Cache writes and examples

- Today is the last day of caches!
  - One important topic we haven't mentioned yet is *writing* to a cache. We'll see several different scenarios and approaches.
  - To introduce some more advanced cache designs, we'll also present some of the caching strategies used in modern desktop processors like Pentiums and PowerPCs.
- On Wednesday we'll turn our attention to peripheral devices and I/O.

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.

Index	V	Tag	Data
…			
110	1	11010	42803
…			

Address	Data
…	
1101 0110	42803
…	

- 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.

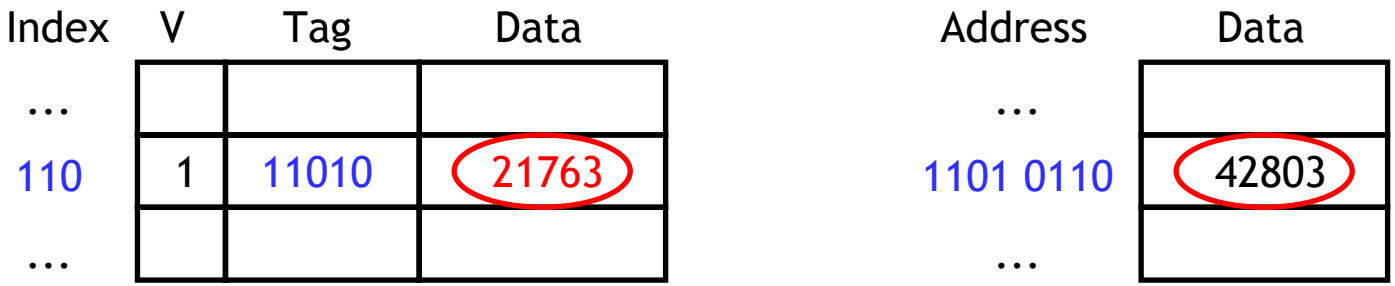
Mem[214] = 21763

Index	V	Tag	Data
…			
110	1	11010	21763
…			

Address	Data
…	
1101 0110	42803
…	

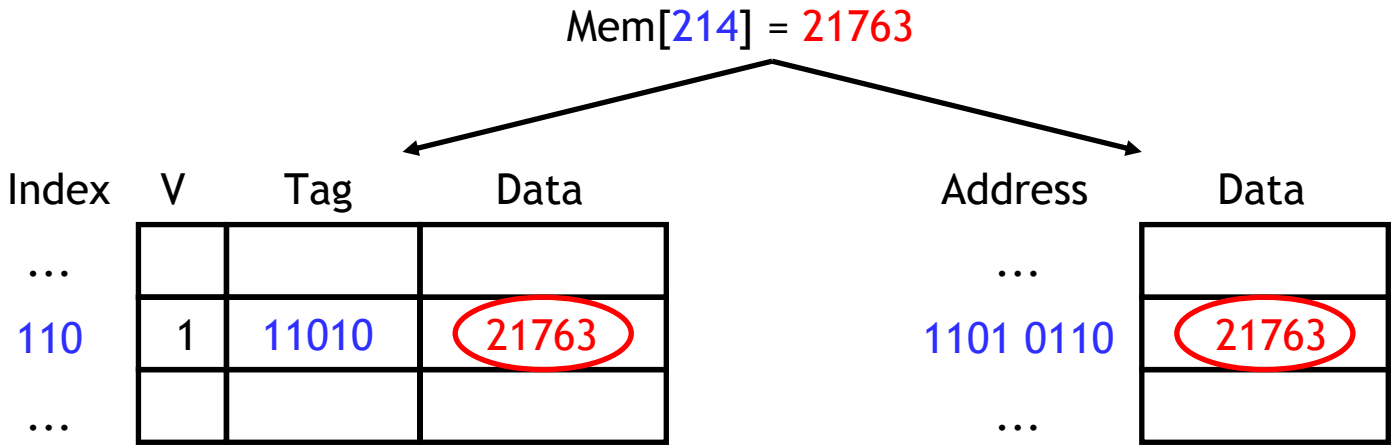
Inconsistent memory

- But now the cache and memory contain different, inconsistent data!
- How can we ensure that subsequent loads will return the right value?
- This is also problematic if other devices are sharing the main memory, as in a multiprocessor system.



