CprE 381: Computer Organization and Assembly Level Programming

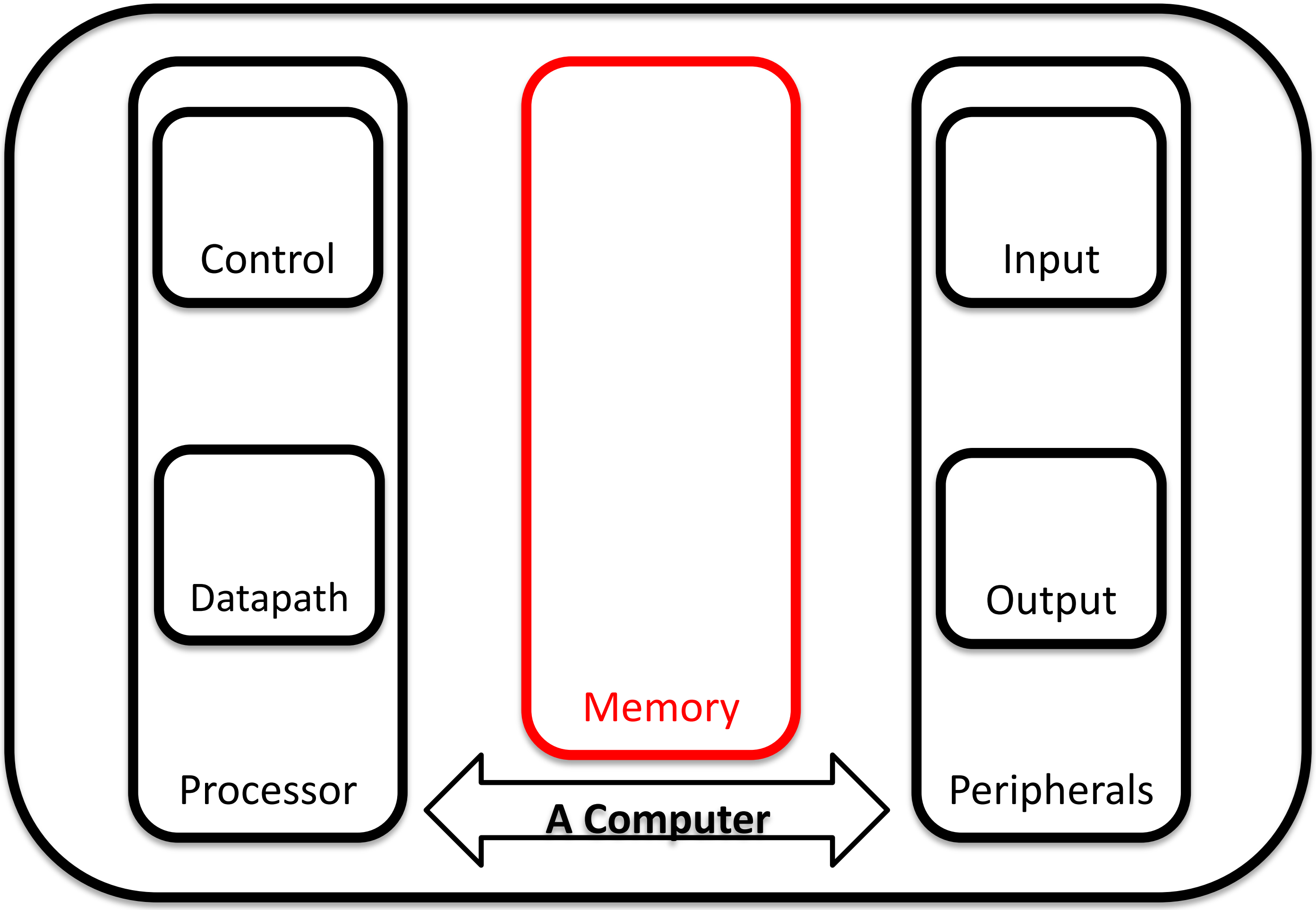
Cache Design

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- Project Part 3a
  - Due this week
  - **WARNING**: Much easier that Part 3b!
- Will be out remainder of this week:
  - But first, "why did you decide to Teach?"
    - Teaching is 40% of my job.
  - Funding tasks
  - Prof. Zambreno will lecture

Remember the System View!



