## Lecture 14: Caching

- Last time:
  - Branch prediction
  - Issuing multiple instructions in each cycle
- Today:
  - What part of the pipeline have we been glossing over?
    - Memory!!
  - Very important to overall machine performance.

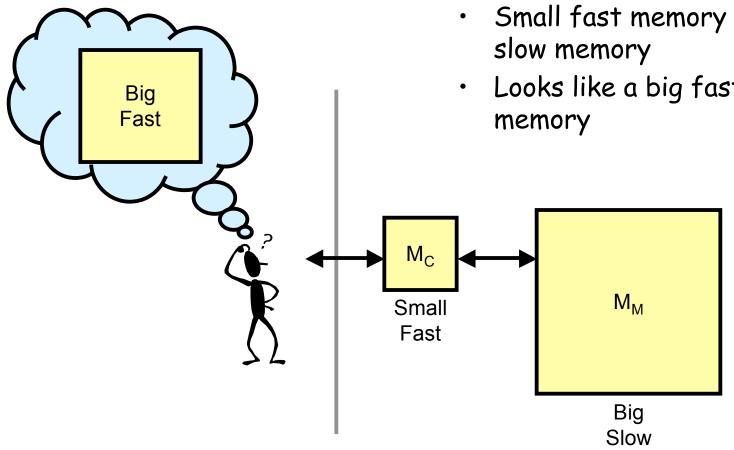
## Memory System Overview

- Memory Hierarchies
  - Latency/Bandwidth/Locality
  - Caches
    - Principles - why does it work
    - Cache organization
    - Cache performance
    - Types of misses (the 3 Cs)
  - Main memory organization
    - DRAM vs. SRAM
    - Bank organization
    - Tracking multiple references
  - Trends in memory system design
- Logical Organization
  - Name spaces
  - Protection and sharing
  - Resource management
    - virtual memory, paging, and swapping
  - Segmentation
  - Capability-based addressing

## The Memory Bottleneck

- Typical CPU clock rate
  - 3 GHz (0.33 ns cycle time)
- Typical DRAM access time
  - 30ns (about 100 cycles)
- Typical main memory access
  - 70ns (210 cycles)
    - DRAM (30), precharge (10), chip crossings (15), overhead (15).
- Our pipeline designs assume 1 cycle access
- How many memory accesses made by typical instruction?
  - 1 instruction word
  - 0.3 data words
- This problem gets worse
  - CPUs get *faster*
  - Memories get *bigger*
- Memory delay is mostly communication time
  - reading/writing a bit is *fast*
  - it takes time to
    - select the right bit
    - route the data to/from the bit
- Big memories are *slow*
- Small memories can be made *fast*

## Cache Memory



- Small fast memory + big slow memory
- Looks like a big fast memory

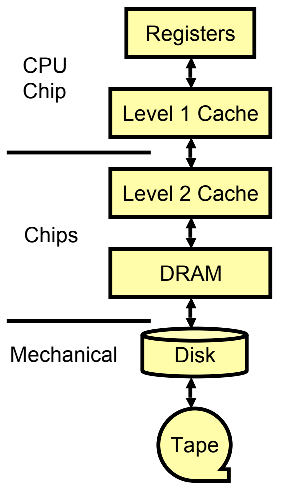
## Bookshelf analogy

- Lots of books on shelves
- A few books on my desk
- One book that I'm reading

- Shelves = main memory
- Desk = cache

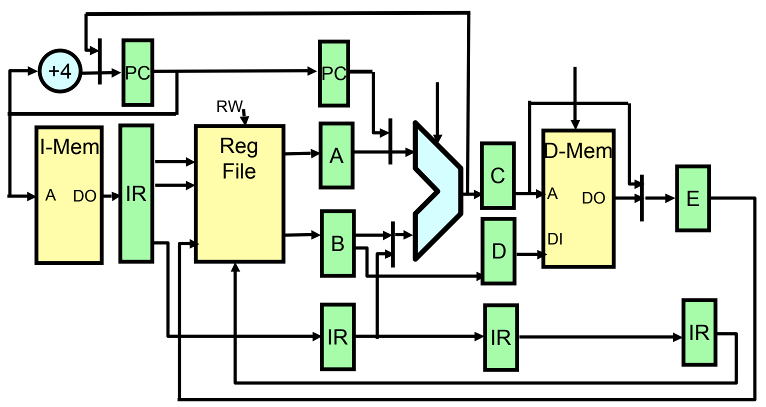
- Book = block
- Page in book = memory location

