RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 12
Memory – Cache

Nagi Naganathan
November 17, 2016

MEMORY & I/O SYSTEMS

Chapter 8

***Digital Design and Computer Architecture*, 2nd Edition**

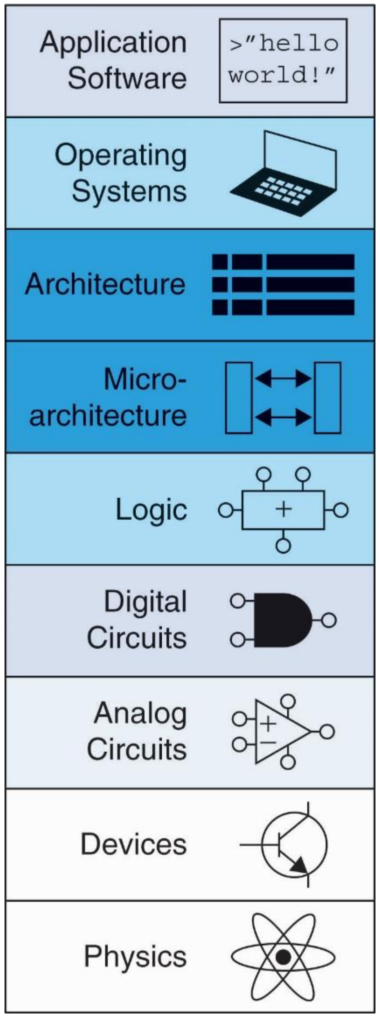
David Money Harris and Sarah L. Harris

ELSEVIER

MEMORY & I/O SYSTEMS

Chapter 8 :: Topics

- Introduction
- Memory System Performance Analysis
- Caches
- Virtual Memory
- Memory-Mapped I/O
- Summary

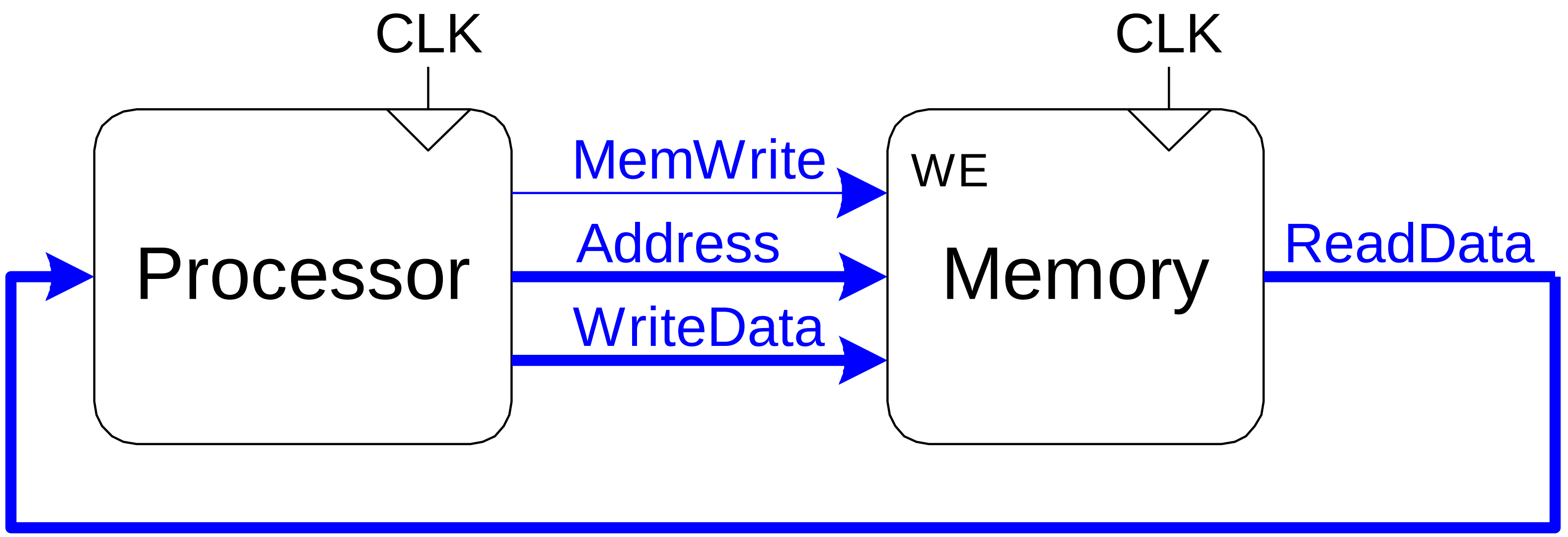


ELSEVIER

Introduction

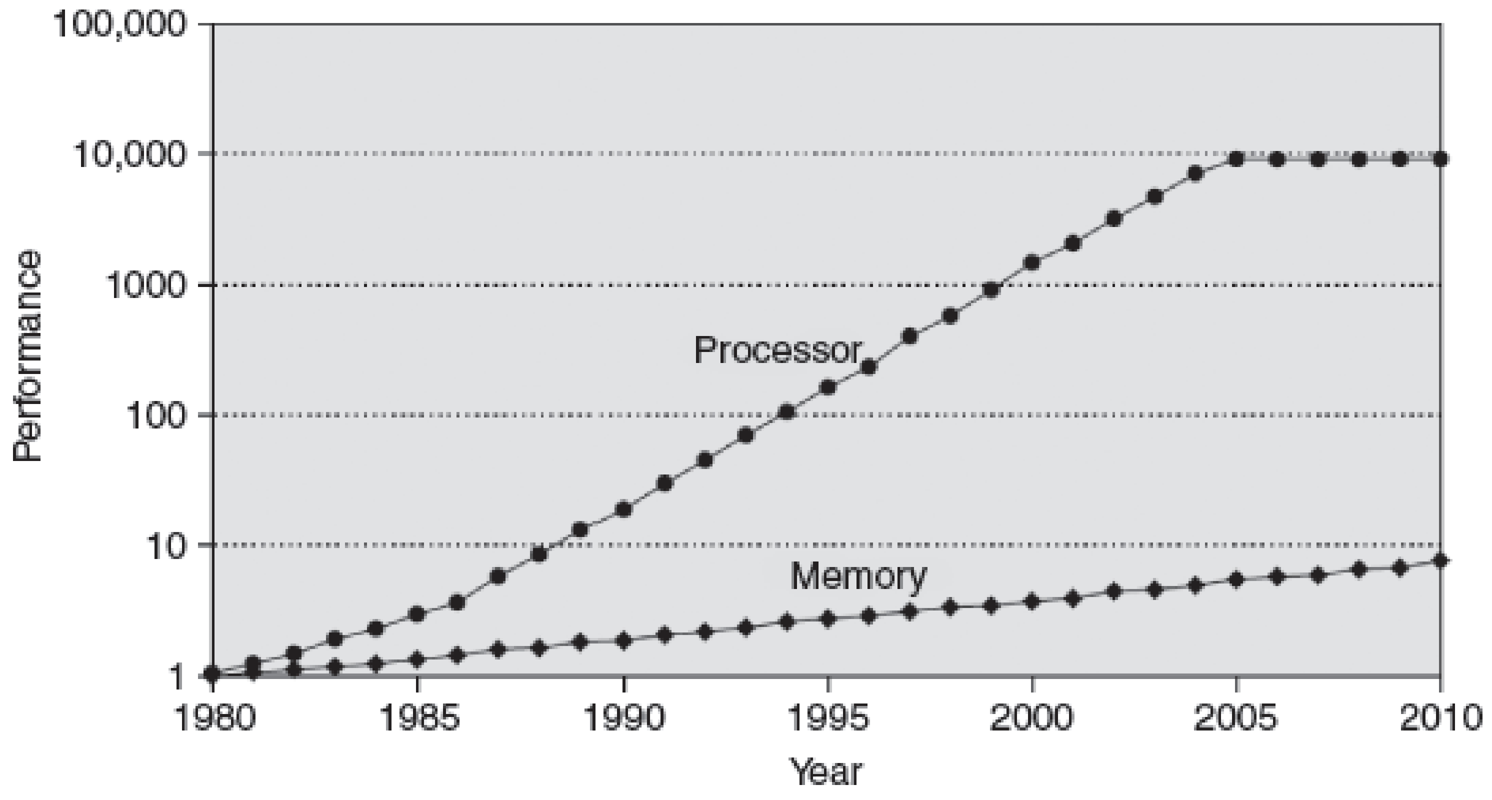
- Computer performance depends on:
  - Processor performance
  - Memory system performance

Memory Interface



Processor-Memory Gap

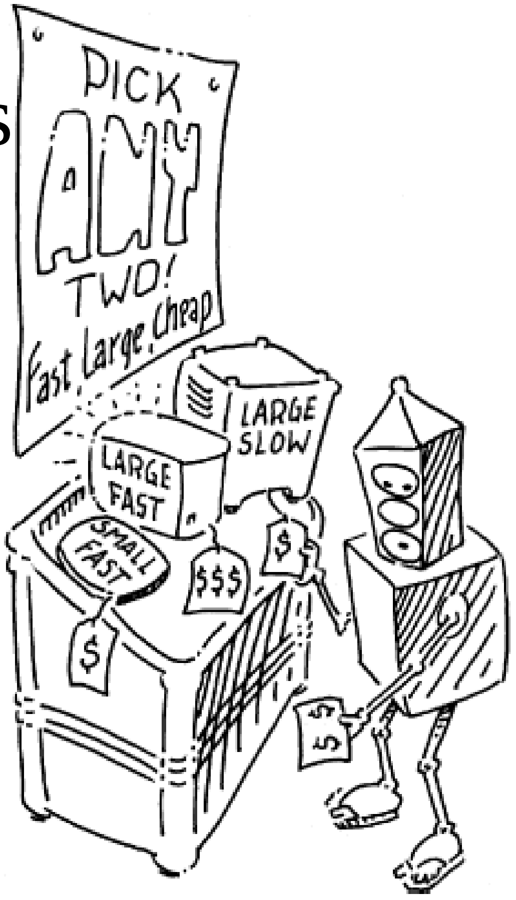
In prior chapters, assumed access memory in 1 clock cycle – but hasn't been true since the 1980's



Memory System Challenge

- Make memory system appear as fast as processor
- Use hierarchy of memories
- Ideal memory:
  - Fast
  - Cheap (inexpensive)
  - Large (capacity)

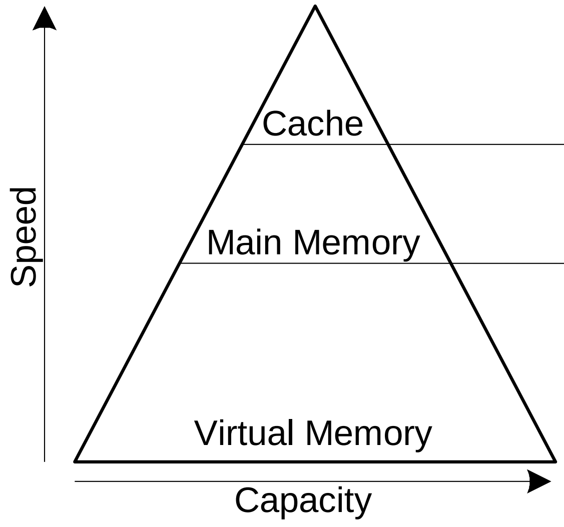
But can only choose two!



Virtual Memory

- Gives the illusion of bigger memory
- Main memory (DRAM) acts as cache for hard disk

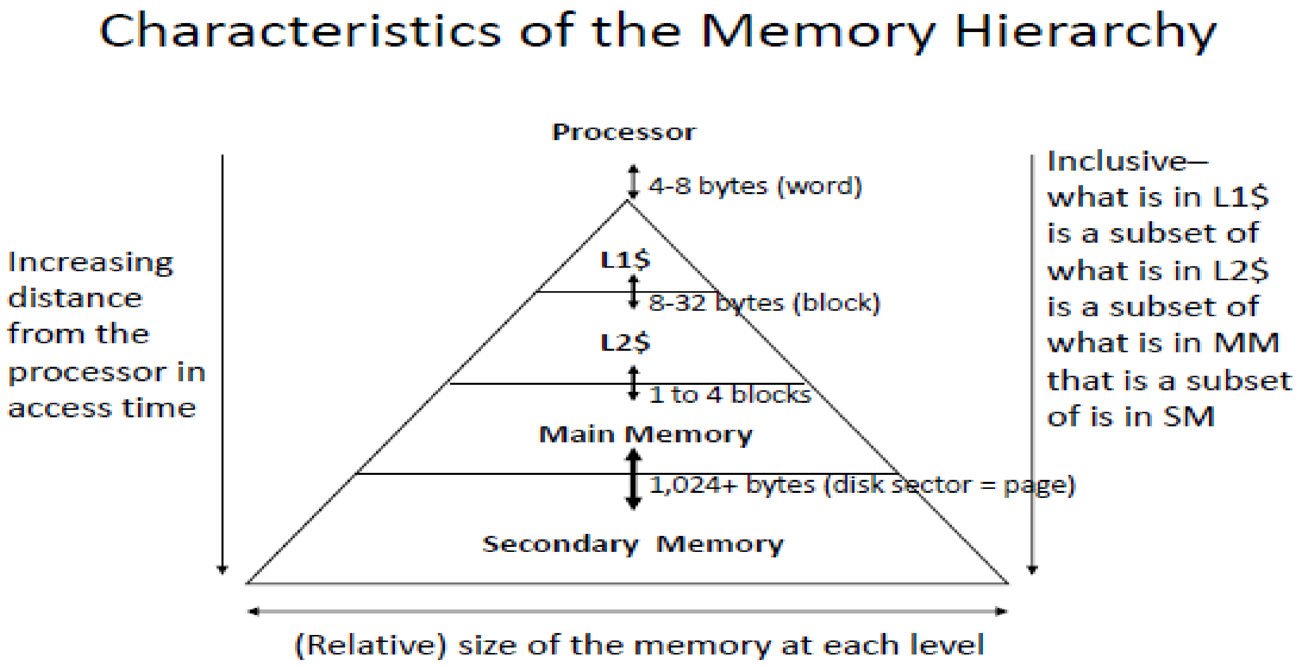
Memory Hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

- **Physical Memory:** DRAM (Main Memory)
- **Virtual Memory:** Hard drive
  - Slow, Large, Cheap

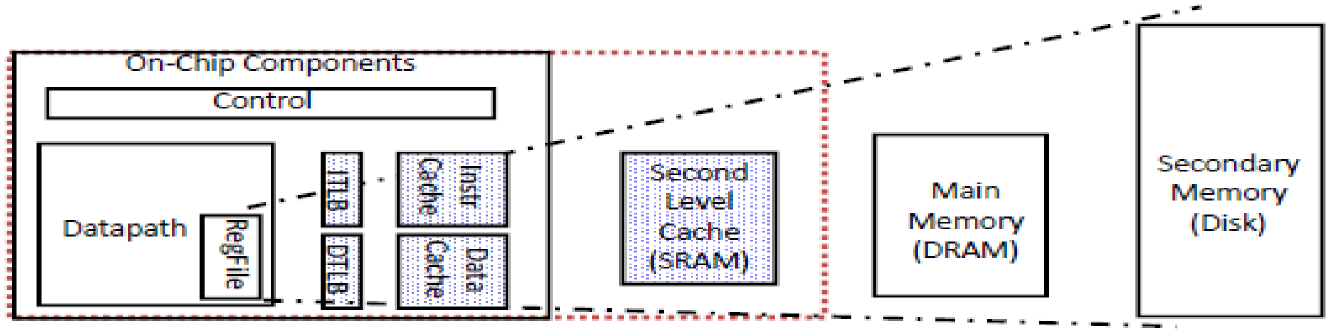
Adapted from misc sources


Characteristics of the Memory Hierarchy
Processor
4-8 bytes (word)
L1$
8-32 bytes (block)
L2$
1 to 4 blocks
Main Memory
1,024+ bytes (disk sector = page)
Secondary Memory
Increasing distance from the processor in access time
Inclusive– what is in L1$ is a subset of what is in L2$ is a subset of what is in MM that is a subset of is in SM
(Relative) size of the memory at each level

Adapted from misc sources

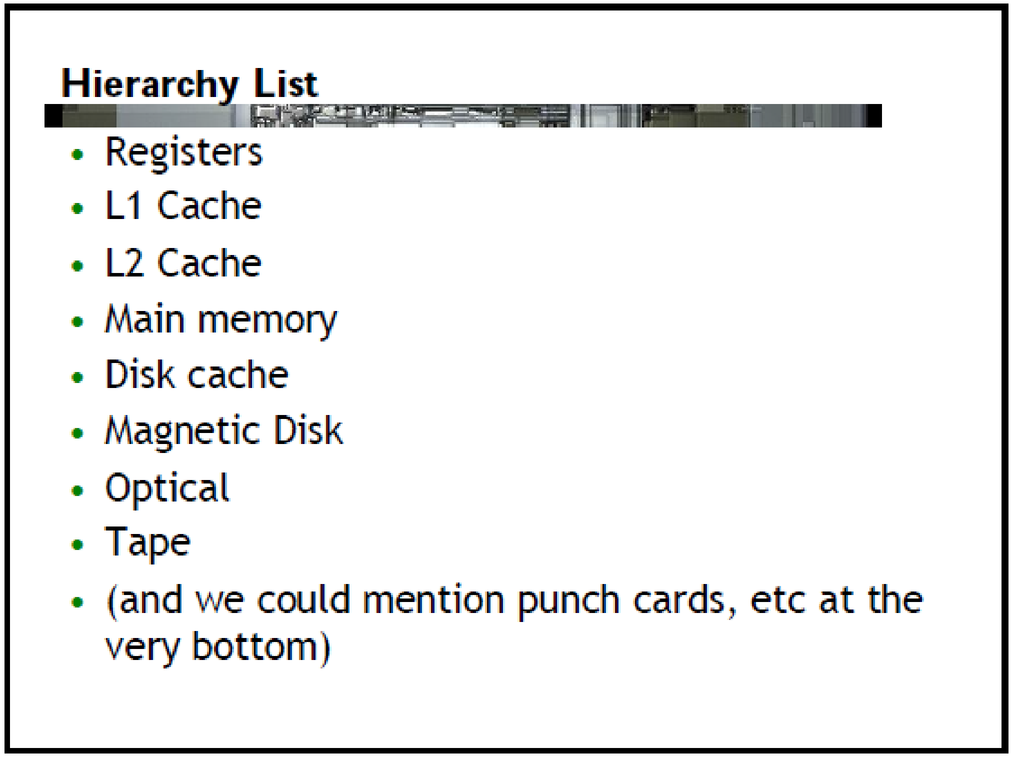
A Typical Memory Hierarchy

- Take advantage of the principle of locality to present the user with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology



Speed (%cycles):	½'s	1's	10's	100's	10,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost:	highest				lowest

