Arquitetura de Computadores

MiEI – 2016/17

DI-FCT/UNL

Aula 19

Endereços e blocos de memória

Exemplo:

endereços de 8 bits

blocos de 4bytes

dado um endereço *E*:

E/4 = número do bloco

E%4 = byte dentro do bloco

ou seja:

4 bytes por bloco = 2^2 → 2 bits

os 2 bits menos significativos indicam bytes dentro do mesmo bloco

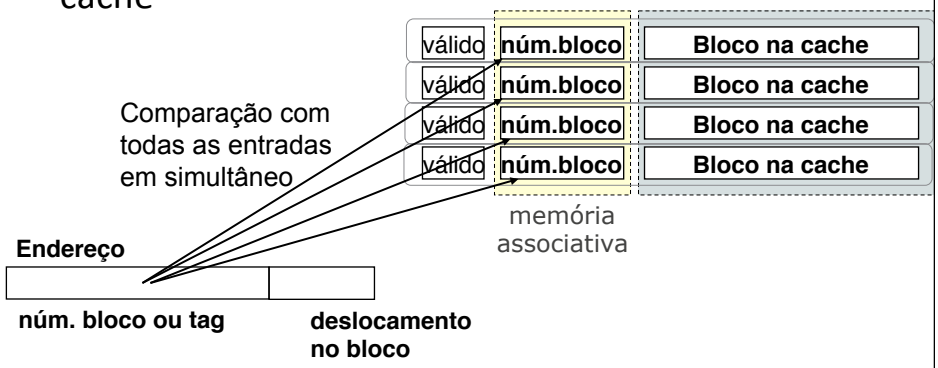
os restantes correspondem ao número do bloco

núm. bloco *deslocamento no bloco*

```
 0= 00000000
 1= 00000001   bloco 0
 2= 00000010
 3= 00000011
 4= 00000100
 5= 00000101   bloco 1
 6= 00000110
 7= 00000111
 8= 00001000
 9= 00001001   bloco 2
10= 00001010
11= 00001011
12= 00001100
13= 00001101   bloco 3
14= 00001110
15= 00001111
16= 00010000
17= 00010001   bloco 4
18= 00010010
19= 00010011
20= 00010100
```

Cache associativa pura

- Usa o número do bloco como **chave** (ou *tag*) na memória associativa para procurar o bloco na cache

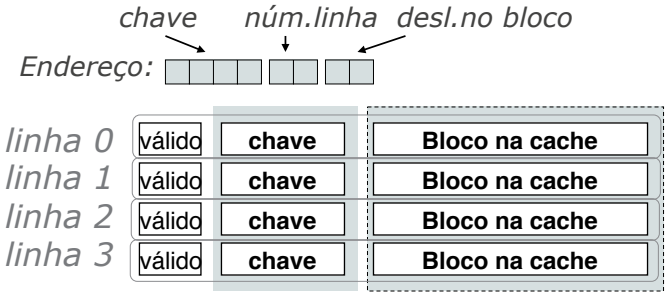


Caches associativas puras

- As Caches Associativas são muito eficientes:
  - qualquer bloco pode estar em qualquer posição
  - a pesquisa do bloco faz-se em paralelo, com tantos comparadores quantos os blocos que cabem na cache
- Mas são complexas e muito caras:
  - cada linha da cache deve guardar todos os bits do número do bloco nessa posição
  - cada comparador é de tantos bits quantos os bits no número do bloco
  - muitos comparadores, um para cada linha da cache

Cache de mapa directo

- É necessário manter para cada linha uma chave (ou *tag*) que identifique qual o bloco realmente na cache de entre os vários possíveis:
  - o que distingue os vários blocos são os bits mais significativos restantes do endereço
- Exemplo: end. de 8bits, cache de 4 linhas e 4 bytes p/bloco:



