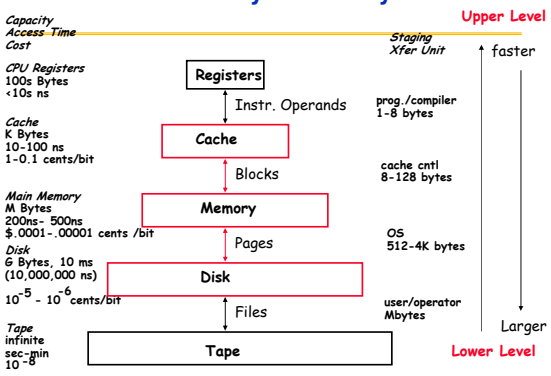
CMSC 411
Computer Systems Architecture
Lecture 14
Memory Hierarchy 1
(Cache Overview)

Levels of the Memory Hierarchy

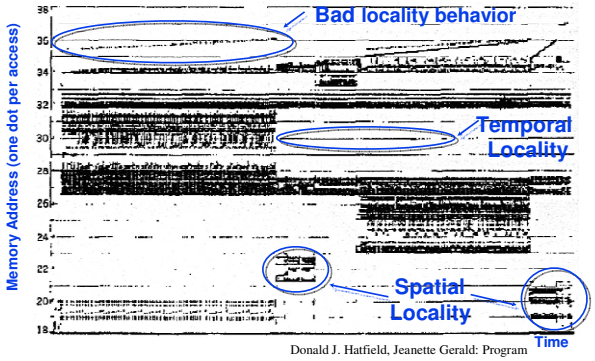


The Principle of Locality

- The Principle of Locality:
  - Program accesses a relatively small portion of the address space at any short period of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Last 15-20 years, HW has relied on locality to improve overall performance

It is a property of programs that is exploited in machine design.

Programs with locality cache well ...



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Issues to consider

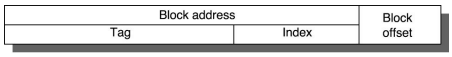
- How big should the fastest memory (cache memory) be?
- How do we decide what to put in cache memory?
- If the cache is full, how do we decide what to remove?
- How do we find something in cache?
- How do we handle writes?

First, there is main memory

- Jargon:
  - frame address – which page?
  - block number – which cache block?
  - contents – the data

Then add a cache

- Jargon: Each address of a memory location is partitioned into
  - block address
    - » tag
    - » index
  - block offset

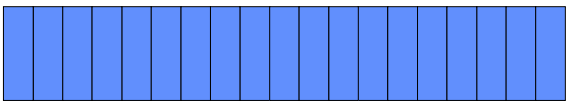


How does cache memory work?

- The following slides discuss:
  - what cache memory is
  - three organizations for cache memory
    - » direct mapped.
    - » set associative
    - » fully associative
  - how the bookkeeping is done
- Important note: All addresses shown are in octal. Addresses in the book are usually decimal.

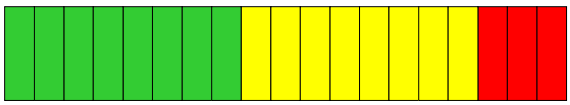
What is cache memory? Main memory first

Main memory is divided into (cache) blocks.
Each block contains many words (32-256 common now).



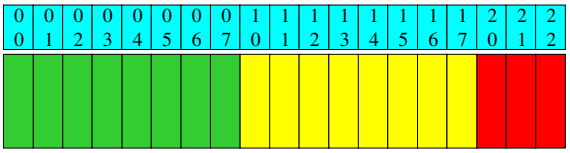
Main memory

Blocks are grouped into frames (pages),
3 frames in this picture.



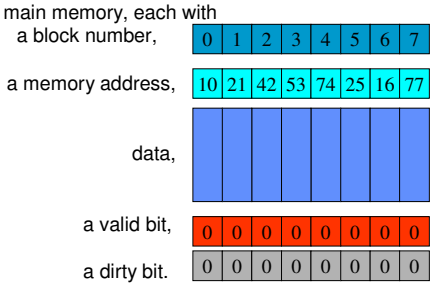
Main memory (cont.)

