Friday 13 Unannounced Quiz, NOT

On a sheet of paper, write one (or more) MIPS assembler instructions (no pseudo ops, please), which after execution, have made no change to the machine state (visible to a programmer).

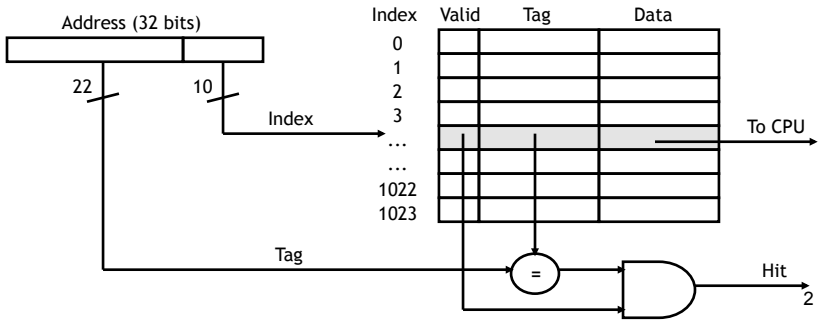
Total Time: 2 minutes

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, that data is sent to the CPU

For a 32-bit memory address and a 2^{10}-byte cache w/ 1 byte blks:



How big is the cache?

For a byte-addressable machine with 16-bit addresses with a cache with the following characteristics:

It is direct-mapped (as discussed last time)

Each block holds one byte

The cache index is the four least significant bits

Two questions:

How many blocks does the cache hold?

4-bit index -> 2^4 = 16 blocks

How many bits of storage are required to build the cache (*e.g.*, for the data array, tags, etc.)?

tag size = 12 bits (16 bit address - 4 bit index)

(12 tag bits + 1 valid bit + 8 data bits) x 16 blocks = 21 bits x 16 = 336 bits

3

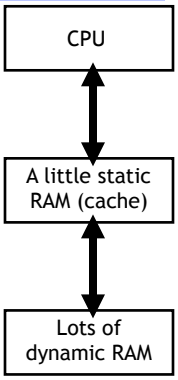
Memory System Performance

To examine the performance of a memory system, we need to focus on a couple of important factors

- 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



4

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

5

Performance example

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

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

6

Performance example

Assume data accesses only. 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)
= 1 cycle + (3% x 20 cycles)
= 1.6 cycles

If the cache was perfect and never missed, the AMAT would be one cycle. But even with just a 3% miss rate, the AMAT here increases 1.6 times!

How can we reduce miss rate?

7

Spatial locality

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.

What can we do?

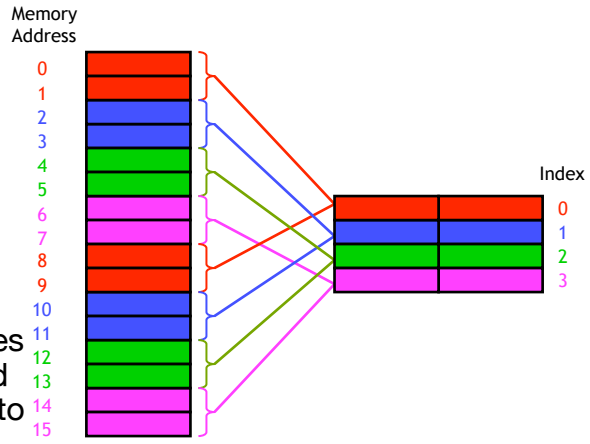
8

Spatial locality

What we can do is make the cache block size larger than one byte

Here we use two-byte blocks, so we can load the cache with two bytes at a time

If we read from address 12, the data in addresses 12 *and* 13 would both be copied to cache block 2



9

Block addresses

Now where should data be placed in the cache?

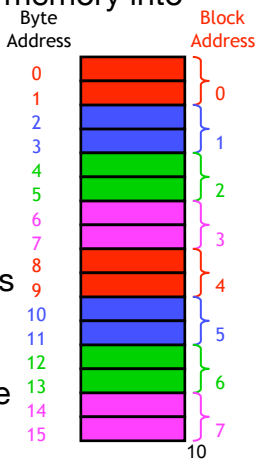
It's time for block addresses! If the cache block size is 2^n bytes, we can conceptually split the main memory into 2^n-byte chunks too

To determine the block address of a byte address *i*, you can do the integer division

$$i / 2^n$$

Our example has two-byte cache blocks, so we can think of a 16-byte main memory as an "8-block" main memory instead

