50.002 COMPUTATIONAL STRUCTURES

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Cache Issues

Natalie Agus (Fall 2018

1 Basic Caching Algorithm

Figure 1 illustrates the basic caching algorithm. We can READ or WRITE to the cache, and it will access the memory on WRITE or when READ command has a cache miss (HIT = 0). k is the row index of the cache, which is the full address for FA caches, or the lower k bits for DM caches. The write strategy assumed here is **write-through**. Refer to Section 9 for the complete caching algorithm.

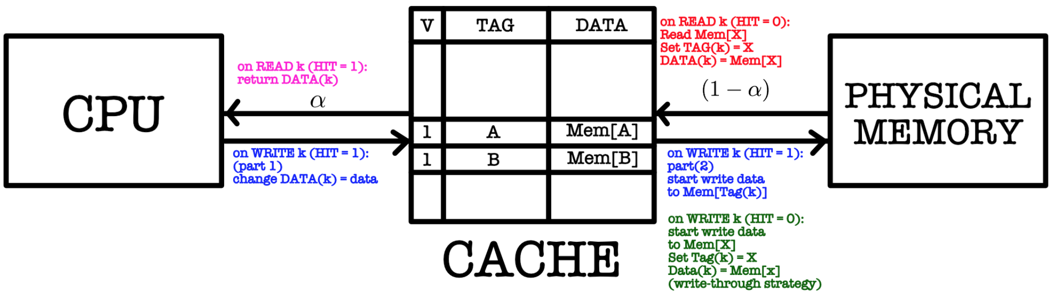

Figure 1

2 Cache Design Issues

The access to the main memory is the bottleneck for performance. Cache is used to reduce the frequency of access to the main memory whenever possible. There are several design issues for cache:

1. Associativity : how many different addresses can be stored in the cache. For DM cache, 2^k **is set to be the size of the cache** (total number of cache lines). A 1-to-1 mapping between each combination of the last k bit to the cache line is required.

2. Replacement strategy
3. Block size
4. Write strategy : when to write from cache to memory

3 Pros and Cons of FA and DM Caches

	FA Cache	**DM Cache**
Tag Content	All address bits	Higher t address bits
Tag Index	None	Lower k address bits
Cache Size	Any	2^k
Data	Mem[A]	Mem[A]
Performance	The gold standard on how well a cache should perform	Performs slower than FA cache on average
Contention	Does not have address contention. Address-data can fit on any cache line.	Has address contention. **The probability of contention is inversely proportional to cache size**.
Cost	Expensive, many boolean circuits for comparators	Cheap, only one comparator in a cache
Replacement Strategy	Yes: LRU, FIFO, Random	No
Mapping	No mapping	Lower k-bits for tag index, upper t-bits for tag content
Application	Good when cache size is small, less important when cache size is large	Bad when cache size is small (more contention), good when cache size is large

Table 1

4 N-way set associative cache

4.1 Purpose

Finding the balance point:

1. Although DM cache is cheap, it suffers from contention problem.
2. Contention mostly occurs within a certain block of addresses (called independent hot-spots)
3. This is due to locality of reference in each of different address range.
4. Hence, some associativity (and not full associativity) is needed.

5. We should allow each location (lower k-bits directly mapped) to be stored in a restricted set of cache lines (see Fig 2 to know what is a 'set' and what is a 'cache line').

4.2 Method

One solution is to build N-way set associative cache. See Figure 2.

1. We have N DM caches.
2. The cells in the same 'row' marked in red is called to belong in the same **set**.
3. The cells in the same 'column' of DM caches marked in blue is said to belong in the same **cache line**.
4. Given the same k-bits lower address, it can be stored in any of the N cache lines in the **same set**.
5. However a different combination of k-bits lower address will have to be direct mapped on different set.
6. Lookup operation: Parallel operation of N direct-mapped cache, each with 2^k lines in the cache line.

