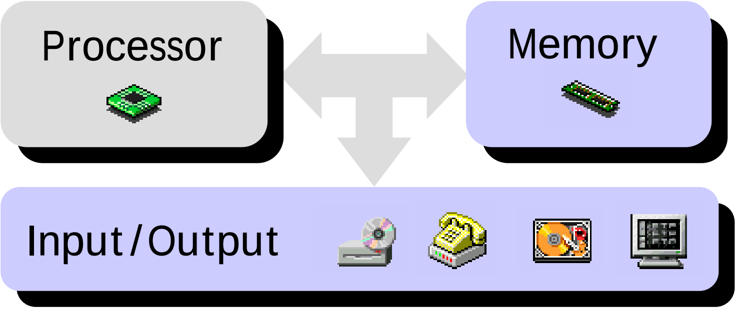
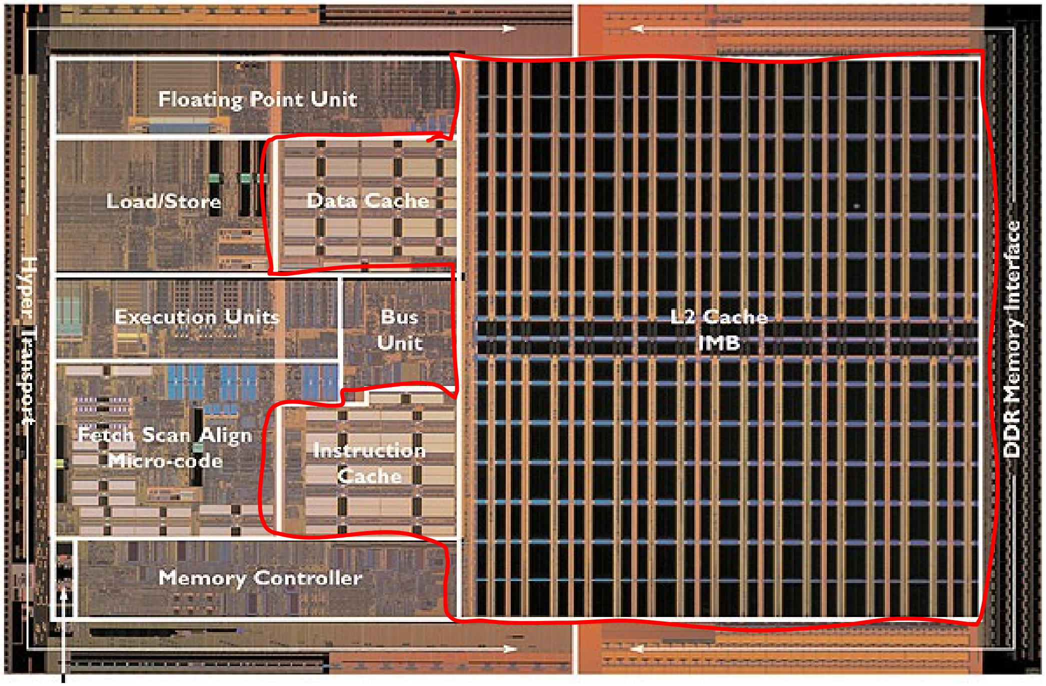
Welcome to Part 3: Memory Systems and I/O

- ☐ We've already seen how to make a fast processor. How can we supply the CPU with enough data to keep it busy?
- ☐ We will now focus on memory issues, which are frequently bottlenecks that limit the performance of a system.
- ☐ We'll start off by looking at memory systems for the next two weeks.



Cache introduction

- ☐ **Today we'll answer the following questions.**
  - **What are the challenges of building big, fast memory systems?**
  - **What is a cache?**
  - **Why caches work? (answer: locality)**
  - **How are caches organized?**
    - Where do we put things -and- how do we find them?



Large and fast

- Today's computers depend upon large and fast storage systems.
  - Large storage capacities are needed for many database applications, scientific computations with large data sets, video and music, and so forth.
  - Speed is important to keep up with our pipelined CPUs, which may access both an instruction and data in the same clock cycle. Things get even worse if we move to a superscalar CPU design.
- So far we've assumed our memories can keep up and our CPU can access memory in one cycle, but as we'll see that's a simplification.