Características da Cache de mapa directo

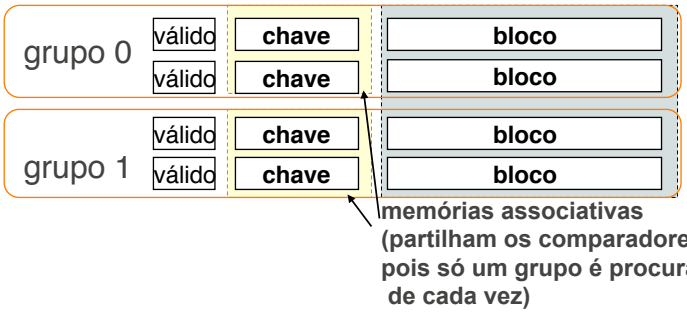
- As Caches de mapa direto são mais baratas do que as Associativas:
  - O mapa da Cache tem menos bits;
  - Não necessita de memória associativa e só precisa de 1 comparador (de tantos bits quantos os na chave dos blocos)
- São eficientes na pesquisa em Cache:
  - usam bits do endereço como índice na Cache e restantes como chave.
- Levam a colisões, mesmo que existam linhas livres!

Cache associativa por grupos (*set associative*)

- Solução híbrida de compromisso
- Procura-se responder às desvantagens das duas técnicas anteriores:
  - evitar as colisões no mapa direto
  - ter menos comparadores que a associativa pura
- Abordagem:
  - cada "linha" de mapa direto dá lugar a um grupo (conjunto) de blocos
  - cada bloco pode ficar em qualquer linha dentro do grupo
  - a busca no grupo é efectuada procurando a chave do bloco usando memória associativa

Cache associativa por grupos

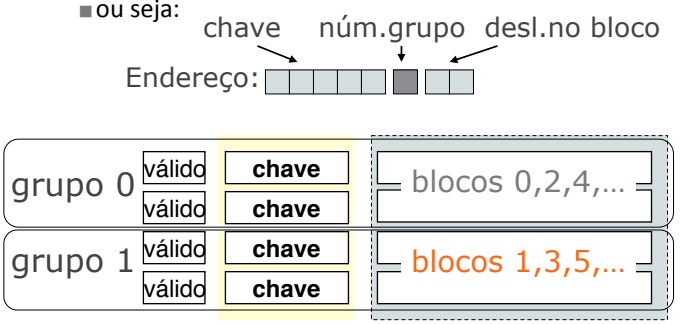
- Os blocos de memória são mapeados diretamente num grupo (ou conjunto de linhas)
  - o núm. do grupo é dado pelo: ***núm.bloco%núm.grupos***
- Dentro do grupo pode ficar em qualquer posição, como se o grupo fosse uma "mini cache associativa pura"

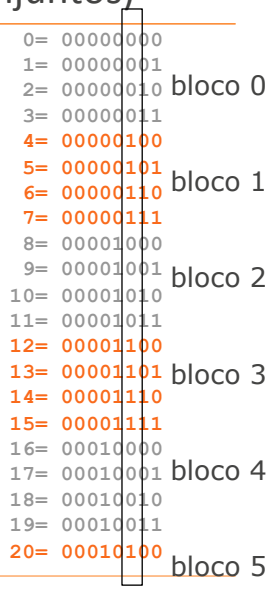


Cache associativa por grupos (ou conjuntos)

- **Exemplo, end. de 8 bits:**
  - **cache com 2 grupos**
  - **4 blocos de 4 bytes, 2 por grupo**
  - núm. do grupo é dado por: **núm.bloco%núm.grupos**

  (como no mapa direto para obter a linha)
  - ou seja:





Quando falha a cache (miss)

- Existe misses quando as localidades não se verificam:
  - por acesso a dados não contíguos em memória
    - ou estruturas que não cabem na cache
  - muda o conjunto de instruções em execução (*working-set*): devido a jumps, calls,…
    - ou este conjunto não cabe todo na cache
  - o SO troca de programa em execução
    - muda de código e de dados…

