LECTURE 11

Memory Hierarchy

MEMORY HIERARCHY

When it comes to memory, there are two universally desirable properties:

- Large Size: ideally, we want to never have to worry about running out of memory.
- Speed of Access: we want the process of accessing memory to take as little time as possible.

But we cannot optimize both of these properties at the same time. As our memory size increases, the time to find a memory location and access it grows as well.

The goal of designing a memory hierarchy is to simulate having unlimited amounts of fast memory.

LOCALITY

To simulate these properties, we can take advantage of two forms of *locality*.

- Temporal Locality: if an item is referenced, it will tend to be referenced again soon.
  - The location of the variable `counter` may be accessed frequently in a loop.
  - A branching instruction may be accessed repeatedly in a given period of time.
- Spatial Locality – if an item is referenced, items whose addresses are close by will tend to be referenced soon.
  - If we access the location of `A[0]`, we will probably also be accessing `A[1]`, `A[2]`, etc.
  - Sequential instruction access also exhibits spatial locality.

MEMORY HIERARCHY

A *memory hierarchy,* consisting of multiple levels of memory with varying speed and size, exploits these principles of locality.

- Faster memory is more expensive per bit, so we use it in smaller quantities.
- Slower memory is much cheaper so we can afford to use a lot of it.

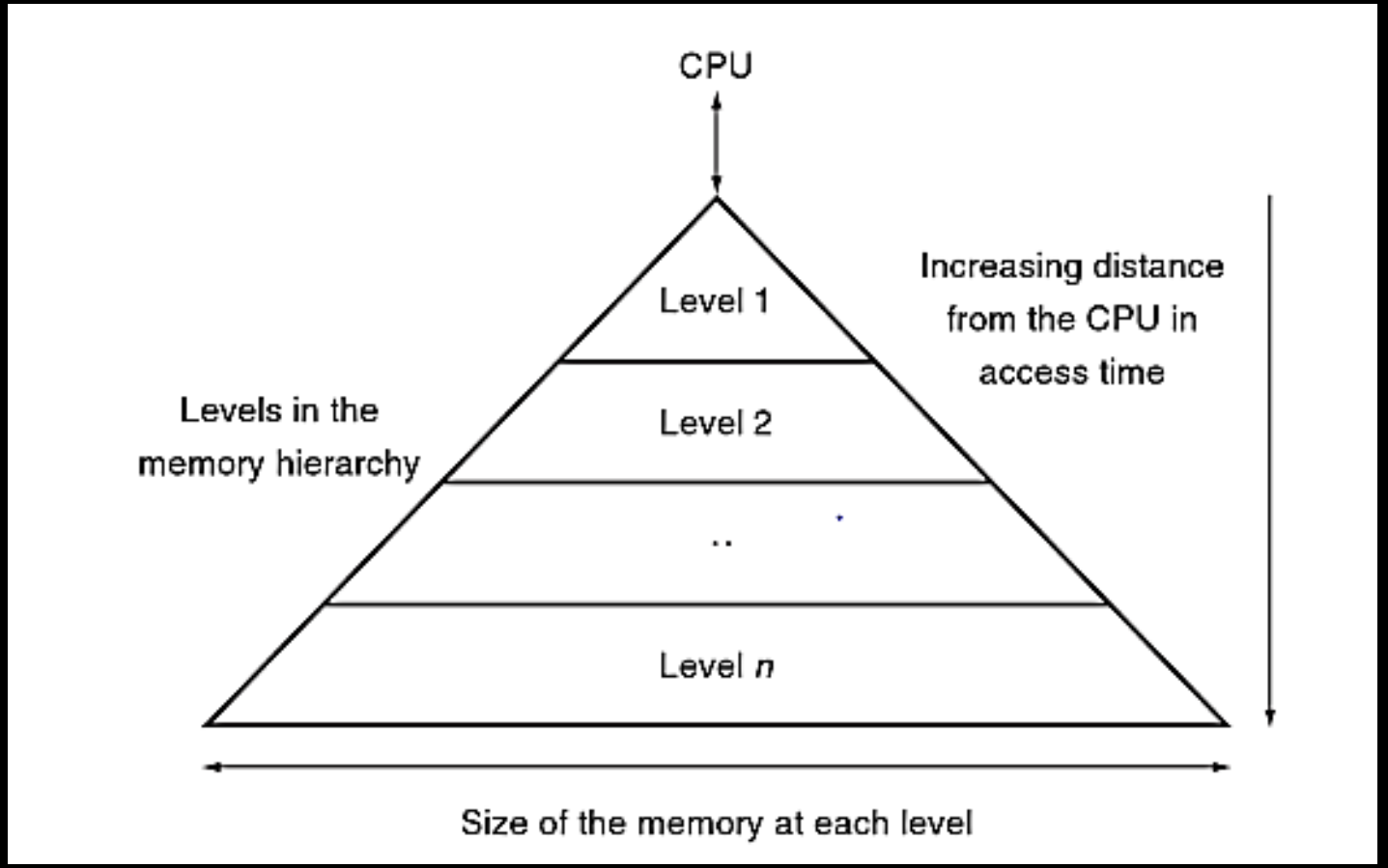
The goal is to, whenever possible, keep references in the fastest memory. However, we also want to minimize our overall memory cost.

MEMORY HIERARCHY

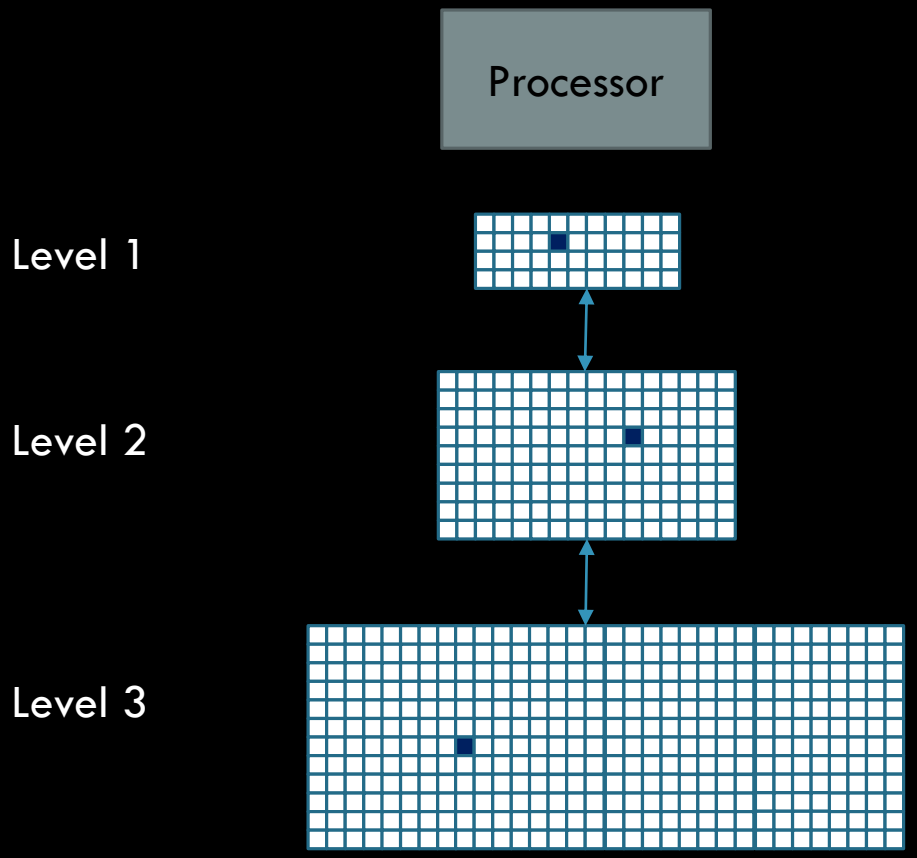
All data in a level is typically also found in the next largest level.

We keep the smallest, faster memory unit closest to the processor.

The idea is that our access time during a running program is defined primarily by the access time of the level 1 unit. But our memory capacity is as large as the level n unit.

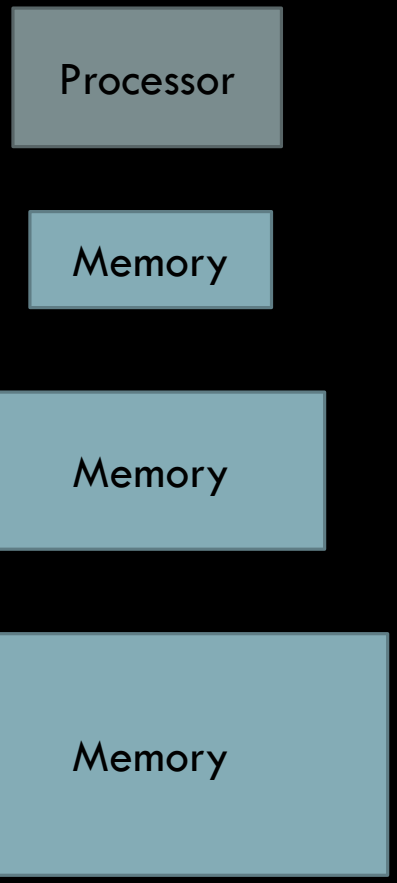


MEMORY HIERARCHY



The unit of data that is transferred between two levels is fixed in size and is called a *block*, or a *line*.