Small or slow

- ❑ **Unfortunately there is a tradeoff between speed, cost and capacity.**

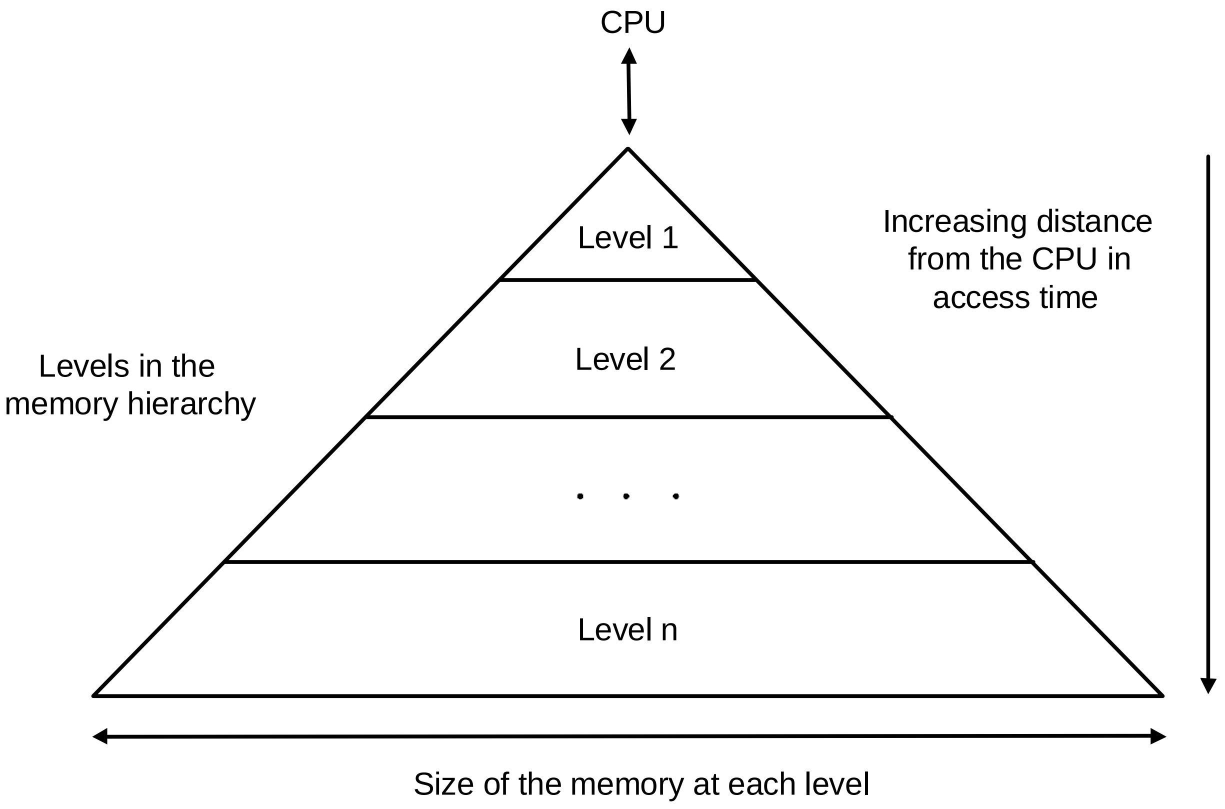
Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- ❑ **Fast memory is too expensive for most people to buy a lot of.**
- ❑ **But dynamic memory has a much longer delay than other functional units in a datapath. If every lw or sw accessed dynamic memory, we'd have to either increase the cycle time or stall frequently.**
- ❑ **Here are rough estimates of some current storage parameters.**

Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~$10	128KB-2MB
Dynamic RAM	100-200 cycles	~$0.20	128MB-4GB
Hard disks	10,000,000 cycles	~$0.001	20GB-200GB

How to Create the Illusion of Big and Fast

- **Memory hierarchy – put small and fast memories closer to CPU, large and slow memories further away**


CPU

Level 1

Level 2

. . .

Level n

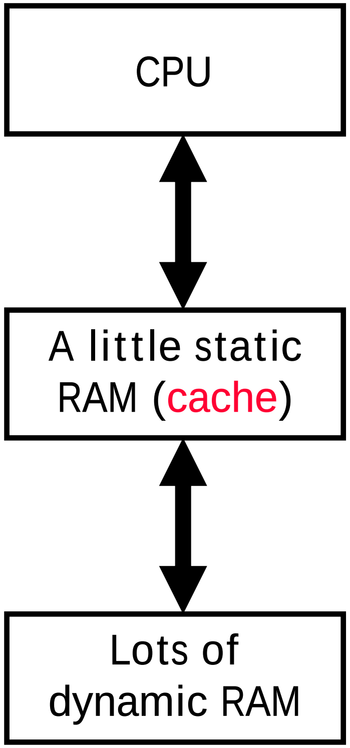
Increasing distance from the CPU in access time

Levels in the memory hierarchy

Size of the memory at each level

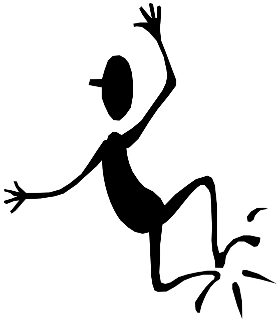
Introducing caches

- ☐ Introducing a cache – a small amount of fast, expensive memory.
  - The cache goes between the processor and the slower, dynamic main memory.
  - It 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.


CPU

A little static RAM (cache)

Lots of dynamic RAM



UNIVERSITY of CALIFORNIA Riverside COMPUTER SCIENCE & ENGINEERING

The principle of locality

- ☐ Why does the hierarchy work?

- ☐ Because most programs exhibit *locality*, which the cache can take advantage of.
  - The principle of **temporal locality** says that 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** says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

Temporal locality in *instructions*

- ❑ Loops are excellent examples of temporal locality in programs.
  - – The loop body will be executed many times.
  - – The computer will need to access those same few locations of the instruction memory repeatedly.
- ❑ For example:

```
Loop:  lw    $t0, 0($s1)
       add   $t0, $t0, $s2
       sw    $t0, 0($s1)
       addi  $s1, $s1, -4
       bne   $s1, $0, Loop
```

  - – Each instruction will be fetched over and over again, once on every loop iteration.

Temporal locality in *data*

- Programs often access the same variables over and over, especially within loops. Below, sum and i are repeatedly read and written.

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

- Commonly-accessed variables can sometimes be kept in registers, but this is not always possible.
  - There are a limited number of registers.
  - There are situations where the data must be kept in memory, as is the case with shared or dynamically-allocated memory.

