

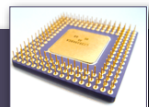


Memory Hierarchy & Cache Memory (Part 1)

(Supplemental)

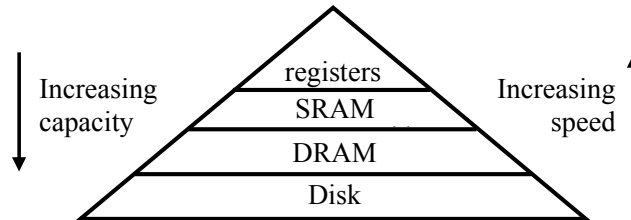
Based on Tarnoff, *Computer Organization and Design Fundamentals* (2007), Chapter 13

Homework/Administrivia



- ▶ **Read** §9.4.1–9.4.3 on two-dimensional arrays – *not covered in lecture*
- ▶ **Exam 2 Bonus** Friday, November 21, in class – *details in previous slide deck*
- ▶ **Read** the following from **Tarnoff-Ch13.pdf** (in Canvas under Files > Readings):
 - ▶ §13.1 – Characteristics of the Memory Hierarchy
 - ▶ §13.4 – Cache RAM
 - ▶ §13.5 – Registers

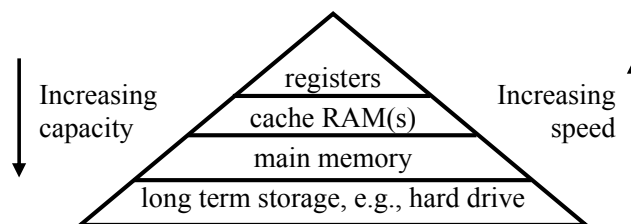
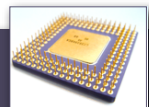
Memory Characteristics (Review)



Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10

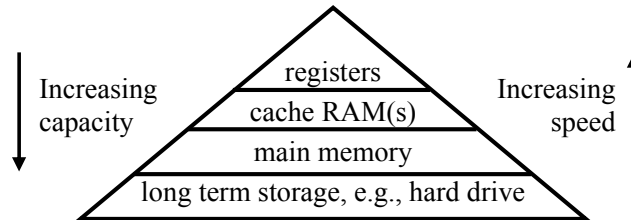
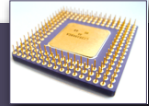
Table: Patterson & Hennessy, p. 378

Memory Hierarchy



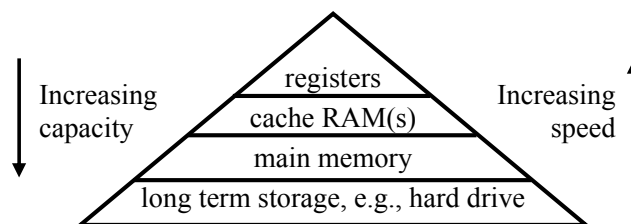
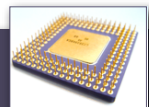
- ▶ A **memory hierarchy** is
 - ▶ an organization of storage devices
 - ▶ that takes advantage of the characteristics of different storage technologies
 - ▶ to improve the overall performance of a computer system

Hard Drive vs. Main Memory



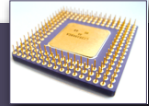
- ▶ Accessing a hard drive is slow
- ▶ Accessing main memory (DRAM) is significantly faster
- ▶ **∴ Load programs and data from the hard drive into main memory as needed**
 - ▶ Amount of RAM is usually much smaller than hard drive capacity – no problem
 - ▶ Programs load the data they need, as they need it – rarely do they need *every* bit of data on the drive!

Main Memory vs. Cache Memory



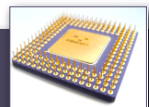
- ▶ Accessing a hard drive is slow
- ▶ Accessing main memory (DRAM) is significantly faster
- ▶ But accessing cache memory (SRAM) is significantly faster than accessing DRAM!
- ▶ **∴ Use cache memory to make accesses to main memory appear faster** (*how?...*)
 - ▶ Amount of cache memory is much smaller than the amount of main memory (*how to deal with this?...*)

Principle of Locality



- ▶ **Principle of Locality:**
 - ▶ Instructions that are executed within a short period of time tend to be close together in memory
 - ▶ Data that are accessed within a short period of time tend to be close together in memory
- ▶ Programs tend to exhibit both
 - ▶ **temporal locality:** if we access this instruction/data now, we are likely to need this same instruction/data again in the near future
 - ▶ **spatial locality:** if we access this instruction/data now, we are likely to need nearby instructions/data in the near future
- ▶ **Q.** Why is this true?

