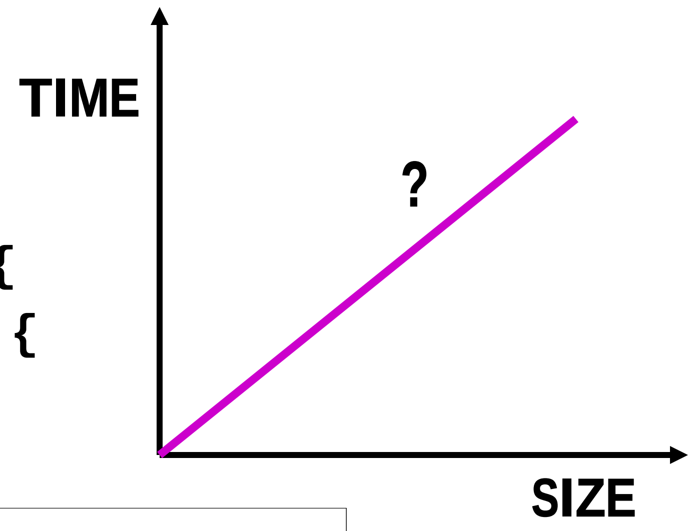
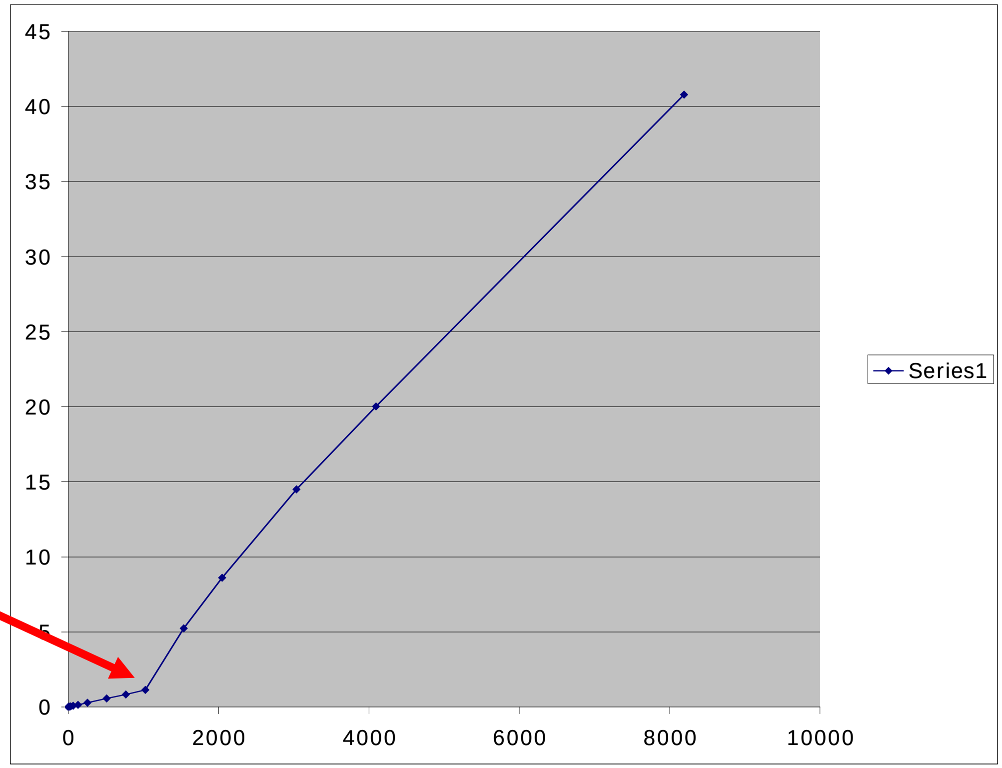
How will execution time grow with SIZE?

```
int array[SIZE];
int sum = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
   for (int j = 0 ; j < SIZE ; ++ j) {
        sum += array[j];
   }
}
```





array fits in cache

Large and fast

- Computers depend upon large and fast storage systems
  - database applications, scientific computations, video, music, etc
  - pipelined CPUs need quick access to memory (IF, MEM)