Write-through caches

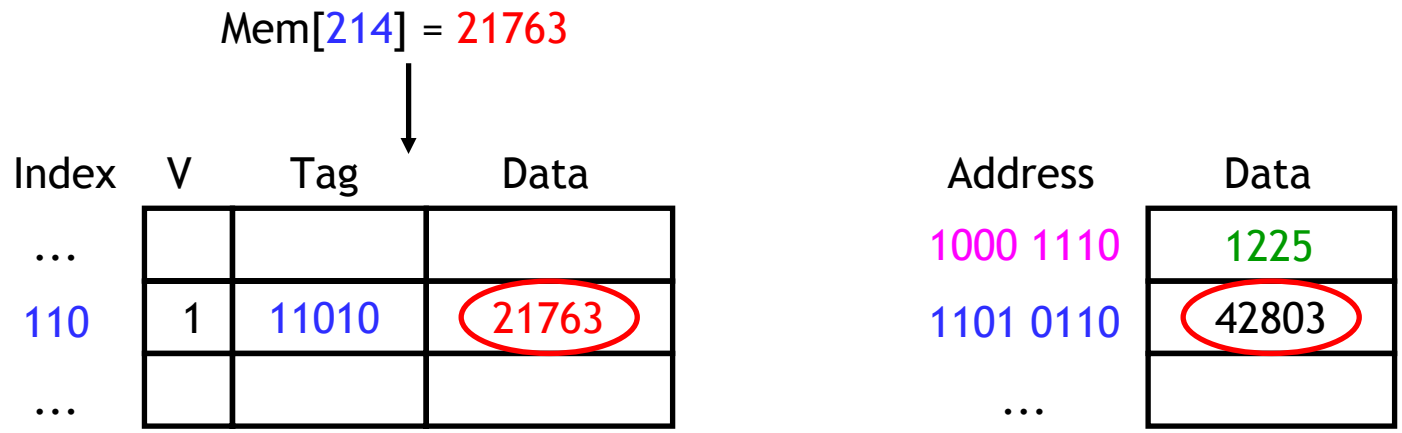
- A write-through cache solves the inconsistency problem by forcing all writes to update both the cache *and* the main memory.



- This is simple to implement and keeps the cache and memory consistent.
- The bad thing is that forcing every write to go to main memory negates the advantage of having a cache—the whole point was to avoid accessing main memory!

Write-back caches

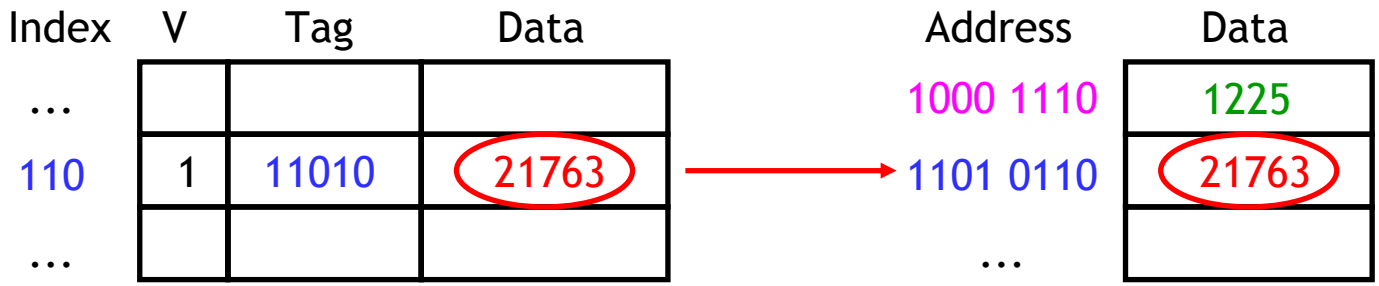
- In a write-back cache, the memory is not updated until the cache block needs to be replaced (e.g., when loading data into a full cache set).
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory as shown before.



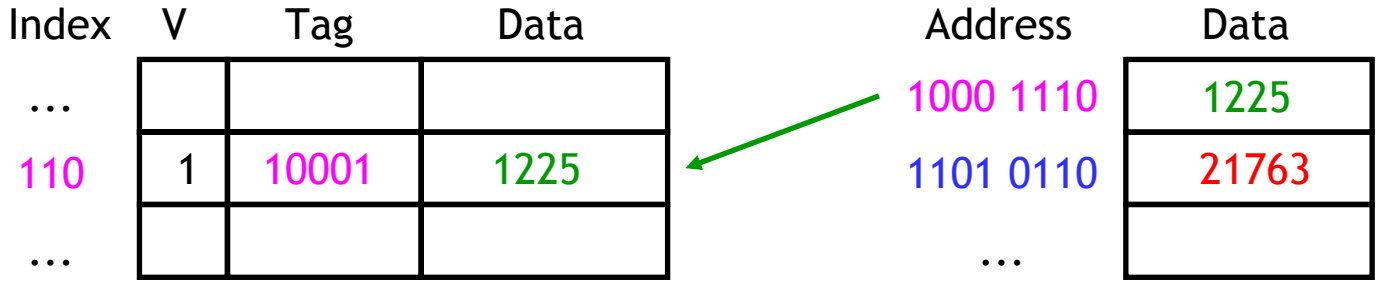
- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data.