MEMORY HIERARCHY



Speed	Size	Cost ($/bit)	Technology
Fastest	Smallest	Highest	SRAM
			DRAM
Slowest	Biggest	Lowest	Magnetic Disk

MEMORY HIERARCHY

There are four technologies that are used in a memory hierarchy:

- SRAM (Static Random Access Memory): fastest memory available. Used in memory units close to the processor called *caches*. Volatile.
- DRAM (Dynamic Random Access Memory): mid-range. Used in main memory. Volatile.
- Flash: Falls between DRAM and disk in cost and speed. Used as non-volatile memory in personal mobile devices.
- Magnetic Disk: slowest memory available. Used as non-volatile memory in a server or PC.

MEMORY HIERARCHY

Technology	Typical Access Time	$ per GiB in 2016
SRAM	0.5-5 ns	$400 - $1000
DRAM	50-70 ns	$3 - $5
Flash	5,000-50,000 ns	$0.30 - $0.50
Magnetic Disk	5,000,000 – 20,000,000 ns	$0.05 - $0.10

MEMORY HIERARCHY TERMS

- Hit: item found in a specified level of the hierarchy.
- Miss: item not found in a specified level of the hierarchy.
- Hit time: time required to access the desired item in a specified level of the hierarchy (includes the time to determine if the access is a hit or a miss).
- Miss penalty: the additional time required to service the miss.
- Hit rate: fraction of accesses that are in a specified level of the hierarchy.
- Miss rate: fraction of accesses that are not in a specified level of the hierarchy.
- Block: unit of information that is checked to reside in a specified level of the hierarchy and is retrieved from the next lower level on a miss.

MEMORY HIERARCHY

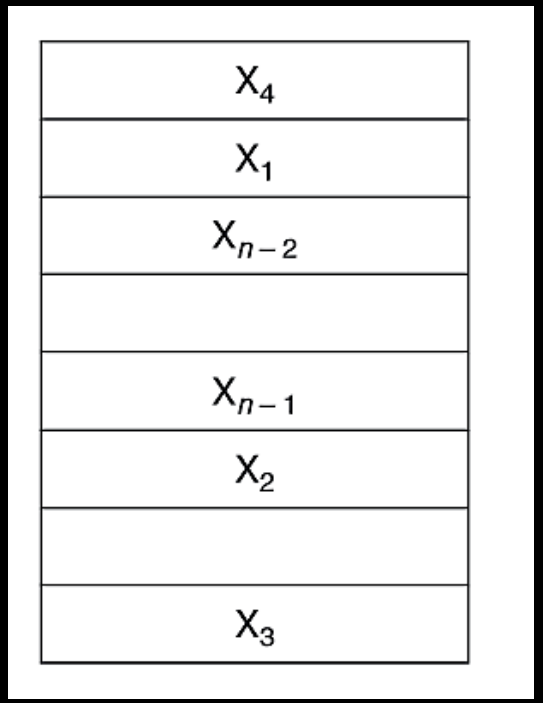
The key points so far:

- Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.
- Memory hierarchy uses smaller and faster memory technologies close to the processor. Accesses that hit in the highest level can be processed quickly. Accesses that miss go to lower levels, which are larger but slower. If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a *true* size equal to that of the lowest (and largest) level.
- Memory is typically a true hierarchy, meaning that data cannot be present in level i unless it is also present in level $i+1$.

CACHES

We'll begin by looking at the most basic cache. Let's say we're running a program that, so far, has referenced $n - 1$ words. These could be $n - 1$ independent integer variables, for example.

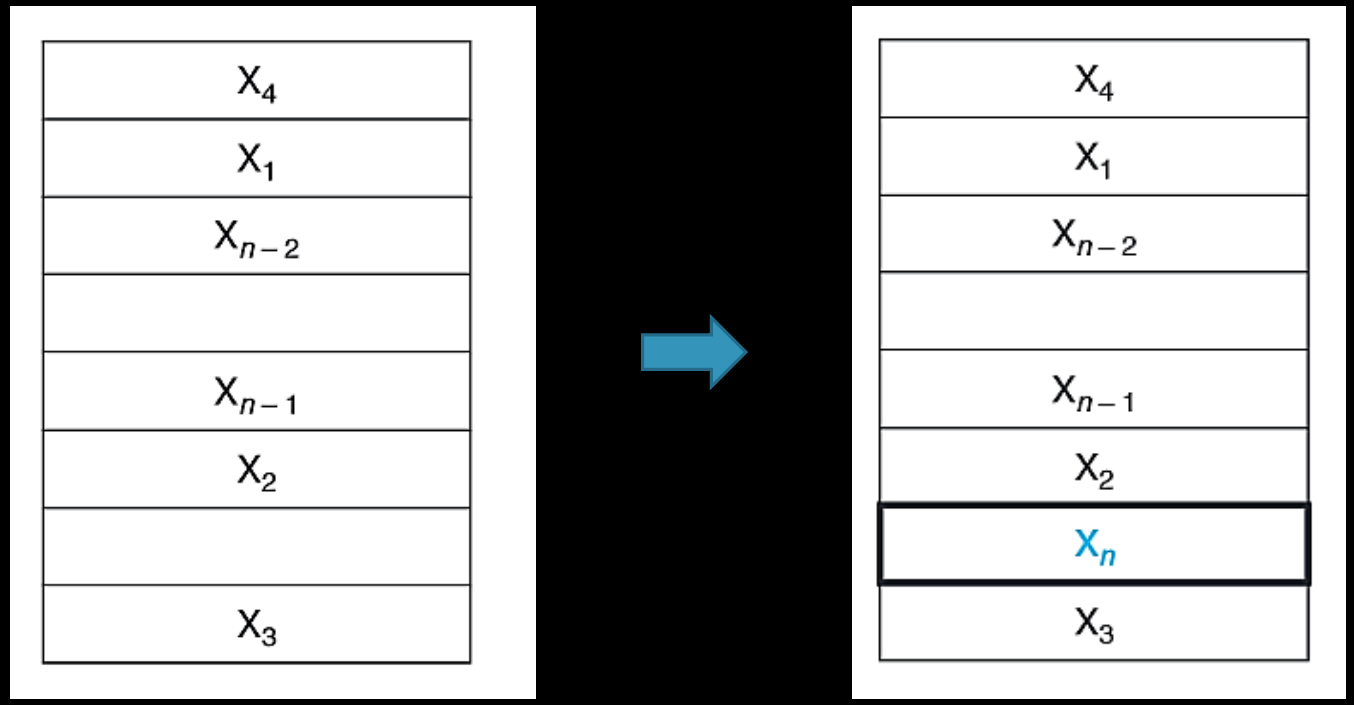
At this point, our cache might look like this (assuming a block is simply 1 word). That is, every reference made so far has been moved into the cache to take advantage of temporal locality.



What happens when our program references X_n?

CACHES

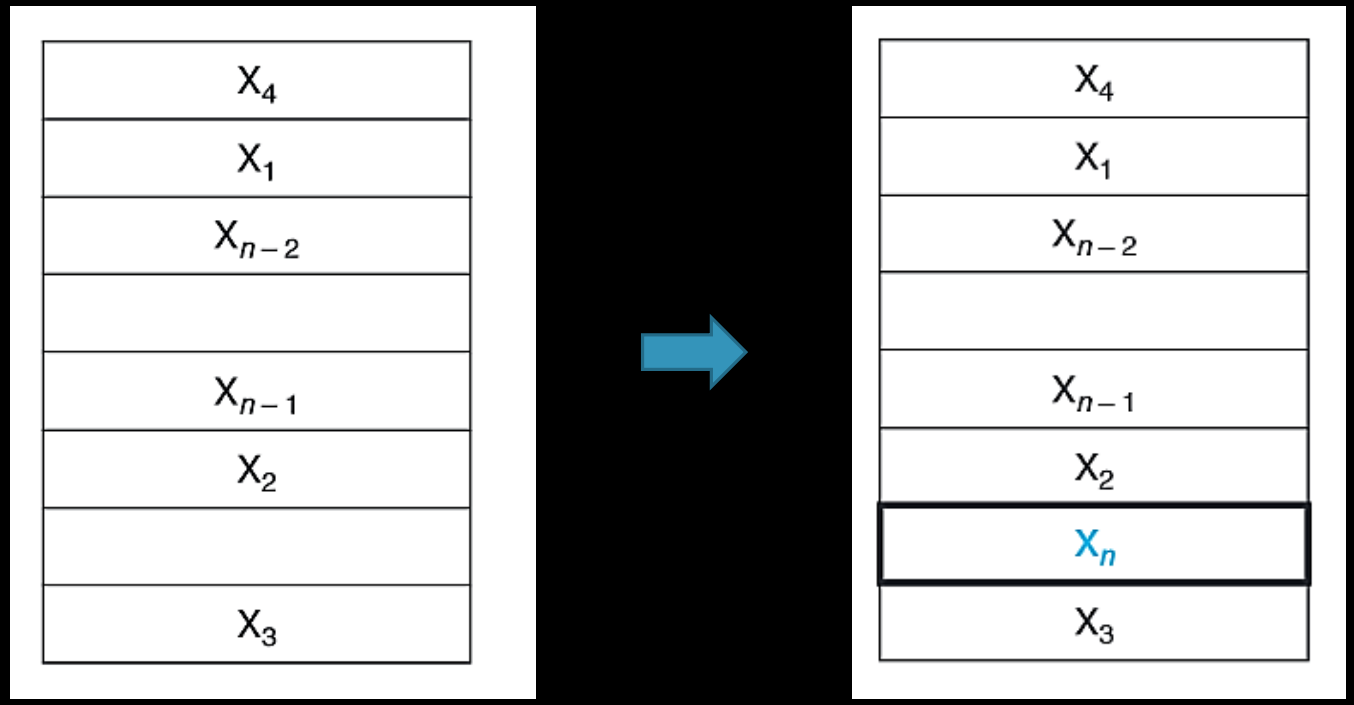
A reference to X_n causes a miss, which forces the cache to fetch X_n from some lower level of the memory hierarchy, presumably main memory.