## The Memory Hierarchy



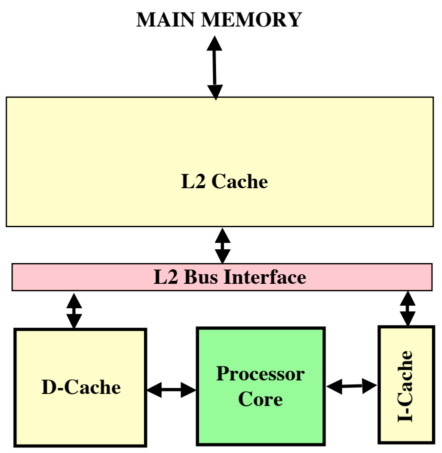
| Latency | Bandwidth | |
|---|---|---|
| 1 cyc | 3-10 words/cycle < 1KB | compiler managed |
| 1-3cy | 1-2 words/cycle 32KB -1MB | hardware managed |
| 5-10cy | 1 word/cycle 1MB - 4MB | hardware managed |
| 30-100cy | 0.5 words/cycle 64MB - 4GB | OS managed |
| $10^6$-$10^7$cy | 0.01 words/cycle 4GB+ | OS managed |

## Where Does the Memory Hierarchy Fit In?



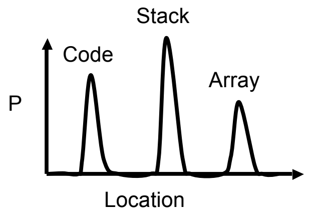
## Typical Cache Organization

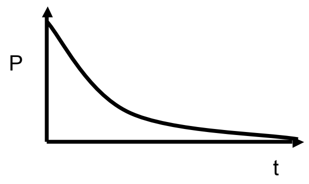
**Alpha 21264: 64KB I-Cache**
**64KB D-Cache**
**> 1MB L2 Cache**



## Locality of Reference

- Spatial Locality
  - likely to reference data *near* recent references
- Temporal Locality
  - likely to reference the same data that was referenced recently



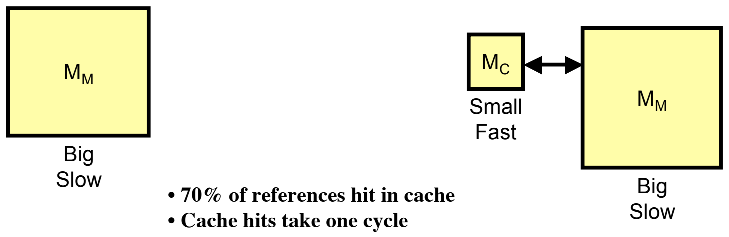


## Program Behavior

- Locality depends on type of program
- Some programs 'behave' well
  - small loop operating on data on stack
- Some programs don't
  - frequent calls to nearly random subroutines
  - traversal of large, sparse data set
    - essentially random data references with no reuse
- Most programs exhibit some degree of locality

## Example

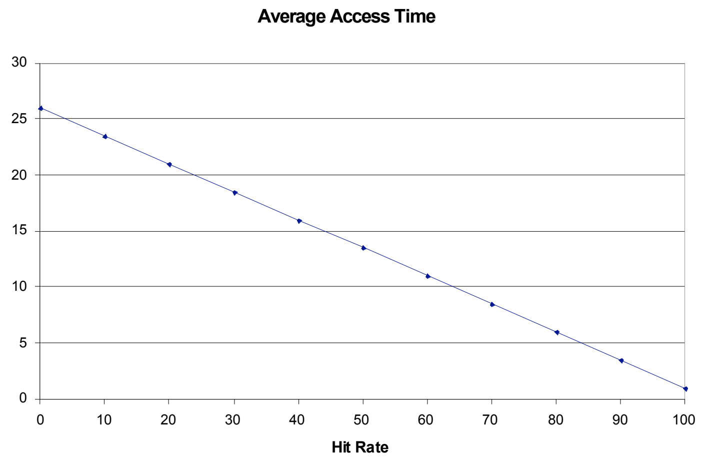
**What is the average memory access time?**