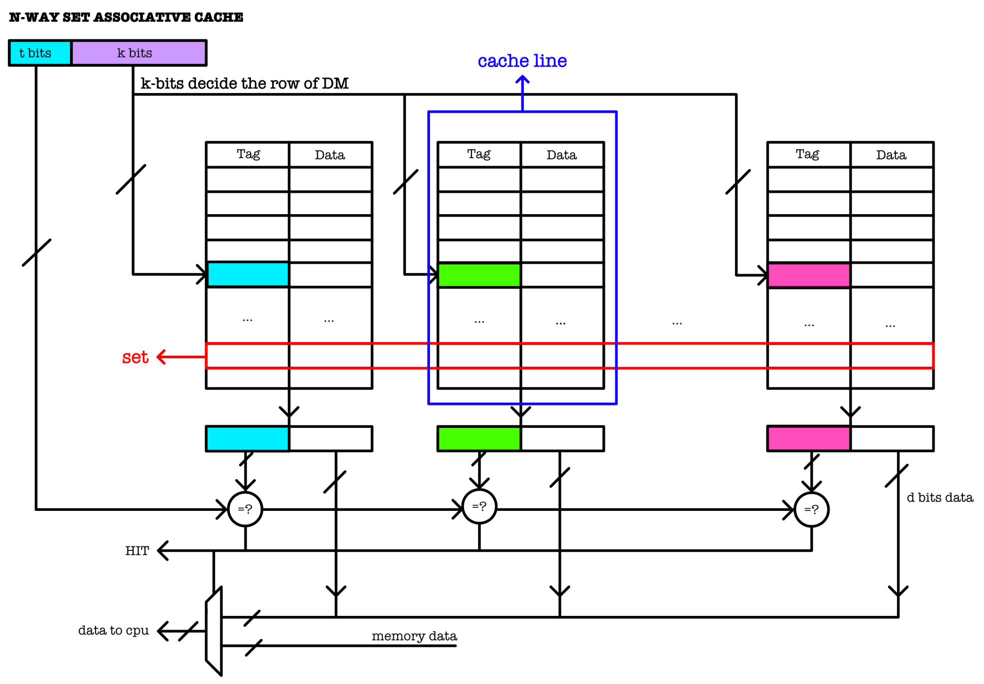

Figure 2

4.3 Replacement Strategy

The replacement strategy is required to find out which of the N-cache lines in the same set can be replaced when there's a cache miss to that particular address request. There are three common strategies:

1. **LRU : Least Recently Used.**
   (a) Replace least recently used item, good when N is small
   (b) Need to keep ordered list of N items ($N!$ orderings),
   (c) Overhead is $O(\log_2 N!) = O(N \log_2 N)$ "LRU bits" per set
   (d) Plus complex logic to re-order the list after every cache access
2. **FIFO / LRR : Least Recently Replaced**
   (a) Replace oldest item
   (b) Overhead is $O(\log_2 N)$ bits/set, because one needs just one pointer (indicator bits) that tells us the oldest item within each set.
3. **Random**
   (a) Use pseudo random generator to get reproducible behavior
   (b) Good when N is huge

There's no one best / winning replacement strategy. One replacement strategy can be better than the other depending on cases.

5 Increasing Block Size

Sometimes we would want to enlarge each line in cache (more data per tag), especially if there's high locality of reference: Here we have:

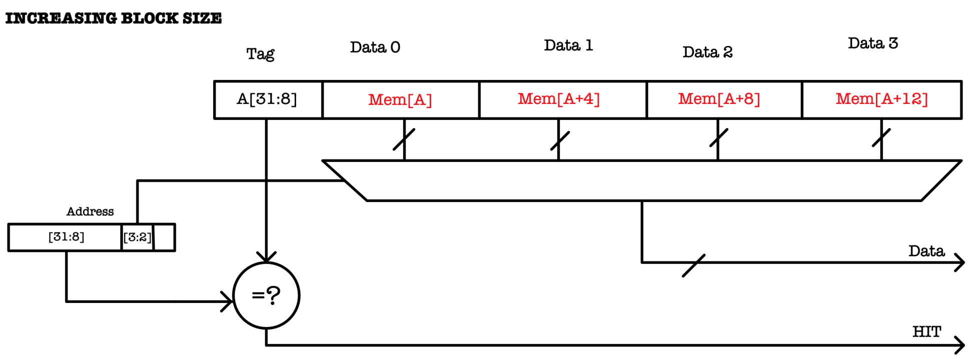

Figure 3

1. Blocks of 2^B words per row, in this case in Figure 3 $B = 2$.

2. Pros: locality of reference means there's a high likelihood that the words from the same block will be required together

