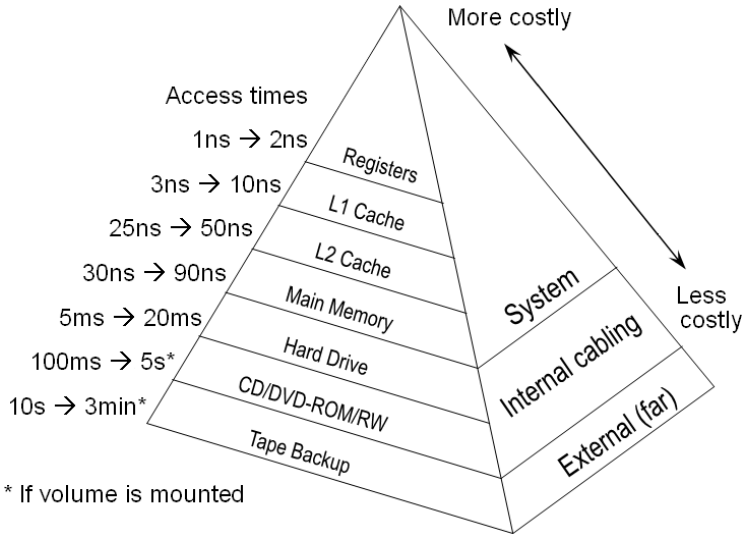
# RAM CACHES



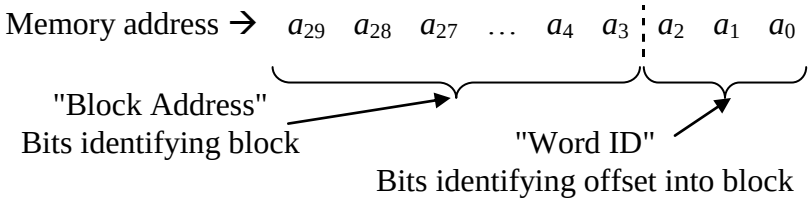

## Memory Access Methods

***Sequential:*** Read through all data until requested data are found. Access time depends on relative positions of start and target. (e.g., tape)
***Direct:*** Jump first to block/sector, then perform sequential search. Faster than sequential – you can "jump" over some data. (e.g., hard disk)
***Random:*** Address identifies location exactly. Access time is consistent and independent of previous access. (e.g., RAM)
***Associative:*** Addressing information stored with data. Data located by comparing desired address with address portion of stored elements. Access time is consistent. (e.g., cache)