Review: Small or Slow

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

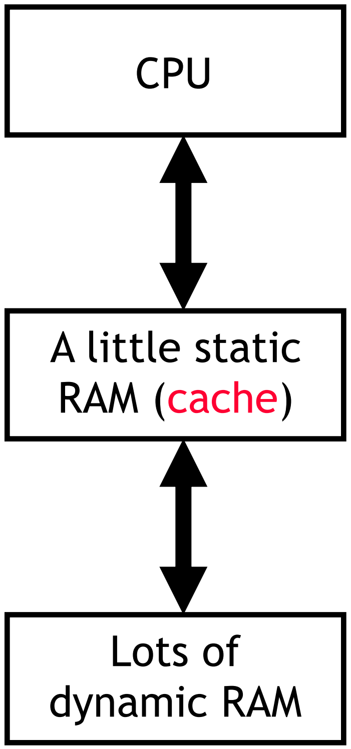
Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- 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	~$1	128KB-128MB
Dynamic RAM	100-200 cycles	~$0.005	256MB-512GB
Hard disks	10,000,000 cycles	~$0.00005	512GB-10TB

Review: Introducing Caches

- Wouldn't it be nice if we could find a balance between fast and cheap memory?
- We do this by introducing a cache, which is 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



Review: The Principle of Locality

- It's usually difficult or impossible to figure out what data will be "most frequently accessed" before a program actually runs, which makes it hard to know what to store into the small, precious cache memory.
- But in practice, 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.

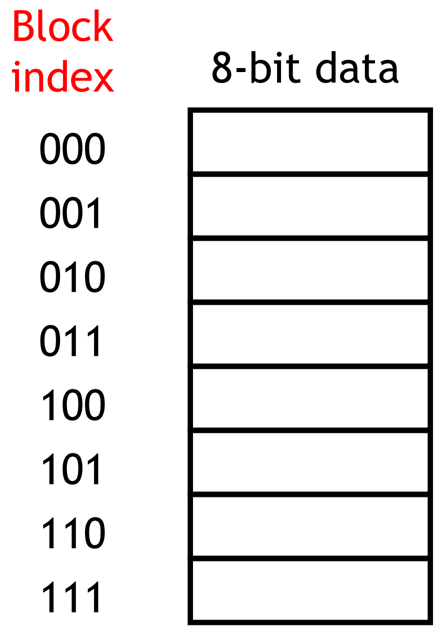
Definitions: Hits and Misses

- 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.

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



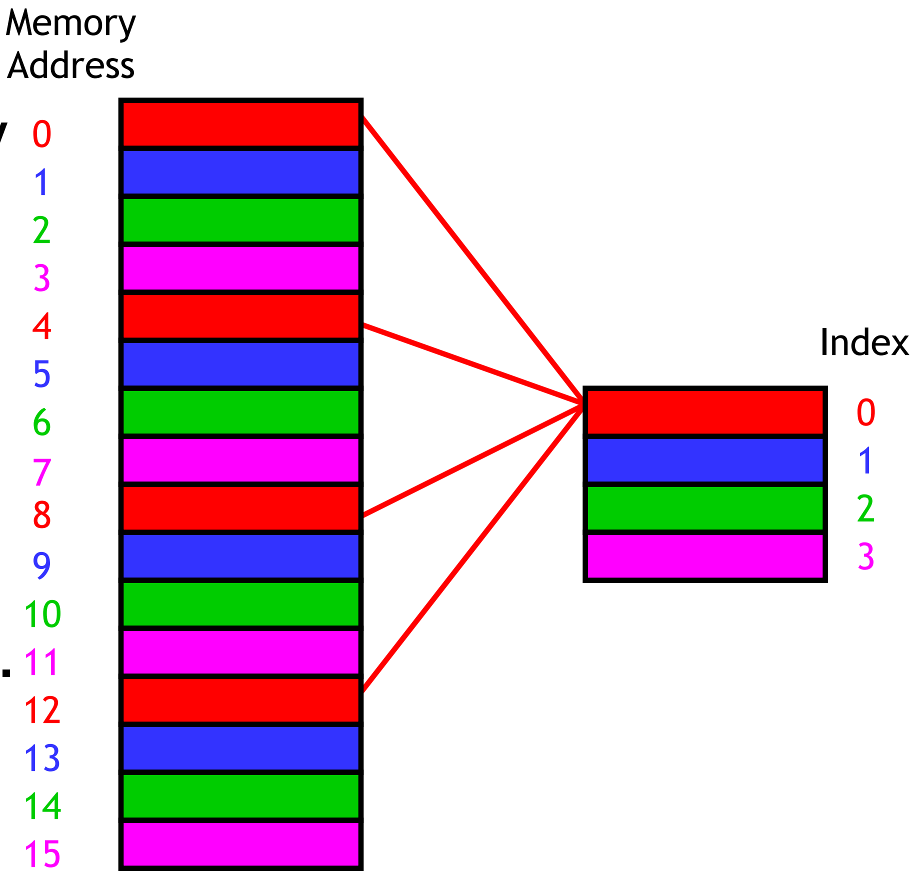
Four Important Questions

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!

