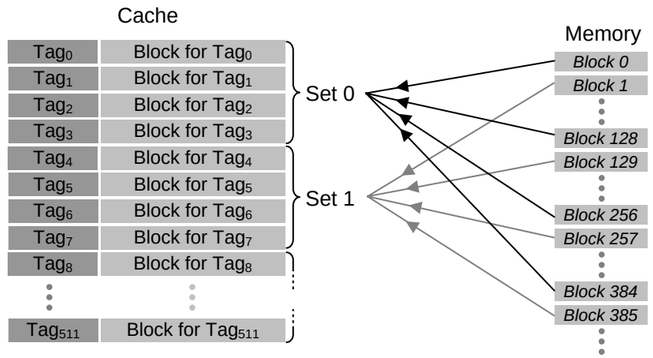
Set Associative Mapping Algorithm

POINTS OF INTEREST:

- Address length is s + w bits
- Cache is divided into a number of sets, v = 2^d
- k blocks/lines can be contained within each set
- k lines in a cache is called a k-way set associative mapping
- Number of lines in a cache = v•k = k•2^d
- Size of tag = (s-d) bits
- Each block of main memory maps to only one cache ***set***, but k-lines can occupy a set at the same time
- Two lines per set is the most common organization.
  - This is called 2-way associative mapping
  - A given block can be in one of 2 lines in only one specific set
  - Significant improvement over direct mapping
  - Replacement algorithm simply uses LRU with a USE bit. When one block is referenced, its USE bit is set while its partner in the set is cleared



Effect of Cache Set Size on Address Partitioning

Tag bits	Set ID bits	Word ID bits	
18 bits	9 bits	3 bits	Direct mapping (1 line/set)
19 bits	8 bits	3 bits	2-way set associative (2^1 lines/set)
20 bits	7 bits	3 bits	4-way set associative (2^2 lines/set)
21 bits	6 bits	3 bits	8-way set associative (2^3 lines/set)
⋮	⋮	⋮	
25 bits	2 bits	3 bits	128-way set associative (2^7 lines/set)
26 bits	1 bit	3 bits	256-way set associative (2^8 lines/set)
27 bits		3 bits	Fully associative (1 big set)

Writing to a Cache

Must not overwrite a cache block unless main memory is up to date
Two main problems:

- If cache is written to, main memory is invalid or if main memory is written to, cache is invalid – Can occur if I/O can address main memory directly
- Multiple CPUs may have individual caches; once one cache is written to, all caches are invalid

Write Through: All writes go to main memory as well as cache

- Multiple CPUs can monitor main memory traffic to keep local (to CPU) cache up to date
- Lots of traffic and slows writes

Write Back: Updates initially made in cache only

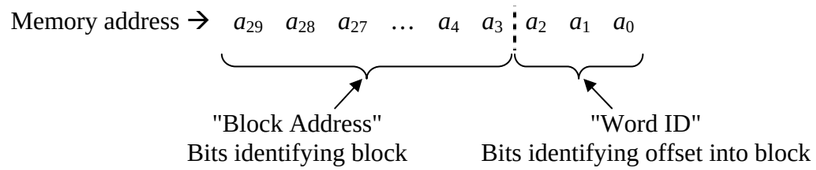
- Update bit for cache slot is set when update occurs
- If block is to be replaced, write to main memory only if update bit is set
- Other caches get out of sync
- I/O must access main memory through cache
- Research shows that 15% of memory references are writes

Multiple Processors with Multiple Caches: Even if a write through policy is used, other processors may have invalid data in their caches

Memory Blocks

DEFINITION: A ***block*** is a group of neighboring words in memory identified by bits of address excluding "w" word ID bits

EXAMPLE: Assume a block uses three word ID bits, i.e., w=3. Memory addresses for this system are therefore broken up as shown in the figure below.



EXERCISE: Which addresses below are contained the same block as the address 0x546A5 for a block size of 8 words?

a.) 0x536A5 a.) 0x546B5 a.) 0x546AF a.) 0x546A0 a.) 0x546C7 a.) 0x546A2

Locality of Reference Principle

DEFINITION: During execution, memory references of both data and instructions tend to cluster together over a short period of time. Examples of instructions that might cluster together are iterative loops or functions. It might even be possible for a compiler to organize data so that data elements that are accessed together are contained in the same block.

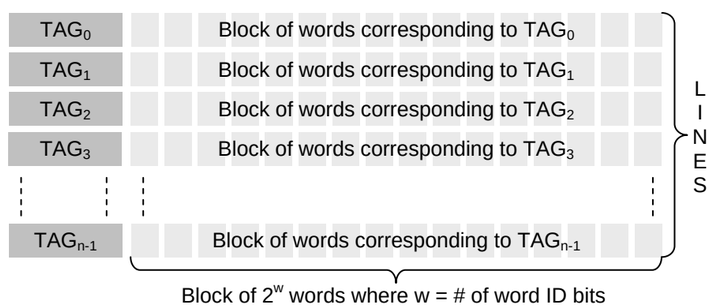
EXAMPLE: Identify how many times the processor "touches" each piece of data and each line of code in the following snippet of code.

```
int values[6] = {9, 34, 67, 23, 7, 3};
int count;
int sum = 0;
for (count = 0; count < 8; count++)
     sum += values[count];
```

General Organization of a Cache

POINTS OF INTEREST:

- Tags are unique identifiers derived from the address of the block contained in the corresponding line.
- When one word is loaded into the cache, all of the words in the same block are loaded into a single line.
- The number of lines in a cache equals the size of the cache divided by the number of words in a block.