Adapted from misc sources



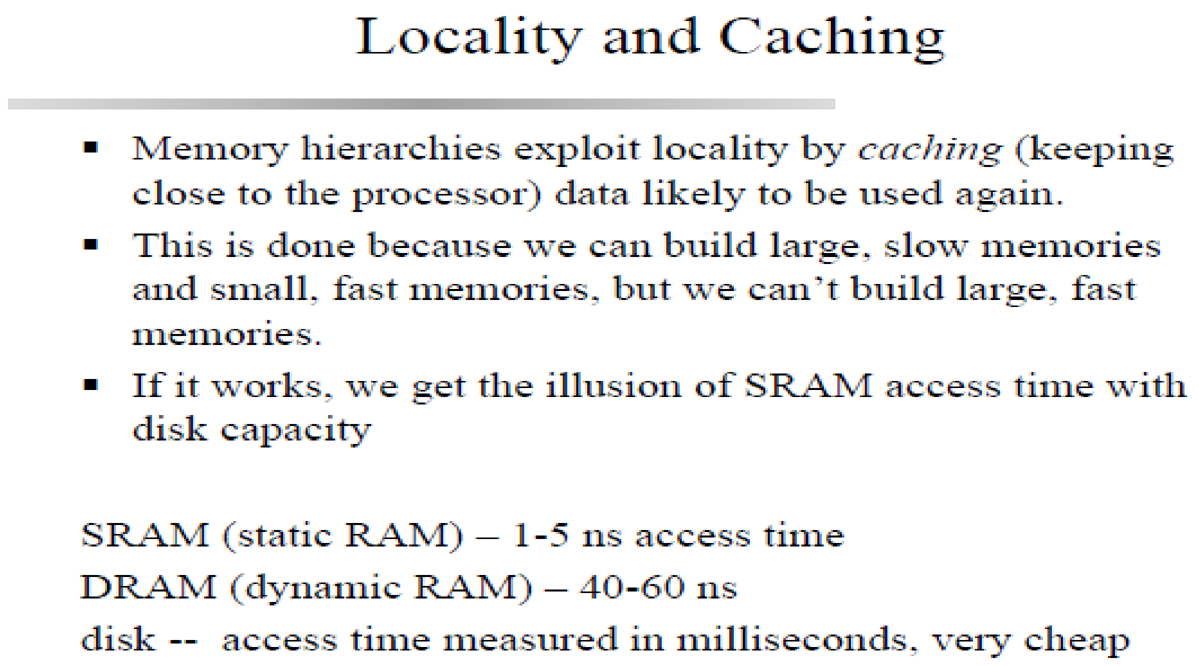
Hierarchy List

- Registers
- L1 Cache
- L2 Cache
- Main memory
- Disk cache
- Magnetic Disk
- Optical
- Tape
- (and we could mention punch cards, etc at the very bottom)

Locality

Exploit locality to make memory accesses fast

- **Temporal Locality:** Near in Time
  - Locality in time
  - If data used recently, likely to use it again soon
  - **How to exploit:** keep recently accessed data in higher levels of memory hierarchy
- **Spatial Locality:** Near in space/distance
  - Locality in space
  - If data used recently, likely to use nearby data soon
  - **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too



Locality and Caching

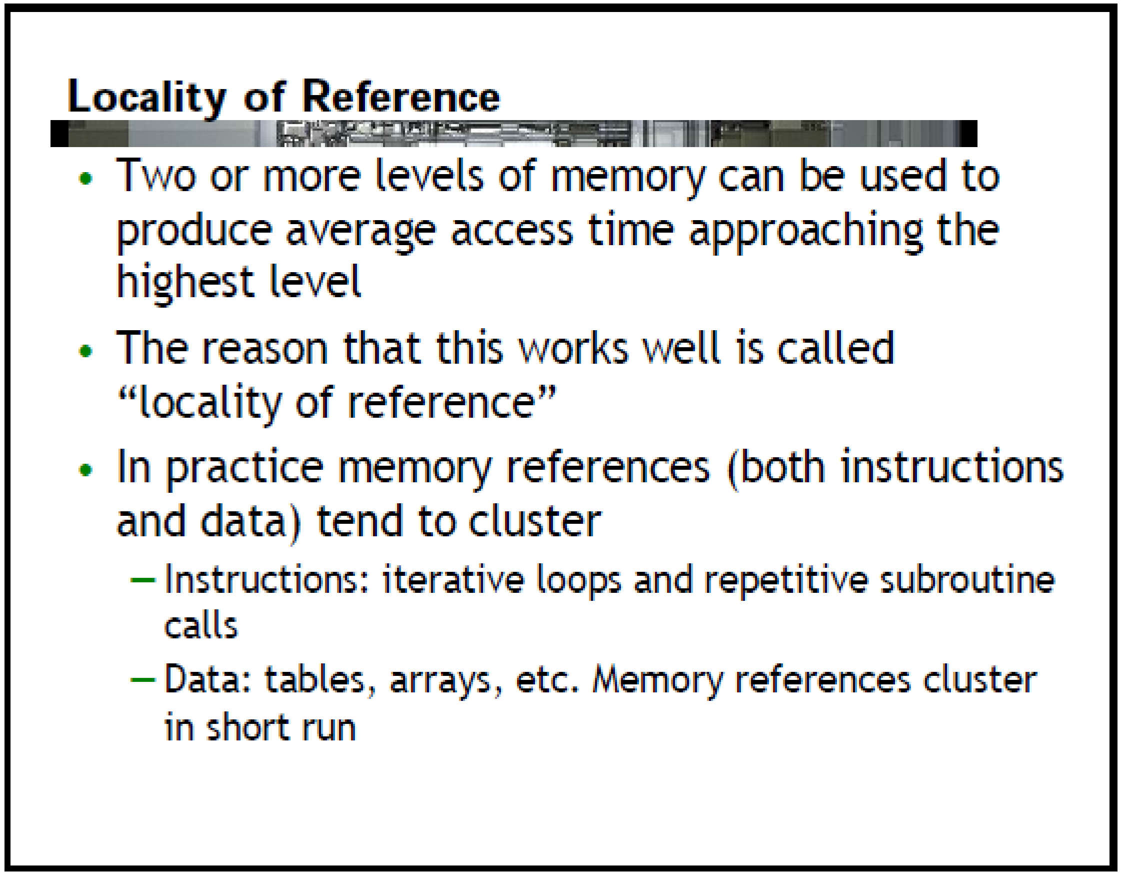
- Memory hierarchies exploit locality by *caching* (keeping close to the processor) data likely to be used again.
- This is done because we can build large, slow memories and small, fast memories, but we can't build large, fast memories.
- If it works, we get the illusion of SRAM access time with disk capacity

SRAM (static RAM) – 1-5 ns access time

DRAM (dynamic RAM) – 40-60 ns

disk -- access time measured in milliseconds, very cheap

Adapted from misc sources



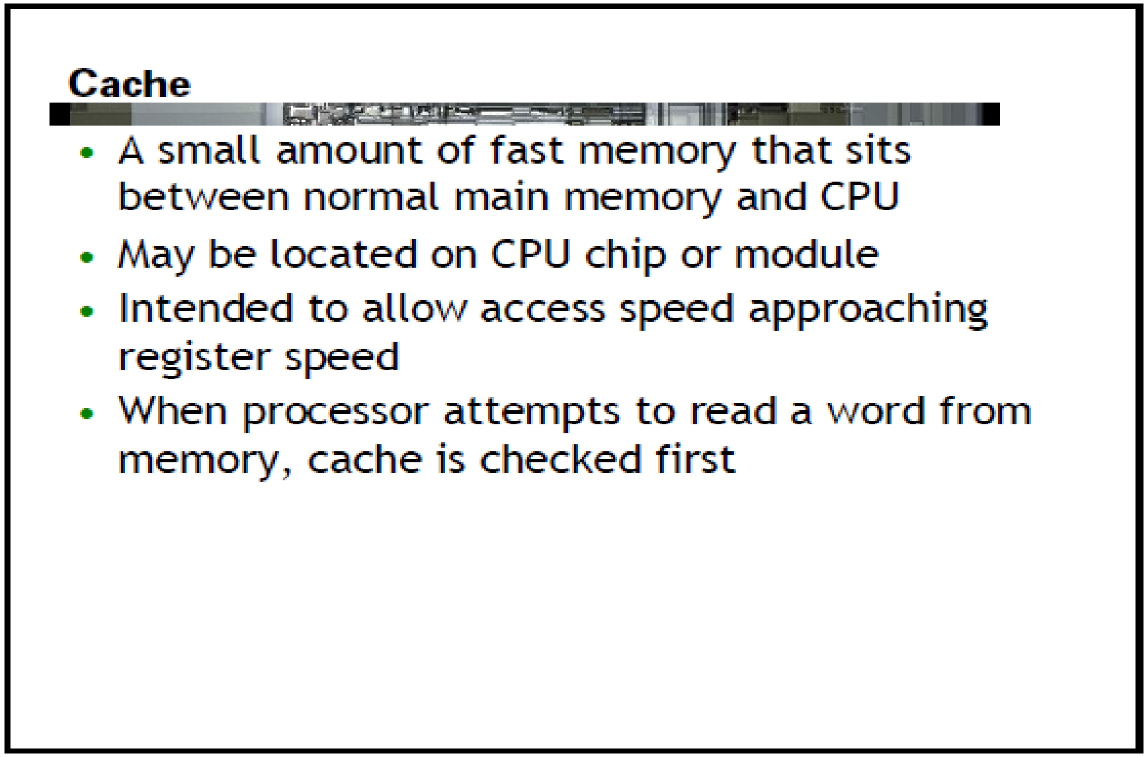
Locality of Reference

- Two or more levels of memory can be used to produce average access time approaching the highest level
- The reason that this works well is called "locality of reference"
- In practice memory references (both instructions and data) tend to cluster
  - Instructions: iterative loops and repetitive subroutine calls
  - Data: tables, arrays, etc. Memory references cluster in short run

Cache

- Highest level in memory hierarchy
- Fast (typically ~ 1 cycle access time)
- Ideally supplies most data to processor
- Usually holds most recently accessed data

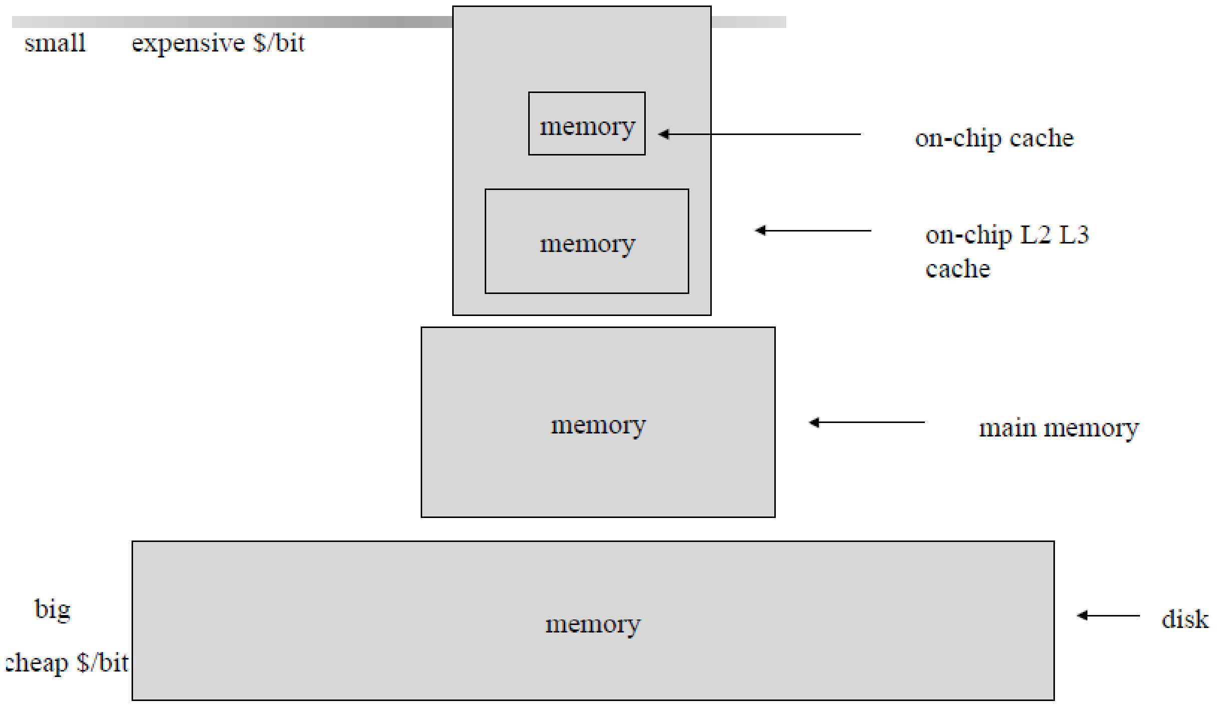
Adapted from misc sources



Cache

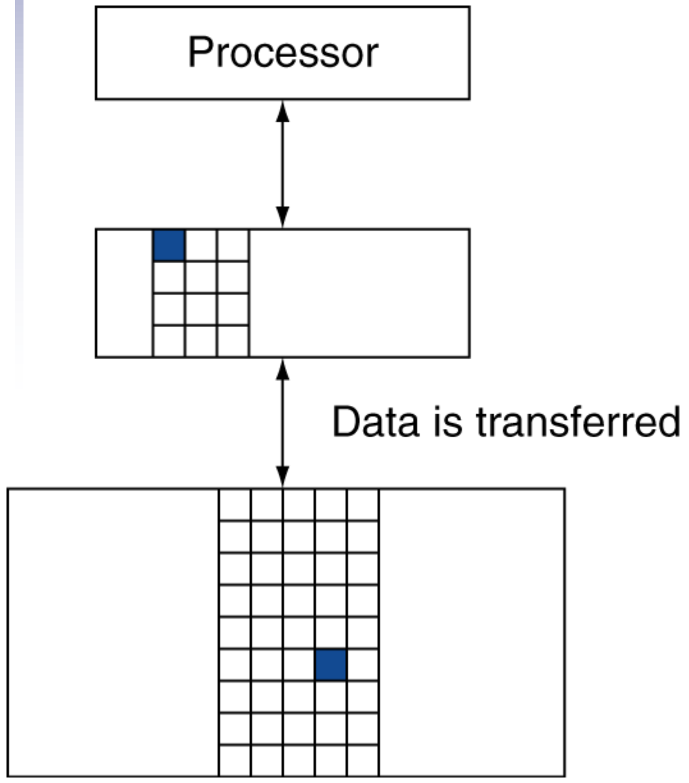
- A small amount of fast memory that sits between normal main memory and CPU
- May be located on CPU chip or module
- Intended to allow access speed approaching register speed
- When processor attempts to read a word from memory, cache is checked first

A typical memory hierarchy



•*so then where is my program and data??*

Memory Hierarchy Levels



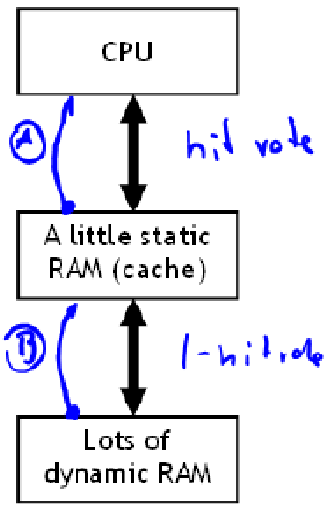
- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level

RUTGERS

Adapted from misc sources

Memory System Performance

- To examine the performance of a memory system, we need to focus on a couple of important factors.
  - A — How long does it take to send data from the cache to the CPU?
  - B — 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.
  - A — 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.
  - B — 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.



Memory Performance

- **Hit:** data found in that level of memory hierarchy
- **Miss:** data not found (must go to next level)

Hit Rate	= # hits / # memory accesses
	= 1 – Miss Rate
Miss Rate	= # misses / # memory accesses
	= 1 – Hit Rate

- **Average memory access time (AMAT):** average time for processor to access data

AMAT $= t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

Hit Rate = 1250/2000 **= 0.625**

Miss Rate = 750/2000 = **0.375** = 1 – Hit Rate

Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- t_{cache} = 1 cycle, t_{MM} = 100 cycles
- **What is the AMAT of the program from Example 1?**

Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- t_{cache} = 1 cycle, t_{MM} = 100 cycles
- **What is the AMAT of the program from Example 1?**

AMAT $= t_{\text{cache}} + MR_{\text{cache}}(t_{MM})$

$= [1 + 0.375(100)]$ cycles

= **38.5 cycles**

Gene Amdahl, 1922-

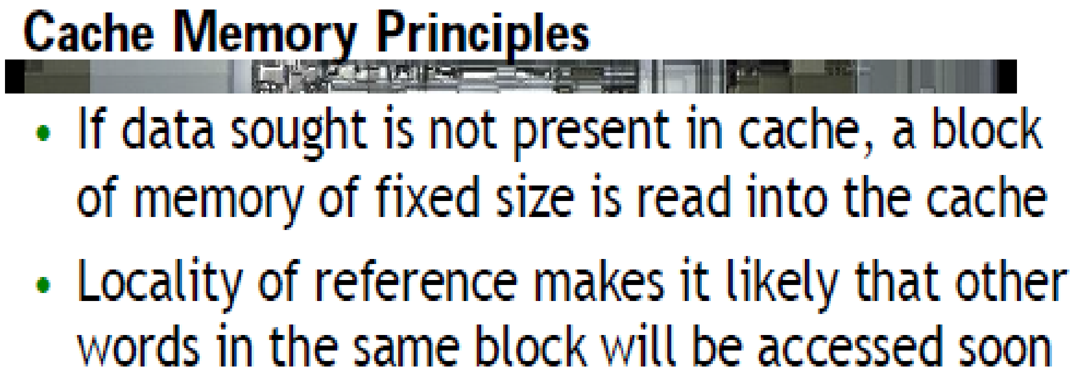
- **Amdahl's Law:** the effort spent increasing the performance of a subsystem is wasted unless the subsystem affects a large percentage of overall performance
- Co-founded 3 companies, including one called Amdahl Corporation in 1970

Cache Design Questions

- What data is held in the cache?
- How is data found?
- What data is replaced?

Focus on data loads, but stores follow same principles

Adapted from misc sources



Cache Memory Principles

- If data sought is not present in cache, a block of memory of fixed size is read into the cache
- Locality of reference makes it likely that other words in the same block will be accessed soon

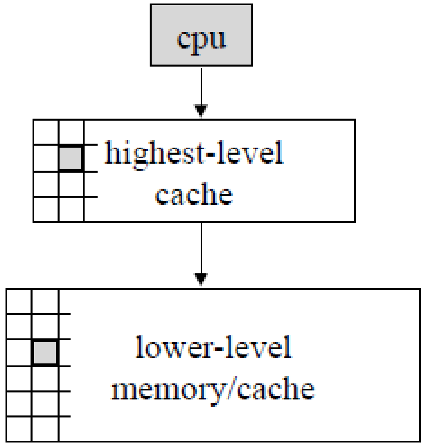
What data is held in the cache?