Cache: Concept

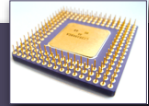


- ▶ Use the Principle of Locality to make main memory accesses appear faster:
- ▶ Place a small, fast SRAM (**cache**) between the processor and main memory



- ▶ Idea:
 - ▶ When the processor requests data, check if it's already in the cache
 - ▶ If not, copy it from main memory into the cache
 - ▶ *Also copy data in nearby memory locations*
- Temporal locality suggests we'll need this data again soon
- Spatial locality suggests we'll need surrounding data too

Cache: Hits and Misses

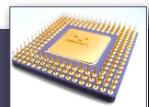


- ▶ Use the Principle of Locality to make main memory accesses appear faster:
- ▶ Place a small, fast SRAM (**cache**) between the processor and main memory

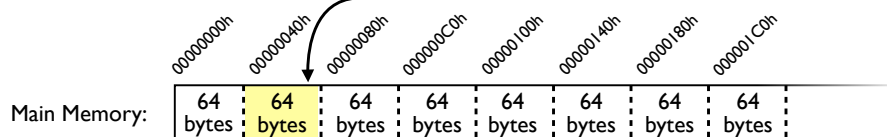


- ▶ Idea:
 - ▶ When the processor requests data, check if it's already in the cache
 - ▶ If the data is already in the cache, it is a **cache hit**
 - ▶ If the data is not already in the cache, it is a **cache miss**
 - ▶ Percentage of accesses that result in cache hits is the **hit rate** the **hit ratio**

Cache: Blocks & Cache Lines

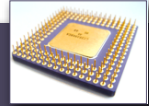


- ▶ Divide main memory into **blocks** of a fixed size
 - ▶ Example below: 64 bytes (64 = 40h)
 - ▶ Note: each 64-byte group starts at a memory address that is divisible by 64 = 40h
- ▶ Cache will store copies of some of these blocks (the ones actively being used at a point in time)
 - ▶ When talking about blocks inside the cache, they are often called **cache lines**
- ▶ If *any* byte in a block is accessed, copy the **entire block** into the cache
 - ▶ E.g., access memory address 00000078h \Rightarrow copy block starting at address 00000040h



On an earlier slide, this is what was meant by “copy data in nearby memory locations”

A Concrete Example



Memory1

Address: 0x00401010

```

0x00401010  b0 00 fe c0 3c ff 72 fa  °.pÀÿrú
0x00401018  6a 00 e8 0b 00 00 00 90  j.è.....
0x00401020  90 90 90 90 cc cc cc cc  ....iiii
  
```

main.asm

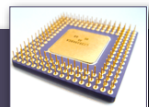
```

.code
main PROC
    mov al, 0
a:    inc al
      cmp al, 255
      jb a
      push 0
      call ExitProcess
      nop
      nop
      nop
      nop
      nop
      main ENDP
  
```

Recall: `nop` is encoded as 90h

- Suppose 64-byte cache lines
 - Note: `main` starts at address 00401010h, which is divisible by 64
- Q.** Suppose data are copied into the cache when the processor fetches the first instruction. What other instructions will also be copied into the cache?
- Q.** Suppose data are copied into the cache when the processor fetches the `cmp` instruction. What other instructions will also be copied into the cache?

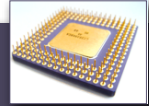
Identifying Blocks (1)



	0 = 0000 0000
Block	1 = 0000 0001
	2 = 0000 0010
	3 = 0000 0011
	4 = 0000 0100
Block	5 = 0000 0101
	6 = 0000 0110
	7 = 0000 0111
	8 = 0000 1000
Block	9 = 0000 1001
	10 = 0000 1010
	11 = 0000 1011
	12 = 0000 1100

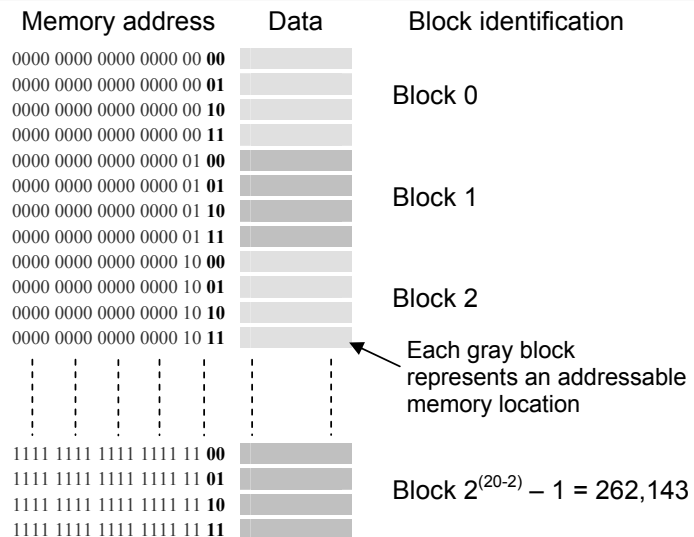
- Suppose we have 4-byte blocks
- Each block starts at a memory address divisible by 4
- The upper bits of all memory addresses within a block are the same!**
 - Only the lowest two bits are different (since $4 = 2^2$)
- Idea:
 - Use the upper bits (that are the same within a block) to number that block
 - This is called the **block number**
- Recall: You can divide by 2^k by right-shifting
- Splitting the bits of a number into two groups gives the quotient and remainder when the number is divided by a power of 2
 - So the block number is the memory address divided by the block size

