Arquitetura de Computadores

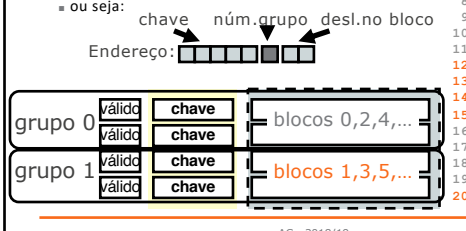
MIEI – 2018/19

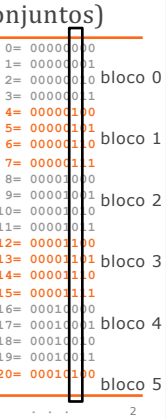
DI-FCT/UNL

Aula 21

Cache associativa por grupos (ou conjuntos)

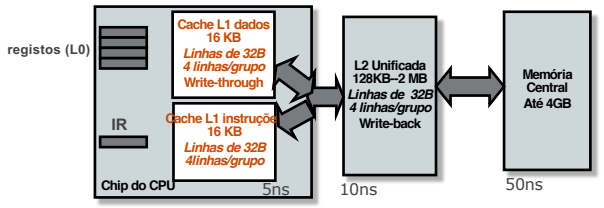
- **Exemplo, end. de 8 bits:**
  - **cache com 2 grupos**
  - **4 blocos de 4 bytes, 2 por grupo**
  - núm. do grupo é dado por: **núm.bloco%núm.grupos** (como no mapa direto para obter a linha)
  - ou seja:



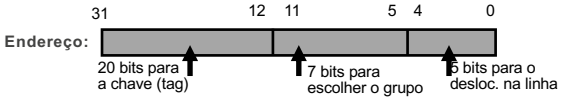


Exemplo de cache do Intel Pentium

As capacidades e tempos dependem do modelo

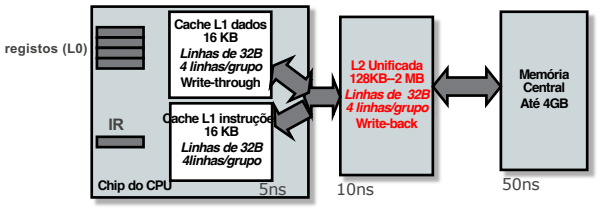


Exemplo Cache L1 (16KB) = 128 grupos * 4linhas/grupo * 32B/linha

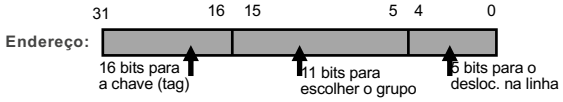


Exemplo de cache do Intel Pentium

As capacidades e tempos dependem do modelo



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



Exemplo `mov (393282),%eax`

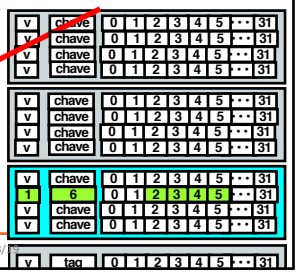
- Tratamento na L2 do endereço: 393282
  - em binário:

`0000 0000 0000 0110 0000 0000 0100 0010`

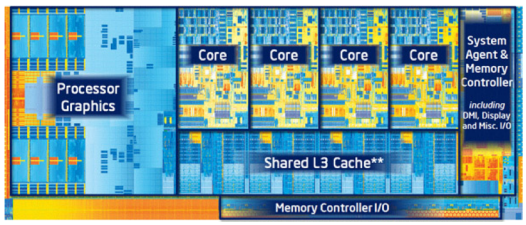
  - dividindo de acordo com a cache L2:

`0000000000000110 00000000010 00010`

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



L3 e 4 cores - 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

Tratamento dos misses

- Se ***Cache miss***: é preciso arranjar espaço para o novo bloco.
- 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*, atualiza memória, antes de usar a linha vítima

Escolha da vítima

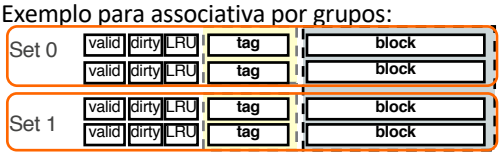
- Tentar prever o que será ou não necessário no futuro:
  - LRU – *Least Recently Used* – eliminar o bloco que há mais tempo não é usado (exige contar tempo)
  - LFU – *Least Frequently Used* – eliminar o bloco que foi referenciado menos vezes (exige contar acessos)
  - FIFO – *First In First Out* – 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