- Ideally, cache anticipates needed data and puts it in cache
- But impossible to predict future
- Use past to predict future – temporal and spatial locality:
  - **Temporal locality:** copy newly accessed data into cache
  - **Spatial locality:** copy neighboring data into cache too

Adapted from misc sources

Cache Fundamentals

- *cache hit* -- an access where the data is found in the cache.
- *cache miss* -- an access which isn't
- *hit time* -- time to access the higher cache
- *miss penalty* -- time to move data from lower level to upper, then to cpu
- *hit ratio* -- percentage of time the data is found in the higher cache
- *miss ratio* -- (1 - hit ratio)

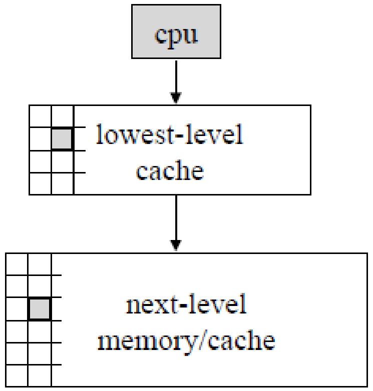


Adapted from misc sources

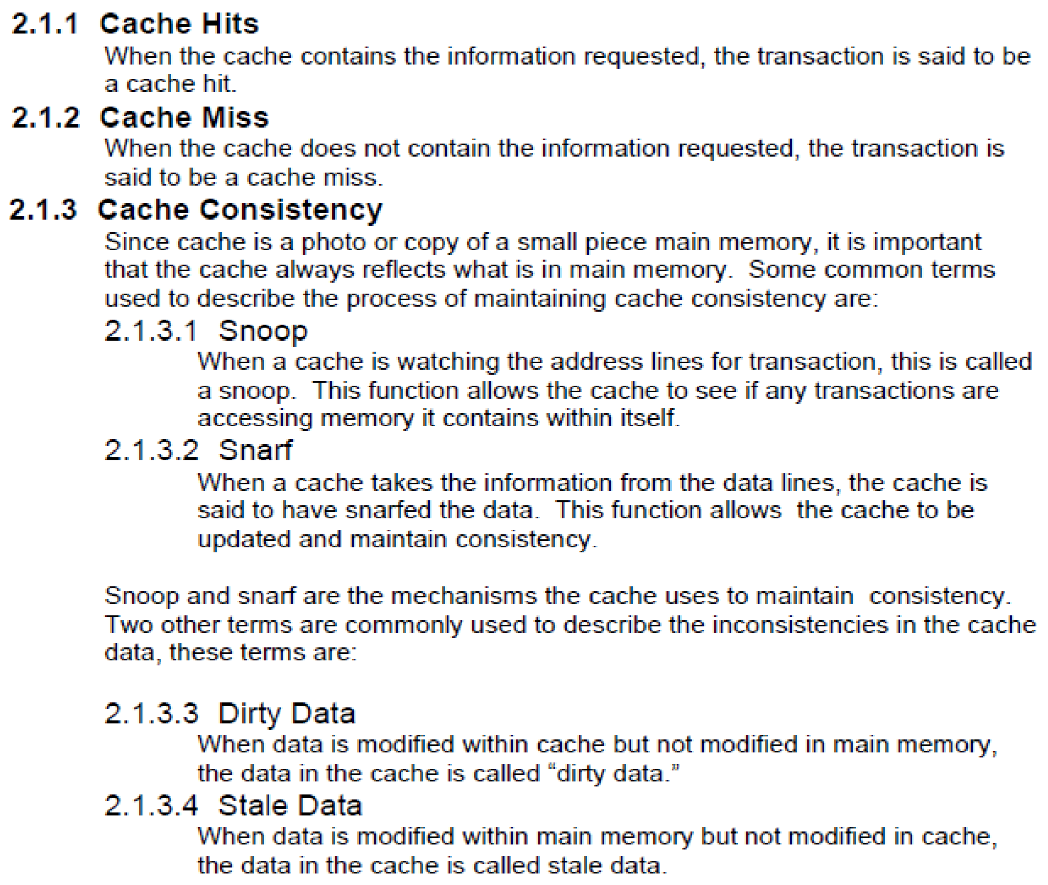
Cache Fundamentals, cont.

- *cache block size* or *cache line size*-- the amount of data that gets transferred on a cache miss.
- *instruction cache* -- cache that only holds instructions.
- *data cache* -- cache that only caches data.
- *unified cache* -- cache that holds both.

  (L1 is unified → "princeton architecture")



Adapted from misc sources



2.1.1 Cache Hits

When the cache contains the information requested, the transaction is said to be a cache hit.

2.1.2 Cache Miss

When the cache does not contain the information requested, the transaction is said to be a cache miss.

2.1.3 Cache Consistency

Since cache is a photo or copy of a small piece main memory, it is important that the cache always reflects what is in main memory. Some common terms used to describe the process of maintaining cache consistency are:

2.1.3.1 Snoop

When a cache is watching the address lines for transaction, this is called a snoop. This function allows the cache to see if any transactions are accessing memory it contains within itself.

2.1.3.2 Snarf

When a cache takes the information from the data lines, the cache is said to have snarfed the data. This function allows the cache to be updated and maintain consistency.

Snoop and snarf are the mechanisms the cache uses to maintain consistency. Two other terms are commonly used to describe the inconsistencies in the cache data, these terms are:

2.1.3.3 Dirty Data

When data is modified within cache but not modified in main memory, the data in the cache is called "dirty data."

2.1.3.4 Stale Data

When data is modified within main memory but not modified in cache, the data in the cache is called stale data.

Cache Terminology

- **Capacity (*C*):**
  - number of data bytes in cache
- **Block size (*b*):**
  - bytes of data brought into cache at once
- **Number of blocks (*B = C/b*):**
  - number of blocks in cache: $B = C/b$
- **Degree of associativity (*N*):**
  - number of blocks in a set
- **Number of sets (*S = B/N*):**
  - each memory address maps to exactly one cache set

How is data found?

- Cache organized into *S* sets
- Each memory address maps to exactly one set
- Caches categorized by # of blocks in a set:
  - **Direct mapped:** 1 block per set
  - ***N*-way set associative:** *N* blocks per set
  - **Fully associative:** all cache blocks in 1 set
- Examine each organization for a cache with:
  - Capacity (*C* = 8 words)
  - Block size (*b* = 1 word)
  - So, number of blocks (*B* = 8)

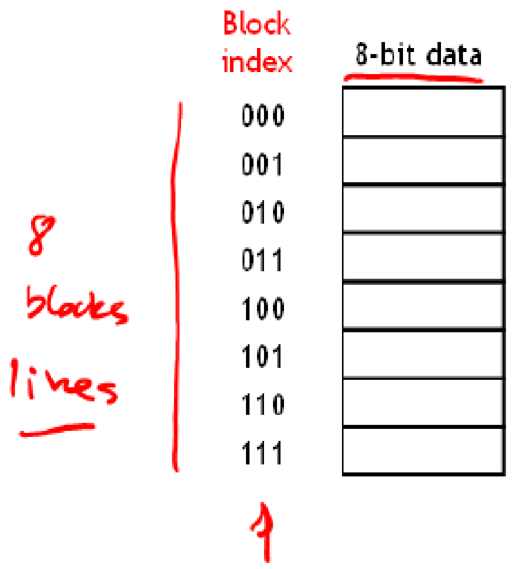
Example Cache Parameters

- ***C*** **= 8** words (capacity)
- ***b*** **= 1** word (block size)
- So, ***B*** **= 8** (# of blocks)

Ridiculously small, but will illustrate organizations

Adapted from misc sources

- 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.

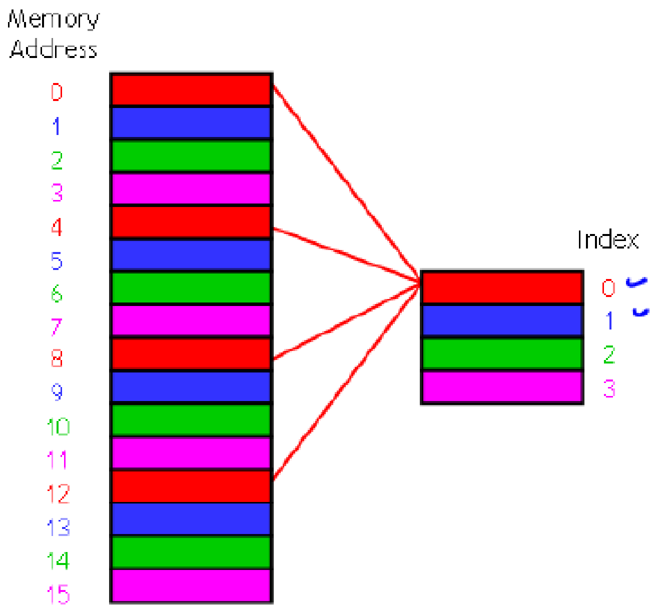


2

Adapted from misc sources

- 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?

address mod 4



4

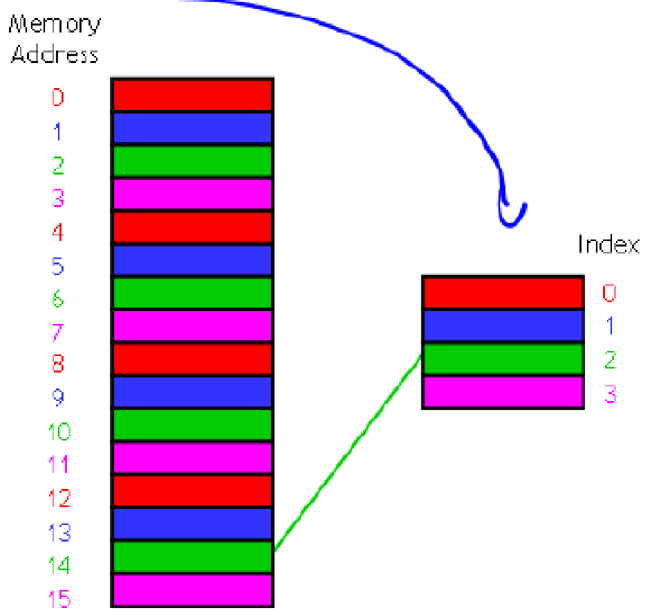
Adapted from misc sources

- 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.

  $14 \bmod 4 = 2$



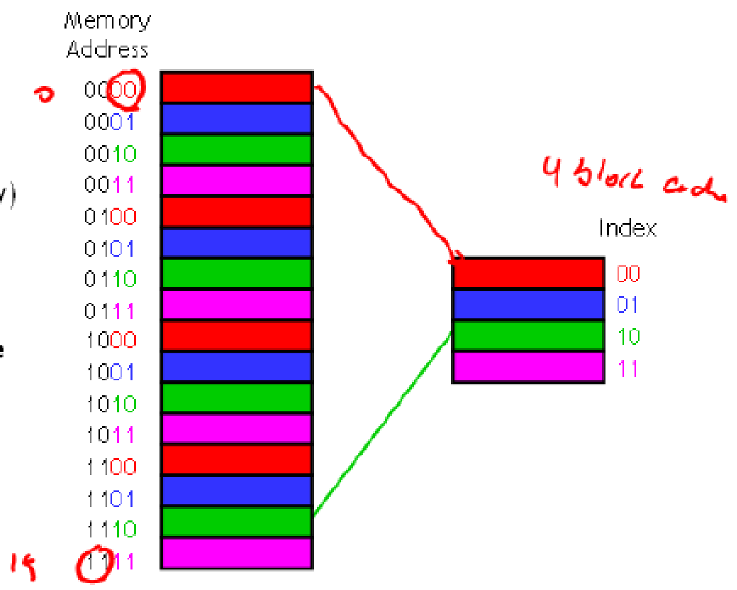
5

RUTGERS

Adapted from misc sources

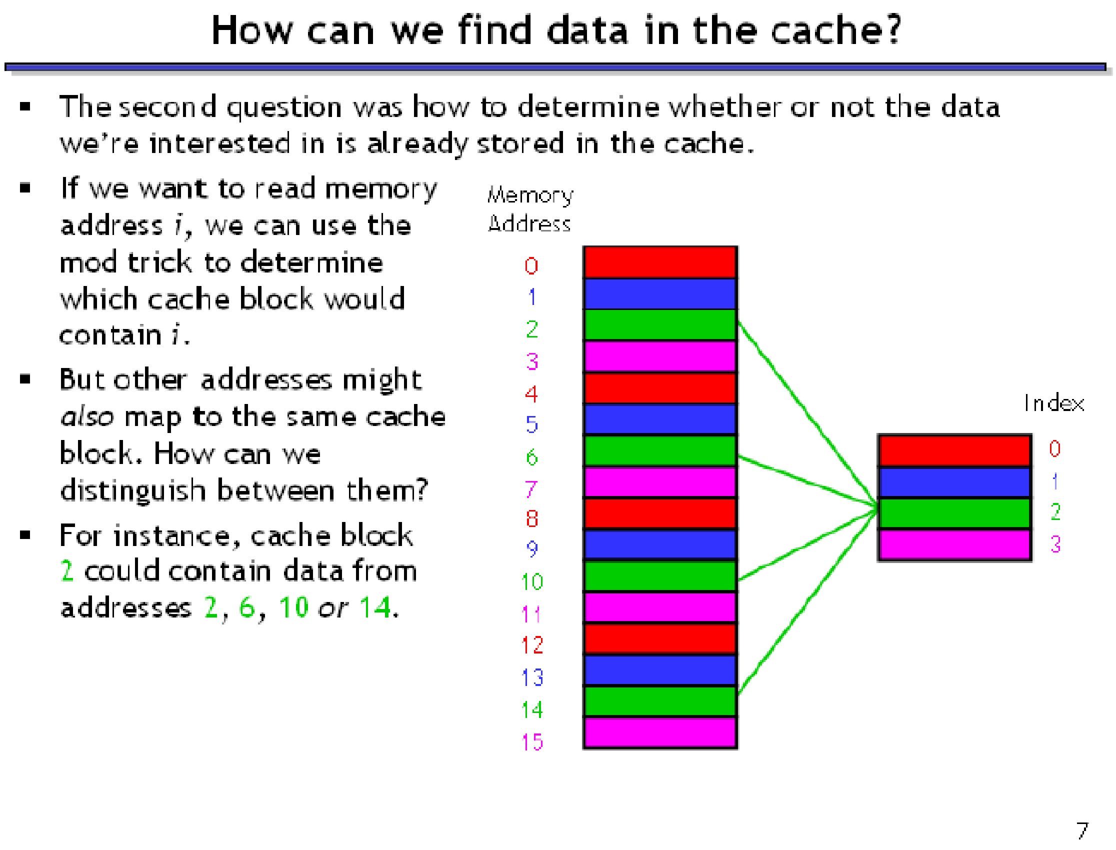
...or least-significant bits

- 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.



6

Adapted from misc sources



How can we find data in the cache?

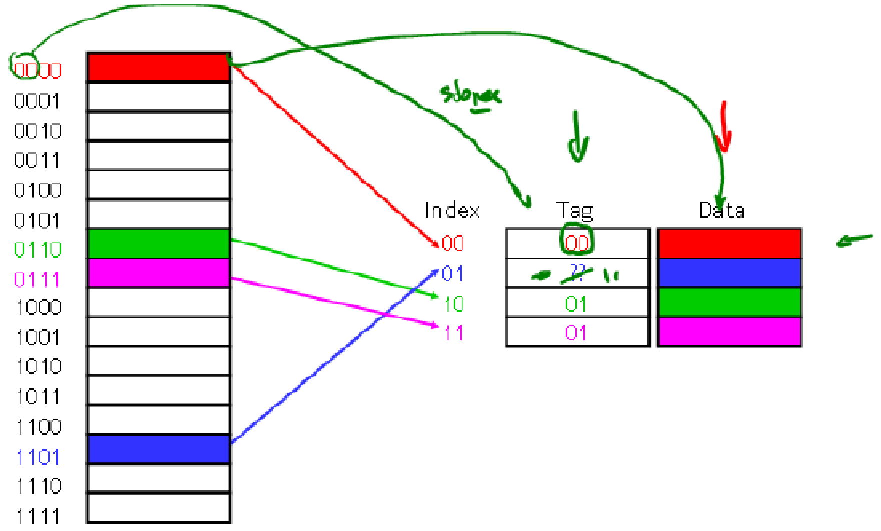
- The second question was how to determine whether or not the data we're interested in is already stored in the cache.
- If we want to read memory address *i*, we can use the mod trick to determine which cache block would contain *i*.
- But other addresses might *also* map to the same cache block. How can we distinguish between them?
- For instance, cache block 2 could contain data from addresses 2, 6, 10 *or* 14.

7

Adapted from misc sources

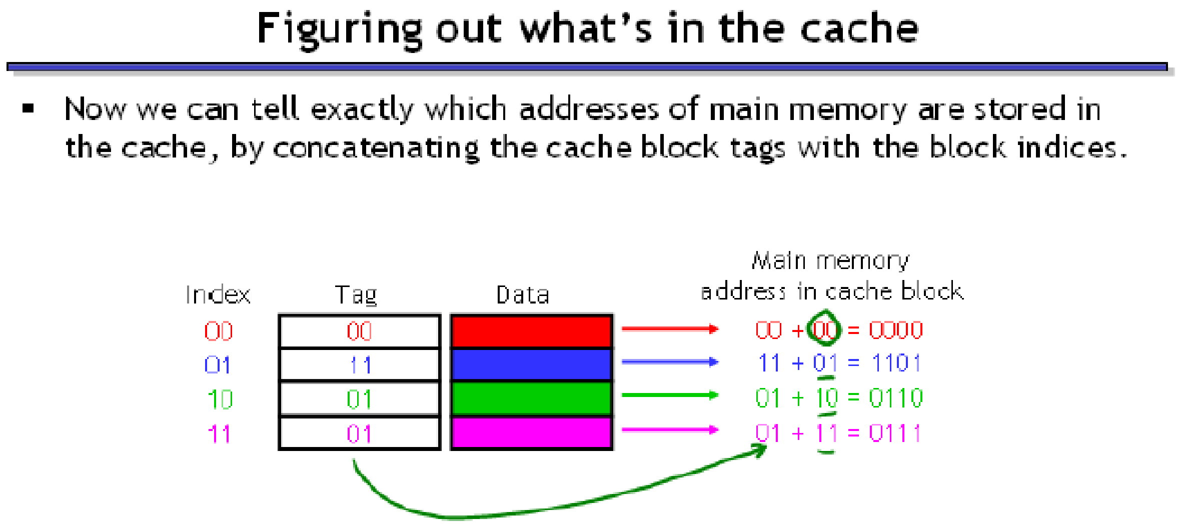
Adding tags

- 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.



8

Adapted from misc sources

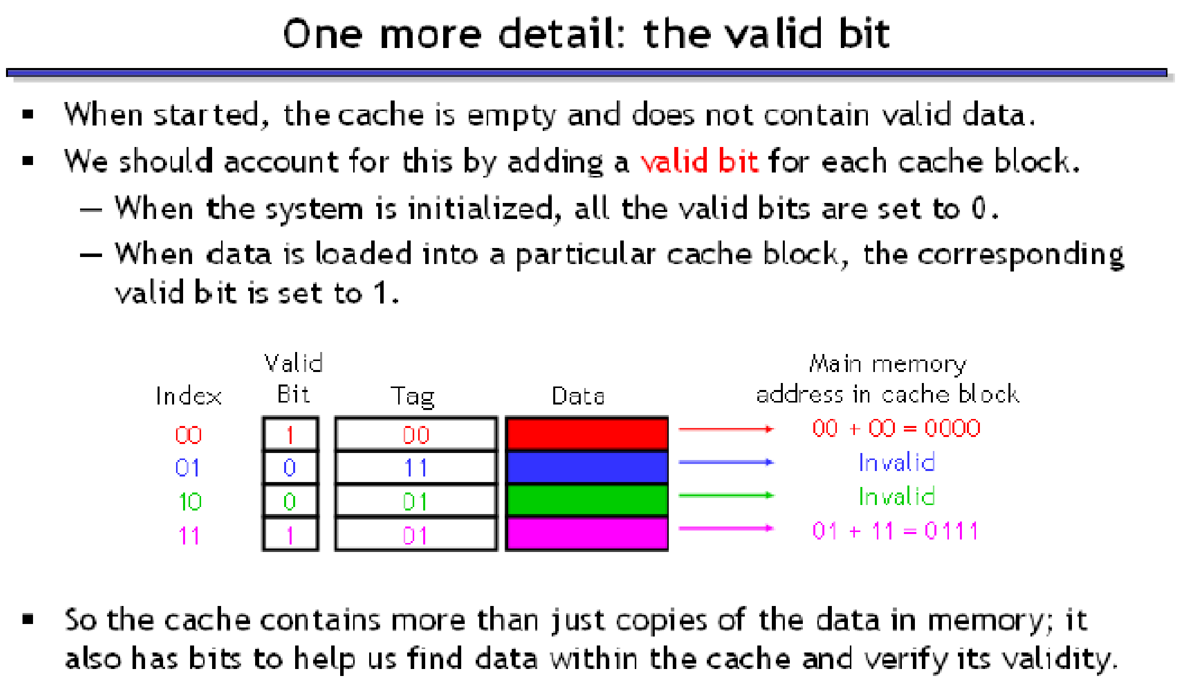


Figuring out what's in the cache

- 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

Adapted from misc sources



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.