For instance, memory addresses 12 and 13 both correspond to block address 6, since 12 / 2 = 6 and 13 / 2 = 6



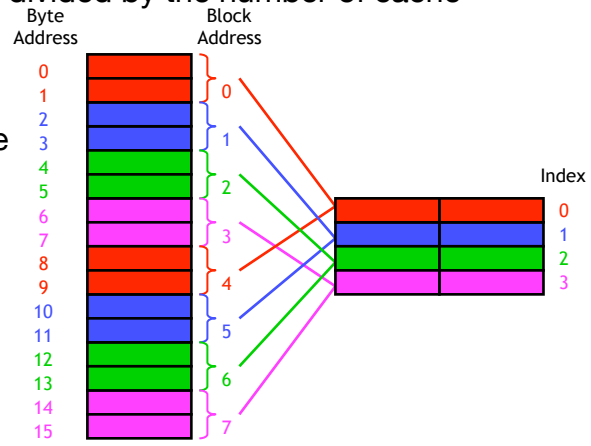
10

Cache mapping

Once you know the block address, you can map it to the cache as before: find the remainder when the block address is divided by the number of cache blocks.

In our example, memory block 6 belongs in cache block 2, since 6 mod 4 = 2.

This corresponds to placing data from memory *byte* addresses 12 and 13 into cache block 2.



11

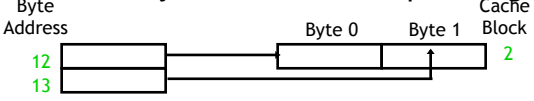
Data placement within a block

When we access one byte of data in memory, we'll copy its entire *block* into the cache, hopefully to take advantage of spatial locality

In our example, if a program reads from byte address 12 we'll load all of memory block 6 (both addresses 12 and 13) into cache block 2.

Note byte address 13 corresponds to the *same* memory block address! So a read from address 13 will also cause memory block 6 (addresses 12 and 13) to be loaded into cache block 2.

To make things simpler, byte *i* of a memory block is always stored in byte *i* of the corresponding cache block.



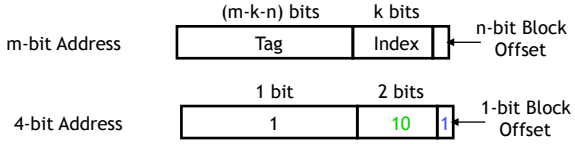
12

Locating data in the cache

Let's say we have a cache with 2^k blocks, each containing 2^n bytes

We can determine where a byte of data belongs in this cache by looking at its address in main memory

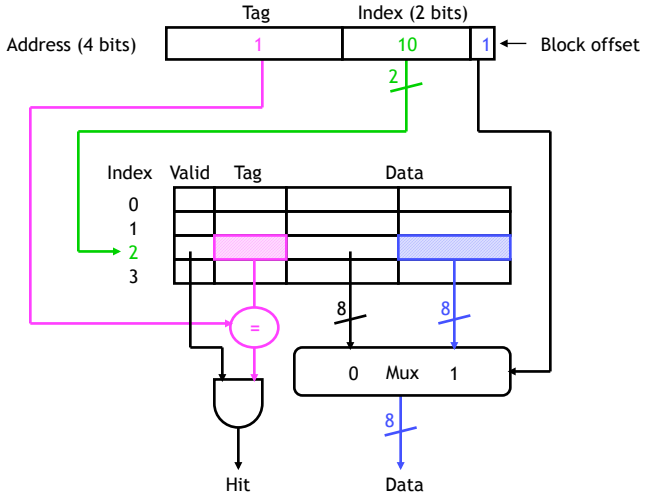
- k bits of the address will select one of the 2^k cache blocks
- The lowest n bits are now a block offset that decides which of the 2^n bytes in the cache block will store the data



Our example used a 2^2-block cache with 2^1 bytes per block. Thus, memory address 13 (1101) would be stored in byte 1 of cache block 2

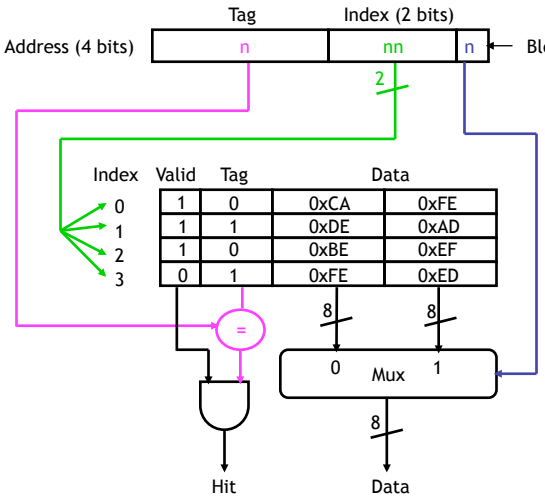
13

A picture



14

An exercise



For the addresses below, what byte is read from the cache (or is there a miss)?