Finishing the write back

- We don't need to store the new value back to main memory unless the cache block gets replaced.
- For example, on a read from Mem[142], which maps to the same cache block, the modified cache contents will first be written to main memory.



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



Write-back cache discussion

- Each block in a write-back cache needs a dirty bit to indicate whether or not it must be saved to main memory before being replaced—otherwise we might perform unnecessary writebacks.
- Notice the penalty for the main memory access will not be applied until the execution of some *subsequent* instruction following the write.
  - In our example, the write to Mem[214] affected only the cache.
  - But the load from Mem[142] resulted in *two* memory accesses: one to save data to address 214, and one to load data from address 142.
  - This makes it harder to predict execution time and performance.
- The advantage of write-back caches is that not all write operations need to access main memory, as with write-through caches.
  - If a single address is frequently written to, then it doesn't pay to keep writing that data through to main memory.
  - If several bytes within the same cache block are modified, they will only force one memory write operation at write-back time.

Write misses

- 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.

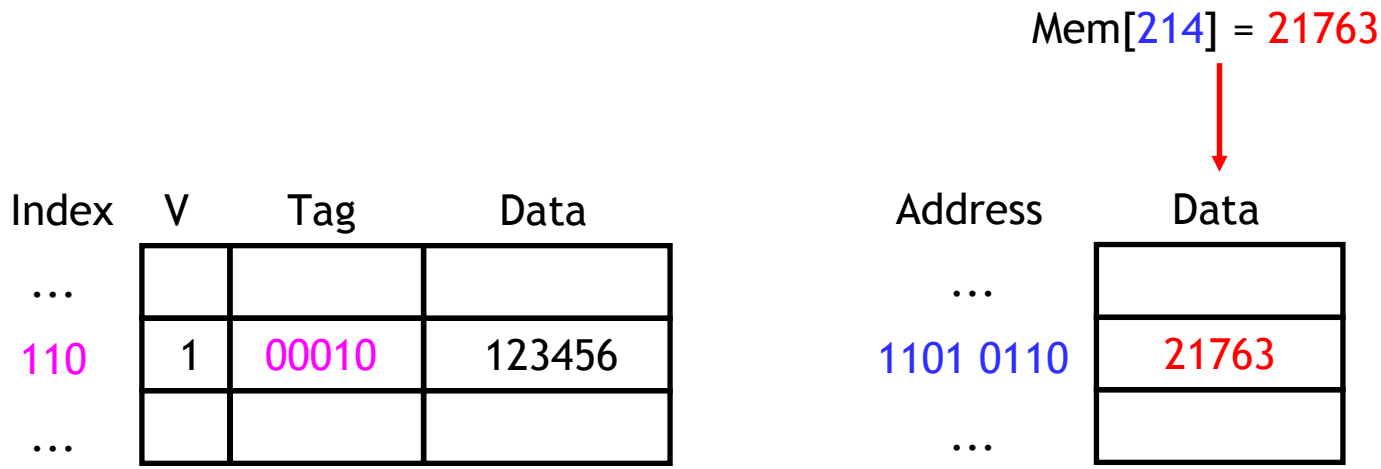
Index	V	Tag	Data
…			
110	1	00010	123456
…			

Address	Data
…	
1101 0110	6378
…	

- When we update Mem[1101 0110], should we *also* load it into the cache?