Two Questions:
1. How do we know if an item is present in the cache?
2. How do we find the item in the cache?

CACHES

A reference to X_n causes a miss, which forces the cache to fetch X_n from some lower level of the memory hierarchy, presumably main memory.



Two Questions:
1. How do we know if an item is present in the cache?
2. How do we find the item in the cache?

One Answer: If each word can go in exactly one place in the cache, then we can easily find it in the cache.

DIRECT-MAPPED CACHES

The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the address of the word in memory.

This creates a direct-mapped cache – every location in memory is mapped directly to one location in the cache.

A typical direct-mapped cache uses the following mapping:

$$(Block\ Address)\ \%\ (Number\ of\ blocks\ in\ the\ cache)$$

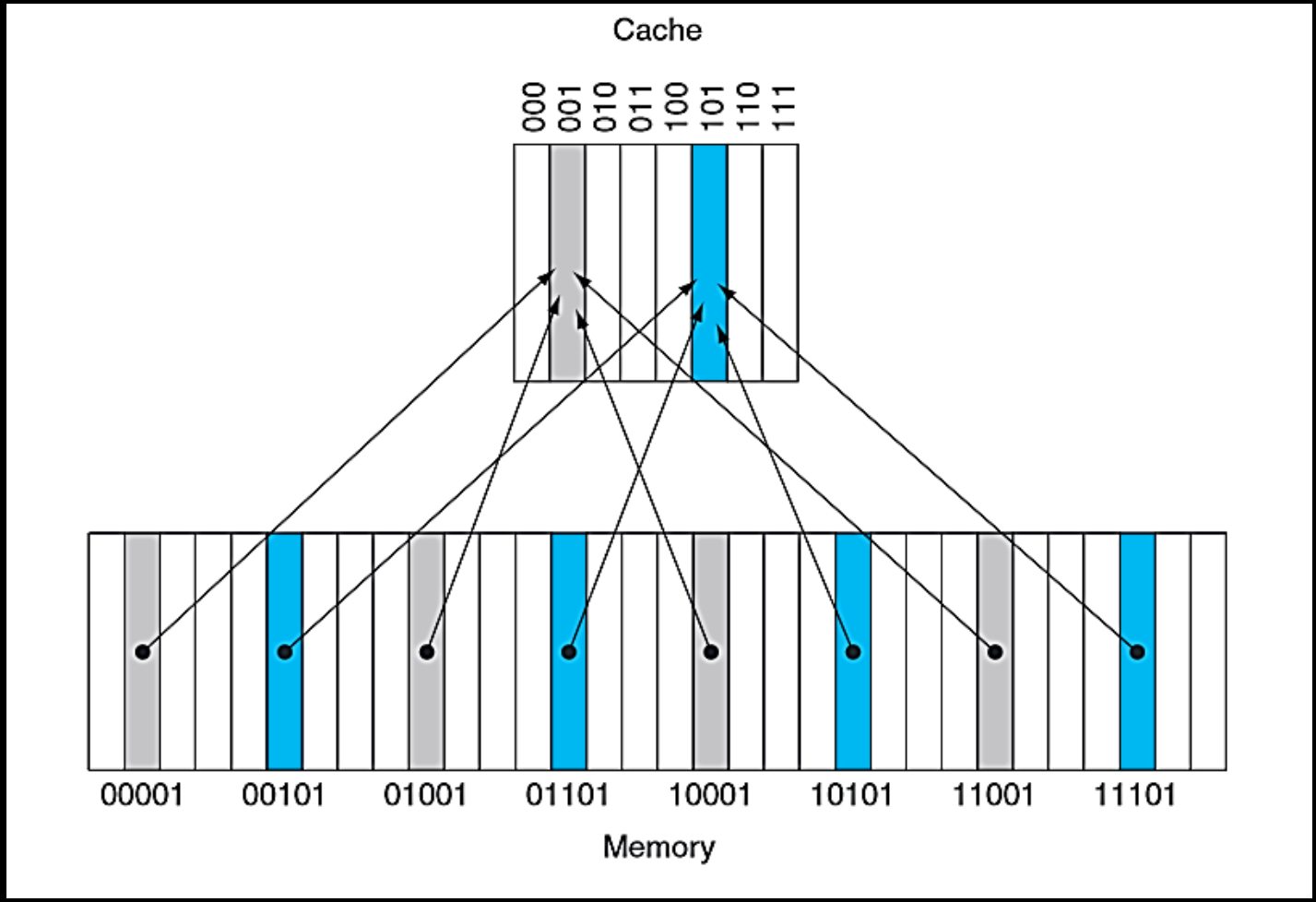
Conveniently, entering a block into a cache with 2^n entries means just looking at the lower n bits of the block address.

DIRECT-MAPPED CACHE

Here is an example cache which contains $2^3 = 8$ entries.

Blocks in memory are mapped to a particular cache index if the lower 3 bits of the block address matches the index.

So, now we know *where* to find the data but we still have to answer the following question: how do we know if the data *we want* is in the cache?



TAGS

To verify that a cache entry contains the data we're looking for, and not data from another memory address with the same lower bits, we use a *tag*.

A tag is a field in a table which corresponds to a cache entry and gives extra information about the source of the data in the cache entry.

What is an obvious choice for the tag?

TAGS

To verify that a cache entry contains the data we're looking for, and not data from another memory address with the same lower bits, we use a *tag*.

A tag is a field in a table which corresponds to a cache entry and gives extra information about the source of the data in the cache entry.

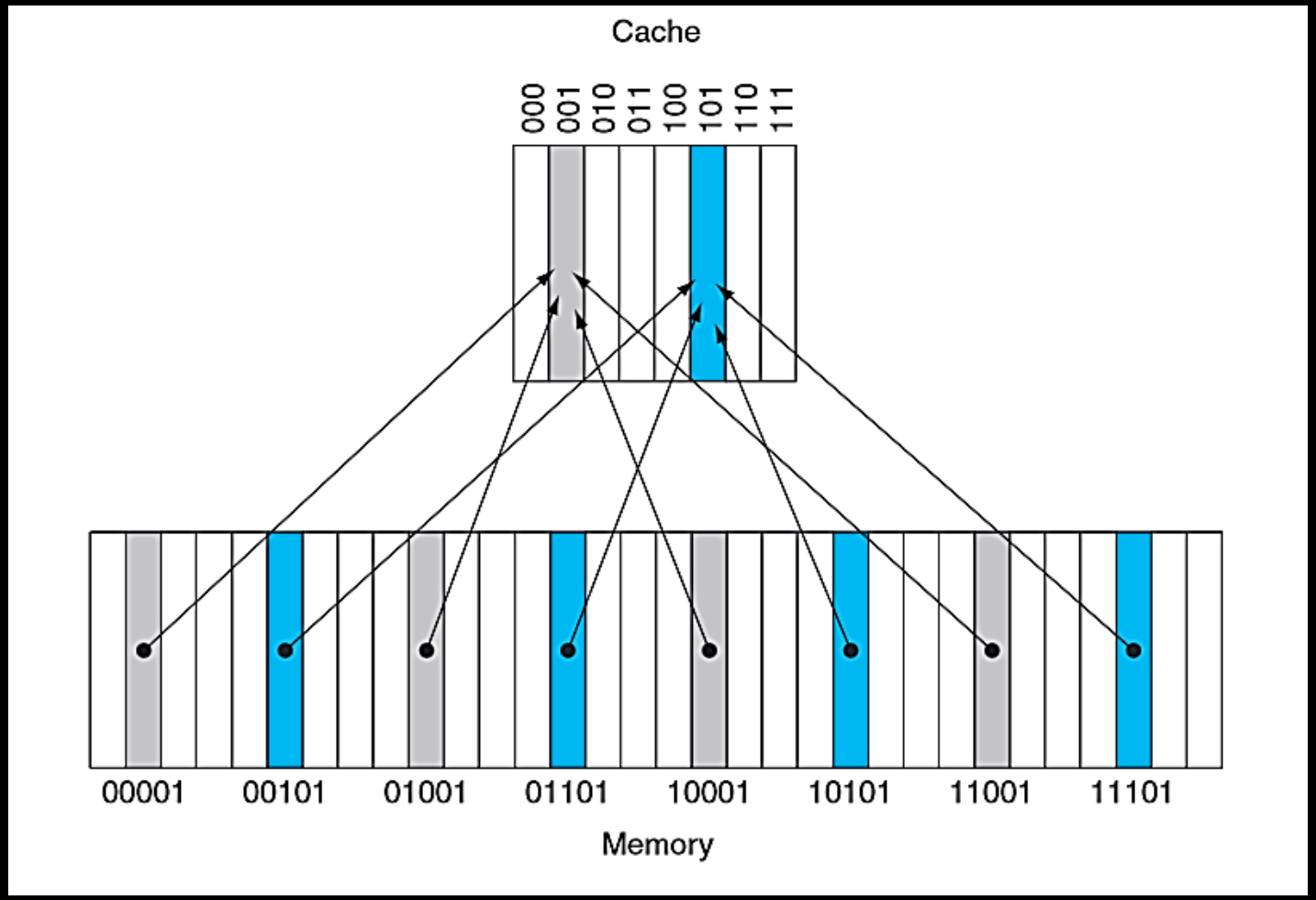
What is an obvious choice for the tag? The upper bits of the address of the block!

