CSE 141: Introduction to Computer Architecture

Memory & Caches

Part I: Basic Memory & Cache Designs

Finally, telling the truth about Memory

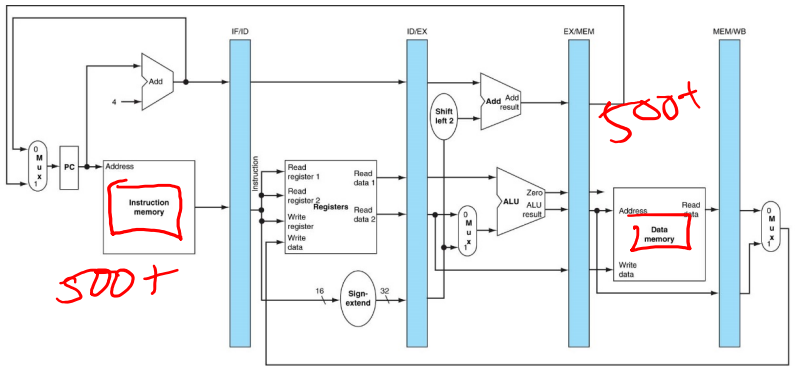
- Up to this point, we've been assuming memory can be accessed in a single cycle.
- In fact, that was true once.
- But cycle time has decreased rapidly (for high performance machines), while memory access time has decreased very little.
- In modern computers, memory latency can be in the neighborhood of 350-500 cycles!

The truth about memory latency

- So then what is the point of pipelining, branch prediction, etc. if memory latency is 500 cycles?
- Keep in mind, 20% of instructions are loads and stores, and we fetch (read inst memory) every instruction.

```
lw  R4, 1000(R2)   IF ID EX M - - - - - - - - -  ...  - - - - - - - WB
lw  R8, 200(R4)      IF  ID  B - - - - - - - - -      - - - - - - - -ID EX M- - - - - - - - -... - - - - - - - WB
add  R10, R8, R10      IF  B - - - - - - - - -        - - - - - - - -  ID  B - - - - - - - - -        - - - - - - - -ID EX M
```

CPI = ~ ??