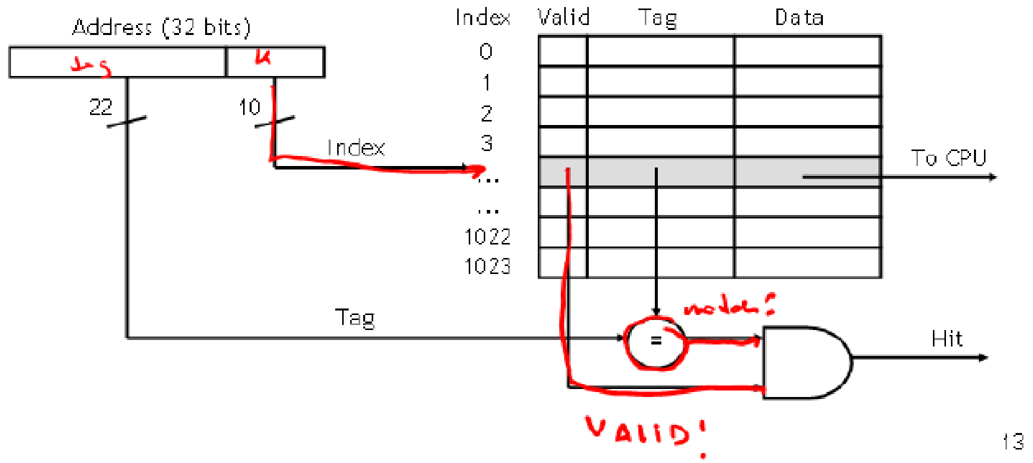
Index	Valid Bit	Tag	Data	Main memory address in cache block
00	1	00	🟥	00 + 00 = 0000
01	0	11	🟦	Invalid
10	0	01	🟩	Invalid
11	1	01	🟪	01 + 11 = 0111

- 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.

Adapted from misc sources

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.

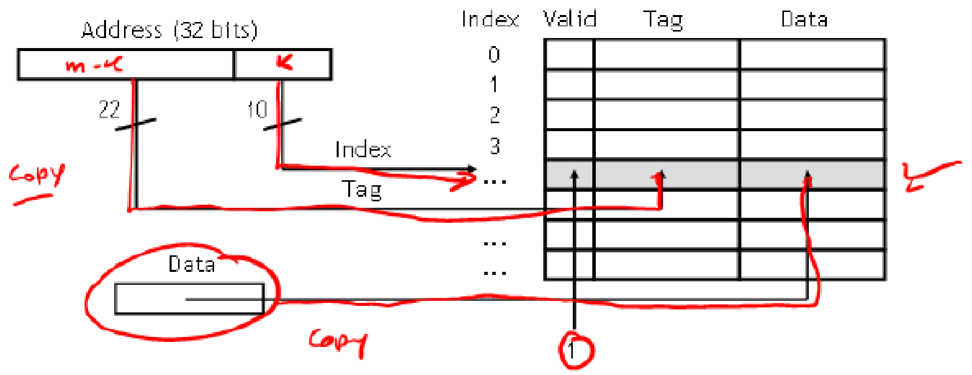


13

Adapted from misc sources

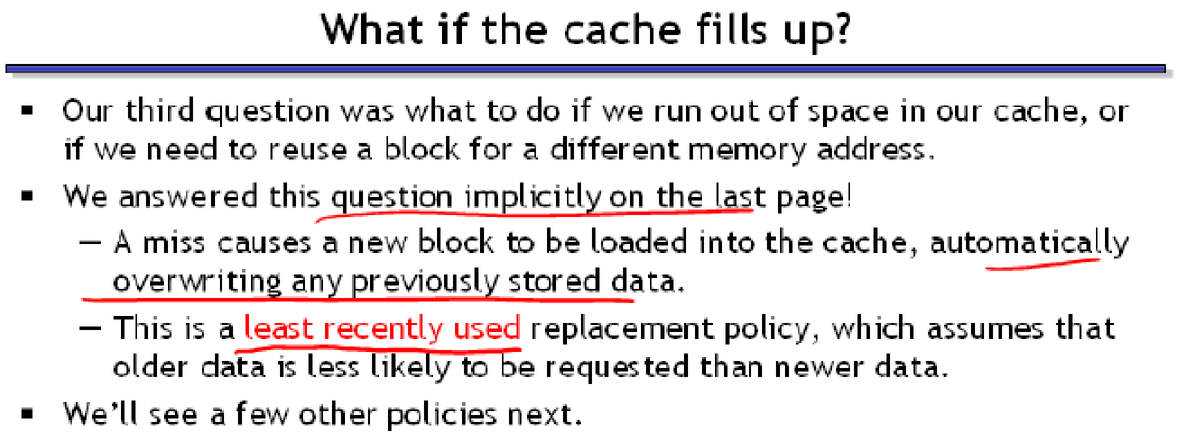
Loading a block into the cache

- 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.



16

Adapted from misc sources

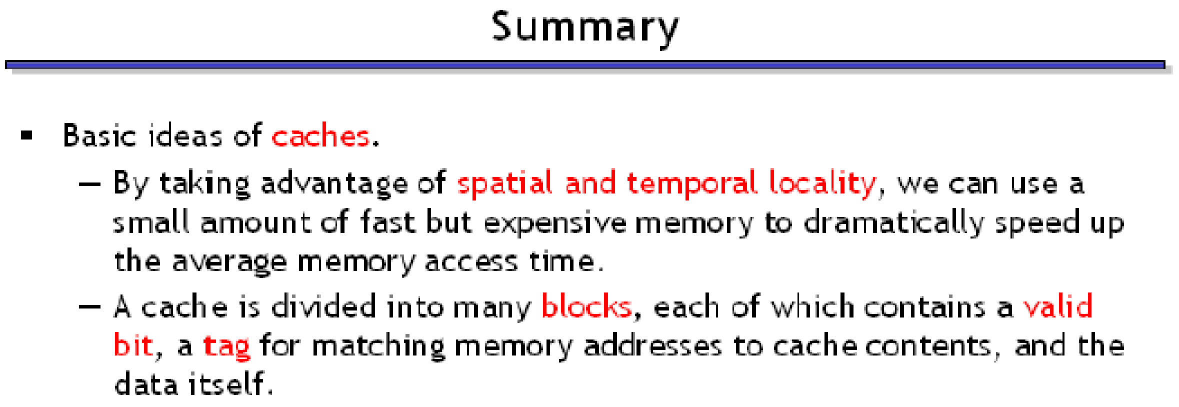


What if the cache fills up?

- 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.

RUTGERS

Adapted from misc sources



Summary

- 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.

Adapted from misc sources

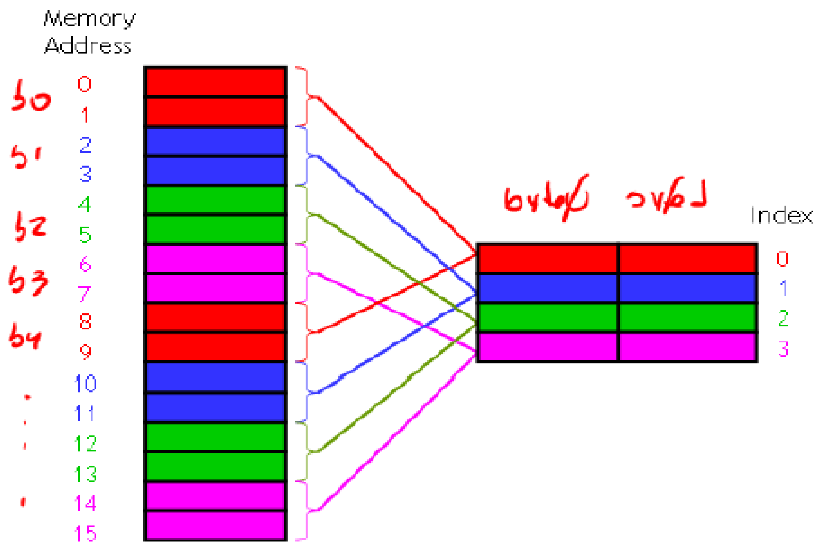
Spatial locality

- 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?

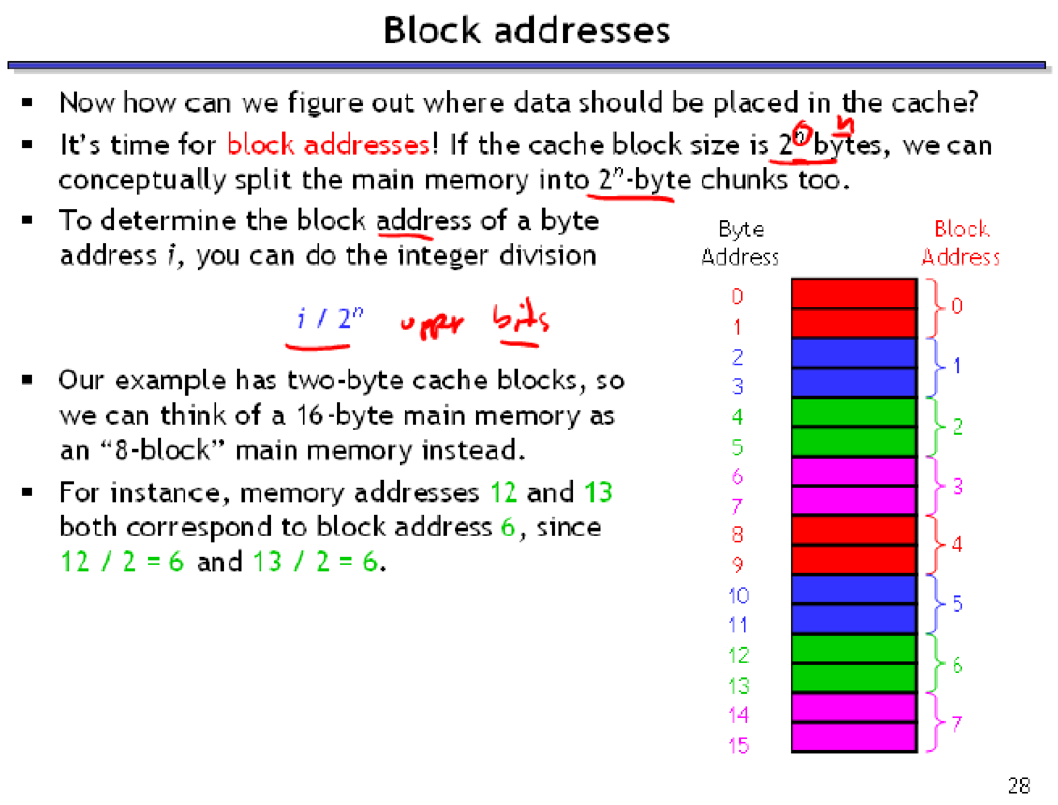
Spatial locality

- 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.



Adapted from misc sources



Block addresses

- Now how can we figure out where data should 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

$$i \mathbin{/} 2^n$$ upper bits

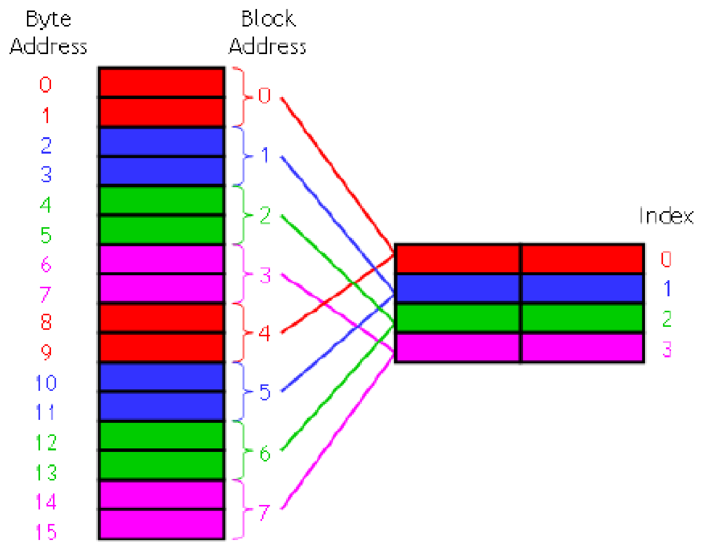
- Our example has two-byte cache blocks, so we can think of a 16-byte main memory as an "8-block" main memory instead.
- For instance, memory addresses 12 and 13 both correspond to block address 6, since 12 / 2 = 6 and 13 / 2 = 6.

28

Adapted from misc sources

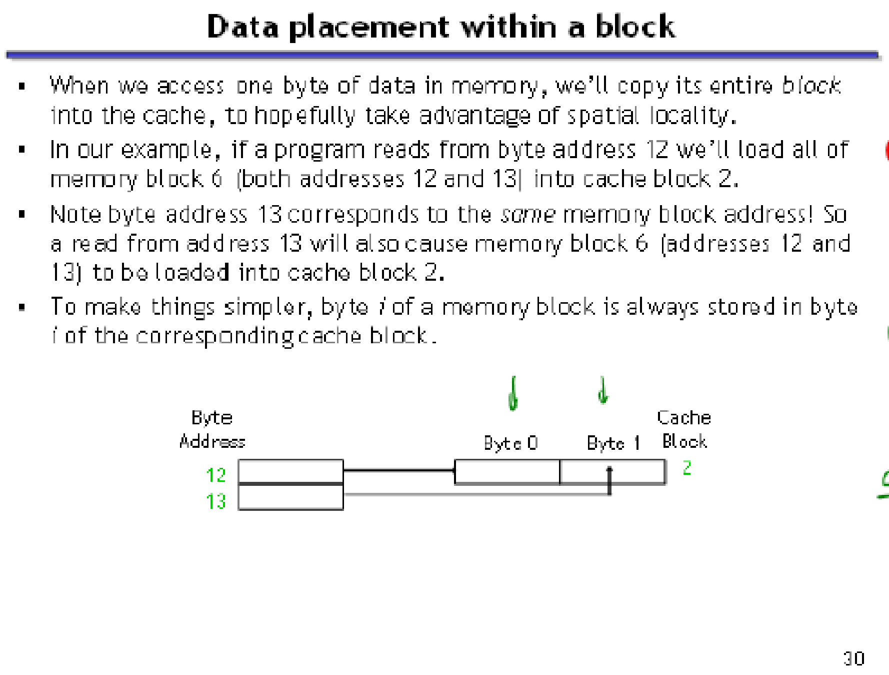
Cache mapping

- 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.



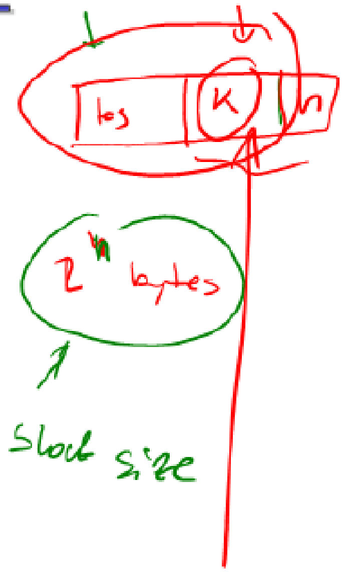
29

Adapted from misc sources

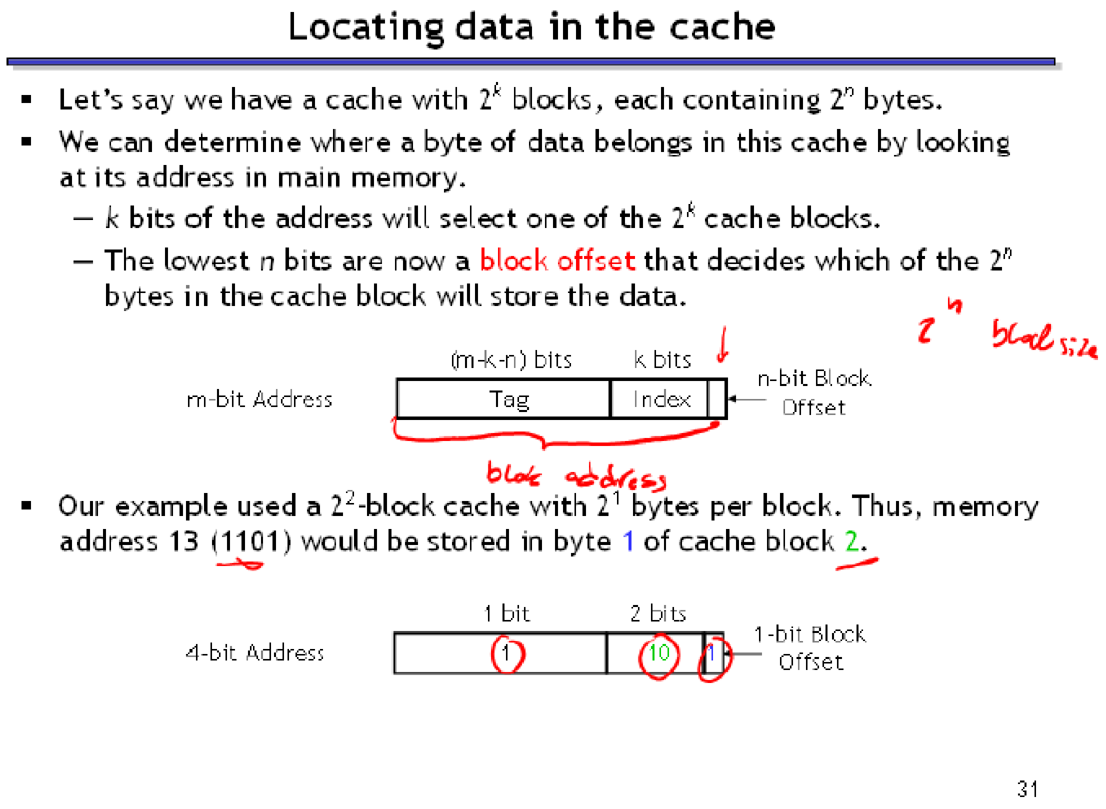


Data placement within a block

- When we access one byte of data in memory, we'll copy its entire *block* into the cache, to hopefully 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.



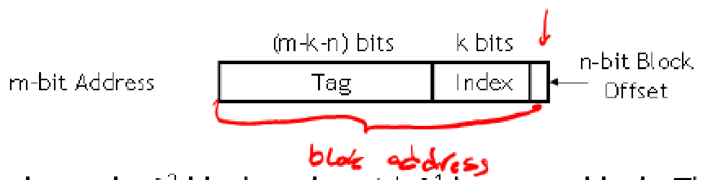
Adapted from misc sources



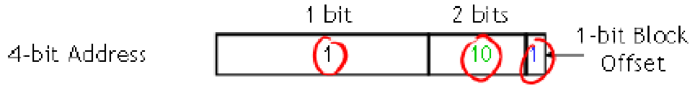
Locating data in the cache

- 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.