Spatial locality in *instructions*

```
sub  $sp, $sp, 16
sw   $ra, 0($sp)
sw   $s0, 4($sp)
sw   $a0, 8($sp)
sw   $a1, 12($sp)
```

- **Nearly every program exhibits spatial locality, because instructions are usually executed in sequence — if we execute an instruction at memory location *i*, then we will probably also execute the next instruction, at memory location *i+1*.**
- **Code fragments such as loops exhibit *both* temporal and spatial locality.**

Spatial locality in *data*

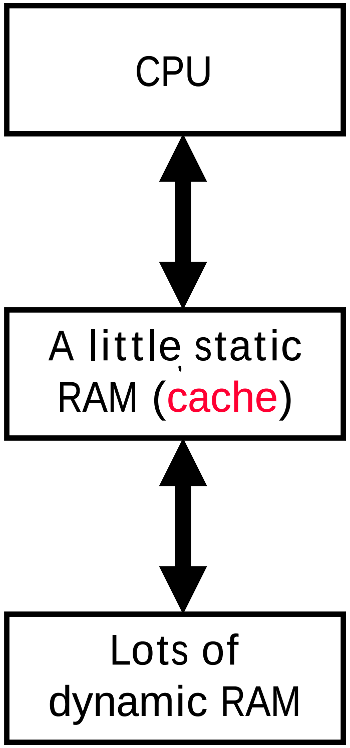
- **Programs often access data that is stored contiguously.**
  - **Arrays, like a in the code on the top, are stored in memory contiguously.**
  - **The individual fields of a record or object like employee are also kept contiguously in memory.**

```
    sum = 0;
    for (i = 0; i < MAX; i++)
        sum = sum + a[i];
```

```
employee.name = “Homer Simpson”;
employee.boss = “Mr. Burns”;
employee.age = 45;
```

How caches take advantage of temporal locality

- ☐ **The first time the processor reads from an address in main memory, a copy of that data is also stored in the cache.**
  - **The next time that same address is read, we can use the copy of the data in the cache *instead* of accessing the slower dynamic memory.**
  - **So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster.**
- ☐ **This takes advantage of temporal locality—commonly accessed data is stored in the faster cache memory.**

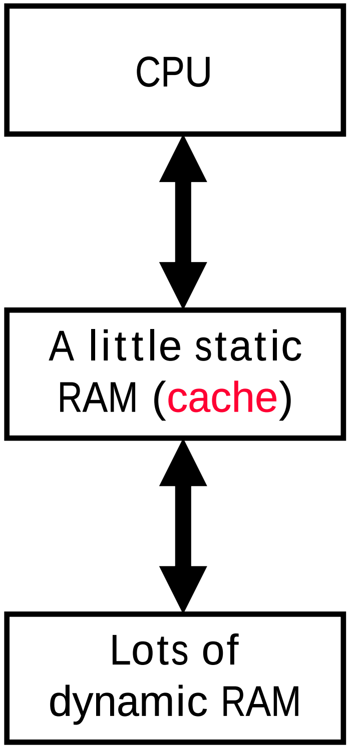

CPU

A little static RAM (cache)

Lots of dynamic RAM

How caches take advantage of spatial locality

- When the CPU reads location *i* from main memory, a copy of that data is placed in the cache.
- But instead of just copying the contents of location *i*, we can copy ***several*** values into the cache at once, such as the four bytes from locations *i* through *i* + 3.
  - If the CPU later does need to read from locations *i* + 1, *i* + 2 or *i* + 3, it can access that data from the cache and not the slower main memory.
  - For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.
- Again, the initial load incurs a performance penalty, but we're gambling on spatial locality and the chance that the CPU will need the extra data.



CPU

A little static RAM (cache)

Lots of dynamic RAM

Definitions: Hits and misses

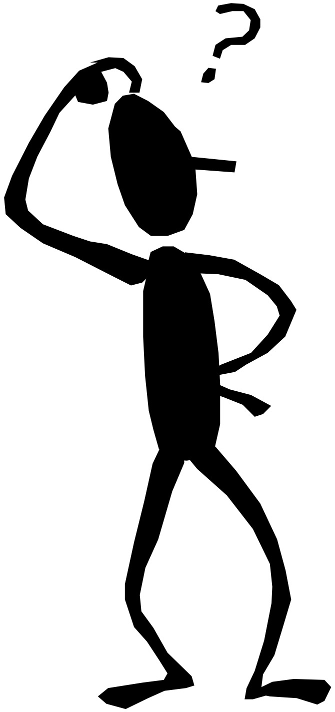
- A **cache hit** occurs if the cache contains the data that we're looking for. Hits are good, because the cache can return the data much faster than main memory.
- A **cache miss** occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.
- There are two basic measurements of cache performance.
  - The **hit rate** is the percentage of memory accesses that are handled by the cache.
  - The **miss rate** (1 - hit rate) is the percentage of accesses that must be handled by the slower main RAM.
- Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster.
- In future lectures, we'll talk more about cache performance.
  - Today we'll talk about implementation!

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.
- Here is an example cache with eight blocks, each holding one byte.