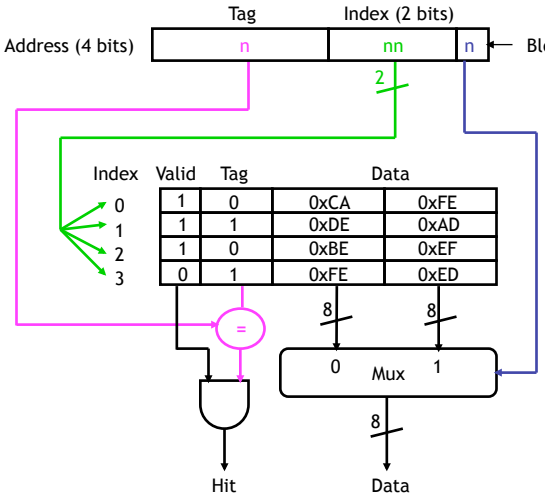
1010

1110

0001

1101

An exercise



For the addresses below, what byte is read from the cache (or is there a miss)?

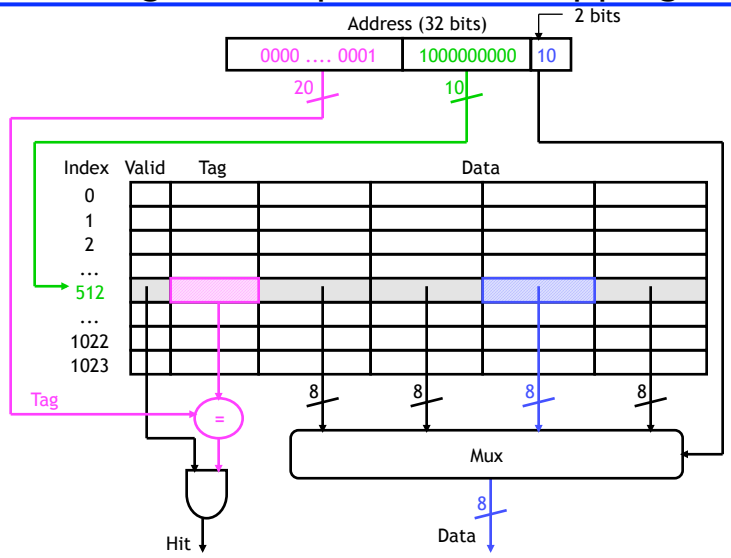
1010 (0xDE)

1110 (miss, invalid)

0001 (0xFE)

1101 (miss, bad tag)

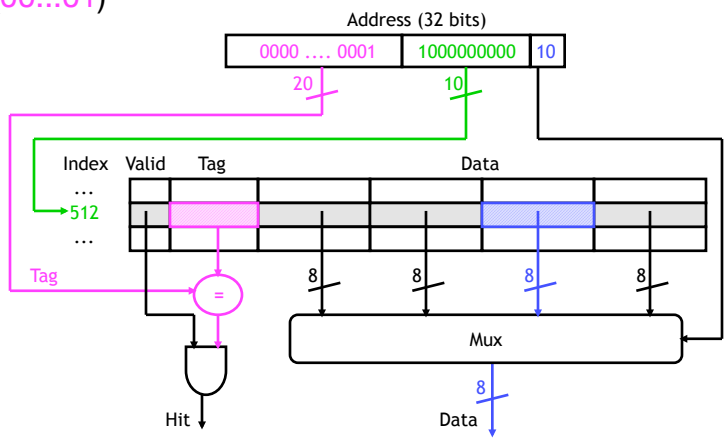
A larger example cache mapping



17

What goes in the rest of cache block?

