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

- **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).

- 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];  // Cache has several lines
} Cache;
```

- **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**)** - A função **accessL1** simula o acesso à cache L1, permitindo ler ou escrever dados no endereço especificado. Primeiro, extrai os bits de offset, index e tag do endereço usando operações AND com máscaras apropriadas, seguido de shifts para alinhar os bits. Verifica se há um hit (linha válida e tag correspondente). Em caso de hit, lê ou escreve no bloco, marcando o bit dirty em caso de escrita e acumulando o tempo gasto pela mesma. Se houver miss, o bloco é carregado da DRAM usando **accessDRAM()**. Se o bloco na linha da cache estiver modificado (dirty), é escrito de volta na memória antes de ser substituído (política de write back). Após isso, realiza-se a operação desejada e adiciona-se o tempo despendido.

Tarefa 2

- **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).

- 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;
```

- **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

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

- Alterámos a struct Cache e criámos uma struct Set.

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

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

- A cache L2 é 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.

- **accessL2()** - Obtemos o index para identificar o set do bloco e verificamos se a tag do endereço bate com a tag da primeira linha. Se sim, há um hit; se não, verificamos a segunda linha. Caso nenhuma condiz, ocorre um miss na cache L2. Nesse caso, substituímos o bloco não acessado há mais tempo, usando o ponteiro tail para referenciar esse bloco. O novo bloco carregado torna-se o mais recente, atualizando tail e head. A cada acesso, os ponteiros são ajustados para refletir o bloco mais e menos acessado.

Testes

Tarefa 1 - Fazer duas operações de escrita consecutivas em dois endereços diferentes que mapeiam para o mesmo bloco. Verificamos que o primeiro resulta num cache miss compulsivo e o segundo um cache miss por conflito que é resolvido carregando o bloco da DRAM diretamente para a cache L1.

Tarefa 2 - Fazer várias operações de escrita consecutivas em endereços diferentes que mapeiam para o mesmo bloco. Os primeiros acessos resultarão em misses, levando a popular ambas as caches. Se acedermos novamente aos endereços conseguiremos maioritariamente L2 hits e ocasionalmente L1 hits.

Tarefa 3 - Verificar se dois endereços diferentes que mapeiam para o mesmo conjunto podem ser armazenados simultaneamente na cache L2. Isto permite verificar que a cache não substitui desnecessariamente dados quando há outra slot disponível no set. Adicionalmente, podemos verificar se um terceiro endereço que mapeia para o mesmo set leva à evicção do primeiro endereço que carregámos para o set (teste da política LRU).