Write around caches

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

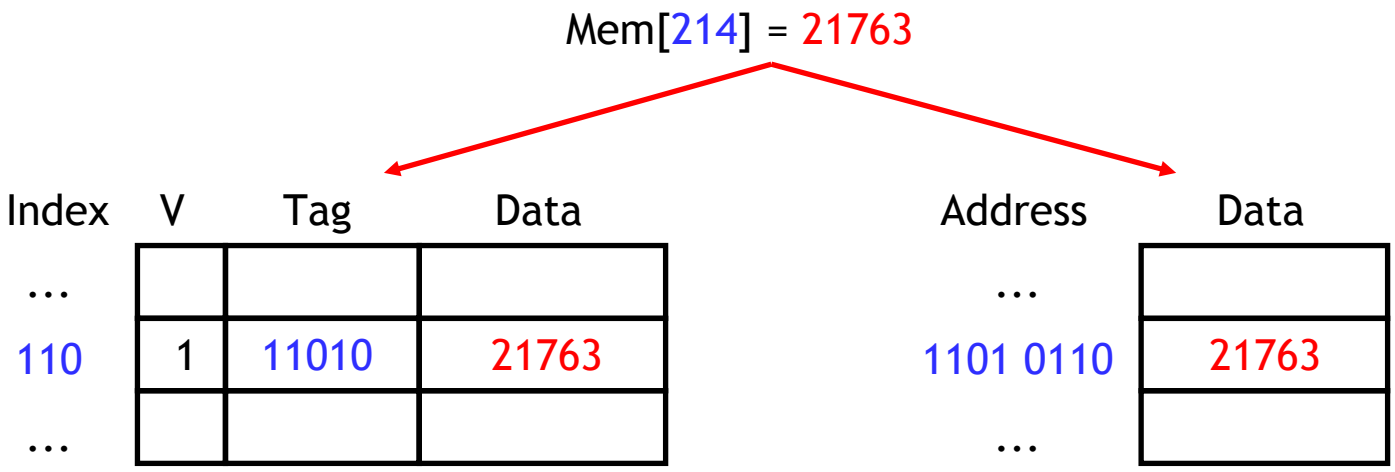


- This is good when data is written but not immediately used again, in which case there's no point to load it into the cache yet.

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

Allocate on write

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



- If that data is needed again soon, it will be available in the cache.

Modern memory systems

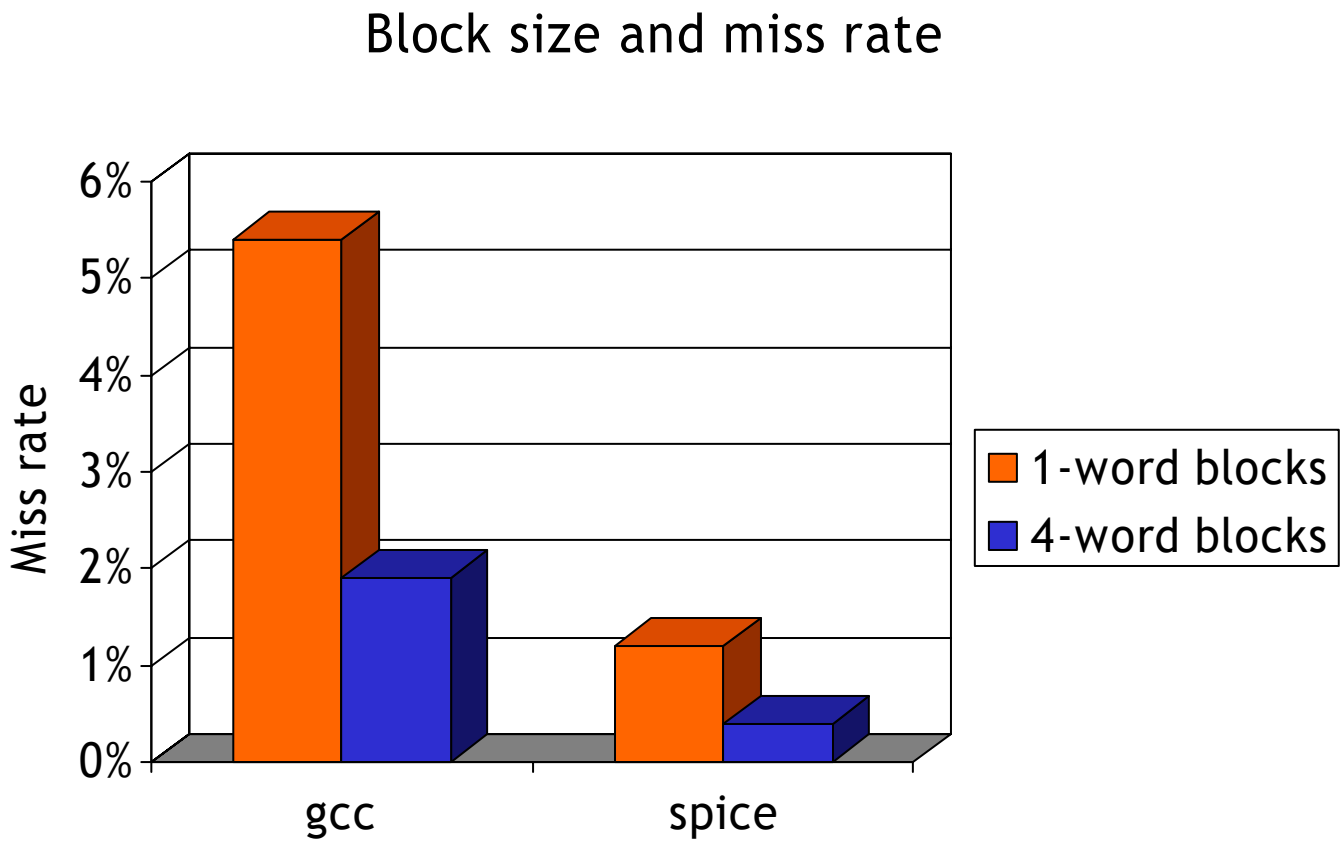
- We'll finish up by describing the memory systems of some real CPUs.
  - This will reinforce some of the ideas we've learned so far.
  - You can also see some more advanced, more modern cache designs.
- The book gives an example of the memory system in the MIPS R2000 CPU, which was popular in the early 90s.
- We'll highlight some of its more interesting features.
  - There are separate caches for instructions and data.
  - Each cache block is just one word wide.
  - A write-through policy is implemented with a write buffer.