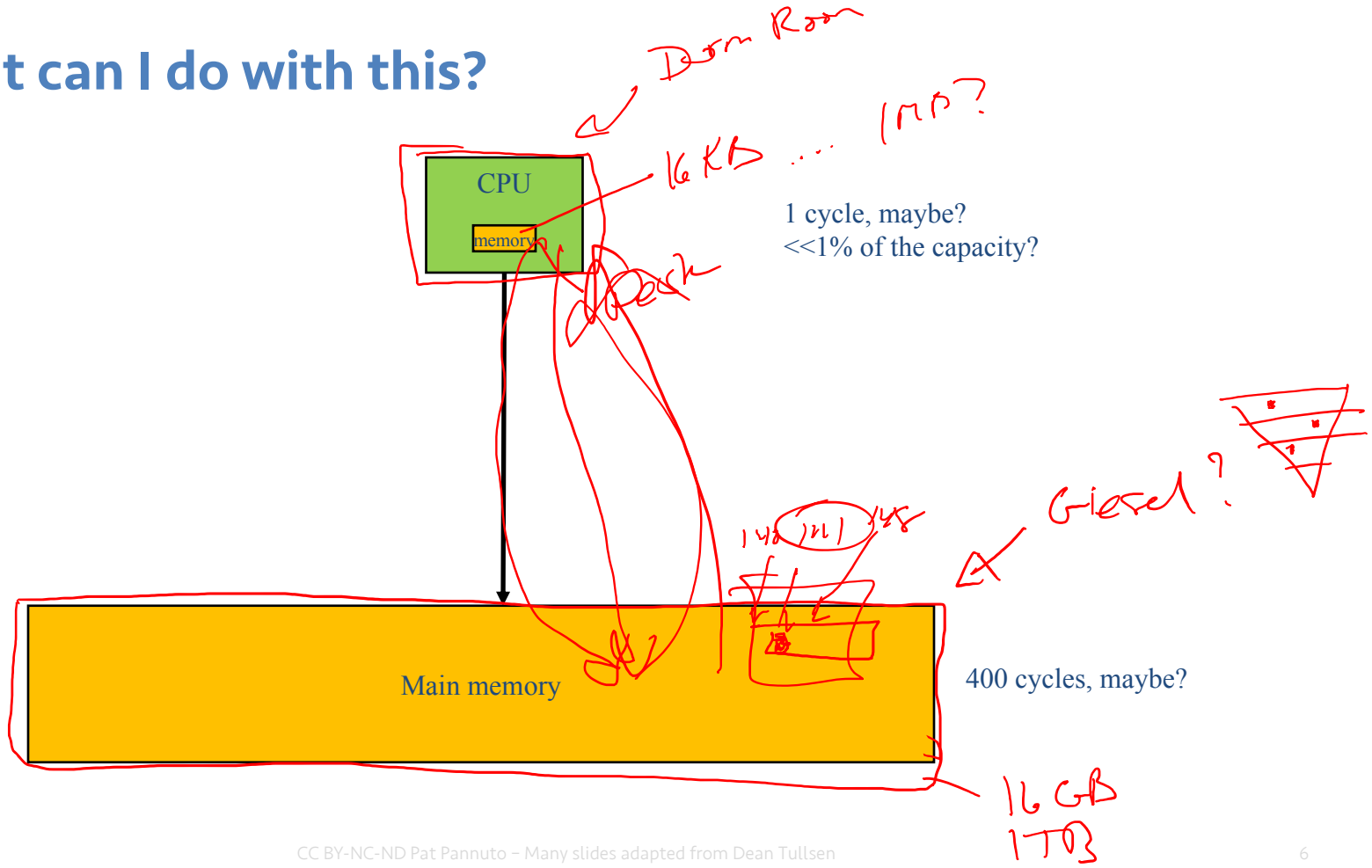


But wait…

- That is assuming DRAM technology, which is necessary for large main memories (multiple gigabytes, for example)
- But we can design much smaller (capacity) memories using SRAM, even on chip.
- If we still want to access it in a cycle, it should be KB, not MB or GB.

→ Limitations from real-world H/W

So what can I do with this?



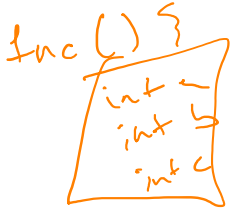
Memory Locality

- Memory hierarchies take advantage of *memory locality*.

Memory Locality

- Memory hierarchies take advantage of *memory locality*.
- *Memory locality* is the principle that future memory accesses are *near* past accesses.

Memory Locality



- Memory hierarchies take advantage of memory locality.
- Memory locality is the principle that future memory accesses are near past accesses.
- Memories take advantage of two types of locality
  - near in time => we will often access the same data again very soon
  - near in space/distance => our next access is often very close to our last access (or recent accesses).

TEMPORAL (time)

SPATIAL

re-use of the same data

diff nearby data

int A[100] , for(i=0 ...) { A[i] }

- (this sequence of addresses exhibits both temporal and spatial locality)
  - 1,2,3,1,2,3,8,8,47,9,10,8,8…