2^n block size

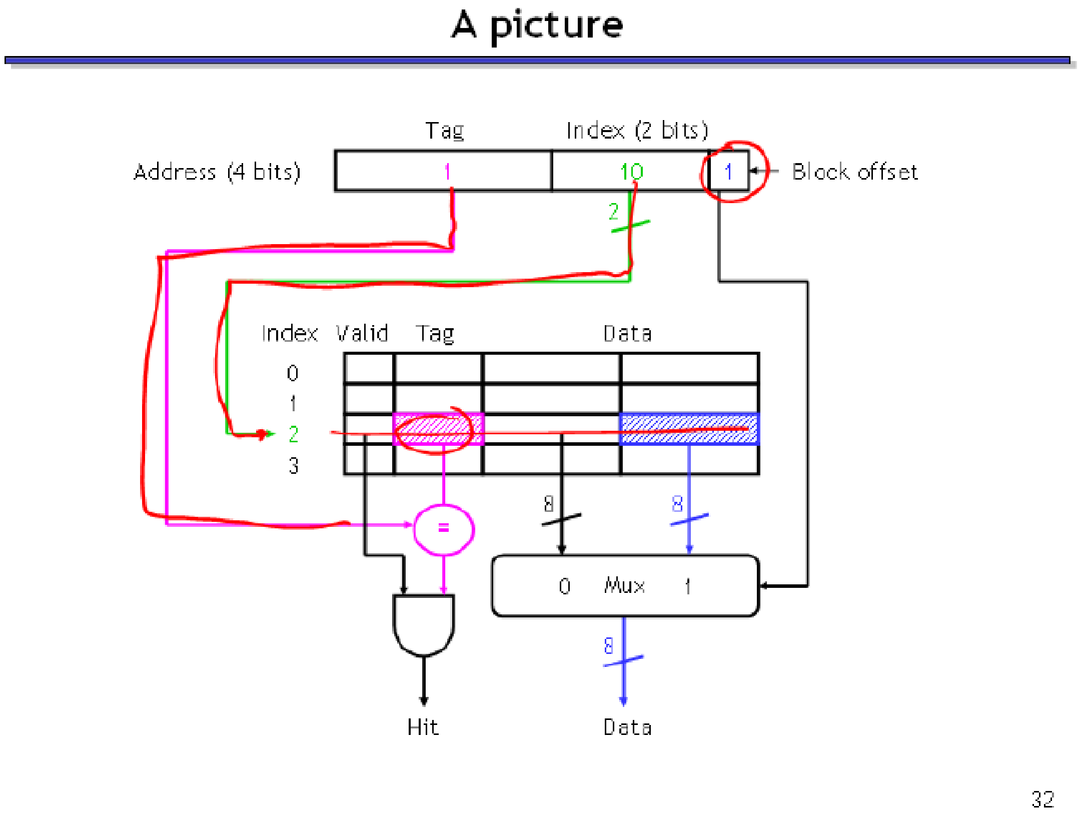


- Our example used a 2^2-block cache with 2^1 bytes per block. Thus, memory address 13 (1101) would be stored in byte 1 of cache block 2.



31

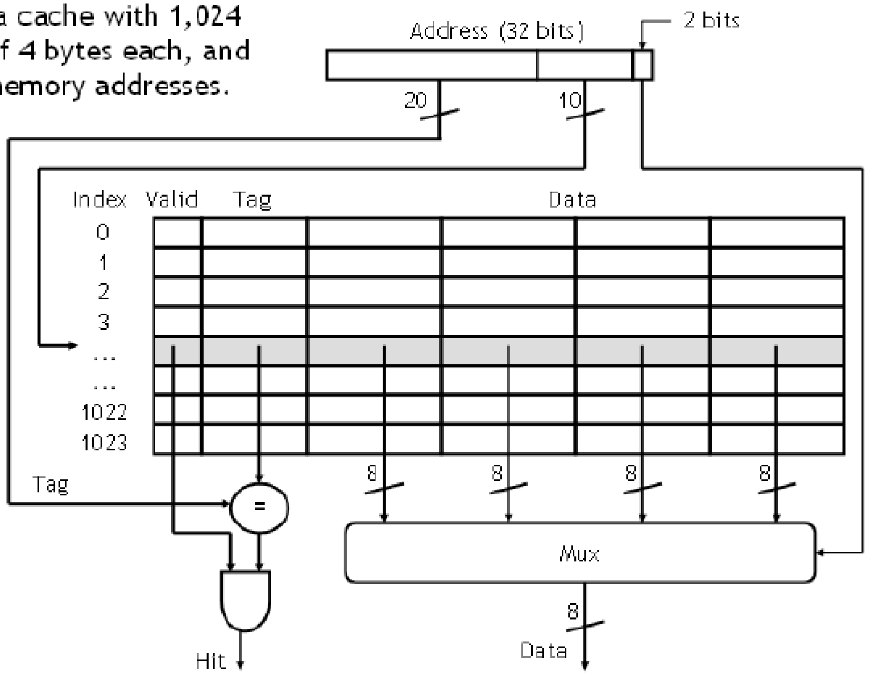
Adapted from misc sources



Adapted from misc sources

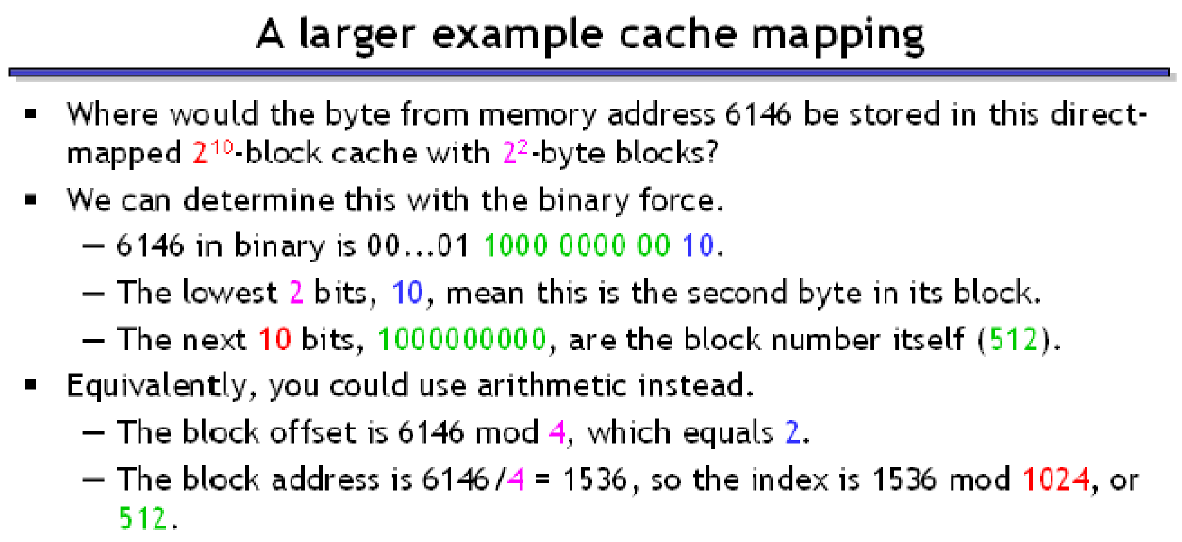
A diagram of a larger example cache

- Here is a cache with 1,024 blocks of 4 bytes each, and 32-bit memory addresses.



36

Adapted from misc sources



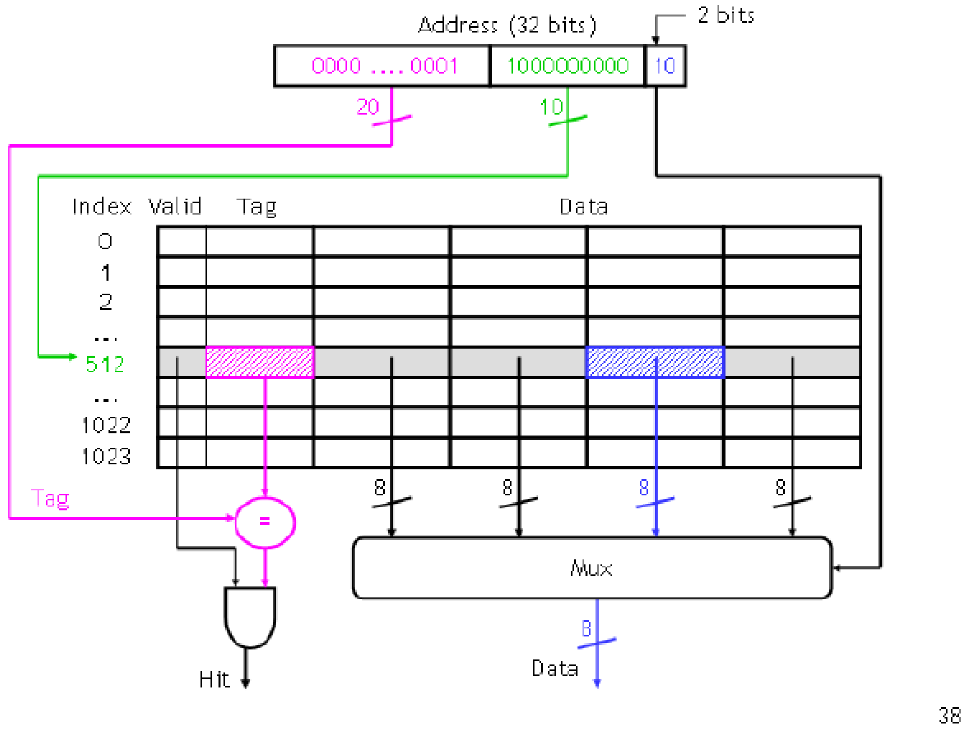
A larger example cache mapping

- Where would the byte from memory address 6146 be stored in this direct-mapped 2^{10}-block cache with 2^2-byte blocks?
- We can determine this with the binary force.
  - 6146 in binary is 00...01 1000 0000 00 10.
  - The lowest 2 bits, 10, mean this is the second byte in its block.
  - The next 10 bits, 1000000000, are the block number itself (512).
- Equivalently, you could use arithmetic instead.
  - The block offset is 6146 mod 4, which equals 2.
  - The block address is 6146/4 = 1536, so the index is 1536 mod 1024, or 512.

RUTGERS

Adapted from misc sources

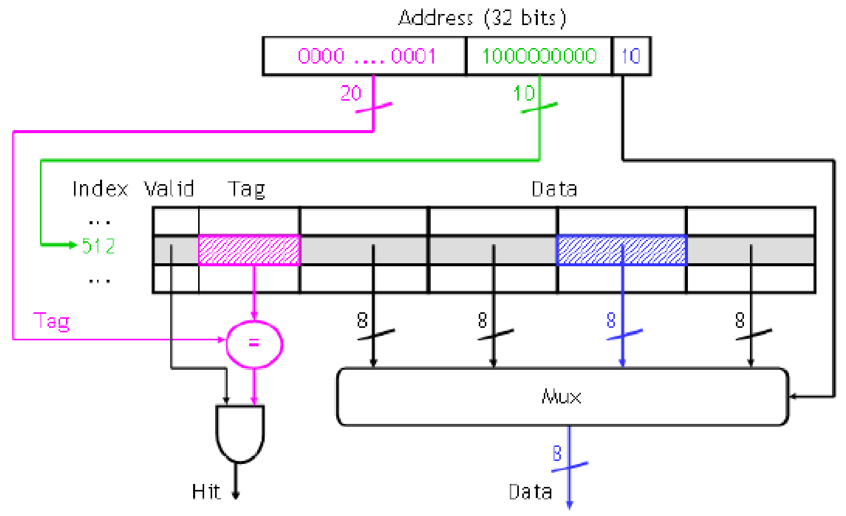
A larger diagram of a larger example cache mapping



Adapted from misc sources

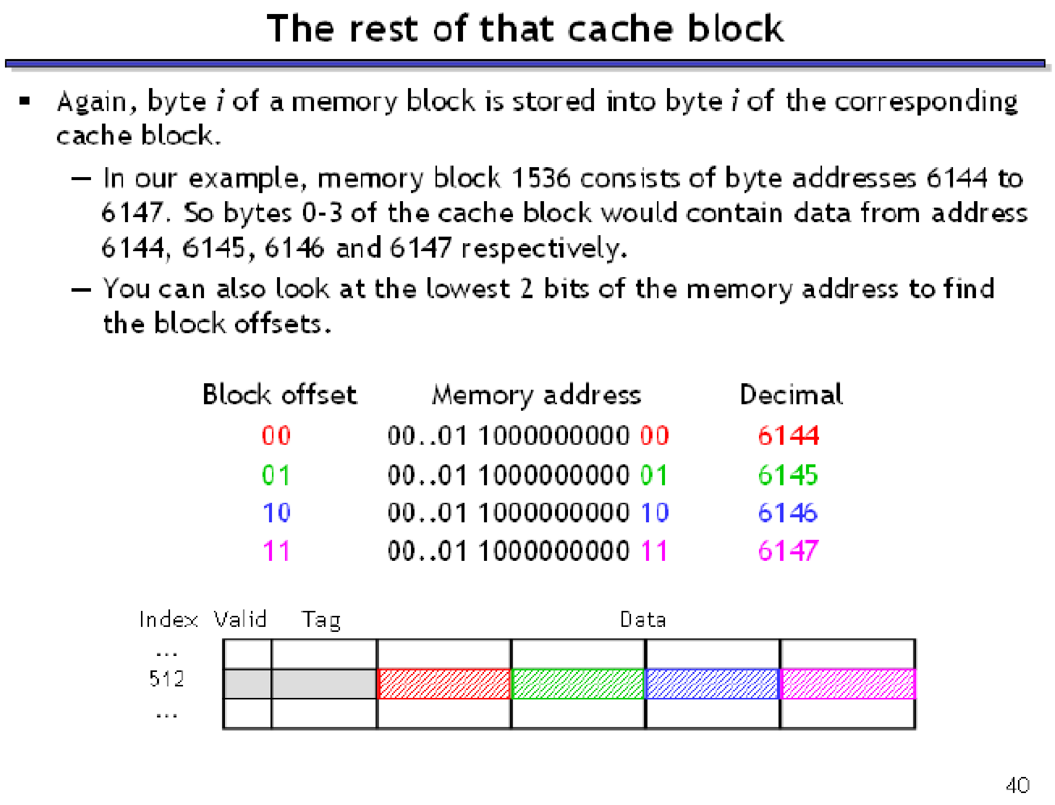
What goes in the rest of that cache block?

- 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).