Unified vs. split caches

- The R2000 uses a split cache design.
  - There is one 64KB cache for instructions and a *separate* 64KB cache for data.
  - This makes sense when programs need to access both instructions and data in the same clock cycle—we used separate instruction and data memories in our single-cycle and pipelined datapaths too!
- In contrast, a unified cache stores both program code and data.
  - This is more flexible, because a value may be stored anywhere in the cache memory regardless of whether it's an instruction or data.
  - Unified caches normally lead to lower miss rates. As an example, the book shows that the miss rate of a unified cache for gcc is 4.8%, while the miss rate for a split cache turned out to be 5.4%.

Block size

- Each 64KB cache in the R2000 is divided into 16K one-word blocks.
  - This doesn't give the R2000 much chance to take advantage of spatial locality, as we mentioned last week.
  - Figure 7.11 of the book shows how the miss rates could be reduced if the R2000 supported four-word blocks instead.



Write buffers

- The R2000 cache is write-through, so on write hits data goes to both the cache and the main memory.
- This can result in slow writes, so the R2000 includes a write buffer, which queues pending writes to main memory and permits the CPU to continue.

- Buffers are commonly used when two devices run at different speeds.
  - If a producer generates data too quickly for a consumer to handle, the extra data is stored in a buffer and the producer can continue on with other tasks, without waiting for the consumer.
  - Conversely, if the producer slows down, the consumer can continue running at full speed as long as there is excess data in the buffer.
- For us, the producer is the CPU and the consumer is the main memory.

Reducing memory stalls

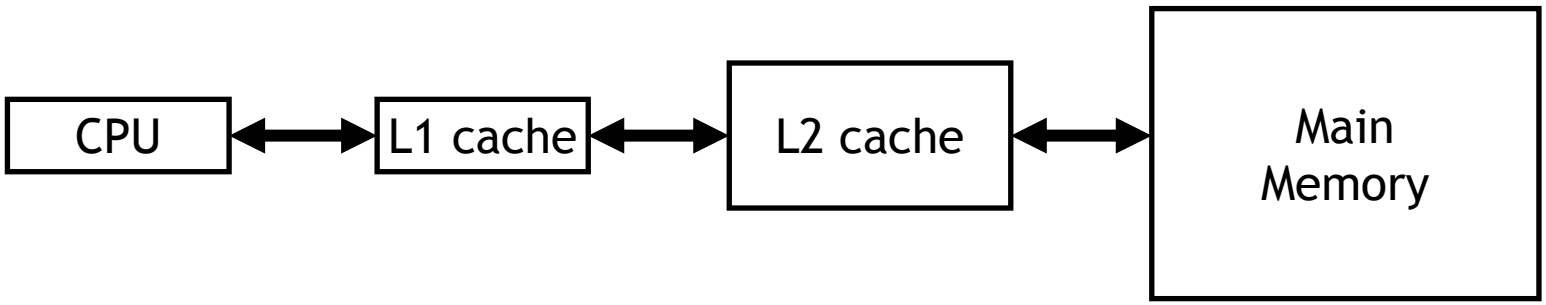
- Most newer CPUs include several features to reduce memory stalls.
  - With a non-blocking cache, a processor that supports out-of-order execution can continue in spite of a cache miss. The cache might process other cache hits, or queue other misses.
  - A multiported cache allows multiple reads or writes per clock cycle, which is necessary for superscalar architectures that execute more than one instruction simultaneously. (A similar concept popped up in Homework 3.)
- A less expensive alternative to multiporting is used by the Pentium Pro.
  - Cache memory is organized into several banks, and multiple accesses to different banks are permitted in the same clock cycle.
  - We saw the same idea last week, in the context of main memory.

Additional levels of caches

- Last week we saw that high miss penalties and average memory access times are partially due to slow main memories.