S + T

S

T

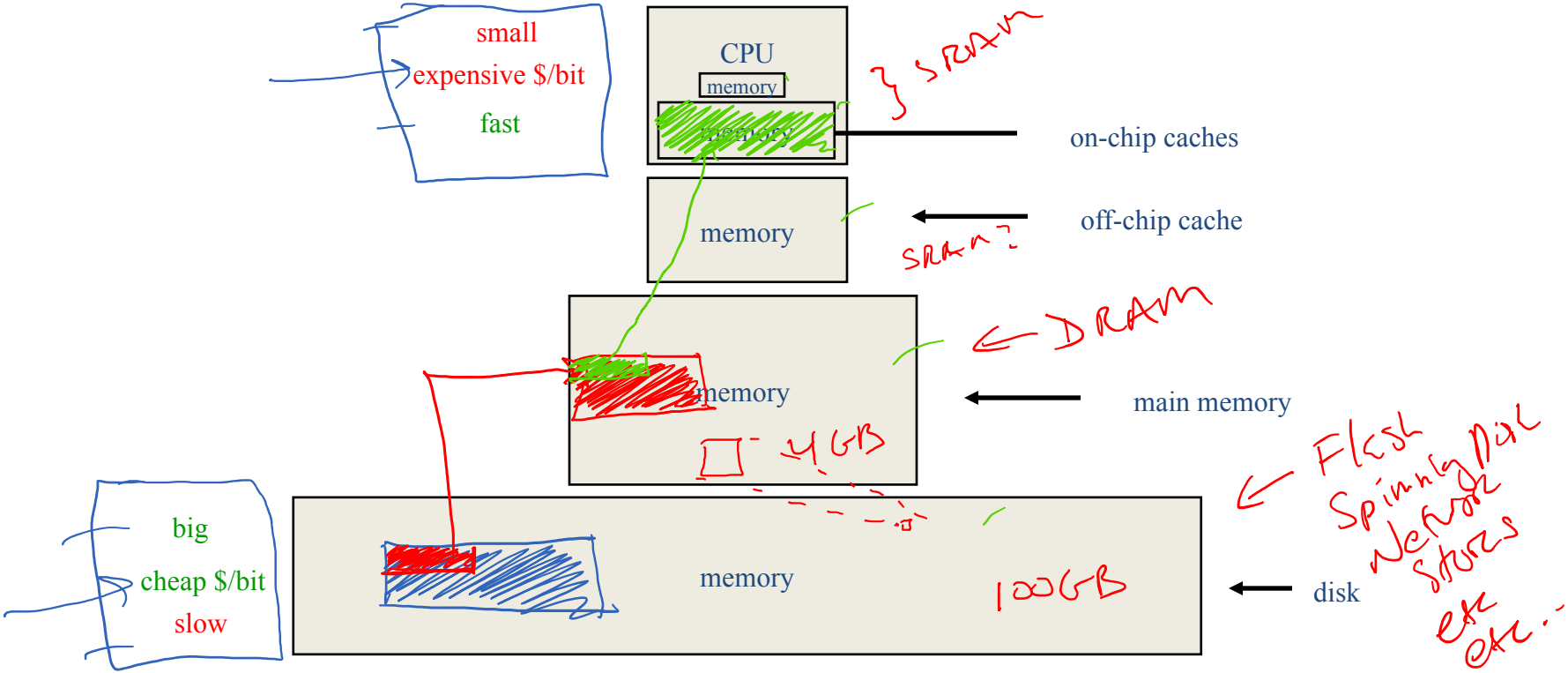
8 ⇒ T

Locality and cacheing

- Memory hierarchies exploit locality by cacheing (keeping close to the processor) data likely to be used again.
- This is done because we can build large, slow memories and small, fast memories, but we can't build large, fast memories.
- If it works, we get the illusion of SRAM access time with disk capacity

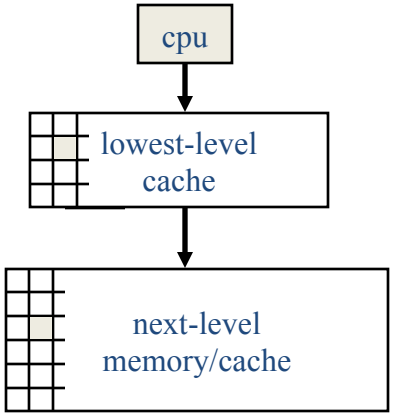
- SRAM access times are ~1ns at cost of $2000 to $5000 per Gbyte.
- DRAM access times are ~70ns at cost of $20 to $75 per Gbyte.
- Disk access times are 5 to 20 million ns at cost of $.20 to $2 per Gbyte.

