# CMSC 411
# Computer Systems Architecture
# Lecture 9
# Memory Hierarchy (cont.)

**Alan Sussman**
**als@cs.umd.edu**

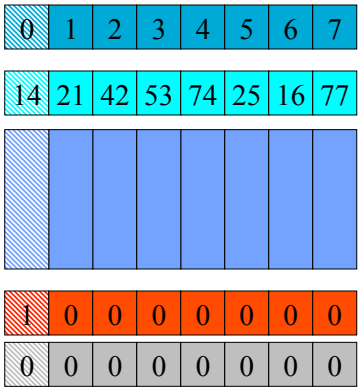
## Administrivia

- **Homework #2 due Tuesday**
- **Project #1 posted later today**
- **Finish reading Appendix B of H&P, start reading Ch. 2, but only up through 2.3 for now**



## Fully associative cache (cont.)

With this result.

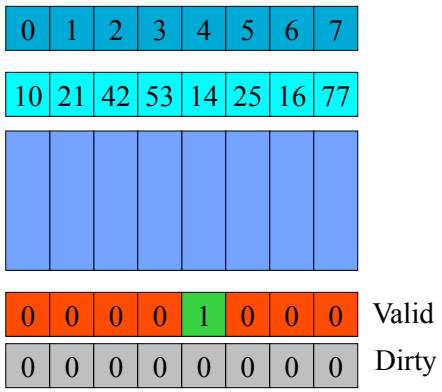





## Managing cache

Use direct mapped cache as an example.

After first read operation, cache memory looked like this.

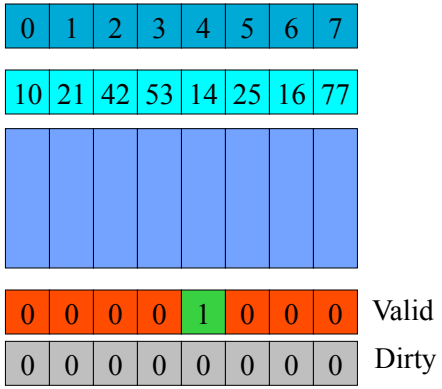


## Managing cache (cont.)

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.

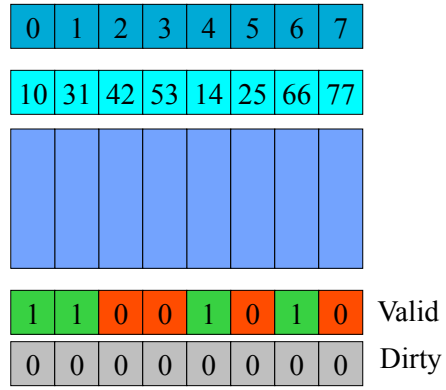





## Managing cache (cont.)

The result looks like this.

Now suppose **write** to block 66.

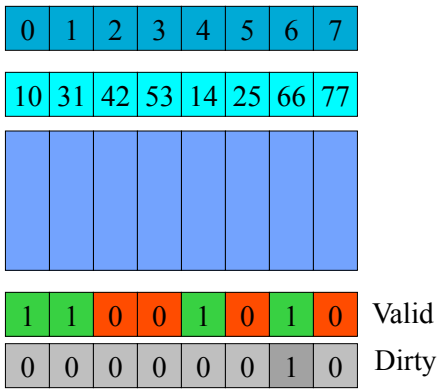





## Managing cache (cont.)

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

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

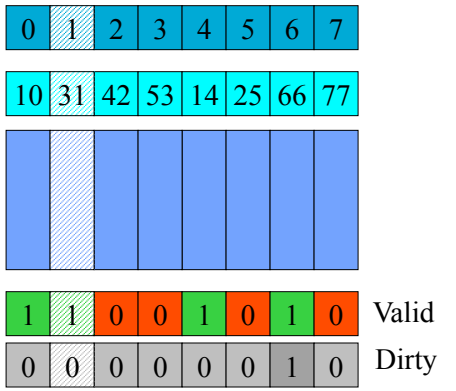





## Managing cache (cont.)

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

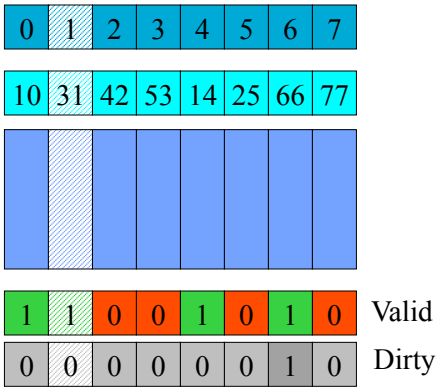
If it is block 41, then must overwrite block 31.