How Should We Cache Data?

- 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?



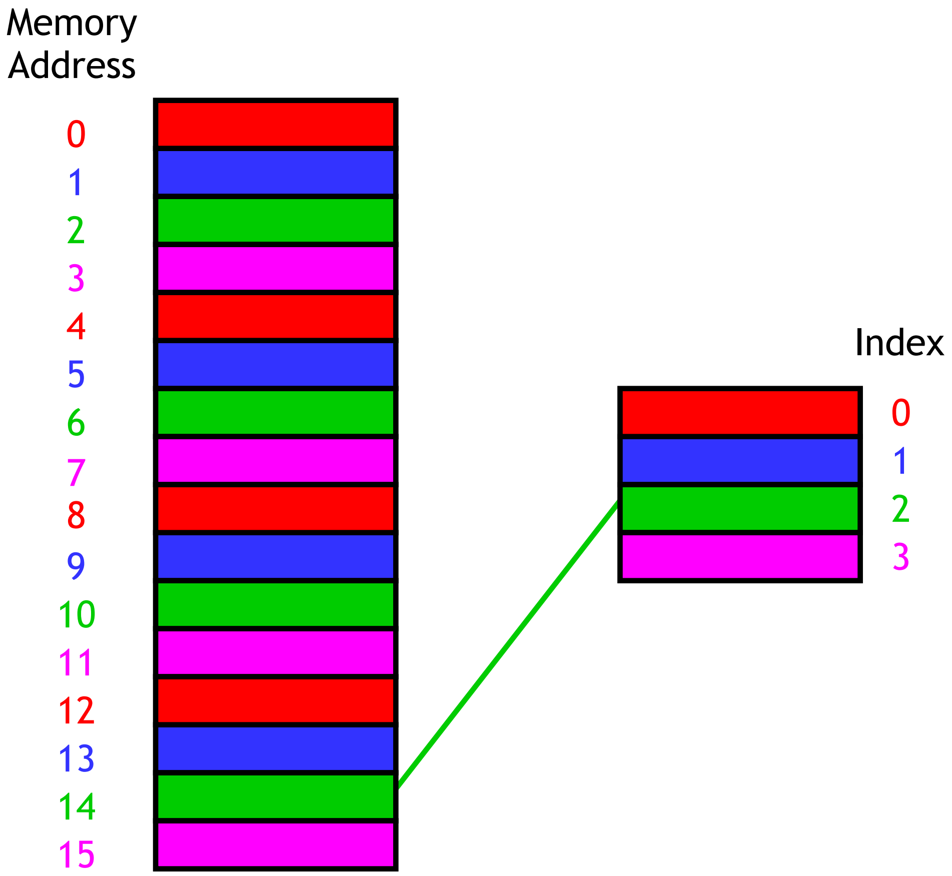
It’s All Divisions

- 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

$$i \bmod 2^k$$

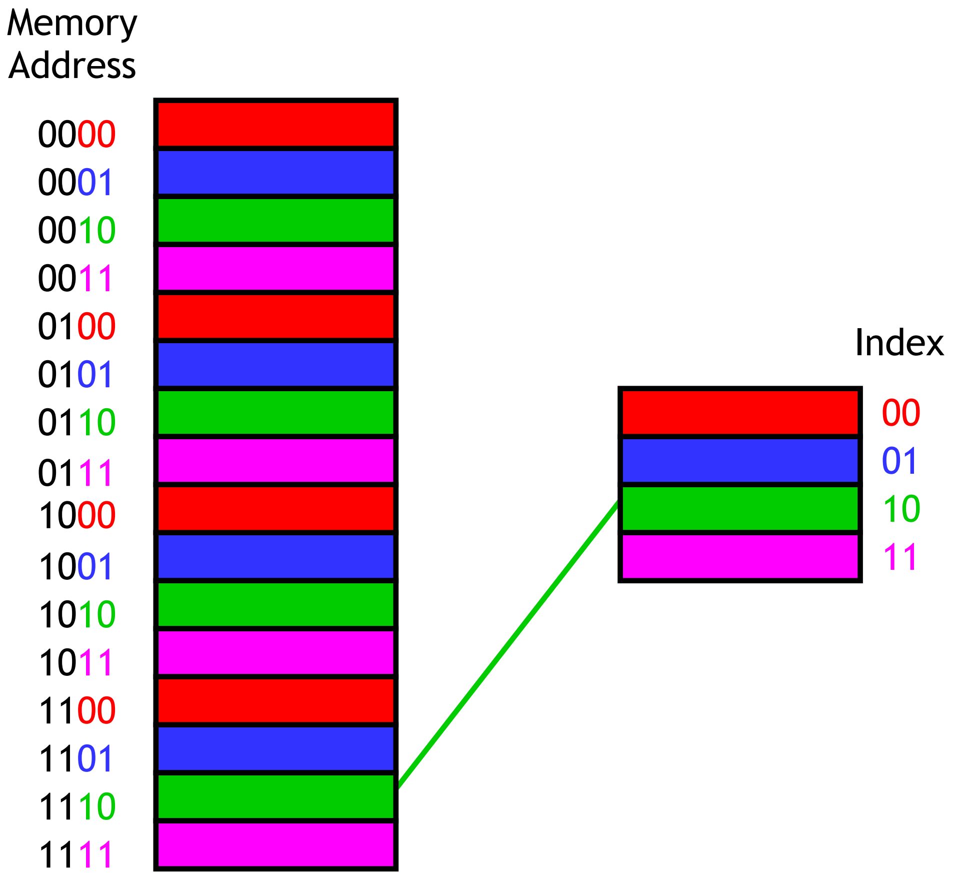