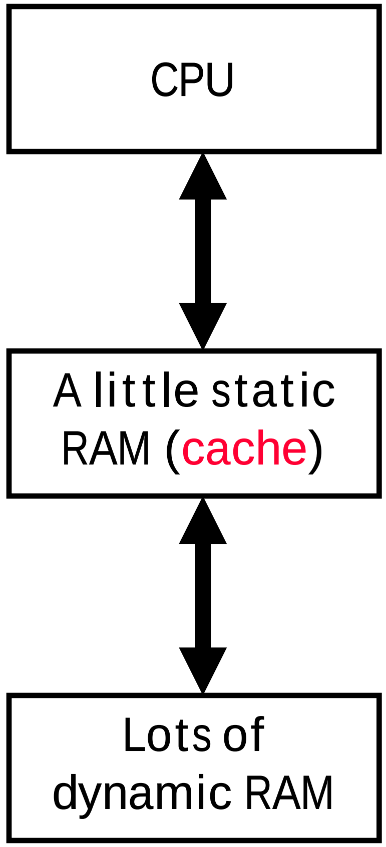
- So far we've assumed that IF and MEM can happen in 1 cycle
  - unfortunately, there is a tradeoff between speed, cost and capacity

Storage	Delay	Cost / MB	Capacity
Static RAM	1-10 cycles	~$5	128KB-2MB
Dynamic RAM	100-200 cycles	~$0.10	128MB-4GB
Hard disks	10,000,000 cycles	~$0.0005	20GB-400GB

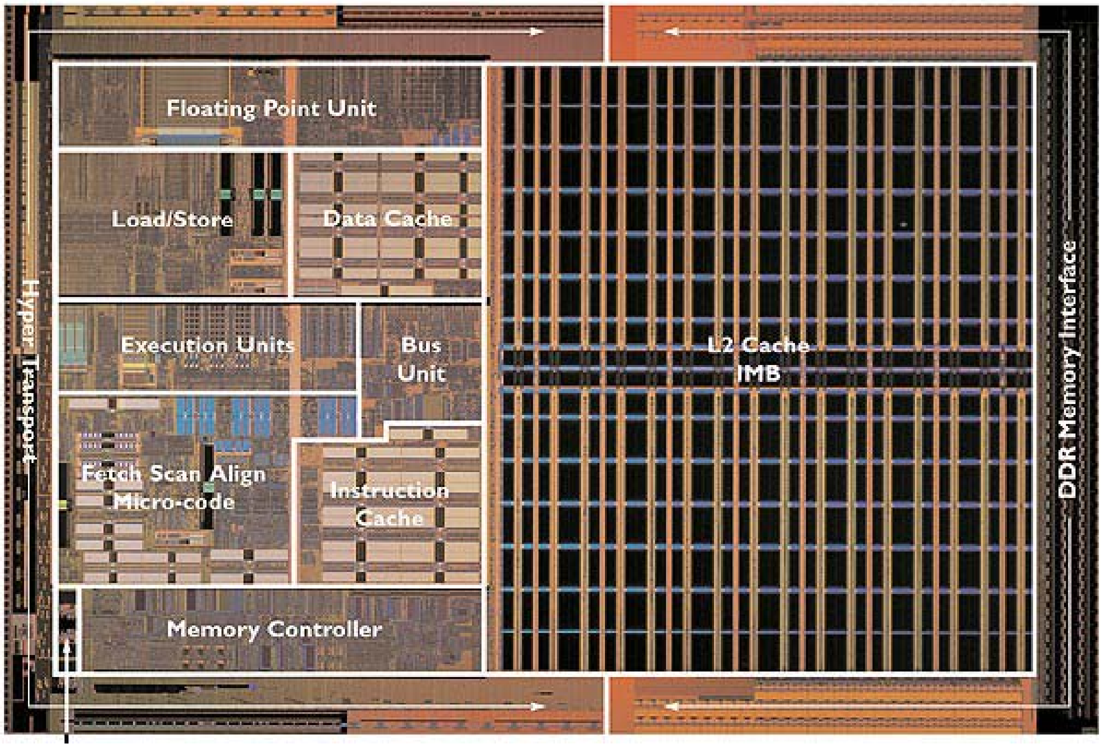
  - fast memory is expensive, but dynamic memory very slow