TAGS

For instance, in this particular example, let's say the block at address 01101 is held in the cache entry with index 101.

The tag for the cache entry with index 101 must then be 01, the upper bits of the address.

Therefore, when looking in the cache for the block at address 11101, we know that we have a miss because 11 != 01.



VALID BIT

Even if there is data in the cache entry and a tag associated with the entry, we may not want to use the data. For instance, when a processor has first started up or when switching processes, the cache entries and tag fields may be meaningless.

Generally speaking, a *valid bit* associated with the cache entry can be used to ensure that an entry is valid.

EXERCISE

Let's assume we have an 8-entry cache with the initial state shown to the right. Let's fill in the cache according to the references that come in listed in the table below.

Note that initially the valid-bit entries are all 'N' for not valid.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)

EXERCISE

The first reference is for the block at address 22, which uses the lower bits 110 to index into the cache. The 110 cache entry is not valid so this is a miss.

We need to retrieve the contents of the block at address 22 and place it in the cache entry.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	$(10110_{two} \bmod 8) = 110_{two}$

EXERCISE

The block at address 22 is now placed in the data entry of the cache and the tag is updated to the upper portion of the address, 10. Also, the valid bit is set to 'Y'.

Now, we have a reference to the block at address 26. What happens here?

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss	(11010_{two} mod 8) = 010_{two}

EXERCISE

We have a miss, so we retrieve the data from address 26 and place it in the cache entry. We also update the tag and valid bit.

Now, we have a reference the block at address 22 again. Now what happens?

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss	(11010_{two} mod 8) = 010_{two}
22	10110_{two}	hit	(10110_{two} mod 8) = 110_{two}

EXERCISE

The correct data is already in the cache! We don't have to update the contents or fetch anything from main memory.

Similarly, we will have another reference to the block at address 26. We do not need to update the cache at all.

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss	(11010_{two} mod 8) = 010_{two}
22	10110_{two}	hit	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	hit	(11010_{two} mod 8) = 010_{two}

EXERCISE

Now, we have a reference to the block at address 16. Its associated cache entry is invalid, so we will need to fetch the data from main memory and update the entry.

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss	(11010_{two} mod 8) = 010_{two}
22	10110_{two}	hit	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	hit	(11010_{two} mod 8) = 010_{two}
16	10000_{two}	miss	(10000_{two} mod 8) = 000_{two}

EXERCISE

Now, we have a reference to the block at address 3. Its associated cache entry is invalid, so we will need to fetch the data from main memory and update the entry.

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss	(11010_{two} mod 8) = 010_{two}
22	10110_{two}	hit	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	hit	(11010_{two} mod 8) = 010_{two}
16	10000_{two}	miss	(10000_{two} mod 8) = 000_{two}
3	00011_{two}	miss	(00011_{two} mod 8) = 011_{two}

EXERCISE

A reference to the block at address 16 causes a hit (as we have already pulled this data into the cache) so we do not have to make any changes.

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss	(11010_{two} mod 8) = 010_{two}
22	10110_{two}	hit	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	hit	(11010_{two} mod 8) = 010_{two}
16	10000_{two}	miss	(10000_{two} mod 8) = 000_{two}
3	00011_{two}	miss	(00011_{two} mod 8) = 011_{two}
16	10000_{two}	hit	(10000_{two} mod 8) = 000_{two}

EXERCISE

Now, we get something interesting. We have a reference to the block at address 18. The lower bits used to index into the cache are 010. As these are also the lower bits of address 26, we have a valid entry but it's not the one we want. Comparing the tag of the entry with the upper portion of 18's binary representation tells us we have a miss.

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss	(11010_{two} mod 8) = 010_{two}
22	10110_{two}	hit	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	hit	(11010_{two} mod 8) = 010_{two}
16	10000_{two}	miss	(10000_{two} mod 8) = 000_{two}
3	00011_{two}	miss	(00011_{two} mod 8) = 011_{two}
16	10000_{two}	hit	(10000_{two} mod 8) = 000_{two}
18	10010_{two}	miss	(10010_{two} mod 8) = 010_{two}

EXERCISE

We fetch the data at address 18 and update the cache entry to hold this data, as well as the correct tag. Note now that a reference to the block at address 26 will result in a miss and we'll have to fetch that data again.

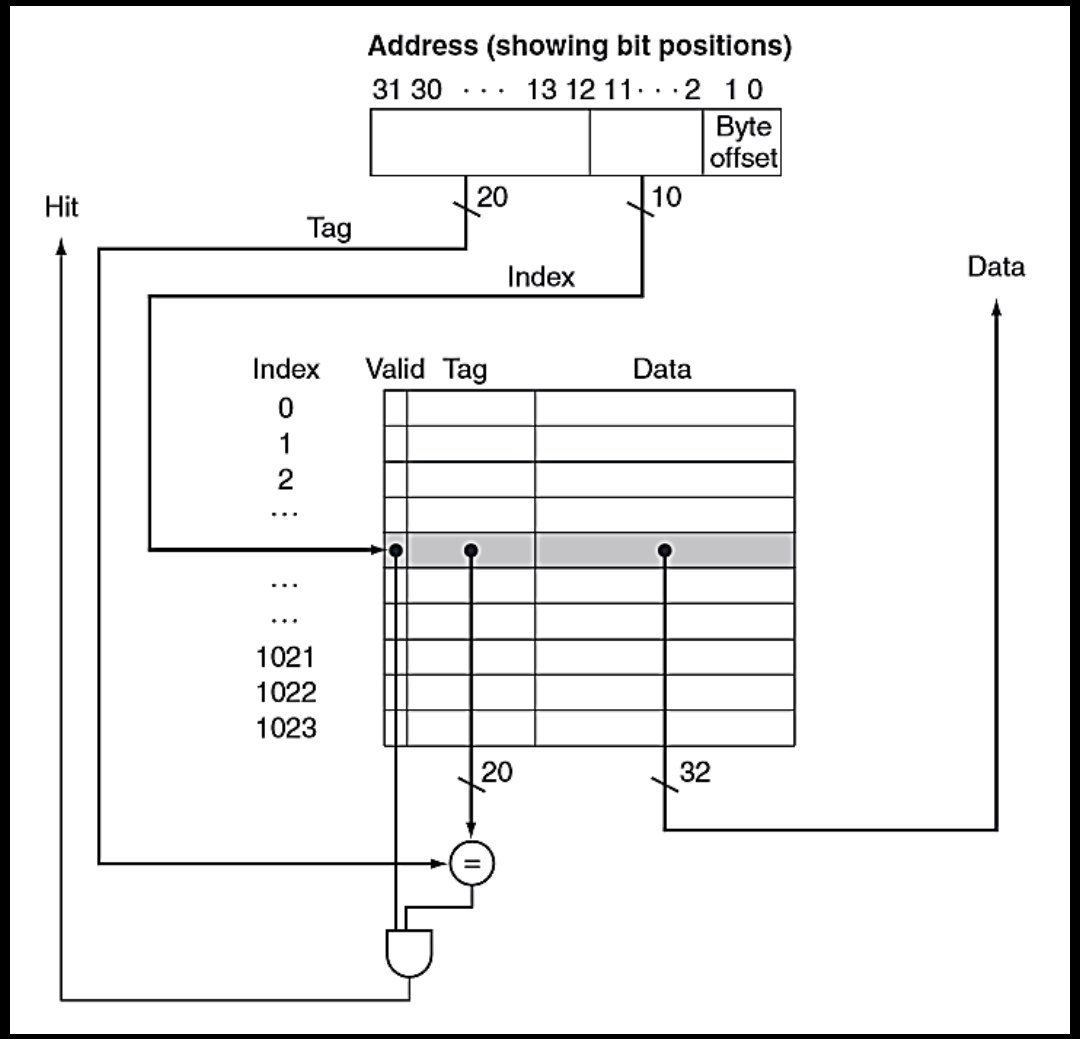
Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	miss	(11010_{two} mod 8) = 010_{two}
22	10110_{two}	hit	(10110_{two} mod 8) = 110_{two}
26	11010_{two}	hit	(11010_{two} mod 8) = 010_{two}
16	10000_{two}	miss	(10000_{two} mod 8) = 000_{two}
3	00011_{two}	miss	(00011_{two} mod 8) = 011_{two}
16	10000_{two}	hit	(10000_{two} mod 8) = 000_{two}
18	10010_{two}	miss	(10010_{two} mod 8) = 010_{two}

PHYSICAL ADDRESS TO CACHE

To the right is a figure showing how a typical physical address may be divided up to find the valid entry within the cache.

- The offset is used to indicate the first byte accessed within a block. Its size is $\log_2 number\ of\ bytes\ in\ block$. For example, a block containing 4 bytes does not need to consider the lower 2 bits of the address to index into the cache.
- The cache index, in this case, is a 10-bit wide lower portion of the physical address (because there are $2^{10} = 1024$ entries).
- The tag is the upper 20 bits of the physical address.



OFFSET

Consider a scheme where a block of memory contains 2 words. Each word is 4 bytes.

Bytes are the smallest addressable unit of memory so a block starting at address 34892896 contains 8 byte-addressable locations.

Because $2^3 = 8$, we need 3 bits to individually identify the addresses in the block. The 4th bit is the first bit common to all addresses in the block.