A typical memory hierarchy



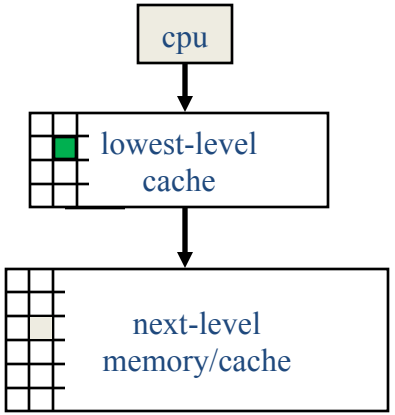
so then where is my program and data??

Cache Fundamentals



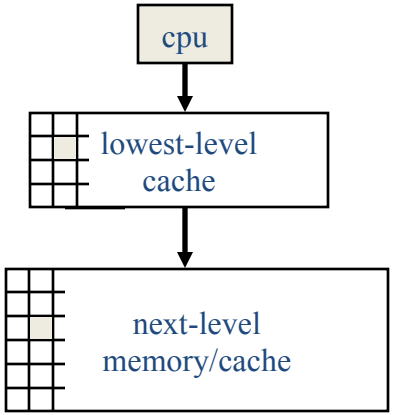
Cache Fundamentals

- *cache hit* -- an access where the data is found in the cache.

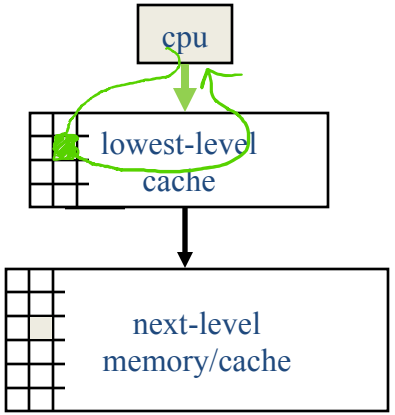


Cache Fundamentals

- cache hit -- an access where the data is found in the cache.
- *cache miss* -- an access which isn't



Cache Fundamentals



- cache hit -- an access where the data is found in the cache.
- cache miss -- an access which isn't
- *hit time* -- time to access the cache

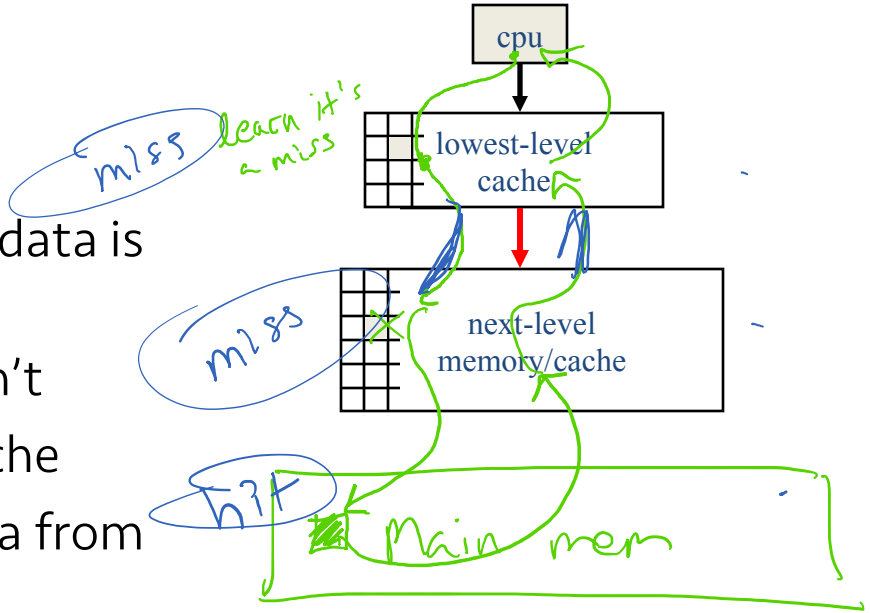
latency

F D X M W

DM

Cache Fundamentals

- cache hit -- an access where the data is found in the cache.
- cache miss -- an access which isn't
- hit time -- time to access the cache
- *miss penalty* -- time to move data from further level to closer

