Lab1 - Grupo nº 29, L05

Francisco Ferro Pereira - **107502**
Francisco Uva - **106340**
Pedro Pais - **107482**

- As constantes e estruturas de dados utilizadas nas soluções estão definidas nos header files da respetiva tarefa: **L1Cache.h**, **L2Cache.h** e **2_way_set_associative.h.**

Tarefa 1

Constantes (definidas em L1Cache.h)

- **L1_NUM_LINES** - número de linhas na cache L1. É obtida dividindo o tamanho da cache pelo tamanho de cada bloco (L1_SIZE / BLOCK_SIZE).

Estruturas de dados (definidas em L1Cache.h)

- Para a cache L1 ter várias linhas, alterámos a struct **Cache,** passando agora a ter um array de structs **CacheLine** para guardar as diferentes linhas**.**

  ```
  typedef struct Cache {
    uint32_t init;
    CacheLine lines[L1_NUM_LINES];  // The Cache now has several lines
  } Cache;
  ```

Funções (definidas em L1Cache.c)

- **InitCache()** - Inicializa a cache definindo todas as linhas como inválidas, com dirty bit a 0 (sem modificações) e com a tag a 0.

- **accessL1(uint32_t** address**, uint8_t** *data**, uint32_t** mode**)** - Simula o acesso à cache L1, permitindo a leitura ou escrita dos dados (data) no endereço (address), dependendo do modo de acesso (MODE_READ ou MODE_WRITE). Obtemos os bits de offset, index e tag ao fazer a operação & (AND) entre o address especificado e as respectivas máscaras. No caso dos bits de index e de tag, posteriormente fazemos shift rights para obter o resultado nos bits de menor peso. Em seguida, verificamos se há hit (se a linha está válida e se a tag do bloco na linha corresponde à tag do endereço). Se sim, dependendo do modo de acesso, escrevemos no bloco (pondo o dirty bit a 1) ou lemos dele (dirty bit não é alterado), adicionando o tempo total despendido pela ação em questão. Em caso de miss, temos de carregar o bloco da DRAM através da função **accessDRAM()** antes de executar a operação de escrita ou leitura**.** Antes disso, verificamos se a linha da cache para onde vamos carregar tem modificação (dirty bit a 1), caso tenha temos de escrever o bloco para memória antes de o substituir de forma a

preservar a integridade dos dados (write back policy). Por fim, executamos a operação e adicionamos o tempo total despendido pela mesma.

Tarefa 2

Constantes (definidas em L2Cache.h)

- **L2_NUM_LINES** - número de linhas na cache L2. É obtida dividindo o tamanho da cache pelo tamanho de cada bloco (L2_SIZE / BLOCK_SIZE).

Estruturas de dados (definidas em L2Cache.h)

- Alterámos a struct **Cache** para acomodar a cache L1 e L2..

```
typedef struct {
  uint32_t init;
  CacheLine L1[L1_NUM_LINES];
  CacheLine L2[L2_NUM_LINES];
} Cache;
```

Funções (definidas em L2Cache.c)

- **InitCache()** - Inicializamos a cache L1 em conjunto com a cache L2, do mesmo modo que na tarefa 1.

- **accessL2(uint32_t** address**, uint8_t** *data**, uint32_t** mode**) -** Simula o acesso a L2. O acesso é feito da mesma forma que em **accessL1()**. No entanto, neste caso acedemos primeiro a L1 e em caso de miss acedemos a L2. Se ao aceder a L2 ocorrer um miss novamente, acedemos à DRAM. No caso da linha desejada estar ocupada por um bloco dirty, a write back policy é feita na seguinte hierarquia:

  L1 -> L2 -> DRAM

- **is_L1_dirty(uint32_t** address**)** - Função que retorna o valor do dirty bit na linha da cache L1 do endereço especificado. É utilizada para preservar o estado do dirty bit ao dar evict de um bloco da cache L1 para a cache L2.

- **is_L2_dirty(uint32_t** address**)** - Função que retorna o valor do dirty bit na linha da cache L2 do endereço especificado. É utilizada para preservar o estado do dirty bit ao carregar um bloco da cache L2 para a cache L1.

Tarefa 3

Constantes (definidas em 2_way_set_associative.h)

- **NUM_SETS** - É obtido dividindo o número de linhas da cache L2 pelo número de associatividade. (L2_NUM_LINES / ASSOCIATIVITY)

Estruturas de dados (definidas em 2_way_set_associative.h)

- Alterámos a struct Cache e criámos uma struct Set.

```
typedef struct {
  uint32_t init;
  CacheLine L1[L1_NUM_LINES];
  Set L2[NUM_SETS];
} Cache;

typedef struct Set {
  CacheLine line1;
  CacheLine line2;
  CacheLine* head;
  CacheLine* tail;
} Set;
```

- A cache L2 é agora um array de sets invés de um array de linhas. Cada Set tem duas linhas (2 é o número de associatividade) e dois ponteiros head e tail, em que a head é o ponteiro para o bloco mais recentemente acedido no set e tail é o ponteiro para o bloco menos recentemente acedido. Estes ponteiros permitem implementar a política de evicção LRU.

Funções (definidas em 2_way_set_associative.c)

- **initCache()** - Inicializa a cache L1 directly mapped e a cache L2 2 way set associative.

- **accessL2()** - Obtemos o index do endereço para descobrir o set do bloco. Descobrindo o set, verificamos se a tag do endereço condiz com a do bloco guardado na primeira linha. Se sim, temos um hit na primeira linha. Se não, testamos com a segunda. Se as tags condizem temos um hit na segunda linha, caso contrário temos um cache L2 miss. Nesta última situação temos de substituir o bloco que já não é acedido há mais tempo para carregar outro da DRAM. Esse bloco está referenciado no ponteiro tail. Ao carregar o novo bloco para a tail, este passa a ser o endereço a ser mais recentemente acedido. Assim, a tail passa a ser a nova head, e a head a nova tail. A cada acesso feito, seja cache hit ou cache miss, os ponteiros head e tail são atualizados de forma a refletir o bloco mais e menos recentemente acedido.