- **70% of references hit in cache**
- **Cache hits take one cycle**
- **Main memory references take 25 cycles**

$$AMAT = Latency_{Hit} + P(miss)*Latency_{Miss}$$

## Impact of Hit Rate



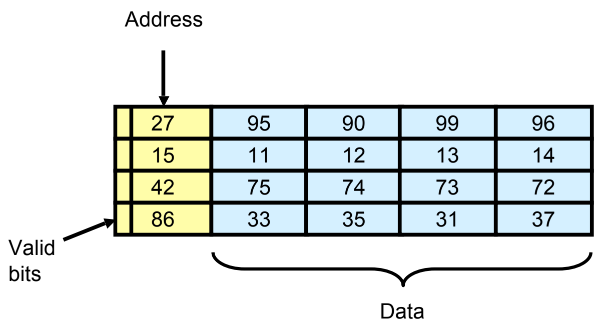
## Two kinds of "fast & small" memory

- Programmer manages it manually
  - Sometimes called a "scratchpad" memory
  - CELL processor uses this approach
- Hardware manages it automatically
  - Invisible to programmer
  - Referred to as a "cache"
  - Most CPUs use this approach
  - Easy for programmers; Hard for hardware

## How does hardware keep track of what's in the fast memory (cache)?

- How does it know what's in the cache 'now'?
- How does it decide what to add to the cache?
- How does it decide what to remove from the cache?
- How does it keep the cache consistent with the off-chip memory?

## Cache Organization



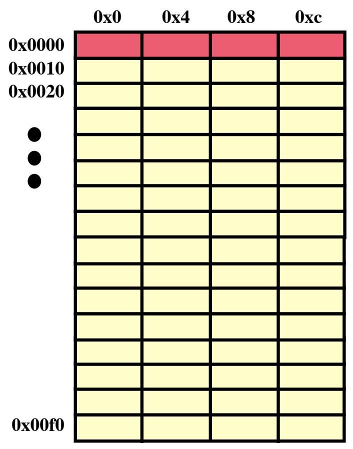
| | | | | |
|---|---|---|---|---|
| 27 | 95 | 90 | 99 | 96 |
| 15 | 11 | 12 | 13 | 14 |
| 42 | 75 | 74 | 73 | 72 |
| 86 | 33 | 35 | 31 | 37 |

- **Where does a block get placed?**
- **How do we find it?**
- **Which one do we replace when a new one is brought in?**
- **What happens on a write?**

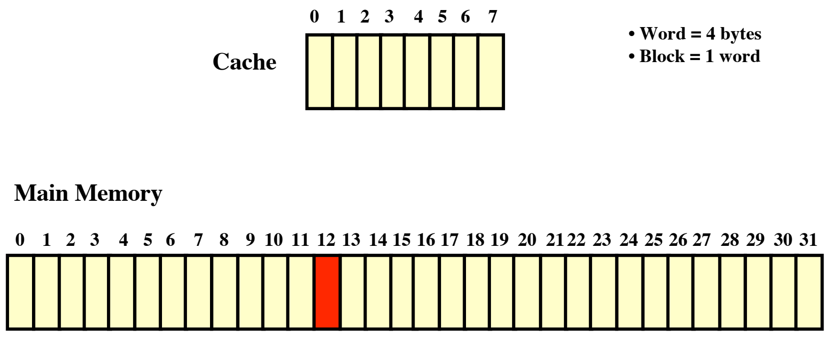
## Cache Definitions

- Cache block (= cache line)

