Caches load multiple bytes per block to take advantage of spatial locality.

If cache block size = $2^b$ bytes, conceptually split memory into $2^b$-byte chunks:

- block addresses = address / $2^b$
- = (address >> $b$)

$n$-bit Address

<table>
<thead>
<tr>
<th>(n-m-b) bits</th>
<th>m bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>

$b$-bit Block Offset
How big is the cache?

Byte-addressable machine, 16-bit addresses, cache details:

- direct-mapped
- block size = one byte
- index = 5 least significant bits

Two questions:

- How many blocks does the cache hold?

- How many bits of storage are required to build the cache (data plus all overhead including tags, etc.)?
Disadvantage of direct mapping

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.

- However, this isn’t really flexible. If a program uses addresses 10, 110, 10, 110, ... then each access will result in a cache miss and a load into cache block 10.

- This cache has four blocks, but direct mapping might not let us use all of them.

- This can result in more misses than we might like.
A fully associative cache

- A fully associative cache permits data to be stored in *any* cache block, instead of forcing each memory address into one particular block.
  - when data is fetched from memory, it can be placed in *any* unused block of the cache.
  - this way we’ll never have a conflict between two or more memory addresses which map to a single cache block.

- In the previous example, we might put memory address 10 in cache block 10, and address 110 in block 11. Then subsequent repeated accesses to 10 and 110 would all be hits instead of misses.

- If all the blocks are already in use, it’s usually best to replace the *least recently used* one, assuming that if it hasn’t used it in a while, it won’t be needed again anytime soon.
The price of full associativity

- However, a fully associative cache is expensive to implement.
  - Because there is no index field in the address anymore, the entire address must be used as the tag, increasing the total cache size.
  - Data could be anywhere in the cache, so we must check the tag of every cache block. That’s a lot of comparators!
Set associativity

- An intermediate possibility is a set-associative cache
  - The cache is divided into *groups* of blocks, called *sets*
  - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set
- If each set has $2^x$ blocks, the cache is an $2^x$-way associative cache
- Here are several possible organizations of an eight-block cache

![Diagram of cache organizations](image)
Writing to a cache

- Writing to a cache raises several additional issues
- First, let’s assume that the address we want to write to is already loaded in the cache. We’ll assume a simple direct-mapped cache:

```
<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>11010</td>
<td>42803</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- If we write a new value to that address, we can store the new data in the cache, and avoid an expensive main memory access [but inconsistent]

```
Mem[1101 0110] = 21763
```

HUGE problem in multiprocessors
Write-through caches

- A **write-through cache** solves the inconsistency problem by forcing all writes to update both the cache and the main memory.

  - This is simple to implement and keeps the cache and memory consistent.

- Why is this not so good?
Write-back caches

- In a write-back cache, the memory is not updated until the cache block needs to be replaced (e.g., when loading data into a full cache set)
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory as shown before
  - The cache block is marked “dirty” to indicate this inconsistency

```
Mem[1101 0110] = 21763
```

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>1101</td>
<td>21763</td>
<td>1101 0110</td>
<td>42803</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data
**Finishing the write back**

- We don’t need to store the new value back to main memory unless the cache block gets replaced
- E.g. on a read from Mem[1000 1110], which maps to the same cache block, the modified cache contents will first be written to main memory

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
<td>1000 1110</td>
<td>1225</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1101 0110</td>
<td>21763</td>
</tr>
</tbody>
</table>

- Only then can the cache block be replaced with data from address 142

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
<td>10001</td>
<td>1225</td>
<td>1000 1110</td>
<td>1225</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1101 0110</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A second scenario is if we try to write to an address that is not already contained in the cache; this is called a write miss.

Let’s say we want to store 21763 into Mem[1101 0110] but we find that address is not currently in the cache.

When we update Mem[1101 0110], should we also load it into the cache?
Write around caches (a.k.a. write-no-allocate)

- With a **write around** policy, the write operation goes directly to main memory *without* affecting the cache.