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

Four important questions

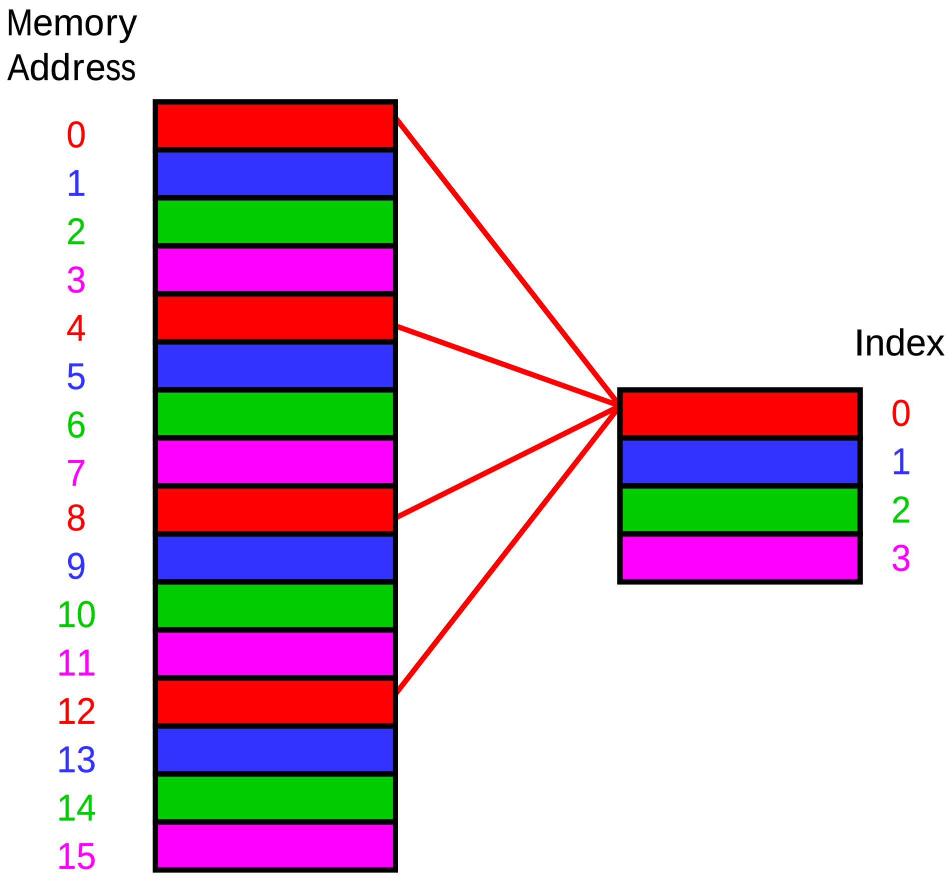


1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to 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.
- ☐ For example, on the right is a 16-byte main memory and a 4-byte cache (four 1-byte blocks).
- ☐ Memory locations 0, 4, 8 and 12 all map to cache block 0.
- ☐ Addresses 1, 5, 9 and 13 map to cache block 1, etc.
- ☐ How can we compute this mapping?



UNIVERSITY of CALIFORNIA Riverside COMPUTER SCIENCE & ENGINEERING

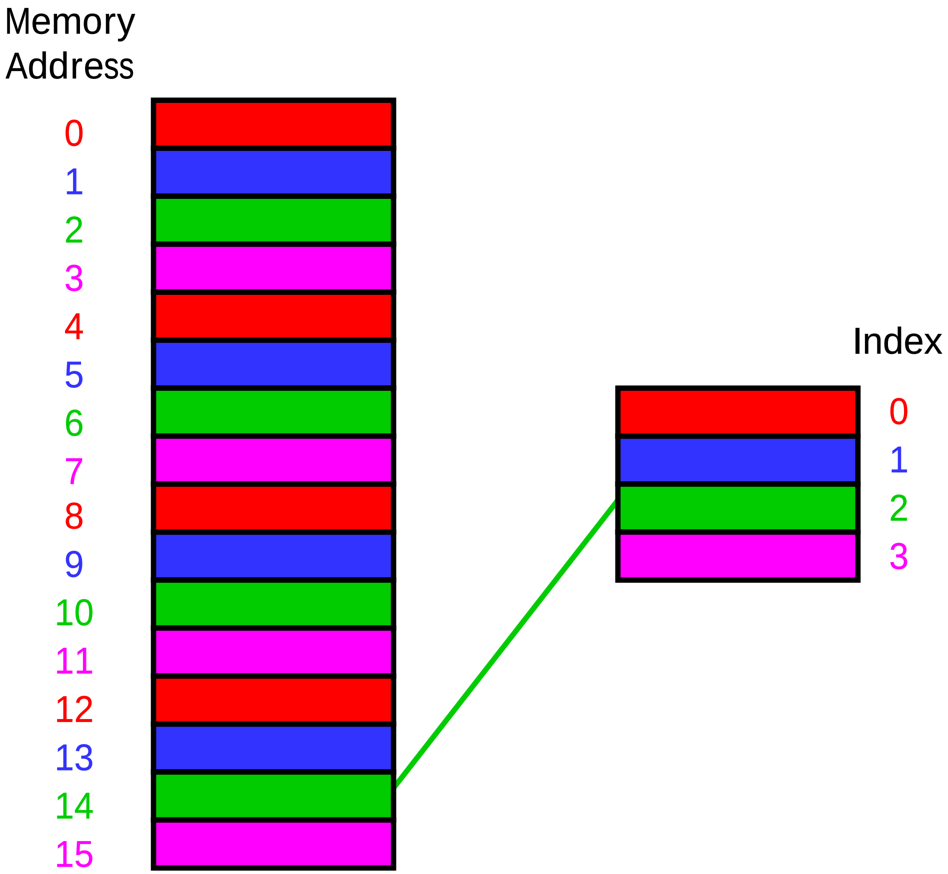
It’s all divisions…

- ❑ One way to figure out which cache block a particular memory address should go to is to use the mod (remainder) operator.
- ❑ If the cache contains 2^k blocks, then the data at memory address *i* would go to cache block index