39

Adapted from misc sources



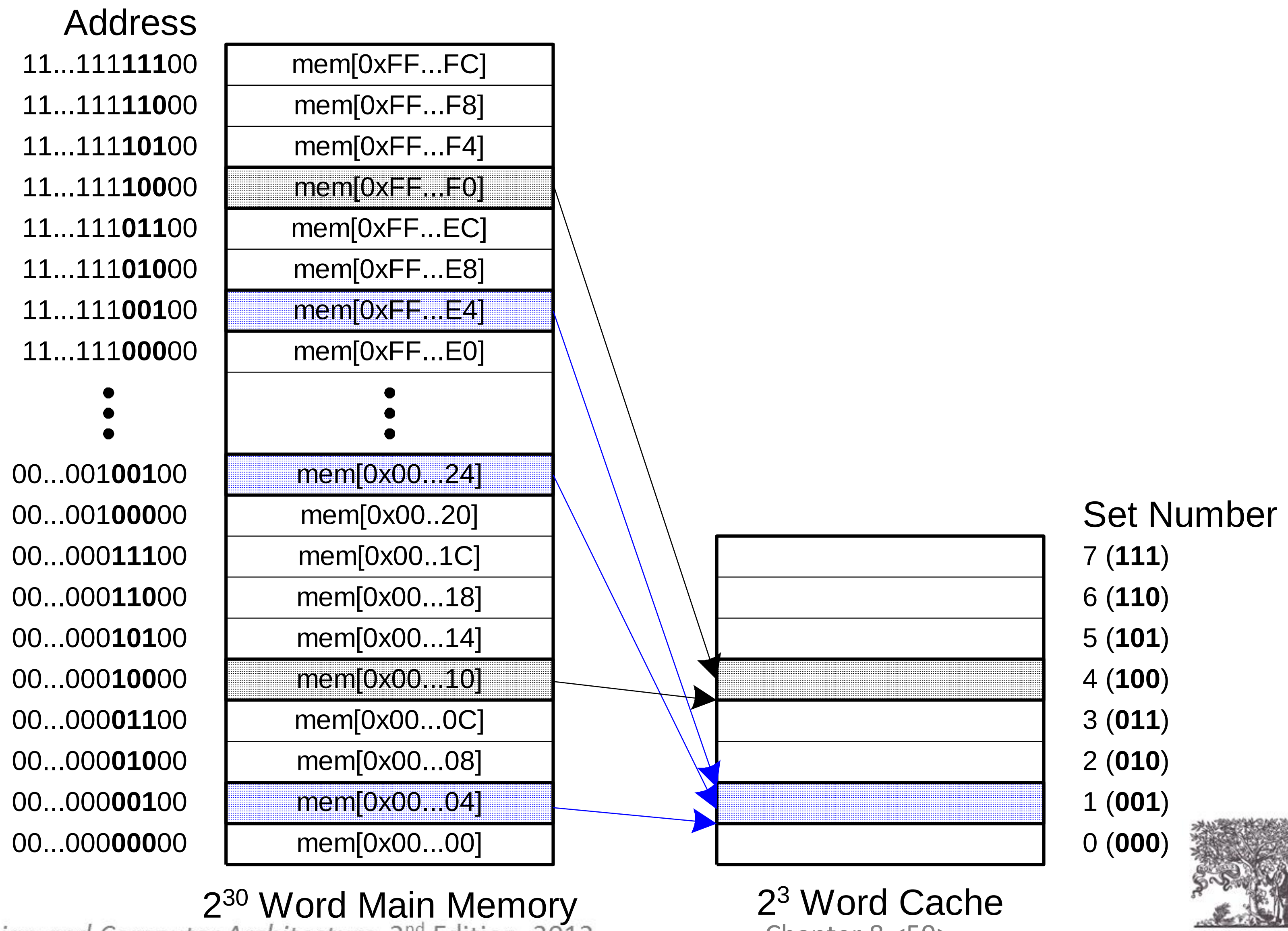
The rest of that cache block

- 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

40

Direct Mapped Cache



2^{30} Word Main Memory

2^3 Word Cache

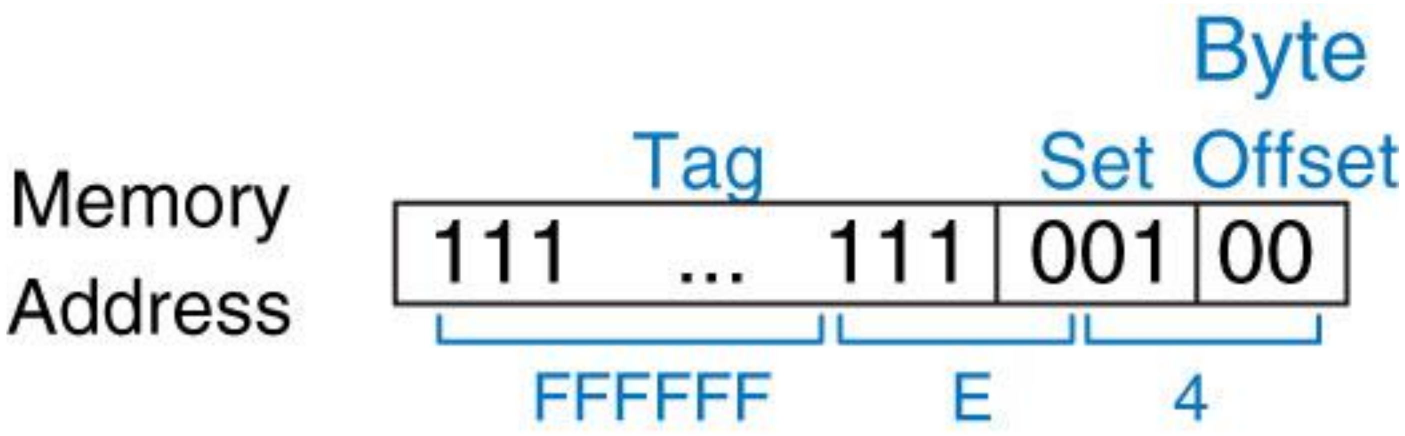
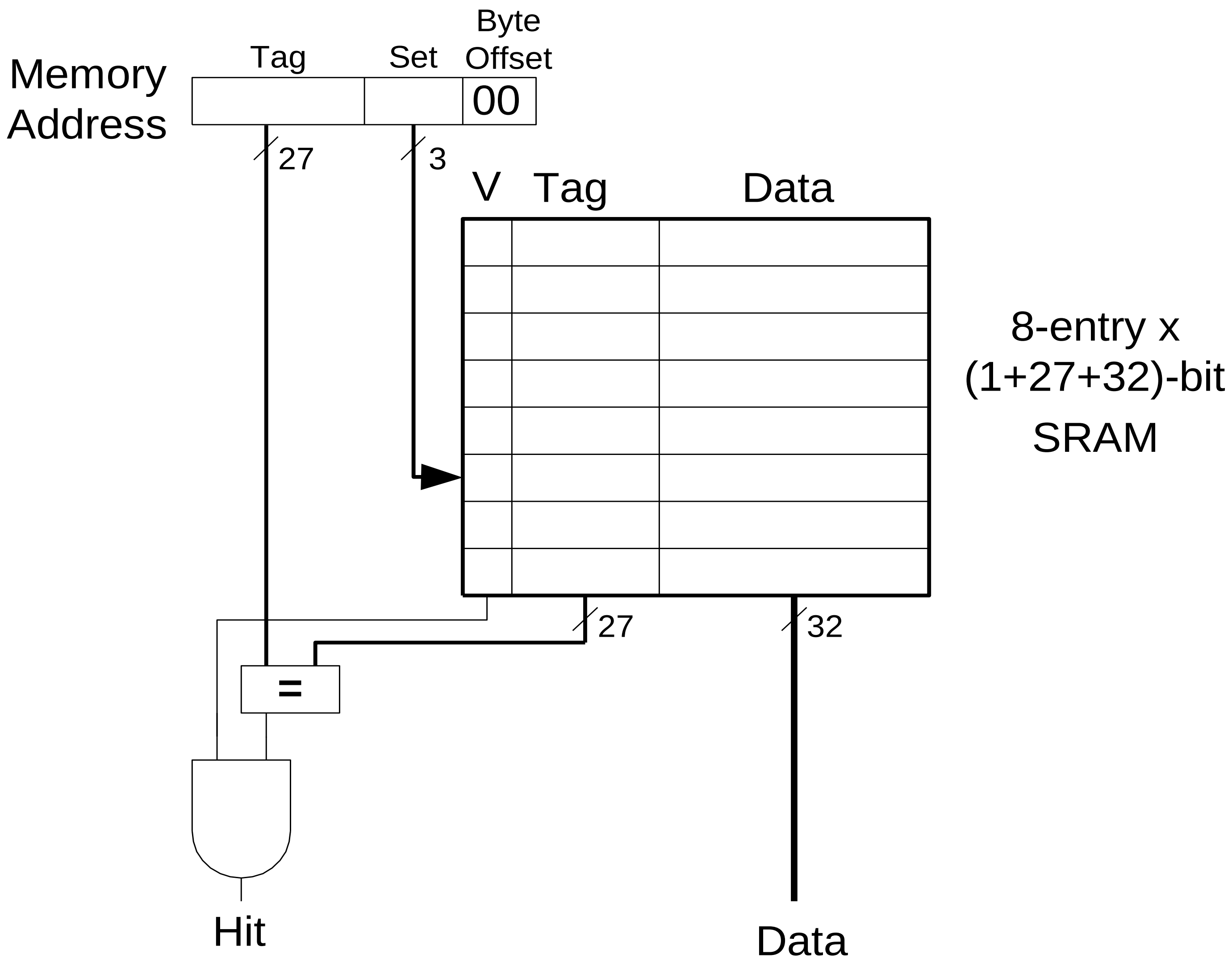
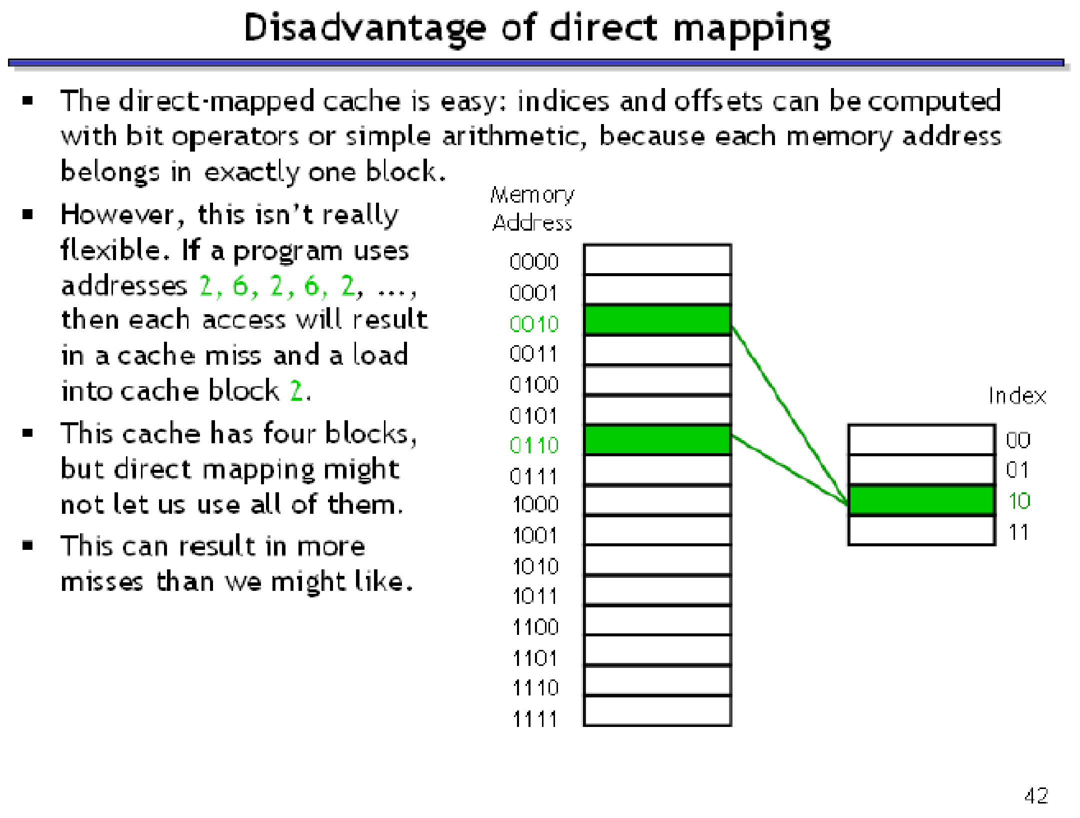


Figure 8.6 Cache fields for address 0xFFFFFFE4 when mapping to the cache in Figure 8.5

Direct Mapped Cache Hardware



Adapted from misc sources

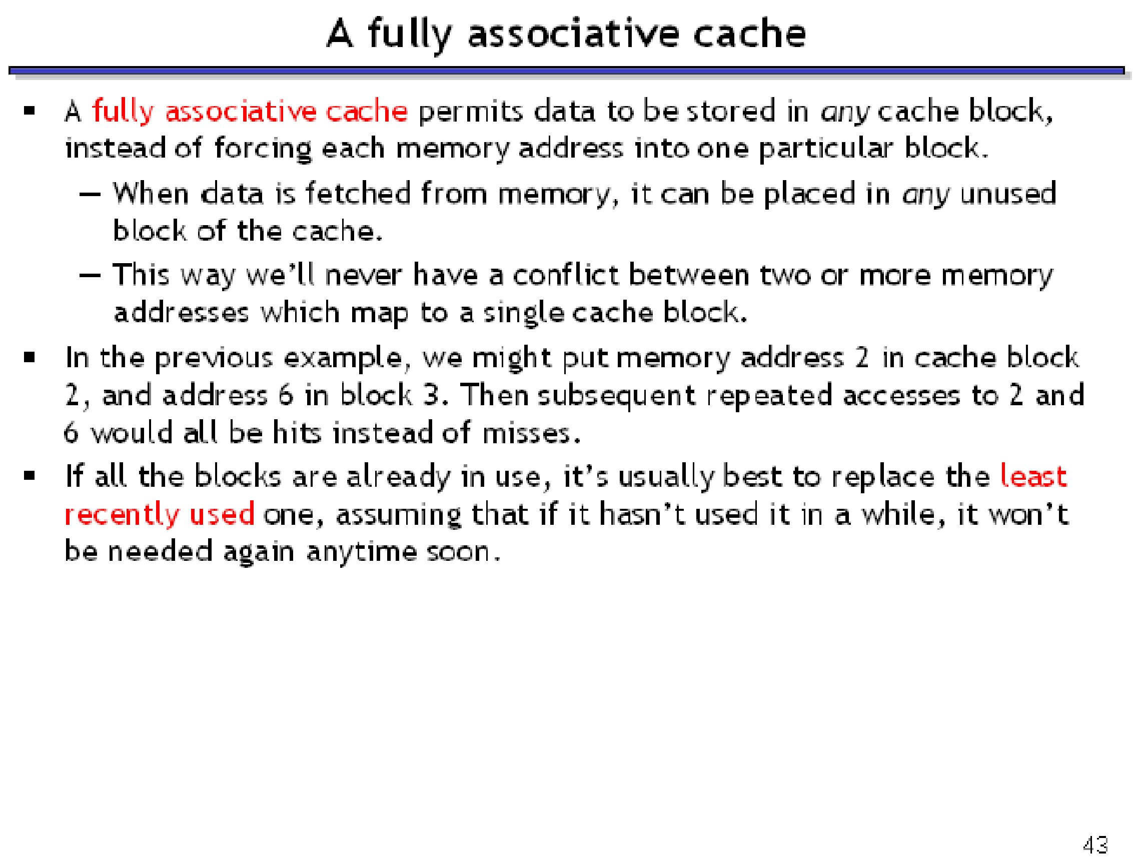


Disadvantage of direct mapping

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, 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.

42

Adapted from misc sources

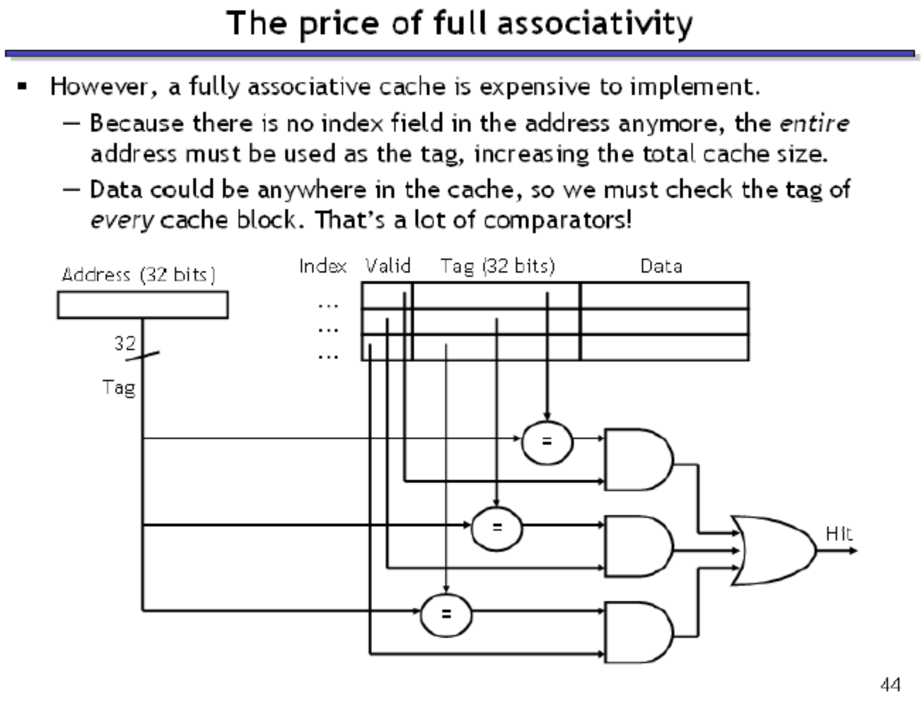


A fully associative cache

- 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 used it in a while, it won't be needed again anytime soon.

43

Adapted from misc sources

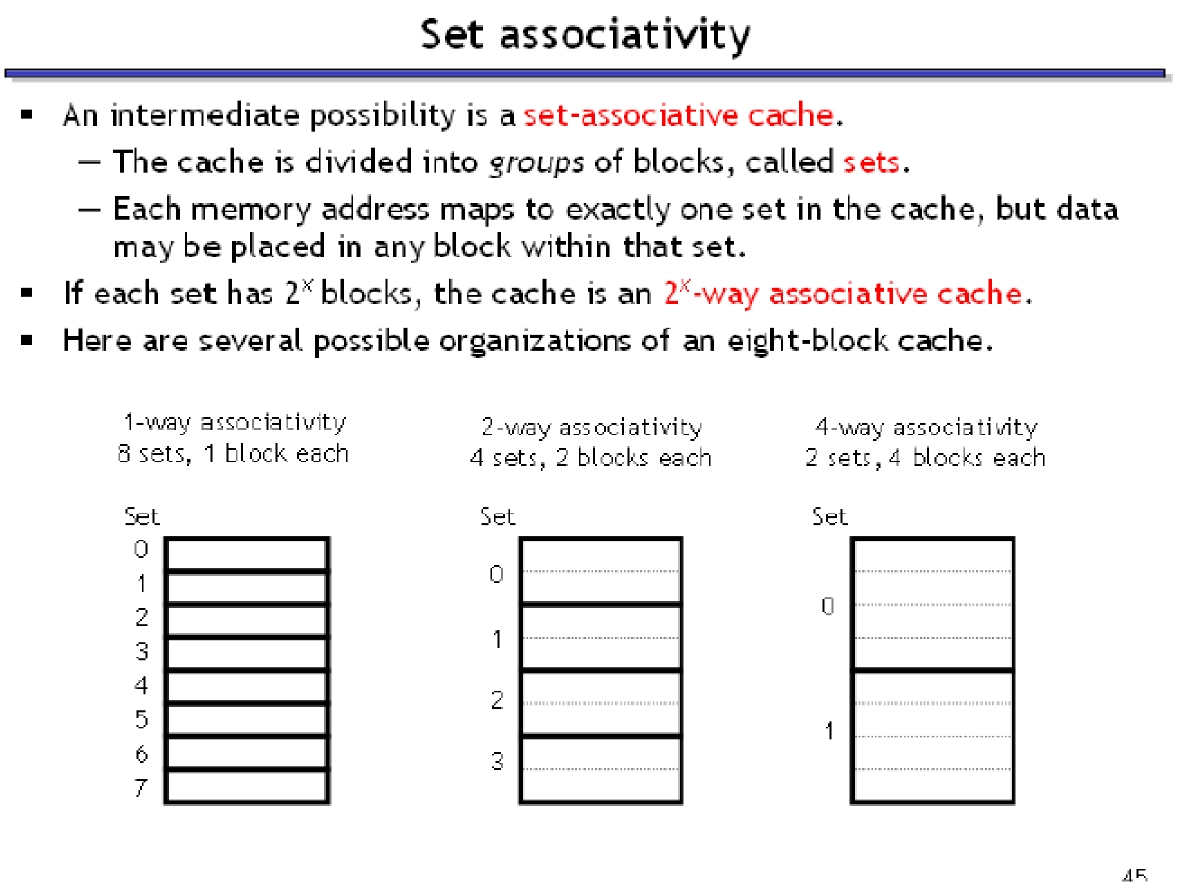


The price of full associativity

- However, a fully associative cache is expensive to implement.
  - 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!

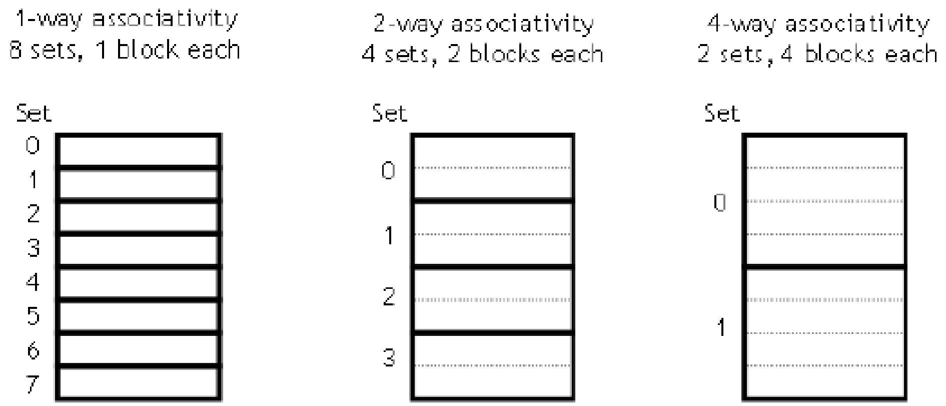
Address (32 bits)
32
Tag
Index
Valid
Tag (32 bits)
Data
...
=
Hit
44

Adapted from misc sources



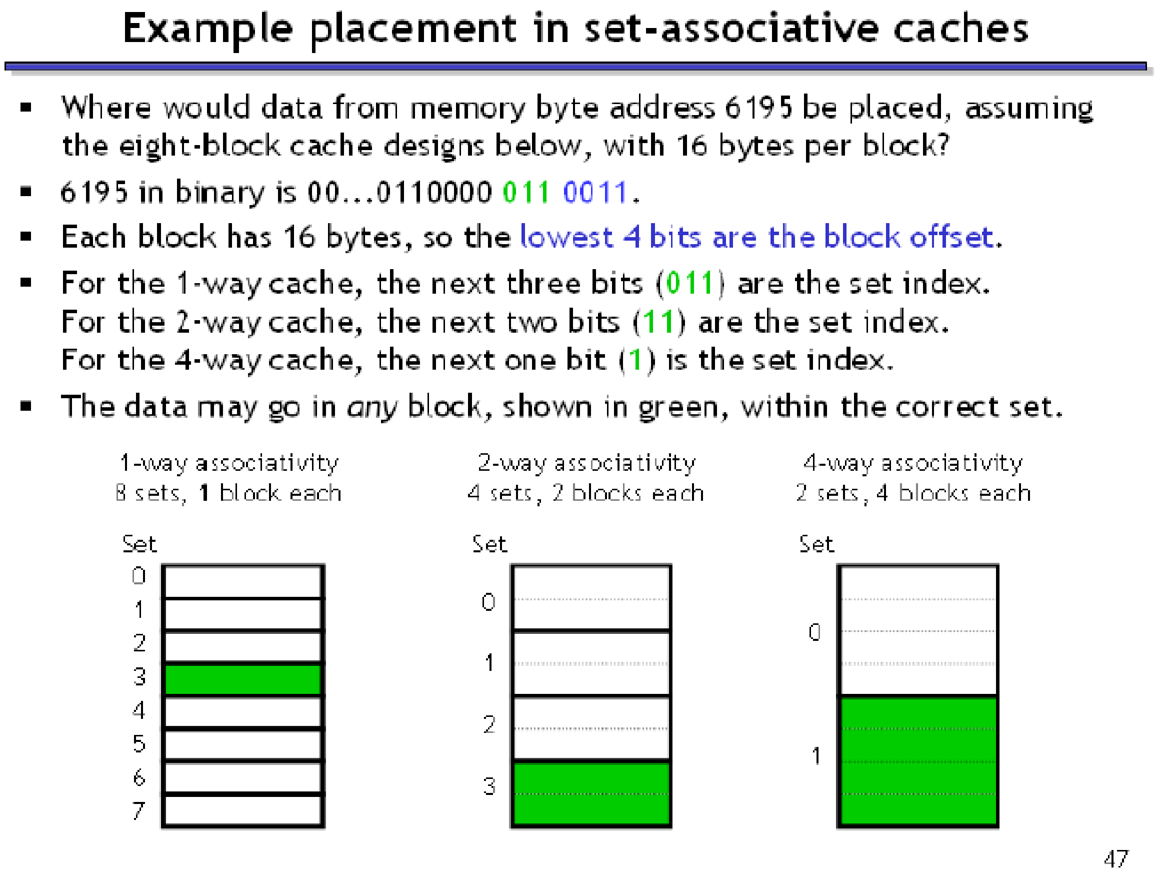
Set associativity

- 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, the cache is an 2^x-way associative cache.
- Here are several possible organizations of an eight-block cache.