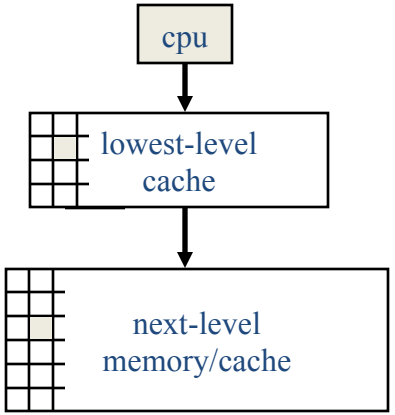


F D X M W

DM

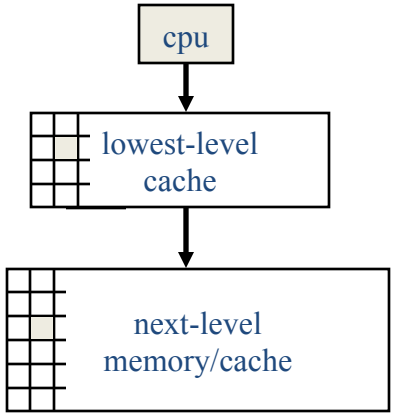
Cache Fundamentals

- cache hit -- an access where the data is found in the cache.
- cache miss -- an access which isn't
- hit time -- time to access the cache
- miss penalty -- time to move data from further level to closer
- *hit ratio* -- percentage of time the data is found in the cache



$$\frac{\#\text{hits}}{\#\text{accesses}} * 100$$

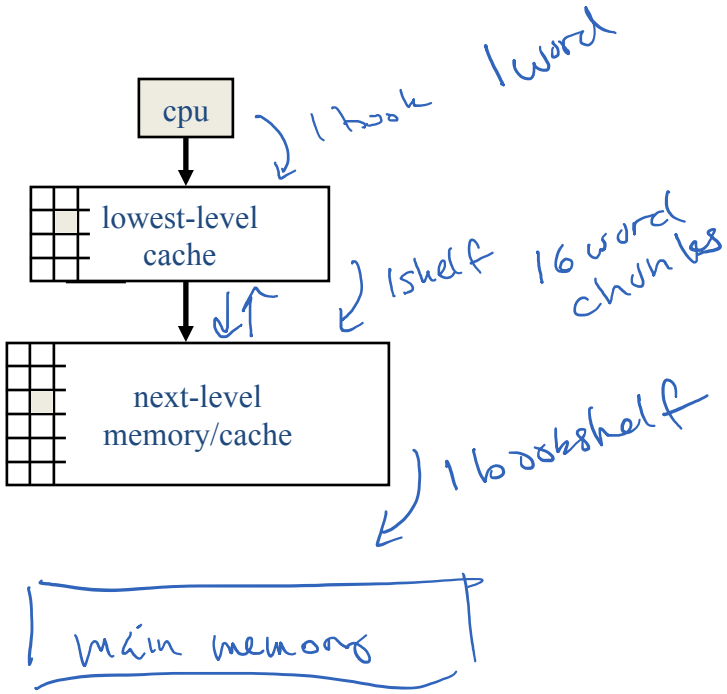
Cache Fundamentals



- cache hit -- an access where the data is found in the cache.
- cache miss -- an access which isn't
- hit time -- time to access the cache
- miss penalty -- time to move data from further level to closer
- hit ratio -- percentage of time the data is found in the cache
- *miss ratio* -- (1 - hit ratio)

Cache Fundamentals, cont.

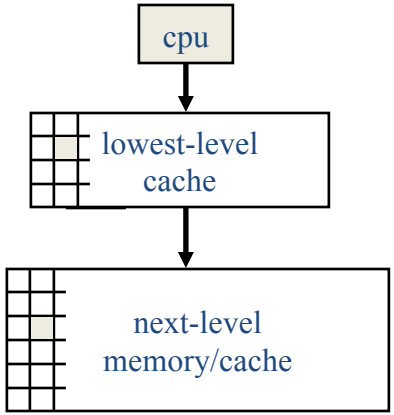
- *cache block size* or *cache line size*– the amount of data that gets transferred on a cache miss.