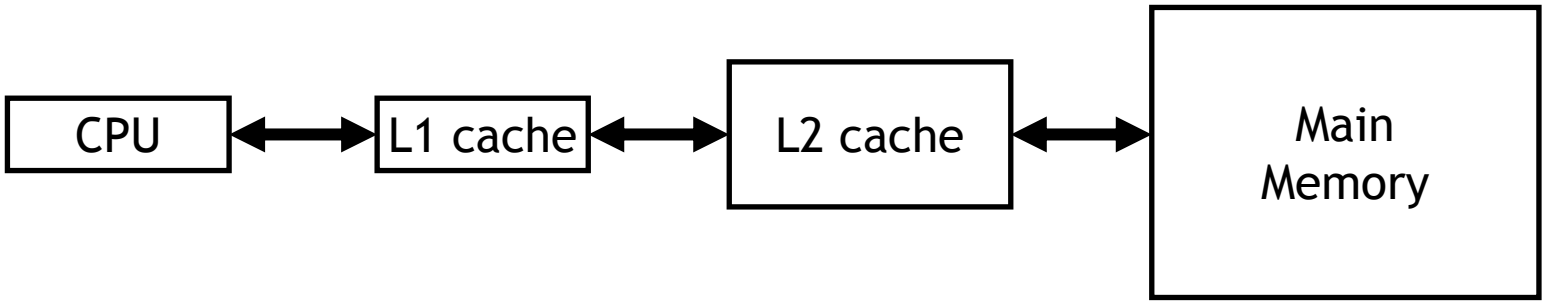
AMAT = Hit time + (Miss rate × Miss penalty)

- How about adding *another* level into the memory hierarchy?



- Most processors include a secondary (L2) cache, which lies between the primary (L1) cache and main memory in location, capacity and speed.

Reducing the miss penalty



- If the primary cache misses, we might be able to find the desired data in the L2 cache instead.
  - If so, the data can be sent from the L2 cache to the CPU faster than it could be from main memory.
  - Main memory is only accessed if the requested data is in *neither* the L1 *nor* the L2 cache.
- The miss penalty is reduced whenever the L2 cache hits.

Multi-level cache design

AMAT = Hit time + (Miss rate × Miss penalty)

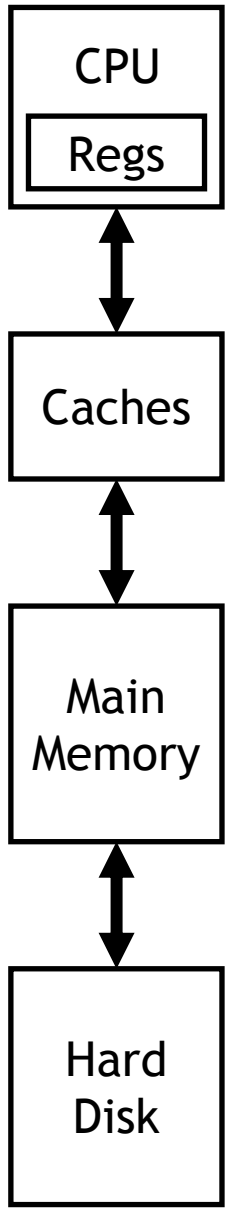
- Adding an L2 cache can lower the miss penalty, which means that the L1 miss rate becomes less of a factor.
- L1 and L2 caches may employ different organizations and policies. For instance, here is some information about the Pentium 4 data caches.

Cache	Data size	Associativity	Block size	Write policy	Latency
L1	8 KB	4-way	64 bytes	write-through	2 cycles
L2	512 KB	8-way	64 bytes	write-back	7 cycles

- The secondary cache is usually much larger than the primary one, so L2 cache blocks are typically bigger as well, and can take more advantage of spatial locality.

Extending the hierarchy even further

- The storage hierarchy can be extended quite a ways.
- CPU registers can be considered as small one-word caches which hold frequently used data.
  - However, registers must be controlled explicitly by the programmer, whereas caching is done automatically by the hardware.
- Main memory can be thought of as a cache for a hard disk.
  - Programs and data are both loaded into main memory before being executed or edited.
- We could go even further, but the picture wouldn't fit.
  - Your hard drive may be a cache for networked information.
  - Companies like Akamai provide mirrors or caches of other web sites.