- For instance, with the four-block cache here, address 14 would map to cache block 2.

$$14 \bmod 4 = 2$$



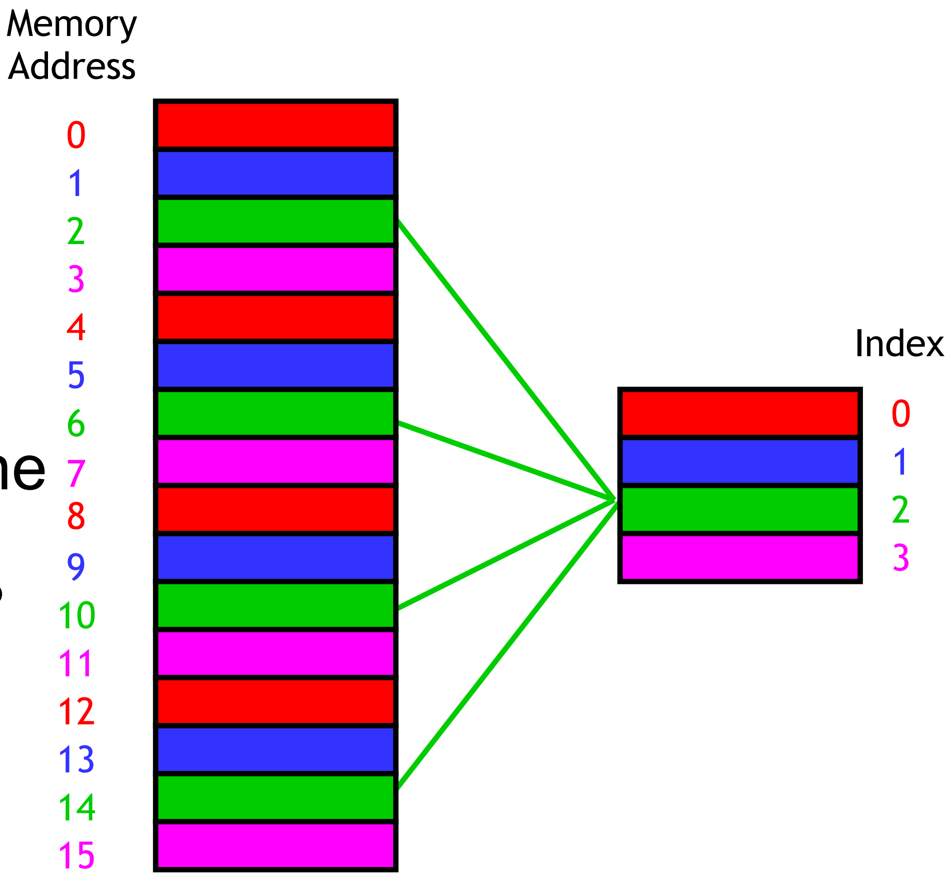
...or Least Significant Bits

- 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.