Tratamento dos misses

- Se ***Cache miss***: é preciso arranjar espaço para o novo bloco. Se associativa por grupos o conflito tem de ser resolvido apenas no grupo.
- Se linha disponível (na cache ou no grupo), marca como válida, atualiza a chave (tag) e o conteúdo do bloco
- Se cache cheia, qual o bloco a ser eliminado da cache ?
  - Um qualquer . . . ou o bloco que no futuro não vai ser necessário. Mas qual é esse?
- Se usando *write-back*, verifica bit *dirty* e atualiza memória, antes de usar a linha vítima

Escolha da vítima

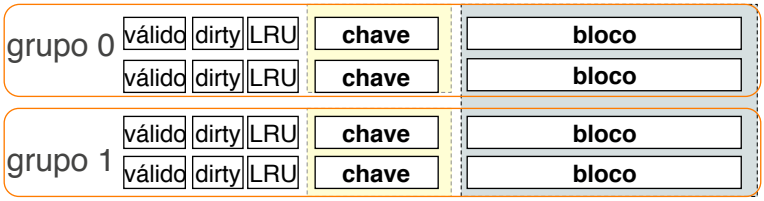
- Tentar prever o que será ou não necessário no futuro:
  - LRU – *Least Recently Used* – eliminar o bloco que há mais tempo não é usado (exige contar tempo)
  - LFU – *Least Frequently Used* – eliminar o bloco que foi referenciado menos vezes (exige contar acessos)
  - FIFO – *First In First Out* – eliminar o bloco mais antigo (exige manter lista ou tempo do 1º acesso)
  - Aleatório – escolhe-se um bloco de forma aleatória (exige um gerador de pseudo-aleatórios)

Pseudo-LRU de 1 bit

- A cada acesso a uma linha, passa a 1 o bit indicando o acesso; as linhas não acedidas recentemente têm o valor a 0
- Quando todas as linhas do grupo forem acedidas, todos os bits são mudados para 0 (estão em igualdade)
- Espera-se que, em caso de miss, deve haver pelo menos uma linha com bit de acesso a 0
- Se todas as linhas de um grupo têm o bit a 0 ou 1, considera-se que os acessos são aproximadamente semelhantes e qualquer uma pode ser substituída

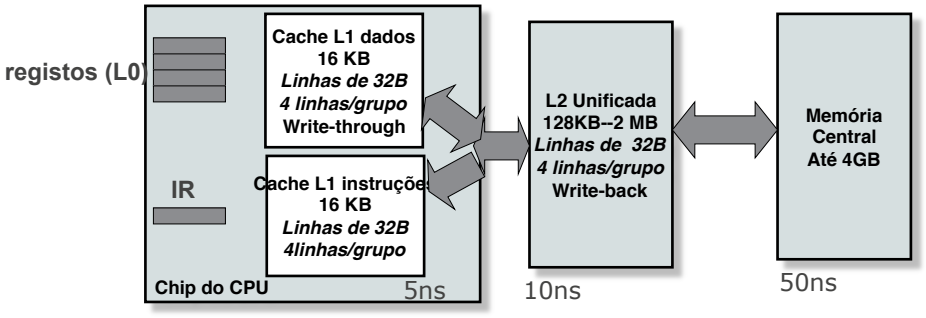
Completando a cache associativa por grupos

- Cada linha tem ainda:
  - Dirty bit –colocado a 1 a cada escrita (apenas se escritas por *write-back*)
  - LRU bit