$$i \bmod 2^k$$

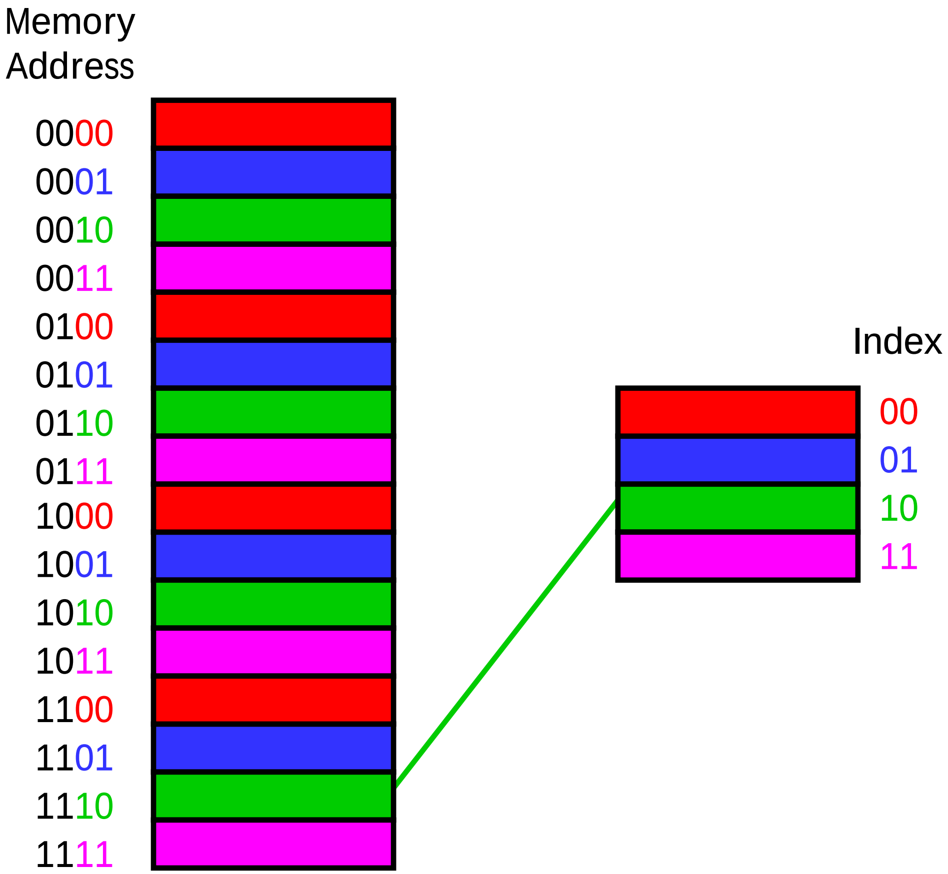
- ❑ For instance, with the four-block cache here, address 14 would map to cache block 2.

14 mod 4 = 2



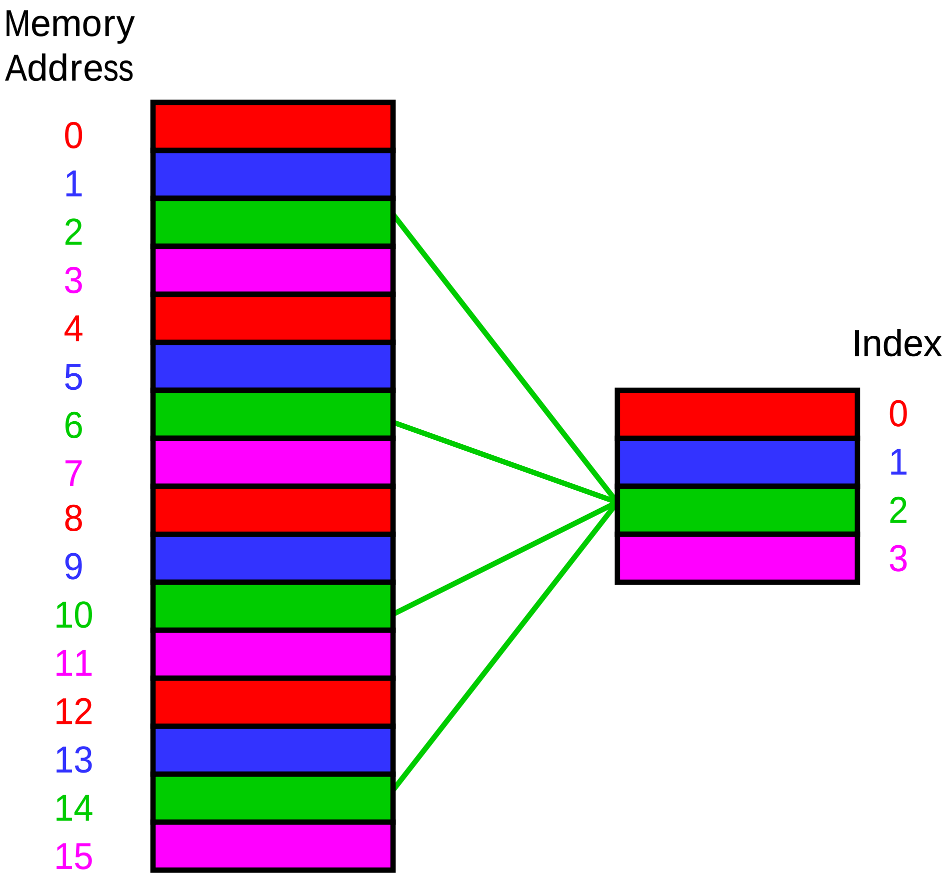
…or least-significant bits

- ☐ An equivalent way to find the placement of a memory address in the cache is to look at the least significant *k* bits of the address.
- ☐ With our four-byte cache we would inspect the two least significant bits of our memory addresses.
- ☐ Again, you can see that address 14 (1110 in binary) maps to cache block 2 (10 in binary).
- ☐ Taking the least *k* bits of a binary value is the same as computing that value mod 2^k.