Introducing caches

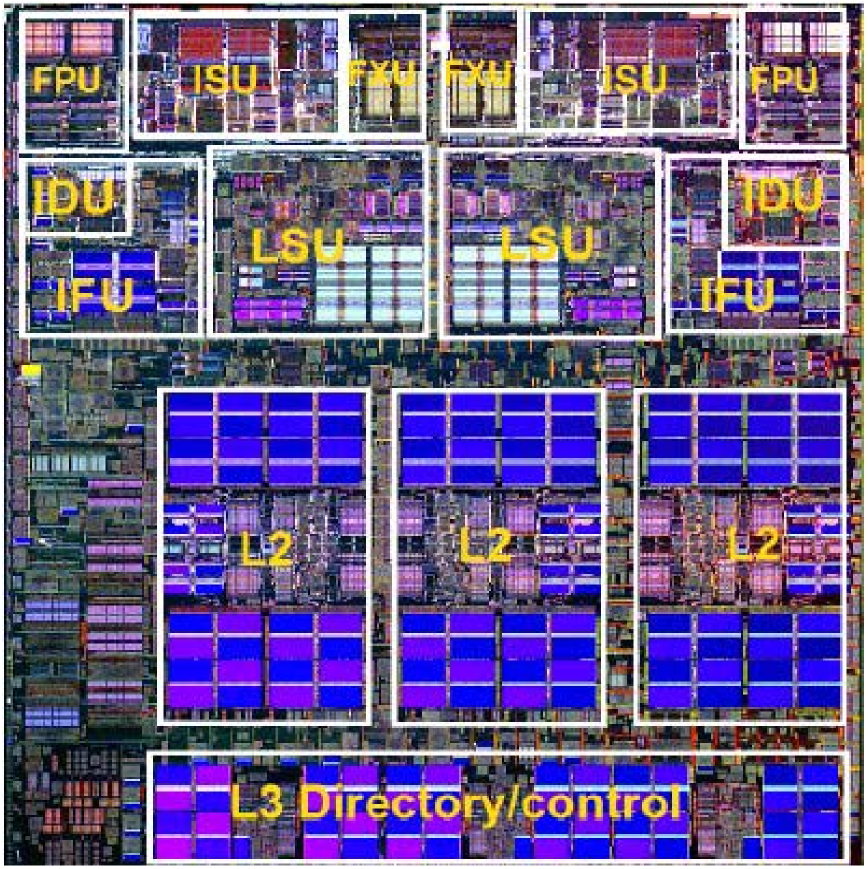
- Caches help strike a balance
- A cache is a small amount of fast, expensive memory
  - goes between the processor and the slower, dynamic main memory
  - keeps a copy of the most frequently used data from the main memory
- Memory access speed increases overall, because we've made the common case faster
  - reads and writes to the most frequently used addresses will be serviced by the cache
  - we only need to access the slower main memory for less frequently used data
- Principle used elsewhere: Networks, OS, ...



Today: Cache introduction



Single-core
Two-level cache



Dual-core
Three-level cache

The principle of locality

- Usually difficult or impossible to figure out "most frequently accessed" data or instructions before a program actually runs
  - — hard to know what to store into the small, precious cache memory
- In practice, most programs exhibit *locality*
  - — cache takes advantage of this
- The principle of temporal locality: if a program accesses one memory address, there is a good chance that it will access the same address again
- The principle of spatial locality: that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses
- *Example*: loops (instructions), sequential array access (data)