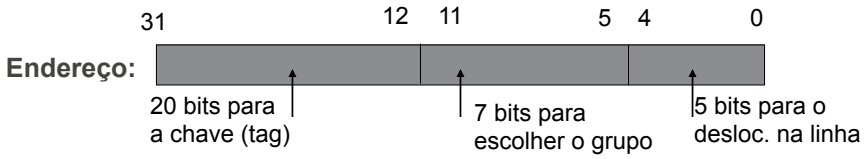


Exemplo de cache do Intel Pentium

As capacidades e tempos dependem do modelo

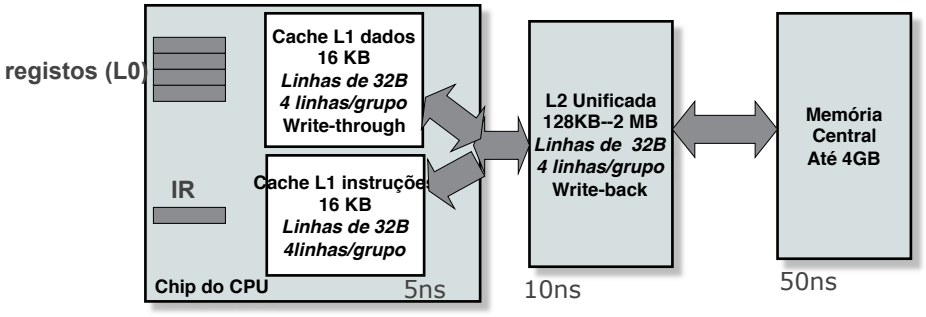


Exemplo Cache L1 (16KB) = 128 grupos * 4linhas/grupo * 32B/linha

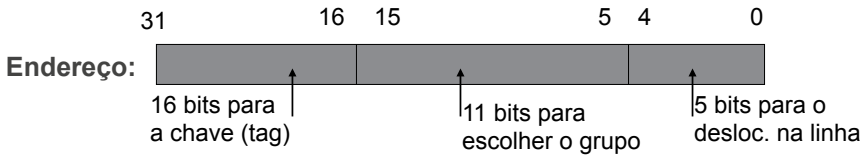


Exemplo de cache do Intel Pentium

As capacidades e tempos dependem do modelo



Exemplo de Cache L2 de 256KB = 2048grupos*4linhas/grupo*32B/linha



Exemplo `mov (393282),%eax`

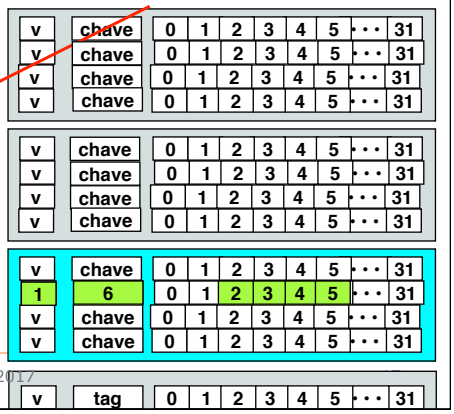
- Tratamento na L2 do endereço: 393282
  - em binário:

`0000 0000 0000 0110 0000 0000 0100 0010`

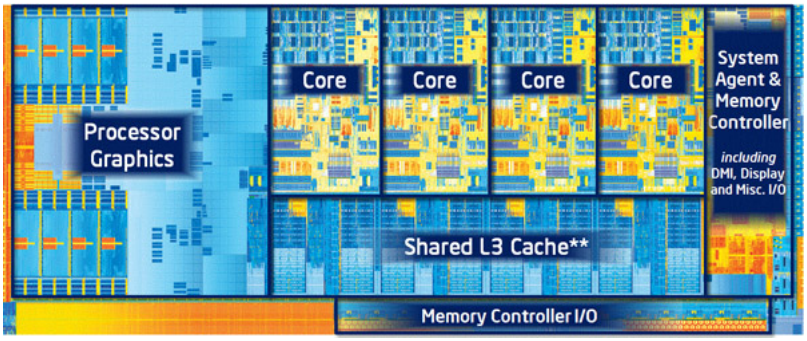
  - dividindo de acordo com a cache L2:

`0000000000000110  00000000010  00010`

deve procurar no grupo 2
a chave= 6
se encontrar
  lê a partir do byte 2
  (4 bytes)



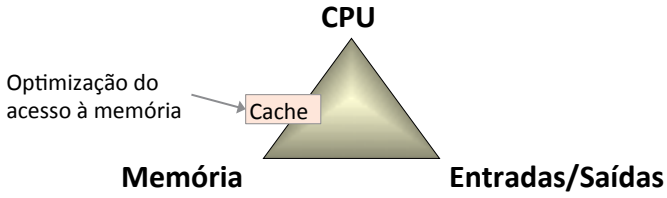
Intel Core i7-3770K



L1/core 32 KB 8-way set associative instruction caches
32 KB 8-way set associative data caches
L2/core 256 KB 8-way set associative caches
L3 shared 8 MB 16-way set associative cache
Max RAM 32 GB

Na arquitectura de computadores

- Procura-se optimizar os vários componentes na medida do respectivo peso nos tempos de execução dos nossos sistemas



Tempo de execução das instruções (CPU)

- Um programa pode executar mais rápido se o CPU o executar em menos tempo:

$$\text{Tempo} = \text{n.inst} \times \frac{\text{n.ciclos}}{\text{inst}} \times \frac{\text{tempo}}{\text{ciclo}}$$

- Menor Tempo se:
  - menos instruções → o CPU implementa as mais variadas operações de que o programa necessita
  - instruções mais rápidas → demoram menos ciclos e/ou cada ciclo pode ser mais curto (mais Hz)

Até aos anos 80

- A abordagem foi suportar directamente no *hardware* (no CPU):
  - Os mais variados tipos de instruções que os programas podem necessitar
    - As mais variadas operações aritméticas e lógicas…
  - Cada instrução suporta os mais variados operandos que o programa pode necessitar
    - Registos, memória (com vários modos de endereçamento), …
- A prioridade é reduzir o tamanho dos programas
  - Claro que também se procura reduzir o tempo de execução de cada instrução

Complexidade dos CPU

- A complexidade dos CPU é influenciada por:
  - Tipos de instruções
  - Número de operandos
  - Tipos de operandos
  - Modos de endereçamento dos operandos
  - Etc…
- O desempenho do CPU é influenciado por essa complexidade:
  - Descodificação mais complexa (recurso a micro-código)
  - Instruções de tamanho variável
  - Resolução do endereço dos operandos e obtenção do seus valores mais complexa/demorada
- Mais complexidade → mais circuitos
  - CPU maior, mais lento, consumindo mais energia, etc…

Exemplos inspirados nos Intel

```
cmp %eax, %ebx
jbe label1

É equivalente a:

cmp %ebx, %eax
jae label1
```

```
mov tabela(%ebx), %eax


Pode ser equivalente a:

add $tabela, %ebx
mov (%ebx), %eax

(se demorarem o mesmo
   tempo…)
```

RISC vs CISC

RISC – *Reduced Instruction Set Computer*

- Nova abordagem (anos 70/80) no desenho dos CPU. Simplificar para conseguir melhor desempenho:
  - Suportar um pequeno conjunto de instruções: as mais usadas
  - Instruções de tamanho fixo: Fetch mais simples e eficiente
  - Descodificação mais simples e eficiente
  - Menos instruções a optimizar, a execução pode ser mais eficiente
  - Usar espaço no CPU para mais registos e mais cache
  - Permitir explorar mais optimizações…
- A abordagem antiga passou a ser referida por:

CISC – *Complex Instruction Set Computer*

Exemplo nas duas abordagens

- Computar: C = A + B

CISC:

```
mov (A), %R1
add (B), %R1
mov %R1, (C)
```

ou mesmo:

```
add (A), (B),(C)
```

RISC:

```
mov (A), %R1
mov (B), %R2
add %R1, %R2
mov %R2, (C)
```

- Qual será mais eficiente?

→depende . . .