- Index
- Offset



## Where Does a Block Go in the Cache?



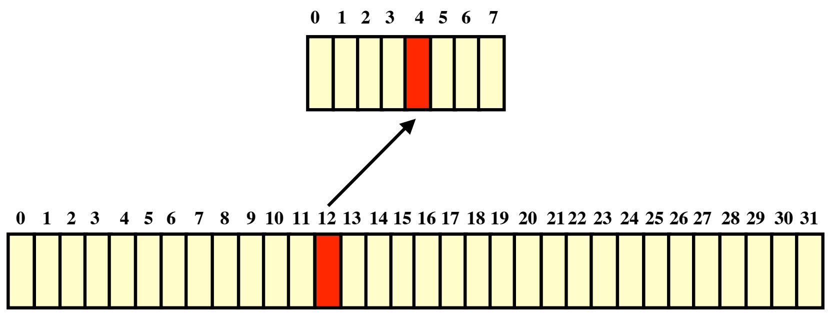
- Word = 4 bytes
- Block = 1 word

- Where do we put block 12?

## Direct Mapped

- **Each block mapped to exactly 1 cache location**

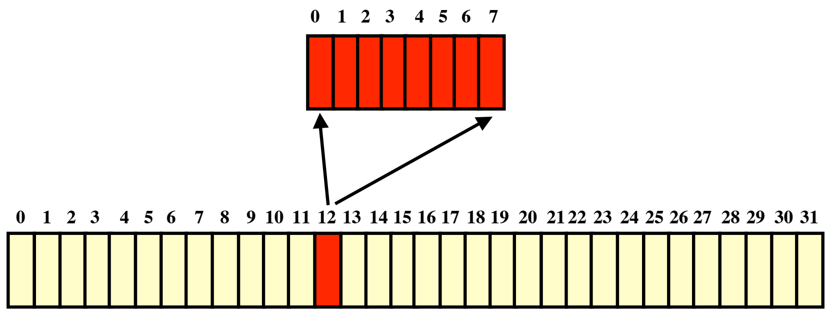
**Cache location = (block address) MOD (# blocks in cache)**



## Fully Associative

- **Each block mapped to any cache location**

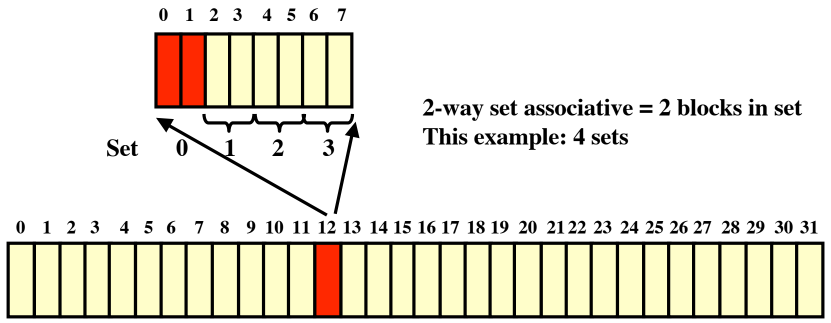
  **Cache location = any**



## Set Associative

- **Each block mapped to subset of cache locations**

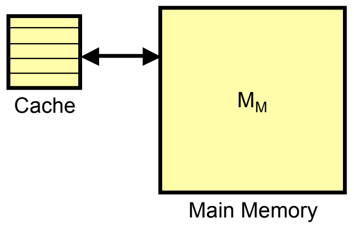
  **Set selection = (block address) MOD (# sets in cache)**



**2-way set associative = 2 blocks in set**
**This example: 4 sets**

## Block Placement

- Mapping function from Big Memory to Small memory
- On block-by-block basis
  - Direct Mapped: 1 place
  - Fully Associative: Anywhere
  - Set Associative: Subset of cache
- Use address to do mapping and lookup



## Taking advantage of Spatial Locality

- Instead of each block in cache being just 1 word, what if we made it 4 words?
- When we get our 1 word instruction or 1 word of data from memory to put in the cache, get the next 3 as well, because they are likely to be used soon!
- Need to add a way to choose which of the 4 words in the block we want when we go to cache… called block offset.