Cache Fundamentals, cont.

F D X M W

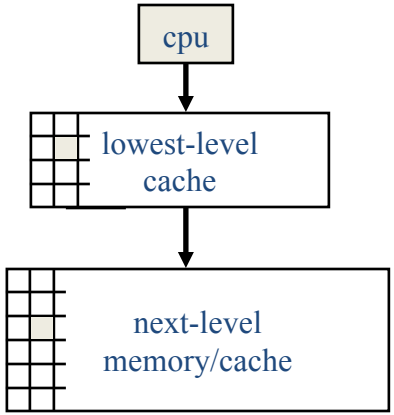
- *cache block size* or *cache line size* – the amount of data that gets transferred on a cache miss.
- *instruction cache* – cache that only holds instructions.



Cache Fundamentals, cont.

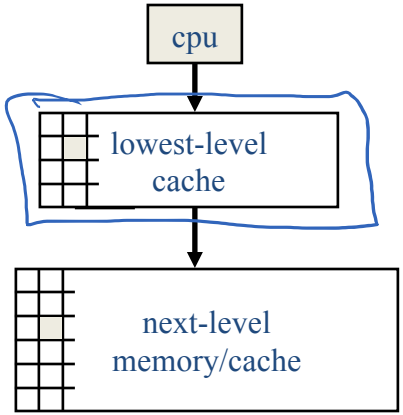
F D X M W

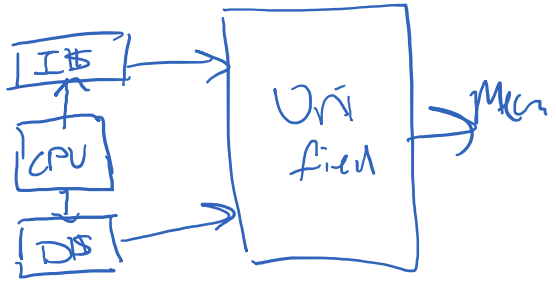
- *cache block size* or *cache line size* – the amount of data that gets transferred on a cache miss.
- *instruction cache* – cache that only holds instructions.
- *data cache* – cache that only caches data.



Cache Fundamentals, cont.

- *cache block size* or *cache line size* – the amount of data that gets transferred on a cache miss.
- *instruction cache* – cache that only holds instructions.
- *data cache* – cache that only caches data.
- *unified cache* – cache that holds both.

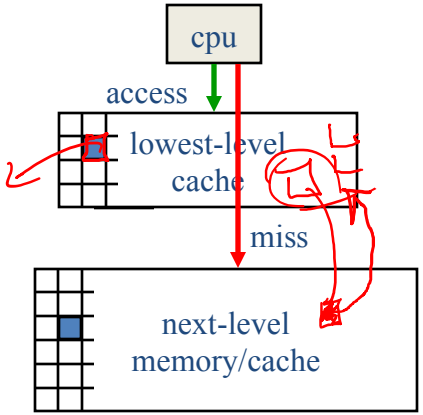




Cacheing Issues

- On a memory access -
  - How do I know if this is a hit or miss?

- On a cache miss -
  - where to put the new data?
  - what data to throw out?
  - how to remember what data this is?



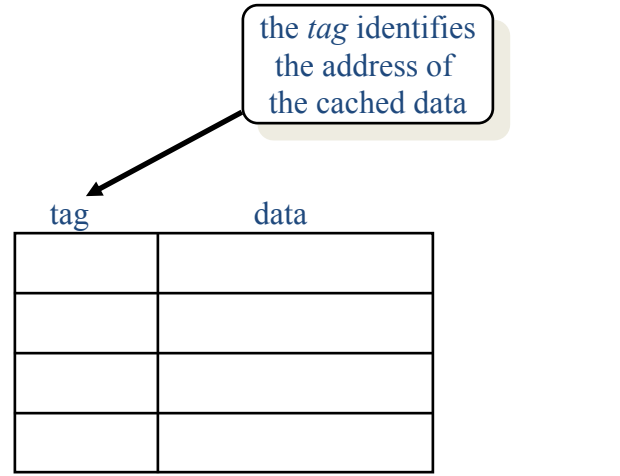
Hardware implications on cache design

- Caches are basically *the* thing that make real workloads fast
- The size of a cache is inversely proportional to its speed
  - Smaller caches are faster
- And every bit counts