45

Adapted from misc sources

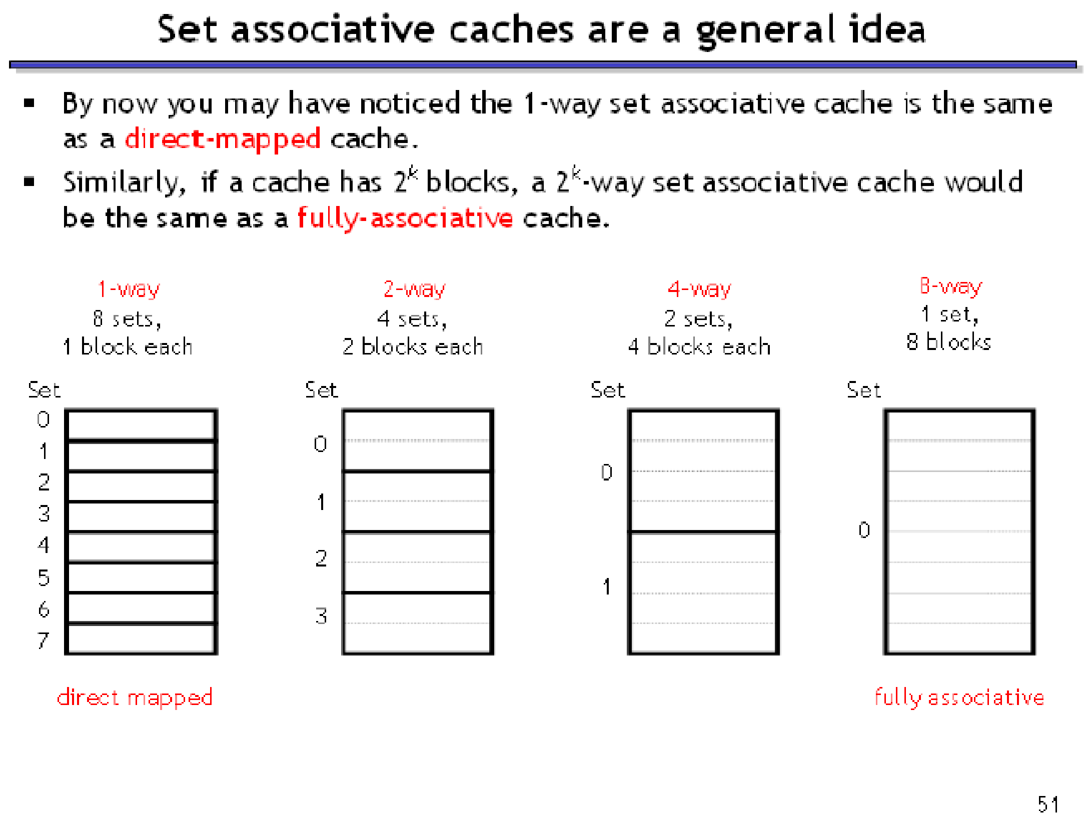


Example placement in set-associative caches

- 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 bytes, so the lowest 4 bits are the block offset.
- For the 1-way cache, the next three bits (011) are the set index.
  For the 2-way cache, the next two bits (11) are the set index.
  For the 4-way cache, the next one bit (1) is the set index.
- The data may go in *any* block, shown in green, within the correct set.

47

Adapted from misc sources



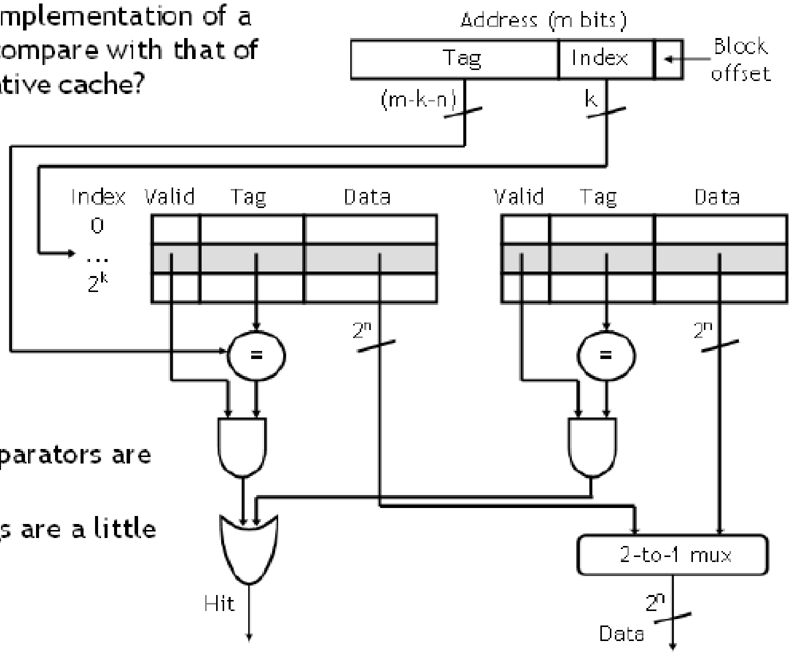
Set associative caches are a general idea

- By now you may have noticed the 1-way set associative cache is the same as a direct-mapped cache.
- Similarly, if a cache has 2^k blocks, a 2^k-way set associative cache would be the same as a fully-associative cache.

Adapted from misc sources

2-way set associative cache implementation

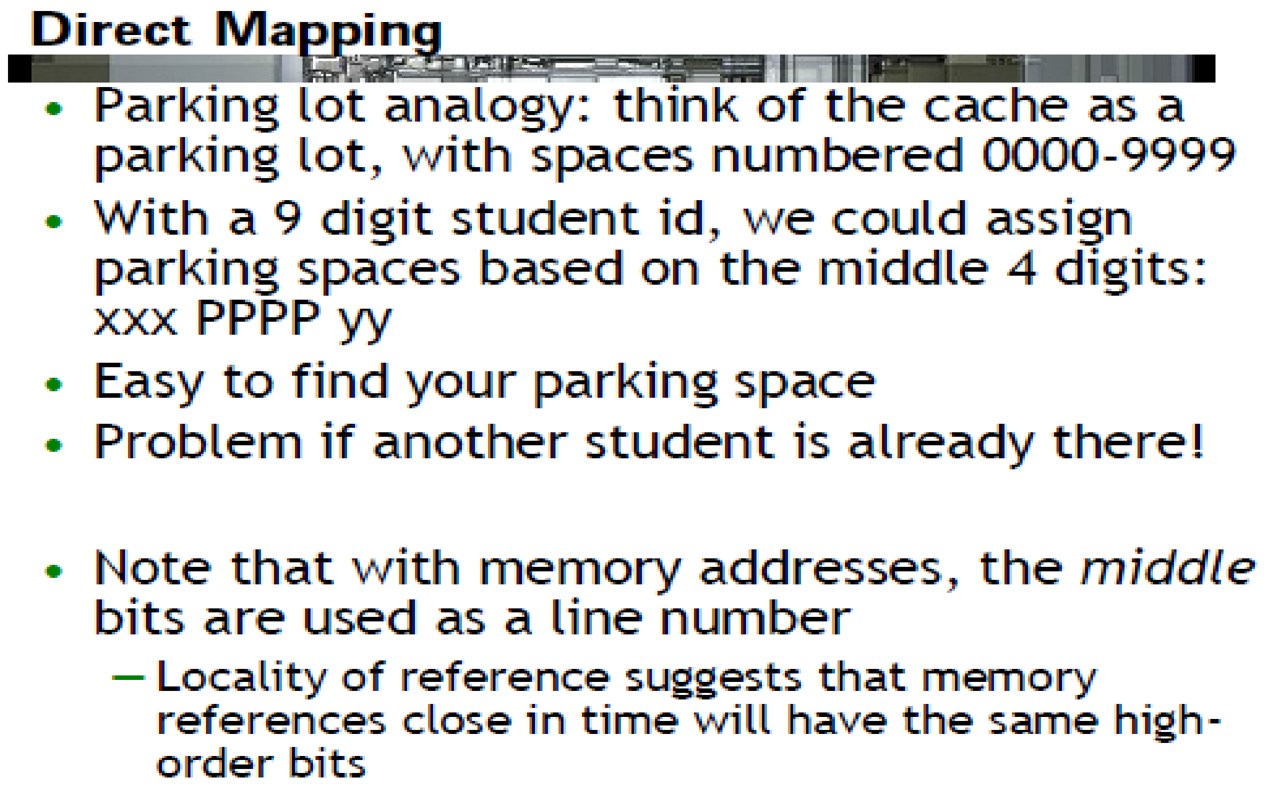
- How does an implementation of a 2-way cache compare with that of a fully-associative cache?



- Only two comparators are needed.
- The cache tags are a little shorter too.

52

Adapted from misc sources

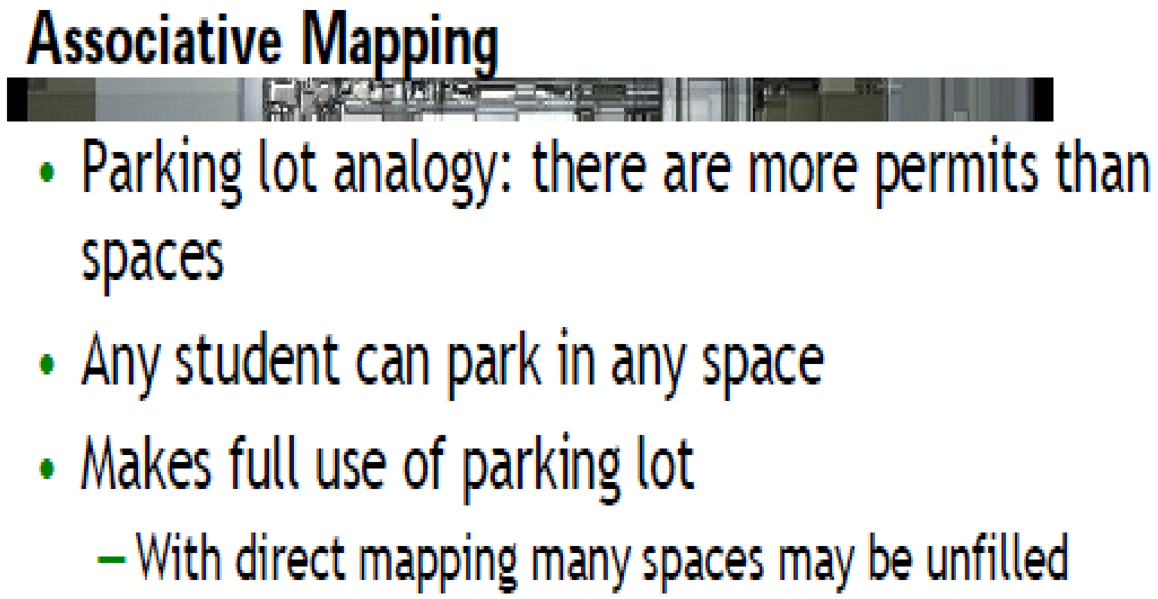


Direct Mapping

- Parking lot analogy: think of the cache as a parking lot, with spaces numbered 0000-9999
- With a 9 digit student id, we could assign parking spaces based on the middle 4 digits: xxx PPPP yy
- Easy to find your parking space
- Problem if another student is already there!

- Note that with memory addresses, the *middle* bits are used as a line number
  - Locality of reference suggests that memory references close in time will have the same high-order bits

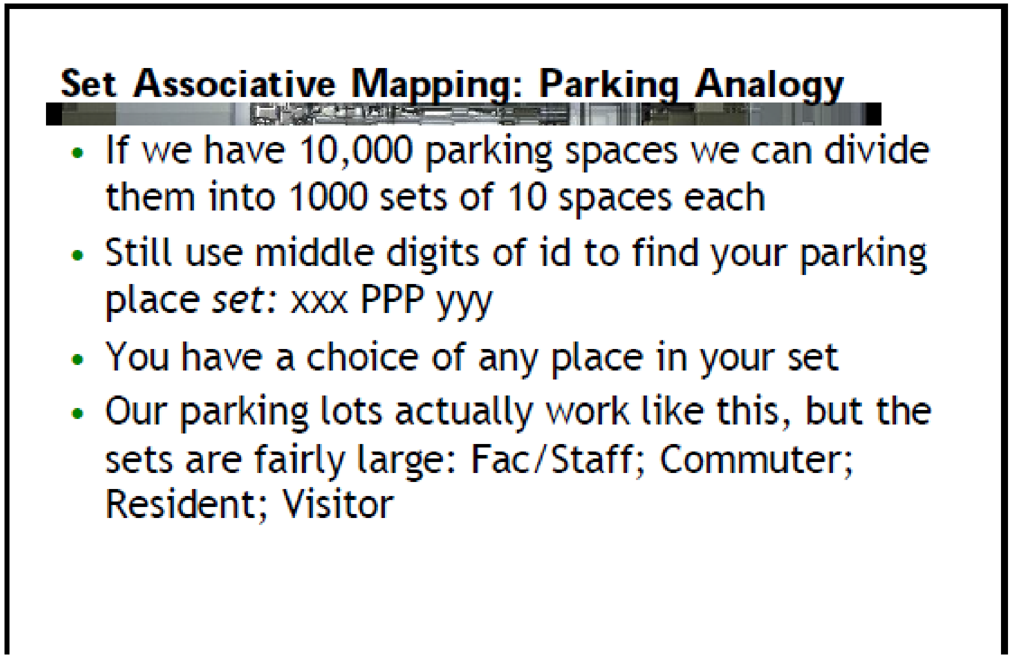
Adapted from misc sources



Associative Mapping

- Parking lot analogy: there are more permits than spaces
- Any student can park in any space
- Makes full use of parking lot
  - With direct mapping many spaces may be unfilled

Adapted from misc sources



Set Associative Mapping: Parking Analogy

- If we have 10,000 parking spaces we can divide them into 1000 sets of 10 spaces each
- Still use middle digits of id to find your parking place *set*: xxx PPP yyy
- You have a choice of any place in your set
- Our parking lots actually work like this, but the sets are fairly large: Fac/Staff; Commuter; Resident; Visitor

Adapted from misc sources

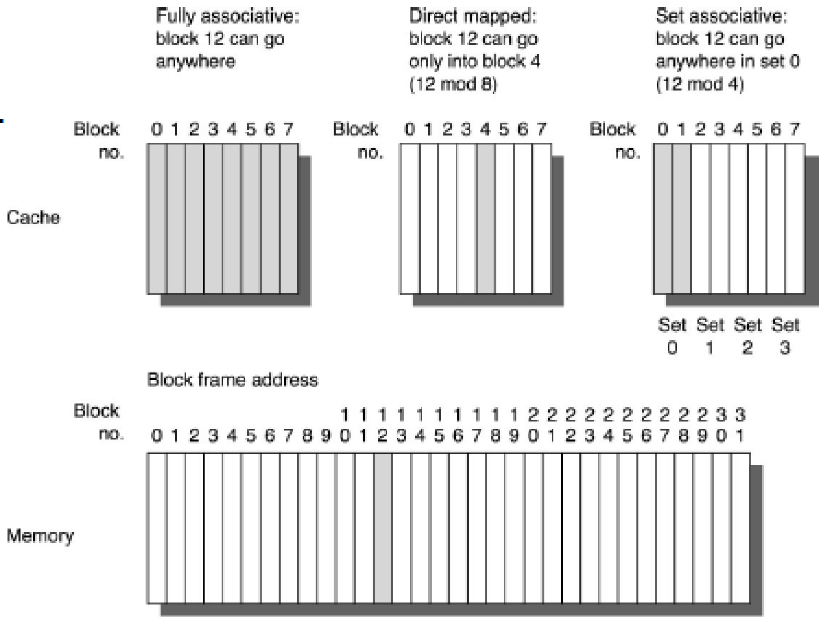
Cache Organization: Where can a block be placed in the cache?

Block 12 placed in 8 block cache:

- Fully associative, direct mapped, 2-way set associative
- S.A. Mapping = Block Number Modulo Number Sets

(associativity = degree of freedom in placing a particular block of memory)

(set = a collection of blocks cache blocks with the same cache index)



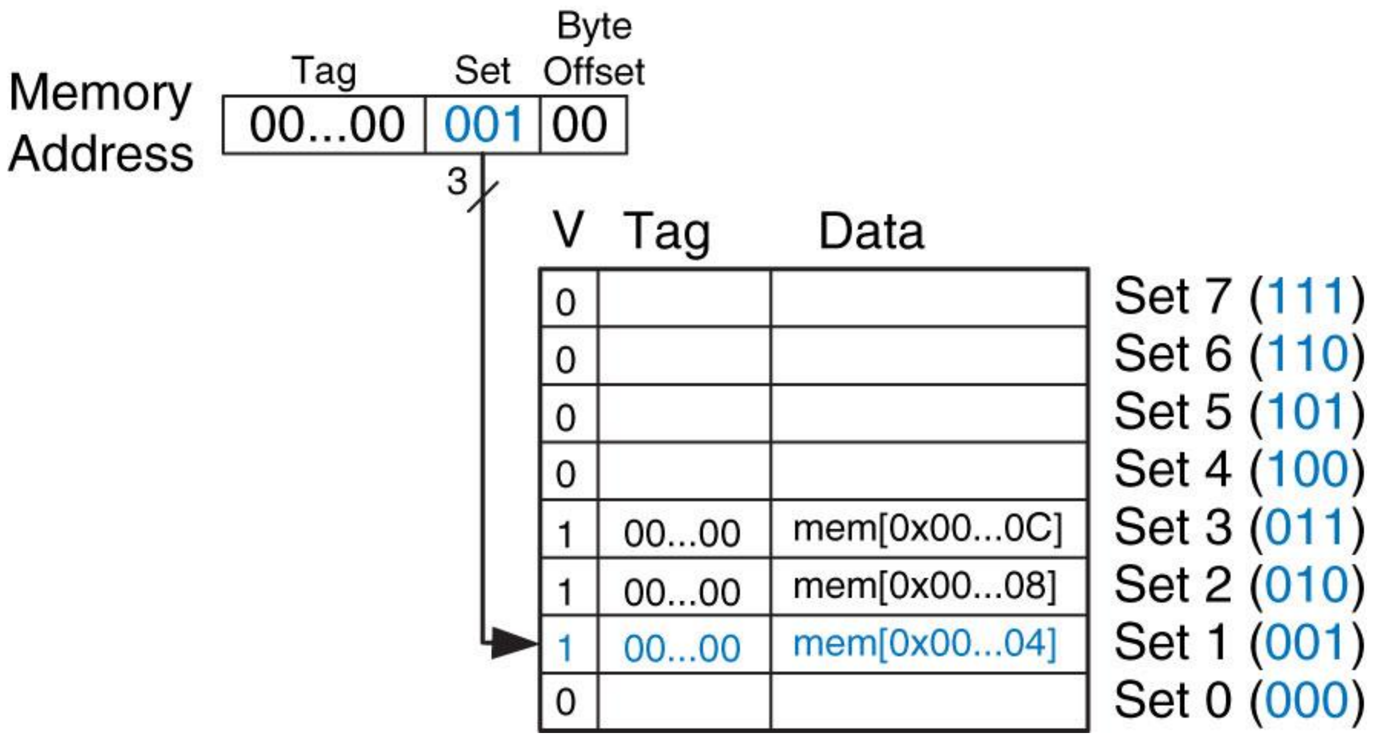
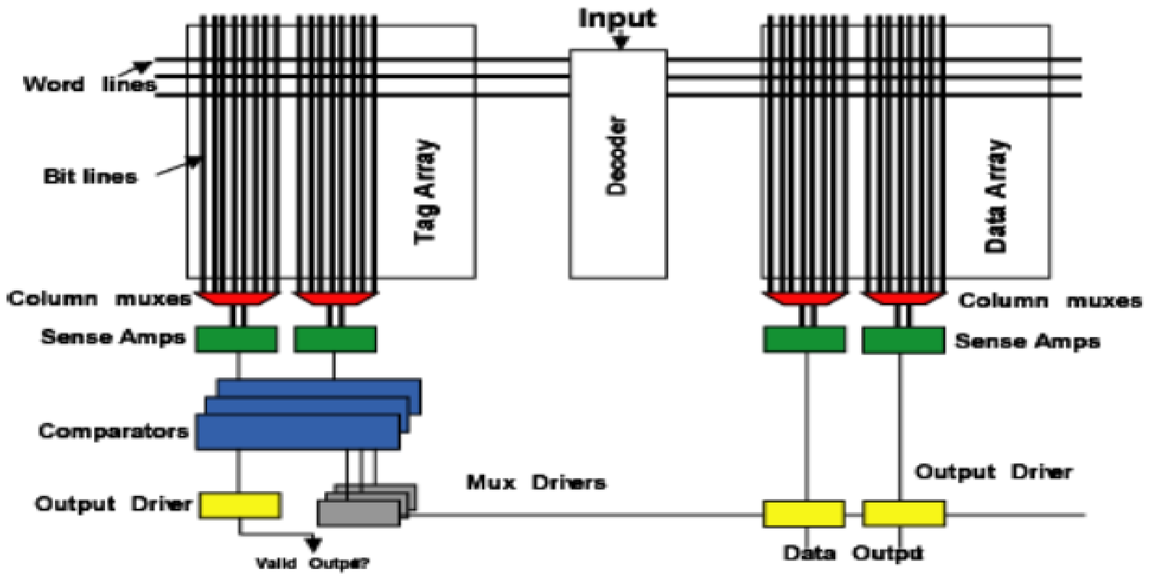