3. Cons: risk of fetching unaccessed words

4. Figure 3 uses byte addressing (sometimes one can use word addressing too, especially for problem sets to make it easier for our computation), so recall that the lower two bits of address is always 0.

6 Byte VS Word Addressing

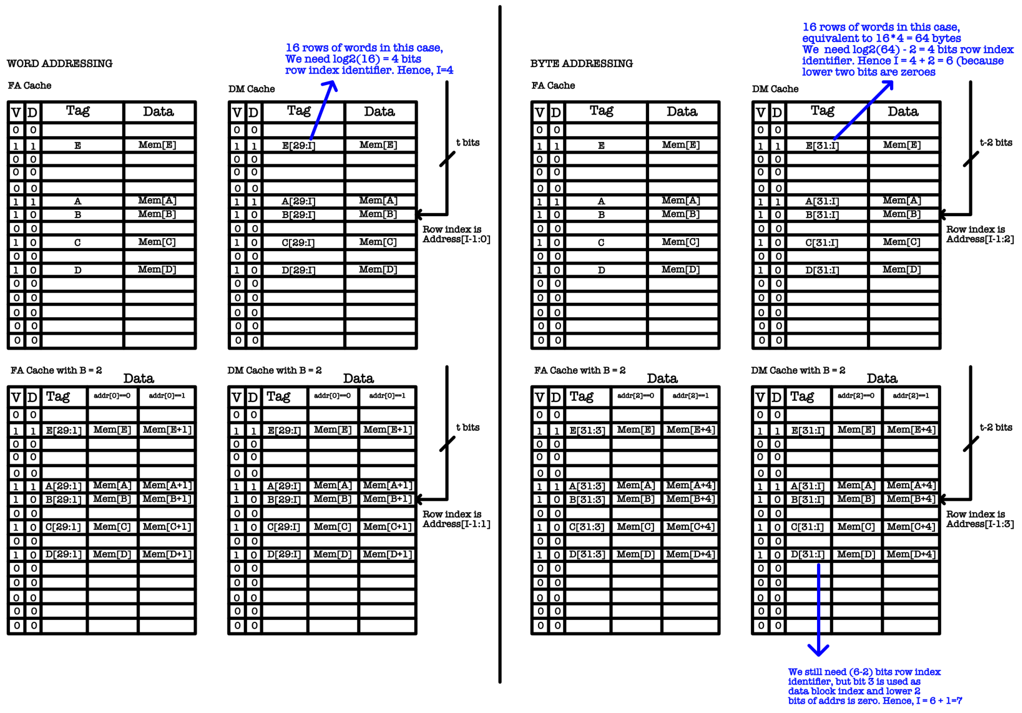
WORD ADDRESSING
FA Cache
DM Cache
16 rows of words in this case, We need log2(16) = 4 bits row index identifier. Hence, I=4
t bits
Row index is Address[I-1:0]
FA Cache with B = 2
DM Cache with B = 2
Row index is Address[I-1:1]
BYTE ADDRESSING
16 rows of words in this case, equivalent to 16*4 = 64 bytes We need log2(64) - 2 = 4 bits row index identifier. Hence I = 4 + 2 = 6 (because lower two bits are zeroes
t-2 bits
Row index is Address[I-1:2]
Row index is Address[I-1:3]
We still need (6-2) bits row index identifier, but bit 3 is used as data block index and lower 2 bits of addrs is zero. Hence, I = 6 + 1=7
V
D
Tag
Data
addr[0]==0
addr[0]==1
addr[2]==0
addr[2]==1
E
A
B
C
D
Mem[E]
Mem[A]
Mem[B]
Mem[C]
Mem[D]
E[29:I]
A[29:I]
B[29:I]
C[29:I]
D[29:I]
E[29:1]
A[29:1]
B[29:1]
C[29:1]
D[29:1]
Mem[E+1]
Mem[A+1]
Mem[B+1]
Mem[C+1]
Mem[D+1]
E[31:I]
A[31:I]
B[31:I]
C[31:I]
D[31:I]
E[31:3]
A[31:3]
B[31:3]
C[31:3]
D[31:3]
Mem[E+4]
Mem[A+4]
Mem[B+4]
Mem[C+4]
Mem[D+4]