Blocks are addressed by their frame number,
and their block number within the frame.



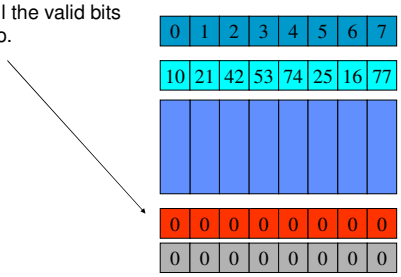
Cache memory

Cache has many, MANY fewer blocks than main memory, each with



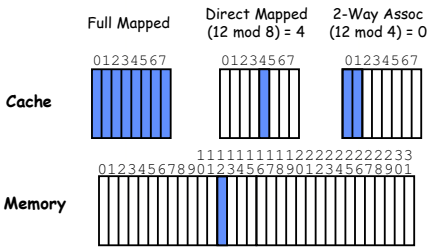
Cache memory (cont.)

Initially, all the valid bits set to zero.



Where can a block be placed?

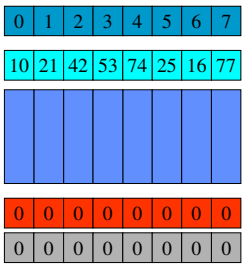
- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative



Cache memory (cont.)

Suppose want to load block 14 (octal) from memory into cache.

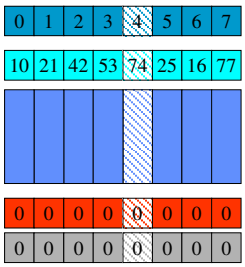
Three ways to organize cache
- direct mapped
- set associative
- fully associative



Direct mapped cache

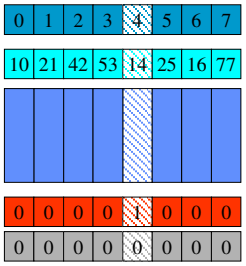
In **direct mapped cache,** block 14 can only be put in the cache block with address 4.

So the cache will no longer hold the block with memory address 74.



Direct mapped cache (cont.)

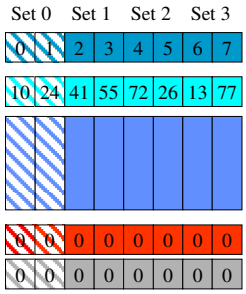
After the load, the contents look like this.



Set associative cache

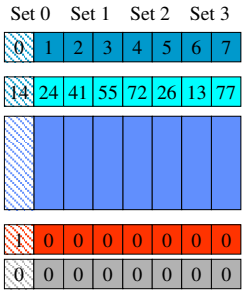
In **set associative cache**, each memory block can be put in any of a set of possible blocks in cache.

For example, if divide cache into 4 sets, block 14 can be put in any block in Set 0 (since last two bits of 14 octal are zero).



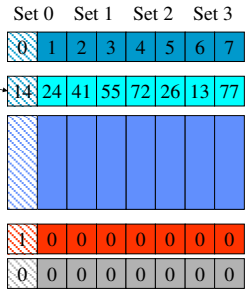
Set associative cache (cont.)

So after loading the block, cache memory might look like this.



Set associative cache (cont.)

Note that the last two bits of the memory block's address always match the set number, so do not need to be stored. This part of the address is called the **index**. The higher order bits are stored, and are called the **tag**.
In these pictures, both index and tag shown.



Recall:

Set associative cache replacement

- Which entry in the set to replace?
- Three common choices:
  - Replace an eligible random block
  - Replace the least recently used (LRU) block
    - » can be hard to keep track of, so often only approximated
  - Replace the oldest eligible block
    - » First In, First Out, or FIFO

Data cache replacement example

SPEC2000, in misses per 1000 instructions

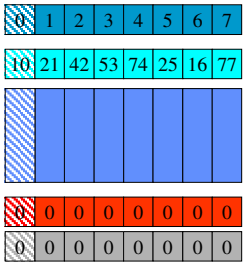
Set associativity

	Two-way			Four-way			Eight-Way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Fully associative cache

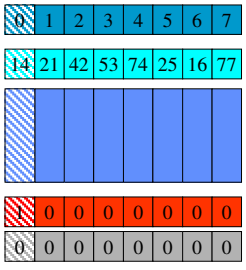
In **fully associative** cache, memory blocks may be stored anywhere.