- 1 bit por linha. A cada acesso a uma linha, passa a 1 o bit indicando o acesso
- 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 mesmo valor neste bit (0 ou 1), considera-se que os acessos são semelhantes e qualquer uma pode ser substituída

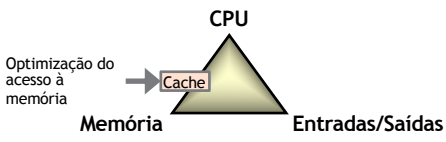
Completando a cache

- Cada linha pode ter, além dos dados:
  - Valid bit – a 1 se entrada válida
  - LRU bit – a 1 se acedida recentemente
  - Dirty bit – a 1 se escrita (apenas se escritas *write-back*)
  - Tag (ou chave) – para distinguir os blocos
- Exemplo para associativa por grupos:



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 (maior 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`	`mov tabela(%ebx), %eax`
É equivalente a:	Pode ser equivalente a:
`cmp %ebx, %eax` `jae label1`	`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 no hardware e nos compiladores…
- 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 . . .

Principais características iniciais

CISC

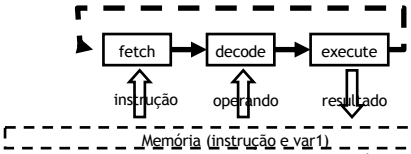
- Muitas instruções
  - Tamanho variável
- Muitos modos de endereçamento
- Instruções demoradas
  - Muitas acedem a memória
  - Nem sempre é possível executar uma instrução num ciclo de relógio
- Poucos registos

RISC

- "Poucas" instruções
  - Tamanho fixo
- Poucos modos de endereçamento
- Instruções eficientes
  - Só load/store acedem a memória
- Muitos registos
- Oportunidade para melhorar a execução de cada instrução, o consumo de energia, aumentar o *clock, o pipeline, o paralelismo, introduzir instruções vectoriais, etc…*

Pipeline de execução

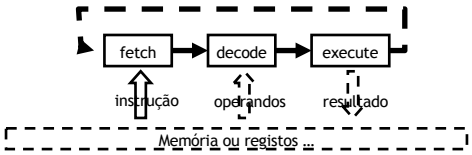
- A execução de uma instrução passa por várias fases ou estágios:
  - Vimos o ciclo: Fetch, decode, execute…
- Nos CISC cada instrução pode exigir vários acessos a memória (exemplo **`add $5, var1`**)



- Com os RISC espera-se necessitar pouco de ir a memória → instruções menos demoradas!

Pipeline

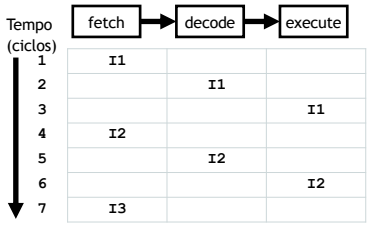
- A execução de uma instrução passa por várias fases:
  - Vimos o ciclo: Fetch, decode, execute…



  - Se em média, cada instrução despender 1 ciclo em cada fase, em média, cada instrução demora 3 ciclos para executar
    - Mas os acessos a memória são "caros"

Pipelining

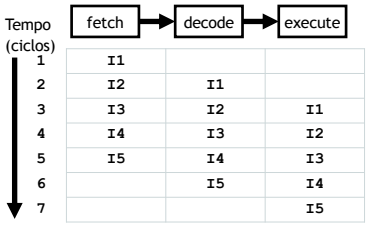
- Execução de várias instruções em sequência:



2 inst / 6 ciclos

Pipelining

- Supondo que a arquitectura permite manter o pipeline sempre ocupado, como numa linha de montagem:



5 inst / 7 ciclos

Pipelining

- Mesmo que cada instrução demore 3 ciclos, o CPU é capaz de concluir **uma instrução em cada ciclo**!
  (mesma latência, mas melhor débito)
- Tempo para executar uma sequência de 1000 instruções:
  - Sem pipelining: 1000x3 = 3000 ciclos
  - Com pipelining:
    3 ciclos para a primeira instrução (pipeline vazio)
    + 1 ciclo por cada uma das restantes
    → 3+999x1 = 1002 ciclos
  - Speedup = 3000/1002 = 2,99 (aprox. 3)