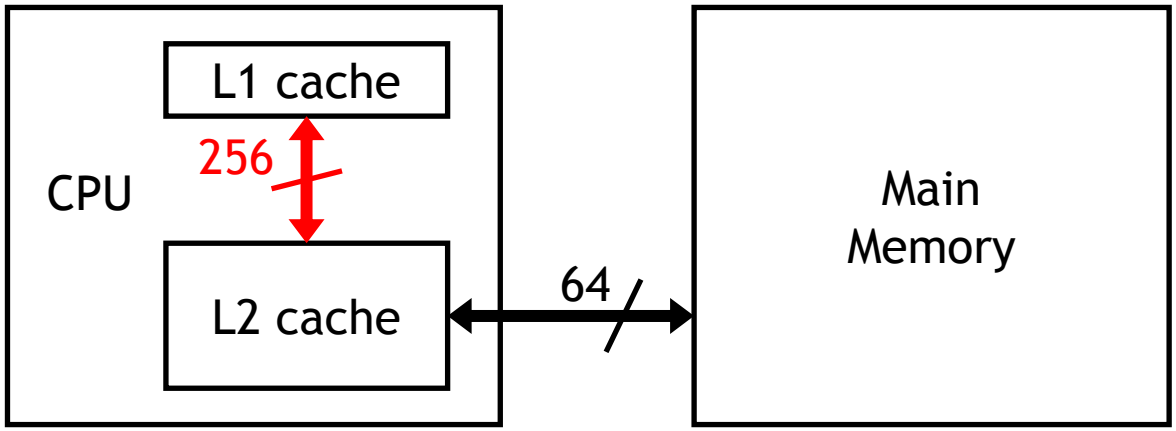


Intel Pentium 4 caches

Cache	Data size	Associativity	Block size	Write policy	Latency
L1	8 KB	4-way	64 bytes	write-through	2 cycles
L2	512 KB	8-way	64 bytes	write-back	7 cycles

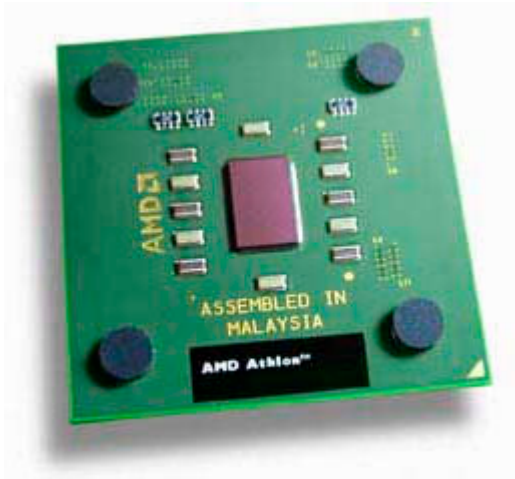
- The Intel Pentium 4 has very small L1 caches.
  - An execution trace cache holds about 12,000 micro-instructions. This avoids the cost of having to re-decode instructions (an expensive task for 8086-based processors) that are read from the cache.
  - The primary data cache stores just 8KB.
- Intel's designers chose to minimize primary cache latency at the cost of a higher miss rate. So level 1 cache hits in the Pentium 4 are very fast, but there are more misses overall.
- The 512KB secondary cache is eight-way associative, with 64-byte blocks.

Pentium 4 memory system



- The Pentium caches are connected by a 256-bit (32-byte) bus, so it takes two clock cycles to transfer a single 64-byte L1 cache block.
- The Pentium caches are inclusive. The L1 caches contain a copy of data which is also in L2, and the L2 cache contains data that's also in RAM.
  - The same data may exist in all three levels of the hierarchy.
  - The effective cache size is only 512KB, since the L1 caches are just a faster copy of data that's also in L2.

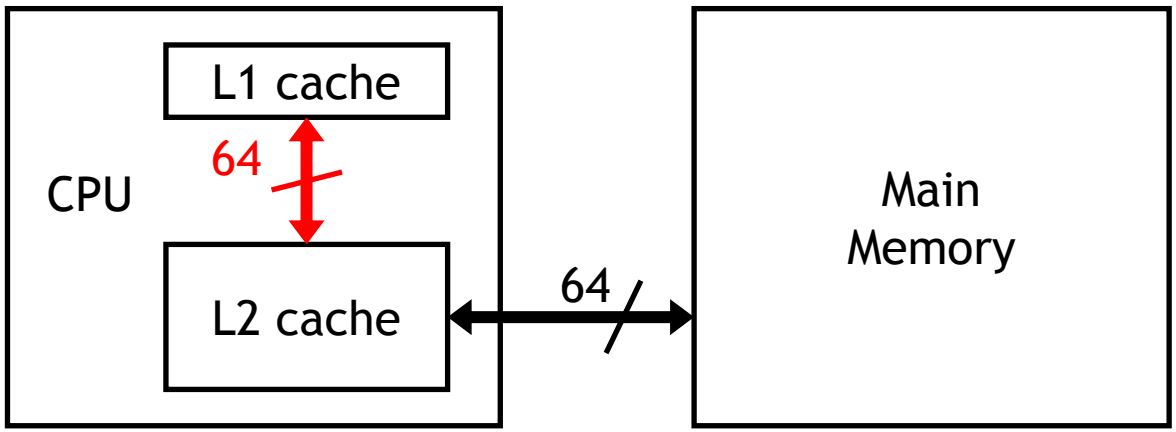
AMD Athlon XP caches



- The AMD Athlon XP is an 8086-compatible processor with several features aimed at lowering miss rates.
- The Athlon has a 128KB split level 1 cache with four-way associativity.
  - This is roughly eight times larger than the Pentium 4's L1 caches.
  - As we saw last week, larger caches usually result in fewer misses.
- The Athlon's 512KB secondary cache also has some notable features.
  - The higher 16-way associativity further minimizes the miss rate.
  - The L1 and L2 caches are exclusive—they never hold the same data. This makes the effective total cache size 640KB.