So block 14 might be put in the first available block -- one with **valid** = 0.



Fully associative cache (cont.)

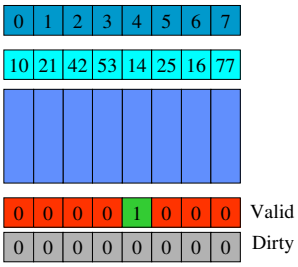
With this result.



Managing cache

Use direct mapped cache as an example.

After first read operation, cache memory looked like this.

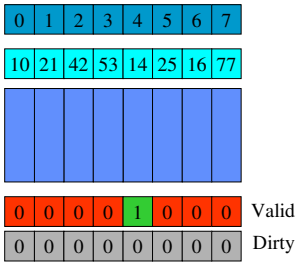


Managing cache (cont.)

If all other memory references involved block 14, no other blocks would need to be fetched from memory.

But suppose eventually need to fetch blocks 10, 31 and 66.

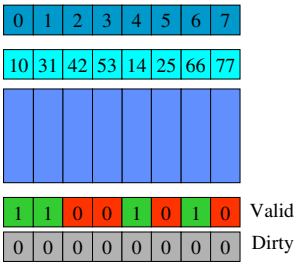
Need to fetch all three, because don't have valid versions of them.



Managing cache (cont.)

The result looks like this.

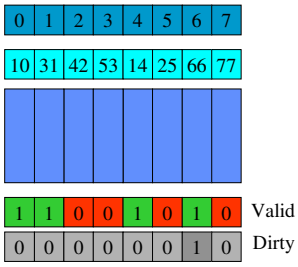
Now suppose **write** to block 66.



Managing cache (cont.)

The block is valid in cache, so don't need to fetch.

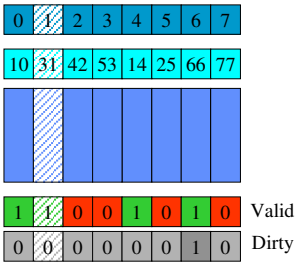
But the **write** operation sets the **dirty** bit for that block.



Managing cache (cont.)

Now suppose need to **read** from a block not in cache.

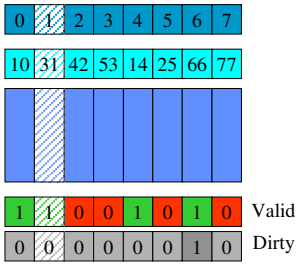
If it is block 41, then must overwrite block 31.



Write-through cache

In **write-through** caches, every **write** causes an *immediate* change both to cache and to main memory.

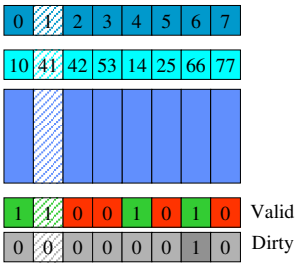
So the **read** just involves fetching the block.



Write-back cache

In **write-back** caches, every **write** causes a change only to cache.

So the **read** involves writing block 31 back to memory if its **dirty** bit is set, then fetching block 41.



Reads easy, writes are not

- Most memory access is read, not write, because read both data and instructions but write only data
- If data requested is not in cache, called a cache miss
- It's easy to make most reads from cache fast
  - Pull the data into a register as soon as it is accessed, while checking whether the address matches the tag
  - If address does not match tag, that is a cache miss, so load a block from main memory to cache, then into the register again
- Can't do this with write:
  - Must verify the address before changing the value of the cache location

Write through vs. write back

- Which is better?
  - Write back gives faster writes, since don't have to wait for main memory
  - Write back is very efficient if want to modify many bytes in a given block
  - But write back can slow down some reads, since a cache miss might cause a write back
  - In multiprocessors or multicore machines, write through might be the only correct solution
    - » Need to preserve data dependences

Cache summary

- Cache memory can be organized as direct mapped, set associative, or fully associative
- Can be write-through or write-back
- Extra bits such as valid and dirty bits help keep track of the status of the cache