Identifying Blocks (2)

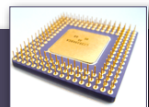


► Suppose:

- 20-bit memory addresses (x86 real-address mode)
- 4-byte cache blocks
- $4 = 2^2$ bytes per block
- **Q.** Given a memory address, what *bitwise operations* would you use to find
 - the block number?
 - the position of that byte within the block?



Identifying Blocks (3)



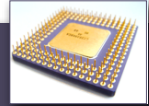
- **Q.** Suppose you have a 1 GB (2^{30} -byte) address space divided into 8-byte blocks.

- *Note:* memory addresses are 30 bits
- (a) How many blocks are in the address space? (*Answer: 2 to what power?*)
- (b) How would a memory address be divided to identify a block and offset?

- **A.** (a) $2^{30} \div 8 = 2^{30} \div 2^3 = 2^{27} = 134,217,728$

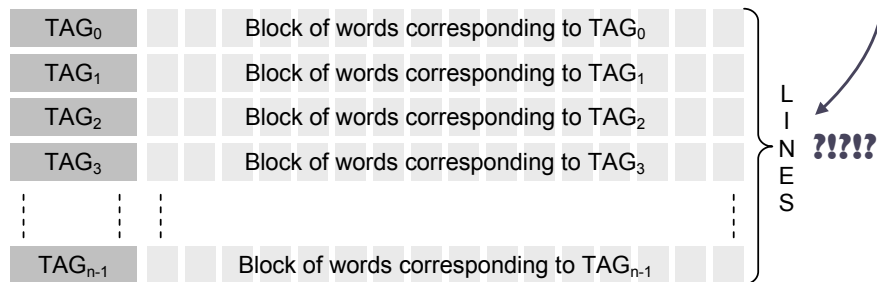
- (b) Memory address \rightarrow
- | | | | | | | | | |
|------------------------|----------|----------|---------|-------|-------|-------------------------|-------|-------|
| a_{29} | a_{28} | a_{27} | \dots | a_4 | a_3 | a_2 | a_1 | a_0 |
| └──────────────────┘ | | | | | | └───┘ | | |
| Bits identifying block | | | | | | Bits identifying offset | | |

Cache Entries

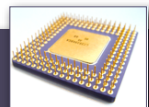


- ▶ A cache consists of several entries
 - ▶ Number of entries is usually a power of 2
- ▶ Each entry contains (at least)
 - ▶ the data from one block of memory
 - ▶ a **tag** that identifies which block of memory is currently stored there

We defined a **cache line** to be synonymous with a *block*. The assigned reading defines a **line** to be an entire cache entry (tag+block), which is unusual.

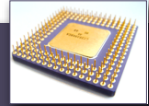


Cache Questions



- ▶ Remember: cache is significantly smaller than main memory
- ▶ Four questions in designing a cache system:
 - ▶ **block placement/mapping function** – where can a block of memory go in the cache?
 - ▶ **block identification** – how to determine if a block is in the cache, and where it is?
 - ▶ **replacement algorithm** – when the cache is full, what do you remove?
 - ▶ **write policy** – when the processor needs to *write* to memory, what happens?