Therefore, the offset to the index is given by $\log_2(num\ bytes\ in\ block)$.

BLOCKS IN A CACHE

We've mostly assumed so far that a block contains one word, or 4 bytes. In reality, a block contains several words.

Assuming we are using 32-bit addresses, consider a direct-mapped cache which holds 2^n blocks and each block contains 2^m words.

How many bytes are in a block?

How big does a tag field need to be?

BLOCKS IN A CACHE

We've mostly assumed so far that a block contains one word, or 4 bytes. In reality, a block contains several words.

Assuming we are using 32-bit addresses, consider a direct-mapped cache which holds 2^n blocks and each block contains 2^m words.

How many bytes are in a block? $2^m * 4 = 2^m * 2^2 = 2^{m+2}$ bytes per block.

How big does a tag field need to be? 32 – (n + m + 2). A block has a 32-bit address. We do not consider the lower m+2 bits because there are 2^{m+2} bytes in a block. We need n bits to index into the cache, m bits to identify the word.

EXERCISE

How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word blocks, assuming a 32-bit address?

EXERCISE

How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word blocks, assuming a 32-bit address?

We know that 16 KB is 4K words, which is 2^{12} words, and, with a block size of 4 words (2^2), 2^{10} blocks.

Each block contains 4 words, or 128 bits, of data. Each block also has a tag that is 32-10-2-2 bits long, as well as one valid bit. Therefore, the total cache size is

$$2^{10} \times (128 + (32 - 10 - 2 - 2) + 1) = 147\ Kbits$$

Or, 18.4 KB cache for 16KB of data.

EXERCISE

Consider a cache with 64 blocks and a block size of 16 bytes (4 words). What block number does byte address 1200 (0100 1011 0000) map to?

EXERCISE

Consider a cache with 64 blocks and a block size of 16 bytes (4 words). What block number does byte address 1200 map to?

First of all, we know the entry into the cache is given by

$$(Block\ Address)\ \%\ (Number\ of\ blocks\ in\ the\ cache)$$

Where the block address is given by $\frac{Byte\ Address}{Number\ of\ bytes\ per\ block}$.

So, the block address is $\frac{1200}{16} = 75$.

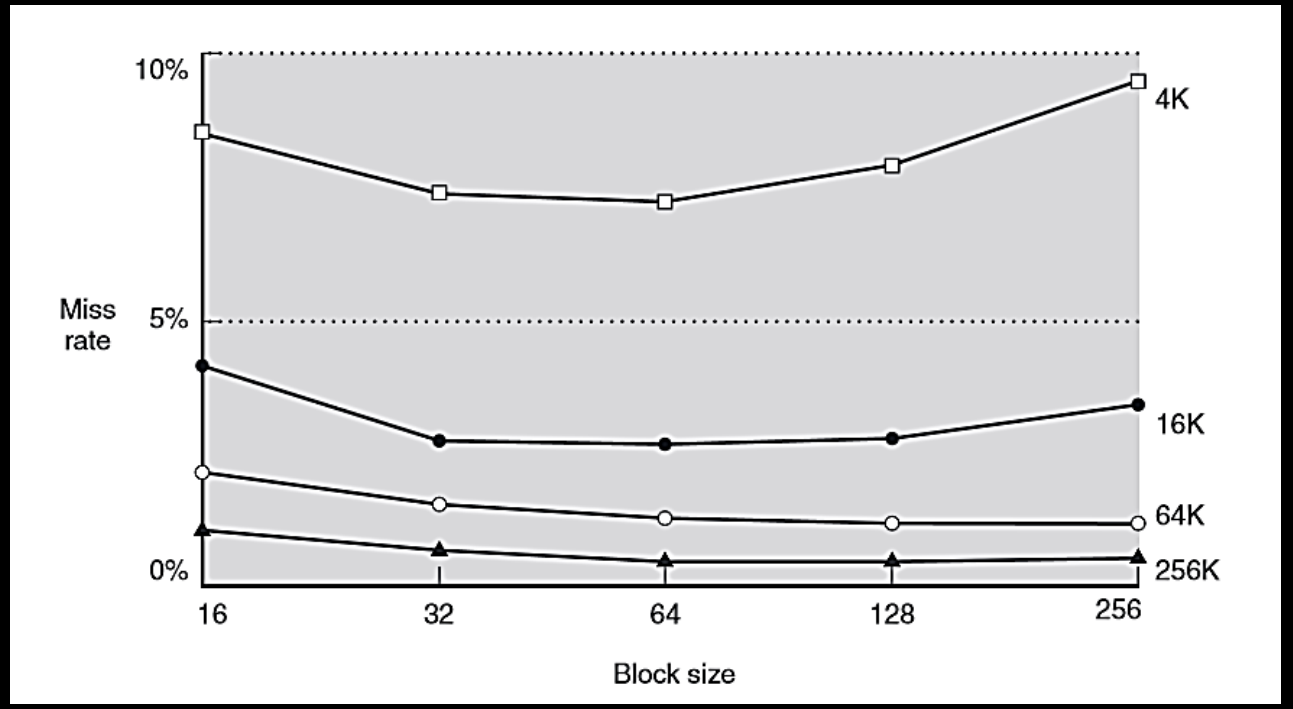
This corresponds to block number $75\ \%\ 64 = 11$. This block maps all addresses between 1200 and 1215.

BLOCK SIZE AND MISS RATE

A larger block size means we bring more contiguous bytes of memory in when we fetch a block. This can lower our miss rate as it exploits spatial locality.

However, in a fixed-size cache, a larger block size means less blocks in a cache – therefore, we may have blocks competing for cache space more often.

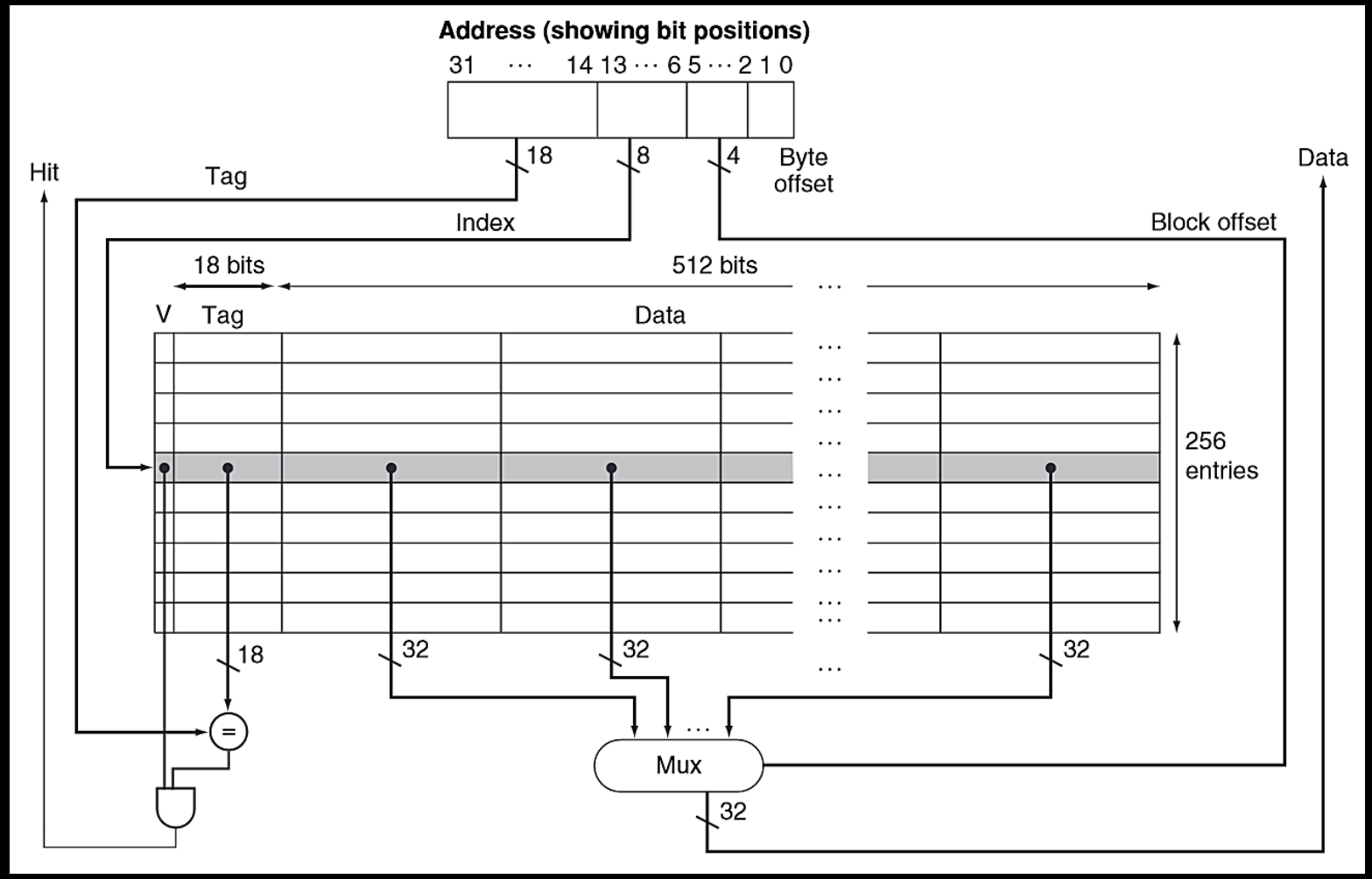
Furthermore, a larger block size takes more time to retrieve from main memory in the case of a miss.