The other three bytes of that cache block come from the same memory block, whose addresses must all have the same index (1000000000) and the same tag (00...01)



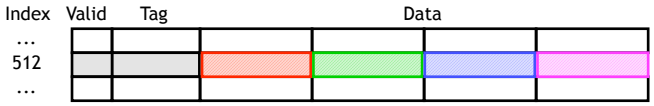
18

The rest of that cache block

Again, byte *i* of a memory block is stored into byte *i* of the corresponding cache block

- In our example, memory block 1536 consists of byte addresses 6144 to 6147. So bytes 0-3 of the cache block would contain data from address 6144, 6145, 6146 and 6147 respectively
- You can also look at the lowest 2 bits of the memory address to find the block offsets

Block offset	Memory address	Decimal
00	00..01 1000000000 00	6144
01	00..01 1000000000 01	6145
10	00..01 1000000000 10	6146
11	00..01 1000000000 11	6147



19

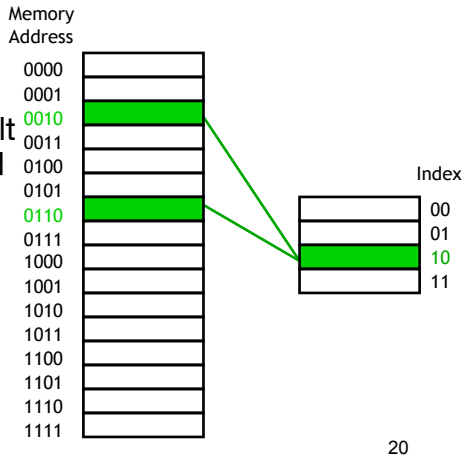
Disadvantage of direct mapping

The direct-mapped cache is easy: indices and offsets can be computed with bit operators, because each memory address belongs in exactly one block.

However, this isn't really flexible. If a program uses addresses 2, 6, 2, 6, 2, ..., then each access will result in a cache miss and a load into cache block 2

This cache has four blocks, but direct mapping might not let us use all of them

This can result in more misses than we might like



20

A fully associative cache

A fully associative cache permits data to be stored in *any* cache block, instead of forcing each memory address into one particular block

- ❖ When data is fetched from memory, it can be placed in *any* unused block of the cache
- ❖ This way we'll never have a conflict between two or more memory addresses which map to a single cache block

In the previous example, we might put memory address 2 in cache block 2, and address 6 in block 3. Then subsequent repeated accesses to 2 and 6 would all be hits instead of misses

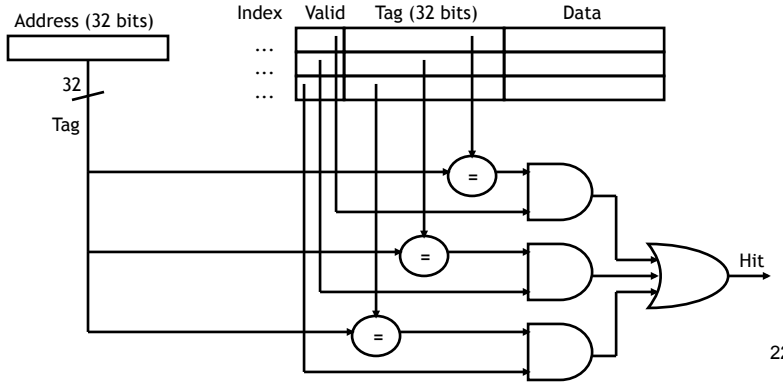
If all the blocks are already in use, it's usually best to replace the least recently used one, assuming that if it hasn't been used it in a while, it won't be needed again anytime soon

21

The price of full associativity

However, a fully associative cache is expensive to build

- Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size.
- Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!



22

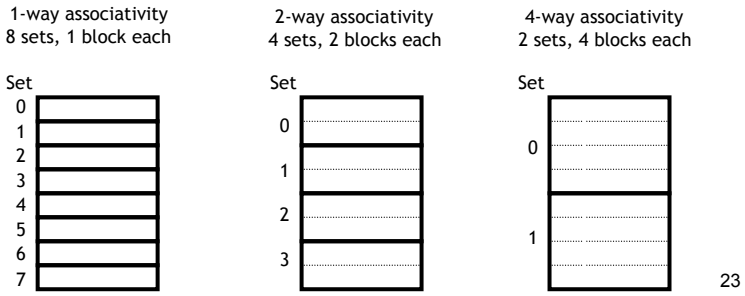
Set associativity

An intermediate possibility is a set-associative cache

- The cache is divided into *groups* of blocks, called sets
- Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set

If each set has 2^x blocks, it's an 2^x-way associative cache

Here are several possible organizations of an 8-blk cache

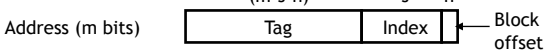


23

Locating a set associative block

We can determine where a memory address belongs in an associative cache in a similar way as before.