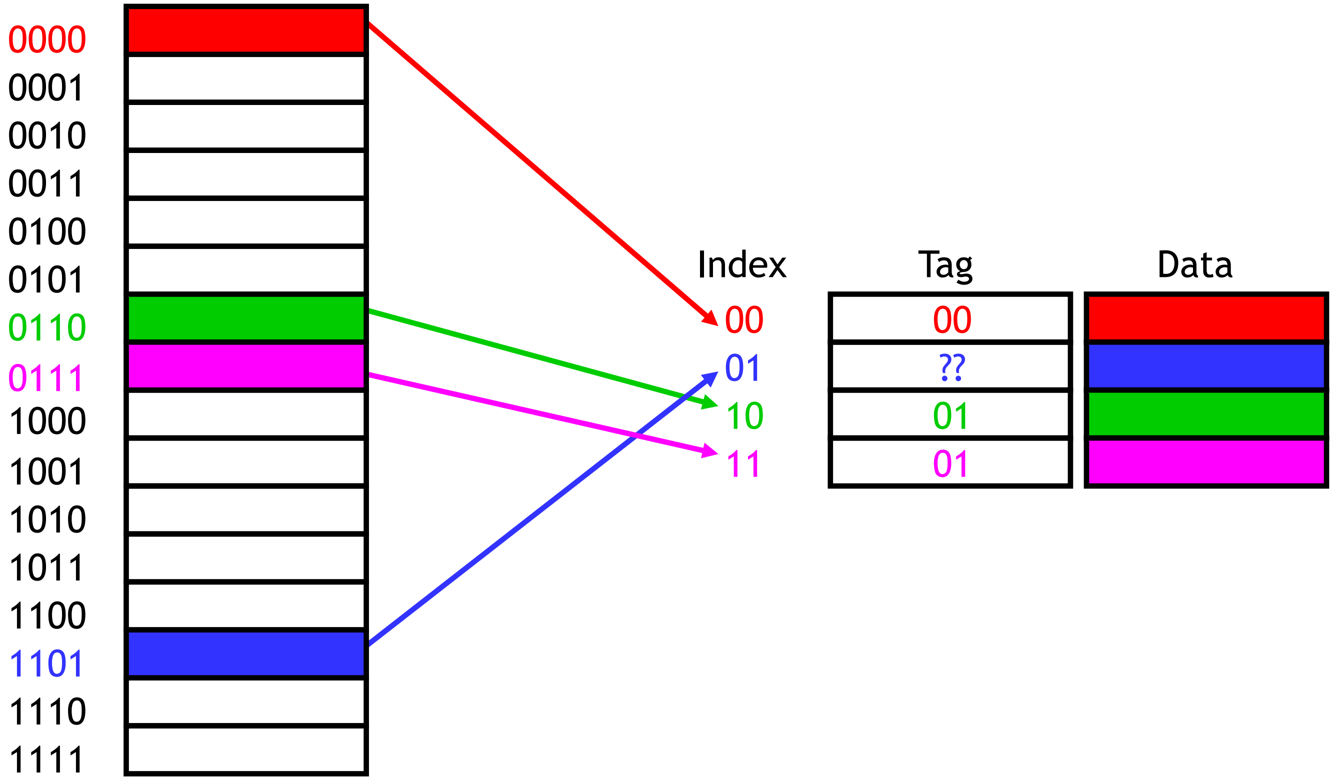
How to Find Data in the Cache?

- 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.