64-BYTE BLOCK CACHE

Here is a 256-entry cache that has 64-byte block entries. That is, each block is 16 words wide.

We index using an 8-bit portion of the address. The individual bytes of the address are identifiable using the lower 6 bits ($2^6 = 64$). However, we don't want to access every byte. We only want to access individual words. This requires 4 bits because $\frac{64}{4} = 16 = 2^4$.



EXERCISE

Assume a direct-mapped cache with 4 blocks and 8 bytes per block.
How is the physical address portioned?

Tag bits	Index bits	Offset bits

Fill in the appropriate information for the following memory references:

Address	Tag	Index	Offset
4			
8			
12			
20			
67			

EXERCISE

Assume a direct-mapped cache with 4 blocks and 8 bytes per block.
How is the physical address portioned?

Tag bits	Index bits	Offset bits
27 [31-5]	2 [4-3]	3 [2-0]

Fill in the appropriate information for the following memory references:

Address	Tag	Index	Offset
4	0	0	4
8	0	1	0
12	0	1	4
20	0	2	4
67	2	0	3

FULLY-ASSOCIATIVE CACHE

We've already seen direct-mapped caches, a simple scheme where every block has one particular cache entry where it can be placed.

In a *fully-associative* cache, any block can be found in any entry of the cache.

To find a block in the cache, we must search the entire cache – therefore, this scheme is only practical for caches with a small number of entries.

SET-ASSOCIATIVE CACHE

The middle ground between direct-mapped and fully-associative is set-associative.

In a *set-associative* cache, there are a fixed number of entries where a particular block may be found. If a set-associative cache allows *n* different entries for a block to be found, it is called an *n-way set-associative cache*.

An n-way set-associative cache may have some number of sets, each containing n blocks. A block address can be mapped to a particular set, in which the block can be placed in any of the n entries.

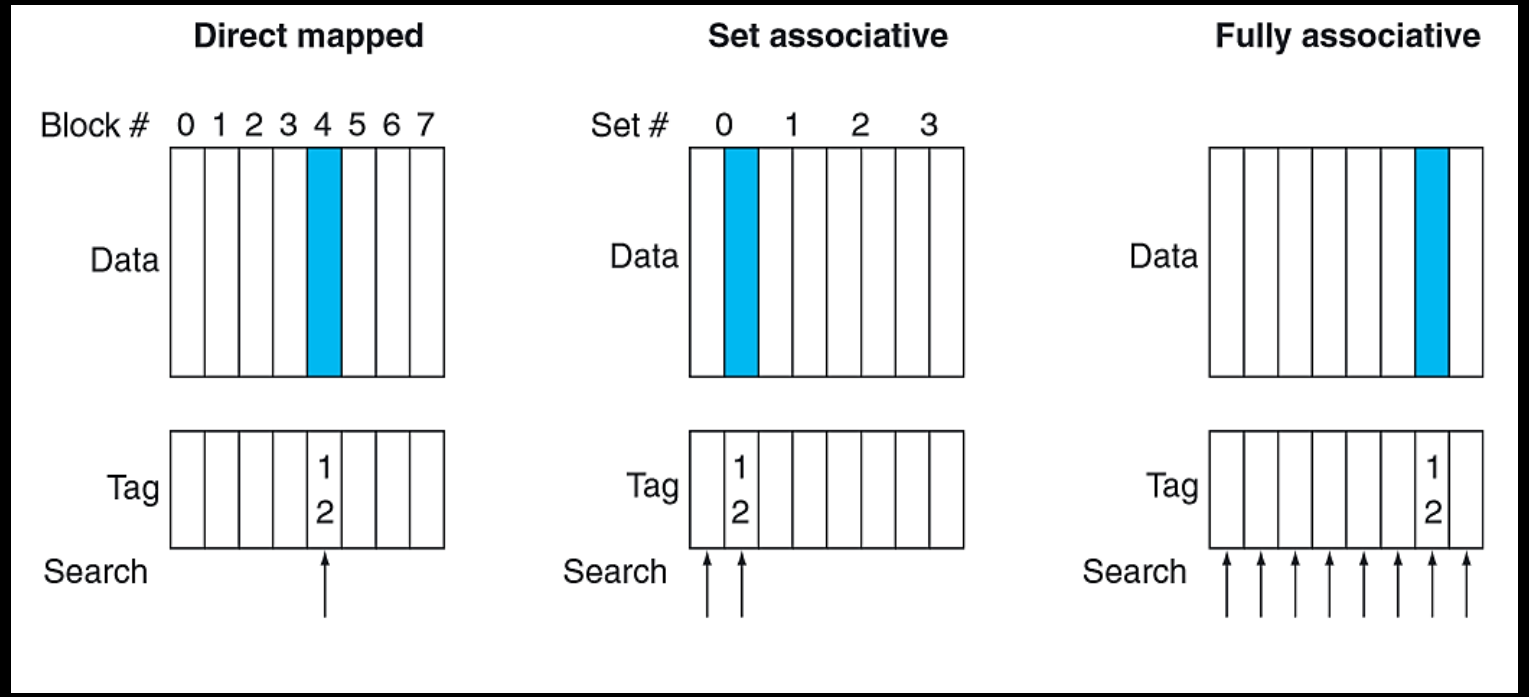
To find a reference in a set-associative cache, we figure out its set based on the address and then search all of the entries in the set.

SET-ASSOCIATIVE CACHE

The example below has a reference with a block address of 12 and each cache organization has 8 entries.

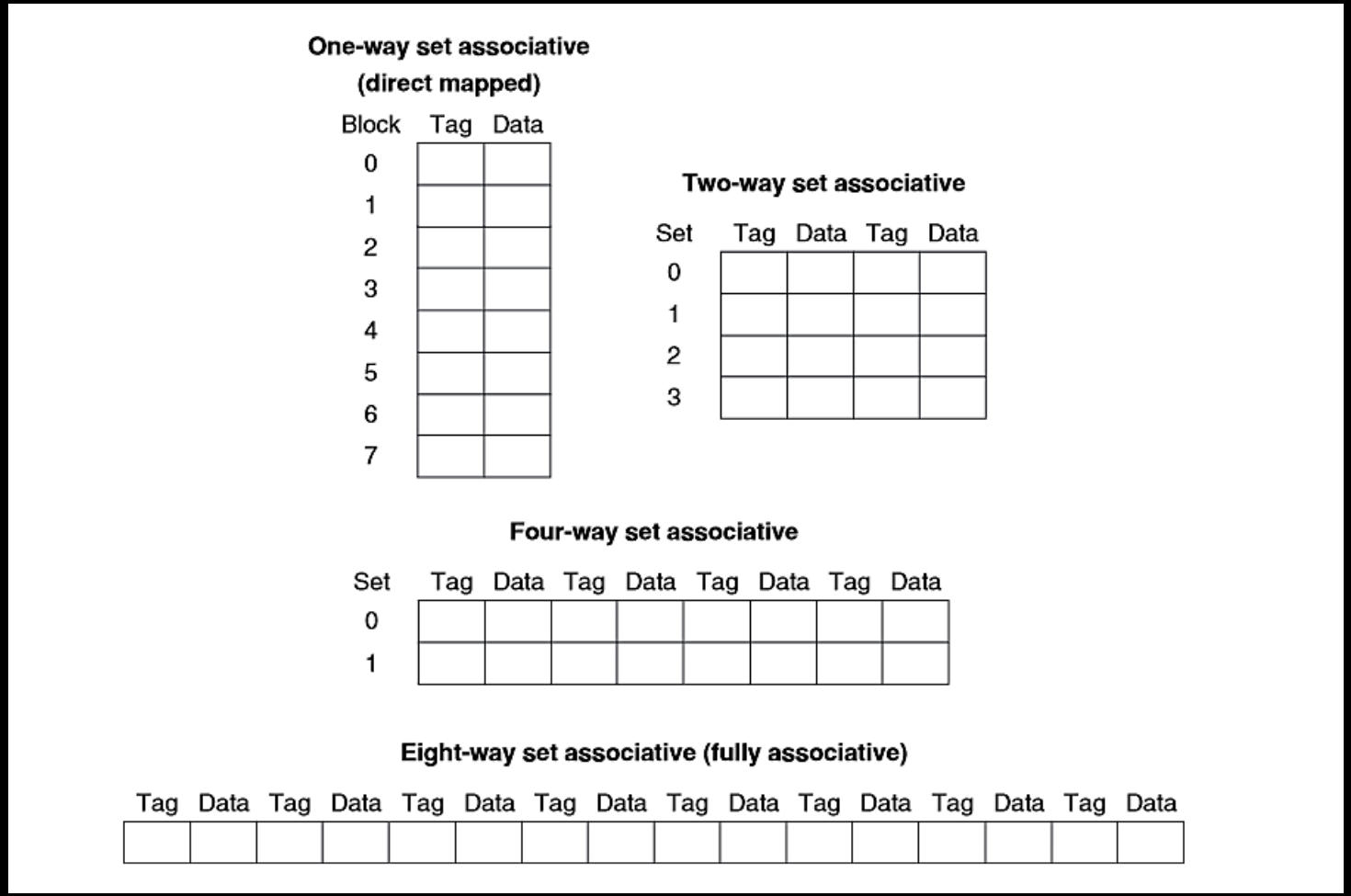
In a set-associative cache, the set can be found using the following:

$$(Block\ address)\%\ (\#Sets)$$



SET-ASSOCIATIVE CACHE

All placement strategies are really a variation on set-associativity.