- This is why caches use as few **bits** as possible to do their work
  - This makes caches tricky to walk through as a human

A simple cache

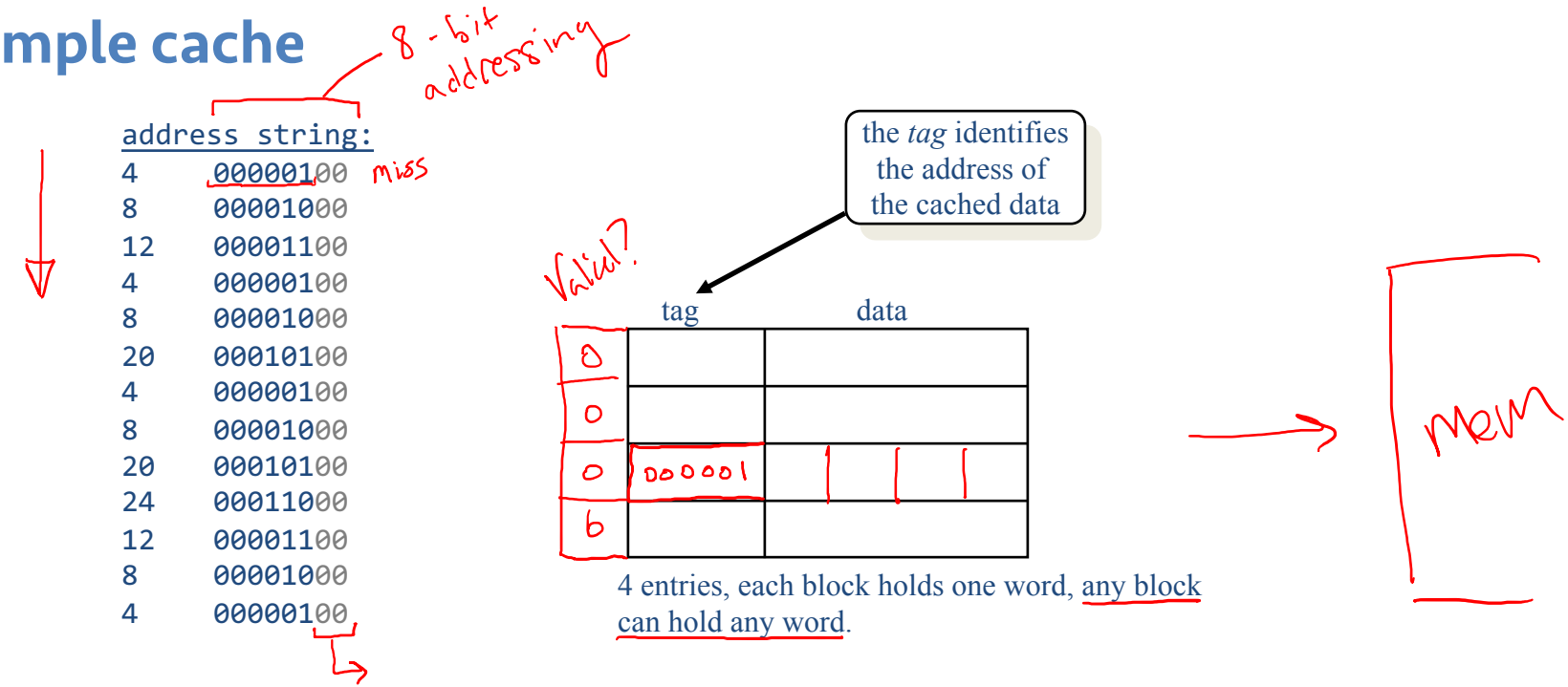
address	string:
4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100



4 entries, each block holds one word, any block can hold any word.