Locality in data

- Temporal: programs often access same variables, especially within loops

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + f(i);
```

- Ideally, commonly-accessed variables will be in registers
  - — but there are a limited number of registers
  - — in some situations, data must be kept in memory (e.g., sharing between threads)

- Spatial: when reading location *i* from main memory, a copy of that data is placed in the cache but also copy *i*+1, *i*+2, ...
  - — useful for arrays, records, multiple local variables

Definitions: Hits and misses

- A cache hit occurs if the cache contains the data that we're looking for ☺
- A cache miss occurs if the cache does not contain the requested data ☹
- Two basic measurements of cache performance:
  - the hit rate = percentage of memory accesses handled by the cache (miss rate = 1 – hit rate)
  - the miss penalty = the number of cycles needed to access main memory on a cache miss
- Typical caches have a hit rate of 95% or higher
- Caches organized in levels to reduce miss penalty

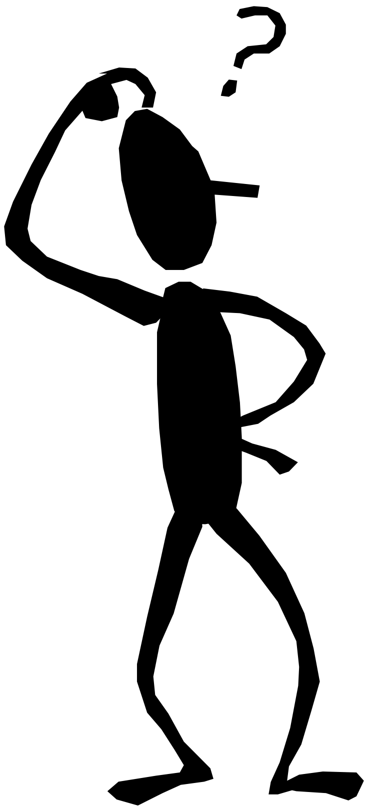
A simple cache design

- Caches are divided into blocks, which may be of various sizes
  - the number of blocks in a cache is usually a power of 2
  - for now we'll say that each block contains one byte (this won't take advantage of spatial locality, but we'll do that next time)

index == row

index	8-bit data
000	
001	
010	
011	
100	
101	
110	
111	

Four important questions



1. When we copy a block of data from main memory to the cache, where exactly should we put it?

2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?

3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?