  ```
  for (int i = 0; i < SIZE; i++)
    a[i] = i;
  ```

- This is good when data is written but not immediately used again, in which case there’s no point to load it into the cache yet.

  $\text{Mem}[1101\ 0110] = 21763$
Allocate on write

- An **allocate on write** strategy would instead load the newly written data into the cache.

\[ \text{Mem}[214] = 21763 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
<td>1101010</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If that data is needed again soon, it will be available in the cache.
Given the following trace of accesses, can you determine whether the cache is **write-allocate** or **write-no-allocate**?

- Assume A and B are distinct, and can be in the cache simultaneously.

<table>
<thead>
<tr>
<th>Access</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss</td>
<td>Load A</td>
</tr>
<tr>
<td>Miss</td>
<td>Store B</td>
</tr>
<tr>
<td>Hit</td>
<td>Store A</td>
</tr>
<tr>
<td>Hit</td>
<td>Load A</td>
</tr>
<tr>
<td>Miss</td>
<td>Load B</td>
</tr>
<tr>
<td>Hit</td>
<td>Load B</td>
</tr>
<tr>
<td>Hit</td>
<td>Load A</td>
</tr>
</tbody>
</table>

Answer: Write-no-allocate

On a write-allocate cache this would be a hit
Memory System Performance

- To examine the performance of a memory system, we need to focus on a couple of important factors.
  - How long does it take to send data from the cache to the CPU?
  - How long does it take to copy data from memory into the cache?
  - How often do we have to access main memory?
- There are names for all of these variables.
  - The **hit time** is how long it takes data to be sent from the cache to the processor. This is usually fast, on the order of 1-3 clock cycles.
  - The **miss penalty** is the time to copy data from main memory to the cache. This often requires dozens of clock cycles (at least).
  - The **miss rate** is the percentage of misses.
Average memory access time

- The **average memory access time**, or **AMAT**, can then be computed

\[
AMAT = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})
\]

This is just averaging the amount of time for cache hits and the amount of time for cache misses

- How can we improve the average memory access time of a system?
  - Obviously, a lower AMAT is better
  - Miss penalties are usually much greater than hit times, so the best way to lower AMAT is to reduce the **miss penalty** or the **miss rate**

- However, AMAT should only be used as a general guideline. Remember that **execution time** is still the best performance metric.
Performance example

- Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles:

- To make AMAT smaller, we can decrease the miss rate
  - e.g. make the cache larger, add more associativity
  - but larger/more complex ⇒ longer hit time!

- Alternate approach: decrease the miss penalty
  - BIG idea: a big, slow cache is still faster than RAM!

- Modern processors have at least two cache levels
  - too many levels introduces other problems (keeping data consistent, communicating across levels)
Opteron Vital Statistics

- **L1 Caches: Instruction & Data**
  - 64 kB
  - 64 byte blocks
  - 2-way set associative
  - 2 cycle access time

- **L2 Cache:**
  - 1 MB
  - 64 byte blocks
  - 4-way set associative
  - 16 cycle access time (total, not just miss penalty)

- **Memory**
  - 200+ cycle access time
Associativity tradeoffs and miss rates

- Higher associativity means more complex hardware
- But a highly-associative cache will also exhibit a lower miss rate
  - Each set has more blocks, so there’s less chance of a conflict between two addresses which both belong in the same set
  - Overall, this will reduce AMAT and memory stall cycles
- Figure from the textbook shows the miss rates decreasing as the associativity increases
Cache size and miss rates

- The cache size also has a significant impact on performance
  - The larger a cache is, the less chance there will be of a conflict
  - Again this means the miss rate decreases, so the AMAT and number of memory stall cycles also decrease

- Miss rate as a function of both the cache size and its associativity
Finally, miss rates relative to the block size and overall cache size

- Smaller blocks do not take maximum advantage of spatial locality
- But if blocks are too large, there will be fewer blocks available, and more potential misses due to conflicts