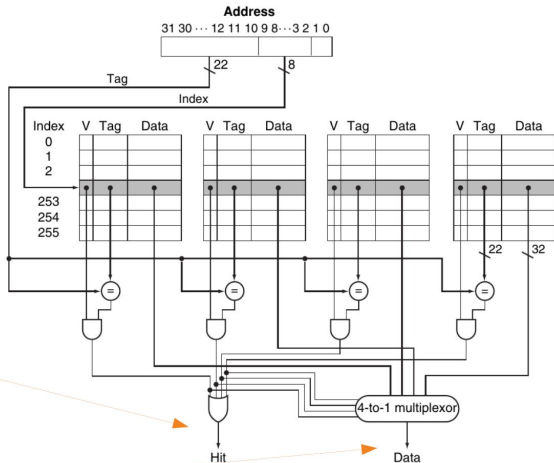
- Quando o processador solicita um endereço X
 - ▶ Os bits que representam o índice em X são usados para encontrar o conjunto
 - ▶ Os campos tag de todos os blocos do conjunto são analisados para verificar em qual bloco do conjunto o dado se encontra
 - ★ Busca em paralelo para economizar tempo
 - ★ Necessário também verificar os bits de validade de cada bloco
 - ★ Pode não encontrar em nenhum bloco: miss
 - ▶ Em caso de hit
 - ★ O campo de offset de X é usado para se obter o deslocamento dentro do bloco

Exemplo

Cache associativa de 4 vias,
blocos de tamanho 4 e com
256 conjuntos

Bit de hit é 1 se qualquer um dos
blocos do conjunto possuir o bit
de validade ligado, e o tag é igual
a parte alta do endereço

Multiplexador para escolher qual
dos blocos será enviado



Aqui sai o bloco inteiro. Os bits mais baixos são
depois são utilizados para sabermos qual parte
do bloco o processador realmente precisa.

Tempo e hardware extras

- Os comparadores e multiplexadores tomam tempo extra
 - ▶ Em uma cache diretamente mapeada, podemos nos livrar de pelo menos o multiplexador e a porta OR que verifica o hit
- A cache associativa de n vias também precisa de hardware extra
 - ▶ No exemplo de 4 vias, precisamos de 4 comparadores em paralelo
 - ▶ Para n vias, precisamos de n comparadores
 - ★ E também de um multiplexador mais complexo
 - ▶ Menor associatividade significa menos hardware
 - ★ Também pode ser um pouco mais rápido
 - ★ E.g., Multiplexadores mais simples
 - ★ No entanto aumentamos os misses

Exercícios

- 1 Considere três caches, todas contendo 4 blocos, que comportam uma palavra cada. Considere ainda que uma cache é totalmente associativa, outra é associativa de 2 vias, e outra diretamente mapeada. Assumindo que as caches estão inicialmente vazias, quantos misses geramos em cada uma delas se requisitarmos os seguintes endereços (nesta ordem): 0000000_2 , 00001000_2 , 00000000_2 , 00000110_2 , e 00001000_2 .
- 2 Considerando uma estrutura de dados baseada em arrays e outra baseada em listas encadeadas, associando ainda sua resposta com os conceitos de cache, discuta sobre:
 - ▶ Qual estrutura é mais eficiente quando analisamos a quantidade de memória principal ocupada?
 - ▶ Qual das estruturas é mais “rápida”?
- 3 Execute o `likwid-topology -c -g` em seu computador e verifique o tamanho, as associatividades e o número de conjuntos (sets) nos diferentes níveis de memória cache de seu computador.

Referências

- D. Patterson; J. Henessy. **Organização e Projeto de Computadores: Interface Hardware/Software**. 5a Edição. Elsevier Brasil, 2017.
- J. Henessy; D. Patterson. **Arquitetura de computadores: Uma abordagem quantitativa**. 6a Edição. 2017.
- STALLINGS, William. **Arquitetura e organização de computadores**. 8. ed. São Paulo: Pearson Education do Brasil, 2010.

Blocos da Cache e Associatividade

Yuri Kaszubowski Lopes

UDESC