Lectures 17-18

- Today:
  - More caches

---

How will execution time grow with SIZE?

```java
int array[SIZE];
int A = 0;
for (int i = 0 ; i < 200000 ; ++ i) {
    for (int j = 0 ; j < SIZE ; ++ j) {
        A += array[j];
    }
}
```

---

How can we find data in the cache?

- The second question was how to determine whether or not the data we're interested in is already stored in the cache.
- If we want to read memory address i, we can use the mod trick to determine which cache block would contain i.
- But other addresses might also map to the same cache block. How can we distinguish between them?
- For instance, cache block 2 could contain data from addresses 2, 10, 14,

---

Actual Data

---

Adding tags

- We need to add tags to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.

---

One more detail: the valid bit

- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a valid bit for each cache block.
- When the system is initialized, all the valid bits are set to 0.
- When data is loaded into a particular cache block, the corresponding valid bit is set to 1.

---

So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity.
What happens on a cache hit
- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest \( k \) bits of the address will index a block in the cache.
  - If the block is valid and the tag matches the upper \( (m - k) \) bits of the m-bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a 2\(^{14}\) byte cache.

What if the cache fills up?
- Our third question was what to do if we run out of space in our cache, or if we need to reuse a block for a different memory address.
- We answered this question implicitly on the last page:
  - A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data.
  - This is a Least Recently Used replacement policy, which assumes that older data is less likely to be requested than newer data.
- We'll see a few other policies next.

Loading a block into the cache
- After data is read from main memory, putting a copy of that data into the cache is straightforward:
  - The lowest \( k \) bits of the address specify a cache block.
  - The upper \( (m - k) \) address bits are stored in the block's tag field.
  - The data from main memory is stored in the block's data field.
  - The valid bit is set to 1.

How big is the cache?
- Suppose we have a byte-addressable machine with 16-bit addresses with a cache with the following characteristics:
  - It is direct-mapped
  - Each block holds one byte
  - The cache index is the four least significant bits
- Two questions:
  - How many blocks does the cache hold?
    \[ 2^4 \times 16 \]
  - How many bits of storage are required to build the cache (e.g., for the data array, tags, etc.):
    \[ 16 \times (1 + 2 + 3) = 16 \times 6 = 96 \text{ bits} \]

More cache organizations
- Now we'll explore some alternate cache organizations:
  - It is LRU
  - How do we reduce the number of potential conflicts?
- We'll first motivate it with a brief discussion about cache performance.
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 store data in the cache?
- How long does it take to fetch data from the cache?
- How often does it take to fetch data from memory?

- How often do we have to access main memory?
- There are many for all of these variables.
  - How often is it necessary to be saved to cache?
  - How often is the cache miss rate?
  - The miss rate is the percentage of misses.

Performance example

- Assume that 30% of the instructions in a program are data accesses. The cache hit ratio is 90% and the hit time is one cycle, but the miss penalty is 20 cycles.

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

- How can we reduce miss rate?

Spatial locality

- One-byte cache blocks don’t take advantage of spatial locality, which predicts that an access to one address will be followed by an access to a nearby address.
- What can we do?

Spatial locality

- What we can do is make the cache block size larger than one byte.

Average memory access time

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

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

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.
- How can we reduce miss rate?
  - Larger cache
  - Multilevel cache
  - Larger block size
  - Prefetching
**Block addresses**

- How can we figure out where data should be placed in the cache?
- It's time for block addresses! If the cache block size is 2^i bytes, we can conceptually split the main memory into 2^i-byte chunks too.
- To determine the block address of a byte address, you can do the integer division

\[
\text{block address} = \left\lfloor \frac{\text{byte address}}{2^i} \right\rfloor
\]

- Our example has two-byte cache blocks, so we can think of a 16-byte main memory as an "8-block" main memory instead.
- For instance, memory addresses 12 and 13 both correspond to block address 5, since

\[
12 / 2 = 6 \quad \text{and} \quad 13 / 2 = 6
\]

**Data placement within a block**

- When we access one byte of data in memory, we’ll copy its entire block into the cache, to hopefully take advantage of spatial locality.
- In our example, if a program reads from byte address 12 we will load all of memory block 6 (both addresses 12 and 13) into cache block 2.
- Note byte address 13 corresponds to the same memory block address 5. So a read from address 13 will also cause memory block 6 (addresses 12 and 13) to be loaded into cache block 2.
- To make things simpler, byte i of a memory block is always stored in byte i of the corresponding cache block.

**Cache mapping**

- Once you know the block address, you can map it to the cache as before: find the remainder when the block address is divided by the number of cache blocks.
- In our example, memory block 1 belongs in cache block 1, since

\[
6 \mod 4 = 2
\]
- This corresponds to placing data from memory byte addresses 12 and 13 into cache block 2.

