## Introduction (1)

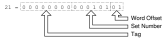
- In the previous lecture, we looked at two basic cache designs:
  - Direct-mapped caches are simple to build but have relatively bad performance.
  - Associative caches improved performance but took a lot of hardware to build.
- The idea here is to examine a trade-off between these two designs, a set-associative cache.
- We also need to examine advanced methods for policy regarding use of the cache:
  - The choice between write-through and write-around.
  - The decision of if write-allocation is performed.
  - Other design issues that require smaller decisions.

## Set-Associative Caches (1)

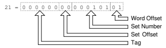
- A set-associative cache combines ideas from previous designs:
  - The address is used to find a single cache set.
  - Each cache set consists of a small number of cache lines.
  - Within each set, searching for an address is fully associative.
- Usually, we say such a cache is *n*-way set-associative.
  - That is, there are *n* places a data item can reside in each set.
- Set-associative caches offer a compromise or trade-off between other designs:
  - The performance is close to an associative cache.
  - The complexity is closer to a direct-mapped cache.

## Set-Associative Caches (2)

- Our address translation might involve another component, the set offset selects which item in the set the address maps onto.
- We could select a random set to put data in or, since the sets are small, use LRU:

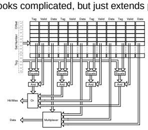


- Or, we could steal some more of the address to determine the set offset deterministically:



## Set-Associative Caches (3)

- The result looks complicated, but just extends previous designs:



- The hardware isn't too expensive since we are dealing with a lower, more constrained level of parallelism.

## Set-Associative Caches (4)

- An an example, consider a cache with 8 lines split into 2 sets where the line size is 1 word and we use LRU to select lines within a set.
  - We feed it an address stream of $1, 6, 34, 23, 9, 34, 6, 1, 9, 41$.

| | | | |
|---|---|---|---|
| 0 | 6,6 | 4 | 1,1 |
| 1 | 34,34 | 5 | 23 |
| 2 | | 6 | 9,9 |
| 3 | | 7 | 41 |

- This generates 6 cache misses and 4 cache hits:
  - We have matched the performance of the fully associative cache from the previous lecture.
  - But our design is much less expensive to build.

## Writing Data (1)

- So far we've ignored what happens when the processor wants to write data into memory: the processor issues the write as usual, but what should the cache do ?
- There are two main problems we need to tackle:
  - If we update the data in the cache, what happens to the data stored in memory ?
  - What happens if the data we want to update isn't even in the cache at all ?
- What ever solutions we decide to use there is one key principle:
  - We should maintain consistency between what is stored in the cache and what it stored in memory.
  - That is, it should be impossible to get confused about the value of data.

## Writing Data (2)

- A write-through cache updates the data in the cache, but also sends the write on to the main memory.
- This is the simplest choice of policy:
  - Since the cache and memory content is always the same, it is always consistent.
  - Need to be careful that we still improve performance of writes.
- However, by ensuring consistency we have caused a lot of unnecessary memory traffic:
  - If we perform a number of writes to the same address, there is no need to keep updating the memory.

## Writing Data (3)

- A write-back caches update the data in the cache, but does not immediately write the data back to main memory.
- The hardware for a write-back cache is more complex:
  - The cache has to remember that lines in the cache are inconsistent with main memory.
  - As well as the valid bit, write-back caches also have a dirty bit for each line.
  - When a block is evicted, the dirty bit is checked to see if it should be written to main memory.
- The write-back scheme removes the unnecessary bus traffic:
  - Now we only write to main memory when we really have to.

## Writing Data (4)

- What happens if the data we want to update is not in the cache ?
  - We can't simply store the new value in the cache.
  - Need to take into account that there may be other data in the same line.
- There are three main choices:
  - Write the data directly to main memory but don't alter any data in the cache.
  - Fetch the data into the cache like a read, before completing the write operation.
  - Allow partially full cache lines so we can write without fetching.

## Design Issues (1)

- Consider the following example situation.
- We have a 1 KB direct mapped cache with $L = 256$ lines and $S = 4$ words per-line.
- The processor issues the cache a load request for address 2.
  - The address 2 maps onto cache line 0.
  - Cache line 0 will hold the data for addresses $0, 1, 2$ and 3.
  - But the cache line doesn't currently hold this data.
- So, the cache needs to load the four addresses $0, 1, 2$ and 3 from main memory to fill line 0.
- The question is which order should we load the addresses ...

## Design Issues (2)

- Critical Word First
  - In this scheme we would load the words in the order $2, 3, 0, 1$.
- Natural Word First
  - In this scheme we would load the words in the order $0, 1, 2, 3$.
- Clearly each has some advantages and disadvantages:
  - Loading the critical word first means the cache can complete the request from the processor faster since it doesn't need to wait for other loads from main memory.
  - Loading the natural word first is much less complex to implement.

## Design Issues (3)

- Often, hardware devices are accessed through the memory system:
  - These are often called memory mapped devices.
  - For example, writing to memory location 40 in main memory might map through and turn on an LED somewhere.
- We need to be careful that our cache is transparent these devices:
  - A load instruction should cause exactly one read from the memory mapped device.
  - A store instruction should cause exactly one write to the memory mapped device.
  - Adjacent locations should not be accessed.
- Often, a cache bypass mode is included so that loads and stores can side-step the cache and go straight to main memory.

## Building Real Caches (1)

| Name | Direct-Mapped | Fully-Associative | Set-/ |
|---|---|---|---|
| Hit-ratio | Lowest | Highest | High |
| Complexity | Lowest | Highest | Med |
| Cost | Lowest | Highest | Med |

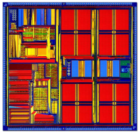
- The choice of cache design depends on a myriad of factors:
  - Cost, power consumption, physical size, instruction mix, expected hit-ratio, memory hierarchy organisation ...
- Usually there is no best solution, just a trade-off:
  - Typically, we might try to maximise hit-ratio per unit of physical space used.

## Building Real Caches (2)

- Where can a line be placed ?
  - One place (direct-mapped), a few places (set-associative), or any place (fully-associative).
- How is a line found ?
  - Direct indexing (direct-mapped), limited search (set-associative), or fully parallel search (fully-associative).
- Which line is replaced when a miss occurs ?
  - Typically, either the least recently used (LRU) or a random choice.
- How are writes handled ?
  - Each level in the hierarchy can use either write-through or write-back
- Which order are items in a line loaded or stored in ?
  - Typically, either critical or natural word first.

## Building Real Caches (3)

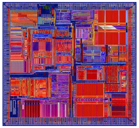
- The MIPS R5000 processor:



- 32 KB L1 instruction cache and 32 KB L1 data cache, both 2-way set-associative.

## Building Real Caches (4)

- The MIPS R10000 processor:



- 32 KB L1 instruction cache and 32 KB L1 data cache, both 2-way set-associative, 512 KB unified, direct-mapped L2 cache.

## Conclusions

- From these processor dies, you can see the caches are a massive proportion of the area !
- This highlights how important they are ... and underlines some of the problems:
  - Need a good compromise between space and performance; set-associative caches give us this.
  - Need a good set of operational policies; various choices, none are the best !
- Once the hardware is in place, we need to write software which is cache conscious.
- That is, although the cache is invisible to the programmer, we need to consider how it will react to our programs.

## Further Reading

- Structured Computer Organisation
  A.S. Tanenbaum.
  Pearson/Prentice-Hall, ISBN: 0-13-148521-0.
  - Chapter 3.3 – Memory
  - Chapter 3.7 – Interfacing
  - Chapter 4.5 – Improving Performance