4. How can *write* operations be handled by the memory system?

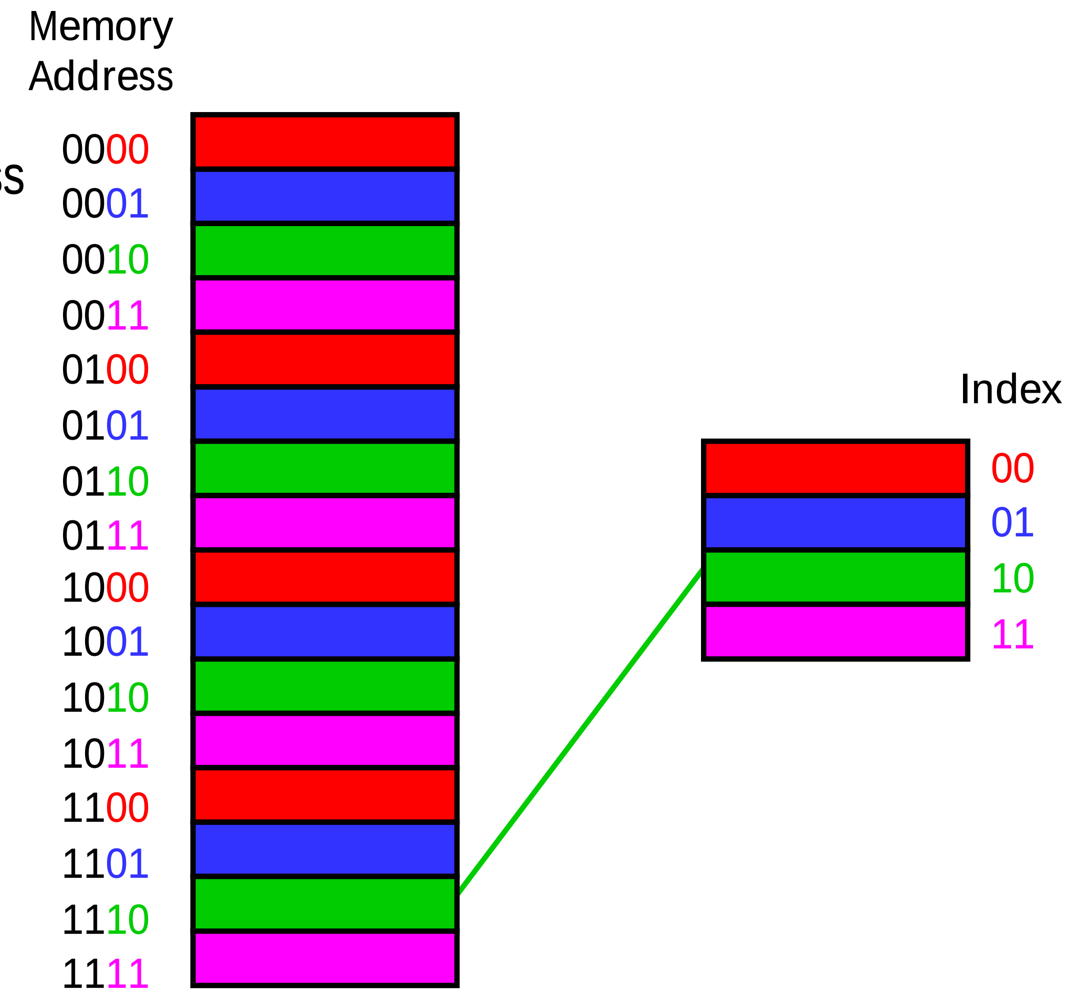
- Questions 1 and 2 are related—we have to know where the data is placed if we ever hope to find it again later!

Where should we put data in the cache?

- A direct-mapped cache is the simplest approach: each main memory address maps to exactly one cache block

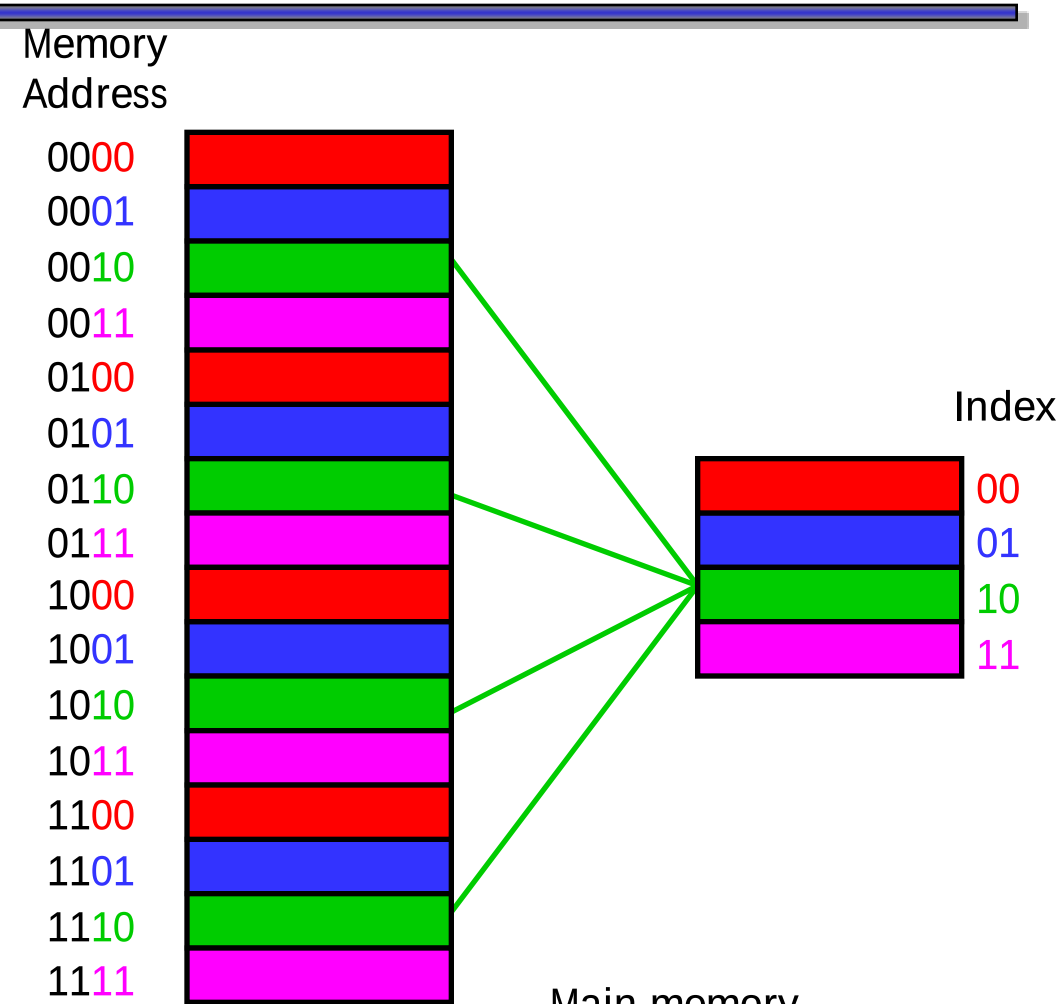
- Notice that index = least significant bits (LSB) of address

- If the cache holds 2^k blocks, index = k LSBs of address



How can we find data in the cache?

- If we want to read memory address *i*, we can use the LSB trick to determine which cache block would contain *i*
- But other addresses might *also* map to the same cache block. How can we distinguish between them?
- We add a tag, using the rest of the address



Index	Tag	Data	Main memory address in cache block
00	00		00 + 00 = 0000
01	11		11 + 01 = 1101
10	01		01 + 10 = 0110
11	01		01 + 11 = 0111

One more detail: the valid bit

- When started, the cache is empty and does not contain valid data
- We should account for this by adding a valid bit for each cache block
  - When the system is initialized, all the valid bits are set to 0
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1

Index	Valid Bit	Tag	Data	Main memory address in cache block
00	1	00		00 + 00 = 0000
01	0	11		Invalid
10	0	01		Invalid
11	1	01		01 + 11 = 0111

- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity

What happens on a memory access

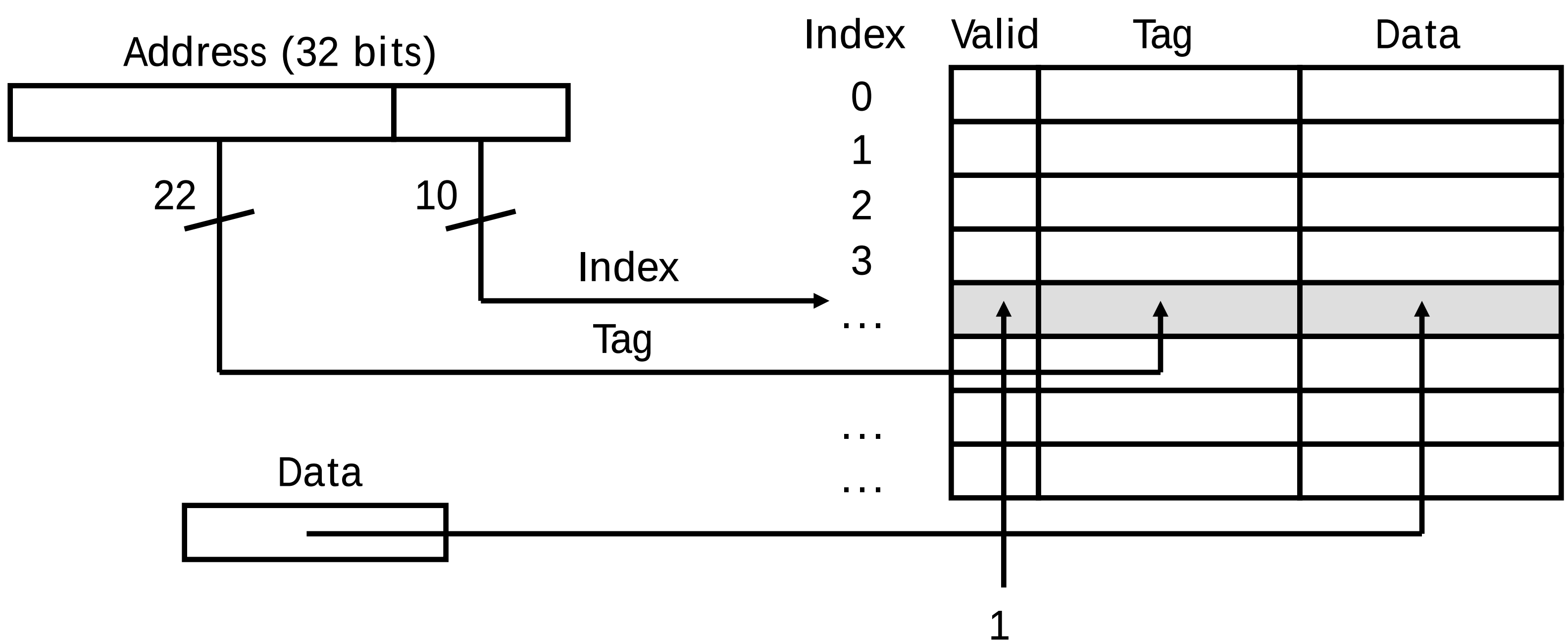
- The lowest k bits of the address will index a block in the cache
- If the block is valid and the tag matches the upper $(m - k)$ bits of the m-bit address, then that data will be sent to the CPU (cache hit)