## 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.

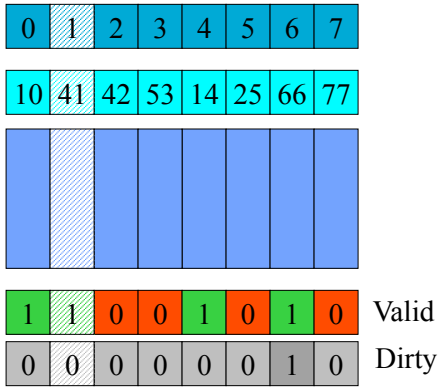





## Write-back cache

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 the data requested is not in cache, call that a *cache miss***
- **It's easy to make most reads from cache fast: just pull the data into a register as soon as it is accessed, while checking whether the address matches the tag. If not, 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

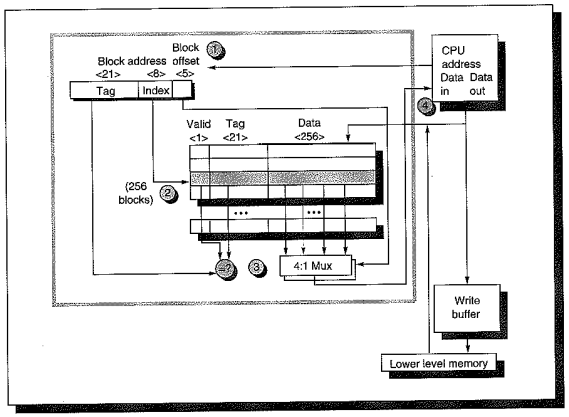
- **Which is better?**
  - **Write back gives faster writes, since don't have to wait for main memory**
  - **Write back is very efficient if want to modify many bytes in a given block**
  - **But write back can slow down some reads, since a cache miss might cause a write back**
  - **In multiprocessors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

## 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

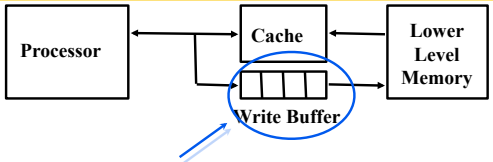


## Example: Alpha 21064






## Write Buffers for Write-Through Caches




**Holds data awaiting write-through to lower level memory**

**Q. Why a write buffer ?** **A. So CPU doesn't stall**

**Q. Why a buffer, why not just one register ?** **A. Bursts of writes are common.**

**Q. Are Read After Write (RAW) hazards an issue for write buffer?** **A. Yes! Drain buffer before next read, or send read 1st after check write buffers.**



## How much do stalls slow a machine?

- Suppose that on pipelined MIPS, each instruction takes, on average, 2 clock cycles, not counting cache faults/misses
- Suppose, on average, there are 1.33 memory references per instruction, memory access time is 50 cycles, and the miss rate is 2%
- Then each instruction takes, on average:

$$2 + (0 \times .98) + (1.33 \times .02 \times 50) = 3.33 \text{ clock cycles}$$

## Memory stalls (cont.)

- To reduce the impact of cache misses, can reduce any of three parameters:
  - main memory access time (miss penalty)
  - cache access (hit) time
  - miss rate



## Example: Apple iMac G5

Managed by compiler

Managed by hardware

Managed by OS, hardware, application

| | Reg | L1 Inst | L1 Data | L2 | DRAM | Disk |
|---|---|---|---|---|---|---|
| Size | 1K | 64K | 32K | 512K | 256M | 80G |
| Latency Cycles, Time | 1, 0.6 ns | 3, 1.9 ns | 3, 1.9 ns | 11, 6.9 ns | 88, 55 ns | $10^7$, 12 ms |