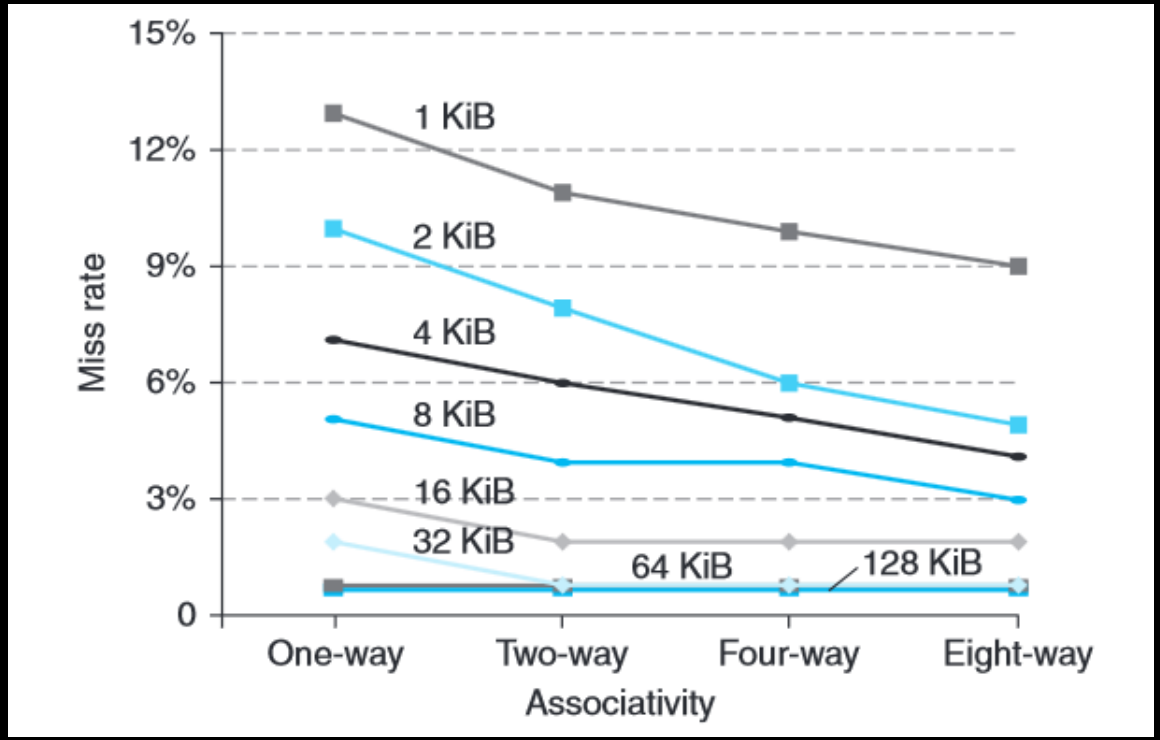


SET-ASSOCIATIVE CACHE

The advantage of increasing the degree of associativity is that, typically, the miss rate will decrease.

The disadvantages are:

- Potential hit time increase.
- More tag bits per cache block.
- Logic to determine which block to replace.



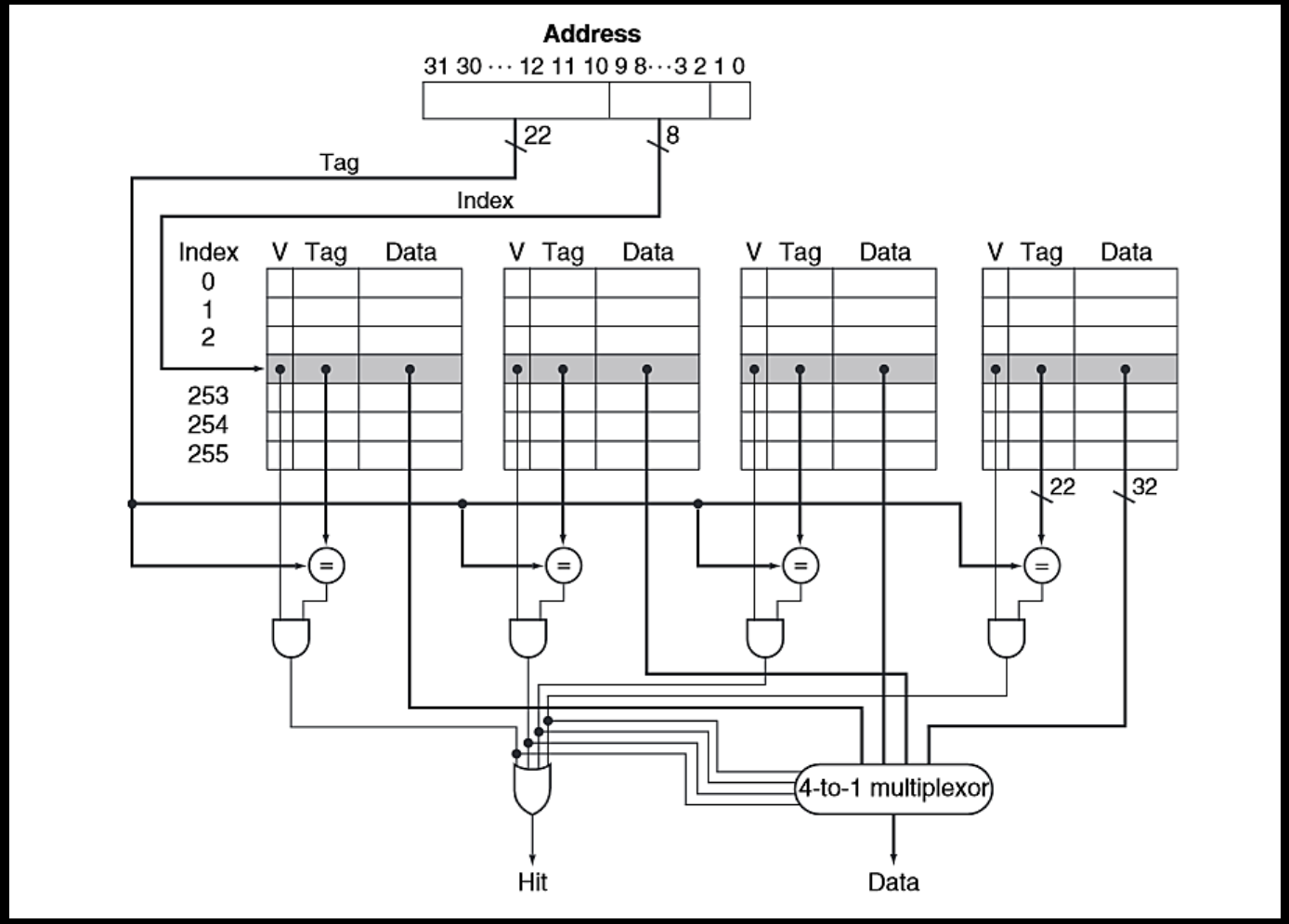
FOUR-WAY SET-ASSOCIATIVE CACHE

Here is a set-associative cache with 256 sets of four blocks each, where each block is one word.

The index tells us which set to look in. We need 8 bits for the index because $2^8 = 256$.

The tag of every entry is compared to the upper 22 bits of the address. If there is a match, and the valid bit is set, we have a hit. The mux selects the data of the entry that resulted in a hit.

Otherwise, we have a miss.



SET-ASSOCIATIVE CACHE

Assume a 2-way set-associative cache with 64 sets and 4 words per block.

How is the physical address partitioned?

Tag bits	Index bits	Offset bits

Fill in the appropriate information for the following memory references:

Address	Tag	Index	Offset
300			
304			
1216			
4404			
4408			

SET-ASSOCIATIVE CACHE

Assume a 2-way set-associative cache with 64 sets and 4 words per block.

How is the physical address partitioned?

Tag bits	Index bits	Offset bits
22	6	4

Fill in the appropriate information for the following memory references:

Address	Tag	Index	Offset
300	0	18	12
304	0	19	0
1216	1	12	0
4404	4	19	4
4408	4	19	8

BLOCK REPLACEMENT

Block replacement strategies for direct-mapped are easy: just write to the entry of the block you are bringing into the cache.

However, in a set-associative cache, there are multiple block entries that can be used. If the set is full, how do we decide which block should be replaced?

- Random: choose a block randomly to replace. Easy to implement.
- Least Recently Used (LRU): replace the least-recently accessed block.
  - Better miss rate than random.
  - Expensive to implement, especially for high associativity.

SET-ASSOCIATIVE CACHE

Assume a 2-way set-associative cache with 64 sets and 4 words per block. Indicate the result of searching for the reference in the cache.

Address	Tag	Index	Offset	Result
300	0	18	12	Miss
304	0	19	0	
1216	1	12	0	
4404	4	19	4	
4408	4	19	8	
9416	9	12	8	
296	0	18	8	
304	0	19	0	
1220	1	12	4	
2248	2	12	8	

SET-ASSOCIATIVE CACHE

Address	Tag	Index	Offset	Result
00 0001 0010 1100	0	18	12	Miss
00 0001 0011 0000	0	19	0	
00 0100 1100 0000	1	12	0	
01 0001 0011 0100	4	19	4	
01 0001 0011 1000	4	19	8	
10 0100 1100 1000	9	12	8	
00 0001 0010 1000	0	18	8	
00 0001 0011 0000	0	19	0	
00 0100 1100 0100	1	12	4	
00 1000 1100 1000	2	12	8	

4 offset bits, 6 index bits

SET-ASSOCIATIVE CACHE

Assume a 2-way set-associative cache with 64 sets and 4 words per block. Indicate the result of searching for the reference in the cache.