**Locating data in the cache**

- Let's say we have a cache with 2^i blocks, each containing 2^k bytes.
- We can determine where a byte of data belongs in this cache by looking at its address in main memory.
- The lowest i bits of the address will select one of the 2^i cache blocks.
- The lowest k bits are now a block offset that determines which of the 2^k bytes in the cache block will store the data.

**A picture**

![A picture of cache memory](image)

**An exercise**

For the address below, what byte is read from the cache (or is there a miss)?

- Address: 1030
- Address: 1110
- Address: 0001
- Address: 1101

![An exercise with cache memory](image)
A larger example cache mapping

- Where would the byte from memory address 6146 be stored in this direct-mapped 512-byte block cache with 32-byte blocks?
- We can determine this with the binary force.
  - 6146 in binary is 00...01 1000 0000 00 00.
  - The lowest 5 bits, 10000, mean this is the second byte in its block.
  - The next 10 bits, 1000000000, are the block number itself (512).
- Equivalently, you could use arithmetic instead.
  - The block offset is 6146 mod 4, which equals 2.
  - The block address is 6146/4 = 1536, so the index is 1536 mod 1024, or 512.

What goes in the rest of that cache block?

- The other three bytes of that cache block come from the same memory block, whose addresses must all have the same index (1000000000) and the same tag (000000).

The rest of that cache block

- Again, byte i of a memory block is stored into byte i of the corresponding cache block.
  - In our example, memory block 1536 consists of byte addresses 6144 to 6147. So bytes 0-3 of the cache block would contain data from address 6144, 6145, 6146 and 6147 respectively.
  - You can also look at the lowest 2 bits of the memory address to find the block offsets.
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.
- But, what happens if a program uses addresses 1, 6, 2, 6, 1, ...?

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.
  - In the previous example, we might put memory address 2 in cache block 1, and address 6 in block 3. Then subsequent repeated accesses to 2 and 6 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 been used in a while, it won’t be needed again anytime soon.

Locating a set associative block

- We can determine where a memory address belongs in an associative cache in a similar way as before.
  - If a cache has 2^s sets and each block has 2^t bytes, the memory address can be partitioned as follows:
    - Address (in bits): 0s-0t, 1s-1t, 2s-2t, 3s-3t, 4s-4t...n-1s-n-1t
    - Tag: 0-0t
    - Index: 1s-1t
    - Block Offset: 2s-2t

- Our arithmetic computations now compute a set index, to select a set within the cache instead of an individual block.

Block Offset = Memory Address mod 2^s
Block Address = Memory Address / 2^s
Set Index = Block Address mod 2^t
Example placement in set-associative caches

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache design below, with 16 bytes per block?
- 6195 in binary is 00...0110000 01110015.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.
- For the 1-way cache, the next three bits (31-3) are the set index.
- For the 2-way cache, the next two bits (11) are the set index.
- For the 4-way cache, the next one bit (1) is the set index.
- The data may go in any block, shown in green, within the correct set.

<table>
<thead>
<tr>
<th>1-way associativity</th>
<th>2-way associativity</th>
<th>4-way associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
</tr>
</tbody>
</table>

Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before.
- For highly associative caches, it’s expensive to keep track of what’s really the least recently used block, so some approximations are used. We won’t get into the details.

<table>
<thead>
<tr>
<th>1-way associativity</th>
<th>2-way associativity</th>
<th>4-way associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
</tr>
</tbody>
</table>

LRU example

- Assume a fully-associative cache with two blocks, which of the following memory references miss in the cache.
- Assume distinct addresses go to distinct blocks.

addresses: A, B, C, D

<table>
<thead>
<tr>
<th>addresses</th>
<th>0 Tags 1 LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>- - 0</td>
</tr>
<tr>
<td>B</td>
<td>- - 1</td>
</tr>
<tr>
<td>C</td>
<td>- - 1</td>
</tr>
<tr>
<td>D</td>
<td>- - 0</td>
</tr>
</tbody>
</table>

Set associative caches are a general idea

- By now you may have noticed the 1-way set associative cache is the same as a direct-mapped cache.
- Similarly, if a cache has 2 blocks, a 2-way set associative cache would be the same as a fully-associative cache.

<table>
<thead>
<tr>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
</tr>
</tbody>
</table>

direct mapped | fully associative

2-way set associative cache implementation

- How does an implementation of a 2-way cache compare with that of a fully-associative cache?

- Only two comparators are needed.
- The cache tags are a little shorter too.
Summary

- Larger block sizes can take advantage of spatial locality by loading data from not just one address, but also nearby addresses, into the cache.
- Associative caches assign each memory address to a particular set within the cache, but not to any specific block within that set.
  - Set sizes range from 1 (direct-mapped) to 2^n (fully associative).
  - Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost.
  - In practice, 2-way through 16-way set-associative caches strike a good balance between lower miss rates and higher costs.
- Next, we’ll talk more about measuring cache performance, and also discuss the issue of writing data to a cache.