Block Placement Policies



▶ Direct-mapped

- ▶ For any block of memory, there's only one place where it can be stored
- ▶ Use $(\text{block number}) \bmod (\text{number of blocks})$ to determine which slot to store it in

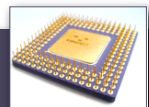
▶ Fully associative

- ▶ Any block of memory can be stored in any slot

▶ Set-associative ← Most commonly used

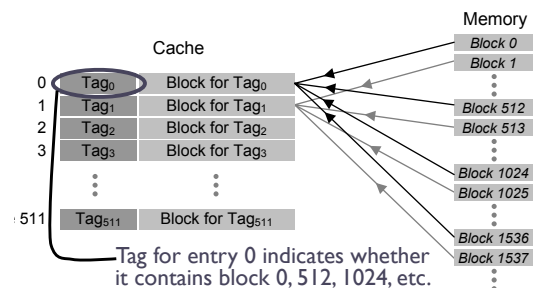
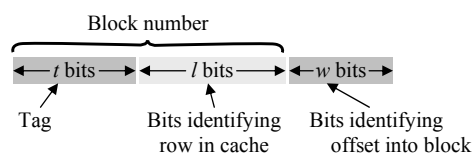
- ▶ Cache entries are grouped together into *sets*
- ▶ The block number determines which *set* the block will be stored in
- ▶ But it can be stored in any slot within that set

Block Placement: Direct Mapping (1)



- ▶ In a **direct-mapped** cache, the block number determines where the block will be stored in the cache
 - ▶ Number rows in the cache from 0; store in $(\text{block number}) \bmod (\text{number of rows})$
 - ▶ If there are 2^l blocks, the lowest l bits of the block number give the row number to store in
 - ▶ $\text{tag} = (\text{block number}) / (\text{number of rows})$
 - ▶ Use the upper bits as a **tag** to identify which block is stored in this row

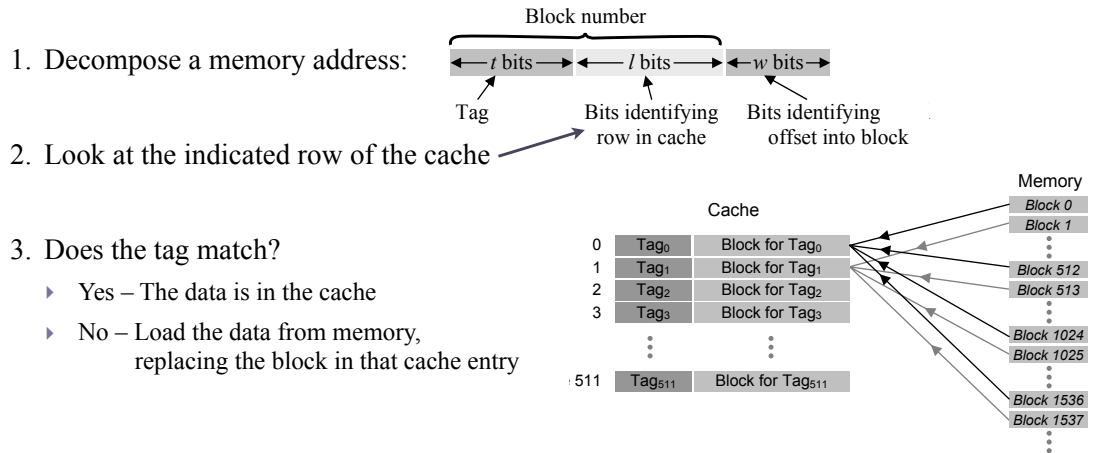
- ▶ Decompose a memory address into:



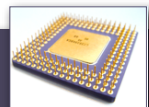
Block Placement: Direct Mapping (2)



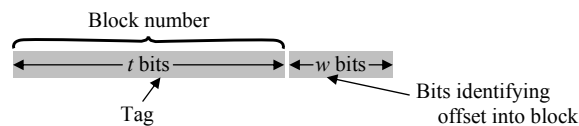
- ▶ **Block Identification** – When a memory read occurs, determine if the corresponding block of memory is in the cache:



Block Placement: Fully Associative

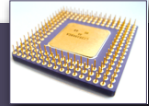


- ▶ In a **fully associative** cache, any block can be stored in any row in the cache
- ▶ Use the entire block number for the tag



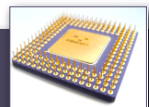
- ▶ To check if a block is in the cache, must check every row – it could be anywhere
 - ▶ Wouldn't this be slow?
 - ▶ No – Hardware can be designed to search all cache entries simultaneously (in parallel)
 - ▶ But this can be expensive

Replacement Policy



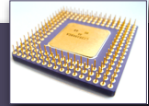
- ▶ In a fully associative or set associative cache, a block could be placed in several possible places
- ▶ If the cache is full, one block must be removed (**evicted**) from the cache – which one?
- ▶ Choose a **replacement algorithm**:
 - ▶ **Least Recently Used (LRU)** ← *Most commonly used*
 - ▶ Replace the block that hasn't been read by the processor in the longest period of time
 - ▶ **First In First Out (FIFO)**
 - ▶ Replace the block that has been in the cache the longest
 - ▶ **Least Frequently Used (LFU)**
 - ▶ Replace the block with the fewest hits since being loaded into the cache
 - ▶ **Random**
 - ▶ Randomly select a block to replace

Cache Questions



- ▶ Remember: cache is significantly smaller than main memory
- ▶ Four questions in designing a cache system:
 - ▶ **block placement/mapping function** – where can a block of memory go in the cache?
 - ▶ **block identification** – how to determine if a block is in the cache, and where it is?
 - ▶ **replacement algorithm** – when the cache is full, what do you remove?
 - ▶ **write policy** – when the processor needs to *write* to memory, what happens?