Figure 8.8 Direct mapped cache contents

Cache Organization



Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

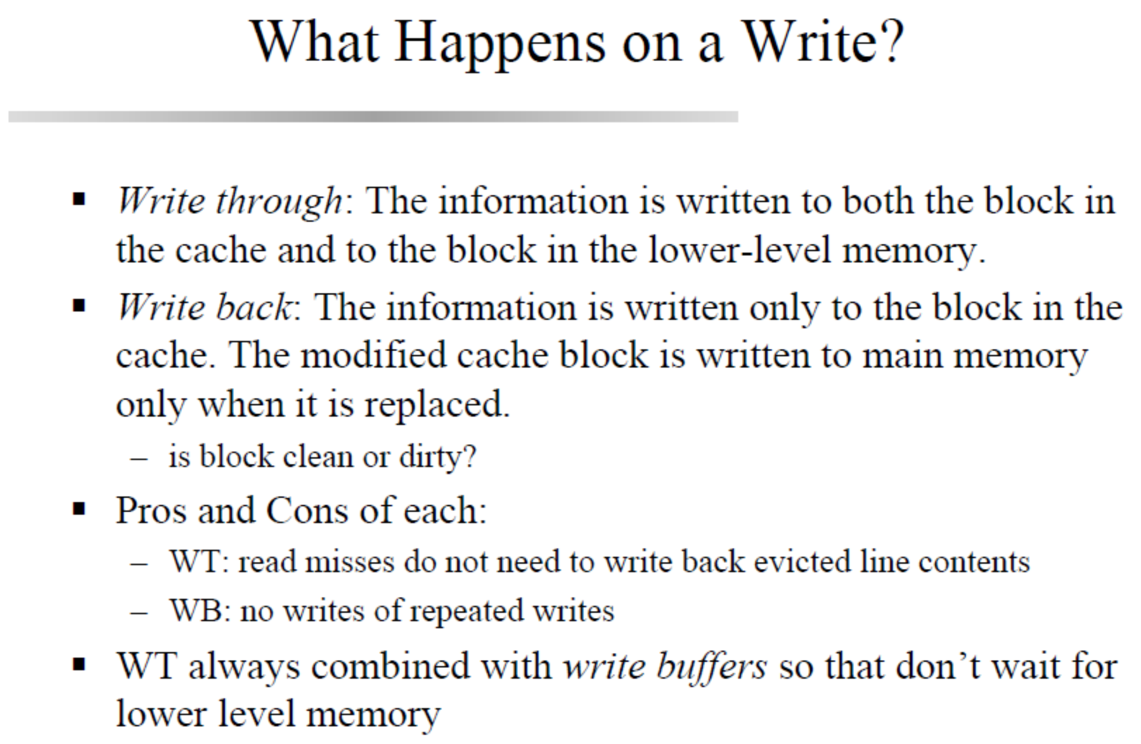
Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

Adapted from misc sources



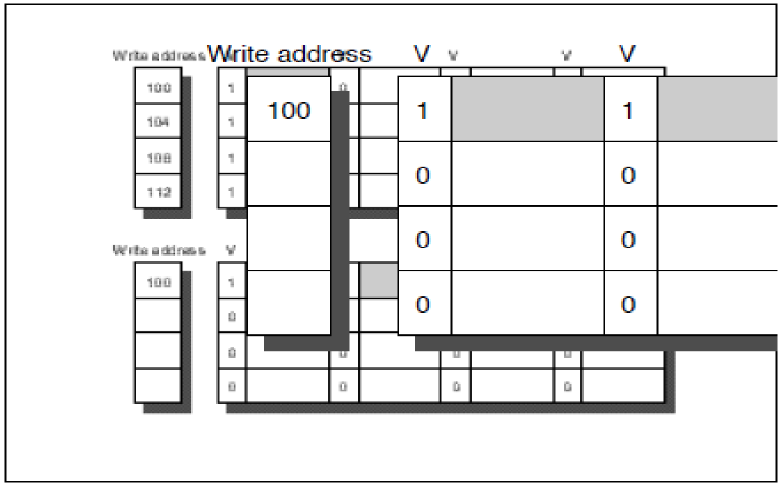
What Happens on a Write?

- *Write through*: The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each:
  - WT: read misses do not need to write back evicted line contents
  - WB: no writes of repeated writes
- WT always combined with *write buffers* so that don't wait for lower level memory

RUTGERS

What About Write Miss?

- *Write allocate*: The block is loaded into cache on a write miss
- *No-Write allocate*: The block is modified in the lower levels of memory but not in cache
- Write buffer allows merging of writes



Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

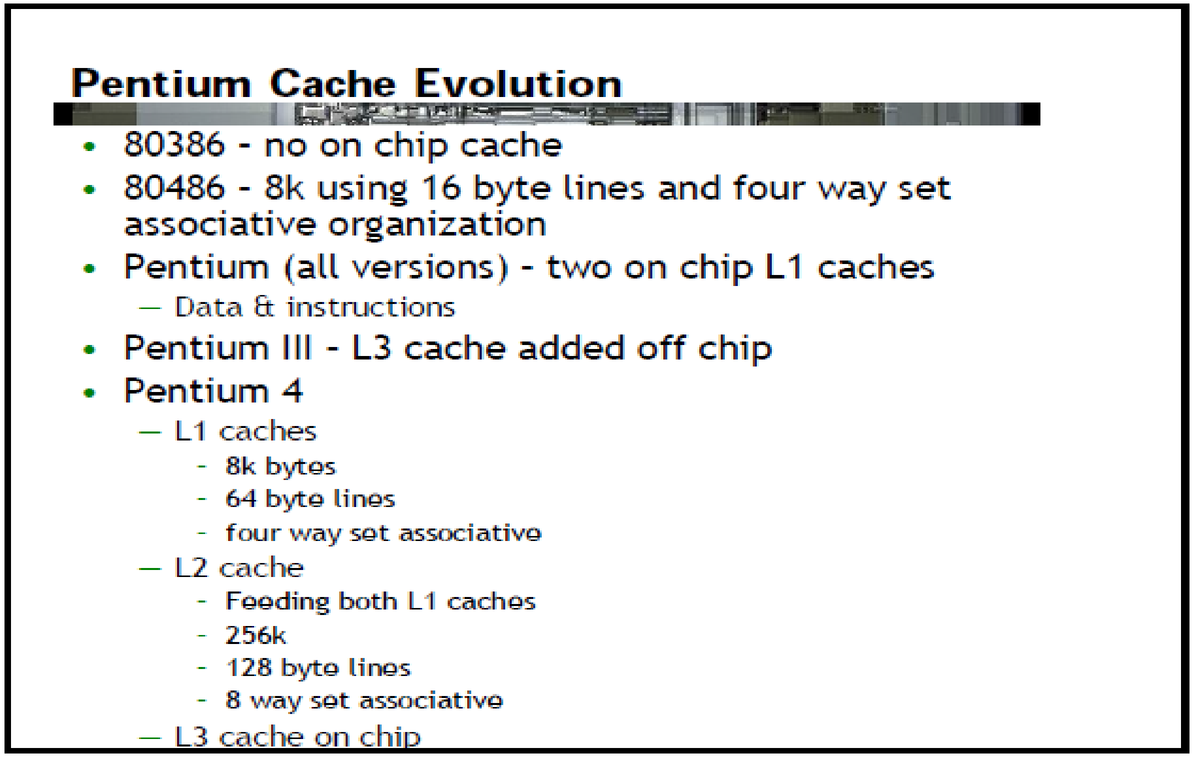
Types of Misses

- **Compulsory:** first time data accessed
- **Capacity:** cache too small to hold all data of interest
- **Conflict:** data of interest maps to same location in cache

Miss penalty: time it takes to retrieve a block from lower level of hierarchy

Multilevel Caches

- Larger caches have lower miss rates, longer access times
- Expand memory hierarchy to multiple levels of caches
- Level 1: small and fast (e.g. 16 KB, 1 cycle)
- Level 2: larger and slower (e.g. 256 KB, 2-6 cycles)
- Most modern PCs have L1, L2, and L3 cache



Pentium Cache Evolution

- 80386 - no on chip cache
- 80486 - 8k using 16 byte lines and four way set associative organization
- Pentium (all versions) - two on chip L1 caches
  - Data & instructions
- Pentium III - L3 cache added off chip
- Pentium 4
  - L1 caches
    - 8k bytes
    - 64 byte lines
    - four way set associative
  - L2 cache
    - Feeding both L1 caches
    - 256k
    - 128 byte lines
    - 8 way set associative
  - L3 cache on chip

Improving Cache Performance

- Reduce the time to hit in the cache
  - smaller cache
  - direct mapped cache
  - smaller blocks
  - for writes
    - Write-through
    - Write-back

- Reduce the miss rate
  - bigger cache
  - more flexible placement (increase associativity)
  - larger blocks (16 to 64 bytes typical)
  - Use “victim cache” – small buffer holding most recently discarded blocks

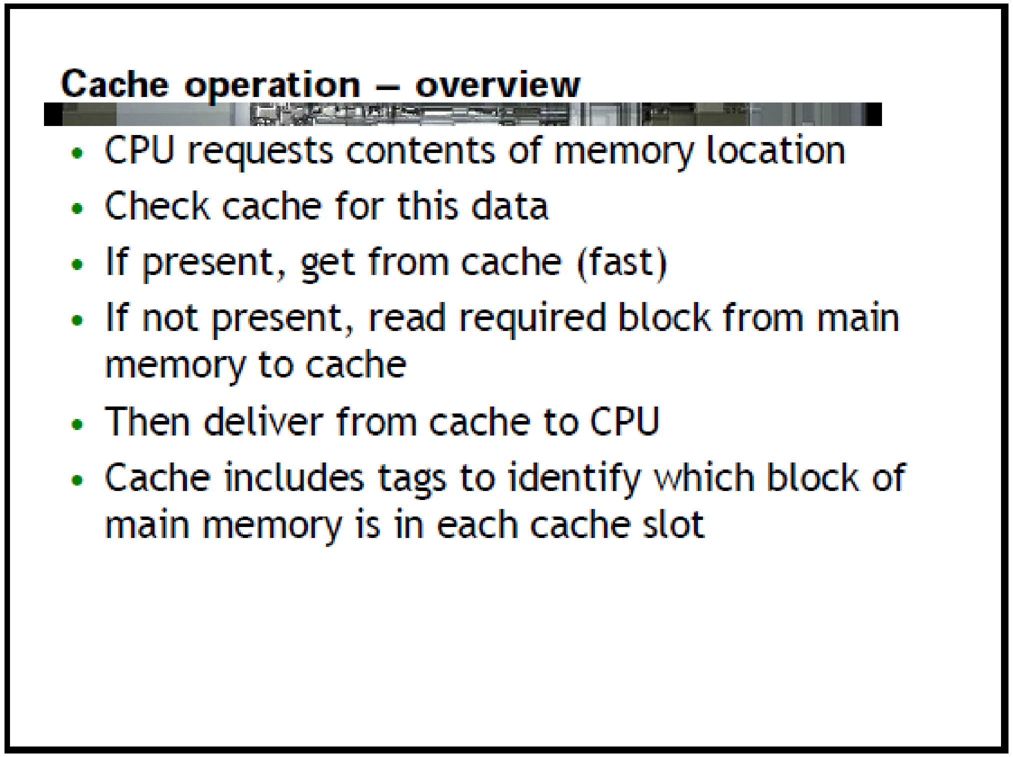
Improving Cache Performance

- Reduce the miss penalty
  - smaller blocks
  - use a write buffer to hold dirty blocks being replaced so don’t have to wait for the write to complete before reading
  - check write buffer (and/or victim cache) on read miss – may get lucky
  - for large blocks fetch critical word first
  - use multiple cache levels – L2 cache not tied to CPU clock rate
  - faster backing store/improved memory bandwidth
    - wider buses
    - memory interleaving, page mode DRAMs

Cache Summary

- **What data is held in the cache?**
  - Recently used data (temporal locality)
  - Nearby data (spatial locality)
- **How is data found?**
  - Set is determined by address of data
  - Word within block also determined by address
  - In associative caches, data could be in one of several ways
- **What data is replaced?**
  - Least-recently used way in the set

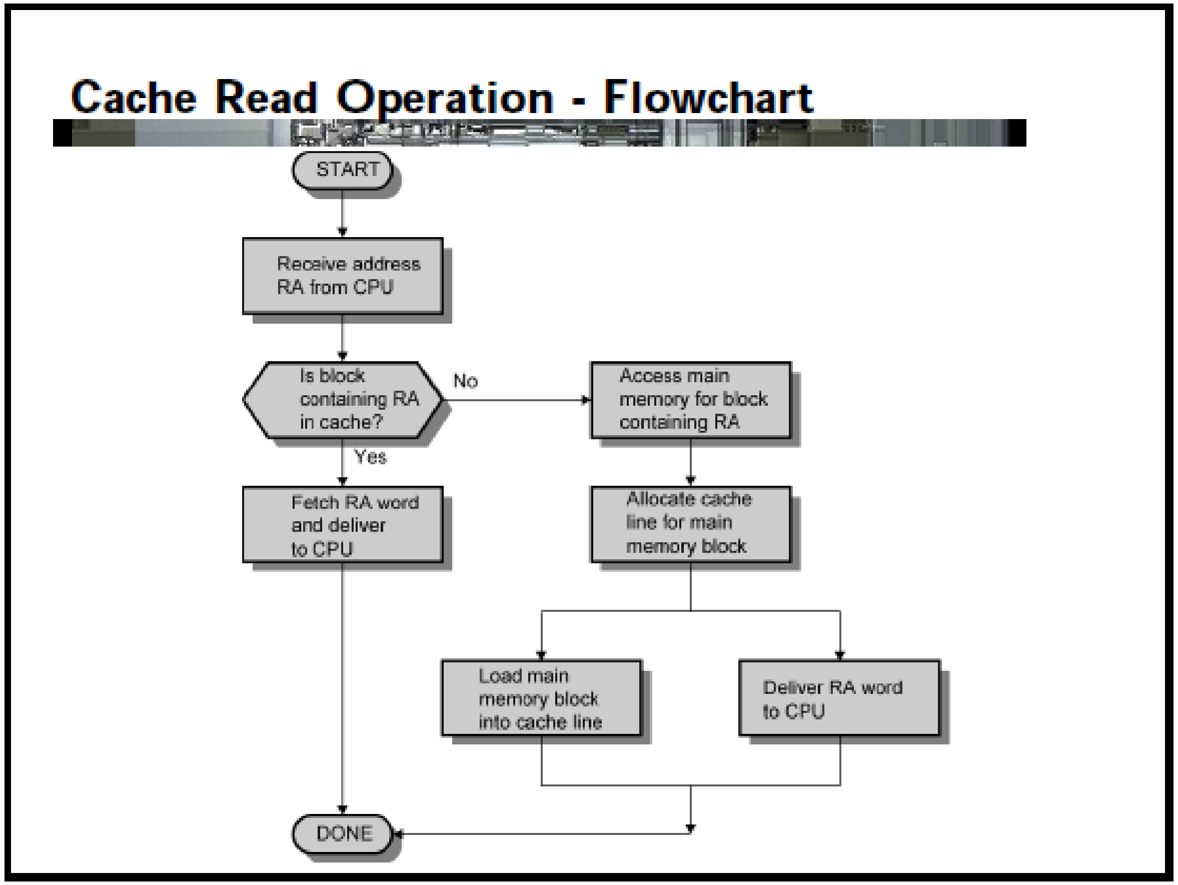
Adapted from misc sources



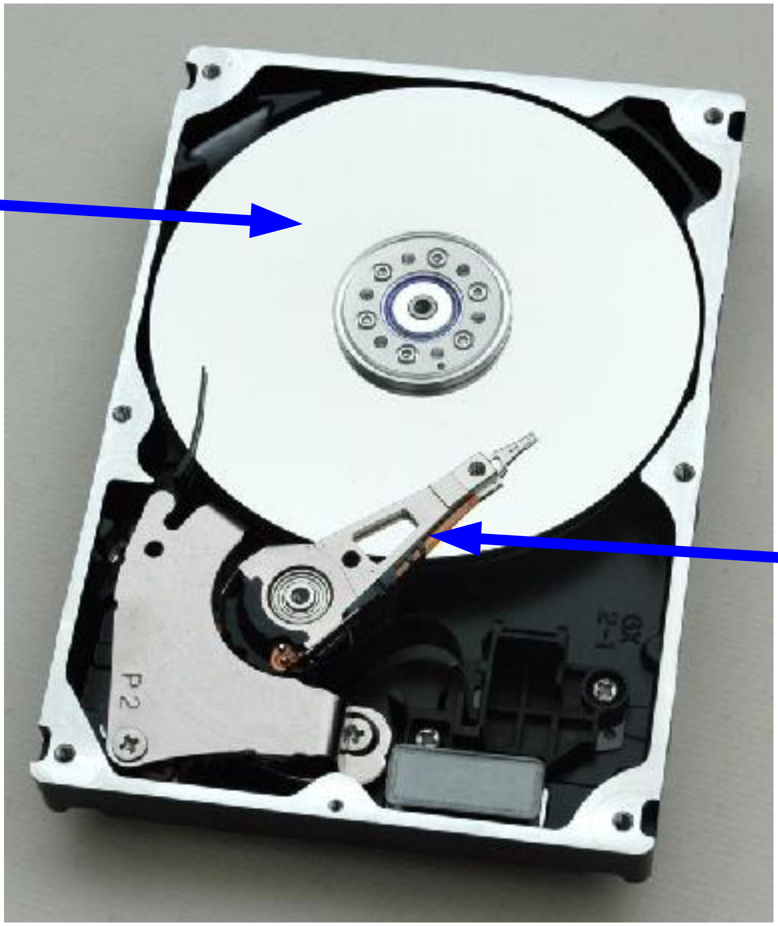
Cache operation – overview

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

Adapted from misc sources


Cache Read Operation - Flowchart
START
Receive address RA from CPU
Is block containing RA in cache?
No
Yes
Access main memory for block containing RA
Fetch RA word and deliver to CPU
Allocate cache line for main memory block
Load main memory block into cache line
Deliver RA word to CPU
DONE

Hard Disk



Takes milliseconds to *seek* correct location on disk

Backup Material