Figure 4

All along up until this point, we learn that in practice, data has byte addressing. However for the ease of calculation in practice questions and problem sets, now we do word addressing. Figure 4 above illustrates the difference between byte addressing that we already know and word addressing.

As shown in Fig, the number of bits needed to address **the same size of RAM** using word address is: # of byte addressing - 2. So in β architecture, the address is **30 bits if word addressing is used**

7 Valid, Dirty, and LRU Bit

In Figure 6, we add two extra bits of storage in the cache : V and D. Although not written, FA and NW caches also have LRU bits for replacement strategy.

7.1 V: Valid Bit

The valid bit indicates that the particular cache row (also called cache line, but it is a different graphical representation from 'cache line' in the N-way set associative cache) contains data from memory and not empty or redundant value. We only check cache lines with valid bit = 1.

7.2 D: Dirty Bit

The dirty bit is set to 1 iff the CPU writes to cache and it hasn't been stored to the memory (memory is outdated).

7.3 LRU: LRU Bit

The LRU bit is present in each cache line (for FA), and each cache set-cacheline cell (for NW), **regardless of the block size because R/W with block size more than 1 is always done in parallel**.

8 Cache Writes

Most memory accesses are READs, but we can handle WRITEs in three different ways. All three methods require CPU to write data to cache first.

8.1 Write-through

CPU writes are done in the cache first by setting TAG = Addr, and Data = new Mem[Addr] in an available cache line, but also written to the main memory immediately. This **stalls** the CPU until write to memory is complete, but memory always holds the "truth", and is never oudated.

8.2 Write-behind

CPU writes are also cached, and write to the main memory is immediate but it is buffered or pipelined. CPU keeps executing next instructions while writes are completed (in order) in the background.

8.3 Write-back

CPU writes are also cached, but not immediately written to the main memory. Memory contents can be ”stale”. Typically CPU will write to the main memory only if the data in cache line needs to be replaced and that this data has been changed or is new. **This requires the dirty bit** in the cache.

9 The Complete Caching Algorithm

```
On READ request ADDR:
     Check for ADDR in cache in locations with V==1,
       if HIT: return the corresponding data from cache to CPU
       else if MISS:
          if DM cache:
              selected data to replace = data in the index
          else if FA cache:
               selected data to replace = data slot with least recent LRU bit
          Check if the selected data is dirty, if D == 1
                write the dirty data to RAM first
          fetch ADDR and its data from RAM
          if DM cache:
               write them to the corresponding row index k
          if FA cache:
               write them to the data slot with that least recent LRU bit
               update LRU bit
          return data to CPU

On WRITE request ADDR:
     Check for ADDR in cache in locations with V==1,
       if MISS:
          if DM cache:
              selected data to replace = data in the index
          else if FA cache:
              selected data to replace = data slot with least recent LRU bit
          Check if the selected data is dirty, if D == 1
               write the dirty data to RAM first
          fetch ADDR and its data from RAM
          if DM cache:
               write them to the corresponding row index k
          if FA cache:
               write them to the data slot with that least recent LRU bit
               update LRU bit

      (note that when this line is reached, it means that its either HIT,
      or MISS code has already been executed)
      if Write-Through:
          write the corresponding data in cache
          write in RAM
          return to CPU
      else if Write-Back:
          write the corresponding data in cache
          set that cache line D = 1
          return to CPU
```