Address	Tag	Index	Offset	Result
300	0	18	12	Miss
304	0	19	0	Miss
1216	1	12	0	Miss
4404	4	19	4	Miss
4408	4	19	8	Hit
9416	9	12	8	Miss
296	0	18	8	Hit
304	0	19	0	Hit
1220	1	12	4	Hit
2248	2	12	8	Miss

WRITING TO THE CACHE

Writing to the cache is a little more complicated than reading from the cache.

Let's say, in the MEM stage of a store word instruction, we write to the data cache. Then, main memory and data cache will have different values for that particular block. In this case, they are said to be *inconsistent*.

There are two solutions to this issue. The method we use becomes our *write policy*.

- Write-through
- Write-back

WRITE POLICIES

- Write-through
  - Always write data into both the cache and main memory (or the next lower level).
  - Easily implemented.
  - Could slow down the processor → use a write buffer to allow the processor to continue executing while the data is written to memory.
  - Cache and memory are always consistent.
- Write-back
  - Only write the data to the cache block.
  - The updated block is only written back to memory when it is replaced by another block.
  - A *dirty bit* is used to indicate whether the block needs to be written or not.
  - Reduces accesses to the next lower level.

What if the block to be written is not in the cache?

WRITE MISS POLICIES

- Write allocate
  - The block is loaded into the cache on a write miss.
  - Typically used with write back.
- No-write allocate
  - The block is not loaded into the cache on a write miss.
  - Block simply updated in main memory.
  - Typically used with write through.

WRITE-THROUGH, NO-WRITE ALLOCATE

Assume a 2-way set-associative cache with 64 cache sets, 4 words per block, and an LRU replacement policy. Fill in the appropriate information for the following memory references.

R/W	Addr	Tag	Index	Offset	Result	Memref	Update Cache?
W	300						
R	304						
R	4404						
W	4408						
W	8496						
R	8500						
R	304						

WRITE-THROUGH, NO-WRITE ALLOCATE

Assume a 2-way set-associative cache with 64 cache sets, 4 words per block, and an LRU replacement policy. Fill in the appropriate information for the following memory references.

R/W	Addr	Tag	Index	Offset	Result	Memref	Update Cache?
W	300	0	18	12	Miss	Yes	No
R	304	0	19	0			
R	4404	4	19	4			
W	4408	4	19	8			
W	8496	8	19	0			
R	8500	8	19	4			
R	304	0	19	0			

WRITE-THROUGH, NO-WRITE ALLOCATE

Assume a 2-way set-associative cache with 64 cache sets, 4 words per block, and an LRU replacement policy. Fill in the appropriate information for the following memory references.

R/W	Addr	Tag	Index	Offset	Result	Memref	Update Cache?
W	300	0	18	12	Miss	Yes	No
R	304	0	19	0	Miss	Yes	Yes
R	4404	4	19	4	Miss	Yes	Yes
W	4408	4	19	8	Hit	Yes	Yes
W	8496	8	19	0	Miss	Yes	No
R	8500	8	19	4	Miss	Yes	Yes
R	304	0	19	0	Miss	Yes	Yes

WRITE-BACK, WRITE ALLOCATE

Assume a 2-way set-associative cache with 64 cache sets, 4 words per block, and an LRU replacement policy. Fill in the appropriate information for the following memory references.

R/W	Addr	Tag	Index	Offset	Result	Memref	Update Cache?
W	300	0	18	12			
R	304	0	19	0			
R	4404	4	19	4			
W	4408	4	19	8			
W	8496	8	19	0			
R	8500	8	19	4			
R	304	0	19	0			

WRITE-BACK, WRITE ALLOCATE

Assume a 2-way set-associative cache with 64 cache sets, 4 words per block, and an LRU replacement policy. Fill in the appropriate information for the following memory references.

R/W	Addr	Tag	Index	Offset	Result	Memref	Update Cache?
W	300	0	18	12	Miss	Yes	Yes
R	304	0	19	0	Miss	Yes	Yes
R	4404	4	19	4	Miss	Yes	Yes
W	4408	4	19	8	Hit	No	Yes
W	8496	8	19	0	Miss	Yes	Yes
R	8500	8	19	4	Hit	No	No
R	304	0	19	0	Miss	Yes (2)	Yes

CACHE MISSES

Let's consider the effect of cache misses for instructions. Assume our miss penalty is 10 cycles and the miss rate is .10.

The average access time for an instruction is given by:

$$hit\ time + miss\ rate * miss\ penalty$$

So, the number of cycles needed to fetch instructions is:

$$\begin{aligned} &\#instructions\ * average\ access\ time \\ &= \#instructions\ * (hit\ time + miss\ rate * miss\ penalty) \\ &= \#instructions\ * (1 + .10 * 10) \\ &= \#instructions * 2.0 \end{aligned}$$

CACHES FOR PIPELINED PROCESSORS

In reality, instructions and data have separate caches.

This allows us to not only avoid structural hazards (when one instruction is being fetched while another accesses memory in the same cycle), but also fine-tune the specs of the cache for each task.

Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
inst1	IF	ID	EX	**MEM**	WB			
inst2		IF	ID	EX	MEM	WB		
inst3			IF	ID	EX	MEM	WB	
inst4				**IF**	ID	EX	MEM	WB

MEMORY HIERARCHY MISSES

Not all misses are equal. We can categorize them in the following way:

- Compulsory Misses
  - Caused by first access to block.
  - Possibly decreased by increasing block size.
- Capacity Misses
  - Caused when memory level cannot contain all blocks needed during execution of process.
  - Can be decreased by increasing cache size.
- Conflict Misses
  - Occur when too many blocks compete for same entry in cache.
  - Can be decreased by increasing associativity.

CRITICAL WORD FIRST AND EARLY RESTART

One way to reduce the penalty for misses is to reduce the time spent waiting for the actual request data, rather than the whole block of data.

Critical word first means to request the missed word first from the next memory hierarchy level to allow the processor to continue while filling in the remaining words in the block, usually in a wrap-around fill manner.

Early restart means to fetch the words in the normal order, but allow the processor to continue once the requested word arrives.

MULTILEVEL CACHES

Three levels of cache all on the same chip are now common, where there are separate L1 instruction and data caches and unified L2 and L3 caches.

- The L1 cache is typically much smaller than L2 cache with lower associativity to provide faster access times. Same with L2 and L3.
- The L1 caches typically have smaller block sizes than L2 caches to have a shorter miss penalty. Same with L2 and L3.
- Lower cache levels being much larger and having higher associativity than higher cache levels decreases their misses, which have higher miss penalties.

MULTILEVEL CACHE PERFORMANCE

The miss penalty of an upper level cache is the average access time of the next lower level cache.

$$Average\ Access\ Time = L1\ Hit\ Time + L1\ Miss\ Rate * (L1\ Miss\ Penalty)$$

where

$$L1\ Miss\ Penalty = L2\ Hit\ Time + L2\ Miss\ Rate * (L2\ Miss\ Penalty)$$

What is the average access time given that the L1 hit time is 1 cycle, the L1 miss rate is 0.05, the L2 hit time is 4 cycles, the L2 miss rate is 0.25, and the L2 miss penalty is 50 cycles?

MULTILEVEL CACHE PERFORMANCE

The miss penalty of an upper level cache is the average access time of the next lower level cache.

$$Average\ Access\ Time = L1\ Hit\ Time + L1\ Miss\ Rate * (L1\ Miss\ Penalty)$$

where

$$L1\ Miss\ Penalty = L2\ Hit\ Time + L2\ Miss\ Rate * (L2\ Miss\ Penalty)$$

What is the average access time given that the L1 hit time is 1 cycle, the L1 miss rate is 0.05, the L2 hit time is 4 cycles, the L2 miss rate is 0.25, and the L2 miss penalty is 50 cycles?

$$Average\ Access\ Time = 1 + .05 * (4 + .25 * 50) = 1.85$$

MULTILEVEL CACHE PERFORMANCE

- Local Miss Rate: the fraction of references to one level of a cache that miss.

Example: $L2\ Miss\ Rate = \frac{Misses\ in\ L2}{Accesses\ to\ L2}$

- Global Miss Rate: the fraction of references that miss in all levels of a multilevel cache.

Example: $Global\ Miss\ Rate = L1\ Miss\ Rate * L2\ Miss\ Rate\ *\ \ldots$

IMPROVING CACHE PERFORMANCE

- Techniques for reducing the miss rate:
  - Increase the associativity to exploit temporal locality.
  - Increase the block size to exploit spatial locality.
- Techniques for reducing the miss penalty:
  - Use wrap-around filling of a line (early restart and critical word first).
  - Use multilevel caches.
- Techniques for reducing the hit time:
  - Use small and simple L1 caches.

APPENDIX: SRAM

- Static Random Access Memory
- Used in caches.
- Has a single access port for reads/writes.
- Access time is 5-10 times faster than DRAM.
- Semiconductor memory that uses ~6 transistors for each bit of data.
- Data is maintained as long as power to the SRAM chip is provided; no need to refresh.

APPENDIX: DRAM

- Dynamic Random Access Memory
- Used for main memory.
- Requires a single transistor per bit (much denser and cheaper than SRAM).
- Data is lost after being read, so we must *refresh* after a read by writing back the data.
- The charge can be kept for several milliseconds before a refresh is required. About 1%-2% of the cycles are used to refresh – accomplished by reading a row of data and writing it back.