If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows.



Our arithmetic computations now compute a set index, to select a *set* within the cache instead of an individual block.

Block Offset = Memory Address mod 2^n

Block Address = Memory Address / 2^n

Set Index = Block Address mod 2^s

24

Placement in set-associative caches

Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?

6195 in binary is 00...0110000 011 0011

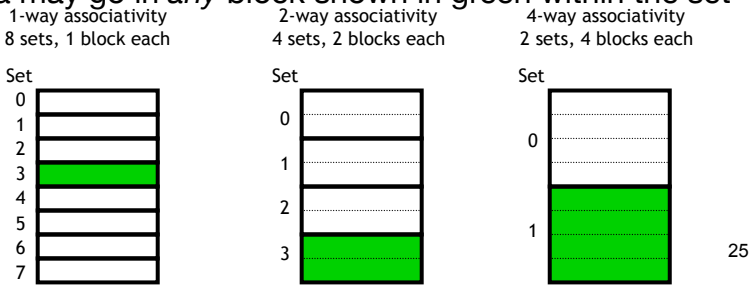
Each block has 16 B, so the least 4 bits are block offset

- 1-way cache, the next three bits (011) are the set index

2-way cache, the next two bits (11) are the set index.

4-way cache, the next one bit (1) is the set index.

Data may go in *any* block shown in green within the set

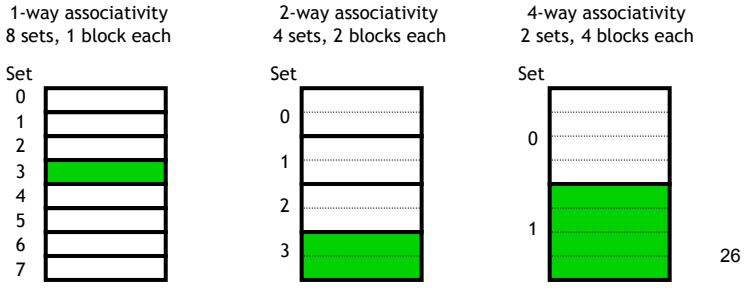


Block replacement

Any empty block in the correct set may be used for data

If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before

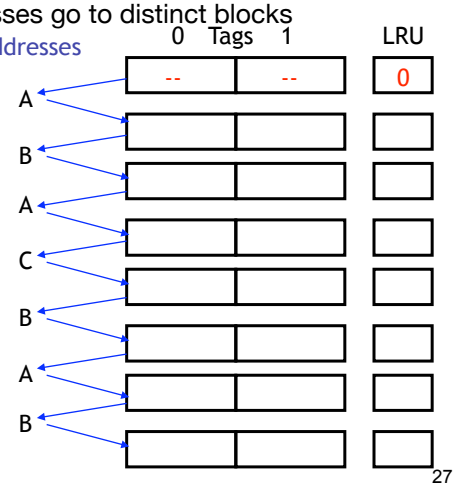
For highly associative caches, it's expensive to keep track of what's really the least recently used block, so some approximations are used. We won't get into the details.



LRU example

Assume a fully-associative cache with 2 blocks, which of the following memory references miss in the cache

- assume distinct addresses go to distinct blocks



27

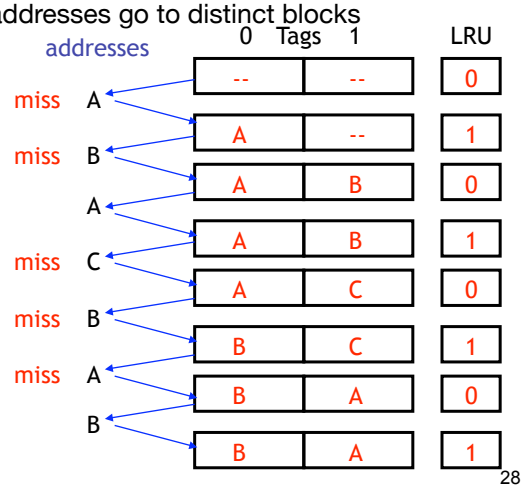
LRU example

Assume a fully-associative cache with 2 blocks, which of the following memory references miss in the cache

- assume distinct addresses go to distinct blocks

On a miss, we replace the LRU.

On a hit, we just update the LRU.



28