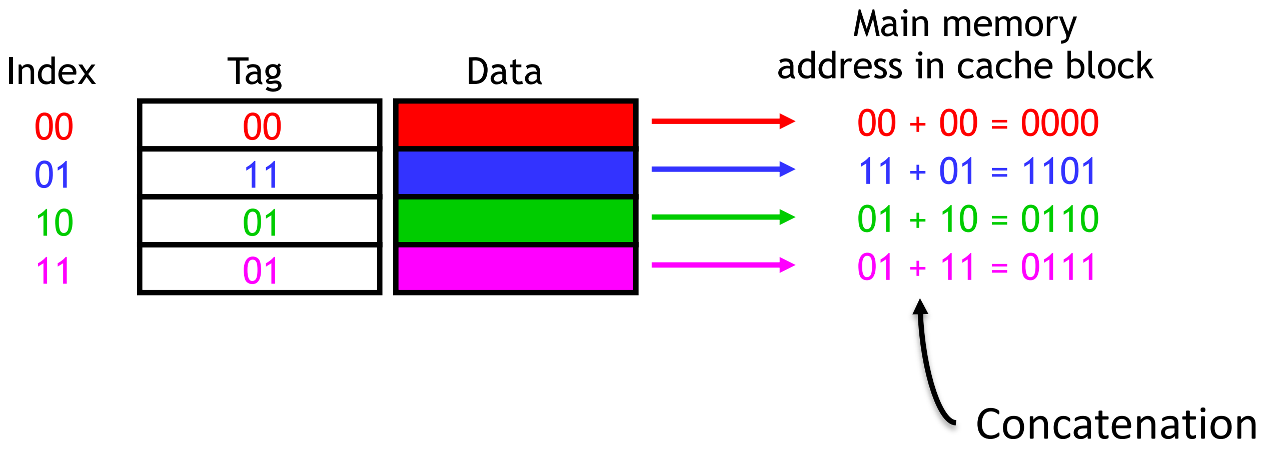
Adding Tags

- 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.



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.



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.

In-Class Exercise

- Consider two cache structures, both initially empty:

Index	Valid Bit	Tag	Data
00	0	00	🟥
01	0	00	🟦
10	0	00	🟩
11	0	00	🟪

Index	Valid Bit	Tag	Data
000	0	0	🟥
001	0	0	🟦
010	0	0	🟩
011	0	0	🟪
100	0	0	🟦
101	0	0	🟧
110	0	0	⬜
111	0	0	🟨

In-class Assessment!

Access Code: $$

Note: sharing access code to those outside of classroom or using access code while outside of classroom is considered cheating

In-Class Exercise

- Consider two cache structures, both initially empty:

Index	Valid Bit	Tag	Data
00	0	00	🟥
01	0	00	🟦
10	0	00	🟩
11	0	00	🟪

Index	Valid Bit	Tag	Data
000	0	0	🟥
001	0	0	🟦
010	0	0	🟩
011	0	0	🟪
100	0	0	🟦
101	0	0	🟧
110	0	0	⬜
111	0	0	🟨

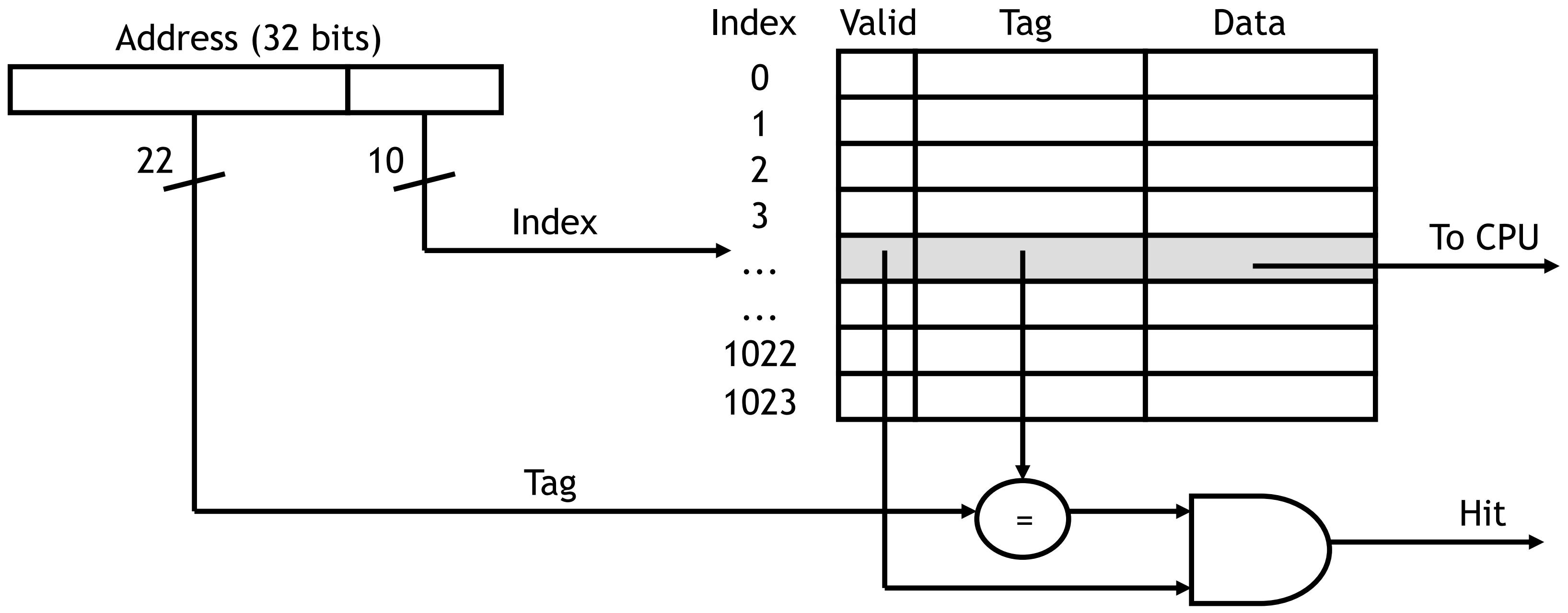
- For the following memory accesses, what are the hit/miss results for the two different caches?

```
Mem[0], Mem[5], Mem[10], Mem[7], Mem[4], Mem[0],
Mem[10], Mem[4], Mem[6], Mem[10], Mem[6]
```