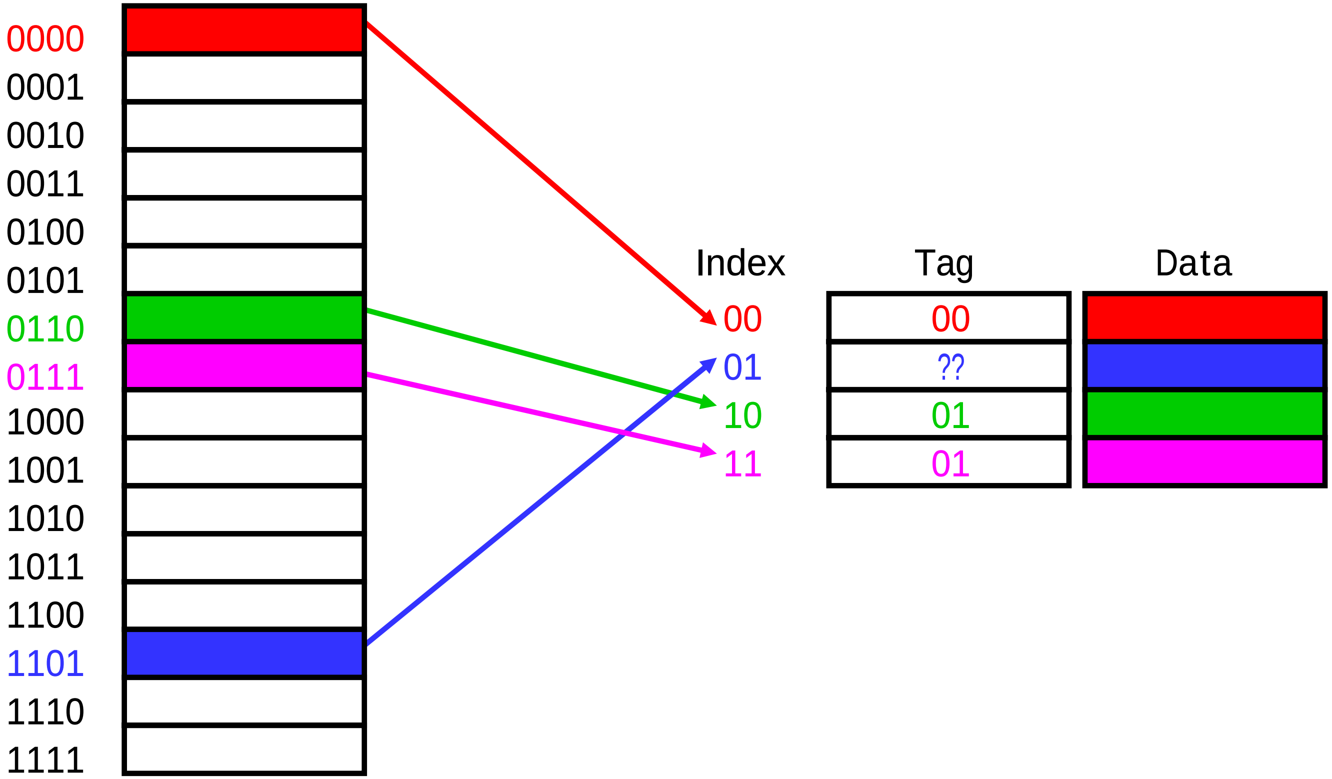
How can we find data in the cache?

- ☐ The second question was how to determine whether or not the data we’re interested in is already stored in the cache.
- ☐ If we want to read memory address *i*, we can use the mod 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?
- ☐ For instance, cache block 2 could contain data from addresses 2, 6, 10 *or* 14.



Adding tags

- We need to add **tags** to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.