Cache Write Policy

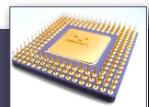


- ▶ Problem: There are now *two* copies of data – one in the cache, one in main memory
- ▶ What happens when the processor wants to *write* to memory?

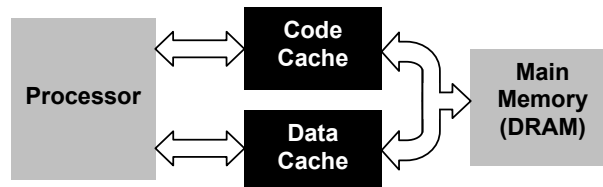


- ▶ Choose a **cache write policy**:
 - ▶ **Write-through**: when the processor writes, the cache updates itself *and* main memory
 - ▶ **Write-back**: only update the cache; update main memory when the block is removed from the cache

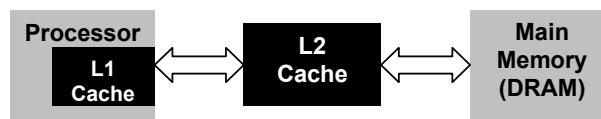
More Advanced Cache Designs



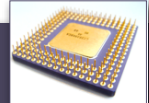
- ▶ If instructions (code) and data are both stored in the same cache, it is a **unified** cache
- ▶ Some processors use **split caches** – separate caches for code (instructions) and data
 - ▶ Instruction cache and data cache often called **I-cache** and **D-cache**, respectively



- ▶ Modern processors use two or three levels of cache memory (**multilevel caches**)



Example: Intel "Sandy Bridge"



- ▶ Intel's Sandy Bridge microarchitecture – released 2011, used in 2nd generation Core i3, i5, i7
- ▶ L1 Cache: 64 KB per core, split, 8-way set associative, write-back, 64-byte cache lines
- ▶ L2 Cache: 256 KB per core, unified, 8-way set associative, write-back, 64-byte cache lines
- ▶ L3 Cache: size varies (1–20 MB), 64-byte cache lines

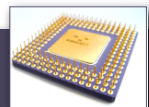
From the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*:

Table 2-10. Cache Parameters

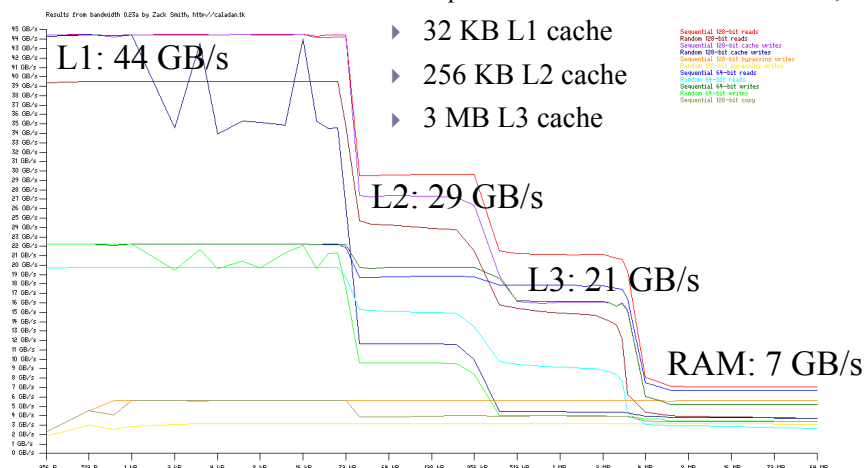
Level	Capacity	Associativity (ways)	Line Size (bytes)	Write Update Policy
L1 Data	32 KB	8	64	Writeback
Instruction	32 KB	8	N/A	N/A
L2 (Unified)	256 KB	8	64	Writeback
Third Level (LLC)	Varies, query CPUID leaf 4	Varies with cache size	64	Writeback

Source: <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>

Does cache memory matter?



- ▶ Example Benchmark on Intel Core i5-520M, Mac OS X 10.6



Source: <http://smith.co/bandwidth.html#graphs>