- Otherwise (cache miss), data is read from main memory and
  - stored in the cache block specified by the lowest k bits of the address
  - the upper $(m - k)$ address bits are stored in the block's tag field
  - the valid bit is set to 1
- If our CPU implementations accessed main memory directly, their cycle times would have to be much larger
  - Instead we assume that most memory accesses will be cache hits, which allows us to use a shorter cycle time
- On a cache miss, the simplest thing to do is to stall the pipeline until the data from main memory can be fetched (and also copied into the cache)

What if the cache fills up?

- We answered this question implicitly on the last page!
  - A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data
  - This is a least recently used replacement policy, which assumes that older data is less likely to be requested than newer data

Loading a block into the cache

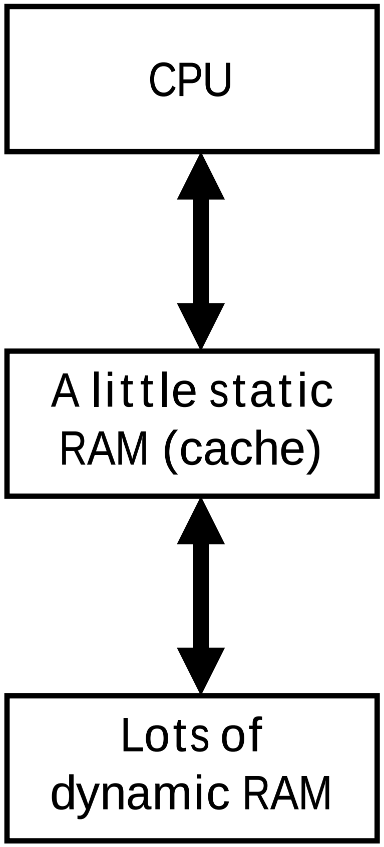
- After data is read from main memory, putting a copy of that data into the cache is straightforward
  - The lowest *k* bits of the address specify a cache block
  - The upper (*m* - *k*) address bits are stored in the block's tag field
  - The data from main memory is stored in the block's data field
  - The valid bit is set to 1



Memory System Performance

- Memory system performance depends on three important questions:
  - How long does it take to send data from the cache to the CPU?
  - How long does it take to copy data from memory into the cache?
  - How often do we have to access main memory?

- There are names for all of these variables:
  - The hit time is how long it takes data to be sent from the cache to the processor. This is usually fast, on the order of 1-3 clock cycles.
  - The miss penalty is the time to copy data from main memory to the cache. This often requires dozens of clock cycles (at least).
  - The miss rate is the percentage of misses.



Average memory access time

- The average memory access time, or AMAT, can then be computed

  AMAT = Hit time + (Miss rate x Miss penalty)

  This is just averaging the amount of time for cache hits and the amount of time for cache misses

- How can we improve the average memory access time of a system?
  - Obviously, a lower AMAT is better
  - Miss penalties are usually much greater than hit times, so the best way to lower AMAT is to reduce the miss penalty *or* the miss rate

- However, AMAT should only be used as a general guideline. Remember that execution time is still the best performance metric.

Performance example

- Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

AMAT = Hit time + (Miss rate x Miss penalty)
=
=

- How can we reduce miss rate?
  - One-byte cache blocks don't take advantage of spatial locality, which predicts that an access to one address will be followed by an access to a nearby address

- We'll see how to deal with this on after Spring Break

Summary

- Today we studied the basic ideas of caches

- By taking advantage of spatial and temporal locality, we can use a small amount of fast but expensive memory to dramatically speed up the average memory access time

- A cache is divided into many blocks, each of which contains a valid bit, a tag for matching memory addresses to cache contents, and the data itself

- Next, we'll look at some more advanced cache organizations