- A cache that can put a line of data anywhere is called F. A.
- The most popular replacement strategy is *LRU* ( ).

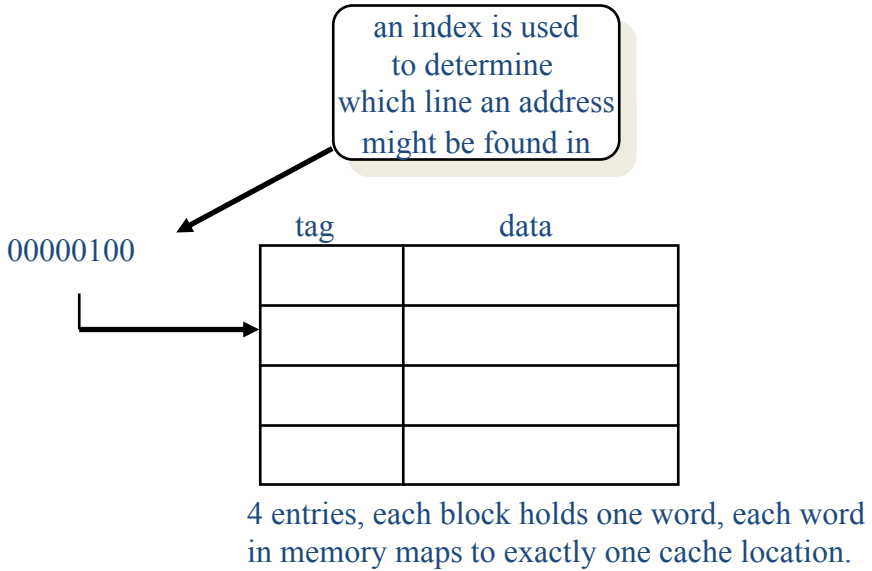
A simple cache



- A cache that can put a line of data anywhere is called **Fully Associative**
- The most popular replacement strategy is *LRU* (**Least Recently Used**).

A simpler cache

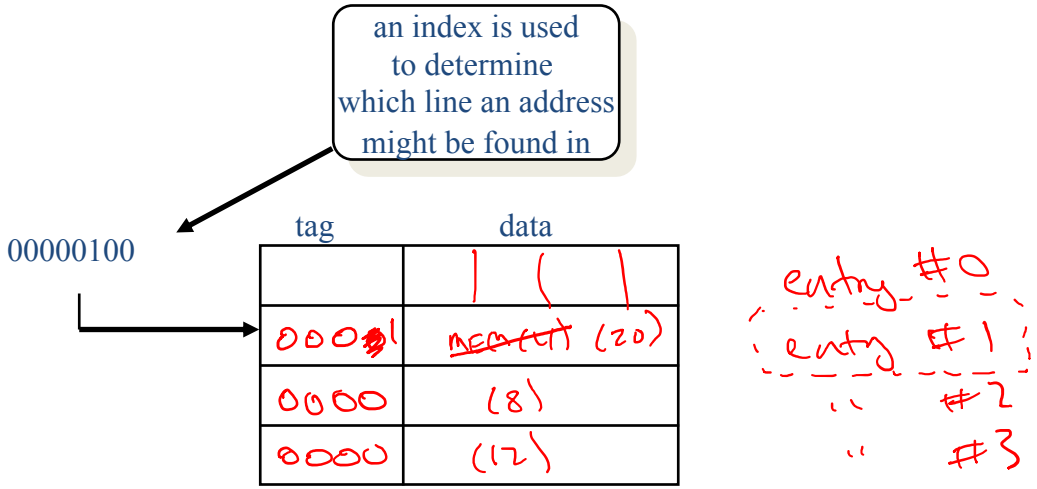
address	string:
4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100



- A cache that can put a line of data in exactly one place is called _____.
- Advantages/disadvantages vs. fully-associative?

A simpler cache

address string:
4
8
12
4
8
20
4
8
20
24
12
8
4



- A cache that can put a line of data in exactly one place is called **direct mapped**
- Advantages/disadvantages vs. fully-associative?