## 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.




## 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.

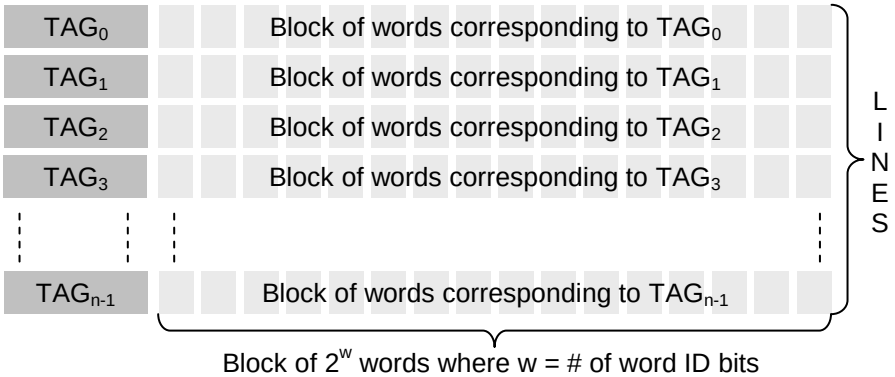
**EXAMPLE:** Identify how many times the processor "touches" each piece of data and each line of code in the following snippet of code.

```
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:**

- When the CPU requests a word, the entire block is loaded into a line of the cache.
- ***Tags:*** Unique identifiers derived from the source address of the block contained in the line.
- The number of lines in a cache equals the size of the cache divided by the number of words in a block.



# Replacement Algorithms

Must have a hardware implemented algorithm to select which line in cache is going to be replaced when there's no room for a new line

**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 slight performance hit wrt/ use-based algorithms LRU, FIFO, and LFU

---

# Direct Mapping Algorithm

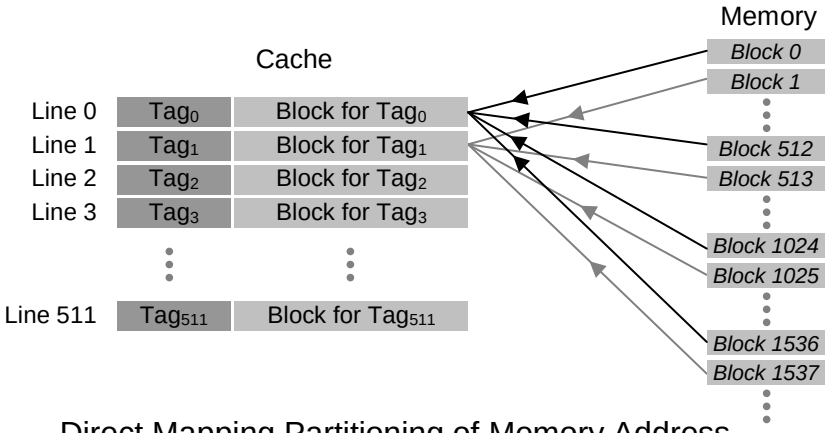
**POINTS OF INTEREST:**

- 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

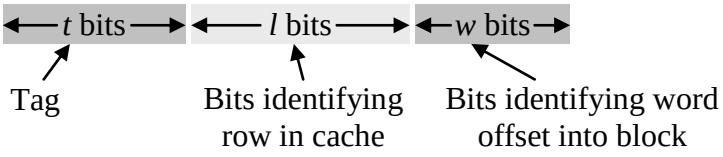
  where

  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




**PROS:** Simple & inexpensive. No need for a replacement algorithm.
**CONS:** If program repeatedly accesses 2 blocks that map to same line, get high cache misses (thrashing)

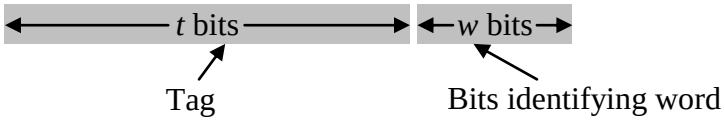
---

# Fully Associative Mapping Algorithm

**POINTS OF INTEREST:**

- A main memory block can load into any 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
- 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

Fully Associative Mapping Memory Address Partitioning

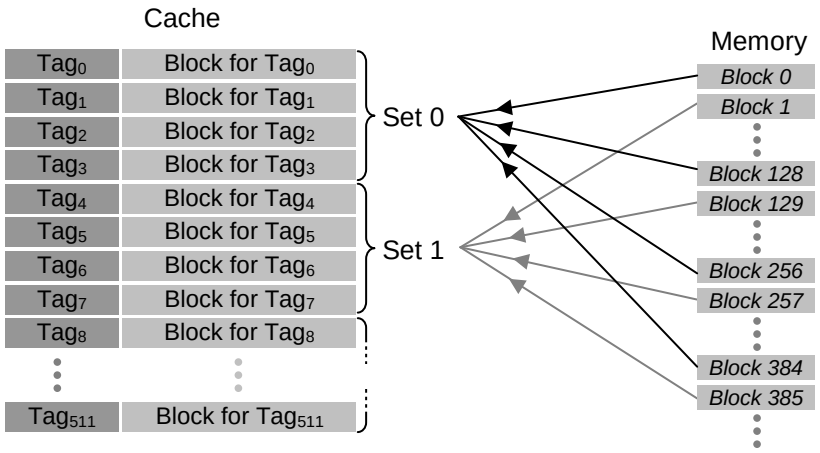



**PROS:** Eliminates thrashing **CONS:** Every line's tag must be examined for a match → expensive and slow

# 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 = $v \cdot k = k \cdot 2^d$
- 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) |

**PROS:** Greatly reduces thrashing. Typically used for cache implementations
**CONS:** More expensive than direct mapping, but not significantly so.

---

# Types of Caches

Beyond the levels of cache in the memory hierarchy, there are many other types of caches

- Split cache – one cache for code and one cache for data
  - One cache feeds instruction decoder and one feeds data to execution unit
  - Supports pipelined architectures and other mechanisms intended to improve performance
  - Instruction cache doesn't need to worry about writes
  - Data cache doesn't need to worry about branch history or decoding information
- Victim cache – holds blocks evicted from cache due to replacement algorithm
- Trace cache – holds instructions already decoded from a previous execution
- Loop buffer – a small trace cache used to speed up the execution of loops

---

# Problems Writing to a Cache

***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, i.e., DMA
- Multiple CPUs may have individual caches; once one cache is written to, all caches are invalid

# Solutions for Writing to a Cache

***Solution 1 – 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

***Solution 2 – 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

***Multiple Processor Solutions:***

- Bus watching with write through – each cache watches the bus to see if data they contain is being written to the main memory by another processor. All processors must be using the write through policy
- Hardware transparency – a "big brother" watches all caches, and upon seeing an update to any processor's cache, it updates main memory AND all of the caches
- Noncacheable memory – Any shared memory (identified with a chip select) may not be cached.

---

# Intel Caches

| Processor | Cache Size | Other Info |
|---|---|---|
| 386DX | L2 (external): 64 KB to 128 KB | write-through |
| 486DX | L1 (internal): 8 KB<br>L2 (external): 128 KB to 256 KB | write-back |
| Pentium | L1 (internal): 8 + 8KB (instr + data)<br>L2 (external): 256 KB to 512 K | write-back |
| Pentium Pro | L1 (internal): 8 + 8KB (instr + data)<br>L2 (internal): 256 KB | write-back |
| Pentium III | L1 (internal): 16 + 16KB (instr + data)<br>L2 (internal): 256 to 512 KB | write-back |
| Intel Core 2 | L1 (internal): 64 KB each processor<br>L2 (internal): 2 MB to 4 MB shared | write-through L1<br>write-back L2 |

- Fetch/Decode Unit
  - Fetches instructions from L2 cache
  - Decode into micro-ops
  - Store micro-ops in L1 cache
- Out of order execution logic
  - Schedules micro-ops based on data dependence and resources
  - May speculatively execute
- Split cache creates performance improvement by separating decoding from scheduling