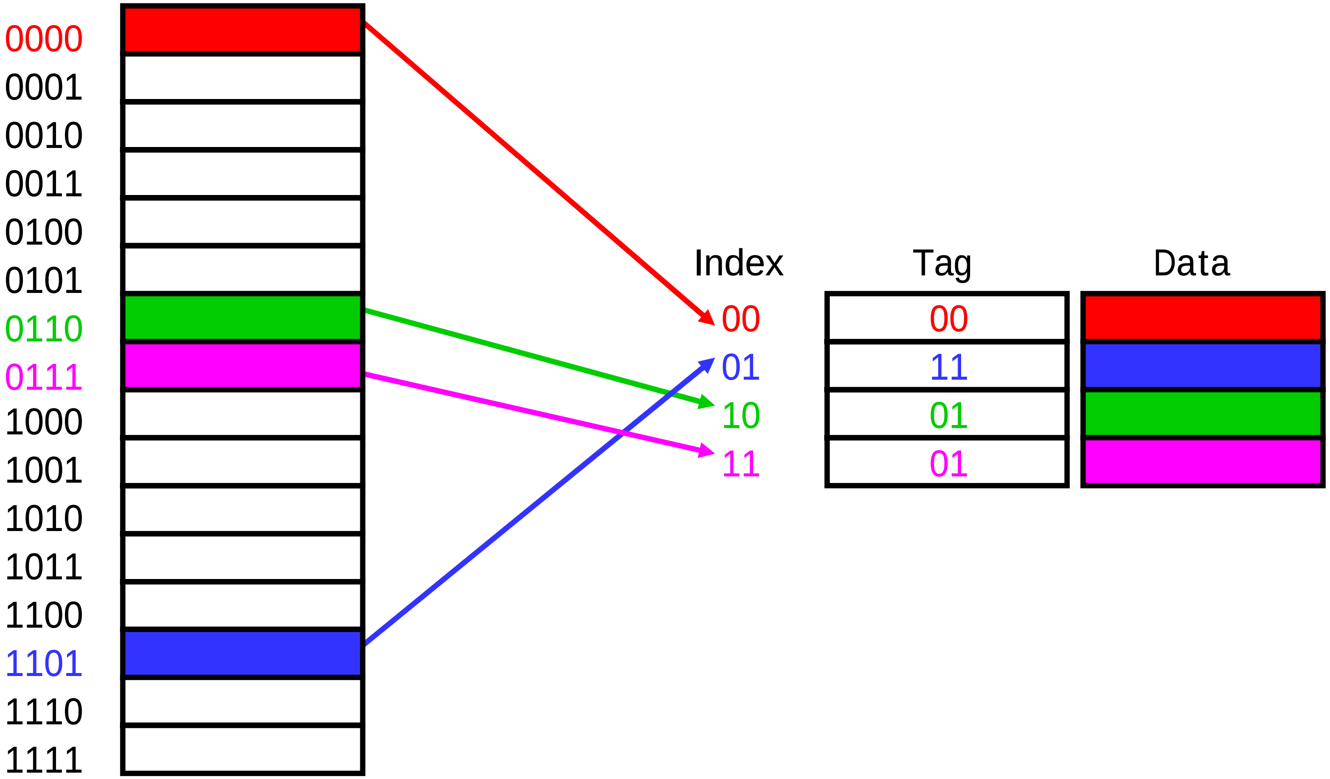
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Index	Tag	Data
00	00	🟥
01	??	🟦
10	01	🟩
11	01	🟪

COMPUTER
SCIENCE & ENGINEERING

Adding tags

- We need to add **tags** to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.



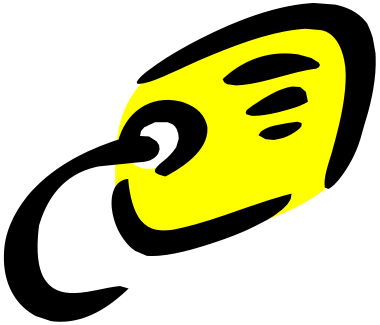
0000 🟥
0001
0010
0011
0100
0101
0110 🟩
0111 🟪
1000
1001
1010
1011
1100
1101 🟦
1110
1111

Index	Tag	Data
00	00	🟥
01	11	🟦
10	01	🟩
11	01	🟪

Figuring out what’s in the cache

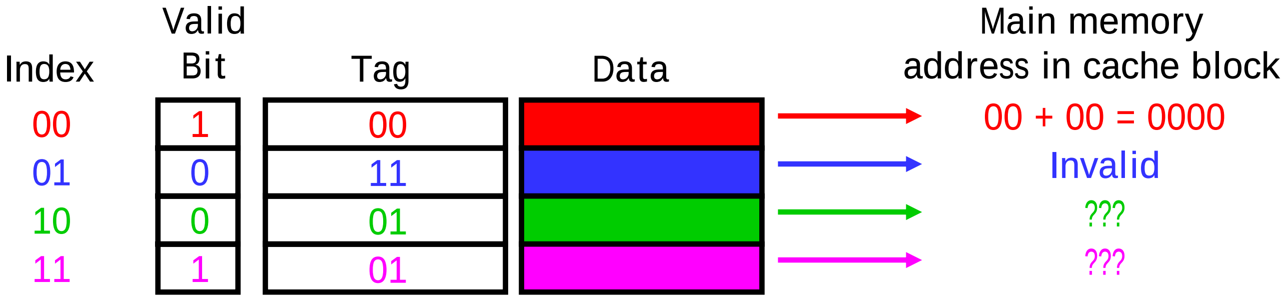
- ☐ **Now we can tell exactly which addresses of main memory are stored in the cache, by concatenating the cache block tags with the block indices.**

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	🟩	???
11	1	01	🟪	???

- 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.