- Is there an access pattern for which the smaller cache will perform better than the larger one?

Preview: 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.

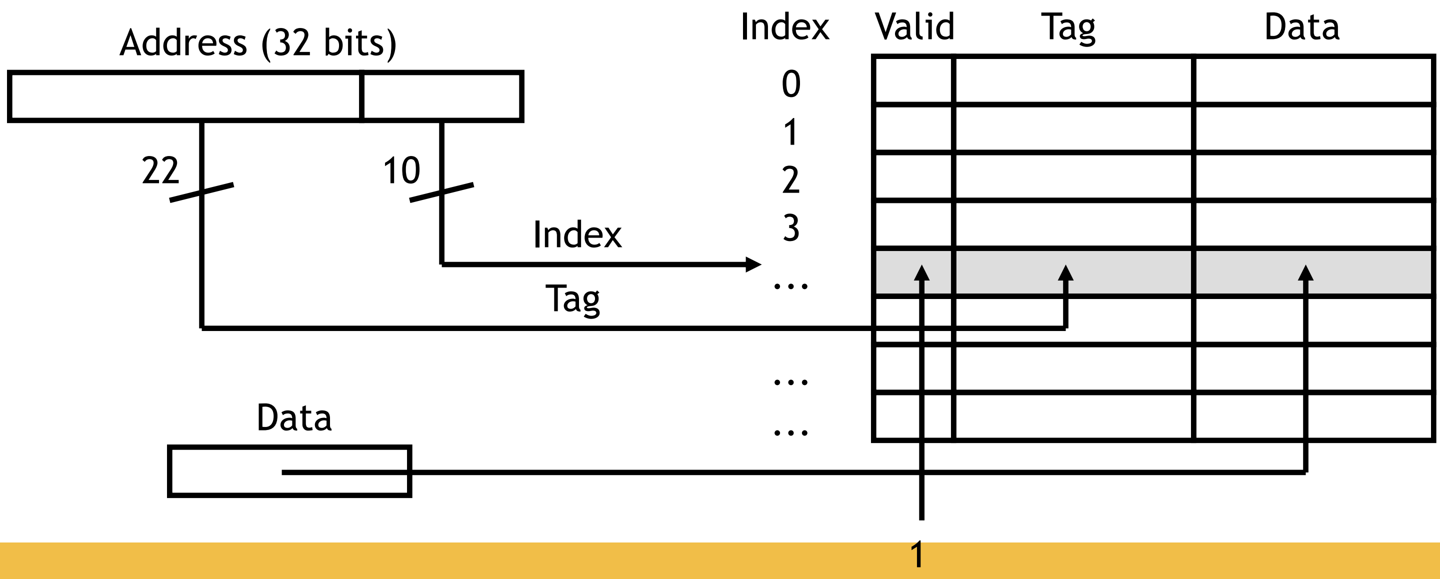


What Happens on a Cache Miss

- 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



Preview: 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 slide!
  - 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.

- This question gets a little more interesting next lecture

Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - Akhilesh Tyagi (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)
  - Morgan Kaufmann