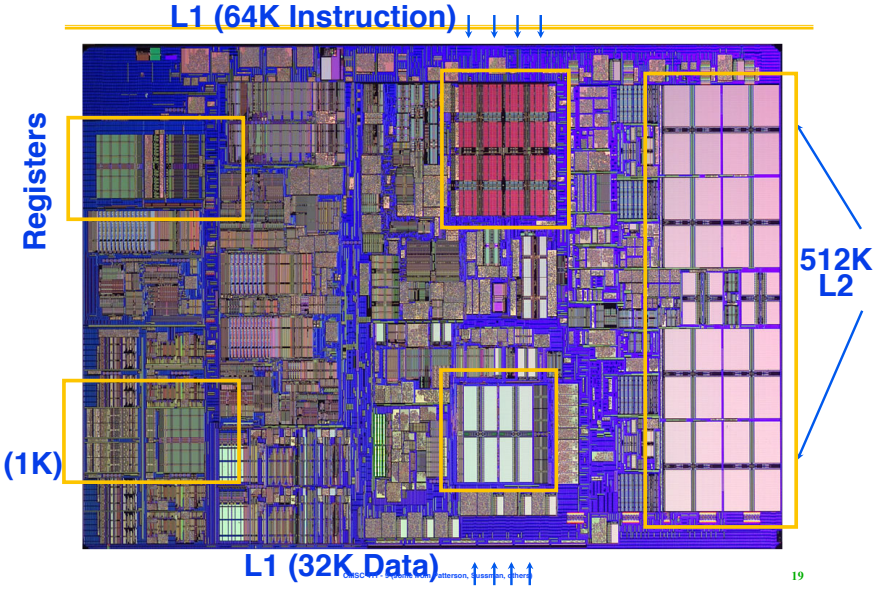
iMac G5 1.6 GHz

**Goal: Illusion of large, fast, cheap memory**

**Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access**



## iMac's PowerPC 970: All caches on-chip






## 5 Basic Cache Optimizations

- Reducing Miss Rate

1. Larger Block size (compulsory misses)
2. Larger Cache size (capacity misses)
3. Higher Associativity (conflict misses)

- Reducing Miss Penalty

4. Multilevel Caches

- Reducing hit time

5. Giving Reads Priority over Writes
   - E.g., Read completes before earlier writes in write buffer

## More Terminology

- 'write-allocate'
  - ensure block in cache before performing a write operation
- 'write-no-allocate'
  - don't allocate block in cache if not already there



## Another write buffer optimization

- Write buffer mechanics, with *merging*
  - An entry may contain multiple words (maybe even a whole cache block)
  - If there's an empty entry, the data and address are written to the buffer, and the CPU is done with the write
  - If buffer contains other modified blocks, check to see if new address matches one already in the buffer – if so, combine the new data with that entry
  - If buffer full and no address match, cache and CPU wait for an empty entry to appear (meaning some entry has been written to main memory)
  - Merging improves memory efficiency, since multi-word writes usually faster than one word at a time



## 5 Basic Cache Optimizations

- Reducing Miss Rate

1. Larger Block size (compulsory misses)
2. Larger Cache size (capacity misses)
3. Higher Associativity (conflict misses)

- Reducing Miss Penalty

4. Multilevel Caches

- Reducing hit time

5. Giving Reads Priority over Writes
   - E.g., Read complete before earlier writes in write buffer



## Don't wait for the whole block on cache miss

- Two ways to do this – suppose need the 10th word in a block:
  - *Early restart*: access the required word as soon as it is fetched, instead of waiting for the whole block
  - *Critical word first*: start the fetch with word 10, and fill in the first few later

## Use a nonblocking cache

- With this optimization, the cache doesn't stop for a miss, but continues to process later requests if possible, even though an earlier one is not yet fulfilled
  - Introduces significant complexity into cache architecture – have to allow multiple outstanding cache requests (maybe even multiple misses)
  - but this is what's done in modern processors



## So Far….

- Fully associative cache
  - memory block can be stored in any cache block
- Write-through cache
  - write (store) changes both cache and main memory right away
  - reads only require getting block on cache miss
- Write-back cache
  - write changes only cache
  - read causes write of dirty block to memory on a replace
- Reads easy to make fast, writes harder
  - read data from cache in parallel with checking address against tag of cache block
  - write must verify address against tag before update
- Reducing memory stalls
  - reduce miss penalty, miss rate, cache hit time
- Reducing miss penalty
  - give priority to read over write misses
  - don't wait for the whole block
  - use a non-blocking cache



## Multi-level cache

- For example, if cache takes 1 clock cycle, and memory takes 50, might be a good idea to add a larger (but necessarily slower) secondary cache in between, perhaps capable of 10 clock cycle access
- Complicates performance analysis (see H&P), but 2nd level cache captures many of 1st level cache misses, lowering effective miss penalty
  - and 3rd level cache has same benefits for 2nd level cache
- Most modern machines have separate 1st level instruction and data caches, shared 2nd level cache
  - and off processor chip shared 3rd level cache