One more detail: the valid bit

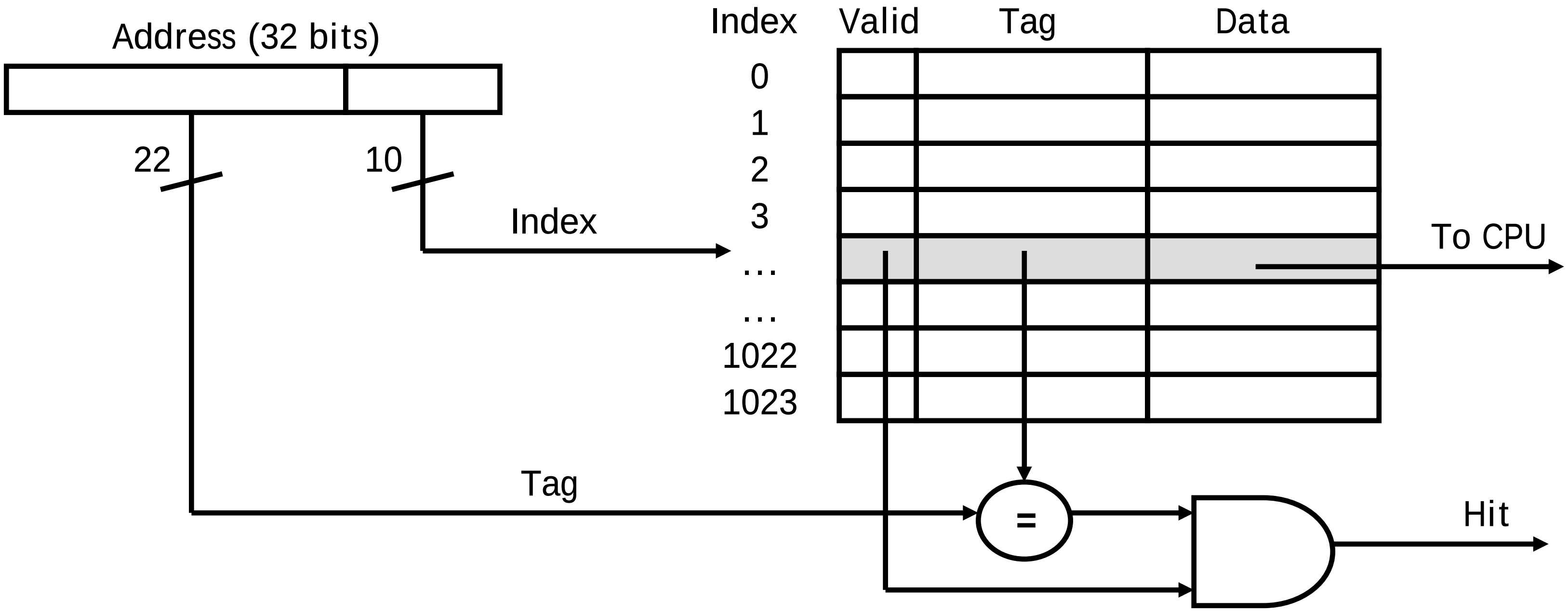
- **When started, the cache is empty and does not contain valid data.**
- **We should account for this by adding a valid bit for each cache block.**
  - **When the system is initialized, all the valid bits are set to 0.**
  - **When data is loaded into a particular cache block, the corresponding valid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

- ☐ When the CPU tries to read from memory, the address will be sent to a **cache controller**.
  - 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.
- ☐ Here is a diagram of a 32-bit memory address and a 2^{10}-byte cache.



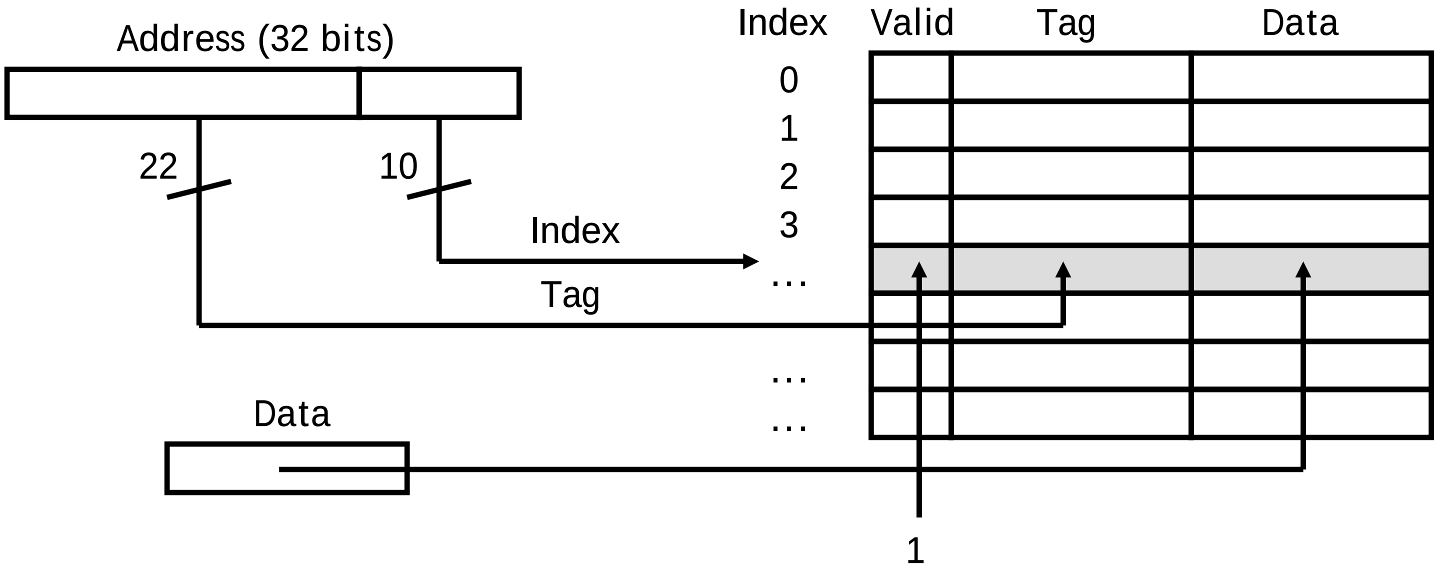
What happens on a cache miss

- **The delays that we've been assuming for memories (e.g., 2ns) are really assuming cache hits.**
  - **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.**

- **However, a much slower main memory access is needed 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).**

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.**



COMPUTER
SCIENCE & ENGINEERING

What if the cache fills up?

- Our third question was what to do if we run out of space in our cache, or if we need to reuse a block for a different memory address.
- 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.
- We'll see a few other policies next week.

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 week we'll look at some more advanced cache organizations and see how to measure the performance of memory systems.