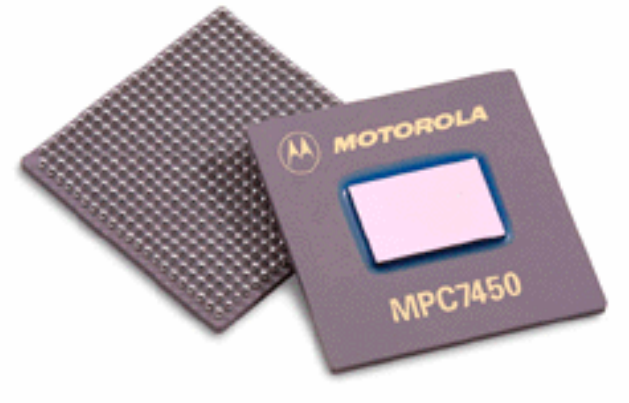
Athlon XP memory system

- The Athlon L1 and L2 caches are connected by a 64-bit bus, as compared to the wider 256-bit bus on the Pentiums.
  - AMD claims their caches have a miss rate that's low enough to offset the larger miss penalty.
  - This illustrates another one of many architectural tradeoffs.



Motorola PowerPC G4

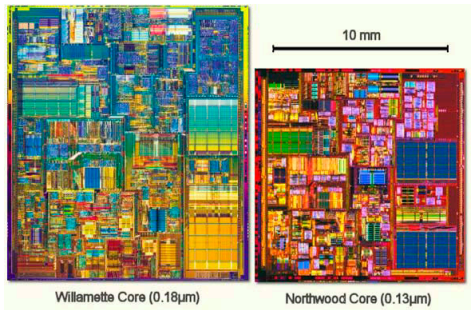
- The Motorola PowerPC G4 has a 64KB, split primary cache with eight-way associativity.
- The on-die 256KB 8-way L2 cache has a 256-bit interface to the L1, just like the Pentium 4.
- The G4 also goes one step further with the memory hierarchy idea, by supporting an external Level 3 or L3 cache up to 2MB.
  - The L3 cache data is not stored on the CPU itself, but the *tags* are. Thus, determining L3 cache hits is very fast.
  - The L3 cache can also be configured as regular main memory, but with the advantage of being much faster.



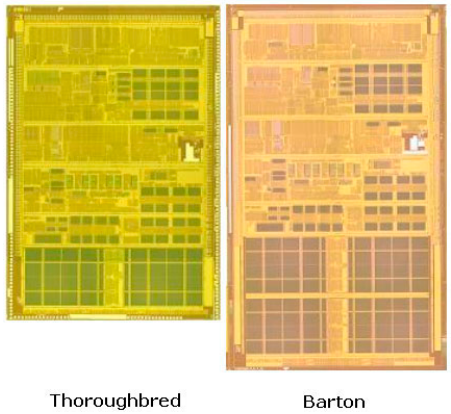
Caches and dies

- As manufacturing technology improves, designers can squeeze more and more cache memory onto a processor die.
- In the old Pentium III days the L2 cache wasn't even on the same chip as the processor itself. Companies sold processor "modules" that were composed of several chips internally.
- The second picture illustrates the size difference between an older Pentium 4 and a newer one. The newer one has an area of just 131 mm^2!
- The last picture shows the dies of an older Athlon with only 256KB of L2 cache, and a newer version with 512KB of L2 cache. You can literally see the doubling of the cache memory.

Tech Report



Tom's Hardware



Hardware.fr

CPU families

- Manufacturers often sell a range of processors based on the same design.

Company	Budget	Standard	High-end
Intel	Celeron	Pentium 4	Xeon MP
AMD	Duron	Athlon XP	Opteron

- In all cases, one of the main differences is the size of the caches.
  - Low-end processors like the Duron and Celeron have only half as much or less L2 cache memory than "standard" versions of the CPU.
  - The high-end Opteron features a 1 MB secondary cache, and Xeons can be outfitted with up to 2 MB of level 3 cache.
- This highlights the importance of caches and memory systems for overall performance.

Summary

- Writing to a cache poses a couple of interesting issues.
  - Write-through and write-back policies keep the cache consistent with main memory in different ways for write hits.
  - Write-around and allocate-on-write are two strategies to handle write misses, differing in whether updated data is loaded into the cache.
- Modern processors include many advanced cache design ideas.
  - Split caches can supply instructions and data in the same cycle.
  - Non-blocking and multiport caches minimize memory stalls.
  - Multilevel cache hierarchies can help reduce miss penalties.
  - The speed and width of memory buses also affects performance.
- Memory systems are a complex topic, and you should take CS333 to learn more about it!