Direct Mapping Algorithm

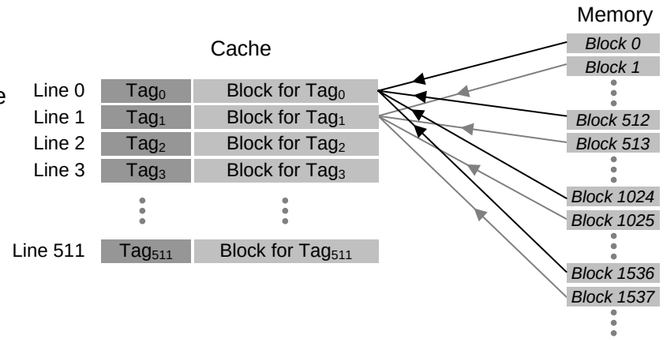
POINTS OF INTEREST:

- Each block of main memory maps to only one cache line – i.e. if a block is in cache, it will always be found in the same place
- Line number is calculated using the function

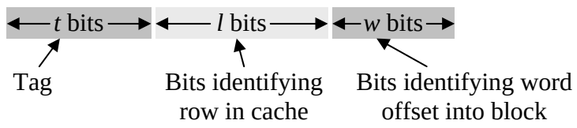
  i = j modulo m

  where

  i = cache line number
  j = block number derived from address
  m = number of lines in the cache
- The memory address is divided into three parts which are from right to left: the word id, the bits identifying the cache line number where the block is stored, and the tag.
- 2^l = number of lines in cache
- 2^w = number of words in a block
- 2^t = number of blocks in memory that map to the same line in the cache.



Direct Mapping Partitioning of Memory Address



EXAMPLE: What cache line number will the following addresses be stored to, and what will the minimum address and the maximum address of each block they are in be if we have a cache with 2^{12} = 4K lines of 2^4 = 16 words to a block in a 2^{28} = 256 Meg memory space?

a.) 0x9ABCDEF b.) 0x1234567 c.) 0xD43F6C2

EXAMPLE: Assume that a portion of the tags in the cache in our example looks like the table below. Which of the following addresses are contained in the cache?

a.) 0x438EE8 b.) 0xF18EFF c.) 0x6B8EF3 d.) 0xAD8EF3

Tag (binary)	Line number (binary)	Addresses wi/ block			
		00	01	10	11
0101 0011	1000 1110 1110 10				
1110 1101	1000 1110 1110 11				
1010 1101	1000 1110 1111 00				
0110 1011	1000 1110 1111 01				
1011 0101	1000 1110 1111 10				
1111 0001	1000 1110 1111 11				

EXAMPLE: For the previous example, how many lines does the cache contain? How many blocks can be mapped to a single line of the cache?

PROS: Simple & inexpensive

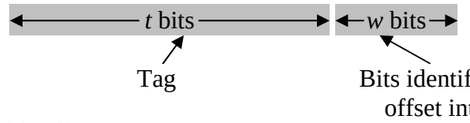
CONS: If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high (thrashing)

Fully Associative Mapping Algorithm

POINTS OF INTEREST:

- A main memory block can load into any line of cache
- Memory address is interpreted as:
  - Least significant w bits = word position within block
  - Most significant s bits = tag used to identify which block is stored in a particular line of cache
- Every line's tag must be examined for a match
- The algorithm for storing is independent of the size of the cache
- Cache searching gets expensive and slower

Fully Associative Mapping Memory Address Partitioning



EXAMPLE: Assume that a portion of the tags in the cache in our example looks like the table below. Which of the following addresses are contained in the cache?

a.) 0x438EE8 b.) 0xF18EFF c.) 0x6B8EF3 d.) 0xAD8EF3

Tag (binary)	Addresses wi/ block			
	00	01	10	11
0101 0011 1000 1110 1110 10				
1110 1101 1100 1001 1011 01				
1010 1101 1000 1110 1111 00				
0110 1011 1000 1110 1111 11				
1011 0101 0101 1001 0010 00				
1111 0001 1000 1110 1111 11				

Replacement Algorithms

There must be a method for selecting which line in the cache is going to be replaced when there's no room for a new line

POINTS OF INTEREST:

- Hardware implemented algorithm for speed
- There is no need for a replacement algorithm with direct mapping since each block only maps to one line – just replace line that is in the way.
- Types of replacement algorithms:
  - Least Recently used (LRU) – replace the block that hasn't been touched in the longest period of time
  - First in first out (FIFO) – replace block that has been in cache longest
  - Least frequently used (LFU) – replace block which has had fewest hits
  - Random – just pick one, only slightly lower performance than use-based algorithms LRU, FIFO, and LFU